Maddaloni Gabriele

[S1436255}

Graphics Programming

Program Overview

Contents

[Program Outline 2](#_Toc481341471)

[Architecture 3](#_Toc481341472)

[Overview 3](#_Toc481341473)

[Foundation 3](#_Toc481341474)

[High Level Encapsulation 4](#_Toc481341475)

[The Display class 4](#_Toc481341476)

[The Transform class 4](#_Toc481341477)

[The Model class 4](#_Toc481341478)

[The Camera class 5](#_Toc481341479)

[The Scene class 5](#_Toc481341480)

[The SceneManager class 5](#_Toc481341481)

[The MainGame class 5](#_Toc481341482)

[InitDisplay and InitSystems 5](#_Toc481341483)

[Run 5](#_Toc481341484)

[GameLoop, ProcessInputs and DrawGame 5](#_Toc481341485)

[Classes 5](#_Toc481341486)

# Program Outline

The program is a simple OpenGL application which only functions to load three external models and renders them with the use of an external shader.

To do so, it proceeds through the following steps:

* Initializes the systems:
  + Creates a window
  + Creates an OpenGL context

The user can at this point perform the following operations:

* Set a clear colour for the background
* Load in models specifying a name for each one and whether to load the internal textures specified in the model’s materials
* Load in additional textures for each model
* Load the shaders to use for rendering
* Create a camera and set it as the active camera for the scene
* Create a new scene and add the models to it
* Set the shader to use on each model

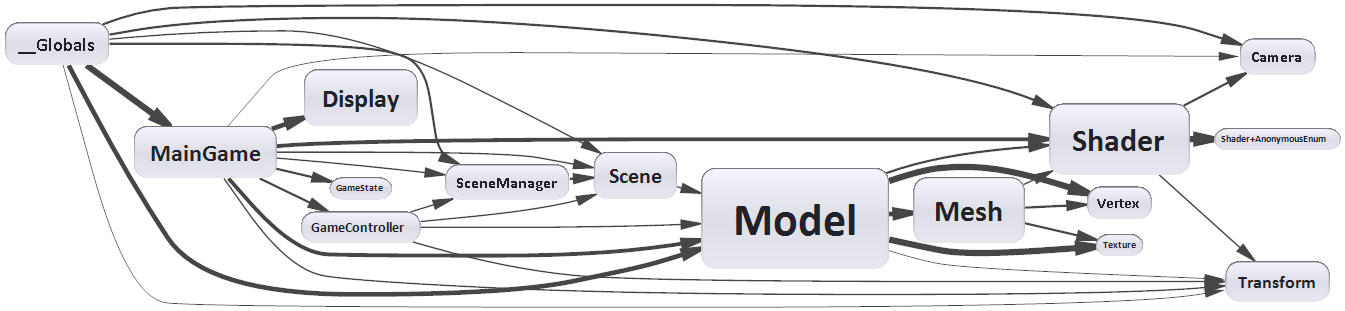
Moreover, the application allows the user to customize properties like position, rotation, and scale for each model. Finally, the user can access an initialization and an update function through a class called GameController, that allows frame-by-frame modifications. Each change can be made consistent through different hardware by the usage of the provided Delta Time variable.

A scene manager is also provided with static members and functions to allow access to the models in each scene. Models can be searched by name or by using an integer to directly reference them in a vector using the GetModel() method.

# Architecture

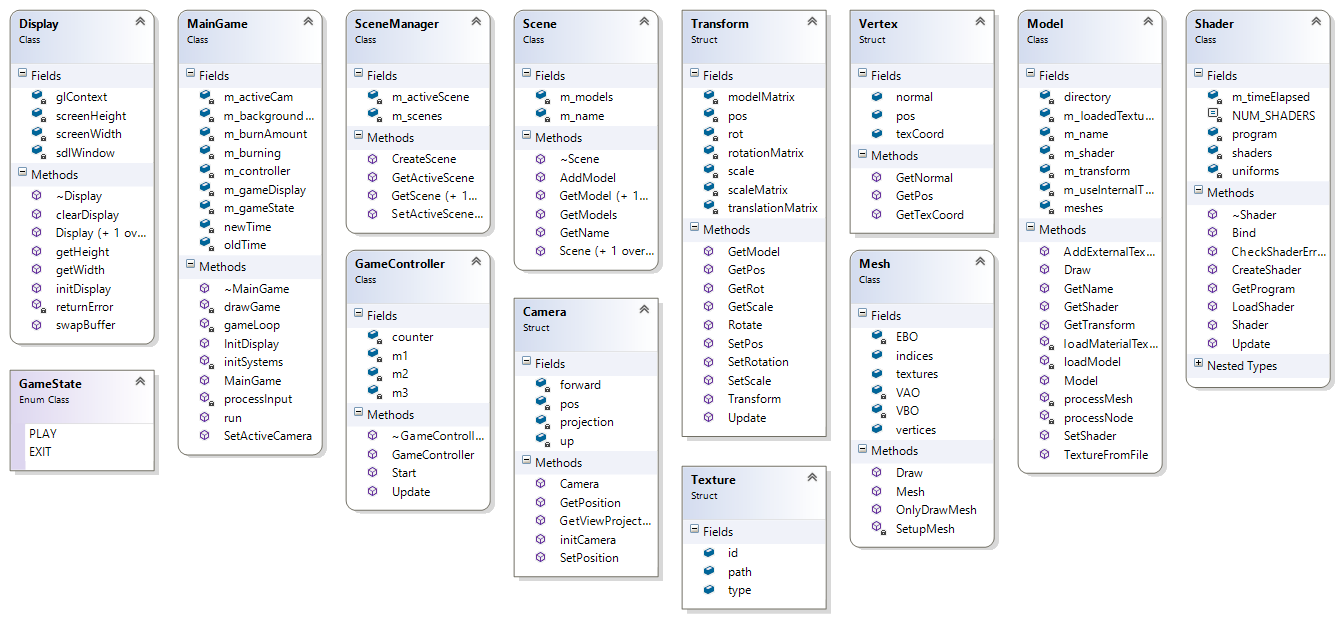
## Dependancy Graph

Following is a dependency graph generated with the software CPPDepend. It shows nicely a view of the architecture, highlighting the encapsulation used by the engine.



## Class Overview

Following is a picture showing a view of the classes, methods and members names were chosen to be self-descriptive.



# Foundation

The building blocks of the application are the fundamental building blocks of every graphics application based in OpenGL:

* Vertex
* Mesh
* Texture
* Shader

The Vertex is a simple struct that holds two glm::vec3, one for position and another one for normal data, and a glm::vec2 for UV information.

The Mesh class has all the information required by OpenGL to draw its polygons, such as vertices, indices and also contains a vector of textures, this vector is only used if during the model loading, the Mesh's internal textures are used. VAO, VBO, and EBO are located in this class as well and are passed to the pipeline through the glBindBuffer and the glBufferData methods, after glBindVertexArray is called to bind the VAO. Additionally this class also handles rendering using two methods. The first is used to draw a mesh binding the internal textures, using glActiveTexture and

glUniform1i(glGetUniformLocation(shader.GetProgram(), (name + number).c\_str()), i);

glBindTexture(GL\_TEXTURE\_2D, this->textures[i].id);

The second one is used to just pass the VBO to the pipeline and to draw the triangles using glDrawElements. The binding of the textures is done in the Model class in this case.

The Texture class is merely a container for information regarding the texture ID (a GLuint), the type of texture and the path to it (both strings).

Finally, the Shader class contains members to identify the shader program, the number of shaders used by the program, used in the linking and compiling process, and an array of GLuint used in conjunction with an enum, to specify uniforms to pass to the shader program at runtime.

The shader also handles the loading of shaders from file and the compiling, including error checking. Moreover, it handles its own binding to the pipeline and takes care of transmitting values to the shader program at runtime using the Update() method.

# High Level Encapsulation

The following classes are used to provide higher level functionality and allow a user to start using the system without having to worry about the lower level core functions of the engine:

* Display
* Transform
* Model
* Camera
* Scene
* SceneManager
* MainGame
* GameController

## The Display class

This class handles the creation of the window used, in conjunction with the OpenGL context, to visualize the rasterized data that comes out of the pipeline. The context is also created and configured in here. This class offers said functionality using the constructor, the initDisplay method, in which Glew also gets initialized, and the swapBuffer() method, that will draw on screen what has been rendered in the context.

## The Transform class

This class, as mentioned before, holds information about position, rotation and scale. In addition to this it also stores the Model matrix, this will be passed to the shader to render the model with the correct parameters and it is used in conjunction with the camera to calculate the MVP matrix. The class also provides move, rotate and scale methods.

## The Model class

This class provides the core functionality to load, draw and manipulate meshes. The methods and members are self-descriptive, to be noted, the methods that handle mesh loading and processing are mostly taken from a tutorial on the Assimp library (*https://learnopengl.com/#!Model-Loading/Model)*. To be noted, the vector of Textures (m\_loadedTextures) is only used if the flag m\_useInternalTextures is set to false during the loading of the external Mesh. Such textures can only be added after the model is created and only by using the appropriate method: AddExternalTexture. The Draw method is a wrapper that will call the appropriate Draw method of the associated Mesh, following the aforementioned rules about internal and external textures. The class also provides a method to set a shader for the model, that has to be called in the MainGame, that will set a pointer to said shader, referenced in the Draw method.

## The Camera class

This class acts as a very rudimentary system that can only be set up during initialization. It sets the projection using the glm::perspective method and is passed to shaders to provide the view-projection matrix. Although very limited, the camera can be re-positioned during runtime using the SetPosition method.

## The Scene class

It offers a way to manage models in a structured way. It contains a vector of references to Models that are imported in the game (preferably in the main function). It provides methods to operate on said vector and is usually accessed in the MainGame class to perform operations on the models. It allows the user to add models to the vector and to search for specific models, and it also contains a name member to allow the SceneManager to search for a specific scene in a similar fashion.

## The SceneManager class

It offers static functionality to store, access and operate on Scenes. It functions similarly to the Scene class does for Models, but it also allows to set an active scene. Only the models present in the active scene are rendered.

## The MainGame class

It acts as a hub between all the systems enabling the game to be initialized, configured and run. Its main methods include:

* InitDisplay
* InitSystems
* Run
* GameLoop
* ProcessInputs
* DrawGame

### InitDisplay and InitSystems

InitDisplay acts as a wrapper for the Display class, initializing it and it also sets the clear colour for the OpenGl context.

InitSystems initializes the variables that will be used to calculate the Delta Time and the burn amount used in the Burn shader.

### Run

This method will call the InitSystems() method, the Start() method of the GameController and it will then proceed to call GameLoop().

### GameLoop, ProcessInputs and DrawGame

The Gameloop method takes care of calculating the Delta Time, it subsequently calls the ProcessInputs method. Then proceeds to call the Update() method of the GameController and finally calls the DrawGame method. All of the methods are passed the just calculated Delta Time for consistent outcomes on different hardware.

The ProcessInputs method is very basic and uses the SDL event system to check if the window is closed, if Escape is pressed, and handles the burn amount variable to be passed to the shader, checking for either the Left and Right arrows or the Spacebar, to be pressed.

The DrawGame method acts in three macro steps: it begins by clearing the background using the provided colour. It then iterates through all the models in the SceneManager’s active scene and for each of them proceeds by doing the following operations:

* Binds the shader associated with the model
* Updates the transform associated with the model
* Updates the shader passing the transform, the active camera, the Delta Time and the burn amount.
* Calls the Draw method of the model

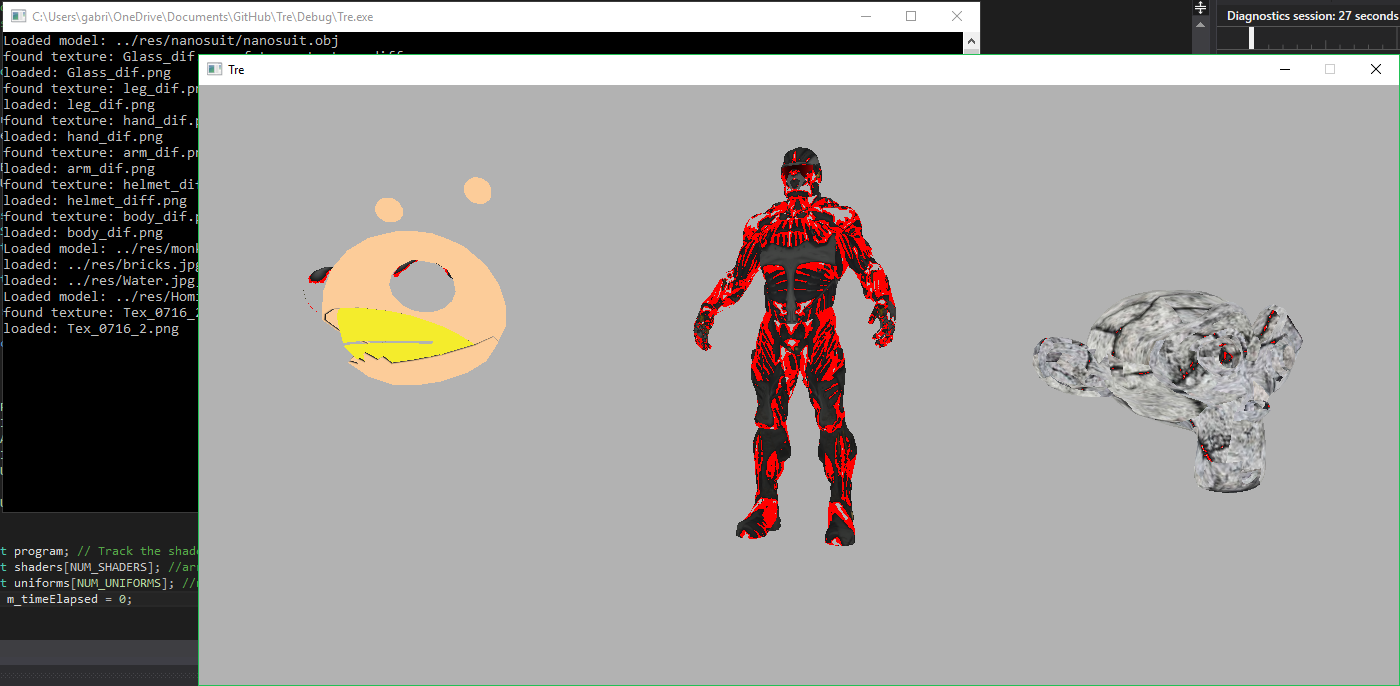
Finally, the SwapBuffer() method is called on the Display object.

## The GameController class

This class offers two methods: Start() and Update(), that allow the user to setup and work with the models. It is the access point of the engine to manipulate models at every frame. Start() can be used alongside the main function to set up initial data for them as well. A very rough outline of Unity's MonoBehavior was used as an inspiration for the class.

# The Burn Shader

The shader used to draw the models was designed to be interactive and work in real-time, following is a picture showing its functionality:

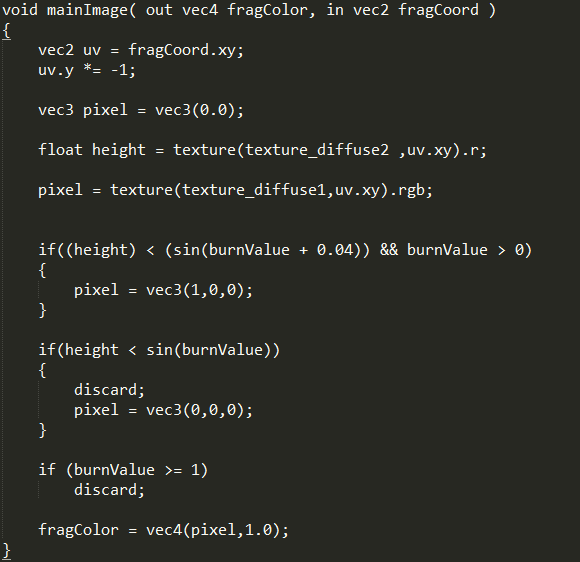


The shader works in a very simple fashion: the Vertex shader is just a pass-through shader that also calculates the position of the vertices using the provided MVP matrix.

The fragment shader (figure at page 9) operates by first calculating the fragment colour using the texture2D function and then proceeds to pass it to the mainImage function to calculate the burn effect.

The function gets the fragment coordinate and flips the Y of it (as we are using the FlipUV setting in Assimp). It then proceeds to calculate the intensity of the red channel for the secondary diffuse texture and stores it in the height variable. After that, it calculates the RGB value of the main diffuse texture and stores it in the pixel variable. The two main calculations are then performed, the first one to generate the red highlight around the “burn”, this is made possible by clamping the burnValue between -1 and 1 (using the sin function) and adding a very small margin to it, to differentiate it from the actual burn value, and then drawing by just using a red colour if that value is less than the red channel intensity cached previously. The second calculation checks if the intensity is less than the pure burnValue (again, clamped as RGB values never exceed 1), and if it is, it simply discards the fragment, making it invisible.

Finally, an additional check is made against fragments with a burn value larger than 1. This is done to discard all fragments that should be completely invisible but due to very high red values in the channel, would still render, creating visual artifacts. The final fragment colour is then set and can be displayed.



# References

The following websites have been used as a reference create the application:

https://learnopengl.com/#!Model-Loading/Model