Aufgabenblatt 5 - Bonus

Nach einer langen Woche dürfen Sie sich heute einmal austoben: Auf diesem Übungsblatt sind einige Spiele und erweiterte Aufgaben zu finden, welche mit Ihren errungenen Erfahrungen gut machbar sein sollten. Zudem könnten Sie selbstverständlich auch nochmal in die Übungsblätter der Vortage schauen oder eigene Ideen und Spiele umsetzen. Viel Erfolg!

Hinweis: *Verschaffen Sie sich einen Überblick, bevor Sie mit einer Aufgabe beginnen, um nichts zu verpassen.*

Verzeichnis der Bonusaufgaben

501

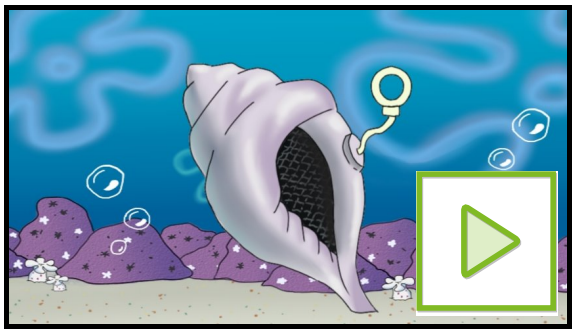
100 Z.

Aufgabe 1 – Magische Miesmuschel

Erstellen Sie ein Programm, das wie eine *Magische Miesmuschel*, auch bekannt als *Magic 8-Ball* funktioniert. Per Mausklick kann die Muschel aktiviert werden, woraufhin sie eine zufällig gewählte Antwort anzeigt (mithilfe von `text(str, x, y)`). Legen Sie einige Antworten fest, und finden Sie eine gute Weise, diese zu speichern und abzurufen. Für die Darstellung könnten Sie auch eine Hintergrundgrafik einbinden.

Optional können Sie auch Animationen und Details einfügen, welche gezeigt werden, wenn die Muschel mit Nutzer:innen „spricht“.

Hinweis: *Werfen Sie für Grafiken einen Blick in die Processing-Reference - insbesondere auf die Klasse* `PImage`*. Übrigens: Processing hat auch über den Reiter „Sketch“ die Option „Datei Einfügen“.*



Bonus: Können Sie die Muschel so umprogrammieren, dass sie bei dem Klicken auf eine ”geheime”Position z.B. nur positive Antworten gibt (aber ansonsten immer noch zufällig ist)?

Hinweis: *Bei einem Klick führt Processing die Funktion* `void mousePressed()` *aus. Es kann Sinn ergeben, dort eine* `boolean mouse` *mit einem Wahrheitswert zu füllen, wenn diese Information im weiteren Programm verwendet werden soll.* Hinweis: *Falls Sie Tasten anstatt der Maus verwenden wollen: Processing aktualisiert bei jedem Tastendruck die* `key`*-Variable, welche dann die gedrückte Taste als* `char`*-Wert speichert.*

502

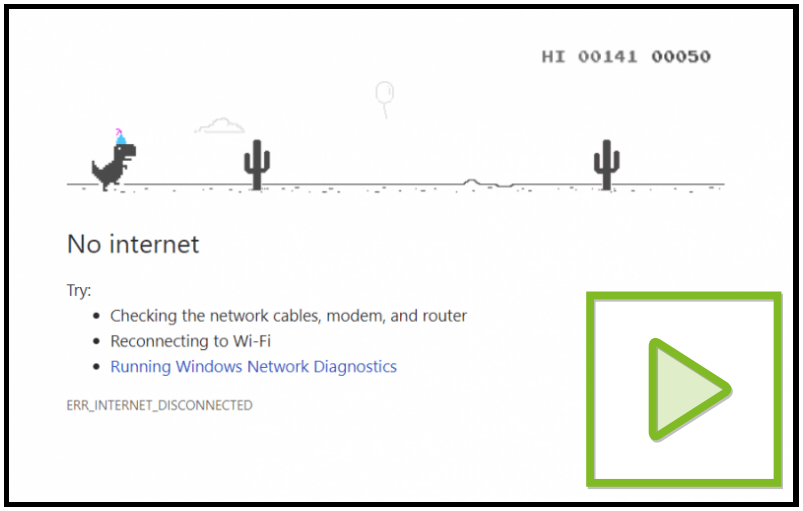
161
Z.

Aufgabe 2 – Dino-Spiel

Wer kennt es nicht, man setzt sich an den Computer, öffnet Chrome und sieht: keine Internetverbindung. Das einzige was man dann sieht, ist das *Jump n' Run Game*, bei dem man einen Dinosaurier springen und ducken lässt.

Programmieren Sie das Dino-Spiel. Dabei dürfen Boden, Dino und Hindernisse auch mit simplen Rechtecken dargestellt werden. Beim ersten Tastendruck soll der Dino anfangen, zu laufen. Dann kommen zufällig generierte Hindernisse, über die der Dino entweder springen soll, oder drunter ducken muss. Mit einer Pfeiltaste (z.B. `w`) kann der Dino zum Springen befohlen werden, und mit einer anderen (z.B. `s`) duckt er. Kollidiert der Dino mit einem Hindernis, ist das Spiel zu Ende.

Optional können Sie dem Spiel auch ein Punkte & Highscore-System verleihen, und Grafiken einfügen.



Hinweis: *Schwierigkeiten mit dem Springen des Dinos? Dazu benötigen Sie eine Variable für y-Position, y-Geschwindigkeit sowie Schwerkraft. Die Schwerkraft wird dann immer mit der Geschwindigkeit, und die Geschwindigkeit mit der Position, verrechnet, wenn der Dino den Boden noch nicht berührt.*

Hinweis: *Schwierigkeiten mit der Kollisionserkennung? Ein wichtiges Prinzip hier ist der Abgleich von Rechtecken („Hitboxes"), oft genannt* `AABB`*. Sie benötigen eine Methode, welche ermittelt, ob sich zwei Hitboxes überschneiden. Hierzu gab es in einem früheren Arbeitsblatt auch eine Aufgabe.*

511
226 Z.

Aufgabe 3 – Space Invaders

Space Invaders war ein Arcade-Spiel, in welchem Spieler:innen sich als Raumschiff am unterem Spielfeld horizontal bewegen konnten, und mit einer Kanone versuchten, die Aliens abzuschießen. Die Aliens bewegen sich dabei nach links und nach rechts. Sobald die Aliens den Rand des Spielfelds erreicht haben, ändern sie die Richtung und bewegen sich ein Stück nach unten. Erreichten die Aliens den unteren Rand des Spielfelds bevor alle getroffen wurden, verlor der/die Spieler:in.

3.1 Grundlagen

Was benötigen wir für unser Spiel?

1. Wir benötigen natürlich ein Raumschiff, welches sich bewegen und schießen können muss. Dazu werden wir eine Klasse namens `class Player` anlegen, in welche wir die notwendigen Methoden zum Anzeigen, Bewegen und Schießen implementieren.
2. Die Gegner werden als eine neue Klasse `class Enemy` implementiert und auch mit Funktionalität (Anzeigen, Bewegen) ausgestattet.
3. Da wir auch die Kugeln welche vom Raumschiff geschossen werden darstellen wollen, benötigen wir zusätzlich eine Klasse `class Bullet`, welche Positionsdaten und Geschwindigkeit speichern soll, und ebenso eine Methode zum Anzeigen besitzen soll.

Zu diesen Klassen werden einige wichtige Verhalten benötigt:

1. Das Raumschiff soll eine Kugel schießen können.
2. Eine Kugel soll, wenn sie einen Gegner trifft, diesen und sich selbst löschen.
3. Eine Kugel soll, falls sie außerhalb des Bildes ist, sich selbst löschen.
4. Alle Gegner sollen synchron bewegt werden.
5. Spieler:innen sollen das Raumschiff durch Tasten bewegen können.
6. Spielverlust und -Gewinn sollten erkannt und angezeigt werden.

Dies sollte Ihnen genug Tipps geben, das Spiel zu realisieren.

Falls Sie irgendwo hängen bleiben sollten und nicht weiter wissen, können Sie in den nächsten Abschnitten nachschauen. Dort wird es allerdings Lösungsansätze und nicht nur Tipps geben.
Versuchen Sie es also gerne erst alleine!

Auf der nächsten Seite sind Lösungsansätze zu finden...

3.2 Variablen und Klassen

Hinweis: *Hier und später werden wir Variablen verwenden, welche in der Aufgabenstellung nicht definiert werden. Extrahieren Sie anhand des Namens der Variable den Sinn dahinter, und implementieren Sie sie. Die meisten dieser Variablen sollten global angelegt werden, und können schon direkt bei der Deklarierung mit einem Wert belegt werden.*

Um Instanzen von Gegnern und Kugeln zu speichern, ist es sinnvoll, Listen statt Arrays zu verwenden. So können dynamisch mehr Gegner und Kugeln hinzugefügt oder vom Spielbrett entfernt werden.

```
Player player;
ArrayList<Bullet> bullets;
ArrayList<Enemy> enemies;
```

Diese Variablen werden wie gewohnt in `setup()` initialisiert:

```
void setup() {
   size( ... );
   player = new Player();

   bullets = new ArrayList<>(); // Zunächst leer!
   enemies = new ArrayList<>();

   for(int i = 0; i < enemyCount; i++) { // enemyCount muss definiert
       werden
      enemies.add(new Enemy()); // Hier müssen später noch Informationen
          eingefügt werden
   }
}
```

Mit einem ähnlichen, bekannten Prinzip können wir daraufhin alle vorliegenden Instanzen anzeigen und ggf. aktualisieren:

```
void draw() {
   background( ... );
   player.show();

   for(Bullet b : bullets) { // Über Listen mit for-each iterieren...
      b.show();
      // ... (Bewegung?)
   }

   for(int i = 0; i < enemies.size(); i++) { // ...oder normalem for
      enemies.get(i).show();
      // ... (Bewegung?)
   }
}
```

3.3 Spielerbewegungen

Für die Bewegungen des Spielers wird die Implementierung eines Events benötigt. Hierfür eignet sich `void keyPressed()`, da dieses beim Gedrückt-halten auch mehrmals aufgerufen wird.

Eine mögliche Implementierung des Events:

```
void keyPressed() {
   switch (key) {
      case 'a':
         player.moveLeft(); // müssen noch implementiert werden!
         break;
      case 'd':
         player.moveRight();
         break;
   }
}
```

Entsprechend müssen Informationen zur Position des Spielers in der Klasse `Player` gespeichert werden.

3.4 Gegnerbewegungen

Charakteristisch für *Space Invaders* ist das synchrone nach links und rechts-Bewegen der Gegner, welche dann die Richtung wechseln und einen Schritt nach vorne machen, bevor sie sich wieder weiterbewegen. Demzufolge werden wir in der Klasse `Enemy` auch Information zu Position sowie Richtung speichern. Zudem sollten die Schritte in einer Variable zählen, sodass wir wissen, wann ein Gegner die Richtung umkehren sollte.

Der Kopf der Klasse könnte ungefähr so aussehen:

```
class Enemy {
   int x, y;
   int direction;
   int steps;

   Enemy(int x, int y) {
      this.x = x;
      this.y = y;
      direction = 1;
   }
}
```

... und die Methode `move()` so:

```
void move() {
   if(steps >= maxSteps) { // maxSteps muss implementiert werden!
      steps = 0;
      direction = -direction;
      y += stepY; // muss implementiert werden!
   } else {
      steps++;
      x += stepX *direction;
   }
}
```

3.5 Schießen

Die Implementierung von der Klasse `Bullet` ist nicht viel anders als die anderen, sodass sie hier nicht weiter erläutert wird. Besonders ist vielmehr die Interaktion zwischen Spieler und Kugel: Hier macht es Sinn, in das `keyPressed()`-Event eine weitere Bedingung einzufügen:

```
void keyPressed() {
   switch (key) {
      ...
      case 'w':
         if (shotCountdown <= 0) { // Implementierung fehlt
            bullets.add(new Bullet(player.x, player.y));
            shotCountdown = ticksUntilNextShot; // hier auch
         }
         break;
   }
}
```

3.6 Kollisionen

Die Bestimmung von Kollisionen ist mit der Suche nach einer überlappenden Instanz von `Bullet` mit einer solchen von `Enemy` gleichzustellen. Es herrscht Überlappung, wenn die Distanz zwischen den Objekten kleiner als die Summe der Radien ist.

Eine Funktion, welche Kollisionen prüft, könnte so aussehen:

```
void checkForCollisions() {
   for(Bullet b : bullets) {
      for(Enemy e : enemies) {
         if(dist(b.x, b.y, e.x, e.y) < bulletSize + enemySize) {
            enemies.remove(e);
            bullets.remove(b);
            break;
         }
      }
   }
}
```

3.7 Erweiterungsideen

Falls Sie das Grundkonstrukt des Spieles erfolgreich programmiert haben, können Sie es gerne noch erweitern. Hier sind einige Anregungen dazu:

- Spieler:innen sollten sich nur innerhalb des Fensters bewegen können.
- Kann man die Formen der Aliens und des Spielers durch Bilder von Raumschiffen ersetzen?
- Die Geschwindigkeit des Spieles könnte sich durch bestimmte Ereignisse verändern.
- „Neu starten“ oder „Zurücksetzen“-Funktionalität.
- Wie wäre es mit Punktzahlen und High-Scores?

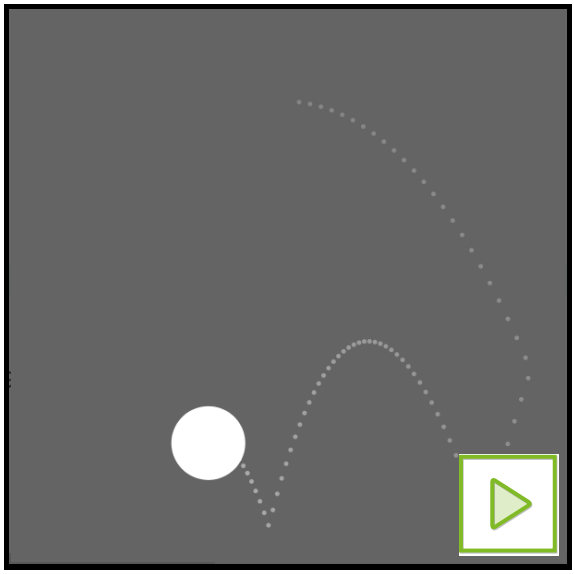
504

59 Z.

Aufgabe 4 – Gummiball

Zeichnen Sie einen Ball ein, der vom oberen Bildschirmrand herabfällt und von Wänden und Boden abprallt.

Erstellen Sie dazu Variablen für Position und Geschwindigkeit des Balls oder alternativ eine Klasse `class Ball`. Erhöhen Sie die Geschwindigkeit nach unten im `draw()`-Event, um Schwerkraft zu simulieren. Zusätzlich können Sie Ihrem Ball, wie auf dem Bild ersichtlich, einen Schweif geben.



Um den Ball abprallen zu lassen, genügt es, die Geschwindigkeit bei Berührung mit einer Wand oder dem Boden zu invertieren (*-1).

Bauen Sie zusätzlich die Funktion ein, dass der Ball sich bei einem Klick mit der linken Taste in Richtung Mauszeiger beschleunigt. Dazu ist es hilfreich, die Differenz zwischen Mauszeiger und Ball zu berechnen.

506

116 Z.

Aufgabe 5 – Flappy Bird

Anfang 2014 wurde das Spiel *Flappy Bird* explosionsartig berühmt - und nur kurze Zeit später verschwand es wieder. Jetzt haben Sie die Gelegenheit, es erneut zum Leben zu erwecken, indem Sie es in Processing realisieren.

Bei *Flappy Bird* gibt es einen Vogel, welcher vom Spieler durch Röhren manövriert werden muss. Der Vogel bewegt sich nur auf der y-Achse, während die Röhren sich nach rechts bewegen. Per Knopfdruck springt der Vogel nach oben, sonst fällt er nach unten. Für jede durchtroffene Röhre wird dem/der Spieler:in ein Punkt verliehen.

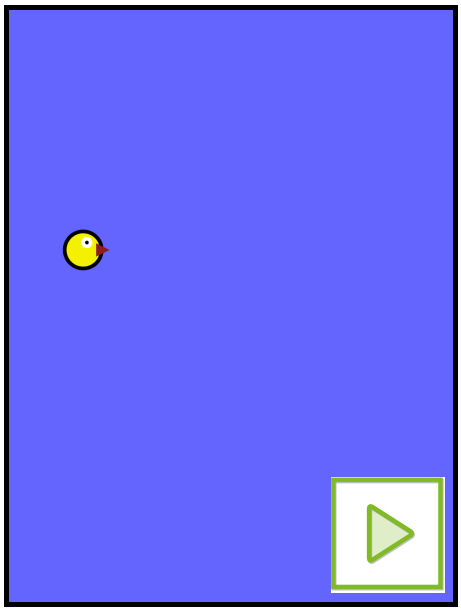
5.1 Der Vogel

Das wichtigste zuerst: der Vogel von Flappy Bird (Übrigens heißt er im Original *Faby*) hat nur zwei Zustände - entweder springt er, oder er fällt. Erstellen Sie die Klasse `class Bird`, welche die Position des Vogels `x, y`, und dessen vertikale Geschwindigkeit `vy` speichert.

Implementiere zunächst die Methoden `jump()`, `show()` und `update()`. Die Methode `jump()` soll mithilfe von `keyReleased()` aufgerufen werden. Die Methoden `show()` (zeichnet den Vogel) und `update()` (aktualisiert dessen Werte) sollen in der `draw()`-Methode aufgerufen werden.

- Mit der (globalen) Variable `gravity` können Sie den Wert festlegen, welcher bei jedem `update()` des Vogels zu `vy` addiert wird - dadurch entsteht ein natürlicher Fall, wenn darauf `vy` in jedem Zug zu `y` addiert wird.
- Die Methode `show()` sollte gut gewählte Farben und Formen haben, um einen mehr oder weniger erkennbaren Vogel an den Positionen `x, y` zu zeichnen.

Bei Ausführung sollten Sie einen Vogel haben, welcher beginnt, zu fallen. Drückt man eine beliebige Taste, „springt“ der Vogel ein bisschen, und fängt dann wieder an, zu fallen.



5.2 Die Darstellung der Röhren

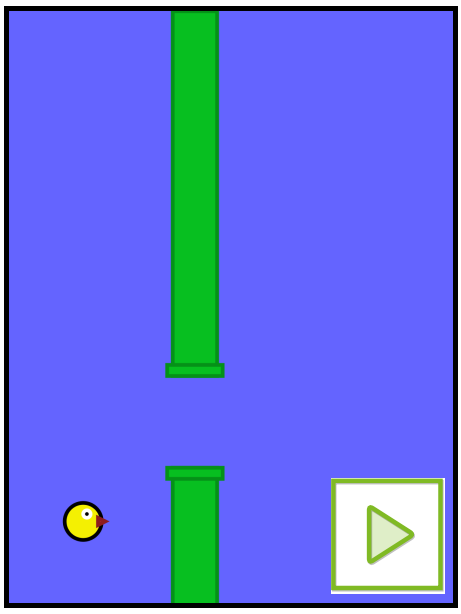
Nun kommt ein anspruchsvoller Teil: Wie kann man zufällig Röhren generieren? Zunächst kümmern wir uns nur um die Darstellung, und noch nicht die Kollision.

Erstellen Sie eine Klasse `class Pipe`, welche die Variablen `x, y` als Position der Röhre speichert (Achtung: jede Röhre besteht aus zwei Röhren - oben und unten!). Zudem benötigen Sie eine (globale) Variable `pipeSpace`, welche definiert, wie viel Abstand zwischen der oberen und unteren Röhre sein soll, und `pipeWidth`, um die Breite von Röhren festzulegen.

Zusätzlich zum Konstruktor (welcher sich für `y` eine Zufallsvariable aussuchen soll), benötigen Sie die Methoden `show()`, welche wie bei `class Bird` auch die beiden Röhren graphisch darstellt (dafür wird `pipeSpace` benötigt). Die Methode `move()` soll die `x`-Position um eine (globale) Variable der Geschwindigkeit verringern (sodass bei jedem Aufruf von `move()` sich die jeweilige Röhre nach links verschiebt).

Um Ihre Klasse auszuprobieren, implementieren Sie eine `myPipe = new Pipe();` in `setup()`, welche dann in `draw()` bei jedem Durchlauf mit `myPipe.move()` und `myPipe.show()` dargestellt werden soll.

Ihr Ergebnis sollte eine Röhre sein, welche von der rechten Seite erscheint, sich langsam dem Vogel annähert, und irgendwann auf der linken Seite wieder verschwindet.



5.3 Röhren in der Endlosschleife

Vielleicht ist Ihnen schon Aufgefallen, dass wir nur eine einzelne Röhre implementiert haben, obwohl wir ja eigentlich unendlich viele haben möchten. Darum kümmern wir uns jetzt!

Mit der Anweisung `pipes = new ArrayList<>()` in `setup()` und `List<Pipe> pipes` als globale Variable können Sie eine sog. *ArrayList* erstellen. Dabei handelt es sich um ein flexibles Array, welches die Arbeit mit „unendlich“ vielen Röhren erleichtern soll. `myPipe` können Sie nun entfernen.

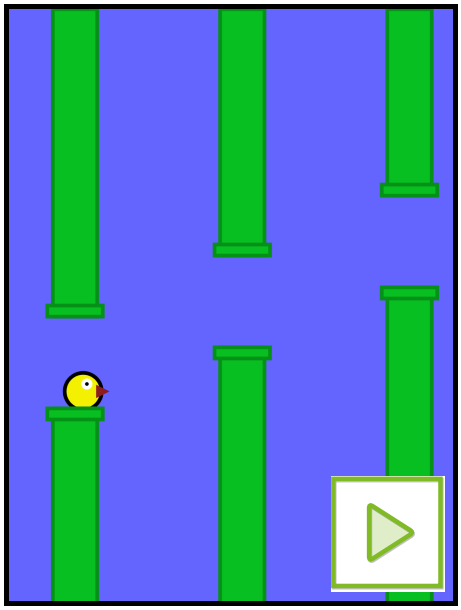
Schreiben Sie eine Methode `generatePipe()`, welche `pipes` eine neue `Pipe` hinzufügen soll. Dafür gibt es bei ArrayLists den Befehl `pipes.add(o)`. Schreiben Sie außerdem eine Methode `loadPipes()`, welche für alle `pipes` die Methoden `move()` und `show()` ausführt. Für Ihre `for`-Schleife können

Sie `pipes.size()` verwenden. Um eine bestimmte Röhre zu bekommen, wird `Pipe p = pipes.get(i)` verwendet.

Prüfen Sie in `loadPipes()` außerdem, ob sich die Röhre nicht mehr im Bildschirm befindet, und entferne sie dann mit `pipes.remove(i--)`.

Zusätzlich sollen Sie über ein festgelegtes Zeitintervall neue Röhren generieren. Erstellen Sie eine Variable `tick`, welche bei jedem `draw()`-Zyklus erhöht wird, und bei einem bestimmten Wert `generatePipe()` aufruft und wieder auf 0 gesetzt wird.

Nun sollten nach und nach Röhren erscheinen, auf zufälligen Höhen und in angemessenem Abstand.



5.4 Kollisionserkennung

Nun kümmern wir uns um die Bestimmung, ob sich der Vogel derzeit in einer Röhre befindet (bzw. mit dieser kollidiert). Dafür erweitern wir `class Bird` um die Methode `deathCheck()`, welche ein `boolean`-Wert zurückgibt.

Die Methode soll über alle vorhandenen Röhren in `pipes` iterieren, und prüfen, ob sich die Instanz von `Bird` in einer der Röhren befindet (dann gibt die Methode `true` aus). Die Bedingungen für `true` sind wie folgt (wobei `p` die derzeit iterierte `Pipe` ist):

- `x` muss größer als `p.x` minus der Hälfte von `pipeWidth` sein, UND...
- `x` muss kleiner als `p.x` plus der Hälfte von `pipeWidth` sein.
- `y` muss kleiner als `p.y` minus der Hälfte von `pipeSpace` sein, ODER...
- `y` muss größer als `p.y` plus der Hälfte von `pipeSpace` sein.

Zudem gibt es eine weitere Möglichkeit, zu sterben:

- `y` ist so groß, dass der Vogel außerhalb des Bildes (auf den „boden“ gefallen) ist.

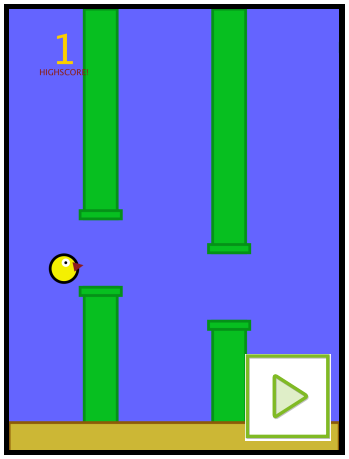
Für den Fall, dass `deathCheck()` wahr ist, können Sie sich einen „Game Over“ Bildschirm überlegen, oder alternativ mit einem Aufruf von `setup()` das Spiel in den Ausgangszustand versetzen (sodass erneut gespielt werden kann). Dafür ist es wichtig, alle initialen Zuweisungen in der `setup()` -Methode zu machen, und nicht außerhalb!

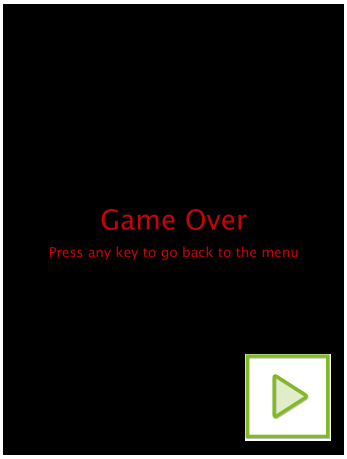
5.5 Erweiterungsideen

Sie haben es geschafft! Nun können Sie das Spiel nach Bedarf noch erweitern. Hier ein paar Anregungen:

- `showFloor()` – wie wär's mit einem Boden?
- `gameOverScreen()` – erzeugt ein Bild, wodurch das Spiel neu gestartet werden kann. Tipp: hierfür ist es sinnvoll, eine Variable `gameOver` einzuführen.
- `showStartMenu()` – ein Startmenü, welches dem Spielenden die Möglichkeit gibt, das Spiel erst zu starten, wenn man bereit ist. Es könnte auch eine Überschrift zeigen.
- `bird.hover()` – für Fortgeschrittene unter Fortgeschrittenen: können Sie eine Methode schreiben, welche dem Vogel den Anschein verleiht, sich leicht nach oben und unten zu bewegen, bevor man das Spiel startet?
- `showScore()` – zeigt die derzeitige Punktzahl an. Sie könnten einen Punkt verleihen, wenn eine Röhre in `loadPipes()` gelöscht wird (da sie dann eindeutig „überlebt“ wurde). Hierfür brauchen Sie selbstverständlich auch eine `score` Variable.
- `highScore` – aufbauend auf `score` könnten Sie eine Höchstpunktzahl einführen, welche nach Spielende anhand von `score` aktualisiert wird.
- Können Sie anhand von `rotate(a)` den Vogel abhängig von dessen derzeitiger `vy` so rotieren lassen, dass er tatsächlich so aussieht, als würde er fliegen?
- `speed` – lassen Sie das Spiel langsam schwerer werden, indem Sie Röhren um `speed` verschieben, und diese Variable immer höher werden lassen.

So könnte das Spiel aussehen, wenn Sie es weiter ausbauen. Natürlich können Sie Ihre eigenen Ideen einfließen lassen, und so das Spiel erweitern.



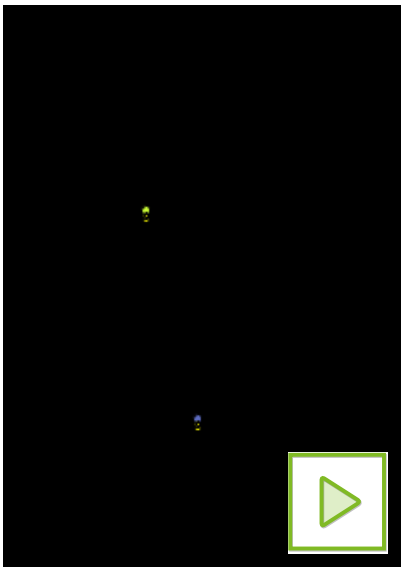


512

168 Z.

Aufgabe 6 – Feuerwerke

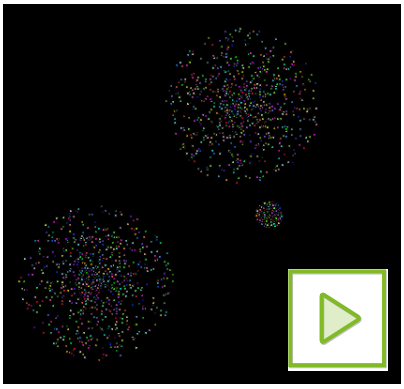
Wir wollen als Abschluss des Brückenkurses ein zufälliges Feuerwerk programmieren, welches wir uns anschauen können.



6.1 Rakete

Erstellen Sie die Klasse `class Rocket`: Eine Rakete braucht **Eigenschaften** wie Geschwindigkeit, Explosionshöhe sowie eine Position auf der x- und y-Achse. Die Startposition auf der x-Achse kann dabei zufällig gewählt werden (eventuell Abstand zum Fensterrand freihalten) und die Startposition auf der y-Achse sollte immer am unteren Fensterrand sein.
Wählen Sie Geschwindigkeit und Explosionshöhe zufällig in sinnvollen Bereichen. Sinnvoll sind auch die folgenden Methoden:

- `void update()`: bewegt die Rakete nach oben bis die Explosionshöhe erreicht wurde und erzeugt dann eine Explosion.
- `void reset()`: setzt Werte der Rakete zurück, damit sie wieder neu losfliegen kann. Achtung: wann soll das passieren? Wie weiß man, dass die Explosion nicht mehr aktuell ist?
- `void show()`: zeichnet die Rakete.



6.2 Explosion

Erstellen Sie die Klasse `class Explosion`: Eine Explosion braucht **Eigenschaften** wie die Anzahl an Partikeln, die erschaffen werden und ein Array für diese. Auch braucht sie einen **Konstruktor**, der die Explosion auf eine Startposition in dem Bild positioniert.

Dabei wird Startposition auf der x-Achse per Parameter übergeben und ist die x-Position der Rakete, wenn diese die Explosionshöhe erreicht. Ebenso ist wird Explosionshöhe per Parameter zur y-Position der Explosion. Auch müssen wir auch alle Partikel an den x- und y-Koordinaten initialisieren.

Außerdem benötigen wir die folgenden Methode:

- `boolean update()`: aktualisiert alle Partikel in unserem Partikel-Array. Achtung: wann hört man auf, Partikel zu aktualisieren - und wie stellt man fest, ob die Explosion fertig ist? Eventuell eignet sich als Rückgabewert dieser Funktion ein `boolean` dafür.

6.3 Partikel

Die Partikel benötigen **Eigenschaften** wie Geschwindigkeit, Flugrichtung und die Position auf der x- und y-Achse, sowie einen **Konstruktor**, der das Partikel auf eine Startposition in dem Bild mit entsprechender Richtung positioniert.

Dabei werden wieder die Startpositionen auf der x und y Achse per Parameter übergeben und Geschwindigkeit und Flugrichtung werden zufällig gewählt. Tipp: Sie können das Orts-Zeit-Gesetz der Gleichförmigen Bewegung mit Schwerkraft verwenden. Eine Umdrehung lässt sich durch eine Multiplikation mit `sin(x)` auf der y-Achse und `cos(x)` auf der y-Achse abbilden. x ist dabei im Bereich 0 bis 2 * PI.

Außerdem benötigen wir die folgende Methode:

- `void update()`: bewegt und zeichnet Partikel auf neue x- / y-Position, welche sich aus Richtung und Geschwindigkeit ergeben. Achtung: Wie erhält man neue Positionen mithilfe der Richtung? Hier könnten Sie auf die Klasse `PVector` oder physikalische Gleichungen zurückgreifen.

6.4 Das Spiel

Initialisieren Sie in `setup()` ein Array mit Raketen, und aktualisieren Sie diese immer wieder in `draw()`.

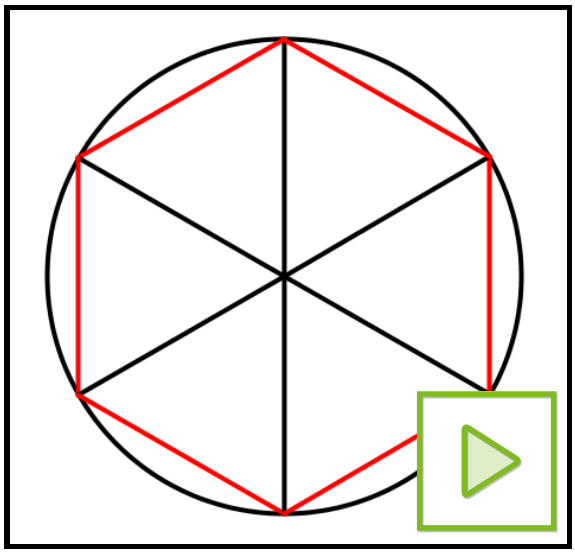
508

63 Z.

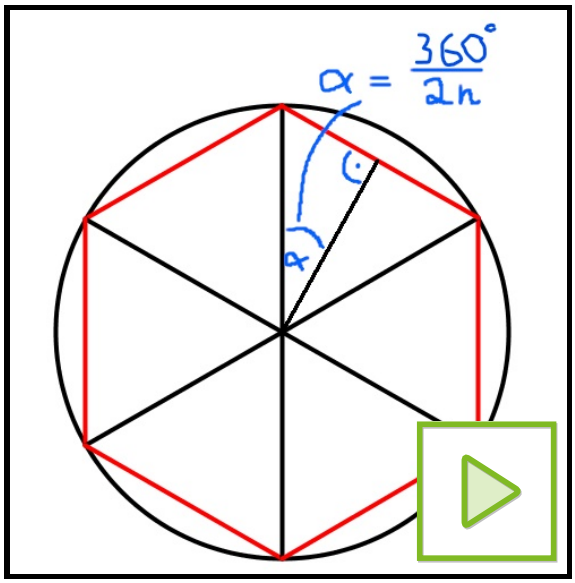
Aufgabe 7 – Pi-Rechner

Haben Sie sich je gefragt, wie man sich den Wert von π („Pi“) herleiten kann? Die kurze Antwort darauf wäre: $\pi = \frac{U}{(2*r)}$ (basierend auf die Formel $U = 2\pi r$) mit dem Umfang U und dem Radius r.

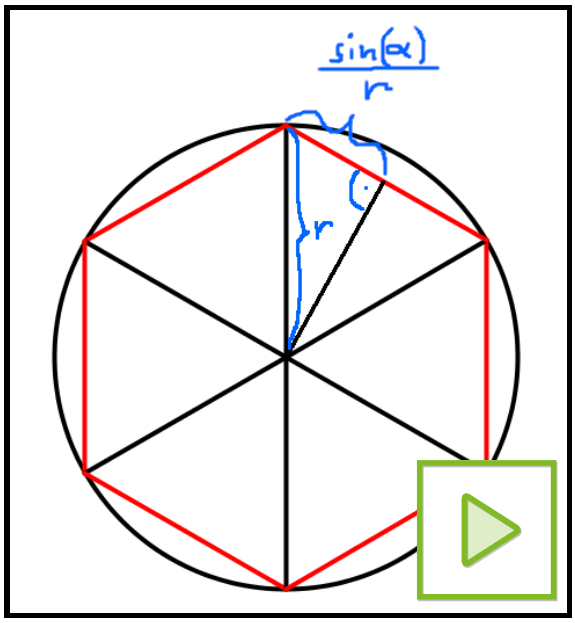
Leider wirft diese Formel ein neues Problem auf: Wir müssen den Umfang eines uns bekannten Kreises (ziemlich genau!) berechnen. Dies sollen Sie in dieser Aufgabe tun.



Unterteilen Sie dazu die Kreisoberfläche in kleine gleichmäßige Abschnitte. Die Anzahl der so entstandenen Dreiecke nennen wir n.



Wenn Sie jetzt eine Grade mittig durch das entstandene Dreieck führen, entsteht ein halb so großes, rechtwinkliges Dreieck.



Mithilfe von Trigonometrie können Sie jetzt die Länge der dem Kreisrand folgenden, roten Ankathete berechnen.

Da Processing Winkel aber in Argumenten für die vorhandenen Trigonometrie-Funktionen nur als Radianten akzeptiert, dürfen Sie `sin(radians(a))` **und** `cos(radians(a))` **verwenden!**

Die Summe aller Abschnitte ergibt dann näherungsweise den Kreisumfang. Setzen Sie dann in $\pi = \frac{U}{(2*r)}$ diesen Umfang und den Radius ein. Tipp: Es bietet sich ein Radius von 1 an! Je kleiner Sie die Abschnitte wählen, desto genauer wird auch das Endergebnis.

503

85 Z.

Aufgabe 8 – Game of Life

Conway's *Game of Life* ist ein Beispiel dafür, wie aus sehr simplen Bildungsregeln sehr komplexe Strukturen entstehen können. Die Simulation findet auf einem Spielfeld mit rechteckigen Zellen statt. Jede Zelle kann entweder lebendig (schwarz) oder tot (weiß) sein. In jeder Generation verändern sich die Zustände dieser Zellen je nach dem Zustand ihrer 8 Nachbarzellen:

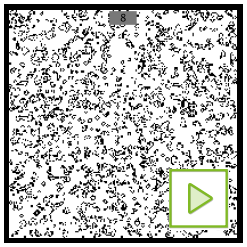
< 2 lebende Nachbarn: Die Zelle stirbt an Vereinsamung.

> 3 lebende Nachbarn: Die Zelle stirbt an Überbevölkerung.

genau 2 lebende Nachbarn: Die Zelle behält ihren alten Zustand.

genau 3 lebende Nachbarn: Die Zelle bleibt lebendig oder wird zum Leben erweckt.

Programmieren Sie Conway's *Game of Life* mit einem zweidimensionalen `boolean[]`-Array in Processing.



Hinweis: *Beginnen Sie mit einem* `boolean[][]` *- Array welches mit zufälligen Werten befüllt wird. Anhand der Regeln wird ein weiteres* `boolean[][]` *- Array, die nächste Generation erstellt und befüllt.*
Hinweis: *Sie können für die einzelnen Aufgaben Funktionen erstellen.*
Bsp. ein Brett zeichnen: `void drawBoard(boolean[][] board),`
ein neues Brett anhand der Regeln erstellen: `void nextGen(mit oder ohne Parameter),`
berechnen der Anzahl an lebendigen Nachbarn einer Zelle:
`int neighbours(boolean[][] currentBoard, int xPos, int yPos)`

505

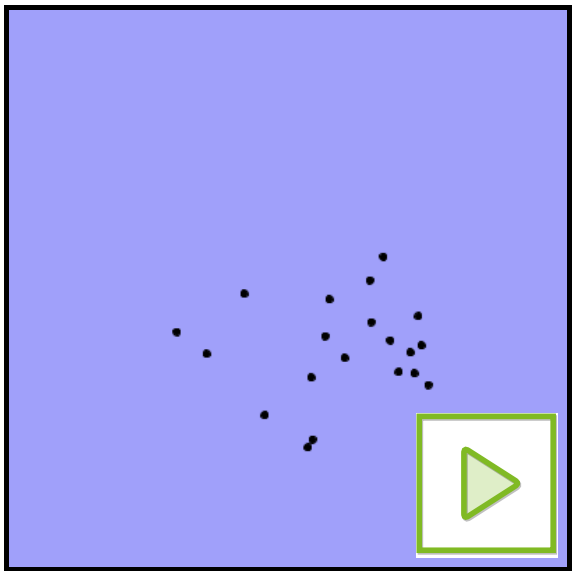
73 Z.

Aufgabe 9 – Schwarmverhalten

Simulieren Sie einen Insektenschwarm, welcher bestimmte Eigenschaften aufweist:

1. Insekten sollen durch Kreise repräsentiert werden.
2. Passiert nichts, soll jedes Insekt eine zufällige Richtung besitzen und sich in diese bewegen. Bei jeder Bewegung soll sich die Richtung leicht verändern (nicht aber völlig zufällig sein), damit die Bewegung natürlich erscheint. Arbeiten Sie mit `constrain(amt, min, max)`, um die Geschwindigkeit zu begrenzen.
3. Insekten dürfen nicht miteinander kollidieren. Verwenden Sie hierzu eine Kollisionserkennung und Positionskorrektur. Dies können Sie mit der Berechnung des Mittelpunkts zwischen den kollidierenden Insekten realisieren, wobei beide Insekten diesen Mittelpunkt als neuen Randpunkt übernehmen sollen.
4. Klickt ein/e Benutzer:in mit der linken Maustaste (`mouseButton == LEFT`), so soll ein neues Insekt an diesem Ort erscheinen.
5. Klickt ein/e Benutzer:in mit der rechten Maustaste, sollen sich alle Insekten auf den Mauszeiger zubewegen.

Erstellen Sie dazu eine Klasse `class Insect` und speichern Sie dessen Instanzen in einem Array oder sogar einer Liste.



507

159 Z.

Aufgabe 10 – Minesweeper

Programmieren Sie das Spiel „Minesweeper“. Dabei geht es darum, auf einem 16 x 16 Felder großen Feld Minen zu finden. in jedem Feld, das keine Mine enthält steht eine Zahl, welche die Anzahl der Minen um dieses Feld herum repräsentiert. Ein Feld ist 50 x 50 Pixel groß, das gesamte Spielfeld 800 x 800 Pixel.

10.1 Spielfeld

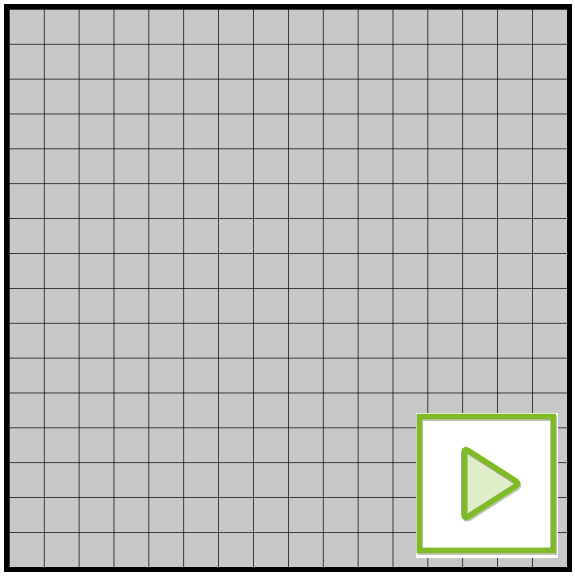
Zunächst einmal muss ein Spielfeld erstellt und gezeichnet werden. Erstellen Sie dazu die Klassen `class Field`, sowie `class Board`, welche die einzelnen Felder bzw. das gesamte Spielfeld darstellen.

Die Klasse `Board` stellt das Spielfeld dar. Legen Sie in diese Klasse eine Variable `Field[][] board = new Board[16][16];` an, welche das Spielfeld beinhaltet. Hierbei handelt es sich um ein 2-Dimensionales Array. Die erste Dimension repräsentiert die x-Position eines Feldes auf dem Spielfeld und die 2. Dimension die y-Koordinate. Ergänzen Sie außerdem einen Konstruktor, in dem für jede Position im Array ein neues `Field` erstellt wird.

Die Klasse `Field` stellt ein einzelnes Feld dar. Erstellen Sie dort die Variablen `int x` und `int y`, welche die x- und y-Koordinate auf dem Spielfeld repräsentieren. Diese sollen im Konstruktor übergeben werden.

Das `Board` muss nun in `void setup()` erstellt werden.

Um das Spielfeld nun zu zeichnen, ergänzen Sie `void draw()` so, dass das Spielfeld gezeichnet wird. Dazu können Sie zum Beispiel alle 50 Pixel sowohl horizontal, als auch vertikal eine Linie zeichnen.



10.2 Die Bomben

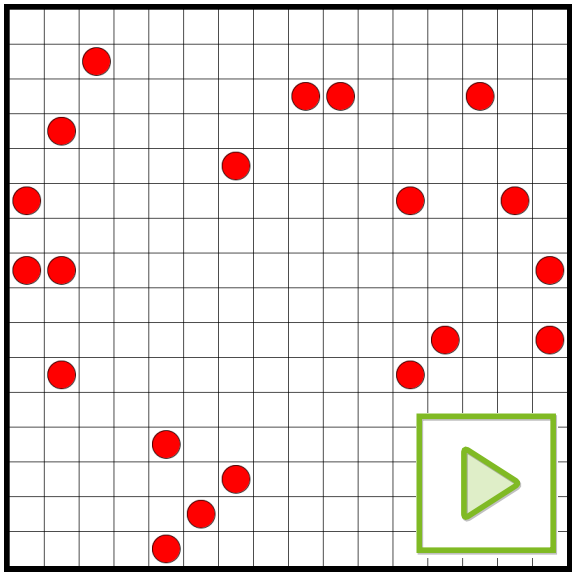
Nun sollen zufällig Bomben auf dem Feld verteilt werden. Dazu erzeugen Sie zunächst eine globale Variable `int bombCount`, welche die Anzahl der Bomben definiert.

Erweitern Sie nun Ihre Klasse `Field` um eine Variable `boolean isBomb`, in der gespeichert wird, ob dieses Feld eine Bombe ist.

Zur Generation der Bomben können Sie nun innerhalb Ihrer Klasse `Board` eine Methode `void generateBombs(int num)` definieren, die innerhalb von `void setup()` aufgerufen wird. Hier soll `isBomb` für so viele Felder, wie in `num` übergeben auf `true` gesetzt werden. Generieren Sie dafür `num` viele zufällige x und y Koordinaten und markieren Sie das entsprechende Feld als `isBomb` Achten Sie darauf, dass keine Bombe an Stellen generiert wird, an denen sich bereits eine Bombe befindet.

Auch sollten die Bomben grafisch anders dargestellt werden, als die üblichen Felder. Ergänzen Sie dafür eine Methode `void show()` in der Klasse `Field`, die, falls ein Feld eine Bombe ist, dieses Feld anders darstellt.

Fügen Sie außerdem Ihrer Klasse `Board` eine Methode `void show()` hinzu, welche die Methode `void show()` der Klasse `Field` für jedes Feld einmal aufruft und in `void draw()` aufgerufen wird.



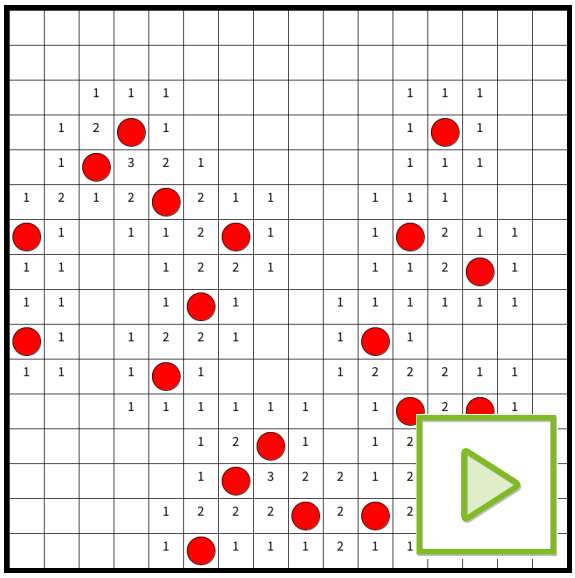
10.3 Die umliegenden Bomben eines Feldes

Der Spielaufbau ist nun schon fast komplett. Es fehlen nur noch die Zahlen auf den Feldern, die angeben, wie viele Bomben an das jeweilige Feld angrenzen. Ergänzen Sie dazu zunächst eine Variable `int surroundingBombs` in der Klasse `Field`

Die Logik dazu soll eine eine neue Methode `int getBombCount()` in der Klasse `Board` liefern. Dort soll für jedes Feld zunächst ein Zähler für die Bomben initialisiert werden, der dann für jede Bombe auf dem umliegenden Feldern hoch zählt. Diese Anzahl soll dann am Ende in das entsprechende `Field` für `surroundingBombs` eingetragen werden. Beachten Sie, dass an den Rändern des Spielfeldes keine Felder kontrolliert werden, die außerhalb des Spielfeldes liegen.

Die Methode `void getBombCount()` soll nun am Anfang in `void setup()` aufgerufen werden.

Danach muss noch `void show()` in der Klasse `Field` angepasst werden. Hier soll die Anzahl umliegender Bomben als schwarzer Text angezeigt werden, wenn das Feld keine Bombe ist und mindestens ein umliegendes Feld eine Bombe ist. Ist das nicht der Fall, soll das Feld einfach leer bleiben.



10.4 Das Aufdecken der Felder

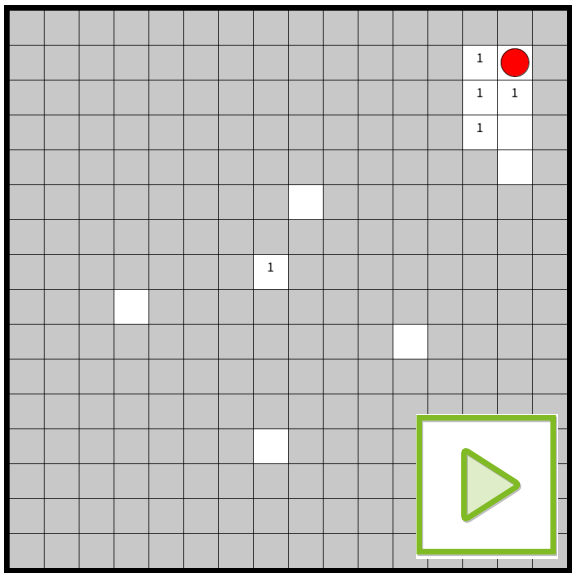
In diesem Teil der Aufgabe geht es darum die Logik zum Aufdecken der Felder zu definieren. Bislang sind alle Felder zu Beginn der Runde aufgedeckt. Das soll sich nun ändern!

Zum Zeigen des Felds legen wir nun eine Methode `void reveal()` und eine neue Variable `boolean isRevealed` in der Klasse `Field` an. Wenn `void reveal()` aufgerufen wird, soll `isRevealed` auf `true` gesetzt werden, sofern sie noch nicht `true` ist.

Anschließend muss noch `void show()` in der Klasse `Field` angepasst werden. Ist ein Feld noch nicht aufgedeckt, darf noch nicht angezeigt werden, was sich auf dem Feld befindet.

Jetzt fehlt nur noch die Logik für das Anklicken der Felder. Verwenden Sie dafür die Funktion `void mousePressed()`. Bei einem Linksklick soll für das Feld an dieser Position `void reveal()` aufgerufen werden.

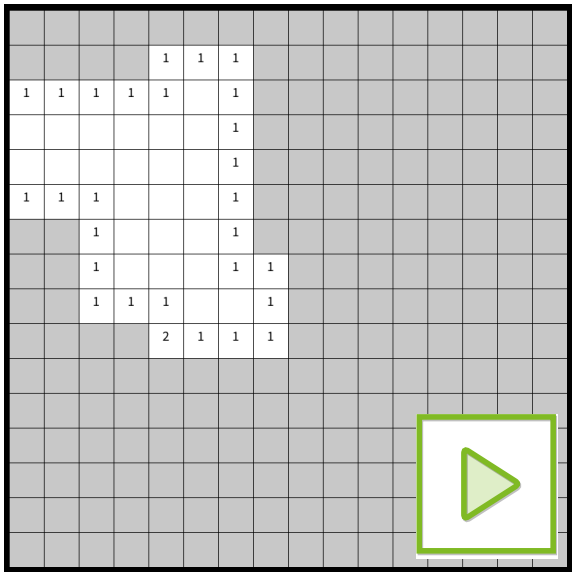
Jetzt können Sie schon Felder mit der Maus aufdecken!



10.5 Das Aufdecken von Feldern ohne angrenzende Bomben

In Minesweeper werden die umliegenden Felder von Feldern ohne angrenzende Bomben automatisch aufgedeckt.

Erweitern Sie hierzu die Methode `void reveal()` so, dass sie `void reveal()` für alle umliegenden Felder aufruft, falls das betreffende Feld ein Feld ohne angrenzende Bomben ist. Achten Sie darauf, dass `void reveal()` nicht mehr für ein Feld aufgerufen wird, das bereits aufgedeckt ist, sonst wird `void reveal()` unendlich oft aufgerufen.



10.6 Spielende

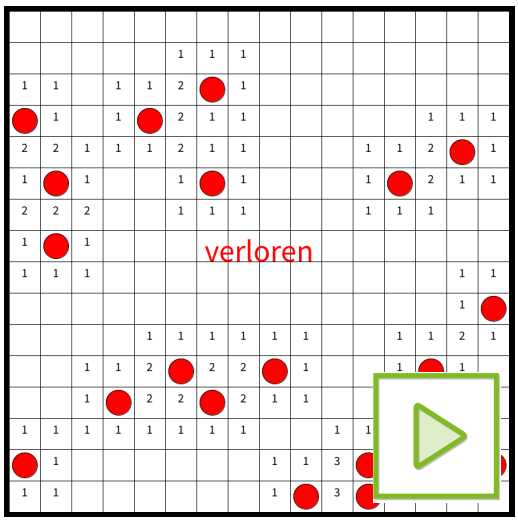
Nun fehlt nur noch eine Benachrichtigung, ob das Spiel vorbei ist und ob es gewonnen wurde. Dazu legen Sie in der Klasse `Board` zwei neue Variablen `boolean gameOver` und `boolean isWon` an, in welchen gespeichert wird, ob Game Over ist und ob das Spiel gewonnen ist.

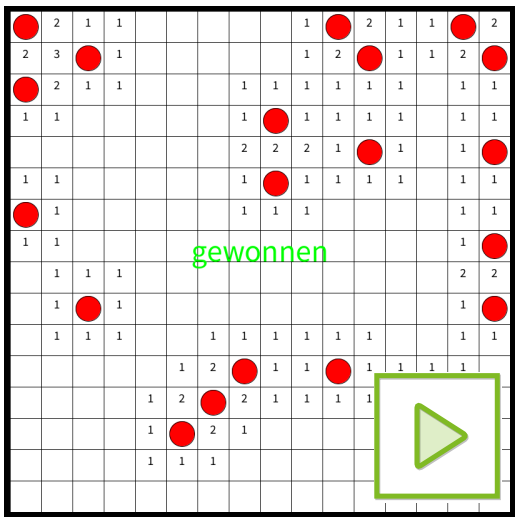
Das Spiel ist vorbei, wenn entweder eine Bombe angeklickt wurde, oder alle Felder, die keine Bombe enthalten aufgedeckt wurden. Für den Fall, dass das Spiel verloren ist und eine Bombe angeklickt wurde, ergänzen Sie die Methode `void reveal()` so, dass `gameOver` auf `true` und `isWon` auf `false` gesetzt wird, wenn das angeklickte Feld eine Bombe ist.

Für den Fall, dass alle Felder ohne Bombe aufgedeckt sind, ergänzen Sie die Klasse `Board` um eine Methode `void isGameOver()`, welche alle Felder durchgeht und `gameOver`, sowie `isWon` auf `true` setzt, wenn kein Feld, welches keine Bombe ist mehr unaufgedeckt ist. Sollte `gameOver` schon `true` sein, hat die Methode nichts zu tun.

Nun muss noch angezeigt werden, ob Game Over ist oder nicht. Modifizieren Sie die Methode `void show()` der Klasse `Board` so, dass zunächst am Anfang `void isGameOver()` aufgerufen wird, um zu aktualisieren, ob das Spiel vorbei ist, und, falls Game Over ist, danach eine Nachricht anzeigt, ob das Spiel gewonnen oder verloren ist. Auch sollte nach einem Game Over das gesamte Feld inklusive der Bomben aufgedeckt sein.

Außerdem können Sie Ihre Methode `void mouseReleased()` so anpassen, dass nur Felder aufgedeckt werden, wenn noch nicht Game Over ist.



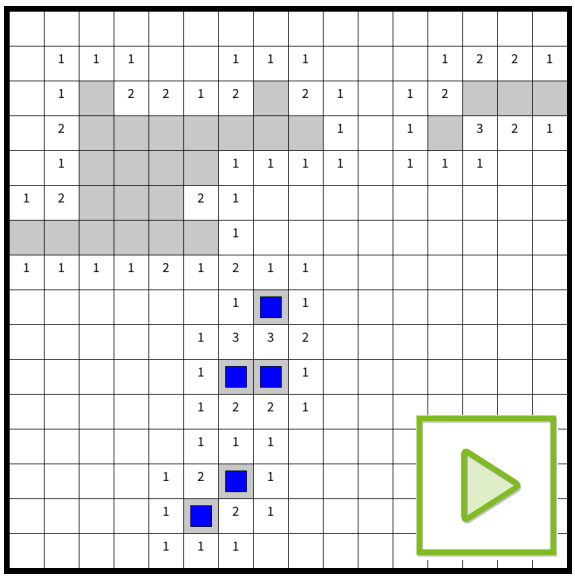


10.7 Bomben markieren

Cool, das eigentliche Spiel ist jetzt vollständig! Es wäre allerdings noch schön, Bomben markieren zu können.

Dazu fügen Sie Ihrer Klasse `Field` eine Variable `boolean isMarked` hinzu, in der Sie speichern können, ob ein Feld markiert ist, oder nicht. Diese Variable soll bei einem Rechtsklick auf ein Feld auf `true` gesetzt werden, sofern das entsprechende Feld noch nicht aufgedeckt ist, und das Spiel noch nicht vorbei ist. Sorgen Sie außerdem dafür, dass ein Feld nur aufgedeckt werden kann, wenn es nicht markiert ist.

Jetzt müssen die Markierungen nur noch angezeigt werden. Ergänzen Sie die Methode `void show()` der Klasse `Field`, sodass das Feld anders aussieht, wenn es markiert ist.



509

153 Z.

Aufgabe 11 – Doodle Jump

Mit dieser Aufgabe werden Sie ihre eigene Version von Doodle Jump erstellen können.

11.1 Plattformen

Damit unser Hüpfemännchen nicht (sofort) ins Nirwana plumpst, brauchen wir einige Plattformen, die sich von oben nach unten bewegen. Und das immer wieder...

Erstellen Sie die Klasse `class Platform`:

Eine Plattform braucht **Eigenschaften** wie Breite und Höhe sowie seine Position auf der x- und y-Achse. Auch braucht sie einen **Konstruktor**, der die Plattform auf eine Startposition in dem Bild positioniert. Die Startposition auf der x-Achse kann zufällig gewählt werden (eventuell Abstand zum Fensterrand freihalten) die Startposition auf der y-Achse legen wir über einen Parameter, der dem Konstruktor übergeben wird, fest.

Außerdem benötigen wir die folgenden Methoden:

- `void move()`: bewegt die Plattform nach unten.
  Achtung: Was soll mit der Plattform passieren, wenn sie einmal durch das Bild gelaufen ist?
- `void resetPosition()`: macht eine durchgelaufene Plattform wiederverwendbar:
  Setzen Sie eine neue zufällige x-Position und schicken Sie die Plattform zurück an den oberen Fensterrand!
- `void show()`: zeichnet die Plattform.

11.2 Spielfigur

Erstellen Sie die Klasse `class Player`:

Die Spielfigur braucht **Eigenschaften** wie Position auf der x- und y-Achse, und außerdem noch eine Variable für die Fallgeschwindigkeit. Auch braucht sie einen **Konstruktor**, der die Spielfigur auf eine Startposition in dem Bild positioniert

Außerdem benötigen wir die folgenden Methoden:

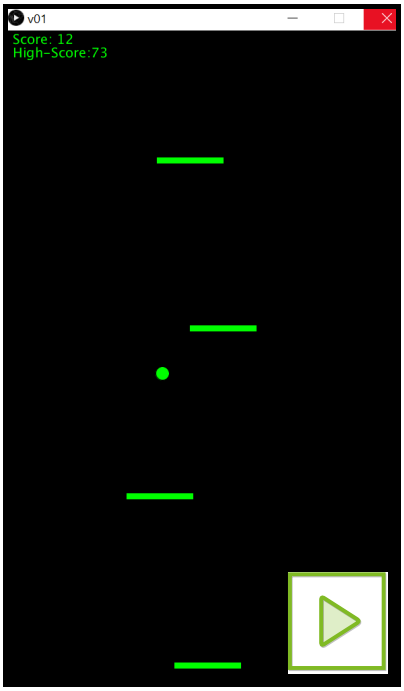
- `void move()`: bewegt die Spielfigur anhand von Tastenanschlägen nach links und rechts.
  Achtung: Was soll passieren, wenn ein Bildrand erreicht ist?
- `void jump()`: berechnet die neue y Position der Spielfigur anhand der Geschwindigkeit und der aktuellen Position.
  Achtung: Wie muss die Geschwindigkeit am Anfang sein und wie muss sie sich verändern, um die Spielfigur durchgehend hüpfen zu lassen?
- `void bounce()`: setzt die Geschwindigkeit zurück die Startwerte.
- `void show()`: zeichnet die Spielfigur.

11.3 Das Spiel

Das Spiel braucht **Eigenschaften** wie Score und Highscore, Anzahl der Plattformen und ein Array mit entsprechend vielen Plattformen. Dazu kommt auch eine Spielfigur.

Außerdem benötigen wir die folgenden Methoden:

- `void initPlatforms()`: erzeugt für jede Stelle im Array eine neue Plattform.
  Achtung: welche Y-Position bekommen die Plattformen jeweils? Wie berechnet man den Abstand im Bezug auf die Fenstergröße?
- `void gameOver()`: berührt die Spielfigur den unteren Spielfeldrand? Wurde ein neuer Highscore erreicht?
- `void playerCollides()`: prüft, ob die Spielfigur eine Plattform berührt.
  Achtung: was muss passieren wenn dies der Fall ist?
- `void drawScore()`: zeige den Score/Highscore an.
- `void updatePlatforms()`: bewegt und zeichnet alle Plattformen.
  Achtung: Welche Funktionen muss jede Plattform ausführen?
- `void updatePlayer()`: bewegt und zeichnet die Spielfigur.
- `void setup()`: initialisiert die Plattformen über (initPlatforms()) und die Spielfigur.
- `void draw()`: rufe die anderen Spiel-Funktionen auf und los geht's!

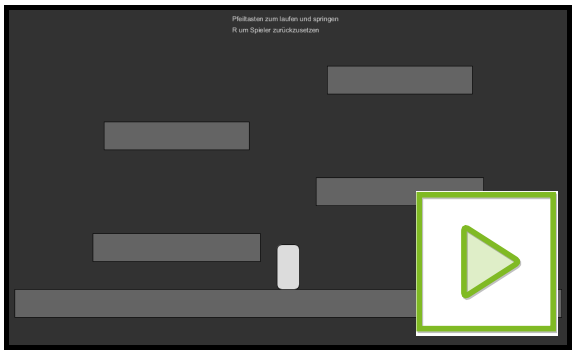


510

205 Z.

Aufgabe 12 – Jump'n'Run

In dieser Aufgabe soll ein einfaches Jump'n'Run Spiel erstellt werden. Dazu ist es notwendig, einen Kollisionschecks inklusive Positionskorrektur und simpler Physik zu implementieren. Das Ergebnis könnte in etwa so aussehen.



Bei Jump'n'Run Spielen ist eine Herausforderung, die Mechanik des Spielercharakters glaubwürdig zu programmieren. Kollisionen mit Objekten der Umgebung sollten exakt sein und der Spieler sollte sich wenigstens annäherungsweise nach den physikalischen Gesetzen wie Momentum und Schwerkraft bewegen.

Zur Vereinfachung werden alle Objekte durch Rechtecke dargestellt. Dies erscheint erstmal etwas simpel, genügt aber in vielen Anwendungsfällen vollkommen. Auch in der Videospielindustrie wird es oft nicht anders gehandhabt.

Wenn es gewünscht ist, können die einfachen Rechtecke auch durch Grafiken für Spieler und Hindernisse eingesetzt werden.

12.1 Physik

Um die Physik des Spielercharakters zu simulieren, werden wir die sogenannte "Verlet Integration"verwenden. Stark vereinfacht geht es dabei darum, das Momentum des Spielcharakters durch dessen Positionsveränderung zu ermitteln. Die Geschwindigkeit eines Objekts ergibt sich somit aus der Differenz der jetzigen Position zur vorherigen Position.

12.2 Kollisionskontrolle

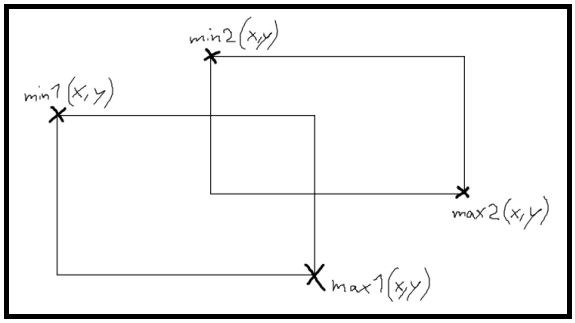
Zusätzlich dazu muss einen Kollisionscheck durchgeführt und gegebenenfalls eine Positionskorrektur erfolgen. Zur Umsetzung des Kollisionschecks wird eine Methode eingesetzt, die prüft, ob sich zwei betrachtete Rechtecke überlappen. Dazu werden die folgenden Koordinaten beider Rechtecke benötigt:

1. linke obere Ecke
2. rechte untere Ecke

Diese beiden Koordinaten werden auch als *Minimum* und *Maximum* eines Rechtecks bezeichnet.

Um nun herauszufinden, ob sich die beiden Rechtecke überlappen, muss man die Minima und Maxima der Rechtecke vergleichen. Dabei ist es entscheidend, ob der maximale x-Wert eines der Rechtecke kleiner als der minimale x-Wert des anderen ist, oder ob selbiges für die y-Werte zutrifft.

Dies kann man sich wie folgt vorstellen:



Und als Code sähe das etwa so aus:

```
boolean rectOverlap(PVector min1, PVector max1, PVector min2, PVector max2)
    {
   if (max1.x < min2.x || max1.y < min2.y || max2.x < min1.x || max2.y <
       min1.y) {
      return false;
   } else {
      return true;
   }
}
```

Hinweis: PVector ist eine vordefinierte Klasse in Processing. Sie bietet einige Methode zur Vektorenrechnung, allerdings werden wir sie in diesem Tutorial nur zur Speicherung von Vektoren verwenden, also für Objekte mit x- und y-Wert.

Hinweis: Unter Processing ist das Minimum eines Rechtecks schlicht die Ausgangskoordinate des Rechtecks, also dessen x- und y-Koordinaten. Das Maximum ergibt sich aus der Summe der x-Koordinate und der Breite (w) des Rechtecks, sowie aus der Summe der y-Koordinate und der Höhe (h) des Rechtecks.

Der folgende Code dient als Denkanstoß für eine Klasse zur Erstellung eines Rechtecks:

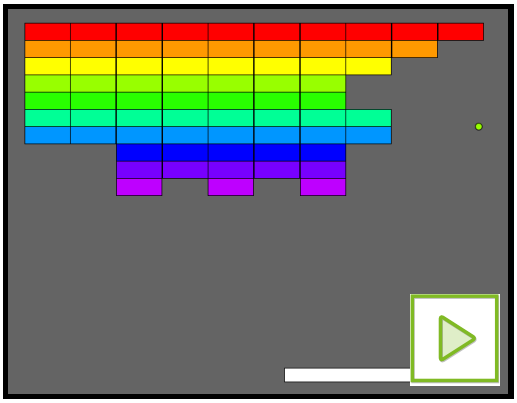
```
class Rect {
   float x;
   float y;
   float w;
   float h;
   Rect(float x, float y, float w, float h) {
      this.x = x;
      this.y = y;
      this.w = w;
      this.h = h;
   }
   PVector getMin() {
      return new PVector(x, y);
   }
   PVector getMax() {
      return new PVector(x+w, y+h);
   }
   void drawThis() {
      rect(x, y, w, h);
   }
}
```

515

151

Aufgabe 13 – Breakout

Programmieren Sie das Spiel *Breakout*. Bei diesem Spiel können Spieler:innen einen Schläger am unteren Spielfeldrand nach links und nach rechts bewegen, um den Ball im Spielfeld zu halten. Am oberen Spielfeldrand befinden sich eine Reihe von Rechtecken, die zerstört werden müssen. Zu Beginn des Spiels wird ein kleiner Ball schräg von dem Schläger nach oben abgeschossen. Wird ein Rechteck getroffen, so wird es zerstört und der Ball prallt ab (ähnlich wie bei Pong). Je nach dem, an welcher Stelle der Ball auf den Schläger trifft, prallt er steiler oder flacher ab.



13.1 Bonusaufgaben

Wenn man so ein schönes Spiel erstellt hat, kann man daran natürlich auch weiterarbeiten. Hier ein paar Ideen um Breakout zu erweitern.

- Die Rechtecke können unterschiedliche Farben haben.
- Die Rechtecke können unterschiedlich viele Treffer benötigen, um zerstört zu werden.
- Die Rechtecke können sich bewegen.
- Die Rechtecke können "Powerups"geben, mit der der Schläger größer oder kleiner wird.
- Die Rechtecke können "Powerups"geben, mit der der Ball schneller oder langsamer wird.
- Es kann verschiedene Arten von Bällen geben, die sich unterschiedlich verhalten.
- Die Erstellung des Spielfelds kann in verschiedenen Leveln unterschiedlich sein.
- Abspeichern der Highscores.

516

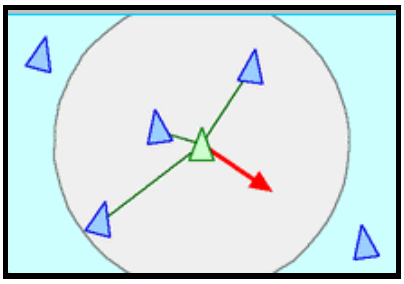
149
Z.

Aufgabe 14 – Schwarmverhalten II

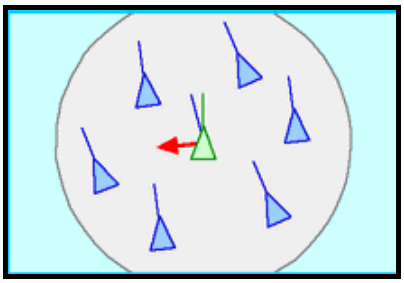
Hinweis: *Diese Aufgabe ist auch über Arrays lösbar, aber es ist einfacher, mit einer Liste flexibler Länge zu arbeiten. Einer ArrayList können Sie über die Funktion add() Werte hinzufügen und über get() auslesen. Nutzen Sie auf jeden Fall die Referenz*

Das Schwarmverhalten von Vögeln oder Fischen lässt sich mit erstaunlich einfachen Regeln simulieren.
Craig Reynolds entwickelte bahnbrechende Simulationsprogramme, unter anderem solche, die Vogelschwärme simulieren. Er prägte den Begriff **boid**, angelehnt an das englische Wort „bird“ für die einzelnen Einheiten des Schwarms.
Die folgenden Regeln und Darstellungen (entnommen http://www.red3d.com/cwr/boids/ einer Seite von Craig Reynolds selbst) erläutern die 3 von ihm beschriebenen Regeln, welche zum Simulieren eines Tierschwarms eingesetzt werden können.

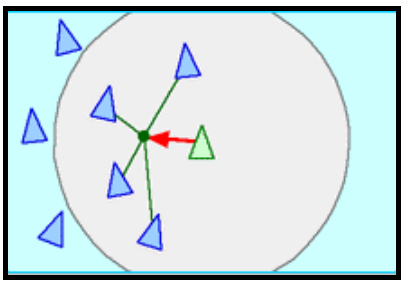
1. Separation: halte **Abstand** zu deinen Nachbarn.



2. Ausrichtung: bilde einen Durchschnittswert aus den **Richtungen**, in die sich deine Nachbarn bewegen und nutze ihn für deine eigene Bewegung.



3. Zusammenbleiben: bilde einen Durchschnittswert aus den **Positionen** deiner Nachbarn. Bewege dich in Richtung dieser errechneten Position.



14.1 Boids

Formulieren Sie zuerst eine Klasse **Boid**.
Ein Boid hat:

- Eine Position, bestehend aus x und y-Wert
- Einen Vektor $\vec{e_1} = \begin{pmatrix} x \\ y \end{pmatrix}$, welcher die Richtung und Länge der letzten Bewegung angibt.

Hinweis: *Überlegen Sie sich, welche Datenstruktur Sie hier nutzen wollen! PVector ist eine Option, aber es gibt natürlich viele Wege zum Ziel!*

14.2 Funktionen

Für die Simulation müssen folgende Funktionen erstellt werden:

- Addieren von Vektoren
- Subtrahieren von Vektoren
- Einen Vektor mit einem Skalar multiplizieren
- die Distanz zwischen zwei durch einen Vektor dargestellten Punkten ermitteln. Hinweis: *Es gibt eine bereits in Prozessing vorhandene Funktion, die hierbei hilfreich sein kann. Haben Sie eine Idee, welche?*
- den Betrag eines Vektors ermitteln Hinweis: *dabei können Sie die zur Distanzermittlung geschriebene Funktion, verwenden, wenn Sie wollen. Bei Bedarf ist auch der Satz des Pythagoras ihr Freund.*
- Boids zufällig im Fenster platzieren
- Anzeigen eines Boids Hinweis: *Sie können Boid als Dreiecke, oder noch einfacher, als Kreise mit jeweils einem der Ausrichtung entsprechenden Strich darstellen.*
- Überprüfen, bei welchen Boids es sich um Nachbarn des aktuellen Boids handelt
- Errechnen eines Vektors, welcher die Regel zur Separation umsetzt
- Errechnen eines Vektors, welcher die Regel zur Ausrichtung umsetzt
- Errechnen eines Vektors, welcher die Regel zum Zusammenbleiben umsetzt
- Kombinieren aller drei Vektoren zu einer endgültigen Bewegung
- (optional) Boids weg von Fensterrand lenken, falls sie diesem zu Nahe kommen
- Anwenden der Bewegung auf einen Boid
- Überprüfen auf aus dem Bildschirm gewanderte Boids und Neuspawnen dieser Boids

Dazu hier noch einige Informationen:

14.2.1 Separation

Um die Separation umzusetzen, ermitteln Sie die Distanz als Vektor $\vec{d_1} = \begin{pmatrix} x \\ y \end{pmatrix}$ zwischen jedem Nachbarn und dem aktuellen Boid.
Nahen Nachbarn muss stärker ausgewichen werden als weiter entfernten. Daher nutzen Sie einen

Vektor, der die Bewegung vom Nachbarn zum Rande ihres Wahrnehmungsbereiches beschreibt. Auf den aktuellen Boid angewandt, würde es sich dabei um eine Bewegung weg von dem Nachbarn halten, die größer ist, je näher der Nachbar ist.
Da ein Boid mehrere Nachbarn haben kann, müssen Sie die einzelnen Bewegungen aufaddieren und den Durchschnitt ermitteln.

14.2.2 Ausrichtung

Die durchschnittliche Ausrichtung der Nachbarn zu Errechnen ist, durch einfaches Summieren jeweiligen Ausrichtungen und Teilen durch die Anzahl der Nachbarn möglich.

14.2.3 Zusammenbleiben

Errechnen Sie einen Mittelpunkt der Nachbarn, indem Sie den Durchschnittswert ihrer Position errechnen. Das Bewegen in Richtung dieses Mittelpunktes sorgt für Einhaltung der Regel.

14.3 Endgültige Bewegung

Sie können die drei Vektoren gewichtet kombinieren, d.h. bestimmen, wie stark die jeweilige Regel in das Ergebnis eingeht. Hierfür sind Skalarmultiplikationen hilfreich.
Wenn sich die Gewichte zu 1 oder einer kleineren Zahl aufaddieren, passiert es leicht, dass die Bewegungsvektoren in einigen Runden den Wert 0 annehmen. verwenden Sie daher Gewichte, die sich zu mehr als 1 aufsummieren und führen Sie ein Geschwindigkeitsmaximum ein. Passen Sie auf, dass wenn Sie Werte reduzieren, um diesem Maximum zu genügen, nicht die Richtung ihrer Bewegung verzerren.
Kombinieren Sie die neu errechnete mit der alten Bewegung, damit das Verhalten ihrer Boids flüssig wirkt. Ein Vogel muss die Richtung, in die er fliegt, auch allmählich wechseln.

14.4 Boids im Bildschirm halten

Da die Boids sich gegenseitig beeinflussen, ist es nicht erfolgversprechend, die Richtung eines Boids, der sich dem Fensterrand nähert, durch das Multiplizieren mit -1 umzukehren. Stattdessen empfiehlt es sich durch das Addieren oder Subtrahieren kleiner Werte, den Boid in die erwünschte Richtung zu „Stupsen“.

14.5 Weitere Überlegungen

Es kann hilfreich sein, sich beim Experimentieren den Bereich, in dem ein Boid andere Boids beachtet, als Kreis einzuzeichnen.
Auch wenn es keinem Vogelschwarm entspricht: wenn Sie die Gewichte auf 1 setzen, und die Boids somit zum Stehen kommen, entstehen bei eingezeichneten Sichtbereichen geometrische, blumenartige Figuren. Experimentieren Sie hiermit!
Spielen Sie mit den von Ihnen verwendeten Werten, besonders den Gewichten. Wie verändert sich das Verhalten Ihres Schwarms?
Wenn Sie das Maximum für unterschiedliche Achsen unterschiedlich festlegen, bewegt ihr Schwarm sich bevorzugt in bestimmte Richtungen. Lassen Sie ihren Schwarm über den Bildschirm sausen und spawnen Sie neue Vögel am entgegengesetzten Bildschirmrand!
Empfinden Sie die Form des Bereichs, in dem andere Boids als Nachbarn wahrgenommen werden, dem Sichtfeld eines Vogels nach!