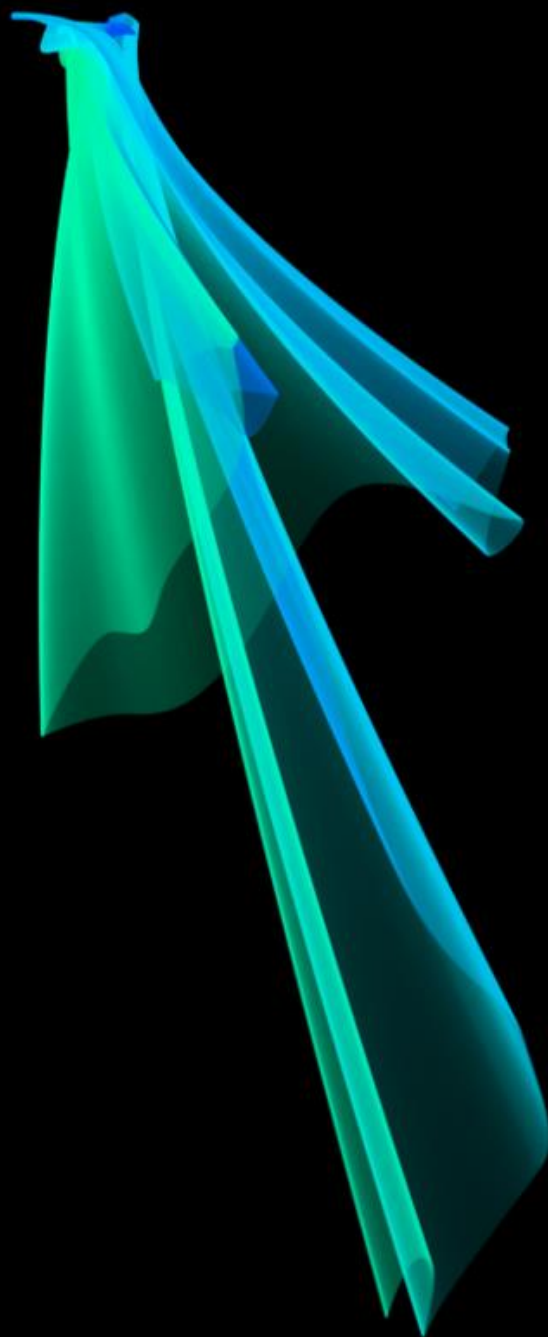


Walter Stein

Programmieren im Kunstunterricht



Walter Stein

Programmieren im Kunstunterricht

3. Auflage 2023

Vorwort

Der Titel des vorliegenden Buches lautet *Programmieren im Kunstunterricht*. Doch müssen Schüler heutzutage überhaupt noch Programmieren lernen? Das zurzeit in allen Medien heiß diskutierte KI-basierte Assistenzsystem *ChatGPT* kann nicht nur in Wechselwirkung mit Menschen Texte in einer sehr gut lesbaren Qualität erzeugen, sondern auch Programmcodes schreiben. Hierzu gibt man in natürlicher Sprache in das Textfeld seinen Programmwunsch ein und nach wenigen Sekunden erhält man den entsprechen Code. Im Kapitel 14 des vorliegenden Buches wird an mehreren Beispielen gezeigt, dass auch bei der Benutzung von *ChatGPT* weiterhin grundlegende Programmierkenntnisse erforderlich sind, da der von *ChatGPT* ausgegebene Code manchmal unvollständig oder fehlerbehaftet ist.

Eine weitere Frage, die man sich heutzutage stellen muss, ist, ob Menschen überhaupt noch Bilder malen sollten. Sprachen manche doch schon vom Tod der Kunst, als ein mittels künstlicher Intelligenz generiertes Bild mit dem Titel *Théâtre d'Opéra Spatial* im August 2022 einen Kunstwettbewerb in den USA gewann. Sein Schöpfer hatte es mit dem Text-zu-Bild-Generator *Midjourney* erstellt. Bei einem Text-zu-Bild-Generator handelt es sich um eine künstliche Intelligenz, in deren Textfeld man eine Kombination von Wörtern eingibt und die dann unter Verwendung riesiger Datenmengen aus dem Internet ein neuartiges Bild erstellt, welches in etwa der eingegebenen Wortkombination entspricht. Die so entstehenden Bilder sind beeindruckend. Wie dies genau funktioniert, wird im Kapitel 15 anhand des Text-zu-Bild-Generators *Dall·E 2* an konkreten Beispielen erklärt.

Walter Stein

Bad Münstereifel im Januar 2023

In der zweiten Ausgabe dieses Buches wurde der Tatsache Rechnung getragen, dass durch den Digitalisierungsschub während der Corona-Pandemie an vielen Schulen verstärkt Tablets angeschafft wurden. Wie man auf ihnen und auch auf einem Smartphone p5.js-Sketches programmiert, wird in Kapitel 13 erläutert. Weiterhin wurde die alte Variablenbezeichnung *var* durch die neuere *let* ersetzt. Der Unterschied und die Gemeinsamkeit werden in Kapitel 5.5 beschrieben. Bei der Erstausgabe dieses Buches konnten Sketches, die Bild- oder Sounddateien enthielten, problemlos mit dem Webbrowser *Firefox* geöffnet werden. Heute verlangen aber alle Webbrowser, dass diese Bild- oder Sounddateien von einem Webserver geliefert werden. Wie man das Problem löst, wird in Kapitel 9.1 gezeigt.

Zum Schluss noch eine Mitteilung für die Menschen, die der digitalen Kunst immer noch sehr skeptisch gegenüberstehen. Im März 2021 hat das renommierte britische Auktionshaus *Christie's* ein NFT-Kunstwerk für 69 Millionen US-Dollar versteigert.

Walter Stein

Bad Münstereifel im März 2021

Programmieren lernen im Kunstunterricht? Hallo!!! Geht's noch?! Was hat der Computer im Kunstunterricht verloren?

Nun, wenn es auch einige Menschen noch nicht gemerkt haben, neben Pinsel, Stift, Spachtel, ... ist der Computer inzwischen ein wichtiges Werkzeug auch für Künstler und Designer geworden. Warum sollte man dieses Werkzeug den Schülern im Kunstunterricht vorenthalten? Ist doch digitale Bildung im Schulunterricht zurzeit ein hochaktuelles Thema. Digitale Bildung ist meines Erachtens aber etwas mehr als im Internet zu recherchieren, Textbearbeitungsprogramme zu nutzen oder einen PowerPoint-Vortrag zu erstellen. Zur Medienkompetenz gehört auch ein Grundverständnis im Programmieren. Wer keine Grundkenntnisse in einer Programmiersprache besitzt, der kann digital auch nicht innovativ und kreativ sein. Warum sind Google, Facebook, Microsoft, Apple, ... keine deutschen Unternehmen?

Kreativität soll gerade im Kunstunterricht gefördert werden. Der Computer selbst ist aber nicht kreativ. Er ist neben Pinsel, Stift, ... aber ein weiteres mächtiges Werkzeug für kreative Menschen, die ihre Ideen in Projekte umsetzen wollen. Der Mensch bleibt also der Gestalter, der den Computer als Werkzeug benutzt.

Im Kunstunterricht können Lehrerinnen und Lehrer ganz konkrete, fachbezogene Aufgaben stellen, die dann mittels Programmierung gelöst werden müssen. Weiterhin besteht die Möglichkeit, fächerübergreifend mit dem Fach Informatik zusammenzuarbeiten. Somit ist das Erlernen einer Programmiersprache kein Selbstzweck, sondern erfolgt, eingebunden in den fachlichen Kontext, zielgerichtet. Dadurch erfahren die Schülerinnen und Schüler, dass ihre Programmierkenntnisse wirklich nützlich sind.

Für unsere Programmierübungen benutzen wir in diesem Buch die JavaScript-Bibliothek **p5.js**. Sie wurde von der Künstlerin und Programmiererin Lauren McCarthy, aufbauend auf der Software Processing, geschaffen und verfolgt auch deren Ziel: Künstlern, Designern, Lehrern sowie Programmieranfängern ein Mittel an die Hand zu geben, mit dem sie recht einfach ihre Ideen digital in Kunstwerke umsetzen können. Mit p5.js benutzt man aber nicht einfach ein fertiges Zeichenprogramm oder Bildbearbeitungsprogramm. Mit p5.js baut man sich seine Werkzeuge zur kreativen Gestaltung selbst und erschließt sich so neue gestalterische Möglichkeiten, deren Ergebnis man in einem Webbrowser der Weltöffentlichkeit zeigen kann.

Das vorliegende Buch mit seinen 96 Beispielprogrammen und 50 Übungsaufgaben ist gedacht als Mutmacher für den Einstieg in die große Welt der digitalen Kunst. Auf seinen 138 Seiten kann es nicht alle Aspekte abdecken, aber wenn der Anfang gemacht ist, vielleicht bekommt man dann Lust auf MEHR!

Walter Stein

Bad Münstereifel im Januar 2019

Inhaltsverzeichnis

1	Einführung	1
2	Erste Übungen mit einem Online-Editor	3
2.1	Grundformen und RGB-Farbraum	3
2.2	Zusammenfassung.....	10
2.3	Übungen	11
3	Farben.....	15
3.1	Farbmischungen	15
3.2	HSB-Farbraum	17
3.3	Zusammenfassung.....	20
3.4	Übungen	21
4	Offline-Editoren.....	23
5	Gestalten mit Variablen, for-Schleifen, if- und else-Anweisungen	25
5.1	Variablen	25
5.2	for-Schleife	27
5.3	if-Anweisung.....	33
5.4	else-Anweisung	35
5.5	let und var	37
5.6	Zusammenfassung.....	39
5.7	Übungen	41
6	Bézierkurven und Perlin noise.....	45
6.1	Bézierkurven.....	45
6.2	Perlin noise 1D.....	50
6.3	Perlin noise 2D.....	55
6.4	Zusammenfassung.....	59
6.5	Übungen	60
7	Eigene Funktionen.....	63
7.1	Smileys.....	63
7.2	Bienenwabe.....	65
7.3	Herzballon	67
7.4	Vier Kerzen	69
7.5	Mausmalprogramm.....	72
7.6	Zusammenfassung.....	73
7.7	Übungen	74

8	Arrays.....	77
8.1	Eindimensionales Array.....	77
8.2	Zweidimensionales Array.....	80
8.3	Zusammenfassung.....	86
8.4	Übungen.....	87
9	Bildbearbeitung.....	91
9.1	Erste Schritte.....	91
9.2	Pixelarray.....	96
9.3	Zusammenfassung.....	100
9.4	Übungen.....	101
10	Objektorientierte Programmierung.....	103
10.1	Warum einfach, wenn es auch komplizierter geht?.....	103
10.2	Häuser und Autos.....	105
10.3	Zusammenfassung.....	111
10.4	Übungen.....	112
11	Rekursive Programmierung.....	113
11.1	Punkte und Quadrate.....	113
11.2	Fraktale.....	115
11.3	Zusammenfassung.....	119
11.4	Übungen.....	120
12	Sounds und Sliders.....	125
12.1	Sounds.....	125
12.1.1	Ballspiel mit Ton.....	125
12.1.2	Geister.....	127
12.1.3	Lissajous Figuren.....	130
12.2	Sliders.....	132
12.3	Zusammenfassung.....	134
12.4	Übungen.....	134
13	Sketche auf dem Tablet und dem Smartphone programmieren.....	137
13.1	Sketche online schreiben.....	137
13.2	Sketche offline schreiben.....	139
13.3	Fingerübungen.....	140
14	Programmieren mit der KI ChatGPT.....	143
14.1	Erster Versuch.....	143
14.2	Zweiter Versuch.....	147
14.3	Fazit.....	150

15	Bilder generieren mittels dem Text-zu-Bild-Generator Dall-E 2.....	151
16	Ausblick.....	155
17	Link- und Literaturverzeichnis	157
18	Stichwortverzeichnis	159

1 Einführung

Du bist schlau! Der Computer ist dumm! Wenn du mit einem Menschen kommunizierst, dann versteht er dich auch, wenn deine Formulierungen nicht zu einhundert Prozent präzise sind. Der dumme Computer versteht dich aber nur, wenn dein Programmcode zu einhundert Prozent präzise ist. Hast du ein Wort großgeschrieben, was kleingeschrieben werden muss, hast du ein Komma vergessen oder eine Klammer nicht geschlossen, dann funktioniert dein Programm nicht. Im günstigsten Fall zeigt dir dein Editor, in welchem du deine Programmzeilen schreibst, die gemachten Fehler an. Dann kannst du die Fehler verbessern. Manchmal zeigt der Editor dir keinen Fehler an und dein Programm funktioniert trotzdem nicht. Dann hast du wahrscheinlich einen Logikfehler gemacht. Dies ist natürlich alles sehr ärgerlich, aber es hat auch eine positive Seite. Das Schreiben von Computerprogrammen zwingt dich zur Korrektheit und fördert dein logisches Denkvermögen.

Auch ist es wichtig, dass dein Programmcode ordentlich geschrieben ist, so dass er auch für fachkundige Fremde gut zu lesen ist. Größeren Programmcode solltest du an wichtigen Stellen auch mit Kommentaren versehen, damit auch du dich zu einem späteren Zeitpunkt in deinem Programm wieder zurechtfindest. Vor einzeilige Kommentare setzt man zwei Schrägstriche *// Kommentar*. Längere Kommentare setzt man zwischen zwei Schrägstriche mit Sternchen */* Kommentar */*.

Wie in anderen Sprachen auch, musst du, um eine Programmiersprache zu beherrschen, Vokabeln lernen. Diese Vokabeln sind Wörter oder deren Abkürzungen, die aus dem englischen Sprachraum kommen. Hier einige Beispiele: *createCanvas* (erstelle eine Leinwand) oder *background* (Hintergrund) oder die Abkürzung *rect* für *rectangle* (Rechteck). Die aufgeführten Beispiele sind Funktionen, die etwas für dich tun, wenn du sie mit den nötigen Werten fütterst. Schreibst du *createCanvas(400, 400)*; so stellt dir p5.js eine 400 Pixel mal 400 Pixel große Leinwand im Webbrowser zur Verfügung, auf der du dich künstlerisch austoben kannst. Schreibst du *background(0)*; so wird deine Leinwand schwarz eingefärbt. Damit die Funktionen dies für dich tun, steckt hinter jeder Funktion, für dich unsichtbar, eine Menge JavaScript-Code. Diesen hat die Künstlerin und Programmiererin Lauren McCarthy für alle Nutzer von p5.js geschrieben, damit diese ihre Ideen leichter in ihre Kunstwerke umsetzen können. Dafür gebührt ihr ein dickes Dankeschön!

Ein weiterer wichtiger Bestandteil deines Programms sind die Variablen. Variablen sind in der Regel Platzhalter für Zahlen. Hier bist du bezüglich der Wahl des Namens frei. Wichtig ist nur, dass man vor den Namen die Abkürzung *let* schreibt, damit p5.js weiß, dass es sich hier um eine Variable handelt. Auch sollte man sich an die Regel halten, dass man den Namen einer Variablen klein schreibt.

Programme nennt man bei p5.js Sketche (Skizzen). Deine Sketche kannst du online im Webeditor von p5.js schreiben. Ihn findet man unter dem folgenden Link: <https://editor.p5js.org/>. Wenn du aber Sketche offline schreiben willst, dann benötigst du einen beliebigen Texteditor und du muss das notwendige Softwarepaket von der Homepage von p5.js herunterladen. Am besten lädt man sich die komplette Library herunter: <https://p5js.org/download/>. Es lohnt, sich auf der Homepage von p5.js (<https://p5js.org>) einmal gründlich umzusehen. Hier findest du Anleitungen, viele Sketche und in der Referenz werden dir alle Funktionen anhand von Beispielen erklärt.

2 Erste Übungen mit einem Online-Editor

2.1 Grundformen und RGB-Farbraum

Die ersten Gehversuche mit p5.js machen wir mit dem folgenden Online-Editor:

<https://editor.p5js.org/>

Dies ist die einfachste und schnellste Art zu ersten Ergebnissen zu kommen. Öffnen wir also unseren Internetbrowser, geben den obigen Link ein und starten unsere ersten Gehversuche im **Online-Editor**. Abbildung 2.1 zeigt den geöffneten Online-Editor mit einem einfachen Beispiel-Sketch.

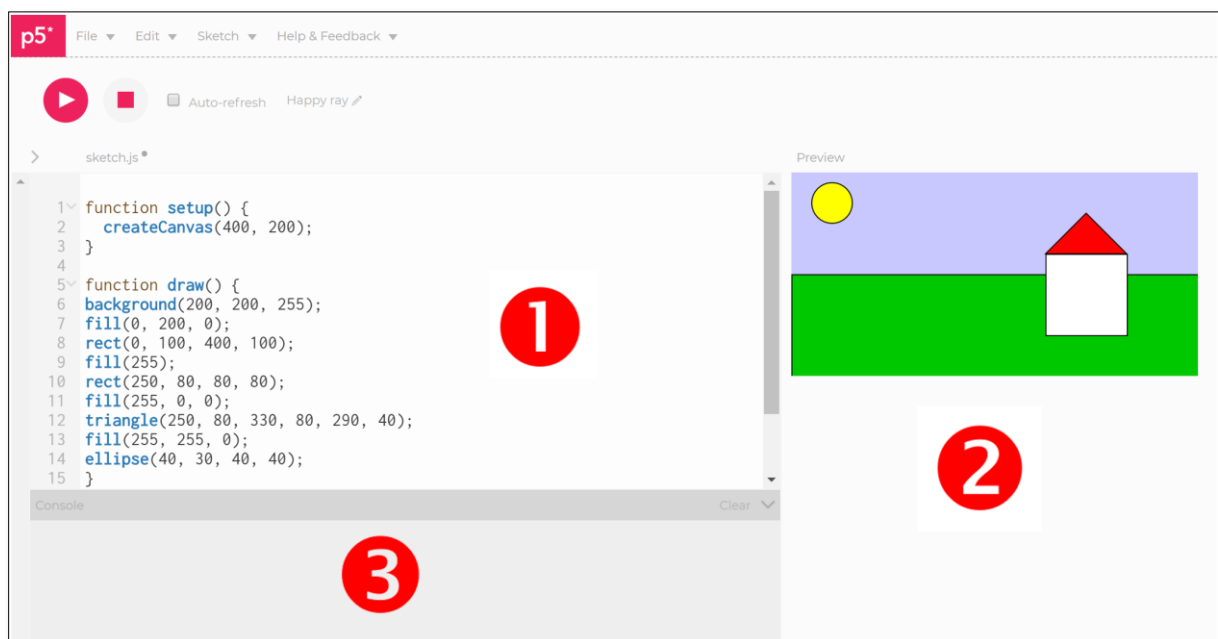


Abbildung 2.1: Der Online-Editor mit einem Beispiel-Sketch

Im Bereich 1 (*code editor*) gibt man seinen Code ein. Drückt man anschließend auf die Play-Taste (▶), so wird der Code im Bereich 2 (Vorschaufenster) solange ausgeführt, bis man auf die Stopp-Taste (■) drückt. Wenn man bei der Codierung Fehler macht, so werden diese im Bereich 3 (Konsole) angezeigt.

Schauen wir uns den obigen Beispiel-Sketch nun etwas genauer an. Der Sketch besteht aus den zwei Hauptfunktionen **function setup()** und **function draw()**. Alles, was zwischen den geschweiften Klammern von **function setup()** steht, wird nur einmal ausgeführt. Den Programmcode zwischen den geschweiften Klammern bezeichnet man als **Codeblock**. In unserem Beispiel wird mit der Funktion **createCanvas(400, 200)** die Größe unserer Leinwand (**Canvas heißt übersetzt Leinwand**) auf 400 Pixel in x-Richtung und 200 Pixel in y-Richtung festgelegt. Mit den Funktionen zwischen den geschweiften Klammern von **function draw()** legen wir fest, was auf der bei **function setup()** generierten Leinwand gezeichnet werden soll. Alles was zwischen den beiden geschweiften Klammern von **function draw()** steht, wird nacheinander Schritt für Schritt abgearbeitet. Ist das Ende erreicht, beginnt der Abarbeitungsvorgang wieder von vorne. Solange, bis dies durch eine Funktion oder durch das Drücken der Stopp-Taste beendet wird. Dies können wir leicht mittels eines kurzen Sketches überprüfen. Dazu benötigen wir die Funktion **print()**. Sie schreibt uns Texte oder Wertangaben in die Konsole. Texte

müssen dabei innerhalb der runden Klammern von `print()` in Anführungszeichen gesetzt werden (siehe Abb. 2.2).

Tragen wir nun zwischen die geschweiften Klammern bei `function setup()` die Funktion `print(„Hallo“)` ein und zwischen die geschweiften Klammern von `function draw()` schreiben wir `print(„Guten Tag“)`. Wenn wir jetzt auf die Play-Taste drücken, sehen wir, dass das Wort „Hallo“ nur einmal in die Konsole geschrieben wird, während „Guten Tag“ solange in die Konsole geschrieben wird, bis wir die Stopp-Taste drücken (siehe Abb. 2.2). Merken wir uns: **Bei `function setup()` wird alles, was zwischen den geschweiften Klammern steht, nur einmal ausgeführt, während bei `function draw()` das, was zwischen den geschweiften Klammern steht, immer wieder erneut ausgeführt wird.** Dies gilt ganz allgemein, also nicht nur für den Text in der Konsole, sondern auch für die Zeichenelemente im Vorschaufenster. Da p5.js aber nicht für Schriftsteller, sondern hauptsächlich für Künstler entwickelt wurde, wollen wir uns nun den grafischen Elementen aus dem Beispiel von Abbildung 2.1 widmen.

```
1 function setup() {
2   print("Hallo");
3 }
4
5 function draw() {
6   print("Guten Tag");
7 }
8
```

Console

Hallo

30 Guten Tag

29 Guten Tag

29 Guten Tag

29 Guten Tag

29 Guten Tag

29 Guten Tag

28 Guten Tag

29 Guten Tag

29 Guten Tag

Abbildung 2.2: Unterschied zwischen `function setup()` und `function draw()`

Wie oben schon erwähnt, wurde in unserem Beispiel-Sketch mit der Funktion `createCanvas(400, 200)` die Größe unserer Leinwand auf 400 Pixel in x-Richtung und 200 Pixel in y-Richtung festgelegt. Um diese Funktion zu beenden, wurde nach der schließenden runden Klammer ein Semikolon gesetzt. **Jede Zeile in unserem Sketch wird mit einem Semikolon beendet.**

Mit der Funktion `background()` legen wir die Hintergrundfarbe der Leinwand fest. Schreibt man zwischen die runden Klammern nur eine Zahl, so kann man hiermit Grautöne sowie die Farben Schwarz und

Weiß darstellen. 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. p5.js 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. Wollen wir einen weißen Hintergrund, dann können wir `background(255)` oder `background(255, 255, 255)` schreiben. Die Farben Rot, Grün und Blau ergeben additiv gemischt die Farbe Weiß. In unserem Beispiel-Sketch wollten wir für den Himmel ein helles Blau. Deshalb haben wir `background(200, 200, 255)` geschrieben.

Mit der Funktion `fill()` kann man die dieser Funktion nachfolgenden Formen mit Farbe füllen. Auch hier verwenden wir den RGB-Farbraum. Mit `fill(255, 0, 0)` füllen wir einen Kreis mit einem satten Rot. In unserem Beispiel-Sketch haben wir uns die Sonne als gelben Kreis gewünscht, deshalb haben wir `fill(255, 255, 0)` geschrieben (Abb. 2.1, Zeile 13), denn Rot und Grün, additiv gemischt, ergibt bekanntlich die Farbe Gelb.

Den Kreis selbst erstellen wir mit der Funktion `ellipse(40, 30, 40, 40)`. 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 keinen Kreis, sondern eine Ellipse.

Entgegen unserer Gewohnheit zeigt bei p5.js die positive y-Achse des Koordinatensystems nach unten und der Koordinatenursprung liegt in der linken oberen Ecke des Anzeigefensters (blau in Abbildung 2.3). Dies müssen wir bei unseren Zeichnungen berücksichtigen.

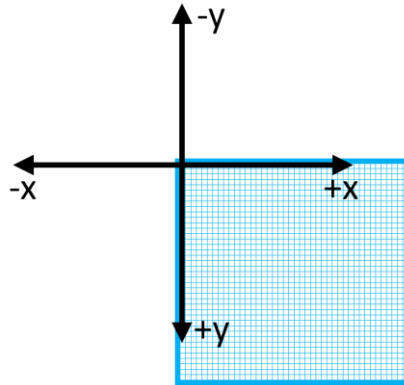


Abbildung 2.3: Lage des Anzeigefensters im Koordinatensystem von p5.js

Den grünen Rasen und die weiße Hauswand in unserem Beispiel-Sketch haben wir als Rechteck gezeichnet. Ein Rechteck zeichnet man mit der Funktion **rect(x, y, Breite, Höhe)**. Mit x und y gibt man den linken oberen Punkt des Rechteckes an. Danach folgend die Angaben für die Breite und die Höhe des Rechtecks (siehe Abb. 2.4 rechts). Für den grünen Rasen in unserem Beispiel-Sketch gilt: **rect(0, 100, 400, 100)** und für die weiße Hauswand gilt: **rect(250, 80, 80, 80)**. Da **function draw()** hintereinander Zeile für Zeile abarbeitet, wird der himmelblaue Hintergrund teilweise vom grünen Rasen überdeckt. Dieser wird zum Teil von der weißen Hauswand überdeckt. Die Reihenfolge der Codezeilen muss also wohlüberlegt sein.

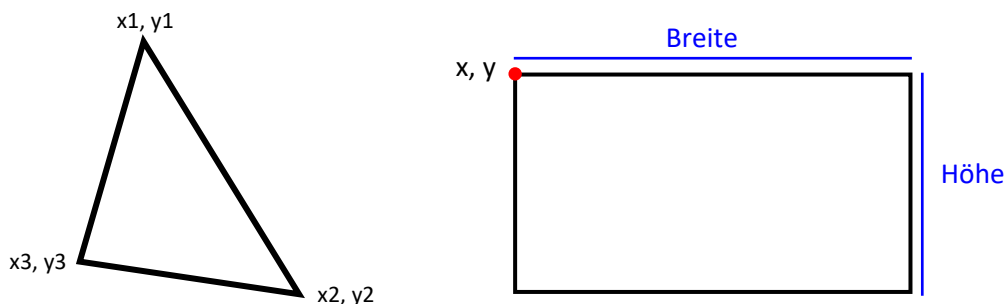


Abbildung 2.4: Konstruktionsangaben für ein Dreieck und ein Rechteck

Das rote Dach des Hauses in unserem Beispiel-Sketch wurde als Dreieck gezeichnet. Ein Dreieck zeichnet man mit der Funktion **triangle(x1, y1, x2, y2, x3, y3)**. Was die x- und y-Werte bedeuten, sehen wir in Abbildung 2.4.

Außer den in der Abbildung 2.1 verwendeten Grundformen gibt es noch weitere Grundformen wie Punkt, Linie, Viereck und Winkel.

Ein Punkt in der Ebene besitzt nur die beiden Koordinatenwerte x und y. Die Funktion hierfür lautet **point(x, y)**. Eine Linie (genau genommen Strecke) ist die Verbindung zwischen zwei Punkten. Die Funktion hierfür lautet **line(x1, y1, x2, y2)**. Die Farbe von Punkten und Linien, sowie die Farben der Umriss von Figuren wie Kreis, Rechteck, ... bestimmen wir nicht mit **fill()**, sondern mit **stroke()**. Auch

hier verwenden wir den RGB-Farbraum. Will man keine Umriss um Figuren zeichnen, dann verwendet man die Funktion **noStroke()**. Die Dicke der Linien und Umriss sowie den Durchmesser der Punkte in Pixel legt man mit **strokeWeight()** fest. Den gewünschten Pixelwert schreibt man zwischen die runden Klammern.

Die Funktion für ein Viereck (siehe Abb. 2.5) ist leicht zu verstehen. Sie lautet **quad(x1, y1, x2, y2, x3, y3, x4, y4)**.

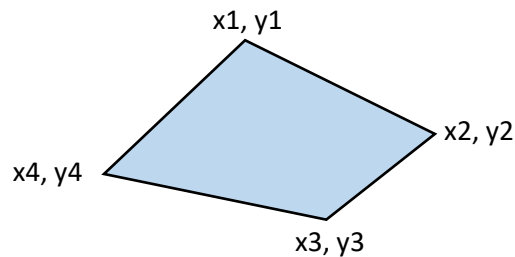


Abbildung 2.5: Koordinaten eines Vierecks

Manchmal möchte man Dinge zeichnen, zum Beispiel eine Krone, die nicht den oben beschriebenen Grundformen entsprechen. Wie dies gelingt, zeigt Abbildung 2.6. Man setzt mit der Funktion **vertex()** unsichtbare Punkte auf seine Leinwand, indem man die x- und y-Werte bei **vertex()** in die runden Klammern schreibt. Diese Punkte verbindet p5.js dann mit Linien. Damit p5.js weiß, dass es dies tun soll, setzt man die **vertex**-Funktionen zwischen **beginShape()** und **endShape()** (siehe Abb. 2.6). Bei **endShape()** kann man wählen, ob die Figur offen oder geschlossen werden soll. Lässt man die Klammer leer (Abb. 2.6, links), dann bleibt die Figur offen. Schreibt man **endShape(CLOSE)**, dann wird die Figur geschlossen (Abb. 2.6, rechts).

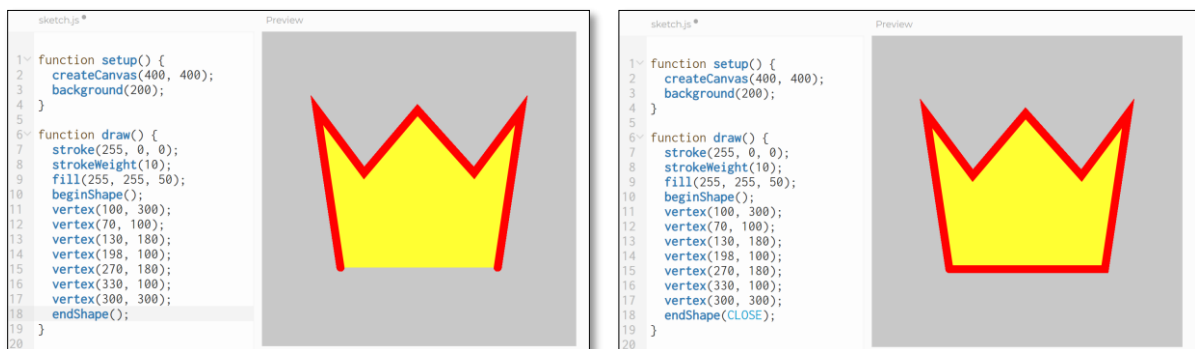


Abbildung 2.6: Offene und geschlossene Krone

Wenn man ein Stück Käsekuchen zeichnen möchte, dann könnte man versucht sein, die Funktion **triangle()** zu benutzen. Da ein Stück Käsekuchen in der Realität aber einen gebogenen Rand hat, liefert dieser Versuch kein befriedigendes Ergebnis. Realitätsnähere Ergebnisse erhält man dagegen mit dem Sketch von Abbildung 2.7.

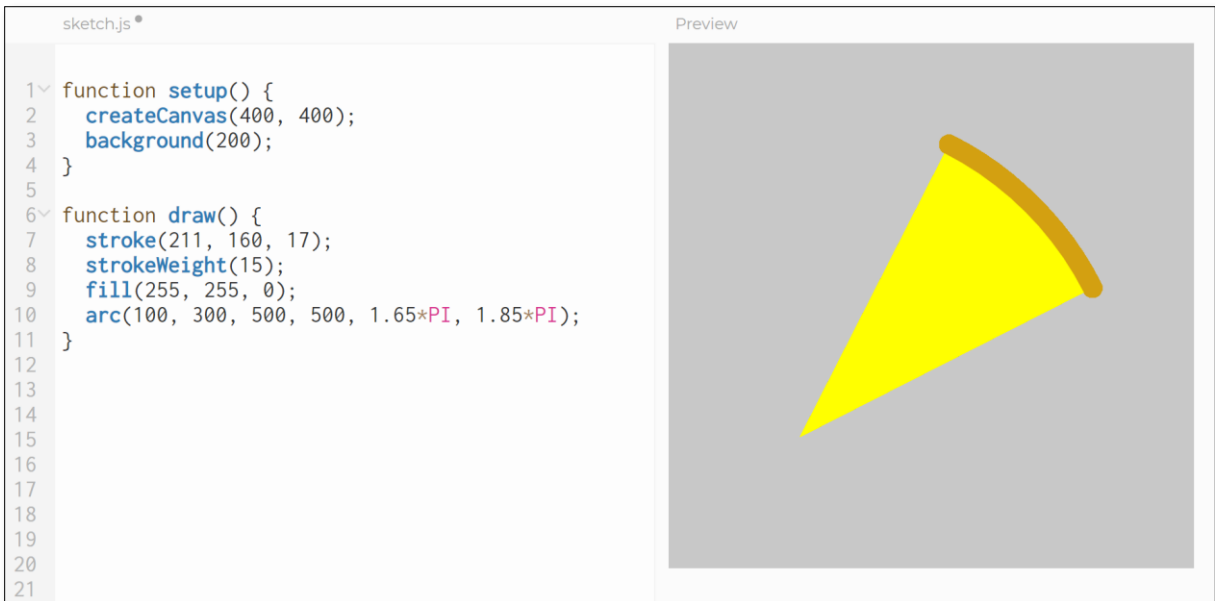


Abbildung 2.7: Ein Stück Käsekuchen, gezeichnet mit der Funktion `arc()`

Wie muss man aber vorgehen, wenn man einen Kreisbogen (Ellipsenbogen) oder einen Kreisabschnitt (Ellipsenabschnitt) zeichnen will? Dazu benötigt man die Funktion **`arc(x, y, Breite, Höhe, Startwinkel, Endwinkel)`**. Mit `x` und `y` legt man den Mittelpunkt des Kreises fest. Mit den Pixelwerten für die Breite und Höhe den Durchmesser. Für den Startwinkel und den Endwinkel muss man die Winkelwerte im Bogenmaß angeben (siehe Abb. 2.7). Wer mit Winkelangaben im Bogenmaß nicht vertraut ist, kann mit der Funktion **`radians(Winkel in Grad)`** Gradzahlen ins Bogenmaß umrechnen lassen. In Zeile 10 von Abbildung 2.7 würde man dann Folgendes schreiben: `arc(100, 300, 500, 500, radians(1.65*180), radians(1.85*180))`; da der Winkel π dem Winkel 180° entspricht. Zu beachten ist allerdings, dass bei `p5.js`, anders als in der Mathematik, der Winkel im Uhrzeigersinn gezählt wird (siehe Abb. 2.8.). Der Bereich zwischen Startwinkel und Endwinkel kann mit `fill()` farbig gefüllt oder leer gelassen werden.

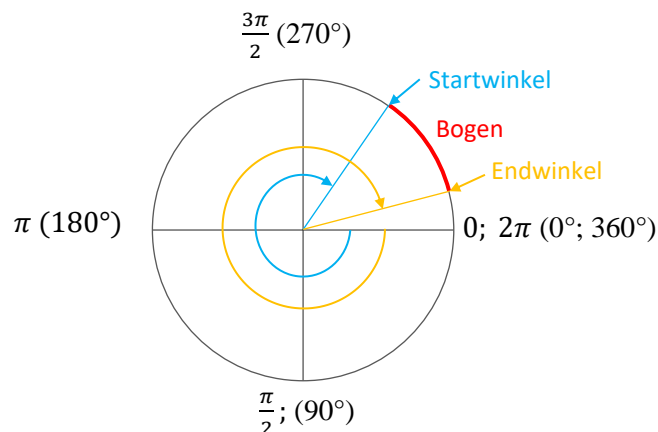


Abbildung 2.8: Winkelzählrichtung bei `p5.js`

Das Bild einer Krone anzufertigen hat uns nicht vor große Herausforderungen gestellt. Wie gehen wir aber vor, wenn wir ein Objekt mit geschwungenen Linien, wie zum Beispiel den Tonkrug von Abbildung 2.9 zeichnen wollen?

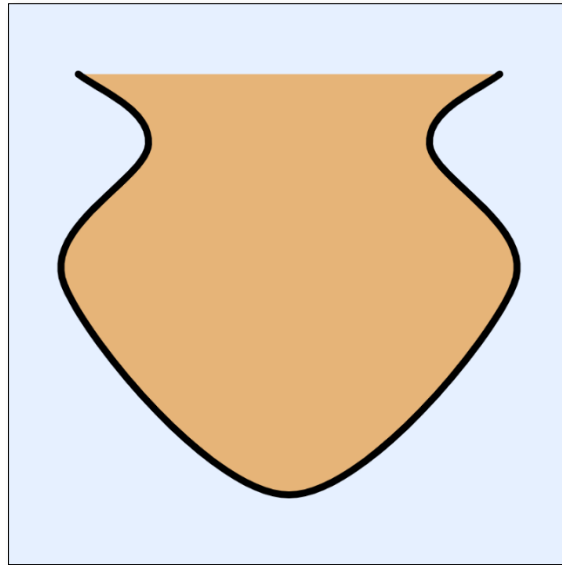


Abbildung 2.9: Bild eines einfachen Tongefäßes

Hier hilft uns die Funktion **curveVertex()**. Wie auch bei unserer Krone, müssen wir geeignete x- und y-Koordinaten für die entscheidenden Punkte der zu erstellenden Form wählen (siehe Sketch in Abb. 2.10). Diese Sketchzeilen werden wieder von *beginShape()* und *endShape()* eingeschlossen. Die so gesetzten Punkte werden jedoch nicht wie bei der Krone mit geraden Linien verbunden, sondern mit geschwungenen Linien. Schauen wir uns die Lage der Punkte in Abbildung 2.11 an.

sketch.js	Preview
<pre>1 function setup() { 2 createCanvas(400, 400); 3 } 4 5 function draw() { 6 background(230, 240, 255); 7 8 stroke(0) 9 strokeWeight(5); 10 fill(230, 180, 120); 11 12 beginShape(); 13 curveVertex(10, 30); 14 curveVertex(50, 50); 15 curveVertex(100, 100); 16 curveVertex(40, 200); 17 curveVertex(200, 350); 18 curveVertex(360, 200); 19 curveVertex(300, 100); 20 curveVertex(350, 50); 21 curveVertex(390, 30); 22 endShape(); 23 } 24</pre>	A preview of the clay pot shape rendered in a sketch environment. The pot is brown with a black outline, matching the image in Abbildung 2.9. It is centered on a light blue background.

Abbildung 2.10: Sketch für die Erstellung des Tongefäßes

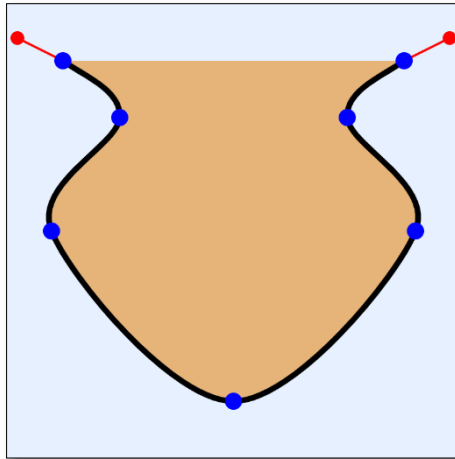


Abbildung 2.11: Tonkrug mit blauen Kurvenstützpunkten und roten Anfangs- und Endpunkten.

Im Sketch von Abbildung 2.10 sehen wir, dass wir neun Punkte zwischen `beginShape()` und `endShape()` gesetzt haben. Durch schwarze geschwungene Linien verbunden wurden aber nur sieben Punkte. In der Abbildung 2.11 nachträglich blau dargestellt. Nicht zum schwarzen Linienzug gehören der erste Punkt (Startpunkt) und der letzte Punkt (Endpunkt) zwischen `beginShape()` und `endShape()`. In der Abbildung 2.11 sind es die beiden roten Punkte; hier von Hand eingezeichnet und mit roten Linien verbunden. Auch wenn sie nicht zur schwarzen Linie gehören, so beeinflusst ihre Lage doch den Beginn und das Ende der schwarzen Linie. Durch Änderung dieser Werte kann man sich leicht hiervon überzeugen. Wenn man solche Formen mit geschwungenen Linien erzeugt, dann ist es oft hilfreich, wenn man im Online-Editor bei *Auto-refresh* ein Häkchen setzt.

Angemerkt sei noch, dass es eine weitere Funktion mit dem Namen **bezier()** gibt, um besonders elegant geschwungene Linien zu erzeugen. Diese, nicht einfach zu bedienende Funktion, behandeln wir in einem späteren Kapitel. Ungeduldige Leser können aber schon mal einen Blick in die Referenz von p5.js werfen (<https://p5js.org/reference/>) und den Suchbegriff *Curves* eingeben.

Zum Schluss dieses Kapitels soll noch eine kleine Ergänzung zum RGB-Farbraum gemacht werden. Außer den drei Zahlen für Rot, Grün und Blau, kann bei `fill()` noch eine vierte Zahl, der **Alpha-Wert** eingefügt werden `fill(0, 0, 255, 155)`. Mehr dazu im Text zu der Abbildung 2.12.

```

1  function setup() {
2    createCanvas(400, 400);
3  }
4
5  function draw() {
6    background(255);
7    fill(255, 0, 0);
8    ellipse(150, 200, 200, 200);
9    fill(0, 0, 255, 155);
10   ellipse(250, 200, 200, 200);
11 }
12

```

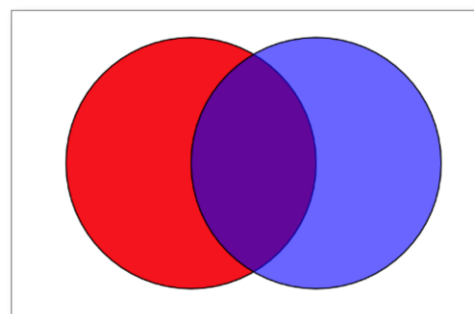


Abbildung 2.12: Mit dem Alpha-Wert (vierte Zahl bei `fill()`) stellt man die Transparenz ein.

Mit dem Alpha-Wert stellt man die Transparenz einer Farbe ein. In unserem Beispiel von Abbildung 2.12 die Transparenz von Blau (Zeile 9). **Je kleiner der Alpha-Wert, desto größer ist die Transparenz. D.h., desto geringer ist ihre Deckkraft. 0 bedeutet 0% Deckkraft und 255 bedeutet 100% Deckkraft.**

2.2 Zusammenfassung

Fassen wir zusammen, was wir in Kapitel 2 gelernt haben, bzw. gelernt haben sollten.

Online-Editor	Im Online-Editor https://editor.p5js.org/ schreibt man seinen Code in für Menschen lesbarer Sprache. Drückt man auf die Play-Taste, so wird der Code im Vorschaufenster ausgeführt. Wenn man bei der Codierung Fehler macht, so werden diese in der Konsole angezeigt.
function setup()	Alles, was zwischen den geschweiften Klammern von <i>function setup()</i> steht, wird nur einmal ausgeführt.
function draw()	Alles, was zwischen den beiden geschweiften Klammern von <i>function draw()</i> steht, wird nacheinander Schritt für Schritt abgearbeitet. Ist das Ende erreicht, beginnt der Abarbeitungsvorgang wieder von vorne.
Codeblock	Den Programmcode zwischen den geschweiften Klammern bezeichnet man als Codeblock.
createCanvas()	Mit <i>createCanvas()</i> legt man die Breite und die Höhe der Leinwand fest.
Koordinatensystem	Entgegen unserer Gewohnheit zeigt bei p5.js die positive y-Achse des Koordinatensystems nach unten und der Koordinatenursprung liegt in der linken oberen Ecke des Anzeigefensters
background()	Mit <i>background()</i> bestimmt man die Hintergrundfarbe der Leinwand. Steht <i>background()</i> bei <i>function draw()</i> , dann wird der Hintergrund bei jedem Durchlauf von <i>function draw()</i> neu gezeichnet. Steht <i>background()</i> bei <i>function setup()</i> , dann wird der Hintergrund nur einmal gezeichnet.
fill()	Mit <i>fill()</i> 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).
Alpha-Wert	Mit dem Alpha-Wert (vierte Zahl bei <i>fill()</i>) stellt man die Transparenz einer Farbe ein. Je kleiner der Alpha-Wert, desto größer ist die Transparenz. D.h., desto geringer ist ihre Deckkraft. 0 bedeutet 0% Deckkraft und 255 bedeutet 100% Deckkraft.
ellipse()	Mit der Funktion <i>ellipse()</i> kann man Ellipsen und Kreise zeichnen. Hierzu schreibt man vier Zahlen in die Klammer: <i>ellipse(x-Wert des Mittelpunktes, y-Wert des Mittelpunktes, Durchmesser in x-Richtung, Durchmesser in y-Richtung)</i> .
print()	Mit <i>print()</i> kann man Wörter oder Daten in die Konsole schreiben lassen.
point()	Mit <i>point(x, y)</i> zeichnet man einen Punkt.
line()	Mit <i>line(x1, y1, x2, y2)</i> zeichnet man eine Linie (Strecke).
rect()	Mit <i>rect(x-Wert des oberen linken Eckpunktes, y-Wert des oberen linken Eckpunktes, Breite, Höhe)</i> zeichnet man ein Rechteck.

triangle()	Mit <i>triangle(x1, y1, x2, y2, x3, y3)</i> zeichnet man ein Dreieck.
stroke()	Mit <i>stroke()</i> legt man die Farben von Linien und die Farben der Umrise von Figuren fest.
strokeWeight()	Mit <i>strokeWeight()</i> bestimmt man die Dicke von Punkten und Linien in Pixel.
noStroke()	Mit <i>noStroke()</i> sorgt man dafür, dass keine Umrise um Figuren gezeichnet werden.
quad()	Mit der Funktion <i>quad(x1, y1, x2, y2, x3, y3, x4, y4)</i> kann man ein beliebiges Viereck zeichnen.
beginShape()	Wenn man eine eigene Form zeichnen will, dann eröffnet man diesen Vorgang mit <i>beginShape()</i> .
endShape()	Wenn man die Punkte für seine eigene Form gesetzt hat, dann schließt man diesen Vorgang mit <i>endShape()</i> . p5.js verbindet diese Punkte dann durch Linien. Die Linien beginnen beim zweiten Punkt und enden beim vorletzten Punkt.
endShape(CLOSE)	Möchte man eine geschlossene Form erstellen, dann schreibt man zwischen die runden Klammern von <i>endShape()</i> das Wort CLOSE.
vertex()	Mit <i>vertex()</i> setzt man unsichtbare Punkte für die Erstellung der eigenen Form. Dazu schreibt man die x- und y-Werte der jeweiligen Punkte zwischen die runden Klammern. Die <i>vertex</i> -Funktionen stehen stets zwischen <i>beginShape()</i> und <i>endShape()</i> .
arc()	<i>arc(x, y, Breite, Höhe, Startwinkel, Endwinkel)</i> Mit x und y legt man den Mittelpunkt des Kreises fest. Mit den Pixelwerten für die Breite und Höhe den Durchmesser. Für den Startwinkel und den Endwinkel muss man die Winkelwerte im Bogenmaß angeben.
radians()	Mit der Funktion <i>radians()</i> kann man Gradzahlen ins Bogenmaß umrechnen lassen. Dazu schreibt man die Gradzahl zwischen die Klammern.
curveVertex()	Mit Hilfe der Funktion <i>curveVertex()</i> kann man geschwungene Linien erzeugen. Man benötigt mindestens vier <i>curveVertex</i> -Funktionen um eine geschwungene Linie zwischen dem unsichtbaren zweiten Kurvenpunkt und dem vorletzten Punkt zu erzeugen. Die <i>curveVertex</i> -Funktionen müssen sich stets zwischen <i>beginShape()</i> und <i>endShape()</i> befinden.

2.3 Übungen

Wenn man etwas weiß, dann bedeutet dies noch lange nicht, dass man das Gelernte auch gut kann. Wenn man zum Beispiel weiß, wie man Fußball oder Schach spielt, dann folgt daraus nicht automatisch, dass man ein guter Fußballspieler, bzw. ein guter Schachspieler ist. Neben Wissen bedarf es stets der Übung. So ist dies auch beim Programmieren. Löse deshalb die folgenden Übungsaufgaben und beginne danach mit eigenen kreativen Übungen.

Deine im Online-Editor erstellten Lösungen kannst du nicht auf deinem PC abspeichern. Wenn du im Online-Editor auf *File* → *save* klickst, dann wirst du aufgefordert, dir einen entsprechenden Account einzurichten, um deine Lösungen abzuspeichern. Wenn du dies nicht möchtest, dann musst du mit *copy and paste* arbeiten.

Aufgabe 1 Erstelle Abbildung 2.13 mit der Funktion *rect()*.

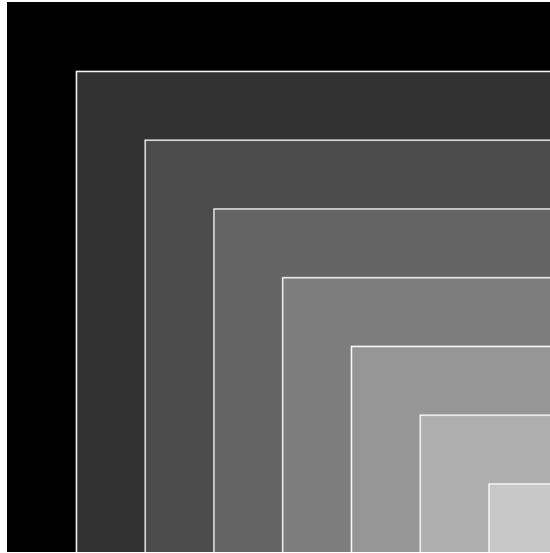


Abbildung 2.13: Graustufen

Aufgabe 2 Erstelle die Abbildungen 2.14 und 2.15 mit der Funktion *arc()*.

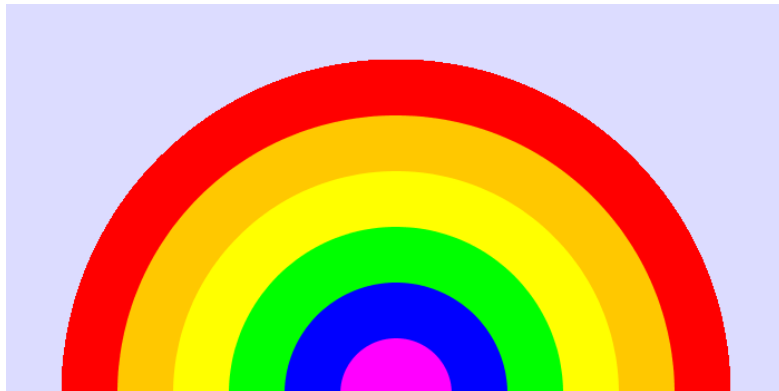


Abbildung 2.14: Regenbogen

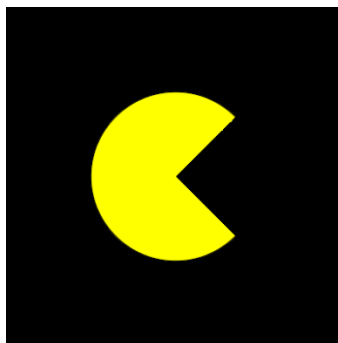


Abbildung 2.15: Pac-Man

Aufgabe 3 Erstelle die Abbildungen 2.16 und 2.17 mit den Funktionen *ellipse()*, *rect()*, *quad()* und *triangle()*.

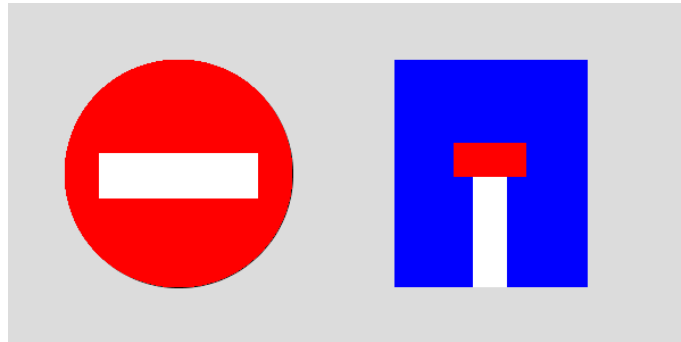


Abbildung 2.16: Verkehrsschilder 01

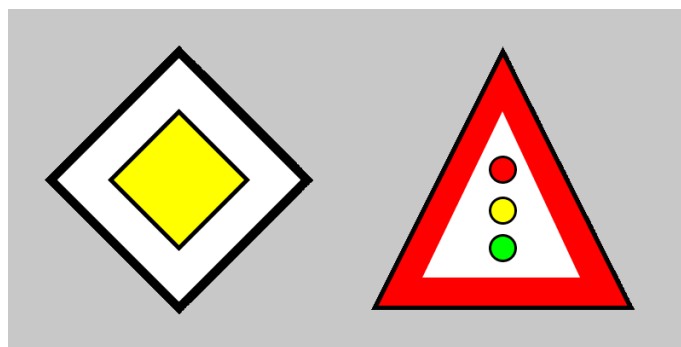


Abbildung 2.17: Verkehrsschilder 02

Aufgabe 4 Erstelle die Abbildungen 2.18 unter der Verwendung von *beginShape()* und *endShape(CLOSED)*.

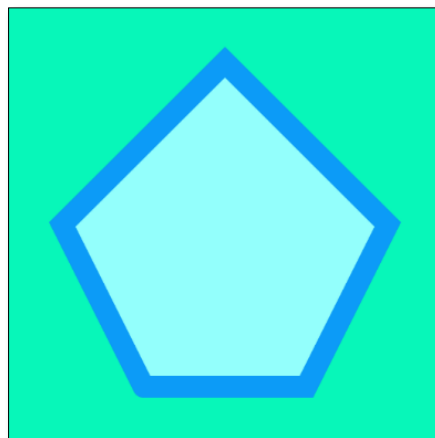


Abbildung 2.18: Fünfeck

Aufgabe 5 Zeichne mit der Funktion *curveVertex()* ein Herz.

3 Farben

3.1 Farbmischungen

Die drei lichtempfindlichen Zapfen in unserem Auge nehmen unterschiedliche Wellenlängenbereiche des Lichtes wahr, welche den Farben Rot, Grün und Blau entsprechen. Aus diesem Grund strahlen auch die Pixel eines Monitors in diesen drei Farben. Andere Farben erzeugt man durch die additive Mischung dieser drei Grundfarben. So entsteht zum Beispiel die Farbe Weiß, wenn man schreibt `fill(255, 255, 255)`. Mit den entsprechenden Zahlenwertkombinationen zwischen 0 und 255 für Rot, Grün und Blau können wir zahlreiche weitere Farben additiv generieren (siehe Abb. 3.1 und 3.2).

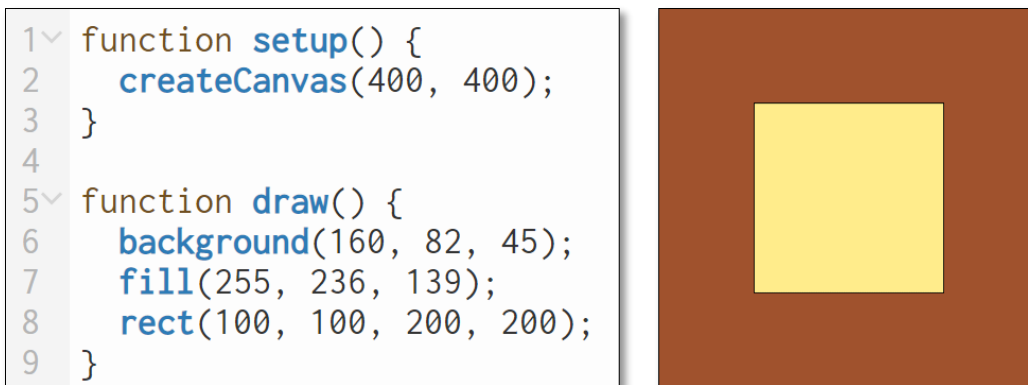


Abbildung 3.1: Farbkombination Sienna (siehe Zeile 6) mit Light Goldenrod (siehe Zeile 7)

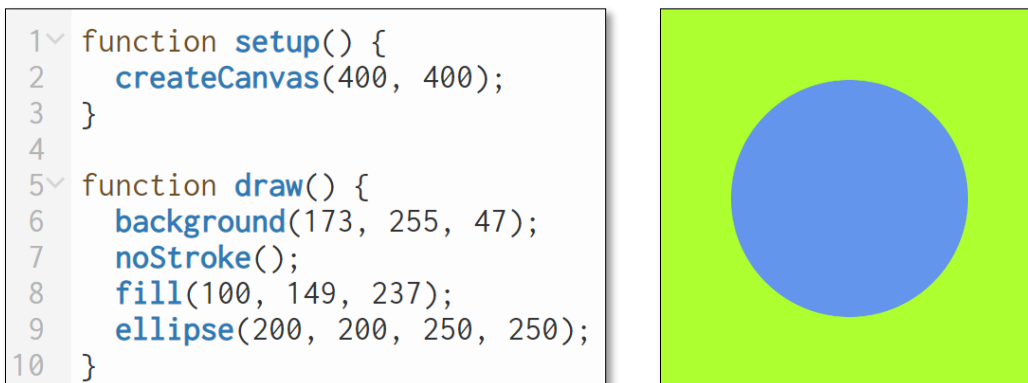


Abbildung 3.2: Farbkombination GreenYellow (siehe Zeile 6) mit CornFlowerBlue (siehe Zeile 8)

Wenn man sich die Zahlenkombinationen in den Abbildungen 3.1 und 3.2 anschaut, dann stellt man sich wahrscheinlich die Frage: „Wie lange muss ich wohl herumprobieren, bis ich meine gewünschte Kombination gefunden habe?“ Die Antwort hierauf lautet: „Anstatt lange herum zu probieren, sucht man sich besser im Internet eine Farbtabelle mit RGB-Werten und überträgt diese Zahlenkombination in seinen Sketch.“ Hilfreich bei der Farbsuche ist auch der letzte Sketch in diesem Buch (Kapitel 12).

In Büchern, die sich mit Farbmischungen beschäftigen, findet man als Beispiel für die additive und subtraktive Farbmischung häufig die folgenden beiden Abbildungen (Abb. 3.3).

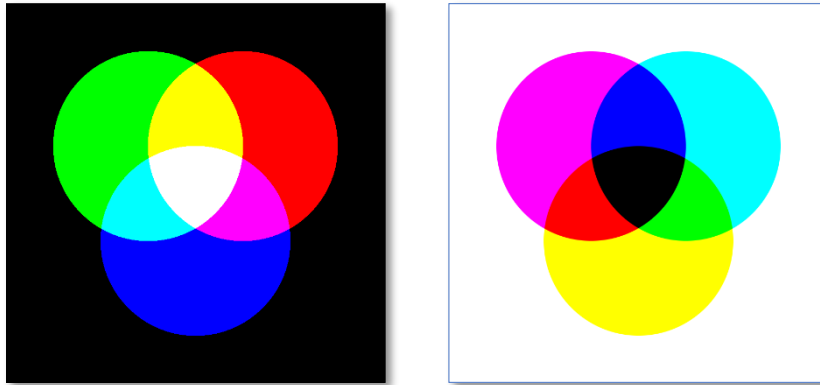


Abbildung 3.3: Additive Farbmischung (links) und subtraktive Farbmischung (rechts)

Wie erstellt man die obigen Abbildungen mit p5js? Der erste Gedanke ist, dass man für die additive Farbmischung die drei Farben Rot, Grün und Blau einfach teilweise übereinander schiebt. Dieses Vorgehen führt jedoch nicht zum gewünschten Ergebnis (siehe Abb. 3.4). Die drei Farben überlagern sich einfach. Grün überdeckt Teile von Rot und Blau überdeckt zum Teil Grün und Rot.

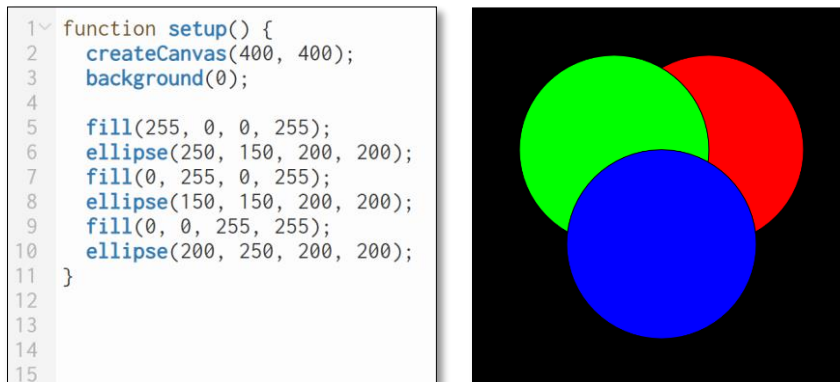


Abbildung 3.4: Misslungene additive Farbmischung

Trotz dieses Misserfolges können wir anhand des obigen Sketches etwas lernen. So sollte es einem auffallen, dass in dem Sketch *function draw()* fehlt. Trotzdem funktioniert er. Wie das? Nun, der Sketch ist im **statischen Modus** geschrieben. D.h., die einzelnen Programmzeilen werden bei *function setup()* nur einmal abgearbeitet und nicht wie bei *function draw()* immer und immer wieder. Doch das einmalige Abarbeiten genügt vollkommen, um das Bild in Abbildung 3.4 zu zeichnen. Dies hätte auch bei allen bisherigen Abbildungen funktioniert. Aus didaktischen Gründen haben wir aber am Anfang auf diesen Hinweis verzichtet.

So, nun wollen wir aber endlich einen Weg finden, wie man die beiden Bilder in Abbildung 3.3 erstellt. Ein Lösungsansatz könnte vielleicht darin bestehen, bei *fill()* eine vierte Zahl, den Alphawert, hinzuzufügen. Mit dem Alphawert kann man, wie schon in Kapitel 2.1 erwähnt, die Transparenz bzw. die Deckfähigkeit der gewählten Farbe festlegen. 255 entspricht 100% Deckfähigkeit und 0 entspricht 0% Deckfähigkeit. Aber egal, welchen jeweiligen Alphawert wir auch wählen, ein überzeugendes Bild ergibt sich leider nicht. Was tun?

Führen wir nun in unserem Online-Editor die folgenden Klicks durch: *Help & Feedback* → *Reference* → *blendMode()*. Hier sehen wir, dass p5.js eine Vielzahl von Möglichkeiten anbietet, um Farben zu mischen. Den Mischmodus, den wir für die additive Farbmischung benötigen, lautet ***blendMod(ADD)***. Für die subtraktive Farbmischung schreiben wir ***blendMode(DIFFERENCE)***. Abbildung 3.5 und 3.6

zeigen uns nun endlich die gewünschten Ergebnisse. Mit den anderen Mischmodi zu spielen ist natürlich nicht verboten ;)

```

1 function setup() {
2   createCanvas(400, 400);
3   background(0);
4
5   blendMode(ADD);
6
7   fill(255, 0, 0, 255);
8   ellipse(250, 150, 200, 200);
9   fill(0, 255, 0, 255);
10  ellipse(150, 150, 200, 200);
11  fill(0, 0, 255, 255);
12  ellipse(200, 250, 200, 200);
13 }
14
15

```

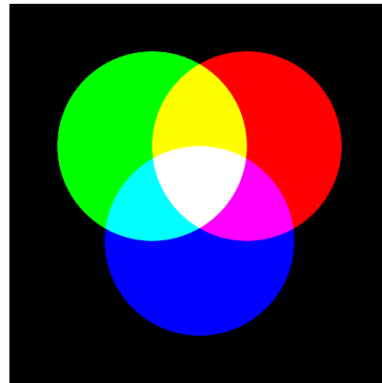


Abbildung 3.5: Additive Farbmischung

```

1 function setup() {
2   createCanvas(400, 400);
3   background(255);
4
5   blendMode(DIFFERENCE);
6
7   fill(255, 0, 0, 255);
8   ellipse(250, 150, 200, 200);
9   fill(0, 255, 0, 255);
10  ellipse(150, 150, 200, 200);
11  fill(0, 0, 255, 255);
12  ellipse(200, 250, 200, 200);
13 }
14
15

```

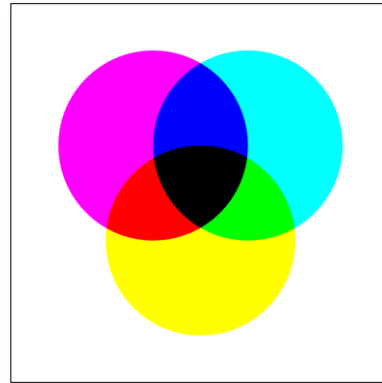
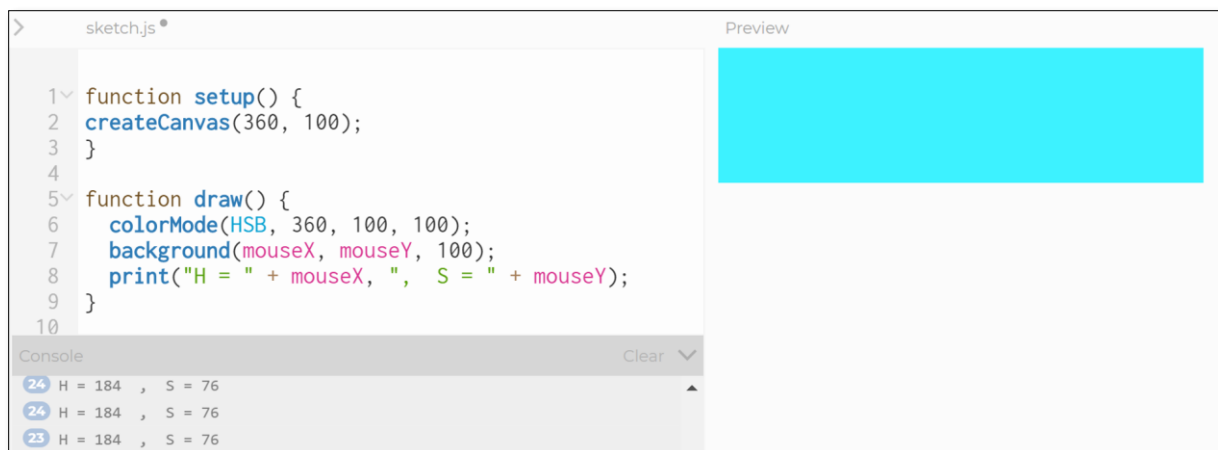


Abbildung 3.6: Subtraktive Farbmischung

3.2 HSB-Farbraum

Den Farbraum RGB (Rot, Grün, Blau) haben wir in den Kapiteln 2.1 und 3.1 kennengelernt. Er wird standardmäßig in p5.js aufgerufen. p5.js bietet aber auch die Möglichkeit im **HSB-Farbraum** zu arbeiten. H steht für **Hue**, was übersetzt **Farbton** heißt. Mit dem Hue-Wert wählt man den Farbton aus. Der Farbton wird also nicht wie im RGB-Farbraum durch Mischung dreier Zahlenwerte bestimmt, sondern durch eine einzige Zahl festgelegt. Der Wertebereich für Hue erstreckt sich von 0 bis 360, weil die unterschiedlichen Farbtöne in der Literatur meist auf einem Kreisring von 0° bis 360° aufgetragen werden. Der Buchstabe **S** im HSB Farbraum steht für **Saturation (Sättigung)**. Sein Wertebereich erstreckt sich von 0 % bis 100 %. Die Zahl 100 steht für volle Farbtintensität. Der Buchstabe **B** im HSB Farbraum steht für **Brightness (Helligkeit)**. Dieser Wertebereich erstreckt sich auch von 0 % bis 100 %. Die Zahl 100 steht für volle Helligkeit.

Nun steht man wieder vor dem Problem, welche Zahl im HSB-Farbraum welcher Farbe entspricht? Für den RGB-Farbraum haben wir uns im Internet entsprechende Tabellen angeschaut. Für den HSB-Farbraum wollen wir dies nun mit einem eigenen Sketch lösen (siehe Abb. 3.7).



```
sketch.js
1 function setup() {
2   createCanvas(360, 100);
3 }
4
5 function draw() {
6   colorMode(HSB, 360, 100, 100);
7   background(mouseX, mouseY, 100);
8   print("H = " + mouseX, ", S = " + mouseY);
9 }
10
```

Preview

Console

```
24 H = 184 , S = 76
24 H = 184 , S = 76
23 H = 184 , S = 76
```

Abbildung 3.7: Sketch 01 zur Bestimmung der Zahlenwerte im HSB-Farbraum

Die einzelnen Zeilen des obigen Sketches sollen nun erläutert werden. Mit `createCanvas(360, 100)` erstellen wir uns bei `function setup()` eine Leinwand, die 360 x 100 Pixel groß ist. Da p5.js standardmäßig im RGB-Farbraum arbeitet, teilen wir p5.js bei `function draw()` mit, dass es ab dieser Zeile im HSB-Farbraum arbeiten soll. Dies gelingt mit `colorMode(HSB, 360, 100, 100)`. Die Buchstaben HSB in der runden Klammer legen den Farbraum fest. Die erste Zahl nach HSB steht für H (Farbton), die zweite für S (Sättigung) und die dritte für B (Helligkeit). Man kann für die drei Zahlen 360, 100, 100 auch andere Werte eingeben, um so einen anderen Wertebereich zwischen 0 und der eingesetzten Zahl festzulegen, doch wir halten uns hier an die übliche Schreibweise.

Mit `background(mouseX, mouseY, 100)` legen wir die Farbe der 360 x 100 Pixel großen Leinwand fest. Wenn wir nun den Mauszeiger über die Leinwand bewegen, dann setzt p5.js automatisch für `mouseX` den x-Wert und für `mouseY` den y-Wert des Mauszeigers ein. D. h., wenn wir den Mauszeiger über die Leinwand bewegen, ändert sich die Farbe der Leinwand, da der x-Wert dem Wert von H (Farbton) und der y-Wert dem Wert von S (Sättigung) im HSB-Raum entspricht. Den Wert von B (Helligkeit) setzen wir auf den Maximalwert 100.

Mit der schon bekannten Funktion `print()` können wir uns Texte und Werte in die Konsole schreiben lassen. Mit `print(„H = “+mouseX, „, S = “+mouseY)` schreibt p5.js den Text zwischen den Klammern in die Konsole und mit `+mouseX` und `+ mouseY` fügt p5.js hinter dem jeweiligen Text den x-Wert, bzw. den y-Wert des Mauszeigers ein.

Damit haben wir unser Ziel erreicht. Wenn wir die H- und S-Werte für eine Wunschfarbe suchen, dann bewegen wir den Mauszeiger solange über die Leinwand, bis sie die gewünschte Farbe annimmt und lesen dann in der Konsole die zugehörigen H- und S-Werte ab. Der Trick hierbei ist, dass wir für die Breite der Leinwand 360 Pixel und für die Höhe 100 Pixel gewählt haben und wir darauf achten, dass sich der Mauszeiger auch auf der Leinwand befindet.

Wie es noch eleganter geht, zeigt uns Abbildung 3.8. Hier lassen wir uns den H- und S-Wert direkt auf der Leinwand anzeigen.

```

1 function setup() {
2   createCanvas(360, 100);
3 }
4
5 function draw() {
6   colorMode(HSB, 360, 100, 100);
7   background(mouseX, mouseY, 100);
8   fill(0, 0, 100);
9   textSize(20);
10  text("H = " +mouseX, 80, 30);
11  text("S = " +mouseY, 180, 30);
12 }
13

```

Abbildung 3.8: Sketch 02 zur Bestimmung der Zahlenwerte im HSB-Farbraum

Neu in diesem Sketch sind die letzten vier Zeilen bei *function draw()*. Mit *fill(0, 0, 100)* legen wir die Farbe des auf der Leinwand erscheinenden Textes fest. Diese Farbe sollte keine der Farben sein, die die Leinwand bei unserer Mausbewegung in x-Richtung annimmt, da wir dann bei einer bestimmten Farbe die Zahlen nicht mehr lesen können. Also entscheiden wir uns für die Farbe Weiß, Schwarz oder Grau. Da wir im HSB-Farbraum arbeiten, müssen wir kurz nachdenken, welche drei Werte wir in die runde Klammer für die genannten drei Farben bei *fill()* eintragen müssen. Schwarz erhalten wir, wenn wir den Wert für B (Helligkeit) auf 0 setzen. Egal welche Werte dann H und S besitzen, für B = 0 sehen wir die Farbe Schwarz. Weiß erhalten wir unabhängig vom H-Wert, wenn wir S = 0 und B = 100 setzen. Um Grautöne zu erhalten, wählen wir bei festem S = 0 für B Werte zwischen 0 und 100, z.B. den Wert 50.

Mit *textSize()* legen wir die Größe der Schrift fest. Die Zeile *text(„H = “ +mouseX, 80, 30)* sorgt dafür, dass auf der Leinwand *H = x-Wert des Mauszeigers* geschrieben wird (In Abbildung 3.8: H = 40). Die Zahlen 80 und 30 in der runden Klammer geben annähernd den linken unteren Startpunkt (x- und y-Wert) des Buchstabens H auf der Leinwand an. Für die Zeile *text(„S = “ +mouseY, 180, 30)* gilt entsprechendes.

Weiterhin sei noch erwähnt, dass es auch für den HSB-Farbraum einen Alpha-Wert gibt, mit dem man die Deckfähigkeit einer Farbe bestimmen kann. Dies ist der vierte Wert in der runden Klammer bei *colorMode()*. Auch hier kann der Wertebereich beliebig festgelegt werden. In Zeile 6 von Abbildung 3.9 wurde mit der Zahl 100 ein Wertebereich für den Alpha-Wert von 0 bis 100 gewählt. Als Farbe für das Rechteck in Abbildung 3.9 wurde mit *fill(0, 0, 100, 50)* die Farbe Weiß mit einer Deckkraft von 50 gewählt.

```

1 function setup() {
2   createCanvas(360, 100);
3 }
4
5 function draw() {
6   colorMode(HSB, 360, 100, 100, 100);
7   background(40, 80, 100);
8   fill(0, 0, 100, 50);
9   rect(80,20, 200, 60);
10 }
11

```

Abbildung 3.9: Auch im HSB-Farbraum gibt es einen Alpha-Wert zur Einstellung der Deckfähigkeit einer Farbe

3.3 Zusammenfassung

Fassen wir zusammen, was wir in Kapitel 3 gelernt haben, bzw. gelernt haben sollten.

Referenz	Im Online-Editor finden wir unter <i>Help & Feedback</i> → <i>Reference</i> Beispiele und Erläuterungen zu zahlreichen Funktionen.
blendMod(ADD)	Mit <i>blendMode(ADD)</i> können wir Farben additiv mischen.
blendMode(DIFFERENCE)	Mit <i>blendMode(DIFFERENCE)</i> können wir Farben subtraktiv mischen.
colorMode()	Mit <i>colorMode()</i> kann man den Farbraum (RGB oder HSB) und den jeweiligen Wertebereich festlegen.
RGB-Farbraum	p5.js arbeitet standardmäßig im RGB-Farbraum. Im RGB-Farbraum werden Farben durch Zahlenkombinationen der Farben Rot, Grün und Blau generiert. So erhält man mit <i>fill(255, 255, 255)</i> zum Beispiel die Farbe Weiß. Mittels einer vierten Zahl, dem Alpha-Wert, kann man die Transparenz, bzw. Deckkraft einer Farbe bestimmen. 0 bedeutet volle Transparenz und 100 volle Deckkraft.
HSB-Farbraum	Beim HSB-Farbraum steht H für Hue (Farbton) . Der Zahlenbereich von Hue erstreckt sich üblicherweise von 0 bis 360. Mit der Zahl für Hue wählt man den Farbton aus. S steht für Saturation (Sättigung) und B für Brightness (Helligkeit) . Der Zahlenbereich für die Farbsättigung und die Helligkeit erstreckt sich jeweils von 0 bis 100. Auch im HSB-Farbraum kann man mit einer vierten Zahl, dem Alpha-Wert, die Transparenz einer Farbe einstellen. Um vom RGB-Farbraum in den HSB-Farbraum zu wechseln oder vom HSB-Farbraum in den RGB-Farbraum zu wechseln, schreibt man colorMode(RGB, 255, 255, 255, 100) bzw. colorMode(HSB, 360, 100, 100, 100) .
mouseX	<i>mouseX</i> ist eine Systemvariable. Dort, wo <i>mouseX</i> in einer Sketchzeile steht, setzt p5.js automatisch den x-Wert des Mauszeigers ein.
mouseY	<i>mouseY</i> ist eine Systemvariable. Dort, wo <i>mouseY</i> in einer Sketchzeile steht, setzt p5.js automatisch den y-Wert des Mauszeigers ein.
text()	Mit der Funktion <i>text()</i> kann man Text und den Wert von Variablen auf die Leinwand schreiben. Der Text muss hierbei in Anführungszeichen gesetzt werden und dem Namen der Variablen wird ein + vorangestellt. Weiterhin muss in der Klammer noch die x- und y-Koordinate des Textanfangs eingegeben werden. Beispiel: <i>text(„H = “ +mouseX, 80, 30)</i> Die Farbe des Textes bestimmt man mit <i>fill()</i> .
textSize()	Mit der <i>textSize()</i> legt man die Größe der Schrift auf der Leinwand fest.

3.4 Übungen

Aufgabe 1 Schreibe einen Sketch, der die Abbildung 3.10 erzeugt (siehe unten).



Abbildung 3.10: Bild zur Aufgabe 1

Aufgabe 2 Mit welchem Mischmodus (*blendMode()*) kannst du die Abbildung 3.11 erzeugen? Schau in der Referenz von p5.js und experimentiere mit unterschiedlichen Mischmodi.

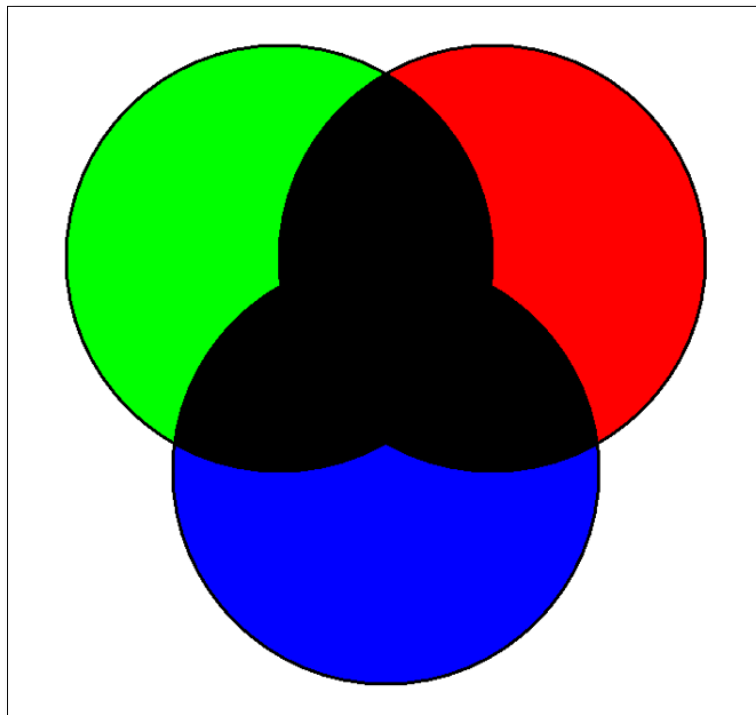


Abbildung 3.11: Bild zur Aufgabe 2

Aufgabe 3 Erstelle die Abbildung 3.12 einmal unter Verwendung des RGB-Farbraumes und einmal unter Verwendung des HSB-Farbraumes.

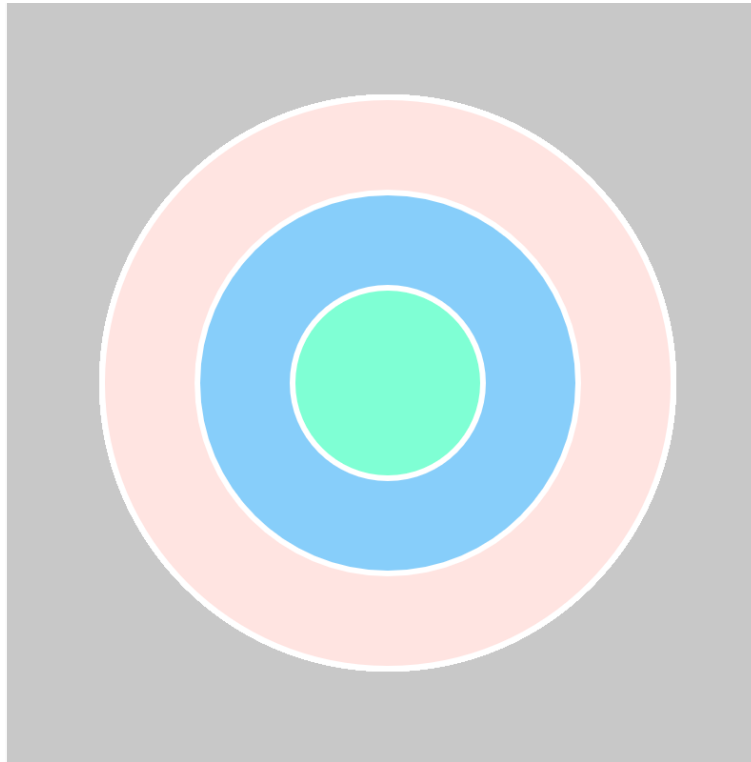


Abbildung 3.12: Bild zur Aufgabe 3

- Aufgabe 4 Mittels des Sketches von Abbildung 3.8 kann man Farbtöne im HSB-Farbraum mittels *mouseX()* und *mouseY()* auswählen. Ändere diesen Sketch so um, dass man mit *mouseX()* und *mouseY()* auch Farbtöne im RGB-Farbraum auswählen kann. Setze dazu den Wert von Rot auf konstant 255. Wenn du alles richtig gemacht hast, dann solltest du durch die Bewegung des Mauszeigers die Farben Rot, Violett, Orange, Gelb und Weiß erzeugen können.
- Aufgabe 5 Informiere dich im Internet zu dem Thema Farbharmonie und schreibe Sketche, mit harmonischen Farbkombinationen.
- Aufgabe 6 Informiere dich im Internet zu dem Thema Farbpsychologie und schreibe Sketche, die Farbkombinationen zu den folgenden Themen generieren: Lebensfreude, Trauer, Wärme, Kälte.

4 Offline-Editoren

Die Sketche, die man im **Online-Editor** von p5.js erstellt hat, kann man, wie schon erwähnt, nach der Einrichtung eines Accounts auf der p5.js-Homepage oder mittels „copy and paste“ auf dem eigenen Computer abspeichern. Möchte man seine Sketche lieber offline bearbeiten und abspeichern, dann muss man sich im Download-Bereich der p5.js-Homepage unter

<https://p5js.org/download/>

ein kleines Software-Paket downloaden. Am besten lädt man sich **p5.js complete** herunter, auch wenn wir einige Paketbestandteile erst später benötigen. Abbildung 4.1 zeigt den Inhalt des Software-Paketes nach dem Entpacken. Für das Erste benötigen wir nur den Ordner *empty-example* und die Dateien *p5.min.js*. Das Addon *p5.sound.js* benötigen wir zu einem späteren Zeitpunkt (Kapitel 12).

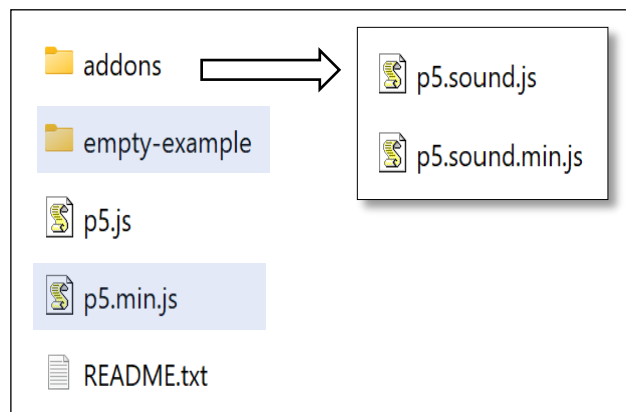


Abbildung 4.1: Inhalt des Software-Paketes p5.js complete

Im Ordner *empty-example* findet man die Datei *sketch.js*. Diese Datei kann man mit einem einfachen **Text-Editor** öffnen und bearbeiten (Abb. 4.2). Komfortabler ist es jedoch, wenn man einen guten **Code-Editor** verwendet. Für mehrere Sketche in diesem Buch wurde der kostenlose Code-Editor *Atom* verwendet. Seit dem 15. Dezember 2022 wird *Atom* jedoch nicht mehr weiterentwickelt. Aber im Internet findet man aktuelle Code-Editor, die man kostenlos downloaden kann.

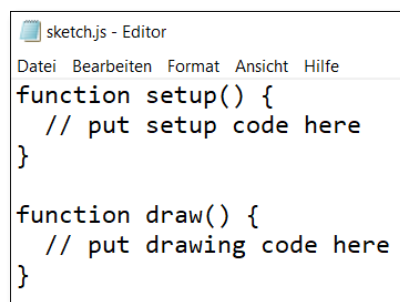


Abbildung 4.2: Die Datei *sketch.js* wurde mit einem einfachen Text-Editor geöffnet

Den Ordner *empty-example* und die Datei *p5.min.js* kopieren wir nach dem Download in die Zwischenablage und fügen beide in einen neuen Ordner ein, der den Namen unseres neu zu erstellenden Sketches trägt. Mittels unseres Text-Editors oder Code-Editors öffnen wir nun den Ordner *empty-example*. Danach klicken wir auf *sketch.js* und schreiben unseren Sketch (siehe Abb. 4.2 u. 4.3).

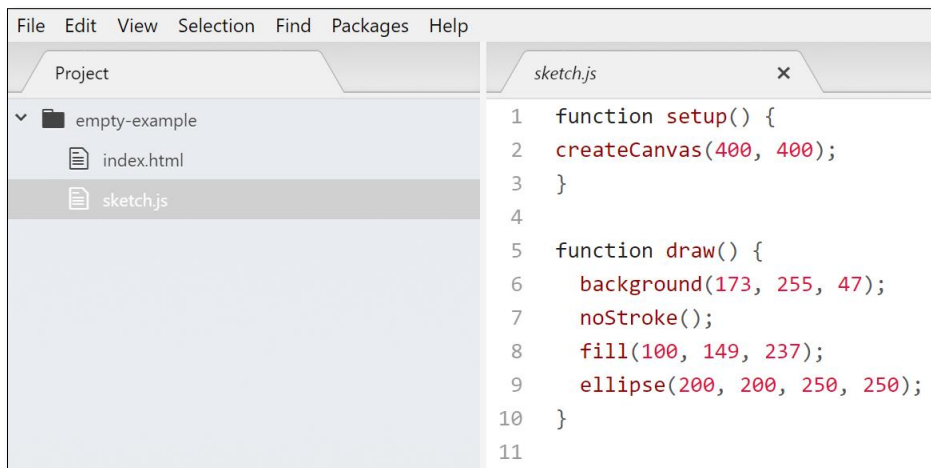


Abbildung 4.3: Die Datei sketch.js im Ordner empty-example wurde im Code-Editor Atom geöffnet und bearbeitet

Klicken wir im Text-Editor oder Code-Editor auf *index.html* (siehe Abb. 4.3), so finden wir in Zeile 6 der sich öffnenden HTML-Datei den Dateinamen *p5.min.js* (siehe Abb. 4.4). Die Datei *p5.min.js*, die sich in unserem Projektordner befinden muss, übersetzt unseren Sketch von Abbildung 4.3 in für Internet-Browser lesbaren Code.

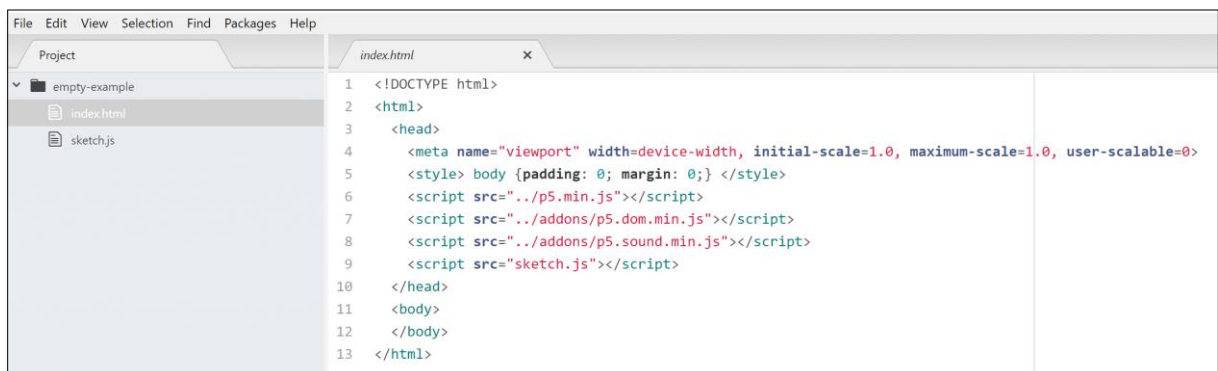


Abbildung 4.4: Im Code-Editor Atom geöffnete HTML-Datei

Nachdem wir unseren Sketch geschrieben und abgespeichert haben, öffnen wir in unserem Projekt-Ordner den Ordner *empty-example* und klicken hier auf *index.html*. Wenn wir alles richtig gemacht haben, dann wird unser Sketch in Firefox, Google Chrome, ... geöffnet.

Ob man lieber mit einem Online-Editor oder mit einem Offline-Editor arbeitet, muss jeder selbst entscheiden.

5 Gestalten mit Variablen, for-Schleifen, if- und else-Anweisungen

5.1 Variablen

Die Verwendung von variablen Größen gestattet es, einen Sketch kürzer, flexibler und lebendiger zu gestalten. In der Mathematik ist zum Beispiel die Variable x ein Platzhalter für eine Zahl. In der Informatik benutzt man Variablen, um Informationen abzuspeichern, die dann im späteren Programmverlauf wieder aufgerufen werden können. In Kapitel 3 haben wir mit `mouseX` und `mouseY` schon zwei Variablen kennengelernt. Da diese beiden Variablen von p5.js fest vorgegeben sind, nennt man sie **Systemvariablen**. `mouseX` ist ein Platzhalter für die x-Koordinate des Mauszeigers im Browserfenster und `mouseY` ist ein Platzhalter für die y-Koordinate des Mauszeigers. Bei jedem Durchlauf von `function draw()` werden für `mouseX` und `mouseY` die aktuellen Koordinatenwerte des Mauszeigers eingesetzt. Bewegen wir unsere Maus über die Leinwand, so können wir damit zum Beispiel eine Figur bewegen. Abbildung 5.1 zeigt dies am Beispiel eines grün gefüllten Kreises.

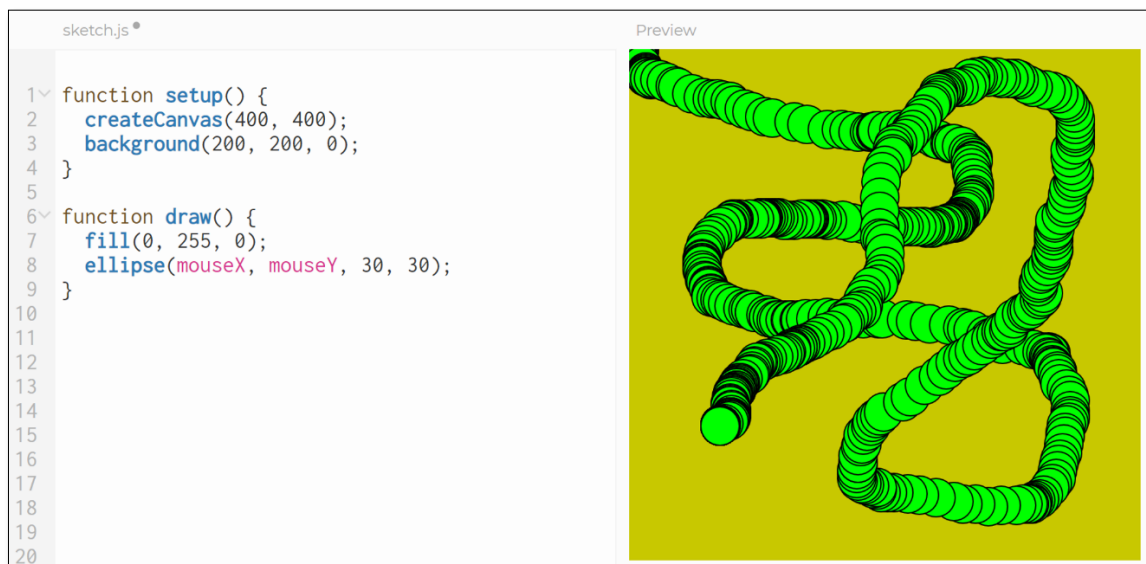


Abbildung 5.1: Mittels Mausbewegung gezeichnete grüne Schlange

Um die in Abbildung 5.1 dargestellte Schlange zu erhalten, haben wir die Funktion `background()` nicht bei `function draw()`, sondern bei `function setup()` eingefügt. Hätten wir `background()` bei `function draw()` eingefügt, dann wären die vorherigen Kreise stets vom Hintergrund übermalt worden und wir würden nur einen, sich auf der Leinwand bewegendem, Kreis sehen.

Abbildung 5.2 zeigt, dass wir unsere Schlange auch farbig gestalten können. Dazu nutzen wir den HSB-Farbraum. Anders als üblich passen wir den Bereich für H und S der Leinwandgröße an. Wir setzen den Maximalwert von H auf 400 statt auf 360 und den Maximalwert von S auch auf 400 statt auf 100. Von `mouseX` und `mouseY` haben wir jeweils den halben Durchmesser des Kreises abgezogen, da sonst beim Start des Sketches in der linken oberen Ecke ein Viertelkreis erschienen wäre. Dies rührt daher, dass beim Drücken der Starttaste im Online-Editor der Mauszeiger sich noch nicht auf der Leinwand befindet und so für den x- und y-Wert des Mauszeigers und damit für den Mittelpunkt des Kreises jeweils die Zahl Null eingesetzt wird.

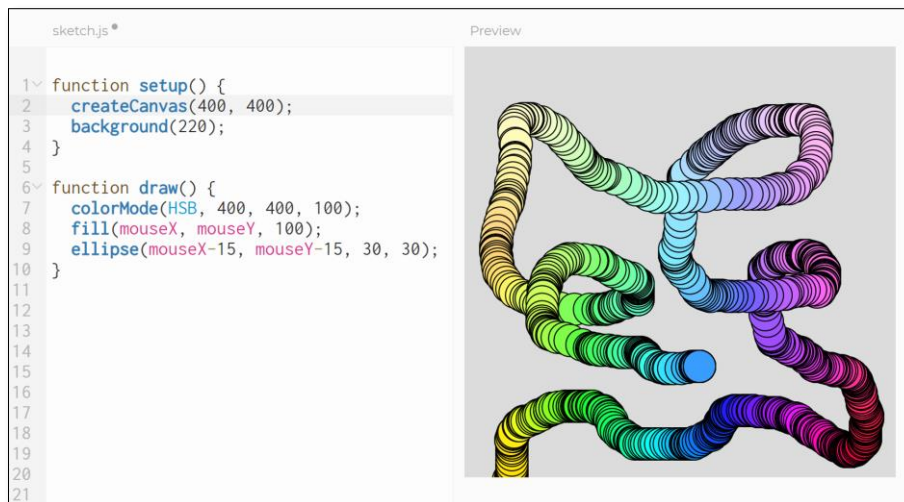


Abbildung 5.2: Bunte Schlange im HSB-Farbraum

Für unseren nächsten Sketch wollen wir mithilfe des Ausdrucks **let** eine eigene Variable erstellen. In die erste Zeile des Sketches in Abbildung 5.3 schreiben wir **let x = 0**. Damit haben wir der Variablen einen Namen (hier x) gegeben. In der Sprache der Informatiker sagt man, man habe eine Variable **deklariert**. Danach haben wir die Variable x **initialisiert**, d.h. wir haben ihr einen konkreten Wert, hier die Zahl Null, zugeordnet. Die Variablen, die vor und nicht in den Codeblöcken von *function setup()* und *function draw()* stehen, nennt man **globale Variablen**. Die Variablen, die innerhalb der Codeblöcke dieser Funktionen deklariert und initialisiert werden, nennt man **lokale Variablen**. Die lokalen Variablen mit der Bezeichnung **let** können nur von Funktionen innerhalb ihrer Codeblöcke aufgerufen werden. Globale Variablen dagegen können von allen Funktionen aus aufgerufen werden.

Schauen wir uns nun den Sketch von Abbildung 5.3 genauer an. Die globale Variable x wird in Zeile 1 deklariert und initialisiert. Aufgerufen wird sie innerhalb von *function draw()*. Beim ersten Durchlauf von *function draw()* wird für x der Wert 0 eingesetzt. D.h., der linke obere Eckpunkt des ersten schwarzen Rechteckes von Abbildung 5.3 besitzt den x-Wert Null. Nachdem das erste schwarze Rechteck gezeichnet ist, wird in Zeile 12 der x-Wert um den Pixelwert 40 erhöht. Beim nächsten Durchlauf von *function draw()* wird bei *rect()* für x der Wert 40 eingesetzt. Beim nächsten Durchlauf 80, dann 120, usw. Auf diese Weise erhalten wir in Abbildung 5.3 ein Zebrastrifen-Muster, ohne zehnmals den Code für ein Rechteck mit anderen x-Werten schreiben zu müssen.

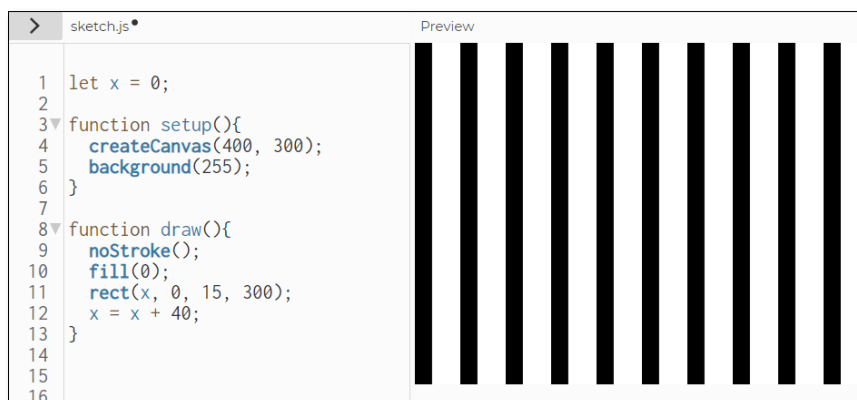


Abbildung 5.3: Zebrastrifen

Unsere Variable `x` können wir auch dazu benutzen, ein farbiges Streifenmuster wie in Abbildung 5.4 zu zeichnen. Der Trick hierbei ist, dass wir im HSB-Farbraum arbeiten und in der Zeile 11 den Farbton (H) bei `fill(x, 100, 100)` mittels des `x`-Wertes festlegen. Da `x` sich von 0 bis 360 in 30er-Schritten bei jedem Durchlauf von `function draw()` erhöht, erhalten wir bei einem 360 Pixel breiten Fenster 12 farbige Streifen, die uns an die Spektralfarben eines Regenbogens erinnern.

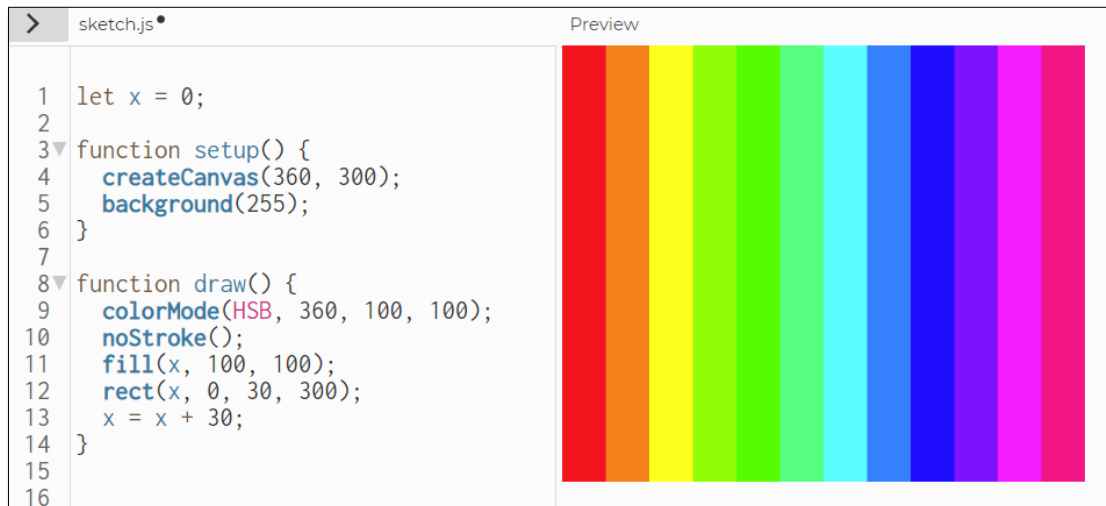


Abbildung 5.4: Streifenmuster in den Spektralfarben

5.2 for-Schleife

Wenn wir in dem Sketch von Abbildung 5.4 die Breite der Rechtecke auf nur einen Pixel, also auf eine Linie festlegen und `x` in Einerschritten erhöhen, dann erhalten wir das Bild eines kontinuierlichen Spektrums (Abb. 5.5). Dies sieht zwar sehr schön aus, doch das Spektrum baut sich nur relativ langsam im Internetbrowser auf. Lässt sich der Aufbau des Bildes nicht irgendwie beschleunigen?

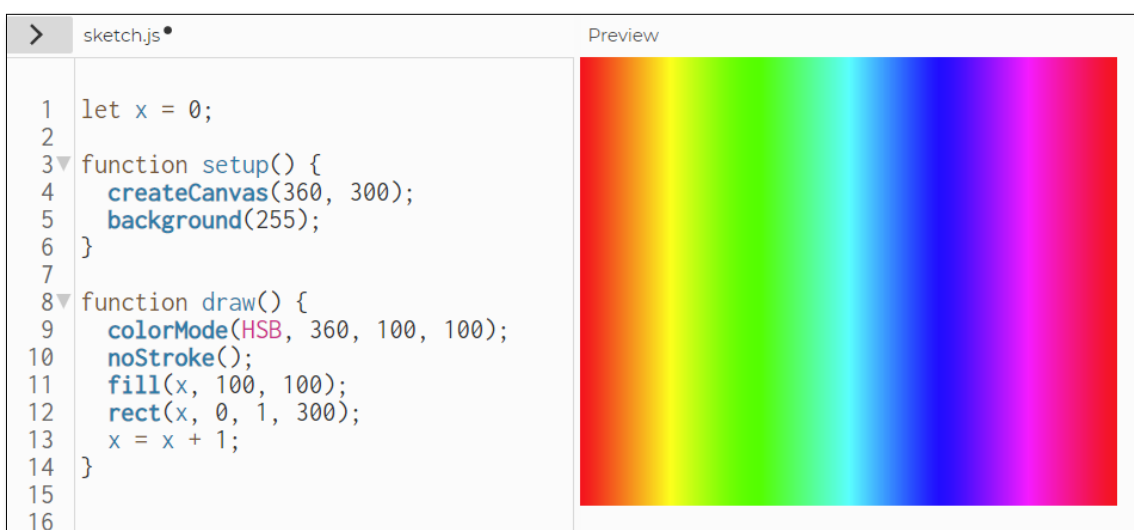


Abbildung 5.5: Spektrum

Ein schnellerer Bildaufbau gelingt mit einer **for-Schleife**. In Zeile 9 von Abbildung 5.6 finden wir eine solche for-Schleife **for(let x = 0; x <= 360; x = x + 1)**. Innerhalb der runden Klammern wird zuerst die Variable x deklariert und initialisiert. Nach dem Semikolon (;) wird mit x <= 360 festgelegt, dass x nur Werte kleiner gleich 360 annehmen darf. Nach dem zweiten Semikolon steht x = x + 1. D.h., bei jedem Durchlaufen der for-Schleife soll x um den Wert 1 zunehmen.

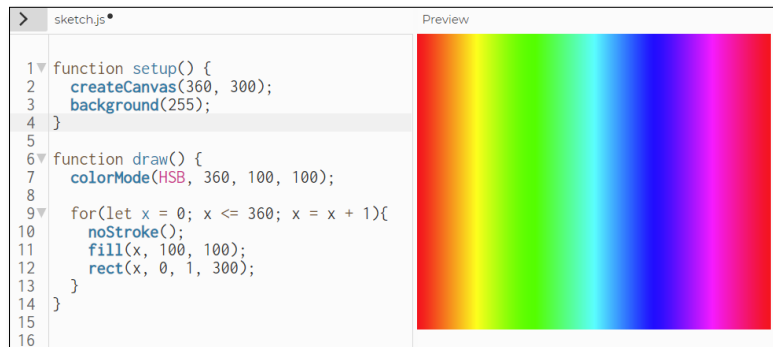


Abbildung 5.6: Dieses Spektrum wurde mittels einer for-Schleife erzeugt

Da die Variable x in einer for-Schleife steht, gilt sie auch nur innerhalb dieser for-Schleife. In diesem Fall ist x also eine **lokale Variable**. Sie wirkt somit nur in dem Bereich zwischen den beiden geschweiften Klammern, die zu der for-Schleife gehören (siehe Abb. 5.6).

Warum gelingt der Aufbau des Spektrums mit einer for-Schleife so viel schneller als ohne for-Schleife? Bei dem Sketch von Abbildung 5.5 wird bei jedem Durchlauf von *function draw()* eine Linie gezeichnet und der Wert von x um 1 erhöht. Beim nächsten Durchlauf von *function draw()* wird wieder eine Linie gezeichnet und der Wert von x wieder um 1 erhöht. Bis alle Linien gezeichnet sind vergeht so einiges an Zeit. Bei der Lösung mit der for-Schleife in dem Sketch von Abbildung 5.6 werden innerhalb der for-Schleife zuerst alle x-Werte berechnet und erst dann zeichnet die Funktion *function draw()* das Spektrum. Dies geht natürlich viel schneller, als wenn nacheinander 360 Linien gezeichnet werden müssen. Um den Wert von x um 1 zu erhöhen, haben wir bisher x = x + 1 geschrieben. Eine Kurzschreibweise für x = x + 1 ist **x++** (siehe Zeile 8 in Abbildung 5.7).

Vielfältige gestalterische Möglichkeiten ergeben sich, wenn wir eine for-Schleife in Verbindung mit der Funktion **random()** verwenden. Die Funktion *random()* stellt dem Programmierer einen Zufalls-generator zur Verfügung. Schreibt man *random(200)*, so generiert die Funktion Zahlen von 0,00... bis einschließlich 199,99.... Schreibt man *random(50, 100)*, so generiert die Funktion Zahlen von 50,00... bis einschließlich 99,99.... Schauen wir uns als Beispiel den Sketch von Abbildung 5.7 an.

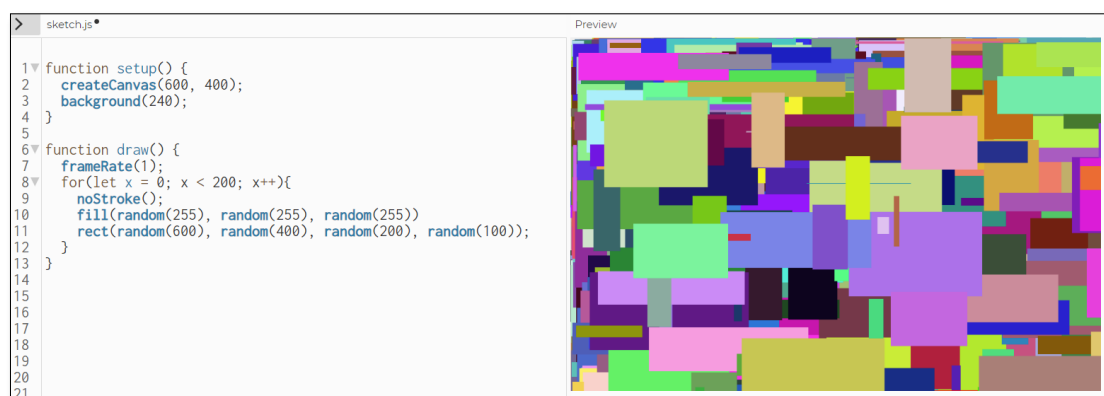


Abbildung 5.7: Mit der Funktion random() erzeugte farbige Rechtecke

Mittels der for-Schleife im Sketch von Abbildung 5.7 werden unter Verwendung des Zufallsgenerators *random()* die Werte für 200 Rechtecke von unterschiedlicher Größe und Farbe berechnet. Anschließend zeichnet die Funktion *function draw()* diese Rechtecke. Danach werden in der for-Schleife mittels *random()* erneut 200 Rechtecke generiert, die von *function draw()* über die vorherigen Rechtecke gezeichnet werden, da sich die Funktion *background()* bei *function setup()* und nicht bei *function draw()* befindet. Dieser Vorgang würde sich normalerweise ca. 60mal in der Sekunde wiederholen und wir würden ein munteres „Geflimmere“ auf dem Bildschirm sehen. Damit wir uns die jeweils entstehenden Bilder aber wenigstens einen Sekunde lang anschauen können, haben wir in dem obigen Sketch die Funktion ***frameRate()*** eingefügt (Zeile 7). Mit *frameRate()* kann man die Bildwiederholungsrate vorgeben. *frameRate(1)* bedeutet ein neues Bild pro Sekunde. *frameRate(0.1)* bedeutet ein neues Bild alle 10 Sekunden.

Wenn wir den Sketch von Abbildung 5.7 nur geringfügig ändern, erhalten wir ein gänzlich anderes Bild (siehe Abb. 5.8). Da die Punkte hier kleiner sind als die bei 5.7 erzeugten Rechtecke, haben wir die for-Schleife 2000 Punkte berechnen lassen, um die Leinwand mit Punkten zu füllen.



Abbildung 5.8: Konfetti im RGB-Farbraum

Wieder eine kleine Änderung und wir erhalten Abbildung 5.9. Dieses Mal soll aber nicht sofort die ganze Leinwand mit Strichen bedeckt sein. Deshalb haben wir in Zeile 8 als Obergrenze $x < 10$ geschrieben. So werden pro Sekunde nur 10 Striche gezeichnet und wir können zuschauen, wie das Bild sich langsam aufbaut.

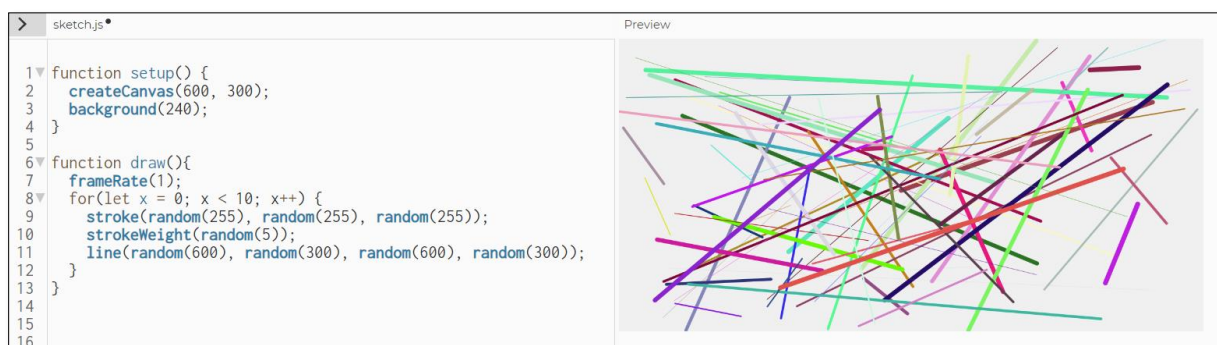


Abbildung 5.9: Bunte Striche

Lassen wir den Zufallsgenerator *random()* die Farben im HSB-Farbraum statt im RGB-Farbraum auswählen, dann erscheinen die Konfettis von Abbildung 5.8 deutlich farbenfroher (siehe Abb. 5.10). Voraussetzung hierfür ist, dass wir für die Sättigung und die Helligkeit den Wert 100 wählen. Neu in diesem Sketch ist die Funktion ***textAlign(CENTER)*** in Zeile 18. Mit ihr kann man die Textmitte auf einen gewünschten x-Wert (hier 300) setzen.

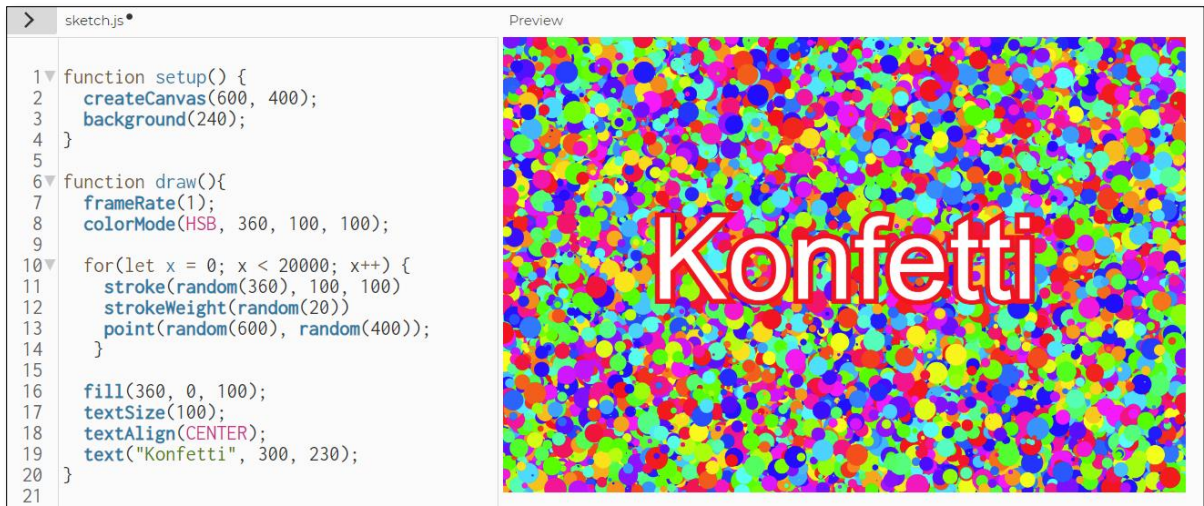


Abbildung 5.10: Konfetti im HSB-Farbraum

Wie in den obigen Abbildungen gezeigt, besitzen wir mit einer for-Schleife und der Funktion *random()* mächtige Werkzeuge zur generativen Bildgestaltung. Unsere kreativen Übungen müssen sich aber nicht auf eine einzelne for-Schleife beschränken. Wie wir im Sketch von Abbildung 5.11 sehen, wurde hier eine zweite for-Schleife in die erste integriert. Diese beiden Schleifen arbeiten zusammen und berechnen die Werte, die die Funktion *function draw()* braucht, um den Farbkasten von Abbildung 5.11 zu zeichnen. Zuerst setzt die innere for-schleife für die lokale Variable „spalten“ die Zahlenwerte 0, 1, 2, 3, 4, 5 ein und lässt die Funktion *random()* die Zufallswerte für die Farben der Farbtöpfe in der obersten Zeile des Farbkastens erzeugen. Danach erhöht die äußere for-Schleife den Wert der lokalen Variablen „zeilen“ um den Wert 1. Anschließend erzeugt die innere Schleife die Werte für die zweite Zeile der Farbtöpfe. Danach erhöht die äußere for-Schleife den Wert der lokalen Variablen „zeilen“ wieder um den Wert 1. Dies wird solange fortgesetzt, bis die Variable „zeilen“ den Wert 3 erreicht hat. Danach zeichnet die Funktion *function draw()* den Farbkasten. Nach einer Sekunde wird mit neu berechneten Werten ein neuer Farbkasten gezeichnet. Weiterhin entnehmen wir der Abbildung 5.11, dass die innere for-Schleife auf die globale Variable *d* zugreift. Mit *d* bestimmen wir den Durchmesser der Farbtöpfe.

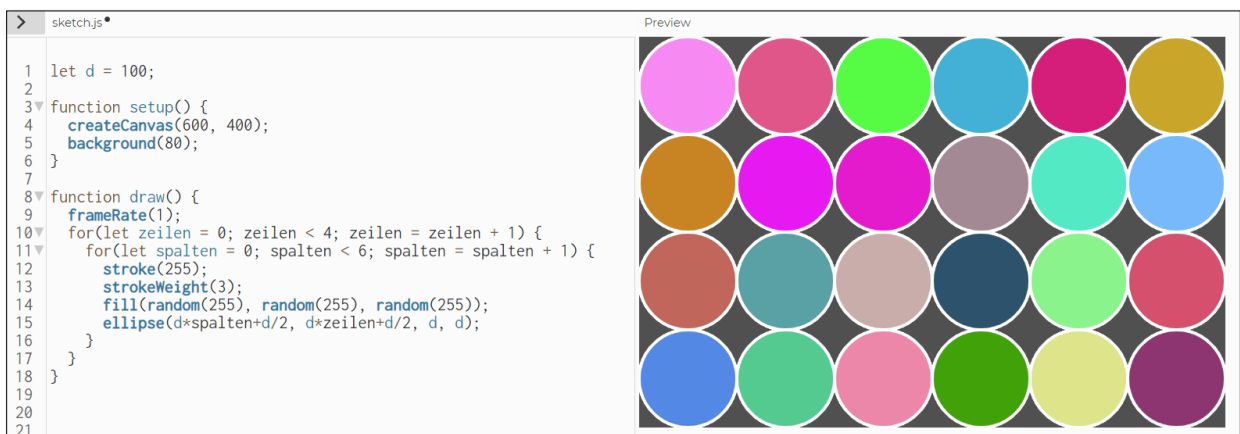


Abbildung 5.11: Mit zwei verschachtelten for-Schleifen generierter Farbkasten

Der Sketch von Abbildung 5.11 liefert zwar ein schönes Ergebnis, doch er ist wenig flexibel. Er passt sich nicht automatisch an, wenn wir mehr oder weniger Farbtöpfe oder eine größere oder kleinere Leinwand benötigen. Wie dies gelingt, zeigt uns der Sketch von Abbildung 5.12.

```
1 let skalierung = 100; // Größe der quadratischen Zellen
2 let spalten;
3 let zeilen;
4
5 function setup() {
6   createCanvas(600, 400);
7   background(80);
8   spalten = width/skalierung; // Anzahl der Spalten wird berechnet
9   zeilen = height/skalierung; // Anzahl der Zeilen wird berechnet
10 }
11
12 function draw() {
13   frameRate(1);
14   for(let y = 0; y < zeilen; y++) {
15     for(let x = 0; x < spalten; x++) {
16       stroke(255);
17       strokeWeight(3);
18       fill(random(255), random(255), random(255));
19       ellipse(x*skalierung+skalierung/2, y*skalierung+skalierung/2, skalierung-skalierung/10, skalierung-skalierung/10);
20     }
21   }
22 }
23
```

Abbildung 5.12: Elegantere Programmierung des Farbkastens

Auf den ersten Blick sieht der Sketch von Abbildung 5.12 recht kompliziert aus. Doch keine Angst. Er wird nun ausführlich erklärt. In den Zeilen 1 bis 3 werden die globalen Variablen deklariert. Der Variablen „skalierung“ wird ein konkreter Wert zugeordnet (initialisiert). Hiermit legen wir den Durchmesser der Farbtöpfe fest.

Bei *function setup()* tauchen zwei neue Größen auf, **width** (Breite) und **height** (Höhe). Mit *createCanvas(600, 400)* wird der Zahlenwert 600 von p5.js der Systemvariablen *width* und der Zahlenwert 400 der Systemvariablen *height* zugeordnet. Ändern wir mit *createCanvas()* die Größe unserer Leinwand, so werden diese neuen Werte automatisch *width* und *height* zugeordnet. Mit *spalten = width/skalierung* und *zeilen = height/skalierung* wird anschließend die Anzahl der Spalten und Zeilen berechnet, die bei gegebener Leinwandgröße und gegebenem Farbtopfdurchmesser auf die Leinwand passen. Der Schrägstrich bedeutet „geteilt“. Ändern wir die Größe der Leinwand oder den Durchmesser der Farbtöpfe, so erfolgt eine automatische Anpassung an die neuen Gegebenheiten (siehe Abb. 5.13). Dies ist sehr praktisch.

Die berechneten Zahlenwerte für *spalten* und *zeilen* bestimmen die möglichen Werte für *x* und *y* in den beiden for-Schleifen. In der Sketchzeile 19 werden dann die Bedingungen für das Zeichnen der Farbtöpfe festgelegt. Mit $x*skalierung+skalierung/2$ und $y*skalierung+skalierung/2$ werden die Mittelpunkte Farbtöpfe um jeweils einen halben Durchmesser nach rechts bzw. nach unten verschoben, damit sich für die Werte $x = 0$ und $y = 0$ keine halben Farbtöpfe im Farbkasten befinden. Mit $skalierung-skalierung/10$ und $skalierung-skalierung/10$ wird der Durchmesser der Farbtöpfe um 10% reduziert, damit sich ihre weißen Ränder nicht überlappen.

Neu im Sketch von Abbildung 5.12 sind auch die beiden Schrägstriche *//*. Sie erlauben es, einen erklärenden Text hinter eine Programmzeile zu schreiben, ohne dass dies den Programmablauf stört. Will man einen längeren Text schreiben, so kann man diesen Text zwischen zwei Schrägstriche mit Sternchen schreiben */* Text */*. Bei längeren Sketchen ist es häufig sinnvoll, einige Erläuterungen einzufügen, damit der Leser oder man selbst nach einiger Zeit noch weiß, wofür die eine oder andere Programmzeile gut ist.



Abbildung 5.13: Farbkästen mit veränderter Skalierung (skalierung = 100, = 50, = 200 v.l.n.r)

Wie flexibel die elegante Programmierung des Sketches von Abbildung 5.12 ist, kann man dem Sketch von Abbildung 5.14 entnehmen. Geringfügige Änderungen in Zeile 19 erzeugen uns anstelle von einem Farbkasten ein Kachelmuster.

Möchte man den ganzen Bildschirm (Display) mit einem Kachelmuster versehen, so schreibt man in Zeile 6 des Sketches von Abbildung 5.14 anstelle von `createCanvas(600, 400)` die Funktion **`createCanvas (displayWidth, displayHeight)`**. Es erfolgt hier ebenfalls eine automatische Anpassung der anderen Größen an die neue Gegebenheit.

Für weitere „Spielereien“ empfehle ich einen Blick in die Referenz von Processing. Siehe: <https://p5js.org/reference/>



Abbildung 5.14: Kachelmuster

Es sei noch angemerkt, dass die Kombination von for-Schleife mit der Funktion `random()` zwar recht schnell brauchbare Ergebnisse liefert, doch darf man nicht vergessen, dass man hierbei die Farbgestaltung zum Teil dem Zufall überlässt. Wenn man dies nicht möchte, dann programmiert man einfach Schritt für Schritt, sozusagen „zu Fuß“ ein schönes Bild. Schaffen wir also ein eigenes Bild (Abb. 5.15) nach den Vorgaben von Künstler Piet Mondrian. Hilfreich ist dabei, wenn man im Online-Editor von p5.js bei Auto-refresh ein Häkchen setzt.

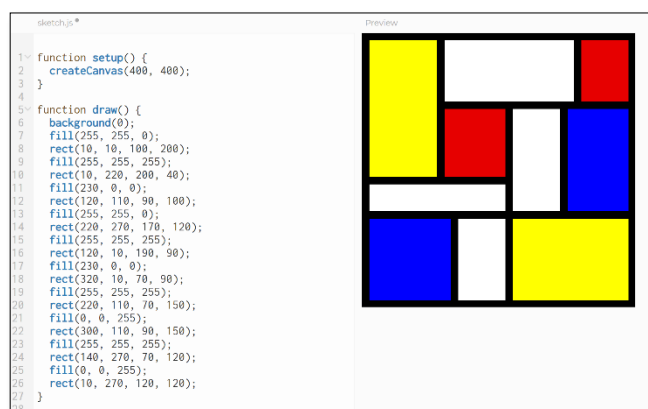


Abbildung 5.15: farbige Rechtecke

5.3 if-Anweisung

Zum Entspannen wollen wir endlich mal ein Spiel, so eine Art zweidimensionales Squash-Spiel programmieren. Hier ist der zugehörige Sketch mit ausführlichen Erläuterungen (Abb. 5.16).

```
1 // Ballspiel
2 let db = 30.0; // Balldurchmesser
3 let xb = 50.0; // x-Startposition des Balls
4 let yb = 50.0; // y-Startposition des Balls
5 let vbx = 8.0; // Startgeschwindigkeit des Balls in x-Richtung
6 let vby = 4.0; // Startgeschwindigkeit des Balls in y-Richtung
7 let s = 100.0; // Größe des Schlägers
8 let p = 0; // Zahl der Treffer (Punkte)
9
10 function setup(){
11   createCanvas(600, 400);
12 }
13
14 function draw(){
15   background(230, 230, 210);
16
17   // Wände
18   stroke(0, 180, 0);
19   strokeWeight(15);
20   line(0, 0, 600, 0);
21   line(0, 400, 600, 400);
22   line(0, 0, 0, 400);
23
24   // Schläger
25   noStroke();
26   strokeWeight(2);
27   fill(255, 0, 0);
28   rect(width, mouseY-s/2, -10, s);
29
30   // Ball
31   xb = xb + vbx;
32   yb = yb + vby;
33   stroke(255, 0, 0);
34   fill(255, 255, 0);
35   ellipse(xb, yb, db, db);
36
37   // Bedingungen für die Reflexion an den waagerechten Wänden
38   if (yb <= db/2 || yb >= height-db/2){
39     vby = -vby;
40   }
41
42   // Bedingungen für die Reflexion an der senkrechten Wand
43   if (xb < db/2){
44     vbx = -vbx;
45   }
46   /* Bedingungen für den Punktgewinn beim Auftreffen auf den Schläger
47   und Erhöhung der Ballgeschwindigkeit in x-Richtung */
48   if ((abs(mouseY-yb) < 50) && xb >= width){
49     p++;
50     vbx = -vbx*1.2;
51   }
52
53   // Ein neuer Ball wird beim Verlassen des Spielfeldes erzeugt
54   if (xb > width+20){
55     vbx = 8.0;
56     xb = random(50, 100);
57     yb = random(50, 350);
58   }
59
60   // Text
61   noStroke();
62   fill(0, 0, 255);
63   textSize(30);
64   text("Punkte " +p, 50, 100);
65   fill(0);
66   text("Zeit " +round((30-millis()/1000)), 50, 50);
67
68   // Zeitliche Begrenzung für das Spiel
69   if (millis()/1000 >= 30 ){
70     fill(255, 0, 0);
71     text("Spielende!", 50, 150);
72     noLoop();
73   }
74 }
75
```

Abbildung 5.16: Sketch Ballspiel

Bei der Programmierung des Sketches von Abbildung 5.16 lernen wir, wie nicht anders zu erwarten, auch einige neue Anweisungen und Funktionen kennen. Bei diesem Spiel wird ein gelber Ball an den grünen Wänden reflektiert und mit dem roten Schläger kann er wieder ins Spielfeld zurückgeschlagen werden (Abb. 5.17). Den Schläger kann man mittels Mausbewegung in y-Richtung bewegen. Im

Spielfeld wir die verbleibende Zeit in Sekunden und die Punktzahl angezeigt. Jedes Mal, wenn man mit dem Schläger den Ball trifft, erhält man einen Punkt. Mit jedem Treffer wird der Ball jedoch schneller und es wird immer schwieriger den Ball zu treffen. Verlässt der Ball an der rechten Seite das Spielfeld, so wird im Spielfeld ein neuer Ball erzeugt. Die Spielzeit beträgt 30 Sekunden.



Abbildung 5.17: 2D-Squash

Schauen wir uns nun aber einmal den Sketch mit seinen neuen Anweisungen und Funktionen an (Abb. 5.16). Als erstes fallen einem die fünf **if-Anweisungen** auf. Da „if“ übersetzt „wenn“ bedeutet, sind diese Anweisungen recht einfach zu verstehen. Betrachten wir die erste if-Anweisung in der folgenden Abbildung etwas genauer (Zeile 37 – 40).

```
37 // Bedingungen für die Reflexion an den waagerechten Wänden
38▼ if (yb <= db/2 || yb >= height-db/2){
39     vby = -vby;
40 }
```

Abbildung 5.18: Erste if-Anweisung im Sketch Ballspiel

In Worte gefasst bedeutet diese if-Anweisung: „Wenn der yb-Wert (Ballmittelpunkt) des Balls kleiner oder gleich dem halben Balldurchmesser ($db/2$) ist oder der yb-Wert des Balls größer oder gleich der Fensterhöhe minus dem halben Balldurchmesser ist, dann ändere den vby-Wert der Ballgeschwindigkeit durch das Vorsetzen eines Minuszeichens um 180° .“ Der x-Wert der Ballgeschwindigkeit bleibt hierbei erhalten. Er wird erst bei der Reflexion an der linken Seitenwand (zweite if-Anweisung) oder bei der Reflexion am Schläger (dritte if-Anweisung) geändert. Diese dritte if-Anweisung bewirkt aber etwas mehr als eine Richtungsänderung. Schauen wir uns die einzelnen Zeilen genauer an (Abb. 5.19).

```
48▼ if ((abs(mouseY-yb) < 50) && xb >= width){
49     p++;
50     vbx = -vbx*1.2;
51 }
```

Abbildung 5.19: Dritte if-Anweisung im Sketch Ballspiel

In den Zeilen von 48 bis 51 steht: „Wenn der Betrag des Abstandes zwischen dem y-Wert des Mauszeigers und dem Ballmittelpunkt kleiner 50 Pixel ist und der xb-Wert des Balls größer oder gleich der Fensterbreite ist, dann soll ein Punkt zum Punktekonto hinzugefügt werden, die x-Richtung des Balls umgekehrt werden und die Geschwindigkeit des Balls um den Faktor 1,2 erhöht werden.“ Die

Betragsfunktion **abs()** sorgt dafür, dass wir stets positive Werte erhalten. Dies ist hier notwendig, da die Rechnung $mouseY - yb$ nicht nur positive Werte, sondern auch negative Werte ergeben kann. Und negative Werte sind nun mal stets kleiner als 50.

Erwähnenswert sind noch die Zeilen 66 und 72.

```
66 | text("Zeit" + round((30 - millis() / 1000)), 50, 50);
72 | noLoop();
```

Abbildung 5.20: Zeile 66 und 72 aus dem Sketch Ballspiel

Die Funktion **millis()** gibt die Zeit in Millisekunden an, die seit dem Start des Sketches vergangen ist. Teilen wir diesen Wert durch 1000, so erhalten wir eine Zeitangabe in Sekunden mit vielen Nachkommastellen. Dieser Wert wird von 30 abgezogen. Mit der Funktion **round()** runden wir das Ergebnis auf einen ganzzahligen Wert, der dann im Fenster angezeigt wird. Wenn entsprechend der letzten if-Anweisung in unserem Sketch 30 Sekunden vergangen sind, dann wird mit der Funktion **noLoop()** das Spiel beendet. Die Funktion **noLoop()** beendet den Schleifendurchlauf von **function draw()**.

Erklärt werden muss noch, was **||** (Zeile 38, Abb. 5.18) und **&&** (Zeile 48, Abb. 5.19) bedeuten. Diese beiden Zeichen sind logische Operatoren. **||** bedeutet **oder** und **&&** bedeutet **und**. Dann gibt es noch das Zeichen **^** für das **exklusive oder** und das Zeichen **!** für die **Negation**.

5.4 else-Anweisung

Bei unserem Spiel 2D-Squash haben wir die if-Anweisung kennengelernt. Ein guter Freund der if-Anweisung ist die **else-Anweisung**. Um ihre Wirkung zu verstehen, schauen wir uns den Sketch von Abbildung 5.22 an. In den Zeilen 7 bis 9 steht die if-Anweisung. Sie ist eine Wenn-dann-Anweisung: **Wenn** es wahr ist, dass eine Tastaturtaste gedrückt ist (**keysPressed == true**), **dann** setze die Füllfarbe auf Rot (fill(255, 0, 0)). Zwischen den runden Klammern der if-Anweisung steht also die Bedingung und zwischen den geschweiften Klammern steht die Anweisung. Was soll aber geschehen, wenn keine Taste gedrückt ist? Hier kommt jetzt die else-Anweisung ins Spiel (Zeile 11 bis 13). Da die if-Anweisung nicht erfüllt ist, setzt die else-Anweisung die Füllfarbe nun auf Grün (fill(0, 255, 0)). Das folgende Diagramm veranschaulicht diesen Vorgang (true = wahr, false = falsch).

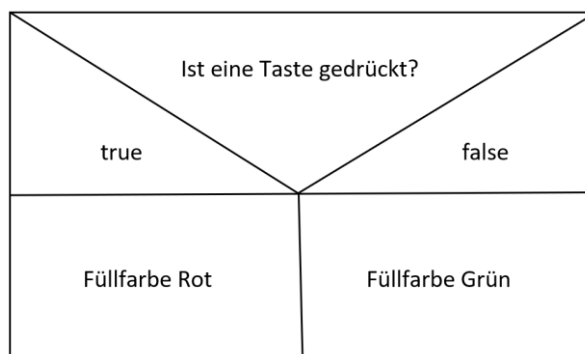


Abbildung 5.21: Struktur-Diagramm zu einer if- und else-Anweisung

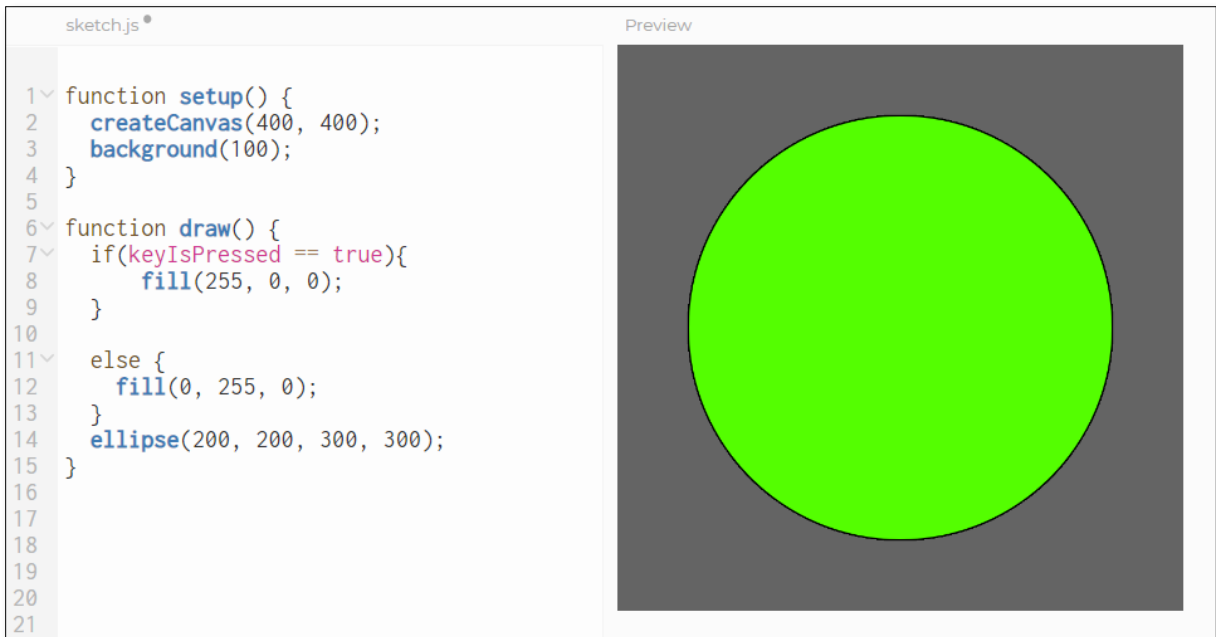


Abbildung 5.22: Sketch zu einer einfachen if- und else-Anweisung

Setzt man anstelle von *true* das Wort *false* in Zeile 7 von Abbildung 5.22 ein, dann erscheint ein roter Kreis und beim Drücken einer Taste ein grüner Kreis.

Wenn man den Sketch, wie in Abbildung 5.22 dargestellt, im Online-Editor programmiert, dann funktioniert er erst, wenn man zu Beginn mit der Maus ins Preview-Fenster (Vorschaufenster) klickt.

Erklärt werden muss noch das doppelte Gleichheitszeichen `==`. Das Zeichen `==` ist ein **Gleichheitsoperator**, während das einfache Gleichheitszeichen `=` ein **Zuweisungsoperator** ist. Dies wird oft verwechselt und sorgt so für böse Programmfehler.

Beispiel Zuweisungsoperator: `let a = i;` Der Variablen *a* wird die Größe *i* zugewiesen.

Beispiel Gleichheitsoperator: `if(x == y){print(„Richtig!“);}` Nur wenn *x* gleich *y* ist, ist der Ausdruck in der runden Klammer wahr und das Wort „Richtig!“ wird in die Konsole geschrieben.

Schreibt man `if(x = y){print(„Richtig!“);}`, dann wird das Wort „Richtig!“ immer in die Konsole geschrieben, auch wenn *x* ungleich *y* ist. Dies geschieht deshalb, weil eine Zuordnung immer wahr ist.

Mit dem Sketch von Abbildung 5.22 konnte man die Farbe des Kreises ändern, wenn man eine beliebige Taste drückte. In dem folgenden Sketch (Abb. 5.23) soll der Kreis nur dann mit der Farbe Rot gefüllt werden und diese Farbe auch nach dem Loslassen der Taste behalten, wenn die Taste *r* gedrückt wird. Er soll nur dann mit der Farbe Grün gefüllt werden und diese Farbe auch nach dem Loslassen der Taste behalten, wenn die Taste *g* gedrückt wird. Dies erreichen wir mit einer if-Anweisung und einer else-if-Anweisung unter Verwendung des Gleichheitsoperators `==` (siehe Abb. 5.23 Zeile 7 und 10). Der Ausdruck `key == „r“` bzw. `key == „g“` ist nur dann wahr, wenn die Taste *r* bzw. die Taste *g* gedrückt wird. **key** ist eine Systemvariable. Diese merkt sich, auch nach dem Loslassen der Taste, den Wert der zuletzt verwendeten Tastaturtaste. Was passiert aber, wenn eine andere Taste als *r* oder *g* gedrückt wurde? Dann wird die else-Anweisung von Zeile 13 bis 15 aktiv und füllt den Kreis mit der Farbe Gelb.

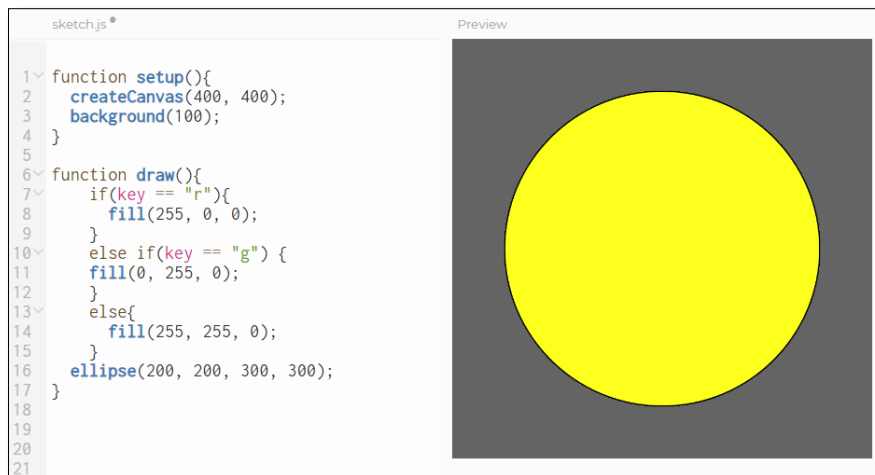


Abbildung 5.23: Dieser Sketch enthält neben einer if- und else-Anweisung auch eine else-if-Anweisung

5.5 let und var

p5.js ist eine Bibliothek der Programmiersprache JavaScript. JavaScript wurde 1995 von Netscape ins Leben gerufen und seitdem immer weiterentwickelt. Es dauerte jedoch stets eine gewisse Zeit, bis sich die Mehrzahl der Programmierer mit den jeweiligen Änderungen vertraut gemacht hatten. Schaut man sich im Internet oder in der Literatur Sketche von p5.js an, dann stellt man fest, dass zum Beispiel die Variable `x` einmal mit **var x** und einmal mit **let x** deklariert wird. Die Variablenbezeichnung *let* löste im Jahre 2015 die Bezeichnung *var* ab (ECMAScript 6). Tauscht man bei den Sketchen hier in diesem Buch die beiden Variablenbezeichnungen gegeneinander aus, dann sieht man, dass sich bei diesen Sketchen die Ergebnisse nicht ändern. Dies ist gerade für Anfänger etwas verwirrend. Ist *var x* nun das Gleiche wie *let x* oder gibt es einen Unterschied? Als globale Variable besitzen sie die gleiche Wirkung. Benutzt man sie jedoch als lokale Variable, so kann sich, je nach Programmcode, der Unterschied bemerkbar machen. Der Hauptunterschied zwischen ihnen ist, dass der Gültigkeitsbereich der lokalen Variable *var* auf die jeweilige Funktion beschränkt ist, innerhalb der sie deklariert wird. Dies kann zum Beispiel die Funktion `function draw()` sein. Der Gültigkeitsbereich der lokalen Variable *let* ist dagegen auf den Block beschränkt, innerhalb der sie deklariert wird*. Unter einem Block versteht man den Codeteil, der zwischen zwei geschweiften Klammern `{ }` steht. Diese Änderung hat man eingeführt, um Probleme zu vermeiden, die es mit *var* bei einigen Sketchen geben kann. Die Abbildungen 5.24 und 5.25 zeigen dies. *if(true)* steht hier für: „wenn es wahr ist ..., dann ...“. In Abbildung 5.24 sieht man, dass die lokale Variable *let x* in den Zeilen 4 und 7 nur im Bereich zwischen den zugehörigen geschweiften Klammern wirkt. In Abbildung 5.25 steht die lokale Variable *var x* ebenfalls in den Zeilen 4 und 7. Da ihr Wirkungsbereich aber auf die ganze Funktion `function setup()` beschränkt ist, gibt es hier ein Problem. Was soll nun in die Konsole geschrieben werden? 20 oder 40? Zeile 8 und Zeile 10 liefern als Ergebnis 40, obwohl `x` doch sowohl 20 wie auch 40 ist. Bei *let x* gibt es dieses Problem nicht.

*Wenn eine Variable mit *let* in einer for-Schleife deklariert wird, dann steht sie nicht zwischen zwei geschweiften Klammern. In diesem Fall ist ihr Wirkungsbereich auf die for-Anweisung beschränkt. Wer sich intensiver mit der Weiterentwicklung von JavaScript beschäftigen möchte, dem sei dieser Link empfohlen: <https://de.wikipedia.org/wiki/JavaScript>

```

1 let x = 10; // globale Variable
2
3 function setup() {
4     let x = 20; // lokale Variable
5
6     if(true){
7         let x = 40; // lokale Variable
8         print(x) // In der Konsole steht 40
9     }
10    print(x); // In der Konsole steht 20
11 }
12
13 function draw(){
14     print(x); // In der Konsole steht 10
15 }
16

```

Abbildung 5.24: Die Variablendeklaration mit `let` gilt nur im jeweiligen Codeblock

```

1 var x = 10; // globale Variable
2
3 function setup() {
4     var x = 20; // lokale Variable
5
6     if(true){
7         var x = 40; // lokale Variable
8         print(x) // In der Konsole steht 40
9     }
10    print(x); // In der Konsole steht 40 anstatt 20
11 }
12
13 function draw(){
14     print(x); // In der Konsole steht 10
15 }
16

```

Abbildung 5.25: Die Variablendeklaration mit `var` gilt innerhalb der jeweiligen Funktion. Dadurch können Probleme entstehen.

5.6 Zusammenfassung

Fassen wir zusammen, was wir in Kapitel 5 gelernt haben, bzw. gelernt haben sollten.

Systemvariable Systemvariablen sind von p5.js fest vorgegeben. Ein Beispiel ist *mouseX*. Die Systemvariable *mouseX* ist ein Platzhalter für die x-Koordinate des Mauszeigers im Browserfenster.

globale Variable Eine globale Variable, wie zum Beispiel `let x = 0`, erstellt der Programmierer vor dem Schreiben der Funktionen *function setup()* und *function draw()*. Alle Funktionen können auf die globalen Variablen zugreifen.

lokale Variable Die Variablen, die in Codeblöcken deklariert und initialisiert werden, nennt man lokale Variablen. Die lokalen Variablen, die mit *let* deklariert werden, können nur innerhalb ihrer Codeblöcke aufgerufen werden. Die lokalen Variablen, die mit der veralteten Bezeichnung *var* deklariert werden, können nur innerhalb ihrer Funktion aufgerufen werden.

deklarieren und initialisieren Schreiben wir `let x = 0`, so haben wir einer Variablen den Namen *x* gegeben und ihr den Wert 0 zugeordnet. In der Sprache der Informatiker sagt man, man habe eine Variable deklariert und initialisiert.

for-Schleife Ein schnellerer Bildaufbau gelingt mittels einer for-Schleife. Hier ein Beispiel.

```
for(let x = 0; x <= 800; x = x + 10) {  
  fill(0);  
  rect(x, 0, 5, 300);  
}
```

In dieser for-Schleife wird innerhalb der runden Klammern zuerst die Variable *x* deklariert und initialisiert. Nach dem Semikolon (;) wird mit `x <= 800` festgelegt, dass *x* nur Werte kleiner gleich 800 annehmen darf. Nach dem zweiten Semikolon steht `x = x + 10`. D.h., bei jedem Durchlaufen der for-Schleife soll *x* um den Wert 10 erhöht werden.

Mittels einer for-Schleife gelingt der Aufbau eines Bildes oder einer Animation viel schneller als ohne for-Schleife, da innerhalb der for-Schleife zuerst alle *x*-Werte berechnet werden und erst danach zeichnet die Funktion *function draw()* das Bild oder die Animation.

doppelte for-Schleife Wie wir in dem folgenden Beispiel sehen, wird hier eine zweite for-Schleife in die erste integriert.

```
for(let zeilen = 0; zeilen < 4; zeilen = zeilen + 1) {  
  for(let spalten = 0; spalten < 6; spalten = spalten + 1) {  
    fill(random(255), random(255), random(255));  
    ellipse(d*spalten+d/2, d*zeilen+d/2, d, d);  
  }  
}
```

Dies beiden Schleifen arbeiten zusammen und berechnen die Werte, die die Funktion *function draw()* braucht, um die Ellipsen zu zeichnen. Zuerst setzt die innere for-schleife für die lokale Variable „spalten“ die Zahlenwerte 0, 1, 2, 3,

4, 5 ein und lässt die Funktion *random()* die Zufallswerte für die Farben der Ellipsen erzeugen. Danach erhöht die äußere for-Schleife den Wert der lokalen Variablen „zeilen“ um den Wert 1. Anschließend erzeugt die innere Schleife die Werte für die zweite Reihe der Ellipsen. Danach erhöht die äußere for-Schleife den Wert der lokalen Variablen „zeilen“ wieder um den Wert 1. Dies wird solange fortgesetzt, bis die Variable „zeilen“ den Wert 3 erreicht hat. Erst danach zeichnet die Funktion *function draw()* die Ellipsen.

random()	Die Funktion <i>random()</i> stellt dem Programmierer einen Zufallsgenerator zur Verfügung. Schreibt man <i>random(200)</i> , so generiert die Funktion Zahlen von 0,00... bis einschließlich 199,99.... Schreibt man <i>random(50, 100)</i> , so generiert die Funktion Zahlen von 50,00... bis einschließlich 99,99....
textAlign(CENTER)	Mit <i>textAlign(CENTER)</i> kann man die Textmitte auf einen gewünschten x-Wert setzen.
width() und height()	<i>width</i> (Breite) und <i>height</i> (Höhe) sind Systemvariablen. Schreiben wir zum Beispiel <i>createCanvas(600, 400)</i> so wird der Zahlenwert 600 von p5.js der Systemvariablen <i>width</i> und der Zahlenwert 400 der Systemvariablen <i>height</i> zugeordnet. Diese beiden Systemvariablen können wir dann im weiteren Verlauf des Sketches benutzen.
displayWidth und displayHeight	Möchte man den ganzen Bildschirm mit der programmierten Abbildung ausfüllen, dann gelingt dies mit <i>createCanvas(displayWidth, displayHeight)</i> .
if-Anweisung	Die if-Anweisung soll anhand des folgenden Beispiels erklärt werden. <pre>if(x > 100) { y = -y;}</pre> Da „if“ übersetzt „wenn“ bedeutet, lesen sich die obigen Zeilen wie folgt. Wenn x größer 100 ist, dann setze vor die Variable y ein Minuszeichen.
abs()	Die Betragsfunktion <i>abs()</i> verwandelt negative Werte in positive Werte.
millis()	Die Funktion <i>millis()</i> gibt die Zeit in Millisekunden an, die seit dem Start des Sketches vergangen ist.
noLoop()	Funktion <i>noLoop()</i> beendet den Schleifendurchlauf von <i>function draw()</i> .
logische Operatoren	 bedeutet oder und && bedeutet und . Das Zeichen ^ steht für das exklusive oder und das Zeichen ! für die Negation .
else	<i>else</i> heißt übersetzt „sonst“. Wenn die Bedingung einer if-Anweisung nicht erfüllt ist, dann entscheidet die nachfolgende else-Anweisung, was gesehen soll.
else if()	Mit <i>else if()</i> wird eine Bedingung nur dann geprüft, wenn die erste Bedingung in der vorhergehenden if-Anweisung nicht erfüllt ist.
keysPressed()	Die Funktion <i>keysPressed()</i> wird immer dann aufgerufen, wenn eine Taste auf der Tastatur gedrückt wird.
key	Die Systemvariable <i>key</i> merkt sich, auch nach dem Loslassen der Taste, den Wert der zuletzt gedrückten Tastaturtaste.

true	<i>true</i> heißt übersetzt „wahr“
false	<i>false</i> heißt übersetzt „falsch“
==	Der Gleichheitsoperator (==) liefert dann die Aussage „wahr“ (<i>true</i>), wenn die Objekte vor und hinter dem doppelten Gleichheitszeichen gleich sind.
=	Der Zuweisungsoperator (=) darf nicht mit dem Gleichheitsoperator verwechselt werden. Er weist zum Beispiel einer Variablen nur eine andere Größe zu.

5.7 Übungen

Aufgabe 1 Ändere den Sketch von Abbildung 5.9 so ab, dass alle zwei Sekunden ein Bild entsteht, welches dem folgenden Bild ähnelt.



Abbildung 5.24: Kritzelbild

Aufgabe 2 Erstelle die folgende Abbildung mit einer for-Schleife im HSB-Farbraum.

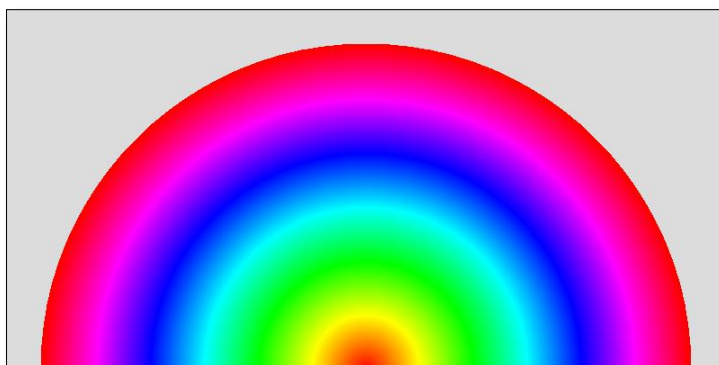


Abbildung 5.257: HSB-Regenbogen

Aufgabe 3 Bei dem Sketch von Abbildung 5.28 (s. u.) baut sich das Streifenmuster nur langsam nacheinander auf. Ändere den Sketch mittels eine for-Schleife so um, dass das Streifenmuster sofort auf der Leinwand erscheint. Verwende *let* anstelle von *var*.

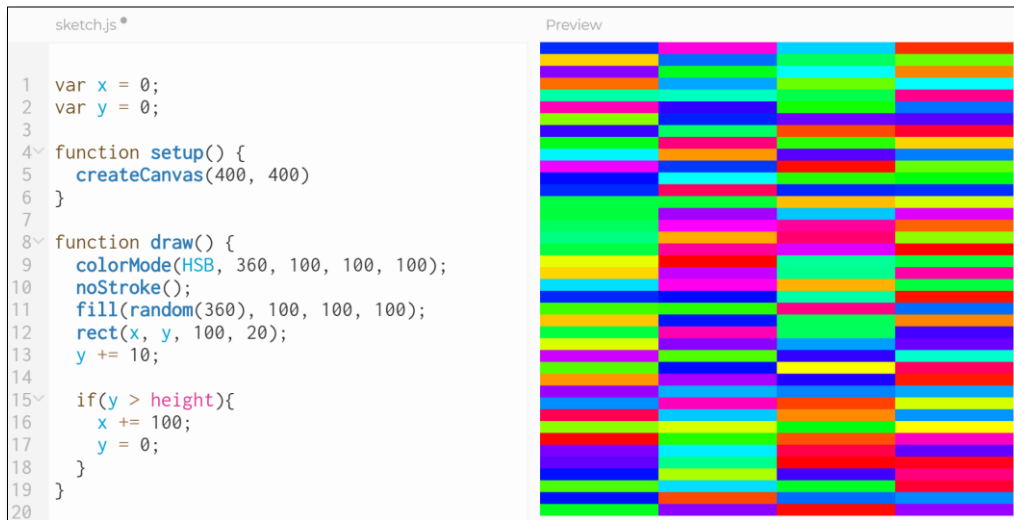


Abbildung 5.268: Die Abbildung erinnert an das Fenster von Gerhard Richter im Kölner Dom

Aufgabe 4 Ändere den Sketch von Abbildung 5.16 und 5.17 so um, dass ein neues Spiel entsteht. Bei diesem neuen Spiel sollen Bälle von der Decke fallen, die mit einem Korb aufgefangen werden müssen (siehe Abb. 5.29). Der Korb kann in x-Richtung mit der Maus bewegt werden. Fängt man mit dem Korb einen Ball, dann bekommt man einen Punkt. Danach fällt mit erhöhter Geschwindigkeit, an einer zufällig festgelegten Stelle, ein neuer Ball von der Decke. Immer wenn man einen Ball fängt, erhöht sich die Fallgeschwindigkeit. Verfehlt man den Ball, dann fällt ein neuer Ball mit der ursprünglichen Fallgeschwindigkeit von der Decke. Nach dreißig Sekunden ist das Spiel beendet.

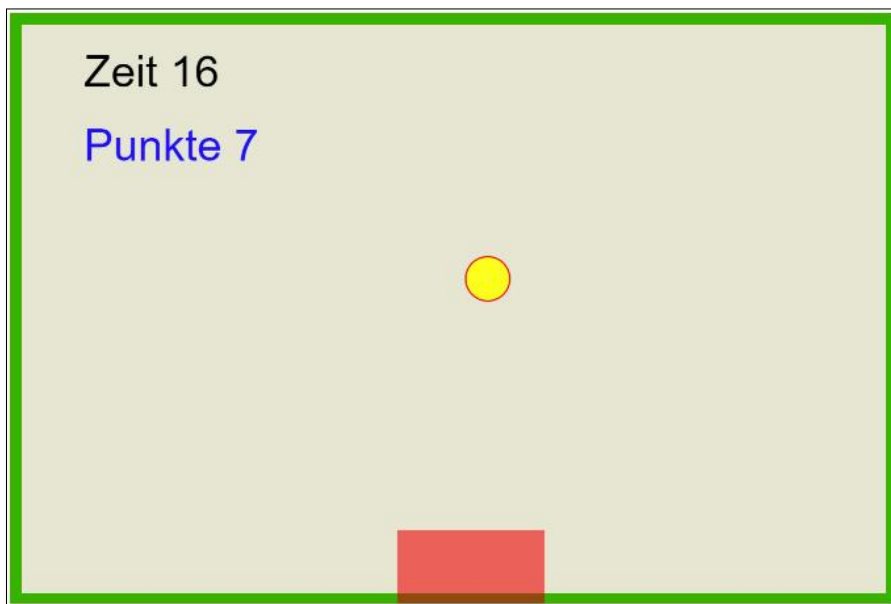


Abbildung 5.27: Fang den Ball!

Aufgabe 5 Schreibe einen Sketch der dreimal „Niemals die Taste x drücken!!!“ auf die Leinwand schreibt (Abb. 5.28, links). Sobald jemand auf die x-Taste drückt, soll auf der Leinwand das rechte Bild erscheinen (Abb. 5.28, rechts). Wenn man eine andere Taste als die x-Taste drückt, dann soll dies nicht geschehen.



Abbildung 5.30: Warnung nicht befolgt!

6 Bézierkurven und Perlin noise

6.1 Bézierkurven

Am Ende von Kapitel 2 haben wir Kurven mit der Funktion `curve()` gezeichnet. Elegantere Kurven erhält man jedoch mit der Funktion `bezier()`. Abbildung 6.1 zeigt vier Beispiele. Die Form der Kurve wird durch zwei **Ankerpunkte** (schwarz in Abbildung 6.1) und zwei **Kontrollpunkte** (rot in Abbildung 6.1) festgelegt. Die Verbindungslinie zwischen einem Ankerpunkt und dem zugehörigen Kontrollpunkt stellt eine Tangente dar, die am jeweiligen Ankerpunkt an der Kurve anliegt. Die Kontrollpunkte und die Tangenten werden in der Regel nicht gezeichnet. Sie wurden in Abbildung 6.1 nur zum besseren Verständnis eingezeichnet. Aus der Lage der vier Punkte wird, nach einem nicht gerade einfachen Verfahren, der Verlauf der Kurve berechnet. Informationen hierzu findet man im Internet. Wir begnügen uns hier jedoch mit einfachen praktischen Anwendungen.

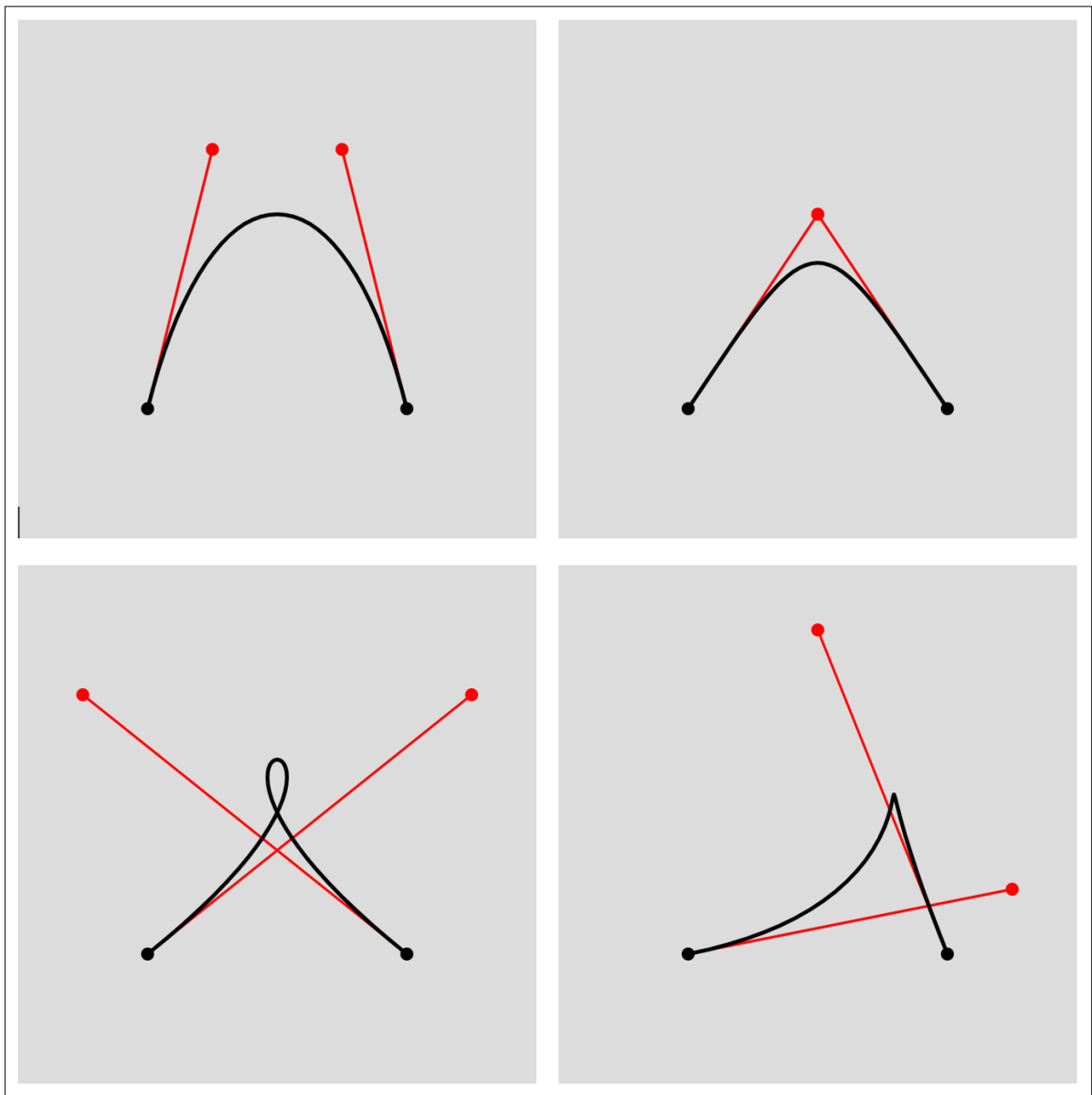


Abbildung 6.1: Vier Bézierkurven mit gleichen Ankerpunkten, aber unterschiedlichen Kontrollpunkten

Der folgende Sketch (Abb. 6.2) soll uns helfen, die ersten einfachen Übungen mit einer Bézierkurve durchzuführen, um so die sich bildenden Formen kennenzulernen. Die beiden Ankerpunkte (schwarz) bleiben bei diesen Übungen fest an Ort und Stelle. Beweglich sind dagegen die Kontrollpunkte (rot). Für sie haben wir am Anfang des Sketches die Variablen `x1`, `y1`, `x2` und `y2` deklariert und initialisiert. Mit diesen Werten kann man nun spielen. Hilfreich ist es, wenn man im Webeditor ein Häkchen bei *Auto refresh* setzt. So sieht man sofort, wie sich die Kurve verändert, wenn man einen Variablenwert ändert. Versuche zuerst unter Verwendung des Sketches von Abbildung 6.2 die Kurven von Abbildung 6.1 nachzuzeichnen. Zeichne danach eigene Kurven.

Der Term `bezier(100, 300, x1, y1, x2, y2, 300, 300)` ist die entscheidende Zeile (Zeile 28) im Sketch von Abbildung 6.2. Das erste (100, 300) und das letzte Zahlenpaar (300, 300) legen den jeweiligen Ort der Ankerpunkte fest. Die beiden mittleren Zahlenpaare (`x1`, `y1`, `x2`, `y2`) legen den jeweiligen Ort der Kontrollpunkte fest. **Merke: Die Funktion `bezier()` benötigt vier Zahlenpaare. Zwei für die beiden Ankerpunkte (erstes und letztes Zahlenpaar) und zwei für die beiden Kontrollpunkte (die beiden mittleren Zahlenpaare).**

The screenshot shows a sketch editor with a code editor on the left and a preview window on the right. The code editor contains the following code:

```

1 let x1 = 150;
2 let y1 = 100;
3 let x2 = 250;
4 let y2 = 100;
5
6 function setup() {
7   createCanvas(400, 400);
8 }
9
10 function draw() {
11   background(220);
12
13   // Kontrollpunkte
14   stroke(255, 0, 0);
15   strokeWeight(10);
16   point(x1, y1);
17   point(x2, y2);
18
19   // Tangenten
20   strokeWeight(2);
21   line(100, 300, x1, y1);
22   line(300, 300, x2, y2);
23
24   // Bézierkurve
25   noFill();
26   stroke(0);
27   strokeWeight(3);
28   bezier(100, 300, x1, y1, x2, y2, 300, 300);
29
30   // Ankerpunkte
31   strokeWeight(10);
32   point(100, 300);
33   point(300, 300);
34 }
35

```

The preview window shows a gray canvas with a black Bézier curve. The curve starts at a black dot at (100, 300) and ends at a black dot at (300, 300). Two red dots represent the control points at (150, 100) and (250, 100). Red lines connect the start point to the first control point and the end point to the second control point, representing the tangents.

Abbildung 6.2: Sketch mit Variablen für die Kontrollpunkte (rot)

Mehrere Bézierkurven kann man auch zu einer Kurve zusammenfügen. Dies wollen wir uns am Beispiel einer Kerzenflamme ansehen (Abbildung 6.3). Hier benötigen wir die folgenden zwei Bézierkurven (Zeile 22 und 23 im Sketch von Abbildung 6.3).

```

bezier(200, 50, 100, 300, 120, 350, 250, 350);
bezier(250, 350, 350, 350, 200, 200, 200, 50);

```

Damit sich eine geschlossene Fläche ergibt, muss der erste Ankerpunkt der ersten Bézierkurve mit dem letzten Ankerpunkt der zweiten Bézierkurve zusammenfallen (violette Wertepaare). Das Gleiche gilt für den letzten Ankerpunkt der ersten Bézierkurve und den ersten Ankerpunkt für die zweite Bézierkurve (blaue Wertepaare).

Die Flammenspitze stellt einen Knick im Kurvenverlauf dar, während die beiden Bézierkurven am unteren Ende der Kerzenflamme kontinuierlich ineinander übergehen. Einen Knick im Kurvenverlauf zu erzeugen ist einfach. Aber wie erzeugt man einen kontinuierlichen Übergang? Dies schauen wir uns in Abbildung 6.4 an.

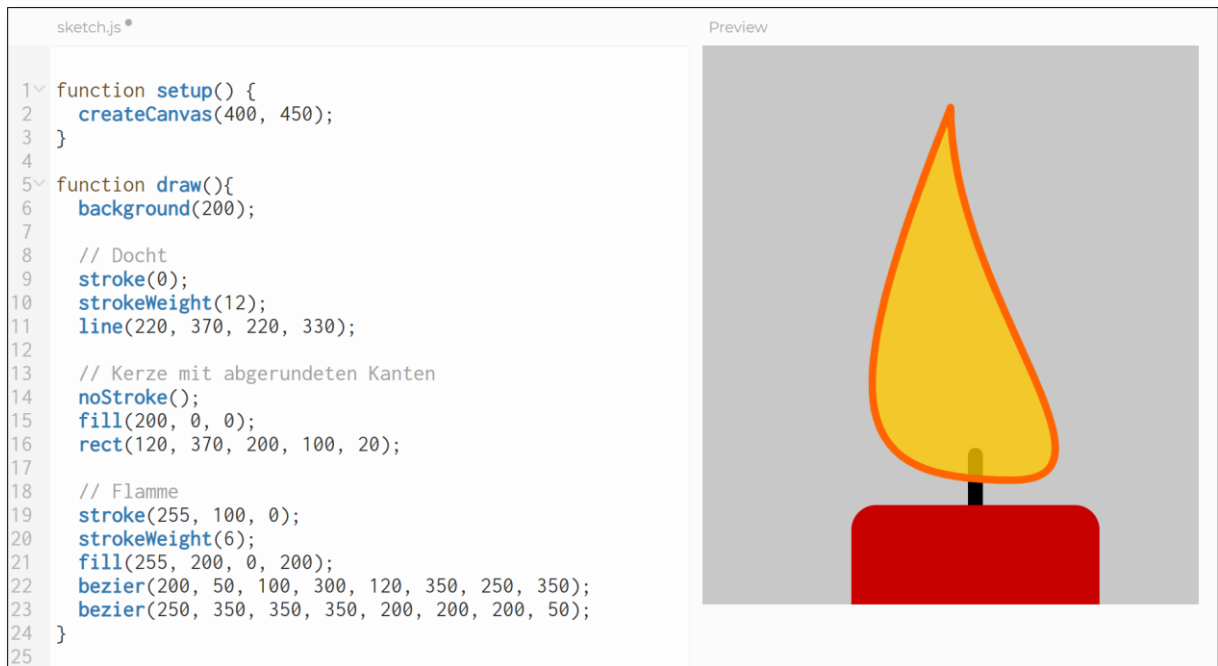


Abbildung 6.3: Zwei zusammengefügte Bézierkurven erzeugen eine Kerzenflamme

In Abbildung 6.4 berühren sich die grüne und die blaue Bézierkurve einmal mit (oben) und einmal ohne Knick (unten). Wenn man einen Knick bei zusammengesetzten Bézierkurven vermeiden will, dann muss man dafür sorgen, dass die Tangenten an den beiden sich überlagernden Ankerpunkten auf einer Geraden liegen (siehe Abb. 6.4).

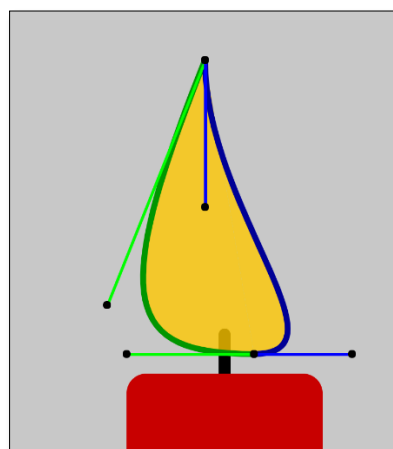


Abbildung 6.4: Kerzenflamme mit eingezeichneten Tangenten

Die rote Kerze von Abbildung 6.3 und 6.4 besitzt **abgerundete Kanten**. Dies erreicht man, wenn man bei der Funktion `rect()` eine **fünfte Zahl** hinzufügt (Zeile 16 im Sketch von Abbildung 6.3). Mit ihr kann man die Stärke der Rundung bestimmen. Je größer die fünfte Zahl, desto ausgeprägter ist die Rundung.

Natürlich kann man auch mehr als zwei Bézierkurven zusammenfügen. Abbildung 6.5 zeigt eine Figur, die aus vier Bézierkurven zusammengesetzt wurde. Da die Figur spiegelsymmetrisch ist, haben wir uns ihre Programmierung dadurch vereinfacht, dass wir den Nullpunkt der x-Achse mit der Funktion `translate()` (Zeile 7 in Abbildung 6.5) vom linken Rand der Leinwand in die Mitte der Leinwand verschoben haben. Erinnern wir uns daran, dass der Ursprung des Koordinatensystems bei p5.js in der linken oberen Ecke der Leinwand liegt. Die erste Zahl in der Klammer von `translate(200, 0)` gibt die Verschiebung in x-Richtung und die zweite Zahl gibt die Verschiebung in y-Richtung an. Da der zweite Klammerwert den Wert Null hat, wird in unserem Beispiel der Nullpunkt der y-Achse nicht nach unten oder oben verschoben.

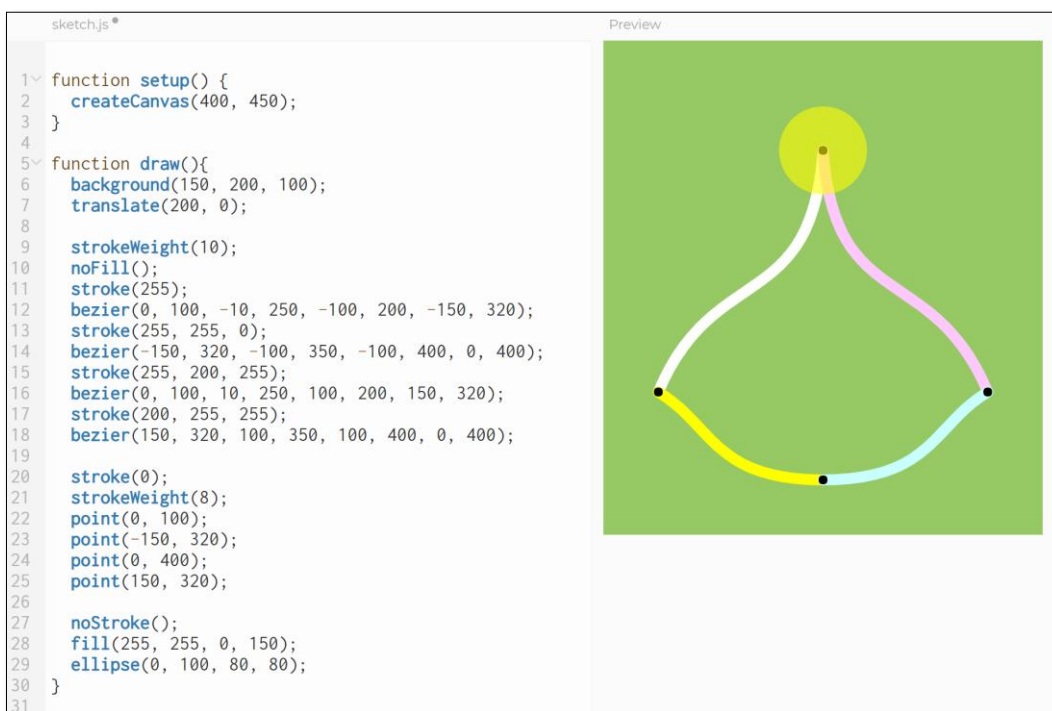


Abbildung 6.5: Eine aus vier Bézierkurven bestehende Figur

Der Sketch von Abbildung 6.5 besitzt einen Nachteil. Man kann die so erstellte Figur nicht mit einer Füllfarbe versehen. Wenn wir dies trotzdem versuchen, dann erhalten wir das folgende Bild (Abb. 6.6). Gefüllt wird nur der Bereich zwischen der Kurve und einer Geraden, die sich zwischen den beiden Ankerpunkten erstreckt.

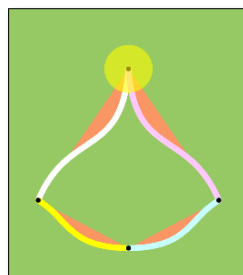


Abbildung 6.6: Ein nicht befriedigender Füllversuch

Wie es trotzdem gelingt unsere Figur mit Farbe zu füllen, zeigt der Sketch von Abbildung 6.7. Hier verwenden wir die Funktion **bezierVertex()** um die Anker- und Kontrollpunkte für jeweils eine Bézierkurve festzulegen. Alle **bezierVertex()**-Funktionen müssen hierbei zwischen **beginShape()** und **endShape()** stehen. Weiterhin muss der erste Ankerpunkt mit der Funktion **vertex()** festgelegt werden. Schauen wir uns die entscheidenden Sketchzeilen (Zeile 13 bis 19) näher an. Alle fünf Ankerpunkte sind in den folgenden Zeilen fett dargestellt. Die beiden anderen, nicht fett dargestellten Wertepaare zwischen den runden Klammern von **bezierVertex()** geben die Kontrollpunkte an.

```
beginShape();
vertex(0, 100);
bezierVertex(-10, 250, -100, 200, -150, 320);
bezierVertex(-100, 350, -100, 400, 0, 400);
bezierVertex(100, 400, 100, 350, 150, 320);
bezierVertex(100, 200, 10, 250, 0, 100);
endShape();
```

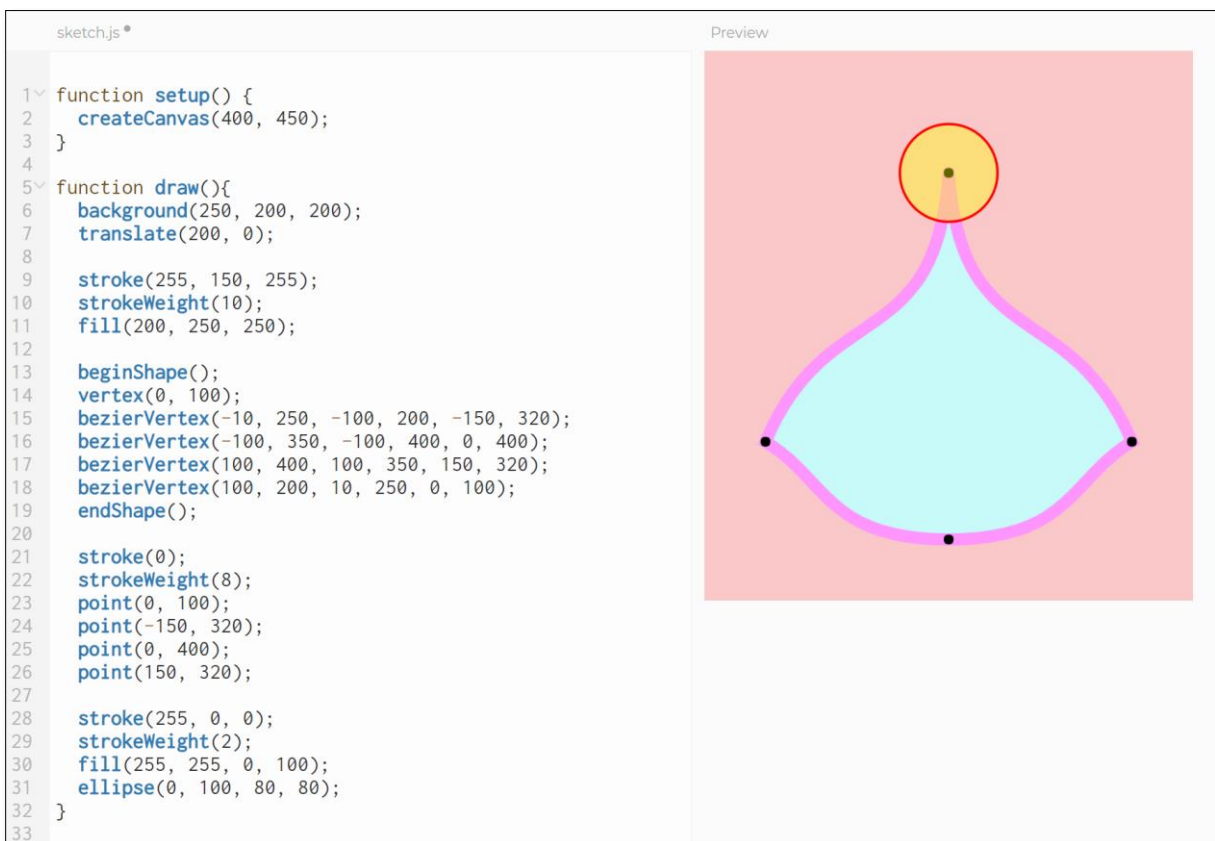


Abbildung 6.7: Aus vier Bézierkurven erstellte Figur mit Füllung

Das zielgerichtete Gestalten mit Bézierkurven bedarf einiges an Übung. Am bestens fängt man ganz einfach an und steigert sich dann zu einem wahren Künstler.

6.2 Perlin noise 1D

Der in Kapitel 6.1 generierte Sketch zeichnet uns unter Verwendung von zwei Bézierkurven eine brennende Kerze. Schön wäre es, wenn sich die Flamme der Kerze auch fortlaufend bewegen würde. Fortlaufende Bewegungen kann man mit den Funktionen **sin()** oder **cos()** erzeugen. Die links in Abbildung 6.8 dargestellte Sinusfunktion besitzt jedoch positive und negative Werte. Möchte man nur mit positiven Werten arbeiten, dann muss man sich mit der Funktion **abs(sin())** den Absolutwert ausgeben lassen (Abb. 6.8, rechts).

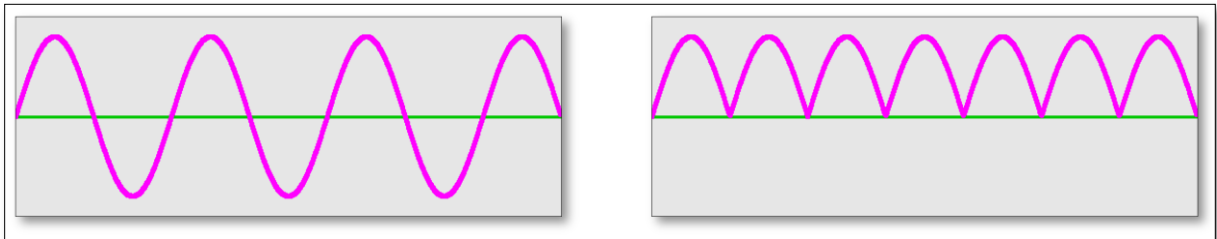


Abbildung 6.8: Mit den Funktionen `sin()` und `abs(sin())` erzeugte Funktionsgraphen

Im Sketch von Abbildung 6.9 benutzen wir in Zeile 24 und 25 die Funktion `abs(sin(w))`, um unsere Kerzenflamme in Bewegung zu versetzen. Der Buchstabe `w` steht hier für den Winkel im Bogenmaß. Den Winkel vergrößern wir in Zeile 26 in Schritten von `0,05`. Jetzt bewegt sich unsere Kerzenflamme. Das Ergebnis ist jedoch nicht unbedingt befriedigend, da diese Bewegung streng periodisch und somit recht langweilig und unnatürlich ist. Man könnte nun versucht sein, mit der Zufallsfunktion `random()` der Bewegung der Kerzenflamme diese strenge Periodizität zu nehmen. Starten wir einen Versuch mit `w = w + random(0.5)` statt `w = w + 0.05` in Zeile 26. Nun ist die Bewegung zwar nicht mehr streng periodisch, doch dafür ruckelt die Flammenbewegung nun. Dies ist auch nicht besonders schön. Was nun?

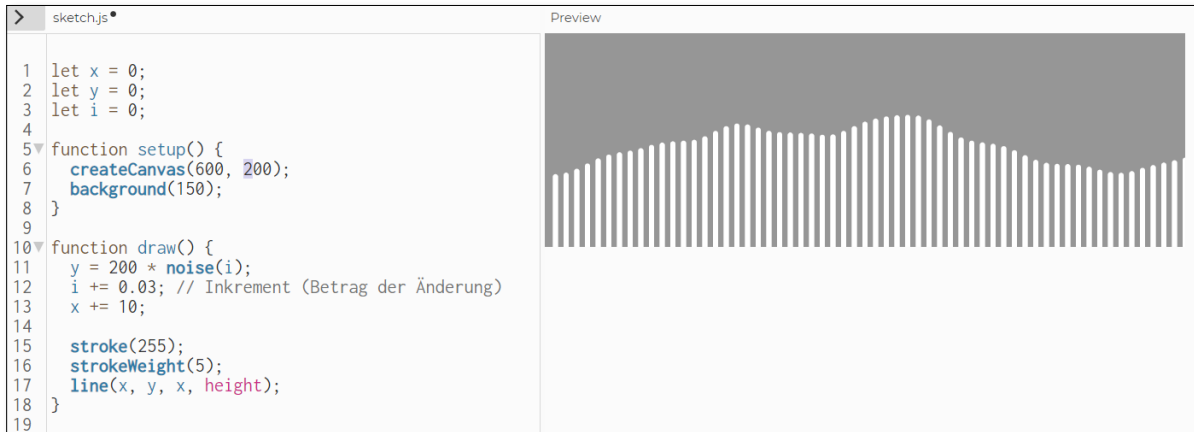
```
1 let w = 0;
2
3 function setup() {
4   createCanvas(400, 450);
5 }
6
7 function draw(){
8   background(200);
9
10  // Docht
11  stroke(0);
12  strokeWeight(12);
13  line(220, 370, 220, 330);
14
15  // Kerze mit abgerundeten Kanten
16  noStroke();
17  fill(200, 0, 0);
18  rect(120, 370, 200, 100, 20);
19
20  // Flamme
21  stroke(255, 100, 0);
22  strokeWeight(6);
23  fill(255, 200, 0, 200);
24  bezier(200, 20+50*abs(sin(w)), 100+50*abs(sin(w)), 300, 120, 350, 250, 350);
25  bezier(250, 350, 350, 350, 200+100*abs(sin(w)), 200, 200, 20+50*abs(sin(w)));
26  w = w + 0.05;
27 }
28
```

Abbildung 6.9: Der Sketch erzeugt eine sich periodisch bewegende Kerzenflamme

Das Zauberwort für die Lösung unseres Problems heißt **Perlin noise**. Eine von Kent Perlin Anfang der achtziger Jahre für den Film *Tron* entwickelte Methode, den Zufall etwas weniger zufällig zu generieren, um so zu natürlich aussehenden Ergebnissen zu kommen. Da sich mit Hilfe von Perlin noise

Programme schreiben lassen, die Texturen, Wolken und Landschaften erzeugen, wird dieses Verfahren häufig bei Computerspielen verwendet.

Wenn wir Zufallszahlen mit der Funktion `random(100)` erzeugen, dann wissen wir, dass wir voneinander unabhängige Zahlen im Bereich von 0 bis 99,999... erhalten. Wenn wir auf die gleiche Art die Funktion `noise()` bei `function draw()` benutzen, so erleben wir eine Überraschung. Schreiben wir zum Beispiel `noise(100)`, so liefert `noise(100)` bei jedem Durchlauf von `function draw()` immer die gleiche Zahl. Dies sieht nun gerade nicht nach Zufall aus. Um dies zu überprüfen, können wir uns mithilfe der Funktion `print()` diese Zahl fortlaufend in die Konsole schreiben lassen. Die hier zu sehende Zahl ist kleiner 1. Auch wenn wir andere Zahlen als 100 zwischen die runden Klammern von `noise()` schreiben, erhalten wir stets eine Zahl kleiner 1. Sie ändert sich beim Durchlauf von `function draw()` nicht. In der Konsole sehen wir stets den gleichen Wert. Für unser Ziel, eine bewegte Kerzenflamme zu erzeugen, benötigen wir bei `function draw()` aber fortlaufend andere Zahlenwerte. Wie erreichen wir dies? Schauen wir uns im Sketch von Abbildung 6.10 die Zeilen 11 und 12 an. Hier ändern wir bei jedem Durchlauf von `function draw()` die Zahl zwischen den runden Klammern von `noise()`. Wir erhöhen sie bei jedem Durchlauf von `function draw()` um den Wert 0,03. Das Fremdwort für diesen festgelegten Änderungsbetrag lautet **Inkrement**. Durch die Änderung der Zahl zwischen den runden Klammern von `noise()` erhalten wir bei jedem Durchlauf auch eine neue Zufallszahl zwischen 0 und 1. Benutzen wir diese Zahlenwerte, multipliziert mit 200, um im Sketch von Abbildung 6.10 die Länge von Linien zu bestimmen, so sehen wir, dass hier keine willkürlichen Zahlen generiert werden, sondern die Zufallszahlen zueinander in Beziehung stehen. Verbinden wir in Gedanken die Linienenden miteinander, so erhalten wir einen recht geglätteten Kurvenverlauf. Ersetzen wir die Zeilen 11 und 12 durch `random(200)`, so erhalten wir ein ganz anderes Bild (Abb. 6.11), da die so generierten Zufallszahlen nicht miteinander in Beziehung stehen.



```
1 let x = 0;
2 let y = 0;
3 let i = 0;
4
5 function setup() {
6   createCanvas(600, 200);
7   background(150);
8 }
9
10 function draw() {
11   y = 200 * noise(i);
12   i += 0.03; // Inkrement (Betrag der Änderung)
13   x += 10;
14
15   stroke(255);
16   strokeWeight(5);
17   line(x, y, x, height);
18 }
19
```

Abbildung 6.10: Mit der Funktion `noise()` und einem Inkrement von 0.03 erzeugte Linienhöhe

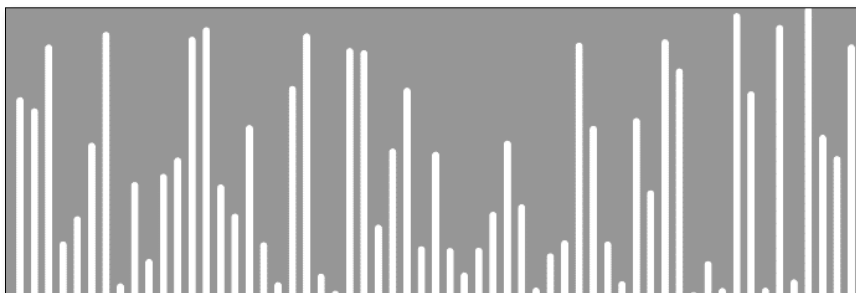


Abbildung 6.11: Mit der Funktion `random()` erzeugte Linienhöhe

Je kleiner das Inkrement, desto geringer ist der Unterschied zwischen den benachbarten Zufallswerten (siehe Abb. 6.12). Große Inkremente erzeugen relativ große Unterschiede zwischen benachbarten Zufallswerten. Die Zufallswerte bewegen sich jedoch stets zwischen 0 und 1, egal wie groß die Zahl zwischen den Klammern ist.

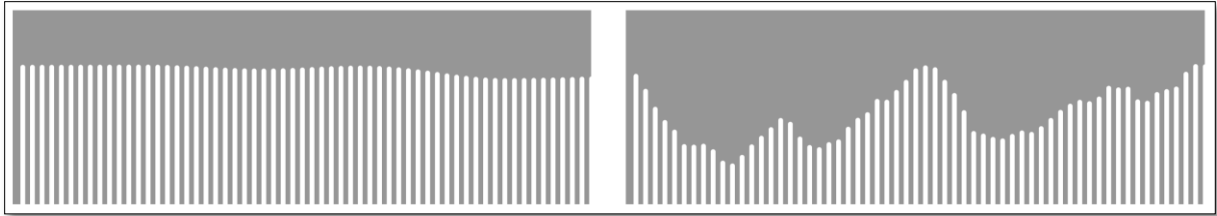


Abbildung 6.12: Die Abbildung links wurde mit einem Inkrement von 0,01 und die Abbildung rechts mit einem Inkrement von 0,1 erstellt

Verwenden wir nun das so erworbene Wissen, um unserer Kerzenflamme eine relativ natürliche Bewegung zu verleihen. Hierzu sind nur geringe Änderungen im Sketch von Abbildung 6.9 erforderlich. Wir fügen eine neue Variable mit dem Namen *i* ein und ersetzen die letzte Zeile des Sketches von Abbildung 6.9 durch *w = 5*noise(i);* und *i += 0.03;* (siehe Zeile 27 und 28 von dem Sketch in Abbildung 6.13). So erhalten wir eine Kerzenflamme, die sich nahezu natürlich bewegt.

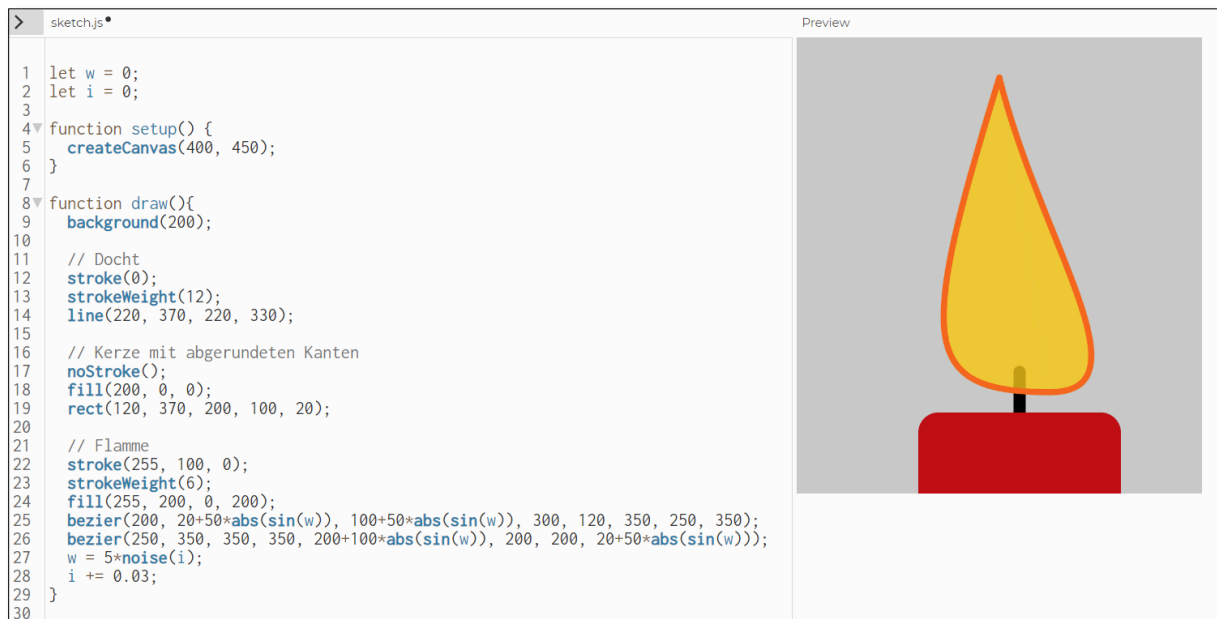


Abbildung 6.13: Mit der Funktion *noise()* erzeugte Bewegung der Kerzenflamme

Für die Zeichnung und Animation der Kerzenflamme von Abbildung 6.13 haben wir zwei, mittels der Funktion *bezier()* erzeugte Kurven zusammengesetzt und diese anschließend mit der Funktion *noise()* animiert. Aber auch mit nur einer Bézierkurve gelangen in Kombination mit Perlin noise interessante Bilder. In dem Sketch von Abbildung 6.14 haben wir für die Anker- und Kontrollpunkte der Bézierkurve die folgenden Variablen eingesetzt (Zeile 25).

`bezier(a1x, a1y, k1x, k1y, k2x, k2y, a2x, a2y);`

Der Buchstabe *a* steht hier für Ankerpunkt und *k* für Kontrollpunkt. Damit nicht alle *x*- und *y*-Werte der Anker- und Kontrollpunkte mit der gleichen, von der Funktion *noise()* generierten Zufallszahl multipliziert werden, haben wir zu der Variablen *w* jeweils einen unterschiedlichen Wert hinzuaddiert (Zeile

14 bis 21). Das Inkrement der Variablen w beträgt bei jedem Durchlauf von *function draw()* 0,03. Um ein ansprechendes Bild zu erhalten, müssen wir die maximale Größe der Variable w begrenzen. Dies gelingt mit einer *if()*-Funktion und der Funktion *noLoop()*. Die Zeilen 27 bis 29 können wie folgt gelesen werden: „Wenn der Wert von w größer 40 ist, dann beende den fortlaufenden Durchlauf von *function draw()*.“ *Loop* heißt übersetzt Schleife und *if* heißt übersetzt wenn.

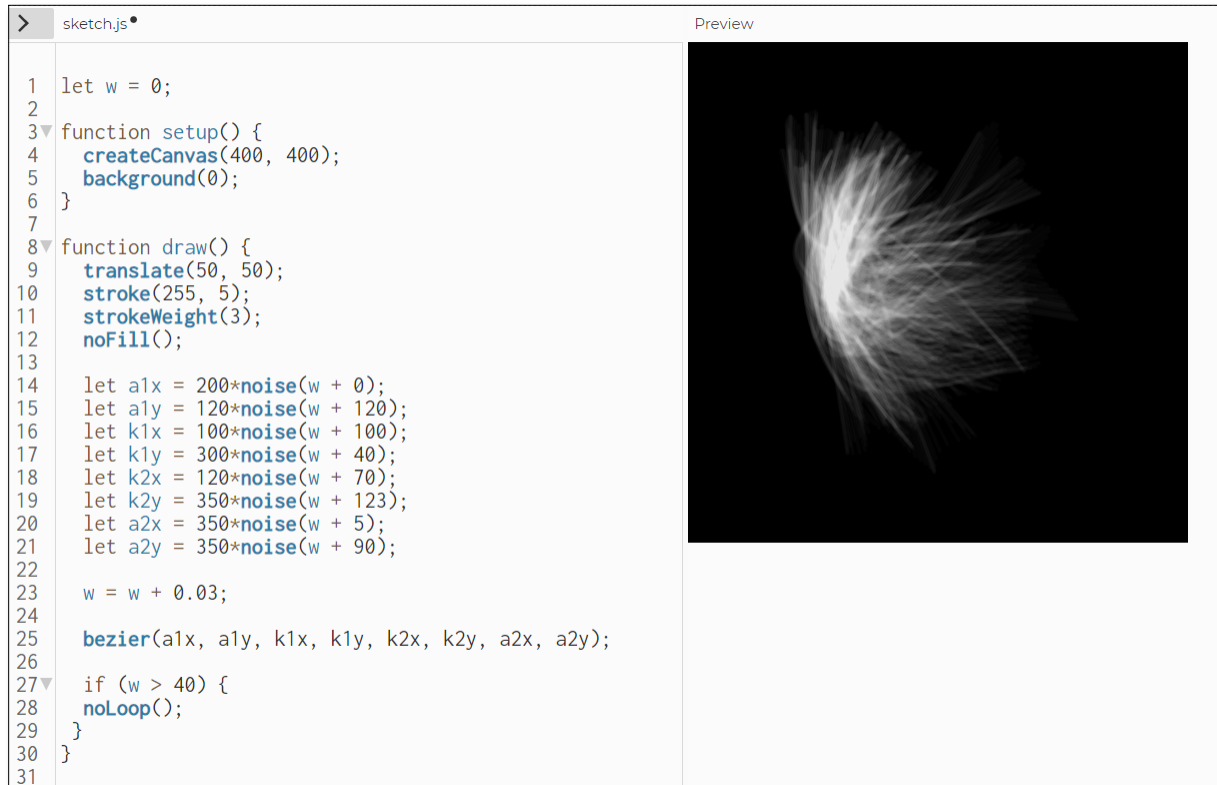


Abbildung 6.14: Sketch für die Zeichnung einer Waldfee

Das mit dem Sketch von Abbildung 6.14 erzeugte Bild kann man unterschiedlich deuten. Ich habe es einfach Waldfee genannt. Bei jedem neuen Start des Sketches von Abbildung 6.14 entstehen ähnliche, aber niemals die gleichen Bilder von der Waldfee (siehe Abb. 6.15).

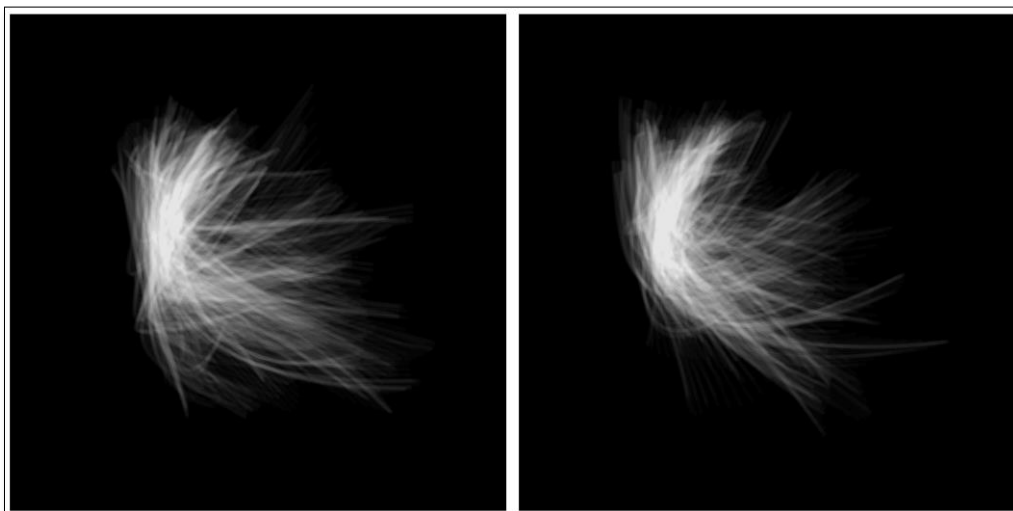


Abbildung 6.15: Der Sketch von Abbildung 6.14 erzeugte stets ähnliche, aber doch unterschiedliche Bilder

Verändert man den Sketch von Abbildung 6.13 nur geringfügig, so ergibt sich ein ganz anderes Bild (siehe Abb. 6.15). Perlin noise ist ein Eldorado für kreative Spielereien und kreatives Gestalten.

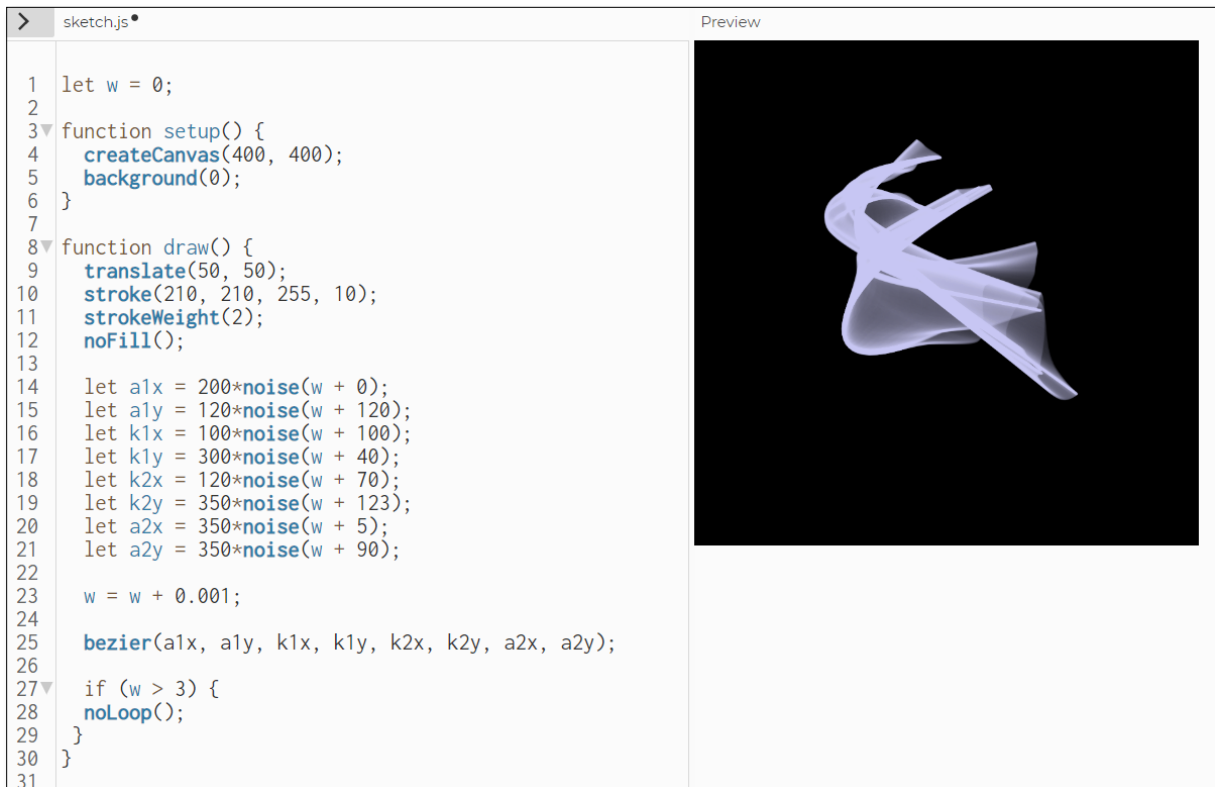


Abbildung 6.16: Seidentuch

Sieht Abbildung 6.16 eher wie ein Seidentuch aus, so kann man die Zeichnungen von Abbildung 6.17 als Tänzerinnen deuten. Auch sie wurden mit dem Sketch von Abbildung 6.16 erzeugt.

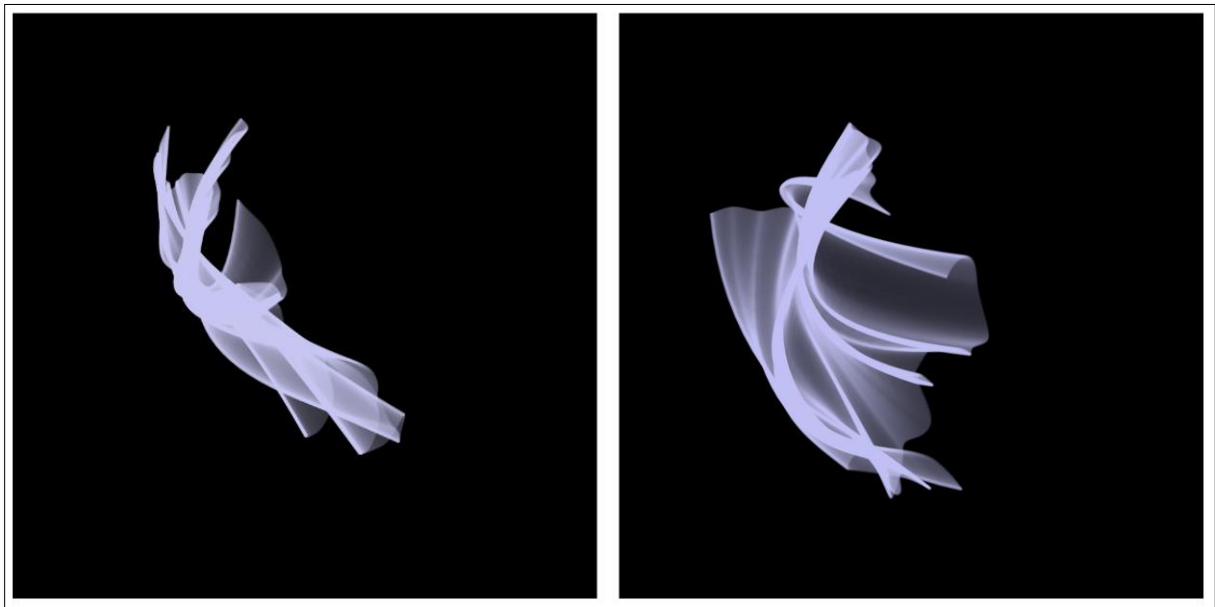


Abbildung 6.17: Tänzerinnen

6.3 Perlin noise 2D

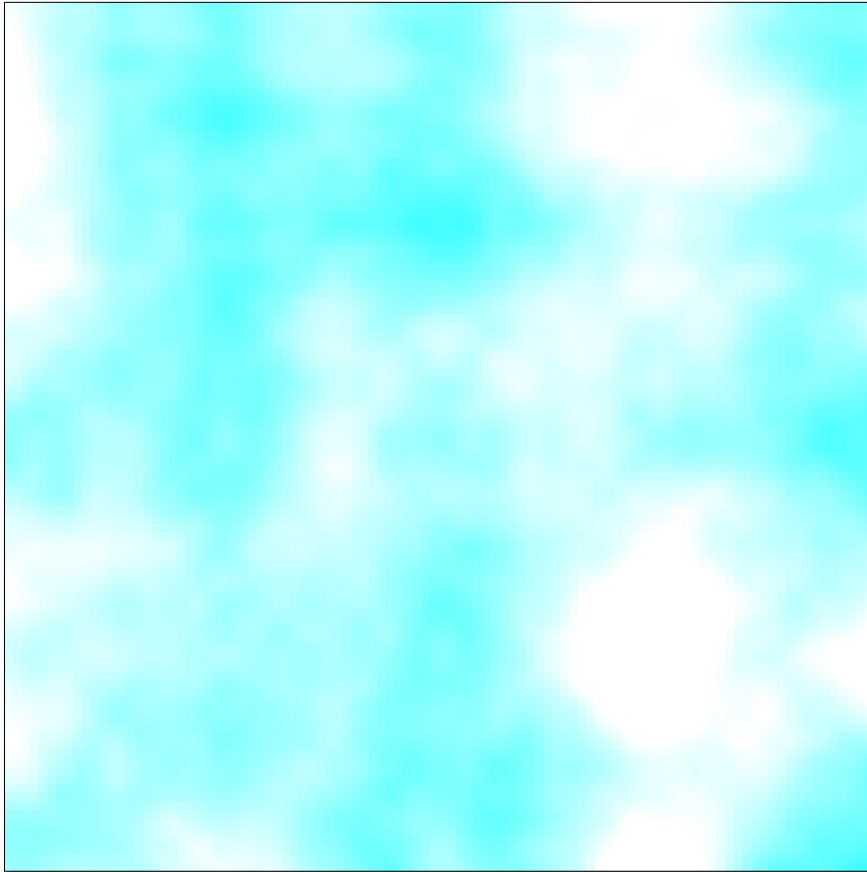


Abbildung 6.18: Wolken

Wie am Anfang von Kapitel 6.2 erwähnt, kann man mit Perlin noise Zufallswerte erzeugen, mit den man nicht nur schöne, sondern auch relativ natürliche Bilder zeichnen kann. Abbildung 6.18 ist ein Beispiel hierfür und Abbildung 6.19 zeigt den zugehörigen Sketch.

```
1 function setup() {  
2   createCanvas(400, 400);  
3   noStroke();  
4 }  
5  
6 function draw() {  
7   for (let y = 0; y < width; y += 4) {  
8     for (let x = 0; x < height; x += 4) {  
9       let r = 400 * noise(0.01 * x, 0.01 * y);  
10      fill(r, 255, 255);  
11      rect(x, y, 4, 4);  
12    }  
13  }  
14 }  
15
```

Abbildung 6.19: Sketch für die Erzeugung von Abbildung 6.18

Der Sketch ist recht kurz und enthält bis auf die Zeile 9 nur bekannte Elemente. Die Zeile 9 muss jedoch näher erklärt werden. Zeile 9:

```
let r = 400 * noise(0.01 * x, 0.01 * y);
```

Wir sehen hier zum ersten Mal, dass die `noise()`-Funktion zwei, durch ein Komma getrennte Elemente enthält. Zum besseren Verständnis vergrößern wir die Rechtecke von Sketch 6.19 um den Faktor 10 (siehe Abb. 6.20).

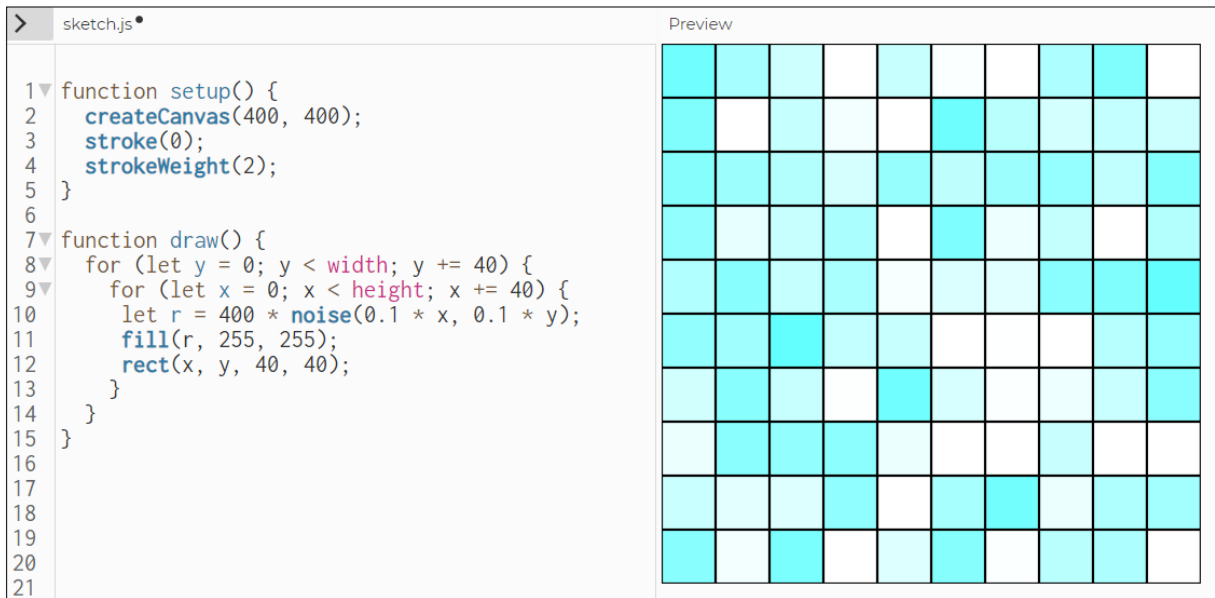


Abbildung 6.20: Wolkenbild mit großen Rechtecken

Entfernen wir nun `y` aus der runden Klammer der `noise`-Funktion (jetzt Zeile 10), so erhalten wir die Abbildung 6.21. Wir sehen, dass sich die Farben der Kästchen ändern, die in einer Zeile liegen. Die Farbe der Kästchen, die in einer Spalte liegen bleiben jedoch gleich.

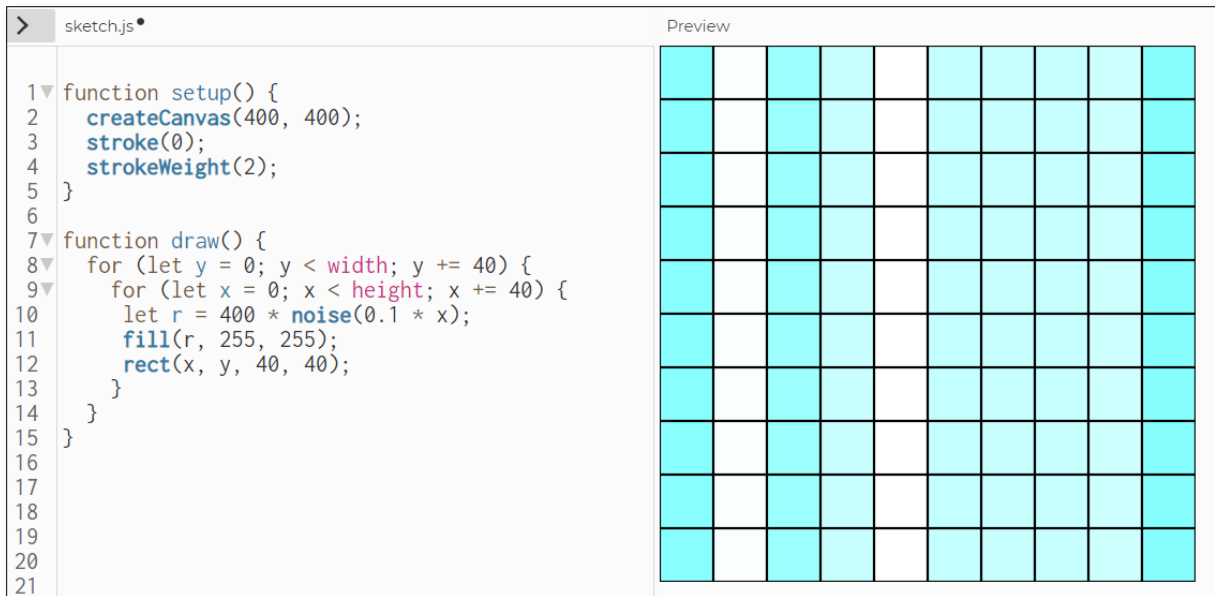


Abbildung 6.21: Die Farben in einer Zeile ändern sich. Die Farben in den jeweiligen Spalten bleiben gleich

Fügen wir nun `y` wieder ein und entfernen stattdessen `x` aus der runden Klammer der `noise`-Funktion, so erhalten wir die Abbildung 6.22. Wir sehen, dass sich die Farben der Kästchen ändern, die in einer Spalte liegen. Die Farbe der Kästchen, die in einer Zeile liegen bleiben jedoch gleich.

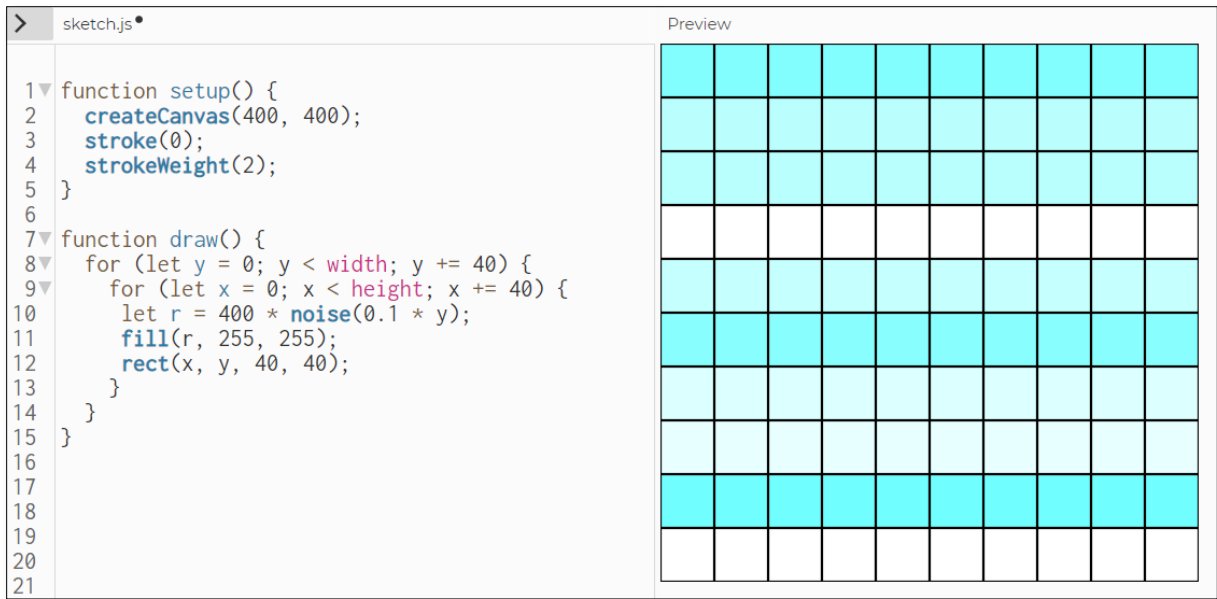


Abbildung 6.22: Die Farben in einer Spalte ändern sich. Die Farben in den jeweiligen Zeilen bleiben gleich

Fazit: Das erste Element in der runden Klammer von *noise()* sorgt für die Farbänderungen in x-Richtung und das zweite Element sorgt für Farbänderungen in y-Richtung. Nur wenn beide zusammenwirken erhalten wir das Bild von Abbildung 6.20. Voraussetzung hierfür ist jedoch, dass die beiden for-Schleifen bei jedem Durchlauf von *function draw()* für sich ändernde x- und y-Werte sorgen. Nur so werden fortlaufend neue Zufallszahlen generiert. Wählen wir zudem noch kleine Rechtecke und kleine Inkrement-Werte, dann erhalten wir das Wolkenbild von Abbildung 6.19.

Bei Perlin noise 1D beeinflussen der vorhergehende und der nachfolgende Wert den Zufallswert des dazwischen liegenden Wertes (siehe Abb. 6.12). Bei Perlin noise 2D beeinflussen 8 benachbarte Werte den von ihnen eingeschlossenen Wert (siehe Abb. 6.23).

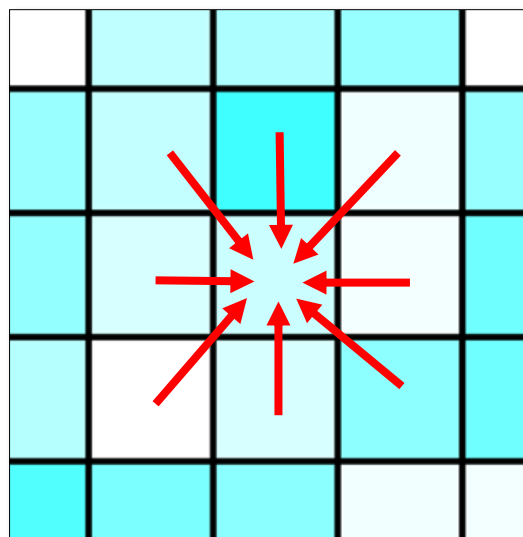


Abbildung 6.23: Die 8 benachbarten Werte beeinflussen den Zufallswert

Die beiden folgenden Abbildungen zeigen weitere Beispiele für die Anwendung von Perlin noise 2D.

```

> sketch.js • Preview
1 function setup() {
2   createCanvas(400, 400);
3 }
4
5 function draw() {
6   noStroke();
7   colorMode(HSB, 360, 100, 100);
8   for (let y = 0; y < width; y += 5) {
9     for (let x = 0; x < height; x += 5) {
10      let h = 180 * noise(0.002 * x, 0.002 * y);
11      fill(h, 100, 100);
12      rect(x, y, 5, 5);
13    }
14  }
15 }
16
17
18
19
20
21

```

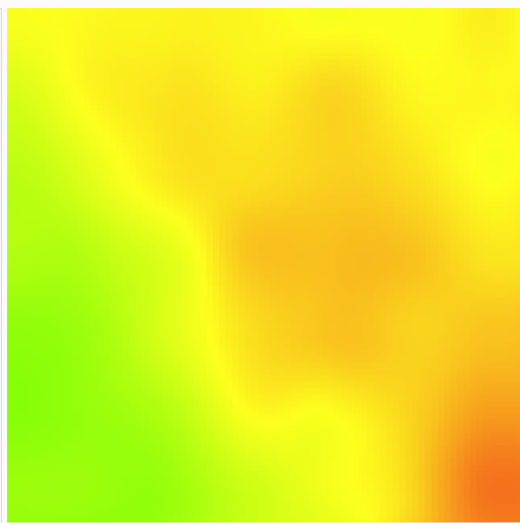


Abbildung 6.24: Perlin noise 2D im HSB-Farbraum 01

```

> sketch.js • Preview
1 function setup() {
2   createCanvas(400, 400);
3 }
4
5 function draw() {
6   noStroke();
7   colorMode(HSB, 360, 100, 100);
8   for (let y = 0; y < width; y += 5) {
9     for (let x = 0; x < height; x += 5) {
10      let h = 360 * noise(0.002 * x, 0.002 * y);
11      fill(h, 50, 100);
12      rect(x, y, 5, 5);
13    }
14  }
15 }
16
17
18
19
20
21

```

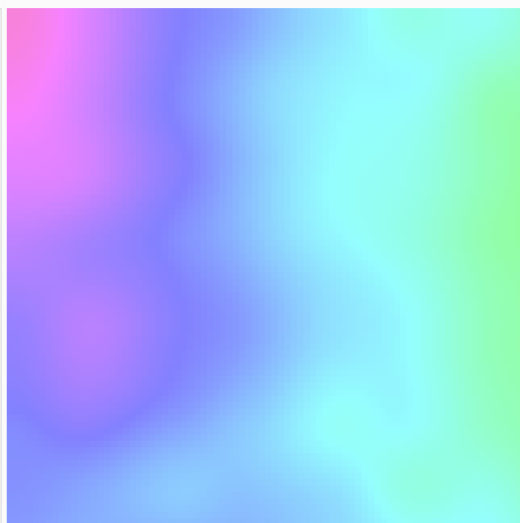


Abbildung 6.25: Perlin noise 2D im HSB-Farbraum 02

Gibt es auch eine dritte Dimension bei Perlin noise? Ja, die gibt es. Dazu muss man ein drittes Element in die runde Klammer bei `noise()` einfügen. Häufig wird die Zeit als dritte Dimension verwendet, um so eine animierte Darstellung zu erhalten. Ersetzen wir die Zeile 9 im Sketch von Abbildung 6.19 zum Beispiel durch die Zeile in der folgenden Klammer (`let r = 400 * noise(0.01 * x, 0.01 * y, second())`), so erhalten wir jede Sekunde ein neues Wolkenbild. Die Funktion `second()` steht in Verbindung mit der Computeruhr und gibt den aktuellen Sekundenwert an. Dadurch erzeugt die Funktion `noise()` jede Sekunde einen neuen Zufallswert.

6.4 Zusammenfassung

Fassen wir zusammen, was wir in Kapitel 6 gelernt haben, bzw. gelernt haben sollten.

- bezier()** Mit der Funktion *bezier()* lassen sich elegante Kurven zeichnen. Sie benötigt vier Zahlenpaare, *bezier(ax1, ay1, kx1, ky1, kx2, ky2, ax2, ay2)*. Zwei für die beiden Ankerpunkte (erstes und letztes Zahlenpaar) und zwei für die beiden Kontrollpunkte (die beiden mittleren Zahlenpaare). Die Verbindungslinie zwischen einem Ankerpunkt und den zugehörigen Kontrollpunkt stellt eine Tangente dar, die am jeweiligen Ankerpunkt an der Kurve anliegt. Die Kontrollpunkte und die Tangenten werden in der Regel nicht gezeichnet.
- rect(x, y, Breite, Höhe, Rundung)** Mit einer fünften Zahl bei der Funktion *rect()* kann man die Stärke der Eckrundung bestimmen. Je größer die fünfte Zahl, desto ausgeprägter ist die Rundung.
- translate()** Der Ursprung des Koordinatensystems bei p5.js liegt in der linken oberen Ecke der Leinwand. Mit der Funktion *translate(x, y)* kann man diesen Ursprung in x-Richtung und y-Richtung verschieben.
- bezierVertex()** Mit der Funktion *bezierVertex()* können wir Anker- und Kontrollpunkte für jeweils eine Bézierkurve festlegen. Will man eine Figur zeichnen, die aus mehreren Kurven besteht, benötigt man auch mehrere *bezierVertex()*-Funktionen. Alle diese *bezierVertex()*-Funktionen müssen hierbei zwischen *beginShape()* und *endShape()* stehen. Weiterhin muss der erste Ankerpunkt mit der Funktion *vertex()* festgelegt werden. In dem unten aufgeführten Beispiel sind die fünf Ankerpunkte fett dargestellt. Die beiden anderen, nicht fett dargestellten, Wertepaare zwischen den runden Klammern von *bezierVertex()* geben die Kontrollpunkte an.
- ```
beginShape();
 vertex(0, 100);
 bezierVertex(-10, 250, -100, 200, -150, 320);
 bezierVertex(-100, 350, -100, 400, 0, 400);
 bezierVertex(100, 400, 100, 350, 150, 320);
 bezierVertex(100, 200, 10, 250, 0, 100);
endShape();
```
- sin()** und **cos()** Die Funktionen *sin()* und *cos()* eignen sich gut dazu, um periodische Vorgänge zu erzeugen. Das Argument in der Klammer wird im Bogenmaß angegeben.
- abs()** Mit der Funktion *abs()* kann man negative Werte in positive verwandeln.
- noise()** Durch die Änderung der Zahl zwischen den runden Klammern von *noise()* erhalten wir eine neue Zufallszahl zwischen 0 und 1. Große Zahlenänderungen zwischen den runden Klammern erzeugen auch relativ große Unterschiede zwischen benachbarten Zufallswerten. Die Zufallswerte bewegen sich jedoch stets zwischen 0 und 1. Diese Zufallszahlen sind aber nicht so zufällig wie die Zahlen, die man mit der Funktion *random()* erzeugt. Die mit *noise()* erzeugten

Zufallszahlen stehen in einem Bezug zueinander. Dies sorgt für natürlich aussehende Muster.

Wenn die Funktion *noise()* zwei durch ein Komma getrennte Zahlenwerte enthält, dann liefert *noise()* Zufallszahlen in x- und y-Richtung.

**second()**

Die Funktion *second()* steht in Verbindung mit der Computeruhr und gibt den aktuellen Sekundenwert an.

## 6.5 Übungen

Aufgabe 1 Wie wirkt sich bei gleicher Tangentensteigung der Abstand der Kontrollpunkte von den Ankerpunkten auf den Bogen aus?

Aufgabe 2 Zeichne mit zwei Bézierkurven das Zeichen für Unendlich (siehe Abb. 6.26).

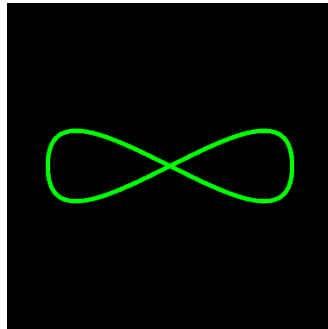


Abbildung 6.26: Das Zeichen für Unendlich

Aufgabe 3 Erzeuge durch die Überlagerung von mehreren Bézierkurven ein Bild, das der folgenden Abbildung stark ähnelt.

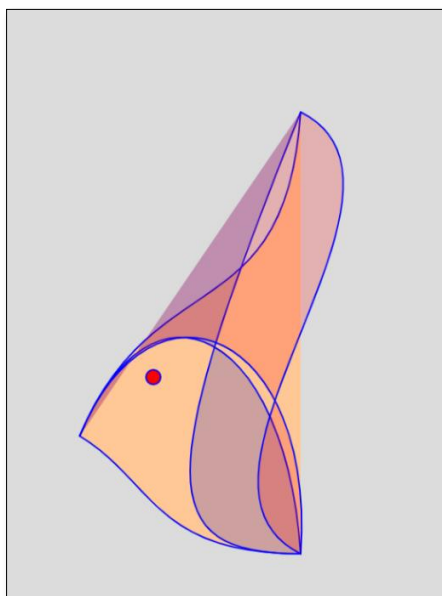


Abbildung 6.27: Paradiesvogel

Aufgabe 4 Erstelle mittels Bézierkurven und Perlin noise die folgende Abbildung.

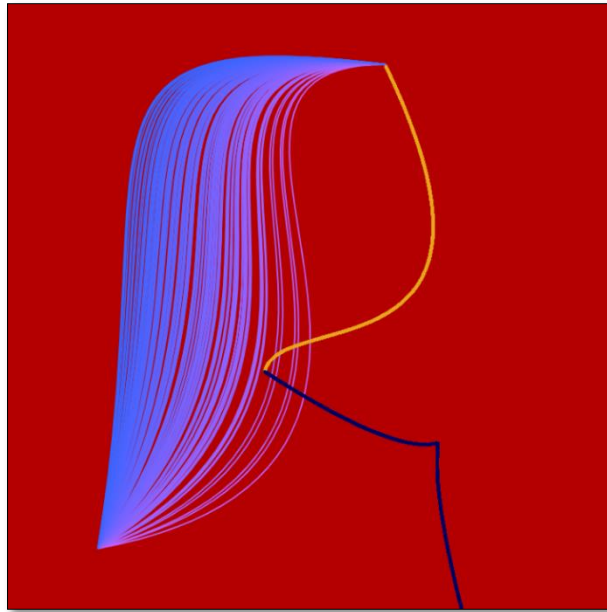


Abbildung 6.28: Mädchen mit blauem Haar

- Aufgabe 5
- Zeichne mit einer Bézierkurve einen roten Luftballon vor grauem Hintergrund, der an einer Schnur befestigt ist.
  - Der Luftballon soll sich im „Wind“ leicht hin und her bewegen. Tipp: Sketch Kerze
  - Vor dem bewölkten Himmel von Abbildung 6.18 soll sich der Luftballon im „Wind“ leicht hin und her bewegen.
  - Der Luftballon, die Schnur und die Wolken sollen sich bewegen.

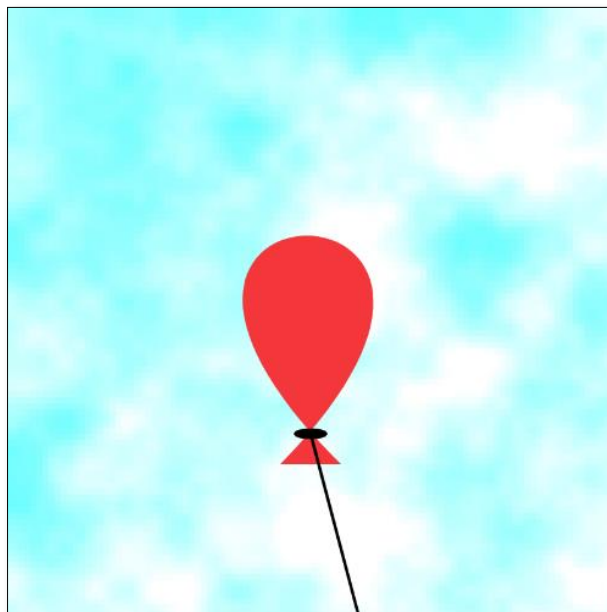


Abbildung 6.29: Sich im Wind wiegender Ballon



## 7 Eigene Funktionen

### 7.1 Smileys

Bisher haben wir schon zahlreiche Funktionen wie *ellipse()*, *rect()*, *line()*, ... kennengelernt. Auf der Homepage von p5.js findet man in der Referenz ( <https://p5js.org/reference/> ) aber mehr Funktionen, als wir hier in diesem Buch besprechen können. Für das eigene kreative Arbeiten lohnt es auf jeden Fall, sich diese Funktionen und ihre Beispiele anzusehen.

Unter p5.js ist es aber auch möglich eigene Funktionen zu programmieren. Wie sinnvoll und nützlich dies ist, soll an den Beispielen der selbsterstellten Funktionen *smileys()* und *sechseck()* gezeigt werden.

Nehmen wir einmal an, wir wollen auf einer Leinwand mehrere Smileys zeichnen. Nachdem wir den Code für den ersten Smiley geschrieben haben (Zeile 7 – 15 in Abbildung 7.1) können wir diesen Codeabschnitt kopieren und in unseren Sketch mit veränderten Werten wieder einfügen (Zeile 17 – 25). Wenn wir nun 6 oder mehr Smileys zeichnen wollen, dann wird der Sketch sehr lang. Kürzer und eleganter ist es, wenn wir für unsere Smileys eine **eigene Funktion** mit dem Namen **function smileys(x, y)** erstellen. Siehe Zeile 11 – 21 in Abbildung 7.2. Die Variablen x und y werden bei *function smileys(x, y)* nur deklariert. Initialisiert werden sie erst bei *function draw()*. Hier wird die Funktion *function smileys()* zweimal aufgerufen und es werden für x und y konkrete Werte eingesetzt. Will man 6 oder mehr Smileys, dann muss man die Funktion *function smileys()* mehrfach bei *function draw()* aufrufen oder man verwendet vorteilhafter eine for-Schleife (Abb. 7.3, Zeile 7 – 9). Will man noch mehr Smileys, so kann man mit zwei verschachtelten for-Schleifen ganz bequem die ganze Leinwand füllen (Abb. 7.4). Obwohl wir nun 18 Smileys gemalt haben, ist der Sketch von Abbildung 7.4 kürzer als der Sketch von Abbildung 7.1, mit dem wir nur zwei Smileys gemalt haben.

Ohne den Sketch von Abbildung 7.4 zu verlängern, kommen wir durch geringfügige Änderungen innerhalb der Funktion *function smileys()* zu neuen interessanten Ergebnissen (siehe Abbildung 7.5). Wer dies alles verstanden hat, für den stellt die Aufgabe 1 im Kapitel 7.7 kein wirkliches Problem dar.



Abbildung 7.1: Sketch für zwei Smileys



Abbildung 7.2: Zwei Smileys wurden mithilfe einer selbsterstellten Funktion gezeichnet



Abbildung 7.3: Sechs Smileys wurden mit Hilfe einer selbsterstellten Funktion und einer for-Schleife gezeichnet

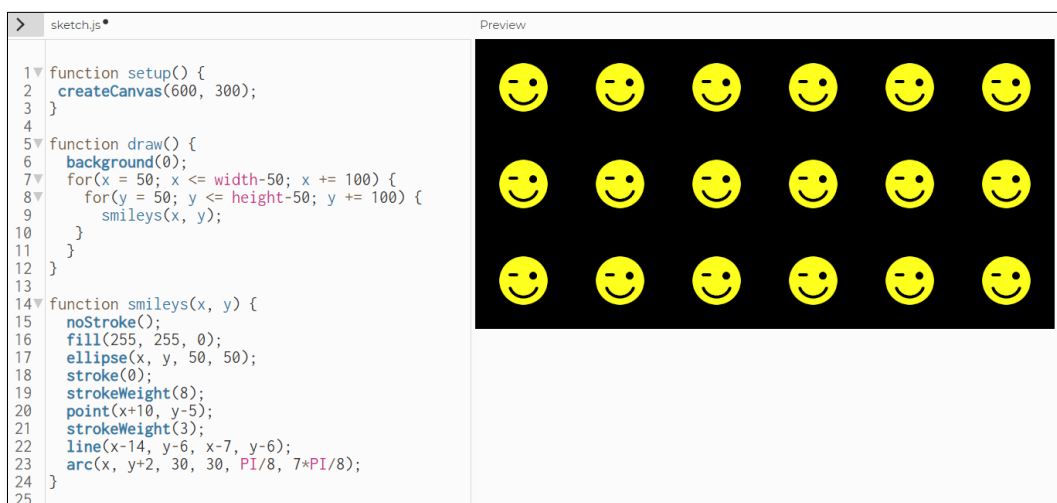


Abbildung 7.4: Die Leinwand wurde mit Hilfe einer doppelten for-Schleife mit Smileys gefüllt

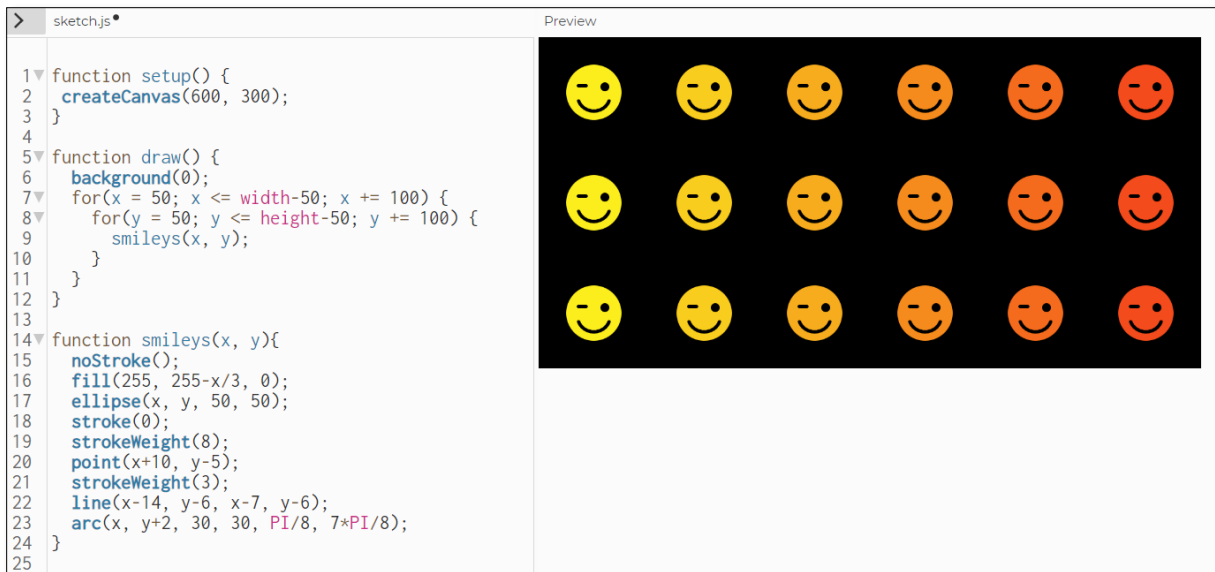


Abbildung 7.5: Smileys in unterschiedlichen Farben

## 7.2 Bienenwabe

Schauen wir uns nun außer der Funktion *function smileys()* noch ein weiteres Beispiel für eine selbst-erstellte Funktion an. Funktionen für das Zeichnen eines Kreises, eines Rechteckes, eines Dreieckes, usw. stellt p5.js uns selbst zur Verfügung. Wenn wir aber ein Wabenmuster wie in Abbildung 7.6 zeichnen möchten, dann benötigen wir die Funktion für ein regelmäßiges Sechseck. In der p5.js-Bibliothek finden wir hierzu jedoch keine Funktion. D.h., wir müssen sie selbst programmieren.

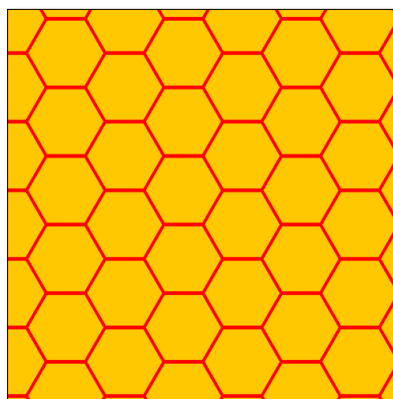


Abbildung 7.6: Bienenwabe

Um das obige Wabenmuster zu zeichnen, ist es sinnvoll, wenn wir zuerst eine eigene Funktion mit dem Namen *function sechseck()* anfertigen. Mit ihr können wir dann unser Wabenmuster erstellen. Abbildung 7.8 zeigte von Zeile 21 bis 30 eine mögliche Lösung für die Funktion *function sechseck()*. Bevor man einen solchen Code schreibt, sollte man, um Frustrationen durch erfolgloses Herumprobieren zu vermeiden, im Vorfeld mit Bleistift und Papier überlegen, wie man ein regelmäßiges Sechseck richtig

konstruiert. Dabei helfen die Winkelfunktionen Sinus und Cosinus und das Wissen, dass man sich ein Sechseck aus sechs zusammengesetzten gleichseitigen Dreiecken mit Winkeln von je  $60^\circ$  vorstellen kann (siehe Abb. 7.7).

Die Eckpunkte eines regelmäßigen Sechsecks liegen alle auf einem Kreis mit dem Durchmesser  $d$  (siehe Abb. 7.7). Den Mittelpunkt dieses Kreises bezeichnen wir mit  $x$  und  $y$ . Mit den Pixelwerten für  $x$  und  $y$  legen wir die Position des Sechsecks auf der Leinwand fest. Für den rechten Eckpunkt des Sechsecks gilt dann  $x + d, y$ . Für den Punkt oben rechts gilt  $x + d \cdot \cos 60^\circ, y - d \cdot \sin 60^\circ$ . Für den linken oberen Eckpunkt gilt  $x - d \cdot \cos 60^\circ, y - d \cdot \sin 60^\circ$ , usw. (siehe Zeile 22 bis 29 in Abb. 7.8).

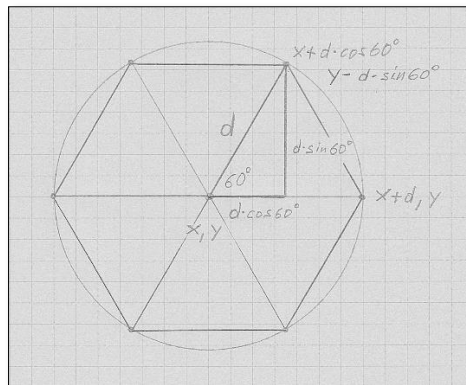


Abbildung 7.7: Mit Bleistift und Papier erstellte Konstruktionsskizze für ein regelmäßiges Sechseck

Im Sketch von Abbildung 7.8 sehen wir, dass die Variablen  $x$ ,  $y$  und  $d$  zwischen den runden Klammern von *function sechseck(x, y, d)* deklariert, aber noch nicht initialisiert werden. Dies geschieht, wie schon in Kapitel 7.1 erwähnt, erst bei *function draw()*. Bei *function draw()* wird auch die Funktion *function sechseck(x, y, d)* aufgerufen, um mit den gewählten Werten das Sechseck zu zeichnen. Neu in diesem Sketch ist die Funktion **angleMode(DEGREES)** bei *function setup()*. Sie sorgt dafür, dass wir die Winkel bequem in Grad angeben können und nicht im Bogenmaß angeben müssen.

```

sketchjs* Preview
1 function setup() {
2 createCanvas(400, 400);
3 angleMode(DEGREES); // Winkel können ab hier in Grad angegeben werden.
4 }
5
6 function draw() {
7 background(220);
8 stroke(255, 0, 0);
9 strokeWeight(3);
10 fill(255, 200, 0);
11 // Nun werden die Variablen x, y und d initialisiert.
12 x = 200;
13 y = 200;
14 d = 100;
15 // Die Funktion sechseck(x, y, d) wird aufgerufen.
16 sechseck(x, y, d);
17 }
18
19 // Die Funktion für die Zeichnung von Sechsecken wird erstellt
20 // und die Variablen x, y und d werden deklariert.
21 function sechseck(x, y, d) {
22 beginShape()
23 vertex(x+d, y);
24 vertex(x+d*cos(60), y-d*sin(60));
25 vertex(x-d*cos(60), y-d*sin(60));
26 vertex(x-d, y);
27 vertex(x-d*cos(60), y+d*sin(60));
28 vertex(x+d*cos(60), y+d*sin(60));
29 endShape(CLOSE);
30 }
31

```

Abbildung 7.8: Mit der selbsterstellten Funktion *function sechseck(x, y, d)* gezeichnetes regelmäßiges Sechseck

Nachdem wir die Funktion *function sechseck()* programmiert haben, können wir nun unter Verwendung von zwei doppelten for-Schleifen (Abb. 7.9) unser Wabenmuster von Abbildung 7.6 erstellen.

```

1 function setup() {
2 createCanvas(400, 400);
3 angleMode(DEGREES);
4 }
5
6 function draw() {
7 background(220);
8 stroke(255, 0, 0);
9 strokeWeight(3);
10 fill(255,200, 0);
11
12 for(let x = 0; x <= width+width/4; x = x + 120) {
13 for(let y = 10; y <= height+height/4; y = y + 70) {
14 d = 40;
15 sechseck(x, y, d);
16 }
17 }
18
19 for(let x = 60; x <= width+width/4; x = x + 120) {
20 for(let y = -95; y <= height+height/4; y = y + 70) {
21 d = 40;
22 sechseck(x, y, d);
23 }
24 }
25 }
26
27 function sechseck(x, y, d) {
28 beginShape()
29 vertex(x+d, y);
30 vertex(x+d*cos(60), y-d*sin(60));
31 vertex(x+d*cos(60), y-d*sin(60));
32 vertex(x-d*cos(60), y-d*sin(60));
33 vertex(x-d, y);
34 vertex(x-d*cos(60), y+d*sin(60));
35 vertex(x+d*cos(60), y+d*sin(60));
36 endShape(CLOSE);
37 }
38

```

Abbildung 7.9: Sketch für die Erstellung des Bienenwabenmusters von Abbildung 7.6

### 7.3 Herzballon

Als weiteres Beispiel für eine eigene Funktion wollen wir eine Funktion mit dem Namen *function herz()* programmieren. Mit ihrer Hilfe wollen wir eine Animation erstellen, bei der ein herzförmiger Luftballon solange aufgeblasen wird bis er platzt. Dieser Vorgang soll sich fortlaufend wiederholen (siehe Abb. 7.10). Dazu benötigen wir zwei if-Anweisungen und unsere im Kapitel 2 erworbenen Kenntnisse über die Erzeugung von geschwungenen Linien mit Hilfe der Funktion *curveVertex()*.

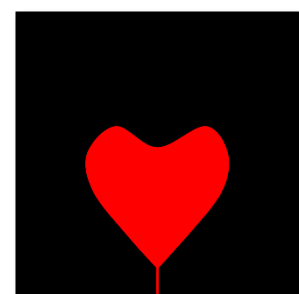


Abbildung 7.10: Herzballon wird aufgeblasen.

Schauen wir uns den hierzu gehörigen Sketch in Abbildung 7.11 im Einzelnen an. Die Funktion *function herz()* enthält außer den Variablen *x* und *y* zur Positionierung des Herzluftballons, noch die Variable *a*. Mit ihr kann die Größe des Herzluftballons verändert werden. Die mit *a* multiplizierten Werte erhält man durch nachdenken (Bleistift und Papier!) und probieren. Es erleichtert die Arbeit, wenn man im Web-Editor bei Auto-refresh ein Häkchen setzt, sodass man seine Änderungen sofort sehen kann.

```

1 // platzende Herzen
2
3 let i = 0;
4
5 function setup() {
6 createCanvas(400, 400);
7 }
8
9 function draw() {
10 background(0);
11 noStroke();
12 fill(255, 0, 0);
13 rect(198, height/1.11, 4, height);
14
15 if(i < 1.4) {
16 i = i + 1/width;
17 }
18
19 x = width/2;
20 y = height/1.1;
21 a = i;
22 herz(x, y, a);
23
24 if(i > 1.3) {
25 fill(255);
26 textSize(150);
27 textAlign(CENTER);
28 text("Peng!", width/2, height/2);
29 i = 0;
30 }
31 }
32
33 function herz(x, y) {
34 beginShape();
35 curveVertex(x, y, a);
36 curveVertex(x+a*45, y-a*50);
37 curveVertex(x+a*100, y-a*120);
38 curveVertex(x+a*110, y-a*180);
39 curveVertex(x+a*75, y-a*223);
40 curveVertex(x, y-a*190);
41 curveVertex(x-a*65, y-a*223);
42 curveVertex(x-a*110, y-a*180);
43 curveVertex(x-a*100, y-a*120);
44 curveVertex(x-a*45, y-a*50);
45 endShape(CLOSE);
46 }
47

```

Abbildung 7.11: Sketch für die Animation eines platzenden Herzluftballons.

Um die Größe des Herzluftballons kontinuierlich zu verändern erzeugen wie in Zeile 3 des Sketches die globale Variable *i*. Mit ihr und der ersten if-Anweisung gelingt es, den Herzluftballon wachsen zu lassen. Überschreitet der Ballon eine gewisse Größe, so sorgt die zweite if-Anweisung dafür, dass der Ballon platzt und die Animation von vorne beginnt.

## 7.4 Vier Kerzen

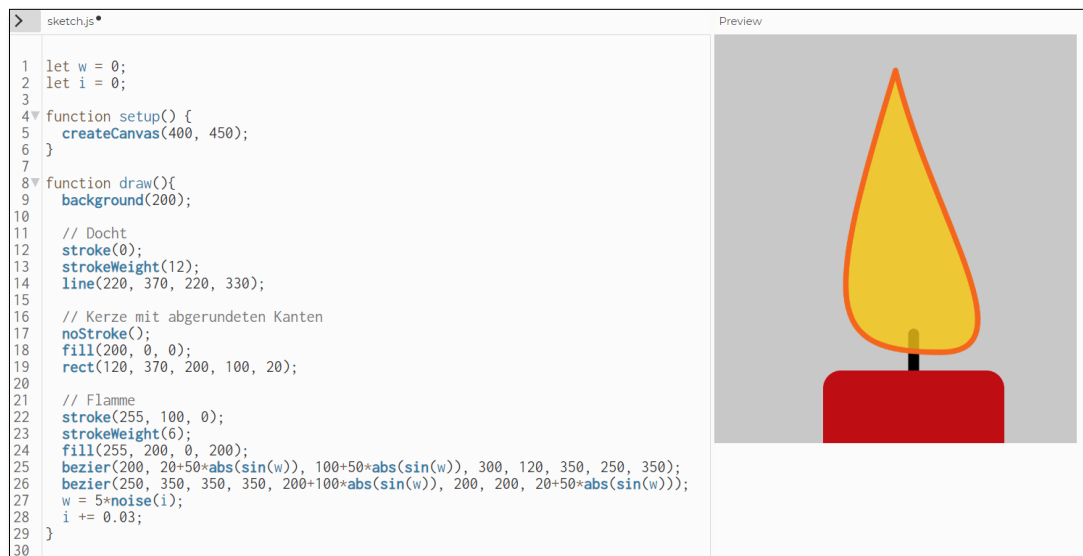


Abbildung 7.12: Erneuter Aufruf von Abbildung 6.13

Schauen wir uns nochmal den Sketch von der Kerze mit der bewegten Flamme aus Kapitel 6 an (siehe Abb. 7.12). Er erzeugt nur eine Kerze. In der Adventszeit benötigen wir jedoch vier Kerzen. Statt den nicht gerade einfachen Code mit seinen Anker- und Kontrollpunkten für die Bézierkurven viermal zu schreiben, sollten wir lieber eine Funktion *kerze* erstellen und diese Funktion viermal aufrufen. Bei jedem Aufruf können wir dann mit der Funktion *translate()* für jede neue Kerze das Koordinatensystem um jeweils 220 Pixel in x-Richtung verschieben. Für die erste Kerze würde dann die Verschiebung 0 Pixel, für die zweite Kerze 220 Pixel, für die dritte 440 Pixel und für die vierte 660 Pixel betragen. Der Abstand zwischen den Kerzenmittelpunkten würde also jeweils 220 Pixel betragen. Der Sketch von Abbildung 7.14 zur Zeichnung von Abbildung 7.13 liefert aber ein anderes Ergebnis. Zwischen Kerze 1 und 2 beträgt der Abstand der Kerzenmittelpunkte 220 Pixel. Zwischen 2 und 3 beträgt er aber 440 Pixel und zwischen 3 und 4 beträgt er 660 Pixel. Die Verschiebungen des Koordinatensystems mit *translate()* in x-Richtung haben sich einfach addiert. Wie können wir dies verhindern?

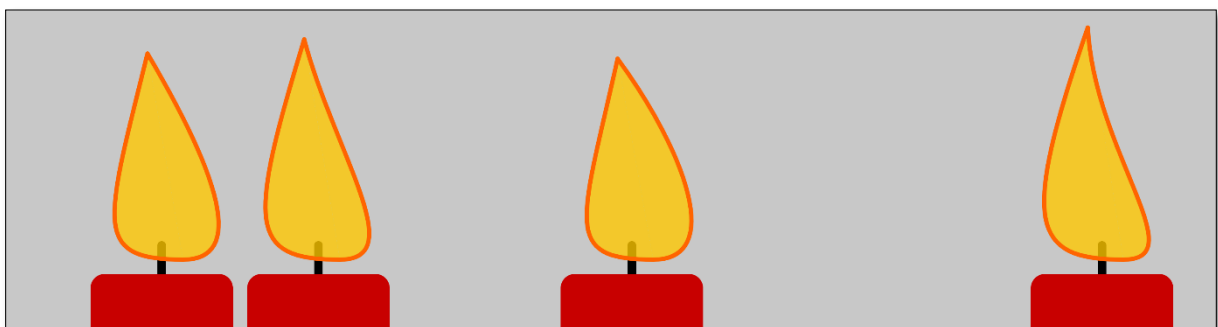


Abbildung 7.13: Vier ungleichmäßig verschobene Kerzen

Eine mögliche Lösung ist, dass wir anstatt der Zahlenkombination 0, 220, 440, 660, die Zahlenkombination 0, 220, 220, 220 schreiben. Auch hiermit bewirken wir infolge der Addition der Werte eine Verschiebung von 0, 220, 440 und 660 Pixel. Bei komplexeren Verschiebungen in x- und y-Richtung gelingt dies ohne Taschenrechner aber nicht mehr so einfach. Eine elegantere Lösung ist die Verwendung des Funktionenpaars *push()* und *pop()* (siehe Abb. 7.15). Die Funktion *push()* speichert

das aktuelle Koordinatensystem und die Funktion `pop()` lädt das gespeicherte Koordinatensystem wieder und vergisst alles, was zwischen `push()` und `pop()` passiert ist. Auf unsere Kerzen bezogen können wir uns dies wie folgt vorstellen.

`push()` speichert den Zustand des Ausgangskoordinatensystems.

Die erste Kerze wird gezeichnet.

`pop()` ruft das Ausgangskoordinatensystem wieder auf.

`push()` speichert den Zustand des Ausgangskoordinatensystems.

Das Koordinatensystem für die zweite Kerze wird um 220 Pixel in x-Richtung verschoben.

Die zweite Kerze wird gezeichnet.

`pop()` ruft das Ausgangskoordinatensystem wieder auf.

`push()` speichert den Zustand des Ausgangskoordinatensystems.

Das Koordinatensystem für die dritte Kerze wird um 440 Pixel in x-Richtung verschoben.

Die dritte Kerze wird gezeichnet.

`pop()` ruft das Ausgangskoordinatensystem wieder auf.

`push()` speichert den Zustand des Ausgangskoordinatensystems.

Das Koordinatensystem für die vierte Kerze wird um 660 Pixel in x-Richtung verschoben.

Die vierte Kerze wird gezeichnet.

`pop()` ruft das Ausgangskoordinatensystem wieder auf.

```
1 let i = 0;
2
3 function setup() {
4 createCanvas(1700, 450);
5 }
6
7 function draw() {
8 background(200);
9
10 // Änderung des Inkrements
11 w1 = 0.4*i;
12 w2 = 1.1*i;
13 w3 = 1.3*i;
14 w4 = 1.8*i;
15 i += 0.03;
16
17 /* Die Funktion kerze wird viermal aufgerufen
18 und den vier Kerzen werden ihre Werte zugeordnet. */
19 kerze(w1, 0, 0);
20 kerze(w2, 220, 0);
21 kerze(w3, 440, 0);
22 kerze(w4, 660, 0);
23 }
24
25 function kerze(w, x, y) {
26 translate(x, y);
27 // Docht
28 stroke(0);
29 strokeWeight(12);
30 line(220, 370, 220, 330);
31
32 // Kerze mit abgerundeten Kanten
33 noStroke();
34 fill(200, 0, 0);
35 rect(120, 370, 200, 100, 20);
36
37 // Flamme
38 stroke(255, 100, 0);
39 strokeWeight(6);
40 fill(255, 200, 0, 200);
41 bezier(200, 20+50*abs(sin(w)), 100+50*abs(sin(w)), 300, 120, 350, 250, 350);
42 bezier(250, 350, 350, 350, 200+100*abs(sin(w)), 200, 200, 20+50*abs(sin(w)));
43 }
44
```

Abbildung 7.14: Mit diesem Sketch wurde Abbildung 7.13 gezeichnet

```

1 let i = 0;
2
3 function setup() {
4 createCanvas(1700, 450);
5 }
6
7 function draw() {
8 background(200);
9
10 // Änderung des Inkrements
11 w1 = 0.4*i;
12 w2 = 1.1*i;
13 w3 = 1.3*i;
14 w4 = 1.8*i;
15 i += 0.03;
16
17 /* Die Funktion kerze wird viermal aufgerufen
18 und den vier Kerzen werden ihre Werte zugeordnet. */
19 kerze(w1, 0, 0);
20 kerze(w2, 220, 0);
21 kerze(w3, 440, 0);
22 kerze(w4, 660, 0);
23 }
24
25 function kerze(w, x, y) {
26 push();
27 translate(x, y);
28 // Docht
29 stroke(0);
30 strokeWeight(12);
31 line(220, 370, 220, 330);
32
33 // Kerze mit abgerundeten Kanten
34 noStroke();
35 fill(200, 0, 0);
36 rect(120, 370, 200, 100, 20);
37
38 // Flamme
39 stroke(255, 100, 0);
40 strokeWeight(6);
41 fill(255, 200, 0, 200);
42 bezier(200, 20+50*abs(sin(w)), 100+50*abs(sin(w)), 300, 120, 350, 250, 350);
43 bezier(250, 350, 350, 350, 200+100*abs(sin(w)), 200, 200, 20+50*abs(sin(w)));
44 pop();
45 }
46

```

Abbildung 7.15: Dieser Sketch enthält das Funktionenpaar `push()` und `pop()`. Mit ihm wurde die Abbildung 7.16 gezeichnet

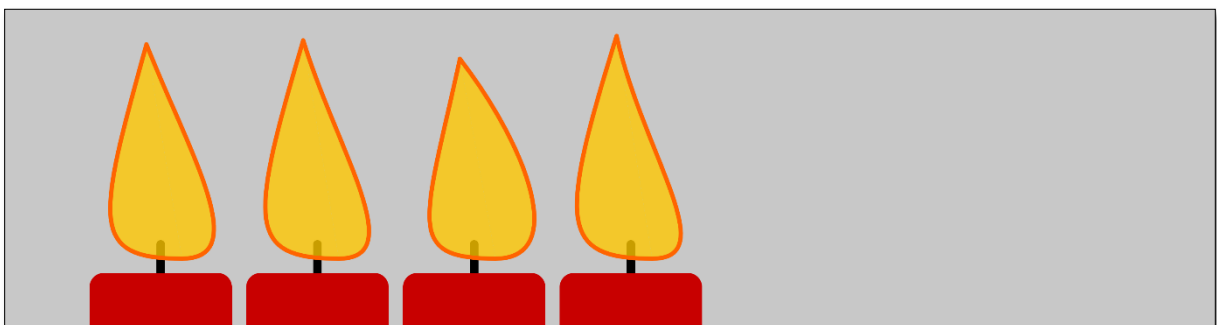


Abbildung 7.16: Mit dem Sketch von Abbildung 7.15 gezeichnete Kerzen

Da die Funktion `noise()` die Kerzenflamme nur dann in Bewegung hält, wenn sich das Inkrement zwischen den runden Klammern von `noise()` bei jedem Durchlauf von `function draw()` ändert, finden wir die Inkremente bei `function draw()` (Zeile 11 bis 15). Den jeweiligen Wert von `noise()` haben wir mit unterschiedlichen Zahlen multipliziert, damit sich die Kerzenflammen nicht alle gleich bewegen.

## 7.5 Mausmalprogramm



Abbildung 7.17: Mit dem Mausmalprogramm von Abbildung 7.18 erstelltes Kinderbild

```
1 let r = 0;
2 let g = 0;
3 let b = 0;
4
5 function setup() {
6 createCanvas(windowWidth, windowHeight);
7 background(240);
8 }
9
10 function draw() {
11 if (mouseIsPressed) {
12 noStroke();
13 fill(r,g,b);
14 ellipse(mouseX, mouseY, 10, 10);
15 }
16 }
17
18 function keyTyped() {
19 if (keyIsPressed && key == "r") {
20 r = 255;
21 }
22 if (keyIsPressed && key == "g") {
23 g = 255;
24 }
25 if (keyIsPressed && key == "b") {
26 b = 255;
27 }
28 }
29
30 function keyReleased() {
31 r = 0;
32 g = 0;
33 b = 0;
34 }
35
```

Abbildung 7.18: Mausmalprogramm

In den vorhergehenden Kapitel haben wir gelernt, wie man eigene Funktionen erstellt und aufruft. Auch wenn es nicht so ganz zur Kapitelüberschrift passt, wollen wir zum Schluss auch einmal vorgefertigte Funktionen verwenden, die sich auch außerhalb von *function draw()* befinden. Mit ihnen wollen wir ein einfaches Malprogramm erstellen, mit dem kleine Kinder mit der Maus oder mit dem Tablet-Stift ein Bild wie Abbildung 7.17 malen können.

Schauen wir uns dazu den Sketch von Abbildung 7.18 an. Zusätzlich zu den bekannten Funktionen *function setup()* und *function draw()* sehen wir zwei neue Funktionen, ***function keyTyped()*** und ***function keyReleased()***. Schauen wir uns zuerst *function keyTyped()* in der Abbildung 7.18 (Zeile 18 bis 28) näher an. Hier werden drei if-Anweisungen aufgerufen, mit der wir die drei Grundfarben Rot, Grün und Blau aufrufen können. Wollen wir zum Beispiel mit der Farbe Rot malen, müssen wir auf unserer Tastatur die Taste r drücken (***keyIsPressed && key == r***). Dadurch ist die if-Anweisung erfüllt und r bekommt den Zahlenwert 255 zugeordnet. Für die Farben Grün und Blau gilt Entsprechendes.

Die Funktion ***keyReleased()*** setzt die Zahlenwerte für die einzelnen Farben wieder auf 0, sobald wir die Tastaturtaste loslassen. Ohne die Funktion *keyReleased()* würden die einzelnen Variablen r, g und b den zugeordneten Wert 255 behalten. Drücken wir die Taste g, nachdem wir zuvor die Taste r gedrückt haben, dann würden wir mit der Farbe Gelb und nicht mit der Farbe Grün malen. Haben wir alle drei Tasten nacheinander gedrückt, so müssten wir fortlaufend mit der Farbe Weiß malen. Hiervor schützt uns die Funktion *keyReleased()* (Zeile 30 bis 34).

Insgesamt stehen in unserem einfachen Mausmalprogramm den Kindern acht Farben zur Verfügung: **Schwarz** (keine Taste ist gedrückt), **Rot** (Taste r ist gedrückt), **Grün** (Taste g ist gedrückt), **Blau** (Taste b ist gedrückt), **Gelb** (die Tasten r und g sind gedrückt), **Violett** (die Tasten r und b sind gedrückt), **Türkis** (die Tasten g und b sind gedrückt), **Weiß** (die Tasten r, g und blau sind gedrückt).

Damit den Kindern das ganze Browserfenster für ihre Malkünste zur Verfügung steht, schreiben wir in Zeile 6 ***createCanvas(windowWidth, windowHeight)***. Um nun mit den einzelnen Farben zu malen, müssen die Kinder die entsprechenden Tastaturtasten und zusätzlich eine Maustaste gedrückt halten. Für die Farbe Schwarz genügt es, wenn nur die Maustaste gedrückt ist, da den Variablen r, g und b dann jeweils die Zahl 0 zugeordnet ist. Ob eine Maustaste gedrückt ist oder nicht, dies überwacht die Systemvariable ***mouseIsPressed*** mit der zugehörigen if-Anweisung (Zeile 11 bis 15).

## 7.6 Zusammenfassung

Fassen wir zusammen, was wir in Kapitel 7 gelernt haben, bzw. gelernt haben sollten.

|                           |                                                                                                                                                                                                                                |
|---------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>eigene Funktionen</b>  | Wir haben gelernt, wie man eigene Funktionen erstellt und aufruft. Da man dies nicht in zwei Zeilen zusammenfassen kann, sei hier auf das Kapitel 7.1 verwiesen.                                                               |
| <b>angleMode(DEGREES)</b> | Die Funktion <i>angleMode(DEGREES)</i> sorgt dafür, dass wir Winkel nicht im Bogenmaß, sondern in Grad angeben können.                                                                                                         |
| <b>push() und pop()</b>   | Die Funktion <i>push()</i> speichert das aktuelle Koordinatensystem und die Funktion <i>pop()</i> lädt das gespeicherte Koordinatensystem wieder und vergisst alles, was zwischen <i>push()</i> und <i>pop()</i> passiert ist. |

|                                                |                                                                                                                                                                                        |
|------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>keyTyped()</b>                              | Die Funktion <i>keyTyped()</i> wird aufgerufen, sobald man eine Taste auf der Tastatur drückt. Welche Tasten hiervon ausgenommen sind entnehme man der Referenz.                       |
| <b>keyReleased()</b>                           | Beim Loslassen einer Tastaturtaste wird die Funktion <i>keyReleased()</i> aufgerufen.                                                                                                  |
| <b>keysPressed</b>                             | <i>keysPressed</i> ist eine Systemvariable, die nur die Zustände <i>wahr</i> oder <i>falsch</i> annehmen kann. Ist eine Tastaturtaste gedrückt ist sie wahr. Ansonsten ist sie falsch. |
| <b>mouselsPressed</b>                          | <i>mouselsPressed</i> ist eine Systemvariable, die nur die Zustände <i>wahr</i> oder <i>falsch</i> annehmen kann. Ist eine Maustaste gedrückt, ist sie wahr. Ansonsten ist sie falsch. |
| <b>createCanvas(windowWidth, windowHeight)</b> | Hiermit erreicht man, dass die Leinwand das ganze Browserfenster einnimmt.                                                                                                             |

## 7.7 Übungen

**Aufgabe 1** Ändere den Sketch von Abbildung 7.5 so um, dass sich das folgende Bild ergibt (Abb. 7.19).

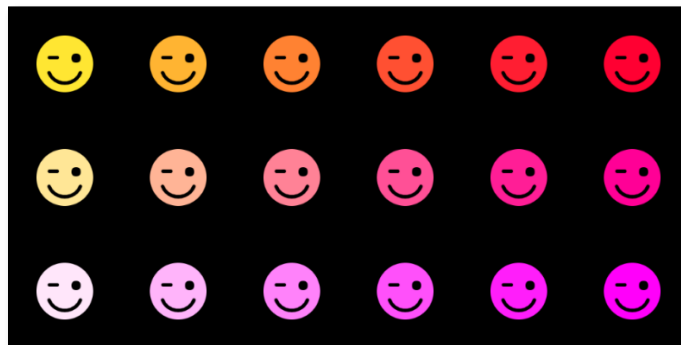


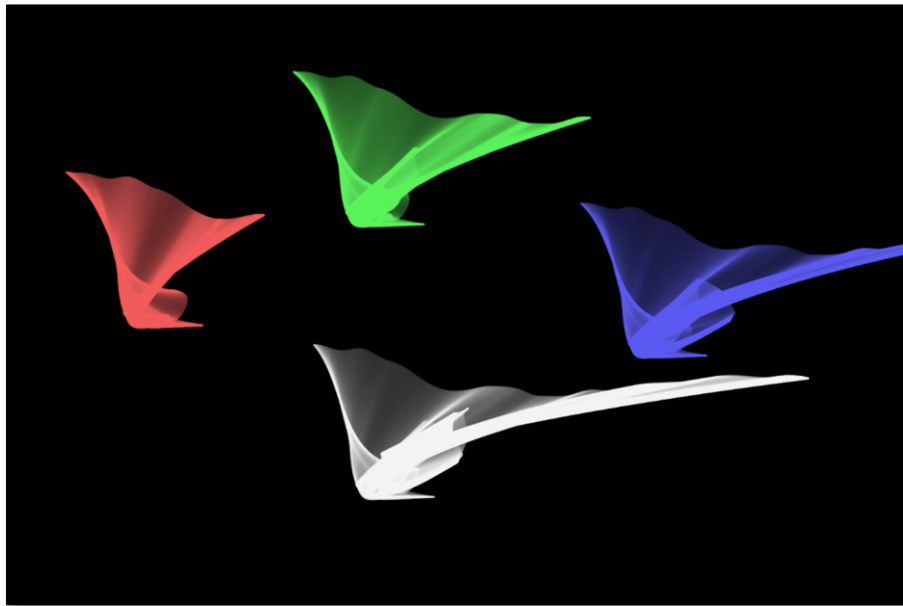
Abbildung 7.19: Smileys von gelb bis violett

**Aufgabe 2** Zeichne mit der Funktion *herz()* fünf Herzen von unterschiedlicher Größe und Farbe (siehe Abb. 7.20).



Abbildung 7.20: Fünf Herzen

**Aufgabe 3** Mit dem Sketch von Abbildung 6.16 kann man Objekte zeichnen, die einem Seidentuch ähneln. Verwandle diesen Sketch in eine Funktion und zeichne hiermit ein Bild, das der Abbildung 7.21 entspricht.



*Abbildung 7.21: Fliegende Seidentücher*

**Aufgabe 4** An jedem Adventssonntag zündet man eine Kerze mehr an. Daraus folgt, dass nach vier Wochen die erste Kerze deutlich kleiner ist als die vierte Kerze (siehe Abb. 7.22). Passe den Sketch von Abbildung 7.15 dieser Gegebenheit an.



*Abbildung 7.22: Vier Adventskerzen*



## 8 Arrays

### 8.1 Eindimensionales Array

Das Wort **Array** heißt übersetzt Ansammlung. Unter Array wollen wir hier einfachheitshalber mal eine Ansammlung von Behältern verstehen, die alle eine Nummer tragen, beginnend bei null. Abbildung 8.1 zeigt uns ein Array, welches aus 7 Behältern besteht. Auch wenn auf dem letzten Behälter von Abbildung 8.1 eine 6 steht, so sind es doch 7 Behälter, da wir, wie in der Informatik üblich, mit dem Zählen bei null beginnen.

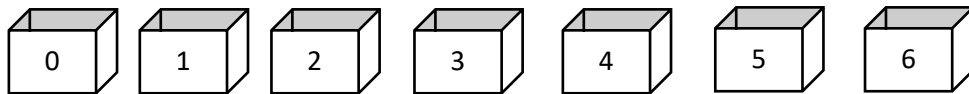


Abbildung 8.1: Ein Array, bestehend aus 7 Behältern

Diese Behälter wollen wir nun, wie in Abbildung 8.2 gezeigt, mit Zahlen füllen.

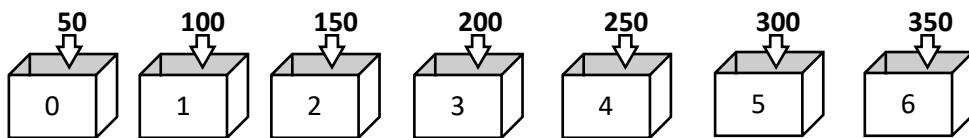


Abbildung 8.2: Die Behälter des Arrays werden mit Zahlen gefüllt

Mit Hilfe eines so befüllten Arrays können wir die Abbildung 8.3 erstellen. Für diese einfache Abbildung verwendet man normalerweise kein Array, doch an diesem einfachen Beispiel verstehen wir besser, wie ein Array funktioniert.

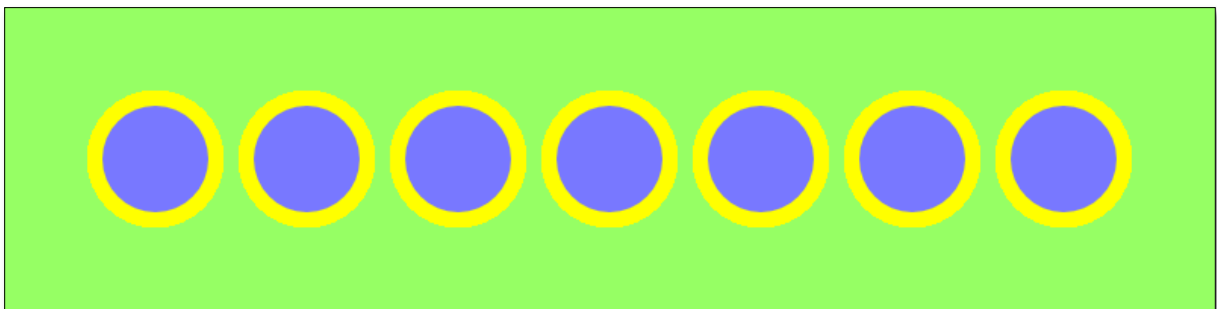


Abbildung 8.3: Sieben Kreise

Der Sketch auf der linken Seite von Abbildung 8.4 zeigt uns, wenn auch auf sehr umständliche Art, wie man die Abbildung 8.3 erzeugen kann. Schauen wir uns den Sketch im Einzelnen an. In Zeile 1 teilen wir p5.js mit `let a = [ ]` mit, dass wir ein Array mit dem Namen a erschaffen wollen. Die eckigen Klammern sind hier der Hinweis auf ein Array. Dieses Array ist jedoch noch leer und somit seine Größe unbestimmt, da zwischen den eckigen Klammern nichts steht. In Zeile 7 bis 13 nummerieren wir die einzelnen Array-Behälter von 0 bis 6 (Zahlen in den eckigen Klammern) und ordnen jedem Behälter eine Zahl entsprechend der Abbildung 8.2 zu. Die Inhalte (Zahlenwerte) der einzelnen Behälter des Arrays benutzen wir anschließend bei `function draw()` als x-Werte für die Erzeugung unserer blauen Kreise von Abbildung 8.3. Deutlich kürzer wird der Sketch, wenn wir, wie in Abbildung 8.4 rechts gezeigt, die Behälter des Arrays direkt in Zeile 1 befüllen.

Die eleganteste Art ein Array durchzunummerieren und zu befüllen ist jedoch die Verwendung einer *for*-Schleife. In Zeile 9 und 10 des Sketches von Abbildung 8.5 benutzen wir eine *for*-Schleife für die Beschriftung der Array-Behälter von 0 bis 6. Anschließend füllen wir in 50er-Schritten jeden Behälter mit der gewünschten Zahl. Mit *print(i, a[i])* (Zeile 15) können wir uns in der Konsole die Index-Werte und die Füll-Werte anzeigen lassen, um zu überprüfen, ob wir alles richtig gemacht haben.

Die Verwendung einer *for*-Schleife war bei diesem einfachen Sketch nicht zwingend notwendig, sie ist aber besonders vorteilhaft, wenn ein Array sehr viele Behälter, zum Beispiel 400 enthält, wie im Sketch von Abbildung 8.7.

```

1 let a = [];
2
3 function setup() {
4 createCanvas(400, 100);
5 background(150, 255, 100);
6
7 a[0] = 50;
8 a[1] = 100;
9 a[2] = 150;
10 a[3] = 200;
11 a[4] = 250;
12 a[5] = 300;
13 a[6] = 350;
14 }
15
16 function draw() {
17 stroke(255, 255, 0);
18 strokeWeight(5);
19 fill(120, 120, 255);
20 ellipse(a[0], 50, 40, 40);
21 ellipse(a[1], 50, 40, 40);
22 ellipse(a[2], 50, 40, 40);
23 ellipse(a[3], 50, 40, 40);
24 ellipse(a[4], 50, 40, 40);
25 ellipse(a[5], 50, 40, 40);
26 ellipse(a[6], 50, 40, 40);
27 }
28
1 let a = [50, 100, 150, 200, 250, 300, 350];
2
3 function setup() {
4 createCanvas(400, 100);
5 background(150, 255, 100);
6 }
7
8 function draw() {
9 stroke(255, 255, 0);
10 strokeWeight(5);
11 fill(120, 120, 255);
12 ellipse(a[0], 50, 40, 40);
13 ellipse(a[1], 50, 40, 40);
14 ellipse(a[2], 50, 40, 40);
15 ellipse(a[3], 50, 40, 40);
16 ellipse(a[4], 50, 40, 40);
17 ellipse(a[5], 50, 40, 40);
18 ellipse(a[6], 50, 40, 40);
19 }
20

```

Abbildung 8.4: Zwei Möglichkeiten, ein Array "zu Fuß" zu befüllen

```

1 let a = [];
2
3 function setup() {
4 createCanvas(400, 100);
5 background(150, 255, 100);
6 }
7
8 function draw() {
9 for (let i = 0; i < 7; i++) {
10 a[i] = 50 + i*50;
11 stroke(255, 255, 0);
12 strokeWeight(5);
13 fill(120, 120, 255);
14 ellipse(a[i], 50, 40, 40);
15 print(i, a[i]);
16 }
17 }
18

```

```

onsole
0 50
1 100
2 150
3 200
4 250
5 300
6 350

```

Abbildung 8.5: Ein Array wird mit einer *for*-Schleife durchnummeriert und anschließend befüllt

Wie in Abbildung 8.6 dargestellt, wollen wir in unserem nächsten Sketch 400 Schneeflocken hinter einem Fenster vom Himmel fallen lassen.

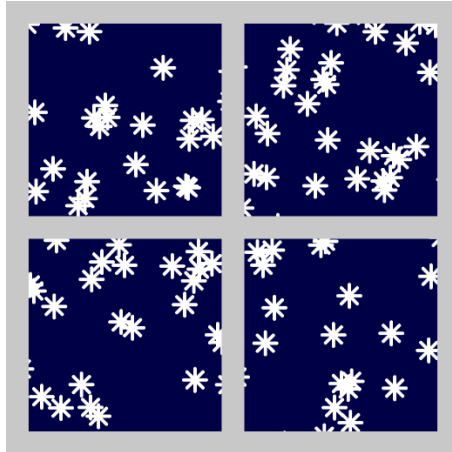


Abbildung 8.6: 400 Schneeflocken fallen vom Himmel

Unser Array bekommt diesmal den Namen `y`. In Zeile 10 des Sketches von Abbildung 8.7 beschriften wir 400 Behälter mit den Nummern von 0 bis 399. In Zeile 11 befüllen wir jeden einzelnen dieser Behälter mit einer Zufallszahl zwischen -1000 und 400.

Bei `function draw()` konstruieren wir eine Schneeflocke aus vier sich kreuzenden Linien (Zeile 22 – 27). Die Werte aus dem Array dienen hierbei als `y`-Werte für jede der 400 Schneeflocken. Deshalb der Name `y` für unser Array. Mit `y[i] += 1` (Zeile 20) sorgen wir dafür, dass sich die Schneeflocken bei jedem Durchlauf von `function draw()` einen Pixel nach unten bewegen. Dabei bleibt der `x`-Wert jeder einzelnen Schneeflocke gleich. Dafür sorgt `var x = i`. Die Programmzeilen für das Fenster bedürfen keiner weiteren Erläuterung.

```

1 let y = [];
2
3 function setup() {
4 createCanvas(400, 400);
5
6 /* Nun werden 400 Behälter erzeugt und jeder der 400
7 Behälter bekommt einen Nummer (0 - 399). Danach wird
8 jeder Behälter mit einer Zufallszahl für y zwischen
9 -1000 und 400 gefüllt. */
10 for(let i = 0; i < 400; i++) {
11 y[i] = random(-1000, 400);
12 }
13 }
14
15 function draw() {
16 background(0, 0, 70);
17
18 // Schneeflocken
19 for (let i = 0; i < 400; i++) {
20 y[i] += 1;
21 let x = i;
22 stroke(255);
23 strokeWeight(3);
24 line(x-7, y[i]+7, x+7, y[i]-7);
25 line(x-10, y[i], x+10, y[i]);
26 line(x-7, y[i]-7, x+7, y[i]+7);
27 line(x, y[i]+10, x, y[i]-10);
28 }
29
30 // Fensterrahmen
31 noFill();
32 stroke(200);
33 strokeWeight(20);
34 rect(10, 10, 380, 380);
35 rect(0, 200, 400, 200);
36 rect(200, 0, 200, 400);
37 }
38

```

Abbildung 8.7: Sketch zur Erzeugung und Animation von 400 Schneeflocken

## 8.2 Zweidimensionales Array

### Eierkarton

Im letzten Kapitel haben wir mit einem eindimensionalen Array den Fall von Schneeflocken animiert. Nun werden wir mit einem zweidimensionalen Array so etwas Verrücktes wie einen dreidimensionalen Eierkarton zeichnen, der sich mit der Maus drehen lässt. Hierzu sind aber einige Vorüberlegungen bezüglich der 3D-Darstellung mit p5.js an einem einfachen Beispiel notwendig. Da ein Bildschirm zweidimensional ist, kann er uns nur die Illusion einer 3D-Darstellung liefern. Mit dem Zusatz **WEBGL** bei der Funktion `createCanvas()` teilen wir p5.js in dem Sketch von Abbildung 8.8 in der Zeile 2 mit, dass wir die Illusion einer 3D-Darstellung wünschen. Dazu verschiebt p5.js den Koordinatenursprung in die Mitte der Leinwand und erzeugt zusätzlich zur x- und zur y-Achse eine unsichtbare z-Achse, die senkrecht auf der Leinwand steht. In Zeile 9 von Abbildung 8.8 steht `ellipse(0, 0, 20, 20)`. Ohne **WEBGL** würde der Mittelpunkt des Kreises auf der oberen linken Ecke der Leinwand liegen. Mit **WEBGL** wird der weiße Kreis in der Mitte der Leinwand gezeichnet.

Um es uns etwas einfacher zu machen, laden wir ein fertiges, von p5.js zur Verfügung gestelltes 3D-Objekt mit `torus(100, 30)` in unseren Sketch (Zeile 13). Die erste Zahl zwischen den runden Klammern gibt den Radius des Torus und die zweite Zahl gibt den Rohrradius an. Um den Torus im dreidimensionalen Raum zu drehen, benutzen wir die Funktionen `rotateX()` und `rotateY()`. Anstelle einer Zahl schreiben wir zwischen die Klammern `mouseX*0.01` und `mouseY*0.01`. So können wir den Torus bequem mit der Maus drehen und erhalten so einen überzeugenden dreidimensionalen Eindruck.

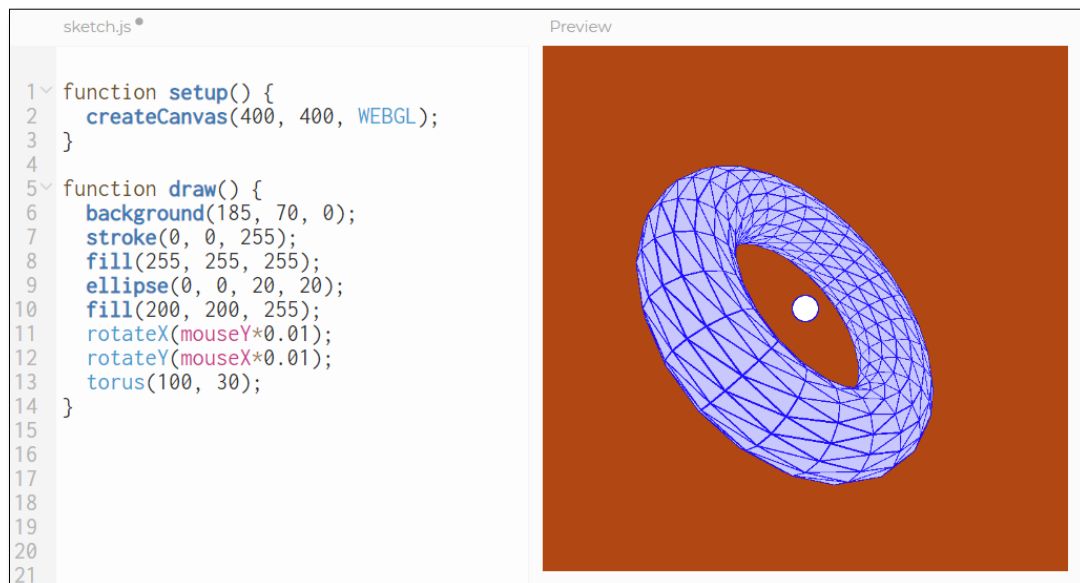


Abbildung 8.8: Mit der Maus drehbarer Torus (Donut)

Kommen wir nun wieder zu der verrückten Idee, einen dreidimensionalen Eierkarton mittels eines zweidimensionalen Arrays zu zeichnen. Wie unser Eierkarton aussieht, zeigt die Abbildung 8.9. Links sehen wir den Eierkarton in der Draufsicht und rechts sehen wir ihn von schräg oben. Die Abbildung 8.10 zeigt den zugehörigen Sketch. Die Zeilen 5, 21 und 22 dürften uns kein Kopfzerbrechen machen, da sie schon im Text zur Abbildung 8.8 erklärt wurden. Beachtung verdient die `if(mouselsPressed){rotateZ(millis/10000);}` von Zeile 24 bis 26. Sie bewirkt Folgendes: Wenn es wahr ist, dass eine Maustaste gedrückt ist, dann wird der Eierkarton solange um die z-Achse gedreht, wie die Maustaste gedrückt bleibt. Für die kontinuierliche Drehung sorgt hierbei `millis() / 10000`. Zur

Erinnerung, die Funktion *millis()* zählt die Millisekunden ab dem Start des Sketches. Auf der nächsten Seite finden wir dann die Erklärungen zu den etwas anspruchsvolleren Sketchzeilen von Abbildung 8.10.

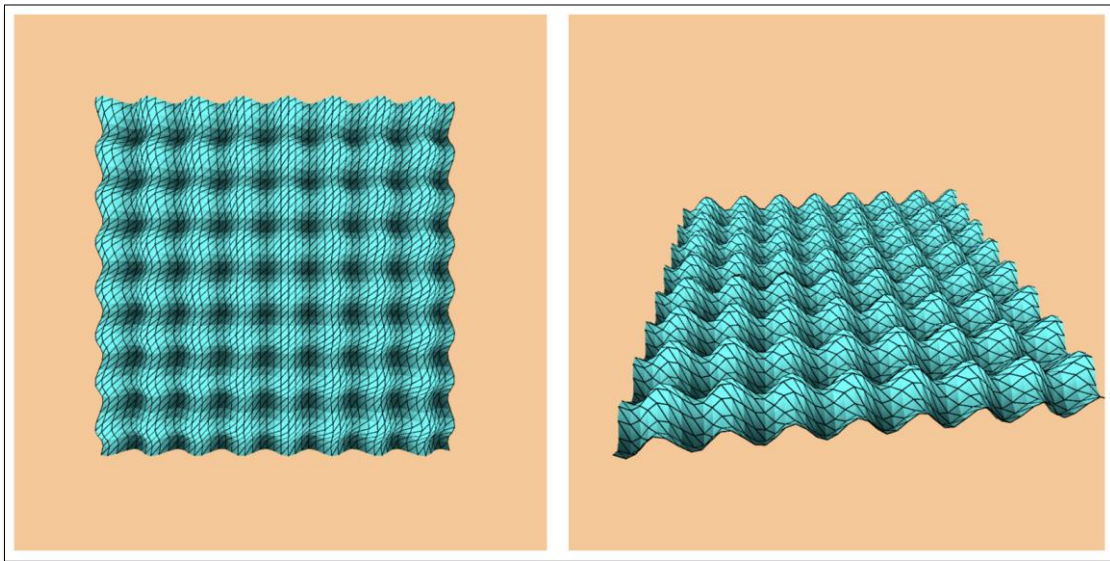


Abbildung 8.9: Eierkarton in der Draufsicht (links) und in der Ansicht von schräg oben (rechts)

```

1 let eierkarton = []; // Das Array eierkarton wird deklariert
2 let skalierung = 10; // Gibt die Größe einer Gitterzelle an
3
4 function setup() {
5 createCanvas(750, 750, WEBGL); // 3D-Einstellung
6 for (let x = 0; x < width; x++) {
7 eierkarton[x] = []; // Ein geschachteltes Array wird erstellt (2D-Array)
8 for (let y = 0; y < height; y++) {
9 // Das zweidimensionale Array wird mit Werten gefüllt
10 eierkarton[x][y] = sin(x) + cos(y);
11 }
12 }
13 }
14
15 function draw() {
16 background(255, 200, 150);
17 stroke(0);
18 strokeWeight(1);
19
20 // Mit der Maus kann der Eierkarton um die x- und y-Achse gedreht werden
21 rotateX(mouseY*0.01);
22 rotateY(mouseX*0.01);
23 // Wenn eine Maustaste gedrückt wird, dreht sich der Eierkarton um die z-Achse
24 if(mouseIsPressed) {
25 rotateZ(millis() / 10000);
26 }
27
28 for (let y = 0; y < 50; y++) {
29 beginShape(TRIANGLE_STRIP); // Zeichnet den Eierkarton mithilfe von Dreiecken
30 for (let x = 0; x < 50; x++) {
31 // Die Farben werden mittels der x- und y-Werte festgelegt
32 fill(0, eierkarton[x][y] * 50 + 200, eierkarton[x][y] * 50 + 200);
33 vertex((x - 25) * skalierung, (y - 25) * skalierung, eierkarton[x][y] * skalierung);
34 vertex((x - 25) * skalierung, (y - 25 + 1) * skalierung, eierkarton[x][y + 1] * skalierung);
35 }
36 endShape();
37 }
38 }
39

```

Abbildung 8.10: Sketch für die Erstellung von Abbildung 8.9

Schauen wir uns zuerst in der Abbildung 8.10 die Zeile 1 an. Hier wird ein leeres eindimensionales Array deklariert. In Zeile 2 legen wir mit der Variablen *skalierung* fest, wie groß unsere Zeichnungseinheiten sein sollen. Nachdem wir in Zeile 5 mit WEBGL p5.js mitgeteilt haben, dass wir eine 3D-Darstellung wünschen, verwenden wir eine doppelte for-Schleife (Zeile 6 bis 12), um aus unserem eindimensionalen Array ein zweidimensionales zu machen, die Array-Behälter zu nummerieren und diese anschließend mit Zahlen zu füllen. Aber jetzt nochmal ganz langsam. In Zeile 7 von Abbildung 8.10 wird das zweidimensionale Array erstellt. Man darf aber jetzt nicht glauben, dass die Array-Behälter auf einer x-y-Ebene aufgestellt werden. Das 2D-Array ist in Wahrheit ein eindimensional geschachteltes Array. Die folgende Abbildung 8.11 verdeutlicht dies. Mit *print(eierkarton)* lassen wir uns das verschachtelte Array in der Konsole anzeigen. Man sieht entsprechend den Zahlenwerten 3 und 5 in den beiden for-Schleifen von Abbildung 8.11, dass das Array drei Arrays enthält. Jedes der drei Arrays kann fünf Zahlen aufnehmen. In unserem Sketch zur Zeichnung des Eierkartons erzeugen wir ein Array, das 750 Arrays enthält, die jeweils 750 Zahlen aufnehmen können.

```

4 function setup() {
5 createCanvas(750, 750, WEBGL); // 3D-Einstellung
6 for (let x = 0; x < 3; x++) {
7 eierkarton[x] = []; // Ein geschachteltes Array wird erstellt (2D-Array)
8 for (let y = 0; y < 5; y++) {
9 // Das zweidimensionale Array wird mit Werten gefüllt
10 eierkarton[x][y] = sin(x) + cos(y);
11 print(eierkarton);
12 }
13 }
14 }

```

Console

15 ▶ [Array[5], Array[5], Array[5]]

Abbildung 8.11: Erstellung eines 2D-Arrays

In der Zeile 10 von Abbildung 8.10 füllen wir mit *eierkarton[x][y] = sin(x) + cos(y)* die Array-Behälter mit Zahlen. Das Zusammenwirken von Sinus- und Cosinusfunktion ergibt die Form eines Eierkartons.

Bei *function setup()* wird das 2D-Array erstellt und mit Zahlen gefüllt. Bei *function draw()* zeichnen wir mit einer doppelten for-Schleife den Eierkarton (Zeile 28 – 37). Neu ist hier die Füllung der Klammer von *beginShape()* mit **TRIANGLE\_STRIP**. Die Funktion **beginShape(TRIANGLE\_STRIP)** verbindet die *vertex()*-Punkte mit Linien zu aneinanderhängenden Dreiecken. Ein einfaches zweidimensionales Beispiel zeigt Abbildung 8.12.

```

> sketch.js
1 function setup() {
2 createCanvas(400, 400);
3 }
4
5 function draw() {
6 background(200);
7 stroke(255, 0, 0);
8 strokeWeight(3);
9
10 for(let y = 0; y < 8; y++) {
11 beginShape(TRIANGLE_STRIP);
12 fill(255, 200, 0);
13 vertex(50, 50);
14 vertex(50, 150);
15 vertex(100, 50);
16 vertex(100, 150);
17 vertex(150, 50);
18 vertex(150, 150);
19 vertex(200, 50);
20 vertex(200, 150);
21 endShape();
22 }
23 }
24

```

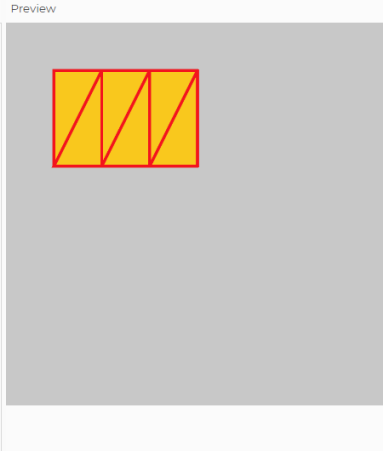


Abbildung 8.12: Zeichnen von zusammenhängenden Dreiecken mit TRIANGLE\_STRIP

Wenn man eine größere Fläche mit einem solchen Gitternetz bedecken will, erspart einem die Verwendung einer doppelten for-Schleife viel Tipparbeit (Abb. 8.13).

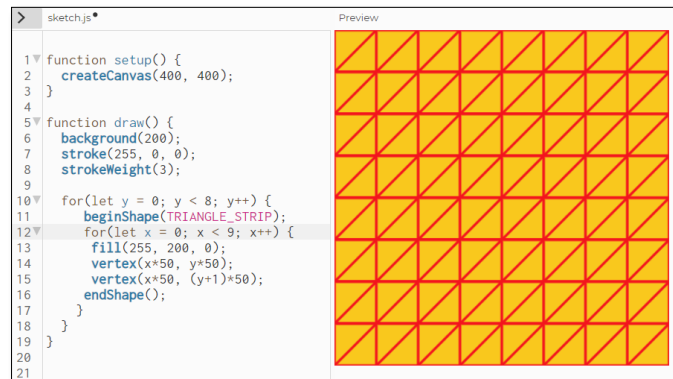


Abbildung 8.13: Zeichnen von zusammenhängenden Dreiecken mit `TRIANGLE_STRIP` und zweier for-Schleifen

Da wir eine 3D-Darstellung wünschen, enthalten die beiden `vertex()`-Funktionen in der Abbildung 8.10 drei Koordinatenwerte, stellvertretend für x, y und z (Zeile 33 und 34). Die x- und y-Werte liegen in der Ebene, während unser Array mit `eierkarton[x][y]*skalierung` und `eierkarton[x][y+1]*skalierung` die Werte für die z-Richtung liefert. Wenn man sich die Abbildungen des Eierkartons (Abb. 8.9) genauer anschaut, dann sieht man, dass das Gitternetz nun nicht mehr aus Dreiecken, sondern aus Vierecken besteht. Dieses Gitternetz bedeckt nun unsere dreidimensionale Eierkarton-Ebene. Die Größe der Vierecke können wir mittels `skalierung` festlegen (Zeile 2 im Sketch von Abbildung 8.10).

Unser Array benutzen wir auch zur Farbgebung für unseren Eierkarton (Zeile 32).

## Landschaft

In Kapitel 6.2 habe ich erwähnt, dass man mithilfe von Perlin noise auch Abbildungen von Landschaften erzeugen kann. Ein einfaches Beispiel zeigt Abbildung 8.14.

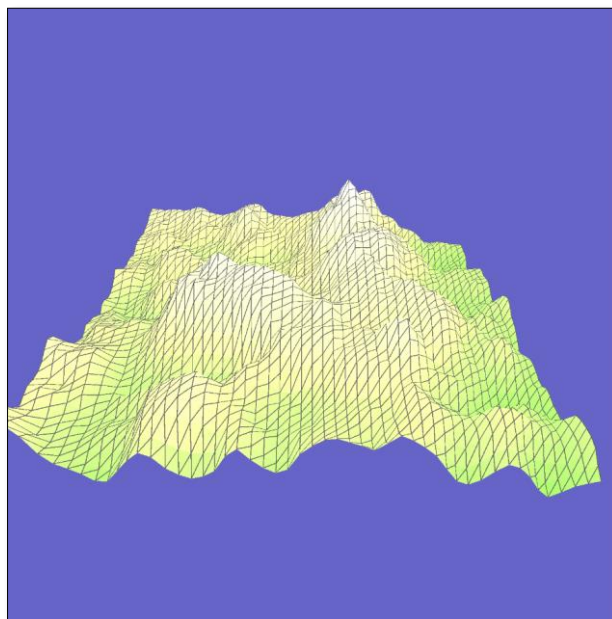


Abbildung 8.14: Mit der Funktion `noise()` und einem 2D-Array erstellte Landschaft

Um eine Landschaft wie in Abbildung 8.14 zu erstellen, muss man den Eierkarton-Sketch nur geringfügig ändern. In Zeile 10 (siehe Abb. 8.10 und Abb. 8.15) ersetzen wir den mathematischen Ausdruck  $\sin(x) + \cos(y)$  einfach durch  $30*\text{noise}(x/15, y/15)$ . Die anderen geringfügigen Änderungen bedürfen keiner weiteren Erläuterung.

```

1 let landschaft = []; // Das Array landschaft wird deklariert
2 let skalierung = 7; // Gibt die Größe einer Gitterzelle an
3
4 function setup() {
5 createCanvas(750, 750, WEBGL); // 3D-Einstellung
6 for (let x = 0; x < 1000; x++) {
7 landschaft[x] = []; // Ein geschachteltes Array wird erstellt (2D-Array)
8 for (let y = 0; y < 1000; y++) {
9 // Das zweidimensionale Array wird mit Werten gefüllt
10 landschaft[x][y] = 30*noise(x/15, y/15);
11 }
12 }
13 }
14
15 function draw() {
16 background(100, 100, 200);
17 stroke(100);
18 strokeWeight(0.5);
19
20 // Die Landschaft wird um PI/3 = 60° um die x-Achse gedreht
21 rotateX(PI/3);
22 // Bei Mausdruck dreht sich die Landschaft um die z-Achse
23 if(mouseIsPressed) {
24 rotateZ(millis() / 10000);
25 }
26
27 for (let y = 0; y < 50; y++) {
28 beginShape(TRIANGLE_STRIP); // Zeichnet die Landschaft mithilfe von Dreiecken
29 for (let x = 0; x < 50; x++) {
30 fill(landschaft[x][y]*15, landschaft[x][y] * 30 , landschaft[x][y] * 10);
31 vertex((x -25) * skalierung, (y) * skalierung, landschaft[x][y] * skalierung);
32 vertex((x-25) * skalierung, (y + 1) * skalierung, landschaft[x][y + 1] * skalierung);
33 }
34 endShape();
35 }
36 }
37

```

Abbildung 8.15: Sketch zur Erstellung der Landschaft von Abbildung 8.14

## Uhr

Zu Beginn dieses Kapitels haben wir die Funktionen `rotateX()` und `rotateY()` kennengelernt, die immer dann benötigt werden, wenn wir etwas im 3D-Raum drehen wollen. Wenn wir nicht im 3D-Raum, d.h. ohne `WEBGL` arbeiten, dann können wir für eine Drehung die Funktion `rotate()` benutzen. Mit ihr kann man Objekte um den Ursprung des Koordinatensystems drehen. Zum Beispiel die zwei Zeiger einer Uhr (Abb. 8.16). Hierzu ist es notwendig, mit der Funktion `translate()` den Ursprung des Koordinatensystems in die Mitte der Leinwand zu verschieben. Wenn man möchte, dass sich die Zeiger der Uhr unabhängig voneinander drehen, dann benötigt man die Funktionen `push()` und `pop()`. Diese haben wir schon im Kapitel 7.4 kennengelernt. Im Sketch zum Zeichnen der Uhr, wird ihr Einsatz nochmal erläutert (Abb. 8.17).

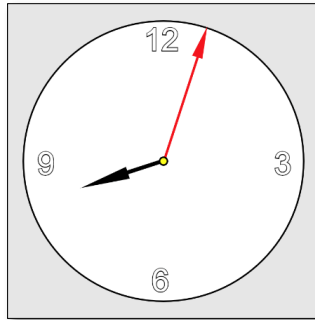


Abbildung 8.16: Uhr mit rotierenden Zeigern

```

1 let a = 0; // Rotationsfaktor für den roten Zeiger
2 let b = 0; // Rotationsfaktor für den schwarzen Zeiger
3
4 function setup() {
5 createCanvas(400, 400);
6 }
7
8 function draw(){
9 background(230);
10 translate(width/2, height/2); // Koordinatenursprung wird in die Fenstermitte verschoben
11
12 // Zifferblatt
13 fill(255);
14 ellipse(0, 0, 355, 355);
15 textSize(40);
16 text("12", -25, -140);
17 text("3", 140, 17);
18 text("6", -15, 165);
19 text("9", -160, 17);
20
21 push(); // Speichert die obige Transformation
22
23 // roter Zeiger
24 rotate(a*PI/60);
25 a = a + 0.4; // Rotationsgeschwindigkeit wird festgelegt
26
27 stroke(255, 0, 0);
28 strokeWeight(3);
29 line(0, 0, 150, 0);
30 fill(255, 0, 0);
31 triangle(140, -5, 170, 0, 140, 5); // Zeigerspitze
32
33 pop(); /* Lädt die oben gespeicherte Transformation wieder und vergisst,
34 was zwischen push() und pop() steht.
35 Somit addiert sich die folgende Rotation nicht zur obigen Rotation */
36
37 // schwarzer Zeiger
38 rotate(b*PI/60.0);
39 b = b + 0.1; // Rotationsgeschwindigkeit wird festgelegt
40
41 stroke(0);
42 strokeWeight(5);
43 line(0, 0, 70, 0);
44 fill(0);
45 triangle(50, -5, 90, 0, 50, 5); // Zeigerspitze
46
47 // Kreis über dem Mittelpunkt der Uhr
48 strokeWeight(2);
49 fill(255, 255, 0);
50 ellipse(0, 0, 10, 10);
51 }
52

```

Abbildung 8.17: Sketch zum Zeichnen der Uhr

## 8.3 Zusammenfassung

Fassen wir zusammen, was wir in Kapitel 8 gelernt haben, bzw. gelernt haben sollten.

### Array

Das Wort Array heißt übersetzt Ansammlung. Unter Array wollen wir einfachheitshalber mal eine Ansammlung von Behältern verstehen, die alle eine Nummer tragen, beginnend bei null. Jeder dieser Behälter kann eine Zahl aufnehmen. Hier ist ein Beispiel: `let a = [ ]`. Die eckigen Klammern zeigen an, dass es sich um ein Array mit dem Namen `a` handelt. Die einfachste Art dieses Array zum Beispiel mit 4 Zahlen zu füllen ist die folgende: `a[0] = 12, a[1] = 5, a[2] = 7, a[3] = 15`  
Um größere Arrays zu befüllen, ist die Verwendung einer `for`-Schleife vorteilhaft.

### 2D-Array

Das 2D-Array ist in Wahrheit ein eindimensional geschachteltes Array. Die folgende Abbildung verdeutlicht dies. In der Konsole wird angezeigt, dass dieses Array drei Arrays enthält. Jedes der drei Arrays kann fünf Zahlenwerte aufnehmen.

```
1 let a = []; // Das Array a wird deklariert
2
3 function setup() {
4 createCanvas(200, 200, WEBGL); // 3D-Einstellung
5 for(let x = 0; x < 3; x++) {
6 a[x] = []; // Ein geschachteltes Array wird erstellt (2D-Array)
7 for(let y = 0; y < 5; y++) {
8 // Das zweidimensionale Array wird mit Werten gefüllt
9 a[x][y] = sin(x) + cos(y)
10 }
11 }
12 print(a)
13 }
14
```

Console

```
▶ [Array[5], Array[5], Array[5]]
```

### WEBGL

Mit `createCanvas(400, 400, WEBGL)` teilen wir `p5.js` mit, dass wir auf einer 400 x 400 Pixel großen Leinwand die Illusion einer 3D-Darstellung wünschen.

### torus()

`torus(100, 30)` ist ein fertiges 3D-Objekt von `p5.js`. Die erste Zahl zwischen den runden Klammern gibt den Radius des Torus und die zweite Zahl gibt den Rohrradius an.

### rotateX()

Hiermit kann man seine Zeichnung um die x-Achse drehen.

### rotateY()

Hiermit kann man seine Zeichnung um die y-Achse drehen.

### rotateZ()

Hiermit kann man seine Zeichnung um die z-Achse drehen.

### rotate()

Hiermit kann man seine Zeichnung um den Ursprung des Koordinatensystems drehen.

### TRIANGLE\_STRIP

Steht zwischen den Klammern von `beginShape()` der Ausdruck `TRIANGLE_STRIP`. Dann werden die folgenden `vertex()`-Punkte mit Linien zu aneinanderhängenden Dreiecken verbunden.

## 8.4 Übungen

Aufgabe 1a) Mit dem Sketch von Abbildung 7.3 wurden sechs Smileys mithilfe einer selbsterstellten Funktion und einer for-Schleife gezeichnet. Auch Array-Behälter kann man mit eigenen Funktionen füllen. Ändere den Sketch von Abbildung 7.3 so um, dass die sechs Smileys mittels eines Arrays gezeichnet werden.



Abbildung 8.18: Die sechs Smileys wurden mithilfe eines Arrays gezeichnet

b) Ändere den Sketch von Aufgabe 1a so um, dass aus den Smileys Kürbisköpfe werden, die hin und her wackeln.



Abbildung 8.19: Hin und her wackelnde Kürbisköpfe

Aufgabe 2 Im Sketch von Abbildung 8.7 wird der Fall von 400 Schneeflocken hinter einem Fenster animiert. Ändere den Sketch so um, dass 900 Regentropfen vom Himmel fallen. Ein Teil von ihnen wird in einem roten Eimer aufgefangen, sodass hier der Wasserspiegel ansteigt (siehe Abbildung). Wenn der Eimer voll ist, soll der Sketch gestoppt werden.

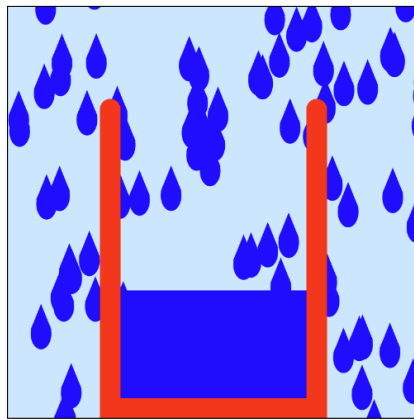


Abbildung 8.20: Regentropfen füllen einen Eimer

Aufgabe 3 Schau in der Referenz von p5.js nach, was sich unter **cone()** verbirgt. Zeichne hiermit die folgende dreidimensionale Abbildung eines Sonnenschirms, der um die x- und z-Achse rotiert. Was bedeutet in dem Beispielsketch der Referenz der Ausdruck **frameCount**? Verändere den Sketch so, dass du den Schirm mittels Mausbewegung drehen kannst.

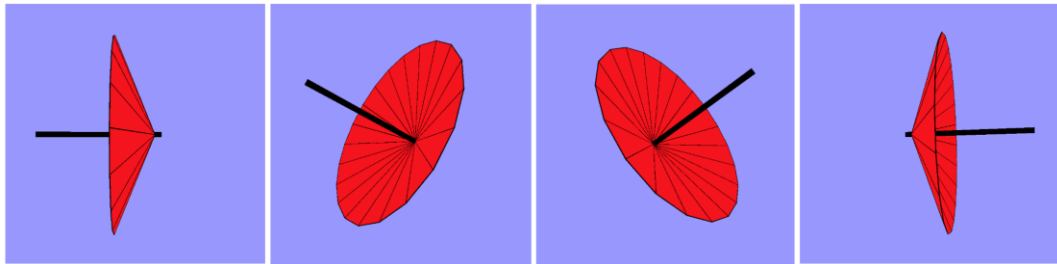


Abbildung 8.21: Dreidimensionaler Sonnenschirm

Aufgabe 4 Zeichne mittels eines Arrays den in der Abbildung dargestellten dreidimensionalen Papierflieger, der mit der Maus um die x- und y-Achse gedreht werden kann. Drückt man eine Maustaste, dann soll er sich um die z-Achse drehen.

Eine der vier folgenden Funktionen kann bei der Konstruktion des Papierflieger hilfreich sein.

$$(x-y)^2+(x+y)^2$$

$$(x-y)^3-(x-y)^3$$

$$60-\text{abs}(x+y)-\text{abs}(x-y)$$

$$30-(x-y)+(x+y)$$

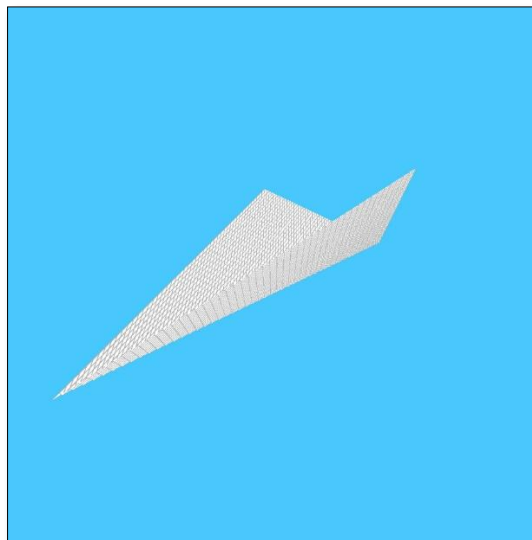


Abbildung 8.22: Papierflieger

Aufgabe 5 Wenn man sich ein Foto von einem Eierkarton anschaut, dann sieht man, dass das mit dem Sketch von Abbildung 8.10 erzeugte Bild mit der Wirklichkeit nicht ganz übereinstimmt (siehe Foto zu dieser Aufgabe und Abbildung 8.9).



Abbildung 8.23: Foto eines Eierkartons

Nimm die notwendigen Änderungen am Sketch von Abbildung 8.10 vor, damit ein Bild wie in der folgenden Abbildung entsteht.

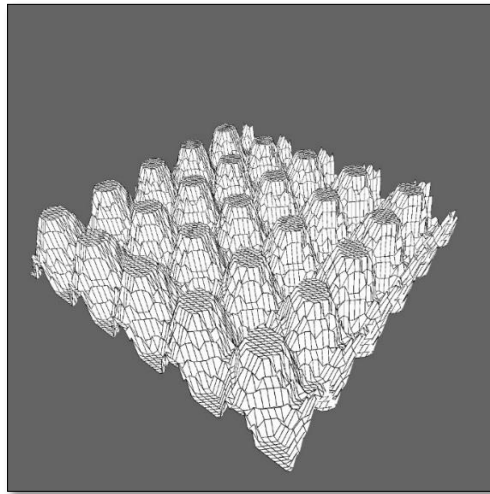


Abbildung 8.24: Verbesserte Darstellung eines Eierkartons

Aufgabe 6 Lasse deiner Kreativität freien Lauf und experimentiere mit beliebigen Funktionen. Fülle hiermit die Behälter eines 2D-Arrays, um neue Bilder zu erzeugen. Die folgenden Abbildungen sind ein Beispiel für freudiges Experimentieren. Mit etwas Fantasie erkennt man eine Segelregatta sowie ein anfliegendes Insekt.

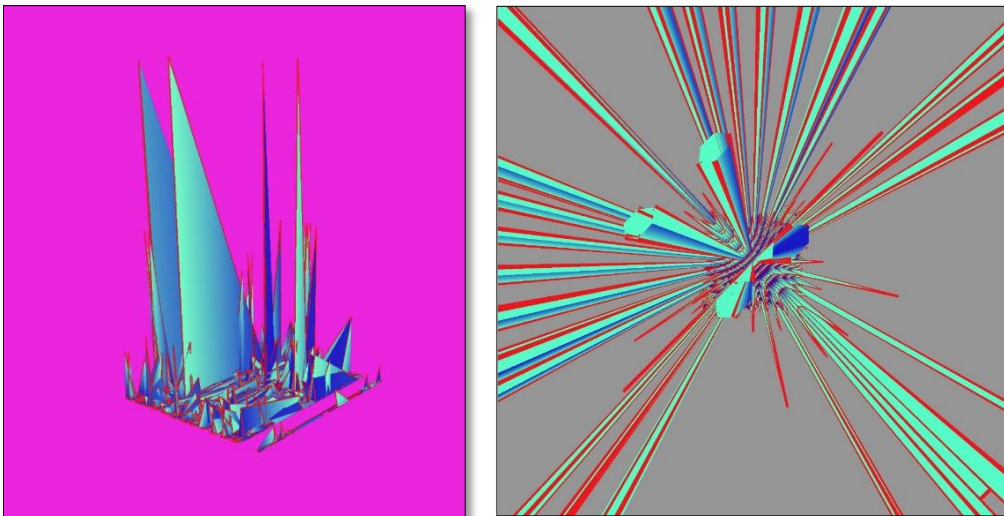


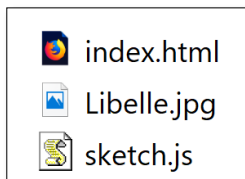
Abbildung 8.25: Segelregatta (links) und anfliegendes Insekt (rechts)



## 9 Bildbearbeitung

### 9.1 Erste Schritte

In diesem Kapitel lernen wir, wie man Fotos mittels p5.js auf die Leinwand bringen und bearbeiten kann. Dazu lernen wir zwei neue Funktionen kennen: ***preload()*** und ***image()***. In unserem ersten Sketch (Abb. 9.1) sehen wir, dass die Funktion ***preload()*** noch vor der Funktion ***setup()*** ausgeführt wird. Dies geschieht deshalb, weil das Hochladen von Fotos oft etwas Zeit in Anspruch nimmt. Erst wenn das Foto hochgeladen ist, beginnt die Funktion ***setup()*** ihre Befehlszeilen abzuarbeiten. Mit ***preload()*** können



Bilddateien mit den Endungen ***"\*.jpg"***, ***"\*.png"*** und ***"\*.gif"*** hochgeladen werden. Voraussetzung hierfür ist jedoch, dass, wie links dargestellt, sich die Bilddatei im gleichen Ordner wie die Datei ***index.html*** und ***sketch.js*** befindet. Außerdem muss, wie in Zeile 4 von dem Sketch in Abbildung 9.1 dargestellt, die Datei korrekt mit ihrem Namen und ihrer Dateierweiterung aufgerufen werden.

In dem Sketch von Abbildung 9.1 haben wir die Größe der Leinwand der Bildgröße angepasst. Man kann die Bildgröße aber auch der Leinwandgröße anpassen. So kann man zum Beispiel schreiben ***image(bild, 0, 0, 600, 400)*** oder wie in Abbildung 9.2 gezeigt ***image(bild, 0, 0, mouseX, mouseY)*** schreiben. Hier liegt der linke Eckpunkt des Bildes bei ***x = 0*** und ***y = 0***, während der untere rechte Eckpunkt durch die x-y-Koordinaten des Mauszeigers bestimmt wird. Schreibt man ***image(bild, 50, 50, 600, 400)*** so liegt der linke obere Bildpunkt bei ***x = 50*** und ***y = 50*** und der untere rechte Bildpunkt des 600x400 Pixel großen Bildes bei ***x = 650*** und ***y = 450***.

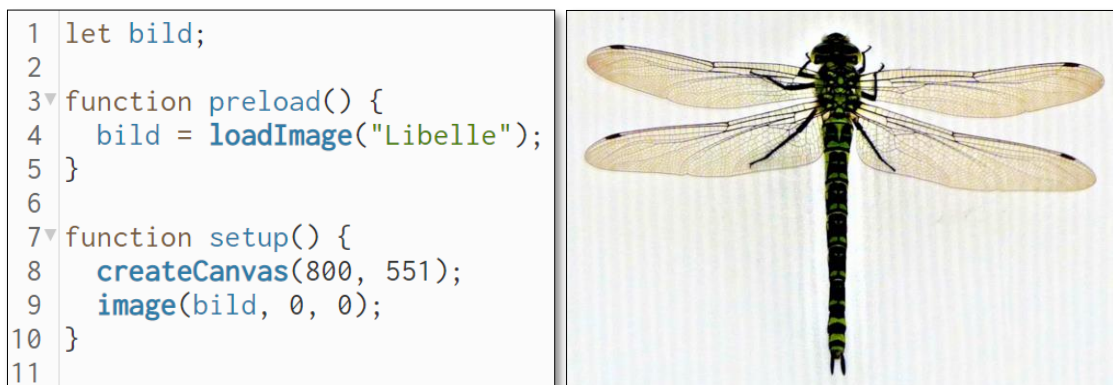
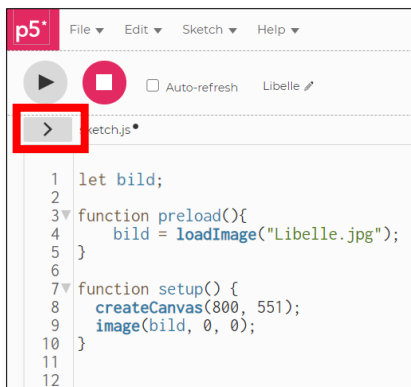


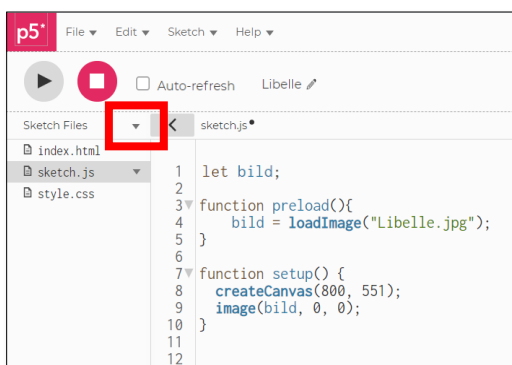
Abbildung 9.1: Mit dem links abgebildeten Sketch können wir ein Foto auf die Leinwand bringen

Wenn wir den Sketch von Abbildung 9.1 offline fertiggestellt haben, dann möchten wir ihn durch einen Klick auf ***index.html*** in einem Webbrowser öffnen. Bei der Erstausgabe dieses Buches im Januar 2019 war dies mit dem Webbrowser ***Firefox*** kein Problem. Doch ***Google Chrome*** und ***Internet Explorer*** weigerten sich aber schon damals, ein lokal gespeichertes Foto hochzuladen. ***Firefox*** ist diesem Beispiel inzwischen gefolgt und erlaubt das Hochladen von Fotos nur, wenn diese von einem Webserver geliefert werden. Man kann das Problem lösen, indem man für sich einen lokalen Webserver erstellt. Hierzu gibt es Anleitungen im Internet. Wesentlich einfacher ist es jedoch, wenn man im p5.js-Webeditor einen Account anlegt (<https://editor.p5js.org/>, rechts oben im Webeditor bei ***Sign up***). Nachdem man sich eingeloggt hat, schreibt man seinen Code wie gewohnt im Browserfenster. Um ein Foto oder auch eine Sounddatei (Kapitel 12) einzufügen, geht man wie folgt vor.

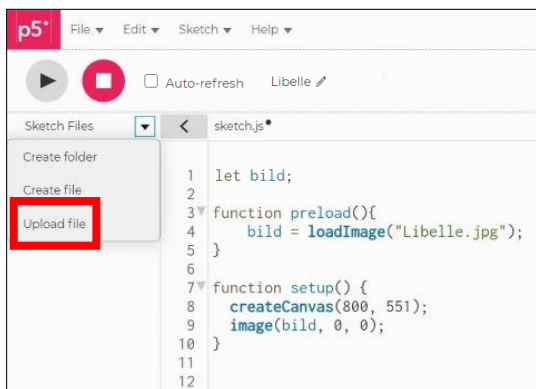
1. Schritt: Klicke auf die rot gekennzeichnete Stelle.



2. Schritt: Klicke auf die rot gekennzeichnete Stelle.



3. Schritt: Klicke auf die rot gekennzeichnete Stelle.



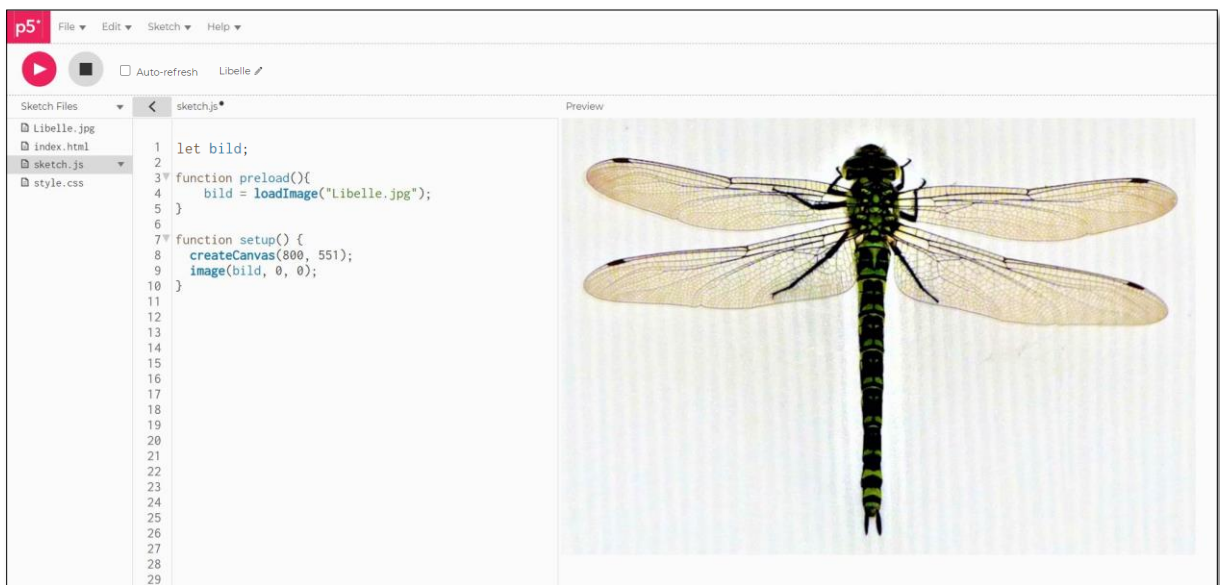
4. Schritt: Schiebe die Bilddatei aus deinem Dateimanager in das Fenster *Upload File*.



5. Schritt: Wenn der File hochgeladen ist, dann kann das Fenster *Upload File* geschlossen werden.



6. Schritt: Nun kann man den Sketch starten.



7. Schritt: Abschließend dem Sketch einen Namen geben und ihn bei p5.js abspeichern.

In unserem sehr einfachen Sketch von Abbildung 9.1 haben wir auf die Funktion *function draw()* verzichtet, da wir das Foto der Libelle nur einmal hochladen wollten. Wollen wir jedoch Änderungen an diesem Foto im Browserfenster vornehmen, so muss sich die Funktion *image()* bei *function draw()* befinden (siehe Zeile 13 bei Abbildung 9.2), da das Foto entsprechend der Vorgaben im Sketch öfters aktualisiert werden muss. Mit der Zeile *image(bild, 0, 0, mouseX, mouseY)* im Sketch von Abbildung 9.2 können wir die Größe des Fotos im Browserfenster mit dem Mauszeiger verändern. Die Zahlenwerte von *mouseX*, *mouseY* geben den unteren rechten Eckpunkt des Bildes an.

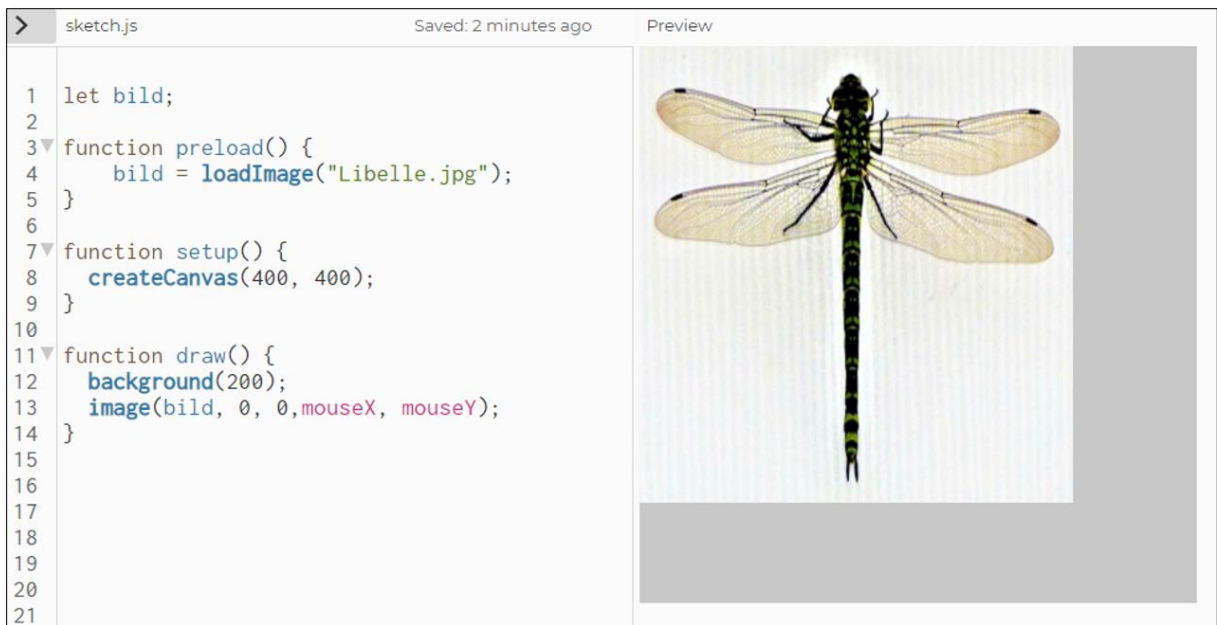


Abbildung 9.2: Mit dem Mauszeiger kann die Größe des Bildes auf der Leinwand geändert werden

In unserem nächsten Sketch wollen wir auf unserer Leinwand 176 Libellen zeichnen. Dazu benutzen wir eine doppelte for-Schleife und statt *mouseX* und *mouseY* schreiben wir nun 50 und 50 für die Größe jedes Einzelbildes (Zeile 13 bis 15 in Abbildung 9.3). Für jeden von den doppelten for-Schleifen generierten x- und y-Wert erzeugt die Funktion *image()* nun ein Bild von der Libelle, dessen oberer linker Eckpunkt die Koordinaten x und y und deren unterer Eckpunkt die Koordinaten x + 50 und y + 50 besitzt. Da unsere Leinwand 800 Pixel breit und 550 Pixel hoch ist (Zeile 8) erhalten wir so 16 x 11 = 176 Bilder von unserer Libelle (Abb. 9.4).

```

1 let bild;
2
3 function preload() {
4 bild = loadImage("Libelle.jpg");
5 }
6
7 function setup() {
8 createCanvas(800, 550);
9 }
10
11 function draw() {
12 background(200);
13 for(let x = 0; x < width; x += 50) {
14 for(let y = 0; y < height; y += 50) {
15 image(bild, x, y, 50, 50);
16 }
17 }
18 }
19

```

Abbildung 9.3: Sketch für die Erzeugung von 176 Libellenbildern

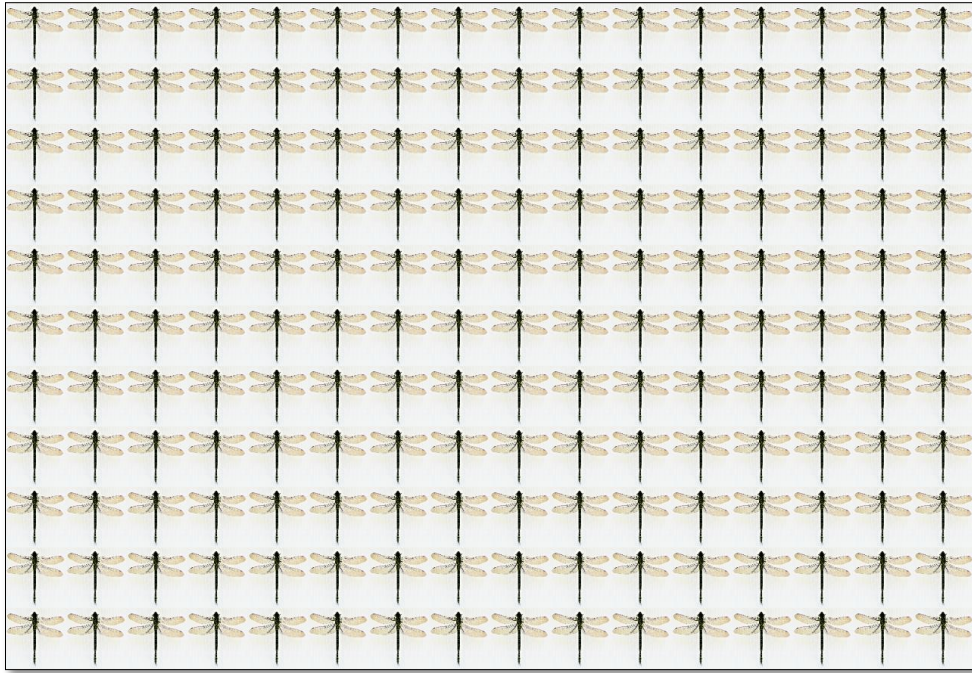


Abbildung 9.4: 176 Libellenbilder

Im Kapitel 8.2 haben wir als erstes 3D-Objekt einen Torus kennengelernt. Wenn wir uns unter dem Suchbegriff 3D in der Referenz von p5.js umsehen, dann entdecken wir noch weitere 3D-Objekte wie *box()*, *sphere()*, ... . Will man den Oberflächen dieser 3D-Objekte eine Farbe geben, dann gelingt dies mit der Funktion *fill()*. Will man jedoch die Oberflächen mit einem Foto versehen, dann muss man die Funktion ***texture()*** verwenden (siehe Abb. 9.5 Zeile 17). Abbildung 9.5 zeigt einen rotierenden Würfel, auf dessen sechs Seiten das Foto der Libelle zu sehen ist.

```

1 let bild;
2 let w = 0;
3
4 function preload() {
5 bild = loadImage("Libelle.jpg");
6 }
7
8 function setup() {
9 createCanvas(700, 700, WEBGL);
10 }
11
12 function draw() {
13 background(100);
14 rotateZ(w);
15 rotateX(w);
16 rotateY(w);
17 texture(bild);
18 box(300, 300, 300);
19 w += 0.005;
20 }
21

```

Abbildung 9.5: Libelle auf einem rotierenden Würfel

## 9.2 Pixelarray

Besucht man in Figueres (Spanien) das Museum von Salvador Dali, so fällt einem ein großformatiges Bild besonders auf (siehe Abbildung 9.6 links). Wenn man das Bild aus der Nähe betrachtet, sieht man Dalis Frau Gala, die aufs Meer schaut. Entfernt man sich jedoch 20 bis 30 Meter von diesem Bild oder betrachtet es unscharf, indem man seine Brille abnimmt oder seine Augen zusammenkneift, so erkennt man das gerasterte Portrait von Abraham Lincoln. Hier wird einem vor Augen geführt, dass einem durch eine analytische detaillierte Untersuchung der Feinstruktur des Bildes ein wesentlicher Teil verborgen bleibt. Erst wenn wir einen, wenn auch unscharfen Überblick wagen, erkennen wir eine wesentliche Aussage des Bildes. Dieses Prinzip der Unschärfe kann man auch auf andere komplexe Strukturen wie Ökosysteme, Politik, Wirtschaft, ... übertragen. Wenn wir über charakteristische, aber nicht zwingend über alle detaillierten Informationen verfügen, können wir, wenn auch unscharf, Wesentliches erkennen.



Abbildung 9.6: Gerasterte Bilder

Doch kommen wir zurück zum Thema Programmieren. Wir wollen nun das Bild des Mannes mit dem Bart von Abbildung 9.6 ähnlich wie das Bild von Abraham Lincoln rastern. Wie dies mit p5.js gelingt, zeigt uns der Sketch von Abbildung 9.7. Dieser Sketch bedarf jedoch einer ausführlichen Erläuterung. In ihm lernen wir zwei neue Funktionen kennen,

**pixelDensity()** und **loadPixels()**.

In Zeile 25 des Sketches von Abbildung 9.7 sehen wir, dass das Bild von dem Mann mit Bart in 15 Pixel mal 15 Pixel große Quadrate zerlegt werden soll. Diese Quadrate müssen jedoch, damit sie bei einer unscharfen Betrachtung des Bildes zur Erkennung der abgebildeten Person führen, mit den entsprechenden Farben des unveränderten Bildes gefüllt werden. Dazu laden wir mit der Funktion **loadPixels()** die Farbinformationen jedes Pixels des Ausgangsbildes in ein **eindimensionales Array** (Zeile 12 in Abbildung 9.7). Ein Pixel auf unserem Monitor besteht normalerweise aus drei engbenachbarten Subpixeln in den Farben Rot, Grün und Blau. p5.js speichert für einen Pixel jedoch noch zusätzlich zu den drei Zahlenwerten für die drei Farben Rot, Grün und Blau (Werte von 0 – 255) den Alpha-Wert (0 – 255) für die Transparenz ab. Veranschaulichen wir uns dies anhand eines einfachen Beispiels. Wie in Abbildung 9.8 dargestellt, sind für die Abspeicherung eines Bildes, welches nur aus  $2 \times 2 = 4$  Pixeln besteht, in einen eindimensionalen Array bereits 16 Array-Behälter mit den Indexzahlen von 0 bis 15 notwendig.

```

1 let bild;
2
3 function preload() {
4 bild = loadImage("Mann_mit_Bart.jpg");
5 }
6
7 function setup() {
8 createCanvas(268, 400);
9 background(200);
10 pixelDensity(1);
11 image(bild, 0, 0);
12 loadPixels();
13 noStroke();
14
15 for(let x = 0; x < width; x += 15) {
16 for(let y = 0; y < height; y += 15) {
17 let i = (x + y * width) * 4;
18
19 let r = pixels[i + 0];
20 let g = pixels[i + 1];
21 let b = pixels[i + 2];
22 let a = pixels[i + 3];
23
24 fill(r, g, b, a);
25 rect(x, y, 15, 15);
26 }
27 }
28 }
29

```

Abbildung 9.7: Sketch zur Bildrasterung

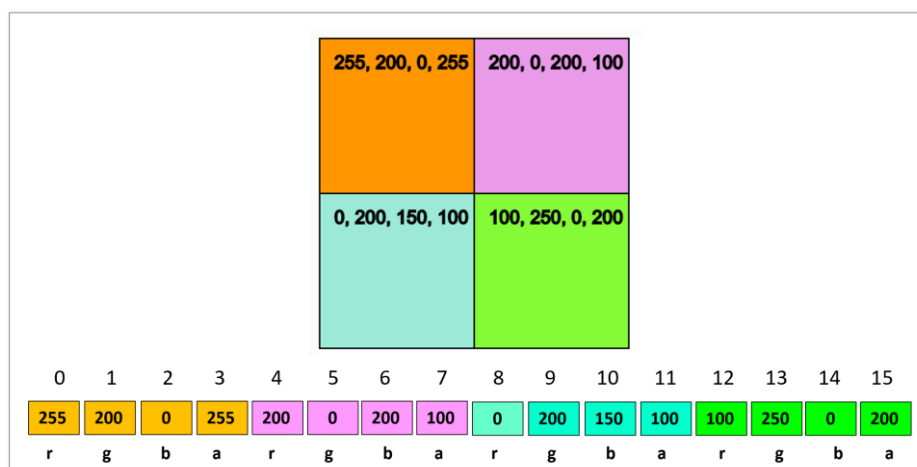


Abbildung 9.8: Ein aus nur 4 Pixeln bestehendes Bild wird mit loadPixels() in einem eindimensionalen Array abgespeichert

Zurück zur Abbildung 9.7. Mit einer doppelten for-Schleife und  $let i = (x + y * width) * 4$  legen wir hier fest, auf welche Behälter in unserem Pixelarray wir in 15er-Schritten zugreifen wollen (Abb. 9.7, Zeile 15 – 17).

Zuerst setzt die innere for-schleife für die lokale Variable y die Zahlenwerte in 15er-Schritten von 0 bis 390 ein, während der Wert für x noch 0 beträgt. Nun erhöht die äußere for-Schleife den x-Wert auf 1. Danach zählt die innere for-Schleife wieder in 15er-Schritten von 0 bis 390. Danach nimmt x den Wert 2 an und die innere for-Schleife zählt wieder in 15er-Schritten von 0 bis 390. Dies wird solange fortgesetzt, bis x den Wert 255 und y den Wert 390 erreicht hat. Welche Zahlenwerte ergeben sich nun für die Variable i, die die Behälter des Arrays durchnummeriert? Hier ein kleiner Auszug aus der sehr langen Zahlenkolonne.

```
x = 195 y = 360 i = 386700
x = 195 y = 375 i = 402780
x = 195 y = 390 i = 418860
x = 210 y = 0 i = 840
x = 210 y = 15 i = 16920
x = 210 y = 30 i = 33000
x = 210 y = 45 i = 49080
```

Diese Werte lassen sich mit der Formel  $let i = (x + y * width) * 4$  von Zeile 17 leicht überprüfen. Zum Beispiel für die Zahlenwerte x = 210 und y = 30:

```
let i = (x + y * width) * 4 = (210 + 30 * 268) * 4 = 33000
```

Was bedeutet nun diese Zahl 33000? Sie gibt die Nummer des Behälters in unserem eindimensionalen Pixelarray an, in dem der erste Wert für den Pixel des Ursprungsbildes mit den Werten x = 210 und y = 30 abgespeichert wurde. Da wir aber 4 Zahlenwerte für Rot, Grün, Blau und Alpha benötigen, schreiben wir innerhalb der for-Schleife i + 0, i + 1, i + 2 und i + 3 und ordnen mittels **pixels[]** den Variablen r, g, b und a die entsprechenden Zahlenwerte aus diesen Array-Behältern zu.

```
let r = pixels[i + 0];
let g = pixels[i + 1];
let b = pixels[i + 2];
let a = pixels[i + 3];
```

Diese vier Zahlenwerte setzen wir dann in die Funktion *fill()* ein, um das jeweilige Rechteck mit der Farbe des Ursprungsbildes zu füllen, die das Ursprungsbild an der Stelle besitzt, an der die x- und y-Werte des Rechteckes liegen. In unserem Beispiel: x = 210 und y = 30.

```
fill(r, g, b, a);
rect(x, y, 15, 15);
```

Jetzt bleibt noch zu klären, was die Funktion *pixelDensity(1)* bewirkt. Mit ihr passen wir die Auflösung des Bildes an die Auflösung unseres Monitors an (siehe Referenz von p5.js).

Mit dem Sketch von Abbildung 9.3 haben wir das Bild der Libelle 176-mal auf unsere Leinwand gezeichnet. Dies können wir natürlich auch mit dem Bild von dem Mann mit dem Bart machen. Interessant wird es jedoch, wenn wir die Sketche von Abbildung 9.3 und Abbildung 9.7 miteinander kombinieren, um so die Abbildung 9.9 zu erhalten. Auf den ersten Blick glauben wir, dass das verkleinerte Ursprungsbild sein Rasterbild nun selbst zeichnet. Das Ursprungsbild dient aber nur scheinbar als „Pixel“ (Rechteck) für das Rechteckbild. Dieser Effekt stellt sich erst dann ein, wenn wir

die vielen Einzelbilder mit dem großen, gerasterten Bild überdecken und den Alphawert halbieren (Zeile 26 in Abbildung 9.9). Erinnern wir uns. Je größer der Alphawert, desto größer ist die Deckkraft der Farbe.



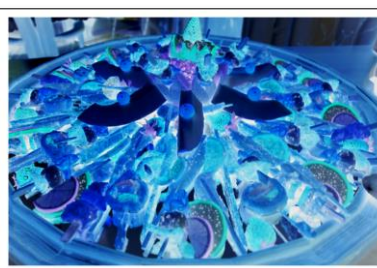
```

1 let bild;
2
3 function preload() {
4 bild = loadImage("Mann_mit_Bart.jpg");
5 }
6
7 function setup() {
8 createCanvas(268, 400);
9 background(200);
10 pixelDensity(1);
11 image(bild, 0, 0);
12 loadPixels();
13 noStroke();
14
15 for(let x = 0; x < width; x += 15) {
16 for(let y = 0; y < height; y += 22) {
17 let i = (x + y * width) * 4;
18
19 let r = pixels[i + 0];
20 let g = pixels[i + 1];
21 let b = pixels[i + 2];
22 let a = pixels[i + 3];
23
24 image(bild, x, y, 15, 22);
25
26 fill(r, g, b, a/2);
27 rect(x, y, 15, 22);
28 }
29 }
30 }
31

```

Abbildung 9.9: Rasterfüllung mit dem Ursprungsbild

p5.js bietet noch viele weitere Möglichkeiten der Bildbearbeitung. Wirf bitte hierzu unter dem Suchbegriff „Image“ einen Blick in die Referenz von p5.js und lasse deiner Kreativität freien Lauf. Abbildung 9.10 zeigt ein Beispiel. Doch Farben zu invertieren ist natürlich nicht besonders kreativ. Da hast du gewiss bessere Ideen.



```

1 let bild;
2
3 function preload(){
4 bild = loadImage("lecker.jpg");
5 }
6
7 function setup() {
8 createCanvas(600, 402);
9 image(bild, 0, 0);
10 filter(INVERT);
11 }
12

```

Abbildung 9.10: Die Farben des linken Fotos wurden mit einer Filterfunktion von p5.js invertiert

## 9.3 Zusammenfassung

Fassen wir zusammen, was wir in Kapitel 9 gelernt haben, bzw. gelernt haben sollten.

**function preload()** Wenn man eine größere Datei wie zum Beispiel eine Bild- oder Sounddatei in einem Sketch benutzen möchte, dann ist es für einen flüssigen Programmablauf sinnvoll, die Datei mittels *function preload()* schon vor *function setup()* aufzurufen. Mit *function preload()* können Bilddateien mit den Endungen `*.jpg`, `*.png` und `*.gif` oder Sounddateien mit den Endungen `„.wav“` oder `„.mp3“` hochgeladen werden. Voraussetzung hierfür ist jedoch, dass sich die Bilddatei oder Sounddatei im gleichen Ordner wie die Datei `index.html` und `sketch.js` befindet.

**loadImage()** Mit der Funktion *loadImage()* kann man ein mit *function preload()* hochgeladenes Bild einer Variablen zuordnen und somit für die weitere Bearbeitung im Sketch zur Verfügung stellen. Hier ein Beispiel.

```
1 var bild;
2
3 function preload(){
4 bild = loadImage("libelle.jpg");
5 }
6
7 function setup() {
8 createCanvas(800, 551);
9 image(bild, 0, 0, 250, 250);
10 }
11
```

**image()** Mit der Funktion *image()* bestimmt man die Position und die Größe des Bildes auf der Leinwand.

**Firefox** Schreibt man einen Sketch mit einer Bilddatei im Online-Editor von p5.js, so gelingt das Hochladen des Fotos nur, wenn es von einem Webserver geliefert wird. Wenn man keinen eigenen Webserver erstellen will, dann schreibt man seinen Sketch mit einem Offline-Editor und speichert ihn auf der Festplatte ab. Durch einen Doppelklick im Dateimanager auf `index.html` in dem entsprechenden Programmordner startet man den Sketch. Ein zufriedenstellendes Ergebnis erhält man jedoch nur, wenn man den Webbrowser *Firefox* verwendet. *Google Chrome* und *Internet Explorer* weigern sich lokal gespeicherte Fotos hochzuladen. Sie möchten Fotos von einem Webserver geliefert bekommen.

**box()** Mithilfe der Funktion *box()* kann man einen Quader zeichnen. Zwischen den Klammern kann man Höhe, Breite und Tiefe angeben.

**sphere()** Mithilfe der Funktion *sphere()* kann man eine Kugel zeichnen. Zwischen den Klammern gibt man den Radius der Kugel an.

**texture()** Mit der Funktion *texture()* kann man die Oberfläche von 3D-Objekten mit Fotos oder Videos versehen.

### loadPixels()

Mit der Funktion *loadPixels()* lädt man die Farbinformationen jedes Pixels des Ausgangsbildes in eine eindimensionales Array. Ein Pixel auf dem Monitor besteht normalerweise aus drei engbenachbarten Subpixeln in den Farben Rot, Grün und Blau. p5.js speichert für einen Pixel jedoch noch zusätzlich zu den drei Zahlenwerten für die drei Farben Rot, Grün und Blau (Werte von 0 – 255) den Alpha-Wert (0 – 255) für die Transparenz ab. Siehe hierzu auch *updatePixels()* in der Referenz auf der Homepage von p5.js.

### pixels[]

Mittels *pixels[]* kann man Variablen die Zahlenwerte aus dem ein-dimensionalen Pixelarray zuordnen. Beispiel:

```
let r = pixels[i + 0];
let g = pixels[i + 1];
let b = pixels[i + 2];
let a = pixels[i + 3];
```

### pixelDensity()

Mit *pixelDensity()* kann man die Auflösung des Bildes an die Auflösung des Monitors anpassen.

## 9.4 Übungen

Aufgabe 1 Erzeuge mit einem eigenen Foto, ein Bild entsprechend der Abbildung 9.4. Anstelle der 176 Teilbilder sollen auf der Leinwand jedoch nur 20 Teilbilder zu sehen sein.

Aufgabe 2 Verfremde dein eigenes Foto so wie in der folgenden Abbildung.



Abbildung 9.11: verfremdetes Bild

Aufgabe 3 Kombiniere zwei unterschiedliche Fotos zu einem Bild, ähnlich der folgenden Abbildung.

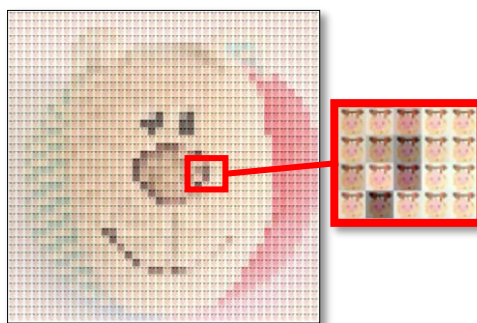


Abbildung 9.12: gerastertes Bild

Aufgabe 4 Mit der Funktion **tint()** kann man seinen Fotos eine andere Farbe geben. Probiere es aus. Wirf hierzu einen Blick in die Referenz von p5.js.



Abbildung 9.13: Flamingos in blau

Aufgabe 5 Lade ein eigenes Foto auf eine rotierende Kugel.



Abbildung 9.14: Stofftiergesicht auf einer Kugeloberfläche

## 10 Objektorientierte Programmierung

### 10.1 Warum einfach, wenn es auch komplizierter geht?

Wenn man sich die Sketche in diesem Kapitel **objektorientierte Programmierung (OOP)** anschaut, dann bekommt man erst einmal einen Schreck. Auf den ersten Blick sieht alles sehr kompliziert aus und man versteht nicht, wozu diese Art der Programmierung nützlich ist.

Aus diesen Gründen beginnen wir mit einem sehr einfachen Beispiel. Wir werden zuerst auf herkömmliche Weise, also ohne objektorientierte Programmierung, drei unterschiedliche Häuser zeichnen. Ein rotes Haus mit einem sehr spitzen Dach, ein grünes Haus mit einem normalen Dach und ein blaues Haus mit einem recht flachen Dach. Diese Aufgabe stellt uns vor keine große Herausforderung (siehe Sketch von Abbildung 10.1).

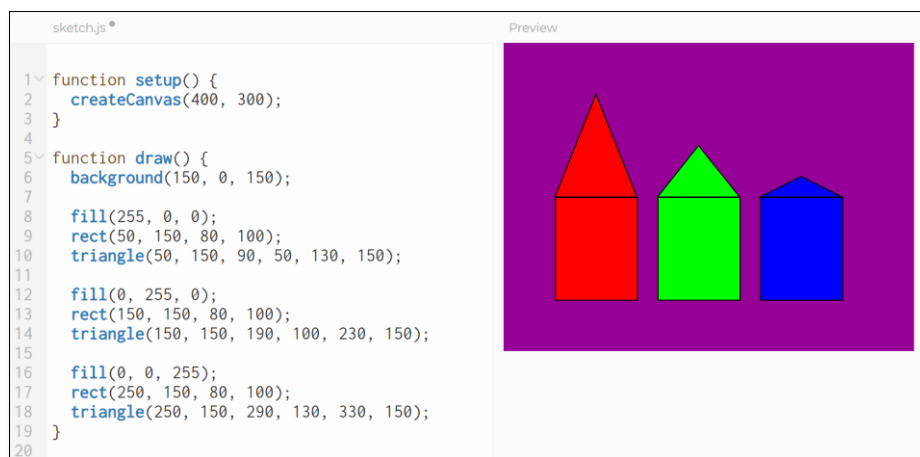


Abbildung 10.1: Drei Häuser ohne objektorientierte Programmierung gezeichnet

Lösen wir nun die gleiche Aufgabe mittels objektorientierter Programmierung, so wird der Sketch recht lang und unübersichtlich (Abb. 10.4) und wir denken zu Recht, dass diese Lösung für diese Aufgabenstellung nicht besonders sinnvoll ist. Doch anhand dieses einfachen Beispiels lassen sich die einzelnen Schritte der objektorientierten Programmierung einigermaßen leicht erklären. Schauen wir uns zuerst den letzten Sketchabschnitt von Abbildung 10.4, den Konstruktor an.

```
22 // KONSTRUKTOR
23 function Haus(tempX, tempY, tempD, tempR, tempG, tempB) {
24
25 this.x = tempX;
26 this.y = tempY;
27 this.d = tempD;
28 this.r = tempR;
29 this.g = tempG;
30 this.b = tempB;
31
32 // Methode
33 this.display = function() {
34 fill(this.r, this.g, this.b);
35 rect(this.x, this.y, 80, 100);
36 triangle(this.x, this.y, this.x + 40, this.y + this.d, this.x + 80, this.y);
37 }
38 }
39
```

Abbildung 10.2: Der Konstruktor dient als Bauplan zum Zeichnen der Häuser

Ein wesentlicher Bestandteil der objektorientierten Programmierung ist der Programmteil **Konstruktor** (Abb. 10.2). Mit dem Konstruktor erstellen wir einen Bauplan für Objekte wie zum Beispiel Häuser, Autos, ... . Zwischen den geschweiften Klammern der Konstruktor-Funktion, in unserem Beispiel *function Haus()*, steht die Funktion *this.display = function()* und zusätzlich noch mehrere Variablen (siehe Abb. 10.2), die die grundlegenden Eigenschaften der mit diesem Bauplan zu bauenden Häuser festlegen. Die Häuser bestehen, entsprechend ihrem Bauplan, stets aus einem Rechteck und einem Dreieck. Die Anzahl der Häuser, ihre Farbe und ihre Dachhöhe können außerhalb des Konstruktors frei gewählt werden.

Bei der objektorientierten Programmierung werden die Variablen mit **this**. und nicht wie bisher üblich mit *let* deklariert. Es ist üblich, diese Variablen als **Instanzvariablen** zu bezeichnen. Auch die hier verwendeten Funktionen bekommen einen anderen Namen. Man nennt sie **Methoden**, obwohl sie genauso funktionieren wie unsere bisher verwendeten Funktionen. Die Namen dieser Methoden werden stets klein geschrieben. Der Name der Konstruktor-Funktion *function Haus(tempX, tempY, tempD, tempR, tempG, tempB)* wird stets großgeschrieben. Zwischen den runden Klammern stehen die Größen, die anschließend den Instanzvariablen zugeordnet werden (Zeile 25 – 30 in Abb. 10.2). Die einzelnen Instanzvariablen haben die folgende Bedeutung. *this.x* und *this.y* sind die Ortskoordinaten für das jeweilige Rechteck und Dreieck. Mit *this.d* kann man die Dachhöhe einstellen und *this.r*, *this.g* und *this.b* stehen für die Farben des RGB-Farbsystem.

Widmen wir uns mit Abbildung 10.3 dem oberen Teil des vollständigen Sketchs (Abb. 10.4). In den Zeilen 1 bis 3 deklarieren wir ganz normal, also mit *let*, drei Häuser. Bei *function setup()* legen wir in den Zeilen 9 bis 11 fest, dass unsere Häuser nach dem Bauplan *Haus* gebaut werden sollen. Zwischen den runden Klammern tragen wir die Werte für die Ortskoordinaten, die Dachhöhe und die Farbe des jeweiligen Hauses ein. Bei *function draw()* rufen wir nun zum Zeichnen der drei Häuser die zugehörige Methode *display()* auf.

```

1 let hausRot;
2 let hausGruen;
3 let hausBlau;
4
5 function setup() {
6 createCanvas(400, 300);
7
8 // x, y, d, r, g, b
9 hausRot = new Haus(50, 150, -100, 255, 0, 0);
10 hausGruen = new Haus(150, 150, -50, 0, 255, 0);
11 hausBlau = new Haus(250, 150, -20, 0, 0, 255);
12 }
13
14 function draw() {
15 background(150, 0, 150);
16
17 hausRot.display();
18 hausGruen.display();
19 hausBlau.display();
20 }
21

```

Abbildung 10.3: Im Hauptteil des Sketches werden die Eigenschaften der Häuser entsprechend dem Bauplan festgelegt

Auf der folgenden Seite sehen wir nun den vollständigen Sketch (Abb. 10.4).

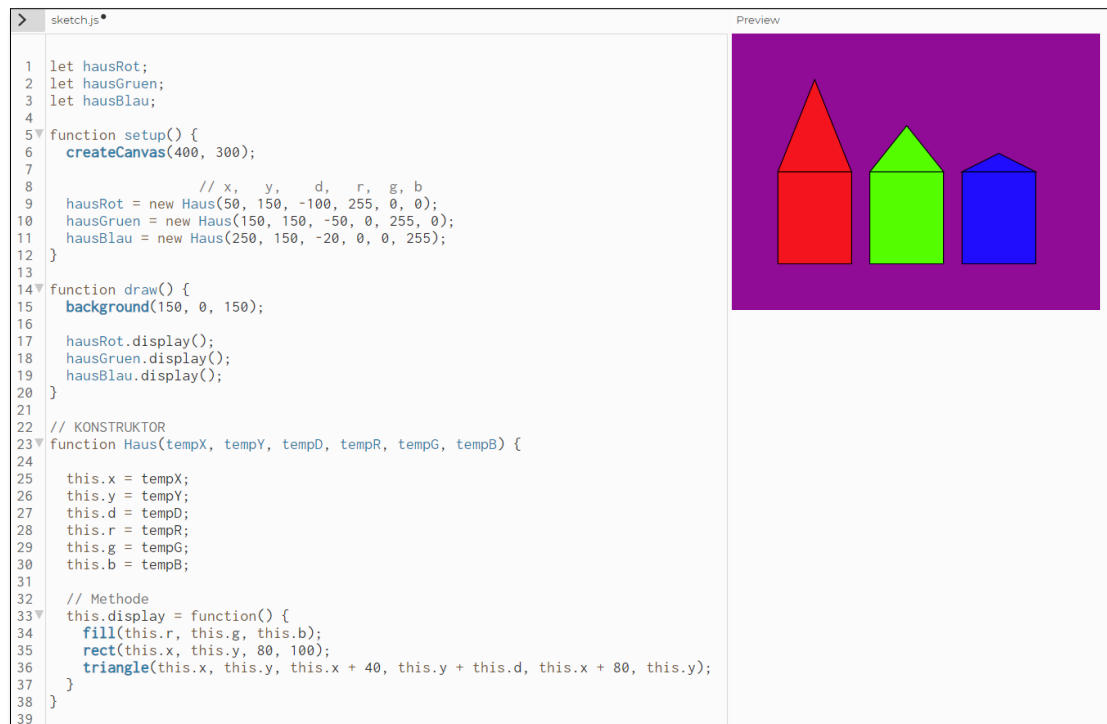


Abbildung 10.4: Mittels objektorientierter Programmierung gezeichnete Häuser

## 10.2 Häuser und Autos



Abbildung 10.5: Eine mit objektorientierter Programmierung gezeichnete Siedlung mit Autoverkehr.

Nun wollen wir mittels objektorientierter Programmierung die Abbildung 10.5 erstellen. Die Häuser sind nun dreidimensional und auf der Straße vor den Häusern fahren zahlreiche Autos vorbei. Bei dieser etwas anspruchsvolleren Aufgabe kann die objektorientierte Programmierung nun auch sinnvoll eingesetzt werden. Die Lösung dieser Aufgabe sieht wie folgt aus (Abb. 10.6).

```

1 let haus = [5];
2 let auto = [5];
3
4 function setup() {
5 // r, g, b, f, zx, zy
6 haus[0] = new Haus(240, 255, 255, 2, 0, 0);
7 haus[1] = new Haus(155, 155, 155, 1.5, 200, 100);
8 haus[2] = new Haus(250, 200, 200, 1.7, 350, 60);
9 haus[3] = new Haus(240, 255, 0, 1.0, 550, 200);
10 haus[4] = new Haus(240, 255, 255, 0.5, 700, 180);
11 // zx, zy, r, g, b, v
12 auto[0] = new Auto(-200, 0, 255, 0, 0, 1);
13 auto[1] = new Auto(200, 30, 255, 255, 0, 1.5);
14 auto[2] = new Auto(-500, 50, 255, 0, 255, 2);
15 auto[3] = new Auto(400, 80, 255, 255, 255, 2.5);
16 auto[4] = new Auto(-300, 110, 0, 255, 255, 3);
17 }
18
19 function draw() {
20 createCanvas(800, 600);
21 background(220, 240, 255);
22 noStroke();
23 fill(0, 180, 0);
24 rect(0, 230, 800, 210);
25 fill(100);
26 rect(0, 440, 800, 600);
27
28 for(i = 0; i < 5; i++){
29 for(j = 0; j < 5; j++){
30 haus[i].display();
31 auto[j].display();
32 auto[j].move();
33 }
34 }
35 }
36
37 // KONSTRUKTOR Haus
38 function Haus(tempR, tempG, tempB, tempF, tempZx, tempZy){
39 this.r = tempR;
40 this.g = tempG;
41 this.b = tempB;
42 this.f = tempF;
43 this.zx = tempZx;
44 this.zy = tempZy;
45
46 // Methode
47 this.display = function(){
48 // Vorderfront
49 stroke(this.r, 0*this.g, 0*this.b);
50 strokeWeight(4);
51 line(this.f*20+this.zx, this.f*140+this.zy, this.f*60+this.zx, this.f*96+this.zy);
52 noStroke();
53 fill(this.r, this.g, this.b)
54 beginShape();
55 vertex(this.f*20+this.zx, this.f*200+this.zy);
56 vertex(this.f*20+this.zx, this.f*140+this.zy);
57 vertex(this.f*60+this.zx, this.f*96+this.zy);
58 vertex(this.f*100+this.zx, this.f*140+this.zy);
59 vertex(this.f*100+this.zx, this.f*200+this.zy);
60 endShape(CLOSE);
61
62 // Dach
63 fill(this.r, 0*this.g, 0*this.b);
64 beginShape();
65 vertex(this.f*60+this.zx, this.f*96+this.zy);
66 vertex(this.f*90+this.zx, this.f*76+this.zy);
67 vertex(this.f*126+this.zx, this.f*116+this.zy);
68 vertex(this.f*100+this.zx, this.f*140+this.zy);
69 endShape(CLOSE);
70
71 // Seite
72 fill(0.8*this.r, this.g, this.b);
73 beginShape();
74 vertex(this.f*100+this.zx, this.f*200+this.zy);
75 vertex(this.f*100+this.zx, this.f*140+this.zy);
76 vertex(this.f*126+this.zx, this.f*116+this.zy);
77 vertex(this.f*126+this.zx, this.f*166+this.zy);
78 endShape(CLOSE);
79 }
80 }
81
82 // KONSTRUKTOR Auto
83 function Auto(tempZx, tempZy, tempR, tempG, tempB, tempV){
84 this.zx = tempZx;
85 this.zy = tempZy;
86 this.r = tempR;
87 this.g = tempG;
88 this.b = tempB;
89 this.v = tempV;
90
91 // METHODEN
92 this.display = function(){
93 //Farbe
94 fill(this.r, this.g, this.b);
95 // Räder
96 ellipse(-190+this.zx, 450+this.zy, 30, 30);
97 ellipse(-30+this.zx, 450+this.zy, 30, 30);
98 // Karosserie
99 beginShape();
100 vertex(-225+this.zx, 450+this.zy);
101 vertex(-200+this.zx, 375+this.zy);
102 vertex(-100+this.zx, 375+this.zy);
103 vertex(-67+this.zx, 415+this.zy);
104 vertex(-15+this.zx, 415+this.zy);
105 vertex(0+this.zx, 450+this.zy);
106 endShape(CLOSE);
107 }
108
109 this.move = function(){
110 this.zx = this.zx + this.v;
111 if(this.zx > width + 400){
112 this.zx = 0;
113 }
114 }
115 }
116

```

Abbildung 10.6: Der zur Abbildung 10.5 gehörige Sketch

Der Sketch für die Erstellung von Abbildung 10.5 ist erschreckende 115 Zeilen lang (siehe Abb. 10.6). Mittels objektorientierter Programmierung sollten die Sketche aber eigentlich übersichtlicher und somit auch verständlicher werden. Wie dies gelingt, soll nun erklärt werden. Der Sketch von Abbildung 10.6 besteht aus drei Teilen. Dem Hauptteil mit *function setup()* und *function draw()* sowie dem Konstruktor *Haus* und dem Konstruktor *Auto*. Schauen wir uns zuerst den *Konstruktor Haus* an (Abb. 10.7).

```

37 // KONSTRUKTOR Haus
38 function Haus(tempR, tempG, tempB, tempF, tempZx, tempZy){
39 this.r = tempR;
40 this.g = tempG;
41 this.b = tempB;
42 this.f = tempF;
43 this.zx = tempZx;
44 this.zy = tempZy;
45
46 // Methode
47 this.display = function(){
48 // Vorderfront
49 stroke(this.r, 0*this.g, 0*this.b);
50 strokeWeight(4);
51 line(this.f*20+this.zx, this.f*140+this.zy, this.f*60+this.zx, this.f*96+this.zy);
52 noStroke();
53 fill(this.r, this.g, this.b)
54 beginShape();
55 vertex(this.f*20+this.zx, this.f*200+this.zy);
56 vertex(this.f*20+this.zx, this.f*140+this.zy);
57 vertex(this.f*60+this.zx, this.f*96+this.zy);
58 vertex(this.f*100+this.zx, this.f*140+this.zy);
59 vertex(this.f*100+this.zx, this.f*200+this.zy);
60 endShape(CLOSE);
61
62 // Dach
63 fill(this.r, 0*this.g, 0*this.b);
64 beginShape();
65 vertex(this.f*60+this.zx, this.f*96+this.zy);
66 vertex(this.f*90+this.zx, this.f*76+this.zy);
67 vertex(this.f*126+this.zx, this.f*116+this.zy);
68 vertex(this.f*100+this.zx, this.f*140+this.zy);
69 endShape(CLOSE);
70
71 // Seite
72 fill(0.8*this.r, this.g, this.b);
73 beginShape();
74 vertex(this.f*100+this.zx, this.f*200+this.zy);
75 vertex(this.f*100+this.zx, this.f*140+this.zy);
76 vertex(this.f*126+this.zx, this.f*116+this.zy);
77 vertex(this.f*126+this.zx, this.f*166+this.zy);
78 endShape(CLOSE);
79 }
80 }
81

```

Abbildung 10.7: Konstruktor *Haus* als Ausschnitt aus dem vollständigen Sketch

Der *Konstruktor Haus* erzeugt mit den drei Flächen *Vorderfront*, *Dach* und *Seite* einen Bauplan für dreidimensionale Häuser. Mit der Instanzvariablen *f* (Faktor) kann die Größe der einzelnen Häuser eingestellt werden. Mit *zx* und *zy* können die Häuser in *x*- und *y*-Richtung verschoben werden. Mit *r*, *g* und *b* wählt man die Farbe des jeweiligen Hauses. Die Methode *this.display = function()* sorgt für die Darstellung der Häuser. Sie wird im Hauptteil des Sketches aufgerufen.

Schauen wir uns nun den *Konstruktor Auto* an (Abb. 10.8).

```

82 // KONSTRUKTOR Auto
83 function Auto(tempZx, tempZy, tempR, tempG, tempB, tempV){
84 this.zx = tempZx;
85 this.zy = tempZy;
86 this.r = tempR;
87 this.g = tempG;
88 this.b = tempB;
89 this.v = tempV;
90
91 // METHODEN
92 this.display = function(){
93 //Farbe
94 fill(this.r, this.g, this.b);
95 // Räder
96 ellipse(-190+this.zx, 450+this.zy, 30, 30);
97 ellipse(-30+this.zx, 450+this.zy, 30, 30);
98 // Karosserie
99 beginShape();
100 vertex(-225+this.zx, 450+this.zy);
101 vertex(-200+this.zx, 375+this.zy);
102 vertex(-100+this.zx, 375+this.zy);
103 vertex(-67+this.zx, 415+this.zy);
104 vertex(-15+this.zx, 415+this.zy);
105 vertex(0+this.zx, 450+this.zy);
106 endShape(CLOSE);
107 }
108
109 this.move = function(){
110 this.zx = this.zx + this.v;
111 if(this.zx > width + 400){
112 this.zx = 0;
113 }
114 }
115 }
116

```

Abbildung 10.8: Konstruktor Auto als Ausschnitt aus dem vollständigen Sketch

Im Konstruktor *Auto* sorgt die Methode *this.display = function()* für die Darstellung der Farben, der Räder und der Karosserie jedes Autos. Eine weitere Methode, *this.move = function()*, befindet sich bei Zeile 109 bis 114 von Abbildung 10.8. Sie sorgt für die Bewegung der Autos. Die Geschwindigkeit der Autos kann mit der Instanzvariablen *this.v* (*v* steht für *velocity* = Geschwindigkeit) verändert werden. Die in der Methode enthaltene *if*-Anweisung sorgt dafür, dass die Autos am linken Bildrand wieder erscheinen, wenn sie den rechten Bildrand überschritten haben. Die Methoden *this.display = function()* und *this.move = function()* werden im Hauptteil des Sketches aufgerufen.

Schauen wir uns nun den Hauptteil des Sketches an (Abb. 10.9).

```

1 let haus = [5];
2 let auto = [5];
3
4 function setup() {
5 // r, g, b, f, zx, zy
6 haus[0] = new Haus(240, 255, 255, 2, 0, 0);
7 haus[1] = new Haus(155, 155, 155, 1.5, 200, 100);
8 haus[2] = new Haus(250, 200, 200, 1.7, 350, 60);
9 haus[3] = new Haus(240, 255, 0, 1.0, 550, 200);
10 haus[4] = new Haus(240, 255, 255, 0.5, 700, 180);
11 // zx, zy, r, g, b, v
12 auto[0] = new Auto(-200, 0, 255, 0, 0, 1);
13 auto[1] = new Auto(200, 30, 255, 255, 0, 1.5);
14 auto[2] = new Auto(-500, 50, 255, 0, 255, 2);
15 auto[3] = new Auto(400, 80, 255, 255, 255, 2.5);
16 auto[4] = new Auto(-300, 110, 0, 255, 255, 3);
17 }
18
19 function draw() {
20 createCanvas(800, 600);
21 background(220, 240, 255);
22 noStroke();
23 fill(0, 180, 0);
24 rect(0, 230, 800, 210);
25 fill(100);
26 rect(0, 440, 800, 600);
27
28 for(i = 0; i < 5; i++){
29 for(j = 0; j < 5; j++){
30 haus[i].display();
31 auto[j].display();
32 auto[j].move();
33 }
34 }
35 }
36

```

Abbildung 10.9: Der Hauptteil des Sketches

Im Hauptteil unseres Sketches deklarieren wir in Zeile 1 und 2 jeweils ein Array für fünf Häuser und fünf Autos. Bei *function setup()* versehen wir sie mit ihren individuellen Eigenschaften wie Farbe, Ortskoordinaten, ... . Bei *function draw()* zeichnen wir zuerst die grüne Wiese und die Straße und rufen anschließend die Methoden auf. Da wir in Zeile 1 und 2 des Hauptsketches zwei Arrays deklariert haben, können wir die einzelnen Häuser und Autos mittels des Indexes *i* oder *j* kennzeichnen. Dies ist sehr praktisch, wenn wir sehr viele Häuser und Autos nach den Bauplänen der beiden Konstruktoren bauen wollen.

Nun lösen wir unser Versprechen ein, dass objektorientierte Programmierung eine feine Sache ist. Wenn wir einmal mittels eines Konstruktors einen Bauplan für Häuser, Autos, ... erstellt haben, dann können wir diese Pläne auch in anderen Sketchen benutzen. Dazu müssen wir sie im Ordner *empty-example* in einem eigenen Ordner abspeichern und die HTML-Datei wie folgt ergänzen:

```
<script src="haus.js"></script>
<script src="auto.js"></script>
```

Siehe hierzu Abbildung 10.10 und 10.11 (Zeile 11 und 12). Hiernach genügt es den Hauptteil unseres Sketches aufzurufen, um den Sketch zu starten.

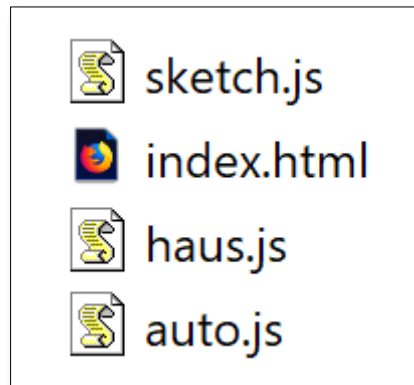


Abbildung 10.10: Inhalt des Ordners *empty-example*.

```
sketch.js x auto.js x haus.js x index.html o
1 <!DOCTYPE html>
2 <html>
3 <head>
4 <meta charset="utf-8">
5 <meta name="viewport" width=device-width, initial-scale=1.0, maximum-scale=1.0, user-scalable=0>
6 <style> body {padding: 0; margin: 0;} </style>
7 <script src="..p5.min.js"></script>
8 <script src="..addons/p5.dom.min.js"></script>
9 <script src="..addons/p5.sound.min.js"></script>
10 <script src="sketch.js"></script>
11 <script src="haus.js"></script>
12 <script src="auto.js"></script>
13 </head>
14 <body>
15 </body>
16 </html>
17
```

Abbildung 10.11: Durch Zeile 11 und 12 ergänzte HTML-Datei.

Wenn man keinen Account auf der p5.js-Homepage eingerichtet hat, dann muss man die oben beschriebenen Vorgänge offline mittels eines Code-Editors (z.B. Atom) durchführen. Den Sketch startet man dann im Dateimanager durch einen Doppelklick auf index.html.

Hier noch ein Tipp. Oft spart man viel Zeit, wenn man sich mit Bleistift und Papier vorab konkrete Gedanken zu dem Projekt macht, welches man durch objektorientierte Programmierung verwirklichen will (siehe Abb. 10.12). Die in Abbildung 10.12 aufgeführten Zahlen sind Millimeterangaben. Sie dienen als Hilfestellung für die Findung der entsprechenden x- und y-Werte für die Pixelangaben.

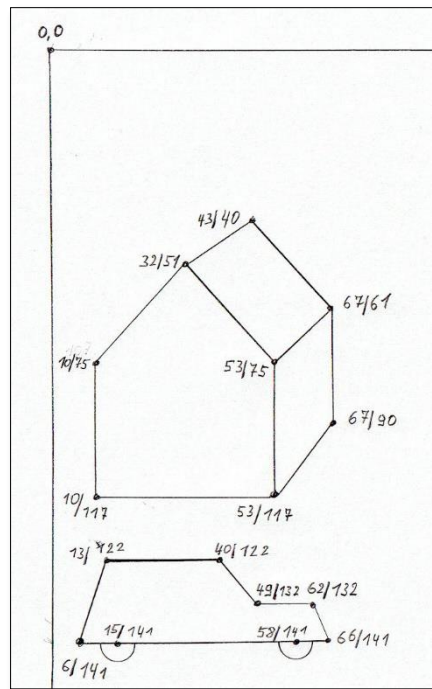


Abbildung 10.12: Vorüberlegung zu dem Sketch von Abbildung 10.6

### 10.3 Zusammenfassung

Fassen wir zusammen, was wir in Kapitel 10 gelernt haben, bzw. gelernt haben sollten.

Bei der objektorientierten Programmierung erstellt man im **Konstruktor** mittels Variablen und Funktionen einen Bauplan für Objekte. Objekte können zum Beispiel Häuser, Autos, ... sein, die auf der Leinwand gezeichnet werden. Eine Variable, die zum Konstruktor gehört, nennt man **Instanzvariable**. Eine Funktion, die zum Konstruktor gehört, nennt man **Methode**. Die Variablen deklariert man im Konstruktor nicht mit `let`, sondern mit `this..` Beispiel: Anstelle von `let x` schreibt man `this.x`

Im Sketchteil außerhalb des Konstruktors kann man nach dem im Konstruktor erstellten Bauplan beliebig viele Objekte erschaffen und ihnen im Rahmen des Bauplans unterschiedliche Eigenschaften zuordnen. Hier ruft man auch die Methoden auf. Ein einfaches Beispiel stellt der Sketch von Abbildung 10.4 im Kapitel 10 dar.

Mittels objektorientierter Programmierung können große Sketche in übersichtliche Blöcke unterteilt werden. Weiterhin können die erstellten Baupläne auch in anderen Sketchen benutzt werden.

## 10.4 Übungen

- Aufgabe 01 Schreibe mittels objektorientierter Programmierung einen Sketch, mit dem man die abgebildeten vier Smileys mit Kulleraugen und unterschiedlicher Blickrichtung zeichnen kann.

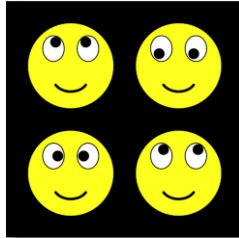


Abbildung 10.13: Smileys mit Kulleraugen

- Aufgabe 02 Die fünf verschieden gefärbten Pac-Mans von Abbildung 10.14 bewegen sich unterschiedlich schnell zwischen den seitlichen Begrenzungen der Leinwand hin und her. Bei jeder Richtungsumkehr zeigt ihr Mund stets in die Bewegungsrichtung. Die Münder der einzelnen Pac-Mans schließen sich unterschiedlich schnell. Löse diese Aufgabe mittels objektorientierter Programmierung.

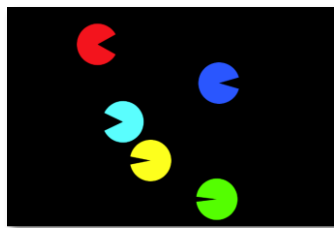


Abbildung 10.14: Sich hin und her bewegende Pac-Mans

- Aufgabe 03 Die hier dargestellten vier Windräder besitzen unterschiedliche Größen und Farben und drehen sich unterschiedlich schnell. Alle wurden nach nur einem Bauplan gezeichnet, der mittels objektorientierter Programmierung erstellt wurde. Schreibe einen entsprechenden Sketch.

Tipp: Mit der Funktion **scale()** kann man Objekte sehr einfach vergrößern oder verkleinern (siehe Referenz). Die Funktionen *push()* und *pop()* können bei der Erstellung des Sketches hilfreich sein.

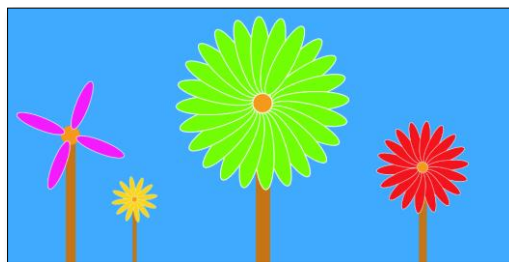


Abbildung 10.15: Vier sich drehende Windräder

# 11 Rekursive Programmierung

## 11.1 Punkte und Quadrate

Manche grafischen Aufgaben kann man mittels **rekursiver Programmierung** sehr elegant lösen. Der Nachteil ist jedoch, dass rekursive Programme nicht immer 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 von Abbildung 11.1. Hier wird ein Punkt immer kleiner und ändert dabei seine Farbe. Der erste Punkt, der gezeichnet wird, hat die Farbe Weiß und einen Durchmesser von 350 Pixeln (siehe Zeile 10 von Abbildung 11.1). Beim nächsten Durchlauf von *function draw()* verringert sich der Durchmesser *d* des Punktes um 40 Pixel und die Farbe ändert sich von Weiß in Richtung Orange. D.h., der Zahlenwert für Rot bleibt bei 255, der Zahlenwert für Grün wird bei jedem Durchlauf um 20 verringert, während der Wert für Blau doppelt so stark, also um 40 verringert wird (Zeile 22).

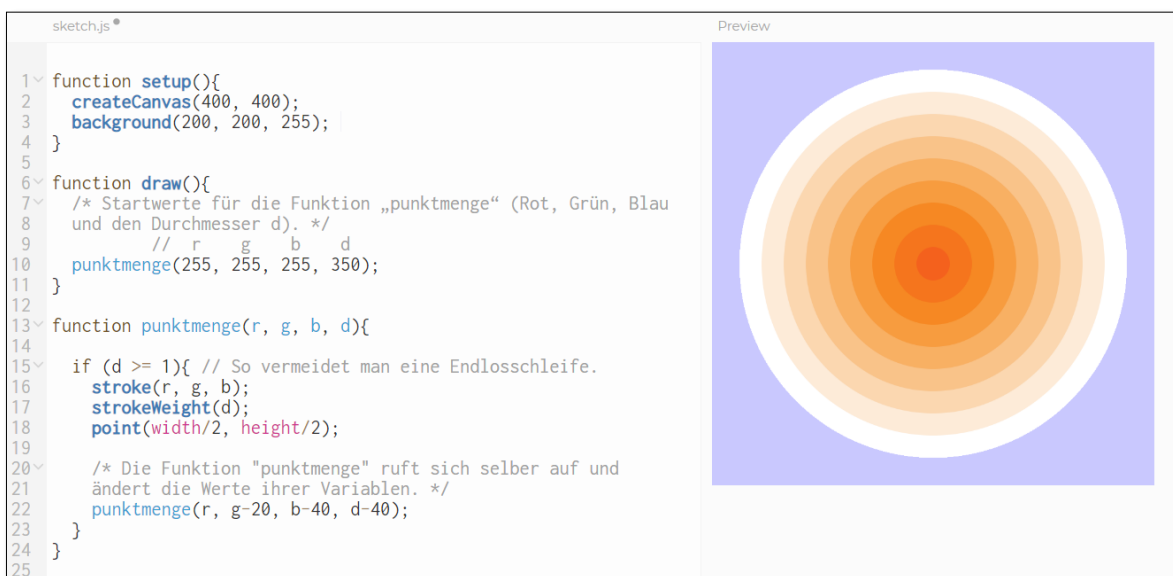


Abbildung 11.1: Ein Punkt wird immer kleiner und ändert seine Farbe

Natürlich hätten wir die Abbildung 11.1 auch mittels einer einfachen Schleife erzeugen können, doch hätten wir dies auch für den folgenden Sketch von Abbildung 11.2 geschafft? Erklärt wird der Sketch durch die eingefügten Kommentare in Abbildung 11.2.

Zur Vorbereitung auf das nächste Kapitel soll die rekursive Programmierung anhand eines weiteren, einfachen Sketches erklärt werden. In dem Sketch von Abbildung 11.3 werden vier Quadrate gezeichnet, die ihre Größe und Farbe ändern und an deren linken oberen Eckpunkt sich ein roter Punkt befindet. Wenn man das Prinzip der rekursiven Programmierung verstanden hat, dann kann man sich die *x*- und *y*-Werte des linken oberen Eckpunktes und die Breite und Höhe der Quadrate ausrechnen. Die entsprechenden Werte sind in der Tabelle unter der Abbildung 11.3 aufgeführt. Mit dem Mauszeiger (Zeile 24 - 28) kann man diese Werte überprüfen.

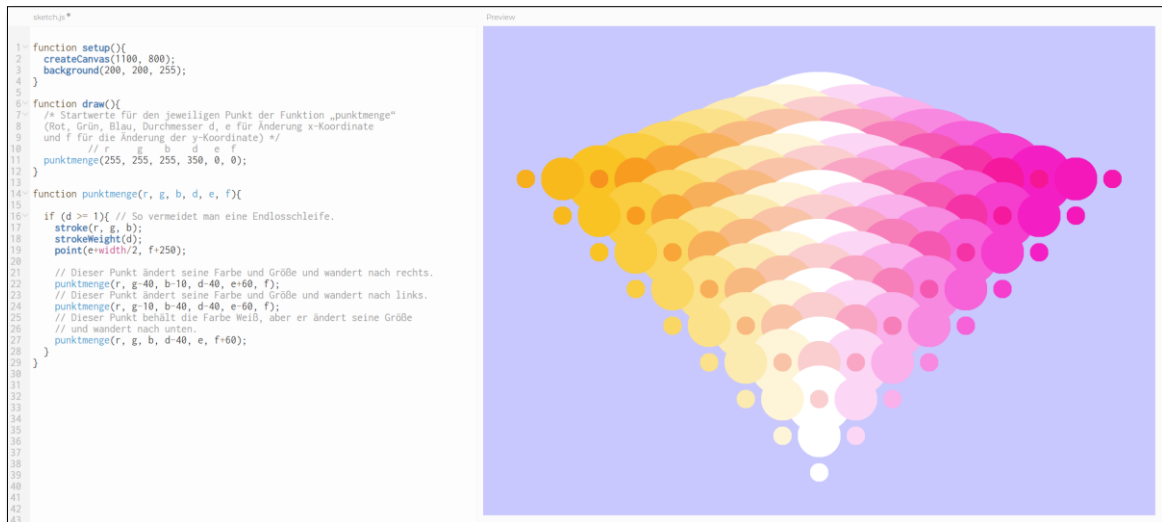


Abbildung 11.2: Punkte ändern ihre Farbe, ihre Größe und ihren Ort

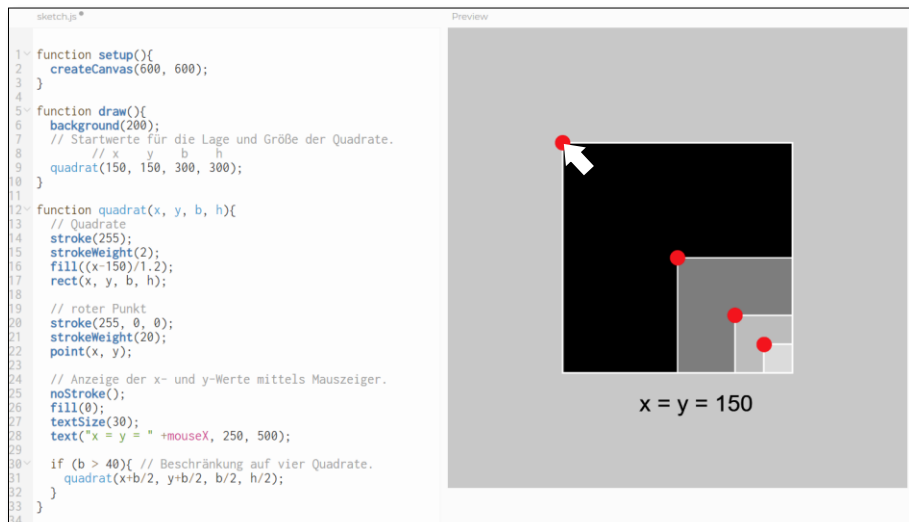


Abbildung 11.3: Quadrate werden kleiner und ändern ihre Farbe von Schwarz zu Hellgrau

Tabelle zur Berechnung der Werte für die einzelnen Quadrate.

| linker oberer Eckpunkt |                      | Breite                      | Höhe                        |
|------------------------|----------------------|-----------------------------|-----------------------------|
| x                      | y                    | b                           | h                           |
| 150                    | 150                  | 300                         | 300                         |
| $150 + 150 = 300$      | $150 + 150 = 300$    | $b/2 = 300/2 = 150$         | $h/2 = 300/2 = 150$         |
| $300 + 75 = 375$       | $300 + 75 = 375$     | $(b/2)/2 = 150/2 = 75$      | $(h/2)/2 = 150/2 = 75$      |
| $375 + 37,5 = 412,5$   | $375 + 37,5 = 412,5$ | $((b/2)/2)/2 = 75/2 = 37,5$ | $((h/2)/2)/2 = 75/2 = 37,5$ |

Es lohnt sich, die Rechnungen in der obigen Tabelle nachzuvollziehen, denn dann versteht man die im folgenden Kapitel aufgeführten Sketche wesentlich besser.

## 11.2 Fraktale

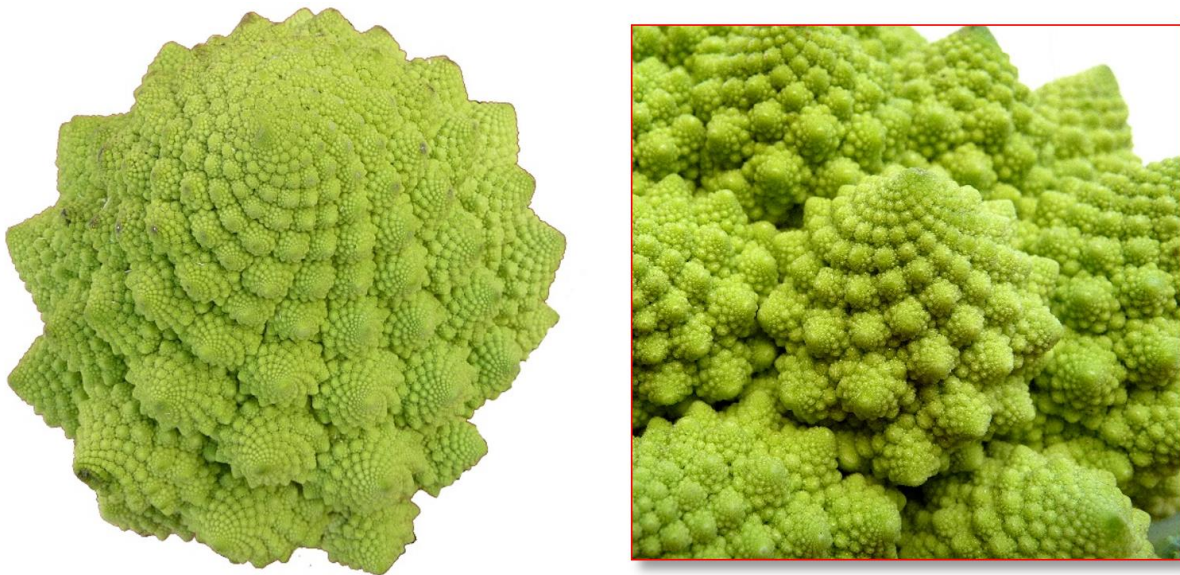


Abbildung 11.4: Romanesco, eine Variante des Blumenkohls rechts: Nahaufnahme

Die rekursive Programmierung eignet sich gut zur Darstellung von Fraktalen, also selbstähnlichen Objekten. Abbildung 11.4 links zeigt ein natürliches Fraktal, eine Variante des Blumenkohls, den Romanesco. Schaut man genauer hin (Abb. 11.4 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. Die Programmierung von fraktalen Strukturen eröffnet uns eine weitere Möglichkeit zur künstlerischen Gestaltung.

Als Beispiel für die Erstellung eines Fraktals mithilfe der rekursiven Programmierung wollen wir Quadrate auf eine 600 Pixel x 600 Pixel große Leinwand zeichnen. Das erste Quadrat soll so groß wie die Leinwand sein, also 600 Pixel x 600 Pixel groß (rotes Quadrat in Abbildung 11.5). Im nächsten Schritt sollen innerhalb des roten Quadrates drei Quadrate mit halber Seitenlänge gezeichnet werden. Diese drei Quadrate (hellblau in Abb. 11.5) sollen, wie in der Abbildung 11.5 dargestellt, versetzt zueinander gezeichnet werden. In die drei hellblauen 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 Quadrate gezeichnet. Usw., usw.. Damit keine Endlosschleife entsteht, setzen wir die minimale Seitenlänge der Quadrate auf 1 Pixel. Zum besseren Verständnis schaue man sich noch einmal den Sketch von Abbildung 11.3 an.

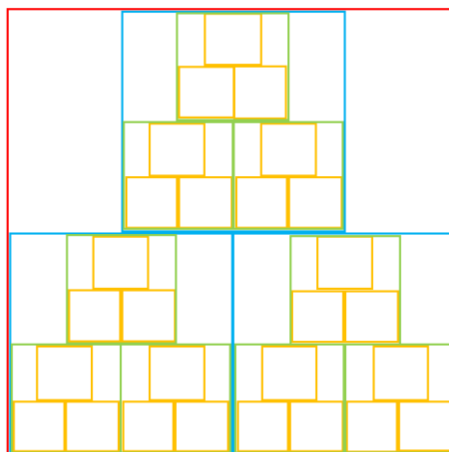


Abbildung 11.5: In jedes neu entstehende Quadrat werden drei neue Quadrate mit der halben Seitenlänge eingefügt

Um es nicht zu kompliziert zu machen, verzichten wir in dem Sketch von Abbildung 11.6 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. 11.6). Die im Sketch gewählte Iterationsvorschrift führt zu einem Attraktor, dem Sierpinski-Dreieck. Aufgrund seiner Selbstähnlichkeit ist das Sierpinski-Dreieck ein fraktales Gebilde.

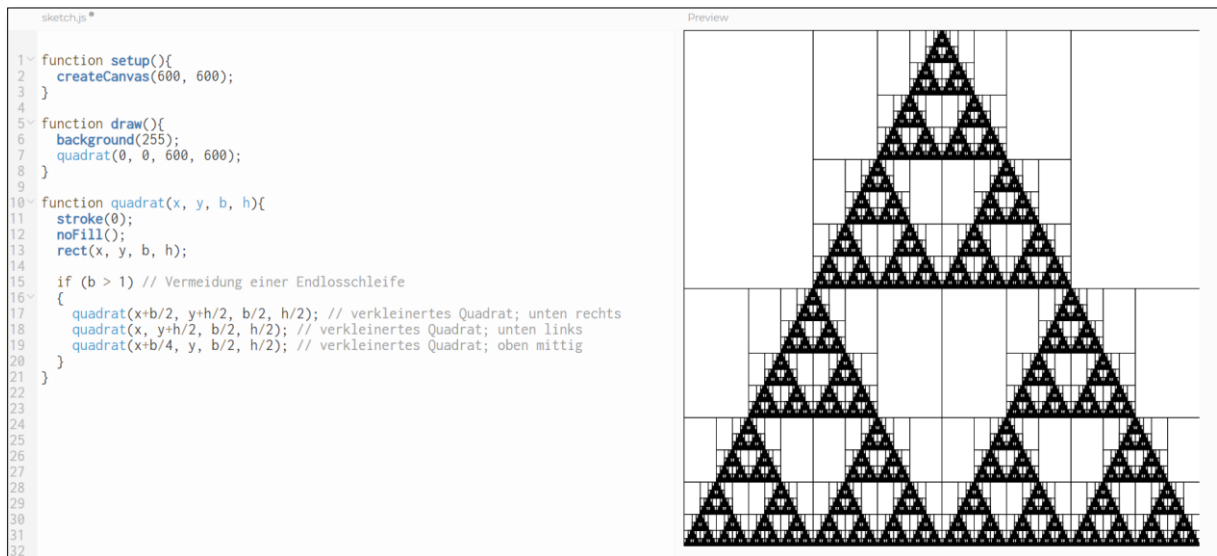


Abbildung 11.6: Die Iterationsvorschrift sorgt dafür, dass aus immer kleiner werdenden Quadraten ein Sierpinski-Dreieck entsteht

Möchte man ein Sierpinski-Dreieck ohne die störenden Linien der Quadrate zeichnen, dann muss man die Quadrate von Sketch 11.6 durch Dreiecke ersetzen (siehe Zeile 16 – 18 von Abbildung 11.7). Mittels `mouseX` in Zeile 15 kann man sich über die Mausbewegung in x-Richtung den Aufbau des Sierpinski-Dreieckes Schritt für Schritt anzeigen lassen (siehe Abb. 11.8).

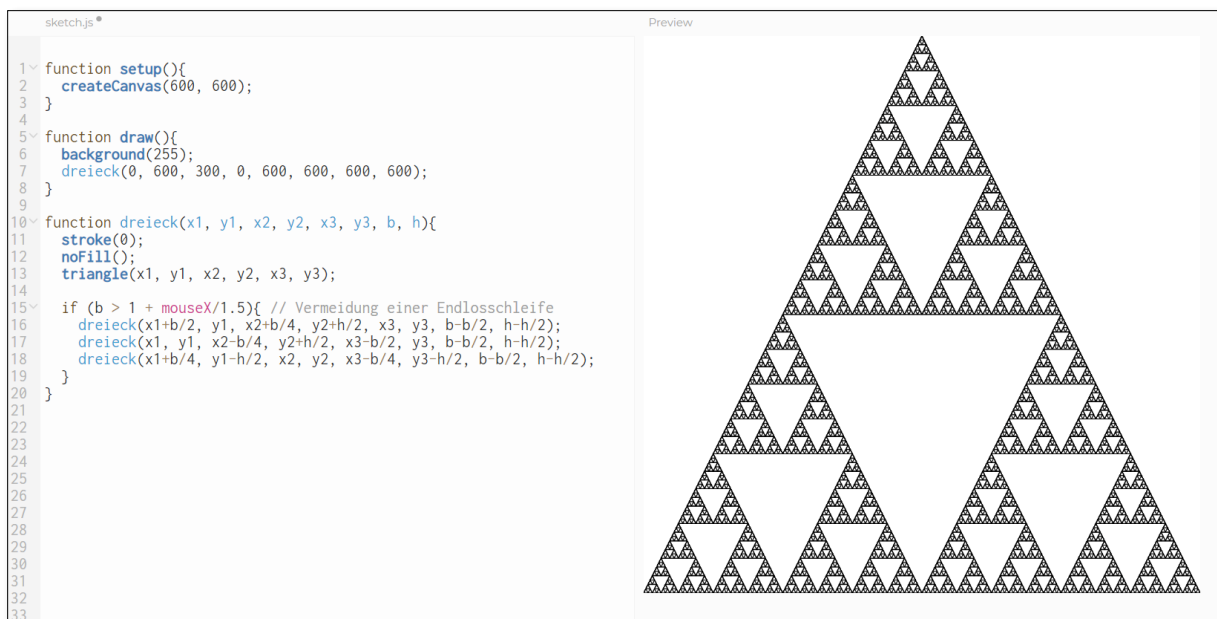


Abbildung 11.7: Ein aus Dreiecken aufgebautes Sierpinski-Dreieck

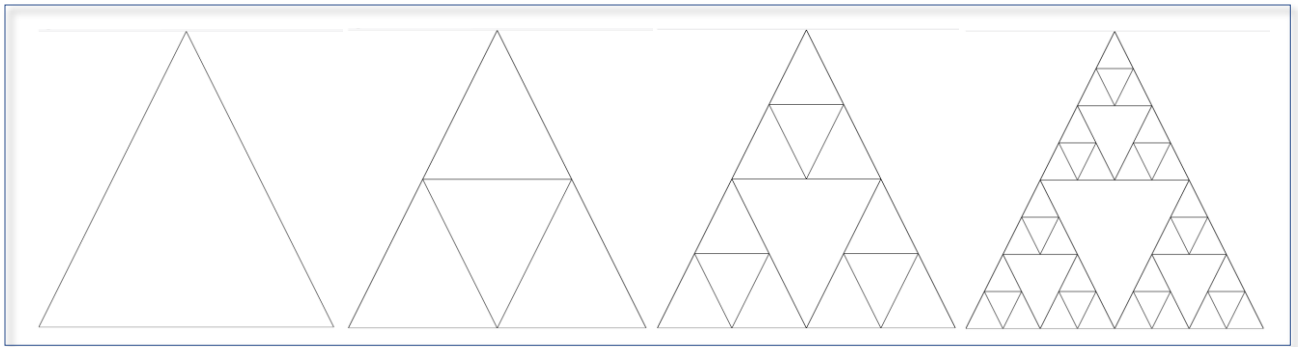


Abbildung 11.8: Entstehung des Sierpinski-Dreiecks

Wenden wir nun das Gelernte auf eine Bildgestaltung an. Aus einem beliebigen Bild können wir unter Verwendung der oben gezeigten Iterationsmethode ein Sierpinski-Dreieck erstellen (Abb. 11.9 und 11.10). Beachten sollte man, dass man die Sketche, die Fotos enthalten, mit dem Webeditor von p5.js erstellen und öffnen muss, wenn nicht über einen lokalen Server verfügt (siehe Kapitel 9.1).



Abbildung 11.9: Die Iterationsvorschrift sorgt dafür, dass auch aus immer kleiner werdenden Bildern ein Sierpinski-Dreieck entsteht

```

1 let bild;
2
3 function preload() {
4 bild = loadImage("haarfrau.jpg");
5 }
6
7 function setup() {
8 createCanvas(600, 600);
9 }
10
11 function draw() {
12 background(255);
13 quadrat(0, 0, 600, 600);
14 }
15
16 function quadrat(x, y, b, h) {
17 stroke(0);
18 noFill();
19 image(bild, x, y, b, h);
20
21 if (b > 1+ mouseX/1.5) // Vermeidung einer Endlosschleife
22 { quadrat(x+b/2, y+h/2, b/2, h/2); // verkleinertes Quadrat; unten rechts
23 quadrat(x, y+h/2, b/2, h/2); // verkleinertes Quadrat; unten links
24 quadrat(x+b/4, y, b/2, h/2); // verkleinertes Quadrat; oben mittig
25 }
26 }
27

```

Abbildung 11.10: Sketch für die Erstellung von Abbildung 10.9

Ein Bild, welches sich selbst wieder als Bild enthält, ist ein Verfahren, dass in der Kunst als „Mise en abyme“ bekannt ist. Mit Hilfe der rekursiven Programmierung können auch wir Bilder erzeugen, die sich selbst wieder als Bild enthalten und somit fraktale Eigenschaften besitzen (siehe Abb. 11.11).



Abbildung 11.11: Bild im Bild im Bild im Bild im ...

Um ein solches Bild (Abb. 11.11) zu erstellen, ist jedoch einiges an Vorüberlegungen erforderlich. Wenn wir das Foto der Dame, die den goldenen Bilderrahmen hält, genau auf die Fläche innerhalb des Bilderrahmens projizieren wollen, dann muss das Seitenverhältnis des Fotos mit dem Seitenverhältnis der Fläche im Bilderrahmen übereinstimmen. Mit einem Zollstock kann man die Breite und die Höhe der Fläche im Bilderrahmen messen und so das Seitenverhältnis berechnen. Das anschließend von der Szene zu machende Foto sollte jedoch einen größeren Bereich erfassen, als in der Abbildung 11.11 dargestellt, damit man mittels eines Bildbearbeitungsprogramms das Foto so beschneiden kann, dass sich das gleiche Seitenverhältnis ergibt wie bei der Fläche innerhalb des Bilderrahmens. Im gleichen Seitenverhältnis erstellen wir bei `function setup()` die Größe unserer Leinwand, damit das Foto diese genau ausfüllt. Im Sketch von Abbildung 11.12 steht in Zeile 8 deshalb `createCanvas(670, 561)`. Dies entspricht dem Seitenverhältnis von Abbildung 11.11. Außer dem Seitenverhältnis muss nun noch die Verkleinerung für jeden Rekursionsschritt ermittelt werden, sowie die x- und y-Koordinaten des linken oberen Eckpunktes des in den Bilderrahmen einzufügenden Fotos (Abb. 11.12 Zeile 20). Dies gelingt

am besten durch Ausprobieren. Weiterhin benötigen wir unsere im Kapitel 9.1 erworbenen Kenntnisse, wie man ein Foto in seinem Sketch aufruft und zur Anzeige bringt.

```
1 let foto;
2
3 function preload(){
4 foto = loadImage("barfrau.jpg")
5 }
6
7 function setup(){
8 createCanvas(670, 561);
9 }
10
11 function draw(){
12 background(200);
13 // x y b h
14 bild(0, 0, 670, 561);
15 }
16
17 function bild(x, y, b, h){
18 image(foto, x, y, b, h);
19 if (b > 10) // Verhinderung einer Endlosschleife
20 { bild(x+b/2.32, y+b/4.238, b/3.05, h/3.02);
21 }
22 }
23
```

Abbildung 11.12: Sketch für die Erstellung von Abbildung 11.11

### 11.3 Zusammenfassung

Fassen wir zusammen, was wir in Kapitel 11 gelernt haben, bzw. gelernt haben sollten.

Selbstähnlichkeit bei Vergrößerungen ist ein typisches Merkmal von Fraktalen. Die Programmierung von fraktalen Strukturen eröffnet neue Möglichkeiten zur künstlerischen Gestaltung. Die Programmierung solcher Strukturen gelingt besonders elegant mittels **rekursiver Programmierung**. In einem rekursiven Programm ruft eine Funktion sich selbst immer wieder auf. Damit hierbei keine Endlosschleife entsteht, die zum Programmabsturz führt, muss im Code unbedingt eine Abbruchbedingung eingefügt sein.

## 11.4 Übungen

- Aufgabe 01 Das folgende Bild soll mittels rekursiver Programmierung gezeichnet werden. Tipp: Die Diagonale in einem Quadrat ist die Quadratwurzel aus 2 multipliziert mit der Seitenlänge des Quadrates. Also:  $d = \sqrt{2} \cdot a$ . Die p5.js-Funktion für die Quadratwurzel lautet: `sqrt()`. Siehe Referenz.



Abbildung 11.13: Fraktales, rotierendes Quadrat

- Aufgabe 02 Fotografiere eine Puppe, ein Stofftier oder ... und erstelle mittels rekursiver Programmierung ein Bild ähnlich der folgenden Abbildung.

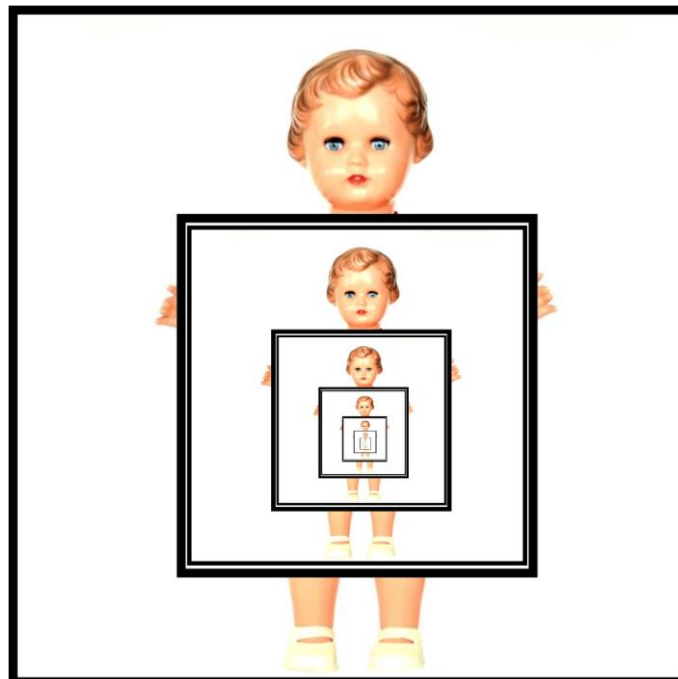


Abbildung 11.14: Bild in Bild

Aufgabe 03 Informiere dich im Internet, was man unter einem Sierpinski-Teppich versteht. Schreibe einen Sketch, mit dem man einen Sierpinski-Teppich aus Kreisen zeichnen kann (siehe Abbildung).

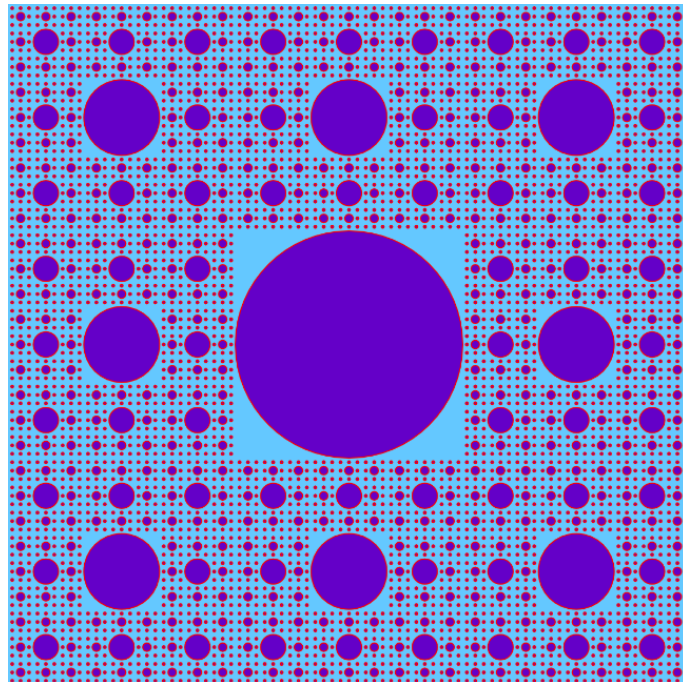


Abbildung 11.15: Sierpinski-Teppich aus Kreisen

Aufgabe 04 Fülle die Kreise in dem Sierpinski-Teppich von Aufgabe 3 mit einem Foto.

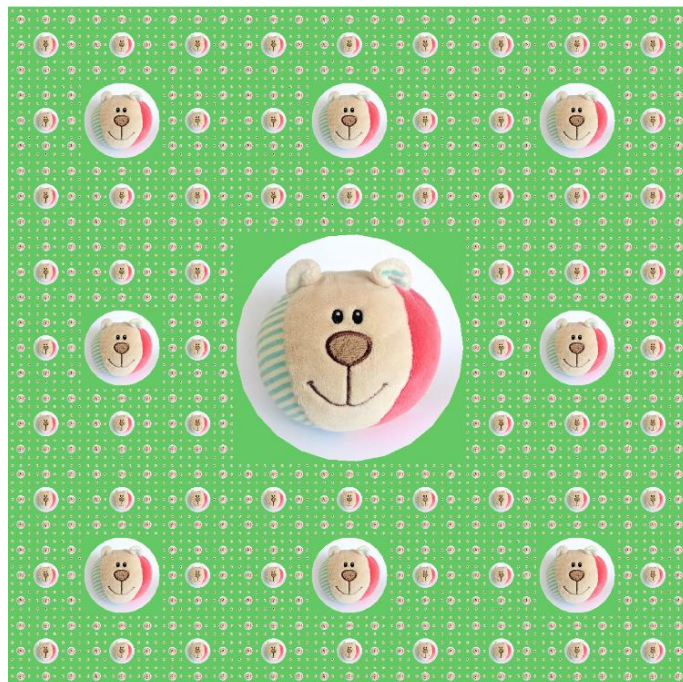


Abbildung 11.16: Sierpinski-Teppich mit Fotos

Aufgabe 05 Die folgende Abbildung zeigt einen Sierpinski-Teppich aus Quadraten. Schreibe einen entsprechenden Sketch.

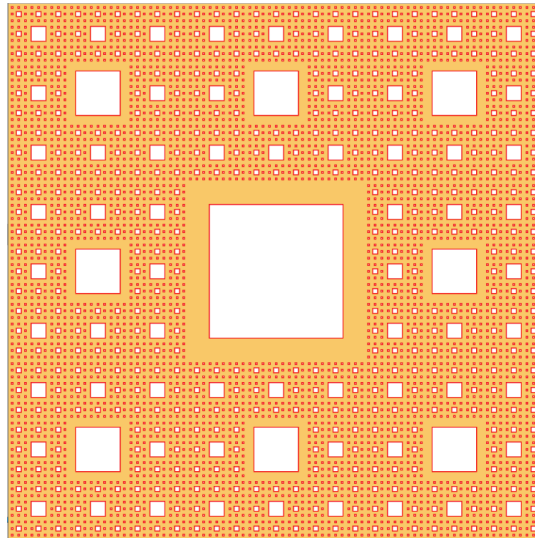


Abbildung 11.17: Sierpinski-Teppich aus Quadraten

Aufgabe 06 Wie die folgenden Abbildungen zeigen, kann auch ein Baum fraktale Eigenschaften besitzen. Schreibe einen Sketch, mit dem man einen Baum in den vier Jahreszeiten mittels rekursiver Programmierung zeichnen kann.

Da die Lösung dieser Aufgabe nicht gerade einfach ist, gibt es hier einige Tipps: Drehe bei jedem Rekursionsschritt das Koordinatensystem um einen bestimmten Winkel und verschiebe seinen Ursprung. Zeichne in diesem Koordinatensystem einen neuen verkleinerten Ast. Benutze hierzu die Funktionen *rotate()* und *translate()*. Damit die Rotationen sich nicht fortwährend addieren, verwende *push()* und *pop()*.

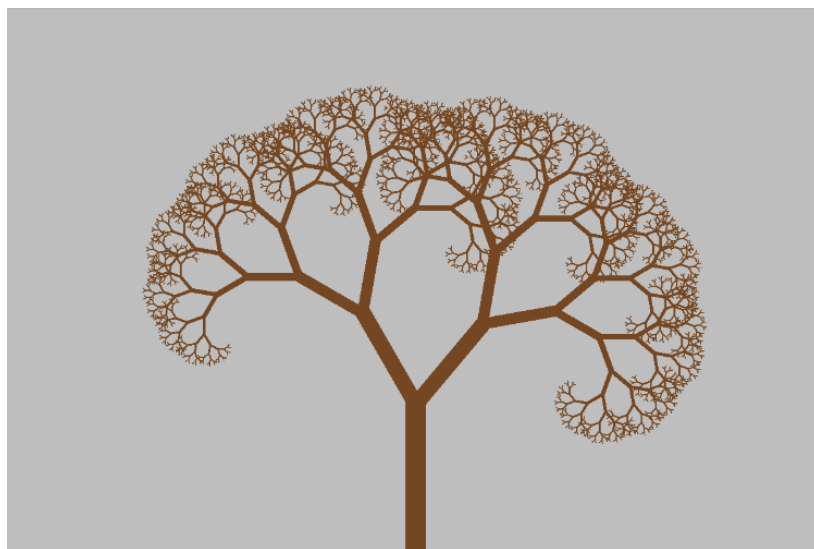


Abbildung 11.18: Baum im Winter

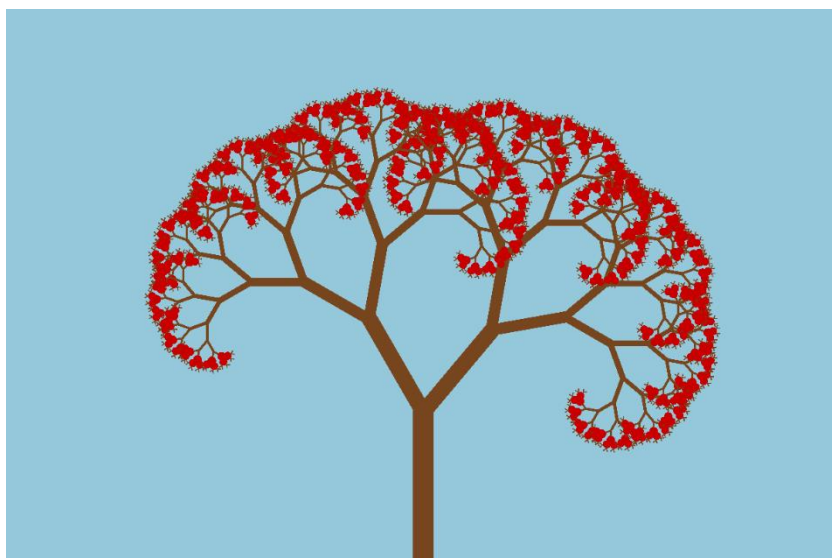
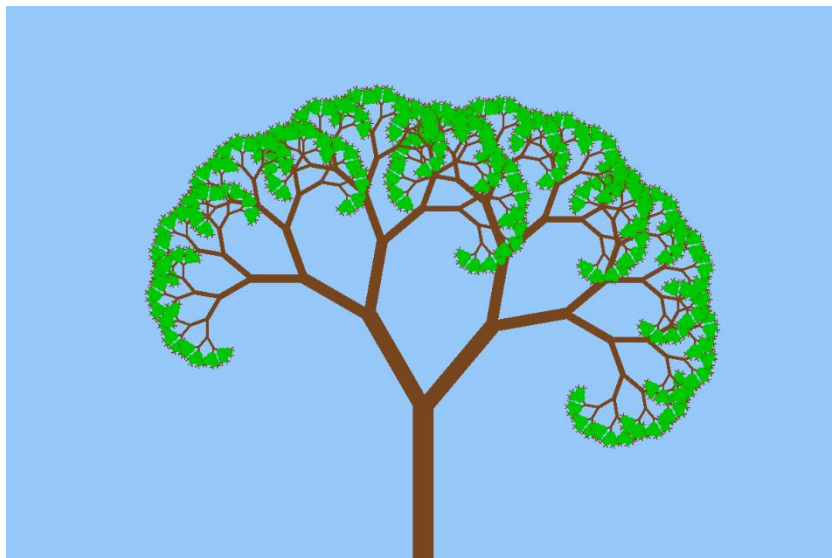
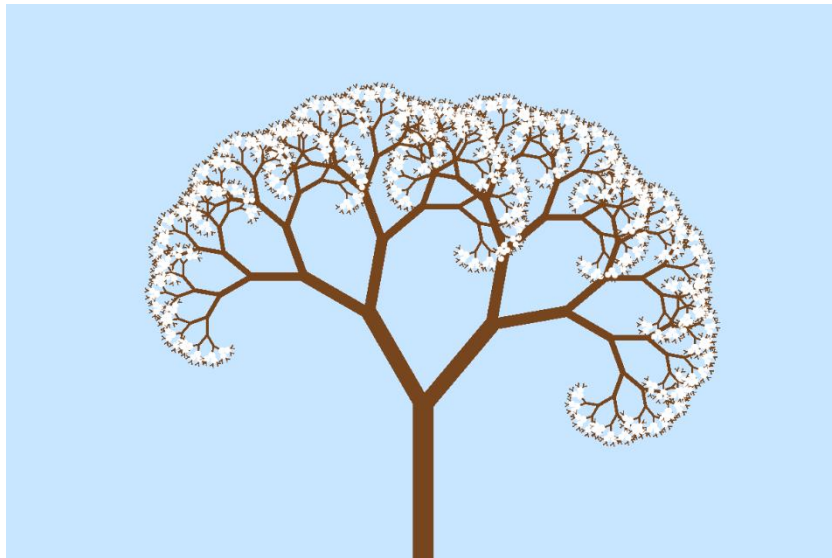


Abbildung 11.19: Fraktaler Baum im Frühling, Sommer und Herbst



## 12 Sounds und Sliders

### 12.1 Sounds

#### 12.1.1 Ballspiel mit Ton

Bisher haben wir uns hauptsächlich auf die Erstellung von Grafiken beschränkt, die man auf Papier ausdrucken kann. Wenn man dynamische Grafiken erzeugt, deren zeitliche Änderung sich auf einem Monitor beobachten lässt, dann wünscht man sich oft, dass man diese Änderung mit Musik oder Tönen untermalen könnte. Ein Beispiel hierfür ist unser Sketch *Ballspiel* aus Kapitel 5 (siehe Abb. 12.1). Schön wäre es, wenn wir beim Auftreffen des Balls auf den Schläger ein „Plopp!“ hören könnten.



Abbildung 12.1: Ballspiel aus Kapitel 5

Erfreulicherweise lässt sich der Funktionsumfang von p5.js durch Bibliotheken erweitern. Eine Bibliothek (Library) ist eine Sammlung von Code zu einem bestimmtem Thema. Wenn man das vollständige Softwarepaket von p5.js unter <https://p5js.org/download/> herunterlädt, so verfügt man sofort über die Bibliothek **p5.sound.js** (siehe Abb. 12.2). Weitere Bibliotheken findet man unter <https://p5js.org/libraries/>.

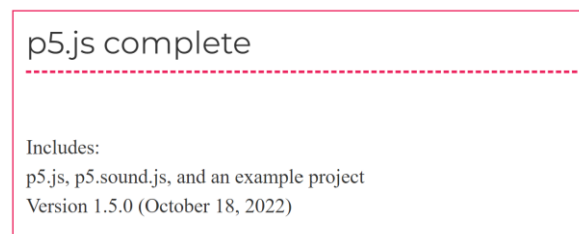


Abbildung 12.2: Im Softwarepaket p5.js complete enthaltene Bibliothek p5.sound.js

Nach dem Herunterladen und Entpacken findet man, wie schon im Kapitel 4 erwähnt, in seinem Dateordner die in Abbildung 12.3 dargestellten Ordner und Dateien.

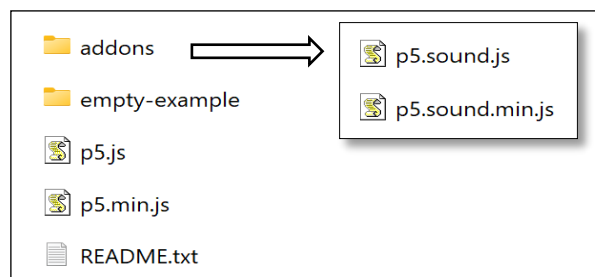


Abbildung 12.3: Softwarepaket p5.js complete

Um unser Ballspiel mit „Plopp!“ zu versehen, benötigen wir *p5.sound.js* und eine Sounddatei *plopp.mp3* oder *plopp.wav*. Eine solche mp3- oder wav-Datei kann man mit einem Audioeditor wie

zum Beispiel dem kostenlosen **Audacity** aufzeichnen (<https://www.audacityteam.org/>). Diese Audiodatei muss in den Ordner *empty-example* platziert werden. Natürlich können wir dem Ordner *empty-example* auch einen anderen Namen geben. Zum Beispiel *Ballspiel\_mit\_Ton*. Abbildung 12.4 zeigt den vollständigen Sketch.

Wie schon im Kapitel 9 erwähnt, kann man Sketche mit Bild- und Audio-Dateien seit 2021 nur dann offline öffnen, wenn man einen eigenen Webserver erstellt hat. Wenn man keinen eigenen Webserver erstellt hat, dann öffnet man den Sketch von Abbildung 12.4 sowie auch alle anderen Sketche von Kapitel 12 im Online-Editor von p5.js.

```

1 // Ballspiel mit Ton
2 let db = 30.0; // Balldurchmesser
3 let xb = 50.0; // x-Startposition des Balls
4 let yb = 50.0; // y-Startposition des Balls
5 let vbx = 8.0; // Startgeschwindigkeit des Balls in x-Richtung
6 let vby = 4.0; // Startgeschwindigkeit des Balls in y-Richtung
7 let s = 100.0; // Größe des Schlägers
8 let p = 0; // Zahl der Treffer (Punkte)
9 let treffer; // Variable zur Verwaltung der Sounddatei
10
11 // Die Sounddatei wird geladen und der Variablen treffer zugeordnet
12 function preload() {
13 treffer = loadSound("plopp.mp3");
14 }
15
16 function setup() {
17 createCanvas(600, 400);
18 }
19
20 function draw() {
21 background(230, 230, 210);
22
23 // Wände
24 stroke(0, 180, 0);
25 strokeWeight(15);
26 line(0, 0, 600, 0);
27 line(0, 400, 600, 400);
28 line(0, 0, 0, 400);
29
30 // Schläger
31 noStroke();
32 strokeWeight(2);
33 fill(255, 0, 0);
34 rect(width, mouseY-s/2, -10, s);
35
36 // Ball
37 xb = xb + vbx;
38 yb = yb + vby;
39 stroke(255, 0, 0);
40 fill(255, 255, 0);
41 ellipse(xb, yb, db, db);
42
43 // Bedingungen für die Reflexion an den waagerechten Wänden
44 if (yb <= db/2 || yb >= height-db/2) {
45 vby = -vby;
46 }
47
48 // Bedingungen für die Reflexion an der senkrechten Wand
49 if (xb < db/2) {
50 vbx = -vbx;
51 }
52 /* Bedingungen für den Punktgewinn beim Auftreffen auf den Schläger,
53 der Erhöhung der Ballgeschwindigkeit in x-Richtung und dem Abspielen
54 der Sounddatei. */
55 if ((abs(mouseY-yb) < 50) && xb >= width) {
56 p++;
57 vbx = -vbx*1.2;
58 treffer.play();
59 }
60
61 // Ein neuer Ball wird beim Verlassen des Spielfeldes erzeugt
62 if (xb > width+20) {
63 vbx = 8.0;
64 xb = random(50, 100);
65 yb = random(50, 350);
66 }
67
68 // Text
69 noStroke();
70 fill(0, 0, 255);
71 textSize(30);
72 text("Punkte " + p, 50, 100);
73 fill(0);
74 text("Zeit " + round((30-millis())/1000), 50, 50);
75
76 // Zeitliche Begrenzung für das Spiel
77 if (millis()/1000 >= 30) {
78 fill(255, 0, 0);
79 text("Spielende!", 50, 150);
80 noLoop();
81 }
82 }
83

```

Abbildung 12.4: Der vollständige Sketch *Ballspiel\_mit\_Ton*

Was haben wir nun im Sketch *Ballspiel* aus Kapitel 5 geändert, damit wir bei einem Treffer mit dem Schläger einen Ton hören? Nun, die Änderungen sind recht gering. Wir sehen sie in den Zeilen 9 bis 14 und in der Zeile 58.

Schauen wir uns die Zeilen 9 bis 14 näher an (Abb. 12.5). Hier begegnen wir wieder der Funktion *preload()*, die wir schon aus Kapitel 9 kennen. Sie wird benutzt, um große Dateien hochzuladen bevor der Sketch seine eigentliche Arbeit beginnt. Im Kapitel 9 waren es Bilddateien und in diesem Kapitel sind es Sounddateien. Weiterhin wird in Zeile 13 der Sounddatei eine Variable zugeordnet, damit man im späteren Programmverlauf darauf zurückgreifen kann. Dies geschieht in Zeile 58 (Abb. 12.6).

```
9 let treffer; // Variable zur Verwaltung der Sounddatei
10
11 // Die Sounddatei wird geladen und der Variablen treffer zugeordnet
12 function preload() {
13 treffer = loadSound("plopp.mp3");
14 }
```

Abbildung 12.5: Der Sketch greift auf die Sounddatei *plopp.mp3* zu

```
58 treffer.play();
```

Abbildung 12.6: Die Sounddatei *plopp.mp3* wird abgespielt

Wenn wir keinen lokalen Server eingerichtet haben, dann müssen wir, wie in Kapitel 9.1 beschrieben, unseren Sketch im Webeditor von p5.js schreiben und hier unsere Sounddatei hochladen.

### 12.1.2 Geister

Schauen wir uns nun ein weiteres Beispiel an. Der folgende Sketch *Geister* wurde mittels objektorientierter Programmierung erstellt. Wie man mittels objektorientierter Programmierung einen Sketch erstellt, dies haben wir in Kapitel 10 gelernt. Deshalb wollen wir uns in dem nun folgenden Text auf die Erklärung der Neuerungen beschränken.

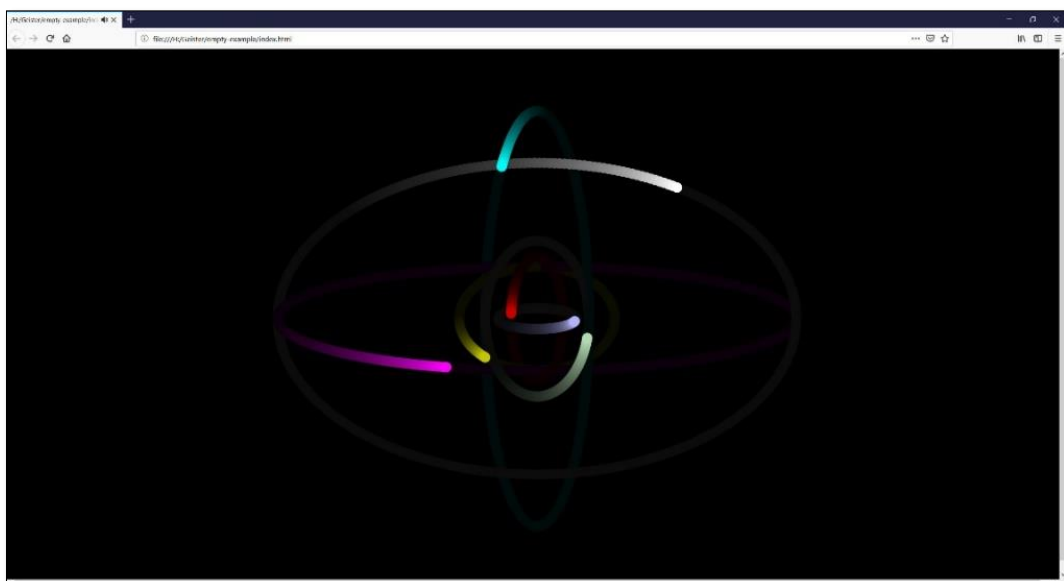


Abbildung 12.7: Die Abbildung der Geisterwelt füllt das ganze Browserfenster aus

Schauen wir uns zuerst die Zeilen 19 und 20 im Konstruktor an (Abb. 12.8). Ein Geist ist hier ein farbiger Kreis, dessen Mittelpunkt sich durch die Kombination der Funktionen *sin()* und *cos()* auf einer Ellipsenbahn bewegt. Mit *rx* und *ry* bestimmt man die Haupt- und Nebenachse der Ellipse. Mit *x* und *y* und dem Divisor *d* bestimmen wir die Winkel in den Argumenten von Sinus und Cosinus. *x* und *y* vergrößern sich bei jedem Durchlauf von *function draw()* um *v* (Zeile 22 und 23 in Abbildung 12.8)

```

1 // Das OBJEKT mit dem Namen "Geister" wird erschaffen.
2
3 // KONSTRUKTOR mit Instanzvariablen
4 function Geister(tempX, tempY, tempRX, tempRY, tempD, tempR, tempG, tempB, tempV) {
5 this.x = tempX;
6 this.y = tempY;
7 this.rx = tempRX;
8 this.ry = tempRY;
9 this.d = tempD;
10 this.r = tempR;
11 this.g = tempG;
12 this.b = tempB;
13 this.v = tempV;
14
15 // METHODEN
16 this.display= function() {
17 fill(this.r, this.g, this.b);
18 ellipse(this.rx*sin(this.x*PI/this.d), this.ry*cos(this.y*PI/this.d), 20, 20);
19 }
20
21 this.move = function() {
22 this.x = this.x + this.v;
23 this.y = this.y + this.v;
24 }
25 }
26

```

Abbildung 12.8: Der Konstruktor dient als Bauplan zum Zeichnen der Geister

Anhand des Hauptsketches (Abb. 12.9) sollen nun die wesentlichen Neuerungen und die Verwendung der Bibliothek *p5.sound.js* erläutert werden. In Zeile 1 erzeugen wir ein leeres Array mit dem Namen *geist*. In Zeile 2 deklarieren wir die Variable *klang*. In den Zeilen 4 bis 6 laden wir den Sound *geister.mp3* in den Sketch und ordnen ihn der Variablen *klang* zu. In Zeile 12 sorgen wir mit *klang.loop()* dafür, dass dieser Sound fortlaufend abgespielt wird, solange die Animation läuft. Man fragt sich natürlich sofort: „Warum steht *klang.loop()* bei *function setup()* und nicht wie bei unserem Sketch *Ballspiel\_mit\_Ton* bei *function draw()*?“ Bei *function draw()* wird der Sound sechzigmal pro Sekunde gestartet. Bei einem kurzen „Plopp!“ ist dies kein Problem. Bei einem längeren Sound hört es sich nicht gut an, wenn der Sound jedes Mal neu gestartet wird, während der vorher gestartete noch läuft. Aus diesem Grund starten wir den Sound bei *function setup()*. Schreiben wir hier *klang.play()* so wird der Sound nur einmal abgespielt. Bei unserem Sketch *Geister* möchten wir ihn aber fortlaufend abspielen, solange die Animation läuft. Deshalb schreiben wir *klang.loop()*.

Mit *createCanvas(windowWidth, windowHeight)* in Zeile 9 bewirken wir, dass die Leinwand das ganze Browserfenster ausfüllt.

Bleibt noch zu klären, wie wir die Kreise (Geister) bei ihrer Bewegung mit einem **Schweif** versehen. Dies gelingt mit dem folgenden Trick (Abb. 12.9, Zeile 26 u. 27). Wir zeichnen bei jedem Durchlauf von *function draw()* ein Rechteck, welches die ganze Leinwand bedeckt und füllen es mit der Farbe des Hintergrundes (hier schwarz). So werden die gezeichneten Kreise fortlaufend übermalt. Da die Füllfarbe aber nur eine Deckkraft von 10 besitzt, wird die Farbe der einzelnen Kreise bei jedem

Übermalen nur abgeschwächt. Dadurch entsteht der Eindruck, dass die Kreise (Geister) einen Schweif hinter sich herziehen.

```
1 let geist = [];
2 let klang;
3
4 function preload() {
5 klang = loadSound("geister.mp3");
6 }
7
8 function setup(){
9 createCanvas(windowWidth, windowHeight);
10 background(0);
11 noStroke();
12 klang.loop();
13
14 // Die einzelnen Objekte (Geister) werden erschaffen.
15 // x, y, rx, ry, d, r, g, b, v)
16 geist[0] = new Geister(0, 0, 50, 120, 60, 200, 0, 0, 0.2);
17 geist[1] = new Geister(0, 0, 150, 100, 100, 200, 200, 0, 0.4);
18 geist[2] = new Geister(0, 0, 100, 150, 60, 180, 220, 180, 0.6);
19 geist[3] = new Geister(0, 0, 75, 20, 30, 180, 180, 255, 0.3);
20 geist[4] = new Geister(0, 0, 500, 300, 130, 255, 255, 255, -0.8);
21 geist[5] = new Geister(0, 0, 500, 100, 130, 255, 0, 255, 0.6);
22 geist[6] = new Geister(0, 0, 100, 400, 130, 0, 255, 255, 0.5);
23 }
24
25 function draw() {
26 fill(0, 10); // Farbe und Deckkraft
27 rect(0, 0, width, height);
28 translate(width/2, height/2);
29
30 // Die Methoden für die einzelnen Objekte werden aufgerufen.
31 for(var i = 0; i <= 6; i++) {
32 geist[i].move();
33 geist[i].display();
34 }
35 }
36
```

Abbildung 12.9: Im Hauptteil des Sketches werden die Eigenschaften der einzelnen Geister entsprechend dem im Konstruktor vorgegebenen Bauplan festgelegt

### 12.1.3 Lissajous Figuren

Nun wird es etwas komplizierter. Wir wollen einen Sketch schreiben, der eine tanzende Lissajous-Figur erzeugt (Abb. 12.10). Im Internet findet man viele Anleitungen, die einem die notwendigen physikalischen Kenntnisse hierzu vermitteln. Zum Beispiel bei <https://de.wikipedia.org/wiki/Lissajous-Figur>

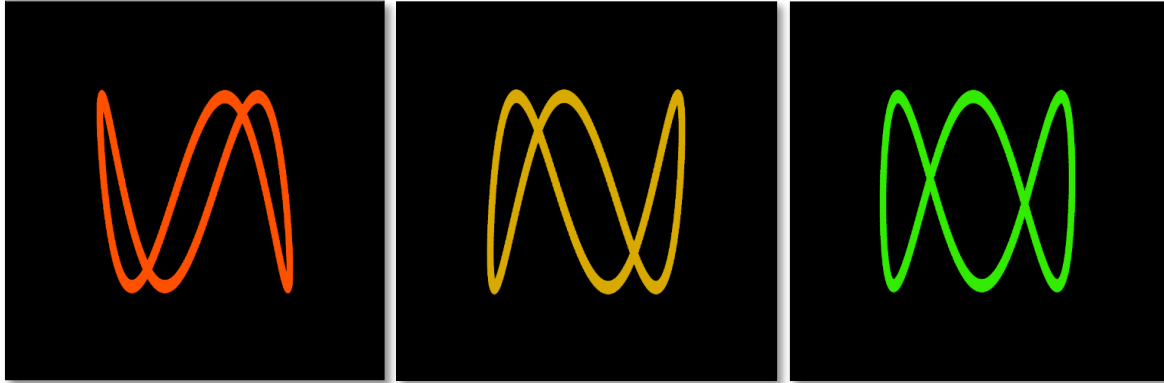


Abbildung 12.10: Tanzende Lissajous-Figuren

```
1 // Lissajous-Figuren
2 let musik; // Variable für den Sound
3 let amp; // Variable für die Lautstärke des Sounds
4
5 let A = 100; // Amplitude der Schwingungen der Lissajous-Figur
6 let w = 0; // Winkel in Grad
7 let z = 0; // Kleinwinkeländerung in Grad
8 let v = 0; // Änderung der Drehgeschwindigkeit
9
10 function preload(){
11 musik = loadSound("walzer.mp3");
12 }
13
14 function setup(){
15 // Die Leinwand wird im Browserfenster verschoben
16 let ort = createCanvas(400, 400);
17 ort.position(300, 200);
18
19 musik.loop(); // Der Sound wird fortlaufend wiederholt
20 amp = new p5.Amplitude(); // Die aktuelle Lautstärke des Sounds wird der Variablen amp zugeordnet
21 }
22
23 function draw(){
24 background(0);
25
26 // Die Lissajous-Figur wird gezeichnet
27 for (let i = 0; i < 1000; i++){ // 1000 Ellipsen werden pro Durchlauf gezeichnet
28 let a = amp.getLevel(); // Über die aktuelle Lautstärke wird die Farbe der Lissajous-Figur bestimmt
29 fill(a*2550, 255-a*1000, 0);
30 noStroke();
31 ellipse((width/2 + A*sin(radians(w+z))), (height/2 + A*sin(radians(3*w+90))), 7, 14);
32 w++; /* Hier wird festgelegt, wie schnell sich die einzelnen Ellipsen
33 zur Erzeugung der Lissajous-Figur in x- und y-Richtung bewegen */
34 }
35 // Nun wird die Drehgeschwindigkeit der gesamten Figur festgelegt
36 z = z + sin(v);
37 v = v + 0.01;
38 }
39
```

Abbildung 12.11: Sketch zur Erzeugung einer tanzenden Lissajous-Figur

Lissajous-Figuren ergeben sich, wenn man zwei zueinander senkrechte Schwingungen überlagert. Die Lissajous-Figur von Abbildung 12.10 wurde mit zwei Schwingungen gleicher Amplitude  $A$  gezeichnet, die ein Frequenzverhältnis von 1:3 und eine Phasenverschiebung von  $90^\circ$  besitzen. Dafür sorgen in der Zeile 31 unseres Sketches (Abb. 12.11) die hier rot hervorgehobenen Elemente.

```
ellipse((width/2 + A*sin(radians(w+z))), (height/2 + A*sin(radians(3*w+90))), 7, 14);
```

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. Dafür sorgt der Wert von  $z$ . Damit sich die Figur nicht nur in eine Richtung dreht, erhöhen und verringern wir den Wert von  $z$  mittels einer Sinusfunktion, da diese bekanntlich Werte zwischen  $+1$  und  $-1$  annimmt. (Abb. 12.11, Zeile 36 und 37). Da bei unserer Animation für jede Stellung der Figur sehr viele kleine Ellipsen gezeichnet werden müssen, muss dies zwingend mit einer *for-Schleife* erfolgen (Zeile 27 bis 34 in Abb. 12.11).

Kommen wir nun zum Thema Sound. Die Bewegung der Lissajous-Figur wollen wir nicht nur mit Musik untermalen, sondern über die aktuelle Lautstärke der Musik soll die Farbe der Figur festgelegt werden. Siehe dazu die Erläuterungen in den Zeilen 3, 20, 28 und 29 von Abbildung 12.11.

Erklärt werden müssen nun noch die Zeilen 16 und 17 von Abbildung 12.11. Sie haben nichts mit dem Thema Sound zu tun. Mit diesen beiden Zeilen können wir die Leinwand an eine beliebige Stelle im Browserfenster platzieren. Dazu deklarieren wir in Zeile 16 eine Variable mit dem Namen *ort* und ordnen ihr die Funktion *createCanvas(400, 400)* zu. In Zeile 17 legen wir dann mit der Funktion *position()* den Ort für den linken oberen Eckpunkt der Leinwand im Browserfenster fest.

Die Bibliothek *p5.sound.js* bietet sehr viele Möglichkeiten, seine Kunstwerke mit Musik und Tönen zu ergänzen. Wer sich hierfür besonders interessiert, der schaue sich den folgenden Link an:

<https://p5js.org/reference/#/libraries/p5.sound>

Außer der Bibliothek *p5.sound.js* findet man auf der Homepage von *p5.js* noch viele weitere Bibliotheken unter dem Link:

<https://p5js.org/libraries/>

## 12.2 Sliders

Als ich im Jahre 2018 die Erstausgabe dieses Buches schrieb, war neben der Bibliothek *p5.sound.js* auch noch die Bibliothek *p5.dom.js*, im vollständige Softwarepaket von *p5.js* enthalten. Mittels der Bibliothek *p5.dom.js* konnten zahlreiche HTML-Elemente außerhalb der Leinwand erzeugt werden, die mit den Elementen auf der Leinwand wechselwirken konnten. Sketche mit solchen Elementen wie zum Beispiel Schieberegler (**Sliders**) und Knöpfe (**Buttons**) funktionierten auch offline. Da heutzutage die Bibliothek *p5.dom.js* im Softwarepaket von *p5.js* fehlt, müssen die Sketche, die solche Elemente enthalten, im Online-Editor von *p5.js* geöffnet werden.

Wir wollen uns in diesem Buch auf ein Beispiel beschränken. Wir werden einen Sketch schreiben, mit dem man einen Farbmischer für RGB-Farben erstellen kann. Die Mischfarbe und die einzelnen RGB-Werte sollen auf der Leinwand angezeigt werden. Die RGB-Werte geben wir aber nicht über die Tastatur ein, sondern mithilfe von Schiebereglern, den sogenannten **Slidern**. Diese Sliders positionieren wir im Browserfenster unterhalb der Leinwand (Abb. 12.12). Den zugehörigen Sketch finden wir in Abbildung 12.13. Er ist leicht zu verstehen. Die einzigen Funktionen, die erläutert werden müssen, sind ***createSlider()*** und ***position()***. Mit ***createSlider(0, 255, 0)*** als Beispiel setzt man den Minimalwert auf 0, den Maximalwert auf 255 und den Startwert auf 0). Mit ***position(50, 600)*** als Beispiel setzt man den linken Eckpunkt des Sliders im Browserfenster auf die x-Koordinate 50 und auf die y-Koordinate 600.



Abbildung 12.12: RGB-Farbmischer

```

1 let rot;
2 let gruen;
3 let blau;
4
5 function setup(){
6 createCanvas(700, 580);
7 background(0);
8
9 // Die Schieberegler (Slider) werden erzeugt und positioniert
10 rot = createSlider(0, 255, 0);
11 rot.position(50, 600);
12 gruen = createSlider(0, 255, 0);
13 gruen.position(250, 600);
14 blau = createSlider(0, 255, 0);
15 blau.position(450, 600);
16 }
17
18 function draw(){
19 let r = rot.value();
20 let g = gruen.value();
21 let b = blau.value();
22
23 stroke(255);
24 fill(r, g, b);
25 rect(20, 20, 660, 400);
26
27 fill(r, 0, 0);
28 rect(50, 460, 140, 70);
29 fill(0, g, 0);
30 rect(250, 460, 140, 70);
31 fill(0, 0, b);
32 rect(450, 460, 140, 70);
33
34 fill(255);
35 textSize(40);
36 textAlign(CENTER)
37 text(r, 120, 510);
38 text(g, 320, 510);
39 text(b, 520, 510);
40
41 textSize(20);
42 textAlign(CENTER)
43 text("rot", 120, 560);
44 text("grün", 320, 560);
45 text("blau", 520, 560);
46 }
47

```

Abbildung 12.13: Sketch zum RGB-Farbmischer

## 12.3 Zusammenfassung

Fassen wir zusammen, was wir in Kapitel 12 gelernt haben, bzw. gelernt haben sollten.

|                     |                                                                                                                                                                                                                                                                                                                                         |
|---------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Bibliotheken</b> | Der Funktionsumfang von p5.js lässt sich durch Bibliotheken erweitern. Siehe: <a href="https://p5js.org/libraries/">https://p5js.org/libraries/</a> .                                                                                                                                                                                   |
| <b>p5.sound.js</b>  | Mittels der Bibliothek <i>p5.sound.js</i> werden unsere Kunstwerke mit Musik oder Tönen untermalt.                                                                                                                                                                                                                                      |
| <b>Slider</b>       | Ein Slider ist ein Schieberegler. Er kann mit der Funktion <i>createSlider()</i> erzeugt und mit der Funktion <i>position()</i> im Browserfenster platziert werden. Zwischen die runden Klammern der Funktion schreibt man die folgenden Werte: <i>createSlider(min, max, Startwert)</i> . Beispiel: <i>createSlider(0, 255, 100)</i> . |
| <b>Schweif</b>      | Ein Schweif hinter einem sich bewegenden Objekt lässt sich erzeugen, wenn man bei <i>function draw()</i> ein, die ganze Leinwand bedeckendes, Rechteck erzeugt, dessen Füllfarbe transparent ist.                                                                                                                                       |

## 12.4 Übungen

**Aufgabe 01** In Kapitel 7.5 (Abb. 7.18) haben wir ein Mausmalprogramm für kleine Kinder programmiert. Schön wäre es, wenn die Kinder beim Malen und beim Betrachten ihres Werkes von Musik begleitet würden und die Musik dann deutlich lauter wird, wenn sie aktiv malen. Schreibe einen entsprechenden Sketch.



Abbildung 12.14: Mausmalprogramm mit Musik

**Aufgabe 02** Schreibe einen Sketch, mit dem man mittels der Maus „Alle meine Entchen“ spielen kann. Die folgende Abbildung dient als Anregung.

# Alle meine Entchen

C4 D4 E4 F4 G4 G4 A4 A4 A4 A4 G4 A4 A4 A4 A4 G4  
 Al-Le mei-ne Ent-chen schwin-men auf dem See, schwin-men auf dem See,  
 F4 F4 F4 F4 E4 E4 G4 G4 G4 C4  
 Köpf-chen in das Was-ser Schwänz-chen in die Höh'.

A4
C4
D4
E4
F4
G4

Abbildung 12.15: Mausklavier

Aufgabe 03 Erstelle einen HSB-Farbmischer (Abbildung 12.16), ähnlich dem RGB-Farbmischer von Abbildung 12.12 erstellt werde.

H = 205
H = 205  
S = 31
H = 205  
S = 31  
B = 60

H
H+S
H+S+B

Abbildung 12.16: HSB-Farbmischer

Aufgabe 04 Wie wäre es mit einem schicken Taschenrechner, so wie der von Abbildung 12.17? Schreibe den hierzu passenden Sketch. Mittels eines Sliders außerhalb der Leinwand sollen die x-Werte von 0 bis 100 eingestellt werden können. Auf der Leinwand erscheinen dann die mathematischen Funktionen und die zugehörigen Funktionswerte (siehe Abbildung). Diese Aufgabe ist recht einfach zu lösen, wenn man sich in der Referenz über die jeweiligen mathematischen Funktionen informiert.

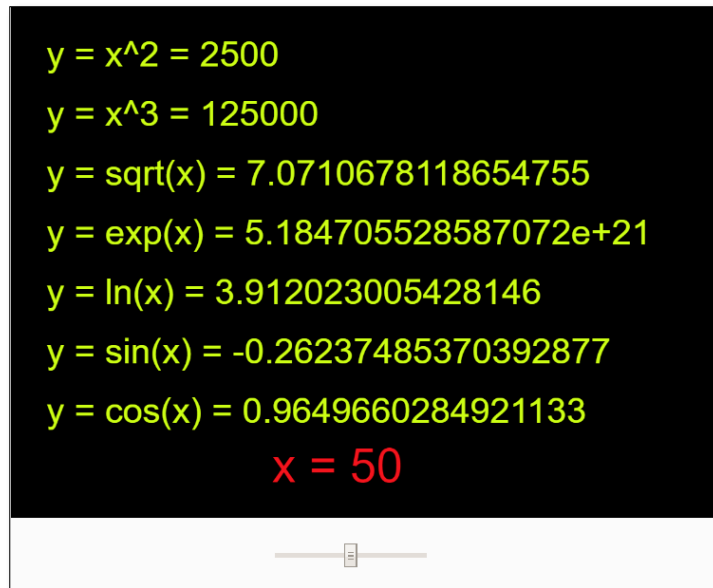


Abbildung 12.17: Bei diesem "Taschenrechner" kann  $x$  mittels eines Sliders eingestellt werden

Aufgabe 05 Wenn man sich in der Referenz von p5.js umschaue, dann sieht man, dass man nicht nur Sliders, sondern mit der Funktion **createButton()** auch Taster außerhalb der Leinwand erzeugen kann. Diese Funktion wurde dazu genutzt, um das folgende Bild zu erschaffen. Durch mehrfachen Druck (Mausklicks) auf die Taste „Drück mich!“, wurden auf der Leinwand unterschiedliche Uhren und Schriftzüge abgebildet. Erstelle ein ähnliches Bild mit deinen eigenen Fotos.



Abbildung 12.18: Uhren und Schriftzüge wurden mittels Mausclick auf die Taste ins Bild eingefügt

## 13 Sketche auf dem Tablet und dem Smartphone programmieren

### 13.1 Sketche online schreiben

Während der Corona-Pandemie gab es in Deutschland an den meisten Schulen einen Digitalisierungsschub. Verstärkt wurden Tablets angeschafft, da diese mobilen Geräte im Gegensatz zu den PCs im Informatikraum, in allen Fächern und auch außerhalb des Schulgebäudes eingesetzt werden können. In der Regel sind Tablets auch preiswerter als Laptops. Ein weiterer Vorteil ist, dass sie neben Kamera und Mikrophon mit zahlreichen Sensoren ausgestattet sind. So können sie zum Beispiel mittels der kostenlosen App *phyphox* der RWTH Aachen für Experimente im Physikunterricht eingesetzt werden.

Doch eignen sich Tablets auch für das Programmieren eines p5.js-Sketches? Sie besitzen schließlich keine Maus und nur eine virtuelle Tastatur. Machen wir zuerst den Maus-Test mit einem iPad und dem Webeditor von p5.js. In dem Sketch von Abbildung 13.1 soll sich die Farbe des Kreises von grün nach rot ändern, wenn man mit dem Finger den Kreis berührt. In Zeile 7 des steht deshalb die if-Bedingung `mouselsPressed == true`. Obwohl wir den Mauszeiger durch unseren Finger ersetzen, so sehen wir, dass dieser Sketch funktioniert, da die Farbe des Kreises sich von grün nach rot ändert. Auch mit einem Tablet mit dem Betriebssystem Windows funktioniert dieser Sketch. Die Display-Koordinaten des Fingers können also die Display-Koordinaten des Mauszeigers ersetzen.

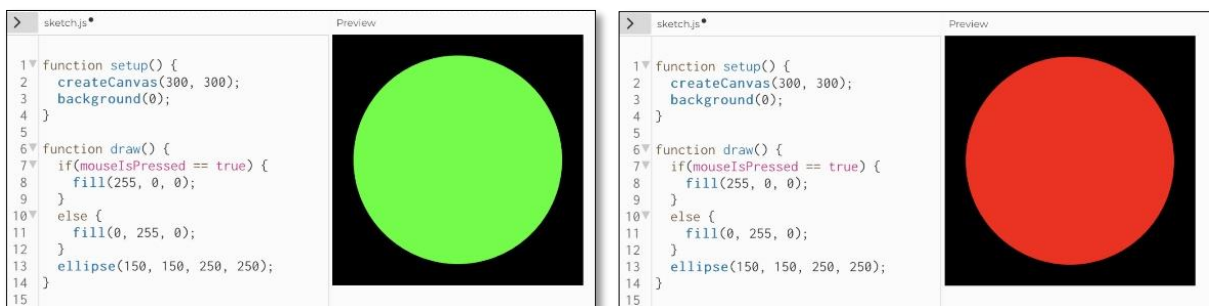


Abbildung 13. 1: `mouselsPressed` funktioniert auch durch Berührung des Displays mit dem Finger

Schreiben wir nun in Zeile 7 von Abbildung 13.1 `keysPressed == true` anstelle von `mouselsPressed == true` und probieren, ob der Sketch nun auch auf dem iPad oder Windows-Tablet funktioniert. Dazu rufen wir auf jedem dieser Tablets die virtuelle Tastatur auf. Egal, auf welche Taste wir auch drücken, der Sketch funktioniert nicht. Doch alle Sketche in diesem hier vorliegenden Buch, in denen keine Tastaturbefehle vorkommen, lassen sich auf Tablets programmieren.

Auf dem relativ großen Bildschirm eines Tablets kann man recht komfortabel programmieren. Auf dem kleinen Bildschirm eines Smartphones ist es etwas mühseliger (Abb. 13.3). Hilfreich ist es, wenn man eine kleine Schriftgröße wählt. Dazu muss man im Webeditor von p5.js rechts oben auf das Zahnrad klicken. Dann öffnet sich das folgende Fenster (Abb. 13.2). Hier wählt man eine entsprechend kleine Schriftgröße und bei Bedarf auch einen dunklen Hintergrund (Abb. 13.3 rechts). Anschließend vergrößert man den Bildschirmausschnitt (zwei Finger spreizen) und schreibt seinen Code. Den im Webeditor von p5.js erstellten Sketch kann man, wenn man einen Account eingerichtet hat, bei p5.js abspeichern.

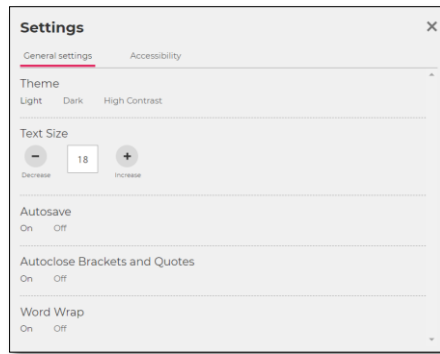


Abbildung 13. 2: Im Webeditor von p5.js kann man nicht nur die Schriftgröße einstellen

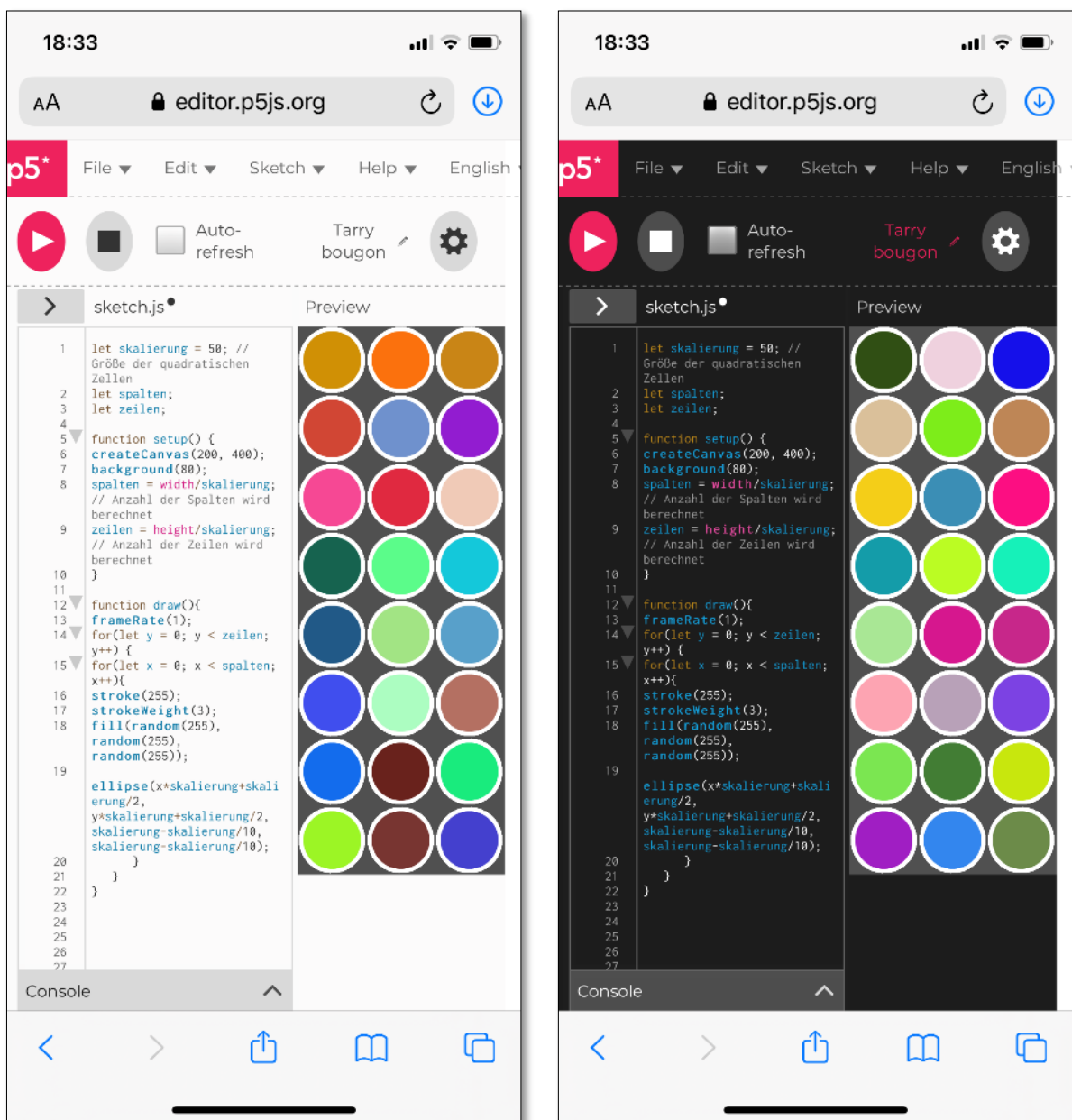


Abbildung 13. 3: Ausschnitt aus einem Sketch auf dem Smartphone in heller und dunkler Einstellung

## 13.2 Sketche offline schreiben

Etwas mehr Platz auf dem Display eines Smartphones hat man, wenn man seine Sketche nicht online im Webeditor schreibt, sondern offline in einem Textprogramm. Auf dem iPhone kann man zum Beispiel seinen Code in der App *Notizen* schreiben und anschließend seinen Code mit copy and paste in den Webeditor von p5.js einfügt. Bevor man mit dem Schreiben in der App *Notizen* beginnt, sollte man jedoch die Auto-Korrektur und Auto-Großschreibung abschalten. Dies gelingt wie folgt: Allgemein → Tastaturen → Auto-Großschreibung und Auto-Korrektur ausschalten.

Ähnliches gilt für Smartphones mit dem Betriebssystem Android. Ein möglicher Weg zum Ausschalten der Autokorrektur ist: Einstellungen → Sprache & Eingabe → Tastatur & Eingabemethoden → Die gewünschte Tastatur auswählen und die Autokorrektur und die Automatische Groß-/Kleinschreibung ausschalten. Je nach Gerät, Tastatur und Android-Version kann der Weg zum Ausschalten der Auto-korrektur von dem hier vorgeschlagenen Weg jedoch etwas abweichen. Wenn man seinen Code geschrieben hat, dann fügt man ihn anschließend mit copy and paste in den Webeditor von p5.js ein.

Wer ein iPhone oder iPad besitzt, kann seine Sketche nicht nur offline schreiben, sondern sich seine Programmiererergebnisse auch offline anschauen. Hierzu muss man sich aus dem App Store die sehr empfehlenswerte App *Processing&p5.js iCompiler* von Frederik Riedel (Frogg GmbH) herunterladen. Mittels dieser App kann man Sketche für Processing (siehe *Walter Stein, Programmieren lernen für den Physikunterricht mit Processing*) und für p5.js schreiben und abspielen. Die Abbildung 13.4 links und mittig zeigt ein Beispiel für einen p5.js-Sketch, der mit der App *Processing&p5.js iCompiler* auf einem iPhone erstellt wurde.

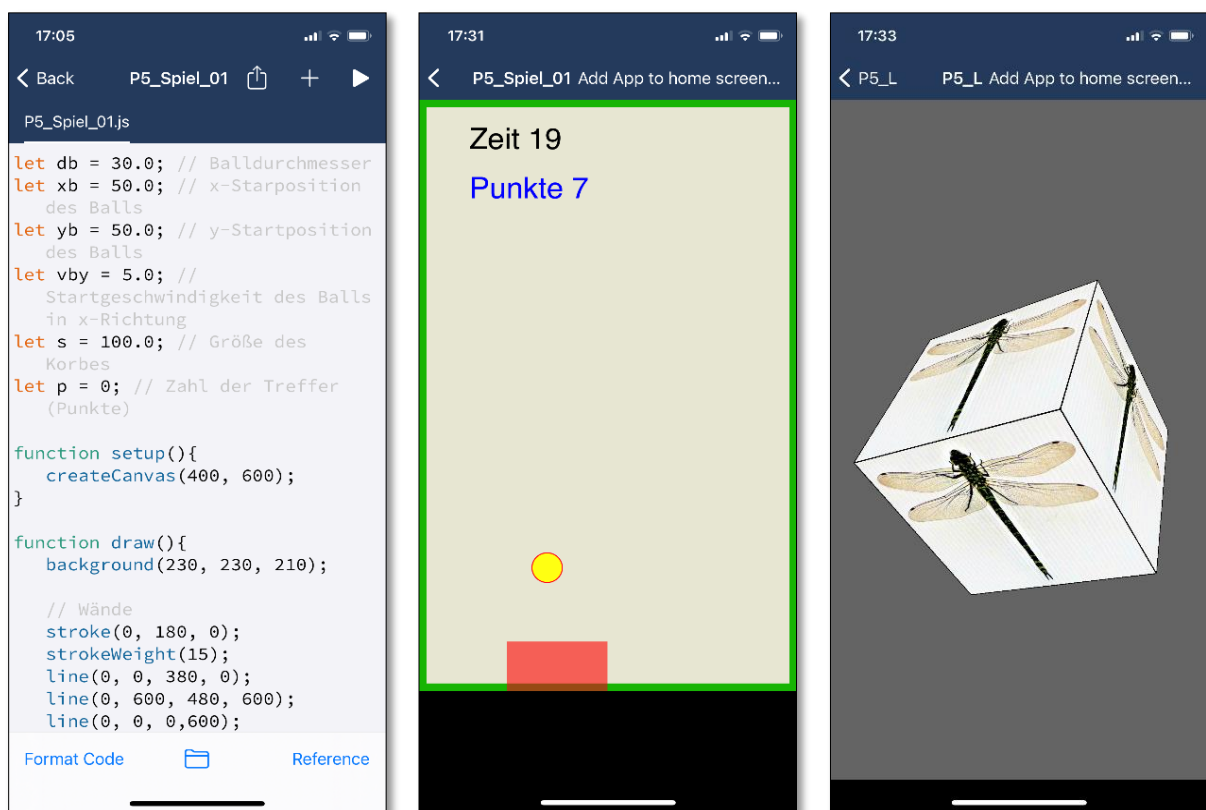


Abbildung 13. 4: Ausschnitt aus einem Sketch, der mit der App *Processing&p5.js iCompiler* offline erstellt und offline geöffnet wurde (links und mittig)

Erfreulicherweise kann man mit dieser App auch Fotos in seine Sketche einbinden (siehe Abb. 13.4 rechts). Wie dies gelingt, zeigt die zugehörige Abbildung 13.5. Bevor oder nachdem man seinen Code geschrieben hat, tippt man unten im Display auf das Ordnersymbol. In dem sich nun öffnenden Fenster tippt man auf *Import* und importiert ein Foto, zum Beispiel aus der App *Dateien*. In unserem Beispiel ist es das Bild einer Libelle. Anschließend startet man den Sketch und sieht, hier in unserem Beispiel, das Bild einer Libelle auf den Seitenflächen eines rotierenden Würfels (Abb. 13.4, rechts).

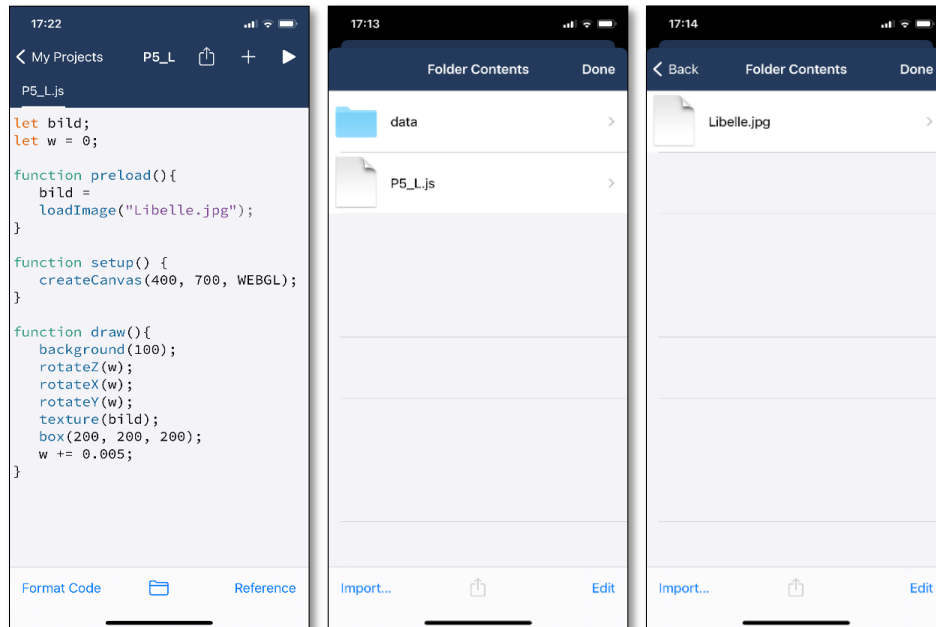


Abbildung 13. 5: Einbindung von Fotos in einen Sketch mittels der App Processing&p5.js iCompiler

Für Android-Smartphones oder Android-Tablets findet man Apps für p5.js im Play Store von Google.

### 13.3 Fingerübungen

Die folgenden Ratschläge gelten für das Betriebssystem iOS 14 von Apple. Für das Betriebssystem Android gelten annähernd die gleichen Regeln.

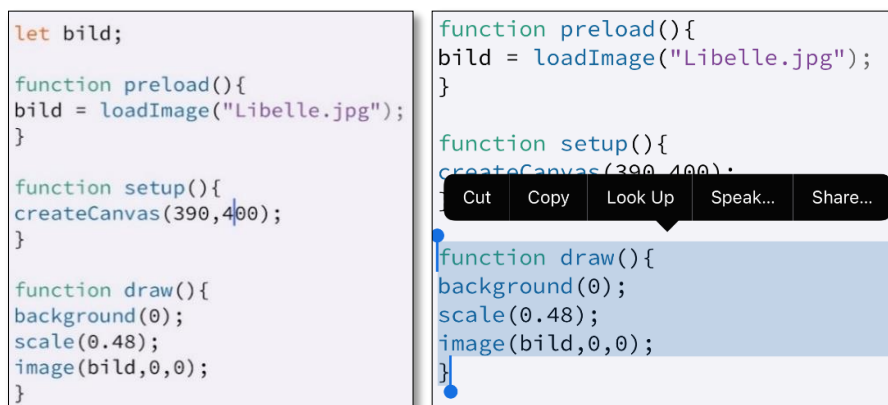


Abbildung 13. 6: Den Cursor auf dem Smartphone zielgenau platzieren(links) und Codeausschnitte kopieren oder ersetzen (rechts)

Wenn man in seinem Code auf dem Smartphone etwas ändern will, dann ist es nicht einfach, mit seinen dicken Fingerkuppen den Cursor genau an der gewünschten Stelle zu platzieren. Einfacher geht es, wenn man seine Fingerkuppe etwas länger auf dem Display verweilen lässt. Nun erscheint über der Fingerkuppe ein langer blauer Strich, den man nun bequem genau an die gewünschte Stelle schieben kann, um hier den Code zu ändern (Abb. 13.6, links).

Möchte man Teile des Codes kopieren oder ausschneiden, dann tippt man zweimal auf den Anfang oder das Ende des Codeabschnittes und zieht dann einen der blauen Marker über den zu kopierenden Teil. Dieser erscheint jetzt blau unterlegt und kann nun kopiert oder ausgeschnitten werden (Abb. 13.6, rechts).

Möchte man den gesamten Code kopieren oder ausschneiden, dann tippt man auf eine beliebige Stelle des Codes. Ein blauer Cursorstrich erscheint. Wenn man nun nochmal auf den blauen Cursorstrich tippt, dann erscheint *Select All*. Tippt man auf *Select All*, dann wird der gesamte Code blau unterlegt und kann nun in die Zwischenablage kopiert oder entfernt werden (Abb. 13.7).



Abbildung 13. 7: Den gesamten Code kopieren oder ausschneiden

Code oder Codeschnipsel, die sich in der Zwischenablage befinden, kann man mit *Paste* an den gewünschten Stellen im Code wieder einfügen (Abb. 13.8). Aber dies ist wohl allen Lesern bekannt;)

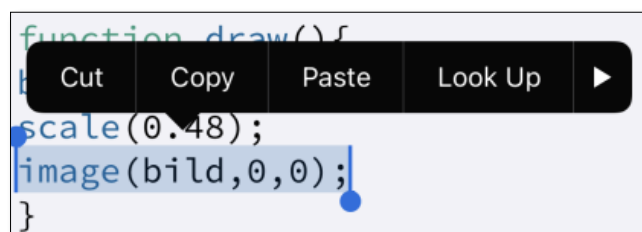


Abbildung 13. 8: Inhalt der Zwischenablage einfügen



## 14 Programmieren mit der KI ChatGPT

### 14.1 Erster Versuch

ChatGPT von der Firma OpenAI ist eine künstliche Intelligenz, die seit Ende November 2022 der Öffentlichkeit zur Verfügung steht und in allen Medien große Aufmerksamkeit findet. Mit ChatGPT kann man nicht nur in beeindruckender Weise kommunizieren. ChatGPT kann auch Fragen zu komplizierten Sachverhalten beantworten, kann Artikel für Journalisten und Referate für Schüler und Studenten schreiben, ... . Sie wird zusammen mit anderen KIs eine neue Zeit des Lernens und Lehrens einleiten.

Warum wird in dem hier vorliegenden Buch *Programmieren im Kunstunterricht* der KI ChatGPT ein eigenes Kapitel gewidmet? Nun, ChatGPT kann auch Programmcodes für unterschiedliche Programmiersprachen schreiben. Auch für p5.js? Probieren wir es aus. Hierzu muss man sich unter dem folgenden Link anmelden und einen Account erstellen ( <https://chat.openai.com/auth/login> ). Nun braucht es etwas Geduld, denn ChatGPT hat weltweit sehr viele Nutzer, sodass häufig die Nachricht erscheint *ChatGPT ist derzeit voll ausgelastet*. Aber nach mehreren Versuchen schafft man die Anmeldung und es erscheint das folgende Fenster (Abb. 14.1).

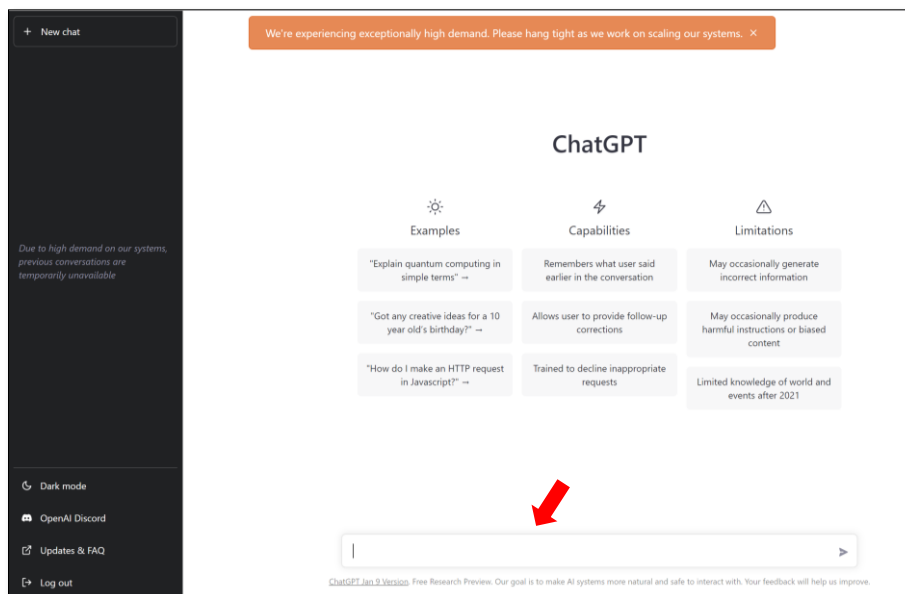


Abb. 14.1: Die Startseite von ChatGPT

Nun kann man in das untere rechteckige Kästchen (roter Pfeil), nennen wir es Kontaktfenster, seinen Programmwunsch schreiben. Für meinem ersten Versuch schrieb ich:

*Schreibe mit der JavaScript-Bibliothek p5.js ein Programm, das ein Mandala zeichnen kann.*

Nach wenigen Sekunden erhielt ich von ChatGPT einen Lösungsvorschlag (Abb. 14.2). Dieses Programm kopierte ich und fügte es in den Online-Editor von p5.js ein. Es funktionierte. Doch das gezeichnete Mandala war mehr als enttäuschend (Abb. 14.3 rechts). Etwas mehr Ehrgeiz hätte ich mir von ChatGPT schon gewünscht. Da sich ChatGPT an das, was vorher passiert ist, erinnert, schrieb ich nun in das Kontaktfenster von ChatGPT:

*Verbessere das Programm, sodass ein komplexeres Mandala gezeichnet wird.*

Nun erhielt ich einen neuen Programmcode (Abb. 14.4 auf Seite 145).

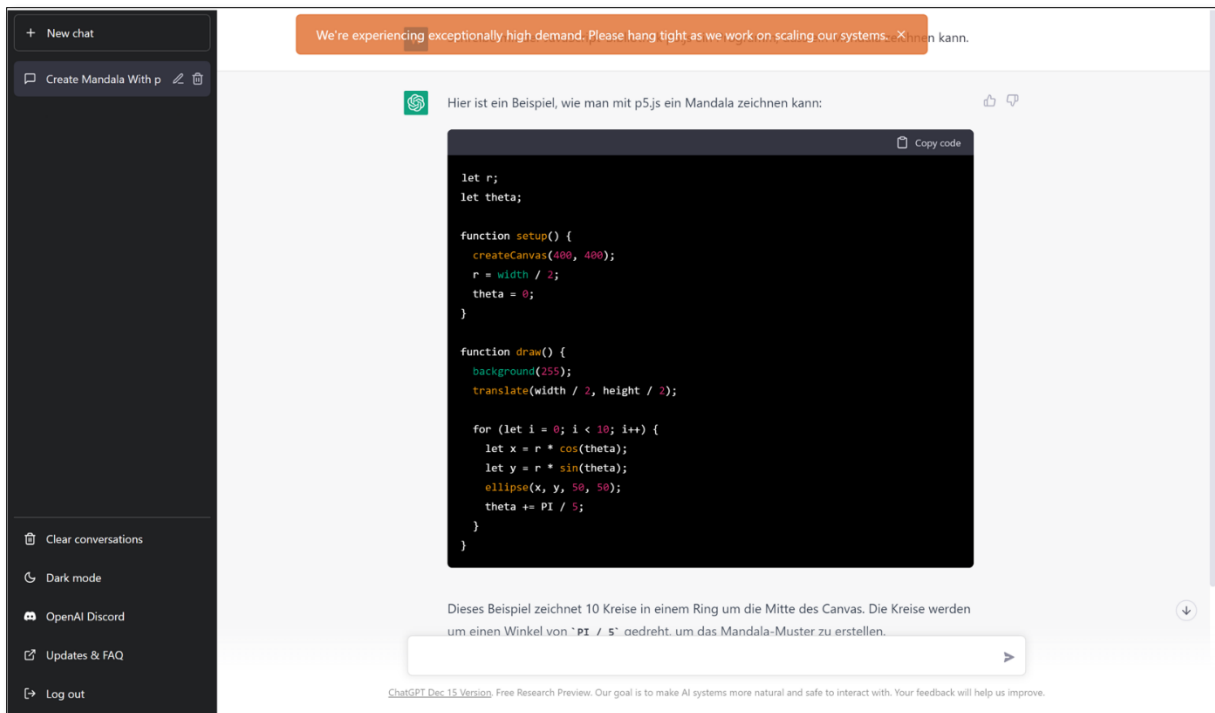


Abb. 14.2: Erster Vorschlag von ChatGPT für die Zeichnung eines Mandalas

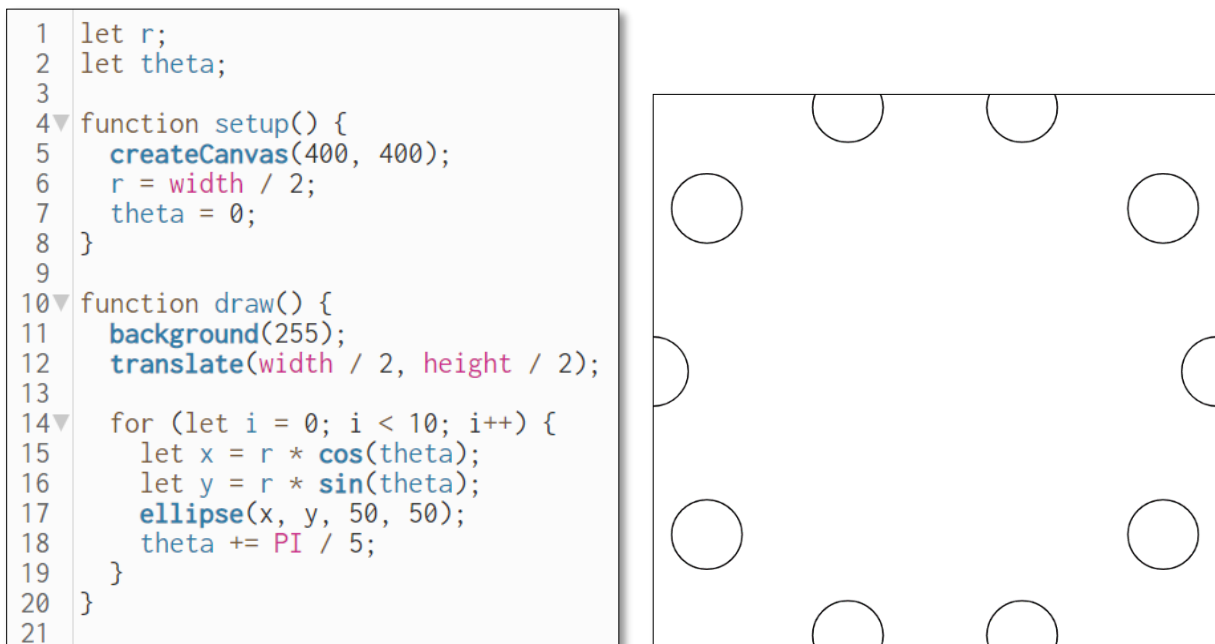


Abb. 14.3: Programm von Abbildung 14.2 im Online-Editor von p5.js und die zugehörige Zeichnung



Hier ist eine Möglichkeit, das Mandala-Programm von vorher zu verbessern, indem man mehrere Ring-Muster hinzufügt und sie in unterschiedlichen Größen und Farben zeichnet:

```
Copy code

let r;
let theta;

function setup() {
 createCanvas(400, 400);
 r = width / 2;
 theta = 0;
}

function draw() {
 background(255);
 translate(width / 2, height / 2);

 for (let n = 0; n < 3; n++) {
 for (let i = 0; i < 10; i++) {
 let x = r * cos(theta);
 let y = r * sin(theta);
 fill(200 - n * 50, 100 + n * 50, 100 + n * 25);
 ellipse(x, y, 50 + n * 25, 50 + n * 25);
 theta += PI / 5;
 }
 r -= 50;
 }
}
```

Dieses Beispiel zeichnet drei Ringe von Kreisen um die Mitte des Canvas. Jeder Ring hat 10 Kreise, die um einen Winkel von  $\pi / 5$  gedreht sind. Die Größe und Farbe der Kreise ändert sich von Ring zu Ring.

Hinweis: Dieses Beispiel setzt voraus, dass p5.js bereits in der HTML-Datei eingebunden wurde.

Abb. 14.4: Zweiter Vorschlag von ChatGPT für die Zeichnung eines Mandalas

Als ich eine Kopie des neuen Programms (Abb. 14.4) von ChatGPT in den Online-Editor von p5.js eingab, funktionierte es leider nicht. Eine Verbesserung des Programms gelang ChatGPT auch nach mehreren Aufforderungen nicht. Dabei war der Fehler mit geringen Programmierkenntnissen leicht zu finden. Der Code von Zeile 6 gehört in die Zeile 13 (Abb. 14.5 links). Nun zeichnete das Programm ein Mandala. Doch besonders hübsch war das neu gezeichnete Mandala aber auch nicht (Abb. 14.5 rechts).

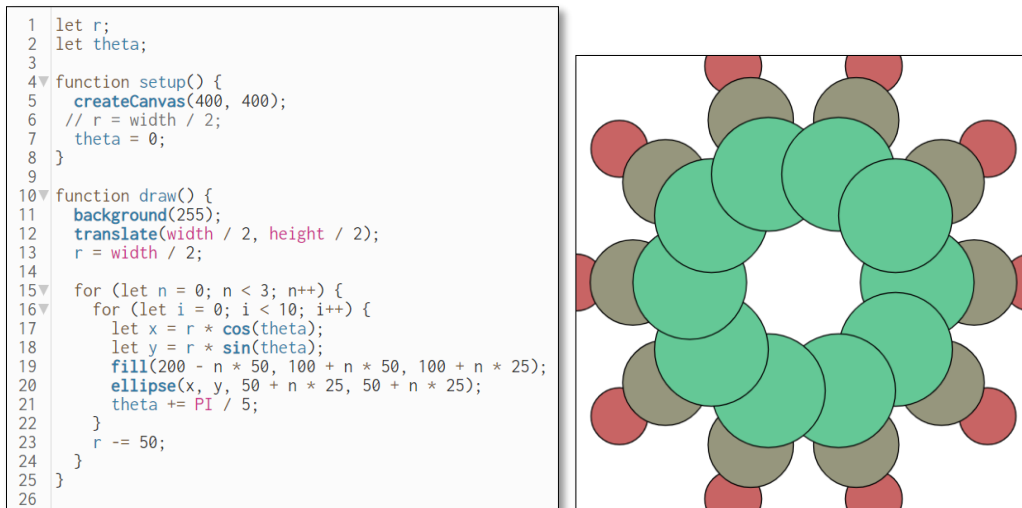


Abb. 14.5: Neues Programm von ChatGPT zur Zeichnung eines Mandalas

Also schreiben wir doch lieber mit den Kenntnissen, die wir in dem vorliegenden Buch *Programmieren im Kunstunterricht* erworben haben, schnell einen eigenen Code zur Zeichnung eines schöneren Mandalas (Abb. 14.6 und Abb. 14.7).



Abb. 14.6: Selbstgeschriebenes Programm zur Zeichnung eines Mandalas

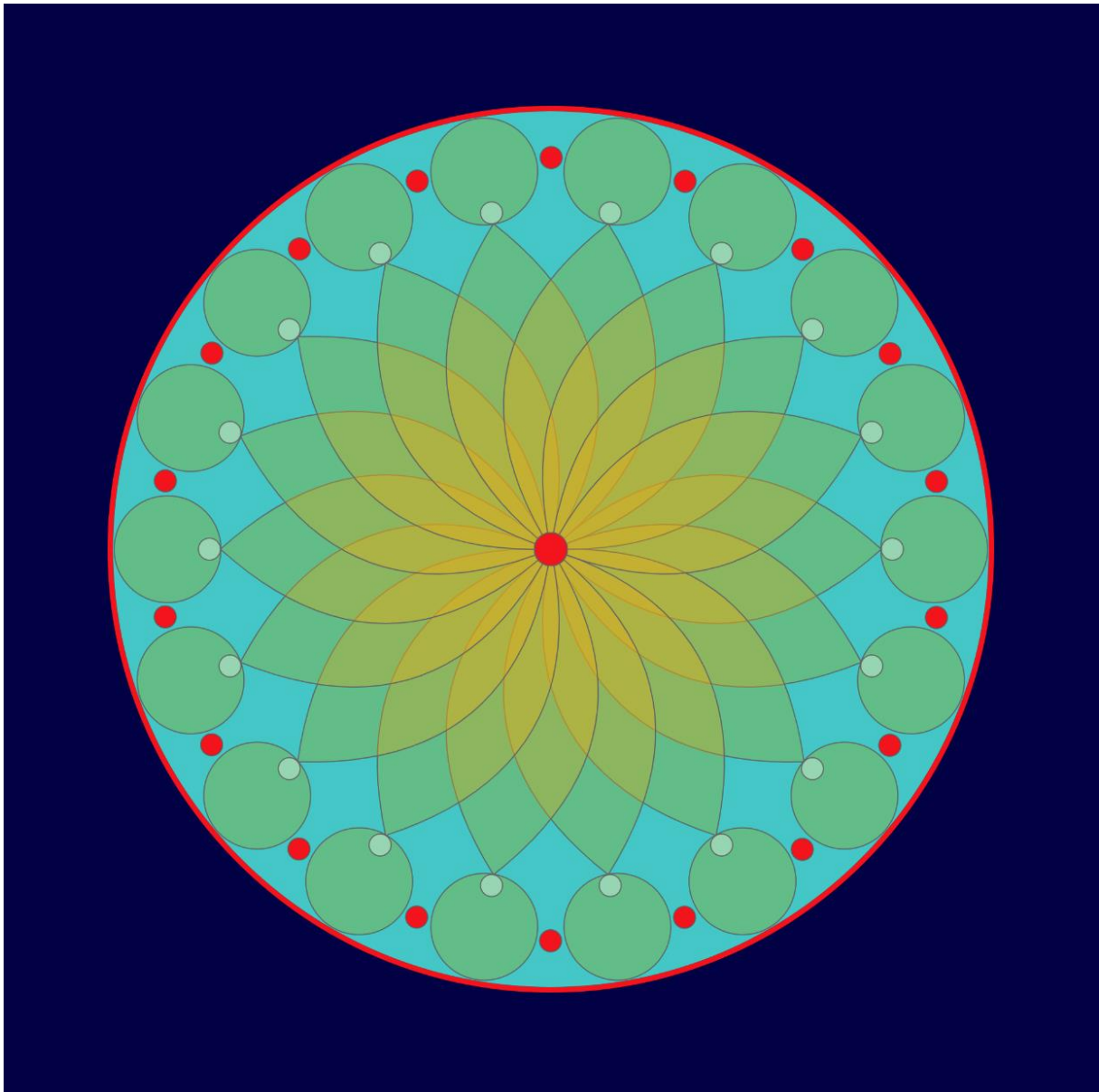


Abb. 14.7: Das von dem Programm in Abbildung 14.6 gezeichnete Mandala

## 14.2 Zweiter Versuch

Nachdem der erste Versuch von ChatGPT, ein Programm zur Erzeugung eines Mandalas zu schreiben, nicht sehr überzeugend war, startete ich einen zweiten Versuch. Dieses Mal stellte ich der KI eine etwas anspruchsvollere Aufgabe. In das Kontaktfenster gab ich den folgenden Satz ein:

*Schreibe mit der JavaScript-Bibliothek p5.js ein Programm, das die Mandelbrotmenge erzeugen kann.*

Den nach wenigen Sekunden erstellten Programmcode gab ich per Copy and Paste in den Online-Editor von p5.js ein. Das Programm funktionierte auf Anhieb (🌸'∪'🌸) Die bildliche Darstellung der Mandelbrotmenge in Form des Apfelmännchens war zwar etwas gestaucht, doch dies ließ sich leicht beheben (Abb. 14.8 und Abb. 14.9).

```

1 // Mandelbrotmenge
2
3 let xmin = -2.5;
4 let xmax = 1;
5 let ymin = -1;
6 let ymax = 1;
7
8 let maxIterations = 100;
9
10 function setup() {
11 createCanvas(1000, 600);
12 pixelDensity(1);
13 }
14
15 function draw() {
16 loadPixels();
17 for (let x = 0; x < width; x++) {
18 for (let y = 0; y < height; y++) {
19 let a = map(x, 0, width, xmin, xmax);
20 let b = map(y, 0, height, ymin, ymax);
21 let ca = a;
22 let cb = b;
23
24 let n = 0;
25 let z = 0;
26
27 while (n < maxIterations) {
28 let aa = a * a - b * b;
29 let bb = 2 * a * b;
30
31 a = aa + ca;
32 b = bb + cb;
33 if (abs(a + b) > 16) {
34 break;
35 }
36 n++;
37 }
38
39 let bright = map(n, 0, maxIterations, 0, 1);
40 bright = map(sqrt(bright), 0, 1, 0, 255);
41
42 if (n === maxIterations) {
43 bright = 0;
44 }
45
46 let pix = (x + y * width) * 4;
47 pixels[pix + 0] = bright;
48 pixels[pix + 1] = bright;
49 pixels[pix + 2] = bright;
50 pixels[pix + 3] = 255;
51 }
52 }
53 updatePixels();
54 }
55

```

Abb. 14.8: Von ChatGPT generiertes Programm für die Zeichnung der Mandelbrotmenge

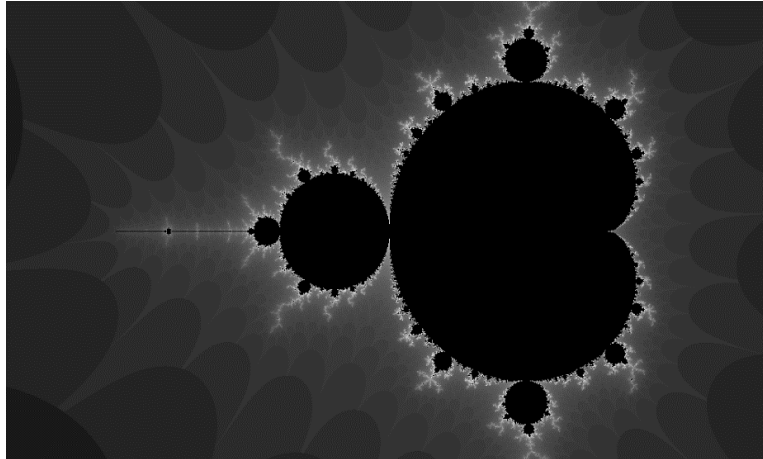


Abb. 14.9: Die gezeichnete Mandelbrotmenge, auch Apfelmännchen genannt

```

1 // Mandelbrotmenge mit Kommentaren
2
3 let xmin = -2.5; // Linke Begrenzung der Mandelbrotmenge
4 let xmax = 1; // Rechte Begrenzung der Mandelbrotmenge
5 let ymin = -1; // Untere Begrenzung der Mandelbrotmenge
6 let ymax = 1; // Obere Begrenzung der Mandelbrotmenge
7
8 let maxIterations = 100; // Maximale Anzahl an Iterationen die durchgeführt werden
9
10 function setup() {
11 createCanvas(1000, 600); // Erstellt ein Canvas von 400x400 Pixel
12 pixelDensity(1); // Setzt die Pixel Density auf 1 (empfohlen für die Performance)
13 }
14
15 function draw() {
16 loadPixels(); // Lädt das Pixel Array
17
18 // Verschachtelung von zwei Schleifen, um jede x,y Koordinate auf dem Canvas durchzugehen
19 for (let x = 0; x < width; x++) {
20 for (let y = 0; y < height; y++) {
21
22 // Map die x,y Koordinaten des Pixels zu einem Punkt auf der komplexen Ebene
23 let a = map(x, 0, width, xmin, xmax);
24 let b = map(y, 0, height, ymin, ymax);
25
26 // Speichert den Anfangswert von a,b für später
27 let ca = a;
28 let cb = b;
29
30 let n = 0; // Initialisiert den Iterationszähler
31 let z = 0; // Initialisiert den Abstand zum Ursprung
32
33 // Führt die Mandelbrot-Berechnung für eine maximale Anzahl von maxiterations durch
34 while (n < maxIterations) {
35 let aa = a * a - b * b; // Realteil der Mandelbrotmenge
36 let bb = 2 * a * b; // Imaginärteil der Mandelbrotmenge
37
38 a = aa + ca; // Aktualisiert den Realteil
39 b = bb + cb; // Aktualisiert den Imaginärteil
40
41 // Überprüft, ob der Punkt außerhalb der Mandelbrotmenge liegt
42 if (abs(a + b) > 16) {
43 break;
44 }
45 n++;
46 }
47
48 // Map die Anzahl der Iterationen auf einen Helligkeitswert
49 let bright = map(n, 0, maxIterations, 0, 1);
50
51 // Wendet eine Wurzel-Funktion an um den Kontrast zu erhöhen
52 bright = map(sqrt(bright), 0, 1, 0, 255);
53
54 // Wenn der Punkt außerhalb der Mandelbrotmenge liegt, dann wird er schwarz
55 if (n === maxIterations) {
56 bright = 0;
57 }
58
59 // Berechnet die Position des Pixels im Pixel-Array
60 let pix = (x + y * width) * 4;
61 pixels[pix + 0] = bright;
62 pixels[pix + 1] = bright;
63 pixels[pix + 2] = bright;
64 pixels[pix + 3] = 255;
65 }
66 }
67 // Aktualisiert das Anzeigefenster mit den Daten im Pixel-Array.
68 updatePixels();
69 }

```

Abb. 20: Der von ChatGPT mit Kommentaren versehene und ab Zeile 60 per Hand ergänzte Code

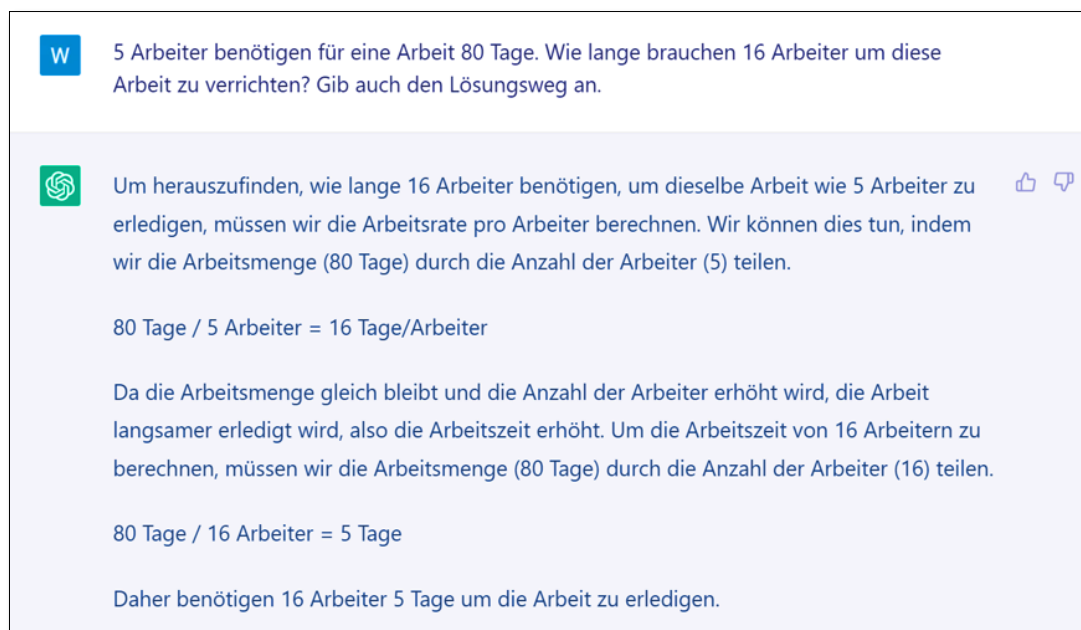
Da der von ChatGPT generierte Code für die Mandelbrotmenge für Programmieranfänger nicht leicht zu verstehen ist, bat ich ChatGPT, den Code mit Kommentaren in deutscher Sprache zu versehen. Diesem Wunsch folgte ChatGPT auch, brach aber das Programm in der Zeile 60 nach *let pix* ab (siehe Abb. 14.20). Auch wiederholte Versuche scheiterten. Den Rest musste ich somit von Hand ergänzen.

### 14.3 Fazit

Der erste Versuch, ein Programm von ChatGPT schreiben zu lassen, war nicht überzeugend, doch das Ergebnis des zweiten Versuchs zeigte, dass ChatGPT ein nützliches Werkzeug für kritische Programmierer sein kann. Um sinnvolle Fragen an ChatGPT zu stellen, um den generierten Code auf Richtigkeit zu überprüfen und um ihn gegebenenfalls zu verbessern, sind jedoch immer noch Programmierkenntnisse erforderlich.

Für den direkten Einsatz im Schulunterricht eignet sich ChatGPT zurzeit jedoch nicht, da das Programm momentan (Januar 2023) durch die weltweit sehr große Zahl von Nutzern meistens überlastet ist, sodass eine Anmeldung zu einem gewünschten Zeitpunkt oft nicht gelingt. Außerdem nimmt es ChatGPT mit seinen gut formulierten und plausiblen Antworten nicht so genau (Abb. 21). Als Lehrerin oder Lehrer sollte man aber trotzdem Schülerinnen und Schüler ermutigen und insbesondere anleiten, ChatGPT zuhause besonders kritisch zu nutzen, denn solche KI-gestützte Assistenzsysteme werden in zunehmendem Maße Einzug in unseren Alltag halten und der Bedarf an Fachkräften die Erfahrungen mit der sinnvollen und kritischen Nutzung von künstlicher Intelligenz haben, wird steigen.

Für den Schulunterricht bedeutet dies aber auch, dass rein formale Aufgabenstellungen wie zum Beispiel *Schreibe einen Aufsatz über ...* weniger Sinn ergeben. Zumal wir erst am Anfang einer rasanten Entwicklung von KI-gestützten Assistenzsystemen stehen. Gute Informationen hierzu findet man unter: <https://www.vkkiwa.de/publikationen/>



The screenshot shows a chat interface with a user question and a ChatGPT response. The user asks: "5 Arbeiter benötigen für eine Arbeit 80 Tage. Wie lange brauchen 16 Arbeiter um diese Arbeit zu verrichten? Gib auch den Lösungsweg an." The ChatGPT response starts with: "Um herauszufinden, wie lange 16 Arbeiter benötigen, um dieselbe Arbeit wie 5 Arbeiter zu erledigen, müssen wir die Arbeitsrate pro Arbeiter berechnen. Wir können dies tun, indem wir die Arbeitsmenge (80 Tage) durch die Anzahl der Arbeiter (5) teilen." It then shows the calculation:  $80 \text{ Tage} / 5 \text{ Arbeiter} = 16 \text{ Tage/Arbeiter}$ . Next, it states: "Da die Arbeitsmenge gleich bleibt und die Anzahl der Arbeiter erhöht wird, die Arbeit langsamer erledigt wird, also die Arbeitszeit erhöht. Um die Arbeitszeit von 16 Arbeitern zu berechnen, müssen wir die Arbeitsmenge (80 Tage) durch die Anzahl der Arbeiter (16) teilen." It then shows another calculation:  $80 \text{ Tage} / 16 \text{ Arbeiter} = 5 \text{ Tage}$ . Finally, it concludes: "Daher benötigen 16 Arbeiter 5 Tage um die Arbeit zu erledigen." There are like and reply icons to the right of the response.

Abb. 21: Auch wenn der Text gut formuliert ist, so ist doch der Lösungsweg und die Lösung falsch. 25 Tage ist die richtige Lösung.

## 15 Bilder generieren mittels dem Text-zu-Bild-Generator Dall-E 2

Dall-E 2 ist eine von OpenAI entwickelte künstliche Intelligenz zur Generierung von fotorealistischen und künstlerischen Bildern durch eine einfache, aber wohlüberlegte Texteingabe. Seit September 2022 ist Dall-E 2 unter <https://openai.com/dall-e-2/> allgemein verfügbar.

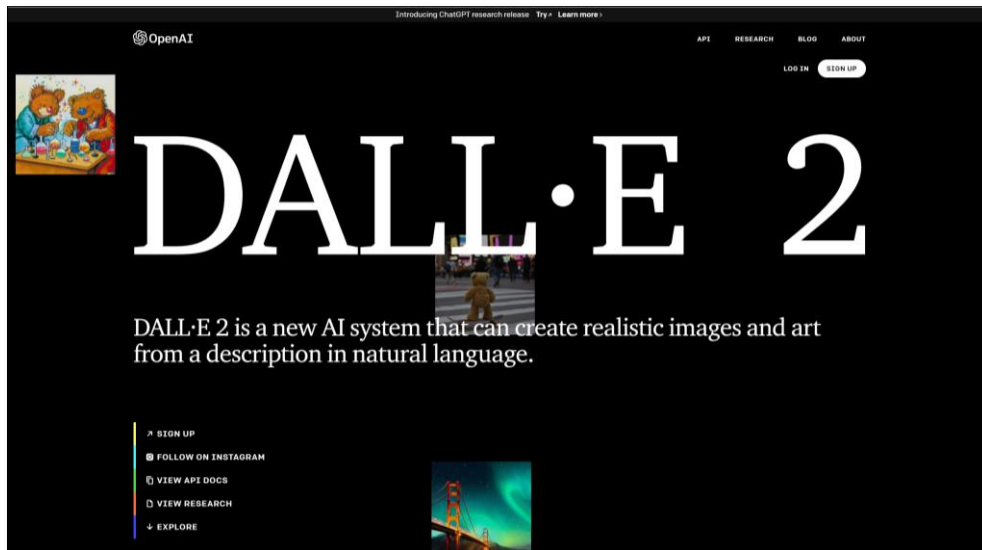


Abb. 15.1: Startseite der Homepage von Dall-E 2

Nachdem man sich auf der Startseite der Homepage von Dall-E 2 (Abb. 15.1) bei *Sign up* einen Account erstellt hat, erhält man als neuer Benutzer kostenlos 50 Creditpoints zur Erstellung seiner ersten Bilder. Sind diese verbraucht, dann erhält man pro Monat kostenlos weitere 15 Creditpoints. Wenn man aber mehr benötigt, dann kann man sich für sich für 15 US-Dollar 115 Creditpoints kaufen. Diese reichen für die Erstellung von ca. 450 Bildern.

Nachdem man sich angemeldet hat, öffnet sich das folgende Fenster (Abb. 15.2).

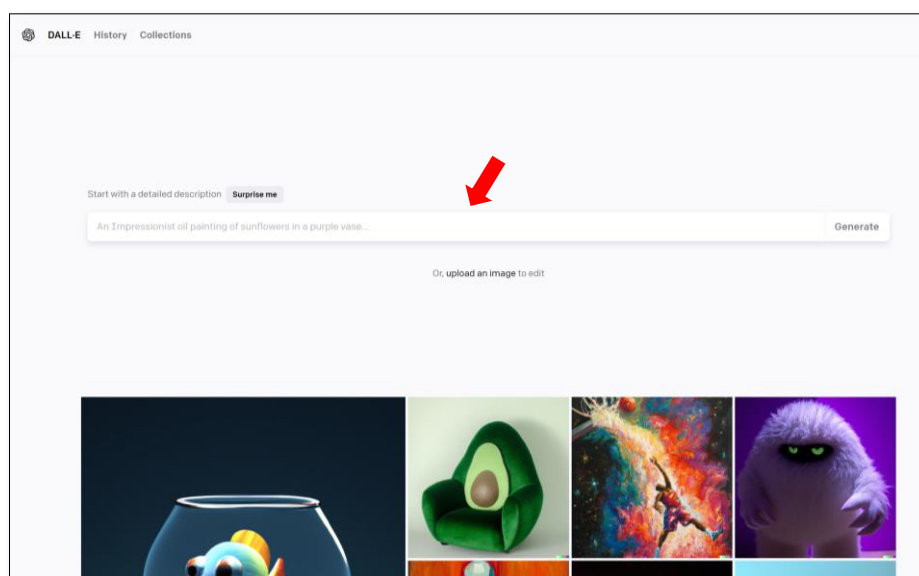


Abb. 15.2: Fenster für die Eingabe der eigenen Prompts

Hier (siehe roter Pfeil in Abbildung 15.2) gibt man seinen Prompt in Englisch oder Deutsch ein. Ein Prompt ist ein Satz oder eine Kombination von Wörtern, mit einer maximalen Größe von 400 Zeichen. Dieser Prompt soll möglichst genau beschreiben, welches Bild Dall-E 2 für einen generieren soll. Hilfen hierzu findet man im Internet. So zum Beispiel in dem kostenlosen Buch *Dall-E 2 prompt book*, welches man sich als PDF-Datei unter dem folgenden Link herunterladen kann: <https://dallery.gallery/the-dalle-2-prompt-book/>

Hier ein erster Prompt von mir:

*3D art style, a snail with wheels and a rusted metal snail shell, powered by steam*

Nachdem ich auf *Generate* geklickt habe, bot Dall-E 2 mir nach wenigen Sekunden vier Lösungen an (Abb. 15.3).



Abb. 15.3: Vier Lösungen zu dem obigen Prompt

Von diesen vier Bildern gefiel mir das dritte Bild am besten. Wurde hier etwa der aktuelle Zustand der Deutschen Bahn dargestellt 😊? Nachdem ich es angeklickt hatte, wurde es auf 1024 x 1024 Pixel vergrößert. Nun hätte ich es auf meine Festplatte laden können. Doch anstelle des quadratischen Formats bevorzugte ich ein rechteckiges Format. Bei Dall-E 2 kann man seine erstellten Bilder auch unter *Edit Image* stilgerecht erweitern (Abb. 15.4). Auf der Startseite der Homepage von Dall-E 2 wird dies überzeugend am Bild *Das Mädchen mit dem Perlenohrgehänge* von Jan Vermeer demonstriert.

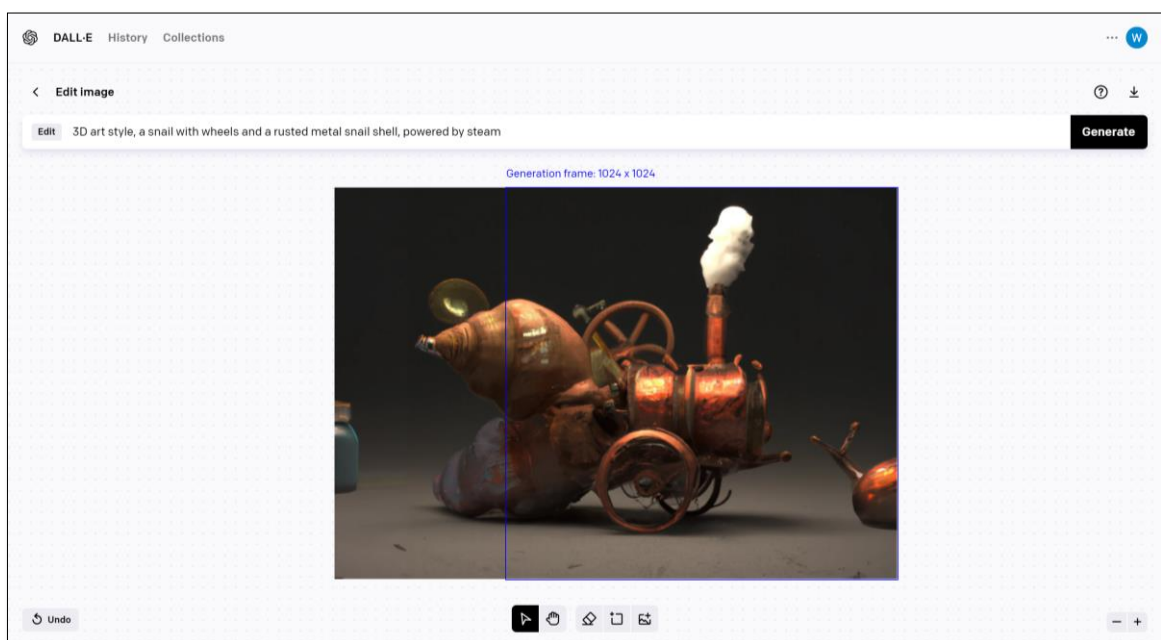


Abb. 15.4: Stilgerechte Erweiterung des Bildes

Das fertige Bild (Abb. 15.5) speicherte ich dann auf meiner Festplatte ab.



Abb. 15.5: Stilgerecht erweitertes Bild

Bei meinem zweiten Versuch, mit Dall-E 2 ein Bild zu generieren, stand Pablo Picassos Kubismus Periode Pate. In das Kommunikationsfeld schrieb ich den folgenden Prompt:

*Cubism, cubist, 1910, Picasso, a young woman with red hair flying over a burning forest on a fire-breathing dragon*

Unter den vier Vorschlägen von Dall-E 2 wählte ich den folgenden Vorschlag aus.



Abb. 15.6: Frau in einem brennenden Wald, die vor einem dämonischen Drachen flieht

Bei den anderen drei Bildern saß, wie gewünscht, eine Frau auf einem Drachen. Doch bei dem Bild von Abbildung 15.6 hat mich die freie Interpretation meines Prompts von Dall-E 2 beeindruckt.

Die besondere Stärke von Dall-E 2 ist die fotorealistische Darstellung. Probieren wir dies einmal mit der folgenden Eingabe aus.

Porcelain, chandelier made from the arms of an octopus, the chandelier hangs from the ceiling by a long chain, there is a candle on each octopus arm, background gray blue



Abb. 15.7: Kronleuchter mit Oktopusarmen

Für einen Dall-E 2 Neuling wie mich, finde ich meine ersten Bilder schon ganz ok. Aber mit etwas mehr Erfahrung gelingen bestimmt noch bessere Prompts und damit bessere Bilder. Meine Prompts bestehen bisher nur aus wenigen Teilen. Den jeweiligen Modifikatoren wie *3D art style*, *Cubism*, *cubist*, *1910*, *Picasso* und *Porcelain* sowie den jeweiligen Beschreibungen der gewünschten Motive.

Neben Dall-E 2 gibt es auch noch andere Text-zu-Bild-Generatoren. Ein für künstlerische Darstellungen sehr leistungsfähiger Text-zu-Bild-Generator, der sich noch in der Beta-Phase befindet, ist *Midjourney*. Die hiermit generierten Bilder sind faszinierend. Siehe: <https://midjourney.com/showcase/recent/>

## 16 Ausblick

Wenn man dieses Buch gründlich gelesen und die Übungen gemeistert hat, dann hat man ein gutes Rüstzeug, um digital kreativ zu werden. Natürlich ist man noch kein Meister. Dazu muss man sich weiter fortbilden und sich von den Werken anderer digital arbeitenden Künstler anregen lassen.

Manche Leser werden sich aber nach dem Lesen von Kapitel 14 und 15 vielleicht fragen, ob sich der Erwerb von grundlegenden Programmierkenntnissen mit p5.js noch lohnt. Diese Frage kann nur mit JA beantwortet werden. Denn zur digitalen Allgemeinbildung gehört auch ein Grundverständnis im Programmieren. Dies haben wir zum Beispiel bei der Nutzung der KI ChatGPT in Kapitel 14 erfahren.

Obwohl Fragen zum Urheberrecht rechtlich noch zu klären sind, werden Text-zu-Bild-Generatoren wie auch andere KI-basierte Assistenzsysteme wie zum Beispiel ChatGPT zunehmend Einzug in den Alltag und in die Berufswelt halten. Aus diesem Grund sollten Schülerinnen und Schüler im Schulunterricht lernen, KI-basierte Assistenzsysteme intelligent zu nutzen und deren Ergebnisse, aber auch den eigenen Lernprozess hierbei kritisch zu reflektieren. Dies stellt insbesondere für Lehrende eine große Herausforderung dar, da es neue Unterrichtskonzepte verlangt. So auch für den Kunstunterricht. Text-zu-Bild-Generatoren sollten aber nur ein weiteres Werkzeug neben p5.js, Pinsel, Stift und Spachtel zur Bildgestaltung sein.

Auch wenn digital erstellte Kunstwerke auf Kunstauktionen schon beachtliche Preise erzielt haben, so ist das geschaffene Produkt doch nur ein Bestandteil des Schöpfungsprozesses. Der kreative Prozess selbst muss für den Künstler eine Quelle der Freude sein, egal mit welchem Werkzeug er sein Werk erstellt.



## 17 Link- und Literaturverzeichnis

### Bücher

Lauren McCarthy, Casey Reas und  
Ben Fry  
2016

**Getting Started with p5.js**  
Verlag: Maker Media

Benedikt Groß, Hartmut Bohnacker,  
Julia Laub und Claudius Lazzeroni  
2018

**Creative Coding im Web - Generative Gestaltung**  
Verlag: Hermann Schmidt

### Links

Die unten aufgeführten Links wurden zuletzt am 30.01.2023 aufgerufen.

- |    |                                                                                                                                                |                                                                                                                                                                                                |
|----|------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 01 | p5.js Webseite                                                                                                                                 | Die ausführlichsten Informationen zu p5.js findet man auf der Homepage von p5.js.<br><a href="https://p5js.org">https://p5js.org</a>                                                           |
| 02 | p5.js Online-Editor                                                                                                                            | p5.js-Sketches können online in einem Webbrowser erstellt werden.<br><a href="https://editor.p5js.org/">https://editor.p5js.org/</a>                                                           |
| 03 | Zahlreiche gute YouTube Videos von Daniel Shiffman zu p5.js findet man unter der <b>Suchbegriffskombination <i>daniel shiffman p5.js</i></b> . |                                                                                                                                                                                                |
| 04 | ChatGPT                                                                                                                                        | Login für ChatGPT<br><a href="https://chat.openai.com/auth/login">https://chat.openai.com/auth/login</a>                                                                                       |
| 05 | Dall-E 2                                                                                                                                       | Startseite von Dall-E 2<br><a href="https://openai.com/dall-e-2/">https://openai.com/dall-e-2/</a>                                                                                             |
| 06 | Anregungen für Prompts                                                                                                                         | Auf dieser Seite erhält man gute Anregungen für die Erstellung von Prompts.<br><a href="https://dallery.gallery/the-dalle-2-prompt-book/">https://dallery.gallery/the-dalle-2-prompt-book/</a> |



## 18 Stichwortverzeichnis

! 35, 40  
&& 35, 40  
/\* Text \*/ 31  
// 31  
^ 35, 40  
|| 35, 40  
= 36, 41  
== 36, 41  
2D-Array 86

### A

abs() 35, 40, 59  
abs(sin()) 50  
Alpha-Wert 9, 10  
angleMode(DEGREES) 66  
Ankerpunkte 45  
arc() 7, 11  
Array 77, 86  
Attraktor 116  
Audacity 126

### B

background() 4, 10  
beginShape() 6, 11  
beginShape(TRIANGLE\_STRIP) 82, 86  
bezier() 45, 59  
bezierVertex() 49, 59  
Bibliotheken 134  
Bleistift und Papier 65, 111  
blendMod(ADD) 16, 20  
blendMode(DIFFERENCE) 16, 20  
box() 100  
Brightness 17, 20

### C

ChatGPT 143  
Codeblock 3, 10  
Code-Editors 3, 23  
colorMode() 18, 20  
cone() 87  
cos() 50, 59  
createButton() 136  
createCanvas 3, 10  
createCanvas(windowWidth, windowHeight) 73, 74  
createSlider() 132  
curveVertex() 8, 11

### D

Dall-E 2 151  
deklarieren 26, 39  
deklariert 26  
displayHeight 32, 40  
displayWidth 32, 40

### E

ellipse() 4, 10  
else 35, 40  
else if() 40  
Endlosschleife 113  
endShape() 6, 11  
endShape(CLOSE) 6, 11

### F

false 41  
fill() 4, 10  
Firefox III, 100, 127  
for-Schleife 28, 39  
Fraktale 115  
frameCount 87  
frameRate() 29  
function draw() 3, 10  
function preload() 100  
function setup() 3, 10  
Funktion, eigene 63

### G

Gleichheitsoperator 36  
globale Variable 26, 39

### H

height() 31, 40  
HSB-Farbraum 17, 20  
Hue 17

### I

if-Anweisung 34, 40  
image() 91, 100  
initialisieren 26, 39  
Instanzvariablen 104

### K

Kanten, abgerundet 48  
key 40  
keysPressed() 35, 40  
keyReleased() 73  
keyTyped() 73, 74  
klang.loop() 128  
klang.play() 128  
Konsole 3  
Konstruktor 104  
Kontrollpunkte 45  
Koordinatensystem 5, 10

## L

Leinwand 3  
let 26, 37  
line() 5, 10  
Lissajous-Figur 130  
loadImage() 100  
loadPixels() 96, 101  
logische Operatoren 40  
lokale Variable 26, 39

## M

Mandala 147  
Mandelbrotmenge 147  
Methoden 104  
Midjourney 154  
millis() 35, 40  
mouseIsPressed 73  
mouseX 18, 20  
mouseY 18, 20

## N

noise() 59  
noLoop() 35, 40  
noStroke() 6, 11

## O

objektorientierte Programmierung 103  
Online-Editor 3, 10  
OOP 103

## P

p5.dom.js 132  
p5.sound.js 125, 134  
Perlin noise 50  
pixelDensity() 96, 101  
pixels[] 98, 101  
point() 5, 10  
pop() 69  
position() 131  
preload() 91  
print() 3, 18  
println() 10  
Prompt 152  
push() 69

## Q

quad (x1, y1, x2, y2, x3, y3, x4, y4) 6, 11

## R

radians() 11  
radians(Winkel in Grad) 7  
random() 28, 40  
rect() 5, 10

Referenz 20  
rekursive Programmierung 113  
RGB-Farbraum 4, 20  
rotate() 84, 86  
rotateX() 80, 86  
rotateY() 80, 86  
rotateZ() 86  
round() 35

## S

Saturation 17, 20  
scale() 112  
Schweif 128, 134  
second() 58, 60  
Selbstähnlichkeit 115, 119  
Sierpinski-Dreieck 116  
sin() 50, 59  
Skalierung 82  
Slider 132, 134  
Smartphone 137  
sphere() 100  
sqrt() 120  
statischer Modus 16  
stroke() 5, 11  
strokeWeight() 6, 11  
Systemvariable 25, 39

## T

Tablet 137  
text() 19, 20  
textAlign(CENTER) 29, 40  
textSize() 19, 20  
texture() 95, 100  
Text-zu-Bild-Generator 151  
this. 104  
tint 102  
torus() 80, 86  
translate() 48, 59  
triangle() 5, 11  
TRIANGLE\_STRIP 82, 86  
true 41

## U

updatePixels() 101

## V

var 37  
Variable 25  
vertex() 6, 11  
Vorschaufenster 3

## W

WEBGL 80, 86  
width 31  
width() 40

## **X**

x++ 28

## **Y**

y-Achse 5, 10

## **Z**

Zuweisungsoperator 36

