Abgabe3

Cannonball

Götz Johann - 532 - 0626963 - johann.goetz@heisl.org

Hecher Michael - 532 - 0625134 - michael.hecher@student.tuwien.ac.at

Inhalt

Implementierung

Gameplay

Vor dem Abschuss eines Cannonballs kann der Spieler seine Umgebung mit einer freien Kamera erkunden. Diese wir per Maus und den WASD-Tasten der Tastatur gesteuert.

Wenn der Spieler weiß in welche Richtung er schießen will, kann er über die Rechte-Maustaste in die „Cannonball-Ansicht“ umschalten (vor dem ersten Abschuss ist der Cannonball noch nicht sichtbar). In der „Cannonball-Ansicht“ wird das Bewegen der Maus die Schussrichtung festgelegt (der Cannonball wird in Blickrichtung der Kamera abgeschossen). Über die Linke-Maustaste wird der Schuss ausgelöst. Je länger die Maustaste gedrückt wird, desto stärker der Schuss. Ab einer gewissen Zeit wird der Cannonball jedoch automatisch abgeschossen. Nach dem Abschuss kann der Cannonball abgebremst werden, um ihn schneller zum Stilstand zu bringen. Allerdings muss zuerst eine gewisse Geschwindigkeit unterschritten werden.

Mit der R-Taste kann der Cannonball an die Startposition zurückgesetzt werden. Sollte der Cannonball aus dem Level fliegen kann mit der R-Taste neu begonnen werden.

Mit dem Maus-Rad kann zwischen verschiedenen Cannonballs gewählt werden (Normal, Schnell, Schwer und Bombe).

Das Ziel wird durch einen blauen Glaszylinder markiert. Der Start durch einen roten Zylinder.

Steuerung

Esc - Mit der Escape-Taste kann das Programm beendet werden.

F2 - Framerateanzeige ein/aus

F3 - Wire Frame ein/aus

F4 - Texturqualität verändern: Nearest Neighbor/Bilinear

F5 - MipMapping-Qualität ändern: Aus/Nearest Neighbor/Linear

F8 - Viewfrustum-Culling ein/aus

F9 - Transparenz ein/aus.

L Maus – Abschuss des Cannonball. Je länger gedrückt, desto stärker der Abschuss.

R Maus – In den Abschussmodus wechseln, bzw. Cannonball anhalten wenn eine gewisse Geschwindigkeit unterschritten wird.

Maus Rad – Cannonball Typ wechseln.

Effekte

Parallax

Im Vertex Shader wird anhand der Nomalen, Tangete und Binormalen des Vertex eine Matrix zur Transformation des Lichtvektors in den Texturraum berechnet. Im Fragment Shader wird unter Zuhilfenahme der Höhe an der Texturposition des Fragments eine neue Texturposition berechnet und die neue Normalen- und Farbinformation für die Berechnung der Farbe des Fragments herangezogen. Zusätzlich wird die Höhe dazu verwendet um das Resultat abzudunkeln. Dies erhöht den Tiefeneffekt.

/bin/shaders/parallax.[vert|frag]

Depth of Field

Zuerst wird die Szene normal in einen Textur-Framebuffer gerendert und zusätzlich die Tiefeninformationen jedes Pixels im Alphakanal gespeichert. In einem Blur Shader wird dieses „scharfe“ Bild über einer Filter-Kernel „geblured“ und in einen weiteren Textur-Framebuffer gespeichert. Zuletzt werden anhand der Tiefeninformationen die beiden Bilder „vermischt“. Fragmente, welche sich weiter weg befinden, erhalten mehr Farbe aus dem verschwommenen Bild

und umgekehrt. Je näher ein Fragment an der Kamera liegt, desto größer der Anteil des scharfen Bildes.

/bin/shaders/blur.[vert|frag]

/bin/shaders/depthOfField.frag

Environment mapping (Glas effect)

An jedem Fragment wird der Lichtvektor an der Oberflächennormale reflektiert. Der reflektierte Vektor wird verwendet, um aus einer Cube Map die Texturinformation zu holen. Die Fragmentfarbe ergibt sich aus der Kombination von Environment Map (Cube Map die auch als Skybox verwendet wird), Oberflächenfarbe und Winkel zwischen Oberflächennormalen und Blickvektor (Kamera zu Oberflächenpunkt). Der Winkel gibt die Transparenz des Objektes an, wodurch das Reflektionsverhalten von Glas simuliert wird.

/bin/shaders/glas.[vert|frag]

Animierte Objekte

Bewegte Objekte sind ausschließlich Objekte die durch die PhysX Engine simuliert werden. Dies sind beispielsweise Hindernisse wie Kisten oder die Cannonballs (siehe). Zu Beginn eines Levels bewegen sich allerdings diese Objekte nicht. Erst wenn ein Cannonball auf eines der durch die PhysX Engine simulierten Objekte trifft oder der Cannonball „Bomb“ eingesetzt wird, ist eine Animation zu sehen. Siehe */src/physic/header/IBody.h* und */src/physic/src/Physic.cpp*.

Frustum Culling

Aus der View-Projection-Matrix werden die sechs Ebenen des Frustums berechnet. Danach werden Bounding Spheres auf Berührung mit dem Frustum getestet. Berührt eine Bounding Sphere das Frustum oder liegt sie im Frustum, wird das Objekt gezeichnet (siehe /src/util/header/frustum.inl).

Experimentieren mit OpenGL

Framebuffer Objekte werden im Depth of Field Shader verwendet.

Vertex-/ Indexbuffer Objekte werden verwendet.

Vertex Arrays werden verwendet um Vertex Buffer Objekte zu beschreiben (siehe OpenGL Spec 3.3 Kapitel 2.8 Vertex Arrays).

Wireframe, Framerate, Texturqualität, MipMapping Qualität und Frustum Culling sind implementiert.

Spieldesign

Das Spieldesign wurde leicht abgeändert, da Testpersonen darauf hingewiesen haben, dass es ihnen besser gefallen würde, wenn man Hindernisse aus dem Weg räumen könnte statt diese mit „Hilfsobjekten“ zu umgehen. Das neue Spielprinzip lässt sich nun wie folge formulieren:

Zu Beginn besitzt der Spieler eine begrenzte Anzahl an Cannonballs. Mit diesen muss er versuchen einen Bestimmten Zielort zu erreichen (ähnlich einem Minigolf Spiel). Hindernisse die den Weg zum Ziel blockieren können mit speziellen Cannonballs aus dem Weg geräumt werden. Dabei kommt es auch darauf an welche Cannonballs verwendet werden. In manchen Levels wird ein leichter, schneller Cannonball von Vorteil sein, da man mit Ihm Hindernisse als „Banden“ verwenden kann. In anderen Levels muss erst der Weg zum Ziel freigesprengt werden. Für jeden Level gibt es jedoch mehrere Lösungswege, um auch ohne Bomben oder leichten Cannonballs ans Ziel zu kommen.

Unterschiedliche Cannonballs können in den Levels eingesammelt werden. Erreicht der Spieler den Zielort, gelangt er zum nächsten Level. Verbraucht er alle seine Cannonballs ist das Spiel verloren. Sind alle Levels bewältigt ist das Spiel gewonnen.

Beleuchtung

Die Levels werden über einen Parallax Shader mit vier Lichtquellen ausgeleuchtet. Siehe */bin/shaders/parallax.vert* und */bin/shaders/parallax.frag*. Diese Lichtquellen werden über Blender exportiert.

Der Parallax Shader verwendet Texturarrays (`sampler2DArray`) für Texturinformationen (ein Array für Farbinformationen, und eines für Detailinformationen wie Normalen und Höhe). Da nur sehr wenige Texturen für einen Level benötigt werden, bietet diese Variante den Vorteil, dass Texturarrays zwischen unterschiedlichen Shadern „geteilt" werden können. Somit müssen die Texturarrays nur einmal vor der draw-Schleife mit `glBindTexture` gebunden werden.

Datenstrukturen

Zur Darstellung von Meshes werden ein Vertex- und ein Indexbuffer verwendet. Objekte liegen in diesen Buffern hintereinander wie in einem Stack.

Zum zeichnen dieser Objekte wird der OpenGL Befehl `glDrawElementsBaseVertex` verwendet. Dazu wird der „Base" Vertex, der „Start" Index und die Anzahl der Indizes des Objektes übergeben. Durch diese Form der der Geometriedaten muss das Spiel nicht zwischen verschiedenen Vertex- und Indexbuffern mit `glBindBuffer` wechseln. Ein weiterer Vorteil von `glDrawElementsBaseVertex` besteht darin, dass die Größe der Indices auf 2 Byte (unsigned short) beschränkt werden kann, auch wenn die Vertices eines Objektes hinter dem Index 65536 im Vertexbuffer liegen (siehe Datei *Render.cpp* unter */src/Render/src/*; insbesondere die Methoden *`Render::update`* und *`Render::createGeometry`*).

Ein Vertex besitzt folgende Attribute:

- Position
- Normale
- Tangente
- Bitangente/Binormale
- Vertexfarbe
- Texturkoordinaten

Die Informationen einer Mesh werden aus einer Datei (*.rvl) geladen. Diese werden mithilfe eines Blender Plugins erzeugt. Der Sourcecode dieses Plugins befindet ist unter */src/blender/rvl.py* zu finden. In diesem Plugin werden auch die Tangenten und Bitangenten für den Parallax Shader berechnet (siehe *`Indices::computeTangent`*). Des Weiteren werden auch noch Informationen zur Physik der Objekte und Lichtquellen exportiert. Items sowie Start und Ende eines Levels werden in einer gleichnamigen *.rvl.def Datei gespeichert.

Features

Ein besonderes Merkmal der Renderengine ist der Verzicht überflüssiger Buffer und Texturen. Es gibt nur einen Vertexbuffer, einen Indexbuffer und jeweils ein Texturarray für Farb- und Detailinformationen. Dadurch muss während des Zeichnens nicht zwischen diversen Buffer gewechselt werden. Somit reduziert sich das Zeichnen der Szene auf das Ändern von Uniform Variablen in Shader und das aufrufen von `glDrawElementsBaseVertex` für die einzelnen Objekte.

Da Frustum Culling verwendet wird, werden nicht sichtbare Objekte nicht gezeichnet.

Durch Parallax Mapping erhalten Texturen einen hohen Detailgrad.

Aufgrund der verwendeten, rechenintensiven Fragment Shader wird ein Depth-Only Pass am Anfang des Draw-Loops ausgeführt, wodurch das mehrfache Berechnen eines Pixels vermieden wird. Dies gilt natürlich nur für Objekte ohne Transparenz.

Externe Libraries

PhysX

Für die Physiksimulation wird die PhysX-Technologie von nVidia verwendet (siehe */src/physic/* und *http://developer.nvidia.com/object/physx.html*).

GLEW
Zum laden der OpenGL Extensions und OpenGL Funktionen wird GLUW verwendet.

OpenAL
Zum Abspielen von Soundeffekten.