

Projektdokumentation

„ZoneWars“

HdM Stuttgart, Studiengang Medieninformatik

Rene Schneider, rs034

Inhaltsverzeichnis

| | | |
|-----|---|----|
| 1.0 | Einführung..... | 2 |
| 2.0 | Architekturüberblick..... | 3 |
| 2.1 | Architektur im Detail: Netzwerkschicht..... | 4 |
| 2.2 | Architektur im Detail: Terrain..... | 6 |
| 2.3 | Architektur im Detail: Einheiten..... | 8 |
| 2.4 | Architektur im Detail: Effekte und Animationen..... | 10 |
| 2.5 | Architektur im Detail: Lua-Integration..... | 11 |
| 2.6 | Architektur im Detail: Einheiten-API..... | 15 |
| 3.0 | Fazit..... | 17 |

1.0 Einführung

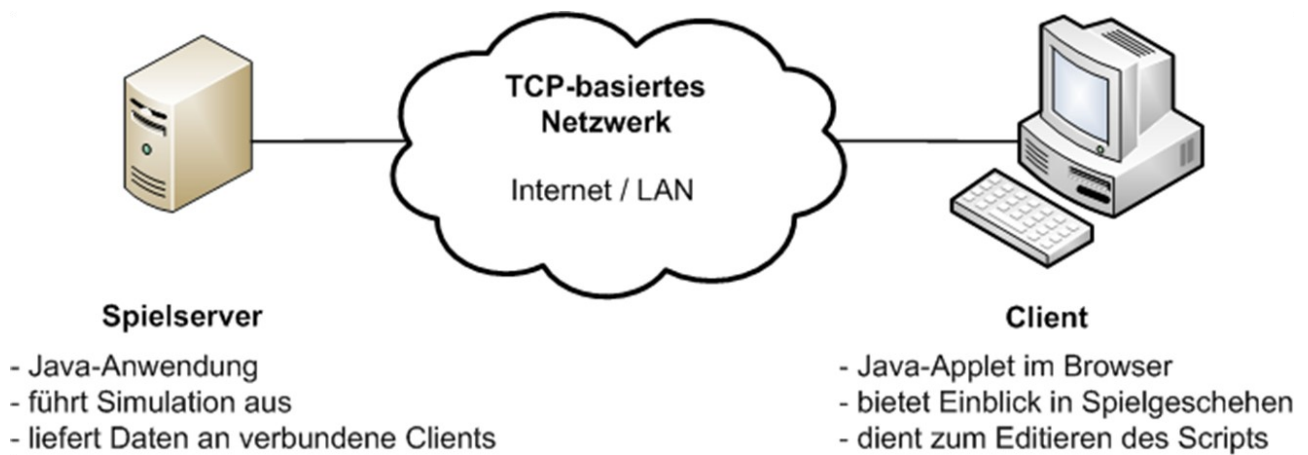
ZoneWars ist ein Multiplayer-Netzwerkspiel ähnlich bekannten Strategiespielen mit bewaffneten Einheiten, einer simulierten Landschaft etc., aber der Besonderheit, dass die Spieler diese Einheiten nicht direkt selbst befehligen. Stattdessen ist es ihre Aufgabe, Verhaltensscripts in einer Scriptsprache für diese Einheiten zu schreiben, um ihnen auf diese Weise eine Art „künstliche Intelligenz“ einzuhauchen. Dadurch stehen dem Spieler fast unbegrenzte taktische Möglichkeiten zur Verfügung, um sich gegen seine Gegner (bzw. deren Scripts) durchzusetzen, denn die einzige Grenze für die „Intelligenz“ der eigenen Einheiten ist das eigene Können als Programmierer.

Der Name „ZoneWars“ ist dabei nicht frei erfunden: vor langer Zeit wurde ein Spiel namens „Core Wars“ in Programmierer- und Hackerkreisen bekannt und hat sich dort bis heute eine gewisse Popularität erhalten. In diesem Spiel schreibt man in einer Assembler-ähnlichen Sprache kleine Programme, die sich im Speicher eines simulierten Rechners ausbreiten und das „Gegnerprogramm“ dabei vernichten sollen. ZoneWars baut auf einer ähnlichen Idee auf, möchte aber mit einem etwas weniger abstrakten Szenario (Einheiten statt nackter Opcodes im Speicher, Landmasse statt Speicher, Scriptsprache statt Assemblerinstruktionen) und Netzwerkfähigkeit eine etwas bessere Zugänglichkeit und den heutigen Möglichkeiten angepasste Multiplayerfähigkeiten bieten.

In dieser Projektdokumentation sind Erläuterungen zur Architektur von ZoneWars, die Kerngedanken bei der Entwicklung und eine abschließende kritische Bewertung des Projektverlaufs und –ergebnisses enthalten.

2.0 Architekturüberblick

ZoneWars basiert auf einer klassischen Client/Server-Infrastruktur. Der Server ist dabei eine Java-Anwendung, die Client-Anwendung läuft als Java-Applet in allen gängigen Browsern und kann dadurch einfach in eine beliebige Internetseite eingebunden werden. Die Kommunikation zwischen beiden läuft über eine während der Spielsession durchgehend gehaltene TCP-Verbindung.

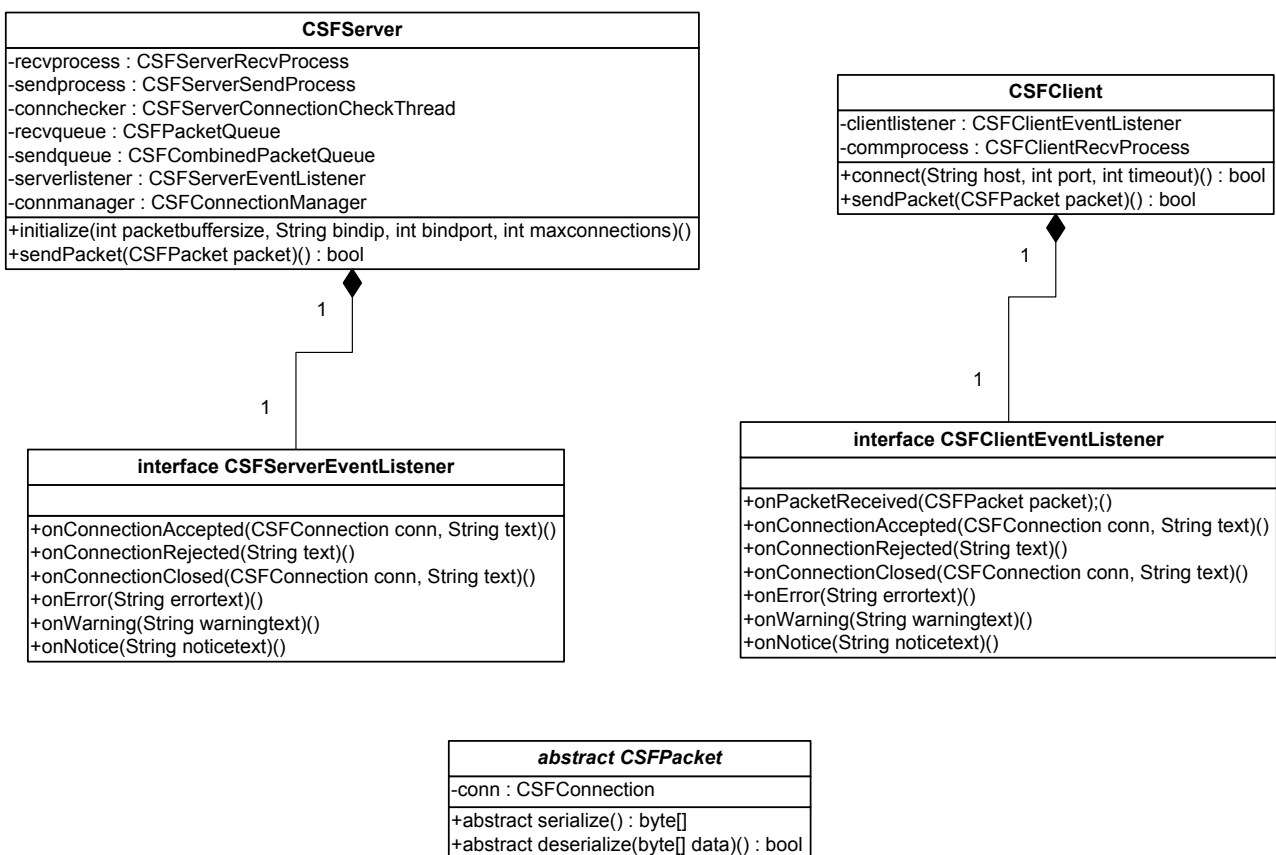


2.1 Architektur im Detail: Netzwerkschicht

Die Netzwerkkommunikation von ZoneWars erfolgt mittels eines selbstentwickelten Frameworks (im folgenden CSF - „Client Server Framework“ genannt), welches sich um die Verbindung kümmert, und einen Mechanismus, der das Versenden ganzer Java-Objekte kapselt. Tatsächlich „über die Leitung“ gehen XML-Nachrichten als serialisierte Repräsentation eines Objekts, die am Zielort wieder zu einem Objekt zusammengesetzt werden.

Diese Art der Kommunikation unterscheidet sich signifikant von gängiger RMI-Methodik, da hier keine Referenzen auf Objekte einer anderen Maschine übergeben werden. Stattdessen wird der State von Objekten direkt übertragen und am Zielort in eine Kopie des ursprünglichen Objekts injiziert (im Endeffekt erfolgt also eine Message-basierte Kommunikation).

Das CSF kümmert sich dabei hauptsächlich um das Management der TCP-Verbindung zur Gegenstelle.



Dies sind die fünf zentralen Klassen bzw. Interfaces des Frameworks selbst. Der Server besitzt eine Instanz von CSFServer, jeder Client eine von CSFClient. Rückmeldung von diesen Instanzen bekommen Client und Server über einen Listener, der bei Verbindungsauf- und abbau, bei Fehlern und Warnungen aufgerufen wird.

Es fällt auf, dass der ClientEventListener eine onPacketReceived-Methode besitzt, der ServerEventListener aber nicht. Der Grund dafür ist das unterschiedliche Paketmanagement beider Komponenten: serverseitig werden die Pakete bei Empfang in einer Queue zwischengelagert, bis der Server sie dort abholt (die Queue ist blockierend, so dass ein Verarbeitungsthread sich einfach „auf die Lauer“ legen kann bis neue Pakete eintreffen). Clientseitig wird für jedes empfangene Paket die onPacketReceived-Methode aufgerufen.

Es gibt außerdem noch einen weiteren Unterschied, der hier nicht direkt sichtbar ist: die Socketverbindung wird serverseitig über SocketChannels aus dem NIO-Paket abgewickelt, clientseitig kommt ein klassischer Socket zum Ein-

satz. Dadurch kann der Server theoretisch viele hundert Clients gleichzeitig halten, ohne an hunderten Threads zu ersticken.

Das sogenannte CSFPacket ist eine abstrakte Klasse, die ein übertragenes Paket darstellt. Sie ist abstrakt, weil es Aufgabe der konkreten Anwendung ist, die Serialisierungs- und Deserialisierungsroutinen zu implementieren.

An dieser Stelle ist dann auch die Grenze der Funktionalität des CSF-Frameworks erreicht. Um die konkrete Serialisierung/Deserialisierung von Java-Objekten kümmert sich ein Teil des Programms (in diesem Fall ZoneWars) selbst, nicht mehr das Framework. Man kann dies positiv oder negativ sehen: das CSF bleibt auf diese Weise relativ leichtgewichtig, weil es sich wirklich nur um das Handling der Verbindung (bzw. der vielen Verbindungen auf Serverseite) kümmern muss und sich aus konkreten Details des Kommunikationsprotokolls heraushält. Andererseits ist es so für die hier gewünschte Anwendung noch relativ „nutzlos“.

Es müssen also von Seiten der Anwendung nun Komponenten bereitgestellt werden, die das CSF als „Verbindungsmanager“ nutzen und darüber den Versand von Java-Objekten kapseln.

Für die Organisation der Serialisierung der Objekte nutze ich ein recht primitives, manuelles Verfahren: jedes zu versendende Objekt implementiert ein Interface, welches feste Methoden zur Serialisierung und Deserialisierung vorschreibt. Diese müssen in jedem Objekt von Hand implementiert werden, d.h. es müssen von Hand alle gewünschten Attribute in das XML-Dokument geschrieben werden, welches anschließend als serialisierte Repräsentation des Objekts „über die Leitung“ geschickt wird. Ebenso muss die Deserialisierung implementiert werden. Hier wäre es ebenso denkbar, einen Automatismus entweder über automatische Codegenerierung (zwecks optimaler Performance) oder über einen Attributzugriff per Reflection (einfacher, aber vermutlich langsamer als generierter Code) zur Vereinfachung der Programmierung einzufügen, denn die für jedes Objekt manuell zu implementierenden Methoden zur Serialisierung/Deserialisierung sind weitestgehend repetitiv. Mit einem Automatismus würde man zwar vermutlich etwas an Flexibilität verlieren, sich aber auch beim Programmieren stark entlasten.

Auch die Aufteilung größerer Objekte in mehrere Pakete für den Versand übers Netzwerk ist in der Anwendung implementiert. Hierum kümmert sich eine Kapselungsklasse der eigentlichen CSFPacket-Klasse, das sogenannte BigPacket. Große Objekte werden einfach vor dem Versand in ein BigPacket gekapselt, beim Empfänger werden sie wieder (für den Rest der Anwendung transparent) zusammengesetzt.

Die Netzwerkschicht von ZoneWars ist an einigen Stellen noch verbesserungswürdig. So erfolgt momentan die XML-Verarbeitung über einen DOM-Baum. Zugunsten besserer Performance wäre hier der Weg über einen SAX-Parser sicher vorteilhaft. Ebenso ist es unbefriedigend, dass für jedes Objekt manuell eine Menge Code zur Serialisierung und Deserialisierung geschrieben und getestet werden muss. Hier wäre eine automatische Codegenerierung besser, um den Aufwand bei „Standardfällen“ gering zu halten und für Spezialfälle trotzdem die Möglichkeit des manuellen Eingriffs zu bewahren.

2.2 Architektur im Detail: Terrain

ZoneWars arbeitet mit isometrischer 2D-Grafik zur Darstellung des Terrains sowie der Einheiten und Effekte. 2D-Grafik kommt zum Einsatz, weil das Spiel ohne Installation eines lokalen Clientprogramms und möglichst direkt im Browser laufen sollte. Den isometrischen Blickwinkel habe ich aus rein ästhetischen Gründen gewählt: ein Blick „aus der Vogelperspektive“ wäre genauso tauglich gewesen und hätte einige Dinge vereinfacht (dazu später noch mehr), aber die isometrische Ansicht ist vor dem Zeitalter der 3D-Echtzeitstrategie der bei weitem dominierende Blickwinkel in allen artverwandten Spielen gewesen, ist daher allen Spielern vertraut und kommt der 3D-Ansicht am nächsten.

Bei einem Spiel mit Einheiten auf virtuellem Terrain ist bereits zu Beginn der Entwicklung eine zentrale Entscheidung zu treffen, die später vieles beeinflusst: wie sieht das Koordinatensystem aus und wie sind die Felder angeordnet? Auch wenn man speziell in modernen, kommerziellen 3D-Echtzeitstrategiespielen die einzelnen Felder des Terrains nicht mehr sieht und nur mit Mühe errahnen kann, so sind sie immer noch im Hintergrund ständig präsent und legen fest, wie sich Einheiten auf dem Terrain bewegen lassen, welche Wege sie nehmen können und welche nicht. Aber während sich in 3D-Spielen die Feldanordnung nicht unbedingt in der Grafikausgabe niederschlägt, besteht zwischen beidem in 2D-Spielen üblicherweise eine enge Bindung: die Felder, auf denen sich die Einheiten bewegen, entsprechen meist auch den Feldern, aus denen die Grafik letztendlich zusammengesetzt wird.

Für ZoneWars habe ich eine Feldanordnung genommen, die schon in einem Klassiker der Spielegeschichte Verwendung fand. Gemeint ist das Wirtschaftsstrategiespiel „Transport Tycoon“, welches sein Terrain aus Vierecken zusammensetzte und damit sowohl flaches Land als auch Berge darstellen konnte. Die Entscheidung für diese konkrete Feldanordnung und die grundlegende Implementierung erfolgten zwar schon vor Beginn des eigentlichen Projekts, ich werde sie in diesem Kapitel aber trotzdem der Vollständigkeit halber kurz vorstellen.

Für die Darstellung eines lückenlosen Terrains waren insgesamt 15 unterschiedlich geformte Tiles (eine Tile ist eines der angesprochenen Vierecke) nötig:

- Eine vollkommen flache Tile für ebenen Boden (A)
- Vier „gerade“ Bergkanten, ausgerichtet nach Nordwesten, Nordosten, Südwesten und Südosten (B)
- Acht „geknickte“ Bergkanten, ausgerichtet nach Norden, Süden, Westen und Osten; jeweils paarweise (C)
- Zwei „doppelt geknickte“ Bergkanten (D)

Beachtet man dann noch einige Grundregeln wie z.B. die, dass zwischen zwei Tiles kein Höhenunterschied größer 1 bestehen darf, kann daraus ein geschlossenes Terrain mit Bergen und Tälern gebildet werden.

Durch Aufhellen bzw. Abdunkeln der Textur der Bergkanten-Tiles (oder im Falle der „geknickten“ Tiles auch nur Teilen davon) kann man dieses Terrain relativ einfach „plastisch“ wirken lassen, so dass der Eindruck einer dreidimensionalen Landschaft entsteht.

Das Koordinatensystem, welches in ZoneWars Verwendung fand, besitzt eine X-Achse in der Horizontalen und eine Y-Achse in der Vertikalen. Es schien das einfachste Koordinatensystem für diese Art Terrain zu sein, erwies sich im Nachhinein aber als Mißgriff. Im praktischen Einsatz ergaben sich vor allem folgende zwei Probleme:

1. Die zueinander versetzten Zeilen (siehe Abb. 2)
2. Die schlechten Proportionsverhältnisse: eine Tile ist ohnehin schon 1,67fach breiter als hoch, und durch die in Punkt 1 angesprochenen versetzten Zeilen verstärken sich diese ungleichen Proportionen noch weiter



Abb. 1: Alle Tile-Arten im Überblick

Durch diese beiden Begebenheiten gestaltete sich das Finden von Antworten auf für die Fortbewegung der Einheiten elementare Fragen wie „Welche Koordinate hat die Tile nordöstlich von der aktuellen Position?“ schwieriger als nötig. Ebenso wird durch das schlechte Proportionsverhältnis die Einschätzung z.B. von Sichtreichweite einer Einheit schwierig: die Einheit mag einen Radius von 10 Feldern Sichtreichweite haben, auf dem Monitor sind 10 Felder in der Horizontalen aber mehr als doppelt so viele Pixel breit als in der Vertikalen hoch.

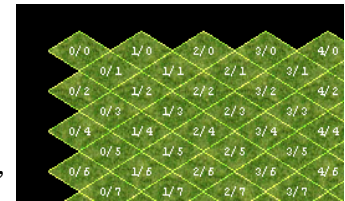


Abb. 2: Koordinaten (X/Y)

Vermutlich wäre es - rückblickend gesehen - besser gewesen, das Koordinatensystem um 45° zu drehen und die Welt quasi als „große Raute“ darzustellen statt in Form eines Rechtecks. Auf diese Weise wäre Problem 1 überhaupt nicht aufgetreten, und die ungleichen Proportionen wären wesentlich weniger stark ausgeprägt.

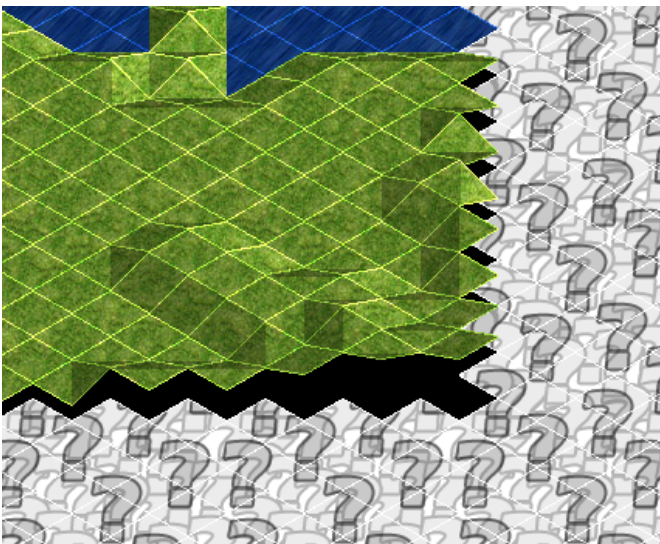


Abb. 3: Unbekanntes Terrain

Technisch gesehen liegen die Daten über das Terrain nicht beim Client, sondern werden in Einheiten von 5×10 -Rechtecken dynamisch vom Server abgerufen, sobald der User in ein noch unbekanntes Gebiet scrollt. Normalerweise geschieht dies vollkommen unsichtbar für den User, da die Daten angefordert werden, noch bevor der User beim Scrollen wirklich das unbekannte Gebiet erreicht. Bei hohen Netzwerklatenzen oder Serverlasten kann es aber passieren, dass die Informationen nicht rechtzeitig ankommen - in diesem Fall sieht der User kurzzeitig eine Fläche voller Fragezeichen, die sofort durch das Terrain ersetzt wird, wenn die Daten beim Client ankommen (Abb. 3).

Mit diesem Mechanismus ist es möglich, beinahe beliebig große Terrains zu bespielen. Das Limit setzt einzig und allein die Speicherausstattung des Servers.

2.3 Architektur im Detail: Einheiten

Das Einheitensystem in ZoneWars ist auf Flexibilität ausgelegt: auch wenn im Spiel momentan nur eine Einheit zur Verfügung steht, wäre es ohne Weiteres möglich, Dutzende oder Hunderte weiterer Einheitentypen zu integrieren.

Technisch wird dies durch eine Basis-Einheitenklasse namens BaseUnit erreicht, die alle Funktionalität zur Verfügung stellt, die bei jeder Einheit ohnehin gleich ist: Verwaltung der Parameter wie Gesundheitsstatus, Energiestatus, Zugehörigkeit zu einem Spieler, aber auch Operationen wie das Zeichnen der Einheit in den Bildschirmpuffer des Clients oder serverseitige Operationen wie das Bewegen der Einheit auf dem Terrain.

Von dieser abstrakten Basisklasse wird für eine konkrete Einheit abgeleitet. Zusätzliche, einheitenspezifische Funktionalität kann dabei hinzugefügt werden, ebenso kann Basisfunktionalität, die in anderer Form als gegeben benötigt wird, überschrieben werden. Außerdem wird beim Ableiten der Einheit das Aussehen der Einheit auf dem Bildschirm festgelegt.

Zum Zeichnen einer Einheit auf dem Bildschirm werden 16 Einzelbilder benötigt:

- Acht Bilder der Einheit auf flachem Terrain, eines für jede mögliche Himmelsrichtung
- Acht weitere Bilder der Einheit auf Bergkanten, jeweils einmal nach oben und einmal nach unten gerichtet



Abb. 4: Alle Einheitengrafiken der momentan einzigsten existierenden Einheit

Für jedes dieser 16 Bilder ist noch ein weiteres nötig: eine Graustufigrafik, die als Maske für die Einfärbung der Einheiten in der Farbe des Spielers, dem sie gehören, dient. Die Maske ist nötig, weil man für die Einheit sonst nicht definieren könnte, welche Flächen eingefärbt werden sollen und welche nicht. Durch dieses System kann dynamisch eine Einheit jeder beliebigen Farbe erstellt werden.



Abb. 5: Compositing der finalen Einheitengrafik aus Basisgrafik plus Einfärbung mittels Maske

Das relativ einfache Modell der ersten Einheit (genannt „Zerstörer“, oder auf Englisch „destroyer“) habe ich in Rhinoceros - eine 3D-Modeling-Anwendung, die ähnlich wie eine CAD-Anwendung funktioniert und sich daher gut für das Modellieren technischer Gegenstände mit ihrem meist „harten“ Charakter eignet - erstellt und anschließend fürs Rendering in 3ds Max 9 exportiert (Abb. 6 & 7). Dort sind beim Rendering auch die Maskenbilder entstanden, was durch die Möglichkeit, einer Textur eine ID zuzuordnen und anschließend ein Bild zu rendern, in welchem die Flächen je nach der ID ihrer Textur eingefärbt sind, relativ einfach war.

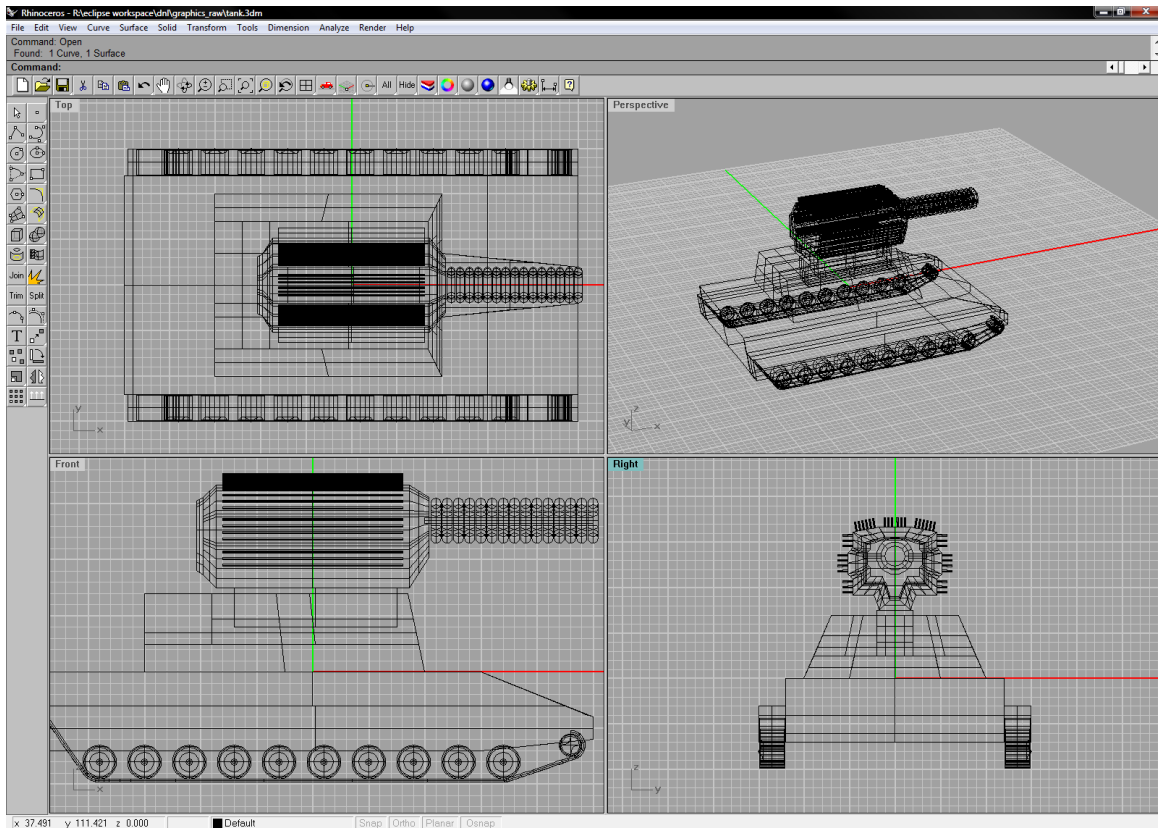


Abb. 6: Drahtgittermodell der Einheit in Rhinoceros

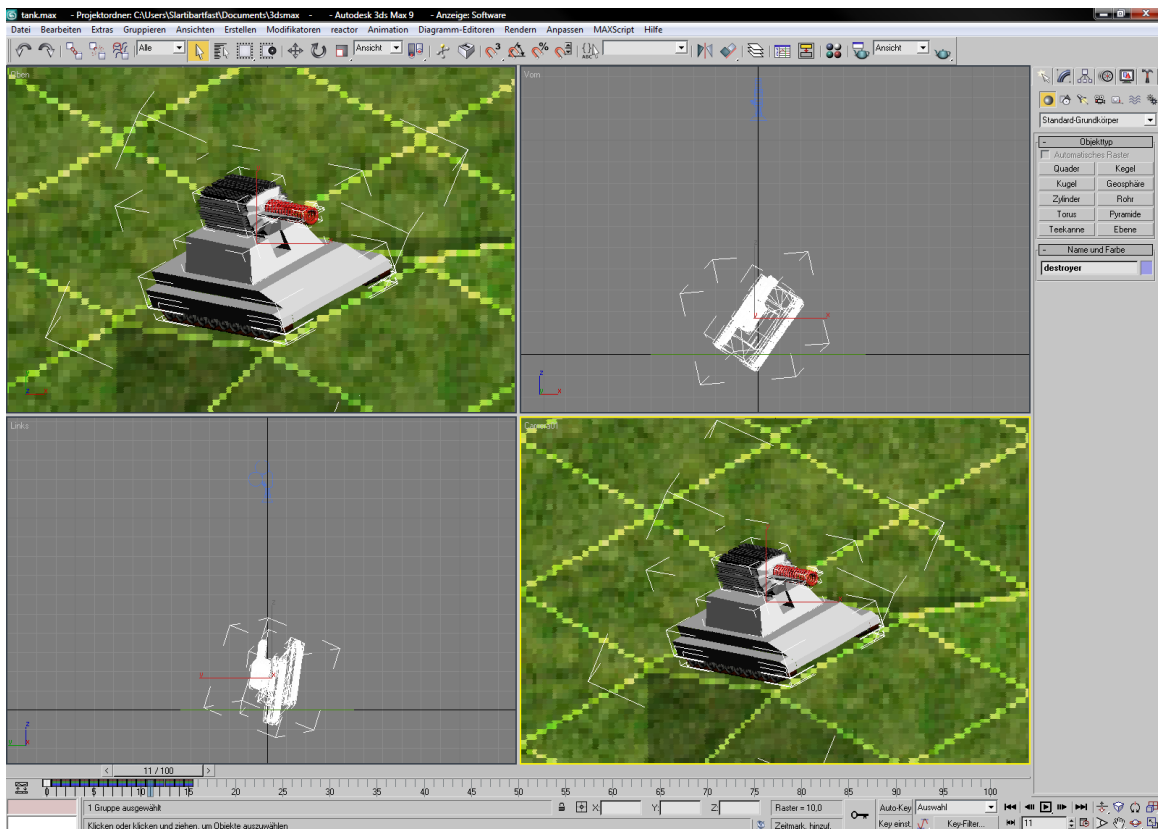


Abb. 7: Einheit beim Rendern in 3ds Max 9 - für jede Position wurde ein eigener Keyframe erstellt

2.4 Architektur im Detail: Effekte und Animationen

Während die Terraingrafik und Einheitengrafik in ZoneWars statisch ist, wird zur Darstellung von Kampfeffekten und der Anzahl zugefügter Schadenspunkte bei Attacken ein animierter Layer über den statischen Grafiklayer im Hintergrund gelegt. Da sich allerdings leider herausgestellt hat, dass im Browser nur unbeschleunigte Grafikoperationen zur Verfügung stehen, bedeutet jeder zusätzliche Effekt eine nicht unerhebliche Belastung für den Clientrechner. Bei der direkten Ausführung außerhalb eines Browsers könnte der ZoneWars-Client seine Grafikoperationen vollständig im Speicher der Grafikkarte durchführen - so wie das alle „normalen“ Spiele auch tun.

Animationen werden intern ähnlich wie Einheiten organisiert: es gibt eine abstrakte Basisklasse für Animationen, die grundlegende Funktionalität zum Zeichnen der Animationen auf den Bildschirm zur Verfügung stellt. Den Verweis auf die konkrete Animationsgrafik bringt die abgeleitete Klasse einer konkreten Animation mit.

Der Aufbau der Animations-Grafikfiles selbst ist „klassisch“: es handelt sich um normale Grafiken, in denen alle Einzelbilder der Animation hinter- und untereinander platziert sind. Beim Laden der Files wird festgelegt, wie viele Bilder sich in X- und Y-Richtung befinden, wie hoch die Framerate beim Darstellen sein soll und wie lange die Animation dargestellt werden soll, alles weitere erledigen die Zeichenroutinen der Basisklasse.



Abb. 8: Zwei Animationsgrafiken

2.5 Architektur im Detail: Lua-Integration

Die Wahl der Scriptsprache, in der die Einheitenscripts geschrieben werden, fiel auf Lua - eine speziell im Spielbereich sehr weit verbreitete Sprache, deren Compiler und Laufzeitumgebung in ANSI-C geschrieben und als Open-Source-Software unter einer auch für kommerzielle Zwecke sehr freigiebigen Lizenz (der „MIT license“) verfügbar sind. Die Gründe dafür sind einfach: Lua hat sich zu einer vergleichsweise einfach zu lernenden, schlanken und dennoch mächtigen Sprache entwickelt, die flott und ressourcensparend ausgeführt werden kann (Speziell die letzteren beiden Argumente sorgten für die Verbreitung im Spielbereich, in dem es auf nahezu Echtzeitfähigkeit und Sparsamkeit im Umgang mit CPU-Zyklen und RAM ankommt).

Die Einfachheit erreicht Lua durch eine vielen gängigen Sprachen ähnelnde Syntax und die Ausführung in einer virtuellen Maschine mit per Garbage Collection verwaltetem Heap-Speicher. Für die Geschwindigkeit sorgt ein schneller Compiler in Lua-Bytecode und ein performanter Bytecode-Interpreter. Wird noch höhere Geschwindigkeit benötigt, kann man zu LuaJIT greifen, einem JIT-Compiler für Lua, der die Ausführungsgeschwindigkeit nochmals je nach konkreter Anwendung bis um den Faktor 6 erhöhen kann.

Die Einbindung des Lua-Interpreters in ein eigenes Programm erfolgt über das direkte Ausführen von Lua-Sourcecode (über diesen Weg werden auch in Lua geschriebene Funktionen definiert, die nicht sofort ausgeführt werden - auszuführender Sourcecode wird stets automatisch in Lua-Bytecode kompiliert), das Aufrufen von Lua-Funktionen aus einer C-Umgebung heraus und das Einbinden von C-Funktionen in die Lua-Laufzeitumgebung zum Aufruf aus Lua-Code.

Nun ist der ZoneWars-Server, welcher die User-Scripts ausführen sollte, in Java geschrieben und läuft innerhalb einer Java-VM. Die einzige Möglichkeit zur Einbindung von Lua war folglich die Nutzung von JNI.

Um eine möglichst flexible Integration zu erhalten, die bei Bedarf auch einfach abseits des Projektkontexts von ZoneWars in einem beliebigen anderen Java-Projekt wiederverwertet werden könnte, wählte ich eine Integration mittels eines in C geschriebenen Wrappers, der die für die Integration wichtigen Lua-Funktionen in Java verfügbar macht. Jedoch konnte und wollte ich nicht alle relevanten Funktionen eins zu eins „durchreichen“ - zur einfacheren Verwendung der Lua-Funktionalitäten in Java wollte ich einen stärker objektorientierten Ansatz verfolgen (Lua selbst ist, wie schon gesagt, in purem C geschrieben und rein prozedural aufgebaut).

Das erste neue Objekt, das ich hierfür erstellte und einführte, ist das `LuaObject`. Lua arbeitet intern mit nur einem einzigen numerischen Datentyp, dem 64 Bit breiten IEEE-Fließkommatyp „double“, sowie einigen weiteren Datentypen für Strings, Tables (die Arrays der Lua-Welt), Funktionen und weitere Spezialzwecke. Darüberhinaus gibt es den wichtigen Sondertyp „nil“, der in etwa einem „null“ in Java entspricht, also einer „nirgendwohin“ zeigenden Referenz. Das `LuaObject` kapselt nun die drei wichtigsten dieser Typen - den numerischen, den String und nil. Eine solche Kapselung hat für die Arbeit auf Java-Seite mehrere Vorteile: zum Einen vereinfacht sich die Handhabung von Werten aus der Lua-Laufzeitumgebung, da der Container für den Wert unabhängig vom Typ des Werts derselbe bleibt, und zum Anderen erlaubt die Kapselung eine Vereinheitlichung mehrerer Funktionen. Lua ist eine dynamisch getypte Sprache; vor einem konkreten Aufruf einer Funktion (ob Lua-Funktion oder C-Funktion, ist egal) steht also nicht fest, welchen Typ die Parameter haben werden. Bei der Integration in eine statisch getypte Sprache wie C oder Java werden aufgrund dieser Tatsache eine große Menge von Funktionen, die an sich dasselbe tun - nur jeweils für einen anderen Datentyp - nötig. Ohne die Kapselung der verschiedenen Datentypen in ein Objekt müssten all diese Funktionen per JNI in Java sichtbar gemacht werden. Mit der Kapselung ist es nur noch eine Funktion, die ein `LuaObject` empfängt oder zurückgibt.

Das zweite Objekt ist der sogenannte `LuaStack`. Dies ist nichts weiter als ein „Bequemlichkeits-Objekt“ zur einfacheren Parameterübergabe beim Aufruf von Lua-Funktionen. Ein Lua-Funktionsaufruf aus C heraus folgt einem ähnlichen Prinzip wie der „nackte“ Funktionsaufruf in Maschinsprache: zunächst pusht man eine Referenz auf die aufzurufende Funktion auf den Stack der Lua-Laufzeitumgebung, anschließend werden der Reihe nach alle zu übergebenden Parameter gepusht. Zu guter Letzt wird der Laufzeitumgebung der Befehl zum Aufruf der Funktion gegeben. Dieser Push-Vorgang wird in Java durchgeführt, zur Vereinfachung der Parameterübergabe an die Funktion, die diesen Vorgang durchführt, existiert der `LuaStack`, der mehrere `LuaObjects` in einer stack-artigen Struktur kapselt.

Das dritte und letzte Objekt ist gleichzeitig das wichtigste: das `LuaEnvironment`. Dieses Objekt kapselt im Grunde die Funktionalität der gesamten Lua-Laufzeitumgebung. Möchte eine Java-Anwendung eine Lua-VM hochfahren und nutzen, so instanziiert sie zunächst das `LuaEnvironment` (oder eine Klasse, die von `LuaEnvironment` erbt) und teilt bereits im Konstruktor mit, wie groß die maximale Heap-Size der neuen Lua-VM sein soll. Bei der Instanzierung wird diese dann auch sofort per JNI hochgefahren und steht anschließend für Code-Aufrufe aus Java heraus bereit. Die dafür nötigen Funktionen stellt ebenfalls das `LuaEnvironment` zur Verfügung.

In den wenigsten Fällen wird man allerdings direkt eine Instanz von `LuaEnvironment` erzeugen - meist wird es eine von `LuaEnvironment` erbende Klasse sein. Über den Mechanismus der Vererbung werden nämlich Java-Funktionen, die innerhalb der Lua-Umgebung verfügbar sein sollen, eingebracht: man schreibt eine neue Klasse, die von `LuaEnvironment` erbt, und realisiert in ihr alle nach Lua zu exportierenden Funktionen in Form von normalen Methoden. Anschließend überschreibt bzw. erweitert man den Konstruktor dieser Klasse und ruft in ihm die Methode `registerJavaFunction` der Klasse `LuaEnvironment` auf, welche dazu dient, Java-Funktionen (bzw. Methoden) in Lua verfügbar zu machen. Dabei werden zwei Parameter übergeben: zum Einen den Namen der Methode in Java, zum Anderen den gewünschten Namen, den diese Funktion in Lua tragen soll.

Den ersten Parameter merkt sich das `LuaEnvironment` selbst in einer `ArrayList` und vergibt dadurch eine aufsteigende Nummer an diese konkrete Funktion. Anschließend wird eine Funktion des C-Wrappers aufgerufen und der gewünschte Name der Funktion in der Lua-Umgebung übergeben. Die Wrapperfunktion tut nun folgendes:

- Zunächst wird eine Liste von Pointern auf Speicherpositionen vergrößert, um einen weiteren Pointer für die neue Funktion aufzunehmen.
- Anschließend werden 56 Bytes an Speicher per `malloc` angefordert
- In diese 56 Bytes werden nun die Opcodes einer C-Funktion kopiert, die folgendem entsprechen:

```
int jfs_0(lua_State *L)      { return callJavaFunction(0); }
```

`callJavaFunction` ist dabei eine Funktion, die nichts weiter tut als eine entsprechende Funktion auf Java-Seite per JNI aufzurufen und den numerischen Parameter (hier 0) zu übergeben.
- Die Binärcodes werden anschließend noch an zwei Stellen gepatched:
 - Der numerische Parameter wird durch eine aufsteigende Nummer (dieselbe, die zuvor in Java für die Java-Funktion vergeben worden ist) ersetzt
 - Die Adresse nach der eigentlichen `call`-Instruktion (also die Zieladresse, oder besser gesagt: der Zieloffset) wird durch einen berechneten Offset ersetzt, damit stets die korrekte Adresse der Funktion `callJavaFunction` angesprungen wird.
- Zu guter Letzt wird ein Pointer auf diese neu erstellte Funktion bei Lua unter dem gewünschten Funktionsnamen registriert.

Diese ganze Mühe ist nötig, weil Lua leider nur die Möglichkeit bietet, eine feste C-Funktion in der Laufzeitumgebung zu registrieren. Um nun von Java-Seite beliebig viele Funktionen registrieren zu können, bleibt daher nur der Umweg über das Erstellen einer Dummy-C-Funktion für jede Java-Funktion, die nichts weiter tut als in Java den Aufruf der „eigentlichen“ Funktion per Reflection anzustoßen. Zur Unterscheidung der Funktionen wird der numerische Parameter herangezogen, anhand dessen auf Java-Seite eindeutig der Name der aufzurufenden Java-Funktion ermittelt werden kann.

Die Parameterübergabe zwischen Java (bzw. C) und Lua läuft über den Lua-Stack, für den `push`- und `pop`-Funktionen per JNI in Java verfügbar gemacht worden sind. Nach dem Aufruf einer Java-Funktion aus Lua heraus kann diese also darüber ihre Parameter empfangen und am Ende ihrer Ausführung die Ergebnisparameter (in Lua kann eine Funktion beliebig viele Parameter empfangen und zurückgeben!) wieder auf den Stack pushen. Wirklich klassisch „zurückgeben“ wird von der Funktion nur die Anzahl Rückgabeparameter auf dem Stack.

Etwas Aufmerksamkeit verdient auch die Möglichkeit, beliebige globale Lua-Variablen als „persistent“ zu registrieren, was bedeutet, dass ihr Inhalt beim Beenden der Lua-VM gespeichert und bei ihrem nächsten Start in der nächsten Spielrunde (für diesen Spieler) wiederhergestellt wird.

Diese Funktionalität ist sowohl mittels Java- als auch Lua-Code realisiert. Zum Einen gibt es eine Java-Funktion `registerPersistentVariable`, welche man mit dem Variablennamen als Parameter aufruft und über welche eine Variable als „zu persistieren“ markiert wird. Diese Liste der zu persistierenden Variablen wird in Java verwaltet. Wird nun die Lua-VM heruntergefahren, wird vor dem „Abschalten“ ein Stück Java-Code aktiv, welcher der Reihe nach alle zu persistierenden Variablen nach ihrem Wert fragt und diesen speichert. Diese Abfrage passiert allerdings nicht allein in Java, hier kommt nun die folgende kleine Lua-Komponente ins Spiel.

```
function simulator_export_variable(variable, varName)
    local var=variable;
    if(variable==nil and varName~=nil) then
        var=getglobal(varName);
    end
    if(var==nil) then return nil; end
    local result=varName.." = ";
    local varstring=simulator_export_variable2(var, 0);
    if(varstring==nil) then return nil; end
    return result..varstring.." ";
end

function simulator_export_variable2(var, indentation)
    if(type(var)=="nil") then
        return "nil";
    end
    if(type(var)=="number") then
        return var;
    end
    if(type(var)=="string") then
        return "\""..string.gsub(var, "\",", "\\\"").."\"";
    end
    if(type(var)=="boolean") then
        if(var==true) then
            return "true";
        else
            return "false";
        end
    end
    if(type(var)=="table") then
        local result="{\n";
        local k,v;
        for k,v in pairs(var) do
            result=result..string.rep("\t",indentation+1)..
                "["..simulator_export_variable2(k,indentation+1).."] = "
                ..simulator_export_variable2(v,indentation+1)..",\n";
        end
        result=result..string.rep("\t",indentation).."}";
        return result;
    end
end
```

Die Funktion `simulator_export_variable` wird von Java aus aufgerufen. Ihr wird der Name einer zu persistierenden Variable übergeben. Sie geht zunächst hin und holt sich statt dem String mit dem Variablennamen die echte Variable über die Funktion `getglobal` (dies ist eine aus Java exportierte Funktion, da nur außerhalb von Lua ein Variablenname in die Variable selbst aufgelöst werden kann). Anschließend beginnt sie, einen String zu konstruieren, der bei Ausführung dieselbe Variable deklarieren und mit dem richtigen Wert befüllen würde. Das geschieht mit der Hilfe der zweiten Funktion, die sich im Falle von Lua-Tables auch rekursiv aufrufen kann (Tables können wiederum Tables enthalten usw.). Entstanden wird also am Ende ein String nach folgendem Schema:

```
persistentVariable1 = "ich bin persistent!";

persistentVariable2 = 12345;

persistentTable = {
    [1] = "Hallo Welt",
    [2] = "ich bin das zweite Element",
    ["abc"] = "ich bin ein assoziatives Element mit einem String als Schlüssel",
    ["table2"] = {
        [1] = "ich bin eine Table innerhalb einer Table",
    },
};
```

Dieser String wird in Java für den Spieler gespeichert und beim nächsten Hochfahren der Lua-VM vor der Ausführung des Einheitenscripts „ausgeführt“, wodurch die Variablen wieder ihren gespeicherten Wert erhalten.

Der Lua-Code, der für das Exportieren der persistenten Variablen zuständig ist, wird ebenfalls bei jedem Hochfahren der VM direkt kompiliert.

2.6 Architektur im Detail: Einheiten-API

Die API, die im Einheitscript verwendet werden kann, um mit den Einheiten zu interagieren, gehört zum Einen zu den zentralen Elementen im Spiel, zum Anderen aber auch zu denen, die noch ausbaufähig und ausbaubedürftig wären. Was allerdings konkret ausgebaut werden müsste, würde sich vermutlich erst wirklich in einem Praxistest mit mehreren Spielern herausstellen, weil es davon abhängt, was die Spieler an Werkzeugen benötigen, um ihre script-technischen Wünsche umsetzen zu können.

Daher ist die momentane API vom Umfang her noch recht klein und bietet nur die Basics. Im Detail sind es folgende Funktionen:

```
xcoord, ycoord = getUnitPosition(unitID)
```

Hiermit kann die Position einer Einheit über ihre UnitID (eine eindeutige Identifikationsnummer, die jede Einheit mit sich trägt) abgefragt werden.

```
orientation = getUnitOrientation(unitID)
```

Diese Funktion liefert die „Blickrichtung“ einer Einheit zurück (in Form eines Strings, entweder „north“, „northwest“, „northeast“, „west“, „east“, „south“, „southwest“ oder „southeast“).

```
targetingInfo = getUnitTargetingInfo(unitID, targetUnitID)
```

Die targetingInfo ist ein String, der besagt, was passieren würde, wenn die erste Einheit die zweite Einheit angreifen würde. Mögliche Antworten sind „attackable“ (die zweite Einheit würde getroffen), „out_of_range“ (die Einheiten sind zu weit auseinander), „out_of_angle“ (die zweite Einheit befindet sich außerhalb des Schusswinkels von 45° in beide Richtungen vor der ersten Einheit) oder „out_of_range_and_angle“ (sowohl außer Reichweite als auch außerhalb des Schusswinkels).

```
health = getUnitHealth(unitID)
```

Hierüber kann die Gesundheit einer Einheit abgefragt werden.

```
health = getUnitPower(unitID)
```

Dies tut dasselbe wie getUnitHealth, gibt aber die Energie der Einheit zurück

```
unitlist = getUnitList()
```

getUnitList gibt eine Liste (in Form einer Lua-Table mit unitIDs) aller aktiven eigenen Einheiten zurück.

```
unitlist = scanForUnits(unitID)
```

Diese Funktion löst einen Scan bei der angegebenen Einheit aus. Ein Scan findet alle anderen Einheiten im Sichtbereich der Einheit und gibt deren unitIDs zurück.

```
unitlist = scanForFriendlyUnits(unitID)
```

Dies tut im Grunde dasselbe wie scanForUnits, gibt aber lediglich die eigenen Einheiten zurück.

```
unitlist = scanForEnemyUnits(unitID)
```

Hiermit werden nur gegnerische Einheiten im Sichtbereich der angegebenen Einheit zurückgegeben.

```
height, isPassable, type = getTerrainInfo(xcoord, ycoord)
```

Mit dieser Funktion können Informationen über ein bestimmtes Feld des Terrains angefordert werden. Die Rückgabewerte bestehen aus der Höhe (0-15, wobei 0 Meeresspiegel ist und automatisch „Wasserfläche“ bedeutet), einer Information, ob das Terrain von Einheiten befahrbar ist oder nicht, und einem String, der den genauen Tile-Typ bestimmt („flat“, „mountain_edge_west“, „mountain_edge_east“, „mountain_edge_south“, „mountain_edge_north“, „mountain_flat_southwest“, „mountain_flat_southeast“, „mountain_flat_northwest“, „mountain_flat_northeast“, „mountain_inner_edge_west“, „mountain_inner_edge_east“, „mountain_inner_edge_south“, „mountain_inner_edge_north“, „mountain_edge_westeast“, „mountain_edge_northsouth“)

Diese Funktionen können in den Einheitscripts genutzt werden. Die Scripts selbst folgen einem vorgegebenen Schema:

```
function prepare()  
    -- hier kann Code eingefügt werden, welcher einmalig zu Beginn der Codeausführung  
    -- für einen Spieler ausgeführt werden soll  
end  
  
function processUnit(unitID)  
    -- hier steht der eigentliche Einheiten-Code, er wird für jede lebende Einheit  
    -- einmal pro Spielrunde ausgeführt. unitID enthält jeweils die ID der Einheit.  
end  
  
function postprocess()  
    -- dieser Code wird einmalig zum Schluss jeder Spielrunde ausgeführt, nachdem  
    -- processUnit für jede Einheit aufgerufen worden ist  
end
```

Welche Aktion die Einheit ausführen soll, wird durch den Rückgabewert von `processUnit` festgelegt.

- „forward“ lässt die Einheit einen Schritt nach vorne machen
- „turnleft“ löst eine Drehung nach links aus
- „turnright“ dreht die Einheit nach rechts
- „attack“ löst einen Angriff aus. Das Ziel des Angriffs bestimmt der zweite Parameter (hier wird die unitID des gewünschten Ziels zurückgegeben), und die Art des Angriffs der dritte („base“ für einen normalen Schuss mit der Standardwaffe, „wave“ für die Area-Effect-Waffe, die gleichzeitig alle Einheiten um die auslösende Einheit herum trifft (für diese Waffe ist allerdings eine volle Energieaufladung nötig)).

Die Aktionen werden dabei aber nicht unmittelbar nach der Rückgabe des Befehls aus `processUnit` ausgeführt. Stattdessen werden sie zunächst gespeichert, bis die Scripts aller Spieler in der laufenden Spielrunde ausgeführt worden sind, so dass alle Aktionen für alle Einheiten feststehen. Anschließend werden die Befehle aller Spieler in einer zufällig bestimmten Reihenfolge nacheinander ausgeführt, wobei mittlerweile ungültig gewordene Befehle (etwa Angriffe, deren Ziel bereits tot ist, oder Einheitenbewegungen auf ein Feld, welches mittlerweile von einer anderen Einheit besetzt ist) ignoriert werden. Auf diese Weise herrscht sowohl bei der Ausführung der Einheitscripts als auch bei der Ausführung der Befehle eine gewisse Gerechtigkeit, da alle Scripts in einer Spielrunde dieselbe Situation vorfinden und kein Spieler durch permanent frühe Ausführung der eigenen Befehle bevorzugt wird.

3.0 Fazit

ZoneWars war ein interessantes Projekt. Speziell die Lua-Integration in Java, die sich als schwieriger herausgestellt hat als ich erwartet hatte, aber auch der Aufbau eines flexiblen Einheiten- und Kampfsystems war lehrreich. Einige Entscheidungen erwiesen sich im Nachhinein leider als falsch bzw. schlecht, z.B. die Entscheidung für das Koordinatensystem. Allerdings kam diese Erkenntnis erst an einem Punkt, als bereits zu viel Code für die bestehenden Verhältnisse geschrieben war, wodurch eine Änderung zu aufwendig geworden wäre. Rückblickend auf mein erstes Softwareentwicklungsprojekt und auf andere Projekte, die ich privat angefangen habe, ist diese Problematik allerdings nichts Neues und scheint in jedem Projekt mehr oder weniger häufig aufzutreten.

Was leider auch etwas enttäuschend war: ich bin leider nicht dazu gekommen, die geplanten Ressourcen, die man mit speziellen Einheiten „ernten“ und in die Produktion von neuen Einheiten investieren kann, umzusetzen. Diese wären allerdings dringend erforderlich, um das Spiel tatsächlich spielen zu können, denn im momentanen Stadium besteht die einzige Möglichkeit, neue Einheiten zu erschaffen, darin, diese über Admin-Kommandos einfach zu erzeugen.

Nichtsdestotrotz konnte ich mit ZoneWars das Ziel, durch praktische Arbeit etwas zu lernen, erreichen, und insofern hat sich das Projekt in jedem Fall gelohnt, auch wenn noch kein für den „produktiven Betrieb“ fertiges Endprodukt dabei entstanden ist.