Project Report

COMP 371 - Concordia University

April 30th, 2021

Steven Labelle - 40005118

Mahdi Chaari - 27219946

Cristian Milatinov - 40098297

PROJECT REQUIREMENTS & DELIVERABLES

1. PA1

- 1.1. Create a GLFW window of size 1024x768 using double buffering.
- 1.2. 128x128 square grid on the XZ plane centered at the origin (0, 0)
- 1.3. Set of three lines, 7 grid units in length representing the X, Y and Z axes in 3 varying colors
- 1.4. Model 2 Letters and 2 Digits corresponding to the first and last letter in each members first name, followed by the first and last digit of their student I.D
  - 1.4.1. The models should be formed by suitably applying transformations on a unit cube
  - 1.4.2. The models should be placed on quarters arcs of a circle within the boundaries of the grid
  - 1.4.3. The models should be able to be independently transformed using a specified set of keys.
- 1.5. Place a virtual camera with an initial focus on the world origin

2. PA2

- 2.1. Build a stage ¼ the height of the Letter/Digit models from step 1.4.1
  - 2.1.1. Texture the stage using a cloth texture
- 2.2. Build a rectangular screen standing above at the back of the stage, supported by pillars on each side.
  - 2.2.1. Texture the screen with a dynamically changing texture set on a 10s timer.
- 2.3. Add a white point light 30 units above the world using the phong illumination model.
- 2.4. Render the scene with shadows using the two-pass shadow algorithm.
- 2.5. Model functionality should be extended to apply shearing using a key.
- 2.6. Models should be randomly relocated along the grid without any model collision using a key.

3. PA3

- 3.1. Ameliorate the CUGL library to be usable with OpenGL 3+
- 3.2. Extend the functionality of the models to apply rotations using Quaternions provided by the CUGL library.

DESIGN

In the first deliverable of the project, we were recommended to use hierarchical modelling in order to ease the implementation of some of the requirements. Requirements we would not only need to implement in the current specification, but as further deliverables become required.

For requirement **1.1.4** of the specifications, we were required to model our letters and digits using transformations applied to a unit cube. This urged us to think carefully about how to structure our code, as this single requirement came with a lot of important design decisions that could simplify many future deliverables. Some things we considered were as follows:

- A single model would be composed of many smaller objects.
- Each child object belonging to the parent would need it's own set of transformations.
- The final pieced together model would need to be treated as a single entity regardless of the number of children that makes up its composition.
- They should be extensible, we expected textures to be required later..

This is what ultimately led to our design structure for entity management (See figure 1). The model and texture are first loaded using our factory class *AssetLoader*. The class makes use of *assimp* to import object files (.obj) to load into our mesh class as well as of *stb_image.h* library to import image files to be loaded in the *Texture* class. Once both model and texture are loaded, they can be combined into the *TexturedMesh* object where they can then be cloned into *Entity* objects.

The *Entity* class is fitted with utility methods to apply and store the most common types of transformations such as scaling, rotations, translations, etc. All related entities that make up a model can then be grouped together and stored inside the *EntityGroup* class, where further transformations can be applied to the group as a whole. What this means is that when rendering an *EntityGroup*, the transformations of each child entity are applied first followed by transformations that belong to the parent entity group. Entity

groups can also be composed of other *EntityGroup objects*, which further eases the process of applying transformations to large groups of entities. This made tasks like positioning or rotating sets of models incredibly easy and efficient.

This design implementation was simple and yet extensible which further eased our transition as more changes became necessary. In our transition from PA1 to PA2, when met with the requirement of adding textures and metallic appearance, all that was required was extending the *Mesh* class into the *TexturedMesh* class with only minor changes to our render function - everything just worked. Likewise, to accommodate the two-pass render algorithm for shadows, we merely had to create a second instance of our *EntityRender* and, instead, load our new shadow shaders and we could re-render by using the same group of entities we had been using previously.

OBJECTIVES

Our objective from the beginning was to make sure we implemented every feature as efficiently as possible. Due to the nature of OpenGL, codebases that do not maintain good practices can easily spiral into a mess that becomes difficult to develop and maintain. Because of this, we wanted to ensure that we practiced the usual programming practices such as DRY code, code splitting and descriptive naming conventions. As a smaller team, it was important that we be efficient with our time, by carefully designing our classes and implementing the aforementioned good practices, it kept our development time of minor changes short so we could focus on the large features.

ARCHITECTURE

The architecture we chose for our project was simple, we focused mainly on separation of concerns. We wanted to encapsulate each related part of the code into reusable objects. Things such as the entity system, the window, the camera are all broken down into their own respective classes, with properties that are essential to each. With this approach we could quickly identify where changes would need to be made.

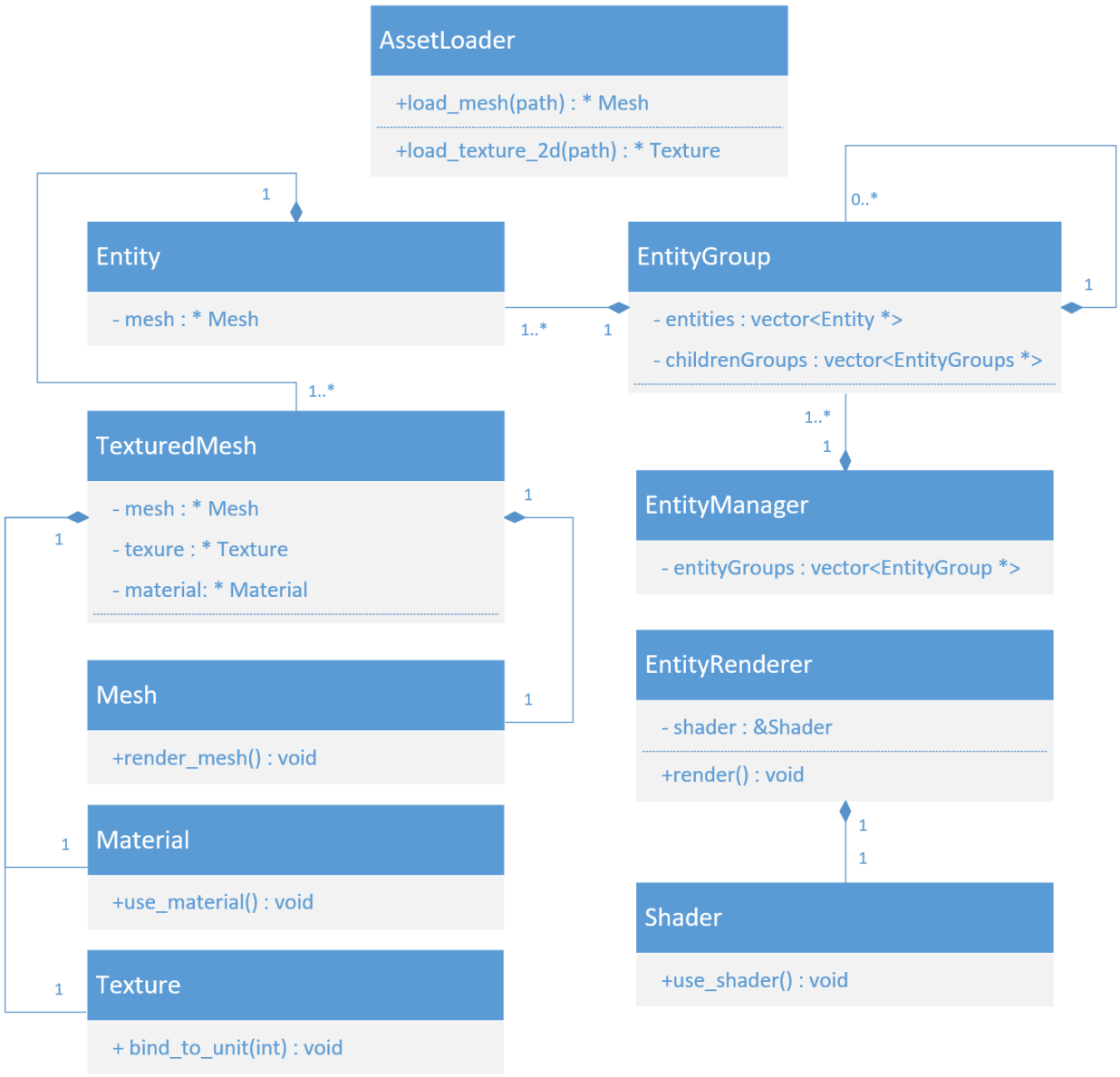

Figure 1. Condensed UML diagram of Entity management structure.

HIGHLIGHTS AND ACCOMPLISHMENTS

The biggest accomplishment is the foundation on which the whole code rests. By following good practices and staying organized from the beginning, adding new features consistently has been made much easier. With the design of the model hierarchy and the object oriented structure of the program, the transitions between the first, second and third programming assignments were smooth. The initial steps took more time to implement and thought had to be put into the design, but it's an investment that saved time later and that avoided technical debt. In many instances, as we've anticipated the features that would need to be added, the new requirements were already implemented or partly implemented. Considering that, a highlight would be the fact that almost half the requirements of Quiz two were already implemented in assignment two. The main and only challenge was creating a spot light with their own shadows.

TECHNOLOGIES

VCPKG - Package manager for C++ libraries to increase ease of setup and portability of the workspace.

OpenGL - Graphics application programming interface.

GLFW3 - Window manager interface.

GLEW - OpenGL Extension Wrangler.

GLM - OpenGL mathematics utilities.

Assimp (Open Asset Import Library) - Used to import more complex 3d .obj files to be used as meshes.

stb_image.h - Image loading library used to import images to be used as textures.

RESOURCES

https://learnopengl.com/

https://concordia.udemy.com/course/graphics-with-modern-opengl/