ERSTELLUNG EINER AUF BLÖCKEN AUFBAUENDEN SPIEL-ENGINE IN JAVA

Schriftlicher Arbeitsbericht der besonderen Lernleistung

Benedikt Vogler

2012/2013

Erasmus-von-Rotterdam-Gymnasium

Betreuender Lehrer: Egon Klaus

INHALTSVERZEICHNIS

Vorwort 4

1. Einleitung 5

1.1. Basis der Engine 5

1.2. Allgemeine Hinweise 5

2. Block-Spiel-Engines 7

2.1. Was ist eine Spiel-Engine? 7

2.2. Was ist eine Block-Welt? 8

2.3. Abgrenzung zu Voxel-Welten 8

2.4. Unterschiede einer gerasterten Block-Welt gegenüber einer rasterlosen Polygon-Welt 9

2.5. Geschichte der Block-Spiele-Welten 10

3. Perspektive 12

3.1. 2,5D 12

3.2. Axonometrie 13

3.3. Besonderheiten der Isometrie 15

4. Der Aufbau der Engine 16

4.1. Der Klassenaufbau 16

4.2. Der Game Loop 16

4.2.1. Konstante oder variable Bildfrequenz 17

4.2.2. Die Klasse Gameplay 17

4.3. Der Controller und der View 17

4.4. GameObject 18

4.4.1. Das Feld value 18

4.4.2. Die weiteren Felder 19

4.4.3. Fabrik-Methode statt Konstruktor 20

4.4.4. Spezialobjekte 20

4.5. Map 21

4.5.1. Entitys 22

4.5.2. Streamen der Karte mit Chunks 22

4.5.3. Formender Blockanordnung 23

4.5.4. Verschiedene Anordnungen der Chunks 25

4.5.5. Der Chunkwechsel 25

4.5.6. Das Speicherformat 27

4.6. Spielfiguren 28

4.6.1. Die Bewegung 28

4.6.2. Der Koordinatenwechsel 29

4.6.3. Kollisionsprüfung 30
4.6.4. Sounds 30
5. Rendering 31
5.1. Besonderheiten beim Rendering von 2D-Grafik 31
5.2. Der einfache Rendering-Algorithmus 32
5.3. Das Rendering von zerteilten Blöcken 34
5.4. Sichtbarkeitsbestimmung mittels Raytracing-ähnlichem Verfahren 35
5.5. Tiefensortierung 36
5.6. Der erweiterte Rendering-Algorithmus 37
5.7. Skalierung 37
6. Mögliche Erweiterungen 39
6.1. Erweiterte Licht-Engine 39
6.2. Splitscreen 39
6.3. Mehr Konfigurationsoptionen 40
7. Werkzeuge 41
7.1. Der Map Editor 41
7.2. Sprite-Werkzeuge 41
8. Auswertung 42
8.1. Der Einsatz für Spiele 42
8.2. Erreichte Zielsetzung 43
Literaturverzeichnis 44
Abbildungsübersicht 45
Anhang 47

VORWORT

Jeder dritte Deutsche spielt Computer- und Videospiele. Laut dem Bundesverband Interaktive Unterhaltungssoftware (BIU) sind es in Deutschland 25 Mio. Menschen, die mehrmals im Monat spielen. Obwohl so weit verbreitet, fanden und finden Computerspiele bei Wissenschaftlern, Politikern und breiten Teilen der Gesellschaft nur wenig Akzeptanz.

Technisch gesehen sind Computerspiele sehr bedeutend. Sie sind u.a. Motor für den Hardwareverkauf (Spielkonsolen, Grafikkarten, Zubehör), da moderne Spiele jedes Jahr an die technischen Grenzen der verfügbaren Hardware stoßen, außerdem sind die benutzten Verfahren für Computerspiele wie Wegfindungs-, KI[1]- oder Rendering[2]-Algorithmen hochkomplex und finden nicht nur bei Computerspielen ihren Einsatz.

Ich sehe die vorliegende Arbeit als Beitrag für ein besseres Verständnis dieses Mediums, indem ich versuche, die Kultur und komplexe Informatik hinter den Spielen ein Stück weit aufzuschlüsseln und in dem hier möglichen begrenzten Umfang darzustellen.

[1] Abk. künstliche Intelligenz.

[2] Rendering bezeichnet die Umwandlung von Daten zu einer Grafik.

1. EINLEITUNG

Die Absicht der vorliegenden Arbeit ist die Erstellung einer Spiel-Engine, also eines Softwaresystems, auf dem sich Spiele aufbauend erstellen lassen. Eine Spiel-Engine sollte dies für möglichst viele Typen erlauben. Jede Spiel-Engine ist auf eine Spielart spezialisiert.[3] Hier ist es die aus Blöcken bestehende Welt in der isometrischen Ansicht.

Die Engine wird so konstruiert, dass später nötige Modifikationen für damit umgesetzte Spiele einfach durchzuführen sind.

Aufgrund der Breite des Themas werden in dieser Arbeit die wichtigsten Kernelemente einer isometrischen Spiel-Engine erläutert.

1.1. Basis der Engine

Die Umsetzung der Engine erfolgt mit Java 7 und der Open Source API Slick2D[4]. Slick2D baut auf LWJGL[5] auf, einer Schnittstelle für Java für OpenGL[6]. Es bietet Methoden zum Erstellen des Grundgerüstes (z.B. Game Loop, siehe Kapitel 4.2.) sowie für eine Implementierung des Tons[7] und der grafischen Ausgabe.

1.2. Allgemeine Hinweise

Die Engine wird in der Arbeit „Wurfelengine“ genannt. Der Name kommt von Würfel-Engine, also einer Engine für Würfel. Der Umlaut „ü“ wurde zu einem „u“ geändert, damit der Name international verständlich ist, was gleichzeitig noch einen kleinen sprachlichen „Witz“ für deutschsprachige Menschen beinhaltet.

Die Wurfelengine steht unter der BSD-3 Lizenz[8] und ist damit Open Source. Es gibt ein GitHub-Repositorium unter https://github.com/Cbeed/Wurfelengine, unter der die aktuellste Version zu finden ist.

---

[3] Vgl. Gregory, Jason: Game Engine Architecture. (Massachusetts 2009) S. 12.

[4] http://www.slick2d.org/ (letzter Zugriff 22.03.2013).

[5] Lightweight Java Game Library.

[6] Plattform- und programmiersprachenunabhängige Programmierschnittstelle für Computergrafik.

[7] Slick 2D benutzt OpenAL (Open Sound Library), eine offene Programmierschnittstelle für Audio.

[8] http://opensource.org/licenses/BSD-3-Clause (letzter Zugriff 22.03.2013).

Die Wörter „Block“ und „Würfel“ werden künftig äquivalent benutzt. Anzumerken ist, dass der Begriff „Block“ in Bezug auf Computerspiele häufiger zu finden ist als das Synonym „Würfel“.

2. BLOCK-SPIEL-ENGINES

2.1. Was ist eine Spiel-Engine?

(Computer-)Spiele sind gewöhnlich modifizierte, modellartige Simulationen der Wirklichkeit. Eine Spiel-Engine enthält alle Kernmethoden für die Verwaltung der Daten, Physik, Grafik und Ton, um diese einfach bilden zu können . Wichtig für den Erfolg eines Spieles ist die Spiel-Engine, also ihre Qualität[9] und der Funktionsumfang. Mit der Wahl der Engine werden implizit die Grenzen der Spielwelt des auf ihr aufbauenden Spiels festgelegt.

Die erste Spiel-Engine, die als Grundlage für zahlreiche Spiele verwendet wurde, hatte der Ego-Shooter „Doom" (1993).[10] In Doom wurde zwischen Spielinhalt und Engine unterschieden, was einen entscheidenden Vorteil hatte: „The value of this separation became evident as developers began licensing games and re-tooling them into new products […] with only minimal changes to the ‚engine' software."[11] Viele Spiele bauten in den folgenden Jahren auf dieser Engine auf.[12]

Produktionskosten lassen sich also durch die Lizenzsierung oder Produktion einer eigenen Spiel-Engine senken, da das Grundgerüst in einem Genre oder einer Reihe ähnlich ist und eine Neuprogrammierung der Grundlagen für jeden neuen Titel damit unnötig wird.

Komplexe 3D-Engines wie z.B. die *Unreal Engine* haben einen sehr breiten Funktionsumfang. Sie bieten viele Funktionen wie künstliche Intelligenz, Physiksimulation, Rendering und Terrainerstellung an.[13]

---

[9] Die Qualität betrifft hier die Genauigkeit der simulierten physikalischen Phänomene, Stabilität, Darstellungsqualität im Verhältnis zur Leistung sowie den Funktionsumfang der Engine. Da ein Spiel die Wirklichkeit abbildet, sollte die Simulation möglichst überzeugend sein. Wenn z.B. physikalischen Phänomene simuliert werden, sollten diese auf der alltäglich erfahrbaren und damit auf der Newton'schen Physik aufbauen.

10 Gregory S. 11.

11 Ebenda.

[12] http://de.wikipedia.org/wiki/Doom#Weitere_Spiele_auf_Basis_der_Engines (letzter Zugriff 22.03.2013).

[13] http://www.unrealengine.com/en/features/ (letzter Zugriff 22.03.2013).

2.2. Was ist eine Block-Welt?

Eine Block-Welt ist eine virtuelle Welt, die auf einem dreidimensionalen Raster aufbaut. Elemente dieses Rasters müssen Blöcke, also dreidimensionale (3D) Objekte sein. Allerdings können auch sehr viele zweidimensionale (2D) Spiele dieser Kategorie hinzugezählt werden, da diese eine dreidimensionale Welt in einer 2D-Perspektive darstellen. Spiele wie z.B. „Tick-Tack-Toe“ oder „Vier in einer Reihe“ gehören nicht in diese Kategorie, da sie keine virtuelle dreidimensionale Welt bilden.

Die Auffassung von diesen Welten als „Block-Welten“ ist ein eher jüngeres Phänomen, da der rasterförmige Aufbau früher bei den meisten Spielen der Normalfall war. Erst seit einigen Jahren werden im Raster angeordnete Blöcke als bewusstes Gestaltungsmittel eingesetzt, da der rasterförmige Aufbau heute technisch nicht mehr zwingend notwendig ist.

Die Wurfelengine benutzt ein dreidimensionales Raster für den Aufbau der Welt (siehe Abb. 1).

Abb. 1: Screenshot des Rasters.

2.3. Abgrenzung zu Voxel-Welten

Oft werden die Würfel in Block-Welten auch als Voxel, eine Mischung zwischen „volumetric“ (engl. volumetrisch) und „Pixel“, also volumenhaltige Pixel, bezeichnet. Die Bezeichnung für Block-Welten ist aber nicht treffend, denn bei Spielen mit Block-Welten bestehen die Blöcke aus vielen Pixeln, wohingegen Voxel-Welten nicht mehr in kleine Einheiten aufgeteilt werden können. Denn Voxel, als 3D-Pixel, sind die kleinste Einheit in der Computergrafik.[14] Daraus resultieren große Unterschiede im Maßstab des Welt-

[14] http://d-nb.info/gnd/4534766-9/about/html (letzter Zugriff 22.03.2013).

enrasters, der gut an den Charakteren erkennbar wird. Bei Block-Welten entspricht oft eine Rastereinheit der Länge 1m in der Spielwelt, bei Voxel-Spielen entspricht dieses meist 5-20 Zentimetern. In Abb. 2 stehen Spielfiguren aus einer Block- und einer Voxel-Welt zum Vergleich.

Abb. 2: Spielcharakter im Spiel „3D Dot Game Heroes“ und „Minecraft“.
Der Charakter aus dem Voxel-Spiel „3D Dot Game Heroes“ ist 15 Rastereinheiten hoch, während beim Block-Spiel „Minecraft“ die Spielfigur zwei Rastereinheiten groß ist. Deutlich sind auch die Pixel in den Texturen bei „Minecraft“ zu sehen, während die Voxel bei „ 3D Dot Game Heroes“ einfarbig sind.

2.4. Unterschiede einer gerasterten Block-Welt gegenüber einer rasterlosen Polygon-Welt

Block-Welten haben zwei Besonderheiten gegenüber gewöhnlichen Spielwelten:

- Die Welt besteht aus Würfeln.
- Es gibt ein Raster, in dem diese Würfel angeordnet werden.

Das Rendering von Block-Welten geschieht mithilfe derselben Algorithmen wie bei 3D- oder normalen 2D-Welten, allerdings können dank der zwei genannten Besonderheiten einige Optimierungen vorgenommen werden.

3D-Spiele die mit Polygonen, also Körpern aus vielen Dreiecken, arbeiten[15] zielen meisten auf eine möglichst große Übereinstimmung der Spielwelt mit der Wirklichkeit. Daher nimmt man bei 3D-Spielen fast immer Körper aus möglichst vielen Dreiecken, die frei angeordnet werden.

Bei Block-Welten entfällt diese freie Anordnung. Das eigentliche Abbild der realen Welt wird in ein simpleres System gefügt. Durch den rasterförmigen tabellarischen Aufbau wird die Welt stilisiert, die Komplexität verringert und die Welt stärker kontrollierbar. Die kreative Leistung sowohl des Entwicklers als auch des Spielers wird erhöht, weil die

[15] In der Computergrafik kann, bis auf wenige andere Rendering-Verfahren, die z.B. auf „Atomen“ aufbauen, nur mit Polygonen ein dreidimensionales Bild erzeugt werden.

Freiheit der Gestaltung durch die Begrenzung des Rasters konzentriert wird. Eine simple Nachbildung der Realität ist also nicht mehr möglich, und dies evoziert eine vermehrt künstlerische Arbeit, ähnlich der Minimal- und Pixel-Art[16].[17] Dadurch ist eine Nähe von Block-Welten und Pixel-Art gegeben. Spieler bauen z.B. im Spiel „Minecraft“ aus den Blöcken Pixel-Art-Bilder (vgl. Abb. 3).

Abb. 3: Eine Pixel-Art Grafik in Minecraft.

2.5. Geschichte der Block-Spiele-Welten

Da das Rendering mit sogenannten „Tilemaps“ anspruchsvolle rasterförmige 2D-Landschaften mit geringem Speicherverbrauch erzeugen konnte, gibt es Block-Welten seit den ersten Computerspielen. Beispiele sind Klassiker wie „Q-Bert“ (1982), „Super Mario Bros.“ (1985) (vgl. Abb. 4) oder „The Legend of Zelda“ (1986).

Abb. 4: Screenshot aus Super Mario Bros.

Nachdem der Egoshooter „Wolfenstein 3D“ 1992 das Zeitalter der 3D-Spiele einläutete[18], gab es lange Zeit kaum Block-Welten in erfolgreichen Titeln.[19] Das Ziel bei großen Titeln war es, u.a. den größten Fortschritt

[16] Eine Form der Computerkunst, bei der manuell Pixel für eine Grafik gesetzt werden.

[17] Die Stilisierung benötigt Abstraktion, also künstlerisches Wirken. Vgl. http://www.beyars.com/kunstlexikon/lexikon_8702.html (letzter Zugriff 22.03.2013).

[18] http://www.schnittberichte.com/schnittbericht.php?ID=3597 (letzter Zugriff 22.03.2013).

[19] Eine Ausnahme bildet die Reihe „Pokémon“, welche noch mit Tilemaps anfing und selbst mit dem Umstieg auf 3D-Grafik den blockartigen Aufbau bis heute teilweise beibehalten hat. Eine Übersicht weiterer bedeutender Computerspiele gibt es unter http://de.wikipedia.org/wiki/Zeittafel_der_Computerspiele (letzter Zugriff 22.03.2013). Ab dem Jahre 1992 finden sich kaum noch Spiele mit Block-Welten.

mit einer realistischeren Grafik zu erreichen. Zwar erschien 2002 der freie Ego-Shooter „Cube“[20], der auf der Cube Engine, einer 3D-Block-Engine, aufbaut, doch die Computerspiele-Szene blieb unverändert. Block-Welten schienen sich nur im technisch anspruchsloseren 2D- und 2,5D[21]-Bereich durchzusetzen, wobei 2,5D-Spiele nur selten auf Block-Welten setzten.

Aufgrund des großen Erfolges des bereits erwähnte Spieles „Minecraft“ (2011) setzte es den Trend für veränderbare 3D-Block-Welten. Minecraft nutzt die Block-Welt nicht nur zur Darstellung, wie es fast alle bisherigen Spiele taten. Die Blöcke sind hier entscheidend für die Spielmechanik, denn der Spieler kann seine Umwelt frei verändern, was in einer dreidimensionalen Polygon-Welt sehr schwierig ist. Spiele wie „Ace of Spades“ (2012), „Brickforce“ (2012), „FortressCraft“ (2011) und „Blockscape“ (noch unveröffentlicht) folgten. Dieses Prinzip der veränderbaren Umwelt griffen dann wiederum 2D-Blockspiele auf, wie z.B „Terraria“ (2011) oder „Blocks That Matter“ (2011).

[20] http://web.archive.org/web/20030306040043/http://www.intergate.com/~spentron/cube/cube.html (letzter Zugriff 22.03.2013)

[21] Eine genauere Erklärung des Begriffs findet sich in Kapitel 3.1.

3. PERSPEKTIVE

3.1. 2,5D

3D-Darstellungen (vgl. Abb. 5), benötigen eine große Rechenleistung und einen großen Aufwand bei der Programmierung und Erstellung der Grafiken. Einfacher Umzusetzen sind zweidimensionale Projektionen. Wenn trotz des Verzichts auf 3D-Grafik ein realitätsnäheres Bild erzeugt werden soll, wird dies mit 2D-Rendermethoden, aber drei Raumachsen erzeugt, genannt „2,5D“ (vgl. Abb. 6). Der Unterschied zu 3D-Spielen liegt in der statischen Kamera, welche sich bei 2,5D nicht um Achsen rotieren lässt, und die fehlende perspektivische Verzerrung, genannt Parallelperspektive.

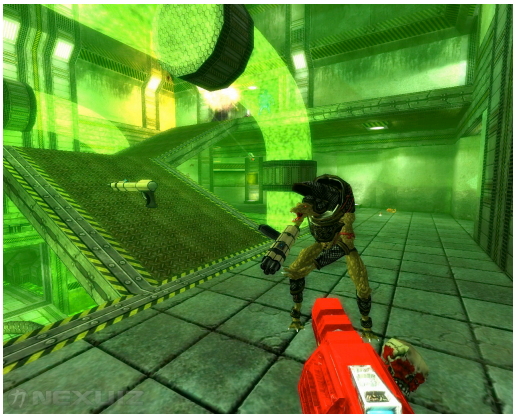


Abb. 5: Screenshot vom Spiel Nexuiz (3D). Es hat eine bewegliche und rotierbare Kamera. Objekte in der Ferne erscheinen kleiner

Abb. 6: Screenshot vom Spiel Sacred (2,5D). Es hat eine beweglichen, aber nicht rotierbaren Kamera. Außerdem gibt es keine Achsenverkürzung.

Bei 3D-Spielen wird der sichtbare Bereich („Viewport“) als Pyramidenstumpf (vgl. Abb. 7) modelliert[22], bei 2D- oder 2,5D-Spielen lässt sich dieser hingegen als Quader darstellen.

[22] In der 3D-Grafik wird dies als Frustum, vom englischen Wort für „Pyramidenstumpf“, bezeichnet (vgl. Gregory, Jason: Game Engine Architecture S. 184).

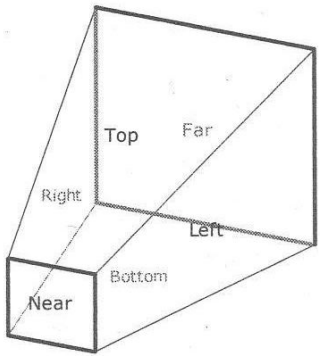


Abb. 7: Der Viewport.
Der sichtbare Bereich in einem 3D-Spiel wird als Kegelstumpf modelliert.

3.2. Axonometrie

Soll ein Spiel mit 2,5 Dimensionen dargestellt werden, wird oft die axonometrische[23] Projektion gewählt. Bei dieser sind die Projektionsstrahlen parallel zueinander und orthogonal zur Projektionsfläche und verlaufen nicht entlang einer der Raumachsen.[24] Dies bedeutet, dass die Fluchtpunkte ins Unendliche gerückt werden und sich damit keine Verkürzung in der Ferne ergibt. Parallelen bleiben damit sichtbar parallel. Das Koordinatensystem sieht dann wie in Abb. 8 dargestellt aus.

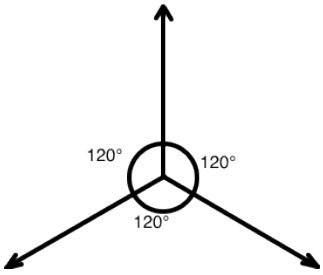


Abb. 8: Das isometrische Koordinatensystem.

Die Axonometrie wird in drei Projektionen unterteilt: „Trimetrie", „Dimetrie" und „Isometrie", abhängig von der Anzahl der gleichen Winkel zwischen den Achsen.[25] Die trimetrische Axonometrie hat keine gleichen Winkel zwischen den Achsen, die Dimetrie hat zwei gleiche Winkel und die Isometrie drei. Die Wurfelengine benutzt die Isometrie.

Wie in Abb. 8 sichtbar, sind bei der Isometrie alle Achsen 120° zueinander gedreht und haben den gleichen Maßstab. Zwei Achsen müssen daher im Winkel von 30° zur Horizontalen verlaufen. Bei Geraden, die im Pixelraster abgebildet werden und im Winkel von 30° verlaufen, steht jeder siebte Pixel alleine und nicht als Paar, was durch die Unre-

[23] Griechisch áxōn „Achse" und métron „Maß".

[24] www-gs.informatik.tu-cottbus.de/gcg1_v06.doc S. 6 (letzter Zugriff 22.03.2013).

[25] Vgl. http://www.significant-bits.com/a-laymans-guide-to-projection-in-videogames (letzter Zugriff 22.03.2013).

gelmäßigkeit eher unästhetisch wirkt (vgl. Abb. 9). Wird hingegen ein Winkel von 26,56° verwendet, kann eine strengere Regelmäßigkeit erreicht werden (vgl. Abb. 10).

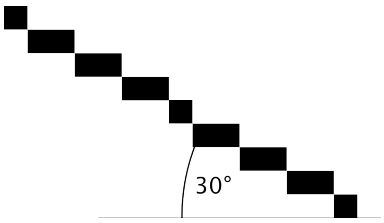


Abb. 9: Eine gepixelte Linie im Winkelt von 30°.

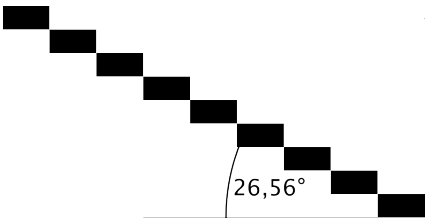


Abb. 10: Eine gepixelt Linie im Winkelt von 26,56°.

Linien wie in Abb. 10 eignen sich also besser, da hier ein ähnlicher Winkel mit einer strengeren Regelmäßigkeit erreicht wird. So ist die Projektion nicht mehr streng isometrisch, sondern eigentlich dimetrisch. Das Ziel bei der Dimetrie in Computerspielen ist die Nachahmung von Isometrie unter Berücksichtigung der zuvor genannten Problematik bei einem Winkel von 30°. Daraus ergibt sich eine Winkelverteilung von 116,56°, 116,56° und 126,88°, die zu etwa 97 % mit der Isometrie übereinstimmt. Aufgrund der großen Übereinstimmung der auf diese Weise benutzten Dimetrie mit der Isometrie, ist nur mit geschärftem Blick der Unterschied erkenntlich, zudem die Dimetrie gewöhnlich mit anderen Winkelverteilungen verwendet wird.[26]

Insbesondere in der frühen Zeit der Computerspiele war die isometrische Darstellung die aufwendigste. Die Geräte hatten eine geringe Auflösung, d.h. der hier beschriebenen Effekt fiel viel stärker ins Gewicht als bei heutigen Bildschirmen. Antialiasing[27], als eine Lösung dieses „Treppen"-Problems, bringt das Problem mit sich, dass die Linien dadurch weniger klar erscheinen. 26,56°-Winkel werden auch heute, trotz höherer Auflösung und Antialiasing, wegen ihrer klaren Struktur bevorzugt verwendet.

An dieser Stelle sei auf die Diplomarbeit „Entwicklung einer isometrischen Graphik-Engine in Java"28 von Thomas Schusters verwiesen, die die Geschichte der Isometrie in Computerspielen näher beleuchtet.

---

[26] Z.B. DIN ISO 5456-3, dort auch mit einer Achsenverkürzung.

[27] Auch als Kantenglättung bezeichnet; ein Verfahren zur Verminderung von unerwünschten Effekten, z.B. Treppeneffekt. Dabei werden Helligkeits- und Farbabstufungen an kontrastierenden Kanten vorgenommen, um einen „weichen" Übergang zu erreichen.

[28] http://www.uni-koblenz.de/~cg/Diplomarbeiten/DA_ThomasSchuster_Web.pdf (letzter Zugriff 22.03.2013).

3.3. Besonderheiten der Isometrie

Ein Block wird isometrisch wie folgt dargestellt: Man blickt auf einen Block mit dem Seitenverhältnissen 1:1:1, der um 45° zur frontalen Ansicht zur Seite gedreht und um 35,26° vertikal gekippt ist, so dass man drei Seiten sehen kann (vgl. Abb. 11). Die Winkelbeziehungen der Seiten ergeben sich aus der Isometrie.

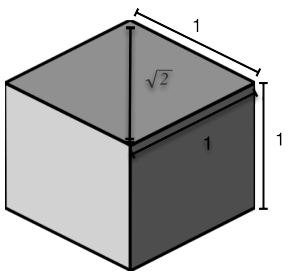


Abb. 11: Ein Block in isometrischer Ansicht.

Zur Berechnung des Betrachtungswinkels bildet man ein Dreieck: Die Diagonale der Seiten (Ankathete) des Blocks ist nach Pythagoras $\sqrt{1^2+1^2}=\sqrt{2}$ lang, die Höhe (Gegenkathete) beträgt 1. Daraus folgt der Betrachtungswinkel $\arctan\left(\frac{1}{\sqrt{2}}\right)\approx 35{,}26°$.

Durch die Isometrie und den rasterförmigen Aufbau wird der jeweils hintere, eine Ebene tiefer liegende Block, wie in Abb. 12 verdeutlicht, komplett verdeckt. Dies kann man, wie in Kapitel 5.4 beschrieben, für Optimierungen nutzen.

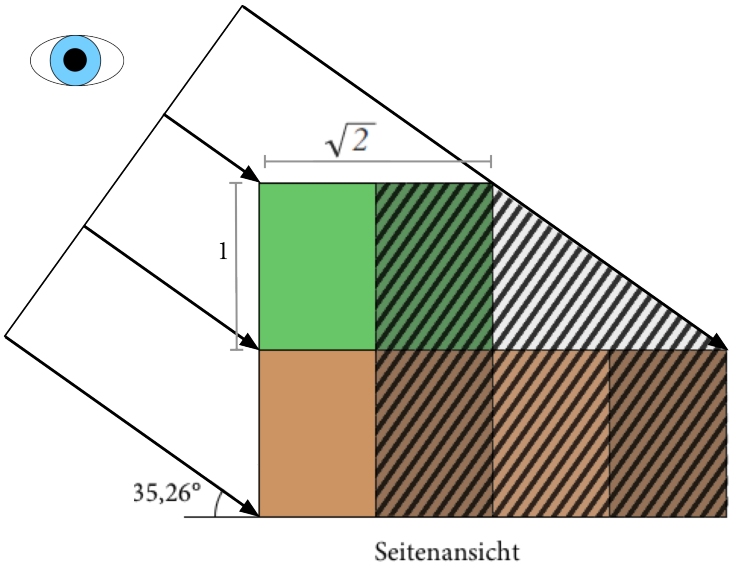


Abb. 12: Die hinteren Blöcke werden bei der Isometrie verdeckt.

4. DER AUFBAU DER ENGINE

4.1. Der Klassenaufbau

Im Folgenden werden die Kernfunktionen und die Klassenarchitektur erläutert (vgl. Anh. 1). Für den genauen Aufbau der einzelnen Klassen sollte die Klassendokumentation (Javadoc[29]) oder der Quellcode betrachtet werden.

4.2. Der Game Loop

Computerspiele laufen fast immer in Echtzeit[30] ab. Daher müssen die Grafik und die Daten ständig überprüft und aktualisiert werden. Dies erreicht man mithilfe einer Endlosschleife, genannt *Game Loop*. Ein Game Loop prüft bei jedem Durchlauf die Eingabe des Benutzers, aktualisiert dann die Spiellogik und rendert schließlich die Szene (vgl. Abb. 13).

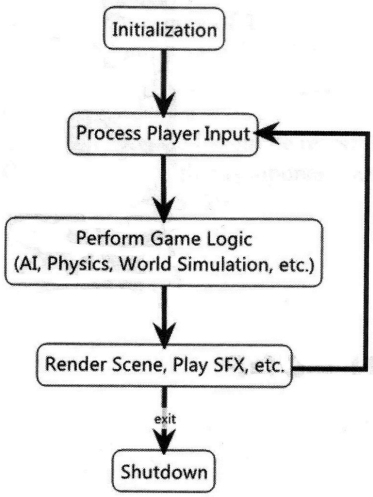


Abb. 13: Ein üblicher simpler Game Loop.

Slick bietet in den Klassen *StateBasedGame* oder *BasicGame* die Funktion für einen simplen Game Loop bereits an. Die Wurfelengine benutzt dafür die Klasse Gameplay die von *StateBasedGame* erbt. In der Wurfelengine wird die Spiellogik in den update-Methoden aktualisiert, während das Rendering der Grafik in den render-Methoden entsprechend stattfindet.

Bei Computergrafiken wird zur Messung der Geschwindigkeit (framerate[31]) die Einheit FPS (frames per second) genutzt, die der Einheit Hz (Hertz) entspricht. Bei Computerspielen sollte die Anzahl der Aktualisierungen nicht unter 60 in der Sekunde fallen, da

[29] Javadoc ist ein Werkzeug für die automatische Erstellung einer HTML-Dokumentation aus dem Java-Quelltext. Die damit generierte Dokumentation wird teilweise auch Javadoc genannt.

[30] Echtzeit meint die Geschwindigkeit der Modellierung von Prozessen, welche in der Realzeit stattfinden. Im Bezug zu Computerspielen wird dieser Begriff hauptsächlich zur Abgrenzung von runden-basierten Spielen verwendet.

[31] Rate der Einzelbilder (=frames).

sonst die Bewegungen nicht mehr flüssig erscheinen. Zudem werden viele Monitore bei 60 Hz betrieben.[32]

4.2.1. Konstante oder variable Bildfrequenz

Eine konstante Framerate kann für manche Features (z.B. eine Playback-Funktion) von existentieller Bedeutung sein oder die Qualität dieser verbessern, da die Zeit zwischen einem Frame immer die gleiche ist.[33] Während der Entwicklung bleibt die Framerate variabel, um die Geschwindigkeit der Algorithmen zu messen. Für die Veröffentlichung eines Spiels jedoch wäre es angebracht, die Framerate bei 60 FPS einzustellen.[34]

4.2.2. Die Klasse Gameplay

Da die Klasse *Gameplay* von der von Slick bereitgestellte Klasse *BasicGameState* erbt, sind die entscheidenden Methoden *update* und *render* bereits implementiert. *update* ist für die Aktualisierung der Daten und die Annahme der Inputs zuständig, während *render* die Daten darstellt.

4.3. Der Controller und der View

Wie auch beim Game Loop Aktualisierung und Rendering getrennt werden, ist es sinnvoll, die Verwaltung der Daten von der Darstellung jener zu trennen. Dazu benutzt die Wurfelengine für das Speichern und Verwalten der Daten die Klasse *Controller* (vgl. Anh. 4) und für die Verwaltung der Anzeigen die Klasse *View* (vgl. Anh. 5). Objektinstanzen dieser Klassen werden in der Klasse *Gameplay* gespeichert.

McShaffry und Graham beschreiben einen View „als Sammlung von Systemen, welche mit der Spiellogik kommunizieren, um das Spiel einem speziellen Beobachter zu präsentieren“[35]. Ein Beobachter ist z.B. der Spieler oder eine künstliche Intelligenz. Bei McSharrfy und Graham übernimmt der View nicht nur die Darstellung, sondern auch die Tonausgabe und die Interpretation des Inputs und mit dieser auch Teile der Logik.

---

[32] Weitere Informationen zum Thema gibt es bei Wikipedia: http://de.wikipedia.org/wiki/Bildwiederholfrequenz (letzter Zugriff 22.03.2013).

[33] Gregory, S. 313f.

[34] Vgl. Gregory, S. 315.

[35] Eigene Übersetzung von: „A game view is collection of systems that communicates with the game logic to present the game to a particular kind of observer.“ (McShaffry, Mike / Graham, David „Rez“: Game Coding Complete, 7. Aufl. Boston 2012, S. 41).

Die Interpretation des Inputs findet bei der Wurfelengine im Controller statt, und der View beschränkt sich auf die Darstellung.

Um die Daten unabhängig von den Views zu machen, werden die Daten unabhängig von der Perspektive behandelt und gespeichert.

Die Wurfelengine hat ihren Hauptfokus auf der Spieleransicht, der zugehörige View, im Sinne von McShaffry und Graham, bildet die Klasse *Camera*. Ein weiterer View ist die Klasse *Minimap*, eine zusätzliche kleine Landkarte zur Orientierung für den Spieler.

Die Beziehung der Klassen lässt sich also wie folgt darstellen:

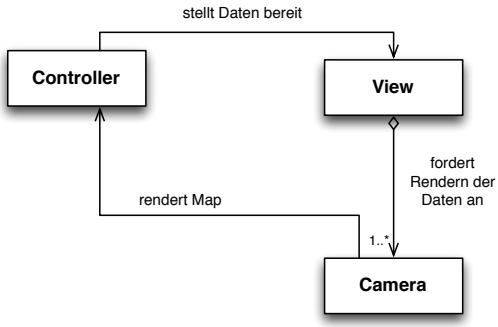


Abb. 14: Die Beziehung zwischen View und Controller.
Die Klasse Camera rendert die Hauptansicht.

4.4. GameObject

Ein `GameObject` (vgl. Anh. 6) ist ein Element, das sich der Spielwelt befindet. Es bildet daher das Grundelement der Engine und ist die Klasse mit den meisten Methoden. Die Klasse ist abstrakt, deswegen werden konkrete Implementierungen der Klasse benötigt. Diese bieten die Klassen *Block* (vgl. Anh. 7) und *Entity* (vgl. Anh. 8). Im jeden Rasterfeld der Spielkarte befindet sich ein Block, Entitys hingegen sind Objekte wie z.B. Lebewesen, Waffenprojektile oder einsammelbare Objekte („Pickups“) und weniger streng an das Raster gebunden sind.

4.4.1. Das Feld value

Es gibt verschiedene Typen von Spielobjekten, die sich in der Grafik und Funktion unterscheiden. Jeder Objekttypus erhält zur einfacheren Identifikation eine eindeutige ID-Nummer. Zusätzlich gibt es eine weitere Unterteilung, die je nach Objekt unterschiedlich behandelt wird. Ein Block mit der ID 15 ist etwa ein Lampe. Eine Subunterteilung durch das Feld `value` (integer) beschreibt in diesem Fall, ob die Lampe aus- (value = 0) oder

eingeschaltet (value = 1) ist. Ein anderer sinnvoller Verwendungszweck der Subunterteilung ist die Angabe der Drehung bei einem nicht regelmäßigen Körper, z.B. bei einem Zaun die Orientierung als oben links oder rechts endend, dargestellt in Abb. 15.

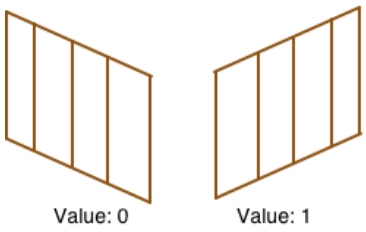


Abb. 15: Ein Block mit zwei Grafiken.
Die Value unterteilt diesen Block erneut.

4.4.2. Die weiteren Felder

Jedes Objekt hat noch eine Reihe weiterer Felder, welche bei der Erstellung initialisiert werden sollten.

Feldbezeichnung	Datentyp	Funktion
`transparent`	boolean	Angabe der Lichtdurchlässigkeit.
`lightlevel`	int	Helligkeit mit Wert zwischen 0 und 100.
`obstacle`	boolean	Ist das Objekt ein Hindernis oder nicht.
`hidden`	boolean	Ist das Objekt versteckt (=unsichtbar gemacht)?
`visible`	boolean	Ist das Objekt sichtbar?
`pos`	int[]	Die Positionierung innerhalb eines Rasterfeldes. Der Koordinatenursprung liegt bei jedem Block in der hinteren linken Ecke.
`dimensionY`	int	Die Höhe des Objekts (z.B. beim Spieler ist sie 2).

KONSTANTEN		
FELDBEZEICHNUNG	DATENTYP	FUNKTION
DIMENSION	int	Die Länge der Diagonalen.
DIM2	int	Die Hälfte der Diagonalen
DIM4	int	Ein Viertel der Diagonalen.
OBJECTTYPESCOUNT	int	Die Anzahl der maximal definierbaren Objekttypen.
GAMEDIMENSION	int	Die Kantenlänge, welche 1m entspricht.
SPRITEPOS	int[][][][]	Hier befinden sich die Informationen über die Position der einzelnen Spritegrafiken im Spritesheet.
NAMELIST	String[]	Es gibt eine Liste mit den Namen, bei der die Indexpositionen mit den IDs übereinstimmen, die z.B. für die Fehlersuche genutzt werden kann.

4.4.3. FABRIK-METHODE STATT KONSTRUKTOR

Anstatt die Objekte wie üblich über den Konstruktor zu erstellen, gibt es die zentrale Methode `public static Block getInstance(int id, int value, int[] coords)`, mit den verkürzten Konstruktoren `getInstance(int id, int value)` und `getInstance(int id)`, welche das Objekt als Rückgabewert enthalten. Der Vorteil dieser Art der Erstellung ist, dass es eine zentrale Stelle gibt, an der eine Zuordnung von Kindklassen zur ID geschieht. Kindklassen, also Spezialobjekte, wie z.B. *ExplosiveBarrel*, können wie gewöhnliche Blöcke (z.B. ein Erdblock), über die ID mit der Fabrik-Methode dynamisch erzeugt werden.

4.4.4. SPEZIALOBJEKTE

Die Wurfelengine bietet eine breite Basis von Klassen als Ausgangspunkt für weitere, eigene Klassen. In Abb. 16 ist das Klassendiagramm zum Package „Gameobjects“ abgebildet, welche alle Klassen der Spielobjekte enthält.

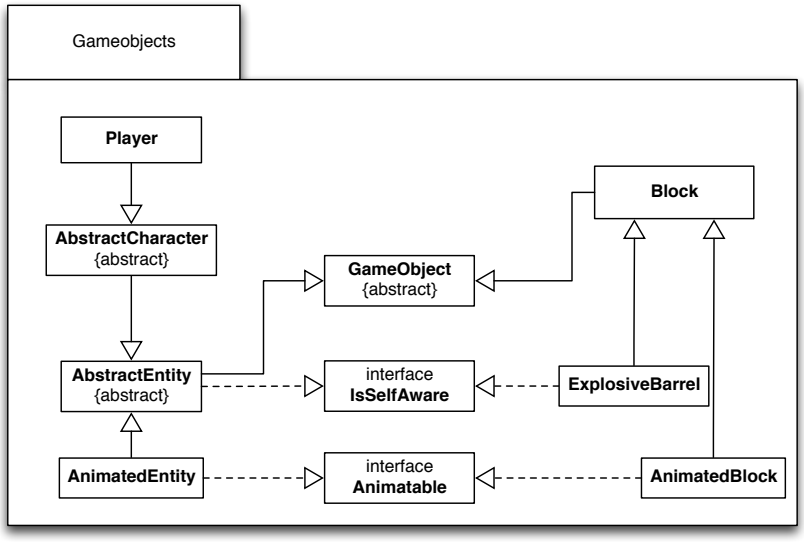


Abb. 16: Das Klassendiagramm des Package Gameobjects.

Um z.B. eine Spielfigur für den Spieler erstellen, wird die abstrakte Klasse `AbstractCharacter` verwendet, eine Klasse für Spielobjekte, die sich in der Karte bewegen können. Diese erbt von `AbstractEntity`, die Elternklasse für Entitys (vgl. Kapitel 4.5.1.). Die Entityklasse ist eine Kindklasse von `GameObject` mit Implementation des Interfaces `IsSelfAware`, eine Klasse für Objekte, die ihre Position in der Karte kennen. Ein Beispiel für einen konkreten `AbstractCharacter` ist die Klasse `Player` (Anh. 9).

Alternativ besteht die Möglichkeit die Hierarchien durch Aggregationen zu ersetzen, wodurch z.B. der *Player* dann nicht am Ende von drei Vererbungen stehen würde. Einige Autoren bevorzugen dies, so etwa John P. Flynt.[36]

4.5. Map

Eine „Map“ (engl. Karte) oder „Level“ bezeichnet in der Spielentwicklung einen Bereich, auf dem das Spiel stattfindet. Er kann je nach Spiel bis zu virtuelle 478.000 km² groß sein.[37]

Die Klasse Map (Anh. 2) der Wurfelengine beinhaltet Blöcke und formt durch die Anordnung dieser die Landschaft. Genannt wird dies *world space*: „World space is a fixed

[36] Flint, John P. [with Salem, Omar]: Software Engineering for Game Developers, Boston 2005, S. 213.

[37] http://www.pcgames.de/Panorama-Thema-233992/Specials/Auf-die-Groesse-kommt-es-an-Die-riesigsten-Spielwelten-der-Geschichte-Zahlen-und-Bilder-793252/ (letzter Zugriff 22.03.2013).

coordinate space, in which the positions, orientations and scales of all objects in game world are expressed."[38] Im Gegensatz zum *world space*, bezeichnet man die Ausgabe der Grafik als *screen space*.

Die Grundlage der Karte bildet ein linkshändiges kartesisches Koordinatensystem (vgl. Abb. 17). Y beschreibt die Tiefe, während X die Breite bezeichnet (vgl. Abb. 18).

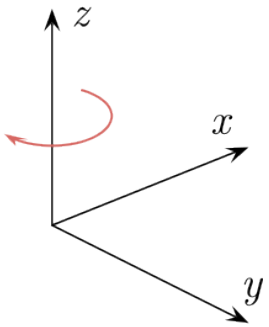


Abb. 17: Ein linkshändisches Koordinatensystem.

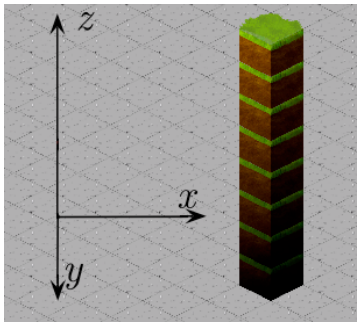


Abb. 18: Die Einteilung der Koordinaten der Map.

Diese Blöcke werden im Feld *data* gespeichert. Dies ist ein dreidimensionales Array für die drei Dimensionen.

4.5.1. Entitys

Die Benutzung von Arrays erlaubt nur einen Block je Rasterfeld. Entitys werden nicht im Raster (also im Feld data der Klasse Map*) gespeichert* sondern *befinden* sich durch ihre Koordinaten nur dort, d.h. werden dort nur gerendert. Die Entitys werden getrennt in einer Liste gespeichert (Feld *entitylist*). Diese Architektur ermöglicht, dass sich beliebig viele Entitys im gleichen Koordinatenfeld befinden können. Dies ist wichtig, da die Entitys nicht nicht die Form der Umwelt modellieren.

4.5.2. Streamen der Karte mit Chunks

Um Ressourcen zu schonen, enthält die Map nicht die ganze Karte, sondern nur den in der Nähe zum sichtbaren Bereich befindlichen Teil. Um eine flüssige Erweiterung der Karte zu ermöglichen, werden Teile der Karte nach dem Übertreten einer Grenze nachgeladen und die übrigen Teile verschoben. Mit diesem Verfahren ist es möglich, riesige Karten zu benutzen, ohne in Performance-Schwierigkeiten zu geraten. Dieses Verfahren wird im Folgenden als „Chunkwechsel" bezeichnet. Ein Kartenausschnitt wird als

[38] Gregory: Game Engine Architecture S. 161

„Chunk“ bezeichnet. Die Map benutzt die gleichnamige Klasse (Anh. 3) zum Verwaltung ihrer Daten.

Da statt einem einmaligen Laden einer Karte bei Kamerabewegung regelmäßig ein Kartenabschnitt nachgeladen wird, kann man von einem Streaming der Karte sprechen. Um das Streamen zu ermöglichen, wird ein Kartenausschnitt aus neun nebeneinander liegenden Chunks gebildet:

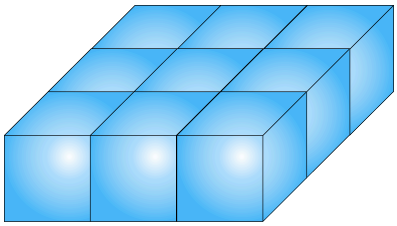


Abb. 19: Neun Chunks bilden den Inhalt der Map.

Die Map enthält selber keine Chunk-Objekte, vielmehr werden die Blöcke aus den Chunks in das bereits erwähnte Feld *data* der Map gefüllt.[39]

Die Daten einer Chunk-Datei werden aufgrund des eigenen Speicherformates und der geringen Größe der Daten nicht mithilfe eines Datei-Streams geladen.

Die Dimensionsgrößen der Chunks und damit der Karte können im Programmcode beliebig groß festgelegt werden und werden beim Laden einer Karte aus den Metainformationen der Speicherdatei ausgelesen werden. So kann z.B. eine Umsetzung vom Spiel „Schach“ mit einer Höhe von nur zwei Blöcken auskommen, im Gegensatz zu einem Spiel, was fünf mal so hohe Chunks und dadurch fünfmal so viele Schleifendurchläufe für viele Methoden benötigt.

4.5.3. Formender Blockanordnung

Es gibt verschiedene Anordnungen der Koordinatensysteme bei rasterförmigen Karten:

Die „Diamantenform“ ist bei isometrischen Spielen sehr weit verbreitet. Dabei verlaufen die Y- und die X-Achse diagonal zum Bild. Die gängige Bezeichnung des Diamanten hat sie bekommen, weil, wie in Abb. 18 dargestellt, bei ihrer Verwendung die Form einer Raute entsteht.

[39] Dazu wird die Methode setChunk(int pos, Chunk newchunk) der Klasse Map benutzt.

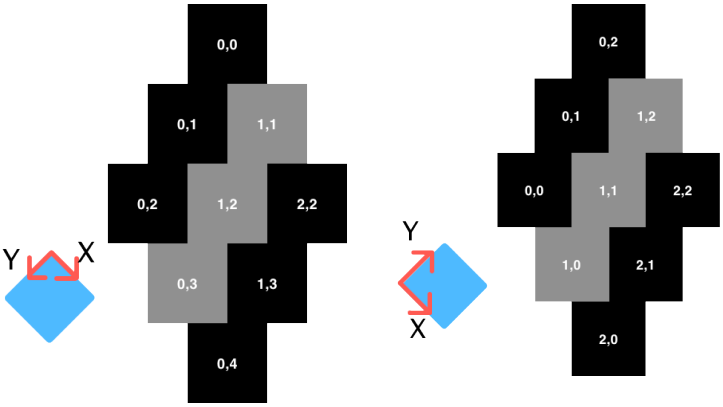


Abb. 20: Zwei mögliche Aufbauvarianten von Karten in der Diamantenform

Die X- und Y-Achse verlaufen bei der Diamantenform (im *game space*), wie bei einem Schachbrett, auf zueinander orthogonalen Geraden, was der Vorteil dieses Formates ist. Die Diamantenform ist quasi ein um 45° gedrehtes und anschließend gestauchtes Koordinatengitter.

Bei den „staggered maps“ („staggered“ - engl. gestaffelt) wird jede zweite Zeile nach rechts oder nach links um die halbe Breite eines Blocks verschoben (vgl. Abb. 19). Die Wurfelengine benutzt dieses Kartenformat. Hierbei ist die Verschiebung positiv, d.h. alle ungeraden Y-Reihen sind um eine halbe Blockbreite nach rechts verschoben. Mit der Rechenoperation y modulo 2 [40] lässt sich feststellen, ob eine Reihe gerade oder ungerade ist.

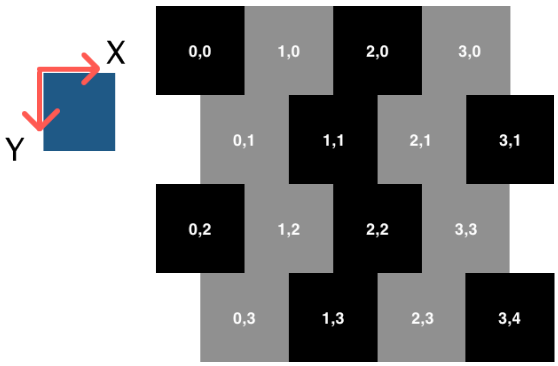


Abb. 21: Der Aufbau einer staggered map.

Die Verschiebung und die dadurch erreichte Unordnung ist im Vergleich zur Diamantenform ein Nachteil. Ein Vorteil bei den *staggered maps* ergibt sich aus dem unkomplizierten Aneinanderfügen mehrerer nebeneinanderliegender Chunks.

[40] In Java: `y % 2`.

4.5.4. Verschiedene Anordnungen der Chunks

Mehrere Chunks im *staggered-map*-Format nebeneinander bilden ein Rechteck parallel zum Bildschirm, was den Chunkwechsel beim Scrollen vereinfacht. Neun Chunks können zu einem großen, weiterhin zum Bildschirm parallelen Kartenausschnitt zusammengebunden werden. In der Wurfelengine werden dazu die Positionen von 0 bis 8 wie folgt nummeriert:

0	1	2
3	4	5
6	7	8

Abb. 22: Die Nummerierung der Chunks beim Gebrauch der staggered maps.

Das Streamen der Karte ist mit den Chunks in der Diamantenform aufwendiger. Wie in Abb. 21 sichtbar, haben die Grafikausgabe und die Chunks ein anderes Format, wodurch die Grenzen des Kartenausschnittes öfters übertreten werden, was jedes Mal ein Chunkwechsel zur Folge hat.

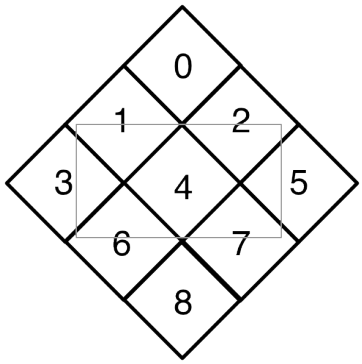


Abb. 23: Ein Kartenausschnitt mit Chunks in Diamantenform.
Das graue Rechteck steht für einen Bildschirm im gängigen 16:9-Format.

4.5.5. Der Chunkwechsel

Nachdem das Verfahren bereits einige Male erwähnt wurde, wird im Folgenden die Umsetzung noch etwas genauer erläutert.

Beim Chunkwechsel wird eine Reihe Chunks aus dem Speicher entfernt, zwei Reihen rücken nach, und eine neue Reihe wird geladen. Dies geschieht in der Methode `setCenter(int newmiddle)` der Klassse *Map*, wenn die Grenzen des Viewports die Gren-

zen des geladenen Kartenstücks fast erreicht haben. Der Parameter `newmiddle` gibt die Nummer der neuen Mitte an. Die Nummerierung der Chunks ist die Gleiche wie in Abb. 22.

Als Erstes muss eine Kopie der Map-Daten erstellt werden, denn die weiterhin benötigten Daten würden sonst durch die Verschiebung überschrieben werden. Die Kopie wird mit der Methode `copyOf3Dim(Block[][][] array)` erzeugt, welche eine tiefe[41] Kopie erstellt:[42]

```
private Block[][][] copyOf3Dim(Block[][][] array) {
        Block[][][] copy;
        copy = new Block[array.length][][];
        for (int i = 0; i < array.length; i++) {
            copy[i] = new Block[array[i].length][];
            for (int j = 0; j < array[i].length; j++) {
                copy[i][j] = new Block[array[i][j].length];
                System.arraycopy(array[i][j], 0, copy[i][j], 0, array[i][j].length);
            }
        }
        return copy;
    }
```

Der Quelltext für diese Methode stammt von Kevin Brock.[43]

Anschließend werden in einer for-Schleife die Koordinaten der Chunks in einer Liste aktualisiert:

```
coordlist[pos][0] += (newmiddle == 3 ? -1 : (newmiddle == 5 ? 1 : 0));
coordlist[pos][1] += (newmiddle == 1 ? -1 : (newmiddle == 7 ? 1 : 0));
```

Bei der Verschiebung eines Chunks gibt es zwei Fälle:

- Der Chunk muss neu generiert oder geladen werden.
- Er kann verschoben werden.

---

[41] Eine tiefe Kopie kopiert in untergeordneten Strukturen alle Objekte, wohingegen eine flache Kopie nur das Gesamtobjekt kopiert, wodurch die Referenzen in den tieferliegenden Ebenen auf dieselben Objekte verweisen.

[42] Die in Java mitgelieferten Methoden erzeugen nur flache Kopien von mehrdimensionalen Arrays.

[43] http://stackoverflow.com/a/2068649/1291117 (letzter Zugriff 22.03.2013)

Ob die Verschiebung möglich ist, prüft die Funktion `isMovingChunkPossible(int pos, int newmiddle)`, welche die Positionsnummer des Chunks und die Nummer des neuen Mittelpunktes der Map benötigt. Hier sind in einer switch-Anweisung alle Fälle, die keine Verschiebung erlauben, aufgelistet. Wenn der Chunk verschoben werden kann, wird aus der Kopie der Chunk mit der Funktion „herauskopiert“ und an der neuen Position eingefügt:

```
setChunk(pos, getChunk(data_copy, pos - 4 + newmiddle));
```

Die geht relativ einfach, da sich die neue Position einfach mathematisch beschreiben lässt:

$$\text{neue Position} = \text{pos} - 4 + \text{newmiddle}$$

Ist die Verschiebung nicht möglich (`isMovingChunkPossible(pos, newmiddle) == false`), wird ein neuer Chunk eingefügt:

```
setChunk(
	pos,
	new Chunk(
		coordlist[pos][0],
		coordlist[pos][1],
		MainMenuState.shouldLoadMap()
		)
	);
```

4.5.6. Das Speicherformat

Damit Karten nicht bei jeder Programmbenutzungzufällig generiert werden müssen, sondern vorher entworfen werden können, benötigt man ein Speicherformat. Eine Karte besteht aus einem Ordner mit einer Meta-Datei und den Chunk-Dateien. Damit die Dateien immer gefunden werden können, müssen diese einheitlich benannt sein. Die Me-

ta-Datei muss immer „map“ heißen. In ihr sind Informationen wie die Version des Kartenformates, der Kartenname, die Spielerposition u.Ä. enthalten. Die Chunk-Dateien haben als Dateinamen die Benennung „chunk“, gefolgt von der X- und Y-Koordinate und getrennt durch ein Komma mit je nach Spiel frei wählbarem Suffix z.B. „chunk5,-2.wec“.

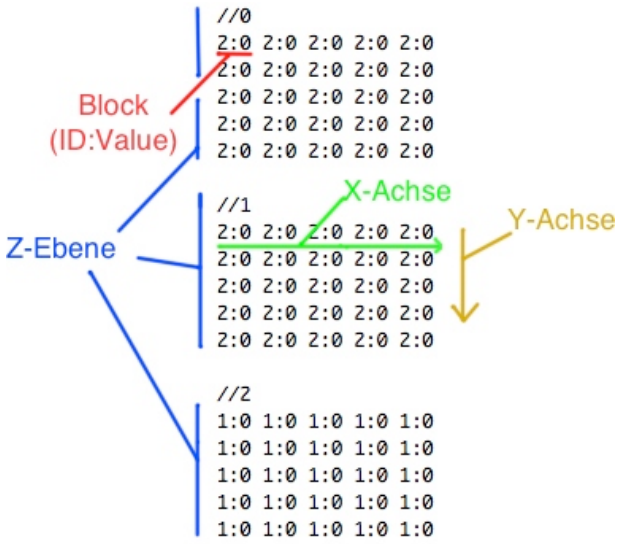


Abb. 24: Erklärung des Speicherformates anhand eines Auszuges Die Kommentarzeilen, eingeläutet durch zwei Schrägstriche, sind optional.

Chunk-Dateien sind Textdateien und können so auch in Text-Editoren erstellt und bearbeitet werden. Eine Erklärung des Aufbaus des Speicherformates findet sich in Abb. 24.

4.6. Spielfiguren

Spielfiguren können mithilfe der Klasse `AbstractCharacter` erzeugt werden. Der Controller kann eine steuerbare Spielfigur haben, die auf die Eingabe des Benutzers reagiert. Diese wird mit `setPlayer(AbstractCharacter player)` gesetzt.

4.6.1. Die Bewegung

Für eine möglichst freie Bewegung der Spielfigur erhält der Charakter einen zweidimensionalen Einheitsvektor[44] für die Richtung (`dir`) sowie einen Faktor (`speed`) für seine Geschwindigkeit. Um bei Bewegung die neue Position zu ermitteln, wird zu der bisherigen Position das Produkt aus Δt (`delta`)[45] und Geschwindigkeit addiert. Die Multiplikation mit Δt dient dazu, die Geschwindigkeit bei unterschiedlichen Ausführungsgeschwindigkeiten konstant zu halten. Es gilt daher allgemein:

[44] Die Z-Komponente ist unabhängig von der horizontalen Bewegung, d.h. der Einheitsvektor beschränkt sich auf X- und Y-Komponenten.

[45] Delta t ist die Zeit seit dem letzten Aufruf und wird von Slick in der *render*- und *update*-Methode des GameLoops übergeben.

$$\overrightarrow{\text{pos_neu}} = \overrightarrow{\text{pos_alt}} + \overrightarrow{\text{dir}} * \text{speed} * \Delta t$$

Die Bewegung der Spielfigur lässt sich mit der Methode `walk(boolean up, boolean down, boolean left, boolean right, float walkingspeed)` setzen und wird in der update-Methode durchgeführt.

4.6.2. Der Koordinatenwechsel

Wenn ein `AbstractCharacter` die Grenzen seines bisherigen Felds übertreten hat, d.h. der Mittelpunkt der Figur liegt außerhalb der Begrenzung des aktuellen Rasterfeldes, soll die Koordinate des Charakters entsprechend gewechselt werden.

Feststellen lässt sich dies mit mit der Methode `public static int getSideID(float x, float y)` der Klasse `GameObject`. Diese Methode gibt die Nummer der Nachbarseite an, auf der sich die Parameter-Koordinaten (x und y) befinden. Die Nummerierung der Seiten ist wie folgt:

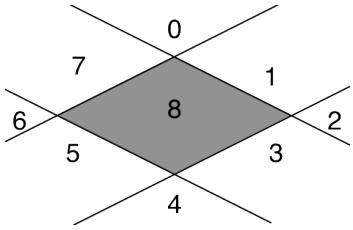


Abb. 25: Die Seitennummerierung

Genauso wie für den Bewegungsalgorithmus wird die Methode für die Umwandlung der Screen-Space- zu Game-Space-Koordinaten benötigt (z.B. für eine Auswahl eines Blocks durch die Maus). Die Ermittlung der Zahl erfolgt mithilfe von vier Ungleichungen.. Häufig wird ein simplerer, aber weniger flexiblerer Ansatz, genutzt, bei dem man eine Grafik (Abb. 26) nimmt, auf der die an der Stelle befindliche Farbe Auskunft über die dort befindliche Seite gibt. [46,47]

Abb. 26: Die von vielen Spielen für die Identifikation der Seiten benutzte Grafik.

---

[46] Kapitel „Mouse Matters“ http://www.gamedev.net/page/resources/_/technical/game-programming/isometric-n-hexagonal-maps-part-i-r747 (letzter Zugriff 22.03.2013).

[47] Steinke, Lennart: Spieleprogrammierung, 3. Aufl. Heidelberg 2008, S. 636.

Je nach Rückgabewert der Funktion wird dann die nötige Veränderung der Koordinaten mit Hilfe von `makeCoordinateStep(int x, int y)` ermittelt.

Die neue Position im neuen Rasterfeldes bildet die an der X- und Y-Achse um den Mittelpunkt des Blocks gespiegelte alte Position.

Bei der vertikalen Bewegung kann anhand der Höhe von der Grundebene (`height`) des Entitys die Koordinate ermittelt werden, womit der Koordinatenwechsel entfällt.

4.6.3. Kollisionsprüfung

Für die Kollisionsprüfung wird vor der Bewegung die neue Position ermittelt:

```
float newx = getPos()[0] + delta * speed * dir[0];
float newy = getPos()[1] + delta * speed * dir[1];
```

Danach wird überprüft, ob die Eckpunkte die neue Position zulassen, also nicht in einem Nachbarfeld mit einem Block mit `isObstacle()==true` stehen würden:

```
if (neighbourNumber != 8 && Controller.getNeighbourBlock(getRelCoords(),
neighbourNumber).isObstacle())
	validmovement = false;
```

Wenn die Bewegung zulässig ist, wird sie ausgeführt.

4.6.4. Sounds

Ein Charakter kann Geräusche abspielen, wenn er läuft oder fällt. Die Sound-Dateien stammen aus Sound-Datenbanken und stehen unter Creative-Commons-Lizenz. Die Angabe der Quellen findet sich in der Datei „Sources.txt“ im Sound-Ordner.

5. RENDERING

Beim Rendering werden die Game-Space-Koordinaten in Screen-Space-Koordinaten umgewandelt. Dabei wird die Achsenverkürzung auf die Daten angewandt.

Durch die Klasse `Gameplay` wird die `render`-Methode des *Views* aufgerufen. Diese ruft wiederum die Rendermethode der Kamera mit `camera.render();` auf. Dort wird der Prozeduraufruf an die Map und die einzelnen Objekte weitergeleitet.

Über das Bild der Spielszene werden nun HUD[48]- und GUI-Elemente gezeichnet. Hier wird mit `Gameplay.MSGSYSTEM.draw();` nur das Nachrichtensystem gerendert,weitere HUD- und GUI-Elemente einzubinden, wäre jedoch möglich.

5.1. Besonderheiten beim Rendering von 2D-Grafik

Spiele in 2,5D werden mit den gleichen Methoden gerendert wie 2D-Welten. Dazu werden sogenannte Sprite-Grafiken verwendet, das sind einzelne vorher erstellte Grafiken, die an einer bestimmten Stelle auf dem Monitor gezeichnet werden. Werden die Grafiken in separaten Dateien gespeichert, muss jede Datei als neue Textur auf den Grafikkontext mithilfe von OpenGL übertragen werden (genannt *bind*). Dabei werden Matrixtransformationen vorgenommen, die sehr rechenaufwendig sind.

Zur Beschleunigung nutzt man Spritesheets, das sind Bilddateien, bei der alle Sprite-Grafiken nebeneinanderliegen. Abb. 27 zeigt eine verkleinerte Version von der in der Wurfelengine benutzten Spritegrafik. Beim Rendering wird nur der benötigte Ausschnitt gerendert, also insgesamt nur ein *bind* für das Spritesheet benötigt statt für jedes Renderingde Sprite.

[48] Abk. Heads-up-Display, ein System, bei dem Informationen ins Sichtfeld projiziert werden.

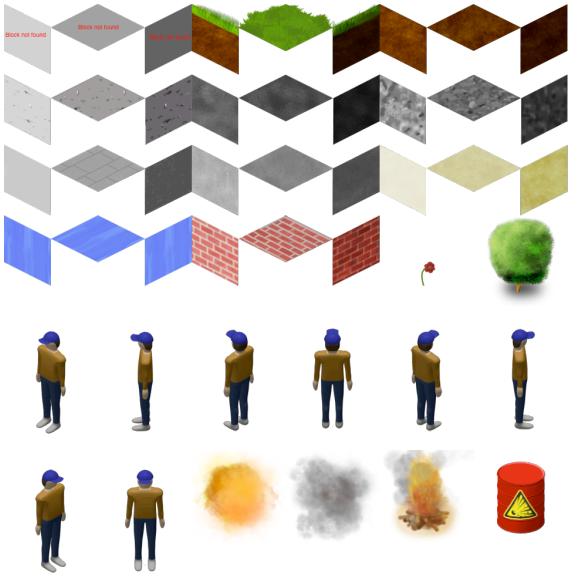


Abb. 27: Ein Spritesheet.

In der Wurfelengine wird der *bind* der Grafik mit `Block.getBlocksheet().startUse();` durchgeführt. Anschließend können nur Grafiken von dieser Textur gerendert werden, da diese an den Grafikkontext bis zum Lösen gebunden sind. Das Lösen erfolgt mit der Prozedur `Block.getBlocksheet().endUse();`, was den Grafikkontext wieder freigibt.

5.2. Der einfache Rendering-Algorithmus

Das Rendering bei isometrischen Ansichten lässt sich einfach mit dem Maler-Algorithmus durchführen. Beim Maler-Algorithmus werden Objekte in der Reihenfolge ihrer Entfernung, beginnend mit den entferntesten Objekten gerendert.

Beim einfachen Rendering-Algorithmus wird jeder Block als ganzer Block an seiner entsprechenden Position gerendert. Dies erfolgt durch drei verschachteltenfor-Schleifen, die von hinten nach vorne ablaufen, x, y und z müssen also von niedrigen zu großen Werten verlaufen. Das Rendering der gesamten Map würde so aussehen:

```
for (int z=0; z < Map.getBlocksZ(); z++)
        for (int y = 0; y < Map.getBlocksY(); y++)
                for (int x = 0; x < Map.getBlocksX(); x++)
                        data[x][y][z].render(new int[]{x,y,z});
```

Das Rendering und die Ermittlung der Position eines Objektes erfolgen dann im `GameObject`-Objekt mit der Methode `render(int[] coords)`:

```
public void render(int[] coords) {
    [...]
    int xpos = getScreenPosX(this, coords);
    int ypos = getScreenPosY(this, coords) - (dimensionY-1)*DIM2;
    image.drawEmbedded(xpos, ypos);
}
```

mit

```
public static int getScreenPosX(GameObject object, int[] coords) {
     return coords[0] * DIMENSION //x-coordinate multiplied by it's dimension in this
direction
            + (coords[1] % 2) * DIM2 //y-coordinate multiplied by it's dimension in this
direction
            + (int) (object.pos[0]); //add the objects position inside this coordinate
}
```

und

```
public static int getScreenPosY(GameObject object, int[] coords) {
    return coords[1] * DIM4 //x-coordinate * the tile's size
           - coords[2] * DIM2  //put higher blocks higher
           + (int) (object.pos[1] / 2)  //add the objects position inside this coordinate
           - (int) (object.pos[2] / Math.sqrt(2));  //take axis shortening into account
}
```

Wie die Codezitate zeigen, lässt sich dies sehr einfach durchführen. Nach eigener Einschätzung setzen fast alle isometrischen Spiele mit einem Raster diese Technik ein, teilweise in modifizierter Weise.

Sollen sich Objekte auch außerhalb des Rasters bewegen können oder andere Dimensionen haben, reicht der Algorithmus aber nicht mehr aus. In der Wurfelengine ist der Algorithmus deshalb noch um zwei Punkte erweitert:

- Nur sichtbare Blöcke werden gerendert:
  - Die Grenzen der Schleifen entstammen den Grenzen des Viewports.
  - Ein Analyse-Algorithmus filtert nicht sichtbare Blöcke.
- Blöcke, die nicht fest im Raster stehen, können je nach ihrer Position vor oder hinter benachbarten Blöcken stehen. Das Rendering der Blöcke, die deshalb vorher gezeichnet werden müssen, wird vorgezogen.

5.3. Das Rendering von zerteilten Blöcken

Die Wurfelengine bietet ein weiteres Verfahren für das Rendering eines Blocks an. Wie zuvor erklärt, wird beim einfachen Algorithmus jeder Block als vollständige Grafik gezeichnet. Der Vorteil davon ist, dass ein Würfel als Ganzer vordefiniert ist und 1:1 gerendert wird, so wie er gespeichert wurde. Man kann den einfachen Algorithmus erweitern, so dass sich ein Block aus drei Seiten zusammensetzt (vgl. Abb. 28).

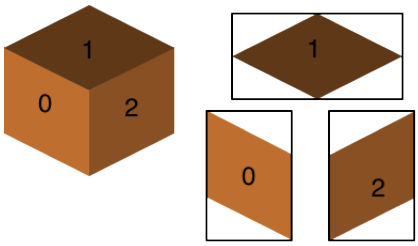


Abb. 28: Ein Block, aufgeteilt in drei Sprite-Grafiken.

Das Verfahren hat einige Vor- und Nachteile:

Der größte Vorteil ergibt sich aus den Optimierungsmöglichkeiten, denn verdeckte Seiten müssen nicht gerendert werden (Erklärung des Sichtbarkeitsbestimmungs-Algorithmus in Kapitel 5.4.). Das Auslassen der nicht sichtbaren Seiten führt, je nach Situation, zu einem Geschwindigkeitsbonus von bis zu 40 %.

Ein weiterer Vorteil ist, dass mehrere Blöcke auf dieselbe Ressource zurückgreifen können, d.h. Grafiken müssen bei nur leichten Variationen nicht doppelt gespeichert werden.

Ein Beispiel: Eine Kiste soll einmal geöffnet und einmal geschlossen darstellbar sein. Benötigt werden jetzt insgesamt vier Seiten, da zwei Seiten gleich bleiben. Dies ist in Abb. 29 dargestellt.

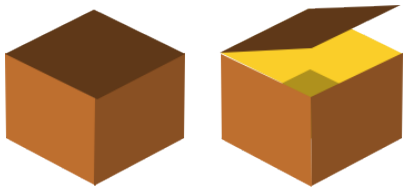


Abb. 29: Zwei Blöcke mit insgesamt vier verschiedenen Seiten.

Für einen einzelnen Block wird, wegen der transparenten Bereiche der Ausschnitte und notwendigen Abständen der 1,5 mal so viel Speicherplatz wie beim Rendering der gesamten Blockgrafik benötigt.

Abb. 30: Die Grafik eines Menschen mit Trennlinien.

Für manche Blöcke ist dieses Verfahren nicht geeignet. Dies sind Blöcke, die keine drei Kanten haben, wie z.B. runde Objekte, Büsche, Menschen usw. Die Grafiken müssen zertrennt werden, um beim Rendering wieder aneinander gefügt zu werden. Dies kann zu „Nähten", wie Abb. 30 zeigt, führen oder zu einer komplizierten Erstellung der Grafiken, wenn die Übergänge stehengelassen werden, um von der Engine doppelt gerendert zu werden.

Werden viele verschiedene komplexe Grafiken benötigt (z.B. für einen Wald), ist das Verfahren für die Entwicklung mühseliger und in der Laufzeit langsamer als der einfache Algorithmus.

Aus diesen Gründen kann das Verfahren daher individuell für jeden Block eingestellt werden.

5.4. Sichtbarkeitsbestimmung mittels Raytracing-ähnlichem Verfahren

Die Karte wird, bevor sie gerendert wird, mithilfe eines vereinfachten Raytracing-Algorithmus[49] abgetastet. In der Analyse wird die Sichtbarkeit der Seiten der Blöcke berechnet, was die Anzahl der zu Renderingden Grafiken minimiert (Hidden Surface Removal).[50] Dazu sendet man für die drei sichtbaren Seiten eines Blocks jeweils einen „Strahl" (engl. „Rays") von der Oberfläche der Karte in Blickrichtung aus, dargestellt in Abb. 31.

[49] Raytracing bezeichnet ein Rendering-Verfahren, bei dem Strahlen, ähnlich den Lichtstrahlen, für jeden Pixel von der Kamera aus in dem Raum gesandt werden und aus den geschnittenen Flächen und den Schnittwinkeln das Bild errechnet wird.

[50] Vgl. http://conitec.net/manual_d/culling.htm (letzter Zugriff 22.03.2013)

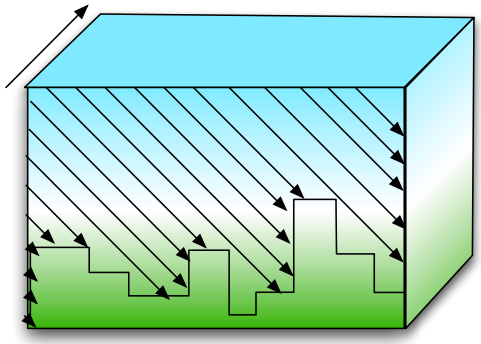


Abb. 31: Raytracing der Map.

Der Strahl setzt die Sichtbarkeit der Seiten, die er passiert, auf `true`, bis er auf Hindernisse stößt, die den Strahl nicht weiter durch lassen. Ist mindestens eine Seite sichtbar, wird die Sichtbarkeit des gesamten Blocks auf `true` gesetzt. Der Quelltext ist komplex (ca. 200 Programmzeilen), weil jede Seite individuell behandelt werden muss. Für Flüssigkeiten müssen zudem weitere Prüfungen erfolgen, damit nur die Oberfläche in einem Gewässer sichtbar ist.

5.5. Tiefensortierung

Falsche Reihenfolgen beim Maleralgorithmus, die entstehen, wenn Objekte größere Raster-Dimensionen besitzen oder sich nicht fest im Raster befinden, kann man beim einfachen Rendering-Algorithmus nur sehr schwer umgehen. Stattdessen kann man eine Tiefensortierung durchführen.

Dies geschieht in drei Schritten:

1. Die Koordinaten und die zugehörige Tiefe von Blöcken, die gerendert werden sollen, werden in einem Objekt gespeichert und einer Liste hinzugefügt.
2. Entities werden ebenfalls, zusätzlich mit ihrer Indexnummer, der Liste hinzugefügt.
3. Die Liste wird mittels eines Sortieralgorithmus, hier Quicksort, sortiert.

Die Tiefe eines Blocks lässt sich anhand seiner Daten wie folgt berechnen:

```
public int getDepth(int y, int z) {
    return (int) (
            DIMENSION * y
            + (y % 2) * DIM2
            + (DIM2-1) * z
```

```
            + pos[1]
            + pos[2]
            + (dimensionY - 1) * DIM4
    );
}
```

Jetzt kann man die Liste in ihrer Reihenfolge Rendering und Sonderfälle beim Rendering einfacher berücksichtigen, denn man muss nur die Tiefe eines Blocks ändern, um die Reihenfolge des Renderings zu beeinflussen.

5.6. Der erweiterte Rendering-Algorithmus

Der erweiterte Rendering-Algorithmus vereint uneingeschränkt Tiefensortierung, Raytracing und die Option Blöcke als Seitentripel zu definieren. Das Rendering geschieht in folgender Reihenfolge:

1. Raytracing (nur bei größeren Veränderungen der Daten nötig)
2. Tiefensortierung
3. Rendering der Liste:
    a. als ganzer Block
    b. oder als Seitentripel

5.7. Skalierung

Beim Rendering müssen unterschiedliche Auflösungen des Ausgabe berücksichtig werden. Um unterschiedliche Auflösungsgrößen auszugleichen, wird der Grafikkontext vor dem Rendering skalliert. `equalizationScale` gibt einen Faktor an, durch den die Breite der Ausgabe immer die Gleiche ist. Unterschiedliche Höhen-/Breitenverhältnisse jedoch lassen sich jedoch so nicht ausgleichen. Zusätzlich zum Faktor `equalizationScale` gibt es bei den Kameras die Möglichkeit zu Zoomen (`zoom`). Beide werden mit der Methode `public float getTotalScale()` zusammengefasst und mit diesem gemeinsamen Faktor wird der Grafikkontext skaliert[51]:

```
Wurfelengine.getGraphics().scale(getTotalScale(), getTotalScale());
```

[51] Methode void scale(float sx, float sy) von org.newdawn.slick.Graphics.

Die ermittelten Screen-Space-Koordinaten stimmen nun nicht mehr mit der tatsächlichen Darstellung überrein[52]; falls die genaue Positionen auf dem Bildschirm benötigt werden sollten, können diese anhand des Skalierungsfaktors nachträglich bestimmt werden.

[52] Ähnliches passiert auch, wenn man z.B. die Auflösung eines Monitors skaliert. Die Auflösung des Bildes beträgt dann z.B. 800 x 600 px und die des Monitors 1600 x 900 px. Es werden also, obwohl die Software 800 px in der Breite nutzt, 1600 px dargestellt.

6. MÖGLICHE ERWEITERUNGEN

Eine Engine ist eine Sammlung von Algorithmen und hat damit keine Grenzen und keinen Punkt, an dem sie objektiv vollendet ist. Es werden darum hier ein paar mögliche sinnvolle Erweiterungen und Verbesserungen vorgestellt.

6.1. Erweiterte Licht-Engine

Während der Entwicklung experimentierte ich lange mit der der Helligkeitsveränderung von einzelnen Seiten beim Rendering-Algorithmus. Ziel war es, die Bewegung einer Sonne durch das Auf- und Abdunkeln der Seiten zu simulieren. Die Sonne ist bei isometrischen Spielen selber nie sichtbar, so dass nur durch Schatten und Reflexion des Lichtes der Eindruck einer sich bewegenden Sonne erweckt werden kann. Der Zugriff auf die einzelnen Seiten ist nur mit dem zusammengesetzten Rendering der Blöcke möglich. Die praktische Umsetzung erwies sich als zu zeitintensiv und musste daher eingestellt werden.

Ich konnte kein isometrisches Spiel finden, was eine vergleichbare Technik ohne 3D-Engine nutzt. Eine beliebte Technik ist das Abdunkeln der gesamten Szene, um einen Tag-/Nacht-Zyklus zu erreichen oder Lichtpunkte in Gewässern. Die Berechnung des Lichts ist in der Wurfelengine sehr einfach gehalten. Für die X-Y-Ebene wird der jeweils höchste Block einer Reihe bestimmt, wo das Helligkeitsmaximum liegt, und von dort aus nimmt die Helligkeit bis zum Minimum, dem untersten Block, ab. Dies könnte man erweitern durch weitere Lichtquellen wie Lampen und Fackeln, was besonders in Kombination mit der seitenspezifischen Beleuchtung ein gutes Ergebnis erbrächte.

6.2. Splitscreen

Wenn mehrere Spieler an einem Gerät mit geteilten Bildschirmen spielen können, nennt man dies Splitscreen. Das ist mit der Wurfelengine prinzipiell möglich. Da aber die Daten auch von der Kamera abhängig sind (z.B. beim Chunkwechsel), ist der Einsatz einer zweiten beweglichen Kamera schwierig, denn wenn sich die beiden Kameras an zwei weit voneinander entfernten Orten befinden, müssen komplett unterschiedliche Daten in der Map vorhanden sein, d.h. für jede Kamera würde eine eigene Map benötigt, welche sich synchronisieren müssten.

6.3. Mehr Konfigurationsoptionen

Für einen breiten Einsatz einer Engine sind viele Optionen für jedes Feature vorteilhaft. Solche könnten z.B. die Berücksichtigung der Achsenverzerrung bei der Bewgungsgeschwidigkeit, eine konfigurierbare Anzahl von Spritegrafiken für die Rotation von Charakteren oder das Verhalten bei unterschiedlichen Auflösungen sein.

7. WERKZEUGE

Der *Map Editor* gehört nicht zur eigentlichen Arbeit, trotzdem wird er hier kurz vorgestellt.

7.1. Der Map Editor

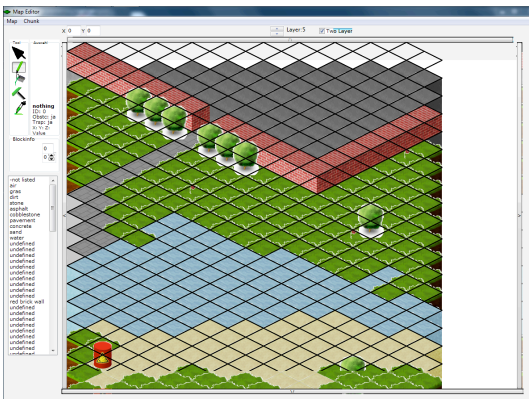


Abb. 32: Screenshot des Leveleditors

Da die Wurfelengine ein eigenes Speicherformat für Karten benutzt, ist es nützlich, ein passendes Werkzeug zur Erstellung von Karten zur Hand zu haben, um Karten nicht im Texteditor erstellen zu müssen, sondern mit einem GUI (Abb. 32). Der Map Editor wurde mithilfe von Delphi erstellt.

Da die Schnittstelle zur Engine nur die Speicherdateien sind, können Entwickler und Entwicklerinnen einen eigenen Editor programmieren und nach ihren Wünschen gestalten.

7.2. Sprite-Werkzeuge

Das Spritesheet ist von Hand sehr mühsam zusammenzusetzen. Als Hilfsmittel empfiehlt sich ein Hilfsprogramm wie das für die Entwicklung der Wurfelengine benutzte Programm „TexturePacker“[53].

Sprites können mit gewöhnlichen 2D-Grafikprogrammen erstellt werden, wie es bei den meisten beiliegenden Sprites getan wurde. Dabei können auch Automatisierungs-Skripte helfen.

Zur Erstellung von hochwertigen Spritegrafiken können 3D-Programme verwendet werden. Für die Grafik des Spieler wurde die freie Software „Blender“[54] benutzt.

[53] http://www.codeandweb.com/texturepacker (letzter Zugriff 22.03.2013).

[54] http://www.blender.org (letzter Zugriff 22.03.2013).

8. AUSWERTUNG

Es gibt eine speziell konfigurierte Version der Engine, mit der sich ihre Geschwindigkeit auf mehreren Computern bestimmen und miteinander vergleichen lässt.

In der folgenden Tabelle sind die Messergebnisse abgebildet:

	Durchschnittliche FPS in einer Minute ohne Skalierung bei 1024 x 768px	Bemerkung
Prozessor: 2,26 GHz Intel Core 2 Duo Speicher: 8 GB 1333 MHz DDR3 Grafikkarte: nVidia GeForce 9400 256 MB OS: Mac OS X 10.8.2 (12C60) Java Version: 1.6.0_43	377,8	
CPU: Intel Pentium M 1,7 GHz Speicher: 1,5 GB 594 MHz Grafikkarte: ATI Mobility Radeon 7500 32MB OS: Windows XP 5.1.2600 Java Version: 1.7.0_09	99,6	Eine Skalierung des Grafikkontextes senkte die FPS stark.
CPU: 1,8 GHz AMD Sempron 3200+ Speicher: 2 GB Grafik: nVidia GeForce 6150 LE OS: Windows XP Java Version: 1.7.0_10	159,2	
CPU: AMD FX8150 (8x 3,9 GHz) Speicher: 16 GB DDR3 Grafikkarte: nVidia GeForce GTX670 OS: Windows 7 Java Version: 1.6.0_20-b12 64Bit	498	Die Anzeige zeigte Werte um ~4400 an. Daher gab es wohl einen Anzeige- oder Messfehler.
CPU: Intel Centrino Duo T6600 (2 x 2,2 GHz) Speicher: 4GB DDR3 Grafikkarte: nVidia GeForce GT240M OS: Windows 8 Pro 64bit Java Version: 1.6.0_20-b12 64Bit	1150	

Die Ergebnisse befinden sich im üblichen Rahmen, gemessen an vergleichbaren Spielen aus der Spieleindustrie. Beim zweiten Test zeigte sich, dass unter normalen Bedingungen (native Auflösung und Skalierung an) nicht mehr als 60 FPS möglich waren, was aber vermutlich an der mangelhaft unterstützen Skalierungsfunktion der Grafikbibliothek auf der Grafikhardware liegt.

8.1. Der Einsatz für Spiele

Durch die isometrische Ansicht bedingt, sind hauptsächlich Spiele aus den Genres Simulation, Strategie, Rollenspiel und Abenteuer für die Engine geeignet.

Möchte man ein Spiel entwickeln, muss eine eigene Kindklasse der Controller-Klasse in der Klasse Gameplay gesetzt werden. Dort befindet sich der Startpunkt für eigene Erweiterungen. Bei der Wurfelengine ist standardmäßig die Klasse `CustomGameController`, in der die Steuerungen der Demoprogramme definiert sind, als Controller gesetzt.

8.2. Erreichte Zielsetzung

Die Wurfelengine besitzt alle Grundlegenden Funktionen, um Spiel auf ihr aufbauend zu erstellen. Die Klassenarchitektur erlaubt, wie gezeigt, viele eigene unkomplizierte Implementationen der Blöcke und Entitys. Das in der Einleitung gesetzte Ziel wurde damit erreicht.

LITERATURVERZEICHNIS

Gregory, Jason: Game Engine Architecture, 2. Aufl. Massachusetts 2009.

McShaffry, Mike / Graham, David „Rez“: Game Coding Complete, 7. Aufl. Boston 2012.

Flint, John P. [with Salem, Omar]: Software Engineering for Game Developers, Boston 2005.

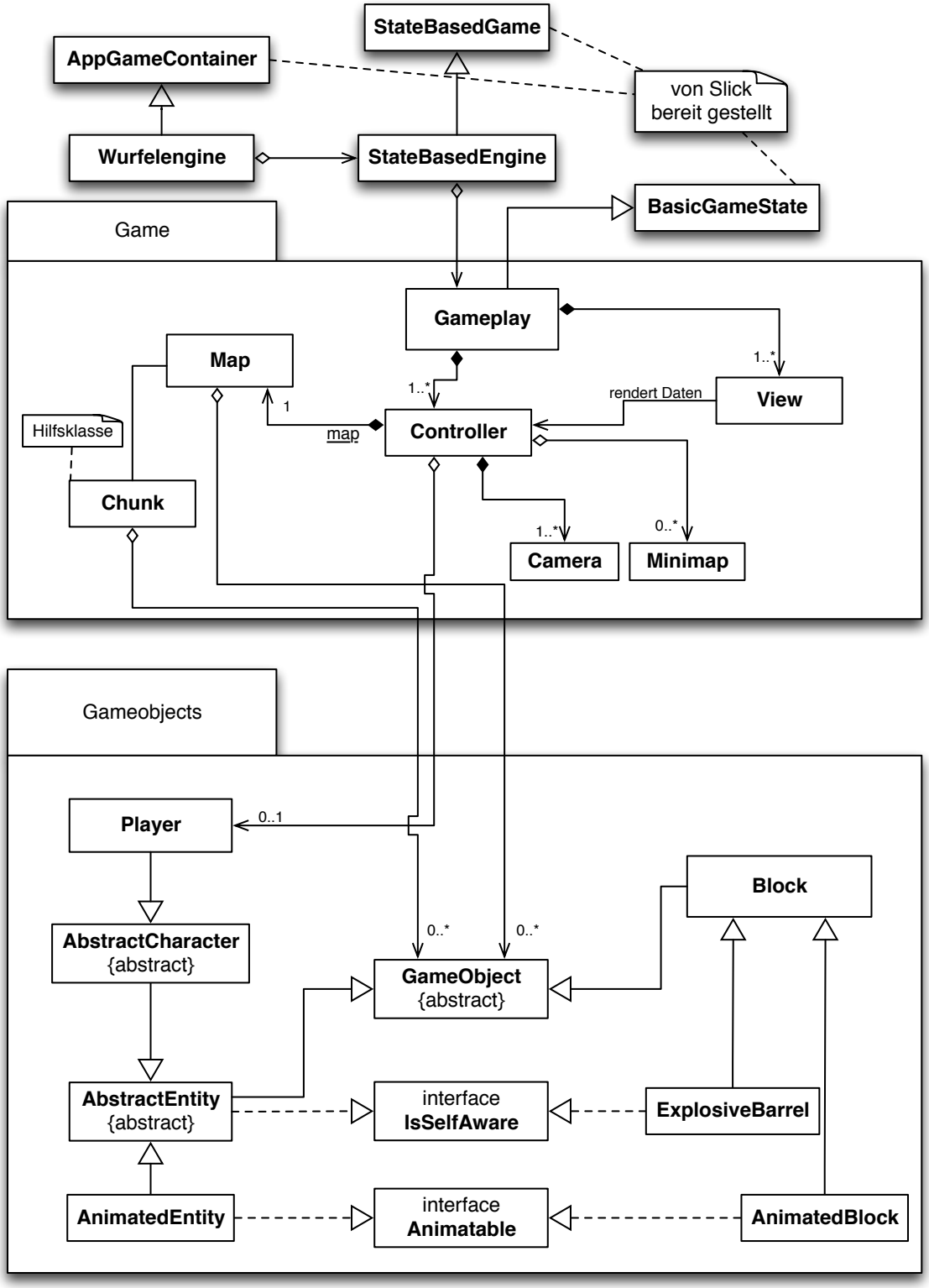
Steinke, Lennart: Spieleprogrammierung, 3. Aufl. Heidelberg 2008.

ABBILDUNGSÜBERSICHT

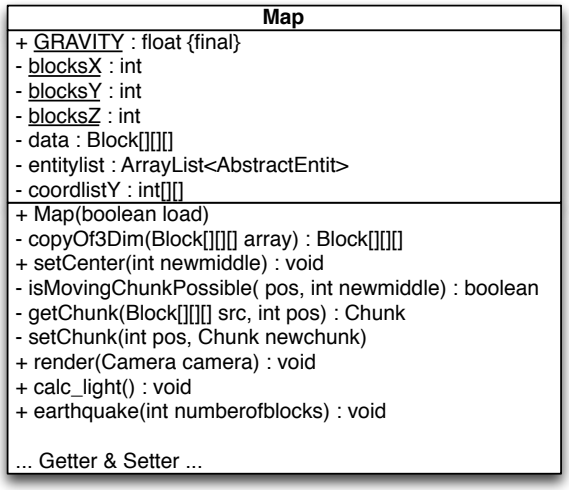
1. Screenshot des Rasters.
2. Spielcharakter im Spiel „3D Dot Game Heroes“ und „Minecraft“. http://itsjuststuffandnonsense.blogspot.de/2010/05/3d-dot-heroes-little-game-that-could.html
3. Eine Pixel-Art Grafik in Minecraft. http://www.wikihow.com/Image:2012-07-10_13.01.43.png
4. Screenshot aus Super Mario Bros. http://galaxyyarena.blogspot.de/2012/06/super-mario-bros.html
5. Screenshot vom Spiel Sacred (2,5D). http://onegaigames.blogspot.de/2010/12/sacred-review.html
6. Screenshot vom Spiel Nexuiz (3D). http://commons.wikimedia.org/wiki/File:Nexuiz_screenshot_02.jpg
7. Der Viewport. Game Engine Architecture S. 184 Abb. 4.29
8. Das isometrische Koordinatensystem.
9. Eine pixelige Linie im Winkel von 30°.
10. Eine pixelige Linie im Winkel von 26,56°.
11. Ein isometrischer Block.
12. Die hinteren Blöcke werden bei der Isometrie verdeckt.
13. Ein üblicher simpler Game Loop. Game Coding Complete S. 176 Abb. 7.2
14. Die Beziehung zwischen View und Controller.
15. Ein Block mit zwei Grafiken.
16. Das Klassendiagramm des Package *Gameobjects*.
17. Ein linkshändisches Koordinatensystem. Urheber: gustavb, cc-by-sa
18. Die Einteilung der Koordinaten der Map.
19. Neun Chunks bilden den Inhalt der Map.
20. Zwei mögliche Aufbauvarianten von Karten in der Diamantenform.
21. Der Aufbau einer *staggered map*.
22. Die Nummerierung der Chunks beim Gebrauch der *staggered maps*.
23. Ein Kartenausschnitt mit Chunks in Diamantenform.
24. Erklärung des Speicherformates anhand eines Auszuges.

25. Die Seitennummerierung.
26. Die von vielen Spielen für die Identifikation der Seiten benutzte Grafik. http://www.gamedev.net/page/resources/_/technical/game-programming/isometric-n-hexagonal-maps-part-i-r747
27. Ein Spritesheet.
28. Ein Block, aufgeteilt in drei Sprite-Grafiken.
29. Zwei Blöcke mit insgesamt 4 verschiedenen Seiten.
30. Die Grafik eines Spielers mit Trennlinien.
31. Raytracing der Map.
32. Screenshot des Leveleditors.

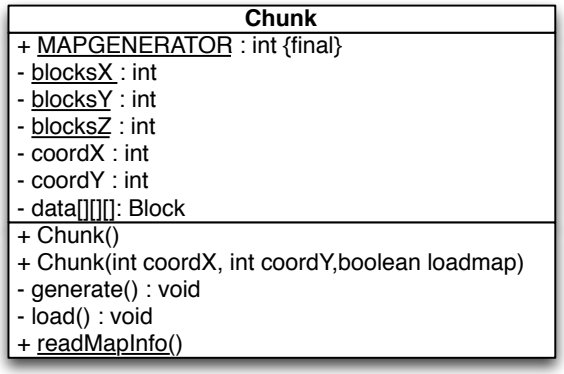
ANHANG



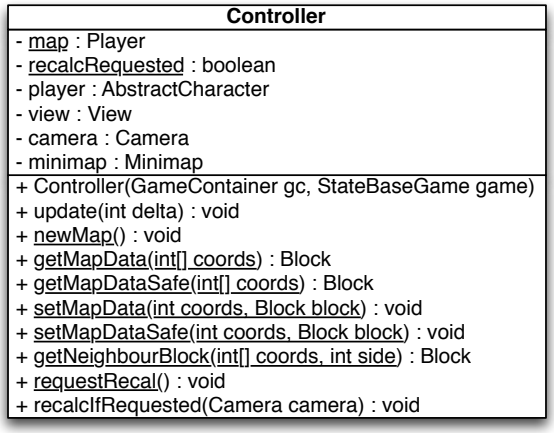
1: Klassendiagramm



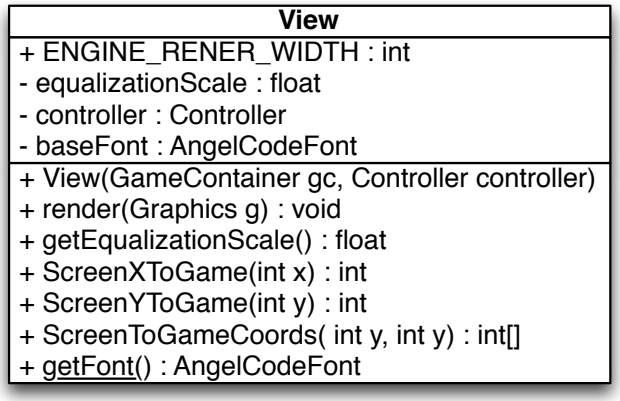
2: Klassendiagramm der Klasse Map



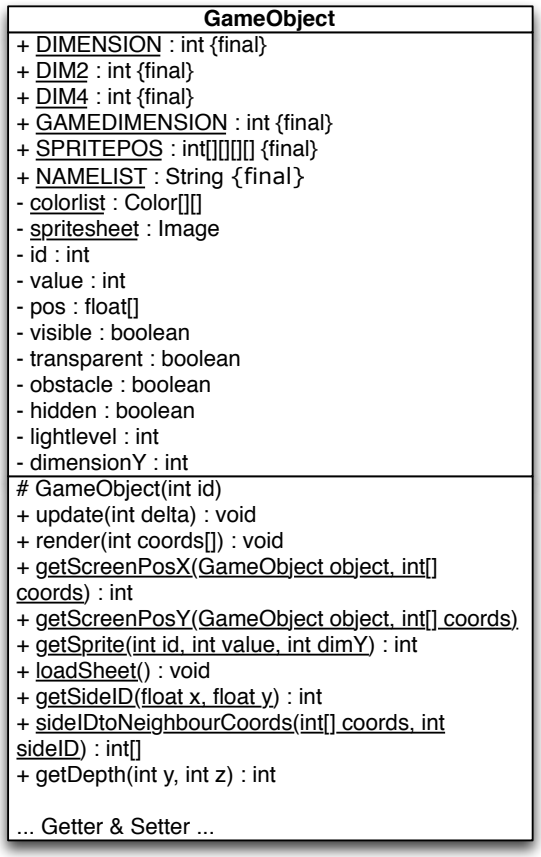
3: Klassendiagramm der Klasse Chunk



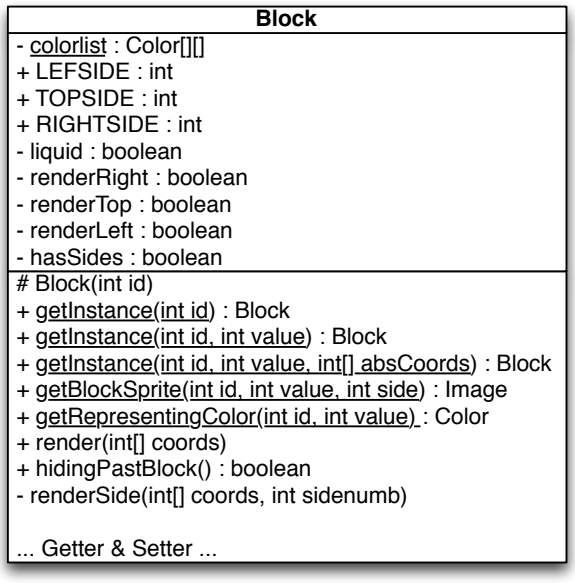
4: Klassendiagramm der Klasse Controller



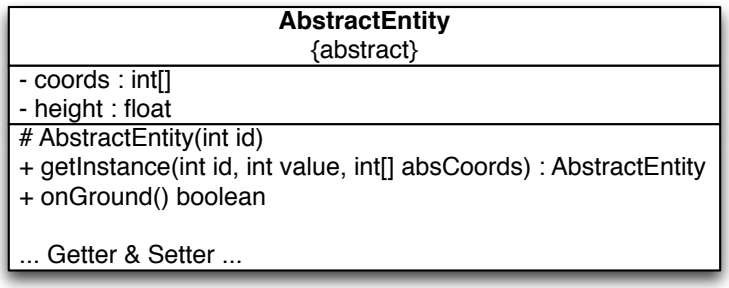
5: Klassendiagramm der Klasse View



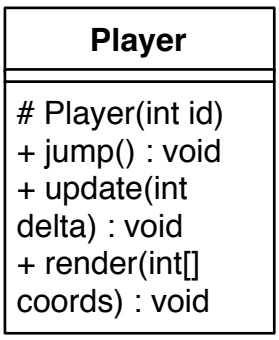
6: Klassendiagramm der Klasse GameObject



7: Klassendiagramm der Klasse Block



8: Klassendiagramm der Klasse AbstractEntity



9: Klassendiagramm der Klasse Player

Selbstständigkeitserklärung

Hiermit versichere ich, dass ich die vorstehende Arbeit selbständig und ohne fremde Hilfe angefertigt und alle Stellen, die wörtlich oder annähernd wörtlich aus Veröffentlichungen genommen sind, als solche kenntlich gemacht habe. Die Versicherung bezieht sich auch auf in der Arbeit gelieferte Zeichnungen, Skizzen, bildliche Darstellungen und dergleichen.

Willich, 21.03.13

Benedikt Vogler