CGUE: Unseen

Gameplay

In Unseen befindet man sich in einem verschlossenen Gewölbe aus Räumen und Gängen und versucht seinen Weg hinaus zu finden. Dabei stellt man früh fest, dass man nicht alleine ist. Zur Steuerung stehen einem die Tasten W, A, S, D (Vorwärts, Linkswärts, Rückwärts, Rechtswärts) und Space (Springen) sowie Shift (Sprinten) zur Verfügung. Das “Backpaddlen”, also rückwärts gehen mittels “S” ist langsamer als jede andere Bewegung.

Es gibt 2 Level. Ziel ist es lebend einen Ausgang zu finden.

Verschlossene Türen lassen sich mittels passender Schlüssel, die gesucht werden müssen, öffnen. Oft verstecken sich diese, man muss also genau suchen. Fällt man in eine Grube so stirbt man. Sterben bedeutet am Anfang des jeweiligen Levels in dem man sich gerade befindet zu re-spawnen um es erneut zu probieren. Man hat unbegrenzt viele Chancen.

Feautures

Schatten (2 Effektpunkte)

Schatten erzeugen wir über **omnidirectional Shadow Maps mit PCF**. Einzelne PointLights (nämlich jene, die in ihrer Bezeichnung “_” am Ende haben) werden als Schatten-erzeugend markiert. Dabei gibt es eines im ersten Level und zwei im zweiten Level. Für jedes dieser Lichter wird eine CubeMap erzeugt. Unser ShadowMapShader rendert dann die Szene aus Sicht der Lichtquelle in alle sechs Richtungen. Dies geschieht pro Licht mit einem Rendering-Pass, da wir einen Geometry Shader benutzen. Die Schatten werden jeden Frame neu berechnet, da sich die Objekte bewegen könnten (z.B. Türen oder das Auge). Die erzeugten ShadowCubes werden dann an unseren Haupt-Shader (ShadowLightingShader) weitergegeben, welcher diese dann beim Rendern berücksichtigt. Zusätzlich wird hier auch noch PCF angewendet, um den Schatten zu verwischen. Darüber hinaus wird der Schatten schwächer mit fallender Distanz, da sonst die Auswirkungen auf die restliche Welt zu auffällig gewesen wären.

Wir sind im Wesentlichen nach diesem Tutorial vorgegangen: http://learnopengl.com/#!Advanced-Lighting/Shadows/Point-Shadows

Schatten im 2. Level.

Bloom (1 Effektpunkt)

Der in der Vorlesung beschriebene Ansatz wurde auf folgender Seite mit Codebeispielen schön beschrieben und entsprach daher großteils unserer Denkrichtung bei der Implementierung: http://learnopengl.com/#!Advanced-Lighting/Bloom.

Dazu gehen wir wie folgt vor: Unser ShadowLightingShader liefert zwei Bilder. Eines ist wie gewollt beleuchtet, das andere besteht ausschließlich aus den hellen Regionen des Bildes. Diese filtern wir über einen Schwellwert heraus (alles höher 0.8).

Das zuletzt genannte Bild wird weiter an unseren GaussShader weitergegeben in dem die Werte in horizontale und vertikale Richtung verwischt werden. Anschließend wird in unserem GlowAddShader das ursprünglich standardgemäß beleuchtete Bild des ersten Shaders mit dem Resultat des Gauss Shaders addiert wodurch ein Verwischeffekt der Farben der hellen Regionen erzeugt wird.

Der Bloom-Effekt ist besonders gut bei den Lichtquellen zu betrachten.

HDR (1 Effektpunkt)

Bei HDR verwenden wir **Reinhard-Tone-Mapping**. Wir sind dabei im Wesentlichen nach http://www.cs.utah.edu/~reinhard/cdrom/tonemap.pdf vorgegangen und haben eigene Anpassungen durchgeführt.

Bei Reinhard Tone Mapping muss zunächst eine durchschnittliche Helligkeit berechnet werden. Reinhard empfiehlt dafür folgende Formel:

$$\bar{L}_w = \exp\left(\frac{1}{N}\sum_{x,y}\log\big(\delta + L_w(x,y)\big)\right)$$

Ist HDR aktiviert, berechnen wir beim Erzeugen des ursprünglichen Ausgabebildes gleichzeitig den Logarithmus des aktuellen Wertes (siehe glowAdd.frag). Delta ist nur dazu da, dass das Ergebnis des Logarithmus definiert bleibt und ist daher bei uns 0.00000001. Der HDRShader (hdrShader.vert / hdrShader.frag) erhält dann das Originalbild und das Logarithmus-Bild. Der Vertex-Shader holt sich mittels Mipmaps den Durchschnittswert des Log-Bildes und exponenziert es, um auf $\bar{L}_w$ zu kommen. Der Fragment-Shader berechnet daraus den neuen Farb-Wert:

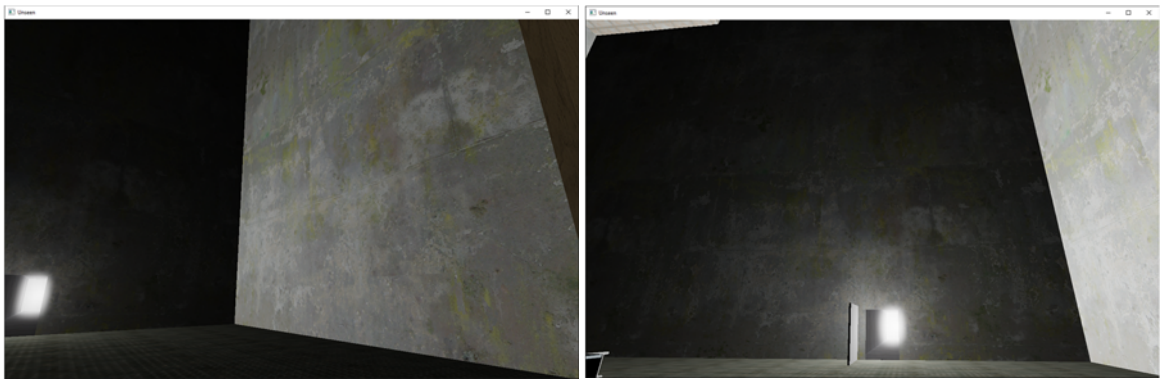
$$L(x,y) = \frac{a}{\bar{L}_w} L_w(x,y)$$

$$L_d(x,y) = \frac{L(x,y)}{1 + L(x,y)}$$

Als a wurde 0.18 gewählt.

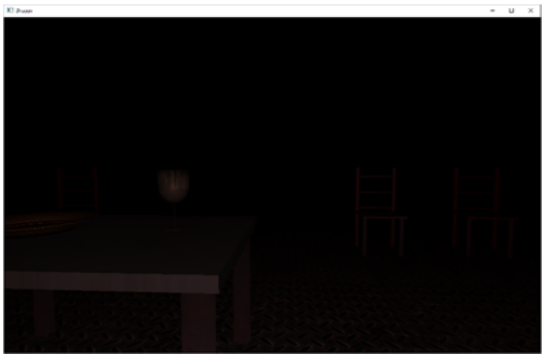
Wir haben noch eine zusätzliche Verbesserung eingebaut: Schaute man zuerst in einer helle und dann auf eine sehr dunkle Szene, wechselte die Intensität zu plötzlich und unnatürlich. Daher haben wir dafür gesorgt, dass es einen schönen langsamen Übergang gibt. Dies haben wir gemacht, in dem wir beim Erzeugen des Log-Bildes das zuletzt generierte Log-Bild und deltaT mitgeben (siehe glowAdd.frag / Renderer.cpp). Überschreitet die euklidische Distanz des Log-Rgb-Vektors an einem Pixel deltaT*8, so wird der Unterschied auf eine Distanz von deltaT*8 hinunterskaliert. Das ermöglicht, dass der Durchschnitt dann ebenfalls nicht so drastisch steigen kann. Schaut man nun also in eine helle Region und dann in eine dunkle Region, passt sich die Helligkeit des Bildes nicht sofort an, sondern geht langsam über.

Es hat sich herausgestellt, dass HDR während dem Großteil des Spiels eher stört und unpassend ist. Daher haben wir es nur am Ende des Spieles eingebaut, wo es recht gut passt. Öffnet man die letzte Tür im Kirchen-Raum, wird HDR aktiviert und greift sogar aktiv ins Gameplay ein, da man den Text am Ende erst lesen kann, wenn man nah genug an der Wand ist, so dass die hellen Werte auf etwas dunkleres gemappt werden.



Unterschiede in der Helligkeit je nach Blickrichtung am Ende des Spieles.

Dunkelheits-Effekt



Gegen Ende des Spieles gibt es eine Szene, in welcher die Lichter ausgehen und man nur mehr die Objekte um sich herum in einem schwachen, rötlichen Licht sieht. Dafür wurde ein eigener Shader (darknessShader.vert / .frag) geschrieben, welcher eine Abwandlung vom normalen Beleuchtungs-Shader ist, nur dass er nur von einem einzigen PointLight mit fixer Intensität ausgeht, welches sich direkt beim Spieler befindet.

Model-Loading

Sämtliche Modelle wurden von uns selbst in Blender erstellt und als Collada-DAE-Dateien gespeichert. Diese lesen wir mittels Assimp (http://www.assimp.org/) ein (siehe Klasse ModelLoader.cpp). Dabei wurde im Wesentlichen nach diesem Tutorial vorgegangen: http://learnopengl.com/#!Model-Loading/Assimp

Unser Modell speichern wir in einer hierarchischen Baumstruktur. Die Objekte mit ihren Vertex-Informationen sind als Mesh-Objekte gespeichert. Diese gehören jeweils zu einem Model-Objekt, welches wieder in einem anderen Model-Objekt enthalten sein kann. Lichter hängen ebenfalls an einem Model-Objekt. Dies hat den Vorteil, dass wir eine Transformation auf ein Model-Objekt automatisch auf alle Kind-Objekte übertragen können. Außerdem haben Lichter bei uns nur auf dem Teilbaum des Szenengraphen Auswirkungen, der bei dem Objekt, in welchem sie sich befinden, beginnt. So ist es möglich, dass ein Licht nur innerhalb eines Raumes für die Beleuchtung berücksichtigt wird.

Animationen

Mehrere Objekte bewegen sich im Spiel, vor allem Türen und das Auge. Eine hierarchische Animation ist bei der Uhr im 2. Level zu betrachten.

Das Pendel schwingt hin und her, während sich auf dem Pendel selbst ein Objekt kreisförmig um den Mittelpunkt des Pendels bewegt.

View-Frustum-Culling

Das View-Frustum-Culling wurde nach diesem Tutorial implementiert http://www.lighthouse3d.com/tutorials/view-frustum-culling/. Konkret haben wir den dort vorgeschlagenen “Geometric Approach” angewendet bei der das View-Frustum aus sechs Flächen besteht deren Oberflächennormalen jeweils in das Innere des Frustums zeigen. Das Vorzeichen der Koordinaten eines beliebigen Punktes im Raum verrät uns auf welcher Seite der Flächen es sich befindet, folglich ist die Unterscheidung der Punkte innerhalb und außerhalb des Frustums sehr leicht. Natürlich liegt es nun nahe diesen Algorithmus für jeden Vertex für alle Meshes anzuwenden. Der Naive Ansatz wäre für jeden Mesh die Vertices zu durchlaufen und zu prüfen ob es sich innerhalb des Frustums befindet oder nicht. Zwecks Effizienz bietet es sich an das Mesh zu zeichnen sofern zumindest ein Vertex innerhalb des Bereichs liegt, doch selbst dann ist die Implementierung sehr ineffizient denn im schlimmsten Fall könnten beim Durchlaufen der Vertices eines Meshes all jene Vertices zuerst geprüft werden, die außerhalb des Frustums liegen, wodurch der Algorithmus viel Zeit verliert bevor entschieden wird, dass dann doch gezeichnet wird.

Daher haben wir uns dazu entschieden sogenannte Bounding-Spheres um unsere Meshes zu legen. Wir berechnen also für jeden Mesh eine Kugel die sie einschließt und prüfen nun nur mehr ob die Kugel sich mit dem Frustum schneidet. Natürlich ist die gewählte geometrische Form ungünstig für flache und große Meshes und gibt großzügig viel Platz weg, aber die Performanzsteigerung ist beachtlich und das Culling dennoch hinreichend effektiv.

Zur zusätzlichen Effizienzsteigerung haben wir im 2. Level außerdem eingebaut, dass der Kirchenraum nicht gerendert wird, solange man sich nicht im Vorraum oder im Kirchenraum selber befindet. Genauso wird der erste Teil des Levels nicht mehr gerendert, sobald die Kirche betreten wurde (dieses „Culling“ scheint im Culling-Zähler nicht auf). Darüber hinaus wurde auch Backface Culling aktiviert (sichtbar über den Wireframe-Mode).

Transparenz

Es gibt einige transparente Objekte im Spiel, so z.B. die Weingläser im 2. Level oder die Kastentür der Uhr (siehe Screenshot bei “Animationen”). Objekte, welche ein transparentes Material besitzen, werden beim Zeichnen erst nach den anderen Objekten gezeichnet (siehe Scene.cpp, Methode draw). Schatten fallen durch die transparenten Flächen durch, da der ShadowMapShader Objekte mit einer Opacity kleiner 1 nicht zeichnet (siehe ebenfalls die Uhr). Auch die Glühbirnen im Spiel sind eigentlich halbtransparente Objekte, das Licht, welches sich in ihnen befindet, geht trotzdem durch sie durch.

F-Tasten

Alle F-Tasten geben ihre Funktion und den Status ihrer Funktion bei Betätigung der jeweiligen Tasten über die Konsole bekannt. Sie sind wie folgt belegt:

F1: Hilfe-Ausgabe

Gibt die Tastaturbelegung für das Gameplay in der Konsole aus.

F2: Frame-Time-Ausgabe

Falls aktiv: Gibt pro Sekunde die Frametime in ms und die Frames per Second in der Konsole aus.

F3: Wireframe-Mode (off / material / green)

Aktiviert den Wireframe-Shader. Modus durch wiederholtes Drücken einstellbar.

F4: Texture-Sampling-Quality-Wechsel (bilinear / nearest neighbour)

Wechselt die Textur-Sampling-Quality. Modus durch wiederholtes Drücken einstellbar.

F5: Mipmapping-Quality-Wechsel (off / nearest neighbour / linear)

Wechselt die Mipmapping-Quality. Modus durch wiederholtes Drücken einstellbar.

F6: Glow-Mode-Stärke (strong / medium / weak)

Variiert die Stärke des Glow-Effektes (niedrigerer Effekt -> mehr FPS)

F7: View-Frustum-Culling-Counter-Ausgabe

Falls aktiv: Gibt pro Sekunde die Anzahl durch Frustum-Culling entfernter und die Anzahl bestehender Meshes in der Konsole aus.

F8: View-Frustum-Culling (off / on)

Aktiviert/deaktiviert das View-Frustum-Culling.

F9: Transparency-Mode (off /on)

Aktiviert/deaktiviert die Unterstützung transparenter Farben.

F10: No-Clip-Mode (off / on)

Der No-Clip-Mode ermöglicht es, durch Wände zu fliegen.

Beleuchtung und Texturierung

Die meisten Objekte besitzen Diffus-Texturen. Einzelne Objekte (z.B. Gitterstäbe, transparente Objekte oder das Uhrpendel) sind texturlos und haben einfach nur eine Farbe.

Die Beleuchtung läuft hauptsächlich über PointLights, welche jeweils einzelnen Räumen zugeteilt sind, so dass sie sich nicht auf die anderen Räume auswirken. Das Auge wird als einziges Objekt von allen Lights beeinflusst. Zur Berechnungen werden aber immer nur die nähersten berücksichtigt.

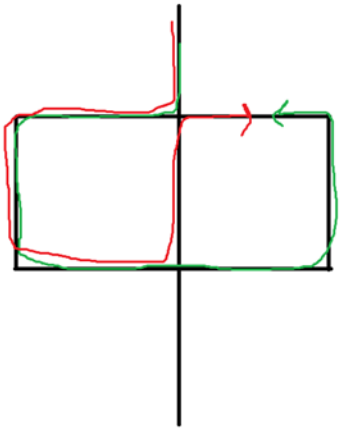
Die Lichtquellen stellen wir darüber hinaus mit Glühbirnen bzw. Deckenlampen dar. Bei den Glühbirnen befindet sich das PointLight innerhalb der Birne, bei den Deckenlampen etwas darunter. Die Lampen werden zusätzlich noch durch DirectionalLights beleuchtet, damit es so ausschaut, als würden sie selber leuchten.

Künstliche Intelligenz

Für das Auge wurde eine Künstliche Intelligenz geschrieben (ordner “ai”), welche im Wesentlichen darauf basiert, dass die Welt quasi mittels eines Graphen dargestellt wird. Dabei gibt es einerseits Kanten, auf welchen sich das Auge nur vor und zurückbewegen kann, und andere Kanten, in welchen sich das Auge innerhalb eines gewissen Raums frei herum bewegen kann.

Damit das Auge sich zum Spieler bewegt, wird regelmäßig den kürzesten Pfad zum Spieler mittels Dijkstra-Algorithmus berechnet.

Zusätzlich sind Sonderbehandlungen in Form von Umleitungen möglich. Ein Knoten im Welt-Graphen kann eine bevorzugte Kante vorgeben. Im 1. Level gibt es daher folgende Sonderbehandlung:



Geht der Spieler den in der Abbildung grün markierten Weg und das Auge ist hinter ihm, nimmt es (meistens) den rot markierten Weg. Der Spieler sollte denken, dass das Auge direkt hinter ihm ist, und ist dann überrascht, wenn es die Abkürzung durch den mittleren Gang genommen hat und plötzlich vor ihm ist.

Collision-Detection

Zu Bewältigung des Collision-Detection-Problems haben wir uns dazu entschieden eine Physics-Engine zu benutzen. Unsere Wahl fiel auf “Bullet Physics” (http://bulletphysics.org/). Es stellte sich heraus, dass die Engine schlecht dokumentiert ist wodurch es zu einigen Komplikationen bei der Implementierung kam. Letzten Endes gelang es uns schrittweise die Physikwelt zu implementieren und Elemente wie Schwerkraft, Restitution und Reibung zu aktivieren und so auch die Kollisionserkennung mittels sogenannten “Rigid Bodies” die wir für ausgewählte Meshes erzeugen. Dazu nehmen wir während des Ladevorgangs der Meshes die am weitesten entfernten Vertices eines Objekts und bilden so eine kollisionsempfindliche Bounding Box (oder Sphere) welche das komplette Objekt umschließt. Es wird nur auf Kollisionen mit dem Spieler geprüft. Bullet wird ebenfalls verwendet, um die Interaktion mit Schlüsseln und Türen zu registrieren.

Sound

Spielgeräusche sind ein wesentlicher Inhalt bei Computerspielen und wertvolle Assets wenn es um Horrorspiele geht. Zur Einbindung hat sich die im Wiki vorgeschlagene Sound Engine “FMOD” (http://www.fmod.org/) angeboten. Jedoch mussten wir auch hier feststellen, dass die Dokumentation in vielerlei Hinsicht ausgesprochen mangelhaft ist. Trotz initaler Hürden gelang die Implementierung. Wir unterscheiden zwischen Standard Stereo Sounds und 3D Sounds. Die Engine benötigt ähnlich wie die Physikengine Informationen der Kamera (Position, Blickrichtung, etc.) und eine Position des zu platzierenden 3D-Sounds und vollführt anhand dessen die korrekte Ausgabe des Sounds auf dem aktiven Treiber. Um das ganze angenehmer zu gestallten haben wir das FMOD-System als Singleton implementiert und eine Sound-Klasse dessen Instanzen einzelne Geräusche oder Musik repräsentieren. Die Engine unterstützt alle gängigen und auch weniger gängigen Musik-Dateienformate. Es wird jedoch von der Engine empfohlen auf Rohdaten zurückzugreifen da eine präzisere Abtastung so möglich sei, weswegen wir überwiegend “.WAV”-Files benutzen.

Ab und zu kann es auf manchen Rechnern bei manchen Lautsprechern manchmal passieren, dass Sounds plötzlich knacksende Störgeräusche enthalten oder mehrfach abgespielt werden. Auch sorgt FMOD gelegentlich für eine kurze Verzögerung, wenn man den silbernen Schlüssel im zweiten Level aufhebt. Dieses Problem konnten wir leider nicht mehr beheben.

Nahezu alle Sound-Files sind mittels entsprechender Musikproduktionssoftware selbst kreiert, einige wenige Elemente haben wir uns aus frei verfügbaren Quellen genommen (https://www.freesound.org).

Walkthrough [Spoiler-Warnung!]

Level 1

Man springt über den Abgrund und geht bei der Kreuzung links. In diesem Gang findet man in einer Bibel einen Schlüssel, welchen man aufheben kann. Danach geht man den Hauptgang weiter zum Raum mit dem Bett. Da man den Schlüssel gefunden hat, kann man hier nun die Türe öffnen. Geht man in den Gang rein, kommt das Auge entgegen und man sollte wegrennen. Rennt man den Gang zurück, muss man bei der Kreuzung links oder rechts abbiegen. Kommt man dann zum Loch über den Abgrund, sollte man nicht grade aus drüber springen, sondern runter springen und von unten wieder zum Bett-Raum zurück. Rennt man stattdessen weiter, wird das Auge möglicherweise plötzlich von vorne kommen. Des Weiteren muss man darauf aufpassen, dass man nicht zu schnell wegrennt, da man sonst bei der zweiten Kreuzung sein könnte, wenn das Auge grade mal bei der 1. ist, dann würde es auch direkt auf den Spieler zukommen. Im Bett-Raum rennt man einfach geradeaus in den dunklen Gang hinein.

Level 2

Man schafft es in der Beginn-Sequenz das Auge abzuschütteln. In dem Raum geht man links zur Uhr, öffnet durch Klick den Kasten und hebt den Schlüssel darin auf. Mit diesem lässt sich dann das Gitter öffnen. Draußen geht man gleich rechts und dann gleich links (nicht rechts, sonst wird das Auge getriggert!). Am Ende des Ganges findet man einen weiteren Schlüssel auf einem Tisch. Diesen hebt man auf und das Auge wird hinter einem getriggert. Man dreht sich um, rennt bei der ersten Kreuzung rechts, dann gradeaus und dann den Gang solange weiter, bis man zu einer offenen Tür rechts kommt. Dort springt man über den Abgrund, öffnet die Tür und rennt in den Raum hinein. Die Tür geht automatisch hinter einem zu.

In dieser Halle findet man auf dem Tisch beim Kreuz einen Schlüssel. Hebt man diesen auf, wird alles dunkel. Steht man vor dem Kreuz, dreht man sich um 90° nach link und geht einfach geradeaus. Man kommt direkt zu einer neu aufgetauchten Türe, welche sich mit dem Schlüssel öffnen lässt.

Man geht dann in diesen Gang hinein, erblickt am Ende des Ganges die Schrift an der Wand, dreht sich um und das Auge steht vor einem. Das Spiel ist vorbei.