HOCHSCHULE FURTWANGEN UNIVERSITY HFU

Mad Maze

FUDGE

FURTWANGEN UNIVERSITY DIDACTIC GAME EDITOR

Studienarbeit

Betreuer: Prof. Jirka Dell´Oro-Friedl

Schmidberger Valentin, 263249

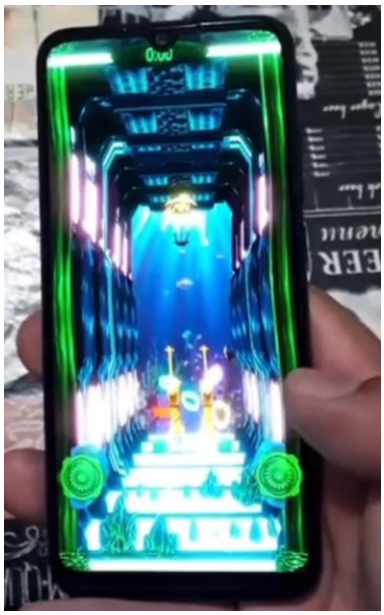
SCAN ME

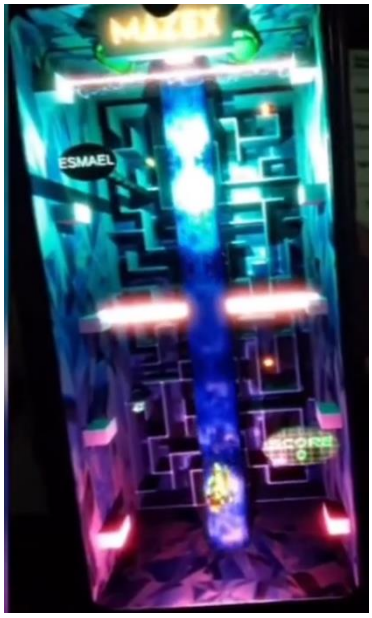
https://github.com/vale-sch/MadMaze

Gliederung

1. Einleitung

Die Idee des Projekts hatte ihren Ursprung in einer Unity Gruppe. In meinen Semesterferien schaute ich ab und zu in die News der Gruppe und sah immer wieder ein interessantes Spielkonzept. Es handelte sich um einfache Prototypen. Das besondere dabei war aber die konzeptionelle Umsetzung des Gyroskopen Sensors (Gyroskope oder Gyros sind Geräte, die Drehbewegungen messen oder beibehalten) innerhalb des Spiels. Sie nutzten den Sensor um pseudo Tiefe zu simulieren, welcher erstaunlich echte Ergebnisse lieferte. Mir gefielen die optischen Ergebnisse und ich wurde motiviert selbst ein Prototyp auf die Beine zu stellen. Um das Ganze mit meinem Studium zu verbinden, fragte ich Jirka ob man so ein Projekt auch als Semesterarbeit umsetzen kann. Das wurde bestätigt und Jirka hatte abgesehen von der Vorstellung, dass das Projekt mit der hochschulinternen Game Engine FUDGE umgesetzt werden sollte, noch so einige andere Ideen auf Lager gehabt.



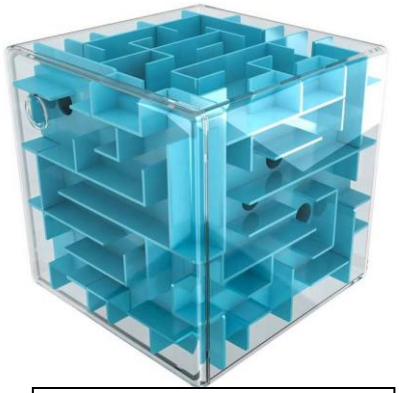


2. Konkretisierung des Konzepts

Durch Gespräche zwischen Jirka und mir und unseren Vorstellungen des Spiels, konkretisierte sich das Konzept immer weiter. Fest stand, es soll eine Art Labyrinth Spiel am Handy werden. Als Basis und zur Vorstellung nahmen wir die alten Holz Labyrinth Spiele an deren Seite man mit einer Art Knauf, die Holz Ebene in horizontaler oder vertikaler schwenken kann. Auf der Ebene befindet sich eine Metall Kugel, die man durch das Labyrinth balancieren muss.

Quelle: Amazon

Das große Alleinstellungsmerkmal des Konzepts war allerdings die Idee, das Labyrinth nicht nur auf einer Ebene stattfinden zu lassen, sondern in jeder denkbaren Ebene. Zur Vereinfachung kann man sich eine Art hohlen Kubus vorstellen in dem sich verschiedene Ebenen befinden die alle bespielbar sein sollen. Das anzustrebende Ziel war, dass der Nutzer das Handy nicht nur vor sich halten muss wenn man auf andere Ebenen gelangen will, vielmehr soll der Nutzer das Endgerät auf die Seite stellen, über sich halten, verkehrt herum halten etc., um auf die anderen Ebenen zu wechseln.



Quelle: knobelbox.com

3. Erste Testversuche

Da das Projekt in FUDGE entstehen sollte, ging es in erster Linie darum, die entsprechende Schnittstelle in WebGL zu finden, um Zugriff auf den Gyroskopen Sensor zu bekommen.

3.1 HTTPS Zertifikat

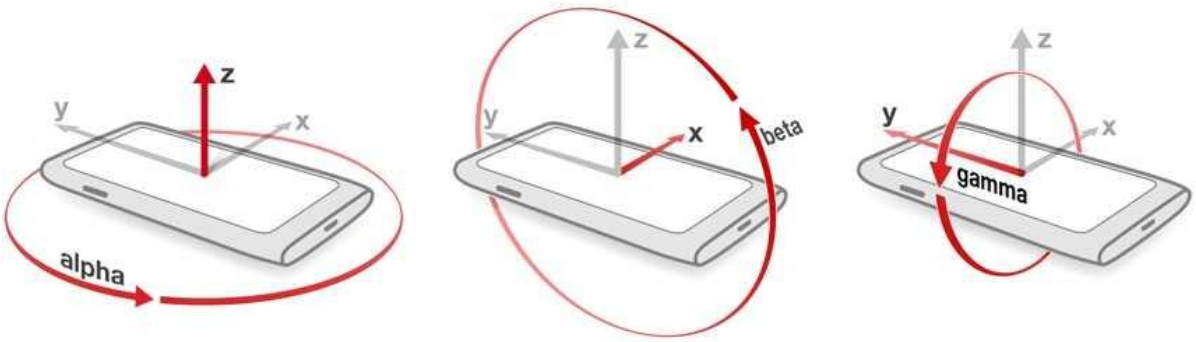
Eine weitere Hürde die sich direkt vor dem Start eröffnete war, dass man eine HTTPS zertifizierte Website als Grundbaustein braucht, um auf geräteinterne Technologien zugreifen zu können. Da ich nicht jeden minimalen Progress auf Github Pages deployen wollte, musste ich die Live-Server Extension von Visual Studio Code mit einem HTTPS Zertifikat austatten. Dazu gibt es eine sehr ausführliche Anleitung im Internet die mithilfe der Open-SSH Software die Vorgehensweiße Schritt für Schritt erklärt(https://webisora.com/how-to-enable-https-on-live-server-visual-studio-code/). Die Konfigurationsdatei *settings.json* sieht dann wie folgt aus:

```
{
    "liveServer.settings.https": {
    "enable": true, //set it true to enable the feature.
    "cert": "C:\\Users\\valiv\\servercertificate\\server.crt", //full path of the certificate
    "key": "C:\\Users\\valiv\\servercertificate\\server.key", //full path of the private key
    "passphrase": "12345"
    },
    "liveServer.settings.port": 5500
    }
```

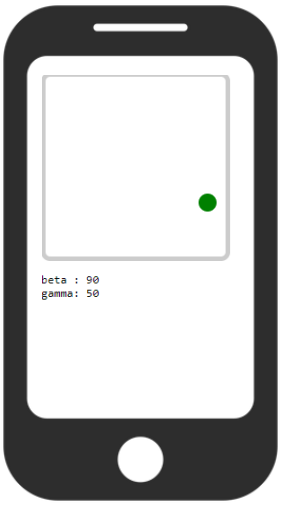
3.2 WebGL Demo

Für das Abgreifen der Gyro-Sensor Werte, gibt es eine Anleitung mit einer kleinen Demo von der *Developer Mozilla* Website, die das Ganze sehr ausführlich erklärt (https://developer.mozilla.org/enUS/docs/Web/Events/Detecting_device_orientation). Die aktuelle Orientierung lässt sich über das Event

„DeviceOrientationEvent“ abfangen, innerhalb des Events kann man dann mit event.*“alpha“ / “beta“ / “gamma“* , jede erdenkliche Orientierung abfragen.



Mithilfe von genanntem Event und der Anleitung baute ich dann die erste kleine Demo für mein Handy. Damit konnte ich dann überprüfen, wie präzise man mit dem Gyro Sensor arbeiten kann und ob das Ergebnis mit meinen Erwartungen übereinstimmt. Allerdings ging es dabei erstmal nur um die *beta* (x-Achse) und *gamma* (y-Achse) Werte.



3.3 FUDGE Demo

Als die Implementierung des Orientation Events in WebGL problemlos funktionierte, wagte ich mich im nächsten Schritt (wieder) an die Game Engine FUDGE. Das Einbinden des Events in FUDGE funktionierte, wie erwartet, reibungslos. Schlussendlich ist das Orientation Event ein einfaches Javascript Event und weil die Game Engine eine webbasierte (WebGL) Engine ist, funktionierte das Ganze out of the box.

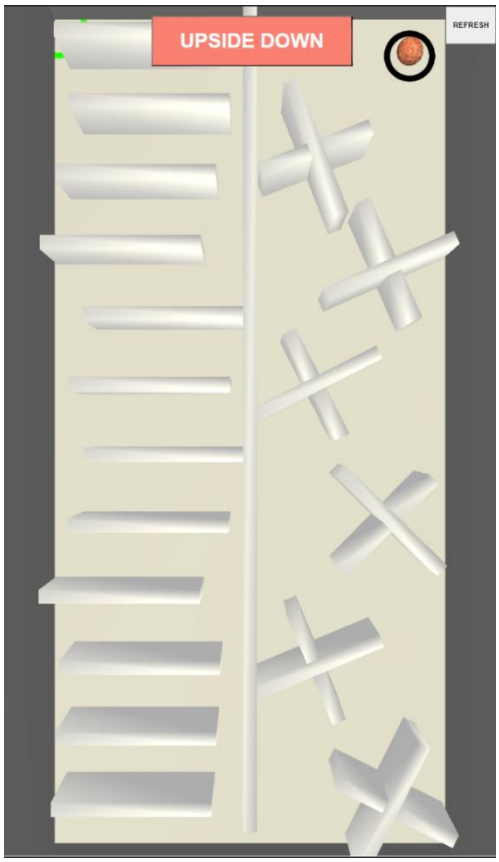
```
window.addEventListener("deviceorientation", handleOrientation, true);
```

Dann ging es noch darum festzustellen, welches mobile Endgerät sich mit der Demo verbindet (Android, iOS, Windows Phone). Über den Node Packet Manager habe ich das Package libdom in das Projekt eingefügt. Durch das gegebene Interface konnte man dann abfragen welches Endgerät sich mit der Demo verbindet. Als Ergebnis erhielt man einen String und über eine Switch-Case Abfrage, konnte ich dann für das jeweilige Betriebsystem das entsprechende Orientation Event abfeuern.

Nach diesem Schritt baute ich zunächst eine kleine Testumgebung auf. Die Physik Engine von FUDGE wurde wieder ausgepackt und ich baute mir eine Sphere mit Collider und Rigidbody. Dazu kamen unterschiedliche Hindernisse mit Collidern. Danach übertrug ich zunächst *beta* (vertikale Achse des Handys) auf die z-Beschleunigung meines Rigidbodys und *gamma* (horizontale Achse des Handys) auf die x-Beschleunigung.

```
rgdbdyBall.applyForce(new f.Vector3(-_event.gamma, 0, -_event.beta));
```

Nachdem der Ball (Sphere) intuitiv gelenkt werden konnte, wurde das Spielfeld noch lebendig gemacht, indem ich den Hindernissen entweder eine zufällige, drehende Rotation (Kreuze) oder Bewegung (Wände) mitgab. Das Ziel des Prototyps war es nun, die Kugel von der rechten Seite oben im Bildschirm, zu der linken oberen Ecke zu manövrieren. Oben links angekommen, musste man dann den „Upside Down“-Button betätigen und das Handy über Kopf halten. Dann kam die Kugel auf einen zu



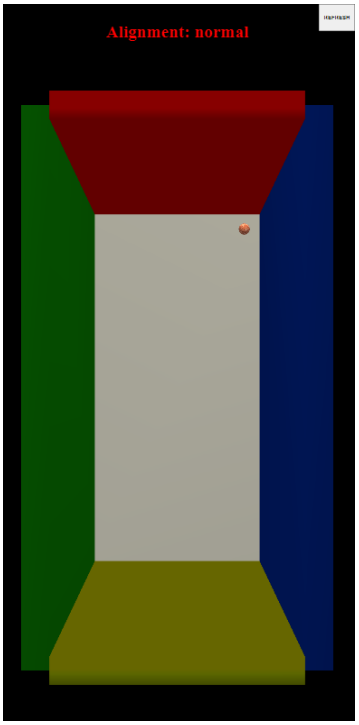
geflogen und fiel dann in einen unsichtbaren Collider und das Spiel gab einen „Du hast gewonnen“ – Text aus.

```
if (upSideDownBool)
  if (Math.abs(_event.beta - oldYAcceleration) > 10)
    rgdbdyBall.applyForce(new f.Vector3(0, _event.beta, 0));
```

4. Testumgebung

Als Spielerei und um wieder mehr in die Arbeitsweise von FUDGE einzutauchen war der oben gezeigte Prototyp ganz nett, aber er setzte nicht das vorhergesehene Konzept um. Vielmehr sollte es für mich in der darauf folgenden Zeit darum gehen, eine Mechanik aufzustellen, die es ermöglichte den Ball frei im Raum zu bewegen. Also setzte ich einen einfachen Testraum auf: dieser bestand aus einem hohlen Rechteck mit 6 Collidern, sodass die Kugel nicht herausfallen konnte. Dann begann für mich das experimentieren mit dem Gyro Sensor.

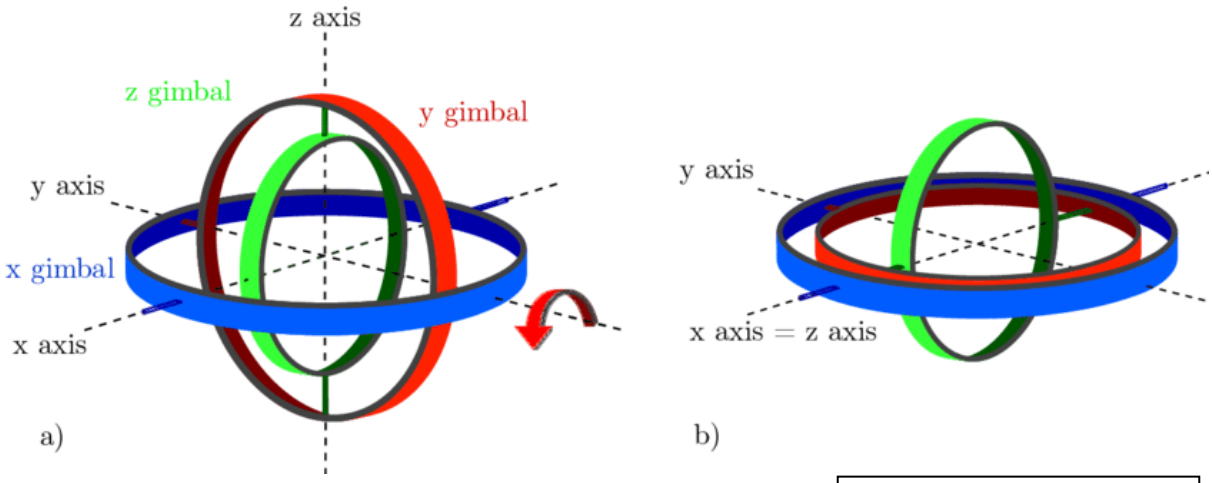
4.1 Umsetzung der grundlegenden Mechanik



Als erstes suchte ich nach Grenzen in den *alpha*, *beta* und *gamma* Werten, um festzulegen ab wann das Gerät gedreht, auf dem Kopf und über dem Kopf steht. Als Hilfestellung habe ich mir einen String der aktuellen Orientierung als HTML-Paragraph Element ausgegeben. Nachdem alle möglichen Stellungen des Handys abgearbeitet waren, ging es im nächsten Schritt darum, innerhalb den Orientierungen die passenden alpha, beta, gamma Werte rauszupicken, um sie dann logisch auf den Rigidbody anzuwenden.

Wie zu erwarten war, war das die größte Baustelle an der ich die nächsten Wochen zu arbeiten hatte. Eine der größten Hürden entstand, wenn die z-Achse des Handys (alpha) rotiert wurde. Denn dann kam es zu dem berühmt berüchtigten Problem des Gimbal Locks.

4.2 Gimbal Lock



Quelle: researchgate.net

Allgemein: ein Gimbal Lock entsteht, wenn 2 der 3 Koordinatenachsen parallel zueinander stehen (in Abbildung: die x und z-Achse). Vorallem kommt dieses Problem vor, wenn man mit Euler Winkeln die Orientierung eines Objektes beschreibt. In der Computergrafik werden Euler Winkel primär als Berechnungsform für die Orientierung eines Objektes genutzt. Ein möglicher Ansatz dieses Problem zu umgehen ist die Nutzung von Quaternionen.

Allerdings stellten Quaternionen für mich keine wirkliche Alternative dar, da Quaternionen einerseits noch nicht, beziehungsweise nicht vollständig in FUDGE implementiert sind und andererseits meinen mathematischen Horizont übersteigen. Also musste ich das Problem mit einem anderen Ansatz lösen. Schlussendlich lief es auf „Try and Error“ heraus: ich überprüfte bei den

bekannten „Problem-Orientierungen“ die Grenzwerte für den Gimbel Lock und fing diese Momente mit einer if-Abfrage ab. Die Gimbel Lock „Momente“ ließen sich recht einfach erkennen, da oft einer der *alpha, beta, gamma* Werte plötzlich von konstanten 20-30 Grad auf über 100-180 Grad schossen. Im Code sah das Ganze dann zum Beispiel für die rechts Orientierung des Handys wie folgt aus:

```
case (Alignment.RIGHTSIDE):
    if (location.isActive) {
        if (Math.abs(event.beta) > 100) {
            this.rgdbdyBall.applyForce(new f.Vector3(this.gamma / 4, 750 / Math.abs(this.gamma), -this.beta / 15));
            return;
        }
        this.rgdbdyBall.applyForce(new f.Vector3(-this.gamma / 4, this.gamma / 10, -this.beta));
    }
    break;
```

In der 2. If-Abfrage befindet sich die Überprüfung des Gimbal Locks. Falls dieser eintritt, arbeite ich mit anderen Werten am Rigidbody. Auch diese Berechnungen wurden solange ausgefeilt bis sie den normalen Werten, ohne Gimbal Lock, am nähesten kamen.

5. Grundlegendes Design

Als die grundlegende Mechanik dem entsprach was ich mir anfangs vorgestellt hatte, ging es für mich als nächstes darum, ein passendes Spieldesign dafür auf die Beine zu stellen. Da das Spiel nur auf mobilen Endgeräten mit begrenzter Bildschirmfläche möglich ist, war die ursprüngliche Idee des hohlen Kubus mit verschiedenen Labyrinth-Ebenen eine eher schwierig umsetzbare Idee, da die Übersichtlichkeit dabei extrem leiden würde. Schon in der oben gezeigten Testumgebung war der Spielspaß durch die weiter weg stehende Kamera sehr begrenzt. Deswegen musste ich mich nochmal von der anfänglichen Idee entfernen und auf der selben technischen Grundlage nochmal neue Design Ideen entwickeln. Ein entscheidender Faktor der neuen Idee lieferte dann das

Perplexus Labyrinth, welches mir Jirka in den darauffolgenden Wochen zum Ausprobieren mitgegeben hatte.

5.1 Perplexus

Das Perplexus Labyrinth ist ein kugelförmiges Labyrinth in dem sich eine Metallkugel befindet. Das Ziel des Spiels ist es, sich durch alle Wege von Start bis Zielpunkt durch zu manövrieren. Das Besondere hierbei ist, dass die verschiedenen kleinen Wegabschnitte alle sehr unterschiedlich sind. Das führt dazu, dass man nur in kleinen Etappen weiter dazu lernt wie man die unterschiedlichen Abschnitte meistern kann. Vorallem die Vielfalt des Spiels brachte mich auf ein neues Spielkonzept.

Quelle: spiele-offensive.de

5.2 Entscheidungen

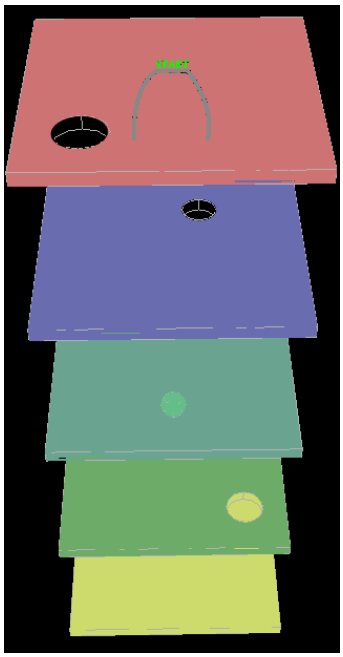
Das Perplexus Labyrinth hat mich in sofern auf ein neues Konzept gebracht, dass ich nicht mehr das gesamte Labyrinth auf den Bildschirm bringen wollte, sondern vielmehr einzelne Wegabschnitte in Level getrennt, mit einer näheren Ansicht der Kugel für den Endnutzer. So entstand ein viel immersiveres und abwechslungsreiches Spielerlebnis und für mich war die Generierung der Level deutlich übersichtlicher und einfacher zu gestalten.

6. Level Generierung

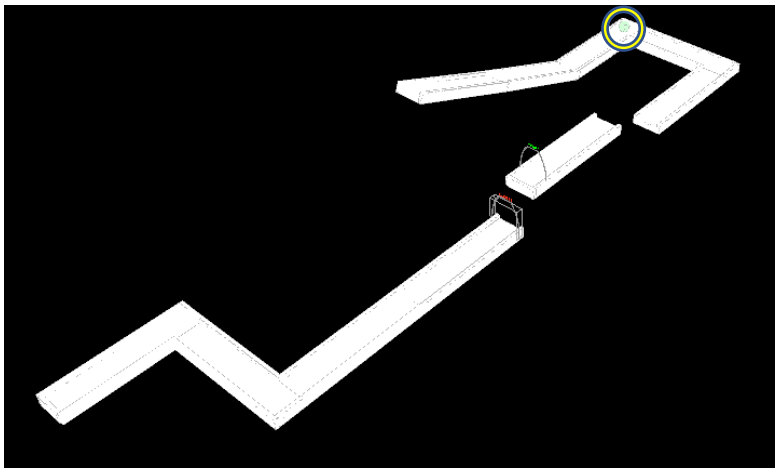
6.1 Umsetzung

Als feststand, dass es verschiedene Level innerhalb meines Spiels geben wird, machte ich mir Gedanken um die technische Umsetzung. Ein Ansatz wäre die Level innerhalb einer großen Szene in größeren Abständen aufzubauen und bei erfolgreichen Absolvieren eines Levels die Position des Balls auf das nächste Level zu setzen. Allerdings ist die Perfomance bei dieser Variante sehr ineffizent und ich entschied mich für eine andere Variante: ich setzte viel mehr auf Graph Instanzen. Mithilfe der Graph Instanzen Funktionalität in FUDGE kann der Entwickler beliebig viele weitere Graphen zur Laufzeit in die gerenderte Szene laden, aber auch wieder entfernen. So baute ich also eine Hauptszene mit dem Ball und passendem Licht. Die Level befinden sich in seperaten Graphen (in der Internal.json), die ich dann zum richtigen Zeitpunkt entweder instanziierte oder wieder entfernte. Somit entstand ein reibungsloses Spielerlebnis, welches dem Spieler genug Perfomance auf dem mobilen Endgerät und gleichzeitig eine Vielfalt an verschiedenen Level ermöglichte.

6.2 Level Übersicht

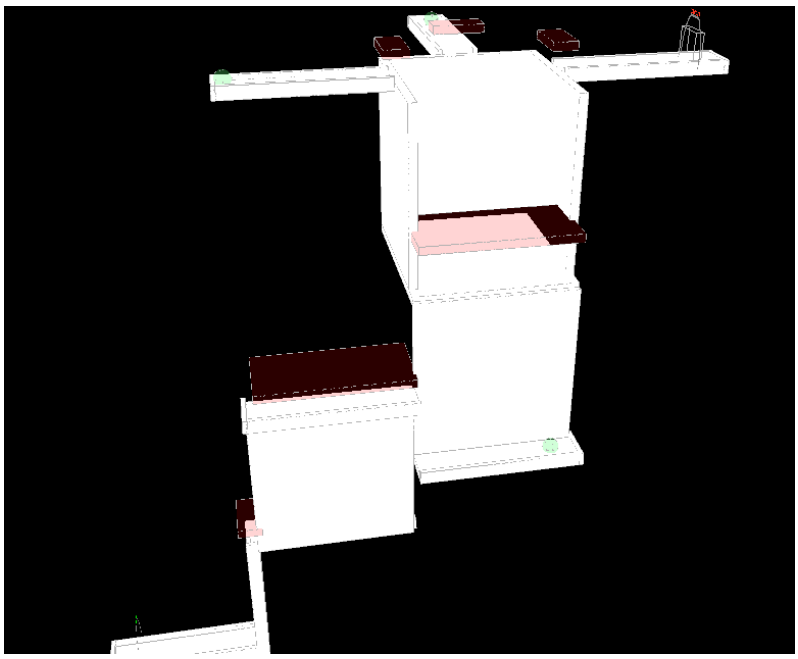


Das erste Level bestand erstmal nur aus fünf übereinander liegenden Plattformen. Diese hatten jeweils eine Textur mit Loch. Dort wo in der Textur ein Loch war, habe ich ein Zylinder Collider eingefügt, der als Trigger fungierte. Auf diesem Trigger liegt ein ComponentScript, welches den Ball als Rigidbody registriert und sobald Kontakt erkannt wird, den Collider der Parent Plattform (rechteckige gesamte Plattform) deaktiviert. Dadurch entsteht die Illusion, dass die Kugel tatsächlich durch das Loch fällt. Beim Verlassen des Triggers wird der Collider der Parent Plattform wieder aktiviert. Das Ziel des ersten Levels ist es also von der obigen roten Plattform bis zu der untersten, gelben Plattform zu gelangen, am besten über die visuell dargstellen Öffnungen. Mein Ziel war es bei diesem Level den Nutzer des Spiels in die Steuerung des Gyro Sensor einzuführen und ihn erstmal nicht zu überfordern.

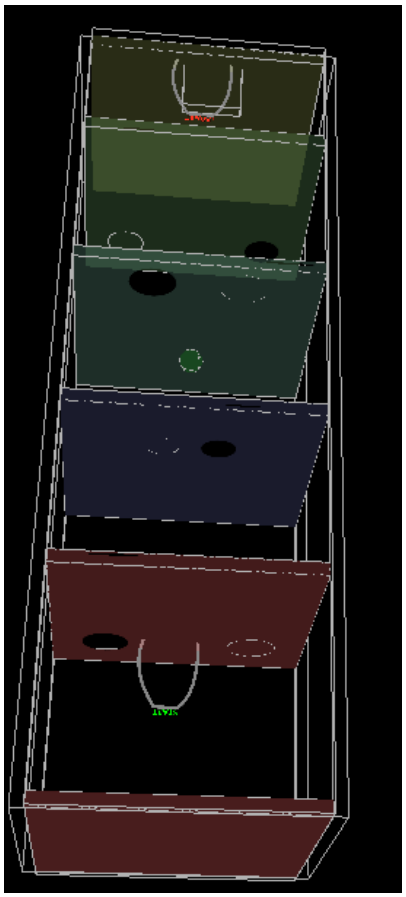


Das zweite Level sollte den Spieler näher an die mögliche Geschwindigkeit der Kugel heranbringen. Auch wenn das Endgerät bei diesem Level immer noch nicht schräg gehalten werden muss, hat es schon die erste knifflige Stelle. In der obigen Abbildung kommt besagte Stelle nach der grünen Kugel (oben rechts im Bild). Dort muss der Endnutzer das Handy

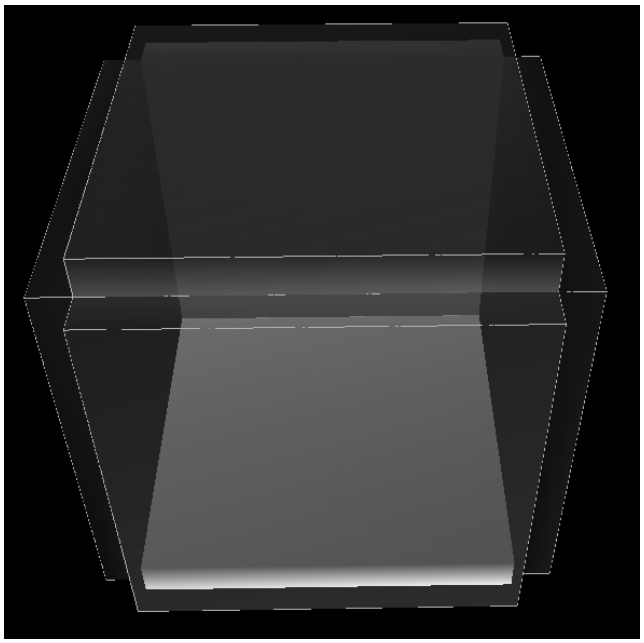
stark zu sich beugen, dass die Kugel an Geschwindigkeit aufnimmt, sodass die Kugel bis zur nächsten Plattform rüber fliegt. Falls man zu wenig weit oder über die Plattform hinaus schießt, stirbt man und muss wieder von vorne anfangen. Es sei denn, man hat die im Bild oben markierte grüne Sphere aufgesammelt. Diese transparente, grüne Sphere dient nämlich als *Checkpoint*. Sobald die Kugel aufgesammelt ist und man fliegt von der Plattform herunter, spawnt man wieder an der Stelle des *Checkpoints* und muss nicht nochmal die vorherige Strecke fahren.



In Level Drei ging es dafür dann schnell um die richtige Neigung des Handys, um im Level weiter voran zu kommen. Tatsächlich wurden in diesem Level fast alle Neigungen des Handys abgedeckt: über Kopf stellen, auf die Seite drehen und das Handy normal zu sich aufstellen. Die leicht transparenten, roten Plattformen dienen als Stopper gegen die Velocity des Rigidbodys, sodass die Kugel nicht über das Ziel hinausschießt und der Spieler noch kurz Zeit hat, das Endgerät wieder in die normale Neigung zu bringen. In der letzten oberen Hälfte von Level Drei gibt es dann noch ein kleines Rätsel: man kann an drei Wänden hochfahren, allerdings führt es nur bei einem Weg zum Ziel. Die anderen Wege besitzen nur einen *Checkpoint* und man muss wieder zurück fahren.



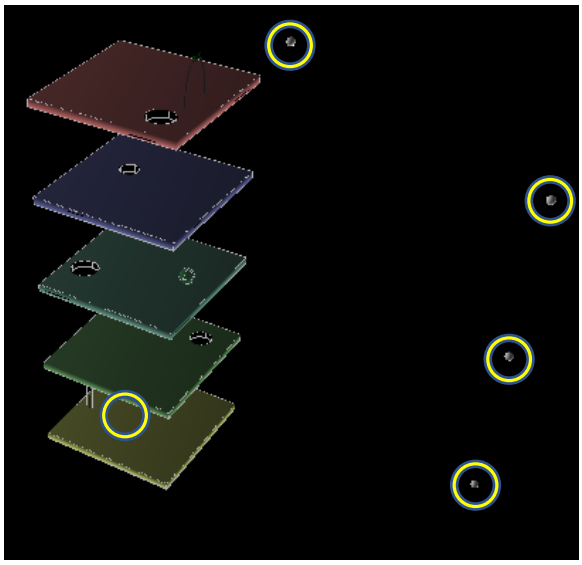
In Level vier geht es dann in die Über Kopf Stellung. Das Level kommt dem Nutzer wahrscheinlich schon bekannt vor: es ist grob gesehen das erste Level, nur auf den Kopf gestellt. Man muss das Handy also über den Kopf halten, um dann durch die illustrierten Löcher der Plattformen durch zu manövrieren. Um über Kopf nach wie vor die Kugel durch die Plattformen sehen zu können, habe diese alle eine gewisse Transparenz bekommen. Bei diesem Level muss der Spieler hohes Geschick beweisen, da das Schwenken der Kugel verkehrt herum ausgeführt werden muss. Um das Level nicht zu einem frustrierenden Erlebnis zu machen, habe ich die Plattformen mit einem unsichtbaren Collider verbunden, sodass man nicht einfach von den Plattformen runterfliegen kann.



Das letzte und fünfte Level ist mehr als Platzhalter gedacht. Es war tatsächlich zuvor meine Testumgebung und besteht lediglich aus einem hohlen Kubus der mit Collidern ausgestattet wurde. Das Level soll für den Spieler nur noch mal eine Umgebung zum Austoben und Ausprobieren der Mechanik bereitstellen.

6.3 Kamera Flug

Als die ersten Level fertig gestellt waren, wollte ich die erste Beta Version einigen Test Spielern zeigen und ihre Reaktionen studieren. Abgesehen davon, dass der Großteil erstmal mit der Mechanik zurecht kommen musste, fiel auf, dass einigen die Übersicht in den Leveln fehlte. Gerade in Level drei kam es oft zu Frustration, da Sie nicht wussten, wo es als Nächstes hingehen sollte und sie deswegen über *Try and Error,* jede erdenkliche Richtung mit dem Spielball versuchten und oft in den Tod stürzten. Für mich war klar, es brauchte noch eine zusätzliche Unterstützung, die dem Spieler zeigte, wie das Level durchzuspielen war. Meine Idee dafür war es, für jedes Level eine Kamerafahrt zu integrieren. Den Kameraflug setzte ich um, indem ich verschiedene Transform Objects platzierte mit einer genauen Position und Rotation. Die Transform Objects wurden dann im Code in einen Array übergeben und der Kamera als Werte übergeben. Sobald die Kamera ihren ersten Punkt erreicht hatte, wurde das nächste Transform Object als Wert übergeben usw., bis die Kamera ihren letzten Wegpunkt erreicht hatte. In der obigen Abbildung wurden die Transform Objects im Editor visualisiert: Dabei handelt es sich um das erste Level und die Kamera fliegt dann dementsprechend von der oberen Plattform nach hinten und unten bis sie die unterste Plattform erreicht und dann Richtung Ziel fliegt. Danach schnellt die Kamera wieder zur Standard View zurück und der Spieler kann das



Level beginnen. Ein nettes Gimmick welches ich noch zusätzlich eingebaut habe ist, dass wenn der Spieler den ersten Checkpoint erreicht und stirbt, wieder die Möglichkeit zur Kamerafahrt hat, aber von der Stelle des Checkpoints aus und nicht mehr von Anfang an des Levels.

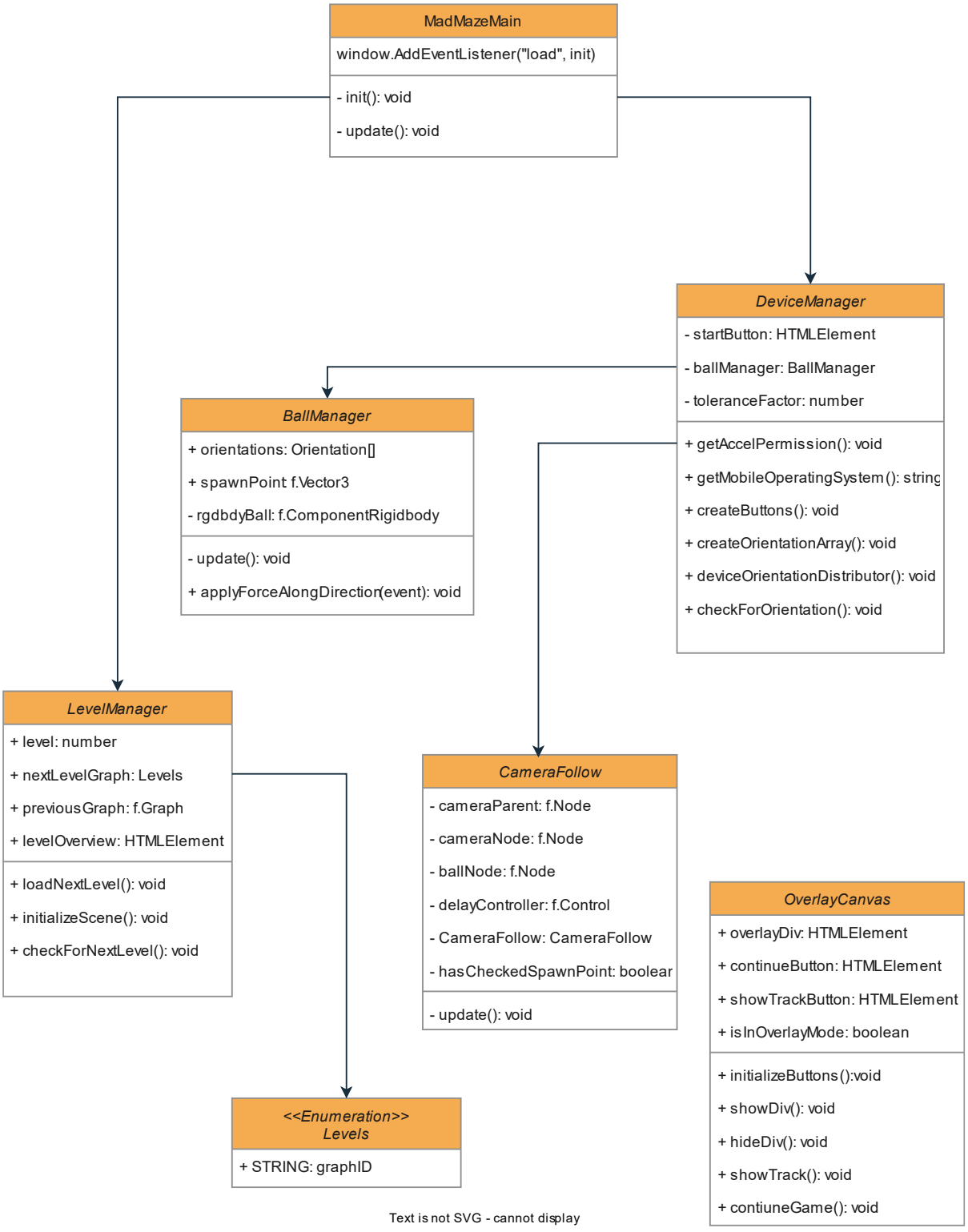
7. Progressive Web App

Als alle Level mit Kamerafahrt ausgebaut waren und alle kleineren Fehler beseitigt waren, ging es im letzten Schritt noch darum, die Web App zu einer Progressiven Web App (PWA) umzubauen. Der große Vorteil für meinen Prototyp ist, dass die Ränder des Browsers wegfallen und man die ganze Bildschirmfläche zum Spielen hat. Die Umsetzung fiel vergleichsweise einfach aus: man legte ein Web-Manifest an (siehe rechts). Danach ließ sich die Website als Lesezeichen auf den Home Screen holen und öffnete sich anschließend als eigenstehende App.

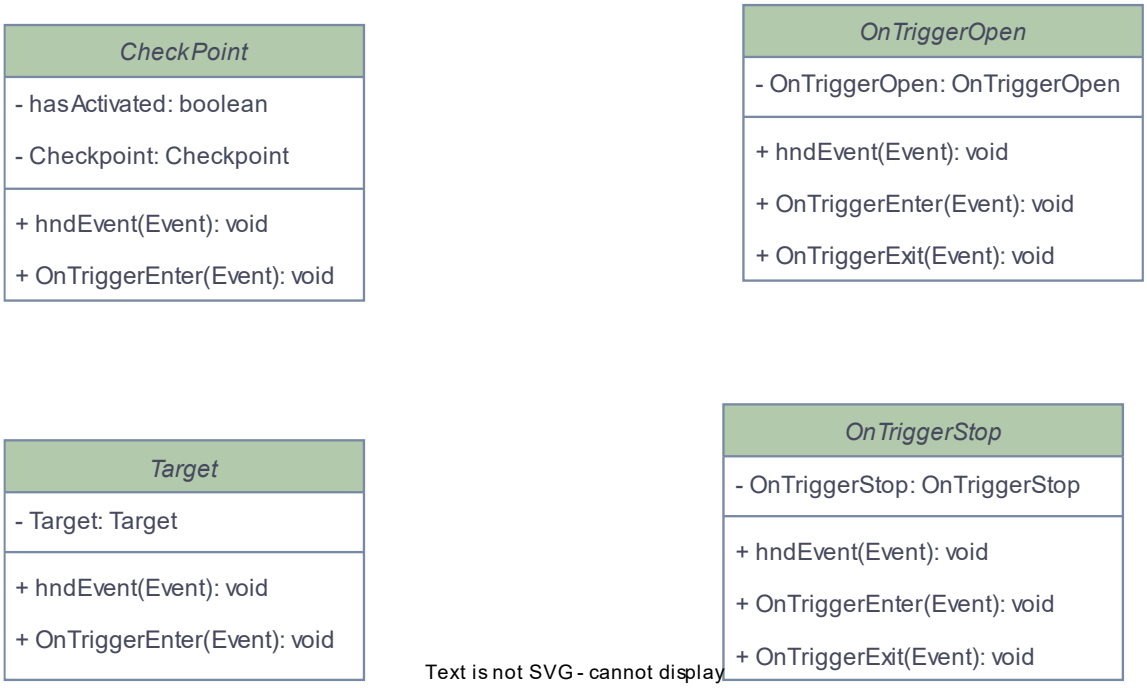
```
{
    "name": "MadMaze",
    "display": "standalone",
    "icons": [{
        "src": "icons/icon-192.png",
        "sizes": "192x192"},
    {
        "src": "icons/icon-512.png",
        "sizes": "512x512"}
    ]
}
```

8. UML

Klassen Struktur



Component Scripts



Text is not SVG - cannot display