CG2 LU

SS 2010

Lukas Pezenka 0625948 – 033 532
Christoph Weinzierl-Heigl 0625044 – 033 532

Shoot Em‘ Up

Umsetzung der Anforderungen

Gameplay

In „Shoot’em’Up“ übernimmt der Spieler die Rolle eines auf einem feindlichen außerirdischen Planeten gestrandeten Space Marines. Er muss sich durch die gegnerischen Monsterhorden ballern. Dazu gibt es drei unterschiedliche Waffentypen:

- BFG (Big Fucking Gun),
- Hammer of God,
- Boomstick

Neben Handfeuerwaffen stehen dem Spieler Granaten zur Verfügung. Diese können bis zu einem festgesetzten Maximum beliebig weit geworfen werden (durch gedrückt halten der rechten Maustaste – beim Loslassen der Taste wird die Granate geworfen) und töten alle Gegner in einem bestimmten Umkreis. Obwohl Granaten schwerer zu meistern sind als Handfeuerwaffen, stellt die Möglichkeit des indirekten Feuers eine Belohnung für erfahrene Spieler dar.

Außerdem gibt es drei Arten von Powerups:

- Health: Regeneriert die Trefferpunkte des Spielers vollständig
- Steroids: Verleiht dem Spieler für eine limierte Zeit größere Schelligkeit und Sprungkraft
- Grenades: Füllt das Granatenmagazin des Spielers wieder vollständig auf

Powerups sind im Level platziert und schweben solange, bis sie vom Spieler eingesammelt werden.

Über den Lauf des Spiels wird eine gewisse Anzahl an Gegnern durch Portale (sog. Spawnpoints) ins Level transportiert. Spawnpoints werden deaktiviert, nachdem alle vorgesehenen Gegner erschaffen wurden.

Hat der Spieler es geschafft alle Gegner zu eliminieren, so öffnet sich ein Portal, das den einzigen Weg zurück zur Erde darstellt. Sobald das Portal erreicht wurde hat ist das Spiel gewonnen.

Nichttriviale Objekte

Wir haben verschiedene Arten von nicht trivialen Objekten

- Level

- Guns
- Powerups
- Animierte Models

All unsere Geometrie laden wir aus .obj Files. Die Objektloader haben wir vollständig selbst entworfen und programmiert. Des Weiteren haben wir eine Reihe an zusätzlichen Fileformaten entwickelt, um die Möglichkeiten von .obj zu erweitern. Siehe dazu das Kapitel „Sonstige Besonderheiten“.

Animierte Objekte

Sowohl unser Spielermodel als auch unsere Gegner werden per Skeletal Animation animiert. Dafür haben wir eigene Dateiformate (mh2, .sk2, .vm2, .af2, .kf2) entwickelt (siehe Kapitel „Sonstige Besonderheiten“). Anzumerken ist, dass das Vertex-Skinning auf der GPU geschieht.

Beschleunigung der Sichtbarkeitsberechnung

View-Frustum Culling haben wir implementiert, jedoch gab es bis zuletzt Schwierigkeiten, wodurch es zu Fehlern bei der Darstellung kommt (Objekte werden gecullt obwohl sie im Frustum sind). Dadurch, dass wir eigene Mathematikklassen für Matrizen, Vektoren, Ebenen & Co. entworfen haben kann es natürlich sein, das sich der Fehler darin versteckt.

Transparenz Effekte

Wir haben Transparenzeffekte bei unserem HUD (Head Up Display) sowie bei Sprites (flache 3D-Objekte) des Levels angewandt.

Das HUD wird mit Hilfe einer orthogonalen Projektionsmatrix in Screen Space Koordinaten auf den Bildschirm projiziert. Des Weiteren verwenden wir additives Blending bei unserer Partikelengine. Die Partikel werden im Abstand zum Betrachter sortiert und dementsprechend gerendert.

Experimentieren mit OpenGL

- Vertex Buffer Objects/Vertex Buffer Arrays: Zwischen den beiden kann In-Game mittels der F6 Taste gewechselt werden
- Texturqualität: Die Texturqualität kann nicht verändert werden, jedoch haben wir uns erlaubt statt des geforderten bi-/trilinearen Filterings auch anisotropisches Filtering anzuwenden
- Wireframe: Mit der Taste F3 kann zwischen normalem Rendering und dem Wireframe-Modus gewechselt werden
- Framerate: Eine Framerate Anzeige kann mit der Taste F2 ein-/ausgeschaltet werden
- Statusausgaben: Werden im HUD links oben angezeigt

Effekte

Partikel Engine

Wir haben eine eigene Partikel Engine entworfen, die lediglich die Physikberechnungen an PhysX auslagert. Es gibt dabei beliebig viele „Partikelemitter“ die jeweils bis zu max. 100 Partikel (festgelegter Wert) managen können. Ein globales Maximum an Partikel gibt es nicht. Die Partikel werden Back-To-Front nach ihrer Entfernung von der Kamera sortiert und in dieser Reihenfolge gerendert. Die einzelnen Partikel werden im Moment additiv geblendet (GL_ONE, GL_ONE).

Anwendung findet dieser Effekt z.B. bei Granaten, Geschossen und an diversen fixen Positionen im Level.

Skeletal Animation + Skinning

Dieser Spezialeffekt wird bei unseren animierten Models (Spielermodel, Gegner) angewendet. Die Vertexinterpolation zwischen zwei Keyframes findet dabei auf der GPU im Vertexshader statt. Dabei werden die Rotationsmatrizen der Bones an den Vertexshader als Parameter übergeben. In unserer Implementation kann ein Vertex von bis zu vier Bones beeinflusst werden. Dabei wird angegeben mit welcher Gewichtung das aktuelle Vertex vom jeweiligen Bone abhängt. Diese 8 Werte (2 für Boneindex und Gewichtung und das ganze 4-mal für bis zu 4 Bones) werden in Tupeln (float2) an den Vertexshader mittels VBA/VBO gestreamt.

Für die Erstellung der Animationen haben wir ein eigenes Tool auf MFC-Basis entworfen. Siehe dazu Kapitel „Sonstige Besonderheiten“.

Normal Mapping

Wird bei unserem Spielermodell verwendet. Tangents und Binormals[1] berechnen wir für alle Models beim Ladevorgang. Durch das geforderte Per-Pixel-Lighting kommt der Effekt schön zur Geltung.

Sonstige Besonderheiten

Models die wir mit diversen Modeling Tools (Blender, MAX, Maya) erstellt haben, exportieren wir ins OGRE XML Format. Für dieses wiederum haben wir einen eigenen Converter geschrieben (ModelConverter), der die Ogre XML Files in unsere eigenen Formate umwandelt.

Dabei werden die Models nach Materialen in Submeshes gesplittet. Dadurch ergeben sich evtl. mehrere einzelne OBJ Dateien welche die Geometrie geordnet nach Materialen enthalten. Diese Trennung erweist sich als nützlich, da die Natur von Open-GL als State-Machine eine Einsparung an State changes mit Performancegewinnen belohnt.

Zusätzlich wird für alle diese OBJ Files ein VM2 File erstellt das für das Vertex-zu-Bone-Mapping zuständig ist. Dieses wird für das Vertex Skinning benötigt.

Besteht ein Model aus mehreren Submeshes, wird ein MH2 File dafür erstellt, welches angibt welche OBJ + VM2 Dateien zu diesem Model gehören.

Unser Model-Converter führt dann zusätzlich aus den OGRE XML Skeleton Dateien eine Umwandlung in unser SK2 Format durch. Darin werden die Bone-Strukturen (Position, Rotation, Stellung im Skelett bzgl. Parent/Child Objekten, etc.) abgelegt.

Für die Animationen (.af2) und Keyframes (.kf2) haben wir ein eigenes Tool (AnimationEditor.exe) auf MFC-Basis entwickelt. Dieser Editor erlaubt die durch den Konverter erstellten Models zu laden und zu animieren. Keyframes und Animationen können dann in die KF2/AF2 Dateien exportiert werden. Die Verwendung separater Files für Keyframes und Animationen hat den Vorteil gegenüber integrierten Formaten, dass ähnliche Models (alle referenzierten Knochen müssen bei beiden Models vorhanden sein und den selben Index haben – dies kann für Skeletttypen wie Biped, Quadped, etc. immer erfüllt werden) dieselben Files nutzen können. Einerseits bringt dies eine Arbeitsersparnis mit

[1] http://jerome.jouvie.free.fr/OpenGl/Lessons/Lesson8.php

sich, andererseits können Änderungen zentral gemacht werden. Das wiederum senkt die aus Redundanz resultierende Fehlermenge.

Das Level wird (mit Ausnahme der Position des Portals, durch welches der Spieler den Planeten verlässt) aus einem Szenen-File ausgelesen. Bei dem Format handelt es sich um eine Eigenproduktion. Eine Besonderheit daran ist, dass es dem Level- / Gamedesigner erlaubt, Objekte mit Attributen zu versehen, welche in der Engine nicht Standardmäßig vorgesehen sind. Dadurch lässt sich das Spiel einfach um neue Objekttypen erweitern, ohne neue (ähnliche) Klassen abzuleiten. Ein Beispiel dafür sind Power-Ups, bei denen es sich letztendlich einfach um Instanzen der GModel-Klasse handelt, denen das Attribut „TYPE“ mit dem Wert „Health“, „Steroids“ oder „Grenades“ zugewiesen ist. Die Spiellogik, die sich auf die verschiedenen Power-Ups bezieht, kann daher an einigen wenigen zentralen Stellen ohne den Overhead weitgehend identischer Klassen implementiert werden.

Externe Libraries

- GLFW (www.glfw.org)
- GLEW (glew.sourceforge.net)
- PhysX (www.nvidia.com)
- OpenAL (connect.creativelabs.com/openal)
- ALUT (connect.creativelabs.com)
- CG (www.nvidia.com)

Entwicklung

Da wir beide Shadermodel 3 Karten verwenden (Nvidia 7xxx Serie/NV4x) war uns der native Einsatz von OpenGL 3 und Geometry-Shadern nicht möglich. Ebenfalls war unser Framework zum Teil schon vor der Übung vorhanden weshalb noch einige Altlasten aus OpenGL 2.x vorhanden sind. Diese sollten im abgegebenen Code aber keine Rolle mehr spielen bzw. nur zu Debug-Zwecken vorhanden sein.