Documentation

Group:

SLAI

Additional Libaries

PhysX 4.1 was used to implement a kinematic character controller (PxCapsuleController), as well as collision of objects.

tinyobjloader was used to implement a simple loader for .obj files.

Features

The player is able to

- Move (WASD)
- Jump (Space)
- Crouch (Left Control)
- Run (Shift)
- Pick up Objects (LMB [Left Mouse Button])
- Throw held objects (RMB [Right Mouse Button])
- Drop held Objects (LMB)
- Interact with terminal to enter "cyberspace" (LMB)
- Toggle spectator mode (F5)

Adjustable Parameters include:

- Toggle fullscreen (F11)
- Lower brightness (O)
- Increase brightness (P)
- Toggle normals (N)
- Adjust refresh rate (trough config file)

Walk-through:

The game features 2 levels. In the first level the player character starts out in a vent he has to traverse by crouching. He then finds himself in a room with a closed door and a terminal. He can interact with the terminal (LMB) to switch to cyberspace to hack the doors security system.

He can interact with the terminal to switch to cyberspace to hack the doors security system. In cyberspace there is a small physics puzzle, in which he has to use the three cubes (BLUE) found around the virtual representation of the real world to traverse and retrieve the data cube (RED) on the other side of the trench.

He then has to put the data cube (RED) into the slot where the terminal was in the real world (marked in RED) to exit cyberspace.

If the Cube was successfully inserted, the door opens and the player enters the second level. Here, the player has to pass a small crouch and jump section, where he has to avoid hitting lasers. Afterwards the player has to make a timed sprint in order to avoid getting hit by a moving surveillance drone that has detection sensors.

If the player gets hit by the sensors or touches the lasers the game is over and the player has the option to restart the game.

If he passes all obstacles, the player is able to deactivate the evil AI inside the tank by interacting with the terminal in front it.

Brief Description of Implementation

- CharacterController: The character controller is implemented as a physx character controller. It uses the move function to move the player in a desired direction. The movement vector changes values depending on the user input. A downward force is applied in order to simulate gravity and through a ray it is checked if the player is currently grounded.
- InputManager: The InputManager uses an array of callbacks assignable to keycodes. For continuous effects, polling is used.
- Collision detection: Collision detection is implemented in the CustomPhysxFilter. Physx triggers are utilised by a custom trigger class that creates a shape using the geometry and sets the filter layer for the shape. The layers are defined with an enum and based on which layer collide with each other certain functions inside the scene manager are triggered. A custom physX filter function is used to disable collision with the cube when held. A custom SimulationCallback is used for the Triggers described above.
- GameObjects: There are different types of GameObjects that have different characteristics, such as lasers which have a start and end point and portables that the player is able to pick up. Each portable game object stores a spawn point that is used to reset them to their original position, if the player restarts the game. Each gameobject can have an assigned collider, which is added to the physx scene and a geometry, which is used for drawing the geometry to the screen. They are positioned and rotated equally.
- ObjectLoader: The object loader loads static objects with complex geometry from OBJ files utilising tinyobjloader. It loops through each shape contained in the file and creates a vertex and index array that is then stored in a Geometry object. It returns an array of static game objects, which are then added to the scene.
- FPCamera: The implemenation of the first person camera which extends the given Camera class. The camera’s orientation is updated based on the mouse movement.
- SceneManager: The SceneManager keeps track of the currently active scene and the scene that is expected to be next so it doesn’t have to draw all scenes every frame. It is also mainly switch between the different levels by setting the player position to the currently active level or restarting the game.

- Level/Scenes: Each level loads the gameobjects when it is initialized and stores them in an array. This array is used to draw the scene each frame.
- RescourceManager: The RescourceManager is used for storing all shaders and materials in one place.
- Materials: The Material class loads textures from DDS files and assigns a shader to them. These shaders are then utilised inside Geometry for the drawing of the Objects.

Gameplay

Mandatory

- 3D Geometry: Complex 3D Geometry is imported and rendered as described in the implementation of the ObjectLoader and the GameObject. All of the static objects in the scene were modelled in blender and imported through OBJ files. In the engine a material, collider and geometry is assigned. Examples for complex geometry are the console in level one, the liquid tanks in level two or the wires hanging from the ceiling in level 2.
- Playable: The game has basic mechanics such as jumping and crouching.
- Advanced Gameplay: The game contains physics based puzzles, multiple levels, interactive objects, and moving and static obstacles.
- Min 60 FPS and Framerate Independence: Framerate indepence is ensured by calculating the time between frames and using it for calculations that move objects, the camera or the player. The games runs at a minimum of 60 FPS (FPS are printed to console).
- Win/Lose Condition: If the player touches a laser or a drone sensor the game is lost and all object that changed position get set to their initial position. If the player is able to overcome all obstacles, solves the puzzles and interacts with the console the game is won. The player gets informed immediately about each outcome.
- Intuitive controls: The controls (W, A, S, D, LMB etc.) are commonly used and intuitive for computer games.
- Intuitive Camera: The game implements a typical first person camera used by most games.
- Illumination model: For the illumination model phong shading is used. The game has one point light that is colored red and updates its position with the position of the player.
- Textures: Each object that is loaded or created has an assigned texture that is sampled within the fragment shader. Textures are loaded from DDS files.
- Moving Objects: The game contains one moving object (the drone) that gets moved by updating its position vector each frame, only if the player is currently in level2, which is ensured by the scene manager.
- Adjustable Parameters: Parameters are adjustable as already mentioned in the feature list above.

Optional:

- Collision Detection (Basic Physics): Basic physics are implemented through the use of physx.

- Advanced Physics: Portable objects were implemented, that can be picked up and transported or pushed away by walking against them. The player also has the option to release them with an applied force, therefore throwing them.

Effects

- Blobby Object Using Marching Cubes:
  The marching cubes algorithm was implemented as a geometry shader. It contains some SDF functions. It expects an array of point primitives and then uses the marching cubes algorithm to generate a strip of triangles with max 15 vertices.
  Resources used:
  For the MC algorithm: http://paulbourke.net/geometry/polygonise/, this was used, but adapted to use less conditional logic, however we then needed to split up the triTable, as it was too large for my GPU to handle. We also added normal calculation.
  For SDF functions: https://iquilezles.org/articles/distfunctions/, a great resource for anything SDF
  This effect can be seen at the end of the second level.
- Vertex Shader Animation: Was implemented by offsetting the the vertex position using layered sine and cosine waves to look like water when applied on a subdivided plane.
  This effect can be seen at the end of the second level in the 4 tanks surrounding the blobby object.
- Procedural Texture: As a starting point we used texture.frag and used the hash function proposed by Hugo Elias here https://www.shadertoy.com/view/flX3R2, we then took inspiration from many other shaders on shadertoy.com and implemented a noise function, which we layerd to look like marble. We then also added a conditional statement to achieve a checkered look.
  This effect can be seen on the walls of the rooms you get sent to when you lose or win.
- Simple Normal Mapping (learnopengl.com: Normal Mapping): We sample from the normal map and adjust the vector, but only if the uniform to sample from the normal map is set to 1. We then construct a form of TBN matrix to adjust the normal vector so it looks right on axis aligned surfaces.
  This effect can be seen on the floor of the first level and be toggled on/off using the N key.
- Bloom/Glow: (https://learnopengl.com/Advanced-Lighting/Bloom): This effect can be seen throughout, but especially well in cyberspace, as the cubes are always treated as if they were bright enough for bloom to take effect, or when the brightness is increased. The only deviation from the resource we used was that we do not do tone mapping and gamma adjustment.
- Shadow Map with PCF (Source: LearnOpenGL - Point Shadows): This effect can be seen throughout the level, as the light source moves with the player, offset slightly to the right. To see it more clearly, one can enter spectator mode, as the light stays fixed on the players physical position. To implement shadow mapping for the point light source, we render the scene from the point light source’s perspective, to a cubemap. We then sample this cubemap in our usual fragment shader for calculating the shadows.