General

Group- / Game Name: Pizza Run
GitHub Link: https://github.com/paulSpoerker/cgue22-Pizza-Run/
Students: Paul Spörker (11912400)
Genre: Jumping Game
Goal: Jump around the map to collect 3 Pizzas and then get to the oven within 50 seconds, without falling into the lava.

Implementation

Vertex Shader Animation
To animate waves in geometry, the uniform “animateWave” (for the textureShader) is set to true before the draw-call. I use this boolean since the “normal” shader and an extra shader, just for wave-like animation, would be almost identical. In the texture.vert the vertices are translated along the x- and z-axis, depending on the sine of time, if the animateWave-boolean is true. Directly after that, the normals are recalculated. Amplitude and frequency can be changed in the texture.vert.
To see the reflections better, increase amplitude in texture.vert:27.
https://medium.com/@joshmarinacci/water-ripples-with-vertex-shaders-6a9ecbdf091f

CPU Particle System
For the particles, I use a single header-file, where methods, variables and a Particle- and Data-struct are defined.
In the constructor, random positions, lifetimes and colors are set for the specified amount of particles. Also necessary buffers are created and particle vertices, positions and data are bound.
When calling draw(float dt) the particles are updated and then drawn.
In the update(dt) function the lifetime of all particles is decreased and their position gets translated upwards. Also their color gets darker, depending on their remaining lifetime.
In the draw() function buffers are bound, updated and finally the particles are drawn using instancing.
In the shader for particles (particles.vert and particles.frag) the particles are always drawn to face the camera.
The particles are emitted above the oven.
https://levelup.gitconnected.com/how-to-create-instanced-particles-in-opengl-24cb089911e2

Environment Map
For the environment map, the cubemap textures are loaded and an environmentTexture and environmentTextureMaterial is constructed. After that, I construct a cube, for which I do not use the Geometry-class, because we want to see the cube from the inside.
To draw the skybox, the skybox-uniform-boolean of the environment shader (environment .vert and environment .frag) is set to true. For that the translation of the viewMatrix is removed (because we want it to stay at the same position in relation to the player).

If the skybox-boolean is false, the object is reflecting the cubemap. To achieve this effect, the fragment shader first calculates the vector from the “eye” to the object. Depending on that vector and the normal of the object, it calculates the reflection vector, with which it samples the texture from the cubemap.
You can see this effect on the “lamp” (flying sphere in the middle).
https://learnopengl.com/Advanced-OpenGL/Cubemaps

Lens Flares
To show lens flares, I created a header file, which calculates the necessary information. First, the position of the light source is converted to screen-coordinates. Depending on the distance between these coordinates and the screen center, the brightness of the lens flares is calculated. If the light source is not on the screen or the brightness is to low, the method returns false, so that no draw calls for the flares are made. The vector between the light and the screen center, determines the scaleVector. The separate lens flares are drawn along this scaleVectors direction and are spaced evenly according to its length.
In the shader for lens flares (lensFlares.vert and lensFlares.frag) the flares are always drawn to face the camera and are always the same size.
The lens flares are visible when facing the “lamp”.
https://www.youtube.com/watch?v=OiMRdkhvwqg

Features

Effects:

- **CPU Particle System** (Advanced Modeling)
  Simulates smoke coming out of the oven.
- **Vertex Shader Animation** (Animation)
  Simulates waves in the lava.
- **Environment Map** (Texturing)
  Use a Cube Map for the environment and reflections on the “lamp”.
- **Lens Flares** (Post Processing)
  Lens Flare effect on the “lamp”, to make it more realistic.

Gameplay-Compulsory:

- **3D Geometry**
  The oven (designed in Blender) gets loaded by my own modelLoader-method (in Geometry.cpp), which reads in a .obj file.
  At first, the vertices and indices are stored in the according list, depending on the prefix of the current line. If the prefix is “f”, the indices are parsed to correctly store the format “int/int/int int/int/int int/int/int”. Finally the “data” variables (indices, normals, uvs, positions) are filled with the according values.
- **Playable**
  Player can be moved around the scene and jump up and down (see Controls below).
  The collision detection is simulated (see Collision-Detection below).

- **Advanced Gameplay**
  The player has to collect 3 pizzas and get to the oven before the time runs out.
  To check if each pizza has been collected, a boolean is updated (e.g. pizzaEaten1). Only if all booleans are true, the game can be won. Additionally, there is a time limit, which is counted in whole seconds (currently 50).
- **Min. 60 FPS and Framerate Independence**
  Game runs at 60 fps. Player movement, jumping, movement of waves, moving objects and particles is framerate independent (using delta).
- **Win/Lose Condition**
  The player has to collect all 3 Pizzas and get to the oven in a certain amount of time to win. If he falls into the lava or takes to long, he loses.
  Each frame, the checkWin() method is called, which checks if the player is to low (fell in lava), all pizzas have been eaten, if he is close enough to the goal and if there is still time left.
  NOTE: Information about fps, eaten Pizzas and winning /losing the game is only printed to the console!
- **Intuitive Controls**
  The player can move with the common WASD Keys (see Controls below).
- **Intuitive Camera**
  Camera can be moved by mouse-movement in all directions.
- **Illumination Model**
  All objects have material and normal vectors, using the header- and cpp-files provided in the template. A point and directional light are defined.
- **Textures**
  All objects are textured, using the provided files. Mipmapping and trilinear filtering are enabled directly after loading texture.
- **Moving Objects**
  There are 3 moving objects. To move moving geometry, the objects are translated in the render loop, based on the sinus of the current time.
- **Documentation**
  You are reading it right now! :)
- **Adjustable Parameters**
  All required parameters can be adjusted in the settings.ini file from the template. The values are retrieved in the main function using the INIReader.

Gameplay-Optional:

- **Collision Detection**
  Collisions are detected between player and the lava / objects to jump on. I implemented it myself (no Bullet or PhysX), since it is no advanced physics. Every object (except the oven) has its own bounding box, stored in the staticCollisionObjects or movingCollisionObjects list. Each frame I check if the player is colliding with any of those objects or not and call the updateJump() method with the according collides-boolean. This method makes the player fall, if he is not colliding with anything (or jump if he is jumping) .

Current Controls:

WASD: Movement
Space: Jump (Up in DebugMode)
L-Shift: Down in DebugMode
Mouse: Camera view direction
Mouse-Wheel: Zoom
Left Mouse Button: Collect Pizza/ get Pizza out of the oven
Esc: Quit
F1: Toggle wire-frame mode
F2: Toggle back-face culling
F3: Toggle SkyBox
F4: toggle DebugMode (free camera-movement)

Additional Libraries

The 2 libraries assimp and bullet are included, but never used.

assimp (https://github.com/assimp/assimp)
was intended to load models, but I decided to write my own ModelLoader (loading takes place in Geometry.cpp).

bullet (https://github.com/bulletphysics/bullet3/releases)
was intended for collision detection, but since the only collision is between the player and one object/lava, I decided to simulate it (checkCollision() in main).