**Inhaltsverzeichnis**

[Abkürzungsverzeichnis II](#_Toc500584948)

[Abbildungsverzeichnis III](#_Toc500584949)

[1 Motivation 1](#_Toc500584950)

[2 Theoretische Grundlagen 2](#_Toc500584951)

[2.1 Beispiel 2](#_Toc500584952)

[2.2 Beispiel 2](#_Toc500584953)

[2.3 Beispiel 2](#_Toc500584954)

[2.3.1 Beispiel 2](#_Toc500584955)

[2.4 Beispiel 3](#_Toc500584956)

[3 Umsetzung 3](#_Toc500584957)

[4 Spielanleitung 3](#_Toc500584958)

[5 Ausblick 3](#_Toc500584959)

[6 Fazit 3](#_Toc500584960)

[7 Literaturverzeichnis 4](#_Toc500584961)

# Theoretische Grundlage

Unserem Spiel liegen folgende physikalischen Vorgänge zu Grunde

* Freier Fall
* Schiefe Ebene

## Freier Fall

Zu Beginn des Spiels fällt der Ball aus der Höhe 30 auf die Ebene. Der Fall wird durch folgenden Quellcode herbeigeführt: shape.position.set( 0, 30, -15);

Die Position des Balles wird nach folgendem Muster ( x- , y- , z-Koordinate) in einem Vektor angegeben.

Bei dem freien Fall fällt der Ball ohne Berücksichtigung des Luftwiderstandes auf die Ebene. Daher gilt die Formel für die gleichmäßige beschleunigte Bewegung. Der Ball wird in der Realität durch den Ortsfaktor, der von dem jeweiligen Ort abhängig ist, beschleunigt. Der mittlere Wert an der Erdoberfläche beträgt circa 9,81 (). In unserem Programm wird folgendermaßen die Erdanziehung eingebracht:

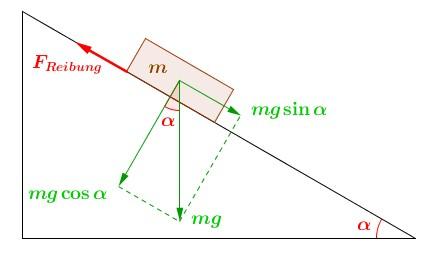
scene.setGravity(**new** THREE.Vector3( 0, -250, 0 ));

Durch den Befehl „setGravity“ lassen sich die Stärke und Richtung verschieden wirkender Kräfte einstellen. In diesem Fall wirkt eine Kraft mit der Särke 250 in negativer Y-Richtung. Wir haben mit 250 einen relativ hohen Faktor gewählt, damit die Kugel schnell auf der Ebene landet und man sofort mit dem Spiel beginnen kann.

## Schiefe Ebene mit Reibung

Nachdem der Ball auf der Ebene aufgekommen ist, gelten die Gesetze der schiefen Ebene mit Reibung. Der Ball wird aufgrund seiner Gewichtskraft () entlang der schiefen Ebene hangabwärts beschleunigt. Die Gewichtskraft ist die Resultierende aus der Normalkraft () und der Hangabtriebskraft ().

* Die Gewichtskraft ist die Masse der Kugel multipliziert mit dem Ortsfaktor, der die Kugel beschleunigt.
* Die Normalkraft wirkt senkrecht zur schiefen Ebene und drückt den Ball auf den Boden. Die Normalkraft würde bei einem weichen Untergrund den Ball zum Einsinken bringen, in unserem Fall ist der Boden jedoch fest.
* Die Hangabtriebskraft wirkt parallel zur schiefen Ebene und beschleunigt den Ball. Da die Gravitationskraft so hoch eingestellt ist, ist folglich auch die Hangantriebskraft sehr groß. Daraus resultiert, dass der Ball schneller auf der Ebene beschleunigt.
* Die Reibungskraft wirkt entgegengesetzt zur Hangabtriebskraft. µ ist hierbei der Haftreibungskoeffizient und hängt von dem Stoff und der Oberflächenbeschaffenheit der Objekte ab.



Der Ball kommt somit erst in Bewegung, wenn die Hangabtriebskraft groß genug ist um die zwischen dem Ball und der Ebene wirkende Reibung zu überwinden.

Nach Umformung erhält man:

Wenn nun der Haftreibungskoeffizient für das Material bekannt ist, kann man den Winkel ab wann das Objekt anfängt zu rutschen ausrechnen.

Die Reibung der unterschiedlichen Objekte (walls, ground, borders) konnte man wie man unten sieht einstellen:

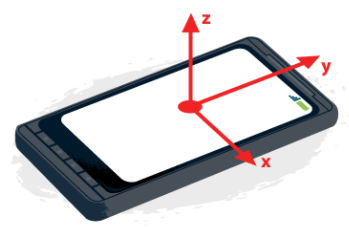
ground\_material = Physijs.createMaterial(**new** THREE.MeshLambertMaterial({ map: loader.load( 'images/wood.jpg' ) }),0.1,0.1);

Die Variabel „ground\_material“ kriegt mehrere Werte zugeschrieben. Es wird ein Bild hochgeladen, das als Oberflächenfarbe des Objektes dient und anschließend wird die Reibung und Abprallstärke festgelegt. Die Reibung und Abprallstärke kann zwischen 0 und 1 liegen. In diesem Fall sind beide Werte gering.

Außerdem kann man noch folgenden Zusammenhang aus der Ähnlichkeit des Kraftdreiecks und der Dreiecks der schiefen Ebene herstellen:

Der Zusammenhang besagt, je kürzer (<->länger) die schiefe Ebene ist, desto höher (niedriger) ist die wirkende Hangabtriebskraft. Dieser Effekt hat auch Auswirkung auf die Spielweise auf unterschiedlich großen Displaygrößen. So wird die Kugel auf einem kleinen Display schneller beschleunigt, als auf einem Großen.

## Deviceorientation:

Die Deviceorientation wird im Programm ausgelöst, wenn von einem Orientierungssensor Daten über die aktuelle Ausrichtung des Tablets oder Smartphones vorhanden sind. Die Daten über die aktuelle Ausrichtung werden mit den Erdkoordinaten verglichen und daraus werden dann Rückschlüsse gezogen. Die Daten werden von einem Magnetometer im Gerät regeneriert. Falls keine Daten vorhanden sind, wird zur Steuerung die Tastatur benutzt.

Beim deviceorientation ist der Koordinatenursprung in der Mitte des Gerätes gelegt. Die Ausrichtung der Achsen ist links auf der Abbildung zu sehen.. Die Rotationsdaten werden im Euler-Winkel zurückgegeben.

Es gibt drei Werte beim deviceorientation: alpha, beta und gamma. Alpha ist die Rotation um die z-Achse. Beta die Rotation um die x-Achse und gamma um die y-Achse. Alpha wird in unserem Spiel nicht gebraucht, weil zur Balanzierung der Kugel auf der Ebene nur die Rotation um die y- und x-Achse entscheident ist. Das sieht dann wie folgt aus:

**var** maxRotation = 0.2;

window.ondeviceorientation = **function**(event) {

ground.rotation.set(

Math.**min**(Math.**max**(Math.**round**(

-event.beta) / 90, -maxRotation), maxRotation),

ground.rotation.y,

Math.**min**(Math.**max**(Math.**round**(

event.gamma) / 90, -maxRotation), maxRotation)

);

ground.\_\_dirtyRotation = true;

};

Da die geolocation API die Werte nur in Grad zurückgibt, während Three.js die in Prozent braucht, dividieren wir um 90. Außerdem wird der Wert im Anschluss noch auf maximal 0.2 bzw. -0.2 begrenzt, damit man das Board nicht weiter als um 20 % kippen kann.

# Umsetzung

## Ansätze

Unser erster Ansatz zur Programmierung des Spiels orientierte sich an der Vorlesung. In den ersten Vorlesungen hatten wir gelernt, wie man mithilfe von three.js 3D-Graphiken erstellt. Diese Kenntnisse haben wir angewandt und schließlich eine Ebene und einen Ball erstellt. Anschließend brachten wir die Rotation der Ebene nach einer Tastaturbelegung mit ein. Wir verbanden den Ball mit der Ebene, sodass bei der Rotation der Ebene ebenfalls der Ball in die Richtung wanderte (siehe Abbildung unten).

//Platte wird um 2 Grad gedreht

**var** alpha = 0.03488888888888888888888888888888888888888888888888889;

**if** (code == 88) cube.rotation.y = cube.rotation.y + alpha;

**if** (code == 89) cube.rotation.y = cube.rotation.y - alpha;

**if** (code == 65) cube.rotation.x = cube.rotation.x + alpha;

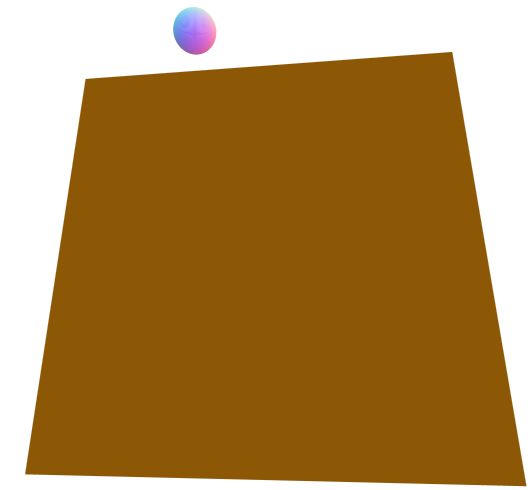
**if** (code == 83) cube.rotation.x = cube.rotation.x - alpha;

**if** (code == 65) {ny\_g = - 0.1 \* g\_const \* (t-t0) \* (t-t0)

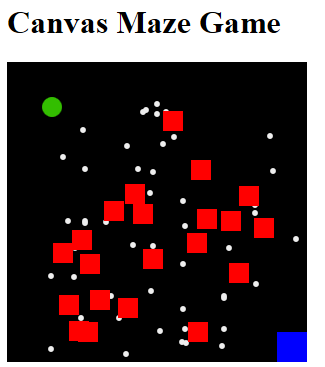
spawnY = (spawnY+ny\_g)

ball.position.set(spawnX, spawnY, 0);

Bei jedem Tastendruck wurde die Ebene um 2 Grad rotiert durch die Addition der Konstanten alpha. Zusätzlich wurde auch der Ball um eine gewisse Beschleunigung verschoben, welches die Hangabtriebskraft (m\*g\*sin(a)) darstellen sollte.

Bei diesem Ansatz sind jedoch viele Probleme aufgetreten. Der Ball ist nicht auf der Ebene gerollt, sondern nur „versetzt“. Es gelang uns auch nicht die perspektivtreue Darstellung des Balles und wir hatten Schwieirigkeiten mit der Verbindung des Balles mit der Ebene, wie man auch sehr gut in der Abbildung rechts erkennen kann.

Da wir bei diesem Ansatz keinen Fortschritt erzielten, versuchten wir unser Spiel zu vereinfachen und es auf 2D-Grafiken zu beschränken.

Bei der Recherche fanden wir im Internet ein Tutorial wie man mithilfe von HTML 5 Canvas ein Space Game kreieren kann. Um ein besseres Verständnis mit den Umgang von HTML5 und Three.js zu kriegen, probierten wir es aus und änderten es zu einem Maze Game um.(Quelle:<https://msdn.microsoft.com/en-us/library/gg589490(v=vs.85).aspx>) Wir lernten viel über die Nutzung von Canvas, Kollisionserkennung durch Farbunterschiede und generell die Vorgehensweise und den Umgang mit Variablen beim Programmieren.

Unser dritter Ansatz beruhte schließlich auf Physi.js, welches ich im Folgenden Kapitel näher beleuchten will.

## Physi.js

Am Ende ist unsere Umsetzung des Spieles auf Physi.js hinausgelaufen, weil sich damit sehr komplexe Sachverhalte im Vergleich zu unseren anderen Ansätzen sehr einfach umsetzen lassen können.

Physi.js ist ein Plugin für das 3D-Framework Threej.s, welches wiederum auf das JavaScript-Port ammo.js von Bullet beruht. Die durch Physi.js erzeugten 3D-Grafiken können ohne zusätzlichen Erweiterungen dargestellt werden, weil es auf WebGL (Web Graphics Library) beruht. Diese Physik-Engine ist eine große Hilfe bei der Simulation von vielen physikalischen Vorgängen. In unserem Fall übernimmt die Physics Library die Simulation des starren Körpers, der Kollisionserkennung, des Freien Falls und der resultierenden Kräfte aus der schiefen Ebene (siehe theoretische Grundlage, reibung etc…..)

## GitHub

Um unser Spiel auch auf mobilen Geräten spielen zu können, haben wir uns ein repository bei Github angelegt. Github war eine große Hilfe beim programmieren des Spieles, weil man sehr gut den aktuellen Quellcode mit älteren Vergleichen kann. Das ist sehr effektiv zur Fehlerfindung. Der Hauptvorteil bestand aber, dass wir unser Spiel über GitHub hosten konnten und somit jederzeit von jedem mobilen Gerät Zugriff auf unser Spiel hatten. Durch folgenden Link: [https://beamerv30.github.io/MazeGame/.](https://beamerv30.github.io/MazeGame/)

## Quellcode

In diesem Absatz werde ich auf den Quellcode eingehen und diesen anhand von einigen exemplarische Beispiele erläutern.

**if** (ball != null && ball.position.distanceTo(hole2.position) < 4) {

**alert**('You lose');

scene.remove(ball);

ball = null;

window.location.reload(true);

}

document.addEventListener('keydown',

**function**( push ) {

**switch**( push.keyCode ) {

//key confiuration

//rotation

**case** 65: //A

ground.rotation.set(Math.**min**(ground.rotation.x + 0.01, maxRotation), ground.rotation.y, ground.rotation.z);

ground.\_\_dirtyRotation = true;

// You may also want to cancel the object's velocity

ground.setLinearVelocity(**new** THREE.Vector3(0, 0, 0));

ground.setAngularVelocity(**new** THREE.Vector3(0, 0, 0));

**break**;

## Herausforderungen

Obwohl Physi.js perfekt zu unserem Beispiel passte, offenbarten sich viele Schwierigkeiten mit dieser Library. Leider gibt es keine umfassende Dokumentation und außerdem sind viele Befehle die wir während unserer Recherche fanden nicht brauchbar, weil sie bereichts veraltet waren. Zudem beruht Physi.js vielerseits auf Three.js und wir hatten damit keinerlei Erfahrung. Da wir nur mit Matlab Erfahrung hatten stellten sich zu Beginn einige Probleme beim programmieren heraus. Zum Beispiel muss man bei JavaScript nach einer if-Abfrage eine geschweifte Klammer setzen bei Matlab jedoch nicht. Solche anfänglichen Schwierigkeiten konnten jedoch nach einigem Ausprobieren schnell gelöst werden. Nichtsdestotrotz sorgten sie zu Anfang zu vielen unerklärlichen Fehlermeldungen.

Eine weitere Herausforderung war, dass oft der Ball, wenn er in der Ecke der Ebene lag, nach der Betätigung der Rotation durch die Ebene durchfiel. Das Problem war, dass die Rotation der Ebene nicht fließend, sonder ruckartig umgesetzt wurde. So wurde das Ende der Ebene um einen Abstand der größer als der Durchmesser des Balles war, erhoben / erniedrigt. Um dieses Problem zu lösen, gibt es mehrere Möglichkeiten. Man kann die Größe bzw. den Durchmesser des Balles erhöhen, die Rotation kleiner machen oder die Ebene tiefersetzen und dann die Höhe der Ebene vergrößern. Wir hatten uns dazu entschieden sowohl die Rotation zu verkleinern als auch die Höhe der Ebene zu vergrößern. Somit wurde das Problem auf dem Computer behoben, jedoch gibt es das Problem teilweise noch bei der Benutzung des Spieles auf Smartphones und Tablets. Das liegt daran, dass die Rotation auf Geräten nicht durch Tastentätigung durchgeführt wird, sondern durch Kippen des Gerätes.

Device.orientation

# Spielanleitung

# Ausblick

# Fazit

# Literaturverzeichnis