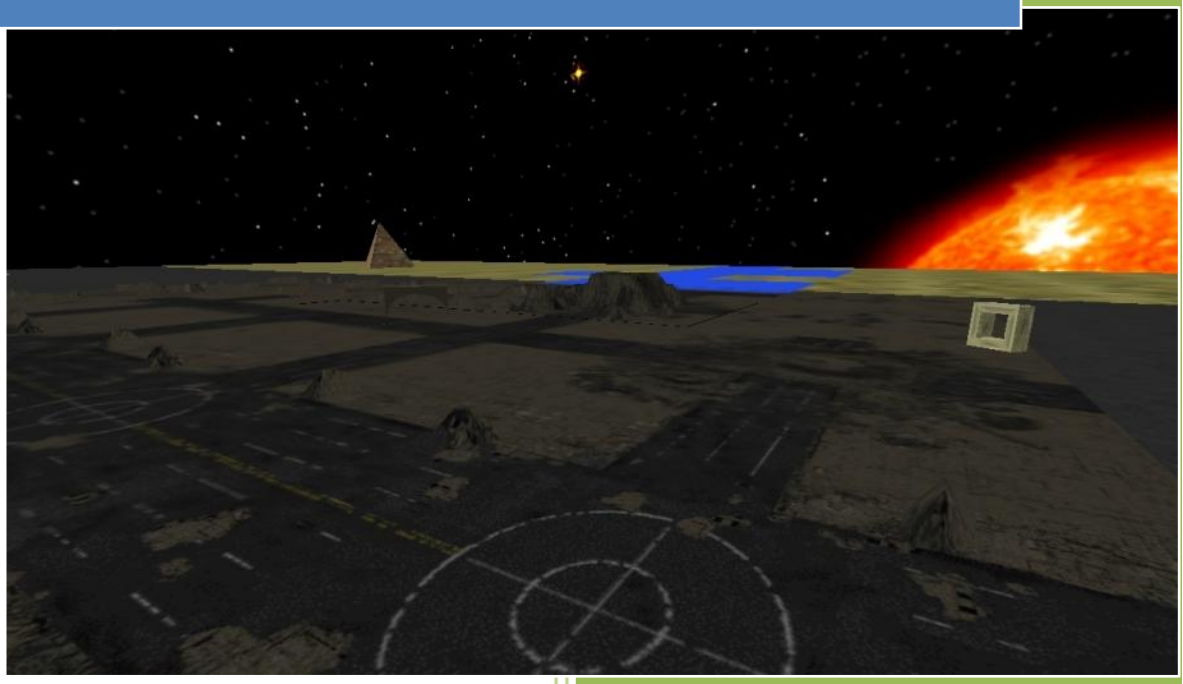
2008

Editor für 2,5D Welten aus Kachelelementen



Dominik Riehl

Diplomarbeit, Uni Kassel

Inhalt

1. Überblick

In dieser Arbeit werden die technischen Grundlagen vorgestellt und erläutert, die bei der Entwicklung eines Terrain-Editors benötigt werden. Ziel ist es, mit diesem 2,5D Welten erschaffen zu können, wie sie in Echtzeit-Strategie-Spielen (RTS) verwendet werden. Dabei wird das für RTS Spiele typische Konzept der „Welt-Kacheln“ verwendet. Das Terrain wird durch Aneinanderlegen mehrerer quadratischer Kacheln aufgebaut. Jede Kachel beinhaltet grafische und symbolische Informationen. Außerdem werden Möglichkeiten gezeigt, mit denen die topografische Struktur des Terrains geformt werden kann. Dies geschieht durch die Manipulation von Kacheln und der Verwendung von Heightmaps. Zusätzlich können geometrische Körper auf dem Terrain platziert und verändert werden. Mittels einer Speichermöglichkeit im XML-Format kann eine erschaffene Welt außerhalb des Editors weiterverwendet werden. Das Terrain wird während der Ausführung des Editors jederzeit voll gerendert in 3D dargestellt. Auch das Editieren erfolgt im 3D Raum. Mittels einer leicht zu steuernden Kamera kann das Terrain aus allen Perspektiven betrachtet werden. Der Editor wird mit Unterstützung einer 3D Engine entwickelt, der Irrlicht Engine. Dementsprechend wird in einigen Kapiteln Bezug auf Funktionen der Irrlicht Engine genommen, die nicht näher erläutert werden. Informationen dazu befinden sich in der Irrlicht Dokumentation. Da nicht auf Details der Engine eingegangen wird, können die in dieser Arbeit vorgestellten Techniken auch mit anderen Engines verwendet werden. Die Entwicklung erfolgt plattformunabhängig damit der Editor auf den gängigsten Plattformen (Linux, Mac OS X, Windows) lauffähig ist.

Für den Editor gibt es keinen speziellen Einsatzzweck. Er wird zwar so entwickelt, dass er gut für RTS-Spiele geeignet ist, aber einsetzbar ist er auch für beliebige Zwecke, in denen das Kachelkonzept von Vorteil ist. Die Kapitel sind so gestaltet, dass sie die technischen Grundlagen erklären und Ideen geben, aber genügend Freiraum für eine Spezialisierung des Editors für einen bestimmten Einsatzzweck lassen. Zusätzlich vorgestellte Ideen und Möglichkeiten sind in meinem Editor nicht umgesetzt.

In Kapitel 2 gibt es einen Einblick in die Historie von RTS-Spielen, sowie grundlegende Informationen wie Terrain ein aufgebaut sein kann. Es werden vor allem die Begriffe Kachel und RTS erläutert und ein Zusammenhang zwischen ihnen hergestellt.
In Kapitel 3 wird für die Entwicklung eine Programmiersprache und eine 3D-Engine ausgewählt. Entscheidend bei der Auswahl sind die im vorherigen Abschnitt gesteckten Ziele. Die Irrlicht Engine wird vorgestellt und ihre Vor- und Nachteile gegenüber alternativen Engines gezeigt.
Nachdem die Entwicklungsgrundlagen geschaffen sind, geht es in Kapitel 4 mit dem Grundaufbau des Editors und den wichtigsten Programmteilen (Klassen) weiter.
In Kapitel 5 werden einige Programmteile näher betrachtet, die für den grundlegenden Ablauf zuständig sind. Das Kapitel folgt dem Prinzip: Eingabe, Verarbeitung, Ausgabe. Da die Ausgabe grafisch erfolgt, werden in diesem Abschnitt alle Elemente erläutert, die selbst zu sehen sind oder für die Anzeige mitverantwortlich sind.
Kapitel 6 umfasst eine Reihe von Funktionen, die das Editieren der Welt überhaupt erst ermöglichen. Dazu zählt das Bearbeiten von Kacheln und die Verwendung von 3D Objekten und Heightmaps.
Abschließend wird in Kapitel 7 eine Möglichkeit zur Speicherung des Terrains mit allen Informationen konstruiert. Der Aufbau der verwendeten XML-Dateien wird im Detail dargestellt.

2. Einführung

In Echtzeit-Strategie-Spielen (RTS - Real-Time-Strategy) können Spieler (von Mensch oder Computer gesteuert) gleichzeitig ihre Aktionen durchführen, um einen Sieg gegen den Gegner zu erringen. Dies kann mit wirtschaftlichen, strategischen und taktischen Mitteln geschehen. Dazu kann der Spieler Gebäude bauen, mit denen er Einheiten ausbilden, Fahrzeuge bauen, Forschung betreiben oder Ressourcen verarbeiten kann. Einheiten haben verschiedene Aufgaben. Dazu zählen militärische Aktionen gegen generische Einheiten oder die Beschaffung von Ressourcen. Ressourcen werden für den Bau von Gebäuden und Fahrzeugen benötigt, sowie die Ausbildung von Einheiten. Die Aufgabe des Spielers ist es ausreichend Ressourcen zu sammeln und diese taktisch so einzusetzen, dass er einen militärischen Vorteil gegenüber seinen Gegnern erlangt. Die finale Aufgabe besteht meist in der Eroberung oder der Vernichtung der feindlichen Basis. Dabei muss gleichzeitig die eigene Basis vor Angriffen geschützt werden.

Abbildung 1: Dune 2, 1992, Entwickler: Westwood

Die ersten RTS-Spiele bestanden komplett aus 2D Grafiken. Das Geschehen wird von oben gerade herunter auf das Spielfeld betrachtet (top-down perspective, *Abbildung 1*). Das Spielfeld ist aus Kacheln aufgebaut von denen jede mit einer 2D Grafik belegt ist. So ergibt sich aus vielen einzelnen Grafiken das gesamte Feld. Die Einheiten und Gebäude werden ebenfalls durch 2D Grafiken repräsentiert, die vom Spieler auf den Kacheln platziert werden können.

Spiele wie Command & Conquer (*Abbildung 2*) boten eine isometrische Perspektive durch vorgerenderte 3D Grafiken. Die technische Umsetzung war weiterhin komplett zweidimensional, nur die optische Wirkung war dreidimensional:

Abbildung 2: Command & Conquer, 1995, Entwickler: Westwood Studios

Total Annihilation (*Abbildung 3*) war das erste RTS Spiel mit dreidimensionalen Einheiten. Die Welt ist dabei 2,5 dimensional aufgebaut:

Abbildung 3: Total Annihilation, 1997, Entwickler: Cavedog

2,5D bezeichnet eine Darstellung, bei der zwar alle Einheiten und Objekte dreidimensional umgesetzt sind, für den Betrachter aber der Eindruck entsteht sie wäre zweidimensional. Dieser Eindruck entsteht durch einen nicht veränderbaren Blickwinkel. Die Sicht auf das Terrain wird also auf eine Perspektive beschränkt. Ohne diese Einschränkung könnten die Objekte auf dem Terrain von allen Seiten betrachtet werden, wodurch ihr 3D Aufbau deutlich zu sehen wäre. Durch eine starre Perspektive fehlt dieser Effekt und ein vollständiger 3D Eindruck bleibt aus.

Der in dieser Arbeit entwickelte Editor soll es ermöglichen, Welten zu schaffen, die mit denen von Total Annihilation vergleichbar sind. Das Kachelkonzept ist mit dem aus „Dungeon Siege“ (Bilas) vergleichbar. Dort wird das Terrain ebenfalls aus 3D Kacheln aufgebaut, die leicht miteinander kombiniert werden können. Durch die Verwendung von Kacheln entstehen Einschränkungen in den Gestaltungsmöglichkeiten aber sie vereinfachen die Erstellung des Terrains auch wesentlich. So kann nach dem Baukastenprinzip schnell eine komplette Welt erschaffen werden. Die verwendeten

Bauteile können frei gewählt werden und sind unabhängig vom Editor. Der Editor stellt nur die Funktionen zur Verwendung bereit.

Terrain

Das Terrain ist das wichtigste und komplexeste Objekt eines RTS Spiels. In RTS Spielen müssen häufig die Aktionen für eine Vielzahl an Einheiten berechnet werden, wodurch die Performance eine große Rolle spielt. Besonders viel CPU Zeit wird für die Analyse des Terrains benötigt, wie z.B. durch Pathfinding-Algorithmen (Wegfindung). Daher ist es von Vorteil, wenn das Terrain so angelegt ist, dass die Analyse des Terrains möglichst wenig CPU Zeit in Anspruch nimmt.

Grundsätzlich kann zwischen zwei Typen von Terrain unterschieden werden. Dem kachelbasiertem Terrain und dem willkürlichem Terrain. Beim kachelbasiertem Terrain wird das Terrain in einzelne Bereiche, meist Quadrate unterteilt. Diese Art der Unterteilung hat ihren Ursprung in klassischen 2D Spielen. In *Abbildung 4* befindet sich im weiß umrandeten Bereich neun mal die gleiche Grafik. So war es möglich mit sehr wenig Speicher auszukommen, da komplette Gebiete mit wenigen kleinen Grafiken aufgebaut werden konnten, anstatt für jedes Gebiet eine neue Grafik zu verwenden, die sich über das komplette Gebiet erstreckt. Für heutige 3D Spiele hat die Einteilung einen noch viel wichtigeren Vorteil. Kacheln können nicht nur Grafiken bzw. Texturen zugeordnet werden, sondern auch logische Informationen. So kann also einem Gebiet, in dem sich die Abbildung eines Sees befindet auch die Information gespeichert werden, dass sich dort ein See befindet. Bei der späteren Analyse des Terrains kann so schnell festgestellt werden, dass dieser Weg nicht mit einfachen Mitteln passiert werden kann. Somit ist es für einen Pathfinding-Algorithmus möglich, alle Kacheln nach ihren Gegebenheiten zu befragen und so schnell den besten Weg zu ermitteln. Je nach Anzahl der Kacheln und der darin gespeicherten Menge an Informationen kann so der Speicherbedarf (des Terrains) schnell ansteigen, was ein Nachteil gegenüber einem willkürlichen Terrain sein kann.

Abbildung 4: 2D Spiel mit Kachelprinzip

Da für RTS Spiele die CPU Belastung für gewöhnlich ein größeres Problem ist als der Speicherbedarf, wird für diesen Editor das Kachelprinzip verwendet. Der Hauptvorteil ist aber die einfache Kombinierbarkeit der Kachelelemente und der damit verbundene schnelle Aufbau des Terrains.

3. Entwicklungsgrundlagen

Bei der Wahl der Entwicklungsumgebung sind eine Reihe an Faktoren zu beachten. Diese ergeben sich aus den in der Übersicht gesteckten Zielen. Die Wichtigsten sind:

- Erstellung einer Datenstruktur für Terrain und Kacheln
- Darstellung des Terrains in einem 3D Raum
- Entwicklung einer Kamerasteuerung
- Verwendung von 3D Objekten
- Verwendung von Texturen (Bild auf der Oberfläche von 3D Objekten)
- Verwendung von Heightmaps
- Lesen und Schreiben von XML-Dateien
- Plattformunabhängigkeit

Die Umsetzung dieser Ziele ist sehr aufwendig wenn sie manuell durchgeführt werden. Daher wird eine 3D-Engine verwendet, die Funktionen und Datenstrukturen zur schnellen Lösung vieler dieser Probleme bereit stellt. Eine 3D-Engine ist eine Programmbibliothek die Implementierungen beinhaltet, die in 3D Programmen häufig benötigt werden. Sie stellt eine breite Palette an grafischen Funktionen zur Verfügung. Bei vielen Engines gibt es neben diesen Funktionen noch weitere Funktionen für Sound, Input und Netzwerk.

Neben dem Funktionsumfang ist bei der Auswahl der Engine die verwendete Programmiersprache wichtig. Engines werden speziell für eine bestimmte Programmiersprache entwickelt. Nur wenige unterstützen mehrere Sprachen. Die Sprachen mit der größten Unterstützung sind C++ und Java. Beide Sprachen sind objektorientiert und geeignet zur Entwicklung einer 3D Anwendung.

Die Wahl fiel auf die Irrlicht Engine, da sie plattformunabhängig ist und für eine kostenfreie Engine einen enormen Funktionsumfang besitzt. Sie ist eine „open source“ Entwicklung, wodurch ihr Quellcode einsehbar und veränderbar ist. Neben den Grafikfunktionen werden auch Funktionen für Maus und Tastatureingaben und das Dateisystem verwendet. Die Irrlicht Engine ist in C++ geschrieben, wodurch auch für die Entwicklung des Editors C++ zum Einsatz kommt. Zur Code-Dokumentation wird Doxygen verwendet und in UnitTest++ werden Tests für den Code entwickelt.

Irrlicht Engine

Irrlicht ist eine Echtzeit 3D-Engine. Sie ist in C++ geschrieben und auch damit verwendbar. Wie auch in anderen 3D-Engines wird ein Szenengraph verwendet. Ein Szenengraph ist eine baumartige objektorientierte Datenstruktur, der die logische Struktur der darzustellenden Szene enthält. Der Graph besteht aus einzelnen Szenenknoten, die hierarchisch miteinander verbunden werden. Der Wurzelknoten beinhaltet die gesamte Szene. Die Kindknoten der Wurzel repräsentieren Objekte der Szene und können selbst weitere Kindknoten besitzen. Je nach Typ können Objekte eine Transformationsmatrix besitzen. Mit dieser lässt sich das Objekt verändern. Besteht ein Objekt aus mehreren kleinen Objekten müssten alle Transformationsmatrizen einzeln verändert werden. Durch die Struktur des Szenengraphs ist es möglich, nur den Elternknoten zu verändern und alle Kindknoten automatisch anpassen zu lassen.

Irrlicht unterstützt Vertex und Pixelshader und mittels der integrierten Partikelengine können Partikeleffekte erzeugt werden. Auch 2D Grafiken können dargestellt werden. Mit dem GUI-System können Fenster und Menüs mit allen gängigen GUI Elementen erstellt werden. Außerdem werden Datenstrukturen für Matrizen und Vektoren bereitgestellt, sowie mathematische Funktionen zum Rechnen mit diesen.

Irrlicht unterstützt eine Vielzahl an Dateiformaten für Drahtgittermodelle (3D-Objekte) und Texturen (Bilddateien). Außerdem kann direkt aus Archiven (.zip) gelesen werden. Für XML-Dateien wird ein eigener Parser bereitgestellt, mit dem die Dateien eingelesen und geschrieben werden können.

Die Irrlicht Engine unterstützt mehrere Grafik APIs. Es empfiehlt sich die Verwendung von OpenGL, da es von den gängigsten Plattformen unterstützt wird. Unter Windows ist die Verwendung von DirectX 8/9 möglich. Für Systeme mit schwächerer Grafikhardware kann ein Software Renderer verwendet werden. Mit diesem sind die Darstellungsmöglichkeiten allerdings eingeschränkt.

Der Quellcode der Engine ist offen einsehbar und steht unter der zlib-Lizenz, die eine kommerzielle Nutzung ermöglicht, ohne den eigenen Quellcode herausgeben zu müssen.

4. Grundaufbau des Editors

Nach der Auswahl der Entwicklungsumgebung kann jetzt die eigentliche Programmierung beginnen. Dazu wird als erstes ein Konzept über den Aufbau des Editors erstellt. Das Konzept verschafft einen Überblick über die Elemente die programmiert werden und ihre Hierarchie. Ein Element kann ein logisches oder visuelles Objekt sein, aber auch einen bestimmten Funktionsbereich abdecken. Die Hierarchie ist nur beispielhaft und kann mitunter abhängig von der Engine sein.

Abbildung 5 zeigt die wichtigsten Elemente und deren Position in der Gesamtstruktur. Die Struktur ist so angelegt, dass jedes Element Zugriff auf seine Kindelemente hat. Der Operator kann also z.B. auf die Dateiverwaltung zugreifen. Umgekehrt ist dies jedoch nicht möglich. Bei Elementen einer Ebene kann eine Zugriffsmöglichkeit bei Bedarf erstellt werden. Der Operator benötigt z.B. Zugriff auf die Eingabeverwaltung. Das Element „Editor“ ist das Wurzelelement, das Zugriff auf alle Elemente besitzt und in dem sich die Hauptschleife des Programms befindet.

Alle Elemente werden in den weiteren Kapiteln noch im Detail erklärt. Im folgenden Abschnitt gibt es einen kurzen Überblick über ihre Bedeutung bzw. Aufgabe.

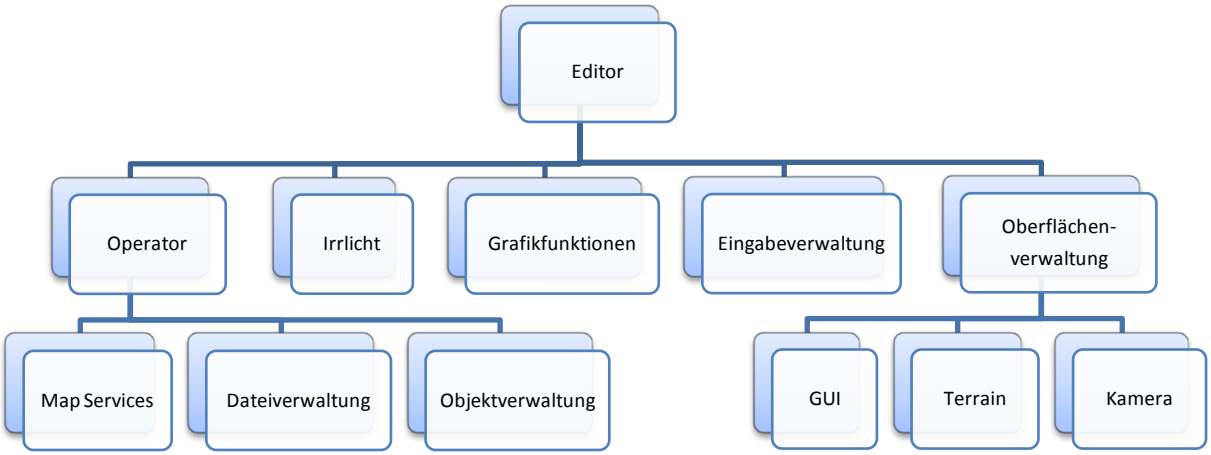


Abbildung 5: logischer Aufbau des Editors

Operator

Der Operator stellt die Hauptsteuereinheit dar. Hier werden alle Benutzereingaben ausgewertet und dazugehörige Funktionen aufgerufen.

Map Services

Die Map Services sind Funktionen für alle Veränderungen, die auf der Karte durchgeführt werden sollen. Es werden alle Funktionen verarbeitet, die im Abschnitt „Editierfunktionen“ vorgestellt werden.

Dateiverwaltung

Die Dateiverwaltung kann Dateien erstellen und diese speichern, sowie vorhandene Dateien einlesen. Sie wird vor allem für das Schreiben und Lesen von XML-Dateien im Abschnitt „Laden/Speichern“ benötigt.

Objektverwaltung
Die Objektverwaltung hält alle verfügbaren 3D Objekte zur Verwendung bereit und verfügt über Funktionen, um diese zu laden und zu erstellen.

Irrlicht
Die Irrlicht Engine wird durch ein eigenes Objekt repräsentiert. Über dieses kann auf alle benötigten Funktionen zugegriffen werden.

Grafikfunktionen
Die Grafikfunktionen aktualisieren und zeichnen jede Szene. Dazu werden als Grundlage die Grafikfunktionen der Irrlicht Engine verwendet. Neben diesen Funktionen können noch zusätzliche Funktionen für programmspezifische Anzeigen erstellt werden.

Eingabeverwaltung
Alle Eingaben werden von der Irrlicht Engine entgegen genommen und an die Eingabeverwaltung weitergeleitet. Diese muss alle relevanten Eingaben zur späteren Verarbeitung durch den Operator speichern.

Oberflächenverwaltung
Die Oberflächenverwaltung verwaltet alle Elemente, die auf dem Bildschirm dargestellt werden sollen bzw. für die Darstellung (Kamera) notwendig sind. Dazu zählt sowohl der 3D Raum, in dem sich das Terrain befindet als auch die 2D Oberfläche der GUI.

GUI
Die GUI umfasst alle Fenster der Oberfläche. Dazu gehören Fenster zur Funktionsauswahl, für Grafiken, ein Menü und eine Informationsanzeige.

Terrain
Das Terrain ist die Grundlage für die Welt, die erstellt werden soll.

Kamera
Um das Terrain im 3-dimensionalen Raum betrachten zu können, wird eine Kamera benötigt.

5. Hauptelemente

Im letzten Kapitel wurde ein Konzept der Editorelemente erstellt. Aus diesem werden jetzt die für viele Programme wichtigen Elemente implementiert und zwar die Eingabe (Eingabeverwaltung), die Verarbeitung (Operator) und die Ausgabe (Oberflächenverwaltung). Im Abschnitt der Oberflächenverwaltung geht es dabei mehr um den Aufbau der Szene als um die eigentliche Ausgabe dieser. Die Ausgabe wird mit wenigen Funktionsaufrufen von der Irrlicht Engine erledigt.

5.1. Eingabeverwaltung

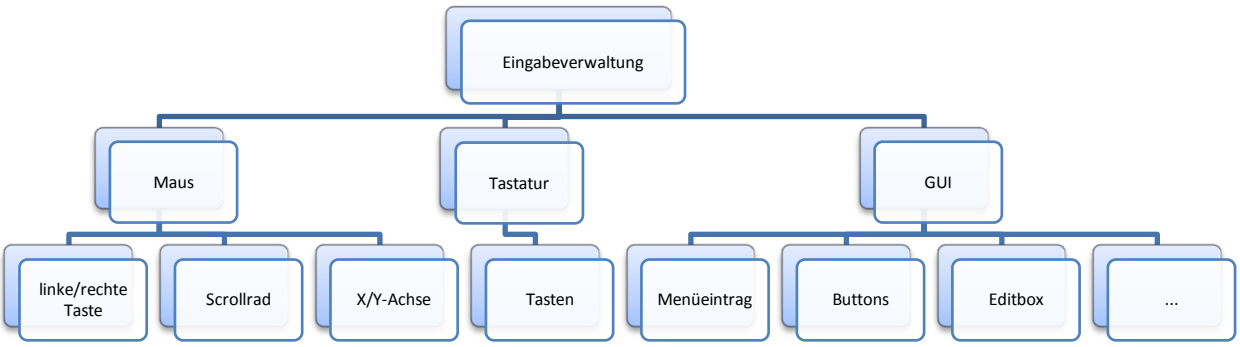
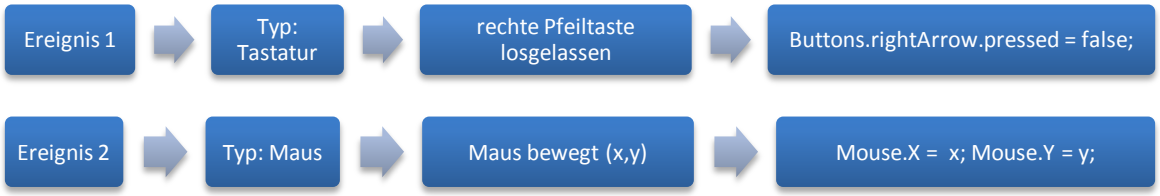


Abbildung 6: logische Struktur der Eingabeverwaltung

Die Erkennung aller Eingaben wird von der Irrlicht Engine vorgenommen. Für die Entwicklung des Editors sind drei Kategorien von Eingaben relevant. Mausaktionen, Tastaturaktionen und Aktionen in der GUI. Bei jeder ausgelösten Aktion löst Irrlicht ein Ereignis aus und sendet dieses an die Eingabeverwaltung. Ein Ereignis ist eine Art Nachricht, in der steht, was für eine Aktion der Benutzer durchgeführt hat. Um diese Nachrichten zu empfangen, muss vorher dem Irrlicht Objekt (siehe Kapitel 4) mitgeteilt werden, dass die Eingabeverwaltung für den Empfang zuständig ist.

Nach Empfang einer Nachricht ist es Aufgabe der Eingabeverwaltung, diese Nachricht auszuwerten. Dazu muss im erstem Schritt der Ereignistyp überprüft werden (Maus, Tastatur, GUI) und anschließend die genaue Aktion. *Abbildung 6* zeigt eine Reihe an möglichen Aktionen. Vor allem GUI Aktionen gibt es aber noch viele weitere. Die folgenden zwei Beispiele zeigen wie Ereignisse aussehen können und wie die Eingabeverwaltung diese nach der Auswertung speichert.



Manche Aktionen führen zu mehreren Ereignissen. So löst ein Klick auf einen Button der GUI einmal das Ereignis „Mausklick“ und zusätzlich das Ereignis „GUI Button aktiviert“ aus. Beide Ereignisse werden von Irrlicht in separaten Nachrichten an die Eingabeverwaltung gesendet und können einzeln ausgewertet werden.

Verteilte Eingabeverwaltung

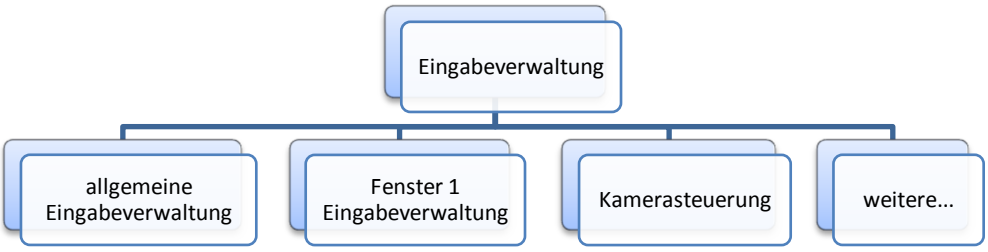


Abbildung 7: logische Struktur der verteilten Eingabeverwaltung

Mit Irrlicht ist es nur möglich, genau ein Objekt zum Empfangen von Ereignissen zu registrieren. Da die Anzahl der möglichen Ereignisse sehr groß werden kann, wird die entsprechende Klasse sehr unübersichtlich. Daher kann es von Vorteil sein, einige GUI Ereignisse direkt dort zu verwalten, wo sich das zugehörige Element befindet. Wird z.B. ein Button in einem Fenster gedrückt, könnte das Ereignis an die Eingabeverwaltung des Fensters weitergeleitet werden. So wäre nur noch eine abgespeckte Eingabeverwaltung für alle allgemeinen Ereignisse notwendig.

Zur Umsetzung wird eine Eingabeverwaltung benötigt, die bei Irrlicht registriert wird und alle Ereignisse entgegen nimmt. Diese werden aber nicht ausgewertet, sondern einfach an die einzelnen Eingabeverwaltungen weitergeleitet. Diese müssen dazu vorher bei der Haupteingabeverwaltung registriert werden. Nach Auswertung eines Ereignisses geben sie an die Haupteingabeverwaltung eine Antwort zurück, ob das Ereignis vollständig abgearbeitet wurde. Ereignisse können nur vollständig abgearbeitet werden, wenn es sich um spezielle Ereignisse wie z.B. das Aktivieren eines GUI Elements handelt. Allgemeine Ereignisse wie z.B. die Bewegung der Maus können hingegen für mehrere Eingabeverwaltungen relevant sein. In diesem Fall wird das Ereignis noch an die allgemeine Eingabeverwaltung weitergegeben. Im Operator, der für die weitere Verarbeitung der Ereignisse zuständig ist, werden nur Ereignisse der allgemeinen Eingabeverwaltung verarbeitet. Die Verarbeitung aller weiteren Ereignisse findet direkt in den jeweiligen Eingabeverwaltungen statt.

5.2. Operator

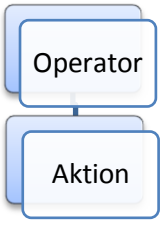


Abbildung 8: ***logische Struktur des Operators***

Der Operator ist die zentrale Steuereinheit des Programms. Er holt sich von der Eingabeverwaltung die neusten Ereignisse und wertet diese aus. Anschließend werden alle notwendigen weiterverarbeitenden Funktionen aufgerufen. Einige Ereignisse lösen spezielle Aktionen aus (siehe Kapitel 6). Nach Auslösen einer Aktion wird diese gespeichert und die folgenden Ereignisse werden in Bezug zu dieser Aktion ausgewertet. So führen also z.B. Mausklicks auf das Terrain zu unterschiedlichen Ergebnissen.

5.3. Oberflächenverwaltung

Aufgabe der Oberflächenverwaltung ist die Verwaltung aller Elemente, die für die Darstellung des Terrains und aller Editierfunktionen notwendig sind. Sie umfasst also alles, was bei Ausführung des Editors auf dem Bildschirm zu sehen sein soll. Im Folgenden wird der Aufbau des Terrains sowie der Kacheln vorgestellt. Außerdem wird eine Kamera erstellt, mit der das Terrain betrachtet werden kann. Zuletzt werden dem Editor GUI Elemente hinzugefügt die das Editieren überhaupt erst ermöglichen. Zuerst wird jedoch das für die Terrain Erstellung notwendige Koordinatensystem definiert.

Koordinatensystem

Mit der Wahl des Koordinatensystems wird festgelegt, nach welchen Koordinaten die Elemente im Raum ausgerichtet sein sollen. Erfolgt dies nicht einheitlich, könnten Kacheln in unterschiedliche Richtungen stehen oder die Kamera das Terrain von der falschen Seite aus zeigen. Das Koordinatensystem selbst wird von Irrlicht vorgegeben und besteht aus den drei Achsen x, y und z. Theoretisch kann das Terrain an jeder beliebigen Stelle platziert werden, solange sich die Kamera in gleicher Ausrichtung befindet. Um aber möglichst einfache Koordinaten zu erhalten, empfiehlt es sich das Terrain in eine der Koordinatenebenen zu legen und die erste Kachel im Ursprung (Nullpunkt) zu platzieren. Ich entscheide mich für die x-y Ebene mit positiven x-Werten und negativen y-Werten wie in *Abbildung 9* dargestellt. Der obere Bereich des Terrains läuft also entlang der x-Achse und der linke Bereich befindet sich entlang der y-Achse. Die Kamera befindet sich im negativen z-Bereich. Alle verwendeten Koordinaten und Formeln der folgenden Kapitel sind so gewählt, dass das Terrain wie gezeigt platziert wird.

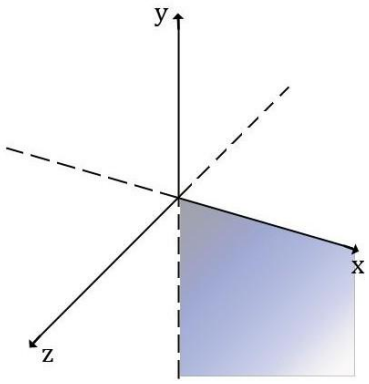


Abbildung 9: verwendetes Koordinatensystem

Terrain

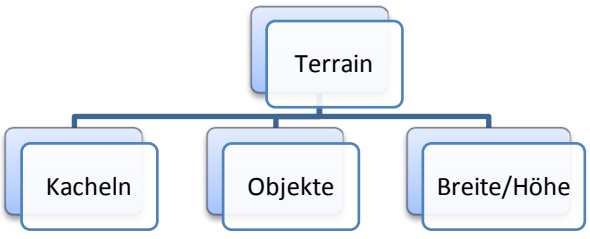


Abbildung 11: logische Struktur des Terrains

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

Abbildung 10: Aufbau eines Terrains mit 16 Kacheln

Um ein Terrain zu erhalten, das leicht und schnell zu editieren ist, wird es aus quadratischen Kacheln aufgebaut. Diese werden in mehreren Reihen und Spalten nebeneinander gelegt. So ergibt sich ein rasterartiger Aufbau. Die notwendigen Informationen für den Aufbau des Terrains sind also die maximale Anzahl an Kacheln pro Reihe und Spalte sowie die Anzahl der Kacheln die sich daraus ergibt. Das Beispiel Terrain in *Abbildung 10* besteht aus 16 Kacheln, die sich in vier Reihen und vier Spalten befinden. Jeder Kachel wird nach dem gezeigten System eine eindeutige Nummer zugeordnet, um eine leichte Identifikation zu ermöglichen.

Für einige Berechnungen wird die Breite und die Höhe des Terrains benötigt. Die Breite berechnet sich aus dem Produkt der Kachelgröße und der Anzahl an Kacheln in einer Reihe (x-Achse). Die Höhe berechnet sich aus dem Produkt der Kachelgröße und der Anzahl an Kacheln in einer Spalte (y-Achse).

Kachel

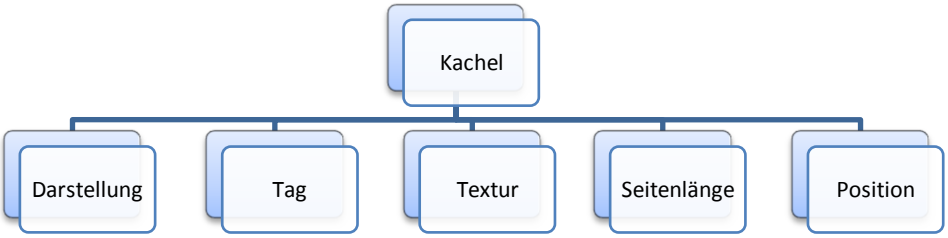


Abbildung 12: logische Struktur einer Kachel

Eine Kachel ist das kleinste Element des Terrains. Jede Kachel erhält Tags, um ihr logische Informationen zuordnen zu können. Außerdem können sie mit einer Textur belegt werden, um ihre optische Erscheinung zu verändern. Jede Kachel sollte standardmäßig mit einer neutralen Textur belegt sein, damit die Kacheln voneinander zu unterscheiden sind. Empfehlenswert ist z.B. eine weiße Textur mit einer schwarzen Umrandung. So entsteht ein Muster wie in *Abbildung 10*. Sowohl das Tag als auch die Textur müssen im Editor jederzeit veränderbar sein.

Um eine Kachel im Raum darstellen zu können, benötigt sie einen 3-dimensionalen Aufbau. Zu beachten ist, dass Kacheln selbst nur 2-dimensional sind, da sie keine Tiefe besitzen. Eine Kachel besteht aus vier Eckpunkten (Vertices). Jedem Vertex (Einzahl von Vertices) wird eine Koordinate im Raum zugeordnet. Der Abstand zweier benachbarter Vertices muss der Seitenlänge der zu bildenden Kachel entsprechen. Um eine quadratische Form zu erhalten, muss die Länge aller Seiten gleich sein. *In Abbildung 13* wird für die Seitenlänge der Buchstabe l verwendet. Die z-Koordinate (Höhe) ist aufgrund der fehlenden Tiefe immer null.

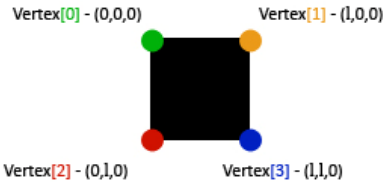


Abbildung 13: eine Kachel mit den Koordinaten ihrer Eckpunkte

Aus den Vertices werden zwei Polygone erstellt, die zusammen die Kachel formen. Polygone haben in Irrlicht immer eine Dreiecksform. Um ein Polygon bilden zu können, müssen die Vertices miteinander verbunden werden. Eine mögliche Reihenfolge ist in *Abbildung 14* dargestellt:

blaues Dreieck: 0,2,3
grünes Dreieck: 0,1,3

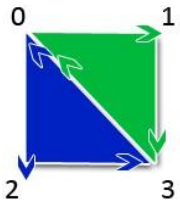


Abbildung 14: eine Kachel aufgebaut aus zwei Dreiecken

Da nun die Kacheln definiert sind, können sie im Raum platziert werden. Um die Koordinaten möglichst einfach zu halten, empfiehlt es sich die erste Kachel im Nullpunkt zu positionieren (mit der linken oberen Ecke). Die zweite Kachel wird rechts (in x-Richtung, sie *Abb. 9* und *10*) neben ihr

angeordnet, also mit den Koordinaten $\begin{pmatrix} Kachelgröße \\ 0 \\ 0 \end{pmatrix}$. Weitere Kacheln können solange in der ersten Reihe platziert werden, bis die maximale Anzahl an Kacheln pro Reihe erreicht ist. Danach geht es in der zweiten Reihe weiter $\begin{pmatrix} 0 \\ Kachelgröße \\ 0 \end{pmatrix}$. Nach diesem Muster können die Positionen aller Kacheln im Raum festgelegt werden und bilden so das Terrain.

Kamera

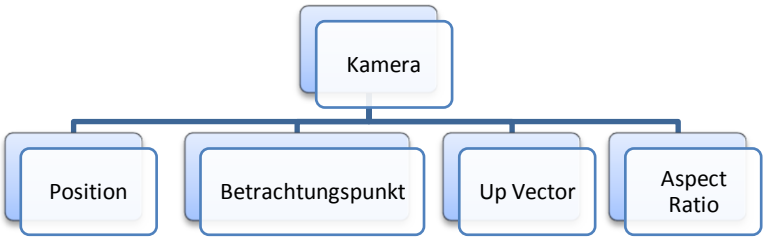


Abbildung 15: logische Struktur einer Kamera

Um das Terrain betrachten zu können, wird eine Kamera benötigt. Alles was für den Anwender zu sehen ist, sieht er aus der Perspektive der Kamera. Daher wird die Kamera so im Raum platziert, dass das Terrain möglichst optimal zu sehen ist. Zusätzlich gilt es Steuerungsmöglichkeiten zu schaffen, die es ermöglichen die Kamera so zu bewegen, dass das Terrain aus allen Perspektiven betrachtet werden kann. Dabei sollte die Steuerung so einfach wie möglich sein, damit sie für jeden Anwender schnell erlernbar ist. Das Kameraobjekt kann mittels Irrlicht erstellt werden und muss nur in einigen Attributen angepasst werden.

Abbildung 16 zeigt, wie die Kamera im Raum steht und wie die Szene betrachtet wird. Das Sichtfeld wird durch zwei Ebenen eingeschränkt, eine kameranahe Ebene und eine entfernte Ebene. Der Bereich zwischen den beiden Ebenen wird am Bildschirm dargestellt. In diesem Bereich sollte sich das Terrain komplett oder zumindest teilweise befinden. Die später vorgestellten Steuerungsmöglichkeiten der Kamera ermöglichen es, das Sichtfeld so zu verschieben, dass z.B. durch einen starken Zoom nur Teile des Terrains zu sehen sind.

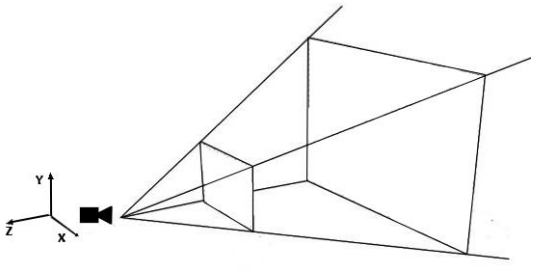


Abbildung 16: Ausrichtung der Kamera im Raum

Startposition

Die Kamera wird so platziert, dass das Terrain möglichst optimal auf dem Bildschirm zu sehen ist. Dazu werden als Informationen die Fenstergröße und die Größe des Terrains benötigt. Die Position der Kamera wird genau auf die Mitte des Terrains gesetzt und so weit nach hinten (z-Koordinate) verschoben, bis das Terrain komplett zu sehen ist. So ergeben sich x- und y-Koordinaten aus der Hälfte der Terrainbreite bzw. Höhe. Bei der y-Koordinate ist darauf zu achten, dass diese negativ sein muss, da das Terrain komplett im negativen y-Bereich platziert wurde. Als z-Koordinate kann das Maximum aus Terrainbreite und Terrainhöhe verwendet werden. Die folgende Formel verdeutlicht dies:

$$Terrain[breite|höhe] \;/\; Kameraentfernung = [vertikale|horizontale]\;\; Ausnutzung\, des\, Sichtbarkeitsbereichs\;(\%)$$

Der Quotient muss eins ergeben, um eine Sichtbarkeit von 100% zu erreichen. So ist sichergestellt, dass das Terrain vollständig zu sehen ist. Je nach Bedarf kann der Abstand mit einem Multiplikator noch etwas verringert oder erhöht werden. Dies ist erforderlich wenn die GUI einen Teil des Bildes verdeckt, z.B. bei einem Menü oder einer Statusleiste. Auch die z-Koordinate muss negativ sein.

$$Position = \begin{pmatrix} Terrainbreite/2 \\ -(Terrainhöhe/2) \\ -(\max(Terrainbreite, Terrainhöhe) * Korrektur) \end{pmatrix}$$

Der Betrachtungspunkt ist der Punkt im Raum, zu dem die Kamera ausgerichtet sein soll. Um eine orthogonale Sicht zu erhalten, dürfen x- und y-Koordinaten nicht von der Position der Kamera abweichen. Die z-Koordinate wird auf die z-Koordinate der Kacheln, also auf null, gesetzt.

$$Betrachtungspunkt = \begin{pmatrix} Terrainbreite/2 \\ -(Terrainhöhe/2) \\ 0 \end{pmatrix}$$

Der Up Vector der Kamera gibt ihr die Orientierungsrichtung in der Szene vor, also ihre Ausrichtung im Raum. Man kann sich den Vektor als Seil durch den Raum vorstellen, an dem die Kamera fest aufgehängt ist. Eine Veränderung der Blickrichtung ist möglich, aber abhängig von der Aufhängungsrichtung. Der Vektor wird für Drehungen der Perspektive verwendet. Er ist wie der Raum dreidimensional und auf die Länge eins normalisiert. Mit den drei Koordinaten x, y und z wird die Kamera in die entsprechende Richtung ausgerichtet. Um das Terrain zunächst ohne Drehung betrachten zu können, wird die Kamera nach der y-Achse ausgerichtet.

$$UpVector = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}$$

Das Aspect Ratio beschreibt das Verhältnis zwischen Breite und Höhe des dargestellten Bereichs. Dieses wird aus der Auflösung des Programmfensters berechnet. Wird der Wert nicht oder falsch gesetzt, führt dies zu einer gestreckten oder gestauchten Darstellung.

$$AspectRatio = \frac{Fensterbreite}{Fensterhöhe}$$

Steuerung

Eine Kamerasteuerung gibt dem Benutzer die Möglichkeit die Position und Ausrichtung der Kamera nach seinen Wünschen zu verändern. Bei einem Editor für eine 3-dimensionale Welt ist es wichtig, dass alles aus jedem Winkel betrachtet werden kann, die Übersicht aber nicht verloren geht. Die hier verwendete Steuerungstechnik wird in ähnlicher Form in vielen 3D Spielen eingesetzt. Unter anderem in RTS Spielen, aber auch in Aufbausimulationen wie Roller „Coaster Tycoon 3“ oder in MMORPGs („Guild Wars“, „Herr der Ringe“). Die folgenden Techniken und Formeln basieren allerdings komplett auf eigenen Ideen.

Kamera bewegen

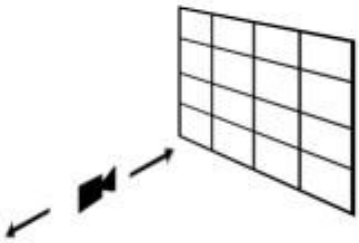


Abbildung 17: Bewegungsmöglichkeiten der Kamera

Eine Bewegung der Kamera soll in x und y Richtung möglich sein. So kann aus orthogonaler Perspektive jeder Bereich des Terrains betrachtet werden. Am besten werden die Pfeiltasten mit der Steuerung belegt.
Mit jedem Druck auf eine Pfeiltaste muss die Kamera also ein paar Pixel bewegt werden. Dazu wird die Anzahl der Pixel in einer Variablen definiert: $move\_speed = 3;$
Die Bewegungsgeschwindigkeit bestimmt, wie schnell die Kamera bewegt werden kann. Bei Drücken der Pfeiltaste nach oben muss die Positionsvariable verändert werden:

$$cam.position.Y \mathrel{+}= move\_speed;$$

Dies würde allerdings die Perspektive verzerren, da der Betrachtungspunkt sich nicht verändert hat. Daher muss auch dieser verändert werden: $cam.target.Y \mathrel{+}= move\_speed;$
Auch das reicht aber noch nicht für eine fehlerfreie Bewegung der Kamera, da die Kamera gedreht sein könnte. Bei einer Drehung z.B. um 180° wäre die Steuerung umgedreht (links|rechts, oben|unten). Für die nötige Korrektur sorgt der UpVector:
$cam.position.Y \mathrel{+}= move\_speed * cam.upVector.Y;$
Die x-Position der Kamera sowie die Koordinaten des Betrachtungspunkts werden auf die gleiche Weise berechnet.

Zoomen

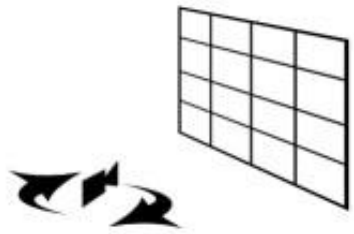


Abbildung 18: Bewegungsrichtung der Kamera bei einem Zoomvorgang

Ein Zoomvorgang ist eine Kamerabewegung in z-Richtung. Dabei ist darauf zu achten, dass nicht durch das Terrain hindurch gezoomt werden kann. Die Kamera muss also immer im negativen z-Bereich bleiben. Die Steuerung kann mit dem Scrollrad oder zwei alternativen Tasten der Tastatur erfolgen.

$$cam.position.Z \mathrel{+}= zoom\_speed;$$

Neigen

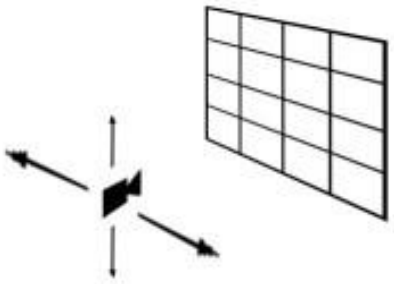


Abbildung 19: Neigungsmöglichkeiten der Kamera

Verändert sich die Position der Kamera gegenüber ihrem Betrachtungspunkt, so entsteht eine Neigung. Dies sollte maximal in einem 90° Winkel möglich sein, da das Terrain so vom orthogonalem Blickwinkel (90°) bis zur flachen Seitenansicht (0°) zu sehen ist. Eine gute Steuerung ist z.B. mit Mausbewegungen innerhalb der y-Achse möglich. Die Steuerung selbst könnte mit Festhalten beider Maustasten aktiviert werden. Für die Berechnung der Koordinaten sind Winkel (angle), Abstand zum Betrachtungspunkt (distance) und der UpVector wichtig. Alle

Winkel werden in Grad verwendet.

$$cam.position.X = cam.target.X - (\frac{distance}{90} * (90 - angle)) * cam.upVector.X;$$
$$cam.position.Y = cam.target.Y - (\frac{distance}{90} * (90 - angle)) * cam.upVector.Y;$$
$$cam.position.Z = -\frac{distance}{90} * angle;$$

Drehen

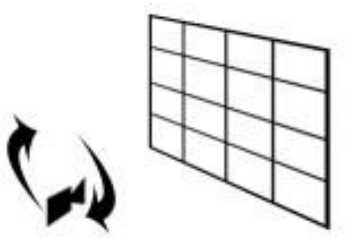
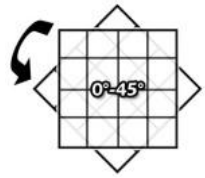
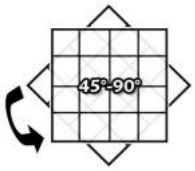


Abbildung 20:
Drehungsmöglichkeiten der Kamera

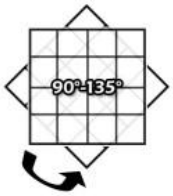
Durch Drehen der Kamera ist es möglich, die Sicht auf das Terrain von 0° bis 360° zu drehen. Bei einer Drehung von z.B. 90° nach links wäre die obere Seite des Terrains als linke Seite zu sehen. Die Steuerung kann wie beim Neigen umgesetzt werden, nur diesmal mit der x-Achse der Maus. Bei einer Drehung verändert sich die Ausrichtung der Kamera und damit der UpVector. Dieser muss je nach Drehwinkel angepasst werden und zwar nach dem folgendem Schema. Die Vektoren geben den jeweiligen UpVector aus einem der acht Drehbereiche an. Die Variable *angle* steht für den Drehungswinkel. Alle Winkel sind zur Veranschaulichung in Grad angegeben. Daher müssen die Vektoren vor der Verwendung erst normalisert werden.



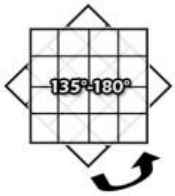
$$\begin{pmatrix} angle \\ 45 \\ 0 \end{pmatrix}$$



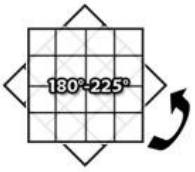
$$\begin{pmatrix} 45 \\ 90 - angle \\ 0 \end{pmatrix}$$



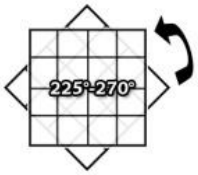
$$\begin{pmatrix} 45 \\ -angle\%45 \\ 0 \end{pmatrix}$$



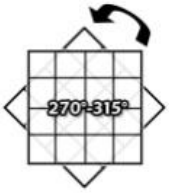
$$\begin{pmatrix} 180 - angle \\ -45 \\ 0 \end{pmatrix}$$



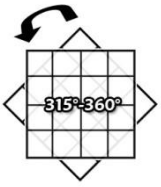
$$\begin{pmatrix} -angle\%45 \\ -45 \\ 0 \end{pmatrix}$$



$$\begin{pmatrix} -45 \\ angle - 270 \\ 0 \end{pmatrix}$$



$$\begin{pmatrix} -45 \\ angle\%45 \\ 0 \end{pmatrix}$$



$$\begin{pmatrix} angle - 360 \\ 45 \\ 0 \end{pmatrix}$$

Beispiel: Im Drehbereich von 0° bis 45° richtet sich die x-Koordinate des UpVectors nach dem Drehwinkel, während die y-Koordinate immer 45 (Grad) beträgt. Nach Normalisierung ist der Vektor bei keiner Drehung (0°) gleich $\begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}$ und bei einer Drehung von 45° gleich $\begin{pmatrix} 1 \\ 1 \\ 0 \end{pmatrix}$. Nach diesem Schema kann für jeden Drehwinkel von 0° bis 360° der zugehörige UpVector berechnet werden.

GUI

Die GUI ist eine grafische Oberfläche, die die Schnittstelle zwischen Anwender und Programm bildet. Mit dieser Schnittstelle können auf einfache Art und Weise Aktionen ausgewählt und durchgeführt werden. Zu Aktionen zählen alle Editierfunktionen (Kapitel 6) sowie zusätzliche Optionen, die z.B. über ein Menü auswählbar sind. Die wichtigsten GUI Elemente sind ein Menü, ein Aktionsfenster, ein Statusfenster und Fenster für die grafischen Elemente. Über das Menü können grundlegende Einstellungen gemacht werden, wie z.B. Laden/Speichern. Im Aktionsfenster befinden sich Buttons (Schaltflächen) mit denen sich die Editierfunktionen durchführen lassen. Im Statusfenster werden die Informationen zu den aktuellen Aktionen angezeigt. Zusätzlich können dort Details zu Kacheln und Objekten angezeigt werden. Grafische Elemente, wie z.B. Texturen oder 3D Objekte benötigen ebenfalls ein Fenster, über das sie ausgewählt werden können.

Der exakte Aufbau aller GUI Elemente kann nach eigenen Vorstellungen durchgeführt werden. Wichtig dabei ist nur, dass für die spätere Anwendung alle Elemente möglichst einfach zu verstehen und schnell zu erreichen sind.

Jedem GUI Element muss eine ID zugeordnet werden (bei Verwendung der Irrlicht Engine). Immer wenn ein GUI Ereignis ausgelöst wird, wird dieses zusammen mit der ID an die Eingabeverwaltung gesendet. Dort muss mittels der ID das zugehörige GUI Element bestimmt werden. So kann dem Ereignis eine Aktion zugeordnet werden, die anschließend vom Operator verarbeitet wird.

6. Editierfunktionen

In diesem Kapitel werden einige Funktionen zum Editieren des Terrains vorgestellt. Die Spanne reicht dabei von sehr gängigen Funktionen für einen Terraineditor (Verwendung von Texturen) bis hin zu Funktionen, die nicht unbedingt notwendig sind (Verwendung von Shadern). Darüber hinaus sind aber noch viele Weitere denkbar. Die hier vorgestellten Funktionen sollen nur einen Überblick über die Möglichkeiten verschaffen. Los geht es mit den Wichtigsten rund um Kacheln wie das Auswählen und Texturieren von ihnen. Diesen folgen Weitere rund um 3D Objekte. Anschließend werden erweiterte Techniken zur Manipulation von Kacheln demonstriert. Zuletzt wird das Terrain um Heightmaps erweitert.

6.1. Kachelauswahl

Bei Auswahl einer Kachel sollen im Statusfenster Informationen über sie angezeigt werden. So kann z.B. angezeigt werden, an welcher Position sie sich im Terrain befindet. Es kann auch angezeigt werden ob ein Objekt auf ihr platziert wurde und um welches es sich handelt. Nach *Abbildung 11* werden Objekte allerdings dem Terrain und nicht einer Kachel zugeordnet (mehr dazu in Abschnitt 6.3.). In den folgenden Funktionen werden Möglichkeiten geschaffen, um Kacheln zu verändern. Auch diese Veränderungen sollten zu sehen sein.

Zur Auswahl der Kachel muss der Benutzer den Mauszeiger „über“ die Kachel bewegen und sie mit einem Mausklick auswählen. „Über“ bedeutet, dass der Mauszeiger die Kachel, aus der aktuellen Perspektive heraus, verdecken soll wie in *Abbildung 21* dargestellt. Die Schwierigkeit liegt dabei darin, dass der Mauszeiger sich nur in einem 2D-Raum bewegt während sich die Kachel in einem 3D-Raum befindet. Um bestimmen zu können, was sich unter dem Mauszeiger befindet, wird ein unsichtbarer Strahl erstellt. Dieser Strahl bewegt sich vom Mauszeiger aus in Abhängigkeit der aktuellen Blickrichtung durch den Raum. In *Abbildung 22* ist der Strahl sichtbar. Auf der blauen Ebene (Bildschirm) befindet sich der Mauszeiger. Anschließend wird mit dem Strahl eine Kollisionserkennung durchgeführt, bei der bestimmt wird, ob eine Berührung mit einem 3D Objekt vorliegt. In *Abbildung 22* ist es die gesuchte Kachel. Die Algorithmen zur Kollisionserkennung stehen durch die Irrlicht Engine zur Verfügung und müssen daher nicht selbst entwickelt werden.

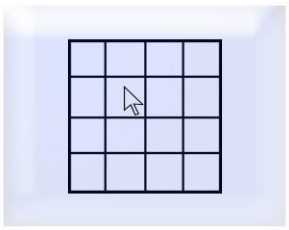


Abbildung 21: Strahl von der Maus aus in Blickrichtung

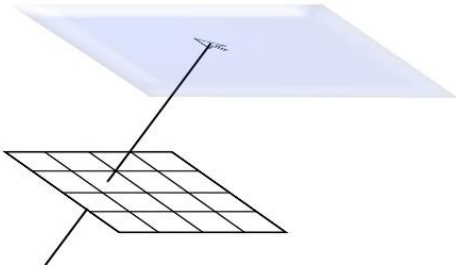


Abbildung 22: Strahl von der Maus aus in Blickrichtung

Die Technik zur Auswahl einer Kachel wird noch häufiger benötigt, da Aktionen oftmals durch die Auswahl eines 3D Objekts ausgeführt werden.

6.2. Kacheln texturieren

Das Texturieren von Kacheln ist die einfachste Möglichkeit, die Optik des Terrains zu verändern. Eine Textur ist nichts anderes als eine Grafik aus einem gängigen Dateiformat. Von Irrlicht werden folgende Formate unterstützt: psd, jpg, png, tga, bmp, pnx.

Um eine Textur einer Kachel hinzuzufügen muss sie zuerst im Auswahlfenster gewählt werden und anschließend die Kachel angeklickt werden. Diese wird wieder mit der im Abschnitt „Kachelauswahl“ beschriebenen Technik ausgewählt. Nun muss der Kachel nur noch die Textur zugeordnet werden.

Bei der Auswahl der Texturen ist auf das Kachelkonzept des Editors zu achten. Nicht jede Textur ist dazu geeignet auf eine Kachel gelegt zu werden. Bei diesen wird ein Übergang erkennbar, wenn sie mehrfach nebeneinander gelegt werden, was zu einer merkwürdigen Optik des Terrains führt. Ziel sollte es bei Verwendung von grafischen Elementen immer sein, ein stimmiges Gesamtbild zu erhalten. In *Abbildung 23* sind bei der ersten Textur deutliche Übergänge zu sehen, während sich bei der zweiten Textur alle Steine so aneinander fügen, dass kein Übergang zu sehen ist.

nicht geeignet **geeignet**

Abbildung 23: Unterschied zwischen Kachel geeigneten und nicht geeigneten Texturen

Texture Splatting

Im vorherigen Abschnitt wurde die Problematik bei mehrfacher Verwendung der gleichen Textur erläutert. Die Verwendung von sehr unterschiedlichen Texturen kann sich ebenfalls negativ auf den optischen Eindruck auswirken. Vor allem an den Übergängen unterschiedlicher Terraintypen wird es problematisch, wie z.B. beim Übergang von Land zu Wasser. Für diesen Zweck müsste eine Übergangstextur erstellt werden. Mit jedem hinzukommenden Terraintyp würde aber auch die Anzahl der benötigten Übergänge steigen. Diese Arbeit kann mit der Technik des „Texture Splatting“ (Granberg, 2007) automatisiert werden. Dabei werden Texturen so übereinander gelegt, dass kein Übergang mehr zu erkennen ist.

Abbildung 24: Produkt aus Textur und alpha map

Zur Durchführung des „Texture Splatting“ werden „alpha maps“ über die Texturen gelegt. Dies sind Grafiken die nur aus Graustufen bestehen und zur Erzeugung von Transparenz dienen. Dazu wird das Produkt aus Textur und „alpha map“ gebildet. Das Ergebnis ist eine Textur, bei der von der ursprünglichen Textur nur die Teile zu sehen sind, die bei der „alpha map“ komplett weiß sind. Je dunkler ein Pixel in der „alpha map“ ist, umso weniger ist von der Textur zu sehen. Bei einem schwarzen Pixel ist nichts mehr zu sehen. In *Abbildung 24* wird dies demonstriert. Anschließend werden alle gewünschten Texturen addiert, um die finale Textur zu erhalten. Dabei können, wie in *Abbildung 25* gezeigt, zwei oder beliebig viele Texturen miteinander addiert werden.

Abbildung 25: Addition von Texturen

Flächen texturieren

Sollen alle Kacheln eines größeren zusammenhängenden Gebiets mit der gleichen Textur belegt werden, kann dies zu einer mühsamen Arbeit werden. Daher ist eine weitere Funktion zum Kachel texturieren sinnvoll. Mit dieser kann ein rechteckiger Bereich über das Terrain gezogen werden und alle darunterliegenden Kacheln werden automatisch mit der ausgewählten Textur belegt. *Abbildung 26* verdeutlicht den Vorgang. Die Herausforderung dabei ist die Bestimmung der unter dem Rechteck liegenden Kacheln. Dazu werden im folgenden Abschnitt zwei selbst entwickelte Lösungsverfahren vorgestellt.

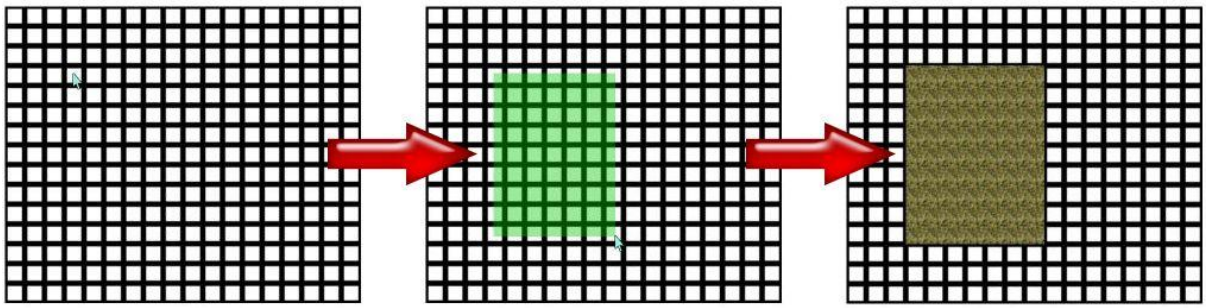


Abbildung 26: Ablauf beim texturieren einer Fläche

Bei beiden Verfahren sind die Berechnungen am einfachsten, wenn die Kamera fest ausgerichtet und nicht beweglich ist. Angenommen das Terrain ist absolut gerade zur Kamera ausgerichtet, also ohne Drehung und Neigung. Der Abstand der Kamera zum Terrain ist so gewählt, dass die Kacheln mit ihrer tatsächlichen Größe am Bildschirm zu sehen sind. Die Kamerasteuerung ist bis auf die Bewegungsmöglichkeit deaktiviert. Beide Verfahren benötigen die Mauskoordinaten der Start- und Endposition des Rechtecks sowie die darunterliegenden Kacheln. Beide Kacheln werden mit dem im Abschnitt „Kachelauswahl“ beschriebenen Verfahren bestimmt.

Verfahren 1 – Nachbarschaftsverfahren
Beim Nachbarschaftsverfahren wird der logische Aufbau des Terrains verwendet. Für die erste und letzte Kachel kann jeweils die Reihe und Spalte in der sie sich befindet berechnet werden. Anschließend können alle gesuchten Kacheln in zwei verschachtelten Schleifen durchlaufen werden.

```
for (int i=startkachel.reihe; i<=endkachel.reihe; ++i)
      for (int j=startkachel.spalte; j<=endkachel.spalte; ++j)
            kachel[i][j].textur = textur;
```

Verfahren 2 – optisches abtasten
Genau wie die erste und letzte Kachel können auch die weiteren Kacheln durch Aussenden eines Strahls und anschließender Kollisionserkennung bestimmt werden. Die Position des Strahls muss immer mit der Kachelgröße weiter verschoben werden.

```
for (int i=startpunkt.y; i<=endpunkt.y; ++kachelgröße)
      for (int j=startpunkt.x; j<= endpunkt.x; ++kachelgröße)
      {
            //Verfahren aus „Kachel auswählen"
            kachel = getCollision(j, i);
            kachel[i][j].textur = textur;
      }
```

Da die Bedingungen noch sehr konstruiert sind, wären die Verfahren im praktischen Einsatz kaum zu gebrauchen. Im folgenden Schritt wird die Kameraposition nicht mehr vorgegeben und es kann nach Belieben gezoomt werden. Für das Nachbarschaftsverfahren ändert sich dadurch nichts, beim optischen Abtasten sind aber weitere Schritte notwendig. Je nach Entfernung der Kamera zum Terrain sind die Kacheln in unterschiedlicher Größe am Bildschirm zu sehen. Sind die Kacheln kleiner zu sehen als ihre tatsächliche Größe ist, dann würden Kacheln übersprungen werden. Daher muss zuerst die Größe der Abtastschritte berechnet werden. Dazu wird vom Startpunkt aus die x-Koordinate in ein Pixel großen Schritten erhöht und jedesmal die darunterliegende Kachel bestimmt, bis die zweite Kachel erreicht ist. Der Vorgang wird weiter bis zur dritten Kachel durchgeführt. Die Differenz der Koordinaten der zweiten und dritten Kachel ergibt die Größe der Abtastschritte. In *Abbildung 27* zeigt der rote Punkt die Position des Mausklicks. Entlang der Linie in Kachel 2 wird die Messung durchgeführt.

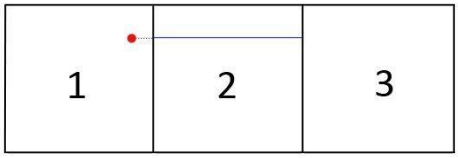


Abbildung 27: Abmessen der Kachelgröße

Die Codeausschnitte zeigen, dass das optische Abtasten wesentlich rechenintensiver ist als das Nachbarschaftsverfahren und daher eher nicht zu empfehlen. Dennoch wird es vorteilhaft, wenn die Einschränkungen der Kamera aufgehoben werden, also wenn auch Drehen und Neigen wieder möglich ist. In diesem Fall ist das Nachbarschaftsverfahren nicht mehr einsetzbar. Mit dem optischen Verfahren können weiterhin alle sichtbaren Kacheln bestimmt werden. Die Abtastschritte müssen vorgegeben werden und müssten zur optimalen Durchführung einen Pixel betragen. Alternativ wäre

noch eine etwas ungenauere aber wesentlich schnellere Durchführung mit größeren Abtastschritten möglich.

6.3. 3D Objekt hinzufügen

Objekte ermöglichen eine individuelle Gestaltungsmöglichkeit des bisher eher statischen Terrains. Mit Bäumen, Steinen oder beliebigen anderen Elementen ist es möglich, die Welt etwas lebendiger werden zu lassen.

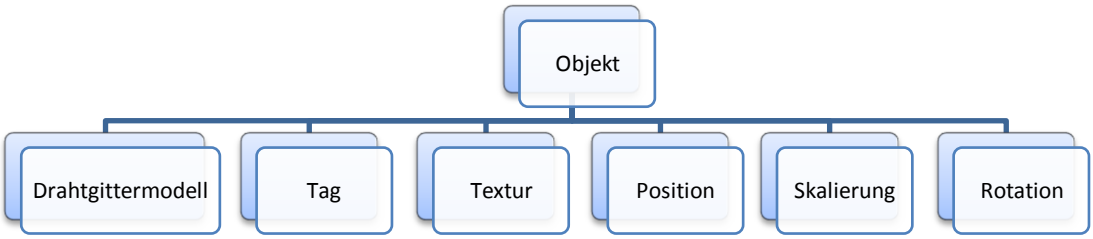


Abbildung 28: logische Struktur eines Objekts

Ein Objekt besteht aus einem Drahtgittermodell (*Abbildung 30*), das mit einer Textur belegt werden kann (*Abbildung 29*). Zur Speicherung und Anzeige von Objekten bietet Irrlicht die notwendigen Klassen. Diese müssen nur mit zusätzlichen Informationen erweitert werden, wie z.B. einem Tag um jedem Objekt Eigenschaften zuordnen zu können. Drahtgittermodelle werden aus Dateien geladen. Alle gängigen Formate werden von Irrlicht unterstützt, z.B. 3ds, b3d, obj, x, md2.

Abbildung 30: Drahtgittermodell einer Kugel

Abbildung 29: Kugel mit Textur

Objekte können frei im Raum platziert werden. Für eine Nutzung der erstellten Karte außerhalb des Editors ist es von Vorteil Objekte der Kachel zuzuordnen, auf der sie sich befinden. Auch im Editor können bei der Auswahl einer Kachel die Objekte auf ihr angezeigt werden. Um ein Objekt frei platzieren zu können, muss eine 3-dimensionale Koordinate gespeichert werden, anhand des Koordinatensystems aus *Abbildung 9*. Die Skalierung ist standardmäßig eins und die Rotation null. So werden Größe und Ausrichtung des Objekts aus der Datei des Drahtgittermodells entnommen.

Objekt auswählen

Um ein Objekt auszuwählen, kann die gleiche Technik wie bei „Kachelauswahl“ verwendet werden. Dazu muss nur eine Typunterscheidung zwischen Kacheln und Objekten erfolgen. Bei der Auswahl einer Kachel ist es vorteilhaft, die Objekte vorher zu deaktivieren. Bei der Kollisionserkennung wird nur die erste Kollision aus Sicht der Kamera bestimmt. Da sich Objekte aus dieser Sicht immer vor Kacheln befinden, kann es zu Überschneidungen kommen, bei denen die Kachel nicht mehr erkannt werden kann. Durch eine Deaktivierung werden die Objekte bei der Kollisionserkennung nicht mehr berücksichtigt.

Nach Auswahl eines Objekts können Informationen über dieses im Statusfenster angezeigt werden. Geeignet dafür sind z.B. die Position, das Tag oder die Objektbezeichnung.

Objekt texturieren

Genau wie Kacheln können auch Objekte mit einer Textur belegt werden, um ihr Aussehen zu verändern. Die Auswahl der Texturen ist hierbei nicht eingeschränkt wie bei den Texturen der Kacheln, da jedes Objekt für sich eigenständig ist und kein Übergang zu benachbarten Objekten geschaffen werden muss.

Um eine Textur korrekt auf ein Objekt zu legen, muss dies in den Eigenschaften des Objekts festgelegt sein. Dazu müssen vor Verwendung im Editor dem Objekt UV-Koordinaten zugeordnet werden. UV-Koordinaten beschreiben Texturkoordinaten (2D), die dazu genutzt werden jedem Punkt des Objekts (3D) eine Koordinate der Textur zuzuweisen. *Abbildung 31* zeigt zwei Würfel mit der gleichen Textur. Die UV-Koordinaten des linken Würfels sind falsch und daher ist die Struktur der Textur nicht zu erkennen. Viele 3D Grafik Programme (z.B. Blender) bieten Funktionen zur automatischen Generierung der Koordinaten, die meist nur im Detail verbessert werden müssen.

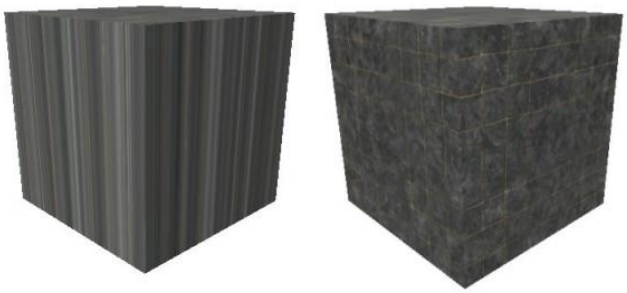


Abbildung 31: Würfel mit falschen UV-Koordinaten (links) und richtigen Koordinaten (rechts)

Objekt editieren

Nachdem ein Objekt ausgewählt wurde, werden Editiermöglichkeiten bereit gestellt, durch die das Objekt angepasst werden kann. Dazu sind Editierfelder nötig, in denen Eigenschaften wie Position, Skalierung und Rotation eingegeben werden können. *Abbildung 31* zeigt beispielhaft wie der Editierbereich für Objekte aussehen kann.

Abbildung 32: Darstellung der Editiermöglichkeiten für Objekte

Beleuchtung durch ein Shaderprogramm

Ein Shaderprogramm ist eine Reihe an Software Instruktionen, die von dem Grafikprozessor (GPU) verarbeitet werden. Dies ist vor allem bei der Berechnung von Grafikeffekten vorteilhaft, da diese von der GPU schneller als von der CPU berechnet werden können. Zusätzlich wird die CPU dadurch entlastet. Im Folgenden wird eine Möglichkeit gezeigt, wie Texturen von Objekten beleuchtet werden können.

Ein Shader kann mittels Irrlicht Funktionen jedem 3D Objekt zugeordnet werden. Die bei der Shader Programmierung verwendbaren Funktionen sind abhängig von der verwendeten Grafik API. Die OpenGL und Direct3D Bibliotheken verwenden drei Typen von Shadern:

- Pixel-Shader (wird in OpenGL als Fragment-Shader bezeichnet) können einzelne Pixel manipulieren. Damit kann der Farbton von Pixeln verändert werden. Pixel-Shader werden häufig für Beleuchtungseffekte und Bump-Mapping (Simulation von Unebenheiten auf Oberflächen) verwendet.
- Vertex-Shader können bestehende Geometrie verändern. Damit lassen sich Effekte wie z.B. Wasserbewegungen erzeugen. Die berechneten Vertices werden meist an den Geometry-Shader weitergereicht.
- Geometry-Shader können Drahtgittermodelle verändern, indem sie Vertices entfernen oder hinzufügen. Im Gegensatz zu Vertex-Shadern gibt es keine Beschränkung auf bestehende Geometrie.

Das Unified-Shader-Model vereint die Funktionen aller drei Shader. Dadurch verschwindet die hardwareseitige Unterscheidung der Shadertypen und jede Shader-Einheit des Grafikchips kann die Berechnungen aller drei Typen durchführen. Der Grafiktreiber entscheidet dabei, welcher Einheit ein Shader-Programm zugeordnet wird, wodurch eine Leistungssteigerung erzielt werden soll. Nur wenige Grafikkarten unterstützen bisher das Unified-Shader-Model.

Zur Beleuchtung von Texturen werden Fragment und Vertex-Shader benötigt.

Für die Programmierung muss die von der API vorgegebene Sprache verwendet werden. Bei OpenGL ist das die „OpenGL Shading Language“ (GLSL) und bei Direct3D wird die „High Level Shader Language“ (HLSL) verwendet. Im folgenden Beispiel wird GLSL verwendet. Diese basiert auf der Programmiersprache C. Es wird je ein Vertex und ein Fragment Programm benötigt. Die Programme können während der Laufzeit des Editors geladen werden. Dadurch ist es leicht möglich, ein bestehendes Shader-Programm zu verändern oder einen neuen Shader zu erstellen.

Ziel des folgenden Shader Programms (GLS) ist die Beleuchtung der Texturen von Objekten. Dazu wird eine Lichtquelle im Raum benötigt, die die Beleuchtungsstärke bestimmt. Der Shader lässt die Seiten des Objekts, die zur Lichtquelle ausgerichtet sind, heller erscheinen als die vom Licht abgewandten Seiten. Dazu werden die Pixel der Textur entsprechend aufgehellt. In den *Abbildungen 33* und *34* wird der Unterschied zwischen beleuchteten und nicht beleuchteten Objekten verdeutlicht.

Vertex Shader

```
/* alle Variablen die mit gl_ beginnen sind interne GLSL Variablen
   mit varying kann von jedem Shader aus auf die Variablen zugegriffen
werden
*/
varying vec3 licht;     // Vektor der Lichtrichtung
varying vec3 normale;   // Normalenvektor der Oberfläche

void main()
{
      //transformieren und normaliseren der Normalen in den
Betrachtungsraum
      normale = normalize(gl_NormalMatrix * gl_Normal);

      //normalisieren der Richtung des Lichts
      licht = normalize(vec3(gl_LightSource[0].position));

      //Koordinaten der Textur in gl_TexCoord speichern
      gl_TexCoord[0] = gl_MultiTexCoord0;

      //Standartfunktion zur Vertextransformation
      gl_Position = ftransform();
}
```

Im Vertex Shader passiert nicht allzu viel. Die Lichtquelle wird analysiert und die Richtung, in der das Licht ausstrahlt, gespeichert. Außerdem wird noch die Textur des Objekts eingelesen. Damit stehen alle Informationen zur Verfügung,die im Fragment Shader benötigt werden. Die verwendeten Funktionen (normalize, ftransform) sind Bestandteil der GLSL.

Im Fragment Shader werden die neuen Farbwerte der Textur berechnet. Die Berechnung wird für jeden Pixel durchgeführt. Dazu werden Farbwerte und Transparenzwerte aus der Textur eingelesen und mit den Werten eines Fragments multipliziert. Ein Fragment ist die Ansammlung aller Informationen, die zur Darstellung eines Pixels notwendig sind. Berechnet wird das Fragment mittels den Beleuchtungswerten und den Materialwerten. Das Fragment stellt also die Veränderung eines Pixels dar.

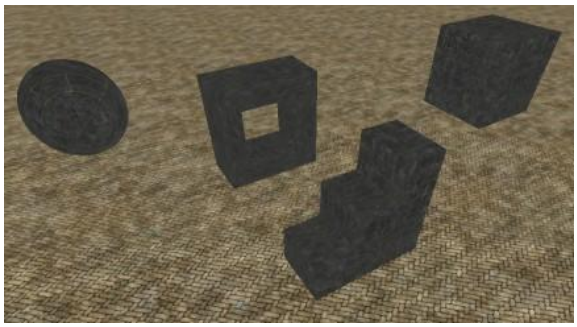


Abbildung 33: 3D Objekte ohne Beleuchtung

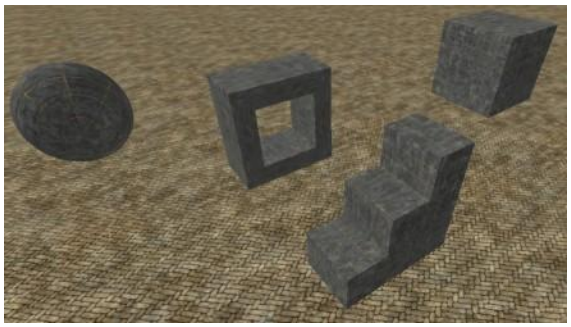


Abbildung 34: 3D Objekte mit Beleuchtungseffekt

Fragment Shader

```
/* uniform Variablen werden vom Programm an den Shader übergeben
   im Typ sampler2D können 2D Texturen gespeichert werden */
uniform sampler2D textur;
//varying Variablen aus dem Vertex Shader
varying vec3 licht, normale;

void main()
{
      //Variablen für Textur und Fragment Werte
      vec3 frag_farbe, textur_farbe;      //Vektoren für RGB Farben
      float frag_alpha, textur_alpha;     //Transparenzwerte

      //Berechnung der Helligkeit des Lichts
      float helligkeit = max(dot(licht,normalize(normal)),0.0);

      //Berechnung der Fragment Farbinformationen
      frag_farbe = gl_FrontMaterial.ambient.rgb + helligkeit *
            gl_FrontMaterial.diffuse.rgb;

      frag_alpha = gl_FrontMaterial.diffuse.a;

      //Auslesen der Textur Farbinformationen eines Pixels
      vec4 texel = texture2D(textur,gl_TexCoord[0].st);
      textur_farbe = texel.rgb;
      textur_alpha = texel.a;

      //Berechnung der neuen Farbwerte
      gl_FragColor = vec4(frag_farbe * textur_farbe, frag_alpha *
            textur_alpha);
}
```

Damit sind beide Programme fertig und können als Materialtyp an die Objekte übergeben werden. Zu beachten ist, dass das gezeigte Programm nur mit OpenGL funktioniert. Bei der Verwendung eines anderen Grafiktreibers muss der Shader entweder deaktiviert werden oder durch einen geeigneten Shader für den jeweiligen Grafiktreiber ersetzt werden.

6.4. Kacheln anheben

Mit den bisherigen Möglichkeiten kann dem Terrain optisch eine Struktur verpasst werden. Dies ist aber aus 3D Sicht platt. Aus 2,5D Sicht ist die Dreidimensionalität von 3D-Elementen nur eingeschränkt zu erkennen. Dennoch kann es für eine Einheit, die sich durch die Welt bewegen soll, ein Unterschied sein, ob diese flach ist oder Unebenheiten besitzt. Eine Steigung kann aufwendiger zu passieren sein als ein flacher Weg. Die folgenden Funktionen sollen es mit einfachen Mitteln ermöglichen, Höhenveränderungen durchzuführen. Alle Methoden stammen aus eigenen Ideen.

Als erstes wird eine einzelne Kachel um eine feste Anzahl an Pixeln angehoben. Anheben bedeutet aufgrund des festgelegten Koordinatensystems (*Abbildung 9*) eine Verringerung der z-Koordinate. Wird nun aber eine Kachel einfach in ihrer z-Koordinate verschoben dann würde sie in der Luft schweben wie es in *Abbildung 36* zu sehen ist. Es fehlen „Wände“ um die entstandene Lücke zu schließen (*Abbildung 35*).

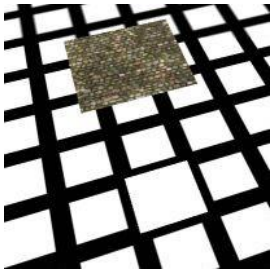


Abbildung 36: angehobene Kachel ohne Wände

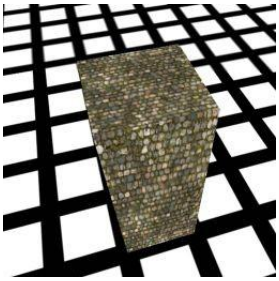


Abbildung 35: angehobene Kachel mit Wänden

Die Wände werden mit der gleichen Technik wie die Kacheln erstellt. Die Breite entspricht dabei der Kachelgröße und die Höhe ergibt sich aus der z-Koordinate der Kachel. Für jede der vier Seiten muss eine eigene Wand erstellt werden und auf diese Seite ausgerichtet werden. Bei jeder Veränderung der Kachelhöhe müssen die alten Wände entfernt und durch Wände mit passender Höhe ersetzt werden. Um Kachel und Wände in einer einheitlichen Optik zu bekommen, werden die Wände immer mit der gleichen Textur wir ihre zugehörige Kachel belegt.

Die Höhenveränderung kann gestuft oder frei erfolgen. Die gestufte Variante kann z.B. mit einem Mausklick aktiviert werden. Die Kachel könnte mit der linken Maustaste angehoben und mit der rechten Maustaste wieder abgesenkt werden. Bei der freien Steuervariante wird bei gedrückter Maustaste die Veränderung der y-Koordinate der Maus erfasst. So kann eine Auf- oder Abwärtsbewegung der Maus auf die Höhe der Kachel übertragen werden.

Kacheln neigen

Nachdem nun Kacheln angehoben werden können, ist es sinnvoll auch einen Weg zu diesen zu schaffen. Dies wird erreicht, indem eine anliegende Kachel so geneigt wird, dass sie eine Rampe zur höher liegenden Kachel bildet. Dabei muss die Kachel so geneigt werden, dass eine Seite auf der ursprünglichen Höhe bleibt und die gegenüberliegende Seite angehoben ist. Wie schon im Anschnitt „Kacheln anheben“ entstehen auch hier wieder Lücken. Diesmal eine rechteckige Lücke und zwei in der Form eines Dreiecks. Die Dreiecke können aufgebaut werden wie es in *Abbildung 14* dargestellt wird. Die Steuerung erfolgt wieder durch ein hoch und runter Bewegen der Maus. So wird die Höhe berechnet, die der höchste Punkt der Kachel bekommen soll. Mit links und rechts Bewegungen kann zwischen den vier Neigungsrichtungen durchgeschaltet werden.

Es müssen zwei Veränderungen an der Kachel vorgenommen werden. Zum einen muss die Kachel mit dem Neigungswinkel gedreht und zusätzlich noch skaliert werden, um ihre Größe anzupassen, da sonst eine Lücke zur nächsten Kachel entstehen würde. Die Länge der Kachel errechnet sich mit dem Satz des Pythagoras:

$$l = \sqrt{Kachelgröße^2 + Höhe^2}$$

In *Abbildung 37* sind die einzelnen Seitenbezeichnungen dargestellt. Zusätzlich ist noch der Neigungswinkel α eingezeichnet.

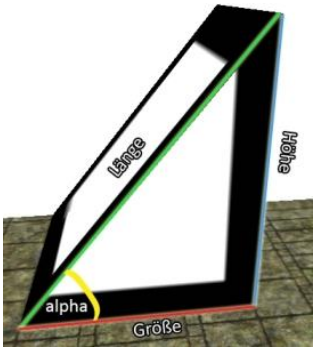


Abbildung 37: Definition der Seiten und Winkel einer Kachel

Der Neigungswinkel kann über die Winkelfunktionen berechnet werden, wie z.B. $\tan\alpha = \frac{v}{u}$. Bei Winkelberechnungen ist zu beachten, dass die Funktionen der math.h die Berechnungen im Bogenmaß durchführen. Der Neigungswinkel in Grad:

$$\alpha = \tan^{-1}\left(\frac{Kachelgröße}{Höhe}\right) * \frac{180}{\pi}\ , \qquad \pi = \tan^{-1}(1) * 4$$

Automatisches Neigen

Beim Anheben von Kacheln kann es praktisch sein, wenn die anliegenden Kacheln automatisch eine Rampe zu ihr bilden. So ist immer sichergestellt, dass es einen Weg zur angehoben Kachel gibt. Beim manuellen Neigen ist dazu viel Feinarbeit notwendig. Es gilt also eine Funktion zu erstellen, die automatisch alle vier Nachbarn so neigt, dass sie einen nahtlosen Weg zur angehobenen Kachel erzeugen. Der folgende Pseudo-Code demonstriert eine einfache Möglichkeit zur Umsetzung:

```
hebe_an(Kachel);        //hebt die ausgewählte Kachel an
überprüfeNachbarn(Kachel);

überprüfeNachbarn(Typ[Kachel])
{
      while(nicht alle Nachbarn wurden überprüft)
      {
            //wählt alle Nachbarn nacheinander aus
            Nachbar = nächsterNachbar(Kachel);

            //Himmelsrichtung ohne Drehung des Terrains
            richtung = berechneRichtung(Kachel, Nachbar);

            if (richtung == Steigungsrichtung von Nachbar
                  && Nachbar hat Nachbar in richtung)
            {
                  nächster_Nachbar = Nachbar von Nachbar in richtung;
                  if (nächster_Nachbar.Höhe == Kachel.Höhe)
                  {
                        hebe_an(Nachbar);
                        entferneSteigung(Nachbar);
                        überprüfeNachbarn(Nachbar);
                  }
            }
            else if (Nachbar hat keine Steigung
                        && Nachbar.Höhe < Kachel.Höhe)
            {
                  //der Nachbar bildet einen Weg zur Kachel
                  neige(Nachbar);
            }
      }
}
```

In diesem Programm geschieht folgendes. Nachdem die ausgewählte Kachel angehoben wurde, werden alle Nachbarn dieser Kachel überprüft. Es sind bei im Terrain liegenden Kacheln vier Nachbarn, bei Randkacheln drei Nachbarn und bei Eckkacheln zwei Nachbarn. Als erstes wird für jeden Nachbar überprüft, ob er bereits eine Steigung besitzt und wie diese ausgerichtet ist. Befindet

sich z.B. links von der betrachteten Kachel eine Kachel mit einer Steigung zu ihrem linken Nachbarn, dann wird die Steigung entfernt und die Kachel auf die Höhe der beiden Nachbarn angehoben. Voraussetzung hierfür ist, dass beide Nachbarn die gleiche Höhe besitzen. Für diese Kachel müssen dann ebenfalls alle Nachbarn überprüft werden, da gegebenenfalls die obere und untere Kachel eine Rampe zu dieser bilden könnten. Besitzt eine Kachel keine Steigung und ist sie niedriger als die ausgewählte Kachel, so kann sie in Richtung Kachel geneigt werden. Die dort ausgeführten Berechnungen entsprechen denen aus dem Abschnitt „Kacheln neigen“.

Mit diesem Programm ist der Aufbau von mehreren Ebenen, wie in *Abbildung 38* demonstriert, in kurzer Zeit möglich.

Der hier beschriebene Sonderfall, bei dem eine Nachbarkachel eine bestimmte Steigungsrichtung hat und die danebenliegende Kachel die gleiche Höhe besitzt, ist nur eine von vielen Möglichkeiten benachbarte Kacheln automatisch zu verändern. Auch Nachbarn mit Steigungen in anderen Richtungen könnten verändert werden.

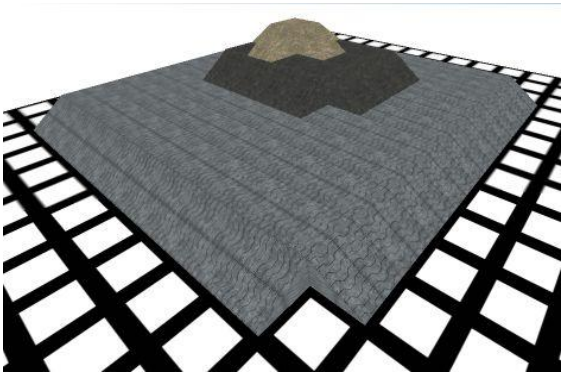


Abbildung 38: Kacheln auf drei Ebenen angehoben

6.5. Heightmaps verwenden

Heightmaps (Höhenfelder) sind zweidimensionale skalare Felder, mit denen dem Terrain eine individuelle Landschaftsstruktur verpasst werden kann. So können Gebirge und unebene Wege geschaffen werden. Jeder Wert der Heightmap steht für eine Höhe. Alternativ können die Werte auch für andere Zwecke interpretiert werden, z.B. für „Bump Mapping“ oder „Displacement Mapping“ (gibt der Oberfläche eines 3D Objekts eine höhere Detailtreue (Donnelly)).

Die Werte einer Heightmap werden üblicherweise aus Bilddateien mit Graustufen geladen. Jeder Farbwert steht für eine Höhe, wobei schwarz für eine minimale Höhe und weiß für eine maximale Höhe steht. Die Werte für minimale und maximale Höhe werden nicht im Bild gespeichert, sondern können beliebig festgelegt werden. Nachdem die Farbwerte eingelesen und in Höhenwerte umgerechnet worden sind, wird ein Drahtgittermodell anhand der Werte erzeugt. Dieses muss nur noch mit einer Textur belegt werden, um die Optik zu vervollständigen. Sämtliche Vorgänge vom Einlesen der Bilddatei bis zur Darstellung des Drahtgittermodells werden von der Irrlicht Engine erledigt.

Graustufenbilder erzeugen

Jedes bestehende Graustufenbild kann zur Erstellung von Heightmaps verwendet werden. Alternativ können die Bilder auch mit vielen Bildbearbeitungsprogrammen selbst erstellt werden. Eine weitere

Möglichkeit ist die zufallsbasierte automatische Generierung von Heightmaps. Die Ergebnisse sind so zwar nicht vorhersehbar, dafür aber variationsreicher.

Eine beliebte Methode zur Berechnung von pseudo-zufälligen Werten ist das „Perlin Noise“-Verfahren *(Perlin)*. Die Ergebnisse sind nicht komplett zufällig sondern werden aufeinander angepasst. Dies wird durch die Berechnung verschiedener Frequenzen erreicht, die übereinander gelegt werden. Um eine Frequenz erstellen zu können werden Zufallswerte zwischen -1.0 und 1.0 benötigt. Zur Berechnung der Werte kann ein beliebiger Zufallsgenerator eingesetzt werden. Wichtig ist nur, dass jede Berechnung mit einem Startwert initialisiert wird und gleiche Startwerte auch zu gleichen Ergebnissen führen. Die Ergebnisse werden interpoliert, um eine kontinuierliche Funktion zu erhalten.

Durch Interpolation können Werte zwischen zwei Messwerten angenähert werden. Es gibt verschiedene Verfahren zur Interpolation (lineare, cosinus, kubische und weitere). Dabei gilt für gewöhnlich, je rechenintensiver das Verfahren ist, umso runder wird die berechnete Funktion. In *Abbildung 39* lässt sich bei der grafischen Darstellung der Funktion schon eine Art Hügellandschaft erkennen.

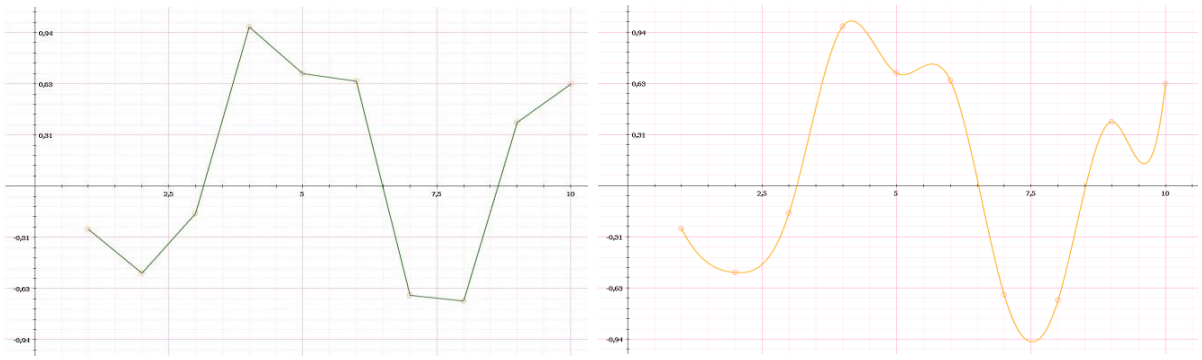


Abbildung 39: interpolierte Funktionskurven; lineare Interpolation (links), kubische Interpolation (rechts)

Ein noch besseres Ergebnis wird erzielt, wenn mehrere Funktionen erzeugt werden und aus diesen ein Mittel errechnet wird. Bislang ist das Ergebnis nur eindimensional, kann aber leicht um eine zusätzliche Dimension erweitert werden. Der folgende Codeausschnitt zeigt eine zweidimensionale Variante.

```
//berechnet für jede Koordinate einen Höhenwert
PerlinNoise(float x, float y)
{
	float total = 0;  //Höhe

	//jede Oktave steht für eine Funktionskurve
	for (int i=0; i<octaves; ++i)
	{
		float frequency = pow(2,i);   //Frequenz=1/Wellenlänge
```

```
            //maximaler Ausschlag der Funktion
            float amplitude = pow(persistence,i);

            total += InterpolatedNoise(x*frequency, y*frequency) *
                        amplitude;
      }
      return total;
}

//berechnet 4 benachbarte Punkte und interpoliert diese
InterpolatedNoise(float x, float y)
{
      float frac_x = x - int(x);    //Teilstück von x
      float frac_y = y - int(y);    //Teilstück von y

      //die noise Funktion ist der Zufallsgenerator, der anhand der
      //Koordinaten und einem (seed) die 4 Funktionswerte berechnet
      float v1 = noise(int(x) + int(y) + seed);
      float v2 = noise(int(x+1) + int(y) + seed);
      float v3 = noise(int(x) + int(y+1) + seed);
      float v4 = noise(int(x+1) + int(y+1) + seed);

      //Interpolation in x-Richtung
      float i1 = interpolate(v1 , v2 , frac_x);
      float i2 = interpolate(v3 , v4 , frac_x);

      //Interpolation in y-Richtung
      return interpolate(i1 , i2 , frac_y);
}
```

Die Funktion „PerlinNoise“ muss für jede Koordinate der Heightmap aufgerufen werden. Die Anzahl der Oktaven definiert die Anzahl der Kurven die überlagert werden sollen. Die Persistenz ist ein frei wählbarer Wert, der die Amplitude beeinflusst. Mit unterschiedlichen Amplituden lassen sich sehr unterschiedliche Funktionen erzielen. Im Gegensatz zum eindimensionalen Fall, bei dem zwei Nachbarwarte interpoliert werden, wird im zweidimensionalen Fall das Ergebnis aus vier benachbarten Feldern berechnet. Dazu werden erst die jeweils in x-Richtung benachbarten Felder interpoliert und anschließend die beiden Ergebnisse in y-Richtung. Das Ergebnis wird noch mit der Amplitude multipliziert und zum Endergebnis mit den Werten aller weiteren Funktionsgraphen addiert.

Heightmaps integrieren

Die Grundlagen sind geschaffen, um Heightmaps zu verwenden. Nach Laden der Bilder und Erzeugen der Drahtgittermodelle, müssen diese noch auf dem Terrain platziert werden. Dazu gibt es mehrere Lösungsmöglichkeiten. Aufgrund des verwendeten Kachelkonzepts sollte jede Heightmap die Größe einer Kachel haben. So können sie einfach auf die Kachel gelegt werden. Die Heightmap bekommt also die gleichen Koordinaten wie die Kachel zugeordnet. Je nach Ausrichtung des Terrains im Raum ist eventuell noch eine Drehung notwendig. Eine alternative Möglichkeit ist, die Heightmaps ähnlich wie Objekte zu behandeln. Sie werden also nicht fest einer Kachel zugeordnet, sondern können frei

im Raum platziert werden. Die Schwierigkeit dabei ist, eine Auswahltechnik zu entwickeln, mit der die Heightmap vom Terrain gewählt und bei Bedarf entfernt werden kann. Sind die Heightmaps hingegen den Kacheln zugeordnet, so kann ihre Auswahl über die Auswahl der Kacheln erfolgen.

Heightmaps editieren

Nach dem Platzieren einer Heightmap auf dem Terrain entspricht das Ergebnis nicht immer den Erwartungen, vor allem bei zufallsgenerierten Heightmaps. Es ist von Vorteil, wenn der letzte Feinschliff direkt im Editor getätigt werden kann, da so alle Veränderungen sofort in dreidimensionaler Form zu sehen sind. Zur Durchführung sind drei Schritte notwendig. Zuerst muss der angeklickte Pixel auf der Heightmap berechnet werden, dann muss das Graustufenbild manipuliert werden und zuletzt muss das neue Drahtgittermodell erzeugt werden. Die folgenden Erklärungen gehen davon aus, dass jede Heightmap an eine Kachel gebunden ist. Bei frei platzierbaren Heightmaps müssen die Techniken angepasst werden.

Für den ersten Schritt werden wieder die Techniken aus dem Abschnitt „Flächen texturieren“ verwendet. Die Kamera wird bis auf die Zoom- und Bewegungsmöglichkeiten blockiert. Anschließend wird die Größe einer Kachel gemessen, in der sie auf dem Bildschirm zu sehen ist. Zusätzlich werden die Abstände des angeklickten Punktes zur nächsten Kachel gemessen. Anhand dieser Daten kann der gewählte Punkt auf der Heightmap berechnet werden: $x = Randabstand(x) * \frac{Kachelgröße}{sichtbare\_Größe}$. Für die y-Koordinate geschieht die entsprechend mit dem Randabstand in y-Richtung.

Im zweiten Schritt soll das Graustufenbild verändert werden, also die Farbwerte einzelner Pixel. Dazu werden drei Angaben benötigt und zwar die Richtung, der Radius und die Stärke. Die Richtung steht für Anheben oder Absenken der Heightmap. Der Radius gibt die Fläche vor, die in einem Schritt editiert werden soll. Im kleinsten Fall wäre dies ein Pixel. Die Form des Editierwerkzeugs kann beliebig sein und ist im einfachsten Fall ein Rechteck. Die Stärke beschreibt, mit welcher Höhenveränderung sich jeder Mausklick auf die Heightmap auswirken soll, also die Intensität mit der sich der Farbwert des Bildes verändert. Beim Anheben ist sie positiv und beim Absenken negativ. Da ein RGB (Kombinationen der Farben Rot-Grün-Blau) Farbwert editiert wird, sind Stärkewerte von 1-255 sinnvoll. Die folgenden Schritte müssen für jeden Pixel innerhalb des Radius durchgeführt werden. Aus dem Graustufenbild der Heightmap wird an der Position des gewählten Pixels die Farbe ausgelesen. Zu jedem der drei Farbwerte Rot, Grün und Blau wird der Stärkewert addiert. Nur so bleibt ein Grauton erhalten. Die Obergrenze von 255 darf nicht überschritten werden. Das gleiche gilt beim Absenken für die Null.

Nachdem nun ein neues Graustufenbild vorliegt, wird im letzten Schritt das alte Drahtgittermodell vom Terrain entfernt, das neue Modell erzeugt und an gleicher Position platziert. Die verwendete Textur muss ebenfalls über das Modell gelegt werden.

Das System kann noch so erweitert werden, dass auch Kacheln editiert werden können, die keine Heightmap besitzen. Dazu wird einfach zu Beginn des Editiervorgangs eine höhenlose Heightmap erzeugt und auf die Kachel gelegt. Dies wird durch ein komplett weißes Bild (RGB-Werte: 255,255,255) erreicht.

7. Laden/Speichern

Die wohl wichtigste Funktion des Editors ist die Speichermöglichkeit. Ohne eine Speicherfunktion wäre die erstelle Karte nutzlos. Die Karte wird in XML-Dateien geschrieben, da diese für Mensch und Computer leicht zu lesen sind. So können auch ohne Verwendung des Editors kleine Veränderungen vorgenommen werden. Das Speicherkonzept wurde von mir selbst entwickelt. Für jede Karte werden drei XML-Dateien angelegt. Die terrain.xml enthält alle Informationen der Kacheln, die objects.xml die Objektinformationen und die options.xml enthält allgemeine Terraininformationen. Im Folgenden wird der Aufbau der Dateien veranschaulicht:

options.xml

```
<Options>
  <terrainx width="500" />     <!-- Breite des Terrains in Pixeln -->
  <terrainy height="300" />    <!-- Höhe des Terrains in Pixeln -->
  <node size="32" />           <!-- Kachelgröße in Pixeln -->
</Options>
```

Für den Aufbau eines leeren Terrains genügen die Werte für Breite und Höhe, sowie die Ausmaße einer Kachel. Weitere Informationen wie die Anzahl der Kacheln pro Zeile und Spalte können daraus berechnet werden.

objects.xml

```
<Objects>
  <Object name="Würfel">          <!-- Name -->
    <Path file="cube.3ds" />      <!-- Dateipfad -->
    <!-- Höhe und Breite in Pixeln -->
    <Size height="25.0" width="25.0" />
    <Pos x="304.0" y="128.0" z="0.0" />    <!-- Positionskoordinaten -->
    <Scale x="1.0" y="1.0" z="1.0" />      <!-- Skalierung -->
    <Rotation x="0.0" y="0.0" z="0.0" />   <!-- Rotation -->
    <Texture file="textures/rock.jpg" />   <!-- Textur -->
    <Tag string="" />                      <!-- Tag -->
  </Object>
  …                                      <!-- weitere Objekte -->
</Objects>
```

Die wichtigsten Eigenschaften eines Objekts sind der Dateipfad zum Drahtgittermodell und seine Position auf dem Terrain. Werte wie Skalierung oder Rotation beschreiben, wie das Drahtgittermodell manipuliert werden soll. Im Tag werden die Informationen, die dem Objekt im Editor zugeordnet werden, in String Form gespeichert. Die Werte für Höhe und Breite sind nicht unbedingt erforderlich, können aber für den weiteren Einsatz der Karte nützlich sein. Sie berechnen sich aus den Ecken der Bounding Box und der Skalierung:

$$height = (edges[1].Y - edges[0].Y) * scale.Y;$$
$$width = (edges[4].X - edges[0].X) * scale.X;$$

terrain.xml

```
<Terrain>
  <Node pos="0" isEmpty="0">        <!-- Position; Existenz -->
    <Path file=" textures/water.jpg" />  <!-- Dateipfad zur Textur -->
    <Tag string="" />                    <!-- Tag -->
    <Height value="0" />                 <!-- Höhe in Pixeln -->
    <Scale x="1.0" y="1.0" />            <!-- Skalierung -->
    <!-- Rotationsrichtung und Winkel -->
    <Rotation direction="" angle="0" />

    <!-- Heightmap mit Graustufenbild, Textur und maximaler Höhe -->
    <Heightmap map="maps/map123/Heightmaps/HM_152.png"
      texture="textures/floor_01.jpg" maxheight="50.0" />
  </Node>
  …                                      <!-- weitere Kacheln -->
</Terrain>
```

Jede Kachel bekommt eine Position zugeordnet. Die Nummerierung folgt dabei dem in *Abbildung 10* dargestellten Prinzip. Die Koordinaten müssen also anhand der Position und den Werten aus der options.xml berechnet werden. isEmpty ist eine zusätzliche Angabe, mit der eine Kachel aus dem Terrain entfernt werden kann. Sie wird nur dann geladen, wenn isEmtpy gleich null ist. Die Höhe einer Kachel ergibt sich aus dem Betrag ihrer z-Koordinate. Die Rotationsrichtung wird mittels der Himmelsrichtungen beschrieben (NORTH, EAST, SOUTH, WEST). Sie folgt dabei dem System aus *Abbildung 40*.

Abbildung 40: Zuordnung von Himmelsrichtungen und Neigungsrichtungen

Für Heightmaps wird ein Unterordner („Heightmaps“) angelegt. In diesem werden alle Graustufenbilder gespeichert. In der XML-Datei sind neben dem Pfad zur Bilddatei noch der Pfad zur Textur und die maximale Höhe der Heightmap wichtig.

8. Ausblick

An dieser Stelle endet meine Entwicklung des Editors. Die Möglichkeiten sind aber noch längst nicht ausgeschöpft. Bisher wurde ein Editor entwickelt, der viele allgemein nützliche Funktionen und Möglichkeiten beinhaltet. Eine Spezialisierung für einen bestimmten Einsatzzweck ist nicht vorhanden. Der Editor kann als Vorlage betrachtet werden, die um Funktionen erweitert werden kann, die für die Verwendung der Karten in bestimmten Programmen benötigt werden. Bei einer Karte für ein RTS Spiel werden z.B. viele Informationen benötigt, die den Spielablauf betreffen.

Szenarien

In RTS Spielen werden Szenarien entwickelt, die auf einer Karte gespielt werden können. Dazu müssen verschiedene Informationen festgelegt werden, wie z.B. Startpositionen der Spieler, Verhalten der Gegner oder Siegesbedingungen. Zur Umsetzung könnte der Editor eine Liste an vorgefertigten Tags anbieten.

Terrain

Bisher ist das Terrain immer rechteckig. Dies könnte verändert werden, indem Kacheln beliebig aneinander gelegt werden können. Auch Kacheln müssen nicht immer quadratisch sind. Sie könnten unterschiedliche Formen und Größen bekommen. Eine rechteckige Form sollte aber möglichst eingehalten werden, da sonst verschiedene Berechnungen unnötig aufwendig werden.

Literaturverzeichnis

GLSL Tutorial [Online]. - http://www.lighthouse3d.com/opengl/glsl/.

Noise and Turbulence [Online] / Verf. Perlin Ken. - http://mrl.nyu.edu/~perlin/doc/oscar.html.

OpenGL [Online]. - http://www.opengl.org/sdk/docs/tutorials/Lighthouse3D/.

Per-Pixel Displacement Mapping with Distance Functions [Online] / Verf. Donnelly William. - http://http.download.nvidia.com/developer/GPU_Gems_2/GPU_Gems2_ch08.pdf.

Programming An RTS Game With Direct 3D [Buch] / Verf. Granberg Carl. - 2007.

Realtime Procedural Terrain Generation [Online] / Verf. Olsen Jacob. - http://oddlabs.com/download/terrain_generation.pdf.

Terrain Analysis in Realtime Strategy Games [Online] / Verf. Pottinger Dave C. - http://www.gamasutra.com/features/gdcarchive/2000/pottinger.doc.

The Continuous World of Dungeon Siege [Online] / Verf. Bilas Scott. - http://www.drizzle.com/~scottb/gdc/continuous-world.htm.

Tile/Map-Based Game Techniques: Handling Terrain Transitions [Online] / Verf. Michael David. - http://www.gamedev.net/reference/articles/article934.asp.

Tiled Terrain [Online] / Verf. Peasley Mark. - http://www.gamasutra.com/view/feature/3028/tiled_terrain.php.

Abbildung 1: http://wire.ggl.com/wp/wp-content/uploads/2007/11/dune2_2.jpg
Abbildung 2: http://www.soldiernet.de/news/cngold.jpg
Abbildung 3: http://www.mpbygames.com/game/windows/total-annihilation/screenshots/gameShotId,6208/