Submission 2

Group Name: Crestfallen
GitHub Link: https://github.com/jphoffmann/cgue20-Crestfallen
Students: Michael Steinkellner, 11705362
Jakob Hoffmann, 01505330

Modifizierte Interaktion:

Allgemein:

- ESC: Exit Game
- F1: Switch between Camera Modes
- F2: Reset to starting position

Player Camera:

- WASD: Bewegung der Kamera
- Maus: Kontrolle der First-Person Kamera
- Space: Springen (Länge des Knopfdruckes bestimmt SPrunghöhe und weite)
- E: Interaktion mit Objekten
- Shift: Doppelte Bewegungsgeschwindigkeit

Free roam Camera zusätzlich:

- Space: Bewegung/ “Flug” nach oben
- C: Bewegung/ “Flug” nach unten

HUD:

- H: Steuerungshilfe an und aus
- T: Spielzeit an und aus
- J: Sprungkraft-Anzeige an und aus
- F7: Informationen zu gezeichneten Objekten ein und aus

Libraries:

Collision: PhysX
Library Files: https://github.com/NVIDIAGameWorks/PhysX
Dokumentation:
https://gameworksdocs.nvidia.com/PhysX/4.1/documentation/physxguide/Index.html
Object-Loader: Assimp
https://www.assimp.org/
Text Rendering: FreeType
https://www.freetype.org
Sound: IrrKlang
https://www.ambiera.com/irrklang/

Features & Implementierung:

Collision Detection: Die Kollisionsdetektion wurde mittels Physx Character Controller implementiert. Da die Kollision Meshes aus Blender importiert werden (gekennzeichnet durch einen mit “Collision” benannten Material) wird der Cooking Prozess mit Triangle Meshes verwendet. Diese werden zu RigidStatic Actors, wobei die Simulation dieser abgeschaltet wurde, um Rechenzeit zu sparen.

Player-Control: Die Steuerung ist in der Klasse PlayerCamera implementiert. In der Methode handleKeyInput wird durch die des Physx Character Controller gelieferten Flags überprüft, ob sich der Charakter in der Luft oder am Boden befindet. Die Bewegung wird mittels polling überprüft, wobei der Bewegungsvektor lediglich mit der View Matrix multipliziert wird um den Bewegungsvektor zu berechnen. Dazu kommt Logik für den Sprung und Gravitation. Die Physiksimulation des Charakters wird in der Player Klasse verwaltet. Hier muss lediglich der Bewegungsvektor von der Kamera befragt werden, der Controller berechnet die nächste Position anhand aller Kollisionen, worauf die Kamera mit dieser Position geupdated wird.

Free roam Kamera: Die Free roam Kamera unterscheidet sich von der Player Kamera nur in der Steuerung. Beide Klassen erben von der abstrakten Klasse Camera, um diese zur Laufzeit wechseln zu können.

FPS and Framerate Independency: Die Framerate wurde in 1920x1080 auf einer GTX1080ti und GT 750M getestet. Sie hängt stark von der Anzahl der Lichter ab, kann daher je nach Konfiguration der Szene zwischen 20-144 FPS schwanken. Es wurde darauf geachtet, in allen Aspekten der Bewegung und dem Fortschritt der Physik Simulation Framerate-Abhängigkeiten zu vermeiden. Die Frametime wurde in diesen Operationen mit einberechnet und auf 144hz, sowie 20 hz getestet.

Object-Loader: Zum Einlesen der Objekte wurde Assimp mit FBX Format verwendet. In diesem Prozess werden auch alle Lichter und Texturen geladen, wobei zwischen normalen Meshes und Collision-Meshes (definiert mit zugewiesenem Material) unterschieden wird. Assimp: https://www.assimp.org/index.php

Textures: Diffuse und Spekulare Texturen werden von unserem Shader unterstützt und können durch Assimp geladen werden. Dazu wird die mittels Sampler abgetastete Farbe der Textur jedes Fragments mit der diffusen, sowie spekularen Komponente jedes Lichtes multipliziert.

Moving Objects: Die Szene besitzt 3 bewegende Platformen, zwei davon in der Mitte/Decke des ersten Raumes. Die Bewegung wird durch in Blender definierte Keyframes gesteuert, zwischen denen je nach Zeitpunkt interpoliert wird.

Illumination Model: Eine beliebige Anzahl (begrenzt durch Performance) an Lichtern und Materialien kann in Blender in die Szene gesetzt werden. Auch die Normalen können so

leicht abgeändert werden, um Kugeln rund oder, Ecken kantig darzustellen. Ein Beispiel von “Smooth Normals” kann bei den Gittern am Startpunkt gesehen werden.

Cel Shading: Im Fragment Shader wird die RGB Farbmatrix in eine HSV Matrix konvertiert, von welcher dann der V Wert diskretisiert wird.
Quellen:
https://tuwel.tuwien.ac.at/pluginfile.php/1721131/mod_page/content/37/CelShading_SS19.pdf?time=1553158642339

CPU Particle System: In der Klasse ParticleGenerator wird ein Container mit allen Partikeln eines Systems (einer Flamme) erstellt. Ein Partikel ist dabei ein zweidimensionales Rechteck mit einer Textur welches sich im Raum bewegt und leicht transparent ist. Jedes Partikel wird an einem festgelegten (leicht zufälligen) Startpunkt erstellt bekommt eine Lebenszeit, nach welcher es erneut startet, und eine Richtung in die es sich bewegt. Mithilfe von Instancing wird dann jedes Frame die Daten der einzelnen Partikel aktualisiert und an die, eigens für das Partikel System erstellten, Shader (Particle.frag und Particle.vert) geleitet. Dort werden sie so gezeichnet, dass sie immer in Richtung des Spielers zeigen. In der Klasse Flames werden alle Particle Systems gemanaged.
Quellen: https://learnopengl.com/In-Practice/2D-Game/Particles
http://www.opengl-tutorial.org/intermediate-tutorials/billboards-particles/particles-instancing/

Text Rendering: Für das Text Rendering wird FreeType verwendet und es wird in der Klasse TextRenderer implementiert. Damit erzeugen wir auf Basis einer ausgewählten Schriftart Texturen für die ersten 128 Ascii zeichen. Diese Texturen haben durchsichtige Elemente wofür wir den Blending Modus von OpenGL umstellen müssen. Wenn ein String gerendert werden soll werden die einzelnen Charaktere ausgelesen und den Texturen zugewiesen.
Quellen: https://learnopengl.com/In-Practice/Text-Rendering

HUD: Das HUD hängt sehr stark mit dem Text Rendering zusammen da es mit Hilfe der Klasse TextRenderer Strings Planar auf den Screen zeichnet. Dafür sind die Shader HUD.frag und HUD.vert verantwortlich. Alle HUD Elemente passen sich in ihrerer Größe und Position an die Auflösung an.
Quellen: https://learnopengl.com/In-Practice/Text-Rendering

Sound: Sound wird mit Hilfe der Library IrrKlang implementiert. Dafür wird in allen Klassen welche Sound abspielen sollen (main, Flames, PlayerCamera) eine SoundEngine erstellt welche dann 2D oder 3D Sounds abspielen kann. Für 2D sound muss einfach nur ein Sound aus den Assets ausgewählt werden und spezifiziert werden ob der sich wiederholen soll oder nicht. Für einen 3D Sound brauchen wir zusätzlich noch eine Position von der er kommen soll, wie weit er sich ausdehnt, die Position des Spielers und seine Blickrichtung. Solche 3D Sounds werden in der Klasse Flames für jede Flamme erzeugt.
Quelle: https://www.ambiera.com/irrklang/

https://www.ambiera.com/irrklang/tutorial-3dsound.html

Advanced Physics (6 Points):

Mit Hilfe des Physx Character Controllers rutscht der Spieler von jeder schiefen Kante hinab. Ein BehaviorCallback wird aktiviert, sobald ein Spieler auf eine Platform springt (welche als PxObstacle definiert wurde), worauf er mit ihr mitfahren kann, ohne abzurutschen. Auch wird (in PlayerCamera) abgefangen, ob der Spieler den Boden oder mit dem Kopf eine Decke berührt, worauf seine Beweglicheit eingeschänkt, oder die Flugbahn verändert wird.

View-Frustum Culling:

Objekte werden in world space gegen das View-Frustum getestet. Dafür wurde beim einlesen der Geometrie jeweils eine Axis Aligned Bounding Box, definiert durch einen min und max Vektor berechnet. Der Culling Prozess testet dabei immer den am weitesten von jeder Plane des View-Frustums entfernten Vektor des Objekts.
Quelle:
http://www.lighthouse3d.com/tutorials/view-frustum-culling/geometric-approach-extracting-the-planes/

Shadow Volumes:

Stencil Shadow Volumes wurden mit der Depth Fail Methode implementiert. Im ersten Pass wird einmal das ambient lighting berechnet und die Szene in den Depth Buffer geschrieben. Für jedes Licht wird nun das Shadow Volume definiert und mit den Depth Werten der Stencil Buffer beschrieben. Schließlich wird mit additiven Blending jedes Objekt beleuchtet.
Die Schatten besitzen jedoch starke Artefakte. Wir nehmen an, dass dies durch die Berechnung der Indices entsteht.
Quellen:
http://ogldev.org/www/tutorial40/tutorial40.html,
https://developer.download.nvidia.com/books/HTML/gpugems/gpugems_ch09.html

Hierarchical Animation:

Bewegende Plattformen besitzen ein Zahnrad welches sie vorantreibt, und sich unabhängig von der Plattform dreht. Auch die Rotation ist mit Keyframes beschrieben und wird interpoliert, wobei die letztendliche Transformation jedes Meshes von den “Eltern” der meshes abhängt.
Quelle:
https://tuwel.tuwien.ac.at/pluginfile.php/1721131/mod_page/content/38/Animation_SS18.pdf

Win/Loose Condition & Anleitung des Spielverlaufs:

Das tatsächliche Gameplay basiert darauf, gezielte Sprünge zu machen um aus dem Turm zu entkommen. Die Tür befindet sich dabei ganz oben, hinter der Startposition des Spielers. um die Tür zu öffnen, muss jedoch erst ein Schlüssel aufgehoben werden, welcher sich ganz oben im Nebenraum befindet. Dieser Raum befindet sich ganz oben, gerade von der Startposition aus. Entkommt der Spieler, so wird ihm seine benötigte Zeit auf einem Win-Screen angezeigt.