Assignment 3-2 – Crossy Road Game

Eileen Lucie Clementine MATHEY (DeptCSE, 49004693, emathey@postech.ac.kr)
Megan Tong LEE (DeptCSE, 49004702, meganlee@postech.ac.kr)
CSED451: Computer Graphics

Functionality

Our program is a rendition of the popular arcade game, Crossy Road. The character, represented by the Character class, moves within a map containing roads, rivers, and obstacles like trees. The player's objective is to successfully reach the flag at the end of the map by navigating through the roads and rivers without colliding with moving vehicles or falling into the water.

You can advance the player left, right, up, and down using the left, right, up, and down arrow keys respectively. You can also enable the all-pass function (cheat mode) by pressing P. Pressing V allows the player to switch between third person, first person, side and above views, while pressing R allows the player to toggle between different rendering modes such as Color and Wireframe with or without depth removal. If the player’s character wins by reaching the end flag, they will be returned to the starting position where they can continue the game. If they die, they are also brought back to the beginning, but their score is reinitialized. The player can also translate the camera along the x, y and z axis using W (forward) A (left) S (backwards) D (right) W (Up) and Space (Down) keys, as well as reinitialize its position with the C key.

Our programming environment utilizes the following setup:

- Visual Studio 2019
- freeglut 3.0.0-2
- GLEW 2.1.0
- GLM 0.9.9.7

Design & Implementation

1. *Utility classes*

   We have created several “utility classes” (or Managers) that allow us to manage our resources, all of them follow the Singleton design pattern of Object-Oriented Programming. One can use the [Object]Manager by using the function get[Object], which will check the already loaded objects and either return it if already present or load it. This allows to not load multiple times the same object and is more efficient memory and computation wise.
   First of all, the MeshManager class is used to load and store meshes, whether they come from a file (cube.obj) or were constructed in the program (the ground, see part 2). The ShaderManager loads shaders from a name, a fragment shader and a vertex shader. We also have a TextureManager but it is not used in this assignment.

2. *Rendering*

   We now have two new classes for rendering our 3D objects without using the fixed OpenGL pipeline:

   Renderable.cpp - represents an object that can be rendered in the 3D scene. It contains information about the object's mesh, shader, global color, and transformation matrix. Here, custom shaders (described in part 4.) are used, instead of the fixed pipeline of OpenGL. The shaderId member variable is set to the ID of a shader program retrieved from the ShaderManager. The transformation matrix is used to position, scale, and rotate the object in the 3D scene. The Renderable class has two constructors: one that takes a color and a path to a mesh, and another that takes a shared pointer to a mesh. Both constructors initialize the mesh, shader, color, and transformation matrix. When a GameObject is created or updated, the transformation matrix associated with its renderable is set to its current position and its size. The draw method activates the shader, sends the projection/view/model matrix and color information to it, binds the vertex array, and then finally draws the elements. If the object has any child renderables, it draws those as well.

Renderer.cpp - responsible for managing the rendering process. The buildWorld method is used to construct the ground for the game world. It creates a series of vertices and indices that represent the ground depending on the position of the paths of the Game, and then creates a Renderable from those. The drawScene method is used to draw the entire scene. It clears the screen, draws the ground, and then draws the roads, player, powerups, trees, and end flag.

3. *Wireframe Implementation*

The Renderer class also allows to toggle between multiple rendering modes, such as Color, Wireframe and Wireframe with hidden face removal. The toggleRenderingMode function alternates between those modes and enables (or disables) OpenGL parameters such as the way the polygons are drawn (through glPolygonMode), the depth testing or the face culling.

Color mode – glPolygonMode is set to GL_FILL and depth testing is active (with culling disabled). This allows for a clean 3D look.

Wireframe mode – glPolygonMode is set to GL_LINE and depth testing is unactive (with culling disabled). This allows the user to see every polygon that was drawn on screen using lines.

Wireframe mode – glPolygonMode is set to GL_FILL and depth testing is active (with culling disabled). This allows the user to see the polygons that were drawn on screen using lines, but removing the ones that would be hidden by other faces on the same object.

4. *Viewing Implementation*

The viewing process is mostly handled by Camera.cpp, which represents a camera in 3D. The camera's position, view mode, and projection matrix are defined here, and the camera can be in one of four view modes: third person, first person, side view, or above view.

The toggleViewMode method changes the camera's view mode and updates its position and projection matrix accordingly. The view mode cycles through third person, first person, side view, and above view each time toggleViewMode is called.

The moveCamera method moves the camera by a given amount in the x, y, and z directions and is called to move the camera according to character movement. There is also the moveCameraOnInput method, which allows the player to translate the camera without being limited by character position. The updateCamera method updates the camera's view matrix based on its current view mode and position. The rotateCamera method rotates the camera around a given axis by a given angle.

The projectionMatrix is used to transform the 3D scene into 2D for rendering on the screen. The viewMatrix is used to transform the 3D scene based on the camera's position and orientation. The glm::perspective function is used to create a perspective projection matrix (used in First/Third person view as well as Above view), and the glm::ortho function is used to create an orthographic projection matrix (used for Side view to mimic the original game's style).

The glm::lookAt function is used to create a view matrix. It takes the camera's position, the point the camera is looking at, and the up vector, and returns a view matrix that represents a camera with those properties. The glm::translate and glm::rotate functions are used to move and rotate the camera, respectively.

5. *Shaders*

We implemented these vertex and fragment shaders:

flatVertex.glsl – used for all loaded renderables (who don't have their actual color information stored in their vertices). It takes in several attributes for each vertex: its position, color, texture coordinates, and normal vector. It also takes in uniform variables for the global color of the renderable and the model, view, projection matrices. It calculates the position of the vertex in screen space, as well as its color, which will be sent to the

Fragment shader after. The color is the result of the multiplication of a shadow color (calculated by using the dot product of a lightDirection vector with the normal) with the global color of the renderable. This (in addition to the structure of loaded meshes) results in a flat shading effect, where each face of the renderable has a single color.

colorVertex.glsl – used for manually constructed meshes for already have the correct color information stores in their vertices and otherwise very similar to colorVertex.glsl.

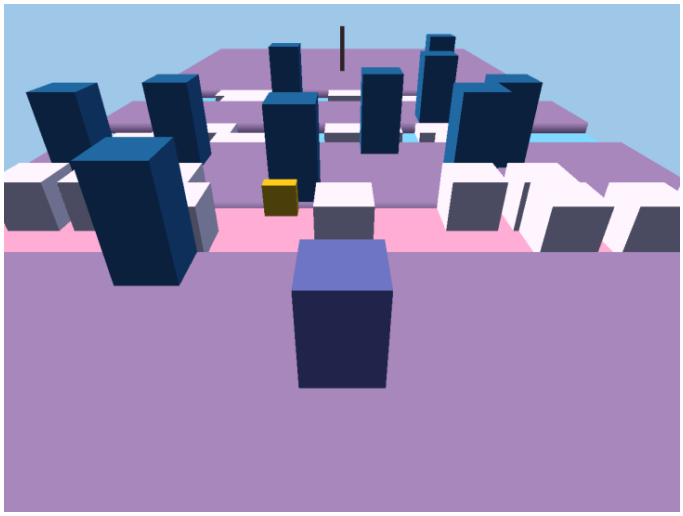
flatFragment.glsl – takes in the color of each vertex (interpolated across the shape) and sets the color of the pixel to this color.

Program Execution Instructions

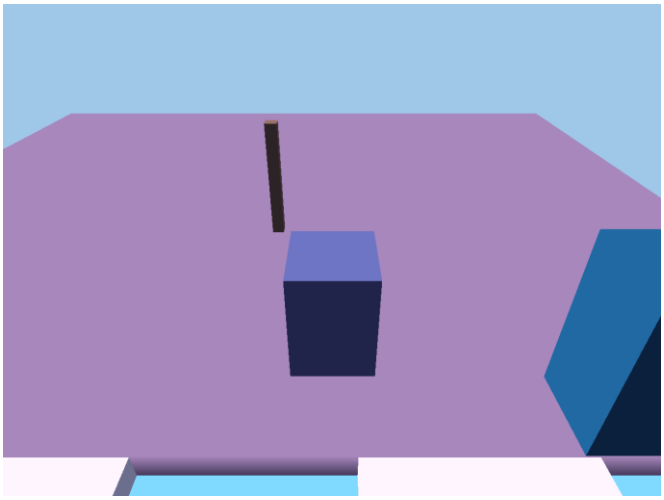
Please extract the files in our zip file into a new folder, and then navigate to 'OpenGL Setup > OpenGL Setup.vcxproj' and open it in Visual Studio 2019. Running the program will cause a new graphical interface to pop up, where you can start to play the game.

Examples

This is the starting display of our game where the default view is third person, and the default rendering mode is wire-frame without hidden line removal.



The player's goal is to reach the pole on the other end of the map.

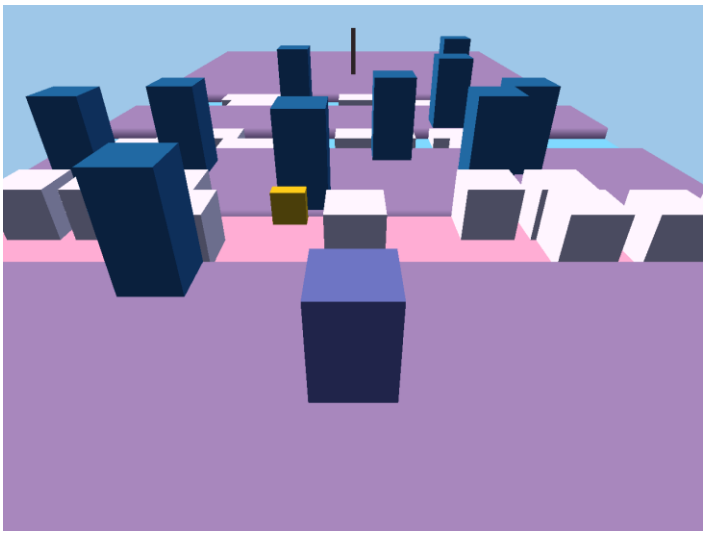


The player's score is calculated based on the Z distance they walk before dying. When reaching the end flag, the player is brought back to the beginning and they can start accumulating score again.

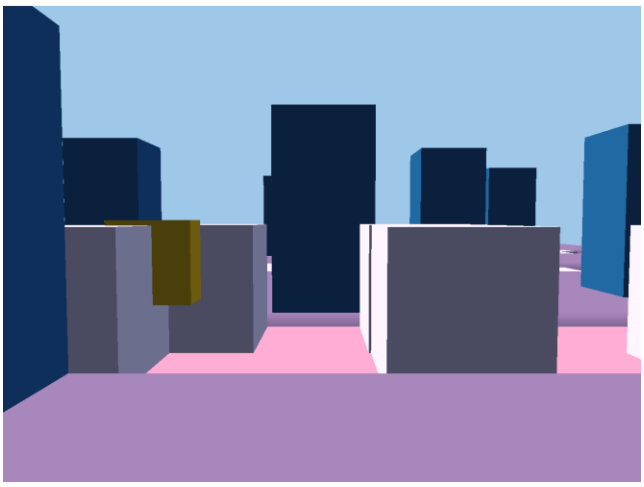
```
Current viewing mode: Third Person
Congratulations! You have reached the top! Current score is 19!
Congratulations! You have reached the top! Current score is 38!
Congratulations! You have reached the top! Current score is 57!
Oh no, your character has died :( Your score was 60!
Oh no, your character has died :( Your score was 4!
```

The player can press V to change between different viewing modes: third person, first person, side view, above view.

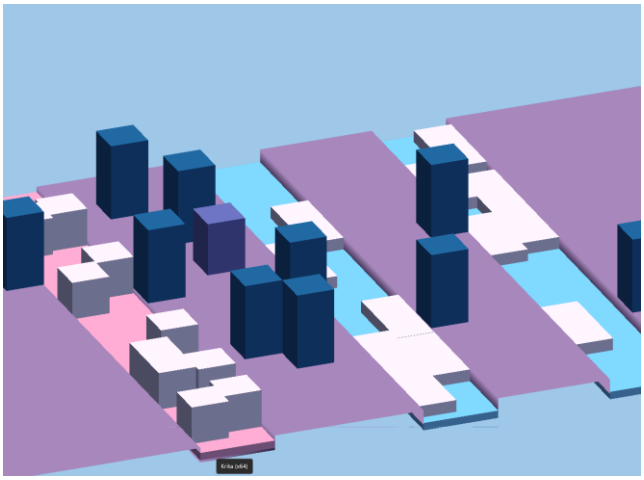
Third person view:



First person view:



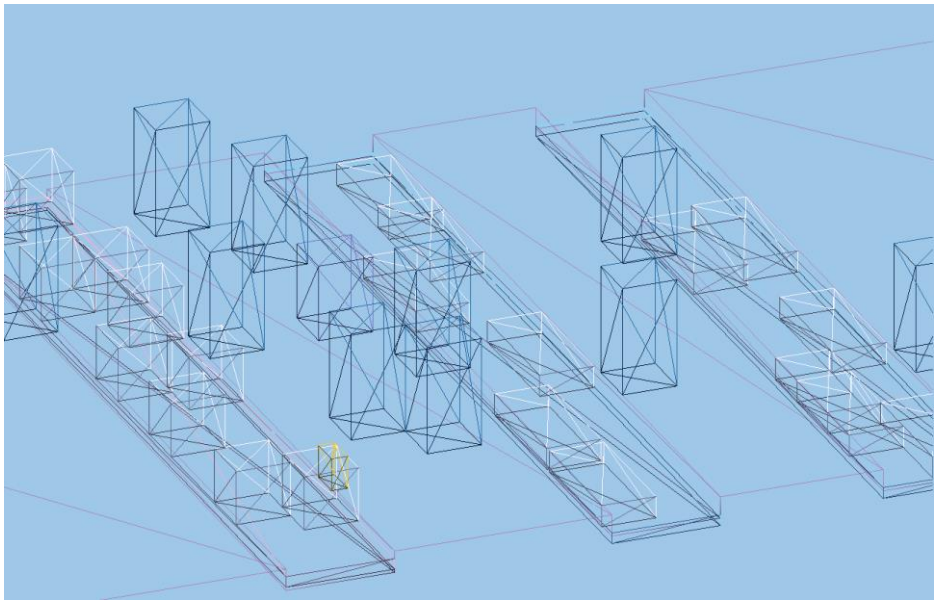
Side view:



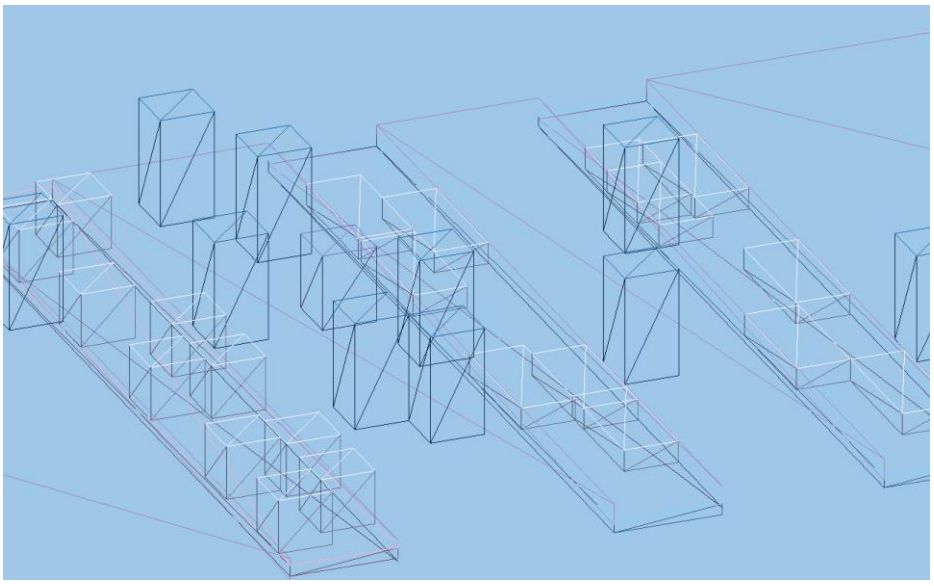
Above view:

Pressing R will alternate between the default color rendering, the wire-frame rendering without hidden line removal, and the wire-frame rendering with hidden line removal (photos with side view):

Wireframe (without hidden line removal):



Wireframe (with hidden line removal):



Discussion

Implementing custom shaders in OpenGL, as opposed to using the fixed pipeline had its own challenges: it was a complex process to understand and implement GLSL and since shaders run on GPU and not CPU, debugging must be done through observing the visual glitches instead of traditional debugging. We also could no longer use the fixed functionality provided by the OpenGL pipeline such as automatic transformation and lighting.

We also decided to use a cube.obj mesh instead of creating ourselves an indexed cube object, which made us learn more about meshes and how to load their information. This was difficult because of the structure of the .obj file, as well as the fact that the vertices have multiple attributes to them. In fact, figuring out how to use the Vertex Array Object in this case was hard, especially since the debugging had to be done visually.

Directions for Improvements

If given more time, we would work on optimizing the shader. One such way would be by pre-calculating the normalized light directions and passing it through the shader as a uniform variable, and this would save on the cost of normalizing the light vector for every vertex. In future assignments, we will also need to make the mesh loader better as for now the indexation is not optimized (which leads to the flat look of our loaded meshes in comparison with that of the ground, a generated mesh with good indexation). This is not really a problem when working with cubes, but as we saw when trying to load complex meshes, it causes the faces to all have the same color and not smoothly transition with each other, which is not what we want. We'd finally like to make our Renderables actually hierarchical, with the children being transformed along with their parent.

Conclusion

This assignment provided us an opportunity to delve into the complexities and intricacies of implementing custom shaders and managing mesh data, allowing us to move away from the fixed pipeline of OpenGL towards a more flexible, but also more demanding, shader-based approach.

Contributions

Eileen Lucie Clementine MATHEY – 50%

Megan Tong LEE – 50%