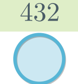
Aufgabenblatt 4 - Objektorientierung

432



17 Z.

Aufgabe 1 – Zähler

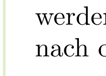
Schreiben Sie eine Klasse `class Counter { ... }`, wessen Konstruktor `Counter(int start){ ... }` ein Wert `start` entgegennimmt, welcher unter der Instanzvariable `int next;` gespeichert werden soll. Wird `void count(){ ... }` bei einer Instanz von `Counter` aufgerufen, soll in der Konsole der aktuelle Wert von `next` ausgegeben werden und `next` um 1 erhöht werden.

Hier ist eine Vorlage der Klasse. Tippen Sie den Code ab und ersetzen Sie die Kommentare in `Counter` mit Code.

```
1  void setup() {
2     Counter c = new Counter(40);
3     c.count(); // Konsole: 40
4     c.count(); // Konsole: 41
5     println("Variable next speichert: " + c.next); // Konsole: ... 42
6     c.count(); // Konsole: 42
7  }
8  class Counter {
9     // next-Variable
10    Counter(int start) {
11       this.next = start;
12    }
13    void count() {
14       // Ausgabe next
15       // Erhöhung next um 1
16    }
17 }
```

434

27 Z.



Aufgabe 2 – 2D-Bewegung

Schreiben Sie ein Programm, in welchem ein Kreis auf einem Raster mit den Pfeiltasten bewegt werden kann. Wird der Pfeil nach oben gedrückt, soll sich der Kreis also um eine festgelegte Distanz nach oben bewegen, und so weiter.

Hier ist eine Vorlage des Programms. Tippen Sie den Code ab und ersetzen Sie die Kommentare.

```
1  // Variablen x und y deklarieren
2  void setup() {
3     // Größe des Zeichenfelds festlegen
4  }
5  void draw() {
6     // Hintergrund zeichnen
7     // Kreis an Stelle (x, y) zeichnen
8  }
9  void keyPressed() {
10    if (key == CODED) {
11       switch(keyCode) {
12          case UP:
13             y -= 20;
14             break;
15          case RIGHT:
16             // ?
17             break;
18          // zwei weitere Cases
19       }
20    }
21 }
```

433

18 Z.

Aufgabe 3 – Dark-Theme

Schreiben Sie ein Programm, dass einen schwarzen Text auf hellem Hintergrund per Mausklick zu einem weißen Text auf dunklem Hintergrund hin-und-her wechselt. Benutzen Sie eine globale Variable `bool isDarkTheme = false;`, welche speichern soll, ob der dunkle Modus derzeit aktiv ist.

Hier ist ein Programm, welches einen Lichtschalter in der Konsole simuliert:

```
boolean schalterStatus = false;

void draw() { }

void mouseClicked() {
   if ( schalterStatus == true ) {
      schalterStatus = false;
      println("Ausgeschaltet!");
   } else {
      schalterStatus = true;
      println("Eingeschaltet!");
   }
}
```

435

23 Z.

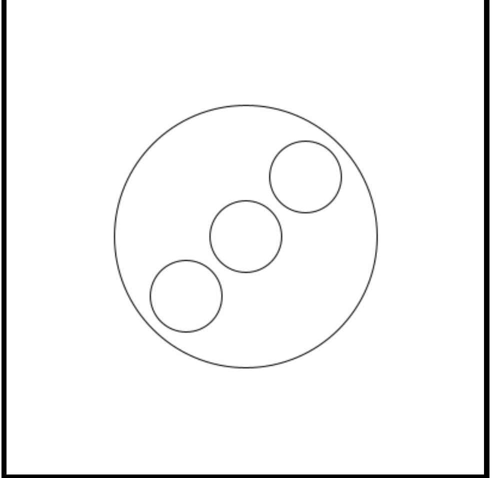
Aufgabe 4 – Objektorientierter Kreis

Schreiben Sie eine Klasse `class Circle { ... }`, welche alle Informationen speichern soll, die zum Zeichnen eines Kreises benötigt werden soll. Der Konstruktor `Circle(int x, int y, int d){ ... }` soll die empfangenen Werte in der Instanz abspeichern. Mit `void show(){ ... }` soll der Kreis mit der gespeicherten Position und Größe gezeichnet werden.

Tippen Sie diesen Code ab und schreiben Sie den Inhalt der Klasse `Circle`. Ihr Ergebnis sollte bei Ausführung das gezeigte Bild zeichnen.

```
void setup() {
   size(400, 400);
   background(255);
   Circle c1 = new Circle(200, 200,
        220);
   c1.show();
   c1.d = 60;
   c1.show();
   c1.x += 50;
   c1.y -= 50;
   c1.show();
   new Circle(150, 250, 60).show();
}

class Circle {
   int x, y, d;
   // ...
}
```



401

22 Z.
32 Z.
54 Z.
50 Z.

Aufgabe 5 – Einstiegsaufgabe: Konfettikanone

Wir wollen fürs jährliche Karneval eine Konfettikanone entwickeln.
Dabei soll per Knopfdruck am derzeitigen Ort der Maus ein Haufen Konfetti sprühen.

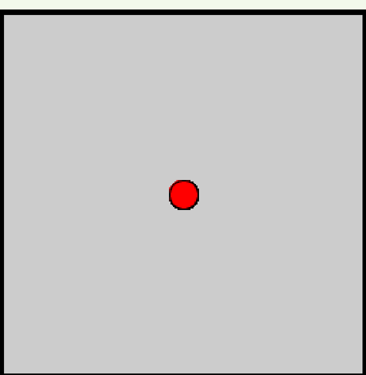
5.1 Konfettipartikel

Zunächst wollen wir ein Konfettipartikel programmieren. Da wir schon jetzt wissen, dass später hunderte generiert werden sollen, ist hier eine Klasse sinnvoll.

Schreiben Sie die Klasse `Particle`, in welcher die Instanzvariablen `float x, y` und `color c`, sowie die Geschwindigkeiten `float vx, vy` gespeichert werden.

Mit dem Konstruktor `Particle( ... )` soll man die `x` und `y`-Variablen, sowie die Farbe `c` festlegen können. Lassen Sie die Geschwindigkeitsvariablen zunächst unberührt.

Schreiben Sie zudem eine Methode `void show()`, welche dieses Partikel an der derzeitigen Position zeichnet. Platzieren Sie dann einen roten Partikel in die Bildmitte.



5.2 Der erste Flug

Was wäre ein Konfettipartikel ohne Bewegung?

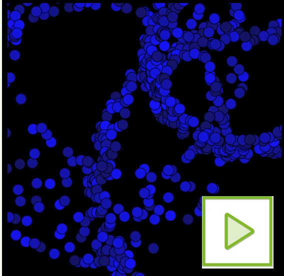
Legen Sie im Konstruktor eine zufällig gewählte horizontale Geschwindigkeit fest, und setzen Sie `vy` zu einem kleinen, positiven Wert (damit die Partikel erst leicht hochspringen, bevor sie nach unten fallen).

Schreiben Sie eine Methode `void move()`, welche auf `x` und `y` die entsprechenden Geschwindigkeiten anwendet, und dann auf `vy` eine Schwerkraft wirken lässt.

Auf der nächsten Seite geht diese Aufgabe weiter...

5.3 Gib mir mehr!

Ein Partikel ist ja schön und gut... Wir wollen aber mehr!
Schreiben Sie zunächst für die Klasse `Particle` eine weitere Methode `boolean isOutside()`, welche `true` ausgibt, wenn das Partikel außerhalb des Bildschirms ist.
Legen Sie dann eine Variable `int count` fest (die Anzahl der Partikel), und mithilfe dessen eine Variable `Particle[] particles` fest. Schreiben Sie in `draw()` eine for-each Schleife, die alle Partikel zeichnen und bewegen soll. Überprüfen Sie davor, ob `p == null` gilt - und machen Sie in diesem Fall nichts (da ein leeres Partikel nicht gezeichnet werden kann).
Schreiben Sie dann eine Funktion `void loadNewParticles(int n)`, welches `n` neue Partikel generieren soll. Verwenden Sie dazu eine Schleife, welche Elemente des `particles`-Array durchläuft und überprüft, ob das vorliegende Partikel entweder außerhalb der Bildfläche ist, oder noch nicht existiert. Brechen Sie die Schleife vorzeitig ab, wenn Sie `n` aktualisierte Partikel erreicht haben.
Überprüfen Sie in `draw()` nun, ob derzeit `mousePressed` `true` ist. In diesem Fall soll `loadNewParticles()` mit einem von Ihnen gewählten Wert aufgerufen werden.



5.4 Wellen

Bauen Sie Ihr Programm so um, dass die Bälle nun keine Schwerkraft mehr haben, sondern stattdessen konstante Faktoren für die Bewegung in die `x`- und `y`-Richtungen.

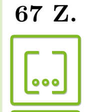
Um dies zu erleichtern, wollen wir nun mit der Klasse `PVector` arbeiten. Dabei soll jeder Ball zwei solcher Vektoren haben: Einen, um die Position zu speichern, und ein weiterer für die Richtung.

Ihr Konstruktor soll nun einen `PVector pos` entgegennehmen und speichern. Wenn Sie eine neue Instanz erzeugen, geben Sie anstatt von `mouseX` und `mouseY` einen Vektor aus diesen: `new PVector(mouseX, mouseY)`. Den Geschwindigkeitsvektor `PVector vel` können Sie mithilfe der Methode `PVector.fromAngle(rad)` generieren.

Hinweis: *Um einen zufälligen Winkel zwischen* 0 *und* 360° *zu erhalten, können Sie* `radians(random(360))`*, oder noch kürzer,* `random(TWO_PI)` *verwenden.*
Hinweis: *Mit* `myVector.x` *und* `myVector.y` *können Sie auf die Werte eines Vektors zugreifen. Mit* `myVector.add(otherVector)` *addieren sie einen anderen Vektor auf einen Vektor.*

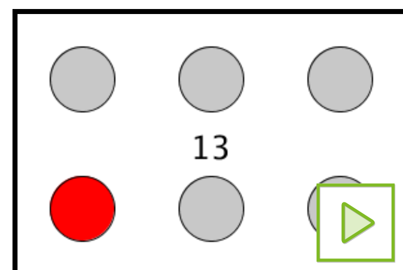
414

67 Z.



Aufgabe 6 – Hau den Maulwurf

Programmieren Sie eine Variante des Spiels „Hau den Maulwurf“, bei dem Maulwürfe oder andere schlagenswerte Figuren aus Löchern im Boden auftauchen und Spielende sie schnell genug durch Klicken der Maustaste treffen muss.

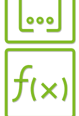


Schreiben Sie eine Klasse `class Mole`, welche Informationen zu Position und derzeitigem Zustand eines Maulwurfs speichern soll. Verwenden Sie ein Array, um alle Maulwürfe zu speichern und über diese zu iterieren.

Hinweis: *Mithilfe von* `dist(mouseX, mouseY, x, y)` *können Sie den Abstand zwischen der Maus und einer Position* `x, y` *messen.*

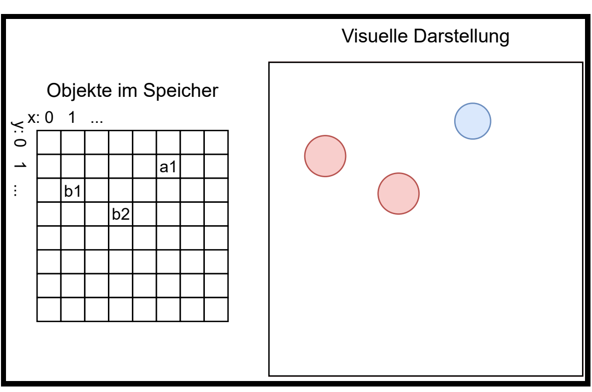
402

83 Z.



Aufgabe 7 – Räuber-Beute-Simulation

Programmieren Sie eine Räuber-Beute-Simulation.



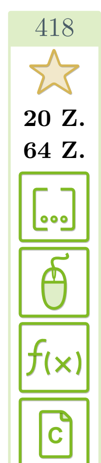
Teilen Sie dazu den Anzeigebereich zunächst in ein Gitter ein, indem Sie eine `PVector posToGrid(PVector pos)`-Funktion schreiben. Diese soll eine Position auf dem Gitter entgegennehmen, und die reale Position auf dem Zeichenbereich ausgeben.

Schreiben Sie nun eine Klasse `Animal`, welche die Position des Tieres (im Speicher, also in einem Zahlenraum von z.B. 0-49) sowie dessen Art (Räuber („2“) oder Beute („1“)?) speichert.

Definieren Sie in der Klasse die folgenden Methoden:

1. `void move()` - hier soll eine zufällige Richtung ausgewählt werden, und in diese drei Schritte bewegt werden.
2. `void show()` - hier soll das Tier gezeichnet werden. Legen Sie die Farbe anhand von der Art fest, und zeichnen Sie das Tier nur, wenn es noch nicht gefressen wurde.
3. `void attack(Animal[] preyArray)` - iterieren Sie über das gegebene Array, und rufen sie bei allen Tieren, welche die gleiche Position wie die Instanz selbst haben, `p.eat()` bzw. `preyArray[i].eat()` auf.
4. `void eat()` - aktualisiert den Wert von `dead` („gefressen“) auf `true`.

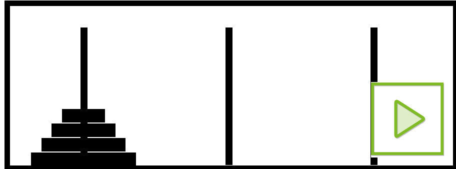
Erstellen Sie dann separate Arrays für Beute- und Jagdtiere, und führen Sie für alle Werte des `prey`-Arrays jeweils die Methoden `p.move()` und `p.show()` auf. Für Elemente des `hunters`-Arrays rufen Sie zu diesen Methoden zusätzlich `h.attack(prey)` auf.



Aufgabe 8 – Türme von Hanoi

Das Spiel **Die Türme von Hanoi** basiert auf einfachen Regeln:

- Es gibt drei Stangen, wobei auf der ersten Stange mehrere Scheiben mit einem Loch in der Mitte aufeinandergestapelt sind.
- Die Scheiben haben unterschiedliche Größe, wobei die größte Scheibe immer unten liegt.
- Der/die Spieler:in muss alle Scheiben vom Startstapel (links) auf den rechten Stapel legen.
- Dabei kann er/sie immer nur die oberste Scheibe von einem Stapel aufnehmen und darf nie eine größere Scheibe auf eine kleinere legen.



8.1 Stapel

Um dieses Spiel zu realisieren, werden Sie zunächst eine Möglichkeit benötigen, einige Schreiben bzw. Werte zu speichern. In der Programmierwelt spricht man hier wie auch im Alltag von einem *Stapel*. Einen solchen Stapel wollen wir nun als Klasse `class Stack` implementieren.

Eine Instanz von `Stack` soll ein Array `int[] values` von Größe 10, sowie eine Variable `int size`, welche die derzeitige Größe (Höhe) des Stapels speichert, haben.

Die Klasse soll die folgenden Methoden implementieren:

1. `void push(int i)` - legt den Wert `i` oben auf den Stapel.
2. `int pop()` - entfernt den obersten Wert des Stapels und gibt ihn zurück.
3. `int[] toArray` - gibt den Stapel als Array aus. Dabei muss hier ein neues Array erstellt werden, da `values` eine feste Größe hat.

Dabei soll der Konstruktor keine Werte festlegen, sondern nur `size` auf 0 setzen und das Array `values` initialisieren.
Machen Sie sich die Variable `size` zu nutzen, um bei `push` und `pop` den richtigen Wert auszugeben oder an der richtigen Stelle zu speichern. Mit `size` können Sie auch die Schleife limitieren, mithilfe wessen Sie das neue Array erstellen, um nicht schon wieder „gelöschte“ Werte auszugeben.

8.2 Das Spiel

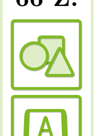
Nun sind wir schon weiter als man denken mag:

Mithilfe von drei dieser Stapel können wir nun die drei Stäbe erstellen. Schreiben Sie das Event `void mouseReleased()`, wo Sie ermitteln, welcher Stab angewählt wurde, oder falls schon einer angewählt wurde, wohin der oberste Wert verschoben werden soll. Wie genau Sie dieses Problem lösen, ist dabei Ihnen überlassen.

Schreiben Sie dann eine Funktion, die an einer bestimmtem `x`-Position einen Turm mithilfe eines Stapels malt, und rufen Sie diese für jeden Stapel in `draw()` auf.

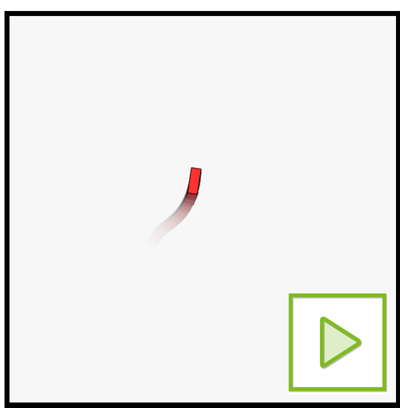
431

66 Z.



Aufgabe 9 – Rennauto

Schreiben Sie ein kleines Rennauto-Spiel.

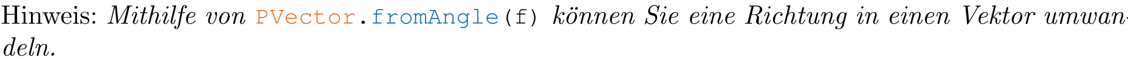


Dafür benötigen Sie eine Klasse `class Car`, welche eine Position `PVector pos` sowie eine Richtung `float facing` eines Autos speichern soll.

Jedes mal, wenn der/die Spieler:in die Pfeiltaste `UP` drückt, soll sich das Auto nach vorne bewegen. Die Pfeiltasten `LEFT` und `RIGHT` sollen das Auto ein kleines bisschen in diese Richtung drehen (bzw. `facing` erhöhen oder reduzieren).
Dies können Sie mit Methoden in der Klasse lösen, oder alternativ eine Funktion schreiben, die die entsprechenden Variablen einer Instanz von `Car` beeinflusst.

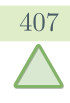
Schreiben Sie zusätzlich eine Methode `void show()` in der Klasse, welche das Auto darstellt. Falls Sie möchten, können Sie das Auto in der richtigen Rotation zeichnen.

Hinweis: *Mithilfe von* `PVector.fromAngle(f)` *können Sie eine Richtung in einen Vektor umwandeln.*

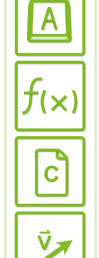


Hinweis: *Wenn Sie die Events* `void keyPressed()` *und* `void keyReleased()` *verwenden, können Sie den jeweiligen* `keyCode` *mit einer* `boolean`*-Variable speichern, und so immer wissen, welche Tasten derzeit gedrückt sind, und welche nicht.*
Hinweis: *Mithilfe von* `myVector.setMag(f)` *können Sie die Länge eines Vektors festlegen (und so das Auto z.B. schneller fahren lassen).*

407

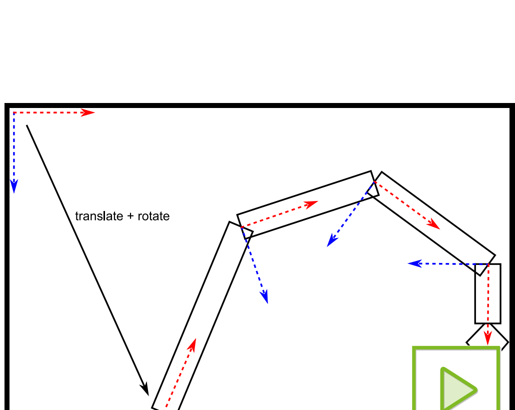


46 Z.



Aufgabe 10 – Roboterarm

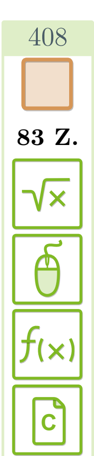
Schreiben Sie ein Processing-Programm, welches einen Roboterarm mit einigen Gelenken simuliert.



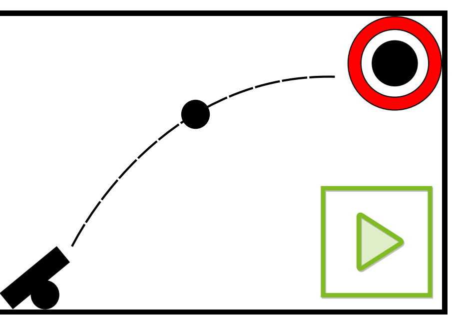
Der/die Benutzer:in soll den Arm bewegen können, indem er/sie mit den Zahlentasten ein Gelenk auswählt, und dann mit den Pfeiltasten das entsprechende Gelenk in eine Richtung bewegt.

Hinweis: *Verwenden Sie eine Klasse* `class Segment`*, um die einzelnen Segmente sinnvoll zu speichern.*
Können Sie eine Methode `void show()` *formulieren, die keine Position entgegennimmt, sondern von einem bereits rotierten Zeichenbereich Gebrauch macht, und diesen nur weiter rotiert und verschiebt?* Hinweis: *Der* `int`*-Wert des Charakters* `1` *ist* `49`*, von* `2` `50`*, usw.*

Falls Sie eine Herausforderung suchen, können Sie auch versuchen, dem Roboterarm die Fähigkeit zu geben, eine Box aufzuheben.



Aufgabe 11 – Kanonenspiel

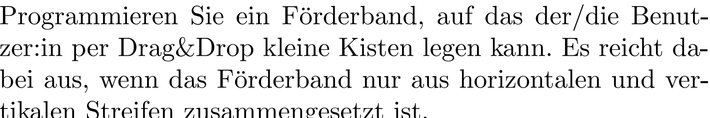


Schreiben Sie ein Spiel, bei dem der/die Benutzer:in mit der Maus eine Kanone ausrichten und mit einem Klick abfeuern kann. Ziel des Spiels ist es, eine Zielscheibe zu treffen.

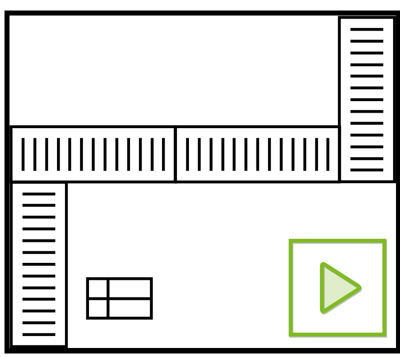
Sie können das Spiel noch interessanter machen, indem Sie z.B. das Ziel bewegen lassen oder Wind einbauen, der sich bei jedem Schuss wieder ändert.

Hinweis: *Die Kanonenkugel bekommt von der Kanone eine Initialgeschwindigkeit. Ab diesem Zeitpunkt wird diese Geschwindigkeit nur noch von der Gravitation beeinflusst (Beschleunigung nach unten).*
Hinweis: *Mithilfe der* `acos(i)`*-Funktion können Sie die Ausrichtung der Kanone berechnen. Geben Sie dafür die Differenz von* `mouseX` *und der x-Position der Kanone, geteilt durch die Distanz zwischen Mausposition und Position der Kanone, in diese Funktion.*
Hinweis: *Setzen Sie eine Variable gleich* `null`*, wenn ein Objekt gelöscht werden soll.*

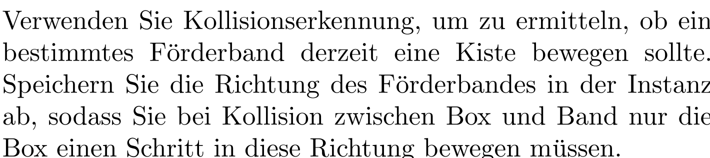
Aufgabe 12 – Förderband



Programmieren Sie ein Förderband, auf das der/die Benutzer:in per Drag&Drop kleine Kisten legen kann. Es reicht dabei aus, wenn das Förderband nur aus horizontalen und vertikalen Streifen zusammengesetzt ist.

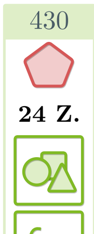


Implementieren Sie dazu eine Klasse `class ProductionLine` für die Förderbänder sowie eine Klasse `class Box` für die Kisten.



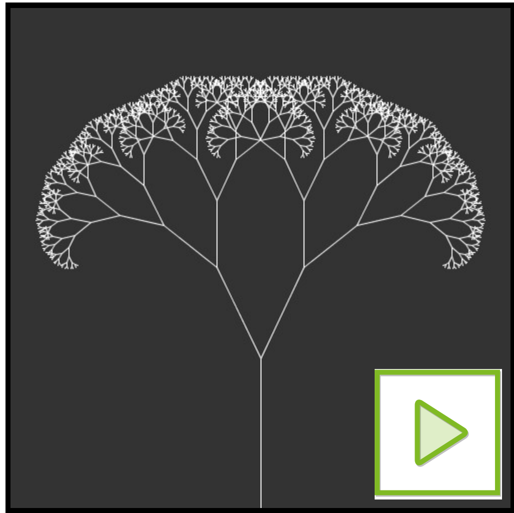
Verwenden Sie Kollisionserkennung, um zu ermitteln, ob ein bestimmtes Förderband derzeit eine Kiste bewegen sollte. Speichern Sie die Richtung des Förderbandes in der Instanz ab, sodass Sie bei Kollision zwischen Box und Band nur die Box einen Schritt in diese Richtung bewegen müssen.

Hinweis: *Mithilfe von* `pmouseX` *und* `pmouseY` *können Sie die Mausposition vor einer Frame abrufen, und so ermitteln, um wie viel eine Box - sollte sie denn angewählt sein - bewegt werden soll.*



Aufgabe 13 – Fraktale Bäume

Zeichnen Sie einen fraktalen Baum. Mit relativ wenig Code kann das Ergebnis am Ende so oder so ähnlich aussehen:



Fangen Sie simpel an:
Nehmen Sie den folgenden Code als Startpunkt, mit dem der „Baumstamm“ und ein Ast dargestellt wird. Sie müssen natürlich noch sinnvolle Werte für die Variablen `len` und `angle` finden.

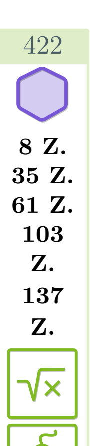
```
translate(width/2, height);
line(0, 0, 0, -len);
translate(0, -len);
rotate(angle);
line(0.67 *len);
```

Zu dem Ast nach rechts soll jetzt noch ein Ast nach links dazu kommen. Das erreichen Sie, indem Sie mit `pushMatrix()` und `popMatrix()` die Verschiebung und Drehung des ersten Astes „rückgängig machen“, und anschließend den zweiten Ast zeichnen.

Dann wollen wir aber nicht jede weitere Verästlung „von Hand“ programmieren, sondern Rekursion für uns arbeiten lassen. Wir schreiben das Programm um, und erstellen eine Funktion (z.B. `void branch(int len)`), die einen Strich zeichnet und sich dann wieder selbst aufruft (mit einem entsprechend kleineren Wert als `len`-Parameter). Diesen Vorgang nennt man *Rekursion*.

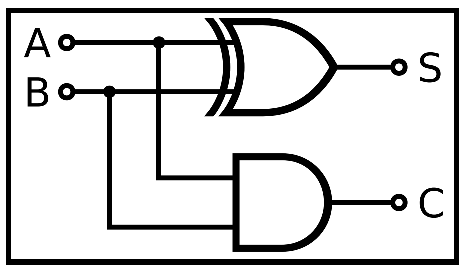
Ganz wichtig dabei: Denken Sie an eine sinnvolle Abbruchbedingung, sonst entsteht eine Endlosschleife.

Hinweis: *Mithilfe von* `frameCount` *könnten Sie den Baum animieren, indem Sie diesen Wert als Faktor auf* `angle` *anwenden.*



Aufgabe 14 – Logische Schaltkreise

Wir wollen einen logischen Schaltkreis programmieren. Dank der Objektorientierung können wir uns dabei auf die sog. „Klassenerweiterung“ verlassen. Dazu werden Sie später in dieser Aufgabe mehr erfahren.



Hinweis: *Diese Aufgabe ist sehr umfangreich und wird Stoff behandeln und erklären, welcher nicht in den Vorlesungen vorkommt. Versuchen Sie sich gern!*

14.1 Logisches Modul

Ein logischer Schaltkreis kann aus vielen verschiedenen Modulen aufgebaut sein:

- Ein **Eingabesignal**, das dauerhaft entweder 1 oder 0 ausgibt.
- Ein **NOT-Gatter**, das den Eingabewert umkehrt.
- Ein **OR-Gatter**, das 1 ausgibt, sobald ein Eingang 1 ist.
- Ein **AND-Gatter**, das nur 1 ausgibt, wenn beide Eingänge den Wert 1 haben. Sonst ist die Ausgabe 0.
- Ein **Schalter**, den der/die Benutzer:in durch Mausklick an und ausschalten kann.

Was diese Elemente alle teilen: Sie müssen mit den anderen Elementen verbunden werden. Deswegen wollen wir zunächst eine abstrakte Klasse `class Module` formulieren, die nicht mehr als Positionsinformationen und die verbundenen Module speichert.

Speichern Sie die Position in einen Vektor `PVector pos` und die verbundenen Module in einem Array `Module[] outgoing` (Ja, sowas ist möglich!). Wie der Name erahnen lässt, werden wir nur die Module in einer Instanz speichern, die von dieser ausgeht.

Schreiben Sie einen Konstruktor, welcher eine Position entgegennimmt und das Module-Array mit der Größe 10 initialisiert.

Ihr Ergebnis sollte also eine Klasse sein, welche theoretisch jede Art von Schalter oder Element sein könnte, wenn es etwas mehr Information speichern würde.

14.2 Die Klassenerweiterung

Nun wollen wir unsere abstrakte Klasse verwenden, um alle Elemente zu erstellen. Dafür werden wir für jedes Element eine Klasse schreiben, welche die Klasse Modul *erweitert*. Zunächst werden wir das Konzept anhand des OR-Gatters erarbeiten.

Die Schreibweise einer solchen Klasse, z.B. im Fall des OR-Gatters ist:
`class OrGate extends Module { ... }.`

Die Erweiterung funktioniert dabei so, wie man es erwarten würde: eine Instanz der Klasse `OrGate` wird implizit auch die Attribute `pos` und `outgoing` haben - so sparen wir uns eine Menge an Schreibarbeit und müssen nicht für jede Art von Modul diese Variablen und später Methoden neu schreiben.

Damit das so reibungslos funktioniert, müssen wir allerdings der Klasse `Module` auch weiterhin die Information mitgeben, die es im Konstruktor benötigt - in unserem Fall die Position des Gates. Für die Kommunikation mit einer sog. Oberklasse gibt es das `super`-Keyword.

So ungefähr sollte der Konstruktor Ihres OR-Gatters aussehen:

```
1 OrGate(PVector pos) {
2     super(pos);
3 }
```

Den Konstruktor selbst könnten wir jetzt natürlich ganz angepasst für das OR-Gatter festlegen.

Schreiben Sie nun in der Klasse `Module` die Methode `void input(boolean input)`. Diese soll nur mit `println(s)` den Text „Modul kann keine Inputs aufnehmen!“ ausgeben.

Nun überschreiben wir diese Methode in der Klasse `OrGate` - wir schreiben also genau so auch hier noch einmal eine Methode `void input(boolean input)`. In dieser Methode sollen nun zwei boolsche Variablen `val1` und `val2` belegen. Verwenden Sie einen Counter, damit der erste Aufruf von `input()` das erste und der zweite Aufruf das zweite Feld belegt. Weitere Aufrufe sollen wieder mit einem `println` beantwortet werden.

Implementieren Sie zusätzlich in beiden Klassen eine `reset()`-Methode. In der Klasse `Modul` wird diese zunächst nichts tun - in `OrGate` hingegen soll sie den Counter sowie die beiden Felder zurücksetzen.

14.3 Visualisierung

Nun haben wir uns viel in einem sehr abstrakten Raum bewegt, ohne überhaupt zu sehen, was wir da programmieren. Es ist an der Zeit, etwas Licht ins Dunkle zu bringen:

Schreiben Sie für beide Klassen eine Methode `void show()`, welche das Modul an der eigenen Position zeichnet. Verwenden Sie dabei `square(x, y, d)`, um die Module zu zeichnen. Verwenden Sie Text oder weitere Zeichenfunktionen, um OR-Gatter als solche erkennbar zu machen.

Schreiben Sie nun die altbekannten `setup()` und `draw()` Funktionen. Speichern Sie dabei ein OR-Gate mithilfe von `Module m1 = new OrGate(new PVector(...)).`

Hinweis: *Mithilfe von* `rectMode(CENTER)` *und* `textAlign(CENTER)` *können Sie Text und Elemente zentriert zeichnen. Mit* `textSize(i)` *können Sie die Textgröße festlegen.*

Instanziieren sie einige Gatter und positionieren Sie sie auf der Bildfläche. Stellen Sie sie mit `m1.show();` in der `draw()`-Funktion dar.

14.4 Mehr Tricksereien und Input

Wir sind nun kurz davor, einen funktionierenden Schaltkreis darstellen zu können. In unserem Fall ist das ein sog. *Eingabesignal*, welches einen festen Wert in den Schaltkreis gibt. Legen wir uns also zunächst eine Klasse `class Input extends Module` an, welche also im Konstruktor zusätzlich zur Position auch einen Wahrheitswert entgegennimmt und speichert.

Nun brauchen wir eine Methode, die das eigene Signal versendet. Schreiben Sie zunächst die Methode `void send(boolean output)` in der Klasse `Module`. Dort soll über das Array `outgoing` iteriert werden, und bei jedem der Module die Methode `m.input(output)` aufgerufen werden. Prüfen Sie vor diesem Aufruf aber, ob das entsprechende Feld des Arrays `null` ist, und überspringen Sie den Methodenaufruf in diesem Fall.

Speichern Sie außerdem den Wahrheitswert der in `send` eingegeben wird in einer neuen Variable `boolean output`, anhand `this.output = output`.

Entscheiden Sie anhand dieser Variable, ob die Farbe des Moduls in `void show()` rot oder weiß sein soll. Außerdem soll die Methode `send` durch `outgoing` iterieren, und bei jedem Modul `m.show()` aufrufen. Achten Sie auch hier auf Felder, die `null` sind. Lassen Sie auch eine Linie von `pos` zu `m.pos` zeichnen.

Schreiben Sie nun in allen Klassen die Methode `calculateSignal()`:

- In `Module` soll ein `println(s)` mit dem Text „Modul kann kein Signal versenden!“ aufgerufen werden.
- In `Input` soll diese Methode `send(b)` mit dem gespeicherten Wahrheitswert aufrufen. Das geht ohne Probleme, weil durch die Vererbung auch die Unterklassen nun die soeben geschriebene Methode haben.
- In `OrGate` wird es etwas komplizierter - das Gate kann nur ein Output senden, wenn beide Felder belegt wurden, der Zähler also genau bei 2 liegt. Ist er kleiner, sind nicht alle Felder belegt - ist er größer, wurde das Signal bereits versendet. Ist diese Bedingung erfüllt, soll das Gate ebenfalls `send(b)` aufrufen - mit dem Wahrheitswert, der dort zu erwarten ist.

Die Methode `calculateSignal()` soll in allen Unterklassen am Ende der Methode `input(b)` aufgerufen werden. Später werden wir diese Methode „von Hand“ in den `Input`-Instanzen aufrufen.

Zuletzt optimieren wir unsere `void show()` Methode der Unterklassen: Rufen Sie als erstes `super.show()` auf - dadurch wird die Box bereits grundlegend gezeichnet, und außerdem werden so die nachfolgenden Module auch gezeichnet. Nach diesem Aufruf sollen Sie weiterhin die Details zeichnen, die das Gate und das Input als solche erkennbar machen.

14.5 Der erste Schaltkreis

Etwas fehlte uns noch: Die Methode `void reset()` in `Module` soll nun für alle Werte von `outgoing` ebenfalls `m.reset()` aufrufen. In Unterklassen müssen Sie nun auch hier jeweils `super.reset()` aufrufen. In `Input` können wir uns die Implementation von `reset()` komplett sparen - dann übernehmen Instanzen dieser Unterklasse einfach die Methode von `Module`.

Nun wollen wir unseren Lorbeerkranz einsammeln und Schaltkreise darstellen:

Definieren Sie ein globales Array `Module[] baseModules`, welches alle Module speichern soll, welche nicht von anderen Modulen aufgerufen werden, also meist alle Eingabesignale und Schalter (welche Sie später noch entwickeln können, wenn sie denn möchten.) Belegen Sie diese Array mit einigen Eingabesignalen.

Fügen Sie dann diesen Eingabesignalen neue Gates zu - da das etwas verwirrend sein könnte, hier ein Codebeispiel:

```
baseModules[0] = new Input(new PVector(100, 100), true);
baseModules[1] = new Input(new PVector(100, 200), false);

Module or = new OrGate(new PVector(200, 150));

baseModules[0].outgoing[0] = or;
baseModules[1].outgoing[0] = or;
```

Nun sind Sie fast soweit! Schreiben Sie in `draw()` nun drei for-each Schleifen - jede davon muss über alle Werten von `baseModules` iterieren und in dieser Reihenfolge diese Methoden aufrufen:

1. `m.reset()`
2. `m.calculateSignal()`
3. `m.show()`

Aufgrund den Abhängigkeiten von Modulen untereinander sind wir nicht in der Lage, diese Schleifen ohne weiteres zu vereinen.

Nun können Sie nach Interesse weitere der aufgelisteten Module realisieren und Schaltkreise bauen. Ihrer Kreativität ist freien Lauf gelassen - solange Sie mit den gleichen Regeln der Klassenerweiterung arbeiten, könnten Sie sich auch eigene logische Module ausdenken.