Clandestien – Submission 1

## Dokumentation

# General

Our game „Clandestien“ is a 3D-puzzler, which takes place in a 20th century styled, haunted mansion. It is a first-person game, where the player is able to roam around the house and explore different puzzles to find a way outside the mansion, as all doors appear to be locked. We plan to incorporate various puzzles through making use of the “effects” listed on the effect list, e.g. a portal that takes the character a level above, which must be activated through throwing a shadow (shadow mapping) on the wall. The game is rendered using OpenGL as graphics API, GLFW for window management, and other external libraries for e.g. collision detection, which will be listed below.

# Gameplay

**3D Geometry**

We modelled our game stage (the mansion) using Blender. The game stage consists of two floors, a few windows and doors. Other than that, we modelled a projector and table, which will play the main part of our shadow portal puzzle. We further plan to model a chimney, which will incorporate our particle system (fire). The Blender objects are then triangulated before exporting them. To make use of our .obj-Files, we have written an object loader ourselves, which can be found in the “Mesh.cpp” class. Object files represent a collection of mesh information, more specifically vertex, normal and texture coordinates, faces and position. <Anand>

**Playable**

Besides basic player input like camera and character movement, we implemented a prototype of our portal, which will be part of our main puzzle. Additionally, we implemented a kinematic character using PhysX, which can interact with objects like the Blender-Ape to move it around. We created static bounding boxes for our game scene in Blender, exported their position and scale data and created static collider using the PhysX API as described in the official Nvidia PhysX SDK documentation. (<https://docs.nvidia.com/gameworks/content/gameworkslibrary/physx/guide/Manual/RigidBodyCollision.html>). Furthermore, we aggregated the static shapes for optimization purposes and for the sake of keeping our colliders grouped. As for the player character, we choose to implement a kinematic capsule character. (https://docs.nvidia.com/gameworks/content/gameworkslibrary/physx/guide/Manual/CharacterControllers.html#kinematic-character-controller

**Min. 60 FPS and Framerate Independence**

**<Anand>**

**Win/Lose Condition**

As our game is a puzzle game, the win/lose condition is given by the nature of the game. The player either succeeds to solve the puzzles, thus leaving the mansion or not.

**Intuitive Controls**

The basic controls of our game (camera and character movement, interaction) are implemented through reading a .ini file, which can be configured before running the executable. The default implementation uses the keys W, A, S, D for movement and E for interacting. Furthermore, the mouse movement mirrors the camera movement (character’s POV). These are standard game controls and should be intuitive for anyone who played a game on PC before.

**Intuitive Camera**

As already mentioned, the camera movement follows the character’s point of view. The camera mirrors the character’s eye movement, by being able to look around in every direction. As we implemented a FPS camera, there is no pivot radius, which would make our camera orbiting around the scene and thus creating a wrong perception of the character’s whereabouts and movement.

**Illumination Model**

Our game features a basic illumination model, which is defined in our shaders. We make use of three basic point lights for the illumination of our game stage. Furthermore, through our implementation of an object loader, we took care of normal and texture vectors needed for a realistic illumination model.

**Textures**

Our textures are loaded into the shader program, and through the use of the texture coordinates loaded by our object-loader we can apply them to different objects like the game scene.

**Adjustable Parameters**

Parameters like screen resolution, full-screen mode, brightness and refresh-rate can be adjusted through our provided .ini-file, by configuring the values before starting the executable.

**Collision Detection (Basic Physics)**

We implemented basic collision detection through the PhysX SDK. The external library was bound in by following the tutorial provided by TUWEL. By following the docs provided by Nvidia (<https://docs.nvidia.com/gameworks/content/gameworkslibrary/physx/guide/Manual/Index.html>), we implemented a basic scene and physics. Then we implemented static colliders and a kinematic capsule controller as mentioned in “Playable”. Through scene queries we plan to detect if the player character is colliding with walls, floors, or other objects.

# Effects

**Bloom/Glow**

After rendering the scene, we render a filtered view of the rendered scene, where parts above a certain brightness are kept, into a smaller framebuffer where it is then blurred. After blurring the smaller buffer, it is recombined with the unmodified scene to achieve a glow around bright spots of light.

**Simple Normal Mapping**

We generate the tangents for the objects at runtime using the uv coordinates combined with the vertex positions. Then the TBN matrix is built in the vertex shader stage using the normal and tangent data and passed to the fragment shader where it is used to transform the normal data out of tangent-space into view-space. This normal is then used for lighting computations.

**CPU Particle System**

We keep an array of particle data on the cpu side, there we do all the calculations for the particle simulation. For rendering the data is copied into gpu memory, transforms are handled in the vertex shader, mesh generation is handled in the geometry shader.

# External Libraries

**PhysX SDK**

The Nvidia PhysX SDK was used to implement simple physics like collision detection. The API provides a framework for creating static and dynamic colliders, materials, physics (like gravity) and character controllers. We used the API for creating a kinematic capsule controller, static and dynamic bounding boxes (shapes), as this is mandatory for our game to be playable. We need scene queries to interact with objects, static colliders to not move through our game stage, dynamic colliders for moving objects.

**Lua (Sol)**

We are not using Lua effectively yet, but plan to integrate it as it is easier to control various aspects of our game with a scripting language. Currently we have a Python-Export script for our Blender-Scene, which in the future should export all the data we need to Lua tables, which are then incorporated in our game through using the bindings provided by Sol. Ultimately we plan to write a short wrapper for those bindings.

**OpenAL**

We are using the OpenAL library to play sound (.wav) during our game. For now we have successfully bound the library into our project and tested it by playing a simple sine wave and .wav-file.