



Hochschule Karlsruhe
Technik und Wirtschaft
UNIVERSITY OF APPLIED SCIENCES

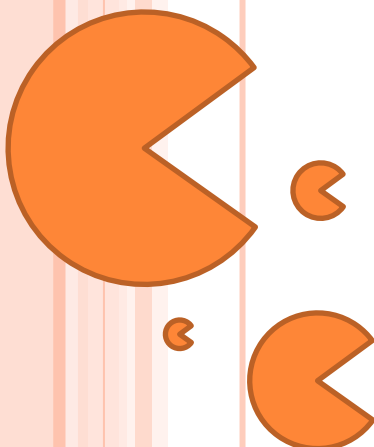
Projektarbeit

Tobias Fritz



Computerspiele in **Java**[™]

Eine Einführung in die 2D
Spielprogrammierung mit Java und Slick.



Inhaltsverzeichnis

Inhaltsverzeichnis	1
Modulbeschreibung	2
Übersicht	3
Aufbau	3
Slick einrichten	4
Manuelles Einrichten	8
Slick Konzept	11
Das Spielkonzept	16
Ein neues Projekt anlegen	16
The first Contact: Ein UFO kommt ins Spiel	17
Mit dem Raumschiff auf Abfangkurs	21
Der Sound macht's aus	24
Ein Text aus Bildern	26
Ein Partikel Feuerwerk	28
Der Code	30
Das Spiel exportieren	36
Spiel Ressourcen	40
Quellen	41

Modulbeschreibung

Du kennst dich mit Java aus hast aber im Studium bisher damit nur langweilige Musteraufgaben programmiert? Du liebst Computerspiele, hast auch eigene Ideen, nur gibt dir niemand die Handwerkzeuge zum Spiele programmieren in die Hand? In diesem Kurs erfährst du wie du mit Hilfe der Slick Engine schnell und einfach 2D Java Spiele programmierst. Diese und weitere Themen erwarten Dich:

- Bilder laden, darstellen und animieren
- Sounds abspielen
- Joystick und Gamepad Ansteuerung
- Effekte mit dem Partikel Editor

Ziel:

Ziel des Lernmoduls ist es dem Teilnehmer in die Lage zu versetzen eigene 2D Java Spiele zu entwickeln. Dazu wird nach einer kurzen Einführung in das Slick Framework, schrittweise ein 2D Arcade Spiel entwickelt. Die Kapitel bauen auf einander auf und werden durch Beschreibungen und Arbeitsanweisungen vorangetrieben. Die Aufgaben sollen zum Ausprobieren motivieren, sind für Folgekapitel aber keine Voraussetzung. Mit Hilfe des iterativen Vorgehens an einem Beispiel, soll neben dem Fachwissen, vor allem die Herangehensweise bei der Spieleprogrammierung näher gebracht werden.

Unterlagen:

Neben diesem Lernmodul sind noch eine Linksammlung und ein Wiki für detaillierte Beschreibungen vorhanden. Wissen dass den Arbeitsfluss stören würde, weil es nur für wenige Leser interessant ist, wird im Wiki ausführlich erläutert.

Der Quellcode ist bis auf standardisierte Methoden Bezeichner¹ in Deutsch verfasst. Das soll den Einstieg für Programmierneulinge erleichtern und hat zugleich den Effekt dass sich eigener Code von Bibliotheksaufrufen gut unterscheidet. *JavaDoc*² Kommentare sind auf ein Minimum reduziert, sobald die Funktion im Text beschrieben wird.

Voraussetzungen:

Für die Übung werden Grundkenntnisse mit Java vorausgesetzt. Vererbung, „For Each“ Schleifen und Exception Handling sollten dir bekannt sein.

Entwicklungsumgebung:

Der Kurs ist für die Entwicklung mit der freien IDE Eclipse³ ausgelegt. Eine andere IDE zu verwenden ist möglich, erschwert aber das Nachvollziehen der Beispiele. NetBeans Nutzer lesen dazu bitte das Kapitel „Beispielprojekt Einrichten mit NetBeans“ im Wiki.

¹ Wie Getter und Setter (`getX()`, `setX()`).

² Quelltext Dokumentation die direkt im Source Code oder als extra HTML-Dateien ausgeliefert wird.

³ Eclipse IDE for Java Developers: <http://eclipse.org>. Auf die Installation eines Sprachpakets wurde verzichtet.

Betriebssystem:

Der Kurs kann gleichermaßen auf Windows, Linux oder Mac durchgeführt werden. Die Beispiele und Screenshots zeigen jedoch stets das Vorgehen unter Windows 7. Weicht eine Beschreibung dem Vorgehen bei Mac oder Linux stark ab, wird dazu an der passenden Stelle ein Hinweis gegeben.

Systemvoraussetzungen:

- Installiertes Java JDK (Java Development Kit) 1.6.
Das JRE (Java Runtime Environment) würde zum Programmieren und Starten der Spiele ebenfalls reichen, ist wegen der fehlenden *JavaDoc* aber nicht zu empfehlen.
- Aktuelle OpenGL fähige Treiber für die Grafikkarte (ATI oder Nvidia).
Achtung: Die von Windows 7 mitgebrachten Standarttreiber reichen nicht.

Übersicht

Slick2D ist eine Sammlung von Tools und Erweiterungen der *LWJGL* (Lightweight Java Game Library). Slick wird in der Community unter der BSD Lizenz mit folgenden Zielen weiterentwickelt:⁴

- Eine schlanke 2D API bereit zu stellen.
- Tools für viele einfache Spiele „Out of the Box“ mit zu liefern.
- Den Übergang von Java2D zu OpenGL zu erleichtern.
- Eine komfortable Veröffentlichung via Webstart zu ermöglichen.
- Erweiterbar und flexibel zu sein.
- Mix and Match – nutze was du willst; nichts ist vorgegeben.
- Hilfe beim Rendern, Sounds, dem Import, der Kollisionserkennung und vielem mehr.

Was die Slick Community explizit nicht möchte:

- Den Entwicklungsprozess vor zu geben.
- Eine 3D API zu integrieren.
- Fehlerfrei zu sein –das wäre unmöglich.

Aufbau

In Abbildung 1 ist der schematische Aufbau von Slick und den darunter liegenden Frameworks dargestellt. Java bringt von Haus aus nur Plattform übergreifende Funktionen mit. Hardware nahe Methoden, wie beispielsweise das Auslesen eines Gamepads, oder das direkte Ansteuern der Grafikkarte, sind mit dieser Standardbibliothek nicht möglich. In diese Bresche springt *LWJGL*, und bietet Schnittstellen zu *OpenGL*⁵ für die Grafik, *OpenAL*⁶ für den Sound und eine eigene Bibliothek zur Ansteuerung von Gamecontrollern.

⁴ Möglichst nahe Übersetzung der Ziele im Slick Wiki.

⁵ OpenGL (**O**pen **G**raphics **L**ibrary): <http://www.opengl.org>

⁶ OpenAL (**O**pen **A**udio **L**ibrary): <http://connect.creativelabs.com/openal/default.aspx>

Intern nutzt *LWJGL* Betriebssystem abhängige Bibliotheken, die sogenannten *native libraries*. Glücklicherweise bringt *LWJGL* natives für die gängigen Systeme (Linux, Mac und Windows) mit und wählt diese auch automatisch aus. Bei einer Desktop Applikation bzw. einem Spiel, das *LWJGL* nutzt, muss daher für den Start auf einem anderen Betriebssystem keine einzige Zeile Code geändert werden.

LWJGL stellt zwar die passenden Techniken für die Spieleprogrammierung in Java zur Verfügung, erhebt aber nicht den Anspruch, das Programmieren besonders einfach zu gestalten. Das übernimmt Slick und bietet eine Abstraktion zu *LWJGL*, um die Programmierung von 2D Computerspielen zu vereinfachen.



Abbildung 1 Aufbau des Slick Frameworks

Slick einrichten

Das Slick Framework besteht aus der Slick API, davon abhängige Bibliotheken, den Native Bibliotheken und einer Reihe von Java Tools. Um einen besonders leichten Einstieg zu gewährleisten, wurde für diesen Kurs ein vorkonfiguriertes Eclipse Projekt erstellt.

Einstiegsprojekt Öffnen

Das Einstiegsprojekt kann hier herunter geladen werden. Nach dem Download braucht es nicht entpackt zu werden. Um das Projekt in einen bestehenden Eclipse Workspace einzubinden reichen drei einfache Schritte aus:

1. In Eclipse unter *File / Import...* auswählen.
2. *General / Existing Projects into Workspace* markieren und dann auf *Next* klicken.
3. *Select archive file > Browser...* auswählen und Pfad zur „SlickGameEinstiegsprojekt.zip“ angeben. Anschließend auf *Finish* klicken (Abbildung 2).

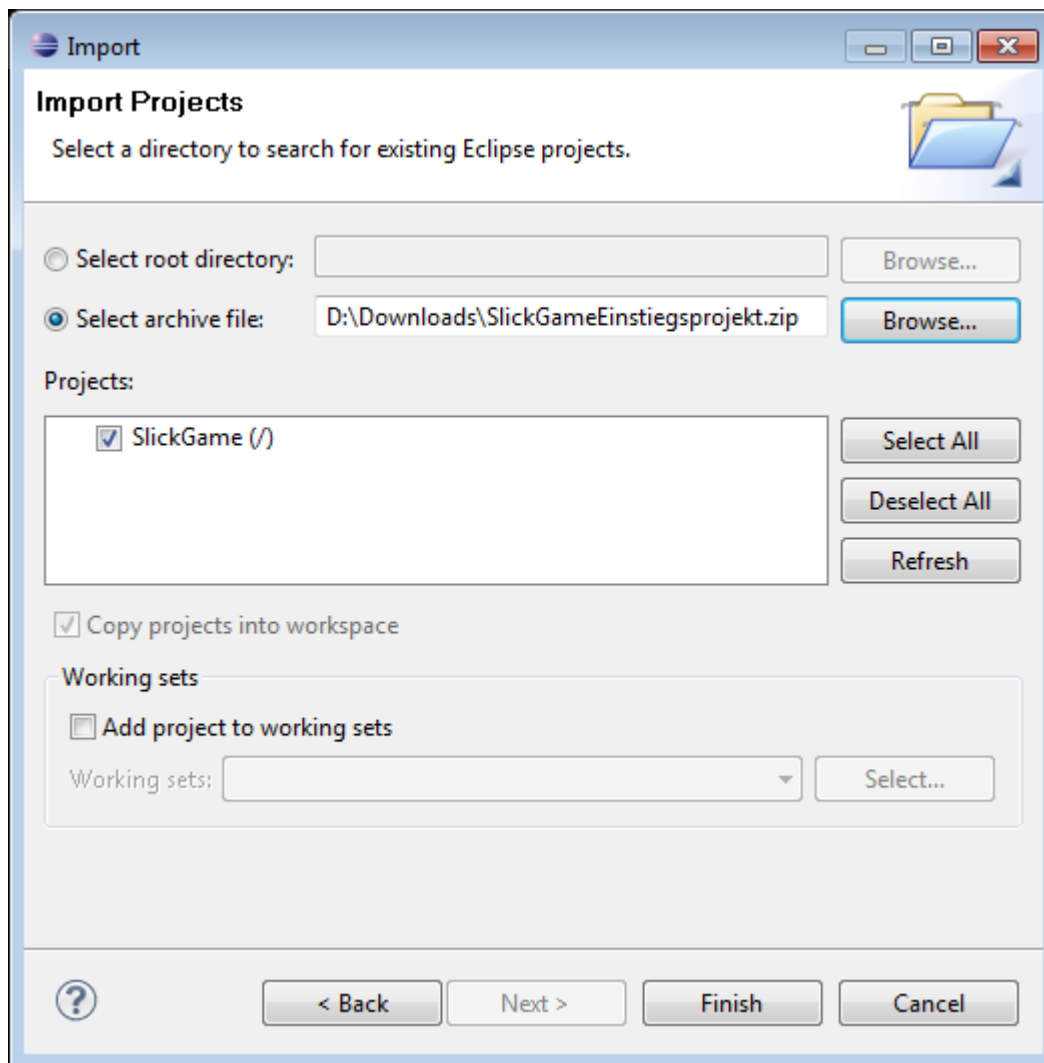


Abbildung 2 Import des Beispielprojekts

Das importierte Projekt sollte nun folgendermaßen aussehen:

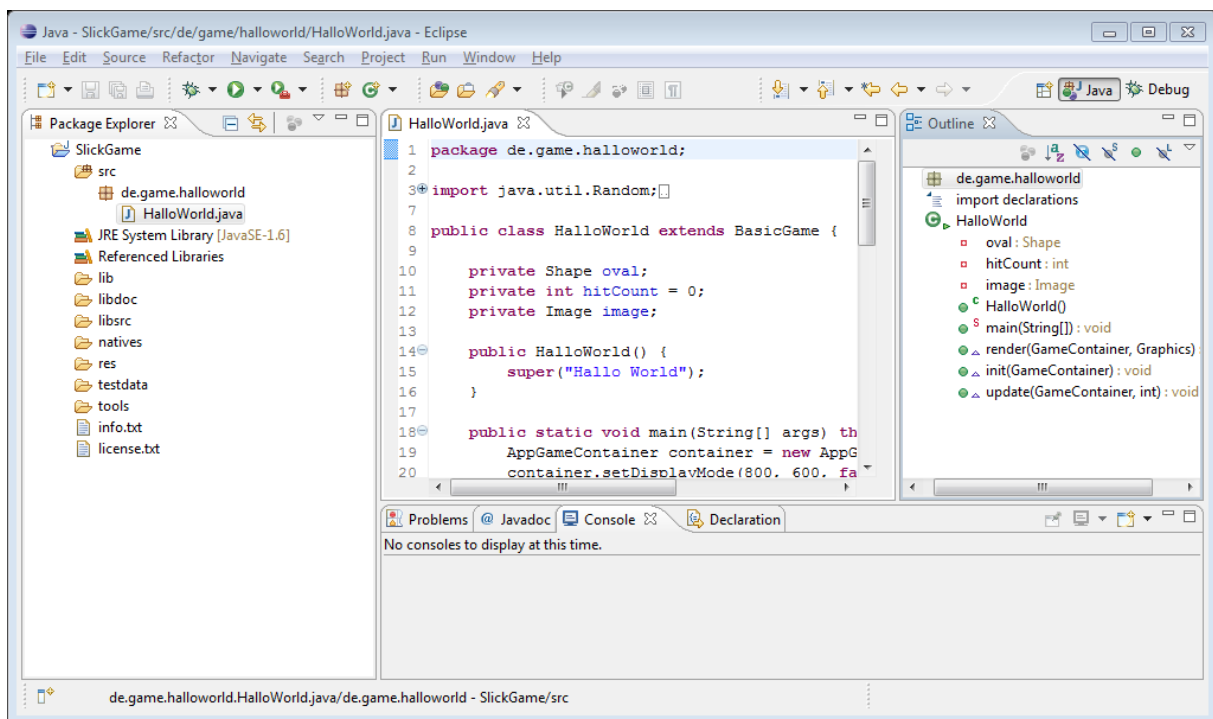


Abbildung 3 Eclipse nach dem Import des Einstiegsprojekts

Um zu testen ob das Projekt erfolgreich eingerichtet ist, starte im *src* Ordner im Package *de.game.helloworld* die Klasse *HelloWorld* -mit *Run / Run As / 1 Java Applikation*. Mit einem Tastendruck auf *Esc* schließt sich das Fenster wieder.

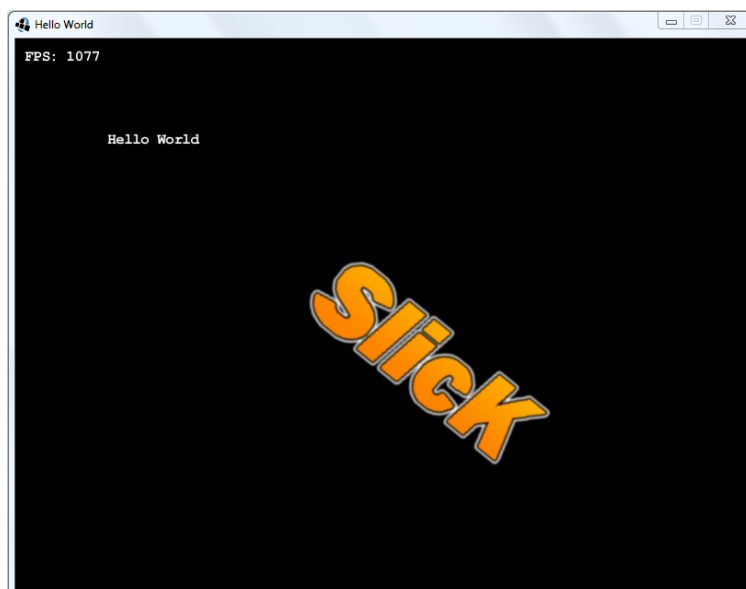


Abbildung 4 Hello World Beispiel nach dem Start

Das Einstiegsprojekt

Im SlickGame Projekt ist alles integriert um ein Spiel in Slick zu entwickeln und zu veröffentlichen. Um ein fertiges Spiel zu starten, ist nur ein kleiner Teil dieser Dateien nötig. Der größte Teil der entpackten fast 30MB, sind die Slick Dokumentation in JavaDoc und

zusätzliche JavaSource Pakete. Slick und die Abhängigen Libraries, wie LWJGL, werden in unterschiedlichen Release Zyklen weiterentwickelt. Für das Projekt wurde die neuste LWJGL Version mit dem nightly build von Slick kombiniert um eine möglichst aktuelle Basis bereit zu stellen. Nähere Infos unter Manuelles Einrichten.

Der Inhalt im Detail

info.txt Enthält Versionsinformationen zu dem Einstiegsprojekt.

Verzeichnisse zum Programmieren des Spiels:

src: Ordner für den Quellcode des Spiels -Eclipse Standard Verzeichnis.

doc: Speicherort für die JavaDoc des Spiels -falls verwendet.

res: Verzeichnis für die Spielressourcen (Bilder, Sounds etc.) -anfangs leer, nur ein Vorschlag.

Verzeichnisse um ein Slick-Spiel auszuführen:

lib: Slick und abhängige Java Bibliotheken.

natives: System abhängige Bibliotheken für Linux, Mac und Windows

Zusätzliche Ordner für die Entwicklung mit Slick:

libdoc: JavaDoc Beschreibung von Slick.

libsrc: Quellcode von Slick und LWJGL. Nicht nötig zum Ausführen eines Slick Spiels, aber nützlich um die Slick Beispiele aus dem Package „org.newdawn.slick.tests“ anzuzeigen.

testdata: Ressourcen um die Slick Beispiele zu starten.

tools: Enthält eine .jar in der die Slick Tools zum Starten bereit liegen: Pedigree (Particle Editor), Hiero (Bitmap Fonts), Scalar (Scale 2/3x) und Packer (Sprite Packing Tool).

Mit diesem Wissen kann direkt mit dem Kapitel Slick Konzept fortgeführt werden. Die Kapitel „Manuelles Einrichten“ und „Die Bibliotheken im Detail“ erklären Details, die für das weitere Vorgehen keine Voraussetzung sind. Wer das Einstiegsprojekt jedoch genau unter die Lupe nehmen, oder es gleich selbst anlegen will, ist dazu herzlich eingeladen.

Die Bibliotheken im Detail

Bibliothek	Funktion	Link
ibxm.jar	Spielt Protracker MOD und Fast Tracker 2 XM Musik Dateien ab.	http://freshmeat.net/projects/ibxm/
lwjgl.jar	Lightweight Java Game Library :	http://lwjgl.org

	Schnittstelle zu OpenGL und OpenAL .	
jinput.jar	Stellt Funktionen zum Auslesen von Gamecontrollern zur Verfügung.	https://jinput.dev.java.net
jogg-0.0.7.jar jorbis-0.0.15	Ermöglicht das Abspielen von OGG Musik Dateien.	http://www.javazoom.net/vorbisapi/documents.html
tools.jar	Die Slick Tools die für das Einstiegsprojekt gepackt wurden.	
slick.jar	API des Slick Frameworks für 2D Computerspiele.	http://slick.cokeandcode.com
tinylinepp.jar	Bestandteil von Slick um das Inkscape Vektorformat zu lesen und dazustellen.	

Manuelles Einrichten

Slick kann aus verschiedenen Quellen bezogen werden.

1. Direkt von der Homepage unter Download Slick

Der Download findet sich auf der [Slick Homepage](#) im rechten Navigationsbereich unter *Download / Gelbes Slick Logo*. Dieses Paket enthält alle Tools und Abhängige Bibliotheken (LWJGL, JInput etc.). Die aktuelle Version dieses Builds ist auf der Startseite im Kopfbereich der Homepage eingeblendet z.B. Aktuell: *Current Build #274*. Leider wird dieses Paket nur selten aktualisiert: Die letzten beiden Versionen lagen mehr als sechs Monate auseinander. Auch die darin verwendete LWJGL Version ist mit v2.0b1 (aktuell v2.5) nicht auf dem neusten Stand. Diese zu aktualisieren steht schon länger an⁷. Glücklicherweise lässt sich dies auch händisch sehr schnell erledigen. Dazu später mehr. Ältere Slick Versionen lassen sich unter SourceForge aus diesem Verzeichnis beziehen: <http://sourceforge.net/projects/java-game-lib/files/>.

2. Aus dem SVN

Slick wird in der Community weiterentwickelt und wie am SVN Feed⁸ zu erkennen ist, werden fast täglich neue Versionen commited. Unter dem Link: <https://bob.newdawnsoftware.com/repos/slick/trunk/> kann die aktuelle Entwickler Version als Source Code bezogen werden. Für den Download benötigt man ein Versionskontrollsystem wie das Eclipse Plugin [Subversive](#) oder das Einzelprogramm [RapidSVN](#). Der SVN Ordner ist leider nicht so gut strukturiert wie das fertige Paket aus 1, da darin noch allerlei Test und Beispielprojekte des Slick Teams enthalten sind. Gepackte .jar-Archive müssen erst noch aus dem Code erst erstellt werden. Da versucht wird im SVN ein möglichst neues LWJGL build mit zu führen, ist dieses aktueller wie das aus 1.

⁷ <http://slick.javaunlimited.net/viewtopic.php?t=1649>

⁸ <http://bob.newdawnsoftware.com/WebSVN/rss.php?repname=Slick&path=&rev=0&sc=0&isdir=1>

3. Nightly Build

Seit Oktober 2010 lässt sich Slick auch als Nightly Build beziehen. Diese sind im Aufbau mit 1. weitgehend identisch, enthalten aber wie 2. den aktuellsten Source Code aus dem SVN. Diese Version wird jede Nacht automatisch mit einem Build Server aktualisiert.

Ein eigenes Paket schnüren

Um den vielen neuen Features der aktuellen Slick Versionen nicht zu entgehen und gleichzeitig in den Genuss des neusten LWJGL Version zu kommen, kombinieren wir das Slick Nightly Build mit dem aktuellen LWJGL Release.

1. Download

Slick Nightly Build herunterladen:

<http://slick.cokeandcode.com/static.php?page=nightly-builds> / Slick Nightly Build

Die Zip Datei für das *Slick-Util Nightly Build* wird nicht benötigt, da alles nötige in der Zip darüber bereits enthalten ist.

LWJGL⁹ herunterladen:

<http://lwjgl.org/download.php>

Entpacke beide Zip Archive in eine beliebiges Verzeichnis.

2. Neues Projekt erstellen

Eclipse öffnen und unter *File / New / JavaProject* ein neues Java Projekt erstellen. In unserem Beispiel: *SlickGame*.

3. Ordner Struktur anlegen:

Erstelle mit *File / New / Folder* folgende Verzeichnisse:

- lib
- libdoc
- libsrc
- natives
- res
- tools

4. Dateien Kopieren

Folgend müssen einige Dateien in das Eclipse Projekt kopiert werden. Dies kannst du direkt per Drag&Drop in Eclipse (View *Package Explorer* oder *Navigation*), oder über den

⁹ Zur Zeit diese Artikels: LWJGL Version 2.5

Windows Explorer, erledigen. Bei der zweiten Variante muss nach den Kopierarbeiten das Eclipse Projekt noch aktualisiert werden: Rechtsklick auf das Projekt / Aktualisieren (F5).

Aus dem Slick Nightly Build Verzeichnis:

Kopiere aus dem Ordner *lib* folgende Dateien nach *lib* im Eclipse Projekt:

- *ibxm.jar*
- *jogg-0.0.7.jar*
- *jorbis-0.0.15.jar*
- *slick.jar*
- *tinylinepp.jar*

Die Ordner *tools* und *testdata* verschieben wir ebenfalls ins Projektverzeichnis.

Benenne den Ordner JavaDoc in *slick* um und kopiere ihn nach *javadoc* im Projekt.

Um später in Eclipse JavaDoc Kommentare und bei Bedarf auch den Source Code der Slick Klassen nachschauen zu können, sichern wir uns noch den Ordner *src*. Diesen Packen wir dazu in ein Zip Archiv namens *slick_src.zip* und verschieben das wiederum in den Ordner *libsrc*.

Dateien aus dem LWJGL Verzeichnis:

Verschiebe aus dem Ordner *jar* folgende Dateien nach *lib* im Eclipse Projekt:

- *lwjgl.jar*
- *jinput.jar*

Verschiebe aus den Ordnern *macosx*, *solaris* und *windows* alle Dateien nach *natives* im Eclipse Projekt.

5. Projekt Konfigurieren:

Damit Eclipse die in den Ordner verteilten Bibliotheken, JavaDocs, Natives und Source Codes auch nutzen kann, müssen deren Pfade noch angegeben werden.

1. Natives:

Navigiere in nach: *Project / Properties / Source*.

Dort muss mit Doppelklick unter *SlickGame / src / Native library location* der Ordner *natives* ausgewählt werden. So findet Eclipse beim Starten eines des Spiels automatisch die nativen Bibliotheken. Soll das Spiel außerhalb der IDE laufen, muss der Pfad zu den Natives wieder extra Angeben werden. Dazu mehr im Kapitel Das Spiel Publizieren.

2. Source

Navigiere in nach: *Project / Properties / Source*.

Um später die die Slick Tools verwenden zu können nehmen wir den Ordner *tools* mit *Add Folder...* in den Source path mit auf.

3. Libraries:

Navigiere in nach: *Project / Properties / Libraries*.

Mit *Add JARs...* fügen wie hier alle .jar- Dateien aus dem Ordner *lib* hinzu -Hier lassen sich alle auf einmal auswählen. Anschließend geben wir für die *slick.jar* noch ein *Source attachmet* mit dem Ziel *libsrc/slick_src.zip* an. Dadurch zeigt uns Eclipse die JavaDoc Kommentare und den Quellcode zu den Slick Klassen an. Das ist sehr nützlich beim Programmieren und unentbehrlich wenn wir und die Slick Beispiele anschauen wollen.

Ob danach alles funktioniert lässt sich mit den Slick Tests leicht überprüfen. Siehe dazu Das Kapitel Für jede Funktion ein Beispiel.




Grundsätzliches Herangehen

Diese Schritt für Schritt Anleitung zum Einrichten ist an mehreren Stellen variabel. Legen wir beispielsweise keinen großen Wert auf das neuste LWJGL Release, hätten wir uns den Download davon sparen können. Die natives hätten wir dann aus den Dateien *natives-linux.jar*, *natives-linux.jar* und *natives-win32.jar* mit einem Zip Programm entpackt.

Wichtig ist das Verständnis für das grundsätzliche Vorgehen: Besorgen und Einbinden der erforderlichen Bibliotheken in den Java Build Path unter Libraries und das Einstellen der natives unter Source. Wahlweise können danach noch JavaDoc und SourceCode Attachments mit den eingebundenen Bibliotheken verknüpft werden. Dieses Vorgehen lässt sich auf nahezu alle anderen Java Bibliothek übertragen - meist sind dabei keine Natives nötig.

Slick Konzept

Das Herzstück eines Spiels ist in Slick die abstrakte Klasse **Game**. Darin enthalten sind die drei Methoden, die den Spielablauf steuern:

Methode	Aufgaben
init() 	<ul style="list-style-type: none"> ○ Zum Laden der Spielressourcen beim Start. ○ Daten initialisieren. ○ Wird ein einziges Mal bei Spielstart aufgerufen.
render() 	<ul style="list-style-type: none"> ○ Ausführen aller Zeichenoperationen -Grafiken, Shapes, Partikel Effekten etc. darstellen. ○ Wird unabhängig von update so oft wie möglich aufgerufen
update() 	<ul style="list-style-type: none"> ○ Enthält die Spielschleife (GameLoop). ○ Spiellogik ○ Bewegungen ○ Wird in einem (möglichst) festen Intervall aufgerufen.

Wollen wir ein Spiel mit Slick programmieren, müssen wir also die Schnittstelle Game implementieren. Eine Klasse die dies bereits tut und uns zusätzlich noch das Abfragen von Gamecontroller Eingaben erleichtert, ist die Klasse **BasicGame**. Mit ihr werden wir auch unser erstes Spiel aufsetzen. Soll das Spiel in unterschiedliche Teile wie z.B. Hauptmenü, Spiel und Credits aufgeteilt werden, erweist sich die Klasse *StateBasedGame* als sehr praktisch. Mit ihr können unabhängige Spielabschnitte programmiert werden, zwischen denen sich ähnlich wie bei PowerPoint Folien, mit Übergangseffekten umschalten lässt.

Das *Game* selbst läuft in einem *GameContainer* ab. Dieser Container entscheidet, ob das Spiel als Desktop Applikation (*AppGameContainer*), als Java Applet im Browser (*AppletGameContainer*), oder eingebettet in einer grafischen Swing Oberfläche (*CanvasGameContainer*) ablaufen soll. Ein weiterer Container für das Android System befindet sich derzeit in Entwicklung.¹⁰

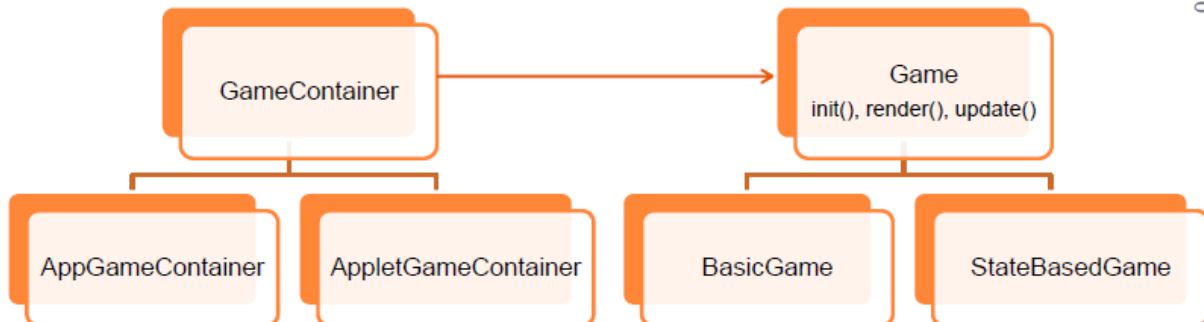


Abbildung 5 Zusammenhang der Slick Game Klassen

Das erste Hello World

Bevor wir mit unserem Spiel richtig loslegen, schauen wir uns am Beispiel eines HelloWorld Programms das Zusammenspiel von *GameContainer* und *Game* an. Darin sollen die Hauptelemente unseres späteren Spiels erkennbar werden:

- Das Laden.
- Das Zeichnen und animieren von Grafiken.
- Das Reagieren auf Tastaturereignisse.

Unser Hello World wird ein Bild auf dem Bildschirm drehen und daneben den Text „Hello World schreiben“.

Ab ins kalte Wasser:

```

//Import der nötigen Slick Klassen
import org.newdawn.slick.*;

public class HelloWorld extends BasicGame {
    private Image image;

    public HelloWorld() {
        //Setzen des Fenstertitels
        super("Hello World");
    }

    public static void main(String[] args) throws SlickException {
        AppGameContainer container = new AppGameContainer(new
            HelloWorld());
        //Fenster mit 1024 x 786 im Fenstermodus (false)
        container.setDisplayMode(1024, 768, false);
        container.start();
    }
}
  
```

¹⁰ <http://slick.javaunlimited.net/viewtopic.php?t=2834>

```

@Override
public void init(GameContainer container) throws SlickException
{
    //Bild logo.png aus dem Verzeichnis retest laden
    image = new Image("retest/logo.png");
}

@Override
public void render(GameContainer container, Graphics g) throws
SlickException {
    //Text und Bild zeichnen
    g.drawString("Hello World", 100, 100);
    g.drawImage(image, 300, 300);
}

@Override
public void update(GameContainer container, int delta) throws
SlickException {
    //Animation: Bild rotieren
    image.rotate(0.05f);
    //Tastenabfrage: Mit Esc-Taste das Spiel beenden
    if(container.getInput().isKeyPressed(Input.KEY_ESCAPE)){
        container.exit();
    }
}
}

```

Das Hello World-Beispiel zeigt das Laden eines Spiels in einem *AppgameContainer* und den Ablauf über *init()*, bis hin zu den Schleifen *render()* und *update()*.

Der Start in main()

Wie andere Java Desktop Anwendungen starten wir unser Spiel auch mit einer *main()*-Methode. In dieser konfigurieren und starten wir lediglich den einen *AppgameContainer*.

Unsere Klasse erbt vom *BasicGame* und ist somit bereit, von einem *GameContainer* aufgenommen zu werden. Durch die Vererbung sind wir gezwungen in der *HelloWorld* Klasse die Methoden *init()*, *update()* und *render()* zu implementieren. Dazu später.

Da unser Spiel auf einem Desktop System laufen soll, nehmen wir einen *AppgameContainer* für unser Spiel. Diesem geben wir im Konstruktor das Spiel das er ausführen soll mit. In unserem Fall eine Instanz der Klasse *HelloWorld*.

Mit `setDisplayMode(1024, 768, false)` definieren wir für unser *HelloWorld* in einem Fenster (`fullscreen = false`) in einer Auflösung von 1024 x 786. Nach dem der Container konfiguriert wurde, starten wir den Spielablauf mit *start()*.

In init() wird geladen

Als erstes ruft der *Gamecontainer* die *init()* Methode unseres Spiels auf. Darin sollten wir unser Spiel initialisieren. Dazu gehört die nötigen Grafiken, Schriften und Sounds, zu laden,

so wie die wichtigsten Objekte anzulegen. Damit stellen wir sicher, dass während dem laufenden Spiel keine Pausen durch Nachladen von Grafiken entstehen. Außerdem können wir später in unserem richtigen Spiel für das laden der Grafiken automatisch einen Fortschrittsbalken anzeigen lassen.

Da bewegt sich was in update!

Nach dem *init()* abgelaufen ist, beginnt der *AppgameContainer* die Methoden *update()* und *draw()* unabhängig voneinander aufzurufen. Lassen wir die Standardeinstellungen wird update ca. alle 15ms Sekunden aufgerufen. Diesen update Intervall können wir mit zwei Methoden beeinflussen:

- `setMinimalUpdateInterval(int delta)`
- `setMaximalUpdateInterval(int delta)`

Damit können wir die maximale und die minimale Zeit zwischen zwei Updates einschränken. Als gute Einstellung hat sich der Wert 25 (Millisekunden) bewährt. Wird beispielsweise ein Raumschiff in Update stets um 2 Pixel nach rechts verschoben, können wir davon ausgehen, dass diese Bewegung annähernd linear abläuft (80px / s).

Soll eine zeitliche Aktion vom update Rhythmus unabhängig ablaufen, kommt die Variable *delta* ins Spiel. Dies ist die Zeit in Millisekunden zwischen zwei update Aufrufen. Um beispielsweise eine Bombe nach 10 Sekunden explodieren zu lassen, brauchen wir lediglich das *delta* in jedem Update aufsummieren. Sobald dann *summeDelta* ≥ 10.000 ist, kann die Sprengung beginnen.

Für unser HelloWorld haben wir kein Intervall angeben, so dass die *update()* und somit die Drehung so schnell wie möglich hintereinander ausgeführt wird. Nach dem Drehen soll auf Tastatureingaben reagiert werden. Das geschieht stets nach dem gleichen Prinzip:

Mit `container.getInput().isKeyPressed(Input.KEY_XXX)`¹¹ abfragen ob die Taste seit dem letzten update gerückt worden ist und dementsprechend Aktionen auslösen.

Mal mir ein Bild: render()!

In der Render Methode werden alle Zeichenoperationen ausgeführt. Dies geschieht folgendermaßen:

1. Einen Zwischenspeicher für das Zeichnen (Framebuffer¹²) leeren.
2. Alle Zeichenoperationen aus *render()* auf dem Framebuffer ausführen.
3. Framebuffer auf dem Bildschirm ausgeben.

Übermalen wir den schwarzen Hintergrund sowieso stets komplett mit einer Grafik, lässt sich Schritt 1 auch überspringen:

```
container.setClearEachFrame(boolean clearEachFrame);
```

¹¹ XXX Steht für eine beliebige Taste. Z.B. `Input.KEY_ESCAPE` (Esc -Taste) oder `Input._KEY_LEFT` (Nach Links).

¹² Der Framebuffer ist immer genau so groß wie die Zeichenfläche im Fenster.

Zum Zeichnen arbeiten wir stets auf dem *Graphics*, das *render()* übergeben bekommt. Manche Klassen bieten auch die Möglichkeit ohne ein *Graphics* zu zeichnen. Beispielsweise:

```
Image.draw();
```

Damit wird das Bild an der Koordinate (0/0) ausgegeben. So lange der Aufruf dabei innerhalb der Methode *render()* erfolgt, kann dabei nichts schief gehen.

Für jede Funktion ein Beispiel

Das HelloWorld Beispiel hat den Aufbau eines einfachen Spiels und den Umgang mit Bildern demonstriert. Slick bietet für jede Funktion eine extra Test Klasse, die deren Verwendung mit einem Beispiel demonstriert. Diese sind im Slick Java Archiv im Paket *org.newdawn.slick.test* zu finden. Dank des eingebundenen Slick Quellcodes, können die Test Klassen mit Doppelklick im Package Explorer direkt geöffnet und betrachtet werden.

Achtung: Zum Starten funktioniert auch hier der Weg über Run / *Run As* > *1 Java Application*, allerdings mit einer Tücke: Der Menüpunkt ist nur verfügbar, wenn die zu startende Class Datei im Package Explorer den aktuellen Fokus besitzt.

Das Editieren eines Beispiels funktioniert nicht auf der Class-Datei, da diese bereits kompiliert ist. Zum Ändern kann man sich den Code aber einfach in eine neue Klasse kopieren. Alternativ hätte man auch gleich den SourceCode von Slick einbinden und dann dort auch die Klassen direkt editieren können. Jedoch wäre dann das Projekt gleich mit

Warnmeldungen aus dem Slick Source Code übersäht –Letzten Endes eine Geschmacksfrage.

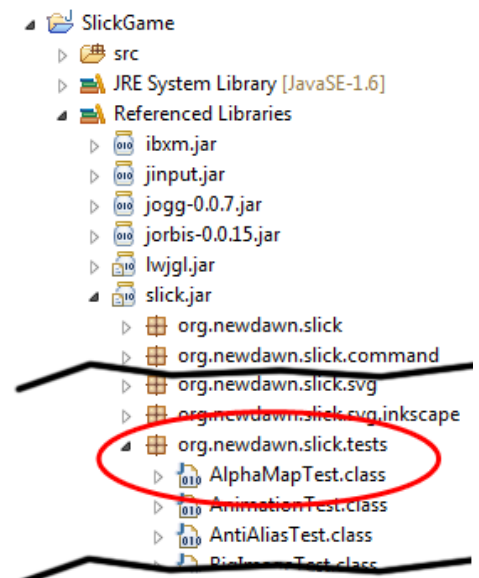


Abbildung 6 Slick Package mit Test Klassen

Hilfe, aber Wo?

Neben den praktischen Beispielen bietet Slick wie andere Bibliotheken auch als erste Anlaufstelle eine **JavaDoc**. Diese wird schon beim Eingeben oder überfahren von Methoden eingeblendet, oder kann mit dem Reiter¹³ *Javadoc* auch permanent angezeigt werden. Wer gerne die gesamte Dokumentation in einem Extra Fenster sehen will, kann dazu lokal die *index.html* aus dem Ordner *libdoc/slick* im Browser öffnen -oder im Web unter:

<http://slick.cokeandcode.com/javadoc/>

Neben der JavaDoc ist vor allem für Einsteiger das Wiki von Slick zu empfehlen:

<http://slick.cokeandcode.com/wiki/doku.php>

Darin sind neben einem Ausführlichen Nachschlagewerk, weitere Tutorials in Englisch enthalten.

Als dritte Anlaufstelle bei Problemen oder Fragen steht die Slick Community im Forum unter:

<http://slick.javaunlimited.net>

mit Rat und Tat zur Seite.

¹³ Kann folgendermaßen eingeblendet werden: *Window / Show View > Other... > Javadoc*

Das Spielkonzept

Für dieses Lernmodul wurde als Projekt ein klassisches Shoot Em Up Arcade Spiel gewählt. Darin soll ein Raumschiff wahlweise mit der Maus, der Tastatur, oder einem Gamepad bewegt werden, um angreifende Ufos ab zu wehren. Dazu kann mit dem Raumschiff auf das

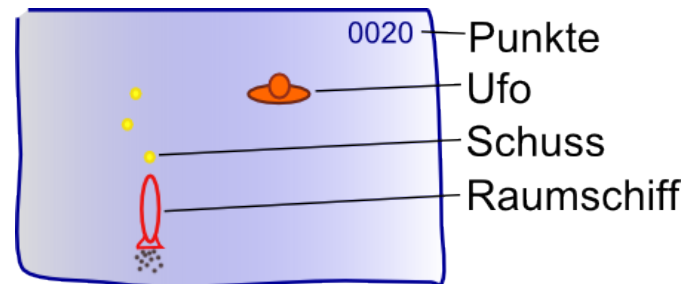


Abbildung 7 Spielaufbau von UfoInvasion

anfliegende Ufo geschossen werden. Bei einem Treffer werden Punkte Hochgezählt und ein neues Ufo zum Angriff auf die Erde geschickt. Der Spieler hat verloren, wenn ein Ufo die untere Bild Bildschirmkante, und somit die Erde erreicht. Das Spiel wird auf den Namen **UfoInvasion** getauft.

Das Spiel ist für diese Einführung bewusst einfach gehalten. Funktionen wie eine Highscore Liste, Mehrere Ufos zu gleichen Zeit, verschiedene Levels und Schwierigkeitsmodi werden in diesem Modul nicht besprochen, oder nur knapp angerissen. Jedoch ist diese Übung darauf ausgelegt genügen Wissen zu vermitteln, um diese Funktionen eigenständig nachreichen zu können.

Ein neues Projekt anlegen

Als erstes legen wir ein Projekt Einstiegsprojekt mit dem Namen **UfoInvasion** an. Dazu ändern wir einfach den Namen des Einstieg Projektes, das wir bereits angelegt haben. Dies müssen wir auch tun wenn für das Spiel ein neues Projekt angelegt werden soll, da im Workspace keine zwei Projekte den gleichen Namen tragen dürfen.

Achtung: Den Projektnamen in Eclipse mit *Refactor / Rename* (Alt + Shift + R) ändern, nicht im Windows Explorer.

Im neuen Projekt legen wir nun im Package *de.game.ufoinvasion* die Klasse *UfoInfavion* an.

Ressourcen

Die in dem Lernmodul verwendeten Ressourcen können [hier](#) heruntergeladen werden: *ressourcen.zip*

Alle darin enthalten Grafiken, Sounds Font, und Musik (mod) Dateien stehen Copyright frei zu Verfügung. Alle stammen aus den Folgenden Quellen:

- <http://www.flashkit.com>
- <http://openclipart.org>
- <http://modarchive.org>

Eine Liste mit weitem Quellen für die Spieleprogramm sind unter Spiel Ressourcen aufgeführt.

Der *res* Ordner aus der *ressourcen.zip* kann direkt in das eben angelegte Projekt kopiert werden.

Arbeitsauftrag 1

Lass die *UfoInvaion* Klasse von *BasicGame* erben und lege eine *main* Methode an, die ein Spiel im Fenstermodus mit der Auflösung 1024 x 765 startet. Als Hintergrund soll schon mal die Galaxie aus dem *res* Verzeichnis dargestellt werden. Außerdem wäre es praktisch das Spiel mit einem Tastendruck auf Esc-Taste beenden zu können.

The first Contact: Ein UFO kommt ins Spiel

Die *UfoInvasion* klasse könnte nach dem Arbeitsauftrag 1 folgendermaßen aussehen:

```
package de.game.ufoinvasion;

import java.util.*;
import org.newdawn.slick.*;

public class UfoInvasion extends BasicGame {

    private Image hintergrund;

    public UfoInvasion() {
        super("Ufo Invasion");
    }

    public static void main(String[] args) throws SlickException {
        AppGameContainer container = new AppGameContainer(new
            UfoInvasion());
        container.setDisplayMode(1024, 768, false);
        container.setClearEachFrame(false);
        container.start();
    }

    @Override
    public void render(GameContainer container, Graphics g) throws
        SlickException {
        hintergrund.draw();
    }

    @Override
    public void init(GameContainer container) throws
        SlickException {
        hintergrund = new Image("res/galaxie.jpg");
    }

    @Override
    public void update(GameContainer container, int delta) throws
        SlickException {
        if (input.isKeyPressed(Input.KEY_ESCAPE)) {
            container.exit();
        }
    }
}
```

Die Anweisung

```
container.setClearEachFrame(false);
```

nimm Slick die Arbeit für das Leeren des Framebuffers ab. An der im Spiel eingeblendeten Framerate (FPS) lässt sich schnell der Geschwindigkeitszuwachs durch diese Maßnahme erkennen: Auf meinem Laptop stieg die Framerate von 359 auf 540 – ein Zuwachs von 66%!

Alle anderen Zeilen entsprechen weitestgehend dem HelloWorld Beispiel.

Nun soll das erste Ufo auftauchen. Dies könnten wir, wie den Hintergrund auch, direkt hinzufügen und an einer festen Position zeichnen lassen. Ein *Image* besitzt in Slick keine Position, sondern wird stets über die *draw()* Methoden am Bildschirm positioniert. Soll das Ufo bewegt werden, muss man das Zeichnen von einer *x* und *y* Variable abhängig machen. Da es später auch andere bewegliche Objekte wie das Raumschiff, oder die Schüsse geben wird, ist es ratsam die Position im jeweiligen Objekt zu speichern. Dazu legen wir eine abstrakte Klasse **SpielObjekt** an.

Ein *SpielObjekt* soll gezeichnet und animiert werden (render und update). Neben der Position in *x* und *y*, speichert es noch ein Bild zum Zeichnen. Getter, Setter und Konstruktoren sind im Klassendiagramm nicht dargestellt.

Die beschriebene Klasse ist mit Hilfe der Eclipse Methoden *generateXX* im Menüeintrag *source*, schnell angelegt:

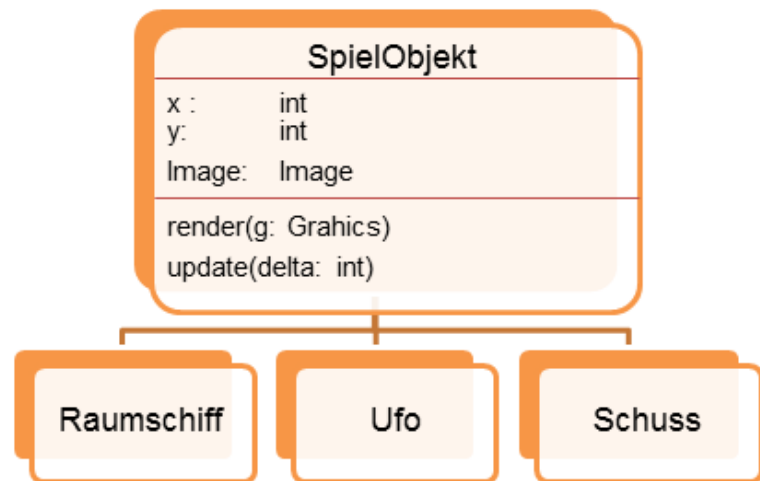


Abbildung 8 UfoInvasion Spielklassen die von SpielObjekt erben

```

package de.game.ufoinvasion;
import org.newdawn.slick.*;

public abstract class SpielObjekt {

    protected int x;
    protected int y;
    protected Image image;

    public abstract void draw(Graphics g);
    public void update(int delta){};

    public SpielObjekt(int x, int y, Image image) {
        this(x, y);
        this.image = image;
    }

    public SpielObjekt(int x, int y) {
        this.x = x;
    }
  
```

```

        this.y = y;
    }

    public SpielObjekt(Image image) {
        this.image = image;
    }

    public SpielObjekt() {
    }

    public int getX() {
        return x;
    }

    public void setX(int x) {
        this.x = x;
    }

    public int getY() {
        return y;
    }

    public void setY(int y) {
        this.y = y;
    };
}

```

Nun fehlt uns nur noch unser unbekanntes Flugobjekt. Dazu erstellen wir die Klasse **Ufo**, die von *SpielObjekt* erbt. Noch schnell den Konstruktor für alle drei Variablen angelegt und in der *draw* Methode das *image* zeichnen lassen. Das Ufo Objekt legen wir in unsere Hauptklasse *UfoInvasion* an und initialisieren es in *init()* mit der Position $x=400$, $y=200$. Damit das Ufo auch gezeichnet wird und sich nachher auch bewegen kann, muss noch in *render()* die Methode

```
ufo.render(g);
```

und in der *update()* die

```
ufo.update(delta)
```

aufgerufen werden.

Ein Bild enthält mehr als nur Farben

Bisher haben wir uns um das Format des Bildes keinerlei Gedanken gemacht. Ein Slick Image kann problemlos mit den Bildformaten PNG, JPG, GIF und TGA umgehen. PNG hat den Vorteil dass es

die Farbvielfalt von JPG mit den Transparenzinformationen eines GIF's kombiniert. Die Transparenz wird von Slick berücksichtigt, so dass keine Umständlichen Alpha Maps für das freistellen von Bildern erforderlich sind. Mit einer Bildbearbeitungssoftware wie Gimp oder Photoshop lässt sich leicht ein weißer Hintergrund in einem JPG entfernen und das Bild anschließend mit Transparenzinformation als PNG speichern. Eine weiterer Komfort in Slick ist, dass alle Bilder nach dem durchlauf von *Init*, geladen sind. Wer in Java schon mit Swing



Abbildung 9
Transparenzinformationen im Bild

Bilder dargestellt hat, kenn die Problematik, dass diese nicht beim Anlegen, sondern beim ersten Zugriff geladen werden.

Positionierung

Das Ufo ist für seinen ersten Flug bereit. Doch wo wird es mit der Position (400/200) auftauchen? Anders als es vielleicht zu erwarten wäre, liegt der Ursprung für das Koordinatensystem auf dem Bildschirm oder einem Fenster nicht in der linken unteren, sondern in der linken oberen Ecke (Abbildung 10). Somit wissen wir jetzt auch, dass wir für eine UFO Bewegung von oben nach unten, dessen Y-Wert schrittweise erhöhen müssen.

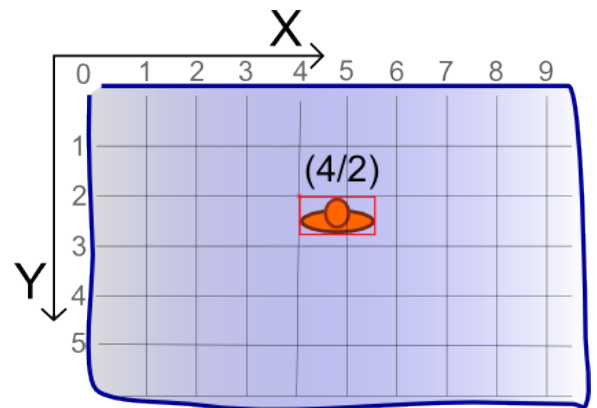


Abbildung 10 Positionierung auf dem Bildschirm

Das UFO lernt fliegen

Das UFO soll sich alle 25 Millisekunden um 2 Pixel nach unten bewegen. Dazu legen wir zuerst in der *main()* den minimalen Updateintervall fest:

```
container.setMinimumLogicUpdateInterval(25);
```

Die Geschwindigkeit legen wir in einer Variable im Ufo an und addieren diese jedes update auf die Y-Position drauf:

```
private double geschwindigkeit = 2;
@Override
public void update(int delta) {
    y += geschwindigkeit;
}
```

Schon bewegt sich unser Ufo nach dem Start gleichmäßig der Erde entgegen. Um das Spiel nach und nach etwas schwieriger zu machen, erhöhen wir die Geschwindigkeit des Ufos stetig:

```
private double geschwindigkeit = 2;
private double beschleunigung = 0.001;
@Override
public void update(int delta) {
    geschwindigkeit += beschleunigung;
    y += geschwindigkeit;
}
```

Arbeitsauftrag 2

Erstelle wie für das Ufo auch eine **Raumschiff**-Klasse, die vom *SpielObjekt* erbt. Das *Raumschiff* soll sich durch seine Update-Methode ständig auf dem Mauszeiger befinden.

Hinweis: `container.getInput().getMouseX();`

Um den Ufos Gegenwehr zu bieten muss das Raumschiff schießen lernen. Erstelle dazu die Klasse **Schuss**. Ein Schuss wird mit einem gelben Kreis gezeichnet.

Der *Schuss* bewegt sich immer mit der Geschwindigkeit von 5 nach oben. Um mehr Schüsse

gleichzeitig auf dem Bildschirm anzuzeigen, musst du eine Liste mit Schüssen in der *UfoInvasion* Klasse anlegen. In *Render* und *Update* musst dann die ganze Liste durchgegangen werden. Mit einem Mausklick (`Input.MOUSE_LEFT_BUTTON`) wird ein Schuss abgefeuert.

Hinweis: `g.fillOval(int x, int y, int weight, int width);`

Mit dem Raumschiff auf Abfangkurs

Nach dem Ausführen des letzten Auftrags kann mit einem Raumschiff auf das UFO geschossen werden. Die Klassen dazu könnten folgendermaßen aussehen:

```
public class Raumschiff extends SpielObjekt {

    private Input input;

    public Raumschiff(Image image, Input input) {
        super(image);
        this.input = input;
    }

    @Override
    public void update(int delta) {
        x = input.getMouseX();
        y = input.getMouseY();
    }

    @Override
    public void draw(Graphics g) {
        image.drawCentered(x, y);
    }
}

public class Schuss extends SpielObjekt {

    private int geschwindigkeit = 5;
    private int radius = 4;

    public Schuss(Sound blasterSound) {
        this.emitter = emitter;
    }

    @Override
    public void update(int delta) {
        y -= geschwindigkeit;
    }

    @Override
    public void draw(Graphics g) {
        g.setColor(Color.yellow);
        g.fillOval(x - radius, y - radius, radius * 2, radius * 2);
    }
}

public class Ufo extends SpielObjekt {

    private double geschwindigkeit = 2;
    private double beschleunigung = 0.001;

    public Ufo(int x, int y, Image image) {
```

```

        super(x, y, image);
    }

    @Override
    public void update(int delta) {
        geschwindigkeit += beschleunigung;
        y += geschwindigkeit;
        if (y >= 1000) {
            y = 0;
        }
    }

    @Override
    public void draw(Graphics g) {
        image.drawCentered(x, y);
    }
}

public class UfoInvasion extends BasicGame {

    private Image hintergrund;
    private Raumschiff raumschiff;
    private List<Schuss> schuesse = new ArrayList<Schuss>();
    private Ufo ufo;

    public UfoInvasion() {
        super("UFO Invasion");
    }

    public static void main(String[] args) throws SlickException {
        AppGameContainer container = new AppGameContainer(new
            UfoInvasion());
        container.setDisplayMode(1024, 768, false);
        container.setClearEachFrame(false);
        container.setMinimumLogicUpdateInterval(25);
        container.start();
    }

    @Override
    public void render(GameContainer container, Graphics g) throws
        SlickException {
        hintergrund.draw();
        raumschiff.draw(g);

        for (Schuss schuss : schuesse) {
            schuss.draw(g);
        }
        ufo.draw(g);
    }

    @Override
    public void init(GameContainer container) throws SlickException {
        hintergrund = new Image("res/galaxie.jpg");
        raumschiff = new Raumschiff(new Image("res/raumschiff.png"),
            container.getInput());
        ufo = new Ufo(400, 200, new Image("res/ufo.png"));
    }

    @Override
    public void update(GameContainer container, int delta) throws
        SlickException {

```

```

Input input = container.getInput();
raumschiff.update(delta);

if (input.isMousePressed(Input.MOUSE_LEFT_BUTTON)) {
    neuerSchuss(mausX, mausY);
}
for (Schuess schuss schuesse){
    schuss.update(delta);
}
if (input.isKeyPressed(Input.KEY_ESCAPE)) {
    container.exit();
}

}

private void neuerSchuss(int mausX, int mausY) {
    Schuss schuss = new Schuss(mausX, mausY);
    schuesse.add(schuss);
}
}

```

Kollisionserkennung

Als nächsten wollen wir, dass die Schüsse unseres Raumschiffs auch treffen. Dazu müssen wir feststellen ob die Position unseres Schusses innerhalb des Ufos liegt. Für solche Kollisionsabfragen bietet Slick Shapes an. Das sind mathematisch beschriebene Flächen, die sich gegenseitig auf Kollision prüfen und auf einem Graphics zeichnen lassen. Aus der Rubrik *Geometry* im Slick Wiki erfahren wir, welche Shapes alle zur Verfügung stehen:¹⁴

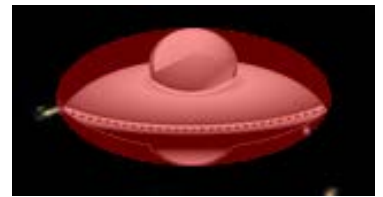


Abbildung 11 Kollisionsfläche des UFOs

Circle, Oval, Rectangle, Polygon und Line. Für die Fläche unseres Ufos ist uns ein *Oval* gut genug. Eine *pruefeKollision* Methode im Ufo ist mit diesem Wissen schnell implementiert:

```

private Shape kollisionsFlaeche;

public Ufo(int x, int y, Image image) {
    super(x, y, image);
    kollisionsFlaeche = new Ellipse(x, y, 60, 30);
}

public boolean pruefeKollision(SpielObjekt spielObjekt) {
    return kollisionsFlaeche.contains(spielObjekt.getX(),
    spielObjekt.getY());
}

```

In der Update muss dann nun nur noch die Position der Kollisionsfläche angepasst werden:

```

public void update(int delta) {
    ...
    kollisionsFlaeche.setCenterX(x);
    kollisionsFlaeche.setCenterY(y);
}

```

Um zu überprüfen ob das Shape auch richtig sitzt können wir in der Render noch schnell die zwei Zeilen zum Zeichnen ergänzen:

¹⁴ Oder man schreibt in Eclipse Shape und drückt dann auf dem Wort strg + t für die Type Hierarchy.


```
g.setColor(Color.red);
g.fill(kollisionsFlaeche);
```

Ab jetzt wird scharf geschossen

Unsere soeben erstellte `pruefeKollision` Funktion muss noch in der Update des Spiels aufgerufen werden. Dazu ergänzen wir die update Schleife der Schüsse wie folgt:

```
for (int i = 0; i < schuesse.size(); i++) {
    Schuss schuss = schuesse.get(i);
    schuss.update(delta);
    if (ufo.pruefeKollision(schuss)) {
        neuesUfo(container, schuss);
    }
}
```

Da wir in der Schleife bei einem Treffer den Schuss entfernen wollen, müssen wir unsere schöne For Each Schleife in eine mit Zählvariablen austauschen. Tun wir das nicht würden wir während des Entfernens eines Schusses aus der Liste, eine *ConcurrentModificationException* von Java vorgeworfen bekommen.

Für Zufallswerte bietet Java neben *Math.random()* auch eine Klasse *Random* an. Der Aufruf

```
new Random().nextInt(42);
```

Liefert eine Zahl von 0 bis 41. Mit diesem Wissen lässt sich die Methode `neuesUfo()` einfach schreiben:

```
private void neuesUfo(GameContainer container, Schuss schuss) {
    schuesse.remove(schuss);
    schuss.verschwinde();
    Random random = new Random();
    ufo.setX(random.nextInt(container.getWidth()));
    ufo.setY(random.nextInt((int) (container.getHeight() * 0.7)));
}
```

Der Sound macht's aus

In unsrem Spiel bewegen sich die Grafiken, aber um das Spiel lebendig erscheinen zu lassen fehlt der Ton. Diesem Missstand ist mit Slick sehr einfach bei zu kommen. Wir brauchen uns wie bei Bildern weder um das Format noch das Laden große Gedanken zu machen. Die Formate WAV, OGG, MOD und XM werden Unterstützt. OGG eignet sich gut für längere Musiktitel und bietet eine ähnlich gute Komprimierung wie MP3. Erstellen lassen sie sich mit Audio Konvertierungsprogrammen wie beispielsweise [BeeSweet](#). Bei den beiden letzten Typen handelt es sich um Computer generierte Musik die sich nur mit Speziellen Playern wie dem [MODPlug Player](#) anhören lassen. Dafür sind selbst Minuten lange Musik Titel nur um die hundert Kilobyte groß. Freie Sounddateien lassen sich unter in den Quellen unter Spiel Ressourcen finden:

Slick unterscheidet zwischen Musik und Sounds. Musik lässt sich nicht von Sounds beeinflussen ist deswegen –wer hätte das gedacht- für lange Hintergrund Stücke gedacht.

Die wichtigen Methoden für Sounds und Musik sind:

```
Music music = new Music("res/sounds/music.mod");  
music.loop();  
  
Sound schuss = new Sound("res/sounds/schuss.wav");  
schuss.play();  
schuss.playAt(-1,0,0); //an der Position (-1,0,0) abspielen (x,y,z)
```

Arbeitsauftrag 3

Füge alle Sounds aus dem Ordner *res/sounds* an den passenden Stellen im Spiel ein.

Ein Text aus Bildern

Wer schon einen Blick in den Ordner *res / fonts* geworfen hat, wird sich wahrscheinlich über dessen Inhalt gewundert haben. Slick nutzt OpenGL und damit direkt die Grafikkarte für die Darstellung auf dem Bildschirm. Diese versteht zwar den Umgang Bilder und selbst aufwendigen 3D Modellen, das darstellen von Text ist ihr jedoch fremd. Darum besteht Text in Slick stets aus Bildern mit Zeichen, den sogenannten Bitmap Fonts. Die klassische Variante ist das *SpriteSheetFont*, bei dem jedes Zeichen genau gleich groß ist. Dies führt jedoch zu einem ungewöhnlichen Schriftbild, bei dem Buchstaben zu große Abstände haben. Darum wurden für OpenGL eigen Schriftformate entwickelt, die Buchstaben als Bild bereithalten und zusätzlich zu jedem Buchstaben Angaben wie Außenabstände und die Unterscheidung¹⁵ kennen: Das *AngleCodeFont*.

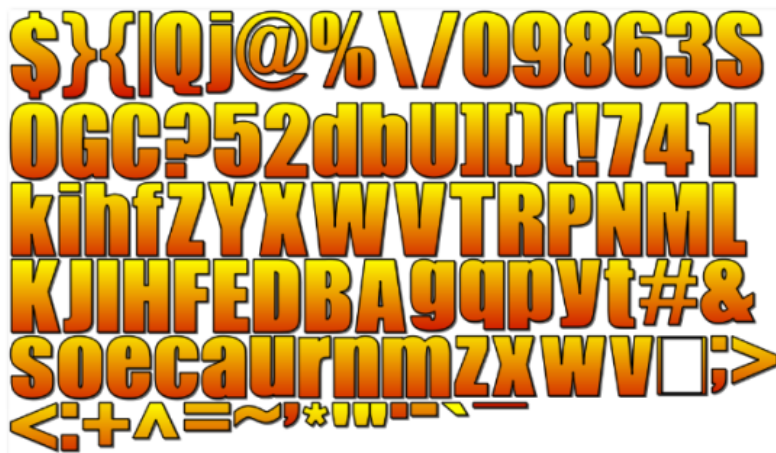


Abbildung 12 PNG Datei einer Angle Code Font mit NEHE Zeichensatz

Solche Schriftarten werden mit Tools wie dem in Slick mitgelieferten Hiero (Bitmap Font Editor) erstellt.

Eine eigene Schrift anlegen

Für das Darstellen der aktuellen Punkte und eines Game Over Schriftzuges am Spielende, brauchen wir in Slick eine Schrift. Die weiße Standard Schriftart in Slick, in der auch die FPS angezeigt wird, ist für diesen Zweck weder groß noch schön genug.

Um *Hiero* zu starten wählt man im *Eclipse Package Explorer* unter *Referenced Libraries* die Bibliothek *tools.jar* aus und wählt im Menü *Run / Run As / 1 Java Applikation*. Im erscheinendem Popup startet man dann mit einem Doppelklick auf *Hiero* das Programm.

Hinweis: Die Slick Tools haben alle samt das Problem nicht richtig zu beenden und müssen aus Eclipse heraus mit *Console / Terminate* (roter viereckiger Knopf) beendet werden.

¹⁵ Auch Kerning genannt: Verschiebung von Buchstaben um ein angenehmes Schriftbild zu erhalten.

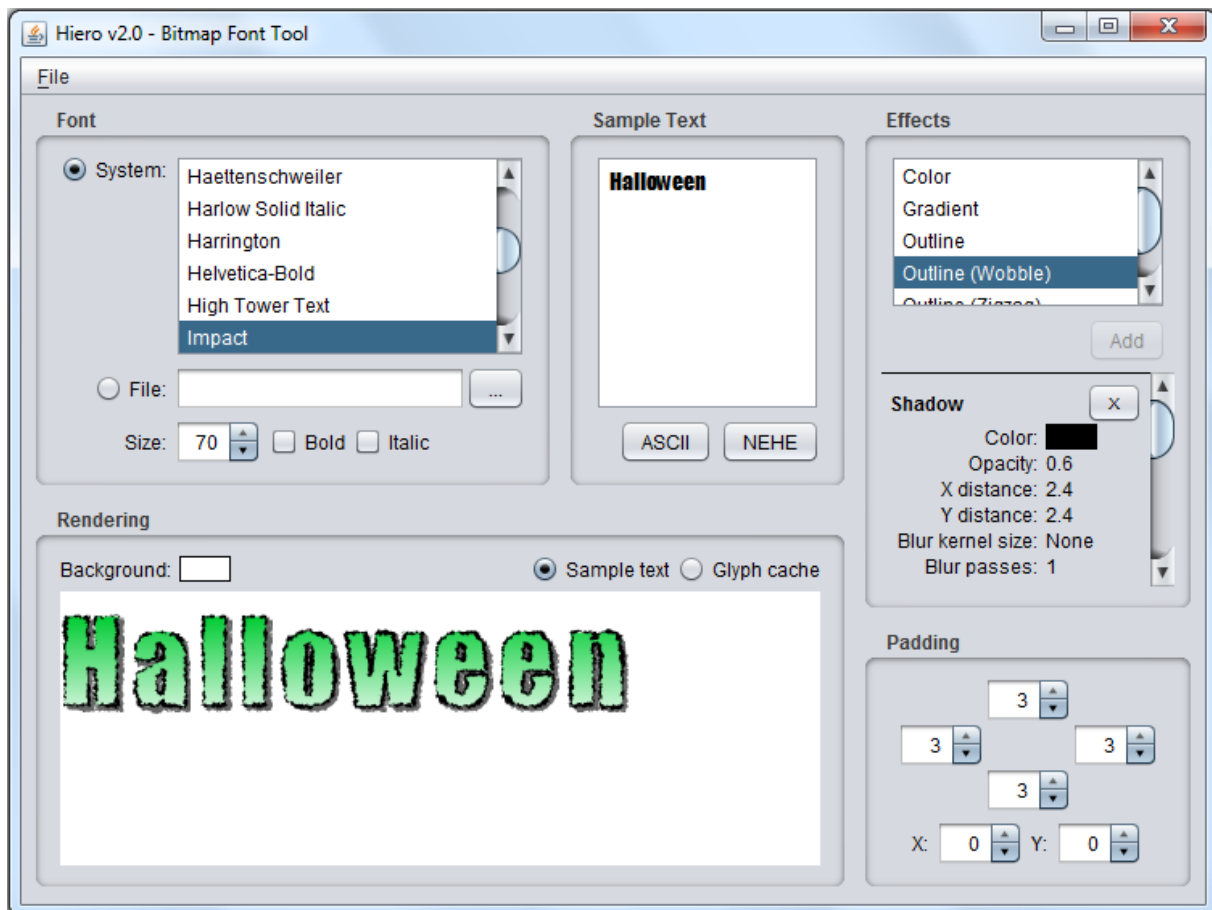


Abbildung 13 Der Hiero Bitmap Font Editor

Mit Hiero lassen sich Schriften mit mehreren Effekten kombinieren. Diese können mit dem *Add* Button hinzugefügt und anschließend angepasst werden. Dabei ist die Reihenfolge beim Einfügen zu beachten: Erst der Schatten, dann der Farbverlauf, dann der Rahmen. Die Effekte vergrößern meist die einzelnen Zeichen, was dann für ein schönes Schriftbild manuell mittels der Außenabständen (Padding) ausgeglichen werden muss.

Bevor man eine Font exportieren kann, muss man ein Paar Einstellungen kontrollieren: Das Textfeld *Sample Text* ist leider irreführende bezeichnet. Nur in ihm aufgeführte Zeichen werden auch beim Export berücksichtigt. Darum muss vor dem Export mit den Buttons *NEHE* oder *ASCII* ein Zeichensatz gewählt werden. Außerdem sind die Werte *Page width* und *Page height* unter *Rendering / Glyph cache* (Radio Button) zu vergrößern. Bei der Standardgröße von 64x64 würden sonst etliche Font Dateien generiert. Nach einem Klick auf *Reset Cache* wird in der DropDownList unter *View*, die für die aktuell eingestellte Page Größe anfallenden Seiten angezeigt. Mit *File / Save BMPFont Files...* kann nun der Export gestartet werden. Die resultierende Einstellungsdatei (.font) und die Bild Datei (.png) werden beide zum Anlegen einer *AngleCodeFont* benötigt.

Text in der eigenen Font ausgeben

Angelegt wird eine *Angelcode Font* hiermit:

```
Font meineSchrift = new AngelCodeFont("res/fonts/score_numer_font.fnt",
                                     new Image("res/fonts/score_numer_font.png"));
```

Wie bei Bildern, gibt es auch für Fonts in Slick zwei Wege diese auszugeben:

1. Direkt über das Font Objekt mit:
`meineSchrift.drawString(...);`
2. Durch das setzen der Schrift auf das Graphics:
`g.setFont(meineSchrift);`
`g.drawString(...);`

Arbeitsauftrag 4

Lege die zwei Klassen **Punkte** und **SpielEnde** an. Mit *Punkte* sollen in vier Stellen formatiert die Punkte am oberen rechten Bildschirmrand angezeigt werden. Jedes abgeschossene UFO soll den Punktestand erhöhen. *SpielEnde* übermalt den Hintergrund mit einem halbtransparenten Schwarz und zeichnet groß den Schriftzug „Game Over“ auf die Bildschirmmitte, sobald ein UFO die untere Bildschirmhälfte erreicht hat.

Hinweise:

- `String.format("%04d", punkte);`
- Der Alphawert einer Farbe wird über dessen öffentliche Variable *a* geändert

Ein Partikel Feuerwerk

Unser Raumschiff bewegt sich bisher scheinbar Antriebslos durchs Weltall. Das soll sich mit einem feurigen Düsenstrahl ändern, den wir mit einem Partikel Effekt realisieren. Slick bringt für solche Zwecke einen Partikel Editor, den die vermutlich Katzen vernarrten Entwickler, auf den Namen **Pedigree** getauft haben. Mit ihm lassen sich Effekte durch einfaches Parametrisieren über eine Oberfläche anpassen. Für die meisten Werte gibt man Bereiche mittels eines Min und Max Wertes an. Für jedes neue Partikel wird dann ein Zufallswert aus dem Bereich gewählt, so dass der Effekt möglichst abwechslungsreich aussieht. Das Ergebnis der Einstellungen ist sofort animiert sichtbar. Durch spielen mit den Werten erhält man so schnell eindrucksvolle Ergebnisse.

Gestartet wird Pedigree auf die gleiche Art wie schon der Font Editor *Hiero*: In Eclipse *Package Explorer* unter *Referenced Libraries* die Bibliothek *tools.jar* selektieren und im Menü *Run / Run As / 1 Java Applikation* anklicken. Im erscheinenden Popup startet man das Toll dann mit einem Doppelklick auf *Pedigree*. Dieser muss nach getaner Arbeit leider hart, über den *Termiate* Button in Eclipse, beendet werden.

Ein Partikel wird aus einem **ParticleEmitter** ausgestoßen. Mehrere *ParticleEmitter* laufen in einem **ParticleSystem** ab. So können beispielweise für eine Explosion mehrere verschiedene Emitter zu einem *ParticleSystem* für die Explosion gebündelt werden: Einer für Rauch, einer für Feuer und ein dritter für eine sich ausbreitende Corona. Mit *Pedigree* können die Emitter entweder einzeln über *Files / Export Emitter* oder auch alle zusammen mittels *Files / SaveSystem* in ein einziges *ParticleSystem* exportiert werden.

Emitter können nicht einzeln, sondern nur innerhalb eines Systems gezeichnet werden.

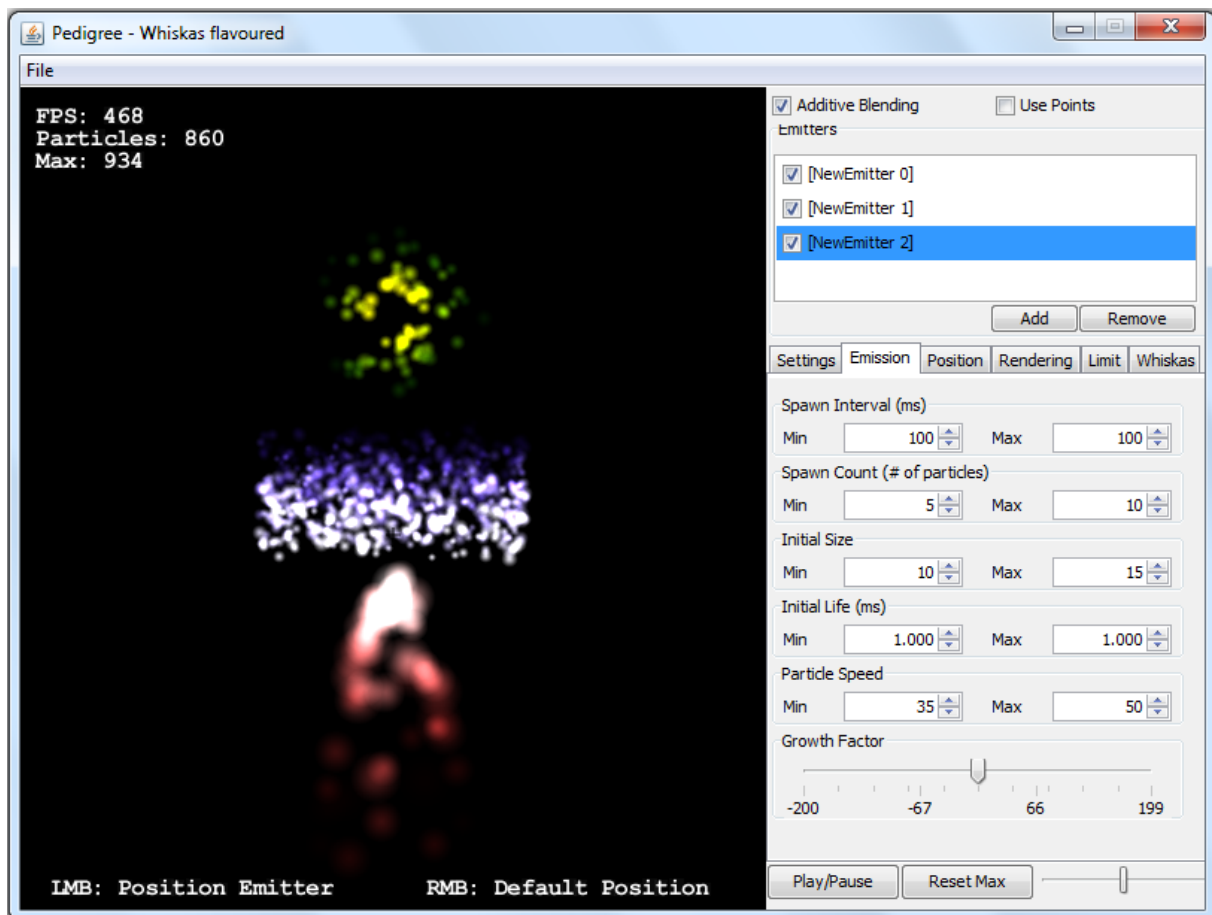


Abbildung 14 Mit dem Partikel Editor lassen sich durch ausprobieren vielerlei Effekte produzieren

Ein Partikelsystem lädt man wie folgt:

```
try {
    partikelSystem =
        ParticleIO.loadConfiguredSystem("mein_partikel_system.xml");
} catch (IOException e) {
    throw new SlickException("Partikel System konnte nicht
        geladen werden", e);
}
```

Schade dass an der Stelle eine *IOException* abgefangen werden muss und nicht wie bei Bildern und Sounds im Fehlerfall eine *SlickException* geworfen wird. Um dem geladenen System Leben einzuhauchen, fehlen noch die Aufrufe von `render()` und `update()`:

```
particleSystem.update(delta);
particleSystem.render();
```

Ein Emitter kann mittels

```
emitter.setPosition(x, y);
```

positioniert werden. Haben wir statt eines ganzen Systems einzelne Emitter in Pedegree gespeichert, laden wir diese so:

```
partikelEmitter =
    ParticleIO.loadEmitter("mein_emitter.xml");
```



Abbildung 15 Raumschiff Partikel

Arbeitsauftrag 5

Im Verzeichnis `\res\particles` liegen fertige Partikel Emitter für einen Strahl des Raumschiffs, für einen Schweif des Schusses und eine Explosion des UFOs. Diese sollen alle in einer extra Klasse **Effekte** geladen werden. Die Emitter sollen dann über Getter abgefragt und den Einzelnen SpielObjekten, zum Zeichnen und setzen der Position, übergeben werden.

Hinweise:

- Die Lebenszeit des Partikel Emitter `schuss_schweif.xml` ist in Pedegree im Reiter *Limits* durch die *Effect Lenght* auf 5 Sekunden limitiert worden. Läuft diese Lebenszeit nach dem laden eines Emitters ab, wird dieser vom ParticleSystem entfernt.
- Mit `emitter.duplicate();` kann ein Emitter geklont werden.

Der Code

Mit den letzten beiden Arbeitsaufträgen ist das UfoInvasion Spiel fertig geworden. Aus einem HelloWorld Beispiel ist eine kleiner Shooter samt Sound, Hintergrundmusik und Partikel Effekten geworden. Der Stand entspricht dem fertigen Spiel das hier als Eclipse Projekt herunter geladen werden kann:

<http://javagaming.tobsefritz.de>

```
public class Effekte extends SpielObjekt {

    private ParticleSystem particleSystem;
    private ConfigurableEmitter raketenRauch;
    private ConfigurableEmitter ufoExplosion;
    private ConfigurableEmitter schussPartikel;

    public Effekte() throws SlickException {
        try {
            particleSystem =
                ParticleIO.loadConfiguredSystem("res/particles/leeres_system.xml");

            schussPartikel =
                ParticleIO.loadEmitter("res/particles/schuss_schweif.xml");
            raketenRauch =
                ParticleIO.loadEmitter("res/particles/raketen_rauch.xml");
            particleSystem.addEmitter(raketenRauch);
            ufoExplosion =
                ParticleIO.loadEmitter("res/particles/ufo_explosion.xml");
        } catch (IOException e) {
            throw new SlickException("Partikel System konnte nicht
                                    geladen werden", e);
        }
    }

    @Override
    public void update(int delta) {
        particleSystem.update(delta);
    }

    @Override
```

```

public void draw(Graphics g) {
    particleSystem.render();
}

public ConfigurableEmitter getSchussEmitter() {
    ConfigurableEmitter emitter = schussPartikel.duplicate();
    particleSystem.addEmitter(emitter);
    return emitter;
}

public ConfigurableEmitter getRaketenRauchEmitter() {
    return raketenRauch;
}

public void ufoExplosion(int x, int y) {
    ConfigurableEmitter explosion = ufoExplosion.duplicate();
    explosion.setPosition(x, y);
    particleSystem.addEmitter(explosion);
}
}

public class Punkte extends SpielObjekt {

    private Font font;
    private int punkte;

    public Punkte(int x, int y, Font font) {
        super(x, y);
        this.font = font;
    }

    @Override
    public void draw(Graphics g) {
        g.setFont(font);
        String punkteMitNullen = String.format("%04d", punkte);
        g.drawString(punkteMitNullen, x, y);
    }

    public void punkte() {
        punkte++;
    }
}

public class Raumschiff extends SpielObjekt {

    private ConfigurableEmitter emitter;
    private Input input;

    public Raumschiff(Image image, Input input, ConfigurableEmitter
        emitter) {
        super(image);
        this.input = input;
        this.emitter = emitter;
    }

    @Override
    public void update(int delta) {
        x = input.getMouseX();
        y = input.getMouseY();
        emitter.setPosition(x, y + 45, false);
    }
}

```



```

@Override
public void draw(Graphics g) {
    image.drawCentered(x, y);
}

}

public class Schuss extends SpielObjekt {

    private int geschwindigkeit = 5;
    private int radius = 4;
    private ConfigurableEmitter emitter;

    public Schuss(int x, int y, Sound blasterSound, ConfigurableEmitter
        emitter) {
        super(x,y);
        this.emitter = emitter;
        emitter.setPosition(x, y);
        blasterSound.playAt(x, y, 0);
    }

    @Override
    public void update(int delta) {
        y -= geschwindigkeit;
        emitter.setPosition(x, y, false);
    }

    @Override
    public void draw(Graphics g) {
        g.setColor(Color.yellow);
        g.fillOval(x - radius, y - radius, radius * 2, radius * 2);
    }

    public void verschwinde() {
        emitter.setEnabled(false);
    }

}

public class SpielEnde extends SpielObjekt {

    private int height;
    private int width;
    private int textWidth ;
    private int textHeight;
    private Color transparent;
    private Font fontGameOver;
    private boolean isGameOver;
    private static final String GAME_OVER = "Game Over";

    public SpielEnde(int height, int width, Font fontGameOver) {
        this.height = height;
        this.width = width;
        this.fontGameOver = fontGameOver;
        transparent = new Color(Color.black);
        transparent.a = 0.8f;
        textWidth = fontGameOver.getWidth(GAME_OVER);
        textHeight = fontGameOver.getHeight(GAME_OVER);
    }

    @Override
    public void draw(Graphics g) {

```

```

        g.setColor(transparent);
        g.fillRect(0, 0, width, height);
        g.setColor(Color.white);
        g.setFont(fontGameOver);
        g.drawString(GAME_OVER, (width / 2) - (textWidth / 2), (height
        / 2) - textHeight);
    }

    public void setGameOver(boolean isGameOver) {
        this.isGameOver = isGameOver;
    }

    public boolean isGameOver() {
        return isGameOver;
    }
}

public abstract class SpielObjekt {

    protected int x;
    protected int y;
    protected Image image;

    public abstract void draw(Graphics g);
    public void update(int delta){};

    public SpielObjekt(int x, int y, Image image) {
        this(x, y);
        this.image = image;
    }

    public SpielObjekt(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public SpielObjekt(Image image) {
        this.image = image;
    }

    public SpielObjekt() {
    }

    public int getX() {
        return x;
    }

    public void setX(int x) {
        this.x = x;
    }

    public int getY() {
        return y;
    }

    public void setY(int y) {
        this.y = y;
    };
}

```

```

public class Ufo extends SpielObjekt {

    private Shape kollisionsFlaeche;

    public Ufo(int x, int y, Image image) {
        super(x, y, image);
        kollisionsFlaeche = new Ellipse(x, y, 60, 30);
    }

    private double geschwindigkeit = 2;
    private double beschleunigung = 0.001;
    @Override
    public void update(int delta) {
        geschwindigkeit += beschleunigung;
        y += geschwindigkeit;
        if (y >= 1000) {
            y = 0;
        }
        kollisionsFlaeche.setCenterX(x);
        kollisionsFlaeche.setCenterY(y);
    }

    @Override
    public void draw(Graphics g) {
        image.drawCentered(x, y);
        Color.red.a = 0.4f;
        g.setColor(Color.red);
        g.fill(kollisionsFlaeche);
    }

    public boolean pruefeKollision(SpielObjekt spielObjekt) {
        return kollisionsFlaeche.contains(spielObjekt.getX(),
            spielObjekt.getY());
    }
}

public class UfoInvasion extends BasicGame {

    private Image hintergrund;
    private Sound soundBlaster;
    private Sound soundExplosion;
    private Raumschiff raumschiff;
    private Effekte effekte;
    private Punkte punkte;
    private List<Schuss> schuesse = new ArrayList<Schuss>();
    private Ufo ufo;
    private SpielEnde gameOver;

    public UfoInvasion() {
        super("UFO Invasion");
    }

    public static void main(String[] args) throws SlickException {
        AppGameContainer container = new AppGameContainer(new
            UfoInvasion());
        container.setDisplayMode(1024, 768, false);
        container.setClearEachFrame(false);
        container.setMinimumLogicUpdateInterval(25);
        container.start();
    }
}

```

```

@Override
public void render(GameContainer container, Graphics g) throws
    SlickException {
    hintergrund.draw();
    raumschiff.draw(g);
    effekte.draw(g);
    for (Schuss schuss : schuesse) {
        schuss.draw(g);
    }
    ufo.draw(g);
    if (gameOver.isGameOver()) {
        gameOver.draw(g);
    }
}

@Override
public void init(GameContainer container) throws SlickException {
    hintergrund = new Image("res/galaxie.jpg");
    effekte = new Effekte();
    raumschiff = new Raumschiff(new Image("res/raumschiff.png"),
        container.getInput(),
        effekte.getRaketenRauchEmitter());
    ufo = new Ufo(400, 200, new Image("res/ufo.png"));
    Font fontPunkte = new
        AngelCodeFont("res/fonts/score_numer_font.fnt",
            new Image("res/fonts/score_numer_font.png"));
    punkte = new Punkte(container.getWidth() - 180, 10,
        fontPunkte);
    soundExplosion = new Sound("res/sounds/explosion.wav");
    soundBlaster = new Sound("res/sounds/schuss.wav");
    Font fontGameOver = new
        AngelCodeFont("res/fonts/game_over_font.fnt",
            new Image("res/fonts/game_over_font.png"));
    gameOver = new SpielEnde(container.getHeight(),
        container.getWidth(), fontGameOver);
    new Music("res/sounds/music.mod").loop();
}

@Override
public void update(GameContainer container, int delta) throws
    SlickException {
    Input input = container.getInput();
    if (!gameOver.isGameOver()) {
        raumschiff.update(delta);
        effekte.update(delta);

        int mausX = input.getMouseX();
        int mausY = input.getMouseY();

        if (input.isMousePressed(Input.MOUSE_LEFT_BUTTON)) {
            neuerSchuss(mausX, mausY);
        }

        for(int i = 0; i < schuesse.size() ; i++) {
            Schuss schuss = schuesse.get(i);
            schuss.update(delta);
            if (ufo.pruefeKollision(schuss)) {
                neuesUfo(container, schuss);
            }
        }

        ufo.update(delta);
    }
}

```

```

// Fenster mit ESC schließen
if (input.isKeyPressed(Input.KEY_ESCAPE)) {
    container.exit();
}
if (ufo.getY() > container.getHeight()) {
    container.setPaused(true);
    gameOver.setGameOver(true);
}
}

private void neuerSchuss(int mausX, int mausY) {
    Schuss schuss = new Schuss(mausX, mausY - 20, soundBlaster,
        effekte.getSchussEmitter());
    schuesse.add(schuss);
}

private void neuesUfo(GameContainer container, Schuss schuss) {
    schuesse.remove(schuss);
    schuss.verschwinde();
    effekte.ufoExplosion(ufo.getX(), ufo.getY());
    Random random = new Random();
    ufo.setX(random.nextInt(container.getWidth()));
    ufo.setY(random.nextInt((int) (container.getHeight() * 0.7)));
    soundExplosion.play();
    punkte.punkte();
}
}

```

Das Spiel exportieren

Bis jetzt lief das Spiel ausschließlich direkt in Eclipse. Um unser Spiel weiter zu geben oder zu veröffentlichen, müssen wir erst aus unsren programmierten Klassen ein Java Archiv (.jar) packen. Dieses kann dann zusammen mit der Slick Bibliothek und den Ressourcen auch auf anderen Computern unabhängig von Eclipse gestartet werden. –Vorausgesetzt auf dem Zielsystem sind Java und OpenGL fähige Grafikkartentreiber installiert.

Bevor wir mit dem Exportieren beginnen, legen wir in unserem Projektverzeichnis einen Ordner *trunk*¹⁶ an. Darin soll in Zukunft immer die aktuellste lauffähige Version unseres Spiels gespeichert werden. Nach dem Anlegen des Ordners, selektieren wir unser *UfoInvasion* Projekt im Package Explorer und wählen unter *File* den Eintrag *Export...* aus. Im drauf hin erscheinenden Export Menü ist *Java / Runnable JAR file* zu selektieren und mit *next* zu bestätigen. Im Dialog *Runnable JAR File Export* (Abbildung 16) wählen wir die *Launch configuraion* *UfoInvasion*, mit der wir unser Spiel bisher gestartet haben. Neben dem Zielpfad *trunk / UfoInvasion.jar* kann noch ein Ant¹⁷ Script gespeichert werden, mit der sich dieser Exportprozess später automatisieren lässt. Den Exportprozess starten wir nun mit *Finish*. Eine Warnmeldung „*JAR export finished with warnings. See details for additional information.*“ kann ohne Bedenken abgeknickt werden. Sie weist darauf hin, dass im Quellcode noch Warnungen wie beispielsweise überflüssige Importanweisungen vorhanden sind. Nach dem Export liegen im *trunk* Ordner eine neue *UfoInvasion.jar*, die unseren

¹⁶ Die Bezeichnung *trunk* ist eine Konvention bei Projekten, die eine Versionsverwaltung nutzen.

¹⁷ Apache Ant ist ein Tool zum automatischen erzeugen von ausführbaren Programmen aus Quelltext.

kompilierten Quellcode enthält und ein Ordner *UfoInvasion_lib*, in den Eclipse automatisch alle nötigen Slick und LWJGL Bibliotheken kopiert hat.

Damit das Spiel auch Zugriff auf die Ressourcen hat, müssen wir noch den Ordner *res* in unseren Ordner *trunk* kopieren. Als letzten Schritt bevor sich das Spiel starten lässt, muss noch der Inhalt des Ordners *natives* in den *trunk* kopiert werden. Um zu testen ob der Export erfolgreich war, starten wir unsere Spiel aus dem Windows Explorer heraus mit einem Doppelklick auf die *UfoInvasion.jar*. Dass die *natives* neben der *.JAR* nicht extra angegeben werden müssen, liegt daran, dass Windows standardmäßig im dem Verzeichnis aus dem ein Programm gestartet wird auch nach *.DLL*-Dateien sucht.

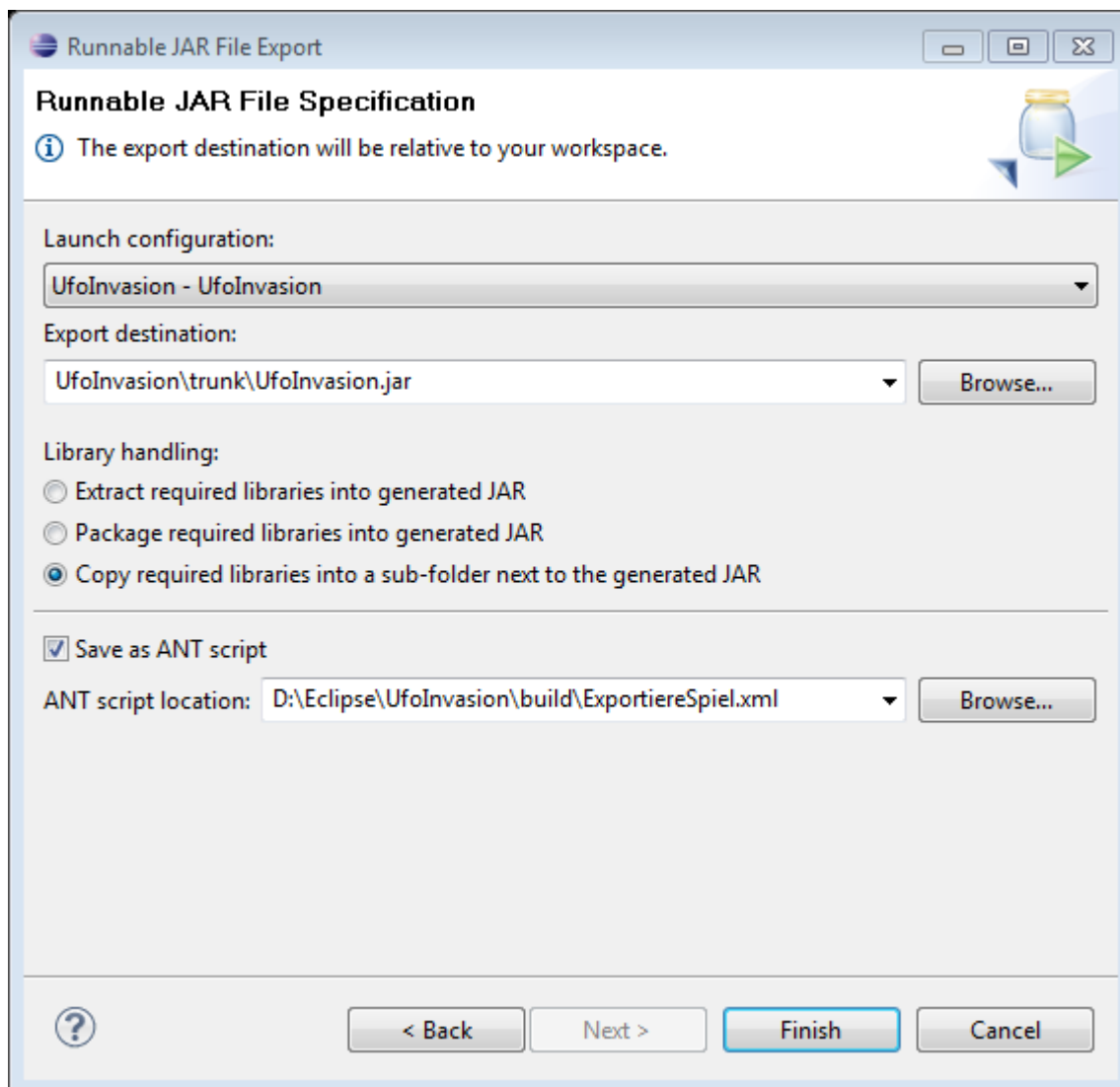


Abbildung 16 Export eines ausführbaren Java Archivs

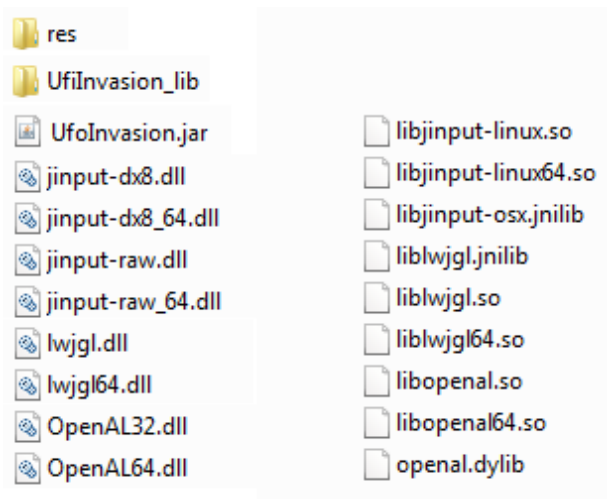


Abbildung 17 Lauffähiges Spiel nach dem Export

Linux den Pfad weissen

Unter Linux muss der Pfad zu den natives extra angegeben werden. Dies geschieht mit folgendem Zusatz beim Aufruf:

`-Djava.library.path=.`

Der Punkt am Ende steht für das *Workig Directory*. Dieses setzen wir über den Befehl

```
cd `dirname $0`
```

auf das aktuelle Verzeichnis. Beides Kopieren wir in ein Skript namens *UnfoInvasion.sh*. Die gesamte *UnfoInvasion.sh* muss dann so aussehen:

```
#!/bin/sh
```

```
cd `dirname $0`
```

```
java -Djava.library.path=. -jar UnfoInvasion.jar
```

Damit ein Doppelklick auf die *UnfoInvasion.sh* auch das Script ausführt, muss unter Linux die Datei noch als ausführbar gesetzt werden. Über die Konsole geschieht dies mit:

```
chmod +x UnfoInvasion.sh
```

Unter Ubuntu reicht alternativ ein Häkchen bei *ausführbar* im Rechtsklickmenü der *.sh* Datei.

Eine exe für Windows Nutzer

Die *UfoInvasion.jar* funktioniert zwar, sieht unter Windows mit dem Java Icon aber nicht gerade nach einem Agrade Spiel aus. Ist auf dem Computer auf dem das Spiel obendrein kein Java installiert, zeigt der Explorer nur ein weißes Blatt für einen unbekannten Dateityp an. Die Lösung: Eine ausführbare Exe für das Spiel.

Das kostenlosen Tool *Launch4j* erstell aus *.jar* Dateien eine solche Exe. Es kann hier herunter geladen werden: <http://launch4j.sourceforge.net>.

Die Einstellungen sind wie in Abbildung 18 vor zu nehmen.

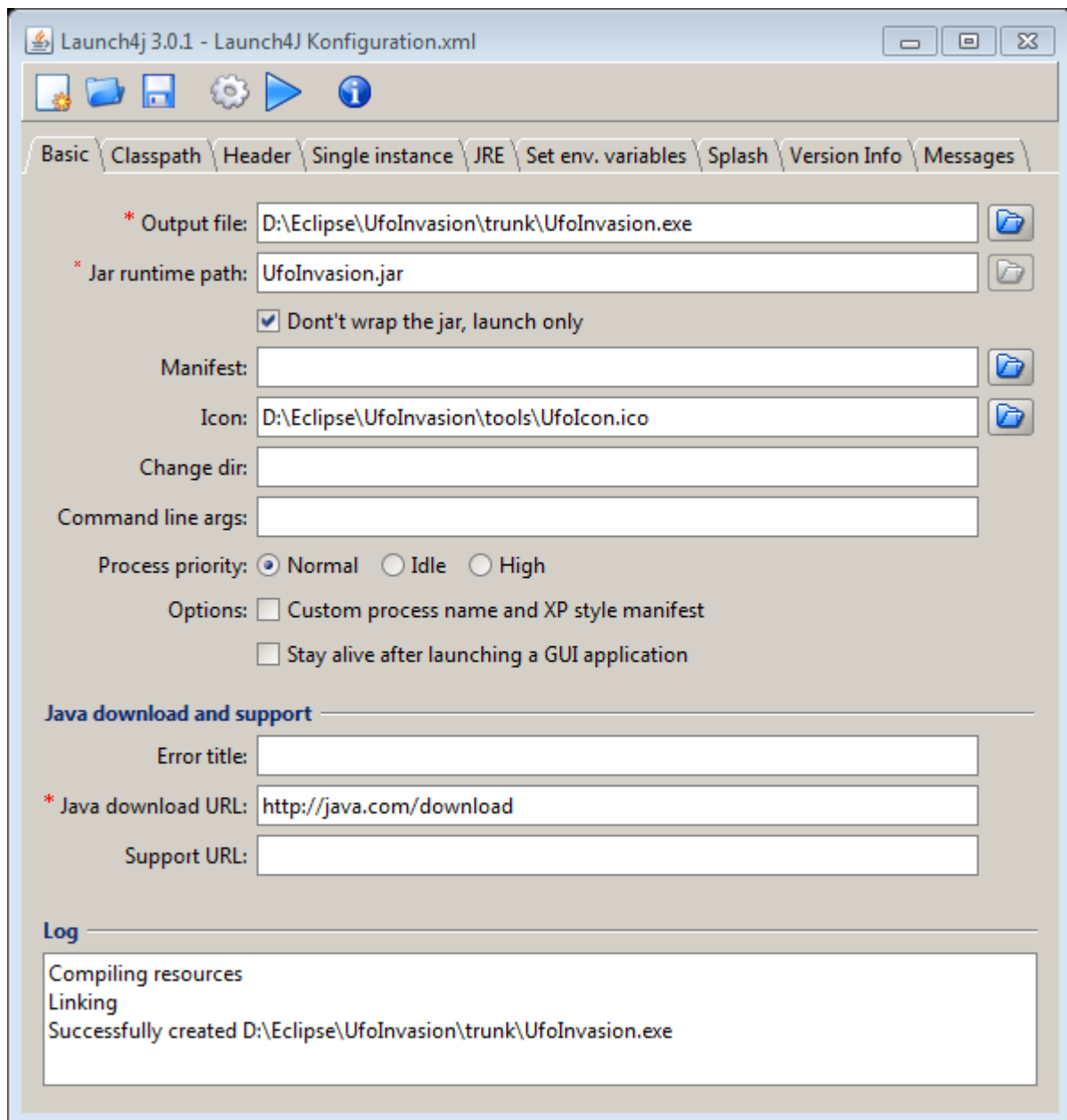


Abbildung 18 Launch4J erstell aus einem Java Archive eine ausführbaren Exe

Statt unsere *UfoInvasion.jar* in die exe auf zu nehmen, sorgen wir mit *Don't wrap the jar, launch only* dafür, dass diese nur gestartet wird. Die .Jar wird also immer noch benötigt und lässt ich auch austauschen, ohne dass eine neue exe werden muss. Im Reiter *JRE* ist dann noch unter *Min JRE version* eine minimale Java Versionsnummer ein zu tragen: „1.5.1“.

Das Icon liegt im Ordner *tools*. Es wurde mit Photoshop gezeichnet und als PNG gespeichert. Diese lässt sich dann mit der Freeware *AveIconifier2* in eine *.ico* umwandeln. Das Programm und eine Anleitung dafür sind hier zu finden:

http://www.vistaico.com/how_to_convert_png_to_ico.htm

Spiel Ressourcen

In den unten aufgeführten Links sind Anlaufstellen um bei der Spieleprogrammierung schnell an Grafiken und Sounds zu gelangen. Die ausgewählten Seiten boten bei der Aufnahme in diese Liste ihre Inhalte kostenlos zur Verfügung. Diese Websites unterliegen der Haftung des jeweiligen Betreibers und haben ihre eigenen Nutzungsbedingungen, die zu beachten sind!

Fonts

- 1001FreeFonts.com
- Dafont.com
- FontyYükle
- Myfont
- OpenFontLibrary
- SimplyTheBestFonts
- TheLeagueOfMoveableType

Icons

- Crystal Clear
- Iconaholic
- IconArchive
- Open Icon Library
- Silk Icons

Music

- ccMixer
- Freesound
- Jamendo
- OpenMusicArchive
- OpSound

Sounds

4. Flashkit
5. Simplythebest Sounds

Sprites

- Game Sprite Archives
- Spriters-Resource
- The ShyGuy Kingdom

Texturen

- TexturenLand

Quellen

- <http://slick.cokeandcode.com/index.php>
- <http://slick.cokeandcode.com/wiki/doku.php>
- <http://slick.javaunlimited.net/>
- <http://www.lwjgl.org/index.php>
- <http://www.opengl.org/>
- <http://connect.creativelabs.com/openal/default.aspx>