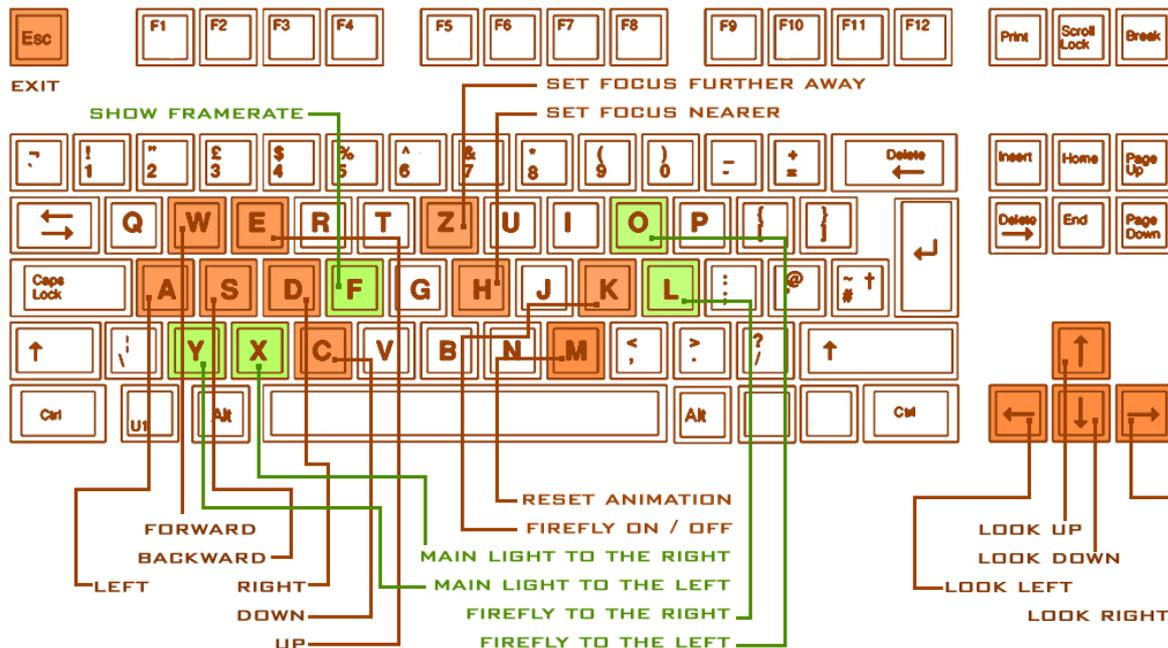


Der Monsterlöwenzahn

Steuerung:



Technisches:

Datenstrukturen für das Laden von Objekten

- Modelle: werden als *.obj eingelesen. Das Auslesen erfolgt in der Model.cpp Klasse.
- Level: befindet sich in der Level.txt Datei. Wird in der Loader Klasse ausgelesen.
- Texturen: sind als *.tga gespeichert und werden auch in der Loader Klasse gelesen.
- Shader: die Shader- Zuweisung ist in der Level.txt bei jedem Objekt angegeben.

Transformationen

Das Ausrechnen der ModelView-, Projektions- und Normal- Matrizen befindet sich momentan im Vertexshader, da wir die Funktionen von GLSL ausnutzen wollen. Die Parameter werden über uniform- Variablen von der Hauptanwendung an den Shadern weitergegeben.

Der Himmel rotiert langsam, um einen dynamischen Eindruck zu erzeugen.

Beleuchtung, Transparenz

Momentan sind zwei Lichtquellen mit diffusem und spekularem Anteilen im Vertexshader implementiert. Die eine ist statisch und dient der allgemeinen Szenenbeleuchtung, die andere ist mobil (ein Glühwürmchen), zusätzlich mit Attenuation versehen, und kann mit den Tasten O und L bewegt werden.

Die Transparenz wird momentan als Alpha- Wert aus den Texturen ausgelesen. Je nach Shader werden alle Fragmente mit Alpha- Wert $< x$ verworfen, wobei x von 0.001 bis 0.8 variiert.

Effekte

Wir haben folgende Effekte implementiert:

Allgemeines

Das Rendern in FBO's wird von der Klasse FrameBufferObjects erledigt. Dort werden die FBO's initialisiert und es existieren Methoden zum binden. Da das Rendern in eine FBO depth Textur bei uns leider nicht in FORWARD_COMPAT funktioniert hat, mussten wir es deaktivieren. Einziges weiteres mögliches Problem bezüglich der verwendeten OpenGL Version könnte das generieren lassen der MIP Maps beim Textur Erstellen unter glfw sein.

Shadow Maps

Quellen:

[1] Vorlesungsfolien: <http://www.cg.tuwien.ac.at/courses/Realtime/slides.html>

[2] Repetitoriumsfolien:

http://www.cg.tuwien.ac.at/courses/Realtime/repetitorium/2009/RTR-Repetitorium_2009.pdf

Wurden in den Shadern **TEXTURED** und **GRASS** implementiert.

Zuerst wird eine depth-Textur aus der Position der Lichtquelle gerendert und als FBO gespeichert.

Die Transformation der Vertices in Licht- Space wird im Vertexshader ausgeführt, da wir die Lichtquellenposition zur Verfügung haben und die Projektionsmatrix der Lichtquelle die der Kamera ist.

Depth of Field

Quellen:

[1] GPU Gems Pt.I Chapter 23. Depth of Field: A Survey of Techniques:

http://http.developer.nvidia.com/GPUGems/gpugems_ch23.html

[2] Paper „Real-Time Depth of Field Simulation“ by Riguer, Tatarchuk, Isidoro

http://ati.amd.com/developer/shaderx/ShaderX2_Real-TimeDepthOfFieldSimulation.pdf

[3] Encelo's Projectz <http://encelo.netsons.org/programming/opengl>

Wurde im Shader **DEPTHOFFILED** implementiert.

In einem ersten Render-Schritt wird die Szene gerendert und in ein FBO mit 2 Texturen gespeichert. Depth und Color Textur. Der Blur Effekt wird in 2 Pässen erzeugt. Im ersten Pass wird in y-Richtung ein blur mit (0.3, 0.2, 0.15)-Filterkern, auf die in das FBO gerenderte Bild, angewandt. Im 2ten Schritt wird der Blur in x-Richtung angewandt. Berücksichtigt wird auch der depth-Wert von Pixeln beim Blur. Punkte die näher an der Kamera liegen als der aktuelle Depth-Wert werden nicht berücksichtigt. Somit sind die Kanten nahe liegender Objekte nicht „verwaschen“.

Animated Grass

Quellen:

[1] GPU Gems Pt.I Chapter 7 Rendering Countless Blades of Waving Grass:

http://http.developer.nvidia.com/GPUGems/gpugems_ch07.html

[2] OpenGL Shading Language, Randi J. Rost, 3rd Edition, Chapter 16 Animation

Wurde im Shader **GRASS** implementiert.

Die Klasse MultipleGameObject kümmert sich um die Erstellung eines Gras Objektes das aus mehreren Grasbüscheln besteht. Eingabe ist dabei ein Objekt das zu duplizieren ist mit dem Shader „grass“.

In der Klasse wird anhand einer Verteilungstextur (grauwert, im raw Format) eine zufällige Grasanordnung mit der Dichte als Grauwert erzeugt. Schwarz entspricht keinem Gras und Weiß mit höchster Dichte. Notwendig sind auch die Grenzen des Objektes auf dem das Gras wachsen soll und der Höhenwert der in einem zusätzlichen Kanal der Textur.

(Zusammenfassend: Verteilung im r-Kanal, Höhe im b-Kanal)

Beim Zeichnen wird das Gras anhand einer Textur animiert. (Ursprünglich auch der Höhe nach versetzt. Jedoch wird das jetzt vorausberechnet.) x und y-Richtung der Textur bestimmen einen Bewegungsvektor (da es sich bei gleichzeitiger Visualisierung Übung so ergeben hat). Dabei werden pro Vertex lookups in der Richtungstextur gemacht. Zusätzlich gibt es eine Windfunktion die den Einfluss des Vektors gewichtet. Hier wurde eine Sinusfunktion für die Schwingung mit einer logarithmischen Verzerrung gewählt um einen hübscheren Effekt zu erzielen.

Generell ist das Gras belastender für die Grafikkarte da relativ viele Büschel auf einmal dargestellt werden. Ursprünglich war die Herangehensweise viele Gras-Objekte zu erzeugen. Da dann jede Büschel mit uniform Variablen versetzt werden musste war die Implementierung dementsprechend langsam. Deshalb werden vorher mehrere Gras-Objekte zusammengefügt und gemeinsam Animiert. Somit gibt es eigentlich nur 10-15 Gras Objekte zu je 3000 Büscheln. Die Animation der Vertices wird anhand deren UV-Koordinaten bewerkstelligt wie im Artikel beschrieben.

Animation

Der Löwenzahn soll sein ganzes Lebenszyklus zeigen – z.B. durch Morphing. Quellen:

[1] OpenGL Shading Language, Randi J. Rost, 3rd Edition, Chapter 16 Animation

Wurde im Shader **MORPHER** implementiert.

Es wird je ein Modell für die 4 Schlüsselpositionen geladen. Das erste befindet sich im File **DemoScene.txt**, die nächsten im File **DemoTargets.txt**. Im Shader befinden sich die Vertex- und Textur-Daten von den Modellen, zwischen denen gerade das Morphen durchgeführt wird. Als Parameter dafür wird die Variable **weight0** übergeben. In der Methode **drawAll** wird das Schalten zwischen Schlüsselpositionen (oder Targets) durchgeführt.

Particle Systems

Quellen:

[1] Freies OpenGL- Wissen für alle! http://wiki.delphigl.com/index.php/GLSL_Partikel_2

Wurde im Shader **PARTICLE** implementiert.

Es wird ein Modell als Referenzgeometrie für die Partikel geladen. Das Verhalten der Partikel wird in der Klasse **ParticleSystem.cpp** verwaltet. Ihre Bewegung wird durch eine Polynomfunktion bestimmt, die sie sanft verlangsamt. Mit der Zeit verlieren sie „Lebenskraft“ und verblassen, bis sie von dem Fragmentshader verworfen werden. Ihr Verhalten verläuft synchron mit der Animation des Löwenzahns.