AUGUSTUS

HTBLuVA Innsbruck, Anichstraße

Höhere Abteilung für Wirtschaftsingenieurwesen

Ausbildungsschwerpunkt Betriebsinformatik

DIPLOMARBEIT

AUGUSTUS
Wirtschaftssimulation mittels Strategiespiel
Künstliche Intelligenz

Ausgeführt im Schuljahr 2006/07 von:

Severin Franz Gassler	5AHWII-04
Georg Haaser	5AHWII-06
Harald Fritz Steinlechner	5AHWII-21
Manuel Manfred Wieser	5AHWII-28

Betreuer:

Ing. Mag. Dr. Armin Mauracher

Innsbruck, am 25.05.2007

EIDESSTATTLICHE ERKLÄRUNG

Ich erkläre an Eides statt, dass ich die vorliegende Diplomarbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche erkenntlich gemacht habe.

Wattens, am 14.05.2007

Die Verfasser:

Severin Franz Gassler

Georg Haaser

Harald Fritz Steinlechner

Manuel Manfred Wieser

Wir widmen diese Arbeit dem Heiligen Gral, aus dem wir ständig neue Energie schöpften - bis er leer war.

Vorwort

Als wir im Jahr 2006 vor der Auswahl unseres endgültigen Diplomarbeitsthemas standen, hatten wir uns schnell dafür entschieden ein Strategiespiel zu programmieren. Diese Entscheidung beruhte vor allem darauf, dass wir durch unser persönliches Interesse und Engagement zu diesem Zeitpunkt bereits einige Erfahrungen mit 3D-Programmierung und Spielmechanik gemacht, und auch in mehreren Kleinprojekten in unserer Freizeit bereits unter Beweis gestellt hatten. So besaßen wir zum Startzeitpunkt der offiziellen Diplomarbeit bereits eine relativ weit fortgeschrittene und viel versprechende 3D-Engine. Die weitere Arbeit an der Diplomarbeit konzentrierten wir dementsprechend auf die Entwicklung der Fuzzy-Logic-Systeme, die spezielle Spielmechanik und die grafischen Inhalte eines Strategiespiels

An dieser Stelle wollen wir uns selbstverständlich bei allen bedanken, die an der Entstehung unserer Diplomarbeit mitgewirkt, und uns Denkanstöße gegeben haben. Voran unserem Betreuungslehrer, Dr. Armin Mauracher, der uns jederzeit und mit vollstem Engagement zur Verfügung stand. Natürlich hätten die uns in der Schule vermittelten Kenntnisse bei weitem nicht ausgereicht um diese Diplomarbeit umzusetzen, weshalb wir uns auch bei allen Online-Communities und Buchautoren bedanken, mit deren Hilfe wir uns die erforderlichen Kenntnisse erarbeitet haben.

Besonderer Dank gilt außerdem all unseren Freunden und Familienmitgliedern, die uns das Leben nicht vergessen ließen und uns vor einer Realitätsentfremdung bewahrten, während wir über Tausend Stunden in diese Arbeit investierten.

Kurzfassung

Das Hauptziel unseres Projektes war die Entwicklung einer Simulation eines antiken Wirtschaftssystems und die Darstellung seiner Abläufe und Beziehungen. Wir entschieden uns für ein antikes System, da ein modernes viel zu komplex gewesen wäre. Um dieses Ziel zu erreichen entschieden wir uns dafür, ein Strategiespiel als Grundlage zu benutzen. In einem Strategiespiel können ökonomische Strukturen und Vorgänge dem Benutzer in einer spielerischen Weise vermittelt werden. Diese spielerische Weise wirtschaftliche Zusammenhänge darzustellen, gibt unserem Spiel einen pädagogischen Charakter. Der/Die BenutzerIn muss lernen das ganze System zu steuern und beherrschen, um seine/ihre Ziele zu erreichen.

Eine der schwierigsten Aufgaben war es, eigenständig handelnde Personen im System zu simulieren. Die Einwohner der Stadt müssen sich z.B. selbst eine Stelle suchen um Geld zu verdienen. Wir mussten simplifizierte Subsysteme implementieren, welche in einer menschlichen Weise agieren und entscheiden. Um diese Komponente zu simulieren bedienten wir uns einer sehr oft verwendeten Methode der Künstlichen Intelligenz (KI): Fuzzy Logic. Fuzzy Logic bietet die Möglichkeit Systeme zu entwerfen, die mit einer Art menschlicher Logik entscheiden. Wir benutzten die Fuzzy Logic hauptsächlich für Abläufe bezüglich der Bürger einer Stadt. Diese Abläufe sind unter anderem das Suchen einer Anstellung, zur Arbeit gehen und das Arbeiten selbst.

Einer der Hauptteile des Projektes war die Realisierung eines computergesteuerten Gegners, der im Grunde alle Aufgaben eines Spielers übernehmen und somit auch gegen den Spieler spielen kann. Der Gegner handelt auf Basis eines eigens dafür entwickelten Systems, welches vorgefertigte Lösungen für alle möglichen Situation bietet, in die der Gegner geraten kann. Wir haben die Verwendung eines neuronalen Netzes für den Computergegner in Betracht gezogen, jedoch konnte dieses in unseren Tests keine ausreichende Performance bieten.

Alle Inhalte und Systemkomponenten, wie das Darstellen von 3D-Modellen, das Navigieren durch die Spielwelt, Kollisionserkennung, Modellierung oder Texturierung, wurden selbst erstellt und entwickelt. Da diese die größte Arbeit am Projekt waren, sollten sie besonders betont werden.

Abschließend soll noch erwähnt werden, dass unser Projekt mittels C++ entwickelt wurde und alle Anforderungen an ein modernes Projekt erfüllt.

Abstract

Main target of our project was to develop a simulation of an ancient economical system and visualize its procedures and dependecies. We decided to simulate an ancient system because a modern system would be far too complex. In order to reach this target we decided to use a real-time strategy game as platform. In a strategy game economical structures and procedures can be shown to a user in some kind of playful way. This playful way of pointing out economical relations gives our game an educational character. The user needs to learn to control the whole system in a satisfying way in order to reach his targets.

One of the most difficult parts was to simulate persons acting in the system. The citizens of the player‘s city have to find a job in order to earn money and many other tasks like that. We had to implement simplified subsystems to decide and act in a human way. For simulating this component we used one of the very common methods of today‘s Artificial Intelligence (AI): Fuzzy Logic. Fuzzy Logic offers the possibility to implement systems deciding through some kind of human logic. We used Fuzzy Logic mainly for realizing procedures concerning citizens and their tasks. Those procedures include finding a job, going to work, working and some other functions like that.

One of the main parts of the project was to realize a computer controlled enemy, which mainly fulfills all the player‘s tasks and plays against the user. This enemy acts on base of a self developed system, which contains predefined Solutions for all possible Situations the computer enemy can be in. We considered about using neuronal networks for realizing the computer enemy but in our tests those networks could not reach the required performance.

All the basic system parts like displaying 3D-Models, navigating through the game-world, collision detection, modelling, texturing and others are self developed. Implementing these parts was our main programming work and should be mentioned.

To sum up there is to mention, that our whole project was developed in C++ and fits the requirements of a modern project.

Impressum

Betreuungslehrer:

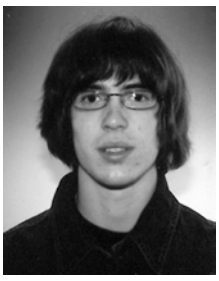
Ing. Mag. Dr. Armin Mauracher

Projektmitglieder:

Severin Franz Gassler
Adolf-Pichler-Weg 12
A-6065 Thaur
gassler@awx.at

Georg Haaser
Innstraße 1
A-6112 Wattens
haaser@awx.at

Harald Fritz Steinlechner
Unterberg 17
A-6111 Volders
steinlechner@awx.at



Manuel Manfred Wieser
Johannesfeldstraße 10
A-6111 Volders
wieser@awx.at

Inhaltsverzeichnis

Spielprinzip

1. Echtzeit-Strategiespiel

„Echtzeit-Strategiespiele (engl. real-time strategy, abgekürtz RTS) stellen ein Computerspiel-Genre dar, bei dem alle Spieler oder Computergegner ihre Handlungen gleichzeitig ausführen und in dessen Vordergrund das wirtschaftliche, strategische und taktische Agieren gegen einen Gegner steht.

[…]

Die Zeitgleichheit der Echtzeit-Strategiespiele setzt den Spieler unter größeren Zeitdruck und verlangt neben strategischer Weitsicht auch schnelle Reaktion und Übersicht in chaotischen Situationen, Karten oder Kämpfen. Zudem wird der Aufbau der Wirtschaft sowie der Steuerung der Einheiten schematisiert und stark vereinfacht.“ [1]

Augustus ist ein Echtzeit-Strategiespiel im Stile von namhaften Genre-Vertretern wie Age Of Empires (Ensemble Studios) oder Warcraft (Blizzard Entertainment). Trotz des winzigen Entwicklerteams von vier Personen, und darausfolgendem geringerem Umfang, werden etliche etablierte Ideen, Prinzipien und Gemeinsamkeiten existierender Strategiespiele in diesem ambitionierten Amateurprojekt vereint.

---

1 Wikipedia - die freie Enzyklopädie: Echtzeit-Strategiespiel. Online im Internet: URL: http://de.wikipedia.org/w/index.php?title=Echtzeit-Strategiespiel&oldid=27395272, zugegriffen am 12. Februar 2007

2. Steuerung und Kartenansicht

Nach Beginn des Spiels findet sich der Spieler auf einer dreidimensionalen, rechteckigen Karte wieder. Aufgrund der Übersicht wird das Geschehen aus der Vogelperspektive betrachtet. Die Karte kann vom Spieler bewegt, gedreht, vergrößert und verkleinert werden, um immer die beste Übersicht zu gewährleisten.

Bereits zu diesem Zeitpunkt wird ein typisches Prinzip moderner Strategiespiele ersichtlich. Jede Einheit und jedes Gebäude des Spielers hat einen gewissen Sichtradius. Gebiete außerhalb des Sichtradius sind in den Kriegsnebel (auch Fog of War) gehüllt. Der Spieler kann im Kriegsnebel zwar landschaftliche Details wie Bäume und Gewässer ausmachen, bekommt aber nichts vom Geschehen und feindlichen Truppenbewegungen in diesen Gebieten mit. [2]

Abb.1: Veranschaulichung von Sichtradius und Kriegsnebel in Augustus

Die Steuerung des Spiels besteht größtenteils aus der Verwendung der grafischen Benutzeroberfläche und dem Auswählen von Gebäuden und Einheiten. Wird ein Objekt ausge-

---

2 vgl. Wikipedia - die freie Enzyklopädie: Echtzeit-Strategiespiel. Online im Internet: URL: http://de.wikipedia.org/w/index.php?title=Echtzeit-Strategiespiel&oldid=27395272, zugegriffen am 12. Februar 2007

wählt passt sich die Benutzeroberfläche an und der Spieler sieht auf einen Blick alle Aktionen die mit dem Objekt in Verbindung stehen.

Objekte können zwar einzeln ausgewählt werden, allerdings gestaltet sich dies bei größeren Einheitengruppen als sehr zeitaufwändig. Deshalb können auch mehrere Einheiten gleichzeitig mithilfe eines Auswahlrahmens ausgewählt werden.

3. Aufbau- und Wirtschaftselemente

Den meisten Echtzeitstrategiespielen liegen Elemente von Aufbauspielen und Wirtschaftssimulationen zugrunde. Auch Augustus verwendet einige dieser Elemente. Im Folgenden wird dies durch ein Beispiel eines Spielverlaufs erläutert.

3.1. Ein typischer Spielverlauf

Am Beginn des Spiels erhält der Spieler zumindest ein Stadtzentrum und fünf Sklaven. Das Stadtzentrum ermöglicht ihm später, auf dem Sklavenmarkt weitere Sklaven zu erstehen.

Den Sklaven können nun Befehle erteilt werden, wie Bäume fällen, auf dem Steinbruch zu arbeiten oder ein weiteres Haus zu errichten. Üblicherweise lässt man zu Beginn einige seiner Sklaven Rohstoffe sammeln und baut mit einer Einheit ein weiteres Gebäude, zum Beispiel ein Wohnhaus.

Dieses Wohnhaus bietet Platz für neue Stadtbewohner, welche dort hoffentlich bald einziehen werden. Doch damit Neubürger sich in der noch jungen Stadt niederlassen muss diese attraktiv wirken. Ein Bürger erwartet ausreichende Rohstoff- und Nahrungsversorgung, hohe Sicherheit innerhalb der Stadt, die Chance einen guten Beruf ausüben zu können und keine übertriebenen Steuern.

Nun hat der Spieler die Möglichkeit weitere Gebäude wie eine Schmiede oder eine Mühle zu bauen. So schafft er Arbeitsplätze und bringt Bürger dazu, in seine Stadt zu kommen. Das Spiel schlägt einen Lohn vor, welcher vom Spieler nach oben oder unten korrigiert werden kann.

Da jetzt mehr Bürger in die Stadt kommen wächst natürlich der Rohstoffbedarf. In einem Sägewerk kann nicht nur Holz gelagert (Sklaven müssen nicht mehr ausschließlich zum Stadtzentrum laufen, sondern können ihr gesammeltes Holz nun auch im Sägewerk ablegen), sondern auch Möglichkeiten zur schnelleren Holzbeschaffung und zur Aufforstung erforscht werden. In der Mühle können Methoden für bessere Feldbewirtschaftung und in einer Schmiede neue Schmiedetechniken für ausgereiftere Waffen erforscht werden.

Allmählich wächst so der Warenbestand und die Stadt verfügt über reichlich Waffen, Nahrung und andere Güter. Mit der Errichtung eines Marktplatzes können Händler angelockt werden, denen überschüssige Warenbestände verkauft werden. Der Marktplatz bietet die Möglichkeit festzulegen, um wie viel Gold Waren verkauft werden und um wie viel man bereit ist beim Kauf einer anderen Ware zu bezahlen. Außerdem kann man bestimmen, wie viel des Lagerbestandes zum Verkauf steht, damit man nicht selbst in einen Engpass kommt.

Während die Stadt wächst entstehen Bürgerschichten, so etwa die Landwirte, Handwerker und Soldaten. Jede Gruppe muss der Stadt Steuern zahlen, welche der Spieler bestimmen kann. Niedrige Steuern führen zu erhöhter Zuwanderung in einer Gruppe, wohingegen hohe Steuern der Stadt dienen um eine bessere Infrastruktur zu bauen oder Handel zu betreiben.

Eine immer wachsende Stadt benötigt natürlich eine gute Verteidigung. Zwar besitzen sie im Kriegsfalle eine Bürgermiliz, jedoch kann diese kein echtes Berufsheer ersetzen. In der Kaserne hat der Spieler die Möglichkeit Soldaten zu rekrutieren und festzulegen wie viele davon Bogenschütze, Reiter oder Fußsoldat werden und wie viel jede Gruppe an Sold erwarten kann. Selbstverständlich benötigt man für das Heer genügend Waffen aus der Schmiede, wie Bogen, Schwerter, Schilde und Lanzen.

3.2. Ressourcen und Handelsgüter

Echtzeit-Strategiespiele arbeiten meistens mit einer geringen Auswahl an Rohstoffen die als Grundlage für den Bau aller Gebäude und Einheiten sowie der Entwicklung sämtlicher Technologien dienen. Rohstoffe können in verschiedenen Formen

abgebaut oder gewonnen werden. Die Rohstoffe sammeln sich auf einem imaginären Rohstoffkonto, sind in ihrer Anzahl begrenzt und in kleinen Vorkommen über die gesamte Spielkarte verteilt.

Im Gegenzug zu den Rohstoffen, sind die Handelsgüter nicht auf der Karte verteilt, sondern nur durch Verarbeitung von Rohstoffen oder durch Handel zu bekommen. Beispiele für Rohstoffe in Augustus sind Gold, Steine, Erz oder Nahrung. Handelsgüter sind zum Beispiel Schwerter und Schilde, da beides aus Erz hergestellt wird.

4. Kampfhandlungen

Verfügt der Spieler über ein Heer kann er mit diesem seine Stadt verteidigen aber natürlich auch seinerseits einen Feind angreifen. Einheiten können Angriffs- und Bewegungsbefehle erteilt werden. Truppen können sich in bestimmten Formationen fortbewegen und angreifen, was je nach Situation defensive und offensive Vorteile verschafft. Auch bewegen sich alle Einheiten in einer Formationen mit derselben Geschwindigkeit, so dass langsamere Einheiten nicht zurückfallen.

Der Spieler kann außerdem jeder Einheit eine Verhaltensweise anordnen. Mögliche Verhaltensweisen sind Aggressiv, Defensiv und Passiv. Aggressive Einheiten greifen jede feindliche Einheit in ihrem Aktionsradius sofort an. Der Aktionsradius entspricht in etwa dem halben Sichtradius der Einheit. Defensive Einheiten greifen erst an, sobald sie selbst vom Feind angegriffen werden und passive Einheiten versuchen sich vom Kampfgeschehen herauszuhalten und keinen Schaden zu nehmen.

Im Rahmen einer Kampfhandlung zwischen zwei Einheiten wird mithilfe zahlreicher Daten vom Spiel der Schaden an der jeweiligen Einheit berechnet. Faktoren zur Ermittlung des Schadens sind Treffsicherheit, Rüstung, Gesundheit, Angriffswert und Angriffsgeschwindigkeit einer Einheit. Auch ist der Typ einer Einheit ausschlaggebend. So hat ein berittener Soldat einen erheblichen Vorteil gegenüber einem Fußsoldaten, jedoch einen Nachteil gegenüber einem Bogenschützen.

5. Ziel des Spiel

Je nach Spielmodus (Gefecht oder Kampagne) sind die Ziele unterschiedlich. In einem offenen Spiel (Gefecht) gilt alleine der Sieg über den Feind, welcher sich mit Waffengewalt erkämpft werden muss. In der Kampagne ist das Hauptziel ebenfalls der Sieg über den Feind, jedoch müssen zuvor kleine Teilziele erreicht werden, wie zum Beispiel die Rohstoffzufuhr sicherstellen oder Gefangene befreien.

In beiden Fällen ist die Führung eines Heeres ohne funktionierende Wirtschaft undenkbar. Es ist also auch Ziel des Spielers eine wirtschaftlich erfolgreiche Stadt aufzubauen und klug zu handeln, auch wenn dies nicht ausdrücklich von ihm verlangt wird.

6. Technologiebaum

Ein Technologiebaum dient in Echtzeit-Strategiespielen zur Übersicht. Man erhält einen Überblick darüber, welches Objekt welche Aktionen ermöglicht und welche Objekte überhaupt existieren.

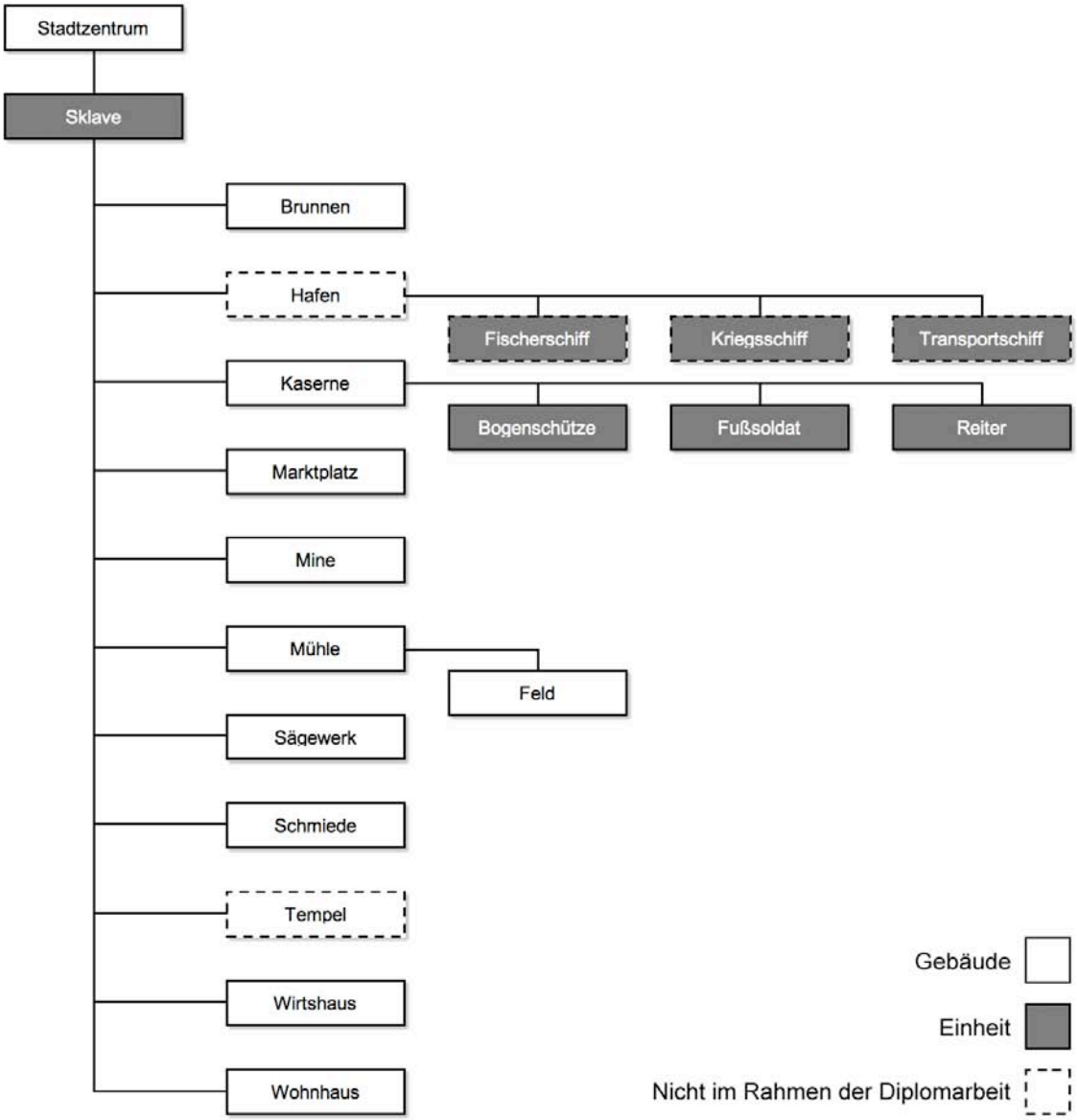


Abb.2: Einheiten/Gebäude-Technologiebaum für Augustus

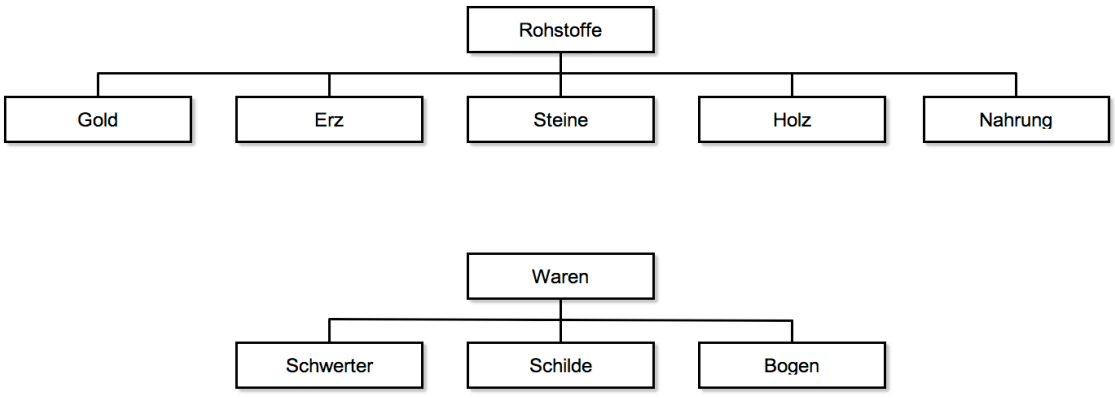


Abb.3: Rohstoffe und Handelsgüter in Augustus

Hintergrundgeschichte

1. Wahl des Szenarios

Bei der Wahl des Szenarios wurden zahlreiche, für das Projekt ausschlaggebende und bereits festgelegte Punkte berücksichtigt. So etwa der Bedarf an einem relativ simplen Wirtschaftssystem und der Beschluss des Verzichts auf moderne Waffen und Kriegsgeräte. Größtenteils war die Entscheidung über das Szenario jedoch dem persönlichen Geschmack überlassen.

Die Wahl fiel schlussendlich auf ein realistisches Szenario in der Antike. Dies ermöglichte uns zum Beispiel Seeschlachten im Mittelmeer, Wüsten- und Küstenschauplätze und historische Bauwerke wie die Sphinx, die Pyramiden und den Leuchtturm von Alexandria in unserem Spiel einzuarbeiten. Es legte auch zugleich Infanterie und Kavallerie als häufigste Einheitentypen fest. Ein weiterer Vorteil des antiken Szenarios sind die weniger komplexen Wirtschaftsverhältnisse zu dieser Zeit.

In den Anfängen der Planungsphase wurde auch entschieden, Augustus mit einer Kampagne und einer Hintergrundgeschichte zu versehen. Es musste also ein Handlungsstrang entworfen werden, der uns erlaubte, unsere Ideen zu verwirklichen.

2. Die Kampagne

Eine Kampagne bezeichnet in Echtzeitstrategiespielen eine Serie von Spielszenarien mit zeitlichem und geschichtlichem Zusammenhang. Kampagnen enthalten Ereignisse und Aufgaben die der Spieler in einem Gefecht oder im Mehrspielermodus üblicherweise nicht antreffen würde. In kommerziellen Echtzeitstrategiespielen wird die Hintergrundgeschichte des Spiels oft mittels vorgerenderter Videos, Sequenzen in Spielgrafik und professioneller Sprachausgabe weitererzählt. Der Spieler wird meist Teil der Geschichte, zum Beispiel als General oder Heeresführer, und interagiert mit anderen Charakteren, welche ihm Aufgaben erteilen. Die Kampagne endet mit Erfüllung aller Aufgaben und Szenarien durch den Spieler.

Für Augustus wurden Sprachausgabe und vorgerenderte Videos von vornherein ausgeschlossen, da dies nur mit den Mitteln eines großen Softwarestudios möglich wäre und nicht mit der zur Verfügung stehenden Zeit und Arbeitskraft der Diplomarbeitsgruppe. Die Story wird in Augustus durch Echtzeit-Kamerafahrten und Texteinblendungen am Beginn eines Spielszenarios erzählt.

2.1. Kurzbeschreibung der Handlung

> 31 vor Christus waren Gaius Octavianus, der spätere Kaiser Augustus, und Marcus Antonius die Herrscher des römischen Reiches. Octavianus bemächtigte sich in diesem Jahre illegalerweise des Testaments von Antonius. Dadurch kam dessen Beziehung zur ägyptischen Königin Cleopatra an die Öffentlichkeit. Antonius ließ Cleopatra römische Staatsgelder zukommen und wollte nach seinem Tode nach Ägypten überführt werden. Es kam zu einem Bürgerkrieg, welcher damit endete, dass Ägypten seine Unabhängigkeit verlor und zur römischen Provinz wurde. Antonius und Cleopatra starben und Octavianus wurde der erste römische Kaiser und erhielt den Namen Augustus. [3]

---

3 vgl. Politische und kulturelle Entwicklung Roms. Online im Internet: URL: http://www.gottwein.de/roge/his_0044.php, zugegriffen im Jänner 2006

2.2. Notizen zur Handlung

Der größte Teil der Handlung ist historisch belegt und wirklich so passiert. Sie wurde jedoch an manchen Stellen etwas modifiziert um sich besser in ein Computerspiel einzufügen. Gewisse Ereignisse sind nicht genau so geschehen wie im Spiel, sie wurden zugunsten des Spielerlebnisses abgeändert. Wer sich näher für die historischen Ereignisse interessiert, sollte sich in Fachliteratur über die Geschehnisse informieren.

2.3. Ausführliche Handlung und Szenariobeschreibung [4]

Abb.4: Zur Verdeutlichung der geographischen Position der Kriegsschauplätze wird dem Spieler in der Kampagne eine stilisierte Karte zur Verfügung gestellt. (Putzger-Bruckmüller: Historischer Weltatlas, 2. Auflage, Wien 2000, Seite14f)

---

4 vgl. Politische und kulturelle Entwicklung Roms. Online im Internet: URL: http://www.gottwein.de/roge/his_0044.php, zugegriffen im Jänner 2006

2.3.1. Erstes Kapitel

723 ad urbe condita

(31 vor Christus)

Gaius Octavianus, der spätere Kaiser Augustus, und Marcus Antonius waren die Herrscher des römischen Reiches.

Octavianus bemächtigte sich in diesem Jahre illegalerweise des Testaments von Antonius. Dadurch kam dessen Beziehung zur ägyptischen Königin Cleopatra an die Öffentlichkeit. Antonius ließ Cleopatra römische Staatsgelder zukommen und wollte nach seinem Tode nach Ägypten überführt werden. Dies entsprach nicht den Vorstellungen der antiken römischen Welt, weshalb es zur Reichsteilung und zur endgültigen Auflösung des Triumvirats kam.

Der Senat stellte sich auf die Seite Octavians und beschließt den Krieg gegen die ägyptische Königin Cleopatra. So konnte man wenigstens formell einem Bürgerkrieg entgehen.

Marcus Antonius unterstützte seine Königin mit allen Kräften. Im Osten Griechenlands versperrten seine Truppen den Gegnern den Weg. Doch Octavians fähigster Feldherr, Agrippa, führte seine Flotte durch das Ionische Meer, mit dem Ziel die Stadt Methone zu erobern und Antonius mit diesem Schachzug zu überraschen.

Gameplay:

Wir sehen eine Küste in einiger Entfernung von Methone, wo die römischen Schiffe ihre Truppen ablegen. Mit dem Helden Agrippa, einigen Fußsoldaten und mehreren Bauern muss nun eine kleine Basis errichtet werden.

Es gibt zahlreiche kleine Nebenziele, die zum Hauptziel führen. Hauptziel des Kapitels ist es die Stadt Methone zu erobern.

2.3.2. Zweites Kapitel

Die Eroberung Methones trug einige Opfer und Verletzte davon. Während Agrippas Truppen die Verwundeten verarzteten und auf die eigene Versorgung achteten, näherte sich vom Norden ein feindliches Heer. Es kam mit der Absicht die Eroberung Methones zu vergelten, und den Vormarsch von Agrippa zu verhindern.

Gameplay:

Der Spieler muss auf die Versorgung achten und das feindliche Heer zurückschlagen.

2.3.3. Drittes Kapitel

Octavian beschloss die römischen Staatsgeschäfte seinem etruskischen Freund Maecenas zu übergeben und machte sich selbst auf den Weg in Richtung Osten.

Aufgrund von Sabotage an seinen Schiffen steuerte er notgezwungen auf die Insel Korfu zu. Die Reparaturarbeiten an den Schiffen wurden von auf Antonius vereideten Bewohner behindert. Um den Angriffen Einhalt zu gebieten sollte die Stadt Corcyra eingenommen werden.

Gameplay:

In gewissen Zeitabständen kommen immer neue Saboteure und wollen die Schiffe zerstören.

Es kommt aber auch in bestimmten Abständen Nachschub (Schiffe mit Soldaten) aus Rom.

Aufgabe ist es ein kleines Camp zu errichten, den Angriff auf Corcyra vorzubereiten und die Stadt einzunehmen.

2.3.4. Viertes Kapitel

2. September, 31 vor Christus

Antonius entschloss sich auf Wunsch Cleopatras zu einer Seeschlacht.

Octavianus Flotte, unter Führung von Agrippa, traf im Golf von Ambracia auf die Flotten von Marcus Antonius und Cleopatra.

Gameplay:

Nördlich der Schneise zum Golf steht die Flotte des Spielers. Die Gegner befinden sich bei Actium. An den Küsten gibt es für die jeweilige Fraktion eine Basis.

Ist der Gegner fast besiegt (er besitzt nur noch ein Viertel seiner Männer) flüchten die ägyptischen Streitkräfte.

2.3.5. Fünftes Kapitel

Antonius folgte Cleopatra nach seiner Niederlage Richtung Ägypten. Sein Landheer ließ er zurück. Es wartete sieben Tage auf seine Rückkehr und lief dann zu Octavianus über. Dieser verfolgte Antonius mit einem kleinen Teil des Heeres über Asien.

724 ad urbe condita

(30 vor Christus)

Ein Späher berichtete Octavianus, dass sich Antonius in einem befestigten Dorf in den Bergen Syriens aufhalte. Da die Bewohner ihn nicht herausgeben wollten wurde der Beschluss gefasst, dieses Dorf zu stürmen.

Gameplay:

Gebirgiges Terrain, wenig Platz, hinterlistige Spione und ein gut verteidigtes Dorf erwarten den Spieler.

2.3.6. Sechstes Kapitel

Es kam die Nachricht zu Octavianus, dass sich Antonius zur Zeit der Attacke schon eine Zeit lang auf dem Weg nach Alexandria befand. Wütend ließ Octavianus Nachschubsschiffe von Rom schicken und zog mit seinem Heer weiter in Richtung Ägypten.

Schon weit vor der Stadt Sile wurde ihm der Weg in das Pharaonenland versperrt.

Gameplay

Große Mauern und Barrikaden, fähige und zahlreiche Gegner und wenig Rohstoffe fordern das Können des Spielers. Nachschubsschiffe aus Rom sichern in bestimmten Zeitabständen die Rohstoffzufuhr.

2.3.7. Siebtes Kapitel

Nahezu zwischenfallslos gelang das Heer Octavians weiter ins Landesinnere. Memphis sollte erobert werden, um den Nil zu stauen und weite Teile Ägyptens bis nach Alexandria vom Wasser abzuschneiden.

Bei den Vorbereitungen zum Angriff näherte sich jedoch vom Norden ein babylonisches Heer welches Octavianus einkesselte und der Stadt Memphis zur Hilfe eilte.

Gameplay:

Es gilt Bedrohungen von zwei Seiten zu überleben und Memphis zu erobern.

2.3.8. Achtes Kapitel

Octavians Heer machte sich auf den Weg in die Hauptstadt Alexandria. Antonius und Cleopatra ließen Alexandria befestigen und jeden kriegsfähigen Ägypter in die Hauptstadt schicken. Von Rom ließ Octavianus eine der größten Flotten der Antike kommen, da es leichter sein würde die Stadt über den Hafen einzunehmen. Eine große Schlacht stand bevor.

Gameplay

Auf einer sehr großen Karte von Alexandria bis zum nächsten Ausläufer des Nils soll der Spieler Alexandria über den Seeweg erobern. Seine Basis befindet sich an einer entfernten Küste.

2.3.9. Epilog

3. August, 30 vor Christus

Nach der Eroberung Alexandrias erhält Antonius die Meldung Cleopatra wäre tot. Daraufhin nimmt er sich das Leben. Da es Cleopatra nicht gelang, Octavianus für ihre Zwecke einzuspannen, wählte sie den Freitod durch einen Schlangenbiss. Caesarion, den Sohn von Cleopatra und Julius Caesar, ließ Octavianus aufspüren und hinrichten.

Ägypten verlor seine Unabhängigkeit und wurde zur römischen Provinz und persönlichem Eigentum von Octavianus. Mit dem Vermögen Cleopatras wurde der Sold an seinen Soldaten beglichen.

Die Bürgerkriege hatten somit ein Ende gefunden, aber auch die Zeit der römischen Republik.

Octavianus wurde der erste römische Kaiser und erhielt den Namen Augustus.

3. Einfluss auf das Projekt

Die Hintergrundgeschichte hatte weit reichenden Einfluss auf die weitere Planung des Projektes. Sie beeinflusste unter anderem

- die Wahl der Schauplätze,
- die erforderliche Anzahl an 3D-Modellen und Texturen,
- das Design der Häuser und Einheiten,
- das Design der Benutzeroberfläche und
- die Art der Kriegsführung im Spiel.

Erst mit der vollständigen Handlung konnte die Planung des Projektes abgeschlossen werden. Auch im weiteren Projektverlauf musste man sich immer wieder Gedanken über die Handlung und das Szenario machen. So hatte man zum Beispiel bei jedem erstellten 3D-Modell die damalige Zeit und den damaligen Stand der Technik zu beachten.

Projektmanagement
Teil A - Theorie

1. Der Projektprozess

Dieser Absatz soll einen grundsätzlichen Ablauf eines Projektes kurz und übersichtlich darstellen.

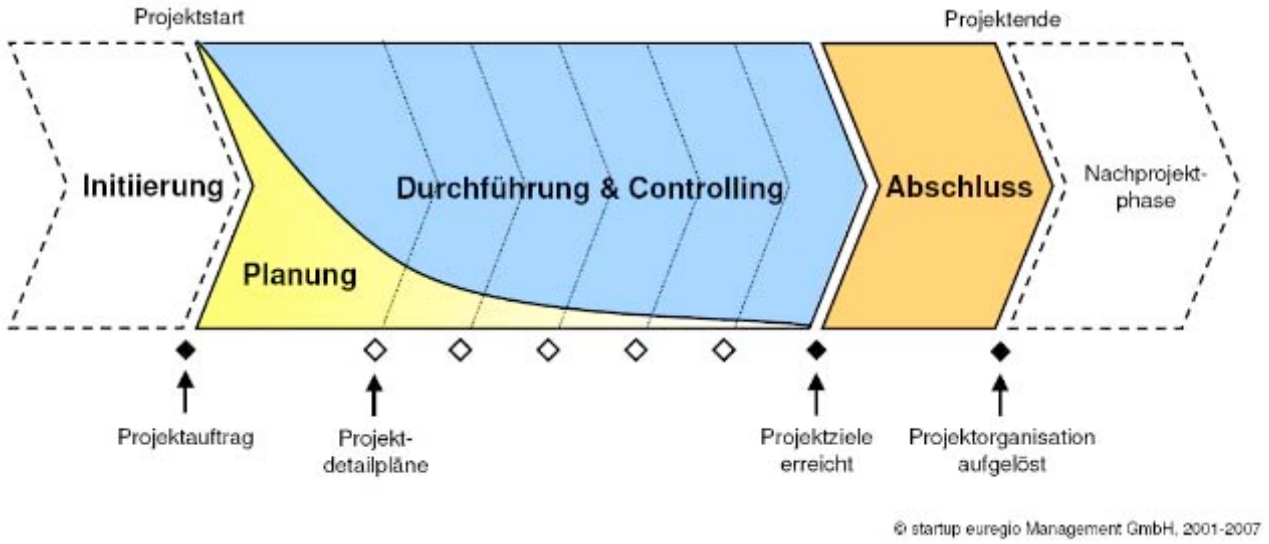


Abb.4: Projektprozess
(http://www.pm-handbuch.com/projektablauf.htm, vom 28.02.2007 um 15:25)

[5]**Die einzelnen Teilprozesse sind:**

- **Initiierung**: Alle für das Projekt wichtigen Informationen(Ideen, Probleme) werden gesammelt, analysiert und geplant. Anschließend wird das Ergebnis in Form eines Projektauftrags gebracht.
- **Planung**: Wenn das Projekt offiziell gestartet ist, konkretisiert das Projektteam in der Planung die Projektinhalte (Ziele, Aufgaben, Risiken etc.).
- **Durchführung & Controlling**: Sobald die Planung einen ausreichenden Detaillierungsgrad erreicht hat, wird mit der Umsetzung begonnen. Parallel dazu steuert und überwacht der Projektmanager den Projektverlauf.
- **Abschluss**: Ein Projekt sollte genauso systematisch beendet werden, wie es begonnen hat. Die Projektergebnisse müssen entsprechend evaluiert werden (on scope, on budget, on time?). Die Ergebnisse des Projektabschlusses sind in einem kurzen Abschlussbericht zu dokumentieren.
- **Nachprojektphase**: In der Nachprojektphase werden die Projektergebnisse genutzt. Oft ist es wichtig und ratsam, auch die Verantwortlichkeiten für die Nachprojektphase klar zu definieren.

Wichtig: Die verschiedenen Teilprozesse können zeitlich parallel ablaufen. So kann es beispielsweise sinnvoll und notwendig sein, Schritte der Planung schon während der Initiierung durchzuführen. Oder umgekehrt können Aufgaben der Initiierung während der Durchführung relevant werden.

---

5 vgl. PM-Handbuch.com - Der Kostenlose Leitfaden für Projektmanager. Online im Internet: URL: http://www.pm-handbuch.com/projektablauf.htm , vom 28.02.2007 15:25

2. Projektstrukturpläne

[6]Ein Projekt ist ein komplexes Gesamtvorhaben, das nur durch eine Aufgliederung in Teilaufgaben bewältigt werden kann. Erst die Strukturierung in möglichst unabhängige Teilaufgaben, ermöglicht eine Arbeitsaufteilung.

Projektstrukturpläne stellen das zentrale Planungsinstrument in einem Projekt dar.

Sie sind weiters Basis für:

- Die zeitliche Projektplanung
- Detaillierte Kostenrechnung
- Laufendes Projektcontrolling

Eigene Bemerkung:

In unserem Projekt spielt die Kostenrechnung keine wichtige Rolle, da es sich um eine Diplomarbeit handelt, welche keinen Auftraggeber besitzt.

Die Kostenrechnung wir deshalb vernachlässigt.

2.1. Aufgaben der Strukturplanung

Um ein Projekt planbar zu machen, muss es in einzelne, gut handhabbare „Portionen“ zerlegt werden.

Es werden drei grundsätzliche Strukturen unterscheiden:

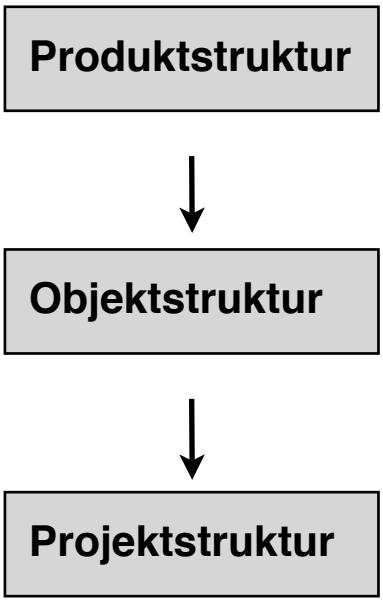
- Die **Produktstruktur**
  Auflistung der zu entwickelnden Produktteile
- Die **Objektstruktur**
  zeigt zusätzlich alle für die Projektrealisierung erforderlichen Objekte

---

6 vgl. Prof. Dipl-Ing. Felix Schwab, o. Univ.-Prof. Dkm. Mag. Dr. Wilfried Schneider, wirkl. HR Prof. (FH) Mag. Ingrid Schwab-Matkovits: EDV Projektentwicklung, 4.überarbeitete Auflage, Wien, MANZ Verlag, 2004. ab Seite 67

- Die **Projektstruktur**
  zeigt weiters alle für die Projektrealisierung erforderlichen Arbeiten („Aufgabenbaum“)

Alle drei Strukturen stehen miteinander in Beziehung und sind **voneinander abhängig!**



2.2. Projektstrukturplan(PSP)

Der PSP ist Voraussetzung für:

- Projektorganisationsplan
- Funktions- und Verantwortungsmatrix
- Balkendiagramm und Netzplan

Ziel des Projektstrukturplanes ist es die Gesamtaufgabe in autonom bearbeitbare Teilaufgaben zu zerlegen. Diese Teilaufgaben werden als Arbeitspakete bezeichnet.

Ein Arbeitspaket ist lt. DIN 69901 „ein Teil des Projektes, der im Projektstrukturplan nicht weiter aufgegliedert ist und auf einer beliebigen Gliederungsebene liegen kann“.

Als Kriterium, wann bei der Untergliederung ein Arbeitspaket erreicht wurde, wird oft die grobe Regel „4/40“-Regel angewandt:

Das bedeutet, dass Aufgaben von unter 4 Stunden zu einem Arbeitspaket zusammengefasst werden sollten und Aufgaben von über 40 Stunden in 2 oder mehrere Arbeitspakete unterteilt werden sollten.

Es gibt verschiedene Formen zur Darstellung der PSP:

- **Objektorientiert**: Die Strukturierung des Projekts richtet sich nach den einzelnen Bestandteilen des Produktes bzw. nach (Teil-)Ergebnissen im Projekt.
- **Tätigkeitsorientiert** (ablauf-, funktionsorientiert): Die Strukturierung des Projektes erfolgt nach den durchzuführenden Arbeitsschritten.
- **Phasenorientiert** (ablauforientiert): Wenn ein streng sequentieller Projektablauf vorliegt, so kann die Strukturierung nach den einzelnen Phasen bzw. den Arbeitsschritten innerhalb der Phasen erfolgen.
- **Gemischt**: Die Strukturierung in den einzelnen Ebenen der Projekthirachie erfolgt nach unterschiedlichen Gesichtspunkten.

[7]**Die klassischen 3 Ebenen eines PSP sind:**

- **Projekttitel** (PT)
- **Teilaufgaben** (TA)
  Teilaufgaben fassen gleichartige Arbeitspakete zusammen. Teilaufgaben sind typischerweise Hauptaufgaben, Funktionen oder Phasen eines Projekts.
- **Arbeitspakete** (AP)

---

7 vgl. PM-Handbuch.com - Der Kostenlose Leitfaden für Projektmanager. Online im Internet: URL: http://www.pm-handbuch.com/planungsphase.htm#2, vom 22.02.2007 12:13

Die Teilaufgaben und Arbeitspakete eines Strukturplans werden nummeriert, um die Übersichtlichkeit und Eindeutigkeit zu verbessern:

TA: 1, 2, 3 ...

AP: 1.1, 1.2, 1.3 ...; 2.1, 2.2 ...

Diese Nummerierung wird als **PSP-Code** bezeichnet.

2.3. [8]Richtlinien zum Erstellen von Strukturpläne

Das Erstellen von Strukturplänen ist ein kreativer sowie "iterativer" Prozess, bei dem es keine Eindeutige Lösungsmöglichkeit gibt. "Iterativ" bedeutet dass der Strukturplan am Beginn nicht vollkommen erstellt werden kann, sondern während des Ablaufes immer wieder verbessert werden sollte.

Grundlegende Ansätze:

- **Top-down** (Deduktion): Der Strukturplan wird ausgehend vom Hauptziel des Projektes erstellt.
- **Bottom-up** (Induktion): Am Beginn werden Projektdetails gesammelt und in eine hierarchische Ordnung gebracht.

In der Praxis erweist sich jedoch meistens eine **Gemischte Form** für passend. Es werden ausgehend von der Top-Down Gliederung, Bottom-up Überlegungen in die Struktur eingegliedert.

Der fertige Strukturplan sollte:

- **Vollständig**: Die Zerlegung in Teilelemente muss so erfolgen, dass am Schluss die Darstellung der Teilelemente wieder die Gesamteinheit ergibt.
- **Überdeckungsfrei**: Die Elemente müssen sich vollkommen unterscheiden und es dürfen keine Wiederholungen vorkommen.

---

8 vgl. Prof. Dipl-Ing. Felix Schwab, o. Univ.-Prof. Dkm. Mag. Dr. Wilfried Schneider, wirkl. HR Prof. (FH) Mag. Ingrid Schwab-Matkovits: EDV Projektentwicklung, 4.überarbeitete Auflage, Wien, MANZ Verlag, 2004. ab Seite 71

3. Zeitplanung

3.1. [9]Meilensteine

Als Meilenstein definiert man ein Zwischenergebnis das inhaltlich und terminlich schon im Vorfeld genau festgelegt ist. Meilensteine sind die Grundlage für die Projektsteuerung, da bei Erreichung ein Projektbericht erstellt wird. Dieser gibt Aufschluss über den tatsächlichen Projektstatus, der anschließend mit dem Soll-Status verglichen wird, um eventuelle Verschiebungen etc. vorzeitig erkennen zu können.

Meilensteine dienen als Orientierungshilfe für die Projektmitarbeiter und den Projektleiter. Die Zusammenfassung der Meilenstein in einer Tabelle wird Meilensteinliste genannt. Je kürzer das Projekt ist, desto kürzer wird die Spanne zwischen den Meilensteinen gewählt. Meilensteinlisten sollten grundsätzlich nur Zeitpunkte enthalten und keine Zeitpunkte.

Beispiel für eine Meilensteinliste:

Meilensteinliste			
Meilenstein	**Ergebnis**	**Soll-Termin**	**Ist-Termin**
1			
2			
3			

In die Spalte “Soll-Termin” wird der am Beginn festgelegte Termin zum Erreichen des jeweiligen Meilensteins eingetragen. In der Spalte “Ist-Termin” trägt man dann schlussendlich den tatsächlichen Termin ein, bei dem man den Meilenstein erreicht hat.

---

9 vgl. Prof. Dipl-Ing. Felix Schwab, o. Univ.-Prof. Dkm. Mag. Dr. Wilfried Schneider, wirkl. HR Prof. (FH) Mag. Ingrid Schwab-Matkovits: EDV Projektentwicklung, 4.überarbeitete Auflage, Wien, MANZ Verlag, 2004. Seite 42

Die Meilensteinliste ist also wie oben schon erwähnt eine einfach Möglichkeit des “Soll - Ist - Vergleichs”.

3.2. Das Balkendiagramm (Gantt-Diagramm)

[10]Das Balkendiagramm ist eines der gebräuchlichsten Hilfsmittel zur zeitlichen Planung bzw. Darstellung des Projektablaufes.

Anhand der Balkendiagramme kann abgelesen werden, wann eine Aktivität beginnt, wie lange sie dauert und wann sie endet.

Wichtig ist, dass die Meilensteine im Balkendiagramm immer speziell gekennzeichnet sind.

Der Vorteil der Diagramme ist bei nicht zu umfangreichen Projekten ihre einfache Handhabung sowie ihre Anschaulichkeit. Bei größeren Projekten werden Balkendiagramme durch Netzpläne ersetzt, da die Zusammenhänge zwischen Aktivitäten und Pufferzeiten nicht mehr auf einen Blick erkennbar sind.

Eigene Bemerkung: Als Grundlage für ein Balkendiagramm verwendet man üblicherweise ein Vorgangsliste, die den Vorgangsnamen, die Dauer und den Vorgänger auflisten.

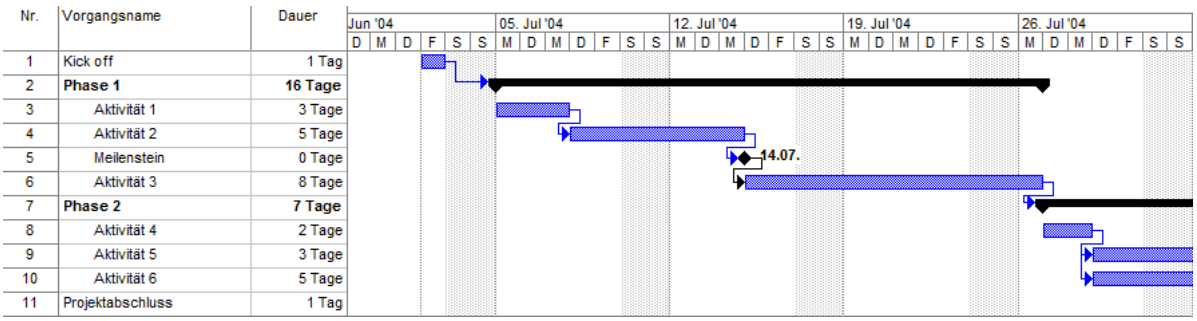


Abb.5: Vorlage Balkendiagramm

(Bildquelle: http://de.wikipedia.org/wiki/Bild:Gantt_diagramm.png, vom 02.04.2007)

---

10 vgl. Prof. Dipl-Ing. Felix Schwab, o. Univ.-Prof. Dkm. Mag. Dr. Wilfried Schneider, wirkl. HR Prof. (FH) Mag. Ingrid Schwab-Matkovits: EDV Projektentwicklung, 4.überarbeitete Auflage, Wien, MANZ Verlag, 2004. ab Seite 77

3.3. Aktions-/Arbeitsplan

[11]Mit Hilfe des Aktionsplans können sehr schnell neue Aufgaben definiert, delegiert und terminiert werden. Dadurch strukturieren Aktionspläne die Abschnitte zwischen den Meilensteinen und helfen die Aufgaben oder auch Arbeitspakete abarbeitbar zu machen.

Welchen Informationen muss ein Aktionspan festhalten:

- **WAS** muss getan werden (Aufgabeninhalt)?
- **WER** ist dafür verantwortlich (Aufgabenträger)?
- **Bis WANN** soll das Ergebnis vorliegen (Termin)?

Beispiel**:**

1. Pos.	1. Was?	1. Wer?	1. Bis Wann?

3.4. Projektstatusbericht

[12]Der Projektleiter ist während der Projektrealisierung dafür verantwortlich, dass alle Projektbeteiligten in ausreichendem Maße über den Projektfortschritt informiert werden. Dies kann in Form von schriftlichen Fortschrittsberichten, Projektsitzungen, Präsentationen, persönlichen Gesprächen o.Ä. erfolgen.

---

11 vgl. Prof. Dipl-Ing. Felix Schwab, o. Univ.-Prof. Dkm. Mag. Dr. Wilfried Schneider, wirkl. HR Prof. (FH) Mag. Ingrid Schwab-Matkovits: EDV Projektentwicklung, 4.überarbeitete Auflage, Wien, MANZ Verlag, 2004. Seite 83

12 vgl. PM-Handbuch.com - Der Kostenlose Leitfaden für Projektmanager. Online im Internet: URL: http://www.pm-handbuch.com/realisierungsphase.htm, vom 02.04.2007 01:02

Darüber hinaus hat das Projektteam für das "Projektmarketing" zu sorgen. Projektmarketing ist die Präsentation und werbende Darstellung eines Projekts innerhalb der beteiligten Anspruchsgruppen (intern und extern). Wesentliche Elemente sind:

- Kommunikation des Projekts unter einem eingängigen Titel
- erfolgsorientierte Darstellung des Arbeitsfortschritts
- regelmäßige Information von Entscheidungsträgern über den Projektfortschritt
- evtl. Gestaltung eines Projekt-Logos, Einrichtung einer Projekt-Webseite

4. Projektteam

[13]Ein leistungsfähiges, zielorientiertes und effizient arbeitendes Projektteam ist einer der wesentlichen Erfolgsfaktoren für den Projekterfolg. Ein motiviertes Team meistert auch schwierige Projektsituationen.

4.1. Einflussfaktoren für die Arbeit im Projektteam

+	-
Identifikation der Gruppenmitglieder mit den Projektzielen	Neid und Konkurrenzverhalten innerhalb der Projektgruppe
Fähigkeit zur Zusammenarbeit	Infragestellen des Projektleiters
Fachwissen der Projektmitglieder	Projektmitglieder vertreten insgeheim andere Interessen

13 vgl. Prof. Dipl-Ing. Felix Schwab, o. Univ.-Prof. Dkm. Mag. Dr. Wilfried Schneider, wirkl. HR Prof. (FH) Mag. Ingrid Schwab-Matkovits: EDV Projektentwicklung, 4.überarbeitete Auflage, Wien, MANZ Verlag, 2004. ab Seite 57

+	-
Fähigkeit zur Kommunikation	einzelne Projektmitglieder in der Projektgruppe haben keinen klaren Auftrag bzw. keine klare Aufgabenstellung
Fähigkeiten zur Kommunikation mit dem Auftraggeber	

Die Kultur des Projektumfelds zeigt sich

- an der Art der Zusammenarbeit
- an der Gesprächskultur (wie wird miteinander geredet)
- an der Art, wie Konflikte ausgetragen werden
- an der Einstellung und Bereitschaft zur (Mehr-) Leistung
- in der Form, ob und wie Leistung belohnt wird
- an der Art, wie Entscheidungen getroffen werden

Besonderen Einfluss auf die Arbeitsfähigkeit eines Teams hat der Projektleiter. Er muss einerseits auf die konsequente Verfolgung der Projektziele achten, andererseits dafür sorgen, dass das Klima in der Gruppe von der gegenseitigen Akzeptanz und Offenheit geprägt wird.

5. [14]Projektcontrolling

Nach erfolgreicher Planung überprüft das Projektcontrolling laufend die Einhaltung der Vorgaben bzw. setzt im Falle von Abweichungen geeignete Maßnahmen. Das Projektcontrolling achtet auf die Einhaltung von Terminen, Kosten und erbrachte Leistungen (Fertigstellungsgrad).

---

14 vgl. Prof. Dipl-Ing. Felix Schwab, o. Univ.-Prof. Dkm. Mag. Dr. Wilfried Schneider, wirkl. HR Prof. (FH) Mag. Ingrid Schwab-Matkovits: EDV Projektentwicklung, 4.überarbeitete Auflage, Wien, MANZ Verlag, 2004. ab Seite 89

5.1. Aufgaben des Projektcontrollings

Je früher in einem Projekt Abweichung vom den geplanten Terminen, Kosten und Leistungen erkannt werden, desto größer sind die Möglichkeiten zur effektiven Gegensteuerung und zur planungskonformen Erreichung des Projektziels.

Aufgaben:

- Planungsabweichungen frühzeitig zu erkennen
- Abweichnungstendenzen zu erkennen und richtig zu interpretieren, um rechtzeitig wirkungsvolle Gegenmaßnahmen setzen zu können.

Voraussetzungen für das Projektcontrolling ist ein geeignetes Projektreporting (in Form von Berichten, Besprechungen,....).

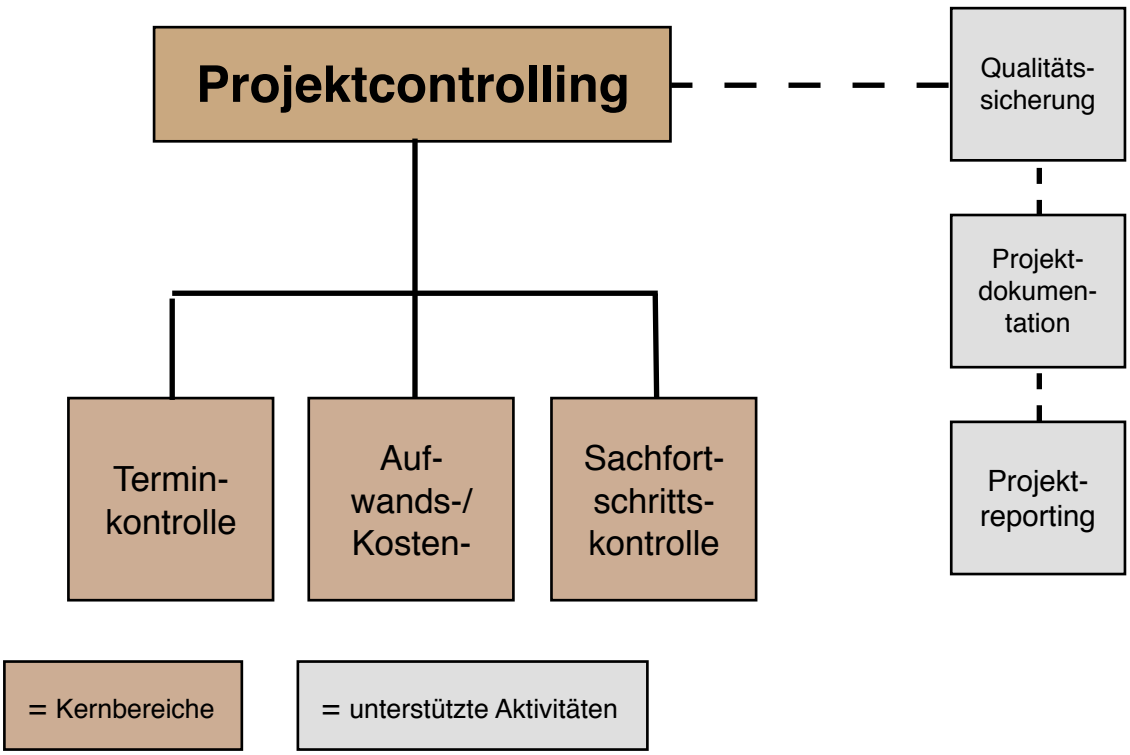


Abb.6: Einfluss Projektcontrolling

(Prof. Dipl-Ing. Felix Schwab, o. Univ.-Prof. Dkm. Mag. Dr. Wilfried Schneider, wirkl. HR Prof. (FH) Mag. Ingrid Schwab-Matkovits: EDV Projektentwicklung, 4.überarbeitete Auflage, Wien, MANZ Verlag, 2004.Seite 89)

Projektcontrolling muss immer auf eine gesamtheitliche Betrachtung des Einzelaspekte Kosten, Termine, Arbeitsfortschritt und Arbeitsqualität abzielen. In jedem dieser Bereiche stehen geeignete Verfahren und Methoden zur Verfügung.

5.2. Terminkontrolle

Es ist wichtig dem Projektleiter regelmäßig und rechtzeitig die „Ist-Termine“ rück zu melden. Der Projektleiter kann dann aufgrund des vordefinierten „Soll-Termins“ herauslesen, ob der Termin eingehalten, überschritten oder vorverlegt vorliegt.

Gründe für eine Terminverschiebung:

- Änderung des Leistungsumfangs oder der Qualitätsanforderung an das zu erstellende Produkt
- unvorhergesehene Probleme bei der Realisierung (technische Probleme, Ressourcenengpässe ...)
- mangelhafte oder unrealistische Schätzung des Aufwands
- unvorhergesehene Personalengpässe durch Krankheiten oder Ausscheiden von Teammitgliedern
- geringe Produktivität des Projektteams durch schlechte Koordination, mangelnde Projekterfahrung oder zu geringe Sachkenntnis

Mögliche Maßnahmen bei einer Verzögerung:

- Mobilisierung zusätzlicher Personalressourcen
- Erhöhung der Arbeitszeit
- Erhöhung der Produktivität durch Einsatz verschiedener Tools
- Outsourcing/Outtasking

[15]**Meilensteine** bieten eine besonders gute Gelegenheit für das Controlling eines Projekts. Anlässlich dieser definierten Zwischenergebnisse können die bisherigen Projektphasen und -schritte kritisch hinterfragt und eventuell notwendige Korrekturen der Projektplanung beschlossen werden.

Bei Meilenstein-Sitzungen sollten alle Projektbeteiligten anwesend sein!

5.3. Möglicher Ablauf eines Projektcontrollings:

1. Laufende **Ermittlung** von aktuellen **Ist-Daten**
   - ★ Qualität, Leistungen, Ergebnisse
   - ★ Kosten, Arbeitsaufwände
   - ★ Zeit, Termine, Bearbeitungsdauern
2. Gegenüberstellung mit den **Soll-Daten**
3. Untersuchung eventueller **Abweichungen**
4. Durchführung von geeigneten **Gegenmaßnahmen**

5.4. Sachfortschrittskontrolle

[16]Im Rahmen der Sachfortschrittskontrolle muss festgelegt werden, zu wieviel Prozent eine bestimmte Teilaufgabe bereits abgeschlossen ist. Wichtig hierbei ist, dass diese Angaben nur den tatsächlichen Fortschritten entsprechen dürfen und nicht den geplanten!

---

15 vgl. PM-Handbuch.com - Der Kostenlose Leitfaden für Projektmanager. Online im Internet: URL: http://www.pm-handbuch.com/realisierungsphase.htm , vom 04.04.2007 20:36

16 vgl. Prof. Dipl-Ing. Felix Schwab, o. Univ.-Prof. Dkm. Mag. Dr. Wilfried Schneider, wirkl. HR Prof. (FH) Mag. Ingrid Schwab-Matkovits: EDV Projektentwicklung, 4.überarbeitete Auflage, Wien, MANZ Verlag, 2004. ab Seite 92

Möglichkeiten zur Fortschrittsmessung:

- Statusschritt-Technik: Bei Projekten mit klar definierten Meilensteinen und Leistungsabschnitten
- 50-50-Technik: Am Beginn einer Arbeit an einem Arbeitspaket wird der Fortschrittsgrad auf 50% gesetzt und bleibt bis zur Beendigung der Arbeit auf diesem Wert. Nach Beendigung wird der Fortschritt auf 100% gesetzt.
- 0-100-Technik: Der Fortschrittsgrad bleibt so lange auf 0%, bis das Arbeitspaket vollständig erfüllt ist. Anschließend wird er auf 100% gesetzt.
- Mengen-Proportionalität: Anwendbar wenn mess- oder zählbare Ergebniseinheiten vorliegen.
- Zeit-Proportionalität: Für bestimmte Projekttätigkeiten, wie zB das Projektmanagement, für die eine „gleichmäßige Verteilung“ über die gesamte Dauer des Projektes angenommen wird.

6. Projektabschluss

[17]Die Evaluierung soll insbesondere den Projekterfolg messen. Diese Evaluierung sollte zumindest die Beantwortung folgender Fragen umfassen:

- War das Projekt "on scope - on budget - on time"?
- Ist der Kunde zufrieden?
- Wie sieht es mit der Stimmung im Projektteam aus?

Darüber hinaus sollte der Projektverlauf mit möglichst allen Beteiligten reflektiert werden. Was lief gut? Was lief schlecht? Was können wir beim nächsten Mal anders machen?

---

17 vgl. PM-Handbuch.com - Der Kostenlose Leitfaden für Projektmanager. Online im Internet: URL: http://www.pm-handbuch.com/abschlussphase.htm, vom 04.04.2007 11:52

6.1. Projektabschlussbericht

Auf der Basis der Projektdokumentation und den Ergebnissen der Projektabschlusssitzung wird ein kurzer Abschlussbericht erstellt. Dieser sollte in komprimierter Form die Ergebnisse der Evaluierung & Reflexion enthalten.

Mögliche Inhalte eines Projektabschlussberichts sind:

- Projektbeschreibung (Ausgangssituation, Ziele, Inhalte, Budget)
- Projektdetailplanung (PSP, Terminplan, Kostenplan)
- Projektrealisierung (Projektänderungen, Controlling-Maßnahmen, Probleme und Störungen während der Durchführung, Erfahrungen)
- Projektergebnisse (Soll-Ist-Vergleich in Bezug auf Qualität, Kosten, Zeit)
- Vorschlag zu weiterem Vorgehen (z.B. Folgeprojekte, Umsetzung)
- Anregungen für künftige Projekte (lessons learned)

Projektmanagement
Teil B - Umsetzung

1. Projektstrukturplan

Projektstruktur	Aufgabenzuteilung	Fertigstellung
1. Schnittstellen		**27.10.2006**
1.1 Grafikprogrammierung	Steinlechner Harald	**=74,5%**
1.1.1 Entwickler Tools		**= 96,6%**
1.1.1.1 AAX-Toolkit		95%
1.1.1.2 AAX Viewer		100%
1.1.1.3 Leveleditor		95%
1.1.2 Darstellung		**=92%**
1.1.2.1 Animation		100%
1.1.2.2 Spezialeffekte		80%
1.1.2.3 Interface	Georg Haaser	90%
1.1.2.4 Darstellung		*= 90%*
1.1.2.4.1 Umgebung		80%
1.1.2.4.2 Spielelemente		100%
1.1.2.4.3 Font-Rendering		100%
1.1.3 Optimierung		**=35%**
1.1.3.1 Sichtbarkeitstests - Culling		80%

1.1.3.2 Pipeline Optimierung		20%
1.2 Programmkern	Steinlechner Harald	**=61,4%**
1.2.1 Benutzerführung	Wieser Manuel	10%
1.2.2 Kamerasteuerung		80%
1.2.3 Filemanagement		100%
1.2.4 Fehlerverwaltung		20%
1.2.5 Cleanup		60%
1.2.6 Portierung (mac,windows Kompatibilität)		80%
1.2.7 Einstellungen + Settingstools		80%
1.3 Audioprogrammierung	Wieser Manuel	**= 100%**
1.3.1 Soundeffekte		100%
1.3.2 Musik		100%
2. Audiovisuelle Inhalte	Wieser Manuel	
2.1 FMVs		
2.1.1 Objekte erstellen		
2.1.2 Composition		
2.1.3 Animation		
2.1.4 Rendern		
2.1.5 Nachbearbeitung		
2.2 Echtzeitmodelle		**=43,75%**
2.2.1 Texturen		40%
2.2.2 Modelle		40%
2.2.3 Animation		60%
2.2.4 Export		35%

2.3 Audio		
3. KI (Künstliche Intelligenz)	Haaser Georg	
3.1 Fuzzy Logic		
3.1.1 Beeinflussung der Spielerseite		
3.1.1.1 Bürger sollen selbst-ständig entscheiden		
3.1.1.2 Faktoren Motivation und Müdigkeit beeinflussen		
3.1.2 Einfluss von Umweltfaktoren (Randomisierung)		
3.1.2.1 Wetter, Naturkatastro-phen		
3.1.2.2 Fruchtbarkeit der An-baugebiete		
3.1.3 Fuzzy-Logik Engine		
3.1.3.1 Fuzzy-Logik Editor		
3.1.3.2 Fuzzy-Logik Auswertung (Defuzzyfierzung)		
3.1.4 Entscheidungsbaum des Gegners		
3.1.4.1 Editor		
3.1.4.2 Auswertung		
3.2 Wegfindung	Steinlechner Harald	**=15%**
3.2.1 Abstraktion		20%
3.2.2 Auswertung		10%
3.3 Wirtschaftssystem	Haaser Georg	
3.3.1 Benutzerschnittstellen		
3.3.2 Regelwerk		
3.4 Einfacher (linearer) Computergegner	Haaser Georg	
3.4.1 Entscheidungsbaum, Editor		
3.4.2 Implementierung		

4. Physik		**=100%**
4.1 Kolissionserkennung		100%
4.2 Vektor- und Matritzenrechnungsframework	Haaser Georg	100%
5. Gameplay	Steinlechner Harald	
5.1 Story		**=50%**
5.1.1 Hintergrundgeschichte		100%
5.1.2 Checkpoints		0%
5.1.3 Kamerafahrten		
5.1.4 Storyimplementation (Hinweistafeln)		
5.2 Spielelemente		**=37,5%**
5.2.1 Einheiten		20%
5.2.2 Gebäude		30%
5.2.3 Technologiebaum		100%
5.2.4 Balancing		0%
6. Test		
6.1 Technische Tests	Steinlechner Harald	
6.2 Inhaltliche Tests	Gassler Severin	
7. Projektmanagement	Gassler Severin	
7.1 Projektplanung		
7.2 Projektcontrolling		
7.3 Projektkoordination		
Projektdokumentation		

Der **Fertigstellungsgrad** hat dem Projektmanagement als Grundlage gedient, da Großteile des Grundprogrammes schon vor der Diplomarbeitszieldefinition erzeugt wurden.

Es handelt sich bei unserer Diplomarbeit um eine Mischform vom „Bottom-Up“ und „Top-Down“ Ansatz.

Die „Bottom-Up“ Teilgebiete sind mit einem hohen Fertigstellungsgrad eingetragen und werden deshalb in der Planung auch nur zu einem geringen Teil berücksichtigt.

Die „Top-Down“ Arbeitsschritte, die vom definierten Hauptziel der Diplomarbeit abgeleitet wurden, haben keinen bzw. einen geringen Fertigstellungsgrad und wurden deshalb vollständig in die Planung miteinbezogen.

Mit Hilfe dieses oben erwähnten Fertigstellungsgrad wurde es uns ermöglicht aus der komplizierten Mischform eine kontinuierliche Zeitplanung zur erzeugen und damit auch schlussendlich auch das Hauptziel erreichen.

Die Spalte **Arbeitszuteilung** soll auf einen Blick zeigen wer in der Diplomarbeit für welche Gebiete zuständig ist und ist später sehr hilfreich für das Erstellen des Arbeitsplans.

2. Zeitplanung

2.1. Meilensteinliste

MEILENSTEINLISTE, vom 20.10.2006

1. Meilenstein

Definition: Mit Hilfe von **Fuzzy Logic** sollen gewisse Teile des Spieles **auf Seite des Spielers** beeinflusst werden und so eine gewisse Unschärfe ins Spiel bringen.

Bis: Letzte Woche vor Weihnachten: **18.12.2006**

Beschreibung: Das Spiel soll mittels Fuzzy Logic auf gewisse Entscheidungen des Spielers reagieren. Genauer gesagt soll der Spieler nicht in der Lage sein alles selbst zu entscheiden denn,

a) Die Bürger sollen selbständig entscheiden ob sie in eine Stadt gehen und welchen Beruf sie dort ausüben -> der Spieler muss also dafür sorgen, dass er eine attraktive Stadt aufbaut.
b) Gewisse Faktoren wie Müdigkeit und Motivation eines Bürgers tragen dazu bei dass sich gewisse Handlungen des Spielers verlangsamen (z.B. bei Müdigkeit -> Haus bauen wird länger dauern) oder überhaupt erst ermöglichen.

Dieser Unsicherheiten tragen natürlich dazu bei, dass das Spiel Auswirkungen zeigt und der Spieler nicht sofort in der Lage ist zu reagieren.

2. **Meilenstein**

Definition: Es soll die Möglichkeit genommen werden, dass der Spieler immer denselben Weg gehen kann -> Einfluss von Umweltfaktoren – Randomisierung

Bis: Mitte Jänner: **15.01.2007 – 19.01.2007**

Beschreibung: Die Umweltfaktoren sollen ein Spiel so beeinflussen, das eine gewisse Unsicherheit ins Spiel gebracht wird. Dies geschieht mit Hilfe einer Randomisierung: Zufällige Ereignisse ändern das Umfeld des Spielers.
Folgend Punkte sollen randomisiert werden:

a. Wetter – Naturkatastrophen -> Schaden
b. Unterschiedliche Fruchtbarkeit der Anbaugebiete

---

3. **Meilenstein**

Definition:

- Einbau eines vereinfachten Computergegners
- Aufbau eines einfachen Wirtschaftssystems
- Story ins Spiel einbauen

Bis: 2. April 2007 (definiert am 15.01.2007)

Beschreibung:

1. Der Computergegner soll im Spiel selber nur ganz einfach implementiert werden, die Vervollständigung erfolgt dann nur theoretisch.
2. Um auch dem Namen Wirtschaftssimulation gerecht zur werden, wird ein Wirtschaftssystem System eingebaut, welches zur Spielzeit (Antike) im Einsatz war (Lohn, Steuern, ...).
3. Der Einbau der Story stellt einen weiteren wichtigen Punkt dar, es sollen vor allem Hinweisfelder sowie spezielle Kamerafahrten eingebaut werden, um dem Spieler ein realistisches, antikes Umfeld zu schaffen.

In unserer Diplomarbeit wird die Meilensteinliste auch genutzt um Teilziele zu definieren. Dies lässt sich dadurch begründen, da es nur sehr wenig Meilenstein gibt und es daher wesentlich übersichtlicher ist die Definition der Teilziele in die Meilensteinliste zu integrieren. Aus diesem Grund sind in der Liste nicht nur die Meilensteine aufgelistet, sondern auch genauestens definiert.

2.2. Das Balkendiagramm

Als aller erstes haben wir eine Vorgangsliste erstellt. Mit Hilfe dieser Vorgangsliste erstellt man dann aus Gründen der Übersichtlichkeit ein Balkendiagramm, welches für das Projektcontrolling von großer Bedeutung ist.

Planungsstart: Montag, 06.11.2006

Nr	Vorgangsname	Dauer	Vorgänger	Zeitraum
1	Fuzzy Logic Editor	2 Wochen		06.11 - 19.11.2006
2	Fuzzy Logic Auswertung	1 Woche	1	20.11 - 26.11.2006
3	Bürger sollen selbstständig entscheiden können	2 Wochen	2	27.11 - 03.12.2006
4	Faktoren "Motivation" "Müdigkeit" einbauen	1 Woche	3	11.12 - 17.12.2006
5	Wegfindung - Abstraktion + Auswertung	1 Woche		06.11 - 12.11.2006
6	Spezialeffekte	1,5 Wochen		13.11 - 22.11.2006
7	Benutzerführung	1,5 Wochen	6	23.11 - 03.12.2006
8	Gameplay - Checkpoints	2 Wochen	7	04.12 - 17.12.2006
9	Erstellen von verschiedenen visuellen Inhalten	6 Wochen		06.11 - 17.12.2006
10	MEILENSTEIN 1			18.12.2006
	Arbeitsaufwand pro Mitarbeiter	6 Wochen		
	Arbeitsaufwand gesamt	18 Wochen		06.11 - 17.12.2006
11	Umweltfaktoren: Wetter, Katastrophen	2 Wochen	10	18.12 - 31.12.2006
12	Randomisierung: Fruchtbarkeit der Anbaugebiete	2 Wochen	11	01.01 - 14.01.2007
13	Fehlerverwaltung	2 Wochen	10	18.12 - 31.12.2006
14	Pipline-Optimierung	2 Wochen	13	01.01 - 14.01.1007
15	Erstellen von verschiedenen visuellen Inhalten	4 Wochen	10	18.12.2006 - 14.01.2007
16	MEILENSTEIN 2			15.01.2007
	Arbeitsaufwand pro Mitarbeiter	4 Wochen		
	Arbeitsaufwand gesamt	12 Wochen		18.12.2006 - 14.01.2007
17	Planungsphase für M3 - Besprechnugen			15.01 - 28.01.2007
18	Wirtschaftssystem - Regelwerk programmieren	3 Wochen	17	29.01 - 18.02.2007
19	Wirtschaftssystem - Benutzerschnittstellen	2 Wochen	18	19.02 - 04.03.2007
20	Entscheidungsbaum Computergegner	2 Wochen	19	05.03. - 18.03.2007
21	Computergegner: Regelwerk	2 Wochen	20	19.03 - 01.04.2007
22	Story - Kamerafahrten	2 Wochen	17	29.01 - 11.02.2007
23	Storyimplementierung	2 Wochen	22	12.02 - 25.02.2007
24	Pipline-Optimierung: sonstige Optimierungen	5 Wochen	23	26.02 - 01.04.2007
25	Erstellen von verschiedenen visuellen Inhalten	9 Wochen	17	29.01 - 01.04.2007
26	MEILENSTEIN 3			02.04.2007
	Arbeitsaufwand pro Mitarbeiter	9 Wochen		
	Arbeitsaufwand gesamt	27 Wochen		29.01 - 01.04.2007

Abb.7: Vorgangsliste,
erstellt am 20.10.2006

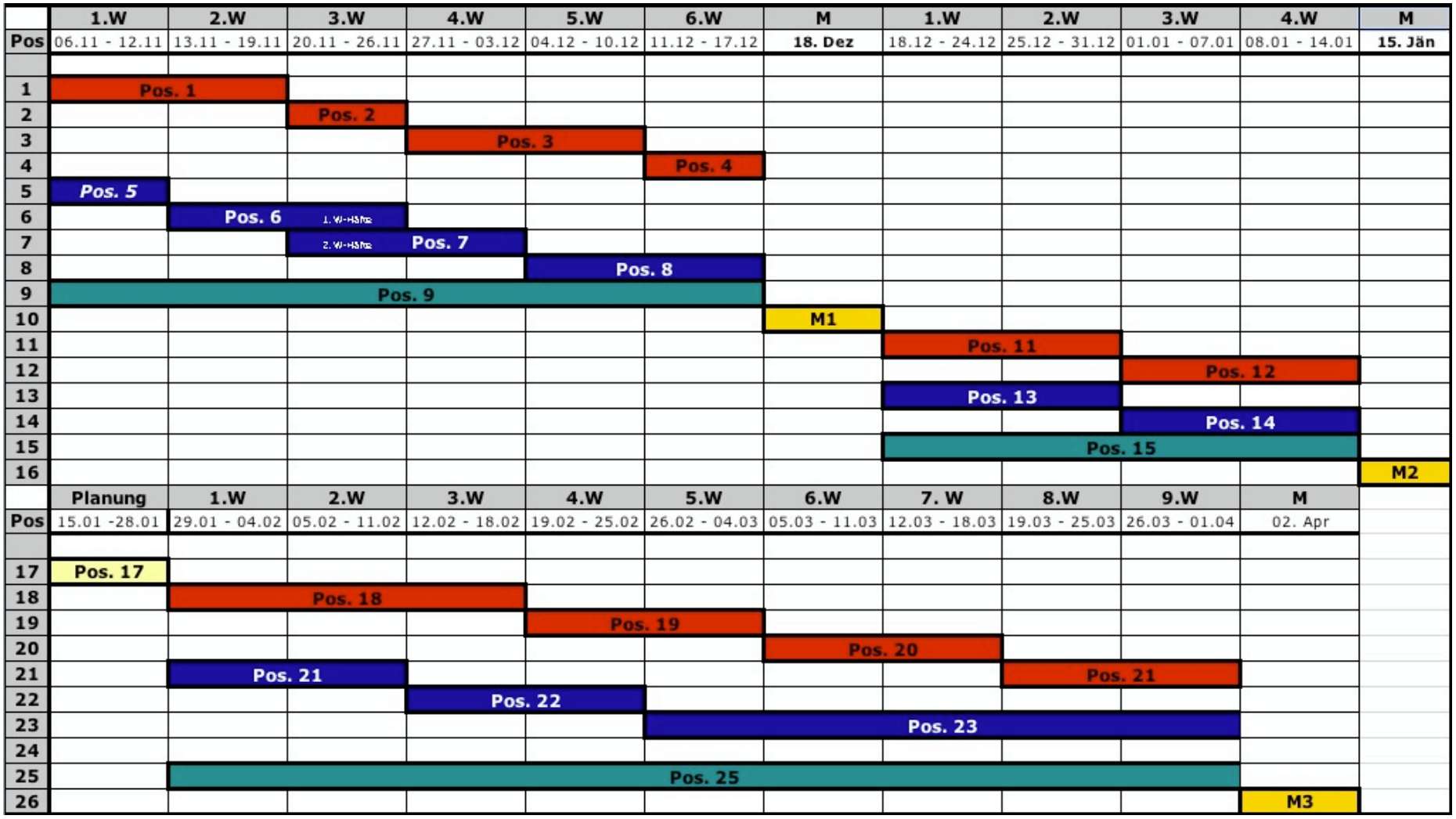


Abb.8: Balkendiagramm,

erstellt am 21.10.2006

Um auch sofort erkennen zu können welcher Mitarbeiter für welches Arbeitspaket zuständig ist, wurden folgende Farbzuweisungen festgelegt:

Nr	Mitarbeiter	Farbe
1	Haaser Georg	rot
2	Steinlechner Harald	blau
3	Wieser Manuel	grün

Abb.9: Farbzuordnung der Mitarbeiter

2.3. Arbeitsplan

In unsere Diplomarbeit wurde hier noch eine zusätzliche Spalte hinzugefügt, um die Koordination untereinander sichtbar zu machen. Außerdem bezieht sich die Pos. auf die Positionsnummer in der Vorgangsliste bzw. Balkendiagramm.

Arbeitsplan

AUGUSTUS

Für: Haaser Georg

Mitarbeiter: Steinlechner Harald, Wieser Manuel, Gassler Severin

Pos	Beschreibung	Koordination	Start	Ende	Bemerkung
1	Fuzzy Logic Editor		06.11.2006	19.11.2006	
2	Fuzzy Logic Auswertung	Steinlechner Harald	20.11.2006	26.11.2006	

3	Bürger sollen in der Lage sein selbstständig zu entscheiden		27.11.2006	10.12.2006	
4	Faktoren:Müdigkeit, Motivation einbauen		11.12.2006	17.12.2006	
10	**MEILENSTEIN1**	Projektteam	19.12.2006		
11	Einbau von Umweltfaktoren: Wetter, Katastrophen	Steinlechner Harald	19.12.2006	31.12.2006	
12	Randomisierung: Fruchtbarkeit der Anbaugebiete	Steinlechner Harald	01.01.2007	14.01.2007	
16	**MEILENSTEIN 2**	Projektteam	16.01.2007		
17	Planungsphase für M3	Projektteam	15.01.2007	28.01.2007	
18	Wirtschaftssystem: Regelwerk programmieren		29.01.2007	18.02.2007	
19	Wirtschaftssystem: Benutzerschnittstellen		19.02.2007	04.03.2007	
20	Entscheidungsbaum Computergegner		05.03.2007	18.03.2007	
21	Computergegner Regelwerk		19.03.2007	01.04.2007	
26	**MEILENSTEIN 3**	Projektteam	02.04.2007		
Hinweis: Die „Pos.“ Bezieht sich auf den Ablaufplan!					

Arbeitsplan

AUGUSTUS

Für: Steinlechner Harald

Mitarbeiter: Haaser Georg, Wieser Manuel, Gassler Severin

Pos	Beschreibung	Koordination	Start	Ende	Bemerkung
5	Wegfindung - Abstraktion		06.11.2006	09.11.2006	
5	Wegfindung - Auswertung		10.11.2006	12.11.2006	
6	Spezialeffekte		13.11.2006	22.11.2006	
7	Benutzerführung		23.11.2006	03.12.2006	
8	Gameplay - Checkpoints		04.12.2006	18.12.2006	
10	**MEILENSTEIN 1**	Projektteam	19.12.2006		
13	Fehlerverwaltung		19.12.2006	31.12.2006	
14	Pipline-Optimierung		01.01.2006	14.01.2007	
16	**MEILENSTEIN 2**		16.01.2007		
17	Planungsphase für M3	Projektteam	15.01.2007	28.01.20007	
22	Story: Kamerafahrten		29.01.2007	11.02.2007	
23	Storyimplementierung		12.02.2007	25.02.2007	
24	Pipline Optimierung fertig; sonstige Optimierungen		26.02.2007	01.04.2007	
26	**MEILENSTEIN 3**	Projektteam	02.04.2007		
Hinweis: Die „Pos.“ Bezieht sich auf den Ablaufplan!					

Arbeitsplan

AUGUSTUS

Für: Wieser Manuel

Mitarbeiter: Haaser Georg, Steinlechner Harald, Gassler Severin

Pos	Beschreibung	Koordination	Start	Ende	Bemerkung
9	GUI-Bild für Erz, Schwert, Schild, und Bogen		02.10.2006	15.10.2006	
9	Mine		16.10.2006	22.10.2006	
9	Kaserne		23.10.2006	29.10.2006	
9	Schmiede		30.10.2006	12.11.2006	
9	Bogenschütze		13.11.2006	26.11.2006	
9	Mühle		27.11.2006	10.12.2006	
9	GUI-Bindeelemete überarbeiten		11.12.2006	17.12.2006	
10	**MEILENSTEIN 1**	Projektteam	19.12.2006		
14	Felder		18.12.2006	27.12.2006	
15	Marktplatz		28.12.2006	07.01.2007	
15	Wirtshaus		08.01.2007	21.01.2007	
16	**MEILENSTEIN 2**	Projektteam	16.01.2007		
	Brunnen		22.01.2007	28.01.2007	
	Hafen		29.01.2007	11.02.2007	
	Fischerboot		12.02.2007	25.02.2007	
	Transportschiff		26.02.2007	11.03.2007	
	Kriegsschiff		12.03.2007	25.03.2007	
26	MEILENSTEIN 3	Projektteam	02.04.2007		

Hinweis: Die „Pos.“ Bezieht sich auf den Ablaufplan!

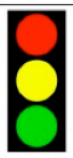
Diese Arbeitsschritte sind im Ablaufplan nicht so detailliert dargestellt, da Sie das Erreichen des Diplomarbeitsziels nicht beeinflussen!

2.4. Projektstatusbericht

Für unsere Diplomarbeit wurden 3 Statusberichte (zu jedem Meilenstein einer) erstellt. Die Vorlage für diesen Bericht stammt von http://www.pm-handbuch.com/realisierungsphase.htm (vom 15.12.2006)

Projektstatusbericht

Projekttitel:	Augustus Wirtschaftssimulation mittels Künstlicher Intelligenz
Projektteam	Haaser Georg, Steinlechner Harald, Wieser Manuel, Gassler Severin
Bericht eingereicht bei:	Dr. Mauracher Armin
Aktuelles Datum:	19.12.2006

Berichtszeitraum:	von 06.11.2006 bis 18.12.2006
Status:	 ☐ kritisch ☐ teilweise kritisch ☒ planmäßig
Kurzbeschreibung Status:	Die ersten Teile der KI (Fuzzy Logic) wurden ins Projekt eingebaut

Beschreibung des Fortschrittes	• **Einbau von Fuzzy Logic:**
	  ○ **Das selbstständige Entscheiden der Bürger, ob sie in eine Stadt einziehen, wurde vollständig eingebaut.**
	  Entscheidend sind die Faktoren:
	  ▪ **Anzahl der Brunnen**
	  ▪ **Anzahl der Verpflegungseinrichtungen**
	  ○ **Die Faktoren „Müdigkeit“ und „Motivation“ spielen eine große Rolle für das Fortschreiten des Spiels.**
	  ▪ **Wenn ein Arbeiter viele Häuser baut, wird er müder und die nachfolgenden Aufgaben werden langsamer bewältigt.**
	  ▪ **Wird z.B. das Wetter schlecht so hat das wieder Auswirkungen auf die Arbeitsgeschwindigkeit des Arbeiters.**
	  ○ Wegfindung: Das Ziel der Wegfindung war, dass sich die Figuren einen sinnvollen Weg durch die Umgebung suchen. (Umgehen von Häusern etc.) Die Wegfindung muss allerdings hinsichtlich der Performance noch verbessert werden.
	• Checkpoints wurden eingerichtet
	• Benutzerführung (einige Menüs fehlen noch!)
	• Spezialeffekte
	• Visuelle Inhalte
	  ○ Modellierung von:
	  ▪ GUI Bild für Erz, Schwert, Schild und Bogen
	  ▪ Mine
	  ▪ Kaserne
	  ▪ Bogenschütze
	  ▪ Mühle
	  ○ GUI-Bindeelemente überarbeitet

Status Termine:	• **Fertig gestellt bis 12.11.2006:**

	○ Wegfindung • **Fertig gestellt bis 19.11.2006:** ○ Fuzzy Logic Editor • **Fertig gestellt bis 23.11.2006:** ○ Spezialeffekte • **Fertig gestellt bis 26.11.2006:** ○ Fuzzy Logic Auswertung • **Fertig gestellt bis 03.12.2006:** ○ Benutzerführung • **Fertig gestellt bis 10.12.2006:** ○ Faktoren „Motivation“ und „Müdigkeit“ (getauscht mit nächstem Punkt; **AP-Nr. 4 vor Ap-Nr.3** Begründung: Reihenfolge unrelevant, AP-Nr. 4 leichter für den Beginn von Fuzzy Logic) • **Fertig gestellt bis 17.12.2006:** ○ Selbstständiges Entscheiden der Bürger ○ Gameplay - Checkpoints ○ Teile der visuellen Inhalte

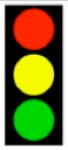
Nächste Schritte bis zum nächsten Meilenstein:	• Performance der Wegfindung verbessern • Benutzerführung verfollständigen • Einfluss von Umweltfaktoren: Wetter, Katastrophen • Randomisierung: Fruchtbarkeit der Anbaugebiete • Fehlerverwaltung • Pipeline - Optimierung • Erstellen von weitern visuellen Inhalten

Nächste Meilensteine:	• Meilenstein2: 15.Jänner • Meilenstein3: **Termin und genaue Definition noch unbekannt!**

Notwendige Entscheidungen	• Definition und Terminvergabe für den 3. Meilenstein!

Projektstatusbericht

Projekttitel:	Augustus Wirtschaftssimulation mittels Künstlicher Intelligenz
Projektteam	Haaser Georg, Steinlechner Harald, Wieser Manuel, Gassler Severin
Bericht eingereicht bei:	Dr. Mauracher Armin
Aktuelles Datum:	15.01.2007

Berichtszeitraum:	von 18.12.2006 bis 15.01.2007
Status:	 ☐ kritisch ☐ teilweise kritisch ☒ planmäßig
Kurzbeschreibung Status:	weitere Teile der KI (Fuzzx Logic) wurden eingebaut; Fehlerverwaltung

Beschreibung des Fortschrittes	• **<u>Randomisierung:</u>**
	Um eine gewisse Unsicherheit ins Spiel zu bringen und den Spieler nicht immer den selben Weg gehen zu lassen wurde eine Randomisierung in folgenden Bereichen einbebaut:
	  ○ Wetter und Naturkatastrophen werden zufällig vom Spiel gesteuert. Auswirkungen hat dies z.B. auf die Motivation der Bürger
	  ○ Die Fruchtbarkeit der Anbaugebiete wurde randomisiert, um nicht immer den gleichen Ernteertrag zu erzielen.
	• Fehlerverwaltung
	• Piplineoptimierung: auf der unerwartet großen Vorstudie für diesen Bereich konnte dieser Beriech nicht vollständig fertig gestellt werden. Dies hat aber keine Auswirkung für den weiteren Projektablauf.
	• Visuelle Inhalte
	  ○ Modellierung von:
	  ▪ Felder
	  ▪ Marktplatz
	  ▪ Wirtshaus

Status Termine:	• **Fertig gestellt bis 31.12.2007:**
	  ○ Randomisierung: Wetter
	  ○ Fehlerverwaltung
	  ○ Visuelle Inhalte: Felder, Marktplatz, Brunnen
	• **Fertig gestellt bis 14.01.2007:**
	  ○ Randomisierung: Fruchtbarkeit der Anbaugebiete

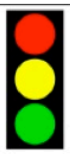
Nächste Schritte bis zum nächsten Meilenstein:	• Optimierung des Spiels • Wirtschaftssystem aufbauen • Dokumentation schreiben • Erstellen von weitern visuellen Inhalten • Piplineoptimierung fertig stellen

Nächste Meilensteine:	• Meilenstein3: **Termin und genaue Definition noch unbekannt!**

Notwendige Entscheidungen	• Definition und Terminvergabe für den 3. Meilenstein! • Entscheidung über die Entwicklung eines Computergegners

Projektstatusbericht

Projekttitel:	Augustus Wirtschaftssimulation mittels Künstlicher Intelligenz
Projektteam	Haaser Georg, Steinlechner Harald, Wieser Manuel, Gassler Severin
Bericht eingereicht bei:	Dr. Mauracher Armin
Aktuelles Datum:	02.04.2007

Berichtszeitraum:	von 15.01.2007 bis 02.04.2007
Status:	 ☐ kritisch ☐ teilweise kritisch ☒ planmäßig
Kurzbeschreibung Status:	Wirtschaftssystem und vereinfachter Computergegner

Beschreibung des Fortschrittes	• **Wirtschaftssystem:**
	Ein einfaches Wirtschaftssystem mit Löhnen etc. wurde ins Spiel eingebaut
	• **Computergegner:**
	Ein einfacher linearer Computergegner ist ab sofort auch im Spiel zu finden, die Vervollständigung erfolgt allerdings nur theoretisch.
	• **Story:**
	Dem Spieler wurde eine realistisches Umfeld in der Antike geschaffen. (Kamerafahrten...)
	• Optimierungen:
	  ○ Fertigstellung der Piplineoptimierung
	  ○ Sonstige Optimierungen
	• Modellierungen von:
	  ○ Brunnen
	  ○ Hafen
	  ○ Fischerboot
	  ○ Transportschiff
	  ○ Kriegsschiff

Status Termine:	• **Fertig gestellt bis 11.02.2007:**
	  ○ Story - Kamerafahrten
	• **Fertig gestellt bis 18.02.2007:**
	  ○ Wirtschaftssystem: Regelwerk
	• **Fertig gestellt bis 25.02.2007:**
	  ○ Storyimplementierung
	• **Fertig gestellt bis 04.03.2007:**
	  ○ Wirtschaftssystem: Benutzerschnittstellen
	• **Fertig gestellt bis 18.03.2007:**
	  ○ Entscheidungsbaum für den Computergegner
	• **Fertig gestellt bis 01.04.2007:**
	  ○ Computergegner: Regelwerk

	○ Optimierungen ○ Erstellen verschiedener visueller Inhalte

Nächste Schritte bis zum Projektende:	• Dokumentation • Vorbereitung auf die Präsentation • Fertigstellung

Nächste Meilensteine:	• keine! **(Projektabschluss)**

Notwendige Entscheidungen	• keine

Game Engine

1. Allgemein

„Spiel-Engines stellen heute die Basis für die Entwicklung von Spielen aller Art dar. Sie enthalten in der Regel Komponenten für die Darstellung (Graphics Engine), zur Berechnung physikalischer Effekte und Erkennung von Kollisionen (Physics Engine) sowie Komponenten für Audio, Animationen, Künstliche Intelligenz und Netzwerkbetrieb mit mehreren Spielern. Sie ermöglichen insbesondere die einfache Modellierung von Spielvorgängen und deren Verknüpfung mit Ereignissen bezüglich der aufgeführten Komponenten.“ [18]

„Eine Game Engine wird auch des Öfteren als „Betriebssystem für Spiele“ bezeichnet. Es existieren Engines, die sich nur für die Erstellung von Spielen eines bestimmten Genres eignen und dafür optimiert sind. So z.B. einige Engines für Spiele aus der Ego-Perspektive, die nur in Gebäudekomplexen angesiedelte Szenarien und keine weitläufigen Außenlandschaften zulassen, dafür aber über sehr ausgefeilte Beleuchtungsmodelle verfügen.

Andere Engines hingegen versuchen eine möglichst große Bandbreite an Funktionalität anzubieten, um die auf ihrer Basis erstellten Spiele so wenig wie möglich im Genre und in den Möglichkeiten einzuschränken. Diese Engines lassen Spieltypen mit völlig unterschiedlichen Anforderungen zu oder ermöglichen sogar die Verschmelzung verschiedener Genres. Ein Beispiel dafür sind Spiele, in denen der Spieler in großflächigen Arealen umherlaufen kann, aber auch in der Lage ist, in der Szene befindliche Flugzeuge zu verwenden, um mit diesen im Stil eines Flugsimulators das gleiche Areal zu überfliegen. Durch die Vielseitigkeit bei der Nutzung einer Engine sind Spiele möglich, die den Spieler wesentlich stärker in das Spielgeschehen eintauchen lassen.“ [19]

---

18 http://www.fit.fraunhofer.de/gebiete/mixed-reality/diplomaTh/DA_MRGameEngine.pdf, 25.02.2007 17:56

19 http://www.atelierprozess.net/hoog/files/diplomarbeit_hoog.pdf, 25.02.2007 18:00

2. Komponenten der Game-Engine

2.1. Darstellung (Graphics Engine)

Als Grafikengine wird jenes Softwaremodul bezeichnet, welches für die Darstellung der Spielgrafik zuständig ist. Für die Verwendung und Steuerung der Grafikhardware wird meist auf eine Grafikschnittstelle zurückgegriffen, die dem Programmierer viel Arbeit erspart, da er sich nicht mehr direkt um die Grafikhardware kümmern muss. In modernen Grafikengines werden meist OpenGL oder DirectX als Grafikschnittstelle verwendet. Die Grafikengine dient somit als Zwischenstück zwischen Spiel und Grafikschnittstelle.

Heutige Grafikhardware ermöglicht bereits äußerst realitätsnahe Darstellung des Spiels. Demzufolge nehmen Grafikengines in ihrer Komplexität und Entwicklungsdauer zu.

Abb.10: Dieses Bild zeigt eindrucksvoll, wie realitätsnah Spiele auf moderner Hardware dargestellt werden können.
(http://www.sg.hu/kep/2006_06/0610cry.jpg, vom 25. Februar 2007)

2.2. Physik (Physics Engine)

„Um ein Spiel realistisch erscheinen zu lassen ist weiters eine physikalische Simulation notwendig. Indem die Gesetze der Dynamik und Kinematik auf Gegenstände und Charaktere im Spiel angewendet werden, kommen deren Bewegungen den Gegenstücken der realen Welt sehr nah. Es können sogar plausible Bewegungen für fiktive Objekte, wie z.B. Raumschiffe, simuliert werden, die dem Betrachter sofort korrekt erscheinen, obwohl die tatsächliche Vergleichsmöglichkeit fehlt. Folge einer physikalisch basierten Simulation ist oft, dass das Spiel insgesamt realistischer auf den Betrachter wirkt und aufgrund der Erwartungskonformität auch intuitiver zu bedienen ist.“ [20]

„Eng verbunden mit der Physik-Simulation ist das Kollisionsmanagement, welches sich in zwei Bereiche teilt: Die Kollisionserkennung dient dem Prüfen, ob zwei Geometrien innerhalb der Szene kollidieren, wobei gewöhnlich auch die Ermittlung von Kollisionszeit und Kontaktpunkt eine wichtige Rolle spielt. Die Kollisionsbehandlung sorgt für die Ausführung von entsprechenden Aktionen nach einer Kollision. Oft wird dabei die physikalisch basierte Simulation verwendet, um das Verhalten der beteiligten Geometrien nach einem Zusammenstoß zu berechnen. Durch das Kollisionsmanagement wird ein wesentlich höherer Interaktionsgrad möglich, der weit über einen linear vorgegebenen Spielverlauf hinausgehen kann. Der Spieler ist durch eigene Handlungen in der Lage, Reaktionen auszulösen, die sich anschließend autonom und individuell weiterentwickeln. Gegenstände der Szene müssen nicht nur statische Kulisse sein, sondern können aktiv in das Spielgeschehen einbezogen und möglicherweise zur Lösung von Aufgaben verwendet werden. Durch die erhöhte Interaktivität erlaubt das Spiel, den angedachten Verlauf zu verlassen und experimentell die Szene zu erforschen.“ [21]

---

20 http://www.uni-koblenz.de/~cg/Diplomarbeiten/DiplomarbeitFugger.pdf, 14.5 2007, 12:00

21 http://www.atelierprozess.net/hoog/files/diplomarbeit_hoog.pdf 25.2 2007, 18:00

2.3. Audio (Sound Engine)

Eine weitere wichtige Komponente einer Game Engine ist die Ausgabe von Soundeffekten und Hintergrundmusik. Die Sound Engine kümmert sich um die Ausgabe, wobei die Lautstärke einzelner Spuren von ihr bestimmt wird. In modernen Spielen wird auch oft Surround-Sound geboten, um die Athmosphäre des Spiels zu intensivieren. Dabei spielt die Position der Tonquelle und die Umgebung eine wesentlich Rolle. Realitätsnahe Audioausgabe und stimmungsvolle Musik kann den Realismus enorm erhöhen und ist somit nicht zu vernachlässigen.

2.4. Eingabe (Input Engine)

Die Input Engine dient dazu, Benutzereingaben von verschiedensten Eingabegeräten zu verarbeiten. Bei der Entwicklung der Input Engine ist auf ein dynamisches Design zu achten, um dem Spieler möglichst viel Ergonomie und Flexibilität bieten zu können. Der Spieler sollte bis zu einem gewissen Grad selbst bestimmen können, mit welchen Geräten und mit welchen Tasten er Eingaben durchführen möchte.

3. Grafik-Engine

3.1. Vorwort

Es gibt eine Vielzahl fertiger Bibliotheken (zum Beispiel OGRE [http://www.ogre3d.org/]) zur Verwendung als Grafik-Engine. Um möglichst flexibel und unabhängig zu sein, haben wir uns entschieden, eine eigene Grafik-Engine zu entwickeln, wodurch wir umfangreiche Kenntnisse in der Grafikprogrammierung erlangten.

3.2. Wahl der Grafikschnittstelle

Früher wurde bei diversen Spielen (zum Beispiel Doom) die gesamte Darstellung im Rahmen der Game Engine selbst entwickelt. Dadurch erfolgte die gesammte Bildberechnung über die CPU. Dies bot den Vorteil, dass keine Grafikhardware notwendig war. Später etablierte sich der Einsatz von Grafikschnittstellen, welche auf standardisierte Funktionalität der Grafikhardware zugreifen. Hauptvorteil dieser Vorgehensweise ist die Auslagerung der Grafikberechnung auf die Grafikhardware, was die CPU entlastet.

Weiters wird dem Programmierer der Grafik Engine ein erheblicher Teil der Arbeit abgenommen, da auf standardisierte Funktionen der Grafikschnittstelle zugegriffen werden kann.

Zur Zeit gibt es zwei Grafikschnittstellen die sehr weit verbreitet sind, nämlich OpenGL und Direct3D. Im Folgenden wird ein Überblick über diese Schnittstellen gegeben.

3.3. OpenGL

„OpenGL (Open Graphics Library) ist die Spezifikation eines plattform- und programmiersprachenunabhängigen APIs (Application Programming Interface) zur Entwicklung von 3D-Computergrafik. Der OpenGL-Standard beschreibt etwa 250 Befehle welche die Darstellung komplexer Szenen in Echtzeit erlauben. Zudem können andere Organisationen, zumeist Hersteller von Grafikkarten, eigene Erweiterungen definieren.

Die Implementierung des OpenGL-API wird in der Regel als Teil der Grafikkartentreiber ausgeliefert. Diese führen entsprechende Befehle der Grafikkarte aus, insbe-

sondere müssen die Treiber auf der Grafikkarte nicht vorhandene Funktionen mithilfe der CPU emulieren.“ [22]

OpenGL wurde im Jahr 1992 erstmal veröffentlicht. Seitdem sind von verschiedensten Firmen ständig Erweiterungen entwickelt worden. Das Modell der Erweiterungen durch einzelne Firmen sorgt für äußerst rasche Implementierung neuer Funktionen. Allerdings tritt unter der Vielzahl von Entwicklern oft das Problem auf, dass bestimmte Grafikkartentreiber vom Programm geforderte Funktionalität nicht unterstützen.

Zur Zeit wird OpenGL auf den meisten Betriebssystemen unterstützt. OpenGL ist auch auf Handhelds und Mobiltelefonen verfügbar. Außerdem kann OpenGL von nahezu allen Programmiersprachen eingebunden werden.

3.3.1. Direct3D

„Direct3D ist eine Programmierschnittstelle (API) für 3D-Computergrafik und Bestandteil von DirectX. Diese API wurde von Microsoft geschaffen, um Windows-Anwendungen einen möglichst direkten Zugriff auf die Hardware des Computers zu ermöglichen. Häufig verwendet wird Direct3D vor allem für Computerspiele, wo sie mit dem plattform- und betriebssystemunabhängigen OpenGL konkurriert.

[…]

Verschiedene Grafikkarten unterstützen Direct3D durch Gerätetreiber, welche die standardisierten API-Befehle von Direct3D auf die Grafikhardware abbilden. Dabei unterscheidet Direct3D zwischen initialisierenden und ausführenden Befehlen. Initialisierende Befehle konvertieren komplexere Datenstrukturen - wie etwa Texturen - in das Grafikkarten-spezifische Format, ausführende Befehle zeigen die derart konvertierten Elemente an. Da das Initialisieren und Konvertieren von Elementen einige Zeit in Anspruch nehmen kann, ist es bei Spielen üblich, dies während des Ladens

22 Wikipedia - die freie Enzyklopädie. OpenGL. Online im Internet: URL: http://de.wikipedia.org/wiki/Opengl 25.Februar 2007

eines neuen Levels zu tun. Ausführende Befehle sind dagegen auf größtmögliche Geschwindigkeit optimiert.

Direct3D wird offiziell nur auf diversen Microsoft Betriebssystemen unterstützt und bietet lediglich einen kleinen Pool and verwendbaren Programmiersprachen. Da Direct3D ein Teil von DirectX ist steht dem Entwickler dafür ein rießiger Funktionsumfang zur Verfügung. [23]

Es existieren neben OpenGL und Direct3D zwar noch andere Grafikschnittstellen, diese haben jedoch für die Spieleprogrammierung keine hohe Relevanz. Für unsere Grafik-Engine wählten wir OpenGL als Schnittstelle, was uns uneingeschränkte Plattformunabhängigkeit garantiert.

---

23 Wikipedia - die freie Enzyklopädie. Direct 3D. Online im Internet: URL: http://de.wikipedia.org/wiki/Direct3D

Einführung in OpenGL

1. OpenGL Vorwort

Das folgende Kapitel sollte kein OpenGL Skriptum darstellen, sondern sollte nur einen kleinen Einblick in dieses Thema geben, da die OpenGL Programmierung doch einen großen Teil der Diplomarbeit darstellt und grundlegendes Verständnis notwendig ist, um Entscheidungen und Aufbau von Augustus sowie die damit verbundenen Tools zu verstehen.

2. Einführung in primitiven OpenGL Code

Aufgrund der großen Funktionsvielfalt können OpenGL Programme kompliziert werden. Die Grundstruktur ist jedoch meistens recht simpel. Grundsätzlich werden ständig Einstellungen für die folgenden Renderoperationen getroffen und anschließend die Funktionen aufgerufen, welche das Rendern auslösen.

Rendering ist der Prozess, welcher vom Computer durchschritten wird, um Bilder aus geometrischen Modellen zu erzeugen. Solche Modelle oder Objekte können sich aus Punkten, Linien und Polygonen zusammensetzen, welche über ihre Vertices (Eckpunkte) definiert sind.

Das gerenderte Bild besteht aus Pixeln, welche auf einem Bildschirm dargestellt werden. Ein Pixel ist die kleinste Einheit, welche auf dem Bildschirm sichtbar ist. Informationen, wie etwa die Farbe über diesen Pixel werden im dafür vorgesehenen Speicher gehalten. Diesen Speicher nennt man Framebuffer. [24]

Das folgende Beispiel soll den grundsätzlichen Aufbau eines OpenGL Programms aufzeigen. Es rendert ein weißes Quadrat auf schwarzen Hintergrund und stellt diese Elemente

---

24 Dave Shreiner, Mason Woo, Jackie Neider, Tom Davis: OpenGL Programming Guide, 4. Auflage, Addison-Wesley, S 5

dar. Alle nicht unmittelbar am Rendering beteiligten Funktionen sind nur vereinfacht (mittels linguistischen Platzhaltern) dargestellt.

Figure 1-1 : White Rectangle on a Black Background

Example 1-1 : Chunk of OpenGL Code

```
#include <whateverYouNeed.h>

main() {

   InitializeAWindowPlease();

   glClearColor (0.0, 0.0, 0.0, 0.0);
   glClear (GL_COLOR_BUFFER_BIT);
   glColor3f (1.0, 1.0, 1.0);
   glOrtho(0.0, 1.0, 0.0, 1.0, -1.0, 1.0);
   glBegin(GL_POLYGON);
      glVertex3f (0.25, 0.25, 0.0);
      glVertex3f (0.75, 0.25, 0.0);
      glVertex3f (0.75, 0.75, 0.0);
      glVertex3f (0.25, 0.75, 0.0);
   glEnd();
   glFlush();

   UpdateTheWindowAndCheckForEvents();
}
```

Abb.11: Dieses Bild zeigt im unteren Bereich (Example 1-1) einen vereinfachen OpenGL Code, welcher das im oberen Teil (Figure 1-1) dargestelltes Resultat erzeugt.
(Dave Shreiner, Mason Woo, Jackie Neider, Tom Davis: OpenGL Programming Guide, 4. Auflage, Addison-Wesley, S 6)

Die **InitializeAWindowPlease**() Routine ist ein Platzhalter für Betriebssystemspezifische Routinen, welche die Umgebung (OS, Treiber der Grafikhardware) für die Verwendung von OpenGL vorbereiten (klassischerweise ein Fenster erzeugen). Diese Befehle sind in der

Regel keine OpenGL Befehle, sondern Funktionen der Betriebssystem API (auf Windows WinAPI).

Die **glClearColor**() Routine legt fest, in welcher Farbe in Zukunft das Fenster gelöscht wird. Die Angabe der Parameter erfolgt in RGBA (Rot, Grün, Blau, Alpha [Durchsichtigkeit]), wobei der Wertebereich sich auf Fließkomma-Werte zwischen 0 und 1 beschränkt. Die Parameterfolge 1,1,1,1 meint somit Weiß und Undurchsichtig.

Die **glClear**() Routine führt das Löschen des Fensters aus, wobei die Parameter angeben, welche Teile des Framebuffers gelöscht werden. Es wird also in der Regel das gesamte Fenster (alle Pixel des Fensters) auf die in glClearColor angegebene Farbe gesetzt.

Die **glColor3f** Routine funktioniert analog zur glClearColor Routine, bezieht sich allerdings nicht auf das Löschen des Bildschirms, sondern alle Render Befehle die nachfolgend auftauchen. Ruft man also anschließend Funktionen für das Zeichnen von Punkten auf, so werden diese Punkte in dieser Farbe ausgefüllt/schattiert.

Die **glOrtho** Routine legt für das Rendering notwendige Koordinatensystem fest und beschreibt, wie das fertige Bild auf den Bildschirm übertragen (projeziert) wird.

Als nächstes folgt ein Block, welcher von **glBegin(GL_POLYGON)** und **glEnd()** umgeben wird. In diesem Block werden die Eckpunkte (Vertices) der zu zeichnenden Kontur festgelegt wobei der Parameter von glBegin festlegt, wie die nachfolgenden Vertices verbunden bzw. schattiert werden sollen.

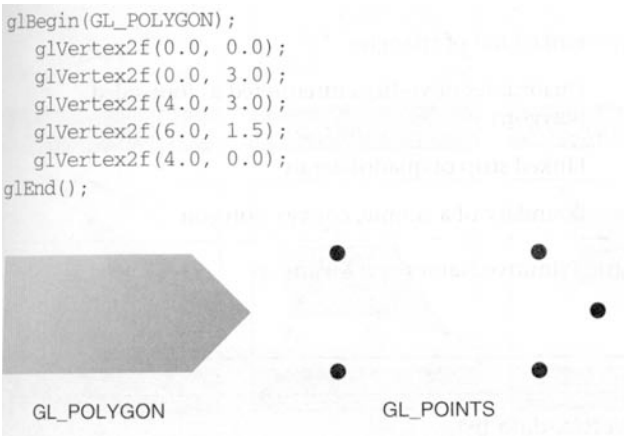


Abb.12: Unterschiede der Zeichenmodi GL_POLYGON und GL_POINTS
(Dave Shreiner, Mason Woo, Jackie Neider, Tom Davis: OpenGL Programming Guide, 4. Auflage, Addison-Wesley, S 43)

Diese Grafik deomstriert die Auswirkung des mode Parameters in glBegin(..). GL_POLYGON bildet also ein Vieleck, welches nach bestimmten Kriterien ausgefüllt (Schattiert) wird.

Die **glFlush** Routine versichert, dass alle bisher aufgerufenen Befehle ausgeführt wurden. Dies läuft darauf zurück, dass OpenGL, die GPU oft asynchron zur CPU steuert. Das heißt die CPU kann schon weitere Aufgaben erledigen, während die GPU noch Geometrie schattiert.

UpdateTheWindowAndCheckForEvents() ist ein Platzhalter für wiederum Betriebssystemspezifische Befehle, welche die errechneten Pixeldaten entgülutig auf das Fenster übertragen und Betriebssystemmeldungen (wie zum Beispiel Tastatur oder Mauseingaben) entgegennehmen und verwalten.

Dieses Beispiel soll keineswegs ein effizientes, gut strukturiertes Programm darstellen, es jedoch trägt wesentlich zum Gundverständnis bei.

OpenGL und die Verwaltung von Einstellungen und Modi

OpenGL wird als state machine bezeichnet, was soviel bedeutet wie: Einstellungen werden getroffen und bleiben bis zur weiteren Änderung genau dieser Einstellungen erhalten. Die derzeitige Farbe ist also eine State Variable. Die Farbe wird einmal gesetzt, und alle weiteren Rendering Calls bedienen sich dieser Farbe.

3. Die OpenGL Pipeline

Die Ausführungsreihenfolgen der in den Renderingprozess angewendeten Funktionen ist in den meisten OpenGL Implementierungen dieselbe.

Vereinfacht ausgedrückt werden Vertex Daten an OpenGL gesendet, diese Daten nach bestimmten Kriterien und Einstellungen bearbeitet und das daraus erstellte Bild wird im Framebuffer abgelegt und steht somit für die weitere Verwendung (Darstellung am Bildschirm) bereitgestellt.

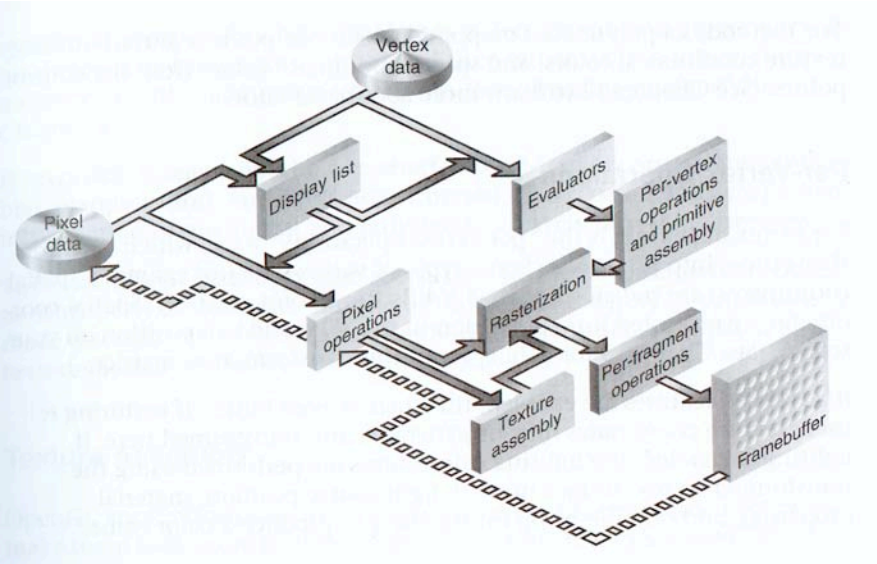


Abb.13: Diese Abbildung zeigt den Schemenhafen Aufbau der OpenGL Pipeline (Dave Shreiner, Mason Woo, Jackie Neider, Tom Davis: OpenGL Programming Guide, 4. Auflage, Addison-Wesley, S 11)

3.1. Display Lists:

Werden Geometriedaten wie im obigen Beispiel direkt mittels glBegin()/glEnd() Block an OpenGL gesendet, so spricht man hier oft von immediate mode. Vertex Daten werden also Vertex für Vertex an OpenGL gesendet. Dies hat einen großen Nachteil hinsichtlich der Rendering Effizienz.

Heutige Grafikhardware ist darauf ausgelegt, unabhängig von anderer Hardware (CPU,RAM) Berechnungen auszuführen. Die GPU ist in vielen Fällen daher asyncron zur CPU. In der „immediate mode“ kann von dieser Optimierung keinerlei Vorteil erzielt werden, da ständig CPU und GPU abwechselnd Daten verarbeiten müssen. Das heißt OpenGL versucht das Bild zu erstellen, für die weitere Berechnung werden allerdings wieder neue Daten benötigt, die von glVertex zur Verfügung gestellt werden.

Um diesem Problem entgegen zu wirken, wurde der Display List Mechanismus entwickelt, welcher es ermöglicht große Datenansammlungen im OpenGL nahen Speicher (meistens Grafikspeicher) abzulegen. Wird also eine Display List ausge-

führt kann das gesamte Objekt, welches in dieser Display list abgelegt ist, ohne weiterer CPU Interaktion gerendert werden.

3.2. Per Vertex Operations

Diese Ebene umfasst alle Operationen, welche pro Vertex durchgeführt werden. Sie beinhaltet die Transformation und Projektion einzelner Vertices. Für erweiterte Rendering Konzepte beinhaltet diese Ebene auch Operationen, die sich mit Lichtberechnung, Nebelberechnung und Vorbereitung für die Verwendung von Texturen auseinander setzen.

3.3. Primitive Assembly

Diese Ebene stellt auch die Verbindung zwischen den einzelnen Vertices her und bildet sogennante Primitive, wobei diese den Clipping-Prozess durchlaufen. Clipping beschreibt die Aufgabe, Bereiche der Geometrie vom weiteren Renderingprozess auszuschließen.

Auf dieser Ebene findet bei Polygonen auch das Culling statt, welches im Allgemeinen unerwünschte Seiten der Primitiven vom Renderingprozess ausschließt.

Ausgang bzw. Ergebnis dieser Ebene sind somit transformierte, geclippte Primitive, welche alle Informationen (Positionen, Texturkoordinaten, Normalvektoren) für die nächste Ebene, die Rasterisierung beinhalten.

3.4. Rasterization

Die Rasterisierung beschäftigt sich mit allen Operationen, welche sich mit der „Umwandlung“ von Geometrischen Daten in Pixeldaten beschäftigen. Hier werden die Primitiven nach verschiedenen Gesichtspunkten in Pixel umgewandelt. Ausgang dieser Ebene sind so genannte Fragmente, wobei jedes Fragment eine zum jeweiligen Pixel korrespondierende Farbe und einen Tiefenwert hat [25]. (Tiefenwert siehe 7. Tiefenbuffer)

---

25 Dave Shreiner, Mason Woo, Jackie Neider, Tom Davis: OpenGL Programming Guide, 4. Auflage, Addison-Wesley,

3.5. Fragment Operations

Vor die von der Rasterization-Ebene berechneten Fragmente im Framebuffer gespeichert werden, werden noch einzelne Operationen pro Fragment ausgeführt.

Diese Operationen sind optional und können demnach aktiviert und deaktiviert werden. Einige dieser Operationen sind Alpha Test, Depth Test, Blending.

4. Primitive

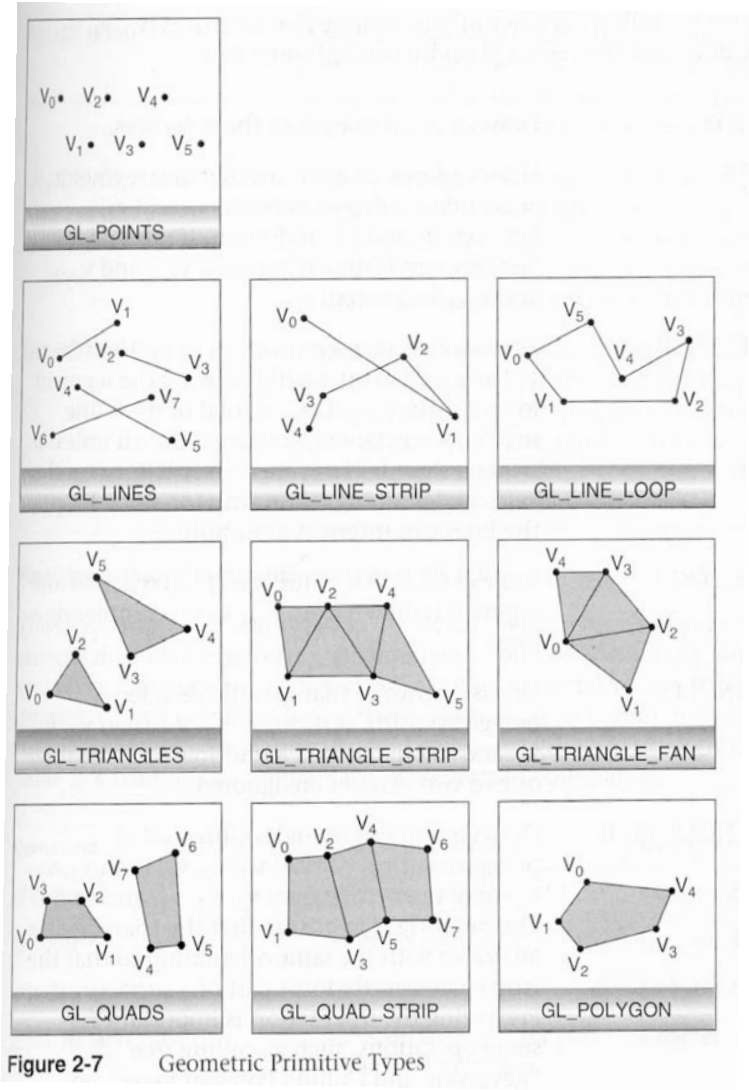


Figure 2-7 Geometric Primitive Types

Abb.14: Dieses Bild zeit alle in OpenGL verfügbaren Zeichenmodi und ihre Auswirkungen auf die Verknüpfung der Vertices
(Dave Shreiner, Mason Woo, Jackie Neider, Tom Davis: OpenGL Programming Guide, 4. Auflage, Addison-Wesley, S 45)

Da in vielen Applikationen häufig Dreiecke verwendet werden, ist die Darstellung von GL_TRIANGLES auf moderner Hardware extrem optimiert. Daher ist es auch nahe liegend diese Optimierung möglichst auszunutzen und Geometrie in Form von Dreiecken an OpenGL zu senden.

5. Viewing

In der 3D Computergrafik wird für die Ausrichtung, Translation und Rotation von Geometrie meistens auf die Matrizenrechnung zurückgegriffen. Das Mathematische Grundverständnis der Matrizenrechnung wird für das Folgende vorausgesetzt. Hierfür verweisen wir auf „Dave Shreiner, Mason Woo, Jackie Neider, Tom Davis: OpenGL Programming Guide, 4. Auflage, Addison-Wesley, Seite 697ff, Kapitel Homogeneous Coordinates and Transformation Matrices“.

Sämtliche OpenGL Transformations setzen sich aus einer 3x3 Rotationsmatrix und einem Translationsvektor 4x1 zusammen. In Kombination ergibt sich so eine 4x4 Matrix. In OpenGL gehen Vertexkoordinaten durch mehrere Koordinatensysteme, bis sie schlussendlich in Fenster-Koordinaten transformiert werden.

Das folgende Bild zeigt alle involvierten Matrizen und die Koordinatensysteme (in OpenGL oft engl. als – space bezeichnet) in welchen sich die Koordinaten nach der jeweiligen Transformation befinden.

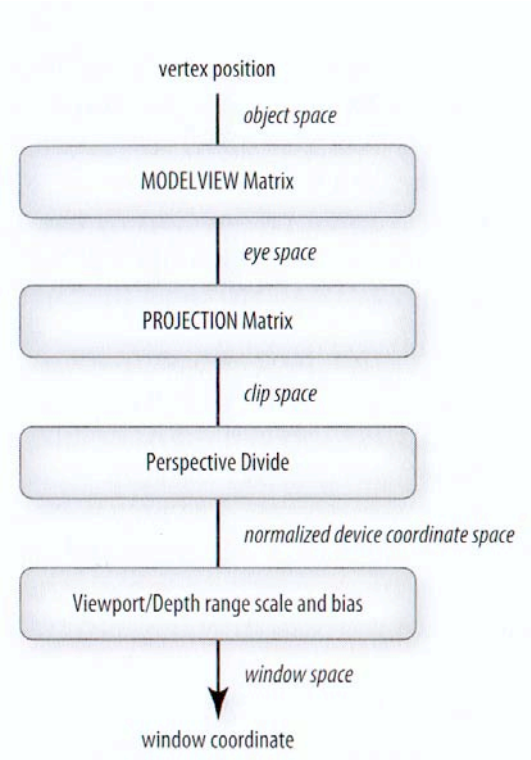


Abb.15: Transformationsablauf in OpenGL

(Randi J. Rost: OpenGL Shading Languate, 2. Auflage, Addison-Wesley, 2006, S 24)

Auch Matrizen verhalten sich in OpenGL wie State Variablen. Das heißt, der einmalig gesetzte Wert bleibt, bis er überschrieben wird. Da für jede Matrix die gleichen Manipulationsfunktionen einsetzbar sein sollen, ist die zu behandelnde Matrix vorher auszuwählen. Diese Auswahl wird über glMatrixMode(..) getroffen wobei der Parameter folgende Werte haben kann:

- **GL_MODELVIEW**
  die Modelview Matrix wird als derzeitige Matrix gewählt.
- **GL_PROJECTION**
  die Modelview Matrix wird als derzeitige Matrix gewählt.
- **GL_TEXTURE**
  Die Texturmatrix wird als derzeitige Matrix ausgwählt. Diese Matrix bezieht sich auf Texturkoordinaten, welche aber im Texturen Kapitel behandelt werden.

Folgende Funktionen erlauben das Manipulieren der jeweiligen Matrix:

- **glLoadIdentity()**
  lädt die Identitätsmatrix

- **glTranslatef(x,y,z)**
  Multipliziert die derzeitige Matrix mit einer Matrix, welche ein Objekt um die angegebenen (x,y,z) Werte verschiebt

- **glRotatef(winkel,x,y,z)**
  Multipliziert die derzeitige Matrix mit einer Matrix, welche ein Objekt um den angegebenen Winkel im Uhrzeigersinn um den Vektor vom Ursprung durch (x,y,z) dreht.

- **glPushMatrix()**
  schreibt den derzeitigen Zustand der derzeitigen Matrix auf einen Stack. Alle folgenden Manipulationen finden auf dieser Ebene des Stacks statt und verändern die darunterliegenden Ebenen des Stacks nicht.

- **glPopMatrix()**
  nimmt die höchste auf dem Stack liegene Matrix vom Stack. Die zwischen glPushMatrix und glPopMatrix erzeugten Manipulationen gehen somit verloren.

Weitere Funktionen Routinen sind gluLookAt, gluPerspective, glFrustum, glOrtho. Für tiefer gehende Informationen verweisen wir auf „Dave Shreiner, Mason Woo, Jackie Neider, Tom Davis: OpenGL Programming Guide, 4. Auflage, Addison-Wesley, Chapter 3, S97ff “

6. Tiefenbuffer

Die bisherigen beschriebenen Features erlauben das Rendern von verschiedensten Primitiven, allerdings nicht das korrekte darstellen von 3D Welten.

Bisher überschreibt jeder aus Primitiven erzeugte Pixel, den entsprechenden Pixel im Framebuffer.

Das Ergebnis ist somit erheblich von der Reihenfolge der Render Aufrufe abhängig.

Würde man also einen Würfel auf den Weltursrpung zeichnen, und anschließend einen Würfel welcher sich aus Sicht der Kamera weiter weg befindet, würde der zuletzt gezeichnete Würfel trotzdem den ersten überdecken, obwohl dieser eigentlich vom zuerst gezeichneten überdeckt würde.

Um diesem Verhalten entgegenzuwirken, wurde dem Framebuffer ein zusätzlicher Buffer angefügt, welcher als Tiefenbuffer (Depthbuffer) bezeichnet wird. In diesem Buffer wird pro Pixel des Framebuffers ein weiterer Wert abgelegt. Dieser Wert wird als Depth Value bezeichnet und wird im Rasterisierungslayer der OpenGL pipeline pro Pixel berechnet. Er gibt die Entfernung vom Viewport zum Pixel an.

Im Fragment Operation Layer wird pro Fragment der Vergleich zwischen dem im Tiefenbuffer vorliegenden Wert und dem Wert des erzeugten Fragments durchgeführt. Stellt der Wert des neuen Pixels eine kürzere Entferunung zum Viewport dar, so wird im Framebuffer der Farb- und Tiefenwert überschrieben. Trifft diese Bedingung nicht zu, so wird dieses Fragment verworfen.

7. Licht und Materialien

> *When you look at a physical surface, your eye's perception of the color depends on the distribution of photon energies that arrive and trigger your cone cells. (See "Color Perception" in Chapter 4.) Those photons come from a light source or combination of sources, some of which are absorbed and some of which are reflected by the surface. In addition, different surfaces may have very different properties - some are shiny and preferentially reflect light in certain directions, while others scatter incoming light equally in all directions. Most surfaces are somewhere in between. OpenGL approximates light and lighting as if light can be broken into red, green, and blue components. Thus, the color of light sources is characterized by the amount of red, green, and blue light they emit, and the material of surfaces is characterized by the percentage of the incoming red, green, and blue components that is reflected in various directions. The OpenGL lighting equations are just an approximation but one that works fairly well and can be computed relatively quickly. If you desire a more accurate (or just different) lighting model, you have to do your own calculations in*

software. Such software can be enormously complex, as a few hours of reading any optics textbook should convince you. In the OpenGL lighting model, the light in a scene comes from several light sources that can be individually turned on and off. Some light comes from a particular direction or position, and some light is generally scattered about the scene. For example, when you turn on a light bulb in a room, most of the light comes from the bulb, but some light comes after bouncing off one, two, three, or more walls. This bounced light (called ambient) is assumed to be so scattered that there is no way to tell its original direction, but it disappears if a particular light source is turned off. Finally, there might be a general ambient light in the scene that comes from no particular source, as if it had been scattered so many times that its original source is impossible to determine. [26]

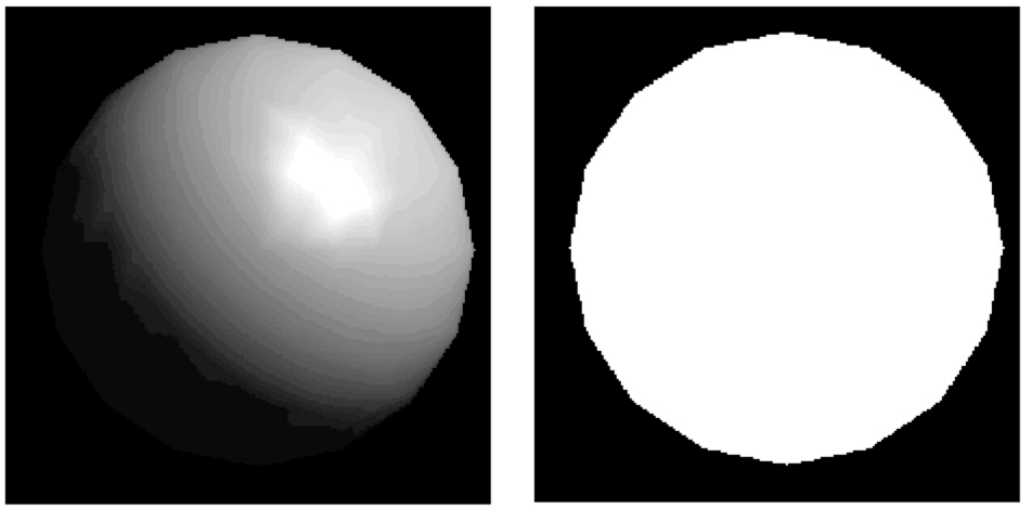


Figure 5-1 : A Lit and an Unlit Sphere

Abb.16: Dieses Bild zeigt den Vergleich einer belichteten und Unbelichteten Kugel (Dave Shreiner, Mason Woo, Jackie Neider, Tom Davis: OpenGL Programming Guide, 4. Auflage, Addison-Wesley, S 178)

Das obige Bild zeigt den Vergleich zweier Kugeln, wobei an der linken Lichtberechnung durchgeführt wurde, und an der anderen, der Rechten nur die Farbe „weiß“ benutzt wurde.

---

26 Dave Shreiner, Mason Woo, Jackie Neider, Tom Davis: OpenGL Programming Guide, 1. Auflage, Addison-Wesley, S 129

Die rechte Kugel erscheint dem Betrachter nicht einmal 3D – vielmehr sieht sie aus wie ein Kreis.

Um eine realistisch wirkende 3D Szene zu erstellen ist es in den meisten Fällen unumgänglich irgendeine Art von Lichtberechnung mit einzubeziehen.

Das in OpenGL verwendete Lichtberechnungsmodell teilt die Lichtberechnung in 4 voneinander unabhängige Komponenten.

> *Ambient illumination is light that's been scattered so much by the environment that its direction is impossible to determine - it seems to come from all directions. Backlighting in a room has a large ambient component, since most of the light that reaches your eye has first bounced off many surfaces. A spotlight outdoors has a tiny ambient component; most of the light travels in the same direction, and since you're outdoors, very little of the light reaches your eye after bouncing off other objects. When ambient light strikes a surface, it's scattered equally in all directions.*
>
> *The diffuse component is the light that comes from one direction, so it's brighter if it comes squarely down on a surface than if it barely glances off the surface. Once it hits a surface, however, it's scattered equally in all directions, so it appears equally bright, no matter where the eye is located. Any light coming from a particular position or direction probably has a diffuse component.*
>
> *Finally, specular light comes from a particular direction, and it tends to bounce off the surface in a preferred direction. A well-collimated laser beam bouncing off a high-quality mirror produces almost 100 percent specular reflection. Shiny metal or plastic has a high specular component, and chalk or carpet has almost none. You can think of specularity as shininess.*[27]
>
> *In addition to ambient, diffuse and specular colors, materials may have an emissive color, which simulates ligh originating from an object. In the OpenGL lighting model, the emissive color of a surface adds intencity to the object, but is unaffected by any*

---

27 Dave Shreiner, Mason Woo, Jackie Neider, Tom Davis: OpenGL Programming Guide, 1. Auflage, Addison-Wesley, S 130

light sources. Also emissive color does not introduce any additional light into the overall scene. [28]

Although a light source delivers a single distribution of frequencies, the ambient, diffuse, and specular components might be different. For example, if you have a white light in a room with red walls, the scattered light tends to be red, although the light directly striking objects is white. OpenGL allows you to set the red, green, and blue values for each component of light independently.[29]

Materialien werden im OpenGL Lichtberechnungsmodell einbezogen, indem für jede Kompontente (ambient, diffuse, specular, emissive) eine Materialfarbe verwendet wird, welche angibt, wie sehr die jeweilige Farbkomponente reflektiert wird.

Für mathematische Hintergründe dieser Berechnungen verweisen wir auf: „Dave Shreiner, Mason Woo, Jackie Neider, Tom Davis: OpenGL Programming Guide, 4. Auflage, Addison-Wesley, Seite 215ff“.

8. Texturen

Theoretisch ist ein perfektes Lichtberechnungsmodell mit dazugehörenden Materialdefinitionen ausreichend, photorealistische Bilder zu rendern. Am Beispiel einer Wand, die realistisch gerendert werden soll, müssten alle einzelnen Ziegelsteine bis ins kleinste Detail ausgearbeitet/ausmodelliert sein, als 3D Modell vorliegen sowie perfekte (realitätsnahe) Lichtberechnung angewandt werden, um die Wand realistisch erscheinen zu lassen.[30]

Dieser Ansatz ist natürlich nicht sinnvoll, da ein gewisser Teil der Farbgebung schon vorher berechnet werden kann. Dieser Mechanismus nennt sich Texturierung. Dabei werden

---

28 Dave Shreiner, Mason Woo, Jackie Neider, Tom Davis: OpenGL Programming Guide, 4. Auflage, Addison-Wesley, S 182f

29 Dave Shreiner, Mason Woo, Jackie Neider, Tom Davis: OpenGL Programming Guide, 1. Auflage, Addison-Wesley, S 130

30 Dave Shreiner, Mason Woo, Jackie Neider, Tom Davis: OpenGL Programming Guide, 4. Auflage, Addison-Wesley, S 360

Bilder, welche in Pixeldaten vorliegen mit Beachtung gewisser Kriterien über das 3D Modell gelegt. Diesen Vorgang nennt man „Texture mapping“.

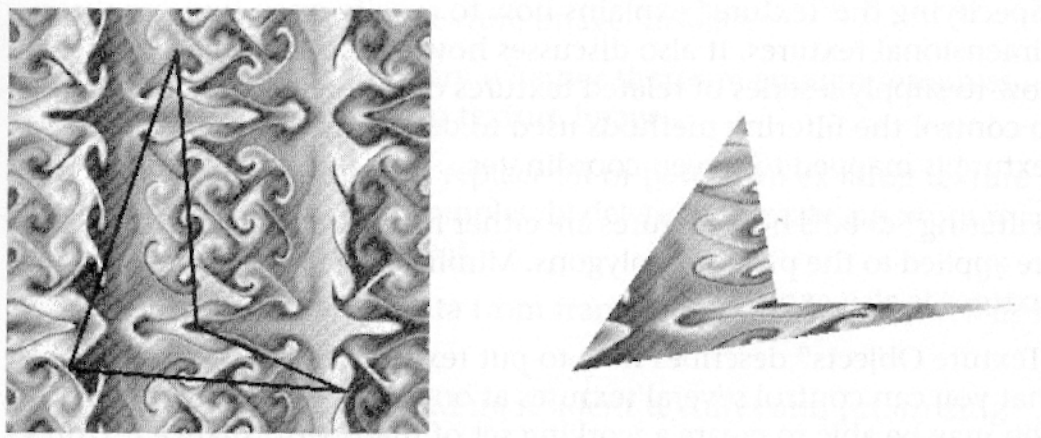


Figure 9-1 Texture-Mapping Process

Abb.17: Diese Abbildung zeigt den Texture-Mapping Prozess angewandt auf ein Vieleck (Dave Shreiner, Mason Woo, Jackie Neider, Tom Davis: OpenGL Programming Guide, 4. Auflage, Addison-Wesley, S 361)

Im Allgemeinen sind Texturen einfache Arrays aus Daten, zum Beispiel Farbwerten. Einzelne Werte solcher Arrays werden oft als Texel bezeichnet.

Die oben gezeigte Abbildung zeigt den Texture mapping Prozess. Die linke Seite des Bildes zeigt die gesamte Textur und die rechte Seite des Bildes zeigt ein Vieleck, auf welches diese Textur „aufgebracht“ wird. Im Grafik Teil in Kapitel 2.5 wird auf dieses Verfahren näher eingegangen.

Um Texturen in OpenGL zu verwenden sind folgende Schritte notwendig:

1. Erzeugen eines OpenGL Textur Objektes
2. Informationen, wie die Textur auf jeden Pixel angewendet wird festlegen. Diese Informationen sollte klären ob Texel dieser Textur mit anderen Pixeln multipliziert werden sollen, oder ob die Texel der Textur andere Pixel ersetzten.
3. Einschalten des OpenGL Textur Modus

4. Zeichnen der Szene, wobei Texturkoordinaten und geometrische Koordinaten für die Berechnung herangezogen werden

Texturen werden von OpenGL im sogenannten Textur Speicher abgelegt. Dies ermöglicht einen effizienten Zugriff auf diese Daten, da der Grafikkarten Treiber für OpenGL immer dafür sorgt, dass diese Daten im schnellen Grafikspeicher vorliegen.

Den Vorgang Daten im OpenGL internen Speicher jeglicher Art abzulegen, nennt man das hochladen (engl. Upload) von Resourcen.

```
GLuint texName;                                          // Variable, um Textur anzusprechen
glGenTextures(1, &texName);                              // Erzeugt OpenGL Texutr Objekt


glBindTexture(GL_TEXTURE_2D, texName);                   // Auswählen der Textur texName


// Folgende Funktionen bestimmen, wie sich die Textur hinsichtlich Vergrößerung/Verkleinerung be-
nimmt bzw. wie sich die Textur über größere Objekte als die Textur ist reproduziert

glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
                    GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
                    GL_NEAREST);


// Diese Funktion lädt Pixel daten, welche im Arbeitspeicher vorliegen (checkImage zeigt auf diese
Daten) in den Texturspeicher hoch. Dafür wird Typ der Daten, sowie dimensionen der Daten angegeben

glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, checkImageWidth,
                 checkImageHeight, 0, GL_RGBA, GL_UNSIGNED_BYTE,
                 checkImage);
```

Die Verwendung der hochgeladenen Textur sieht wie folgt aus:

```
// Diese Funktion legt fest, wie Pixel der Textur auf das derzeitge Fragment wirken. In diesem Fall
(GL_REPLACE) wird das Fragment einfach überschrieben.

glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_REPLACE);

// Auswählen der zu verwendenden Textur
glBindTexture(GL_TEXTURE_2D, texName);

// Zeichnen der Geometrie
   glBegin(GL_QUADS);
      glTexCoord2f(0.0, 0.0); glVertex3f(-2.0, -1.0, 0.0);
      glTexCoord2f(0.0, 1.0); glVertex3f(-2.0, 1.0, 0.0);
      glTexCoord2f(1.0, 1.0); glVertex3f(0.0, 1.0, 0.0);
      glTexCoord2f(1.0, 0.0); glVertex3f(0.0, -1.0, 0.0);
   glEnd();
```

Mit der Funktion glTexCoord2f werden die zu verwendenden Texturkoordinaten (sieh Kapitel 2.5 im Grafik Teil) für den jeweiligen Vertex festgelegt.

Im obigen Beispiel wurde eine 2 Dimensionale Textur verwendet. Das heißt das Texturbild bestand aus einem 2 Dimensionalen Array, welches Texel enthielt.

In OpenGL ist allerdings auch die Verwendung von 1D und 3D Texturen möglich.

Falls die Texturkoordinaten nicht bekannt sind, sich allerdings leicht berechnen lassen (mittels Projektion einer Textur), ist die Angabe der Texturkoordinaten nicht notwendig, allerdings Angaben, wie sich diese Texturkoordinaten berechnen. Diese Berechnung findet dann pro Vertex statt und bildet somit die Texturkoordinaten.

In OpenGL besteht auch die Möglichkeit, mehrere Texturen gleichzeitig auf ein Fragment der gleichen Geometrie anzuwenden. Dieses Verfahren nennt sich Multitexturing und wird vor allem bei Spezialeffekten besonders interessant. Die Zahl der auf ein Fragment anwendbaren verschiedenen Texturen variiert von Grafikhardware zu Grafikhardware. Oft handelt es sich hier um 4 verschiedene Texturen. Die einzelnen Texturen beim Multitexturinng werden oft auch Texturlayer genannt.

5. Shader

Die in den vorigen Kapiteln behandelten Verfahren waren bis vor wenigen Jahren fix in den Treiber implementiert. Der Entwickler hatte nur über Schnittstellen, die die Grafikschnittstelle bot, Möglichkeit das jeweilige Resultat (also Resultat der Lichtberechnung, des Multitexturing,...) zu beeinflussen. Vor allem bei Multitexturing war dies relativ schwierig, da OpenGL und DirectX nie ausreichende Flexibilität hinsichtlich der Kombination von Texturen bot. Weiters bot diese fest in die Grafikkartentreiber implementierte Funktionaliät nur die Möglichkeit, die Lichtberechnung pro Vertex durchzuführen, wodurch der Realitätsgrad erheblich gesenkt wurde, da für gute Lichtberechnung hohe Vertex bzw. Polygonzahlen notwendig waren, dies aber von der Grafikhardware hinsichtlich der Performance nicht verantwortbar war.

In vielen Computergrafikbranchen war es schon früher möglich solche Funktionalität über kleine Programme selbst zu definieren. Bei Grafikkarten wie der GeforceFX und der ATI

Radeon 9500 war es erstmals möglich Per Vertex Operationen und Per Fragment Operationen selbst zu programmieren und dadurch die fixe Funktionalität durch eigene Programmierwünsche zu ersetzen.

Diese kleinen Programme nannte man Shader Programme. Zu Beginn wurden sie in eigenen Assembler Sprachen entwickelt, vom Grafikkartentreiber kompiliert und ersetzten somit die Fixed Pipeline des Treibers.

Mit OpenGL 2.0 (bzw. der SHADING_LANGUAGE extension) wurde erstmals in der OpenGL Serie die Möglichkeit geboten, diese kleinen Programme über eine High Level Sprache zu programmieren.

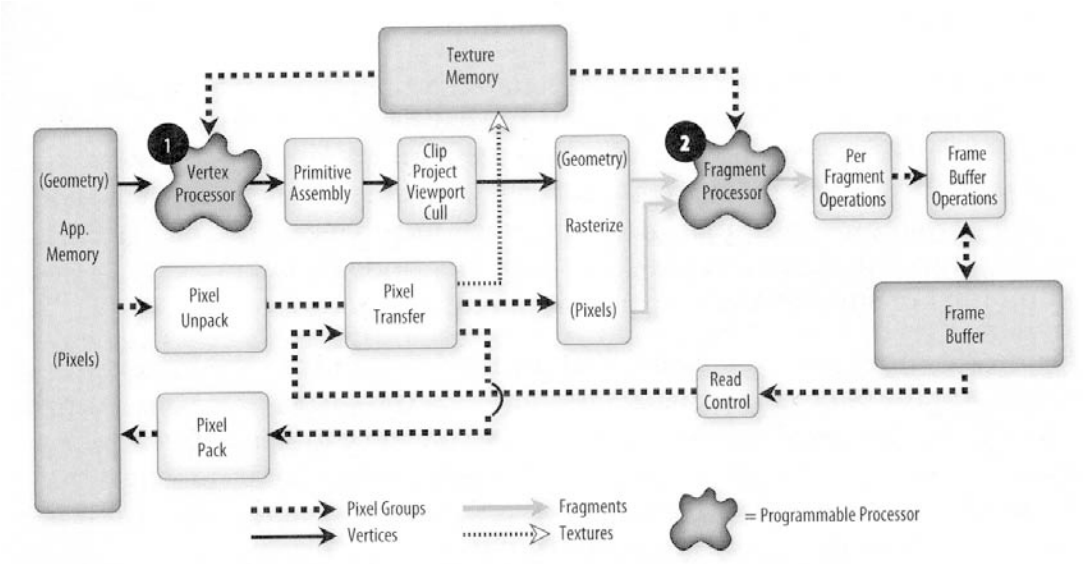


Abb.18: Diese Abbildung zeigt die OpenGL-Pipeline, wobei Vertex- und Pixelsprozessor hervorgehoben sind

(Randi J. Rost: OpenGL Shading Languate, 2. Auflage, Addison-Wesley, 2006, S 39)

Die Einheit der Per Vertex Operationen, hier Vertex Processor genannt konnte vom Programmierer selbst bestimmt werden. Dabei ist es nicht möglich Fixed Pipeline Funktionalität zu erweitern, sondern es ist immer notwendig die gesamte Einheit neu zu programmieren.

Aufgaben des Vertex Processors sind:

- Vertex Transformation
- Normal transformation und Normalisierung
- Textur Koordinatengenerierung bzw. Transformation
- Lichtberechnung
- Behandlung der Materialien

Die Einheit der Per Fragment Operationen, hier Fragment Processor genannt hat folgende Aufgaben, wobei sie sich immer auf einzelne Fragmente beziehen. In der Fixed Pipeline hat der Fragment Processor folgende Aufgaben:

- Texturzugriffe für Texturierung einzelner Fragmente
- Nebel

Fuzzy Logic Allgemein

1. Einführung:

Fuzzy Logic ist eine geläufige Methode der künstlichen Intelligenz. Die Theorie wurde entwickelt, um eine Art menschliche Logik zu imitieren. Im Gegensatz zur bool'schen Logik arbeitet Fuzzy Logic nicht mit scharfen Grenzen zwischen Zuständen. Sie ermöglicht Computersystemen mit „menschlichen" Begriffen (wenig, mittel, viel usw.) zu arbeiten. Dabei werden exakte Werte unscharfen Zuständen zugeordnet (Fuzzyfizierung). Diese Zustände werden mit Hilfe eines einfachen Regelwerkes, dessen Struktur im Prinzip der eines bool'schen Regelwerkes entspricht, verknüpft und auf Ausgänge geführt. Um wieder für ein Computersystem „brauchbare" Werte zu erhalten werden aus den unscharfen Zuständen wieder konkrete (scharfe) Werte errechnet (Defuzzyfizierung).

Funktionsschema eines Fuzzy Reglers:

Abb.19: Funktionsschema eines Fuzzy Reglers

Eigene Begriffe, die im Zusammenhang mit Fuzzy Logic wichtig sind:

Linguistische Variable:

Eine Variable (z.B. Geschwindigkeit, ...) , welche unscharfe Werte annehmen kann. (z.B. schnell , viel , stark, ...). Diese Zustände werden durch Fuzzy-Sets repräsentiert

Ein Beispiel:

Hier im Beispiel die Repräsentation der linguistischen Variable Bremskraft:

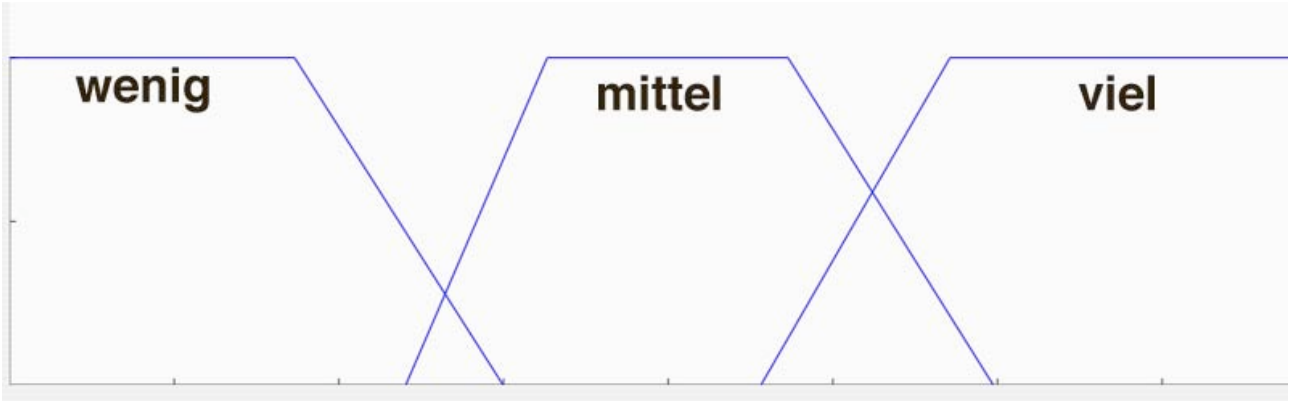


Abb.20: Eine linguistische Variable

X - Achse ... scharfe Eingangswerte [beliebig skaliert (je nach Art der Größe)]

Y - Achse ... Zugehörigkeit zum jeweiligen Zustand [0 ... 1]

wenig, mittel und viel weden Fuzzy-Sets oder Zustände genannt

Die Funktionskurven von wenig, mittel und viel werden Zugehörigkeitsfunktionen genannt

Fuzzy-Set:

Ein Fuzzy-Set repräsentiert einen linguistischen Zustand (z.B.: schnell, mittel, wenig) und erzeugt zu jedem Eingangswert eine Zugehörigkeit (Membership μ). Man spricht von einer Zugehörigkeitsfunktion. Für die Form der Zugehörigkeitsfunktionen gibt es prinzipiell keine Vorgaben, es haben sich jedoch einfache Formen wie Trapeze und Dreiecke als sinnvoll erwiesen, da komplexere Formen einen wesentlich höheren Rechenaufwand mit sich bringen und nicht zwingend eine Verbesserung des Systems erzielen.

2. Fuzzyfizierung

2.1. Einführung

Fuzzyfizierung bezeichnet den Vorgang, bei dem die exakten Eingangswerte des Systems unscharfen Zuständen zugeordnet werden, wobei zu jedem dieser Zustände eine Zugehörigkeit errechnet wird. Im Detail bedeutet dies das man scharfe Werte zu einem oder mehreren Fuzzy-Sets einer linguistischen Variable zuordnet.

2.2. Funktionsweise

Um eine Fuzzyfizierung zu ermöglichen ist es erforderlich Fuzzy-Sets und deren Zugehörigkeitsfunktion zu definieren, welche die zu verwendende linguistische Variable beschreiben. Durch einsetzen eines konkreten Wertes in die Zugehörigkeitsfunktionen aller, zur linguistischen Variable gehörenden, Fuzzy-Sets werden Zugehörigkeiten zu den einzelnen Fuzzy-Sets errechnet.

Beispiel:

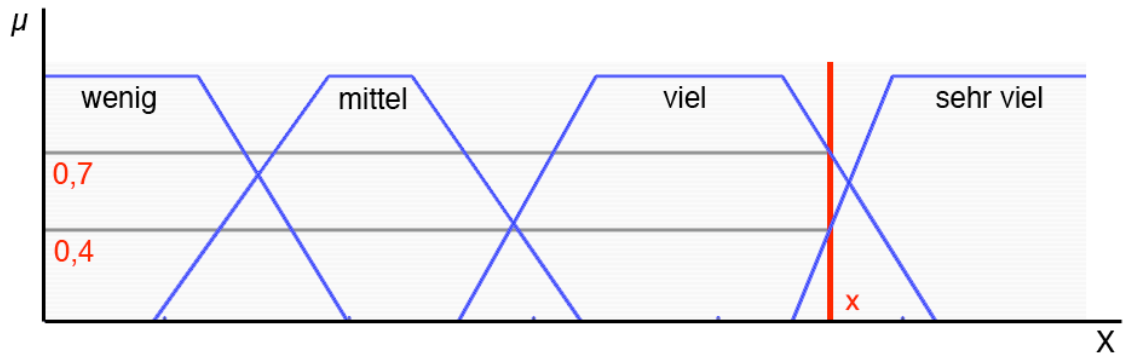


Abb.21: Beispiel Funtionsweise

Wie im Beispiel ersichtlich werden dem scharfen (exakten) Wert x zwei Zugehörigkeiten zugeordnet. Man spricht hier davon, dass der Wert x zu 70% viel und zu 40% sehr viel ist. Diese Zugehörigkeiten geben an in wie fern ein Wert zu den entsprechenden Fuzzy-Sets „gehört“. Im Beispiel werden keine Zugehörigkeiten zu den Fuzzy-Sets wenig und mittel angegeben, da diese 0 sind und für die weitere Berechnung daher irrelevant. Die Anzahl der Fuzzy-Sets, die einem scharfen Wert zu-

geordnet werden ist prinzipiell nicht begrenzt. Das bedeutet, dass auch mehr als zwei Zugehörigkeiten vorhanden sein dürfen.

3. Regelwerk

3.1. Einführung

Das Regelwerk verarbeitet die fuzzyfizierten Daten mit Hilfe von Regeln und weist Ausgangsvariablen die entsprechenden Zustände zu. Die Ausgangsvariablen werden wie Eingangsvariablen durch Fuzzy-Sets und deren Zugehörigkeitsfunktion repräsentiert. Hier liegt der gravierende Vorteil eines Fuzzy-Reglers gegenüber einem mathematischen Modell. Diese Regeln ähneln sehr stark menschlichen Formulierungen, da sie mit Begriffen wie **viel**, **wenig**, usw. arbeiten. Bei diversen Regelungsaufgaben ermöglicht dies eine Erstellung der Regeln von einem Fachmann, der für eine mathematische Lösung des Problems über unzureichende Kenntnisse verfügen würde.

3.2. Funktionsweise:

Fuzzy-Regeln arbeiten mit logischen Operatoren wie **AND**, **OR**, **NOT**, **IS**, etc. und verknüpfen damit die fuzzyfizierten Eingangswerte.

Um die Funktionsweise zu verdeutlichen bedienen wir uns eines Beispiels, da es eine einfache Veranschaulichung des Systems ermöglicht.

Beispiel:

Es gilt die **Bremskraft** eines PKWs in Abhängigkeit von **Geschwindigkeit** und **Abstand zum Vordermann** zu berechnen.

Das Regelwerk erhält als Eingang folgende Werte:

Geschwindigkeit:

70% schnell
40% mittel

Abstand zum Vordermann:

30% wenig
80% mittel

Es gilt nun diese Zustände mit Hilfe von Regeln zu verknüpfen um auf eine angemessene Bremskraft zu schließen.

Dazu verwenden wir folgende Regeln:

IF (**Geschwindigkeit** IS schnell) AND [(**Abstand** IS wenig) OR (Abstand IS mittel)]
THEN **Bremskraft** IS hoch

IF (**Geschwindigkeit** IS schnell) AND (**Abstand** IS hoch) THEN **Bremskraft** IS gering

IF (**Geschwindigkeit** IS mittel) AND (**Abstand** IS wenig) THEN **Bremskraft** IS mittel

IF (**Geschwindigkeit** IS mittel) AND [(**Abstand** IS hoch) OR (**Abstand** IS mittel)]
THEN **Bremskraft** IS gering

IF (**Geschwindigkeit** IS langsam) THEN **Bremskraft** IS gering

Im folgenden setzen wir die Fuzzy-Werte der einzelnen Eingänge ein. Dabei ist zu beachten, dass alle Regeln mit allen möglichen Zustandskombinationen geprüft werden müssen. Die Zugehörigkeiten zu den einzelnen Zuständen werden folgendermaßen verarbeitet:

Bei **AND**-Verknüpfungen wird das Minimum der Zugehörigkeiten gebildet und als Zugehörigkeit für den Ausgangszustand verwendet:

IF (**Geschwindigkeit** IS mittel) AND (**Abstand** IS wenig) THEN **Bremskraft** IS mittel

Geschwindigkeit ist zu **40% mittel** und **Abstand** ist zu **30% wenig**. Daraus folgt, nach der oben stehenden Regel, dass die **Bremskraft** zu **30% mittel** ist.

Bei **OR**-Verknüpfungen wird hingegen das Maximum der Zugehörigkeiten gebildet:

IF (**Geschwindigkeit** IS schnell) AND [(**Abstand** IS wenig) OR (Abstand IS mittel)]

THEN **Bremskraft** IS hoch

Hier stoßen wir auf eine Hintereinanderausführung von **AND** und **OR**, wobei hier zuerst das Maximum der veroderten und anschließend das Minimum der verundeten Prüfung gebildet wird.

Der **Abstand** ist zu **30% wenig** und zu **80% mittel**. Die **Geschwindigkeit** ist zu **70% schnell**. Hier eine Darstellung des Problems:

70% AND (**30%** OR **80%**) -> **min**(0.7 , **max**(0.3 , 0.8)) = **0.7**

Demzufolge ist die **Bremskraft** zu **70% hoch**.

Nachdem alle Regeln angewandt wurden erhalten wir folgenden Ergebnis:

Bremskraft:

40% gering
30% mittel
70% hoch

Um von diesen Fuzzy-Werten auf Werte zu kommen mit Hilfe derer beispielsweise ein hydraulisches Bremssystem arbeiten kann bedienen wir uns der Defuzzyfizierung.

4. Defuzzyfizierung:

4.1. Einführung

Um die unscharfen Ausgänge des Regelwerks wieder in konkrete Zahlenwerte zu übersetzen bedienen wir uns der Defuzzyfizierung. Hierzu ist es erforderlich zu den jeweiligen Ausgangs Fuzzy-Sets Zugehörigkeitsfunktionen definiert worden sind.

Hierzu werden in der Praxis im Allgemeinen zwei Methoden verwendet, welche im Folgenden genauer erläutert werden.

4.2. Funktionsweise

Mamdani:

Bei Mamdani arbeitet man mit der Fläche unter der Zugehörigkeitsfunktion des Ausganges, wobei man nur die Fläche von der X-Achse bis zum Wert der Zugehörigkeit verwendet. Dies wendet man auf alle Fuzzy-Sets der linguistischen Variable des Ausganges an und erhält so eine zusammengesetzte Fläche. Anschließend berechnet man mit Hilfe der numerischen Integration den Flächenschwerpunkt dieser zusammengesetzten Fläche. Für den zu erzeugenden scharfen Wert ist allerdings nur X-Komponente des Schwerpunktes von Bedeutung.

Um unser Beispiel fortzusetzen:

Diese Abbildung zeigt die linguistische Variable der Bremskraft:

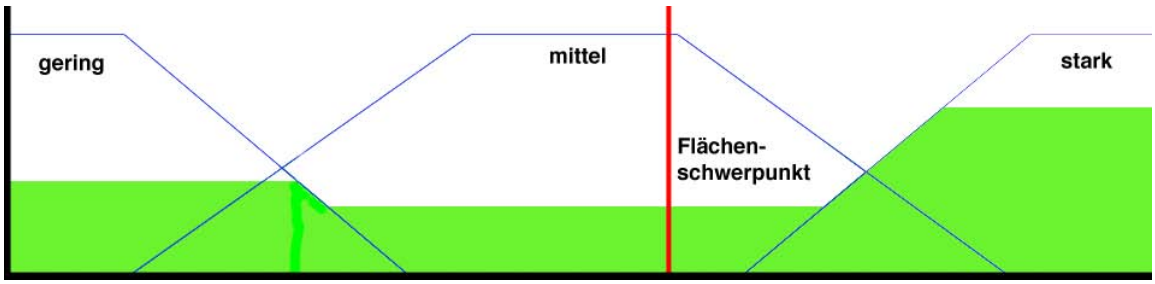


Abb.22: Flächenschwerpunkt mamdani

Hier erhalten wir einen Flächenschwerpunkt, der für eine exakte (scharfe) Bremskraft steht.

Sugeno:

Bei Sugeno sind im Vergleich zu Mamdani die Flächen uninteressant. Man arbeitet mit schmalen Streifen, deren Höhen genau den Zugehörigkeiten der Fuzzy-Sets entsprechen. Hiermit wird die Berechnung des Flächenschwerpunktes massiv vereinfacht, da die Berechnung mit wesentlich einfacheren Formen von statten geht. Diese Tatsache bringt natürlich eine Geschwindigkeitsverbesserung mit sich. Die Genauigkeit des Systems leidet vor allem dann unter dieser Vereinfachung wenn die Zugehörigkeitsfunktionen der Fuzzy-Sets nicht symmetrisch sind.

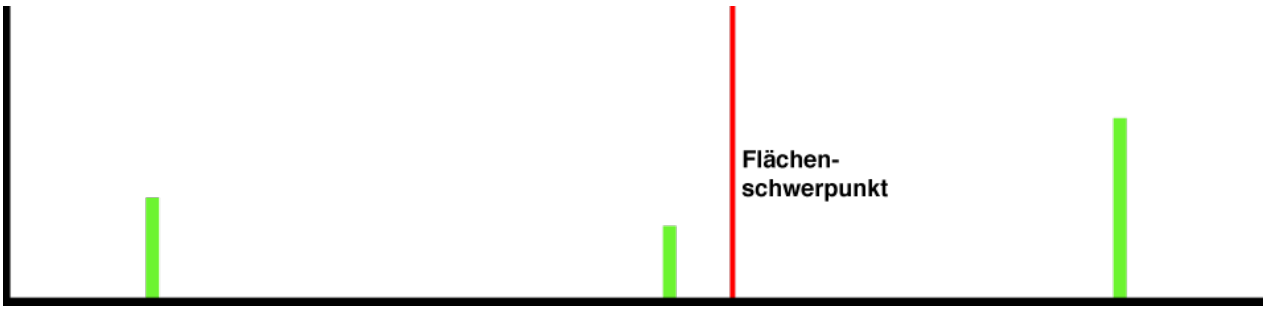


Abb.23: Flächenschwerpunkt sugeno

Wie hier erkennbar erhält man durch die Defuzzyfizierung mit Hilfe von Sugeno nahezu den gleichen Flächenschwerpunkt auf eine schneller und einfachere Art.

Künstliche Intelligenz in unserer Simulation

1. Einführung:

Um in unserer Simulation eine menschliche Komponente abzubilden bedienten wir uns der Methoden der künstlichen Intelligenz. Im allgemeinen werden im Zusammenhang mit künstlicher Intelligenz meist zwei Methoden genannt:

- **Neuronale Netze**
- **Fuzzy Logic**

Beide Systeme ermöglichen es menschliches Verhalten gewissermaßen abzubilden. Wir entschieden uns zwar dafür Fuzzy Logic einzusetzen, jedoch auf die Vorteile von neuronalen Netzen zu verzichten. Im folgenden wird erläutert welche Beweggründe uns zu dieser Entscheidung führten.

2. Warum keine neuronalen Netze?

Neuronale Netze sind prinzipiell ein Modell eines menschlichen Gehirns. Durch die Vernetzung vieler Neuronen, welche strikte Rechenanweisungen ausführen und einer Lernregel erreicht ein neuronales Netz eine Art Lernfähigkeit. Einsatzgebiete für neuronale Netze sind zum Beispiel Schrifterkennung oder auch intelligente Suchalgorithmen. Das System kann nur dann lernen, wenn ihm auf irgend eine Art beigebracht wird, wie es sich in bestimmten Situationen verhalten soll.

Beispiel:

Ein Schrifterkennungssystem erkennt einen Buchstaben falsch. Der Bediener bemerkt dies und korrigiert den Fehler entsprechend. Hier kann das neuronales Netz dahingehend verändert werden, dass es diesen Buchstaben in Zukunft, wenn er ähnlich geschrieben wird, richtig erkennt. So kann das System auch an außergewöhnliche Handschriften angepasst werden.

Aus der Tatsache heraus, dass ein neuronales Netz eine Art Rückmeldung über die Richtigkeit seiner Ergebnisse braucht, lässt sich erklären weshalb es in unserer Simulation nicht zum Einsatz eben dieser kommt.

Das einzige Einsatzgebiet in dem nach unseren Überlegungen der Einsatz eines neuronalen Netzes sinnvoll gewesen wäre ist die Steuerung des Computergegners. Hierbei eröffnete sich uns das Problem, dass es in unserer Simulation keine sinnvoll einsetzbaren Rückmeldungen für die Richtigkeit der Entscheidungen des Computergegners gab, was für die Lernfähigkeit des Computergegners zwingend erforderlich gewesen wäre.

Nachdem wir diesbezüglich einige Überlegungen angestellt hatten kamen wir zu dem Schluss, dass ein neuronales Netz ohne die damit verbundene Lernfähigkeit keine zufrieden stellende Lösung für unseren Computergegner darstellte.

3. Warum Fuzzy Logic?

Für unsere Simulation ist es wichtig menschliche Verhaltensmuster zu imitieren um das Verhalten von Menschen in einem Wirtschaftssystem aufzuzeigen. Nach reichlicher Überlegung und eingehendem Studium der heute üblichen Methoden der künstlichen Intelligenz kamen wir zu dem Schluss, dass Fuzzy Logic eine für uns optimale Lösung in vielen Bereichen darstellt.

Mit Hilfe von Fuzzy Logic ist es möglich aus mehreren exakten Kennzahlen neue exakte Werte zu erhalten, wobei deren „Berechnung“ mit Hilfe einer Art menschlichen Logik erfolgt.

Diese Tatsache kommt uns bei vielen Aufgaben sehr entgegen, da wir eben diese menschliche (unexakte) Komponente simulieren wollen.

4. Computergegner

4.1. Einführung:

Es war Teil unserer Diplomarbeit einen Computergegner zu implementieren, welcher sich prinzipiell wie ein Mensch verhält und das gesamte Wirtschaftssystem versucht zu steuern. Wie oben erläutert erschien uns der naheliegende Einsatz von neuronalen Netzen zur Realisierung des Computergegners nicht sinnvoll. Wir beschlossen also ein Einfacheres System zu entwickeln, mit Hilfe dessen wir einen zufrieden stellenden Gegner realisieren konnten.

4.2. Grundlegende Überlegungen:

Der von uns verwendete Ansatz verbindet strikte, unveränderbare Komponenten mit einer dynamischen Komponente. Als ersten Schritt legten wir Strategien fest, die der Gegner verfolgen kann. Diese Strategien sind:

- Wirtschaftlicher Aufbau:
- Vergrößern der Einwohnerzahl:
- Vergrößern der Armee:

Um dem Gegner eine Art Persönlichkeitsprofil zu geben werden die „Wichtigkeiten“ der einzelnen Strategien verändert. Wichtigkeit bedeutet in diesem Zusammenhang den prozentuellen Anteil der Gesamtspielzeit den der Gegner in die Verfolgung dieser Strategie investieren soll.

4.3. Realisierung:

Nachdem ein Computer mit Begriffen wie „eine Stadt vergrößern“ nicht arbeiten kann entwickelten wir eine Art Baumstruktur, welche vorgefertigte, fixe Abläufe für

die Verfolgung bestimmter Strategie enthält. Um diese Struktur veränderbar zu halten realisierten wir deren Erstellung über ein eigens dafür entwickeltes Format.

Hier ein Beispiel:

Die derzeitige Strategie des Gegners besagt, dass er die Einwohnerzahl seiner Stadt vergrößern soll. Der Ablauf der ihm dies ermöglicht ist in der oben erwähnten Baumstruktur gespeichert. So sieht eine vereinfachte Darstellung folgendermaßen aus:

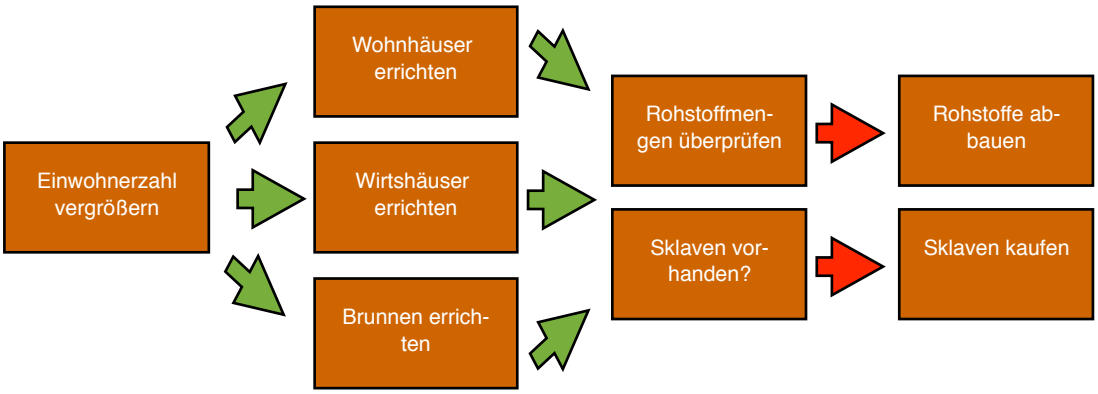


Abb.24: Beispiel Baumstruktur

Diese Beispiel zeigt lediglich einen kleinen Ausschnitt des Netzes und dient lediglich zur Verdeutlichung des Funktionsprinzips.

Erläuterung:

Dieses Beispiel zeigt, dass der Computergegner Wirtshäuser, Brunnen und Wohnhäuser errichten muss um das gewünschte Ziel zu erreichen. Um Beispielsweise ein Wirtshaus bauen zu können muss er erstens über genügend Rohstoffe und zweitens über mindestens einen Sklaven verfügen, der die Arbeit verrichten kann. Sollte eine der beiden Bedingungen nicht erfüllt sein so muss er eben diesen Missstand an Sklaven beziehungsweise Rohstoffen ausgleichen, was in dem von uns erstellten System auch eine Änderung der Strategie nach sich ziehen kann.

5. Fuzzy Logic Systeme in unserer Simulation:

5.1. Funktionsweise:

5.1.1. Defuzzyfizierung:

Wie in der vorhergehenden Einführung bereits erläutert existieren zwei gebräuchliche Methoden um diese Problematik zu lösen:

- **Mamdani**:

  Bei dieser Methode wird der benötigte Flächenschwerpunkt der resultierenden Fläche mit Hilfe der numerischen Integration errechnet. Ein Vorteil dieses Verfahrens ist, dass es mit allen beliebigen Formen von Fuzzy-Sets funktioniert und dabei eine relativ hohe Genauigkeit liefert.

- **Sugeno:**

  Hierbei werden die Resultierenden Flächen durch schmale Streifen der gleichen Höhe ersetzt. Dadurch wird die Schwerpunktberechnung massiv vereinfacht und beschleunigt. Der gravierende Nachteil dieser Methode wirkt sich bei asymmetrischen Fuzzy-Sets aus. In diesem Fall ist die Genauigkeit der Methode sehr gering.

- **Auswahl der Methode:**

  Um uns die Entscheidung zu erleichtern führten wir Geschwindigkeitstests mit beiden Methoden durch. Nachdem Sugeno nur eine unwesentliche Beschleunigung des Systems brachte und wie oben ausgeführt nicht an die Genauigkeit von Mamdani heranreichen kann entschieden wir uns dafür Mamdani einzusetzen. Dadurch bietet unser System den Vorteil, dass es mit allen beliebigen Formen von Fuzzy-Sets funktioniert und eine höhere Genauigkeit erreicht.

5.1.2. Struktur des Systems:

Unsere selbst entwickelte Fuzzy Logic Lösung realisierten wir durch eine objektorientierte Struktur, da dies erstens zweitens die Implementierung im Vergleich zu einer prozeduralen Struktur massiv vereinfacht und zweitens dem heutigen Stand der Software-Architektur entspricht. Hier nun ein Klassendiagramm, das den wesentlichen Aufbau des Systems verdeutlicht:

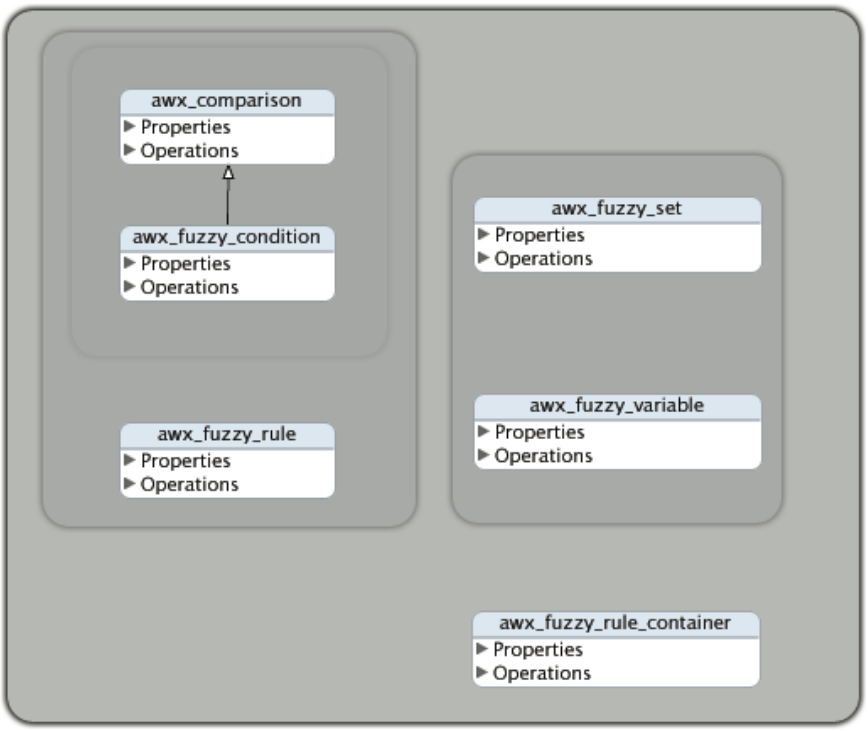


Abb.25: Klassendiagramm Fuzzy Logic

- **awx_comparison** und **awx_fuzzy_condition** dienen zur Speicherung und Auswertung von Bedingungen.
- **awx_fuzzy_rule** beinhaltet eine Bedingung (**awx_fuzzy_condition**) und eine liste von Effekten, welche ausgeführt werden wenn die Bedingung erfüllt ist.
- **awx_fuzzy_set** ist eine Struktur, die die einzelnen Kontrollpunkte für ein Fuzzy-Set enthält. Darüber hinaus ermöglicht diese Klasse die Berechnung der Zugehörigkeit abhängig vom Eingangswert.

- **awx_fuzzy_variable** verbindet beliebig viele Fuzzy-Sets (**awx_fuzzy_set**) zu einer Fuzzy-Variable. Diese Klasse ermöglicht es sowohl die einem Eingangswert entsprechenden Zugehörigkeiten zu berechnen als auch bei gegebenen Zugehörigkeiten einen Flächenschwerpunkt zu berechnen. Dieser Flächenschwerpunkt dient zur Defuzzyfizierung.

- **awx_fuzzy_rule_container** verbindet beliebig viele Fuzzy-Variablen (**awx_fuzzy_variable**) mit darauf anzuwendenden Regeln (**awx_fuzzy_rule**). Die Klasse bietet die eigentliche Funktionalität des Fuzzy Logic Systems. Die Funktion **calculate()** erzeugt aus exakten Eingangswerten exakte Ausgangswerte, wobei die dazwischenliegende Berechnung über Fuzzy Logic erfolgt.

5.1.3. Regelwerk:

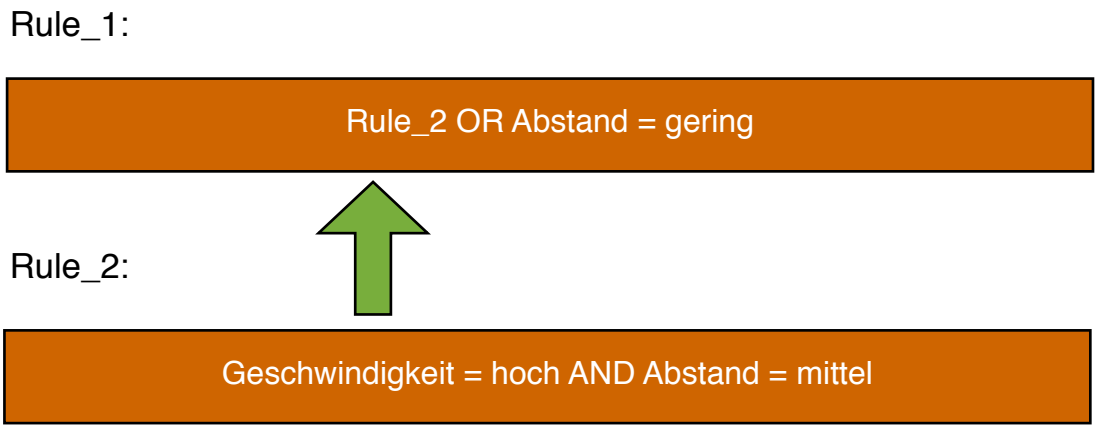
Das Regelwerk in dem alle Regeln gespeichert werden sollten erwies sich als einer der am schwierigsten zu realisierenden Teile des Fuzzy Logic Systems. Um Vorrangregeln und Klammern zu unterstützen entschieden wir uns dafür eine rekursive Baumstruktur zur Speicherung der Regeln zu verwenden.

Ein Beispiel zum besseren Verständnis:

Rule_1:

(Geschwindigkeit = hoch AND Abstand = mittel) OR Abstand = gering

Diese Bedingung zerfällt in unserem Regelsystem in 2 Bedingungen, wobei das Ergebnis der zuerst Auszuwertenden in die zweite Bedingung als Wert eingesetzt wird:



Dieses System kann bis zu jeder beliebigen Tiefe verwendet werden und hat keine Grenze für die Anzahl der Klammerebenen pro Regel.

5.1.4. Entwicklung:

Die Implementierung des Fuzzy Logic Systems wurde aus Gründen der Übersichtlichkeit und Einheitlichkeit von einem einzigen Teammitglied übernommen. Dadurch wurden exakte Schnittstellenbeschreibungen innerhalb des Systems hinfällig. Wir definierten lediglich die Schnittstellen des Systems nach außen exakt, da diese auch von anderen Teammitgliedern verwendet werden würden. Georg Haaser wurde mit der Implementierung des Systems betraut und konnte seine Arbeit daran termingerecht abschließen.

Bei der Entwicklung trat ein massives Problem auf, da ich (Georg Haaser) ursprünglich davon ausgegangen war, dass Klammern und Auswertungsreihenfolgen in der Bool‘schen Logik überflüssig seien. Diese Annahme ist nicht richtig und wurde von mir vorschnell getroffen. Ein einfaches Beispiel um dies zu verdeutlichen:

(0 && 0) ll 1 -> 1

0 && (0 ll 1) -> 0

Zeichenerklärung:
ll ... Bool‘sches Oder
&& ... Bool‘sches Und

Zu dieser Annahme kam ich dadurch, dass ich die Auswertungsreihenfolge in unzähligen Beispielen änderte und nie auf ein Beispiel kam, bei dem diese eine Rolle spielte. Der Fehler dabei war selbstverständlich, dass ich keinen allgemeinen Beweis dafür geführt hatte (Was auch nicht möglich ist, da die Reihenfolge eine Rolle spielt). Unter dieser falschen Annahme konzipierte ich ein System um die Regeln des Fuzzy Logic Systems auszuwerten. Nachdem ich das System fertiggestellt hatte wurde ich durch ein offensichtliches Fehlverhalten des Systems darauf aufmerksam, dass die Auswertungsreihenfolge sehr wohl von Bedeutung ist. Ich konzipierte das nun wesentlich kompliziertere System dafür neu und implementierte es. Dieser unvorhergesehene Eingriff ins System war mit enormem Aufwand verbunden, da einige Schnittstellen zum Regelwerk neu definiert werden mussten, was natürlich an allen Punkten an denen diese verwendet wurden weitere Anpassungen nach sich zog. Ich konnte die entstandene Verzögerung jedoch kompensieren und das System termingerecht fertigstellen.

Durch diesen Fehler wurden mir die möglichen Konsequenzen mangelhafter Recherche, Planung und Beschäftigung mit einem Thema erneut bewusst.

Von diesem massiven Fehler abgesehen traten bei der Entwicklung des Systems keine erwähnenswerten oder unvorhergesehenen Probleme auf.

Der Test der Einzelkomponenten des Systems erfolgte wie in der Software Entwicklung üblich durch eine Vielzahl von Testfällen.

Der Test des Gesamtsystems erwies sich als schwierig, da der gesamte Rechenweg, den das System durchgeht, sehr komplex ist. Mangels anderer Alternativen überprüfte ich auch diese Funktionalität anhand von unterschiedlichen Testfällen. Ich rechnete die Testfälle händisch durch und verglich mein Ergebnis mit dem Ergebnis des Systems. In diesen Tests wies das System keine Fehler auf.

5.2. Einsatzgebiete:

5.2.1. Bürgerzuwanderung:

In unserer Simulation gilt es eine attraktive, wirtschaftlich florierende Stadt aufzubauen. Je attraktiver eine Stadt ist, desto mehr Bürger wollen in dieser Stadt einziehen. Nun galt es eine Kennzahl für die Attraktivität einer Stadt zu berechnen.

Hier entschieden wir uns ein Fuzzy Logic System einzusetzen, welches aus mehreren Einflussgrößen auf die Attraktivität der aufgebauten Stadt schließt.

In unserem sehr stark vereinfachten System beschlossen wir nur einige der zahlreichen Einflussfaktoren zu berücksichtigen:

- **Anzahl der Brunnen in der Stadt pro Einwohner:**
  Dieser simplifizierte Wert tritt in unserem Spiel an die Stelle einer Wasserversorgungsgüte.

- **Anzahl der Wirtshäuser in der Stadt pro Einwohner:**
  Dieser Faktor steht für eine Güte der Infrastruktur der aufgebauten Stadt.

- **Verfügbare Wohnmöglichkeiten:**
  Dieser Faktor wird nicht direkt im Fuzzy Logic Zuwanderungssystem berücksichtigt, da die Anzahl der freien Wohnungen ein exakter Wert ist und eine unscharfe Interpretation dazu führen könnte, dass mehr Bürger in die Stadt einziehen als Wohnungen verfügbar sind.

- **Berufsmöglichkeiten:**
  Ähnlich den Wohnmöglichkeiten werden auch die Berufsmöglichkeiten nicht direkt im Fuzzy Logic Zuwanderungssystem berücksichtigt. Es wird lediglich geprüft ob Stellen verfügbar sind.
  Nachdem zuwandernde Bürger über keine Stelle verfügen kann hier keine Berufsattraktivität verglichen werden.

5.2.2. Arbeitsgeschwindigkeit:

Die Arbeitsgeschwindigkeit von Sklaven und Bürgern ist variabel und kann nicht durch einen Wert exakt festgelegt werden, da sie von vielen Faktoren abhängt.

Wir entscheiden uns auch hier dazu ein Fuzzy Logic System einzusetzen um die komplexe Problematik der Arbeitsgeschwindigkeit zu lösen.

Die Arbeitsgeschwindigkeit wird beim Errichten von Gebäuden, beim Abbau von Rohstoffen und bei der Arbeit eines Bürgers verwendet.

Wir entschieden uns dazu folgende Einflussfaktoren zu verwenden:

- **Motivation:**
  Die Motivation eines Stadtbewohners entspricht im wesentlichen der Attraktivität der Stadt. Je attraktiver die Stadt ist in der ein Bürger arbeitet desto schneller verrichtet er seine Arbeit.

- **Müdigkeit:**

  Die Müdigkeit ist ein Wert, der abhängig von der verrichteten Arbeit erhöht wird. Anstrengende Arbeit macht einen Bürger schneller müde als eine weniger anstrengende Arbeit. Mit dieser Größe bezwecken wir, dass es nicht wirtschaftlich ist eine Arbeit immer von den gleichen Einwohnern verrichten zu lassen.
  Bei einem Einwohner der nicht arbeitet wird die Müdigkeit langsam wieder geringer, was bedeutet, dass sich der Einwohner gewissermaßen erholt.
  Ein schlafender Bürger erholt sich wesentlich schneller als ein wacher Bürger.

- **Wettersituation:**

  Dieser Einflussfaktor scheint auf den ersten Blick etwas deplatziert. Bei genauerer Betrachtung der Situation wird jedoch klar, dass zum Beispiel bei Bauarbeiten an einem Gebäude das Wetter einen sehr starken Einfluss auf die Arbeitsgeschwindigkeit haben kann.
  Bei arbeiten, die im inneren von Gebäuden verrichtet werden wird dieser Einfluss ignoriert.

5.2.3. Berufswahl:

Hier galt es ein System zu entwickeln, welches die Aufgabe der Auswahl von Stellen für Bürger ermöglicht. Wir lösten diese sehr komplexe Problematik mit Hilfe eines System, das eine Berufsattraktivität errechnet und aufgrund dieser Entscheidet welche Stelle ausgewählt wird. Bei dieser Auswahl werden natürlich nur Stellen berücksichtigt, die nicht bereits vergeben sind.

Ein Bürger der neu in die Stadt kommt und dementsprechend noch keine Stelle hat wählt die attraktivste Stelle.

Ein Bürger der bereits einen Beruf ausübt kann natürlich seine Stelle wechseln, wenn eine andere freie Stelle attraktiver ist als der zur Zeit von ihm ausgeübte. Nachdem ein Wechsel desr Stelle in der Realität mit beträchtlichen Risiken verbunden ist, wechselt der Bürger diese nur dann wenn die

verfügbare Alternativanstellung wesentlich attraktiver ist als seine Derzeitige. Diese Verfeinerung des Systems erreichen wir dadurch, dass die neue Stelle um 20% attraktiver sein muss als die Derzeitige, damit ein Bürger zur neuen Stelle wechselt.

Um die oben erwähnte Attraktivität zu berechnen werden folgende Einflussgrößen verwendet:

- **Einkommen**:
  Das Einkommen, das der Bürger für seine Arbeit in einem gewissen Beruf erhält ist natürlich ein wesentlicher Faktor für die Attraktivität eines Berufes.

- **Körperliche Anstrengung, die der Beruf mit sich bringt:**
  Diese Anstrengung fließt in die Berechnung negativ ein. Das bedeutet, dass ein sehr anstrengender Beruf beispielsweise nicht besonders attraktiv ist. Ein weniger anstrengender Beruf mit ansonsten gleichen Rahmenbedingungen ist natürlich attraktiver.

- **Arbeitsdauer pro Tag:**
  Je geringer die Arbeitsdauer pro Tag eines Berufes ist desto attraktiver wird dieser.

- **Die Zeit des Arbeitsbeginnes die mit dem Beruf verbunden ist:**
  Ein früher beziehungsweise später Arbeitsbeginn wirkt sich negativ auf die Attraktivität des Berufes aus. Der Rahmen in dem sich diese Zeit bewegen kann geht von 5:00 Uhr bis 12:00 Uhr.

6. Fuzzy Logic Editoren

6.1. Einführung:

Um unsere Fuzzy Logic Systeme erstellen zu können, mussten wir uns Gedanken über Editoren machen. Nachdem wir einige existierende Systeme studiert hatten und zu dem Schluss gekommen waren, dass alle vorhandenen Systeme nicht unseren Anforderungen entsprechen, beschlossen wir die notwendigen Werkzeuge zur Erstellung von Fuzzy Logic Systemen selbst zu implementieren. Unser Hauptaugenmerk lag dabei auf Funktionalität und nicht auf Benutzerfreundlichkeit beziehungsweise Stabilität, da diese Editoren nur von uns selbst, also den Entwicklern der Software, verwendet werden würden.

Im Zuge dessen entwickelten wir zwei Editoren, um unsere Fuzzy Logic Systeme zu erstellen. Hierzu verwendeten wir die Programmiersprache Cocoa, welche eine sehr umfangreiche Bibliothek zur Erstellung von graphischen Oberflächen mitliefert. Cocoa ist eine objektorientierte Programmiersprache, die auf C++ aufsetzt und dieses um eine mächtige Klassenbibliothek erweitert. Das Hintergrundsystem wurde, um Kompatibilität zu unserer restlichen Software zu gewährleisten, in C++ implementiert. Cocoa existiert nur auf **Mac OS X** Systemen und damit erstellte Programme sind auch nur auf dieser Plattform lauffähig. Dadurch, dass das Hintergrundsystem in C++ implementiert wurde, erhielten wir die Plattformunabhängigkeit unserer Simulation aufrecht.

6.2. Fuzzy-Variablen Editor:

Als erstes entwickelten wir einen Editor zur Erstellung von soggenannten Fuzzy-Variablen. Dieser Editor dient dazu mehrere Fuzzy-Sets durch einzelne Kontrollpunkte zu erstellen und diese zu benennen. Darüber hinaus kann ein Bereich festgelegt werden in dem sich der zu fuzzyfizierende Wert bewegen kann. Das Zusammenspiel dieser einzelnen Fuzzy-Sets nennt man Fuzzy-Variable. Eine Fuzzy-Variable entspricht grundlegend dem Begriff der linguistischen Variable.

Hier ein Screenshot des Programms:

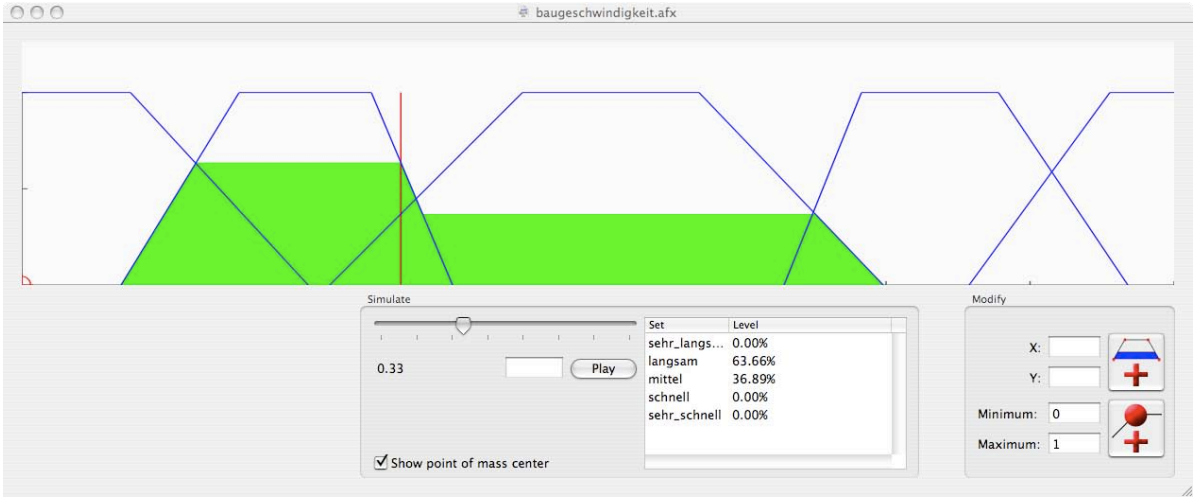


Abb.26: Fuzzy Variablen Editor 1

Das Beispiel zeigt die Fuzzy Repräsentation der linguistischen Variable der Baugeschwindigkeit. Diese besteht aus den Fuzzy-Sets **sehr_langsam**, **langsam**, **mittel**, **schnell** und **sehr_schnell**, wie man in der Tabelle rechts unten auf der Abbildung erkennen kann.

Dieser Editor bietet die Möglichkeit neue Fuzzy-Variablen zu erstellen sowie die Möglichkeit bestehende zu editieren, indem man zum Beispiel bestimmte Punkte verschieben, den Wertbereich oder auch die Bezeichnungen der Fuzzy-Sets ändern kann.

Ein Beispiel:

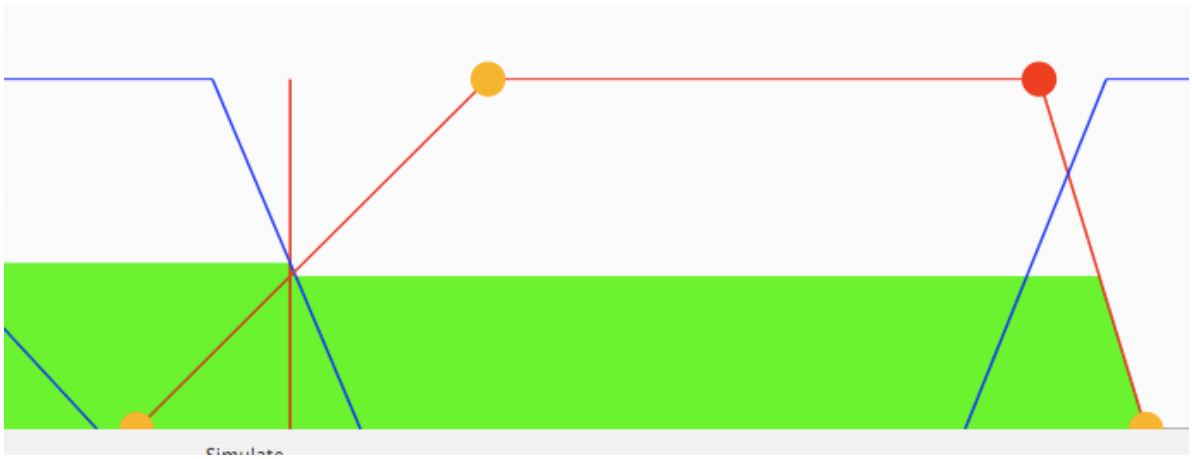


Abb.27: Fuzzy Variablen Editor 2

Hier wurde der dritte Punkt des Fuzzy-Sets **mittel** der Baugeschwindigkeit verschoben.

In unserem System ist es prinzipiell möglich beliebige Formen von Fuzzy-Sets zu erstellen, so lange sich diese aus linearen Funktionen zusammensetzen lassen.

Hier ein Beispiel für eine sehr komplizierte Form von Fuzzy-Sets:

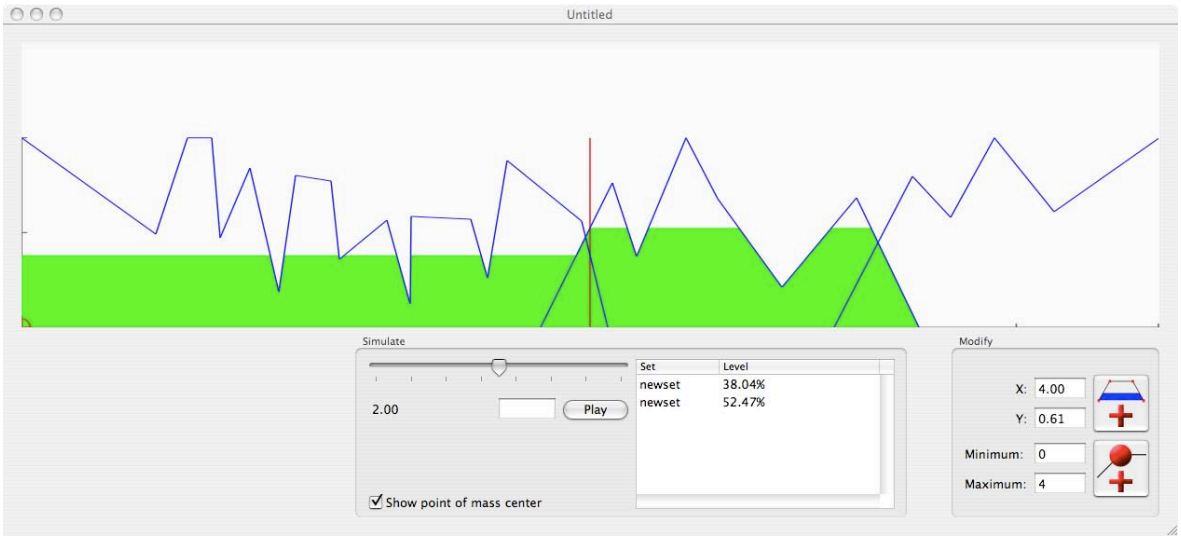


Abb.28: Fuzzy Variablen Editor 3

In der Praxis werden meist dreieckige oder trapezförmige Fuzzy-Sets eingesetzt, da kompliziertere Formen eine höhere Rechenleistung erfordern und nur eine zu vernachlässigende Verbesserung des Systems mit sich bringen. Die oben gezeigte Form dient lediglich dazu die Funktionalität unserer Software zu zeigen und entbehrt inhaltlich jeder Sinnhaftigkeit.

Die eingegebenen Daten werden von unserem Editor in einem selbst entwickeltem Binärformat gespeichert, für welches wir die Bezeichnung „**afx**“ wählten.

6.3. Fuzzy-System Editor:

Dieser Editor dient dazu mehrere Fuzzy-Variablen, die wie oben erläutert erstellt wurden, mit Hilfe von Bedingungen zu verknüpfen und die Ergebnisse auf Ausgänge des Systems zu führen. Es ist nicht zwingend erforderlich, dass ein hiermit erstelltes Fuzzy Logic System nur einen Ausgang hat. Das System unterstützt beliebig viele Eingans- und Ausgangsvariablen.

Grundlegend besteht das Programm aus zwei Teilen:

Eine Oberfläche, die es ermöglicht Fuzzy-Variablen zum System hinzuzufügen, diese als Ein- oder Ausgang zu deklarieren und eine Simulation des Systems durch verändern der Eingangswerte durchzuführen:

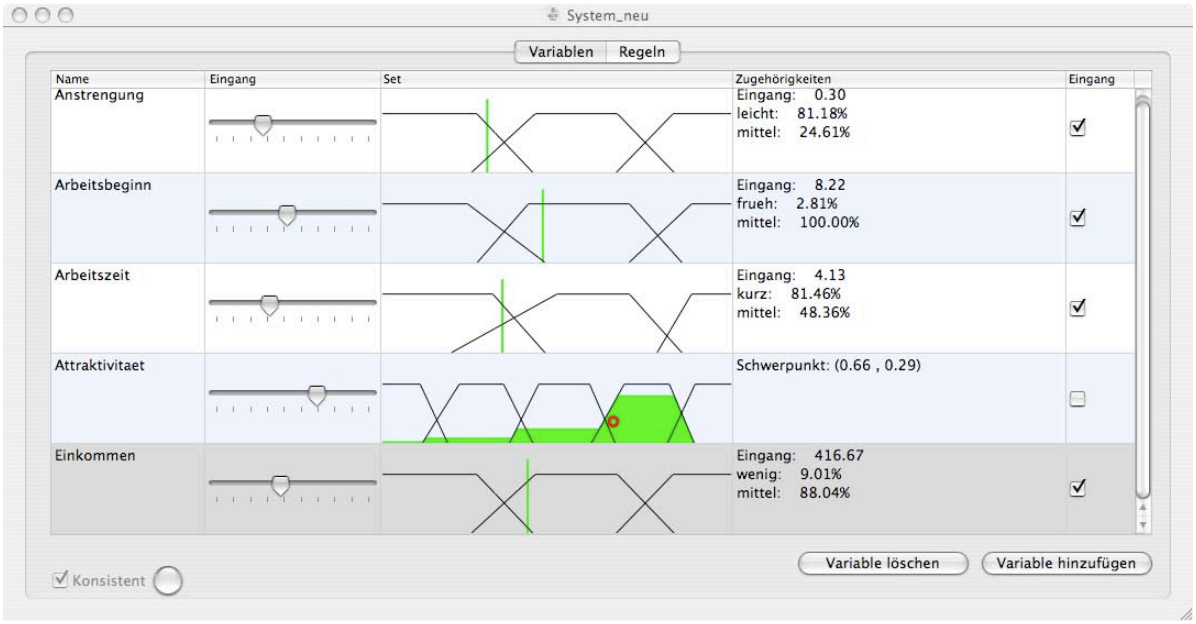


Abb.29: Fuzzy System Editor 1

Hier im Beispiel das System zur Berechnung der Berufsattraktivität.

Dieses System verfügt beispielsweise über 4 Eingänge und einen Ausgang. Die Spalte „**Eingang**“ am rechten Rand enthält eine Tickbox, welche angibt ob eine Variable ein Eingang des Systems ist oder nicht.

Der zweite große Teil des Programms dient zur Festlegung der Regeln mit denen die Eingangsvariablen verknüpft werden sollen. Die Regeln können auf zwei verschiedenen Arten festgelegt werden:

- **Textuelle Eingabe:**

  Hierbei muss berücksichtigt werden, dass man sich an eine Syntax halten muss und dass falsche Eingaben zu undefiniertem Verhalten führen können. Dies rührt daher, dass wir auf eine Syntaxprüfung verzichtet haben um die Software, die lediglich einen kleinen Teil unserer Gesamtarbeit darstellt, so simpel wie möglich zu halten. Nachdem diese Software nur von uns, den Entwicklern, verwendet wird stellt dies kein wirkliches Problem dar.

- **Regeleditor:**

  Dieses Unterprogramm ermöglicht ein vereinfachtes Erstellen von Regeln und unterstützt den Anwender dabei syntaxkonforme Regeln festzulegen.

Hier ein Screenshot des Regeleditors:

Abb.30: Regeleditor

Natürlich existiert auch eine Übersichtsoberfläche, die alle zum System gehörenden Regeln auflistet und es ermöglicht diese zu ändern, zu löschen und neue Regeln hinzuzufügen:

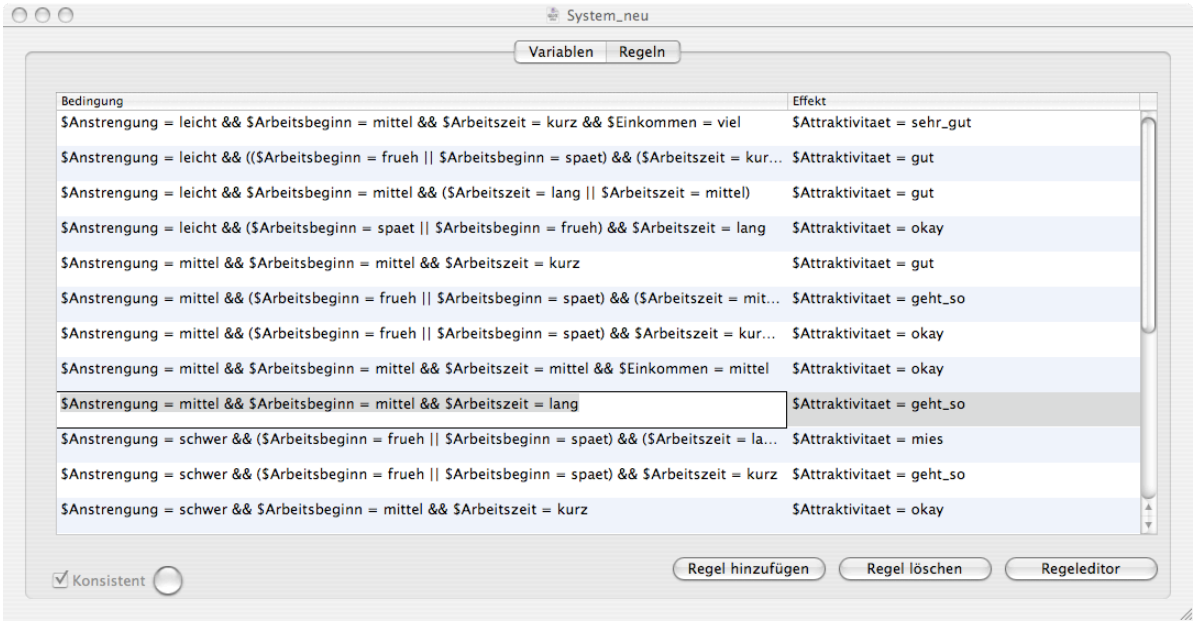


Abb.31: Fuzzy Variablen Editor 2

Das Beispiel zeigt einen Teil der Regeln, die zur Berechnung der Berufsattraktivität verwendet werden. Wie hier deutlich, wird ist es möglich in dieser Ansicht bereits vorhandene Regeln zu editieren.

Links unten auf der Maske gibt eine Tickbox an ob alle möglichen Zustände der Eingänge von den vorhandenen Regeln abgedeckt werden. Dies ist nicht systembedingt erforderlich. Einige Anwendungen des Systems erfordern jedoch ein System, welches zu jeder Eingangskonstellation einen Ausgangswert liefert.

Die Daten, welche mit Hilfe dieses Editors erstellt und editiert werden können werden in einem von uns entwickelten Binärformat gespeichert. Diese Repräsentation des Fuzzy-Systems beinhaltet die Definitionen der verwendeten Variablen sowie die existierenden Regeln. Dieses Format wurde von uns „**afsx**“ genannt.

Grundaufbau der Engine

1. Libraries

OpenGL ist jediglich Grafikschnittstelle, und stellt keinerlei Funktionalität hinsichtlich Fenstermanagement und Benutzereingaben zur Verfügung. Das zugrunde liegende Programm hat die Aufgabe alles für OpenGL vorzubereiten.

Um die für OpenGL notwendige Umgebung (OpenGL Context) zu schaffen, kann der Entwickler auf Betriebssystemspezifische API zurückgreifen. Auf Windows also unter anderem auf die WinAPI. Dadurch würde die Applikation allerdings die plattformunabhängikeit verlieren.

Um diese Problematik zu umgehen, wurde entschieden, auf eine weitere Library zurückzugreifen, welche das Zwischenstück zwischen OpenGL und dem Betriebssystem darstellen sollte.

Solche Libraries sind zum Beispiel: GLUT (Graphic Libray Utiltity Toolkit), GLfw, SDL (Simple Direct Layer)

SDL ist dabei die am weitesten verbreitete Library. Sie bietet außerdem weitgehende Audio Unterstützung und auf SDL können auch Direct3D Programme aufgesetzt werden. Weiters gibt es einige Erweiterungen für SDL, welche die Funktionalität um das decodieren verschiedenster Bildformate, Threads und vieles mehr erweitern.

Die Wartung und Aktualität dieser Library war auch ein Grund uns für diese Library zu entscheiden.

2. Aufbau

2.1. Gameloop

Sieht man von Menüs ab, so entspricht der Aufbau des Programms dem klassischen Aufbau von Applikationen, insbesondere Spiele mit Benutzerinteraktion.

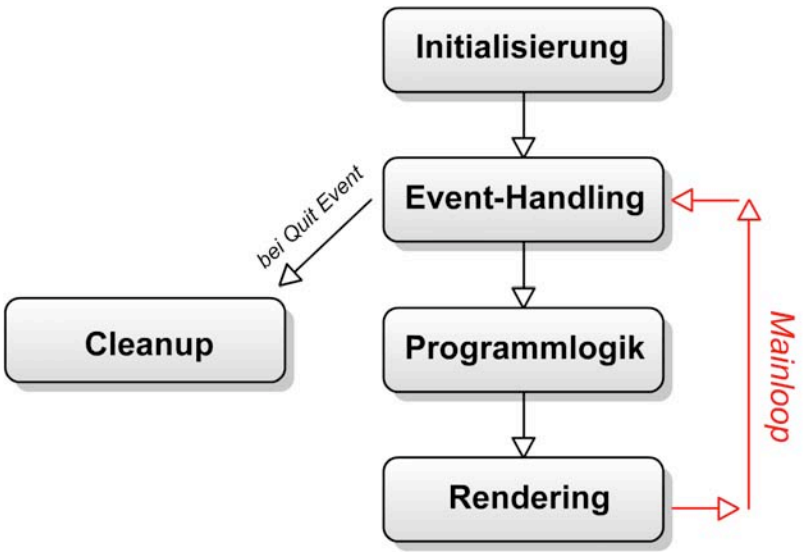


Abb.32: Gameloop

Alle Prozeduren des Gameloops werden prozedural im selben Thread ausgeführt. Es wurde entschieden, den Gameloop möglichst einfach zu halten, was auch die Verwendung von mehreren Threads ausschließt. Durch die Trennung der Programmlogik und Rendering mittels verschiedener Threads, könnte die Applikation durch den asynchronen Programmablauf höhere Performance erzielen, da die Grafikhardware nicht ständig auf die Programmlogik und das Event-Handling bzw. Instruktionen der CPU warten müsste.

Die Cleanup Prozedur entält alle notwendigen Funkionen zum deallokieren von Speicher und freigeben von Speicher der Grafikkarte.

2.2. Verwaltung von Events

Nachtrichten vom System, die von SDL abgefangen werden, werden als Events bezeichnet. Solche Events können Informationen, über gedrückte Tasten der Tastatur,

Maus oder anderen Eingabegeräten sein. Es können aber auch Meldungen beziehen, die sich auf den Status des Programms beziehen. Dies ist zum Beispiel die Meldung, dass das Programm gerade Minimiert oder der Befehl zum Schließen gegeben wurde.

Der folgende Quellcode-Ausschnitt arbeitet alle Events, die das Betriebssystem derzeit bereichtstellt ab.

```
SDL_Event event;

    while(SDL_PollEvent(&event))
    {
        switch( event.type )
        {
            case SDL_KEYDOWN:

                awx_keyboard_down(event.key.keysym.sym,0,0);
                break;

            case SDL_KEYUP:

                awx_keyboard_up(event.key.keysym.sym,0,0);
                break;

            case SDL_QUIT:

                awx_shutdown(0);
                break;

            case SDL_MOUSEMOTION:

                if(controls.mouse_down)
                {
                    awx_motion_mouse(event.motion.x,event.motion.y);
                }
                else awx_passive_mouse(event.motion.x,event.motion.y);

                break;

            case SDL_MOUSEBUTTONDOWN:
                awx_mouse(event.button.button,event.button.state,
                          event.button.x,event.button.y);
                break;
```

```
            case SDL_MOUSEBUTTONUP:
                awx_mouse(event.button.button,event.button.state,
                          event.button.x,event.button.y);
                break;
        }
    }
```

Je nach Event werden dann Funktionen von Augustus aufgerufen, welche bestimmte Aufgaben abbarbeiten. Im Beispiel des SDL_QUIT Events wird die shutdown Methode aufgerufen, welche das Cleanup des auslöst, und somit alle spieldaten aus dem Arbeitsspeicher entfernt.

Werkzeuge der Engine

1. Allgemeines

Die 3D – Szene, wie sie in 3D Spielen benötigt wird, wird in Dateien abgelegt, um bei der Spielinitialisierung geladen zu werden.

„Eine 3D-Szene enthält die Beschreibung von Objekten, die auf verschiedene Arten definiert sein können. Verbreitet ist die Darstellung durch folgende Primitiven:

Polygone werden durch drei oder mehr Punkte eindeutig beschrieben, die dreidimensionale Koordinaten haben. Weiterhin ist die Angabe eines Normalenvektors üblich, aus dessen Richtung die sichtbare Seite des Polygons bestimmt wird.

[eigene Bemerkung: Grundaufbau der Engine 5. Primitive].

NURBS sind eine bestimmte Form sogenannter Splines Ein Spline ist eine Kurve die durch mehrere Punkte im Raum exakt definiert ist. Mit einem Netz aus NURBS lässt sich eine Fläche definieren. Diese Darstellungsweise hat den Vorteil, dass sich sehr komplexe Flächen über verhältnismässig wenige Punkte definieren lassen. Um die Fläche darzustellen rechnet der Computer diese dann in Polygone um. Man kann sich den Unterschied zwischen NURBS- und Polygonenflächen ähnlich dem von Vektor- und Pixeldarstellung im 2D-Bereich vorstellen. Sie eignen sich besonders für gekrümmte Flächen, da diese auch bei starker Vergrösserung nicht „kantig“ werden.

3D-Grafikprogramme speichern diese Punkte/Polygone/NURBS/Objekte in Listen ab, die dann in einem bestimmten Format (meist als ASCII-Datei) auf den Datenträger geschrieben werden.“ [31]

---

31 vgl. 3D-Computergrafik. Online im Internet: URL: http://www.biologie.de/biowiki/3D-Computergrafik 21. April 2007,13:54

Gängige Formate sind:

- c4d - Cinema 4d Format
- obj - Ein von Format zum Austausch zwischen verschiedenen Anwendungen.
- max - 3D Studio Max Format - mit parametrischen Objekten
- 3ds - einfaches 3D Studio Max Format, Objekte werden als Meshes gespeichert

Eine weitere Form des Speicherung ist die mittels Programmcode. So z. B. das VRML-Format, das sich nicht nur auf die Listenförmige Angabe von Punkt- oder Kantenlisten beschränkt, sondern wo in einer Art Programmiersprache die 3D-Szene beschrieben wird. Diese Dateien kann man, Sprachkenntnisse vorausgesetzt, auch ohne einen besonderen Editor erstellen oder manipulieren. [32]

OpenGL gibt uns die Möglichkeit einzelne Primitiven und NURBS hardwarebeschleunigt zu berechnen und darzustellen [siehe auch. „Einführung in OpenGL - 5. OpenGL-Pipeline]. Im Bereich der Echtzeitgrafik, hauptsächlich bei Computerspielen, hat sich die Verwendung von Primitiven, meist Dreiecken, durchgesetzt. Dies ist einerseits auf die leichte Handhabung, andererseits auf diverse Optimierung der Grafikkarten diesbezüglich zurückzuführen.

Die Grafikengine sollte die notwendigen Hilfsmittel zur Verfügung stellen, um Information über die 3D–Szene abstrahierter zu handhaben. OpenGL arbeitet auf einer sehr grundlegenden Ebene, eine Engine arbeitet bereits mit sehr komplexen Objekten, zum Beispiel einer Spielfigur. Sie bietet dem Grafiker außerdem Möglichkeiten seine Modelle möglichst komfortabel in das Spiel zu integrieren, ohne von ihm Wissen über die interne Handhabung und die Darstellung über OpenGL zu verlangen.

---

[32] vgl. 3D-Computergrafik. Online im Internet: URL: http://www.biologie.de/biowiki/3D-Computergrafik 21. April 2007,13:54

Die Wahl des Modell Formates, welches alle notwendigen Informationen über ein Objekt in der 3D Szene beinhaltet, ist für die Programmierung der Grafikengine ausschlaggebend.

2. Grundsätzlicher Aufbau eines Modell Formates

Für das Rendering fundamentale Informationen sind:

- Positionen der Objekteckpunkte (Vertices)
- Anordnungen der Vertices für korrekten Aufbau der Dreiecke
- Normalvektoren (für Definition der Dreiecks-Vorderseite)
- Texturkoordinaten

Anhand des RTG Modellformates lässt sich dies sehr gut veranschaulichen, da der Aufbau sehr simpel ist und das Format außerdem in Textform vorliegt. Veranschaulicht wird dies mittels einer Ebene, welche aus 2 Dreiecken besteht.

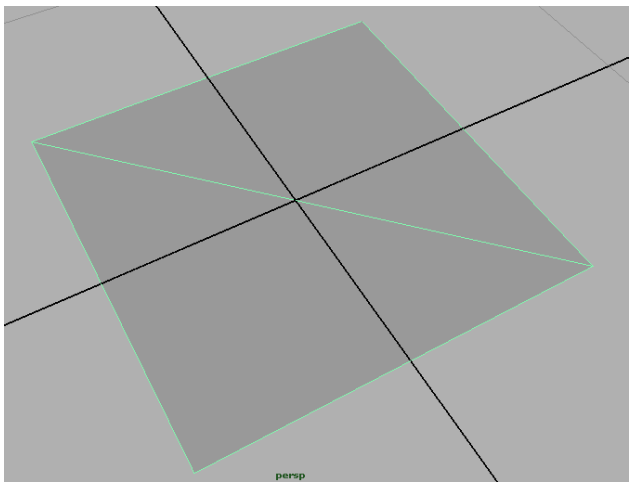


Abb.33: Lichtberechnete Ebene bestehend aus 2 Dreiecken.

Mit aktivierter Texturierung sieht das Resultat wie folgt aus:

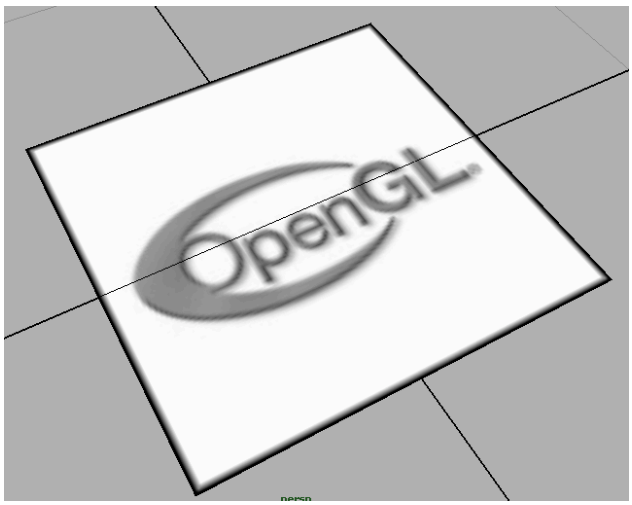


Abb.34: Texturierte Ebene bestehend aus 2 Dreiecken

Wobei folgende Textur verwendet wurde:

Abb.35: OpenGL Logo, verwendet als Textur im obigen Beispiel
(Using the Open GL Logohttp://www.opengl.org/about/logos/ 14.5 2007, 14:00)

Die Modellinformationen werden wie folgt abgelegt. Dabei handelt es sich um das obige Beispielobjekt, in Form einer RTG Datei:

```
HEADER_TITLE Alias Rtg Output
HEADER_VERSION 4.04

NUMBER_OF_OBJECTS 1

OUTPUT_VERT_NORMS    on
OUTPUT_VERT_COLORS   off
OUTPUT_TEX_COORDS    on
OUTPUT_POLY_NORMS    off
OUTPUT_HIERARCHY     on
OUTPUT_LOCAL         off
SHOW_INDEX_COUNTERS off
OUTPUT_MATERIALS     on
OUTPUT_ANIMATION    off
```

Format Header

```
MATERIAL_LIST 1
    MATERIAL 0
    NAME lambert1
    AMBIENT 0.50 0.50 0.50
    DIFFUSE 0.40 0.40 0.40
    SPECULAR 0.00 0.00 0.00
    EMMISION 0.00 0.00 0.00
    SHININESS 0.00
    TRANSPARENCY 0.00

END_MATERIAL_LIST
```

Informationen über Materialien des Objektes

```
HIERARCHY_LIST HX 1 top_level
0 P pPlane1
    tran: 0.00 0.00 0.00
    rot: 0.00 0.00 0.00
    scal: 1.00 1.00 1.00

END_HIERARCHY_LIST
```

Verschiebungen/Rotationen des Objektes

```
OBJECT_START pPlane1 v4 n4 t4 p2
USES_MATERIAL 0 lambert1

VERTEX local
  -0.500000 0.000000 0.500000
  0.500000 0.000000 0.500000
  -0.500000 -0.000000 -0.500000
  0.500000 -0.000000 -0.500000
```

Eckpunkte (Vertices) des Modelles

```
NORMAL
  0.000000 1.000000 -0.000000
  0.000000 1.000000 -0.000000
  0.000000 1.000000 -0.000000
  0.000000 1.000000 -0.000000
```

Normalvektoren einzelner Vertices

```
TEXCOORD
  -0.000000 0.000000
  1.000000 0.000000
  -0.000000 1.000000
  1.000000 1.000000
```

Texturkoordinaten einzelnen Vertices

```
POLYGON
v 0 1 2 n 0 1 2 t 0 1 2
v 1 3 2 n 1 3 2 t 1 3 2

OBJECT_END
```

Reihenfolgen bzw. Indizes, wie die einzelnen Vertices zu Dreieken verbunden werden

3. Auswahl des Modell Formates

3.1. Anforderungen an das Format:

- Einfachheit; Einfache Beschreibung der Primitiven, anstatt aufwändiger Szenenbeschreibungen
- Beinhaltung aller für das Rendering erforderlichen Informationen (Vertices, Primitive, Texturkoordinaten, Normalvektoren)
- Möglichkeit zum Speichern von Animationen
- Performance-Aspekte: Aufwändigkeit des Aufbaues, Größe der Dateien, vor allem bei Animationen relevant.

Oft werden leider Programmspezifische Informationen benötigt, welche natürlich kein Format bieten kann. Aus folgenden Gründen wurde entschieden, ein eigenes Modellformat einzusetzen:

- Für animierte Modelle benötigen wir die Möglichkeit, bestimmte Frames der Animation mit Bezeichnungen und Aktionen zu versehen. Dies soll am Beispiel des 3D Modells des Arbeiters veranschaulicht werden. Die folgende Grafik zeigt die einzelnen Animationen des Modelles:

1) Stehen, Atmen	2) Laufen	3) Arbeiten (Hämmern)	4) Verteidigen	5) Sterben

Abb.: Gliederung einer Animation in einzeln ansprechbare Teilanimationenen.

Die „Stehen, Atmen“ Animation sollte wiederholend abgespielt werden, wenn der Arbeiter gerade nichts anderes zu tun hat. Sobald der letzte Frame abgespielt wurde, muss die Animation wieder von vorne beginnen

Die „Arbeiten (Hämmern)“ Animation sollte beim Abspielen des letzten Frames eine Aktion, wie zum Beispiel das Inkrementieren des Baufortschrittes eines Gebäudes, an welchem dieser gerade arbeitet, ausführen.

- Unser Format sollte zusätzlich Positionen von Partikelsystemen und anderen Spezialeffekten beinhalten.
- Informationen über spezielle Zustände von Häusern, wie dem Fortschritt am Gebäude bzw. den Grad der Zerstörung werden benötigt.

4. Spezifikation von APX

Im Folgenden wird, wenn von APX gesprochen wird, immer unser spezielles, proprietäres Modellformat gemeint.

4.1. Anforderungen und Inhalt des Formats:

- alle für das Rendern von Primitiven notwendigen Informationen (Vertices, Normalvektoren, Texturkoordinaten)
- muss das Speichern von Animationen ermöglichen
- Informationen über Animationssteuerung, Bildaktionen
- Informationen für die Lichtberechnung (Materialen, Shaderdefinitionen)
- Informationen für Spezialeffekte (z.B.: Positionen von Partikelsystemen)
- Hohe Performance hinsichtlich der Ladezeit
- Verfügbarkeit von Schnittstellen zur Modeling-Software

Um unabhängig von Modeling-Software zu bleiben, wurde entschieden ein Format zu benutzen, welches von verschiedener Modeling-Software unterstützt wird, um dieses anschließend in APX zu konvertieren.

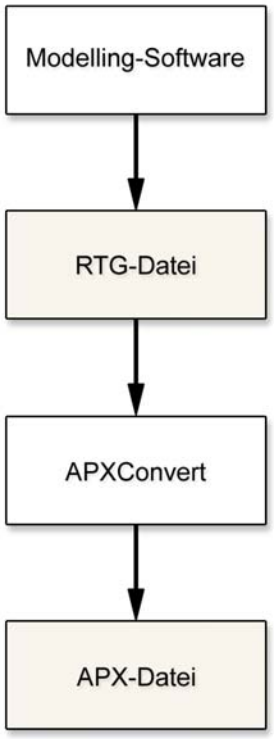


Abb.36: 4056: Konvertierungsweg aus der Modellling-Software

5. APXConvert

5.1. Anforderungen

Die Einführung von APX machte die Entwicklung einer Konvertierungssoftware, welche von RTG zu APX konvertiert, notwendig. Weiters sollte diese Konvertierungssoftware hohen Komfort für die Einstellung aller notwendigen Parameter bieten. Im Folgenden wird diese Konvertierungssoftware immer als APXConvert bezeichnet.

Um dem Grafiker das Einstellen von Materialien und Animationen zu erleichtern, wurde entschieden diese Konvertierungssoftware mit einer Echtzeit-Darstellung auszustatten.

Um einzelne Teile von 3D Objekten separat zu transformieren bzw. separat anzusprechen wurde entschieden, diesem Editor die Möglichkeit zu geben, mehrere RTG Modelle zu importieren und diese als Gesamtobjekt zu exportieren, wobei die

einzelnen RTGs separat ansprechbar bleiben. Informationen über alle RTGs bzw. APX Objekte einer Szene werden in einer weiteren Datei gespeichert. Diese Datei wird im folgendem stets als AAX bezeichnet.

AAX Container und APX Objekte besitzen jeweils eigene, interne Koordinatensysteme. Wobei diese über Transformationsmatrizen definiert sind. Die folgende Grafik soll die verschiedenen Koordinatensysteme veranschaulichen.

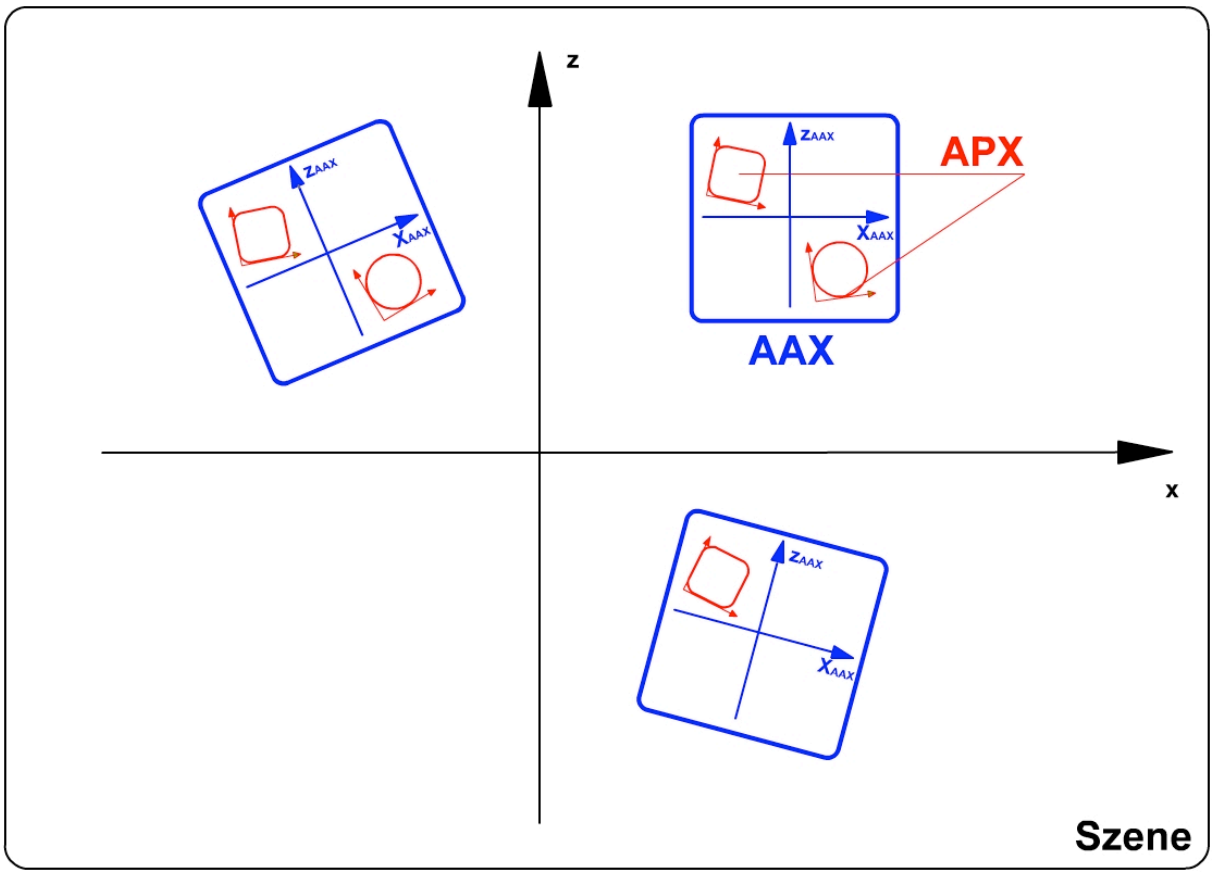


Abb.37: Schematische Darstellung von AAX-Containern, welche APX Objekte beinhalten in der 2D Ebene.

Somit ergeben sich folgende Hauptfunktionen von APXConvert:

- Laden von beliebig vielen RTG Modellen
- Positionieren, Drehen von diesen Modellen
- Echtzeit 3D Darstellung aller geladenen Modelle (mit Echtzeit – Schatten um den Grafiker den visuellen Eindruck wie im Spiel zu geben)
- Animationen
    - Laden einzelner Modelle, welche zu einer Animation assembliert werden.
    - Einstellmöglichkeiten von Bildbezeichnungen, Bildaktionen (Gehe zu Bildnummer, Gehe zu Bild mit Bezeichnung, Stop)
    - Userschnittstelle (Abspielen, Stoppen, Zurückspulen, usw.) für Echtzeitvorschau der Animation
- Hinzufügen von beliebig vielen Markierungen und Partikelsystemen
- Interface zum Einstellen von Materialien und Shadern
- Export von APX und AAX

5.2. Realisierung

5.2.1. Allgemein

Im Folgenden handelt es sich keinesfalls um eine komplette Programmdokumentation, sondern um einen kurzen Einblick in die Funktionen dieses Programms.

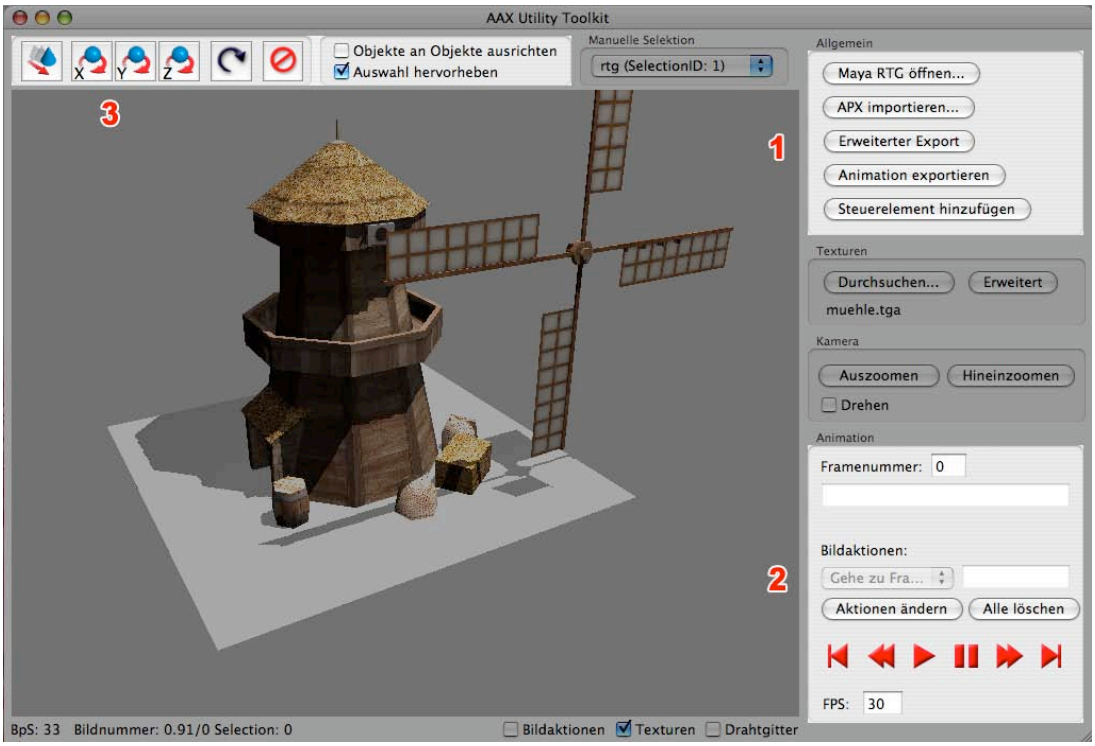


Abb.38: Screenshot von APXConvert mit importiertem Mühlenmodell und dem Mühlrad als einzelnes Objekt.

- Dieser Funktionsblock widmet sich dem Öffnen/Speichern und dem Hinzufügen von Steuerelementen, wie zum Beispiel Partikelsystemen.
- Diese Funktionen ermöglichen Einstellungen hinsichtlich der Animationssteuerung.
- Die Translations- und Rotationsfunktionen ermöglichen das Platzieren von einzelnen Objekten im Raum, sowie die Platzierung dieser auf anderen durch automatische Kollisionserkennung.

5.2.2. Texturierung

Jedem einzelnen Objekt soll eine Textur zugewiesen werden. Um auch in Zukunft eine solide Basis darzustellen und das Überlagern von mehreren Texturen, sowie erweiterte Effekte wie Bumpmapping zu ermöglichen, wurde entschieden beliebig viele Texturen pro Objekt zu ermöglichen.

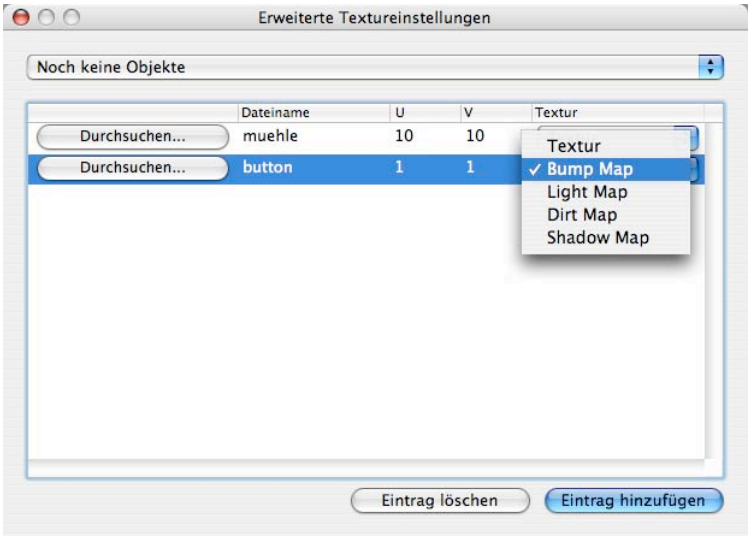


Abb.39: Screenshot, in welchem Texturen für ein APX Objekt ausgewählt werden.

5.2.3. Einstellungen von Materialien

Für die Lichtberechnung werden im Spiel Materialien herangezogen. Ein Material wird beschrieben über:

- Farbe der diffusen Lichtreflexion
- Farbe, die von der Umgebung reflektiert wird (ambient)
- Farbe, in welcher Lichtquellen totalreflektiert werden

Diese Einstellungen werden über das Materialeinstellungsfenster getroffen.

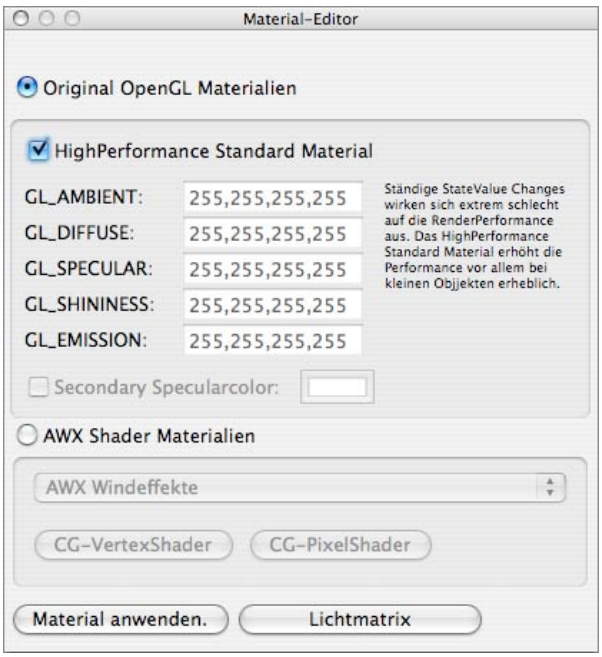


Abb.40: Screenshot des Fensters, in welchem Materialeinstellungen getroffen werden.

5.2.4. Transformationen

Einzelne Objekte der Szene können über die Manuelle Transformation exakt positioniert und gedreht werden.

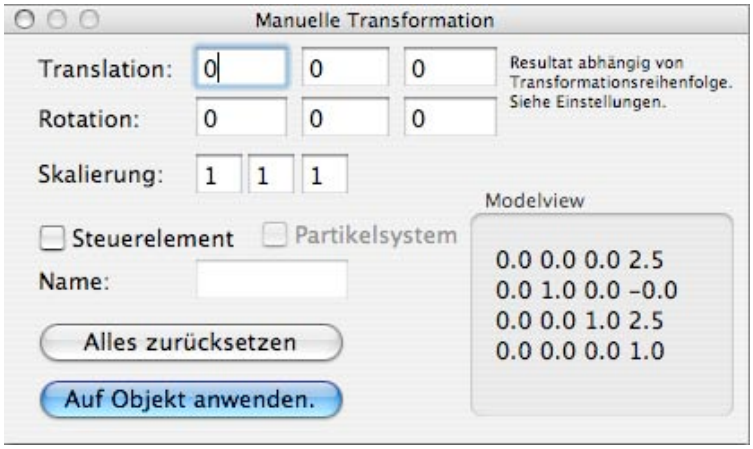


Abb.41: Screenshot des Fensters, welches manuelle Transformation von APX-Objekten ermöglicht.

5.2.5. Renderoptionen

Für alle Objekte der Szene (AAX) geltende Render-Einstellungen können hier ebenfalls getroffen werden.

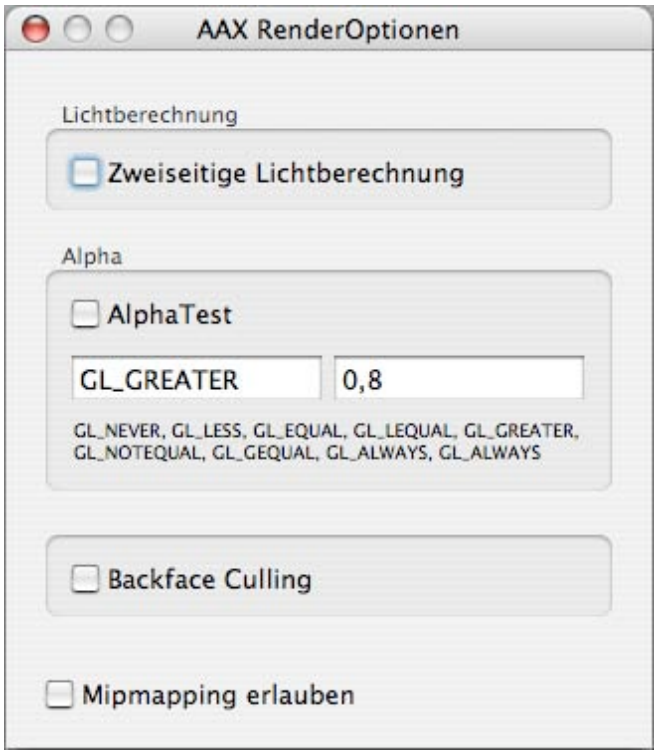


Abb.42: Screenshot des Fensters, welches das Einstellen der Renderoptionen erlaubt.

- Zweiseitige Lichtberechnung: Diese Einstellung aktiviert die Lichtberechnung auch für Hinterseiten von Dreiecken (das heißt, entgegen die Richtung des Normalvektors). Dies wird beispielsweise beim Rendering von Objekten verwendet, welche aus nur einer Ebene bestehen (zum Beispiel Sträucher).

- Alpha Test bzw. Backface Culling: Diese Einstellung aktiviert den Alpha Test bzw. Backface Culling für alle Objekte der Szene. Backface Culling dient dazu, die Hinterseiten von Primitiven vom weiteren Rendering Prozess auszuschließen, und somit die Performance zu erhöhen. Der Alpha Test dient dazu, gewisse Fragmente vom weiteren Rendering Prozess auszuschließen. Dies findet Beispielsweise beim Rendering von Bäumen Einsatz, indem Fragmente mit einer gewissen Opacity (Durchsichtigkeit) verworfen werden.

- Mipmapping erlauben: Deaktivierung dieser Option unterdrückt MipMap-Filter für alle Texturen, der Objekte dieser Szene.

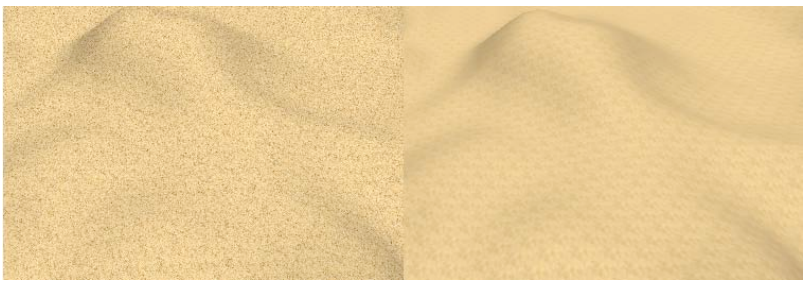


Abb.43: Diese Abbildung zeigt einen Screenshot des Leveleditors, wobei im linken Screenshot Mipmapping deaktiviert, und im rechten Mipmapping aktiviert ist.

6. Leveleditor

6.1. Allgemein

Bisher ist es möglich, einzelne Objekte zu transformieren und anzusprechen und diese Objekte in Objektsammlungen (AAX) zusammenzufassen, welche wiederum als Gesamtheit transformiert werden können. Theoretisch könnte die Spielwelt schon aufgebaut werden. Allerdings werden für eine Spielweltdefinition weitere Daten benötigt.

Darunter fallen:

- Spezielle Attribute für Rohstoffe wie Steine, Bäume und Sträucher. Darunter fallen zum Beispiel die Holzkapazität eines Baumes
- Exakte Positionierung von Objekte des Levels (jeglicher Vegetation) auf dem Gelände (Terrains).
- Benutzerdefinierte Texturierung des Terrains.
- Möglichkeit zum Festlegen begehbarer Stellen des Terrains
- Möglichkeit zum Festlegen von Levelspezifischen Story-Elementen wie Kamerafahrten in welchen die für jeden Level die dafür vorgesehene Geschichte erzählt wird.

Um Level Designern ihre Arbeit so weit als möglich zu erleichtern, wurde entschieden, ein weiteres Tool, den so genannten LevelEditor zu entwickeln.

Um dem Entwickler visuelles Feedback zu liefern, wurde entschieden eine 3D Ansicht ins Programm zu integrieren. Die Darstellung des Terrains erfordert allerdings genauere Betrachtung. Im Kapitel „Geländedarstellung“ wird auf diese Problematik gesondert eingegangen.

6.2. Grundaufbau des LevelEditors

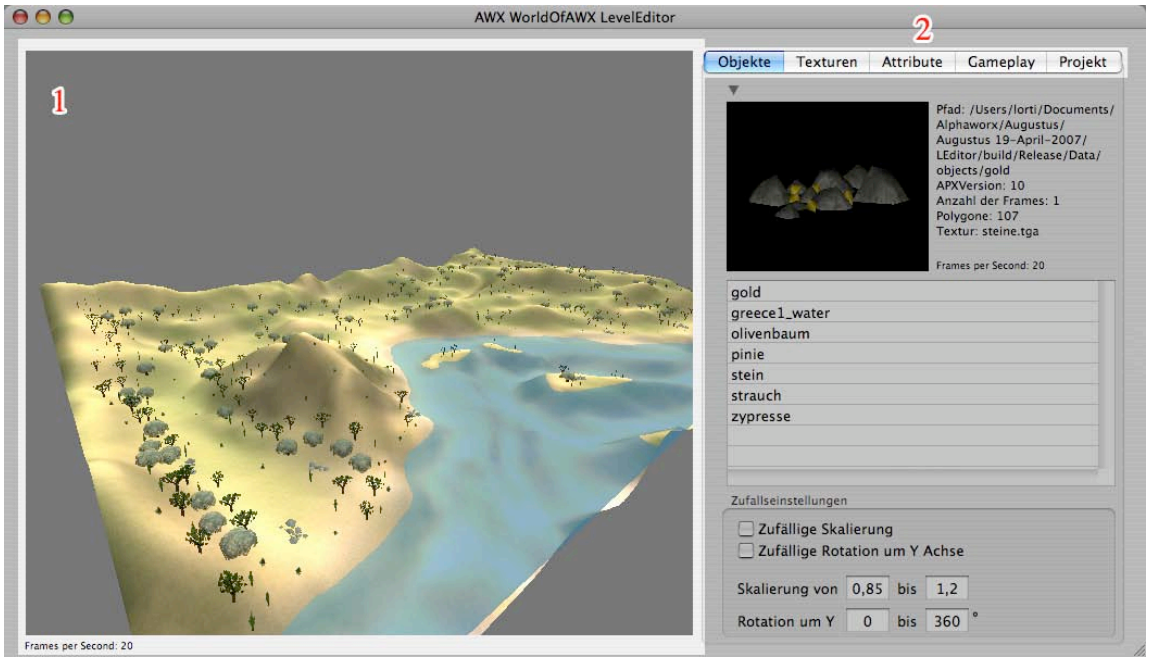


Abb.44: Screenshot des LevelEditors

Wie in (1) ersichtlich, wurde eine 3D Ansicht gewählt, um eine möglichst dem Spiel ähnelnde Vorschau zu bieten.

Um alle Funktionen möglichst Platz sparend zu platzieren und funktionell zu gliedern, wurde entschieden, die verschiedenen Funktionsgruppen in Tabs zu platzieren.

Die einzelnen Funktionsgruppen unter (2) sind:

- Objekte: Alle Funktionen, welche sich mit dem Platzieren und Anordnen von Objekten beschäftigen.
- Texturen: Alle Funktionen, welche sich mit dem Texturieren des Terrains beschäftigen.
- Attribute: Alle Funktionen, welche sich mit dem nachträglichen Ändern von Objekten beschäftigen.
- Gameplay: Alle Funktionen, welche sich mit der Platzierung und Anordnung von Gameplay-Objekten beschäftigen. Darunter fällt zum Beispiel die Positionierung von den Startpunkten der Spieler dieses Levels. Außerdem umfasst dieses Tab alle Funktionen, welche sich mit dem Markieren von für Einheiten (Soldaten, Arbeiter, Bürger) begehbarem Terrain beschäftigen.
- Projekt: Dieser Tab umfasst alle Funktionen, welche das Gesamtverhalten des Levels festlegen. Darunter fallen Levelbeschreibungen, Kamerafahrten und Untertitel.

6.3. Hinzufügen von Objekten im Terrain

Der Objekte Tab ermöglicht das Platzieren gewünschter Objekte bzw. Vegetation im Level. Die Wahl der Position wird über die Maus gewählt und anschließend platziert.

Der Objekte Tab enthält daher eine Liste der im Leveleditor vorhandenen Objekte und eine 3D Ansicht, welche das jeweilig ausgewählte Objekt visualisiert.

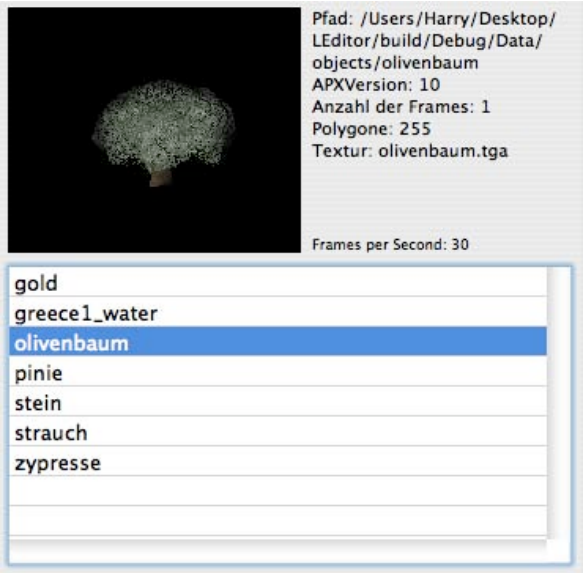


Abb.45: Diese Abbildung zeigt die Tabelle der verfügbaren Objekte und die 3D-Anischt dieses Objektes.

6.4. Texturieren des Terrains:

Für die Texturierung des Terrains steht dem Designer ein auswählbarer Pool von Texturen zur Verfügung. In einem Level sind die sich gleichzeitig im Pool befindlichen Texturen auf 4 begrenzt. Genauere technische Beschreibungen und Hintergründe können im „Geländedarstellung“ eingesehen werden.

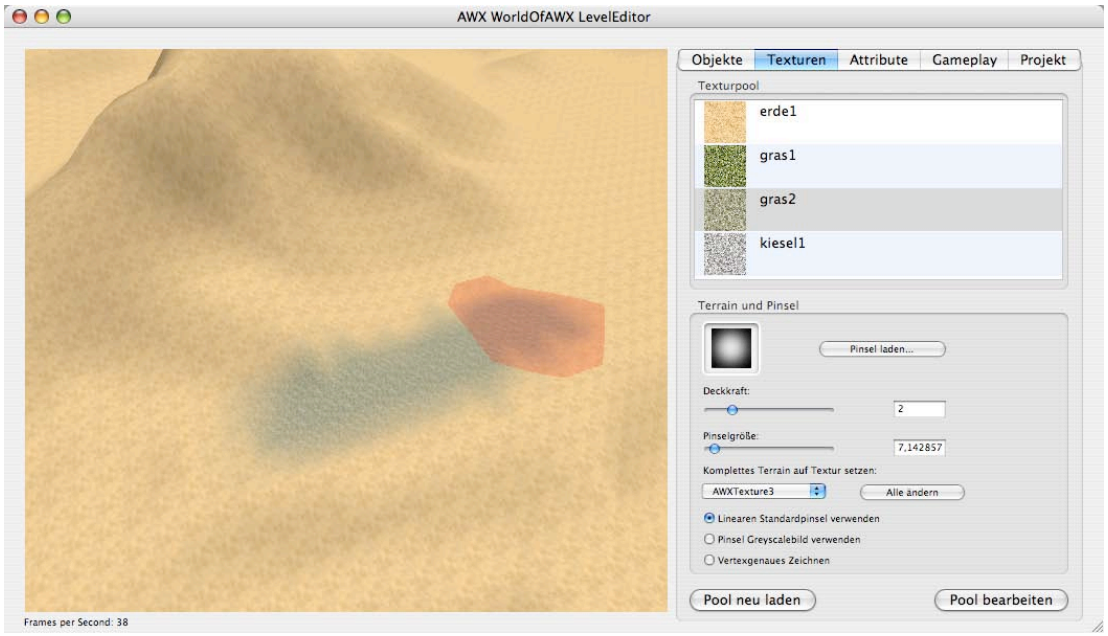


Abb.46: Dieses Bild zeigt den Texturiermodus des Terrains, wobei der Cursor in Rot dargestellt wird.

Der Texturiermodus wurde so entwickelt, dass der Benutzer die Möglichkeit hat, wirklich auf dem Terrain zu „malen“. Diese Art von Texturierung steht in Konkurrenz zur Texturierung im Bildbearbeitungsprogramm. Die verwendete Variante bietet jedoch den Vorteil, dass der Benutzer in Realtime die Änderung am Terrain sieht.

Dem Benutzer steht ein äußerst flexibler Pinsel zur Verfügung. Einstellbar sind:

- Größe des Pinsels
- Deckkraft des Pinsels
- Form des Pinsels

Der Texturpool ist auch nachträglich änderbar, da stets mit Texturnummern gearbeitet wird. Intern wird also nicht eine bestimmte Textur auf das Terrain projiziert, sondern nur das betreffende Terrainstück mit einer entsprechenden Zahl markiert.

Abb.47: Dieses Bild zeigt einen Screenshot des Fensters, in welchem der Texturpool festgelegt wird.

Das Auswählen des Pools:

Auf der Linken Spalte werden alle Texturen, die im LevelEditor-Verzeichnis unter Texturen vorliegt, angezeigt. Durch Bewegen dieser Texturen auf die rechte Spalte werden sie dem derzeitigen Pool angehängt. Die Größe dieses Pools ist auf 4 beschränkt, da unsere Zielhardware maximal 4 Texturlayer unterstützt.

6.5. Attribute

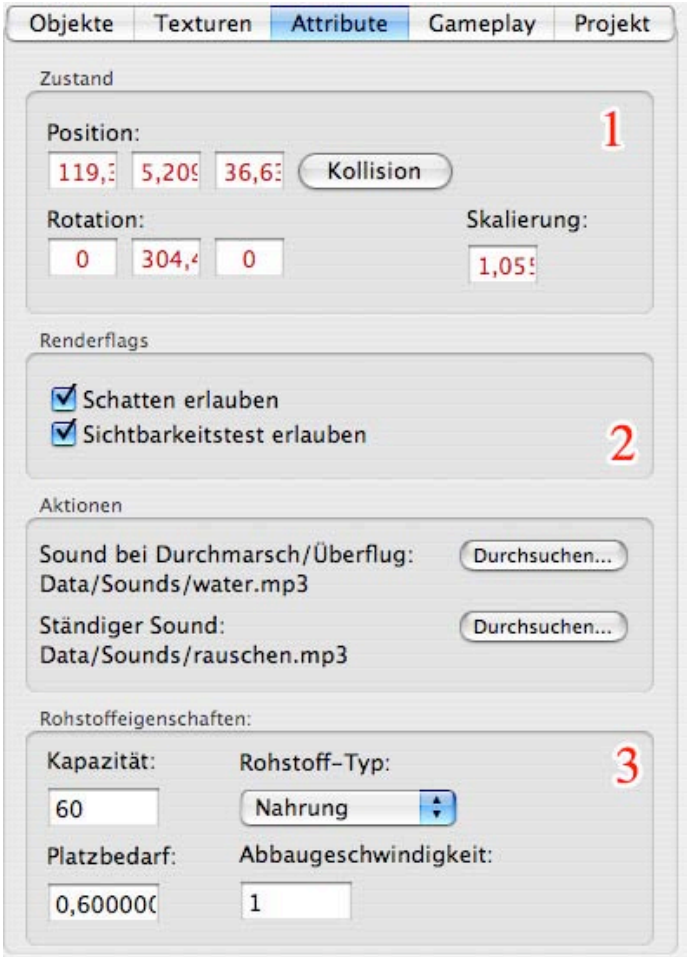


Abb.48: Dieses Bild zeigt den Attribute Tab

Die Funktionen von (1) beziehen sich auf bereits platzierte Objekte (und ausgewählte), deren Attribute man ändern will. Über den Kollision-Button ist es außerdem möglich, ein möglicherweise schwebendes Objekt auf dem Terrain zu platzieren. Die Einstellungen „Schatten erlauben“ und „Sichtbarkeitstests erlauben“ beziehen sich ebenfalls auf ausgewählte Objekte. Wenn „Schatten erlauben“ deaktiviert ist,

wird im Spiel jegliche Schattenberechnung deaktiviert. Ein klassisches Beispiel, wo diese Optionen notwendig sind, ist das Wasser. Die Wasseroberfläche sollte auf keinen Fall einen Schatten auf den Grund werfen, da Wasser im Spiel ohnehin bestimmten Shadern und Einstellungen unterliegt, um das Wasser möglichst realistisch darzustellen. Die in (3) einstellbaren Attribute richten sich ebenfalls an das derzeit ausgewählte Objekt. Diese Attribute beschreiben den Typ des Rohstoffes, falls dieses Objekt überhaupt dem Rohstoffabbau dienlich ist, die Rohstoffkapazität dieses konkreten Objektes, wie viel Platz eine Rohstoffeinheit im Rucksack eines Arbeiters benötigt und einen Faktor namens „Abbaugeschwindigkeit“, welcher den Abbau des Rohstoffes hemmen oder beschleunigen kann. Beispielsweise sollte Gold langsamer abgebaut werden als Holz.

6.6. Gameplay

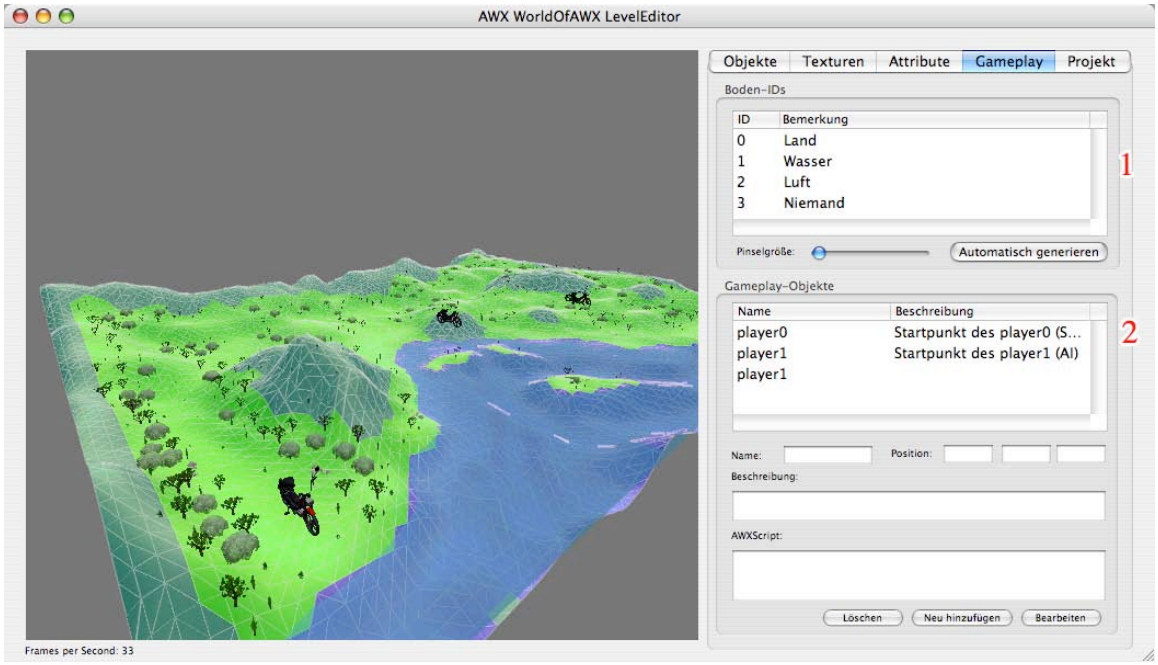


Abb.49: Dieses Bild zeigt einen Screenshot des Gameplay Tabs und der 3D-Ansicht

Im Spiel besteht die Möglichkeit die Steigung des unter Einheiten liegenden Terrains zu bestimmen, und daraus abzuleiten, ob diese Steigung für die Einheit zu bewältigen ist. Oft tritt jedoch der Fall auf, dass Designer Einheiten von schwieriger beschreibbarem Terrain fernhalten wollen. Um dies zu ermöglichen wurde entschieden ein System einzusetzen, welches pro Punkt des Terrains eine Identifikati-

onsnummer für die Art des Bodens speichert. Ist die Art des Bodens mit den Bodenarten der Einheit nicht kompatibel, so ist dieser Teil des Terrains für die jeweilige Einheit nicht begehbar. Um diese Bereiche festzulegen, wurde ein System wie jenes bei der Texturierung eingesetzt, um die Boden-Identifikationsnummern auf das Terrain zu malen. In 1 sind alle verfügbaren Bodenarten aufgelistet. Man kann die Bodenart und Pinselgröße auswählen mit denen man das Terrain bemalen möchte. Um auch hier Feedback an den Designer zu bieten, wurde entschieden, das Terrain in diesem Modus gesondert zu rendern. Und zwar wird das Terrain abhängig von der ausgewählten Bodenart eingefärbt.

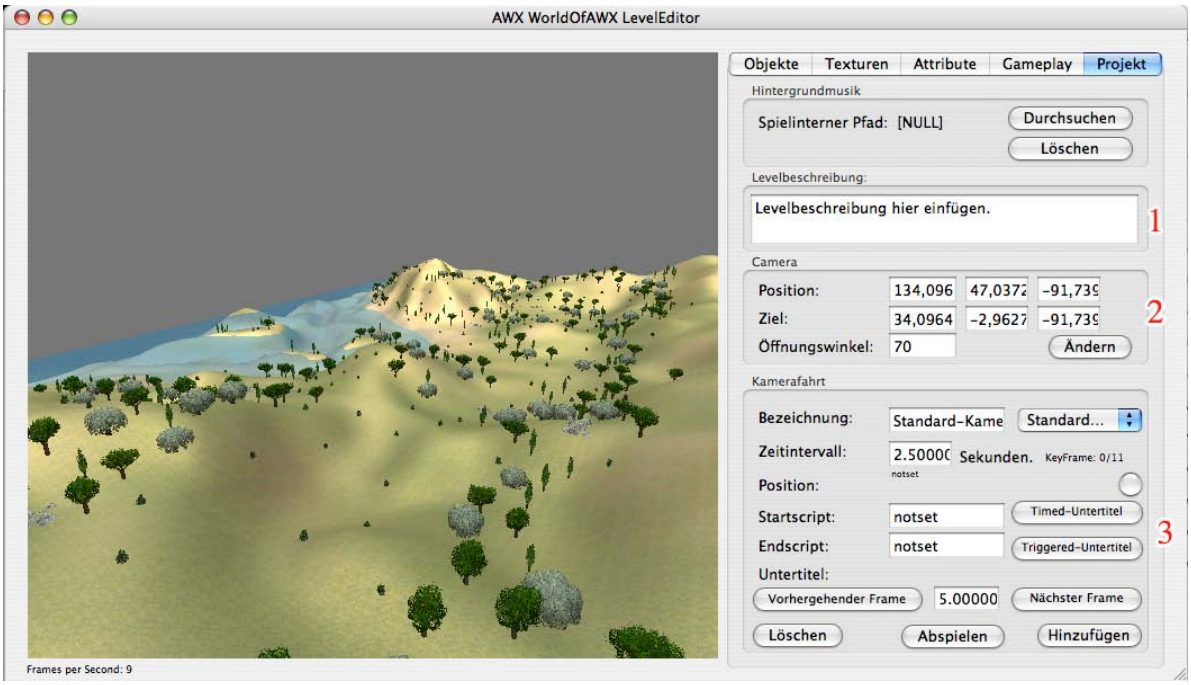


Abb.50: Dieses Bild den Leveleditor mit aktiviertem Projekt Tab

6.7. Projekt

Der „Projekt“-Tab umfasst einige globale Leveleinstellungen, wie zum Beispiel Levelbeschreibung und Hintergrundmusik. Weiters wird hier die Konfiguration von Storyelementen wie Kamerafahrten und Untertitel ermöglicht.

Die Kamerafahrt besteht aus einzelnen Schlüsselpositionen (Positionen zwischen denen die endgültige Kameraposition interpoliert wird). Zwischen diesen Schlüsselbildern (im Programm als Frame bezeichnet) kann beliebig gespult werden. Um Un-

tertitel passgenau zu diesen Framezeitpunkten aufzurufen, wurde die Möglichkeit geschaffen, jedem dieser Frames einen Untertiteltext zuzuweisen.

In vielen Situationen ist dies jedoch nicht erwünscht. Zum Beispiel sollte es möglich sein, unabhäng von der Kamerafahrt Untertitel anzuzeigen. Um dies zu ermöglichen, wurde entschieden einen weiteren UnteritelManager einzuführen, welcher sich außschließlich nach eingegebenen Zeiten richtet.

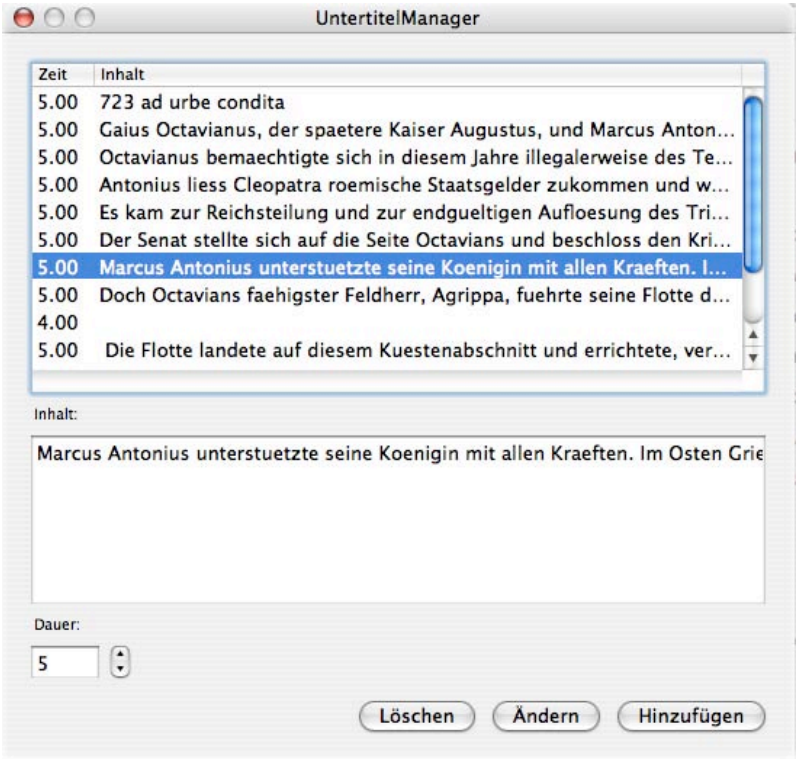


Abb.51: Dieses Bild zeigt den Untertitelmanager mit, wobei schon Untertitel eingetragen wurden.

Jedes Untertitelsegment ist somit einzeln anzusprechen, der Untertiteltext zu ändern, und die Zeitdauer festzulegen.

Geländedarstellung

1. Geometrie

Für die Darstellung eines Terrains gibt es den weit verbreiteten Ansatz von Heightmaps.

„Eine Heightmap ist ein 2-dimensionales Array, das Informationen über eine Landschaft enthält. Dazu wird einfach an jedem Knoten der entsprechende Höhenwert gespeichert.

Der Vorteil von Heightmaps ist, dass sie sehr leicht zu implementieren sind. Ein Nachteil ist jedoch, dass immer nur ein Höhenwert pro Knoten gespeichert werden kann und man so keine Felsvorsprünge oder Höhlen realisieren kann. Eine interessante Möglichkeit ist, Heightmaps mit Algorithmen wie Perlin-Noise zur Laufzeit zur erstellen.“ [33]

Es wurde allerdings entschieden, das Terrain wie jedes andere Objekt zu rendern, anstatt eine Lösung mittels Heightmaps zu entwickeln.

Einige Gründe dafür:

- Die Engine bot bereits die Möglichkeit beliebig detaillierte Objekte zu laden und zu rendern.
- Für den Grafiker ist eine direkte Modellierung des Terrains weniger aufwändig.
- Entwicklung einer heightmapbasierten Lösung hätte zusätzlichen Entwicklungsaufwand mit sich gezogen.

---

33 vgl. Wikipedia - die freie Enzyklopädie: Heightmap. Online im Internet: URL: http://wiki.delphigl.com/index.php/Heightmap 1. März

- Die niedrigere Flexibilität gegenüber einer heightmapbasierten Lösung hinsichtlich dynamischer Detailstufen wurde bewusst in Kauf genommen um Programmierressourcen zu sparen.

2. Texturierung

Bei der Texturierung von größeren Terrains stößt man schnell an die Grenzen der modernen Grafikhardware, da bei klassischer Texturierung (eine Textur, welche das gesamte Gelände bedeckt) die Texturgrößen in unverwaltbare Höhen schießen.

Eine solche Texturierung scheint angesichts der vielen Wiederholungen gewisser Bodenfarben ohnehin nicht sinnvoll.

Hier setzt das so genannte „Texture Splatting“ an:

„Texture Splatting ist eine Methode, um durch Texturen Terrains möglichst realistisch darzustellen. Die komplette Landschaft wird mit einigen gekachelten Grund-Texturen gepflastert. An unterschiedlichen Stellen bekommen die Texturen dabei eine unterschiedliche Gewichtung, so daß z.B. an einem Ort Gras wächst, an anderen hingegen Geröll herumliegt.“[34]

Um Informationen über die Gewichtungen der einzelnen Texturen zu speichern wird oft auf spezielle dafür vorgesehene Texturen zurückgegriffen, welche diese Informationen enthalten. In einigen Engines (z.B Battlefield 2) wird auf gewisse Werte (zum Beispiel Höhe der Punkte) des Terrains zurückgegriffen um diese Gewichtungen zu bestimmen.

Das folgende Bild sollte den Texturierprozess veranschaulichen:

---

34 Wikipedia - die frei Enzyklopädie. Texture Splatting. Online im Internet: URL: http://wiki.delphigl.com/index.php/Texture_Splatting 1. Mai 2007

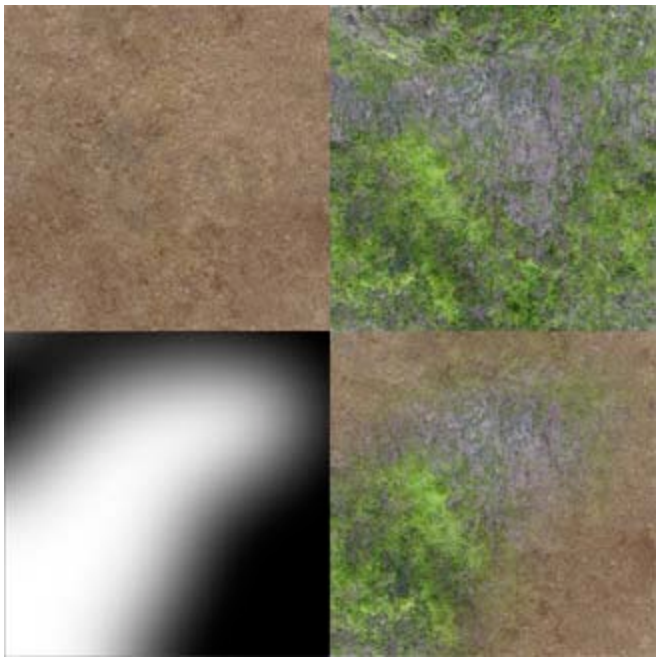


Abb.52: Dieses Bild zeigt verschiedene Texturen, welche für Texture-Splatting benützt werden könnten.
(http://wiki.delphigl.com/index.php/Bild:Splatting_Blending.jpg 1.Mai 2007)

Die Grundtexturen, welche für die Texturierung benutzt werden sind die oberen Teilbilder. Das Bild links stellt die Gewichtungstextur dar, wobei weiß die Benutzung der Stein/Gras Textur (Rechts oben) angibt. Schwarz hingegen bedeutet, es sollte nur die Erden Textur (Links oben) benutzt werden. Alle Werte dazwischen (Graustufen) geben somit das exakte Mischungsverhältnis an.

Rendering dieses Ansatzes (man beachte die mehrmaligen Aufrufe von RenderTerrain):

```
glColor4d(1, 1, 1, 0);
glBindTexture(GL.TEXTURE_2D, tex1);
RenderTerrain();
glEnable(GL.BLEND);
glColor4d(0, 0, 0, 1);
glBindTexture(GL.TEXTURE_2D, alpha2);
glBlendFunc(GL.ONE, GL.ONE);
RenderTerrain();
glColor4d(1, 1, 1, 0);
glBindTexture(GL.TEXTURE_2D, tex2);
glBlendFunc(GL.DST_ALPHA, GL.ONE_MINUS_DST_ALPHA);
RenderTerrain();
```

[vgl.:http://wiki.delphigl.com/index.php/Texture_Splatting]

„Zu beachten ist, daß die Alpha-Maps nicht wie im Beispiel die gleiche Größe zu haben brauchen, wie die Grund-Texturen, denn die Alpha Maps ziehen sich auf die ein oder andere Weise Úber die gesamte Karte (was nicht bedeuten muss, daß diese nicht in kleinere Abschnitte aufgeteilt ist und evtl. nur bei Bedarf erzeugt wird.), während die Grund-Texturen gekachelt immer wieder auftauchen.

Aus Geschwindigkeitsgründen sind jedoch meistens PixelShader im Einsatz: 3 Alphamaps werden zusammen in den Kanälen für die Farben rot, grün und blau gesteckt und an die erste Textur Einheit gebunden. Die 3 zugehörigen Grundtexturen werden dann an die restlichen Textur Einheiten gebunden und schon kann man mithilfe eines Pixel Shaders 3 Grundtexturen in einem Durchgang anzeigen. Hat man 8 Textureinheiten zur Verfügung, kann man diese natürlich mit einer weiteren zusammengebastelten Alphamap und 3 weiteren Grundtexturen belegen und kann so in einem Durchgang 6 Grundtexturen anzeigen. Ist das nicht genug rendert man die Landschat mehrfach und blendet alle weiteren Stufen dazu.„[35]

Limitierung durch Grafikhardware:

Unsere Zielhardware war zum Entwicklungszeitpunkt sehr begrenzend. Es handelte sich dabei um Radeon7200 Grafikkarten, welche nur 2 Texturlayer und keine Fragmentshader unterstützten. Ein Rendering mittels mehrerer Renderdurchgänge (wie oben beschrieben) hielten wir für wenig sinnvoll, also hoben wir die Zielhardware auf GeforceFX Niveau, welche auf verschiedensten Treibern mindestens 4 Texturlayer unterstützt. Dadurch erschloss sich uns die Möglichkeit Fragmentshader zu benutzen, da die GeforceFX diese bereits unterstützte.

Für den Grafikdesigner war es Anforderung, mindestens 4 verschiedene Bodentexturen zu benutzen, was aber bei einer Höchstzahl von 4 nicht umzusetzen ist, da die Gewichtungsinformationen keinen Platz mehr hätten. Ein Rendern mittels mehreren Renderdurchgängen (wie im obigen Beispiel) war aufgrund der hohen Polygonzahlen der Terrains ebenfalls nicht möglich.

---

35 Wikipedia - die freie Enzyklopädie. Texture Splatting. Online im Internet: URl: http://wiki.delphigl.com/index.php/Texture_Splatting 1. Mai 2007

Dieses Problem führte uns zum Ansatz einer per Vertex Gewichtung. Dieser Ansatz ist äußerst ungewöhnlich, jedoch praktikabel, da für Gewichtungsinformationen keine Texturlayer benötigt werden.

Es wird also schon im Leveleditor pro Vertex des Terrains ein Vektor mit 4 Komponenten gespeichert, welcher die Gewichtungen für alle 4 Bodentexturen enthält. Der größte Nachteil dieses Verfahrens ist, dass Übergänge nur zwischen Vertices, anstatt Pixel (wie bei Texturen) stattfinden können, und somit die Qualität des Überganges stark von der Zahl der Unterteilungen des Terrains abhängig ist.

Abb.53: Diese Abbildung zeigt einen Screenshot aus dem Strategiespiel „Age of Empires III“ (http://empireearth.free.fr/aoe-iii/age-of-empires-iii.jpg 1.Mai 2007 20 Uhr)

Hier sieht man eindrucksvoll die Qualität der Texturübergänge mittels Gewichtungstextur. Unsere Implementierung hingegen kann nur mit der folgenden Übergangsqualität aufwarten:

Abb.54: Dieses Bild zeigt einen Ausschnit aus Augustus und veranschaulicht das verwendete Texturierverfahren.

Der Rote und Blaue Punkt dient nur zur Erklärung. Der Vertex unter dem roten Punkt hat volle Zugehörigkeit (Gewichtung 1) zur dunkelgrünen Textur. Der Vertex unter dem blauen hat volle Zugehörigkeit zur hellgrünen Textur. Dazwischen wird die Gewichtung linear interpoliert. Die kleinstmögliche Einheit welche also eine andere Textur hat besteht aus 2 Dreicken. Mit höherer Dreieckzahl kann dieses Verhalten verbessert werden, jedoch bleibt die pixelbasierte Lösung wie in Age Of Empires III immer genauer.

Abb.55: Dieses Bild zeigt das entwickelte Verfahren im Einsatz in Augustus.

Die Umsetzung

Die Programmierung dieses Effektes liegt genau im Einsatzgebiet eines Vertex und Fragmentshader Sets (Zusammenspiel aus Vertex und Fragmentshader).

Um die Gewichtungen in die Shader zu übermitteln wurde auf das System der generischen Attribute zurückgegriffen. Es wird also neben Texturkoordinaten, Vertices und Normalen ein weiteres Attribut hinzugefügt. Das Hinzufügen eines sochen Attributes funktioniert nur in Verbindung mit Shadern.

Weitere Anforderung an das Terrainrendering ist der sogenannte „fog Of War“ (siehe Karte und Spielprinzip, Seite 9). Das Prinzip des Fog of War kommt in Strategiespielen oft zum Einsatz, wobei es nichts weiter bedeutet, als dass unbekannte Teile des Terrains als dunkel oder Schwarz dargestellt werden. Ob und wie viel sich ein Vertex im Fog Of War befindet wird auch mittels Vertex Attribute gesendet.

Das Rendering selbst, wurde mittels Vertex Buffer Objects realisiert. Dazu verweisen wir auf: http://developer.nvidia.com/attach/6427 und
http://oss.sgi.com/projects/ogl-sample/registry/ARB/vertex_buffer_object.txt

Nachdem wir das Verfahren ausprogrammiert haben, stießen wir auf schwerwiegende Performance Probleme. Nach eingehender Analyse (siehe http://ati.amd.com/developer/gdc/GDC2005_OpenGL_Performance.pdf) stellten wir fest, dass der Schwachpunkt bzw. die begrenzende Einheit der Rendering Pipeleine der Vertexprozessor ist. Dieser war zum damaligen Zeitpunkt allerdings nicht mehr weiter zu optimieren und so war es nahe liegend, nicht sichtbare Geometrie schon vor dem Vertexprozessor der Rendering Pipeline „auszuschließen“. Dies realisierten wir über Frustum Culling (http://www.flipcode.com/articles/article_frustumculling.shtml , http://www.lighthouse3d.com/opengl/viewfrustum/) , wobei wir das Terrain in kleinere Sektoren unterteilten und beim Rendering nur sichtbare Sektoren zum Rendern in Auftrag gaben.

Partikelsysteme

1. Allgemeines

Um die Spielwelt dynamischer erscheinen zu lassen, werden in modernen Spielen oft Partikelsysteme eingesetzt.

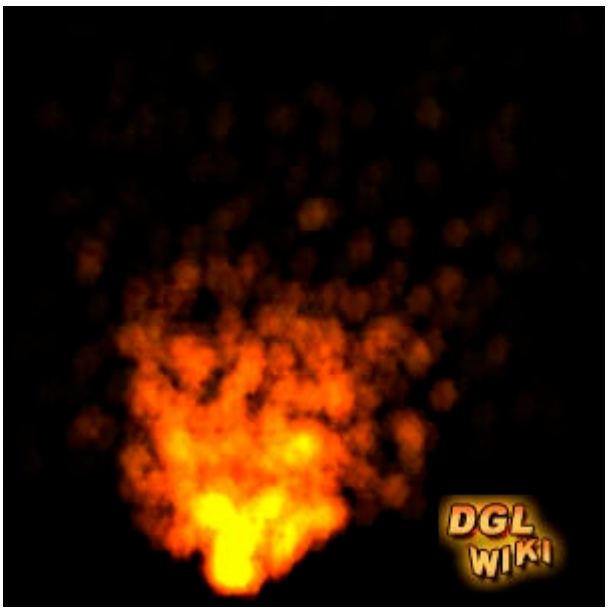


Abb.56: Dieses Bild zeigt ein Feuer Partikelsystem im Einsatz (http://wiki.delphigl.com/index.php/Bild:Partikel_feuer.png 1.Mai 2007 22:08)

„Partikelsysteme sind Grafikeffekte die aus vielen, meist primitiven, voneinander unabhängig berechneten Einzelteilen (Partikeln) bestehen. Bei den Effekten die man mit Partikelsystemen simuliert (oder erzielen möchte) handelt es sich üblicherweise um Objekte die nicht (oder nur schwer) durch ein Polygonmodell darstellbar wären.

Typische Anwendungsfälle sind:

- Wasser (vom Gebirgsbach bis zum Springbrunnen)
- Feuer
- Rauch
- Wolken
- Nebel
- Explosionen
- Feuerwerke
- Wetter (Schnee und Regen)

Also primär Objekte wo nicht nur die Oberfläche interessant ist, sondern das gesamte Volumen.“ [36]

Ein Partikelsystem ist also eine Ansammlung von Partikeln, die dynamisch erzeugt, gelöscht und bewegt werden.

2. Typische Eigenschaften eines Partikels

- Position: Derzeitige Position im Raum
- Größe: Größe des einzelnen Partikels
- Größe-Änderungsgeschwindigkeit: Ob sich der Partikel im Laufe seiner Lebenszeit skaliert, und wenn ja, in welchen Richtungen
- Alter und Lebenserwartung: Dauer, bis der Partikel gelöscht wird
- Farbe/Textur: Farbe des Partikels bzw. Textur des Pixels. Häufig werden die Farben animiert, um zum Beispiel alte Partikel in anderen Farben erscheinen zu lassen.

---

36 Wikipedia - die freie Enzyklopädie. Partikelsystem. Online im Internet: URL: http://wiki.delphigl.com/index.php/Partikelsysteme 1. Mai 2007 21:40

3. Rendering

Das Rendering der einzelnen Partikel wird meistens über Billboards realisiert. Das Rendering mittels Billboarding bedeutet, dass man Objekte immer in Richtung des Betrachters orientiert. Im Falle der Partikel , welche aus viereckigen Ebenen im Raum bestehen, werden die Partikel immer so gedreht, dass die gesamte Grundfläche der Partikel vom Betracher sichtbar ist.

Um das Partikelsystem realistischer erscheinen zu lassen werden für alle Werte, Zufallswerte benützt.

4. Editor

Um den auch bei der Erstellung der Partikelsysteme dem Designer möglichst die Arbeit zu erleichtern, wurde entschieden einen eigenen Editor für Partikelsysteme zu entwickeln.

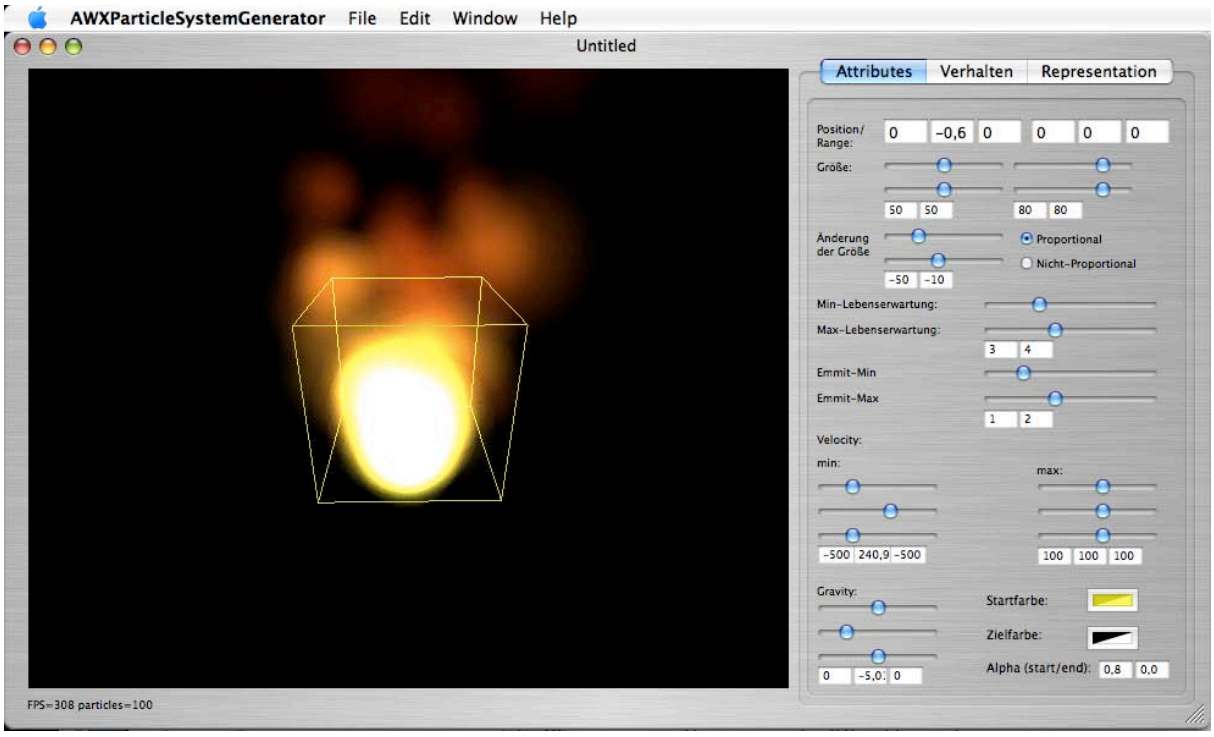


Abb.57: Dieses Bild zeigt einen Screenshot aus dem Partikelsystem-Editors aus Augustus

Die von diesem Editor erstellten Partikelsysteme können exportiert werden, und in Augustus angewendet werden, wobei die Position des Partikelsystems von Augustus gesteuert wird.

Grafik

1. Vorwort

"Die Computergrafik ist ein Teilbereich der Informatik, der sich mit der künstlichen Erzeugung von Bildern mit Hilfe von Computern beschäftigt. Mit derartigen Verfahren erzeugte Bilder nennt man auch Computergrafiken.

[...]

In der Unterhaltungs- und Werbebranche reicht das Spektrum der Computergrafik von der Erstellung einfacher Cartoons bis zu computergenerierten Werbespots oder ganzen Filmen mit vielen Spezialeffekten; in der Filmtechnik bezeichnet man gerenderte Bilder auch als Computer Generated Imagery (CGI). In modernen Computerspielen kommt Computergrafik systematisch zum Einsatz. Die Computergrafik ist heute nicht selten auch ein künstlerisches Ausdrucksmittel." 37

Bereits vor dem Start der Entwicklung von Augustus wurde im Projektteam an Applikationen mit 3D-Computergrafik gearbeitet. Das damals erworbene Wissen konnte in Augustus sinnvoll erweitert und eingesetzt werden. Technische Details der Darstellung wurden allerdings bereits in früheren Kapiteln erläutert. Im folgenden Kapitel wird auf die künstlerisch-technischen Aspekte bei der Erstellung der Computergrafiken für Augustus eingegangen. Im Anhang befinden sich Bilder der wichtigsten Modelle und deren Texturen.

---

37 Wikipedia - die frei Enzyklopädie. Computergrafik. Online im Internet: URL: http://de.wikipedia.org/w/index.php?title=Computergrafik&oldid=29995377, zugegriffen am 24. April 2007

2. Erstellen eines Gebäudes

Wie dem Kapitel "Spielprinzip" entnommen werden kann, benötigt man für ein Echtzeit-Strategiespiel wie Augustus zahlreiche Modelle von Gebäuden und Spielfiguren. Ein solches Modell durchläuft stets ähnliche Phasen bis zur Fertigstellung:

- Suchen von Referenzen und Inspiration
- Konzeptzeichnung des Modells
- Modellierung
- Texturkoordinaten festlegen
- Textur zeichnen
- Konvertierung mittels APXConvert

Bei den Figuren müssen außerdem ein Animationsgerüst und Animationen erstellt werden.

Abhängig vom gewünschten Ergebnis und der Komplexität eines Objektes kann die Arbeit an einem Modell zwischen wenigen Stunden und mehreren Wochen dauern. Das Wirtshaus, welches als Beispiel für dieses Kapitel ausgewählt wurde, beanspruchte ungefähr 16 Stunden, verteilt auf vier Wochen.

Abb.58: Fertiges 3D-Modell des Wirtshauses

Im folgenden werden nun die notwendigen Schritte erklärt, die schlussendlich zu einem fertigen Modell eines Wirtshauses führen.

2.1. Referenzen und Inspiration

Der Erste Schritt besteht aus dem Auffinden von Informationen und Ideen bezüglich des gewünschten Modells. Um einen glaubhaften Menschen zu modellieren bedarf es z.B. einigen Kenntnissen über Anatomie und vielen Referenzbildern. Wer ein Schiff oder einen Sportwagen auf dem Computer nachbilden will wird sich auf die Suche nach Blaupausen und technischen Zeichnungen begeben.

Da beim Wirtshaus keine besonderen Fachkenntnisse notwendig sind, kann man sich sofort dem Design widmen. Das Internet ist hierbei eine große Hilfe. So kann man z.B. mit der Google-Bildersuche oder Seiten wie deviantART viele Bilder und Zeichnungen von Wirtshäusern finden.

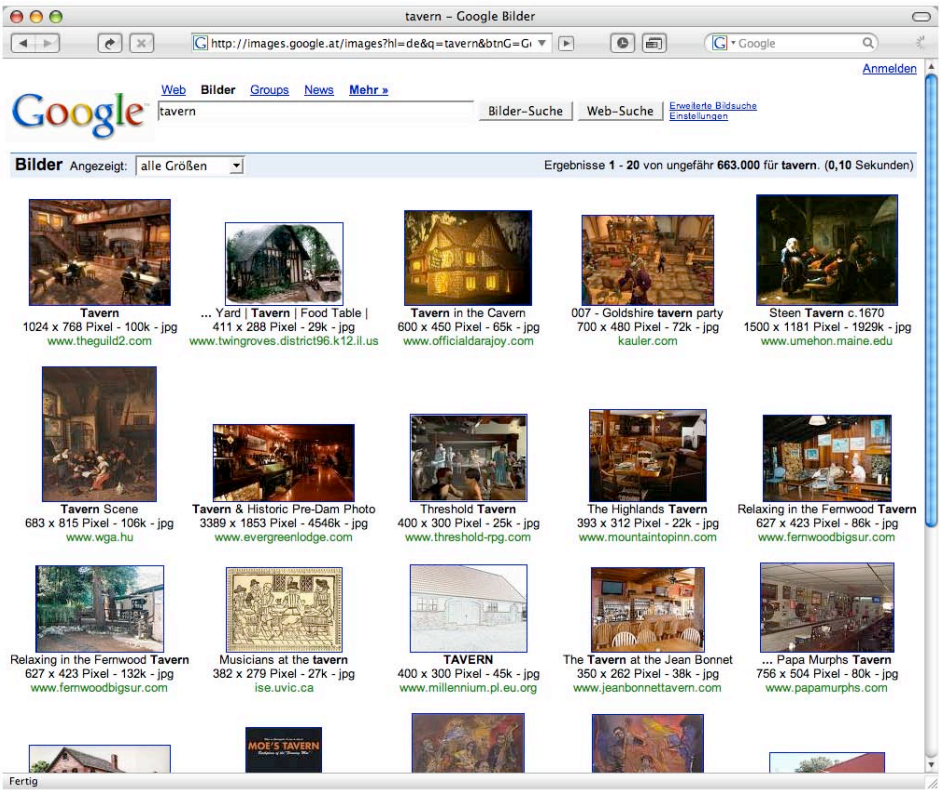


Abb.59: Google-Bildersuche mit dem Stichwort "tavern"

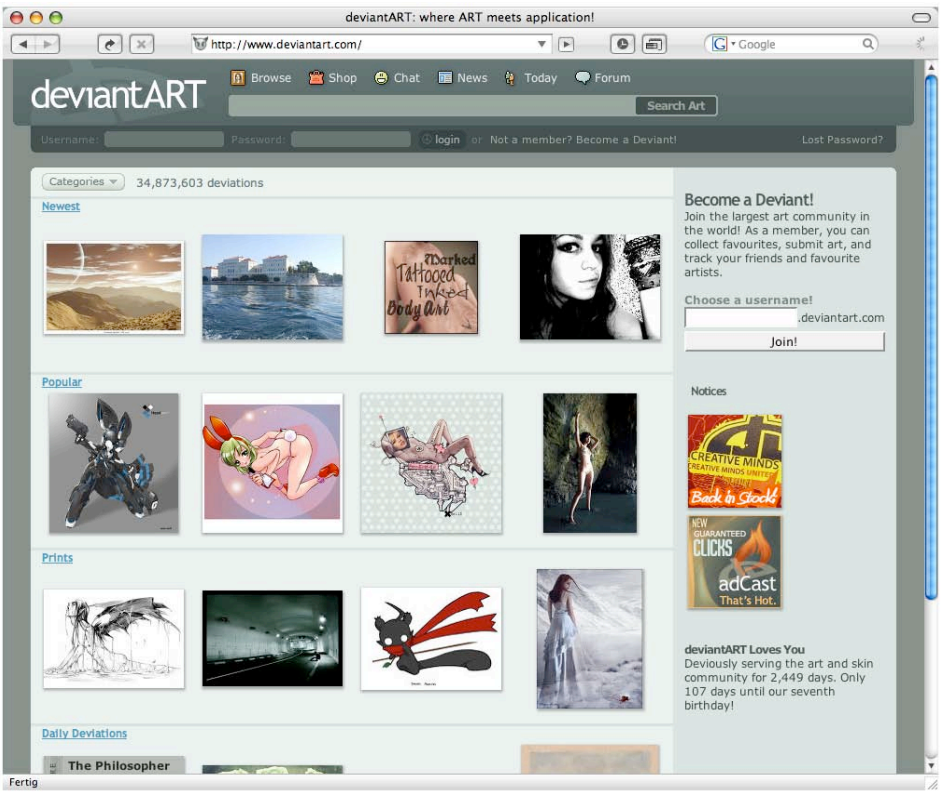


Abb.60: deviantArt, eine Community und Plattform für Künstler

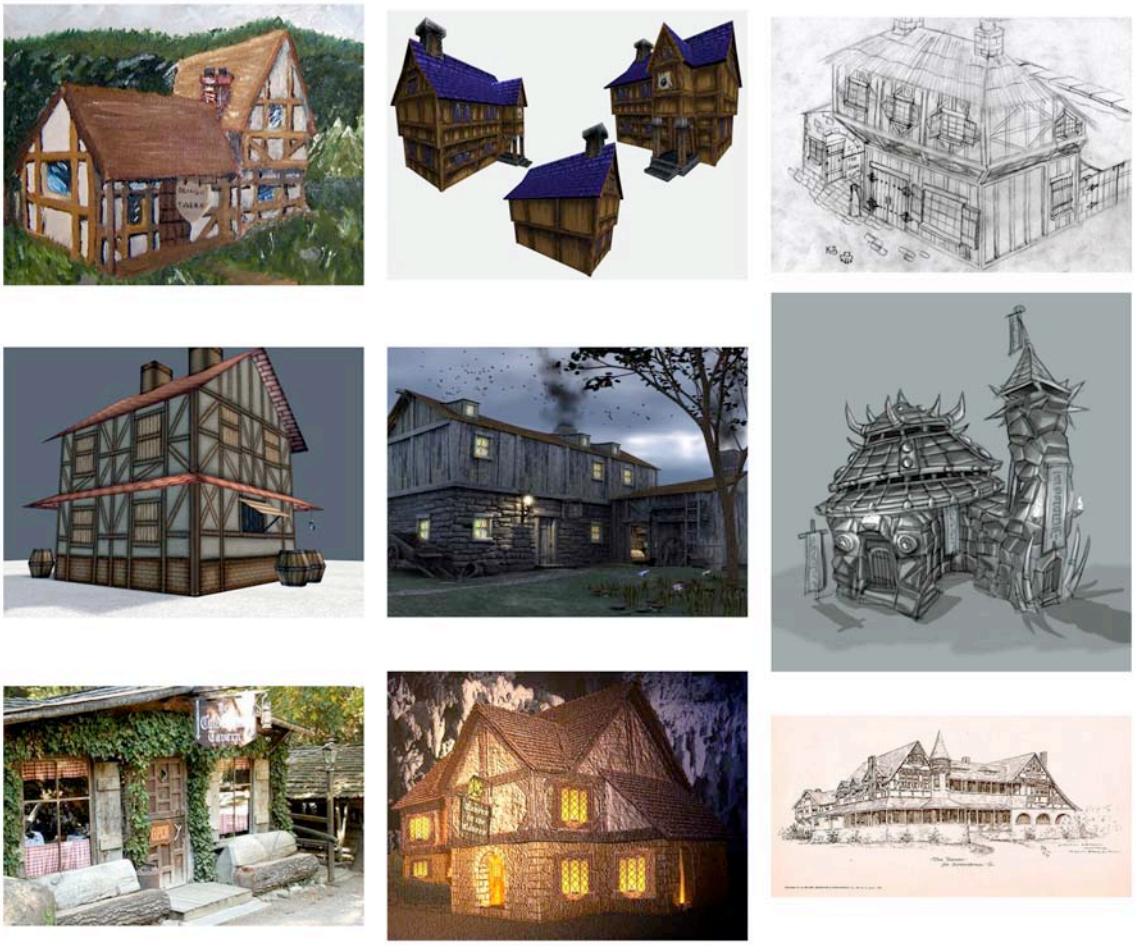


Abb.61: Bilder die dem Wirtshaus in Augustus als Inspiration dienten
(Alle Bilder stammen von http://www.deviantart.com und sind Eigentum der jeweiligen Künstler)

2.2. Konzeptzeichnung

Ein sehr wichtiger Schritt ist das Anfertigen von Konzeptzeichnungen. So können sich Projektmitarbeiter und der Designer selbst ein gutes Bild vom fertigen Produkt machen. Eine gute Konzeptzeichnung dient im weiteren Verlauf als Leitfaden und beugt Missverständnissen vor. Da man ein Konzept meist rasch zeichnen möchte, werden größtenteils klassische Werkzeuge wie Bleistift und Papier benutzt.

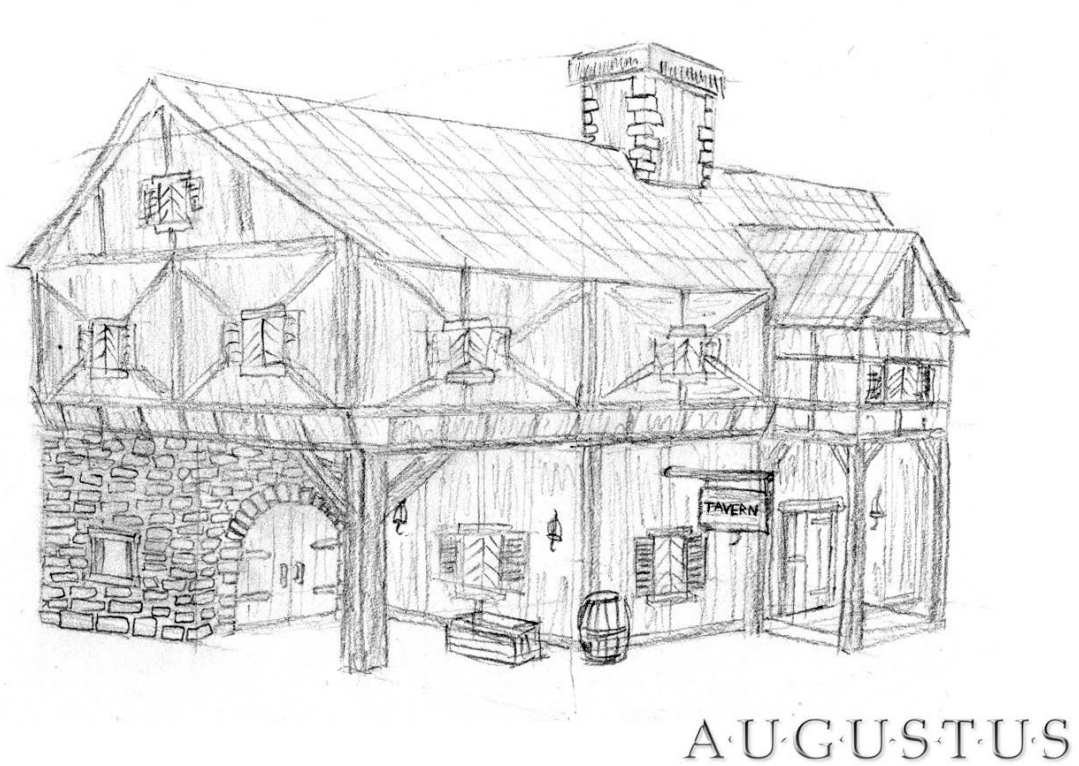


Abb.62: Eine von drei Zeichnungen eines Wirtshauses, welche als finales Konzept ausgewählt wurde

2.3. Grenzen der Echtzeitdarstellung

"In game development, the programmers who build the code engine typically perform tests to determine exactly how many polygons the engine can draw per second, on a reference machine that the game players are likely to have. Then the game designer sets the frame rate at which the game must refresh for smooth action, and the total engine performance number is divided by that frame rate to deliver the polygon budget. The Polygon budget is how many polygons can be on screen in each frame, before the game engine begins to choke and can't draw them all in time. After the polygon budget has been determined, the game designers list all the elements that need to be onscreen at one time, and divide the polygon budget up into a budget for each element." [38]

---

38 Inside Softimage 3D. Online im Internet. URL: http://www.digitalproducer.com/pages/soft_3d_p1.htm, zugegriffen am 24. April 2007

Nimmt man z.B. an, eine Engine könne 3 000 000 beleuchtete, texturierte Dreiecke pro Sekunde rendern, dann würde dies bedeuten man hätte ein Budget von 100 000 Dreiecken, wenn man möchte, dass die Engine mit 30 Bildern pro Sekunde läuft. Frühe Tests mit Grundbestandteilen der Engine von Augustus ergaben auf Testcomputern 2 000 000 bis 9 000 000 Dreiecke pro Sekunde.

Um die einzelnen Dreiecks-Budgets für Gebäude und Einheiten in Augustus zu ermitteln bediente man sich nur groben Schätzgrößen, da fest stand, dass sich die Grafikhardware im Laufe der Entwicklung rasch weiterentwickeln würde. Mit 100 000 Dreiecken pro Bild als Berechnungsgrundlage kam man auf:

- 5 000 Dreiecke für das Terrain
- 30*1000 Dreiecke für Gebäude
- 100*600 Dreiecke für Einheiten
- 5 000 Dreiecke für Pflanzen und sonstige Objekte

Zusätzlich zu den darstellbaren Dreiecken muss aber auch die Größe der Texturen und des Grafikspeichers betrachtet werden. Diese Rechnung ist leider noch etwas ungenauer als die für das Dreiecks-Budget, da neben den Pixeldaten viele weitere Daten im Grafikspeicher gespeichert werden. So ist die Größe eines Pixels auf dem Bildschirm im Speicher annäherungsweise:

32 Bit RGBA + 24 Bit Tiefenbuffer + 8 Bit Stencil Buffer = 64 Bit

Mit Double Buffering kommt man so auf 128 Bit pro Pixel. Bei einer Auflösung von 1024*768 errechnet sich dann eine Gesamtgröße von 12 MB. Würde man jetzt die Größe aller Daten im Grafikspeicher ausrechen so käme man bei 64 MB Speicher nur mehr auf ungefähr 30 MB für die Texturen. Diese könnte man anschließend wie folgt aufteilen:

- 11 verschiedene Gebäude mit einer 512*512 Textur á 1 MB
- 5 verschiedene Einheiten mit einer 256*256 Textur á 1 MB
- 4 Geländetexturen á 0,75 MB

- 5 Pflanzentexturen á 0,28 MB
- Texturen für die Benutzeroberfläche á ca. 8 MB

Zwar bringt das Überladen des Grafikspeichers keine erheblichen Probleme mit sich, doch ist mit einer weitaus höheren Performance zu rechnen, wenn alle für das Rendering notwendigen Daten im Grafikspeicher Platz finden.

Bei Augustus hatte man sich darauf geeinigt, das jeweils beste Verhältnis von Auflösung zu Qualität der Textur zu wählen, indem man die Objekte aus der Entfernung betrachtet, in der man sie als Spieler am öftesten sieht. Somit schwanken die Texturgrößen von 128*128 für einen Arbeiter bis zu 1024*1024 für den Marktplatz (Der Markt ist auch zugleich das Modell mit den meisten Dreiecken).

Beim Wirtshauses kommt man mit diesen Berechnungen auf ein Dreiecks-Budget von maximal 1000 Dreiecken und eine 1024*1024 Textur.

2.4. Modellierung

Der nächste Schritt besteht im Modellieren des Objekts. Da es einem gewöhnlichen Menschen nicht möglich ist, die Positionen und Größen der einzelnen Dreiecke per Hand einzugeben, benötigt man nun eine 3D-Modeling-Software wie Blender, Autodesk Maya, Maxon Cinema 4D oder NewTek Lightwave 3D, wobei es egal ist, welches der Programme man zum Modellieren verwendet. Die folgenden Abbildungen stammen aus Maya.

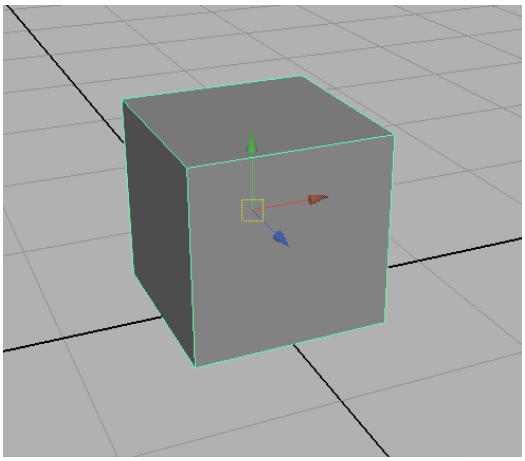


Abb.63: Das Wirtshaus wurde mit einem Würfel begonnen.

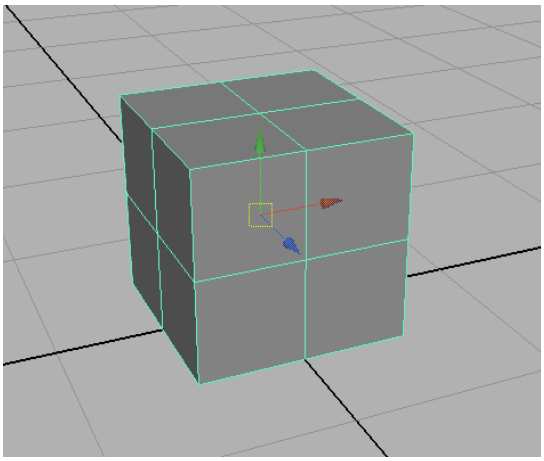


Abb.64: Da das Erdgeschoss des Hauses nicht quaderförmig ist, wird der Würfel weiter unterteilt. Dadurch kann später ein Stück unten ausgeschnitten werden.

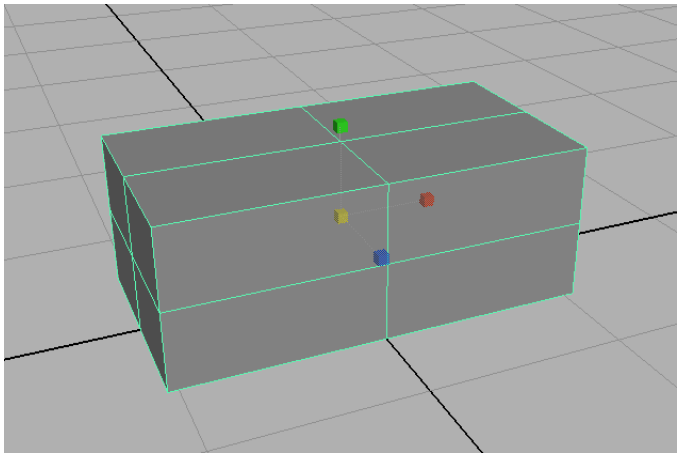


Abb.65: Der Würfel wird skaliert um den Dimensionen des Hauses auf der Konzeptzeichnung zu entsprechen.

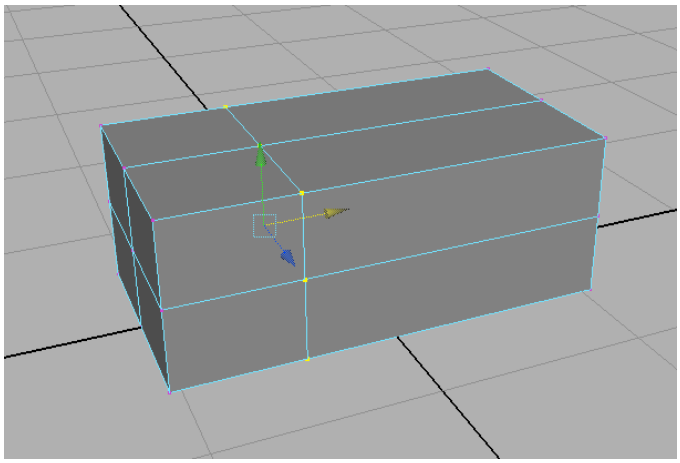


Abb.66: Nun werden die mittleren Vertices verschoben, damit der Ausschnitt an der richtigen Position ist.

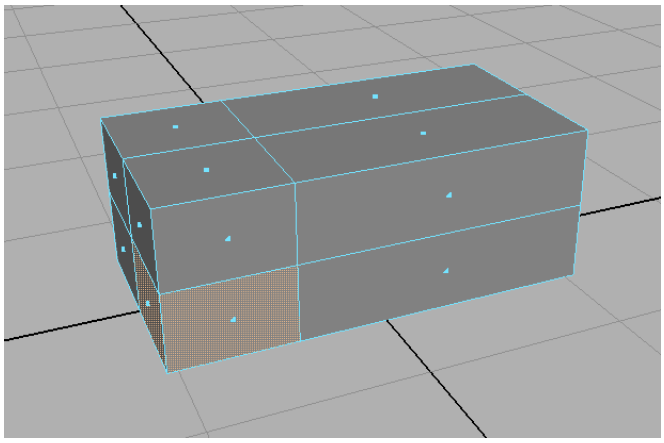


Abb.67: Die zwei markierten Flächen werden gelöscht.

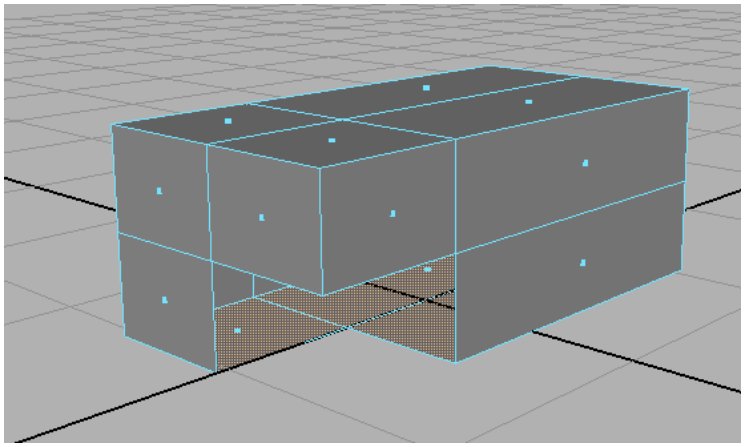


Abb.68: Ebenso wird der Boden des Objektes gelöscht, da man diesen im Spiel nie gesehen hätte.

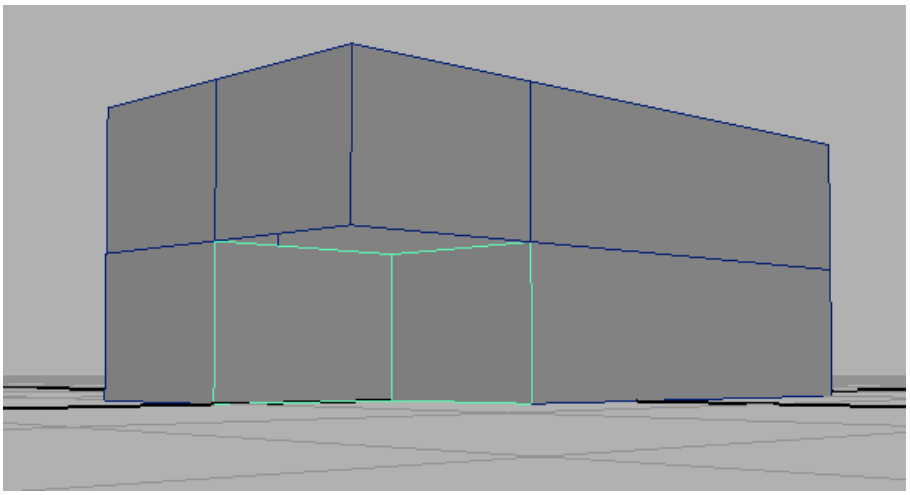


Abb.69: Zwei Polygone bilden die neuen Wände. Ein drittes Polygon für die Decke muss nicht erstellt werden, da man dieses wieder nicht sehen würde.

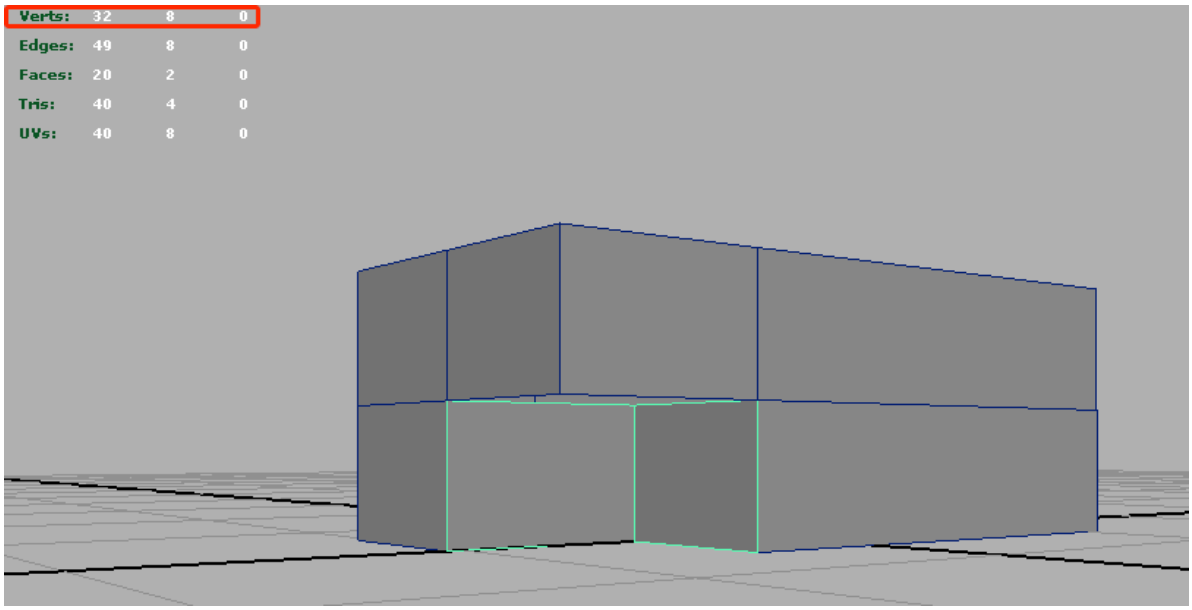


Abb.70: Es befinden sich nun drei Objekte in der Szene. Das bedeutet, dass sechs Vertices doppelt vorhanden sind. In der linken oberen Ecke werden 32 Vertices angezeigt.

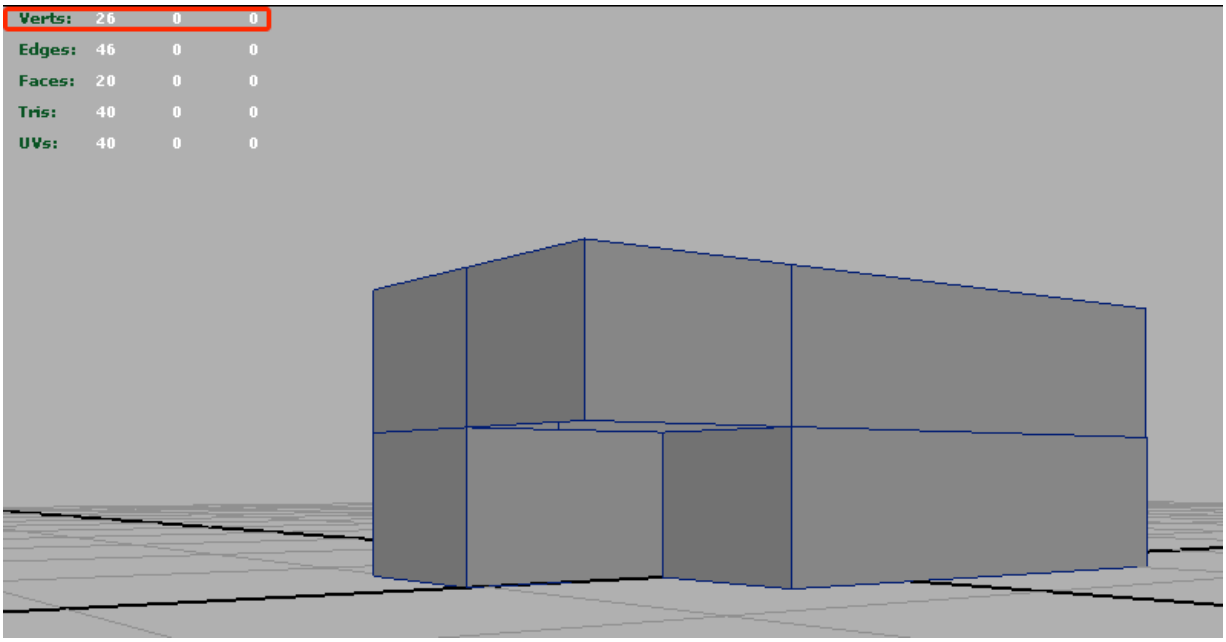


Abb.71: Verbindet man die drei Objekte zu einem einzigen verringert sich die Vertexzahl auf 26 und die Kanten von 49 auf 46.

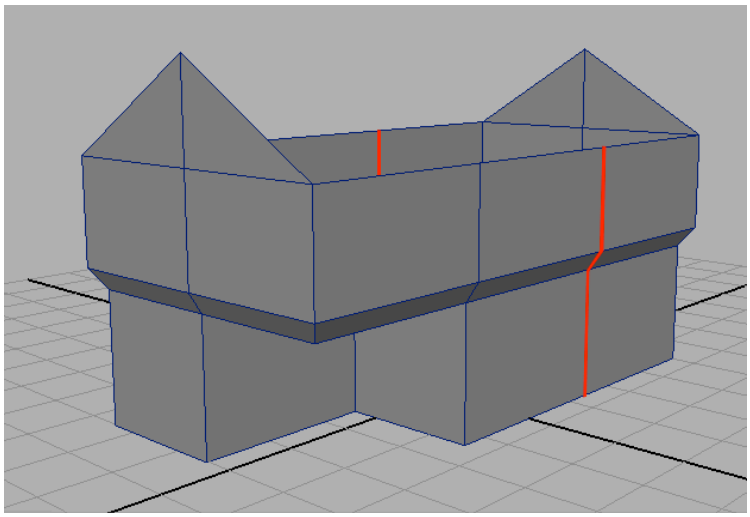


Abb.72: Mit einigen Schnitten und Polygonen kommt man schließlich zu diesem Objekt. Der rot markierte Schnitt hilft uns später in der Textur Platz zu sparen.

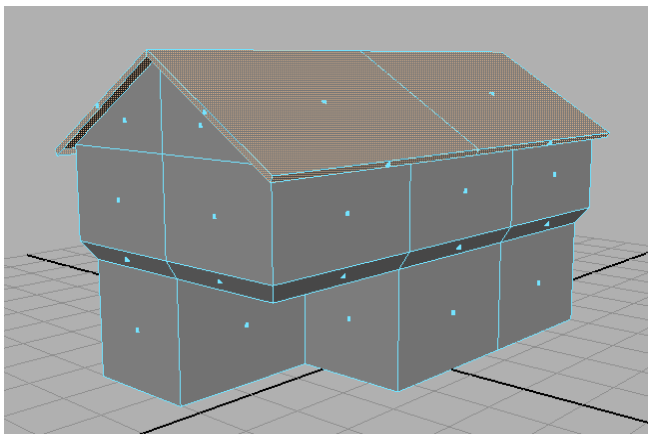


Abb.73: Das Dach ist ebenfalls in der Mitte geteilt. Später wird dieselbe Textur für beide Hälften benutzt.

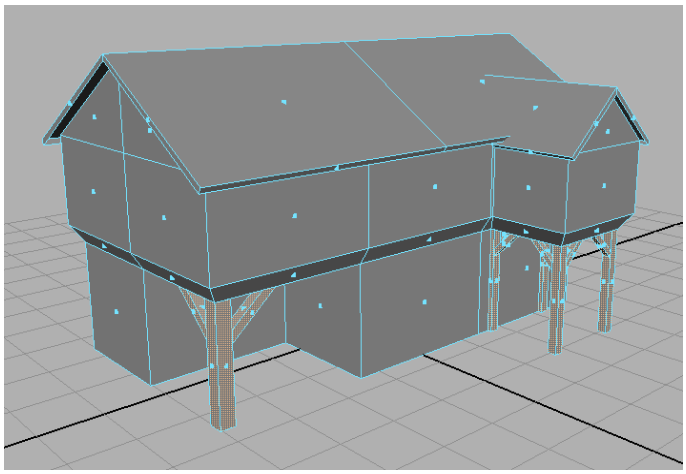


Abb.74: Nun wird begonnen die noch fehlenden Details anzufertigen.

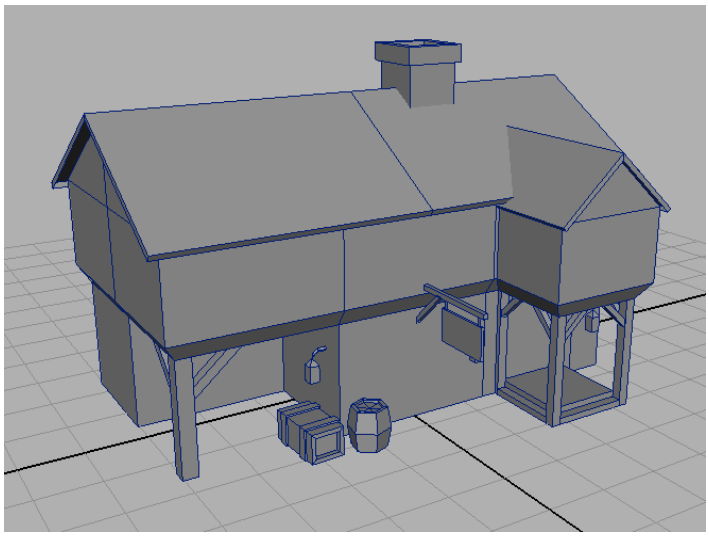


Abb.75: So sieht die Vorderseite des fertigen Wirtshauses aus.

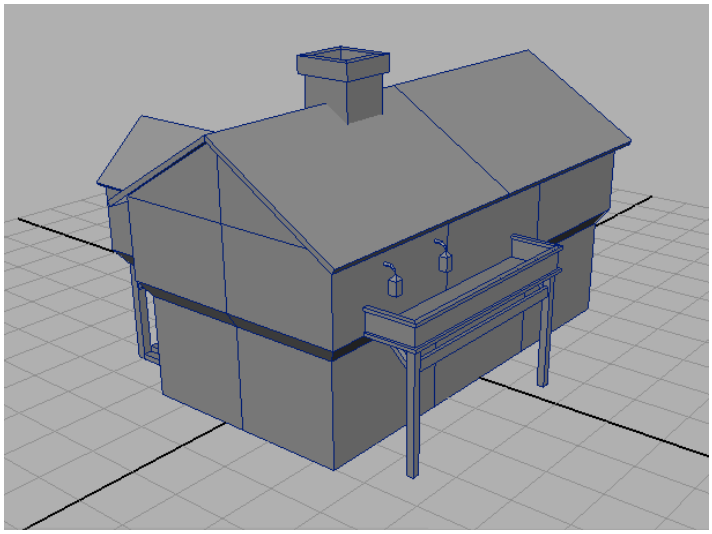


Abb.76: Auf der Rückseite befindet sich ein kleiner Balkon.

Das fertige Modell hat 699 Dreiecke. Es liegt also weit unter dem Maximum von 1000 Dreiecken. Jetzt wird noch die Building History gelöscht und die Datei bereinigt. Dazu kann man z.B. das Modell als .obj exportieren und in einer neuen Datei wieder importieren. Das mag übertrieben erscheinen, jedoch erkennt man mit dieser Methode etwaige Fehler in der Geometrie relativ rasch.

2.5. Texturkoordinaten

Bevor die Textur gezeichnet werden kann, muss noch festgelegt werden wie diese auf das Objekt projiziert wird. Dies geschieht mittels Texturkoordinaten. Jedem Vertex des Modells wird ein Punkt in der Texturebene zugewiesen.

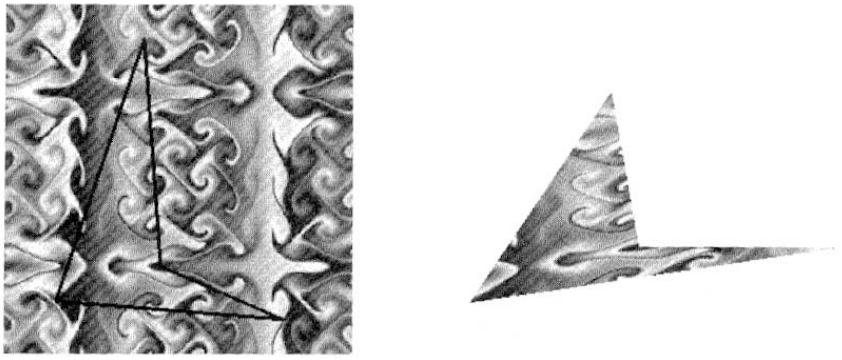


Abb.77: Ein texturiertes Polygon und dessen Texturkoordinaten
(Shreiner Dave, Woo Mason, Neider Jackie, Davis Tom: OpenGL Programming Guide, 4. Auflage 2003, S. 361)

"In der Regel gibt es für ein 3D-Modell oder eine 3D-Szenerie mehrere Bilder, die als Texturen verwendet werden. Das 3D-Modell besteht in der Regel aus Polygonen. Man kann nun jedem Polygon eine Textur zuweisen (natürlich können mehrere Polygone die selbe Textur verwenden) und jedem Eckpunkt des Polygons zwei Texturkoordinaten, die angeben, welchem Punkt in der zweidimensionalen Textur der Eckpunkt des Polygons entspricht. Um die Position innerhalb einer Textur zu bestimmen, hat sich ein Koordinatensystem eingebürgert, in dem (0,0) die linke untere Ecke und (1,1) die rechte obere Ecke der Textur bezeichnet; die beiden anderen Ecken sind naturgemäß (1,0) und (0,1). Die beiden Koordinaten werden meist u und v genannt und sind in der Regel Fließkommazahlen. [...]

In 3D-Modellen mit vielen Polygonen wird oft eine einzige Textur für das ganze Modell verwendet, sodass jeder Punkt des Modells nur einen Satz Texturkoordinaten (und nicht unterschiedliche Texturkoordinaten für die verschiedenen Polygone, die

diesen Punkt verwenden) hat, weil dieses Format für hardwarebeschleunigte 3D-Grafik und auch für den Designer des 3D-Modells besonders günstig ist." [39]

In Maya hilft einem beim Erstellen der Texturkoordinaten der UV Texture Editor in dem man die UV-Koordinaten bewegen, skalieren, rotieren und anordnen kann. In manchen Programmen gibt es spezielle Algorithmen mit denen die Texturkoordinaten automatisch generiert werden können.

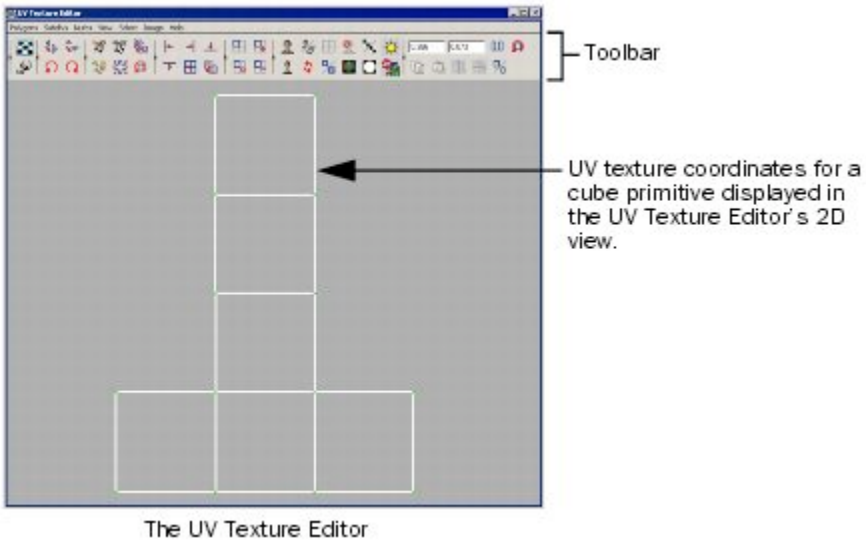


Abb.78: Der UV Texture Editor von Maya

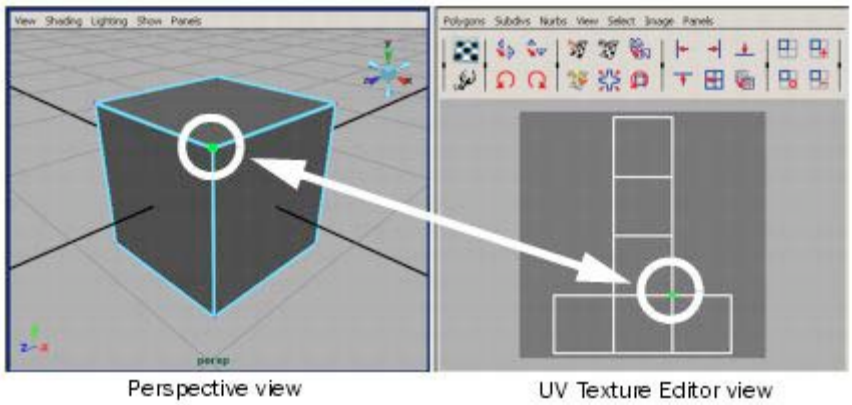


Abb.79: Ein Eckpunkt des Modells und dessen entsprechender Punkt im UV Texture Editor

39 Wikipedia - die freie Enzyklopädie. Texture Mapping. Online im Internet: URL: http://de.wikipedia.org/w/index.php?title=Texture_Mapping&oldid=30674984, zugegriffen am 24. April 2007

Jede Fläche des Wirtshauses benötigt nun möglichst unverzerrte Texturkoordinaten. Dies ist ein langwieriger und aufwändiger Prozess, da viele kleine Teilflächen im UV-Koordinatensystem Platz finden müssen und die Anordnung der einzelnen Flächen wichtig ist um möglichst keinen Platz auf der Textur zu verschwenden. Große ungenutzte Flächen auf der Textur bedeuten unter anderem eine überflüssige Belastung für den Grafikspeicher.

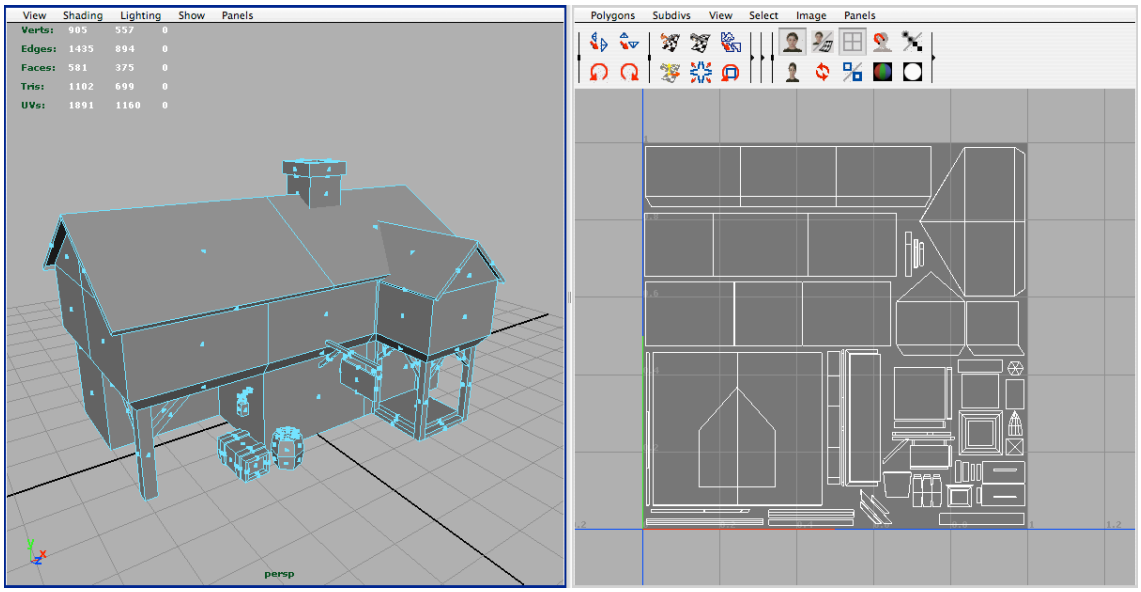


Abb.80: Links das Wirtshaus-Modell, rechts dessen fertige UV-Koordinaten

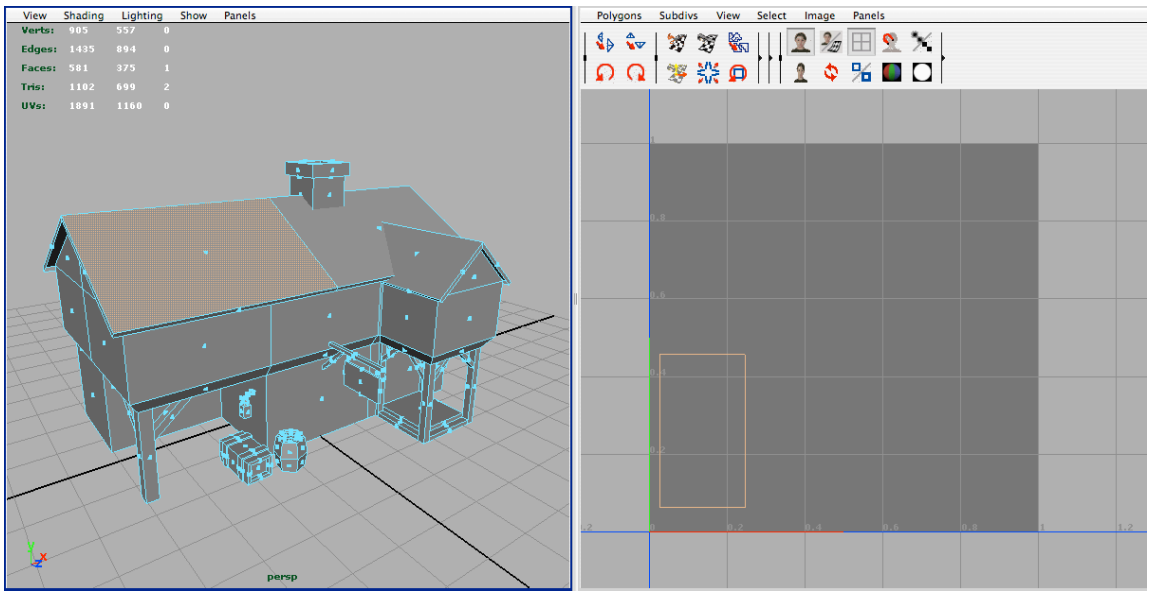


Abb.81: Position einer Fläche im Raum und auf der Texturebene

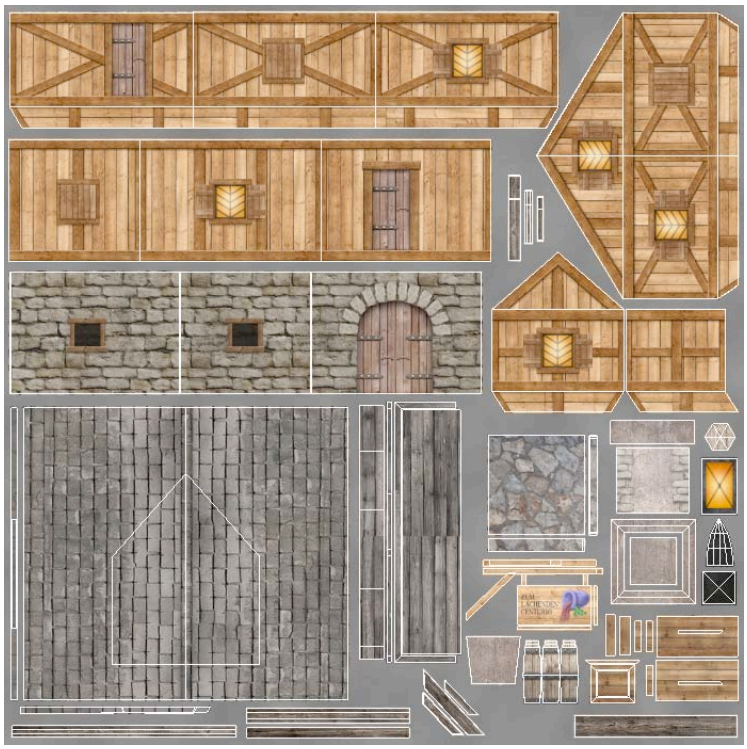


Abb.82: Die fertige Textur mit den dazugehörenden Texturkoordinaten

Schon beim Positionieren der UV-Koordinaten wurde bedacht, welche Texturteile man öfter verwenden könne. So wurde fast jeder Teil der Holzwände im ersten Stock zwei mal verwendet und die Steinwand auf der Rückseite des Hauses besteht aus drei identischen Texturteilen. Auch das Dach des Hauses wird doppelt verwendet. Zu beachten galt außerdem, möglichst wenig Fläche der Textur zu verschwenden. Jeder graue Bereich in der obigen Abbildung ist im Grunde verschwendeter Texturplatz.

2.6. Die Textur

Das Zeichnen der Textur ist in vielen Fällen mit einem größeren Aufwand verbunden als das Modellieren und Festlegen der Texturkoordinaten. Um die Arbeit einigermaßen zu erleichtern und zu beschleunigen wird oft auf bereits vorhandene Texturen oder auf Fotos zurückgegriffen. Es gibt jedoch auch vollständig gemalte Texturen, d.h. keine Fotos oder andere Texturen wurden in der neuen Textur verarbeitet. Mit moderner Malsoftware und einem Grafiktablett ist es durchaus möglich, Texturen komfortabel am Computer zu malen. Die Texturen von Augustus sind teilweise

gemalt, jedoch sind die meisten mithilfe von Fotos entstanden, da z.B. das Malen von realistischem Gras oder Holz sehr zeitraubend ist.

Bevor auf das eigentliche Anfertigen der Wirtshaustextur eingegangen wird, wird nun erklärt, wie man an geeignete Fotos für Texturen gelangt, und wie diese vor der Verwendung bearbeitet werden müssen.

2.6.1. Die Digitalkamera als Quelle

Die erste Möglichkeit ist, alle Fotos selbst zu schießen. Dazu benötigt man eine einigermaßen gute Kamera und sollte mobil sein. Es hilft auch, sich zuerst eine Liste aller benötigten Texturen anzulegen, obwohl es häufiger der Fall ist, dass man zu viele anstatt zu wenige Fotos schießt. Auch muss man sich auf verwunderte Blicke von Passanten gefasst machen, wenn man voller Begeisterung Kanaldeckel und dutzende Mauern fotografiert.

Bei der Aufnahme selbst, muss man einige Erfordernisse von Texturen beachten. So ist es meist ratsam, seine Fotos nicht bei starkem Sonnenlicht an einem wolkenlosen Tag zu schießen. Direktes Sonnenlicht wirft dunkle, harte Schatten, die man auf Texturen nicht vorfinden möchte. Deshalb, da man beim Einrichten einer 3D-Szene so viele Freiheiten wie möglich haben möchte und vorbeleuchtete Texturen einem eine gewisse Positionierung von Lichtern auferlegen, um nicht sonderbar zu wirken. Ein wolkiger Tag, vielleicht sogar mit leichtem Nebel, ist geeigneter, da das Sonnenlicht von den Wolken verstreut wird und weichere Schatten erzeugt.

Auch ist auf ein Blitzlicht zu verzichten da dieses unerwünschte Reflektionen auf der Oberfläche erzeugt. Um eine Verzerrung des Bildes zu vermeiden sollte man nicht mit Weitwinkelobjektiven arbeiten und möglicherweise die Oberfläche aus einiger Entfernung mittels Zoom fotografieren. [40]

---

40 vgl. Phillipp Strahl: Photographing for Textures. Online im Internet: http://www.highend3d.com/maya/tutorials/general/256-1.html [Stand 26.4.2007]

Abb.83: Einige im Vorfeld von Augustus entstandene Fotografien

2.6.2. Das Internet als Quelle

Aus Kosten-, Zeit- oder sonstigen Gründen kann natürlich auch auf das selbstständige Fotografieren verzichtet werden. Über das Internet oder Sammlungen auf CDs und DVDs lassen sich Unmengen an Fotografien zusammentragen. Viele Seiten im Internet stellen kostenlos Texturen zum Download bereit und erlauben meist auch eine kommerzielle Nutzung.

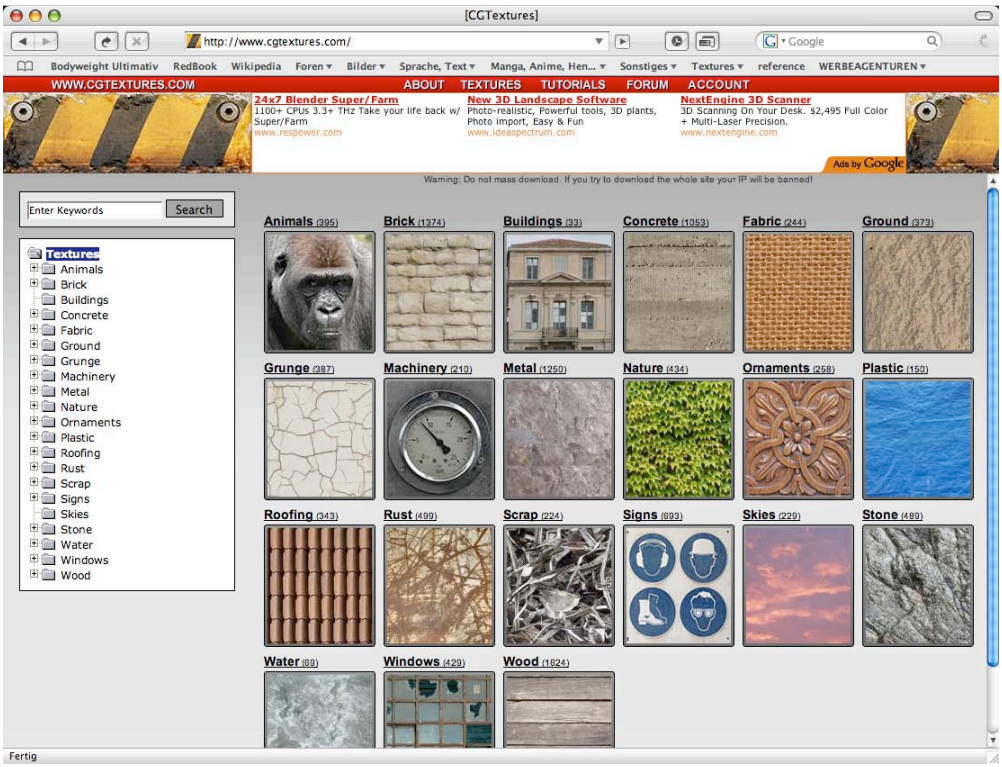


Abb.84: CGTextures stellt Fotografien zur Weiterverarbeitung in Texturen zur Verfügung.

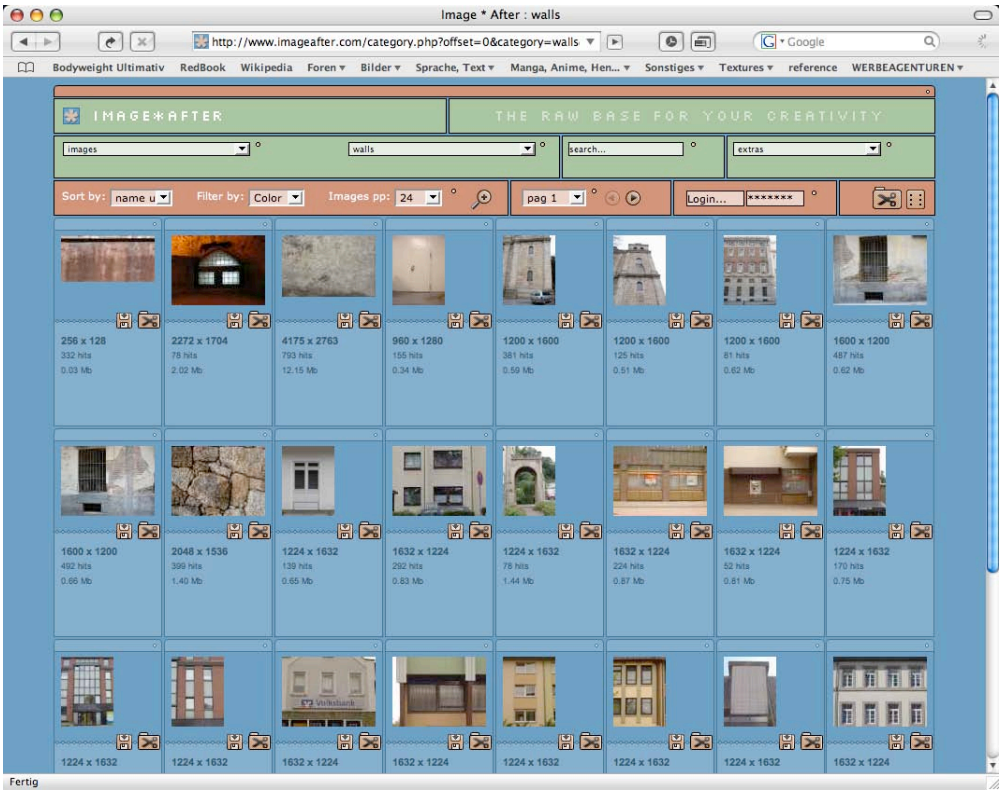


*Abb.85: image*after ist eine ebenfalls eine große Datenbank mit Texturen und Fotografien.*

2.6.3. Bearbeitung der Fotografien in Adobe Photoshop

Hat man genügend Fotos und Quellmaterial zusammengetragen kann man dieses natürlich nicht ohne weiteres als Textur verwenden. Einige Schritte sind erforderlich um z.B. eine in X- und Y-Richtung wiederholbare Grastextur für das Terrain zu bekommen. Auch sind meist der Kontrast und die Farbwerte der Fotografien nicht perfekt geeignet.

Im folgenden befindet sich ein Beispiel, wie man von einem Foto einer Mauer auf eine in X-Richtung wiederholbare Textur kommt, sodass man schließlich beliebig lange Mauern mit der Textur überziehen kann.

Abb.86: So sieht das Originalbild aus dem Internet aus. Würde man dieses Bild unbearbeitet als Textur benützen, würden auch unaufmerksame Spieler sehr starke Kanten zwischen den Wiederholungen bemerken.

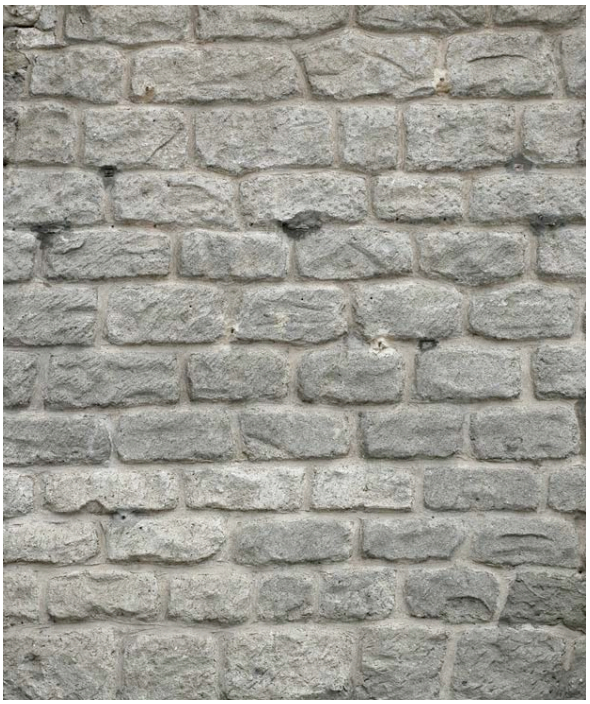


Abb.87: Im ersten Abschnitt werden unbrauchbare Bildteile, wie der schwarze Rand, einfach weggeschnitten.

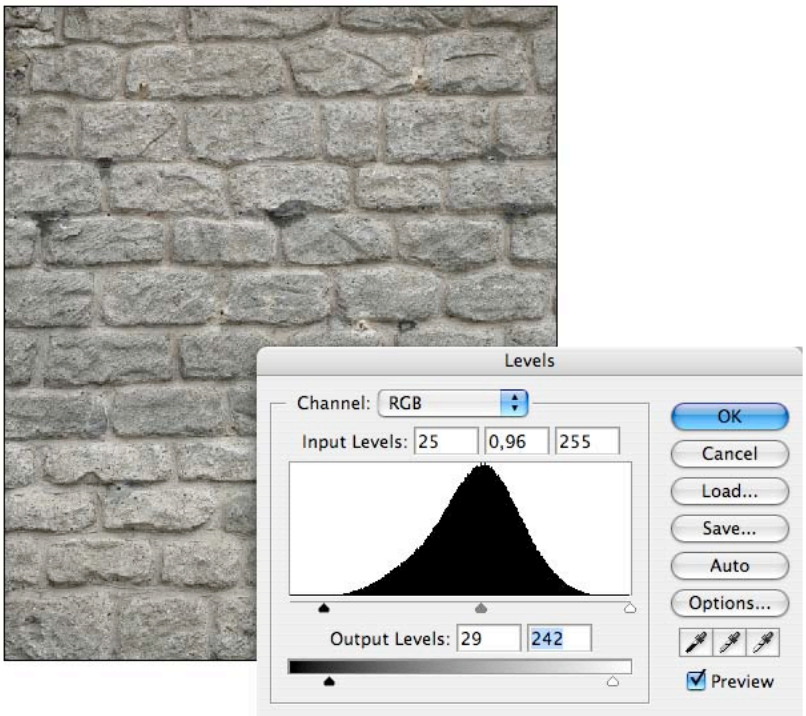


Abb.88: Hier werden die Farbbalance und die Tonwerte bearbeitet. In Photoshop hilft einem dabei ein Histogramm. Mit den drei Pfeilen unterhalb des Histogramms lässt sich die Intensität der Schatten, Mitteltöne und Highlights eines Bildes einstellen.

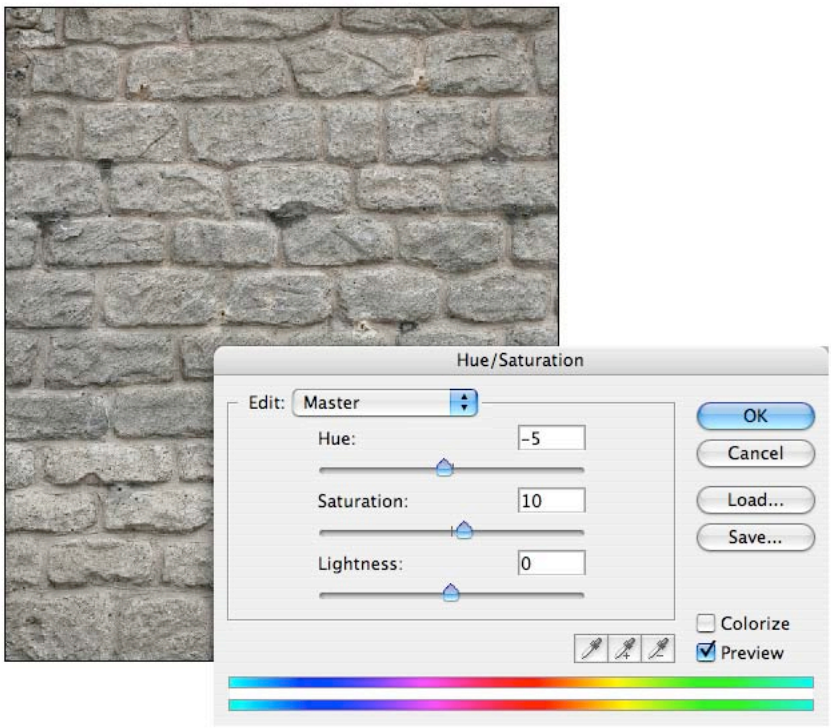


Abb.89: In einem weiteren Dialogfenster lassen sich die einzelnen Farben eines Bildes in Farbton, Sättigung und Helligkeit verändern. Hier wird das Bild leicht rötlich eingefärbt.

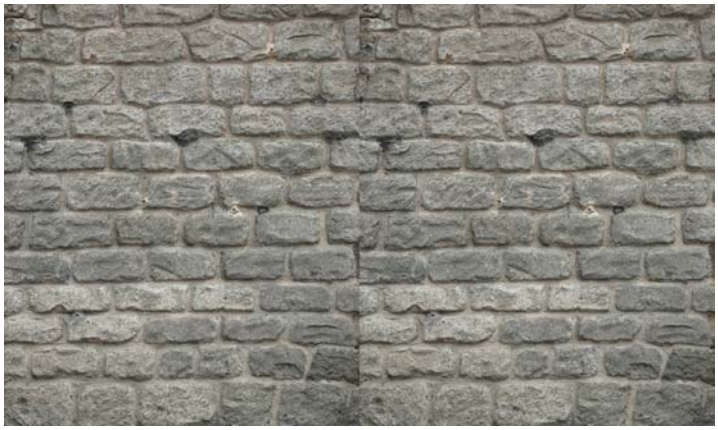


Abb.90: Das Bild wird jetzt kopiert und die Bilder werden aneinander gelegt. So würde die aktuelle Textur aussehen, wenn man sie in X-Richtung wiederholen würde. Eine auffällige Kante ist zu sehen.

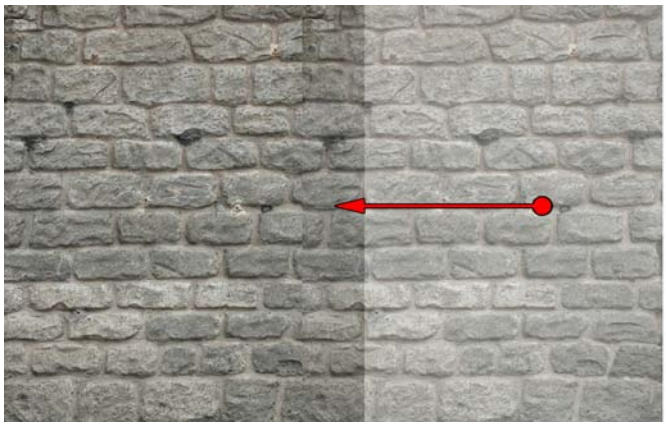


Abb.91: Wir schieben einen Teil des kopierten Bildes über das Originalbild da wir die Kante ausbessern möchten.

Abb.92: Mit einem Radiergummiwerkzeug wurde das kopierte Bild so bearbeitet, dass kein Stein mehr abgeschnitten aussieht.

Abb.93: Hier wurden im Bereich der Kante Tonwertkorrekturen vorgenommen um die Kante weiter zu verbergen.

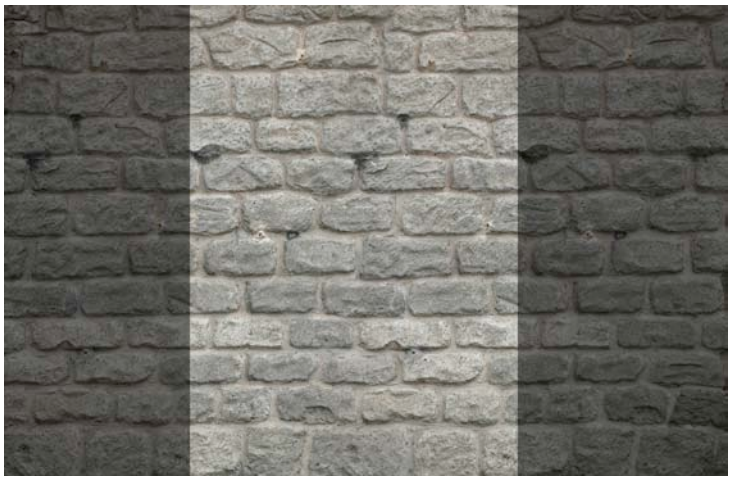


Abb.94: Der markierte Bereich des Bildes wird wieder ausgeschnitten und wiederholt betrachtet.

Abb.95: Die Kante ist zwar nicht mehr zu sehen, jedoch sind einige Bildelemente immer noch zu auffällig. So zum Beispiel die schwarzen Flecken im oberen Bereich der Textur.

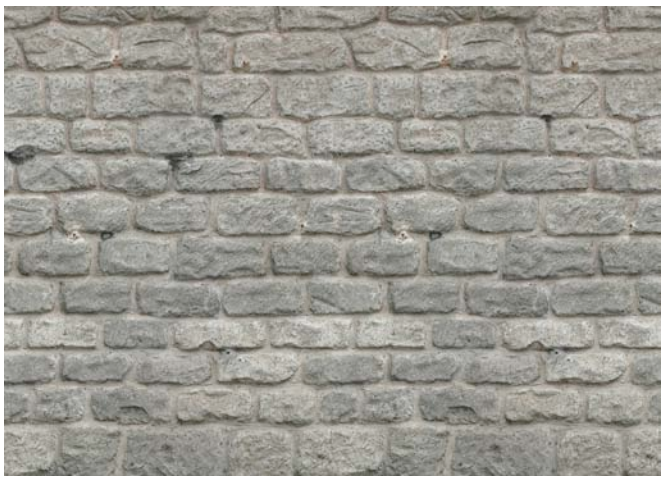


Abb.96: Um auffällige Elemente verschwinden zu lassen wurde das Bild einmal wiederholt und es wurden bestimmte Bildelemente übermalt und einige Variationen eingearbeitet.

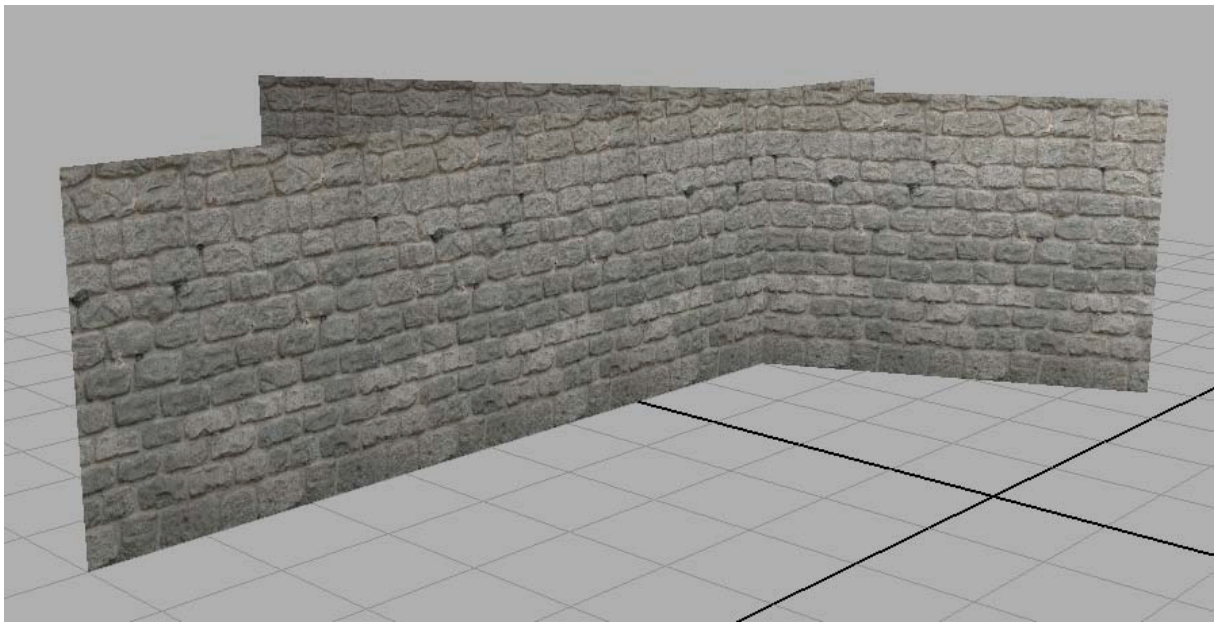


Abb.97: Zum Schluss sollte die Textur noch auf einem 3D-Objekt getestet werden.

2.6.4. Zeichnen der Textur in Photoshop

Zurück zum Wirtshausbeispiel; Bei der Textur des Wirtshauses wurde damit begonnen die verschiedenen Materialien festzulegen. Dazu wurden einige Fotografien wie im vorigen Abschnitt bearbeitet und anschließend auf der UV-Karte verteilt. Nach einigen Farb- und Tonwertsanpassungen wurden zum Schluss noch Elemente wie Türen, Fenster und Laternen gezeichnet.

Abb.98: Zahlreiche Fotografien vermitteln einen ersten Eindruck von der Farbzusammenstellung des Wirtshauses.

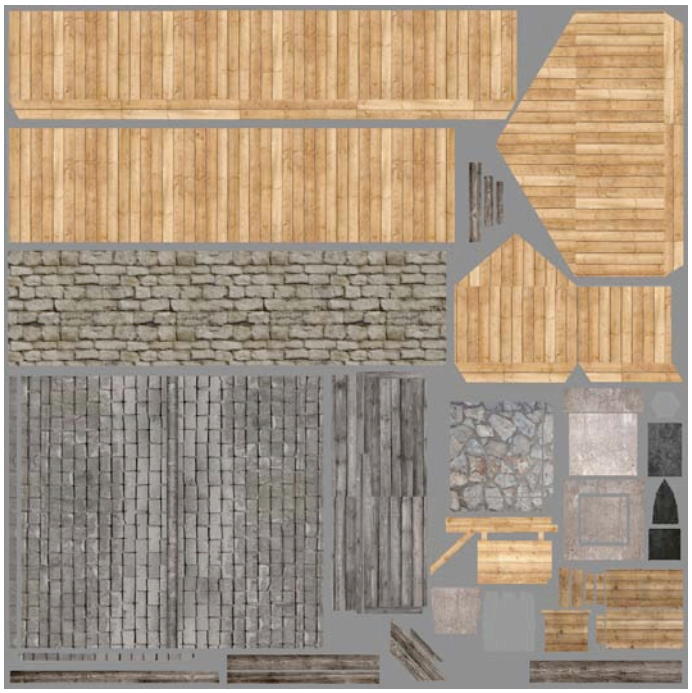


Abb.99: Mittels Tonwertkorrektur und Farbanpassung erlangt man ein einheitliches Bild.

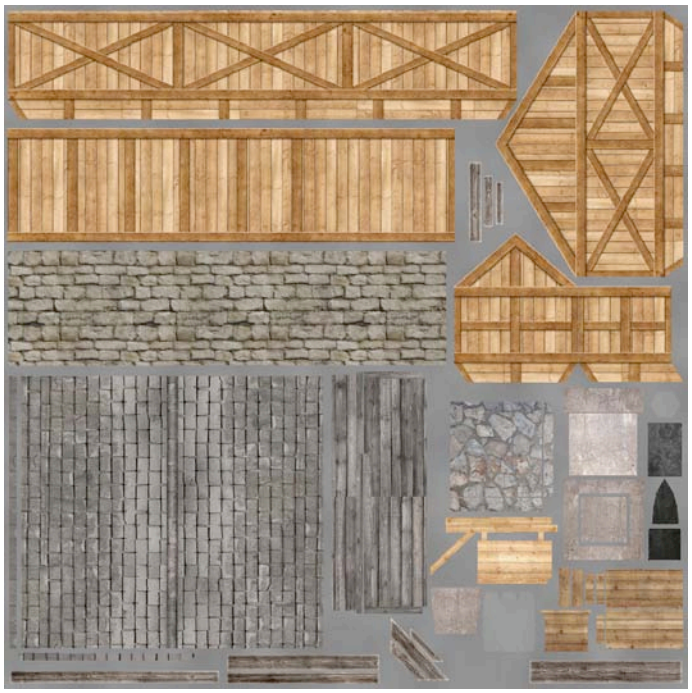


Abb.100: Wenn die Grundtextur zufrieden stellend ist, kann man beginnen Details einzuarbeiten. Hier sind dies die Stützen und Querstreben der Holzwände.

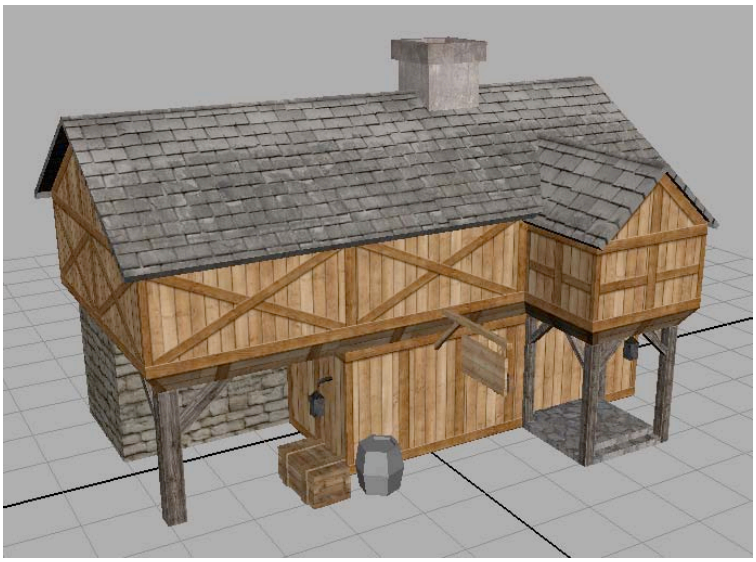


Abb.101: Zwischendurch sollte man seine Textur immer wieder auf dem Modell begutachten um Form- oder Positionsfehler zu erkennen.

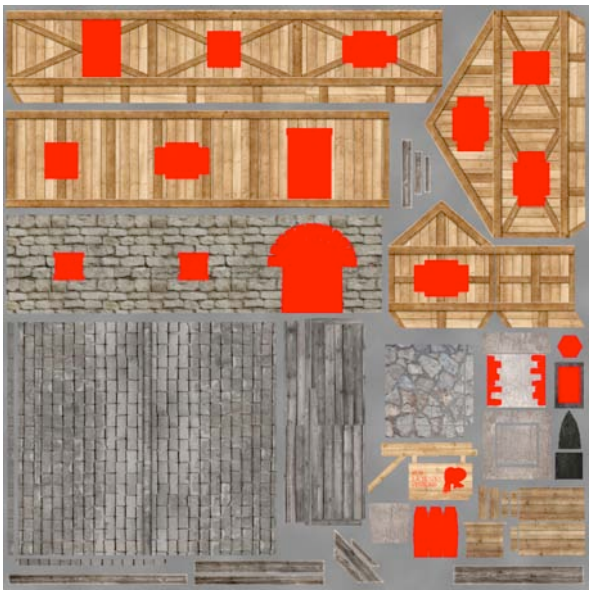


Abb.102: An den rot markierten Stellen mussten Details per Hand im Photoshop gezeichnet werden.

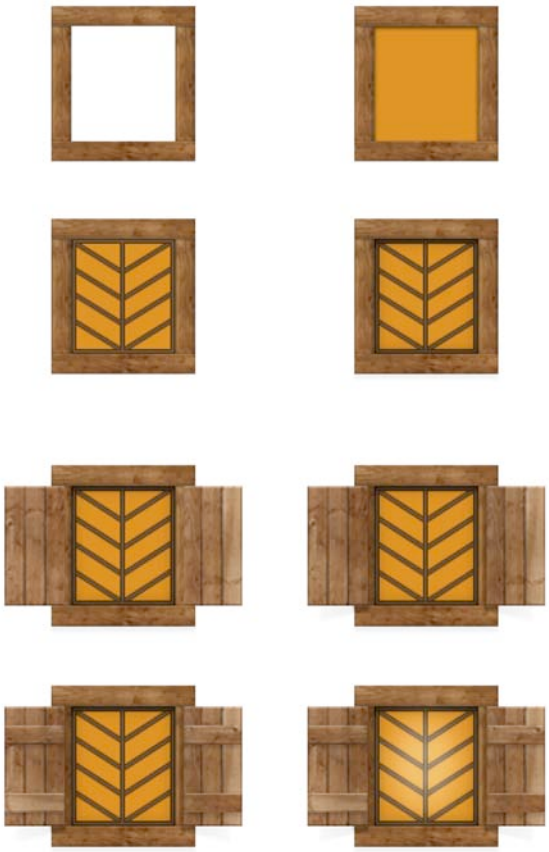


Abb.103: Von links nach rechts und von oben nach unten sieht man hier die Entstehung eines Fensters für das Wirtshaus.

2.6.5. Fertige Textur

Nachdem alles gezeichnet wurde, wird die in Photoshop erstellte Textur ins Targa-Format exportiert. Dieses "Truevision Advanced Raster Graphics Array" speichert Daten unkomprimiert mit bis zu 32 Bit pro Pixel als Rastergrafik und kann vom Spiel gelesen werden. Aus Performancegründen wird die Targa-Datei noch von 1024*1024 Pixel auf 512*512 Pixel skaliert.

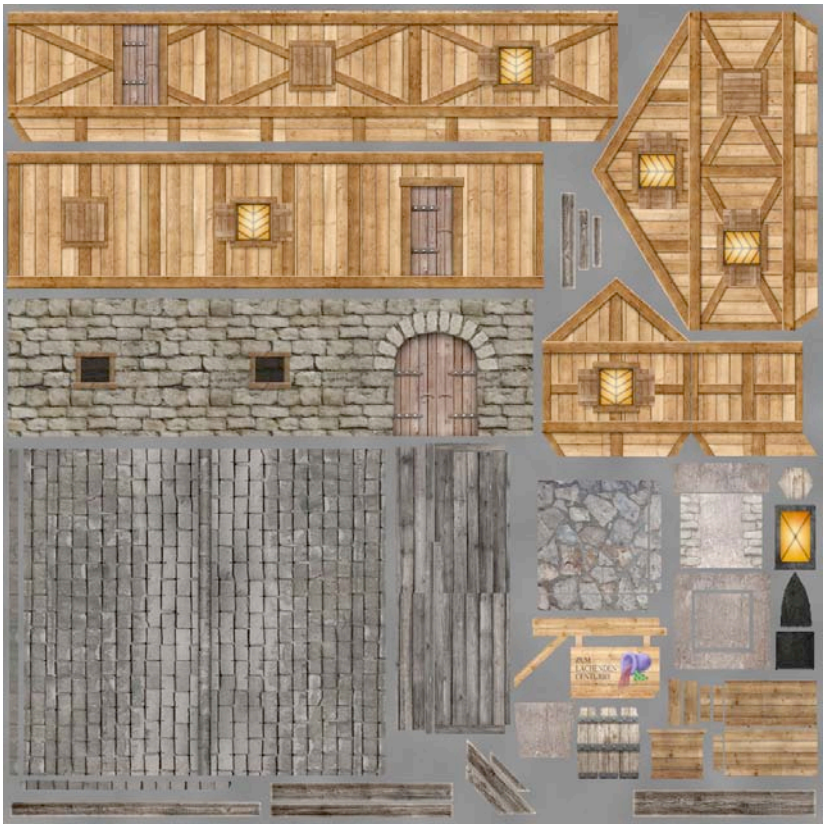


Abb.105: Die fertige, im Spiel verwendete, Textur des Wirtshauses

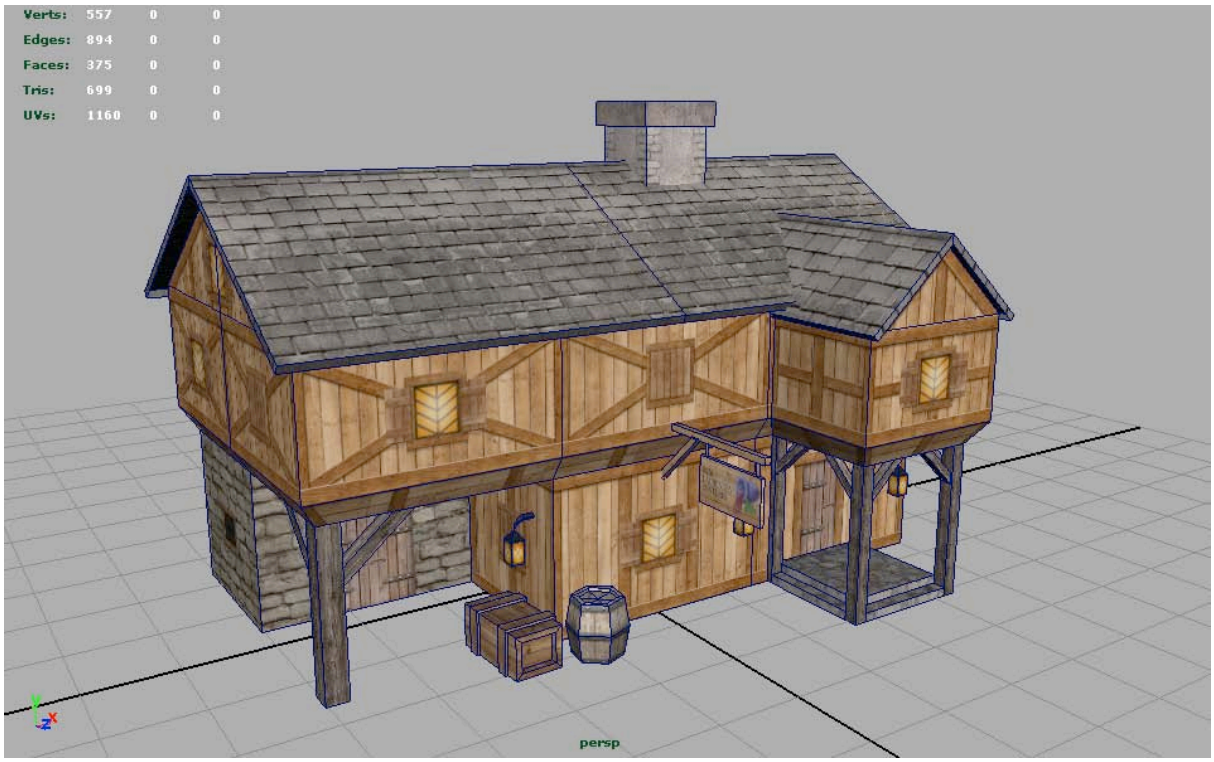


Abb.106: Die fertige Targa-Textur auf dem 3D-Modell in Maya

2.7. Konvertierung

Da nun das Modell, die Texturkoordinaten und die Textur fertig gestellt sind, kann man das Wirtshaus für das Spiel konvertieren. Zuvor sollte man noch schnell drei verschiedene Modelle aus dem fertigen Modell generieren, sodass das Haus im Spiel Baustufen vorweisen kann. Dies erfolgt meist durch einfaches Löschen bestimmter Teile.

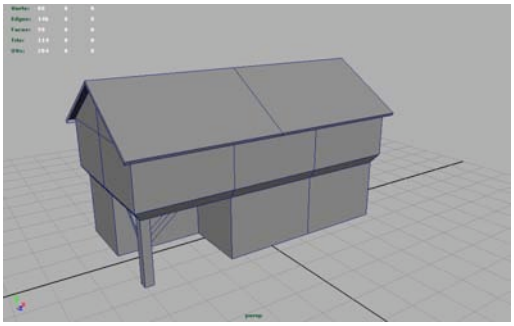


Abb.107: Erste Baustufe

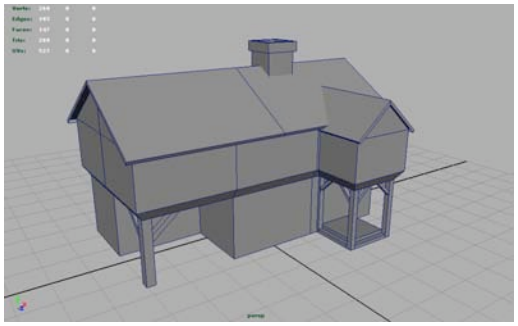


Abb.108: Zweite Baustufe

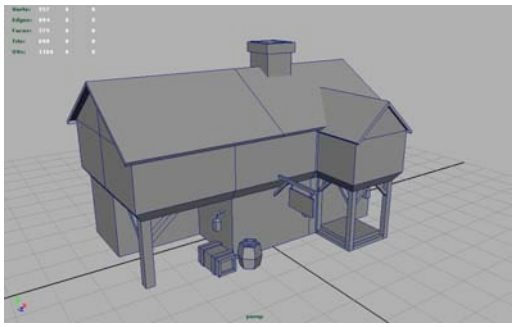


Abb.109: Dritte Baustufe

Die drei Baustufen werden nun trianguliert, d.h. das Modeling-Programm unterteilt die Modelle in Dreiecke. Abschließend werden alle als RTG-Datei exportiert.

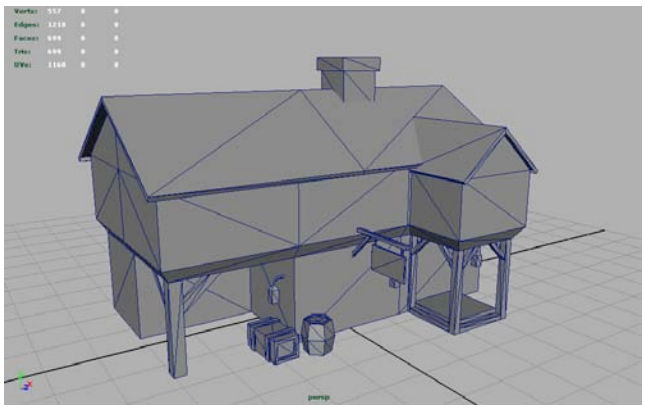


Abb.110: Ein trianguliertes Modell das ausschließlich aus Dreiecken besteht

Im APXConvert müssen die RTG-Dateien in der richtigen Reihenfolge importiert und mit der Targa-Textur belegt werden. Einstellungen für die Transparenz oder

zweiseitige Lichtberechnung entfallen beim Wirtshaus. Die vier Dateien werden vom APXConvert in ein APX-Paket konvertiert, was ungefähr fünf Sekunden in Anspruch nimmt.

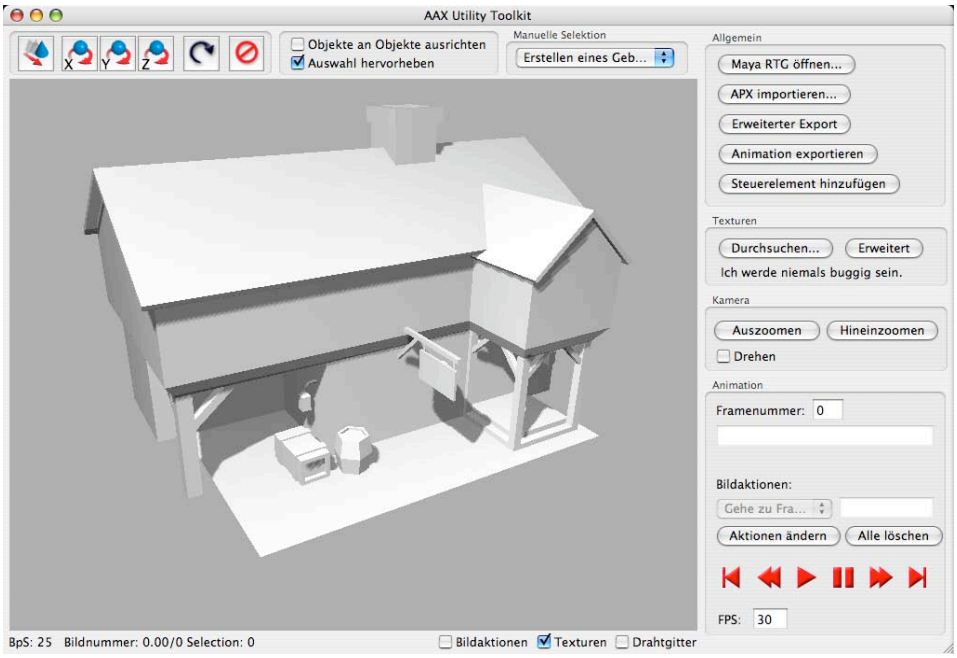


Abb.111: APXConvert mit RTG-Dateien des Wirtshauses

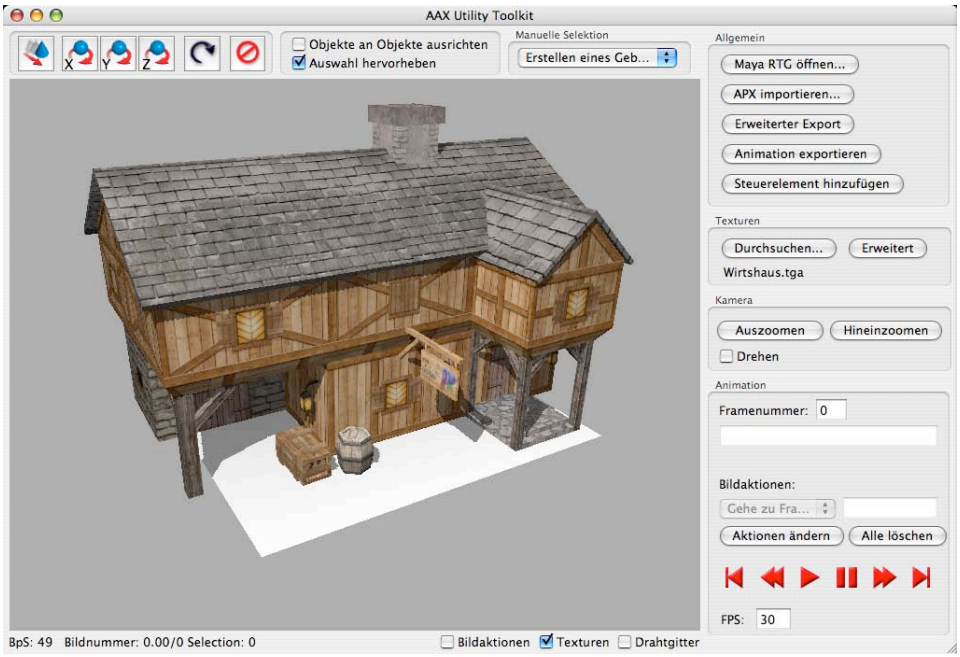


Abb.112: Texturiertes Wirtshaus im APXConvert

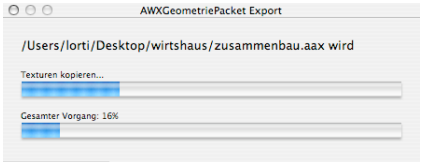


Abb.113: Erstellen des APX-Paketes

Wenn man sich nun den Ordner des APX-Paketes ansieht, fehlen zwei Dateien. Die button.tga, welche die Schaltfläche im Spiel darstellt, und die config.att, welche Informationen und Werte für das jeweilige Modell enthält. Die button.tga beinhaltet meist einen Screenshot des Modells im APXConvert und ist eine 64*64 Pixel große Targa-Datei. Die config.att ist eine modifizierte XML-Datei mit dem Namen des Modells, einer Beschreibung, den maximalen Trefferpunkten, den im Gebäude erstellbaren Einheiten und weiteren Informationen.

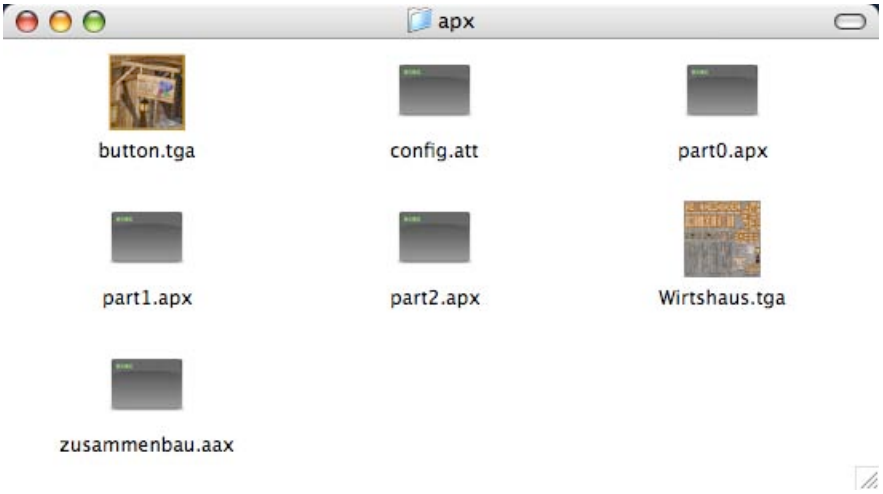


Abb.114: Vollständiges APX-Paket für das Wirtshaus

2.8. Darstellung im Spiel

Wurden alle Schritte richtig ausgeführt und das APX-Paket in den entsprechenden Spielordner kopiert, wird das Spiel gestartet. Gibt es beim Laden keine Fehlermeldung wurden die Geometrie- und Texturdaten vom Spiel erkannt. Sollte das Spiel dennoch beim Bau eines Wirtshauses abstürzen gibt es Probleme mit der Konfigurationsdatei.

Abb.115: Das fertige Wirtshaus im Spiel

3. Erstellen einer Figur

Das Erstellen einer Figur unterscheidet sich vom Erstellen eines Gebäudes in erster Linie durch die Notwendigkeit von Animationen. Die Abläufe beim Modellieren und Texturieren ähneln sich stark. Jedoch wird von den Figuren zusätzlich noch verlangt, dass diese gehen, laufen oder arbeiten. Deshalb wird in diesem Kapitel hauptsächlich auf diesen besonderen Aspekt eingegangen.

3.1. Referenzen, Modellierung und Texturierung

Um einen einigermaßen glaubhaften Menschen zu modellieren bedarf es gewissen Grundkenntnissen der Anatomie. Es gibt hierzu spezielle Bücher und Websites die sich mit den notwendigen Anatomiekenntnissen eines Künstlers befassen. Es wird dort mehr auf die äußere Form und die Bewegungsabläufe eines Menschen eingegangen, als auf die Funktionen und Details von inneren Organen und medizinische Belange. Das Studieren von Aktfotos und anderen Referenzen ist beim Modellieren eines Körpers sehr wichtig.

Für die Rüstungsdetails wurden mehrere Historienbücher und teils auch fiktive Geschichten über das antike Rom als Grundlage verwendet, um eine authentische Kleidung der Soldaten und Bürger zu gewährleisten. Mangels Quellinformationen werden hier keine der über 50 Referenzbilder abgebildet.

Als Beispiel für dieses Kapitel dient ein römischer Bogenschütze mit einem Lorica Segmentata gennanten Glieder- oder Spangenpanzer, Caligae, Galea, einem Bogen und einem Köcher. Das Budget für das Modell sind 800 Dreiecke und eine Textur mit einer Auflösung von 256*256 Pixel.

Abb.116: Das 716 Dreiecke umfassende 3D-Modell des Bogenschützen

Abb.117: Fertiges 3D-Modell mit Textur

3.2. Rigging

"In computer animation, parts of a 3D model are rigged so that an animator can easily select and move parts of the model (such as an arm or leg) without having to painstakingly select every single vertex in the desired part." [41]

Rigging wird der Vorbereitungsprozess vor der eigentlichen Animation genannt. Dabei werden Mechanismen und Hilfsobjekte erstellt, die es dem Animateur erheblich erleichtern das Modell zu bewegen, deformieren und animieren. In den Anfängen der Computerspiele wurden Animationen noch mit Vertexverschiebung oder gar nur mit Translation, Rotation oder Skalierung des ganzen Modells angefertigt. Heutzutage wäre das Bewegen einzelner Vertices ein unendlich aufwändiger Prozess, da moderne Spiele bereits Modelle mit über 10 000 Dreiecken benutzen.

Beim Rigging wird im ersten Schritt meist ein Skelett für das jeweilige Modell erstellt. Dieses Skelett besteht aus mehreren verbundenen Gelenken und wird später anstelle der einzelnen Vertices bewegt. Dies wird dadurch ermöglicht, dass alle Vertices mit einer Gewichtung zu einem Gelenk gehören und sich mit diesem mitbewegen. Der Vorgang bei dem ein Modell mit einem Skelett verbunden wird nennt man Skinning.

---

41 Wikipedia - die freie Enzyklopädie. Rig. Online im Internet: URL: http://en.wikipedia.org/wiki/Rig, zugegriffen am 30. April 2007

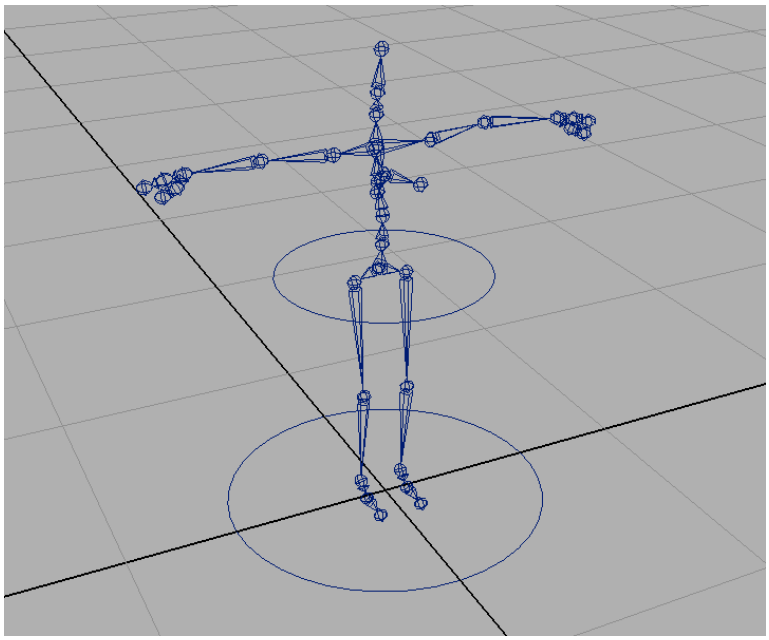


Abb.118: Ein fertiges Skelett für den Bogenschützen

Um dem Animateur die Arbeit weiter zu erleichtern, lassen wir ihn nicht die Gelenke direkt drehen. Um einen Arm zu positionieren müsste man z.B. drei Gelenke richtig ausrichten. Deshalb verwendet man hier die aus der Robotik bekannte Inverse Kinematik.

" Die inverse Kinematik [...] bezeichnet das Gegenstück zur direkten Kinematik und beschäftigt sich mit der Frage, wie aus der Lage (Position und Orientierung) des Endeffektors [...] die Gelenkwinkel der Armelemente bestimmt werden können. Sie spielt damit eine wichtige Rolle bei der Bewegung von Industrierobotern und bei der Computeranimation von Charakteren.

Bei der inversen Kinematik wird das letzte Glied der kinematischen Kette, der sogenannte Endeffektor, bewegt und in die gewünschte Lage gebracht. Die übrigen Glieder der Kette müssen dann entsprechend den Freiheitsgraden ihrer Gelenke passende Lagen einnehmen.

Vergleichen lässt sich dies mit dem menschlichen Arm, der mit seinen Gelenken auch eine solche kinematische Kette darstellt: Bringt man beispielsweise die Hand in eine bestimmte Lage, nehmen Handgelenk, Ellenbogen und Schulter automa-

tisch ebenfalls bestimmte Stellungen ein. Genau diese (Gelenkwinkel-)Stellungen müssen über die inverse Kinematik bestimmt werden." 42

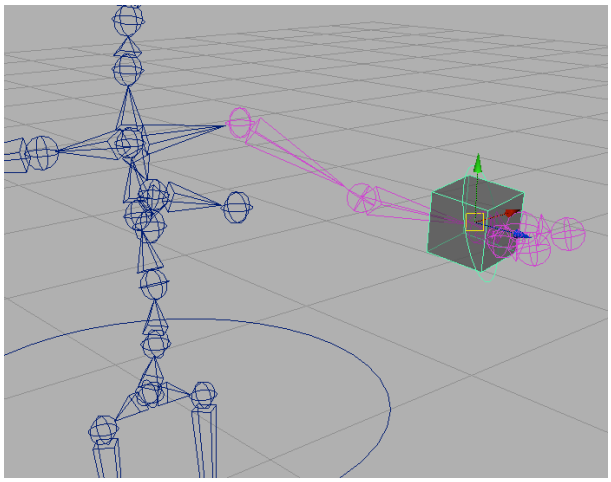


Abb.119: Der Würfel ist der Endeffektor des Arms. Bewegt man den Würfel so folgen ihm das Schulter- und das Ellbogengelenk.

Es werden zusätzlich zur inversen Kinematik noch weitere Feinheiten in das Rig eingebaut, die an dieser Stelle nicht ausführlich beschrieben werden. Anstatt die Gelenke zu bewegen, bewegt der Animateur schlussendlich nur noch die Würfel und Kreise, die man in der nächsten Abbildung sieht. Er kann die Hände und Füße positionieren, die Positionen der Knie- und Ellbogengelenke feineinstellen, die Hüfte über den Kreis in der Mitte bewegen, das ganze Skelett über den Kreis am Boden bewegen, den Oberkörper und die Wirbelsäule mit dem großen Quader in der Mitte bewegen und den Kopf der Figur drehen.

---

42 Wikipedia - die freie Enzyklopädie. Online im Internet: URL: http://de.wikipedia.org/w/index.php?title=Inverse_Kinematik&oldid=30686673, zugegriffen am 1. Mai 2007

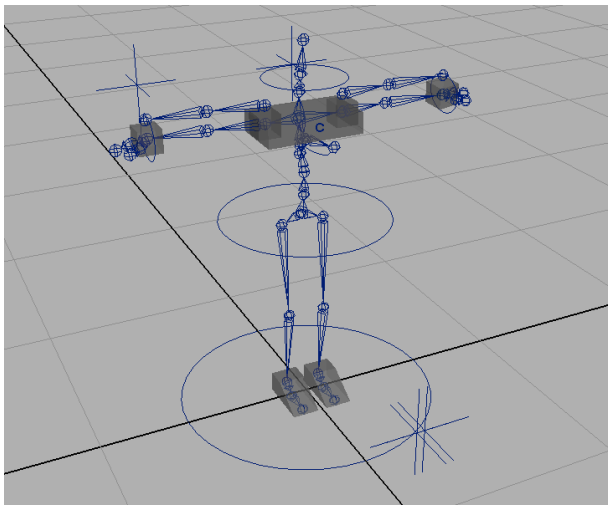


Abb.120: Das fertige Rig mit allen Kontrollmöglichkeiten

Abb.121: Fertiges Rig mit verbundenem Modell und eingestellten Gewichten

Wenn das Rig vollständig und alle Gewichte beim Skinning eingestellt sind, testet der Animateur das Rig noch in verschiedensten Posen. In großen Firmen gibt es meist Animateure und separate Charakter TDs (Technical Directors), die einzig und allein für die Rigs zuständig sind. Sind die Animateure mit den Rigs nicht zufrieden werden diese zu den TDs zurückgeschickt und verbessert.

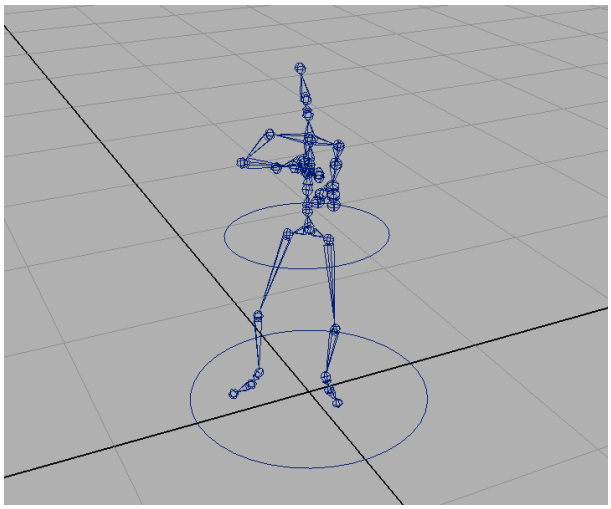


Abb.122: Skelett des Bogenschützen in Zielpose

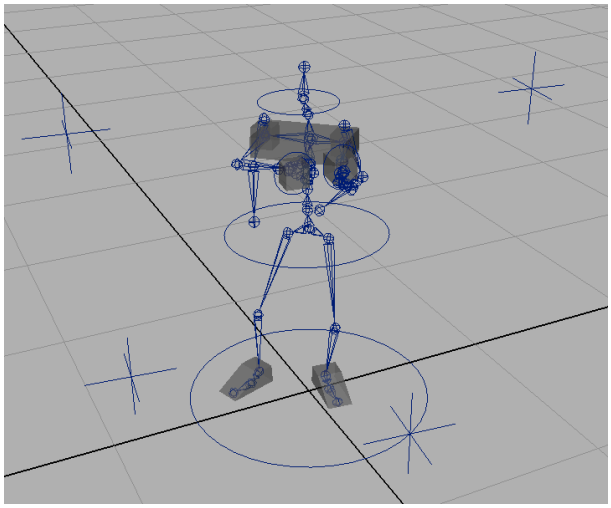


Abb.123: Skelett mit Kontrollobjekten in Zielpose

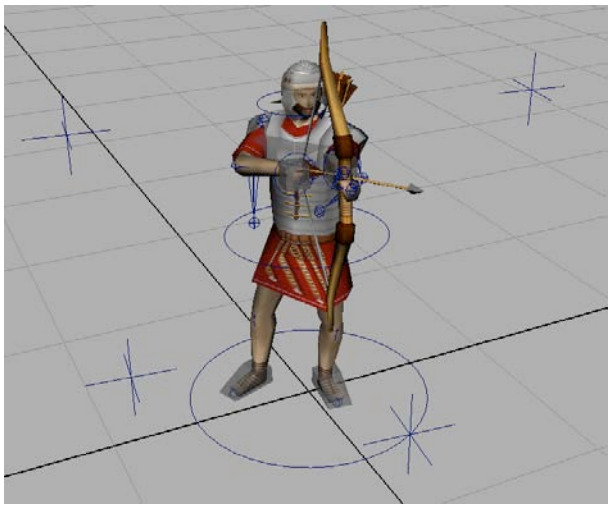


Abb.124: Das deformierte Modell in Zielpose

3.3. Animation

Mit dem fertigen Rig können endlich die Animationen erstellt werden. Dies erfolgt mithilfe einer Zeitleiste und Schlüsselbildern. Die unterschiedlichen Positionen und Drehwinkel der Kontrollobjekte können in Schlüsselbildern zu verschiedenen Zeitpunkten gespeichert werden. Zwischen den Schlüsselbildern wird dann interpoliert. In Maya können die einzelnen Schlüsselbilder auch noch kopiert oder verschoben werden um Feineinstellungen bei den Animationen vorzunehmen.

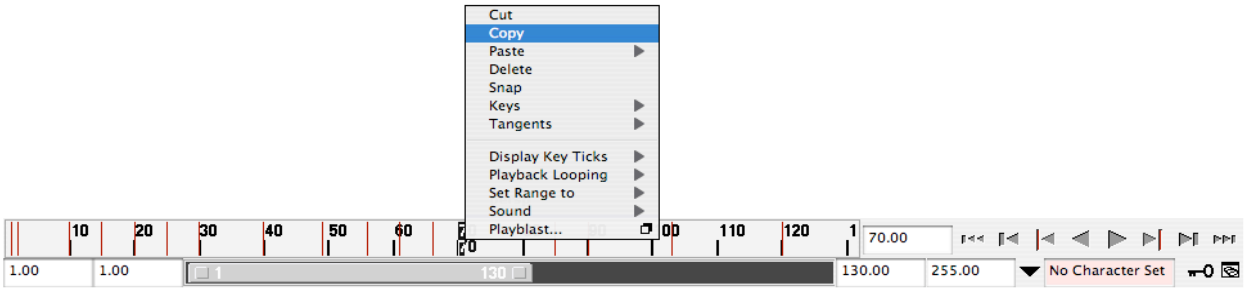


Abb.125: Rote Querstriche symbolisieren Schlüsselbilder auf der Zeitleiste in Maya.

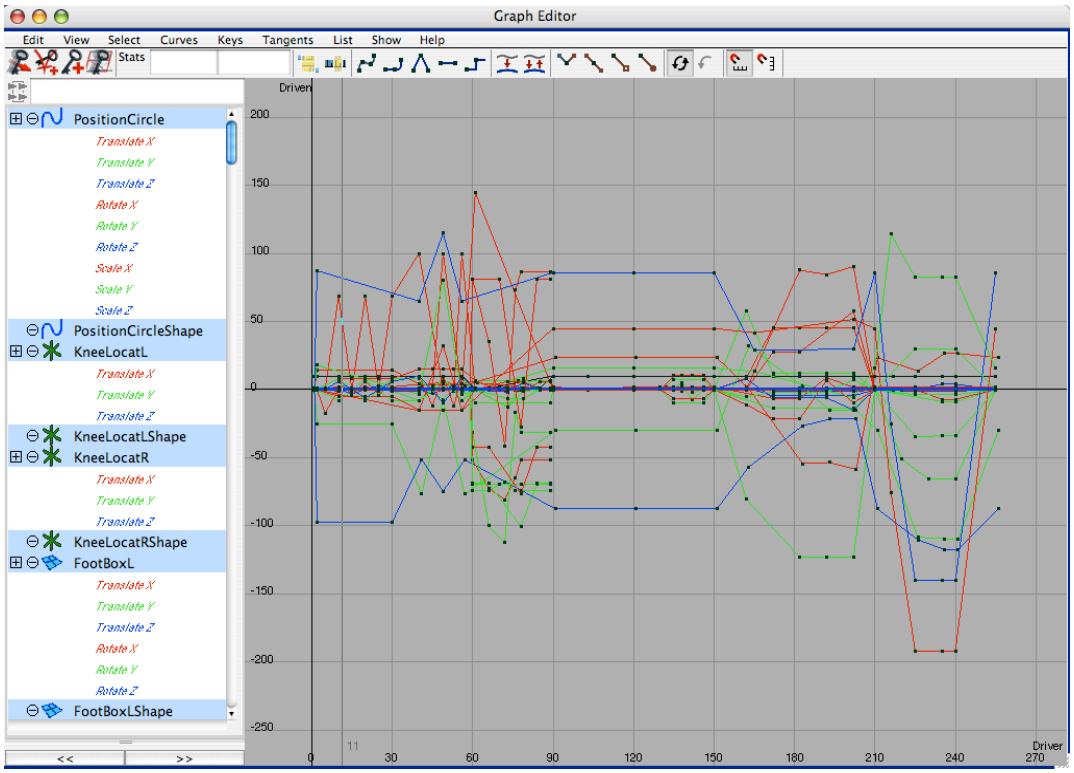


Abb.126: Der Graph Editor von Maya bietet eine Übersicht über die Änderung eines Zahlenwertes, z.B. der Verschiebung eines Objektes in Z-Richtung, über die Zeit.

Abb.127: Der Gehzyklus des Bogenschützen Bild für Bild

3.4. Konvertierung

Die Konvertierung einer Figur erfolgt gleich wie bei einem Gebäude mithilfe des APXConverts. Es wird einfach jedes einzelne Bild einer Animation als RTG abgespeichert, da das Spiel zurzeit keine Schlüsselbildanimationen unterstützt, und im APXConvert geladen. Dem jeweils ersten Bild einer Animation wird ein Name gegeben und mitsamt der Textur wird vom APXConvert ein APX-Paket erzeugt.

Ausblick

Da dieses Projekt ohnehin schon den üblichen Aufwand einer HTL-Diplomarbeit überschritten hat, war es uns selbstverständlich nicht möglich, alle Ideen umzusetzen. Einige Punkte, die wir nicht umgesetzt haben, und welche noch implementiert werden könnten, wären:

Im Bereich der Künstlichen Intelligenz:

- Ausarbeitung des Computergegners hinsichtlich Ökonomie
- Heranziehen von Fuzzy-Logic-Systemen zur Entscheidungsfindung beim Computergegner, wodurch dieser dynamischer reagieren könnte
- Möglicher Einbau eines Neuronalen Netzes für den Computergegner

Im Bereich der Grafik-Engine:

- Verwendung dynamischer Detailstufen (Level of Detail) beim Rendering; ergänzend zum bereits bestehenden Culling. Besonders beim Terrain-Rendering könnte dies die Performance enorm erhöhen.
- Einbindung von Animationen mittels Schlüsselbildern: Durch die Interpolation der Objektkoordinaten zwischen Keyframes könnte der Speicherbedarf vor allem bei längeren Animationen enorm reduziert werden.
- Einbindung besserer Lichtberechnungsmodellen und Bump Mapping, um die Modelle realistischer erscheinen zu lassen.
- Einbau von dynamischem Tag- und Nachtwechsel, sowie weiteren Wettersituationen wie Stürme oder Überschwemmungen

Im Bereich des Gameplays:

- Erweiterung und Ausbau des Kampfsystems und der Truppensteuerung
- Einbau weiterer Einflussfaktoren in das Spiel

Abschließend ist zu sagen, dass unsere ganze Arbeit nur mit einem funktionierenden Projektmanagement abgeschlossen werden konnte!

Glossar

AAX	Informationen über alle **APX**-Objekte einer Szene werden in einer AAX-Datei gespeichert.
Alpha	Alpha ist der vierte Farbwert eines Pixels in einem RGBA-Bild. Er wird zur Beschreibung der Transparenz verwendet.
APX	Als APX wird das spezielle Modellformat von Augustus bezeichnet. Es beinhaltet alle für die Darstellung notwendigen Informationen.
APXConvert	Das APXConvert ist ein Programm zur Erstellung von **APX**- und **AAX**-Dateien.
Arbeitsplan	Arbeitspläne strukturieren die Abschnitte zwischen den Meilensteinen und helfen die Aufgaben, oder auch Arbeitspakete, abarbeitbar zu machen.
Augustus	Augustus war der erste römische Kaiser, er trug zuvor den Namen Gaius Octavianus.
Defuzzifizierung	Bei der Defuzzifizierung werden die unscharfen Ausgänge des **Fuzzy Logic**-Regelwerks wieder in konkrete Zahlenwerte übersetzt.
Direct3D	Direct3D ist eine Programmierschnittstelle für 3D-Computergrafik und Bestandteil von DirectX.
Echtzeit-Strategiespiel	Echtzeit-Strategiespiele stellen ein Computerspiel-Genre dar, bei dem alle Spieler ihre Handlungen gleichzeitig ausführen und in dessen Vordergrund das wirtschaftliche, strategische und taktische Agieren gegen einen Gegner steht.

Engine Eine Engine ist das technische Grundgerüst eines Computerspiels. Es ermöglicht die Verwendung verschiedener Komponenten für die Darstellung, das Erkennen von Kollisionen, Künstliche Intelligenz oder Tonausgabe.

Fog of War siehe **Kriegsnebel**

Fuzzy Logic Fuzzy Logic ist eine Methode der **Künstlichen Intelligenz**. Sie wurde entwickelt, um eine Art menschliche Logik zu imitieren. Im Gegensatz zur bool'schen Logik arbeitet Fuzzy Logic nicht mit scharfen Grenzen zwischen Zuständen. Sie ermöglicht Computersystemen mit „menschlichen" Begriffen, wie „wenig", „viel" oder „mittel", zu arbeiten.

Exakte Werte werden unscharfen Zuständen zugeordnet und mithilfe eines einfachen Regelwerks verknüpft und auf Ausgänge geführt. Die Ausgänge werden zum Schluss **defuzzifiziert**.

Fuzzy Set Ein Fuzzy Set dient zur Beschreibung eines Zustandes einer **linguistischen Variable**.

Gameplay Als Gameplay werden die Erfahrungen und Aktionen, die ein Spieler während des Spielens macht, zusammengefasst.

GLUT GLUT (Graphic Library Utility Tookit) dient als Zwischenstück zwischen OpenGL und dem Betriebssystem. Siehe dazu auch **SDL**.

Inverse Kinematik Bei der inversen Kinematik wird das letzte Glied der kinematischen Kette bewegt und in die gewünschte Lage gebracht. Die übrigen Glieder der Kette müssen dann entsprechend den Freiheitsgraden ihrer Gelenke passende Lagen einnehmen. Sie spielt eine Rolle große Rolle in der Robotik.

Kampagne	Eine Kampagne bezeichnet in **Echtzeit-Strategiespielen** eine Serie von Spielszenarien mit zeitlichem und geschichtlichem Zusammenhang.
Kriegsnebel	Gebiete außerhalb des Sichtradius einer Spielfigur sind in den Kriegsnebel gehüllt. Der Spieler kann im Kriegsnebel zwar landschaftliche Details ausmachen, bekommt aber nichts vom Geschehen in diesen Gebieten mit.
Künstliche Intelligenz	Künstliche Intelligenz (KI) ist ein Teilgebiet der Informatik, das sich mit dem Nachbilden von intelligentem Verhalten befasst.
LevelEditor	Der LevelEditor ist ein Programm zur Erstellung der Spielwelt.
Linguistische Variable	Eine linguistische Variable ist eine Variable (z.B. Geschwindigkeit), welche unscharfe Werte annehmen kann (z.B. schnell, viel oder stark).
Mamdani	Methode zur **Defuzzifizierung**
Meilenstein	Als Meilenstein definiert man ein Zwischenergebnis im Projekt, das inhaltlich und terminlich schon im Vorfeld genau festgelegt wird.
Neuronales Netz	Ein neuronales Netz ist ein künstliches Modell eines menschlichen Gehirns. Es wird als Methode der **Künstlichen Intelligenz** zur Entscheidungsfindung benutzt.
Normalvektoren	Die Richtung eines Normalvektors bestimmt die Vorderseite eines Polygons.
OpenGL	OpenGL (Open Graphics Library) ist eine plattform- und programmiersprachunabhängige Schnittstelle zur Entwicklung von 3D-Computergrafik.
Partikelsystem	Partikelsysteme sind Grafikeffekte die aus vielen voneinander unabhängig berechneten Einzelteilen (Partikeln) bestehen. Ein typischer Anwendungsfall ist die Darstellung von Feuer.

Polygon	Wird in der Computergrafik von Polygonen gesprochen werden meist Dreiecke gemeint, da man mithilfe dieses kleinsten Vieleckes alle größeren Vielecke erstellen kann.
Primitive	Ein Primitiv bezeichnet die kleinste geometrische Form in einem Programm für Computergrafik. So zum Beispiel Punkte, Linien oder Polygone.
Projektstrukturplan	Projektstrukturpläne stellen das zentrale Planungsinstrument in einem Projekt dar. Ziel des Projektstrukturplanes ist es die Gesamtaufgabe in autonom bearbeitbare Teilaufgaben zu zerlegen.
Rasterization	Bei der Rasterisierung werden geometrische Daten und Formen in Pixeldaten zur Darstellung umgewandelt.
Rigging	Beim Rigging werden Mechanismen und Hilfsobjekte erstellt, die es dem Animateur erheblich erleichtern das 3D-Modell zu bewegen, deformieren und animieren.
SDL	SDL stellt unter anderem Funktionen für Benutzereingaben und Fenstermanagement zur Verfügung. Es erstellt die für eine Grafikschnittstelle notwendige Umgebung und dient als Zwischenstück zum Betriebssystem.
Skinning	Der Vorgang beim **Rigging**, bei dem ein Skelett mit einem 3D-Modell verbunden wird, bezeichnet man als Skinning.
Sugeno	Methode zur **Defuzzifizierung**
Targa	Das "Truevision Advanced Raster Graphics Array" ist ein Bildformat, welches Daten unkomprimiert mit bis zu 32 Bit pro Pixel als Rastergrafik speichern kann.
Technologiebaum	Ein Technologiebaum dient in **Echtzeit-Strategiespielen** zur Übersicht darüber, welches Objekt welche Aktionen ermöglicht und welche Objekte überhaupt existieren.
Textur	Eine Textur ist ein Pixelbild, welches unter Beachtung gewisser Kriterien übern ein 3D-Modell gelegt wird.

Texturkoordinaten	Die Texturkoordinaten legen fest, wie eine Textur auf das Objekt projiziert wird. Jedem Vertex des Modells wird ein Punkt in der Texturebene zugewiesen
Vertex	Als Vertex wird ein Eckpunkt eines 3D-Objektes bezeichnet. Ein Dreieck besitzt 3 Vertices.

Quellenverzeichnis

3D-Computergrafik:

URL: http://www.biologie.de/biowiki/3D-Computergrafik

ANTHONY ROSSANO: Inside Softimage 3D:

URL: http://www.digitalproducer.com/pages/soft_3d_p1.htm

CRYTEK GMBH, Crysis-Screenshot:

URL: http://www.sg.hu/kep/2006_06/0610cry.jpg

DAVE SHREINER, MASON WOO, JACKIE NEIDER, TOM DAVIS:

OpenGL Programming Guide, 4. Auflage, Addison-Wesley

Diplomarbeit mit dem Thema Game Engine

URL:http://www.fit.fraunhofer.de/gebiete/mixed-reality/

diplomaTh/DA_MRGameEngine.pdf

JOCHEN HOOG: Architecture_Engine_1.0, 2004

URL: http://www.atelierprozess.net/hoog/files/diplomarbeit_hoog.pdf

PHILLIPP STRAHL: Photographing for Textures

URL: http://www.highend3d.com/maya/tutorials/general/256_1.html

PM-Handbuch.com - Der Kostenlose Leitfaden für Projektmanager

URL: http://www.pm-handbuch.com/

Politische und kulturelle Entwicklung Roms

URL: http://www.gottwein.de/roge/his_0044.php, zugegriffen im Jänner 2006

PUTZGER-BRUCKMÜLLER: Historischer Weltatlas, 2. Auflage, Wien 2000

Prof. Dipl-Ing. FELIX SCHWAB, Univ.-Prof. Dkm. Mag. Dr. WILFRIED SCHNEIDER, Prof. (FH) Mag. INGRID SCHWAB-MATKOVITS: EDV Projektentwicklung, 4. überarbeitete Auflage, Wien 2004, MANZ Verlag

Torben Fugger: 3D-Game-Engine, 2004

URL: http://www.uni-koblenz.de/~cg/Diplomarbeiten/DiplomarbeitFugger.pdf

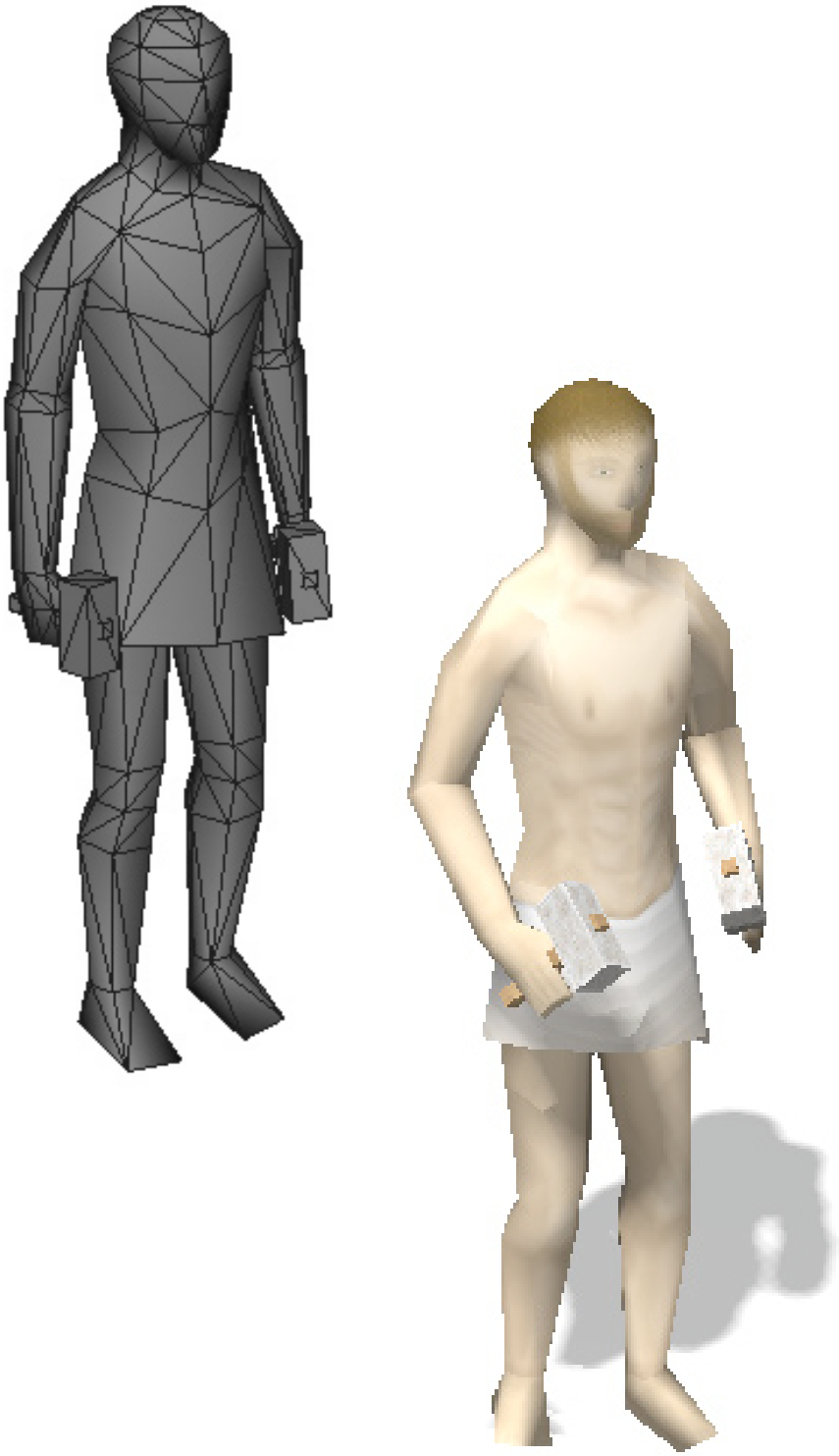
Wikipedia - Die freie Enzyklopädie

URL: http://www.wikipedia.org/

Abbildungsverzeichnis

Anhang

Modellgalerie



Arbeiter
512 Dreiecke

Der Arbeiter ist ein Sklave, welcher Gebäude errichten und Rohstoffe abbauen kann.

Soldat
553 Dreiecke

Der Fußsoldat ist mit Schild und Schwert ausgerüstet und in großen Formationen gut im Nahkampf einsetzbar.

Bogenschütze
716 Dreiecke

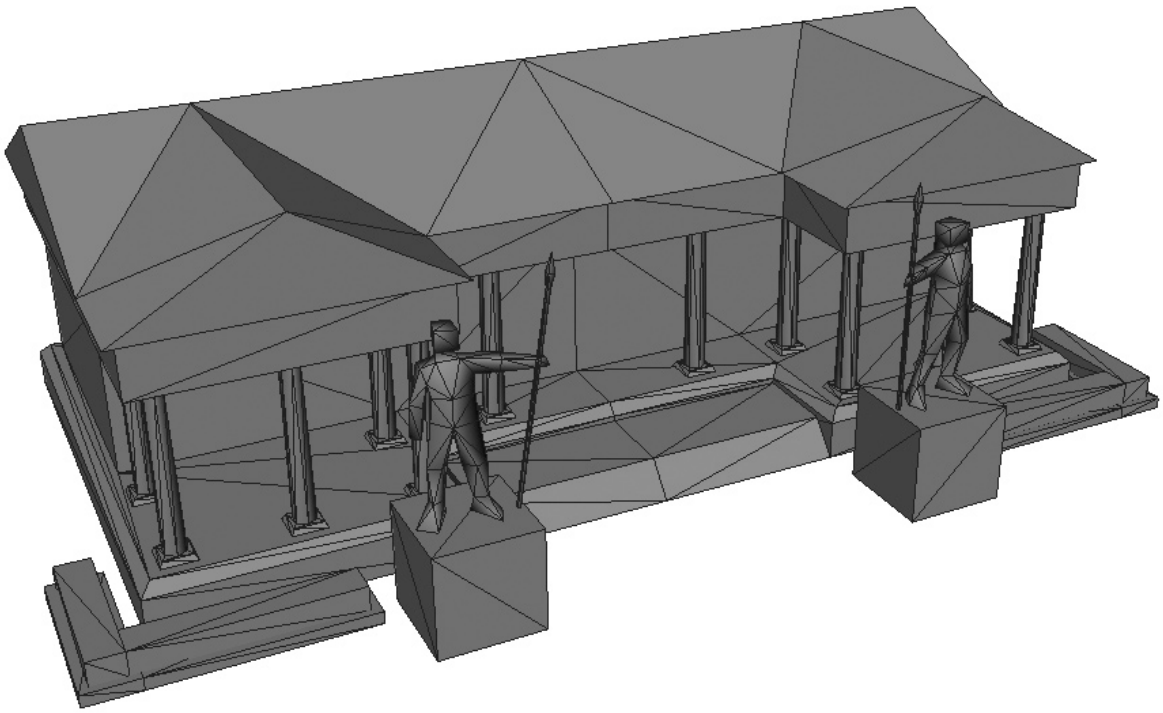
Der Bogenschütze agiert auf Distanz und ist daher im Nahkampf benachteiligt.

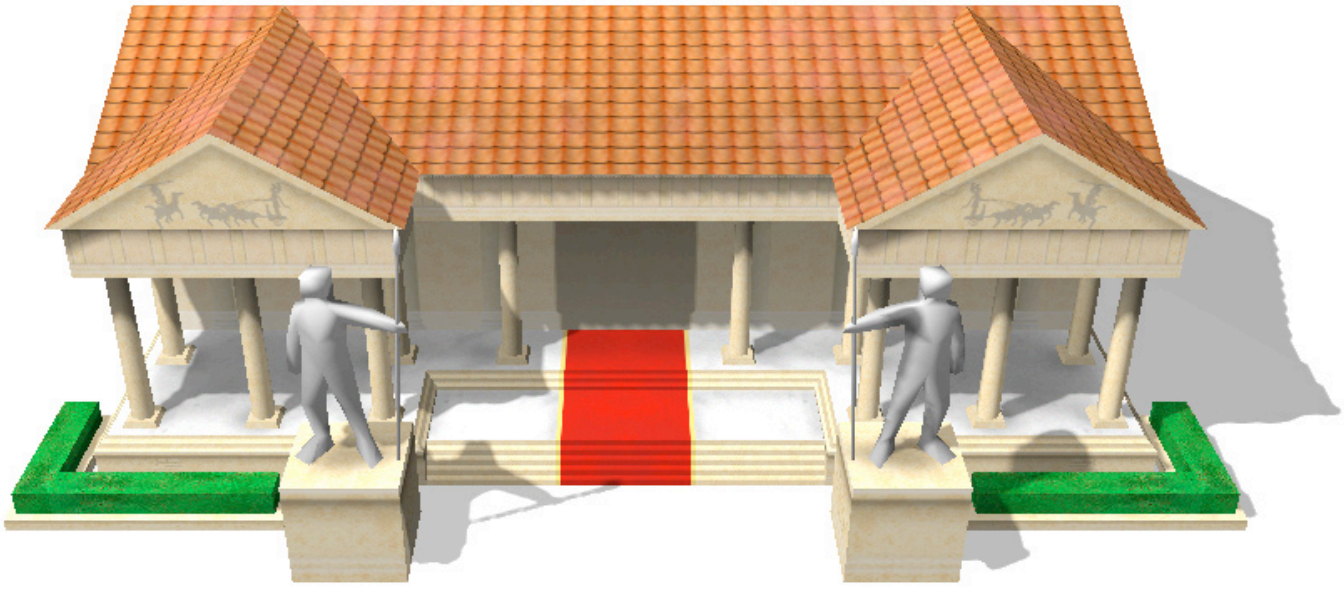
Bürger
541 Dreiecke

Bürger bewohnen die Stadt.Sie benötigen Wohnungen und Arbeitsplätze und zahlen Steuern an den Staat.

Reiter
793 Dreiecke

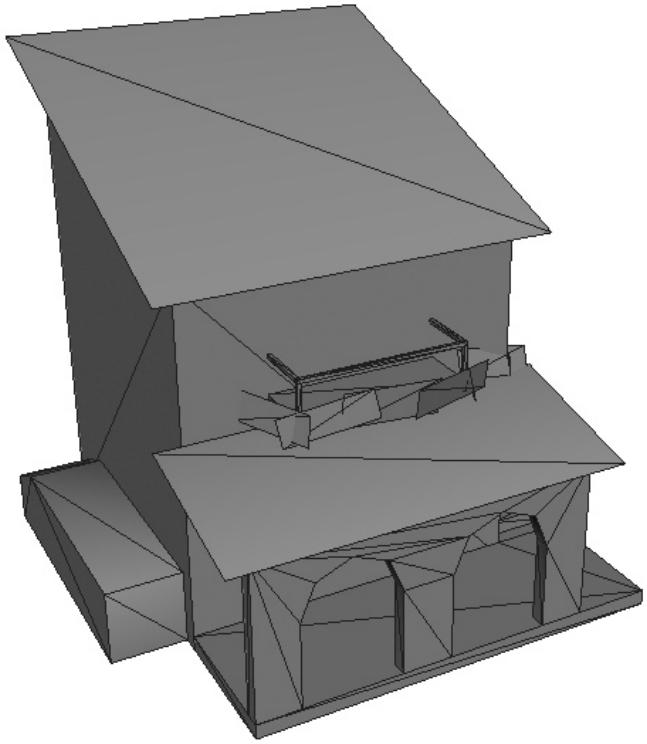
Berittene Einheiten sind im Vorteil gegenüber Fußsoldaten, da sie wesentlich schneller sind.





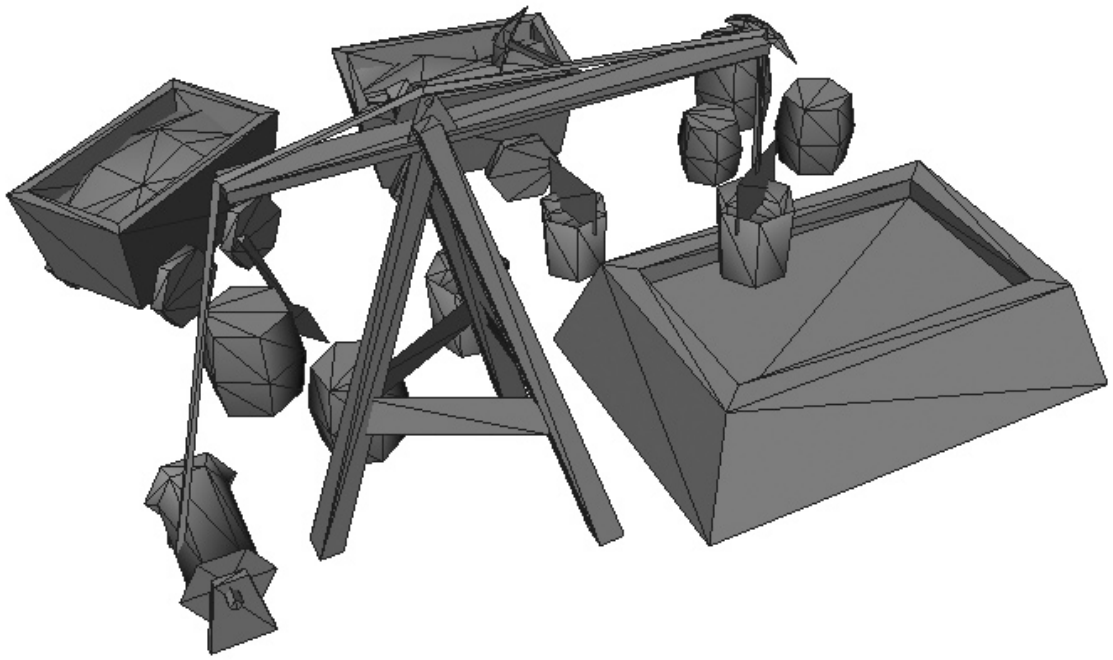
Rathaus
1112 Dreiecke

Das Herz einer Stadt ist das Rathaus. Von hier aus wird die Stadt geplant und die wichtigsten Entscheidungen werden getroffen.



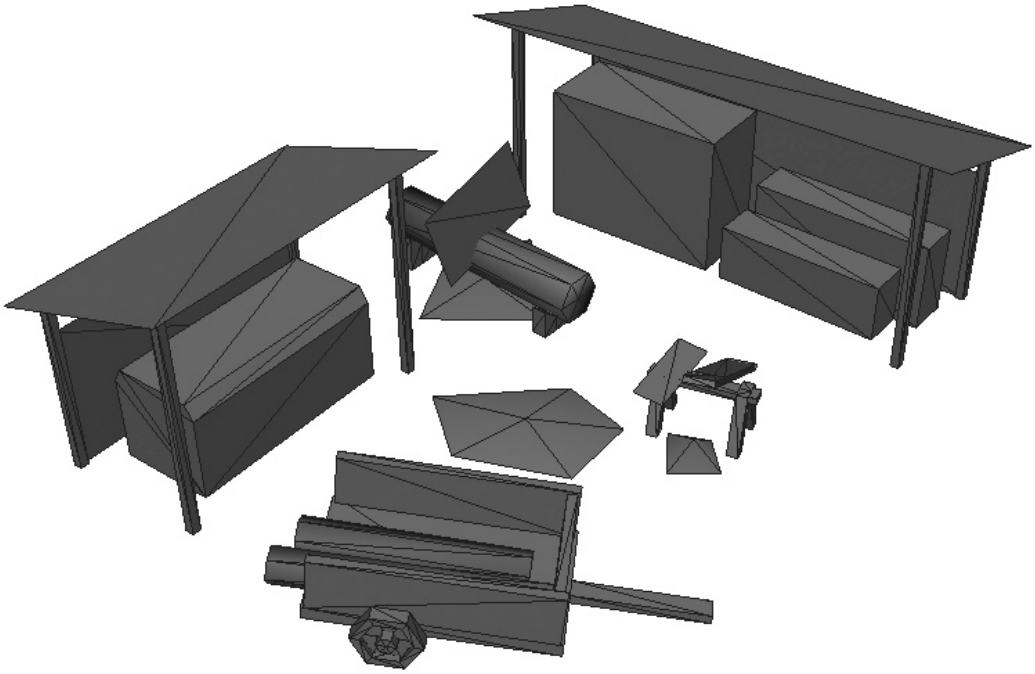
Familienhaus
310 Dreiecke

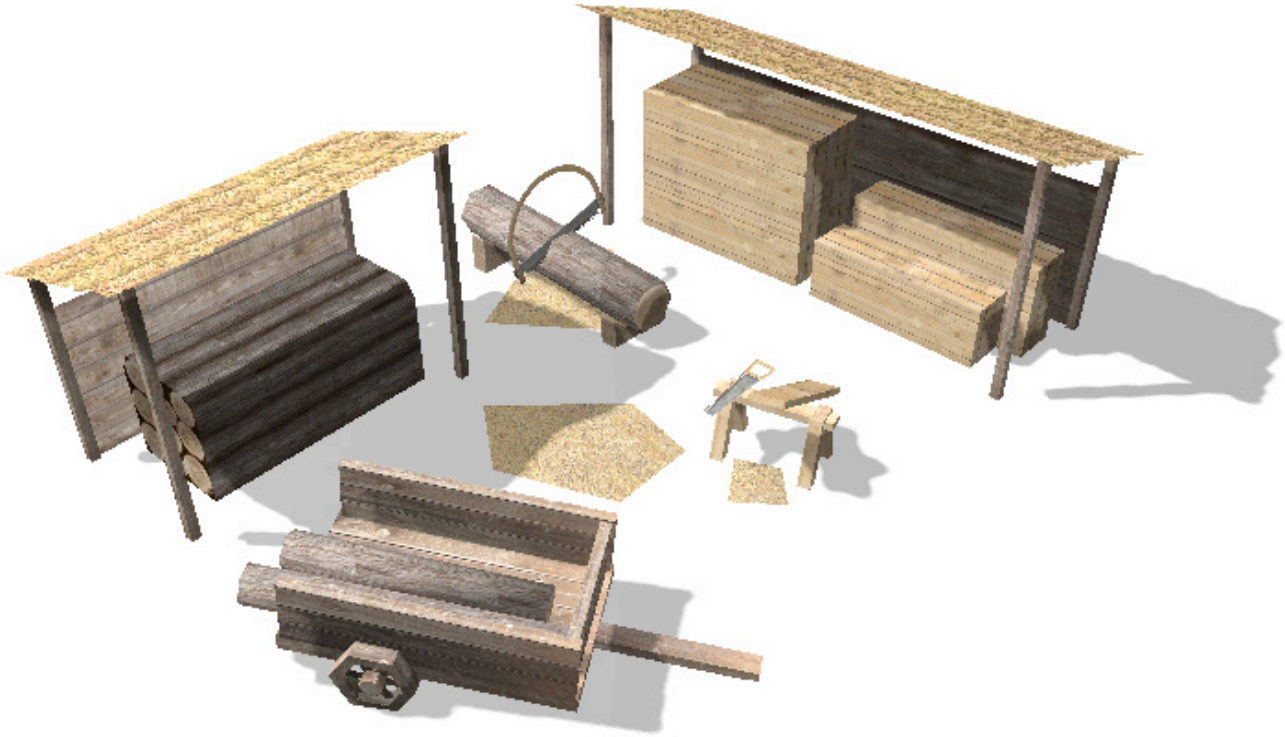
Befinden sich keine Häuser in der Stadt so kommen auch keine Bürger.



Mine
724 Dreiecke

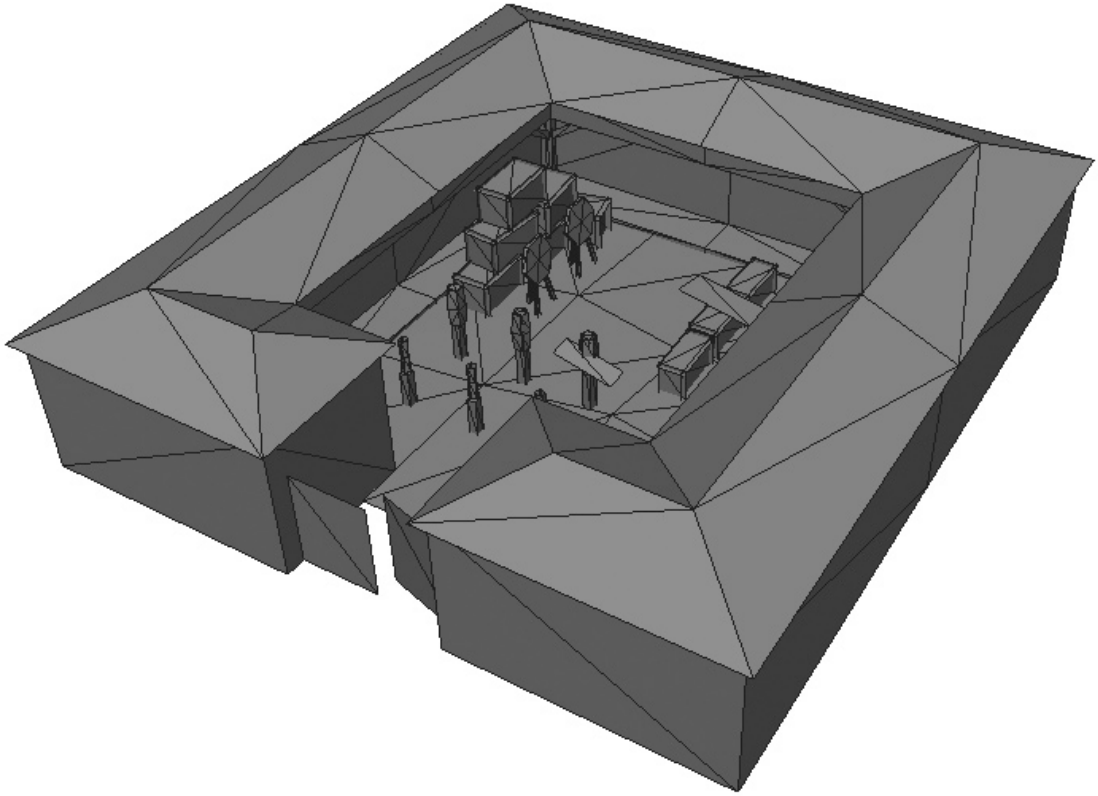
Die Mine dient als Lagerplatz für Gold, Metalle und Steine. Sie sollte in der Nähe eines Steinbruchs aufgestellt werden.

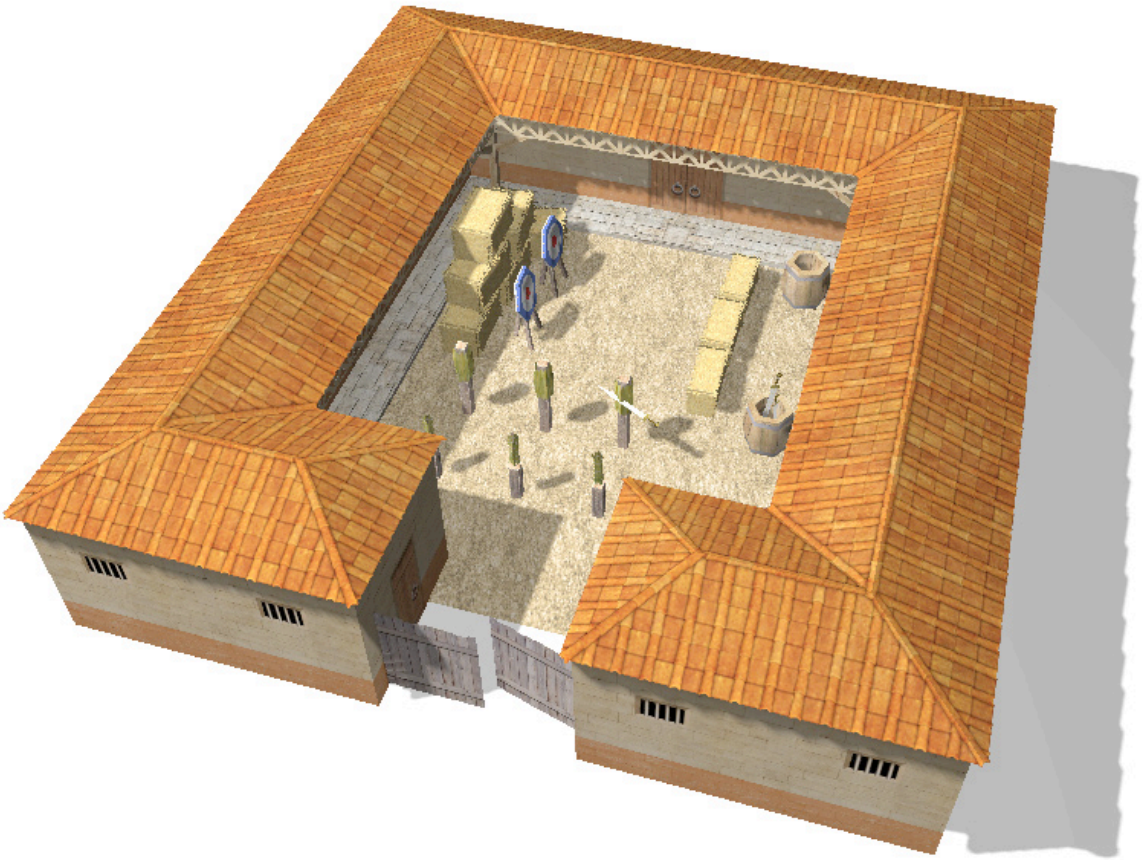




Sägewerk
461 Dreiecke

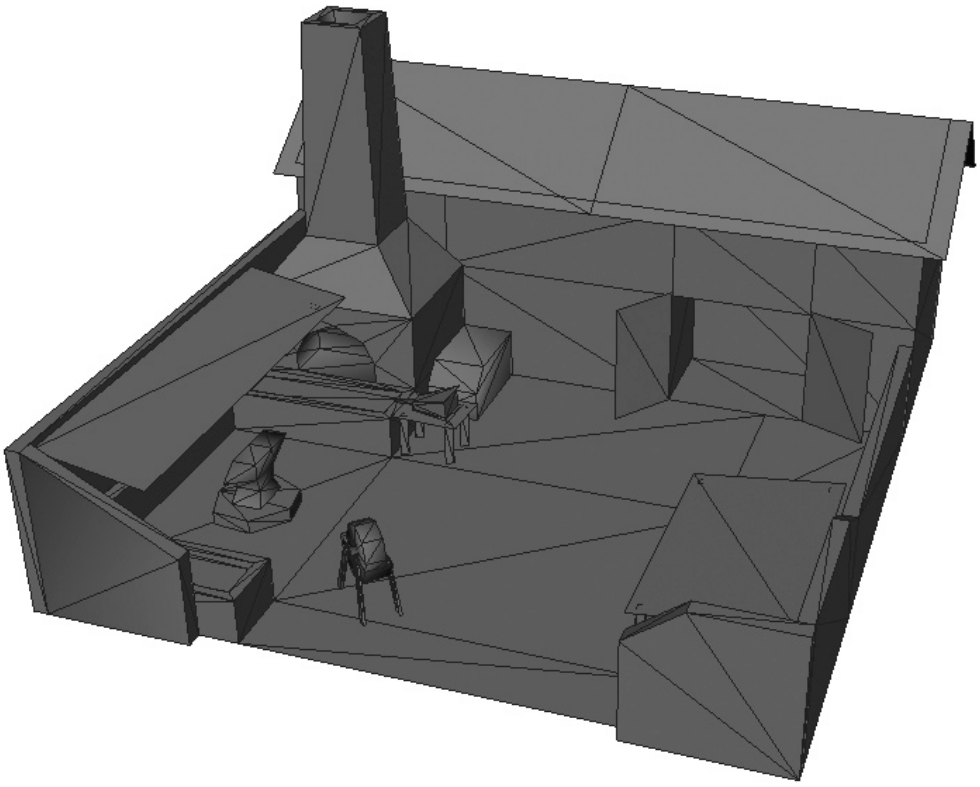
Ein Sägewerk dient der Lagerung, Verarbeitung und dem Transport von Holz. Es sollte am Waldesrand aufgebaut werden.





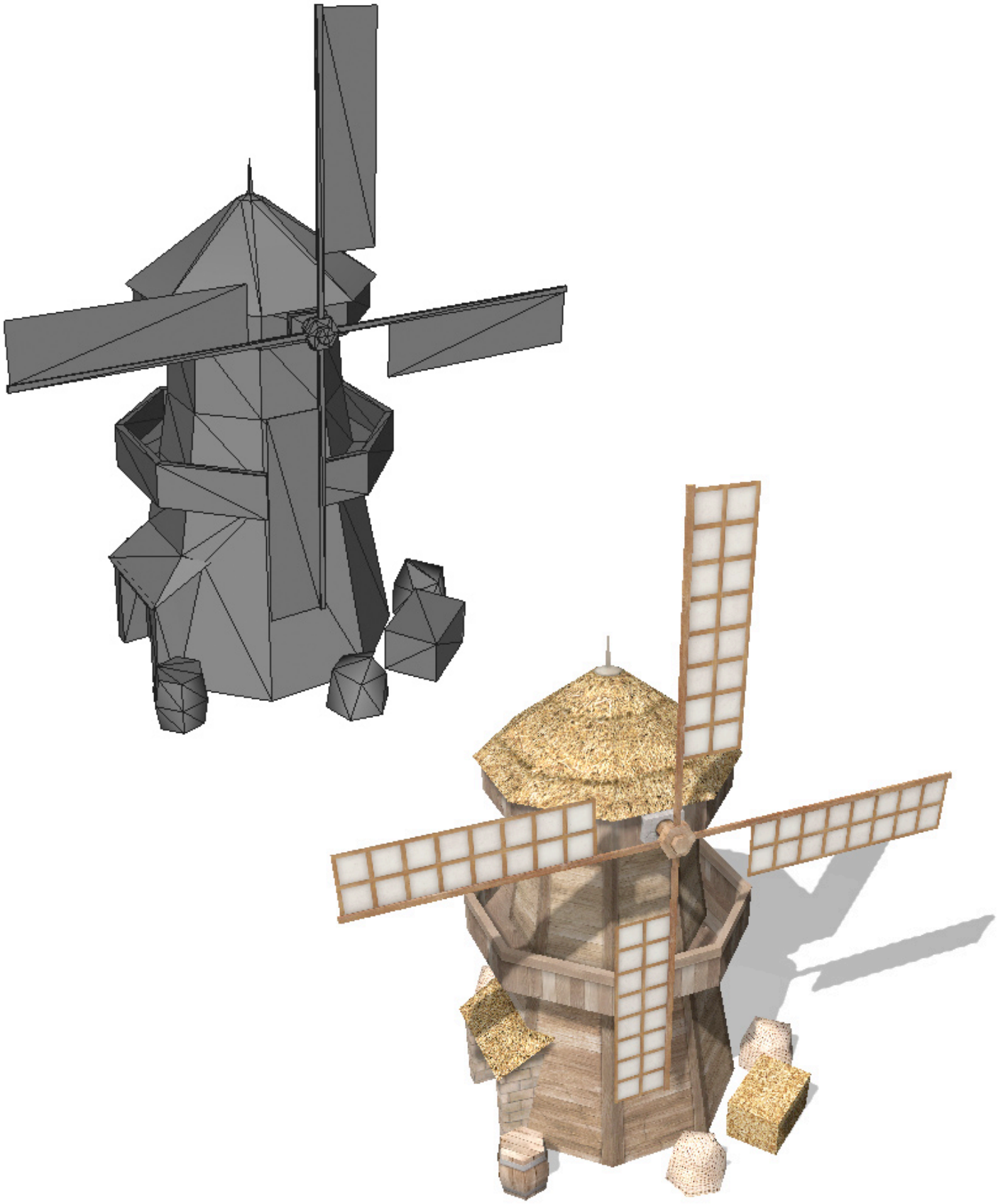
Kaserne
734 Dreiecke

In der Kaserne werden Fußsoldaten, Reiter und Bogenschützen ausgebildet. Hier kann auch der Sold der Einheiten festgelegt werden.



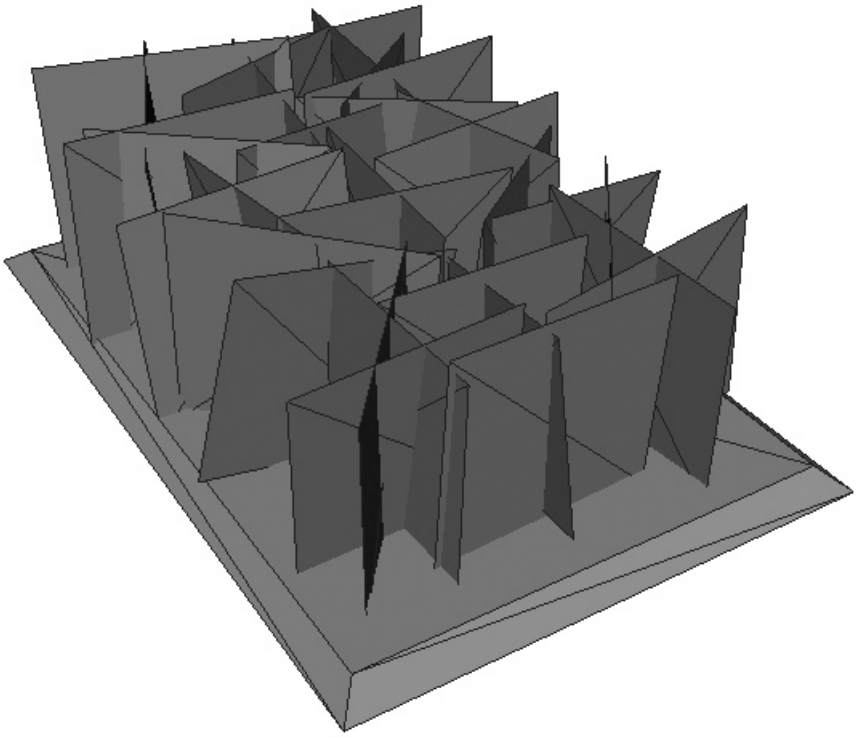
Schmiede
485 Dreiecke

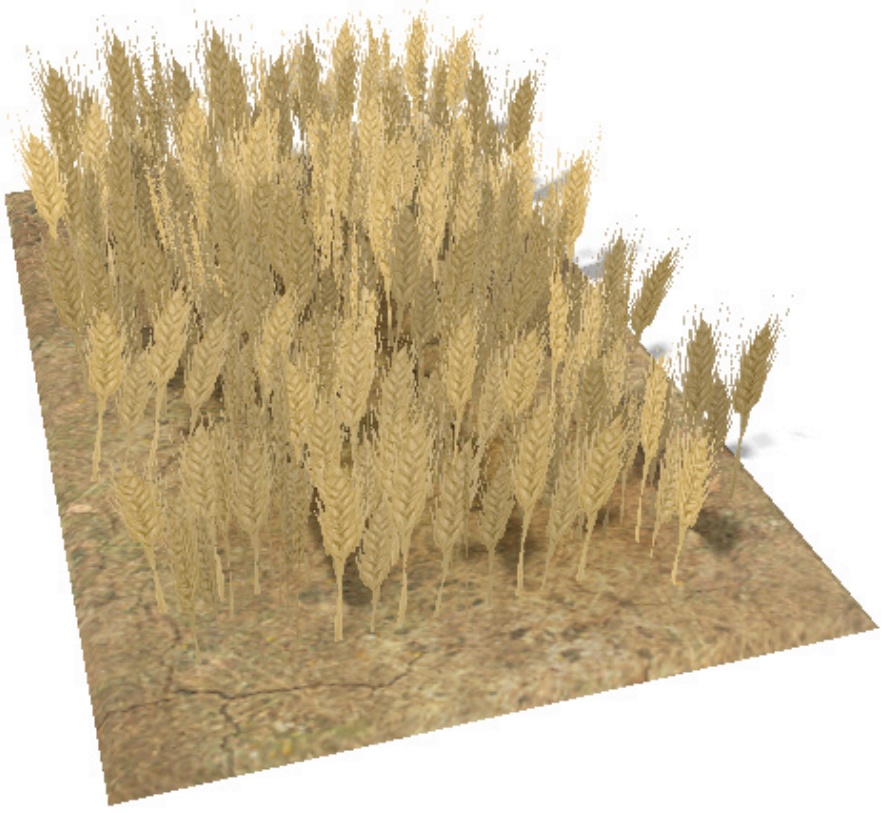
Die Schmiede dient der Herstellung von Waffen- und Rüstungsteilen ohne denen eine Armee nicht funktionieren kann.



Mühle
458 Dreiecke

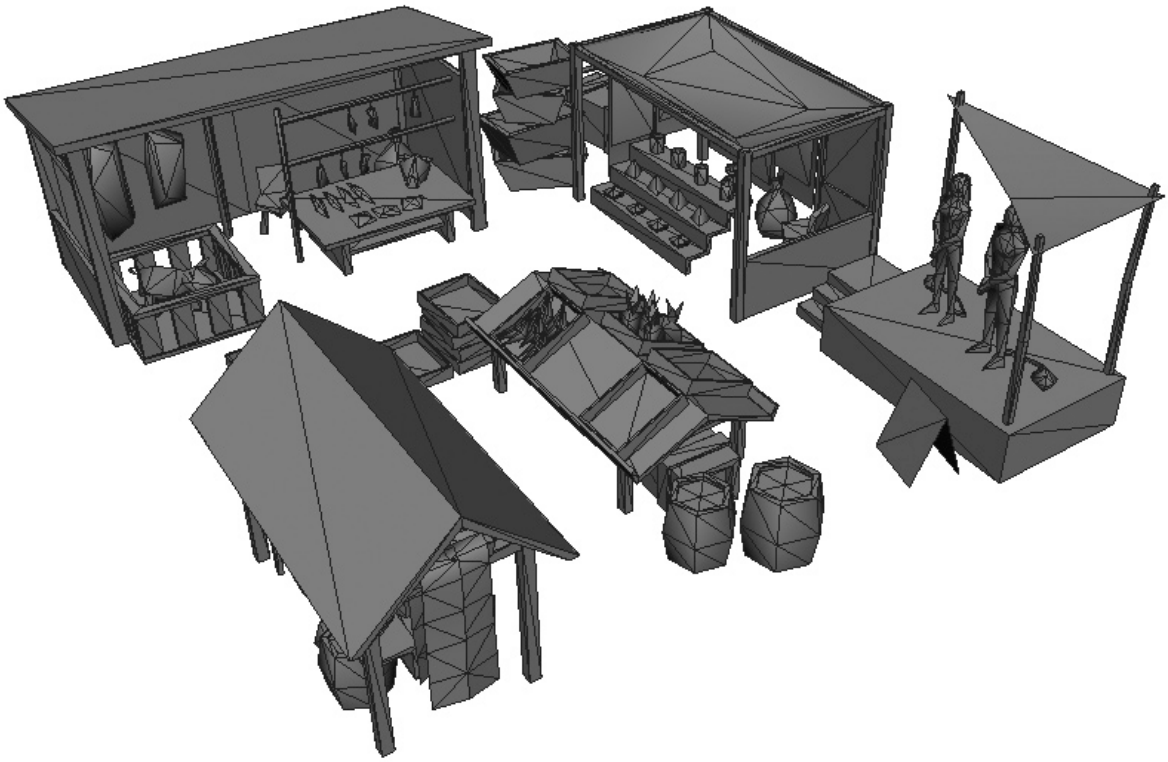
EIne Mühle ermöglicht das Errichten von Getreidefeldern und die Produktion von Nahrungsmitteln.





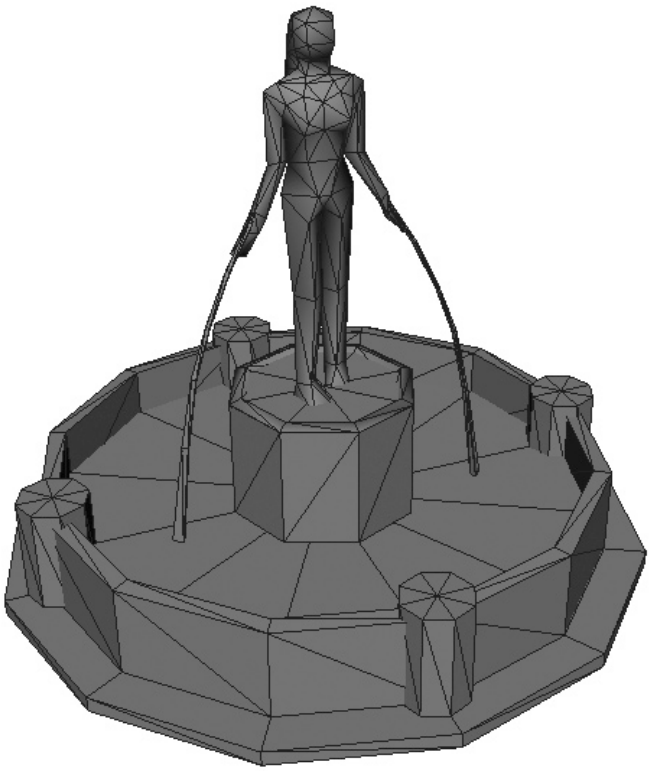
Getreidefeld
72 Dreiecke

Getreidefelder stehen meist in der Nähe von Mühlen und müssen von Arbeitern bewirtschaftet werden.



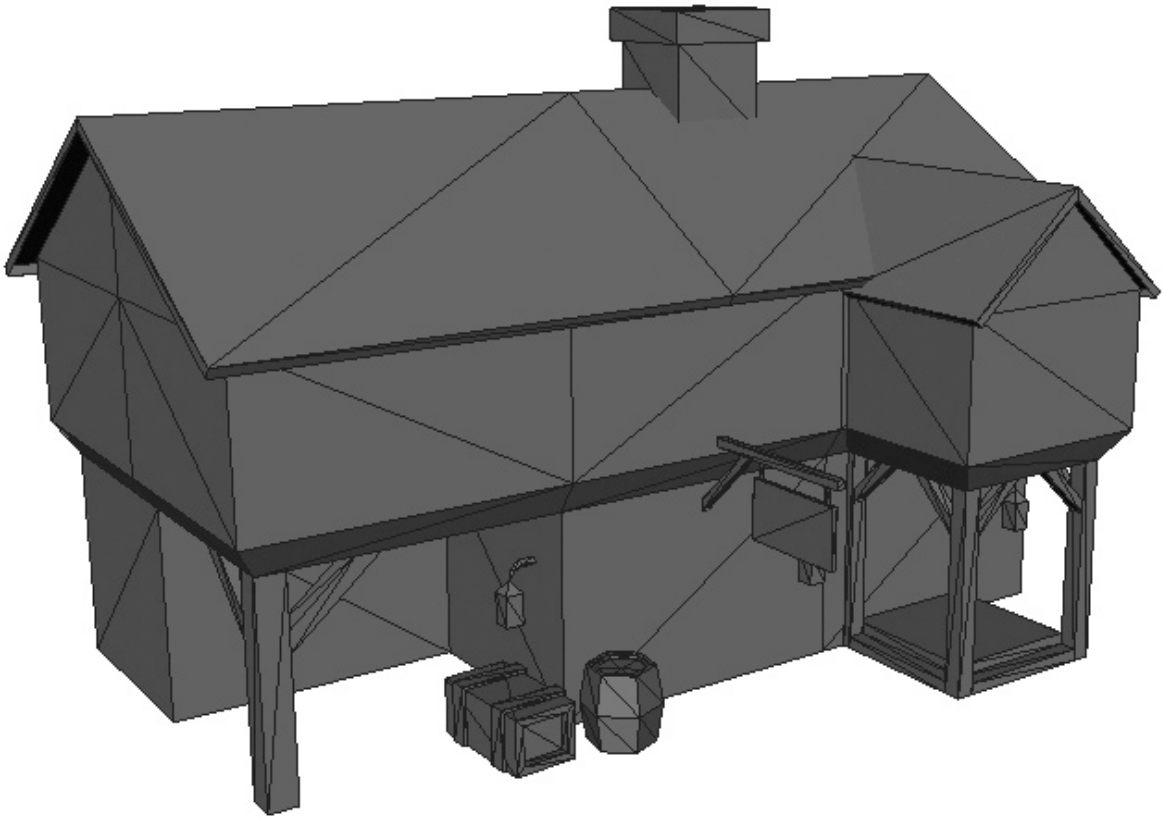
Markt
3768 Dreiecke

Auf dem Markt kommen Händler aus verschiedensten Ländern zusammen um ihre Güter und Waren zu verkaufen.



Brunnen
668 Dreiecke

Ein Brunnen dient der Trinkwasserversorgung eines Stadtteiles. Bürger möchten keine Stadt ohne Brunnen bewohnen.



Wirtshaus
699 Dreiecke

Im Wirtshaus können sich die Bürger nach einem harten Arbeitstag entspannen und bekommen Speis und Trank.