Individual Project Report

Computer Graphics CS7GV6

Yahong Zhu 21333370

3D Models in the Game

Several 3D models were added to create a scene in the game development, they were all textured by diffuse maps and some of them are specular objects which have a specular map to help its glow effect with lighting in the scene. Models were downloaded mostly from Sketchfab and assembled manually.

Fig.1 Terrain model

Fig.2 Character model

The first model is combined with different objects to create the basic terrain of the game. It includes a basic plane, plants, stones, steps, pillars, and a small pavilion as an obj format and the material data were stored in an mtl file. The final outline is shown in Fig.1.

The second model is the main character in the game which can be controlled by the player. Same as the terrain model, the character model was textured by a diffuse map. Additionally, several joints and bones were created by the artist attached with a running animation and stored in a dae format model file which allowed us to load the animation in the code and implement skeleton animation afterward in our game. The model was shown in Fig.2 in a running gesture.

Fig.3 Lantern model

Fig.4 Barrel model

The third model in the scene is a lantern with both diffuse map and specular map shown in Fig.3. It is located near the front of the steps and shows a shiny effect based on the light source and the camera view. The last model used is the barrel shown in Fig.4 which rolls down from the

steps. This model was rendered multiple times to create several rolling objects in the scene and the player is going to control the character jump over the rolling barrels to avoid losing scores. Both models are stored in obj format as well. Finally, we import our models using the assimp library to our scene.

Skybox was applied to the scene as well. In resources, we also added a skybox folder which contains all six jpg format photos for 6 faces of the skybox.

Techniques in the Game

1) Keyboard Control

The game allows players to control the character using the keyboard. Pressing A and D enable players to change the direction the character faces, and W and S are for controlling the character moving forward and backward. In addition, pressing the SPACE key allows players to jump in the game to avoid hitting by the barrel and losing scores.

This feature was implemented in the processInput function which checks the key pressed in the keyboard and changes the parameters for controlling the transformation of the character, for instance, current_rotate_speed, translat_step2, current_speed, jump_upward_speed, etc. While implementing jump features for the character, we also need to check if it is on the sky since we want our character back on the land to simulate the gravity, so it can't stay on the sky after pressing the SPACE key. We use these parameters to calculate the position on the x-axis, y-axis and z-axis and apply these variables in the transformation matrix which sets up the translation, rotation, and scaling of the model.

2) Transformation

Transformations were applied to most of the models in the scene to control the position, rotation, scaling as well as controlling the movement of the character and the rolling translation of the barrels through time. We used GLM library for the math calculation of the transformation matrix. First, we initialized a 4×4 matrix and set all the values of diagonal elements equal to one. Then, we applied the rotation, scaling and translation by glm::rotate, glm::scale and glm::translate. After setting all the transformations of the model, we applied this transformation matrix by calling setMat4 function in Shader class to set the uniform model matrix in the vertex shader. The result can be seen in Fig.5 on the right with rolling barrels and the moving character.

Fig.5 Transformation

3) Hierarchy and Skeleton Animation

The movement of arms and elbows rotating on the body of the character is hierarchy transformation, as well as the movement of legs and feet. The arms can be considered as the children of the body, and the elbows are the children of the arms, then we have a basic

hierarchical structure of the model. The hierarchical structure is vital for the skeleton animation since this transformation can be seen as a part of the skeleton animation.

We started skeleton animation by creating customized animation on the model. This was part of the modeling work done by the artist who create the model. We downloaded this model and imported it into the scene using assimp library. Then we loaded all the meshes in the model and then, processing all the vertices, indices, normals, texture coordinates and bone data (ids, weights) in loadModel function. A bone structure includes its id, name, offset and children. The last step of this function was to create a hierarchy by calling readSkeleton function with all the bones information. We processed bones information recursively from the node of the model and if all the bones were successfully processed, the function would return true. Then, we loaded the animation by loadAnimation function. We read the duration and the transformation data of the bones in this function and loaded positions, rotations, and scales for each bone by tracking keyframes of the model. After that, we were able to get the current pose of each frame and load bones transformation into the vertex shader in shader_character.h. In the vertex shader, we calculated the exact bone transformation by multiplying the transformation of each bone and its weights on each axis. The vertex positions were calculated by multiplying the bones transformation matrix and the model matrix. We passed the vertex positions to the fragment shader for other calculations. Finally, we passed all the information needed to render the model into the getPose function with animation, skeleton, time and currentPose information, etc. That function allows us to finally render the pose in the game loop by every frame.

We processed the texture using createTexture function to render the model in the scene with proper texture in the diffuse map. The result of this skeleton animation is the character in the scene running with arms and legs rotating in a certain range around the root node.

4) Diffuse and Specular Light

The basic light for lighting up the scene is the ambient light which can be easily calculated in the fragment shader using the ambient values and the texture of the objects. It doesn't give any special effect on the objects but builds the foundation for diffuse lighting and specular lighting.

Diffuse lighting and specular lighting are added to the scene. We set up the light position at (1.0f, -0.35f, 2.0f) at the beginning as the light source of the diffuse light and passed that into our fragment shader in lantern.fs. Then in the vertex shader lantern.vs, we first calculated the fragment's position by multiplying the model matrix and the vertex position attribute and passed it to the fragment shader as well as the normal attribute. All the variables required were set we can now start to calculate the diffuse light in the fragment shader. We first normalized the normal and then use the light position and fragment position to get the light direction vector which represent light's direction. We normalized it to get the direction only. Finally,


Fig.6 Diffuse lighting result

the diffuse lighting was calculated by the dot product of the normal of the object and the light direction vector. If the value was less than zero, we ignore that value and set it zero as the result. We applied the diffuse light on the object by multiplying the diffuse light vector and the light color variable. Therefore, the effect of a diffuse light would allow us to see brighter surfaces when more object faces were facing the light source. An example is shown in Fig.6 as part of the hat of the main character was brighter when it was facing the light source.

Specular lighting is more or less similar to diffuse light. But the only different thing is the light intensity relies on the view direction as well e.g., the camera direction. The calculation is based on the reflective properties of the material so we would like to set shininess of the material first. We created a material structure in the fragment shader that included diffuse, specular and the shininess. The next step was to get the view direction by the view position which represents the camera position and the fragment position. We called the reflect function to calculate the reflection direction. Then, we calculated the specular light using shininess to decide how to shine our objects would be. We also added the specular map to the objects and calculate the specular color by the specular map. The specular map was stored in the green channel, but we would use the RGB channel to have a fun result. Finally, we added the ambient light, the diffuse light, and the specular light together to get the ultimate effect of lighting in the scene which is shown in Fig.7. As we can see, the specular light on the stone on the left bottom is multi-colored since we use all three channels of the specular map. And the lights on the tree reflect mostly a green color.


Fig.7 Specular lighting result

5) Camera View

There are two camera views in the game, the larger view is a camera fly with the main character and the smaller view located on the top right corner is the top-down view of the other camera.

To split the whole screen we used the glViewport(0, 0, 1200, 900) function in the game loop before rendering anything to set up the position of the view and the size of the view. In our game, the bigger view was set as 1200×900 as the same large as the whole screen. Then, we rendered all the models and setting the camera view by initializing a camera and set the view matrix, the projection matrix. We passed these two matrices to the vertex shader to combine the camera view with the model shown on the screen by calculating them with the transformation matrix of objects afterward at a certain distance. The camera class has several attributes to set up a camera in the scene. Except for the regular settings, we added distance_from_user to represent the distance between the character and the camera. In this way, the camera was able to follow the character at a

certain distance. And the other attribute in addition, was the angle_around_user. By changing the value of it, we can easily make the camera rotate with the character. We can also rotate the main camera with our mouse by mouse_callback function changing the offset in x-axis and y-axis of the camera. After all the camera settings were done, we started to render the first view by rendering all our models. Then, we can get a larger view on the screen.

The smaller view was implemented similarly to the larger one. After the first rendering of the larger view, we set the second view by calling the glViewport(800, 600, 400, 300) function again. We set the smaller top-down view on the top right corner of the screen with a size of 400×300. The second camera was set by setting different Yaw and Pitch values to get the new angle of the camera direction. We rendered all objects in the second viewport with the second camera, then we got the smaller view on the screen. It reflected the movement of models in a top-down angle which is shown in Fig.8.

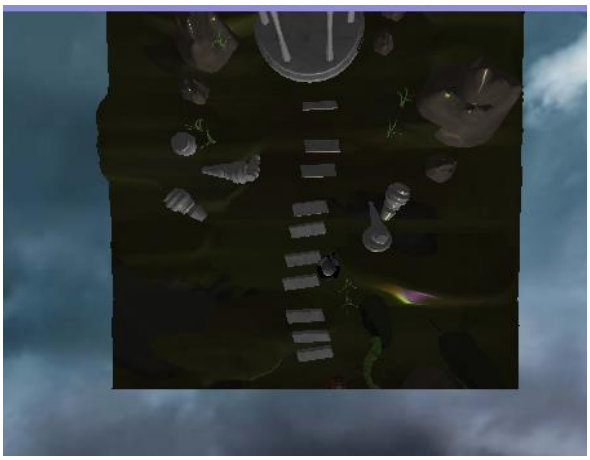

Fig.8 Top-down camera view

6) Skybox

To make our game more delicate, we created a skybox to simulate the sky and the environment around our models.

A skybox is a cube rendered with six individual 2D textures on the six faces. We created this cube map by creating the textures first, then calling glTexImage2D six times on six different targets. After that, we read the textures files from the folder by calling stbi_load function. Then, we set texture parameters by glTexParameteri function as always. We did this process in the loadCubemap function and return the texture Id. We also need a new shader for skybox, so we created skybox.vs and skybox.fs in our project. Particularly, we used samplerCube type uniform for the textures of the skybox in the fragment shader. Finally, we disabled the depth writing and were able to draw the skybox. The one last thing of creating a skybox was removing the translation part of the view matrix since it translates our skybox as well while moving our character. The rotation would not affect the skybox because we want our skybox to rotate with the camera. The result is shown in Fig.9.

Fig.9 Skybox

7) Scoring

The last element added in the game was the score calculated. This was implemented by two techniques, simple collision detection, and text rendering.

When the character was hit by the rolling barrels, the score was lost. To detect the collision, we calculated the Euclidean distance between the position of the character and the position of the barrel. If the distance is less than 0.3, we decided the character was hit by the barrel and the score would be subtracted 1. We calculated this in the CheckCollision function, and it returned a boolean value that indicates the result.

To show the score on the screen, we need to use freetype library which helps us render texts. Then we added the font file to the resources folder and load the font file into the initialized face. We defined the font size from the face by FT_Set_Pixel_Sizes function. After that, FT_Set_Pixel_Sizes function was called in a loop to load all 128 characters in the face. We generated texture for the font and set it by glTexParameteri. At last, we stored the characters for later use. To render the text, we need to set up a new shader for the text as well. In the vertex shader we calculate the position by multiplying the coordinates with a projection matrix and passed the texture coordinates to the fragment shader. In the fragment shader, mono-colored bitmap image of the glyph and a color uniform for adjusting the text's final color was used. We sampled the color value of the bitmap texture and multiplied the RGB colors by the textColor uniform to change the text color. An orthographic projection matrix was used since we don't need perspective while rendering text. Finally, we created the VBO and VAO for rendering the outline of the text. We rendered the string by creating the function RenderText and called that function in our game loop to show the score in every frame. The result can be seen in Fig.10.

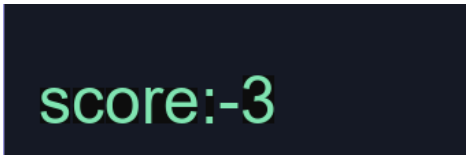

Fig.10 Text rendering

Final Scene

Fig.11 Final Scene

The final scene is shown in Fig.11. More details were presented in the video. The YouTube video link is available here: https://youtu.be/3bJD4PC0lbw. We can eventually use the keyboard to control the character playing that 3D game and change the view by moving the mouse.

Future Work

This game can be improved by adding treasures in the scene for the player to collect and earn their scores. This is pretty similar to the implementation of the barrel, but accurate collision detection and reaction are needed in this game as well. After the player collects the treasures, the treasures need to disappear from the scene. More elements could be added to make the game look delicate as well e.g., ladder for climbing, animals or monsters, water pools, point lights, etc.