Submission 2

Group/Game Name: #69 / Frisbee Frenzy

Description of Implementation

Features

- Frisbee Game where you have to throw a frisbee around some obstacles into a target area to win the game. There are moving obstacles in the game, that have to be avoided. It is a third-person game. All Objects are loaded in using the assimp library. The game uses cel-shading + edge detection as a postprocessing effect. It also features shadow-maps and procedural textures.

General

Our game is devided into two levels. At the beginning the first level is loaded by loading .obj file using assimp-library to the _geometryMap, where all objects of a level are stored. The classes AssimpGeometry and AssimpMesh are written with the help of this learnOpenGL tutorial: LearnOpenGL - Assimp. In the main render loop, this map gets iterated through and all objects are drawn. Furthermore in the loop the scene is rendered into a framebuffer object which has a color attachment. This Texture is then used to acchieve the post processing effect of edge detection (more on that in the section “Effects”). For objects that need to be updated, we wrote the update function, which is called every frame.

Movement & camera

In RUNNING status, the player can move freely using the WASD keys. The function processInput(…) listens to key events and moves the player and camera to the correct position. When throwing a disc, the position is fixed and only the camera rotation is possible.

```
void processInput(GLFWwindow* window, float deltaT) {
  if (glfwGetKey(window, GLFW_KEY_ESCAPE) == GLFW_PRESS)
    glfwSetWindowShouldClose(window, true);

  float cameraSpeed = static_cast<float>(_cameraSpeed * deltaT);
  if (glfwGetKey(window, GLFW_KEY_W) == GLFW_PRESS)
    cameraPos += cameraSpeed * cameraFront;
  if (glfwGetKey(window, GLFW_KEY_S) == GLFW_PRESS)
    cameraPos -= cameraSpeed * cameraFront;
  if (glfwGetKey(window, GLFW_KEY_SPACE) == GLFW_PRESS)
    cameraPos += cameraSpeed * cameraUp;
  if (glfwGetKey(window, GLFW_KEY_LEFT_CONTROL) == GLFW_PRESS)
    cameraPos -= cameraSpeed * cameraUp;
  if (glfwGetKey(window, GLFW_KEY_A) == GLFW_PRESS)
    cameraPos -= glm::normalize(glm::cross(cameraFront, cameraUp)) * cameraSpeed;
  if (glfwGetKey(window, GLFW_KEY_D) == GLFW_PRESS)
    cameraPos += glm::normalize(glm::cross(cameraFront, cameraUp)) * cameraSpeed;
}
```

The camera listens to mouse movement events with the `mouse_callback(...)` function. Using the x-position and y-position, the offset is calculated and viewing direction are adapted accordingly. The camera and the player's movement are implemented with help from the leanOpenGL totorial: LearnOpenGL - Camera.

Moving Obstacles

All Moving obstacles are transformed using the same principle. A modified model matrix is applied to simulate continuous movement of the model. For instance, the thrown disc is first translated to the origin to perform a safe rotation, and then moved back to simulate movement in the throwing direction. This creates the illusion of throwing an actual rotating disc in the air. The update is performed every frame in the render loop. For the other obstacles, such as the opening door and moving wall, only the translate operator is applied under the restrictions of minimum and maximum position.

```
if (_discMovement < maxDiscMovement && _discFlying) {
//rotate disc in flight
frisbeeModel->transform(glm::translate(glm::mat4(1), -discPosition));
frisbeeModel->transform(glm::rotate(glm::mat4(1), glm::radians(-3.0f), glm::vec3(0, 1, 0)));
frisbeeModel->transform(glm::translate(glm::mat4(1), discPosition));
frisbeeModel->transform(glm::translate(glm::mat4(1), _discDirection * discSpeed * deltaT));
_discMovement += discSpeed * deltaT;
}
```

HUD

The HUD was implemented by loading images from files using the SFML library. The images were then rendered onto a 2D-quad using orthographic projection. This implementation can be found in the `hud.cpp/hud.h` class as well as in the `hud.vert` and `hud.frag` shaders. Shader handling is done by the `Shader.h` file from the ECG-framework.

Controls

- WASD: move the player.

- STRG + Space: move up and down.
- E: Pick up the disc if near it
- H: Toggle HUD
- If in throwing mode: LEFT_MOUSE_BUTTON: drag up or down to control the distance of the throw. release button to throw.
- R: Reload level
- B: Win the level (debug)
- 1 + 2: quick-switch between levels
- Q or ESC: Quit the game

Effects

Cel-Shading

The main shader for the program is a cel-shader, wich can be found in cel.frag. Cel shading was chosen to acchieve a cartoon stlye game. This is done by having only 4 layers of different light values, as calculated in the cel function of the fragment shader.

```
vec3 cel(vec3 n, vec3 l, vec3 diffuseC) {
    float d = length(l);
    l = normalize(l);
    vec3 r = reflect(-l, n);
    float diffuseF = max(0, dot(n, l));
    float quantizedDiffuse = floor(diffuseF * levels) / levels;
    return quantizedDiffuse * diffuseC;
}
```

Edge-Detection

Complementing our cel-shading, we have implemented edge detection as a postprocessing effect. I have not found great tutorials about this topic, as i wanted to acchieve a special look. After loosing way too many hours on this topic, i finally came to a solution. I used the depth-values from the scene in combination with the normals of the scene to get all edges between differen geometries. These values were stored in a Framebuffer object that the main scene is rendered to in the render loop. There are three color attachments to this framebuffer: one for depth, one for normals and one for the cel shaded colors. These three textures are then used in the edge.frag shader to color all fragments correctly. The scene is then finally rendered to a screen sized quad with all effects combined.

Framebuffer-Tutorial: LearnOpenGL.com

Procedural Textures

Procedural textures were implemented using noise. This was achieved by writing a function that generates a 2D checkered grid with yellow and dark gray colors. In the dark gray cells, the generated noise is visible in the form of larger noise blocks, giving it a carbon-like look. The texture is generated using the fragment shader.

Procedural Textures-Tutorial: upvector.com

Shadow Maps with PCF

To acchieve this effect, we render the scene from the perspective of the light into a framebuffer object. This fbo has a cubemap attachment to store the depth-values of the scene in all six directions of the light (we use a point light as main light so the shadows have to be casted in every direction). When making the main shading run with the cel.frag shader, the shadow is calculated for each fragment using the depth information from the cubemap. In this method a bias is introduced to take care of any shadow-acne. We tried very hard to find a way to get rid of the peter panning as well, but couldnt find a way, where both acne and panning were eliminated. We therefore settled on this solution, where we have no shadow acne, but very noticable peter panning. PCF is implemented by taking multiple samples around the fragment and then averaging the results to get smooth shadow-edges.
Shadow maps have been implemented with help of this learnOpenGl turorial.

mandatory gameplay features implemented

3d Geometry, Playable, Advanced Gameplay, Min 60 FPS and Framerate Independence, Win/Lose Condition, Intuitive controls, Intuitive Camera, Illumination model, Textures, Moving Objects, Documentation

Libraries

- **Assimp** (Object Loader): https://github.com/assimp/assimp
- **stb 2.28** (Image Loading): https://github.com/nothings/stb