Joris Schroeven
1305864

AG1101A - Programming Games

Table of Contents

Joris Schroeven
1305864

Introduction

The project is a very basic game engine which can render an abstract 3D environment created using C++ and DirectX. It uses some software design patterns to structure the engine and allow possible future extension of the code.

Minimum System Requirement

- Windows Operating System
- DirectX Version 11 or higher
- (Optional) FBX SDK Version 2014.2.1 or higher

Usage of the Application

The application has a limited functionality regarding movement and can only be performed by using the keyboard.

- **W** or **Up-Arrow**: Move Forward
- **S** or **Down-Arrow**: Move Backwards
- **A** or **Left-Arrow**: Rotate Left
- **D** or **Right-Arrow**: Rotate Right
- **Q**: Move Upwards
- **E**: Move Downwards
- **R**: Look Upwards (only applies to the object representing the player)
- **F**: Look Downwards (only applies to the object representing the player)

Project Features

The project has a number of basic features that are at the current state of the engine not coherent to make a proper game environment, therefore result is a rather abstract environment. The features are as following:

- Loading and generating custom models (TXT is enabled, FBX and OBJ are disabled)
- Basic Lighting
- Textures (Render to Texture is disabled)
- Camera and “Object representing the player” control.
- FPS and CPU counters
- 2 Different particle systems (Falling particles and Flame)
- Basic Terrain

Project Code Structure

The project has been built up by using a number of software design patterns. A number of patterns in each of the three different design pattern categories described in the by the “Gang of Four” have been used throughout the project.

Creational Patterns:

- **Factory Method Pattern**: This design pattern is used by the ModelFactory class to load certain types of model formats such as TXT, FBX and OBJ. (Only TXT is fully supported by the application)
- **Singleton Pattern**: Used for the Shader, Graphics and Input classes.
- **(Hiden) Builder Pattern**: Used by the Map class to build GameObject objects. This pattern is considered hidden as the class name (Map) does not suggest it is a Builder. On a later development of the engine it is likely that the GameObject Builder should be extracted to a proper, separated, class.

Structural Patterns:

- **Composite Pattern**: This is the main pattern used throughout the application. The GameObject class is separated into 3 different components:
    - **Input Component**: Acts as input or “change” component as it handles manual input (by keyboard/mouse controlled by the user) or AI behavior (input controlled by the application). The InputComponent serves as an abstract class and is used as a base for the PlayerInputComponent, CPUCounter, FPSCounter and AIComponent (not fully implemented).
    - **Physics Component**: Not fully implemented, but the basics are present for future extension of the engine. It will handle all kind of physics, it can be mainly seen as the component handling collisions and or physics. The PhysicsComponent class serves as an abstract class and is to be used as a base class for future implementations.
    - **Graphics Component**: Contains the logic to graphically represent the game object. The GraphicsComponent class is the abstract base class for the TextComponent, FireComponent, ParticleComponent and TerrainComponent.

Behavioral Patterns:

- **Template Method Pattern**: This pattern is used in conjunction to the Composite Pattern. The abstract classes are each of the 3 main components of the GameObject class (InputComponent, PhysicsComponent, GraphicsComponent).
- **State Pattern:** Used by the Game class. It has several states but only one state has been implemented (the Playing state), yet the base structure has been designed for different states to be used such as the LoadingMainMenu, MainMenu and LoadingMap states.

The following UML class diagram shows the different classes used in the application and groups them as packages together to highlight the component pattern that is the key pattern in this project.

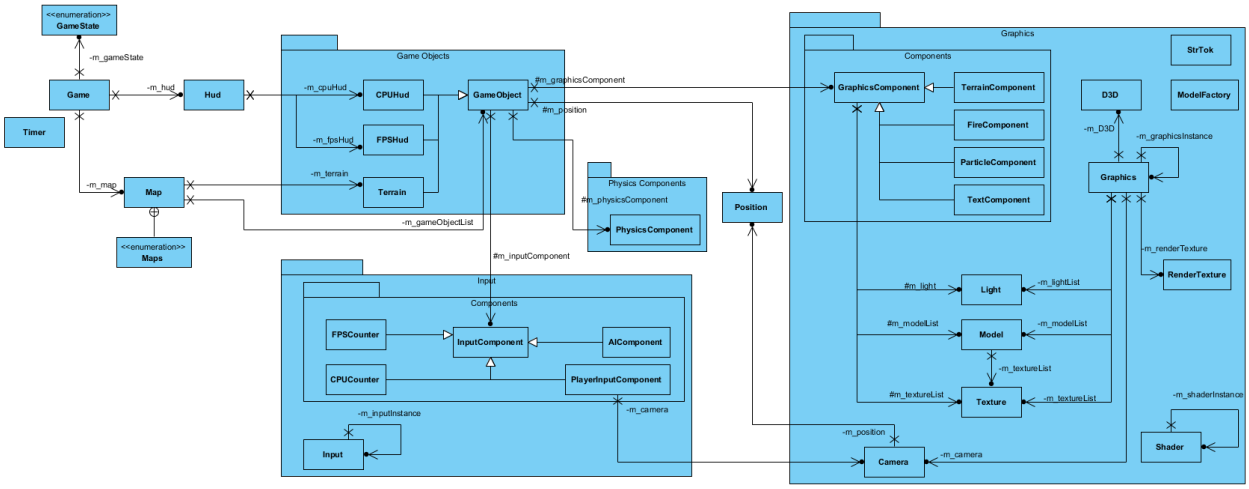

Figure 1: UML Class Diagram (Detailed version available in the project zip file) - Generated using Visual Paradigm

The component pattern is a a very commonly used pattern amongst game programming as it mainly allows a single class (GameObject) to bridge multiple domains (Input, Physics, Graphics and possibly others) without coupling the domains themselves together (Bob Nystrom).
As each of the domains are decoupled and don't need to know about each other in most of the cases there are certain exceptions. The Camera class is one of them, yet it isn't a component of the graphics package, it still is considered part of the graphics as it is used to calculate the view matrix.

In combination with the component pattern the template method pattern makes it easy to extend the base components (Input, Physics and Graphics) to more advanced components. Although the majority of the methods in the abstract class are implemented, not all of them are overridden and not all the superclasses override the same base methods.

The ModelFactory makes use of the factory pattern. It has the responsibility to load data from TXT, FBX or OBJ files and convert this data into useable objects (Models & Textures) which are to be stored in the Graphics class as it keeps a list of the different Models, Textures and Lights. It must be noted that the FBX and OBJ loaders of the model factory are not used or commented as only TXT model loading is fully supported. If the engine was to be developed further a choice of object file format should be chosen and the model factory should be adjusted to be able to load those files types. The FBX and OBJ support has only been added to the project to show the general intent of the responsibilities of the ModelFactory class.
This is also the reasoning behind the Model class having a list of textures as the OBJ file format is not simply a model or mesh, but also contains a reference to a texture file.

The engine uses a number of shaders which are grouped in a single file for each shader (vetrex and pixel shader) and is managed by the Shader class. This grouping highly reduces the amount of shader files and shader classes which would otherwise be required in order to use different shaders for different purposes.

States are used by the Game class to keep track of the current state of the game. The default state is the LoadingMainMenu state, but for this project the user interfaces (such as menu's) haven't been worked out and the states are created to show the intent of what they are used for. As certain game states perform no viable actions for this project, the next state is assigned in order to make the game reach the Playing state which is the only state that has been fully implemented. The LoadingMap state has only been partially implemented to allow the loading of a map file.

The Hud class is created upon game creation with the intent that it should control the menus. As menu's are not implemented, its current (and only) responsibility is to maintain the CPU and FPS Huds. If the engine is further developed it should also maintain different menus and other 2D aspects of the game (such as radar in a shooter game, or a score, or any other 2D interface).

Map is a class that has a double functionality in this project, in further development it should be divided into two separated classes. Its main responsibility is to maintain the environment and all what's going on in there. The second responsibility (which doesn't really belong in this class and thus should be separated out in further development) is the loading of a map, or in code-terms the creation of GameObject instances (or children of the GameObject class). This creation is actually the hidden builder pattern described in the list of patterns above. The map consists of two parts, a terrain part which currently is a static generated terrain of 100x100 units and a list of game objects. On further development the former of these two should be extended accordingly to allow the terrain information (such as dimension, textures, and other information) to be stored within the map file alongside the game objects present in that map.

Singletons have been used for the Input, Graphics and Shader classes. The usage of a singleton for the Input class is very useful to allow validating or retrieving user input (by keyboard or mouse) from different locations. For example the Game loop uses the Input singleton to check for the exit key, such high-level validation is required to allow exiting the application at any given time in case something is incorrect on a deeper level (e.g. PlayerInputComponent not tied to a game object). The other usage occurs in the PlayerInputComponent, but as Input is a singleton it can also be used by multiple other components or even by the AIComponent to use certain information to its own benefit.
The Shader singleton is a result of grouping different shader methods together. As there is only one single shader file to be loaded, the singleton pattern serves perfectly as multiple instances that would load different files can be avoided by extending the existing shader file and thus creating a single shader file again resulting that a single shader class to handle it all is sufficient.
Last singleton in the project is the Graphics singleton. Usage of the singleton for this class is due to the fact that there is only a single display device used and the graphics are only to be rendered on that single display device.
When further development of the engine is to be done, it is likely that the classes which implement Builder patterns are also implementing the singleton pattern.

References

The “Gang of Four”: Gamma E., Helm R., Johnson R., Vlissiders J., 1994, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, USA.

Bob Nystrom, 2009, Component Pattern, Last accessed January 2014, http://gameprogrammingpatterns.com/component.html

Taigi, 2013, Importing and Displaying FBX files, Last Accessed January 2014, http://gamedev.stackexchange.com/questions/46802/importing-and-displaying-fbx-files

RasterTek, DirectX Tutorials, Last accessed January 2014, http://www.rastertek.com

Baychaser, Medieval Models, Last accessed January 2014, http://tf3dm.com/user/baychaser

D. House, Wavefront OBJ Loading, Last accessed January 2014, http://people.cs.clemson.edu/~dhouse/courses/405/docs/brief-obj-file-format.html