Projekt: OpenGL und Spieleentwicklung

Oliver Bartels & Sebastian Funck

3. August 2015

Inhaltsverzeichnis

1 Allgemeines

1.1 Aufgabe

Die Aufgabe die wir uns gestellt haben war Spieleentwicklung mit OpenGL. Genauer gesagt eine Spieleengine zuentwickeln und mit dieser ein Spiel zu programmieren, welches sich an Spielen wie Dwarf Fortress, Prison Architect und Rim World orientiert. Es geht in dieses Spielen darum, dass man Spielfiguren auf einer Karte hat. Diese können nicht direkt befehligt werden, sondern viel mehr kann man nur Arbeitsaufträge erstellen.

Zudem haben die Spielfiguren in den genannten Spielen Bedürfnisse die befriedigt werden wollen wie Hunger, Durst, Schlaf, Moral und Gesundheit. Das Hauptziel ist oftmals einfach nur dass die Spielfigurengruppe überlebt.

1.2 Arbeitsteilung

Wir haben uns jeden Donnerstag in den 3 Präsenzstunden Aufgaben überlegt die als nächstes gemacht werden müssen. In den ersten zwei Wochen war dies recht schwierig, da wir versucht haben genau festzulegen was wir zum Ende der nächsten Woche geschafft haben wollten.

Ein Problem in den ersten zwei Wochen war, dass wir zwar ungefähr wussten wohin wir wollten, allerdings oft der Plan sich nach 1-2 Tagen wieder verändert hat weil andere Funktionalitäten aufeinmal wichtiger bzw. notwendiger waren. Dies hat dann den zum Anfang der Woche festgelegten Plan häufig über den Haufen geworfen.

Nach den ersten zwei Wochen sind wir dazu übergegangen all unsere Ideen, welche wir irgendwann mal ins Spiel bringen wollen, grob in Tickets festzuhalten. Sobald wir gemerkt haben ein Ticket wird in den nächsten 1-2 Tagen gemacht werden müssen, wurde dann die Aufgabenbeschreibung von uns entsprechend detaillierter verfasst. Somit haben wir flexibel auf lange Sicht planen können, und trotzdem hatten wir meistens die nächsten 1-2 Tage fest geplant woran wir arbeiten müssen.

Als unterstützendes Tool haben wir die Website Trello benutzt, indem wir die dortige Blackboardfunktionionalität als simples Ticketsystem verwendet haben.

Die Tatsache, dass wir in einer WG zusammen wohnen, hat auch für eine schnelle Rückkopplung und recht gute Problemlösung gesorgt und hat uns häufig geholfen.

1.3 Workload

Unser Workload war recht hoch, da wir ein doch sehr komplexes und großes Thema angegangen sind. Wir haben beide jede Woche mindestens 10 Stunden Arbeit investiert, teilweise bis an die 20 Stunden, aber wir haben im Vorfeld mit diesem Workload gerechnet und uns war auch der Schwierigkeitsgrad dieser Aufgabe bewusst. Es gab durchaus

Wochen in denen wir uns verschätzt haben, was aber dann an Problemen die während der Implementation anderer Features aufgefallen sind und behoben werden mussten lag.

1.4 Testbarkeit

Wir haben festgestellt, dass es nicht so leicht ist Tests für Spiele zu schreiben, jedoch konnten die kritischen Bereiche der Spieleengine gut abgedeckt werden. Viele Überlegungen kamen erst am Ende des Projektes, da wir selber nicht genau wussten wo diese kritischen Bereiche in unserem Code wirklich sind bzw. wie wir sie am besten testen können. Zum Beispiel sind die mathematischen *Value-Klassen* und das Pathfinding von elementarer Bedeutung und müssen daher gut getestet werden. Auch das Zusammenspiel von `World`, `GameObject` und `Component` sind zentraler Bestandteil der Spieleengine.

2 Code Konventionen

2.1 *Value*-Klassen

In den Klassen im package `de.orangestar.engine.values` werden fast ausschließlich öffentliche Exemplar-Variablen benutzt. Dies hat den Hintergrund, dass wir alle Instanzen dieser Klassen als *values* behandeln.

Value-Klassen sind zwar Objekte, jedoch wollen wir diese wie primitive Datentypen behandeln. Dass bedeutet *Value*-Klassen dürfen lediglich primitive Datentypen oder Referenzen auf andere *value*-Klassen enthalten.

Der Grund für diese Entscheidung ist, dass der Code übersichtlich und verständlich bleibt wenn wir zum Beispiel nur den x Wert der Position einer Transformation erhöhen wollen. Statt

```
Vector3f position = transform.getPosition();
position.setX(position.getX() + 10.0f);
```

können wir dies nun etwas intuitiver schreiben als

```
transform.position.x += 10.0f;
```

2.2 *Builder*-Klassen

Diese kommen zum Einsatz wenn eine Instanz einer Klasse über das Factory-Pattern erzeugt werden soll und die Anzahl der Parameter so groß ist, dass der Methoden-Kopf unübersichtlich lang werden würde.
Um dies zu vermeiden werden *Builder*-Klassen verwendet. Instanzen eben dieser können wie gewünscht das zu erzeugende Objekt konfigurieren und abschließend durch den Aufruf der Methode `build` das Objekt erzeugen (oder optional eine Exception werfen, falls die Konfiguration ungültig war).

3 Architektur

3.1 Ideen sowie Quellen

Viele unserer Pattern die Verwendung gefunden haben finden sie auf gameprogrammingpatterns.com

3.2 Allgemein

Allgemein `Engine.java` ist das Herz der Engine. Die Engine besteht aus einer kontinuierlich laufenden `mainloop` und ist somit eine Echtzeit-Anwendung. Innerhalb der `mainloop` ist die Engine in die verschiedenen Zuständigkeitsbereiche Logik und Darstellung unterteilt und die Logik ist zusätzlich in Spiel-Logik, Eingabe-Logik und Physik-Logik unterteilt. Für jeden dieser Bereiche gibt es einen zuständigen Manager:

- GameManager (Logik)
- RenderManager (Darstellung)
- InputManager (Eingabe-Logik)
- PhysicsManager (Physik-Logik)

Die Logik, und damit der Zustand der Spielwelt, wird in festen Intervallen aktualisiert. `GameManager.TICK_RATE` definiert die Zeit (in Millisekunden) zwischen zwei solcher Intervalle und sollte so gewählt sein, dass genug Zeit für die Darstellung der Spielwelt übrig bleibt. Üblicherweise werden mindestens 30 Frames pro Sekunde für ein flüssig wirkendes Bild benötigt, vermehrt geht die Spieleentwicklung aktuell zu einem Standard bei 60 Frames pro Sekunde über.

Die Implementation orientiert sich an der GameLoop von *gameprogrammingpatterns.com*.

3.3 World, GameObjects und Components

Die Spielwelt wird durch die Singletonklasse `World` repräsentiert und besteht aus `GameObjects`, die Objekte in der Spielwelt darstellen. Hierunter fallen zum Beispiel Spielfiguren, Bäume, Gegenstände, die Spielkarte auf der sich die eben Genannten befinden und vieles mehr.

Jedes `GameObject` kann mit Komponenten ausgestattet werden (siehe Komponenten-Pattern) die das `GameObject`, in den **2.1** genannten Zuständigkeitsbereichen, repräsentieren.

Analog zu den Zusändigkeitsbereichen gibt es folgende Komponenten:

- LogicComponent (Spiel-Logik)
- RenderComponent (Darstellung)
- InputComponent (Eingabe-Logik)

- PhysicComponent (Physik-Logik)

Der Lebenszyklus einer Komponente besteht aus den Zuständen *Initialisierung*, *Aktualisierung* & *Deinitialisierung*. Sobald die Komponente einem Spielobjekt zugewiesen wird, wird diese initialisiert und ist benutzbar.

Solange das Spielobjekt Teil der Spielwelt ist, wird die Komponente kontinuierlich aktualisiert. Bei der Aktualisierung einer Komponente wird ihre Hauptfunktionalität ausgeführt. Sobald die Komponente von ihrem zugewiesenen Spielobjekt entfernt wird, wird diese deinitialisiert und ist nicht mehr benutzbar. Im besten Falle ist die Komponente so implementiert dass sie jedoch durch erneute Initialisierung wieder benutzbar wird.

Durch den Komponenten-Pattern lassen sich einfach Komponenten an Spielobjekten austauschen und man kann verschiedene Verhalten eines Spielobjekts in verschiedene Komponentenimplementationen auslagern.

3.4 Flyweight & Model

Das Problem dass das Flyweight-Pattern beheben soll ist, dass es oft in der Spieleprogrammierung vor kommt das es gewisse `GameObject`-Klassen gibt die in großen Zahlen auftreten, aber nur einen sehr geringen Rendering-Aufwand haben. Da für jedes `GameObject` so theoretisch ein DrawCall getätigt werden müsste, würde sehr viel Overhead durch die OpenGL-Befehle produziert.

Um dies zuvermeiden wird das *Flyweight-Pattern* angewendet. Dieses besteht aus vielen Flyweight-Instanzen die nur den Logikteil enthalten und bei ihrer Erstellung der Model-Instanz lediglich mitteilen dass sie existieren. Somit wird nur ein DrawCall erzeugt und trotzdem haben wir für jedes Spielobjekt eine unabhängige `GameObject`-Instanz.

Genutzt wird das Pattern bei den Items, den Bäumen und den Props.

4 Rendering

4.1 OpenGL-API

Die OpenGL-API-Schnittstelle wird durch die Bibliothek *LWJGL* bereitgestellt und im folgenden soll die Funktionsweise von OpenGL 3.1+ grob erklärt werden.

Um OpenGL überhaupt benutzen zu können, muss erst einmal der sog. OpenGL-Kontext durch `GLFW.glfwCreateWindow`, `GLFW.glfwMakeContextCurrent` und `GLContext.createFromCurrent` erzeugt werden.

Einige Begrifflichkeiten:

- Vertex: Ein Punkt im OpenGL-Raum. Typischerweise bestehend aus Position im OpenGL-Raum, Farbe und/oder Texturkoordiante.
- Buffer: Ein Container für primitive Datentypen, mit zur Erzeugung festgelegter Größe. Wird unteranderem dafür benutzt um Vertexdaten im VRAM der Grafikkarte zuspeichern.
- Vertexbuffer (VBO): Ein Buffer der Vertexdaten enthält.
- Indexbuffer: Ein Buffer der Integers enthält.
- Instancebuffer: Ein Buffer der 4x4 Transformations-Matrizen enthält
- VertexArrayObject (VAO): Ein Containerobjekt für mehrere Buffer.
- Shader: Ein Shaderprogram das einem C-Programm ähnelt und auf einer GPU dafür sorgt wie die Vertices durch OpenGL dargestellt werden sollen.
- Texture: Bilddaten die im VRAM der Grafikkarte gespeichert sind.
- Drawcall: Der Aufruf an der OpenGL-API dass etwas gerendert werden soll.

OpenGL arbeitet seit 3.1+ ausschließlich mit Buffer, VertexArrayObjects und Shadern, davor wurden ausschließlich Konfigurationsbefehle benutzt.

Einige Rendertechniken:

- Normales Rendering: Es werden Vertices in einen Vertexbuffer geschrieben und der Shader rendert Vertex nach Vertex. Wird ein Vertex mehrfach benutzt wird dieser mehrfach in den Vertexbuffer geschrieben.
- Indexed Rendering: Jeder Vertex wird maximal einmal in den Vertexbuffer geschrieben. Zusätlich zum VertexBuffer wird ein Indexbuffer erzeugt in denen die Reihenfolge der Vertices geschrieben wird. Somit wird das Datenvolumen dass an die Grafikkarte gesendet wird bei hoher Vertex Mehrfachnutzung erheblich reduziert.
- Instanced Rendering: Wie normales Rendering, jedoch wird zusätzlich ein Instancebuffer erzeugt. Der Inhalt des Vertexbuffer wird pro Instanz im Instancebuffer gerendert. Dies ist sehr performant bei vielen Objekten die die gleiche Vertexstruktur haben.

- Instanced-Indexed Rendering: Wie der Name schon vermuten lässt, Instanced Rendering und Indexed Rendering kombiniert. Also zusätzlich zum Vertexbuffer wird noch ein Instancebuffer und ein Indexbuffer erzeugt.

Generell lässt sich sagen, dass wenn möglich, nur Indexed und Instanced-Indexed Renering verwendetwerden sollte um die Datenmenge die an die Grafikkarte geschickt wird und die Anzahl der Drawcalls minimal zu halten.

Um einen Drawcall vorzubereiten muss zuerst ein VAO erzeugt werden. Anschließend werden die der Rendertechnik entsprechenden Buffer erzeugt und dem VAO zugewiesen. Nun werden die jeweiligen Daten/Vertices in die Buffer geschrieben und im VRAM der Grafikkarte gespeichert. Damit die Grafikkarte weiß wie die Vertices dargestellt werden sollen muss noch ein Shaderprogramm und gegebenenfalls eine Textur gesetzt werden. Abschließend wird der Drawcall getätigt und es kommt zur grafischen Ausgabe.

4.2 Batches

Da OpenGL eine C orientierte API ist werden keine Objekte aus den Funktionsaufrufen zurückgegeben, sondern in aller Regel nur Integers die wie Pointer verstanden werden können (oft *Handle* oder *ID* genannt). Die Batches haben das Ziel die Kommunikation mit der OpenGL-Schnittstelle zu abstrahieren, sodass wir uns nicht um Buffer, VAO's und ähnlichem kümmern müssen, da hier recht schnell Fehler gemacht werden können die schwer zu debuggen sind, und in aller Regel der Code in diesen Bereichen gleich ist.

Die Batches sollen also lediglich Vertices, Indizes, Instanzen, Texturen und Shader übergeben bekommen, und die direkte Kommunikation mit der OpenGL-Schnittstelle übernehmen

4.3 Actors

Actors sind eine weitere Abstraktionsebene über den Batches um abschließend das verbleibende Low-Level Rendering (Vertices, Indizes und Shader) zu abstrahieren. Somit kann man lediglich mit Actors, Texturen und Instanzen schnell und ohne viel Aufwand und Wissen über OpenGL, grafische Ausgaben erzeugen.

5 Logik

5.1 Player

Ein `Player` ist eine Spielfigur in unserer Spielwelt und alle Spielfiguren können sich unabhängig voneinander bewegen. Eine Spielfigur ist ein `Gameobject` und besitzt eine logische Komponente, eine grafische Komponente und eine physikalische Komponente. Die logische Komponente kümmert sich um die KI und Fortbewegung der Spielfigur. Sie setzt die Geschwindigkeit auf den Achsen entsprechend an der physikalischen Komponente, die dann effektiv die Spielfigur bewegt. Die grafische Komponente stellt lediglich das Bild der Spielfigur dar.

Damit die Spielfiguren auch alle als eine Einheit Befehle zugeordnet bekommen können, gibt es die `PlayerGroup`-Klasse, der die Spielfiguren automatisch hinzugefügt werden. Diese verteilt die Befehle an die Spielfiguren sobald eine Aktion und eine Spielfigur ohne Aktion verfügbar ist.

Die Aktionen, die die Gruppe erhält, werden per Benutzereingabe erzeugt.

5.1.1 Actions & AIActions

Eine Aktion (`Action`) bietet eine Funktion an, die in den meisten Fällen die Spielwelt oder das Verhalten beeinflusst. Aktionen können zusammen mit einem Aktionen-Automat (`ActionProcessor`) als Zustandsautomat betrachtet werden, bei dem die Aktionen die Zustände sind und der Aktionen-Automat der Automat. Jede Aktion hat eine Folgeaktion, die ausgeführt wird sobald die Aktion beendet ist. Ist diese Folgeaktion jedoch `null` hat der komplette Automat geendet.

In Ergänzung zur `Action` und zum `ActionProcessor` gibt es die `AIAction` und den `AIActionProcessor`, die KI-Aktionen ausführen können. Hierfür wird derzeit eine `PlayerLogicComponent` benötigt die als KI-Einheit fungiert. (Wir planen derzeit diese KI-Funktionalität aus der `PlayerLogicComponent` in eine Klasse `AILogicComponent` zu extrahieren.)

5.2 Tree

Wie der Name schon sagt handelt es sich hierbei um Bäume die in der Spielwelt zufinden sind. Da in unserem Spiel viele Bäume generiert werden, wobei sich jeder Baum exakt gleich verhält, bietet es sich an das *Flyweight & Model-Pattern* anzuwenden. Jeder Baum ist ein `Gameobject` und wird mit einer physikalischen Komponente bestückt, die lediglich dafür gebraucht wir um zu erkennen welcher Baum angeklickt wurde.

Da wie schon erwähnt alle Bäume sich gleich verhalten und auch gleich Aussehen, benötigen wir noch ein `TreeModel` welches eine logische Komponente erhält um die alle existierenden Bäume zu verwalten und eine grafische Komponente, um alle Baum Instanzen darzustellen.

5.3 Item

Items sind Gegenstände die in der Welt liegen können. Es war angedacht, dass Spielfiguren diese Gegenstände später auch aufnehmen und herumtragen können, und das wenn z.B. Gebäude gebaut werden dafür Holz benötigt wird.

Bei den Items nutzen wir das *Flyweight & Model-Pattern*, und zusätzlich noch das Prototype-Pattern um unser Spiel einfach erweiterbar zu halten und gleichzeitig eine einfache Veränderung sowie Erstellung während der Laufzeit bei den Items durchzuführen. Wir haben eine `Item`-Klasse, welche zur Erstellung einen Gegenstandstypen (`ItemType`) benötigt. Dieser Gegenstandstyp ist ein Prototyp eines Gegenstandes und legt sein Verhalten bzw. seine visuelle Darstellung fest.

Um ein Gegenstand zu erzeugen benutzen wir einen `ItemSpawner`, der einen Spawner im *Prototyp-Pattern* darstellt. Damit benötigt man nur einen `ItemSpawner` der einen bestimmten Prototypen besitzt und sobald wir weitere Gegenstände, die dem Prototypen entsprechen sollen, erzeugen wollen, brauchen wir lediglich erneut auf den `ItemSpawner` zugreifen und dieser erzeugt eine neue Instanz mit den selben Eigenschaften.

5.4 Props

Props sind Dekorationen in der Spielwelt. Zum Beispiel Blumen, Steine oder Seesterne die keinen funktionalen Nutzen haben und lediglich zur Ausschmückung der Spielwelt dienen. Auch hier haben wir das *Flyweight & Model-Pattern* und das *Prototype-Pattern* analog zum Item angewendet.

5.5 Map

Die `Map` ist die Spielkarte auf der sich die Spielfiguren befinden. Sie besteht aus mehren Regionen (`MapChunk`). Im Gegensatz zu den Spielen die wir als Vorbilder genommen haben, haben wir das Ziel, dass die Spielkarte quasi unendlich groß ist. Wenn der Spieler die Kamera in eine beliebige Richtung bewegt, sollen die MapChunks, sofern sie nicht schon generiert wurden, kontinuierlich zur Laufzeit erzeugt werden.

Ein großes Problem war dass die MapChunks nicht flüssig ineinander übergingen, sondern harte Kanten hatten. Dies lag daran dass wir zur erst einfach `Random` als Pseudozufallsfunktion genutzt hatten. Durch einige Recherche haben wir dann erfahren dass man in so einem Fall eine kohärente zwei-dimensionale Zufallsfunktion benötigt. Wir haben uns hierbei für die öffentlich freiverfügbare Simplex-Zufallsfunktion entschieden.

5.6 Map-Generierung (Siehe `MapChunkGenerator.generateMapChunk`)

5.6.1 NoiseMap-Generierung (Siehe `MapChunkGenerator.generateNoiseMap`)

Zuerst wird eine kohärente Pseudo-Zufallsfunktion benötigt, die die Eigenschaft hat bei gleicher Eingabe, gleiche Ergebnisse zuliefern. Kohärent bedeutet in diesem Fall dass es zwischen zwei Ergebnissen immer einen fließenden Übergang gibt (Parameter sind Fließkommazahlen).

Zusätzlich benötigen wir einen NoiseMap-Generator der Mithilfe der Pseudo-Zufallsfunktion eine Noisemap erstellt. Eine Noisemap ist ein zwei-dimensionales double Array und enthält Werte zwischen `-1.0d` und `1.0d`.

Der NoiseMap-Generator wird mit den kohärenten Zufallszahlen gefüttert und erzeugt so eine natürlich aussehende, zusammenhänge NoiseMap, die nun noch in ein verarbeitbares Format konvertiert werden muss. Der Einfachheit halber wird das NoiseMap-Array noch in ein Array mit Werten zwischen `0.0d` und `1.0d` konvertiert.

5.6.2 Tile/Surface-Generierung (Siehe `MapChunkGenerator.generateTiles`)

Bevor aus der NoiseMap eine TileMap wird, die grafisch dargestellt werden kann, muss ein sogenanntes TileSet erstellt werden. Ein TileSet ist eine Texture die aus mehreren kleinen Unterbildern besteht (siehe `texture/tileset.png`). Ein solches Unterbild wird auch Tile genannt. Mehrere Tiles die Teil eines gleichen Oberflächentyps sind (wie z.B. Gras, Sand, …) werden in einem `Surface` zusammen gefasst. Ein `TileSet` besteht also aus mehreren Oberflächen, die aus mehreren Tiles bestehen. In der Methode `MapChunkGenerator.generateTiles` wird lediglich eine NoiseMap in ein zweidimensionales Tile Array übersetzt.

5.6.3 TileMap-Generierung (Siehe `TileMap.setData`)

Eine `TileMap` ist ein `Actor`, die ein `TileSet` und ein zwei-dimensionales `Tile` Array visualisiert. Besonders an dieser `TileMap` Implementation ist, dass zwischen zwei Tiles die verschiedenen Oberflächen zugehören, automatisch Übergänge generiert werden.

6 Pathfinding

Damit die Spielfiguren auf der `Map` laufen können, muss dafür gesorgt werden, dass sie mit einer Zielkoordinate selbstständig zum Ziel finden. Um dies zu erreichen, bedienen wir uns des Pathfinding (Wegfindungs) Algorithmus *A* Search*. Dieser Algorithmus sucht aus 2 Punkten, dem Start- und dem Endpunkt, den kürzesten Weg und gibt diesen an die Spielfiguren zurück, diese können dann mit diesen Pfad den perfekten Weg zum Ziel finden.
Wenn dieser Algorithmus falsche Werte ausgibt oder nicht funktionieren würde, könnte so gut wie keine Interaktion mit der Spielwelt geschehen, daher ist er in der Funktion für unser Spiel sehr wichtig.
Bei einer *A** Suche werden heuristische Werte benutzt, also Abschätzungen auf den noch zu gehenden Weg zum Ziel von der aktuellen Position aus. Eine Koordinate in unserer Welt wird von dem Algorithmus ausgewählt,wenn der schon gegangene Weg bis zu der Position und die Abschätzung zusammen addiert am geringsten ist. Dabei werden alle erreichbaren, aber noch nicht besuchten Knoten zur Berechnung verwendet.

Sobald das Ziel erreicht wurde, wird der Weg rückwärts konstruiert, da wir wissen wie wir den vorherigen Knoten eines Knotens speichern. Nachdem der Weg konstruiert wurde, wird er zurück an die Spieler gesendet und von dort aus wird eine Spielfigur den konstruierten Weg ablaufen. Zur Erläuterung von diesem Algorithmus kann auch folgende Grafik herangezogen werden: A*-Search Visualisierung

7 Tests

Bei der Testabdeckung haben wir uns darauf verständigt, zuerst die elementaren Funktionen der Spieleengine ab zudecken, da diese das Grundgerüst für alles weitere bilden. Besonders wichtig sind hierbei die Tests der *Value-Klassen*, da diese von essentieller Bedeutung für die Korrektheit des Projektes sind.

Zu den elementaren Funktionen der Spieleengine gehören die mathematischen Klassen, das Pathfinding, das Zusammenspiel zwischen `GameObject` und `Component`, und das Zusammenspiel zwischen `World` und `GameObject`.

8 Benutzerbeschreibung

8.1 Starten des Programms

Zum Starten unseres Spiels haben wird mindestens Java 8, sowie eine Grafikkarte die OpenGL 3.3 unterstützt benötigt. Es gibt zwei Möglichkeiten das Spiel zu starten, zum einen direkt aus Eclipse heraus, allerdings kann es sein dass einige Einstellungen in Eclipse getätigt werden müssen, oder durch starten der folgenden `OrangeStar\target\de.orangestar-0.0.1-SNAPSHOT.jar`. Die Jar-Datei sollte so konfiguriert sein dass ein simples Doppelklicken zum starten genügt (sofern *.jar-Dateien automatisch mit Java geöffnet werden).

8.2 Spielablauf

Das Spiel hat noch kein genaues Spielziel, es ist momentan ein sog. Sandkasten Spiel in dem man die Welt mittels verschiedener Aktionen verändern kann. Die Kamera lässt sich mit den Tasten W,A,S & D steuern und man kann mit dem Mausrad den Zoomfaktor einstellen.

Derzeit kann man in der Welt durch einen Linksklick auf einen Baum eine Baum-Fällen-Aktion erstellen, die dann von einer Spielfigur ausgeführt wird und dazu führt dass der Baum gefällt wird und ein Stück Holz hinterlässt.

Mit einem Rechtsklick in die Spielwelt wird eine *Gras-Pflanzen-Aktion* erzeugt, die bewirkt dass sobald eine Spielfigur diese Aktion zugewiesen bekommt, die Spielfigur sich an die Position begibt und, sofern an dieser Position noch kein Gras wächst, Gras pflanzt. Vorraussetzung ist dass die Position in der Spielwelt von der Spielfigur auch erreicht werden kann. Wenn eine Position nicht erreicht werden kann, dann bewegt sich die Spielfigur nicht, dies gilt für beide Aktionen.

9 Biliotheken und Code Dritter

9.1 Anwendung

9.1.1 LWJGL

LWJGL ist ein sogenanntes C-Binding der OpenGL-Schnittstelle in Java. Sie ermöglicht es die Grafikkarte über OpenGL-Befehle anzusprechen und somit grafische Ausgabe zu erzeugen.

9.1.2 Apache Commons

Die einzige Klasse die aus dem Apache Commons benutzt wird ist die `NodeCachingLinkedList`. Das Problem ist, dass sehr häufig neue Tilemaps erzeugt und berechnet werden müssen. Da wir im vornherein nicht wissen wie viele Vertices hierbei erzeugt werden, müssen diese in einer Liste gespeichert werden.
Dies führt bei Benutzung einer Listen-Standard-Implementation des JRE dazu, dass der Speicher von den Nodes geflutet wird und der Garbage-Collector diese recht häufig aufräumen muss und somit die CPU blockiert. Um dies zu entgehen wird nun die `NodeCachingLinkedList` verwendet, da bei dieser Implementation Nodes wiederverwendet werden.

Coherent Simplex Noise Java Implementation

`Coherent Simplex Noise` ist eine Pseudo-Zufallsfunktion und wird verwendet um kohärente MapChunks zu generieren. Die hierfür benötigte Pseudo-Zufallsfunktion muss von den Parametern abhängig, immer die gleichen Pseudozufallszahlen generieren. Die Entscheidung für diese Implementation fiel aufgrund der organisch wirkenden erzeugten Zufallskarten, der guten Performance und der Public-Domain Lizenz.

MurmurHash3 Java Implementation

`MurmurHash3` ist eine sehr performante Hashfunktion, zukosten der Anzahl von Kollisionen. Daher ist diese Funktion nicht für Sicherheitszwecke geeignet, allerdings für andere Gebiete des Hashen.
Zuerst war angedacht diese Hashfunktion als Pseudo-Zufallsfunktion zur MapChunk-Generierung zu benutzten, allerdings hat sich die `Coherent Simplex Noise` Pseudo-Zufallsfunktion als geeigneter erwiesen.

9.1.3 Lizenzen

- LWJGL
  BSD-License
  http://www.lwjgl.org/license
- Apache Commons
  Apache License 2.0
  http://commons.apache.org/proper/commons-daemon/license.html
- Coherent Simplex Noise Java Implementation
  Public-Domain
  https://gist.github.com/KdotJPG/9bbab7d3655b82811b24
- MurmurHash3 Java Implementation
  Public-Domain
  https://github.com/yonik/java_util/blob/master/src/util/hash/MurmurHash3.java

10 Fazit und Ausblick

Das Fazit ist soweit recht positiv, da wir eine 2D-Spieleengine aufgebaut haben und die elementaren Grundlagen des Spiels gelegt haben. Die Spielfiguren bewegen sich auf der Karte, sie arbeiten erstellte Actions ab, die `Map` wird kontinuierlich dynamisch generiert und sieht unseren Vorstellungen entsprechend aus. Was wir leider nicht ganz erreicht haben ist eine textuelle Darstellung auf dem Bildschirm und ein wirkliches Spielziel, da der Aufwand eine vernünftige Engine mit guter Architektur zuschreiben doch sehr viel Zeit gekostet hat.

Was uns zum Schluss einige Probleme beim Testen bereitet hat ist die ausgiebige Nutzung von Singletons, besonders bei der `World`. Wie wir in den letzten Tagen selber und durch ein Video von einem *Google Clean Code Talk* gelernt haben erschweren diese das Testen enorm, da zwischen der Ausführung von zwei Testmethoden die Singletons nicht zurückgesetzt werden und man sich mit unschönen Methoden behelfen muss. Dies hat wieder etwas Zeit am Ende gekostet da einige Anpassungen an der Architektur gemacht werden mussten. Wir planen nach dem Projekt diese Singletons soweit wie möglich noch zu entfernen und zu überlegen inwieweit *Dependency Injection* in einer Echtzeitanwendung Sinn macht.

Ein Makel der bei der Spielwelt noch existiert ist, dass noch keine Übergangstiles zwischen zwei MapChunks generiert werden, und so an einigen Stellen noch harte Übergänge sind.

Alles in allem war der Aufwand riesig, hat uns aber entsprechend auch Spaß gemacht und wir werden das Spiel auf jedenfall in unserer Freizeit noch weiter entwickeln.

Ziele für die Zukunft:

- Mehr Interaktion ermöglichen
- Text Darstellung
- evtl. Portierung auf eine andere Programmiersprache (C++) um auch dort unsere Kenntnisse zu erweitern
- Ziel und Szenario des Spiels konkretisieren