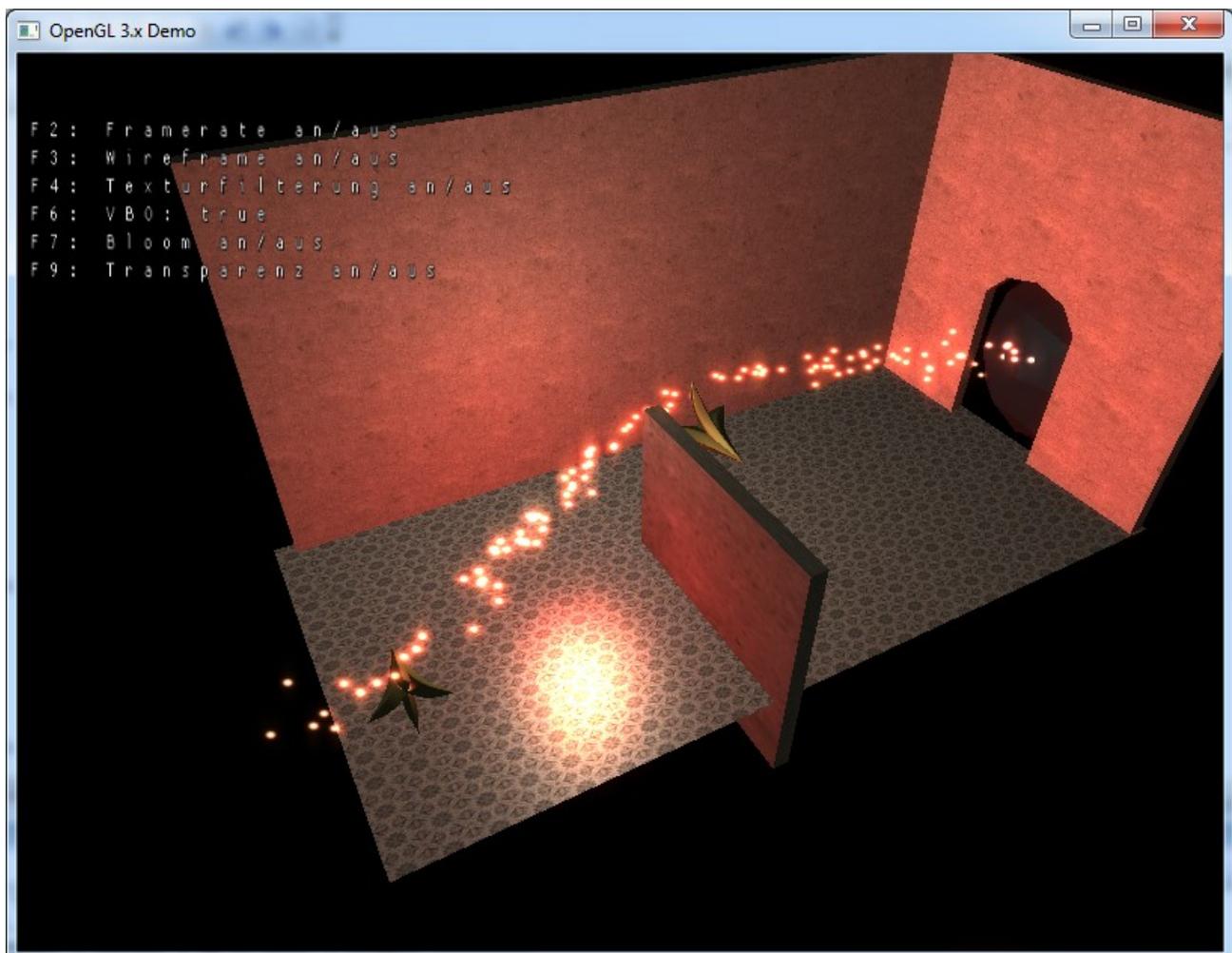


Visitorium

Andrea Diwald
Markus Schütz



Inhaltsverzeichnis

Implementierung.....	3
Kameramodell.....	3
Startwerte.....	3
Interaktion.....	3
Bewegte Objekte.....	3
Forces.....	3
Partikel.....	3
TextureMapping.....	3
Beleuchtung und Materialien.....	4
Szenengraph.....	4
PartikelSystem.....	4
Camera.....	4
MeshNode.....	4
Mesh.....	5
SubMesh.....	5
ResourceManager.....	5
MeshManager.....	5
Matrix4x4.....	5
Vector2/3/4.....	5
Level/Scene.....	5
LevelManager.....	5
VertexShader/FragmentShader/ShaderProgram.....	5
OpenGLRenderer.....	5
Main.cpp.....	6
(Game-)Features.....	6
Checkpoint.....	6
ParticleSystem / Emitter.....	6
Forces.....	6
Spezialeffekte.....	7
Partikelsystem.....	7
Spiegelung.....	7
Bloom.....	8
Tools.....	8
Besonderheiten.....	9
Ogre XML Loader.....	9
LevelLoader.....	9
Kollisionsberechnung.....	9
Mathe.....	9
Libraries.....	10
TinyXML.....	10
Glew.....	10
Glfw.....	10

Gameplay

Ziel des Spiels ist es, von Partikelsystemen ausgestrahlte Partikel mithilfe von Forces in bestimmte Ziele zu Lenken.

Interaktion

- WASD**: Verschiebt die Kamera entlang der xz-Ebene, wobei die Blickrichtung der Kamera mit einbezogen wird.
- MouseWheel**: Zoomen der Kamera
- RechteMaustaste + MouseMove**: Rotiert die Kamera um die y-Achse

Nichttriviale Objekte

Als Objektdateiformat für Meshes wird OgreXML verwendet.

Das Spiel setzt sich aus Objekten wie z.B. dem Force Objekt, Wänden und dem Checkpoint zusammen.

Animierte Objekte

Directional Force

Wird über hierarchische Animation animiert.

Die Hierarchie schaut folgendermaßen aus:

node

 head (Die Pfeile)

 head1 (1. Pfeil)

 head2 (wie 1. Pfeil nur mit roll($\pi/2$) transformiert)

 ring (Die Ringe)

 ring1 (1. Ring)

 ring2(2.Ring)

head1 und head2 werden beide mit roll(time*2) und translate(0,0, sin(glfwGetTime()*3) * 0.01) transformiert. Daraus ergibt sich ein rotieren sowie eine kontinuierliche vor und rückwärtsbewegung.

Ring wird mittels yaw und roll rotiert. Zusätzlich wird ring1 mit einem weiteren pitch und einem roll rotiert.

Transparenz

Die Partikel werden mit einer Textur mit Alpha Kanal gerendert. Damit es keine Probleme mit dem z-Buffer gibt werden die Partikel nach Entfernung sortiert und von hinten nach vorne gerendert. Zusätzlich wird das Schreiben in den z-Buffer deaktiviert damit Partikel mit gleicher Entfernung sich nicht gegenseitig verdecken.

Experimentieren mit OpenGL

VertexBufferObjects

Standardmäßig werden alle umfangreicheren Objekte mit VBOs gerendert. Dazu wird für jedes geladene Mesh gleich ein VBO generiert welches für alle kommenden Zeichenvorgänge verwendet wird.

Auch Partikel werden in ein VBO gespeichert. Da aber jedes einzelne Partikel bei jedem Frame seine Position ändert muss das VBO jedes Frame neu aufgebaut werden daher wird ein dynamisches VBO verwendet.

Mit F6 kann zwischen Vertex Arrays und VBOs gewechselt werden. Performanceunterschiede gibt es allerdings kaum was vermutlich daran liegt, dass keine Polygonreichen Objekte vorkommen.

FrameBufferObjects

Framebuffer werden für Effekte wie Bloom und Spiegelung verwendet. Wenn eine Szene gerendert wird, dann wird sie erstmal in einen eigenen Framebuffer(dessen Größe sich der Fenstergröße anpasst) gerendert. Der Inhalt des Framebuffers wird anschließend entweder in den SystemFramebuffer gerendert oder, bei aktiviertem Bloom, mit Speziellen Shadern in weitere Framebuffer gerendert und am Schluss in den SystemFramebuffer.

MipMapping

Wird z.B. vom Blur Shader verwendet. Der Blur Shader braucht keine hochdetaillierten Texturen sondern arbeitet mit niedrigeren MipMap Leveln. Das führt zu einem enormen Performancegewinn, da Der Blur Kernel viel kleiner sein kann.

Textur-Qualitätseinstellungen

Standardmäßig wird Bilineare Filterung verwendet. Mit F4 kann auf Nearest Neighbour umgeschaltet werden.

Implementierung

Kameramodell

Folgendes Verhalten wurde für die Kamera festgelegt:

Die Kamera hängt an 3 SceneNodes an, die für translation und rotation verantwortlich sind:

camNode -> camTransNode -> camRotNode -> camera

- camNode**: Die Kamera schaut stets auf die Position dieses Nodes.
- CamTransNode**: Durch Translation entlang der z-Achse wird die Entfernung der Kamera zum camNode festgelegt.
- CamRotNode**: Durch Rotation entlang der y-Achse dreht sich die Kamera schließlich um das camNode
- camera**: Die Kamera selbst wird nicht transformiert.

Startwerte

Zu Beginn wird die Kamera um 15u entlang der z-Achse verschoben und Anschließend um -0.7 entlang der x-Achse und um $3.14/4$ entlang der y-Achse rotiert.

Bewegte Objekte

Forces

Forces lassen sich durch den Benutzer durch Mausbewegung mit gleichzeitig gedrückter linker Maustaste bewegen. Mit Tabulator wird das nächste Force selektiert.

Partikel

Die Partikel bewegen sich automatisch in die vom Emitter vorgegebene Richtung und können durch Forces abgelenkt werden.

TextureMapping

Das Material "SpecularMaterial" kann eine Textur auf das Objekt auftragen. Dazu wird für jedes Modell Texturkoordinaten sowie eine Textur benötigt. In `SpecularMaterial.draw()` werden dem ShaderProgram die Texturkoordinaten als Attribute (kann sich für jeden Vertex ändern) und die Textur als uniform `sampler2D`(gilt für alle Verticen) übergeben. Der Shader holt sich die Texturfarbe an einer bestimmten Stelle mit der Funktion `texture2D`, in die der Sampler sowie die (zwischen den Verticen interpolierten) Texturkoordinaten übergeben werden.

Beleuchtung und Materialien

Jedes der verwendeten Modelle hat Texturkoordinaten und Normalen und wird auch mit einer Textur versehen.

Im Programm werden zuerst alle Materialien erstellt. Für jedes Material wird eine Instanz der Klasse `SpecularMaterial` angelegt. Folgende Einstellungen können dabei vorgenommen werden:

- Textur
- specularExponent
- specularCoefficient
- ambientColour
- opacity

In den Mesh XML Files steht für jedes SubMesh der Name des zu verwendenden Materials.

Wenn es dann ans Rendern geht, dann wird für jedes SubMesh in der Szene die Methode

```
void draw(SubMesh *subMesh, Scene *scene, Matrix4x4 *world);
```

aufgerufen. Diese Methode bindet die Shader, übergibt die entsprechenden Uniform/Attribute Values und rendert das Ganze. Dabei wird auch jedes Licht der Szene(derzeit bis zu 30) an den Shader übergeben.

Szenengraph

Im Grunde ist alles was man in die Szene reinsetzt ein SceneNode. Egal ob Kamera, Licht, PartikelSystem, Partikel oder sonstiges, alles leitet sich von SceneNode ab.

Die Klasse SceneNode selbst bietet alles, was ein Szenengraph begehrt: Methoden zum Transformieren und Strukturieren.

PartikelSystem

Die Klasse PartikelSystem erzeugt Partikel und schießt diese in bestimmte Richtungen ab. Dazu wird erst ein "Masta" Partikel angegeben, das dann laufend kopiert und in die Szene gesetzt wird. Jedes Mal wenn für ein PartikelSystem die update() Methode aufgerufen wird, wird auch für alle Partikel des Systems update() aufgerufen.

Camera

Bietet zusätzlich zum SceneNode noch Funktionen zur Berechnung von View und Perspective Matrizen.

MeshNode

An jedem MeshNode hängt ein Mesh, das durch Aufrufen der Methode MeshNode.draw() gerendert wird.

Mesh

Besteht aus mehreren SubMeshes

SubMesh

Beinhaltet die eigentlichen Vertex-Informationen wie Position, Normale, Farbe und Texturkoordinaten sowie die Indices der Dreiecke und das zu verwendende Material.

ResourceManager

Die ResourceManager laden und verwalten Ressourcen wie Meshes, Materialien, Shader und Texturen. MaterialManager und ResourceManager (für alles außer Materialien) sind Singletons.

MeshManager

Lädt und speichert Meshes. Die loadMesh Funktion kann zurzeit Ogre XML Meshes mitsamt Texturkoordinaten, Farben, Position und Materialien auslesen.

Matrix4x4

Selbstgeschriebene Matrizenklasse. Bietet statische Methoden zur Erzeugung diverser Transformationsmatrizen, überladene Operatoren zum einfacheren Rechnen und nützliche Funktionen wie getInverse(), getTranspose(), etc.

Vector2/3/4

Vector3 wird hauptsächlich für Positionen verwendet und bietet auch entsprechende Mathematik.

Vector2 und Vector4 sind vereinfachte Varianten davon, die eigentlich nur zum Speichern von Texturkoordinaten bzw. Farben da sind.

Level/Scene

Scene ist die Grundklasse und dient dazu, den Szenengraphen zu verwalten. Damit Kameras, Lichter, Meshes etc. jederzeit ohne Durchsuchen des Szenengraphen verwendet werden können, müssen diese direkt durch die von Level/Scene zur Verfügung gestellten create** Funktionen erzeugt werden. Eine komplette Liste dieser speziellen Nodes erhält man durch entsprechende Getter.

Die Szene wird durch Aufruf der Methode Level->render() gerendert. Sie ruft für Meshes, Spiegel, Forces, etc. deren draw Methoden auf.

LevelManager

Lädt die Leveldaten aus XML

VertexShader/FragmentShader/ShaderProgram

Speichern Handles und SourceCode der Shader.

OpenGLRenderer

OpenGLRenderer.init() erstellt einen OpenGL Context, das Fenster und stellt alles brav ein.

Main.cpp

In der Main Methode werden derzeit noch hardcoded alle Materialien definiert und das erste Level aufgebaut. Anschließend wird die main-loop durchlaufen, die zuerst die Szene aktualisiert(und dabei zuerst den Input und anschließend die weitere Spiellogik handhabt) und anschließend rendert.

(Game-)Features

Checkpoint

Ein Checkpoint zeichnet auf, wieviele Partikel pro Sekunde ihn erreichen. Siegesbedingung für einen Level ist, dass genug Partikel durch alle Checkpoint eines Levels fliegen.

ParticleSystem / Emitter

Streut Partikel in die Szene. Mögliche Einstellungen sind Ursprung, Richtung, Anzahl der Partikel pro Sekunde, Lebensdauer der Partikel und maximal gleichzeitig aktive Partikel.

Forces

Lenken auf verschiedene Art und Weise Partikel ab.

Richtungforces: Lenken Partikel in eine bestimmte Richtung

Gravitation: Ziehen Partikel an.

Spezialeffekte

Folgende Spezialeffekte wurden implementiert:

- Partikelsystem
- Spiegelung
- Bloom

Die wichtigsten Quellen für die Spezialeffekte waren die, die FBOs erklärt haben:

- <http://www.gamedev.net/reference/articles/article2331.asp>
- http://www.songho.ca/opengl/gl_fbo.html

Partikelsystem

Das in ParticleSystem.h definierte Partikelsystem ermöglicht es ein einfaches Partikelsystem zu erstellen und zu rendern.

Zuerst wird eine „Masta“ Partikel Instanz erstellt, in der Geschwindigkeit und Beschleunigung eines Partikels festgelegt sind. Dieses Masta-Partikel wird vom Partikelsystem für jedes neu zu erzeugende Partikel kopiert.

Mögliche Einstellungen sind

- ParticlesPerSecond: Anzahl der Partikel die pro Sekunde in die Szene gesetzt werden
- maxParticles: Maximale Anzahl der von einem Partikelsystem gleichzeitig handhabbaren Partikel. Wenn das maximum erreicht wird, werden keine neuen Partikel erstellt bis eines der vorhandenen wieder gelöscht wurde.
- TimeToLive: Lebenszeit. Wenn ein Partikel seine Lebenserwartung erreicht hat, wird es gelöscht.
- SpawnRadius: Neue Partikel werden an einer zufälligen Position innerhalb der durch den spawnRadius definierten Kugel erzeugt.

Die update(time) Methode erstellt erst neue Partikel und aktualisiert anschließend alle erzeugten. Für jedes Partikel wird berechnet wohin es sich bewegen soll. Anschließend wird per Raycast kontrolliert, ob das Partikel auf dem Weg zu seiner neuen Position irgendein Objekt schneiden würde. Wenn nicht, wird es an die neue Position gesetzt. Wenn es ein Mesh schneidet, dann wird es gelöscht und wenn es einen Spiegel schneidet, dann wird es reflektiert.

Gezeichnet werden die Partikel mit einem eigenen Partikel Shader. Jedes Partikel wird als Punkt an den Shader geschickt. Der Vertex Shader transformiert sie ins normalisierte Koordinatensystem, der Geometrie Shader macht aus jedem Punkt ein Quad und der Fragment Shader zeichnet das Quad mit einer transparenten Partikeltextur.

Spiegelung

Für die Spiegelung(Mirror.h) wird die Kamera an der Spiegelebene gespiegelt und aus der gespiegelten Perspektive in einen Framebuffer gerendert. Objekte hinter der Spiegelebene werden durch eine zusätzliche ClipPlane weggeclipt.

Gerendert wird der Spiegel auf ein Quad dessen uv Koordinaten genau den x/y Koordinaten am Schirm entspricht unter der Annahme, dass der Schirmkoordinaten von 0 bis 1 gehen.

Quellen:

- <http://www.opengl.org/resources/code/samples/mjktips/Reflect.html>
- <http://www.informatik-forum.at/showthread.php?80855-zus%E4tzliche-clipping-plane-OpenGL-3.2>

Bloom

Um einen Bloom Effekt zu erzielen wird zuerst die Gesamte Szene in einen Framebuffer gerendert. Anschließend wird der Kontrast des Bildes mit einem contrast Shader stark erhöht. Im nächsten Schritt wird ein niederes MipMap Level des Kontrasterhöhten Bildes weichgezeichnet. Das weichgezeichnete Bild wird mit dem Original kombiniert und tada → Bloom Effekt.

Tools

Alle Objekte wurden mit Blender modelliert und als Ogre xml exportiert.

Besonderheiten

Ogre XML Loader

Der MeshManager liebt sich die Meshdaten aus Ogre XMLs. Das Format ist recht straight forward. Zum auslesen des XML wird tinyxml verwendet.

LevelLoader

Lädt Levels aus XML Dateien.

Kollisionsberechnung

Zur Kollisionsberechnung werden Raycasts verwendet. Wenn ein Partikel von A nach B möchte, dann wird per Raycast geschaut ob dazwischen irgendein Dreieck geschnitten wird. Wenn ein normales Objekt geschnitten wird, dann wird das Partikel gelöscht und wenn ein Spiegel getroffen wird, dann wird das Partikel reflektiert.

Mathe

Die ganze Vektoren und Matrizenmathematik wird von eigens dafür geschriebenen Klassen behandelt.

Libraries

TinyXML

TinyXML is a small, simple XML parser for the C++ language. It is free and open source software, distributed under the terms of the license of zlib/libpng.

Siehe Wikipedia (<http://en.wikipedia.org/wiki/TinyXML>)

Glew

The OpenGL Extension Wrangler Library (GLEW) is a cross-platform open-source C/C++ extension loading library.

<http://glew.sourceforge.net/>

Glfw

GLFW is a free, Open Source, multi-platform library for creating OpenGL contexts and managing input, including keyboard, mouse, joystick and time.

<http://www.glfw.org/>