Projektarbeit

Halo Live Stream

Marvin Hofmann,
Tobias Haslop

17.05.2013

Betreuer: Herr Hörnlein, Herr Loos

Inhaltsverzeichnis

Einleitung

Projektanlass

Im Rahmen der allgemeinen Hochschulreife ist es unsere Aufgabe eine Projektarbeit zu einem selbstgewählten Thema durchzuführen, wobei wir unser Ergebnis sowie unsere Projektarbeit dokumentieren sollen.

Als Anlass unseres Projektes nehmen wir den fehlenden Zuschauermodus in der PC Version des erfolgreichen First Person Shooters „Halo: Kampf um die Zukunft“.

Projektziel

Der Begriff Zuschauermodus bedeutet, dass der Zuschauer von einem dritten Gerät in der Lage ist, das Spiel aus verschiedenen Perspektiven zu verfolgen. Die Möglichkeit des Zuschauens wird, sofern überhaupt vorhanden, grundsätzlich nicht sehr häufig benutzt. Sie wird jedoch spätestens bei wichtigen kompetitiven Spielen, also z. B. beim Finale eines Turniers, zu einem äußerst nützlichen Feature.

Allerdings ist bei der Implementierung eines Zuschauermodus die Zuschauerzahl normalerweise limitiert.

Dieses Problem wird für gewöhnlich mit einem Video-Stream gelöst, der dann mittels einer eigenen Streaming-Lösung oder Streaming-Plattformen wie twitch.tv an eine Großzahl von weiteren Zuschauern verteilt wird.

Der Nachteil dieser Lösung ist, dass die Zuschauer von der Kameraführung des einzigen direkten Nutzer des Zuschauermodus abhängig sind und die Kamera nicht selbst nach Belieben führen können.

Als Projektziel wählen wir deshalb eine Streaming-Lösung für Halo zu entwickeln, die dem Zuschauer über eine Website eine selbst kontrollierbare Kamera bietet. Die Darstellung des Spiels soll allerdings nicht in vollem Umfang, sondern nur ansatzweise, erfolgen, da das Projekt sonst zu umfangreich und komplex geraten würde.

Fachlicher Teil

Streaming-Konzept

Unsere Lösung berücksichtigt die beim Streaming übliche Trennung zwischen Source-Server, Verteiler-Server und Client.

Der Source-Server entspricht in diesem Fall dem Halo-Server. Der Ansatz, um die Daten auszulesen erfolgt über ein Skript für die Halo-Server-Erweiterung „Phasor“ [http://haloapps.wordpress.com/], das einen Einsprungspunkt für unsere Programmierung bietet. In diesem Skript senden wir die Spieldaten zum Verteiler-Server

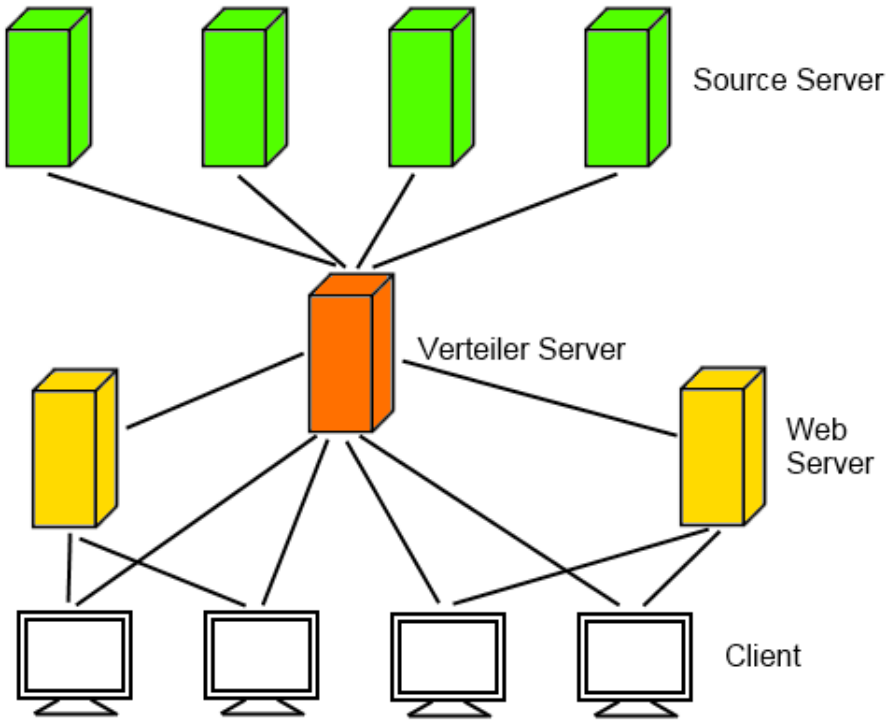
Für die Verteilung ist ein Java-Programm vorgesehen, das Verbindungen zu mehreren Halo-Servern führen kann, um die Spieldaten zu erhalten. Dieses Programm ist weiterhin dafür verantwortlich, eine Liste mit verfügbaren Spielen zu führen und auf Anfrage eines Clients die Spieldaten weiterzuleiten.

Als Client fungiert eine Flash-Anwendung, die die Aufgabe hat die Spieldaten darzustellen. Die Darstellung erfolgt unter anderem als dreidimensionale Szene. Außerdem bietet der Client eine Zeitleiste, mit der zu einem beliebigen Zeitpunkt im Spiel gesprungen werden kann.

Schließlich ist noch die Website zu erwähnen, welche die Liste von verfügbaren Spielen anzeigt und dem Flash-Client mitteilt, welches Spiel der Zuschauer gewählt hat. Außerdem erhält der Zuschauer den Flash-Client von der Website.

Die unten zu sehende Grafik stellt eine mögliche Situation dar. Bei dieser Darstellung sollte erwähnt werden, dass die einzelnen Teile unserer Lösung Programme sind, die nicht auf verschiedenen Rechnern laufen müssen, sondern rein theoretisch auch alle auf demselben.

Des Weiteren ist zu sehen, dass jeder Teil mehrfach vorhanden sein kann. Selbst mehrere Verteiler-Server sind möglich, jedoch ist in diesem Fall darauf zu achten, dass jeder Source-Server zu allen Verteiler-Servern sendet und die Verteiler-Server unabhängig voneinander agieren. Die Daten werden in diesem Fall redundant gespeichert.



Schnittstellen

Da unsere Lösung eine Verbindung zwischen den Nutzern (den Server-Betreibern und den Zuschauern) und dem Verteiler-Server voraussetzt, haben wir uns zwei Schnittstellen überlegt:

Der Verteiler hört per TCP auf Verbindungsanfragen von Halo-Servern. Sobald ein Halo-Server eine Verbindung aufnimmt und einige Daten über sich verschickt hat, wird dies in der Datenbank der Website gespeichert und Clients können auf das laufende Spiel live zugreifen. Der Halo-Server schickt nun die Spieldaten an den Verteiler, ohne eine Antwort zu erwarten. Da das asynchrone Auslesen der Antwort des Verteilers den Startprozess der Halo-Spiele zu stark verlangsamt hat, vertraut der Halo-Server darauf, dass seine Daten ankommen.

Der Client holt sich seine Daten auch über TCP, dabei muss er nach einem Handshake ein Spiel definieren, von dem er die Daten gesendet bekommen möchte. Die ID dieses Spiels bekommt er über die Webeinbindung übergeben, dies passiert dynamisch beim anklicken auf den Link eines Spiels. Der Verteiler-Server holt sich dann das Spiel aus der Datenbank (oder ein live laufendes) und schickt die Daten Zeilenweise zum Client. Da Flash eine asynchrone TCP-Verbindung verwendet und dies einige Probleme verursachte, haben wir eine Acknowledge-Antwort eingebaut, wo der Client alle 100 Zeilen eine Empfangsbestätigung senden muss. Dies macht die Übertragung zwar langsamer, hilft dem Client aber, nicht von Daten überrannt zu werden und ist in unseren Tests nicht negativ aufgefallen. Bei Spielen, die live sind und die schon einige Zeit laufen, ist es für den Client oft schwer, die Menge an Daten aufzuholen, gerade bei einer langsamen Verbindung. Dies ist der große Nachteil des Acknowledge-Packets. Da dies aber erst spät auftritt und die meisten Zuschauer einem Spiel beitreten, kurz nachdem es gestartet ist, ist dieses Problem zu vernachlässigen.

Da sich der Verteiler und die Website eine Datenbank teilen, war diese Verbindung fast schon automatisch dadurch realisiert, das php die Daten in der Datenbank ausliest, die von Java erzeugt werden. Für Live-Spiele wurde noch eine neue Tabelle in die Datenbank eingeführt.

Halo-Server

Etwas genauere Erklärung der Arbeitsweise bzw vor allem die Erklärung und Lösung von versch. Entscheidungsmöglichkeiten

Die Halo-Server-Erweiterung „Phasor“ [http://haloapps.wordpress.com/] ermöglicht es LUA-Scripts zu laden und ruft bei Spielereignissen die entsprechenden Funktionen in den Scripts auf, wie z. B. OnPlayerJoin. Relevante Daten können mit LUA direkt aus dem Speicher gelesen werden, da die Struktur der Speicherbelegung von Halo zu einem großen Teil bekannt ist.

Da LUA von Haus aus keine Möglichkeit bietet, Sockets zu öffnen, mussten wir uns in die Bibliothek LuaSocket [http://w3.impa.br/~diego/software/luasocket/] einarbeiten. Da der Halo-Server das Script aufruft, war es zudem noch schwierig, überhaupt die Möglichkeit zu schaffen, externe Bibliotheken einzuladen.

Verteiler-Server

Etwas genauere Erklärung der Arbeitsweise bzw vor allem die Erklärung und Lösung von versch. Entscheidungsmöglichkeiten

Der Java-Server muss gleichzeitig zwei verschiedene Arten von Verbindungen annehmen können, er besteht aus vier Teilen:

1. Der Verbindung mit den Halo-Servern, um die Spiel-Daten zu erhalten
2. Der Verbindung mit der Datenbank, um Spiele zu speichern und wieder zu laden
3. Der Verbindung mit den Flash-Clients, die sich die Spiel-Daten herunterladen können
4. Der Verbindung mit dem Webserver[http://halo.marv-productions.de], dem gespeicherte Spiele und verbundene Halo-Server mitgeteilt werden.

Eine Datenbank-Klasse übernimmt Teil 2 und 4, diese ist thread-safe programmiert, damit verschiedene unabhängig von einander agierende Flash-Clients/Halo-Server sich nicht in die Quere kommen.

Um die Flash-Clients mit Daten zu versorgen, hört ein Thread auf dem Port 33333 und verschiebt eingehende Verbindungen in eine eigens dafür geschriebene Klasse. Jede Instanz dieser Klasse läuft in einem eigenen Thread und verschwindet, sobald die Verbindung geschlossen wird. Wichtig dabei ist, dass alle Daten nur innerhalb der Instanz dieser Klasse referenziert werden, damit der Garbage-Collector diese sofort wieder freigeben kann.

Die Verbindung mit den Halo-Servern funktioniert ähnlich wie die der Flash-Clients auf Port 3333, wobei Clients statt auf ein gespeichertes Spiel auch auf das laufende Spiel des Servers verbinden können. Dabei muss der Server dafür sorgen, dass erhaltene Daten nicht nur gespeichert sondern auch unmittelbar (live) an alle Clients versendet werden.

Die an die Flash-Clients verschickten Daten werden in einer First-In-First-Out-Queue gespeichert, das heißt, dass die Zeilen in der richtigen Reihenfolge aus der Datenbank ausgelesen werden und in der gleichen Reihenfolge dann auch an den Client geschickt werden. Der Vorteil an dieser Art der Zwischenspeicherung entfaltet sich erst voll bei Live-Spielen, wo ein Teil des Spiels schon gespeichert ist, aber währenddessen konstant Daten dazukommen. Gelöst wurde das durch ein Spiegeln der schon vorhandenen Daten. Die Queue wird beim Live-Server registriert und neu ankommende Daten werden an das Ende der Queue geschrieben, während die alten Daten in umgekehrter Reihen-

folge an das vordere Ende angehängt werden. Dies verhindert jegliche Race-Conditions (Race-Conditions sind Effekte, die auftreten, wenn der Ausgang einer Operation davon abhängt, welcher Thread zuerst auf die gemeinsamen Daten zugreift), die normalerweise dadurch auftreten, dass während der kurzen Zeit, in der alte Daten gebuffert werden schon neue erscheinen, die wieder gebuffert werden und so weiter. Da die alten Daten zuerst kopiert werden müssen, entsteht eine kurze Wartezeit, da die Daten erst rausgeschickt werden können, wenn das Laden der alten Daten fertig ist, dieser ist aber höchstens Bruchteile von Sekunden groß und so gut wie nicht spürbar.

Ein großes Problem mit dem Verteiler ist der verbrauchte Arbeitsspeicher gewesen. Unser Root-Server hatte eine Limitierung zu ca. 100Mb Ram, was bei mehreren Clients, die Live ein Spiel anschauen sehr schnell belegt ist. Da sich das komplette Programm beendet, sobald zu viel Speicher gebraucht wird, ist bei jeder neuen Verbindung eine Prüfung eingebaut, die schaut, wie viel Arbeitsspeicher noch verfügbar ist und wie viel die jeweilige Verbindung ungefähr verbrauchen könnte. Sollte dieser Wert kritisch werden, wird die Verbindung abgelehnt. Dabei wurde entschieden, dass Streaming-Server eine höhere Priorität bekommen und deren Verbindung bevorzugt behandelt werden, damit Spieldaten nicht verloren gehen. Das kann bedeuten, dass Verbindungen von Clients abgebrochen werden oder langsamer werden, da deren zwischengespeicherte Daten stark reduziert werden. Spiele, die von mehreren Clients abgerufen werden, werden zentral gespeichert, damit keine redundante Speicherung erfolgt. Da dies bei Live-Spielen nicht möglich ist, wird da nur der bisher noch nicht gesendete Teil gespeichert und die Spieldaten des gesamten Spiels frühzeitig in die Datenbank geschrieben. Da ein Spiel in der Datenbank nur ein Eintrag ist und bei jedem Wegschreiben der Spieldaten der schon gespeicherte Teil nicht wieder in den Ram geladen werden soll, wird über eine SELECT-Anweisung der schon vorhandene Teil ausgelesen und direkt wieder per UPDATE den neuen Daten vorangestellt. Da beide Befehle in einer Transaktion laufen, bekommt Java von den verarbeiteten Daten nichts mit.

Trotz all dieser Vorkehrungen gelang es uns nicht zu verhindern, dass der Verteiler auch bei kleineren Lasten abstürzt. Ein zur Verfügung stehender Arbeitsspeicher von nur 100 Megabyte ist leider ab gewissen Datenmengen einfach voll. Würde unser Root-Server mehr Kapazitäten besitzen, würde dieses Problem sofort wegfallen.

Eingebundene Bibliotheken

- MySQL [http://dev.mysql.com/downloads/connector/j/]: Datenbank-Connector

Client

Um die Funktionsweise des Clients zu verstehen, der in der aktuellen Programmiersprache für Flash, Actionscript 3, geschrieben wurde, werden im Folgenden die Aufgaben einiger wichtiger Klassen beschrieben. Davor wird erläutert aus welchen Daten eine Spielsituation besteht und dann geht es um das Problem, wie die Spieldaten während des Abspielens gespeichert werden.

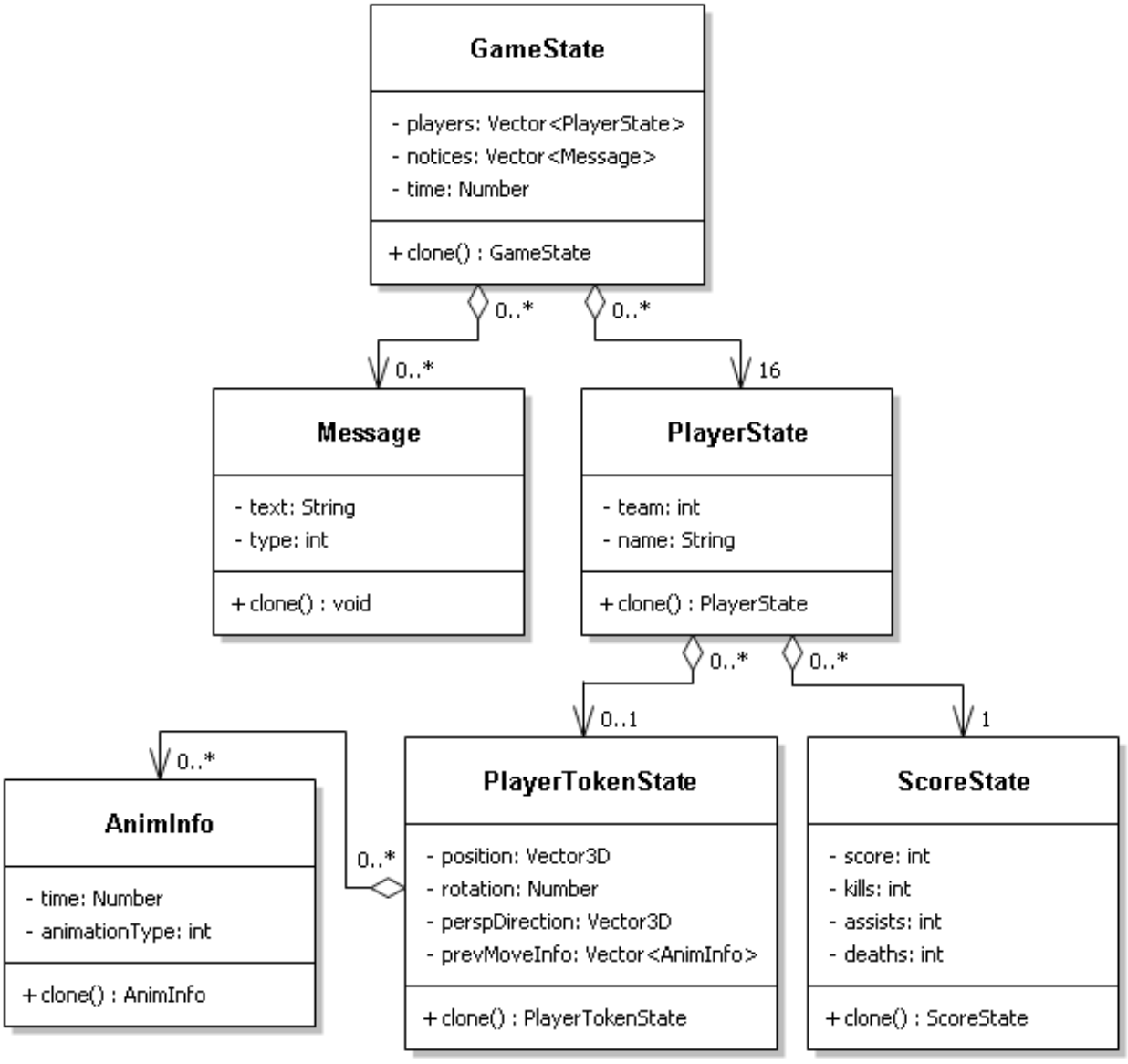
Repräsentation einer Spielsituation

Eine Spielsituation wird durch eine GameState Instanz repräsentiert, die einen Vektor von 16 PlayerStates beinhaltet. Als Index fungiert die machine id, die ein Spieler beim Beitritt des Servers erhält und die im Flash-Client als player id bezeichnet wird (nicht zu verwechseln mit der player id, die von Halo-Server Administratoren bei der Spieleridentifikation zum Serververbot genutzt wird und den Raum von 1-16 belegt, anstatt den Raum von 0-15).

Ein PlayerState beschreibt grundsätzlich einen Spielerzustand, der sich aus Name, Teamzugehörigkeit, Score und einem eventuell vorhandenen PlayerTokenState.

Ein nicht vorhandener PlayerTokenState bedeutet, dass der Spieler zwar mit dem Server verbunden ist, aber seine Spielfigur sich zur Zeit nicht im Spiel befindet, weil er beispielsweise kurz zuvor gestorben ist. Um den Animationszustand der Spielfigur korrekt darzustellen ist es außerdem nötig alle vorherigen Animationszustände und ihre Dauer zu speichern. Hierbei muss so weit in der Zeit zurückgegangen werden, bis die gespeicherte Dauer größer ist als die Animationsübergangsdauer.

Außerdem ist vorgesehen, dass jede Spielsituation eine Reihe von Nachrichten enthalten kann, wie z. B. Chatnachrichten oder Hinweise auf ein Spielereignis, die für eine bestimmte Dauer angezeigt werden und dementsprechend entweder Elemente des Vektors sind oder nicht.

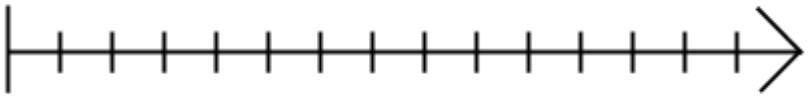


Format der Spieldaten im Speicher

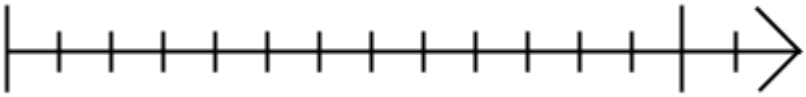
Um die Daten zu speichern gibt es verschiedene Möglichkeiten:

- Eine Liste, die alle Spielsituation in 33ms Abständen enthält (oder einer anderen geringen Dauer). Der Nachteil dieser Variante ist der eindeutig zu hohe Speicherverbrauch, da es theoretisch sehr lange dauernde Spiele mit sehr vielen Spielern geben kann.

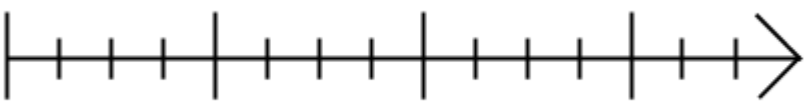
-
- Eine zweite speicherschonendere Variante ist nur die Änderungen der Spieldaten zu speichern und die Spielsituation aus den gespeicherten Änderungen zu berechnen.
  Allerdings soll der Client soll ungefähr alle 10 bis 50 Millisekunden ein Bild mit der aktuellen Spielsituation anzeigen und falls die Berechnung zu lange dauert, wird überhaupt kein Bild angezeigt.



- Der Ansatz der zweiten Variante kann weitergedacht werden: Die Berechnung dauert länger je mehr Änderungen zwischen dem Ausgangspunkt und dem zu berechnenden Zeitpunkt liegen. Der Ausgangspunkt war bei der zweiten Variante der Anfang des Spiels, da dieser bekannt ist: Es sind keine Spieler mit dem Server verbunden. Stattdessen kann man auch die zuletzt angezeigte Spielsituation als Ausgangspunkt für die Berechnungen nehmen. Da zwischen der zuletzt angezeigten Spielsituation und der als nächstes zu berechnenden Spielsituation in jedem Fall kaum Änderungen geschehen sind, wird die Berechnung sehr kurz dauern. Diese Variante lässt allerdings keine beliebigen Zeitsprünge zu, wie man sie bei einer Zeitleiste erwartet.



- Die vierte Variante, die auch in unserem Produkt implementiert wurde, ist eine Kombination aus den vorherigen Varianten: Grundsätzlich werden nur Änderungen gespeichert, aber um immer einen zeitlich nahe gelegenen Ausgangspunkt für die Berechnung zu haben, werden in bestimmten Zeitabständen vollständige Spielsituation gespeichert. Die Zeitabstände sind in diesem Fall natürlich deutlich größer als in der ersten Variante; In unserer Implementierung betragen die Abstände 1000 ms. Wird diese Zahl kleiner gewählt, wird mehr Speicher belegt. Wird die Zahl hingegen größer gewählt, steigt das Risiko, dass die Anwendung „einfriert“ da die Berechnung des Spielstands länger dauert. Diese Lösung ähnelt der Speicherung von Videos, bei der größtenteils Änderungen gespeichert werden, aber auch sichergestellt wird, dass in Abständen Keyframes vorkommen, die Situationen beschreiben.



Eine weitere Verbesserung wäre die Spielsituation zeitlich rückwärts zu berechnen, sofern dies mit den Daten vereinbar ist.

Außerdem ist noch zu erwähnen, dass die Speichereinsparung durch das Speichern von Änderungen statt Situationen durchaus variieren kann. Dies hängt davon ab, wie oft die gespeicherten Daten sich überhaupt ändern. Unsere Variante macht in Anbetracht von Daten wie den Score-Werten oder der Teamzugehörigkeit sehr viel Sinn. Wenn man allerdings nur Positionen der Spieler speichert, ist die Speichereinsparung im Vergleich zur 1. Variante bei einem aktiven Spiel gar nicht mehr bedeutsam.

Eingebundene Bibliotheken

- Away3D [http://away3d.com/]: 3D engine
- Robotlegs [http://www.robotlegs.org/]: Model View Controller Framework
- Robotlegs, Away3D Integration [http://www.paultondeur.com/2011/03/07/robotlegs-away3d-4-broomstick-integration-with-a-wink-at-unity/]
- Signals: Eine Alternative zu den üblichen Events [https://github.com/robertpenner/as3-signals]
- Robotlegs, Signals Integration [http://joelhooks.com/2010/02/14/robotlegs-as3-signals-and-the-signalcommandmap-example/] [https://github.com/Stray/robotlegs-utilities-SignalMediator]
- Robotlegs Variance Utility [http://guyinthechair.com/tag/robotlegs-variance-utility/]

Weitere wichtige Klassen

- InitAppCommand:
  Nachdem am Anfang einige Einstellungen für die Depenency Injection in der Context Klasse von Robotlegs vorgenommen wurden, wird die execute() Methode von InitAppCommand gestartet. Diese Klasse startet das Laden der Assets und der Spieldaten.
- Away3DAssetsService:
  Diese Klasse lädt die Assets und benachrichtigt den Rest der Anwendung über den Fortschritt des Ladens. Falls es bei der Ladeanweisung angegeben wurde, gibt diese Klasse den Assets einen Namen.
- LiveStreamService + LineParser:
  Diese beiden Klassen laden die Spieldaten über eine TCP-Socketverbindung vom Verteiler-Server und fügen sie dem GameReplay Service hinzu. Hierbei werden nicht zwangsläufig Live-Spiele übertragen, sondern auch gespeicherte Spiele sind möglich. Außerdem benachrichtigen sie den Rest der Anwendung über den Fortschritt des Ladens und Parsens.
- LogLoaderService + LogParser:
  Diese beiden Klasse lösen dieselbe Aufgabe wie LiveStreamParser und LineParser, aber in diesem Fall werden die Daten aus einer Datei gelesen und nicht über eine Socketverbindung übertragen.
- GameReplayService:
  Die GameReplayService Klasse bietet Funktionen zum Starten und Stoppen des Spiels, die Möglichkeit zu einem beliebigen Zeitpunkt zu springen, sowie eine Funktion zum Einfügen von Spielinformationen. Wenn das Spiel läuft wird der Rest der Anwendung bei der Berechnung von jedem neuen Bild mit der aktuellen Spielsituation versorgt. Hierzu besitzt die Klasse einen Vektor, der mehrere Spielintervalle enthält.

- GameInterval:
  Die GameInterval Klasse hält die Änderungen im Spiel innerhalb eines zeitlichen Abschnitts und die Spielsituation zu Beginn dieses Abschnitt. Mit getGameGameState(time) kann die Spielsituation berechnet werden.
- GameStateChange:
  Das GameStateChange Interface verlangt von implementierenden Klassen eine get-Methode für die Zeit, eine Methode, um sich selbst einem GameInterval hinzuzufügen und eine Methode, um die Änderung auf eine Spielsituation anzuwenden.
- PlayerStateChange:
  Bei diesem Interface verhält es sich ähnlich wie bei GameStateChange. Der Unterschied besteht unter anderem darin, dass die Methode zum Anwenden der Änderung einen PlayerState anstatt eines GameState als Parameter erwartet.
- LoadingScreen:
  Diese Klasse zeigt den Fortschritt beim Laden der Assets, der Spieldaten und den Fortschritt beim Parsen der Spieldaten an. Der Loadingscreen wird der Anzeigehierarchie hinzugefügt.
- ControlBar:
  Eine Instanz dieser Klasse wird nach dem Parsen der Spieldaten und Laden der Assets erzeugt. Die ControlBar zeigt den zeitlichen Fortschritt des Spiels an und besitzt Buttons um den Vollbildmodus umzuschalten und um das Spiel zu stoppen oder zu starten. Die ControlBar wird der Anzeigehierarchie hinzugefügt.
- Scoreboard:
  Zeigt eine Tabelle mit Score Informationen zu allen Spielern an. Das Scoreboard wird der Anzeigehierarchie hinzugefügt.
- SpectatorView:
  Diese Klasse zeigt die Szene an. Sie besitzt auch eine Spieler ID, die die Kameraposition und Perspektive bestimmt. Grundsätzlich wird die Ego-Perspektive des Spielers verwendet. Wenn die spieler id -1 beträgt, verändert sich die Perspektive durch Benutzereingaben. Die Steuerung entspricht dem bereits in Halo implementierten „debug camera“ Modus. Für eine Splitscreenansicht können mehrere Instanzen dieser Klasse erstellt werden. Die SpectatorView wird der Anzeigehierarchie hinzugefügt.
- GameMap3D:
  Diese Klasse repräsentiert die Spielkarte und erstellt zusätzlich 16 Cyborg Instanzen. Die GameMap3D wird der Szenenhierarchie hinzugefügt.
- Cyborg:
  Diese Klasse repräsentiert eine Spielfigur zugeordnet zu einer Spieler ID (der Name „Cyborg“ stammt aus den Namen der Assets). Ein Cyborg wird der Szenenhierarchie hinzugefügt. Die Klasse ist dazu in der Lage für einen beliebigen PlayerTokenState den dazugehörigen Status der Spielfigur darzustellen.

Ergebnis

Bedienung für Halo-Server Betreiber

Vorraussetzung ist das der Server die Erweiterung „Phasor“ benutzt. Falls diese Bedingung erfüllt ist, kann das Script mit dem Dateinamen livestream.lua in den von Phasor vorgesehenen Skripte-Ordner gespeichert werden (für gewöhnlich „%user%/Documents/My Games/Halo/scripts“).

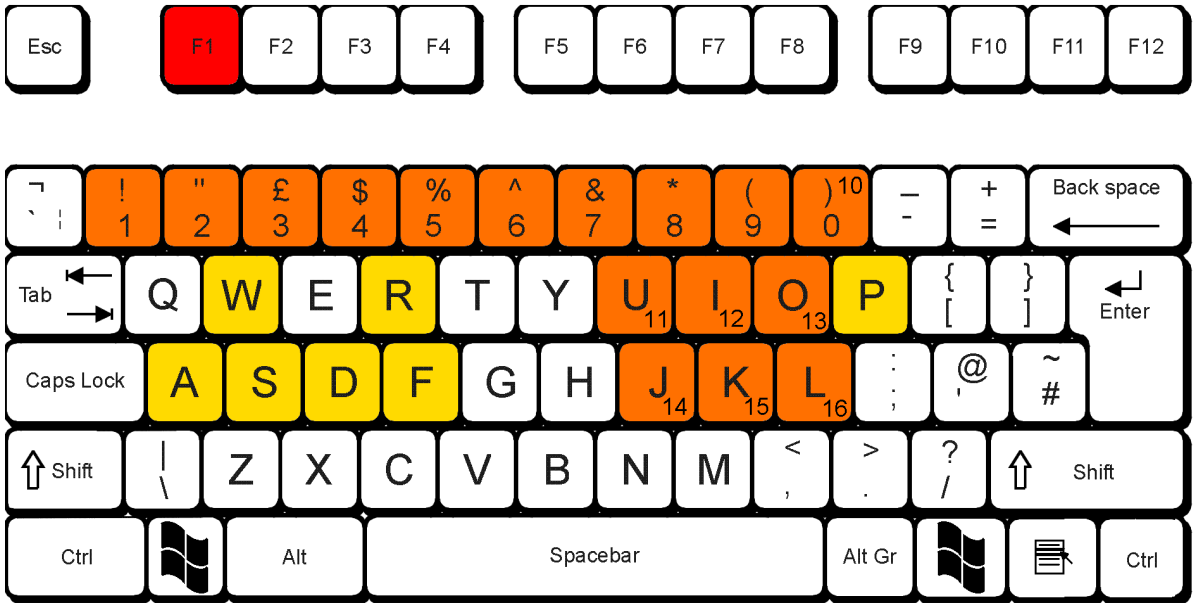
Um Socketverbindungen zu ermöglichen ist es des Weiteren nötig die Socketbibliothek für Phasorskripte einzubinden. Dazu müssen alle Dateien des libs Ordners aus unserem Paket in das Installationsverzeichnis des Halo-Servers kopiert werden.

Bedienung für Zuschauer

Zum Anschauen von Spielen besucht man die Website http://halo.marv-productions.de und wählt aus den dort aufgelisteten Spielen das gewünschte aus. Nachdem das Spiel geladen hat, kann man durch drücken der Play-, oder der Live-Taste das Spiel starten. Mit der F1 Taste kann zudem der Punktestand angezeigt werden.

Es gibt zwei verschiedene Zuschauerperspektiven:

- Ego-Perspektive eines Spielers: Um in diese Sicht zu wechseln, muss die Nummer des Spielers gedrückt werden. Die bis zu16 verschiedenen Spieler können mit den Tasten 1,2,3,4,5,6,7,8,9,0,u,i,o,j,k und l angesprochen werden. Falls sich kein Spieler mit der entsprechenden Nummer im Spiel befindet, wird ein schwarzer Bildschirm angezeigt. Wenn der Spieler zur Zeit nicht auf der Spielkarte repräsentiert wird, weil er etwa gerade gestorben ist, wird eine abgedunkelte Ansicht seiner letzten bekannten Position angezeigt.
- Freie Kamera-Perspektive: Diese Sicht kann mit der Taste p aktiviert werden. Die Steuerung erfolgt genauso wie die Steuerung der Debug_Camera, die von Map-Entwicklen in Halo verwendet wird. Das bedeutet, dass die Tasten a, w, d und s für die Vorwärts-, Rückwärts- und Seitwärtsbewegungen vorgesehen sind. Mit der Taste r steigt die Kamera und mit der Taste f sinkt die Kamera. Die Kamera kann mittels Verschieben der Maus bei gehaltener Maustaste gedreht werden.



Tastenbelegung des Clients

Systemvoraussetzungen

Der Flash-Client benötigt einen Computer, auf dem Flash installiert ist und einen Webbrowser. Dies sollte heutzutage so gut wie jeder Computer erfüllen.

Der Verteiler-Server stellt keine Anforderungen an die CPU, man sollte aber ca. 5Mb pro 10 Minuten Spiel plus 10% pro Zuschauer an Ram einrechnen. Ein Gigabyte Ram sollte für 50-100 streamende Server plus einige Zuschauer ausreichen. Ausprobieren konnten wir diese Werte nicht! Eine MySQL-Datenbank ist erforderlich.

Der Halo-Server benötigt Phasor und unser Script plus Bibliotheken.

Aufgabenverteilung

Grundsätzlich war Marvin für das LUA-Script verantwortlich, damit er die Schnittstelle zwischen Halo-Server und Verteiler-Server selbst definieren konnte, jedoch war Tobias fürs Lesen der Daten aus dem Speicher verantwortlich und hatte mehr Erfahrung mit dem Testen,der Funktionsweise und Besonderheiten von eingebundenen Scripts in Phasor.

Qualität des Produkts

Das entwickelte Produkt besitzt etliche Makel, die sich aus der niedrigen Zielsetzung ergeben. Diese schreibt nur eine ansatzweise Darstellung des Spiels vor.

Dies bedeutet unter anderem, dass wirklich nur die Basisanimationen der Bewegung (vorne, hinten, rechts links) sowie des Umschauens implementiert sind. Selbst von diesen Basisanimationen existieren bereits verschiedene Varianten für die verschiedenen Waffen die eine Spielfigur halten kann. Das entwickelte Produkt unterscheidet nicht zwischen diesen, sondern benutzt immer dieselbe Variante. Außerdem werden weder Projektile oder Granaten angezeigt noch die Waffen selbst.

Des Weiteren wird keine Schild- und Lebensanzeige der Spieler angezeigt. Auch fehlt die Szenerie, zu der z. B. große Steine und Bäume zählen, die nicht in den Daten der Spielkarte enthalten sind. Zusätzlich wird nicht zwischen den verschiedenen Spielmodi, wie „Capture the Flag“, „King of the Hill“ und „Race“ unterschieden, weshalb kein Score Wert im Scoreboard angezeigt werden kann. Auch wird nicht betrachtet, ob es sich um ein Teamspiel handelt oder jeder Spieler für sich selbst spielt. Das bedeutet, dass die Spieler keinen Farben zugeordnet werden können. Zudem werden bei der Darstellung der Spielkarte nicht die korrekten Lichtquellen verwendet und unsere gesamte Lösung funktioniert nur für eine der 19 verfügbaren Spielkarten.

Diese Liste an Makeln kann fortgeführt werden und liegt an der Komplexität der Aufgabe. Die Entwicklung eines Zuschauermodus sollte nämlich eigentlich zusammen mit der Entwicklung des Spiels erfolgen, da beim Zuschauermodus in den Bereichen der Darstellung und in einem großen Teil der Logik exakt dieselben Aufgaben bewältigt werden. Die entsprechenden Klassen sollten sinngemäß wiederverwendet werden, was die Entwicklung eines zusätzlichen Zuschauermodus stark vereinfacht.

Die von uns entwickelte Lösung basiert nicht auf der Wiederverwendung der bereits vorhandenen Spiele-Engine. Darunter leidet letztlich auch die Qualität unserer Lösung.

Ein weiteres Problem bei unserem Produkt ist die lange Ladezeit des Livestreams für ein bereits länger laufendes Spiel. Andererseits werden keine Daten gespeichert, wenn keine Personen auf dem Server sind und auch sonst erscheint unsere Lösung extrem speicherfreundlich verglichen mit Videostreams.

Uns ist beim Planen der Verbindung zwischen Verteiler-Server und Client der Fehler unterlaufen in jedem Fall die Daten von Beginn des Spiels an zu laden. Die langen Ladezeiten, die bei einem fortgeschrittenen Spiel entstehen können, bis man den Livestream sieht, hätten vermieden werden können.

Im oberen Bild ist ein Screenshot aus dem eigentlichen Spiel zu sehen. Bei einem Videostream verliert das Bild etwas an Qualität, aber hauptsächlich wird die Qualität von den Grafikeinstellungen des Aufnehmenden abhängen.

Im unteren Bild ist ein Screenshot von unserem Produkt zu sehen. Es bietet keine verschiedenen Grafikeinstellungen an und eine flüssige Wiedergabe in Blick auf grafische Berechnungen hängt ausschließlich von der Leistungsfähigkeit des Zuschauersystems ab.