Computer Graphics Assignment

Julian Heng (19473701)

October 27, 2019

Contents

Game Workflow

game.c

`game.c` is responsible for initialising GLAD as well as handle all of the game's logic. It has a `Backend` struct that is passed from functions to functions which keeps track of all the existing models, game settings, compiled shaders, loaded textures and the camera. After initialising GLAD and construct all the models, it will enter the main program loop where each frame of the game is calculated and drawn until the user quits the game.

Lighting is handled in `game.c`. The function `setupShader` would setup which lighting to use depending on the game state. There are 3 states: player has no torch, player has torch or turn on global illumination. The way to change how lighting is done, we set the variables in the light struct within the shader program. The light struct contains 3 variables: ambient, diffuse and specular. If we wanted to have global illumination, all 3 variables are set to maximum. If the player has a torch, we set a higher value to all 3 variables than what we would've set if the player doesn't have a torch.

Collision detection is handled by the function `checkHitBox`. It takes in a `Backend` struct, a position vector and the distance. It would exit pre-emptively if the player has already lost, but would check if the distance between the camera and the position vector is less than the provided distance.

Switching between orthographic and perspective view is simple as it's a difference of which function to call. Pressing P would toggle the view and would call either `glm_perspective` or `glm_ortho`.

camera.c

`camera.c` is responsible moving the player around the world as well as responding to mouse movements and zooming. Additionally, it is also handling the player jumping. The formula for the player jumping is simply an inverse parabola with roots at (0, 0) and (0, t), where t is the duration of the jump.

Each of the movements to the camera is simply just add or subtracting the change in time with the velocity of the camera. For forwards and backwards movement, we stop the player from floating around the map by removing the y component of the camera's front vector. Thus when we normalise the front vector, there's no y component and only the x and z components of the camera's position is changed.

box.c

`box.c` is responsible for storing a box's position, scaling, rotation, as well as being able to draw itself. In addition to this, combinations of boxes can be achieved by simply chaining them like a list and for any change to the box will also be applied to the boxes that are chained.

Each box stores both the model position and the world position. This is to ensure that we can still transform the model as a whole model without any deformation to other parts of the model. For example, rotation of a model would first have to translate the box to the model position, then apply the rotation, then translate to the world position. Without the set of model position, the model would deform.

Each box also stores a list of textures as well as a `Material` struct. The list of textures, while unutilized, supports for combining textures together, overlaying each other. The `Material` struct is also used to store the values set to the shader for that specific box. So we could set the box to be shiny or not. Colour is not used because we're using textures instead of object colour.

Within `box.c`, the `draw` function calls other functions like `setupShader`, `setupTexture` and `setupModelMatrix`. This is done such that we can override those functions if we ever need to. For example, for animating the sheep's legs or the wolf's tail would require additional transformation to one specific part of the model and not the entire model. As such, we would need to override `setupModelMatrix` such that we can rotate the model without affecting the entire model.

`setupShader` would set the different uniform variables within the shader program. All of the box's material properties is stored within the Material struct, so each box can have individual material. `setupTextures` binds the textures of a box to be drawn. `setupModelMatrix` would just create a transformation matrix to be passed to the shader program.

The order of transformation to a model is as follows:

1. Scale the box
2. Translate the box relative to the model
3. Rotate the box according to the rotation vector
4. Translate the box relative to the world

After drawing a box, it would invoke the same draw function to every box that is attached to the current box.

models.c

`models.c` is responsible to constructing models. There are 8 composite objects, excluding the ground. As such, manually coding out all the models would be a tedious task. As such, the macro `MAKE_MODEL` in `macros.h` would help reduce the amount of lines to create models. The macro takes in a `Box` pointer representing the root, another `Box` pointer for the model of interest, and 4 arrays that refers to the specification of each box which are the position and scaling, the texture, the material and the drawing function for that particular box. This forms a sort of map to be applied to each model. Each model is attached to the root box, and the root box is what get's added to the engine's model table to be easily referenced when drawing the models.

Ground model

The ground model is very simple. Root model is at (-50, -50) and all subsequent models is attached until it's reached (50, 50). The material for the ground would not be specular so a normal material will be assigned to it. The ground is scaled 5 times in the x and z dimensions but is shortened in the y dimension. The reason for looping the ground model is to loop the ground texture for a better appearance.

Tree model

Just like the ground model, the tree model is very simple. Loop to create the trunk and one box for the leaves. Within the game, we draw multiple trees by having the same model being drawn at different positions. As for material, it is using the same material as the ground.

Wolf model

A more complex composite model. It is made of 8 boxes and requires a specific function for drawing the tail. The head has 2 models as to display the face texture for the wolf as I do not know how to assign different texture just for one face of a model.

I decided to only animate the tail as to reduce the workload required. As such, animating the tail is simple as it's rotation is relative to where it is in the model and not to the world position. Thus a simple sin wave is used to rotate the tail. The only problem is that it clashes with the camera's rotation when holding the wolf. Each time the yaw of the camera changes, it would need to update the rotation of the model, which overrides the tail's animation and reset's it position.

Sheep model

Just like the wolf model, same structure with different scaling applied. The legs have an extra model to make the wool of the sheep attach to the legs as well, meaning that a leg is actually 2 models. More textures are also required, totalling at 3 for the sheep's wool, face and skin. Just like the wolf, a specific function for drawing the sheep's leg is required.

When animating the legs, given the infrastructure as discussed earlier with drawing models relative to both the model and the world, it is simple to implement the rotation of the legs. The difficult part of the whole process is when we would need to alternate the direction of each leg. This is solved by using a static variable that keeps a check on which leg should be going which direction. This would also have to take into account of the fact that the leg are two models.

Table model

For the table model, it's quite similar to the wolf, minus the tail. Simple top with 4 legs. The table top uses the same material as the ground, but the legs are using a shiny material to make it seem like it's made of metal. As such, the legs of the table reflect specular light.

Torch model

For ease on the coding, I did not implement displaying the torch on the view once the player has picked it up, opting instead for a torch spotlight. Nonetheless, the torch is one of the animations that plays when the game starts. It can be seen floating on top of the table, rotating on the y axis and oscillating vertically. The torch has a button the side, a light that is off on the top and a body that is metallic in material.

Sign model

A simple composite object that displays a warning on the front. Uses the same material as the ground and is presented to the player when the game starts. Uses the same method of displaying texture on one side by using 2 boxes.

Trap model

Once the wolf is picked up, the traps will be drawn on the x and z axis, in between the trees. The trap model is possibly the most complex model due to the number of spikes or teeth present. The trap is metallic, thus uses the shiny material. It is drawn in the same manner as the trees.

Data structures

`list.c` and `hashtable.c` are the only data structures used in this game. Hash tables are used to keep track of all textures, shaders and models in the game. Linked lists are used for when we need a dynamic array, like storing attached models to models and textures. These data structures are my own implementation.

References and External Libraries

This assignment used:

- GLAD
  - Used for loading OpenGL
- GLFW
  - Used for interfacing with OpenGL
- stb_image
  - Used for loading images to textures
- cglm
  - Used for a `C` version of the `glm` math library

Resources used:

- learnOpenGL
  - Basic OpenGL tutorial
- write-a-hash-table
  - Writing a hash table in `C`