

Fakultät   
Ingenieurwissenschaften und Informatik

Advanced Rich Media Applications   
Sommersemester 2018

Projektbericht

über das Thema

Erstellung eines Multiplayer Shooter Spiels unter Verwendung von NodeJS, ThreeJS, CannonJS, MySQL und Socket.IO

|  |  |
| --- | --- |
| Autoren: | Dennis Lammers (675452)  Henri Hesse (671798)  Alexander Beiserz (123456) |
| Prüfer: | Dipl.-Inf. (FH) Björn Plutka |
| Abgabedatum: | 14.09.2018 |

Inhaltsverzeichnis

|  |  |
| --- | --- |
| Einleitung | 1 |
| Architektur | 2 |
| Ablauf | 3 |
| Modelle und Animation | 3 |
|  |  |

// Note: Tabelle vor Abgabe unsichtbar machen, wie auf dem Deckblatt ganz unten!

Einleitung

Die erstellte Anwendung ist ein webbasiertes Multiplayer-Shooter-Spiel im Science Fiction Genre. Es gibt zwei Teams, die gegeneinander im Team-Deathmatch-Szenario antreten und versuchen, möglichst viele Punkte für ihr Team zu gewinnen.



Für die Visualisierung wird die browserbasierte 3D-Grafikbibliothek ThreeJS genutzt. Durch die Verwendung von WebGL profitieren Benutzer von GPU-beschleunigten Berechnungen und der daraus resultierenden, hohen Performance. Außerdem wird dieses Framework stetig weiterentwickelt und ermöglicht eine einfache Einbindung von eigenen 3D-Modellen und Animationen. Desweiteren hat ThreeJS eine integrierte Audiobibliothek, die für Soundeffekte im Spiel genutzt wird.

Das Verhalten von Objekten im dreidimensionalen Raum, sowie die Erkennung von Kollisionen wird durch die Physics-Engine – CannonJS – berechnet.

Das Multiplayer-Feauture, der Chat, sowie die Kommunikation von Server und Clients wird durch Sockets mit Verwendung von Socket.IO ermöglicht.

Durch Nutzung von MySQL können Daten serverseitig gespeichert und bei Bedarf abgefragt werden.

Die Login- und Registrierungsseite, sowie Textelemente im Spiel werden durch Nutzung von CSS-Flex und Bootstrap positioniert und benutzerfreundlich gestaltet. Desweiteren hilft jQuery bei der Editierung von HTML-Elementen.

Für die Versionsverwaltung kommt git mit Hilfe von Sourcetree zum Einsatz.

Architektur

Die Webanwendung läuft auf einem NodeJS-Express-Server, der über die Datei *app.js* im Root-Verzeichnis des Projektes gestartet wird. Der Server übernimmt das Routing, sowie den Request-Response-Mechanismus, um dem Client bei Anfrage die Datei *index.html* aus dem *public* Ordner zukommen zu lassen. Dieser Ordner beinhaltet alle für den Client einsehbaren Dateien, das Root-Verzeichnis bleibt unsichtbar. Die *index.html* ist in diesem Projekt die einzige HTML-Datei. Dort sind alle für die Anwendung benötigten Ressourcen eingebunden, sodass das Spiel bei dem Client direkt gestartet werden kann.

Folgende Übersicht veranschaulicht die Struktur des Projektes:



Im Root-Verzeichnis des Projektes befindet sich zudem die Datei *package.json*, die alle für die Webanwendung benötigten Module als Abhängigkeiten deklariert. Mit dem Paketverwaltungssystem npm können diese Module einfach und schnell eingebunden werden.

Eine serverseitige MySQL-Datenbank speichert nach erfolgreicher Registrierung eines Benutzers dessen Daten ab. Die ThreeJS- und CannonJS-Instanzen laufen lokal bei dem Client. Während des Spiels kommunizieren die Clients und der Server durch Sockets, um so den Stand des Spiels bei allen Clients zu synchronisieren.

Ablauf

Im Folgenden soll die Funktionsweise der Anwendung durch einen beispielhaften Ablauf dargestellt werden:

Ein Benutzer öffnet die Website des Spiels. Automatisch werden alle für das Spiel benötigten Ressourcen geladen und der Benutzer als Beobachter in das Spiel integriert. Bevor er sich registriert, kann er also schon anderen Spielern zuschauen. Bei erfolgreicher Registrierung wird eine E-Mail an den Benutzer geschickt und die Daten in der Datenbank gespeichert. Meldet sich der Benutzer nun an, so werden die eingegebenen Daten mit denen aus der Datenbank verglichen. Passen die Datensätze zusammen, so wird der Benutzer als Spieler dem Team, das weniger Spieler besitzt, hinzugefügt.

Da diese Zustandsänderung alle Spieler betrifft, muss der neue Stand des Spiels mit allen Spielern synchronisiert werden. Dies geschieht durch Socket-Events. Der neue Spieler bekommt die Daten aller aktiven Spieler und integriert diese in seine Spieldateien, wobei alle anderen Spieler lediglich den neuen Spieler integrieren müssen. Jetzt ist der Stand des Spiels bei allen Spielern wieder gleich. Um dies auch fortlaufend gewährleisten zu können, müssen jetzt alle Zustandsänderungen des neuen Spielers, wie z.B. Änderungen der Position und Rotation der Spielfigur über die Socket Schnittstelle an alle anderen Spieler geschickt werden. Andersherum müssen auch alle Änderungen der anderen Spieler auf die lokale Spielinstanz übertragen werden. Das Aktualisieren dieser Daten geschieht pro Frame genau einmal und synchronisiert die Spieler dadurch miteinander.

Beim Schießen von Kugeln wird ähnlich vorgegangen. Der Benutzer drückt die linke Maustaste und es wird der Richtungsvektor vom Kamera- in den Weltraum berechnet. Dieser wird zusammen mit der Spieler-Id an alle weiteren Spieler übertragen und löst dort das Schuss-Event aus. Wie bei dem Spieler, der an einer bestimmten Position im Weltraum in eine bestimmte Richtung geschossen hat, so wird lokal bei den anderen Spielern auch eine Kugel mit genau diesen Werten erzeugt. Durch die Physics Engine CannonJS verhält sich die Kugel bei jedem Spieler gleich. Wird ein Spieler von einem Gegenspieler des anderen Teams getroffen, so verliert dieser Lebensenergie, oder stirbt. Bei der Kollision zwischen Kugel und Spieler kann durch die Spieler-Id zwischen Freund oder Feind differenziert werden und durch Übergabe an den Server, bei einem Tod entschieden werden, wer den „Kill“ gutgeschrieben bekommt.

Stirbt ein Spieler, so erscheint er mit voller Lebensenergie in seiner Basis wieder und das gegenerische Team bekommt einen Punkt gutgeschrieben. Bei Verlassen eines Spielers wird ein Socket-Event bei allen anderen Spielern ausgelöst, das anhand der richtigen Id alle Ressourcen dieses Spielers löscht.

Nachfolgend soll eine Liste mit verwendeten Socket-Events und vereinfachtem Code die Multiplayer-Funktionalitäten verdeutlichen:

io.on('connection', function(socket)

{

socket.on('new user', function(username)

{

socket.emit('allplayers', getAllPlayers() );

socket.broadcast.emit('new user', socket.player);

socket.broadcast.emit('chat message', message );

});

socket.on('move', function(moveData)

{

socket.broadcast.emit('move', moveData);

});

socket.on('movingstate', function(state)

{

socket.broadcast.emit('movingstate', data);

});

socket.on('shoot', function(shootData)

{

socket.broadcast.emit('shoot', shootData);

});

socket.on('score', function(score)

{

io.emit('score', score );

});

socket.on('chat message', function(msg)

{

io.emit('chat message', msg);

});

socket.on('disconnect',function()

{

socket.broadcast.emit('remove', player.id );

socket.broadcast.emit('chat message', player.message );

});

});

Visualisierung

Zur Darstellung aller im Spiel sichtbaren Elemente wurde ThreeJS (eine Javascript Bibliothek) als Schnittstelle zwischen unserer Applikation und WebGL genutzt. ThreeJS ermöglicht es schnell und einfach 3D-Modelle in Javascript zu importieren, zu animieren und mit Licht, Schatten und weiteren Effekten direkt im Browserfenster anzuzeigen.

Alle Modelle wurden in 3ds Max Modelliert, Texturiert und Animiert.

Für die bestmögliche Kompabilität zwischen 3ds Max und ThreeJS wurden verschiedenste Exporter und Loader(Importer) getestet, wobei der von "cgdev.org" für 3ds max entwickelte json Exporter in kombination mit dem "JDLoader" als Importmodul für ThreeJs die besten Ergebnisse lieferte.

Animation

Dank des JDLoaders können Animationen direkt aus dem Importierten Mesh in ThreeJs geladen werden,

var mixer = new THREE.AnimationMixer(mesh);

und z.B. für die walkcycle Animation des Charakters mit play() und pause() beliebig pausiert werden.

Damit alle Animationen während des Spielsverlaufs richtig ausgeführt werden können, wurden alle "AnimationMixer" in ein Array gefasst und in der update() funktion auf den richtigen Stand gesetzt.

for (var i = 0; i < Game.mixers.length; ++i)

{

Game.mixers[i].update(delta);

}

CannonJS

Wie bereits erwähnt, wurde für Collisionen wie z.B. zwischen Spieler und Wänden CannonJS verwendet.

Für eine bestmögliche Performance und fehlerfreie Physikalische berechnung, haben wir auf konvexe und konkave Kollisionsobjekte verzichtet und alles auf primitive Geometrien beschränkt. Für die Umsetzung wurden deshalb die Wände und andere Collisionsobjekte des Levels in 3ds Max durch Quader nachempfunden.

Da Körper in CannonJS wie folgt erzeugt werden müssen:

new CANNON.Box( new CANNON.Vec3(länge,breite,tiefe) );

benötigt man also Höhe, Länge, Breite, Position und Rotation eines jeden Quaders.

Leider werden beim voraus erwähnten json Exporter nur die absoluten Koordinaten eines jeden Objektes exportiert, sodass Rotation und Position jeweils einen Nullvektor aufweisen.

Die Rückrechnung wäre ein Unnötiger Aufwand, da die benötigten Werte in 3ds Max ja ohnehin schon bestehen.

Deshalb wurden wieder verschieden Exporter und Loader getestet, und in den importierten Objektstrukturen nach eben diesen Werten gesucht.

Der OpenColladaExporter, welcher Daten ebenfalls im Json Format speichert, tut genau das. Nun werden also in der Funktion 'createPhysX(collada);' die Scheitelpunkte, Position und Rotation eines jeden Objektes extrahiert und die Scheitelpunkte durch Min() und Max() Operationen zu Längen-, Breiten- und Höhen-werten weiterverarbeitet.

Spezielles

Charakter

Der Charakter wurde als einzelnes Model Exportiert, um diesen im Spiel unabhängig vom Level zu bewegen. Im Spiel setzt sich der Charakter aus folgenden Elementen zusammen:

-Model

-Kamera

-SphereBody

-SphereBody2

Wobei es sich bei den SphereBodys um CannonJs Kugeln handelt.

Wenn ein Charakter sich bewegt, bestimmt zunächst die Rotation der Kamera, welche durch die PointerlockControll klasse berechnet wird in welche Richtung bewegt werden soll. Nun wird der SphereBody, welche den Gesetzen der Physik unterliegt in eben diese Richtung bewegt. Falls dieser mit einem Objekt kollidiert, wird diese Bewegung eingeschränkt.

Nun wird die Position des SphereBodys auf alle anderen Elemente des Charakters rückübertragen.

Beide SphereBodys sind ausserdem noch zuständig für die Kollisionsüberprüfung mit geschossen.

Da der Charakter nicht Kugel- sondern vielmehr Kapselförmig ist, liegt der SphereBody2 immer um den Radius verschoben über dem SphereBody um auch dort auf Kollision mit geschossen melden zu können.

Das Model wird nur den Gegnerischen Charakteren angehängt, da es sonnst die freie Sicht aus der Egoperspektieve stören würde.

Wenn ein Gegnerischer Charakter anfängt oder aufhört zu laufen(benachrichtigung über Socket siehe oben) wird die Animation über den Animationmixer gesteuert.

Fahrstühle

Eine kleine Herausforderung, zusätzlich zum Import und Export der Modelle, stellten die Animierten Fahrstühle dar. Diese wurden im Level Animiert Exportiert. Da die CannonJS Objekte allerdings wie oben beschrieben über Parameter erstellt werden, weisen diese keine Animationen auf. Um nun eine Synchrone Animation zwischen Visuellen Objekten und den Kollisionsobjekten zu erhalten, wurde die y Position des Kolliderobjekes wir folgt in der update() Funktion angepasst.

//Update Elevator Position Collider <=> Model

if(Game.elevatorR&&Game.elevatorRCol){

var a=Game.elevatorR.position.y;

a+=(Game.elevatorRCol.aabb.lowerBound.y);

Game.elevatorRCol.position.y=a;

}

if(Game.elevatorL&&Game.elevatorLCol){

var a=Game.elevatorL.position.y;

a+=(Game.elevatorLCol.aabb.lowerBound.y);

Game.elevatorLCol.position.y=a;

}

Die Addition des lowerBound Wertes hängt mit der relativen Verschiebung des Drehpunktes(Pivots) zusammen, worauf hier aber nicht weiter eingegangen werden soll.