Architekturdokument

Buddler Joe

May 19, 2019

Inhalt

1 Einleitung

1.1 Aufbau und Zweck des Systems

Dieses Dokument beschreibt die Software-Architektur des Spiels Buddler Joe, welches in einer Client-Server-Architektur als online Multiplayer Spiel gespielt werden kann.

Es sollen sich beliebig viele Spieler im Spiel anmelden können und diese sollen unabhängig voneinander Spiele auf einem Server spielen können. Weiter soll das Spiel entweder als Client oder als Server gestartet werden können. Dadurch ermöglichen sich lokale sowie globale Spiele. Weiteres zur Netzwerkstruktur im Abschnitt Net.

Um den Spielern eine abgeschlossene Spielumgebung mit ihren Mitspielern zu ermöglichen, sollen die Spieler in Lobbies unterteilt werden. Diese Lobbies steuern dann auch den Spielablauf und die gesamte Spiellogik eines Spieles. Mehr dazu im Abschnitt Lobbies.

Das Spiel selber soll über ein General User Interface (GUI) spielbar sein und alle funktionen des Spiels sollen über das GUI erreichbar sein. Dazu haben wir uns für die in openGL üblichen Stages entschieden. Weiteres dazu im Abschnitt Stages.

Das Spiel soll über klare Regeln verfügen, welche auch vom Server überprüft und nötigenfalls umgesetzt werden sollen. Dadurch wird ein Schiedsrichtes nötig, welcher Serverseitig die Regeln bei allen Spielern kontrolliert. Weiteres heirzu im Abschnitt Schiedsrichter, der bei den Lobbies erwähnt wird.

Um das Spiel angemessen spielen zu können und grafisch in 3D darstellen zu können, wird es schnell nötig, sich eine Engine zu erstellen, mit denen man die Spielwelt und alle Objekte darin darstellen kann. Dazu mehr im Abschnitt Engine.

2 Net

Der ganze Net Abschnitt spielt bei einem Multiplayer-Spiel natürlich eine grosse Rolle. Nur mit einer klar definierten Client-Server-Struktur funktioniert eine Reibungslose Kommunikation zwischen dem Server und dem Client. Die Netzwerkkommunikation funktioniert bei unserem Projekt primär aus einer Server- und Clientlogic, welche via Sockets miteinander kommunizieren.

2.1 Serverseite

Der Server erstellt dabei für jeden Client auf dem Server einen eigenen Clientthread, welcher die Kommunikation von diesem Spieler mit dem Server darstellt. Dabei ist zu beachten, dass die Threads parallel laufen und es deshalb zu threadingproblemen bei Serverseitig gespeicherten Informationen wie dem Highscore kommen kann. Um dies zu verhindern wurden in der Entwicklung immer threadsichere Datenstrukturen und Datentypen angewendet.

Der Server hat eine Spielerliste mit allen Spielern und deren Client Ids, welche bei jedem Client individuell ist. In dieser Liste werden die jeweiligen NetPlayers abgespeichert mit den notwendigen Informationen. Zusätzlich hat der Server im Net Packet noch andere Speicherklassen instanziert, wie der Highscore oder die History.

Grundsätzlich erledigen die Packete auf Serverseite ihre Arbeit via Methodenaufrüfen in den jeweiligen Klassen. Dadurch wird ein betrachtlicher Teil der Gamelogik in die Packete ausgelagert, was auch deren Anzahl und komplexität erklärt. Dadurch behält man als Entwickler jedoch den genauen Überblick, wo was gemacht wird. Dadurch erledigt der Server nicht viel mehr Arbeit, als die Packete korrekt entgegenzunehmen und zu erstellen. Der Rest wird meist von den Packeten selber erledigt. Auch Klassen, wie zum Beispiel der Referee (dazu im Abschnitt Schiedsrichter bei den Lobbies mehr) oder Serverblocks erledigen selber, ohne das zutun der Packete keine Arbeit.

2.2 Clientseite

Beim Client gilt eigentlich das selbe wie beim Server. Die gesamte Netzwerklogik wird beim Client auf einem separaten Thread ausgeführt, dies um nicht in Konflikt mit der Grafischen Darstellung von OpenGL auf dem Mainthread zu kommen. Wenn die Netzwerklogik auf dem gleichen Thread laufen würde, würde dies durch das Warten auf Packete den Mainthread blockieren und somit die gesamte Grafik blockieren. Dadurch können die Packete beim client auch nicht direkt auf das grafische Interface zugreifen und müssen dies über Hilfsmethoden oder über inderekten Weg machen.

Dadurch ist zwar weiterhin ein grosser Teil der Spiellogik in den Packeten veranktert, jedoch ohne direkten Einfluss auf das grafische zu haben. Zum grafischen mehr im Abschnitt Stages und Engine. Wie auch bei der Clientseite gibt es zwar auch hier grundsätzlich unabhängige Klassen wie den Pingmanager, welcher von selber merkt, wenn die Verbindung unterbrochen wurde. Jedoch gilt auch hier, dass die Klasse ohne ein dazugehöriges Packet keinen Nutzen erfüllen würde.

2.3 Netzwerkprotokoll

Das Netzwerkprotokoll wird in diesem Dokument nicht weiter besprochen, da dies schon im Netzwerkprotokoll-Dokument umfassend erläutert wurde. Wichtig ist hier nur, dass die vorherig genannte Arbeit der Packete jeweils in der processData() Methode erfolgt und die Validierung grundsätzlich immer ausgeführt wird um sicher zu gehen, dass ein zu verarbeitender Datenstring korrekt ist.

2.4 Ping-Manager

Der Ping-Manager kümmert sich sowohl beim Server, wie auch beim Client, um die Konstante Verbindung zum jeweilig anderen Netzwerkteilnehmer. Der Ping Manager erstellt ein Mal pro Sekunde ein Ping Packet und wartet dann auf die Antwort des anderen Teilnehmer durch ein Pong Packet. Übersteigt die Wartezeit eines Antwortpackets zehn Sekunden, so wird dieser Client oder Server als Disconnected erachtet. Dabei wird der Spieler auf Serverseite aus allen Listen gelöscht und vom Spiel entfernt und auf der Clientseite wird der Spieler in die Settings geleitet, wo er sich wieder verbinden kann.

3 Stages

Stages spielen eine wichtige Rolle in der Strukturierung des Programmcodes bezüglich des GUI, jede Stage-Klasse entspricht einem Menu. Die Klasse „Game“ führt dabei eine List über alle gerade aktiven Stages (Menus).

3.1 Wichtigste Methoden und deren Aufrufe

Hier wird kurz allgemein auf die wichtigsten Methoden der verschiedenen Stage Klassen eingegangen. Der Schwerpunkt soll dabei auf deren Integration in den gesamten Code liegen und nicht auf deren exakten Implementation.

3.1.1 init()

Jede Stage Klasse hat eine Methode „init()“ diese wird bei Programmstart in allen Stages ausgeführt und ist dafür zuständig, sämtliche Texturen der jeweiligen Stage zu laden. Dadurch wird sichergestellt, dass zu keinem späteren Zeitpunkt mehr Verzögerungen durch Laden entstehen können. Aufgerufen wird die „init()“-Methode also nur einmal pro Programmausführung und zwar von der Methode „loadGame“ der Game klasse.

3.1.2 update()

In diese Methode springt der Gameloop in jedem durchlauf einmal, falls die Stage gerade aktiv ist. Hier wird eine Liste über alle anzuzeigenden GUI-Elemente geführt und verwaltet. Hier wird auch der Userinput vom Inputhandler entgegengenommen und darauf folgende Aktionen getriggert. Die wohl wichtigsten Aufrufe in dieser Methode sind hier die beiden „render“-Aufrufe ganz am Ende, diese lassen erst denn GuiRenderer und dann denn TextMaster alle aktuellen Grafiken anzeigen.

3.1.3 done()

Des Weiteren haben alle Stages die Text anzeigen auch zwingend eine Methode done() um diesen wieder aus der RenderListe des Textmaster zu löschen.

3.2 Stagewechsel

Um von einer Stage in eine andere zu wechseln (um von einem Menu in das andere zu gelangen), müssen folgende Schritte vollzogen werden. Die gerade aktive Stage muss aus der List der aktiven Stages der Game-Klasse entfernt werden und die gewünschte nächste Stage muss hinzugefügt werden. Zudem sollte die done()-Methode beider Stages aufgerufen werden um sicherzustellen, dass keine Texte im falschen Menu angezeigt werden (dies stellte lange ein Problem dar). Im darauf folgenden durchlauf des Gameloops wird nun die neue Stage „ausgeführt“ und so auch angezeigt.

3.3 Stapeln von Stages

Es ist möglich mehrere Stages gleichzeitig aktiv zu haben. Im Spiel ist dies beispielsweise mit den Stages „Playing“ und „GameMenu“ der Fall. Für die richtige Anzeige ist zu beachten, dass jene Stage welche die Methode update() zuerst ausführt und damit auch die „render“-Aufrufe, hinter der darauffolgenden Stage angezeigt wird. So können beliebig viele Stages übereinander gelegt werden. Jedoch sind dadurch auch bei allen aktiven Stages die Inputinterpretationen der update()-Methode aktiv, hier muss also genau überprüft werden in welchem Fall ein Input wirklich gewünscht ist.

4 Lobby

Die Lobby stellt eine wichtige Komponente für unser Multiplayergamer dar. Sie verfügt über wichtige Informationen über das Spiel und die Spieler. Ebenfalls steuert sie den Spielablauf und die gesamte Spiellogik eines Spiels.

4.1 Konstruktor

Nachdem sich ein Spieler dazu entschieden hat, eine neue Lobby zu erstellen, indem er den «create»-Button drückt und den gewünschten Namen eingibt, verschickt dieser ein PacketCreateLobby Paket an den Server. Der Server verarbeitet die Informationen des Pakets und erstellt folglich eine neue Lobby. Bei der Erstellung der Lobby, also einem Aufruf des Konstruktors, werden wichtige Information gespeichert. Die Lobby wird in der History unter «Open Lobbies» gespeichert. Nebst dem Lobbynamen, dem Ersteller der Lobby und der Mapsize nimmt die Lobby den Status «open» an. «Open» bedeutet, dass zu diesem Zeitpunkt Spieler der Lobby beitreten und aus der Lobby austreten können. Weiter wird eine neue Map generiert, welche später alle Spieler beim Rundenstart erhalten. Zuletzt wird eine Lobbyloop gestartet, welche durchgehend bis zum Schliessen der Lobby durchlaufen wird und die Informationen updatet.

4.2 Lobbyloop

Die run()-Methode enthält eine Schleife, welche wie vorhin erwähnt, vom Öffnen der Lobby bis zur Schliessung der Lobby aktiv ist. Die Lobby kann zwei weitere Status annehmen, «running» und «finished». Die Lobbyloop verhält sich entsprechend dem Lobbystatus.

- Befindet sich die Lobby im «open» Status, kontrolliert sie bei jedem Durchlauf die Anzahl Spieler in der Lobby und wann der letzte Eintritt erfolgte. Stand die Lobby über zwei Minuten leer, löscht sich die Lobby. Anderenfalls passiert nichts.

- Tritt die Lobby in den «running» Status über, indem alle Spieler den «Ready»-Button aktivieren, speichert sich die Lobby in der History unter «Lobbies Of Running Games» ab. Im «running» Status verwaltet die Lobbyloop in jedem Durchlauf das laufende Spiel. Immer wieder wird die Anzahl lebender Spieler kontrolliert. Falls sich

keine lebenden Spieler mehr im laufenden Game befinden, wird das Spiel beendet. Das Spiel wird ebenfalls beendet, indem ein Spieler als Erster 3000 Gold gesammelt hat. Im ersten Fall wird der Status intern von der Lobby aus auf «finished» gesetzt und es erfolgt eine Gewinnerauswertung. Im zweiten Fall wird der Status indirekt durch die ServerPlayer.increaseCurrentGold()-Methode auf «finished» gesetzt, wenn ein Spieler Gold aufsammelt und dabei einen Goldstand von 3000 erreicht. Die Lobby verwaltet ebenfalls eine Liste von gespawnten Items in der Map. Eine weitere wichtige Aufgabe der Lobby ist die Verwaltung von Referees für jeden Spieler in der Lobby. Indem jeder Spieler dem Server ein PacketLifeStatus verschickt, wenn er einen anderen Spieler bei einem Verlust oder bei einer Aufnahme eines Herzens gesichtet hat, leitet die Lobby die Nachrichten an den Referee des jeweiligen Spielers weiter. Jeder Referee sammelt Nachrichten bezüglich seines Clients. Er entscheidet, ob der Lebensstand seines Clients verändert werden darf oder nicht. Nachdem eine Entscheidung vom Server gefällt wurde, verschickt dieser ein PacketLifeStatus Paket an alle Clients mit dem entgültigen Lebensstand des betroffenen Spielers.

- Im «finished» Status löscht sich die Lobby aus den «running Games» und die Lobbyloop wird nicht mehr ausgeführt. Sie archiviert den Lobbynamen und den Gewinner der Runde, falls dieser tatsächlich 3000 Gold gesammelt hat und nicht indem alle Spieler besiegt wurden beziehungsweise beide Herzen verloren haben. Danach wird ein PacketEndGame an alle Spieler der Lobby geschickt, welches den Gewinner und die Spieldauer enthält. Dabei wird die aktive «Playing» Stage gelöscht und die «Gameover» Stage hinzugefügt. Damit alle Spieler nach dem Spielende in der gleichen Lobby landen, wird ein Transfer durchgeführt. Es wird automatisch eine neue Lobby mit denselben Spielern erstellt und allen Spielern, die sich nicht in einer Lobby befinden, ein PacketLobbyOverview geschickt, damit die Liste der offenen Lobbies upgedatet wird.

5 Map

Die Map für das Spiel erstellt der Server beziehungsweise die Lobby, indem sie eine ServerMap erstellt wird. Beim Rundenstart wird allen Spielern, die sich in der Lobby befinden, ein PacketBroadcastMap Paket verschickt. Dort wird die Map als Array verpackt. Beim Verarbeiten des Pakets (clientseitig) wird das Array, welches die ServerMap enthält, in der ClientMap.reloadMap() übernommen. Somit hat jeder Spieler die gleiche Map.

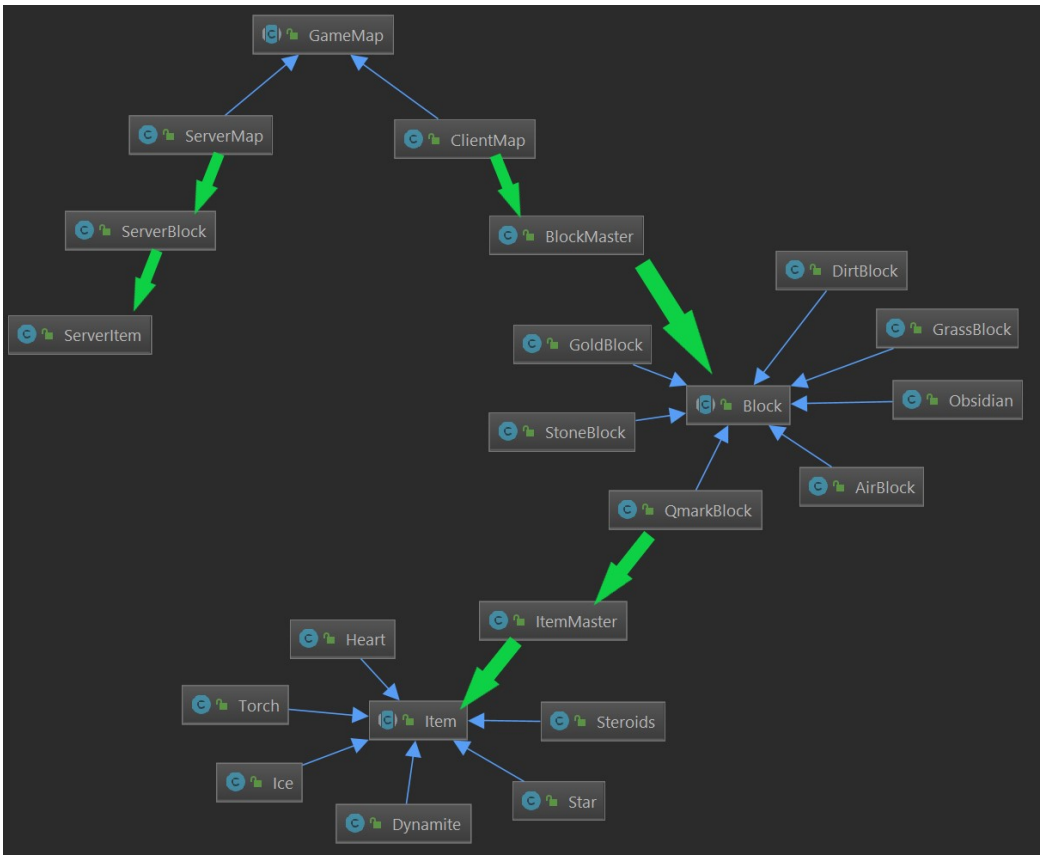

Figure 1: Zusammenhänge
Die blauen Pfeile stellen die Vererbung dar. Die grünen Pfeile visualisieren, wer durch wen aufgerufen wird.

5.1 Serverseite

5.1.1 ServerMap

generateMap()
Im Konstruktor der ServerMap-Klasse wird die Methode generateMap() aufgerufen. In dieser Methode wird ein 2-dimensionales Array erstellt. In das Array werden die SimplexNoise-Werte gespeichert. Danach wird ein neues 2-dimensionales Array erstellt, welches anhand der Werte im vorherigen Array Objekte der Klasse ServerBlock speichert.

5.1.2 ServerBlock

Eine Instanz der ServerBlock Klasse kann verschiedene Typen(Enum) haben: Obsidian, QMark, Gold, Grass, Stone, Air oder Dirt. Je nach Typ hat die Instanz eine gewisse «Hardness». Mit Hardness ist die Beständigkeit des Blocks gemeint. Durch mehrere Zerstörversuche kann ein Block zerstört werden. Die Hardness nimmt durch Zerstörversuche ab. Erst wenn die Hardness des Blocks kleiner gleich 0 ist, wird der Block zerstört. Wird ein ServerBlock zerstört, der vom Typ «Qmark» ist, wird zufälligerweise eines der Items generiert und es wird ein PacketSpawnItem Paket an die Clients in der Lobby verschickt. Beim Erstellen des PacketSpawnItem Pakets wird im Konstruktor das ServerItem gespeichert. Ein ServerItem Objekt enthält wichtige Informationen über das Item.

5.2 Clientseite

5.2.1 ClientMap

Bei der Verabreitung des PacketBroadcastMap Pakets auf der Clientseite wird die ServerMap, welche in einem Array gespeichert ist, übernommen. Dies geschieht in der ClientMap.reload()-Methode.

reload()
In der ClientMap.reload()-Methode werden über den BlockMaster die Blöcke(Abb. 1) generiert. Dabei werden die 3D-Koordinaten übergeben, um die Blöcke im GUI an der richtigen Position zu platzieren.
Erhält beispielsweise ein Spieler ein PacketSpawnItem Paket, so wird entsprechend dem Typ über den ItemMaster ein neuer Block generiert, welcher aber nun das neue Item enthält.

6 Master Klassen

Wir wenden das Konzept von "Master Klassen" konsequent an in unserem Projekt und erleichtern damit vor allem die Schnittstellen zwischen verschiedenen Projektkomponenten.

6.1 Was ist eine Master Klasse?

Jede Projektkomponente welche weitere, kleinere Teile unter sich zusammenfasst, kann von einer Master Klasse profitieren. In unserem Projekt haben wir für die meisten solcher Projektteile eine statische Klasse eingerichtet, welche **die untergebenen Einzelteile initialisiert, erstellt, organisiert, verwaltet, und löscht**. Dazu werden je nach Anforderungen die entsprechenden Methoden geschrieben und oft wird ein ENUM verwendet um die untergebenen Teile abschliessend aufzulisten.

Dies macht den Umgang mit Objekten sehr viel einfacher und weniger Fehleranfällig, da die Master Klasse einen grossen Teil der Arbeit übernimmt. Es bietet ausserdem eine Schnittstelle mit dem Programmteil, so dass von ausserhalb mit diesen Objekten gearbeitet werden kann, ohne dass man sich um deren Verwaltung kümmern muss.

6.2 Beispiel einer Master Klasse: BlockMaster.java

Am besten lassen sich Masterklassen an einem Beispiel illustrieren. Die BlockMaster Klasse ist für alle Blöcke in unserer Spielwelt zuständig und stellt die einzige Schnittstelle zu de n Blöcken dar. Der BlockMaster unterhält zwei Listen, mit Referenzen zu allen aktiven Blöcken:

1. Eine Liste mit allen Blöcken, unabhängig vom Typ
2. Eine HashMap mit einer Liste für jeden BlockTyp

Der BlockMaster implementiert folgende Methoden:

ENUM BlockTypes. Eine abschliessende Auflistung aller BlockTypen mit ID und ANSI-Repräsentation.

Initialisierung. Ruft die Init Methode für jeden BlockTyp auf. Die Blocktypen initialisieren i.d.R. ihre 3D Modelle und Texturen.

Block erstellen. Erstellt einen Block eines Typs unter Angabe der Welt und Grid Koordinaten. Dies speichert den Block direkt in beiden Listen welche vom Blockmaster unterhalten werden und fügt den Block der render liste hinzu.

Update. Diese Funktion wird jedes Frame ausgeführt während ein Spiel läuft und iteriert durch jeden Block. Blöcke welche als "zerstört" markiert wurden, werden aus allen Listen gelöscht und leere Listen werden entfernt. Anschliessend wird, falls nötig, die Position des Blocks neu Berechnet (etwa für fallende oder wackelnde Steinblöcke).

Get List. Vom Blockmaster kann eine aktuelle Liste aller Blocks angefragt werden um z.B. Kollisionsberechnungen zu machen

Clear. Löscht alle Blöcke aus allen Listen um ein neues Spiel zu starten.

Das Erstellen eines neuen Blocks in unserem Spiel - und somit die Schnittstelle zu den Blocks - ist über BlockMaster.generateBlock(Typ, Position) denkbar simpel. Auch das löschen via block.setDestroyed(true) wird komplett vom Blockmaster übernommen. In der SpielLogik müssen wir nie mit einem Block-Objekt direkt interagieren, ausser wir sind an den Eigenschaften eines Blocks interessiert.

6.3 Liste von Master Klassen

Folgende Projektteile sind mit Master Klassen organisiert:

- **Audio Master**. Der AudioMaster verwaltet die openAL BufferIDs vom allen Audio Dateien und hält diese sortiert und Kategorieren.
- **Particle Master**. Initialisiert Partikelsysteme und verwaltet alle einzelnen, durch die Systeme generierten Partikel.
- **Render Master**. Verwaltet alle renderbaren Objekte (entities) und Terrains. Bereitet die Shader vor, setzt openGL Parameter und sortiert die Objekte nach Typ (3D Modell), so dass Objekte mit dem gleichen Modell miteinander gerendert werden.
- **Text Master**. Verwaltet alle Text Objekte.
- **Block Master**. Verwaltet alle Block Objekte.
- **Item Master**. Verwaltet alle Item Objekte.
- **Light Master**. Verwaltet alle Lichtquellen und berechnet, wieviele und welche Lichtquellen gerendert werden.
- **Stage Master**. Die Game Klasse verwaltet alle Stage Objekte.
- **NetPlayer Master**. Verwaltet alle Netzwerk Spieler (nicht den aktiven, lokalen Spieler).

7 Game Engine

Für Buddler Joe haben wir basierend auf openGL eine eigene Game Engine entwickelt.

7.1 Dependencies

Wir benutzen die LightWeight Java Game Library (LWJGL: lwjgl.org) für sämtliche dependencies. Konkret stellt uns LWJGL low-level Bindings für openGL und openAL zur Verfügung, sowie Bindings für GLFW: Eine library für Fenster, Oberflächen und openGL/AL Kontext. Da LWJGL keine eigene Mathe-Library mehr mitliefert, haben wir uns für die empfohleme JOML entschieden um uns die Vektor- und Matrixberechnungen zu erleichtern und um keine Effizienz zu verlieren.

7.2 Objekt Parsing

Wir verwenden keine externen Libraries um Objekte (3D Modelle, Text, Terrains, GUIs, etc) in unser Spiel zu laden, sondern wir haben sämtliche Parser manuell geschrieben oder aus angegebenen Quellen übernommen und für unsere Zwecke angepasst. Einzig das in LWJGL enthaltene "stb" wird beim laden von Audio und Bild Dateien für einen Zwischenschritt verwendet. Dies ermöglicht sehr hohe Performance, da wir wenig bis gar keinen Overhead haben, jedoch macht es den Code Fehleranfällig und wir müssen uns selbst um die Optimierung kümmern, was uns sicher nicht immer gelungen ist. Eine Library dafür wäre wünschenswert gewesen, jedoch war keine auf der Whitelist.

Als weitere Schwierigkeit kommt hinzu, dass die geparsten Objekte oft direkt als Buffer an openGL übergeben werden müssen. Manuelle Umwandlung vom Objekten zu (Byte)Buffer war eine grosse Herausforderung.

7.3 Rendering

Wir unterscheiden zwischen fünf Kategorien von renderbaren Objekten:

- Text
- GUI/HUD Texturen
- Partikel
- Terrain
- Entities (3D Modelle)

Alle Kategorieren haben signifikant unterschiedliche Anforderungen. Terrain und Entities brauchen z.B. Lichtberechnungen, Partikel sind alles Quads, HUD Texturen brauchen nur Bildschirmkoordinaten, etc. Wir haben für jede Kategorie jeweils ein eigenes Shaderprogramm mit einem Vertex- und Fragmentshader geschrieben. Die Shader sind wie von openGL vorgesehen in GLSL programmiert und als Assets abelegt.

8 Player

Die Klasse «Player» ist hauptsächlich für die Bewegung der Spielfigur verantwortlich. Sie erweitert die Klasse «NetPlayer», jedoch verwendet sie keine Methoden der Netplayerklasse. Jeder Spieler besitzt Informationen über die eigene Spielfigur, sowie auch über die anderen Spielfiguren. Die eigene Spielfigur wird mittels Playerklasse bearbeitet, die anderen Spieler via der Netplayerklasse. Dies, weil der Spieler über seine eigene Spielfigur, im Vergleich zu den anderen Spielfiguren, mehr wissen muss. Die ganze Bewegung im Spiel wird mittels Orts-, Richtungs- und Beschleunigungsvektoren berechnet. So ist die Kollision mit der Umgebung der Figur klar geregelt.

8.1 NetPlayerMaster

Die Klasse «NetPlayerMaster» enthält alle netPlayer einer Lobby. Die Klassen «netPlayer» und «Player» werden mittels Pakete wie PacketPos oder Paket-Defeated aktualisiert. Somit ist der Spieler in der Lage Informationen über andere Spieler abzurufen.

8.2 Position Update

Der Server erhält jede Sekunde Daten über die Position der Spieler, welche er mit den Daten von anderen Spielern vergleichen kann. Da jeder Spieler Daten zu allen Spielern Daten schickt, hat der Server die Möglichkeit alle Daten zu vergleichen. So kann er Spieler ausfindig machen, welche betrügen wollen.

8.3 Items Update

Da die Items im Spiel eine grosse Auswirkung hat auf das Spiel, werden diese vom Server streng überwacht. Die Daten zu den Items werden von den Spielern an den Server geschickt. Der Server vergleicht die Daten und bestimmt die Folgerung daraus. Die Folgerung ist immer jene, welche die meisten Spieler behaupten. Spielen vier Spieler gemeinsam und ein Spieler schickt andere Daten als die übrigen drei, dann gelten die Daten, welche von den drei Spielern kommt. So wird verhindert, dass sich Spieler unerlaubt einen Vorteil ergaunern können.

List of Figures