Mother Of All Gravity Games

Bernhard Trummer
Linux User Group Graz
für die Linuxtage04
bernhard.trummer@gmx.at

8. Mai 2004

Übersicht

Mother Of All Gravity Games: Einblick in die Geschichte und in die Zukunft des Spiels.

Aufbau der Levels: Dieser Abschnitt wird grob erklären, wie die Levels für das Spiel aufgebaut sind.

Architektur von MoagG: Hier soll die grundlegende Strukturierung des Source Codes erklärt werden.

Die Game-Engine: Den Abschluß macht die Erklärung einiger interner Abläufe, die anhand des Source Codes verdeutlicht werden.

Mother Of All Gravity Games

http://moagg.sourceforge.net/

MoagG is a 2D gravity game, combining the idea and fun of several existing games, such as Space Taxi (C64) and Gravity Force (Amiga).

Bernhard Trummer: Project Maintainer und Core Development.

Andreas Granig: GUI/Core Development und Web Site.

Kevin Krammer: Qt Map Editor.

Geschichte 1/2

April 2003: Erste Brainstorming- und Planungs-Session. Das dabei herausgekommene „Pflichtenheft“ ist auch heute noch bei MoagG unter `doc/moagg.tex` dabei.

Mai 2003: Beginn der Entwicklung.

11.07.2003: Import des Source Codes auf SourceForge.

Oktober 2003: Beginn der Entwicklung von moaggedit, dem Qt-basierten Map Editor inkl. Import auf SourceForge.

Geschichte 2/2

08.12.2003: Erste Release von MoagG, welche im Vergleich zur aktuellen Version zwar noch sehr unvollständig, aber im Prinzip schon spielbar war.

09.12.2003: Erste Release von moaggedit.

12.04.2004: Release von der aktuellen Version MoagG 0.8.

27.04.2004: (Längst überfällige) Release von moaggedit 0.3.

Roadmap für die Zukunft

- Mehr Kachel-Grafiken für abwechslungsreichere Spielfelder.
- Mehr Levels in unterschiedlichen Schwierigkeitsgraden.
- Zusätzliche Single-Player Spieltypen.
- Fertigstellung aller offenen Punkte für die Release 1.0.
- Implementierung der Multiplayer-Fähigkeit über das Netzwerk.
- Erweiterung mit zusätzlichen Multiplayer-Spieltypen.

Warum SDL?

- Ursprünglich standen ClanLib, Java2D und SDL zur Auswahl. Für alle drei wurde ein Testprogramm geschrieben, um die generelle Benutzbarkeit und die Performance auszuloten.
- Obwohl die ClanLib zuerst sehr interessant ausgesehen hat, schied sie letztendlich aus, da intern alle Blit-Operationen auf OpenGL abgebildet werden.
- Java2D ist zwar ein *sehr* umfangreiches Framework, konnte aber von der Performance her nicht überzeugen.
- Daher fiel die Entscheidung letztendlich auf die SDL.

Aufbau der Levels

- Jedes Level setzt sich aus einem Map- und einem Level-File zusammen.
- Die Spezifikation beider Formate liegt dem MoagG-Source bei und kann unter `doc/level.tex` eingesehen werden.
- Das Map-File enthält ausschließlich den Aufbau des Spielfelds.
- Die Objekte (Plattformen, Geschütztürme, etc.), sowie der Typ des Levels sind im Level-File gespeichert.
- Auf diese Weise ist es möglich, ein und die selbe Map in unterschiedlichen Levels zu verwenden.

Format der Map-Files

- Map-Files werden als Binärfiles gespeichert.
- Eine Map besteht intern aus einer Größenangabe und aus mindestens einem Layer.
- Jeder Layer ist in Kacheln aufgeteilt, die 16x16 Pixel groß sind.
- Die verfügbaren Kachel-Grafiken sind in unterschiedliche Kategorien zusammengefaßt (Ziegelsteine, Beton, etc.).
- In einem Layer können nur Kacheln aus einer Kategorie verwendet werden.

Format der Level-Files

Level-Files werden als XML-Files gespeichert, in denen u.a. folgende Informationen enthalten sind:

- Das Map-File für dieses Level.
- Globalen Einstellungen, wie z.B. die Gravitation und Reibung.
- Aufbau des Spielfelds (Plattformen, Hindernisse, etc.)
- Informationen für den Spielablauf (z.B. Startposition, Zielplattform).

Beispiel tutorial01.xml

```
1  <level type="race">
2    <playground map="tutorial01.map">
3      <platform number="0" x="3" y="24" w="10"/>
4      <platform number="1" x="26" y="24" w="10"
5                left="yellow" right="yellow"/>
6    </playground>
7
8    <gamecontrol>
9      <startposition platform="0" fuel="20"/>
10     <platform number="1"/>
11   </gamecontrol>
12 </level>
```

Architektur von MoagG

Der Source Code von MoagG ist in Module aufgeteilt, die in Bibliotheken zusammengefaßt sind. Diese Strukturierung wurde bewußt so gewählt, damit man als Entwickler gezwungen ist, sich vernünftige Schnittstellen zwischen den Modulen zu überlegen und diese auch zu verwenden.

Module 1/2

libMoaggCore: Diese Library enthält Hilfsklassen, um das Arbeiten mit Dateien/Verzeichnissen, mit SDL-Funktionen und mit XML-Strukturen zu vereinfachen.

libMoaggSound: Diese Library enthält alles was für die Wiedergabe von Sound und Musik benötigt wird. Dazu gehört eine `JukeBox` und eine `SoundFactory`, von der Musikstücke (im mod-Format) bzw. Soundeffekte abgerufen werden können.

libMoaggObjects: Diese Library enthält alle Objekte (Schiffe, Plattformen, Magnete, etc.) die im Spiel vorkommen. Ausgehend von einer gemeinsamen Basisklasse `ObjectBase` ist hier eine Objekt-Hierarchie aufgebaut, um diverse Gemeinsamkeiten von Objekten zusammenzufassen.

Module 2/2

libMoaggGame: Die wesentlichen Bestandteile dieser Library sind das Spielfeld (`PlayGround`) und die State-Machines für die unterschiedlichen Spieltypen, die auf der Klasse `GameControlBase` basieren.

libMoaggGui: Diese Library enthält die grafische Oberfläche basierend auf der libparagui. Ferner ist hier die Hauptschleife des Spiels enthalten.

moagg: Enthält das Hauptprogramm (`int main()`) und kümmert sich um die Initialisierung bzw. Shutdown.

testsuite: Enthält *cppunit* Modul-Tests einiger wichtiger Komponenten.

Die Game-Engine

Seit die libparagui für die grafische Oberfläche verwendet wird, ist es nicht mehr möglich, das Event-System der SDL direkt zu verwenden. Um die Keyboardabfragen für die Steuerung des Schiffs trotzdem durchführen zu können, wurde daher das Wrapper-Widget `PlayGroundMenu` eingeführt, über das auch die Hauptschleife der Game-Engine läuft.

Die eigentlichen Aktionen der Hauptschleife passieren in der Methode `PlayGround::update()`. Jeder Schritt wird hier über einen entsprechenden Visitor durchgeführt, der eine Liste aller Objekte durchläuft.

Aktionen der Hauptschleife

- Überzeichne alle Objekte mit dem ursprünglichen Inhalt des Spielfelds.
- Aktualisiere alle Objekte.
- Überprüfe Kollisionen zwischen allen Objekten.
- Entferne alle Objekte, die nicht mehr benötigt werden. Dazu gehören Objekte, die in eine Kollisionen verwickelt waren oder dessen Lebenszeit beendet ist (z.B. Partikel).
- Zeichne alle verbliebenen Objekte neu.

Interne Darstellung des Displays

- Die Map des Levels wird in eine `SDL_Surface` eingelesen und bleibt danach als *Map Surface* unverändert gespeichert.
- Eine zweite gleich große `SDL_Surface`, die *Shadow Surface*, wird mit der *Map Surface* initialisiert.
- Objekte werden nur auf die *Shadow Surface* gezeichnet.
- Aus der *Shadow Surface* wird ein rechteckiger Bereich (*View Box*) auf der Display Surface dargestellt. Ein Verschieben der *View Box* entspricht dann einem Scrolling.

Darstellung des Displays

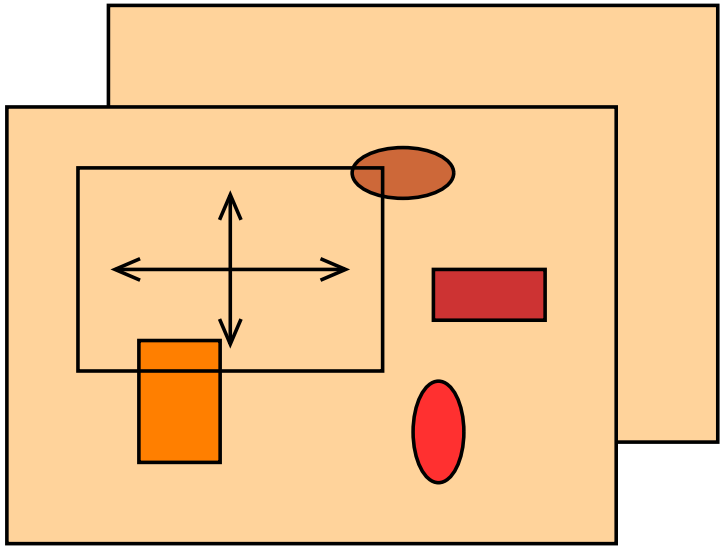

Abbildung 1: Map Surface, Shadow Surface und View Box

Aktualisierung des Displays

Ist kein Scrolling notwendig, werden nur die Änderungen der Objekte in Form einer Liste von `SDL_Rects` auf das Display Surface übernommen. Auf diese Weise muß für jedes Frame nur ein relativ geringer Teil des Displays aktualisiert werden, was zu einer sehr geringen CPU-Auslastung führt.

Muß für das nächste Frame gescrollt werden, so muß aus der *Shadow Surface* der gesamte Bereich der *View Box* auf das Display übernommen werden. Um die CPU-Auslastung zu minimieren, findet allerdings nur bei jedem zweiten Frame ein Update auf das Display statt.

Implementierung des Spielablaufs

Zur Zeit existieren die beiden Spieltypen `RaceGame` und `RescueGame`. Beide basieren auf einer `GameControlBase` Basisklasse, welche intern eine State-Machine für den Spielablauf besitzt.

Die API beinhaltet in erster Linie Aktionen, die sich aus dem Spielverlauf ergeben. Dazu gehören:

- Landung des Schiffs auf eine Plattform.
- Abheben des Schiffs von einer Plattform.
- Kollisionen zwischen zwei Objekten bzw. einem Objekt mit dem Spielfeld.

Ausgewählte Codestellen 1/3

ObjectBase.h: Die Basisklasse aller Objekte, die im Spiel vorkommen.

ShipSurfaces::init(): Einmalige Initialisierung der rotierten `SDL_Surfaces`, die dann von der `Ship` Klasse verwendet werden.

PlayGround::update(): Hier werden die vorher angeführten Aktionen der Hauptschleife durchgeführt.

Ship::X::update(): Code für die Aktualisierung des Raumschiffs, wenn es sich im *Landed* bzw. *Flying* State befindet.

Ausgewählte Codestellen 2/3

ControllableObjectBase::updateVelocityAndPosition(): Das physikalische Modell für die Aktualisierung der Geschwindigkeit und der Position für Raumschiffe und Raketen.

PlayGround::do_collidesWithPlayGround(): Eine im Vergleich zu `SDL_TOOLS::isCollision()` optimalere Implementierung für die Kollisionsabfrage zwischen einem Objekt und dem Spielfeld.

GameControlBase::onCollision(): Aktionen, die bei Kollisionen zwischen zwei Objekten bzw. zwischen Objekt und Spielfeld passieren sollen.

RaceGame::Running::onShipLanded(): Aktionen, die beim Landen des Raumschiffs ausgelöst werden.

Ausgewählte Codestellen 3/3

GameInterface.h: Viele Objekte müssen immer wieder Eigenschaften des Spielfelds abfragen bzw. auch verändern. Da allerdings per Design kein direkter Zugriff von Objekten auf das Spielfeld möglich ist, existiert das `GameInterface`, über das die gewünschten Aktionen durchgeführt werden müssen.

SoundInterface.h: Während des Spiels muß von mehreren Stellen her immer wieder dafür gesorgt werden, daß Soundeffekte bzw. Musik abgespielt wird. Das `SoundInterface` sorgt hier dafür, daß die internen Abläufe in der libMoaggSound nach außen hin versteckt werden.

LevelTest.cpp: Stellt sicher, daß alle Levels syntaktisch korrekt sind und geladen werden können.

Call for Volunteers

Was ich für MoagG noch relativ dringend benötige, sind:

- Zusätzliche Kachel-Grafiken für Maps.
- Viele zusätzliche Levels.
- Cliparts (Bäume, Gebäude, etc.), ev. in Form von Kacheln.
- Alternative Grafiken für Schiffe, Raketen, Hindernisse, etc.
- Soundtracks im mod-Format.