

Clandestien – Submission 2

Dokumentation

1. 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. Puzzles are solved by user interaction e.g. moving around objects or text input. 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.

How to solve it: The player must pick up a film reel laying on the floor by pressing E while its in the middle of the screen (looked at by the player). Then, the film reel must be inserted into the projector, causing a spotlight to go on. Furthermore, the gargoyle must be moved by pressing E and holding one of the movement keys. He must be moved in the middle between the spotlight and wall, casting a shadow and opening a portal to the second floor. Next to the projector a poster can be found, which translates the Enochian alphabet. The key up the floor is hexed, and the hex must be lifted by pressing the right sequence of keys that is given as a hint on the same table (the interaction can be left with the R key). The letters of the alphabet can be decrypted downstairs. Once the key is in the inventory, the player must move back to the first floor, where he is able to finally open the door to the exit. Before fully exiting, he must navigate through a maze to get to the finish point.

2. 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. 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.

Playable

Besides basic player input like camera and character movement, the player is able to interact with objects in specific ways, such as moving them around (Gargoyle) or picking them up into the inventory (Filmreel). The interaction key can be set before launch (.ini config), and its default value is “E”.

Advanced Gameplay

Our game features a portal, which can take the character from one level to another and must be opened through solving a puzzle (moving the gargoyle so it casts a shadow, and activating

the projector with the right film reel). The game is designed so the player needs to move through the portal to achieve goals (like getting a key for one of the doors). Furthermore, we evaluate user input through an input manager that enables us to call multiple registered key callbacks when needed.

Min. 60 FPS and Framerate Independence

The program currently runs well above 60fps on the systems we could test it on. We achieve framerate independence by taking the delta time of the current frame into consideration in all the calculations.

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 and one spotlight which casts shadows for one of our puzzles. 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.

Moving Objects

As already explained, our Gargoyle object is movable by the player character. The movement is smooth and follows the character's pace to seem realistic (pushing/pulling). It can be moved by holding the E key while moving.

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. Furthermore we implemented some dynamic objects, which are queried by raycasts so the player is able to interact with them.

3. 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.

Shadow Map with PCF

The shadow maps are only applied for the spotlight, which the projector casts after being activated as it is crucial for our puzzle to act like this. We use a depth shader to acquire depth maps before our scene is rendered, which are then passed as texture to our standard shaders. The shadow values are then being calculated in the fragment shader. We use a bias matrix and the “texture()” function to get rid of aliasing effects like “peter panning” as described in various tutorials.

4. 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.