Informationsdokument zu „The Tale of the Cat“, Abgabe 2

- Neue Version vom 24.6.2013 -

Anmerkungen:

- Die Ladezeit der .exe-Datei kann einige Sekunden in Anspruch nehmen.
- Beim Ausführen der .exe-Datei erscheint im Konsolenfenster die Meldung „Ein GL-Fehler ist aufgetreten!“ - diese ist als gegenstandslos zu betrachten; es liegt kein Fehler vor und das Spiel startet problemlos.
- Das Spiel läuft stets im Vollbildmodus.

Implementierung und Beschreibungen bzgl. Erfüllung der Bewertungskriterien:

Allgemein:
SceneObject stellt ein Objekt der Szene dar und bietet für alle gleichermaßen eine show()-Methode an, mit der das Objekt am Bildschirm dargestellt wird. Außerdem muss von jeder vererbten Klasse die move()-Methode implementiert werden, die zur Berechnung der Position dient.

„Free movable camera“:
Die Bewegung der Kamera ist fest an die Bewegung der Spielfigur gebunden, d.h. sie kann jederzeit frei bewegt werden, indem der Avatar bewegt wird: Fliegt die Schnecke höher oder niedriger, so folgt die Kamera ihr; dreht der Protagonist sich, so dreht sich auch die Kamera.

„Moving objects“
Neben – natürlich – dem Helden des Spiels wurden auf dem Testlevel noch zahlreiche weitere bewegliche Objekte platziert: „Günther“ (Guenther.cpp) genannte Backstein-Golems in Würfelform, welche in kreisförmigen Flugbahnen patroullieren und Lasergeschoße (Klasse „GegnerProjektil“) auf die Spielfigur abfeuern. Die Bewegung dieser Schurken, die nun auch mit einem sich drehenden Rotorblatt versehen wurden (siehe Kapitel „Animated Objects“), wird durch die oben erwähnte move()-Methode (die Variante aus der Guenther-Klasse) geregelt.
Abgefeuerte Projektile, ob nun von Gegnern oder die von der Schnecke geschossenen Wollknäuel (Klasse „Projektil“) werden jeweils innerhalb der Klasse eines schießenden Objekts erstellt und bewegen sich dann mit ihrer eigenen move()-Methode (spezifiziert in den besagten „Projektil“- und „GegnerProjektil“-Klassen) fort; dass diese immer wieder aufgerufen wird, solange sie existieren, wird wiederum innerhalb der move()-Methode des Schützen sichergestellt.
Projektile (woher sie auch kommen) werden zerstört, wenn sie a) einen Gegner (im Falle eines Wollknäuels) bzw. die Schnecke (im Falle eines gegnerischen Schusses) berühren und damit Schaden anrichten, b) wenn sie auf ein Hindernis bzw. die Umgebung treffen oder c) bereits fünf Sekunden existieren, ohne mit jemandem oder etwas kollidiert zu sein (dann sind sie so weit weg, dass sie für uns irrelevant sind, und werden „entsorgt“).

„Animated Objects“:
Wie oben beschrieben werden die animierten Objekte durch die Günthers repräsentiert, welche aus zwei separaten Meshes zusammengesetzt sind und somit hierarchische Animationen verwenden: Einerseits der Haupt-Körper des Gegners, andererseits sein Rotorblatt, welches auch über eine eigene .cpp-Klasse samt move()-Methode verfügt und sich stets dreht, solange der zugehörige Günther existiert. Wird der besagte Günther besiegt, so wird in seiner kaputt()-Methode, die bewirkt, dass er aus dem Level entfernt wird, gleichzeitig auch die kaputt()-Methode des Rotorblatts aufgerufen (das Rotorblatt gilt von der Programmlogik her auch als „Gegner“, aber verfügt über keine Kollisionsabfrage, richtet somit auch keinen Schaden an und ihr Bestehen ist tatsächlich komplett an den zugehörigen Günther-Körper gebunden).

„Complex Objects“:
Wie schon bei der letzten Abgabe wurden sämtliche Modelle und Texturen von uns selbst designt; die einzige Ausnahme ist die Grastextur am Boden, welche aus GIMP stammt, und es soll noch erwähnt werden, dass zwei Texturen (für Katze und Schnecke) auf selbst gemachten Fotografien realer Tiere basieren. Ansonsten wurde alles mittels den Programmen GIMP (Texturen) und Blender (Modelle) selbst erstellt. Bereits bei der ersten Abgabe wurden triviale Objekte vermieden; nun gibt es mit den „Metallinseln“ und insbesondere dem detaillierten Turm im Zentrum des Levels noch komplexere Objekte als zuvor.
Außerdem „reitet“ nun die namensgebende Katze auf der Wollknäuel-Kanone der fliegenden Schnecke und ist somit stets sichtbar.

„Texture mapping“:
Objekte werden mittels des Assimp Loaders geladen, von wo aus auch Texturkoordinaten, die Normalen, jegliche Materialien sowie auch die Namen der Texturdatei ausgelesen werden. Mittels DevIL werden die Texturen dann eingelesen und in ein für OpenGL brauchbares Format umgewandelt.

„Lighting and materials“:
Vertex-Shader und Fragment-Shader ergeben im Zusammenspiel einen Blinn-Phong-Shader, wobei also diffuses Licht, spekulares Licht und – hard-coded – ambientes Licht (indem R-, G- und B-Kanal mit 0.1 multipliziert werden) berücksichtigt werden. Beleuchtung und Glanzpunkte auf der fliegenden Schnecke kommen besonders zur Geltung, wenn sie (mittels längerem Druck auf die linke oder rechte Pfeiltaste) um die eigene Achse gedreht wird. Weiters wurde jedem vorhandenen Objekt jeweils via Blender ein anderes Material zugewiesen,was sich in unterschiedlichen diffuse- und specular-Werten äußert.
Außerdem wurde für die zweite Abgabe nun auch eine Shadow Map implementiert (siehe „Effects“-Kapitel).

„View Frustum Culling“:
Den Kern unserer Implementierung des View Frustum Cullings stellt die Methode SceneObject::objektSichtbarkeitstest() dar, welche sich an dem Tutorial http://r3dux.org/2011/02/how-to-perform-manual-frustum-culling/ orientiert und einen bool-Wert zurückliefert: „true“, wenn das Objekt sichtbar ist, sich also innerhalb des Frustums befindet, „false“, wenn nicht.
Dieses Ergebnis ist aber natürlich nur dann relevant, wenn das View Frustum Culling aktuell aktiviert ist (ansonsten ändert es gar nichts). Aber wenn dem so ist (Variable „frustumCulling“ aus Level.cpp == true), dann werden nur Objekte angezeigt, welche als Ergebnis obiger Methode „true“ erhalten.
Für die Sichtbarkeitsüberprüfung werden übrigens die Bounding Spheres aus der Kollisionsabfrage übernommen (bzw. ihre Radien).

„Transparency“:
Das große Gewässer in unserem spielbaren Level ist transparent; sofern selbiger Effekt nicht manuell vom User abgeschaltet wird (siehe unten bei „Experimenting with OpenGL“). Je nach Blickwinkel (Einfallswinkel) auf das Wasser ändert sich der Transparenzfaktor, indem der Alpha-Kanal des gezeichneten Pixels verändert wird.

„Controls“:
Die Steuerung der Schnecke wurde seit der ersten Abgabe massiv erweitert:
linke/rechte Pfeiltaste: Schnecke (und Kamera) nach links/rechts um die eigene Achse drehen.
obere Pfeiltaste: Sinkflug (bewusst invertiert); die Kamera folgt der Schnecke.

untere Pfeiltaste: Die Schnecke steigt nach oben auf, gefolgt von der Kamera.
Leertaste oder Taste F: Kurz drücken, um gewöhnliche Wollknäuel-Projektile abzuschießen. Wird die Schusstaste jedoch für ca. 3 Sekunden gedrückt gehalten, so wird beim Loslassen ein goldenes Wollknäuel verschossen, welches mehr Schaden als die Standardwaffe anrichtet.
Taste A (gedrückt halten): Abbremsen
Taste S (gedrückt halten): Geschwindigkeit der Schnecke erhöhen
Anmerkung zu Boost/Bremse: Beschleunigung ist nicht unbegrenzt möglich – siehe unten („Gameplay“)
F1: Tutorial-Video einblenden
Escape: Das Spiel beenden
Weitere Tastenkommandos, welche nicht direkt mit dem Gameplay zu tun haben, sind im Kapitel „Experimenting with OpenGL“ aufgeführt.

„Gameplay“:
Siehe „Controls“: Die Schnecke kann frei umherfliegen, Wollknäuel in normaler oder aufgeladener, stärkerer Form verschießen (siehe oben) und damit Gegner bekämpfen und besiegen sowie freilich auch selbst Schaden nehmen (ihre Lebensenergie wird durch die Augen-Anzahl der Würfelanzeige auf dem schwebenden Monitor angezeigt), wenn sie ein gegnerisches Projektil, einen Gegner oder ein Hindernis bzw. die Umgebung berührt (nachdem sie erwischt wurde, ist sie aber jeweils für einen sehr kurzen Zeitraum unbesiegbar). Ebenso sind Boost und Bremse möglich (siehe oben): Die „Hasen-und-Schildkröten-Anzeige“ in der linken unteren Ecke des Bildschirms gibt dabei einerseits die aktuelle Geschwindigkeit an (Zeiger in der Mitte = Standardgeschwindigkeit, Zeiger leicht nach links geneigt = Brems-Stufe 1, Zeiger stark nach links geneigt = Brems-Stufe 2, Zeiger leicht nach rechts geneigt = Boost-Stufe 1, Zeiger stark nach rechts geneigt = Boost-Stufe 2), welche dadurch bestimmt wird, ob/wie lange der entsprechende Knopf gedrückt gehalten wird.
Andererseits zeigt sie auch, wie lange noch geboostet/gebremst werden kann bzw. wie lange es noch dauert, bis wieder ein Boost- oder Bremsmanöver möglich ist: Wird die Anzeige völlig rot, so wurde entweder die maximale Boost-/Brems-Zeit überschritten oder die Boost-/Brems-Taste losgelassen; in diesem Fall muss abgewartet werden.
Für die Kollisionsabfrage zweier Objekte wird (mittels colDet) das eine Objekt als „complex object“ und das andere als Kugel (bounding sphere) angenommen: In der Regel wird innerhalb einer Klasse, in welcher eine collision detection durchgeführt wird, das „eigene“ Objekt (= eine Instanz besagter Klasse) als complex object angesehen und das andere (mit welchem eventuell kollidiert wird) als bounding sphere. Beispiel: In der Schnecke.cpp-Klasse wird bei der Kollisionsabfrage die Schnecke selbst als complex object angesehen und ein Gegner als bounding sphere.
Wurden alle Gegner im Level besiegt, gilt das Spiel als gewonnen und ein kurzes Gratulationsvideo wird auf dem schwebenden Monitor in der linken unteren Bildschirmecke eingespielt; danach wird das Programm automatisch beendet. Analoges geschieht im Falle eines „Game Over“.

Effekte:

- Shadow Maps (with PCF) [1.5 Effektpunkte]: Schatten für sämtliche Objekte, nicht nur für die fliegenden. Dafür wurde ein eigener Shader erstellt, welcher sich darum kümmert; jener setzt sich aus ShadowMap.frag (Fragment-Shader) und ShadowMap.vert (Vertex-Shader) zusammen. Die Implementierung orientiert sich an Vortrag und Folien aus der CGUE-Vorlesung.
- Animated Textures (Video Stream) [1 Effektpunkt]: Es schwebt stets ein (als 3D-Modell modellierter) Monitor in der linken unteren Ecke des Bildschirms, welcher selbst über ein Display verfügt, auf dem Videos eingespielt werden können. Wird die Taste F1 gedrückt, so ist darauf ein Tutorial-Video zu sehen, welches die Steuerung des Spiels erklärt; wird das Spiel gewonnen, erscheint dort ein kurzes Gratulations-Video und wer verliert, sieht dort ein

„Game Over“-Video. Ansonsten läuft darauf ein Katzenfilm. Alle dieser Videos wurden selbst gedreht. Es wurde FFmpeg für die Enkodierung des Videos verwendet und die Implementierung des Vorgangs, ein Bild aus einem Video-Stream zu erhalten, orientiert sich an folgendem Tutorial: http://dranger.com/ffmpeg/tutorial01.html .

- Water (+Frasnel-Shading,Normal Mapping) [0.5 Effektpunkte] + Reflection [1 Effektpunkt]: Das große Gewässer im spielbaren Level zeigt diese Effekte schön auf: Es werden Wellen simuliert (Normal Mapping), ein Übergang zwischen „durchsichtig“ und „undurchsichtig“ ist sichtbar (Frasnel Shading) und die Umgebung spiegelt sich in der Wasseroberfläche (Reflection). Für die Abgabe vom 19.6., als der Effekt noch nicht sichtbar funktionierte, wurden etwa http://www.gamedev.net/topic/613658-view-matrix-to-reflected-view-matrix/ und http://khayyam.kaplinski.com/2011/09/reflective-water-with-glsl-part-i.html als Ressourcen herangezogen; für die aktuelle, funktionstüchtige Variante wurde insbesondere folgendes Tutorial verwendet: http://adrianboeing.blogspot.co.at/2011/02/ripple-effect-in-webgl.html . Extra für die Wasseroberfläche wurde ein eigener Shader implementiert, welcher sich aus Wasser.frag und Wasser.vert zusammensetzt. Für die Spiegelung wurde eigens die komplette Szene gespiegelt und in einer separaten Textur gespeichert.

Experimenting with OpenGL – Allgemein:

- Vertex-Buffer-Objects (VBO) und Vertex Array Objects (VAO): Alle Vertices werden in den Vertex Buffer geladen und man bindet diese mit dem Vertex Array; man muss also nur noch das Vertex Array aktivieren, um die Szene zu zeichnen.
- Buffer-Objects: FBO (Frame Buffer Object) – in besagtes FBO wird die Shadow Map gezeichnet, um sie dann als Textur verwenden zu können.
- Mip Mapping: In der vorliegenden Abgabe defaultmäßig aktiviert, um eine geladene Textur automatisch als nächstkleinere Textur vorzuberechnen, um bessere Interpolationsergebnisse zu erringen. Die Standardeinstellung ist hier „Linear“, aber das Mip-Mapping kann auch auf „Nearest Neighbour“ umgeschaltet oder ganz ausgeschaltet werden (siehe unten).
- Texture Sampling Quality: Bilineares Filtering wird defaultmäßig verwendet, es kann aber auch auf Nearest Neighbour-Interpolation umgeschalten werden (siehe unten).

Experimenting with OpenGL – Tastenkommandos:

- F1: Tutorial-Video wird auf dem Monitor-Display in der linken unteren Ecke des Bildschirms eingespielt („Help“).
- F2: Über dem Display des besagten schwebenden Monitors wird die Frame Time sowie die Anzahl der gerenderten Objekte (wichtig für Frustum Culling!) ein- bzw. ausgeblendet („Frame Time on/off“).
- F3: Der Wire Frame-Modus wird aktiviert/deaktiviert („Wire Frame on/off“).
- F4: Die Texture Sampling Quality wird umgestellt: Vom defaultmäßigen „Bilinear“ zu „Nearest Neighbour“ und wieder zurück. Auf dem Monitor-Display wird jeweils eine entsprechende Feedback-Message eingeblendet.
- F5: Die Mip Mapping Quality wird umgestellt: Vom defaultmäßigen „Linear“ zu „Off“ (als ganz ausgeschaltet) zu „Nearest Neighbour“ und dann wieder zu „Linear“ usw., wobei jeweils auf dem Monitor-Display eine entsprechende Feedback-Message eingeblendet wird.
- F8: Viewfrustum Culling ist defaultmäßig aktiviert und wird hiermit aus- bzw. wieder eingeschaltet. Der Unterschied wird durch den Unterschied in der (die mittels F2 einsehbaren) Anzahl der gerenderten Objekte veranschaulicht, welche sich mit dem Ein- und Ausschalten sichtlich ändert (Frame Time und Anzahl der gerenderten Objekte bleiben solange eingeblendet, bis sie wiederum mittels F2 deaktiviert werden; ein Druck auf F8 ändert nichts an der An- bzw. Abwesenheit dieser Anzeigen).
- F9: Die defaultmäßig aktivierte Transparenz wird aus- und wieder eingeschaltet. Der

Unterschied ist sehr stark merkbar, wenn das in dem spielbaren Level ohnehin omnipräsente Wasser betrachtet wird: Mal ist es transparent, mal völlig non-transparent, je nach Einstellung.

Features:

Von nun an werden auch Punkte gezählt! Aktuell läuft dies so ab: Wird ein Gegner besiegt, erhält man 100 Punkte (geregelt in der move()-Methode von level.cpp), wie auch auf dem Punktezähler in der linken unteren Bildschirmecke zu lesen. Wird danach ohne Gegentreffer ein weiterer Gegner besiegt, erhält man 200 Punkte, beim dritten Gegner in einer Reihe ohne selbst erhaltenen Schaden 300 etc.; es wurde also auch ein Multiplikator implementiert! Dies läuft so ab, dass ganz am Ende der punktestandAdd()-Methode in Schnecke.cpp der „punktemultiplikator“ (ist anfangs 1) inkrementiert wird; sofern zwischenzeitlich nicht die schadenNehmen()-Methode der Schnecke, welche den Multiplikator resetten würde, ausgeführt wird (sprich, wenn sie zwischenzeitlich keinen Schaden nimmt), bringt der nächste Gegner 100 * 2 Punkte und so weiter. Je weniger Fehler der Spieler macht, desto mehr Punkte erhält er also!
Ansonsten siehe oben: Im Vergleich zur letzten Abgabe wurden die Fähigkeiten der Schnecke stark erweitert (Superschuss, Boost/Bremse); auch agieren die Gegner nun auch gefährlicher und schießen nun ebenfalls. Außerdem sind neben den fliegenden „Günther“-Golems nun auch stationäre Laserkanonen zu bekämpfen.
Geplante Features wie Extrawaffen haben es leider nicht in die vorliegende Abgabe geschafft, sind aber teils bereits zu weiten Teilen im Code enthalten.

Beleuchtung und Texturierung:

Eine „point light“-Lichtquelle (Sonne) erhellt die Szene, wobei jedes Objekt texturiert wurde und jedem Objekt ein Material zugewiesen wurde (siehe oben). Es wird ein Blinn-Phong-Shader verwendet (ebenfalls s.o.).
Weiters existiert ein Extralicht (ebenfalls Point Light), um den Monitor zu erhellen, damit die darauf sichtbaren Anzeigen immer gut sichtbar sind (auch in schattigen Umgebungen).

Zusätzliche Libraries und Quellen:

- Assimp Loader - http://assimp.sourceforge.net/
- Blender - http://www.blender.org/
- colDet für die Kollisionsabfrage - http://sourceforge.net/projects/coldet/ [neu für die zweite Abgabe]
- Corel Video Studio Pro X4 Student Version für den Videoschnitt – im Lehrmittelzentrum TU erworben [neu für die zweite Abgabe]
- DevIL (incl. ILU.dll und ILUT.dll) - http://openil.sourceforge.net/
- Fotos einer Katze und einer Schnecke - selbst gemacht, teils mittels GIMP nachbearbeitet; besonderer Dank an den Kater Max und die freundliche Schnecke mit dem leider unbekannten Namen! [neu für die zweite Abgabe]
- FFmpeg für die Videos - http://www.ffmpeg.org/ [neu für die zweite Abgabe]
- GIMP zur Erstellung der Texturen - http://www.gimp.org/
- GLFW - http://www.glfw.org/
- GLEW - http://glew.sourceforge.net/
- GLM - http://glm.g-truc.net/
- Tutorials zu den Effekten und zum Frustum Culling, welche bereits oben (in den entsprechenden Sektionen) in Form von Links aufgeführt wurden.
- Noch enthalten, aber nicht mehr verwendet: libpng16.dll - http://www.libpng.org/pub/png/libpng.html