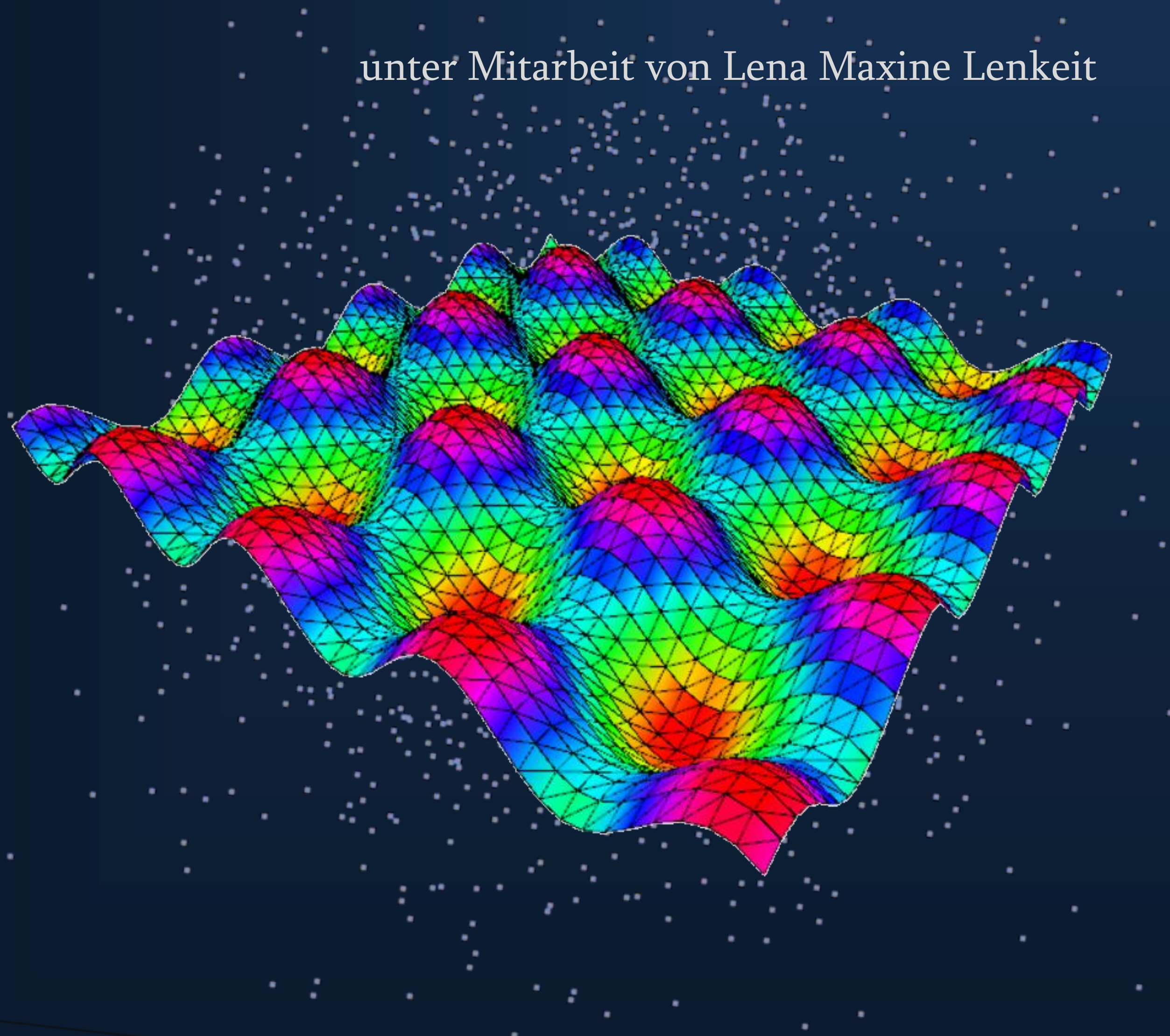
Walter Stein

Programmieren lernen für den Physikunterricht mit Processing

Ein Lern- und Aufgabenbuch für Physiklehrer und Physikschüler der gymnasialen Oberstufe

unter Mitarbeit von Lena Maxine Lenkeit



Programmieren lernen für den Physikunterricht mit Processing

Ein Lern- und Aufgabenbuch für Physiklehrer und Physikschüler der gymnasialen Oberstufe

Walter Stein

unter Mitarbeit von Lena Maxine Lenkeit

Vorwort

Medienkompetenz ist meines Erachtens etwas mehr als im Internet zu surfen, Textbearbeitungsprogramme nutzen zu können, einen PowerPoint-Vortrag zu erstellen oder mit fertigen Simulationen zu spielen. Zur Medienkompetenz gehört auch ein Grundverständnis im Programmieren, denn unser Leben wird in immer stärkerem Maße von Software bestimmt. Wer keine Grundkenntnisse in einer Programmiersprache besitzt, der kann digital auch nicht innovativ sein. Warum sind Google, Facebook, Microsoft, Apple, … keine deutschen Unternehmen?

Kann das Schulfach Informatik dafür sorgen, dass Deutschland in Zukunft in Sachen Digitalisierung nicht noch weiter abgehängt wird? Wohl kaum, denn dazu müsste es verbindlich an allen Schulen eingeführt werden. Dies wird aber in den nächsten Jahren kaum realisiert werden. Eine andere Idee liegt diesem Buch zugrunde. Programmieren sollte man nach dieser Idee nicht nur im Unterrichtsfach Informatik lernen, sondern auch in den Fächern Physik, Mathematik, Biologie, ... Hier können ganz konkrete, fachbezogene Aufgaben gestellt werden, die mittels Programmierung gelöst werden müssen. Somit ist das Erlernen einer Programmiersprache in diesem Buch kein Selbstzweck, sondern es erfolgt zielgerichtet anhand von physikalischen Problemstellungen. Dadurch erfährt der Schüler, dass das Gelernte wirklich nützlich ist.

Dieses hier vorliegende Buch habe ich für Physikschüler und Physiklehrer der gymnasialen Oberstufe geschrieben . Es trägt den Titel „Programmieren lernen für den Physikunterricht mit Processing“. Mit „für den“ und nicht „in dem“ meine ich, dass Programmieren lernen nicht auf Kosten der verbindlichen physikalischen Lehrinhalte gehen soll. Das Buch ist so geschrieben, dass Schüler und Lehrer sich anhand der zahlreichen Beispiele, die sich über die Physik der gesamten Oberstufe erstrecken, selbstständig die hierzu notwendigen Programmierkenntnisse aneignen können. Der Physiklehrer muss den Schülern also nicht Programmieren beibringen. Er sollte die von den Schülern erstellten physikalischen Simulationen schon verstehen, aber unterrichten soll er Physik.

Als leicht zu erlernende Programmiersprache wurde die auf Java basierende kostenlose Programmiersprache Processing gewählt. Sie enthält eine eigene Entwicklungsumgebung und eignet sich besonders gut zur Erstellung von Grafiken, Animationen und physikalischen Simulationen. Durch dieses optische Feedback wirkt sie gerade auf Programmieranfänger sehr stark motivierend.

An dieser Stelle möchte ich besonders meiner ehemaligen Schülerin Lena Maxine Lenkeit danken. Ihr verdanke ich zahlreiche Anregungen und sie war stets bereit viele meiner Fragen zu beantworten. Auch hat sie Sketche zu diesem Buch beigetragen. Weiterhin danke ich Matthäus Mader, der sich die Mühe gemacht hat, mein Skript bezüglich Orthographie kritisch durchzulesen.

Ich wünsche allen Lesern viel Freude bei der Erstellung ihrer eigenen physikalischen Simulationen.

Walter Stein — Bad Münstereifel im Februar 2018

Inhaltsverzeichnis

1 Einführung in Processing ... 1
1.1 Zeichnen von ebenen geometrischen Figuren ... 1
1.2 Statischer und aktiver Modus ... 5
1.3 Zusammenfassung ... 9
1.4 Aufgaben ... 11
2 Mechanik ... 12
2.1 Einfache Bewegungen in der Pixelwelt ... 12
2.1.1 Bewegungen mit konstanter Geschwindigkeit ... 12
2.1.2 Beschleunigte Bewegungen ... 16
2.1.3 Zusammenfassung ... 20
2.1.4 Aufgaben ... 21
2.2 Wurfbewegungen ... 23
2.2.1 Senkrechter Wurf ... 23
2.2.2 Waagerechter Wurf ... 25
2.2.3 Schräger Wurf ... 26
2.2.4 Zusammenfassung ... 29
2.2.5 Aufgaben ... 29
2.3 Impulserhaltung ... 30
2.3.1 Unelastischer Stoß ... 30
2.3.2 Elastischer Stoß ... 32
2.3.3 Zusammenfassung ... 34
2.3.4 Aufgaben ... 34
2.4 Kreisbewegungen ... 36
2.4.1 Bewegungen von Himmelskörpern ... 36
2.4.2 Uhr mit pushMatrix und popMatrix ... 39
2.4.3 Corioliskraft ... 41
2.4.4 Zusammenfassung ... 45
2.4.5 Aufgaben ... 46
2.5 Einführung in die Vektorrechnung ... 47
2.5.1 Addition und Subtraktion ... 48
2.5.2 Multiplikation eines Vektors mit einer Zahl ... 53
2.5.3 Skalares Produkt ... 57
2.5.4 Kreuzprodukt ... 60

2.5.5 Übersicht über die Rechenoperationen für Vektoren ... 63
2.5.6 Zusammenfassung ... 64
2.5.7 Aufgaben ... 65
3 Felder ... 69
3.1 Einführung ... 69
3.2 Feldstärke ... 69
3.3 Potenzial ... 73
3.4 Potenzialgebirge ... 87
3.5 Gravitationsgesetz ... 93
3.6 Zusammenfassung ... 106
3.7 Aufgaben ... 108
4 Elektrik ... 111
4.1 Spannungsteiler ... 111
4.2 Isotope im E- und B-Feld ... 115
4.3 Schraubenbahn und Spiralbahn ... 123
4.4 Bewegte Leiterschleifen im homogenen Magnetfeld ... 130
4.5 Wechselstromkreis ... 137
4.6 Zusammenfassung ... 143
4.7 Aufgaben ... 145
5 Schwingungen ... 148
5.1 Harmonische Schwingungen ... 148
5.2 Gedämpfte Schwingungen ... 153
5.3 Überlagerung von Schwingungen ... 155
5.4 Processing und Soundkarte ... 159
5.5 Zusammenfassung ... 167
5.6 Aufgaben ... 168
6 Wellen ... 171
6.1 Wellenmaschine ... 171
6.2 Stehende Welle ... 175
6.3 Stehende Welle 3D animiert ... 177
6.4 Hertzsches Gitter ... 180
6.5 Doppelspalt ... 182
6.6 Farbmischungen ... 185
6.7 Spektren ... 187
6.8 Zusammenfassung ... 199
6.9 Aufgaben ... 200

7 Quantenphysik und Atomphysik 202
7.1 Doppelspalt „quantenphysikalisch“ 202
7.2 Zeigerformalismus 205
7.3 Die vier sichtbaren Linien der Balmerserie des Wasserstoffs 213
7.4 Orbitalmodelle des Wasserstoffs 216
7.5 Zusammenfassung 219
7.6 Aufgaben 220
8 Ideale Gase und Festkörper 222
8.1 Ideale Gase 222
8.1.1 Ideales Gas ohne Teilchenwechselwirkung 222
8.1.2 Entropie 225
8.1.3 Ideales Gas mit Wechselwirkungen 228
8.2 Festkörper 231
8.2.1 Festkörper 01 231
8.2.2 Lennard-Jones-Potenzial 233
8.2.3 Festkörper 02 237
8.2.4 Festkörper 03 241
8.3 Zusammenfassung 244
8.4 Aufgaben 245
9 Kernphysik und Teilchenphysik 247
9.1 Nebelkammer 247
9.2 Zählrohr und Nulleffekt 249
9.3 Atomkern 252
9.4 Betaminuszerfall 256
9.5 Zerfallsgesetz 260
9.6 Paarbildung und Zerstrahlung 263
9.7 Zusammenfassung 266
9.8 Aufgaben 266
10 Relativitätstheorie 269
10.1 Massenzunahme, Längenkontraktion und Zeitdilatation 269
10.2 Rotverschiebung 272
10.3 Aufgaben 276
11 Chaos und Fraktale 279
11.1 Deterministisches Chaos 279
11.1.1 Wege ins Chaos 279
11.1.2 Rundungsproblem 282

11.2 Fraktale ... 289
11.3 Zusammenfassung ... 298
11.4 Aufgaben ... 299
12 Processing und Arduino ... 300
12.1 Auf- und Entladevorgang eines Kondensators ... 300
12.2 Entfernungsmessung ... 306
12.3 Zusammenfassung ... 308
12.4 Aufgaben ... 309
13 Kommentierte Literatur- und Linkliste ... 311
14 Stichwortverzeichnis ... 313

1 Einführung in Processing

Was erwartet uns?

Processing Development Environment (PDE), size(), background(), fill(), ellipse(), println(), point(), line(), triangle(), rect(), stroke(), strokeWeight(), noStroke(), void setup(), void draw(), statischer Modus, aktiver Modus, translate(), Referenz

1.1 Zeichnen von ebenen geometrischen Figuren

Processing ist eine auf Java basierende, leicht zu erlernende Programmiersprache zur Erstellung von Grafiken, Animationen und Simulationen. Durch dieses optische Feedback wirkt sie gerade auf Programmieranfänger sehr stark motivierend. In dem vorliegenden Buch wollen wir Schritt für Schritt lernen, wie man mittels Processing physikalische Sachverhalte optisch ansprechend darstellen und simulieren kann. Processing ist kostenlos und kann für die Plattformen Windows, Mac OS X und Linux unter dem folgenden Link heruntergeladen werden:

https://www.processing.org/download/

Nach dem Downloaden muss die Datei noch entpackt und in den Programmordner des Computers verschoben werden. Processing muss nicht installiert werden. Ein Doppelklick auf *processing.exe* genügt, um Processing zu starten. Nun öffnet sich die Entwicklungsumgebung (Processing Development Environment, kurz PDE) von Processing (Abb. 1.1).

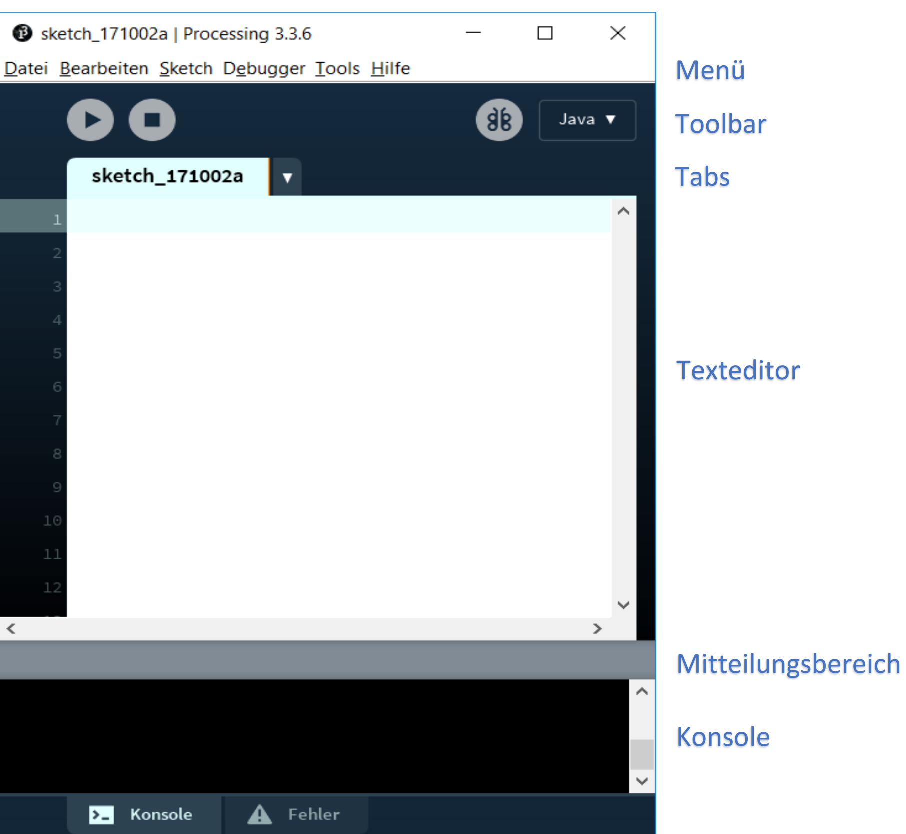
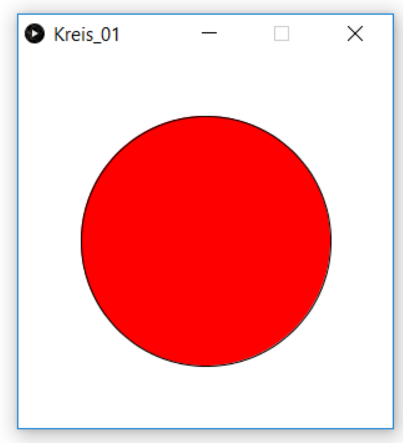


Abbildung 1.1: Die Entwicklungsumgebung von Processing (PDE)

Bevor wir nun unser erstes Programm schreiben, schauen wir uns einmal einige Beispiele an. Dazu klicken wir im Menü auf *Datei* und dann auf *Beispiele*. Anhand der vielen Beispiele bekommen wir einen ersten Eindruck von der Leistungsfähigkeit von Processing.

Nachdem wir uns genügend Beispiele angeschaut haben, schreiben wir nun mal mutig unseren ersten Sketch. Sketch? Bei Processing ist es üblich ein Programm Sketch (Plural Sketches) zu nennen. Sketch bedeutet übersetzt Skizze und kann als Hinweis darauf verstanden werden, dass es sinnvoll ist, bei etwas umfangreicheren Programmieraufgaben seine Gedanken vorab mit Bleistift und Papier zu ordnen. Bei unseren sehr einfachen Sketches in Kapitel 1 ist dies jedoch nicht notwendig.

Zeichnen wir als erstes einen roten Kreis auf weißem Grund. Dazu schreiben wir vier Zeilen in den Texteditor (siehe Abb. 1.2). Klicken wir nun auf das Dreieck in der Toolbar, dann werden unsere Programmzeilen kompiliert, d.h. in einen für den Computer ausführbaren Code umgewandelt, sodass er ein Fenster mit einem roten Kreis auf weißem Grund erzeugen kann. Wenn wir das Fenster wieder schließen wollen, dann klicken wir auf das Kreuz im Fenster oder in der Toolbar auf das schwarze Quadrat rechts neben dem Dreieck in der Toolbar.



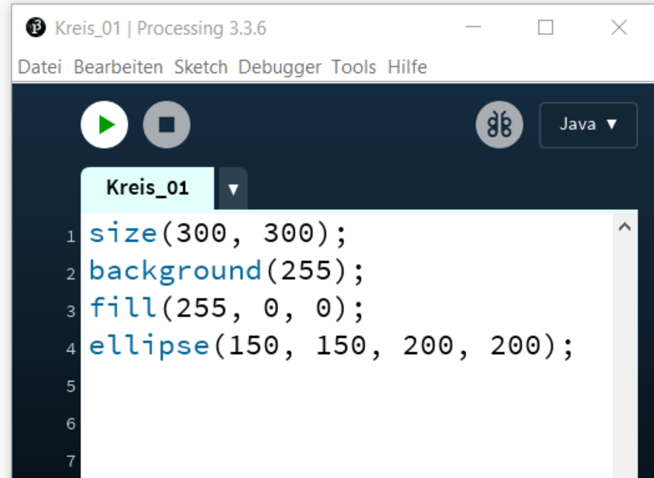


Abbildung 1.2: Unser erster Sketch

Was die Worte im Texteditor bedeuten, kann man mit etwas Englischkenntnissen erraten oder sich wie folgt von Processing anzeigen lassen. Als Beispiel kennzeichnen wir hierzu mit gedrückter Maustaste das Wort „**size**", drücken die rechte Maustaste und klicken dann auf "Suche in Referenz" (Abb. 1.3). Hier lesen wir dann unter anderem: „Defines the dimension of the display window width and height in units of pixels." Unser Fenster ist also 300 Pixel breit und 300 Pixel hoch. Mit den anderen Worten im Texteditor könnten wir genauso verfahren, doch sie sollen nun kurz erläutert werden.

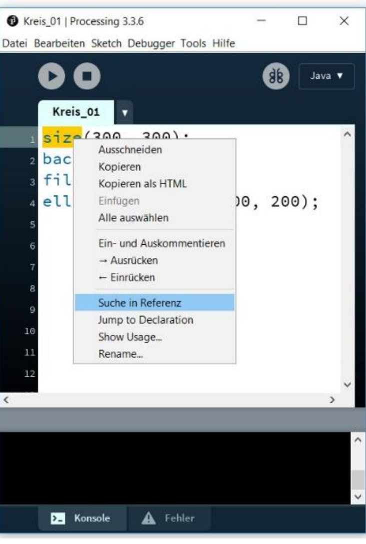


Abbildung 1.3: Suche in der Referenz

Mit der Funktion ***background****(255)* legen wir die Hintergrundfarbe des Fensters fest. Die Zahl 255 in der Klammer entspricht der Farbe Weiß. Die Zahl 0 entspricht der Farbe Schwarz. Die dazwischenliegenden Zahlen geben Grautöne an. Wenn man

den Hintergrund mit anderen Farben füllen will, dann muss man in der runden Klammer drei Zahlenwerte angeben. Processing arbeitet standardmäßig im RGB-Farbraum. D.h., die erste Zahl steht für die Farbe Rot, die zweite für die Farbe Grün und die dritte für die Farbe Blau. 255 bedeutet volle Sättigung. Kleinere Werte ergeben eine geringere Sättigung. Das Gleiche gilt auch für die Funktion ***fill()***. Mit *fill(255, 0, 0)* füllen wir den Kreis mit einem satten Rot. Hätten wir uns einen gelben Kreis gewünscht, dann hätten wir *fill(255, 255, 0)* geschrieben, denn Rot und Grün additiv gemischt ergeben bekanntlich die Farbe Gelb. Den Kreis selbst erstellen wir mit der Funktion ***ellipse****(150, 150, 200, 200)*. Die erste Zahl gibt den x-Wert des Kreismittelpunktes an. Die zweite Zahl gibt den y-Wert des Kreismittelpunktes an. Die dritte Zahl den Durchmesser in x-Richtung und die vierte Zahl den Durchmesser in y-Richtung. Weichen die beiden letzten Zahlen voneinander ab, dann erhalten wir eine Ellipse.

Jede Zeile in unserem Sketch wird mit einem Semikolon beendet. Vergessen wir dies, so erhalten wir im Mitteilungsbereich einen entsprechenden Hinweis (Abb. 1.4).

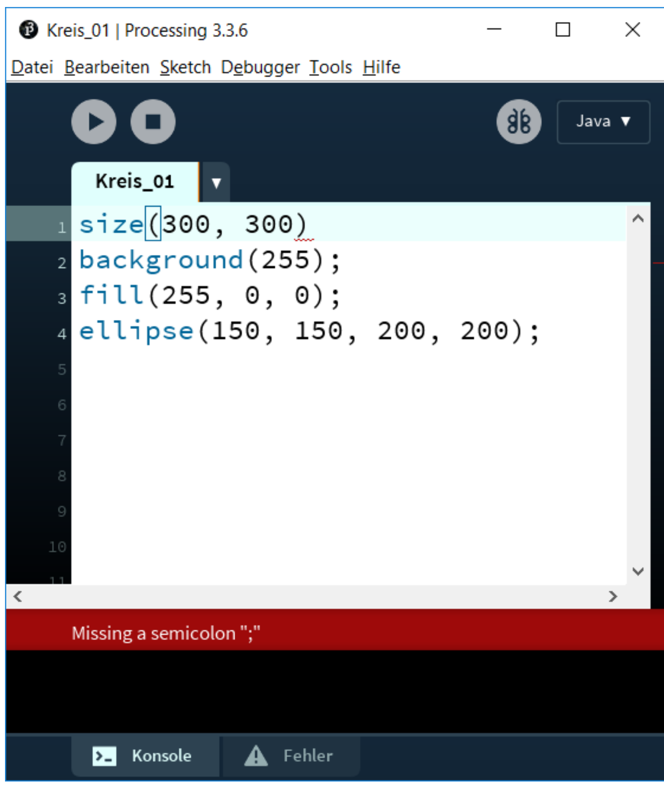


Abbildung 1.4: Ein Fehler wird im Mitteilungsbereich angezeigt

Im Texteditor können wir unseren Sketch auch mit Kommentaren versehen (Abb. 1.5). Für unseren sehr einfachen Sketch ist dies nicht unbedingt notwendig, doch bei komplexeren Sketches dient es dem besseren Verständnis, wenn man den Programmcode zu einem späteren Zeitpunkt nochmal nachvollziehen will. Damit der Computer die Kommentare von den eigentlichen Programmzeilen unterscheiden kann, setzt man vor den Kommentar zwei Schrägstriche. Größere Textpassagen schreibt man so: */* Text */* (siehe Abb. 1.5).

Mit der Funktion ***println()*** kann man dafür sorgen, dass Processing Texte oder Zahlenwerte in die Konsole schreibt (siehe Abb. 1.5). Der Text in der runden Klammer muss jedoch in Anführungszeichen gesetzt werden. Hier ein Beispiel: *println(„Autor Fritzchen Müller“).*

Nun müssen wir unserem Sketch noch einen Namen geben und abspeichern. Dazu klicken wir im Menü auf *Datei* und danach auf *Speichern* bzw. auf *Speichern unter*.

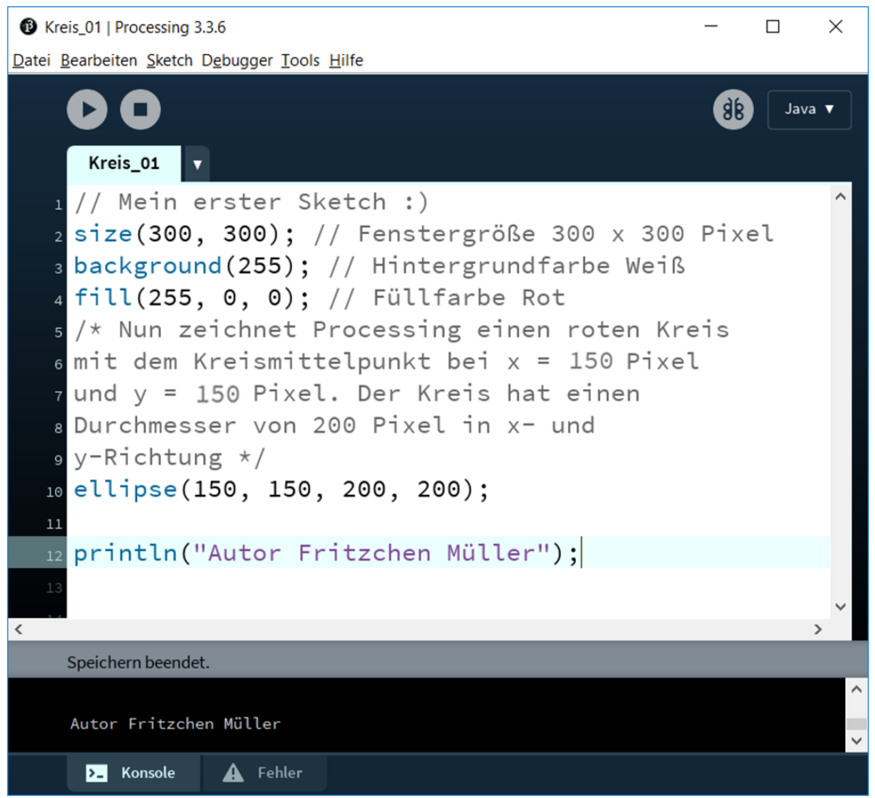


Abbildung 1.5: Ein Sketch mit Kommentaren und einer Information in der Konsole

Außer einem Kreis lernen wir nun noch einige andere Basisfiguren, wie Punkt, Linie, Rechteck und Dreieck kennen (Abb. 1.6). Ein Punkt (point) besitzt in einer zweidimensionalen Zeichnung nur die Koordinaten x und y. Man zeichnet ihn mit der Funktion **point(x, y).** Beispiel: *point(20, 100).* Mit der Funktion **line(x1, y1, x2, y2)** kann man eine Linie zeichnen. Hierzu muss man den Anfangspunkt und den Endpunkt der Linie angeben. Beispiel: *line(50, 30, 100, 150).* Mit **triangle(x1, y1, x2, y2, x3, y3)** zeichnet man ein Dreieck (siehe Abb. 1.6). Beispiel: *triangle(300, 20, 350, 180, 280, 150).* Will man ein Rechteck zeichnen, so gibt man in der Funktion **rect(x, y, Breite, Höhe)** mit x und y den linken oberen Punkt des Rechteckes an und danach die Breite und die Höhe des Rechtecks (Abb. 1.6). Beispiel: *rect(180, 50, 120, 55).*

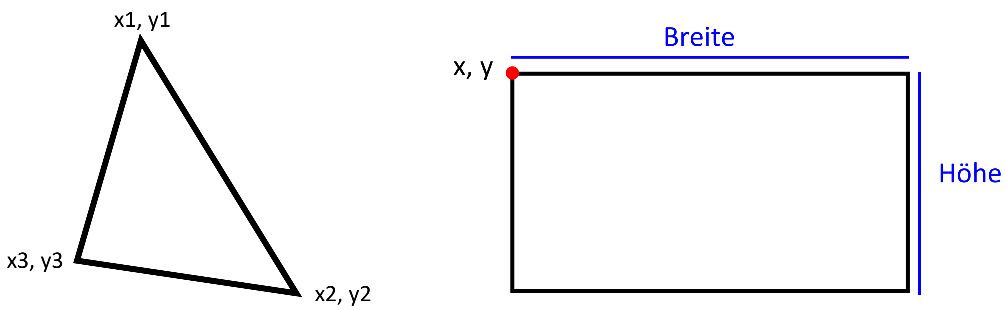
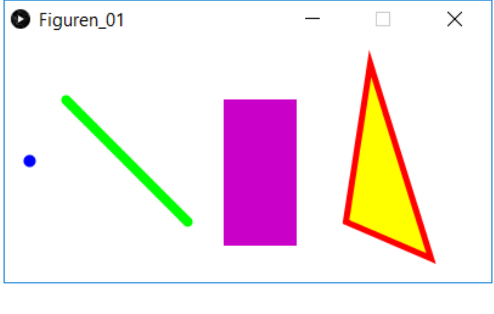


Abbildung 1.6: Konstruktionsangaben für ein Dreieck und ein Rechteck

Die Figuren von Abbildung 1.7 (links oben) zeichnet der Sketch *Figuren_01*. Wenn wir uns die Programmzeilen im Texteditor genau anschauen, dann sehen wir drei weitere Funktionen, **stroke()**, **strokeWeight()** und **noStroke().** Mit *stroke()* legen wir die Farben von Linien und die Farbe der Ränder von Figuren fest. Mit *strokeWeight()* bestimmen wir die Dicke der Linien in Pixel und *noStroke()* sorgt dafür, dass keine um Figuren gezeichnet werden.



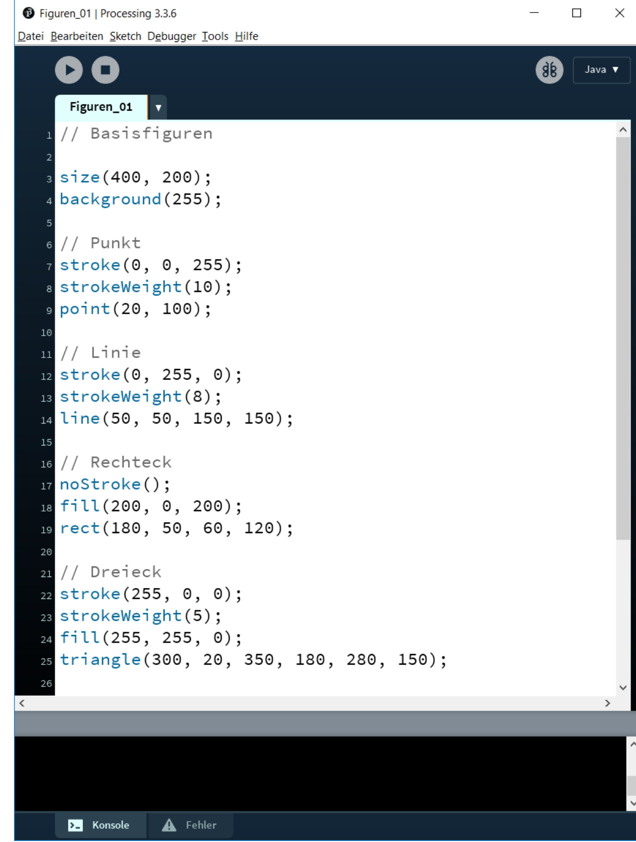


Abbildung 1.7: Ein Sketch, der vier Basisfiguren zeichnet

1.2 Statischer und aktiver Modus

Linien zeichnen

Die obigen Sketches haben wir im **statischen Modus** geschrieben. D.h., wir haben einfach Zeile unter Zeile geschrieben. Sobald wir den Sketch starten, arbeitet Processing brav alle Zeilen nacheinander ab und macht dann Schluss. Wenn wir aber eine Animation oder Simulation erstellen wollen, dann benötigen wir den **aktiven Modus**. Mehr dazu erfahren wir im Kapitel 2. Doch werfen wir auch hier schon mal einen Blick auf einen aktiven Sketch (Abb. 1.8). Zuerst wird die Variable x mittels des vorangestellten Wortes int dem ganzzahligen Bereich zugeordnet und anschließend gleich Null gesetzt. Nun folgt die Funktion **void setup()**, die in den geschweiften Klammern zwei Programmzeilen enthält. Danach folgt die Funktion **void draw()**, die in den geschweiften Klammern vier Programmzeilen enthält. Den Programmteil zwischen geschweiften Klammern nennt man **Codeblock**. Anhand von *void setup()* und *void draw()* erkennt Processing, dass es nun im aktiven Modus arbeiten muss. Wie geht Processing nun vor? Es arbeitet die Zeilen zwischen den geschweiften Klammern von *void setup()* in der vorgegebenen Reihenfolge einmal ab. Danach arbeitet es die Zeilen zwischen den geschweiften Klammern von *void draw()* nicht nur

einmal ab, sondern immer wieder erneut. Solange bis man auf den Stopp-Button drückt. In dem Sketch von Abbildung 1.8 werden also fortlaufend rote Linien gezeichnet, obwohl hier nur einmal *line(x, 10, x, 190)* steht. Diese Linien sind jeweils um 10 Pixel gegeneinander versetzt, da in der geschweiften Klammer von *void draw()* x = x + 10 steht. Bei jedem Durchlauf von *void draw()* wird also der Wert 10 zu x hinzuaddiert, sodass der x-Wert für die jeweils folgende Linie um 10 Pixel größer ist als der x-Wert der vorhergehenden Linie.

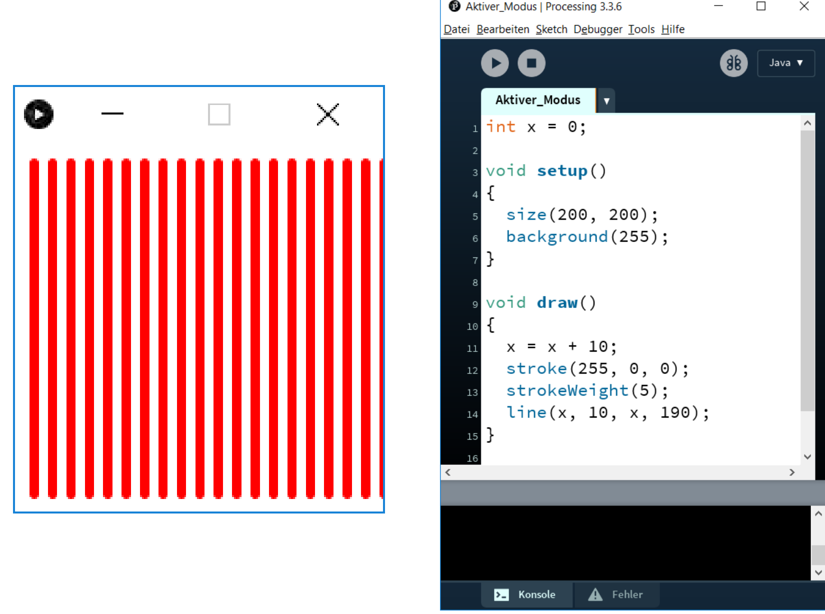


Abbildung 1.8: Sketch im aktiven Modus

Graphen zeichnen

Schauen wir uns noch ein zweites Beispiel für den aktiven Modus an. Wir wollen den Graphen der Funktion y = 2·x + 50 zeichnen. In einem Tabellenkalkulationsprogramm wie *Excel* oder *Calc* ist dies recht einfach (siehe Abb. 1.9 links). Bei Processing gibt es jedoch ein Problem, wenn man die Funktionswerte wie gewohnt eingibt. Wir erhalten keine steigende, sondern eine fallende Gerade (Abb. 1.9 rechts). Warum ist das so?

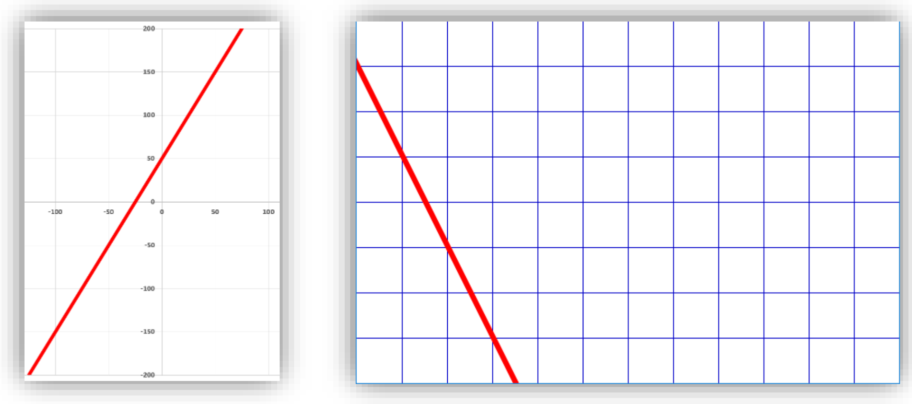


Abbildung 1.9: Graph mit Excel gezeichnet (links) und Graph mit Processing gezeichnet (rechts)

Entgegen unserer Gewohnheit zeigt bei Processing die positive y-Achse des Processing-Koordinatensystems nach unten und der Koordinatenursprung liegt in der linken oberen Ecke des Anzeigefensters (blau in Abbildung 1.10). Wenn wir also einen Graphen zeichnen wollen, dann müssen wir dies berücksichtigen.

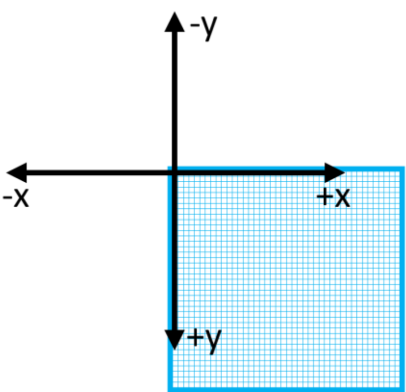


Abbildung 1.10: Anzeigefenster von Processing im Processing-Koordinatensystem

Mit welchen Tricks können wir aber den Sketch so ändern, dass wir eine gewohnte Darstellung erhalten? Wenn wir unsere Funktion y = 2·x + 50 mit minus Eins multiplizieren, also y = -(2·x + 50) schreiben, dann wird der Graph an der x-Achse gespiegelt und wir erhalten einen ansteigenden Graphen. Leider liegt der Graph nun außerhalb des Anzeigefensters und ist nicht mehr zu sehen. Was nun?

Mit der Funktion **translate ()** können wir den Koordinatenursprung des Processing-Koordinatensystems verschieben. Mit *translate(300, 200)* verschieben wir in unserem folgenden Sketch das Processing-Koordinatensystem um 300 Pixel in x-Richtung und um 200 Pixel in y-Richtung und somit in die Mitte unseres 600 Pixel breiten und 400 Pixel hohen Fensters. Nun wird unsere Funktion wie gewohnt dargestellt (Abb. 1.11). Der Trick mit der Kombination von minus Eins und *translate()* hilft uns auch, wenn wir Messwerte in Processing grafisch darstellen wollen.

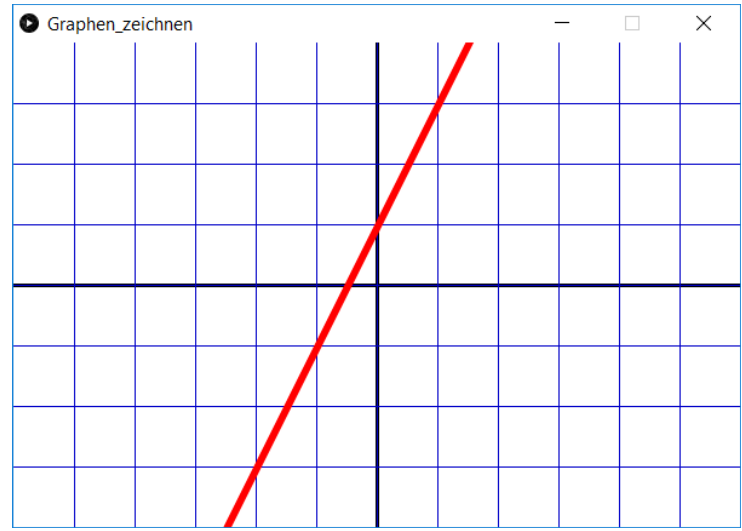


Abbildung 1.11: Richtige Darstellung des Graphen im Anzeigefenster von Processing.
Der Abstand der Rasterlinien beträgt 50 Pixel.

Sketch 01: Graphen_zeichnen

```
// y = m*x+b

int x = -150; // Startwert für x
int y = 0; // Startwert für y
int m = 2; // Steigung
int b = 50; // Schnittpunkt mit der y-Achse
int r = 0; // Variable für die Rasterzeichnung

void setup()
{
  size(600, 400); // 600 Pixel breites und 400 Pixel hohes Fenster
  background(255); // weißer Hintergrund

  // Koordinatensystem wird gezeichnet
  stroke(0); // Linienfarbe schwarz
  strokeWeight(3); // Liniendicke 3 Pixel
  line(300, 0, 300, 400); // y-Achse
  line(0, 200, 600, 200); // x-Achse
}

void draw()
{
  // Ein Raster wird gezeichnet
  r = r + 50; // Schrittweite 50 Pixel
  stroke(0, 0, 200); // Linienfarbe abgeschwächtes blau
  strokeWeight(1); // Liniendicke 1 Pixel
  line(r, 0, r, 400); // senkrechte Linien
  line(0, r, 600, r); // waagerechte Linien

  // Der Ursprung des Koordinatensystems wird in die Fenstermitte verschoben
  translate(300, 200);

  // Die Funktion wird gezeichnet
  y = -(m*x+b); // Die Funktion mit ihren Variablen
  x = x + 1; // x nimmt bei jedem Durchlauf von void draw() um 1 zu

  stroke(255, 0, 0); // Punktfarbe rot
  strokeWeight(6); // Punktdurchmesser 6 Pixel
  point(x, y); // Punkt
}
```

Referenz von Processing

In dieser kurzen Einführung haben wir schon einige Begriffe und Funktionen von Processing kennengelernt. In den folgenden Kapiteln werden wir weitere kennenlernen. Eine Übersicht über die zahlreichen Begriffe, Funktionen und ihre Parameter findet man in der **Referenz** von Processing (Abb. 1.12). Die Referenz öffnet sich im Webbrowser (Firefox, Google Chrome, …), wenn man im Menü auf *Hilfe* und dann auf *Referenz* klickt. Dazu muss man nicht online sein, da sich die Referenz im Programmordner von Processing befindet. Mit dem Shortcut STRG+F (Mac: CMD+F) kann man dann einzelne Begriffe suchen. Klickt man einen so gefundenen Begriff an, dann erhält man eine ausführliche Beschreibung mit jeweils einem oder mehreren Sketches als Beispiele.

Abbildung 1.12: Ein Ausschnitt aus der Referenz von Processing

1.3 Zusammenfassung

Fassen wir zusammen, was wir in Kapitel 1 gelernt haben, bzw. gelernt haben sollten.

Processing Development Environment (PDE)

Im Texteditor der Processing-Entwicklungsumgebung (Processing Development Environment) schreibt man seinen Sketch in für Menschen lesbarer Sprache. Klickt man auf das Dreieck in der Toolbar, dann verwandelt der in der Entwicklungsumgebung integrierte Compiler den Text in die für den Computer lesbare Maschinensprache und startet anschließend die Runtime-Engine, die das Anzeigefenster öffnet und den Sketch ausführt.

size()

Mit *size()* legt man die Breite und die Höhe des Anzeigefensters fest. Im aktiven Modus muss *size()* als erste Zeile bei *void setup()* stehen.

background()

Mit *background()* bestimmt man die Hintergrundfarbe des Anzeigefensters. Steht *background()* bei *void draw()*, dann wird der Hintergrund bei jedem Durchlauf von *void draw()* neu gezeichnet. Steht *background()* bei *void setup()*, dann wird der Hintergrund nur einmal gezeichnet.

fill()

Mit *fill()* kann man die gezeichneten Formen mit Farbe füllen. Steht nur eine Zahl in der Klammer, dann bedeutet die Zahl 0 Schwarz, die Zahl 255 Weiß und die Zwischenwerte ergeben einen Grauwert. Setzt man drei Zahlen zwischen die runden Klammern, dann werden die Formen mit den Farben des RGB-Farbraums gefüllt (rot, grün, blau).

ellipse()

Mit der Funktion *ellipse()* kann man Ellipsen und Kreise zeichnen. Hierzu schreibt man vier Zahlen in die Klammer: *ellipse(x-Wert des Mittelpunktes, y-Wert des Mittelpunktes, Durchmesser in x-Richtung, Durchmesser in y-Richtung).*

println()	Mit *println()* kann man Worte oder Daten in die Konsole schreiben lassen. Gerade zur Überprüfung von berechneten Zahlenwerten ist diese Funktion sehr nützlich.
point()	Mit *point(x, y*) zeichnet man einen Punkt.
line()	Mit *line(x1, y1, x2, y2)* zeichnet man eine Linie.
triangle()	Mit *triangle(x1, y1, x2, y2, x3, y3)* zeichnet man ein Dreieck.
rect()	Mit *rect(x- Wert des oberen linken Eckpunktes, y-Wert des oberen linken Eckpunktes, Breite, Höhe)* zeichnet man ein Rechteck.
stroke()	Mit *stroke()* legen wir die Farben von Linien und die Farben der Umrisse von Figuren fest.
strokeWeight()	Mit *strokeWeight()* bestimmen wir die Dicke der Linien in Pixel.
noStroke()	Mit *noStroke()* sorgt man dafür, dass keine Umrisse um Figuren gezeichnet werden.
void setup()	void setup() wird beim Start des Sketch nur einmal ausgeführt. Hier werden die Anfangsbedingungen festgelegt. Als erste Zeile steht stets die Funktion *size().* Variablen die in *void setup()* aufgeführt sind können nicht bei *void draw()* verwendet werden.
void draw()	Die Funktion *void draw()* wird nach dem Start des Sketch immer und immer wieder ausgeführt, solange bis der Stopp-Button gedrückt wird. Sie eignet sich somit sehr gut für Animationen und Simulationen.
statischer Modus und aktiver Modus	
	statischer Modus ohne *void setup()* und *void draw()*
	aktiver Modus mit *void setup()* und *void draw()*
Codeblock	Den Programmcode zwischen den geschweiften Klammern bezeichnet man als Codeblock.
translate()	Mit der Funktion *translate()* können wir den Koordinatenursprung des Processing-Koordinatensystems im Anzeigefensters verschieben.
Referenz	Eine Übersicht über die zahlreichen Begriffe, Funktionen und ihre Parameter findet man in der Referenz von Processing. Die Referenz öffnet sich im Webbrowser (Firefox, Google Chrome, …), wenn man im Menü auf *Hilfe* und dann auf *Referenz* klickt. Dazu muss man nicht online sein, da sich die Referenz im Programmordner von Processing befindet. Mit dem Shortcut STRG+F (Mac: CMD+F) kann man dann einzelne Begriffe suchen. Klickt man einen so gefundenen Begriff an, dann erhält man eine ausführliche Beschreibung mit einem oder mehreren Sketches als Beispiele.

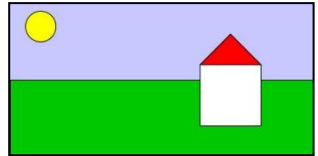
1.4 Aufgaben

1. Zeichne in einem 300 x 400 Pixel großen Fenster ein rotes Dreieck mit grünem Rand auf schwarzem Hintergrund.

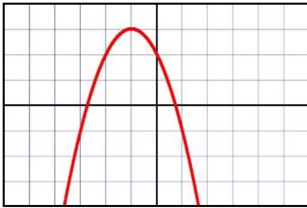
2. Zeichne in die Mitte eines quadratischen Fensters einen weißen Kreis ohne Rand auf violettem Hintergrund. In der Konsole soll der folgende Text stehen: „Kreis ohne Rand“.

3. Zeichne das rechts abgebildete Gesicht mit zwei schwarzen Punkten (keine Kreise) als Augen und einer Linie als Hutkrempe.

4. Zeichne ein 400 x 200 Pixel großes Bild entsprechend der Abbildung rechts. Entwerfe das Bild zuerst mit entsprechenden x-y-Werten auf Papier.



5. Eine Parabel wird durch die Gleichung $y = a \cdot x^2 + b \cdot x + c$ beschrieben. Schreibe einen Sketch, der die in der Abbildung rechts dargestellte Parabel zeichnet. Die Parabel ist nach unten geöffnet und besitzt die folgenden Parameter: a = - 0,02, b = -2 und c = 100. Der Abstand der Gitternetzlinien soll 50 Pixel betragen.



2 Mechanik

Was erwartet uns?

int, float, text(), textSize(), frameRate, frameRate(), millis(), +timer, || (logische ODER), if(), else if(), noLoop(), radians(), && (logisches UND), ! (logisches NICHT), ^ (logisches exklusives ODER), round(), textAlign(CENTER), rotate(), pushMatrix(), popMatrix(), boolean, true, false, add(), sub(), PVector(), mult(), div(), dot(), mag(), angleBetween(), cross(), P3D, rotateX, rotateY, rotateZ,

2.1 Einfache Bewegungen in der Pixelwelt

2.1.1 Bewegungen mit konstanter Geschwindigkeit

Ein Punkt in der Pixelwelt

Nachdem wir in Kapitel 1 gelernt haben wie man unterschiedliche Figuren zeichnet, wollen wir diese nun auch animieren, d.h., auf unserem Monitor in Bewegung setzen. Bewegungen in unserem Alltag zu beobachten sind wir gewohnt und ihre Deutung bereitet uns in der Regel keine Kopfschmerzen. Bei der Beurteilung von Bewegungen in der Pixelwelt müssen wir jedoch etwas mehr nachdenken. Wenn wir Bewegungen in der Pixelwelt von Processing beobachten, dann sehen wir einen Film mit einer bestimmten Anzahl von Bildern pro Sekunde. In der Regel beträgt die Bildwiederholungsrate 60 Bilder pro Sekunde. Mit dem folgenden kleinen Sketch können wir uns die Bildwiederholungsrate (**frameRate**) in der Konsole von Processing anzeigen lassen (siehe Abb. 2.1).

```
void draw()
{
  println (frameRate);
}
```

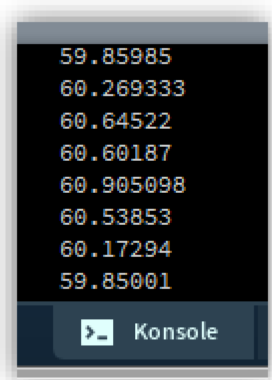


Abbildung 2.1: frameRate

Wenn wir eine höhere Bildwiederholungsrate wünschen, z.B. 100 Bilder pro Sekunde, so können wir dies im Sketch mit
frameRate(100)
einstellen. Vorrausetzung ist aber, dass die eingebaute Grafikkarte eine solche Bildwiederholungsrate leisten kann.

```
void draw()
{
  frameRate(100);
  println (frameRate);
}
```

Nun werden wir aber endlich ein so einfaches Objekt wie einen Punkt in Bewegung setzen. Der folgende Sketch besteht aus drei separaten Teilen. Zuerst wird die Variable x genannt (**deklariert**) und mit ***int*** dem ganzzahligen Bereich zugeordnet (**definiert**). Danach wird der Variablen x auch ein konkreter Wert, hier Null, zugeordnet (**initialisiert**). Bei *void setup()* wird mit *size(800, 100)* die Fenstergröße auf 800 x 100 Pixel festgelegt. *void setup()* wird im Sketch nur einmal durchlaufen, während der Sketchteil *void draw()* immer wieder durchlaufen wird. Einzelheiten werden im Sketch selber erklärt.

Sketch 01: Punkt_in_der_Pixelwelt

```
int x = 0; // Die Variable x wird dem ganzzahligen Bereich zugeordnet
           // und auf den Wert Null gesetzt.

void setup()
{
  size(800, 100); // Das Fenster ist 800 Pixel breit und 100 Pixel hoch.
}

void draw()
{
  background(255); // Der Fensterhintergrund ist weiß

  strokeWeight(1); // Die folgenden beiden Linien haben eine Strichdicke
                   // von einem Pixel
  line(100, 0, 100, 100); // Startlinie
  line(700, 0, 700, 100); // Ziellinie

  x = x + 1; // x wird bei jedem Durchlauf von void draw()um 1 Pixel
             // vergrößert

  stroke(255, 0, 0); // Der Punkt hat die Farbe Rot.
  strokeWeight(10); // Der Punkt hat einen Durchmesser von 10 Pixel.
  point(x, 50); // Bei jedem Durchlauf von void draw()wandert der Punkt
                // in der Höhe von y = 50 um einen Pixel nach rechts
  println(frameRate); // Die Bildwiederholungsrate wird in der Konsole
                      // angezeigt
}
```

Bewegung mit v = konstant

Starten wir nun die Animation, so sehen wir im sich öffnenden Fenster (siehe Abb. 2.2) einen roten Punkt, der sich vom linken Fensterrand über die Startlinie zur Ziellinie und danach weiter zum rechten Fensterrand bewegt.

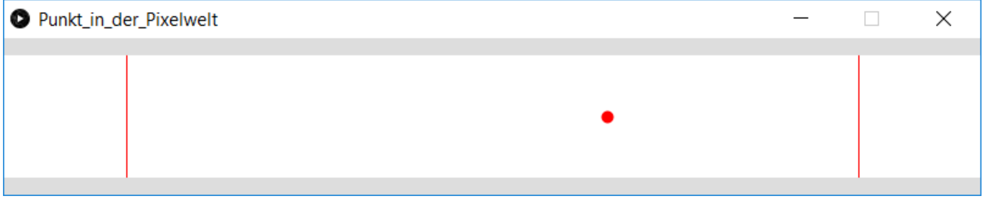


Abbildung 2.2: Ein Punkt bewegt sich in der Pixelwelt

Die Animation im obige Sketch läuft mit einer Bildwiederholungsrate von 60 Bildern pro Sekunde ab. Der Abstand zwischen den beiden roten Linien beträgt 600 Pixel. Da der rote Punkt aufgrund der Einstellung *x = x + 1* bei jedem Durchlauf von *void draw* () einen Pixel weiter nach rechts rückt, bewegt er sich in einer Sekunde um 60 Pixel nach rechts. In 10 Sekunden hat er die ganze Strecke von 600 Pixel durchlaufen. Überprüfen kann man dies mit der Stoppuhr des Smartphones. Stellen wir nun in *void setup()* mit *frameRate(30)* einen Bildrate von 30 Bildern pro Sekunde ein, so braucht der rote Punkt 20 Sekunden um die Strecke zwischen den roten Linien zu durchlaufen.

Nun wollen wir den ganzen Vorgang etwas physikalischer betrachten. Die Gleichung für eine Bewegung mit konstanter Geschwindigkeit lautet: $s = s_0 + v \cdot t$. Wenn wir zuerst $s_0 = 0$ setzen, dann vereinfacht sich die Gleichung zu $s = v \cdot t$. Die unabhängige Variable ist t. Sie bestimmt zusammen

mit der Konstanten v, wie s mit der Zeit zunimmt. Doch um welchen Betrag soll t bei jedem Durchlaufen von void draw() zunehmen? Um diese Frage zu beantworten muss man die Bildwiederholungsrate des Computers kennen oder sie sich in der Processing-Konsole anzeigen lassen. Bei einer Bildwiederholungsrate von 60 Bildern pro Sekunde ist es sinnvoll, wenn bei jedem Durchlauf von *void draw()* zu dem Wert von t der Wert $\frac{1}{60}$ hinzuaddiert wird. So wächst der Wert von t in einer Sekunde von 0 auf den Wert 1 an. Schreiben müssen wir im Sketch **float** t = 0 und t = t + 1.0/60.0. Wir dürfen nicht int t = 0 und t = t + 1/60 schreiben, da 1/60 nicht ganzzahlig ist und somit keine int-Zahl ist. Processing möchte hier mit der Angabe **float** t, 1.0 und 60.0 mitgeteilt bekommen, dass wir keine int-Zahl für t sondern eine **float**-Zahl erwarten, also eine Zahl mit Nachkommastellen. Testen wir nun mit dem folgenden Sketch unsere Überlegungen. Wenn wir v = 60.0 setzen und $t = t + \frac{1.0}{60.0}$, dann erwarten wir, dass der rote Punkt die Strecke von 600 Pixel in 10 Sekunden zurücklegt. Wenn wir wieder die Zeit stoppen, dann sehen wir, dass wir richtig gedacht haben.

Sketch 02: v_gleich_konstant

```
float s0 = 0; // Ort des Punktes beim Start des Sketches
float s = 0; // Ort des Punktes zum Zeitpunkt t
float v = 60; // Geschwindigkeit des Punktes
float t = 0; // Zeit

void setup()
{
  size(800, 100);
}

void draw()
{
  background(255);

  strokeWeight(1);
  line(100, 0, 100, 100); // Startlinie
  line(700, 0, 700, 100); // Ziellinie

  t = t + 1.0/60.0; // Bei der frameRate 60 wächst t nach 60
   // Durchläufen, also in einer Sekunde, auf den Wert von 1 an
  s = s0 + v*t; // Bewegungsgleichung mit v = konstant

  stroke(255, 0, 0);
  strokeWeight(3);
  point(s, 50);

  println(frameRate);
}
```

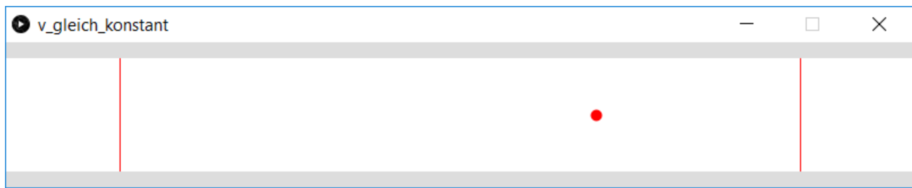


Abbildung 2.3: Bewegung mit v = konstant

Sekundentimer

Wenn wir kein Smartphone zur Hand haben, dann können wir uns mit **millis()** einen sogenannten Timer erschaffen, der die Zeit im Fenster von Processing anzeigt. Der Timer startet, sobald der Sketch aufgerufen wird. Er gibt die Zeit der vergangenen Millisekunden seit dem Start des Sketches an. Soll die Zeit in Sekunden angezeigt werden, dann teilen wir diesen Wert einfach durch 1000. Im folgenden Sketch durchwandert der Punkt ein 600 Pixel breites Fenster mit der Geschwindigkeit v = 120. In dem Moment, in dem der Punkt das Fensterende erreicht hat, sollte der Timer bei einer Bildwiederholungsrate von 60 Bildern pro Sekunde den Wert 5 anzeigen. Wichtig ist, dass wir den Wert des Timers bei *void draw()* abrufen. Rufen wir den Wert des Timers im ersten Teil des Sketches ab, dann zählt dieser nicht mit.

Damit der Timer nicht nur im Hintergrund läuft, müssen wir ihn wie folgt im Fenster darstellen.

```
  int timer = millis()/1000; // Unser Timer gibt die Zeit in Sekunden an
  fill(0); // Textfarbe
  textSize(30); // Textgröße
  text("t in s = " +timer, 200, 30); // Text, Timer und Lage des Textes
```

In den obigen Programmzeilen sehen wir zum ersten Mal die Funktionen **textSize()** und **text()**. Mit *textSize()* legen wir die Größe der Schrift in unserem Fenster fest. In die Klammer der Funktion text() schreibt man den Text stets in Anführungszeichen. Mit den beiden Zahlen (x und y) in der Klammer bestimmt man den linken Fußpunkt des Textes. Mit **+timer** in der letzten Zeile des obigen Sketches sorgt man dafür, dass die Zeit im Fenster angezeigt wird, die nach dem Start des Sketchs vergangen ist. Zu beachten ist weiterhin, dass die Textfarbe nicht mit *stroke*, sondern mit *fill* festgelegt wird. Bei *text()* wird nur der zu schreibende Text und nicht *timer* in Anführungszeichen gesetzt. Vor dem Wort *timer* muss ein Pluszeichen gesetzt werden, damit der Wert des Timers an den Text angehangen wird. In Abbildung 2.4 sehen wir das Ergebnis unserer Überlegungen und darunter den vollständigen Sketch.

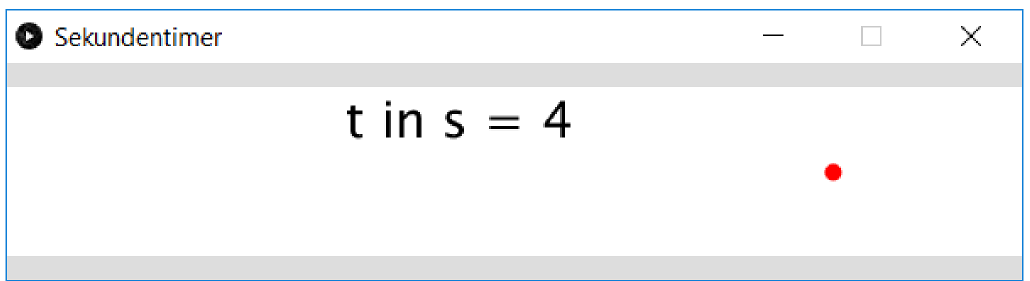


Abbildung 2.4: Anzeige der Zeit im Fenster

Hier ist nun der vollständige Sketch.

Sketch 03: Sekundentimer

```
float s0 = 0; // Ort des Punktes beim Start des Sketches
float s = 0; // Ort des Punktes zur Zeit t
float v = 120; // Geschwindigkeit des Punktes
float t = 0; // Zeit

void setup()
{
  size(600, 100); // 600 Pixel breites und 100 Pixel hohes Fenster
```

```
}

void draw()
{
  background(255);

  t = t + 1.0/60.0; // Bei der frameRate 60 wächst t nach 60 Durchläufen
                    // auf den Wert von 1 Sekunde an
  s = s0 + v*t; // Bewegungsgleichung mit v = konstant

  stroke(255, 0, 0);
  strokeWeight(10);
  point(s, 50);

  int timer = millis()/1000; // Unser Timer zeigt die Zeit in Sekunden
                             // an
  fill(0); // Textfarbe
  textSize(30); // Textgröße
  text("t in s = " +timer, 200, 30); // Text, Timer und Lage des Textes
  println(frameRate);
}
```

2.1.2 Beschleunigte Bewegungen

Die Gleichung $s(t) = v \cdot t + s_0$ gilt für Bewegungen mit konstanter Geschwindigkeit. Für gleichmäßig beschleunigte Bewegungen gilt $s(t) = \frac{1}{2} a \cdot t^2 + v_0 \cdot t + s_0$. Ändern wir unseren letzten Sketch also entsprechend um. In dem neuen Sketch (s.u.) schreiben wir s = s0 + v0*t + 0.5*a*t*t. Die Angabe 0.5 bedingt aber, dass wir auch die anderen, in der Gleichung enthaltenen Variablen als float- und nicht mehr als int-Werte definieren dürfen.

Überlegen wir uns nun, wie lange der rote Punkt bei einer Beschleunigung von a = 5 ms^{-2} braucht, um das 600 Pixel große Fenster zu durchwandern. Ein Pixel soll 1m betragen. Damit es etwas einfacher wird, setzen wir s0 = 0 und v0 = 0. Dadurch vereinfach sich die obige Gleichung zu s = 0.5*a*t*t. Setzen wir die genannten Werte ein, so erhalten wir die folgende Gleichung 600 = 0.5*5*t*t. Daraus folgt $t = \sqrt{\frac{600\ m}{0.5*5\ ms^{-2}}} \approx 15{,}49\ s$. Der rote Punkt durchläuft das Fenster somit in 15,49 Sekunden. Nun müssen wir wieder dafür sorgen, dass unsere Uhr richtig läuft. Dazu muss man wie im vorherigen Sketch die Bildwiederholungsrate seines Computers kennen oder man schreibt einfach anstelle von t = t + 1.0/60.0 in seinem Sketch

t = t + 1.0/frameRate

Processing setzt dann den Wert für die *frameRate* selber ein. So erhalten wir für jede Bildwiederholungsrate den richtigen Zeitwert. D.h., es ist eine Sekunde vergangen, wenn *void draw()* so oft durchlaufen wurde, wie es die *frameRate* angibt.

Testen wir nun mit dem folgenden Sketch, ob wir eine beschleunigte Bewegung beobachten können und ob unser oben berechneter Zeitwert von 15,49 Sekunden stimmt. Die genauen Werte können wir uns mit *println(frameRate, s, t);* in der Konsole anzeigen lassen.

Sketch 04: beschleunigte_Bewegung_mit_Sekundentimer

```
float s0 = 0; // Ort des Punktes beim Start des Sketchs
float v0 = 0; // Geschwindigkeit des Punktes beim Start des Sketchs
float s = 0; // Ort des Punktes zur Zeit t
float a = 5; // Beschleunigung
```

```
float t = 0; // Zeit

void setup()
{
  size(600, 100); // 600 Pixel breites und 100 Pixel hohes Fenster
}

void draw()
{
  background(255);

  t = t + 1.0/frameRate; // Bei z.B. einer frameRate von 60 wächst t
                         // nach 60 Durchläufen auf den Wert von 1 an
  s = 0.5*a*t*t; // Bewegungsgleichung mit a = konstant

  stroke(255, 0, 0);
  strokeWeight(10);
  point(s, 50);

  int timer = millis()/1000; // Unser Timer gibt die Zeit in Sekunden an
  fill(0); // Textfarbe
  textSize(30); // Textgröße
  text("t in s = " +timer, 200, 30); // Text, Timer und Lage des Textes

  // Die Werte mit Nachkommastellen können wir in der Konsole ablesen
  println(frameRate, "s in m =", s, "t in s =", t);
}
```

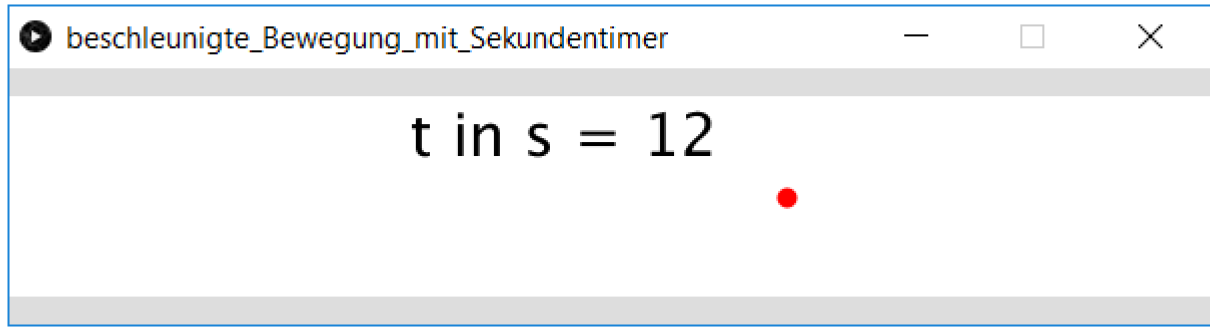


Abbildung 2.5: Beschleunigte Bewegung – Screenshot bei 12 Sekunden

In der Konsole steht s in m = 599.7255 und t in s = 15.48839. Dies bedeutet, dass wir richtig gedacht und richtig programmiert haben.

Autorennen

Nun wollen wir ein Autorennen auf einer Kurzstrecke simulieren. Ein Pixel soll hier wieder einem Meter entsprechen. Anstelle von Punkten stellen wir die beiden Autos als Rechtecke dar (siehe Abb. 2.6). Das rote Auto fährt von Beginn an mit einer Geschwindigkeit von 45 ms^{-1}. Das blaue Auto besitzt keine Anfangsgeschwindigkeit. Es beschleunigt jedoch während der ganzen Fahrt mit konstant 10 ms^{-2}. Die gelbe Ziellinie ist 500 m vom Startort (linke Bildgrenze) entfernt. Welches Auto überfährt als erstes vollständig die gelbe Ziellinie? Die Zeit soll möglichst genau im Fenster angezeigt werden. Sobald ein Auto die Ziellinie überfahren hat, soll die Uhr angehalten werden.

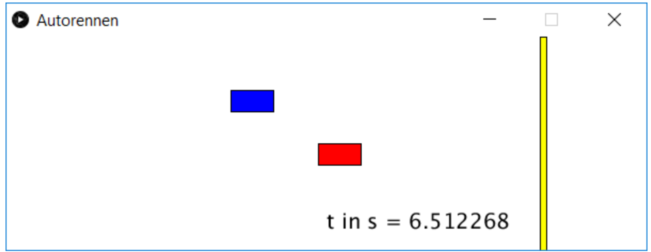


Abbildung 2.6: Autorennen

Um die obigen Bedingungen umzusetzen, müssen wir wieder etwas nachdenken und auch etwas Neues lernen. Im vorhergehenden Sketch hatten wir die Idee, den Wert von t entsprechend der Gleichung **t = t + 1/*frameRate*** zu erhöhen. *frameRate* bedeutet Bildwiederholungsrate pro Sekunde. Wir wissen also, dass genau eine Sekunde vergangen ist, wenn *void draw()* so oft durchlaufen worden ist, wie die *frameRate*. Bei einer *frameRate* von z.B. 60 also nach 60 Durchläufen. Aufgrund dieses Wissens, brauchen wir auch keinen Timer mehr mit int timer = millis()/1000 in unseren Sketch einzubauen. Es genügt, wenn wir mit *text("t in s = " +t, 300, 180)* die Zeit im Fenster anzeigen lassen.

Neu im unten aufgeführten Sketch sind auch die folgenden Zeilen. Sie bedürfen aber einer näheren Erklärung. Frei übersetzt bedeuten sie: „Wenn das hintere Ende des roten Autos (s1) oder das hintere Ende des blauen Autos (s2) die 500 Pixel entfernte, 6 Pixel breite Ziellinie überfahren hat, dann halte den Schleifendurchlauf an."

```
if (s1 >= 507 || s2 >= 507)
{
  noLoop();
}
```

Wir begegnen hier zum ersten Mal einer **if-Anweisung**. Sie kann so gelesen werden: „Wenn die Bedingung in der runden Klammer erfüllt ist, dann tue das, was zwischen den geschweiften Klammern steht." In der runden Klammer steht (s1 >= 507 || s2 >= 507). Das Zeichen >= für „größer gleich" ist wohl jedem bekannt. Das Zeichen **||** steht für das logische **ODER**. Auf meiner Tastatur befindet sich der senkrechte Strich auf der Taste, wo auch < und > zu finden sind. Um den senkrechten Strich zu schreiben, muss man die AltGr-Taste gedrückt halten. Zwischen den geschweiften Klammern steht **noLoop()**. Loop heißt übersetzt Schleife und mit *noLoop()* kann man den Schleifendurchlauf von *void draw()* anhalten. Eigentlich ganz einfach.

Sketch 05: Autorennen

```
// Autorennen mit Zeitangabe im Fenster

float s1 = 0; // rotes Auto
float s2 = 0; // blaues Auto
float v = 45; // rotes Auto
float a = 10; // blaues Auto
float t = 0.0;// Zeit

void setup()
{
  size(600, 200); // Fenstergröße
}

void draw() {
  background(255); // weißer Hintergrund
```

```
  fill(255, 0, 0);
  rect(s1, 100, 40, 20);  // rotes Auto

  fill(0, 0, 255);
  rect(s2, 50, 40, 20); // blaues Auto

  fill(255, 255, 0);
  rect(500, 0, 6, 200); // gelbe Ziellinie

  t = t + 1/frameRate; /* Es ist eine Sekunde vergangen, wenn void
  draw()so oft durchlaufen wurde wie die Bildwiederholungsrate pro
  Sekunde */

  s1 = v*t; // rotes Auto
  s2 = 0.5*a*t*t; // blaues Auto

  /* Mit dieser if-Anweisung wird die Schleife beendet,
   sobald ein Auto vollständig die Ziellinie passiert hat */
  if (s1 >= 507 || s2 >= 507)
  {
    noLoop();
  }

  // Mit +t wird die aktuelle Zeit in Sekunden im Fenster angezeigt
  fill(0);
  textSize(20);
  text("t in s = " +t, 300, 180);

  println(t, s1, s2);
}
```

Wenn wir uns die Daten für t, s1 und s2 mit *println(t, s1, s2)* in der Konsole anzeigen lassen, dann können wir die t-s-Diagramme der beiden Autos mit einem Tabellenkalkulationsprogramm, wie zum Beispiel Excel oder Calc zeichnen lassen. Dazu muss man mit der gedrückten linken Maustaste einige Werte in der Konsole markieren und anschließend die Tastenkombination strg + A drücken. Nun sind alle Werte markiert. Mit der Tastenkombination strg + C kopieren wir diese Werte in die Zwischenablage und fügen sie danach mit dem Textkonvertierungsassistenten des Tabellenkalkulationsprogramms in unser Tabellenkalkulationsprogramm ein. Man achte darauf, dass Processing einen Punkt als Dezimaltrennzeichen benutzt. Deutschsprachige Tabellenkalkulationsprogramme benutzen dagegen ein Komma.
Das Ergebnis nach dem Einfügen sehen wir in Abbildung 2.7. Bei ca. 9 Sekunden überholt das blaue Auto das rote Auto. Wie man solche Diagramme in Processing erstellt lernen wir später.

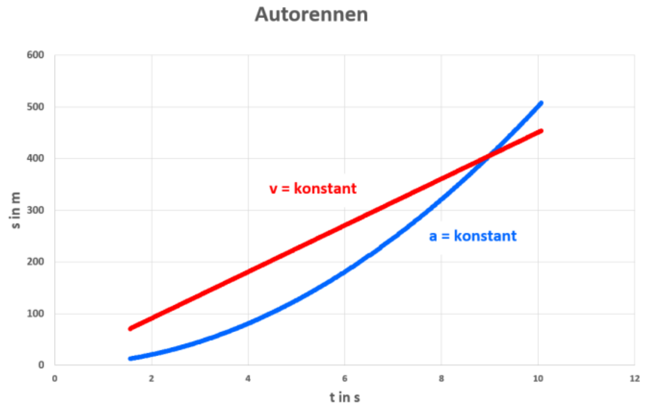


Abbildung 2.7: Diagramm Autorennen

2.1.3 Zusammenfassung

Fassen wir zusammen, was wir in Kapitel 2.1 *Einfache Bewegungen in der Pixelwelt* gelernt haben, bzw. gelernt haben sollten.

deklarieren, definieren und initialisieren

int x = 0; Zuerst wird die Variable x genannt (**deklariert**) und mit *int* dem ganzzahligen Bereich zugeordnet (**definiert**). Danach wir der Variablen x auch ein konkreter Wert, hier Null, zugeordnet (**initialisiert**).

void setup() wird nur einmal durchlaufen

void draw() wird immer wieder durchlaufen

int Mit *int* (Integer) legt man fest, dass die folgende Variable, zum Beispiel x, zum Datentyp der ganzen Zahlen gehört. Der Datentyp Integer kann Zahlenwerte von 2.147.483.647 bis -2.147.483.648 speichern.

float *float* ist ein Datentyp für Fließkommazahlen, also Zahlen mit einem Komma.

t = t + 1.0/frameRate Es dauert genau eine Sekunde, bis t um 1 erhöht wird.

millis() Mit Hilfe von millis() kann man sich die verstrichene Zeit in Millisekunden anzeigen lassen.

Konsole Wenn man sich die Werte im Sketch in der Konsole anzeigen lässt, dann kann man überprüfen, ob man die vorher mit Bleistift und Papier berechneten Werte auch wirklich erhält.

Mit strg+A und strg+C kann man die Werte der Konsole in die Zwischenablage kopieren und diese dann anschließend in ein Tabellenkalkulationsprogramm einfügen.

if-Anweisung Beispiel

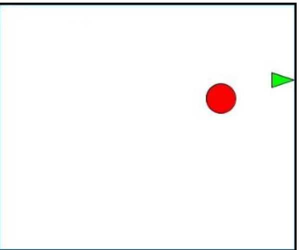
```
if (x >= 100)
{
noLoop();
}
```

Wenn die Bedingung in der runden Klammer erfüllt ist, dann führt Processing das aus, was zwischen den geschweiften Klammern steht.

noLoop() Mit noLoop() kann man den Schleifendurchlauf von void draw beenden.

|| Das Zeichen || steht für das logische Oder.

2.1.4 Aufgaben



1. In einem 400 Pixel breiten und 300 Pixel hohen Fenster befindet sich am Boden bei x = 300 Pixel ein roter Luftballon. Bei x = 50 Pixel und y = 70 Pixel befindet sich die rechte Spitze eines kleinen grünen Dreiecks. Beim Start des Sketches fliegt der Luftballon mit einer konstanten Geschwindigkeit nach oben und das grüne Dreieck mit konstanter Geschwindigkeit nach rechts. Sobald einer von beiden den Rand des Fensters berührt, soll die Bewegung von beiden gestoppt werden (siehe Abbildung). Simuliere diesen Vorgang und überprüfe die Richtigkeit deiner Programmierung mit unterschiedlichen Geschwindigkeiten.

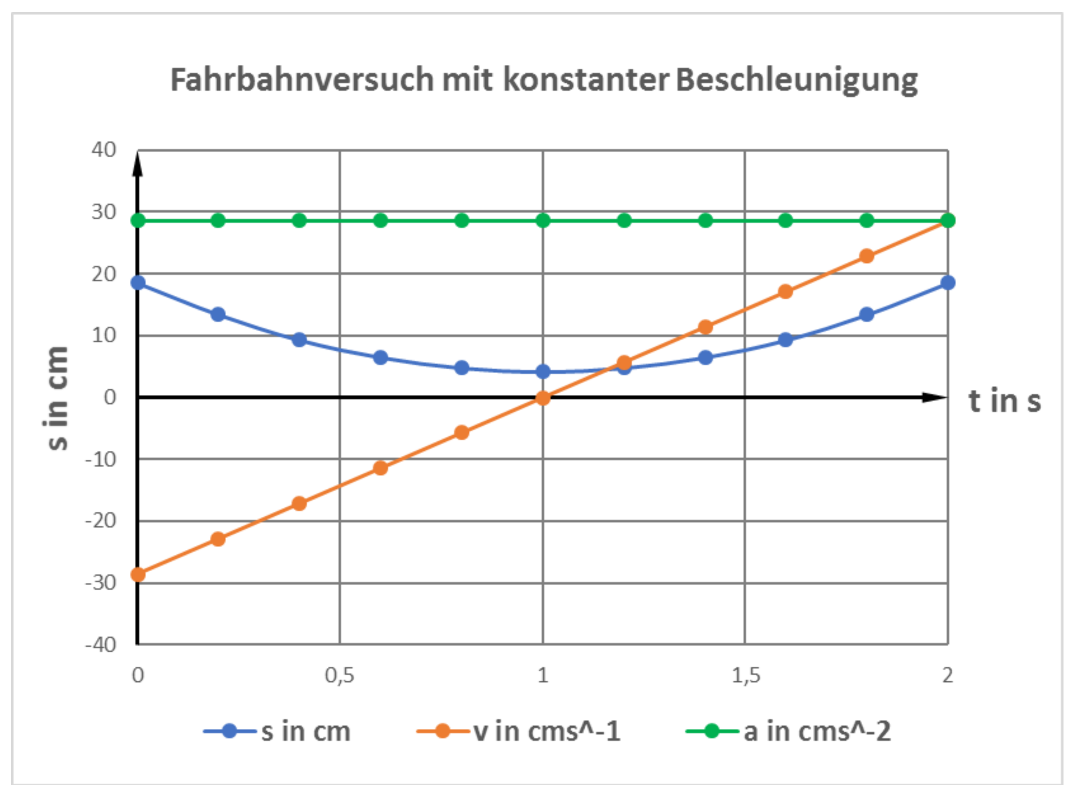
2. Simuliere folgende Gegebenheit in einem 600 Pixel breiten und 200 Pixel hohen Fenster mit hellgrauem Hintergrund. Die Bildwiederholungsrate (*frameRate*) soll 60 Bilder pro Sekunde betragen.
   Ein gelber Punkt mit einem Durchmesser von 20 Pixel bewegt sich mit einer konstanten Geschwindigkeit von v = 20 Pixel/s in einer Höhe von 120 Pixel von links nach rechts durch das Fenster. Ein violetter Punkt bewegt sich in einer Höhe von 80 Pixel mit einer konstanter Beschleunigung a = 5 Pixel/s² von rechts nach links durch das Fenster. Wenn beide Punkte sich begegnen, sollen sie ihre Farbe tauschen. Berechne mit Bleistift und Papier, welche Zeit bis zur Begegnung der beiden Punkte vergangen ist. Überprüfe dein Ergebnis mithilfe der Konsole.

3. Simuliere den freien Fall einer Kugel ohne Luftreibung. Die Fallstrecke soll 400 m betragen (1 Pixel ≙ 1 m). Die Fallzeit soll in Sekunden und die Geschwindigkeit in Meter pro Sekunde im Fenster angezeigt werden. Wenn die Kugel 400 m tief gefallen ist, soll die Simulation stoppen. Überprüfe die angezeigten Werte für die Fallzeit und die Endgeschwindigkeit mittels einer Rechnung mit Bleistift und Papier. $g = 9{,}81\ ms^{-2}$

4. Schreibe einen Sketch für den freien Fall einer Kugel mit Luftreibung. Die Fallstrecke soll wie in Aufgabe 3 wieder 400 m betragen (1 Pixel ≙ 1 m). Die Fallgeschwindigkeit soll im Fenster angezeigt werden. Wenn die Kugel 400 m tief gefallen ist, soll die Simulation stoppen. Die Werte für a_{Luft}, v und s sollen in der Konsole angezeigt werden. Schaue hier nach, ab welcher Fallstrecke gilt: v = konstant.

   Tipp: Die Luftreibung beschleunigt mit $a_{Luft} = k \cdot v^2$ den fallenden Körper in Gegenrichtung zur Erdbeschleunigung g. Wenn $a_{Luft} = g$ ist, dann fällt der Körper mit konstanter Geschwindigkeit. Verwende für die Konstante k den Wert k = 0,013. Die Konstante k fasst die Werte für den Widerstandsbeiwert, die Dichte der Luft, die Schattenfläche der Kugel und die Masse der Kugel zusammen. Da wir für die Bewegung mit Luftreibung keine einfachen Gleichung aufstellen können, müssen wir die Aufgabe numerisch lösen. D.h., für kleine Zeiträume Δt betrachten wir die resultierende Beschleunigung $g - a_{Luft}$ als konstant.

5. Simuliere in einem 500 Pixel breiten und 200 Pixel hohen Fenster den in dem folgenden Diagramm dargestellt Bewegungsvorgang eines Wagens auf einer Fahrbahn. Da der Bewegungsvorgang nur 2 Sekunden dauert, soll er im Sketch in Zeitlupe dargestellt werden (Faktor 10). Für den Maßstab der Strecke soll gelten: 10 cm ≙ 200 Pixel. Wenn der Wagen die rechte Seite des Fensters berührt, soll er stehenbleiben. Außerdem soll im Fenster die Geschwindigkeit des Wagens und in der Konsole die Geschwindigkeit und die Zeit angezeigt werden. Wenn du alles richtig gemacht hast, dann ist bei der Zeitlupendarstellung die Geschwindigkeit des Wagens bei 10 Sekunden Null.



6. Durch die unten angegebene Gleichung wird die Geschwindigkeit eines Körpers beschrieben, der sich geradlinig, aber mit a ≠ konstant bewegt. Simuliere die Bewegung des Körpers in einem 850 Pixel breiten und 200 Pixel hohen Fenster für den Zeitraum von 0 bis 5,0 Sekunden. Ein Pixel soll ein Zentimeter betragen. Der Körper (Rechteck 60 Pixel breit und 30 Pixel hoch) soll innerhalb dieser 5,0 Sekunden das Fenster nicht verlassen. Benutze, um dies zu erreichen, die Funktion *translate()* und eine möglichst hohe Bildwiederholungsrate. Die Bewegung des Körpers soll nach genau 5,0 Sekunden beendet werden. An welchem Ort befindet sich der Körper dann? Lasse diesen Wert im Fenster und in der Konsole anzeigen. Vergleiche den Wert mit dem Wert, den du mit Bleistift und Papier ausgerechnet hast.

$$v(t) = -\frac{3}{2} ms^{-3} t^2 + 6ms^{-2} t - 3ms^{-1}$$

Tipp: Bevor du mit dem Programmieren beginnst, muss du durch Integration die obige Gleichung v(t) in eine Gleichung für s(t) umwandeln. s_0 kann gleich Null gesetzt werden.

2.2 Wurfbewegungen

2.2.1 Senkrechter Wurf

Beim senkrechten Wurf nach oben werden wir Gelerntes wiederholen und natürlich auch etwas Neues lernen. Bei *void setup()* lassen wir den Hintergrund zeichnen und setzen die Bildwiederholungsrate auf 1. Damit erreichen wir, dass bei *void draw()* jede Sekunde unser Objekt gezeichnet wird, ohne dass der Hintergrund neu gezeichnet wird. Jedes gezeichnete Bild des Objektes bleibt damit im Fenster sichtbar. So können wir die beschleunigte Bewegung beim senkrechten Wurf besser beobachten (siehe Abb. 2.8).

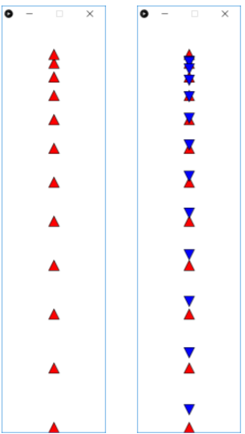


Abbildung 2.8: Senkrechter Wurf nach oben (links Aufstieg, rechts Aufstieg und Abstieg)

Beim senkrechten Wurf nach oben ist die Abwurfgeschwindigkeit v nach oben gerichtet und die Erdbeschleunigung g nach unten gerichtet. Für die Höhe h gilt $h(t) = h_0 + v_0 \cdot t - 0{,}5 \cdot g \cdot t^2$. In dem unten folgenden Sketch schreiben wir aber h = 800 – v0*t + 0.5*g*t*t. Warum $h_0 = 800$ und umgekehrte Vorzeichen? Nun die Antwort ist einfach. **Bei Processing zeigt die positive y-Achse nach unten.** Wenn wir also vom Boden unseres 800 Pixel hohen Fensters starten wollen, dann müssen wir für h_0 den Wert 800 eingeben. Wenn unser Körper, im Sketch in Form eines Dreieckes, an Höhe gewinnen will, dann müssen wir vom Wert 800 etwas abziehen. Also $-v_0 \cdot t$. Die Erdbeschleunigung g zeigt nach unten. Sie bremst den Körper ab und bewegt ihn wieder in Richtung Boden. Also $+0{,}5 \cdot g \cdot t^2$.

Im weiteren Verlauf des Sketches (s. u.) folgt eine if-Anweisung, die bewirkt, dass ein rotes Dreieck mit der Spitze nach oben gezeichnet wird, solange der Betrag von v_0 größer als der Betrag von $g \cdot t$ ist. Der Körper steigt solange nach oben, bis er vollständig abgebremst ist. Dann gilt $v_0 = g \cdot t_S \Rightarrow$

$$t_S = \frac{v_0}{g}$$

t_S steht für Steigzeit. Setzen wir die Werte aus unserem Sketch ein.

$$t_S = \frac{v_0}{g} = \frac{120\ ms^{-1}}{9{,}81\ ms^{-2}} \approx 12{,}23\ s$$

Die so berechnete Steigzeit können wir in unserer Animation überprüfen.

Neu im folgenden Sketch ist die Anweisung **else if**. Mit ihr können wir eine zweite Bedingung einfügen. Sie wird nur dann ausgeführt, wenn die erste Bedingung nicht erfüllt ist.

```
else if (v0 < g*t)
{
```

```
    fill(0, 0, 255);
    triangle(50, h+20, 60, h, 40, h);
  }
```

Wenn die Bedingung $v_0 < g \cdot t$ im obigen Sketchausschnitt wahr ist, dann soll ein blaues Dreieck mit einer Spitze nach unten gezeichnet werden.

Nun folgt der vollständige Sketch, in dem alles nochmal beschrieben wird. Es ist nicht verboten, den Sketch und damit die Simulation mit unterschiedlichen Werten von v_0 zu testen.

Sketch 06: senkrechter_Wurf-nach_oben

```
// Senkrechter Wurf nach oben

float x = 100; // x-Koordinate des Startortes
float h = 800; // y-Koordinate des Startortes
float v0 = 120; // Abwurfgeschwindigkeit
float g = 9.81; // Erdbeschleunigung
float t = 0; // Die Zeitzählung startet bei Null

void setup()
{
  size(200, 800); // Das Fenster ist 200 Pixel breit und 800 Pixel hoch.

  /* Die Anweisung für den Hintergrund steht bei void setup()
   damit die einzelnen Bewegungsschritte sichtbar bleiben */
  background (255);

  /* Damit wir die Wurfbewegung besser verfolgen können, wird eine
   geringe Bildwiederholungsrate eingestellt */
  frameRate(1); // Bei einer frameRate von 1 wird jede Sekunde ein Bild
                // gezeichnet
}

void draw()
{
  /* Da bei Processing die y-Achse nach unten zeigt,
   müssen die Vorzeichen von v und g entsprechend
   angepasst, d.h. umgekehrt werden */
  h = 800 - v0*t + 0.5*g*t*t;
  t = t + 1; // Pro Durchlauf wird t um 1 erhöht

  /* Wenn die folgende Bedingung wahr ist, wird ein rotes Dreieck
   mit einer Spitze nach oben gezeichnet */
  if (v0 > g*t)
  {
    fill(255, 0, 0);
    triangle(x, h-20, 110, h, 90, h);
  }

  /* Wenn die nun folgende Bedingung wahr ist, dann wird ein blaues
   Dreieck mit einer Spitze nach unten gezeichnet */
  else if (v0 < g*t)
  {
    fill(0, 0, 255);
    triangle(x, h+20, 110, h, 90, h);
  }
}
```

2.2.2 Waagerechter Wurf

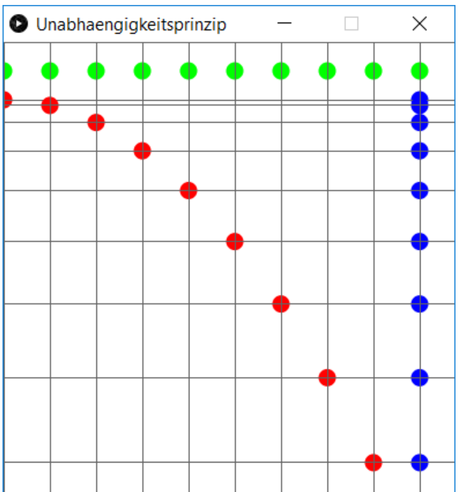


Abbildung 2.9: Unabhängigkeitsprinzip der Bewegung am Beispiel des waagerechten Wurfes

Anhand des waagerechten Wurfes lässt sich sehr schön das Unabhängigkeitsprinzip der Bewegung zeigen. In unserem Sketch startet eine rote Kugel, dargestellt durch einen roten Kreis, mit v_{01} = 40 ms^{-1} in x-Richtung. Bei ihrer Bewegung in x-Richtung bewirkt die Anziehungskraft der Erde, dass die rote Kugel konstant mit g = 9,81 ms^{-2} in y-Richtung beschleunigt wird.

Beim Start der roten Kugel beginnt von gleicher Starthöhe eine blaue Kugel ihren freien Fall und eine grüne Kugel startet mit der gleichen Geschwindigkeit wie die rote Kugel in x-Richtung. Die grüne Kugel bewegt sich jedoch in der Schwerelosigkeit. Die Bildwiederholungsrate beträgt ein Bild pro Sekunde.

Unser Sketch fertigt eine Stroboskopaufnahme der Bewegungen der drei Kugeln an und legt auf das Bild ein Gitternetz (siehe Abb. 2.9). Anhand dieses Bildes kann man sehr gut erkennen, dass die Bewegung der roten Kugel eine Überlagerung einer Bewegung mit v_x = konstant und a_y = konstant ist.

Hier ist nun der Sketch, der die obige Abbildung 2.9 erzeugt hat.

Sketch 07: Unabhaengigkeitsprinzip

```
// Unabhängigkeitsprinzip der Bewegung

float x1 = 0; // roter Kreis
float y01 = 50; // roter Kreis
float y1; // roter Kreis
float v01 = 40; // roter Kreis
float x2 = 360; // blauer Kreis
float y02 = 50; // blauer Kreis
float y2; // blauer Kreis
float x3 = 0; // grüner Kreis
float y3 = 25; // grüner Kreis
float g = 9.81; // Erdbeschleunigung
float t = 0;

void setup()
{
  size(390, 390);
```

```
  background (255);
  frameRate(1);
}

void draw()
{
  // Ein Linienraster wird gezeichnet
  stroke(100);
  strokeWeight(1);
  line(0, y1, 400, y1);
  line(x1, 0, x1, 400);

  // waagerechter Wurf (gleichzeitige Bewegung in x- und y-Richtung)
  x1 = v01*t; // Bewegung in x-Richtung
  y1 = y01 + 0.5*g*t*t; // Bewegung in y-Richtung
  noStroke();
  fill(255, 0, 0); // roter Kreis
  ellipse(x1, y1, 15, 15);

  // freier Fall (nur Bewegung in y-Richtung)
  y2 = y02 + 0.5*g*t*t;
  fill(0, 0, 255); // blauer Kreis
  ellipse(x2, y2, 15, 15);

  // Bewegung mit konstanter Geschwindigkeit nur in x-Richtung
  x3 = v01*t;
  fill(0, 255, 0);
  ellipse(x3, y3, 15, 15); // grüner Kreis

  t = t + 1;
}
```

2.2.3 Schräger Wurf

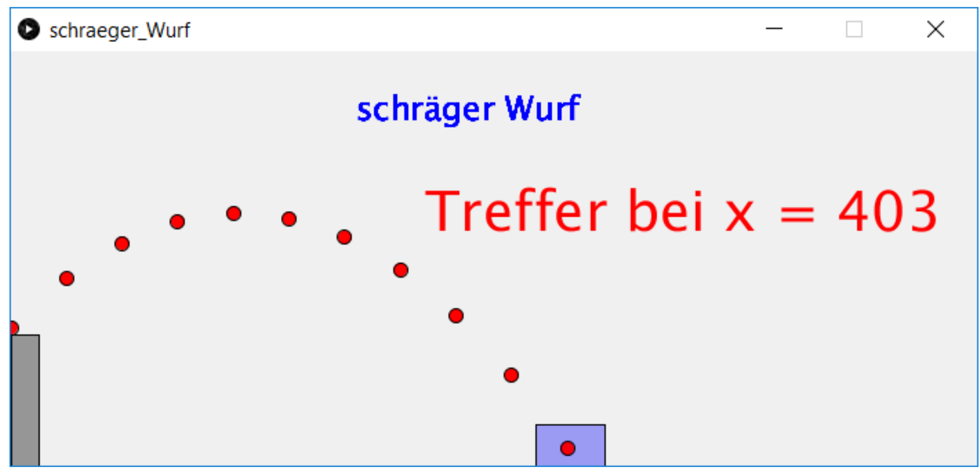


Abbildung 2.10: Schräger Wurf in einen Eimer

Der schräge Wurf ohne Luftreibung ist eine Kombination aus senkrechtem Wurf nach oben und waagerechtem Wurf. Den Vektor $\vec{s}$ können wir in seine x- und y-Komponente zerlegen (siehe Abb. 2.11). Dadurch können wir in den beiden letzten Übungen erlerntes Wissen wieder anwenden. Neu lernen müssen wir jedoch, wie man einen Winkelwert in Processing eingibt. Um die ganze Sache etwas anspruchsvoller zu machen, wollen wir einen roten Ball aus einer gewissen Höhe

abwerfen und ihn in einen am Boden stehenden Eimer befördern (siehe Abb. 2.10). Wenn wir den Eimer treffen, dann soll im Fenster „Treffer bei x = …“ erscheinen.

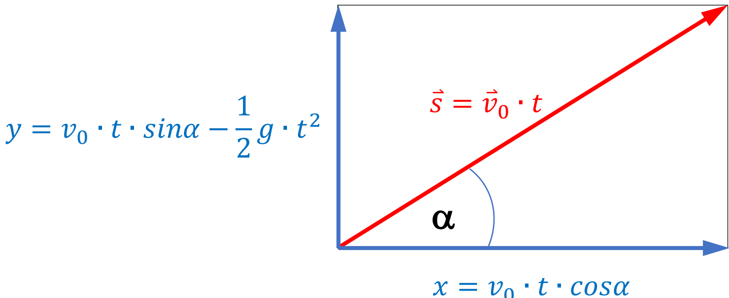


Abbildung 2.11: x- und y-Komponente beim schrägen Wurf

Widmen wir uns zuerst der Frage, wie man bei Processing Winkelwerte eingibt. Processing verlangt grundsätzlich Winkelangaben im Bogenmaß. Also 2π statt 360° oder $\frac{\pi}{2}$ statt 90°. Mit der Funktion **radians()** können wir jedoch Gradzahlwinkelwerte ins Bogenmaß umwandeln. Somit schreiben wir die Gleichungen von Abbildung 2.11 wie folgt.

```
x = v0*t*cos(radians(winkel));
y = - v0*t*sin(radians(winkel)) + 0.5*g*t*t;
```

Die veränderten Vorzeichen in der Funktion für y sind dem Umstand geschuldet, dass, wie schon erwähnt, bei Processing die positive y-Achse nach unten zeigt. Erinnern wir uns an den Sketch zum senkrechten Wurf nach oben.
Da wir den Ball ja nicht vom Boden aus, sondern in einer gewissen Höhe abwerfen wollen, müssen wir die Gleichung für y noch um die Größe y_0 erweitern.

```
y = y0 - v0*t*sin(radians(winkel)) + 0.5*g*t*t
```

Nachdem die Winkelfrage geklärt ist, können wir uns dem Eimer widmen. Wie können wir in unserem Sketch dafür sorgen, dass im Fenster „Treffer bei x = …“ anzeigt wird, wenn der Ball im Eimer landet. Du wirst es wahrscheinlich schon erraten haben. Natürlich mit einer logischen Operation. Setzen wir den Eimer zum Beispiel auf den Boden (y = 300) an die Stelle x = 380 und machen ihn 50 Pixel breit und 30 Pixel hoch (siehe Abb. 2.12).

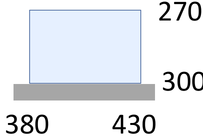


Abbildung 2.12: Eimer auf dem Boden

Der Ball befindet sich im Eimer, wenn seine x- und y-Werte innerhalb der in Abbildung 2.12 aufgeführten Koordinaten liegen. Da ein Ball kein Punkt ist, müssen wir berücksichtigen, dass er in unserem Sketch einen Durchmesser von 10 Pixel hat. Somit lautet unsere logische Operation:

```
if (x > 385 && x < 425 && y > 265 && y < 305)
{
  fill(255, 0, 0);
  textSize(40);
  text("Treffer bei x = " +round(x), 300, 130 );
}
```

Nachdem wir im *Sketch Autorennen* schon den logischen Operator **||** für **ODER** kennengelernt haben, benutzen wir nun den logischen Operator **&&** für **UND**. Damit dürfte die Bedingung in der obigen if-Anweisung hinreichend erklärt sein. An dieser Stelle möchte ich dann auch schon auf einen dritten und einen vierten logischen Operator hinweisen. Der Operator **!** steht für die logische Negation **NICHT** und der Operator **^** für das **exklusive ODER**.

Erläutert werden muss jetzt noch die Funktion **round()**. Sie rundet den x-Wert auf ganzzahlige Werte. Somit verhindern wir, dass bei einem Treffer im Fenster die Nachkommastellen angezeigt werden.

Damit wir sehen, ob sich der rote Ball auch im Eimer befindet, haben wir den Eimer transparent gezeichnet. Wie gelingt dies? Wenn man einen blauen Eimer zeichnen will, dann schreibt man fill(0, 0, 255). **Mit einer vierten Zahl können wir die Transparenz einer Farbe einstellen.**

```
fill(0, 0, 255, 10);
```

Je kleiner die Zahl, desto größer ist die Transparenz. D.h., desto geringer ist ihre Deckkraft. 0 bedeutet 0% Deckkraft und 255 bedeutet 100% Deckkraft.

Damit haben wir alles erklärt und können uns nun den vollständigen Sketch anschauen. Aber nicht vergessen!!! Den Eimer verschieben und mit Abwurfgeschwindigkeit und Abwurfwinkel spielen. Bei welchem Winkel erreicht man die größte Wurfweite? Gibt es nur eine Kombination von Geschwindigkeit und Winkel, um den Eimer zu treffen?

Sketch 08: schraeger_Wurf

```
// Schräger Wurf

float x = 0; // x-Wert für den Mittelpunkt des Balls
float y0 = 200; // Abwurfhöhe
float y = 0; // y-Wert für den Mittelpunkt des Balls
float v0 = 57; // Abwurfgeschwindigkeit
float g = 9.81; // Erdbeschleunigung
float t = 0; // Zeit
float winkel = 45; // Winkel in Grad

void setup()
{
  size(700, 300);
  background (240);
  frameRate(2); // Zwei Durchläufe pro Sekunde
}

void draw()
{
  fill(0, 0, 255); // Schriftfarbe
  textSize(24); // Schriftgröße
  text("schräger Wurf", 250, 50);

  x = v0*t*cos(radians(winkel)); // Bewegung in x-Richtung
  y = y0 - v0*t*sin(radians(winkel)) + 0.5*g*t*t; // Bewegung in
                                                   // y-Richtung
  t = t + 1;

  fill(255, 0, 0);
```

```
  ellipse(x, y, 10, 10); // roter Ball

  fill(150);
  rect(0, y0 + 5, 20, 300); // graue Abschussrampe

  fill(0, 0, 255, 10);
  rect(380, 270, 50, 30); // blauer, transparenter Eimer

  // Angabe ob der Eimer getroffen wurde oder nicht getroffen wurde
  if (x > 385 && x < 425 && y > 265 && y < 305)
  {
    fill(255, 0, 0);
    textSize(40);
    text("Treffer bei x = " +round(x), 300, 130 );
  }
}
```

2.2.4 Zusammenfassung

Fassen wir zusammen, was wir in Kapitel 2.2 *Wurfbewegungen* gelernt haben, bzw. gelernt haben sollten.

else if() Mit *else if* wird eine Bedingung nur dann geprüft, wenn die erste Bedingung nicht erfüllt ist.

radians() Processing verlangt grundsätzlich Winkelangaben im Bogenmaß. Mit der der Funktion *radians()* kann man Gradzahlwinkelwerte ins Bogenmaß umwandeln.

logische Operatoren **||** ODER **&&** UND **!** NICHT **^** exklusives ODER

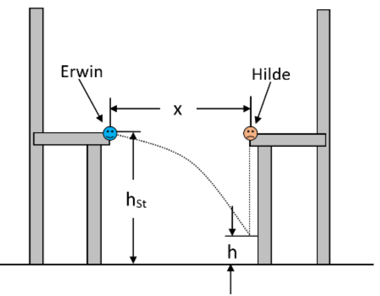
round() *round()* rundet eine Zahl mit Nachkommastellen auf einen ganzzahligen Wert.

spielen Wenn man erfolgreich einen Physik-Sketch geschrieben hat, dann sollte man auch mit den Variablen spielen, um möglichst viel Physik zu lernen.

2.2.5 Aufgaben

1. Ändere den Sketch *schraeger_Wurf* so um, dass der Eimer von links nach rechts durch das Anzeigefenster wandert und er für jegliche Winkel zwischen 0° und 80° den Ball fängt. Die Funktion *background(240)* soll hierzu von *void setup()* nach *void draw()* verschoben werden. Wenn der Eimer getroffen wird, dann soll im Fenster der gerundete x-Wert des Balls angezeigt werden und der Eimer stehenbleiben. Benutze dazu als letzte Zeile in *void draw()* die Funktion *noLoop().*

2. Die beiden Flöhe Erwin und Hilde sitzen sich im Abstand vom x = 150 mm auf zwei Stuhlkanten (h_{St} = 460 mm) verliebt gegenüber. Plötzlich wird Hilde vor lauter Liebe ohnmächtig und stürzt von der Kante. Erwin springt im gleichen Moment mit v_x = 500 mms^{-1} in x-Richtung von seiner Stuhlkante.

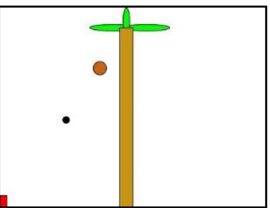
   

   In welcher Höhe über dem Fußboden kann Erwin seine Hilde schnappen und sich mit ihr am Stuhlbein festhalten?

   Löse die Aufgabe zuerst mit Bleistift und Papier. Schreibe anschließend einen Sketch, der diesen Vorgang simuliert. Sobald Erwin seine Hilde geschnappt und sich mit ihr am Stuhlbein festgehalten hat, soll die Fanghöhe h als gerundeter, ganzzahliger Wert im Fenster angezeigt werden. Da der Bewegungsvorgang sehr schnell abläuft, soll er in Zeitlupe dargestellt werden. Anstelle der Stühle können Rechtecke gezeichnet werden. g = 9, 81 ms^{-2} = 9810 mms^{-2}

   Tipp: Wähle eine möglichst hohe Bildwiederholungsrate. Dadurch wird der Bewegungsablauf nicht nur in Zeitlupe dargestellt, sondern so erhältst du auch ein genaueres Ergebnis, da sich bei jedem Durchlauf von *void draw()* x und y sich nur in kleinen Schritten ändern. Dadurch kannst du das Intervall „Erwin trifft Hilde“ sehr klein machen.

3. Simuliere den folgenden Vorgang in einem 400 Pixel breiten und 300 Pixel hohen Fenster. Ein Sportschütze zielt mit seinem Luftgewehr genau auf eine Kokosnuss, die sich in 25,0 m Höhe befindet. Der horizontale Abstand der Gewehrmündung zur Kokosnuss beträgt 15,0 m. In dem Moment, als die Kugel in einer Höhe von 1,7 m mit einer Geschwindigkeit von v = 200 ms^{-1} das Gewehr verlässt, löst sich die Kokosnuss vom Baum. Trotzdem trifft die Kugel die Kokosnuss. Der Einfluss der Luftreibung soll nicht berücksichtigt werden. Schreibe in die Kopfzeile deines Sketches eine Begründung dafür, warum die Kugel die fallende Kokosnuss trifft, obwohl auf die ruhende gezielt wurde.

   1 Pixel ≙ 1 dm; g = 9,81 ms^{-2} = 98,1 dms^{-2}

   Tipp: Benutze die Anweisung *else if()*.

2.3 Impulserhaltung

2.3.1 Unelastischer Stoß

Beim zentralen, geraden, vollkommen unelastischen Stoß zweier Körper bewegen sich die Körper nach dem Stoß gemeinsam mit der Geschwindigkeit u weiter. Es gilt:

$$m_1 \cdot v_1 + m_2 \cdot v_2 = (m_1 + m_2) \cdot u \implies u = \frac{m_1 \cdot v_1 + m_2 \cdot v_2}{m_1 + m_2}$$

Aufgrund der bisher erworbenen Kenntnisse dürfte die Programmierung eines entsprechenden Sketches uns nicht vor allzu große Probleme stellen. Als echte Physiker wollen wir mit unserer

Simulation aber auch spielen. Deshalb richten wir sie so ein, dass wir die Massen m_1 und m_2 sowie deren Geschwindigkeiten v_1 und v_2 frei wählen können. Wenn eine Masse doppelt so groß ist wie die andere, dann soll sie im Fenster auch doppelt so groß dargestellt werden (siehe Abb. 2.13).

Bevor du die jeweilige Simulation mit unterschiedlichen Massen und Geschwindigkeiten startest, versuche bitte vorauszusagen, wohin sich die Körper nach dem Stoß bewegen. Es ist auch eine gute Übung, wenn du die Geschwindigkeit u mit Bleistift und Papier ausrechnest und dein Ergebnis mit der Angabe in der Konsole vergleichst.

Abbildung 2.13: Unelastischer Stoß zweier Körper

Da sich die beiden Körper zuerst getrennt voneinander mit den Geschwindigkeiten v_1 und v_2 und nach dem Stoß gemeinsam mit der Geschwindigkeit u bewegen, müssen wir eine Fallunterscheidung vornehmen. Dies gelingt am besten mit zwei if-Anweisungen (siehe unten).

Neu im dem Sketch ist die Anweisung **textAlign(CENTER)**. Da align mit ausrichten übersetzt werden kann, erklärt sich diese Bedingung fast von selbst. Man muss nur wissen, dass mit *CENTER* nicht die Mitte des Fensters gemeint ist, sondern die Mitte des Textes. Die Mitte des Textes wird auf einen selbst zu bestimmenden x-Wert gesetzt.

```
  textAlign(CENTER); // Setzt die Mitte des Textes auf den unten
                     // gewählten x-Wert (200).
  text("vollkommen unelastischer Stoß", 200, 40);
```

Sketch 09: unelastischer_Stoss

```
// vollkommen unelastischer Stoß

float x1 = 0; // Ortskoordinate des gelben Körpers vor dem Start
float x2 = 350; // Ortskoordinate des roten Körpers vor dem Start
float v1 = 1; // Geschwindigkeit des gelben Körpers vor dem Stoß
float v2 = -1; // Geschwindigkeit des roten Körpers vor dem Stoß
float u1; // Geschwindigkeit des gelben Körpers nach dem Stoß
float u2; // Geschwindigkeit des roten Körpers nach dem Stoß
float m1 = 1; // Masse des gelben Körpers
float m2 = 1; // Masse des roten Körpers
float t = 1; // Zeiteinheit pro Frame

void setup()
{
  size(400, 200); // Fenstergröße
}

void draw()
{
```

```
  background(255); // Hintergrundfarbe

  fill(0, 0, 255); // Textfarbe
  textSize(24); // Textgröße
  textAlign(CENTER); // Setzt die Mitte des Textes auf den unten
                     // gewählten x-Wert (200).
  text("vollkommen unelastischer Stoß", 200, 40);

  fill(255, 255, 0); //gelber Körper
  rect(x1, 200 - m1*20, 50, m1*20); // Die Höhe des gelben Körpers
                                    // wächst mit seiner Masse.

  fill(255, 0, 0);// roter Körper
  rect(x2, 200 - m2*20, 50, m2*20); // Die Höhe des roten Körpers wächst
                                    // mit seiner Masse.

  if (x1 <= x2 - 50)
  {
    /* Nur solange die if-Bedingung erfüllt ist, nimmt x1 um das gleiche
       Streckenstück v1*t zu und x2 um das gleiche Streckenstück v2+t
       ab, da v2 negativ ist. */
    x1 = x1 + v1*t; // Ortskoordinate von Gelb vor dem Stoß
    x2 = x2 + v2*t; // Ortskoordinate von Rot vor dem Stoß
  }

  if (x1 >= x2 - 51)
  {
    u1 = (m1*v1+m2*v2)/(m1+m2); // Geschwindigkeit des gelben Körpers
                                // nach dem Stoß
    u2 = u1; // Da m1 und m2 nach dem Stoß eine Einheit bilden, gilt
             // u1 = u2.

    x1 = x1 + u1*t; // Ortskoordinate des gelben Körpers nach dem Stoß
    x2 = x2 + u2*t; // Ortskoordinate des roten Körpers nach dem Stoß

    println("u1 = u2 = ", u1); // Die Werte für u1 und u1 werden in der
                               // Konsole angezeigt.
  }
}
```

2.3.2 Elastischer Stoß

Die Gleichungen für die beiden Geschwindigkeiten u_1 und u_2 der Körper nach dem zentralen, geraden, vollkommen elastischen Stoß erhält man aus den Kombination von Impuls- und Energieerhaltungssatz. Sie lauten:

$$u_1 = \frac{2m_2 \cdot v_2 + (m_1 - m_2) \cdot v_1}{m_1 + m_2} \quad \text{und} \quad u_2 = \frac{2m_1 \cdot v_1 + (m_2 - m_1) \cdot v_2}{m_1 + m_2}$$

Abbildung 2.14: Vollkommen elastischer Stoß

Während sich beim unelastischen Stoß die beiden Körper nach dem Stoß gemeinsam mit nur einer Geschwindigkeit bewegen, trennen sich beim elastischen Stoß die beiden Körper nach dem Stoßprozess wieder und bewegen sich mit unterschiedlichen Geschwindigkeiten weiter. Diese Geschwindigkeitsänderungen berücksichtigen wir in unserem Sketch innerhalb der if-Anweisung ganz einfach mit v1 = u1 und v2 = u2. Siehe hierzu die Erläuterungen im Sketch.

Auch bei dieser Simulation gilt: Erst abschätzen wie sich die beiden Körper nach dem Stoß verhalten, dann u_1 und u_2 mit Bleistift und Papier ausrechnen und erst danach die Simulation starten. So freut man sich auch, wenn die Werte in der Konsole die eigenen Überlegungen bestätigen.

Sketch 10: elastischer Stoß

```
// vollkommen elastischer Stoß

float x1; // Ortskoordinate des gelben Körpers
float x2 = 550; // Ortskoordinate des roten Körpers
float v1 = 1; // Geschwindigkeit des gelben Körpers
float v2 = -1; // Geschwindigkeit des roten Körpers
float u1; // Geschwindigkeit des gelben Körpers nach dem Stoß
float u2; // Geschwindigkeit des roten Körpers nach dem Stoß
float m1 = 1; // Masse des gelben Körpers
float m2 = 1; // Masse des roten Körpers
float t = 1; // Zeit

void setup()
{
  size(600, 200); // Fenstergröße
}

void draw()
{
  background(255); // Hintergrundfarbe
  fill(0, 0, 255); // Schriftfarbe
  textSize(32); // Schriftgröße
  textAlign(CENTER); // Die Mitte des Textes wird auf den x-Wert 300
                     // gesetzt
  text("vollkommen elastischer Stoß", 300, 50);

  fill(255, 255, 0); // gelber Körper
  rect(x1, 200 - m1*20, 50, m1*20);

  fill(255, 0, 0);// roter Körper
  rect(x2, 200 - m2*20, 50, m2*20);
```

```
  x1 = x1 + v1 * t; // Ortskoordinate des gelben Körpers zur Zeit t
  x2 = x2 + v2 * t; // Ortskoordinate des roten Körpers zur Zeit t

  if (x1 >= x2 - 50 && v1 - v2 > 0) /* Bei der Bedingung v1 - v2 > 0
  bewegen sich die beiden Körper aufeinander zu und bei x1 >= x2
  berühren sie sich. */
  {
    u1 = (2*m2*(v2)+(m1-m2)*v1)/(m1+m2); // Geschwindigkeit von Gelb
                                        // nach dem Stoß wird berechnet
    u2 = (2*m1*v1+(m2-m1)*(v2))/(m1+m2); // Geschwindigkeit von Rot nach
                                         // dem Stoß wird berechnet

    v1 = u1; // v1 nimmt den Wert von u1 an
    v2 = u2; // v2 nimmt den Wert von u2 an

    println("Gelb u1 =", u1, "     ", "Rot u2 =", u2);
  }
}
```

2.3.3 Zusammenfassung

Fassen wir zusammen, was wir in Kapitel 2.3 *Impulserhaltung* gelernt haben, bzw. gelernt haben sollten.

textAlign(CENTER)	Mit textAlign(CENTER) kann man die Mitte eines Textes auf einen gewünschten x-Wert gesetzen.
Bleistift und Papier	Vor dem Programmieren die physikalischen Werte mit Bleistift und Papier ausrechnen und nach der Programmierung mit den Werten in der Konsole von Processing vergleichen.

2.3.4 Aufgaben

1. Untersuche mit dem Sketch *unelastischer_Stoss* die folgenden Vorgänge. Berechne aber vor dem Start der jeweiligen Simulation die Geschwindigkeit der beiden Körper nach dem Stoß mit Bleistift und Papier und vergleiche deine Ergebnisse mit den Werten in der Konsole.

   a) Der gelbe Körper hat die doppelte Masse des roten Körpers und bewegt sich von links nach rechts. Der rote Körper hat den doppelten Geschwindigkeitsbetrag des gelben Körpers. Er bewegt sich von rechts nach links. Wie bewegen sich die beiden Körper nach dem Stoßprozess?

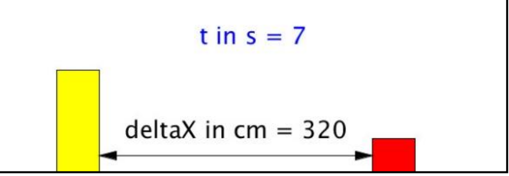
   b) Beim Start der Simulation startet der rote Körper bei x_2 = 100 Pixel. Er bewegt sich mit einer Geschwindigkeit von v_2 = 0,5 von links nach rechts. Der gelbe Körper startet bei x_1 = 0 Pixel und bewegt sich ebenfalls von links nach rechts. Seine Geschwindigkeit beträgt v_1 = 1,0. Beide Körper haben die gleiche Masse. Mit welcher Geschwindigkeit bewegen sie sich nach dem Stoß?

2. Untersuche mit dem Sketch *elastischer_Stoss* die folgenden Vorgänge. Berechne aber vor dem Start der jeweiligen Simulation die Geschwindigkeiten der beiden Körper nach dem Stoß mit Bleistift und Papier und vergleiche deine Ergebnisse mit den Werten in der Konsole.

   a) Der gelbe Körper hat die doppelte Masse des roten Körpers und bewegt sich von links nach rechts. Der rote Körper hat jedoch den doppelten Geschwindigkeitsbetrag des gelben Körpers und bewegt sich von rechts nach links. Wie bewegen sich die beiden Körper nach dem Stoßprozess?

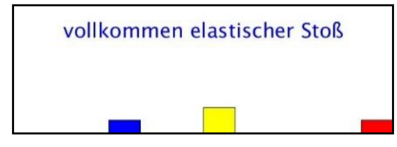
   b) Der rote Körper ruht bei x_2 = 250 Pixel. Der gelbe hat die dreifache Masse des roten Körpers und bewegt sich mit v = 1,0 auf den roten Körper zu. Welche Geschwindigkeit besitzen die beiden Körper nach dem Stoß?

3. Ein gelber Körper (m_1 = 6 kg) und roter Körper (m_2 = 2 kg) stehen auf einer 600 cm langen Fahrbahn und berühren sich bei x = 200 cm. Nach genau 3,0 Sekunden stoßen sie sich gegenseitig voneinander ab. Wie weit sind sie nach 7,0 Sekunden voneinander entfernt, wenn der gelbe Körper sich von rechts nach links mit v_1 = -20 cms^{-1} bewegt? Löse diese Aufgabe zuerst mit Bleistift und Papier.

   Die beiden Körper können, wie auch bei den vorhergehenden Aufgaben, im Fenster als Rechtecke dargestellt werden. Im Fenster soll die Zeit von 0 bis 7 Sekunden ganzzahlig angezeigt werden. Nach 7 Sekunden soll zusätzlich zur der Zeit auch der formelmäßig im Sketch berechnete Abstand der beiden Körper in ganzzahligen Zentimeterwerten angezeigt werden (siehe Abbildung) und die Simulation angehalten werden. Wähle eine hohe Bildwiederholungsrate, um genaue Ergebnisse zu erhalten.

4. Wagen 3 (blau, m = 1 kg) ruht bei x = 150 cm und Wagen 2 (gelb, m = 2 kg) ruht bei x = 300 cm. Von rechts kommend stößt Wagen 1 (rot, m = 1 kg, v_3 = -1 cms^{-1}) vollkommen elastisch gegen Wagen 2 und anschließend stößt Wagen 2 vollkommen elastisch gegen Wagen 3. Mit welcher Geschwindigkeit bewegt sich Wagen 3 von rechts nach links?

   Berechne die Geschwindigkeiten von Wagen 1, 2 und 3 nach dem Stoß von Wagen 2 mit Wagen 3 mittels Bleistift und Papier und simuliere diesen Vorgang in einem 600 Pixel breiten Fenster (1 Pixel ≙ 1 cm). Lasse dir die Geschwindigkeiten von Wagen 1, 2 und 3 nach dem Stoß von Wagen 2 mit Wagen 3 in der Konsole anzeigen und überprüfe die Richtigkeit deiner Programmierung mithilfe des Energieerhaltungssatzes.

2.4 Kreisbewegungen

2.4.1 Bewegungen von Himmelskörpern

Sonne, Planet und Mond

Nachdem wir uns in dem vorangegangenen Kapitel mit Simulationen zur Translation beschäftigt haben, wollen wir nun einige Sketche schreiben, die Kreisbewegungen zum Thema haben. In unserem ersten Sketch werden wir einen Planeten mit seinem Mond um eine Sonne kreisen lassen (siehe Abbildung 2.15). Diese Simulation machen wir rein zur Veranschaulichung, d.h., noch ohne Gravitationsgesetz. Dieses behandeln wir im Kapitel Felder.

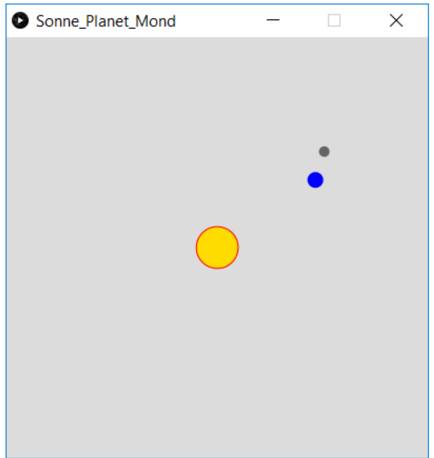


Abbildung 2.15: Abbildung zum Sketch Sonne_Planet_Mond

Um etwas auf einem Kreis zu bewegen, benötigen wir die Funktion ***rotate()*** mit der Angabe des Winkels im Bogenmaß. Will man den Winkel in Grad eingeben, dann schreibt man **rotate(radians())**. *radians()* rechnet den Winkelwert, den man in die Klammer schreibt, in das Bogenmaß um. Mit *rotate()* dreht sich das Koordinatensystem um seinen Ursprung im Uhrzeigersinn. Alles, was wir nach dem Aufruf von *rotate()* zeichnen, dreht sich mit. Da bei Processing der Ursprung des Koordinatensystems links oben in der Ecke des Fensters sitzt, würde unser Planet mit seinem Mond immer wieder aus dem Fenster verschwinden. Wie lösen wir dieses Problem?

Mit der Anweisung ***translate()*** können wir den Koordinatenursprung innerhalb des Fensters verschieben. In unserem Sketch verschieben wir mit *translate(200, 200)* den Ursprung in die Mitte unseres 400 x 400 Pixel großen Fensters. Alles, was wir nun zeichnen, wird jetzt auch zur Mitte hin verschoben. Wenn wir für die Sonne *ellipse(0, 0, 40, 40)* schreiben, dann liegt ihr Mittelpunkt nicht links oben in der Fensterecke, sondern genau in der Mitte des Fensters und der Planet ist mit *ellipse(80, 80, 15, 15)* je 80 Pixel von der Mitte entfernt.

Der Mond soll mit dem Planeten um die Sonne kreisen und dabei gleichzeitig den Planeten umrunden. Wie gelingt dies? Wir verschieben das Koordinatensystem ein zweites Mal mit *translate(80, 80)* genau in den Mittelpunkt des Planeten. Mit *rotate(w2)* geben wir dem Mond einen eigenen Winkel, der sich bei jedem Durchlauf von *void draw()* entsprechend unserer Vorgabe ändert.

Nun, dies war doch alles gar nicht so schwierig. An dieser Stelle möchte ich dem Leser auch mal einen Blick in die Referenz von Processing empfehlen. Was steht hier zu den Anweisungen *rotate()* und *translate()?*

Sketch 11: Sonne_Planet_Mond

```
// Sonne Planet Mond

float w1; // Winkel 1
float w2; // Winkel 2

void setup()
{
  size(400, 400);
}

void draw()
{
  background(220);

  translate(200, 200); // Verschiebung des Koordinatensystems in die
                       // Fenstermitte
  rotate(w1); // Rotation des Koordinatensystems um den Winkel w1

  // Die Sonne sitzt im Ursprung des verschobenen Koordinatensystems
  stroke(255, 0, 0);
  strokeWeight(1);
  fill(255, 220, 0);
  ellipse(0, 0, 40, 40);

  // Der Planet rotiert mit dem Winkel w1 um die Sonne
  noStroke();
  fill(0, 0, 255);
  ellipse(80, 80, 15, 15);

  // Der Ursprung des mit w1 rotierenden Koordinatensystems wird nun in
  // den Mittelpunkt des Planeten verschoben
  translate(80, 80);
  rotate(w2); // Das so verschobene Koordinatensystem rotiert nun
              // zusätzlich noch mit dem Winkel w2

  // Der Mond rotiert mit dem Winkel w2 um den Planeten und gemeinsam
  // mit ihm mit w1 um die Sonne
  noStroke();
  fill(100);
  ellipse(20, 20, 10, 10);

  w1 = w1 + 0.01; // Winkel w1
  w2 = w2 + 0.05; // Winkel w2
}
```

Der Mond umkreist seinen Planeten

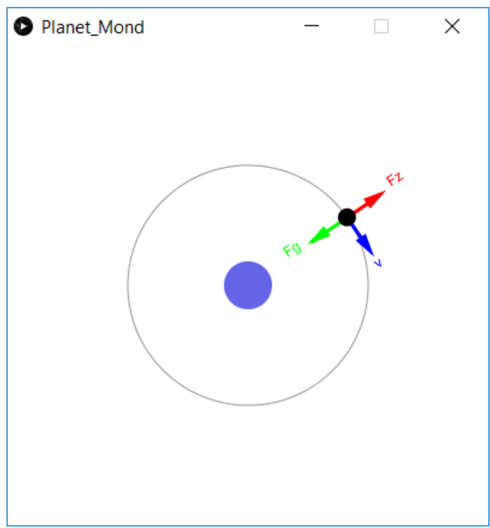


Abbildung 2.16: Der Mond umkreist seinen Planeten

Mit *rotate()* können wir also den ganzen Fensterinhalt rotieren lassen. Nutzen wir dies doch einmal, um den Mond, während er um seinen Planeten kreist, mit Vektorpfeilen für die Gravitationskraft F_g, die Zentrifugalkraft F_z und die Geschwindigkeit v zu versehen. Leider gibt es bei Processing keine Bildvorlage für Vektorpfeile. Diese müssen wir uns aus einer Linie und einem Dreieck selber basteln. Ein Dreieck zeichnet man mit ***triangle(x₁, y₁, x₂, y₂, x₃, y₃)***. Die richtigen Koordinatenpaare zu finden ist nicht so einfach, wenn der Vektor schräg steht. Leichter geht es, wenn der Vektor horizontal oder senkrecht steht. Aus diesem Grund positionieren wir den Mond so, dass er beim Start des Sketches so steht wie in Abbildung 2.17. Zum Beispiel hat das Dreieck für den roten Vektorpfeil in unserem Sketch die Koordinaten *triangle(120, -5, 140, 0, 120, 5)*. Man erkennt, dass es auf der mittels *translate()* verschobenen x-Achse liegt. Die anderen Dreiecke lassen sich aufgrund der Lage auch so einfach zeichnen.

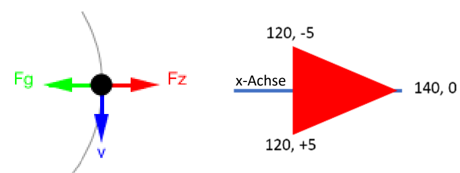


Abbildung 2.17: Lage des Mondes mit seinen Vektoren

Der nun folgende Sketch ist leicht zu verstehen. Deshalb können wir auf weitere Erklärungen verzichten.

Sketch 12: Planet_Mond

```
// Planet Mond

float w; // Winkel

void setup()
{
  size(400, 400);
}
```

```
void draw()
{
  background(255);

  translate(200, 200); // Verschiebung des Koordinatensystems in die
                       // Fenstermitte
  rotate(w); // Rotation des Koordinatensystems um den Winkel w

  // Erde
  fill(100, 100, 230);
  ellipse(0, 0, 40, 40);

  // Umlaufbahn
  stroke(150);
  strokeWeight(1);
  noFill();
  ellipse(0, 0, 200, 200);

  // Linien für die Vektoren
  stroke(0, 255, 0);
  strokeWeight(3);
  line(65, 0, 100, 0); // Fg
  stroke(0, 0, 255);

  stroke(255, 0, 0);
  strokeWeight(3);
  line(130, 0, 100, 0); // Fz

  stroke(0, 0, 255);
  strokeWeight(3);
  line(100, 0, 100, 20); // v

  // Spitzen (Dreiecke) und Bezeichnungen für die Vektoren
  noStroke();
  fill(255, 0, 0);
  triangle(120, -5, 140, 0, 120, 5); // Fz
  text("Fz", 145, 0);

  fill(0, 255, 0);
  triangle(80, -5, 60, 0, 80, 5); // Fg
  text("Fg", 40, 0);

  fill(0, 0, 255);
  triangle(105, 20, 100, 40, 95, 20); // v
  text("v", 97, 50);

  // Mond
  fill(0);
  ellipse(100, 0, 15, 15);

  w = w + 0.01; // Winkelzuwachs im Bogenmaß
}
```

2.4.2 Uhr mit pushMatrix und popMatrix

Wir wollen nun in Processing eine Uhr bauen, deren Zeiger unterschiedlich schnell rotieren (siehe Abb. 2.16). Als Erstes zeichnen wir die beiden Zeiger. Wie oben schon erwähnt, machen wir es uns

einfach und legen beide Zeiger auf die x-Achse und lassen sie im Ursprung des Koordinatensystems beginnen. Danach verschieben wir mit *translate()* den Ursprung in die Mitte des Fensters.

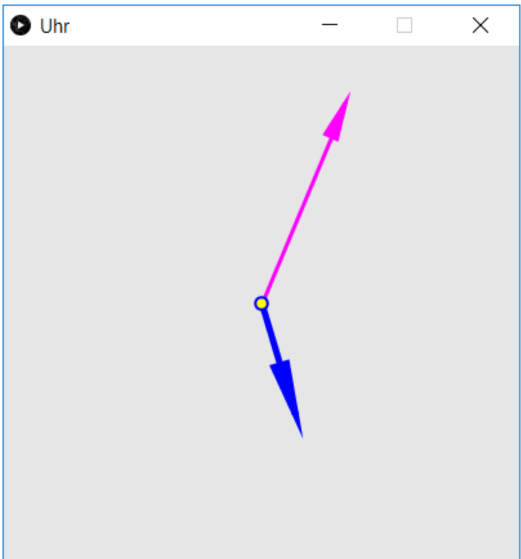


Abbildung 2.16: Uhr

Das Hauptproblem bei diesem Sketch ist, dass die beiden Zeiger unabhängig voneinander mit unterschiedlicher Geschwindigkeit rotieren sollen. Die Betonung liegt hier auf dem Wort unabhängig. Denn wenn wir in *void draw()* zweimal hintereinander *rotate()* eingeben, dann rotieren die beiden Zeiger zwar mit unterschiedlichen Geschwindigkeiten, aber nicht unabhängig voneinander. Die beiden Rotationgeschwindigkeiten addieren sich einfach. Um unser Ziel der unabhängigen Rotation zu erreichen, müssen wir etwas Neues lernen. Dies sind die beiden Funktionen **pushMatrix()** und **popMatrix().**

***pushMatrix()* speichert das aktuelle Koordinatensystem und *popMatrix()* lädt das gespeicherte Koordinatensystem wieder und vergisst alles, was zwischen *pushMatrix()* und *popMatrix()* gestanden hat.** Somit addiert sich in unserem Sketch die Rotation des blauen Zeigers nicht zur Rotation des violetten Zeigers.

Bei *rotate()* gibt man den Winkel im Bogenmaß oder mittels *radians()* im Gradwinkel ein. Wir geben ihn diesmal wieder im Bogenmaß ein. Doch mit der Besonderheit, dass wir ihn als ein Vielfaches von $\frac{\pi}{60}$ eingeben. Dies ist ein sehr feines und gut zu regulierendes Schrittmaß. Für den violetten Zeiger gilt *rotate(a*PI/60)* mit *a = a + 0.4* und für den blauen Zeiger gilt *rotate(b*PI/60)* mit *b = b + 0.1*. D.h., der violette Zeiger dreht sich viermal schneller als der blaue Zeiger.

Sketch 13: Uhr

```
// Uhr

float a; // Rotationfaktor für den violetten Zeiger
float b; // Rotationsfaktor für den blauen Zeiger

void setup()
{
  size(400, 400);
  smooth(2); // Linien und Dreiecke werden geglättet
}
```

```
void draw()
{
  background(230);
  translate(200, 200); // Koordinatenursprung wird in die Fenstermitte
                       // verschoben

  pushMatrix(); // Speichert die obige Transformation

  // violetter Pfeil
  rotate(a*PI/60);
  a = a + 0.4; // Rotationsgeschwindigkeit wird festgelegt

  stroke(255, 0, 255);
  strokeWeight(3);
  line(0, 0, 150, 0);
  fill(255, 0, 255);
  triangle(140, -5, 170, 0, 140, 5); // Zeigerspitze

  popMatrix(); /* Lädt die oben gespeicherte Transformation wieder und
                  vergisst, was zwischen pushMatrix und popMatrix steht.
   Somit addiert sich die folgende Rotation nicht zur obigen Rotation */

  // blauer Pfeil
  rotate(b*PI/60.0);
  b = b + 0.1; // Rotationsgeschwindigkeit wird festgelegt

  stroke(0, 0, 255);
  strokeWeight(5);
  line(0, 0, 70, 0);
  fill(0, 0, 255);
  triangle(50, -5, 90, 0, 50, 5); // Zeigerspitze

  // Kreis über dem Mittelpunkt der Uhr
  strokeWeight(2);
  fill(255, 255, 0);
  ellipse(0, 0, 10, 10);
}
```

2.4.3 Corioliskraft

Die Corioliskraft sorgt bei vielen Menschen für Kopfzerbrechen. Sie ist ja schließlich nur eine Scheinkraft und ob sie existiert oder nicht, hängt von dem Bezugsystem ab, in dem sich der Beobachter befindet. Mit unserem Sketch wollen wir hier ein wenig Klarheit schaffen.

Am Nordpol stehen zwei Forscher. Herr Rot steht genau auf dem Nordpol und Herr Blau ein Stück weiter weg. Plötzlich wirft Herr Rot eine faule Tomate in Richtung Süden und trifft Herrn Blau. Natürlich kommt es zum Streit.

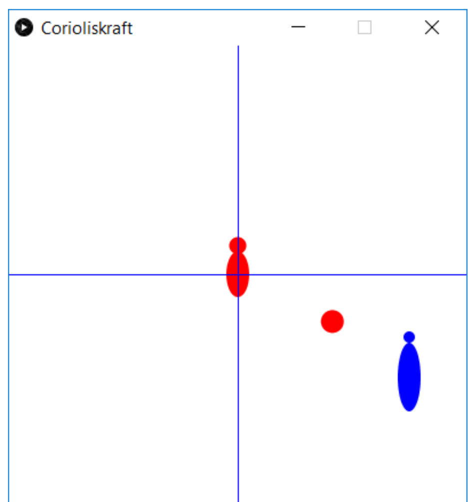


Abbildung 2.17: Tomatenwurf aus Sicht von Herrn Blau

Herr Blau, der sich als ruhend im blauen Bezugsystem betrachtet, behauptet, Herr Rot hätte absichtlich auf ihn gezielt. Herr Rot dagegen in seinem roten Bezugsystem behauptet, dass er nicht auf Herrn Blau gezielt hat, sondern die Tomate so geworfen hat, dass sie an Blau hätte vorbeifliegen müssen. Nach einigem Hin und Her beruhigen sich die beiden und einigen sich darauf, dass eine geheimnisvolle Kraft, die Corioliskraft die Tomate abgelenkt haben müsste. Doch wie war es wirklich? Wir wissen, dass die Erde rotiert. Und damit geben wir Herrn Rot recht. Aufgrund der Erdrotation hat Herr Blau sich in den Wurf von Herrn Rot hineingedreht und wurde so von der geradeaus fliegenden Tomate getroffen (siehe Abb. 2.18).

Für diese, bzgl. der Geschwindigkeit der Erdrotation etwas übertriebene Geschichte wollen wir einen Sketch schreiben. In diesem Sketch soll ein einfacher Wechsel zwischen dem blauen und roten Bezugssystem möglich sein. In diesem Sketch benötigen wir bekannte Anweisungen wie die if-Anweisung, logische Operatoren und *noLoop*. Neu ist jedoch der Datentyp **boolean**. Dieser Datentyp kann nur zwei Werte annehmen: **true** oder **false**, also wahr oder falsch. Damit steuern wir in unserer if-Anweisung sehr einfach, ob das blaue Bezugsystem rotieren soll oder nicht.

```
if (Rotation == true)
{
  rotate(w); // Rotation mit dem Bogenmaß w
  w = w - 0.002; // Rotation gegen den Uhrzeigersinn
}
```

In dem obigen Sketchausschnitt steht zwischen den runden Klammern der if-Anweisung *Rotation == true*. Hier sehen wir zum ersten Mal ein doppeltes Gleichheitszeichen auf. Wie unterscheidet sich das doppelte Gleichheitszeichen von einem einfachen Gleichheitszeichen? Bei Processing ist das einfache Gleichheitszeichen ein **Zuweisungsoperator**. Schreibt man a = 5, dann ordnet man der Variablen a den Wert 5 zu. Mit dem doppelten Gleichheitszeichen überprüft man, ob zwei Werte oder Begriffe gleich sind. Das doppelte Gleichheitszeichen ist also ein **Gleichheitsoperator**. Beispiel: Wenn a gleich b ist, dann zeichne einen rot ausgefüllten Kreis.

```
if (a == b)
{
  fill(255, 0, 0);
  ellipse(50, 100, 20, 20);
}
```

Soll das blaue Bezugsystem rotieren, dann schreiben wir im Sketch bei der Deklaration und Initialisierung

```
boolean Rotation = true;
```

Soll das blaue Bezugsystem nicht rotieren, dann schreiben wir

```
boolean Rotation = false;
```

Dadurch ist die Bedingung in der if-Anweisung nicht erfüllt und das blaue Bezugsystem dreht sich nicht. Mit *boolean* und der Zuordnung *true* oder *false* kann man also bequem Animationen ein- oder ausschalten.

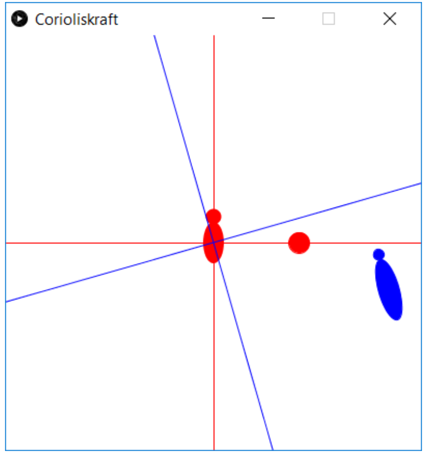


Abbildung 2.18: Der Tomatenwurf aus Sicht von Herrn Rot

Im Sketch tauchen noch drei weitere if-Anweisungen auf. Schauen wir uns zuerst die if-Anweisungen für die Bewegung der Tomate im Bezugsystem von Herrn Blau an. Diese if-Anweisung sorgt dafür, dass die Tomate sich auf Herrn Blau zubewegt, wenn der Winkel gleich Null ist und x <= 150 ist. Bei x = 150 hat die Tomate Herrn Blau getroffen.

```
if (x <= 150 && w == 0)
{
  noStroke();
  fill(255, 0, 0);
  ellipse(x, y, 20, 20);
  x = x + 0.6;
  y = y + 0.3;
}
```

Die folgende if-Anweisung beschreibt den Vorgang aus Sicht von Herrn Rot. Sie sorgt dafür, dass die Tomate geradeaus fliegt, wenn der Winkel w ungleich Null ist und x <= 150. Bei x = 150 hat die Tomate Herrn Blau getroffen.

```
if (x <= 160 && w != 0)
{
  fill(255, 0, 0);
  ellipse(x, 0, 20, 20);
  x = x + 0.6;
}
```

Dir letzte if-Anweisung sorgt dafür, dass die Animation dann stoppt, wenn der Winkel im Bogenmaß kleiner -1 ist. Das Minuszeichen steht deshalb, weil das blaue Koordinatensystem sich gegen den Uhrzeigersinn dreht. Ein positive Drehrichtung ist in Processing eine Drehrichtung im Uhrzeigersinn (siehe Kapitel 1).

```
  if (w < -1)
  {
    noLoop();
  }
```

Damit dürfte der Sketch Corioliskraft ausreichend erklärt sein.

Sketch 14: Corioliskraft

```
// Corioliskraft

float x;
float y;
float w; // Winkel
boolean Rotation = false; // Mit true rotiert das rote Koordinatensystem
                          // Mit false rotiert es nicht

void setup()
{
  size(400, 400);
}

void draw()
{
  background(255);

  translate(200, 200);

  // ruhendes Koordinatensystem (rot)
  stroke(255, 0, 0);
  line(0, -200, 0, 200);
  line(-200, 0, 200, 0);


  // Tomate aus Sicht des roten Beobachters
  if (x <= 160 && w != 0)
  {
    fill(255, 0, 0);
    ellipse(x, 0, 20, 20);
    x = x + 0.6;
  }

  // roter Beobachter im roten Koordinatensystem
  noStroke();
  fill(255, 0, 0);
  ellipse(0, 0, 20, 40);
  ellipse(0, -25, 15, 15);

  if (Rotation == true)
  {
    // Rotation mit dem Bogenmaß w
    rotate(w);
    w = w - 0.002; // Rotation gegen den Uhrzeigersinn
  }
```

```
  // rotierendes Koordinatensystem (blau)
  stroke(0, 0, 255);
  line(-300, 0, 300, 0);
  line(0, -300, 0, 300);

  // Tomate aus Sicht von Blau. Vereinfacht dargestellt als gerade
  // Bewegung
  if (x <= 150 && w == 0)
  {
    noStroke();
    fill(255, 0, 0);
    ellipse(x, y, 20, 20);
    x = x + 0.6;
    y = y + 0.3;
  }

  // blauer Beobachter im blauen Koordinatensystem
  fill(0, 0, 255);
  ellipse(150, 90, 20, 60);
  ellipse(150, 55, 10, 10);

  if (w < -1)
  {
    noLoop(); // Wenn w kleiner -1 ist, dann stoppt die Animation
  }
}
```

2.4.4 Zusammenfassung

Fassen wir zusammen, was wir in Kapitel 2.4 *Kreisbewegungen* gelernt haben, bzw. gelernt haben sollten.

rotate() Mit rotate() kann man das Koordinatensystem um seinen Ursprung drehen. Der Winkel in der Klammer muss im Bogenmaß eingesetzt werden. Alles, was nach dem Aufruf von *rotate()* gezeichnet wird, dreht sich mit.

rotate(radians()) Möchte man den Winkel in Grad einsetzen, dann kann man ihn mit radians() ins Bogenmaß umwandeln.

triangle() Mit der Funktion triangle() kann man ein Dreieck zeichnen.

Vektorpfeile Vektorpfeilspitzen lassen sich am einfachsten zeichnen, wenn die Linien, an die die Dreiecke angefügt werden, auf der x- oder y-Achse liegen.

pushMatrix() pushMatrix() speichert das aktuelle Koordinatensystem.

popMatrix() popMatrix() lädt das gespeicherte Koordinatensystem wieder und vergisst alles, was zwischen pushMatrix() und popMatrix() gestanden hat.

boolean Dieser Datentyp kann nur zwei Werte annehmen: **true** oder **false**, also wahr oder falsch. Damit steuert man ganz bequem den Ablauf eines Sketches. Im folgenden Beispiel rotiert das Koordinatensystem nur, wenn in der Klammer Rotation == true steht. Steht hier Rotation == false, dann rotiert es nicht.

```
   if (Rotation == true)

{
rotate(w); // Rotation mit dem Bogenmaß w
```

```
w = w - 0.002; // Rotation gegen den Uhrzeigersinn
}
```

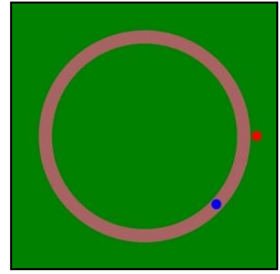
Zuweisungsoperator und Gleichheitsoperator

In dem obigen Sketchausschnitt steht zwischen den runden Klammern der if-Anweisung *Rotation == true*. Wie unterscheidet sich das doppelte Gleichheitszeichen von einem einfachen Gleichheitszeichen? Bei Processing ist das einfache Gleichheitszeichen ein Zuweisungsoperator. Schreibt man a = 5, dann ordnet man der Variablen a den Wert 5 zu. Mit dem doppelten Gleichheitszeichen überprüft man, ob zwei Werte oder Begriffe gleich sind. Das doppelte Gleichheitszeichen ist also ein Gleichheitsoperator. Beispiel: Wenn a gleich b ist, dann zeichne einen rot ausgefüllten Kreis.

```
if (a == b)
{
  fill(255, 0, 0);
  ellipse(50, 100, 20, 20);
}
```

2.4.5 Aufgaben

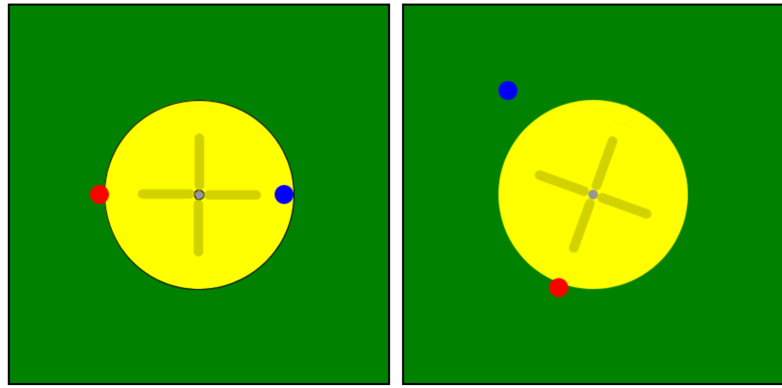
1. Erweitere den Sketch *Sonne_Planet_Sonne* so, dass um den Mond auch noch ein Satellit kreist.
2. Ändere den Sketch *Planet_Mond* so um, dass der Mond einmal im und einmal gegen den Uhrzeigersinn um den Planeten kreist. Verwende hierzu die Variable *boolean*. Der Vektorpfeil für die Geschwindigkeit v soll entsprechend der jeweiligen Drehung stets in die richtige Richtung zeigen.
3. Folgender Vorgang soll simuliert werden. Hase Heinz (roter Punkt in der Abbildung) und Igel Erich (blauer Punkt in der Abbildung) machen einen Wettlauf auf einer Kreisbahn. Hase Heinz steht noch neben der Laufbahn, als Igel Erich hier schon Position bezogen hat. Großzügig wie Heinz ist, erlaubt er Erich einen Vorsprung von einer halben Kreisbahn. Erst dann springt Heinz auf die Laufbahn und läuft mit der 1,4-fachen Winkelgeschwindigkeit von Erich hinter Erich her. Erich läuft mit einer Winkelgeschwindigkeit von 0, 6 rad/s. Nach 1,75 Kreisbahnen hat Heinz Erich eingeholt. Die Laufbahn hat einen Durchmesser von 300 Pixel und eine Breite von 20 Pixel (siehe Abbildung).

   Berechne mit Bleistift und Papier, welche Zeit Erich und welche Zeit Heinz benötigt, um 1,75 Kreisbahnen zurückzulegen. Lasse dir diese Zeiten zur Kontrolle auch in der Konsole anzeigen. Hohe Bildwiederholungsraten liefern ein besseres Ergebnis. Vorausgesetzt, die Grafikkarte kann dies auch leisten.

   Tipp: *pushMatrix()* und *popMatrix()* verwenden.
4. Nach ihrem Wettlauf vergnügen sich Hase Heinz und Igel Erich auf dem Kinderspielplatz. Erich setzt sich auf die Drehscheibe und Hase Heinz dreht sie immer schneller und schneller im Uhrzeigersinn. Nach genau 3,5 Umdrehungen fliegt Igel Erich tangential von der Drehscheibe.

Simuliere den geschilderten Vorgang in einem 400 mal 400 Pixel großen Fenster, entsprechend den folgenden Abbildungen.



2.5 Einführung in die Vektorrechnung

In der Physik unterscheidet man zwischen skalaren Größen und vektoriellen Größen. Skalare Größen werden durch eine einfache Zahl mit Maßeinheit vollständig beschrieben. Zum Beispiel Temperatur, Dichte, … . Vektorielle Größen, wie die Kraft, die Geschwindigkeit, … können mit Hilfe von Pfeilen anschaulich dargestellt werden. D.h., sie verfügen über eine Länge (Betrag) und eine Richtungsangabe. Jeder Vektor kann auch mittels seiner Komponenten x und y, bzw. x, y, und z bei dreidimensionalen Vektoren, dargestellt werden (siehe Abb. 2.19). Ein Vektor beinhaltet also mehrere Werte. Dies ist der Grund dafür, dass man Vektoren nicht wie einfache Zahlen addieren oder multiplizieren kann. Anstelle von +, -, … stehen in Processing die Rechenzeichen **add**, **sub**, …. Diese Regeln für die Rechnung mit Vektoren wollen wir uns nun in Processing näher ansehen. Ein Blick in die Referenz von Processing mit dem Suchbegriff ***PVector*** zeigt uns, dass Processing zahlreiche Werkzeuge bzgl. der Vektorrechnung anbietet. Eine Auflistung findet man im Kapitel 2.5.5. Die wichtigsten Regeln werden nun anhand von einfachen Beispielen erläutert. Beachten müssen wir hierbei, dass die Vektoren bei Processing immer im Koordinatenursprung beginnen und die positive y-Achse bei Processing nach unten zeigt (siehe Abb. 2.19). Die positive z-Achse für dreidimensional orientierte Vektoren zeigt aus der Zeichenebene heraus, also auf uns zu.

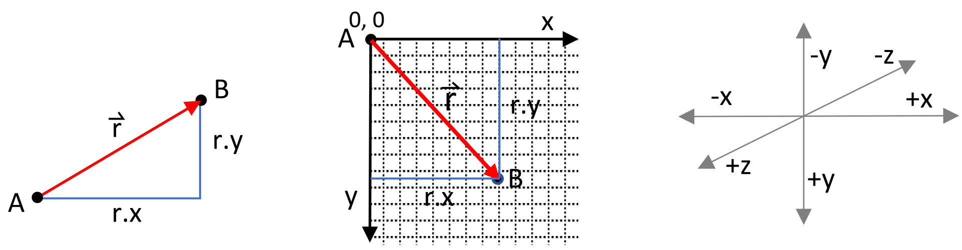


Abbildung 2.19: Ein Vektor mit seinen Komponenten und seine Darstellung in Processing

2.5.1 Addition und Subtraktion

Vektoren sind keine einfachen Variablen, sondern eine Klasse für sich. Deshalb unterscheiden sie sich in Processing auch in ihrer Schreibweise von den einfachen Variablen. An diese Schreibweise muss man sich aber erst einmal gewöhnen. Um eine einfache Variable zu deklarieren und zu initialisieren, schreiben wir beispielsweise: `int v = 15.` Bei einem Vektor schreiben wir:

```
PVector A = new PVector(50, 80, 0);
```

`PVector A` kann in Analogie zu `int v` gesetzt werden. Mit `new PVector(50, 80, 0);` werden dem Vektor A konkrete Wert zugeordnet. Dies entspricht bei `int v = 15` der Zahl **15**, die der Variablen **v** zugeordnet wird. In der Klammer hinter PVector stehen die x-, y- und z-Koordinaten des Vektors A. Bei zweidimensionalen Vektoren genügt es, nur die x- und y-Koordinaten aufzuführen.

Addition

Addieren wir nun die Kräfte $\vec{F}_1$ und $\vec{F}_2$ vektoriell und lassen uns das Ergebnis $\vec{F}_3$ in der Konsole anzeigen. Hier ist der entsprechende Sketch.

```
PVector F1 = new PVector(150, 60);
PVector F2 = new PVector(100, 240);
PVector F3 = PVector.add(F1, F2);
println("F1 =", F1, "F2 =", F2, "F3 =", F3);
```

Wie oben schon erwähnt addiert man Vektoren bei Processing mit **add** und nicht mit +. Auch darf man den Punkt vor dem add nicht vergessen. In der Konsole steht nach der Addition:

F1 = [150.0, 60.0, 0.0]   F2 = [100.0, 240.0, 0.0]   F3 = [250.0, 300.0, 0.0]

Wir sehen, dass Processing einfach die Koordinatenwerte von $\vec{F}_1$ und $\vec{F}_2$ addiert hat, um so den Vektor $\vec{F}_3$ zu erhalten. Da wir keine z-Komponente angegeben haben, steht für z der Wert Null.

Wie kann man nun diese drei Vektoren $\vec{F}_1$, $\vec{F}_2$ und $\vec{F}_3$ in Processing zeichnen? Ein Vektorpfeil besitzt einen Endpunkt mit einer Pfeilspitze. In den Klammern werden bei Processing jedoch nur die Koordinatenwerte für den Endpunkt eines Vektors angegeben. Der Anfangspunkt liegt stets im Koordinatenursprung. Schauen wir uns den folgenden Sketch und die zugehörige Abbildung an. Da wir nur etwas zeichnen wollen und nichts bewegen wollen, schreiben wir den folgenden Sketch nicht im aktiven Modus mit *void setup()* und *void draw()* sondern schlicht und einfach im statischen Modus. Im statischen Modus arbeitet Processing alle Zeilen nacheinander ab und macht dann Schluss. Am Beispiel von Vektor $\vec{F}_1$ soll noch auf die besondere Schreibweise für die Linien, die die Vektoren darstellen, hingewiesen werden. Da die Vektoren im Ursprung des Koordinatensystems beginnen, steht für den Anfangspunkt der entsprechenden Linie 0, 0. Für den Endpunkt der Linie gibt man F1.x und F1.y ein. Also die Koordinatenwerte von Vektor $\vec{F}_1$.

```
line(0, 0, F1.x, F1.y);
```

Sketch 15: Vektoraddition_01

```
// Vektoraddition 01

PVector F1 = new PVector(150, 60);
PVector F2 = new PVector(100, 240);
```

```
size(350, 350);
background(255);

PVector F3 = PVector.add(F1, F2);

// Vektor F1 (grün)
stroke(0, 255, 0);
strokeWeight(5);
line(0, 0, F1.x, F1.y);

// Vektor F2 (blau)
stroke(0, 0, 255);
strokeWeight(5);
line (0, 0, F2.x, F2.y);

// Vektor F3 (rot)
stroke(255, 0, 0);
strokeWeight(5);
line (0, 0, F3.x, F3.y);

println("F1 =", F1, "F2 =", F2, "F3 =", F3);
```

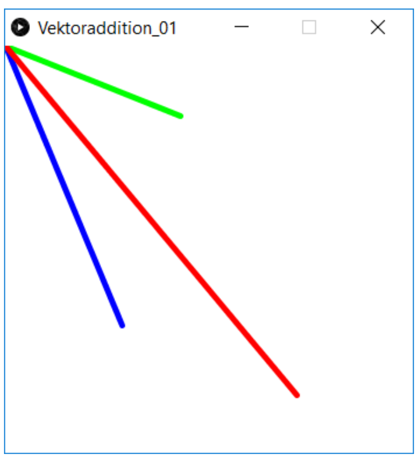


Abbildung 2.20: Die rote Linie stellt den resultierenden Vektor dar, der sich aus der Addition des grünen und blauen Vektors ergibt.

In der Abbildung 2.20 zeigt sich uns leider ein ungewohntes Bild. Erstens fehlen bei allen Vektoren die Pfeilspitzen und zweitens beginnen sie, wie oben schon erwähnt, alle im Ursprung. Für die Pfeilspitzen könnten wir kleine Dreiecke zeichnen und sie den Vektorlinien überlagern. Da die Vektorlinien jedoch schräg stehen, ist dies nicht unmöglich, aber doch sehr umständlich. Machen wir uns das Leben etwas einfacher und zeichnen bei unserem nächsten Sketch stattdessen lieber ungefährliche Kinderpfeile ☺, die anstelle einer gefährlichen Spitze einen Kreis an ihrem Ende haben.

Bei der Vektoraddition mit Bleistift und Papier sind wir es gewohnt, den Anfangspunkt des zweiten Vektors an die Spitze des ersten Vektors zu legen und anschließend den Anfangspunkt des ersten Vektors mit dem Endpunkt des zweiten Vektors zu verbinden, um so den resultierenden Vektor zu erhalten. Wenn wir unseren obigen Sketch etwas abändern, erhalten wir auch eine solche Darstellung.

Sketch 16: Vektoraddition_02

```
// Vektoraddition 02

PVector F1 = new PVector(150, 60);
PVector F2 = new PVector(100, 240);

size(350, 350);
background(255);

PVector F3 = PVector.add(F1, F2);

// Vektor F1 (grün)
stroke(0, 255, 0);
strokeWeight(5);
line(F2.x, F2.y, F3.x, F3.y);
fill(0, 255, 0);
ellipse(F1.x + F2.x, F1.y + F2.y, 10, 10);

// Vektor F2 (blau)
stroke(0, 0, 255);
strokeWeight(5);
line (0, 0, F2.x, F2.y);
fill(0, 0, 255);
ellipse(F2.x, F2.y, 10, 10);

// Vektor F3 (rot)
stroke(255, 0, 0);
strokeWeight(5);
line (0, 0, F3.x, F3.y);
fill(255, 0, 0);
ellipse(F3.x, F3.y, 10, 10);

println("F1 =", F1, "F2 =", F2, "F3 =", F3);
```

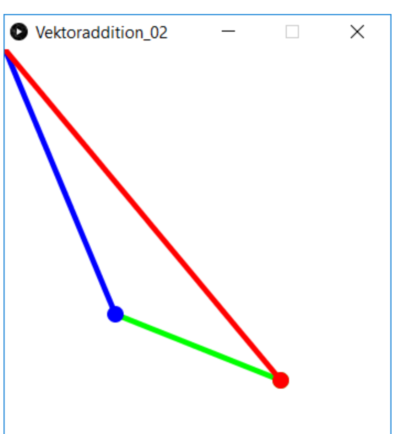


Abbildung 2.21: Der grüne Vektor wurde in die Spitze des blauen Vektors verschoben

Subtraktion

Bei der Subtraktion ersetzen wir einfach *add* durch *sub* und lassen uns das Ergebnis wieder in der Konsole anzeigen.

```
PVector F1 = new PVector(150, 60); // grün
PVector F2 = new PVector(100, 240); // blau
PVector F3 = PVector.sub(F1, F2);
println("F1 =", F1, "F2 =", F2, "F3 =", F3);
```

Konsolenwerte: F1 = [150.0, 60.0, 0.0] F2 = [100.0, 240.0, 0.0] F3 = [50.0, -180.0, 0.0]

Wir sehen, Processing hat richtig gerechnet.

In der Konsole steht F3 = [50.0, -180.0, 0.0]. D.h., wir erhalten für den Vektor $\vec{F}_3$ einen negativen y-Wert. Damit wir ihn trotzdem in einem Fenster darstellen können, müssen wir mit *translate()* das Koordinatensystem entsprechend verschieben. Dabei müssen wir auch bedenken, dass der grüne Vektor $\vec{F}_1$ infolge der Rechnung $\vec{F}_3 = \vec{F}_2 - \vec{F}_1$ seine Richtung umkehrt. Um dies zu verdeutlichen, wurden in Abbildung 2.22 die Koordinatenachsen eingezeichnet. Auch muss dies bei der Zeichnung der Linien und Ellipsen beachtet werden (siehe Sketchausschnitt unten).

```
// Vektor F1 (grün)
stroke(0, 255, 0);
strokeWeight(5);
line(F2.x, F2.y, -F1.x+F2.x, -F1.y+F2.y);
fill(0, 255, 0);
ellipse(-F1.x+F2.x, -F1.y+F2.y, 10, 10);
```

Tja, tja! Bei der Vektorrechnung muss man viel nachdenken. Aber wenn man es einmal begriffen hat, dann steht einem ein sehr mächtiges Werkzeug zur Verfügung.

Sketch 17: Vektorsubtraktion

```
// Vektorsubtraktion

PVector F1 = new PVector(150, 60); // grün
PVector F2 = new PVector(100, 240); // blau

size(500, 500);
background(255);

translate(200, 100);

// Das verschobene Koordinatensystem wird gezeichnet
line(-200, 0, 300, 0);
line(0, -100, 0, 400);

PVector F3 = PVector.sub(F1, F2);

// Vektor F1 (grün)
stroke(0, 255, 0);
strokeWeight(5);
line(F2.x, F2.y, -F1.x+F2.x, -F1.y+F2.y);
fill(0, 255, 0);
ellipse(-F1.x+F2.x, -F1.y+F2.y, 10, 10);

// Vektor F2 (blau)
stroke(0, 0, 255);
strokeWeight(5);
line (0, 0, F2.x, F2.y);
fill(0, 0, 255);
ellipse(F2.x, F2.y, 10, 10);
```

```
// Vektor F3 (rot)
stroke(255, 0, 0);
strokeWeight(5);
line (0, 0, -F1.x+F2.x, -F1.y+F2.y);
fill(255, 0, 0);
ellipse(-F1.x+F2.x, -F1.y+F2.y, 10, 10);

println("F1 =", F1, "F2 =", F2, "F3 =", F3);
```

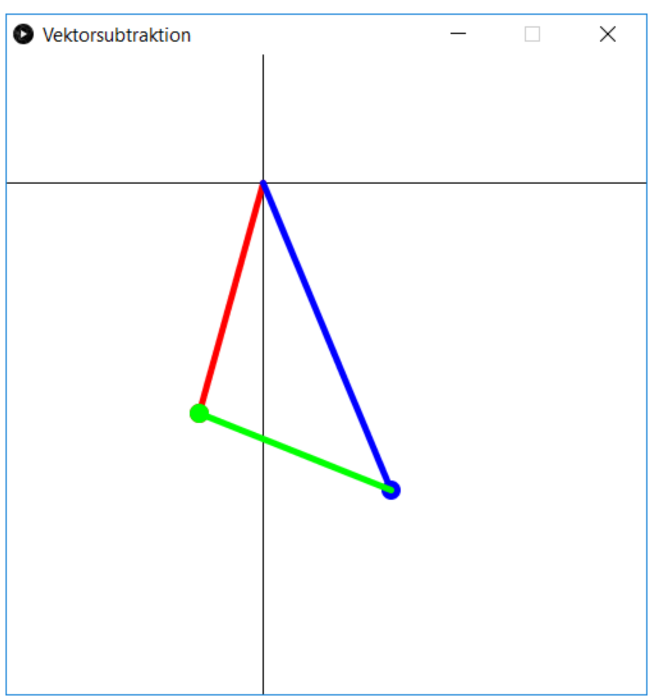


Abbildung 2.22: Von dem blauen Vektor wird der grüne Vektor subtrahiert

Anmerkung

So wie man gewohnt ist 3 = 5 – 2 zu rechnen, so ist man auch versucht Vektoren auf ähnliche Weise zu subtrahieren. Dieses unten aufgeführte Beispiel ist jedoch falsch. Wenn man sich die Werte der drei Vektoren in der Konsole anzeigen lässt, bemerkt man diesen Fehler.

```
PVector A = new PVector(150, 60);
PVector B = new PVector(100, 240);
PVector C;

C = A.sub(B);
```

Falsch!

```
println("A =", A, "B =", B, "C =", C);
```

In der Konsole steht: A = [50.0, -180.0, 0.0] B = [100.0, 240.0, 0.0] C = [50.0, -180.0, 0.0]

Den Werten aus der Konsole entnimmt man, dass durch die obige falsche Addition auch die Koordinaten des Vektors $\vec{A}$ verändert wurden. Er besitzt nun die gleichen Werte wie Vektor $\vec{C}$. Dies sollte bei einer korrekten Subtraktion jedoch nicht der Fall sein. Wie richtig subtrahiert wird sehen wir unten. Gleiches gilt auch für die Addition von Vektoren.

```
PVector A = new PVector(150, 60);
PVector B = new PVector(100, 240);
PVector C = PVector.sub(A, B);
println("A =", A, "B =", B, "C =", C);
```

Richtig!

A = [150.0, 60.0, 0.0] B = [100.0, 240.0, 0.0] C = [50.0, -180.0, 0.0]

2.5.2 Multiplikation eines Vektors mit einer Zahl

Bei Vektoren muss man drei Arten der Multiplikation unterscheiden. Die einfachste Multiplikation ist die Multiplikation eines Vektors mit einer Zahl. Im Physikunterricht lernt man dann noch das Skalare Produkt zum Beispiel anhand der Formel für die Arbeit $W = \vec{F} \cdot \vec{s} = F \cdot s \cdot cos\alpha$ kennen sowie das Kreuzprodukt anhand der Formel für das Drehmoment $\vec{M} = \vec{r} \times \vec{F} \implies M = r \cdot F \cdot sin\alpha$.

Bei der Multiplikation eines Vektors mit einer Zahl ändert sich der Betrag des Vektors, aber nicht seine Richtung. Erhöhen wir die Geschwindigkeit $\vec{v}$ zum Beispiel um den Faktor 3, um so die neue Geschwindigkeit $\vec{u}$ zu erhalten, so schreiben wir

```
PVector u = PVector.mult(v, 3)
```

D.h., wir deklarieren den Vektor $\vec{u}$ und initialisieren ihn mit 3·v. Der Rest des unten aufgeführten Sketches dient der grafischen Darstellung. Der rote Vektor $\vec{u}$ wurde dünn gezeichnet, damit man den blauen Vektor $\vec{v}$ noch sehen kann.

Um zu überprüfen, ob unsere Programmierung richtig ist, lassen wir uns die Werte für $\vec{u}$ und $\vec{v}$ in der Konsole anzeigen. Wie wir sehen, haben wir alles richtig gemacht.

Konsolenanzeige: v = [100.0, 100.0, 0.0]   u = [300.0, 300.0, 0.0]

Sketch 18: Vektor_mal_Zahl

```
// Multiplikation eines Vektors mit einer Zahl

size(400, 400);
background(255);

// blauer Vektor v
PVector v = new PVector(100, 100, 0); // Der Vektor v wird deklariert
                                       // und initialisiert
stroke(0, 0, 255);
strokeWeight(10);
line(0, 0, v.x, v.y);
fill(0, 0, 255);
ellipse(v.x, v.y, 15, 15);
textSize(40);
text("v", 65, 50);

// roter Vektor u
PVector u = PVector.mult(v, 3); // Der Vektor u wird deklariert und mit
                                // 3 mal v initialisiert
strokeWeight(1);
stroke(255, 0, 0);
```

```
strokeWeight(4);
line(0, 0, u.x, u.y);
fill(255, 0, 0);
ellipse(u.x, u.y, 8, 8);
textSize(40);
text("u", 210, 200);

println("v =", v, "u =", u); // Hiermit überprüfen wir, ob unsere
                             // Programmierung richtig ist
```

In der Konsole steht: v = [100.0, 100.0, 0.0]   u = [300.0, 300.0, 0.0]

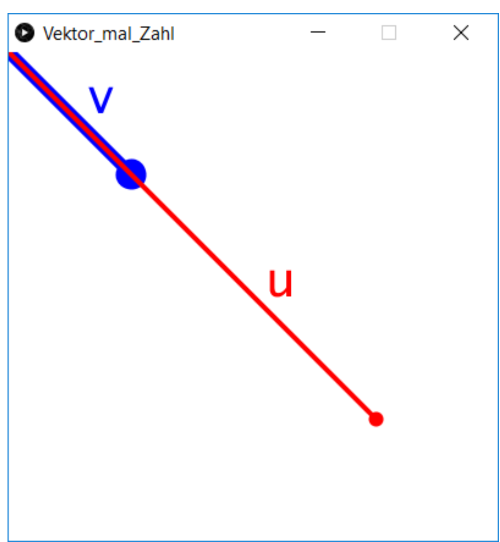


Abbildung 2.23: Multiplikation des Vektors v mit der Zahl 3

Bewegung vektoriell

Wenden wir unser neu erworbenes Wissen nun auf die geradlinig beschleunigte Bewegung zweier Körpern an. Auf beide Körper soll jeweils eine Kraft nach der Gleichung $\vec{F} = m \cdot \vec{a}$ einwirken. Hierbei sollen die Größen der beiden Massen und die jeweils auf sie einwirkenden Kräfte frei wählbar sein. Für den unten aufgeführten Sketch haben wir für den roten Körper eine Masse von m = 10 kg und für den grünen Körper eine Masse von m_2 = 40 kg gewählt. Auf beide Körper wirkt während der ganzen Strecke eine Kraft von 20 N ein. Wie lange braucht der rote und wie lange braucht der grüne Körper, um die 100m-, die 400m- und die 900m-Marke zu erreichen?

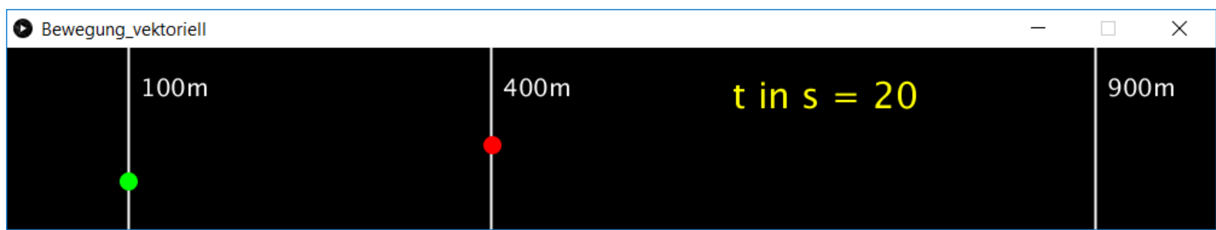


Abbildung 2.24: vektorielle Beschleunigung

Damit wir die Richtigkeit unserer Programmierung überprüfen können, rechnen wir zuerst die Beschleunigungen schriftlich aus.

$F_1 = m_1 \cdot a_1 \Longrightarrow a_1 = \frac{F_1}{m_1} = \frac{20\ N}{10\ kg} = \mathbf{2ms^{-2}}$ roter Körper

$F_2 = m_2 \cdot a_2 \Longrightarrow a_2 = \frac{F_2}{m_2} = \frac{20\ N}{40\ kg} = \mathbf{0,5ms^{-2}}$ grüner Körper

Nun rechnen wir noch aus, in welcher Zeit die Körper die 100m-, die 400m- und die 900m-Marke erreichen.

$s = \frac{1}{2}at^2 \Longrightarrow t = \sqrt{\frac{2 \cdot s}{a}}$ Setzen wir die obigen Werte ein, dann erhalten wir die folgenden Werte.

	100 m	400 m	900 m
roter Körper	10 s	20 s	30 s
grüner Körper	20 s	40 s	60 s

Wie können wir nun diese Werte in unserem Sketch überprüfen? Die Werte für die Kräfte und die Beschleunigungen können wir uns in der Konsole anzeigen lassen. Damit überprüfen wir, ob wir bei der Vektorrechnung keinen Fehler gemacht haben.

Um den Vektor $\vec{a}$ zu erhalten müssen wir den Vektor $\vec{F}$ durch m dividieren. Wie man multipliziert haben wir oben gelernt. Wenn man multiplizieren kann, dann kann man auch dividieren. Man muss *mult* einfach durch *div* ersetzen.

```
PVector a1 = PVector.div(F1, m1)
PVector a2 = PVector.div(F2, m2)
```

Welche Werte stehen nun in der Konsole?

F1 = [20.0, 0.0, 0.0]   F2 = [20.0, 0.0, 0.0]   a1 = [2.0, 0.0, 0.0]   a2 = [0.5, 0.0, 0.0]

Da m_1 = 10 kg und m_2 = 40 kg beträgt, folgt: Wir haben richtig dividiert.

Die Beschleunigungen $\vec{a}_1$ und $\vec{a}_2$ bewirken eine Ortsänderung der beiden Körper. Wie formulieren wir dies in unserem Sketch? Schauen wir uns dies am Beispiel von Körper 1 an.

```
v1.add(PVector.mult(a1, t));
r1.add(PVector.mult(v1, t));
```

Da $\vec{v} = \vec{a} \cdot t$ ist, multiplizieren wir mit *mult* den Vektor $\vec{a}_1$ mit t und addieren mit *add* diesen Geschwindigkeitszuwachs zum Vektor $\vec{v}_1$. Anschließend multiplizieren wir in der zweiten Zeile den so erhaltenen Vektor $\vec{v}_1$ mit t und addieren den so erhaltenen Ortszuwachs zu Vektor $\vec{r}_1$. Das Gleiche machen wir auch für den Körper 2. Bei jedem Durchlauf von *void draw()* werden somit $\vec{v}$ und $\vec{r}$ vergrößert. $\vec{a}$ bleibt konstant.

Nun müssen wir noch überprüfen, ob die beiden Körper entsprechend den Werten in der obigen Tabelle auch zum richtigen Zeitpunkt am richtigen Ort sind. Dazu zeichnen wir zuerst Ortsmarken. Für die 100m-, die 400m- und die 900m-Marke können wir bei 100 Pixel, 400 Pixel und 900 Pixel Linien in unsere Abbildung einfügen. Doch wie überprüfen wir die Zeit? Erinnern wir uns daran, was wir in Kapitel 2.1.1 gelernt haben. Wenn wir die Bildwiederholungsrate (*frameRate*) unseres Computers kennen oder wir sie uns mit *println(frameRate)* in der Konsole anzeigen lassen, dann

können wir die Zeitschritte beim Durchlaufen von *void draw()* so einstellen, dass in genau einer Sekunde Processing den Wert für t um 1 erhöht hat. In dem folgenden Sketch betrug die Bildwiederholungsrate 60 Bilder pro Sekunde. Somit muss der Zeitschritt auf $\frac{1}{60} \approx 0,016667$ eingestellt werden (*float t = 0.016667*). Nun müssen wir uns noch von der im Computer eingebauten Uhr die Zeit zur Überprüfung im Fenster anzeigen lassen. Dies haben wir aber auch schon in Kapitel 2.1.1 gelernt. Es gelingt so:

```
int timer = millis()/1000;
fill(255, 255, 0);
textSize(30); // Textgröße
text("t in s = " +timer, 600, 50);
```

oder so:

```
t = t + 1/frameRate;
fill(255, 255, 0);
textSize(30); // Textgröße
text("t in s = " +t, 600, 50);
```

Damit ist soweit alles erklärt und es wird Zeit, den Sketch *Bewegung_vektoriell* zu testen. Und ganz wichtig: Mit den Werten der Variablen spielen! Wie groß muss zum Beispiel die Kraft auf m_2 = 40 kg sein, damit m_2 gemeinsam mit m_1 = 10 kg die 900m-Marke erreicht?

Sketch 19: Bewegung_vektoriell

```
// beschleunigte Bewegung infolge einer konstant bleibenden Kraft

// Die einfachen Variablen werden deklariert und initialisiert
float m1 = 10.0; // Masse des roten Körpers
float m2 = 40.0; // Masse des grünen Körpers

/* Zeitschritt bei jedem Durchlauf von void draw() bei
einer Bildwiederholungsrate von 60 Bildern pro Sekunde */
float t = 0.016667;

// Die Vektoren werden deklariert und initialisiert.
PVector r1 = new PVector(0, 80); // Ortsvektor des roten Körpers
PVector r2 = new PVector(0, 110); // Ortsvektor des grünen Körpers
PVector v1 = new PVector(0, 0); // Geschwindigkeit des roten Körpers
PVector v2 = new PVector(0, 0); // Geschwindigkeit des grünen Körpers
PVector F1 = new PVector(20, 0); // Kraft, die auf den roten Körper
einwirkt
PVector F2 = new PVector(20, 0); // Kraft, die auf den grünen Körper
                                 // einwirkt
PVector a1 = PVector.div(F1, m1); // Beschleunigung des roten Körpers
PVector a2 = PVector.div(F2, m2); // Beschleunigung des grünen Körpers

void setup()
{
  size(1000, 150);
}

void draw()
{
  background(0);

  // senkrechte Linien für 100 m, 400 m und 900 m
  stroke(255);
  strokeWeight(2);
  line(100, 0, 100, 150);
```

```
  line(400, 0, 400, 150);
  line(900, 0, 900, 150);

  // Beschriftung der Linien
  fill(255); // Textfarbe
  textSize(20); // Textgröße
  text("100m", 110, 40);
  text("400m", 410, 40);
  text("900m", 910, 40);

  // Geschwindigkeit und Ortskoordinate des roten Körpers
  v1.add(PVector.mult(a1, t)); // Bei jedem Durchlauf wird v1 um a1 mal
                               // t vergrößert
  r1.add(PVector.mult(v1, t)); // Bei jedem Durchlauf wird r1 um v1 mal
                               // t vergrößert

  stroke(255, 0, 0);
  strokeWeight(15);
  point(r1.x, r1.y); // Der rote Punkt wird an den Endpunkt von Vektor
                     // r1 gezeichnet

  // Geschwindigkeit und Ortskoordinate des roten Körpers
  v2.add(PVector.mult(a2, t)); // Bei jedem Durchlauf wird v2 um a2 mal
                               // t vergrößert
  r2.add(PVector.mult(v2, t)); // Bei jedem Durchlauf wird r2 um v2 mal
                               // t vergrößert

  stroke(0, 255, 0);
  strokeWeight(15);
  point(r2.x, r2.y); // Der grüne Punkt wird an den Endpunkt von Vektor
                     // r2 gezeichnet

  // Die Zeit lassen wir uns im Fenster anzeigen
  int timer = millis()/1000; // Unser Timer zeigt die Zeit in Sekunden
                             // an
  fill(255, 255, 0); // Textfarbe
  textSize(30); // Textgröße
  text("t in s = " +timer, 600, 50); // Text, Timeranzeige und Lage des
                                     // Textes

  // Wir überprüfen in der Konsole, ob die Ergebnisse der Division
  // richtig sind
  println("F1 =", F1, "F2 =", F2, "a1 =", a1, "a2 =", a2);
}
```

2.5.3 Skalares Produkt

Bei der skalaren Multiplikation zweier Vektoren erhält man keinen neuen Vektor, sondern eine Zahl (Skalar). Bei der Rechenoperation für $W = \vec{F} \cdot \vec{s} = F \cdot s \cdot cos\alpha$ muss deshalb in Processing vor dem W das Wort *float* und nicht *PVector* stehen (siehe unten). Die skalare Multiplikation, also die Rechnung $F \cdot s \cdot cos\alpha$, erfolgt bei Processing mit der Anweisung **dot**. Mit *println(W)* können wir uns das Ergebnis dieser Rechenoperation in der Konsole anzeigen lassen.

```
PVector F = new PVector(100, -173.2, 0);
PVector s = new PVector(250, 0, 0);
float W = F.dot(s);
println(W);
```

Obwohl die oben aufgeführte Rechenoperation nur vier Zeilen beträgt, ist der folgende Sketch recht lang. Dies liegt daran, dass wir natürlich den Ehrgeiz haben, die beiden Vektoren in einer beschrifteten Abbildung grafisch darzustellen (siehe Abb. 2.25).

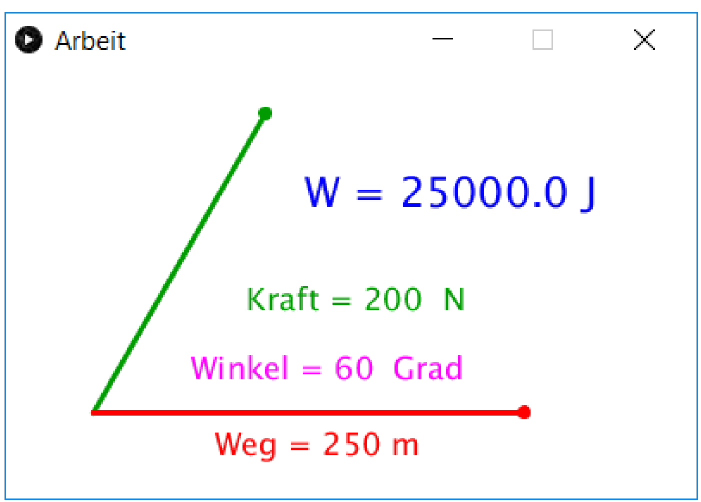


Abbildung 2.25: Berechnung der Arbeit als skalares Produkt

Aufgrund unseres Ehrgeizes lernen wir selbstverständlich auch etwas Neues. Welche Rechenoperation führt zum Beispiel die Funktion **mag()** in der folgenden Zeile durch?

```
int Betrag = round(F.mag());
```

Mit `int Betrag = F.mag()` können wir uns den Betrag von Vektor $\vec{F}$ angeben lassen. Aus der obigen Angabe seiner Koordinaten `PVector F = new PVector(100, -173.2, 0)` kann man ihn ja schließlich nicht so leicht erraten. Das Wort *round* vor *(F.mag())* sorgt dafür, dass wir einen ganzzahlig gerundeten Wert erhalten.

Neu ist auch **angleBetween()**. Mit `PVector.angleBetween(F, s)` erhalten wir den Winkel zwischen dem Vektor $\vec{F}$ und $\vec{s}$. Mit *round* und *degrees* davor erhaltenwir eine gerundete Winkelangabe in Grad.

```
int Winkel = round(degrees(PVector.angleBetween(F, s)));
```

Was die restlichen Codezeilen in dem folgenden Text bedeuten, sollten wir inzwischen wissen. Wie immer können wir auch mit den Werten der Variablen spielen. Wie müssen zum Beispiel die Koordinaten von $\vec{F}$ und $\vec{s}$ gewählt werden, damit gilt W = 0? Bei aller Spielerei sollte man nicht vergessen die Ergebnisse stets mit dem Taschenrechner zu überprüfen.

Sketch 20: Arbeit

```
// Arbeit als skalares Produkt

PVector F = new PVector(100, -173.2, 0);
PVector s = new PVector(250, 0, 0);

size(400, 250);
background(255);
```

```
int W = F.dot(s); // Berechnung der Arbeit W als skalares Produkt

// Berechnung der für die Zeichnung noch fehlenden Werte.
int Betrag = round(F.mag()); // Der Betrag der Vektors F wird berechnet
                             // und ganzzahlig gerundet.
float Winkel = round(degrees (PVector.angleBetween(F, s))); // Der
     // Winkel zwischen F und s wird in Grad berechnet und gerundet

// Koordinatensystem wird verschoben, damit die Werte ins Fenster passen
translate(50, 200);

// Die Vektoren werden gezeichnet
//Kraft als grüner Vektor
stroke(0, 155, 0);
strokeWeight(3);
line(0, 0, F.x, F.y);
fill(0, 155, 0);
ellipse(F.x, F.y, 5, 5);

// Weg als roter Vektor
stroke(255, 0, 0);
strokeWeight(3);
line(0, 0, s.x, s.y);
fill(255, 0, 0);
ellipse(s.x, s.y, 5, 5);

// Die Abbildung wird beschriftet
// Arbeit in J
fill(0, 0, 255);
textSize(24);
textAlign(CENTER);
text("W = " +W, 200, -120);
text(" J", 285, -120);

// Kraft in N
fill(0, 155, 0);
textSize(18);
text("Kraft = " +Betrag, 140, -60);
text("N", 210, -60);

// Weg in m
fill(255, 0, 0);
textSize(18);
text("Weg = 250 m", 130, 25);

// Winkel in Grad
fill(255, 0, 255);
textSize(18);
text("Winkel = " +Winkel, 110, -20);
text("Grad", 195, -20);

// Überprüfung der Werte mittels Konsole
println("Betrag der Kraft = ", Betrag, "  s =", s, "  Winkel =", Winkel,
"  W =", W);
```

2.5.4 Kreuzprodukt

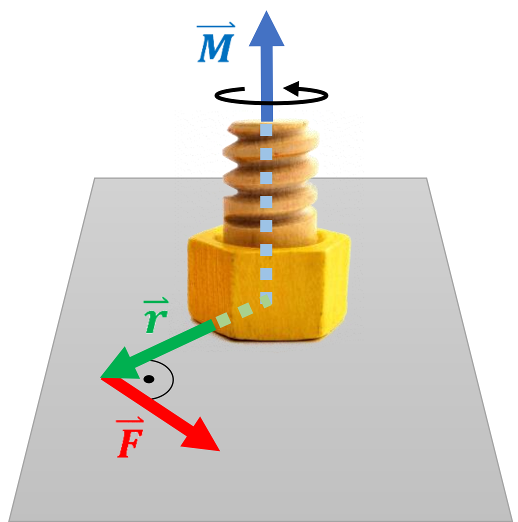


Abbildung 2.26: Vektorstellungen beim Drehmoment

Als Beispiel für ein Kreuzprodukt schauen wir uns die Gleichung für das Drehmoment $\vec{M}$ an.
Sie lautet:

$$\vec{M} = \vec{r} \times \vec{F} = r \cdot F \cdot \sin(\sphericalangle r, F)$$

Mittels `PVector M = F.cross(r);` können wir $\vec{M}$ im folgenden Sketch berechnen.

```
PVector r = new PVector(50, 0, 0); // Vektor von der Drehachse zur Kraft
PVector F = new PVector(0, 100, 0); // Die Kraft steht in diesem
                                    // Beispiel senkrecht zu r
PVector M = F.cross(r);
println(M);
```

Damit wir das Ergebnis der so durchgeführten Rechnung leicht überprüfen können, haben wir die Koordinaten im obigen Sketch so gewählt, dass $\vec{r}$ auf der x-Achse und $\vec{F}$ auf der y-Achse liegt. Da $\vec{r}$ und $\vec{F}$ senkrecht zueinanderstehen, beträgt der Winkel zwischen ihnen 90° und der Sinus nimmt somit den Wert 1 an. Der Vektor $\vec{M}$ steht wiederum senkrecht auf der von $\vec{r}$ und $\vec{F}$ aufgespannten Ebene. In der Konsole steht M = [0.0, 0.0, -5000.0]. D.h., $\vec{M}$ zeigt in die Zeichenebene hinein (siehe Abb. 2.27 links). Mit der Rechten-Hand-Regel können wir die Richtung von $\vec{M}$ überprüfen.

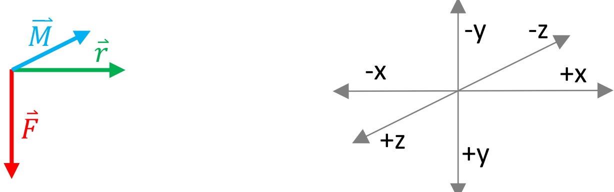


Abbildung 2.27: Vektorstellungen beim Drehmoment und Koordinatensystem von Processing

In der Konsole werden, wie oben schon aufgeführt, die Koordinaten für $\vec{M}$ wie folgt angezeigt: [0.0, 0.0, -5000.0]. Das Minuszeichen vor der z-Koordinate ist korrekt, da bei Processing der negative Teil der z-Achse in die Zeichenebene hinein zeigt (siehe Abb. 2.27 rechts). Weiterhin können wir feststellen, dass wir die Reihenfolge von F und r bei `PVector M = F.cross(r)` richtig gewählt haben. Hätten wir `PVector M = r.cross(F)` geschrieben, dann

hätte $\vec{M}$ aus der Zeichenebene herausgezeigt. Dies wäre falsch gewesen. Bei der Multiplikation zweier Vektoren gilt bekanntlich:

$$\vec{r} \times \vec{F} \neq \vec{F} \times \vec{r}.$$

Vektorielle Animation einer Drehbewegung

Bei der Existenz eines Drehmomentes muss sich zwar nicht zwingend etwas bewegen, doch denkt man bei diesem Wort natürlich auch an eine Drehbewegung (siehe Abb. 2.26). Wir wollen nun einen Sketch schreiben, der uns ein Bild entsprechend der Abbildung 2.28 liefert. Hierbei soll $\vec{r}$ auf der x-Achse und $\vec{F}$ auf der z-Achse liegen. $\vec{F}$ soll an der Spitze (Kreis) von $\vec{r}$ angreifen. $\vec{r}$ und $\vec{F}$ sollen bei der Animation kontinuierlich um die y-Achse rotieren und $\vec{M}$ hierbei natürlich in die richtige Richtung entlang der y-Achse zeigen. Dies verlangt nach einer 3D-Darstellung. Da unser Bildschirm aber nur zweidimensional ist, müssen wir Processing mitteilen, dass wir wenigstens die Illusion einer 3D-Darstellung wünschen. Dies gelingt für unser 600 x 600 Pixel großes Fenster mit

```
size(600, 600, P3D)
```

P3D bedeutet, dass Processing eine z-Komponente entsprechend der Abbildung 2.27 simulieren soll. Aus diesem Grund müssen wir bei unseren Vektoren auch einen Wert für z angeben.

```
PVector r = new PVector(50, 0, 0); // Ortsvektor
PVector F = new PVector(0, 0, -80); // Kraftvektor
```

In der Konsole lassen wir uns mit `println("r =", r, "F =", F, "M =", M)` wieder anzeigen, ob wir bei der Vektorrechnung alles richtig gemacht haben. In der Konsole steht:

r = [50.0, 0.0, 0.0]   F = [0.0, 0.0, -80.0]   M = [0.0, -80.0, 0.0]

Wir entnehmen den Klammerwerten, dass die Vektoren in die richtige Richtung zeigen. $\vec{r}$ zeigt nach rechts in x-Richtung, $\vec{F}$ zeigt in z-Richtung nach hinten und $\vec{M}$ zeigt in y-Richtung nach oben. Das Einzige was auf den ersten Blick stört, ist, dass der Wert für $\vec{M}$ zu klein ist. Denn 50 · (-80) = -4000. Der Wert -4000 ist jedoch zu groß, um ihn auf dem Bildschirm darzustellen. Deshalb haben wir das Kreuzprodukt nachträglich noch mit 0,02 multipliziert (siehe unten). Somit gilt: -4000 · 0,02 = -80. Also, alles ok!

```
PVector M = F.cross(r).mult(0.02);
```

Neu bei dem folgenden Sketch ist, dass wir unsere Zeichenobjekte um die y-Achse rotieren lassen wollen. Dies gelingt mit

```
rotateY(i);
i = i + 0.005;
```

Mit **rotateX**, **rotateY** und **rotateZ** kann man das Koordinatensystem und damit seine Zeichenobjekte um die jeweilige Achse rotieren lassen.

Damit dürften beim Lesen des folgenden Sketches eigentlich keine Fragen mehr auftauchen.

Sketch 21: Drehmoment

```
// Drehmoment M = r x F

float i; // Mit i steuern wir die Rotationsgeschwindigkeit um die
         // y-Achse
PVector r = new PVector(50, 0, 0); // Ortsvektor
PVector F = new PVector(0, 0, -80); // Kraftvektor

void setup()
{
  /* Mit P3D erzeugt Processing auf dem zweidimensionalen Bildschirm
   die Illusion eines dreidimensionalen Bildes */
  size(600, 600, P3D);
}

void draw()
{
  background(255);

  /* Der Betrag von Vektor M beträgt 50 x 80 = 4000 Pixel. Damit er im
     Fenster dargestellt werden kann, multiplizieren wir ihn mit dem
     Faktor 0.02 */
  PVector M = F.cross(r).mult(0.02);

  // Das Koordinatensystem wird in die Fenstermitte verschoben
  translate(300, 300, 300);

  // Koordinatenkreuz zeichnen
  stroke(200);
  strokeWeight(1);
  line(-300, 0, 300, 0);
  line(0, -300, 0, 300);

  // Rotation um die y-Achse
  i = i + 0.005;
  rotateY(i);

  // F-Vektor (rot)
  stroke(255, 0, 0);
  strokeWeight(5);
  line(r.x, r.y, 0, 0, F.y, F.z);
  strokeWeight(15);
  point(0, F.y, F.z);

  // r-Vektor (grün)
  stroke(0, 255, 0);
  strokeWeight(5);
  line(0, 0, 0, r.x, r.y, 0);
  fill(0, 255, 0);
  strokeWeight(15);
  point(r.x, r.y, 0);

  // M-Vektor (blau)
  stroke(0, 0, 255);
  strokeWeight(5);
  line(0, 0, 0, M.x, M.y, M.z);
  strokeWeight(15);
  point(M.x, M.y, M.z);

  // Die einzelnen Vektoren werden beschriftet
```

```
  fill(0, 0, 255);
  textSize(20);
  text("M", 0, -40, -20);
  fill(0, 255, 0);
  text("r", 25, -10);
  fill(255, 0, 0);
  text ("F", -10, -10, -50);

  println("r =", r, "F =", F, "M =", M);
}
```

In der Animation drehen sich auch die Buchstaben mit den Vektoren. Als Vektorspitzen verwenden wir wieder ausgefüllte Kreise (sichere Kinderpfeile ☺).

Abbildung 2.28: Drehmoment - Rotierende Vektoren

2.5.5 Übersicht über die Rechenoperationen für Vektoren

In der Referenz von Processing finden wir noch zahlreiche weitere Möglichkeiten, um Rechenoperationen mit Vektoren durchzuführen (siehe unten). Viele davon werden wir im weiteren Verlauf des Buches noch kennenlernen.

set()	Setzt die Komponenten des Vektors
random2D ()	Erzeugt einen neuen 2D-Einheitsvektor mit einer zufälligen Richtung.
random3D ()	Erzeugt einen neuen 3D-Einheitsvektor mit einer zufälligen Richtung.
fromAngle ()	Erzeugt einen neuen 2D-Einheitsvektor zu einem Winkel
copy()	Erzeugt eine Kopie des Vektors
mag ()	Berechnet den Betrag des Vektors
magSq ()	Berechnet den Betrag des Vektors und quadriert ihn
add()	Addiert zwei Vektoren
sub ()	Subtrahiert zwei Vektoren

mult ()	Multipliziert einen Vektor mit einem Skalar
div ()	Dividiert einen Vektor durch einen Skalar
dist ()	Berechnet den Abstand zwischen zwei Punkten
dot()	Berechnet das skalare Produkt von zwei Vektoren
cross()	Berechnet das Kreuzprodukt
normalize()	Normalisiert den Vektor auf eine Länge von 1 (Einheitsvektor)
limit()	Begrenzt die Größe des Vektors
setMag ()	Legt den Betrag dieses Vektors fest
heading()	Berechnet den Drehwinkel für diesen Vektor
rotate()	Dreht den Vektor um einen bestimmten Winkel (nur 2D)
lerp ()	Interpoliert den Vektor linear zu einem anderen Vektor
angleBetween ()	Berechnet den Winkel zwischen zwei Vektoren
array ()	Gib eine Darstellung des Vektors als Float-Array zurück

2.5.6 Zusammenfassung

Fassen wir zusammen, was wir in Kapitel 2.5 *Einführung in die Vektorrechnung* gelernt haben, bzw. gelernt haben sollten.

Schreibweise von Vektoren

```
PVector A = new PVector(50, 80, 10)
```

Angabe des Betrages eines Vektors

```
float Betrag = F.mag();
```

Angabe des Winkels zwischen zwei Vektoren

```
float Winkel = PVector.angleBetween(F, s);
```

Addition von Vektoren

```
PVector F3 = PVector.add(F1, F2);
```

Subtraktion von Vektoren

```
PVector F3 = PVector.sub(F1, F2);
```

Multiplikation eines Vektors mit einer Zahl (Hier Vektor $\vec{v}$ mal 3)

```
PVector u = PVector.mult(v, 3)
```

Division eines Vektors durch eine Zahl
Im Beispiel wird Vektor $\vec{F}_1$ durch die skalare Größe m_1 geteilt.

```
PVector a1 = PVector.div(F1, m1)
```

Ortsänderungen durch den Beschleunigungsvektor $\vec{a}$

```
v1.add(PVector.mult(a1, t));
r1.add(PVector.mult(v1, t));
```

Skalares Produkt

```
float W = F.dot(s);
```

Kreuzprodukt

```
PVector M = F.cross(r);
```

Linien und andere Zeichenobjekte mittels Vektorkoordinaten zeichnen

```
line(F2.x, F2.y, F3.x, F3.y);
```

3D-Darstellung durch einfügen von P3D in size()

```
size(600, 600, P3D)
```

rotateX, **rotateY** und **rotateZ**

Hiermit kann man seine Zeichenobjekte um die jeweilige Achse rotieren lassen.

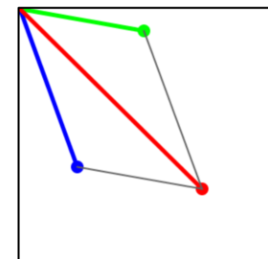
Referenz

In der Referenz von Processing findet man zahlreiche Möglichkeiten um Rechenoperationen mit Vektoren durchzuführen.

2.5.7 Aufgaben

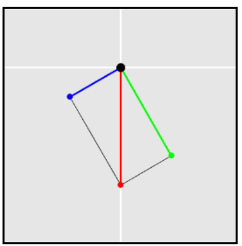
1. Die Abbildung rechts zeigt die Addition der beiden Vektoren F1 (grün) und F2 (blau) zum resultierenden Vektor F3 (rot). F1 hat einen Betrag von 150 N und ist gegenüber der x-Achse um 10° im Uhrzeigersinn gedreht. F2 hat einen Betrag von 200 N und ist gegenüber der x-Achse um 70° im Uhrzeigersinn gedreht.

   Schreibe einen Sketch im statischen Modus, der die Abbildung rechts erzeugt und lasse dir den Betrag von Vektor F3 in der Konsole anzeigen. Überprüfe das Ergebnis mittels Bleistift und Papier.



Tipp: Lege die Vektoren F1 und F2 zuerst auf die x-Achse und drehe sie anschließend um die angegebenen Winkel. Wie dreht man einen Vektor und wie lässt man sich seinen Betrag anzeigen? Gehe mit dem Stichwort *PVector* in die Referenz von Processing und schaue dir hier die Erläuterungen zu *rotate()* und zu *mag()* an.

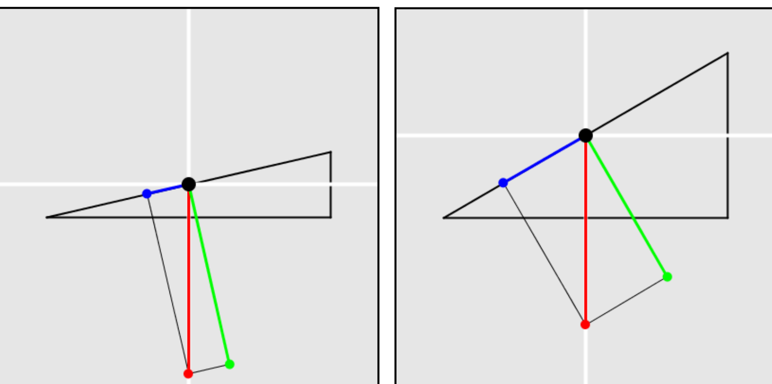
2. Ein Körper (schwarzer Punkt in der Abbildung) liegt fest auf einer schiefen Ebene und erfährt eine Gewichtskraft von 200 N. Entsprechend der Neigung der schiefen Ebene wird die Gewichtskraft in eine Normalkraft und eine Hangabtriebskraft zerlegt. Damit das Programmieren etwas einfacher wird, soll die schiefe Ebene in dieser Aufgabe unsichtbar bleiben und der schwarze Körper im Mittelpunkt des verschobenen Koordinatensystems ruhen (siehe Abbildung). Erst in der Aufgabe 3 soll die schiefe Ebene zusammen mit den drei Vektoren gezeichnet werden.



   Schreibe einen Sketch, in dem man den Steigungswinkel der schiefen Ebene von 0° bis 90° frei wählen kann und sich die Vektoren der Normalkraft und der Hangabtriebskraft automatisch anpassen. Lasse dir in der Konsole die Beträge der drei Vektoren und den Steigungswinkel anzeigen und überprüfe diese Werte mit Bleistift und Papier.

3. Wie in Aufgabe 2 liegt ein Körper (schwarzer Punkt) fest auf einer schiefen Ebene und erfährt eine Gewichtskraft von 200 N. Diesmal soll die schiefe Ebene aber sichtbar sein und sich ihr Steigungswinkel langsam von 0° bis 30° ändern. Dadurch wird der Körper mit angehoben und die Größen von Normalkraft und Hangabtriebskraft ändern sich mit zunehmendem Winkel (siehe Abbildung). Lasse dir in der Konsole die Beträge der drei Vektoren und den Steigungswinkel anzeigen und überprüfe diese Werte mit Bleistift und Papier.

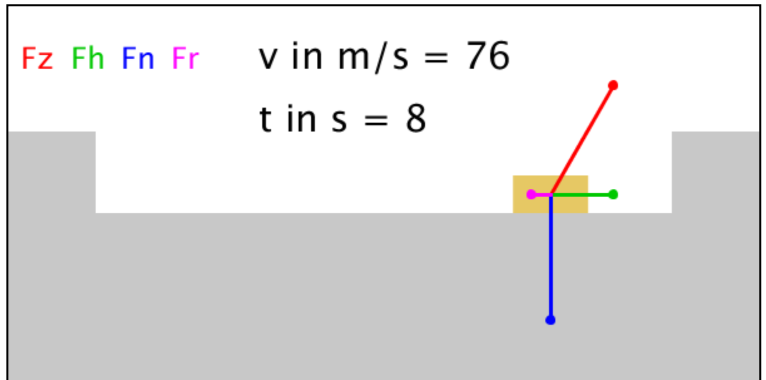
   Tipp: Die Funktionen *tan(), atan()* und *degrees()* können dir bei deiner Programmierung eine Hilfe sein. Informiere dich in der Referenz von Processing über die Eigenschaften dieser Funktionen.



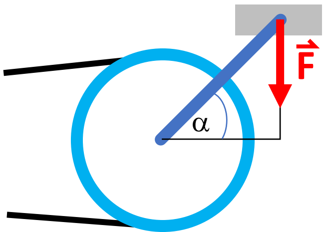
4. Ein Holzklotz (m = 4 kg) wird 400 m über einen Steinboden gezogen. Die ziehende Kraft hat einen Betrag von F_Z = 100 N und greift in einem Winkel von 60° zur Waagerechten am Mittelpunkt des Holzklotzes an. Die Gleitreibungszahl zwischen Holzklotz und Steinboden beträgt μ_G = 0,4.

Simuliere den Bewegungsvorgang mithilfe von Vektoren. Im Fenster sollen entsprechend der folgenden Abbildung die wirkenden Kräfte, die Zeit und die Geschwindigkeit angezeigt werden. Berechne die Endgeschwindigkeit des Holzklotzes mit Bleistift und Papier und vergleiche deinen Wert mit der Anzeige im Fenster.

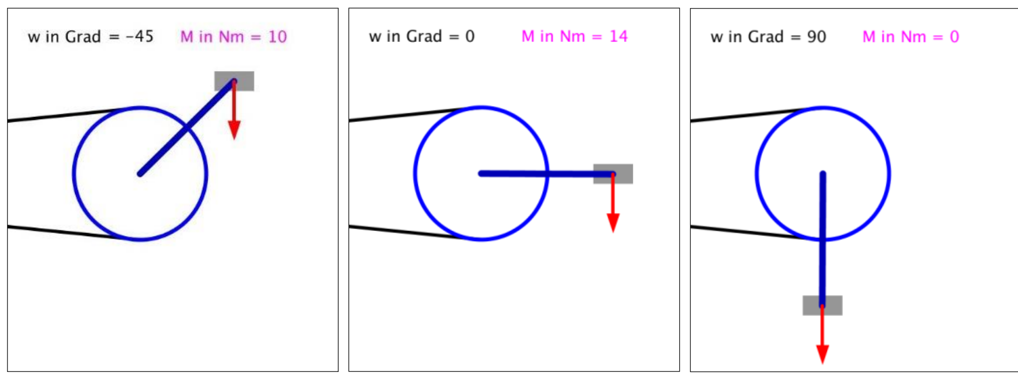
Tipp: Da die Vektoren stets im Ursprung des Koordinatensystems von Processing beginnen, ist es hilfreich, wenn man für die x- und y-Werte bei *translate()* variable Größen einsetzt.



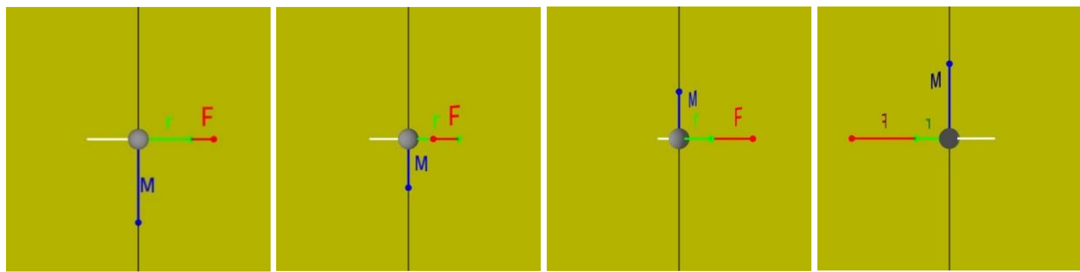
5. Auf das Pedal eines Fahrrades wirkt wie abgebildet eine konstante Kraft mit einem Betrag von 70 N. Die Richtung der Kraft zeigt stets vertikal nach unten. Der Abstand des Pedals zur Drehachse des Zahnrades beträgt 0,20 m. Der Winkel α beträgt zu Beginn der Drehbewegung -45°, gemessen zur Horizontalen (siehe Abbildung rechts).



   Simuliere mit Processing den Drehvorgang von -45° bis 90° (siehe Abbildung unten). Die Größe von Winkel und Drehmoment sollen während der Drehbewegung im Fenster angezeigt werden. Überprüfe diese Werte anhand einer Rechnung mit Bleistift und Papier.



6. Eine dunkelgraue Kugel mit zwei weißen seitlichen Stangen hängt in einer zähflüssigen Flüssigkeit. Die Kugel rotiert um ihr vertikale Achse, da sie von einer Kraft F in Bewegung gehalten wird. Die Kraft F greift senkrecht an einem Stangenende an. Sie hat am Anfang einen Betrag von F = 80 N und einen Abstand von 50 cm vom Mittelpunkt der Kugel. Die Kraft nimmt langsam bis zum Wert Null ab. Danach ändert sie ihre Richtung und ihr Betrag nimmt wieder von Null bis 80 N zu (siehe Abbildung).



Schreibe einen Sketch, der diesen Vorgang dreidimensional simuliert. Die Vektoren $\vec{F}$ und $\vec{r}$ sowie das Drehmoment $\vec{M}$ sollen entsprechend der oberen Abbildung dargestellt werden. Als Vektorpfeilspitzen sollen wieder Punkte verwendet werden.

Tipp: Schaue dir den Sketch *Drehmoment* im Buch nochmal ganz genau an.

Bei einer 3D-Darstellung kann man die Kugel als Punkt zeichnen lassen. Besser sieht es aber aus, wenn man eine echte dreidimensionale Kugel zeichnet. Dies gelingt in Processing wie folgt:

```
// dunkelgraue Kugel
noStroke();
fill(150);
lights();
sphere(10);
```

Eine solche dreidimensionale Kugel sitzt stets im Mittelpunkt des Koordinatensystems, sodass man nur ihren Radius wählen kann. Mit *lights()* kann man sie beleuchten. Informationen zu der Funktion *sphere()* findet man in der Referenz von Processing.

3 Felder

Was erwartet uns?

E für die Basis 10, exp für die Basis e, pow(), for(), width, height, mouseX, mouseY, verschachtelte for-Schleifen, sqrt(), Array, i++ statt i = i + 1, length, Verhinderung der Division durch Null durch Addition eines kleinen Wertes im Nenner, colorMode(), HSB-Farbraum, vertex(), beginShape(), endShape(), beginShape(TRIANGLE_STRIP), mouseDragged(), mouseWheel(), a += b statt a = a + b, normalize(), objektorientierte Programmierung (OOP), Klasse (class), Instanzvariable, Konstruktor, Hauptsketch, Nebensketch, set(), continue, sphere(), lights(),

3.1 Einführung

Was ein Kartoffelfeld ist, weiß jeder. Was man in der Physik unter einem Feld versteht, dies wissen viele nicht. Wer sollte auch schon auf die Idee kommen, dass der leere Raum, also das Vakuum, Träger von Eigenschaften sein kann. Nehmen wir mal an, du hast dir einen Sender zusammengelötet und zu Testzwecken sendest du Beethovens Neunte. Nach fünf Minuten explodiert dein selbstgebastelter Sender. Ist nun auch Beethovens Neunte explodiert? Natürlich nicht. Der leere Raum hat Beethovens Neunte. Auf dem Mars oder an einem anderen Ort im Weltraum kann man sie, einen empfindlichen Empfänger vorausgesetzt, zeitversetzt noch hören.

Wenn das Vakuum also Träger von Eigenschaften sein kann, dann versuchen Physiker natürlich, diese Eigenschaften mathematisch zu beschreiben und sich von der Mathematik dann auch ein Bild hiervon zeichnen zu lassen. Schauen wir uns zuerst mal die Gleichungen für das Gravitationsfeld und das elektrostatische Feld einer Punktladung an.

	Potenzial	Feldstärke	Kraft
Gravitationsfeld	$V(r) = -\gamma \cdot \frac{M}{r}$	$\frac{dV_{(r)}}{dr} = G(r) = \gamma \cdot \frac{M}{r^2}$	$F_G(r) = \gamma \cdot \frac{M \cdot m}{r^2}$
elektrostat. Feld	$\varphi(r) = \frac{1}{4\pi\varepsilon_0\varepsilon_r} \cdot \frac{Q}{r}$	$-\frac{d\varphi_{(r)}}{dr} = E(r) = \frac{1}{4\pi\varepsilon_0\varepsilon_r} \cdot \frac{Q}{r^2}$	$F_E(r) = \frac{1}{4\pi\varepsilon_0\varepsilon_r} \cdot \frac{Q \cdot q}{r^2}$

Hätte man analog zur Gravitation auch in der Elektrostatik die ganzen Größen des ersten Bruches zu einer Größe zusammengefasst, dann sähen sich die Gleichungen für beide Felder fast zum Verwechseln ähnlich. Eine Vektorgleichung für G und E erhält man, wenn man die obigen Betragsgleichungen mit dem Einheitsvektor multipliziert.

3.2 Feldstärke

Beginnen wir mit der Feldstärke, da wir diese mit unseren bisherigen Programmierkenntnissen am einfachsten veranschaulichen können. Doch bevor man mit der Erstellung etwas umfangreicherer Programme beginnt, sollte man mit Bleistift und Papier seine Ideen und Gedanken ordnen. Welche Masse M soll zum Beispiel ein felderzeugender Planet haben? In welchem Abstand vom Planeten ergeben sich Feldstärken, die man in Processing so darstellen kann, dass sich die Vektorpfeile nicht überlappen?

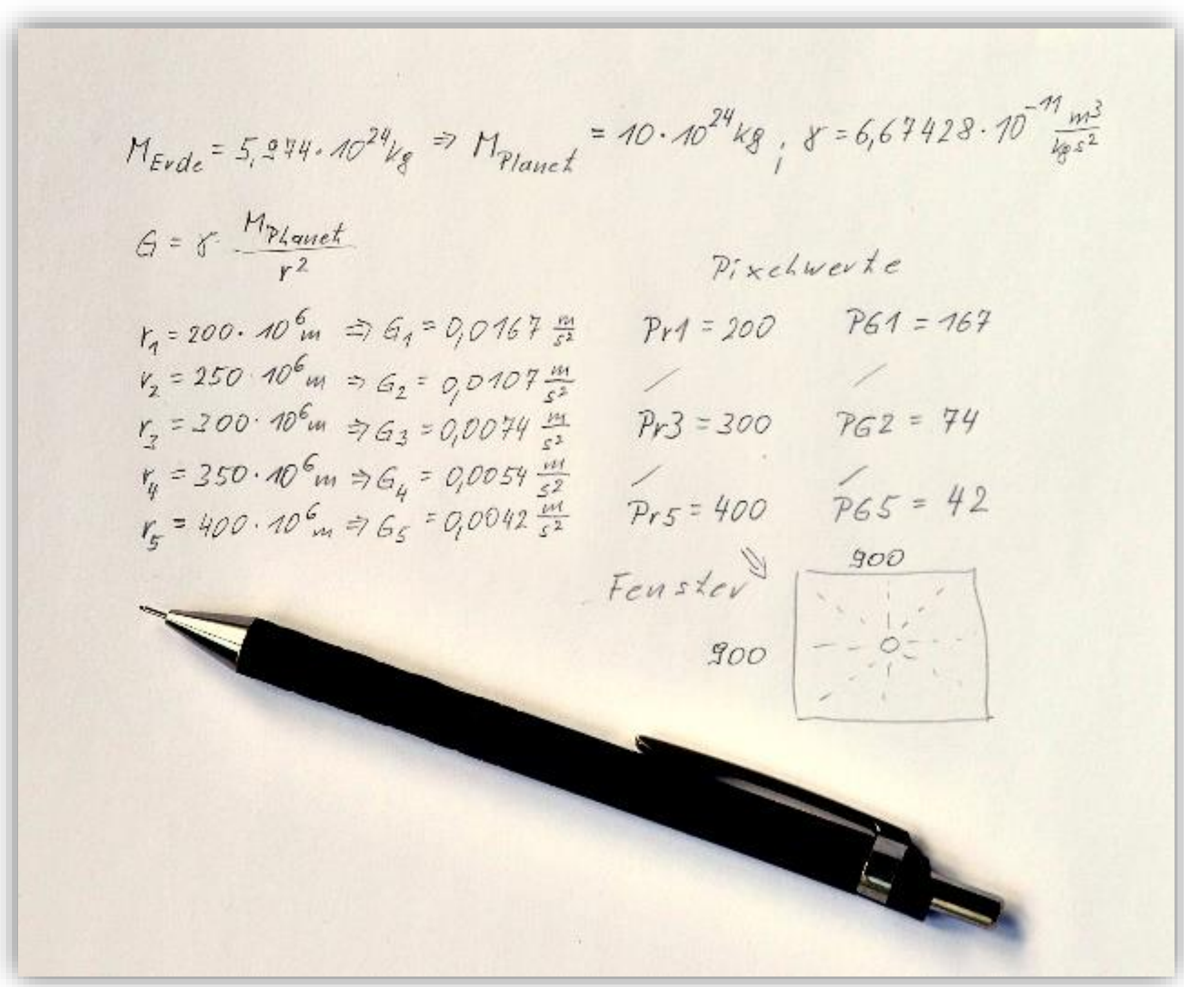


Abbildung 3.1: Vorüberlegungen für den Sketch "Vektorfeld_um_einen_Planeten"

Wie der Abbildung 3.1 zu entnehmen ist, haben wir uns bei der Festlegung der Planetenmasse an der Masse der Erde orientiert. Die Auswahl der Abstände für den Anfangspunkt der Vektoren erfolgte so, dass sich keine Vektoren überlappen. Da wir wissen, dass das Gravitationsfeld eines Planeten radialsymmetrisch ist und die Vektoren zum Mittelpunkt des Planeten zeigen, können wir mit der Betragsgleichung rechnen und unsere Zeichnung entsprechend anpassen.

Obwohl wir auf dem Blatt Papier schon Zahlenwerte ausgerechnet haben, führen wir diese Rechnungen um des Lernen willens auch in Processing durch. Problem Nummer eins: Wie gibt man **Zahlen mit Zehnerpotenzen** in seinem Sketch ein? Hier ist die Lösung.

```
float M = 1.0E25; // Masse des Planeten in kg
float Gamma = 6.67428E-11; // Gravitationskonstante in m*kg^-1*s^-2
```

E steht für die Basis 10. Also 1.0E25 bedeutet $1{,}0 \cdot 10^{25}$ und 6.67428E-11 bedeutet $6{,}67428 \cdot 10^{-11}$. Aber es gibt nicht nur die Basis 10. So steht in Processing **exp** für die Basis e = 2,7182817 (Eulersche Zahl). Mit exp(1) = 2.7182817 kann man dies leicht überprüfen. Wie rechnet man aber 5^3? Hierfür gibt es in Processing die Anweisung **pow()**. pow(5, 3) steht für $5^3 = 125$.

Auf unserem Blatt Papier (Abb. 3.1) haben wir ausgerechnet, dass bei $r_1 = 200 \cdot 10^6$ m die Gravitationsfeldstärke $G_1 = 0{,}0167$ ms^{-2} beträgt. Da wir so große und so kleine Zahlen in unserer winzigen Pixelwelt von 900 Pixel mal 900 Pixel nicht eins zu eins darstellen können, müssen wir

einen Maßstab einführen. Bei unseren Vorüberlegungen haben wir uns für die folgenden Maßstäbe entschieden: $r_1 = 200 \cdot 10^6$ m ≙ 200 Pixel ⇒ 1 Pixel ≙ $1 \cdot 10^6$ m und $G_1 = 0{,}0167$ ms^{-2} ≙ 167 Pixel ⇒ 1 Pixel ≙ $1 \cdot 10^{-4}$ ms^{-2}. Wenn wir aber schon einen Maßstab festlegen, dann sollten wir ihn auch im Fenster darstellen (siehe Abb. 3.2).

Bevor wir nun einige Einzelheit des folgenden Sketches besprechen, schauen wir und zuerst einmal die Abbildung an, die dieser Sketch generiert hat (Abb. 3.2).

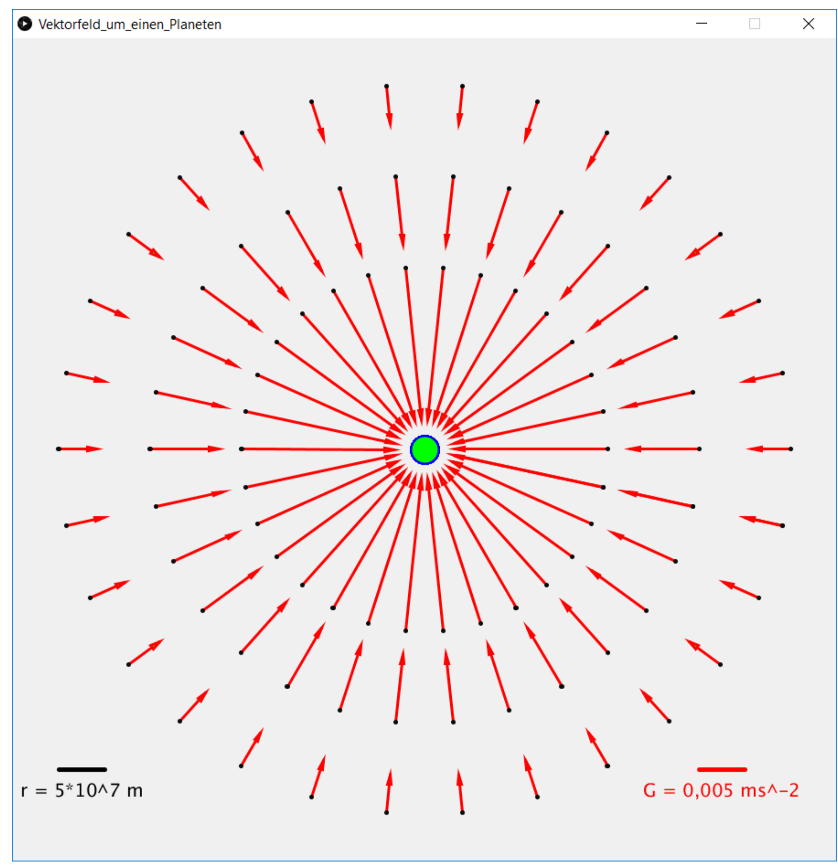


Abbildung 3.2: Vektorfeld um einen Planeten

In Abbildung 3.2 sehen wir Vektorpfeile mit richtigen Pfeilspitzen. Aber in Kapitel 2.5 wurde noch behauptet, dass es schwierig ist, an schräge Linien kleine Dreiecke anzufügen. Dies stimmt. Und aus diesem Grund fügen wir im Sketch auch keine Dreiecke an schräge Linien in Handarbeit ein. Wir fügen nur den drei Linien kleine Dreiecke an, die auf der positiven x-Achse liegen. Mit *rotate(Winkel)* lassen wir diese Vektorpfeile in Winkelabständen von 2*PI/30 um den Ursprung drehen, bis entsprechend der if-Bedingung *if (Winkel <= 2*PI)*, eine Volldrehung abgeschlossen ist. So erhalten wir 30 Vektorpfeile. Diese versehen wir an ihrem anderen Ende noch mit einem kleinen schwarzen Punkt, damit man den Ort, für den der Vektorpfeil die Feldstärke darstellt, besser erkennen kann.

Sketch 01: Vektorfeld_um_einen_Planeten

```
float M = 1.0E25; // Masse des Planeten in kg
float gamma = 6.67428E-11; // Gravitationskonstante in m*kg^-1*s^-2
float winkel; // Winkel im Bogenmaß
float r1 = 200E6; // Abstand in m
float r3 = 300E6; // Abstand in m
```

```
float r5 = 400E6; // Abstand in m
float Pr1 = 200; // Abstand in Pixel
float Pr3 = 300; // Abstand in Pixel
float Pr5 = 400; // Abstand in Pixel
float G1; // Gravitationsfeldstärke in m*s^-2
float G3; // Gravitationsfeldstärke in m*s^-2
float G5; // Gravitationsfeldstärke in m*s^-2
float PG1; // Gravitationsfeldstärke in Pixel
float PG3; // Gravitationsfeldstärke in Pixel
float PG5; // Gravitationsfeldstärke in Pixel


void setup()
{
  size(900, 900);
  background(240);

  // Maßstab für r
  stroke(0);
  strokeWeight(5);
  line(50, 800, 100, 800);
  fill(0);
  textSize(20);
  textAlign(CENTER);
  text("r = 5*10^7 m", 75, 830);

  // Maßstab für G
  stroke(255, 0, 0);
  strokeWeight(5);
  line(750, 800, 800, 800);
  fill(255, 0, 0);
  textSize(20);
  textAlign(CENTER);
  text("G = 0,005 ms^-2", 775, 830);
}

void draw()
{
  // Berechnen der Gravitationsfeldstärke
  G1 = (gamma * M)/(r1 * r1);
  G3 = (gamma * M)/(r3 * r3);
  G5 = (gamma * M)/(r5 * r5);

  // Umrechnung der Gravitationsfeldstärke in Pixelwerte
  PG1 = 1E4 * G1;
  PG3 = 1E4 * G3;
  PG5 = 1E4 * G5;

  // Verschiebung des Ursprungs in die Fenstermitte
  translate(450, 450);

  // Planet zeichnen
  fill(0, 255, 0);
  stroke(0, 0, 255);
  strokeWeight(2);
  ellipse(0, 0, 30, 30);

  if (winkel <= 2*PI)
  {
    winkel = winkel + 2*PI/30; // Winkelzuwachs um 30 Vektorpfeile zu
                               // erhalten
```

```
    rotate(winkel); // Rotation des Koordinatensystems mit den folgenden
                    // Zeichnungsobjekten

    // Vektorpfeile zeichnen
    stroke(255, 0, 0);
    strokeWeight(3);
    line(Pr1, 0, Pr1-PG1, 0);
    triangle(Pr1-PG1+10, -2, Pr1-PG1, 0, Pr1-PG1+10, 2);
    line(Pr3, 0, Pr3-PG3, 0);
    triangle(Pr3-PG3+10, -2, Pr3-PG3, 0, Pr3-PG3+10, 2);
    line(Pr5, 0, Pr5-PG5, 0);
    triangle(Pr5-PG5+10, -2, Pr5-PG5, 0, Pr5-PG5+10, 2);

    stroke(10);
    strokeWeight(5);
    point(Pr1, 0);
    point(Pr3, 0);
    point(Pr5, 0);
  }
}
```

3.3 Potenzial

Äquipotenziallinien

Eine andere Art, die durch eine Masse oder eine Ladung veränderten Eigenschaften des leeren Raumes darzustellen, sind Äquipotenziallinien. Dazu muss man Punkte gleicher Feldstärke miteinander verbinden. In unserem letzten Sketch wären dies zum Beispiel die Anfangspunkte der Vektoren mit gleichem Betrag. Verbindet man sie, so erhält man konzentrische Kreise.

Wir können natürlich auch bestimmte Potenzialwerte vorgeben und für das radialsymmetrische Feld den Abstand r von der felderzeugenden Masse berechnen. Neben Bleistift und Papier kann hierbei auch ein Tabellenkalkulationsprogramm bei der Planung eines entsprechenden Sketches hilfreich sein. Es hilft bei der Berechnung von r und bei der Festlegung des Maßstabes (siehe Abb. 3.3).

$$V = -\gamma \cdot \frac{M}{r} \Rightarrow r = -\gamma \cdot \frac{M}{V}$$

	A	B	C	D
1	V in J/kg	r in m	d in m	**d in Pixel**
2				
3	-1,40E+07	4,7673E+07	9,5347E+07	**95**
4	-1,30E+07	5,1341E+07	1,0268E+08	**103**
5	-1,20E+07	5,5619E+07	1,1124E+08	**111**
6	-1,10E+07	6,0675E+07	1,2135E+08	**121**
7	-1,00E+07	6,6743E+07	1,3349E+08	**133**
8	-9,00E+06	7,4159E+07	1,4832E+08	**148**
9	-8,00E+06	8,3429E+07	1,6686E+08	**167**
10	-7,00E+06	9,5347E+07	1,9069E+08	**191**
11	-6,00E+06	1,1124E+08	2,2248E+08	**222**
12	-5,00E+06	1,3349E+08	2,6697E+08	**267**
13	-4,00E+06	1,6686E+08	3,3371E+08	**334**
14	-3,00E+06	2,2248E+08	4,4495E+08	**445**
15	-2,00E+06	3,3371E+08	6,6743E+08	**667**

Abbildung 3.3: Berechnung der Kreisdurchmesser für die Äquipotenziallinien

Nun fragt man sich vielleicht, warum wir uns nicht mit der Berechnung von r zufriedengegeben haben, sondern auch noch den Kreisdurchmesser d berechnet haben? Erinnern wir uns an Kapitel 1. Wenn man einen Kreis zeichnen will, dann schreibt man zum Beispiel: *ellipse(100, 150, 20, 20)*. Die erste Zahl gibt den x-Wert und die zweite Zahl den y-Wert des Kreismittelpunktes an. Die beiden letzten Zahlen geben nicht den Radius des Kreises an, sondern die dritte Zahl gibt den Durchmesser in x-Richtung und die vierte Zahl den Durchmesser in y-Richtung an. Somit kann man mit der Anweisung *ellipse* eben nicht nur Kreise, sondern auch Ellipsen zeichnen.

Im dem folgenden Sketch begegnet uns zum ersten Mal eine **for-Schleife**. Wir werden sie noch sehr häufig verwenden, da sie das Programmieren komfortabler macht. Schauen wir uns ein einfaches Beispiel an.

```
for (int x = 0; x <= width; x = x + 10)
{
  line(x, 0, x, height);
}
```

In der runden Klammer hinter *for* wird zuerst die Variable x deklariert und initialisiert (`int x = 0)`. Nach dem Semikolon kommt dann die Bedingung (`x <= width)`) und anschließend der Auftrag, was mit der Variablen x geschehen soll, solange die Bedingung erfüllt ist. Der Wert von x wird solange um 10 vergrößert, bis der x-Wert Fensterbreite erreicht hat. **Width** heißt Breite und **height** heißt Höhe. Die eigentliche Anweisung kommt nun in der geschweiften Klammer. Da x Schritt für Schritt um den Betrag 10 erhöht wird, zeichnet Processing nun im Abstand von 10 Pixel solange senkrechte Linien, bis der Fensterrand erreicht ist. Dies geht nicht nur sehr schnell, sondern ist auch recht praktisch. Beachten muss man, dass die Variable in der for-Schleife eine **lokale Variable** ist. D. h., sie besitzt nur innerhalb der for-Schleife ihre Gültigkeit.

Wie verwenden wir nun die for-Schleife in unserem Sketch für die Erstellung der Äquipotenziallinien?

```
for (float V = -1.4E7; V >= -1.4E7 && V <= -2.0E6; V = V + 1.0E6)
{
  r = -1*GAMMA*M/V;
  d = 2*r/1.0E6;
  .....
}
```

Zuerst deklarieren und initialisieren wir die Variable V (`float V = -1.4E7`), die für das Gravitationspotenzial steht. Danach geben wir die Bedingung ein (`V >= -1.4E7 && V <= -2.0E6`) und anschließend, in welchen Schritten V vergrößert werden soll (`V = V + 1.0E6`). Mit diesen V-Werten werden dann die Werte für r und d berechnet (siehe geschweifte Klammer). Die Pünktchen in der obigen geschweiften Klammer bedeuten, dass in der for-Schleife des folgenden Sketches noch mehr passieren soll, als nur die Berechnung von r und d (siehe Sketch).

Sketch 02: Aequipotenziallinien_um_einen_Planeten

```
float r; // Radius in m
float gamma = 6.67428E-11; // Gravitationskonstante in m*kg^-1*s^-2
float M = 1.0E25; // Masse des felderzeugenden Planeten in kg
float d; // Durchmesser der Kreise für die Äquipotenziallinien in Pixel

void setup()
{
  size(800, 800);
}
```

```
void draw()
{
  background(255);
  translate(400, 400); // Verschiebung des Koordinatenursprungs

  // Planet zeichnen
  fill(0, 255, 0);
  stroke(0);
  strokeWeight(2);
  ellipse(0, 0, 30, 30);

  // Berechnung der Radien und der Durchmesser
  for (float V = -1.4E7; V >= -1.4E7 && V <= -2.0E6; V = V + 1.0E6)
  {
    r = -1*gamma*M/V;
    d = 2*r/1.0E6;

    // Überprüfung der Rechenergebnisse
    println("V in J/kg = ", V, "   d in Pixel = ", d);

    // Zeichnen der Äquipotenziallinien
    if (d > 50 && d < 800)
    {
      stroke(0, 0, 255);
      strokeWeight(2);
      noFill();
      ellipse(0, 0, d, d);
    }
  }

  // Beschriftung der Äquipotenziallinien
  stroke(0);
  strokeWeight(5);
  line(50, 800, 100, 800);
  fill(255, 100, 0);
  textSize(20);
  textAlign(CENTER);
  text("V = -2*10^6 J/kg", 0, -350);
  text("V = -3*10^6 J/kg", 0, -230);
  text("V = -4*10^6 J/kg", 0, -175);

  // Maßstab für r
  stroke(0);
  strokeWeight(5);
  line(-350, 340, -250, 340);
  fill(0);
  textSize(20);
  textAlign(CENTER);
  text("r = 1*10^8 m", -300, 370);
}
```

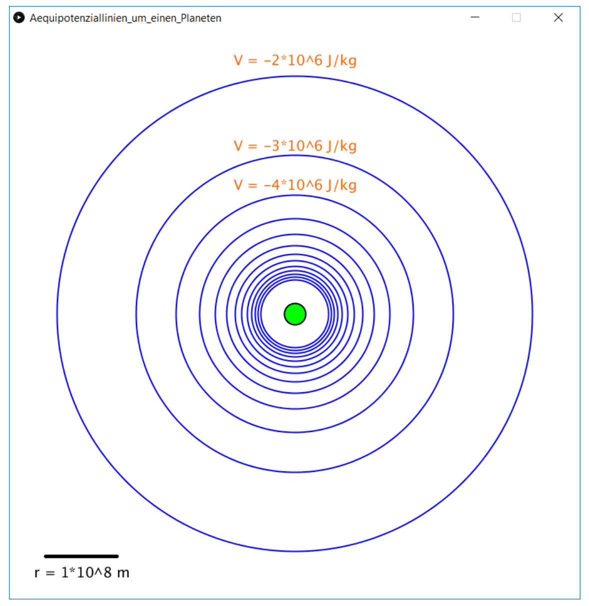


Abbildung 3.4: Äquipotenziallinien um einen Planeten

Potenzialtrichter

Abbildung 3.5 zeigt den Graphen des Gravitationspotenzials $V_{(r)} = -\gamma \cdot \frac{M}{r}$. Der eingezeichnete Planet hat eine Masse von M = 10^{25} kg.

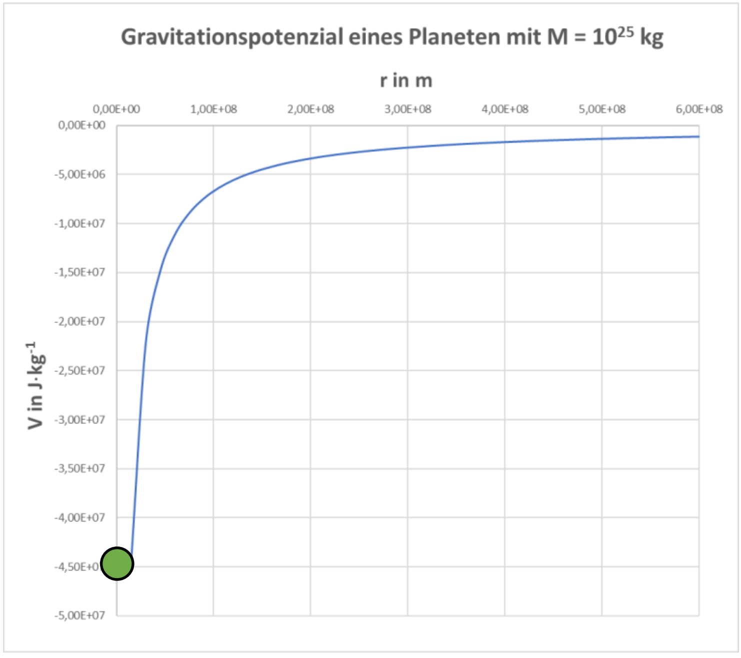


Abbildung 3.5: Der Graph der Funktion V(r)

Der Graph in Abbildung 3.5 wurde mit einem Tabellenkalkulationsprogramm gezeichnet. Etwas vereinfacht können wir ihn auch in Processing zeichnen (siehe Sketch Graph_ Potenzial). Dazu müssen wir jedoch einen Maßstab festlegen.

Sketch 03: Graph_ Potenzial

```
void setup()
{
  size (700, 400);
}

void draw()
{
  background(255);
  translate(50, 50);

  // Koordinatensystem zeichnen
  stroke(0);
  line(-30, 0, 600, 0);
  line(0, -30, 0, 300);
  fill(0);
  triangle(580, -5, 600, 0, 580, 5);
  triangle(-5, -10, 0, -30, 5, -10);
  textSize(30);
  text("V", 20, -15);
  text("r", 580, -15);

  // Graph zeichnen
  // Maßstab: 1 Pixel entspricht r = 10^6 m und die Zahl 10000
  // entspricht
     M = 10^25 kg
  for (float r = 33; r >= 10 && r <= 600; r = r + 1)
  {
    float M = 10000;
    float V = M/r;
    stroke(0, 0, 255);
    strokeWeight(3);
    point(r, V);
  }

  // Planet zeichnen
  fill(0, 255, 0);
  stroke(0);
  ellipse(0, 300, 60, 60);
}
```

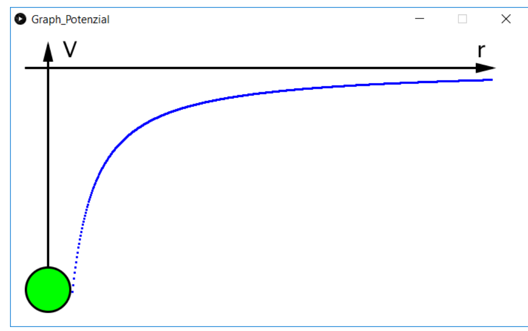


Abbildung 3.6: Graph V(r) mit Processing gezeichnet

Wenn man den Graphen von Abbildung 3.6 ganz schnell um die Hochwertachse rotieren lässt, dann erhält man das Bild eines Potenzialtrichters. Zeichnen wir nun einmal rein qualitativ, also ohne die Gleichung $V_{(r)} = -\gamma \cdot \frac{M}{r}$ einen solchen Potenzialtrichter. Dadurch sehen wir deutlicher, wie vorteilhaft der Einsatz einer *for-Schleife* ist. Man achte darauf, wie in der *for-Schleife* des folgenden Sketches gleich zwei Variablen (r und V) deklariert und initialisiert werden. Um eine räumliche Darstellung zu erhalten, wurden keine Kreise, sondern Ellipsen gezeichnet. Hier ist der Sketch mit einigen ausgewählten Äquipotenziallinien.

Sketch 04: Potenzialtrichter

```
size (400, 350);
background(255);

// Planet zeichnen
fill(0, 255, 0);
ellipse(200, 300, 15, 15);

// Zwei Variablen werden in der for-
Schleife deklariert und initialisiert
for (float r = 1, V = 1; V >= 1 && V <= 200; r = r + 0.6, V = V + 5)
{
  // Äquipotenziallinien zeichnen
  noFill();
  stroke(0, 0, 255);
  strokeWeight(1);
  ellipse(200, 100 + V, 350/(r), 100/(r));
}
```

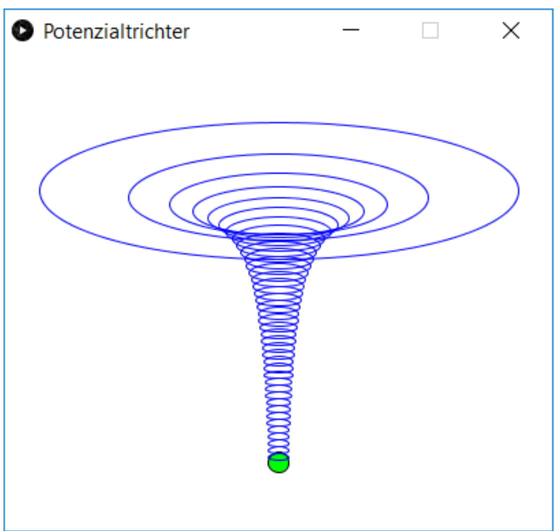


Abbildung 3.7: Potenzialtrichter als Klappbild

Warum wurde die r-Achse und die V-Achse weggelassen? Ohne diese Achsen erhalten wir ein schönes Klappbild. Je nachdem wie man sich die Abbildung 3.7 anschaut, erhält man einmal den Eindruck, als würde man von unten auf den Potenzialtrichter schauen, oder aber den Eindruck, als würde man von oben auf den Potenzialtrichter schauen. Einfach mal ausprobieren.

Potenzialberg

Bisher haben wir nur den Raum um felderzeugende Massen betrachtet. Widmen wir uns nun den felderzeugenden Ladungen. Im Gegensatz zu den stets anziehenden Massen, können Ladungen anziehend und abstoßend sein. D.h., im radialsymmetrischen elektrostatischen Feld gibt es nicht nur Potenzialtrichter, sondern auch Potenzialberge. Eine positive felderzeugende Ladung Q erzeugt nach der Gleichung $\varphi(r) = \frac{1}{4\pi\varepsilon_0\varepsilon_r} \cdot \frac{Q}{r}$ einen Potenzialberg und eine negative Ladung einen Potenzialtrichter. Wirken mehrere Ladungen zusammen, so ergibt sich ein Potenzialgebirge.

Wie können wir ein solchen Potenzialberg dreidimensional darstellen? Denken wir uns eine Ebene entsprechende der Abbildung 3.8, die durch die x- und y-Achse aufgespannt wird. Den Vektor $\vec{r}$ können wir mittels seiner x- und y-Koordinaten darstellen und seinen Betrag mithilfe des Satzes von Pythagoras $r^2 = x^2 + y^2 \Longrightarrow r = \sqrt{x^2 + y^2}$ berechnen. Das Potenzial $\varphi_{(r)}$ steht dann senkrecht auf der x-y-Ebene und zeigt in z-Richtung.

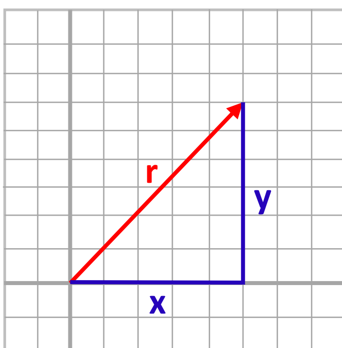


Abbildung 3.8: Die x- und y-Koordinate von Vektor r

Für unsere 3D-Darstellung benötigen wir wieder den Zusatz P3D. Also size(800, 800, P3D). Um die ganze Darstellung etwas interessanter zu machen, fügen wir im Sketch die folgenden Zeilen ein.

```
rotateY(0.006*mouseX);
rotateX(PI/2 + 0.006*mouseY);
```

rotateY ist die Anweisung für die Rotation um die y-Achse um den Winkelwert im Bogenmaß, der in der runden Klammer steht. Diesen Wert verändern wir mit der x-Koordinate des Mauszeigers, indem wir die Maus in x-Richtung verschieben. Hierfür steht **mouseX**. Bei *rotateX* wählen wir mit PI/2 einen anderen Startwert und verändern den Winkelwert mit der y-Koordinate des Mauszeigers. Hierfür steht **mouseY**. Durch die beiden obigen Zeilen können wir so unseren Potenzialberg beliebig drehen und ihn somit von allen Seiten betrachten.

Neu in diesem Sketch ist auch, dass wir dieses Mal keine Äquipotenziallinien zeichnen, sondern die Ebene um die Ladung Q mit Punkten abdecken. Die Berechnung der x-, y- und phi-Koordinaten (phi steht im Sketch für das Potenzial φ) dieser Punkte gelingt uns mit zwei **verschachtelten *for-Schleifen***. Für $\frac{Q}{4\pi\varepsilon_0\varepsilon_r}$ wurde der Wert 100 eingesetzt und die Pixelwerte für r wurden frei gewählt. Für das Zeichnen der Punkte müssen natürlich auch alle drei Koordinaten verwendet werden: *point(x, y, phi).*

In der zweiten Schleife sehen wir zum ersten Mal die Funktion **sqrt()**. *sqrt* steht für square root und heißt übersetzt Quadratwurzel.

Der rote Punkt für die positive Ladung müsste eigentlich genau auf der Kreuzung von x- und y-Achse liegen. Doch weil es so viel schöner aussieht, wurde er etwas angehoben:)

Sketch 05: Potenzialberg

```
void setup()
{
  size(800, 800, P3D); // P3D sorgt für eine 3D-Darstellung
}

void draw()
{
  background(255);
  translate (400, 500);

  // Rotation um die y-Achse bei einer Mausbewegung in x-Richtung
  rotateY(0.006*mouseX);
  // Rotation um die x-Achse bei einer Mausbewegung in y-Richtung
  rotateX(PI/2 + 0.006*mouseY);

  // Zeichnen einer x- und y-Achse
  stroke(200);
  strokeWeight(3);
  line(-300, 0, 300, 0);
  line(0, -300, 0, 300);

  // Zeichnen des Potenzialberges
  for (float x = -250; x <= 250; x = x + 7)
  {
    for (float y = -250; y <= 250; y = y + 7)
    {
      float r = 0.005*sqrt(x*x + y*y); // r wird mit Hilfe des Satzes
                                        // von Pythagoras berechnet
      float  phi = 100/r;

      stroke(0);
      strokeWeight(3);
      point(x, y, phi);
    }
  }

  // Zeichnen der positiven felderzeugenden Ladung
  stroke(255, 0, 0);
  strokeWeight(60);
  point(0, 0, 110);
}
```

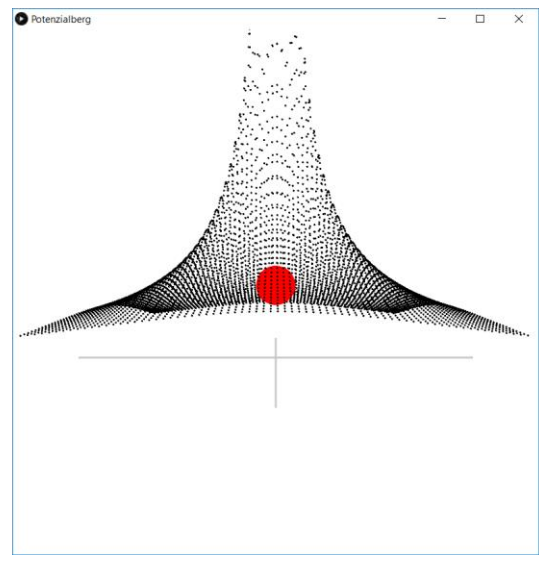


Abbildung 3.9: Potenzialberg über einer positiven Ladung

Array

Bevor wir die Darstellung eines Potenzialgebirges angehen, wollen wir zuerst etwas Neues lernen. Das Neue trägt den Namen **Array**.

Das Wort **Array** heißt übersetzt Ansammlung. Unter Array wollen wir hier einfachheitshalber mal eine Ansammlung von Kisten verstehen, die alle eine Nummer tragen, beginnend bei null. Auch wenn auf der letzten Kiste in Abbildung 3.10 eine 6 steht, so sind es doch 7 Kisten, da wir, wie in der Informatik üblich, mit dem Zählen bei null beginnen.

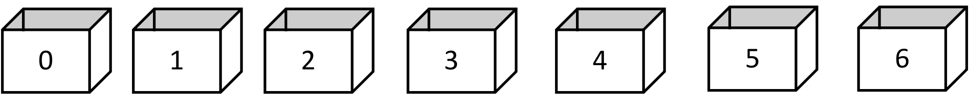


Abbildung 3.10: Ein Array von sieben Kisten

In Processing schreibt man für so eine Ansammlung von 7 Kisten:

```
float[] A = new float[7];
```

A ist der Name für unser Array. Die eckigen Klammern zeigen uns an, dass es sich hierbei um ein Array handelt. Mit *float* teilen wir mit, dass dieses Array die Eigenschaft besitzt, float-Zahlen aufnehmen zu können. Wir können also in jede Kiste eine *float*-Zahl, wie zum Beispiel 1.4, 2.3, usw. hineinstecken. Dies gelingt so:

```
A[0] = 1.4;
A[1] = 2.3;
A[2] = 6.8;
A[3] = 0.2;
A[4] = 1.1;
A[5] = 4.3;
A[6] = 5.0;
```

Mit *println(A)* können wir uns dann die Inhalte unserer Kisten in der Konsole anzeigen lassen.

[0] 1.4
[1] 2.3
[2] 6.8
[3] 0.2
[4] 1.1
[5] 4.3
[6] 5.0

Sieben Kisten mit der Hand zu befüllen geht noch. Wenn wir aber 700 Kisten haben, dann ist dies doch sehr lästig. Hier kann uns eine *for-Schleife* in Zusammenarbeit mit einer Gleichung helfen. Hier ist ein Beispiel für die Befüllung von 700 Kisten mit Quadratzahlen. Im Sketch steht **i++**. Dies ist eine kürzere Schreibweise für i = i + 1.

```
int[] quadratzahl = new int[700];
for (int i = 0; i < 700; i++)
{
quadratzahl[i] = i*i;
}
println(quadratzahl);
```

In der Konsole können wir uns das Ergebnis ansehen.

[0] 0
[1] 1
[2] 4
[3] 9
….

[697] 485809
[698] 487204
[699] 488601

Schnitt durch einen Potenzialtrichter mittels Array zeichnen

Versuchen wir nun mal mit unserem 700-Kisten-Array einen Schnitt durch einen Potenzialtrichter zu erstellen. Mit

```
float [] V = new float[700];
```

deklarieren und initialisieren wir unser Array. Mit der folgenden *for-Schleife* und der in ihr enthaltenen Gleichung befüllen wir das Array.

```
for (int r = 1; r < 700; r = r + 1)
{
      V[r] = k/r;
}
```

V ist das Formelzeichen für das Gravitationspotenzial $V(r) = -\gamma \cdot \frac{M}{r}$. In unserem Sketch haben wir k = γ·M gesetzt und durch 10^{11} geteilt, um so einen praktikablen Maßstab zu erhalten. Da die positive y-Achse in Processing nach unten zeigt, haben wir das Minuszeichen weggelassen. Dadurch erhalten wir einen Potenzialtrichter.

Unsere 700 Kisten haben die Nummern 0 bis 699. In unsere *for-Schleife* wird r initialisiert und in Einerschritten von 1 auf 699 vergrößert. Wir beginnen deshalb nicht bei null, damit wir nicht durch null teilen müssen. In der *for-Schleife* befindet sich noch eine *if-Anweisung*. Mit ihr begrenzen wir den Zeichnungsbereich für V(r).

Das Array hilft uns auch bei der Zeichnung des Graphen. Mit den folgenden Zeilen verbinden wir viele kurze Linien zum Graphen V(r), da V[r] und r sich gemeinsam ändern.

```
// Wir verbinden schrittweise einen Punkt mit seinem vorhergehenden
// Punkt
    line(r, V[r], r-1, V[r-1]);
    line(-r, V[r], -r+1, V[r-1]);
    line(-r, V[r], -r+1, V[r-1]); // Die Linien werden um die x-Achse
                                  // gespiegelt
```

Die restlichen Zeilen in dem folgenden Sketch sind altbekannte Zeichenübungen.

Sketch 06: Potenzialtrichter_mit_Array

```
float k = 6674.28; // k = Gamma*M im Maßstab 1 zu 10^-11.
                   // Also k = Gamma*M/10^11

size(700, 700);
background(255);
translate(350, 100);

float [] V = new float[700]; // Das Array mit dem Namen V enthält 700
                             // "Kisten"

// Die "Kiste" 1 des Arrays V befüllen wir bis "Kiste" 699 mit den
// Werten
// für das Gravitationspotenzial V
for (int r = 1; r < 700; r = r + 1)
{
  V[r] = k/r;

  // Zeichnen der V(r)-Funktion
  if (r>= 11 && V[r] < 550)
  {
    stroke(0, 0, 255);
    strokeWeight(3);
// Wir verbinden schrittweise einen Punkt mit seinem vorhergehenden
// Punkt
    line(r, V[r], r-1, V[r-1]);
    line(-r, V[r], -r+1, V[r-1]);
  }
}

// Zeichnen des Koordinatensystems
stroke(150);
line(0, 600, 0, -40);
line(-330, 0, 330, 0);
fill(150);
triangle(-5, -20, 0, -40, 5, -20);
triangle(310, -5, 330, 0, 310, 5);
triangle(-330, 0, -310, -5, -310, 5);
textSize(30);
text("V", 20, -30);
```

```
text("r", 310, -15);
text("r", -315, -15);

// Zeichnen des Planeten
stroke(0);
strokeWeight(2);
fill(0, 255, 0);
ellipse(0, 560, 25, 25);

println(V);
```

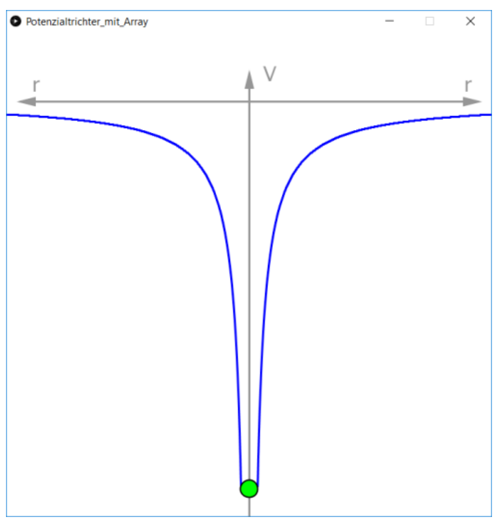


Abbildung 3.11: Schnitt durch einen Potenzialtrichter

Äquipotenziallinien mit einem Array zeichnen

Übung macht den Meister. Deshalb versuchen wir jetzt einmal mittels eines Arrays, die Äquipotenziallinien um unseren altbekannten Planeten mit der Masse M = 10^{25} kg zu zeichnen. Die Werte für das Potenzial V geben wir vor und berechnen hieraus wie folgt die zugehörigen Radien.

$V(r) = -\gamma \cdot \frac{M}{r} \Rightarrow r = -\gamma \cdot \frac{M}{V_{(r)}} = -\frac{k}{V_{(r)}}$ mit k = $6{,}67428 \cdot 10^{-11}$ $m^3kg^{-1}s^{-2} \cdot 10^{25}$ kg = $6{,}67428 \cdot 10^{14}$ m^3s^{-2}

Um handhabbare Werte zu erhalten, legen wir für k einen Maßstab von 1 : 10^{11} fest (siehe Sketch).

Kommen wir nun zu den wesentlichen Zeilen des Sketches. Zuerst überlegen wir uns, mit wie vielen konzentrischen Kreise wir unser radialsymmetrisches Gravitationsfeld darstellen wollen. Anschließend deklarieren und initialisieren wir unser Array mit dem Namen P.

```
float [] P = new float[35];
```

Danach erstellen wir mittels einer *for-Schleife* eine Zahlenreihe von 0 bis 34 für die 35 "Kisten" des Arrays P. Anstatt in der runden Klammer der *for-Schleife* i < 35 einzugeben, schreiben wir diesmal *i < P.length*. **length** steht für die Länge des Arrays. Wenn wir also auf die Idee kommen sollten, die Größe unseres Arrays zu ändern, dann passt sich die *for-Schleife* automatisch an. Dies ist sehr praktisch. Mit P[i] = V füllen wir die „Kisten“ mit den Werten für das Gravitationspotenzial V, nachdem wir vorher beschlossen haben, dass das Potenzial in 10er-Schritten (`int V = 10*i)` ansteigen soll.

```
for (int i = 0; i < P.length; i++)
{
  int V = 10*i; // Die Werte für das Potenzial werden berechnet
                //(0 bis 34)

  P[i] = V; // Jede der 35 "Kisten" wird mit einem Wert für das
            // Gravitationspotenzial V gefüllt
```

Mit diesen Potenzialwerten wird dann r berechnet.

```
float r = k/(P[V] + 0.00000001);
```

Mit dem Trick, einen kleinen Wert zu P[V] hinzu zu addieren, verhindern wir die Division durch Null, ohne das Ergebnis großartig zu verändern. Ein Blick in die Konsole zeigt dies (siehe unten). Der sehr große erste Wert spielt für unsere Zeichnung keine Rolle, da er außerhalb der Zeichenfläche liegt.

Konsolenwerte

~~V in 10^6 J/kg = -0.0   r in 10^5 m = 6.67428E11~~
V in 10^6 J/kg = -10.0   r in 10^5 m = 667.428
V in 10^6 J/kg = -20.0   r in 10^5 m = 333.714
……

V in 10^6 J/kg = -320.0   r in 10^5 m = 20.857124
V in 10^6 J/kg = -330.0   r in 10^5 m = 20.22509
V in 10^6 J/kg = -340.0   r in 10^5 m = 19.630234

Damit sind die wesentlichen Teile des Sketches erklärt. Der Rest ist wieder „Zeichnerei“.

Vielleicht fragt man sich, wo eigentlich die Vorteile in der Verwendung eines Arrays liegen? Nun, in unserem Beispiel kann man sehr komfortabel die Werte für das Potenzial V und die Anzahl der Äquipotenziallinien festlegen. Arrays sind immer dann von Vorteil, wenn man es mit einer Vielzahl von Werten zu tun hat.

Sketch 07: Aequipotenziallinien_mit_Array

```
float k = 6674.28; // Maßstab: k = Gamma*M/10^11

size(700, 700);
background(255);
translate(350, 350);

float [] P = new float[35]; // Array P mit 35 "Kisten" wird deklariert
                            // und initialisiert

// Zahlenreihe von 0 bis 34 erstellen
for (int i = 0; i < P.length; i++)
{
```

```
  int V = 10*i; // Die Werte für das Potenzial werden berechnet
                //(0 bis 34)

  P[i] = V; // Jede der 35 "Kisten" wird mit einem Wert für das
            // Gravitationspotenzial V gefüllt

  /* Berechnung des Abstandes r der Äquipotenziallinien vom Mittelpunkt
   des Planeten. Um die Division durch Null bei der Berechnung von r zu
   verhindern, vergrößern wir den Divisor um einen kleinen Wert */

  float r = k/(P[i] + 0.00000001);

  // Zeichnen der Äquipotenziallinien
  stroke(0, 0, 255);
  strokeWeight(1);
  ellipse(0, 0, 2*r, 2*r); // Die Werte 2*r stehen für die
                           // Kreisdurchmesser

  println("V in 10^6 J/kg =", -P[i], "   r in 10^5 m =", r);
}

// Zeichnen des Planeten
stroke(0);
strokeWeight(2);
fill(0, 255, 0);
ellipse(0, 0, 40, 40);
```

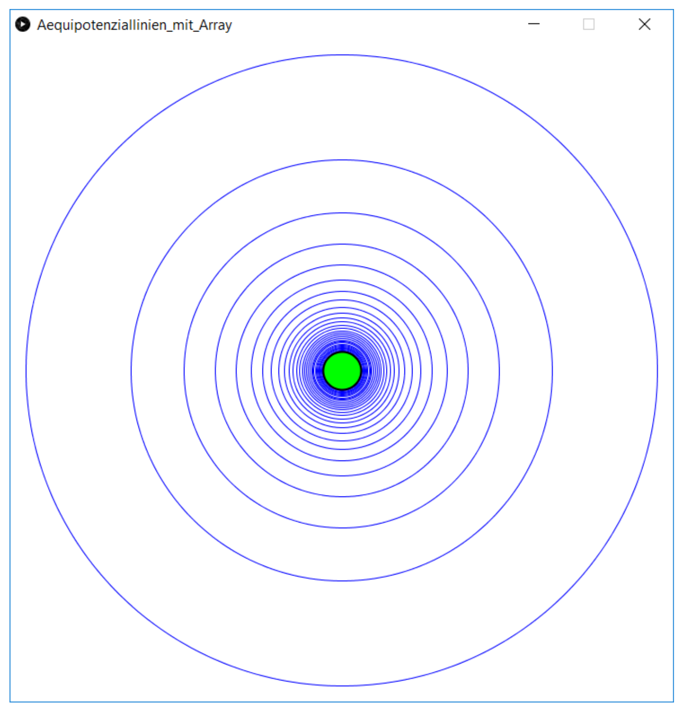


Abbildung 3.12: Mit einem Array erstellte Äquipotenziallinien

3.4 Potenzialgebirge

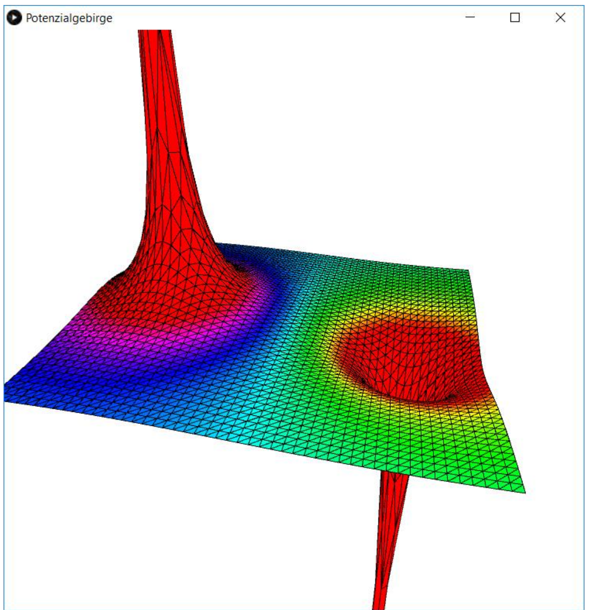


Abbildung 3.13: 3D-Darstellung von Potenzialberg und Potenzialtrichter

Wenn man die Kisten (siehe Abb. 3.10) nicht hintereinander aufstellt, sondern auf einer Fläche verteilt, dann spricht man von einem zweidimensionalen Array. Für ein zweidimensionales Array aus zum Beispiel 2500 Kisten schreibt man

```
float[][]B = new float[50][50];
```

Ein solches zweidimensionales Array von der Größe 50 x 50 wollen wir nun für die Erstellung eines Potenzialgebirges nutzen. In unser 50 x 50 großes Array setzen wir eine positive Ladung bei x = 10 und y = 25 und eine negative Ladung bei x = 40 und y = 25 (siehe Abb. 3.14). Die Potenziale der beiden Ladungen überlagern sich nach dem Superpositionsprinzip im gesamten Array-Bereich.

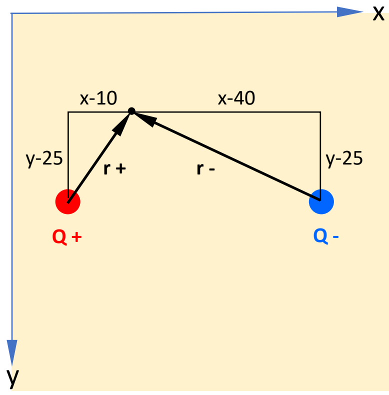


Abbildung 3.14: Lage der potenzialerzeugenden Ladungen

Wenn man für unsere Pixelwelt $\frac{1}{4\pi\varepsilon_0\varepsilon_r} = 1$ setzt, dann gilt entsprechend der Abbildung 3.14:

$$\varphi = \varphi_{positiv} + \varphi_{negativ} = \frac{Q_+}{r_+} + \frac{Q_-}{r_-}$$

Mit konkreten Zahlenwerten für die einzelnen Potenziale erhält man:

$$\varphi_{positiv} = \frac{50}{\sqrt{(x-10)^2+(y-25)^2+0{,}01}} \qquad \varphi_{negativ} = \frac{-50}{\sqrt{(x-40)^2+(y-25)^2+0{,}01}}$$

Um die Division durch Null zu verhindern, wurde in der Wurzel wieder ein kleiner Wert hinzuaddiert. In Processingschreibweise lautet die Gleichung für φ nun:

```
50/sqrt(pow(x - 10, 2) + pow(y - 25, 2) + 0.01) - 50/sqrt(pow(x - 40, 2)
+ pow(y - 25, 2) + 0.01);
```

Nun muss das zweidimensionale Array noch befüllt werden. Dazu rufen wir die einzelnen Array-Elemente mittels der doppelten *for-Schleife* nach und nach auf und ordnen dann jedem Array-Element mit der oben besprochenen Gleichung einen Potenzialwert zu.

```
for (int y = 0; y < 50; y++)
{
  for (int x = 0; x < 50; x++)
  {
    potenzialGebirge[x][y] = 50/sqrt(pow(x - 10, 2) + pow(y - 25, 2) +
    0.01) - 50/sqrt(pow(x - 40, 2) + pow(y - 25, 2) + 0.01);
  }
}
```

Nun müssen wir die 50 x 50 = 2500 Array-Werte noch grafisch darstellen. Dies gelingt uns mit dem folgenden Sketchteil. Teile hiervon müssen jedoch etwas näher erläutert werden.

```
colorMode(HSB, 360, 100, 100);

for (int y = 0; y < 50 - 1; y++)
{
  beginShape(TRIANGLE_STRIP);
  // Funktion beginShape()
  for (int x = 0; x < 50; x++)
  {
    fill(potenzialGebirge[x][y] * 50 + 180, 100, 100);
    vertex((x - 25) * skalierung, (y - 25) * skalierung,
    potenzialGebirge[x][y] * skalierung);
    vertex((x - 25) * skalierung, (y - 25 + 1) * skalierung,
    potenzialGebirge[x][y + 1] * skalierung);
  }
  endShape();
}
```

Um eine farblich ansprechende Darstellung zu erhalten, wählen wir mit **colorMode(HSB, 360, 100, 100)** den Farbraum HSB. Wie sich dieser vom Farbraum RGB unterscheidet, wird ausführlich im Kapitel 6.6 erläutert.

In der *for-Schleife* begegnet uns die Funktion **vertex().** Mit ihr kann man einen Eckpunkt für beliebige Formen (englisch Shape) festlegen. Damit Processing weiß, wo die Form beginnt und wo sie endet, muss man Processing dies mit **beginShape()** und **endShape()** mitteilen. Im obigen Sketchausschnitt sind die runden Klammern jedoch nicht leer, sondern innerhalb der Klammern steht TRIANGLE_STRIP. Die Funktion **beginShape(TRIANGLE_STRIP)** generiert zu vorgegebenen Punkten aneinanderhängende Dreiecke. Ein einfaches Beispiel zeigt Abbildung 3.15. Mit unserem zweidimensionalen Array generieren wir sehr viele Punkte mit den Koordinaten x, y und z. Da wir

eine 3D-Darstellung wünschen, besitzt jeder dieser Punkte Koordinaten im dreidimensionalen Raum. Diese Punkte sind dann die Eckpunkte von den zusammenhängenden Dreiecken, mit denen wir unser Potenzialgebirge dreidimensional grafisch darstellen. Die Größe dieser Dreiecke können wir mittels *float skalierung* (ganz oben im Sketch) festlegen.

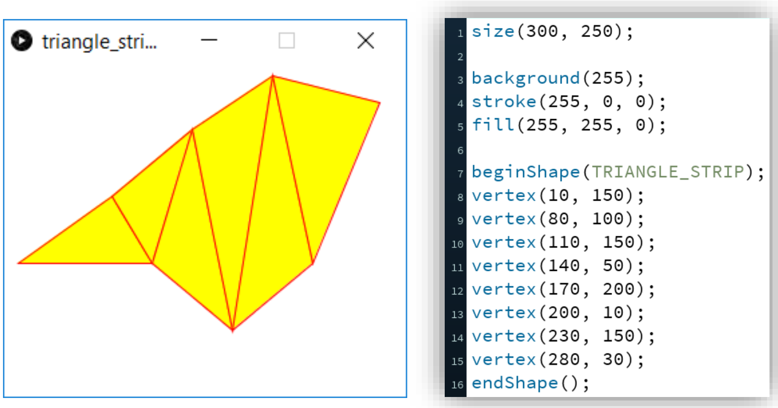


Abbildung 3.15: Zusammenhängende Dreiecke zeichen mit TRIANGLE_STRIP

void mouseDragged() und **void mouseWheel(MouseEvent event)** dienen dazu, das Potenzialgebirge im Fenster zu bewegen. Siehe hierzu die Erläuterungen im Sketch und in der Referenz von Processing unter dem Suchbegriff „mouse".

Sketch 08: Potenzialgebirge

```
// Potenzialgebirge

float skalierung = 10; // Gibt die Größe einer Gitterzelle an
float rotationX = PI/4; // Rotation um die x-Achse (Startwert entspricht
45°)
float rotationZ = 0; // Rotation um die z-Achse
float positionZ = 0; // Verschiebung entlang der z-Achse
float[][] potenzialGebirge = new float[50][50]; // zweidimensionales
Array // zur Speicherung der Potenzialwerte

void setup()
{
  size(750, 750, P3D);

  // Das zweidimensionale Array wird mit Potenzialwerten gefüllt
  for (int y = 0; y < 50; y++)
  {
    for (int x = 0; x < 50; x++)
    {
      potenzialGebirge[x][y] = 50/sqrt(pow(x - 10, 2) + pow(y - 25, 2) +
      0.01) - 50/sqrt(pow(x - 40, 2) + pow(y - 25, 2) + 0.01);
    }
  }
}

/* Die folgende Funktion wird aufgerufen, wenn die Maustaste gedrückt
   ist und die Maus bewegt wird */
void mouseDragged()
```

```
{
  /* Die Differenz der Mauskoordinaten (aktuelle Position und vorherige
   Position) wird verkleinert und zur Rotation entlang der x- bzw.
   z-Achse addiert */
  rotationX += (mouseY - pmouseY) * -0.01;
  rotationZ += (mouseX - pmouseX) * -0.01;
}

// Die folgende Funktion ändert die Werte durch das Drehen des Mausrades
void mouseWheel(MouseEvent event)
{
  // Die Bewegung des Mausrads wird zur Verschiebung entlang der z-Achse
  // addiert
  positionZ += event.getCount() * 100;
}

void draw()
{
  colorMode(RGB, 255, 255, 255); // Farbraum RGB für den Hintergrund
  background(255, 255, 255);
  stroke(0, 0, 0);
  strokeWeight(1);

  translate(width / 2, height / 2, 0); // Die Null dient hier als
            // Hinweis,dass die z-Koordinate nicht verändert wird
  rotateX(rotationX);
  rotateZ(rotationZ);
  translate(0, 0, positionZ); // Das Koordinatensystem kann mit dem
                // Mausrad in z-Richtung verschoben werden

  colorMode(HSB, 360, 100, 100); // Farbraum für das Potenzialgebirge

  // Zeichnet das Potenzialgebirge als eine mit Dreiecken dargestellte
  // Oberfläche
  for (int y = 0; y < 50 - 1; y++)
  {
    beginShape(TRIANGLE_STRIP); // Siehe Abbildung in der Referenz zur
                                // Funktion beginShape()
    for (int x = 0; x < 50; x++)
    {
      // Die Farben werden entsprechend der Potenzialgröße angepasst
      // (HSB)
      fill(potenzialGebirge[x][y] * 50 + 180, 100, 100);
      vertex((x - 25) * skalierung, (y - 25) * skalierung,
      potenzialGebirge[x][y] * skalierung);
      vertex((x - 25) * skalierung, (y - 25 + 1) * skalierung,
      potenzialGebirge[x][y + 1] * skalierung);
    }
    endShape();
  }
}
```

Potenzialgebirge animiert

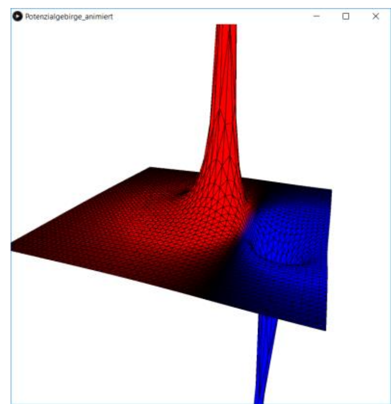


Abbildung 3.16: Animiertes 3D-Potenzial einer positiven und einer negativen Ladung

Den Sketch *Potenzialgebirge* wollen wir nun leicht abändern. Obwohl der HSB-Farbraum eine sehr schöne Farbverteilung geliefert hat, benutzen wir dieses Mal nur den RGB-Farbraum. Das positive Potenzial soll rot und das negative Potenzial blau dargestellt werden. Schwarz dargestellt wird der Bereich um den Wert Null.

Das interessante an dem neuen Sketch ist, dass sich die positive Ladung auf die negative Ladung zubewegt. Dies gelingt uns in der folgenden Zeile mit der Funktion *millis()*, die uns die Anzahl der Millisekunden angibt, die seit dem Programmstart vergangen sind.

```
potenzialGebirge[x][y] = 50/sqrt(pow(x - (10 + millis()/1000.0), 2)
+ pow(y - 25, 2) + 0.01) - 50/sqrt(pow(x - 40, 2) + pow(y - 25, 2) +
0.01);
```

Weitere Erläuterungen findet man im Text zum Sketch *Potenzialgebirge* und in den kommentierten Zeilen des neuen Sketches *Potenzialgebirge_animiert*.

Sketch 09: Potezialgebirge_animiert

```
// Potenzialgebirge animiert

float skalierung = 10; // Gibt die Größe einer Gitterzelle an
float rotationX = PI / 4; // Rotation um die x-Achse (Startwert
                          // entspricht 45°)
float rotationZ = 0; // Rotation um die z-Achse
float positionZ = 0; // Verschiebung entlang der z-Achse
float[][] potenzialGebirge = new float[50][50]; // zweidimensionales
                         // Array zur Speicherung der Potenzialwerte

void setup()
{
  size(750, 750, P3D);
}

/* Die folgende Funktion wird aufgerufen, wenn die Maustaste gedrückt
   ist und die Maus bewegt wird */
void mouseDragged()
{
/* Die Differenz der Mauskoordinaten (aktuelle Position und vorherige
Position) wird verkleinert und zur Rotation entlang der x- bzw. z-Achse
addiert */
  rotationX += (mouseY - pmouseY) * -0.01;
```

```
  rotationZ += (mouseX - pmouseX) * -0.01;
}

// Die folgende Funktion ändert die Werte durch das Drehen des Mausrades
void mouseWheel(MouseEvent event)
{
  /* Die Bewegung des Mausrads wird zur Verschiebung entlang der z-Achse
   addiert.
   Siehe Beispiel in der Processing-Referenz bei mouseWheel() */
  positionZ += event.getCount() * 100;
}
void draw()
{
  // Das zweidimensionale Array wird mit Potenzialwerten gefüllt
  for (int y = 0; y < 50; y++)
  {
    for (int x = 0; x < 50; x++)
    {
      // Die positive Ladung bewegt sich in Richtung negativer Ladung
      potenzialGebirge[x][y] = 50 / sqrt(pow(x - (10 + millis() /
      1000.0), 2) + pow(y - 25, 2) + 0.01) - 50 / sqrt(pow(x - 40, 2)
      + pow(y - 25, 2) + 0.01);
    }
  }

  background(255, 255, 255);
  stroke(0, 0, 0);
  strokeWeight(1);

  translate(width / 2, height / 2, 0); // Die Null dient als Hinweis,
                      // dass die z-Koordinate nicht verändert wird
  rotateX(rotationX);
  rotateZ(rotationZ);
  translate(0, 0, positionZ); // Das Koordinatensystem kann mit dem
                      // Mausrad in z-Richtung verschoben werden

  // Zeichnet das Potenzialgebirge als eine mit Dreiecken dargestellte
  // Oberfläche
  for (int y = 0; y < 50 - 1; y++)
  {
    beginShape(TRIANGLE_STRIP); // Siehe Abbildung in der Referenz zur
                                // Funktion beginShape()
    for (int x = 0; x < 50; x++)
    {
      // Die Farben werden entsprechend der Potenzialgröße angepasst
      // (RGB)
     fill(potenzialGebirge[x][y] * 255, 0, potenzialGebirge[x][y] * -255);
      vertex((x - 25) * skalierung, (y - 25) * skalierung,
      potenzialGebirge[x][y] * skalierung);
      vertex((x - 25) * skalierung, (y - 25 + 1) * skalierung,
      potenzialGebirge[x][y + 1] * skalierung);
    }
    endShape();
  }
}
```

3.5 Gravitationsgesetz

Gravitationsgesetz vektoriell 01

Wie in Kapitel 2.4.1 versprochen, wollen wir nun die Bewegung eines Planeten um seine Sonne vektoriell mittels des Gravitationsgesetzes simulieren. Die Betragsgleichung für das Gravitationsgesetz lautet $F_G = \gamma \cdot \frac{M \cdot m}{r^2}$. Daraus ergibt sich mit $F = m \cdot a$ für die Gravitationsbeschleunigung

$$a = \frac{F}{m} = \frac{F_G}{m} = \frac{\gamma \cdot M \cdot m}{m \cdot r^2} = \frac{\gamma \cdot M}{r^2}$$

Für die Masse der Sonne setzen wir M = $2 \cdot 10^{30}$ kg und fassen γ und M für unsere Simulation zur Konstante k zusammen.

$$k = \gamma \cdot M = 6{,}67428 \cdot 10^{-11} \frac{m^3}{kg \cdot s^2} \cdot 2 \cdot 10^{30} kg = 1{,}3349 \cdot 10^{20} \frac{m^3}{s^2}$$

Als Maßstab für k in unserer Simulation wählen wir 1 : 10^{18}. Also $133{,}49 \;\hat{=}\; 1{,}3349 \cdot 10^{20} \frac{m^3}{s^2}$. Für den Abstand zwischen Sonne und Planet wählen wir 1 : 10^9. D.h., 100 Pixel entsprechen 10^9 m.

Die Sonne setzen wir fix in den Mittelpunkt des Fensters. Dies entspricht zwar nicht ganz der Realität, da auch eine Sonne etwas um den Mittelpunkt ihres Systems kreist, es erleichtert uns aber vorerst die Programmierung. Dem Geschwindigkeitsvektor des Planeten geben wir eine Komponente in negative x-Richtung, damit der Planet um die Sonne kreist und nicht in sie hineinstürzt. Sinnvollerweise wurde der Startort des Planeten hierfür vertikal unter der Sonne gewählt.

```
PVector Sonne = new PVector(200, 200);
PVector Planet = new PVector(200, 300);
```

Die x-Werte für Sonne und Planet sind gleich. Die y-Werte unterscheiden sich um 100 Pixel.

Die einzelnen Programmschritte in dem folgenden Sketch sollten dir bekannt sein. Wenn nicht, dann schaue noch mal in das Kapitel *2.5 Einführung in die Vektorrechnung* und in das Kapitel *2.5.6 Zusammenfassung*. Eine Programmzeile verdient jedoch, erwähnt zu werden.

```
PVector a = PVector.sub(Sonne, Planet).normalize().mult(aB);
```

Hier werden drei Rechenschritte in einer Zeile durchgeführt. Zuerst wird die vektorielle Differenz zwischen Sonne und Planet berechnet, dann wird der Vektor auf 1 normiert und anschließend wird dieser Wert mit dem Betrag aB der Beschleunigung multipliziert. Damit erhält man den Beschleunigungsvektor.

Auch bei diesem Sketch sollte man nicht vergessen, mit den einzelnen Werten zu spielen. Was passiert, wenn wir die Masse der Sonne so vergrößern, dass gilt k = 230? Was passiert, wenn wir die Geschwindigkeit des Planeten von – 1.0 auf – 2.0 erhöhen?

Sketch 10: Gravitationsgesetz_01

```
float k = 133.49; // k = Gamma*M im Maßstab 1:10^18
float t = 1; // Zeit

// Der Abstand von 100 Pixel zwischen Sonne und Planet entspricht 10^9 m
PVector Sonne = new PVector(200, 200); // Ortsvektor der Sonne
```

```
PVector Planet = new PVector(200, 300); // Ortsvektor des Planeten
PVector v = new PVector(-1.0, 0); // Geschwindigkeit des Planeten

void setup()
{
  size(400, 400);
}

void draw()
{
  background(255);

  /* Der Abstand r zwischen Sonne und Planet und der Betrag aB der
   Gravitationsbeschleunigung werden berechnet
   Um die Division durch 0 zu verhindern, wird ein kleiner Wert auf die
   Distanz addiert */
  float r = Sonne.dist(Planet) + 1;
  float aB = k/(r*r);

  /* Zuerst wird die vektorielle Differenz zwischen Sonne und Planet
   berechnet, dann auf 1 normalisiert und anschließend mit dem Betrag aB
   der Beschleunigung multipliziert */
  PVector a = PVector.sub(Sonne, Planet).normalize().mult(aB);

  v.add(PVector.mult(a, t)); // a wird mit t multipliziert und zu Vektor
                             // v addiert
  Planet.add(PVector.mult(v, t)); // v wird mit t multipliziert und zu
                                  // dem Ortsvektor Planet addiert

  // Sonne
  stroke(255, 0, 0);
  fill(255, 200, 0);
  ellipse(Sonne.x, Sonne.y, 40, 40);

  // Planet
  stroke(100);
  strokeWeight(2);
  fill(100, 100, 255);
  ellipse(Planet.x, Planet.y, 15, 15);
}
```

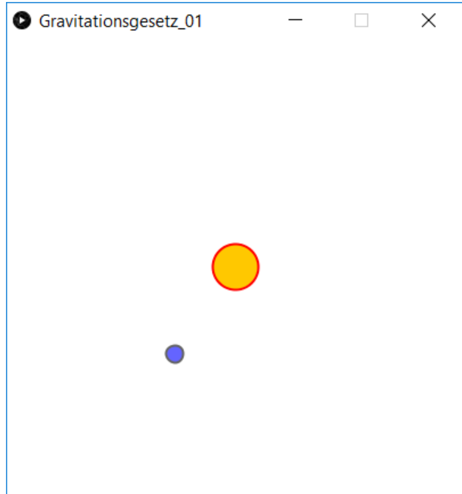


Abbildung 3.17: Ein Planet kreist um die Sonne

Gravitationsgesetz vektoriell 02

Nun wollen wir etwas grundlegend Neues lernen. Wir lernen, was man in Processing unter einer **Klasse** versteht. Dies ist der Einstieg in die **objektorientierte Programmierung**. Kurz: **OOP**. Dies ist ein recht komplexes Konzept, doch wir gehen es ganz einfach an. Dazu benutzen wir unseren vorhergehenden Sketch *Gravitationsgesetz_01* und schreiben ihn so um, wie man es in Processing von einer *Klasse* erwartet. Da wir den Sketch *Gravitationsgesetz_01* verstanden haben (Hoffentlich!), können wir uns voll darauf konzentrieren, was eine *Klasse* ausmacht.

Obwohl die Mitglieder einer Klasse viele unterschiedliche Eigenschaften besitzen können, so haben sie doch auch einige grundlegende Eigenschaften gemeinsam. Nehmen wir als Beispiel die Klasse der Planeten. Alle Planeten haben die folgenden grundlegenden Eigenschaften gemeinsam. Ihre Masse ist so groß, dass sie eine annähernde Kugelform besitzen, aber nicht so groß, dass sie infolge von Kernfusion selber leuchten. Weiterhin kreisen sie um einen Fixstern (Sonne) und sie haben ihre Umlaufbahn aufgrund ihrer Gravitationskraft freigeräumt. Trotzdem unterscheiden sich die einzelnen Planeten untereinander. Der Saturn hat andere Eigenschaften als die Erde und diese hat andere Eigenschaften als die Venus. Die Eigenschaften der Mitglieder einer Klasse nennt man in Processing **Instanzvariablen**.

Wie erzeugt man nun in Processing eine Klasse? Hier ein Überblick über die wesentlichen Schritte, die danach anhand unseres Sketches näher erklärt werden.

KLASSENNAME

{

INSTANZVARIABLEN

KONSTRUKTOR

METHODEN (FUNKTIONEN)

}

Damit unser Sketch *Gravitationsgesetz_02* übersichtlicher wird, schreiben wir die Klasse nicht auf die Karteikarte Gravitationsgesetz_*02* sondern auf eine extra Karteikarte mit dem Namen *Planet* (siehe Abb. 3.18). Um eine neue Karteikarte anzulegen, klickt man auf das kleine Dreieck in der PDE (siehe Abb. 3.18).

Abbildung 3.18: Ausschnitt aus der PDE von Processing

Den Sketch auf der Karteikarte Gravitationsgesetz_*02* nennen wir **Hauptsketch**. Den Sketch auf der Karteikarte *Planet* nennen wir **Nebensketch**. Schauen wir uns zuerst den Nebensketch Schritt für Schritt an. Zuerst geben wir der Klasse einen Namen und setzen die offene und geschlossene geschweifte Klammer. Alles was wir danach schreiben muss innerhalb dieser Klammern stehen.

```
class Planet
  {
  }
```

Klassennamen beginnen traditionell immer mit einem Großbuchstaben.

Nun werden die Instanzvariablen genannt. In unserem einfachen Fall ist es nur eine.

```
// INSTANZVARIABLEN
float k = 133.49; // k = Gamma*M im Maßstab 1:10^18
```

Danach konstruieren wir die einzelnen Objekte der Klasse. Aus diesem Grund heißt dieser Programmabschnitt Konstruktor.

```
// KONSTRUKTOR
PVector Sonne = new PVector(200, 200);
PVector Planet = new PVector(200, 300);
PVector v = new PVector(-1.0, 0);
float t = 1;
```

Den so erschaffenen Objekten fügen wir nun die entsprechenden Eigenschaften zu. Mit der Funktion *void move()* legen wir fest, wie die Bedingungen für die Bewegung des Planeten sind. Mit der Funktion *void display()* legen wir die Art der Darstellung von Sonne und Planet fest. Die Namen für diese Funktionen, die man auch Methoden nennt, sind frei wählbar. Man schreibt sie üblicherweise immer klein.

```
// Methoden(FUNKTIONEN)

void move()
{
  float r = Sonne.dist(Planet) + 1;
  float aB = k/(r*r);
  PVector a = PVector.sub(Sonne, Planet).normalize().mult(aB);
  v.add(PVector.mult(a, t));
  Planet.add(PVector.mult(v, t));
}

void display()
{
  stroke(255, 0, 0);
  fill(255, 200, 0);
  ellipse(Sonne.x, Sonne.y, 40, 40);

  stroke(100);
  strokeWeight(2);
  fill(100, 100, 255);
  ellipse(Planet.x, Planet.y, 15, 15);
}
```

Fassen wir den Ablauf nochmal zusammen.

KLASSENNAME

{

INSTANZVARIABLEN

KONSTRUKTOR

METHODEN (FUNKTIONEN)

}

Vom Hauptsketch Gravitationsgesetz_*02* wollen wir nun auf die Klasse Planet zugreifen. Wie gelingt dies? Als Erstes geben wir unserem Planeten den Namen Horst und sagen, dass er zur Klasse Planet gehört.

```
Planet Horst;
```

Dies ist so ähnlich wie bei unseren primitiven Variablen. Zum Beispiel: int x;

Danach teilen wir Processing bei *void setup()* mit, dass Horst ein neuer Planet ist.

```
Horst = new Planet();
```

Letztendlich rufen wir bei *void draw()* die Funktionen aus der Klasse Planet für Horst auf und starten unseren Sketch.

```
Horst.move();
Horst.display();
```

Das Ergebnis ist das gleiche wie bei Sketch Gravitationsgesetz_*01*. Natürlich fragt man sich jetzt, warum wir diesen ganzen komplizierten Aufwand betrieben haben. Nun, der Sketch Gravitationsgesetz_*02* diente nur dazu, um auf möglichst einfache Art und Weise zu erklären, wie eine Klasse aufgebaut ist und wie man sie nutzen kann. Welche großen Vorteile Klassen bieten, erfahren wir erst bei deutlich komplexeren Programmen als beim Programm Gravitationsgesetz_*02*. Unser nächster Sketch *Sonnensystem* ist schon ein guter Weg in diese Richtung.

Sketch 11: Gravitationsgesetz_02

Hauptsketch

```
Planet Horst; // Unser Planet aus der Klasse Planet bekommt den Namen
Horst

void setup()
{
  size(400, 400);
  Horst = new Planet(); // Planet Horst wird der Klasse Planet
zugeordnet
}

void draw()
{
  background(255);
  // Die Methoden (Funktionen) aus der Klasse Planet werden aufgerufen
  Horst.move(); // Planet Horst bewegt sich entsprechend den Eigen-
                // schaften seiner Klasse
  Horst.display(); // Die Darstellung von Horst erfolgt entsprechend
                   // den Eigenschaften seiner Klasse
}
```

Nebensketch

```
class Planet
{
  // INSTANZVARIABLEN

  float k = 133.49; // k = Gamma*M im Maßstab 1:10^18

  // KONSTRUKTOR
```

```
  // Der Abstand von 100 Pixel zwischen Sonne und Planet entspricht
  // 10^9 m
  PVector Sonne = new PVector(200, 200); // Ortsvektor der Sonne
  PVector Planet = new PVector(200, 300); // Ortsvektor des Planeten
  PVector v = new PVector(-1.0, 0); // Geschwindigkeit des Planeten
  float t = 1; // Zeit

  // METHODEN (FUNKTIONEN)

  void move()
  {
    /* Der Abstand r zwischen Sonne und Planet und der Betrag aB der
       Gravitationsbeschleunigung werden berechnet.
       Um die Division durch 0 zu verhindern, wird ein kleiner Wert
       auf die Distanz addiert */
    float r = Sonne.dist(Planet) + 1;
    float aB = k/(r*r);

    /* Zuerst wird die vektorielle Differenz zwischen Sonne und Planet
       berechnet, dann auf 1 normalisiert und anschließend mit dem
       Betrag aB der Beschleunigung multipliziert. */
    PVector a = PVector.sub(Sonne, Planet).normalize().mult(aB);

    v.add(PVector.mult(a, t)); // a wird mit t multipliziert und zu
                               // Vektor v addiert
    Planet.add(PVector.mult(v, t)); // v wird mit t multipliziert und zu
                                    // dem Ortsvektor Planet addiert
  }

  void display()
  {
    // Sonne
    stroke(255, 0, 0);
    fill(255, 200, 0);
    ellipse(Sonne.x, Sonne.y, 40, 40);

    // Planet
    stroke(100);
    strokeWeight(2);
    fill(100, 100, 255);
    ellipse(Planet.x, Planet.y, 15, 15);
  }
}
```

Abbildung 3.19: Ein Planet kreist um seine Sonne

Sonnensystem

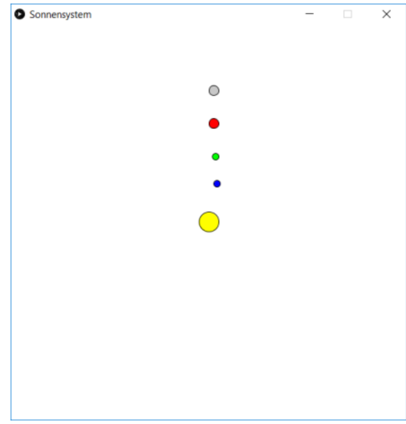


Abbildung 3.20: Planetenstellung kurz nach dem Start der Simulation

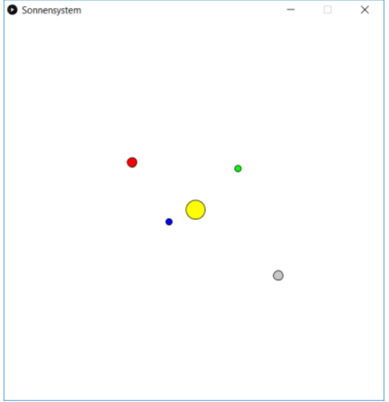


Abbildung 3.21: Vier Planeten umkreisen ihre Sonne

In dem nun folgenden Sketch werden wir erfahren wie leistungsfähig das Arbeiten mit Klassen ist. Schauen wir uns zuerst den Nebensketch an (siehe unten). Die Klasse bekommt den Namen *Gravi*. Danach werden die Instanzvariablen aufgelistet. In dem unten aufgeführten Sketchausschnitt sehen wir, welche Größen sich hinter den aufgeführten Buchstaben verbergen.

Im Konstruktor werden die Instanzvariablen nochmal aufgelistet. Warum? Die Größen, die unter *Instanzvariablen* aufgeführt sind, gelten für alle möglichen Himmelskörper. Im Konstruktor teilen wir Processing mit, dass es für diese Instanzvariablen die temporären Werte verwenden soll, die wir in unseren folgenden Methoden (Funktionen) und im Hauptsketch konkret verwenden werden.

```
class Gravi
{
  // INSTANZVARIABLEN
  float m; // Masse der Himmelskörper
  float D; // Durchmesser der Himmelskörper
  PVector c;// Farbe der Himmelskörper
  PVector r; // Ortsvektor der Himmelskörper
  PVector v; // Geschwindigkeit der Himmelskörper
```

```
PVector a = new PVector(); // a hat den Startwert (0, 0)

// KONSTRUKTOR
Gravi(PVector rTemp, PVector vTemp, float mTemp, float DTemp, PVector cTemp)
{
  r = rTemp;
  v = vTemp;
  m = mTemp;
  D = DTemp;
  c = cTemp;
}
```

Nun werden drei Methoden (Funktionen) aufgeführt. Eine für die Kraft, eine für die Bewegung und eine für die optische Darstellung.

```
void force(Gravi P, int G)
void move(float t)
void display()
```

Schauen wir uns die Methoden nun im Einzelnen an. Beginnend mit *void force(Gravi P, float G).*

```
void force(Gravi P, float G)
{
  PVector n = PVector.sub(P.r, r);
  float d = n.mag() + 1;
  n.normalize();
  PVector F = PVector.mult(n, G*m*P.m/(d*d));
  a.add(PVector.div(F, m));
}
```

Im Unterschied zu den uns schon bekannten Methoden void setup() und void draw() ist die runde Klammer bei *void force(Gravi P, float G)* nicht leer. In ihr werden die Größen P und G definiert. P ist der Name für einen beliebigen Planeten P aus der Klasse Gravi. G entspricht der Gravitationskonstante. P wechselwirkt gravitativ mit allen anderen Objekten im Sonnensystem. Dies organisiert das Array im Hauptsketch.

r ist der Ortsvektor des betracheten Himmelskörpers und P.r steht für den Ortsvektor des Wechselwirkungspartners von P. n ist der vektorielle Abstand zwischen P und dem hier betrachteten Himmelskörper. Mit float d wird der Betrag von Vektor $\vec{n}$ berechnet. Die Zahl 1 wird hinzuaddiert, um in der übernächsten Zeile die Division durch null zu verhindern.

Mit *n.normalize()* wird der Vektor $\vec{n}$ auf 1 normiert. Er wird zum Einheitsvektor. Anschließend wird der Einheitsvektor mit dem Betrag der Gravitationskraft multipliziert, um so den Vektor für die Kraft $\vec{F}$ zu erhalten. Zu guter Letzt wird der neue Beschleunigungsvektor $\vec{a}$ berechnet und zu dem bestehenden Vektor $\vec{a}$ hinzuaddiert.

Kommen wir nun zur Methode *void move(float t)*.

```
void move(float t)
{
  v.add(PVector.mult(a, t));
  r.add(PVector.mult(v, t));
  a.set(0, 0);
}
```

Auch hier ist die runde Klammer nicht leer. In ihr wird die Größe t für die Zeit definiert. Ihr konkreter Zahlenwert kann im Hauptsketch flexibel gewählt werden. Der Rest dürfte soweit klar sein, bis auf ***a.set(0, 0).*** An dieser Stelle wird der Vektor $\vec{a}$ wieder auf den Wert 0 gesetzt. Sonst würden sich die $\vec{a}$-Werte laufend addieren.

Die letzte Methode ist *void display().*

```
void display()
{
  fill(c.x, c.y, c.z);
  ellipse(r.x, r.y, D, D);
}
```

Die Farbe haben wir bei den Instanzvariablen sinnvollerweise als Vektor definiert, da sie drei Komponenten enthält. Diese sind nicht x, y und z sondern rot, grün und blau im Bereich von 0 bis 255. Den Wert für die Farbkomponenten unserer Planeten wählen wir wieder im Hauptsketch. Ebenso den Startort r.x und r.y sowie den Durchmesser D.

Nun wird es Zeit, sich dem Hauptsketch zu widmen. Hier erleben wir, dass sich die Mühe bei der Erstellung der Klasse *Gravi* wirklich gelohnt hat. Mit

```
Gravi[] Planeten = new Gravi[5];
```

erzeugen wir ein Array für Himmelskörper mit dem Namen Planeten aus der Klasse *Gravi*. Die Anzahl der Planeten können wir nun recht komfortabel festlegen, da ihre gemeinsamen Grundeigenschaften durch die Methoden in der Klasse *Gravi* schon festgelegt sind. Wir haben uns für 5 Planeten entschieden. Wir hätten aber auch drei oder 10 wählen können. Den Planeten [0] reservieren wir für die Sonne.

In *void setup()* legen wir nun die ganz konkreten Werte für unsere fünf Himmelskörper fest. Auch hier haben wir wieder die freie Auswahl. Wir müssen nur die Reihenfolge beachten, die wir im Konstrutor der Klasse *Gravi* festgelegt haben. Die Reihenfolge im Konstruktor ist:

```
Gravi(PVector rTemp, PVector vTemp, float mTemp, float DTemp, PVector cTemp)
```

Also: Ortsvektor $\vec{r}$, Geschwindigkeitsvektor $\vec{v}$, Masse m, Durchmesser der Himmelskörper D und Farbe $\vec{c}$ als Vektor.

```
void setup()
{
  size(600, 600);
  Planeten[0] = new Gravi(new PVector(300, 300), new PVector (0, 0), 2000, 30, new PVector (255, 255, 0)); // Sonne
  Planeten[1] = new Gravi(new PVector(300, 240), new PVector (5, 0), 2, 10, new PVector (0, 0, 255)); // blauer Planet
  Planeten[2] = new Gravi(new PVector(300, 200), new PVector (4, 0), 2, 10, new PVector (0, 255, 0)); // grüner Planet
  Planeten[3] = new Gravi(new PVector(300, 150), new PVector  (3, 0), 3, 15, new PVector (255, 0, 0)); // roter Planet
  Planeten[4] = new Gravi(new PVector(300, 100), new PVector  (3, 0), 4, 15, new PVector (200, 200, 200)); // grauer
Planet
}
```

In *void draw()* rufen wir mit einer verschachtelten for-Schleife Funktionen der Planeten aus unserem Array auf. Die hier aufgeführten Bedingungen *i < Planeten.length* und *j < Planeten.length* machen uns unabhängig von der Anzahl der gewählten Planeten. Mit *length* erfolgt eine automatische Anpassung an die Größe des Arrays.

```
void draw()
{
  background(255);
  for (int i = 0; i < Planeten.length; i++)
```

```
  {
    for (int j = 0; j < Planeten.length; j++)
    {
      if (i == j)
        continue;

      Planeten[i].force(Planeten[j], 1);
    }
  }

  for (int i = 0; i < Planeten.length; i++)
  {
    Planeten[i].move(0.1);
    Planeten[i].display();
  }
}
```

Welche Funktion hat das Wort **continue** im oberen Sketchausschnitt? Für i == j würde der Himmelskörper mit sich selber wechselwirken. Mit *continue* vermeiden wir dies. Die Operation i == j wird so einfach ausgelassen.

Nach dem Durchlaufen der beiden verschachtelten for-Schleifen wird eine dritte for-Schleife aufgerufen. Bei *Planeten[i].move(0.1)* können wir die Zeitschritte für die Simulation frei wählen. Für kleine t-Werte (hier t = 0.1) erhält man eine stabile Simulation. Wenn man t vergrößert, dann läuft alles schneller ab. Änderungen im Sonnensystem machen sich dann aber auch schneller bemerkbar.

Bevor wir uns nun den gesamten Sketch ansehen, möchte ich nochmal darauf hinweisen, dass man dank der erzeugten Klasse *Gravi* sehr komfortabel mit unserem Sonnensystem spielen kann. Man kann im Hauptsketch nicht nur ganz leicht die Anzahl der Planeten ändern, sondern auch ihre Eigenschaften. Was passiert, wenn wir die Masse eines Planeten deutlich erhöhen? Wie reagieren die anderen Planeten und die Sonne darauf? Die Sonne ist ja nicht ortsfest, sondern unterliegt auch der Gravitationswechselwirkung. Was passiert, wenn wir die Masse eines Planeten so großmachen wie die Masse der Sonne? Können in einem solchen Doppelsternsystem die anderen Planeten noch existieren? Also: Spielen! Spielen! Spielen!

Sketch 12: Sonnensystem

Hauptsketch

```
Gravi[] Planeten = new Gravi[5]; // Array für Himmelskörper mit dem
                                 // Namen Planet aus der Klasse Gravi

void setup()
{
  size(600, 600);
  // Nun legen wir die ganz konkreten Werte für unsere Himmelskörper
  // fest Reihenfolge der Werte: Ortsvektor r, Geschwindigkeitsvektor v,
  // Masse m, Durchmesser der Himmelskörper D, Farbe c
  Planeten[0] = new Gravi(new PVector(300, 300), new PVector (0, 0),
  2000, 30, new PVector (255, 255, 0)); // Sonne
  Planeten[1] = new Gravi(new PVector(300, 240), new PVector (5, 0), 2,
  10, new PVector (0, 0, 255)); // blauer Planet
  Planeten[2] = new Gravi(new PVector(300, 200), new PVector (4, 0), 2,
  10, new PVector (0, 255, 0)); // grüner Planet
```

```
  Planeten[3] = new Gravi(new PVector(300, 150), new PVector  (3, 0), 3,
  15, new PVector (255, 0, 0)); // roter Planet
  Planeten[4] = new Gravi(new PVector(300, 100), new PVector  (3, 0), 4,
  15, new PVector (200, 200, 200)); // grauer Planet
}

void draw()
{
  background(255);
  for (int i = 0; i < Planeten.length; i++)
  {
    for (int j = 0; j < Planeten.length; j++)
    {
      if (i == j)
        continue; /* Für i == j würde der Himmelskörper mit sich selber
                     wechselwirken. Mit continue vermeiden wir dies.
                     Die Operation i == j wird einfach ausgelassen.
                     In der Referenz steht: When run inside of a for or
                     while, it skips the remainder of the block and
                     starts the next iteration */

      Planeten[i].force(Planeten[j], 1); // Die Gravitationskonstante G
                                         // wurde hier gleich 1 gesetzt
    }
  }

  for (int i = 0; i < Planeten.length; i++)
  {
    Planeten[i].move(0.1); // Für kleine t-Werte (hier t = 0.1) erhält
                           // man eine stabile Simulation
    Planeten[i].display();
  }
}
```

Nebensketch

```
class Gravi
{
  // INSTANZVARIABLEN
  float m; // Masse der Himmelskörper
  float D; // Durchmesser der Himmelskörper
  PVector c;// Farbe der Himmelskörper
  PVector r; // Ortsvektor der Himmelskörper
  PVector v; // Geschwindigkeit der Himmelskörper
  PVector a = new PVector(); // a hat den Startwert (0, 0)

  // KONSTRUKTOR
  Gravi(PVector rTemp, PVector vTemp, float mTemp, float DTemp,
  PVector cTemp)
  {
    /* Die obigen Instanzvariablen gelten für alle möglichen Himmels-
       körper.Nun teilen wir Processing mit, dass es für die
       Instanzvariablen die temporären Werte verwenden soll, die wir in
       unseren folgenden Methoden verwenden */
    r = rTemp;
    v = vTemp;
    m = mTemp;
    D = DTemp;
    c = cTemp;
  }
```

```
  // METHODEN (FUNKTIONEN)
  /* Die Reihenfolge der Methoden ist hier im Nebensketch nicht von
     Bedeutung. Wohl aber bei void draw() im Hauptsketch */

  void force(Gravi P, float G) /* P ist der Name für einen beliebigen
  Planeten aus der Klasse Gravi. G entspricht der Gravitations-
  konstante */
  {
    PVector n = PVector.sub(P.r, r); /* r ist der Ortsvektor des nun
     betrachteten Himmelskörpers.
     P.r steht für den Ortsvektor des Wechselwirkungspartners P.
     n ist der vektorielle Abstand zwischen P und dem hier betrachteten
     Himmelskörper */

    float d = n.mag() + 1; /* d ist der Betrag des Vektors n. Um die
    Divison durch null zu verhindern wird ein kleiner Wert (hier 1)
    hinzuaddiert.*/
    n.normalize(); // Der Vektor n wird auf 1 normalisiert. Er wird zum
                   // Einheitsvektor

    PVector F = PVector.mult(n, G*m*P.m/(d*d)); /* m ist die Masse des
    hier betrachteten Himmelskörpers. P.m ist die Masse des
    Wechselwirkungspartners P */

    a.add(PVector.div(F, m)); // Die neue Beschleunigung wird berechnet
                              // und zum Vektor a addiert.
  }

  void move(float t) // Der konkrete t-Wert kann im Hauptsketch
                     // eingegeben werden.
  {
    v.add(PVector.mult(a, t));
    r.add(PVector.mult(v, t));
    a.set(0, 0); // Setzt den Vektor a wieder auf den Wert 0. Sonst
                 // würden sich die a-Werte laufend addieren
  }

  void display()
  {
    fill(c.x, c.y, c.z); // Die konkreten Werte werden im Hauptsketch
                         // eingegeben.
    ellipse(r.x, r.y, D, D); // Die konkreten Werte werden im
                             // Hauptsketch eingegeben.
  }
}
```

Doppelsternsystem

Zum Schluss dieses Kapitels noch ein ganz einfacher Sketch zur Entspannung. Ohne Array, ohne Vektor und ohne Klasse. In diesem einfachen Sketch soll ein Doppelsternsystem, bestehend aus einem roten Riesen und einem hellen begleitenden Stern dargestellt werden. Natürlich wollen wir hierbei auch etwas Neues lernen.

Mit P3D erzeugen wir eine dreidimensionale Darstellung. Mit *mouseX* können wir mittels der Mausbewegung in x-Richtung unseren roten Riesen und seinen Begleiter um die y-Achse bewegen. Wenn wir nun den roten Riesen mit *ellipse()* zeichnen, dann sieht er nach einer 90°-Drehung recht flach aus. Er hat nicht das Aussehen einer Kugel, sondern das einer Scheibe. Mit

sphere(50) können wir aber eine richtige Kugel mit dem Radius von 50 Pixel erzeugen, die jedoch stets im Koordinatenursprung sitzt. Ein besonders eindrucksvoller Effekt ergibt sich, wenn wir diese Kugel mit der Funktion **lights()** so beleuchten lassen, als würde diese Beleuchtung von seinem hellen Begleiter erfolgen. Viel Spaß beim Ausprobieren.

Sketch 13: Doppelstern

```
void setup()
{
  size(800, 400, P3D);
}

void draw()
{
  background(0);
  translate(400, 200);
  rotateY(0.01*mouseX); // Rotation um die y-Achse durch die Bewegung
                        // der Maus in x-Richtung

  // roter Riese
  noStroke();
  fill(255, 0, 0);
  lights(); // Beleuchtung
  sphere(50); // Radius der Kugel (roter Riese)

  // hell leuchtender Stern
  stroke(255);
  strokeWeight(30);
  point(0, 0, 200); // x-, y- und z-Koordinate
}
```

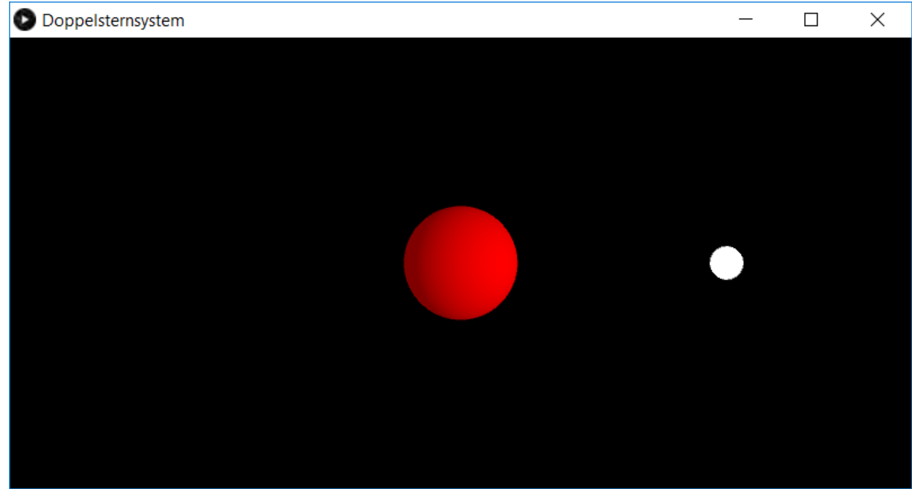


Abbildung 3.22: Roter Riese und sein Begleiter

3.6 Zusammenfassung

Potenzen und Wurzel

E E steht für die Basis 10 2.5E9 bedeutet $2{,}5 \cdot 10^9$

exp() exp steht für die Basis e = 2,7182817 exp(2) bedeutet e^2 = 7.389056

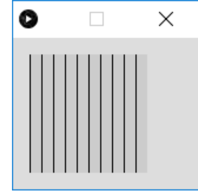
pow() Beispiel: pow(5, 3) steht für 5^3 = 125

sqrt() sqrt() steht für square root und heißt übersetzt Quadratwurzel, sqrt(9) = 3

for-Schleife for (Deklarierung und Initialisierung; Bedingung; Auftrag)

Beispiel:

```
for (int x = 0; x <= width; x = x + 10)
{
   line(x, 0, x, height);
}
```



Beachten muss man, dass die Variable (im Beispiel x) in der for-Schleife eine **lokale Variable** ist. D. h., sie besitzt nur innerhalb der for-Schleife ihre Gültigkeit.

width Breite des Fensters

height Höhe des Fensters

mouseX mouseX ist eine Systemvariable mit dem x-Wert des Mauszeigers im Sketch-Fenster. Sie wird von Processing selber erzeugt.

mouseY mouseY ist eine Systemvariable mit dem y-Wert des Mauszeigers im Sketch-Fenster.
Sie wird von Processing selber erzeugt.

Array Das Wort Array heißt übersetzt Ansammlung. Hier ein Beispiel:

```
float[] P = new float[4];
```

P ist der von uns gewählte Name für das Array. Die eckigen Klammern zeigen uns an, dass es sich hierbei um ein Array handelt. Mit *float[4]* teilen wir mit, dass dieses Array die Eigenschaft besitzt, vier *float*-Zahlen aufnehmen zu können.

Dies ist die einfachste Art, ein Array zu füllen: A[0] = 1.4; A[1] = 2.3; A[2] = 6.8; A[3] = 0.2;

Wenn wir jedoch sehr viele Zahlen in ein Array schreiben wollen, dann ist eine *for-Schleife* in Zusammenarbeit mit einer Gleichung sehr hilfreich.

length steht für die Länge des Arrays.
Hier ein Beispiel für ein Array mit dem Namen P:

```
for (int i = 0; i < P.length; i++)
```

Für ein **zweidimensionales Array** schreibt man: `int[][]B = new int[4][4];`

Für ein **dreidimensionalen Array** schreibt man: `int[][][]C = new int[4][4][4];`

i++ Die Kurzschreibweise i++ steht für i = i + 1.

+= Die Kurzschreibweise a += b steht für a = a + b

Wenn man in einem Sketch die Divison durch Null verhindern will, dann kann man zum Divisor einen sehr kleinen Wert hinzuaddieren. Dieser Zusatz darf das Ergebnis aber nur unwesentlich verändern.

Klasse In einer Klasse können mehrere Objekte erzeugt werden, die die gleichen Grund-eigenschaften besitzen. Mithilfe der in der Klasse enthaltenen Methoden (Funktionen) können die Eigenschaften der Objekte festgelegt werden.

Eine Klasse ist wie folgt aufgebaut:

```
KLASSENNAME
{
INSTANZVARIABLEN
KONSTRUKTOR
METHODEN (FUNKTIONEN)
}
```

Beim Programmieren unterscheidet man zwischen **Hauptsketch** und **Nebensketch** und ordnet diesen unterschiedliche Karteikarten zu. Auf der Karteikarte Nebensketch programmiert man die Klasse und von der Karteikarte Hauptsketch kann man auf die Objekte einer Klasse zugreifen.

set() Beispiel: Mit a.set(0, 0) kann man den Vektor $\vec{a}$ auf null setzen.

continue Mit continue kann in einer for- oder while-Schleife bei einer bestimmten Bedingung ein Schritt im Sketch übersprungen werden.

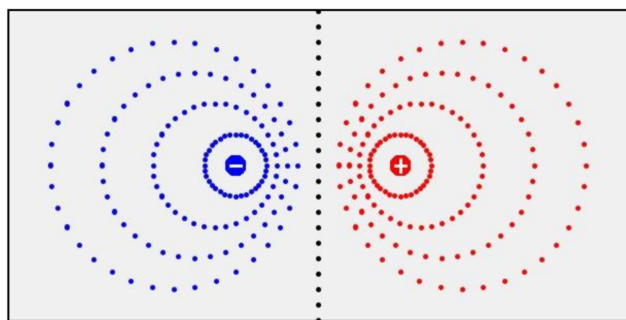
sphere() Mit sphere() kann man eine Kugel zeichnen, die jedoch stets im Koordinatenursprung sitzt.

lights() Mit lights() kann man sein Zeichnungsobjekt, z.B. eine Kugel beleuchten.

3.7 Aufgaben

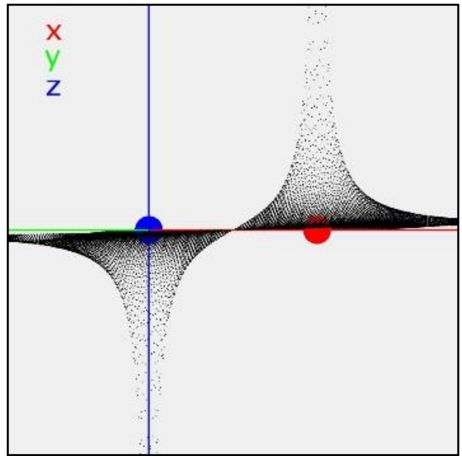
1. Stelle in Anlehnung an den Sketch *Vektorfeld_um_einen_Planeten* das Vektorfeld der elektrischen Feldstärke E um eine positive Ladung (Q = 0,6 C) grafisch dar.

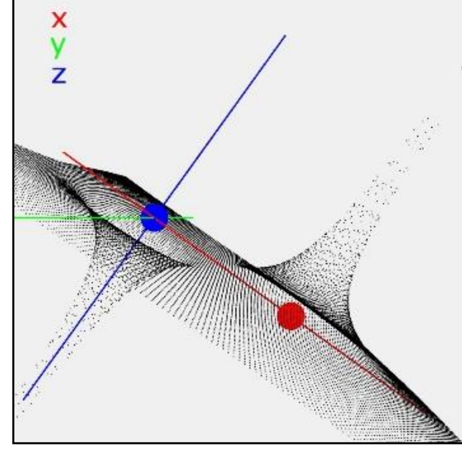
2. Entsprechend der Abbildung unten, sollen die Äquipotenziallinien als punktierte Kreise im Bereich einer negativen und einer positiven Ladung mit Processing grafisch qualitativ dargestellt werden.



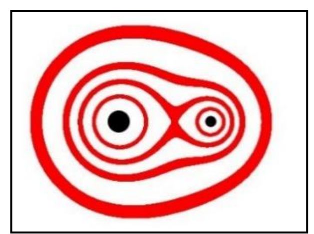
3. Erweitere den Sketch *Potenzialberg* um eine negative Ladung, die einen Potenzialtrichter erzeugt (siehe Abbildung unten). Die negative Ladung (Q2 = $-7 \cdot 10^{-7}$ C) soll im Ursprung des x-y-z-Koordinatensystems liegen. Die positive Ladung (Q1 = $7 \cdot 10^{-7}$ C) soll 300 Pixel (≙ 300 m) entfernt auf der x-Achse liegen. Beim Start des Sketches sollen Potenzialtrichter und Potenzialberg wie in der Abbildung unten links dargestellt zu sehen sein. Mithilfe der Maus soll das Potenzialgebirge um die z-Achse drehbar sein. $\varepsilon_0 = 8{,}8542 \cdot 10^{-12}$ As/Vm

   Tipp: Bezüglich der Addition der Potenziale der beiden Ladungen lies dir nochmal den Text zum Sketch *Potenzialgebirge* durch.



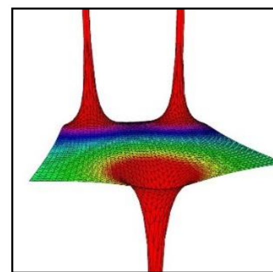


4. In einem fiktiven Szenario befinden sich zwei Massen (m_1 = 1,4·10^{12} kg, m_2 = 6,0 · 10^{11} kg) im Abstand von 125 m voneinander (1 Pixel = 1 m). In der Ebene zwischen den beiden Massen sollen die Orte gleichen Gravitationspotenzials V wie rechts gezeigt durch Äquipotenziallinien dargestellt werden. Gravitationskonstante = 6,6726·10^{-11} $m^3kg^{-1}s^{-2}$



   Anmerkung: Die Äquipotenziallinien in der Abbildung sind relativ dick und ihre Dicke nimmt nach außen hin zu. Dies rührt daher, dass Processing bei float-Zahlen nach der siebten Stelle rundet und man somit nicht einfach ein Gleichheitszeichen setzen darf. Gleich bedeutet, dass die Zahlen auch in allen Nachkommastellen übereinstimmen. Dies ist nur für wenige Orte in der Ebene zwischen den beiden Massen gegeben. Um eine sinnvolle Darstellung zu erhalten, muss man ein kleines Intervall angeben. Beispiel: V >= -1.0 - 0.05 && V <= -1.0 + 0.05.

5. Füge dem Sketch *Potenzialgebirge* einen zweiten Potenzialberg hinzu (siehe Abbildung rechts). Auch dieses Potenzialgebirge soll mit der Maus drehbar sein.



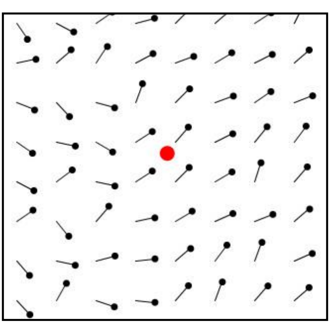
6. Spiele mit den Werten im Sketch *Gravitationsgesetz_01*.
   a) Was passiert, wenn die Masse der Sonne so vergrößert wird, dass gilt k = 230?
   b) Was passiert, wenn die Geschwindigkeit des Planeten von -1.0 auf -1.5 erhöht wird?

7. Füge im Sketch *Sonnensystem* eine zweite Sonne mit der gleichen Masse wie die erste Sonne hinzu. x = 100 und y = 500 sind die Ortskoordinaten der neuen Sonne. Der Geschwindigkeitsvektor hat die Koordinaten v_x = 0 und v_y = -1. Bleiben bei diesem Doppelsternsystem die vier Planeten im Fenster oder werden sie herauskatapultiert?

8. Der Sketch *Sonnensystem* soll wie folgt geändert werden. Vier Sterne (m_1 = m_2 = m_3 = m_4 = 2000) wechselwirken gravitativ. Sie besitzen die Ortskoordinaten: x_1 = 20, x_2 = 80, x_3 = 140, x_4 = 200 und y_1 = 50, y_2 = 150, y_3 = 250, y_4 = 350. Beim Start der Simulation besitzen alle vier Sterne den gleichen Geschwindigkeitsvektor mit den Koordinaten v_x = 10 und v_y = 0. Der Durchmesser der Sterne soll im Fenster 20 Pixel betragen. Erreichen alle vier Massen das rechte Ende des Fensters?

9. Die Abbildung rechts zeigt einen Ausschnitt aus einem Vektorfeld, in dem sich eine positive Ladung in Richtung der Vektorpfeile bewegt. Die Vektorpfeilspitzen wurden hier einfachheitshalber wieder als Punkte dargestellt (sichere Kinderpfeile ;).

   Erzeuge mittels eines zweidimensionalen Arrays in einem 800 mal 800 Pixel großen Fenster ein Vektorfeld entsprechend der Abbildung rechts. Die Abbildung rechts zeigt jedoch nur einen Ausschnitt aus dem Vektorfeld. Unterteile das Fenster in Zellen von 40 mal 40 Pixel.



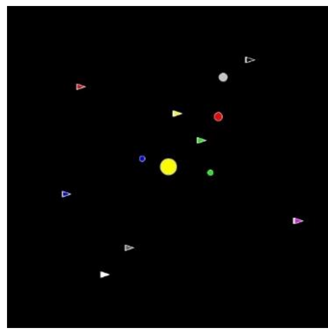
Das zweidimensionale Array *PVector[][] E* soll dann wie folgt gefüllt werden:

E[y][x] = new PVector(1 + random(-0.75, 0.75), sin(x * 0.3) + random(-0.75, 0.75)).normalize();

Informiere dich in der Processing-Referenz über die Funktion *random().* Die Funktion normalize() normalisiert den Vektor auf die Länge 1 (Einheitsvektor). Damit die Ladung sich in Richtung des Vektorpfeils in der jeweiligen Zelle bewegt, kann durch die folgende Sketchzeile eine Beziehung zwischen der Position der Ladung und dem Zelleninhalt hergestellt werden:

ladungPosition.add(E[(int)ladungPosition.y / 40][(int)ladungPosition.x / 40]);

10. Der Sketch *Sonnensystem* soll in den Sketch *Raumflotte* umbenannt werden und um eine Karteikarte für die Klasse Raumschiffe erweitert werden (siehe Abbildung unten). In diesem Nebensketch soll der Bauplan für Raumschiffe in der Form von einfachen Dreiecken konstruiert werden. Im Hauptsketch sollen anhand dieses Bauplans 8 Sternenkreuzer mit unterschiedlicher Farbe und unterschiedlicher Geschwindigkeit vom linken Bildrand aus das Sonnensystem durchfliegen, ohne dabei mit der Sonne oder einem Planeten zu kollidieren (siehe Abbildung rechts).



Raumflotte | Gravi | Raumschiffe

```
1 Gravi[] Planeten = new Gravi[5];
2 Raumschiffe[] Sternenkreuzer = new Raumschiffe[7];
```

4 Elektrik

Was erwartet uns?

Runden auf eine bestimmte Anzahl von Nachkommastellen, Unicode, Sketchoptimierung, Filme mit saveFrame() und Movie Maker erstellen, Zeichnen einer Schraubenbahn und einer Spiralbahn, keyPressed, mousePressed, arc(), Modulo-Operator %, atan(), degrees()

4.1 Spannungsteiler

In der Sekundarstufe I hat man ja schon einiges zum Thema Elektrik gelernt. So zum Beispiel das Ohm'sche Gesetz, die Gesetze der Reihen- und Parallelschaltung, …. Um diese Kenntnisse aufzufrischen, wollen wir nun in Processing einen aktiven Schaltplan für eine Reihenschaltung zeichnen (Abb. 4.1). Aktiv bedeutet, dass wir die Widerstandswerte R1 und R2 wählen sowie mit dem roten Schieberegler die Spannung U einstellen können. Die Werte für I, U1 und U2 sollen entsprechend den oben genannten Gesetzen im Fenster angezeigt werden.

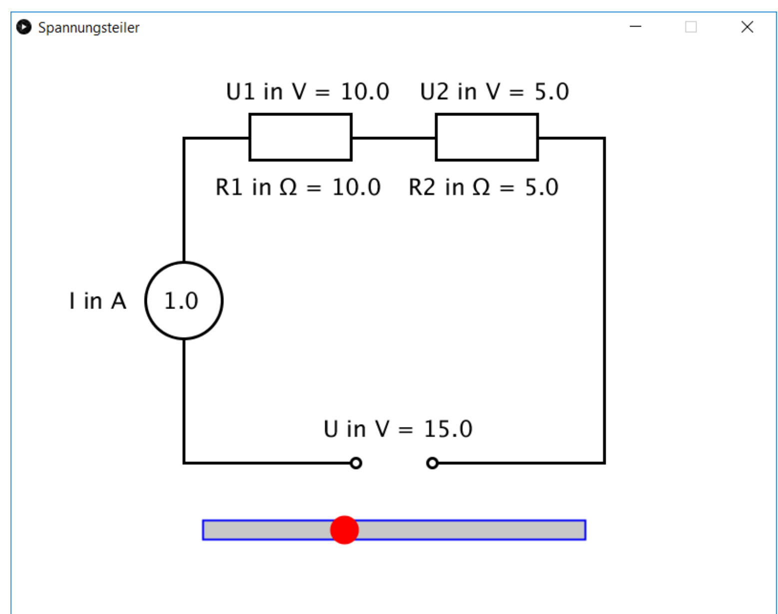


Abbildung 4.1: Simulation eines unbelasteten Spannungsteilers

Diese Programmieraufgabe ist nicht zu kompliziert. Die bekannten Funktionen beginShape(), vertex() und endShape() benutzen wir, um die Leitungen für unseren Stromkreis zu zeichnen (siehe Sketch). Mit *beginShape()* beginnen wir die Form und mit *endShape()* beenden wir sie. Die Koordinatenpunkte setzen wir mit *vertex().* Sie werden automatisch durch Linien miteinander verbunden.

Anschließend zeichnen wir die Kontakte der Spannungsquelle, die Widerstände und das Strommessgerät ein. Den Schieberegler zur Einstellung der Spannung U konstruieren wir mit der uns schon bekannten if-else-Anweisung.

Die ganze „Zeichnerei" scheint allerdings recht aufwendig zu sein. Doch hier kann man sich die Arbeit sehr erleichtern, wenn man in der Menüleiste (ganz oben in der Entwicklungsumgebung von Processing) auf *Sketch* und dann auf ***Optimieren*** klickt (siehe Abb. 4.2).

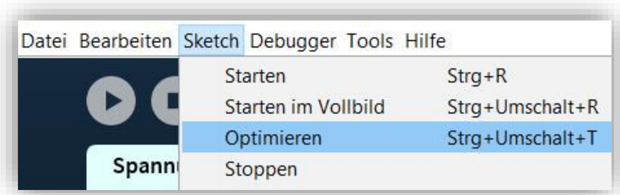


Abbildung 4.2: Ausschnitt aus der Entwicklungsumgebung (PDE) von Processing

Nun wird man aufgefordert seinen Sketch abzuspeichern. Danach sieht man in seinem Sketch unterstrichene Werte und kleine Quadrate (siehe Abb. 4.3). Alle unterstrichenen Werte kann man mit gedrückter linker Maustaste durch Schieben vergrößern oder verkleinern. Das Gute hieran ist, dass man diese Änderungen direkt in der Zeichnung im geöffneten Fenster sehen kann. So kann man in unserem Beispiel die Widerstände, das Strommessegerät und auch die Texte dahin schieben, wo man sie hinhaben möchte. Über die Quadrate kann man die Farben einstellen.

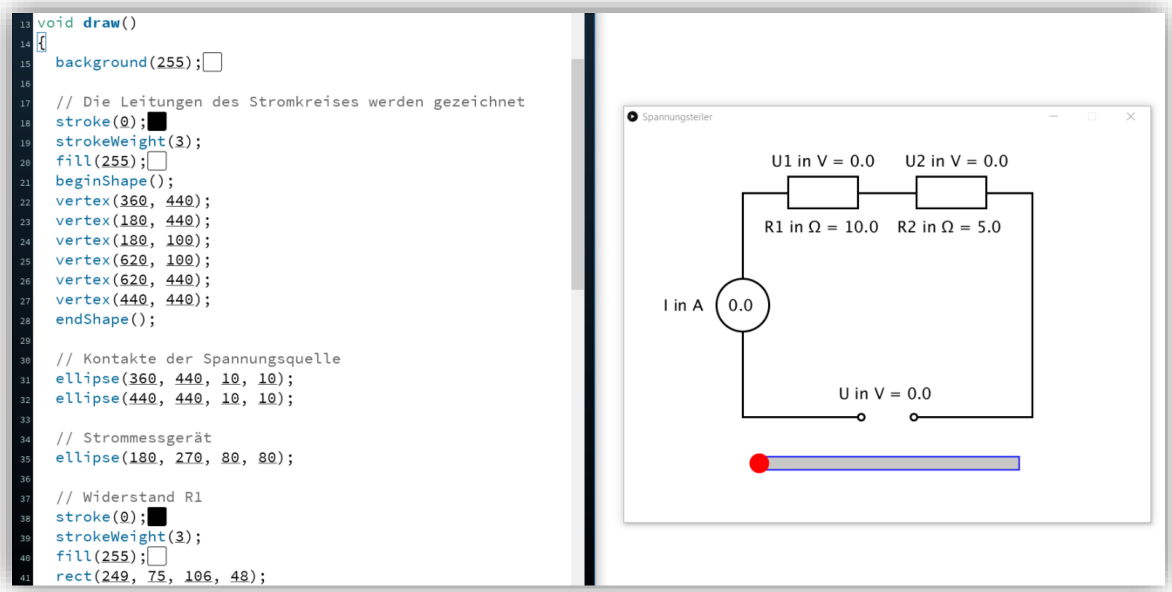


Abbildung 4.3: Optimieren der Zeichnung mithilfe von Sketch → Optimieren

In der Zeichnung sollen, wie in Abbildung 4.1 dargestellt, die einzelnen Werte für die Spannungen, den Strom und die Widerstände angegeben werden. Für die Widerstandswerte und die Spannung U ist dies unproblematisch. Sobald jedoch Größen mittels Division berechnet werden, erhalten wir in der Regel viele Stellen hinter dem Komma. In der bildlichen Darstellung macht dies keinen guten Eindruck, zumal die Gefahr besteht, dass sich die Zahlen mit ihren zahlreichen Nachkommastellen überlappen. Was tun? Nun, wir haben gelernt, dass man mit *round()* Kommazahlen auf ganze Zahlen runden kann. Doch auch diese Lösung ist hier nicht das Gelbe vom Ei. Das Strommessgerät würde dann nur ganzzahlige Amperewerte anzeigen. Eine oder zwei Stellen hinter dem Komma wären schon gut. Am Beispiel von U1 soll erklärt werden, wie es gelingt, eine Zahl mit einer Nachkommastelle anzuzeigen.

```
(float)round(10*U1)/10
```

Wir nehmen den Wert von U1 mal 10. Dadurch rutscht das Komma eine Stelle nach rechts. Dann runden wir mit *round* und erhalten so eine int-Zahl. Also einen ganzzahligen Wert, der um den

Faktor 10 zu groß ist. Deshalb teilen wir ihn wieder durch 10. Jetzt würden wir wieder einen ganzzahligen Wert erhalten, wenn nicht vor dem gesamten Ausdruck *(float)* stehen würde. Er bewirkt, dass wir ein Ergebnis mit Kommastelle erhalten. Das Ganze nochmal als Zahlenbeispiel.

$$2{,}347899023 \cdot 10 = 23{,}47899023 \rightarrow 23 \rightarrow 2{,}3$$

Hätten wir nicht durch 10, sondern durch 100 geteilt, dann hätten wir zwei Stellen hinter dem Komma erhalten.

Noch etwas Neues lernen wir in dem Textteil des Sketches Spannungsteiler. Schauen wir uns die folgende Zeile an: `text("R1 in \u03A9 = " +R1, 213, 160)`. Was bedeutet \u03A9? Wir wissen, dass der Widerstand in Ohm gemessen wird. In einem normalen Textbearbeitungsprogramm können wir hierfür recht einfach den griechischen Buchstaben Ω eingeben. In Processing ist dies nicht so einfach. Es sei denn, man kennt den Unicode für das Zeichen Ω. Unicode ist ein internationaler Standard, bei dem für alle bekannten Schriftzeichen ein digitaler Code festgelegt wird. Den Unicode für das griechische Alphabet findet man in der Tabelle nach dem Sketch *Spanungsteiler*. Damit Processing weiß, dass es sich zum Beispiel bei 03A9 um Unicode handelt und nicht um die Ziffernfolge 03A9, muss vor den Unicode \u gesetzt werden. Wenn es sinnvoll ist, werden wir in unseren zukünftigen Sketchen auch griechische Buchstaben benutzen.

Sketch 01: Spannungsteiler

```
float U; // Spannung in Volt
float U1; // Teilspannung in Volt
float U2; // Teilspannung in Volt
float R1 = 10; // Widerstand in Ohm
float R2 = 5; // Widerstand in Ohm
float I; // Strom in Ampere

void setup()
{
  size(800, 600);
}

void draw()
{
  background(255);

  // Die Leitungen des Stromkreises werden gezeichnet
  stroke(0);
  strokeWeight(3);
  fill(255);
  beginShape();
  vertex(360, 440);
  vertex(180, 440);
  vertex(180, 100);
  vertex(620, 100);
  vertex(620, 440);
  vertex(440, 440);
  endShape();

  // Kontakte der Spannungsquelle
  ellipse(360, 440, 10, 10);
  ellipse(440, 440, 10, 10);

  // Strommessgerät
  ellipse(180, 270, 80, 80);

  // Widerstand R1
```

```
  stroke(0);
  strokeWeight(3);
  fill(255);
  rect(249, 75, 106, 48);

  // Widerstand R2
  stroke(0);
  strokeWeight(3);
  fill(255);
  rect(444, 75, 106, 48);

  // Konstruktion des Schiebereglers
  stroke(0, 0, 255);
  strokeWeight(2);
  fill(200);
  rect(200, 500, 400, 20);

  if (mouseX > 200 && mouseX <= 600 && mouseY > 500 && mouseY < 520)
  {
    U = ((mouseX)-195)/10;

    stroke(255, 0, 0);
    strokeWeight(20);
    point(mouseX, 510);
  } else
  {
    U = 0;
    stroke(255, 0, 0);
    strokeWeight(20);
    point(205, 510);
  }

  // Berechnung von I und den Teilspannungen U1 und U2
  I = U/(R1+R2);
  U1 = R1*I;
  U2 = R2*I;

  // Beschriftung
  fill(0);
  textSize(24);
  text("U in V = " +U, 326, 413);
  text("U1 in V = " +(float)round(10*U1)/10, 224, 60);
  text("U2 in V = " +(float)round(10*U2)/10, 427, 60);
  text("R1 in \u03A9 = " +R1, 213, 160);
  text("R2 in \u03A9 = " +R2, 415, 160);
  text("I in A     " +(float)round(10*I)/10, 60, 279);
}
```

Unicode (hexadezimal) für griechische Buchstaben

Name	Kleinbuchstaben		Großbuchstaben	
	Buchstabe	Unicode	Buchstabe	Unicode
Alpha	α	03B1	Α	0391
Beta	β	03B2	Β	0392
Gamma	γ	03B3	Γ	0393
Delta	δ	03B4	Δ	0394

Epsilon	ε	03B5	Ε	0395
Zeta	ζ	03B6	Ζ	0396
Eta	η	03B7	Η	0397
Theta	θ	03B8	Θ	0398
Jota	ι	03B9	Ι	0399
Kappa	κ	03BA	Κ	039A
Lambda	λ	03BB	Λ	039B
My	μ	03BC	Μ	039C
Ny	ν	03BD	Ν	039D
Xi	ξ	03BE	Ξ	039E
Omikron	ο	03BF	Ο	039F
Pi	π	03BD	Π	03A0
Rho	ρ	03C1	Ρ	03A1
Sigma	σ	03C3	Σ	03A3
Tau	τ	03C4	Τ	03A4
Ypsilon	υ	03C5	Υ	03A5
Phi	φ	03C6	Φ	03A6
Chi	χ	03C7	Χ	03A7
Psi	ψ	03C8	Ψ	03A8
Omega	ω	03C9	Ω	03A9

4.2 Isotope im E- und B-Feld

Isotope im homogenen E-Feld

In der folgenden Simulation soll eine Isotopenquelle Isotope mit gleicher Anfangsgeschwindigkeit durch eine Blende in das homogene elektrische Feld eines Kondensators schicken. Die Isotope, die sich in ihrer Masse und ihrer Ladung unterscheiden können, wollen wir in dieser Simulation beobachten (siehe Abb. 4.4).

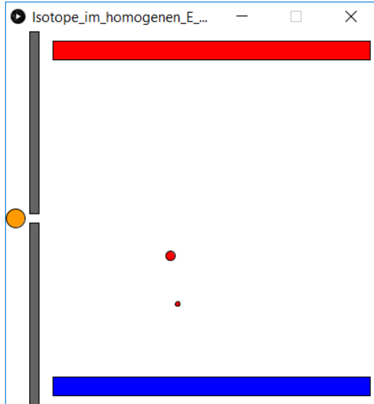


Abbildung 4.4: Zwei Isotope gleicher Ladung aber unterschiedlicher Masse im homogenen E-Feld

Das Zeichnen der Apparatur dürfte für uns kein Problem mehr sein. Auch die Programmierung der Bewegung der Isotope stellt keine besonders hohen Anforderungen, wenn man mit dem physikalischen Sachverhalt und den entsprechenden Gleichungen vertraut ist. Da alle Isotope mit konstanter Geschwindigkeit v_x durch die Blende fliegen, benötigen wir noch eine Gleichung, die die Bewegung in y-Richtung beschreibt. Diese kann man sich wie folgt herleiten.

$$y = \frac{1}{2}at^2 = \frac{1}{2}\frac{F}{m} \cdot t^2 = \frac{E \cdot Q \cdot t^2}{2 \cdot m} = \frac{U \cdot Q \cdot t^2}{2 \cdot d \cdot m}$$

Eine sinnvolle Beobachtungszeit für die Bewegung der Isotope in unserer Pixelwelt liegt bei ca. vier Sekunden. Ergeben sich für diesen Zeitraum auch sinnvolle Werte für y? Setzen wir als Beispiel in die obige Gleichung einmal konkrete Werte für das Eisenisotop ${}^{56}_{26}Fe$ ein sowie U = 10 V für die Kondensatorspannung und d = 0,05m für den Plattenabstand.

$$y = \frac{U \cdot Q \cdot t^2}{2 \cdot d \cdot m} = \frac{10\,V \cdot 9 \cdot 1{,}6 \cdot 10^{-19}C \cdot (4s)^2}{2 \cdot 0{,}05\,m \cdot 55{,}92 \cdot 1{,}66 \cdot 10^{-27}kg} = 1{,}55 \cdot 10^9\,m$$

Dieser klassisch berechnete Wert gibt nicht nur in der Pixelwelt keinen Sinn, sondern er entspricht auch nicht der physikalischen Realität. Wenn das Isotop diese Strecke in vier Sekunden zurücklegen würde, dann müsste es mit Überlichtgeschwindigkeit geflogen sein. Und dies ist nach Albert Einstein verboten. Aber auch wenn man relativistisch rechnen würde, wäre die zurückgelegte Strecke viel zu groß für eine sinnvolle Darstellung in der Pixelwelt. Besser ist es mit Werten zu rechnen, die an die Pixelwelt angepasst sind. Auch mit ihnen werden wir aussagekräftige Ergebnisse bekommen. Schauen wir uns dazu die Simulation einmal an.

Wie wir in der Simulation sehen, bewegen sich bei gleicher Ladung die Isotope mit der größeren Masse langsamer zu der anziehenden Kondensatorplatte als die Isotope mit kleinerer Masse. Demzufolge treffen bei gleichen Anfangsgeschwindigkeit v_x die Isotope mit der größeren Masse in einer größeren Entfernung von der Blende auf die Kondensatorplatte als die leichteren. Somit wäre eine Isotopentrennung möglich. Aber! In der Regel emittieren Isotopenquellen die Isotope nicht mit der gleichen Geschwindigkeit. Wie die Isotopentrennung trotzdem gelingen kann, dies erfahren wir nicht in diesem Sketch, sondern in dem danach folgenden.

Bei dem unten folgenden Sketch *Isotope_im_homogenen_E_Feld* handelt es sich um einen waagerechten Wurf im elektrischen Feld. Dieser ist eng verwandt mit dem waagerechten Wurf im Gravitationsfeld von Kapitel 2.2.2. Doch in den beiden Simulationen sehen wir einen deutlichen Unterschied. Während im Gravitationsfeld die Fallgeschwindigkeit der Körper unabhängig von ihrer Masse ist, beobachten wir, wie oben schon erwähnt, dass bei gleicher Ladung die leichten Isotope sich schneller zur negativen Platte bewegen als die schwereren. Dies ist auch physikalisch korrekt, wie die Gleichungen für beide Fälle zeigen. Beim E-Feld (Gleichung siehe oben) bleibt die Masse der Isotope im Nenner stehen, beim Gravitationsfeld (siehe folgende Gleichung) kann die sich im Gravitationsfeld bewegende Masse m herausgekürzt werden.

Gravitationsfeld: $$y = \frac{1}{2}at^2 = \frac{1}{2}\frac{F}{m} \cdot t^2 = \frac{\gamma \cdot \frac{M \cdot m}{r^2}}{2 \cdot m} t^2 = \frac{\gamma \cdot M}{2 \cdot r^2} t^2$$

Abschließend seien noch zwei Dinge angemerkt. Den Durchmesser der gezeichneten Isotope legen wir über die Massenangabe fest, damit wir die leichteren von den schwereren Isotopen unterscheiden können. Die negative Kondensatorplatte zeichnen wir blau und die positive Platte rot.

Sketch 02: Isotope_im_homogenen_E_Feld

```
float U = 10; // Kondensatorspannung
float d = 10; // Abstand der Kondensatorplatten
float Q1 = 0.1; // Ladung von Isotop 1
float Q2 = 0.1; // Ladung von Isotop 2
float m1 = 5; // Masse von Isotop 1
float m2 = 10; // Masse von Isotop 2
```

```
float t1; // Startzeit für Isotop 2
float t2; // Startzeit für Isotop 2
float t = 1; // Zeit
float vx = 1.5; // Geschwindigkeit in x-Richtung
float y1; // Bewegung von Isotop 1 in y-Richtung
float y2; // Bewegung von Isotop 2 in y-Richtung
float x1; // Bewegung von Isotop 1 in x-Richtung
float x2; // Bewegung von Isotop 1 in y-Richtung

void setup()
{
  size (400, 400);
}

void draw()
{
  background(255);

  // Isotopenquelle wird gezeichnet
  fill(255, 155, 0);
  ellipse(10, 200, 20, 20);

  // Die beiden Blenden werden gezeichnet
  fill(100);
  rect(25, 210, 10, 195);
  rect(25, 0, 10, 190);

  // Die beiden Kondensatorplatten werden gezeichnet
  fill(255, 0, 0);
  rect(50, 10, 340, 20);
  fill(0, 0, 255);
  rect(50, 370, 340, 20);

  // Die Isotope fliegen mit konstanter Geschwindigkeit in x-Richtung
  t1 = t1 + t;
  t2 = t2 + t;
  x1 = vx*t1;
  x2 = vx*t2;

  // Isotop 1 wird gezeichnet.
  y1 = 200 + U*Q1*t1*t1/(2*d*m1);
  fill(255, 0, 0);
  ellipse(x1+20, y1, m1, m1);

  // Isotop 2 wird gezeichnet.
  y2 = 200 + U*Q2*t2*t2/(2*d*m2);
  fill(255, 0, 0);
  ellipse(x2+20, y2, m2, m2);

  // Isotop 1 wird neu erzeugt, wenn das vorhergehende die untere
  // Kondensatorplatte berührt hat
  if (y1 > 370)
  {
    x1 = 0;
    y1 = 200;
    t1 = 0;
  }

  // Isotop 2 wird neu erzeugt, wenn das vorhergehende die untere
  // Kondensatorplatte berührt hat
  if (y2 > 370)
```

```
  {
    x2 = 0;
    y2 = 200;
    t2 = 0;
  }
}
```

Massenspektrometer

Abbildung 4.5: Massenspektrometer

Um Isotope unterschiedlicher Masse zu trennen, schickt man sie zuerst durch einen Geschwindigkeitsfilter und anschließend durch ein Magnetfeld. Im Geschwindigkeitsfilter kreuzt man ein homogenes Magnetfeld mit einem homogenen elektrostatischen Feld. Einen solchen Geschwindigkeitsfilter können die Isotope nur bei einer ganz bestimmten Geschwindigkeit geradlinig durchfliegen. Dann ist die auf sie wirkende Lorentzkraft entgegengesetzt gleich der elektrischen Kraft. Diese Geschwindigkeit ist unabhängig von der Masse der Isotope. Erst im zweiten Teil der Apparatur fliegen die Isotope aufgrund ihrer unterschiedlichen Masse auf unterschiedlich großen Kreisbögen.

Das Zeichnen der Apparatur dürfte uns vor keine großen Probleme stellen. Etwas mehr überlegen muss man jedoch bei der Programmierung der Isotopenbahn. Innerhalb des Geschwindigkeitsfilters setzen wir näherungsweise $F_{resultierend} = F_{elektrisch} - F_{Lorentz}$. Hierbei lassen wir unberücksichtigt, dass $F_{elektrisch}$ stets vertikal in Richtung der negativen Platte zeigt, während $F_{Lorentz}$ bei einer Bahnkrümmung radial zum Mittelpunkt der gekrümmten Bahn zeigt.

Bei der Programmierung der halbkreisförmigen Bahn der Isotope im zweiten Teil der Apparatur muss man daran denken, dass der Winkel bei Processing im Uhrzeigersinn gezählt wird (siehe Abb. 4.6).

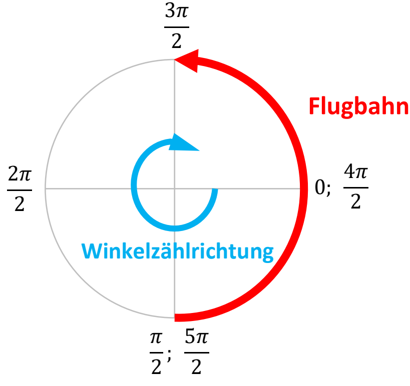


Abbildung 4.6: Flugbahn der Isotope und Winkelzählrichtung

Wenn das Isotop auf die rechte Blende oder auf eine der beiden Kondensatorplatte trifft, dann werden die Variablen wieder auf ihre Startwerte zurückgesetzt. Die Bedingungen hierfür werden im Sketch selber erklärt.

Interessant ist noch die folgende if-Anweisung am Ende des Sketches.

```
if (keyPressed)
{
saveFrame("Bilder/M01_####.tif");
}
```

Sie wird ausführlich nach dem Sketch *Massenspektrometer* beschrieben.

Sketch 03: Massenspektrometer

```
// Massenspektrometer

float Q = 8; // Q beeinflusst v nicht
float B1 = 2.0; // magn. Flussdichte in Teil 1
float B2 = 1.0; // magn. Flussdichte in Teil 2
float E = 4.0; // elektr. Feldstärke
float FE; // elektr. Kraft
float FL1; // Lorentzkraft in Teil 1
float FL2; // Lorentzkraft in Teil 2
float v = 2.0; // Geschwindigkeit der Isotope
float t1; // Zeit für die Bewegung in x-Richtung
float t2; // Zeit für die Strecke s
float t3 = 1; // Zeit für die Bewegung auf dem Kreisbogen
float dt = 0.5; // Zeitzuwachs
float m = 250; // m beeinflusst v nicht
float x;
float s; // Rechtswert für die Bewegung auf dem Kreisbogen
float y;
float r; // Radius des Kreisbogens
float w = PI/2; // Winkel im Bogenmaß
float dw; // Winkelzuwachs

void setup() {
  size (600, 400);
}

void draw() {
  background(255);

  // Die beiden Blenden werden gezeichnet
  fill(100);
  rect(25, 205, 10, 195);
  rect(25, 0, 10, 195);
  rect(320, 205, 10, 195);
  rect(320, 0, 10, 195);

  // Die beiden Kondensatorplatten werden gezeichnet
  fill(255, 0, 0); // Die positive Platte ist rot
  rect(50, 10, 255, 20);
  fill(0, 0, 255); // Die negative Platte ist blau
  rect(50, 370, 255, 20);

  // Zeichen für die beiden B-Felder werden gezeichnet
  noFill();
```

```
stroke(0, 255, 0);
strokeWeight(3);
ellipse(70, 70, 30, 30);
line(60, 80, 80, 60);
line(60, 60, 80, 80);
ellipse(470, 70, 30, 30);
line(460, 80, 480, 60);
line(460, 60, 480, 80);
fill(0, 255, 0);
textSize(32);
text("B1", 100, 80);
text("B2", 500, 80);

// FE und FL1 werden berechnet.
if (x <= 310)
{
  FE = Q*E;
  FL1 = Q*v*B1;
}

// Die Isotope werden gezeichnet.
noStroke();
fill(255, 0, 0);
ellipse(x+10, y, 7, 7);

x = v*t1; // Bewegung in x-Richtung
t1 = t1 + dt;
s = v*t2; // Bewegung auf der Kreisbahn
t2 = t2 + dt;

// Die Ablenkung in y-Richtung wird berechnet
if (x > 25 && x < 310)
{
  y = 200 + ((FE-FL1)*t1*t1)/(2*m); // Dies ist nicht ganz korrekt,
                   // da FL die Teilchen auf eine Kreisbahn zwingt
}

// Isotopenquelle wird gezeichnet
fill(255, 155, 0);
ellipse(10, 200, 20, 20);

/* Wenn das Isotop mit einer Kondensatorplatten kollidiert,
 dann werden alle Variablen auf ihre Ausgangswerte zurückgesetzt */
if (y < 30 || y > 370)
{
  x = 0;
  s = 0;
  y = 200;
  t1 = 0;
  t2 = 0;
}

// Es wird überprüft, ob das Isotop mit der rechten Blende kollidiert
if (x > 310)
{
  // Kollision mit der rechten Blende
  if (y < 195 || y > 205)
  {
    // Die Variablen werden auf ihre Ausgangswerte zurückgesetzt
    x = 0;
    y = 200;
```

```
      t1 = 0;
      s = 0;
      t2 = 0;
    }
  }

  /* Wenn das Isotop die Blendenöffnung passiert, dann werden die
     Variablen x, y und t1 auf ihre Ausgangswerte zurückgesetzt */
  if (x > 320)
  {
    x = 0;
    y = 200;
    t1 = 0;
  }

  // Das Isotop auf der Kreisbahn wird gezeichnet
  if (s >= 320)
  {
    r = (m*v)/(Q*B2);

    fill(255, 0, 0);
    ellipse((330 + r*cos(w)), ((200-r) + r*sin(w)), 7, 7);

    // Ist diese if-Bedingung erfüllt, wird der Winkel vergrößert
    if (w >= 3*PI/2 && w <= 5*PI/2)
    {
      dw = v*t3/r; // Winkelzuwachs
      w = w - dw;
    }

    // Ist diese if-Bedingung erfüllt, wird der Winkel auf seinen
    // Startwert zurückgesetzt
    if (w <= 3*PI/2 && x >= 320)
    {
      w = 5*PI/2;
    }
  }

  /* Solange eine beliebige Taste auf der Tastatur gedrückt gehalten
     wird, speichert Processing den Fensterinhalt pro Schleifendurchlauf
     als Bild im Ordner Bilder ab */
  if (keyPressed)
  {
    saveFrame("Bilder/M01_####.tif");
  }

  println("FE = ", FE, "FL1 = ", FL1, "E/B1 = ", E/B1, "v = ", v,
          "2r = ", 2*r);
}
```

Kommen wir nun wie versprochen zur Beschreibung der if-Anweisung am Ende des Sketches.

```
if (keyPressed)
{
saveFrame("Bilder/M01_####.tif");
}
```

Wenn man eine schöne Simulation erstellt hat, dann möchte man sie auch seinen Mitschülern oder seinem Lehrer zeigen. Processing bietet die Möglichkeit an, einen Film von einer Simulation zu erstellen, den man dann auf sein Smartphone übertragen kann. Wie man mit Processing einen Film erstellen kann, soll nun erklärt werden.

Mit der Anweisung **saveFrame()** können wir eine durchnummerierte Bildsequenz unserer Animation speichern. Die Aufzeichnung beginnt, wenn die Animation gestartet wird und endet, wenn die Animation beendet wird. Dies kann mit dem Start- und Stoppbutton der Processing Entwicklungsumgebung (Processing Development Environment = PDE) geschehen. Deshalb sollte die Anweisung *saveFrame()* am Ende des Sketches stehen. Will man den Aufzeichnungsprozess mit ***mousePressed*** oder ***keyPressed*** steuern, dann muss man dies, wie in unserem Sketch *Massenspektrometer* geschehen, mit einer if-Anweisung veranlassen. Dies hat den Vorteil, dass nicht bei jedem Programmstart eine Unmenge von Bildern abgespeichert wird.

Bei *saveFrame()* kann man zwischen den runden Klammern einige Eigenschaften für die Abspeicherung der Bilder festlegen. Hier ein Beispiel.

saveFrame(„Bilder/M01_####.tif“)

Das Wort *Bilder* vor den Schrägstrich gibt den Namen des Ordners an, den Processing erstellt, um die einzelnen Bilder hierin abzuspeichern. Der Ordner muss natürlich nicht Bilder heißen, sondern sein Name kann frei gewählt werden. Auch kann der Ort des Ordners frei gewählt werden. Sinnvoll ist es meines Erachtens, wenn man ihn im zugehörigen Sketchordner integriert. Als Namen für die einzelnen Bilder haben wir *M01* gewählt. *M01* steht hier für Massenspektrometer 01. Die folgenden vier Hashtags sind Platzhalter für die Durchnummerierung der Bilder. Mit vier Hashtags können die Bilder von 1 bis 9999 durchnummeriert werden. Mit fünf Hashtags von 1 bis 99999, usw. Durch einen Punkt von den Hashtags getrennt ist das Dateiformat. Folgende Formate sind möglich. JPEG (.jpg), PNG (.png), TIFF (.tif) und TARGA (.tga).

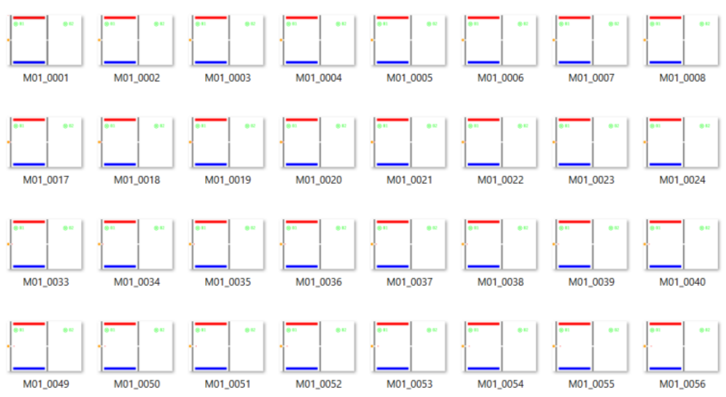


Abbildung 4.7: Bei jedem Durchlauf von void draw() wird ein Bild abgespeichert

Nachdem die Animation im Sketch abgespielt wurde und die Bilder abgespeichert sind (siehe Abb. 4.7), kann man aus diesen Einzelbildern den Film erstellen. Dazu klickt man in der PDE auf Tools und dann auf **Movie Maker**. Es öffnet sich nun das folgende Fenster (Abb. 4.8).

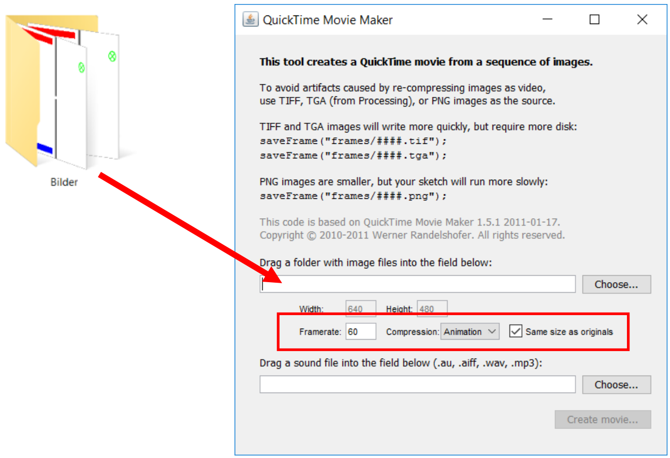


Abbildung 4.8: Einfügen der abgespeicherten Bilder in Movie Maker

Wie in Abbildung 4.8 dargestellt, ziehen wir nun den Ordner *Bilder* in das freie Feld von Movie Maker. Anschließend passen wir die Framerate des Films der Sketch-Framerate an und setzen bei *Same size as originals* ein Häkchen, um die Bildgröße des Films an die Bildgröße des Sketchfensters anzupassen. Andere Bildgrößen sind natürlich auch einstellbar. Bei *Compression* haben wir die Wahl zwischen Animation, JPEG und PNG. Anschließend klicken wir auf *Create movie* und Processing erstellt den Film im Dateiformat .mov (siehe Abb. 4.9).

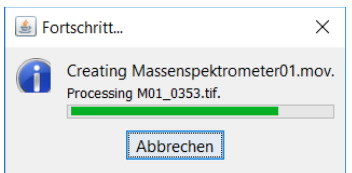


Abbildung 4.9: Movie Maker erstellt den Film

Danach kann man sich z.B. mit dem VLC mediaplayer das Video anschauen oder auf sein Smartphone überspielen. Eventuell muss man es mit einem Videobearbeitungsprogramm vorher noch in ein anderes Dateiformat umwandeln.

4.3 Schraubenbahn und Spiralbahn

Schraubenbahn

Schießt man Elektronen schräg in ein homogenes Magnetfeld ein, so bewegen sie sich auf einer Schraubenbahn. Dies ist eine dreidimensionale Bewegung, die so manchen Schüler an die Grenzen seines räumlichen Vorstellungsvermögens bringt. Helfen wir diesen Schülern, in dem wir einen Sketch schreiben, in dem man eine Schraubenbahn von allen Seiten betrachten kann. Um optische Irritationen zu vermeiden, versehen wir die Schraubenbahn noch mit einer grünen Mittellinie.

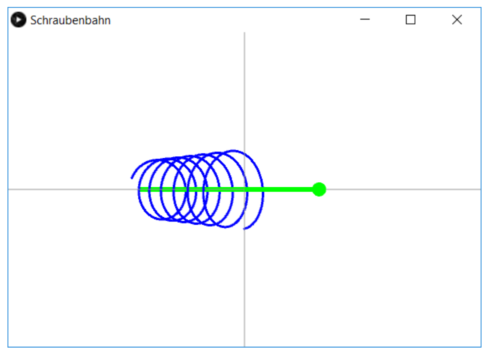


Abbildung 4.10: Ausschnitt aus dem Film "Schraubenbahn"

Das Programmieren des Sketches stellt keine große Herausforderung dar, wenn man verstanden hat, wie eine Schraubenbahn zustande kommt. Die Lorentzkraft zwingt die Elektronen auf eine Kreisbahn, die sich aber nicht schließen kann, weil die Elektronen in unserem Sketch noch eine zusätzliche Geschwindigkeitskomponente in z-Richtung besitzen. In der *for-Schleife* im Sketch sorgen die beiden Terme *(r*cos(radians(w2)))* und *(r*sin(radians(w2)))* für die Kreisbewegung und der Term *z = z + 0.08* für die Vorwärtsbewegung, die ein Schließen des Kreises verhindert. w2 steht für den Winkel in Grad, der von 90° auf 2500° anwächst, sodass die Elektronen fast sieben Umläufe machen. Mit der Funktion *radians()* wird der Winkel ins Bogenmaß umgewandelt. Hier ist der entsprechende Sketchausschnitt.

```
for (w2 = 90, z = 0; w2 < 2500 && z < 300; w2++, z = z + 0.08)
{
  stroke(0, 0, 255);
  strokeWeight(3);
  point((r*cos(radians(w2))), (r*sin(radians(w2))), z);
}
```

Die so entstehende Schraubenbahn lassen wir um die y-Achse rotieren, damit man sie von allen Seiten betrachten kann.

Da wir seit dem letzten Sketch wissen, wie man eine Simulation filmt, können wir nun auch die Rotation der Schraubenbahn um die y-Achse filmen. Wenn wir aber nicht wollen, dass während der ganzen Laufzeit des Sketches Bilder aufgezeichnet werden, sondern nur dann, wenn wir es möchten, dann müssen wir in unseren Sketch die folgende if-Anweisung einfügen.

```
if (keyPressed)
{
saveFrame("Bilder/SB_####.jpg");
stroke(255, 0, 0);
strokeWeight(50);
point(0, 0, 0);
}
```

Mit dieser if-Anweisung keyPressed werden, wie schon im Sketch *Massenspektrometer* erwähnt, nur dann Bilder aufgezeichnet, wenn eine beliebige Taste der Computertastatur gedrückt gehalten wird. Zusätzlich erscheint während dieser Zeit ein roter Punkt im Sketchfenster. Dieser Punkt wird aber nicht mit aufgezeichnet, da die entsprechenden Programmzeilen erst nach *saveFrame()* im Programmablauf abgearbeitet werden. Hätten wir die Programmzeilen vor *saveFrame()* geschrieben, wäre der Punkt mit aufgezeichnet worden.

Sketch 04: Schraubenbahn

```
float z;
float r = 50; // Radius der Schraubenbahn
float w1; // Winkel in Grad für die Rotation um die y-Achse
float w2; // Winkel in Grad für Kreisbewegung der Elektronen

void setup()
{
  size(600, 400, P3D); // 3D-Raum
}

void draw()
{
  background(255);

  // Koordinatenkreuz zeichnen
  stroke(150);
  strokeWeight(1);
  line(0, 200, 600, 200);
  line(300, 0, 300, 400);

  // Verschiebung des Koordinatenursprungs in die Fenstermitte
  translate(300, 200, 0);

  // Drehbewegung der Schraubenbahn um die y-Achse
  rotateY(radians(w1));
  w1 = w1 + 0.3;

  /* Zur Vermeidung von optischen Irritationen wird die
   Schraubenbahn mit einer Mittellinie versehen */
  stroke(0, 255, 0);
  strokeWeight(6);
  line(0, 0, -80, 0, 0, 200);
  strokeWeight(18);
  point(0, 0, -80);

  // Schraubenbahn zeichnen mit 7 Umdrehungen minus 20°
  //(7 * 360° - 20° = 2500°)
  for (w2 = 90, z = 0; w2 < 2500 && z < 300; w2++, z = z + 0.08)
  {
    stroke(0, 0, 255);
    strokeWeight(3);
    point((r*cos(radians(w2))), (r*sin(radians(w2))), z);
  }


  /* Die Bilder für den Film werden solange aufgezeichnet, wie eine
     Taste auf der Tastatur gedrückt wird.
     Während der Aufzeichnung erscheint ein roter Punkt im Fenster, der
     jedoch nicht mit aufgezeichnet wird. */

  if (keyPressed) // Beliebige Taste auf der Computertastatur
  {
    saveFrame("Bilder/SB_####.jpg");
    stroke(255, 0, 0);
    strokeWeight(50);
    point(0, 0, 0);
  }
}
```

Spiralbahn (Zyklotron)

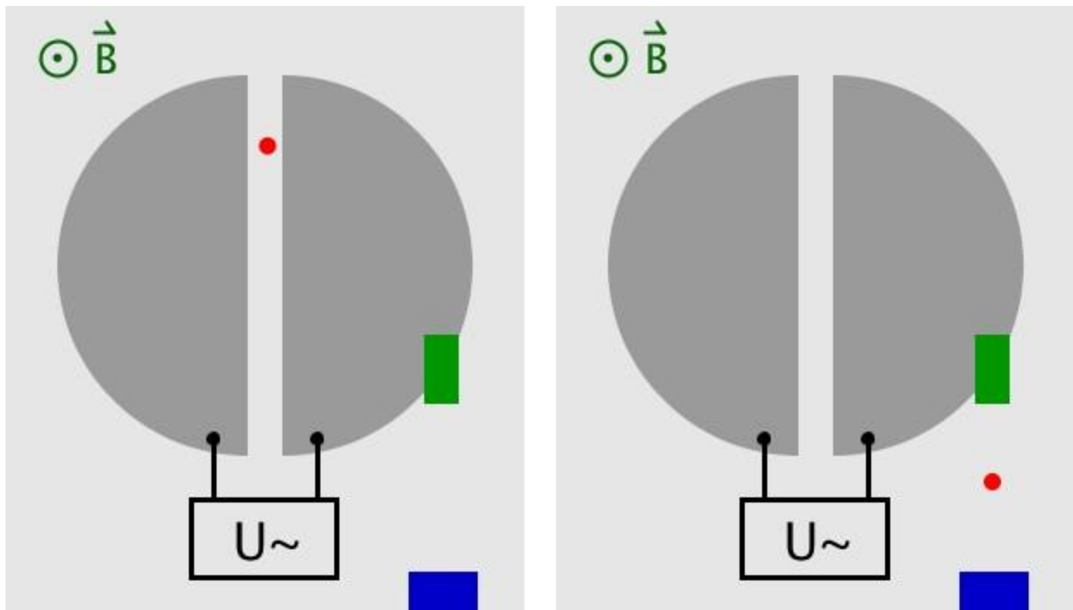


Abbildung 4.11: Simulation eines Zyklotrons

Wie man eine Schraubenbahn programmiert, dies haben wir im vorhergehenden Sketch gelernt. In einem Zyklotron bewegen sich die Ladungen (z.B. Protonen, Heliumkerne, …) jedoch nicht auf einer Schraubenbahn, sondern auf einer Spiralbahn. Wie man diese programmiert, lernen wir jetzt. Doch zuvor noch einige Worte zur Funktionsweise eines Zyklotron. Wenn man eine flache Metalldose in der Mitte durchschneidet und etwas auseinanderzieht, dann hat man zwei halbkreisförmige Elektroden, die man Duanten nennt. Sie sind in der Abbildung 4.11 grau dargestellt. An diese Duanten wird eine hochfrequente Wechselspannungsquelle mit konstanter Frequenz angeschlossen, sodass z.B. Protonen im Spalt zwischen den beiden Duanten beschleunigt werden. Im Innern der Duanten herrscht kein elektrisches Feld. Da sich die beiden Duanten in einem homogenen Magnetfeld befinden, werden die Protonen durch die Lorentzkraft abgelenkt. In der obigen Abbildung 4.11 zeigt das Magnetfeld auf den Betrachter zu. Das grüne Rechteck in der Abbildung stellt die Ablenkvorrichtung dar, die dafür sorgt, dass die Protonen aus dem Zyklotron ausgekoppelt werden können, um sie auf ein Target (Ziel) zu schießen. In der Abbildung 4.11 ist das Target blau dargestellt.

Nun kommen wir wieder zu unserer Anfangsfrage: Wie programmiert man eine Spiralbahn? Schauen wir uns die folgenden Sketchzeilen an.

```
if (w <= 8*PI)
{
  x1 = r * cos(w);
  y1 = r * sin(w);

  w = w + dw;
  r = r + dr;

  noStroke();
  fill(255, 0, 0);
  ellipse(x1, y1, 10, 10);
 }
```

Die Bewegung des Protons auf einer Kreisbahn mit dem Radius r gelingt mit x1 = r * cos(w) und y1 = r * sin(w), wenn der Winkel w beim Durchlaufen von void draw() um dw zunimmt. Entsprechend ändern sich die Werte von x1 und y1 (siehe Abbildung 4.12).

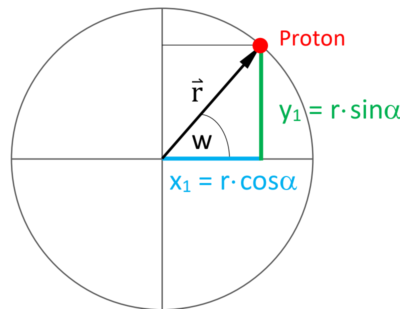


Abbildung 4.12: Bewegung eines Protons auf einer Kreisbahn

Der Winkel w nimmt also bei jedem Durchlauf von *void draw()* um dw im Bogenmaß zu. Da aber auch gleichzeitig der Radius r um dr zunimmt, kann sich die Kreisbahn nicht schließen. Somit wandert der Mittelpunkt des Protons auf einer Spiralbahn nach außen.

Das Zeichnen der beiden halbkreisförmigen Duanten gelingt nicht mit *ellipse()* sondern mit **arc()**. Mit arc() kann man einen Kreisbogen zeichnen. Ein Kreisbogen mit einem Winkel π ergibt einen Halbkreis. Schauen wir uns hierzu die folgende Sketchzeile an: arc(10, 0, 220, 220, 3*PI/2, 5*PI/2). Die ersten beiden Zahlen 10 und 0 geben den Mittelpunkt des Kreises an. Die Zahlen 220 und 220 geben den Durchmesser des Kreises in x- und y-Richtung an. Die Zahlen 3*PI/2 und 5*PI/2 geben den Startwinkel und den Endwinkel des Bogens an (5*PI/2 - 3*PI/2 = PI). Mit anderen Zahlenwerten kann man beliebige Kreis- und auch Ellipsenbögen zeichnen. Man muss jedoch beachten, wie in Processing die Winkelangabe erfolgt. Siehe hierzu Abbildung 4.13.

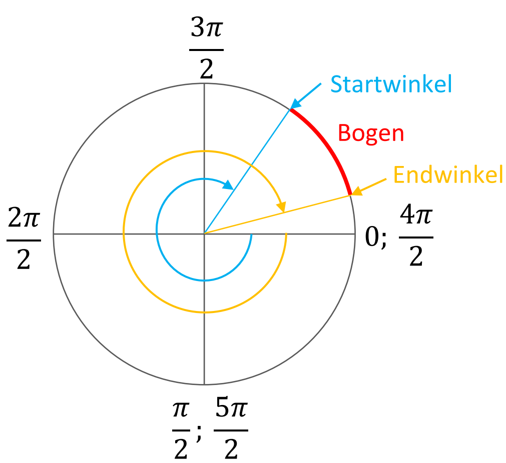


Abbildung 4.13: Zeichnen eines Kreisbogens in Processing

Beim Zeichnen der gesamten Zyklotron-Apparatur greifen wir wieder auf das Processing-Werkzeug *Optimieren* (Tools → Optimieren) zurück. Es erleichtert die Positionierung der einzelnen Zeichnungsobjekte sehr.

Das Einzige, was jetzt noch zu erwähnen ist, ist die Erstellung des Wechselspannungszeichens ~ hinter dem Buchstaben U. Es in Processing zu zeichnen ist etwas mühsam. Einfacher geht es, wenn wir es mittels Unicode als Sonderzeichen einfügen. Im Kapitel Spannungsteiler wurde erklärt, wie man griechische Buchstaben einfügt. Der entsprechende Unicode wurde hier mittels einer Tabelle

angegeben. Doch diese Tabelle enthält kein Wechselspannungszeichen. Wie kommt man an den Unicode für das Wechselspannungszeichen? Hier helfen Textverarbeitungsprogramme wie Microsoft Word oder LibreOffice Writer.

Bei *Microsoft Word* geht man wie folgt vor (siehe hierzu Abb. 4.14).

Einfügen → Symbol → Schriftart und Subset (Unterbereich) wählen, welche das gewünschte Sonderzeichen enthält (siehe Abbildung 4.14) → Sonderzeichen anklicken → unten rechts *Unicode (hex)* wählen → Zeichencode (in Beispiel 007E) übernehmen und bei Processing wie folgt einfügen: text("U\u007E", -20, 170);

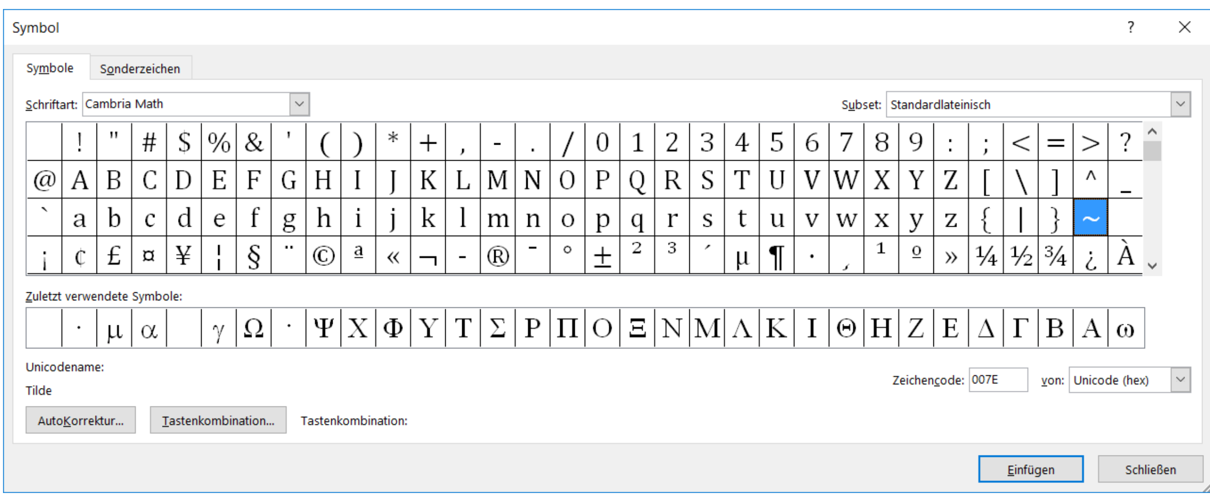


Abbildung 4.14: Unicodezeichen finden mit Microsoft Word

In *LibreOffice Writer* funktioniert es ähnlich (siehe hierzu Abb. 4.15).

Einfügen → Sonderzeichen → Schriftart und Bereich wählen, die das gewünschte Sonderzeichen enthält (siehe Abbildung 4.15) → Sonderzeichen anklicken → Zeichencode unten rechts (in Beispiel U+2248) übernehmen und bei Processing wie folgt einfügen: text("U\u2248", -20, 170);

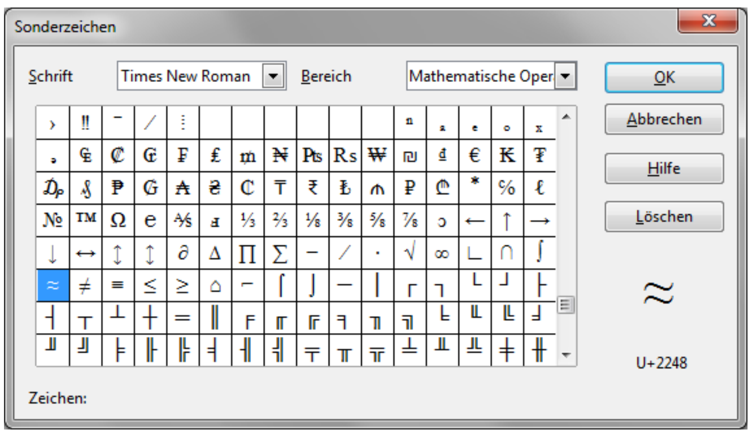


Abbildung 4.15: Unicodezeichen finden mit LibreOffice Writer

Ich hoffe, dass es nun keine Probleme mehr gibt, den folgenden Sketch zu verstehen. Vielleicht bekommt man sogar Lust die Bewegung des Protons im Zyklotron zu filmen.

Sketch 05: Zyklotron

```
// Zyklotron

float w = 0.0; // Winkel im Bogenmaß
float dw = 0.1; // Winkelzunahme im Bogenmaß
float r = 0.0; // Radius
float dr = 0.4; // Zunahme von r
float x1 = 0; // x-Wert für das Proton auf der Spiralbahn
float y1 = 0; // y-Wert für das Proton auf der Spiralbahn
float y2 = 0; // y-Wert das Proton auf der Geraden

void setup()
{
  size(300, 350);
}

void draw()
{
  background(230);
  translate (150, 150);

  // Die Spiralbahn wird für 4 Umläufe des Protons gezeichnet
  if (w <= 8*PI)
  {
    x1 = r * cos(w);
    y1 = r * sin(w);

    w = w + dw;
    r = r + dr;

    noStroke();
    fill(255, 0, 0);
    ellipse(x1, y1, 10, 10);

    /* Der Winkel w nimmt bei jedem Durchlauf von void draw um
     dw im Bogenmaß zu. Da gleichzeitig der Radius r um dr zunimmt,
     kann sich die Kreisbahn nicht schließen. Somit wandert der
     Mittelpunkt des Protons auf einer Spiralbahn nach außen */
  }

  // Nach 8 Umläufen soll sich das Proton auf einer Geraden
  //bewegen
  if (w >= 8.0*PI)
  {
    w = w + dw;
    y2 = y2 + 20;

    noStroke();
    fill(255, 0, 0);
    ellipse(102, -15+y2, 10, 10);
  }

  // Die beiden Duanten mit Ablenkvorrichtung und
  // Target werden gezeichnet.
  noStroke();
  fill(100, 150);
  arc(-10, 0, 220, 220, PI/2, 3*PI/2); // linker Duant
  arc(10, 0, 220, 220, 3*PI/2, 5*PI/2); // rechter Duant
  fill(0, 150, 0);
  rect(92, 40, 20, 40); // Ablenkvorrichtung
```

```
  fill(0, 0, 200);
  rect(83, 177, 40, 25); // Target

  // Zeichnen der Angaben zum B-Feld
  fill(0, 100, 0);
  textSize(24);
  text("B", -100, -112);
  ellipse(-120, -120, 5, 5);
  stroke(0, 100, 0);
  strokeWeight(2);
  noFill();
  ellipse(-120, -120, 20, 20);
  line(-99, -135, -88, -135);
  line(-92, -141, -88, -135);

  // Spannungsquelle wird gezeichnet und beschriftet
  noFill();
  stroke(0);
  strokeWeight(3);
  rect(-43, 135, 84, 45);
  line(-30, 135, -30, 100);
  line(30, 135, 30, 100);
  ellipse(-30, 100, 5, 5);
  ellipse(30, 100, 5, 5);
  fill(0);
  textSize(30);
  text("U\u007E", -20, 170);
}
```

4.4 Bewegte Leiterschleifen im homogenen Magnetfeld

Induktion

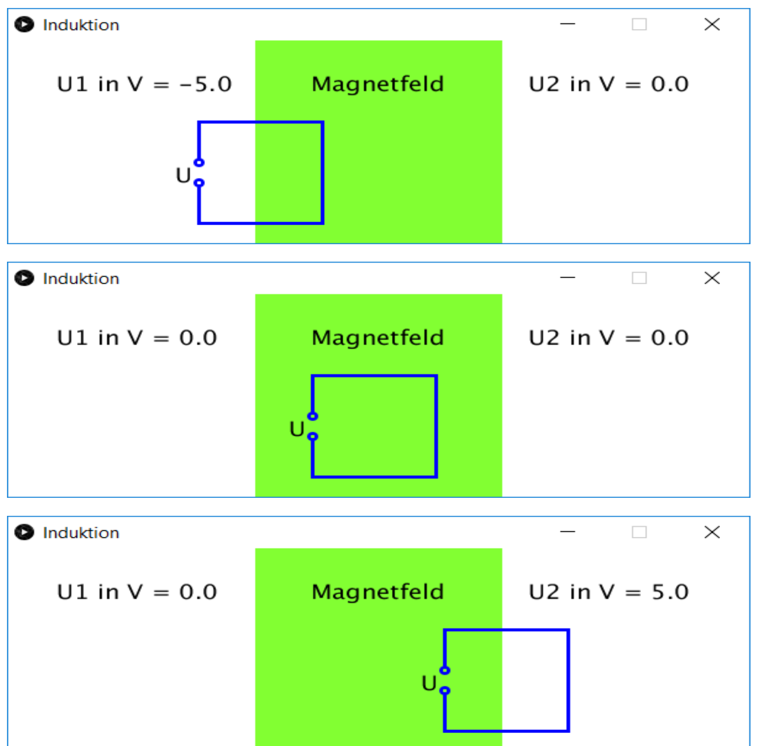


Abbildung 4.16: Eine Leiterschleife bewegt sich mit konstanter Geschwindigkeit durch ein homogenes Magnetfeld

Wenn sich in einer Spule der magnetische Fluss Φ ändert, dann entsteht an den Enden der Spule eine Spannung U_i. Für eine Spule mit n Windungen gilt:

$$U_i = -n\frac{d\Phi}{dt} = -n\ \frac{d(\vec{B}\cdot\vec{A})}{dt}$$

Wenn es sich wie in der obigen Abbildung um eine Leiterschleife handelt, also um eine Spule mit nur einer Windung und der Vektor der Flussdichte $\vec{B}$ parallel zum Flächenvektor $\vec{A}$ steht, dann vereinfacht sich die Gleichung wie folgt:

$$U_i = -\ \frac{d(B\cdot A)}{dt}$$

In unserem Sketch, der die obige Abbildung erstellt hat, wird eine Leiterschleife mit konstanter Geschwindigkeit v in x-Richtung durch ein homogenes, sich nicht änderndes Magnetfeld bewegt. Dies bedeutet, dass die vom Magnetfeld durchflossene Fläche A sich beim Eindringen in das Magnetfeld nach der Gleichung $A = h\cdot\ b = h\cdot v\cdot t$ und beim Austreten aus dem Magnetfeld nach der Gleichung $A = h\cdot\ b - h\cdot v\cdot t$ zeitlich ändert. Somit ergibt sich für die Spannung U_1 beim Eindringen der Leiterschleife in das Magnetfeld:

$$U_i = U_1 = -B\ \frac{dA}{dt} = -B\frac{d(h\cdot v\cdot t)}{dt} = -B\cdot h\cdot v$$

Für das Heraustreten erhält man:

$$U_i = U_2 = -B\ \frac{dA}{dt} = -B\frac{d(h\cdot b - h\cdot v\cdot t)}{dt} = B\cdot h\cdot v$$

Programmtechnisch ist der folgende Sketch recht einfach. Hinweisen möchte ich nur noch auf den folgenden Sketchausschnitt.

```
if (s>=200-b && s <=200)
{
U1 = -B*h*v;
} else U1 = 0.0;
```

Wenn man *if* mit *wenn* und *else* mit *sonst* übersetzt, dann ist der obige Sketchausschnitt leicht zu verstehen. In der folgenden Abbildung 4.17 sehen wir das entsprechende Flussdiagramm.

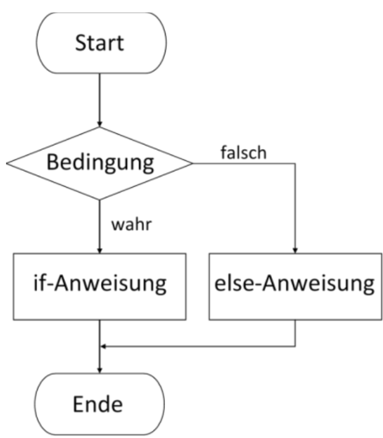


Abbildung 4.17: Flussdiagramm

Sketch 06: Induktion

```
// Induktion

float B = 0.05; // magn. Flussdichte
float t = 1; // Zeitkonstante
float U1; // Spannung beim Eintritt in das Magnetfeld
float U2; // Spannung beim Austritt aus dem Magnetfeld
float v = 1; // Geschwindigkeit der Leiterschleife
float s; // Ortskoordinate der Leiterschleife
float b = 100; // Breite der Leiterschleife
float h = 100; // Höhe der Leiterschleife

void setup()
{
  size(600, 200);
}

void draw()
{
  background(255);

  // Magnetfeld
  noStroke();
  fill(130, 256, 50);
  rect(200, 0, 200, 200);

  // Leiterschleife
  stroke(0, 0, 255);
  strokeWeight(3);
  noFill();
  beginShape();
  vertex(s, 115);
  vertex(s, 80);
  vertex(s+100, 80);
  vertex(s+100, 180);
  vertex(s, 180);
  vertex(s, 145);
  endShape();
  // Kontakte der Leiterschleife
  ellipse(s, 120, 6, 6);
  ellipse(s, 140, 6, 6);

  // Bewegung der Leiterschleife
  s = s + v * t;

  // Berechnung der Induktionsspannungen
  if (s>=200-b && s <=200)
  {
    U1 = -B*h*v;
  } else U1 = 0.0;

  if (s+b>=400 && s <=400)
  {
    U2 = B*h*v;
  } else U2 = 0.0;

  // Beschriften und runden auf zwei Stellen hinter dem Komma
  fill(0);
  textSize(20);
```

```
  text("U1 in V = " +(float)round(100*U1)/100, 39, 50);
  text("U2 in V = " +(float)round(100*U2)/100, 421, 50);
  text("Magnetfeld", 245, 50);
  text("U", s-20, 140);
}
```

Rotierende Leiterschleife

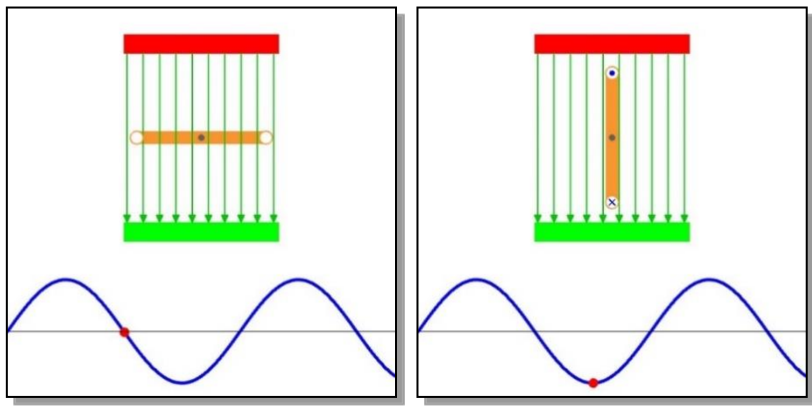


Abbildung 4.18: Rotierende Leiterschleife in einem homogenen Magnetfeld

Im dem vorhergehenden Sketch *Induktion* bewegte sich eine Leiterschleife mit konstanter Geschwindigkeit durch ein homogenes Magnetfeld. Im nun folgenden Sketch soll sich die Leiterschleife mit konstanter Winkelgeschwindigkeit ω in einem homogenen Magnetfeld drehen. Dadurch verändert sich zwar nicht die Größe der Fläche A der Leiterschleife, aber es ändert sich die Größe der vom Fluss durchsetzten Fläche A_F. Für A_F gilt $A_F = A \cdot \cos\varphi = A \cdot \cos(\omega \cdot t)$. Da die Flussdichte B konstant ist, lässt sich die entstehende Induktionsspannung wie folgt berechnen.

$$U_i = -B\,\frac{dA_F}{dt} = -B \cdot A\,\frac{d(cos \cdot (\omega \cdot t))}{dt} = -B \cdot A \cdot \omega \cdot \sin(\omega \cdot t)$$

Wie die Gleichung zeigt, erhalten wir eine sinusförmige Wechselspannung.

In unserem neuen Sketch soll eine rotierende Leiterschleife erzeugt werden, bei der auch, wie in der Abbildung 4.18 zu sehen, die Elektronenstromrichtung angezeigt wird. Gleichzeitig soll mittels eines roten Punktes qualitativ die Höhe der Induktionsspannung auf einer Siunusfunktion angezeigt werden.

Das Zeichnen der Apparatur und das Zeichnen der blauen Sinusfunktion mit dem roten Punkt dürften keine Probleme bereiten. Auch *pushMatrix()* und *popMatrix()* sind uns aus Kapitel 2.4.2 bekannt. Um jedoch die Elektronenstromrichtung auf den Schnittflächen der Leiterschleife richtig darzustellen, müssen wir etwas Neues lernen. Nach jeder 180°-Drehung nimmt der Elektronenstrom kurzzeitig den Wert null an und wechselt dann seine Richtung. D.h., der Punkt (Elektronen fließen auf den Betrachter zu) und das Kreuz (Elektronen fließen vom Betrachter weg) müssen beim 180°-Durchgang kurz verschwinden und dann gewechselt werden. Wie soll das in der laufenden Animation gelingen? Hier helfen der **Modulo-Operator %** und ein bisschen Nachdenken. Um zu brauchbaren Ergebnissen zu kommen, muss man natürlich die Funktionsweise des *Modulo-Operators* % kennen. Sie lautet: **Der Modulo-Operator % gibt bei einer ganzzahligen Division den Rest zurück.** Bei der Schreibweise muss man die folgende Reihenfolge beachten: Dividend - % - Divisor. Zum besseren Verständnis ist in der folgenden Tabelle ein Zahlenbeispiel aufgeführt.

Dividend	Divisor	Rest	Schreibweise
0	5	0	0%5
1	5	1	1%5
2	5	2	2%5
3	5	3	3%5
4	5	4	4%5
5	5	0	5%5
6	5	1	6%5
7	5	2	7%5
8	5	3	8%5
9	5	4	9%5
10	5	0	10%5
11	5	1	11%5
12	5	2	12%5
13	5	3	13%5
14	5	4	14%5
15	5	0	15%5

Wie man der Tabelle entnehmen kann, verändern sich die Restwerte zyklisch. D.h., sie wiederholen sich regelmäßig, obwohl der Dividend immer weiter ansteigt. Dies machen wir uns für die Animation der Elektronenstromrichtung in der rotierenden Leiterschleife zunutze. Schauen wir uns dazu den folgenden Sketchausschnitt an.

```
int w2 = w % 360;
```

Der Rotationswinkel w nimmt bei fortlaufender Rotation der Leiterschleife immer weiter zu. Bei jeder vollen Drehung um 360°. Der Rotationswinkel w ist also der Dividend und 360 der Divisor. Mittels des *Modulo-Operators %* erhalten wir den jeweiligen Restwert, den wir w2 nennen. Mit einer if-Anweisung können wir nun bestimmen, was bei welchem Restwert geschehen soll.

```
if (w2 >= 10 && w2 <= 170)
{
  stroke(0, 0, 255);
  strokeWeight(8);
  point(-100, 0);
  strokeWeight(2);
  line(105, 5, 95, -5);
  line(95, 5, 105, -5);
}

if (w2 >= 190 && w2 <= 350)
{
  stroke(0, 0, 255);
  strokeWeight(8);
  point(100, 0);
  strokeWeight(2);
  line(-105, 5, -95, -5);
  line(-95, 5, -105, -5);
}
```

Wenn wir möchten, dass nur ein Wechsel zwischen Punkt und Kreuz erfolgen soll, dann würde zwischen den runden Klammern der if-Anweisung stehen (w2 >= 0 && w2 <= 180) und (w2 >= 180 && w2 >= 0). Da wir aber wissen, dass keine Induktionsspannung und damit auch kein Elektronenstrom erzeugt wird, wenn die Enden der Leiterschleife sich parallel zu den Magnetfeldlinien bewegen, möchten wir für diesen Fall – ehrgeizig wie wir sind – auch keinen Punkt und kein Kreuz auf den Schnittflächen der Leiterschleife sehen. Aus diesem Grund ist, wie in dem Sketchausschnitt zu sehen, das Anweisungsintervall verkürzt.

Nun, was gibt es noch zu erwähnen? Zyklische Bewegungen eignen sich besonders gut für die Erstellung eines Videos. Wenn wir mit *saveFrame()* die notwendigen Bilder hierfür aufzeichnen, dann verlangsamt sich aber der Ablauf unserer umfangreichen Animation sehr deutlich.

Sketch 07: Rotierende Leiterschleife

```
// rotierende Leiterschleife

int w = 0; // Winkel in Grad
float Umax = 80;
float ymax = 80;
float Ut; // Spannung zeitabhängig
float U; // nach unten verschobener Spannungswert

void setup()
{
  size(600, 600, P3D);
}

void draw()
{
  background(255);
  translate(width/2, 200);

  // Magnetpole
  noStroke();
  fill(255, 0, 0);
  rect(-120, -160, 240, 30);
  fill(0, 255, 0);
  rect(-120, 131, 240, 30);

  // magn. Feldlinien
  for (float x = -115; x >= -115 && x <= 120; x = x + 25.2)
  {
    stroke(0, 200, 0);
    strokeWeight(2);
    line(x, -131, x, 120);
    fill(0, 200, 0);
    triangle(x-5, 120, x+5, 120, x, 131);
  }

  // Speichern des aktuellen Koordinatensystems in der Zwischenablage
  pushMatrix();

  // Rotation um die z-Achse
  rotateZ(radians(w));
  w++;
```

```
// Leiterschleife
fill(250, 150, 50);
noStroke();
rect(-100, -10, 200, 20);
stroke(250, 150, 50);
fill(255);
ellipse(-100, 0, 20, 20);
ellipse(100, 0, 20, 20);
stroke(100);
strokeWeight(10);
point(0, 0);

// Elektronenstromrichtung
int w2 = w % 360; // Modulo-Operator

if (w2 >= 10 && w2 <= 170)
{
  stroke(0, 0, 255);
  strokeWeight(8);
  point(-100, 0);
  strokeWeight(2);
  line(105, 5, 95, -5);
  line(95, 5, 105, -5);
}

if (w2 >= 190 && w2 <= 340)
{
  stroke(0, 0, 255);
  strokeWeight(8);
  point(100, 0);
  strokeWeight(2);
  line(-105, 5, -95, -5);
  line(-95, 5, -105, -5);
}

// Das in der Zwischenablage gespeicherte Koordinatensystem wird wieder
// eingefügt
popMatrix();

// blaue Sinusfunktion mit x-Achse
for (float x = -300, a = 0; x >= -300 && x <= 300; a = a + 0.5, x = x + 0.5)
{
  float y = -ymax*sin(radians(a));
  fill(0, 0, 255);
  noStroke();
  ellipse(x, y + 300, 5, 5);

  stroke(0);
  strokeWeight(1);
  line(-300, 300, 300, 300);
}

// Ein roter Punkt bewegt sich auf der Sinusfunktion
U = 300 - Umax*sin(radians(w));
stroke(255, 0, 0);
strokeWeight(15);
point(-300 + w, U);
noStroke();
```

4.5 Wechselstromkreis

Siebkette

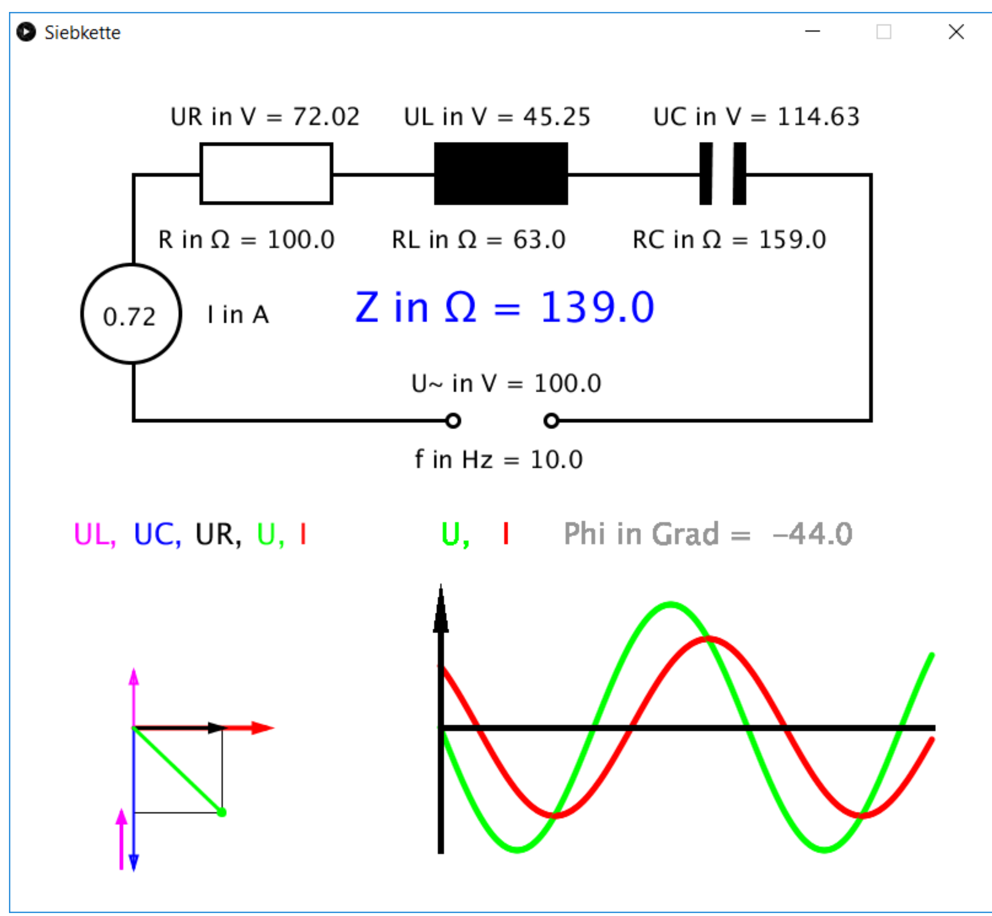


Abbildung 4.19: Reihenschaltung von ohmschem Widerstand, Spule und Kondensator im Wechselstromkreis

Wenn wir die obige Abbildung 4.19 betrachten, dann erahnen wir bereits, was jetzt am Ende des Kapitels Elektrik auf uns zukommt. Viel, viel Programmierarbeit und die korrekte Anwendung unserer physikalischen Kenntnisse. Dies muss man jedoch positiv sehen, denn es ist eine gute Wiederholung und damit eine Festigung des bisher Gelernten.

Bei der Schaltung in Abbildung 4.19 handelt es sich um eine Reihenschaltung von ohmschem Widerstand, Spule und Kondensator, an die eine Wechselspannung angelegt wird. Eine solche Schaltung nennt man auch Siebkette, da der sie durchfließende Strom bei einer bestimmten Frequenz sein Maximum erreicht. Dies ist dann der Fall, wenn die induktive Wirkung der Spule und die kapazitive Wirkung des Kondensators sich gegenseitig aufheben und nur der ohmsche Widerstand die Höhe der Stromstärke bestimmt.

Auffallend in Abbildung 4.19 ist, dass die einfache Summe der Teilspannungen größer ist als die anliegende Gesamtspannung. Bei Wechselstromschaltungen mit induktiven Widerständen R_L und kapazitiven Widerständen R_C müssen die Spannungen unter Berücksichtigung der Phasenverschiebung zwischen Strom und Spannung vektoriell addiert werden. Aus diesem Grund programmieren wir in unserem Sketch *Siebkette* ein dynamisches Zeigerdiagramm, welches auf die eingestellten Werte flexibel reagiert und uns die Richtungen der Einzelspanngen U_R, U_L und U_c sowie die Gesamtspannung U und den Strom I anzeigt. Zusätzlich stellen wir den sinusförmigen Verlauf

von Gesamtspannung und Strom sowie die Phasenverschiebung zwischen den beiden in einem Extradiagramm dar.

Als echte Physiker wollen wir natürlich mit der Schaltung „spielen". D.h., wir möchten die einzelnen Werte ändern können und uns dann in unserer Simulation anschauen, was passiert. Da der Sketch sehr lang und damit auch unübersichtlich wird, entscheiden wir uns für eine objektorientierte Programmierung (OOP). Die Werte, die wir verändern wollen, sollen übersichtlich im Hauptsketch einzustellen sein und der Rest kommt in den Nebensketch. Dazu erstellen wir eine Klasse mit dem Namen *Sieb*. Sie enthält die Instanzvariablen und die Methoden (Funktionen). Wie dies funktioniert, haben wir bei Kapitel 3.3 in dem Abschnitt *Gravitationsgesetz vektoriell 02* und dem Abschnitt *Sonnensystem* gelernt. Erinnern wir uns nochmal an den prinzipiellen Aufbau einer Klasse.

KLASSENNAME

{

INSTANZVARIABLEN

KONSTRUKTOR

METHODEN (FUNKTIONEN)

}

Das Aufstellen der Instanzvariablen dürfte einem guten Physikschüler nicht schwerfallen, sodass wir dies hier auch nicht kommentieren müssen. Interessanter ist hier schon der folgende Inhalt des Konstruktors.

```
Sieb(float Rtemp, float Ltemp, float Ctemp)
{
  R = Rtemp;
  L = Ltemp;
  C = Ctemp;
}
```

Er sorgt mit der Deklaration der Variablen Rtemp, Ltemp, Ctemp und den Zuweisungen R = Rtemp, L = Ltemp und C = Ctemp dafür, dass wir im Hauptsketch die Werte für den ohmschen Widerstand R, die Induktivität L und die Kapazität C frei wählen können und die Methoden (Funktionen) im Nebensketch diese Werte bei ihrer Ausführung berücksichtigen.

Wie man einen Schaltplan zeichnet (siehe *void schaltung()*), dies haben wir schon in Kapitel 4.1 kennengelernt. Auch an dieser Stelle sei nochmal darauf hingewiesen, dass man mit *Sketch → Optimieren* sehr viel Zeit und Nerven sparen kann. Ansonsten benötigt die Methode *void schaltung()* keine weiteren Erläuterungen.

Bei der Methode *void messwerte()* soll nochmal an dem Beispiel von UR daran erinnert werden, was die folgende Zeile bedeutet.

```
text("UR in V = " +(float)round(100*UR)/100, 130, 60);
```

Da UR eine float-Zahl ist, wir aber in unserem Schaltplan nicht zu viele Nachkommastellen angeben möchten, runden wir auf zwei Stellen hinter dem Komma. Dazu multiplizieren wir UR mit 100 und runden dann mit *round()* auf einen ganzzahligen Wert. Anschließend teilen wir durch 100

und sorgen so zusammen mit *float* dafür, dass wir eine Zahl mit zwei Nachkommastellen erhalten. Bei der Multiplikation und Division mit der Zahl 10 erhalten wir eine Nachkommastelle.

Bei der Methode *void zeiger()* wird das Koordinatensystem mit *translate(100, 550)* verschoben. Dies erleichtert die Erstellung des Zeigerdiagramms und später bei der Methode *void sinus()* auch die Erstellung der zeichnerischen Darstellung der U(t)- und I(t)-Funktion. In beiden Fällen muss man jedoch daran denken, dass bei Processing die positive y-Achse nach unten zeigt und wir den Nullpunkt verschoben haben. Da der Strom I, der die Schaltung durchfließt sehr gering ist, wurde er zwecks einer besseren Darstellung bei *void sinus()* mit dem Faktor 100 multipliziert.

Zwei Anweisungen in *void sinus()* sind neu. Mit **atan()** kann man den Tangens zurück ins Bogenmaß verwandeln und mit der Anweisung **degrees()** kann man das Bogenmaß in eine Gradzahl verwandeln.

Soweit zum Nebensketch. Kommen wir nun zum Hauptsketch. Ohne die zusätzlichen Erläuterungen ist der Hauptsketch sehr übersichtlich und trotzdem können wir hier alle physikalischen Größen bequem verändern (siehe unten). Dies ist u.a. der große Vorteil des Arbeitens mit einer Klasse.

Hauptsketch ohne Erläuterungen:

```
Sieb Franz;

void setup()
{
  size (800, 700);
  Franz = new Sieb(100.0, 1.0, 100E-6);
}

void draw()
{
  background(255);
  Franz.Schaltung();
  Franz.Messwerte(100, 10);
  Franz.Zeiger();
  Franz.Sinus(10);
}
```

Die Erläuterungen zu den einzelnen Sketchzeilen findet man im Sketch selbst.

Nun kann man mit den einzelnen Werten spielen und schauen, was sich im Fenster alles verändert. Welche Resonanzfrequenz f_0 ergibt sich für die folgende Wertekombination R = 100 Ω, L = 1 H und C = 100 µF? Welcher Strom fließt dann durch die Siebkette? Welche Teilspannungen liegen dann an den einzelnen Widerständen an? Wie groß ist die Phasenverschiebung zwischen Strom I und Spannung U?

Da unser Pixelfenster relativ klein ist, sollte man bei der Wahl der einzelnen Werte darauf achten, dass sie auch vernünftig im Fenster dargestellt werden können. Dies ist auch der Grund dafür, dass sich die Frequenzwerte f_1 in *void messwerte()* und f_2 in *void sinus()* unterscheiden. f_1 dient zur richtigen Berechnung der Widerstandswerte von R_L und R_C. Die hiervon unabhängige Wahl von f_2 erlaubt dagegen eine sinnvolle zeichnerische Darstellung der U(t)- und I(t)-Funktion.

Sketch 08: Siebkette

Hauptsketch

```
// Siebkette

Sieb Franz; // Die Siebkette aus der Klasse Sieb bekommt den Namen Franz

void setup()
{
  size (800, 700);

  // Werte für R in Ohm, L in Henry und C in Farad wählen
  Franz = new Sieb(100.0, 1.0, 100E-6);
}

void draw()
{
  background(255);

  // Die Methoden (Funktionen) aus der Klasse Sieb werden aufgerufen
  Franz.schaltung(); // Schaltplan wird gezeichnet
  Franz.messwerte(100, 10); // Werte für U in Volt und f1 in Hertz
                            // eingeben
  Franz.zeiger(); // Zeigerdiagramm wird berechnet und gezeichnet
  Franz.sinus(10); // Wert für f2 in Hertz eingeben
  // f2 statt f1 dient zur besseren Darstellung der U- und I-Funktion
}
```

Nebensketch

```
class Sieb // KLASSENNAME
{
  // INSTANZVARIABLEN

  float U; // Spannung in Volt
  float UR; // Teilspannung in Volt
  float UL; // Teilspannung in Volt
  float UC; // Teilspannung in Volt
  float R; // ohmscher Widerstand in Ohm
  float RL; // induktiver Widerstand in Ohm
  float RC; // kapazitiver Widerstand in Ohm
  float Z; // Wechselstromwiderstand der Siebkette
  float I; // Strom in Ampere
  float f; // Frequenz in Hertz
  float L; // Induktivität in Henry
  float C; // Kapazität in Farad

  // KONSTRUKTOR

  Sieb(float Rtemp, float Ltemp, float Ctemp)
  {
    R = Rtemp;
    L = Ltemp;
    C = Ctemp;
  }

  // METHODEN (FUNKTIONEN)

  void schaltung()
```

```
{
  // Die Leitungen des Stromkreises werden gezeichnet
  stroke(0);
  strokeWeight(3);
  fill(255);
  beginShape();
  vertex(355, 300);
  vertex(100, 300);
  vertex(100, 100);
  vertex(700, 100);
  vertex(700, 300);
  vertex(440, 300);
  endShape();

  // Kontakte der Spannungsquelle
  ellipse(360, 300, 10, 10);
  ellipse(440, 300, 10, 10);

  // Strommessgerät
  ellipse(98, 213, 80, 80);

  // Widerstand R
  stroke(0);
  strokeWeight(3);
  fill(255);
  rect(155, 75, 106, 48);

  // Widerstand RL
  stroke(0);
  strokeWeight(3);
  fill(0);
  rect(346, 75, 106, 48);

  // Widerstand RC
  stroke(0);
  strokeWeight(3);
  fill(0);
  rect(562, 75, 35, 48);
  stroke(255);
  strokeWeight(17);
  line(580, 51, 579, 124);
}

void messwerte(float U, float f1)
{
  // Berechnung von RL, RC und Z
  RL = 2*PI*f1*L;
  RC = 1/(2*PI*f1*C);
  Z = sqrt(pow(R, 2) + pow((RL-RC), 2));

  // Berechnung von I und den Teilspannungen UR, UL und UC
  I = U/Z;
  UR = R*I;
  UL = RL*I;
  UC = RC*I;

  // Beschriftung
  fill(0);
  textSize(20);
  text("f in Hz = " +f1, 329, 339);
  text("U\u007E in V = " +U, 326, 277);
```

```
  text("UR in V = " +(float)round(100*UR)/100, 130, 60);
  text("UL in V = " +(float)round(100*UL)/100, 320, 60);
  text("UC in V = " +(float)round(100*UC)/100, 523, 60);
  text("R in \u03A9 = " +R, 120, 160);
  text("RL in \u03A9 = " +(float)round(10*RL/10), 310, 160);
  text("RC in \u03A9 = " +(float)round(10*RC/10), 506, 160);
  text("" +(float)round(100*I)/100, 75, 223);
  text("I in A", 160, 221);
  textSize(34);
  fill(0, 0, 255);
  text("Z in \u03A9 = " +(float)round(10*Z/10), 280, 220);
}

void zeiger()
{
  translate(100, 550);

  // Rechteck
  noFill();
  stroke(0);
  strokeWeight(1);
  rect(0, 0, UR, (-UL + UC));

  // Zeiger für I
  stroke(255, 0, 0);
  strokeWeight(4);
  line(0, 0, 150*I, 0);
  triangle(150*I-10, -3, 150*I, 0, 150*I-10, 3);

  // Zeiger für UR, UL, UC, UL - UC und U
  stroke(0);
  strokeWeight(3);
  line(0, 0, UR, 0);
  strokeWeight(3);
  triangle(UR-10, -3, UR, 0, UR-10, 3);
  stroke(255, 0, 255);
  strokeWeight(2);
  line(0, 0, 0, -UL);
  triangle(-3, -UL+10, 0, -UL, 3, -UL+10);
  stroke(0, 0, 255);
  strokeWeight(2);
  line(0, 0, 0, UC);
  triangle(-3, +UC-10, 0, UC, 3, UC-10);
  stroke(255, 0, 255);
  strokeWeight(3);
  line(-10, UC, -10, -UL+UC);
  triangle(-13, -UL+8+UC, -10, -UL+UC, -7, -UL+8+UC);
  stroke(0, 255, 0);
  line(0, 0, UR, (-UL + UC)); // U
  ellipse(UR, (-UL + UC), 5, 5); // Kinderpfeilspitze

  // Texte
  textSize(24);
  fill(255, 0, 255);
  text("UL,", -50, -150);
  fill(0, 0, 255);
  text("UC,", 0, -150);
  fill(0);
  text("UR,", 50, -150);
  fill(0, 255, 0);
  text("U,", 100, -150);
```

```
    fill(255, 0, 0);
    text("I", 135, -150);
  }

  void sinus(float f2)
  {
    float Phi = atan((RL - RC)/R); // Berechnung der Phasenverschiebung

    // Sinusfunktion mit Achsen
    for (float x = 250, t = 0; x >= 250 && x <= 650; t = t + 0.0004,
         x = x + 1)
    {
      // Wechselspannung
      float Ut = Z*I*sin(2*PI*f2*t);
      fill(0, 255, 0);
      noStroke();
      ellipse(x, Ut, 5, 5);

      // Wechselstrom
      float It = I*sin(2*PI*f2*t+Phi);
      fill(255, 0, 0);
      noStroke();
      ellipse(x, 100*It, 5, 5);
    }
      // Koordinatensystem
      stroke(0);
      strokeWeight(5);
      line(250, 0, 650, 0);
      line(250, 100, 250, -100);
      fill(0);
      triangle(247, -80, 250, -100, 253, -80);

      // Text für U- und I-Funktion
      textSize(24);
      fill(0, 255, 0);
      text("U,", 250, -150);
      fill(255, 0, 0);
      text("I", 300, -150);
      fill(150);
      text("Phi in Grad =  " +(float)round(100*(degrees(Phi))/100),
           350, -150);

  }
}
```

4.6 Zusammenfassung

Zeichnen von Vielecken

Mit den Funktionen *beginShape, vertex()* und *endShape()* können beliebige Vielecke gezeichnet werden.

beginShape() Startet die Aufzeichnung der zu zeichnenden Form.

vertex() Hiermit legt man die Koordinaten x, y bzw. x, y, z der Eckpunkte der zu zeichnenden Form fest.

endShape() Beendet die Aufzeichnung der zu zeichnenden Form.

Runden auf Nachkommastellen

Wenn man Zahlenwerte in Zeichnungen darstellt, dann ist es oft unzweckmäßig, zu viele Nachkommastellen anzugeben. Besser ist es, auf eine oder zwei Nachkommastellen zu runden. Dies soll nun an einem Zahlenbeispiel erklärt werden. Die Zahl 2,3423107836 soll auf zwei Stellen hinter dem Komma gerundet werden. Dazu multiplizieren wir die Zahl mit dem Faktor 100, runden sie auf einen ganzzahligen Wert und teilen dann durch 100.

2,3423107836 → 234,23107836 → 234 → 2,34

In Processing sieht dies so aus:

```
text("U in V = " +(float)round(100*U)/100, 130, 60);
```

Das Wort *float* ist notwendig, damit das Ergebnis der Division auch mit den Nachkommastellen angegeben wird.

Unicode

Mit Unicode kann man alle möglichen Sonderzeichen im Processing-Fenster darstellen. Beispiel: Der Unicode für das Zeichen Ω lautet 03A9 und wird in Processing wie folgt eingegeben: text("R in \u03A9 = " +R, 213, 160);
Im Fenster steht dann z.B.: R in Ω = 45 an dem Ort x = 213 und y = 160. Den Unicode für Sonderzeichen findet man in Textbearbeitungsprogrammen bei Sonderzeichen.

Sketchoptimierung

In der PDE auf der Karteikarte *Sketch* → *Optimieren* aufrufen. Anschließend können mit gedrückter Maustaste Koordinatenwerte geändert werden. Diese Änderungen werden live im Processing-Fenster anzeigt. Dies hilft sehr bei der Erstellung von umfangreicheren Zeichnungen.

Filme erstellen

saveFrame()

Die Anweisung *saveFrame()* speichert bei jedem Durchlauf von *void draw()* ein Bild des geöffneten Fensters ab. Beispiel: *saveFrame(„Bilder/B_####.jpg“)*
Im Beispiel werden die Bilder im Ordner Bilder mit der Bezeichnung B_0001 bis B_9999 abgespeichert.

Movie Maker

Die mit *saveFrame()* abgespeicherten Bilder fügt *Movie Maker* zu einem Film zusammen. Den *Movie Maker* findet man auf der Karteikarte *Tools*.

keyPressed

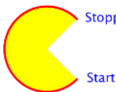
keyPressed ist eine boolesche Systemvariable. Sie ist wahr, wenn eine beliebige Taste auf der Tastatur gedrückt wird, und falsch, wenn keine Tasten gedrückt werden. Zu beachten ist, dass es auch eine Funktion ***keyPressed ()*** gibt. Sie wird aufgerufen, wenn eine beliebige Taste auf der Tastatur gedrückt wird.

Modulo-Operator %

Der *Modulo-Operator %* gibt den Rest einer Division zurück. Dies ist bei der Erstellung von Animationen sehr hilfreich, da der *Modulo-Operator %* ein zyklisches Verhalten zeigt. So nimmt zum Beispiel bei einer Kreisbewegung der Winkel immer weiter zu. Teilen wir diesen Winkel

jedoch durch 360, so erhalten wir mit dem Modulo-Operator fortlaufend Werte zwischen 0 und 359. Beispiel: *Winkel%360*

arc()
Mit *arc()* kann man offene und geschlossene Bögen zeichnen. Hierbei muss man berücksichtigen, dass bei Processing der Winkel im Uhrzeigersinn zunimmt. Hier ist ein Beispiel für einen nicht geschlossenen roten Bogen mit gelber Füllung: arc(x, y, 200, 200, PI/4, 7*PI/4); (Die Farbangaben sind in dieser Zeile nicht enthalten.)

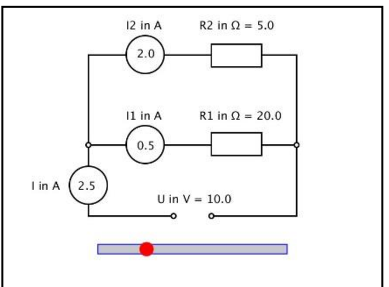


atan()
atan() gibt den Winkel im Bogenmaß zu einem Tangenswert an.

degrees()
degrees() verwandelt das Bogenmaß in eine Gradzahl.

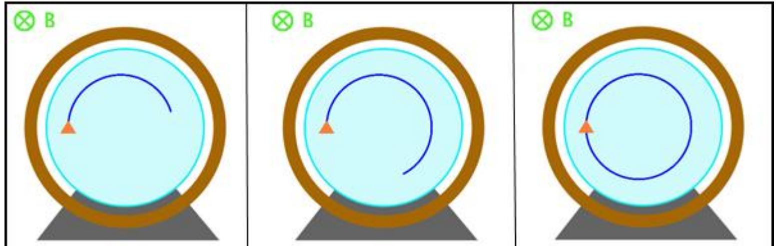
4.7 Aufgaben

1. Schreibe einen Sketch, mit dem sich die rechts abgebildete Parallelschaltung simulieren lässt. Wie im Sketch *Spannungsteiler* soll die Spannung mit einem Schieber regelbar sein.



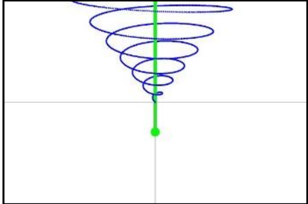
2. Ändere den Sketch *Massenspektrometer* so um, dass die Apparatur gleichzeitig von drei Isotopen mit unterschiedlicher Masse aber gleicher Geschwindigkeit durchflogen wird. Die Flugbahnen der Isotope sollen in Abhängigkeit von ihrer Masse wie in der Abbildung rechts dargestellt werden.

3. Um die Programmierung zu vereinfachen, soll mittels Processing die Elektronenbewegung in einem Fadenstrahlrohr nur qualitativ simuliert werden. Qualitativ bedeutet hier, dass die Zahlenwerte für die Ladung Q, die Flussdichte B und die Masse m der Elektronen frei gewählt werden können. Im Unterschied zum Sketch *Massenspektrometer* soll die Lorentzkraft diesmal jedoch vektoriell mit dem Kreuzprodukt $\vec{F}_L = Q \cdot (\vec{v} \times \vec{B})$ berechnet werden, sodass die Lorentzkraft stets zum Zentrum der Kreisbewegung zeigt. Der Flug der Elektronen auf der Kreisbahn soll, wie unten dargestellt, in Zeitlupe zu beobachten sein. Der braune Kreisring stellt das Helmholtzspulenpaar dar. Alle Farben sollen in diesem Sketch mithilfe des *Color Selectors* bestimmt werden.

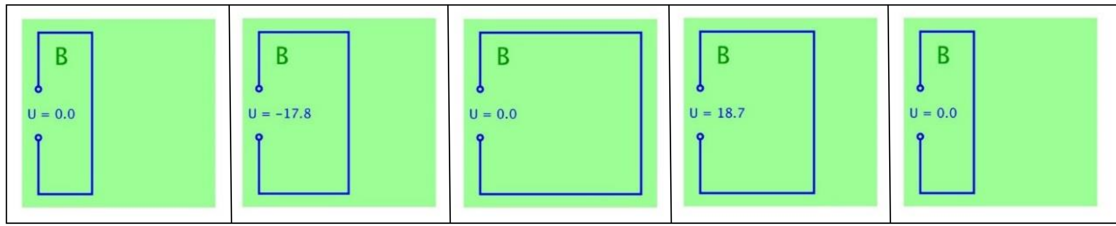


Tipp: Da man bei einer Simulation nicht mit unendlich kleinen Zeitschritten rechnen kann, schließt sich die Kreisbahn der Elektronen bei wiederholtem Durchlauf nicht korrekt. Will man diese Abweichungen möglichst gering halten, wählt man möglichst kleine Zeitschritte und, damit die Simulation nun nicht zu langsam abläuft, eine möglichst hohe Bildwiederholungsrate. Weiterhin setzt man die Werte für die Orts- und Geschwindigkeitskoordinate nach jedem vollständigen Kreisumlauf wieder auf die Ausgangswerte zurück. Alternativ kann man speziell bei diesem Sketch auch die Simulation nach einem Kreisumlauf stoppen.

4. Verändere den Sketch *Schraubenbahn* derart, dass Processing eine dreidimensionale Spiralbahn zeichnet, die um die x-Achse rotiert (siehe Abbildung rechts). Erstelle von dieser Simulation ein Video. Die Bilder für das Video sollen nur dann aufgezeichnet werden, wenn eine beliebige Maustaste gedrückt gehalten wird.

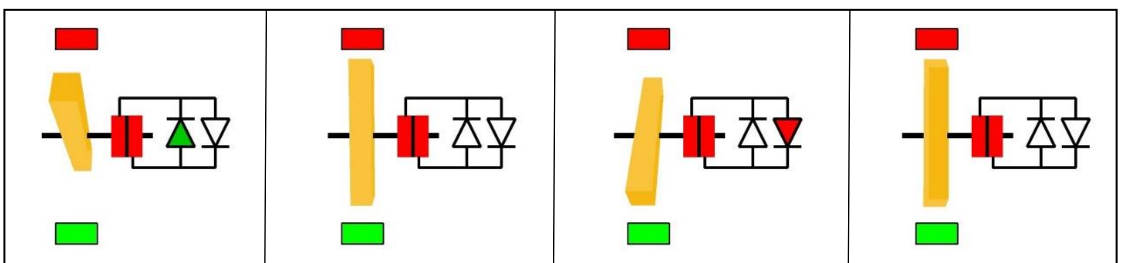


5. Wenn sich der magnetische Fluss in einer Leiterschleife zeitlich ändert, dann kann man an ihren Enden eine Spannung messen. Zwei unterschiedliche Möglichkeiten der Spannungserzeugung mittels einer Leiterschleife wurden in Kapitel 4.4 näher untersucht. Eine dritte Möglichkeit besteht darin, die Größe einer Leiterschleife in einem Magnetfeld zu ändern. Schreibe einen Sketch, der eine Leiterschleife simuliert, deren Größe sich fortlaufend periodisch ändert. Diese Leiterschleife soll sich in einem homogenen Magnetfeld befinden und die Induktionsspannung soll, auf eine Nachkommastelle gerundet, an den Enden der Leiterschleife angezeigt werden (siehe Abbildung).



Tipp: Hilfreich bei Planung des Sketches sind die Gleichungen von Kapitel 4.4.

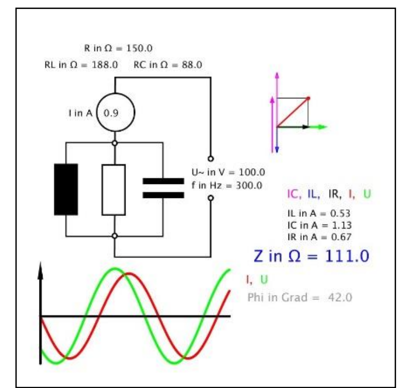
6. Die folgende Abbildung zeigt einen schematisch gezeichneten Wechselstromgenerator in Seitenansicht. An seine beiden Schleifringe sind zwei Leuchtdioden mit entgegengesetzter Polung angeschlossen, die je nach Stellung der Spule abwechselnd aufleuchten. Steht die zwischen Nord- und Südpol rotierende Spule senkrecht, dann leuchten die beiden Leucht-dioden für einen kurzen Moment nicht (siehe Abbildung).



Die Spule (hellbraun in der obigen Abbildung) soll in der 3D-Simulation schematisch als Quader mittels der Funktion *box()* dargestellt werden. Sie soll um die x-Achse rotieren. Informationen zu der Funktion *box()* findet man in der Referenz von Processing. Die Leitungen zwischen den beiden Schleifringen und den Leuchtdioden kann man mit der Funktion *vertex()* zeichnen. Mit *Sketch* → *Optimieren* kann man die einzelnen Punkte leicht in die richtige Position bringen. Mithilfe des Modulo-Operators % gelingt es, die Leuchtdioden abwechselnd und im richtigen Moment leuchten zu lassen. Lese dir vor Beginn deiner Programmierung nochmal den Text zum Sketch *rotierende_Leiterschleife* durch.

Erstelle auch von dieser Simulation ein Video. Die Bilder für das Video sollen aber nur dann aufgezeichnet werden, wenn eine beliebige Maustaste gedrückt gehalten wird.

7. Schreibe analog zum Sketch *Siebkette* einen Sketch für die rechts dargestellte Parallelschaltung von induktivem, ohmschem und kapazitivem Widerstand. Wie im Sketch *Siebkette* sollen die Werte für den Strom I, den Scheinwiderstand Z und die Phasenverschiebung φ zwischen Strom und Spannung in Abhängigkeit von der Frequenz und der Spannung dargestellt werden. Verwende die folgenden Werte: $U_{\sim}$ = 100 V, R = 150 Ω, L = 0,1 H, C = 6 μF.



Überprüfe die im Fenster angezeigten Werte durch eine Rechnung mittels Bleistift und Papier. Für welche Frequenz beträgt der Scheinwiderstand Z = 150 Ω? Wie groß ist dann die Phasenverschiebung zwischen Strom und Spannung?

5 Schwingungen

Was erwartet uns?

Color Selector, abs(), Linienteilstücke miteinander verbinden, importieren von Bibliotheken, Soundausgabe mittels der Bibliotheken Minim und Sound

5.1 Harmonische Schwingungen

Federpendel

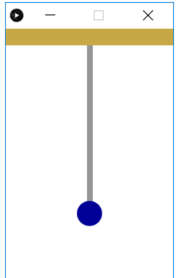


Abbildung 5.1: Federpendel

Wenn bei einer Feder die rücktreibende Kraft proportional zur Auslenkung ist, dann kann die Differenzialgleichung hierfür mit dem Lösungsansatz $y = y_{max} \cdot \sin(\omega \cdot t)$ gelöst werden, wenn man $\omega = \sqrt{\frac{D}{m}}$ setzt. Damit erhalten wir in Processing-Schreibweise y = ymax*sin((sqrt(D/m))*t)

Diese Gleichung beschreibt eine harmonische Schwingung und ist das Herzstück des folgenden Sketches. Wenn wir die Schraubenfeder als dicke graue Linien zeichnen, dann stellt uns die Programmierung des Sketches vor keine großen Probleme. Auch wenn wir mit einer if-else-Anweisung dafür sorgen, dass die Feder überdehnt wird, wenn die Masse größer als 8 kg ist.

Trotzdem wollen wir etwas Neues lernen. In der obigen Abbildung sehen wir, dass die Schraubenfeder an einem braunen Balken befestigt ist. Doch wie muss man die Farben Rot, Grün und Blau kombinieren, damit sich die Farbe Braun ergibt? Einfach ausprobieren? Dies kann dauern. Einfacher ist es in der PDE bei Karteikarte *Tools* auf *Farbauswahl* zu klicken. Dann öffnet sich das Fenster ***Color Selector*** (Abb. 5.2) und man kann ganz bequem mit dem Mauszeiger die gewünschte Farbe auswählen. Den so erhaltenen Wert (hier #C6A946) kopiert man dann in seinen Sketch.

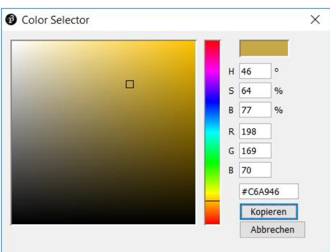


Abbildung 5.2: Farben auswählen im Color Selector

Sketch 01: Federpendel

```
// Federpendel

float D = 0.05; // Federkonstante
float m = 3; // Masse in kg
float y; // Auslenkung (Elongation)
float ymax = 50; // maximale Auslenkung (Amplitude)
float t; // Zeit

void setup()
{
  size(200, 300);
}

void draw()
{
  background(255);
  translate(100, 200);

  if (m >= 2 && m <= 8)
  {
    y = ymax*sin((sqrt(D/m))*t);
    t = t + 0.5;

    // Feder
    stroke(150);
    strokeWeight(7);
    line(0, -200, 0, y);
    noStroke();

    // Kugel
    fill(0, 0, 150);
    ellipse(0, y, 10*m, 10*m);

    // Balken
    fill(#F2A70F);
    rect(-100, -200, 200, 20);
  } else
    // überdehnte Feder
    stroke(150);
  strokeWeight(3);
  line(0, -200, 0, 100);

  // Balken
  noStroke();
  fill(#C6A946);
  rect(-100, -200, 200, 20);
}
```

Schwingkreis

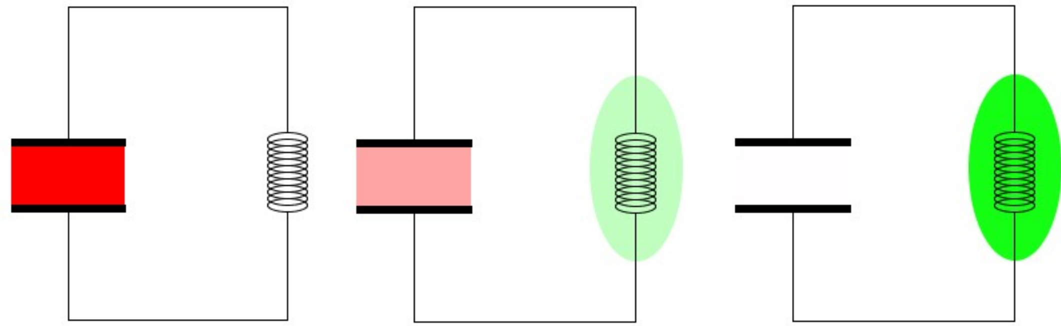


Abbildung 5.3: Schwingkreis ohne Dämpfung

Schließt man einen geladenen Kondensator an eine Spule an, so entlädt sich der Kondensator über die Spule. Der hierbei durch die Spule fließende Strom baut ein Magnetfeld auf, welches aber wieder zusammenbricht, wenn der Kondensator entladen ist. Hierdurch entsteht eine Induktionsspannung, die den Kondensator in umgekehrter Richtung wieder auflädt. Besitzt dieser Stromkreis keinen ohmschen Widerstand und ist die Abstrahlung vernachlässigbar, dann wiederholt sich der Vorgang fortlaufend. In einem solchen Schwingkreis führen die Elektronen harmonische Schwingungen mit der Schwingungsdauer $T = 2\pi \cdot \sqrt{L \cdot C}$ aus.

Diesen Vorgang wollen wir entsprechend der Abbildung 5.3 mittels Processing simulieren. Die Stärke des jeweiligen Feldes soll hierbei durch die Intensität der Farben dargestellt werden. Die Farbe Rot stellt das elektrische Feld im Kondensator dar. Sie soll in ihrer Intensität immer weiter bis zum Wert Null (Farbe Weiß) abnehmen und anschließend wieder bis zum Maximum zunehmen. Während das elektrische Feld abnimmt, nimmt das magnetische Feld und damit die Intensität der Farbe Grün zu. Wie programmiert man dies?

Wenn man bei *fill(255, 0, 0)* nur den Wert für die Farbe Rot verändert, dann erhält man bei *fill(0, 0, 0)* die Farbe Schwarz. Laut Vorgabe soll aber der Raum zwischen den beiden Kondensatorplatten die Farbe Weiß erhalten, wenn das elektrische Feld den Wert Null hat. Aus diesem Grund müssen wir die Werte für die Farben Grün und Blau verändern und den Wert für Rot unverändert bei 255 lassen, denn *fill(255, 255, 255)* ergibt die Farbe Weiß und *fill(255, 0, 0)* ein sattes Rot. *fill(255, 150, 150)* ergibt zum Beispiel ein blasses Rot. Zu beachten ist, dass die Werte für Grün und Blau gleich groß sein müssen und beide gleichmäßig zu- und abnehmen. Für das Magnetfeld gilt entsprechendes.

Jetzt müssen wir uns nur noch überlegen, wie wir programmtechnisch die Farbwerte zyklisch zu- und abnehmen lassen. Wenn wir an das Federpendel in unserem letzten Sketch denken, dessen Elongation periodisch zu- und abnimmt, dann ist alles klar. Eine Sinus- oder Cosinusfunktion eignet sich bestens hierfür. Wie wir jedoch wissen, gibt es keine negativen Farbwerte. Doch dieses Problem lösen wir mit der Funktion **abs()**. Sie gibt den Betrag der Zahl oder Funktion an, die zwischen den runden Klammern steht. Abbildung 5.4 veranschaulicht dies.

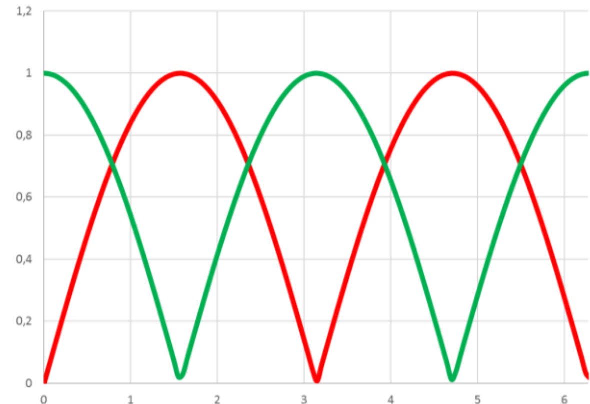


*Abbildung 5.4: Absolutbeträge von **Sinus** und **Cosinus***

Die in Abbildung 5.4 dargestellten Funktionen ändern sich periodisch zwischen 0 und 1. Deshalb müssen wir sie noch mit 255 multipilieren. Wenn wir die Bezeichnung EF für die Farbwerte des E-Feldes und BF für die Farbwerte des B-Feldes festlegen, dann erhalten wir:

$$EF = 255 \cdot \sin(\omega \cdot t) \quad und \quad BF = 255 \cdot \cos(\omega \cdot t)$$

Also: *fill(255, EF, EF)* für das E-Feld und *fill(BF, 255, BF)* für das B-Feld. EF soll beim Start der Simulation Null sein und BF maximal, da der Kondensator vollständig geladen ist und in der Spule noch kein Magnetfeld vorhanden ist.
Da die Schwingungsdauer T des Schwingkreises von der Größe der Induktivität und der Größe der Kapazität abhängt, müssen wir dies noch in die obigen Gleichungen einbringen. Für die Schwingungsdauer T gilt: $T = 2\pi \cdot \sqrt{L \cdot C}$. Setzen wir dies in die obigen Gleichungen ein.

$$EF = 255 \cdot \sin(\omega \cdot t) = 255 \cdot \sin\left(\frac{2\pi}{T} \cdot t\right) = 255 \cdot \sin\left(\frac{2\pi}{2\pi\sqrt{L \cdot C}} \cdot t\right) = 255 \cdot \sin\left(\frac{t}{\sqrt{L \cdot C}}\right) \quad \text{und}$$

$$BF = 255 \cdot \cos\left(\frac{t}{\sqrt{L \cdot C}}\right)$$

Der Rest des Sketches *Schwingkreis* dürfte keine großen Kopfschmerzen bereiten. Und nun viel Spaß beim Spielen mit den Werten für L und C.

Sketch 02: Schwingkreis

```
// Schwingkreis

float L = 1; // Induktivität in H
float C = 1; // Kapazität in F
float EF; // Farbwert für das E-Feld
float BF; // Farbwert für das B-Feld
float t; // Abspielzeit

void setup()
{
  size(400, 400);
}
```

```
void draw()
{
  background(255);

  // sich ändernde Stärke des E- und B-Feldes
  EF = abs(255*sin(t/sqrt(L*C)));
  BF = abs(255*cos(t/sqrt(L*C)));
  t = t + 0.01;

  noStroke();
  // Farbe des E-Feldes
  fill(255, EF, EF);
  rect(90, 183, 86, 45);

  // Farbe des B-Feldes
  fill(BF, 255, BF);
  ellipse(300, 200, 70, 140);

  // Spule
  for (float a = 163; a >= 212 || a >= 113; a = a - 5)
  {
    stroke(0);
    noFill();
    ellipse(299, 65 + a, 30, 10);
  }

  // Kondensator
  fill(0);
  rect(90, 178, 86, 5);
  rect(90, 228, 86, 5);

  // Leiterbahnen
  noFill();
  beginShape();
  vertex(133, 178);
  vertex(133, 78);
  vertex(299, 78);
  vertex(299, 173);
  endShape();

  beginShape();
  vertex(133, 233);
  vertex(133, 315);
  vertex(299, 315);
  vertex(299, 233);
  endShape();
}
```

5.2 Gedämpfte Schwingungen

Gedämpfte Schwingung

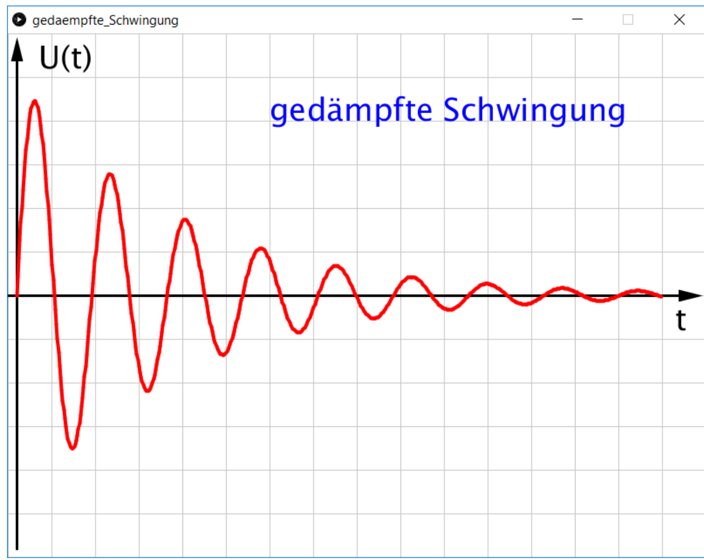


Abbildung 5.5: Gedämpfte Schwingung eines Schwingkreises

In der Realität verlieren das Federpendel wie auch der Schwingkreis beim Schwingungsvorgang fortlaufend Energie, die in Wärme umgewandelt wird, sodass ihre Amplituden y_{max} bzw. U_{max} mit der Zeit immer kleiner werden (siehe Abb. 5.5). Dies wird durch die folgenden Gleichungen beschrieben.

Federpendel $y(t) = y_{max} \cdot e^{-k \cdot t} \cdot \sin(\omega \cdot t)$

Schwingkreis $U(t) = U_{max} \cdot e^{-k \cdot t} \cdot \sin(\omega \cdot t)$

Im realen Schwingkreis ist es der ohmsche Widerstand von Spule und Leitungen, der die Umwandlung von elektrischer Energie in Wärme bewirkt. Zusätzlich vergrößert der ohmsche Widerstand auch die Schwingungsdauer. Es gilt:

$$k = \frac{R}{2L} \quad \text{und} \quad T = \frac{2\pi}{\sqrt{\frac{1}{LC} - \frac{R^2}{4L^2}}} \quad \text{und damit} \quad U(t) = U_{max} \cdot e^{-\frac{R}{2L} \cdot t} \cdot \sin(\sqrt{\frac{1}{LC} - \frac{R^2}{4L^2}} \cdot t)$$

Die Gleichung sieht auf den ersten Blick ein wenig erschreckend aus. Doch wir sollten inzwischen so viel gelernt haben, dass wir diese Gleichung problemlos in Processing eingeben können. Wenn wir im Vorfeld k ausrechnen und bedenken, dass die positive y-Achse bei Processing nach unten zeigt, dann sieht die Lösung wie folgt aus:

```
U = -Umax*exp(-k*t)*sin(t*sqrt((1/(L*C))-(pow(R, 2)/(4*pow(L, 2)))));
```

Interessant ist noch der folgende Sketchausschnitt.

```
// Der Graph wird gezeichnet.
stroke(255, 0, 0);
strokeWeight(4);
line(xvor, Uvor, x, U);
xvor = x;
Uvor = U;
```

```
      x = x + 1.2;
```

Die Anweisung *line()* zeichnet eine Linie zwischen dem Punkt mit den Koordinaten xvor, Uvor und dem Punkt mit den Koordinaten x, U. Beim nächsten Durchlauf von *void draw()* zeichnet Processing wieder eine Linie. Damit ihr Anfangspunkt mit dem Endpunkt der vorhergehenden Linien zusammenfällt, schreiben wir xvor = x und Uvor = U. Mit diesem Trick erhalten wir einen Graphen, der aus vielen kleinen zusammenhängenden Linien besteht.

Nachdem nun alles Wichtige erklärt ist, heißt es wieder: Spielen! Spielen! Spielen! Wie hängt die Schwingung von L, C und R ab? Einfach ausprobieren!

Sketch 03: gedämpfte Schwingung

```
// Gedämpfte Schwingung

float t; // Abspielzeit
float U; // Spannung in V
float Umax = 250;
float Uvor; // Spannungswert vor dem jeweiligen Durchlauf von void
draw()
float k; // Dämpfungsfaktor
float x; // Ablenkung in x-Richtung
float xvor; // x-Wert vor dem jeweiligen Durchlauf von void draw()
float L = 0.8; // Induktivität in Henry
float C = 0.8; // Kapazität in Farad
float R = 0.15; // Ohmscher Widerstand in Ohm

void setup()
{
  size(800, 600);
  background(255);

  // Raster wird gezeichnet.
  stroke(200);
  strokeWeight(1);
  for (float x = 10; x < width; x += 50)
  {
    line(x, 0, x, height);
  }
  for (float y = 0; y < height; y += 50)
  {
    line(0, y, width, y);
  }

  // Achsen werden gezeichnet
  stroke(0);
  strokeWeight(3);
  line(0, 300, 790, 300);
  line(10, 10, 10, 590);
  fill(0);
  triangle(5, 30, 10, 10, 15, 30);
  triangle(770, 295, 790, 300, 770, 305);

  // Text wird eingefügt
  textSize(36);
  fill(0);
  text("U(t)", 35, 40);
  text("t", 765, 340);
  fill(0, 0, 255);
```

```
  text("gedämpfte Schwingung", 300, 100);
}

void draw()
{
  translate(10, 300); // Verschiebung des Koordinatenursprungs

  // Funktion für die gedämpfte Schwingung
  k = R/(2*L);
  U = -Umax*exp(-k*t)*sin(t*sqrt((1/(L*C))-(pow(R, 2)/(4*pow(L, 2)))));
  t = t + 0.07;

  // Der Graph wird gezeichnet.
  stroke(255, 0, 0);
  strokeWeight(4);
  line(xvor, Uvor, x, U); // rot

  /* Den Variablen xvor und Uvor wird der aktuelle x-Wert
   und U-Wert zugeordnet. Dieser Wert wird dann in der nächsten
   Runde für xvor und Uvor eingesetzt werden. Dadurch sind die
   einzelnen Linienstücke miteinander verbunden */
  xvor = x;
  Uvor = U;
  x = x + 1.2;
}
```

5.3 Überlagerung von Schwingungen

Fouriersynthese

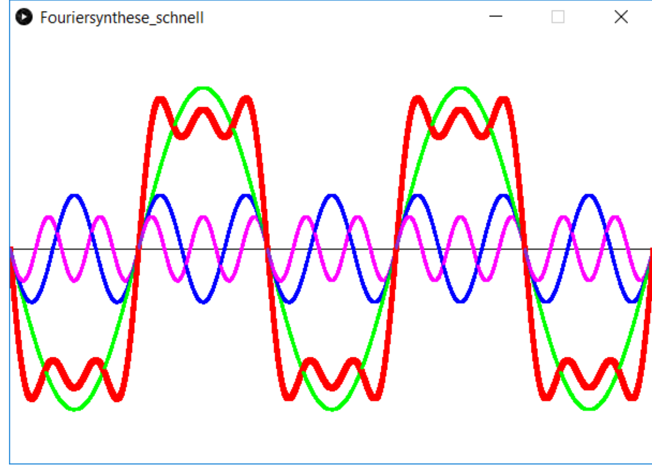


Abbildung 5.6: Addition dreier Sinusschwingungen zu einer Rechteckschwingung (rot)

Wenn man sinus- oder cosinusförmige Schwingungen addiert, bzw. subtrahiert, dann kann man zahlreiche neu Schwingungsformen erzeugen. In der obigen Abbildung 5.6 wurden drei Sinusfunktionen addiert, sodass sich eine nahezu rechteckige Signalform (rot) ergibt. Wenn man noch mehr Sinusfunktionen sinnvoll überlagert, dann erhält man ein immer besseres Rechtecksignal. Die Fouriersche Reihe hierfür lautet:

$$y_{(x)} = \frac{4}{\pi}\left(\frac{sin1x}{1} + \frac{sin3x}{3} + \frac{sin5x}{5} + \cdots\right)$$

In unserem Sketch wollen wir aber nicht nur die Rechteckfunktion zeichnen, sondern auch die Sinusfunktionen, die addiert diese ergeben. Deshalb schreiben wir zuerst

```
y1 = A*sin(w);  y2 = A*sin(3*w)/3;   y3 = A*sin(5*w)/5;
```

und danach

```
y = y1 + y2 + y3;
```

Der Sketch hierzu ist schnell programmiert und deshalb programmieren wir ihn gleich zweimal. Warum? Weil wir etwas lernen wollen. Den ersten Sketch programmieren wir mit einer *for-Schleife* und den zweiten Sketch ohne eine *for-Schleife*. Wenn wir nun von beiden Sketches die Funktionen zeichnen lassen, dann erleben wir, obwohl das Endergebnis bei beiden richtig ist, einen gewaltigen Unterschied. Der Sketch mit der *for-Schleife* ist mit dem Zeichnen sofort fertig und bei dem Sketch ohne *for-Schleife* wartet man eine gefühlte Ewigkeit. Warum ist das so?

Bei der *for-Schleife* werden alle Koordinatenwerte ausgerechnet und sofort gezeichnet. Bei dem Sketch ohne *for-Schleife* wird bei jedem Durchlauf von *void draw()* je ein Punkt jeder Funktion ausgerechnet und dann gezeichnet. Beim nächsten Durchlauf von *void draw()* wird der nächste Punkt ausgerechnet und gezeichnet, usw., usw.. Dies dauert!

Langsam sein ist aber manchmal auch von Vorteil. Bei unserem Sketch *gedämpfte Schwingung* haben wir bewusst auf eine *for-Schleife* verzichtet. So konnten wir sehr schön sehen, wie sich die Funktion entwickelt und man erlangte dabei eine Vorstellung davon, wie der Pendelkörper schwingt.

Sketch 04: Fouriersynthese_schnell

```
// Fouriersynthese schnell

float A = 150;
float y;
float y1;
float y2;
float y3;
float yvor;
float y1vor;
float y2vor;
float y3vor;
float xvor;

void setup()
{
  size(600, 400);
  background(255);
  line(0, height/2, width, height/2);
}

void draw()
{
  translate(0, 200);
  for (float w = 0, x = 0; x <= width; w = w + 0.1*PI/120, x = x + 0.1)
  {
    y1 = A*sin(w);
```

```
    y2 = A*sin(3*w)/3;
    y3 = A*sin(5*w)/5;
    xvor = x;
    y1vor = y1;
    y2vor = y2;
    y3vor = y3;

    strokeWeight(2);
    stroke(0, 255, 0);
    line(xvor, y1vor, x, y1);
    stroke(0, 0, 255);
    line(xvor, y2vor, x, y2);
    stroke(255, 0, 255);
    line(xvor, y3vor, x, y3);

    stroke(255, 0, 0);
    strokeWeight(5);
    y = y1 + y2 + y3;
    xvor = x;
    yvor = y;
    line(xvor, yvor, x, y);
  }
}
```

Sketch 05: Fouriersynthese_langsam

```
// Fouriersynthese langsam

float A = 150;
float y;
float y1;
float y2;
float y3;
float yvor;
float y1vor;
float y2vor;
float y3vor;
float x;
float xvor;
float w;

void setup()
{
  size(600, 400);
  background(255);
  line(0, height/2, width, height/2);
}

void draw()
{
  translate(0, 200);

  w = w + 0.1*PI/120;
  x = x + 0.1;

  y1 = A*sin(w);
  y2 = A*sin(3*w)/3;
  y3 = A*sin(5*w)/5;
  xvor = x;
  y1vor = y1;
```

```
  y2vor = y2;
  y3vor = y3;

  strokeWeight(2);
  stroke(0, 255, 0);
  line(xvor, y1vor, x, y1);
  stroke(0, 0, 255);
  line(xvor, y2vor, x, y2);
  stroke(255, 0, 255);
  line(xvor, y3vor, x, y3);

  stroke(255, 0, 0);
  strokeWeight(5);
  y = y1 + y2 + y3;
  xvor = x;
  yvor = y;
  line(xvor, yvor, x, y);
}
```

Lissajous-Figuren

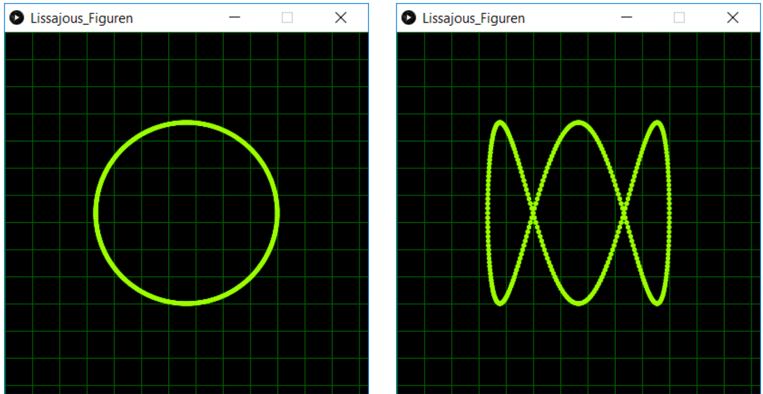


Abbildung 5.7: Zwei unterschiedliche Lissajous-Figuren

Nach der Fouriersynthese eines Rechtecksignals wollen wir uns nun der Erzeugung von Lissajous-Figuren widmen. Sie ergeben sich, wenn man zwei zueinander senkrechte Schwingungen überlagert. Auf der mit dem folgenden Sketch erstellten Abbildung 5.7 sehen wir links einen Kreis. Ein gleiches Bild würden wir auch auf einem Oszilloskop erhalten, wenn wir an die x- und y-Ablenkung je eine Wechselspannung gleicher Frequenz anschließen, von der die eine gegenüber der anderen eine Phasenverschiebung von $\pi/2$ besitzt. Bei der rechten Abbildung ist die Phasenverschiebung ebenfalls $\pi/2$. Das Frequenzverhältnis beträgt jedoch 1 : 3.

Besonders schön anzusehen ist es, wenn man die Lissajous-Figuren in Bewegung setzt. Dies gelingt, wenn wir in unserem Sketch den Winkel bei einem Sinus ein klein wenig verändern. Bei dem unten aufgeführten Sketch dreht sich die rechte Figur der Abbildung 5.7.

Da bei dieser Animation für jede Stellung der Figur sehr viele kleine hellgrüne Kreise gezeichnet werden müssen, muss dies zwingend mit einer *for-Schleife* erfolgen.

Sketch 06: Lissajous_Figuren

```
// Lissajous-Figuren

float A = 100; // Amplitude der Schwingungen
float w; // Winkel in Grad
float z; // Kleinwinkeländerung in Grad

void setup()
{
  size(400, 400);
}

void draw()
{
  background(0);

  // Raster wird gezeichnet.
  stroke(0, 100, 0);
  strokeWeight(1);
  for (int x = 0; x < width; x += 30)
  {
    line(x, 0, x, height);
  }
  for (int y = 0; y < height; y += 30)
  {
    line(0, y, width, y);
  }

  // Die Lissajous-Figur wird gezeichnet
  for (float i = 0; i < 1000; i++) // 1000 Kreise werden pro Durchlauf
                                   // gezeichnet
  {
    fill(155, 255, 0);
    noStroke();
    ellipse((width/2 + A*sin(radians(w+z))), (height/2 +
             A*sin(radians(3*w+90))), 5, 5); w++; // Hier wird
// festgelegt, wie schnell sich die einzelnen Kreise im Fenster bewegen
  }

  z = z + 0.3; // Hiermit wird die Geschwindigkeit der Drehung der Figur
               // bestimmt
}
```

5.4 Processing und Soundkarte

Hörtest

Bisher haben wir Schwingungen nur grafisch dargestellt. Es wäre aber schön, wenn wir die Schwingungen, die wir in unseren Sketches programmieren, auch hören könnten. So könnten wir zum Beispiel mit einem selbstgeschriebenen Sketch *Hoertest* überprüfen, wie gut wir noch hören können. Dazu müsste Processing aber auf die Soundkarte in unserem Computer zugreifen können. Wie können wir dies erreichen?

Zum Glück gibt es freundliche Programmierer wie *Damien Di Fede* und *Anderson Mills,* die hier eine Lösung gefunden haben und uns diese in Form der Bibliothek *Minim* zur Verfügung stellen.

Eine **Bibliothek** (engl. Library) ist eine Sammlung von Klassen. Die Bibliothek *Minim* müssen wir aus dem Internet herunterladen und dann in Processing einfügen. Dabei hilft uns Processing. Wie dies funktioniert, sehen wir in Abbildung 5.8.

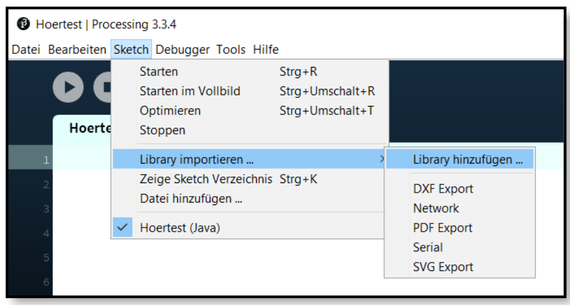


Abbildung 5.8: Hinzufügen einer Bibliothek

Nun öffnet sich das folgende Fenster (Abb. 5.9).

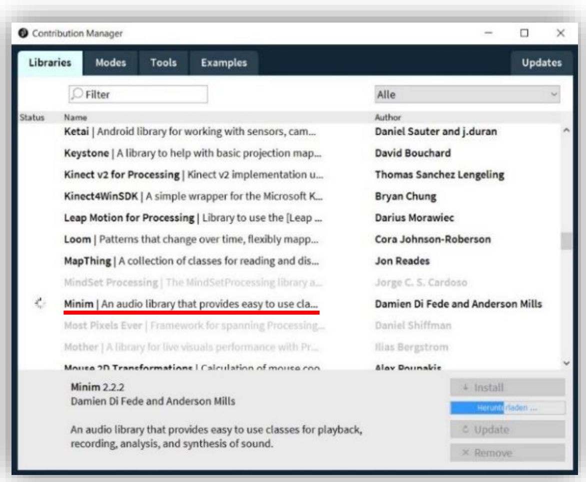


Abbildung 5.9: Übersicht über zahlreiche Bibliotheken

Hier auf *Herunterladen* und dann auf *Installieren* klicken. Danach befindet sich die Library *Minim* im *Ordner Contributed Libraries*. Überprüfen kann man dies, indem man in der Processing PDE auf *Datei* und dann auf *Beispiele* klickt (siehe Abb. 5.10).

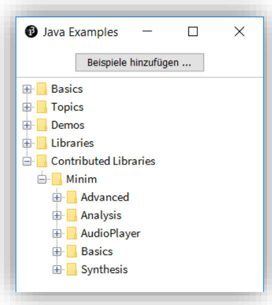


Abbildung 5.10: Die importierte Bibliothek Minim befindet sich im Ordner Ordener Contributed Libraries

Wenn wir nun in unserem Sketch die Bibliothek *Minim* oder einzelne Klassen hieraus verwenden wollen, dann müssen wir *Minim* in unseren Sketch einfügen. Dazu kann man den folgenden Code schreiben *import ddf.minim.** oder man benutzt entsprechend der Abbildung 5.11 die PDE.

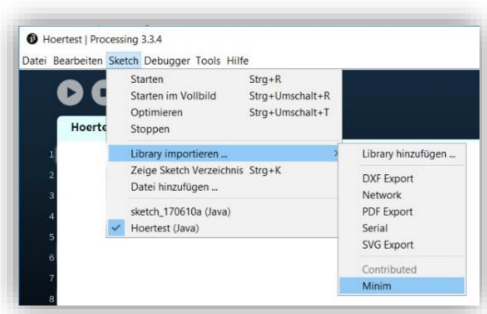


Abbildung 5.11: Einfügen einer Bibliothek in einen Sketch

Dann erhält man allerdings Zuviel des Guten (siehe Abb. 5.12)

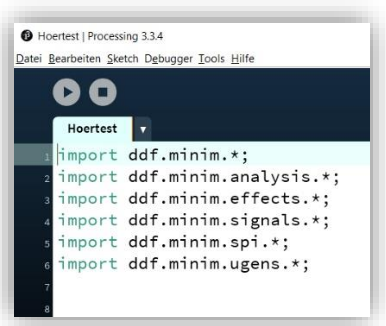


Abbildung 5.12: Ergebnis einer Einfügung

Für unseren Sketch *Hoertest* mit einem einfachen Sinuston brauchen wir jedoch nur *ddf.minim.** und *ddf.minim.signals.**.

Wenn wir wissen wollen, was in dem Ordner *ddf.minim.signals.** alles enthalten ist, dann müssen wir, wie in Abbildung 5.13 dargestellt, einen Blick in den entsprechenden Ordner werfen. Wir öffnen also unseren privaten Processing-Ordner (nicht den Programm-Ordner) und klicken dann auf *libraries→minim →src →ddf →minim →signals.*

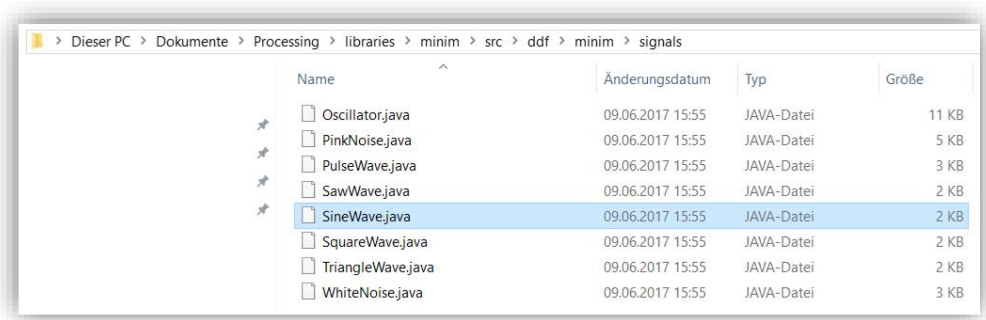


Abbildung 5.13: Inhalt der einzelnen Klassen im Ordner signals

Für unseren Hörtest interessieren wir uns für die Klasse *SineWave.java*. Diese müssen wir in unserem Sketch aufrufen.

Wie geht es aber jetzt weiter? Dazu muss man sich schlau machen. Zum Glück enthält *Minim* einen Ordner mit zahlreichen Dokumentationen (Processing → libraries → minim → documentation) und einen Ordner mit vielen Beispielen (Processing → libraries → minim → examples). In dem folgenden Sketch *Hoertest* werden die einzelnen Programmschritte kommentiert.

Wenn wir den Sketch abspielen, dann hören wir einen Ton mit ansteigender Frequenz. Sobald wir nichts mehr hören, drücken wir eine Maustaste. Dann sehen wir, bis zu welcher Frequenz wir noch hören können. Der Mauszeiger muss sich beim Klick im geöffneten Fenster befinden. Je älter man wird, umso erschreckender ist das im Fenster angezeigte Ergebnis.

Sketch 07: Hoertest

```
// Hörtest

import ddf.minim.*;
import ddf.minim.signals.*;

// Unser Hörtest aus der Bibliothek Minim bekommt den Namen Ohr :)
Minim Ohr;

// Aus dem Ordner ddf.minim.signal.* rufen wir die Klasse Sinewave auf
SineWave Sinus;

/* Damit der Sinus über die Soundkarte ausgegeben werden kann, rufen wir
   aus dem Ordner ddf.minim.* die Klasse AudioOutput auf und geben ihr
   den Namen out */
AudioOutput Out;

// Nun deklarieren wir noch f für die Frequenz
int f;

void setup()
{
  size(400, 200);

  Ohr = new Minim(this);

  // Nun legen wir fest, dass wir den Ton mono abspielen wollen
  Out = Ohr.getLineOut(Minim.MONO);

  /* Nun legen wir die Werte für die Frequenz, die Amplitude und die
     Samplerate fest */
  Sinus = new SineWave(f, 1.0, 44100);

  // Nun übergeben wir unser Sinussignal an die Soundkarte
  Out.addSignal(Sinus);
}

void draw()
{
  background(255);
```

```
/* Wir setzen unser Sinussignal auf den Frequenzwert f, der in
   10er-Schritten ansteigt */
Sinus.setFreq(f);
f = f + 10;

// Die aktuelle Frequenz schreiben wir ins Fenster
fill(0, 0, 255);
textSize(36);
textAlign(CENTER);
text("f in Hz = " +f, 200, 100);

/* Um den Ablauf zu stoppen, muss sich der Mauszeiger beim KLICK im
   Fenster befinden */
if (mousePressed)
{
  noLoop();
  Sinus.setFreq(0);
}
```

Schwebung 01

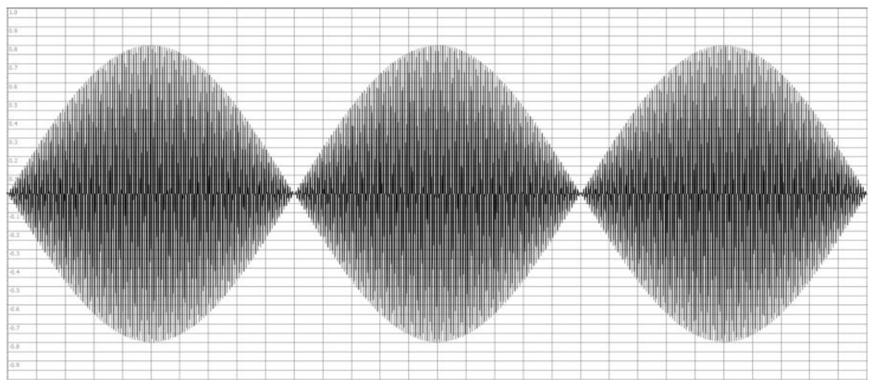


Abbildung 5.14: Veranschaulichung einer Schwebung, die durch die Überlagerung von zwei Schwingungen leicht unterschiedlicher Frequenz entstanden ist. Die Abbildung wurde mit dem Audio-Editor GoldWave erzeugt.

Wenn man die Schwingungen zweier Stimmgabeln leicht unterschiedlicher Frequenz überlagert, dann hört man einen Ton, der in seiner Intensität fortlaufend zu- und abnimmt. Dieses Phänomen nennt man Schwebung. Für die Frequenz dieser Schwebung, also für die Abfolge von Laut und Leise gilt $f_{Schwebung} = f_1 - f_2$. Beispiel: Wenn f_1 = 440 Hz und f_2 = 442 Hz, dann hören wir einen Ton mit der mittleren Frequenz von 441 Hz und einer Schwebungsfrequenz von 2 Hz. Der Ton wir also zweimal in der Sekunde laut.

Wenn man den Sketch *Hoertest* verstanden hat, dann versteht man auch den folgenden Sketch *Schwebung*. Nur eins ist verwunderlich. Zwischen den geschweiften Klammern von *void draw()* steht nichts. Warum lassen wir *void draw()* also nicht einfach weg? Man kann es ja mal probieren, aber dann bleiben die Lautsprecher stumm. Der Grund dafür, dass man *void draw()* nicht einfach weglassen kann ist, dass Processing sonst den Sketch nicht mehr fortlaufend aktualisiert und somit nicht die Funktionen aufruft, die Minim zum Ausgeben von Sound braucht. Minim muss ja dauerhaft Daten an die Soundkarte senden, deswegen muss der Sketch auch durchgehend aktualisiert werden.

Wer viel Freude an akustischen Sachverhalten und an Musik hat, für den lohnt es sich, sich tiefer in die Bibliothek *Minim* einzuarbeiten.

Sketch 08: Schwebung_01

```
// Schwebung

// Aufruf von Bibliothek und Ordner
import ddf.minim.*;
import ddf.minim.signals.*;

// Unsere Schwebung aus der Bibliothek Minim bekommt den Namen Schweb
Minim Schweb;

// Aus dem Ordner ddf.minim.signal.* rufen wir für y1 und y2 die Klasse
// Sinewave auf
SineWave y1, y2;

/* Damit der Sinus über die Soundkarte ausgegeben werden kann,
rufen wir aus dem Ordner ddf.minim.* die Klasse AudioOutput auf und
geben ihr den Namen out */
AudioOutput out;

// Nun deklarieren wir die Frequenzen f1 und f2
float f1 = 400.0;
float f2 = 400.5;

void setup()
{
  Schweb = new Minim(this);

  // Der Ton soll Mono abgespielt werden
  out = Schweb.getLineOut(Minim.MONO);

  /* In die Klammer fügen wir die Werte für die Frequenz,
   die Amplitude und die Samplerate ein */
  y1 = new SineWave(f1, 0.6, 44100);
  y2 = new SineWave(f2, 0.6, 44100);

  // Nun übergeben wir unsere Sinussignale an die Soundkarte
  out.addSignal(y1);
  out.addSignal(y2);
}

// Ohne void draw() wird die Sounddatei nicht abgespielt
void draw()
{
}
```

Schwebung 02

Die Sketche *Hörtest* und *Schwebung_01* haben wir mit der Bibliothek *Minim* erstellt. Eine weitere, recht einfache Möglichkeit, um mit Processing auf die Soundkarte zuzugreifen, bietet die **Bibliothek Sound** der Processing Foundation. Wie im Abschnitt *Hörtest* für *Minim* beschrieben, kann man auch die Bibliothek *Sound* mithilfe von Processing ohne Probleme aus dem Internet herunterladen und installieren. Nach der Installation findet man die Bibliothek *Sound* in der PDE von Processing unter Datei → Beispiele → Libraries → Sound (Abb. 5.15 links). Informationen zu

der Bibliothek *Sound* findet man unter: https://processing.org/reference/libraries/sound/ (Abb. 5.15 rechts zeigt einen Ausschnitt von dieser Seite).

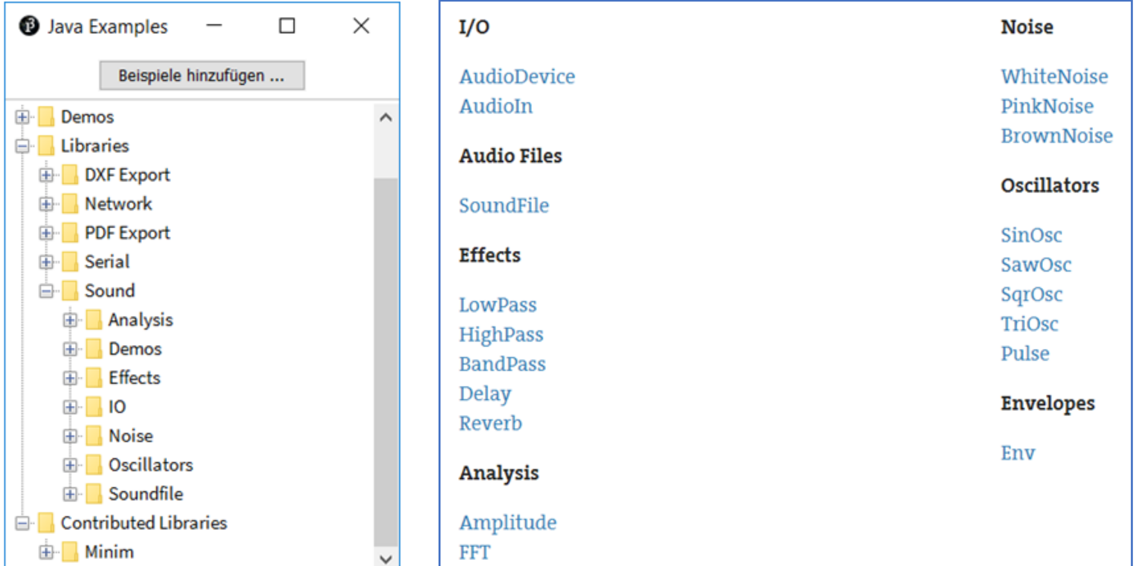


Abbildung 5.15: Hinzugefügte Bibliothek Sound mit ihren Unterordnern (links) und Überblick über ihre Hauptfunktionen (rechts).

Mittels der Bibliothek *Sound* wollen wir, um einen Vergleich zum *Mimin* zu haben, nochmal eine Schwebung erstellen. So kann jeder selbst beurteilen, welche der beiden Bibliotheken für welche Zwecke am besten geeignet ist. Die einzelnen Programmschritte im folgenden Sketch *Schwebung_02* werden im Sketch selber erläutert.

Sketch 09: Schwebung_02

```
// Schwebung 02

// Die Bibliothek Sound wird importiert
import processing.sound.*;

// Zwei Sinusoszillatoren mit den Namen sine1 und sine2 werden
// deklariert
SinOsc sine1;
SinOsc sine2;

// Die beiden Frequenzen für die Schwebung werden initialisiert
float f1 = 440.0;
float f2 = 440.5;

void setup()
{
  // Den beiden Oszillatoren werden ihre Frequenzen und Amplituden
  // zugeordnet

  // Sinusoszillator 1
  sine1 = new SinOsc(this);
  sine1.freq(f1); // Frequenz
  sine1.amp(0.5); // Amplitude
  sine1.play(); // Startet den Sinusoszillator
```

```
  // Sinusoszillator 2
  sine2 = new SinOsc(this);
  sine2.freq(f2); // Frequenz
  sine2.amp(0.5); // Amplitude
  sine2.play(); // Startet den Sinusoszillator

  // Fenster und Text werden erzeugt
  size(300, 200);
  background(255);
  fill(0, 0, 255);
  textSize(24);
  textAlign(CENTER);
  text("Schwebung", 150, 50);
  text("f1 in Hz = " +f1, 150, 100);
  text("f2 in Hz = " +f2, 150, 150);
}

// Ohne void draw() funktioniert der Sketch nicht
void draw()
{
}
```

Rechtecksignal

Bei dem obigen Sketch *Schwebung_02* haben wir zwei Schwingungen addiert. Da dies meines Erachtens recht einfach war, sollte es uns nun auch keine Probleme bereiten, mehr als zwei Schwingungen zu addieren. So können wir zum Beispiel mittels Fouriersynthese ein hörbares Rechtecksignal generieren. Mit Processing gezeichnet haben wir es schon am Anfang von Kapitel 5.3. Hier können wir auch nochmal nachschlagen, um zu sehen, wie die Fourierreihe für ein Rechtecksignal aufgebaut ist. In unserem Sketch *Rechtecksignal* benutzen wir nur vier Sinusschwingungen. Dies genügt aber, um sehr deutlich den Unterschied zwischen einem angenehmen Sinuston und einem nervenden Rechtecksignal zu hören.

Sketch 10: Rechtecksignal

```
// Rechtecksignal

// Die Bibliothek Sound wird importiert
import processing.sound.*;

// Vier Sinusoszillatoren mit den Namen sine1, sine2, sine3 und sine4
// werden deklariert
SinOsc sine1;
SinOsc sine2;
SinOsc sine3;
SinOsc sine4;

// Die vier Frequenzen und die Amplitude A werden initialisiert
float f1 = 1*440;
float f2 = 3*440;
float f3 = 5*440;
float f4 = 7*440;
float A = 0.5;

void setup()
```

```
{
  // Den vier Oszillatoren werden ihre Frequenzen und Amplituden
  // zugeordnet

  // Sinusoszillator 1
  sine1 = new SinOsc(this);
  sine1.freq(f1);
  sine1.amp(A/1);
  sine1.play(); // Startet den Sinusoszillator sine1

  // Sinusoszillator 2
  sine2 = new SinOsc(this);
  sine2.freq(f2);
  sine2.amp(A/3);
  sine2.play(); // Startet den Sinusoszillator sine2

  // Sinusoszillator 3
  sine3 = new SinOsc(this);
  sine3.freq(f3);
  sine3.amp(A/5);
  sine3.play(); // Startet den Sinusoszillator sine3

  // Sinusoszillator 4
  sine4 = new SinOsc(this);
  sine4.freq(f4);
  sine4.amp(A/7);
  sine4.play(); // Startet den Sinusoszillator sine4
}

// Ohne void draw() funktioniert der Sketch nicht
void draw()
{
}
```

5.5 Zusammenfassung

Color Selector Um den *Color Selector* zu öffnen muss man in der PDE bei Karteikarte *Tools* auf *Farbauswahl* klicken. Hier kann man ganz bequem mit dem Mauszeiger die gewünschte Farbe auswählen. Den so erhaltenen Wert kopiert man dann in seinen Sketch.

abs() Die Anweisung *abs()* gibt den Betrag der Zahl oder der Funktion an, die zwischen den runden Klammern steht.

Linien verbinden Will man Linien in Processing miteinander verbinden, dann schreibt man zum Beispiel: *line(xvor, yvor, x, y);*
xvor = x;
yvor = y;
Beim Durchlauf von *void draw()* zeichnet Processing so Linie an Linie.

Bibliotheken Eine Bibliothek (engl. Library) ist eine Sammlung von Klassen, die uns die Programmierarbeit sehr erleichtert. Bibliotheken können wir aus dem Internet herunterladen und in Processing einfügen. Dabei hilft uns Processing. Wie dies funktioniert, sehen wir in Abbildung 5.16 und Abbildung 5.17.

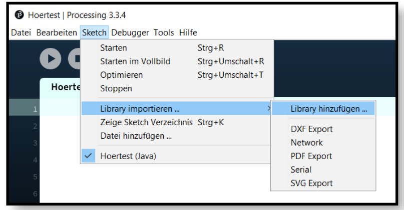


Abbildung 5.16: Importieren einer Bibliothek

Soundausgabe

Mittels der Bibliotheken ***Minim*** oder ***Sound*** können wir über Processing die Soundkarte unseres Computers ansprechen und so Töne, Musik, … ausgeben. Wenn man entsprechend der Abbildung 5.16 auf Library hinzufügen geklickt hat, dann öffnet sich das Fenster von Abbildung 5.17. Hier finden wir u.a. die Bibliothek *Minim* und die Bibliothek *Sound*.

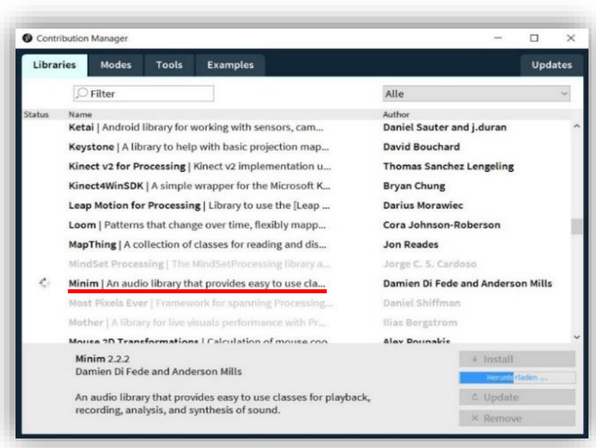
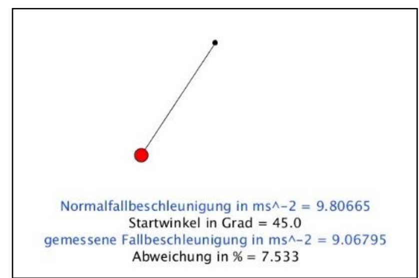


Abbildung 5.17: Ausschnitt aus der Sammlung von Bibliotheken

5.6 Aufgaben

1. Im Physikunterricht wird die Erdbeschleunigung g (Fallbeschleunigung) in der Regel mithilfe eines Fadenpendels und der Gleichung $T = 2\pi\sqrt{\frac{l}{g}}$ bestimmt. Hierbei machen manche Schüler den Fehler, dass sie den Auslenkungswinkel des Fadens zur Senkrechten zu groß wählen. Sie vergessen, dass die obige Gleichung eine Näherungsgleichung ist, die nur für kleine Winkel brauchbare Werte liefert. Um auch für große Winkel gute Werte zu erhalten, muss man die folgende Gleichung verwenden.



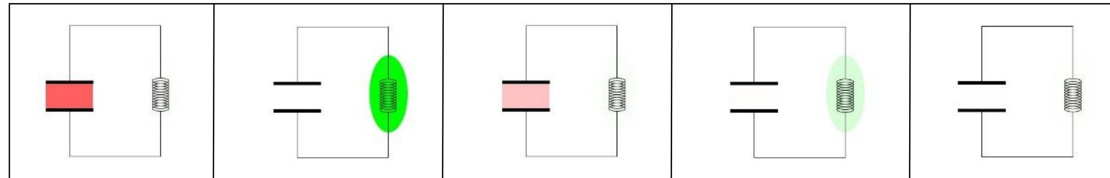
$$T = 2\pi\sqrt{\frac{l}{g}} \cdot [1 + \left(\frac{1}{2}\right)^2 \cdot sin^2\frac{\alpha}{2} + \left(\frac{1\cdot 3}{2\cdot 4}\right)^2 \cdot sin^4\frac{\alpha}{2} + \left(\frac{1\cdot 3\cdot 5}{2\cdot 4\cdot 6}\right)^2 \cdot sin^6\frac{\alpha}{2} + \cdots]$$

Schreibe einen Sketch entsprechend der obigen Abbildung, bei dem im Fenster angezeigt wird, wie groß der prozentuale Fehler ist, wenn man den Auslenkungswinkel zu groß wählt.

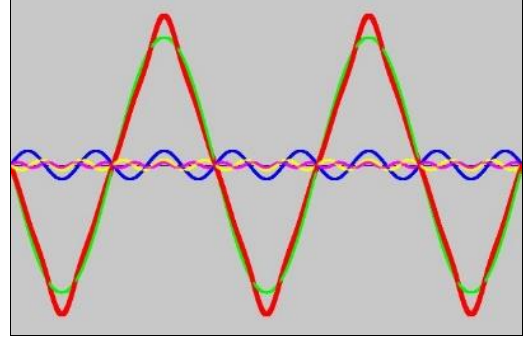
Berechne in deinem Sketch mittels der obigen Gleichung den genauen Wert von T für ein 2 m langes Pendel unter der Annahme, dass am Ort des Experimentes die Normalfallbschleunigung $g = 9{,}80665\ ms^{-2}$ herrscht. Der Winkel α soll hierbei frei wählbar sein. Diesen Wert für die Schwingungsdauer T messen die Schüler auch bei ihrem Experiment zur g-Bestimmung. Setzen sie diesen Wert für T aber nun in die Gleichung $T = 2\pi\sqrt{\frac{l}{g}}$ ein, dann erhalten sie bei großen Winkeln eine deutliche Abweichung vom Wert $g = 9{,}80665\ ms^{-2}$ (siehe Abbildung).

In dem Fenster deines Sketches sollen aber nicht nur die in der Abbildung aufgeführten Werte dargestellt werden, sondern auch ein animiertes Fadenpendel, welches in Abhängigkeit vom Auslenkungswinkel die Schwingungsdauer T besitzt, die mit der großen Gleichung berechnet wurde.

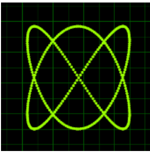
2. Ändere den Sketch *Schwingkreis* so, dass der Schwingkreis eine gedämpfte Schwingung ausführt. D.h., die Farben für das E-Feld im Kondensator und das B-Feld in der Spule sollen sich entsprechend der abnehmenden Amplitudengröße ändern (siehe Abbildung).



3. Im Sketch *Fouriersynthese_schnell* wurde ein Rechtecksignal generiert. Generiere nun mittels Fouriersynthese ein Dreieckssignal.



4. Schreibe einen Sketch der die rechts dargestellte Lissajous-Figur generiert. Das Frequenzverhältnis beträgt 2:3 und die Phasendifferenz π. Die Amplituden der beiden zueinander senkrechten Schwingungen sind gleich groß.



5. Erstelle einen Sketch mit dem man das Lied „Alle meine Entchen“ spielen kann. Durch Klicken mit der Maus auf die jeweilige Klaviertaste (siehe Abbildung unten) soll der entsprechende Ton über die Soundkarte wiedergegeben werden. Hier ist die Tonfolge: C4, D4, E4, F4, G4, G4, A4, A4, A4, A4, G4, A4, A4, A4, A4, G4, F4, F4, F4, F4, E4, E4, G4, G4, G4, G4, C4. Die zugehörigen Frequenzen sind: A4 = 440 Hz, C4 = 262 Hz, D4 = 294 Hz, E4 = 330 Hz, F4 = 349 Hz, G4 = 392 Hz. Benutze zur Erstellung des Sketches die Bibliothek *Minim*.

Tipp: Bevor du mit dem Programmieren beginnst, erzeuge mittels eines Audioeditors, z.B. Audacity, die oben angegebenen Töne als mp3-Dateien. Lege diese Dateien in den Data-Ordner deines Sketches. Informiere dich anschließend in der Dokumentation von *Minim* speziell über das Thema *AudioPlayer*. Für jeden Ton muss man einen eigenen AudioPlayer erzeugen. Mit *play()* spielt man den Ton ab und mit *cue(0)* setzt man den Abspielvorgang wieder auf null zurück. Die Dokumentation von Minim findest du auf deinem Computer in dem Ordner Processing unter: libraries → minim → documentation → index.

Hilfreich können auch die Funktionen *mousePressed()* und *mouseReleased()* sein. (siehe Referenz von Processing).

6 Wellen

Was erwartet uns?

loadPixels(), updatePixels(), Pixel-Array, blendMode(ADD), blendMode(SUBTRACT), HSB-Farbraum, colorMode(),

6.1 Wellenmaschine

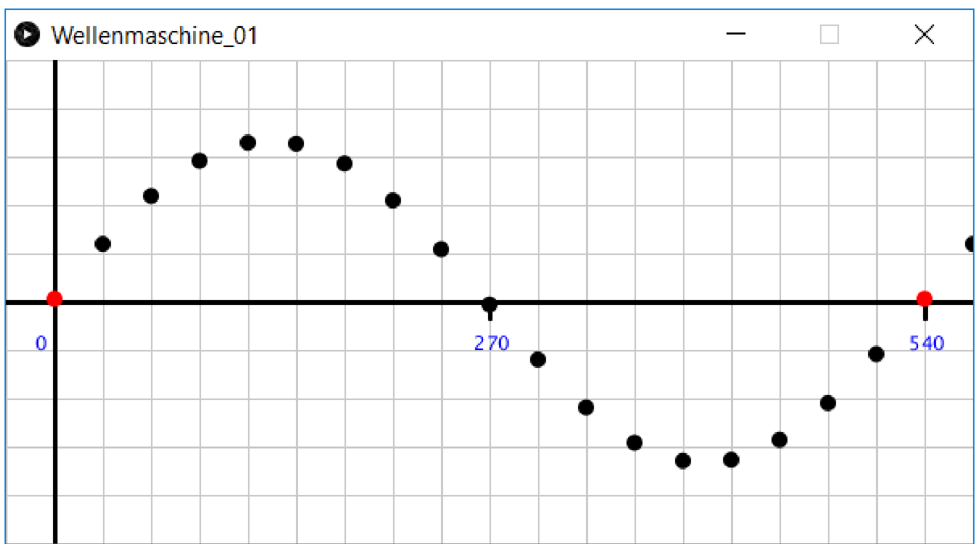


Abbildung 6.1: Wellenmaschine

Bisher haben wir die Schwingungen einzelner Oszillatoren programmiert. Wenn man aber eine Vielzahl von Oszillatoren koppelt und einen hiervon in Schwingungen versetzt, dann breitet sich dieser Schwingungszustand entlang der Kette der gekoppelten Oszillatoren aus. Regt man den linken roten Oszillator in Abbildung 6.1 zu sinusförmigen Schwingungen an, so erhalten wir das Bild einer Sinuswelle. Zeitversetzt nehmen die schwarzen Oszillatoren den Schwingungszustand des roten Oszillators an. Man sagt, sie schwingen phasenverschoben. Sehr schön kann man dies mit einer Wellenmaschine demonstrieren. Da wir aber gerade keine Wellenmaschine zur Hand haben, schreiben wir einen entsprechenden Sketch. Um die Vorteile einer for-Schleife noch einmal zu verdeutlichen, werden wir zwei Varianten unseres Sketches schreiben. Einen Sketch mit und einen ohne for-Schleife.

Im ersten Sketch *Wellenmaschine_01* führen wir alle Oszillatoren per Hand einzeln auf und sehen, dass er gut funktioniert. Erläutert werden muss dieser Sketch nicht, da er sehr einfach gestrickt ist.

Um den zweiten Sketch zu programmieren, muss man jedoch schon einiges von dem wissen, was wir bisher gelernt haben. Dafür sieht er aber auch eleganter aus. Mittels einer *for-Schleife* ersparen wir uns die Arbeit, jeden Oszillator einzeln zu zeichnen und *if, else* sowie der *Modulo-Operator* helfen uns beim Einfärben der Oszillatoren.

Erläutert werden muss jetzt nur noch die folgende Gleichung.

```
ellipse(30+i*30, A*sin(radians(w-i*20)), 10, 10);
```

In der *for-Schleife* *for (int i = 0; i < 20; i++)* läuft i von 0 bis 19. Somit nimmt die x-Koordinate jedes Oszillators (Kreis) um 30 Pixel gegenüber seinem benachbarten Oszillator zu. Gleichzeitig wird von dem Winkel w im Sinus jeweils i*20 Grad abgezogen. Dadurch wird in der Abbildung 6.1 der nach rechts versetzte Oszillator etwas höher gezeichnet. Da der Winkel w bei jedem Durchlauf von *void draw()* um ein Grad erhöht wird, erhalten wir eine animierte Bewegung aller Oszillatoren. Wir beobachten eine Wellenbewegung.
In unserem Sketch *Wellenmaschine_02* wird außer der Funktion *void draw()* noch eine zweite Funktion mit der Bezeichnung *void gitter(int gitterabstand)* aufgeführt. Dies ist nicht zwingend notwendig, doch so lernen wir, dass man sich nicht in jedem Sketch auf die Funktion void draw() beschränken muss, sondern auch in anderen Funktionen gezeichnet werden kann. Weiterhin ist die Funktion *void gitter(int gitterabstand)* nicht leer, sondern deklariert in der Klammer eine neue Variable, die den Gitterabstand festlegt. Den Abstand der Gitternetzlinien legen wir bei *void draw()* beim Aufrufen der Funktion fest.

Sketch 01: Wellenmaschine_01

```
// Wellenmaschine 01

float A = 100; // Amplitude
float w = 0; // Winkel in Grad

void setup()
{
  size(600, 300);
  frameRate(40); // Bilder pro Sekunden
}

void draw()
{
  background(255);

  // Raster
  stroke(200);
  strokeWeight(1);
  for (int x = 0; x < width; x += 30)
  {
    line(x, 0, x, height);
  }
  for (int y = 0; y < height; y += 30)
  {
    line(0, y, width, y);
  }

  // Koordinatensystem
  stroke(0);
  strokeWeight(3);
  line(0, 150, 600, 150);
  line(30, 0, 30, 300);

  //  Skala
  stroke(0);
  line(300, 150, 300, 160);
  line(570, 150, 570, 160);
  fill(0, 0, 255);
  textSize(12);
  text("0", 18, 180);
```

```
text("270", 290, 180);
text("540", 560, 180);

// Verschiebung des Koordinatenursprungs
translate (0, 150);

// Der Winkel wird um ein Grad pro Schritt erhöht
w++;

// Die einzelnen Oszillatoren werden erzeugt
noStroke();
fill(255, 0, 0); // roter Oszillator
ellipse(30, A*sin(radians(w)), 10, 10);

fill(0, 0, 0);  // schwarzer Oszillator
ellipse(60, A*sin(radians(w-20)), 10, 10);

fill(0, 0, 0);  // schwarzer Oszillator
ellipse(90, A*sin(radians(w-40)), 10, 10);

fill(0, 0, 0);  // schwarzer Oszillator
ellipse(120, A*sin(radians(w-60)), 10, 10);

fill(0, 0, 0);  // schwarzer Oszillator
ellipse(150, A*sin(radians(w-80)), 10, 10);

fill(0, 0, 0);  // schwarzer Oszillator
ellipse(180, A*sin(radians(w-100)), 10, 10);

fill(0, 0, 0);  // schwarzer Oszillator
ellipse(210, A*sin(radians(w-120)), 10, 10);

fill(0, 0, 0);  // schwarzer Oszillator
ellipse(240, A*sin(radians(w-140)), 10, 10);

fill(0, 0, 0);  // schwarzer Oszillator
ellipse(270, A*sin(radians(w-160)), 10, 10);

fill(0, 0, 0);  // schwarzer Oszillator
ellipse(300, A*sin(radians(w-180)), 10, 10);

fill(0, 0, 0);  // schwarzer Oszillator
ellipse(330, A*sin(radians(w-200)), 10, 10);

fill(0, 0, 0);  // schwarzer Oszillator
ellipse(360, A*sin(radians(w-220)), 10, 10);

fill(0, 0, 0);  // schwarzer Oszillator
ellipse(390, A*sin(radians(w-240)), 10, 10);

fill(0, 0, 0);  // schwarzer Oszillator
ellipse(420, A*sin(radians(w-260)), 10, 10);

fill(0, 0, 0);  // schwarzer Oszillator
ellipse(450, A*sin(radians(w-280)), 10, 10);

fill(0, 0, 0);  // schwarzer Oszillator
ellipse(480, A*sin(radians(w-300)), 10, 10);

fill(0, 0, 0);  // schwarzer Oszillator
ellipse(510, A*sin(radians(w-320)), 10, 10);
```

```
  fill(0, 0, 0);  // schwarzer Oszillator
  ellipse(540, A*sin(radians(w-340)), 10, 10);

  fill(255, 0, 0);  // roter Oszillator
  ellipse(570, A*sin(radians(w-360)), 10, 10);

  fill(0, 0, 0);  // schwarzer Oszillator
  ellipse(600, A*sin(radians(w-380)), 10, 10);
}
```

Wenn wir nun den folgenden Sketch mit dem vorhergehenden vergleichen, dann sehen wir, dass dieser wesentlich eleganter programmiert ist.

Sketch 02: Wellenmaschine_02

```
// Wellenmaschine 02

float A = 100; // Amplitude
float w = 0; // Winkel in Grad

void setup() {
  size(600, 300);
  frameRate(40); // Bilder pro Sekunden
}

void draw() {
  background(255);

  /* Raster zeichnen
   Der Wert zwischen den runden Klammern gibt den
   Abstand der Gitterlinien an */
  gitter(30);

  // Koordinatenursprung verschieben
  translate(0, 150);

  // Der Winkel wird um ein Grad pro Durchlauf erhöht
  w++;

  // Die einzelnen Oszillatoren werden erzeugt
  noStroke();

  for (int i = 0; i < 20; i++)
  {
    if (i*20%360 == 0) // Modulo-Operator gibt nur den Rest der Division
                       // zurück
      fill(255, 0, 0); // rote Oszillatoren
    else
      fill(0, 0, 0); // schwarze Oszillatoren

    /* Der Kreismittelpunkt wird jeweils um i*30 nach rechts verschoben
     und der Winkel wird jeweils um i*20 verringert */
    ellipse(30+i*30, A*sin(radians(w-i*20)), 10, 10);
  }
}
```

```
void gitter(int gitterabstand)
{
  // Gitternetz
  stroke(200);
  strokeWeight(1);
  for (int x = 0; x < width; x += gitterabstand)
    line(x, 0, x, height);

  for (int y = 0; y < height; y += gitterabstand)
    line(0, y, width, y);

  // Koordinatensystem
  stroke(0);
  strokeWeight(3);
  line(0, 150, 600, 150);
  line(30, 0, 30, 300);

  // Skala
  stroke(0);
  line(300, 150, 300, 160);
  line(570, 150, 570, 160);
  fill(0, 0, 255);
  textSize(12);
  text("0", 18, 180);
  text("270", 290, 180);
  text("540", 560, 180);
}
```

6.2 Stehende Welle

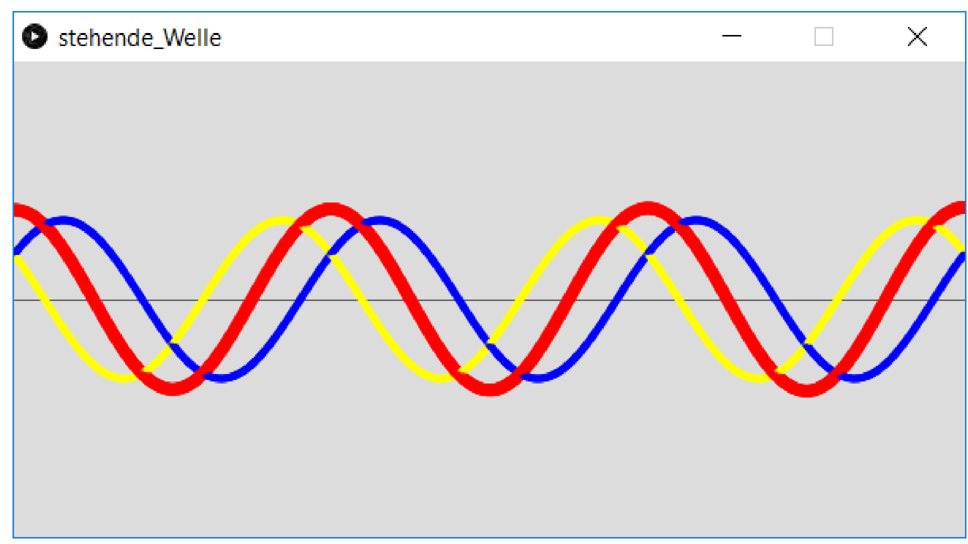


Abbildung 6.2: Die gelbe Welle läuft von links nach recht und die blaue Welle läuft von rechts nach links. Die resultierende rote Welle stellt die stehende Welle dar.

Anstatt jeden Oszillator mit einer eigenen Gleichung zu beschreiben, kann mithilfe der unten aufgeführten Wellengleichungen der Schwingungszustand jedes einzelnen Oszillators der Wellenmaschine zu jeder Zeit t und an jedem Ort x bestimmt werden.

$y(x,t) = y_{max} \cdot sin 2\pi(\frac{t}{T} - \frac{x}{\lambda})$ Welle läuft von links nach rechts

$y(x,t) = y_{max} \cdot sin2\pi(\frac{t}{T} + \frac{x}{\lambda})$ Welle läuft von rechts nach links

Diese Gleichungen gelten nicht nur für mechanische Wellen, sondern auch für elektromagnetische Wellen, wenn man zum Beispiel y durch E für die elektrische Feldstärke ersetzt.

In unserem neuen Sketch wollen wir entsprechend der beiden Wellengleichungen eine gelbe Welle von links nach rechts durch das Fenster laufen lassen und eine blaue Welle von rechts nach links. Wenn wir beide Wellen addieren, beobachten wir das interessante Phänomen der stehenden Welle (In der Abbildung 6.2 rot dargestellt). Warum heißt dieses Phänomen eigentlich stehende Welle? Abgesehen von den Oszillatoren an den Knotenpunkten schwingen doch alle Oszillatoren munter auf und ab. Bei unserer Simulation beobachten wir ortsfeste Knoten und ortsfeste Schwingungsbäuche. Da die rote Welle weder nach rechts noch nach links läuft, spricht man von einer stehenden Welle.

Der Sketch zur Simulation einer stehenden Welle ist leicht zu programmieren. Man muss jedoch darauf achten, dass der Zuwachs der Zeit t sehr klein gehalten wird, sonst kann man den Schwingungsvorgang nicht gut beobachten.

Bei periodischen Bewegungen in Processing muss man auch auf die Bildwiederholungsrate achten. Denn sie bestimmt zusammen mit dem Sketch, wie wir die Bewegungsänderung auf dem Bildschirm sehen. Diesen Effekt bemerkt man auch in manchen Filmen. Die Postkutsche fährt vorwärts, obwohl die Räder sich rückwärts drehen. Dieser Effekt entsteht, wenn die Filmkamera das nächste Bild aufzeichnet, wenn das Rad sich z.B. nur um 355° gedreht hat.

Sketch 03: stehende_Welle

```
// stehende Welle

float y1; // Welle 1
float y2; // Welle 2
float y3; // Summe Well 1 + Welle 2
float y1max = 50; // Amplitude der Welle 1
float y2max = 50; // Amplitude der Welle 2
float t; // Zeit
float T = 30; // Periodendauer
float Lambda = 200; // Wellenlänge

void setup()
{
  size(600, 300);
}

void draw()
{
  background(220);

  translate(0, 150);

  stroke(100);
  strokeWeight(1);
  line(0, 0, width, 0);

  for (float x = 0; x < 600; x++)
  {
```

```
    y1 = y1max*sin(2*PI*((t/T)-(x/Lambda))); // Welle 1 läuft von links
                                              // nach rechts
    y2 = y2max*sin(2*PI*((t/T)+(x/Lambda))); // Welle 2 läuft von rechts
                                              // nach links
    y3 = y1 + y2; // stehende Welle

    noStroke();
    fill(255, 255, 0);
    ellipse(x, y1, 5, 5); // gelbe Welle 1
    fill(0, 0, 255);
    ellipse(x, y2, 5, 5); // blaue Welle 2
    fill(255, 0, 0);
    ellipse(x, y3, 8, 8); // stehende Welle (rot)
  }

  t = t + 0.1;
}
```

6.3 Stehende Welle 3D animiert

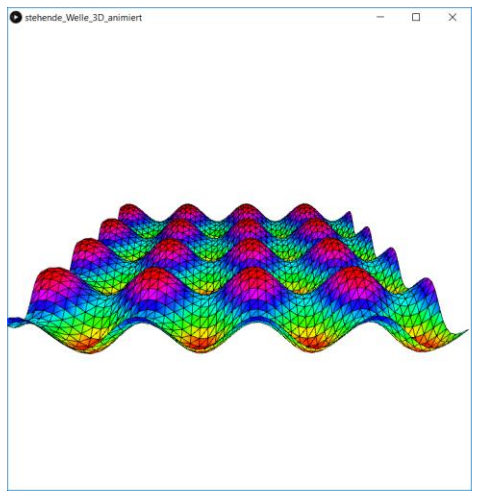


Abbildung 6.3: Animierte stehende Welle in 3D

Erinnern wir uns an den Sketch *Potenzialgebirge_animiert* aus dem Kapitel 3. Wenn wir in diesem Sketch die Gleichung für das Potenzial durch die folgende Gleichung ersetzen

```
stehendeWelle[x][y] = sin(x * 0.5 + millis() / 1000.0) + cos(x * 0.5 -
millis() / 1000.0) + sin(y * 0.5 + millis() / 1000.0) + cos(y * 0.5 -
millis() / 1000.0);
```

erhalten wir die dreidimensionale Animation einer stehenden Welle (Abb. 6.3). Der gelb unterlegte Teil der Gleichung liefert uns eine stehende Welle entsprechend der linken Grafik in Abbildung 6.4. Der grün unterlegte Teil liefert uns eine stehende Welle entsprechend der rechten Grafik in Abbildung 6.4. Zusammen ergeben sie die obige Abbildung 6.3.

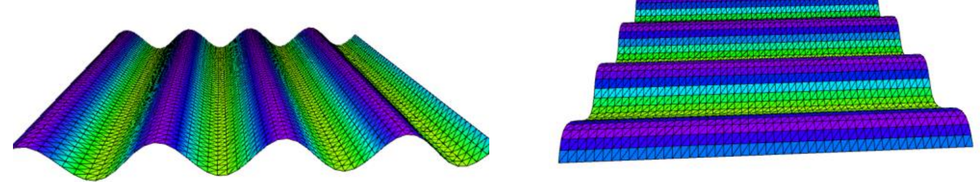


Abbildung 6.4: Stehende Welle in x-Richtung (links) und stehende Welle in z-Richtung (rechts)

Die restlichen Programmschritte müssen hier nicht mehr beschrieben werden, da sie sehr ausführlich im Text zum Sketch *Potenzialgebirge_animiert* (Kapitel 3) erläutert wurden. Die Sketche *Potenzialgebirge_animiert* und *stehende_Welle_3D_animiert* unterscheiden sich also nur in den beiden Gleichungen bezüglich des jeweiligen physikalischen Sachverhaltes.

Sketch 04: stehende_Welle_3D_animiert

```
// stehende Welle 3D animiert

float skalierung = 10; // Gibt die Größe einer Gitterzelle an
float rotationX = PI/4; // Rotation um die x-Achse (Startwert entspricht
                        // 45°)
float rotationZ = 0; // Rotation um die z-Achse
float positionZ = 0; // Verschiebung entlang der z-Achse
float[][] stehendeWelle = new float[50][50]; // zweidimensionales Array
                                  // zur Speicherung der Potenzialwerte

void setup()
{
  size(750, 750, P3D);
}

/* Die folgende Funktion wird aufgerufen, wenn die Maustaste gedrückt
ist und die Maus bewegt wird */
void mouseDragged()
{
  /* Die Differenz der Mauskoordinaten (aktuelle Position und vorherige
   Position) wird verkleinert und zur Rotation entlang der x- bzw.
   z-Achse addiert */
  rotationX += (mouseY - pmouseY) * -0.01;
  rotationZ += (mouseX - pmouseX) * -0.01;
}

// Die folgende Funktion ändert die Werte durch das Drehen des Mausrades
void mouseWheel(MouseEvent event)
{
  positionZ += event.getCount() * 100;
}

void draw()
{
  // Das zweidimensionale Array wird mit Werten gefüllt
  for (int y = 0; y < 50; y++)
  {
    for (int x = 0; x < 50; x++)
    {
```

```
      stehendeWelle[x][y] = sin(x * 0.5 + millis() / 1000.0) + cos(x *
      0.5 - millis() / 1000.0) + sin(y * 0.5 + millis() / 1000.0) +
      cos(y * 0.5 - millis() / 1000.0);
    }
  }

  colorMode(RGB, 255, 255, 255); // Farbraum RGB für den Hintergrund
  background(255, 255, 255);
  stroke(0, 0, 0);
  strokeWeight(1);

  translate(width / 2, height / 2, 0); // Die Null dient hier als
      // Hinweis, dass die z-Koordinate nicht verändert wird
  rotateX(rotationX);
  rotateZ(rotationZ);
  translate(0, 0, positionZ); // Das Koordinatensystem kann mit dem
                  // Mausrad in z-Richtung verschoben werden

  colorMode(HSB, 360, 100, 100); // Farbraum HSB für die stehende Welle

  // Zeichnet die stehende Welle als eine mit Dreiecken dargestellte
  // Oberfläche
  for (int y = 0; y < 50 - 1; y++)
  {
    beginShape(TRIANGLE_STRIP); // Siehe Abbildung in der Referenz zur
                                // Funktion beginShape()
    for (int x = 0; x < 50; x++)
    {
      // Die Farben werden entsprechend den Elongationen der stehenden
      // Welle angepasst (HSB)
      fill(stehendeWelle[x][y] * 50 + 180, 100, 100);
      vertex((x - 25) * skalierung, (y - 25) * skalierung,
            stehendeWelle[x][y] * skalierung);
      vertex((x - 25) * skalierung, (y - 25 + 1) * skalierung,
            stehendeWelle[x][y + 1] * skalierung);
    }
    endShape();
  }
}
```

6.4 Hertzsches Gitter

Abbildung 6.5: Wellen vor und hinter dem Hertzschen Gitter

Die mit dem folgenden Sketch erstellte Abbildung 6.5 zeigt die Wellenverhältnisse vor und hinter einem Hertzschen Gitter. Bei einem Hertzschen Gitter trifft eine elektromagnetische Welle auf eine regelmäßige Anordnung von Metallstäben, die parallel zum elektrischen Feldstärkevektor der Welle angeordnet sind. Die in der Abbildung gelb dargestellte Welle geht zwischen den Gitterstäben hindurch, regt aber auch die Elektronen in den Metallstäben zu Schwingungen mit der Frequenz der einfallenden Welle an. Die so beschleunigten Elektronen senden nun ihrerseits eine elektromagnetische Welle aus (In der Abbildung 6.5 grün und violett dargestellt). Da die elektrische Feldstärke auf Metalloberflächen null ist, besitzen die von den Stäben ausgehenden Wellen einen Phasensprung von π, bzw. 180° gegenüber der einfallenden Welle. Die entsprechenden Wellengleichungen lauten:

$$E(x,t) = E_{max} \cdot sin2\pi(\frac{t}{T} - \frac{x}{\lambda})$$ gelbe Welle

$$E(x,t) = -E_{max} \cdot sin2\pi(\frac{t}{T} - \frac{x}{\lambda})$$ violette Welle

$$E(x,t) = -E_{max} \cdot sin2\pi(\frac{t}{T} + \frac{x}{\lambda})$$ grüne Welle

Aufbauend auf dem Sketch *stehende_Welle*, ist die Programmierung des Sketches *Hertzsches_Gitter* nun ein Kinderspiel. Wenn wir die Simulation ablaufen lassen, dann sehen wir, dass sich die Wellen hinter dem Hertzschen Gitter auslöschen und vor dem Hertzschen Gitter eine stehende Welle entsteht. Im Experiment mit Mikrowellen kann man so über den doppelten Knotenabstand vor dem Gitter recht einfach die Wellenlänge messen.

Sketch 05: Hertzsches_Gitter

```
// Hertzsches Gitter

float E1; // Welle 1
float E2; // Welle 2
float E3; // Welle 3
float E; // Summe von zwei Wellen
float E1max = 50; // Amplitude der Welle 1
float E2max = 50; // Amplitude der Welle 2
float E3max = 50; // Amplitude der Welle 3
float t; // Zeit
float T = 30; // Periodendauer
float Lambda = 150; // Wellenlänge

void setup()
{
  size(600, 350);
}

void draw()
{
  background(220);

  translate(0, 200);

  // x-Achse wird gezeichnet
  stroke(100);
  strokeWeight(1);
  line(0, 0, width, 0);

  for (float x = 0; x < 600; x++)
  {
    E1 = E1max*sin(2*PI*((t/T)-(x/Lambda))); // gelbe Welle 1 läuft von
                                             // links nach rechts
    E2 = -E2max*sin(2*PI*((t/T)+(x/Lambda))); // grüne Welle 2 läuft von
                                              // rechts nach links
    E3 = -E3max*sin(2*PI*((t/T)-(x/Lambda))); // violette Welle 3 läuft
                                          // von links nach rechts

    // Die einfallende Welle und die vom Gitter ausgehenden Wellen
    // werden gezeichnet
    noStroke();
    fill(255, 255, 0);
    ellipse(x, E1, 5, 5); // Welle 1 (gelb)
    fill(0, 220, 0);
    ellipse(x-300, E2, 5, 5); // Welle 2 (grün)
    fill(255, 0, 255);
    ellipse(x+300, E3, 5, 5); // Welle 3 (violett)

    // Die beiden resultierenden Wellen werden gezeichnet
    if (x < 300)
    {
      E = E1 + E2;

      fill(255, 0, 0);
      ellipse(x, E, 8, 8); // resultierende Welle (rot)
    }

    if (x > 300)
    {
```

```
      E = E1 + E3;

      fill(255, 0, 0);
      ellipse(x, E, 8, 8); // resultierende Welle (rot)
    }
  }

  // Hertzsches Gitter
  stroke(0, 0, 255);
  strokeWeight(10);
  line(300, 100, 300, -100);

  // Beschriftung
  fill(0, 0, 255);
  textSize(36);
  textAlign(CENTER);
  text("Hertzsches Gitter", 300, -150);

  t = t + 0.1;
}
```

6.5 Doppelspalt

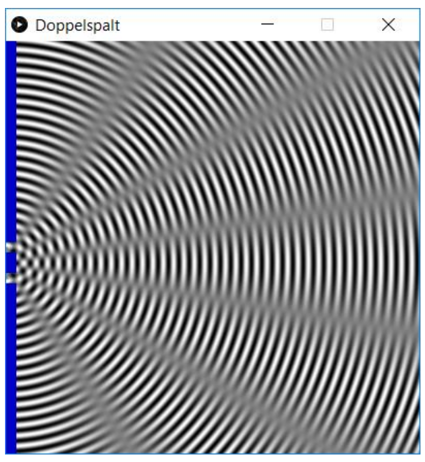


Abbildung 6.6: Interfrenzmuster hinter einem Doppelspalt

Nachdem uns der vorhergehende Sketch vor keine geistige Herausforderung gestellt hat, wollen wir nun unser Gehirn wieder etwas auf Trab bringen. Dazu fertigen wir den Sketch *Doppelspalt* an, der uns die Abbildung 6.6 für unterschiedliche Wellenlängen zeichnen kann. Damit dies gelingt, müssen wir uns zuerst mit den beiden neuen Funktionen **loadPixels()** und **updatePixels()** vertraut machen. Mit *loadPixels()* laden wir die Information über jedes einzelne Pixel in unserem Fenster in ein eindimensionales Array. Da wir bei *void setup()* allen Pixeln die Farbe Weiß gegeben haben, besitzen sie an allen Stellen im Array die Information „Ich bin weiß.“. Die Farbe der Pixel wollen wir nun so ändern, dass sich das Interferenzmuster von Abbildung 6.6 ergibt. Wenn wir dies geschafft haben, rufen wir *updatePixels()* auf, um diese Änderungen abzuspeichern.

Bevor wir nun den Bereich zwischen *loadPixels()* und *updatePixels()* mit Code füllen, müssen wir uns zuerst die physikalischen Gegebenheiten ins Gedächtnis rufen. Wenn eine Welle mit parallelen Wellenfronten auf den Doppelspalt trifft, dann gehen von jedem Einzelspalt Kreiswellen

aus, die sich zu dem in Abbildung 6.6 dargestellten Interferenzmuster überlagern. Für jedes Pixel im Fenster muss also bestimmt werden, ob es die Farbe Weiß (Wellenberg), Grau (Auslöschung) oder Schwarz (Wellental) erhalten soll. Dazu berechnen wir die Elongationen der Wellen von Spalt 1 und Spalt 2 zuerst getrennt, um sie dann anschließend zu addieren. Schauen wir uns zuerst die Welle 1 vom oberen Spalt an (siehe Abbildung 6.7). Den Abstand eines Pixels von der Spaltöffnung bezeichnen wir mit r1. Die Größe von r1 können wir mit dem Satz von Pythagoras $c^2 = a^2 + b^2$ berechnen. Dazu schreiben wir in unserem Sketch: float r1 = sqrt(x*x+(y-y1)*(y-y1)). Siehe hierzu Abbildung 6.7. Da der obere Spalt in der Mitte den Wert y1 = 200 besitzt und das Fenster 400 Pixel hoch ist, schwankt der Wert in der Klammer (y – y1) zwischen -200 und +200. Nun berechnen wir mit r1/L (L = Wellenlänge) wie oft die Wellenlänge auf r1 abgetragen werden kann. Wenn wir diesen Wert mit dem Vollwinkel 2π multiplizieren, erhalten wir den Winkel w1, den wir für die Berechnung der Elongation e1 am Orte r1 benötigen. Oder anders erklärt: Immer wenn die Welle um λ (L) fortgeschritten ist, dann hat sich der Zeiger im Zeigerdiagramm jeweils um den Winkel w = 2π , bzw. um ein ganzzahliges Vielfaches von 2π gedreht. Somit gilt: $w = \frac{r1}{L} \cdot 2 \cdot \pi$. w1 ist also nicht der Winkel zwischen x und r1 (siehe Abb. 6.7), sondern der Winkel, den wir in die folgende Sinusfunktion einsetzen: float e1 = 0.5 * sin(w1). So können wir innerhalb der doppelten *for-Schleife* für jeden Punkt im Fenster die Elongation e1 ausrechnen. Für die Elongation e2 gilt Entsprechendes. Der Faktor 0.5 vor dem Sinus sorgt dafür, dass bei der Addition e = e1 + e2 der Wert nicht größer als 1 wird.

Nun können wir dem Array pixels[y * width + x] die Elongation e so zuordnen, dass daraus Farbwerte von schwarz bis weiß werden. Dies gelingt mit color(128 + e * 128). e ändert sich im Bereich von -1 bis +1. Dadurch ändert sich der Wert in der Klammer von 0 (schwarz) bis 256 (weiß). Nun müssen wir nur noch mit *updatePixels()* diese Farbwerte den einzelnen Pixeln zuordnen.

Eine Anmerkung noch zu dem Array pixels[y * width + x]. In der doppelten *for-Schleife* wird die innere *for-Schleife* immer zuerst vollständig durchlaufen. Danach rückt die äußere *for-Schleife* einen Schritt vor. Dann wird die innere Schleife wieder ganz durchlaufen. Danach rückt die äußere *for-Schleife* wieder einen Schritt vor. Und so weiter. D.h., x läuft von Null bis zum Fensterrand. Danach erhöht sich der y-Wert um 1. Nun läuft x wieder von Null bis zum Fensterrand. Und so weiter.

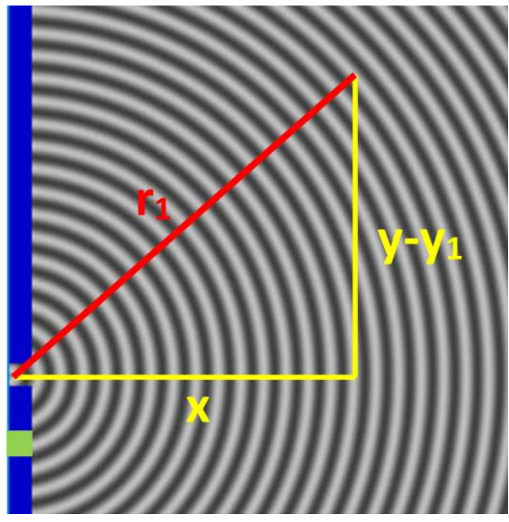


Abbildung 6.7: Skizze zur Berechnung des Abstandes von einem Pixel zu einer Spaltöffnung.

Sketch 06: Doppelspalt

```
// Doppelspalt

float y1 = 200; // Mittelpunkt der vom oberen Spalt ausgehenden
Kreiswelle
float y2 = 230; // Mittelpunkt der vom unteren Spalt ausgehenden
Kreiswelle
float L = 10; // Wellenlänge

void setup()
{
  size (400, 400);
  background(255);
}

void draw()
{
  loadPixels(); // Lädt die Pixel der ganzen Fläche als ein
                // eindimensionales Array.

  // Mittels der beiden for-Schleifen wird die Farbe der einzelnen Pixel
  // verändert.
  for (int y = 0; y < height; y++) //Spalten durchlaufen
  {
    for (int x = 0; x < width; x++) //Zeilen durchlaufen
    {
      // Welle 1, Berechnung von r1 mittels Pythagoras
      float r1 = sqrt(x*x+(y-y1)*(y-y1)); // Der Wert einer y-Klammer
                                          // schwankt zwischen +200 und -200
      float w1 = 2 * PI * r1/L ; // Der Winkel w1 wird in Abhängigkeit
                                 // von r berechnet.
      float e1 = 0.5 * sin(w1); // e1 = Elongation 1

      // Welle 2, Berechnung von r2 mittels Pythagoras
      float r2 = sqrt(x*x+(y-y2)*(y-y2)); // Der Wert einer y-Klammer
                                          // schwankt zwischen +170 und -230
      float w2 = 2 * PI * r2/L ; // Der Winkel w2 wird in Abhängigkeit
                                 // von r berechnet.
      float e2 = 0.5 * sin(w2); // e2 = Elongation 2

      float e = e1 + e2; // Elongation am Überlagerungsort

      /* Das Pixel-Array pixels[] wird mit Farbwerten zwischen 0
        (schwarz)und 256 (weiß) gefüllt, da die Elongation e sich
         zwischen +1 und -1 ändert. */
      pixels[y * width + x] = color(128 + e * 128);
    }
  }
  updatePixels(); //Geänderte Pixel werden gezeichnet

  // Der Doppelspalt wird gezeichnet
  noStroke();
  fill(0, 0, 200);
  rect(0, 0, 10, 195); // oben
  rect(0, 205, 10, 20); // mittig
  rect(0, 235, 10, 165);  // unten
}
```

6.6 Farbmischungen

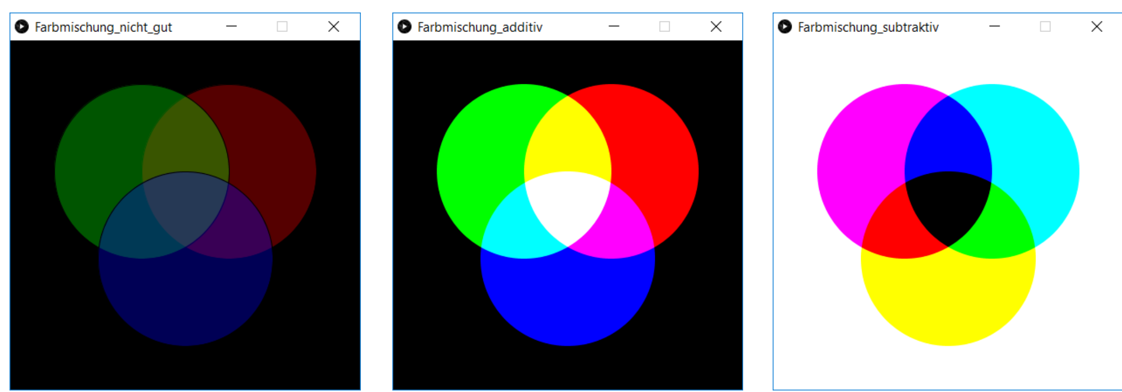


Abbildung 6.8: Links: keine gute additive Farbmischung,
Mitte: gute additive Farbmischung, Rechts: gute subtraktive Farbmischung

Nachdem wir uns beim Sketch *Doppelspalt* so angestrengt haben, kommt nun wieder etwas Erholsames. Elektromagnetische Wellen in einem Wellenlängenbereich von 380 nm bis 780 nm können vom menschlichen Auge wahrgenommen werden und im Gehirn zu Farbeindrücken verarbeitet werden. Die drei lichtempfindlichen Zapfen im Auge nehmen dabei unterschiedliche Wellenlängenbereiche wahr, welche den Farben Rot, Grün und Blau entsprechen. Aus diesem Grund strahlen auch die Pixel eines Monitors in diesen drei Farben. Andere Farben entstehen durch die additive Mischung dieser drei Grundfarben. So entsteht zum Beispiel die Farbe Weiß, wenn man in Processing schreibt *fill(255, 255, 255)*.

Wir wollen nun einen Sketch schreiben, der zeigt, welche Farben sich ergeben, wenn man diese drei Grundfarben mischt. Dazu überlagern wir drei mit den Grundfarben gefüllte Kreise derart, dass sich ein aus vielen Physikbüchern bekanntes Bild ergibt (siehe Abb. 6.8 Mitte). Mit dem nun aufgeführten Sketch erreichen wir unser Ziel jedoch nicht, da die erste Farbe im Überschneidungsbereich von der zweiten verdeckt wird und beide von der dritten Farbe (siehe Abb. 6.9). Die Farben sind halt nicht transparent.

```
size(400, 400);
background(0);

fill(255, 0, 0); // rot
ellipse(250, 150, 200, 200);
fill(0, 255, 0); // grün
ellipse(150, 150, 200, 200);
fill(0, 0, 255); // blau
ellipse(200, 250, 200, 200);
```

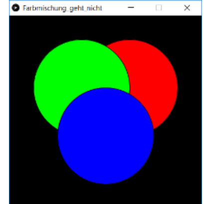


Abbildung 6.9: Keine Farbmischung

Ein anderer Lösungsansatz könnte darin bestehen, bei *fill* eine vierte Zahl, den Alphawert, hinzuzufügen. Mit dem Alphawert kann man die Transparenz bzw. die Deckfähigkeit der gewählten Farbe festlegen. 255 entspricht 100% Deckfähigkeit und 0 entspricht 0% Deckfähigkeit. Aber egal, welchen Alphawert wir auch wählen, ein überzeugendes Bild ergibt sich nicht. Das beste Ergebnis sehen wir in Abbildung 6.8 ganz links. Was tun?

Werfen wir in der Referenz von Processing einen Blick auf die Anweisung **blendMode()**. Hier sehen wir, dass Processing eine Vielzahl von Mischmöglichkeiten anbietet.

> *BLEND - linear interpolation of colours: C = A*factor + B. This is the default blending mode.*
> ***ADD - additive blending** with white clip: C = min(A*factor + B, 255)*

*SUBTRACT - subtractive blending with black clip: C = max(B - A*factor, 0)*
*DARKEST - only the darkest colour succeeds: C = min(A*factor, B)*
*LIGHTEST - only the lightest colour succeeds: C = max(A*factor, B)*
DIFFERENCE - subtract colors from underlying image.
EXCLUSION - similar to DIFFERENCE, but less extreme.
MULTIPLY - multiply the colors, result will always be darker.
SCREEN - opposite multiply, uses inverse values of the colors.
REPLACE - the pixels entirely replace the others and don't utilize alpha (transparency) values

Für unsere additive Farbmischung benötigen wir also *blendMode(ADD)*. Damit erhalten wir das gewünschte Ergebnis (Abb. 6.8 Mitte). Und wenn wir schon einmal dabei sind, dann können wir die Farben auch noch subtraktiv mischen (Abb. 6.8 rechts). Dies gelingt mit *blendMode(SUBTRACT).* Hier sind nun die beiden dazu passenden Sketche.

Sketch 07: Farbmischung_additiv

```
// additive Farbmischung

size(400, 400);
background(0);

blendMode(ADD); // ADD = additive Farbmischung

fill(255, 0, 0, 255); // rot
ellipse(250, 150, 200, 200);

fill(0, 255, 0, 255); // grün
ellipse(150, 150, 200, 200);

fill(0, 0, 255, 255); // blau
ellipse(200, 250, 200, 200);
```

Sketch 08: Farbmischung_subtraktiv

```
// subtraktive Farbmischung

size(400, 400);
background(255);

blendMode(SUBTRACT); //SUBTRACT = subtraktive Farbmischung

fill(255, 0, 0, 255); // rot
ellipse(250, 150, 200, 200);

fill(0, 255, 0, 255); // grün
ellipse(150, 150, 200, 200);

fill(0, 0, 255, 255); // blau
ellipse(200, 250, 200, 200);
```

6.7 Spektren

Kontinuierliches Spektrum 01

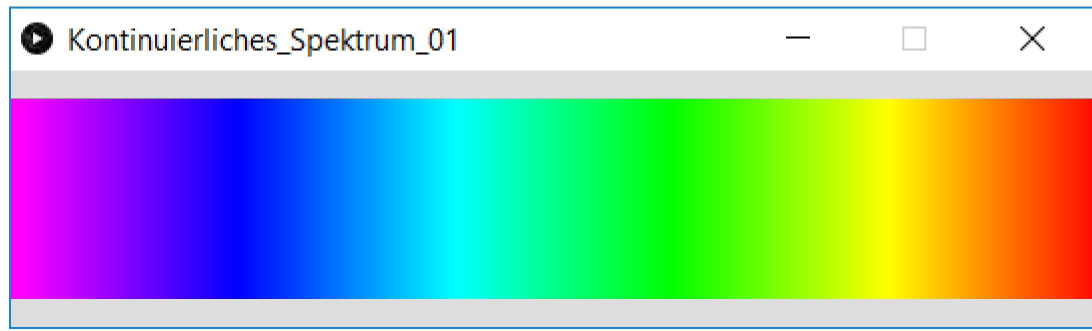


Abbildung 6.10: Kontinuierliches Spektrum

Nachdem wir Farben gemischt haben, wollen wir uns nun mit dem Thema Spektren beschäftigen. Erinnern wir uns an Kapitel *5.1 Harmonische Schwingungen*. Hier haben wir in der PDE bei Karteikarte *Tools* auf *Farbauswahl* geklickt und so den *Color Selector* aufgerufen. Im Fenster Color Selector sieht man u. a. ein kontinuierliches Spektrum. Klickt man an einer bestimmten Stelle auf dieses Spektrum, so werden einem rechts daneben die Werte für die Farbräume RGB und HSB angezeigt. Den Farbraum RGB (Rot, Grün, Blau) kennen wir aus den bisherigen Übungen, da er standardmäßig in Processing aufgerufen wird. Dem **Farbraum HSB** sind wir schon mal ohne weitere Erklärung im Sketch *Potenzialgebirge* (Kapitel 3) begegnet. Nun wollen wir uns aber etwas ausführlicher mit ihm beschäftigen. Schauen wir uns den Farbraum HSB im *Color Selector* genauer an (siehe Abb. 6.11). Wenn wir mit gedrückter linker Maustaste von unten nach oben durch das Spektrum fahren, dann sehen wir, dass sich der Wert für H von 0° bis 360° ändert. H steht für **Hue**, was übersetzt **Farbton** heißt. Er wird deshalb in Grad angegeben, weil die unterschiedlichen Farbtöne meist auf einem Kreisring aufgetragen werden. Der Buchstabe **S** im HSB Farbraum steht für **Saturation (Sättigung)**. Wenn wir mit gedrückter Maustaste waagerecht von links nach rechts durch das große Farbfenster im Color Selector fahren, dann ändert sich sein Wert von 0 bis 100. 100 steht für volle Farbintensität. Der Buchstabe **B** im HSB Farbraum steht für **Brightness (Helligkeit)**. Wenn wir mit gedrückter Maustaste senkrecht von unten nach oben durch das große Farbfenster im Color Selector fahren, dann ändert auch er sich von 0 bis 100. 100 steht für volle Helligkeit.

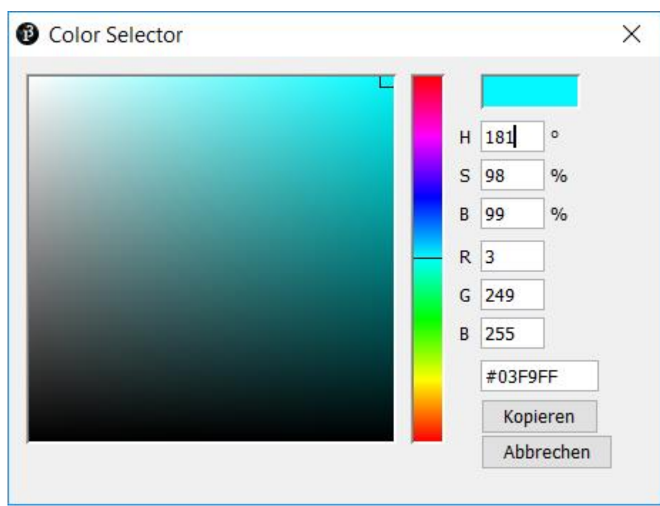


Abbildung 6.11: Der Color Selector von Processing

Die erste Idee, die sich aufdrängt, wenn man in Processing ein kontinuierliches Spektrum in Abhängigkeit von der Wellenlänge des Lichtes zeichnen will, ist die Folgende. Man sucht sich zum Vergleich in einem Physikbuch oder im Internet ein farbiges Spektrum mit Wellenlängenangabe. Nun klicken wir mit dem Mauszeiger auf eine Farbe im Spektrum des *Color Selectors*, um so den zugehörigen Hue-Wert zu erhalten. Auf dem Spektrum aus dem Physikbuch lesen wir die zur ausgewählten Farbe passende Wellenlänge ab. Die so ermittelten Wertepaare H und λ tragen wir in ein Tabellenkalkulationsprogramm ein. Hier fügen wir eine Trendlinie ein und lassen uns auch die Gleichung dazu anzeigen (siehe Abb. 6.12). Mittels dieser Gleichung können wir nun den sehr kurzen und einfachen Sketch *Kontinuierliches_Spektrum_01* schreiben, der uns ein schönes kontinuierliches Spektrum zeichnet (siehe Abb. 6.10).

Neu in diesem Sketch ist, dass wir zuerst Processing mitteilen müssen, dass es nicht standardmäßig im RGB-Farbraum arbeiten soll, sondern im HSB-Farbraum. Dies gelingt mit der Funktion

colorMode(HSB, 360, 100, 100);

Die Zahlenwerte in der Klammer hinter HSB geben den H-Wertebereich in Grad, den Sättigungsbereich in Prozent und den Helligkeitsbereich in Prozent an. Nachdem der Farbbereich festgelegt ist, geben wir bei allen Zeichnungsobjekten die Farbe wie in dem folgenden Beispiel an.

stroke(181, 100, 100);

Selbstverständlich kann man die einzelnen Werte auch über eine Gleichung bestimmen. In unserem Sketch ist dies der Farbton (Hue): stroke(Hue, 100, 100).

In der *for-Schleife* unseres Sketches bestimmen wir die Farben der Linien in Abhängigkeit von der Wellenlänge im Bereich von 380 nm bis 650 nm. Damit die Zeichnung nicht zu kurz gerät und auch nicht bei 0 nm beginnt, schreiben wir: line(2*L-760, height, 2*L-760, 0). D.h., wir verdoppeln die Wellenlänge und ziehen, da wir das Spektrum erst ab 380 nm zeichnen wollen, $2 \cdot 380 = 760$ ab.

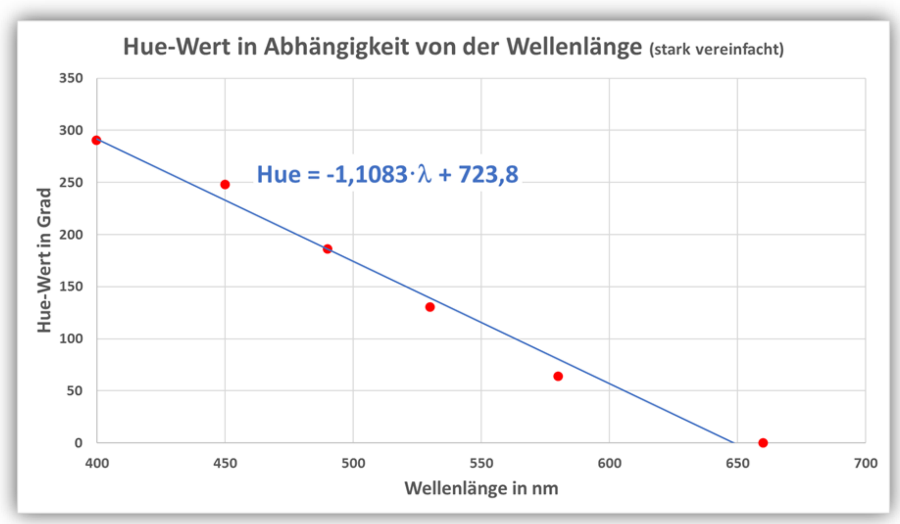


Abbildung 6.12: Sehr vereinfachte Darstellung des Zusammenhangs zwischen Wellenlänge und H-Wert

Sketch 09: Kontinuierliches_Spektrum_01

```
// kontinuierliches Spektrum 01

float Hue;

void setup()
{
  size(540, 100);
  background(0);
}

void draw()
{
  // Farbraum, H-Wertebereich, Sättigungsbereich, Helligkeitsbereich
  colorMode(HSB, 360, 100, 100);

  for (float L = 380; L <= 650; L = L + 0.5) // L = Wellenlänge
  {
    Hue = -1.1083*L + 723.8;
    stroke(Hue, 100, 100); // Farbwert, Sättigung, Helligkeit
    line(2*L-760, height, 2*L-760, 0);
  }
}
```

Hg-Absorptionsspektrum 01

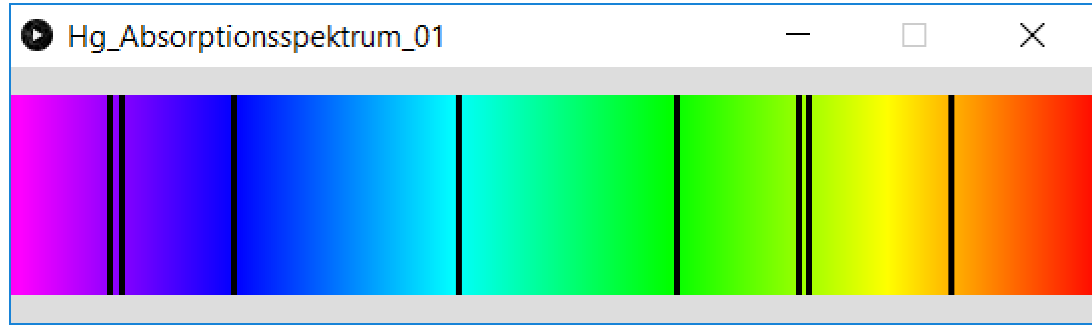


Abbildung 6.13: Die Absorptionslinien von Quecksilber in einem farblich nicht ganz korrekten kontinuierlichen Spektrum.

Überprüfen wir nun mittels der Absorptionslinien von Quecksilber (Hg), ob uns die Zuordnung zwischen Wellenlänge und Farbe gelungen ist. Dazu überdecken wir das kontinuierliche Spektrum mit schwarzen Linien, die den Hauptwellenlängen von Quecksilber entsprechen (siehe Abb. 6.13 und Sketch *Absorptionsspektrum_01*).

Für den kurzwelligen Bereich sieht es noch recht gut aus, doch dass die rechte Doppellinie nicht im gelben Bereich liegt, ist schlicht und einfach enttäuschend. So einfach wie gedacht ist es scheinbar doch nicht. Wir benötigen also einen neuen Ansatz. Mehr dazu unter der Überschrift *Kontinuierliches Spektrum 02*.

Sketch 10: Hg_Absorptionsspektrum_01

```
// Hg-Absorptionsspektrum 01

float Hue;

void setup()
{
  size(540, 100);
  background(0);
}

void draw()
{
  // Farbraum, H-Wertebereich, Sättigungsbereich, Helligkeitsbereich
  colorMode(HSB, 360, 100, 100);

  for (float L = 380; L < 650; L = L + 0.5) // L = Wellenlänge
  {
    Hue = -1.1083*L + 723.8;
    stroke(Hue, 100, 100); // Farbwert, Sättigung, Helligkeit
    line(2*L-760, height, 2*L-760, 0);
  }

  // Hg-Absorptionslinien
  stroke(0);
  strokeWeight(3);
  line(2*404.66-760, height, 2*404.66-760, 0);
  line(2*407.78-760, height, 2*407.78-760, 0);
  line(2*435.83-760, height, 2*435.83-760, 0);
  line(2*491.60-760, height, 2*491.60-760, 0);
  line(2*546.07-760, height, 2*546.07-760, 0);
  line(2*576.96-760, height, 2*576.96-760, 0);
  line(2*579.07-760, height, 2*579.07-760, 0);
  line(2*614.95-760, height, 2*614.95-760, 0);
}
```

Kontinuierliches Spektrum 02

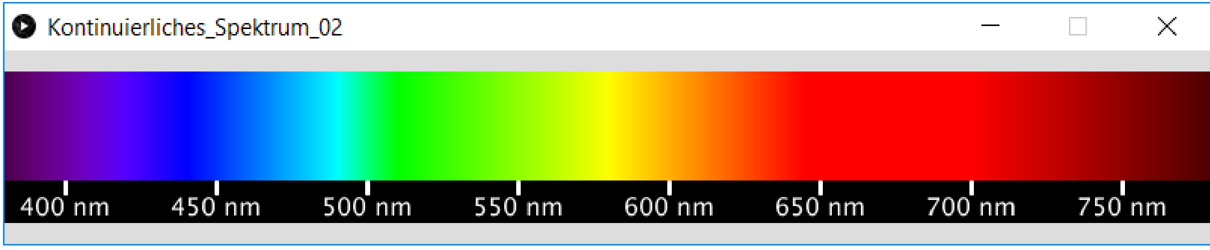


Abbildung 6.14: Verbessertes kontinuierliches Spektrum

Wie wir im letzten Abschnitt erkennen mussten, besteht zwischen Farbton und Wellenlänge leider kein einfacher linearer Zusammenhang. Bessere Ergebnisse erzielt man, wenn man den einzelnen Farbton-Abschnitten des HSB-Farbraumes unterschiedliche Wellenlängenbereiche zuordnet. Ausführliche Informationen hierzu findet man in dem Artikel von Uwe Kern [10].

Die Zuordnung der einzelnen Bereiche gelingt uns in dem folgenden Sketch mit verschachtelten if-else-Anweisungen. Und wenn wir es besonders gut machen wollen, dann senken wir an den

beiden Enden des Spektrums auch die Helligkeit ab, da unser Auge den UV- und Infrarotbereich bekanntlich nicht wahrnehmen kann.

Sketch 11: Kontinuierliches_Spektrum_02

```
// Kontinuierliches Spektrum 02

float h; // Farbton
float s = 100; // Sättigung
float b; // Helligkeit
float f = 0.1644562; // Faktor

void setup()
{
  size(800, 100);
  background(0);

  // Achsenbeschriftung
  stroke(255);
  strokeWeight(3);
  line(2*400-760, 73, 2*400-760, 80);
  line(2*450-760, 73, 2*450-760, 80);
  line(2*500-760, 73, 2*500-760, 80);
  line(2*550-760, 73, 2*550-760, 80);
  line(2*600-760, 73, 2*600-760, 80);
  line(2*650-760, 73, 2*650-760, 80);
  line(2*700-760, 73, 2*700-760, 80);
  line(2*750-760, 73, 2*750-760, 80);

  fill(255);
  textSize(16);
  textAlign(CENTER);
  text("400 nm", 2*400-760, 95);
  text("450 nm", 2*450-760, 95);
  text("500 nm", 2*500-760, 95);
  text("550 nm", 2*550-760, 95);
  text("600 nm", 2*600-760, 95);
  text("650 nm", 2*650-760, 95);
  text("700 nm", 2*700-760, 95);
  text("750 nm", 2*750-760, 95);
}

void draw()
{
  // Farbraum, H-Wertebereich, Sättigungsbereich, Helligkeitsbereich
  colorMode(HSB, 360, 100, 100);

  // Berechnung der h- und b-Werte und zeichnen der einzelnen
  // Wellenlängenbereiche
  for (float L = 380; L >= 380 && L <= 781; L = L + 0.5)
  {
    stroke(h, s, b); // Hue-Wert, Sättigung, Helligkeit

    if (L >= 380 && L < 440)
    {
      h = 360*(2.0/3.0 + f*(L - 440.0) / (380.0 - 440.0));
      b = 100*(0.3 + 0.7*(L - 380.0) / (420.0 - 380.0));
      line(2*L-760, 70, 2*L-760, 0);
    } else if (L >= 440 && L < 490)
```

```
    {
      h = 360*(2.0/3.0 - f*(L - 440.0) / (490.0 - 440.0));
      b = 100;
      line(2*L-760, 70, 2*L-760, 0);
    } else if (L >= 490 && L < 510)
    {
      h = 360*(1.0/3.0 + f*(L - 510.0) / (490.0 - 510.0));
      b = 100;
      line(2*L-760, 70, 2*L-760, 0);
    } else if (L >= 510 && L < 580)
    {
      h = 360*(1.0/3.0 - f*(L - 510.0) / (580.0 - 510.0));
      b = 100;
      line(2*L-760, 70, 2*L-760, 0);
    } else if (L >= 580 && L < 645)
    {
      h = 360*(0.0 + f*(L - 645.0) / (580.0 - 645.0));
      b = 100;
      line(2*L-760, 70, 2*L-760, 0);
    } else if (L >= 645 && L < 700)
    {
      h = 0;
      b = 100;
      line(2*L-760, 70, 2*L-760, 0);
    } else if (L >= 700 && L < 780)
    {
      h = 0;
      b = 100*(0.3 + 0.7*(L - 780.0) / (700.0 - 780.0));
      line(2*L-760, 70, 2*L-760, 0);
    }
  }
}
```

Hg-Absorptionsspektrum 02

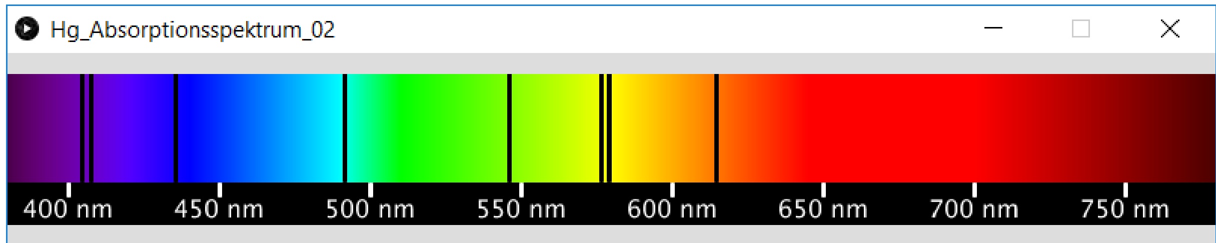


Abbildung 6.15: Verbessertes kontinuierliches Spektrum mit Hg-Absorptionslinien

Wenn wir nun in den Sketch *Kontinuierliches_Spektrum_02* die Absorptionslinien von Quecksilber (Hg) einfügen, dann sehen wir (Abb. 6.15), dass unser neuer Ansatz erfolgreich war.

Sketch 12: Hg_Absorptionsspektrum_02

```
// Hg-Absorptionsspektrum 02

float h; // Farbton
float s = 100; // Sättigung
float b; // Helligkeit
float f = 0.1644562; // Faktor
```

```
// Wellenlängen des Hg-Spektrums
float L1 = 404.66;
float L2 = 407.78;
float L3 = 435.83;
float L4 = 491.60;
float L5 = 546.07;
float L6 = 576.96;
float L7 = 579.07;
float L8 = 614.95;

void setup()
{
  size(800, 100);
  background(0);

  // Skala
  stroke(255);
  strokeWeight(3);
  line(2*400-760, 73, 2*400-760, 80);
  line(2*450-760, 73, 2*450-760, 80);
  line(2*500-760, 73, 2*500-760, 80);
  line(2*550-760, 73, 2*550-760, 80);
  line(2*600-760, 73, 2*600-760, 80);
  line(2*650-760, 73, 2*650-760, 80);
  line(2*700-760, 73, 2*700-760, 80);
  line(2*750-760, 73, 2*750-760, 80);

  fill(255);
  textSize(16);
  textAlign(CENTER);
  text("400 nm", 2*400-760, 95);
  text("450 nm", 2*450-760, 95);
  text("500 nm", 2*500-760, 95);
  text("550 nm", 2*550-760, 95);
  text("600 nm", 2*600-760, 95);
  text("650 nm", 2*650-760, 95);
  text("700 nm", 2*700-760, 95);
  text("750 nm", 2*750-760, 95);
}

void draw()
{
  // Farbraum, H-Bereich in Grad, Sättigungsbereich, Helligkeitsbereich
  colorMode(HSB, 360, 100, 100);

  for (float L = 380; L >= 380 && L <= 781; L = L + 0.5)
  {
    stroke(h, s, b); // Hue-Wert, Sättigung, Helligkeit

    if (L >= 380 && L < 420)
    {
      h = 360*(2.0/3.0 + f*(L - 440.0) / (380.0 - 440.0));
      b = 100*(0.3 + 0.7*(L - 380.0) / (420.0 - 380.0));
      line(2*L-760, 70, 2*L-760, 0);
    } else if (L >= 380 && L < 440)
    {
      h = 360*(2.0/3.0 + f*(L - 440.0) / (380.0 - 440.0));
      b = 100;
      line(2*L-760, 70, 2*L-760, 0);
    } else if (L >= 440 && L < 490)
```

```
    {
      h = 360*(2.0/3.0 - f*(L - 440.0) / (490.0 - 440.0));
      b = 100;
      line(2*L-760, 70, 2*L-760, 0);
    } else if (L >= 490 &&  L < 510)
    {
      h = 360*(1.0/3.0 + f*(L - 510.0) / (490.0 - 510.0));
      b = 100;
      line(2*L-760, 70, 2*L-760, 0);
    } else if (L >= 510 && L < 580)
    {
      h = 360*(1.0/3.0 - f*(L - 510.0) / (580.0 - 510.0));
      b = 100;
      line(2*L-760, 70, 2*L-760, 0);
    } else if (L >= 580 && L < 645)
    {
      h = 360*(0.0 + f*(L - 645.0) / (580.0 - 645.0));
      b = 100;
      line(2*L-760, 70, 2*L-760, 0);
    } else if (L >= 645 && L < 700)
    {
      h = 0;
      b = 100;
      line(2*L-760, 70, 2*L-760, 0);
    } else if (L >= 700 && L < 780)
    {
      h = 0;
      b = 100*(0.3 + 0.7*(L - 780.0) / (700.0 - 780.0));
      line(2*L-760, 70, 2*L-760, 0);
    }
  }

  // Hg-Absorptionslinien
  stroke(0);
  strokeWeight(3);
  line(2*L1-760, 70, 2*L1-760, 0);
  line(2*L2-760, 70, 2*L2-760, 0);
  line(2*L3-760, 70, 2*L3-760, 0);
  line(2*L4-760, 70, 2*L4-760, 0);
  line(2*L5-760, 70, 2*L5-760, 0);
  line(2*L6-760, 70, 2*L6-760, 0);
  line(2*L7-760, 70, 2*L7-760, 0);
  line(2*L8-760, 70, 2*L8-760, 0);
}
```

Hg-Emissionsspektrum

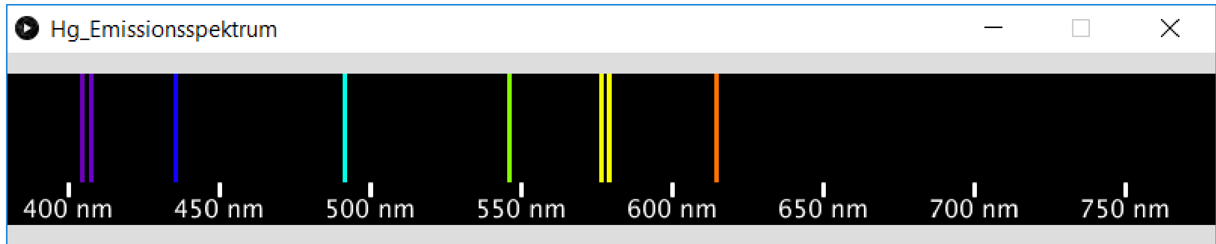


Abbildung 6.16: Emissionsspektrum von Quecksilber

Und weil es mit dem Absorptionsspektrum so gut funktioniert hat, zeichnen wir auch noch das Emissionsspektrum von Quecksilber (Abb. 6.16).

Sketch 13: Hg-Emissionsspektrum

```
// Hg-Emissionsspektrum

float h; // Farbton
float s = 100; // Sättigung
float b; // Helligkeit
float f = 0.1644562; // Faktor

// Wellenlängen des Hg-Spektrums
float L1 = 404.66;
float L2 = 407.78;
float L3 = 435.83;
float L4 = 491.60;
float L5 = 546.07;
float L6 = 576.96;
float L7 = 579.07;
float L8 = 614.95;

void setup()
{
  size(800, 100);
  background(0);

  // Achsenbeschriftung
  stroke(255);
  strokeWeight(3);
  line(2*400-760, 73, 2*400-760, 80);
  line(2*450-760, 73, 2*450-760, 80);
  line(2*500-760, 73, 2*500-760, 80);
  line(2*550-760, 73, 2*550-760, 80);
  line(2*600-760, 73, 2*600-760, 80);
  line(2*650-760, 73, 2*650-760, 80);
  line(2*700-760, 73, 2*700-760, 80);
  line(2*750-760, 73, 2*750-760, 80);

  fill(255);
  textSize(16);
  textAlign(CENTER);
  text("400 nm", 2*400-760, 95);
  text("450 nm", 2*450-760, 95);
  text("500 nm", 2*500-760, 95);
  text("550 nm", 2*550-760, 95);
  text("600 nm", 2*600-760, 95);
  text("650 nm", 2*650-760, 95);
  text("700 nm", 2*700-760, 95);
  text("750 nm", 2*750-760, 95);
}

void draw()
{
  // Farbraum, H-Wertebereich, Sättigungsbereich, Helligkeitsbereich
  colorMode(HSB, 360, 100, 100);

  // Berechnung der h- und b-Werte und zeichnen der Spektrallinien
  h = 360*(2.0/3.0 + f*(L1 - 440.0) / (380.0 - 440.0));
```

```
  b = 100*(0.3 + 0.7*(L1 - 380.0) / (420.0 - 380.0));
  stroke(h, s, b); // Hue-Wert, Sättigung, Helligkeit
  strokeWeight(3);
  line(2*L1-760, 70, 2*L1-760, 0);

  h = 360*(2.0/3.0 + f*(L2 - 440.0) / (380.0 - 440.0));
  b = 100*(0.3 + 0.7*(L2 - 380.0) / (420.0 - 380.0));
  stroke(h, s, b); // Hue-Wert, Sättigung, Helligkeit
  strokeWeight(3);
  line(2*L2-760, 70, 2*L2-760, 0);

  h = 360*(2.0/3.0 + f*(L3 - 440.0) / (380.0 - 440.0));
  b = 100*(0.3 + 0.7*(L3 - 380.0) / (420.0 - 380.0));
  stroke(h, s, b); // Hue-Wert, Sättigung, Helligkeit
  strokeWeight(3);
  line(2*L3-760, 70, 2*L3-760, 0);

  h = 360*(1.0/3.0 + f*(L4 - 510.0) / (490.0 - 510.0));
  b = 100;
  stroke(h, s, b); // Hue-Wert, Sättigung, Helligkeit
  strokeWeight(3);
  line(2*L4-760, 70, 2*L4-760, 0);

  h = 360*(1.0/3.0 - f*(L5 - 510.0) / (580.0 - 510.0));
  b = 100;
  stroke(h, s, b); // Hue-Wert, Sättigung, Helligkeit
  strokeWeight(3);
  line(2*L5-760, 70, 2*L5-760, 0);

  h = 360*(1.0/3.0 - f*(L6 - 510.0) / (580.0 - 510.0));
  b = 100;
  stroke(h, s, b); // Hue-Wert, Sättigung, Helligkeit
  strokeWeight(3);
  line(2*L6-760, 70, 2*L6-760, 0);

  h = 360*(1.0/3.0 - f*(L7 - 510.0) / (580.0 - 510.0));
  b = 100;
  stroke(h, s, b); // Hue-Wert, Sättigung, Helligkeit
  strokeWeight(3);
  line(2*L7-760, 70, 2*L7-760, 0);

  h = 360*(0.0 + f*(L8 - 645.0) / (580.0 - 645.0));
  b = 100;
  stroke(h, s, b); // Hue-Wert, Sättigung, Helligkeit
  strokeWeight(3);
  line(2*L8-760, 70, 2*L8-760, 0);
}
```

Hg-Absorptionsspektrum im RGB-Farbraum

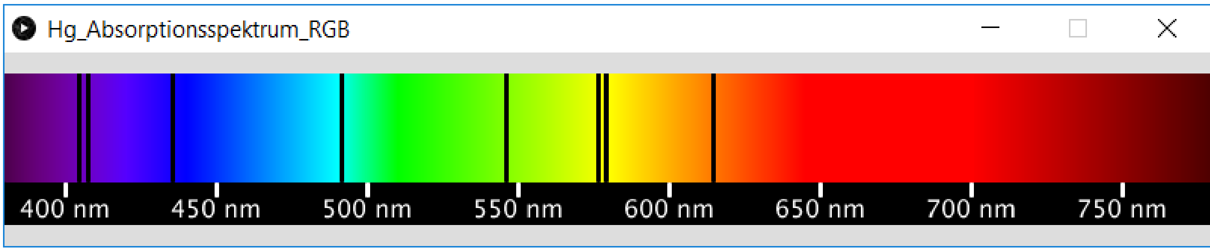


Abbildung 6.17: Im RGB-Farbraum erstelltes kontinuierliches Spektrum mit Hg-Absorptionslinien

Auch im RGB-Farbraum gelingt die Erstellung von Spektren (Abb. 6.17). Hier hilft der Algorithmus von Dan Bruton [9]. Die Farben Rot, Grün und Blau werden je nach Wellenlängenbereich in ein entsprechendes Verhältnis gesetzt (siehe Abb. 6.18). Eine Dämpfung an den Rändern des Spektrums erfolgt über den Faktor f (siehe Sketch). Burton erwähnt noch einen Wert mit der Bezeichnung γ. Ihn kann man ohne einen größeren Fehler zu begehen gleich 1 setzen, sodass er in dem folgenden Sketch keine Berücksichtigung findet.

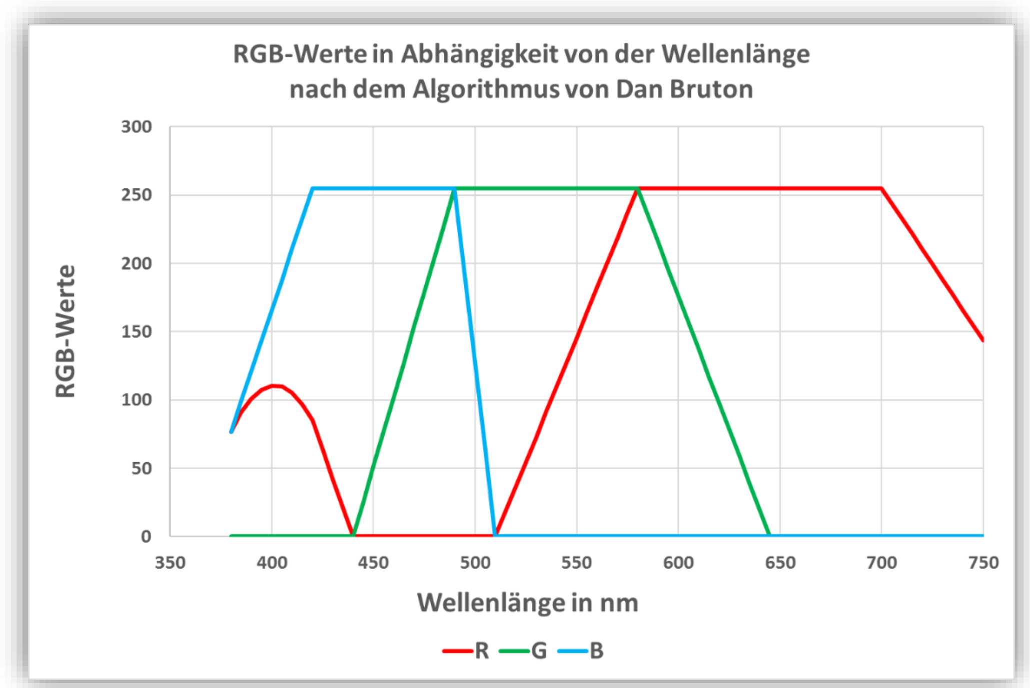


Abbildung 6.18: RGB-Werte in Abhängigkeit von der Wellenlänge nach Dan Bruton

Sketch 14: Hg_Absorptionsspektrum_RGB

```
// Hg-Absorptionsspektrum im RGB-Farbraum

int L; // Wellenlänge
float f; // Faktor
float rot;
float gruen;
float blau;

// Wellenlängen des Hg-Spektrums
float L1 = 404.66;
float L2 = 407.78;
float L3 = 435.83;
float L4 = 491.60;
float L5 = 546.07;
float L6 = 576.96;
float L7 = 579.07;
float L8 = 614.95;

void setup()
{
  size(800, 100);
```

```
  background(0);

  // Achsenbeschriftung
  stroke(255);
  strokeWeight(3);
  line(2*400-760, 73, 2*400-760, 80);
  line(2*450-760, 73, 2*450-760, 80);
  line(2*500-760, 73, 2*500-760, 80);
  line(2*550-760, 73, 2*550-760, 80);
  line(2*600-760, 73, 2*600-760, 80);
  line(2*650-760, 73, 2*650-760, 80);
  line(2*700-760, 73, 2*700-760, 80);
  line(2*750-760, 73, 2*750-760, 80);

  fill(255);
  textSize(16);
  textAlign(CENTER);
  text("400 nm", 2*400-760, 95);
  text("450 nm", 2*450-760, 95);
  text("500 nm", 2*500-760, 95);
  text("550 nm", 2*550-760, 95);
  text("600 nm", 2*600-760, 95);
  text("650 nm", 2*650-760, 95);
  text("700 nm", 2*700-760, 95);
  text("750 nm", 2*750-760, 95);
}

void draw()
{
  for (float L = 380; L >= 380 && L <= 781; L = L + 0.5)
  {
    stroke(f*rot, f*gruen, f*blau);

    if (L >= 380 && L < 440)
    {
      rot = -255*((L - 440.0) / (440.0 - 380.0));
      gruen = 0.0;
      blau = 255.0;
    } else if (L >= 440 && L < 490)
    {
      rot = 0.0;
      gruen = 255*((L - 440.0) / (490.0 - 440.0));
      blau = 255;
    } else if (L >= 490 &&  L < 510)
    {
      rot = 0.0;
      gruen = 255;
      blau = -255*((L - 510.0) / (510.0 - 490.0));
    } else if (L >= 510 && L < 580)
    {
      rot = 255*((L - 510.0) / (580.0 - 510.0));
      gruen = 255;
      blau = 0.0;
    } else if (L >= 580 && L < 645)
    {
      rot = 255;
      gruen = -255*((L - 645.0) / (645.0 - 580.0));
      blau = 0.0;
    } else if (L >= 645 && L < 781)
    {
      rot = 255;
```

```
        gruen = 0.0;
        blau = 0.0;
      } else
      {
        rot = 0.0;
        gruen = 0.0;
        blau = 0.0;
      }

      if (L >= 380 && L < 420)
      {
        f = 0.3 + 0.7 * (L - 380.0) / (420.0 - 380.0);
      } else if (L >= 420 && L < 701)
      {
        f = 1.0;
      } else if (L >= 701 && L < 781)
      {
        f = 0.3 + 0.7 * (780.0 - L) / (780.0 - 700.0);
      } else
      {
        f = 0.0;
      }

      line(2*L-760, 70, 2*L-760, 0);
    }

    // Hg-Absorptionslinien
    stroke(0);
    strokeWeight(3);
    line(2*L1-760, 70, 2*L1-760, 0);
    line(2*L2-760, 70, 2*L2-760, 0);
    line(2*L3-760, 70, 2*L3-760, 0);
    line(2*L4-760, 70, 2*L4-760, 0);
    line(2*L5-760, 70, 2*L5-760, 0);
    line(2*L6-760, 70, 2*L6-760, 0);
    line(2*L7-760, 70, 2*L7-760, 0);
    line(2*L8-760, 70, 2*L8-760, 0);
}
```

6.8 Zusammenfassung

loadPixels() — Mit *loadPixels()* laden wir die Information über jedes einzelne Pixel in unserem Fenster in ein eindimensionales Array. Nun können wir die Eigenschaften der Pixel verändern.

updatePixels() — Mit *updatePixels()* können wir die Veränderungen, die wir nach dem Aufrufen von *loadPixels()* vorgenommen haben, abspeichern.

Pixel-Array — Ein Pixel-Array ist ein eindimensionales Array zur Bearbeitung von Pixeleigenschaften, welches mit *loadPixels()* gefüllt und nach der Bearbeitung mit *updatePixels()* abgespeichert wird. Siehe hierzu *pixels[]* in der Referenz.

blendMode(ADD) — Mit *blendMode(ADD)* werden sich überlagernde Farben additiv gemischt.

blendMode(SUBTRACT) Mit *blendMode(SUBTRACT)* werden sich überlagernde Farben subtraktiv gemischt. Weitere Möglichkeiten der Farbmischung findet man in der Referenz unter *blendMode().*

colorMode() Standardmäßig arbeitet Processing im RGB-Farbraum. Möchte man im HSB-Farbraum arbeiten, dann benötigt man die Funktion *colorMode().*

```
colorMode(HSB, 360, 100, 100);
```

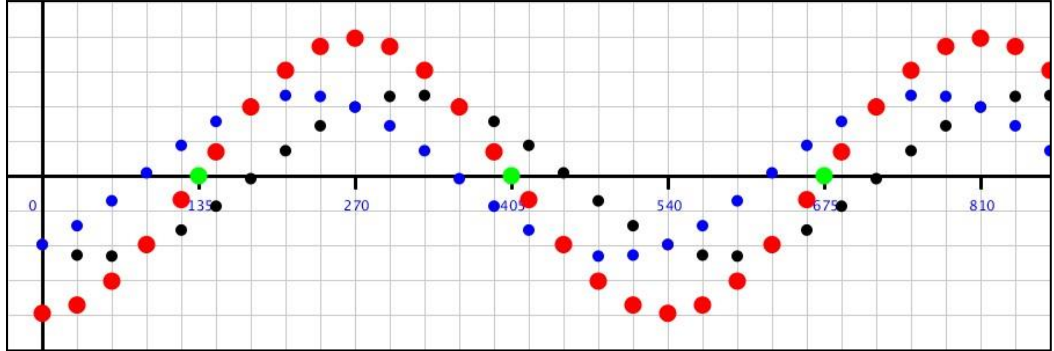
Die Zahlenwerte in der Klammer hinter HSB geben den **H**-Wertebereich in Grad, den Sättigungsbereich **S** in Prozent und den Helligkeitsbereich **B** in Prozent an.

HSB-Farbraum Beim HSB-Farbmodell wird der Farbton **H** (Hue) als Gradzahl zwischen 0° und 360° auf einem Kreis angegeben. Rot ≙ 0°, Grün ≙ 120° und Blau ≙ 240°. Die Farbintensität oder Sättigung **S** (Saturation) wird in Prozent angegeben. 100% entspricht der vollen Sättigung. Die Helligkeit **B** (Brightness) wird ebenfalls in Prozent angegeben. 100% entspricht der vollen Helligkeit. Die HSB-Werte und die zugehörigen Farben kann man sich im *Color Selector* von Processing anzeigen lassen.

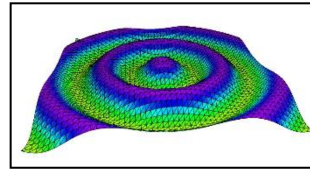
Color Selector Den *Color Selector* von Processing findet man unter *Tools* → *Farbauswahl*. In seinem Fenster kann man mit dem Mauszeiger Farben auswählen. Der *Color Selector* zeigt dann die entsprechenden Werte als RGB-Werte, als HSB-Werte und in hexadezimaler Schreibweise an.

6.9 Aufgaben

1. Ändere den Sketch *Wellenmaschine_02* so um, dass mit der Wellenmaschine eine stehende Welle demonstriert werden kann (siehe Abbildung unten). Hierbei soll die schwarze Welle von links nach rechts laufen und die blaue von rechts nach links. Die Knotenpunkte der stehenden Welle (Rot) sollen die Farbe Grün erhalten.



2. Schreibe den folgenden Sketch: Im Mittelpunkt des Fensters sitzt ein Erreger, der kontinuierlich eine Kreiswelle erzeugt. Diese Kreiswelle soll wie im Sketch *stehende_Welle_3D_animiert* dreidimensional animiert dargestellt werden (siehe Abbildung).

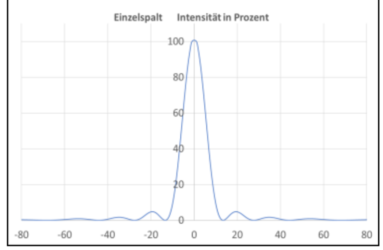


3. Entsprechend dem Sketch *Doppelspalt* soll auch für einen Einzelspalt das zugehörige Interferenzmuster erzeugt werden (siehe Abbildung oben rechts). Hierbei hilft das Huygensche Prinzip, nach dem man sich in der Spaltebene viele schwingende Erreger von Elementarwellen vorstellen kann.

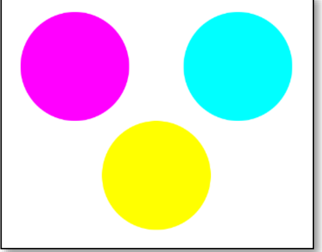
   Auf den ersten Blick erscheinen einem die Nebenmaxima im Processingfenster recht dunkel. Man muss jedoch bedenken, dass das Interferenzmuster des Einzelspaltes ein sehr helles Hauptmaximum und nur schwach ausgeprägte Nebenmaxima besitzt (siehe Abbildung unten rechts).

   Tipp: Will man im Fenster den Kontrast zwischen Maxima und Minima verstärken, dann kann man in der folgenden Programmzeile die Elongation e zum Beispiel um den Faktor 2 vergrößern.

   ```
   pixels[y * width + x] = color(128 + 2*e * 128);
   ```



4. Die Grundfarben der subtraktiven Farbmischung sind Gelb, Magenta und Cyan. In unserem Sketch *Farbmischung_subtraktiv* haben wir jedoch die Farben Rot, Grün und Blau in den einzelnen Programmzeilen eingegeben. Warum erschienen im Fenster aber die Farben Gelb, Magenta und Cyan? Was wird hier von wem subtrahiert? Wie ändern sich die Farben, wenn man einen schwarzen Hintergrund wählt? Um dies zu verstehen, erstelle einen Sketch der die Abbildung rechts erzeugt.



5. Schreibe einen Sketch, der das Absorptionsspektrum von Helium erzeugt.

6. Schreibe einen Sketch, der das Emissionsspektrum von Helium erzeugt.

7 Quantenphysik und Atomphysik

Was erwartet uns?

random(), PImage, loadImage(), imageMode(), image(), randomGaussian(), while(), break,

7.1 Doppelspalt „quantenphysikalisch“

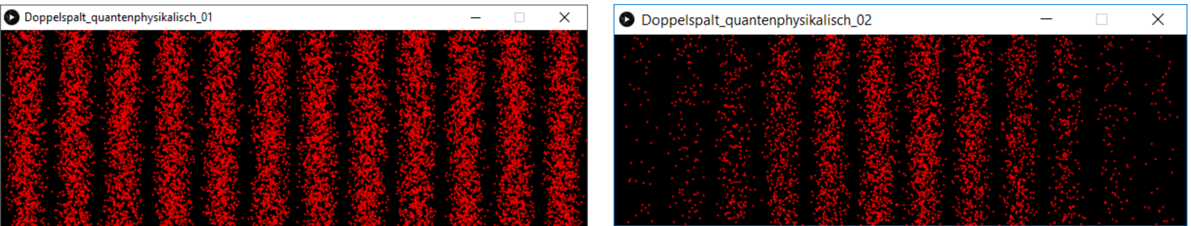


Abbildung 7.1: Interferenzmuster ohne Randverdunkelung (links) und mit Randverdunkelung (rechts)

Doppelspalt ohne Randverdunkelung

Was ist Licht? Nobelpreisträger Richard P. Feynman schreibt in seinem Buch *QED – Die seltsame Theorie des Lichts und der Materie* hierzu Folgendes: „In Wirklichkeit ist das Verhalten des Lichts das von Teilchen. … Alle Instrumente, die empfindlich genug sind, schwaches Licht aufzufangen, haben stets dasselbe entdeckt: Licht besteht aus Teilchen." Hierzu sei angemerkt, dass die Lichtteilchen, die sogenannten Photonen, nicht als Teilchen im klassischen Sinne verstanden werden dürfen. Photonen sind Quantenobjekte, deren Verhalten nur mithilfe von Wahrscheinlichkeiten beschrieben werden kann. Wenn zum Beispiel ein Photon einen Doppelspalt passiert, dann können wir nicht mit Sicherheit voraussagen an welcher Stelle es auf dem nachfolgenden Schirm auftrifft. Treten jedoch sehr viele Photonen durch den Doppelspalt, so erkennen wir das Streifenmuster, welches uns aus der klassischen Wellenlehre bekannt ist.

Diesen Vorgang, wie aus stochastisch verteilten Einzelereignissen ein Streifenmuster entsteht, wollen wir in unserem Sketch allerdings nur qualitativ darstellen (siehe Abb. 7.1). Qualitativ deshalb, weil die Lösung der Schrödingergleichung für den Doppelspalt in der gymnasialen Oberstufe nicht thematisiert wird, bzw. thematisiert werden kann.

Um die stochastische Verteilung der Photonenauftreffpunkte auf dem Schirm (Abb. 7.1) darzustellen, benötigen wir den Zufallsgenerator ***random()*.** Enthält die Klammer nur einen Zahlenwert, so generiert die *random*-Funktion eine *float*-Zahl zwischen Null und dem Zahlenwert. Stehen zwei Werte in der Klammer, so werden *float*-Zahlen zwischen diesen beiden Werten erzeugt.

Mit unserem Sketch *Doppelspalt_quantenphysikalisch_01* erzeugen wir die Abbildung 7.1 links. Der zweifache Einsatz der Funktion *random()* sorgt im Rahmen der gegebenen Gleichungen für eine zufällige Verteilung der roten Punkte (Photonenlokalisationen) im Fenster. Alle wichtigen Programmzeilen werden im Sketch selbst erklärt.

Sketch 01: Doppelspalt_quantenphysikalisch_01

```
// Doppelspalt "quantenphysikalisch" ohne Randverdunklung

float n = 50; // n ist der Abstand zwischen zwei Maxima

void setup()
{
  size(600, 200);
  background(0);
}

void draw()
{
  // Um die Entwicklung des Streifenmusters besser beobachten zu können,
  // wird die Bildwiederholungsrate reduziert

  frameRate(2);

  for (float x = 0; x < 600; x++)
  {
    /* p nimmt für 0, PI, 2*PI, 3*PI, ... also für x = 0, x = 50,
       x = 100, x = 150, den Wert 0 an.
       Für PI/2, 3*PI/2, 5*PI/2, nimmt p den Wert 1 bzw. -1 an. Dies ist
       für n = 50 bei x = 25, x = 75, x = 125,  der Fall */

    float p = sin(PI * x / n);

    /* random(1) generiert einen Wert zwischen 0 und 1. Wenn dieser Wert
     kleiner p*p, also kleiner 1 ist, dann ist die if-Bedingung erfüllt
     und es wird ein Punkt am Orte x gezeichnet. Der y-Wert für diesen
     Punkt wird unten mittels random(height) festgelegt.
     Wenn p = 1 oder p = -1 ist, dann ist die Wahrscheinlichkeit am
     höchsten, dass ein Punkt im Fenster gezeichnet wird.
     Für Werte p < 1 ist die Wahrscheinlichkeit geringer */

    if (random(1.0) < p * p)
    {
      stroke(255, 0, 0);
      strokeWeight(2);
      point(x, random(height)); // random(height) generiert einen y-Wert
                                // zwischen 0 und der Fensterhöhe
    }
  }
}
```

Doppelspalt mit Randverdunkelung

Abbildung 7.1 rechts zeigt an den Rändern eine Abdunkelung des Streifenmusters. Damit kommt sie der Realität etwas näher. Diese Randverdunkelung erreichen wir mithilfe einer e-Funktion. Der Sketch *Doppelspalt_quantenphysikalisch_02* unterscheidet sich also von dem Sketch *Doppelspalt_quantenphysikalisch_01* nur durch diese zusätzliche e-Funktion. Anstelle von

```
float p = sin(PI * x / n)
```

schreiben wir nun

```
float p = sin(PI * x / n)* exp(pow((x / 600 - 0.5) * 2.7, 2) * -1);
```

Schauen wir uns die hinzugekommene e-Funktion nun schrittweise etwas näher an. Der Klammerwert $\frac{x}{600} - 0{,}5$ nimmt Werte von - 0,5 bis + 0,5 an, da x von 0 bis 600 läuft. Da der Wertebereich mit 2,7 multipliziert und anschließend quadriert wird, verändert er sich von + 1,8225 über 0 bis + 1,8225. Somit ändert sich der Wert der e-Funktion von +6,1873 über 1 (Hinweis: $e^0 = 1$) bis +6,1873. Dies würde jedoch eine Verdunklung in der Mitte des Streifenmusters ergeben (Abb. 7.2 links). Da wir aber eine Abdunkelung an den Rändern wünschen, multiplizieren wir die e-Funktion noch mit -1 (Abb. 7.2 rechts).

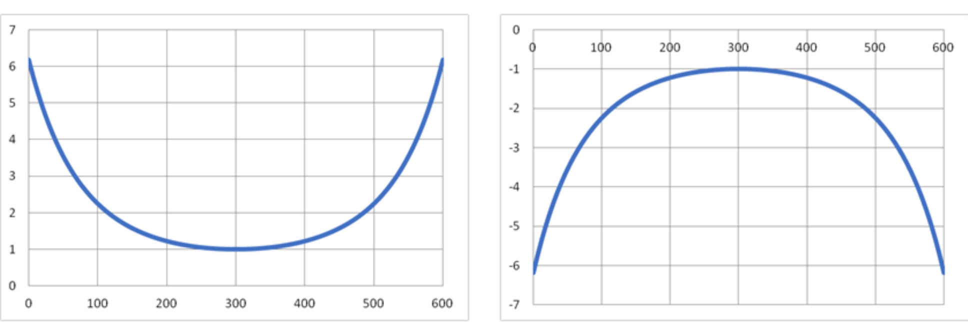


Abbildung 7.2: e-Funktion aus dem Sketch Doppelspalt_quantenphysikalisch_02 vor und nach der Multiplikation mit -1

Sketch 02: Doppelspalt_quantenphysikalisch_02

```
// Doppelspalt "quantenphysikalisch" mit Randverdunklung

float n = 50; // n ist der Abstand zwischen zwei Maxima

void setup()
{
  size(600, 200);
  background(0);
}

void draw()
{
  // Um die Entwicklung des Streifenmusters besser beobachten zu können,
  // wird die Bildwiederholungsrate reduziert.

  frameRate(2);

  for (float x = 0; x < 600; x++)
  {
    /* p nimmt für 0, PI, 2*PI, 3*PI, ... also für x = 0, x = 50,
       x = 100, x = 150, den Wert 0 an.
       Für PI/2, 3*PI/2, 5*PI/2, nimmt p den Wert 1 bzw. -1 an. Dies ist
       für n = 50 bei x = 25, x = 75, x = 125, der Fall. Die e-Funktion
       bewirkt eine Randverdunkelung. */

    float p = sin(PI * x / n) * exp(pow((x / 600 - 0.5) * 2.7, 2) * -1);


    /* random(1) generiert einen Wert zwischen 0 und 1. Wenn dieser Wert
     kleiner p*p, also kleiner 1 ist, dann ist die if-Bedingung erfüllt
     und es wird ein Punkt am Orte x gezeichnet. Der y-Wert für diesen
     Punkt wird unten mittels random(height) festgelegt.
     Wenn p = 1 oder p = -1 ist, dann ist die Wahrscheinlichkeit am
     höchsten, dass ein Punkt im Fenster gezeichnet wird.
```

```
     Für Werte p < 1 ist die Wahrscheinlichkeit geringer */

    if (random(1.0) < p * p)
    {
      stroke(255, 0, 0);
      strokeWeight(2);
      point(x, random(height)); // random(height) generiert einen y-Wert
                                // zwischen 0 und der Fensterhöhe
    }
  }
```

7.2 Zeigerformalismus

Zeigerdiagramme haben wir schon im Kapitel Elektrik beim Wechselstromkreis kennengelernt. Welche Bedeutung hat das Zeigermodell nun in der Quantenphysik? Nun, das Verhalten von Quantenobjekten wird sehr gut durch die Schrödingergleichung beschrieben. Diese Differenzialgleichung ist jedoch für die meisten Versuchsanordnungen nicht gerade einfach zu lösen.

Zum Glück hat der Nobelpreisträger Richard P. Feynman uns mit seinem Buch *QED DIE SELTSAME THEORIE DES LICHTS UND DER MATERIE* einen Formalismus an die Hand gegeben, mit dem man das Verhalten von Quantenobjekten mithilfe eines Computers recht einfach und trotzdem mit der richtigen Wahrscheinlichkeit voraussagen kann. Anhand eines Sketchs für einen Einzelspalt soll dies nun erläutert werden (siehe Abbildung 7.3). Ein von der Quelle Q ausgehendes Quantenobjekt (Photon, Elektron, …) durchläuft den Einzelspalt in Richtung des weiß dargestellten Schirmes (Abb. 7.3). Wie groß ist nun die Wahrscheinlichkeit, dass es am Punkte E des Schirmes auftrifft? Um dies mithilfe des Zeigermodells zu ermitteln, müssen wir möglichst viele Wege berücksichtigen, die das Quantenobjekt von Q nach E nehmen kann. In der linken Abbildung 7.3 sind sechs Wege in der Farbe blau eingezeichnet. In der rechten Abbildung sind 51 Wege eingezeichnet. Je mehr Wege wir bei unserer Simulation berücksichtigen, desto genauer wird unsere Aussage darüber, mit welcher Wahrscheinlichkeit das Quantenobjekt bei E auf den Schirm trifft.

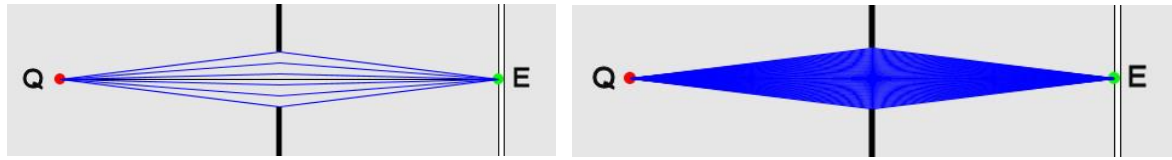


Abbildung 7.3: Links sind sechs Wege für ein Quantenobjekt durch einen Einzelspalt eingezeichnete, rechts 51 Wege

Tasten wir mit dem grünen Empfänger E den ganzen Schirm ab, so erhalten wir bei einer Wellenlänge von 10 Pixeln und einer Spaltbreite von 50 Pixeln die folgende Wahrscheinlichkeitsverteilung (genauer Wahrscheinlichkeitsdichteverteilung) (Abb. 7.4).

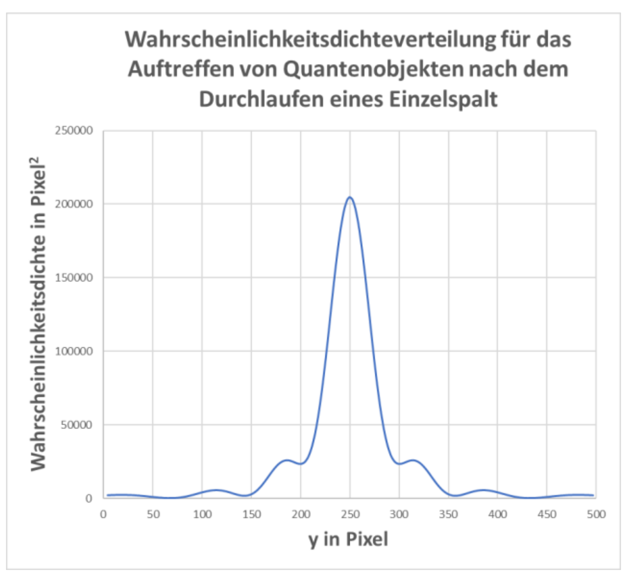


Abbildung 7.4: Wahrscheinlichkeitsdichteverteilung hinter einem Einzelspalt

Wie folgt nun aus der Abbildung 7.3 das Diagramm in Abbildung 7.4? Denken wir uns ein Rad mit dem Umfang der Wellenlänge λ und einem aufgemalten Zeiger am Ort der Strahlungsquelle Q (siehe Abb. 7.5). Dieses Rad rollen wir nun entlang des blau gezeichneten Weges bis zum Ort des Empfängers E und merken uns die Stellung des Zeigers am Orte E. Dies wiederholen wir für alle Wege, die wir in unseren Sketch vom Ort Q zum Ort E eingezeichnet haben. Also in unserem Sketch 51mal.

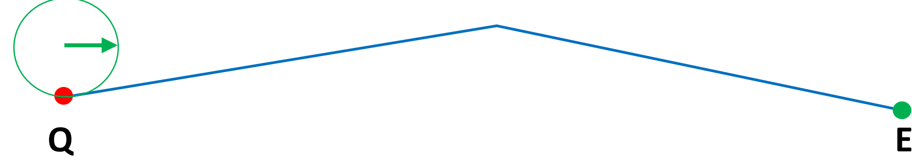


Abbildung 7.5: Ein Rad mit Zeiger rollt entlang des blauen Weges von der Quelle zum Empfänger

Anschließend addieren wir alle so ermittelten Zeiger vektoriell. Das Quadrat des Betrages des resultierenden Vektors (roter Vektor in Abb. 7.6) ist ein Maß für die Wahrscheinlichkeit, das Quantenobjekt am Orte E anzutreffen.

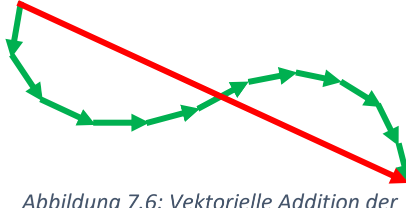


Abbildung 7.6: Vektorielle Addition der Einzelpfeile

Diesen Vorgang wiederholen wir für alle Pfade, die zum jeweiligen Ort auf der Leinwand führen. Fügen wir diese Werte nun in ein Tabellenkalkulationsprogramm, wie z.B. Excel ein, so erhalten wir das Diagramm von Abbildung 7.4. Soweit die Vorüberlegungen.

Die einzelnen Programmschritte in dem nun folgenden Sketch *Zeigerformalismus 01* werden im Sketch selber erläutert.

Angemerkt sei nur noch, dass in unserem Sketch eine **while()-Schleife** verwendet wird. In der uns bekannten *for-Schleife* ist die Anzahl der Schleifendurchläufe durch die in ihr aufgeführten

Bedingungen festgelegt. Eine *while()-Schleife* wird dagegen solange durchlaufen, wie die in ihr formulierte Bedingung erfüllt ist. Dies ist auf der einen Seite sehr praktisch, wenn man im Vorfeld nicht weiß, wie viele Schleifendurchläufe man benötigt, um einen bestimmten Zustand zu erreichen. Auf der anderen Seite besteht aber auch die Gefahr, dass man eine **Endlosschleife** erhält, wenn man innerhalb der *while()-Schleife* keine Abbruchbedingung einfügt. In unserem Sketch beenden wir die *void draw()-Schleife* mit *noLoop()* in der letzten if-Anweisung. Mit **break** verlassen wir dann die *while()-Schleife*.

Sketch 03: Zeigerformalismus

```
// Zeigerformalismus für den Einzelspalt

// Zuerst werden die Variablen definiert. Einige davon als Vektor, da
// wir, wie im Text erwähnt, auch vektoriell rechnen müssen.
PVector Q; // Ortsvektor der Quelle der Quantenobjekte
PVector E; // Ortsvektor des Empfängers
PVector M; // M ist der Vektor, der auf die Spaltöffnung zeigt
PVector PfadResultierend = new PVector(0, 0); /* Vektorsumme der Pfeile
von allen berücksichtigten Wege zwischen der feststehenden Quelle Q und
dem feststehenden Empfänger E */
float EinzelspaltX = 250; // Lage des Einzelspaltes
float EinzelspaltBreite = 50;
float Wellenlaenge = 10;
float PfadAbstand = 1; // Abstand der Pfade in der Spaltöffnung in Pixel
boolean AufbauNeuZeichnen = false; // Die Variable AufbauNeuZeichnen vom
                                   // Typ boolean wird deklariert

void AufbauZeichnen()
{
  background(230, 230, 230);

  // Die Elemente des Aufbaus werden flexibel initialisiert, sodass er
  // sich bei Änderung der obigen Variablen automatisch anpasst
  strokeWeight(1);
  stroke(0);
  line(Q.x, height / 2, E.x, height / 2); // Mittellinie
  fill(255);
  rect(E.x, 0, 5, height); // Schirm hinter dem Empfänger
  noStroke();
  fill(255, 0, 0);
  ellipse(Q.x, Q.y, 10, 10); // Quelle
  fill(0, 255, 0);
  ellipse(E.x, E.y, 10, 10); // Empfänger
  fill(0);
  rect(EinzelspaltX - 2.5, 0, 5, (height - EinzelspaltBreite) / 2);
  // Obere Blende des Einzelspaltes
  rect(EinzelspaltX - 2.5, height - (height - EinzelspaltBreite) / 2, 5,
  (height - EinzelspaltBreite) / 2); // Untere Blende des Einzelspaltes
}

void setup()
{
  size(1000, 500);

  // Initialisierung der oben definierten Vektoren
  Q = new PVector(50, height / 2); // Lage der Quelle
  E = new PVector(450, height / 2); // Lage des Empfängers
```

```
  M = new PVector(EinzelspaltX, (height - EinzelspaltBreite) / 2);
  // Die Spitze des M-Vektors liegt zuerst an der Oberseite der
  // Spaltöffnung

  AufbauZeichnen();
}

void mousePressed() // Die mousePressed()-Funktion wird aufgerufen, wenn
                    // eine Maustaste gedrückt ist.

  E.y = mouseY; // Der Empfänger nimmt bei gedrückter Maustaste den
                // y-Wert des Mauszeigers an
  M.y = (height - EinzelspaltBreite) / 2;

  PfadResultierend.set(0, 0); /* Setzt das Programm auf den Startwert
         zurück. D.h., für jede Stellung des Empfängers kann die
         PfadResultierende bei null beginnend neu berechnet werden */

  loop(); // Die draw()-Funktion wird wieder durchlaufen
  AufbauNeuZeichnen = true; //Der Versuchsaufbau wird beim nächsten
                      // Aufruf der draw()-Funktion neu gezeichnet
}

void mouseDragged() // Die mouseDragged()-Funktion wird aufgerufen, wenn
                 // eine Maustaste gedrückt ist und sich die Maus bewegt
{
  E.y = mouseY;
  M.y = (height - EinzelspaltBreite) / 2; //Setzt das Programm auf die
                                          // Startwerte zurück

  PfadResultierend.set(0, 0); /* Setzt das Programm auf den Startwert
     zurück. D.h., für jede Stellung des Empfängers kann die
     PfadResultierende bei null beginnend neu berechnet werden */

  loop(); // Die draw()-Funktion wird wieder durchlaufen
  AufbauNeuZeichnen = true; // Der Versuchsaufbau wird beim nächsten
                      // Aufruf der draw()-Funktion neu gezeichnet
}

void draw()
{
  if (AufbauNeuZeichnen == true)
  {
    AufbauZeichnen(); // Der Aufbau wird gezeichnet
    AufbauNeuZeichnen = false; //Die Aufbauzeichnung wird nun nicht mehr
                               // neu gezeichnet
  }

  while (true) // while()-Schleife
  {
    stroke(0, 0, 255);
    line(Q.x, Q.y, M.x, M.y); // Linie von der Quelle zur Spaltmitte
    line(M.x, M.y, E.x, E.y); // Linie von der Spaltmitte zum Empfänger

    float PfadLaenge = Q.dist(M) + M.dist(E); // Die Pfadlänge wird
                                              // berechnet
    float Phase = PfadLaenge / Wellenlaenge; // Die Phase (Winkel) wird
                                             // berechnet
    PVector PfadVektor = new PVector(cos(Phase * 2 * PI) * 10, sin(Phase
    * 2 * PI) * 10); // Am Anfang (Phase = 0) liegt der Zeiger auf der
                     // Rechtswertachse
```

```
    stroke(0, 200, 0);
    // Nun werden die einzelnen Pfadvektoren (grün) gezeichnet.
    // Beginnend am Anfang der Pfadresultierenden.

    line(PfadResultierend.x + 600, PfadResultierend.y + 100,
    PfadResultierend.x + PfadVektor.x + 600, PfadResultierend.y
    + PfadVektor.y + 100);

    PfadResultierend.add(PfadVektor); // Die grünen Pfadvektoren werden
                                      // aneinandergereiht

    M.y += PfadAbstand; /* Der Abstand der einzelnen Pfade wird um den
    PfadAbstand so lange erhöht, bis die untere Spaltblende erreicht
    ist */

    if (M.y > height - (height - EinzelspaltBreite) / 2)
    {
      strokeWeight(3);
      stroke(255, 0, 0);
      line(600, 100, 600 + PfadResultierend.x, 100 +
PfadResultierend.y);
      // Die Resultierende wird gezeichnet

      noStroke();
      noLoop(); // Schleifendurchlauf von void draw() wird gestoppt
      println(E.y, ",", PfadResultierend.mag() *
PfadResultierend.mag());
      /* Die Werte für die Lage des Empfängers und für das Quadrat der
       PfadResultierenden werden in die Konsole geschrieben */

      break; // Nun wird die while-Schleife verlassen
    }
  }
}
```

In der grafischen Darstellung erhält man nun das folgende Bild.

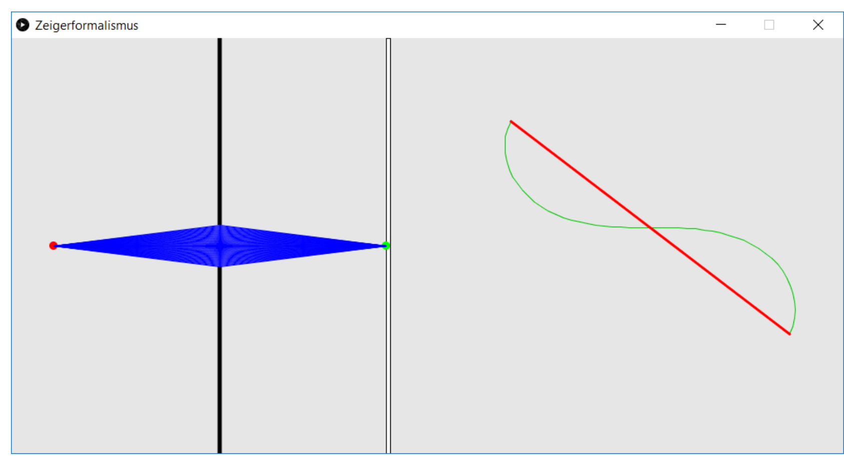


Abbildung 7.7: Mit dem Sketch Zeigerformalismus erstellte Abbildung

Wie das Bild der aneinandergereihten grünen Pfadvektoren aussieht, hängt von der Wellenlänge und der Spaltöffnung ab (siehe Abb. 7.8).

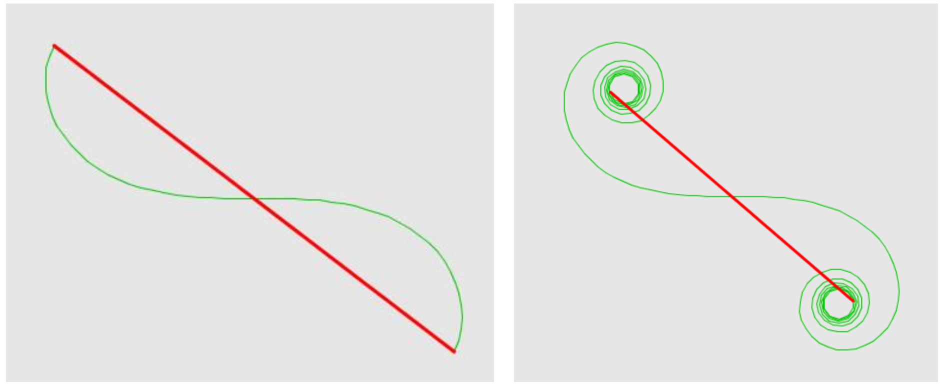


Abbildung 7.8: links λ = 10 und Spaltbreite = 50, rechts λ = 10 und Spaltbreite = 250

Werte für den Einzelspalt grafisch darstellen

Nun wollen wir die Wahrscheinlichkeit grafisch darstellen, mit der ein Quantenobjekt an einem bestimmten Ort auf dem Schirm angetroffen werden kann. Dazu müssen wir uns in dem obigen Sketch *Zeigerformalismus* die Quadrate der Beträge des resultierenden roten Vektors in der Konsole mit dem Befehl *println(PfadResultierend.mag() * PfadResultierend.mag())* sinnvoll auflisten lassen. Sinnvoll bedeutet, wir klicken mit der linken Maustaste den oberen Rand der Leinwand an (siehe Abb. 7.9) und ziehen dann mit gedrückter linker Maustaste den Empfänger ganz langsam nach unten. Ganz langsam deshalb, damit auch nahezu alle Werte vom Computer erfasst und in die Konsole geschrieben werden können. Nur so können wir später einen optisch ansprechenden Graphen zeichnen. In der Praxis erhalten wir jedoch weniger Werte als das Fenster hoch ist. Wichtig ist auch, dass hinter jedem Wert ein Komma steht, damit sich diese leichter zur grafischen Darstellung verwenden lassen.

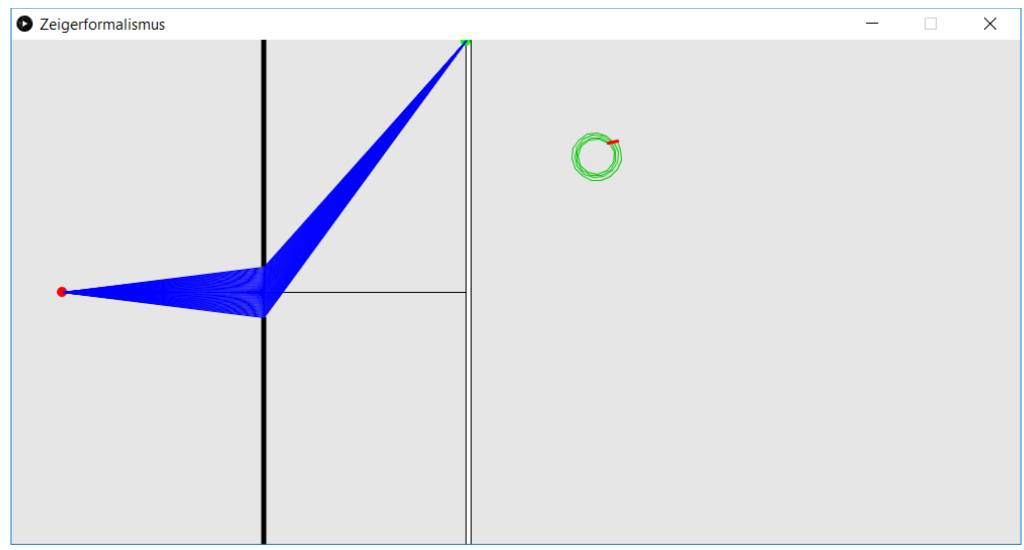


Abbildung 7.9: Startposition zur Aufzeichnung der Betragsquadrate des roten resultierenden Zeigers

Nun kennzeichnen wir einige Werte in der Konsole mit gedrückter linker Maustaste. Danach drücken wir die Tastenkombination *strg+A,* um alle Werte zu markieren und anschließend drücken wir die Tastenkombination *strg+C,* um alle Werte in der Zwischenablage zu speichern. Diese Werte fügen wir nun in ein Array innerhalb des neuen Sketches mit dem Namen *Diagramm_Einzelspalt* ein (siehe unten). Nach dem Einfügen löschen wir die Leerstellen und die erste deutlich zu große

Zahl. In der *for-Schleife* bei *void draw()* müssen wir nun noch die richtige Anzahl der im Array eingefügten Werte angeben. Diese müssen wir jedoch nicht einzeln zählen, sondern wir können die Anzahl der Werte anhand der Nummerierung der Programmzeilen ausrechnen. Bei unserem Sketch steht der erste Wert in der Programmzeile 7 und der letzte Wert in der Programmzeile 442. Also 442 – 6 = 436. In die *for-Schleife* geben wir i < 435 ein, da Processing ja bei null anfängt zu zählen und wir zur Darstellung der letzten Linie noch einen Wert hinzuzählen.

```
for (int i = 0; i < 435; i++) // Das Array enthält 436 Zahlenwerte.
{
  stroke(255, 0, 0);
  strokeWeight(2);
  line(2*i, -0.003*y[i], 1+2*i, -0.003*y[i+1]);
}
```

Da die y-Werte sehr groß sind und die y-Achse bei Processing nach unten zeigt, multiplizieren wir sie mit dem Wert -0.003. Damit unsere Darstellung in x-Richtung nicht zu schmal wird, verdoppeln wir außerdem mit 2*i noch den Wert für die Rechtswertachse.

Sketch 04: Diagramm_Einzelspalt

```
// Diagramm Einzelspalt erstellt mittels Zeigerformalismus

float[] y =
{
      101.76762 ,
      124.06428 ,
      124.06428 ,
      124.06428 ,
      137.54747 ,
      152.69514 ,
      152.69514 ,
      169.52687 ,
      188.03511 ,
      208.18205 ,
      253.83586 ,
      279.20328 ,
      306.42593 ,
      335.33948 ,
      366.0146 ,
      432.837 ,

      usw., usw.

      230.14172 ,
      208.18178 ,
      188.03502 ,
      169.52702 ,
      152.6954 ,
      137.54745 ,
      124.064316 ,
      112.14457 ,
      101.76748 ,
      93.00298 ,
      85.68338 ,
      79.838615 ,
};
```

```
void setup()
{
  size(900, 600);
  background(255);
}

void draw()
{
  translate(0, 600);
  for (int i = 0; i < 435; i++) // Das Array enthält 436 Zahlenwerte
  {
    stroke(255, 0, 0);
    strokeWeight(2);
    line(2*i, -0.003*y[i], 1+2*i, -0.003*y[i+1]);
  }
}
```

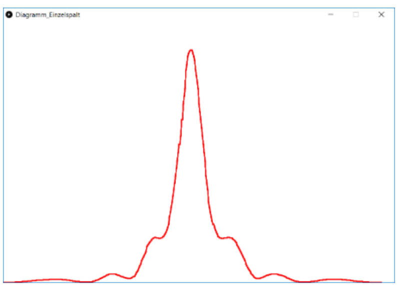


Abbildung 7.10: Wahrscheinlichkeitsverteilung für das Auftreffen eines Quantenobjektes hinter einem Einzelspalt, dargestellt mit dem Sketch „Diagramm_Einzelspalt"

Anstelle des obigen Sketches können wir natürlich auch etwas komfortabler ein Tabellenkalkulationsprogramm zur Darstellung des Graphen verwenden. Dann können wir auch die Wahrscheinlichkeit in Abhängigkeit von der Lage E.y des Empfängers darstellen. Am Beispiel von Excel 2016 soll dies nun kurz erläutert werden.

Daten aus der Processing-Konsole in Excel einfügen

Einige Daten in der Konsole mit dem Mauszeiger markieren. Dann *Strg + A* drücken, um alle Daten in der Konsole zu markieren. Anschließend mit *Strg + C* die Daten in die Zwischenablage kopieren. Nun Excel öffnen. Auf *Einfügen* klicken und die Daten mittels des *Textkonvertierungs-Assistenten* einfügen. Bei Schritt 3 von 3 des *Textkonvertierungs-Assistenten* auf *Weitere* klicken und bei *Dezimaltrennzeichen* einen Punkt und bei *1000er-Trennzeichen* ein Leerzeichen wählen.

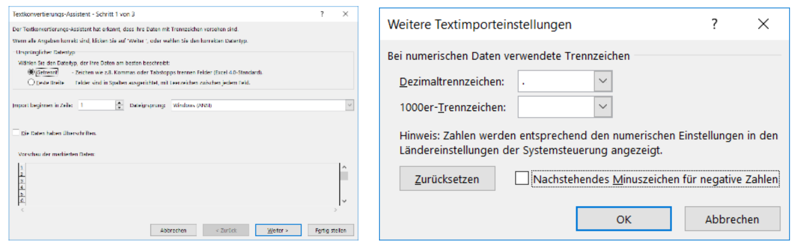


Abbildung 7.11: Textkonvertierungsassistent von Microsoft Office 2016

Nachdem man auf *Fertig stellen* geklickt hat, werden die Daten in das Excelblatt eingefügt. Nach dem Entfernen von unnötigen Stellen hinter dem Komma und der Formulierung von Spaltenüberschriften erhält man die folgende Tabelle und schließlich auch das gewünschte Diagramm.

	A	B
1	y in Pixel	Intensität in Pixel^2
2	5	2192
3	4	2160
4	4	2160
5	4	2160
6	4	2160
7	4	2160
8	4	2160
9	5	2192
10	6	2223

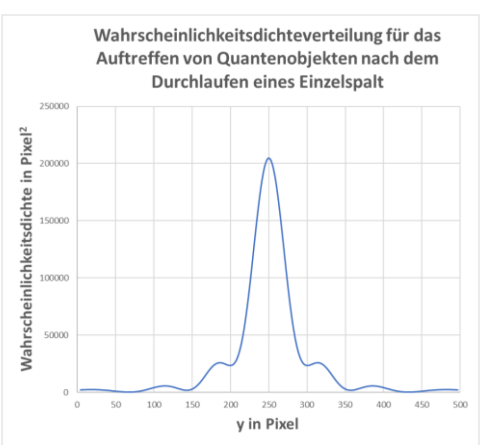


Abbildung 7.12: Tabellenausschnitt und fertiges Excel-Diagramm

7.3 Die vier sichtbaren Linien der Balmerserie des Wasserstoffs

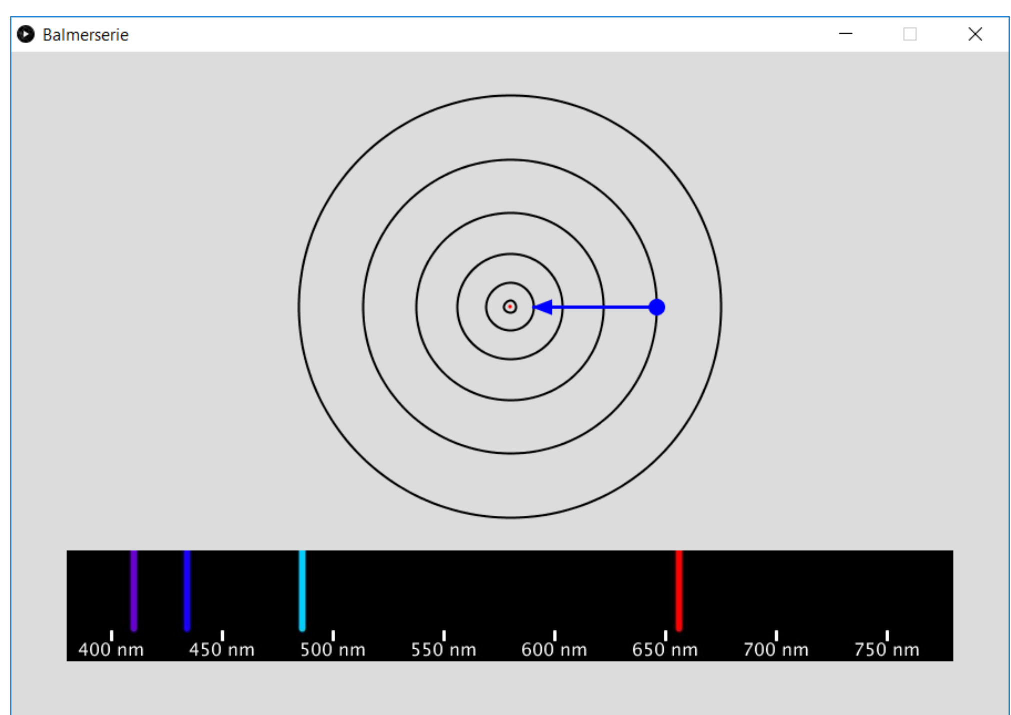


Abbildung 7.13: Erzeugung der 434 nm-Linie beim Sprung des Elektrons von der 5. auf die 2. Bahn.

Mittels des Bohrschen Atommodells lässt sich die Energieabgabe des Wasserstoffatoms und damit das zugehörige Linienspektrum sehr gut herleiten. Teilweise sichtbar sind die Linien der Balmer-Serie (siehe Abb. 7.13). Hierbei handelt es sich um Elektronensprünge von einer höher gelegenen Bahn auf die zweite Bahn. Wir wollen nun einen Sketch schreiben, der die obige Abbildung 7.13

erzeugt. Zusätzlich sollen die richtigen Bahnsprünge des Elektrons angezeigt werden, wenn man mit dem Mauszeiger auf die jeweilige Spektrallinie zeigt.

Die sichtbaren Linien der Balmer-Serie besitzen die folgenden Wellenlängen: 656 nm, 486 nm, 434 nm und 410 nm. Im Kapitel *Spektren* haben wir gelernt, wie man das zugehörige Emissionsspektrum mithilfe von Processing erzeugen kann. Ein solches Spektrum speichern wir mithilfe von *saveFrame()* als Bild ab und kopieren es anschließend in den Ordner unseres neuen Sketches. Nun müssen wir es mit ***PImage*** und einem frei wählbaren Namen, z.B. *Bild* deklarieren und mit ***loadImage*** in unseren Sketch laden. Anschließend legen wir noch fest, an welchem Platz im Fenster das Bild erscheinen soll. Wie unten dargestellt soll die Mitte des Bildes *HS_0001.jpg* bei x = 450 Pixel und y = 500 Pixel liegen.

```
Bild = loadImage("HS_0001.jpg");
imageMode(CENTER);
image(Bild, 450, 500);
```

Nachdem das Bild eingefügt ist, müssen wir noch die x-und y-Pixelwerte der einzelnen Spektrallinien ermitteln. Dies gelingt mit *println(mouseX, mouseY).* Befindet sich der Mauszeiger auf einer Spektrallinie, dann lesen wir die zugehörigen Pixelwerte in der Konsole ab. Die so ermittelten Werte benötigen wir, um mittels einer if-Anweisung den entsprechenden Elektronensprung anzuzeigen. Die Radien der Elektronenbahnen berechnet man mit $r_n = 5{,}293 \cdot 10^{-11} m \cdot n^2$. Wir benötigen jedoch einen für unsere Zeichnung praktikablen Kreisdurchmesser und wählen: $d_n = 2 \cdot 5{,}293 \cdot n^2$.

Der Rest des Sketches dürfte selbsterklärend sein.

Sketch 05: Balmerserie

```
// Die vier sichtbaren Linien des Wasserstoffs

PImage Bild;

void setup()
{
  size(900, 600);
}

void draw()
{
  background(220);

  // Bild des Wasserstoffspektrums wird eingefügt
  Bild = loadImage("HS_0001.jpg");
  imageMode(CENTER);
  image(Bild, 450, 500);
  println(mouseX, mouseY);

  // Atomkern und Elektronenbahnen werden gezeichnet
  noStroke();
  fill(255, 0, 0);
  ellipse(450, 230, 3, 3);

  stroke(0);
  strokeWeight(2);
  noFill();
```

```
  ellipse(450, 230, 10.6, 10.6);
  ellipse(450, 230, 42.4, 42.4);
  ellipse(450, 230, 95.2, 95.2);
  ellipse(450, 230, 169.7, 169.7);
  ellipse(450, 230, 264.6, 264.6);
  ellipse(450, 230, 381.0, 381.0);

  // Elektronensprünge werden gezeichnet
  if (mouseX >= 600 && mouseX <= 604 && mouseY >= 450 && mouseY <= 520)
  {
    stroke(0, 0, 255);
    strokeWeight(15);
    point(450+47.6, 230);
    strokeWeight(3);
    line(450+47.6, 230, 450+21.2, 230);
    fill(0, 0, 255);
    triangle(450+23, 230, 450+36, 225, 450+36, 235);
  }

  if (mouseX >= 259 && mouseX <= 263 && mouseY >= 450 && mouseY <= 520)
  {
    stroke(0, 0, 255);
    strokeWeight(15);
    point(450+84.7, 230);
    strokeWeight(3);
    line(450+84.7, 230, 450+21.2, 230);
    fill(0, 0, 255);
    triangle(450+23, 230, 450+36, 225, 450+36, 235);
  }

  if (mouseX >= 156 && mouseX <= 160 && mouseY >= 450 && mouseY <= 520)
  {
    stroke(0, 0, 255);
    strokeWeight(15);
    point(450+132.3, 230);
    strokeWeight(3);
    line(450+132.6, 230, 450+21.2, 230);
    fill(0, 0, 255);
    triangle(450+23, 230, 450+36, 225, 450+36, 235);
  }

  if (mouseX >= 107 && mouseX <= 111 && mouseY >= 450 && mouseY <= 520)
  {
    stroke(0, 0, 255);
    strokeWeight(15);
    point(450+190.5, 230);
    strokeWeight(3);
    line(450+190.5, 230, 450+21.2, 230);
    fill(0, 0, 255);
    triangle(450+23, 230, 450+36, 225, 450+36, 235);
  }
}
```

7.4 Orbitalmodelle des Wasserstoffs

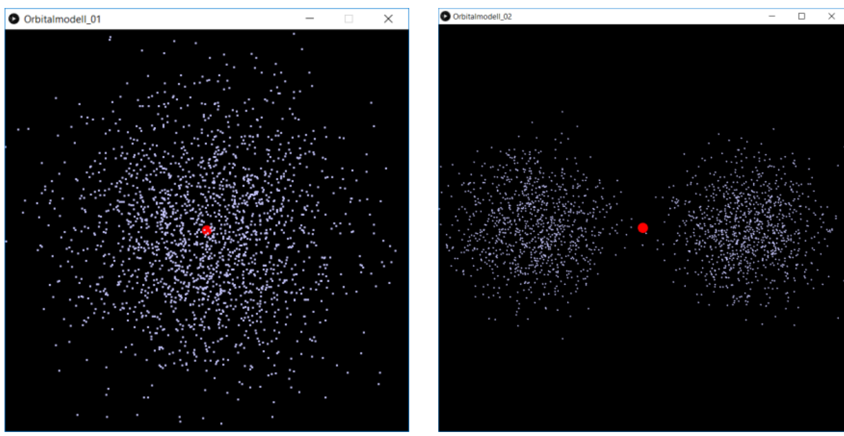


Abbildung 7.14: Orbitalmodelle: links n = 1, l = 0, m = 0 und rechts n = 2, l = 1, m = 0

Die Wellenfunktion für das Orbital mit den Quantenzahlen n = 1, l = 0 und m = 0 lautet:

$$\frac{1}{\sqrt{\pi}} \cdot (\frac{Z}{a_0})^{\frac{3}{2}} \cdot e^{-\frac{Z \cdot r}{a_0}}$$

Sie liefert uns eine kugelförmige Elektronenverteilung um den Atomkern. Die Wellenfunktion für das Orbital mit den Quantenzahlen n = 2, l = 1 und m = 0 lautet:

$$\frac{1}{4\sqrt{2\pi}} \cdot (\frac{Z}{a_0})^{\frac{3}{2}} \cdot \frac{Z \cdot r}{a_0} \cdot e^{-\frac{Z \cdot r}{2 \cdot a_0}} \cdot cos\theta$$

Sie liefert uns eine hantelförmige Elektronenverteilung.

Wenn man solche Gleichungen in Processing eingeben muss, dann kommt nicht unbedingt große Freude auf. Machen wir es uns deshalb etwas einfacher und lernen trotzdem etwas Neues.

Orbitalmodell 01

Das Orbital für die Quantenzahl-Kombination n = 1, l = 0 und m = 0 besitzt eine kugelförmige Wahrscheinlichkeitsverteilung für das Elektron des Wasserstoffatoms. Für die Darstellung der Wahrscheinlichkeitsverteilung benutzen wir dieses Mal nicht die Funktion *random(),* sondern die Funktion ***randomGaussian()***. *randomGaussian()* generiert nicht einfach eine Zahl zwischen einem Minimalwert und einem Maximalwert. *randomGaussian()* generiert Zahlen um den Mittelwert 0 mit einer Standardabweichung von 1. Zahlenwerte um den Mittelwert 0 besitzen eine höhere Wahrscheinlichkeit, generiert zu werden, als Zahlwerte, die weit vom Mittelwert 0 entfernt sind (siehe Abb. 7.15). Dies liefert in dem hier vorliegenden Fall bessere Wahrscheinlichkeitsverteilungen als die Funktion *random().*

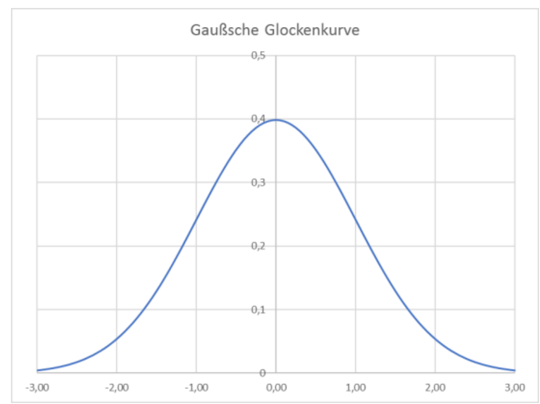


Abbildung 7.15: Normalverteilung der mit randomGaussian() generierten Werte

In unserem Sketch *Orbitamodell_01* setzen wir mit *translate()* den Ursprung des Koordinatensystems in die Mitte des Fensters. Damit erreichen wir, dass die Funktion *randomGaussian()* uns die Antreffwahrscheinlichkeit des Elektrons um den Atomkern generiert (siehe Abb. 7.14 links).

Die *for-Schleife* sorgt für eine sehr schnelle Darstellung der Aufenthaltsorte des Elektrons. Ohne die *for-Schleife* sieht man bei der Simulation nur einzelne Pünktchen aufblitzen. Einfach mal ausprobieren, um sich von der Wirksamkeit einer *for-Schleife* zu überzeugen.

Sketch 06: Orbitalmodell_01

```
// H-Atom Orbitalmodell für n = 1, l = 0, m = 0

float x;
float y;

void setup()
{
  size(600, 600);
}

void draw()
{
  background(0);
  translate(300, 300);

  // Atomkern wird gezeichnet
  noStroke();
  fill(255, 0, 0);
  ellipse(0, 0, 15, 15);

  // Antreffwahrscheinlichkeit des Elektrons
  // Die for-Schleife wird 2000mal durchlaufen
  for (int i = 0; i < 2000; i++)
  {
    x = 100*randomGaussian();
    y = 100*randomGaussian();

    stroke(200, 200, 255);
    strokeWeight(3);
    point(x, y);
  }
}
```

Orbitalmodell 02

Das Orbital für die Quantenzahl-Kombination n = 2, l = 1 und m = 0 besitzt eine hantelförmige Wahrscheinlichkeitsverteilung für das Elektron des Wasserstoffatoms (siehe Abb. 7.14 rechts). Dieses Orbital wollen wir nun dreidimensional darstellen und es soll zusätzlich noch mit der Maus um die x- und y-Achse drehbar sein. D.h., dieser Sketch wird bzgl. der Programmierung etwas anspruchsvoller sein als der recht einfache Sketch *Orbitalmodell_01*.

Auch bei diesem Sketch generiert die Funktion *randomGaussian()* uns Zahlwerte um den Mittelwert Null. Diese multiplieren wir mit dem Faktor 40 und verschieben in der dreifachen for-Schleife die Werte in positive und negative Richtung (siehe Sketch).

Anzumerken ist noch, dass wir bei *size()* P3D einfügen müssen, damit Processing weiß, dass es bei der Rotation um die x- bzw. y-Achse die entsprechende 3D-Darstellung berechnen muss. Um den Atomkern als dreidimensionale Kugel zu zeichnen, verwenden wir die Funktion *sphere()*. Würden wir die Funktion *ellipse()* verwenden, dann würden wir bei der Rotation eine Kreisscheibe in Seitenansicht sehen.

Sketch 07: Orbitalmodell_02

```
// H-Atom Orbitalmodell für n = 2, l = 1, m = 0

int a = 200; // Anzahl der Spalten
int b = 200; // Anzahl der Zeilen
int c = 200; // Anzahl der z-Werte

float x;
float y;
float z;

void setup()
{
  // P3D erlaubt die dreidimensionale Darstellung
  size(800, 800, P3D);
}

void draw()
{
  background(0);

  // Verschiebung des Koordinatenursprungs
  translate(400, 400);

  // Rotation um die x- und y-Achse
  rotateX(0.008*mouseY);
  rotateY(0.008*mouseX);

  // Atomkern wird gezeichnet
  noStroke();
  fill(255, 0, 0);
  sphere(10);

  // Berechnung der Aufenthaltsorte für die Elektronen
  for (int i = 0; i < a; i = i + 20)
  {
    for (int j = 0; j < b; j = j + 20)
    {
      for (int k = 0; k < c; k = k + 20)
```

```
        {
          x = 40 * randomGaussian() + i;
          y = 40 * randomGaussian() + j;
          z = 40 * randomGaussian() + k;

          stroke(200, 200, 255);
          strokeWeight(3);
          point(x+30, y+30, z+30);

          x = 40 * randomGaussian() - i;
          y = 40 * randomGaussian() - j;
          z = 40 * randomGaussian() - k;

          // Die Aufenthaltsorte des Elektrons werden gezeichnet
          stroke(200, 200, 255);
          strokeWeight(3);
          point(x-30, y-30, z-30);
        }
      }
    }
}
```

7.5 Zusammenfassung

random() Die Funktion *random()* stellt einen Zufallsgenerator dar. Enthält die Klammer nur einen Zahlenwert, so generiert die *random*-Funktion eine float-Zahl zwischen Null und diesem Zahlenwert. Stehen zwei Werte in der Klammer, so werden float-Zahlen zwischen diesen beiden Werten erzeugt.

randomGaussian() *randomGaussian()* generiert nicht einfach eine Zahl zwischen einem Minimalwert und einem Maximalwert. *randomGaussian()* generiert Zahlen um den Mittelwert 0 mit einer Standardabweichung von eins. Zahlenwerte um den Mittelwert 0 besitzen eine höhere Wahrscheinlichkeit, generiert zu werden, als Zahlwerte, die weit vom Mittelwert 0 entfernt sind. Diese Methode liefert für bestimmte Fälle bessere Wahrscheinlichkeitsverteilungen als die Funktion *random().*

PImage *PImage* ist ein Datentyp für die Speicherung von Bildern. Möchte man zum Beispiel das Bild einer Rakete in seinem Sketch verwenden, so schreibt man: *PImage* Rakete. Man verwendet also eine ähnliche Schreibweise wie *int* x. x wird so dem Datentyp der ganzen Zahlen zugeordnet und Rakete wird dem Datentyp Bilder zugeordnet. Ausführliche Informationen hierzu findet man in der Processing Referenz.

loadImage() Mit l*oadImage()* können Bilder der folgenden Formate .gif, .jpg, .tga und .png in eine Variable vom Typ *PImage* in einen Sketch geladen werden. Dazu müssen sich die Bilder aber im Ordner des jeweiligen Sketchs befinden.

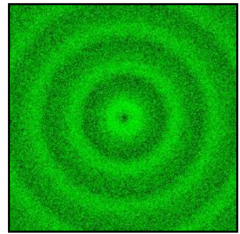
imageMode() Mit *imageMode()* können Bilder im Fenster verschoben, bzw. platziert werden. So kann zum Beispiel mit *imageMode(CENTER)* die Bildmitte auf vorgegebene x- und y-Werte gesetzt werden.

image() Mit image() kann man die Lage eines Bildes im Fenster festlegen (siehe Referenz).

while() In einer *for-Schleife*, ist die Anzahl der Schleifendurchläufe durch die in ihr aufgeführten Bedingungen festgelegt. Eine *while()-Schleife* wird dagegen solange durchlaufen, wie die in ihr formulierte Bedingung erfüllt ist. Dies ist auf der einen Seite sehr praktisch, wenn man im Vorfeld nicht weiß, wie viele Schleifendurchläufe man benötigt, um einen bestimmten Zustand zu erreichen. Auf der anderen Seite besteht aber auch die Gefahr, dass man eine **Endlosschleife** erhält, wenn man innerhalb der *while()-Schleife* keine Abbruchbedingung einfügt. Man kann eine *while()-Schleife* mit *break* verlassen.

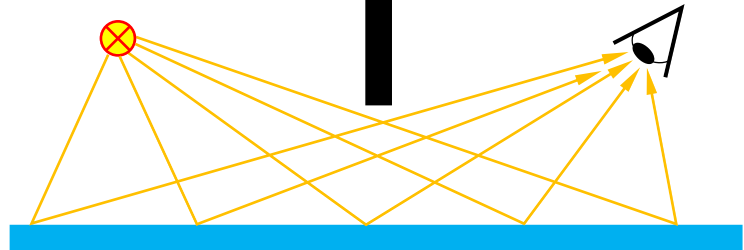
break Mit *break* kann man Kontrollstrukturen, wie *while(), for()* oder *switch()* verlassen.

7.6 Aufgaben



1. Mit einem stark abgeschwächten grünen Laser baut sich bei einem Interferenzversuch auf einer hochempfindlichen Fotoplatte nach und nach das rechts dargestellte Interferenzmuster Photon für Photon auf. Schreibe einen Sketch, der das rechst abgebildete Interferenzmuster erzeugt.

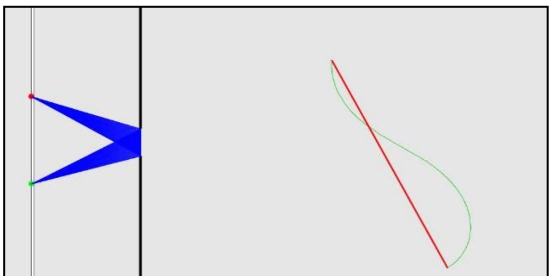
2. Den Zeigerformalismus von Richard Feynman haben wir im Sketch *Zeigerformalismus* auf einen Einzelspalt angewendet. Nun soll überprüft werden, ob der Zeigerformalismus auch bezüglich der Reflexionen von Photonen an einem Spiegel sinnvolle Ergebnisse liefert. Aufgrund unserer Kenntnisse aus der klassischen Optik erwarten wir, dass das Licht am Spiegel nach dem Reflexionsgesetz reflektiert wird. In der Quantenphysik geht man jedoch davon aus, dass die Photonen alle möglichen Wege gehen. Einige dieser Wege sind in der folgenden Abbildung eingetragen.



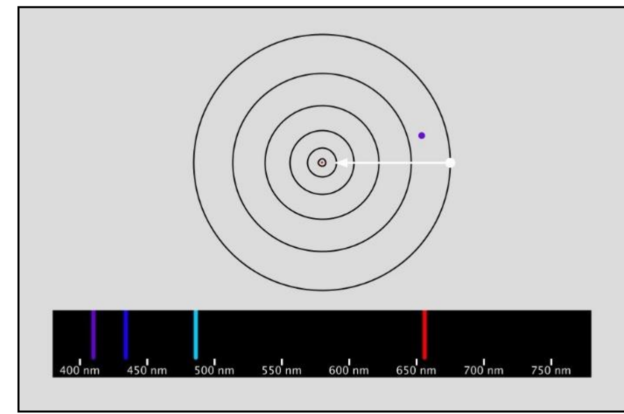
Wenn wir in der obigen Zeichnung das Auge durch eine Leinwand ersetzen, dann erwarten wir auf der Leinwand einen hellen Lichtfleck entsprechend dem Reflexionsgesetz. D.h., hier sollte

die Addition der einzelnen Pfeile ein Maximum ergeben. Überprüfe dies mithilfe eines Sketches.

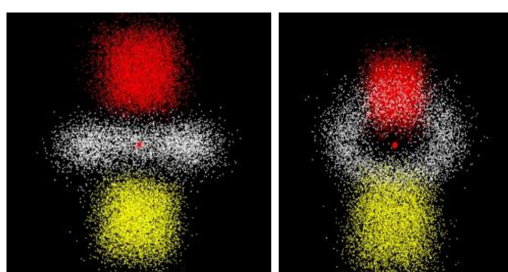
Tipp: Verwende im Sketch *Zeigerformalismus* den Spalt als Spiegel. Ändere dazu nur zwei Zeilen in der Funktion *setup().* Mehr ist nicht notwendig.



3. Erweitere den Sketch *Balmerserie* so, dass beim Bahnsprung eines Elektrons ein Photon mit der Farbe der Spektrallinie ausgesandt wird.



4. Die Abbildung rechts zeigt annähernd das Orbitalmodell für ein Wasserstoffatom mit den Quantenzahlen n = 3, l = 2 und m = 0. Schreibe in Anlehnung an den Sketch *Orbitalmodell_02* einen Sketch der die rechte Abbildung erzeugt. Die weiß dargestellte Wahrscheinlichkeitsdichte in den rechten Abbildungen hat die Form eines Torus. Informiere dich vor dem Beginn deiner Programmierung über die Eigenschaften eines Torus.



8 Ideale Gase und Festkörper

Was erwartet uns?

box(), return, ArrayList< >, get(), *=

8.1 Ideale Gase

8.1.1 Ideales Gas ohne Teilchenwechselwirkung

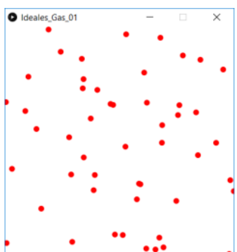


Abbildung 8.1: Ideales Gas ohne Teilchenwechselwirkung

Im Unterschied zu einem realen Gas wirken bei einem idealen Gas keine anziehenden Kräfte zwischen den Teilchen. Ein Stoßprozess zwischen zwei Teilchen kann hier mit einem Stoßprozess zwischen zwei Billardkugeln verglichen werden.

Fangen wir nun aber mal ganz einfach an. In unserem Sketch *Ideales_Gas_01* verzichten wir auf jegliche Wechselwirkung zwischen den einzelnen Teilchen. Nur mit den Wänden des Behälters führen die Teilchen vollkommen elastische Stöße aus. Im Sketch *Ideales_Gas_02* werden wir die Wechselwirkung zwischen den Teilchen dann berücksichtigen.

Widmen wir uns zuerst dem Nebensketch von *Ideales_Gas_01.* Hier erschaffen wir die Klasse Teilchen mit den Instanzvariablen Ortsvektor x, Geschwindigkeitsvektor v, Farbvektor c und d für den Teilchendurchmesser. Im Konstruktor bekommen die Vektoren und die Variable den Zusatz „Temp“, damit Processing weiß, ob die Größe aus der Klasse oder aus der jeweiligen Funktion gemeint ist. Im Hauptsketch werden dann für xTemp, vTemp, ... konkrete Werte eingegeben. Für die Funktionen in der Klasse verwenden wir die Instanzvariablen.

Die Reflexion eines Teilchens an einer Wand des Behälters gelingt, wenn man die entsprechende x- oder y-Komponente des Vektors v mit -1 multipliziert.

Im Hauptsketch erzeugen wir ein Array vom Datentyp *Teilchen* und geben ihm den Namen *teilchen*.

```
Teilchen[] teilchen
```

Welche Eigenschaften der Datentyp *Teilchen* besitzt, dies haben wir in unserem Nebensketch festgelegt. Vergleichbar ist dies mit *int x*. int ist der Datentyp und x der Name der Variablen. Nun müssen wir unser Array noch initialisieren:

```
Teilchen[] teilchen = new Teilchen[50];
```

Unser so initialisiertes Array kann 50 Teilchen aufnehmen (Nr. 0 bis Nr. 49). In den beiden *for-Schleifen* im Hauptsketch schreiben wir aber nicht i < 50, sondern i < teilchen.length. So können

wir ganz bequem bei *Teilchen[] teilchen = new Teilchen[50]* die Array-Größe in den eckigen Klammern ändern und in den *for-Schleifen* erfolgt eine automatische Anpassung.

Nun müssen wir noch den im Konstruktor erstellten Größen xTemp, vTemp, … konkrete Werte in der richtigen Reihenfolge zuordnen. Konkret heißt in unserem Fall, konkrete Zufallswerte. Die im Konstruktor festgelegte Reihenfolge ist: Ortsvektor, Geschwindigkeitsvektor, Teilchendurchmesser und Teilchenfarbe. Also: xTemp, vTemp, dTemp und cTemp.

teilchen[i] = new Teilchen(new PVector(random(0, width), random(0, height)), new PVector(random(-1, 1), random(-1, 1)), 10, new PVector(255, 0, 0));

Zum Schluss rufen wir für jedes Teilchen die folgenden drei Funktionen auf.

teilchen[i].move(1);

teilchen[i].wand();

teilchen[i].zeichnen();

Bei *teilchen[i].wand()* und *teilchen[i].zeichnen()* sind die runden Klammern leer. Bei *teilchen[i].move(1)* steht jedoch eine 1 in der Klammer, die die Zeitspanne pro Durchlauf angibt. Da wir im Nebensketch mit *void move(float t)* die Variable t eingeführt haben, müssen wir diese noch festlegen. Dies machen wir im Hauptsketch.

Sketch 01: Ideales_Gas_01

Hauptsketch

```
// Ideales Gas ohne Teilchenwechselwirkung

// Das Array vom Datentyp Teilchen mit dem Namen teilchen wird
// initialisiert
Teilchen[] teilchen = new Teilchen[50]; // Das Array kann 50 Teilchen
                                        // aufnehmen (Nr. 0 - Nr. 49)

void setup()
{
  size(400, 400);

  for (int i = 0; i < teilchen.length; i++)
  {
    // Nun werden die konkreten x- und y-Werte für xTemp, vTemp, dTemp,
    // und cTemp (color) für jedes Teilchen festgelegt
    teilchen[i] = new Teilchen(new PVector(random(0, width), random(0,
    height)), new PVector(random(-1, 1), random(-1, 1)), 10,
    new PVector(255, 0, 0));
  }
}

void draw()
{
  background(255);

  // Nun werden alle Teilchen mit den zugehörigen Funktionen aufgerufen
  for (int i = 0; i < teilchen.length; i++)
  {
    teilchen[i].move(1);
```

```
    teilchen[i].wand();
    teilchen[i].zeichnen();
  }
}
```

Nebensketch

```
// Die Klasse Teilchen wird deklariert
class Teilchen
{
  // INSTANZVARIABLEN
  PVector x, v, c; // Ortsvektor, Geschwindigkeitsvektor, Farbvektor
  float d; // Durchmesser der Teilchen

  // KONSTRUKTOR
  // Im Hauptsketch können für xTemp, vTemp, ... konkrete Werte
  // eingegeben werden
  Teilchen(PVector xTemp, PVector vTemp, float dTemp, PVector cTemp)
  {
    /* Nun werden die Instanzvariablen der Klasse Teilchen den Variablen
     der Funktionen zugeordnet, die im Konstruktor erstellt wurden */
    x = xTemp;
    v = vTemp;
    c = cTemp;
    d = dTemp;
  }

  // METHODEN
  // Funktion für die Teilchenbewegung
  // Die Größe von t wird im Hauptsketch festgelegt
  void move(float t)
  {
    // Zum aktuellen Wert des Vektors x wird der Wert v*t hinzuaddiert
    x.add(PVector.mult(v, t)); // v = konstant --> a = 0
  }

  // Funktion zum Zeichnen der Teilchen
  void zeichnen()
  {
    noStroke();
    fill(c.x, c.y, c.z); // Die Farbe ist hier eine vektorielle Größe
    ellipse(x.x, x.y, d, d); // Die x- und y-Komponenten des Vektors x
                             // geben den Ort eines Teilchens an
  }

  // Die Funktion wand sorgt für die Reflexion der Teilchen
  void wand()
  {
    if (x.x <= 0)
      v.x = -v.x;
    if (x.y <= 0)
      v.y = -v.y;
    if (x.x >= width)
      v.x = -v.x;
    if (x.y >= height)
      v.y = -v.y;
  }
}
```

8.1.2 Entropie

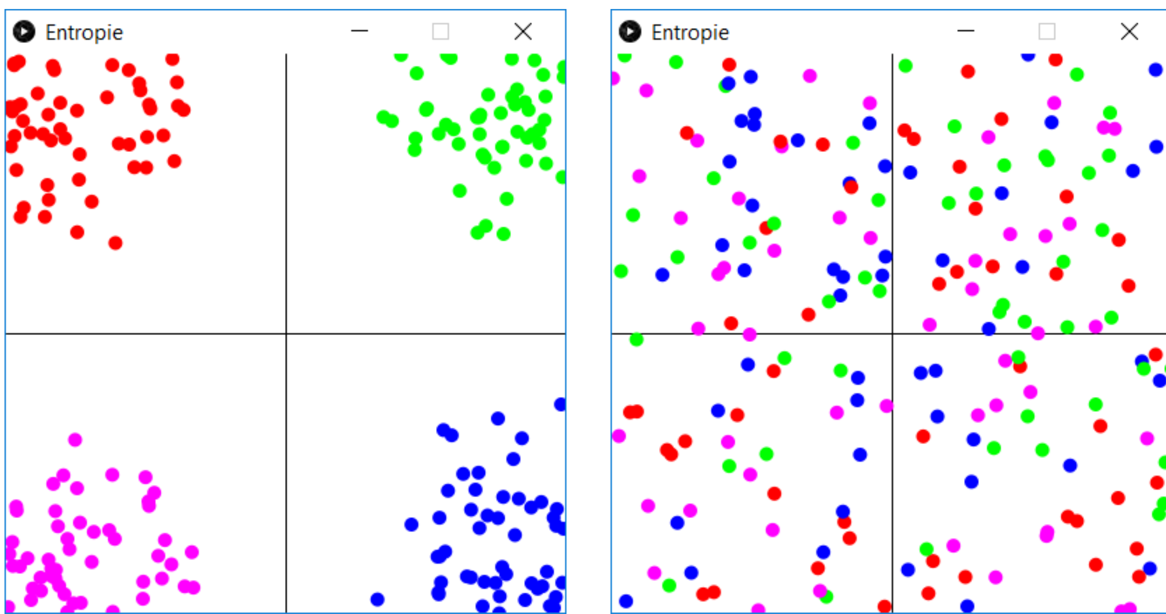


Abbildung 8.2: Vier Gase im geordneten Zustand (links) und im ungeordneten Zustand (rechts)

In unseren Sketch *Ideales_Gas_01* wurden 50 Teilchen per Zufallsgenerator generiert und mit den Eigenschaften Ort, Geschwindigkeit, Durchmesser und Farbe versehen. Nun wollen wir den Sketch so verändern, dass in jeder Ecke des Fensters farblich unterschiedliche Gasteilchen erzeugt werden (Abb. 8.2 links). Da wir den Sketch *Ideales_Gas_01* objektorientiert programmiert haben, gelingt diese Änderung ohne großen Aufwand. Wir müssen nur drei neue Arrays in den Hauptsketch einfügen. Am Nebensketch müssen wir keine Änderungen vornehmen. Alternativ kann man auch ein großes Array benutzen.

Wenn wir die Simulation starten, sehen wir schon nach kurzer Zeit, dass die vier idealen Gase sich vermischt haben (Abb. 8.2 rechts). Wir beobachten einen irreversiblen Prozess. Irreversibel heißt, dass es sehr, sehr, sehr unwahrscheinlich ist, dass unsere Simulation von selbst wieder ihren Ausgangszustand einnimmt. Die Wahrscheinlichkeit hierfür ist nicht Null, aber doch äußerst gering. Wir müssen wohl eine Ewigkeit warten, bis dies passiert. Aber für den Fall, dass es nun doch einmal geschehen sollte, kann man mit der folgenden Zeile die Simulation stoppen.

```
if (!(mousePressed && mouseButton == LEFT))
```

Diese Zeile kann man wie folgt übersetzen. Wenn die linke Maustaste nicht gedrückt ist, dann führe die Anweisungen in den nachfolgenden Zeilen aus. Solange die Maustaste gedrückt ist, wird die Simulation angehalten. Das Ausrufezeichen ! steht bei Processing für das logische NICHT.

Diese Simulation ist eine anschauliche Darstellung der Entropiezunahme, da das Teilchensystem von einem Zustand höherer Ordnung (geringe Wahrscheinlichkeit) in einen Zustand geringerer Ordnung (hohe Wahrscheinlichkeit) übergeht.

Sketch 02: Entropie

Hauptsketch

```
// Irreversibler Prozess

// Die Arrays vom Datentyp Teilchen mit dem Namen teilchen... werden
//initialisiert
Teilchen[] teilchenRot = new Teilchen[50]; // Das Array kann 50 rote
Teilchen aufnehmen
Teilchen[] teilchenBlau = new Teilchen[50]; // Das Array kann 50 blaue
Teilchen aufnehmen
Teilchen[] teilchenGruen = new Teilchen[50]; // Das Array kann 50 grüne
Teilchen aufnehmen
Teilchen[] teilchenViolett = new Teilchen[50]; // Das Array kann 50
violette Teilchen aufnehmen

void setup()
{
  size(400, 400);

  for (int i = 0; i < 50; i++)
  {
    // Nun werden die konkreten x- und y-Werte für xTemp, vTemp, dTemp,
    // und cTemp (color) für jedes Teilchen festgelegt
    teilchenRot[i] = new Teilchen(new PVector(random(0, 100), random(0,
    100)), new PVector(random(-1, 1), random(-1, 1)), 10, new
PVector(255,
    0, 0));
    teilchenBlau[i] = new Teilchen(new PVector(random(300, 400),
    random(300, 400)), new PVector(random(-1, 1), random(-1, 1)), 10,
    new PVector(0, 0, 255));
    teilchenGruen[i] = new Teilchen(new PVector(random(300, 400),
random(0,
    100)), new PVector(random(-1, 1), random(-1, 1)), 10, new PVector(0,
    255, 0));
    teilchenViolett[i] = new Teilchen(new PVector(random(0, 100),
    random(300, 400)), new PVector(random(-1, 1), random(-1, 1)), 10,
    new PVector(255, 0, 255));
  }
}

void draw()
{
  background(255);
  stroke(0);
  line(0, height/2, width, height/2);
  line(width/2, 0, width/2, height);

  // Nun werden alle Teilchen mit den zugehörigen Funktionen versehen
  /* Die folgenden zwei Funktionen werden nur ausgeführt, wenn die
     linke Maustaste nicht gedrückt ist */
  if (!(mousePressed && mouseButton == LEFT))
  {
    for (int i = 0; i < 50; i++)
    {
      teilchenRot[i].move(1);
      teilchenRot[i].wand();

      teilchenBlau[i].move(1);
      teilchenBlau[i].wand();
```

```
      teilchenGruen[i].move(1);
      teilchenGruen[i].wand();

      teilchenViolett[i].move(1);
      teilchenViolett[i].wand();
    }
  }

  // Diese Funktion wird immer ausgeführt
  for (int i = 0; i < 50; i++)
  {
    teilchenRot[i].zeichnen();
    teilchenBlau[i].zeichnen();
    teilchenGruen[i].zeichnen();
    teilchenViolett[i].zeichnen();
  }
}
```

Nebensketch

```
// Die Klasse Teilchen wird deklariert
class Teilchen
{
  // INSTANZVARIABLEN
  PVector x, v, c; // Ortsvektor, Geschwindigkeitsvektor, Farbvektor
  float d; // Durchmesser der Teilchen

  // KONSTRUKTOR
  // Im Hauptsketch können für xTemp, vTemp, ... konkrete Werte
  // eingegeben werden
  Teilchen(PVector xTemp, PVector vTemp, float dTemp, PVector cTemp)

  {
    /* Nun werden die Instanzvariablen der Klasse Teilchen den Variablen
       der Funktionen zugeordnet, die im Konstruktor erstellt wurden */
    x = xTemp;
    v = vTemp;
    c = cTemp;
    d = dTemp;
  }

  // METHODEN
  // Funktion für die Teilchenbewegung
  // Die Größe von t wird im Hauptsketch festgelegt
  void move(float t)
  {
    // Zum aktuellen Wert des Vektors x wird der Wert v*t hinzuaddiert
    x.add(PVector.mult(v, t)); // v = konstant --> a = 0
  }

  // Funktion zum Zeichnen der Teilchen
  void zeichnen()
  {
    noStroke();
    fill(c.x, c.y, c.z); // Die Farbe ist hier eine vektorielle Größe
    ellipse(x.x, x.y, d, d); /* Die x- und y-Komponenten des Vektors x
     geben den Ort eines Teilchens an */
  }

  // Die Funktion wand sorgt für die Reflexion der Teilchen
```

```
  void wand()
  {
    if (x.x <= 0)
      v.x = -v.x;
    if (x.y <= 0)
      v.y = -v.y;
    if (x.x >= width)
      v.x = -v.x;
    if (x.y >= height)
      v.y = -v.y;
  }
}
```

8.1.3 Ideales Gas mit Wechselwirkungen

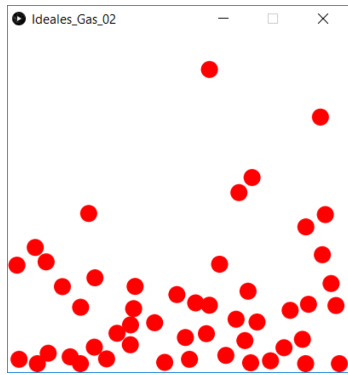


Abbildung 8.3: Ideales Gas mit Wechselwirkungen

Bei unserem neuen Sketch *Ideales_Gas_02* sollen die einzelnen Gasteilchen nun miteinander wechselwirken können. Dies gelingt mit der Funktion *void collision(Teilchen kollisionsPartner)* im Nebensketch. Zusätzlich sollen die Gasteilchen der Anziehungskraft der Erde unterliegen und weiterhin soll man das Gas abkühlen und aufheizen können.

Im Hauptsketch können wir die folgenden Einstellungen vornehmen. Mit *PVector g = new PVector(0, 1)* sorgen wir dafür, dass die Erdbeschleunigung nur in y-Richtung wirkt. Die Größe 1 für die Erdbeschleunigung hat sich für unsere Pixelwelt durch Ausprobieren als sinnvoll erwiesen. Bei *teilchen[i].move(0.1, g)* können wir die Geschwindigkeit der Simulation einstellen. Aber Vorsicht! Stellen wir die Geschwindigkeit zu hoch ein, dann „explodiert" das Gas. Welchen Energieverlust die Teilchen bei der Reflexion an einer Wand erfahren, bestimmen wir mit *teilchen[i].wand(0.5).* Der Wert 0.5 bedeutet, dass die Gasteilchen hier 50% ihrer Energie verlieren.

Weitere Erläuterungen, insbesondere zu den einzelnen Funktionen findet man im Sketch selbst.

Sketch 03: Ideales_Gas_02

Hauptsketch

```
// Ideales Gas 02 mit Teilchenwechselwirkung, Erdbeschleunigung und
// Energieregulierung
```

```
// Das Array vom Datentyp Teilchen mit dem Namen teilchen wird
// initialisiert
Teilchen[] teilchen = new Teilchen[50]; // Das Array kann 50 Teilchen
                                        // aufnehmen (Nr. 0 - Nr. 49)

PVector g = new PVector(0, 1); // Die Erdbeschleunigung findet nur in
                               // y-Richtung statt

void setup()
{
  size(400, 400);

  for (int i = 0; i < teilchen.length; i++)
  {
    // Nun werden die konkreten x- und y-Werte für xTemp, vTemp, dTemp,
    // cTemp (color) und aTemp für jedes Teilchen festgelegt
    teilchen[i] = new Teilchen(new PVector(random(0, width), random(0,
    height)), new PVector(random(-5, 5), random(-5, 5)), 10,
    new PVector(255, 0, 0), new PVector(0, 0));
  }
}

void draw()
{
  background(255);

  // Nun werden alle Teilchen mit den zugehörigen Funktionen aufgerufen
  for (int i = 0; i < teilchen.length; i++)
  {
    for (int j = 0; j < teilchen.length; j++)
    {
      if (i==j)
        continue; /* Damit die Teilchen nicht mit sich selber
        wechselwirken, wird ein Schleifendurchlauf übersprungen */

      // Prüft für das i-te Teilchen, ob eine Kollision mit dem j-ten
      // Teilchen vorhanden ist
      teilchen[i].collision(teilchen[j]);
    }
  }

  for (int i = 0; i < teilchen.length; i++)
  {
    teilchen[i].move(0.1, g);
    teilchen[i].wand(0.5);
    teilchen[i].zeichnen();
  }
}
```

Nebensketch

```
// Die Klasse Teilchen wird deklariert
class Teilchen
{
  // INSTANZVARIABLEN
  PVector x, v, c, a;
  float r; // Teilchenradius
  float m = 1; // Teilchenmasse
```

```
// KONSTRUKTOR
// Im Hauptsketch können für xTemp, vTemp, ... konkrete Werte
   eingegeben werden
Teilchen(PVector xTemp, PVector vTemp, float rTemp, PVector cTemp,
         PVector aTemp)
{
  // Nun werden die Instanzvariablen der Klasse Teilchen den Variablen
  // der Funktionen zugeordnet, die im Konstruktor erstellt wurden
  x = xTemp;
  v = vTemp;
  r = rTemp;
  c = cTemp;
  a = aTemp;
}

// METHODEN
// Funktion für die Teilchenbewegung
void move(float t, PVector g) //Funktion
{
  a.add(g);
  v.add(PVector.mult(a, t)); // a bleibt bei dieser Rechnung konstant
  x.add(PVector.mult(v, t)); // v bleibt bei dieser Rechnung konstant
  a.mult(0); // a wird mit 0 multipliziert, damit a nicht immer weiter
             // anwächst
}

// Funktion zum Zeichnen der Teilchen
void zeichnen()
{
  noStroke();
  fill(c.x, c.y, c.z);
  ellipse(x.x, x.y, r * 2, r * 2);
}

// Funktion für die Teilchenkollision
void collision(Teilchen kollisionsPartner)
{
  PVector normal = PVector.sub(kollisionsPartner.x, x); // Der
  // vektorielle Abstand zwischen zwei Teilchen wird berechnet

  float distance = normal.mag(); // Der Betrag des Vektors "normal"
                                 // wird „distance" genannt

  // Wenn sich die Teilchen nicht berühren, dann wirkt keine Kraft auf
  // sie
  if (distance >= r + kollisionsPartner.r)
    return; // Die Funktion sofort verlassen

  normal.div(distance); // Erstellung des Einheitsvektors

  // Zu a wird ein negativer Wert hinzuaddiert (a = F/m)
  PVector force = PVector.mult(normal, 10 * (distance -
  (r + kollisionsPartner.r)));
  a.add(PVector.div(force, m));
}

// Funktion für die Teilchenreflexion an der Wand
// Im Hauptsketch kann man mit dem Wert für E die Energieänderung bei
// der Teilchenreflexion an der Wand einstellen
// Also abkühlen E < 1 oder heizen E > 1
void wand(float E)
```

```
  {
    if (x.x <= r)
    {
      v.x = -v.x * E;
      x.x = r;
    }
    if (x.y <= r)
    {
      v.y = -v.y * E;
      x.y = r;
    }
    if (x.x >= width - r)
    {
      v.x = -v.x * E;
      x.x = width - r; // So dringen die Teilchen nicht in die Wände ein
    }
    if (x.y >= height - r)
    {
      v.y = -v.y * E;
      x.y = height - r; // So sinken die Teilchen nicht unter den
                        // unteren Rand
    }
  }
}
```

8.2 Festkörper

8.2.1 Festkörper 01

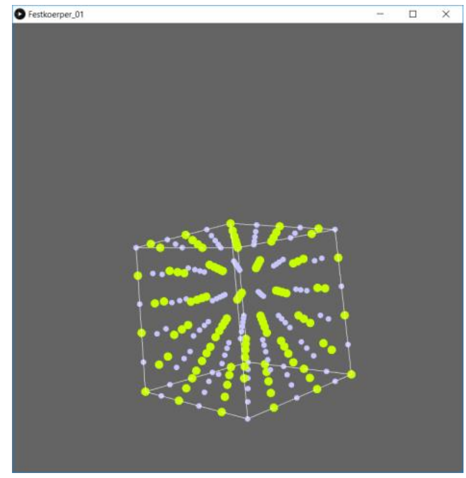


Abbildung 8.4: NaCl-Kristall

Wenn wir nur ein einfaches dreidimensionales Kristallgitter, zum Beispiel NaCl zeichnen wollen, dann können wir auf den Sketch *Orbitalmodell_02* zurückgreifen. Wir verzichten in unserem neuen Sketch jedoch auf die Zufallsfunktion *randomGaussian()*. i, j und k sind dann die Koordinaten der Atome, die wir durch farbige Punkte darstellen. So zeichnet uns Processing dann einen einfachen Festkörper, den wir mit der Maus im 3D-Raum drehen können.

Erwähnenswert in dem folgenden Sketch ist die Zeile mit dem *Modulo Operator %.*

if ((i + j + k) % 2 == 0)

Diese Zeile sorgt dafür, dass abwechselnd Chlor- und Natriumatome gezeichnet werden.

Damit wir auch bei diesem einfachen Sketch etwas Neues lernen, umgeben wir die regelmäßige Anordnung der Atome mit den transparenten Oberflächen eines Würfels. Dies gelingt mit der Funktion ***box().*** Geben wir in die runde Klammer nur eine Zahl ein, so zeichnet Processing einen dreidimensionalen Würfel mit den entsprechenden Kantenlängen. Benötigt man eine rechteckige Box, dann gibt man drei Werte in die runde Klammer ein. Mit diesen Werten bestimmt man die x-, y- und z-Ausdehnung der Box.

Sketch 04: Festkoerper_01

```
// NaCl-Kristallgitter 3D

int a = 6;
int b = 6;
int c = 6;

void setup()
{
  size(800, 800, P3D);
}

void draw()
{
  background(100);
  translate(400, 400);
  rotateY(0.008*mouseX);
  rotateX(0.008*mouseY);

  for (int i = 0; i < a; i ++)
  {
    for (int j = 0; j < b; j ++)
    {
      for (int k = 0; k < c; k ++)
      {
        /* Mit dem Modulo-Operator % sorgen wir dafür, dass
         abwechselnd Chlor- und Natriumionen gezeichnet werden */
        if ((i + j + k) % 2 == 0)
        {
          stroke(200, 255, 0); // Chlorionen hellgrün
          strokeWeight(15);
        } else
        {
          stroke(200, 200, 255); // Natriumionen hellblau
          strokeWeight(10);
        }
        point(i * 40, j * 40, k * 40);
      }
    }
  }

  // Eine Box wird um den Festkörper gezeichnet
  translate(100, 100, 100);
  stroke(255);
  strokeWeight(1);
  noFill();
  box(200);
}
```

8.2.2 Lennard-Jones-Potenzial

Veranschaulichung

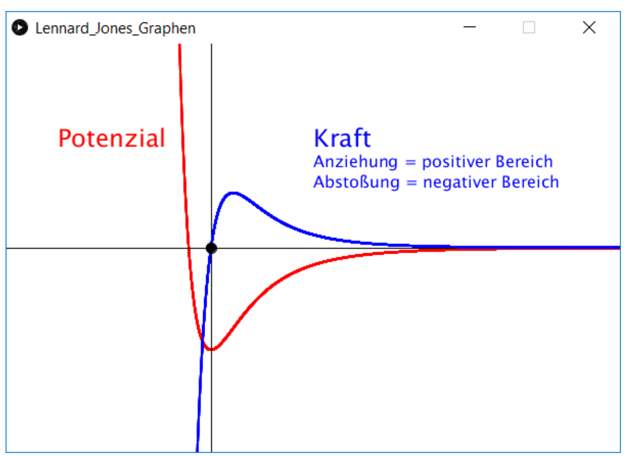


Abbildung 8.5: Lennard-Jones-Potenzial und die hieraus folgende Kraft

Während wir in vorhergehenden Kapitel nur einen einfachen Festkörper gezeichnet haben, wollen wir nun etwas tiefer in den physikalischen Sachverhalt einsteigen. Auf den ersten Blick ist es doch verwunderlich, das neutrale Atome sich zu einem festen Körper zusammenschließen können.

In der Schule lernt man, dass Atome und Moleküle nach außen hin elektrisch neutral sind. Dies stimmt nur, wenn sie sich in einem für sie relativ großen Abstand von zum Beispiel einigen Millimeter befinden. Im Nahbereich ziehen sie sich aufgrund ihrer Ladungsverteilung zuerst gegenseitig an und erfahren, wenn sie sich allzu nahekommen, eine starke abstoßende Kraft. In einem bestimmten Abstand heben sich die anziehende und die abstoßende Kraft auf. Beschrieben wird dies durch das Lennard-Jones-Potenzial. Nun ist ein Potenzial aber keine Kraft. Wie Potenzial und Kraft zusammenhängen, verdeutlichen wir uns mal kurz an dem uns bekannten elektrischen Potenzial φ(r). Dazu müssen wir auch einmal differenzieren.

Vergessen, wie man differenziert? Für einfache Funktionen kann man sich diese simple Regel merken: Die Funktion mit dem Exponenten der Variablen multiplizieren und dann vom Exponenten der Variablen den Wert 1 abziehen. Also: $-\frac{1}{r} = -r^{-1} \Rightarrow r^{-2} = \frac{1}{r^2}$

Wenn man das Potenzial φ(r) einer felderzeugenden Ladung Q nach r differenziert, erhält man die Feldstärke E(r). Wenn man die Feldstärke E(r) mit der Ladung q multipliziert, erhält man die Kraft F(r), die Q auf q ausübt.

$$\varphi(r) = -\frac{1}{4\pi\varepsilon_0\varepsilon_r} \cdot \frac{Q}{r} \qquad \frac{d\varphi}{dr} = E(r) = \frac{1}{4\pi\varepsilon_0\varepsilon_r} \cdot \frac{Q}{r^2} \qquad F(r) = \frac{1}{4\pi\varepsilon_0\varepsilon_r} \cdot \frac{Q \cdot q}{r^2}$$

Diese Überlegungen kann man auf das Lennard-Jones-Potenzial übertragen. Dazu benötigen wir aber eine entsprechende Gleichung. Ein Blick in Wikipedia hilft uns hierbei. Eine für unseren Zweck gut handzuhabende Formel ist die folgende Gleichung für das Lennard-Jones-(12, 6)-Potenzial.

$$V(r) = \varepsilon \cdot \left[\left(\frac{r_m}{r}\right)^{12} - 2\left(\frac{r_m}{r}\right)^{6}\right] \qquad \frac{dV(r)}{dr} = \varepsilon \cdot \left(-12 \cdot \frac{{r_m}^{12}}{r^{13}} + 12\frac{{r_m}^{6}}{r^{7}}\right) \qquad \mathrm{F(r)} = \varepsilon \cdot \left(-12 \cdot \frac{{r_m}^{12}}{r^{13}} + 12\frac{{r_m}^{6}}{r^{7}}\right) \cdot q$$

Der Term $-12\frac{{r_m}^{12}}{r^{13}}$ beschreibt die abstoßende Kraft und der Term $+12 \cdot \frac{{r_m}^{6}}{r^{7}}$ beschreibt die anziehende Kraft, wenn man die Gleichung $\frac{dV(r)}{dr}$ mit q multipliziert. Für den Teilchenabstand $r = r_m$

nimmt der Wert in der Klammer und damit auch F(r) den Wert Null an. Für Abstände $r < r_m$ wird der Bruch $\frac{r_m{}^{12}}{r^{13}}$ deutlich schneller größer als der Bruch $\frac{r_m{}^{6}}{r^{7}}$. D.h., für $r < r_m$ dominiert die abstoßende Kraft. Für $r > r_m$ dominiert die anziehende Kraft. Für große Entfernungen streben beide Terme gegen Null und somit wirkt auf ein weit entferntes Teilchen keine Kraft.

Den Ort, an dem sich beide Kräfte aufheben, legen wir in unserer Pixelwelt mit $r_m = 200$ fest. Dies ist auch der Ort, an dem die eckige Klammer in der Potenzialfunktion V(r) den Wert -1 annimmt. Dies ist der kleinste Wert, den die eckige Klammer annehmen kann. Hier besitzt die Potenzialfunktion ihren Tiefpunkt. Mit ε können wir diesen Tiefpunkt verschieben. Setzen wir in unserer Pixelwelt $\varepsilon = 100$, sodass sich ein gut sichtbarer Potenzialverlauf ergibt. Für die Ladung wählen wir den Wert $q = 40$.

Bevor wir nun eine entsprechende Animation für die Teilchenbewegung programmieren, veranschaulichen wir uns zuerst mithilfe eines Sketches den Verlauf von V(r) und F(r) in einem Diagramm (siehe Abb. 8.5).

Mit dem Sketch *Lennard_Jonnes_Graphen* wurde der Potenzialverlauf (rot) und der Kräfteverlauf (blau) grafisch dargestellt. Dort, wo der Potenzialverlauf ein Minimum hat ($r = r_m = 200$), ist die Ableitung dieser Funktion Null und damit ist auch die Kraft auf ein Teilchen (In Abbildung 8.5 schwarz dargestellt) gleich Null.

Widmen wir uns nun dem unten aufgeführten Sketch *Lennard_Jonnes_Graphen*, mit dem das obige Diagramm erstellt worden ist. Wichtig ist, dass man bei der Eingabe der Gleichungen daran denkt, dass in Processing die y-Achse nach unten zeigt. D.h., man muss vor die Gleichungen für das Potenzial und für die Kraft ein Minuszeichen setzen, damit man eine vertraute grafische Darstellung erhält. Also:

```
V = -epsilon * (pow(rm,12)/pow(r,12) - 2 * pow(rm,6)/pow(r,6));

F = -q * epsilon * (-12*pow(rm,12)/pow(r,13) + 12 * pow(rm,6)/pow(r,7));
```

Die unten aufgeführten Programmschritte in unserem statischen Programm ohne *void setup()* und *void draw()* sollten uns vertraut sein. Vielleicht sollte nochmal kurz erwähnt werden, was pow(r,13) bedeutet. Ganz einfach, pow(r,13) steht für r^{13}.

Sketch 05: Lennard_Jones_Graphen

```
// Lennard-Jones-Potenzial und Kraft

float V; // Potenzial
float F; // Kraft
float rm = 200;
float q = 40; // Ladung
float epsilon = 100;

translate(0, 200);
size (600, 400);
background(255);

// Das Lennard-Jones-Potenzial wird rot gezeichnet
for (float r = 40; r < 600; r = r + 0.01)
{
  V = -epsilon * (pow(rm, 12) / pow(r, 12) - 2 * pow(rm, 6) / pow(r, 6));
  noStroke();
  fill(255, 0, 0);
```

```
  ellipse(r, V, 2, 2);
  stroke(0);
  line(0, 0, 600, 0);
  line(200, -300, 200, 300); // y-Achse bei V = Minimum und F = 0
}

// Der Kräfteverlauf nach Lennard-Jones wird blau gezeichnet
for (float r = 40; r < 600; r = r + 0.01)
{
  F = -q * epsilon * (-12*pow(rm, 12) / pow(r, 13) + 12 * pow(rm, 6) /
  pow(r, 7));
  noStroke();
  fill(0, 0, 255);
  ellipse(r, F, 2, 2);
  stroke(0);
  line(0, 0, 400, 0); // x-Achse
}

// Schwarzes Teilchen bei F = 0
fill(0);
ellipse(rm, 0, 10, 10);

// Beschriftung
textSize(24);
fill(255, 0, 0);
text("Potenzial", 50, -100);

textSize(24);
fill(0, 0, 255);
text("Kraft", 300, -100);
textSize(16);
text("Anziehung = positiver Bereich", 300, -80);
text("Abstoßung = negativer Bereich", 300, -60);
```

Simulation

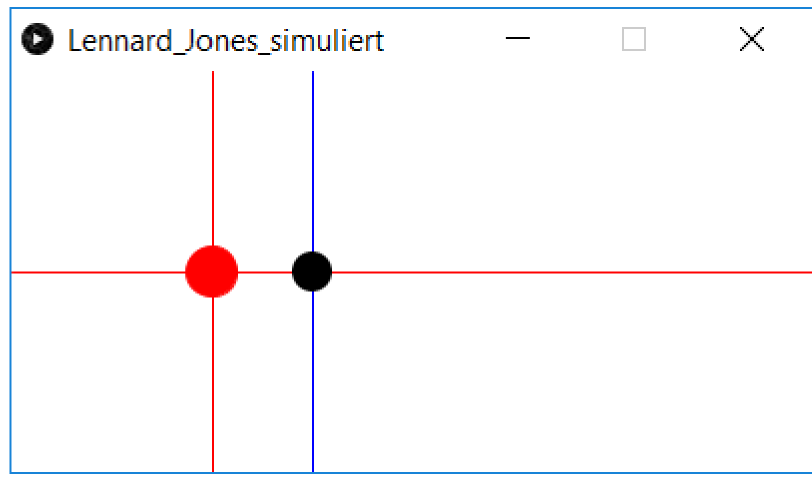


Abbildung 8.6: Simulation einer Teilchenbewegung im Lennard-Jones-Potenzial

Nachdem wir die Gleichungen und die Graphen für das Lennard-Jones-Potenzial und die daraus sich ergebende Kraft kennengelernt haben, können wir nun versuchen, die Annäherung zwischen zwei Teilchen zu simulieren. Jetzt können wir natürlich nicht mehr im statischen Modus arbeiten,

sondern müssen den ganzen Vorgang vektoriell im dynamischen Modus darstellen. Doch keine Panik! Wir können bei unserer Arbeit auf den Sketch *Gravitationsgesetz_01* zurückgreifen, mit dem wir die Bewegung eines Planeten um seine Sonne simuliert haben. Die im Ursprung ruhende Sonne ersetzen wir durch ein Teilchen, welches wir „Felderzeuger" nennen. Den Planeten ersetzen wir durch ein Teilchen mit dem Namen „Teilchen". Anstelle des Gravitationsgesetzes setzen wir die Gleichung für die Kraft ein, die wir aus dem Lennard-Jones-Potenzial gewonnen haben. Da wir keinen Graphen zeichnen, sondern die Bewegung des Teilchens simulieren wollen, können wir auf das Minuszeichen vor der Gleichung verzichten. Weiterhin soll das Teilchen, im Gegensatz zu dem Planeten, keine Anfangsgeschwindigkeit besitzen und zudem eine Dämpfung erfahren, damit es im Abstand r_m zur Ruhe kommen kann.

Welche Zahlenwerte nun für r_m, ε und q geeignete Werte sind, dies müssen wir ausprobieren. Sehr hilfreich ist, wenn man sich in der Konsole z. B. die Werte für d und aB (siehe Sketch unten) anzeigen lässt. Die Dämpfung realisieren wir, indem wir die Geschwindigkeit v bei jedem Durchlauf mit einem negativen Wert, im Beispielsketch mit dem Wert -0.01, multiplizieren. So erhalten wir eine dämpfende, der Bewegung entgegengerichtete Beschleunigung.

Sketch 06: Lennard_Jones_simuliert

Hauptsketch

```
// Lennard Jones simuliert

Lennard L; // Die Klasse Lennard bekommt den Namen L

void setup()
{
  size(400, 200);
  L = new Lennard(); // Die Klasse Lennard wird aufgerufen
}

void draw()
{
  background(255);

  // Die Funktionen aus der Klasse Lennard werden aufgerufen
  L.move();
  L.display();
}
```

Nebensketch

```
class Lennard
{
  // INSTANZVARIABLEN
  float t = 1;
  float q = 5;
  float epsilon = 1;
  float rm = 50;

  // KONSTRUKTOR
  PVector Felderzeuger = new PVector(0, 100);
  PVector Teilchen = new PVector(120, 100);
  PVector v = new PVector(0, 0);
```

```
  // METHODEN
  void move()
  {
    // Verschiebung des Koordinantenursprungs
    translate(100, 0);

    // Abstand d zwischen Felderzeuger und Teilchen wird berechnet
    float d = Felderzeuger.dist(Teilchen);

    // Der Betrag der Beschleunigung wird berechnet
    float aB = q*epsilon * (-12*pow(rm, 12)*pow(d, -13) + 12 * pow(rm, 6) * pow(d, -7));

   /* Zuerst wird die vektorielle Differenz zwischen Felderzeuger und
      Teilchen berechnet, dann auf 1 normalisiert und anschließend mit
      dem Betrag aB der Beschleunigung multipliziert */
    PVector a = PVector.sub(Felderzeuger,Teilchen).normalize().mult(aB);

    // Dämpfung durch entgegengesetzte Beschleunigung
    a.add(PVector.mult(v, -0.01));

    // a wird mit t multipliziert und zum Vektor v addiert
    v.add(PVector.mult(a, t));

    // v wird mit t multipliziert und zum Ortsvektor Teilchen addiert
    Teilchen.add(PVector.mult(v, t));
  }

  void display()
  {
    // Koordinatensystem mit im Ursprung ruhendem Felderzeuger
    stroke(255, 0, 0);
    line(0, 0, 0, 200);
    line(-100, 100, 300, 100);
    fill(255, 0, 0);
    ellipse(Felderzeuger.x, Felderzeuger.y, 25, 25);

    // Die blaue Linie zeigt die Ruheposition des Teilchens an
    stroke(0, 0, 255);
    line(rm, 0, rm, 400);

    fill(0); // Teilchen
    ellipse(Teilchen.x, Teilchen.y, 20, 20);
  }
}
```

8.2.3 Festkörper 02

Nachdem wir den Sketch für die Wechselwirkung zweier Teilchen nach dem Lennard-Jones-Potenzial erfolgreich programmiert haben, wollen wir nun versuchen, die Wechselwirkung zwischen einer größeren Anzahl von Teilchen zu simulieren. Für den Namen Teilchen wollen wir in unserem neuen Sketch jedoch den Namen *Atom* verwenden, da man bei *Atom* zwischen Einzahl und Mehrzahl unterscheiden kann. Dies gelingt beim Wort Teilchen nicht.

Um die Wechselwirkung von vielen Atomen zu simulieren, erstellen wir am besten ein Array. Wie man ein Array für eine Vielzahl von Objekten anfertigt, dies haben wir schon öfters kennengelernt. In diesem Sketch wollen wir aber ein dynamisches Array erstellen. D.h., wir wollen uns offenhalten, wie viele Atome wir bei unserer Simulation verwenden. Dies hängt schließlich auch

von der Rechenleistung unseres Computers ab. Hierzu findet man in der Processing-Referenz den Ausdruck **ArrayList**. In unserem Sketch (siehe unten) schreiben wir:

```
ArrayList<Lennard> Atome = new ArrayList<Lennard>();
```

Mit einer doppelten *for-Schleife* (siehe unten) legen wir vorläufig fest, dass das Array 10 x 10 = 100 Atome enthalten soll (siehe unten). Wollen wir mehr oder weniger Atome, so lässt sich dies hier leicht ändern.

```
for (int x = 0; x < 10; x++)
{
  for (int y = 0; y < 10; y++)
  {
    Lennard Atom = new Lennard(new PVector(100 + x * 15, 100 + y *
    15), new PVector(0, 0), 1, 10, new PVector(0, 0, 0));

    Atome.add(Atom); //Das Atom an Position (x,y) wird der Liste aller
                     // Atome hinzugefügt
  }
}
```

Die Bedeutung der einzelnen Vektoren und der Zahlenwerte für das *Lennard Atom* in der obigen *for-Schleife* können wir uns natürlich im Nebensketch anschauen. Ich führe sie aber der Einfachheit halber hier mal getrennt auf.

Ortsvektor	`new PVector(100 + x * 15, 100 + y * 15)`
Geschwindigkeitsvektor	`new PVector(0, 0)`
Masse	`1`
Durchmesser	`10`
Farbe	`new PVector(0, 0, 0)`

Die Werte für den Ortsvektor eines einzelnen Atoms sind natürlich nicht vom Himmel gefallen. Sie wurden so gesetzt, dass die Atome bei ihrer regelmäßigen Anordnung zu einem zweidimensionalen Festkörper sich zuerst gegenseitig anziehen. Kommen sie sich zu nahe, dann wirken die abstoßenden Kräfte. Dieses Wechselspiel zwischen Anziehung und Abstoßung kann man sich sehr gut in der Simulation anschauen. Nach einiger Zeit ist die vorherige Ordnung zerstört (Abb. 8.7).

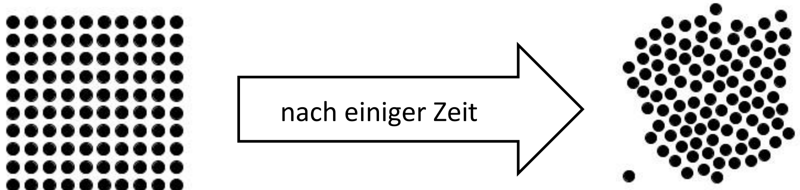


Abbildung 8.7: Eine regelmäßige Anordnung von 100 wechselwirkenden Atomen verliert ihre Ordnung

Erhöht man die Zahl der Atome auf 20 x 20 = 400, so wird die Rechenleistung des Computers schon sehr gefordert. Man beobachtet nach einer zwischenzeitlichen Musterbildung einen Zerfall der Gitterstruktur. Die potenzielle Energie der Startlage war wohl etwas zu hoch. Nun ist es Zeit für interessierte Physiker, mit dem Spielen zu beginnen.

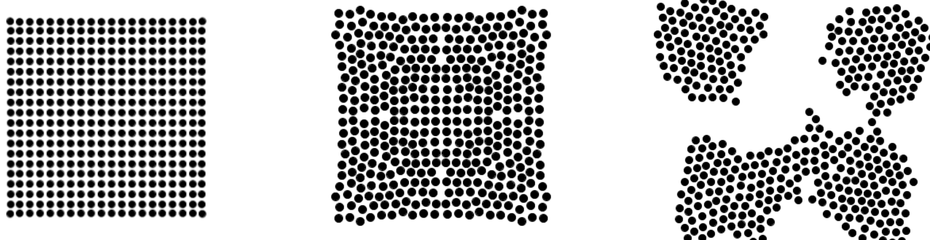


Abbildung 8.8: Zerfall einer Gitterstruktur von 400 wechselwirkenden Atomen

Im Nebensketch begegnet uns noch die Funktion ***get()***. Da wir mit ArrayList ein dynamisches Array verwenden, müssen wir anstelle von *Atome[i]* den Ausdruck *Atome.get(i*) verwenden.

Sketch 07: Festkoerper_02

Hauptsketch

```
// Festkörper 02

// Erstellt eine Liste aller Atome aus der Klasse Lennard
ArrayList<Lennard> Atome = new ArrayList<Lennard>();

void setup()
{
  size(400, 400);

  for (int x = 0; x < 10; x++)
  {
    for (int y = 0; y < 10; y++)
    {
      // Lennard(PVector rTemp, PVector vTemp, float mTemp, float DTemp,
      // PVector cTemp)
      Lennard Atom = new Lennard(new PVector(100 + x * 15, 100 + y *
      15), new PVector(0, 0), 1, 10, new PVector(0, 0, 0));
      Atome.add(Atom); //Das Atom an Position (x,y) wird der Liste aller
                       // Atome hinzugefügt
    }
  }
}

void draw()
{
  background(255);
  for (int i = 0; i < Atome.size(); i++) // Bei ArrayList wird nicht
                                         // .length, sondern .size
verwendet
  {
    for (int j = 0; j < Atome.size(); j++)
    {
      if (i == j)
        continue; // Wenn i == j (Wechselwirkung mit sich selbst), dann
      // überspringe diesen Schritt und fahre mit dem Programm fort

      Atome.get(i).force(Atome.get(j)); //Anstatt Atome[i] benutzt man
                                        // bei ArrayList Atome.get(i)
    }
  }
```

```
  for (int i = 0; i < Atome.size(); i++)
  {
    Atome.get(i).move(0.1);
    Atome.get(i).display();
  }
}
```

Nebensketch

```
class Lennard
{
  // INSTANZVARIABLEN
  PVector r; // Ortsvektor der Atome
  PVector v; // Geschwindigkeitsvektor der Atome
  PVector a = new PVector(); // Der Beschleunigungsvektor a hat den
                             // Startwert (0, 0)
  PVector c; // Farbe (Color) der Atome
  float epsilon = 1; // Tiefe der Potenzialmulde
  float rm; // Abstand zwischen zwei Atomen für F = 0
  float m = 1; // Masse eines Atoms
  float D; // Durchmesser eines Atoms

  // KONSTRUKTOR
  Lennard(PVector rTemp, PVector vTemp, float mTemp, float DTemp,
          PVector cTemp)
  {
    r = rTemp;
    v = vTemp;
    m = mTemp;
    D = DTemp;
    c = cTemp;
    rm = 1.5*D;
  }

  // METHODEN
  void move(float t)
  {
    v.add(PVector.mult(a, t));
    r.add(PVector.mult(v, t));
    a.set(0, 0); // Setzt den Vektor a wieder auf den Wert 0. Sonst
                 // würden sich die a-Werte laufend addieren
  }

  void display()
  {
    fill(c.x, c.y, c.z);
    ellipse(r.x, r.y, D, D);
  }

  // Funktion zum Berechnen des Lennard-Jones Potenzials
  void force(Lennard WechselwirkungsPartner)
  {
    PVector n = PVector.sub(WechselwirkungsPartner.r, r);
    float d = n.mag();
    n.div(d);
    PVector F = PVector.mult(n, epsilon * (-12*pow(rm, 12)*pow(d, -13) +
    12 * pow(rm, 6) * pow(d, -7)));
    a.add(PVector.div(F, m)); // Beschleunigungszuwachs a = F/m
  }
}
```

8.2.4 Festkörper 03

Unseren Festkörper 02 haben wir in der Schwerelosigkeit erzeugt. Erweitern wir den Sketch *Festkoerper_02* nun so, dass er sich bei seiner Erzeugung im Schwerefeld der Erde befindet. D.h., zu der Beschleunigung a, die die Atome aufgrund ihrer gegenseitigen anziehenden und abstoßenden Kräfte erfahren, addieren wir noch die Erdbeschleunigung g. Ob g = 9,81 ms^{-2} in unserer Pixelwelt sinnvoll ist, dies müssen wir ausprobieren. Durch Probieren erhalten wir g = 1 als zweckmäßigen Wert für unseren Sketch *Festkoerper_03.*

Wenn wir den Sketch *Festkoerper_03* starten, dann sehen wir, dass der Festkörper zu Boden fällt und sich anschließend beim Aufprall aufgrund der hohen kinetischen Energie in seine Einzelteile auflöst. Es bildet sich ein Gas. Damit die Atome unser Fenster nicht verlassen, müssen wir dafür sorgen, dass sie an den Wänden reflektiert werden. Dies gelingt durch die Multiplikation der entsprechenden Geschwindigkeitskomponenten mit dem Faktor -1. Im Nebensketch schreiben wir deshalb zum Beispiel **v.y *= -1**. Dies ist die Kurzschreibweise für v.y = -1*v.y.

Interessant wäre es, wenn wir dieses Gas abkühlen könnten. Dann sollte sich am Boden des Fensters wieder ein Festkörper bilden. Mithilfe der beiden Maustasten und dem folgenden Programmteil gelingt uns die Erhöhung und die Reduzierung der kinetischen Energie der einzelnen Atome.

```
if (mousePressed)
{
  if (mouseButton == LEFT)
  {
    for (int i = 0; i < Atome.size(); i++)
    {
     Atome.get(i).v.mult(1.01); // Geschwindigkeit der einzelnen Atome
                                // wird erhöht.
    }
  } else if (mouseButton == RIGHT)
  {
    for (int i = 0; i < Atome.size(); i++)
    {
     Atome.get(i).v.mult(0.99); // Geschwindigkeit der einzelnen Atome
                                // wird verringert.
    }
  }
}
```

Schauen wir uns das Ergebnis der Verringerung der kinetischen Energie an.

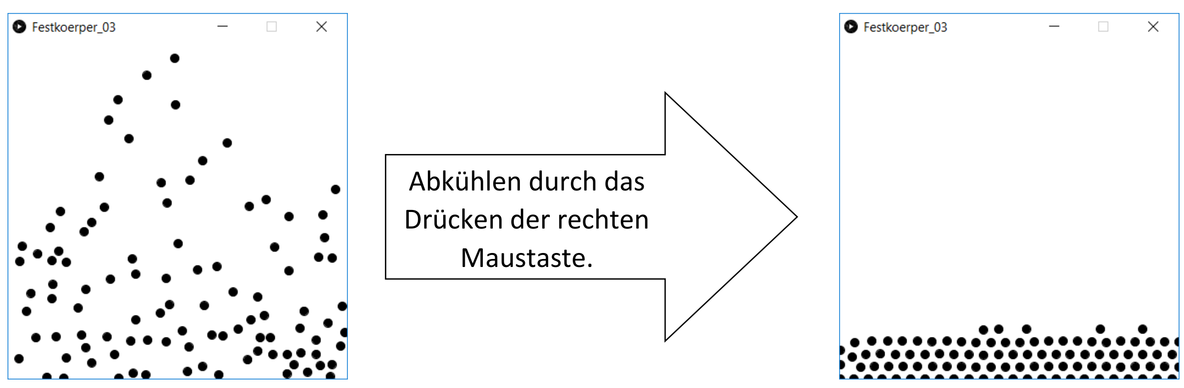


Abbildung 8.9: Kühlt man das Gas ab, so bildet sich am Boden des Behälters ein Festkörper

Sketch 08: Festkoerper_03

Hauptsketch

```
// Festkörper 03

//Erstelle eine Liste aller Atome aus der Klasse Lennard.
ArrayList<Lennard> Atome = new ArrayList<Lennard>();

void setup()
{
  frameRate(500); // Eine möglichst große Bildwiederholungsrate sorgt
                  // für einen flüssigen Bewegungsablauf.
  size(400, 400);

  for (int x = 0; x < 10; x++)
  {
    for (int y = 0; y < 10; y++)
    {
      Lennard Atom = new Lennard(new PVector(100 + x * 25, 100 + y * 25),
      new PVector(0, 0), 1, 10, new PVector(0, 0, 0));
      Atome.add(Atom); // Das Atom an Position (x,y) wird der Liste
                       // aller Atome hinzugefügt
    }
  }
}

void draw()
{
  background(255);
  for (int i = 0; i < Atome.size(); i++) // Bei ArrayList wird nicht
                                         //.length, sondern .size verwendet
  {
    for (int j = 0; j < Atome.size(); j++)
    {
      if (i == j)
        continue; // Wenn i == j (Wechselwirkung mit sich selbst), dann
          // überspringe diesen Schritt und fahre mit dem Programm fort.

      Atome.get(i).force(Atome.get(j)); //Anstatt Atome[i] benutzt man
                                        // bei ArrayList Atome.get(i)
    }
  }

// Mit den beiden Maustasten können wir die Geschwindigkeit und damit
// die  kinetische Energie der Atome vergrößern oder verkleinern.
  if (mousePressed)
  {
    if (mouseButton == LEFT)
    {
      for (int i = 0; i < Atome.size(); i++)
      {
        Atome.get(i).v.mult(1.01); // Geschwindigkeit der einzelnen
                                   // Atome wird erhöht.
      }
    } else if (mouseButton == RIGHT)
    {
      for (int i = 0; i < Atome.size(); i++)
      {
```

```
        Atome.get(i).v.mult(0.99); // Geschwindigkeit der einzelnen
                                   // Atome wird verringert.
      }
    }
  }

  for (int i = 0; i < Atome.size(); i++)
  {
    Atome.get(i).move(0.01, new PVector(0, 1)); // t hat den Wert 0.01
               // und der Vektor g hat die Werte g.x = 0 und g.y = 1.
    Atome.get(i).walls();
    Atome.get(i).display();
  }
}
```

Nebensketch

```
class Lennard
{
  // Instanzvariablen
  PVector c; // Farbe eines Atoms
  PVector r; // Ortsvektor eines Atoms
  PVector v; // Geschwindigkeitsvektor eines Atoms
  PVector a = new PVector(); // Der Beschleunigungsvektor a hat den
                             // Startwert (0, 0)
  float epsilon = 1; // Tiefe der Potenzialmulde
  float rm; // Abstand zweier Atome für F = 0
  float m = 1; // Masse eines Atoms
  float D; // Durchmesser eines Atoms

  // Konstruktor
  Lennard(PVector rTemp, PVector vTemp, float mTemp, float DTemp,
          PVector cTemp)
  {
    r = rTemp;
    v = vTemp;
    m = mTemp;
    D = DTemp;
    c = cTemp;
    rm = 2.0*D;
  }

  // Methoden
  void move(float t, PVector g) // Der Vektor g steht für die
                                // Erdbeschleunigung.
  {
    a.add(g); // g wird zu a hinzu addiert.
    v.add(PVector.mult(a, t));
    r.add(PVector.mult(v, t));
    a.set(0, 0); // Setzt den Vektor a wieder auf den Wert 0. Sonst
                 // würden sich die a-Werte laufend addieren.
  }

  // Damit die Atome unser Fenster nicht verlassen, werden sie an den
  // Wänden reflektiert
  // Weiterhin wird verhindert, dass sie in einer Wand steckenbleiben
  void walls()
  {
    if (r.y > height)
```

```
    {
      v.y *= -1;  // Reflexion am Boden. v.y *= -1 entspricht
                  // v.y = -1*v.y
      r.y = height; //Verhindert, dass die Teilchen in der Wand stecken
                     // bleiben
    }
    if (r.y < 0) // Reflexion an der Decke
    {
      v.y *= -1;
      r.y = 0;
    }
    if (r.x > width)
    {
      v.x *= -1; // Reflexion an der rechten Wand.
      r.x = width;
    }
    if (r.x < 0)
    {
      v.x *= -1; // Reflexion an der linken Wand
      r.x = 0;
    }
  }

  void display()
  {
    fill(c.x, c.y, c.z);
    ellipse(r.x, r.y, D, D);
  }

  void force(Lennard WechselwirkungsPartner)
  {
    PVector n = PVector.sub(WechselwirkungsPartner.r, r);
    float d = n.mag();
    n.normalize();
    PVector F = PVector.mult(n, epsilon * (-12*pow(rm, 12)*pow(d, -13) +
    12 * pow(rm, 6) * pow(d, -7)));
    a.add(PVector.div(F, m));
  }
```

8.3 Zusammenfassung

box() Mit der Funktion *box()* kann man sehr einfach einen Quader dreidimensional zeichnen. Hierzu gibt man die Werte für Höhe, Breite und Tiefe in die Klammer ein. Gibt man nur einen Wert ein, dann zeichnet Processing einen Würfel. Voraussetzung hierfür ist, dass bei *size()* P3D eingefügt wird.

length *length* steht für die Größe eines Arrays, bei dem man sich nicht von vornherein auf eine bestimmte Länge festlegen will. Hier ist ein Beispiel für ein Array mit dem Namen Peter: for (int i = 0; i < Peter.length; i++)

length darf nicht verwechselt werden mit *length(). length()* steht für die Größe eines Strings.

return Wenn bestimmte, selbstgesetzte Bedingungen erfüllt sind, dann kann man mit *return* bewirken, dass eine Funktion sofort verlassen wird. So wird verhindert, dass die nach *return* folgenden Anweisungen ausgeführt werden.

ArrayList<> Mit *ArrayList*<> kann man ein dynamisches Array erstellen. Damit hält man sich offen, wie viele Elemente ein Array enthalten soll. Man kann also leicht Elemente hinzufügen oder entfernen.

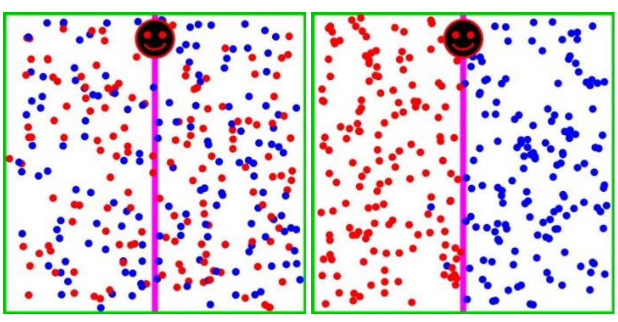
get() Wenn man kein statisches, sondern ein dynamisches Array verwendet, dann muss anstelle von *Elemente[i]* der Ausdruck *Elemente.get(i*) verwendet werden.

***=** *= ist eine Kurzschreibweise. Beispiel: a *= b entspricht a = a*b

8.4 Aufgaben

1. Ändere den Sketch *Ideales_Gas_01* so um, dass sich 100 unterschiedlich große Kreise mit unterschiedlichen Farben unterschiedlich schnell vor einem schwarzen Hintergrund bewegen (siehe Abbildung rechts).

2. Nachdem wir im Kapitel *Ideale Gase und Festkörper* einiges über die Bewegung von Teilchen und ihre Reflexion an Wänden gelernt haben, wollen wir dieses Wissen nun nutzen, um ein Spiel zu programmieren. Die Abbildung rechts zeigt einen Raum, der von drei grünen Wänden begrenzt ist, an denen der gelbe Tennisball reflektiert wird. An der rechten Raumseite ist keine Wand, sodass der Ball hier den Raum verlassen kann. Mit dem roten Schläger kann der Ball jedoch am Verlassen des Raumes gehindert werden. Der rote Schläger kann mit der Maus in y-Richtung bewegt werden. Trifft man den Ball, dann wird er zurück in den Raum reflektiert und man erhält einen Punkt gutgeschrieben. Bei jedem Treffer mit dem Schläger erhöht der Ball aber seine Geschwindigkeit in x-Richtung. Verfehlt man den Ball, dann verlässt er den Raum. Im gleichen Moment wird jedoch ein neuer Ball im Raum mit zufälligen, aber sinnvollen x- und y-Werten erzeugt. Das Spiel dauert 30 Sekunden. Im Fenster werden die noch verbleibende Zeit und die erreichten Punkte angezeigt. Programmiere diesen Sketch.



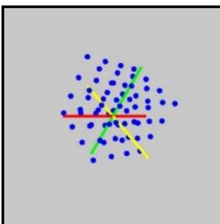
3. Der Sketch *Entropie* soll nun im Sinne des Maxwellschen Dämons geändert werden. Im Fenster befindet sich beim Start des Sketches eine Mischung von 150 roten Teilchen mit hoher kinetischer Energie und 150 blauen Teilchen mit geringer kinetischer Energie. Beide Fensterhälften sind durch eine violette Wand getrennt. Der Maxwellsche Dämon sorgt dafür, dass die energiereichen roten Teilchen die violette Wand nur von rechts nach links passieren können und die energiearmen blauen Teilchen die Wand nur von links nach rechts passieren können. Nach einiger Zeit stellt sich im Widerspruch zum zweiten Hauptsatz der Thermodynamik eine

Energiedifferenz zwischen der linken und der rechten Fensterhälfte ein (siehe Abbildung). Mit dieser Energiedifferenz kann dann eine Wärmekraftmaschine betrieben werden.

Informiere dich zuerst über den Konflikt zwischen dem zweiten Hauptsatz der Thermodynamik und dem Maxwellschen Dämon. Schreibe anschließend einen Sketch, der entsprechend der obigen Abbildung aus einem ungeordneten Zustand einen geordneten Zustand erzeugt.

4. Der Sketch *Ideales_Gas_02* soll so geändert werden, dass das Gas an den Wänden kondensiert. Hierzu soll die Gravitationskraft ausgeschaltet werden.

5. Der Sketch *Festkörper_02* generiert einen zweidimensionalen Festkörper. Ändere den Sketch so um, dass ein dreidimensionaler Festkörper entsteht, bei dem alle Teilchen miteinander wechselwirken. Der dreidimensionale Festkörper soll mit der Maus um die x- und y-Achse drehbar sein.



9 Kernphysik und Teilchenphysik

Was erwartet uns?
verblassen, strokeCap(ROUND), abgerundete Ecken des Rechteckes
Wiederholung von: arc(), auf Stellen hinter dem Koma runden, Minim

9.1 Nebelkammer

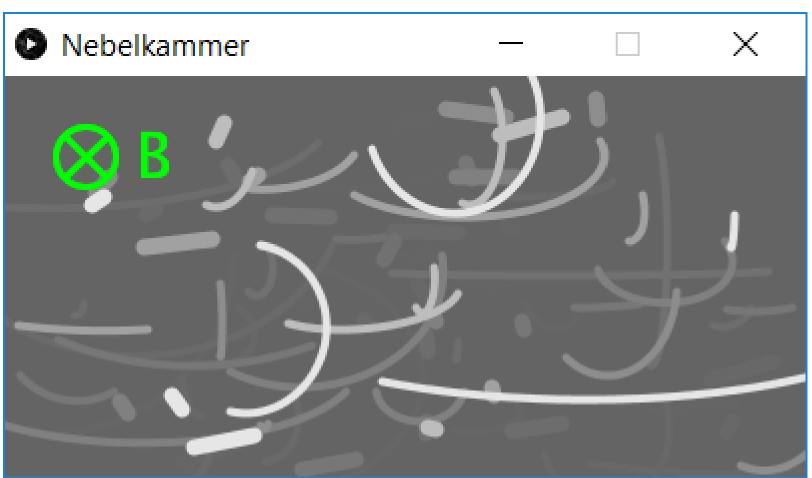


Abbildung 9.1: Frische und verblassende Nebelspuren in einer kontinuierlich arbeitenden Nebelkammer

Mittels einer kontinuierlich arbeitenden Nebelkammer kann man die Flugbahnen ionisierender Teilchen beobachten. Diese hochenergetischen, geladenen Teilchen erzeugen durch Stoßionisation Spuren von Ionen. Diese Ionen dienen dann als Kondensationskeime, sodass sich Nebeltröpfchenspuren entlang der Flugbahn bilden. Dadurch werden die Flugbahnen der ionisierenden Teilchen sichtbar. Nach einiger Zeit lösen sich diese Nebelspuren jedoch wieder auf.

Legt man an die Nebelkammer ein Magnetfeld an, so kann man bei nicht zu schnellen Elektronen (Beta-Minus-Teilchen geringer Energie) dünne gekrümmte Bahnen beobachten. Die Bahnen der Alphateilchen sind aufgrund der relativ großen Masse der Alphateilchen kurz, nahezu gerade und deutlich dicker als die Bahnen der Betateilchen, da die Alphateilchen auf ihrem Weg durch den übersättigten Dampf deutlich mehr Ionen erzeugen und somit mehr Energie pro Streckeneinheit verlieren.

Unsere Aufgabe ist es nun, einen Sketch zu schreiben, der das stochastische Auftreten und langsame Verschwinden der Nebelspuren von Beta- und Alphateilchen grafisch darstellt. Der Sketchteil, der die Bahnen der Alphateilchen erzeugt, ist leicht zu verstehen und bedarf deshalb keiner weiteren Erläuterung. Den Sketchteil zur Erzeugung der Nebelspuren von Betateilchen sollte man eigentlich auch verstehen. Wer jedoch im Kapitel 4.3 nicht ganz so gut aufgepasst hat, dem soll hier nochmal geholfen werden. In einem homogenen Magnetfeld bewegen sich Betateilchen auf kreisförmigen Bahnen, wenn sie senkrecht zu den Feldlinien in das Magnetfeld eintreten. Solche kreisförmigen Bahnen lassen sich in Processing gut mit der Funktion *arc()* darstellen. Treten Betateilchen schräg ins Magnetfeld ein, so beschreiben sie Schraubenbahnen, die je nach Betrachtungswinkel wie verbogene Kreisbahnen aussehen. Unterschiedliche Kreisbögen und auch unterschiedlich verbogene Kreisbögen lassen sich mit der Funktion *arc()* in

Kombination mit der Funktion *random()* generieren. Hier eine Beispielzeile aus unserem Sketch *Nebelkammer*.

```
arc(random(20, 380), random(20, 180), random(50), random(50), random(-1, 1), PI /random(1, 2));
```

Der erste rote Ausdruck legt per Zufallsgenerator den x-Wert für den Mittelpunkt des „Kreisbogens“ zwischen 20 und 380 fest. Der zweite rote Ausdruck den y-Wert zwischen 20 und 180. Die beiden grünen bestimmen den „Kreisdurchmesser“ in x- und y-Richtung zwischen 0 und 50 Pixel und die beiden blauen den Anfangs- und den Endwinkel des „Kreisbogens“ im Bogenmaß. Beachten muss man, dass bei Processing der Winkel im Uhrzeigersinn gezählt wird.

Wie man Nebelspuren erzeugt, wissen wir nun. Doch wie können wir sie langsam wieder verschwinden lassen? Hierzu benutzen wir einen Trick. Bei *void draw()* verzichten wir auf die Funktion *background()*. Stattdessen lassen wir ein fenstergroßes, graues und transparentes Rechteck erzeugen. Die erste Zahl in der Klammer von *fill()* gibt den Grauwert und die zweite Zahl gibt die Transparenz an.

```
noStroke();
fill(100, 100); // Grauwert (erste Zahl), Transparenz (zweite
                // Zahl)
rect(0, 0, 400, 200);
```

Bei jedem Aufruf von *void draw()* werden die zuvor erzeugten weißen Nebelspuren mit transparentem Grau abgedunkelt, bis sie nach mehreren Durchläufen ganz verschwunden sind.

Sketch 01: Nebelkammer

```
// Nebelkammer

float xa = 10;
float xb = 390;
float x1;
float x2;
float x3;
float ya = 10;
float yb = 190;
float y1;
float y2;
float y3;

void setup()
{
  size(400, 200);
  frameRate(1); // Bildwiederholungsrate
}
void draw()
{
  /* Anstelle von background() zeichnen wir ein fenstergroßes,
    transparentes Rechteck. So verblassen die Nebelspuren bei jedem
    Aufruf von void draw()immer mehr */
  noStroke();
  fill(100, 100); // Grauwert (erste Zahl), Transparenz (zweite Zahl)
  rect(0, 0, 400, 200);

  // Zeichnen der Nebelspuren
  stroke(230);
```

```
  strokeCap(ROUND); // rundes Linienende
  noFill();

  // Spuren von langsamen Elektronen
  strokeWeight(4);
  // Mittelpunkt (x, y), Breite, Höhe, Anfangswinkel, Endwinkel
  arc(random(20, 380), random(20, 180), random(50), random(50),
  random(-1, 1), PI /random(1, 2));
  arc(random(20, 380), random(20, 180), random(100), random(100),
  random(-1.2, 1.2), PI /random(1, 2));
  arc(random(20, 380), random(20, 180), random(150), random(150),
  random(-1.5, 1.5), PI /random(1, 2));
  arc(random(10, 390), random(10, 190), random(380), random(190),
  random(0, 1.5), PI /random(1, 2));

  // Spuren von Alpha-Teilchen
  strokeWeight(8);
  x1 = random(xa, xb);
  x2 = random(xa, xb);
  x3 = random(xa, xb);
  y1 = random(ya, yb);
  y2 = random(ya, yb);
  y3 = random(ya, yb);
  line(x1, y1, x1+random(0, 5), y1+random(0, 10));
  line(x2+20, y2+5, x2-random(5, 15), y2+random(0, 20));
  line(x3+5, y3, x3+random(0, 15), y3-random(0, 10));


  // Zeichen für das B-Feld
  noFill();
  stroke(0, 255, 0);
  strokeWeight(3);
  ellipse(40, 40, 30, 30);
  line(30, 30, 50, 50);
  line(51, 30, 30, 50);
  fill(0, 255, 0);
  textSize(32);
  text("B", 65, 51);
}
```

9.2 Zählrohr und Nulleffekt

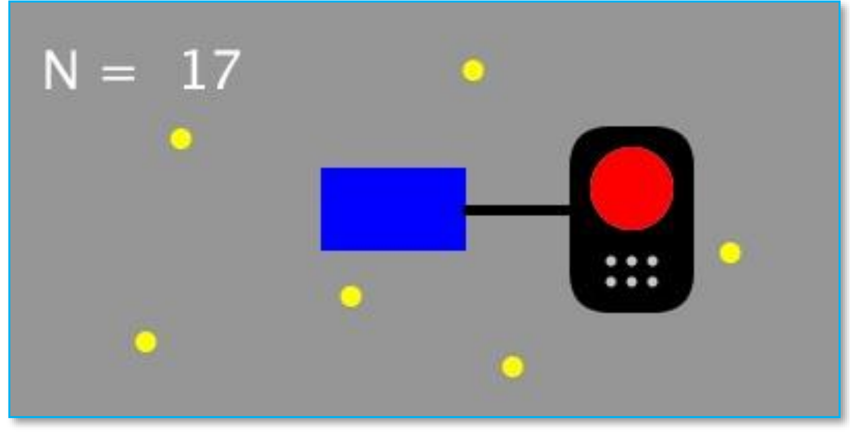


Abbildung 9.2: Zählrohr mit Elektronik. Die gelben Punkte stellen die Hintergrundstrahlung dar.

Wenn man in seinem Zimmer einen Geigerzähler einschaltet, so hört man in unregelmäßigen Abständen Knackgeräusche, obwohl sich kein radioaktives Präparat im Raum befindet. Man nennt diesen Effekt „Nulleffekt“. Die radioaktive Strahlung, auf die der Geigerzähler hierbei anspricht, stammt aus dem Mauerwerk, aus dem Boden und von radioaktiven Teilchen, die sich in der Luft befinden. Weiterhin trägt die kosmische Strahlung zu unserer natürlichen Strahlenbelastung bei.

In unserem Sketch wollen wir dies simulieren. Wir zeichnen entsprechend der Abbildung 9.2 ein Zählrohr (blaues Rechteck) mit zugehöriger Elektronik (schwarzes abgerundetes Rechteck) und erzeugen per Zufallsgenerator radioaktive Strahlung (gelbe Punkte). Wird das Zählrohr getroffen, dann soll ein Knackgeräusch ertönen und die rote Warnlampe am Gehäuse aufleuchten. Ebenso soll die Anzahl der Treffer im Fenster angezeigt werden.

Das Zeichnen des Geigerzählers und der zufällig aufblitzenden gelben Punkte zur Darstellung der natürlichen radioaktiven Strahlung wird uns keine Kopfschmerzen bereiten. Anders ist es mit den Knackgeräuschen. Hierzu muss Processing auf die Soundkarte zugreifen können. Wie dies gelingt, wurde im *Kapitel 5.4 Processing und Soundkarte* erklärt. Erinnern wir uns: Zuerst muss man die Bibliothek *Minim* aus dem Internet herunterladen und dann in den Processing-Ordner *Contributed Libraries* einfügen. Mit der Anweisung *import ddf.minim.** können wir nun in unserem Sketch alle Klassen aus der Bibliothek Minim verwenden. Wir benötigen jedoch nur *Minim* mit der Variablen *minim* und *Audioplayer* mit der Variablen *rekorder*.

Damit der Audioplayer über die Soundkarte ein Knackgeräusch abspielen kann, muss sich im Ordner *data*, der sich innerhalb des Sketchordners befinden muss, auch eine Datei *knack.mp3* befinden. Diese kann man zum Beispiel mit dem kostenlosen Audioeditor *Audacity* erstellen. Mit der folgenden Programmzeile laden wir dann *knack.mp3* in unseren Rekorder.

```
rekorder = minim.loadFile("knack.mp3")
```

Erzeugt nun der Zufallsgenerator *random()* einen gelben Punkt, der innerhalb des blauen Rechteckes liegt, welches das Zählrohr darstellt, dann werden in der if-Anweisung unseres Sketches *Nulleffekt* die folgenden Zeilen aufgerufen.

```
rekorder.cue(0); // Setzt den Rekorder auf 0 Millisekunden
rekorder.play(); // Spielt die mp3-Datei ab
```

Würden wir nur ***rekorder.play()*** schreiben, dann würde die Audiodatei *knack.mp3* nur einmal aufgerufen. Um dies zu verhindern, setzen wir mit der Funktion ***rekorder.cue()*** den Rekorder vor jedem Abspielen des Tons auf Null.

Zum Schluss noch eine Bemerkung zu der folgenden Programmzeile.

```
rect(270, 60, 60, 90, 20)
```

Was bedeutet die fünfte Zahl in der Klammer? Mit ihr kann man die Radien für die abgerundeten Ecken des Rechteckes bestimmen. Dadurch sieht unser schwarzes Gehäuse etwas schicker aus.

Sketch 02: Nulleffekt

```
// Nulleffekt

// Aufruf der Bibliothek Minim
import ddf.minim.*;
Minim minim; // Minim bekommt den Namen "minim"
```

```
AudioPlayer rekorder; // Der Audioplayer bekommt den Namen "rekorder"

float x;
float y;
int N;

void setup()
{
  size(400, 200);

  // Die Variable minim wird erstellt
  minim = new Minim (this);

  // Die mp3-Datei wird in den Rekorder geladen
  rekorder = minim.loadFile("knack.mp3");
}

void draw()
{
  frameRate(15); // Bildwiederholungsrate
  background(150);

  // Teilchen der Untergrundstrahlung
  noStroke();
  fill(255, 255, 0);
  ellipse(x, y, 10, 10);
  x = random(0, 400);
  y = random(0, 200);

  // Geigerzähler
  fill(0, 0, 255);
  rect(150, 80, 70, 40); // Zählrohr
  stroke(0);
  strokeWeight(5);
  line(220, 100, 270, 100); // Verbindung
  noStroke();
  fill(0);
  rect(270, 60, 60, 90, 20); // Die fünfte Zahl bestimmt die
                             // Kantenrundung des Gehäuses
  fill(200);
  ellipse(300, 90, 40, 40); // Kontrolllampe
  // Lautsprecheröffnungen
  ellipse(290, 125, 5, 5);
  ellipse(300, 125, 5, 5);
  ellipse(310, 125, 5, 5);
  ellipse(290, 135, 5, 5);
  ellipse(300, 135, 5, 5);
  ellipse(310, 135, 5, 5);


  if (x > 150 && x < 230 && y > 80 && y < 120)
  {
    // Kontrolllampe soll rot aufleuchten
    noStroke();
    fill(255, 0, 0);
    ellipse(300, 90, 40, 40);

    // Treffer werden gezählt
    N = N + 1;

    // Lautsprechergeräusch
```

```
    rekorder.cue(0); // Setzt den Rekorder auf 0 Millisekunden
    rekorder.play(); // Spielt die mp3-Datei ab
  }

  // Anzeige des Nulleffektes
  fill(255);
  textSize(26);
  text("N =  " +N, 30, 50);
}
```

9.3 Atomkern

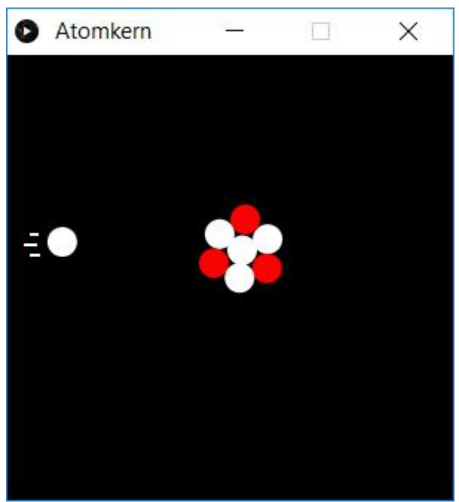


Abbildung 9.3: Ein Lithiumkern wird mit einem Neutron beschossen. Die Bewegungsstriche wurden per Hand eingefügt.

Die positiv geladenen Protonen in einem Atomkern stoßen sich aufgrund des geringen Abstandes mit sehr großer Coulombkraft ab. Warum bleiben sie trotzdem im Kern? Es muss also eine anziehende Kraft geben, die stärker ist als die abstoßende Coulombkraft. Nach dem Standardmodell der Teilchenphysik bestehen Protonen und Neutronen aus je drei Quarks. Diese drei Quarks sind Träger der Farbladung rot, grün und blau und halten aufgrund der Farbkraft, auch starke Kraft oder Gluonenkraft genannt, zusammen. Da die drei Farbladungen der Quarks zusammen weiß ergeben, wirkt die Farbkraft nicht in größeren Abständen. Nur im Nahbereich schimmert etwas von der Farbladung durch. Dies ist der Grund dafür, dass die Nukleonen in einem Atomkern zusammenhalten, wenn sie sich sehr nahekommen. Ab ca. $2 \cdot 10^{-15}$ m wirkt die Farbkraft zwischen den Nukleonen anziehend. Ab $0{,}5 \cdot 10^{-15}$ m erfahren die Nukleonen jedoch eine abstoßende Kraft, die verhindert, dass der Kern in sich zusammenstürzt. Das Kernpotenzial ist also näherungsweise, aber wirklich nur näherungsweise, vergleichbar mit dem Lennard-Jones-Potenzial (siehe Kapitel 8.2.2). In Wirklichkeit ist den Physikern eine vollständige Beschreibung der Kernkraft bis heute noch nicht gelungen.

In unserem Sketch *Atomkern* unterscheiden wir drei Kraftwirkungen. Die abstoßende Coulombkraft, die nur zwischen den Protonen wirkt und mit r^2 abnimmt. Die anziehende starke Kraft zwischen den Nukleonen (Protonen und Neutronen) die mit r^4 abnimmt und die abstoßende starke Kraft zwischen den Nukleonen, wenn sich diese zunahekommen. Sie nimmt mit r^6 ab. Spin-Spin- und andere Wechselwirkungen bleiben in unserem einfachen Sketch unberücksichtigt. Durch diese Vereinfachung können sich in unserem Sketch allerdings Atomkerne ergeben, die es

in der Wirklichkeit nicht gibt. Trotzdem erhalten wir einige brauchbare Ergebnisse. Der Sketch *Atomkern* dient uns aber hauptsächlich als Wiederholung dafür, wie man ein Vielteilchensystem mittels einer Klasse programmiert. Zur Auffrischung des bisher Gelernten schaue man sich nochmal im Kapitel 8 die Sketche *Ideales_ Gas_02* und *Festkoerper_02* an.

Um uns offen zu halten, aus wie vielen Nukleonen unser Atomkern bestehen soll, erstellen wir im Hauptsketch ein dynamisches Array. Wir schreiben

```
ArrayList<Teilchen> teilchenListe = new ArrayList<Teilchen>();
```

In der doppelten *for-Schleife* im Hauptsketch legen wir deshalb auch keine konkrete Array-Größe fest, sondern schreiben stattdessen *size().* Siehe hierzu den folgenden Auszug aus dem Hauptsketch.

```
for (int i = 0; i < teilchenListe.size(); i++)
{
  for (int j = 0; j < teilchenListe.size(); j++)
  {
    if (i == j)
      continue;

    teilchenListe.get(i).wechselwirkung(teilchenListe.get(j));
  }
}
```

In dem obigen Sketchausschnitt begegnen wir wieder *get(i).* In einem statischen Array würden wir *teilchenliste[i]* schreiben. Da wir mit *ArrayList* aber ein dynamisches Array verwenden, müssen wir *teilchenliste.get(i)* schreiben. Weitere Informationen zu *ArrayList* findet man in der Referenz von Processing.

Mittels unseres Sketches können wir Atomkerne zusammenbasteln und auch Kernspaltung betreiben. Dazu benutzen wir unsere Maustasten. Mit der linken Maustaste können wir ruhende Protonen im Fenster erzeugen und mit der rechten Maustaste ruhende Neutronen. Erzeugen wir ein Nukleon im Bereich der anziehenden starken Kraft eines anderen Nukleons, so schließen sie sich zu einem Atomkern zusammen. Ist der Abstand der beiden Nukleonen allerdings zu gering, dann erfahren sie den stark abstoßenden Anteil der starken Kraft. Weiterhin bewegen sich Protonen voneinander weg, wenn ihre Entfernung so groß ist, dass die abstoßende Coulombkraft größer ist als die anziehende starke Kraft.

In der Abbildung 9.3 sehen wir einen stabilen Lithiumkern ($^{7}_{3}Li$), der entsprechend den obigen Ausführungen mittels Mausklick erstellt wurde. Ihn wollen wir nun mithilfe eines schnellen Neutrons spalten. Schnelle Neutronen können wir durch Drücken der mittleren Maustaste erzeugen. Das Ergebnis dieser Kernspaltung hängt dabei von der Geschwindigkeit des Neutrons ab. Diese Geschwindigkeit können wir in der folgenden Zeile aus dem Nebensketch ändern.

```
teilchenListe.add(new Teilchen(new PVector(mouseX, mouseY), new
PVector(1, 0), 0))
```

Diese Zeile ist wie folgt zu lesen. Dem Array *teilchenListe* wird ein neues Teilchen hinzugefügt, welches die x- und y-Koordinaten des Mauszeigers besitzt. Weiterhin hat es die Geschwindigkeit 1 in x-Richtung und die Ladung 0. Nun können wir mit dem Spielen beginnen. Wir können die Geschwindigkeit ändern und schauen, welche Spaltprodukte sich ergeben. Ändern wir die Ladung von 0 auf 1, so verwandeln wir das Neutron in ein Proton. Bei welcher Geschwindigkeit kann es

aufgrund der abstoßenden Coulombkraft den Kern nicht mehr erreichen? Also: Spielen, spielen, spielen.

Sketch 03: Atomkern

Hauptsketch

```
// Atomkern

ArrayList<Teilchen> teilchenListe = new ArrayList<Teilchen>(); // Liste
                                                              // aller Teilchen

void setup()
{
  size(500, 500);
}

void mousePressed()
{
  // Proton hinzufügen durch Drücken der linken Maustaste
  if (mouseButton == LEFT)
    teilchenListe.add(new Teilchen(new PVector(mouseX, mouseY),
    new PVector(0, 0), 1));

  // Neutron hinzufügen durch Drücken der rechten Maustaste
  if (mouseButton == RIGHT)
    teilchenListe.add(new Teilchen(new PVector(mouseX, mouseY),
    new PVector(0, 0), 0));

  // Schnelles Neutron hinzufügen durch Drücken der mittleren Maustaste
  if (mouseButton == CENTER)
    teilchenListe.add(new Teilchen(new PVector(mouseX, mouseY),
    new PVector(1, 0), 0));
}

void draw()
{
  background(0, 0, 0);

  for (int i = 0; i < teilchenListe.size(); i++)
  {
    for (int j = 0; j < teilchenListe.size(); j++)
    {
      if (i == j) // Wechselwirkung mit sich selbst verhindern!!!
        continue;

      // Wechselwirkung des i-ten Teilchens mit dem j-ten Teilchen
      // berechnen
      teilchenListe.get(i).wechselwirkung(teilchenListe.get(j));
    }
  }

  for (int i = 0; i < teilchenListe.size(); i++)
  {
    // Aufruf der Funktionen
    teilchenListe.get(i).bewegen(5); // In der Klammer steht der Wert
                                     // für den Zeitschritt dt
    teilchenListe.get(i).zeichnen();
  }
```

```
}
```

Nebensketch

```
class Teilchen
{
  // INSTANZVARIABLEN
  float q; // Teilchenladung

  // Position, Geschwindigkeit und Beschleunigung
  PVector x = new PVector(), v = new PVector(), a = new PVector();

  // KONSTRUKTOR
  Teilchen(PVector xTemp, PVector vTemp, float qTemp)
  {
    x = xTemp;
    v = vTemp;
    q = qTemp;
  }

  // METHODEN
  // Wechselwirkung mit einem anderen Teilchen berechnen
  void wechselwirkung(Teilchen wechselwirkungspartner)
  {
  // Betrag der vektoriellen Differenz
    float abstand = PVector.sub(wechselwirkungspartner.x, x).mag();

    PVector richtung = PVector.sub(wechselwirkungspartner.x,
    x).normalize(); // Einheitsvektor zur Bestimmung der Richtung

    // Änderung des Beschleunigungsvektors a durch die abstoßende
    // Coulomb-Kraft
    a.add(PVector.mult(richtung, -0.1 * wechselwirkungspartner.q * q /
    pow(abstand, 2))); // Fc = -0.1 * q1 * q2 / r^2

    // Änderung des Beschleunigungsvektors a durch den anziehenden Teil
    // der starken Kraft (vereinfacht, ohne e-Funktion)
    a.add(PVector.mult(richtung, 100.0 / pow(abstand, 4)));
    // F = 100/r^4

    // Änderung des Beschleunigungsvektors a durch den abstoßenden Teil
    // der starken Kraft (vereinfacht, ohne e-Funktion)
    a.add(PVector.mult(richtung, -40000.0 / pow(abstand, 6))); // F =
                                                    // -40000.0 / r^6
    // Die Konstanten -0.1, 100.0 und -40000.0 wurden experimentell,
    // also durch Ausprobieren ermittelt
  }

  // Das Teilchen entsprechend der wirkenden Kräfte bewegen
  void bewegen(float dt)
  {
    a.add(PVector.mult(v, -0.001)); /* Leichtes "Abkühlen" der
 Simulation, indem eine schwache, der Bewegungsrichtung der Teilchen
 entgegengesetzte Beschleunigung wirkt. Der Faktor -0.001 hat die
 Maßeinheit s^-1 */

    v.add(PVector.mult(a, dt)); // v1 = v0 + a0 * dt
    x.add(PVector.mult(v, dt)); // x1 = x0 + v1 * dt
    a.set(0, 0, 0); // Beschleunigung auf Null zurücksetzen
  }
```

```
  // Protonen und Neutronen zeichnen
  void zeichnen()
  {
    noStroke();
    // Protonen (Teilchenladung q = 1) werden rot gezeichnet, Neutronen
    // (Teilchenladung q = 0) weiß
    if (q > 0)
      fill(255, 0, 0);
    else
      fill(255, 255, 255);
    ellipse(x.x, x.y, 20, 20);
  }
}
```

9.4 Betaminuszerfall

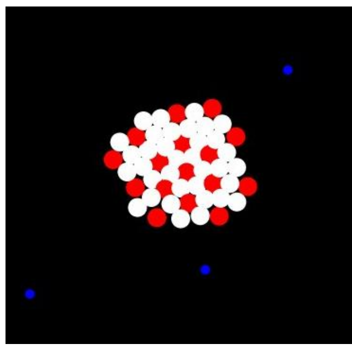


Abbildung 9.4: Neutronen des Kerns wandeln sich durch Aussendung eines Elektrons in ein Proton um

Unser Sketch *Atomkern* hat den Nachteil, dass man auch einen Atomkern zusammenbauen kann, der nur aus Neutronen besteht. In Wirklichkeit gibt es solche Kerne nicht. Besitz ein Atomkern zu viele Neutronen, so verwandeln sich solange Neutronen unter Aussendung eines Elektrons und eines Antineutrinos in Protonen um, bis wieder ein einigermaßen ausgeglichenes Verhältnis herrscht. Auch frei existierende Neutronen verwandeln sich mit einer Halbwertszeit von ca. 610 Sekunden in ein Proton unter Aussendung eines Elektrons und eines Antineutrinos. Eine solche Umwandlung nennt man Betaminuszerfall.

Unseren Sketch *Atomkern* wollen wir nun so ändern, dass er den oben genannten Bedingungen gerecht wird. Dazu müssen wir nur wenige Änderungen im Nebensketch vornehmen. Der Hauptsketch bleibt so wie er ist. Die Änderungen im Nebensketch sind gelb hervorgehoben (siehe unten). Die Idee, die diesen Änderungen zugrunde liegt, besteht darin, dass ein Neutron nur dann nicht zerfällt, wenn es ein Proton als Nachbarn hat. Hat es nur Neutronen als Nachbarn oder ist es frei, dann zerfällt es in ein Proton und in ein Elektron. Das Antineutrino vernachlässigen wir.

Als erste Änderung begegnen wir dem Datentyp boolean. Dieser Datentyp kennt nur true (wahr) oder false (falsch). Hiermit können wir steuern, ob ein Neutron zerfällt oder nicht. Gilt `boolean istFrei = true`, dann zerfällt das Neutron. Ist der Wechselwirkungspartner des Neutrons ein Proton und ist dieses Proton weniger als 25 Pixel entfernt, dann gilt `istFrei = false` und das Neutron zerfällt nicht. Diese Entscheidung findet in der folgenden Sketchzeile statt.

```
if (wechselwirkungspartner.q > 0 && abstand < 25)
  istFrei = false;
```

Da der Zerfall von Neutronen ein statistischer Prozess ist, besitzen die einzelnen Neutronen eine jeweils unterschiedliche Lebensdauer. Diesem Umstand werden wir mit der folgenden if-Anweisung gerecht. Die Funktion *random()* generiert eine Zahl zwischen 0 und 1. Nur wenn diese Zahl kleiner oder gleich 0,0025 ist, wird ein Elektron mit einer Geschwindigkeit zwischen -1 und +1 und der Ladung -1 erzeugt, sowie die Ladung des Neutrons von q = 0 in q = 1 geändert. Das Neutron wird zum Proton. Anschließend wird *istFrei* wieder auf *true* gesetzt.

```
if (q == 0 && istFrei && random(0.0, 1.0) <= 0.0025)
{
  teilchenListe.add(new Teilchen(new PVector(x.x, x.y),
  new PVector(random(-1, 1), random(-1, 1)), -1));
  q = 1;
}

istFrei = true;
```

Die restlichen Änderungen sind leicht zu verstehen und werden deshalb nur im Sketch kommentiert.

Sketch 04: Betaminuszerfall

Hauptsketch

```
// Betaminuszerfall

// Liste aller Teilchen
ArrayList<Teilchen> teilchenListe = new ArrayList<Teilchen>();

void setup()
{
  size(500, 500);
}

void mousePressed()
{
  // Proton hinzufügen durch Drücken der linken Maustaste
  if (mouseButton == LEFT)
    teilchenListe.add(new Teilchen(new PVector(mouseX, mouseY),
    new PVector(0, 0), 1));

  // Neutron hinzufügen durch Drücken der rechten Maustaste
  if (mouseButton == RIGHT)
    teilchenListe.add(new Teilchen(new PVector(mouseX, mouseY),
    new PVector(0, 0), 0));

  // Schnelles Neutron hinzufügen durch Drücken der mittleren Maustaste
  if (mouseButton == CENTER)
    teilchenListe.add(new Teilchen(new PVector(mouseX, mouseY),
    new PVector(1, 0), 0));
}

void draw()
{
  background(0, 0, 0);

  for (int i = 0; i < teilchenListe.size(); i++)
```

```
  {
    for (int j = 0; j < teilchenListe.size(); j++)
    {
      if (i == j) // Wechselwirkung mit sich selbst verhindern!!!
        continue;

      teilchenListe.get(i).wechselwirkung(teilchenListe.get(j));
// Wechselwirkung des i-ten Teilchens mit dem j-ten Teilchen berechnen
    }
  }

  for (int i = 0; i < teilchenListe.size(); i++)
  {
    // Aufruf der Funktionen
    teilchenListe.get(i).bewegen(5); // In der Klammer steht der Wert
                                     // für den Zeitschritt dt
    teilchenListe.get(i).zeichnen();
  }
}
```

Nebensketch

```
class Teilchen
{
  // INSTANZVARIABLEN
  float q; // Teilchenladung
  boolean istFrei = true; // Bedingung für den Beta-minus Zerfall des
                          // Neutrons
  PVector x = new PVector(), v = new PVector(), a = new PVector();
  // Position, Geschwindigkeit und Beschleunigung

  // KONSTRUKTOR
  Teilchen(PVector xTemp, PVector vTemp, float qTemp)
  {
    x = xTemp;
    v = vTemp;
    q = qTemp;
  }

  // METHODEN
  // Wechselwirkung mit einem anderen Teilchen berechnen
  void wechselwirkung(Teilchen wechselwirkungspartner)
  {
    if (q < 0 || wechselwirkungspartner.q < 0) // Elektronen sollen
                                               // nicht wechselwirken
      return;

    float abstand = PVector.sub(wechselwirkungspartner.x, x).mag();
    // Betrag der vektoriellen Differenz
    PVector richtung = PVector.sub(wechselwirkungspartner.x,
    x).normalize(); // Einheitsvektor zur Bestimmung der Richtung

    if (wechselwirkungspartner.q > 0 && abstand < 25) // Wenn ein Proton
                   // in der Nähe ist, soll das Neutron nicht zerfallen
      istFrei = false;

    // Änderung des Beschleunigungsvektors a durch die abstoßende
    // Coulomb-Kraft
```

```
    a.add(PVector.mult(richtung, -0.1 * wechselwirkungspartner.q * q /
    pow(abstand, 2))); // Fc = n * q1 * q2 / r^2

    // Änderung des Beschleunigungsvektors a durch den anziehenden Teil
    // der starken Kraft (vereinfacht, ohne e-Funktion)
    a.add(PVector.mult(richtung, 100.0 / pow(abstand, 4))); // F =
                                                            // n*/r^4

    // Änderung des Beschleunigungsvektors a durch den abstoßenden Teil
    // der starken Kraft (vereinfacht, ohne e-Funktion)
    a.add(PVector.mult(richtung, -40000.0 / pow(abstand, 6))); //F =
                                                               // n*/r^6
    // Die Konstanten -0.1, 100.0 und -40000.0 wurden experimentell,
    // also durch Ausprobieren ermittelt
  }

  // Das Teilchen entsprechend der wirkenden Kräfte bewegen
  void bewegen(float dt)
  {
    a.add(PVector.mult(v, -0.001)); /* Leichtes "Abkühlen" der
Simulation, indem eine schwache, der Bewegungsrichtung der Teilchen
entgegengesetzte Beschleunigung wirkt. Der Faktor -0.001 hat die
Maßeinheit s^-1 */

    v.add(PVector.mult(a, dt)); // v1 = v0 + a0 * dt
    x.add(PVector.mult(v, dt)); // x1 = x0 + v1 * dt
    a.set(0, 0, 0); // Beschleunigung auf null zurücksetzen

    if (q == 0 && istFrei && random(0.0, 1.0) <= 0.0025) // Ist es ein
                           // freies Neutron, so soll es zufällig zerfallen
    {
      teilchenListe.add(new Teilchen(new PVector(x.x, x.y),
      new PVector(random(-1, 1), random(-1, 1)), -1)); // Schnelles
                                                        // Elektron aussenden
      q = 1; // Neutron in ein Proton umwandeln
    }

    istFrei = true; // Bedingung für den Betaminuszerfall zurücksetzen
  }

  // Protonen und Neutronen zeichnen
  void zeichnen()
  {
    noStroke();
    // Protonen (Teilchenladung q = 1) werden rot gezeichnet, Neutronen
    // (Teilchenladung q = 0) weiß, Elektronen (Teilchenladung q = -1)
    // blau
    if (q > 0)
    {
      fill(255, 0, 0);
      ellipse(x.x, x.y, 20, 20);
    } else if (q == 0)
    {
      fill(255, 255, 255);
      ellipse(x.x, x.y, 20, 20);
    } else if (q < 0)
    {
      fill(0, 0, 255);
      ellipse(x.x, x.y, 10, 10);
    }
  }
```

}

9.5 Zerfallsgesetz

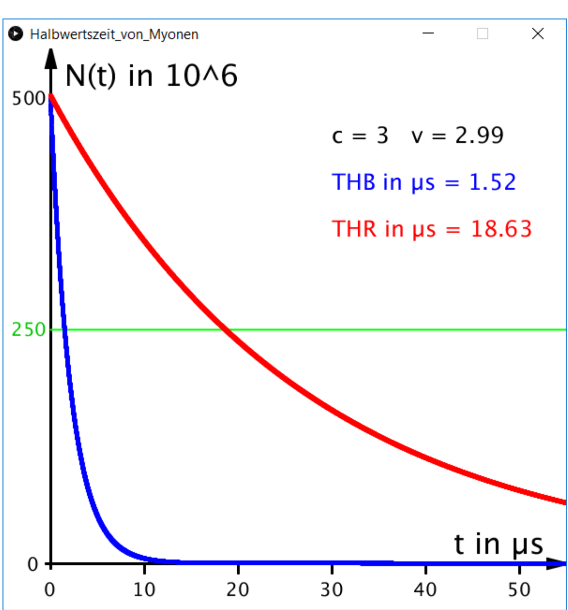


Abbildung 9.5: Exponentielle Abnahme eines Myonenpaketes.
Ruhende Myonen (blauer Graph), bewegte Myonen (roter Graph).

Myonen bilden im Standardmodell der Teilchenphysik zusammen mit Elektronen und Tauonen die Gruppe der Leptonen. Im Gegensatz zum Elektron ist das Myon jedoch nicht stabil. Es zerfällt mit einer Halbwertszeit von 1,52 µs in ein Elektron, in ein Elektron-Antineutrino und in ein Myon-Neutrino. In Teilchenbeschleunigern beobachtet man für Myonen mit Geschwindigkeiten nahe der Lichtgeschwindigkeit jedoch eine wesentlich größere Halbwertszeit als 1,52 µs. Dies ist eine Folge der relativistischen Zeitdilatation.

Die Gleichung für die Zeitdilatation lautet:

$$t_r = \gamma \cdot t_b \quad \text{mit dem} \quad \text{Verzerrungsfaktor } \gamma = \frac{1}{\sqrt{1-\frac{v^2}{c^2}}}$$

t_r ist die Zeit, die im ruhenden System gemessen wird und t_b die Zeit, die im bewegten System vergeht.

Da wir auf unserer Tastatur kein γ-Taste haben, ersetzen wir in unserem Sketch γ durch Z und schreiben:

```
Z = 1/(sqrt(1-((v*v)/(c*c))));
```

v ist die Geschwindigkeit der Myonen, die der ruhende Beobachter im Laborsystem misst. Die Lichtgeschwindigkeit c setzen wir in unserer Pixelwelt einfachheitshalber auf c = 3.

Mithilfe unseres Sketches wollen wir nun das Zerfallsgesetz für ruhende und bewegte Myonen in einem Diagramm darstellen. Das Zerfallsgesetz $N(t) = N(0) \cdot e^{-\lambda t}$ beschreibt die exponentielle Abnahme der noch nicht zerfallenen Teilchen. In unserem Fall also die der Myonen. Für die Zerfallskonstante λ schreiben wir in unserem Sketch ein K. K berechnet sich mithilfe der Halbwertszeit T_H wie folgt: $K = \frac{ln2}{T_H}$

Würde ein Beobachter mit den Myonen mitfliegen, so würde er eine Halbwertszeit von T_{HB} = 1,52 µs messen, da die Myonen ja relativ zu ihm ruhen (blauer Graph in Abbildung 9.5). Der ruhende Beobachter im Labor misst dagegen eine größere Halbwertszeit T_{HR}, da der Verzerrungsfaktor Z für $v \neq 0$ größer 1 ist (roter Graph).

Wenn man dies verstanden hat, dann ist die Programmierung des Sketches nicht sonderlich schwierig. Man muss nur daran denken, dass die positive y-Achse bei Processing nach unten und die negative y-Achse nach oben zeigt. Aus diesem Grund schreiben wir $N(t) = -N(0) \cdot e^{-\lambda t}$ und verschieben den Koordinatenursprung.

Sketch 05: Halbwertszeit von Myonen

```
// Halbwertszeit von Myonen

float N0 = 500; // Anzahl der Elementarteilchen zur Zeit tB = tR = 0
float N1; // Anzahl der noch nicht zerfallenen Elementarteilchen
          // gemessen im bewegten System (blauer Graph)
float N2; // Anzahl der noch nicht zerfallenen Elementarteilchen
          // gemessen im ruhenden System (roter Graph)
float K1 = 0.4560179; // Zerfallskonstante im bewegten System vom
                      // mitbewegten Beobachter gemessen (blau)
float K2; // Vom ruhenden Beobachter gemessene Zerfallskonstante im
          // bewegten System (rot)
float THB; // Halbwertszeit im bewegten Bezugssystems vom mitbewegten
           // Beobachter gemessen
float THR; // Halbwertszeit vom ruhenden Beobachter gemessen
float v = 2.99; // Relativgeschwindigkeit der Myonen
float c = 3; // Lichtgeschwindigkeit in unserer Pixelwelt
float Z; // Verzerrungsfaktor


void setup()
{
  size(600, 600);

  Z = 1/(sqrt(1-((v*v)/(c*c)))); // Verzerrungsfaktor Z wird berechnet
  THB = log(2)/K1; // Berechnung der Halbwertszeit für das bewegte
                   // System
  THR = Z*THB; // Berechnung der Halbwertszeit für das ruhende System
  K2 = log(2)/THR; // Berechnung der Zerfallskonstante für das ruhende System
}

void draw()
{
  background(255);

  // Koordinatensystem
  stroke(0);
  strokeWeight(3);
  line(45, 550, 600, 550);
  line(50, 0, 50, 555);
  fill(0);
  triangle(45, 20, 50, 0, 55, 20);
  triangle(580, 545, 600, 550, 580, 555);
  textSize (32);
  text("N(t) in 10^6", 65, 40);
  text("t in µs", 480, 540);
```

```
  textSize(20);
  text("500", 8, 60);
  text("0", 25, 558);
  text("0", 43, 585);
  line(150, 550, 150, 560);
  text("10", 135, 585);
  line(250, 550, 250, 560);
  text("20", 235, 585);
  line(350, 550, 350, 560);
  text("30", 335, 585);
  line(450, 550, 450, 560);
  text("40", 435, 585);
  line(550, 550, 550, 560);
  text("50", 535, 585);

  // Linie für die Halbwertszeit
  stroke(0, 255, 0);
  strokeWeight(2);
  line(50, 300, 600, 300);
  fill(0, 200, 0);
  textSize(20);
  text("250", 8, 307);


  translate(50, 550); // Verschiebung des Koordinatenursprungs

  for (float t = 0; t < 600; t = t + 0.01)
  {
    // Funktionen für die exponentielle Abnahme
    N1 = -N0*exp(-K1*t); // bewegtes System (blau)
    N2 = -N0*exp(-K2*t); // ruhendes System (rot)

    // Die Graphen werden gezeichnet
    stroke(0, 0, 255);
    strokeWeight(5);
    point(10*t, N1);
    stroke(255, 0, 0);
    point(10*t, N2);
  }

  // Beschriftung
  fill(0);
  textSize(24);
  text("c = 3   v = " +v, 300, -450);
  fill(0, 0, 255);
  text("THB in µs = " +(float)round(100*THB)/100, 300, -400);
  fill(255, 0, 0);
  text("THR in µs = " +(float)round(100*THR)/100, 300, -350);
}
```

9.6 Paarbildung und Zerstrahlung

Paarbildung

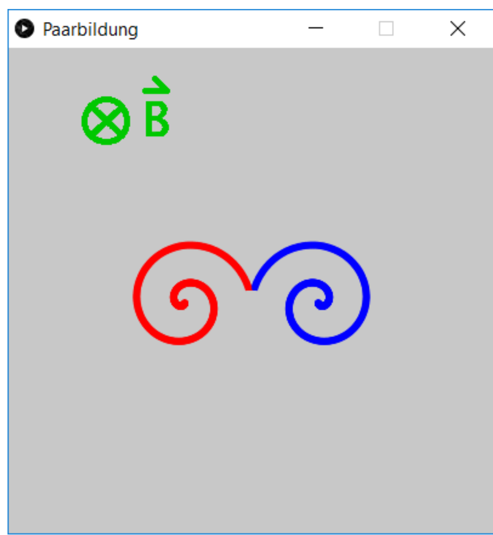


Abbildung 9.6: Spiralförmige Bahnen von Positron (rot) und Elektron (blau)

Ein Elektron-Positron-Paar kann entstehen, wenn ein energiereiches Photon (Gammaquant) in das elektrische Feld eines schweren Atomkerns eindringt. Das Gammaquant verwandelt sich hierbei in Materie (Elektron) und Antimaterie (Positron) und in Bewegungsenergie der beiden Teilchen. Einen kleinen Teil der Energie des Photons nimmt auch der Atomkern auf. Die Spur des Elektrons und die des Positrons kann man in einer Nebelkammer sichtbar machen. Legt man an die Nebelkammer ein Magnetfeld an, dann beobachtet man zwei spiralförmige Nebelspuren.

In unserem Sketch wollen wir den oben beschriebenen Vorgang ohne Berücksichtigung des Atomkerns sichtbar machen. Dazu müssen wir uns nochmal in Erinnerung rufen, wie man eine Spiralbahn programmiert. Hier hilft der Sketch *Zyklotron* aus dem Kapitel 4.3, obwohl hier mit zunehmendem Winkel der Radius immer größer wird. In unserem Sketch *Paarbildung* sollen die Radien der beiden Teilchenbahnen aufgrund von Energieverlusten jedoch abnehmen (siehe Abb. 9.6). Dies dürfte uns aber keine Probleme bereiten. Auch das Verschwinden des Photons bei der Paarentstehung gelingt ganz einfach. Im Moment der Paarentstehung nimmt es die Farbe des Hintergrundes an. Die anderen Zeilen des Sketches sind ebenfalls leicht zu verstehen, sodass an dieser Stelle keine weiteren Erklärungen mehr notwendig sind.

Sketch 06: Paarbildung

```
// Paarbildung

float w; // Winkel im Bogenmaß
float dw = 0.04; // Winkelzunahme im Bogenmaß
float r = -55.0; // Radius
float dr = -0.2; // Abnahme von r
float x1; // x-Wert für das Elektron
float y1; // y-Wert für das Elektron
float x2; // x-Wert für das Positron
float y2; // y-Wert für das Positron
float y3 = 400; // Startwert Gammaquant
```

```
void setup()
{
  size(400, 400);
}

void draw()
{
  // Gammaquant wird gezeichnet
  if (y3 > 210)
  {
    background(200);
    noStroke();
    fill(255, 255, 0);
    ellipse(200, y3, 10, 10);
    y3 = y3 - 3;
  } else
  {
    if (w <= 3.5*PI && w >= 0)
    {
      // Gammaquant verschwindet
      noStroke();
      fill(200);
      ellipse(200, 210, 20, 20);

      // Spiralbahn für das Elektron wird berechnet
      x1 = 255 + r * cos(w);
      y1 = 210 + r * sin(w);

      // Spiralbahn für das Positron wird berechnet
      x2 = 145 - r * cos(w);
      y2 = 210 + r * sin(w);

      w = w + dw; // Winkel wird vergrößert
      r = r - dr; // Radius wird verkleinert

      // Die Spiralbahnen werden gezeichnet
      noStroke();
      fill(0, 0, 255);
      ellipse(x1, y1, 6, 6);
      fill(255, 0, 0);
      ellipse(x2, y2, 6, 6);
    }
  }

  // Zeichnen der Angaben zum B-Feld
  fill(0, 200, 0);
  textSize(40);
  text("B", 110, 73);
  stroke(0, 200, 0);
  strokeWeight(4);
  noFill();
  ellipse(80, 60, 35, 35);
  line(67, 72, 93, 48);
  line(67, 47, 93, 72);
  line(112, 35, 130, 35);
  line(130, 35, 120, 25);
}
```

Paarzerstrahlung

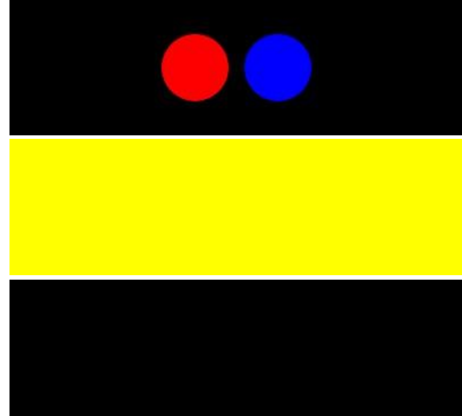


Abbildung 9.7: Positron und Elektron bewegen sich aufeinander zu und zerstrahlen

Der entgegengesetzte Prozess zur Paarbildung ist der Prozess der Paarzerstrahlung. Stößt ein Positron mit einem Elektron zusammen, so zerstrahlen beide in Energie. D.h., es bilden sich aufgrund des Impulserhaltungssatzes mindestens zwei Gammaquanten.

Diesen Vorgang wollen wir in unserem Sketch wie folgt darstellen. Ein Positron (rot) und ein Elektron (blau) bewegen sich aufeinander zu (Abb. 9.7 oben). Im Moment des Zusammenstoßes wird der gesamte Raum von einem Gammablitz erfüllt (Abb. 9.7 mittig). Danach herrscht Dunkelheit (Abb. 9.7 unten). Den zeitlichen Ablauf wollen wir mit der Uhr des Computers und der Funktion *millis()* steuern. Die Funktion *millis()* gibt die Zeit in Millisekunden an, die seit dem Start des Sketches vergangen sind. Mit der Variablen t können wir dann steuern, wie lange der Lichtblitz zu sehen ist. Mehr muss zum Sketch *Zerstrahlung* nicht erklärt werden.

Sketch 07: Zerstrahlung

```
// Zerstrahlung von Positron und Elektron

int t = 1600; // Zeit in Millisekunden
float x1 = 60; // Startwert Positron
float x2 = 340; // Startwert Elektron

void setup()
{
  size(400, 120);
}

void draw()
{
  background(0);

  // Positron
  fill(255, 0, 0);
  ellipse(x1, 60, 60, 60);

  // Elektron
  fill(0, 0, 255);
  ellipse(x2, 60, 60, 60);
```

```
  // Bewegung von Positron und Elektron
  x1 += 1.5;
  x2 -= 1.5;

  int ComputerUhr = millis();
  if (ComputerUhr > t)
  {
    background(255, 255, 0); // Zerstrahlung
  }
  if (ComputerUhr > 1.2*t)
  {
    background(0);
  }
}
```

9.7 Zusammenfassung

verblassen Möchte man eine gezeichnete Form im Fenster langsam verblassen lassen, so fügt man bei *void draw()* anstelle von *background()* ein fenstergroßes, transparentes Rechteck ein. Bei jedem Durchlauf von *void draw()* wird die gezeichnete Form erneut von dem transparenten Rechteck überdeckt, bis sie schließlich nicht mehr sichtbar ist.

strokeCap(ROUND) Mit *strokeCap(ROUND)* kann man die Enden von Linien abrunden.

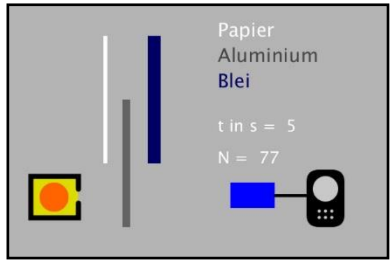
Rechteck mit abgerundeten Kanten

Möchte man ein Rechteck mit abgerundeten Ecken zeichnen, dann schreibt man bei seiner Konstruktion eine fünfte Zahl in die Klammer. Beispiel: `rect(270, 60, 60, 90, 20)` Mit der fünften Zahl bestimmt man die Radien für die abgerundeten Ecken des Rechtecks.

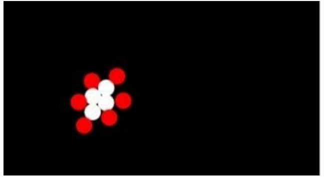
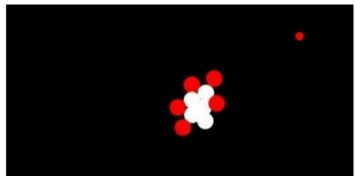
9.8 Aufgaben

1. Im Sketch *Nebelkammer* erschienen die Teilchenbahnen als Ganzes und wurden danach immer blasser. In dem nun zu schreibenden Sketch sollen die Teilchen einzeln erscheinen, sich auf gekrümmten Bahnen bewegen und einen verblassenden Schweif hinter sich herziehen (siehe Abbildung rechts). Sobald ein Teilchen den Fensterrand erreicht, soll im Fenster an einem zufälligen Ort ein neues Teilchen erscheinen.

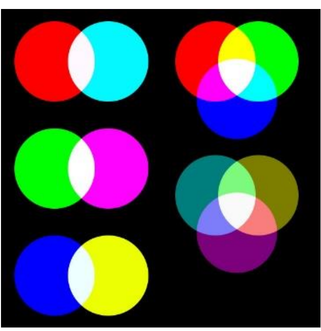
2. Der Sketch *Nulleffekt* soll wie folgt geändert werden. Links im Fenster soll sich in einem Bleibehälter mit einer Öffnung ein radioaktives Präparat befinden, welches α-, β- und γ-Strahlung aussendet. In den Strahlengang kann ein Blatt Papier, eine Aluminiumplatte oder eine Bleiplatte gestellt werden (siehe Abbildung). Im Fenster sollen die sich ergebenden jeweiligen Zählraten N für 5 Sekunden entsprechend der eingestellten Absorber qualitativ richtig angezeigt werden.



3. Ändere den Sketch *Betaminuszerfall* so um, dass man aufgrund einer zu großen Anzahl von Protonen im Atomkern einen Betapluszerfall beobachten kann (siehe Abbildung).

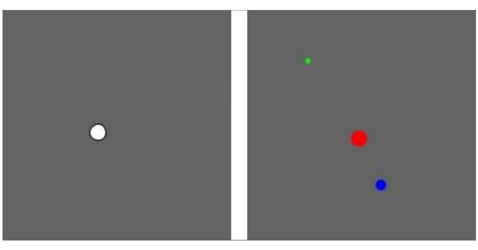
4. Erinnern wir uns daran was wir im Kapitel Farbmischungen gelernt haben. Dieses Wissen können wir nun benutzen, um aus Quarks aufgebaute Mesonen und Baryonen darzustellen. Diese Teilchen halten aufgrund der Farbladung der Quarks zusammen. Die Farbladung dieser aus Quarks zusammengesetzten Teilchen muss jedoch immer die Farbe Weiß ergeben. In der Abbildung rechts sehen wir drei mögliche Farbkombinationen für Mesonen, die aus einem Quark und einem Antiquark bestehen. Die Farben Rot, Grün und Blau ergeben gemischt mit ihren entsprechenden Komplementärfarben Cyan, Magenta und Gelb die Farbe Weiß. Die drei Quarks des Baryons tragen die Farben Rot, Grün und Blau. Die drei Quarks des Antibaryons tragen die Farben Cyan, Magenta und Gelb. Schreibe einen Sketch, der die Abbildung rechts oben generiert.



   Tipp: Benutze den *Color Selector* von Processing.

5. Ein freies Neutron dringt in ein homogenes Magnetfeld ein und verwandelt sich in ein Elektron und ein Proton (ß$^-$-Zerfall). Das entstehende Antineutrino soll hierbei vernachlässigt werden. Schreibe einen Sketch, der diesen Sachverhalt in einem 800 Pixel breiten und 200 Pixel hohen Fenster qualitativ simuliert (siehe Abbildung). Berechne die Teilchenbahnen vektoriell. Verwende vereinfachend die folgenden Werte: $m_{Elektron}$ = 1, m_{Proton} = 1836,1, $m_{Neutron}$ = 1838,7 sowie $q_{Elektron}$ = -1, q_{Proton} = 1; q_{Neuton} = 0. Beachte, dass das Elektron aufgrund seiner geringen Masse wesentlich stärker abgelenkt wird als das Proton. Um eine gute Darstellung zu erhalten, simuliere diesen ß$^-$-Zerfall mit kleinen Zeitschritten und einer hohen Bildwiederholungsrate.

6. Wie in Aufgabe 5 dringt ein freies Neutron in ein homogenes Magnetfeld ein und verwandelt sich in ein Elektron, ein Proton und ein Antineutrino. Das entstehende Antineutrino soll hierbei im Gegensatz zu Aufgabe 5 nicht vernachlässigt werden. Auch soll berücksichtig werden, dass das Elektron und das Proton aufgrund ihrer Bewegung auf einer Kreisbahn Energie verlieren (beschleunigte Ladungen strahlen). Schreibe einen Sketch, der diesen Sachverhalt in einem 800 Pixel breiten und 800 Pixel hohen Fenster qualitativ simuliert. In diesem Fenster sollen die Teilchen als Einzelobjekt und nicht wie bei Aufgabe 5 als Teilchenbahnen zu sehen sein. Damit die charakteristischen Bewegungen der Teilchen im Fenster besser beobachtet werden können, müssen die Massenunterschiede zwischen den Teilchen aber deutlich reduziert werden. Die Abbildung oben zeigt zwei Ausschnitte aus dem 800 x 800 großen Fenster. Links das Neutron bevor es zerfallen ist und rechts die Zerfalls-produkte.



10 Relativitätstheorie

Was erwartet uns?

Wiederholung von: PImage, loadImage(), pushMatrix(), popMatrix(), Algorithmus von Dan Bruton, round(),

10.1 Massenzunahme, Längenkontraktion und Zeitdilatation

Abbildung 10.1: Simulation von Zeitdilatation, Massenzunahme und Längenkontraktion

Menschen waren und sind von den Aussagen der speziellen Relativitätstheorie von Albert Einstein immer wieder aufs Neue fasziniert, da Zeitdilatation, Massenzunahme und Längenkontraktion ihren Alltagserfahrungen widersprechen. Mathematisch betrachtet ist die spezielle Relativitätstheorie jedoch so einfach, dass sie im normalen Schulunterricht behandelt werden kann.

Für die drei Größen Zeitdilatation, Massenzunahme und Längenkontraktion wollen wir nun in Processing eine Simulation schreiben. Die hierzu notwendigen Gleichungen lauten:

Zeitdilatation: $\Delta t_r = \gamma \cdot \Delta t_b$ Massenänderung: $m_r = \gamma \cdot m_b$ Längenänderung: $l_r = \frac{l_b}{\gamma}$

Verzerrungsfaktor: $\gamma = \frac{1}{\sqrt{1-\frac{v^2}{c^2}}}$

In unserem Sketch ersetzen wir γ durch Z und schreiben:

```
tr = tb*Z; // Zeitdilatation
mr = mb*Z; // Massenzunahme
lr = lb*(1/Z); // Längenkontraktion
Z = 1/(sqrt(1-((v*v)/(c*c)))); // Verzerrungsfaktor
```

Die mit dem Index b versehenen Werte, sind die Werte, die die Besatzung der Rakete misst. Die mit dem Index r versehenen Werte sind die Werte, die der ruhende Beobachter misst (siehe Abb. 10.1).

Die Lichtgeschwindigkeit c setzen wir in unserer Pixelwelt auf den Wert 3. Die Ruhemasse mr und die bewegte Masse mb geben wir im Fenster nur als Zahlenwerte an (Abb. 10.1). Die vom ruhenden und bewegten Beobachter gemessene Länge der Rakete in Bewegungsrichtung stellen wir jedoch bildlich und als Zahlenwerte dar. Dazu speichern wir in unserem Sketchordner das Bild einer Rakete ab. Mit *PImage Rakete* deklarieren und mit *Rakete = loadImage("Rakete.jpg")* laden wir das Bild der Rakete in unseren Sketch.

Den unterschiedlichen Ablauf der Zeit stellen wir mit zwei unterschiedlich schnell rotierenden Zeigern dar. Der rote Zeiger zeigt die Zeit an, die die Besatzung in der Rakete misst. Der grüne Zeiger zeigt die Zeit an, die der ruhende Beobachter misst. Für die voneinander unabhängigen Rotationsgeschwindigkeiten der beiden Zeiger benutzen wir, wie schon im Sketch *Uhr* von Kapitel 2.4.2, die beiden Funktionen **pushMatrix()** und **popMatrix()**. *pushMatrix()* speichert das aktuelle Koordinatensystem und *popMatrix()* lädt das gespeicherte Koordinatensystem wieder und vergisst alles, was zwischen *pushMatrix()* und *popMatrix()* gestanden hat. Somit addiert sich in unserem Sketch die Rotation des roten Zeigers nicht zur Rotation des grünen Zeigers.

Nun können wir mit unterschiedlichen Werten für v mit unserer Simulation „spielen“. Am Ende des Sketches sorgen wir aber mit einer if-Anweisung dafür, dass für Relativgeschwindigkeiten v größer oder gleich c keine Simulation erfolgt, da nach den Aussagen der Relativitätstheorie solche Geschwindigkeiten nicht erreicht werden können.

Sketch 01: Relativitätstheorie

```
// Relativitätstheorie

PImage Rakete; // Bild der Rakete
float v = 2.5; // Relativgeschwindigkeit der Rakete
float c = 3; // Lichtgeschwindigkeit in unserer Pixelwelt
float Z; // Verzerrungsfaktor
float x = 0; // Ort der Rakete
float tr; // Zeit im Ruhesystem
float tb = 0.01; // Bordzeit (bewegtes System)
float mr; // Masse im Ruhesystem gemessen
float mb = 50; // Masse an Bord gemessen (bewegtes System)
float lr; // Raketenlänge im Ruhesystem gemessen
float lb = 100; // Raketenlänge von der Besatzung gemessen
float a; // Rotationsfaktor für den roten Zeiger (Ruhesystem des
         // Beobachters)
float b; // Rotationsfaktor für den grünen Zeiger (bewegtes System,
         // Borduhr)
float dt = 1; // dt ist die Zeit zwischen zwei Bildern der Simulation

void setup()
{
  size(800, 400);

  // Bild der Rakete wird geladen
  Rakete = loadImage("Rakete.jpg");

  // Verzerrungsfaktor Z wird berechnet
  Z = 1/(sqrt(1-((v*v)/(c*c))));
```

```
  mr = Z*mb; // Massenzunahme
  lr = lb*(1/Z); // Längenkontraktion
  tr = tb*Z; // Zeitdilatation
}

void draw()
{
  background(100, 100, 255);

  // Beschriftung
  fill(100, 255, 100); // Textfarbe Ruhesystem
  textSize(24); // Textgröße
  text("Ruhesystem: grüner Uhrzeiger", 10, 40);
  fill(255, 200, 200); // Textfarbe bewegtes System
  text("bewegtes System: roter Uhrzeiger (Borduhr)", 10, 80);
  fill(255); // Textfarbe
  text("c = 3   v = " +v, 10, 120);
  fill(255, 200, 200); // Textfarbe
  text("mb = " +mb, 10, 160);
  text("lb = " +lb, 10, 200);
  fill(100, 255, 100); // Textfarbe
  text("mr = " +mr, 160, 160);
  text("lr = " +lr, 160, 200);

  // Beobachter im Ruhesystem
  stroke(100, 255, 100);
  noFill();
  ellipse(400, 360, 15, 15);
  line(400, 368, 400, 380);
  line(400, 380, 390, 400);
  line(400, 380, 410, 400);
  line(400, 375, 380, 365);
  line(400, 375, 420, 365);

  // Darstellung der Längenkontraktion
  // Bild der Rakete wird eingefügt.
  image(Rakete, x, 270, lr, 40);
  x = x + v*dt; // Bewegung der Rakete in x-Richtung

  /* Wenn die Rakete rechts das Fenster verlässt,
   dann erscheint sie wieder links im Fenster */
  if (x > width)
  {
    x = -lr;
  }

  // Die Uhr im Ruhesystem wird erzeugt.
  translate(650, 100); // Koordinatenursprung wird verschoben.

  pushMatrix(); // Speichert die obige Transformation.

  // Kreis
  stroke(100);
  fill(230);
  ellipse(0, 0, 125, 125);

  // grüner Pfeil
  rotate(a*PI/30.0);
  a = a + 8.0*tr; // Rotationsgeschwindigkeit des Zeigers im Ruhesystem.
```

```
  stroke(0, 255, 0);
  strokeWeight(3);
  line(0, 0, 60, 0);
  fill(0, 255, 0);
  triangle(50, -5, 60, 0, 50, 5); /* Pfeilspitze liegt auf der
           Rechtswertachse des verschobenen Koordinatensystems */

  popMatrix(); /* Lädt die oben gespeicherte Transformation wieder.
   Somit addiert sich die folgende Rotation nicht zur obigen Rotation */

  // Die Uhr im bewegten System wird erzeugt (Borduhr)
  // roter Pfeil
  rotate(b*PI/30.0);
  b = b + 8.0*tb; /* Rotationsgeschwindigkeit des Zeigers im bewegten
                     System */

  stroke(255, 0, 0);
  strokeWeight(3);
  line(0, 0, 50, 0);
  fill(255, 0, 0);
  triangle(40, -5, 50, 0, 40, 5);/* Pfeilspitze liegt auf der
  Rechtswertachse des verschobenen Koordinatensystems */

  // Wenn v >= c wird die Simulation nicht ausgeführt
  if (v >= c)
  {
    fill(255, 0, 0);
    textSize(60);
    text("v größer oder gleich c", -580, 200);
    noLoop();
  }
}
```

10.2 Rotverschiebung

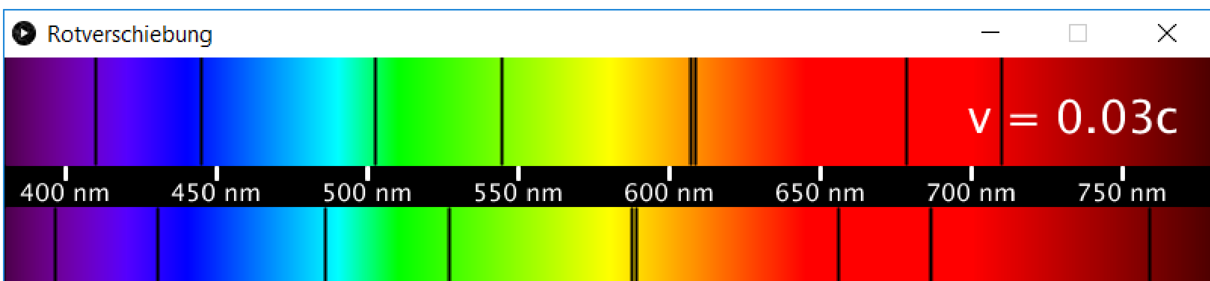


Abbildung 10.2: Hauptabsorptionslinien im Sonnenspektrum, oben rotverschobenes Spektrum

Eine Rotverschiebung von Spektren kann mehrere Ursachen haben. Eine Ursache kann die Relativbewegung zwischen Sender und Empfänger sein. Weitere Ursachen können die Ausdehnung der Raumzeit sowie gravitative Effekte sein. In unserem Sketch betrachten wir nur die Relativbewegung zwischen Sender und Empfänger, also den optischen Dopplereffekt. Wenn sich der Sender S vom Empfänger E entfernt, dann lautet die Gleichung:

$$f_E = f_S \cdot \sqrt{\frac{c-v}{c+v}} \quad \text{mit } f = \frac{c}{\lambda} \quad \text{folgt} \quad \lambda_E = \frac{\lambda_S}{\sqrt{\frac{c-v}{c+v}}}$$

Wenn wir nun auf der als ruhend betrachteten Erde das rotverschobene Spektrum eines Sterns betrachten, der sich von der Erde fortbewegt, dann können wir anhand der rotverschobenen Spektrallinien bestimmen, mit welcher Geschwindigkeit er sich von der Erde fortbewegt. Zum Vergleich benötigen wir aber ein nicht verschobenes Vergleichsspektrum (unten in Abbildung 10.2).

In unserem Sketch geben wir jedoch die Fluchtgeschwindigkeit vor und schauen uns dann im Fenster die Verschiebung des Spektrums an. Als Vergleichsspektrum dienen uns dazu die Haupt-absorptionslinien im Spektrum unserer Sonne λ_1 = 759,37 nm, λ_2 = 686,72 nm, λ_3 = 656,28 nm; λ_4 = 589,0 nm, λ_5 = 587,56 nm, λ_6 = 527,04 nm, λ_7 = 486,13 nm, λ_8 = 430,79 nm und λ_9 = 396,85 nm (siehe Abb. 10.2 unten). Das den Absorptionslinien untergelegte kontinuierliche Spektrum erstellen wir, wie im Sketch *Hg-Absorptionsspektrum_RGB* im Kapitel 6.7 beschrieben, mit dem Algorithmus von Dan Bruton.

Auch an dieser Stelle soll noch ein letzes Mal die Bedeutung der folgenden Zeile erklärt werden.

```
text("v = " +(float)round(100*v/3)/100, 700, 50);
```

Sie dient dazu, im Fenster die Fluchtgeschwindigkeit v relativ zur Lichtgeschwindigkeit c auf nur zwei Stellen hinter dem Komma anzugeben. Würden wir einfach v/3 schreiben, so würden wir viele Nachkommastellen erhalten. Um dies zu vermeiden, multiplizieren wir den Quotienten mit 100. Dadurch rutscht das Komma zwei Stellen nach rechts. Dann runden wir auf einen ganzzahligen Wert und teilen diesen anschließend durch 100. So erhalten wir eine float-Zahl mit zwei Nachkommastellen. Damit dürfte der Sketch hinreichend erklärt sein

Sketch 02: Rotverschiebung

```
// Rotverschiebung

float c = 3; // Lichtgeschwindigkeit in unserer Pixelwelt
float v = 0.1; // Fluchtgeschwindigkeit
int L; // Wellenlänge
float F; // Faktor
float rot;
float gruen;
float blau;

// Wellenlänge des Senders im bewegten System
float LS1 = 759.37;
float LS2 = 686.72;
float LS3 = 656.28;
float LS4 = 589.0;
float LS5 = 587.56;
float LS6 = 527.04;
float LS7 = 486.13;
float LS8 = 430.79;
float LS9 = 396.85;

// Empfangene Wellenlängen im ruhenden System
float LE1 = LS1/sqrt((c-v)/(c+v));
float LE2 = LS2/sqrt((c-v)/(c+v));
float LE3 = LS3/sqrt((c-v)/(c+v));
```

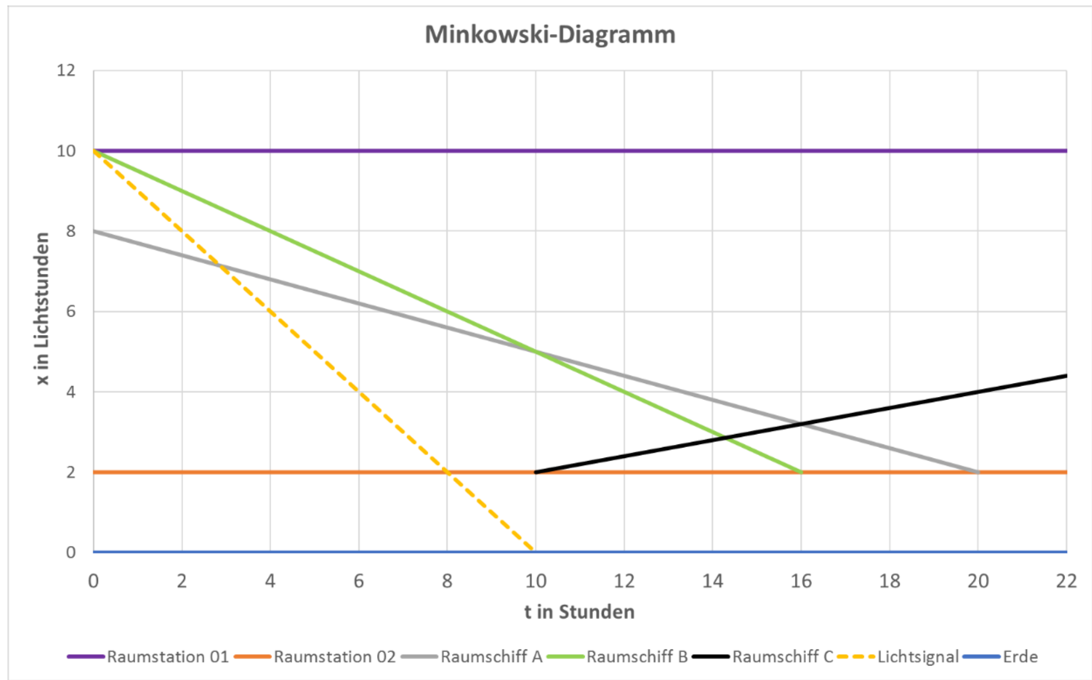
```
float LE4 = LS4/sqrt((c-v)/(c+v));
float LE5 = LS5/sqrt((c-v)/(c+v));
float LE6 = LS6/sqrt((c-v)/(c+v));
float LE7 = LS7/sqrt((c-v)/(c+v));
float LE8 = LS8/sqrt((c-v)/(c+v));
float LE9 = LS9/sqrt((c-v)/(c+v));

void setup()
{
  size(800, 150);
  background(0);

  // Achsenbeschriftung
  stroke(255);
  strokeWeight(3);
  line(2*400-760, 73, 2*400-760, 80);
  line(2*450-760, 73, 2*450-760, 80);
  line(2*500-760, 73, 2*500-760, 80);
  line(2*550-760, 73, 2*550-760, 80);
  line(2*600-760, 73, 2*600-760, 80);
  line(2*650-760, 73, 2*650-760, 80);
  line(2*700-760, 73, 2*700-760, 80);
  line(2*750-760, 73, 2*750-760, 80);

  fill(255);
  textSize(16);
  textAlign(CENTER);
  text("400 nm", 2*400-760, 95);
  text("450 nm", 2*450-760, 95);
  text("500 nm", 2*500-760, 95);
  text("550 nm", 2*550-760, 95);
  text("600 nm", 2*600-760, 95);
  text("650 nm", 2*650-760, 95);
  text("700 nm", 2*700-760, 95);
  text("750 nm", 2*750-760, 95);
}

void draw()
{
// Das kontinuierliche Spektrum wird gezeichnet
  for (float L = 380; L >= 380 && L <= 781; L = L + 0.5)
  {
    stroke(F*rot, F*gruen, F*blau);

    if (L >= 380 && L < 440)
    {
      rot = -255*((L - 440.0) / (440.0 - 380.0));
      gruen = 0.0;
      blau = 255.0;
    } else if (L >= 440 && L < 490)
    {
      rot = 0.0;
      gruen = 255*((L - 440.0) / (490.0 - 440.0));
      blau = 255;
    } else if (L >= 490 &&  L < 510)
    {
      rot = 0.0;
      gruen = 255;
      blau = -255*((L - 510.0) / (510.0 - 490.0));
    } else if (L >= 510 && L < 580)
    {
```

```
    rot = 255*((L - 510.0) / (580.0 - 510.0));
    gruen = 255;
    blau = 0.0;
  } else if (L >= 580 && L < 645)
  {
    rot = 255;
    gruen = -255*((L - 645.0) / (645.0 - 580.0));
    blau = 0.0;
  } else if (L >= 645 && L < 781)
  {
    rot = 255;
    gruen = 0.0;
    blau = 0.0;
  } else
  {
    rot = 0.0;
    gruen = 0.0;
    blau = 0.0;
  }

  if (L >= 380 && L < 420)
  {
    F = 0.3 + 0.7 * (L - 380.0) / (420.0 - 380.0);
  } else if (L >= 420 && L < 701)
  {
    F = 1.0;
  } else if (L >= 701 && L < 781)
  {
    F = 0.3 + 0.7 * (780.0 - L) / (780.0 - 700.0);
  } else
  {
    F = 0.0;
  }

  line(2*L-760, 70, 2*L-760, 0);
  line(2*L-760, 100, 2*L-760, 150);
}

// rotverschobene Absorptionslinien
stroke(0);
strokeWeight(2);
line(2*LE1-760, 70, 2*LE1-760, 0);
line(2*LE2-760, 70, 2*LE2-760, 0);
line(2*LE3-760, 70, 2*LE3-760, 0);
line(2*LE4-760, 70, 2*LE4-760, 0);
line(2*LE5-760, 70, 2*LE5-760, 0);
line(2*LE6-760, 70, 2*LE6-760, 0);
line(2*LE7-760, 70, 2*LE7-760, 0);
line(2*LE8-760, 70, 2*LE8-760, 0);
line(2*LE9-760, 70, 2*LE9-760, 0);

// nicht verschobene Absorptionslinien
line(2*LS1-760, 100, 2*LS1-760, 150);
line(2*LS2-760, 100, 2*LS2-760, 150);
line(2*LS3-760, 100, 2*LS3-760, 150);
line(2*LS4-760, 100, 2*LS4-760, 150);
line(2*LS5-760, 100, 2*LS5-760, 150);
line(2*LS6-760, 100, 2*LS6-760, 150);
line(2*LS7-760, 100, 2*LS7-760, 150);
line(2*LS8-760, 100, 2*LS8-760, 150);
line(2*LS9-760, 100, 2*LS9-760, 150);
```

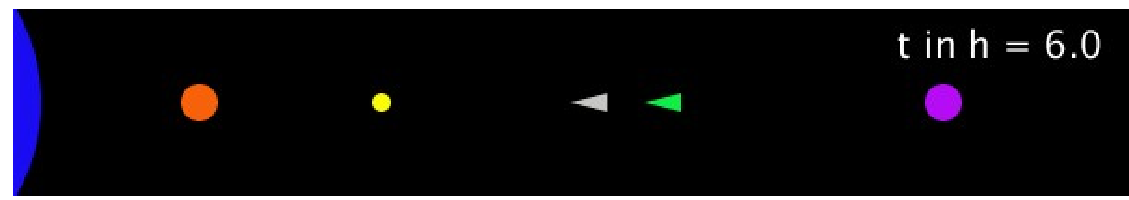
```
    // Text
    fill(255);
    textSize(30);
    text("v = " +(float)round(100*v/3)/100, 700, 50);
    text("c", 770, 50);
}
```

10.3 Aufgaben

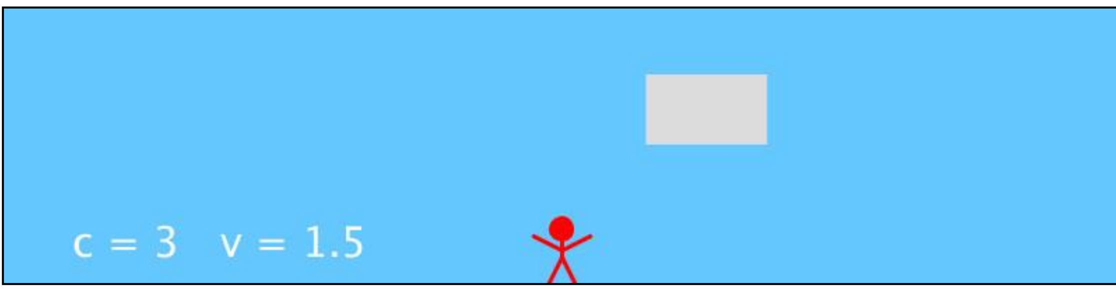
1. Ein Minkowski-Diagramm ist ein Raum-Zeit-Diagramm, welches zur Darstellung von Vorgängen im Rahmen der speziellen Relativitätstheorie benutzt wird. Es besitzt der Einfachheit halber nur eine Raumdimension x. Üblicherweise wird die Zeit auf der Hochwertachse und die Raumdimension x auf der Rechtswertachse aufgetragen. In der Abbildung unten tragen wir aber die Zeit auf der Rechtswertachse auf. So erhalten wir ein aus der Mechanik vertrautes t-x-Diagramm. In der Abbildung wurde die Zeit in Stunden und an der Hochwertachse der Ort in Lichtstunden aufgetragen. Dies hat zur Folge, dass die Geradensteigung von Lichtsignalen 45° beträgt. Die Graphen von bewegten Objekten im Minkowski-Diagramm nennt man Weltlinien. Solche Weltlinien können niemals eine Steigung von größer 45° besitzen, da sich nichts schneller als das Licht bewegen kann. Schreibe einen Sketch, der die Bewegungsvorgänge in dem unten eingefügten Minkowski-Diagramm simuliert.



Bedenke, dass sich alle Körper in dem obigen Diagramm auf einer Linie befinden. Eine mögliche Lösung zeigt die Abbildung unten. Die Erde (blau) ist nur symbolisch dargestellt. Die Zeit wird rechts oben angezeigt. Erstelle mit *saveFrame()* eine Bildfolge von deiner Simulation und überprüfe anhand dieser Bilder, ob die Begegnungen der einzelnen Objekte (Lichtsignal, Raumschiffe, rote Raumstation) auch zu den im obigen Diagramm angegebenen Zeiten erfolgen.



2. Ein weißer Ziegelstein fliegt von links nach rechts durch das Fenster. Bei sehr großen Geschwindigkeiten beobachtet man von einem ruhenden System aus, dass der Ziegelstein eine Längenkontraktion in Bewegungsrichtung erfährt und seine Masse zunimmt. Schreibe einen Sketch, der diesen Sachverhalt wiedergibt. Die Massenzunahme soll so dargestellt werden, dass die Farbe des Ziegelsteins sich von Weiß (v = 0) nach Schwarz (v ≈ c) ändert.



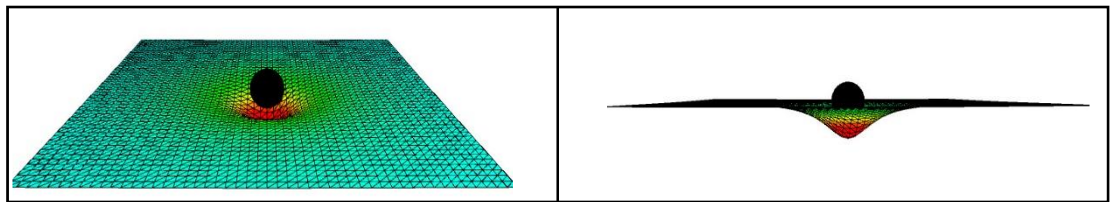
3. Nach der allgemeinen Relativitätstheorie beeinflussen große Massen den Zeitablauf. Bringt man zum Beispiel eine Uhr A in die Nähe einer großen Masse m, so geht sie langsamer als eine sehr weit entfernte Uhr B. Es gilt die Gleichung $t_A = t_B(1 - \frac{\gamma \cdot m}{c^2 \cdot r})$. Hierbei steht γ für die Gravitationskonstante, m für die Masse des Himmelskörpers (z.B. Schwarzes Loch), r für den Abstand der Uhr A vom Massenmittelpunkt, c für die Lichtgeschwindigkeit und t für die Zeit. Schreibe einen Sketch, mit dem man diesen Sachverhalt simulieren kann (siehe Abbildung unten). Benutze die folgenden Werte: Masse des Schwarzen Loches in der Abbildung $m = 2 \cdot 10^{33}$ kg, Abstand der Uhr vom Mittelpunkt des Schwarzen Loches $r = 3 \cdot 10^6$ m, Lichtgeschwindigkeit $c = 3 \cdot 10^8$ ms^{-1}, Gravitationskonstante $g = 6{,}6726 \cdot 10^{-11}$ $m^3kg^{-1}s^{-2}$ (g steht im Sketch für γ).

   a) Löse die Aufgabe unter der Verwendung von *pushMatrix()* und *popMatrix()*. Siehe Sketch *Relativitätstheorie*.

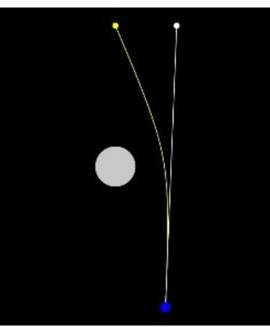
   b) Löse die Aufgabe ohne *pushMatrix()* und *popMatrix()*. Lasse die Zeiger mittels *cos()* und *sin()* rotieren. Auf die Pfeilspitzen kann hierbei verzichtet werden.

4. Nach Albert Einstein krümmt Masse die Raumzeit. Stelle mit Hilfe eines Sketches diesen Sachverhalt wie unten dargestellt anschaulich dar.

Tipp: Erinnere dich an den Sketch *Potenzialgebirge*.



5. Da bewegte Photonen Energie und damit auch Masse besitzen, können sie durch andere Massen abgelenkt werden. Beobachtet wurde dies zum ersten Mal 1919 bei einer totalen Sonnenfinsternis. Diese Beobachtung verhalf Einsteins allgemeiner Relativitätstheorie zum Durchbruch. Schreibe einen Sketch, mit dem man die Lichtablenkung in einem Gravitationsfeld qualitativ simulieren kann (siehe Abbildung). Der gelbe Punkt in der Abbildung stellt die wirkliche Position des Licht aussenden Sterns dar. Der weiße Punkt gibt den Ort an, an dem ein Beobachter auf der Erde (blauer Punkt) den Stern sieht.



Tipp: Benutze das Gravitationsgesetz und setze die Masse des Photons gleich 1.

11 Chaos und Fraktale

Was erwartet uns?

Vergleich float, double und BigDecimal, logistische Gleichung, strokeCap(SQUARE), rekursive Programmierung, Fraktale aus der Beispielbiliothek

11.1 Deterministisches Chaos

11.1.1 Wege ins Chaos

Gleiche Startwerte

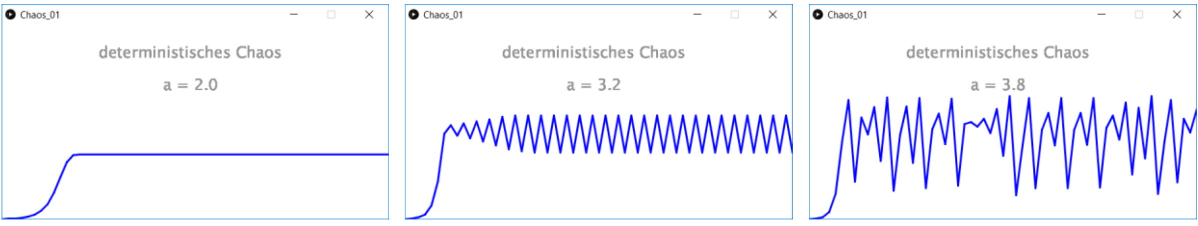


Abbildung 11.1: Wege ins Chaos a = 2,0 ein Attraktor, a = 3,2 zwei Attraktoren, a = 3,8 Chaos

Unter Chaos verstehen die meisten Mensch einen Zustand kompletter Unordnung. Physiker und Mathematiker untersuchen jedoch eine ganz besondere Art von Chaos, das deterministische Chaos. Mit konkreten Gleichungen versuchen sie den Zustand nichtlinearer, dynamischer Systeme wie zum Beispiel Doppelpendel oder nichtlinearem elektrischen Serienschwingkreis langfristig vorrauszuberechnen. Da das Endergebnis aber hier von kleinsten Anfangsstörungen abhängt, gelingt eine langfristige Vorhersage in der Regel nicht.

Das determinitische Chaos untersuchen wir in den folgenden Sketchen mit der logistischen Gleichung

$$x_{n+1} = a \cdot x_n \cdot (1 - x_n)$$

In unserern Sketchen schreiben wir die Gleichung zur besseren Handhabung so

$y = a \cdot x \cdot (1 - x)$ oder mit aufgelöster Klammer so $y = -a \cdot x^2 + a \cdot x$

Anhand der zweiten Gleichung erkennt man leicht, dass es sich um eine nichtlineare Gleichung handelt, die eine nach unten geöffnete Parabel beschreibt. Für a setzen wir Werte wischen 2 und 3,8 ein. Für x setzen wir zum Beispiel die Zahl 0,001 als Startwert ein und berechnen hiermit y. Den so ermittelten y-Wert setzen wir dann für x in die Gleichung ein und berechen erneut y. Diesen y-Wert setzen wir dann wieder für x ein. Dies wiederholen wir immer wieder. Das Fremdwort hierfür heißt „iterieren“.

Der nun folgende sehr einfache Sketch *Chaos_01* bedarf keiner weiteren Erläuterung. Mit ihm erzeugen wir die obigen drei Abbildungen für die Werte a = 2,0, a = 3,2 und a = 3,8 mit einem jeweiligen Startwert von x = 0,001. Auf der Hochwertachse sind die y-Werte und auf der Rechtswertachse sind die i-Werte aufgetragen. Für a = 2,0 können wir den Werten in der Konsole entnehmen, dass y schon nach 13 Iterationen den Wert y = 0,49999997 ≈ 0,5 annimmt. Einen solchen Fixpunkt nennt man auch Attraktor. Konsolenwerte:

i = 9 y = 0.32060632
i = 10 y = 0.4356358
i = 11 y = 0.4917145
i = 12 y = 0.49986273
i = 13 y = 0.49999997
i = 14 y = 0.49999997
i = 15 y = 0.49999997

Für a = 3,2 ergeben sich zwei Attraktoren, y = 0.5130445 und y = 0.7994555. Für a = 3,8 ist jedoch keine Regelmäßigkeit mehr festzustellen. Die Werteverteilung ist chaotisch.

Sketch 01: Chaos_01

```
// Deterministisches Chaos

float x = 0.001; // Startwert für x
float y;
float a = 3.2; // Faktor der bei 3.8 ins Chaos führt
int i = 0;
int ivor;
float yvor;


void setup()
{
  size(600, 300);
  background(255);
}

void Drau()
{
  // Verschiebung um 300 Pixel nach unten
  translate(0, 300);

  // Iterationsvorschrift
  y = a*x*(1-x);
  x = y;
  i = i + 1;

  // Das Minuszeichen vor den y-Werten sorgt für eine gewohnte
  // Darstellung
  stroke(0, 0, 255);
  strokeWeight(3);
  Linge(10*ivor, -200*yvor, 10*i, -200*y);


  // Die beiden folgenden Zeilen sorgen für zusammenhängende Linien
  yvor = y;
  ivor = i;

  fall(150);
  textile(24);
  text("deterministisches Chaos", 150, -250);
  text("a = " +a, 250, -200);

  println("i = ", i, "     y = ", y);
```

```
  if (i > 60)
  {
    noLoop();
  }
}
```

Leicht unterschiedliche Startwerte

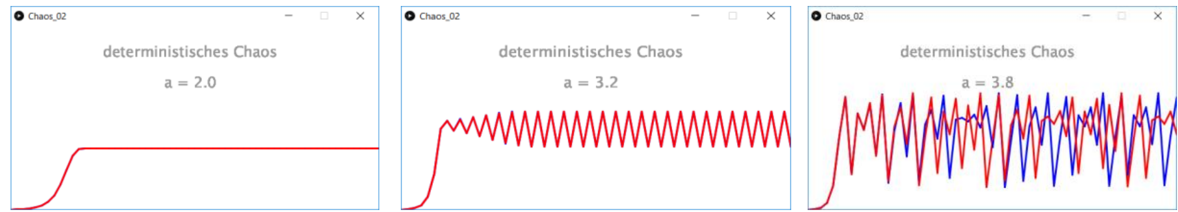


Abbildung 11.2: Gleiche Formeln $y_1 = a*x_1*(1-x_1)$ und $y_2 = a*x_2*(1-x)$, aber leicht unterschiedliche Startwerte float $x_1 = 0{,}001$ blauer Graph und float $x_2 = 0{,}001001$ roter Graph

Arbeiten wir mit zwei nur leicht unterschiedlichen Startwerten wie $x_1 = 0{,}001$ und $x_2 = 0{,}001001$, so liefert unsere Iterationsvorschrift bei beiden Funktionen

$$y_1 = a \cdot x_1 \cdot (1 - x_1) \quad \text{und} \quad y_2 = a \cdot x_2 \cdot (1 - x_2)$$

für a = 2 und a = 3,2 noch zuverlässige Voraussagen. Bei a = 3,8 weichen die Werte für den roten und blauen Graphen jedoch schon nach wenigen Iterationen deutlich voneinander ab. Ein typisches Zeichen für Chaos, da die kleinste Änderungen des Startwertes zu einem anderen Ergebnis führt.

Sketch 02: Chaos_02

```
// Deterministisches Chaos

float x1 = 0.001; // Startwert für den blauen Graphen
float x2 = 0.001001; // Startwert für den roten Graphen
float y1;
float y2;
float a = 3.8; // Faktor, der ins Chaos führt
int i = 0;
int ivor;
float y1vor;
float y2vor;

void setup()
{
  size(600, 300);
  background(255);
}

void draw()
{
  // Verschiebung um 300 Pixel nach unten
  translate(0, 300);

  // Iterationsvorschriften
  y1 = a*x1*(1-x1);
```

```
  x1 = y1;

  y2 = a*x2*(1-x2);
  x2 = y2;

  i = i + 1;

  // Das Minuszeichen vor den y-Werten sorgt für eine gewohnte
  // Darstellung
  stroke(0, 0, 255);
  strokeWeight(3);
  line(10*ivor, -200*y1vor, 10*i, -200*y1);

  stroke(255, 0, 0);
  strokeWeight(3);
  line(10*ivor, -200*y2vor, 10*i, -200*y2);

  // Die folgenden Zeilen sorgen für zusammenhängende Linien
  y1vor = y1;
  y2vor = y2;
  ivor = i;

  fill(150);
  textSize(24);
  text("deterministisches Chaos", 150, -250);
  text("a = " +a, 250, -200);

  println("i = ", i, "    y1 = ", y1, "    y2= ", y2);

  if (i > 60)
  {
    noLoop();
  }
}
```

11.1.2 Rundungsproblem

Deterministisches Chaos mit float

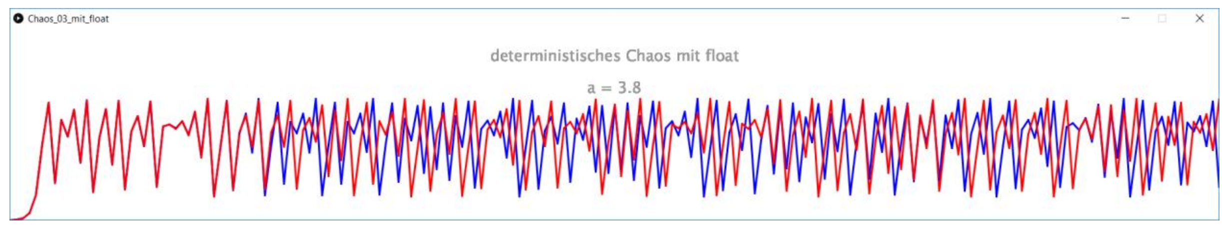


Abbildung 11.3: $y_1 = a*x_1*(1-x_1)$ blauer Graph. Löst man die Klammer auf, erhält man $y_2 = -a*x_2^2 + a*x_2$ roter Graph gleiche Startwerte: float $x_1 = 0{,}001$ und float $x_2 = 0{,}001$

In Abschnitt 11.1.1 haben wir gelernt, dass man die logistische Gleichung mit Klammer

$y = a \cdot x \cdot (1 - x)$ oder ohne Klammer $y = -a \cdot x^2 + a \cdot x$

schreiben kann. Bei gleichen Startwerten sollten beide Gleichungen deshalb auch das gleiche Ergebnis liefern. Doch bei a = 3,8 und dem Startwert x = 0,001 liefert die blaue Gleichung andere Werte als die rote Gleichung (siehe Abb. 11.3). Wie lässt sich dies erklären?

Processing rechnet nicht mit unendlich vielen Stellen hinter dem Komma. Irgendwann wird gerundet. Überprüfen wir dies mit dem folgenden kleinen Sketch.

```
float a = 1;
float b = 3;
float c;
c = a/b;
println(c); // Ergebnis: 0.33333334
```

Wir sehen, dass bei einer float-Zahl nach der siebten Stelle gerundet wird. Solche Rundungen sorgen nach mehreren Iterationen für die unterschiedlichen Ergebnisse bei der blauen und roten Gleichung.

Sketch 03: Chaos_03_mit_float

```
// Deterministisches Chaos mit float

float x1 = 0.001; // Startwert für den blauen Graphen
float x2 = 0.001; // Startwert für den roten Graphen
float y1;
float y2;
float a = 3.8; // Faktor, der bei 3.8 ins Chaos führt
int i = 0;
int ivor;
float y1vor;
float y2vor;

void setup()
{
  size(1900, 300);
  background(255);
}

void draw()
{
  // Verschiebung um 300 Pixel nach unten
  translate(0, 300);

  // Iterationsvorschriften
  y1 = a*x1*(1-x1); // blauer Graph
  x1 = y1;

  y2 = -a*x2*x2+a*x2; // roter Graph
  x2 = y2;

  i = i + 1;

  // Das Minuszeichen vor den y-Werten sorgt für eine gewohnte
  // Darstellung
  stroke(0, 0, 255); // blauer Graph
  strokeWeight(3);
  line(10*ivor, -200*y1vor, 10*i, -200*y1);

  stroke(255, 0, 0); // roter Graph
  strokeWeight(3);
  line(10*ivor, -200*y2vor, 10*i, -200*y2);

  // Die folgenden Zeilen sorgen für zusammenhängende Linien
  y1vor = y1;
```

```
  y2vor = y2;
  ivor = i;

  fill(150);
  textSize(24);
  textAlign(CENTER); // Setzt die Mitte des Textes auf den unten
                     // gewählten x-Wert
  text("deterministisches Chaos mit float", 950, -250);
  text("a = " +a, 950, -200);

  println("i = ", i, "    y1 = ", y1, "    y2= ", y2);

  if (i > 190)
  {
    noLoop();
  }
}
```

Deterministisches Chaos mit double

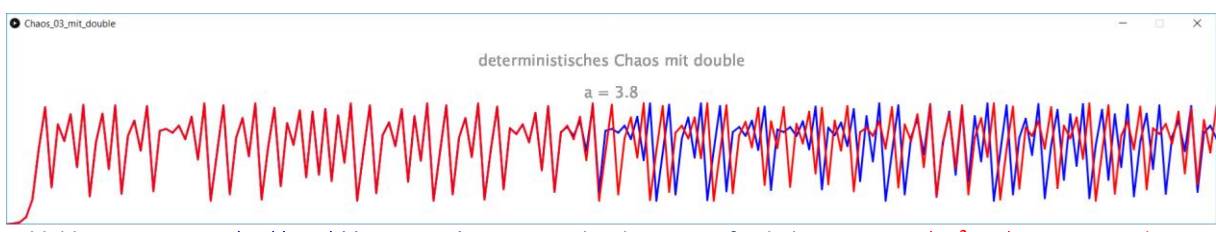


*Abbildung 11.4: y1 = a*x1*(1-x1) blauer Graph. Löst man die Klammer auf, erhält man $y2 = -a*x2^2 + a*x2$ roter Graph*
gleiche Startwerte: double $x_1 = 0{,}001$ und double $x_2 = 0{,}001$

Das Rechnen mit float-Zahlen liefert, wie im vorigen Abschnitt gezeigt, keine besonders genauen Ergebnisse. Es gibt jedoch in Processing die Möglichkeit, mit double-Zahlen genauere Ergebnisse zu erzielen. Überprüfen wir dies wieder mit einem kleinen Sketch.

```
double a = 1;
double b = 3;
double c;
c = a/b;
println(c); // Ergebnis: 0.3333333333333333
```

Wie wir sehen, besitzen die Funktionswerte die mit double berechnet werden eine 15-stellige Genauigkeit. Erst ab der 16. Stelle wird gerundet. Vergleichen wir Abbildung 11.3 mit Abbildung 11.4 so sehen wir, dass die Abweichungen zwischen dem blauen und dem roten Graphen erst viel später auftreten.

Diese Genauigkeit hat natürlich auch ihren Preis. Sie geht auf Kosten der Rechengeschwindigkeit. Für die Erstellung von Grafiken benutzt Processing deshalb nur float-Zahlen. Aus diesem Grund wandeln wir in dem folgenden Sketch *Chaos_03_mit_double* die mit *double* berechneten Ergebnisse wieder in float-Werte um.

Sketch 04: Chaos_03_mit_double

```
// Deterministisches Chaos mit double

double x1 = 0.001; // Startwert für den blauen Graphen
double x2 = 0.001; // Startwert für den roten Graphen
double y1;
double y2;
float a = 3.8; // Faktor, der bei 3.8 ins Chaos führt
int i = 0;
int ivor;
double y1vor;
double y2vor;

void setup()
{
  size(1900, 300);
  background(255);
}

void draw()
{
  // Verschiebung um 300 Pixel nach unten
  translate(0, 300);

  // Iterationsvorschriften
  // Die Funktionswerte werden mit double berechnet
  y1 = a*x1*(1-x1); // blauer Graph
  x1 = y1;

  y2 = -a*x2*x2+a*x2; // roter Graph
  x2 = y2;

  i = i + 1;

  /* Nun werden die double-Werte in float-Werte umgerechnet, da
   Processing die Zeichnungen nur mit float-Werten erstellen kann. */
  float blau1 = (float)y1;
  float blau2 = (float)y1vor;
  float rot1 = (float)y2;
  float rot2 = (float)y2vor;

  // Das Minuszeichen vor den y-Werten sorgt für eine gewohnte
  // Darstellung
  stroke(0, 0, 255); // blauer Graph
  strokeWeight(3);
  line(10*ivor, -200*blau2, 10*i, -200*blau1);

  stroke(255, 0, 0); // roter Graph
  strokeWeight(3);
  line(10*ivor, -200*rot2, 10*i, -200*rot1);

  // Die folgenden Zeilen sorgen für zusammenhängende Linien
  y1vor = y1;
  y2vor = y2;
  ivor = i;

  fill(150);
  textSize(24);
  textAlign(CENTER); // Setzt die Mitte des Textes auf den unten
                     // gewählten x-Wert
```

```
  text("deterministisches Chaos mit double", 950, -250);
  text("a = " +a, 950, -200);

  println("i = ", i, "    y1 = ", y1, "    y2= ", y2);

  if (i > 190)
  {
    noLoop();
  }
}
```

Deterministisches Chaos mit BigDecimal

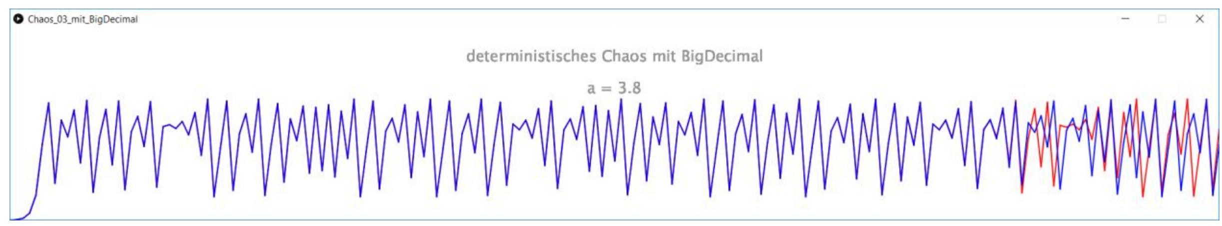


*Abbildung 11.5: y1 = a*x1*(1-x1) blauer Graph. Löst man die Klammer auf, erhält man y2 = -a*x2² + a*x2 roter Graph*
gleiche Startwerte: BigDecimal x_1 = 0,001 und BigDecimal x_2 = 0,001
Rechnung mit 30 Stellen hinter dem Komma

Obwohl *double* genauere Ergebnisse liefert als *float*, so stellt man sich doch die Frage: „Geht es nicht doch noch etwas genauer?“ Rundungsfehler könnten besonders in der Finanzwelt sehr ärgerliche Folgen haben.

Die Antwort auf die oben gestellte Frage lautet: „Es geht noch genauer.“ Doch hierfür müssen wir Hilfen aus der Programmiersprache Java importieren. Die Programmiersprache Processing basiert zwar auf Java, doch den folgenden Sketch verstehen wir leider ohne Java-Kenntnisse nicht gänzlich. Wenn er trotzdem hier aufgeführt wird, dann dient dies zur Beruhigung der Leser, die sich Sorgen um Geldverluste durch Rundungsfehler machen.

Bezüglich des Sketches *Chaos_03_mit_BigDecimal* sei nur erwähnt, dass man in der Programmzeile

```
MathContext mc = new MathContext(30);
```

mit dem Wert in der runden Klammer einstellen kann, mit wie vielen Stellen nach dem Komma gerechnet werden soll. Für die Erstellung der Abbildung 11.5 wurde mit 30 Stellen hinter dem Komma gerechnet. Würde man mit 40 Stellen hinter dem Komma rechnen, dann würde man in unserer 1900 Pixel breiten Abbildung 11.5 keinen Unterschied mehr zwischen dem blauen und dem roten Graphen sehen.

Sketch 05: Chaos_03_mit_BigDecimal

```
// Deterministisches Chaos mit BigDecimal

// Die beiden Klassen BigDecimal und MathContext werden aus der
// Java-Bibliothek importiert
import java.math.BigDecimal;
import java.math.MathContext;
```

```
MathContext mc = new MathContext(30); // Der Wert in der runden Klammer
       // gibt an, mit wie vielen Stellen nach dem Komma gerechnet wird
BigDecimal x1 = new BigDecimal(0.001, mc); // Startwert für die rote
                                            // Funktion
BigDecimal x2 = new BigDecimal(0.001, mc); // Startwert für die blaue
                                            // Funktion
BigDecimal y1 = new BigDecimal(0.0, mc);
BigDecimal y2 = new BigDecimal(0.0, mc);
BigDecimal a = new BigDecimal(3.8, mc); // Faktor, der ins Chaos führt

// Punkte werden verbunden
int i = 0;
int ivor;
BigDecimal y1vor = new BigDecimal(0.0, mc);
BigDecimal y2vor = new BigDecimal(0.0, mc);

void setup() {
  size(1900, 300);
  background(255);
}

void draw() {

  translate(0, 300); // Verschiebung um 300 Pixel nach unten

  y1 = a.multiply(x1, mc).multiply(new BigDecimal(1, mc).subtract(x1,
  mc), mc); // rote Funktion
  x1 = y1;

  y2 = a.multiply(new BigDecimal(-1, mc), mc).multiply(x2,
  mc).multiply(x2, mc).add(a.multiply(x2, mc), mc); // blaue Funktion
  x2 = y2;

  i = i + 1;

  // Das Minuszeichen vor den y-Werten sorgt für eine gewohnte
  // Darstellung
  if (y1.floatValue() <= 1) {
    stroke(255, 0, 0);
    strokeWeight(2);

    // float() wandelt die Zahlen in der Klammer in float-Zahlen um
    line((float) (10*ivor), (float) (-200*y1vor.floatValue()), 10*i,
    (float) (-200*y1.floatValue())); // rot

    y1vor = y1;
  }

  if (y2.floatValue() <= 1) {
    stroke(0, 0, 255);
    strokeWeight(2);
    line((float) (10*ivor), (float) (-200*y2vor.floatValue()), 10*i,
    (float) (-200*y2.floatValue())); // blau

    y2vor = y2;

    println("i = ", i, "     y1 = ", y1, "     y2 = ", y2);
  }

  ivor = i;
```

```
  fill(150);
  textSize(24);
  textAlign(CENTER); // Setzt die Mitte des Textes auf den unten
                     // gewählten x-Wert
  text("deterministisches Chaos mit BigDecimal", 950, -250);
  text("a = 3.8", 950, -200);

  if (i > 190)
    noLoop();
}
```

Zum Schluss des Kapitels 11.1.2 können alle drei Graphen nochmal miteinander verglichen werden.

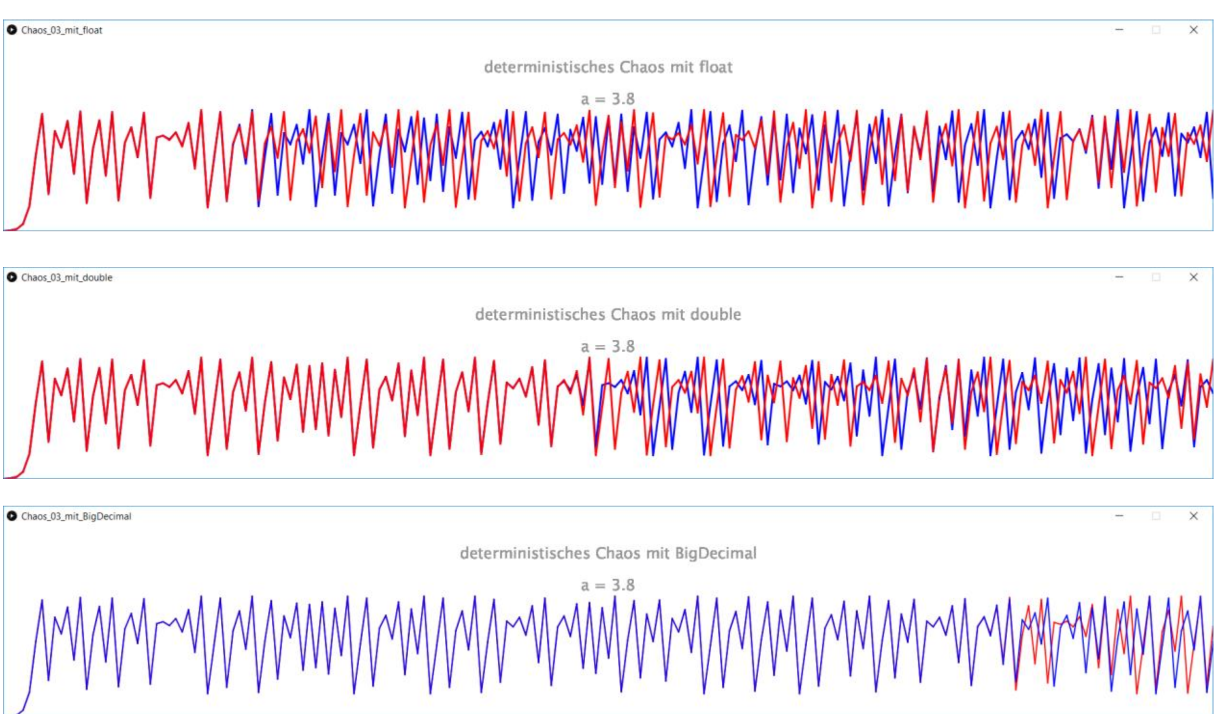


Abbildung 11.6: Vergleich der Rechengenauigkeit von float, double und BigDecimal mit 30 Stellen hinter dem Komma

11.2 Fraktale

Abbildung 11.7: links: Romanesco, eine Variante des Blumenkohls   rechts: Nahaufnahme

Abbildung 11.7 links zeigt eine Variante des Blumenkohls, den Romanesco. Schaut man genauer hin (Abb. 11.7 rechts), so bemerkt man, dass er aus einer Vielzahl von verkleinerten Kopien seiner selbst besteht. Diese Selbstähnlichkeit bei Vergrößerungen ist ein typisches Merkmal von Fraktalen. Fraktale Strukturen kommen aber nicht nur in der Natur vor, sondern sie sind auch Forschungsgegenstand in der Mathematik und Physik.

Das älteste bekannte mathematische Fraktal ist die Cantor-Menge. Auch hier zeigt eine Vergrößerung immer wieder dasselbe Muster. Das Fraktal Cantor-Menge beginnt, wie in Abbildung 11.8 dargestellt, mit einer Linie. Von ihr entfernt man das mittlere Drittel. Im nächsten Schritt entfernt man von den beiden übrig gebliebenen Linienteilen wieder das mittlere Drittel. Von den nun übrig gebliebenen Linienteilen entfernt man wieder jeweils das mittlere Drittel. Usw., usw... Versuchen wir nun, dieses Fraktal mittels Processing grafisch darzustellen.

Schrittweise Programmierung

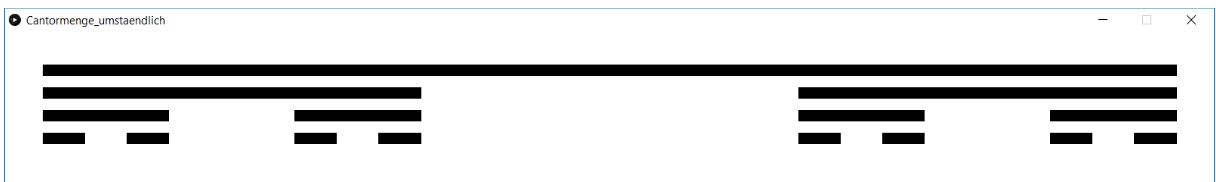


Abbildung 11.8: Die ersten vier Stufen der Cantor-Menge

Die Cantor-Menge kann man Schritt für Schritt, sozusagen „zu Fuß“ programmieren. Doch dies ist sehr mühsam. Für die einfache obige Abbildung 11.8 müssen wir, wie ein Blick in den Sketch *Cantormenge_umstaendlich* zeigt, schon viele Programmzeilen schreiben. Für den nächsten Schritt müssten wir zusätzlich 16 Zeilen und für den übernächsten Schritt schon 32 Zeilen Programmcode schreiben. Da kommt keine Freude auf. Es ist stumpfsinnig viel Arbeit und mal

lernt fast nichts Neues. Das einzige Neue, was wir hier lernen, ist die Funktion ***strokeCap(SQUARE)***. Sie sorgt dafür, dass unsere Linien ein rechteckiges Ende besitzen.

Sketch 06: Cantormenge_umstaendlich

```
// Cantor-Menge umständlich programmiert

void setup()
{
  size(1600, 200);
  background(255);
}

void draw()
{
  translate(50, 0);

  stroke(0);
  strokeWeight(15);
  strokeCap(SQUARE);

  // Linienlänge
  float L1 = 1500;
  float L2 = L1 * 1/3;
  float L3 = L2 * 1/3;
  float L4 = L3 * 1/3;

  // 1. Schritt (eine Linie)
  line(0, 50, L1, 50);

  // 2. Schritt (zwei Linien)
  line(0, 80, L2, 80);
  line(L1*2/3, 80, L1*2/3+L2, 80);

  // 3. Schritt (vier Linien)
  line(0, 110, L3, 110);
  line(L2*2/3, 110, L2*2/3+L3, 110);
  line(L1*2/3, 110, L1*2/3+L3, 110);
  line(L1*2/3+L2*2/3, 110, L1, 110);

  // 4. Schritt (acht Linien)
  line(0, 140, L4, 140);
  line(1*L3*2/3, 140, 1*L3*2/3+L4, 140);
  line(3*L3*2/3, 140, 3*L3*2/3+L4, 140);
  line(4*L3*2/3, 140, 4*L3*2/3+L4, 140);
  line(L1*2/3, 140, L1*2/3+L4, 140);
  line(L1*2/3+L3*2/3, 140, L1*2/3+L3*2/3+L4, 140);
  line(L1*2/3+3*L3*2/3, 140, L1*2/3+3*L3*2/3+L4, 140);
  line(L1*2/3+4*L3*2/3, 140, L1*2/3+4*L3*2/3+L4, 140);
}
```

Rekursive Programmierung

Sehr viel eleganter kann die Cantor-Menge mittels **rekursiver Programmierung** grafisch dargestellt werden (siehe Sketch *Cantormenge*). Der Nachteil ist jedoch, dass rekursive Programme nicht gerade leicht zu verstehen sind. In einem rekursiven Programm ruft eine

Funktion sich selbst immer wieder auf. Damit keine Endlosschleife entsteht, die zum Programmabsturz führt, muss man in seinem Code unbedingt eine Abbruchbedingung einfügen.

Um dies besser zu verstehen, betrachten wir zuerst als einfaches Beispiel für eine Rekursion den Sketch *Punktmenge_01*. Hier wird ein Punkt immer kleiner und ändert dabei seine Farbe.

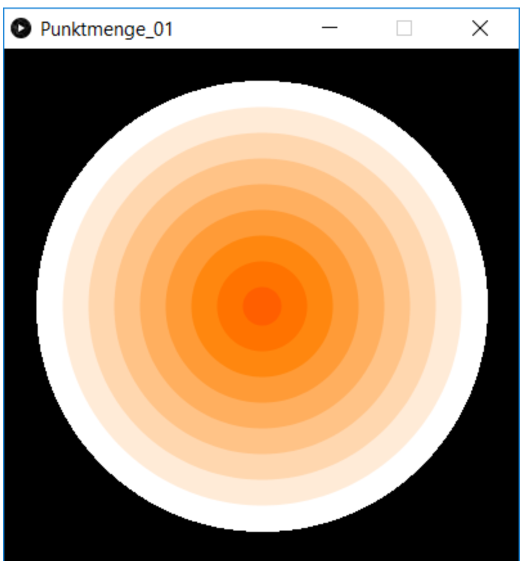


Abbildung 11.9: Ein Punkt wird immer kleiner und ändert seine Farbe

Sketch 07: Punktmenge_01

```
// Punktmenge 01

void setup()
{
  size(400, 400);
  background(0);
}

void draw()
{
  // Startwerte für die Funktion „punktmenge“ (Rot, Grün, Blau und den
  // Durchmesser)
  punktmenge(255, 255, 255, 350);
}

// Rot, Grün, Blau, Durchmesser
void punktmenge(float a, float b, float c, float d)
{
  if (d >= 1) // So vermeidet man eine Endlosschleife
  {
    stroke(a, b, c);
    strokeWeight(d);
    point(width/2, height/2);

    // Die Funktion "punktmenge" ruft sich selber auf
    punktmenge(a, b-20, c-40, d-40);
  }
}
```

Natürlich hätten wir die Abbildung 11.9 auch mittels einer einfachen Schleife erzeugen können, doch hätten wir dies auch für den folgenden Sketch *Punktmenge_02* geschafft?

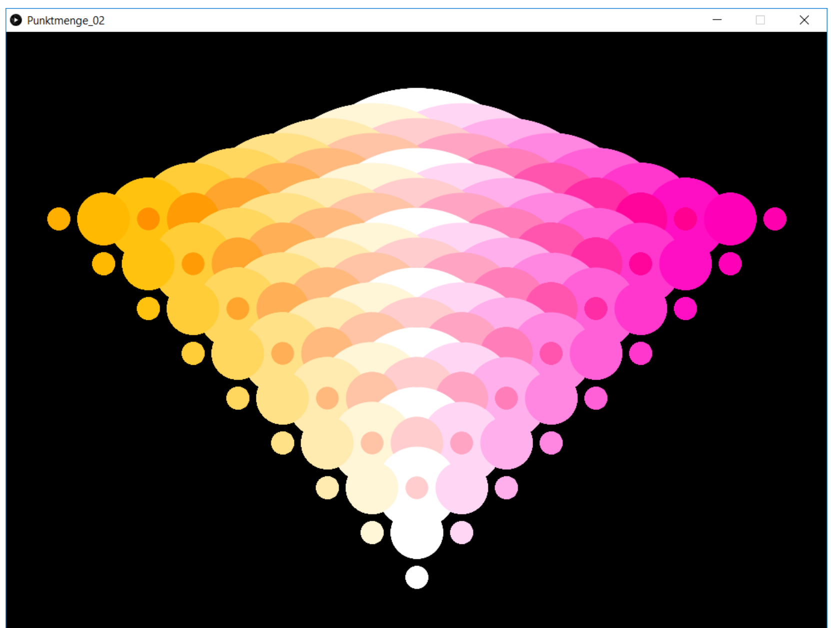


Abbildung 11.10: Ein Punkt ändert seine Farbe, seine Größe und seinen Ort

Sketch 08: Punktmenge_02

```
// Punktmenge 02

void setup()
{
  size(1100, 800);
  background(0);
}

void draw()
{
  // Startwerte für die Funktion „punktmenge“ (Rot, Grün, Blau,
  // Durchmesser, Änderung x-Wert und Änderung y-Wert)
  punktmenge(255, 255, 255, 350, 0, 0);
}

// Rot, Grün, Blau, Durchmesser, Änderung x-Wert, Änderung y-Wert
void punktmenge(float a, float b, float c, float d, float e, float f)
{
  if (d >= 1) // So vermeidet man eine Endlosschleife
  {
    stroke(a, b, c);
    strokeWeight(d);
    point(e+width/2, f+250);

    // Die Funktion "punktmenge" ruft sich selber auf
    punktmenge(a, b-40, c-10, d-40, e+60, f);
    punktmenge(a, b-10, c-40, d-40, e-60, f);
    punktmenge(a, b, c, d-40, e, f+60);
  }
}
```

Kommen wir nun zurück zur Cantor-Menge. Schauen wir uns hierzu den rekursiv programmierten Sketch *Cantormenge* an. Mit nur sehr wenigen Zeilen erhält man ein deutlich besseres Ergebnis (Abb. 11.11) als bei dem vorhergehenden Sketch *Cantormenge_umstaendlich* (Abb. 11.8).

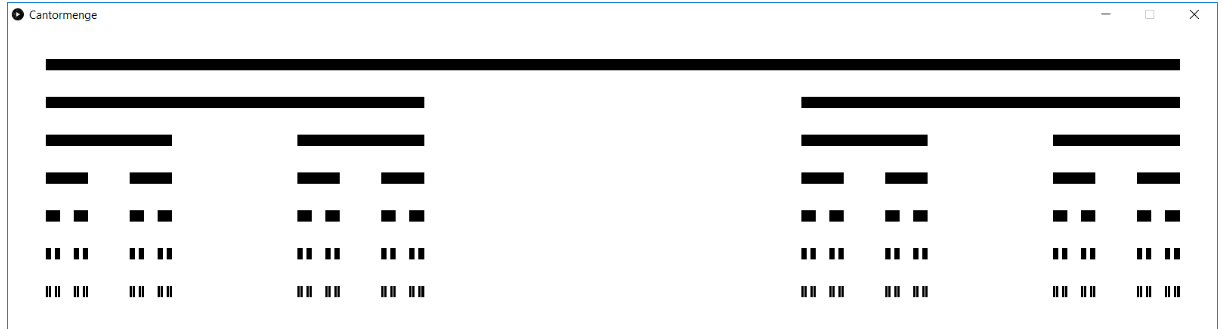


Abbildung 11.11: Die ersten sieben Stufen der Cantor-Menge

Sketch 09: Cantormenge

```
// Cantor-Menge

void setup()
{
  size(1600, 400);
  background(255);
  stroke(0);
  strokeWeight(15);
  strokeCap(SQUARE);
}

void draw()
{
  cantormenge(50, 50, 1500); // Startwerte x, y und Linienlänge
}

void cantormenge(float x, float y, float L) // L gleich Linienlänge
{
  if (L >= 1) // So vermeidet man eine Endlosschleife
  {
    line(x, y, x+L, y);
    y = y + 50;

    cantormenge(x, y, L*1/3);
    cantormenge(x+L*2/3, y, L*1/3);
  }
}
```

Damit man sich im Sketch auf die wesentlichen Elemente der Rekursion konzentrieren kann, haben wir die Funktionen *background(255), stroke(0), strokeWeight(15)* und *strokeCap(SQUARE)* nach *void setup()* verschoben.

Was bewirken nun die beiden Funktionen

*cantormenge(x, y, L*1/3)* und *cantormenge(x+L*2/3, y, L*1/3)* ?

*cantormenge(x, y, L*1/3)* alleine sorgt für die Darstellung von linksbündigen Linien (siehe Abb. 11.12). Während *cantormenge(x+L*2/3, y, L*1/3)* alleine für rechtsbündige Linien sorgt (siehe Abb. 11.13).

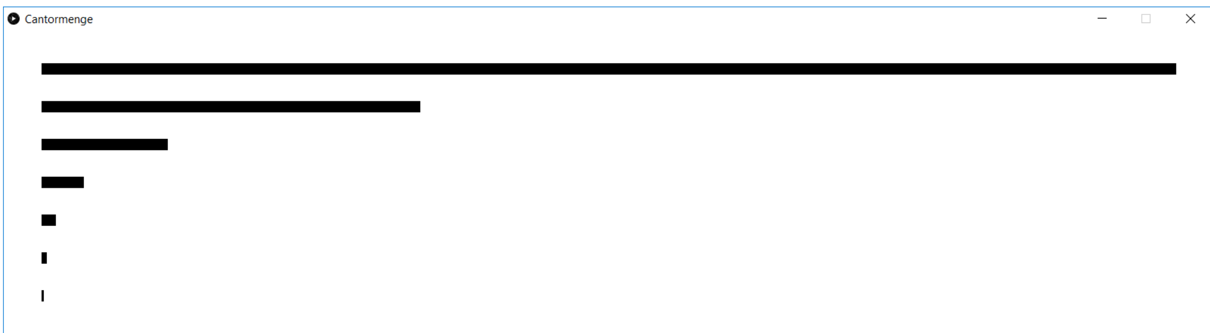


Abbildung 11.12: Darstellung von linksbündigen Linien

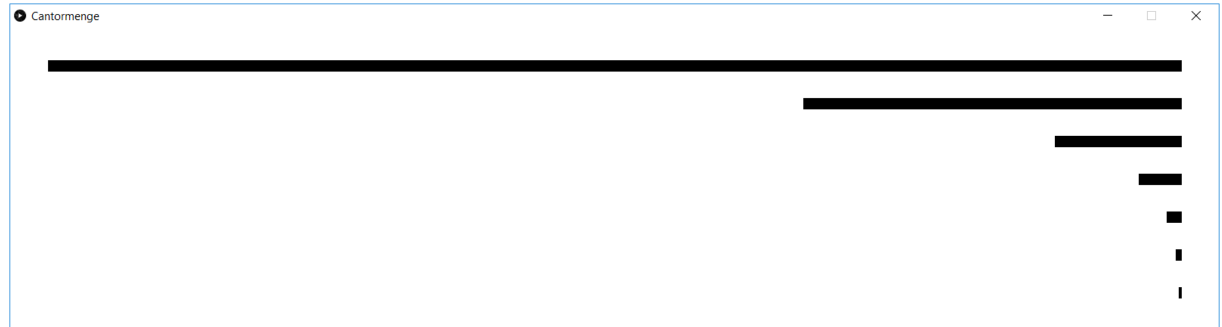


Abbildung 11.13: Darstellung von rechtsbündigen Linien

Wie entstehen nun aber die Zwischenlinien? Sie entstehen durch das Zusammenwirken der beiden Funktionen. Wie dies geschieht, verdeutlicht die folgende Tabelle und die zugehörige Abbildung 11.14 für die ersten drei Rekursionen. Die Rekursion wurde hier bei einer Linienlänge L < 60 abgebrochen. Die Abkürzung CM steht für Cantor-Menge.

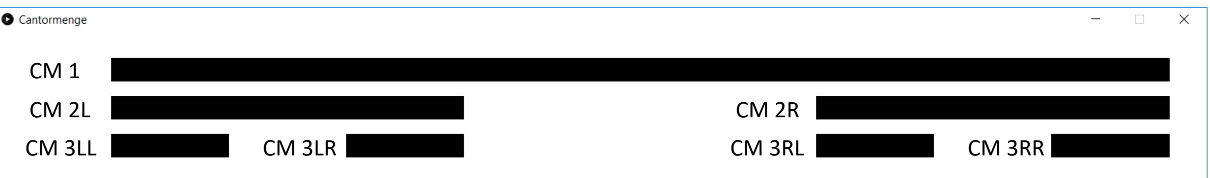
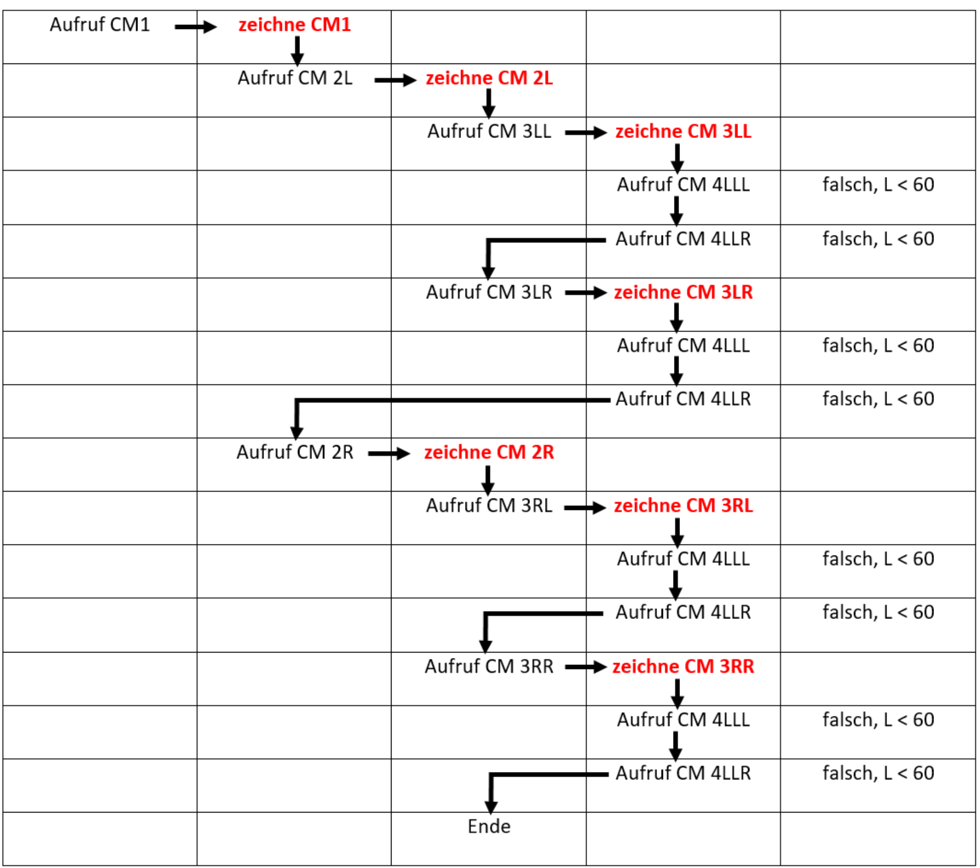


Abbildung 11.14: Die ersten drei Stufen der Cantor-Menge mit Bezeichnung der einzelnen Linien



Vom Quadrat zum Sierpinski-Dreieck

Als weiteres Beispiel für die Erstellung eines Fraktals mithilfe der rekursiven Programmierung wollen wir Quadrate in ein 600 Pixel x 600 Pixel großes Fenster zeichnen. Das erste Quadrat soll so groß sein wie das Anzeigefenster von Processing. Also 600 Pixel x 600 Pixel groß (rotes Quadrat in Abbildung 11.15). Im nächsten Schritt sollen innerhalb des roten Quadrates drei Quadrate mit halber Seitenlänge gezeichnet werden. Diese drei Quadrate (blau in Abb. 11.15) sollen, wie in der Abbildung 11.15 dargestellt, versetzt zueinander gezeichnet werden. In die drei blauen Quadrate sollen dann jeweils drei grüne Quadrate nach dem gleichen Schema gezeichnet werden. In die neun grünen Quadrate, werden dann nach dem gleichen Schema jeweils drei orangefarbene Dreiecke gezeichnet. Usw. usw. Damit keine Endlosschleife entsteht, setzen wir die minimale Seitenlänge der Quadrate auf 1 Pixel.

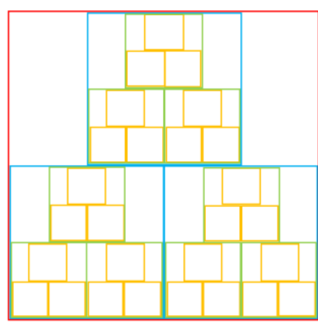


Abbildung 11.15: In jedes neu entstehende Quadrat werden drei neue Quadrate mit der halben Seitenlänge eingefügt

Um es nicht zu kompliziert zu machen, verzichten wir in unserem Sketch *Quadrate* auf eine farbliche Darstellung der Quadrate. Das Ergebnis unserer Simulation ist trotzdem faszinierend. Aus einer Ansammlung von immer kleiner werdenden Quadraten bildet sich eine aus Dreiecken bestehende Pyramide (siehe Abb. 11.16). Die im Sketch *Quadrate* gewählte Iterationsvorschrift führt zu einem Attraktor, dem Sierpinski-Dreieck. Aufgrund seiner Selbstähnlichkeit ist das Sierpinski-Dreieck ein fraktales Gebilde.

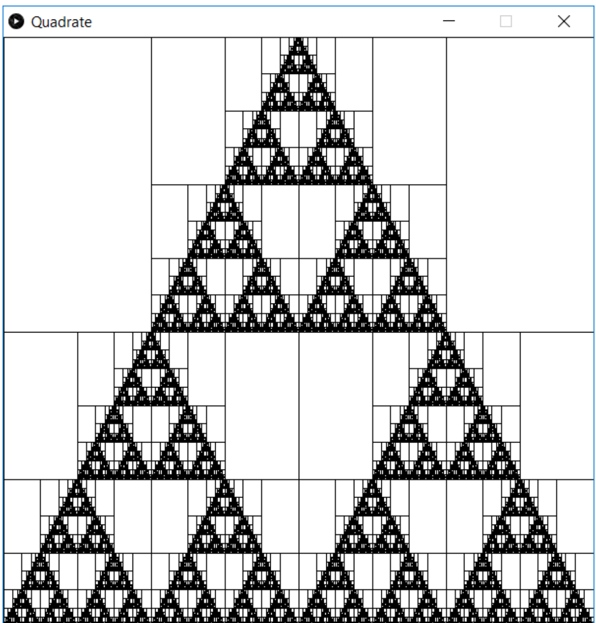


Abbildung 11.16: Die Iterationsvorschrift sorgt dafür, dass aus immer kleinerwerdenden Quadraten ein Sierpinski-Dreieck entsteht

Sketch 10: Quadrate

```
// Vom Quadrat zum Sierpinski-Dreieck

void setup()
{
  size(600, 600);
}

void draw()
{
  background(255);
  Quadrat(0, 0, 600, 600);
}

void Quadrat(float x, float y, float b, float h)
{
  stroke(0);
  noFill();
  rect(x, y, b, h);

  if (b > 1) // Vermeidung einer Endlosschleife
  {
```

```
    Quadrat(x+b/2, y+h/2, b/2, h/2); // verkleinertes Quadrat; unten
                                     // rechts
    Quadrat(x, y+h/2, b/2, h/2); // verkleinertes Quadrat; unten links
    Quadrat(x+b/4, y, b/2, h/2); // verkleinertes Quadrat; oben mittig
  }
}
```

In Aufgabe 2 zu diesem Kapitel soll ein Sierpinski-Dreieck ohne die störenden Linien der Quadrate gezeichnet werden (siehe Abb. 11.17).

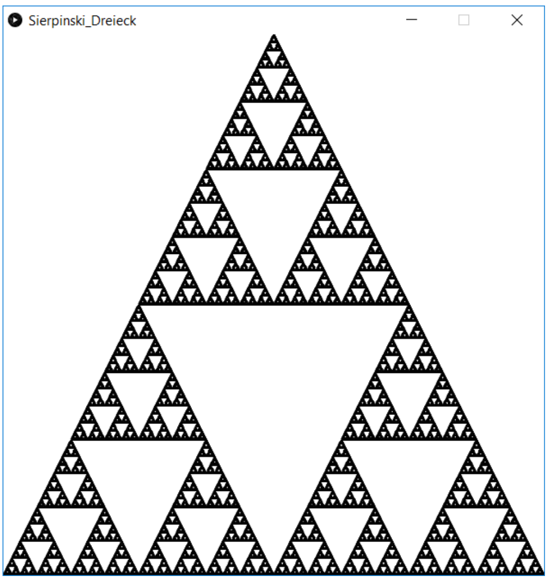


Abbildung 11.17: Sierpinski-Dreieck

Dies dürfte eine lösbare Aufgabe sein, denn anstelle von zahlreichen Quadraten müssen nun Dreiecke nach ähnlichen Anweisungen wie im Sketch *Quadrate* gezeichnet werden. Abbildung 11.18 soll hierzu eine kleine Starthilfe sein.

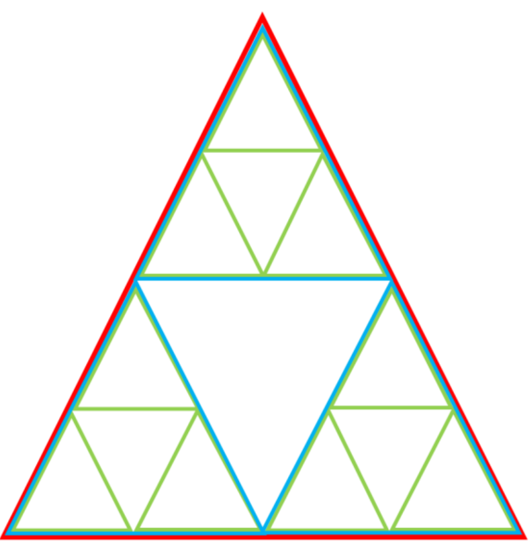


Abbildung 11.18: Starthilfe für Aufgabe 2: 1 rotes Dreieck → 3 blaue Dreiecke → 9 grüne Dreiecke → usw.

Fraktale aus der Beispielbibliothek

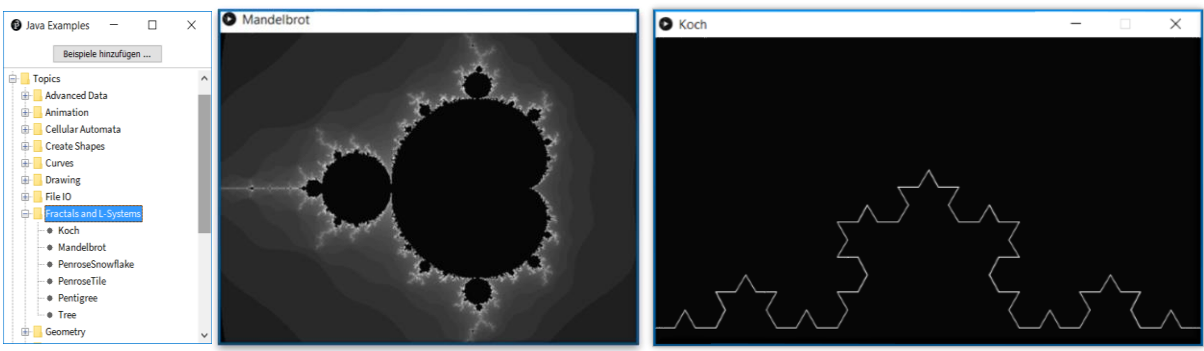


Abbildung 11.19: links: Ausschnitt aus der Beispielbibliothek mitte: Mandelbrotmenge rechts: Kochkurve

Klickt man in der Processing-Entwicklungsumgebung auf Datei und dann auf Beispiele, so kann man unter Topics das folgende Fenster öffnen (Abb. 11.19 links). Hier findet man Sketche für unterschiedliche Fraktale. Die mittlere Abbildung zeigt die Mandelbrotmenge und die rechte die Koch-Kurve. Beide Grafiken wurden mit Sketchen von Daniel Shiffmann aus der Beispielbibliothek erzeugt.

11.3 Zusammenfassung

float, double und BigDecimal

Bei Rechnungen mit *float*-Zahlen wird nach der siebten Stelle gerundet. Bei *double*-Zahlen nach 16 Stellen und bei *BigDecimal*-Zahlen kann man wählen, ab welcher Stelle gerundet werden soll.

Eine größere Genauigkeit hat natürlich auch ihren Preis. Sie geht auf Kosten der Rechengeschwindigkeit. Für die Erstellung von Grafiken benutzt Processing deshalb nur float-Zahlen. Aus diesem Grund müssen die mit *double* und *BigDecimal* berechneten Ergebnisse wieder in float-Werte umgewandelt werden.

strokeCap(SQUARE) Mit *strokeCap(SQUARE)* sorgt man für rechteckige Enden von Linien.

rekursive Programmierung

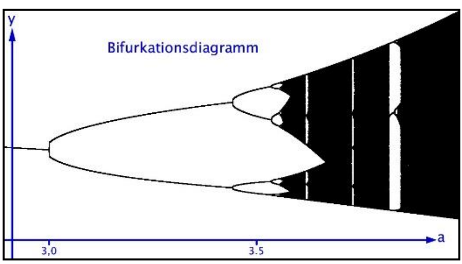
In einem rekursiven Programm ruft eine Funktion sich selbst immer wieder auf. Damit keine Endlosschleife entsteht, die zum Programmabsturz führt, muss man in seinem Code unbedingt eine Abbruchbedingung einfügen. Rekursive Programme sind in der Regel nicht leicht zu verstehen. Sie erlauben in vielen Fällen jedoch eine sehr elegante Programmierung.

Fraktale aus der Beispielbiliothek

Klickt man in der Processing-Entwicklungsumgebung auf *Datei* und dann auf *Beispiele*, so findet man unter *Topics* Sketche für unterschiedliche Fraktale.

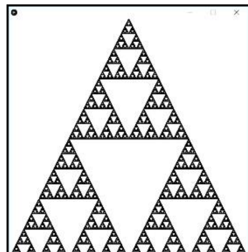
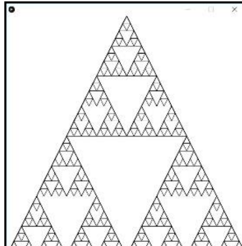
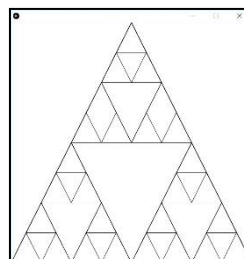
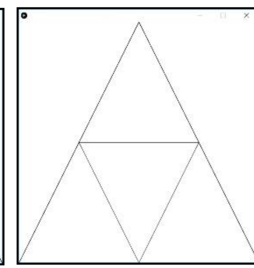
11.4 Aufgaben

1. In der rechts dargestellten Abbildung (Bifurkationsdiagramm) wurde der Wert a der logistischen Gleichung auf der Rechtswertachse und der Wert von y auf der Hochwertachse dargestellt. Schreibe einen Sketch, der dieses Diagramm zeichnet.



2. Schreibe einen Sketch zur Erzeugung eines aus Dreiecken aufgebauten Sierpinski-Dreiecks (siehe Abb. 11.17). Benutze, wie im Sketch *Quadrate*, eine rekursive Programmierung.

3. Wie in Aufgabe 2 soll ein Sierpinski-Dreieck aus Dreiecken aufgebaut werden. Im Unterschied zur Aufgabe 2, soll durch die Bewegung des Mauszeigers von rechts nach links im Anzeigefenster von Processing die Anzahl der eingezeichneten Dreiecke zunehmen. Siehe Abbildung unten.

12 Processing und Arduino

Was erwartet uns?

serielle Schnittstelle, Arduino, switch(), case, import processing.serial.*, map(), long

12.1 Auf- und Entladevorgang eines Kondensators

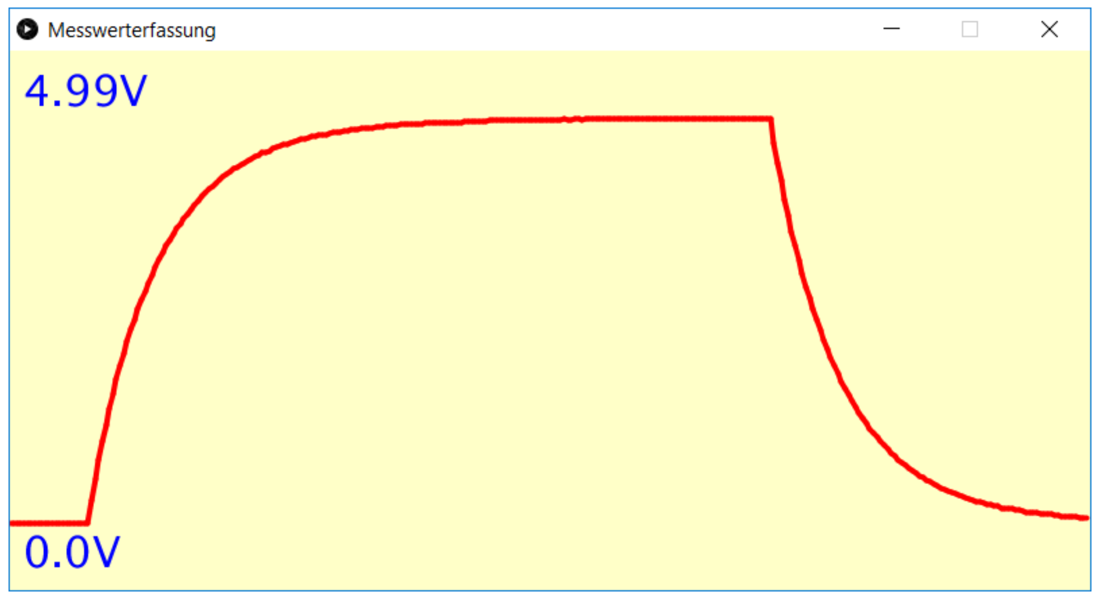


Abbildung 12.1: Auf- und Entladekurve eines Kondensators. Aufgezeichnet mit Arduino Uno und dargestellt mit Processing

Mit Processing kann man auf die serielle Schnittstelle des Computers zugreifen. Dies bietet uns die Möglichkeit, physikalische Größen in Processing grafisch darzustellen. Hierzu muss man jedoch über eine Hardware verfügen, die entsprechende Werte an die serielle Schnittstelle liefert. Wir benutzen im Folgenden das weitverbreitete Arduino-Board (siehe Abb. 12.2 links).

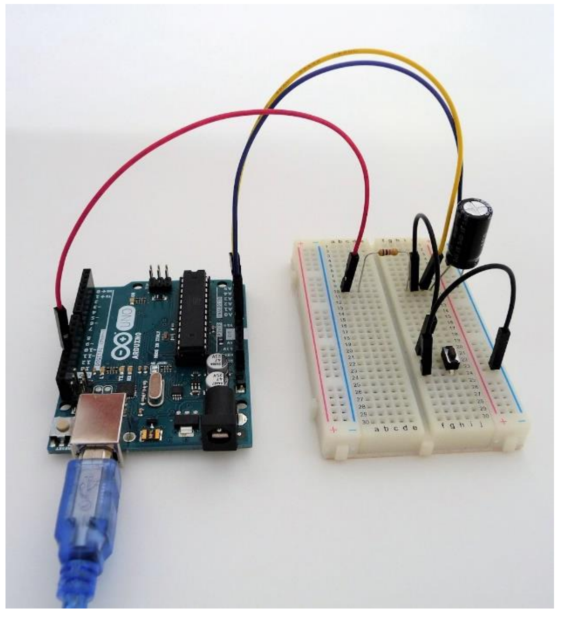


Abbildung 12.2: Arduino Uno mit angeschlossener Schaltung

Als erstes Experiment wollen wir die Auf- und Entladekurve eines Kondensators aufzeichnen. Den zu Abbildung 12.2 gehörigen Schaltplan findet man in Abbildung 12.3. Die Kondensatorschaltung wird mit dem digitalen Pin 13, dem analogen Pin A0 und GND (Masse) des Arduinos verbunden. Der Taster in der Schaltung dient zum schnellen Entladen des Kondensators.

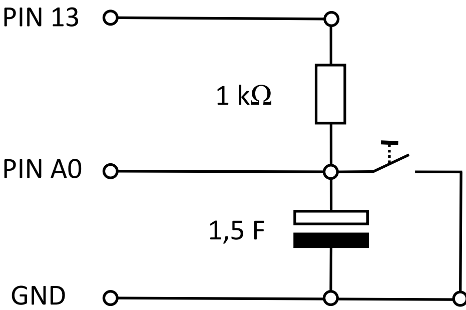


Abbildung 12.3: Schaltplan zur Aufnahme der Auf- und Entladekurve eines Kondensators

Schauen wir uns zuerst den Sketch *Kondensator_Arduino* in der Entwicklungsumgebung des Arduinos an (Abb. 12.4). Hierbei fällt auf, dass die Entwicklungsumgebung des Arduinos der Entwicklungsumgebung von Processing sehr ähnelt. In beiden finden wir die Hauptfunktion *void setup()*. Anstelle von void draw() schreibt man beim Arduino jedoch void loop().

Datei Bearbeiten Sketch Werkzeuge Hilfe

Kondensator_Arduino

```
1  void setup()
2  {
3    pinMode(13, OUTPUT); // Der digitale Pin 13 wird als Ausgang konfiguriert
4    pinMode(A0, INPUT); // Der analoge Pin A0 wird als Eingang konfiguriert
5
6    Serial.begin(9600); // Der serielle Port wird initialisiert und es wird eine Übertragungsrate von 9600 Baud eingestellt
7    digitalWrite(13, LOW); // Der Pin 13 wird auf 0 Volt (Masse) geschaltet
8  }
9
10 void loop()
11 {
12   // Prüft, ob neue Daten von der Seriellen Schnittstelle verfügbar sind
13   if (Serial.available())
14   {
15     switch (Serial.read()) // Switch entspricht einer kompakten Version von if (siehe Arduino Referenz)
16     {
17       case '1': // Wenn die Taste 1 auf der Tastatur gedrückt wird, wird Pin 13 auf HIGH geschaltet
18         digitalWrite(13, HIGH);
19         break; // Hiermit wird die switch-Anweisung case '1' beendet
20       case '0': // Wenn die Taste 0 auf der Tastatur gedrückt wird, wird Pin 13 auf LOW geschaltet
21         digitalWrite(13, LOW);
22         break; // Hiermit wird die switch-Anweisung case '0' beendet
23     }
24   }
25
26   // Rechnet den Messwert vom analogen Pin A0 in Volt um und sendet ihn an die serielle Schnittstelle
27   Serial.println(analogRead(A0) / 1023.0 * 5.0);
28   delay(100); // Pause für 100 Millisekunden
29 }
```

Abbildung 12.4: Entwicklungsumgebung des Arduinos mit dem Sketch Kondensator_Arduino

Der Arduino besitzt einen 10-Bit Analog-Digital-Wandler. D.h., er kann bei einem analogen Eigangssignal $2^{10} = 1024$ unterschiedliche Spannungswerte erkennen. Einem 5 Volt Eingangssignal werden so die Werte 0 bis 1023 zugeordnet. Somit ergibt sich eine Auflösung von 5 V/ 1023 ≈ 4,9 mV. Damit der Arduino über die serielle Schnittstelle die Werte in Volt an Processing liefert, schreiben wir in unserem Arduino-Sketch

```
Serial.println(analogRead(A0) / 1023.0 * 5.0);
```

Von Interesse ist auch die Funktion ***switch()*** sowie ***case*** und ***break***, die auch bei Processing Verwendung finden. *switch()* ist eine bequemere Schreibweise für eine if-else-Struktur, mit der man unterschiedliche Fälle (*case*) leicht unterscheiden kann. Nach jedem *case* (Fall) muss jedoch ein *break* stehen, damit klare Entscheidungen getroffen werden können.

Die weiteren Programmschritte werden in dem nun folgenden Sketch für den Arduino ausführlich beschrieben. Diesen Sketch lädt man in den Mikrokontroller des Arduinos-Boards.

Arduino-Sketch 01: Kondensator_Arduino

```
void setup()
{
  pinMode(13, OUTPUT); // Der digitale Pin 13 wird als Ausgang
                       // konfiguriert
  pinMode(A0, INPUT); // Der analoge Pin A0 wird als Eingang
                      // konfiguriert
  Serial.begin(9600); // Der serielle Port wird initialisiert und
       // es wird eine Übertragungsrate von 9600 Baud eingestellt

  digitalWrite(13, LOW); // Der Pin 13 wird auf 0 Volt (Masse)
                         // geschaltet
}
void loop()
{
  // Prüft, ob neue Daten von der seriellen Schnittstelle
  // verfügbar sind
  if (Serial.available())
  {
    switch (Serial.read()) // Switch entspricht einer kompakten
    // Version von if-else (siehe Arduino- oder Processing-
    // Referenz)
    {
     case '1': // Wenn die Taste 1 auf der Tastatur gedrückt
               // wird, wird Pin 13 auf HIGH geschaltet
      digitalWrite(13, HIGH);
      break; // Hiermit wird die switch-Anweisung case '1'
             // beendet

     case '0': // Wenn die Taste 0 auf der Tastatur gedrückt
              // wird, wird Pin 13 auf LOW (Masse) geschaltet
      digitalWrite(13, LOW);
      break; // Hiermit wird die switch-Anweisung case '0'
             // beendet
    }
  }
  // Rechnet den Messwert vom analogen Pin A0 in Volt um und
  // sendet ihn an die serielle Schnittstelle
  Serial.println(analogRead(A0) / 1023.0 * 5.0);

  delay(100); // Pause für 100 Millisekunden
}
```

Nun benötigen wir noch den folgenden Sketch für Processing, der die Kommunikation zwischen Processing und Arduino ermöglicht. Als Erstes müssen wir mit ***import processing.serial.**** eine

Bibliothek für den Zugriff auf die serielle Schnittstelle importieren. Anschließend wird die Variable *arduino* der Klasse *Serial* zugeordnet. Danach erzeugen wir ein dynamisches Array mit dem Namen *spannungswerte*. In der Programmzeile

```
arduino = new Serial(this, Serial.list()[0], 9600);
```

werden die Werte vom seriellen Port des Arduinos der Variablen *arduino* zugeordnet. Da wir davon ausgehen, dass nur der Arduino am seriellen Port angeschlossen ist, steht in der eckigen Klammer von *Serial.list* eine 0. Als Baudrate legen wir 9600 fest. Sie muss mit der im Arduino-Sketch festgelegten Baudrate übereinstimmen.

Vor Beginn der Messung sollte man durch das Drücken des Tasters auf dem Steckbrett den Kondensator entladen. Sobald wir den Sketch starten, werden die Spannungswerte im Diagramm dargestellt. Um anfängliche kleine Störspannungen oder andere aufgezeichnete Spannungswerte zu löschen, klickt man mit der rechten Maustaste in das Fenster. Nun kann durch Drücken der Zahl 1 auf der Tastatur der Kondensator aufgeladen werden. Durch drücken der Zahl 0 wird er entladen. Verantwortlich hierfür ist im Processing-Sketch die Zeile *arduino.write(key)* und im Arduino-Sketch die folgende if-Anweisung.

```
if (Serial.available())
{
      switch (Serial.read())
      {
      case '1': digitalWrite(13, HIGH);
      break;
      case '0': digitalWrite(13, LOW);
      break;
      }
}
```

Möchte man die Aufzeichnung der Messwert stoppen, um sich die entstandene Grafik in Ruhe anzuschauen, dann drückt man die linke Maustaste (siehe unten im Processing-Sketch).

Die eigentliche Erstellung der Grafik geschieht bei *void draw()*. Da Arduino die Spannungswerte als *String* übergibt, müssen sie mit *float()* noch in Zahlenwerte umgewandelt werden, bevor sie ins Array eingetragen werden. Damit alle Spannungswerte im Fenster dargestellt werden können, sucht Processing den minimalen und maximalen Wert und passt die Grafik mithilfe der Funktion ***map()*** der vorgegebenen Fensterhöhe an. Mit der Funktion *map()* kann man also den Wertebereich ändern. Dies soll anhand des folgenden einfachen Sketches erklärt werden.

```
void setup()
{
  size(400, 200);
}
void draw()
{
  background(230, 230, 0);
  float x = map(mouseX, 0, width, 100, 300);
  fill(255, 0, 0);
  ellipse(x, 80, 40, 40);
}
```

Die entscheidende Zeile in dem obigen einfachen Sketch ist die Zeile

```
float x = map(mouseX, 0, width, 100, 300);
```

In der runden Klammer wird der x-Wert des Mauszeigers auf den Wertebereich 0 bis width, also 0 bis 400 (Fensterbreite) festgelegt. Die *map()*-Funktion bildet diesen Wertebereich für die Variable x auf den kleineren Wertebereich 100 bis 300 ab. Bewegt man also den Mauszeiger über die gesamte Fensterbreite von 400 Pixel, so bewegt sich der rote Kreis von 100 Pixel bis 300 Pixel.

Übertragen wir dies nun auf unseren Sketch *Messwerterfassung*.

```
float y1 = height - map(spannungswerte.get(i), minimum, maximum, 50, height-50);
```

Hier werden die Spannungswerte dem Wertebereich von minimum bis maximum zugeordnet. Die *map()*-Funktion bildet diesen Bereich auf den Bereich +50 bis height-50 ab. +50 und -50 deshalb, damit die Grafik nicht an den oberen und unteren Rand des Fensters stößt. Da bei Processing die positive y-Achse nach unten zeigt, steht die Auf- und Entladekurve leider auf dem Kopf. Aus diesem Grund steht ein Minuszeichen vor der *map()*-Funktion. Dadurch wird der Graph an der x-Achse gespiegelt und er wird nun wie gewohnt dargestellt. Nun liegt er aber oberhalb des Fensters. Aus diesem Grund verschieben wir ihn mit *height* wieder nach unten in das Fenster.

In der *for-Schleife* in unserem Sketch *Messwerterfassung* sorgen wir mit der Programmzeile

```
float x1 = width * (i / (float) spannungswerte.size());
```

weiterhin dafür, dass unabhängig von der Aufzeichnungsdauer und damit unabhängig von der Anzahl der Messwerte, die Grafik sich der Fensterbreite anpasst.

Processing-Sketch 02: Messwerterfassung

```
// Bibliothek von Processing für serielle Schnittstellen importieren
// (.* bedeutet alles importieren)
import processing.serial.*;

// Die Variable arduino wird der Klasse Serial zugeordnet
Serial arduino;

// Das dynamische Array mit dem Namen "spannungswerte" wird erstellt
// Bei ArrayList muss float großgeschrieben werden (also Float)
ArrayList<Float> spannungswerte = new ArrayList<Float>();

void setup()
{
  size(800, 400);

  /* Die Werte vom seriellen Port des Arduinos werden der Variablen
     "arduino" zugeordnet.

  Serial.list() gibt ein Array aller angeschlossenen seriellen Geräte
     zurück.
     Hier wird davon ausgegangen, dass nur der Arduino angeschlossen ist

     und er somit an erster Stelle des Arrays liegt.
     Die Baudrate beträgt wie im Arduino-Sketch 9600 */
  arduino = new Serial(this, Serial.list()[0], 9600);
}

void mousePressed()
{
  if (mouseButton == RIGHT) // Wurde die rechte Maustaste gedrückt?
```

```
    spannungswerte.clear(); // Liste der bisher aufgezeichneten
                            // Spannungswerte wird gelöscht
}

void keyPressed()
{
  arduino.write(key); // Sendet über die serielle Schnittstelle die
                      // gedrückte Taste an den Arduino
}

void draw()
{
  background(255, 255, 200);
  fill(0);
  stroke(0);

  // Liest eine Zeile Text als String von der seriellen Schnittstelle
  // des Arduinos ein
  String spannungString = arduino.readStringUntil('\n';);

  // Ist ein neuer Messwert vorhanden, dann wird in die Liste aller
  // Spannungswerte der neue Spannungswert eingetragen
  if (spannungString != null)
    spannungswerte.add(float(spannungString)); // Mit float() werden die
                                               // Strings in float-Zahlen umgewandelt

  // Sucht den kleinsten Spannungswert aus der Liste aller
  // Spannungswerte
  float minimum = 0;
  for (int i = 0; i < spannungswerte.size(); i++)
  {
    if (i == 0)
      minimum = spannungswerte.get(i);

    if (spannungswerte.get(i) < minimum)
      minimum = spannungswerte.get(i);
  }

  // Sucht den größten Spannungswert aus der Liste aller Spannungswerte
  float maximum = 0;
  for (int i = 0; i < spannungswerte.size(); i++)
  {
    if (i == 0)
      maximum = spannungswerte.get(i);

    if (spannungswerte.get(i) > maximum)
      maximum = spannungswerte.get(i);
  }

  // Zeichnet die Spannungswerte als zusammenhängende Linien
  for (int i = 0; i < spannungswerte.size() - 1; i++)
  {
    float x1 = width * (i / (float) spannungswerte.size());
    float y1 = height - map(spannungswerte.get(i), minimum, maximum, 50,
                        height-50);
    float x2 = width * ((i + 1) / (float) spannungswerte.size());
    float y2 = height - map(spannungswerte.get(i + 1), minimum,maximum,
                       50, height-50);
    stroke(255, 0, 0);
    strokeWeight(4);
    line(x1, y1, x2, y2);
```

```
  }

  // Zeichnet den Text für den oberen und unteren Spannungswert
  fill(0, 0, 255);
  textSize(32);
  textAlign(LEFT, BOTTOM);
  text(minimum + "V", 10, height-10);
  textAlign(LEFT, TOP);
  text(maximum + "V", 10, 10);

  if (mouseButton == LEFT)// Wurde die linke Maustaste gedrückt?
  {
    noLoop();
  }
}
```

12.2 Entfernungsmessung

Abbildung 12.5: Entfernungsmessung mit Ultraschall-Modul HC-SR04

Mit unserem Processing-Sketch *Messwerterfassung* können wir alle Daten, die der Arduino an die serielle Schnittstelle liefert, grafisch darstellen. Somit können wir mit entsprechenden Sensoren eine Vielzahl von physikalischen Größen erfassen. Abbildung 12.5 zeigt eine Entfernungsmessung mittels Ultraschall. Im Processing-Sketch *Messwerterfassung* könnten wir bei der Ultraschallmessung auf die folgenden Programmzeilen verzichten. Sie stören aber bei der Ultraschallmessung nicht. Einzig ändern sollten wir im Textteil die Angabe „V“ in „cm“.

```
void keyPressed()
{
arduino.write(key);
}
```

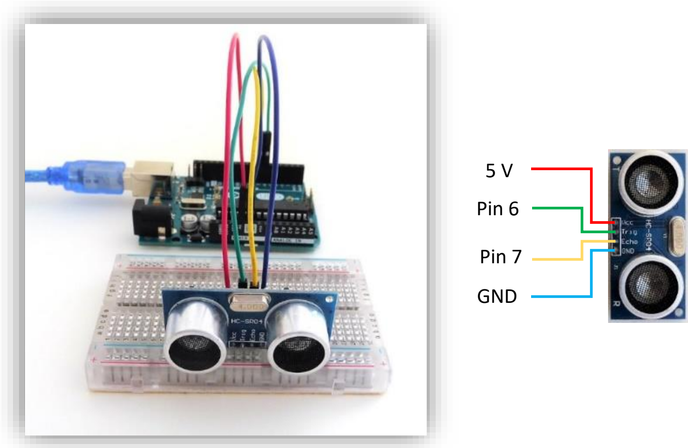


Abbildung 12.6: Ultraschall-Modul HC-SR04 mit Anschlussbelegung für den Arduino Uno

In Abbildung 12.6 ist das Ultraschallmodul HC-SR04 abgebildet. Es besteht aus einem Ultraschall-sender, einem Ultraschallempfänger und der zugehörigen Elektronik. Es kann entsprechen der Abbildung 12.6 direkt an den Arduino angeschlossen werden, da es nur einen Strom von 15 mA benötigt. Mit dem Ultraschallmodul HC-SR04 können Entfernungen von 2 cm bis 4 m mit einer Genauigkeit von 3 mm gemessen werden. Dazu sendet das Modul Ultraschallimpulse mit einer Frequenz von 40 kHz aus. Der Öffnungswinkel des Sendeimpulses beträgt ca. 15°. Um einen Messzyklus auszulösen muss der Arduino ein Triggersignal von mindestens 10 µs an den Triggereingang (*Trg*) des Ultraschallmoduls senden. Nun sendet das Modul acht 40 kHz Impulse aus und wartet anschließend auf die reflektierten Signale. Die Zeit zwischen dem Aussenden und dem Empfang des einzelnen Echos wird über den Anschluss *Echo* dem Arduino in Mikrosekunden mitgeteilt. Hieraus muss nun die Entfernung in cm zum Objekt berechnet werden, damit der Arduino diesen Wert an die serielle Schnittstelle schicken kann. Dies gelingt wie folgt. Bei 20 °C beträgt die Schallgeschwindigkeit v_{Schall} = 343 m/s. Diese rechnen wir nun in cm pro µs um.

$$v_{Schall} = 343\frac{m}{s} = 34300\frac{cm}{s} = 34{,}3\frac{cm}{ms} = 0{,}0343\frac{cm}{\mu s}$$

Mit $s = v \cdot \frac{t}{2} = 0{,}0343\frac{cm}{\mu s} \cdot \frac{t}{2} = 0{,}01715\frac{cm}{\mu s} \cdot t$ können wir nun den Abstand in cm berechnen, wenn wir die Zeit t in µs einsetzen. Wir haben die Zeitangabe durch 2 geteilt, weil das Ultraschall-modul über seinen *Echo*-Pin dem Arduino die Zeit mitteilt, die das Ultraschallsignal für den Hin- und Rückweg braucht. Mithilfe des folgenden Sketches *Ultraschall_Arduino* kann der Arduino diese Information an die serielle Schnittstelle senden. Eine ausführliche Erläuterung findet man im Sketch selbst.

Arduino-Sketch 03: Ultraschall_Arduino

```
void setup()
{
  pinMode(6, OUTPUT); // Der digitale Pin 6 wird als Ausgang
                      // konfiguriert
  pinMode(7, INPUT); // Der digitale Pin 7 wird als Eingang konfiguriert
```

```
  Serial.begin(9600); // Der serielle Port wird initialisiert und es
                      // wird eine Übertragungsrate von 9600 Baud eingestellt
}

void loop()
{
  digitalWrite(6, LOW); // Pin 6 wird auf LOW gesetzt
  delayMicroseconds(2); // Es wird zwei Mikrosekunden gewartet
  digitalWrite(6, HIGH); // Pin 6 wird auf HIGH gesetzt
  delayMicroseconds(10); // Es wird zehn Mikrosekunden gewartet
  digitalWrite(6, LOW); // Pin 6 wird auf LOW gesetzt

  long Zeitdauer = pulseIn(7, HIGH); /* Wenn man Mikrosekunden zählen
  will, dann ist es gut, wenn man den Variablentyp "long" verwendet, da
  die Anzahl der gezählten Mikrosekunden schnell sehr groß sein kann.
  Die Variable "long" kann Zahlenwerte von ca. -10^19 bis ca. +10^19
  speichern */

    float Abstand = Zeitdauer * 0.01715; /* Die gemessene Zeitdauer wird
   in eine Entfernung (Abstand in cm) umgerechnet und in eine float-Zahl
    umgewandelt */

    Serial.println(Abstand); // Der gemessene Abstandswert wird an die
                             // serielle Schnittstelle gesendet

    delay(100); // 100 Millisekunden Pause
}
```

12.3 Zusammenfassung

Arduino — Das Arduino-Board ist ein Mikrocontroller-Board mit einer eigenen, auf Java basierenden Entwicklungsumgebung (IDE), die der Entwicklungsumgebung von Processing sehr ähnelt. Die Arduino IDE unterstützt die Sprachen C und C ++ zur Programmierung des auf dem Board befindlichen Mikrokontrollers.
Über die analogen und digitalen Eingänge des Boards können mittels Sensoren physikalische Größen aufgezeichnet werden und Motoren, LEDs, usw. gesteuert werden.

serielle Schnittstelle — Der USB-Anschluss, der den Arduino mit dem Computer verbindet, ist eine moderne serielle Schnittstelle. Bei einer seriellen Schnittstelle werden die Bits nacheinander übertragen.

import processing.serial.* — Mit *import processing.serial.** lädt man eine Bibliothek für den Zugriff auf die serielle Schnittstelle in seinen Sketch. Das * bedeutet, dass alle Klassen der Bibliothek geladen werden.

switch() und **case** — Die Funktion *switch()* ist eine bequemere Schreibweise für eine if-else-Struktur, mit der man unterschiedliche Fälle (*case*) leicht unterscheiden kann. Nach jedem *case* (Fall) muss jedoch ein *break* stehen, damit klare Entscheidungen getroffen werden können.

map() Mit der Funktion *map()* kann ein Wertebereich verändern werden. Dies soll anhand der folgenden Sketch-Zeile erklärt werden.

```
float x = map(mouseX, 0, width, 100, 300);
```

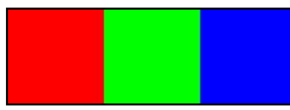
In der runden Klammer wird der x-Wert des Mauszeigers auf den Wertebereich von 0 bis width, also 0 bis Fensterbreite festgelegt. Die *map()*-Funktion bildet diesen Wertebereich für die Variable x auf den kleineren Werteberich 100 bis 300 ab. Bewegt man also den Mauszeiger über die gesamte Fensterbreite, so bewegt sich zum Beispiel ein Kreis mit dem obigen x-Wert als Kreismittelpunkt von 100 Pixel bis 300 Pixel.

long Die Variable *long* kann Zahlenwerte von ca. $\pm 10^{19}$ speichern

12.4 Aufgaben

1. Benutze den Arduino-Sketch *Kondensator_Arduino* und den Processing-Sketch *Messwerterfassung*, um mittels eines LDRs und eines Widerstandes Helligkeitsschwankungen aufzuzeichnen.

2. Die drei Farben Rot, Grün und Blau einer RGB-LED sollen im Processingfenster per Mausklick ein- und ausgeschaltet werden. Schreibe den hierzu passenden Processing-Sketch und den zugehörigen Arduino-Sketch.



   Tipp: Nicht die drei Vorwiderstände vor der RGB-LED vergessen.

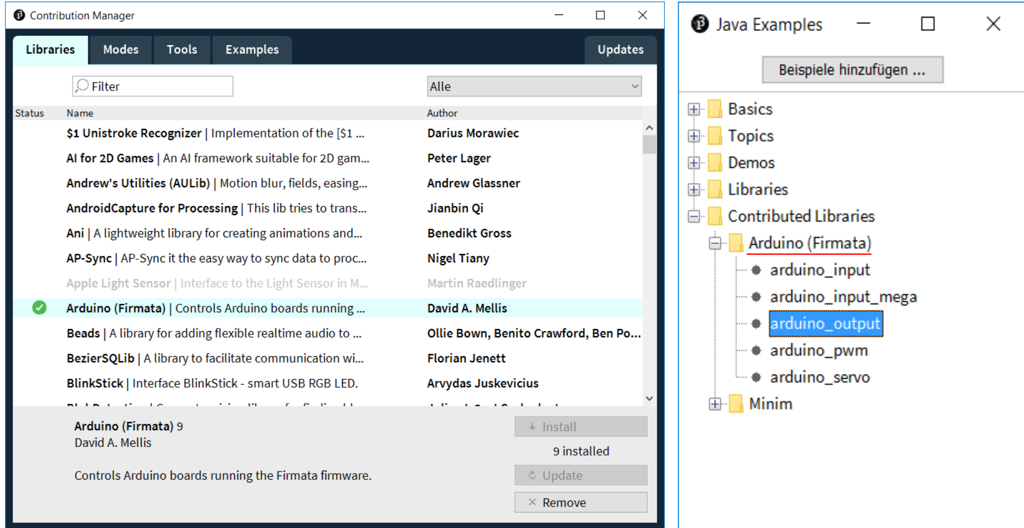
3. Steuere mit der Maus einen Suchscheinwerfer. Klebe dazu eine superhelle weiße LED auf einen Servo (siehe Abbildung). Schreibe den hierzu passen Processing-Sketch und den zugehörigen Arduino-Sketch.

   Tipp: Benutze die Servo-Bibliothek (*#include <Servo.h>*) von Arduino.

4. Im Processingfenster sollen mittels Mausklick die einzelnen Leuchtdioden einer Siebensegmentanzeige (Abbildung rechts) ein- und ausgeschaltet werden.

   Bei den vorherigen Aufgaben haben wir für ihre Lösung stets zwei Sketche geschrieben. Einen Processing-Sketch und einen Arduino-Sketch. Wenn wir uns das Schreiben eines Arduino-Sketches ersparen wollen, dann können wir den Arduino-Sketch *StandardFirmata* auf unser Arduino-Board hochladen. Dieser Sketch ist schon in der Beispielbibliothek der Arduino-Entwicklungsumgebung enthalten (*Datei* → *Beispiele* → *Firmata* → *StandardFirmata*). Damit Processing mit dem Arduino kommunizieren kann, müssen wir nun noch die Bibliothek *Arduino (Firmata)* der Processing-Bibliothek hinzufügen (siehe Abbildung unten links). Aus der Bibliothek *Arduino (Firmata)* rufen wir jetzt den Sketch *arduino_output* auf (siehe Abbildung unten rechts). Nun können wir von Processing aus die einzelnen Leuchtdioden der Siebensegmentanzeige ein- und ausschalten. Voraussetzung hierfür ist jedoch, dass auf der Hardwareseite alles richtig angeschlossen ist und sich vor den einzelnen Leuchtdioden der Siebensegmentanzeige Vorwiderstände befinden.

Auch für andere Aufgabenstellungen findet man in der Bibliothek *Arduino (Firmata)* fertige Sketche.



13 Kommentierte Literatur- und Linkliste

Die unten aufgeführten Links wurden am 26.02.2018 aufgerufen.

Buch in deutscher Sprache

01	Bartmann, Erik 2010	**Processing** Verlag: O'Reilly

Bücher in englischer Sprache

02	Reas, Casey & Fry, Ben 2014	**Processing** **A Programming Handbook for Visual Designers and Artists** Verlag: MIT Press
03	Reas, Casey & Fry, Ben 2015	**Getting Started with Processing** Verlag: MakerMedia
04	Shiffman, Daniel 2015	**Learning Processing** **A Beginner's Guide to Programming Images, Animation, and Interaction** Verlag: Morgan Kaufmann *Materialien zu diesem Buch findet man unter:* http://learningprocessing.com/
05	Shiffman, Daniel 2012	*Ein Buch für Fortgeschrittene* **The Nature of Code** **Simulating Natural Systems with Processing** Verlag: Von Daniel Shiffman selbst verlegt. *Informationen zu diesem Buch findet man unter:* http://natureofcode.com/

Empfehlenswerte Links zu Processing

06	Kipp, Michael	*Eine sehr gute und ausführliche, deutschsprachige Einführung in Processing und Java* **Programmieren lernen – Processing und Java** http://michaelkipp.de/processing/
07	Processing Webseite	*Die größte Fülle an Informationen findet man auf der* ***Homepage von Processing****.* www.processing.org/
08	Shiffman, Daniel	***Video Tutorials von Daniel Shiffman*** *in englischer Sprache findet man unter:* http://learningprocessing.com/ *oder bei YouTube (*www.youtube.com*). Zum Beispiel unter: processing tutorial daniel shiffman Introduction*

Links zum Thema Farbmodelle und Wellenlänge

09 Bruton, Dan **Approximate RGB values for Visible Wavelengths**
http://www.physics.sfasu.edu/astro/color/spectra.html

10 Kern, Uwe **Darstellung sichtbarer Wellenlängen in üblichen Farbmodellen**
Die TEXnische Komödie 4/2005, S. 16–25
www.olos.de/~ukern/publ/tex/pdf/dtk200504.pdf

14 Stichwortverzeichnis

! 28
&& 28
*= 245
|| 18, 28
+= 107
3D-Darstellung 61

A

abs() 150, 167
add 47
aktiver Modus 10
angleBetween() 58
Äquipotenziallinien 73
arc() 145, 127
Arduino 300, 308
Array 81, 106
ArrayList<> 238, 245
atan() 139, 145
Atomkern 252
Attraktor 279, 296
Audioplayer 250

B

background() 2, 9
beginShape() 88, 111, 143
beginShape(TRIANGLE_STRIP) 88
Betaminuszerfall 256
Bibliothek 160, 167
Bibliothek Sound 164
BigDecimal 286, 298
Bildwiederholungsrate 176
blendMode() 185
blendMode(ADD) 199
blendMode(SUBTRACT) 200
Bohrsches Atommodell 213
boolean 42, 45
box() 232, 244
break 207, 220, 302
Brightness 187

C

Cantor-Menge 289
case 302, 308
Chaos 279
Codeblock 5, 10
Color Selector 148, 167, 200
colorMode() 88, 200
continue 102, 107

D

definieren 12, 20
degrees() 139, 145
deklarieren 12, 20
Division durch Null 85, 107
Doppelspalt 182, 202
dot 57
double 284, 298
Drehmoment 60
dynamisches Array 253

E

E 70, 106
ellipse() 3, 9
else if() 23, 29
Endlosschleife 207
endShape() 88, 111, 143
Entropie 225
exklusives ODER 28
exp() 70, 106

F

false 42
Farbmischung 186
Farbraum HSB 88, 187
Farbraum RGB 88
Farbton 187
Federpendel 148
Feldstärke 69
fill() 3, 9
Filme erstellen 122, 144
float 14, 20, 298
Flussdiagramm 131
for-Schleife 74, 106, 156, 183
Fouriersynthese 155
Fraktal 289, 298
frameRate() 12, 16
fromAngle () 63

G

Gedämpfte Schwingung 153
Geigerzähler 250
get() 239, 245
Gleichheitsoperator 42, 46

H

Halbwertszeit 260
Hauptsketch 95, 107
height 74, 106

Hertzsches Gitter 180
Hörtest 159
HSB-Farbraum 200
Hue 187

I

i++ 82, 107
ideales Gas 222
if-Anweisung 18, 20
image() 220
imageMode() 220
import processing.serial.* 302, 308
Induktion 130
initialisieren 12, 20
Instanzvariable 95
int 12, 20
Interferenzmuster 183

K

Kernspaltung 253
keyPressed 122, 144
Kinderpfeile 49
Klasse 95, 107
Kondensator 301
Konsole 3, 20
Konstruktor 96
Koordinatensystem 7
Kreisbahn 127
Kreuzprodukt 53, 60, 65
Kristallgitter 231

L

Längenänderung 269
length 84, 244
Lennard-Jones-Potenzial 233
lights*()* 105, 107
line() 4, 10
Linien verbinden 167
Lissajous-Figuren 158
loadImage 214
loadImage() 219
loadPixels() 182, 199
logische Operatoren 29
logistischen Gleichung 279
lokale Variable 74
long 309

M

mag() 58
map() 303, 309
Massenänderung 269
millis() 15, 20
Minim 159
Mitteilungsbereich 3
Modulo-Operator 133, 144
mouseDragged() 89
mousePressed 122
mouseWheel() 89
mouseX 79, 106
mouseY 79, 106
Movie Maker 122, 144

N

Nebelkammer 247
Nebensketch 95, 107
NICHT 28
noLoop() 18, 20
noStroke() 5, 10

O

objektorientierte Programmierung 95
ODER 18, 28
OOP 95
optimieren 111
Orbitalmodell 216
Ordner Contributed Libraries 160

P

P3D 61
Paarbildung 263
Paarzerstrahlung 265
PDE 1, 9
PImage 214, 219
Pixel-Array 199
point() 4, 10
popMatrix() 40, 45
Potenzialberg 79
Potenzialtrichter 76
pow() 70, 106
println() 3, 10
Processing Development Environment 1, 9
pushMatrix() 40, 45
PVector 47

Q

Quadratwurzel 79
Quantenobjekt 205

R

radians() 27, 29
random() 202, 219
random2D () 63
random3D () 63
randomGaussian() 216, 219
reales Gas 222
Rechtecksignal 166

rect() 4, 10
Referenz 2, 8, 10, 65
rekorder.cue() 250
rekorder.play() 250
rekursive Programmierung 290, 298
return 245
RGB-Farbraum 3, 197
rotate() 36, 45
rotate(radians()) 36, 45
rotateX 61
rotateY 61
rotateZ 61
Rotverschiebung 272
round() 29, 112, 144

S

Saturation 187
saveFrame() 122, 144
Schraubenbahn 123
Schwebung 163
Schwingkreis 150
Selbstähnlichkeit 289
Semikolon 3
serielle Schnittstelle 300, 308
set() 63, 101 107
Siebkette 137
Sierpinski-Dreieck 295
size() 9
skalare Multiplikation 57
skalares Produkt 53, 65
Sketch 2
Sketchoptimierung 144
Soundausgabe 168
Spektren 187
spezielle Relativitätstheorie 269
sphere() 105, 107
Spiralbahn 126
sqrt() 79, 106
statischer Modus 5, 10
stehenden Welle 176
stroke() 5, 10
strokeCap(ROUND**)** 266
strokeCap(SQUARE) 290, 298
strokeWeight() 5, 10
switch() 302, 308

T

text() 15
textAlign(CENTER) 31, 34
Texteditor 2
textSize() 15
timer 15
Toolbar 2
translate() 7, 10, 36
Transparenz 28
triangle() 4, 10, 38, 45
true 42, 45

U

Ultraschall 306
UND 28
Unicode 113, 144
updatePixels() 182, 199

V

Vektor 47
Vektoraddition 48
Vektorpfeile 45
Vektorsubtraktion 51
verblassen 266
verschachtelte for-Schleifen 79
vertex() 88, 111, 143
Verzerrungsfaktor 269
void draw() 5, 10, 20
void setup() 5, 10, 20

W

Wellenmaschine 171
while()-Schleife 206, 220
width 74, 106

Y

y-Achse 7

Z

Zehnerpotenz 70
Zeigerformalismus 205, 210
Zeitdilatation 269
Zufallsgenerator 202
zusammenhängende Linien 154
Zuweisungsoperator 42, 46
Zyklotron 126