Dokumentation - Parallels

UPDATE for Final Presentation:

- there now is a key (F3) that toggles the usage of specular (roughness map)
- a total of 8 lights have been added to the scene so the lighting has been improved
- a few new Objects have been added to the scene with different materials

Libraries:

Im Projekt werden zwei zusätzliche Libraries verwendet:

- ASSIMP: zum Laden der Objekte (https://github.com/assimp/assimp)
- Physx: für einfache Physik und Collision Handling

Important Controlls:

WASD + Space - Movement
F8 - toggle View Frustum culling

Features of the Game:

Das Spiel hat keine wirklichen zusatz Features, da das Gameplay eigentlich die Aufgabe meines Partners war, der mich kurz vor der Abgabe im Stich gelassen hat.
Es ergab sich also, dass das Spiel zu einem 3d Jump and Run wurde.

3D Geometry:

Das importieren von Geometrie passiert in den Klassen Model und Mesh, wobei ein Model Objekt nach dem Import alle Meshes enthält, die sich im importierten File befinden. Im Mesh werden alle Informationen, die das Mesh selbst betreffen gespeichert, wie zb. die Vertices, aber auch die Information zum Material und die Id der Textur (falls zutreffend). Wir nutzen objs.

Win/Lose Condition:

Man verliert, wenn man die Lava am Boden der Kammern berührt und gewinnt, wenn man alle Kammern durchquert hat. Implementiert ist das durch ein Überprüfen, ob gewissen x bzw. z Werte erreicht wurden.

Intuitive Controlls:

Der Character lässt sich mit WASD und SPACE(Jump) steuern so wie es üblich ist.

Intuitive Camera:

Es wird eine first-Person Kamera verwendet.

Illumination:

Der großteil der Szene wird von einem directional Light beleuchtet (hier will ich noch nachbessern, bin jetzt aber nicht mehr dazu gekommen). Es werden die Materials aus dem .mtl file übernommen

Textures:

Die Lava am Boden der Kammern ist texturiert.

Adjustable Parameters:

Die jeweiligen Parameter sind in der config Datei im assets Ordner anpassbar.

Collision Detection:

Es wird PhysX für die Collision Detection verwendet. Als Collisions Objekte werden die Bounding Boxen verwendet, die auch für das View Frustum Culling verwendet werden. Die für die Collision Detection notwendige Simulation findet in der simulation Klasse statt

View-Frustum Culling:

View Frustum Culling ist mithilfe eines Kd-Trees implementiert. Dazu wird zuerst in jedem Frame in der protagonist Klasse ein Frustum erstellt, welches aus 6 Ebenen besteht. Zusätzlich wird für jedes Objekt eine Axis aligned bounding box berechnet, die dann gegen das Frustum getestet wird. Nur wenn die Box das Frustum schneidet, wird das Mesh auch an die GPU weitergegeben.

Hier habe ich mich an diesem Tutorial orientiert:
https://learnopengl.com/Guest-Articles/2021/Scene/Frustum-Culling

Vertex Shader Animation:

Die Vertex Shader Animation findet in PBR.vert statt (also im Vertex Shader). Hierfür wird zum einen übergeben, ob das Mesh animiert werden soll, und zum anderen die Zeit. Im Vertex Shader wird dann mithilfe von sinus Funktionen ein z Offset berechnet. Die Normal werden neu berechnet, indem wir die Richtungsableitung entlang der x und y Koordinate berechnen und dann das Kreuzprodukt der entstandenen Vektoren nehmen.

Die Animation wird für die Lava verwendet.

Orientiert habe ich mich hierfür an den Folien:
https://tuwel.tuwien.ac.at/pluginfile.php/2415211/mod_page/content/32/Animation_SS18.pdf?time=1619523068902

Specular Map:

Die Specular Map, wird so wie eine normale Textur gleich beim Objekt laden importiert und dann an die Grafikkarte übergeben. Im Fragment Shader wir dann anhand der UV-Koordinaten der Richtige Wert aus der Textur gelesen.

Specular Map wird auch bei der Lava verwendet.

Physically Based Shading:

Zum Code gibt es hier nicht viel zu sagen, da er sich nicht viel von den Tutorials unterscheidet, aber PBR ist der Grund warum wir von .dae auf .obj gewechselt sind, da die .mtl Files (die mit den .obj Files verbunden sind) bessere Information zu den verwendeten Texturen und Materials geben (zumindest bei den aus Blender exportierten Modellen). Bei den .dae fehlte unter anderem die metallic Componente (sie war auch nicht unter anderem Namen vorhanden).
Beim Physically Based Shading habe ich mich an den folgenden Tutorials orientiert:

- https://learnopengl.com/PBR/Theory
- https://learnopengl.com/PBR/Lighting

kd-Tree (Culling):

Wie Oben schon erwähnt verwenden wir einen Kd-Tree, um das View Frustum Culling zu realisieren. Dazu übergeben wir alle importierten Meshes an den Tree, der als erstes eine World Box (eine Bounding Box, in der alle Objekte liegen) berechnet. Liegen mehrere Objekte in dieser Box, dann wird nach der besten Teilung gesucht (Achse und Position auf der Achse) um die Kosten die durch die Teilung entstehen so gering wie möglich zu halten. Alle Objekte werden also entsprechend in zwei Nodes aufgeteilt. Dies passiert solange, bis sich nur noch ein Objekt in jeder Node befindet.
Beim Zeichnen wird der Baum durchlaufen und bei jeder Node wird überprüft, ob sie innerhalb des Frustums liegt. Ist dies der Fall, dann werden auch deren Kind Knoten durchlaufen, sonst nicht. Wenn ein Blatt Knoten durchlaufen wird, wird überprüft, ob das darin sich befindende Mesh innerhalb des Frustums befindet und dann dementsprechend gezeichnet.

Der Aufbau des Baumes wurde größten Teils von https://developer-blog.net/kdtree-in-c/ übernommen, die Traversierung des Baumes wurde selbst implementiert.