Gary Sangwell

SAN12366949

CGP2011M

Games Programming

Assignment 1

Table of Contents

Tool chain ... 3
  Development ... 3
Technical features ... 4
  Entity System ... 4
  Standard format model loading ... 4
  Deferred rendering ... 4
  Post processing ... 5
  Skyboxes ... 6
  Fragment shader lighting ... 6
  Billboarding ... 6
  UI Text ... 8
  Particle effects ... 8
  Positional audio ... 9
  Player control ... 10
  Zombie AI ... 10
Reflection ... 10
Bibliography ... 11

Tool chain

For this assignment I decided to continue using SDL2, OpenGL and C++, and so this became the base tool chain for the assignment. In addition to this, I also used a three libraries to further support the game: irrklang, assimp and freetype2.

The game compiles under both Windows Visual Studio 2013, and Ubuntu GCC. Other build files can be generated using the premake4 files.

Development

Git hub was used for versioning throughout the development process, and the final Git hub repository used for the assignment can be found here:

https://github.com/sangwe11/Games-Programming-CGP2011M

Some videos of various programming techniques I tried during the development can be found on YouTube:

https://www.youtube.com/channel/UChoyagvFscFrf0B7bLpaGEA

Technical features

Entity System

An Entity System consists of 3 main parts, Entities, Systems and Components. Entities can be represented as either an object containing all of its components, or merely as an ID into Component vectors. I used the latter approach and instead store all of the components within the class EntityManager.

This should allow for better cache concurrency when a system iterates through a particular type of Component. If the Components were stored inside the Entity class, iterating through all instances of a particular Component type would cause multiple cache misses, due to them being spread all about in memory.

A Component will be represented by the data it needs to function, and any constructing / destructing code. This is slightly different from some Entity Systems, in which Components are just data and all processing is done by the System for that Component. I chose the former approach here as it meant I could decouple what System controlled a particular Component from the Component class, Components no longer need to know what System they belong to, and instead only the System needs to know what Components it processes.

Component update implementation is handled by Systems, iterating through the list of Components of a particular type, processing it, and then moving onto the next Component. A System can handle multiple Components if desired, and does this by providing an update function for each Component type, and then adding that to the core class's main function list. A priority is also given between 0 – 100, and update functions are called in ascending order.

Standard format model loading

Assimp was used to help load standard file formats into a single common file structure, which I then used to pass the model data to OpenGL. Assimp did the majority of the work handling the parsing of the model files, however it still required work to change the format assimp provides into a format OpenGL will accept. I used 5 main classes for this: MeshRenderer, Model, Mesh, Material, Texture2D.

Deferred rendering

A deferred rendering pipeline was implemented, taking advantage of multiple render targets to render to 4 different textures in one render pass. All of the geometry attributes are rendered to 4 textures: world space position, diffuse rgba, world space normal, and specular rgb / intensity. These textures are then read back into the fragment shaders later when rendering the lighting.

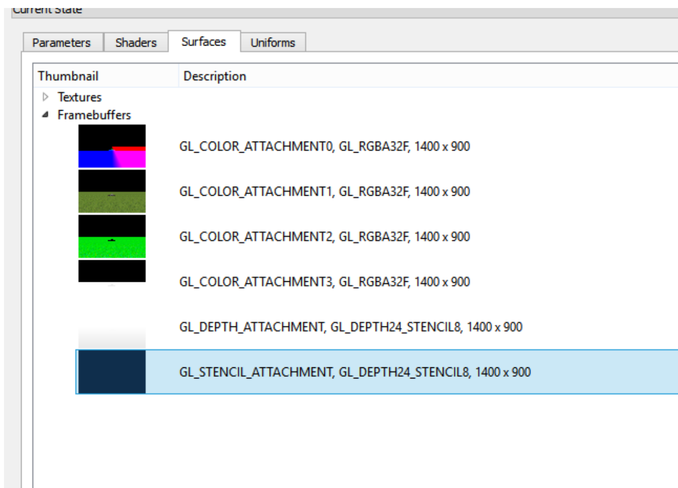

Figure 1: Screenshot of API trace showing deferred rendering drawing to multiple render targets.

Post processing

Support for post processing shaders on a per camera basis has been implemented, with two types of example post processing filters written. A basic attempt at a night vision shader based on luminosity was made part of the game, with the player being able to use the goggles for short periods of time to see well.

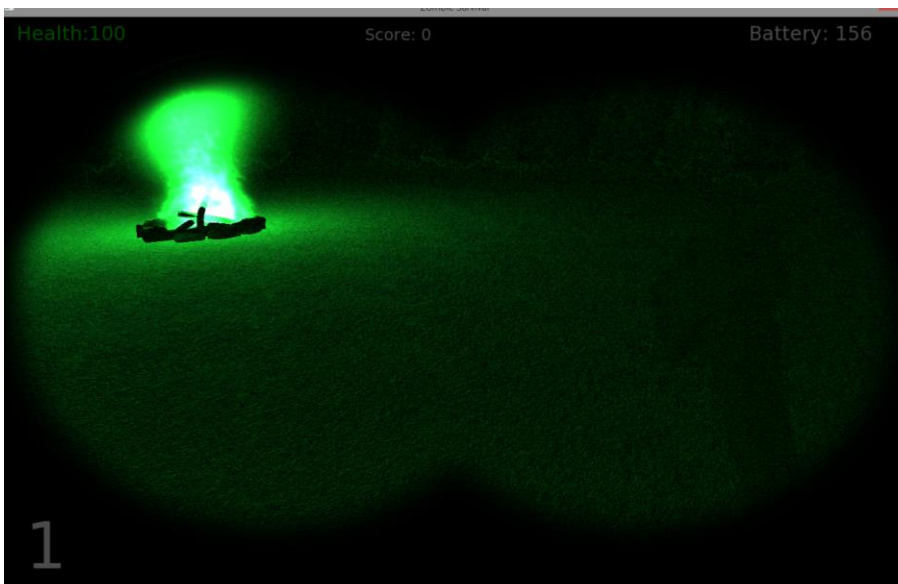

Figure 2: Screenshot showing the night vision post processing shader in effect.

An attempt at a FXAA post processing shader has also been implemented, and is enabled in the game by default.

Skyboxes

Support for rendering skyboxes was added into the engine. A skybox was used within the game to help make it look like the game was set at night, with a starry night sky being chosen as the skybox.

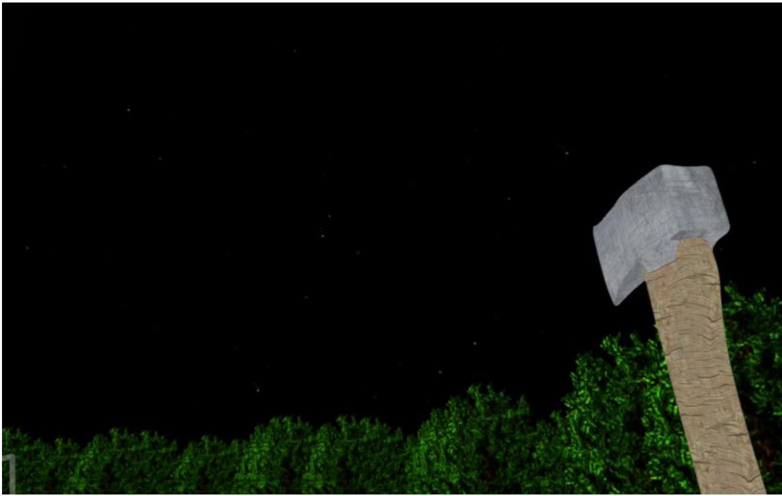

Figure 3: Screenshot showing the use of skybox rendering to fill the empty screen space with a "starry night".

Deferred lighting

I originally planned to use forward rendering, in which lighting calculations are performed when each object is drawn and only one rendering pass is needed. This can however result in a big performance drop when lots of lights are used in a scene, as every object is lit by every light in the scene, regardless of whether that light actually lights the object at all. In addition to this, lighting calculations could be performed on objects that then get drawn over, depending on the order of the draw calls.

Deferred lighting is a technique to delay lighting until after all the geometry has been drawn, so lighting calculations are only performed on the pixels that make it to the screen. This technique does impose some challenges however, and requires the scene to be rendered multiple times.

To carry out directional lighting, a screen space quad is simply drawn to the screen, and attributes are sampled from the Framebuffer textures to calculate the lighting for each fragment. This technique becomes even more complex however when considering point light and spot lights, which only have influence over certain pixels on the screen.

The same use of a screen space quad could also be applied to point light and spot lights, but could result in lighting calculations being done on a lot of pixels that the light does not influence, especially if the radius of the light is rather small. A better technique would be to render a point light as a sphere, and a spot light as a cone. This would result in the fragment shader only being run on pixels that are influenced by a particular light source, as well as give us early culling of light sources that have no influence of pixels in the camera view.

To implement Deferred Lighting in OpenGL, I will make use of Framebuffers and Multiple Rendering Targets, in order to render the scene multiple times.

1. The first pass will render the geometry, and store the output of the fragment shader into different textures attached to the framebuffer using Multiple Rendering Targets. I have chosen to store the following attributes, to later use for lighting: Position, Normal, DiffuseColor, and SpecularColor.
2. For each light, another rendering pass is done using a screen space quad for Directional lights, a sphere for Point lights and a cone for Spot lights. The attributes for each pixel are sampled from the textures in the Framebuffer that were used for drawing in the first pass, and the results are written to a Final texture. Blending is turned on before starting to render the lights, in order to blend the output of the lights.
3. Once all the lighting passes are done, a final pass for any post processing techniques is carried out, allowing for effects such as camera bloom etc.
4. The final texture is then drawn as a screen space quad to the default framebuffer to display the result on the screen.

Bill boarding

Billboard rendering was implemented to render trees as 2D planes instead of full 3D models. My original intention was to use this as part of a level of detail system, which would replace full 3D models with billboard equivalents as the player got further away. After looking into it however, I deemed it would take too long to try to implement, and so instead just decided to go with rendering all trees using billboards.

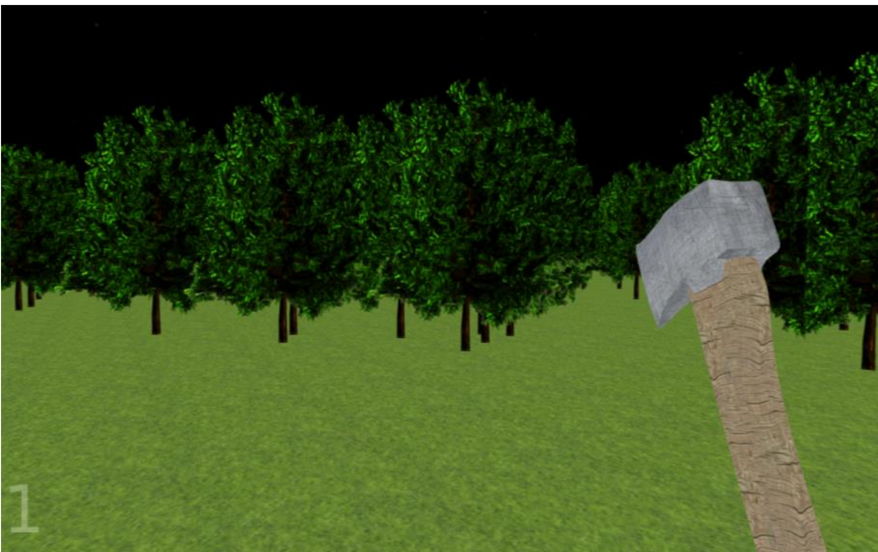

Figure 5: This screenshot shows developing the billboard rendering technique used to render the trees in game.

UI Text

Rendering text in OpenGL proved harder than I thought it would be, as OpenGL has no native functions for drawing text. I looked around for libraries and found several older ones in C++, but struggled to get them working reliably, if at all. In the end I decided to use freetype2 to load and raster fonts into 2D textures, than I could then draw as screen space quads.

Figure 6: Screenshot showing the implementation of freetype2 to render texture to the screen by drawing textured quads.

Particle effects

A simple particle effect system was implemented using OpenGL transform feedback. This means all particle updating and rendering happens on the GPU, and requires minimal interaction from the CPU. My original plan was to use the particle system to render some across the map, but I struggled to get this looking realistic without rendering 1000s and 1000s of particles. In the end I used the particle effect system to create a small camp fire at the player start. Two different effects created with the same system can be seen: a smoke plume and the fire.

Figure 7: Screenshot of developing the campfire for the spawn point in the game.

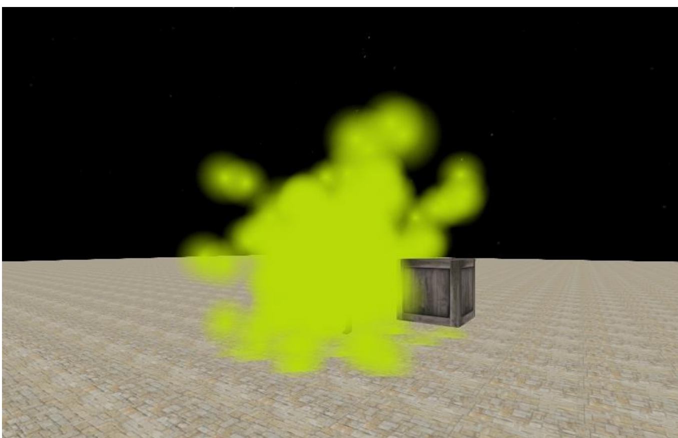

Figure 8: Another screenshot of experimenting with the particle system to try and create different effects.

Positional audio

Irrklang was used to implement position audio, and was a big improvement over the SDL_Mixer library I was using previously. Irrklang did a lot of the work for this, and handled panning the sound and volume automatically, based on a given position and listen position. To implement into the Engine, I had to create 4 classes: AudioSource, AudioSource2D, AudioListener and Audio system.

These components wrapped the functionality provided by irrklang and allowed me to use the positional audio with the entity system. This meant I could simply add an audio listener to the player, and an audio source somewhere else in the game and the sound would be panned and the volume varied depending on the direction and distance to the sound source. I also implemented another component AudioSource2D later on, to handle sounds that did not require a 3D position. This was used in game for the ambient zombie noise.

Player control

The player was given FPS style controls, which allows them to roam freely throughout the level. The player also has two items, a torch and night vision goggles that can be used to help see the enemies better. The Engine is able to handle multiple types of input, including controllers and support hot plugging of controllers. By default the player can use the keyboard and mouse to move and attack, but is overridden by a controller if one is plugged in.

Zombie AI

Some basic zombie AI was added to make the enemies attack the player. The original plan was to try to implement A* to allow the zombies to path find to the player. This turned out to be extremely difficult in OpenGL, C++ and 3D, and so in the end I settled with simply moving the zombies towards the player's position. Once within a certain range, the enemies then attack the player.

Reflection

To reflect, it was clear that I spent far too much time experimenting with games programming techniques and developing more of a technical demo / engine. This became apparent in the last few days of the assignment, when it dawned that the assignment should be a complete game.

A lot of time was spent trying to implement 3D physics, most of which worked fine and can be seen in some of the videos available on YouTube. However I struggled to get collisions working properly, and spent far too much time trying to fix them. In the end I still hadn't got them working, and so have not included the physics in the game. In hind sight this was a massive waste of time, but however a very good learning curve.

I also did not manage to implement animation like I had hoped, as it proved a lot harder than I anticipated. This has resulted in the game looking subpar, as it is difficult to tell when the enemies are attacking the player or not.

Overall a lot of technical features were implemented, but too much time was spent on making more of an engine that an actual game.

Bibliography

Sounds effects were sourced from here:

http://soundbible.com/

http://www.freesfx.co.uk/

3d models were sourced from here

http://tf3dm.com/