FOX AND FOUND

In our game, you take on the role of a mother fox whose babies have wandered off and gotten lost in the wild nature. Your task is to find all five babies within a time limit of four minutes. The babies have hidden themselves in various locations, so you'll need to search high and low to locate them.

As you navigate through the game world, you must be cautious of the bears that roam the area. If a bear catches you, you will lose one of your three lives. Losing all your lives will result in a game over, so it's essential to avoid the bears as much as possible.

GAMEPLAY

3D Geometry

When creating and loading the fox mother, fox babies, and bears in our game, we utilized external .obj files that were exported from *Blender*. We loaded these files into the game using the *Assimp*-library (https://www.assimp.org/). To read the .obj file format with *Assimp*, we implemented a routine that involved creating a mesh for each object. We found helpful guidance and code examples for this process on the website "*Learn OpenGL*" (https://learnopengl.com/Model-Loading/Assimp).

Playable + Advanced Gameplay + Win/Lose Condition

To meet this requirement, we have implemented several enhancements to ensure a sophisticated gameplay experience. Building upon the foundation of the Playable section, we have focused on refining the game logic and design, as well as increasing the complexity of the gameplay.
In our game, the narrative revolves around a mother fox whose babies have wandered off and become lost in the wild nature. As the player controls the mother fox, the primary objective remains the same: to locate and collect all five babies within a time limit of four minutes. When navigating through the big game world, one has to look for evil bears, that would steal a life from you and respawn you to the start-point when you collide with them.

If you can find all baby foxes within that four minutes, you win the game. But if you fail that within the limited time or lose all your three lives, you lost it.

Min. 60 FPS and Framerate Independence

We have ensured that our game never drops below the 60FPS threshold. Typically, the game runs at around 160FPS in Build Mode and between 250 and 450FPS in Debug Mode, but players can activate a feature by pressing the F4 key to cap the frame rate at a consistent 60FPS. Additionally, we calculate the delta time in each

game loop, which is then multiplied with character movement to maintain frame rate independence.

Intuitive Controls + Intuitive Camera

In our game, players control the fox mother using the W and S keys to move forward and backward, and the A and D keys to move left and right. To change the direction, players can rotate the fox and the camera around the x-axis using the mouse. The camera is implemented as a Third-Player-Camera, which means it always stays behind and follows the mother-fox. To collect a baby fox, players simply need to locate one and click the space-bar.

Illumination Model + Textures

In our game, we have a sun, which is implemented as a Point-Light. Moreover, each object has an assigned material and contains normal vectors. The terrain and skybox have also a texture, which was attached with the help of UV-coordinates.

Moving Objects

The “enemies”, or rather bears, are moving around in a circle, in order to make it a bit more difficult for the player to avoid them.

Adjustable Parameters

It is possible to adjust the Screen Resolution, the Fullscreen-Mode, the Refresh-Rate and the Brightness in the settings.ini file.

Collision Detection (Basic Physics)

In order to place and move everything on the static terrain, the physics-engine “PhysX” (https://github.com/NVIDIAGameWorks/PhysX) was used to create the static environment (terrain) and use character controllers for each object to collide with that environment.

Heads-Up Display

We have incorporated a Heads-Up Display (HUD) in our game, displaying important information at the top of the screen. The HUD includes indicators for the player's remaining lives, the time remaining, and the number of baby foxes found. To create the HUD, we utilized the *FreeType* library (https://freetype.org/), which allowed us to implement a class and functions for rendering text. For this, we also found helpful guidance and code examples on the website "*Learn OpenGL*" (https://learnopengl.com/In-Practice/Text-Rendering).

Furthermore, we have included additional features in the lower half of the game window. Players can view the current frames per second (FPS) and toggle the wireframe mode on or off using the F2 and F3 keys, respectively.

At the start of the game, we implemented a loading screen using *FreeType*, providing visual feedback to the player while objects and functions are being loaded. Additionally, a help screen is displayed, showcasing the essential functions and controls of the game. Players can toggle the help screen on and off by pressing the "H" key. When the help screen is active, time is paused to allow players to read and understand the instructions without any time pressure.

EFFECTS

Terrain

Tessellation from Heightmap

A terrain is rendered by creating a plane mesh out of quads and taking the height for each vertex from a heightmap that is loaded as a texture. The heightmap was taken from Thin Matrix (OpenGL 3D GameTutorial 21:Terrain Height Maps, (https://www.dropbox.com/s/dcul3fnrnejue7x/heightmap.png?dl=0). The terrain is generated in the Terrain.h/Terrain.cpp class. Tessellation shaders were written and loaded using a TessellationShader class. In the tessellation control shader (tess.tcs), the tessellation levels are set depending on the distance between the vertex and the camera. In the tessellation evaluation shader (tess.tes), the new positions are calculated – the height is taken from the heightmap texture and the normal are calculated.

Texturing

Environment Map

We have created a skybox using a large cubemap texture. The skybox is made up of a cube with a six-sided texture, which moves along with the player to prevent them from passing through it. In order to do that, an own shader was implemented (skybox.vert and skybox.frag) and the code for that was implemented in the Skybox class. The finished skybox gives the player the illusion that they are in a vast environment, larger than it actually is.

Once the skybox was loaded and implemented, we were able to perform environment mapping. To simulate a small pond (Water.h/Water.cpp), we created a cylinder that reflects the skybox, creating a reflective surface with the cubemap.vert and cubemap.frag shaders. This adds a visually appealing element to the game world and enhances the overall atmosphere.
In order to implement the code for the skybox and the environment mapping, we used again the help of the website “Learn OpenGL” to get inspired by their shaders and code examples and understand the theoretical backgrounds (https://learnopengl.com/Advanced-OpenGL/Cubemaps).

Post Processing

Lens Flare

To give the flare effect when viewing the light source, lens flare textures from Thin Matrix () were rendered onto the screen. To enable lens flare, a texture class (FlareTexture.h/cpp) was written. The textures get passed to get FlareRenderer that creates a quad for the textures, starts the query and does the occlusion test to prevent the lens flare effect to be visible when objects are blocking the view and renders the flares. The FlareManager class calculates the screen coordinates and flare positions before passing them to the FlareRenderer.

Shading

Cel Shading

Cel shading was used to give the terrain a more “toon” look by discretizing the colors to a specified number of levels. This was implemented in the fragment shader (tess.frag). To get discretize the values for the levels, the colors depending on the texture for the terrain were transformed into HSV values, the discretized and finally transformed back to RGB values to pass out if the fragment shader. The functions to efficiently transform RGB to HSV and vice versa were taken from Toon HSV simplify - NatLab(https://www.shadertoy.com/view/Ddlyzr).

Additional Libraries:

STB Image Loader:

Was used to load various textures e.g. for the terrain and the skybox.

Assimp Object Loader:

Was used to load the .obj files as meshes into the game

PhysX Nvidia Engine:

Was used to implement collision detection and gravity

FreeImage:

Was used to create a Material when loading an .obj File into the scene

FreeType Text Renderer:

Was used to implement the Heads-Up Display with rendered Text