

Medienprojekt Dokumentation

“Slack'n'Hay”

Hack'n'Slay Spiel für Android

von Tilman Börner, Jan Rabe und Tom Wallroth

Ein kleines Vorwort

Unser Medienprojekt ist ein „Hack'n'Slay“-Spiel für Android. Da wir für diese mobile Plattform entwickelten stellten sich ungeahnte Problemstellungen und Herausforderungen, auf die wir in dieser Dokumentation näher eingehen wollen. Da Android Apps in Java geschrieben werden, haben wir keine neue Programmiersprache lernen müssen, allerdings haben wir uns sehr darin vertiefen müssen, wie die Java-VM (insbesondere die Dalvik-VM von Android) funktioniert. Zudem mussten wir uns mit der Android-SDK vertraut machen, welche die von der Dalvik-VM verlangten „DEX“-Files erzeugt. Diese sind eine Variante der Java Class-Files und können von der Dalvik-VM ausgeführt werden.

Im Anschluss finden sich weitere Ausführungen über die Architektur des Spiels und unsere Umsetzung einzelner Komponenten.

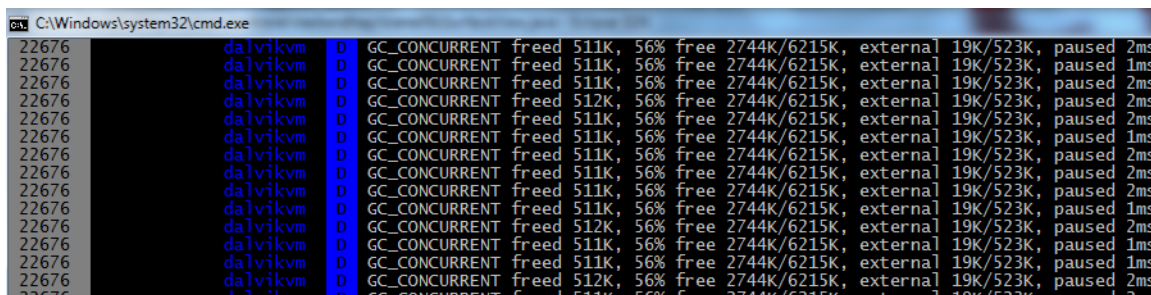
Die Dalvik-VM und ihre Eigenheiten

Google entwickelte die Dalvik-Virtual-Machine für die Android Plattform, sodass Java besonders effektiv auf mobilen Systemen mit wenig RAM (Minimalanforderung 64MB) und einem recht schwachen ARM-Prozessor (minimum ~450MHZ) laufen kann. Das bedeutet, dass die von Java eigentlich erzeugten Class-Files, wie oben schon beschrieben, einfach zu groß sind und zu viele Redundanzen enthalten.

Der Garbage Collector

Daraus ergibt sich, dass zum Beispiel der Garbage Collector, anders als bei der normalen Java-VM nur ab und an aufgerufen wird, da er die komplette VM zum Stop bringen muss. Durch das schwache System braucht der Garbage Collector dabei auch einige Zeit, um alle nicht referenzierten Objekt freizugeben. Da wir allerdings nicht nur eine simple App schreiben, bei welcher es nicht besonders tragisch wäre, wenn sie mal einen kleinen Aussetzer hat, sondern ein Spiel, welches möglichst zu jeder Zeit komplett flüssig laufen muss, stellten sich uns damit neue Herausforderungen.

Für uns bedeutet das, dass wir alle benötigten Ressourcen (Variablen, Objekte, Bilder, Sounds usw.) direkt zum Start des Spiels allozieren müssen, und alle nicht mehr verwendeten Objekte aufbewahren um sie später wieder benutzen zu können. Durch diese Taktik verhindern wir, dass der Garbage Collector der Dalvik-VM unseren Spielfluss stört, da dieser keine Objekte entfernen muss.



```
C:\Windows\system32\cmd.exe
22676 dalvikvm D GC_CONCURRENT freed 511K, 56% free 2744K/6215K, external 19K/523K, paused 2ms
22676 dalvikvm D GC_CONCURRENT freed 511K, 56% free 2744K/6215K, external 19K/523K, paused 1ms
22676 dalvikvm D GC_CONCURRENT freed 511K, 56% free 2744K/6215K, external 19K/523K, paused 2ms
22676 dalvikvm D GC_CONCURRENT freed 512K, 56% free 2744K/6215K, external 19K/523K, paused 2ms
22676 dalvikvm D GC_CONCURRENT freed 511K, 56% free 2744K/6215K, external 19K/523K, paused 2ms
22676 dalvikvm D GC_CONCURRENT freed 511K, 56% free 2744K/6215K, external 19K/523K, paused 1ms
22676 dalvikvm D GC_CONCURRENT freed 511K, 56% free 2744K/6215K, external 19K/523K, paused 2ms
22676 dalvikvm D GC_CONCURRENT freed 511K, 56% free 2744K/6215K, external 19K/523K, paused 2ms
22676 dalvikvm D GC_CONCURRENT freed 511K, 56% free 2744K/6215K, external 19K/523K, paused 1ms
22676 dalvikvm D GC_CONCURRENT freed 512K, 56% free 2744K/6215K, external 19K/523K, paused 2ms
22676 dalvikvm D GC_CONCURRENT freed 511K, 56% free 2744K/6215K, external 19K/523K, paused 1ms
22676 dalvikvm D GC_CONCURRENT freed 511K, 56% free 2744K/6215K, external 19K/523K, paused 1ms
22676 dalvikvm D GC_CONCURRENT freed 512K, 56% free 2744K/6215K, external 19K/523K, paused 2ms
22676 dalvikvm D GC_CONCURRENT freed 511K, 56% free 2744K/6215K, external 19K/523K, paused 2ms
```

Ausgabe der Log Konsole: der Rasende Garbage Collector

Um das zu erreichen, haben wir ein von Google bereitgestelltes Debugtool (Dalvik Debug Monitor Server) benutzt, welches es mit dem "Allocation Tracker" ermöglicht Mitschnitte der von der VM getätigten Allozierungen zu machen. Diese Mitschnitte konnten wir dann analysieren um herauszufinden, an welcher Stelle in unserem Code Objekte erzeugt werden, um die Erzeugung neuer Objekte möglichst zu verhindern. Checkliste der benötigten Codeänderungen:

- Alle in einem Objekt benötigten Variablen werden bei dessen Erzeugung angelegt
- Alte Objekte müssen wiederverwendet werden können (eine "tote" Spielfigur kann später woanders wieder erweckt werden)
- Alle variablen sind final
- keine Collections benutzen
- keine for-each Loops (möglicherweise selbst die Zählvariable schon vorher allozieren)
- Alle String-Manipulationen sind böse
- Texturen nur einmal laden und binden
- Vertice-, Texture-, Normalkoordinaten, sowie Index-, und Colorwerte einmal beim Start allozieren und während der Laufzeit nur die IDs der Buffer den Objekten zuweisen (Vertex Buffer Objects)

Über "Slack'n'Hay"

Wir wollten ein möglichst einfaches Spielprinzip für eine noch unbekannt Plattform umsetzen, wobei sich recht schnell herausgestellt hat, dass selbst dieses einfache Spielprinzip viele Tücken in sich birgt. Die wichtigsten Konzepte hinter dem Spiel werden wir in diesem Kapitel näher beschreiben.

Das Grid

Das Grid ist die räumliche Grundlage des Spiels; es stellt eine zweidimensionale Fläche dar, auf der einzelne Spielobjekte positioniert werden können.

Um Dinge wie Positionierung, Kollisionsabfragen und Pathfinding zu vereinfachen, teilt das Grid einen Teil der theoretisch verfügbaren, kontinuierlichen Fläche in einzelne, gleich große Zellen auf, die jeweils höchstens ein Spielobjekt enthalten können. Zwischen diesen Zellen bestehen *Nachbarschaftsbeziehungen*, die durch *Richtungen* konkretisiert werden; das heißt, eine Zelle kann z.B. der nördliche Nachbar einer anderen sein.

Während der kontinuierliche Raum also endlos ist, sind die Abmessungen des Grids festgelegt. Das Grid begrenzt somit den für das Spiel nutzbaren Raum.

Alle Punktkoordinaten innerhalb einer Zelle sind identisch mit dieser; umgekehrt sind die Koordinaten des Mittelpunkts der Zelle deren Normalkoordinaten, die als die Position eines eventuell enthaltenen, ruhenden Spielobjekts gelten. Das Grid übernimmt dabei die Abbildung von Grid-Raum (Zellen) zu kontinuierlichem Raum und umgekehrt.

Dadurch kann die interne Geometrie der Zellen flexibel gestaltet werden, ohne dass das Auswirkungen auf andere Elemente des Spiels hat: zum einen können die Zellen z.B. quadratisch sein oder auch sechseckig, woraus sich entsprechende Richtungen ergeben; zum anderen lässt sich die Zellengröße im Verhältnis zum kontinuierlichen Raum beliebig skalieren.

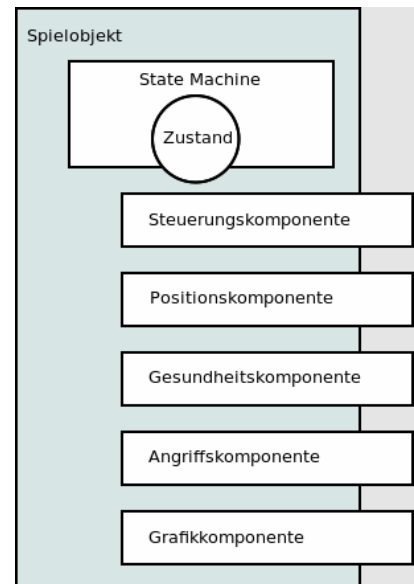
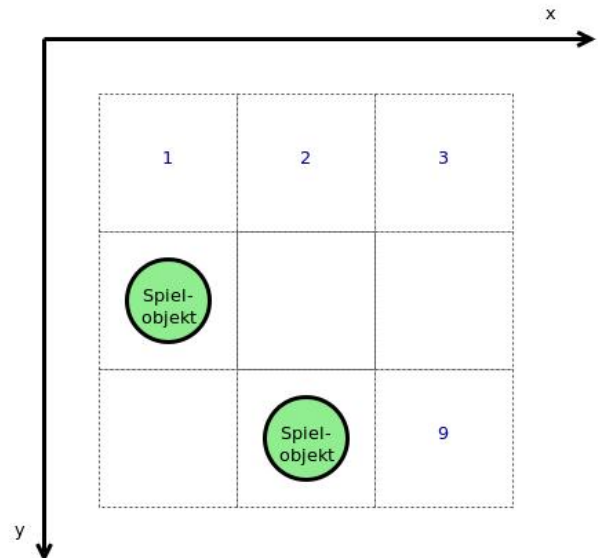
Definierende Parameter des Grids sind damit die vertikale und horizontale Anzahl der Zellen, ihre Größe, sowie die realen Koordinaten des Punkts, an dem die erste Zelle im kontinuierlichen Raum verankert ist. Die Positionen der restlichen Zellen und die gesamte Ausdehnung des Grids ergeben sich daraus.

Das Interface zu diesem Grid wird durch die Positionskomponente der Spielobjekte dargestellt. Weiteres zum Thema der Spielobjekte und ihrer Komponenten findet sich im folgenden Kapitel.

Die Gameobjects und ihre Komponenten

Die elementaren Bestandteile des Spiels sind zweifellos die Spielobjekte; damit sind vor allem alle Spielelemente gemeint, die vom Spieler als atomare Einheiten wahrgenommen werden, also seine eigene Spielfigur, Gegner, Häuser und so weiter.

Ihre Beteiligung am Spiel lässt sich in verschiedene Aspekte gliedern: sie besitzen eine Position im Raum, verschiedene grafische und auditive Darstellungen, vielleicht Eigenschaften wie Gesundheit, Angriffs- und Verteidigungsmöglichkeiten, und sie zeigen ein bestimmtes Verhalten.



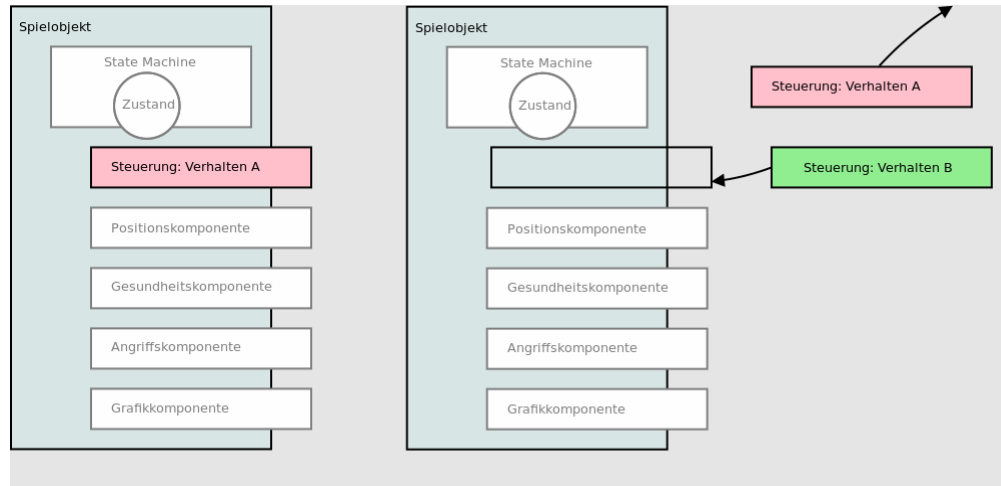
Die zeitliche Dimension wird über eine State Machine realisiert, von der jedes Spielobjekt eine eigene besitzt. Der State Machine haben wir ein gesondertes Kapitel gewidmet.

Wir haben uns entschieden, diese Aspekte so modular wie möglich zu realisieren. Das geschieht in Form von sogenannten Komponenten, die in ihren verschiedenen Arten für einzelne Aspekte zuständig sind. In ihren konkreten Implementierungen können sie so diese Aspekte auf verschiedene Arten umsetzen; so lassen sich aus einzelnen Komponenten verschiedene Typen von Spielobjekten zusammensetzen.

Die einzelnen Komponenten können dabei auf die Dienste anderer Komponenten zugreifen: so bedient sich beispielsweise die Grafikkomponente bei der Positionskomponente um zu ermitteln, an welcher Stelle das Spielobjekt zu zeichnen ist.

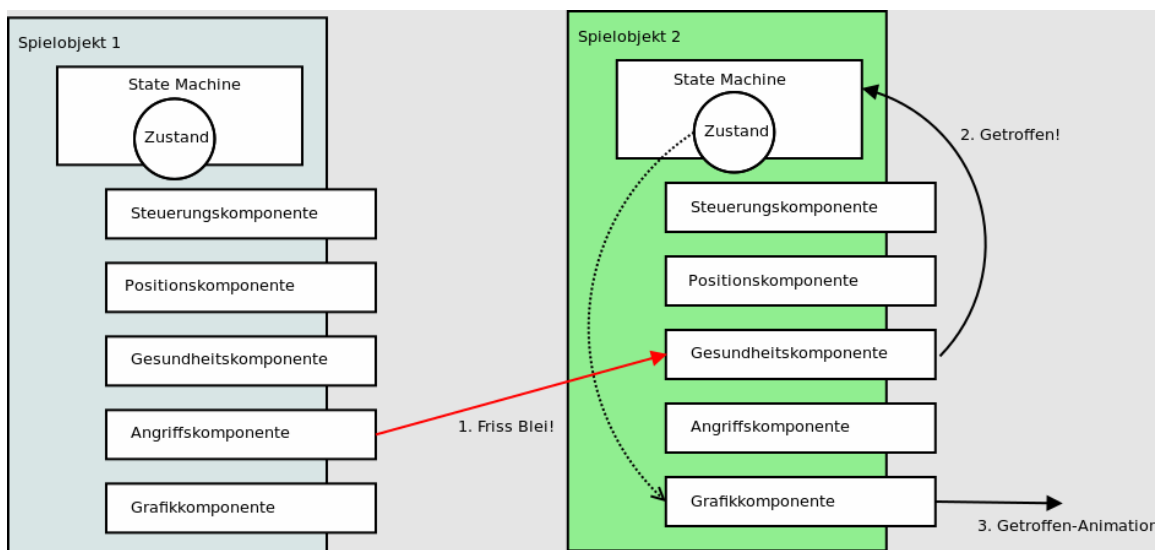
Weiterhin können sich die Komponenten in ihrer Funktion nach den durch die State Machine verwalteten Zuständen richten, sowie umgekehrt der

State Machine Vorschläge für Zustandsübergänge machen; Einzelheiten dazu finden sich im Kapitel über die State Machine.



Modularität von Komponenten

Die Interaktion zwischen einzelnen Spielobjekten findet ebenfalls über ihre Komponenten statt. Wenn beispielsweise Spielobjekt A ein anderes Spielobjekt B angreift, so wird *Angriffskomponente A* versuchen, *Verteidigungskomponente B* anzugreifen, die dann nach ihrem Ermessen *Gesundheitskomponente B* Schaden zufügen und eventuell einen *Getroffen*-Zustand vorschlagen kann.

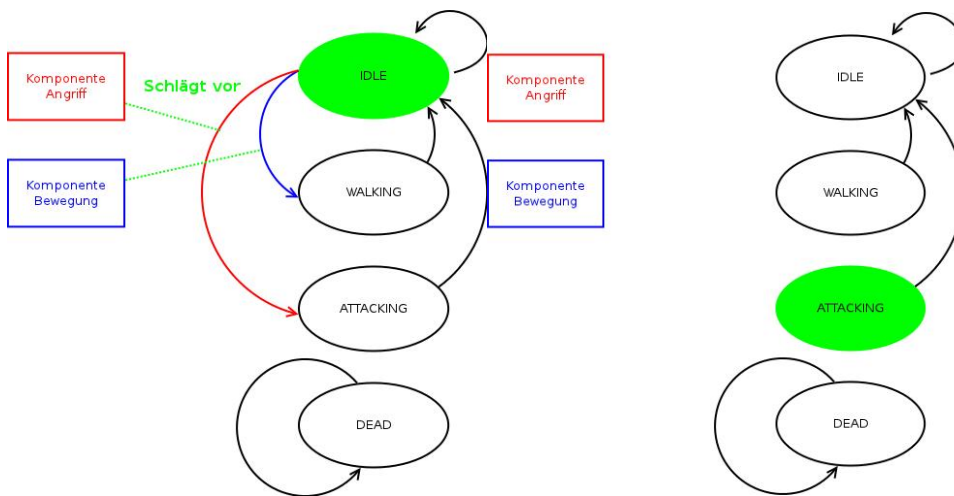
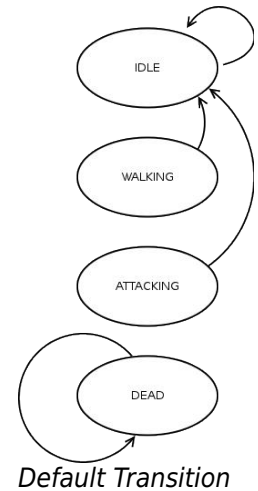


komplexe Interaktion zwischen zwei Spielobjekten

Die Statemachine

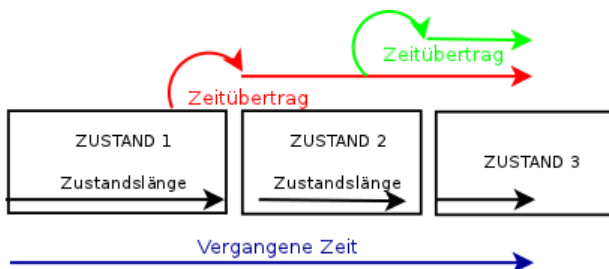
Jedes Gameobject muss eine Statemachine besitzen, sodass das Spiel überhaupt voranlaufen kann, also Zeit verstreichen kann. Das bedeutet, dass bei jedem Durchlauf der Gameloop alle Gameobjects die gerade verstrichene Zeit erhalten und diese ihren Statemachines mitteilen. Eine Statemachine ist im Grunde genommen ein DFA, welcher durch zwei Faktoren zu Zustandsübergängen bewegt wird.

1. Durch Zeit: Nach dem Ablauf einer bestimmten Zeit wird die sog. "default transition" durchgeführt. Diese "default transition" sorgt dafür, dass jeder Zustand nur für eine festgelegte Zeit aktiv sein kann, und danach aufhört, bzw wieder von vorne beginnt.
2. Durch Vorschläge: Zustände können von den Komponenten der Gameobjects vorgeschlagen werden, dass heißt, dass z.B. die Movementkomponente dem Gameobject vorschlagen kann in den "Walking" Zustand zu wechseln. Dabei muss natürlich beachtet werden, dass diese Vorschläge eine Piorisierung besitzen müssen, sodass sich die Statemachine immer in einem konsistenten Zustand befindet. Im Folgenden wird der Ablauf bei zwei vorgeschlagenen Zuständen dargestellt, wobei die Zustände, die weiter unten im Diagramm liegen, eine höhere Piorität besitzen.



Da der "Attacking"-Zustand eine höhere Piorität hat, hat er vorrang vor dem "Walking" Zustand.

Zusätzlich hat die Statemachine die Möglichkeit einen Zeitübertrag bei Zustandsübergängen mitzugeben, sodass eine niedrigere Framerate nicht dafür sorgt, dass das Spiel langsamer läuft.



Editor

Damit die Elemente leichter zu positionieren sind, haben wir einen Welt Editor mit Java Swing gebaut, welcher die Grundlegenden Funktionen besitzt Spielelemente, Bodentexturen, Kamera und die Spielreposition zu hinzuzufügen, sowie diese via Drag & Drop verschieben zu können. Außerdem kann die Weltgröße definiert werden. Die Welt wird in einer XML Datei gespeichert und von der Applikation geparkt.

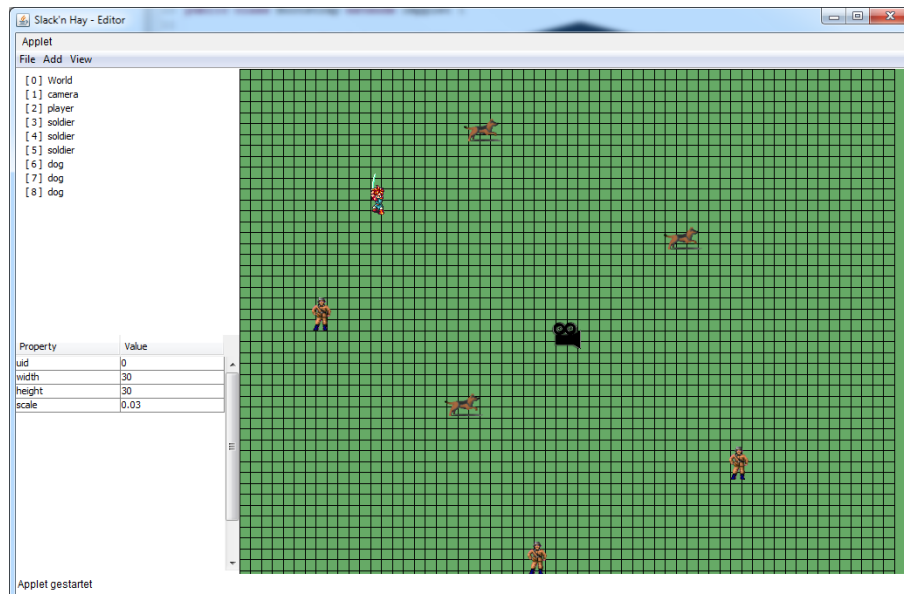
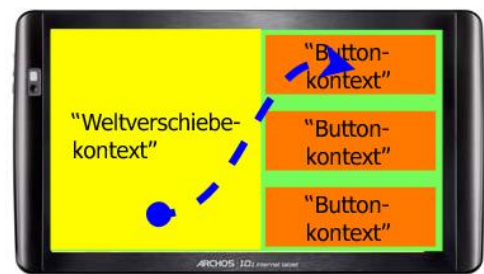


Abbildung 1: Welt Editor

Der Touchscreen

Zur Ansteuerung des Touchscreens mussten wir die Android API nutzen, welche es ermöglicht mehrere Zeiger (also Finger) gleichzeitig zu erfassen. Das erforderte eine Umgewöhnung, da wir durch die Entwicklung auf Desktoprechnern einfach mit einer Mausposition rechnen können, wobei die Maus auch immer vorhanden sein und eine Position besitzen muss. Beim Touchscreen müssen mehrere Finger erkannt und einem Kontext zugeordnet werden. Wir legen den Kontext fest, sobald ein Finger den Touchscreen zum ersten mal berührt; Falls der Finger den Bildschirm in der linken Bildschirmhälfte berührt, wird dieser Finger als Input zum verschieben der Welt interpretiert, auf der Rechten Seite würde er einem Knopfdruck entsprechen. Diese Kontextfestlegung und -abhängigkeit erlaubt es, dass der Benutzer z.B. auf der linken Seite beginnt die Welt zu verschieben ohne mit dem selben Finger versehentlich einen Button auf der rechten Seite zu drücken, selbst wenn er den Finger in die rechte Bildschirmhälfte bewegt.

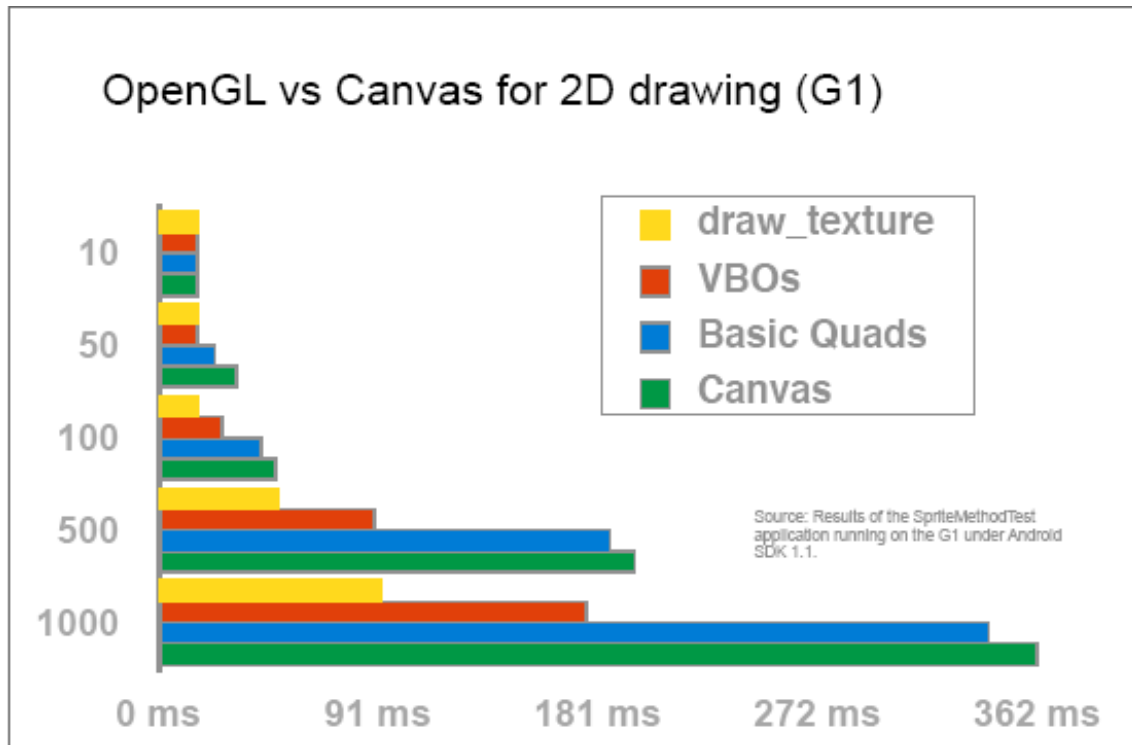
Darüberhinaus können diese Kontexte jetzt genauer spezifiziert werden: man könnte erlauben, dass ein bestimmter Button einen Gesten-Kontext hervorruft, welcher eine Geste mit dem Finger in eine bestimmte Aktion wandeln würde.



Der Kontext der ersten Berührung bleibt erhalten

Hardware-beschleunigte 3D Anzeige

Seit Android 1.5 wird OpenGL ES (Open Graphic Library for Embedded Systems) von Android unterstützt. Da die Systemressourcen von Android sowieso schon rar sind, haben wir uns entschieden das Rendering des Spiels auf nativen Code und den Grafikchip zu verlagern. Im Gegensatz zum direkten Zeichnen auf einen Canvas haben wir durch die Verwendung von OpenGL ES einen großen Performanceboost erhalten. Folgendes Diagramm veranschaulicht dieses Verhalten. Es gibt vier Möglichkeiten etwas zu zeichnen, wobei alles über 100 Objekte nur noch mit OpenGL sinnvoll darstellbar sind.



Quelle: Chris Pruett <http://replicaisland.net>

Trotzdem bedeutete die Benutzung von OpenGL ES eine große Umstellung, weil nun natürlich auch dieselben oben beschriebenen Probleme auftraten. Wir mussten also möglichst alle verwendeten Texturen besonders platzsparend unterbringen. Um das zu bewerkstelligen benutzten wir SpriteSheets. Ein SpriteSheet ist eine große Textur, welche alle benötigten Grafiken enthält. Wenn man nun eine Person auf den Bildschirm rendern möchte, muss man also nur einen Teil der Textur auf den Bildschirm zeichnen (siehe Abb.). Jedoch ist die Findung der Texturkoordinaten relativ umständlich zu realisieren, da OpenGL das UV Mapping clampt, was bedeutet, dass es nur einen Wert zwischen 0 und 1 gibt, egal wie groß die Textur ist, die man lädt. Daraus folgt, dass man die Texturkoordinaten erst einmal von absoluten Texturkoordinaten und Größen aller Einzelbilder in Werte zwischen 0 und 1 umrechnen muss.

Unser Hauptcharakter besteht aus etwa 270 Einzelbildern. Jedoch sind etwa 100 davon nur durch Spiegelung zu bestimmen. Das erspart natürlich den Platz auf dem SpriteSheet, welcher nun anderen Grafiken zur Verfügung steht.



Teil eines unserer Sprite Sheets

VBOs (Vertex Buffer Objects)

Um maximale Performance aus der Applikation zu kitzeln, haben wir VBOs verwendet. Die Idee ist es, alles oft verwendete einmalig mit einer ID zu versehen und in den Grafikspeicher zu laden, bei Bedarf diese IDs den gewünschten Objekten während der Laufzeit nur zuzuweisen. Das bedeutet, dass nur noch einzelne Integer während der Laufzeit an den Grafikspeicher übertragen werden, anstelle alles jedes mal neu zu übertragen.

Mit diesem tollen und gezwungenermaßen notwendigen Feature kommen natürlich auch komplizierte und super schlecht dokumentierte Implementationsschwierigkeiten. Zudem bietet OpenGL auch nur sehr begrenzte Debugmöglichkeiten, sodass man als Entwickler erst einmal im dunklen tappt, um eine erstmalige funktionsfähige Minimalversion zum Laufen zu bringen. Besonders der Android-Emulator ist bei OpenGL völlig überfordert. Er kann seit der Umstellung auf VBOs nichts mehr anzeigen, da diese nicht fehlerfrei im Emulator implementiert wurden.

Um den Anschein einer Spielwelt zu erzeugen, brauchen wir eine Grundfläche und Spielelemente, die darauf stehen. Zuerst haben wir diese Elemente zweidimensional gezeichnet und daraufhin die Kamera leicht angewinkelt. Jedoch sieht dies immer noch nicht aus wie gewünscht, da die Spielelemente wie plattgewalzt aussehen. Es wäre doch viel besser, wenn die texturierten Planes immer zur Kamera ausgerichtet wären.



angewinkelte Kamera

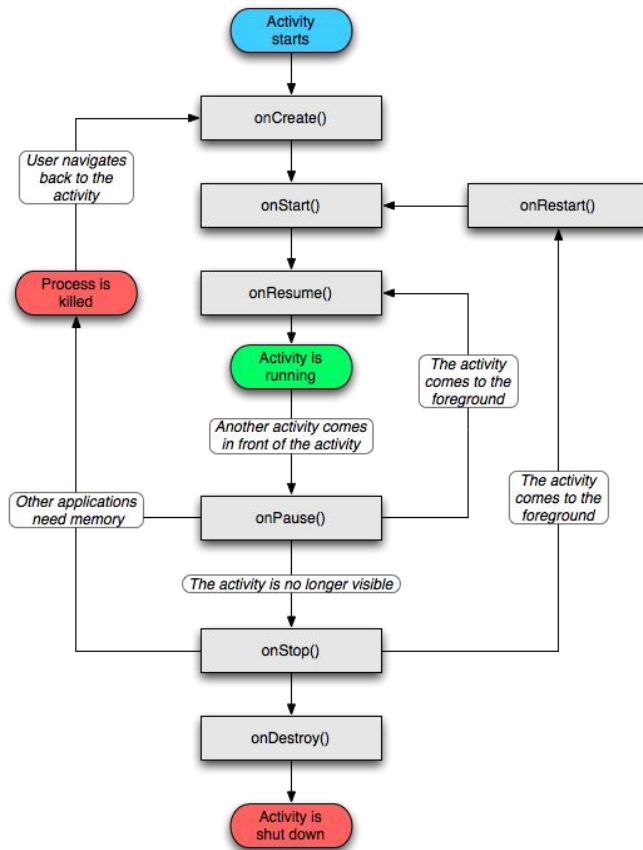
Das haben wir dann natürlich auch umgesetzt. Dadurch sehen die Objekte nun aus, als würden sie im 3D Raum existieren. Das heißt, dass beim Bewegen der Kamera die Abstände und die Größen zwischen den Elemente nach hinten hin kleiner werden, bzw. nach vorn hin größer; außerdem sind Parallaxeeffekte zu sehen.



an Kamera ausgerichtete Spielelemente

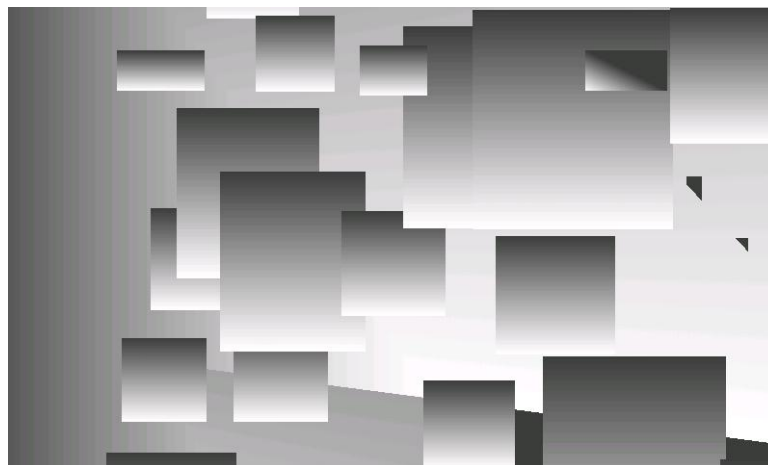
Android: Lebenszyklus einer Applikation

Eine Android Applikation hat verschiedene Phasen. Dabei werden alle Initialisierungen in der onCreate()-Phase abgewickelt. Wenn die Applikation geschlossen wird, so springt diese jedoch nicht wie gedacht in die Phase onStop(), sondern nach onPause() und nach Reaktivierung in onResume(). Das hat zur Folge, dass nicht alle Texturen erneut initialisiert werden.



Activity LifeCycle

Auch ein erneutes Laden führt manchmal zu seltsamen Effekten. Das kommt daher, dass Android sich selbstständig darum kümmert, wenn irgendwo Speicher fehlt, so wird dieser von den laufenden Applikation geholt. Das bedeutet, dass der Entwickler sich selbstständig um das erneute Laden kümmern muss.



willkürlich Speicherentladen: untexturierte Planes