Projektdokumentation

Crazier Machines

01.07.2020

Dokumentationsattribute	
Projektname	Crazier Machines
Namen der Teammitglieder	Simon Weck, Paul Amelingmeyer, Patryk Watola, Benjamin Jäger

1. Problemstellung

Bei unserem Projekt handelt es sich um eine Neuinterpretation des PC-Spiels “Crazy Machines”, welches im Jahr 2004 von FAKT Software entwickelt wurde. Im Spiel müssen Rätsel in Form von Physikexperimenten gelöst werden.

Screenshot aus “Crazy Machines” von FAKT Software, 2004

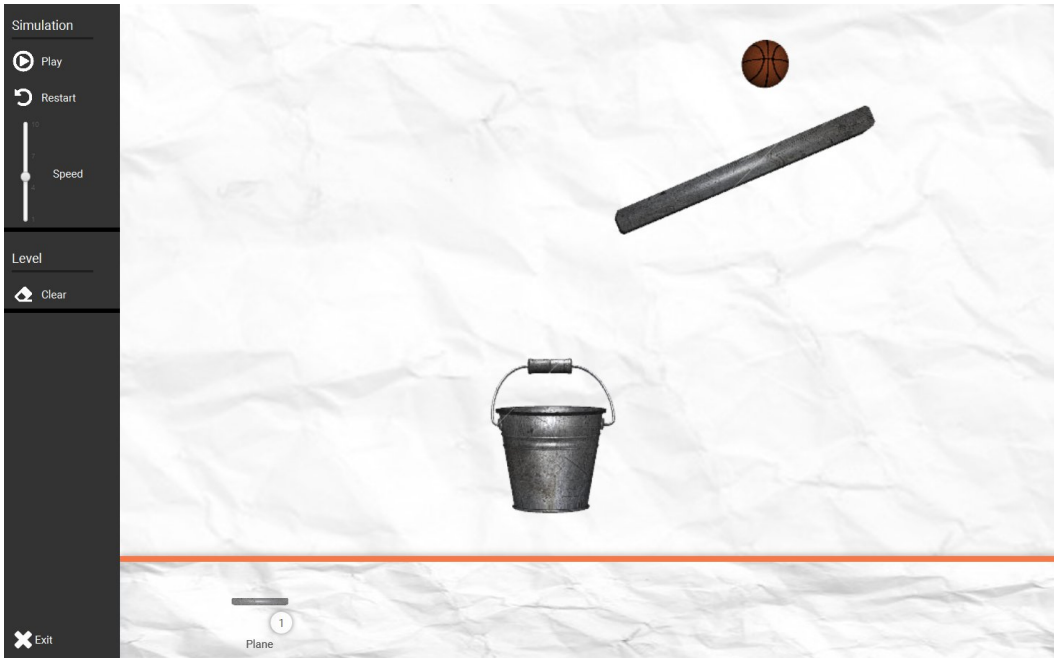
Die einzelnen Level bestehen aus einer Anordnung von vorgegebenen Bauteilen, in die vom Spieler zusätzliche Bauteile hinzugefügt werden müssen, um eine Kettenreaktion zu vervollständigen und somit die Aufgabenstellung zu erfüllen.

Zusätzlich gibt es einen Editor, in dem der Nutzer Elemente frei platzieren kann, um mit der Physik herum zu experimentieren oder selber Rätsel für seine Freunde zu erstellen.

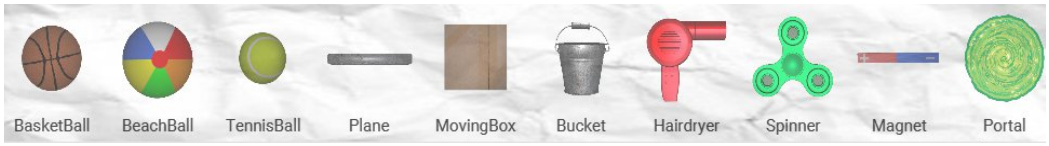
2. Kugelbahn und ausgewählte Elemente

Wir visualisieren die Kugelbahn in 3D mit der Java-OpenGL-Anbindung JOGL. Die Objekte bewegen sich dabei in einer Ebene, sodass Kollision, Bewegung etc.

in 2D stattfinden. Der Benutzer hat jedoch durch Bewegen der Kamera die Möglichkeit, die Objekte in 3D zu betrachten. Die Objekte sind dabei dreidimensional und größtenteils in externen Programmen wie Maya erstellt oder aus 3D-Modelldatenbanken bezogen und in das Projekt importiert.
Unsere Kugelbahnen sind die einzelnen kleinen Level, die der Nutzer zur Auswahl hat. Innerhalb eines Levels gibt es einige Objekte wie Ebenen oder Bälle, die bereits platziert wurden, welche der Nutzer nicht bearbeiten kann. Dazu hat der Nutzer in einer Leiste einige Objekte zur Verfügung, die er frei platzieren kann um das Level zu schaffen. Ziel ist es immer den Basketball in einen Eimer zu bringen.



Hier sieht man ein Level, in dem der Nutzer eine Ebene hat, die er beliebig platzieren, rotieren oder skalieren kann, um den Basketball in den Eimer zu befördern. Schafft er dies erscheint ein Fenster, das zum Sieg gratuliert und bei Klick auf einen Button den Nutzer zurück zur Levelauswahl befördert.

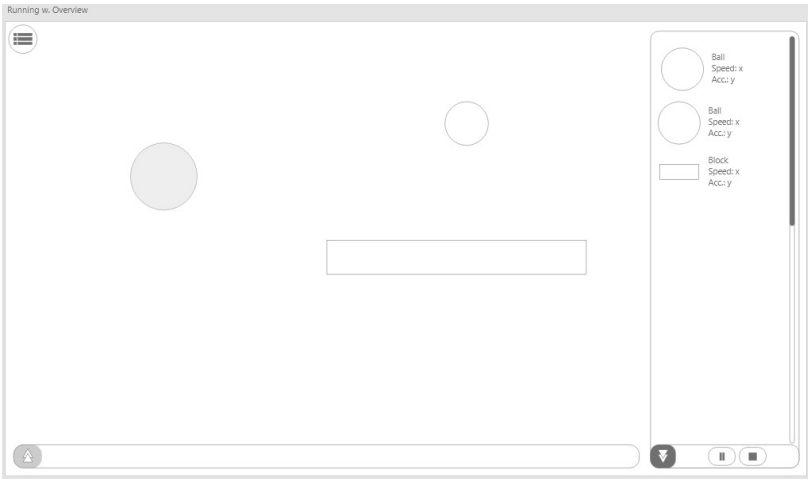


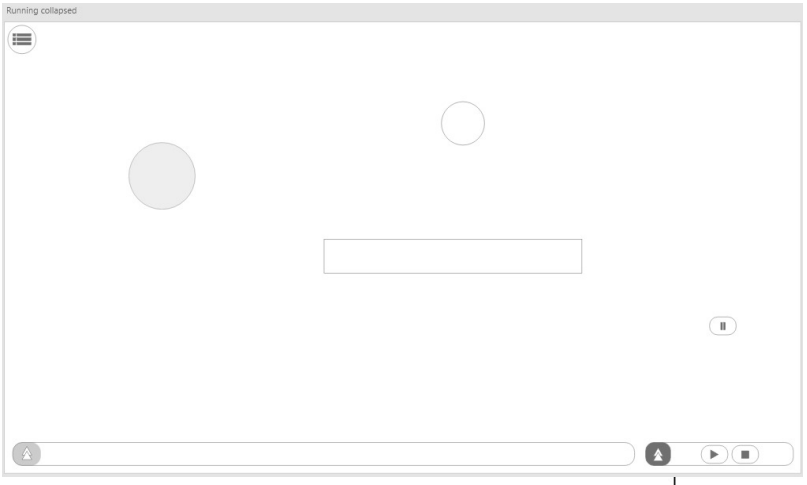
Wir bieten eine Vielzahl an Objekten an. Wir haben drei bewegliche Bälle, die sich jeweils in der Größe und dem Gewicht unterscheiden, eine statische Ebene und einen Eimer, sowie ein Portal, welches ein bewegliches Objekt an einen anderen Ort teleportiert. Unser drehendes Objekt ist ein sogenannter ”Fidget Spinner”, welcher sich kontinuierlich dreht. Außerdem haben wir einen Föhn, welcher in einem dreieckigen Bereich Luft herausbläst und Objekte damit abstößt. Der Magnet besitzt zwei Pole und zieht bzw. stößt alle Objekte abhängig von der Entfernung ab. Zuletzt haben wir noch einen Umzugskarton, der ähnlich wie die Bälle ein bewegliches Objekt darstellt, aber als Viereck modelliert worden ist.

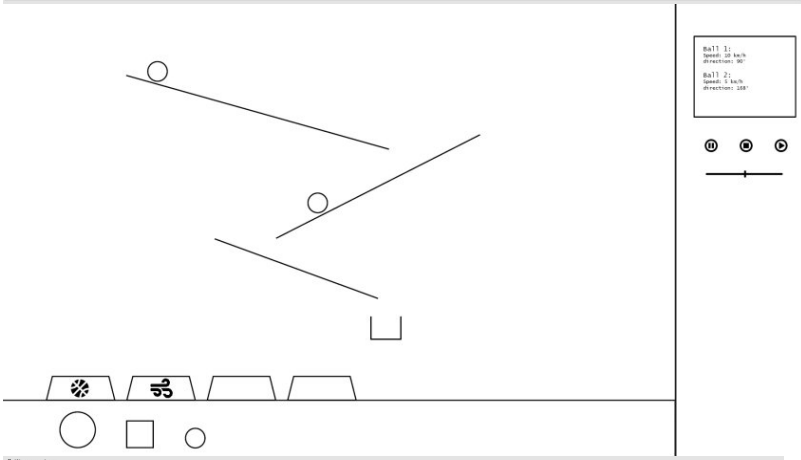
3. Benutzeroberfläche

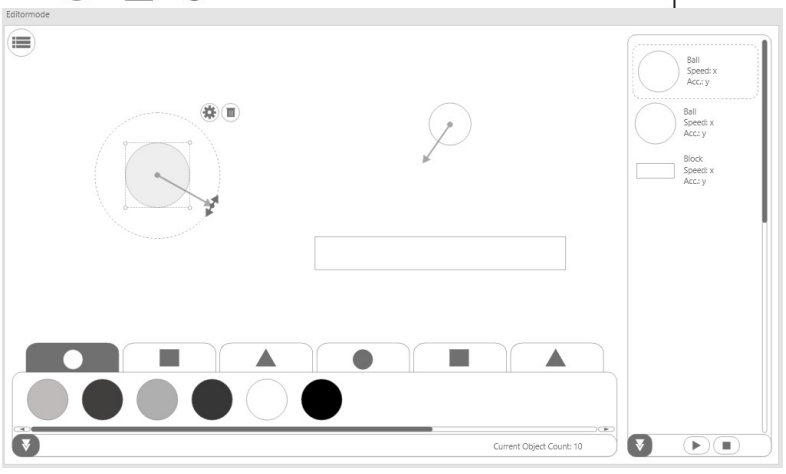
Bei der Benutzeroberfläche haben wir uns grundsätzlich an Standards der Interaktion mit einer Oberfläche aus der Vorlesung gehalten. Dabei war uns sehr wichtig, dass der Nutzer jederzeit Feedback auf sein Handeln erhält, wie zum Beispiel das Feedback beim Hovern über einen Button oder ein Objekt, wo sich der Maus-Cursor ändert, damit der Nutzer merkt, dass er sich gerade über einem klickbaren Bereich befindet. Außerdem haben wir viel mit kleinen Icons und Bildchen gearbeitet, damit der Nutzer UI-Elemente schneller erkennt - ohne dafür permanent lesen zu müssen. Dabei haben wir auch auf die Verständlichkeit der UI geachtet und zum Beispiel Tooltips an die Icons geheftet, damit im Notfall die Dinge klar und verständlich sind. Über die Einstellungen kann der Nutzer sich die Oberfläche auch zu einem gewissen Grad personalisieren, wie zum Beispiel das Abändern des Hintergrunds oder die Einstellung von Kontrast sowie Helligkeit der Applikation. Zu unserer Oberfläche gehören auch kleine Soundeffekte, die abgespielt werden und Musik, die das Erlebnis des Programms noch einmal abrunden. Textfelder, um Objekteigenschaften zu verändern, wurden so modifiziert, dass der Nutzer auch keine Möglichkeit hat falsche Eingaben zu tätigen, was das Programm zum Abstürzen bringen könnte. Das Fenster, in welchem sich die gesamte Oberfläche befindet, ist auch beliebig skalierbar, da viel mit Scrollleisten gearbeitet wurde, die sich gut für UI-Vergrößerung bzw. -Verkleinerung eignen.

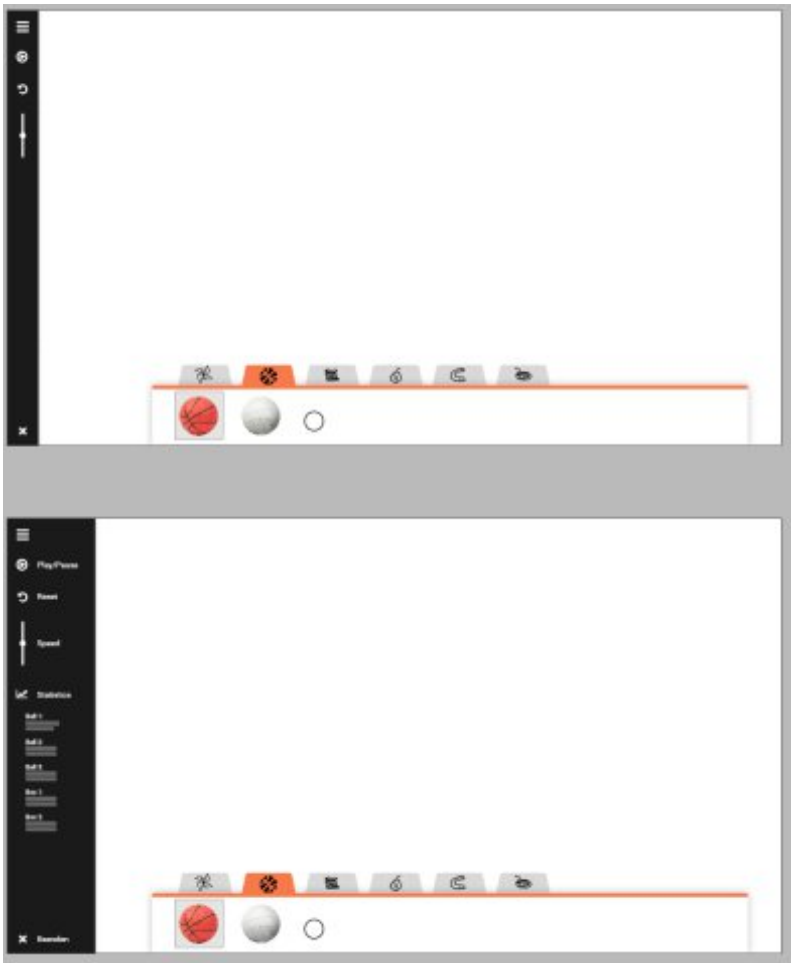
a. Konzept




Running collapsed


Ball 1:
Speed: 10 km/h
direction: 90°
Ball 2:
Speed: 5 km/h
direction: 180°


Editormode
Ball
Speed: x
Acc.: y
Ball
Speed: x
Acc.: y
Block
Speed: x
Acc.: y
Current Object Count: 10



b. Hauptmenü

Im Hauptmenü gibt es fünf Optionen, die der Nutzer auswählen kann. Er kann über “Exit” das Programm beenden, über “Settings” Einstellungen vornehmen, über “Load Level” ein zuvor selbst erstelltes Level auswählen

und spielen, über “Level Editor” ein eigenes Level erstellen und über “Level Selection” kann der Nutzer aus zwölf bereits erstellten Leveln wählen und diese spielen - ähnlich einem Kampagnenmodus. Alle Buttons besitzen einen mittels CSS implementierten Hover-Effekt, der ihnen Schlagschatten gibt sowie einen im Kasten erscheinenden Haken, damit ersichtlicher wird, dass der Button auswählbar ist.

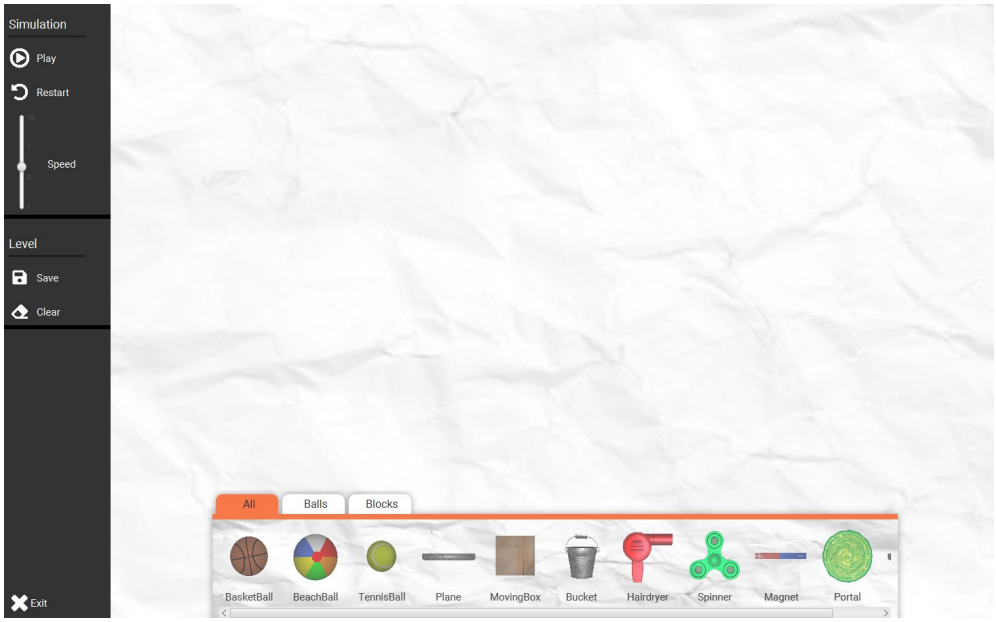
c. Einstellungen

Im Einstellungsmenü kann eine Vielzahl an Einstellungen vorgenommen werden. Erwähnenswert ist hier, dass der Hintergrund der Simulation selbst ausgewählt werden kann. Der Nutzer kann selbst ein Bild auswählen oder zwischen drei bereits abgelegten Bildern auswählen, sobald er auf das Bild in den Einstellungen klickt. Der Hintergrund wird auch hier in den Einstellungen als Preview angezeigt.

d. Level Auswahl

Hier kann der Nutzer zwischen zwölf bereits erstellten Leveln auswählen und diese spielen. Erwähnenswert hier ist, dass Level, die bereits geschafft wurden, grün markiert werden und diese speichern sich auch lokal in einer Datei. Das heißt, beendet man das Programm und öffnet es später wieder, werden die geschafften Level immer noch grün markiert.

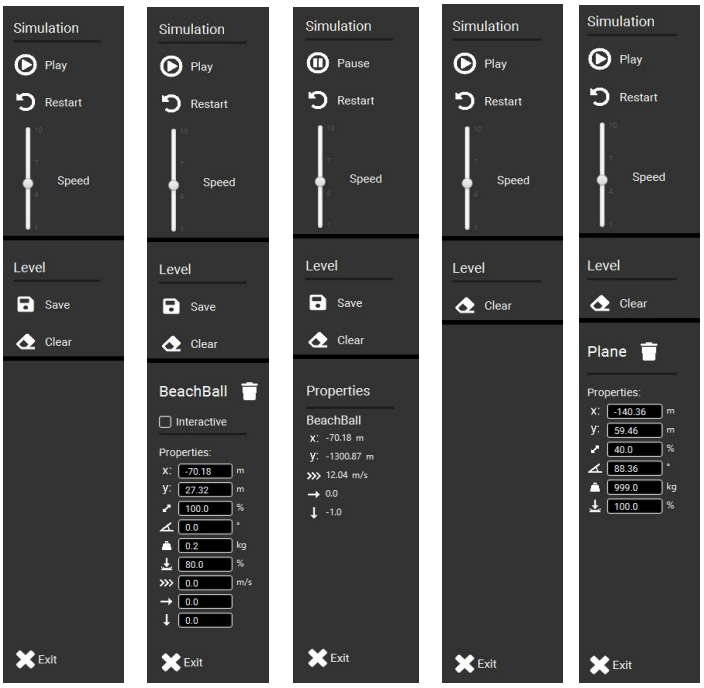
e. Level Editor



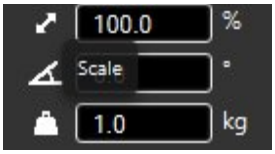
Wählt man “Level Editor” im Hauptmenü aus, gelangt man zu diesem Screen.
Der Level Editor setzt sich aus 3 Komponenten zusammen: die linke schwarze Leiste ist die SideBar, in der Mitte befindet sich das Canvas, auf dem die Objekte für die Simulation gezeichnet werden, und die TabPane

im unteren Bereich wird genutzt, um Objekte in die Szene zu setzen.

i. SideBar



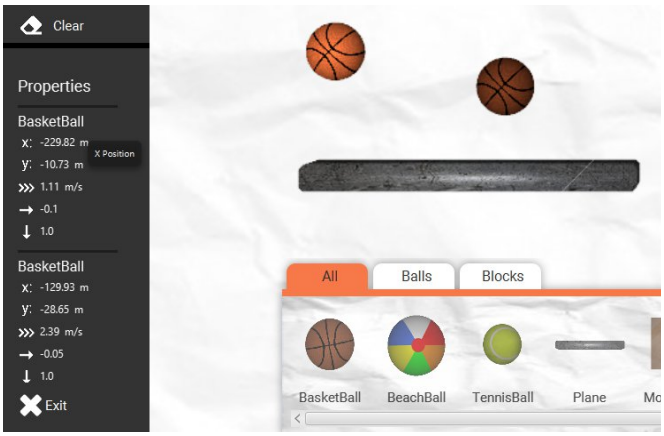
Die SideBar ist die Kontrolleinheit der Simulation. Über die Pause, Restart und Playback-Anzeige kann die Simulation gesteuert werden. Wird der Play-Button betätigt, wandelt er sich zu einem Pause-Button. Darunter im Levelabschnitt befinden sich zwei Buttons. Der Save-Button speichert das Level in eine lokale Datei. Dieser Button wird nur angezeigt, wenn man sich im Level Editor befindet - spielt man eines der Level wird dieser nicht mehr angezeigt. Der Clear-Button löscht alle Objekte, die sich gerade in der Szene befinden.
Wählt man ein Objekt durch Mausklick in der Szene aus, erscheinen in der Sidebar die Eigenschaften des Objektes (SideBar 2). Dieser Block enthält den Namen des Objektes, ein Button zum Löschen des Objektes und diverse Textfelder über die die gezeigte Eigenschaft eingestellt werden kann. Dazu gibt es eine Checkbox, welche das Objekt interaktiv macht - das heißt folgendes: speichert man das Level und spielt es später, kann man dieses Objekt im Level selbst platzieren, um das Level zu schaffen. Alle Textfelder sind mit einer Einheit und einem Icon versehen, um die gezeigte Eigenschaft schneller zu erkennen. Die Icons besitzen auch ein Tooltip der noch einmal die Eigenschaft als Text anzeigt, sollte man das Icon nicht verstehen.



Der Eigenschaftsblock passt sich dynamisch an je nachdem welches Objekt ausgewählt wurde. Wird zum Beispiel ein Magnet ausgewählt, hat dieser noch eine Eigenschaft, welche die Stärke des Magneten einstellbar macht, ein Ball hingegen hat diese

Eigenschaft nicht. Dafür hat dieser aber Eigenschaften wie die Geschwindigkeit und Bewegungsrichtung. Außerdem aktualisiert sich der Block live. Sollte man zum Beispiel einen Ball auswählen und durch die Szene ziehen, wird die Position direkt aktualisiert. Die Textfelder um die Eigenschaften sind auch so modifiziert, dass sie nur positive und negative Zahlen annehmen, und ändern die Eigenschaft des Objektes direkt ohne noch einmal Enter drücken zu müssen.

Befindet sich ein bewegliches Objekt in der Szene, welches Eigenschaften hat wie Bewegungsrichtung und Geschwindigkeit, werden diese live angezeigt, wenn die Simulation gestartet wird (SideBar 3).
Hovert man über diesen Bereich, leuchtet der Ball auf, der mit diesen Eigenschaften verbunden ist.

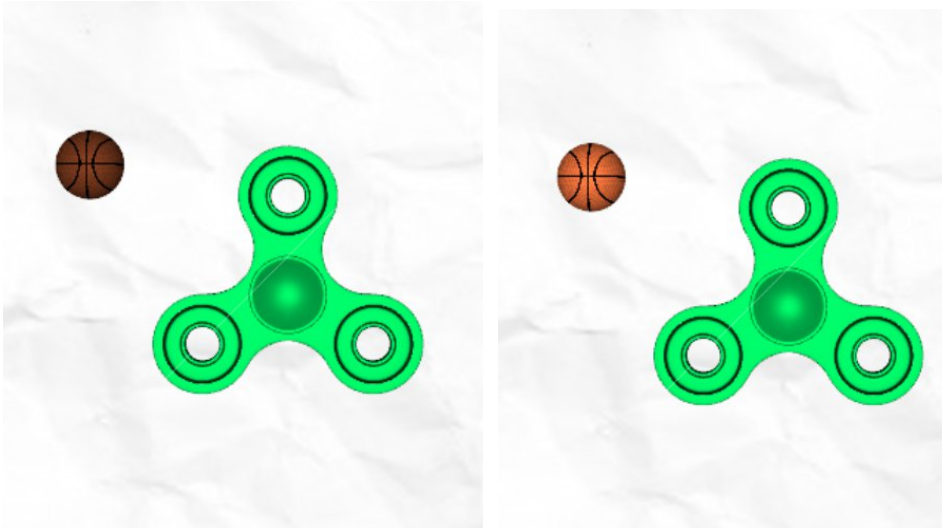


Auch hier werden Tooltips für die entsprechenden Eigenschaften angezeigt und hier kann man gut erkennen, dass der Ball aufleuchtet, wenn man über den Bereich hovert.

Im 4. und 5. SideBar-Bild sieht man noch einmal die angepasste SideBar, wenn man ein Level spielt. Dort kann das Level nicht gespeichert werden und bei den Eigenschaften des Objektes fehlt die Checkbox, die das Objekt interaktiv macht.
Die SideBar besitzt dazu ganz unten einen Exit-Button, welcher einen zurück zum Hauptmenü bringt.

ii. Canvas

Wir arbeiten mit JOGL um die Objekte dreidimensional auf dem Canvas zu zeichnen. Mit gehaltenem Rechtsklick kann man die Szene rotieren und mit gehaltener mittleren Maustaste kann man die Szene verschieben.



Hovert man über ein Objekt, so wird dieses heller und der Mauszeiger ändert sich zu einer Hand. Klickt man jetzt auf das Objekt, öffnet sich eine Oberfläche, mit der man das Objekt verschieben, skalieren und rotieren kann (siehe Abschnitt “Rotieren und Positionieren von Objekten”).

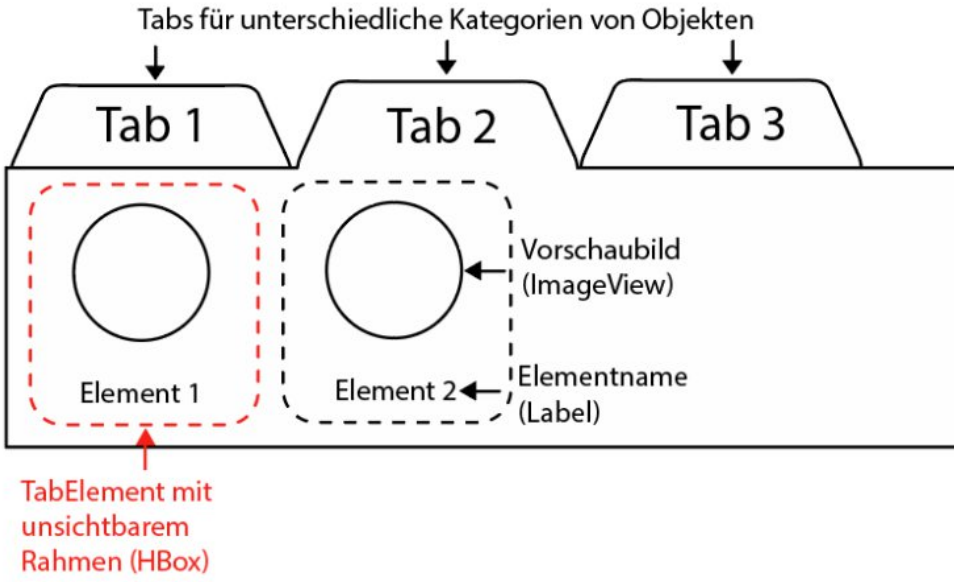
iii. TabPane

Die TabPane im unteren Bereich ist eines der zentralen Elemente des Editors und ist dafür zuständig, dem Nutzer alle Objekte für die

Szene zur Verfügung zu stellen. Sie besteht aus mehreren Tabs mit ScrollPanes, in welchen “TabElements” angezeigt werden. TabElement ist eine Klasse, welche alle Informationen über das Objekt beinhaltet, wie Name und Vorschaubild. Diese Elemente können mittels Drag-and-Drop in die Szene gezogen werden, um Objekte zu erstellen.

iv. Drag-and-Drop

Das Drag-and-Drop ist umgesetzt worden, indem an die einzelnen Objekte in der TabPane ein EventListener gehängt wurde. Wenn ein Drag gestartet wird, dann wird über ein Attribut das aktuell gedraggte Objekt in der TabPane gespeichert.
Da das Drag-and-Drop über mehrere Java-Klassen hinweg funktioniert, wird das Objekt beim Start des Drags als Attribut gespeichert. Dieses befindet sich dann in der EditorTabPane und wird am Ende des Drag-and-Drop-Prozesses wieder zurückgesetzt. Sobald der Drag gestartet wurde, wird das gedraggte Objekt angefragt und das Vorschaubild wird geladen. Dieses wird für die Animation benötigt. Um es zu animieren ist innerhalb der Szene eine StackPane mit zwei Elementen, eine BorderPane für das Layout und eine Pane, auf welcher das Bild dargestellt wird, gesetzt worden. Da wir OpenGL nutzen, um die Simulation darzustellen, war das temporäre Zeichnen neuer Objekte relativ schwierig. Daher haben wir uns dafür entschieden ein Bild des Objekts innerhalb der Szene zu zeichnen. Vor der Animation werden dann BorderPane und Pane in ihrer Reihenfolge getauscht und nach der Animation dementsprechend wieder zurückgetauscht. Beim Drop wird, nachdem wieder die BorderPane das vorderste Element innerhalb der StackPane ist, der Konstruktor für das Objekt aufgerufen. Hierbei wird die Mausposition aus der OpenGL-Szene bestimmt und dem Konstruktor übergeben. Danach wird das Attribut zurückgesetzt.



f. Rotieren und Positionieren von Objekten

Wählt man ein Objekt in der Szene aus, erscheint diese Oberfläche. Diese ist zur Objekttransformation da. Über den Kreis kann das Objekt rotiert werden, über das Viereck skaliert und im Zentrum kann das Objekt neu positioniert werden. Alle Features funktionieren über einen Maus-Drag. Der Maus-Cursor ändert sich jeweils auch immer, wenn man über einen der drei Bereiche hovert, um dem Nutzer deutlich zu machen, dass er hier Skalieren, Rotieren oder Positionieren kann.
Da wir mit JOGL arbeiten, müssen wir Methoden selbst schreiben, welche erkennen ob der Mauscursor sich über einem Objekt befindet oder nicht. Dazu nutzen wir die Boxen jedes Objektes, welche auch zur Kollisionserkennung innerhalb der Simulation genutzt werden. Dadurch können wir erkennen, ob über ein Objekt gehovert wird oder ob auf ein Objekt geklickt wurde.

Dafür sind die statischen “ObjectPickingMethods” vorhanden. Diese

Klasse enthält verschiedene Methoden, die sowohl die Objekte selbst als auch das Interface um sie herum auswählen können. Die Methoden zur Erkennung haben wir selbst implementiert.
Die gesamte Erkennung eines Objekts läuft also zweischrittig: nacheinander wird jedes Objekt betrachtet, die Art überprüft und dann geschaut, ob das Objekt auch ausgewählt ist oder nicht. Ist es angeklickt worden, wird es als ausgewählt markiert. Wenn nicht, dann wird zum nächsten Objekt gegangen.
Der sogenannte “ObjectTransformer” stellt allen voran die Visualisierung des damit verbundenen Nutzerinterfaces dar. Um die Rotation und Positionierung für den User zu vereinfachen, wird jenem Nutzer ein Kreis um das ausgewählte Objekt angezeigt. Wird die Maus entlang des Kreises gedrückt gehalten, so wird das Objekt rotiert. Wird auf das Objekt selbst geklickt, so kann das Objekt positioniert werden.
Die Erkennung des Ziehens entlang des Interfaces wird ebenso über die ObjectPickingMethods realisiert. Zusätzlich werden drei verschiedene Booleans für jedes GameObject verwendet. Der Reihe nach wird erst einmal überprüft wo der Nutzer überhaupt zu ziehen beginnt. Hierfür wird mittels MouseClick-Listener die aktuelle Mausposition erfasst und mittels der ObjectPickingMethods folgenden drei Arealen zugewiesen: im Objekt selbst (der Nutzer versucht das Objekt zu bewegen), am Rechteck um das Objekt drumherum (der Nutzer versucht das Objekt zu skalieren) oder am Kreis um das Objekt herum (der Nutzer versucht das Objekt zu rotieren). Diese drei Fälle dürfen aber niemals gleichzeitig ausgeführt werden, da der Nutzer sonst aus Versehen das Objekt gleichzeitig rotieren und skalieren könnte. Somit setzt der Klick in eine der drei Areale erst einmal nur die bereits angesprochenen Booleans auf true oder false. Wenn das Objekt skaliert werden soll, wird der “rotatable”-Boolean auf true gesetzt und alle anderen auf false. Für die beiden anderen Transformationen passiert das genauso. Danach wird über den MouseDragged-Listener überprüft, welcher Boolean genau true ist und darauf basierend die entsprechende Transformationsmethode verwendet.

4. Simulation

a. Bewegungssimulation

Grundsätzlich unterscheiden wir in unserer Simulation zwischen beweglichen Objekten (z.B. Basketball, Legostein etc.) und unbeweglichen Objekten (z.B. Wand, festgeschraubte Ebene etc.). Beide Objekte haben neben Rotation, Skalierung und Position auch ein Masse-Attribut, welches für die korrekte Kollisionsbehandlung benötigt wird. Ein unbewegliches Objekt kann sich, wie der Name schon sagt, nicht bewegen und besitzt eine unendliche Masse (Masse = 9999999). Es ist also quasi wie angeschraubt. Ein bewegliches Objekt besitzt, anders als das unbewegliche Objekt, die Attribute Beschleunigung und Geschwindigkeit. Die Beschleunigung verändert die Geschwindigkeit,

welche wiederum die Position des Objektes ändert. Ein Durchlauf der Bewegungssimulation sieht wie folgt aus:

```
increaseAcceleration(0, -9.807f); //Gravitation

increaseAcceleration(-velocityX*0.5f , -velocityY*0.5f); //air friction

calculateVelocity();
calculatePosition();

increaseRotation(-velocityX);
resetAcceleration();

((DynamicCollisionContext) collisionContext).checkCollisions();
```

1. Beschleunigung um Gravitation erhöhen
2. Beschleunigung um kleinen Luftwiederstand verringern
3. Geschwindigkeit berechnen
4. Position berechnen
5. Rotation erhöhen
6. Beschleunigung zurücksetzen
7. Kollisionserkennung

In unserer Simulation arbeiten wir mit einem kartesischen Koordinatensystem, wobei der Nullpunkt genau in der Mitte ist und nicht wie üblich oben links in der Ecke, wobei die X- und Y- Achsen nur positive Werte haben. Die Gravitationskraft ist also ein Vektor mit der Y-Komponente -9,807 und der X-Komponente 0. Das heißt, die Gravitationskraft zieht alle Objekte gerade nach unten.
Wir arbeiten dabei mit der Einheit ein Meter pro Pixel.

```
public void increaseAcceleration(float dx,float dy) {
    this.accelerationX+=dx;
    this.accelerationY+=dy;
}
```

increaseAcceleration() erhöht dabei die Beschleunigung um die gegebenen Werte dx und dy.

```
public void calculateVelocity() {
   //  v = v + a * t
   //  acceleration = m/s²
   //  time         = s
   //  velocity     = m/s

   velocityX = velocityX + accelerationX * SimulationControler.getUpdateTimeInSeconds();
   velocityY = velocityY + accelerationY * SimulationControler.getUpdateTimeInSeconds();
}
```

calculateVelocity() berechnet die neue Geschwindigkeit des Objektes durch die Funktion v + a * t mit a = Beschleunigung in Meter pro Sekunde², t = Zeit in Sekunden und v = letzte Geschwindigkeit in Meter pro Sekunde. t ist dabei der Abstand in Sekunden, welcher zwischen der letzten Berechnung und der jetzigen Berechnung liegt. Dieser kann über einen Schieberegler in der Oberfläche vergrößert bzw. verkleinert werden,

um die Simulation zu beschleunigen bzw. zu verlangsamen.

```
public void calculatePosition() {
    //  s = s + v * t + 0,5 * a * t*t
    //  velocity     = m/s
    //  acceleration = m/s²
    //  time         = s
    float xNew = x + velocityX * SimulationControler.getUpdateTimeInSeconds() + 0.5f
            * accelerationX * (float)Math.pow(SimulationControler.getUpdateTimeInSeconds(),2);

    float yNew = y + velocityY * SimulationControler.getUpdateTimeInSeconds() + 0.5f *
            accelerationY * (float)Math.pow(SimulationControler.getUpdateTimeInSeconds(),2);

    setX(xNew);
    setY(yNew);
}
```

calculatePosition() berechnet die neue Position des Objektes anhand der Beschleunigung und der Geschwindigkeit, welche zuvor errechnet wurden. Die neue Position errechnet sich aus s + v * t + 0,5 * a * t * t mit s = letzte Position in Meter, v = Geschwindigkeit in Meter pro Sekunde, a = Beschleunigung in Meter pro Sekunde² und t = Zeit in Sekunden.

```
public void resetAcceleration() {
    this.accelerationX=0;
    this.accelerationY=0;
}
```

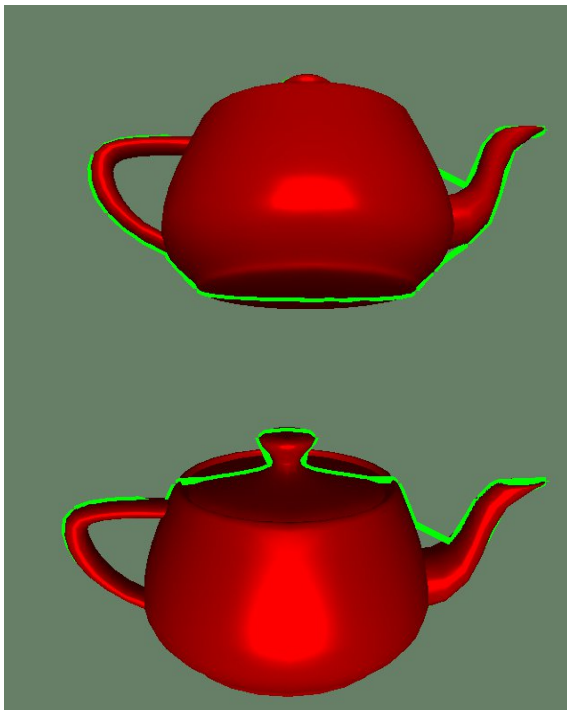
Die resetAcceleration()-Funktion setzt die Beschleunigung nach Berechnung der neuen Geschwindigkeit und Position wieder auf null.

Bei der Rotation der Objekte haben wir etwas getrickst, denn hier nehmen wir einfach die Beschleunigung in X-Richtung, damit es wenigstens so aussieht als ob sich der Ball korrekt dreht.
Nach Ablauf der Bewegungssimulation wird dann noch die Kollision mit anderen Objekten getestet und eine Kollisionsbehandelung vorgenommen, falls nötig.

Wir trennen in unserer Simulation auch das tatsächliche Zeichnen der Objekte vom Ablauf der Simulation. Das heißt, der Ablauf, der gerade geschildert wurde, findet in einem separaten Thread statt. Das Zeichnen der Objekte ist in einem anderen Thread, welcher versucht in einer bestimmten Framerate die Objekte zu zeichnen. Damit können wir sicherstellen, dass die Simulation auch dann zeitlich korrekt abläuft, falls der Computer es nicht schafft die Objekte in angemessener Zeit zu zeichnen. Eine Simulation, bei der ein Ball fünf Sekunden zum Stillstand braucht, wird also immer fünf Sekunden dauern, auch wenn der Computer es nicht schafft die Objekte 60-mal die Sekunde neu zu zeichnen.
Generell braucht der Computer auch beim Zeichnen der Objekte mehr Rechenpower, weil dort aufwändigere Dinge berechnet werden.
Ein weiterer praktischer Vorteil durch diese Aufteilung ist, dass wir die Aufrufrate der Bewegungssimulation ändern könnten, ohne dass dabei die Aufrufrate des Zeichnens der Objekte geändert wird.

b. Kollisionserkennung

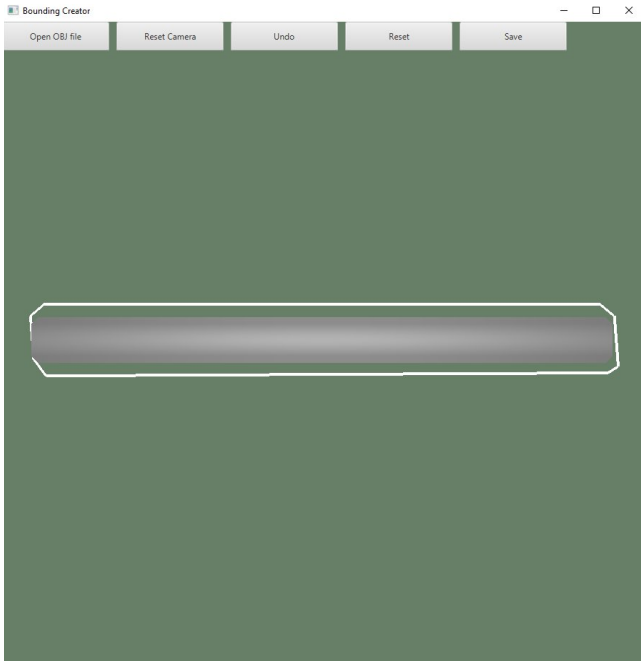
Generell besitzt unsere Simulation nur Polygone(2D) und Kreise um Kollisionen zu erkennen. Wir können Polygon/Polygon-, Kreis/Kreis und Polygon/Kreis-Kollisionen erkennen. Jedes Objekt besitzt ein Bounding. Das ist eine Kombination aus 2D Polygonen und/oder Kreisen, welche den Rahmen des Objektes approximieren sollen, um akkurate Kollisionen zu erkennen. Da wir extern modellierte 3D Objekte nutzen, welche komplexe Formen haben können, nutzen wir dieses System, um korrekte und möglichst realistische Kollisionen zu erkennen.
Wir haben zunächst an einer Methode gearbeitet, um aus 3D Objekten automatisch ein Polygon zu errechnen, welche alle Punkte des Modells einschließt. Diese Methode haben wir jedoch später verworfen, da es erstens ein konkaves Polygon erstellt, welches noch in konvexe Polygone aufgeteilt werden müsste, damit wir mit unserem System eine Kollision erkennen können (wir testen Polygon/Polygon-Kollisionen mit konvexen Polygonen), und zweitens die Polygone bei sehr runden Formen sehr komplex werden und die Kollisionserkennung dann zu lange dauern könnte.



Polygon (in Grün) automatisch durch einen Algorithmus erstellt

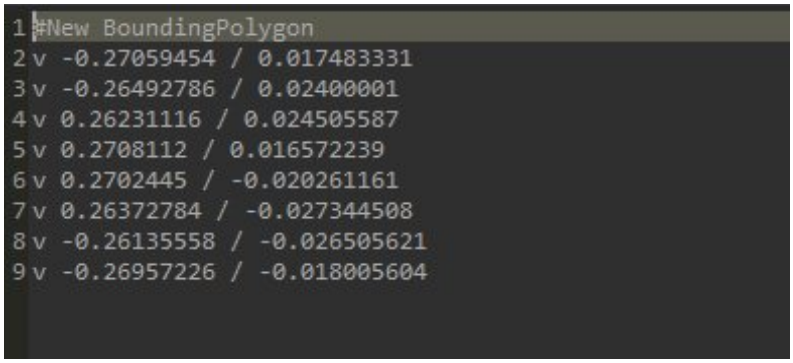
Das geht etwas mehr in die Thematik Geometrieverarbeitung, aber wir haben für die Errechnung dieses Polygons eine Abwandelung des Gift-Wrapping-Algorithmus verwendet (https://en.wikipedia.org/wiki/Gift_wrapping_algorithm) und es besitzt 114 Punkte. Der Algorithmus hat es auch nicht immer geschafft jede Form annehmbar genug zu approximieren. Vor allem bei Formen, die stark in sich gestülpt sind (z.B. ein Eimer), hatte der Algorithmus Probleme die Form richtig zu interpretieren.

Da wir diesen Ansatz verworfen hatten, haben wir uns entschieden, das Polygon selbst anzugeben. Wir haben uns dafür ein kleines separates Programm erstellt, wobei wir per Mausklick Punkte des Polygons angeben können und visuelles Feedback bekommen, wie das Polygon am Ende aussieht und wie gut es die tatsächliche Form approximiert. Hier muss auch aufgepasst werden, dass die erstellten Polygone konvex sind, damit die Kollisionserkennung funktioniert. Wenn es manchmal so aussieht, als ob der Ball in der Luft kurz vor der Ebene abprallt, liegt das daran, dass das Polygon, welches die Form approximieren sollte, ungenau per Hand gesetzt wurde.



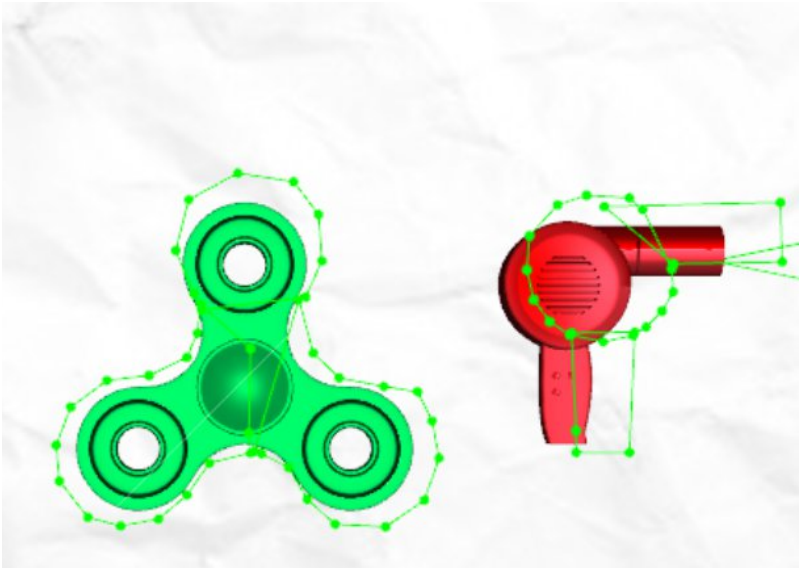
Polygon (in weiß) erstellt “per Hand”

Damit haben wir die Möglichkeit auch mehrere Polygone anzugeben, um Objekte mit einer konkaven Grundform zu approximieren (z.B. einen Eimer).
Die erstellten Polygone werden dann in einer Textdatei exportiert und dann wieder ins Projekt importiert sobald das Objekt erstellt wird.



Beispiel Textdatei, welche die Polygone speichert

“#New BoundingPolygon” leitet ein neues Polygon und “v” einen neuen Punkt mit XY Position ein. Hier besteht das Bounding aus einem Polygon mit acht Eckpunkten. Es ist das Bounding für die oben gezeigte Ebene.



So sehen beispielsweise die Boundings von zwei Objekten aus, mit denen wir die Kollision testen. Die Boundings werden etwas weiter vorne in der z Achse gerendert, weshalb es hier so aussieht, als ob sie nicht so akkurat sind, jedoch approximieren die Boundings die Grundform sehr gut.

Am Ende hat jedes Objekt eine Liste mit Polygonen und eine mit Kreisen, mit denen wir die Kollision testen. Manche Objekte bestehen nur aus einem Polygon, manche aus mehreren Polygonen oder einer Kugel z.B. besteht eine Kugel nur aus einem Kreis und die Polygonliste bleibt leer. Um eine Kollision zwischen zwei Objekten festzustellen, könnten wir alle Polygone und Kreise des ersten Objekts gegen die Polygone und Kreise des zweiten Objektes testen. Dies machen wir für alle Objekte in der Szene und erkennen alle vorhandenen Kollisionen.

Wir testen, wie bereits gesagt, Polygon/Polygon- , Kreis/Kreis- und Polygon/Kreis-Kollision.
Die Kreis/Kreis-Kollision ist sehr einfach. Dort überprüfen wir nur ob die Radien der beiden Kreise größer sind als die Distanz zwischen den beiden Kreisen. Sind die addierten Radien größer, liegt eine Kollision vor. Die Distanz wird über den Satz des Pythagoras ermittelt.

```
public boolean checkCollision(BoundingCircle circle) {
    return circle.getRadius() + radius > Util.getDistance(circle.getX(), circle.getY(), x, y);
}
```

Mit Util.getDistance():

```
public static float getDistance(float x1,float y1,float x2,float y2) {
    return (float)Math.sqrt((x2-x1)*(x2-x1)+(y2-y1)*(y2-y1));
}
```

Die Formel lautet also r1 + r2 > sqrt((x1 - x2) * (x1 - x2) + (y1-y2) * (y1-y2)) mit r1, x1, y1 für Kreis 1 und r2, x2, y2 für Kreis 2.

Ist diese Aussage wahr, liegt eine Kollision vor.

Die Polygon/Kreis-Kollision ist die langsamste Erkennung, denn dort durchlaufen wir alle Geradenstücke des Polygons und überprüfen ob sich eine Linie mit dem Kreis schneidet. Um eine Kollision einer Geraden mit einem Kreis zu ermitteln, wird vorher der nächste Punkt der Geraden und dem Kreis berechnet und dann wird getestet ob dieser innerhalb des Kreises ist.

```
public boolean checkCollision(BoundingPolygon polygon) {
    for (int i = 0; i < polygon.getPoints().length; i++) {
         Vector2f p1 = polygon.getPoints()[i];
         Vector2f p2 = polygon.getPoints()[i + 1 == polygon.getPoints().length ? 0 : i + 1];

         if(checkCollisionLineCircle(p1,p2))
             return true;
    }

    return false;
}
```

Hier durchlaufen wir die Liste aller Punkte des Polygons und suchen uns immer zwei nacheinander folgende Punkte der Liste. Dieses bilden ein Geradensegment, welches in der Methode checkCollisionLineCircle genutzt wird, um zu überprüfen ob dieses Geradenstück mit dem Kreis kollidiert. Dies wird mit allen Geradenstücken des Polygons gemacht und liegt nirgends eine Kollision vor, wird falsch zurückgegeben - es liegt also keine Kollision vor.

```
public boolean checkCollisionLineCircle(Vector2f p1, Vector2f p2) {

    float len = Util.getDistance(p1, p2);

    float dot = ( (x-p1.x)*(p2.x-p1.x) + (y-p1.y)*(p2.y-p1.y) ) / (float)Math.pow(len, 2);

    float closestX = p1.x + (dot * (p2.x- p1.x));
    float closestY = p1.y + (dot * (p2.y- p1.y));

    float d1 = Util.getDistance(closestX, closestY,p1.x,p1.y);
    float d2 = Util.getDistance(closestX, closestY,p2.x,p2.y);

    if (!(d1+d2 >= len && d1+d2 <= len))
      return false;

    float distX = closestX - x;
    float distY = closestY - y;
    float distance = (float)Math.sqrt( (distX*distX) + (distY*distY) );

    if (distance < radius)
        return true;

    return false;
}
```

Hier ist die Methode, welche checkt ob eine Linie, die durch p1 und p2 definiert wurde, mit einem Kreis kollidiert. Zunächst berechnen wir die Länge der Linie und speichern diese. Dann berechnen wir das Skalarprodukt der Linie und dem Kreis, denn damit können wir den nächsten Punkt (closestX, closestY) zwischen der Linie und dem Kreis berechnen. Nun testen wir ob dieser Punkt innerhalb des Kreises ist und geben wahr bzw. falsch zurück. Um zu testen, ob ein Punkt innerhalb eines Kreises ist, berechnen wir den Abstand des Punktes zum Kreis und falls dieser Abstand kleiner ist als der Radius des Kreises, liegt er

innerhalb des Kreises.

Die Polygon/Polygon-Kollision verläuft recht schnell, denn dort nutzen wir das “Separating Axis Theorem”(SAT). Der große Vorteil ist hierbei, dass der Test nicht komplett durchlaufen werden muss, um eine Kollision auszuschließen. Bei der Kreis/Polygon-Kollision müssen wir wirklich durch jedes Geradensegment durchlaufen um eine Kollision auszuschließen, weshalb dieser Test sehr aufwendig sein kann, wenn ein Polygon viele Kanten hat. Das Separating Axis Theorem projiziert nämlich beide Polygone auf eine Ebene und testet, ob dazwischen noch eine Linie gezogen werden könnte, also eine Separating Axis. Dies wird für alle Geradensegmente gemacht und erst, wenn nirgendwo eine Linie zwischen gezogen werden kann, liegt eine Kollision vor. Existiert ein Fall, wobei zwischen den Polygonen doch eine Linie Platz hat, kann der Test sofort abgebrochen werden, welches der Laufzeit zugutekommt. Dieser Test funktioniert jedoch nur mit zwei konvexen Polygonen, deshalb war es so wichtig für die Erstellung der Polygone sicherzustellen, dass diese konvex sind. Wir könnten einen langsameren Algorithmus verwenden, welcher auch für konkave Polygone benutzbar ist, welcher jedoch bei vielen komplexen Polygonen vermutlich zu langsam laufen wird. Man könnte z.B. alle Kanten der beiden Polygone gegeneinander testen und überprüfen, ob sich zwei Geraden berühren, dies würde aber zu extrem vielen Tests führen, da wirklich jedes Geradenstück gegen jedes andere getestet werden muss um eine korrekte Kollisionserkennung zu berechnen.

```
//loop over all points
for (int i = 0; i < polyontmp.getPoints().length; i++){

    //calculate normal
    Vector2f p1 = polyontmp.getPoints()[i];
    Vector2f p2 = polyontmp.getPoints()[i + 1 == polyontmp.getPoints().length ? 0 : i + 1];

    Vector2f normal = new Vector2f(p2.y - p1.y, p1.x - p2.x);

    //project Polygon1 onto axis
    float minA = 0;
    float maxA = 0;
    for(Vector2f p : points){
        float projected = normal.x * p.x + normal.y * p.y;
        if (minA == 0 || projected < minA)
            minA = projected; //find most left Point of Polygon1
        if (maxA == 0 || projected > maxA)
            maxA = projected; //find most right Point of Polygon1
    }

  //project Polygon2 onto axis
    float minB = 0;
    float maxB = 0;
    for(Vector2f p : polygon.getPoints()){
        float projected = normal.x * p.x + normal.y * p.y;
        if (minB == 0 || projected < minB)
            minB = projected; //find most left Point of Polygon2
        if (maxB == 0 || projected > maxB)
            maxB = projected;  //find most right Point of Polygon2
    }

    //check if there is space between left and right most Points of each polygon
    if (maxA < minB || maxB < minA)
        return false;

}
```

Beim ”Separating Axis Theorem” durchlaufen wir alle Geradensegmente beider Polygone, die getestet werden sollen. Dabei werden die beiden Polygone jeweils auf eine Axe projiziert. Durch den Normalenvektor des entsprechendem Geradensegment und dem Skalarprodukt mit dem Punkt und diesem Normalenvektor, wird ein Punkt auf diese Axe projiziert. Macht man dies mit jedem Punkt der jeweiligen Polygone, sind alle Punkte auf eine Axe projiziert und man kann betrachten, ob dazwischen Platz ist.

Alle Tests funktionieren auch unabhängig davon, ob sich das Objekt gedreht hat oder nicht, daher ist vollkommen egal, wie die Objekte ausgerichtet werden: die Kollision wird immer korrekt erkannt.
Objekte können dadurch z.B. wild rotierend durch die Gegend fliegen. Wir haben dadurch komplette Freiheit bei der Gestaltung der Szenen und auch komplette Freiheit bei der Auswahl der Objekte, die wir darstellen wollen - durch die Kollisionserkennung mit unserem Bounding-System.

c. Kollisionsbehandlung

Die Kollisionsbehandlung erfolgt in zwei Schritten.

1. Kollision entfernen
2. Neue Geschwindigkeiten der Objekte berechnen

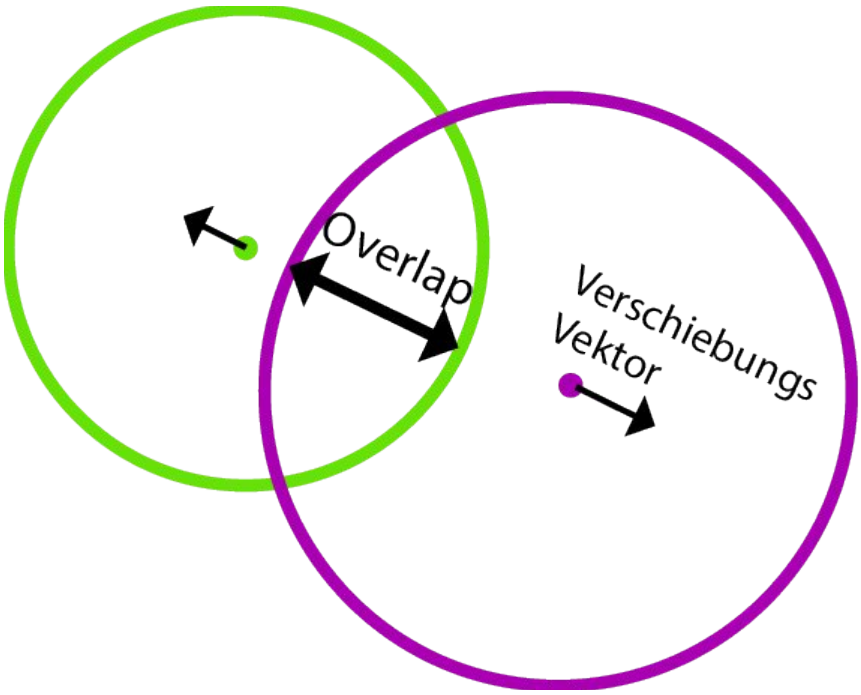
Liegt eine Kollision vor, muss diese zunächst entfernt werden, damit die Kollision im nächsten Durchlauf nicht noch einmal erkannt wird. Dadurch würde ein Zittern der Objekte entstehen, da diese immer wieder kollidieren und ihre Geschwindigkeit umkehren. Ein weiterer Vorteil, die Kollision

aufzuheben, ist, dass Objekte zu keinem Zeitpunkt ineinanderstecken, was die Illusion zerstören könnte, dass die Objekte tatsächlich solide sind. Würde ein sehr schneller Ball auf eine Ebene fliegen und man würde die Simulation sehr langsam abspielen, hätte man einen kurzen Moment indem der Ball tiefer in der Ebene ist ehe er abprallt und zurückfliegt. Entfernt man die Kollision jedoch nach dem Auftreten direkt, würde man dieses Phänomen nicht betrachten können. Wenn mehrere Kollisionen in einer Kette auftreten (z. B. ein Bällebad), könnte es zum Beispiel auch vorkommen, dass Bälle länger ineinanderstecken. Daher sollte man nur Kollisionen überprüfen, wenn sich die Objekte aufeinander zu bewegen. Bei ausgefallen Polygonen können die Objekte auch trotzdem kollidieren, sollten sich die Objekte nicht aufeinander zu bewegen. Deshalb entfernen wir die Kollision direkt nach Auftreten, anstatt lediglich zu überprüfen, ob sich die Objekte noch aufeinander zu bewegen, wie es uns vorgeschlagen wurde. Außerdem würde man ein grundsätzliches Gesetz der Physik ignorieren, wenn zwei Objekte ineinander existieren können.

Um die Kollision zu entfernen, müssen wir wieder trennen zwischen Polygon/Polygon-Kollisionsentfernung, Kreis/Kreis-Kollisionsentfernung und Polygon/Kreis-Kollisionsentfernung.
Die Kreis/Kreis-Kollisionsentfernung ist sehr eindeutig, denn wir können die Länge um welche die beiden Kreise sich überlappen errechnen und beide Kreise um die Hälfte dieser Länge voneinander wegbewegen.

```
float distance = Util.getDistance(object2.getX(), object2.getY(), object1.getX(), object1.getY());

float overlap = 0.5f*(distance- r1 - r2);

float object1X = object1.getX()-overlap*(object1.getX()-object2.getX())/distance;
float object1Y = object1.getY()-overlap*(object1.getY()-object2.getY())/distance;

float object2X = object2.getX()+overlap*(object1.getX()-object2.getX())/distance;
float object2Y = object2.getY()+overlap*(object1.getY()-object2.getY())/distance;

object2.setX(object2X);
object2.setY(object2Y);

object1.setX(object1X);
object1.setY(object1Y);
```

Zuerst errechnen wir die Distanz zwischen beiden Kreisen, denn damit und den beiden Radien der Kreise, können wir die Länge berechnen, um den sich die beiden Kreise überschneiden. Diese Länge wird mit 0.5 multipliziert, da wir nun beide Kreise um diese Distanz voneinander wegbewegen. Die Richtung, um die wir die Beiden Kreise nun verschieben, errechnen wir aus der Distanz zwischen den beiden Objekten, welche wir normieren.



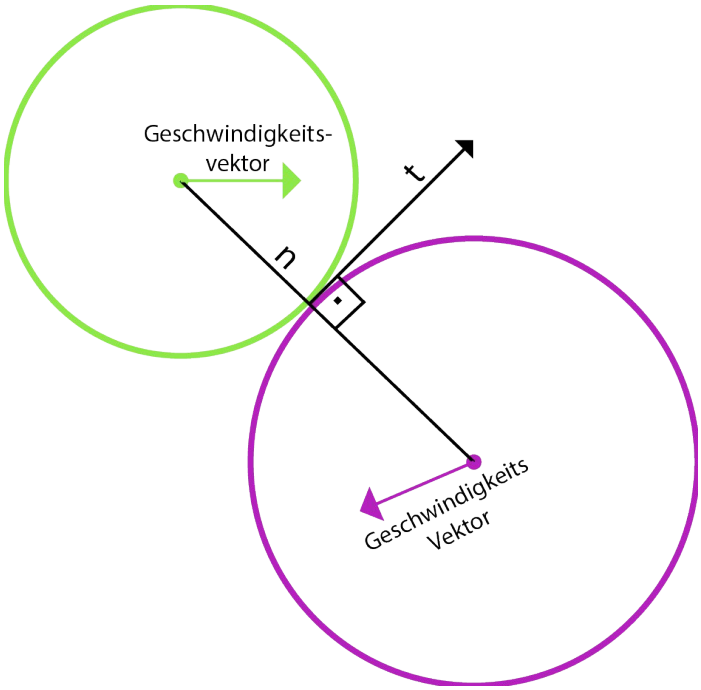
Die Polygon/Polygon-Kollisionsentfernung und Polygon/Kreis-Kollisionsentfernung sind nicht so eindeutig - hier wird wie bei der Kreis/Kreis-Kollisionsentfernung der Verschiebungsvektor aus der Position der beiden Objekte errechnet und normiert, jedoch ist es schwierig die Länge, die sich die beiden Objekte überschneiden, konkret zu errechnen, da wir davon ausgehen, dass wir mit komplexen Polygonen arbeiten, welche sehr kantig sein können. Daher arbeiten wir mit einer Annäherung und gehen von einer festen sehr kleinen Überschneidung der Objekte aus und verschieben beide Objekte um diese Überschneidung. Danach testen wir ob die Objekte immer noch kollidieren. Falls dies der Fall ist, werden erneut beide Objekte um diese Überschneidung verschoben. Dies wird solange gemacht, bis die Objekte nicht mehr kollidieren.

```
do {
    float overlap = 0.1f;

    float object1X = object1.getX() - overlap*nx;
    float object1Y = object1.getY() - overlap*ny;

    float object2X = object2.getX() + overlap*nx;
    float object2Y = object2.getY() + overlap*ny;

    object1.setX(object1X);
    object1.setY(object1Y);

    object2.setX(object2X);
    object2.setY(object2Y);

} while(context1.checkCollisionPolygonCircle2(context2));
```

nx bzw. ny ist hierbei der normierte Verschiebungsvektor. Die Überschneidungslänge ist hier „overlap“ und diese können wir beliebig klein setzen, um den Fehler so klein wie möglich zu halten.

Alle Kollisionsentfernungsmethoden existieren auch für die unbeweglichen Objekte, denn dort darf sich natürlich nur das bewegliche Objekt bei Kollision wegbewegen und nicht das unbewegliche Objekt.

Wurde die Kollision entfernt, müssen noch die neuen Geschwindigkeiten nach dem Aufprall errechnet werden.



Haben wir 2 Bälle gegeben, mit jeweils einem Mittelpunkt und einem Geschwindigkeitsvektor, müssen wir zunächst die Berührnormale t berechnen, denn damit können wir später die neuen Geschwindigkeitsvektoren berechnen. Die Normale t liegt senkrecht auf dem Vektor n, daher können wir über die Mittelpunkte der beiden Kreise zunächst diesen Vektor berechnen und darüber dann den Vektor t.

```
float distance = Util.getDistance(object1.getX(), object1.getY(), object2.getX(), object2.getY());

float nx = (object2.getX()-object1.getX())/distance;
float ny = (object2.getY()-object1.getY())/distance;

float tx = -ny;
float ty = nx;
```

Den Vektor n berechnen wir, wie bereits erwähnt, über die beiden Mittelpunkte der Kreise - hier objekt1 und objekt2 genannt. Diesen Vektor normieren wir bereits, somit ist der Vektor t auch normiert. Um aus dem Vektor n den Vektor t zu errechnen, können wir einfach die x und y Komponenten tauschen und die y Komponente dabei mit -1 multiplizieren.

```
float directionBall1 = object1.getVelocityX()*tx + object1.getVelocityY()*ty;
float directionBall2 = object2.getVelocityX()*tx + object2.getVelocityY()*ty;
```

Nun berechnen wir das Skalarprodukt des Vektors t und des Geschwindigkeitsvektors der jeweiligen Bälle und erhalten so schon

einmal die korrekte Richtung, die die beiden Bälle nach der Kollision haben. Stellen wir uns das Skalarprodukt der Vektoren vor, macht das auch Sinn, denn der grüne Ball würde nach oben rechts abprallen und der violette nach unten links.
Die neue Bewegungsrichtung stimmt schon mal, nun müssen wir noch errechnen wie hoch die Geschwindigkeit der Bälle in die jeweilige Richtung ist, also wie lang die jeweiligen Vektoren sind. Bei Kollision können die Bälle Energie übertragen bzw. verlieren, abhängig von einer Reihe an Faktoren wie Masse oder Materialeigenschaften der Objekte.

$$
\begin{aligned}
v_1 &= \frac{m_1 - m_2}{m_1 + m_2} u_1 + \frac{2m_2}{m_1 + m_2} u_2 \\
v_2 &= \frac{2m_1}{m_1 + m_2} u_1 + \frac{m_2 - m_1}{m_1 + m_2} u_2
\end{aligned}
$$

Mit dieser Formel können wir die neuen Geschwindigkeiten errechnen wobei m1 = Masse in kg des ersten Objektes, m2 = Masse in kg des zweiten Objektes, u1 = Geschwindigkeit in m/s des ersten Objektes und u2 = Geschwindigkeit in m/s des zweiten Objektes.
Ist eines der Objekte bei Kollision ein statisches Objekt, ist die Masse dieses Objektes unendlich und somit kürzt sich die Formel so um, dass sich die Geschwindigkeit des abgeprallten Objektes lediglich umkehrt. Zusammen mit der Richtung und der Länge des Geschwindigkeitsvektors kollidieren die Objekte physikalisch korrekt.

Die Materialeigenschaften, die Einfluss darauf haben wie stark ein Objekt von einem anderen Objekt abprallt, mussten wir im Endprodukt auch vereinfachen. Ursprünglich war geplant mit dem Parameter k als Restitutionskoeffizient zu arbeiten, welcher zwischen null und eins liegt, wobei eins der perfekte elastische Stoß ist, wobei keine kinetische Energie durch Wärme oder ähnliches verloren geht, und null für den perfekten unelastischen Stoß wobei beide Objekte zum Stillstand kommen. Haben wir zwei Objekte mit zwei unterschiedlichen Materialien, besitzt diese Kombination einen Parameter k, welchen man online finden kann, der beschreibt wie viel kinetische Energie bei dem Aufprall verloren geht. In unserem Projekt vereinfachen wir dies und jedes Objekt besitzt einen Parameter k und nicht jede Objektkombination. Besitzen beide Objekte also einen k Wert von ein,s liegt ein Gesamtwert von zwei vor, also ein perfekter elastischer Stoß. Besitzen beide den Parameter null, liegt ein perfekter unelastischer Stoß vor. Der Parameter wird bei Kollision mit der Geschwindigkeit der jeweiligen Objekte multipliziert, um somit einen kinetischen Energieverlust zu simulieren.

```
float k = (object1.getCoefficientOfRestitution()+object2.getCoefficientOfRestitution())/2;

object1.setVelocityX(object1.getVelocityX()*k);
object1.setVelocityY(object1.getVelocityY()*k);
```

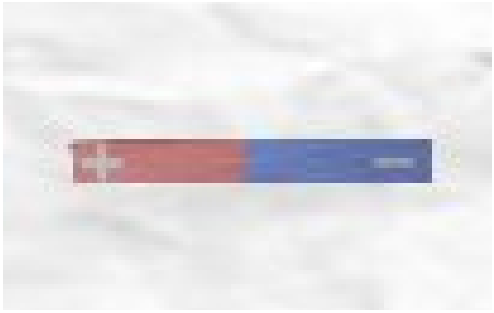
Hier wird der Parameter k aus den beiden Werten der zwei kollidierenden Objekte errechnet und durch zwei geteilt, um einen Wert zwischen null und eins zu erhalten mit dem wir die Geschwindigkeit skalieren können. Es ist keine gute Annäherung an Materialeigenschaften, jedoch kann man damit grob Simulieren, wie kinetische Energie bei Kollision verloren geht.

d. Ausgewählte Elemente

i. Fidget Spinner

Der Fidget Spinner ist ein sich mit konstanter Geschwindigkeit drehendes Objekt, welches sich durch die Formel s + v * t dreht, mit s = aktuelle Rotation in Grad, v = Rotationsgeschwindigkeit in Grad pro Sekunde und t = Zeit in Sekunden. Objekte prallen korrekt vom dem sich drehenden Objekt ab und die Geschwindigkeit der Rotation kann über die Oberfläche gesteuert werden, um z.B. den Spinner auch rückwärts rotieren zu lassen.

Magnet



Der Magnet hat zwei Seiten, eine positiv und eine negativ geladene. In unserem Modell sind alle Kugeln gleich stark negativ geladen, wodurch eine Seite des Magneten sie abstößt und die andere sie zu sich zieht.
Aufgrund der Komplexität von Magneten und dem großen Maßstab unserer Szene (ein Pixel entspricht einem Meter) mussten ein paar Annahmen getroffen werden:
a) Der Stabmagnet ist nach dem Gilbert Model modelliert. Er ist

nicht vollständig geladen, sondern hat an jedem Ende einen einzelnen Punktmagneten. Dieser Punktmagnet stellt ein magnetisches Monopol dar, welches ausschließlich positiv oder negativ geladen ist
b) Beide Punktmagneten sind gleich stark geladen und nur der nähere von beiden wirkt auf das Objekt ein
c) Nur Bälle/ Kugeln werden von dem Magneten angezogen. Diese sind selber Punktmagneten mit einer Ladung um die magnetische Kraft in der Simulation deutlicher zu zeigen

Die Berechnung der Krafteinwirkung des Magneten findet dann über die Formel

$$F = \frac{\mu q_{m1} q_{m2}}{4\pi r^2}$$

(Quelle: https://en.wikipedia.org/wiki/Force_between_magnets, letzter Zugriff: 30.06.2020 13:34)
in Abhängigkeit der Distanz r in Meter, der Ladungen qm1 und qm2 in Ampere/Meter und dem Faktor μ für die magnetische Permeabilität (in der Simulation auf 1 gesetzt) statt.
Das Magnetfeld ist zudem radial um den Pol aufgespannt und wirkt zu jeder Zeit auf die Kugeln ein, wobei die Kraft ab einer gewissen Distanz, abhängig von der Stärke des Magneten, gegen Null läuft.
Die ermittelte Kraft wird mit einem normierten Vektor zum Pol oder vom Pol weg multipliziert um die genaue Richtung der Krafteinwirkung zu bestimmen.

Die Werte für den Magneten wurden aus einer Tabelle für Neodym-Magnete entnommen.
(Quelle: https://www.supermagnete.de/physical-magnet-data, letzter Zugriff: 30.06.2020, 13:37)

ii. Föhn

Der Föhn ist ein abstoßendes Element, das sämtliche beweglichen Objekte in einem bestimmten Bereich mit einer von der Entfernung zum Objekt abhängenden Beschleunigung von sich stößt.

In der Öffnung des Föhns liegt die Spitze eines gleichschenkligen Dreiecks, dass den Einflussbereich des durch den Trockner generierten Windes darstellt. Mittels Polygon/Punkt-Kollisionsabfrage wird dann überprüft, ob sich ein bewegliches Objekt innerhalb des Dreiecks befindet. Ist dem so, so wirkt eine Kraft auf den Ball. Ist das Objekt innerhalb des Dreiecks und hat die minimale Distanz zur Öffnung des Föhns, so wird die maximale Beschleunigung angewandt. Ist das Objekt zwar noch innerhalb des Dreiecks, aber hat die maximale Distanz zum Objekt, so wird keine Beschleunigung mehr angewandt. Ist das Objekt irgendwo innerhalb des Dreiecks, so wird die passende Beschleunigung anhand der Distanz zum Objekt errechnet.
Die Beschleunigung, die zwischen der Maximalen und keiner Beschleunigung errechnet wird, wird in der sogenannten forceFunction() bestimmt. Diese sieht wie folgt aus:

```
private float forceFunction(float distance) {
    return (dmax - distance) * amax;
}
```

iii. Portal

Das Portal ist ein kleines Gimmick, welches ein hereinfliegendes Objekt zu einem anderen Portal teleportiert. Dabei behält das Objekt seine Geschwindigkeit und Beschleunigung. Lediglich die Position wird verändert.

5. Übersicht / Verteilung der Arbeitspakete

Arbeitspakete	**Bearbeitet von**
Spiele Logik, UI Programmierung (Menüs, Soundeffekte, Canvas), Bewegungssimulation, Kollisionserkennung, 3D Rendering, Fidget Spinner, Bälle, Eimer	Simon Weck
3D Kugel, Objekt Transformation UI, Partikel, Föhn, Isomatte	Benjamin Jäger
UI Design UI Programmierung (SideBar), Kollisionsbehandlung, Portal, Umzugskarton	Paul Amelingmeyer
UI Konzept Design, UI Programmierung (TabPane), Drag & Drop, Magnet	Patryk Watola

6. Quellen

http://www.jeffreythompson.org/collision-detection/index.php
Kollisionserkennung
https://en.wikipedia.org/wiki/Elastic_collision
Elastische Kollision
https://gamedevelopment.tutsplus.com/tutorials/collision-detection-using-the-separating-axis-theorem--gamedev-169
Separating Axis Theorem (SAT)
https://free3d.com/3d-model/hairdryer-v1--777578.html
Föhn 3D-Modell
“Crazy Machines” FAKT Software 2004
Sound Effekte
Texturen
https://www.turbosquid.com/de/3d-models/free-obj-mode-bucket/949871
Eimer 3D-Modell
https://de.vecteezy.com/vektorkunst/135533-basketball-textur
Basketball Textur
https://www.kingsloot.de/media/catalog/product/cache/1/image/9df78eab33525d08d6e5fb8d27136e95/k/i/kingsglass_rick-morty-portal_fuerlootbook_preview.png

Portal Textur
https://free3d.com/de/3d-model/fidget-spinner-71204.html
Fidget Spinner 3D-Modell
https://hipwallpaper.com/view/lDHdNe
Background2
https://image.freepik.com/free-photo/view-white-crumpled-paper_1194-7544.jpg
Background3
https://hipwallpaper.com/view/ESded
Background1