Tutorial CG2LU SS2008

Planung, Design und Umsetzung eines Spieles - Ein Step by Step-Guide

Einleitung

Das folgende Dokument soll zur Unterstützung bei der Absolvierung der CG2LU dienen. Eine identische Kopie dieses Spieles ist keinesfalls erfordert noch gewünscht. Das hier vorgestellte Spiel „Super J" soll auf gängige Fehler und Probleme hinweisen und brauchbare Lösungswege aufzeigen. Der Gesamtumfang des fertigen Spiels geht im Fall Super J weit über das hinaus, was ihr bei CG2LU umsetzen sollt. Wir haben auf eine separat entwickelte Engine und einen Level Editor zurückgreifen können, außerdem hatten wir entsprechend mehr Zeit als die für CG2LU veranschlagten 4 Monate. Ihr könnt hier aber sehen, wie wir einzelne Elemente des Gameplays und des Contents konkret umgesetzt haben, und euch davon inspirieren lassen, wie man mit relativ geringem Aufwand entsprechende Elemente umsetzen kann.
Im folgenden Dokument sind alle Planungen und der Arbeitsaufwand bei der Umsetzung eines Spiels für CG2LU dokumentiert. Es beginnt mit den grundlegenden Gedanken und der Entscheidung welches Spiel hergestellt werden soll, dann wird detaillierter auf den Aufbau des Spiels eingegangen und mit ein paar generellen Tipps zu gutem Game Design abgerundet. Super J dient dabei immer wieder als Beispiel.

Grundlegende Idee

Die grundlegende Idee für ein Spiel kommt erfahrungsgemäß beim Konsumieren von Büchern, Filmen oder anderen Spielen.

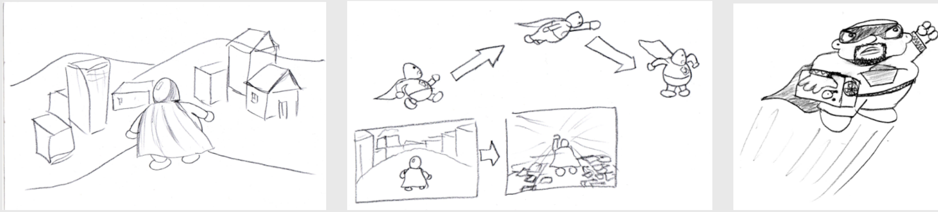
Beispiel Super J:
Wir wollen ein Spiel mit einem etwas verrückten Superhelden machen. Inspiration von Incredibles, Superman, Comics...

Wie hier ersichtlich ist die grundlegende Idee nur **ein einziges Design- oder Gameplayelement**. Es kann ein Setting sein, eine Szene, eine Epoche, eine Stimmung, Charaktere, eine Gegend, eine Situation…

Konzept

Hier wird nun die etwas vage Idee zu konkreteren Formen gebracht. Hilfreich sind hier ein paar Skizzen (auch grausam schlechte Coderart! „Ich kann nicht zeichnen" gilt hier nicht. Außerdem braucht ihr sowieso Skizzen für die 1. Abgabe…) und der Versuch das Spiel in einem Satz zu erklären („Logline").

Beispiel Super J:

- **Logline:** Verrückter persönlichkeitsgespaltener Superheld will die Erde einfrieren.
- **Skizzen:** 

Jetzt solltet ihr euch auf ein **Genre**, den **Präsentationsstil** (ergibt sich zumeist aus der grundlegenden Idee), die **Steuerung**, das **Gameplay** der einzelnen Spielelemente und den **Umfang** festlegen. Hier helfen **Brainstorming-Sessions**, in denen die einzelnen Teammitglieder ihre Ideen und Vorstellungen einbringen. Am Schluss wird sortiert, welche Ideen sich wie miteinander verbinden lassen um ein großes stimmiges Konzept zu erhalten. Es ist an dieser Stelle wichtig, dass alle Teammitglieder dieselbe Vorstellung von der Präsentation und dem Gameplay des Spiels haben! Oft kommt es zu Verwirrungen, gerade wenn man davon ausgeht, dass „eh alles klar" ist.
Normalerweise gibt es jetzt eine Person (den **Game Designer**), der diese Punkte genau festlegt, und an die sich die anderen Gruppenmitglieder zu halten haben. Dadurch vermeidet ihr, dass ihr eine Woche vor der Abgabe auf die Idee kommt, noch ein paar coole Features einzubauen, die euch gerade eingefallen sind, aber viel Zeit kosten oder am Ende das gesamte Gameplay umkrempeln. Das nennt sich „**Feature Creep**" kommt *immer* bei *allen* Teams vor, also stellt euch darauf ein. In der Industrie passiert das auch oft, aber bei CG2LU habt ihr eine fixe Deadline die ihr einhalten müsst, also achtet genau darauf was für Designänderungen sich umsetzen lassen!

Umfang

Überprüft sicherheitshalber noch extra den **Umfang** eures Projekts, da viele Ideen sehr ambitioniert und interessant sind, sich in der kurzen für CG2LU zur Verfügung stehenden Zeit aber nicht umsetzen lassen. Normalerweise benötigt man doppelt so lang für Features wie geplant (das klingt abgedroschen, stimmt aber tatsächlich). Vergesst vor allem nicht die Zeit einzurechnen, die ihr in **Content-Arbeit** (Modelle, Texturen, Sounds…) investieren müsst. Selbst wenn ihr euch entscheidet auf fertigen Content zurückzugreifen, muss der Content erst gefunden, gefiltert und möglicherweise umformatiert werden, bis er benutzbar ist.
Daher ist es besonders wichtig, euch von Anfang an zu überlegen welche Dinge sicher *nicht* umgesetzt werden, den Umfang eures Spiels also **klar einzugrenzen** (z.B. „wir wollen Split-Screen, aber keinen Online-Multiplayer").

An dieser Stelle könnt ihr euch auch schon erste Gedanken machen, ob ihr **zusätzliche Libraries** z.B. für Physik benötigt, und wie ihr im Entwicklungsverlauf den **Content** erzeugen und in euer Spiel bringen wollt. Das beeinflusst die Arbeit der nächsten Monate entscheidend, und sollte daher möglichst früh entschieden werden. Besonders wichtig ist es hier, auf die **Level-Design-Pipeline** zu achten. Viele Gruppen planen nur die Asset Pipeline für Modelle und Texturen, vergessen aber, sich zu überlegen, wie Level-Daten ins Spiel kommen sollen! Denkt daran dass ihr kaum die Zeit haben werdet ein Level Design-Tool selbst zu programmieren, greift daher lieber auf bestehende Lösungen zurück.

Effekte

Die für CG2LU geforderten Effekte ergeben sich normalerweise von selber **aus dem Spielkonzept**. Einige Effekte wie Partikelsysteme oder eine Schattentechnik (das können auch simple projektive Schatten sein) lassen sich in fast jedes Spiel sinnvoll einbinden, und sind für einen ästhetischen Gesamteindruck nicht zu vernachlässigen.
Reiht die Effekte gleich nach der **Wichtigkeit** für das Spielkonzept. Manche Effekte sind für das Gameplay unerlässlich (z.B. Schatten in einem Stealth-Spiel), andere sind der Stimmung und dem Spielgefühl zuträglich, aber es gibt vielleicht auch unnötige aufgesetzte Effekte (z.B. ein schwebender gebumpmappter Cube in der Mitte des Levels), die ihr vermeiden solltet. Zieht eventuell nochmal das Konzept zu Rate, ob und wie ein Effekt dem Spielgefühl zuträglich ist!

Beispiel Super J:

- **Genre:** Freeroaming Action-Adventure.
- **Stil:** Comicstil, verwackelte Hütten, große einfärbige Flächen, verfremdete Realität.
- **Gameplay:** Mission erfüllen (Sammeln von X Objekten, Töten von X Gegnern…). Möglichst freie Bewegung (laufen, fliegen, fahren…). Objekte sammeln für Level-Ups.
- **Steuerung:** 3rd Person, Tastatur für Bewegung, Maus für Umschauen und Zielen.
- **Content:** Selbstproduzierter Content für kohärenten Stil. Modelle mit 3DS Max und Maya, Import über Ogre XML. Welt wird mit separatem Level Editor erzeugt (steht bereits zur Verfügung).
- **Externe Libs:** DevIL für Texturen, OpenAL für Sound, Newton für Collision Detection.

Super J-Konzept:

Um die Lehrveranstaltung positiv zu absolvieren wird eine 3D-Umsetzung des Konzepts angestrebt. Bei Freeroaming mit Fliegen bietet sich das sowieso an. Es ist klar, dass die Steuerung auf die Klassiker WASD und Mouse-Look setzt, sowie ein einheitliches, großes Level erforderlich ist. Der Level wird mit einem separaten Tool erzeugt. Der Spieler hat in dem Level Missionen zu erfüllen.
Da wir eine komplette Stadt umsetzen wollen, erfordert das eine relativ große Anzahl an Modellen, die sicherlich nicht in einem einheitlichen (und von uns gewünschten) Stil einfach im Internet zu bekommen sind. Wir streben keinen realistischen Look an, sondern passend zum Spielthema einen Comiclook. Das erlaubt es uns selbstgemachte Low-Poly-Modelle mit simplen einfarbigen Texturen zu benutzen. Um den Arbeitsaufwand weiter zu beschränken wird massiv auf das Wiederverwenden von Content gesetzt.
Im Gegensatz zu anderen Freeroaming-Spielen wie GTA wird es keine Cut-Scenes und Dialogsequenzen geben, die Missionsbeschreibung erfolgt ausschließlich in Textform. Passanten und funktionstüchtige Fahrzeuge werden ebenfalls ausgelassen. Der Einfachheit halber laden wir die ganze Welt beim Start in den Speicher und vermeiden damit Streaming, was sehr aufwändig zu implementieren wäre.
Für die Modelle verwenden wir wahlweise 3DS-Max oder Maya und importieren über das „Ogre XML"-Format. Das Terrain wird über eine Heightmap im Level Editor importiert, die verschiedenen Bodentypen (Gras, Erde…) werden automatisch nach Winkel und Höhe generiert. Das Straßennetz wird aus einer separaten Bilddatei importiert. Zum Platzieren von Objekten in der Welt verwenden wir ebenfalls den fertigen Level Editor, den wir für ein Praktikum bereits programmiert haben. Alternativ wäre es auch möglich gewesen die Objekteigenschaften und -positionen aus Textdateien zu laden. Bei der großen Anzahl an Objekten und dem WYSIWYG-Interface bietet sich hier aber natürlich der Editor an. Die Missionsinformationen werden aus separaten Textdateien importiert, die Missionsbeschreibungen sind einfache Bilddateien.

Super J-Effektliste:
Aus dem Konzept ergibt sich nun fast von alleine die Effektliste:

- **Projektive Schatten:** Objekte werfen Schatten auf das Terrain, um sie in der Welt zu verankern, und damit der Spieler immer weiß, wo er sich befindet, wenn er fliegt.
- **Partikelsysteme:** Explosionen, Power-Up-Effekte, Schusstreffer…
- **Wasser:** Wird als animierter Mesh mit darüberrollender Textur implementiert.
- **Radialer Motionblur:** Wenn der Spieler fliegt oder schnell fährt verschwimmt die Umgebung in einem Tunnelblick. Das wird über einen simplen radialen Blur ausgehend von der Bildschirmmitte erzielt.
- **Per-Pixel-Lighting mit Specular Highlights:** In der Glatze des Hauptcharakters spiegelt sich die Sonne. Da der Charakter ständig in Bewegung ist kommt dieser Effekt gut zur Geltung.
- **Postprocess-Effekte:** Wenn der Spieler Health verliert, so verblassen die Farben. Bei kritischer Health beginnt der Bildschirm außerdem rot zu pulsieren. Wird der Spieler getroffen, so leuchtet der Bildschirm rot auf und der Kontrast wird verstärkt.

Die meisten Gruppen kommen in ihrer Effektliste locker über die geforderte Effektpunktezahl. Daher empfiehlt es sich die **Effekte nach Priorität zu ordnen**, um bei Zeitmangel Effekte zu streichen, ohne den Gesamteindruck des Spiels drastisch zu verschlechtern. In unserem Fall sind Schatten und Parikelsysteme die primären Effekte. Motionblur verstärkt den Spielspaß beim Fliegen und Fahren sicherlich, ist aber nicht essentiell. Perpixelshading und Postprocess-Effekte sind als Eyecandy einzustufen und deshalb tertiär. Schließlich ließe sich ein Health-Indikator als normaler HUD-Element implementieren, oder auch als einfaches rotes halbtransparentes Quad, das über den Bildschirm gelegt wird (und natürlich kein Render-to-Texture benötigt).

Wenn ihr diese Ideen nun alle in ein Dokument verwandelt, dann ist euch die volle Punktezahl für die 1. Abgabe schon so gut wie sicher!

Umsetzung

Nun beginnt der ernsthafte Teil der Arbeit und es sollten die ersten konkreten Gedanken über die technische Realisierbarkeit und über die Organisation des Projektes gemacht werden. Ganz wichtig ist es, früh genug nachzudenken, welche Libs verwendet werden, und diese bezüglich Funktionalität und Interface zu evaluieren, um schnell mögliche Probleme bei der Integration zu erkennen.

Collision Detection

Collision Detection macht einen wichtigen Teil in der Implementierung fast aller Spiele aus, und ist **nicht trivial**! Daher solltet ihr euch rechtzeitig überlegen, wie ihr das handhabt.
Überlegt euch, ob ihr eine eigene Collision Detection implementieren wollt, oder auf fertige Physikengines zurückgreifen wollt. Generell besteht Collision Detection nicht nur aus dem Auffinden von Intersections (**Detection**), sondern auch mit dem Behandeln dieser Probleme (**Response**).
Normalerweise ist gerade die *Response* sehr aufwändig und problematisch selbst zu programmieren, da man hier sehr leicht auf Probleme mit der **Float-Genauigkeit** stößt, was dazu führen kann dass Objekte langsam durch anderen durchgleiten oder herumzittern.
Solltet ihr also tatsächlich eine **komplette 3D-Collision Detection** benötigen, bietet sich tatsächlich eine fertige Lösung an. Bei Physikengines müsst ihr aber darauf achten, wie gut sie mit vom Spieler gesteuerten Characters zurechtkommen. Beim Spieler wollt ihr ein langsames Beschleunigen und

Weiterschlittern im Normalfall eher vermeiden. Eventuell könnt ihr auch nur den Collision Detection-Teil einer Physikengine verwenden, ohne die Physiksimulation zu benutzen?

Sehr viele Collision Detection-Probleme lassen sich aber einfach auf **2.5D** oder überhaupt auf **2D herunterbrechen** und sind damit sehr einfach zu lösen (z.B. wenn der Spieler nicht springen kann, ist eine vollständige 3D-Kollision für die Spielerbewegung unnötig). Oft reicht es um die einzelnen Objekte in der Welt simple **Bounding Primitives** (Kugeln, AA-Boxen, Zylinder) zu legen, mit denen die Kollision trivial zu berechnen ist. Vollständige Dreieckskollision ist nur in seltenen Fällen wirklich nötig. Auch in diesem Fall reicht es oft aus, „Proxy-Geometrie" zu verwenden, also eine stark simplifizierte Version des Modells.

Engine-Architektur

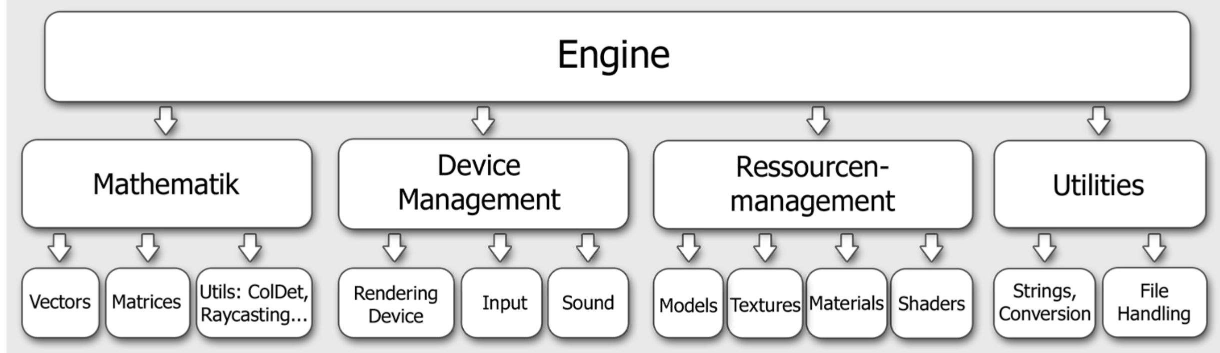
Ihr werdet für euer Spiel eine eigene kleine Game Engine implementieren müssen. Sie muss nicht umfangreich sein, und ihr müsst ohnehin davon ausgehen dass ihr sie nach der LU nicht weiterverwenden könnt, da *alle* Gruppen kurz vor den Deadlines unter Zeitdruck geraten, und daher hässliche Hacks an der Tagesordnung sind. Das ist kein Problem und bei kommerziellen Projekten genauso Gang und Gebe.
Dennoch solltet ihr daran denken, den Engine-Code und den Game-Code entsprechend zu trennen. Denn mit einer guten Engine gehen auch die Hacks schneller, und die Gefahr von Flüchtigkeitsfehlern sinkt.
Das Benützen von **wohldefinierten Interfaces** verringert nicht nur die Gefahr von Fehlern (wenn irgendwelche Game-Objekte und die Engine sich ständig gegenseitig beeinflussen, können leicht ungewollte Seiteneffekte entstehen. In C++ besteht hier die Gefahr dass ihr gar nicht merkt dass ihr z.B. in falsche Speicherbereiche schreibt. Benutzt daher unbedingt den Debug Modus zum entwickeln, die vorhandenen STL-Container führen diese Checks dann selbständig aus), sondern **beschleunigt auch die Kompilierungszeit** enorm, da weniger Abhängigkeiten zwischen den Quellcodedateien entstehen.

Eine Engine stellt normalerweise eine **Mathematik-Bibliothek** zur Verfügung, die Klassen für **Vektoren** und **Matrizen** oder **Quaternionen** beinhaltet. Generell ist zu sagen, dass für fast alle Rotationen Matrizen ausreichen. Quaternionen sind sehr schwer zu verstehen, daher ist die Implementierung nicht trivial und Bugs sind schwer zu finden. Wir empfehlen daher, sie nur zu verwenden, wenn ihr sie unbedingt braucht. Ein Vorteil von Quaternionen ist „spherical linear interpolation" (SLERP), wobei mit gleichmäßiger Winkelgeschwindigkeit auf dem kürzesten Weg zwischen zwei Rotationen linear interpoliert wird. Solltet ihr das nicht benötigen, so empfehlen wir auf Matrizen zurückzugreifen, da diese wesentlich einfacher zu verstehen und debuggen sind. Außerdem lassen sie sich gleich 1:1 an OpenGL übergeben und müssen nicht erst umgewandelt werden. Von Euler-Winkeln für Orientierung solltet ihr absehen, da dort das **Gimbal Lock**-Problem entstehen kann.

Normalerweise kümmert sich die Engine auch um **Ressourcenverwaltung**. Das heißt dass sie den Rendercontext bereitstellt, Modelle und Texturen lädt, und Sounds und Musik abspielt, und den Status von Maus und Tastatur zurückgibt. Als Manager für Modelle, Texturen etc. reicht oft eine simple STL-List oder -Map. Das erlaubt euch bereits, vom Game-Code aus jederzeit Ressourcen über den Dateinamen anfordern zu können. Sind die Ressourcen bereits vorhanden, so wird die

vorhandene Referenz zurückgegeben, ansonsten wird das Objekt vorher intern geladen und in die Liste hinzugefügt. Das verhindert auch das Laden von doppelten Ressourcen, und erleichtert das Tracking von möglichen Memory Leaks, da die Objekte zentral verwaltet werden.
Ihr müsst bei CG2LU nicht sklavisch alle Engine-Klassen komplett modular vom Game-Code trennen, aber überlegt euch zumindest, für alle wichtigen Ressourcen-Typen eigene Manager zu schreiben, da euch das später die Arbeit enorm erleichtern kann, wenn es darum geht schnell neuen Content zu importieren. Wenn ihr ein Template oder eine Basisklasse für diese Manager schreibt, dann habt ihr die einzelnen Typen-Manager in wenigen Minuten implementiert.

Beispiel Super J:

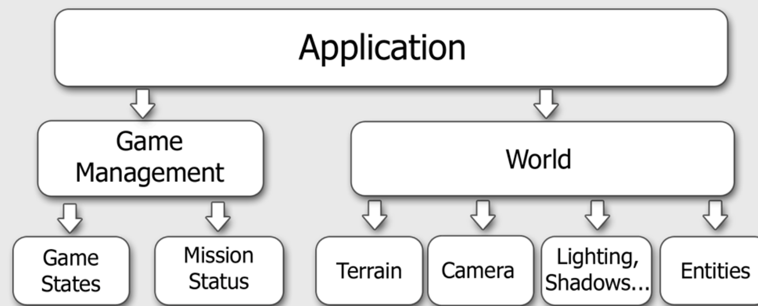


Gamecode-Architektur

Grundsätzlich sollte die Spiellogik und das Rendering *immer* getrennt werden. Das bedeutet, dass euer Main Loop aus 2 Funktionen besteht: einer **Update**-Funktion, die die Spiellogik aller Objekte ändert, und einer **Render**-Funktion, die den aktuellen Zustand auf den Bildschirm bringt.
Es soll also möglich sein zweimal hintereinander „Render“ aufzurufen, ohne dass sich das Bild ändert. Das ist sinnvoll um eine **framerate-unabhängige Spiellogik** zu erreichen.

Euer Spiel muss immer wissen in welchem Zustand es ist. Spiellogik und Rendering von einem Menü unterscheiden sich grundlegend von In-Game-Situationen. Daher empfiehlt es sich, den Spielzustand über „**Game States**“ zu steuern. Im simpelsten Fall ist das einfach ein Enum, das einen von mehreren möglichen Zuständen annimmt. Dabei implementiert jeder Game State eine eigene Update- und Render-Funktion, um genau das zu aktualisieren, was momentan relevant ist.
Wenn ihr beispielsweise wollt, dass während einem Menü das Spiel im Hintergrund weiterläuft, so ist das kein Problem, indem ihr im Menü-Game State einfach zusätzlich zur Update-Funktion der Menü-Logik noch die Update-Funktion des Levels aufruft.
Wollt ihr einen Pause-Modus in euer Spiel einbauen, so wäre das ein Game State, der zwar die Render-Methode des Levels aufruft, nicht aber die Update-Funktion.
Auf diese Art und Weise lässt sich der Spielablauf schnell, simpel und sauber strukturieren. Game States lassen sich auch leicht im Nachhinein noch in vorhandenen Code einbauen, da sie den bestehenden Main Loop einfach umschließen. Zusätzliche Spielzustände wie ein Gewinn- und Game Over-Bildschirm oder Credits lassen sich damit in wenigen Minuten einbauen, ohne in vorhandenen Code eingreifen zu müssen.

Beispiel Super J:



Bei der 2. Abgabe war bereits der Level-Loading- und –Rendering-Teil des Spiels fertig. Die Spiellogik war aber nur rudimentär vorhanden. Kollision mit der Heightmap funktionierte über simples bilineares Filtern der Höheninformation, Gegner und Missionsziele waren noch keine vorhanden.
Für die 3. Abgabe wird dann die Physik komplett für alle Entities implementiert. Wir können leider nicht auf simple Bounding Boxen für die Entities zurückgreifen, da Super J fliegen kann, und damit auf den Dächern von Häusern und auf anderen Objekten landen können muss. Da diese Objekte meistens komplexe nicht-gerade Formen beinhalten, müssen wir auf eine komplexere Collision Detection zurückgreifen. Könnte Super J nicht fliegen, würde eine simple 2D-Kollision mit dem Grundriss von Level-Objekten in Kombination mit dem Heightmap-Testing bereits ausreichen! Für Schüsse kann immer noch Raycasting implementiert werden, da hier keine Collision *Response* notwendig ist. Daher haben wir eine fertige Collision Detection Library benutzt. Außerdem werden für die 3. Abgabe eine Missionsstruktur und Gegner mit einer sehr simplen AI eingebaut. Auf der Rendering-Seite werden die Spezialeffekte hinzugefügt, sowie eine simple lineare Interpolation zwischen Keyframes für komplexe Animation.

Level-Datenstrukturen

Überlegt euch wie ihr euer Level intern repräsentieren wollt. Dabei müsst ihr euch vor allem überlegen, was mit dynamischen, beweglichen Objekten passiert. Statische Objekte lassen sich einfach beim Laden in z.B. einen Octree einordnen, aber was passiert mit Objekten die zur Laufzeit ihre Zelle ändern können? Und was passiert mit Objekten die dann vielleicht in 2 Zellen hineinragen? Hier empfiehlt es sich generell, eher auf simplere Methoden zurückzugreifen. Ein kompletter hierarchischer Szenegraph ist oftmals unnötig und eher „Overengineering" als nützlich. Oft reicht es schon alle Objekte im Spiel einfach in einer Liste zu halten und nacheinander zu traversieren und zu rendern. Wenn ihr größere Levels habt, reicht oft eine simple an den Hauptachsen orientierte Raumaufteilung wie Quadtree, Octree oder KD-Tree aus. Von BSP-Trees ist abzuraten, da sie sehr aufwändig zu berechnen sind und sich nicht dynamisch aktualisieren lassen.
Solltet ihr eine Physikengine verwenden, dann überlegt euch auf jeden Fall, sehr große Objekte (z.B. die Levelgeometrie) räumlich zu zerteilen, da ihr mit riesigen nicht-konvexen Dreiecksmeshs die Physikengine stark belastet. Mehrere kleinere Meshs sind hier sicher effizienter.
Wie immer gilt auch hier: Oftmals ist die simpelste Methode die effizienteste! Z.B bringt es nichts, einen komplexen Octree aufzubauen, wenn man ohnehin nur 10 Objekte im Level hat, und das Bottleneck der Applikation beim Rendering eines komplexen Fragment Shaders liegt. Man sollte nicht in Versuchung geraten hier verfrüht zu optimieren, da einen komplexere Datenstrukturen oftmals in der Flexibilität stark einschränken, und selbst kleine Änderungen der Level-Geometrie sehr aufwändig werden können.

Beispiel Super J:
Wir benutzen keinen klassischen Quadtree, aber haben die Welt entlang der X- und Z-Achse in quadratische Blöcke aufgeteilt. Dabei besteht die gesamte Welt aus 4*4 „Areas“, die wiederum aus je 4*4 „Blocks“ bestehen. Ein Block ist 160*160m groß. Ein Block enthält einen Terrain-Vertexbuffer und ist damit die kleinste Einheit des Terrains, die auf einmal gerendert und gecullt werden kann. Für Culling verwenden wir einfaches **View Frustum Culling**.
Die Objekte („Entities“) im Spiel sind in einer einfachen Liste gespeichert. Da es hier aber sehr viele Entities geben kann (momentan etwa 3000), wäre es höchst ineffizient alle Entities für Spiel-Logik und Rendering zu traversieren. Daher hält jeder Terrainblock zusätzlich eine Liste an den Entities, die in diesem Block positioniert sind. Bewegt sich eine Entity von einem Block in den anderen, wird diese Information automatisch aktualisiert. Für den Check, in welchem Block sich eine Entity befindet, wird nur ihre Position herangezogen, die Größe des Entity-Modells ist unwesentlich. Daher lassen sich Entities einfach über das Frustum Culling der Blocks cullen. Damit Entities, die vielleicht teilweise aus einem Block hinausragen, nicht fälschlich weggecullt werden, wird die Bounding Box aller Blocks um einen fixen Wert vergrößert (20m in X- und Z-Richtung). Die Spiellogik wird nur für Entities ausgeführt, die sich in Blocks nahe der Kameraposition befinden.

Wiederverwendbarkeit von Code

Wenn man seine Klassen gut plant, kann man viel **Code wiederverwenden**. Wenn ihr bestimmte Spielelemente (Gegner, Projektile, Bonusgegenstände…) einmal implementiert habt, so lassen sich oft mit wenig Aufwand verschiedene Variationen davon erstellen. Gleichzeitig wirkt euer Spiel dadurch vollständiger und variantenreicher. Nutzt diese Gelegenheit, um euer Spiel mit geringem Aufwand entsprechend aufzupeppen!

Beispiel Super J:
Alle Gegner leiten sich von derselben Basisklasse ab, da sie fast dieselben Verhaltensmuster zeigen (Sichtbarkeitscheck mit J, Rotation zu J, Angriff). Die einzelnen Gegner unterscheiden sich nur in dem Code, der ihre Animation steuert, und ihrem konkreten Bewegungsmuster (unbeweglich, laufen, schweben).
Viele Game States in Super J basieren ebenfalls auf demselben Code (Missionsbriefing und –erfolg, Logo, Credits und How-To-Play, Titelbild und Game Over).

Auch bei **Effekten** lässt sich viel Code wiederverwenden: Partikelsysteme sind ein wunderbares Instrument, um Aktionen im Spiel visuell zu unterstützen. Schüsse, Treffer, Explosionen etc. Aber auch rein kosmetische Effekte wie Rauch, Nebel, sprühende Funken… sind schnell implementiert und erhöhen den visuellen Gesamteindruck enorm.

Beispiel Super J:
In Super J gibt es etwa 15 verschiedene Typen von Partikelsystemen, die alle mit maximal 2 Bildschirmseiten Code auskommen. Dabei hält die Basisklasse lediglich Render-Information (Textur, Blending-Modus) und eine Liste an Partikeln mit Informationen zu Position, Größe, Farbe und Animationsfaktoren, wie sich diese Attribute über die Zeit verändern sollen. Bereits mit so einfachen Mitteln lässt sich eine große Anzahl an visuellen Effekten umsetzen.

Content

Level Creation-Methoden

Da ihr nicht die Zeit haben werdet, ein eigenes Level-Design-Tool zu programmieren (außer natürlich euer Spiel ist um das Tool herum aufgebaut, z.B. „Incredible Machine“), solltet ihr auf vorhandene Lösungen zurückgreifen.
In den letzten Jahren haben viele Studenten erfolgreich mit Maya gearbeitet, da sich hier über „Custom Attributes“ so gut wie jede erdenkliche Zusatzlogik für Objekte zur Szene hinzufügen lässt, und da sich das **FBX-Format** gut mit dem FBX SDK auslesen lässt.
Für viele Spiele bietet es sich auch an, die Terraininformation aus einer **Heightmap-Textur** auszulesen, und die Texturierung des Geländes mit verschiedenen Typen automatisch beim Laden erfolgen zu lassen (Gras als Standardtyp, Erde an steileren Stellen, Felsen bei noch stärkerer Steigung, Sand rund um Wasser…). Eine sehr einfache Methode um dem Wasser zusätzliche Tiefe zu verleihen (wird in Super J angewendet) ist, dass das Terrain unterhalb der Wasseroberfläche einfach graduell mit der Tiefe immer bläulicher und dunkler wird. Das erzeugt einen zusätzlichen Tiefeneindruck, und verhindert, dass das Wasser nur wie eine simple auf irgendeiner Höhe aufgespannte halbtransparente Ebene wirkt (was es natürlich weiterhin ist!).
Es bietet sich ebenfalls an die einzelnen Objekte eines Levels in einer **Textdatei** in einem simplen leicht parsbaren Format zu definieren (XML ist dafür u.U. schon komplexer als notwendig). Eine einfache Liste an Objekten reicht hier oft aus.

Style-Guide

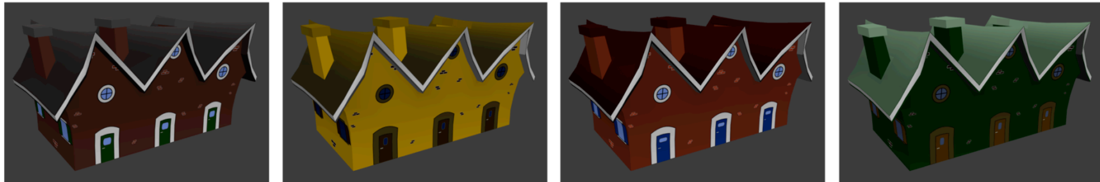
Benutzt **Texturen** auf allen Objekten! Texturen machen einen *sehr* großen Teil des optischen Eindrucks von Objekten aus, während die polygonale Form eher zu vernachlässigen ist, und auch sehr simpel gehalten werden kann.
Benutzt **hohen Kontrast** in euren Spielen. Stellen, die dem Licht abgewandt sind, sollten auch entsprechend dunkler sein. Der Spieler soll jederzeit bereits anhand der Beleuchtung der Objekte festmachen können, wie sie im Raum stehen. Überlegt euch auch, den Spieler mit Licht zu „leiten“, also wichtige Objekte mit einem zusätzlichen lokalen Licht zu bestrahlen, oder umgekehrt den Raum rundherum „abzudunkeln“ (z.B. mit entsprechenden Texturen).
Ihr könnt auch händisch in die Texturen dunklere Stellen malen („Dirt Maps“), um im Schatten liegende Teile von Objekten, oder Punkte die nur sehr wenig Licht von außen erhalten, noch weiter zu betonen. Das verstärkt den Tiefeneindruck, den der Betrachter von dem Objekt bekommt, und bringt mehr als zusätzliche Geometrie. Das Tool „CrazyBump“ bietet die Möglichkeit solche Texturen als „**Occlusion Maps**“ zu generieren. Diese Texturen könnt ihr einfach mit der normalen Objekttextur zusammenmultiplizieren.
Benutzt wenn möglich keine Specular Highlights in Kombination mit Per-Vertex-Shading, da hier durch das Gouraud-Shading der Polygone die einzelnen Polygonkanten zusätzlich verstärkt werden und das Specular Highlight nicht sanft über das Objekt wandert sondern eher drüber“flackert“.

Content Reuse

Da der Aufwand, eine komplette Freeroaming-Welt zu schaffen, sehr groß ist, wird stark auf die Wiederverwendbarkeit von Content Wert gelegt. Vorweg kann man beim Comic-Look des Spiels schon mit geringerem Poly-Count pro Model arbeiten, da hier ein eckiges Aussehen mit prägnanter

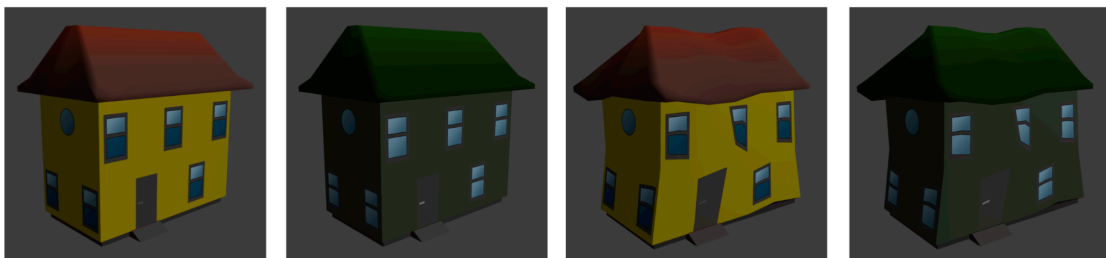
Silhouette von Vorteil ist. Um den Arbeitsaufwand weiter zu minimieren kann man verschiedene Methoden anwenden, die für jeden Grafikstil anwendbar sind:

- Die einfachste Methode ist, den Informationsgehalt des Models möglichst **komplett über die Textur** zu erhalten. Damit muss man nur jeweils die Textur tauschen, ohne die Geometrie ändern zu müssen. Als Beispiel in Super J dienen hier die Reihenhäuser im Dorfbereich. Für dasselbe Modell werden 4 unterschiedliche Texturen verwendet.



- Variante 2 wäre ein **Basismodel** zu entwerfen und dieses dann in verschiedene Varianten auszuarbeiten. In Super J wären das die LKWs. Alle 4 Arten von LKWs basieren auf demselben Grundmodell.

- Variante 3 wäre, Sets von **verschiedenen Modellen und Texturen** zu kombinieren, sodass dieselben Texturen auf alle Modelle anwendbar sind, da alle Modelle die gleichen Texturkoordinaten benutzen. Am Beispiel Super J wären das die einzelnen Häuser am Hügel im Dorfbereich. Diese haben zwar ein unterschiedliches Aussehen (2 Typen), die Texturen sind aber für alle Varianten komplett gleich gesetzt. Die Textur basiert hier einfach auf den erforderlichen Farben für Wand, Dach etc. sowie einer Fenster-Textur. Wenn man jetzt verschiedene Texturen benutzt (5 Typen), dann kommt man bereits auf 10 Variationen!



Mit diesen drei Methoden kann man in relativ kurzer Zeit eine große Anzahl an kohärent aussehenden aber einfach zu erstellenden Modellen produzieren.

Gameplay dos and don`ts - The 10 Game Design-Commandments.

Die hier folgende Liste an Tipps und Tricks geht zwar exemplarisch auf Super J und damit ein 3rd Person-Spiel ein, ist aber natürlich auch auf andere Spiele anwendbar.

1. **Thou shall feed back!**
   Jede (Inter-)Aktion im Spiel soll mit einem **verständlichen Feedback** beantwortet werden. Idealerweise multimodal. D.h. sowohl optischer Reiz und Sound oder Rumble-Funktion des Controllers. Wichtig: Das gilt nicht nur für aktive Spieleraktionen (wie Aufsammeln von wichtigen Items) sondern auch für Spieler-unabhängige Aktionen (z.B. sollte ein Schuss nicht einfach verschwinden wenn er auf ein Objekt trifft, sondern ein Sound und ein visueller Einschuss-Effekt sollten abgespielt werden. Woher weiß der Spieler sonst ob der Schuss nicht womöglich durch das Objekt hindurch gegangen ist?). Viele dieser Effekte lassen sich sehr schnell über Billboards oder Partikelsysteme einbauen, wenn man bereits eine Partikelsystem-Basisklasse implementiert hat. Daher empfiehlt es sich hier auf eine erweiterbare Architektur zu achten. Viele Effekte werden sehr ähnlich sein (unterschiedliche Arten von Rauch oder Funken), und lassen sich daher in nur wenigen Minuten aus bestehenden Klassen bauen. Diese Effekte tragen aber *enorm* zum Spielgefühl bei und lassen das Spiel viel fertiger und responsiver wirken.

   Beispiel Super J:
   - Beim Sammeln von Items ertönt ein Sound und ein Partikelsystem spawnt.
   - Bei Treffer vom Gegner: Bildschirm leuchtet rot, Sound.
   - Bei Schuss: Sound, Mündungsfeuer

2. **Thou shall be informed!**
   Jede spielrelevante Information soll **klar und jederzeit ersichtlich** sein. Bei Einsatz eines HUDs möglichst verständliche Symbole verwenden. Bilder und Farben sind schneller und einfacher zu verstehen als textuelle Information, brauchen weniger Platz und wirken weniger aufdringlich.
   Bei In-Game-Darstellung kann man Informationen direkt am Spielermodell sichtbar machen (Schadenstexturen, Animationen) oder als Fullscreen-Effekte umsetzen.
   Man sollte sich gründlich überlegen ob man überhaupt ein explizites HUD einsetzen will, da es die Immersivität des Spiels reduziert. Daher wird in vielen modernen Spielen darauf verzichtet.

   Beispiel Super J:
   - HUD: Pfeil zeigt Richtung des nächsten Missionsziels an.
   - In Game: Farben faden aus bei niedriger Health, Bildschirm pulsiert rot als Warnung

3. **Thou shall know what you are doing!**
   WASD bleibt WASD bleibt WASD. Spieler erwarten gewisse **Standards bei der Steuerung** von Spielen, an die ihr euch so weit wie möglich halten solltet.
   Außerdem sollten alle Spieleingaben möglichst **unmittelbar umgesetzt** werden (z.B. keine zwei-Sekunden-Animation beim losgehen oder stehenbleiben). Der Spieler sollte immer das Gefühl haben die Kontrolle über die Spielfigur zu besitzen. Die Steuerung sollte über alle Teile des Spiels gleich bleiben und sich nicht pro Level, Spielmodus oder Menübildschirm ändern.

4. **Thou shall know what's happening!**
   Konsistentes und **nachvollziehbares Gameplay**. Objekte eines „Typs“ sollten sich gleich verhalten. Gegenstände werden immer gleich aufgesammelt, Schalter immer gleich aktiviert etc. Auch die Collision Detection sollte nachvollziehbar sein. Grundlos durchlässige Objekte oder unsichtbare Wände sollten so weit wie möglich vermieden werden.

5. **Thou shall have fun!**
   Spielfluss und -geschwindigkeit sind aufrecht zu erhalten. Man sollte unnötig repetitive

Tätigkeiten vermeiden, die den Spielfortschritt bremsen, weil man Angst hat, das Spiel könnte sonst zu kurz werden.
Als Entwickler ist es schwer, die Zeit einzuschätzen, die ein Spieler für eine Aktion benötigen wird. Normalerweise sollte man Tätigkeiten auch nicht öfter als dreimal hintereinander wiederholen, bevor sie den Spieler langweilen. Daher sollten **verschiedene Aktionen** (Kampf, Rätsel, Aufsammeln von Items…) immer entsprechend **durchmischt** werden. Im Idealfall mischt man actionreichere und ruhigere Passagen, da das den subjektiven Tempo-Eindruck noch verstärkt.

Beispiel Super J:

- Konsistentes Tempo innerhalb einer Mission, aber jede Mission wird kampflastiger als die vorhergehende.
- Zwischen den Missionen fliegt man zurück zum Ideas Bin.

6. Thou shan't frustrate the player!

Schwierigkeitsgrad und Spielkomplexität sollten ansteigend sein. Der Spieler sollte vorsichtig an das Gameplay herangeführt werden, man sollte vermeiden mit frustig schweren Stellen zu beginnen. Auch hier gilt, dass es als Entwickler nicht einfach ist, die Schwierigkeit einer Spielsituation einzuschätzen. Hier hilft Feedback von externen Testern.
Wenn euer Gameplay sehr komplex ist, und aus vielen Teilen besteht, dann überlegt, nicht von Anfang an alles zugänglich zu machen, sondern erst nach und nach zu präsentieren. Außerdem könnt ihr so den Spieler länger an euer Spiel fesseln, da er ständig neue Dinge entdeckt, und nicht nach wenigen Sekunden schon den Eindruck hat, alles gesehen zu haben.

Beispiel Super J:

- Missionen haben einen ansteigenden Schwierigkeitsgrad und eine ansteigende Spielgeschwindigkeit.
- Mit jeder Mission kommt ein neuer Charakter ins Spiel, was das Gameplay erweitert.

7. Thou shall know where you are!

Objekt- und Kameraposition sinnvoll setzen. Für den Spieler sollte immer ersichtlich sein, wo es weiter geht. Es empfiehlt sich die Spielwelt entsprechend zu **strukturieren** und nach Möglichkeit Objekte mit aus der realen Welt bekannten Größen zu benutzen (Tür, Tisch, Auto…), damit der Spieler ein Gefühl für den Raum bekommt.
Eine gute Methode, den Spieler durch einen Level zu leiten, ist, **Licht** einzusetzen (den Ausgang aus einem Raum oder wichtige Objekte beleuchten) oder „**Landmarks**" einzusetzen (Objekte die von vielen Punkten im Level sichtbar sind und dem Spieler die Orientierung erleichtern). Große leere Räume sind zu vermeiden!
Die Kameraposition sollte immer so gewählt sein, dass man den größtmöglichen Anteil des Spielgeschehens einfängt. Der Spieler sollte immer **freie Sicht** auf den Spieler-Charakter haben. Wenn sich Objekte zwischen dem Spieler und der Kamera befinden, so kann man die Objekte semitransparent erscheinen lassen (wichtig: tiefensortieren!) oder das über Raycasting gar nicht erst zulassen und die Kamera in dem Fall näher an den Spieler heran bewegen. Weitwinkel größer als 90° ist zu vermeiden, da es entsprechend der Projektionsmatrix in einer „Verzerrung in die Tiefe" resultiert!

Beispiel Super J:

- Wiedererkennbare Stadtteile (Großstadt, Dorf, Hafen) und Landmarks (Ideas-Bin, Brücken…).
- Große leere Räume (außer der Wüste) wurden vermieden, indem Hügel mit Bäumen bepflanzt wurden und Straßen zusätzliche Struktur erzeugen.

8. **Thou shan't blind the player!**

   Visuelle Effekte sollen dem Spiel, Gameplay und Stil angepasst werden und die Atmosphäre unterstützen, nicht aber zum reinen Selbstzweck verwendet werden. Man sollte auch aufpassen den Bildschirm **nicht mit Effekten überladen** bis das Spiel unspielbar wird (z.B. starker ständig aktiver Motion Blur, der das gesamte Spielgeschehen verschwimmen lässt).

Beispiel Super J:

- Field of View-Verzerrung zum besseren Geschwindigkeitseindruck.
- Motionblur: Geschwindigkeitseindruck, aber die Mitte des Bildschirms bleibt scharf.

9. **Thou shall use established features**

   Objekte, die dem Spieler **aus der realen Welt bekannt** sind, sollten im Spiel die gleiche Funktion erfüllen wie in der Wirklichkeit, außer es gibt eine gute Begründung. Auch hier stellt der Spieler Erwartungen, die zu erfüllen sind. Z.B. sollten Türen immer in den nächsten Raum führen (oder versperrt sein) und nicht das Licht einschalten (außer die Türen dienen als Portale in eine andere Welt wie in Monsters Inc.).

Beispiel Super J:

- Wasser: Charakter muss schwimmen und läuft nicht unter der Wasseroberfläche am Grund entlang (Bei einem Jesus-Spiel wäre am Wasser gehen ein „established feature", da es vom Spieler erwartet wird).

10. **Thou shall listen up!**

    Tutoren-Feedback ist nicht zur Allgemeinbelustigung oder als Alternative zu einem Kinobesuch gedacht, es soll den Ausbildungsfortschritt fördern und den bestmöglichen Abschluss der Übung erleichtern.

Viel Spaß bei CG2LU im Sommersemester 2008!