Spacecraft - Projektbericht & Dokumentation

Stephan Soller, Benjamin Thaut, Michael Zügel

08.01.2011

Inhaltsverzeichnis

1 Aufgabenstellung

Das Ziel war es eine Game Engine mit zugehörigem Spiel zu entwickeln. Da der enorme Aufwand der hinter solch einem Projekt steckt allgemein bekannt ist, wurde entschieden das Spiel in einem Weltraum Szenario anzusiedeln, da es sowohl von den grafischen Inhalten als auch von der Programmierseite das Gerne mit dem geringsten Aufwand ist. Die anfänglichen technischen Ziele waren eine voll funktionsfähige 3D Engine, mit üblichen Features wie Echtzeitschatten, deferred lighting und Partikeleffekten. Außerdem sollte das Spiel auf eine kooperative Spielweise ausgelegt werden, die Zwischen mehreren Spielern über das Netzwerk erfolgt. Hierzu sollte das Spiel Multiplayerunterstüzung bieten, die jedoch nur auf das Lokale Netzwerk beschränkt werden sollte um nicht in die Verzögerungs- und Bandbreitenprobleme des Internets zu laufen. Inhaltlich geplant waren mehrere Schiffe: Eine Fregatte, einen Jäger, einen Munitionstransporter, und eine Raumstation. Als Waffen waren für die Fregatte eine Hauptwaffe sowie Flaks, Raketen und Maschienengewehere zu Jägerabwehr vorgesehen. Für den Jäger waren Maschinengewehere, Raketen und ein Gaussgewehr vorgesehen. Das kooperative Ziel des Spiels sollte es sein bestimmte Missionsziele zusammen zu erfüllen. Das geplante Szenario sollte wie folgt aussehen: Zwei feindliche Flotten mit mehreren Fregatten kämpfen gegeneinander, die Spieler sollten nun kooperativ einige Missionen erfüllen um ihrer Flotte zum Sieg zu verhelfen. Als Missionen waren unter anderem die Zerstörung der Schildgeneratoren von feindlichen Fregatten, sowie das eskortieren eigener Munitionsnachschübe geplant.

2 Probleme und Erfahrungen während der Projektarbeit

2.1 Allgemein

Während der Projektarbeit kam es vermehrt zu Missverständnissen zwischen einzelnen Teammitgliedern aufgrund von digitalen Verständigungsmitteln wie Messagern, und Online-Telefonie. Durch persönliche Treffen in der Hochschule konnten diese aber immer schnell gelöst werden. Meist halfen dabei Skizzen auf Whiteboards. Die Multiplatformentwicklung auf Windows und Linux erwies sich als sehr hilfreich, da dadurch schneller schwere Fehler gefunden werden konnten. Manche Fehler wurden nur von den Debugging Meachnismen unter Linux aufgedeckt, andere konnten besser unter Windows gefunden werden.

2.2 Zeitknappheit

Trotz der anfänglichen Planung bei der der nötige Aufwand für ein Spieleprojekt berücksichtigt wurde, konnten leider in der gegebenen Zeit nicht alle geplanten Funktionen und Inhalte eingebaut werden. So vielen die Missionsziele komplett weg, stattdessen wurde ein Deathmatch implementiert, bedingt durch den geringen dafür nötigen Zeitaufwand und der allgemein hohen Akzeptanz bei Spielern. Auch einige der Waffen vielen dem Zeitmangel zum Opfer. So haben die Jäger in der finalen Version nur Maschinengewehre und die Fregatten nur ihre Hauptgeschütze und Flaks. Der deferred Renderer konnte eben-

falls aus Zeitgründen nicht implementiert werden. Dafür ist das Spiel jedoch mit einigen Algorithmen zur Bandbreiteneinsparung bedingt über das Internet spielbar.

2.3 D 2.0 Fluch und Segen

Die Wahl der Programmiersprache D 2.0 entpuppte sich gleichzeitig als Fluch und Segen. In vielen Teilen des Entwicklungsprozesses konnte eine höhere Produktivität als mit C++ oder C erreicht werden. Dies ist unter anderem bedingt durch den eingebauten Garbage Collector und das allmächtige Template System mit dem viel repetitiver Code generiert werden kann, den man ansonsten von Hand schreiben müsste. So werden große Teile des nötigen Netzwerkcodes generiert wie später beschrieben. Die Anbindung von Funktionen an die Script-Engine wird auch generiert. Auf der anderen Seite ist die Toolchain von D 2.0 noch nicht ausgereift, Debugger liefern teilweise keine oder nur schlechte Ergebnisse. Zudem hatten wir mit Bugs in der Standardlibrary und der Runtime zu kämpfen. Einige Teile der Standardlibrary mussten deshalb kopiert und angepasst werden, andere waren gar nicht vorhanden und mussten komplett selbst geschrieben werden. Darunter fallen unter anderem alle Containerklassen (stack, vector, linked list, etc.) und manuelles Speichermanagement um den Garbage Collector zu entlasten. Ein Bug in der Runtime die alle Threads anhält bevor der Garbage Collector anfängt zu arbeiten führt unter Windows immer noch zum unregelmäßigen Absturz des Spiels. Selbst der Versuch den Fehler in der Runtime selbst zu beheben führte leider nicht zum Erfolg. Es ist weiterhin unbekannt warum der Fehler überhaupt auftritt, Schuld könnte eventuell sogar inkonsistentes verhalten der Win-API sein oder sogar ein Windows Kernel Bug. Auch der Compiler selbst hat noch den einen oder anderen Bug, wodurch auch einige Zeit auf Workarounds verwendet werden musste.

3 Dokumentation

3.1 Verwendete Bibliotheken

Assimp http://assimp.sourceforge.net/
: Importer liefert Rohdaten von den Modellen die dann verarbeitet werden und in internen Datenstrukturen gespeichert.

Lua http://www.lua.org
: Scriptsprache

SDL http://www.libsdl.org
: Zur Fenstererzeugung und zum Input-Handling

SDL image http://www.libsdl.org/projects/SDL_image/
: Zum laden verschiedener Bildformate (.jpg, .png, .tiff, etc.)

OpenGL http://www.opengl.org

3D Grafikkarten API

OpenAL http://connect.creativelabs.com/openal/default.aspx
3D Sound API

libvorbis http://xiph.org/vorbis
Zum laden von .ogg Musikdateien

TinyXml http://www.grinninglizard.com/tinyxml/
Zum laden und speichern von XML Dateien. Es wurde jedoch ein D port verwendet.

Freetype http://www.freetype.org/index2.html
Zum erzeugen von Buchstabengrafiken für das Textrendering

3.2 Nebenläufigkeit

Da aktuelle Computer fast immer einen Dual-Core wenn nicht sogar einen Quad-Core Prozessor besitzen wurde im Rahmen der zeitlichen Möglichkeiten darauf geachtet die Engine auf Multicoreprozessoren auszulegen. Dazu wurde der Renderer in einen eigenen Thread ausgelagert und mit dem Extraktor-Pattern an den Simulationsthread angehängt.

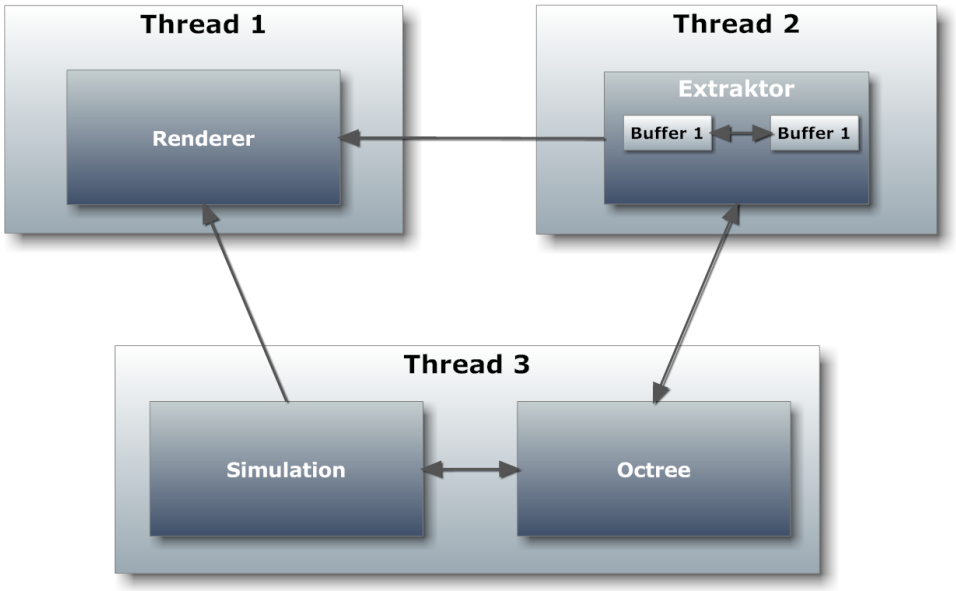


Abbildung 1: Threadunterteilung

Hierbei dient der Octree als zentraler Anlaufplatz. Im Octree sind alle für die Spielewelt relevanten Objekte hinterlegt. Der Extraktor hält sich zwei Buffer mit allen nötigen Daten für das renderern. Durch dieses Double-Buffering ist der Rendervorgang vom Extrahiervorgang entkoppelt sodass diese gleichzeitig stattfinden können. Als Nachteil entsteht

jedoch eine kleine zusätzliche Verzögerung bis der aktuelle Stand der Spielwelt auf dem Bildschirm erscheint. Diese ist allerdings so gering, dass sie von Menschen nicht bemerkt werden kann. Die Verzögerung beträgt 1 Renderframe was im Normalfall 16.6 ms entspricht. Mit diesem Setup laufen immer zwei Threads gleichzeitig, entweder der Renderer und die Simulation, oder der Renderer und der Extraktor. Dies führt dazu, dass wir einen Dual-Core Prozessor fast zu 100% auslasten können. Als zentrale Anlaufstelle bei diesem Pattern dient der Octree. Darin sind alle für die Spielwelt relevanten Gameobjekte abgelegt und der zu renderde Abschnitt der Spielewelt kann schnell erfragt werden. Durch diesen Ansatz, erhält man View-Frusturm-Culling kostenlos.

3.3 Tools

Zur schnelleren und effektiveren Entwicklung unseres Spiels wurden einige Tools in die Engine eingbaut. Für Programmierer unerlässlich ist eine ingame Konsole, die man zum Debuggen der Spielobjektzustände oder zum verändern der Spielwelt in Echtzeit verwenden kann. Ist es möglich für Änderungen sofort ein visuelles Feedback zu bekommen, verbessert dies die Entwicklungsgeschwindigkeit enorm. Zusätzlich wurden in den Renderer Möglichkeiten eingebaut vom jeder Stelle in der Engine debugging Ausgaben auf den Bildschirm zeichnen zu können. Dies umfasst unter anderem Linien, Boxen, Text, etc. Um langsame Funktionen im Programmcode besser finden zu können, wurde ein eigener Profiler implementiert, damit können im Programmcode einzelne Abschnitte markiert werden und dann in Echtzeit im Spiel mitverfolgt werden wieviel Zeit diese benötigen. Das Benutzen von normalen Profilern war leider nicht möglich, da diese zu viel Zeit für die Messungen benötigen, da sie in der Regel jeden Funktionsaufruf einzeln messen. Da es für unserer Grafiker unerlässlich war seine Modelle möglichst früh schon im Spiel sehen zu können, wurde außerdem eine spezielle Startoption für das Spiel implementiert, in welchem es nur ein einzelnes Model mit zugehörigen Texturen lädt und diese anzeigt. So konnte schnell festgestellt werden ob das Modell richtig exportiert wurde und ob es das erwünschte Aussehen im Spiel aufweist.

3.4 Scripting-Engine

Als Scriptsprache wurde Lua eingesetzt. Dazu wurden Tempaltes geschrieben die D Funktionen einfach und schnell an Lua anbinden können und die nötige Verbindung zwischen dem D und dem Lua Garbage Collector aufbauen. Dazu wird das Objekt immer beim D Garbage Collector allokiert. Beim Lua Garbage collector wird dann nur ein kleines Stück Speicher allokiert, dass dazu dient die Referenz auf das D Objekt abzulegen. Dieses Speicherstück wird dann beim D Garbage Collector als festes Root angemeldet. Gibt der Lua Garbage Collector das Speicherstück frei, wird durch eine Callback Funktion das feste Root beim D Garbage Collector entfernt.

Lua wird unter anderem für die Ingame Konsole verwendet, so kann von der ingame Konsole aus direkt der Zustand der Spielwelt verändert werden. Außerdem gibt es eine Set von Variablen die bei der Script Engine registiert sind. Über diese Variablen können

andere Teile der Engine kontroliert werden. Z.b. können debugging Ausgaben an und abgeschaltet werden und Einstellungen des Renderers geändert werden.

3.5 XML-Konfigurationsdatein

Viele Einstellungen werden in XML Dateien abgelegt, unter anderem die Einstellungen für Partikelsysteme und die Beleuchtungseinstellungen für den Renderer. Diese XML Dateien können während des Spiel verändert und neu geladen werden um die Auswirkungen sofort sehen zu können. Der hohe Aufwand zum Serialisieren und Deserialisieren von Daten wurde dabei umgangen indem mit D Templates der dafür nötigen Quelltext generiert wird. Es ist lediglich nötig im Quelltext eine Struktur zu definieren die alle benötigten Daten enthält. Diese Struktur kann dann mit einer einzigen Zeile Quelltext aus einer XML Datei eingelesen werden.

3.6 Spielwelt Datenstruktur

Zur Speicherung der Spielwelt wurde ein Octree gewählt, da besonders oft erfragt werden muss welche Objekte sich im Umkreis eines bestimmten Punktes befinden und diese Abfrage von einem Octree am effektivsten beantwortet werden kann. Um zusätzliche Implementierungskomplexität zu vermeiden und das Problem von Objekten zu umgehen, die immer im Wurzelknoten liegen, weil sie genau auf der Grenze zwischen zwei Knoten liegen oder zu groß sind, wurde ein loose Octree implementiert. Bei dieser Implementierung kann jedes Objekt maximal in einem Knoten enthalten sein, was den Verwaltungsaufwand verringer. Benachbarte Knoten überlagern sich um 50% sodass Objekte die genau auf einer Grenze zwischen zwei Knoten liegen würden, trotzdem in einen Knoten hineinpassen. Aufgrund der rießigen Spielwelt die für ein Weltraumspiel notwendig ist und der Tatsache das floats doch schnell ungenau werden umso weiter man sich vom Ursprung entfernt, haben wir einen eigenen Positionsdatentyp implementiert der die Position als Position in einer Gitterzelle sowieso den Offset zu dieser Gitterzelle speichert. Für die Gitterzelle verwenden wir short Werte wobei jede Gitterzelle einen km^3 darstellt. Damit können wir einen Gesamtraum von $65536^3 km^3$ abdecken. Sollte jemals mehr Platz erforderlich werden, können anstatt den shorts, ints oder sogar 64 bit ints verwendet werden.

3.7 Kollisionserkennung

Zur Implementierung von Kollisionserkennung sind erst einmal einige Matheklassen notwendig. Von uns implementiert wurde Bounding Box, Strahl, Ebene, Dreieck, 3er Vektor, 4er Vektor, 4x4 Matrizen. Anschließend mussten Algorithmen implementiert werden die die Überschneidung zwischen Strahl und Ebene, Ebene und Ebene, Dreieck mit Dreieck und Strahl mit Dreieck berechnen. Zusätzlich waren für die Boundingboxen Algorithmen für Überschneidungstests und Umschließungstests notwendig. Mit dieser Basis ist es dann möglich eine Kollisionserkennung vornzunehmen. Dazu speichern wir für jedes Objekt eine Boundingbox. Außerdem besitzt jedes Modell noch ein low-poly Kollisionsmesh. Für jedes

Objekt wird zuerst vom Octree erfragt welche Objekte sich mit der Boundingbox des Objekts das getestet werden soll überschneiden. Mit allen Ergebnissen dieser Abfrage wird dann ein genauer Test mithilfe des Kollisionsmeshes durchgeführt. Beide Kollisionsmeshes werden gegeneinander getestet indem alle Dreiecke mit allen Dreiecken des anderen Meshes auf Überschneidungen getestet werden. Wurde eine Überschneidung gefunden hat eine Kollision stattgefunden. Im Falle der Schüsse besitzen die diese kein Kollisionsmesh. Stattdessen wird ein Strahl mit allen Dreicken des anderen Meshes geschnitten und dann bei ausreichend naher Schnittdistanz eine Kollision eingeleitet.

3.8 Partikelsysteme

Jedes Partikelsystem produziert bei einem Extraktionsschritt alle nötigen Daten für den Renderer. Manche Partikelsystem simulieren tatsächlich eine bestimmte Anzahl von Partikeln, andere Partikelsysteme sind prozedural, und generieren die Partikel basierend auf einer Formel und der fortschreitenden Zeit. Als Partikeltypen existieren auf die Kamera gerichtet Partikel sowie, Richtungsgerichtete Partikel. Für Effekte wie Explosionen oder umherfliegende Trümmerstücke werden die zur Kamera gerichteten Partikel verwendet. Für die Schildeffekte und den Ring der riesigen Explosion werden die gerichteten Partikel verwendet. Die Partikel können dabei in zwei verschiedenen Modi gerendert werden. Der erste Modus rendert mit additivem Blending und erfordert keine Tiefensortierung. Der zweite Modus rendert mit alpha Blending und erfordert Tiefensortierung. Diese Tiefensortierung erfolgt allerdings aus Performancegründen nur Partikelsystemlokal. Überschneiden sich also zwei Partikelsysteme kann es zu Darstellungsfehleren kommen die allerdings in der Regel dem Betrachter nicht auffallen, außer er achtet gezielt darauf. Alle Partikelgrafiken sind in einer einzigen Textur gespeichert sodass alle Partikel in einem Schritt gezeichnet werden können. Durch die zwei verschiedenen rendermodi sind somit lediglich zwei Draw Calls nötig um alle Partikel zu zeichnen. Würden mehrere Texturen für die Partikel verwendet werden, würde die Anzahl der benötigten Draw Calls stark steigen.

3.9 Input handling

Da der Input nur in dem Thread abgefragt werden kann, dem der Fenstercontext gehört, müssen die Inputevents an den Simulationsthread weitergeleitet werden. Dazu wurden die Thread Messages der Standardbibliothek verwendet. Hierdurch entsteht eine kleine zusätzliche Verzögerung der Benutzereingaben die allerdings vernachlässigt werden kann.

3.10 Netzwerk-Architektur

Da Spiele relativ enge Echtzeit-Anforderungen stellen erfordern sie normalerweise auch entsprechend komplexe Netzwerk-Architekturen. Damit das Spiel möglichst schnell auf Benutzer-Eingaben reagiert biete sich z.B. eine verteilte Simulation der Spielwelt auf Basis des UTP-Netzwerk-Protokolls an. In so einer Peer-to-Peer Architektur würden alle beteiligten Clients ihre eigene Simulation berechnen und diese dann mit jeweils anderen

Spielern synchronisieren. Dadurch können Benutzereingaben unmittelbar in der lokalen Simulation des Clients angewandt werden wodurch der lokale Spieler keine Latenz wahrnimmt. Anschließend werden die Benutzereingaben an alle verbundenen Clients weiter verteilt damit diese sie in ihren Simulationen anwenden können.

Da jeder Client seine eigenen Simulation ausführt können jedoch für das Spiel relevante Ereignisse bei jedem Client unterschiedlich auftreten. Feuert z.B. ein Spieler während einer Bewegung eine Waffe ab so ist die Richtung und damit die Flugbahn der Waffe stark von dieser Bewegung abhängig. Da bei dem Client des Spielers die Eingabe, die zur Bewegung geführt hat, unmittelbar angewendet wird trifft er das Ziel. Bei anderen Clients, die diese Bewegung je nach Netzwerk-Latenz erst einige hundert Millisekunden später anwenden können, verfehlen die Geschosse jedoch ihr Ziel. Dieser Effekt ist besonders bei schnellen Rotationen von Spielern sehr stark bemerkbar. Problematiken wie diese machen es bei einer solchen Peer-to-Peer Architektur sehr komplex und aufwändig die einzelnen Simulationen möglichst konsistent zu halten.

Hinzu kommt, dass das UDP-Protokoll zwar die Netzwerk-Latenz möglichst gering hält, dafür allerdings sehr wenige Garantien liefert. So können z.B. gesendete Pakete verloren gehen oder auch in der falschen Reihenfolge beim Empfänger ankommen. Diese Effekte müssten bei der Kommunikation und Synchronisierung der Simulationen berücksichtigt werden und ggf. durch höhere Protokoll-Schichten ausgeglichen werden.

Auf Grund der engen zeitlichen Begrenzung wurde für das Spiel daher eine einfacherer Netzwerk-Architektur gewählt. Alle Clients senden ihre Eingaben zu einem zentralen Server. Dieser Server simuliert die Spielwelt und wendet alle Client-Eingaben auf diese Simulation an. Anschließend werden die resultierenden Änderungen der Simulation an alle Clients verteilt. Durch eine zentrale Simulation kann eine konsistente Spielwelt sehr einfach garantiert werden. Für die Kommunikation wurde statt UDP das TCP-Protokoll verwendet, dass sowohl den Empfang der Daten als auch die korrekte Reihenfolge der empfangen Daten garantiert. Beide Design-Entscheidungen erhöhen die Latenz von Benutzereingabe. Jedoch ist die angepeilte Umgebung das lokale Netzwerk und die niedrige Latenz und hohe Bandbreite in dieser Umgebung erlauben diese Kompromisse.

3.10.1 Infrastruktur für Kommunikation

Clients und Server können über zwei Arten miteinander kommunizieren:

Events Ereignisse wie z.B. Benutzereingabe oder die Zerstörung eines GameObjects werden in Form von *Events* übertragen. Dabei handelt es sich lediglich um eine ID sowie optionale Daten, die über das Netzwerk übertragen werden. Ein genauer Zeitpunkt für das Ereignis wird nicht festgelegt. Sobald das Event bei dem Empfänger ankommt wird es ausgelöst. Lediglich die Reihenfolge der Events wird durch die Verwenden von TCP als Protokoll garantiert. Events können von Client zum Server (z.B. Benutzereingabe) und vom Server zum Client (z.B. GameObject X zerstört) gesendet werden. Jedoch nicht von Client zu Client. Das wäre technisch zwar möglich, allerdings könnte ohne diese Einschränkung eine konsistente Spielwelt nicht garantiert

werden.

Synchronisierte GameObjects Auf dem Server können Variablen in Klassen als *netvar* markiert werden. Alle Änderungen an solchen Variablen werden nach einem Simulationsschritt an alle Clients verteilt. Zu jedem GameObject auf dem Server existiert auf jedem Client ein passendes lokales Gegenstück. Sobald ein Client Änderungen vom Server empfängt werden die Variablen seiner lokalen GameObjects mit den Werten vom Server überschrieben. Diese Art der Kommunikation erfolgt ausschließlich vom Server zum Client.

Um die Kommunikation effizient verwenden zu können werden große Teile des dafür notwendigen Quelltextes automatisch generiert. Hier ermöglicht das Compile-Time Metaprogramming der Sprache D aus Methoden-Signaturen und Aufrufen entsprechende Event-Strukturen und Funktionen zu generieren. Der Quelltext um die GameObjects anhand der `netvar`-Markierungen zu synchronisieren wird ebenfalls automatisch generiert. Einzig der Code um Datentypen wie int, float, Positionen, Rotationen, usw. in einen Datenstrom zu serialisieren und aus einem Datenstrom zu deserialisieren musste per Hand geschrieben werden.

3.10.2 Non-Blocking I/O

Um das Netzwerk-System gut in die Hauptschleife des Spiels zu integrieren werden Sockets im Non-Blocking I/O Modus verwendet. Da die Socket APIs von Windows als auch Linux auf der BSD Socket API basieren ist die Funktionsweise bei beiden Betriebssystemen identisch. Syntaktische Unterschiede (unterschiedliche Funktionsnamen und Fehlercodes, usw.) werden bereits durch die D Standardbibliothek vereinheitlicht.

Im Non-Blocking I/O Modus werden Daten nur gelesen, wenn in den Netzwerk-Buffern des Betriebssystems auch Daten vorhanden sind. Sind keine Daten verfügbar so kehrt die Lesefunktion sofort mit einem speziellen Fehlercode zurück. Dadurch wird das Spiel *nicht* angehalten bis Daten eintreffen. Auf die gleiche Art werden Daten nur gesendet wenn das Betriebssystem dazu bereit ist. Durch diesen Modus können Netzwerk-Daten sehr effizient verarbeitet werden sobald sie anfallen. Andernfalls wird die Hauptschleife des Spiels direkt fortgesetzt und ein weiterer Simulationsschritt wird berechnet.

3.10.3 Grober Aufbau

In der gewählten Architektur ergeben sich für den Server und die Clients grob folgende Tätigkeiten in der jeweiligen Hauptschleife:

Server

1. Sammelt Input aller Spieler (Events von Clients zum Server)

2. Wendet Input auf Spielwelt an (Verändert z.B. Beschleunigung eines Spielers

oder erstellt ein neues Projektil)

3. Simulationsschritt wird ausgeführt (Beschleunigungen und Geschwindigkeiten resultieren in veränderten Positionen, Kollisionen werden erkannt und gehandhabt, usw.)

4. Verursachte Änderungen werden an alle Clients verteilt (Events vom Server zu allen Clients, neue Werte für synchronisierte GameObjects)

Client

1. Falls Benutzereingabe vorliegt wird diese an den Server gesendet (Event vom Client zum Server)

2. Falls neue Daten vom Server vorliegen werden diese auf die lokalen GameObjects angewandt. Bei speziellen Events werden auch neue GameObjects erstellt. Die Auswirkungen der Benutzereingabe werden so in einem späteren Zyklus für den Client sichtbar.

3. Clientseitige Simulation wird ausgeführt (z.B. Clientside prediction)

Der Server selbst verfügt über keine Ausgabe der Spielwelt. Er verwendet weder einen graphischen Renderer noch eine Soundausgabe. Der Server bietet lediglich eine Konsole um das Spiel zu administrieren (z.B. eine Levelwechsel auszulösen oder das Spiel zu beenden). Die Teilung von Simulation (Server) und Anzeige (Client) erhöht nochmals den Grad der Parallelisierung des Spiels.

3.11 Gameplay

3.11.1 Kern-Mechanismen

Als Kern-Mechanismus des Spiels wurde ein möglichst vielen Spielern vertrautes System gewählt: Jeder Spieler verfügt über eine Anzahl an Hitpoints die bei jedem Treffer verringert wird sowie eine Waffe um Projektile abzufeuern. Sind null Hitpoints erreicht so wird der Spieler zerstört. Wie oft ein Spieler zerstört wird und wie viele andere Spieler er Zerstört wird protokolliert. Dadurch können sich Spieler miteinander vergleichen und eine Konkurrenz zwischen den Spielern entsteht. Diese einfache Konkurrenz-Mechanik bildet den Kern von *Deathmatch*.

Um die Spieler-Begegnungen interessanter zu gestalten wurden relevante Objekte in der Spielwelt mit zusätzlichen Schilden ausgestattet. Diese Schilde können eine gewisse Anzahl and Hitpoints absorbieren. Im Gegensatz zu normalen Hitpoints regenerieren die Schilde sich jedoch kontinuierlich. So muss ein Spieler zuerst die Schilde eines Gegners durchbrechen um seine Hitpoints verringern zu können. Dadurch erholen sich Ziele von gelegentlichen Zufallstreffern automatisch und ein Spieler muss eine Strategie entwickeln um seine Feuerkraft zeitlich auf ein Ziel zu konzentrieren. Andernfalls wird der Angriff

durch die sich ständig regenerierenden Schilde zu stark abgeschwächt.

Als Primärwaffe dient den Spielern eine schnell feuernde und stark streuende Projektilwaffe. Diese Waffe enthält ebenfalls einen Mechanismus um den Spieler zu mehr strategischen Vorgehen zu zwingen. Jeder Schuss baut Hitze auf. Diese Hitze baut sich kontinuierlich wieder ab, jedoch langsamer als die Waffe im Dauerfeuer Hitze aufbaut. Ist eine gewisse Temperatur erreicht überhitzt die Waffe und muss erst einige Sekunden abkühlen, bevor sie wieder verwendet werden kann. Dieses Gameplay-Element ergänzt die Schild-Mechanik und zwingt den Spieler dazu nur dann zu feuern wenn wirklich Erfolgsaussichten auf nennenswerte Treffer bestehen.

3.11.2 Teams

Um verschiedene Arten von Zusammenspiel abbilden zu können ist es oft notwendig Spieler und neutrale NPCs (Non playable characters) gruppieren zu können. So, dass z.B. NPCs einer Fraktion nur gegen NPCs einer anderen Fraktion kämpfen, aber beide NPC-Fraktionen keine Spieler angreifen. Diese Konstellation ist z.B. nützlich um in einem Deathmatch-Level ein "Hintergrund-Feuerwerk" aufzubauen. Eine allgemeinere Anwendung von Teams stellt z.B. das Team-Deathmatch (Team gegen Team) oder ein Kooperativ-Modus (Spieler gegen NPCs) dar.

Damit alle Modi möglichst einfach abgedeckt werden können kann jedes relevante GameObject einem Team zugeteilt werden. Dabei ist das Team eine Nummer von -127 bis 128. Alle der 255 verfügbaren Teams sind in drei Arten eingeteilt:

Team 0 Das Team mit der Nummer 0 nimmt eine besondere Stellung ein. Alle Spieler in Team 0 können sich gegenseitig, sowie alle anderen Teams angreifen. Dieses Team ist das *free for all* Team und wird z.B. für alle Spieler im Deathmatch verwendet. Im Scoreboard wird dieses "Team" grau dargestellt.

Team 1 – 128 Alle positiven Teams fungieren als Teams in herkömmlichen Sinne. Spieler im selben Team können nicht angegriffen werden (Schüsse prallen ab und verursachen keinen Schaden). Markiert ein Spieler ein GameObject des selben Teams so wird es im HUD grün dargestellt. Im Scoreboard wird das eigene Team ebenfalls grün dargestellt, während andere positive Teams rot angezeigt werden. Dieses Team wird z.B. bei kooperativen Leven verwendet bei dem alle Spieler in Team 1 und feindliche NPCs in Team 2 sind.

Team -127 – -1 Die negativen Teams sind für neutrale NPCs vorgesehen. Markiert ein Spieler ein NPC aus einem negativen Team so wird dieser auf dem HUD blau hervorgehoben. NPCs in einem negativen Team greifen nur GameObjects anderer negativer Teams an. D.h. ein NPC in Team -1 würde einen anderen NPC in Team -2 als Fein erkennen und angreifen. Spieler in Team 0 oder 1 würde er jedoch ignorieren.

Spieler können während des Spiels ihr Team beliebig wechseln. Aus Zeitgründen ist das gegenwärtig jedoch leider nur über die Konsolen-Funktion `team()` möglich. Durch die-

ses Handhabung der Teams ist sehr flexibles Teamplay möglich. So können sich z.B. in einem Deathmatch-Spiel einige Spieler direkt verbünden indem sie von Team 0 in ein gemeinsames positives Team wechseln (z.B. Team 1). Bei kooperativen Spielen ermöglicht es Spielern z.B. spontan in das Team der NPCs zu wechseln und diese zu verteidigen statt sie anzugreifen.

3.11.3 NPCs

Da das Spiel für kooperatives Gameplay ausgelegt wurde sind NPCs (non playable characters) ein notwendiges Element. Sie ermöglichen ein *Player vs. Environment* Gameplay indem alle Spieler vereint gegen Raumschiffe spielen, die vom Computer gesteuert werden.

Als wichtigster NPC fungiert die Fregatte. Fregatten sind sowohl Orientierungspunkte für den Spieler als auch wichtige Elemente im Kooperationsmodus, da es dort die Aufgabe der Spieler ist, gegnerische Fregatten zu zerstören. Des weiteren waren computergesteuerte Jäger sowie Munitionstransporter geplant. Aus Zeitgründen mussten wir uns allerdings auf die Fregatte beschränken und konnten keine Bewegungs-KI für NPCs implementieren.

Eine Fregatte an sich ist unbeweglich und kann sich nicht direkt verteidigen. Jedoch verfügt eine Fregatte über zwei schwere Geschütze sowie zwei Flak-Geschütze. Diese Geschütze sind als eigene GameObjects realisiert und können relativ zur Fregatte positioniert werden. Dadurch ist es möglich die Fregatte einfach mit weiteren Geschützen zu erweitern. Für größere Geschütze erfordert das allerdings entsprechende Halterungen im Fregatten-Model um glaubhaft zu wirken. Kleinere Geschütze wie MGs zur Punktverteidigung könnten allerdings ohne Modifikation des Models hinzugefügt werden. Das jedes Geschütz ein eigenes GameObject ist bietet einen weiteren wichtigen Vorteil: Spieler können die Geschütze einzeln zerstören. So ist es Spielern möglich ein Flak-Geschütz zu zerstören und damit die Fregatte auf einer Seite gegen weitere Angriffe verwundbarer zu machen.

Damit die Geschütze auch sinnvoll das Feuer eröffnen musst eine Ziel-KI implementiert werden. Dabei sind Parameter wie z.B. maximaler Anstellwinkel der Läufe, horizontale und vertikale Rotationsgeschwindigkeiten, usw. anpassbar. In die Zielberechnung fliesen auch die Geschwindigkeit des Ziels ein, so dass ein Ziel auch bei hoher Distanz relativ genau getroffen werden kann. Der Schwierigkeitsgrad eines NPCs lässt sich anschließend durch eine Streuung festlegen. Bei einer hohen Streuung wird das Ziel oft verfehlt und ein angreifender Spieler hat leicht Erfolg. Bei geringer oder keiner Streuung stehen die Chancen eines angreifenden Spielers jedoch sehr schlecht, der Schwierigkeitsgrad wird als hoch empfunden und anderen Strategien müssen erarbeitet werden.

Die gleiche KI wird von allen Geschützen verwendet. Jedoch haben die Geschütz-Typen unterschiedliche bevorzugte Ziele. So feuert eine Flak normalerweise auf Jäger währen ein schweres Geschütz normalerweise auf andere Fregatten feuert. Um diese Verhalten zu erhalten mussten alle relevanten GameObjects mit einem *Thread-Level* markiert werden. Jedes Geschütz verfügt über ein bevorzugtes Thread-Level und greift bevorzugt Einheiten mit diesem Level an. So bevorzugen Flak-Geschütze Jäger als Ziele, während die schweren Geschütze der Fregatte bevorzugt andere Fregatten beschießen. Erst wenn kein Ziel des

bevorzugten Thread-Levels in Reichweite ist werden andere Ziele angegriffen. So feuern auch schwere Geschütze auf den Spieler falls kein anderes Ziel verfügbar ist.

Jedes Geschütz (auch die Waffe des Spielers selbst) weist eine Streuung auf. Diese Streuung wird für jeden Schuss innerhalb konfigurierter Begrenzungen zufällig berechnet. Durch einen zentralen Server entfällt die Übertragung und Abstimmung von Gameplay relevanten Zufallszahlen zwischen den Clients da diese Berechnung ausschließlich zentral auf dem Server erfolgt.

Alle abgefeuerten Geschosse verfügen zudem über eine begrenzte Lebenszeit. Nach dieser Zeit verschwinden sie ohne Auswirkungen aus der Spielwelt. Diese Lebenszeit sowie andere Waffen-Parameter können über XML-Dateien festgelegt werden. Einzige Ausnahme ist das Flak-Geschütz, bei dem die Lebenszeit eines Geschosses dynamisch berechnet wird. Dadurch explodieren die Geschosse immer in der nähe des angepeilten Zieles, auch wenn es verfehlt wird.

3.11.4 Steuerung

Damit sich ein Spieler im Spiel schnell zurecht findet ist eine intuitive Steuerung notwendig. Daher wurde eine Steuerung implementiert die der normaler Spiele sehr ähnlich ist:

- Durch horizontale und vertikale Mausbewegungen kann der Spieler nach links oder rechts und nach oben oder unten schauen.
- Die Tasten `q` und `e` ermöglichen es den Jäger des Spieler links oder rechts herum zu rotieren.
- Mit der gängigen WASD-Steuerung kann man nach vor beschleunigen (`w`), bremsen (`s`) sowie nach links (`a`) und rechts (`d`) beschleunigen.
- Zudem kann man mit der `Leertaste` und `Strg` direkt nach oben oder unten beschleunigen.

Damit sich Spieler im 3D-Raum schneller zurecht finden wurde zunächst ein künstlicher Horizont implementiert. Dadurch verhält sich die Rotation nach links und rechts wie die normale Kopfbewegung eines Menschen: die Rotation erfolgt immer um eine Achse senkrecht nach oben vom Horizont und nicht um die Aufwärts-Achse der aktuellen Spieler-Sicht. Diese Steuerung funktioniert sehr gut falls sich das Spielgeschehen auf einer 2D-Ebene konzentriert. In einem freien 3D-Raum wiederum gerät die Sichtrichtung sehr schnell nah an die Aufwärts-Achse des Horizonts, wodurch man sich nur noch im Kreis dreht statt nach links und rechts. Dem könnte man entgegen wirken indem der Spieler den Horizont selbst verschieben kann. In einem relativ schnellen und reflexlastigen Spiel ist das jedoch eine sehr große Ablenkung und Behinderung. Dennoch wäre diese Art der Steuerung für andere Arten von Spielen geeignet (z.B. 3D-Strategiespielen).

Um keine Totpunkte in der Steuerung zu erhalten wurde statt eines künstlichen Horizonts eine komplett freie 3D-Steuerung implementiert. Dadurch kann sich der Spieler ohne Ein-

schränkung in jede Richtung drehen. Diese Steuerung irritiert neue Spieler allerdings da durch die horizontale und vertikale Rotation die Welt schnell in Schieflage gerät. Das ist eine für Menschen ungewohnte Wirkung und Perspektive und erfordert zur Korrektur eine manuelle Rotation um die eigene Achse.

Zusätzlich kommen in der Steuerung einige Dämpfungsmechanismen zum Einsatz. Beschleunigt der Spieler z.B. so wird die Bewegung, die von der Beschleunigungsrichtung abweicht, permanent gedämpft. Dadurch können Spieler schneller ihre Flugrichtung ändern.

3.11.5 Prozedurale Levels

Levels werden im Spiel durch spezielle LUA-Skripte realisiert. Jedes Level enthält ein LUA-Script für den Server und eines für den Client.

Das Server-Script legt über spezielle LUA-Funktionen auf dem Server die nötigen GameObjects für das Level an und nimmt Einstellungen vor. So werden z.B. an speziellen Positionen Fregatten erstellen und über einfache mathematische Operationen Asteroidengürtel generiert. Da LUA eine voll funktionsfähige Programmiersprache ist fällt die algorithmische Generierung von Inhalten sehr leicht. Zudem kann festgelegt werden ob sich erstellte GameObjects überlappen dürfen oder nicht. So kann schnell eine glaubwürdig wirkende Kulisse aufgebaut werden.

Das Server-Script regelt ebenfalls welche GameObjects welchen Teams zugewiesen sind und in welches Team neue Spieler automatisch eingeteilt werden. Dadurch lassen sich sowohl Deathmatch als auch kooperative Spielmodi realisieren.

Das Client-Script eines Levels lädt Elemente die nur auf dem Client relevant sind. Dazu wird ebenfalls über spezielle LUA-Funktionen die Skybox mit entsprechenden Texturen geladen, neue Belichtungseinstellungen vorgenommen und entsprechende Hintergrundmusik gestartet.

Die Scripte eines Levels werden erste geladen und ausgeführt, wenn das entsprechende Level gestartet wird. Dadurch ist es möglich ein Level-Script zu bearbeiten während das Spiel läuft. Nach der Veränderung lädt man über einen Konsolen-Befehl das Level im Spiel neu und sieht augenblicklich die Auswirkungen der neuen Scripte. Dadurch ist es einfach ein Level auf diese Art zu erstellen.

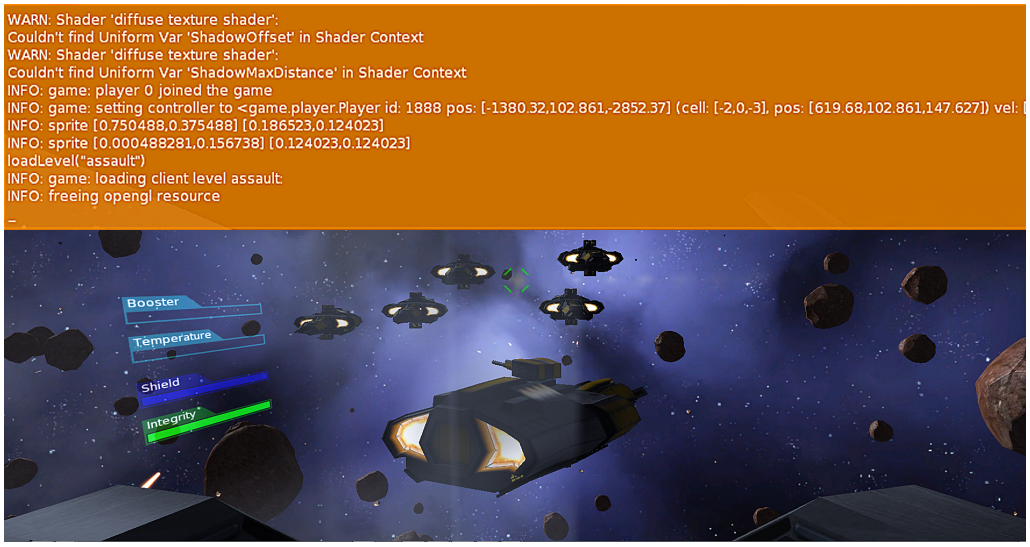


Abbildung 2: Ingameconsole und Levelchange

3.12 Content

3.12.1 Grafiken

Für alle Texturen kam das PNG (Portable Network Graphics) zum Einsatz. Dieses Dateiformat hat viele Eigenschaften, die wir bei der Entwicklung des Spiels gebrauchen konnten. So ist dieses Format zwar komprimiert, es geht aber kein Inhalt verloren. Es arbeitet Verlustfrei. Genauso wurde die Unterstützung von Alphakanälen für beispielsweiße einige Partikel in Spacecraft verwendet.

Die Farbtiefe von 16 Bits kommt vor allem den Normalmaps zu Gute. Animationen wurden über einen manuellen Sprite Atlas geregelt. Dieser wurde von Hand zusammengebaut und mit Werten in Form von XML Dateien an das Spiel weitergegeben.

In Anbetracht der Arbeitsweise von Grafikkarten und deren Limitierung, gab es von Anfang an mehrere Reglementierungen, um möglichst viel Performance zu erreichen.

Die aktuell maximal darstellbare Größe von Texturen auf einer Grafikkarte beträgt für den Mainstream (Grafikkarten ab 100 Euro und nicht über 5 Jahre alt) $4096 \cdot 4096$ Pixel. Dieses wollten wir ausreizen aber nicht überschreiten. Grafiken mit einer Auflösung in Zweierpotenz-Schritten können besser verarbeitet werden. Dies ist zwar nicht verpflichtend, doch wenn man dies vom Anfang an im Workflow mit berücksichtigt, lassen sich Schwierigkeiten von Anfang an vermeiden.

3.12.2 Speicherformat Collada

Als Schnittstelle für Models kommt Collada zum Einsatz. Ein offenes Format welches die Modeldaten sowie deren Animationen und Eigenschaften in einem XML–Dokument speichert. Weil es nicht auf ein spezielles Spiel oder Plattform zugeschnitten ist, werden besonders viele Informationen und Deklarationen darin spezifiziert. Da dieses Format nichts binär speichert, ist eine manuelle Korrektur und Bearbeitung mit jedem normalen Texteditor möglich. Jedoch mussten für das Spiel Anpassungen vorgenommen werden. Laut der Spezifikation des Datenformats (Common_Profil) sieht der Phongshader keine Normalmaps zur Darstellung vor. Da war es am einfachsten, die Spezifikation für unsere Zwecke anzupassen.

Dies hatte aber zur Folge, dass 3D Studio Max sowie jedes andere Tool diese Erweiterung nicht unterstützt hat, weder bei der Darstellung noch bei der Erstellung der .dae Daten. Deshalb musste bei jedem Exportieren die Datei von Hand nachbearbeitet werden.

3.12.3 SetBumpmap

setBumpmap ist ein kleines über C# und dem .net Framework geschriebenes Programm welches die Normalmap mit in die XML Struktur aufnimmt, ohne dabei viel Zeit zu verlieren. Da es relativ oft vonnöten war, entsprechend die Models zu testen, war es der nächst logische Schritt ein Tool zu entwickeln, welches diese mühselige Arbeit übernimmt. Dies geschieht über einen XML-DOM-Parser, welcher das Dokument liest, validiert und dann entsprechend die Daten verändert beziehungsweise hinzufügt. Dafür wurde ein kleines GUI gebastelt das sich per Button und drag&drop einfach bedienen lässt. So ist in drei kleinen Schritten die Bumpmap eingebaut:

1. Programm öffnen
2. Modelldatei auf den richtigen Button ziehen
3. beliebige Bilddatei auf den zweiten Button ziehen
4. Das Model wird automatisch mit den neuen Daten gespeichert.

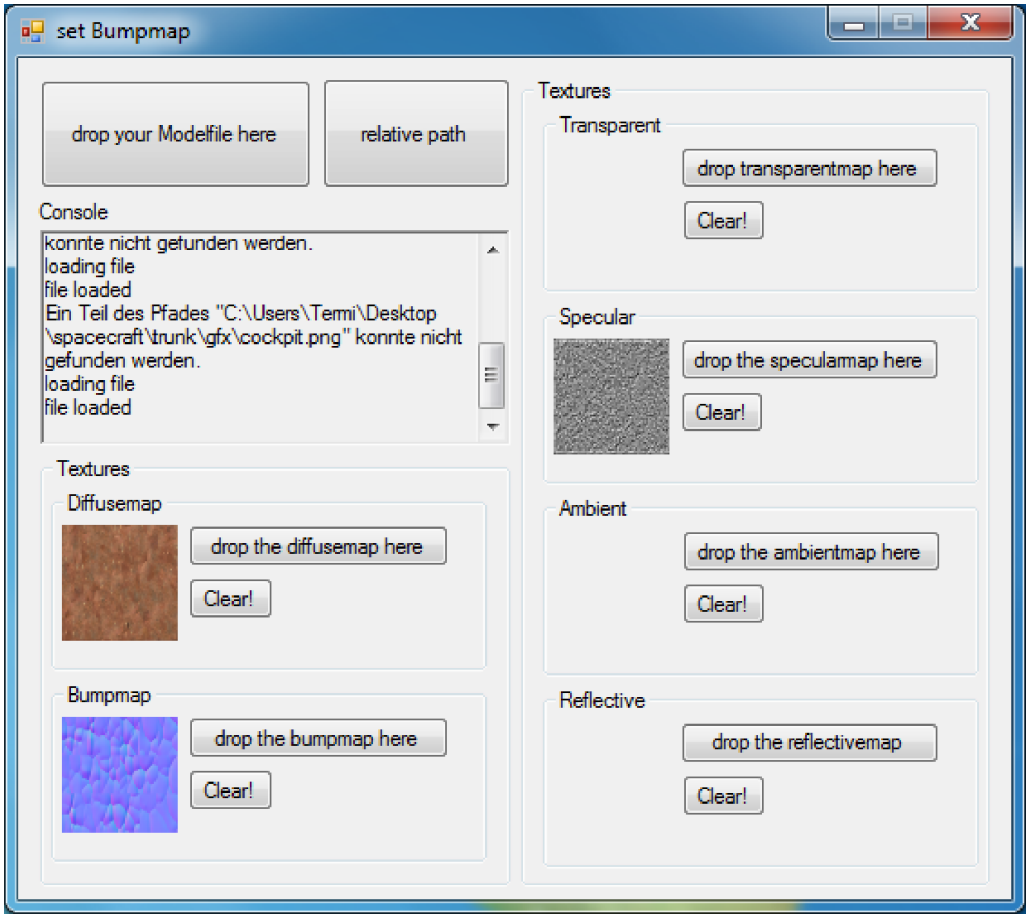


Abbildung 3: Applikation setBumpmap

Die zweite Ausbaustufe 3 bekam mehr Funktionen, ebenfalls um den Workflow zu optimieren. So konnte man ab nun jede Textur per Drag&Drop sofort austauschen und gleichzeitig anzeigen lassen. Weitere Funktionen wie zum Beispiel die Pfade relativ zum Programm umzuschreiben, haben bis zum jetzigen Zeitpunkt leider noch nicht ins Programm gefunden und werden noch vor Veröffentlichung ergänzt.

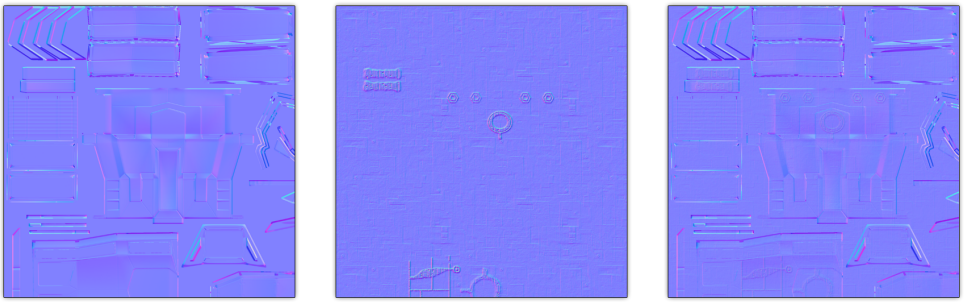
3.12.4 Normalmaps

Normalmaps manipulieren die Normalen einer Fläche wodurch mehr Geometrie dargestellt werden kann als wirklich vorhanden. Dafür sind Texturkoordinaten auf dem Objekt nötig.

Für die Erstellung von Normalmaps gibt es mehrere Möglichkeiten. Angewendet wurden hier zwei verschiedene, um in der Kombination das beste Ergebnis zu erzielen.

3.12.5 Generierung der Normalen aus einem High Polygon Model

Zusätzlich zum Spielmodel wird eine Version mit mehr Details erzeugt. Dort werden Details hinzugefügt die später als veränderte Normale in die Datei mit eingerechnet werden. Dafür muss dieses Model nicht erneut texturiert werden. Über einen Raytracer werden die Koordinaten vom Highpolymodel auf das Spielmodel übertragen.



Generierte Normalmap aus Highpoly Model | Generierte Normalmap aus Diffusemap | Kombinierte Normalmap

Abbildung 4: Normalmap Unterschiede

Die zweite Methode benutzt die Diffusetextur, um Strukturen aus der Textur zu gewinnen. Die Höheninformationen werden über Berechnungen durch unter anderem Graustufen der Textur nachempfunden und daraus eine Normalmap generiert. Diesen Prozess kann man über Einstellungen wie zum Beispiel der Stärke optimieren. Diese Methode lässt die Geometrie vollkommen außer Acht, weshalb eine Kombination ein besseres Ergebnis bringt. Probleme hierbei sind die Normalenberechnung an sich. Man kann nicht eine Textur auf die andere überlagern, da hierdurch die Normalen beeinflusst werden und nicht mehr die korrekte Ausrichtung haben. Deshalb musste hierfür die Software CrazyBump erworben werden, die diesen Schritt übernimmt.

Bei der Fregatte wurde noch ein dritter Schritt notwendig. So wurden die von Hand gezeichneten Platten, die über das Model verstreut wurden dazu verwendet, eine einfache Heightmap zu erstellen. Ein Grauton von 128 RGB markiert hierbei die neutrale Höhe. Hellere Farben bedeuten einen Höhenanstieg und dunklere Farbwerte eine Vertiefung. Diese Heightmap wurde, um Ressourcen zu sparen, ebenfalls als Normalmap umgerechnet und mit den anderen beiden kombiniert. Dies war möglich, da CrazyBump Heightmaps umrechnen und die Normalmaps zusammenführen kann.

3.12.6 Probleme bei der Normalmaperstellung

Zuerst wurde versucht, mit gegebenen Mitteln die Normalmaps zusammenzurechnen. (Photoshop und 3D Studio Max). Leider war das Resultat nicht zu gebrauchen. Es zeigten sich

an etlichen Stellen Artefakte durch versehentlich manipulierte Normale. Leider hat CrazyBump den Nachteil, dass Texturen über 2048px nicht mit einander verrechnet werden können. So musste die Textur, die in $4096 \cdot 4096$ Pixel vorlag jeweils in 2048 Pixel große Texturen unterteilt und einzeln kombiniert werden. Bei drei Normalmaps à vier Unterteilungen waren für jede Ausfertigung zwölf Arbeitsschritte notwendig.

3.12.7 Probleme mit Collada / 3D Studio Max

3D Studio Max macht viele Dinge nicht wirklich regelkonform. Normalen werden über eine Funktion in 3D Studio Max zwar richtig wiedergegeben, aber leider wird diese Funktion weder beim Exportieren angewendet noch übergeben (X Form).

Die Assimp Library kommt nicht mit den von 3D Studio Max generierten Animationen klar, weshalb diese einen Fehler verursachen.

3D Studio Max erstellt in unregelmäßigen Abständen defekte Modele welche zum Teil eine $1 \cdot 10^{27}$ große Positionen erstellt hatten. Diese Models waren ebenfalls nicht lesbar. 3D Studio Max konnte dieses Model aber wieder korrekt importieren.

3.12.8 Designprozess des Spiels

Von Anfang an war klar, dass wir einen etwas schmutzigeren, mit Augenmerk auf die Funktion gelegten Style annehmen wollten, angelehnt an die Konzeptzeichnungen von Homeworld 2 (welches bis zum jetzigen Zeitpunkt noch nicht veröffentlicht wurde und wohl auch nie wird).

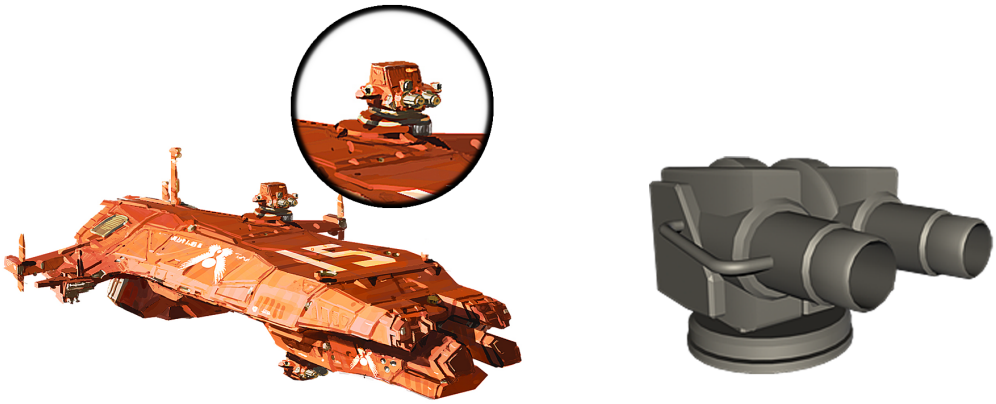


Abbildung 5: Schweres Geschuetz - Konzept

So wurde z.B. auch das erste schwere Geschütz einer Zeichnung von Homeworld nach-

empfunden. Das Geschütz wurde dahingehend verändert, dass es seine Geschosse auch im vertikalen Winkel anpassen konnte, welches das Konzept nicht vorsah. Da aber insgesamt weder dieses Design noch dessen Realisierung ansprechend war, wurde es – ebenso wie weitere Konzepte teils aus Zeitgründen, teils wegen anderer Varianten fallen gelassen.

Ausgehend davon das beide Fraktionen Menschen sind die sich erst vor kurzer Zeit gespaltet hatten ergaben sich dadurch einige Vorteile. Die Technologie ist auf beiden Seiten die gleiche. Bedingt durch die Kürze des Konflikts war es noch nicht möglich Wissenschaftliche Errungenschaften für sich zu nutzen. Zudem haben beide Teams den gleichen Technologiebaum sei es von der Vergangenheit bis zum jetzigen Zeitpunkt. Deshalb konnte die Geometrie stark an das jetzt bzw. nicht zu ferne Vergangenheit angelehnt werden.

Weitere Ideen waren z.B. das es keine Menschenrasse gibt sondern das die Fraktionen weiterentwickelte Lebensformen sind die via Gedankenkontrolle das Schiff steuern konnten. Herauskristallisiert hat sich hier die Kurzbeschreibung "Gehirne in Flaschen".

Der andere Ansatz war das dieser Krieg alleine von ferngesteuerten Drohnen ausgefochten wird. Was zur Konsequenz hat das Fenster sowie strategisch wichtige Punkte wie z.B. die Lebensversorgung keine Beachtung finde, wohl jedoch die Steuerung der Schiffe.

Alle Waffen verfügen über verschiede Projektile unterschiedlichen Kalibers, weshalb Form und Verhalten stark an historische Waffen angelehnt wurden. Der Aufbau jeder einzelnen Waffe, Heavy, Flack, MG, erinnert zumindest zum Teil an die Technologie aus dem zweiten Weltkrieg bzw. heutige Technik. So ist das schwere Geschütz stark an das Aussehen eines Geschützturmes eines Zerstörers der heutigen Zeit angelehnt. Als Vorlage wurden die Geschütztürme der "Bismarck" verwendet, auch wenn diese um einige Formen und Kanten erweitert wurden. Lediglich die Dimensionen sind bei allen Waffen um einiges größer. Erste Entwürfe der Flak beruhten auf einer Zwillingskanone. Die MG ist zum jetzigen Zeitpunkt eine Waffe, welche auf einer Drehhalterung montiert wurde.

Diese Konventionen ergaben aber auch Probleme. Dadurch dass der Mensch diese Gegenstände durchaus wiederzuerkennen weiß, hat er innerlich Proportionen dafür in der Vorstellung, die er dann versucht auf das Objekt anzuwenden. So fällt es beim jetzigen Aussehen der Fregatte schwer zu glauben, dass dieses Schiff über 500 Meter lang sein soll. Andere Indikatoren wie zum Beispiel die Größe der Schrift und die Größe der Fenster versuchen dieses Missverhältnis auszugleichen, was aber eher schlecht als recht funktioniert. Die nötige Konsequenz wäre, die Größe der Geschütze zu verringern damit die Proportionen mehr Sinn ergeben.

Eine weitere Designidee waren die Panzerplatten, welche die Technik schützen soll. An allen relevanten Teilen sind meterdicke Metallplatten angebacht. Die Fregatte soll von vorne rleativ wenig Angriffsfläche bieten und wurde so designed, dass aufreffende Projektile welche direkt von vorne kommen, über die Platten abgelenkt werden.

3.12.9 Designhintergrund Spacestation

Bei der Spacestation wurde versucht, ein modulares System aufzubauen. Mit gerade einmal zwei Objekten (ein Verbindungsstück sowie einer Kapsel) lassen sich nahezu unendlich viele verschiedene Geometrien bilden. Somit wäre auch hier eine prozedurale Form via luascript vorstellbar. Leider macht hierbei aber die Kollsion Probleme. Zwar wurden beide Stücke so designed, dass jede Kollision über einen Quader abgedeckt werden kann. Es reicht hierfür sogar eine normale Boundingbox, jedoch wäre es bei größeren Modellen ein großer Aufwand, jede Box einzeln auf Durchdringung testen zu lassen. Deshalb sind in der aktuellen Form nur zwei, schon fertig zusammengebaute Stationen verfügbar. Hierbei wurde aber die Kollisionen so angepasst, dass selbst bei einer Raumstation mit über 60 Einzelelementen lediglich rund 100 Einzelprüfungen durchzuführen sind.

Abbildung 6: Kollisionsmodell Spacestation

Die farbigen Boxen in Abbildung 6 zeigen das Kollisionsmodell. Es wurde überlappend gearbeitet, um möglichst wenige Rechenschritte für die Kollisionsberechnung durchzuführen.

Dieses Bild ist etwas trügerisch, da der Scanline Renderer die Verdeckungen geschickt rausrechnet.

3.12.10 Workflow

In Figur 7 wurden stark vereinfacht die Arbeitsschritte dargestellt, die bei nahezu jedem Model eingehalten wurde. So war die Konzeptionsphase ein relativ großer Teil des Projektes.

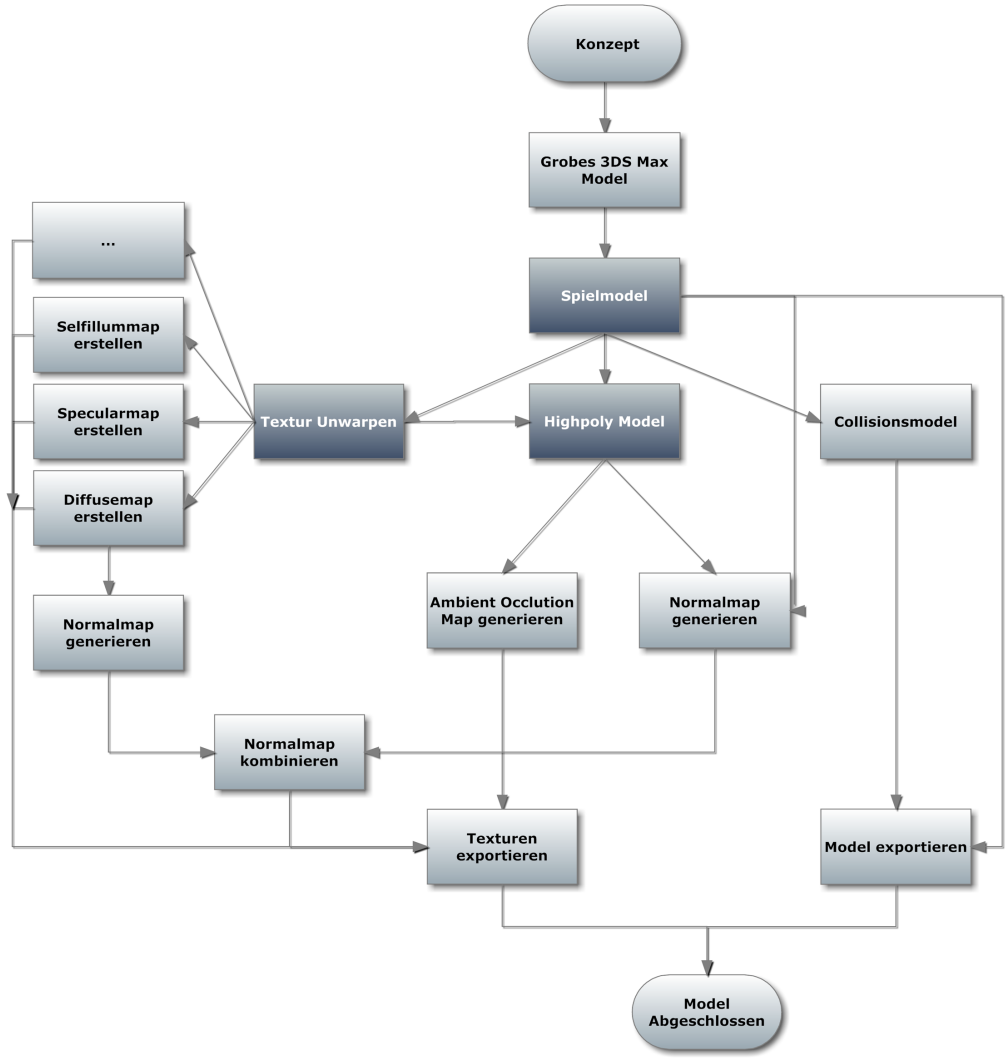


Abbildung 7: Workflow

Der Aufwand war je nach Model unterschiedlich. So waren die drei verschiedenen Asteroiden relativ einfach über einen Modifier Stack in 3D Studio Max zu lösen. Ein einfacher Quader konnte in diese Form gebracht werden. Das Kollisionsmodel und die Highpoly

Version waren verschiedene Detailgrade, die man beim Quader einstellen konnte. Im Gegensatz dazu hat die Fregatte und deren Geschütze die meiste Projektzeit verschlungen.

Die im Laufe des Projekts angefallenen Grafiken/Sprites wurden entweder aus schon vorhandenem Material genommen oder in Photoshop neu gezeichnet. Hierbei war der Workflow ein gänzlich anderer. Es gibt viele Beispiele aus anderen Spielen, wie ein Sprite aussehen muss, um eine entsprechende Wirkung zu erzielen. Meist stand aber ein Foto am Anfang der Arbeit. Die daraus entnommenen Farbwerte gaben einen guten Anhaltspunkt für die Entwicklung. Über die verschiedenen Photoshop Pinsel-Einstellungen wie:

- Struktur
- Streuung
- Transfer
- Rauschen
- Kantenglättung

war es möglich, eine relativ natürliche Struktur der Pinsel wiederzugeben.

3.12.11 Skybox

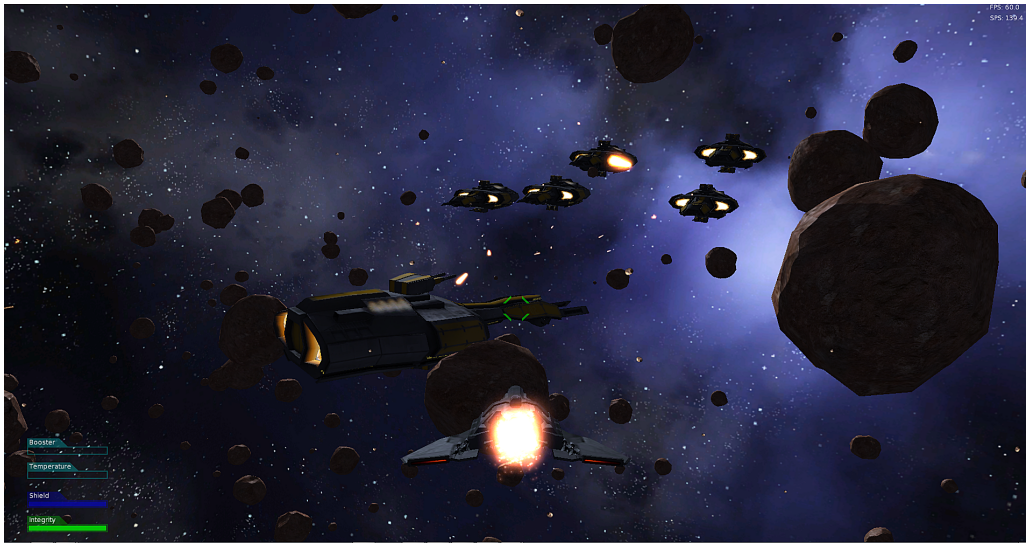


Abbildung 8: Im Spiel, Player vs. Environment

Wie in den meisten Spielen setzt sich der Hintergrund aus einer Cubemap zusammen, welche sich nicht mit der Spielerposition mit bewegt. Sechs Texturen, welche für jede Seite des Würfels stehen, ergeben eine Cubemap. Das geübte Auge kann die unterschiedlichen Pixeldichten erkennen. Dies kommt daher, dass zum Rand hin mehr Pixel zur Verfügung

stehen und sich dadurch Abgrenzungen abzeichnen.

Bisher wurden drei Hintergründe in das Spiel integriert. Der erste, die Sternenkarte, ist komplett frei von Nebel oder Anomalien. Die zweite Karte hat mehr Abwechslung durch zwei verschiedene Nebel, siehe Abbildung 8. Der dritte Hintergrund ist komplett in Orange gehüllt, siehe Abbildung 9.

Entstanden sind die Bilder in 3D Studio Max über einen prozeduralen Aufbau, welcher über “Mix Layer“ die verschiedenen Sternkonstellation darstellt. Darunter fallen die Sternen-dichte, Konzentration, Helligkeit und einen Standard “Nebel“ welcher größere Sternhaufen darstellen soll.

Die Nebel sind durch Volumenlichter entstanden. Über Fraktale sowie andere Algorithmen wurden sich überlagernde Lichter genommen, die solch ein Resultat wie in den beiden Abbildungen dargestellt hervorrufen. Wichtig hierbei war, dass die Cubemap nicht zu hell ist, da ansonsten das HUD überblendet und der Spieler dadurch Probleme mit der Steuerung bekommt. Die Karten wurden mit Hilfe einer Refract Cubemap in 3D Studio Max in die sechs Richtungen im Raum unterteilt und umbenannt exportiert.

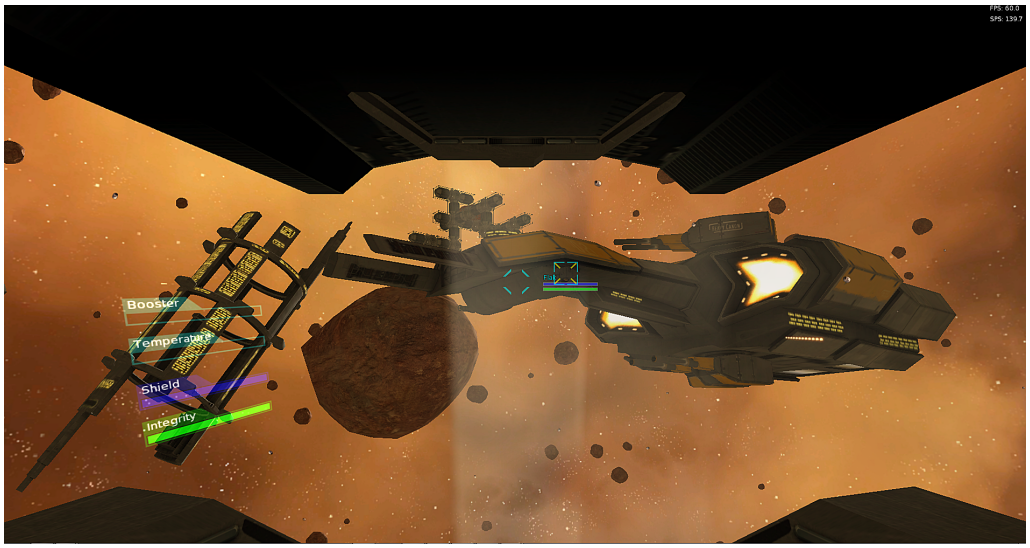


Abbildung 9: Im Spiel, Player vs. Player

4 Aussichten

Geplant ist, dass dieses Projekt bis zu seinem logischen Ende fortgeführt wird. Das Team strebt eine online-Veröffentlichung an, so dass auch Andere die Möglichkeit haben werden, dieses Spiel zu spielen. Allerdings sind bis dahin noch einige Umbaumaßnahmen und Ergänzungen geplant.