Fish Salad – 3. Abgabe

Kurzbeschreibung:

In der nun endgültigen Fassung unseres Spiels tauchen Sie in der Rolle eines Raubfisches in die gefährliche Unterwasserwelt ein. Ihr Ziel ist es, sich in der Nahrungskette weiter nach oben zu arbeiten, indem Sie die Fische auf Ihrem Speiseplan jagen, dabei wachsen und Ihren größeren, noch unbezwingbaren Fressfeinden aus dem Weg gehen. Mit jedem komplettierten Level wechseln Sie in ein anderes Unterwasserszenario, d.h. Sie spielen einen anderen Fisch, werden von anderen Fischen umgeben und müssen daher Ihren Speiseplan sowie die Liste der Feinde anpassen.

Datenstrukturen:

Wurden im Laufe der Zeit immer wieder verfeinert. Nach einigen Experimenten mit dem 3DS-Format setzten wir zum Zeitpunkt der 2. Abgabe noch auf das MD2-Format, da sich keyframe-basierte Animation ideal für einfache Fischbewegungen eignet und ein aufgebohrter Loader (unterstützt Models mit bis zu 65.536 vertex-TexCoord-Paare und bis zu 65.536 Dreiecke) relativ einfach implementiert werden konnte. Da das MD2-Format allerdings aufgrund der seinerzeit aktuellen Hardware in einigen Punkten, u.a. der Präzision der vertex-Positionen, der Texturkoordinaten und v.a. der Normalvektoren (welche wir vorher bei jedem Programmstart für jedes Modell neu berechneten und so die Ladezeit signifikant erhöht wurde) gewichtige Defizite aufweist, entschlossen wir uns kurzerhand, ein **eigenes Model-Format** zu entwickeln und einzusetzen. Dieses auf den Namen FSM (Fish Salad Model) getaufte Format baut auf dem MD2-Format auf, ist daher binär und bietet keyframe-basierte Animation mit fixen Texturkoordinaten und Dreiecken, die Vertexkoordinaten und die gemittelten Normalvektoren werden in den keyframes gespeichert. Allerdings benutzten wir generell je ein word (16 bit) zur Kompression der einzelnen Koordinaten an Stelle eines einzelnen bytes (256fache Präzision) und speicherten auch die Koordinaten des Normalvektors anstatt eines heutzutage unbrauchbaren Index‘. Zur Konvertierung exportierten wir unsere in **3ds max** modellierten, texturierten und animierten Objekte frame für frame als simple Wavefront objects (*.obj, sehr einfach zu parsen) und schrieben ein kleines command-line-utility, welches mehrere (jeweils einen keyframe repräsentierendem) .obj-files zu einem kompakten .fsm-file fusioniert. MD2-Modelle werden weiterhin unterstützt, allerdings benutzen wir in der vorliegenden Version aufgrund der genannten Vorteile exklusiv unser proprietäres Format.
Die Modelle werden wie das Spiel selbst durch **XML-Dateien** konfiguriert (und durch Benutzung der MSXML-COM-Komponenten geparst), wodurch Fein-Tuning und Erweiterung des Spiels mühelos zu bewältigen sind.
Details: Neben einer Model- und einer Texture-Klasse mit entsprechenden Managern wird eine ModelInstance-Klasse benutzt. In der Model-Klasse werden die keyframes gespeichert, in welchen wiederum die Eckpunkte (werden evtl. dupliziert, wenn ein Eckpunkt verschiedene Texturkoordinaten besitzt) und ihre gemittelten Normalvektoren gespeichert werden. Die Texturkoordinaten der Eckpunkte und die Dreiecke mit den Eckpunkt-Indizes werden frame-unabhängig in der Model-Klasse gespeichert.

Kameramodell:

Der Fisch des Spielers wird in einer **3rd-persion-view** betrachtet, d.h. die Kamera befindet sich fix relativ zur Spielerposition und -rotation. Der Spieler bewegt sich mit konstanter Geschwindigkeit in einem virtuell unendlichen Ozean mit level-abhängiger Tiefe fort (eigentlich bewegt sich der Spieler selbst nicht, sondern wird im world-space stets um den Ursprung gerendert; alle anderen Objekte werden in die entgegengesetzte Richtung bewegt). Durch die Benutzung der Pfeil- oder WASD-Tasten wird er um seine yaw- und pitch-Achsen gedreht, wodurch sich selbstverständlich die Richtung des Fisches entsprechend verändert.

Bewegte Objekte:

Alle Objekte verfügen über eine eigene **Vorwärtsbewegung**. Generell enthält jedes Objekt eine eigene Modelmatrix, welche die Position, Rotation und Skalierung des Objektes angibt, sowie einen skalaren Wert für die Geschwindigkeit. Eine Kollision von „ebenbürtigen“ (sich nicht fressenden) Fischen wird durch eine **Rotation** und eine **Seitwärtsbewegung** behandelt, sodass die Fische sich in der Regel wieder voneinander entfernen.

Texture mapping:

Bei den Texturen benutzen wir verschiedene Formate. Für die Texturen der Modelle (allesamt ohne Alpha-Kanal) setzten wir das komprimierte **DXT1**-Format (natürlich ohne alpha-bit) ein, da die fixe Kompressionsrate von 6:1 mit minimal sichtbarem Qualitätsverlust beeindruckend ist. In diesen mit Photoshop und einem nVidia-plugin kreierten .dds-files wurden außerdem sämtliche Mipmaps (unter Benutzung des hochqualitativen Mitchell-Filters) integriert. So sind diese Texturen erstens sehr schnell geladen, da keine Dekompression durch eine externe image-library, keine Laufzeit-Kompression durch OpenGL und keine Laufzeit-Generierung der Mipmaps vonnöten ist, und zweitens benötigen sie nur ein Sechstel des Grafikspeichers.
Bei den restlichen Texturen (z.B. für den HUD) benutzten wir generell das **PNG**-Format, welches eine verlustfreie Kompression bietet, und das ebenfalls verlustfreie **TIFF**-Format bei Texturen, welche einen Alpha-Kanal erfordern. Diese Texturen werden auch nicht zur Laufzeit komprimiert, bieten also eine optimale Qualität. Diese „generischen“ Texturen werden durch die Einbindung der DevIL-Bibliothek geladen.
Bei den selbsterstellten mappings der Modelle kam **UV-mapping** zum Einsatz.
Die caustics wurden durch **multi-texturing** realisiert.

Beleuchtung/Materialien:

Alle Objekte werden durch **eine direktionale Lichtquelle** beleuchtet. Diese befindet sich hinter und ober der ursprünglichen Spielerposition, dreht sich allerdings natürlich nicht wie die Kamera mit dem Spieler mit. Allen Objekten wird dasselbe Material zugewiesen. Sofern vom System unterstützt kommen auch **Vertex- und Fragmentshaders** zum Einsatz, welche in **Cg** implementiert wurden. So unterstützt das shader-Paar **per-pixel-lighting**, **per-pixel-fog** und **caustics**.

Gameplay:

Wie bereits eingangs erwähnt, geht es darum, (meist kleinere) Futterfische durch eine Kollision zu fressen und nicht mit (meist größeren) Fressfeinden zu kollidieren. Levelintros sollten weitere Auskunft über den Speiseplan und die Feinde geben.
Bei jeder Kollision mit einem Futterfisch wird ein Fresssound abgespielt und der gefressene Fisch verschwindet im Bauch des Alter Egos. Außerdem wächst dieser dabei leicht und seine score und sein level-progress (sichtbar in der linken oberen Ecke des HUDs) steigen, alles in Abhängigkeit des Nahrungsgehaltes des gefressenen Fisches. Die anderen Fische können sich natürlich auch gegenseitig fressen und somit dem Spieler wertvolle Proteine wegschnappen.
Kollidiert man mit einem Feind, ertönt ein durch Mark und Bein gehender Schrei und ein Leben des Alter Egos wird abgezogen (sichtbar in der rechten oberen Ecke des HUDs). Außerdem ist er für einige Sekunden unverwundbar, was durch Blinken und einen motion-blur-Effekt visualisiert wird. Wurde das letzte Leben verloren, ist das Spiel beendet, das Programm muss mit Esc terminiert werden.
Wurde das score-Limit des Levels erreicht, erscheint ein Intro des neuen Levels mit nützlichen Informationen. Mit Enter wird das Level mit einem zusätzlichen Bonusleben begonnen, wobei das Alter Ego nun durch einen neuen Fisch repräsentiert wird, welcher sich in einer neuen Umgebung mit anderen Fischen befindet und sich demzufolge anpassen muss.
Hat man alle 3 Levels geschafft, ist das Spiel beendet und muss mit Esc terminiert werden.

Nichttriviale Objekte:

Über solche verfügen wir genug (9 Fischmodelle), unser einziges triviales Szenenobjekt ist der Boden. Informationen zum eigenen Format finden sich weiter oben im Datenstrukturen-Unterkapitel.
Die Bandbreite der Komplexität reicht von 2.000/3.000 (weiß ich nicht mehr auswendig :)) bis zu ziemlich genau 10.000 Dreiecke (das Maximum weiß ich natürlich noch :)) pro Modell. Der wireframe-Modus kann mit der F3-Taste aktiviert und wieder deaktiviert werden.

Animierte Objekte:

Wie bereits im Datenstrukturen-Unterkapitel erwähnt, benutzen wir keyframe-basierte Animation in unserem Modell-Format und interpolieren zwischen diesen keyframes. Von den 9 Modellen sind 7 animiert, bei 2 Modellen hatten wir leider etwas Probleme (beim Rochen, da dessen „Flügelschwünge“ für uns Amateure nicht trivial zu animieren war, und beim Leopardenhai, da sich hierbei regelmäßig 3ds max mit einem unschönen crash verabschiedete). Im Allgemeinen wurden je 4 (davon 3 unterschiedliche) keyframes für die Bewegung der Schwanzflosse generiert (mit dem object-space-path-deformer-modifier), einzig beim Schwertfisch experimentierten wir mit 8 (davon 5 unterschiedlichen) frames und erzielten so eine etwas flüssigere Animation.

Beschleunigung der Sichtbarkeitsberechnung:

View frustum culling wurde natürlich implementiert und lässt sich mit der F8-Taste deaktivieren und wieder aktivieren. Hierbei wird vor dem Rendern eines Objekts geprüft, ob sich die bounding sphere des Objekts komplett außerhalb einer der 6 Ebenen der view-frustum-Pyramide befindet und in diesem Fall das Objekt nicht gerendert (und auch nicht zwischen den keyframes interpoliert logischerweise). Die Überprüfung findet im world-space statt; zur Extraktion der Ebenen aus der kombinierten projection- und view-Matrix wurde ein nützliches PDF-Dokument (http://www2.ravensoft.com/users/ggribb/plane%20extraction.pdf) herangezogen.

Transparenz-Effekte:

Transparenz-Effekte werden einerseits beim HUD- und font-rendering benutzt. Außerdem werden sie beim motion-blur-Effekt benutzt, um die alte Szene mit einem geringeren Alpha-Wert zu rendern und den color-buffer des FBOs über den back-buffer zu blenden. Des Weiteren benutzen Level-Intros und Game-Outros Transparenz-Effekte.

Experimentieren mit OpenGL:

Alle geforderten „Experimente“ wurden implementiert. Die display lists lassen sich nicht deaktivieren, da wir diese nur für das font-rendering benutzen, da der Rest zu dynamisch ist, um mit display lists höhere Performance zu erreichen. Insbesondere VBOs werden exzessiv benutzt (jedes Modell besitzt einen VBO für die fixen Texturkoordinaten und einen für die fixen Dreiecke mit den vertex-indices; jede Instanz eines Modells enthält einen interleaved VBO mit den dynamischen vertex-Koordinaten und den assoziierten Normalvektoren).
Auch die Transparenz-Effekte können nicht deaktiviert werden, da ansonsten der HUD zu hässlich und der motion-blur-Effekt nicht zu benutzen ist.
Um diese 2 Punkte zu kompensieren, lassen sich einerseits mit der F11-Taste die shaders deaktivieren, wodurch das per-pixel-lighting und die caustics deaktiviert werden. Außerdem wurde seit der 2. Abgabe anisotropisches texture-filtering hinzugefügt. Mit der F12-Taste kann man durch

die verfügbaren Anisotropie-Level des Texturfilters iterieren (standardmäßig wird 8fach anisotropisches trilineares filtering verwendet).

Spezialeffekte:

Per-pixel-lighting:

Inspiriert durch einen auf DirectX und Assembler-shaders-code zugeschnittenen MSDN-Artikel (http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dndrive/html/directx11192001.asp) wurde **object-space**-per-pixel-lighting nach dem **Blinn-Phong-Beleuchtungsmodell** in den shaders implementiert (object-space-lighting findet in dem Artikel keine Erwähnung, eignet sich aber unserer Meinung nach für unser Szenario weitaus besser als world-space-lighting, welches eine doppelte Transformation des Eckpunkts im vertex-shader erfordern würde).
Vor dem Rendern der Eckpunkte eines Modells werden die Positionen der (direktionalen oder nicht-direktionalen) Lichtquelle und des Betrachters in den object-space des zu rendernden Objekts transformiert und dem vertex-shader übergeben. Dieser berechnet für jeden gerenderten Eckpunkt die normalisierten Vektoren zur Lichtquelle und zum Betrachter. Anhand dieser 2 Vektoren wird der von Blinn eingeführte Halbvektor durch Addition und Normalisierung berechnet. Licht-, Halb- und Normalvektoren werden an den fragment-shader weitergeleitet, d.h. diese werden über den Fragmenten interpoliert. Daher erfolgt im fragment-shader für jedes Fragment eine weitere notwendige Normalisierung, bevor die Winkel zwischen Normal- und Lichtvektor (bestimmt die Intensität des diffusen Beleuchtungsterms) sowie zwischen Normal- und Halbvektor (bestimmt abhängig vom 1. Winkel sowie einem shininess-Parameter die Intensität des specular-Terms) berechnet werden und die finale Beleuchtungsformel des Fragments so maßgeblich beeinflusst wird.

Per-pixel-fog

Pixelgenauer Nebel ist zwar heutzutage bereits über fixed-function-Funktionalität gegeben, allerdings mussten wir diese aufgrund der Benutzung von shaders nachprogrammieren. Aus diesem Grund leitet der vertex-shader die in den clip-space transformierte Position des Eckpunkts an den fragment-shader weiter. Dieser Vektor wird also wieder über den Fragmenten interpoliert. Im fragment-shader wird die Länge des Vektors bestimmt und diese in die (GL_EXP2-)Nebel-Formel laut OpenGL-Dokumentation (einzige Referenz) eingesetzt.

Caustics

In einer Unterwasserwelt dürfen caustics natürlich nicht fehlen. Aus diesem Grund bedienten wir uns 32 frei verfügbarer caustic-Texturen, welche die zeitliche Veränderung der caustics repräsentieren. Per multi-texturing wird also abhängig von der verstrichenen Zeit seit dem letzten frame die vorherige oder eine neue caustic-Textur als 2. Textur gebunden. Im vertex-shader werden anhand der Distanzen des in den clip-space transformierten Eckpunkts von 2 fixen Ebenen automatisch die Texturkoordinaten für die caustic-Textur berechnet und an den fragment-shader weitergeleitet. Dieser liest das entsprechende Texel aus und benutzt den Grauwert, um die diffusen und specular-Terme der Beleuchtungsformel zu skalieren.
Die Texturen und der Ansatz wurden von einem Tutorial (http://www.opengl.org/resources/code/samples/mjktips/caustics/) übernommen. Allerdings flossen genügend eigene Änderungen ein, da das recht alte Tutorial noch multi-pass-rendering ohne shaders benutzt sowie die Koordinaten im object-space berechnet.

Motion-blur:

Ist das Alter Ego des Spielers temporär unverwundbar, wird dies u.a. durch einen motion-blur- bzw. tracing-Effekt angezeigt. In diesem Fall benutzen wir eine Textur, in welche unter Benutzung von **Framebuffer Extensions (FBO)** die Szene (Boden + Fische) hinein gerendert wird. Vor jedem rendering pass werden die Alpha-Werte der Textur abhängig von der verstrichenen Zeit verkleinert,

damit die alte Szene etwas „ausbleicht“. Anschließend wird die neue Szene mit vollem Alpha-Wert in die Textur gerendert, sodass die neue Szene die alte komplett überlagert. Anschließend wird die Textur unter Benutzung von **alpha-blending** über den back-buffer geblendet; sie enthält somit die alte Szene für den nächsten frame.
Den Ansatz lieferte ein Artikel (http://www.codeproject.com/opengl/MotionBlur.asp), allerdings wird dort höchstwahrscheinlich kein FBO benutzt (code wurde ignoriert, da die 3 Schritte sehr intuitiv sind).

Sonstige Features:

Quaternionen:
Zur Beschreibung von Rotationen im Raum werden Quaternionen benutzt. Diese sorgen für genauere Ergebnisse, wenn Rotationen verknüpft werden und sind in diesem Falle auch etwas performanter.

SSE-intrinsics:
Wo sinnvoll, werden optional (durch eine Präprozessor-Direktive) SSE-intrinsics zur Erhöhung der Performance eingesetzt, z.B. beim Interpolieren zwischen zwei keyframes eines Modells. Bei Benutzung von VBOs werden die Eckpunkte samt Normalvektoren direkt in den Puffer im Grafikspeicher hinein interpoliert.

Eigene collision detection:
Wir implementierten unsere eigene collision detection. Aufgrund der meist länglichen Form der Fische entschieden wir uns für bouding volumes in Form von Ellipsoiden, welche anhand der bounding boxes im object-space berechnet werden und effizient in andere spaces transformiert werden können. Die Idee zur Benutzung dieses speziellen bounding volumes stammt zu 100% von uns. Eine Überlappung zweier bounding ellipsoids führt zu einer Kollision, welche entweder durch Fressen oder durch Ausweichen behandelt wird.
Um eine bessere Performance zu erreichen, erfolgt der Ellipsoid-Check erst nach der Überlappung der gröberen bounding spheres.

Externe Bibliotheken:

- **DevIL** zum Laden aller gängigsten Bildformate
- **OglExt** (http://www.julius.caesar.de/index.php/OglExt) für die komfortable Benutzung der OpenGL extensions bzw. von OpenGL-Funktionalität, welche erst nach v1.1 Einzug hielt
- nVidia **Cg** Toolkit (http://developer.nvidia.com/object/cg_toolkit.html) zur Laufzeit-Übersetzung von Cg-shadern samt Anpassung an die jeweilige Grafikkarte
- **Audiere** (http://audiere.sourceforge.net/) zum Abspielen gängiger Audioformate