

„Erstellung einer interaktiven 3D-Applikation mit Three.js“

PROJEKTDOKUMENTATION

Computergrafik und Animation

Sommersemester 2020

Name: Jonas Oelke

Matrikelnummer.: 241427

Inhaltsverzeichnis

| | | |
|--------|--|---|
| 1. | Einleitung | 2 |
| 2. | Aufbau der Applikation..... | 2 |
| 2.1. | Allgemeine Struktur / Szenegraph | 2 |
| 2.2. | Lichtquellen | 3 |
| 2.2.1. | HemisphereLight..... | 3 |
| 2.2.2. | DirectionalLight | 3 |
| 2.3. | Meshes und Groups | 4 |
| 2.3.1. | Objekt A: Fortress.js - Die Festung..... | 4 |
| 2.3.2. | Wall.js – Die Mauer..... | 4 |
| 2.3.3. | Gate.js – Das Tor | 5 |
| 2.3.4. | Tower.js – Der Wachturm | 6 |
| 2.3.5. | Objekt B: Mortar.js – Der Mörser | 7 |
| 2.3.6. | KeyFromFile.js – Das Zielobjekt | 8 |
| 3. | Ablauf der Applikation | 8 |
| 4. | Ausblick..... | 9 |

1. Einleitung

Die folgende Dokumentation schildert die Hintergründe zum Aufbau und zur Entstehung einer webfähigen 3D-Echtzeit-Anwendung. Als Anwendungsfall wird hier ein kleines Spiel herangezogen. Der Spieler wird durch einen Mörser abgebildet, über den dieser die Kontrolle verfügt. Ziel ist es ein Schlüsselobjekt einzusammeln, das umringt von einer Festung und weiteren gegnerischen Mörsern bewacht wird.

Entstanden ist diese Idee aus einer Faszination für die simulierte Physik in der 3D-Echtzeitgrafik. In der Ideenfindungsphase wurde schnell klar, dass komplexe Simulationen bspw. von Wasser oder Explosionen im Rahmen dieses Projekts schwer zu realisieren sein würden – und auch Annäherungen an ein solches Verhalten über zahlreiche kleine Box-Meshes aufgrund von Performance-schwierigkeiten eher auszuschließen sind.

Jedoch sollte von vornherein der Spieltrieb des Anwenders geweckt werden, indem er die Möglichkeit bekommt die ihm dargebotene Welt nach Belieben zu zerstören. Ursprünglich stand auch die Idee einer Pyramide, die aus einzelnen Kisten besteht im Raum – die Festung ist als eine Abwandlung dieser Idee zu betrachten, da diese ebenfalls aus aufeinandergestapelten Objekten besteht. Jedoch bietet eine Festung als eigenständig erstelltes Objekt in Three.JS mehr Möglichkeiten den Anforderungen hinsichtlich der Komplexität und Integration bspw. Animationen und Constructive Solide Geometries gerecht zu werden.

Um dem Anwender die Möglichkeit zu geben, selber Einfluss auf diese Welt zu nehmen und dem vielleicht unbewusst existierenden Drang gerecht zu werden, einfach mal etwas zu zerstören, ist hier Nutzung von Echtzeitanwendungen unerlässlich. Durch eine 3D-Welt bekommt der Nutzer zusätzlich einen weitaus realistischeren Eindruck der simulierten Welt, in der die Physik aber dennoch sehr ähnlich der in der Realität ist.

2. Aufbau der Applikation

2.1. Allgemeine Struktur / Szenegraph

Die Abbildung 1 zeigt den Szenegraphen, der die allgemeine Struktur der Anwendung darstellt. Dabei wird ersichtlich, dass die eigenen Klassen zumeist die Klasse Three.Group erweitern. Insbesondere „Fortress“ beinhaltet Objekte aus drei weiteren Unterklassen, die dann erst die tatsächlichen Meshes erzeugen.

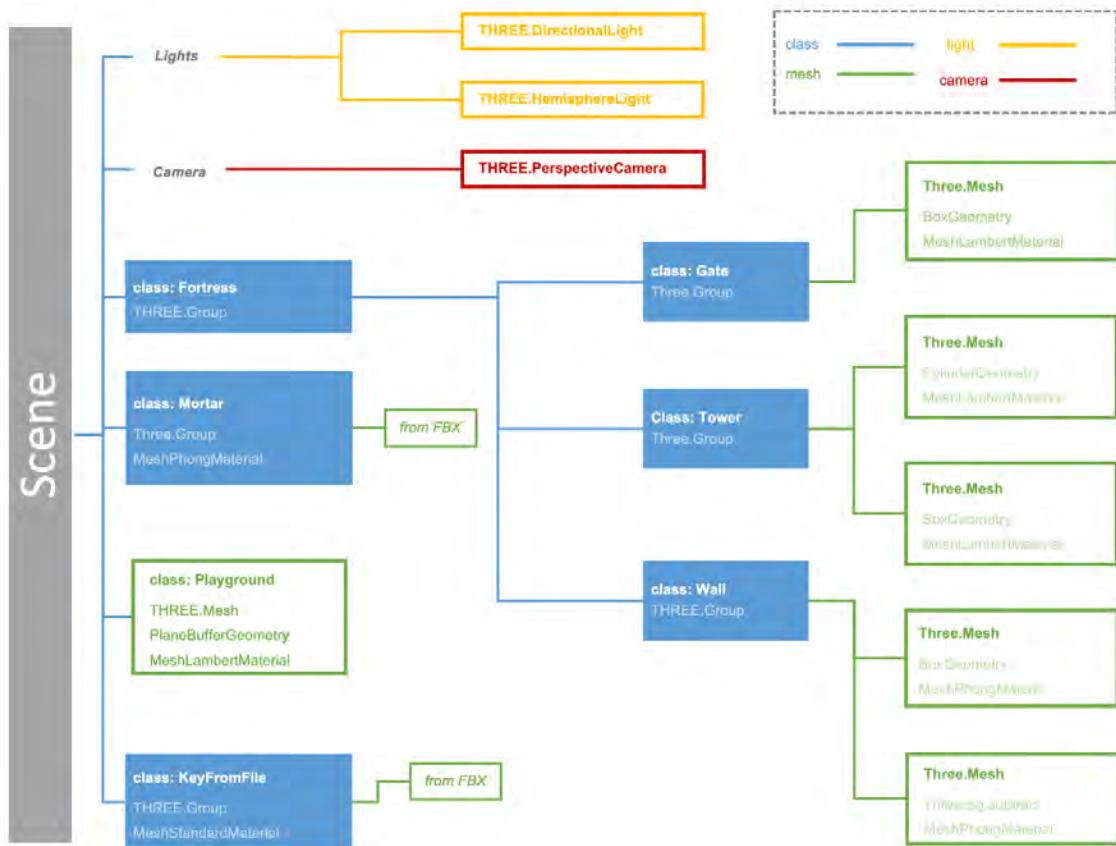


Abbildung 1: Szenenraph

2.2. Lichtquellen

2.2.1. HemisphereLight

In der Szene kommen insgesamt zwei Lichtquellen vor. Durch ein *HemisphereLight* wird zuerst grundsätzliches Umgebungslicht unter freiem Himmel simuliert.

Dabei wurden die Farbeinstellungen für Skycolor und Groundcolor so gewählt, dass von dem Himmel ein helles und leicht blaues Licht und vom Boden ein dunkles leicht grünes Licht ausgeht, um die Reflektionen durch den Rasen zu berücksichtigen. Hieraus begründet sich auch der Einsatz gegen ein *AmbientLight*, da dieses alle Elemente gleichermaßen und in gleicher Farbe aufgehellt hätte.

2.2.2. DirectionalLight

Das *DirectionalLight* wird genutzt, um eine gerichtete Lichtquelle zu erzeugen, die der einer Sonne bei wolkenlosem Himmel nahekommt. Die Position und Intensität wurde so gewählt, dass Schatten verhältnismäßig kurz und die Szene sehr hell daherkommt, um die Mittags- oder frühe Nachmittagssonne zu simulieren. Durch die geringe Größe des Mörsers und seine grundsätzlich recht offene Beschaffenheit erzeugt dieser nur einen recht kleinen Schatten unter sich, der jedoch klar erkennbar wird, sobald der Mörser durch die Pfeiltasten in der Szenerie bewegt wird.

2.3. Meshes und Groups

2.3.1. Objekt A: Fortress.js - Die Festung

Die Klasse **Fortress.js** erweitert die Klasse THREE.Group und ist damit das Elternobjekt für alle Objekte in der Festung. Daher enthalten erst die Kindobjekte dieser Klasse die tatsächlich sichtbaren Meshes. In dieser Klasse finden hingegen die Festlegungen der Positionen und Rotationen der untergeordneten Objekte statt. Weiterhin besteht die Festung in der Szenerie (abgesehen von den Mörsern auf den Türmen) komplett aus Primitiven.



Abbildung 2: Instanz der Klasse Festung

2.3.2. Wall.js – Die Mauer

Als zentraler Bestandteil der gesamten Festung wird in der Klasse Wall.js die Mauer erzeugt. Diese kann durch den Konstruktor mit einer variablen Boxgröße (=boxSize) erzeugt werden. Um aus Performancegründen die Anzahl der Objekte in der Szene gering zu halten und dem User dennoch das Gefühl zu geben, er könne die Festung zerstören (also in Einzelteile zerlegen) wurden den Box-Meshes beim Instanziieren in der Fortress.js eine Größe von 100x100 Einheiten zugewiesen.

In den meisten Fällen grenzt eine Mauer an einen Wachturm an. Der Wachturm ist bei dieser Festung rund und wird durch einen Zylinder dargestellt. Daher ließe sich eine aus Würfeln bestehende Mauer nicht direkt neben einen Turm stellen lassen ohne, dass die Würfel entweder in den Turm hineinragen oder es zu einer sichtbaren Lücke zwischen Mauer und Turm kommen würde. Daher wurde hier eine Constructive Solid Geometry mittels threecsg.js erzeugt, bei dem ein Zylinder, gleicher Art wie der des Wachturms, von einem Würfel, gleicher Art der aus der Mauer, an der entsprechenden Position abgezogen wurde. Das hierbei entstandene Mesh stellt nun das Randstück der Mauer als Verbindung zum Turm da.

Es stellte sich jedoch als recht komplex heraus eine Textur auf das berechnete Mesh zu applizieren. Grundhierfür sind die bei der Berechnung verlorengegangenen UV-Koordinaten, anhand derer die Position auf der Textur, mit der auf dem Face des Meshs verknüpft wird. Dieses Problem ließe sich dadurch Lösen, die UV-Koordinaten neu zu berechnen. Weiterhin ließe sich mittels des THREE.SimplifyModifier die Anzahl der Faces wahrscheinlich verringern, sodass es mit ein bisschen Aufwand mögliche wäre die UV-Koordinaten händisch festzulegen. Es Zeitgründen wurde jedoch bei diesem Projekt auf eine aufwendige rechnerische und programmiertechnische Implementierung dieser Verfahren weitestgehend verzichtet. Es wurde lediglich die Funktion von Stackoverflow-User „Tamlyn“ (gepostet am 15.

Dezember 2014)¹ implementiert und auf das erzeugte Mesh angewandt. Damit konnten die UV-Koordinaten auf der Vorder- und Rückseite korrekt berechnet werden, für die anderen Seiten konnte diese Funktion jedoch keine adäquaten Ergebnisse erzielen. Da anschließend lediglich die Oberseite des Meshes zu sehen ist, wurde nun eine Plane erzeugt, die minimal über dem Mesh liegt und sich mit ihm in einer Gruppe befindet. Auf dieser Plane, die die Größe der Box ohne Abzug des Zylinders hat, wurde eine separate Texture-Map gelegt. Bei dieser handelt es sich um ein bearbeitetes PNG der ursprünglichen Textur, das jedoch in dem Bereich, der von dem Mesh subtrahiert wurde, transparent ist.

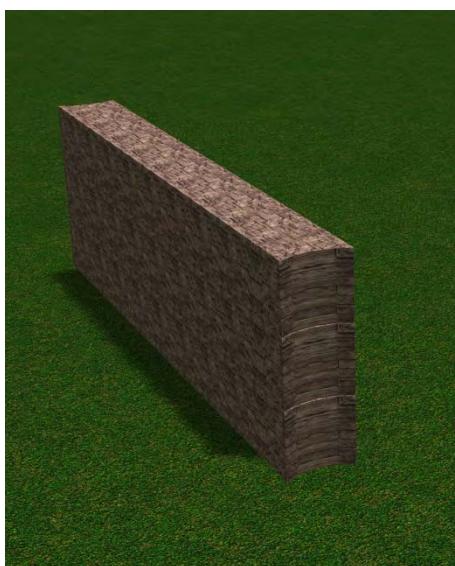


Abbildung 3: Mauer mit Blick auf das CSG-Objekt

Im Konstruktor der Wall-Klasse kann als optionaler Parameter „gateWall“ übergeben werden. Hiermit wird ausgedrückt, dass diese Mauer an ein Objekt der Klasse „Gate“ angrenzen wird und damit statt der CSG an einer Seite mit einer Box endet. Die flache Seite wird sich in diesem Fall immer auf der rechten Seite befinden, sodass die Mauer ggf. um 180° gedreht werden muss, um der Richtung zu entsprechen.

Bedingt durch die Tatsache, dass die `physicalBodies` mit den Child-Elementen dieser Klasse verknüpft sind, würden direkte Rotationen und Translationen des Wall-Objekts keine Bewegung der `physicalBodies` implizieren, da sich die Koordinaten der Child-Elemente (die relativ zu den Koordinaten der Elternklasse sind) nicht

ändern. Daher überschreibt die Klasse „Wall“ die Methode „rotateY()“. Die neue Methodewendete eine Rotation nicht mehr auf die Klasse selber an, sondern überträgt diese Rotation mittels einer Rotationsmatrix auf die Position der Kindobjekte, die sich daraufhin um ihren lokalen Ursprung drehen. Im Anschluss wird die Methode „updatePositions()“ der physikalischen Welt aufgerufen, mit der die Positionen der `physicalBodies` analog hierzu überschrieben werden.

2.3.3. Gate.js – Das Tor

Das Tor zählt zu den verhältnismäßig simpelsten Elementen der Festung. Neben den grundsätzlichen Parametern wie Größe, passendem `physicalBody` und einer hölzernen Textur werden im Konstruktor die beiden Tweens zum öffnen und schließen des Tors definiert. Da das Öffnen und Schließen des Tors einer Rotation um die X-Achse (und nicht um die eigene Mitte) entspricht, besteht die Klasse „Gate“ als Erweiterung der Klasse `THREE.Group` trotzdem nur aus einem Kindobjekt, das

¹ Javascript - THREE.js generate UV coordinate – Stack Overflow:

<https://stackoverflow.com/questions/20774648/three-js-generate-uv-coordinate> (letzter Zugriff: 20. Juni 2020)

innerhalb dieser Gruppe um die Hälfte seiner Höhe nach oben (Y-Richtung) verschoben worden ist. Somit entspricht die untere Kante des Tors dem Pivot-Punkt der Klasse „Gate“. Die Methode „rotateX()“ ist analog zu Wall.js überschrieben, um die Rotation mittels einer Rotationsmatrix wieder zurück an das Kindobjekt weiterzugehen. Es entsteht der Eindruck, dass sich das Tor um die X-Achse rotiert.

2.3.4. Tower.js – Der Wachturm

Der Wachturm stellt ein wichtiges Objekt für die gesamte Festung dar. Er grenzt an die Mauern an und bildet das Gegenstück zu den Constructive Solid Geometries der Mauer (siehe 2.3.2 Wall.js – Die Mauer). Der Wachturm besteht aus einem Zylinderobjekt mit einem Radius von 90 Einheiten, das den Standfuß des Turms abbildet. Ebenfalls ist hierauf die Standardtextur der Festung appliziert, die bezogen auf die Y-Achse drei mal wiederholt wurde, um der Texturhöhe der Mauer zu entsprechen.

Darüber befindet sich ein weiterer Zylinder, der einen unteren Radius von 90 und einen oberen Radius von 120 Einheiten besitzt. Dieser stellt die Plattform für den Mörser da. Daher ist auf die Oberseite dieses Zylinders eine hölzerne Textur angewandt, die einen parkettartigen Fußboden darstellt. Die Steintextur der Festung wurde acht mal um den Zylinder gewickelt und in der Höhe einmal angewandt.

Die Plattform auf dem Turm bzw. die steinerne Begrenzung (im Quellcode „top“ genannt) besteht ähnlich wie die Mauer aus einzelnen Meshes. Ausgehend von einem Einheitsvektor auf der X-Achse (mit dem Ursprung im Mittelpunkt der Zylinderfläche), wird dieser mit einem Skalar multipliziert, um den Vektor auf die Länge des Zylinderradius (120 Einheiten) zu bringen. Dabei wird noch die Hälfte der Tiefe der Box subtrahiert, damit die Box mit dem Rand des Zylinders abschließt. Über eine For-Schleife wird nun dieser Vektor sukzessive mit einer Rotationsmatrix multipliziert, die den Vektor um $360^\circ / 20$ Segmente = 18° zum Mittelpunkt der Kreisfläche dreht. Anschließend wird das Mesh selber noch gedreht, um die Außenfläche der Box kongruent auf das Kreissegment zu legen.

Diese Berechnungen werden über eine weitere Schleife zwei mal ausgeführt bei dem jeweils ein Y-Offset der Boxhöhe addiert wird, um eine zweite Reihe zu erzeugen, bei der die Meshes und die physicalBodies nur bei jedem zweiten Durchlauf der Szene hinzugefügt werden.

Die Meshes in der unteren Reihe ragen auf der Innenseite leicht ineinander, daher kann der physicalBody nicht dieselben Dimensionen haben wie das Mesh. Um eine



Abbildung 4: Wachturm mit Mörser auf der Plattform

initiale Kollision zu verhindern wurde der Einfachheit halber der `physicalBody` in seiner Breite leicht verringert.

Auf der Plattform befindet sich jeweils eine Instanz des Objekts B, dem Mörser (siehe auch 2.3.5 Objekt B: Mortar.js – Der Mörser). Dieser ist in der Lage über der Aufruf der Funktion `aim()` sich in Richtung eines Zielobjekts zu rotieren. Als Zielobjekt dieser Instanz ist der gegnerische Mörser des Spielers festgelegt. Mit jeder Bewegung der vom Anwender gesteuerten Instanz des Mörsers wird die Rotation und die Distanz neu berechnet, kommt der Spieler zu nahe, schießt der Mörser in Richtung des Spielers und es wird ein entsprechendes Geräusch abgespielt.

2.3.5. Objekt B: Mortar.js – Der Mörser



Abbildung 5: Importiertes FBX-Modell des Mörser

Dieser Mörser ist ein Importiertes FBX-Objekt, das mittels `THREE.FBXLoader()` geladen wird. Das 3D-Objekt enthält jedoch physikalische Texturen und Maps, die von den Materialien in `Three.js` nicht unterstützt werden und beim Rendern Probleme verursachen. Daher ist es notwendig direkt beim Import ein neues Material zu erstellen, dass ausschließlich die Diffuse- und die Normal-Map zu dem Mörser beinhaltet. Dieses wird anschließend auf alle Kindobjekten der Group angewandt.

Der Mörser bietet unter anderem die Möglichkeit eine Kugel abzufeuern. Die Beschleunigung dieser Kugel lässt sich entweder zu Beginn über den Konstruktor festlegen, wird Anhand des Abstands des Mörsers zu einem übergebenen Zielobjekt berechnet oder wird zufällig aus einem gewählten Bereich generiert. Die Anwendung ist so programmiert, dass der Spieler eine Kugel durch Drücken der Leertaste abfeuern kann. Der Mörser besitzt eine physikalische Box um sich herum, die jedoch den Lauf nicht komplett miteinschließt. Etwas über der physikalischen Box wird die Kugel platziert, um keine initiale Kollision der beiden Objekte herbeizuführen. Die Neigung des Laufs beträgt nach augenscheinlicher Begutachtung etwa 28° - diese Neigung wird durch eine Rotationsmatrix auch auf den Richtungsvektor der Bewegung der Kugel angewandt.

Weiterhin besitzt der Mörser durch die Methode `aim()` die Fähigkeit auf ein übergebenes Objekt zu zielen. Dabei wird ein zweidimensionaler Richtungsvektor von der Position des Mörsers zur Position des Objekts berechnet und der Winkel zwischen der X-Achse und dem Richtungsvektor als Rotation auf das Mörser-objekt angewandt. Diese Funktion ist ebenfalls in der Lage den Abstand zum Zielobjekt zu berechnen und ab einem bestimmten Abstand die `fire()` Methode aufzurufen und damit auf das Zielobjekt zu schießen. Im Fall des abstandsbedingten Aufrufs der `fire()` Methode wird eine abstand-abhängige Beschleunigung des Geschosses berechnet und der Klasse als Property zugewiesen (`this.velocity`).

2.3.6. KeyFromFile.js – Das Zielobjekt

Ziel der Applikation ist es, dass der Spieler mit dem Mörser, den er mittels der Pfeiltasten kontrollieren kann, den Schlüssel einsammelt. Die erzeugte Instanz dieser Klasse wird mittig in der Festung platziert, sodass der Mörser des Spielers zwischenzeitlich von allen vier Seiten durch die Türme angegriffen wird.

Da das FBX-Modell des Schlüssels keine Texturen beinhaltet, wird während des Ladevorgangs ein neues MeshStandardMaterial erzeugt, dem eine goldene Farbe sowie passende metalness und roughness Werte zugewiesen werden.

Dem Schlüssel hängt eine Animation an, die das Objekt innerhalb von fünf Sekunden um 360° entlang der Y-Achse dreht. Durch die Drehung wird dem Spieler verdeutlicht, dass es sich hierbei anscheinend um das Objekt handelt, das er einsammeln muss.

Bei jeder Vorwärtsbewegung des Spielerobjekts wird die Distanz zur Position des Key-Objekts überprüft. Kommt der Spieler diesem angemessen nahe, gilt das Objekt als eingesammelt. In diesem Fall wird ein klassischer „coin collection“-Sound abgespielt und der Schlüssel von der Szene entfernt.



Abbildung 5: Importiertes FBX-Modell des Schlüssels

3. Ablauf der Applikation

Bevor die eigentliche Szenerie geladen wird, bekommt der Spieler in seinem Browserfenster kurze Instruktionen, welches Ziel er verfolgt und wie die Steuerung funktioniert. Nach einem weiteren Klick wird die Szene geladen. Der Spieler sieht die Szene, mit der Festung in der Mitte, die Kamera zeigt die Festung sowie den Mörser des Spielers. Versucht der Spieler sich nun, wie in dem Startbildschirm erklärt, mit den Pfeiltasten zu bewegen, wird er feststellen, dass sich das Mörser vor ihm rotiert (linke und rechte Pfeiltaste) oder sich nach vorne bewegt.

Durch den Hinweis, auf das Tor zur Festung wird der neugierige Spieler geneigt sein, auf das Tor zu klicken. Dieses lässt sich daraufhin öffnen oder schließen. Hat der Spieler die Festung jedoch zuvor schon attackiert, steht ihm diese Option nicht mehr offen, da die Animation nach einem Angriff deaktiviert wird.

Der Spieler wird sich nun mit seinem Mörser vor zur Festung wagen und evtl. die Festung angreifen. Sobald der Spieler eine bestimmte Nähe zur Festung aufgebaut hat, wird er feststellen, dass er von den Mörsern auf den Türmen angegriffen wird – das wird durch einen Schuss-Sound unterstrichen. Sollte der Spieler bis dahin noch nicht festgestellt haben, dass alle vier Mörser jeden Schritt verfolgen und jederzeit auf ihn zielen, so wird er dies spätestens in kurzer Distanz zur Festung feststellen.

Ihm steht es nun offen, wie er seinen Angriff vollziehen möchte. Er könnte erst die Türme angreifen und die gegnerischen Mörser damit außer Schussweite bringen oder direkt die Mauer einreißen und zum Schlüssel navigieren. Beide Optionen sind legitim und können zum Erfolg führen. Früher oder später wird der Spieler jedoch feststellen, dass die gegnerischen Geschosse ihn durchaus von seinem Weg abbringen können oder ihm Trümmer der Festung im Weg liegen. Er kann versuchen diese vor sich herzuschieben, was jedoch meist nicht sehr effektiv ist.

Hat der Spieler es geschafft, in die Festung einzudringen und sich dem Schlüssel zu nähern, ertönt ein Geräusch, dass ihm akustisch signalisiert, dass er den Schlüssel aufgesammelt hat – weiterhin verschwindet der Schlüssel nun aus der Szene. Zum Schluss kann der Spieler den Rest der Festung nach Belieben in Schutt und Asche legen.

4. Ausblick

Die Anwendung stellt an sich bereits die grundlegenden Elemente eines Spiels bzw. eines Levels von einem vollwertigen Spiel bereit. Das Ziel ist jedoch verhältnismäßig einfach zu erreichen.

Um dem Spieler eine höhere Motivation zu geben, könnten seine abgefeuerten Geschosse oder die Zeit zum Erreichen des Schlüssels angezeigt werden. Weiterhin könnte im Browser ein Fortschrittsbalken sichtbar sein, der den Gesundheitszustand des Spielers anzeigt. Dieser würde sich mit jedem gegnerischen Geschoss, das den Spieler trifft, verschlechtern.

Weiterhin könnte eine zweite Kamera implementiert werden, die aus der Perspektive des Mörsers schaut, sodass der Spieler mit einer Taste schnell zwischen den Perspektiven wechseln kann. Genauso könnte eine Kameraverfolgung hinzugefügt werden, sodass sich die Kamera bei einer Bewegung des Mörsers entsprechend mitbewegt.

Ebenso ließe sich das Verhalten der Gegnerischen Mörser verbessern. Diese könnten sich bspw. nach einem Aufprall auf den Boden aufrichten und den Spieler verfolgen und somit noch gezielter angreifen. Es sind auch weitere gegnerische Angriffselemente denkbar: Bspw. ein Turm von Bogenschützen, aus dem Pfeile auf den Spieler geschossen werden. Ebenso ließe sich die Festung nach Belieben ausbauen.

Genauso ließe sich eine Steuerung des Laufs integrieren, sodass der Spieler (wie auch die gegnerischen Mörser) in der Lage wären flacher oder steiler zu schießen.

Es zeigt sich, dass es eine Vielzahl an Erweiterungsmöglichkeiten der bestehenden Szene gibt - zusätzlich besteht die Möglichkeit unterschiedliche Level und Aufgaben zu integrieren.

Grenzen werden dem Ideenreichtum fast nur durch die physikalische Simulation gesetzt. Brennende Geschosse bspw. wären mit den bisherigen Bibliotheken nicht möglich – weiterhin gestaltete sich die eine Implementierung von Explosionen und dynamischen Zerstörungen schwierig. Die Zerlegung von vergleichsweisen großen Körpern wie dem Zylinder der Türme in kleinere Meshes gehen zulasten der Performance. So war ursprünglich geplant eine Explosion eines Meshes durch die Erzeugung mehrerer kleinerer Meshes mit einer zufälligen Beschleunigung zu simulieren. Eine Grundfunktion dafür befindet sich auch noch in dem Quellcode und könnte Grundlage für erneute Versuche der Richtung sein. Je nach Anzahl der erzeugten Meshes und physicalBodies verringert sich die Framerate soweit, dass ein Spielen teilweise unmöglich wird. Daher muss hierbei mit besonderer Rücksicht auf die Performance gearbeitet werden und evtl. ein Schutzmechanismus implementiert werden, der die Anzahl der zu generierenden Meshes reduziert sobald ein Rückgang der Framerate verzeichnet wird.