**CSED451-01 Computer Graphics (Spring 2023) Assignment #3-2**

우아팀

Heewoo Lee (20180113; hwlee00), Xin Cui (20212208; xincui)

**1. Introduction of the Game**

This 3D adaptation of the "Wind Runner" game is developed with C++ and OpenGL Shading Language (GLSL). In this game, players control a character with three motion poses, capable of changing three view modes (a third-person perspective, a third-person orthogonal view, and a first-person perspective view), as well as jumping to collect stars for scoring points while avoiding obstacles like fireballs, holes, and mushrooms. In the current development stage, wireframe rendering is utilized for object visualization.

**1.1. User Interface**

The game features a user-friendly 3D graphical user interface (GUI) with 3D objects, including a character (in 3 poses), terrain with varying heights, holes, high-speed fireballs, patterned stars, and a visualization of the player's score. It provides a forward 3D perspective and a horizontal 2D third-person view (Figure 1).

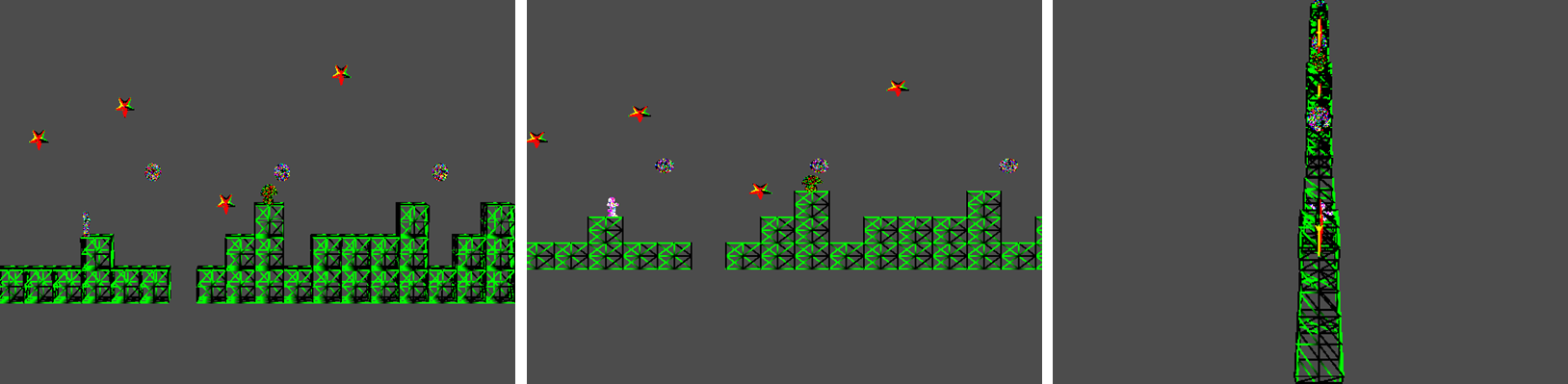


Figure 1. The user interface of the game.

**1.2. Functionality**

This game enables players to manipulate a character exhibiting three distinct motion poses and switch between three viewing modes (a third-person perspective, a third-person orthogonal view, and a first-person perspective view). The character can jump to collect stars for increasing scores while avoiding obstacles such as fireballs, holes, and mushrooms. Collision detection is implemented to identify encounters with these obstacles. The game's map, consisting of terrains, star patterns, and fireballs, is generated randomly to maintain variety and challenge.

**1.3. How to Play**

Instruction for how to play the 3D game is shown as follows.

1. Press "1" to switch the game perspective to a third-person perspective view.
2. Press "2" to switch the game perspective to a third-person orthogonal view.
3. Press "3" to switch the game perspective to a first-person perspective view.
4. Press the spacebar to make the character jump over holes or onto higher terrains. Please note that the character can only jump once during a jump.
5. Press the spacebar to make the character have a small jump for either stepping on or jumping over mushroom obstacles when the character is near the mushrooms. If the character passes a mushroom obstacle from the side without a jump, the game will end.
6. Press the spacebar to make the character jump or keep going to avoid fireballs. If the character touches a fireball, the game will end.
7. Press the spacebar to make the character jump or keep going to collect stars. Each star collected will increase your score by 5 points.
8. End the game by pressing the "q", "Q", or "esc" buttons on your keyboard.
9. The final score will be displayed on the screen.
10. Have fun playing the 3D game.

**2. Design and Implementation**

**2.1. Programming Environment**

* C++
* OpenGL Shading Language (GLSL)
* Visual Studio 2019

**2.2. Implementation**

**2.2.1. Overview**

In this program, a combination of source files, header files, and GLSL shader files is employed. The entry point of the program is provided by "main.cpp" where the primary functions are located. The base class for game objects, along with the structure and functionality of 3D models, is defined in "Model.cpp" "Model.h" "Object.cpp" and "Object.h" The core game logic, encompassing the implementation of the game loop, event handling, and rendering, is contained within "Game.cpp" and "Game.h". Furthermore, "Angel.h", "mat.h" and "vec.h" encompass several OpenGL-related libraries and provide functionality for working with matrices and vectors. Lastly, "vshader.glsl" and "fshader.glsl" establish the vertex and fragment shaders utilized in the rendering pipeline, while "InitShader.cpp" comprises functions for loading, compiling, and linking GLSL shader programs.

**2.2.2. Implementation Details**

**(1) Shader**

The shaders are implemented with shaders using the GLSL. Vertex and fragment shaders, stored in separate files "vshader.glsl" and "fshader.glsl", are read, compiled, and linked into a GLSL program. Game objects are subsequently rendered using these shaders, with the vertex and fragment shaders dictating their positions and colors, respectively. In particular, shaders are initialized in "InitShader.cpp" through the functions "readShaderSource()" and "InitShader()". The former reads the shader files' source code (vertex and fragment shaders) and returns a NULL-terminated string containing the code. The latter creates a GLSL program object from the vertex and fragment shader files, compiling, linking, and attaching the shaders to the program object before returning the created GLSL program's ID. Additionally, the fragment shader file "fshader.glsl" is designed to set the output color (FragColor) to a constant yellow color (RGBA: 1.0, 1.0, 0.0, 1.0). The vertex shader file "vshader.glsl" calculates the final vertex position (gl\_Position) by multiplying the input vertex position (vPosition) with the model view and projection matrices. Lastly, the "Game::shaderInit()" function in "Game.cpp" initializes the shaders and acquires attribute and uniform variable locations from them.

**(2) Model Handling**

* **Model Loading:** A custom "Model" class is designed to manage the loading and storage of each object file through the Model::load function. This function handles the loading process for each model and utilizes GLuint arrays to store the IDs of the models. Within the Model::load function, a custom OBJ file loader is implemented to extract vertex positions, face vertex indices, and other pertinent information from the file. Subsequently, the vertex positions and face vertex indices are bound to Vertex Buffer Objects (VBOs) and Element Buffer Objects (EBOs), respectively. The VBO ID, EBO ID, and the number of face indices are stored in an array for future reference.
* **Model Transform:** The model transform function primarily manipulates the model's transformation matrix in the "Object" class, which is a 4x4 matrix representing the model-view transformation. This matrix is linked to the shader program's uniform variable, 'model\_view'. As the original model loaded from the OBJ file is excessively large, an initial scale-down factor is multiplied by the transformation matrix. Additionally, for objects other than the character that need to be drawn multiple times, glPushMatrix and glPopMatrix have been replaced with matrix stack push and pop operations.

**(2) View Mode**

Three view modes are implemented, comprising a third-person perspective, a third-person orthogonal view, and a first-person perspective view. These modes can be switched using keyboard input, with the "currentViewMode" variable determining the active mode. The "Perspective" and "Ortho" functions are employed to handle different view modes and generate the corresponding projection matrices. The update() function is responsible for modifying the projection matrix based on the active view mode. In the third-person perspective mode, the Perspective() function is utilized to create a perspective projection matrix with a 20-degree field of view. For the third-person orthographic mode, the Ortho() function is employed to generate an orthographic projection matrix, defining a box-shaped viewing volume. Finally, in the first-person perspective mode, a perspective projection matrix with a narrower field of view is used (Figure 2).

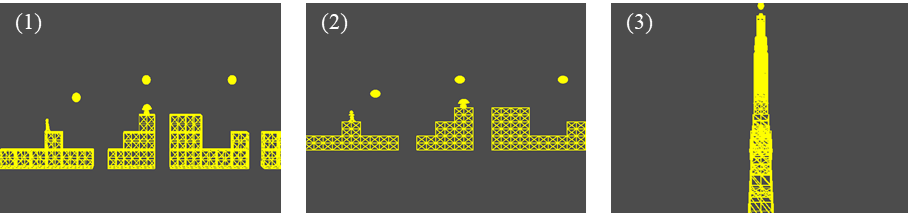


Figure 2. View mode. (1) third-person perspective view, (2) third-person ortho view, (3) first-person perspective view.

(3) Map Generation

* **Terrain and Mushroom:** The terrain and mushroom are generated in "Cube" class, which consists of cubes arranged at different levels. The "randomLevel()" fucntion generates a random sequence of levels for the cubes. The draw() method renders the cubes and the mushrooms on top of them. To generate the game map, the Cube class defines an array called "level", which stores the height of each cube in the map. The randomLevel() function fills this array with random numbers, with 0 representing no cube, and other numbers representing the height of the cube. The "draw()" function iterates through the levels array and renders cubes and mushrooms accordingly. The "mushroom->draw()" function is called if there is a cube at the current position and the mushroom is marked as drawable.
* **Fireball and Star:** Take the fireball as an example, where the star is similar. The fireballs are generated and drawn with the “Fireball” class . “initPosition()” function is used to set eh initial position of the fireball. A random fireball position is generated by “randomGeo()” by calculating a random x position, and a random y position, and setting the z position to 0. The function then adds the generated position to the firePos deque. “draw()” function draws the fireballs in their current positions. It updates the fireball positions based on their speed and checks if any fireball has gone out of the viewport. If a fireball goes out of the viewport, it is removed from the “firePos” deque, and a new fireball is generated using the randomGen() function.

**(4) Character Animation**

* Jump: The jump function is defined in the “Character” class called “setJumping”, which is responsible for initiating the jumping action of the character. When the character is not currently in a jumping or falling state, and its position is close to the baseline, the state variable is set to 1, which represents the jumping state. If a mushroom is detected ahead, the character will perform a lower jump with half the usual height. Otherwise, the character will perform a regular jump with a predetermined height.

**(5) Collision Detection**

* The collision detection is handled with “isCollision()”,which checks whether the character collides with other objects such as fireballs. The bounding box of the character using the left, right, top, and bottom edges of the character’s position, is retrieved from the game instance. Overlapping edges between the character’s and object’s bounding boxes. If there's an overlap, the function returns true, indicating a collision. Otherwise, it returns false. The “gameOverChecker()”utilizes the “isCollision()” function to check for collisions between the character and objects. If a collision is detected, the game state is set to 'not on game' by calling “game->setOnGame(false)”.
* When the character collides with the stars, the star vanished (Figure 3).

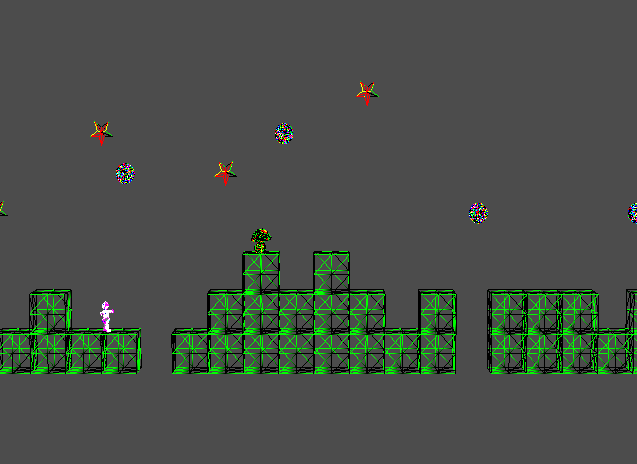


Figure 3. Star vanish effect (illustrated)

**(6) New Feature**

A fancy function is added. The fragment shader is used to set the output color based on the input color and alpha value. The color is added when loading the model. Using function glUniform1d, we was able to transfer alpha value into fragment shader.

**3. Discussion**

During the implementation process, we faced challenges aligning objects due to their varying locations. To address this, we developed code to extract each model's center and size information by storing the minimum and maximum vertex positions. We then aligned the models by initializing their positions to (0,0,0) in the world space based on the extracted model center (Figure 4).

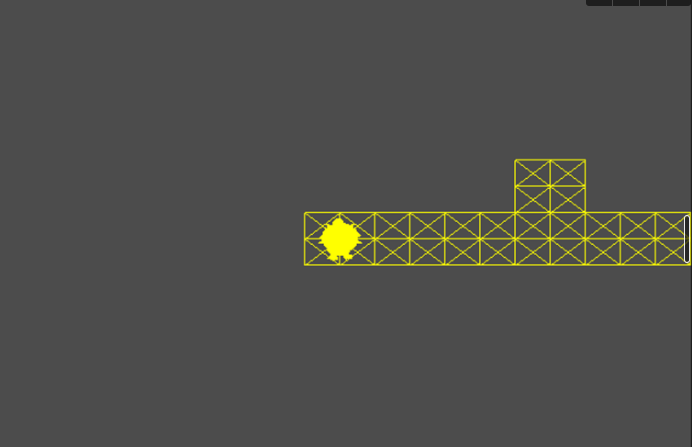


Figure 4. Alignment of all models at the center

**4. Conclusion**

A 3D running game has been implemented using C++ and GLSL, offering a comprehensive array of features such as character animation, switchable view modes, randomized terrain with varying heights, and a jump function for avoiding obstacles or collecting stars through collision detection. In conclusion, we have successfully developed and demonstrated this 3D game utilizing GLSL, in alignment with our predefined objectives.

**5. Directions for improvements**

The interaction design, UX/UI design, and playability of the current game could be enhanced in future development. A dynamic camera that adapts to the character, along with keyboard-controlled view modes, and the ability to zoom in and out, can be implemented to make the 3D game more intuitive. Furthermore, improvements to the user interface design, such as adding a pause menu and settings menu, can be developed to elevate the overall UX/UI experience. Lastly, the complexity of the terrain can be increased by introducing multiple levels and diverse obstacles, which can augment the game's playability.

**References**

<https://thecherno.com/opengl>

<https://chat.openai.com/chat>

**Note**

Only new features different from previous assignments updated in "Assignment 3-2" are reported.