

# Projekt Dokumentation

**Projekt: Escape 3D**

Themen:

Effekte

## **SpotLight**

Der Spieler verfügt über eine Taschenlampe. Dazu gibt es eine Klasse SpotLight, die bei jedem Update die Position und Kamerablickrichtung mit den Daten der Kamera aktualisiert. Der Beleuchtungswinkel ist im simple Shader fix implementiert.

<http://www.mbsoftworks.sk/index.php?page=tutorials&series=1&tutorial=20>

## **Animated Textures**

Für die Animated Textures besitzt Material.cpp eine Vektor von Texturen. Jede Textur stellt einen Frame des Bildes dar. In der Draw Methode vom Material wird entschieden welche Textur zum Shader übergeben werden soll. Alle Frames werden beim initialisieren des Spiels geladen, Voraussetzung dafür ist das im Blender die Textur mit foldername.texturname.jpg abgespeichert wird. Dann sucht der Image Loader alle Bilder die im Folder "foldername" sind und mit einer fortlaufenden Nummer gekennzeichnet sind.

Die Animated Textures kann man auf der Luftstation erkennen. Man gehe bis zum Ende des Korridores, wo sich ein großes Objekt mit zwei länglichen Teilen im Kreis bewegt. Wenn man unten von der Seite das Objekt ansieht, erkennt man auf jeder Seite 2 Bildschirme, welche leicht die Farbe ändern. Aufwendigere Animationen funktionieren zwar genauso gut, und würde man hier vielleicht besser erkennen. Aber um das Spiel Design nicht zu stören bleibt dies eine nur dezente Animation der Farben.

Idee über Implementation von hier bekommen:

[http://www.swiftless.com/tutorials/opengl/texture\\_animation.html](http://www.swiftless.com/tutorials/opengl/texture_animation.html)

## **Shadowmapping**

Konnten wir leider nicht fertigstellen. Der Status Quo bezüglich des Shadowmapping befindet sich im Branch "shadow". Wird aber noch bis zum 2. Spielevent nachgeliefert.

<http://www.opengl-tutorial.org/intermediate-tutorials/tutorial-16-shadow-mapping/>

## **Normal Mapping**

Normal Maps kann man, unter anderem, auf dem Corridor sehen und auf den Bar Obstacles (Längliche Hindernisse die sich in der ersten Szene auf dem Corridor Part mit dem Fenstern befinden.)

Implementation: Zur Erzeugung der NormalMaps (Texturen) wurde die Textur mit Hilfe von Photoshop in Schwarz Weiß umgewandelt. In Blender wurde diese schwarzweiß Textur mit dem Displacement Modifier auf ein Plane Object angewandt. Mit dem daraus resultierenden Mesh, wurde mittels Blender schließlich die Normalmap generiert. (für mehr infos dazu siehe: [http://cgcookie.com/blender/2010/06/30/normal\\_maps\\_blender\\_2\\_5/](http://cgcookie.com/blender/2010/06/30/normal_maps_blender_2_5/))

In c++:

Beim Einlesen jeder Textur wird zusätzlich nach einer Normalmap im Ordner normalmaps gesucht. Falls keine Normalmap gefunden wurde, wird eine Flache Standard Normal Map verwendet.

Im Objectloader werden beim Einlesen der Vertices auch zusätzlich die Tangenten und Bitangenten berechnet. Am Schluss werden noch alle gerade eingelesenen/erstellten Daten bei der Initbuffer Methode der TexturedMesh Klasse dem Shader übergeben.

Shader:

Im Vertex Shader wird eine Matrix erstellt, welche mithilfe der Tangente, Bitangente und Normale, Vektoren von dem Tangentenspace in den ViewSpace transformiert.

Diese Matrix wird dann auch im Fragmentshader nach dem einlesen der Normalmap verwendet. (nachdem der gerade eingelesene Vektor der Normalmap auf den Bereich -1 - 1 abgebildet wurde)

Implementationshilfe von folgenden Seiten:

<http://www.opengl-tutorial.org/intermediate-tutorials/tutorial-13-normal-mapping/>

[http://en.wikibooks.org/wiki/GLSL\\_Programming/GLUT/Lighting\\_of\\_Bumpy\\_Surfaces](http://en.wikibooks.org/wiki/GLSL_Programming/GLUT/Lighting_of_Bumpy_Surfaces)

## Illumination

Der gesamte Korridor wird von der Sonne, welche sich links im Bild befindet, beleuchtet. Als einzig zweite Lichtquelle dient die Taschenlampe, welche automatisch in die Richtung des Spielers leuchtet.

## Additional Librarys

ImageLoading: resIL Image Loader (<http://sourceforge.net/projects/resil/>)

Physics: Bullet Physics (<http://bulletphysics.org/wordpress/>)

sound: irrKlang (<http://www.ambiera.com/irrklang/>)

ObjectLoader: assimp (<http://assimp.sourceforge.net>)

glut

glew

glfw

Maths: glm (<http://glm.g-truc.net/0.9.5/index.html>)

## Used Tools

- 3D-Object: Blender
- Video Capturing: fraps
- Textur Erstellung: Photoshop
- Free Textures: [www.cgtextures.com](http://www.cgtextures.com)

## Features of the Game

- Spieler kann die Schwerkraft ändern
- Spieler muss auf die Energie der Taschenlampe achten und kann diese wieder aufladen
- Spieler muss auf seinen Sauerstofflevel achten und ggf. aufladen

## Brief Description of Submission 2 Implementations

### GamePlay

Der Spieler befindet sich zuerst in einem Durchgang zu einem größeren Corridor Abschnitt. Zuerst muss er zur Türe hingehen diese mittels Leertaste öffnen und nach wenigen Sekunden kann er diese passieren. Nun eröffnet sich für ihn ein großer Corridor. Begrüßt wird man mit einem Großen Ventilator. Da dies die Wartungskorridore der Raumstation sind, ist so etwas keine Überraschung. In der Mitte des Korridores verläuft von einem Ende zum anderen eine Luftschleife wo die restliche Luft verteilt wird.

Wenn man sich einer Korridor Wand annähert kann man mittels "Tab" Taste die Wand hinauf laufen. Dies funktioniert da der Spieler magnetische Stiefel anhat, welche, selbst wenn man von einem Hindernis runterspringt zur nächstgelegenen Korridor Wand haften bleiben.

Vorsicht vor dem Berühren des großen Ventilators, er stellt eine Gefahrenquelle dar, die dem Spieler Schaden zufügen kann, indem er seinen Anzug beschädigt und dadurch mehr Luft verliert.

Die Taschenlampe ist auch ein wichtiger Aspekt des Spieles. Die Raumstation besitzt nur noch Strom für die allernotwendigsten Funktionen. Gerade deine Taschenlampe bleibt dir um dich durch den Korridor zu navigieren. Aber dieses Licht geht auch schnell aus. falls die Batterien (Gelber Balken ausgehen sollten) kann man aber die Taschenlampe für kurze Zeit einschalten, da diese aus Sicherheitsgründen mittels Bewegung wieder aufladbar ist (zumindest für eine halbe Sekunde). Im Raum kann es auch eine Luft Station geben, welche deinen Raumanzug schnell wieder mit Luft versorgen kann, und deine Taschenlampe auflädt. Diese benutzt man auch mittels Leertaste. sobald man nah genug dort ist, sollte sich schon bald die Batterie und die Luft Anzeige erhöhen. Nachdem man alle Hindernisse überwunden hat kann man hoffentlich weiter zum nächsten Korridor gehen und schlussendlich das halb zerstörte Raumschiff verlassen.

## Effekte

siehe Effekte

### Animated Objects

Dieses kann man bei den großen Ventilator, der sich am Ende des ersten Levels befindet, Hier dreht sich der das ganze objekt um einen Lüftungsschacht und um dieses objekt selbst dreht sich ein eigener torus.

Hierfür wird bei jedem Update die aktuelle ModelMatrix mit der ModelMatrix des eventuell vorhanden Parentobjekts multipliziert.

### Frustum Culling

View Frustum Culling funktioniert bei uns auf einer Ebene. Das heißt es wird nur überprüft ob ein Objekt vor oder hinter der Kamera liegt... Da sich unser spielfeld auf einem Corridor befindet, und beim Start des spiels selbst mittels 6 Plane View Frustum (so wie hier beschrieben: <http://www.lighthouse3d.com/tutorials/view-frustum-culling/>) alle Elemente sichtbar sein müssen (sonst sieht man zb hinten beim Korridor den Weltall), haben wir uns entschieden nur ein 1 Plane View Frustum zu implementieren. Das hat zur Folge das am Anfang des Spiels keine Objekte auf Grund des View Frustum Culling entfernt werden können. Jedoch, werden mehr und mehr Elemente entfernt, desto weiter man sich im Raum fortbewegt. Um sehen zu können, wieviele Elemente jetzt wirklich gerendert werden, muss man noch zusätzlich die F7 Taste drücken (verhindert Spam im Konsolenfenster).

### Transparency

Kann man zum Beispiel bei den Fenstern des Korridors sehen. Diese kann man auch mittels F9 ein und ausschalten.

### Experimenting with OpenGL

wir verwenden vaos um die Vertex Daten dem Server zu überreichen.

Mit Frame Buffer Objects haben wir herumexperimentiert, jedoch ist uns dies misslungen. siehe Punkt Effekte->Shadowmapping

Die MidMap Qualität sowie die Textur Qualität kann mittels vorgesehenen Tasten ein und ausgeschaltet werden.

## Other special Features in your Game

Spieler kann verschiedene Einstellungen vor dem Spiel treffen.

zb größe des Bildschirms, Fullscreen, Framerate, ob Musik an/aus sein soll, ob die Soundeffekte eingeschaltet sind.

Zusätzlich kann man auch einen "debugMode" einschalten, hier kann man nicht sterben und die TexturQualität beim Einlesen ist deutlich verringert (ermöglicht schnellere Ladezeit).

Im Spiel selbst kann man mittels F6 kann man zwischen einer Free Moving Kamera und der Spieler Kamera wechseln.

Die Restlichen F Tasten wurden so belegt wie von der Submission vorgegeben.

Mittels Enter Taste kann man, falls vorhanden zum nächsten Level wechseln.

Die Taste '1' Toggled den Physics Debug View, welcher jedoch nicht so funktioniert wie er sollte.

Mit der Taste '3' kann man den Spieler schneller bewegen (was jedoch nicht Ziel des Spiels ist, sondern nur zu Debug Zwecken genutzt wird).

Um nicht unnötig viele Texturen zu laden, verwenden wir eine TexturCollection, diese speichert sich einen wiederkehrenden Key (TexturName) falls dieser Key noch nicht vorhanden ist, dann wird die Textur erst geladen. ansonsten wird einfach ein Pointer zu der bereits geladenen Textur übergeben. Dieses Konzept bietet nicht nur schnellere Ladezeiten sondern auch einen effizienteren Speicherverbrauch. (Das Konzept der "Collections" wird auch nochmals beim speichern und Laden der Physics Elemente eingesetzt ([PhysicCollection::setPhysics](#) und [getPhysics](#)))

## Mesh Loading

Die Mesh Objekte wurden eigens per Blender erstellt. Im Blender wurden sowohl die UV Daten zu den entsprechenden Vertices gemapped, als auch die Objekte (grob) richtig angeordnet (Sodass nur noch eine grobe Verschiebung und keine Feinjustierung innerhalb der Applikation stattfinden muss)

Zum Export der Blender Daten, wurde das .dae Format gewählt. Die exportierten Dateien können mitsamt deren Texturen in den Unterordnern des Ordner Meshes gefunden werden).

Das Laden dieser Daten wurde mit der Externen Library Assimp implementiert. Die Genaue Implementation kann in der Klasse "game/sceneobjects/helper/ObjectLoader.h" gefunden werden.

Derzeit wird nur ein Mesh Objekt per Objectloader geladen. Falls dieses Mesh Objekt aus mehreren Meshes (bzw. Nodes) besteht. wird es hier richtig transformiert und gesammelt als nur ein Mesh Objekt ausgegeben.

Nachdem die Mesh Daten (vertices, indices, uv, etc) geladen wurden, werden vorhandene Texturen geladen. Derzeit wird jeweils nur die erste Textur geladen und dem MeshObjekt übergeben.

## Textures

Texturen werden wie im Bereich "Mesh Loading" bereits erwähnt, zu allererst in der Klasse ObjectLoader geladen. Assimp ermittelt den Namen der ersten Textur und übergibt diesen Wert der TextureCollection (game/sceneobjects/TextureCollection.h) Klasse. Diese ist eine Statische Klasse, welche eine Map aller Texturen besitzt. Der Key der Map ist der Name der Textur und das Value ist die Texture Klasse wo dann endgültig die Textur eingelesen wird.

Falls die Textur bereits existiert wird sie kein zweites mal von der Datei geladen. Es wird nur der Pointer zur bereits geladen File übergeben.

In der Klasse Texture findet das eigentliche Image Loading statt. Die Bilder werden mit der Externen Library ResIL (weiterentwicklung der DevIL Library) geladen und in der Methode setTexture() der Grafikkarte übergeben.

Die Texturen werden dann immer bei der Draw Methode des SceneObjects gebunden.

## Scenes

Die Scenes Klasse kümmert sich um das Laden aller wichtiger Objekte. Ein wichtiger Bestandteil der Scenes Klasse ist das Laden der Mesh Daten. Jedes .dae Objekt das wir im Nachhinein brauchen werden (bzw brauchen könnten) wird anfangs hier geladen. Weiters werden hier auch die Shader geladen. Schließlich werden Shader und Mesh Daten zu einem SceneObject zusammengebaut. Die Klasse kümmert sich auch um das Löschen der Mesh Objekte und Shader. SimpleArchitectures wie zum Beispiel der Corridor (welcher in so gut wie jeden Scene gebraucht werden könnte) werden zwar hier geladen, um das Löschen kümmert sich jedoch das SceneObject der scene selbst.

Eine Scene soll im späteren Verlauf dann die verschiedenen Levels darstellen. Ein Level wird im Grunde immer einen Weg (von Eingangstür bis Ausgangstür) besitzen.

Alle SceneObjects die in der Scene vorkommen, werden in als Kinder in das SceneObjects mit dem Namen scene übergeben. (dies geschieht im Grunde in der Methode initScene()).

## SceneObjects

Ein SceneObject ist die Klasse wo das endgültige Objekt definiert ist. Es besitzt eine Objektmatrix, welche die grundlegende Orientierung im Raum beinhaltet und einer ModelMatrix, welche für die endgültige Position im Raum verantwortlich ist (eigene Transformation + Transformation aller Parents)

In der Methode initBuffer() werden die verschiedenen (Mesh) Daten den VBO und VAO übergeben.

SceneObjects besitzen zusätzlich zwei wichtige Methoden: die Draw und die Update Methode. Diese Methode werden von verschiedenen Klassen von der CoreEngine bis hin zu den einzelnen SceneObjects weiter propagiert. Zuerst wird nur die draw bzw update Methode des "scene" SceneObjects aufgerufen, welche dann die gleiche Methode aller Kinder aufruft (usw.). Zu guter Letzt können SceneObjects auch noch bestimmte behaviours besitzen (siehe Abschnitt: Object Behaviour Classes), welche sich schlussendlich in der Methode update bemerkbar machen.

## Object Behaviour Classes

die Klassen "game/sceneobjects/helper/ObjectChanger.h" und "game/sceneobjects/helper/ObjectChangerFunction.h" kümmern sich um das Verhalten bestimmter Objekte zu bestimmen. Die Klasse ObjectChanger kann mehrere ObjectChanger Functions besitzen und kümmert sich darum das gewisse Aspekte korrekt ausgeführt werden (zb repeatable interaction [siehe später])

Die ObjectChangerFunction beinhaltet nur jeweils eine Veränderungsfunktion. Diese Funktion besitzt verschiedene Attribute. Zum Beispiel kann man angeben das sich eine Funktion in einer Endlosschleife wiederholt oder ein bestimmtes Ende besitzt. Weiters lässt sich auch definieren, ob eine Funktion erst ausgeführt wird, wenn die Interaktionstaste ("Leertaste") gedrückt wird. Nachdem eine Interaktion die mittels Leertaste ausgeführt wurde beendet wurde, kann man auch definieren, das diese Methode repeatable ist. Das bedeutet das wenn man erneut die Leertaste betätigt die gleiche Transformation für den gleichen Zeitraum ausgeführt wird.

Weiters kann man auch noch definieren wie lange eine Funktion andauert und wie viele Einheiten diese Funktion das Objekt bewegt.

Weitere Funktionen kann man in der ObjectChangerFunction mittels Statischer Methoden, nach dem Muster, der Type Definition "ChangerFunction" spezifizieren.

## InteractionManager (Controls)

Die Spezifikation der Spiel Steuerung befindet sich in der Klasse "engine/InteractionManager.h". Die Methode keyboardInput() wird an glfw als keyboard Input Callback function übergeben (glfwSetKeyCallback). WASD bewegt die Kamera wie gewohnt in dem 3D Raum. Mit dem Tabulator kann der User die Gravitation ändern.

Die Leertaste wird dazu verwendet um mit gewissen Objekten zu interagieren. Die Escape Taste dient dazu das Spiel zu beenden. Um eine Ruckelfreie und Reibungslose bewegung im Raum zu gewährleisten wird die nur der erste press dieser Taste registriert und die Funktion forward in einen vektor geschrieben. Anschließend bei der Update methode des InteractionsManagers wird jede Funktion dieses Vektors einmal mit dem derzeitigen deltaT aufgerufen. Sobald die Taste wieder losgelassen wird, wird auch die Funktion wieder aus dem Vektor entfernt und die Bewegung gestoppt.

## Light

Momentan gibts ein Pointlight. Die Positionen sowie die Intensität der Lichter werden vorläufig in der Scenes.cpp bei jedem draw gesetzt und dem Shader übermittelt. Die Berechnungen für das Licht werden im Fragmentshader vorgenommen.

Hierzu wird für jeden Pixel die Normale berechnet sowie dessen Position in Weltkoordinaten. Anschließend wird ein Vektor, der vom Pixel zur Lichtquelle zeigt, berechnet. Mit diesen drei Werten kann dann die Helligkeit des Pixels berechnet.

## Camera

Die Kamera ist in der der Datei Camera.cpp definiert und enthält alle wesentliche Eigenschaften, wie Position, Ausrichtung, Farplane und Nearplane. Bewegungen werden vom

InteractionManager über die Methoden goForward, goBackwards, goLeft and go Right angenommen. Für die Blickrichtung ist die Funktion offsetOrientation zuständig. In dieser Methode wird auch provisorisch die Bewegungsfreiheit des Spieler eingeschränkt, so dass es so aussieht, als würde man auf der einen Plane gehen können und nicht herum fliegen. Über die methode matrix() bekommt der User eine Matrize aus eine Matrizenmultiplikation von der ViewMatrix und der ProjectionsMatrix.