Joris Schroeven
1305864

AG1107A - Network Game Development

Introduction

The project is a very basic game built on top of the engine constructed during semester 1. The engine can render a 3D environment and is created using C++ and DirectX. It uses some software design patterns to structure the engine and allow possible future reuse of the engine and easy extension of the existing game code.
The game created with the engine has a very simple design and objective. You can play it in a singleplayer or multiplayer environment. You'll be spawned in a procedurally generated maze and the goal is to find the exit. The first player to reach the exit wins the game.

Minimum System Requirement

- Windows Operating System
- DirectX Version 11 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
- **M**: Map Overview (3 seconds overview with 30 seconds timeout)
- The following movement functionality is only available when "cheat mode" is unlocked by first typing the word "cheat" which will enable the functionality upon typing the last letter.
- **R**: Move Upwards
- **F**: Move Downwards
- **T**: Look Upwards (only applies to the object representing the player)
- **G**: Look Downwards (only applies to the object representing the player)

In menu selections the mouse can be used to click buttons and in the Single Player Menu state the keyboard can be used to input numeric values (see Game States for more details).

To use the multiplayer aspect of the game a Game Server must be run. By default the Game Server will run on localhost and use UDP port 5225. This is also the default host to which the game will connect. Both server and game take 2 arguments as input, the first argument is the IP address (ipv4 only) to which the server will bind or the client will connect and the second argument is the port to be used. If arguments are used, both arguments are mandatory and should be supplied in the correct format.

Project Features

The project builds on top of the features that were created for the Programming Games module of semester 1 and have been extended or revised to fit the needs to create this basic game. The features are as following:

- General Engine Features:
  - Loading and generating custom models (TXT format)
  - Basic Lighting
  - Textures (DirectX default formats supported)
  - Camera
  - FPS and CPU counters
  - Post-Processing Effects (Blur)
  - Networking (UDP using ENet Library)
  - Collision detection (Box, Sphere and Cylinder)
- Game Features:
  - Procedurally generated Maze
    - Prim's Algorithm: Used to generate the maze
    - Dijkstra's algorithm: Used to determine the exit location
  - Multiplayer (Client-Server model)
    - Dead Reckoning: uses a linear model with rotation, time and direction thresholds to update/correct player position and direction.
    - Consistent maze generation: Time is used to seed the random number generator to allow recreation of the exact same maze across all clients who're in the same map.

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 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 System, Game, Shader, Graphics and Input classes.
- **(Hidden) Builder Pattern**: Used by the Map class (and the classes who inherit the Map class) to build GameObject objects. This pattern is considered hidden as the class name (Map) does not suggest it is a Builder.

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), AI behavior (input controlled by the application) and external behavior (input received from an external component such as a game server). The InputComponent serves as an abstract class and is used as a base for the PlayerInputComponent, ExternalInputComponent, CPUCounter, FPSCounter and AIComponent (the latter one is not fully implemented as the game doesn’t have any non-player controlled objects).
- **Physics Component**: The engine can handle simple collision detections for Box, Sphere and Cylinder. Other (more complex) shapes of collision detection can be added later on. 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 and implemented by the GameState class. The GameState class serves as the abstract class which is to be implemented by the different GameStates (For this game there are 4 states: Main Menu, Single Player Menu, Maze Game and Finished Game).

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.

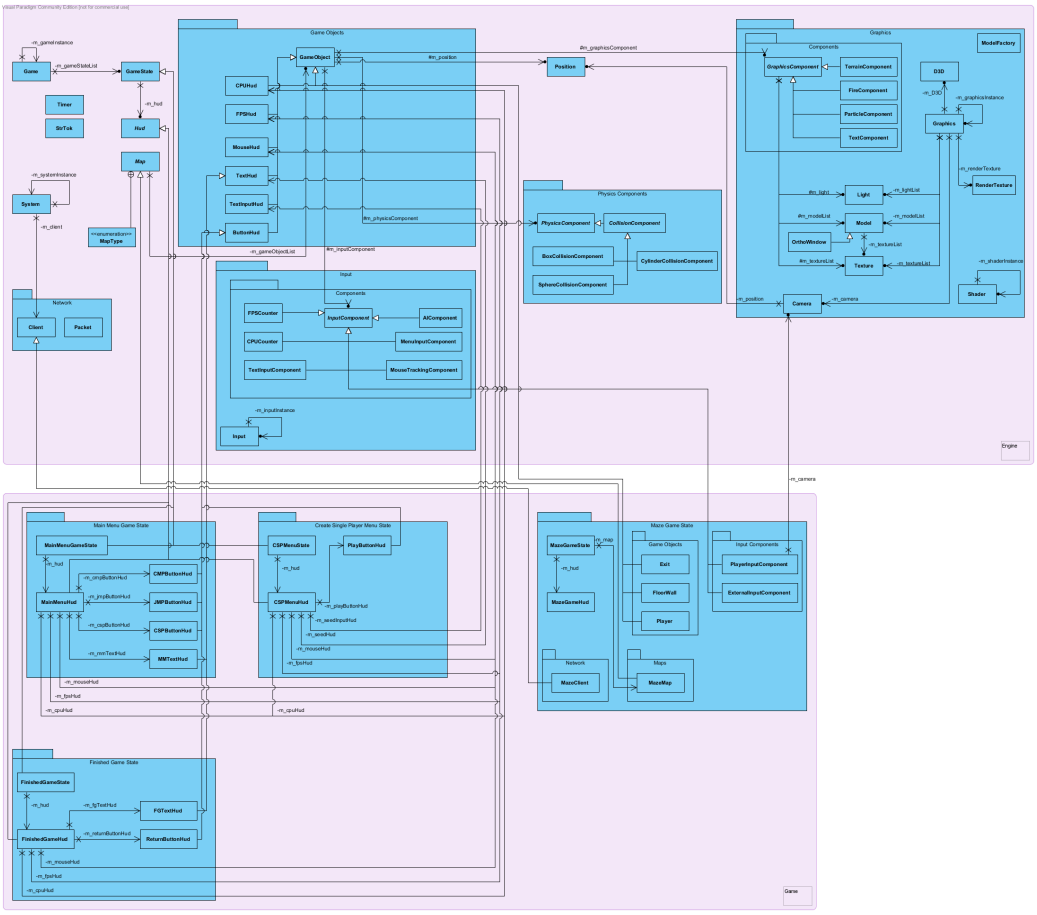

Figure 1: UML Class Diagram (Detailed version available in the project zip file)

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, although 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 model data of different type/file extensions such as TXT, FBX or OBJ files and convert this data into useable objects (Models & Textures) which are to be stored in the Graphics class. The Graphics class keeps a list of the different Models, Textures and Lights while the GraphicsComponent of the GameObjects reference to these Models, Textures and Lights.

It must be noted that the FBX and OBJ loaders of the model factory are not used in this project and have been commented out as only TXT model loading is fully supported. The FBX and OBJ support was added in a previous iteration of the engine 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.

A post-processing blur effect is also part of the engine. This blur effect is enabled and disabled in the graphics class. The game enables the blur effect whenever the user requests a map overview. The overview is a live blur render of the map as the camera's position is altered to a top-view for the duration of the overview.

GameStates are used by the Game class to keep track of the current state of the game. This is also the only place where the more abstract and reusable engine is linked to the game itself. The Game class includes the first/default state of the game, constructs the object and initializes it.

The Hud class is used within the abstract GameState class as each game state is required to inherit from it and will always contain a hud of some sort. In case of menus the Hud is used to allow user interaction where in the actual game the hud is rather used to display info to the user and in further development also allow in-game menus with interactions. For this project the CPU and FPS huds are used through every game state to allow the user to keep track of performance.

Map is also an abstract class from which should be inherited to create actual maps. In this project MazeMap inherits directly from it to implement its functionality. The Map class has the main responsibility to maintain the environment and all what's going on in there. The second responsibility is the loading (or rather procedural generation for this project) 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. For this project the only functional map is the MazeMap which is a procedurally generated maze. Upon creation of the map a Seed is supplied which is then used to initialize the random number generator to generate the maze using Prim's Algorithm. During the generation a second algorithm, Dijkstra's algorithm, is used to calculate the exit of the maze based on the initialization position for prim's algorithm (top left of the maze).
The usage of the Seed is an important aspect to this project as it allowed the recreation of previously generated mazes or replication of them on physically different machines which is part of the multiplayer aspect of the game.

Singletons have been used for the System, Game, Input, Graphics and Shader classes. The usage of a singleton for these classes is justified as only one of them should exist in a single instance of the game and should be accessible from anywhere at any time throughout the code to retrieve or adjust information of these specific classes.

Networking is a new part added to the engine to allow the creation of multiplayer games in a client-server setup. At creation of the networking part the engine was copied and stripped of its Input and Graphics components to serve as a base for the server part of this game. The server engine was further modified to fit its role as server in a better way but still keeping the majority of the original design patterns intact.
The Dead reckoning part of the project uses a linear model with certain thresholds. When a player entity turns more than 5 degrees in any direction, holds down or releases a key or hasn't updated his position for more than 2 seconds it'll update its position and direction information. This combination allows for the fairly simple linear model to work more precisely at a much lower cost of complexity and comes close to what a more complex model would be able to realize.

The final important aspect of this project is the separation of engine and game with an absolute minimum dependency. Both server and game engine's have been designed and built in such way that the engine part can be extracted and re-used at any given moment. This separation was made on purpose for possible future usages of the base engine in future projects. This separation can also be clearly seen in the class diagram (indicated by the light-red rectangular boxes surrounding groups of classes) and the project structure in visual studio.

Game States

The game consists of 4 different states as following:

- **Main Menu state**: this is the default/startup state. Here the user can choose which game type he wishes to play (singleplayer or multiplayer) and in case of multiplayer choose to create a new game (maze) or join an existing game.
- **Single Player Menu state**: this is an intermediate menu that is accessed when singleplayer game is chosen. Is is in place to allow the user to input a seed value which has been obtained from the external maze generator.
- **Maze Game state**: this is the actual game state in which the game is played. Upon initialization of this state the maze is procedurally generated using either the inputted seed (in case of singleplayer), the generated seed (in case no seed was inputted for singleplayer) or the external seed (generated by the game server in case of multiplayer).
- **Finished Game state:** this is the end-game state telling the player if he won or lost the game (In case of singleplayer you can't actually lose as you're the only player).

Network Protocols and Architecture

The multiplayer part of the game makes use of the ENet library made by Lee Salzman. This library provides certain beneficial aspects of the TCP protocol such as reliability and sequencing of packets while underlyingly it uses the UDP protocol to transmit data.
A client-server model was chosen as network architecture. Multiple clients can connect to the same server with a limitation of 2000 clients per server.

Clients that join the server will send either a create or join multiplayer packet once the connection has been established. Based on that packet the server will either create a new map instance when the create packet was received or search for the first free map that isn't full yet and add the player to the existing map. Up to 4 players can join the same map. When all maps are full the game will automatically create a new map instance for the player to join. Once the player has been added to the map instance the server replies and sends the initialize map packet which will be received and processed by the client to initialize the map on the client side.

The engine has its own implementation of a Packet class which uses a character array to store data. The class provides simple methods for writing and reading the majority of standard data types such as integers, shorts, floats, booleans, etc… As more data is written in the packet, the internal buffer is expanded as required.
Once the packet has been filled with data and is ready to be sent another method can be accessed to send the packet. Internally the packet class will encode the buffered data if a send key is provided, add this encoded data to an ENet packet and send this ENet packet making use of the ENet library methods.
The data encoding has been added in as a feature to prevent "cheating" (in this case packet spoofing or packet altering) or at least complicate it to such a level that a higher level of knowledge than the average cheater is required.

All packets sent across from client to server or visa versa are reliable and sequenced. ENet allows to send unreliable and unsequenced packets as well, but this functionality hasn't been used as dead reckoning was implemented and all other packets must be reliable and sequenced due to their importance.

Critical Appraisal

The organisation of the code looks very decent and structured with the use of different software design patterns. The engine separation allows for future expansion and reuse of code. Yet with all these efforts to create tidy and manageable code, it lacks coherency. A lot of game data and information is scattered across many different components and isn't directly accessible.

Yet the strategy of making a component based engine and game makes it efficient if it were to be used by a larger team of developers as a single developer could work on a single component without bothering code of other developers.

An alternative to this component based strategy could be to embed the component's functionality into the class the uses/has these components. This would generate more coherent code, but makes it less manageable by a larger team as multiple developers would work on the same class as well as give a single class multiple responsibilities.

It would also generate more load on the engine as for example if you'd embed collision detection into the game object class, the game object itself would only have 1 collision "component", it be either box, sphere or cylinder. With embedded code this means that the collision calculations for those 3 collision models are within the game object class, making it "heavy" in code and "engine load", yet only 1 model is used and the 2 other would be overhead and not used at all. Whereas in a component structure only the required collision component is loaded and used, creating no overhead code and less engine load. Therefore the choice of a component based engine with these benefits outweigh the disadvantage of losing coherency.

Reflection on what was learned

In regards of procedural methods I've learned a lot through the development of this project as I've built the code to generate the maze just from watching the referenced video of Prim's and Dijkstra's algorithm. It gave me a better view and feeling with the project while other pre-made procedural libraries that I've tried during my research period didn't give me a good idea of where I wanted to take the project to. With those pre-made libraries or pieces of code I tend to feel less in control of code and what's happening inside.
This was especially true in a later stage of development where I've starting working on the networking part of the project. With an earlier research I found the UDT library and it appeared to be decent until I started implementing it. It didn't offer sufficient control over the networking part which I required for this project, lacking certain functionality which I would of expected to be implemented and available in the library. This goes to show that even when reading through the library and its documentation, it can still prove to be insufficient which I only figured out half way through implementation. This has mainly learned me that I should make a proof of concept application for the library to ensure it fulfills all my requirements before I start the actual implementation.

The networking aspect posed a minor challenge as I already had experience in that area. Only the dead reckoning part was something I've never implemented before, yet the article of Peter Soxberger on the topic gave me a good idea on how to implement it.

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.

Jesse Aronson, September 1997, Dead Reckoning: Latency Hiding for Networked Games, http://www.gamasutra.com/view/feature/3230/dead_reckoning_latency_hiding_for_.php
Miguel Gomez, October 1999, Simple Intersection Tests For Games, Last accessed May 2014, http://www.gamasutra.com/view/feature/131790/simple_intersection_tests_for_games.php

Lee Salzman, 2002, ENet, Last accessed May 2014, http://enet.bespin.org/

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

Austin Takechi, January 2012, Random Maze-Generating with Prim's Algorithm (UNITY3D), Last accessed May 2014, https://www.youtube.com/watch?v=ucWX34Vrel8

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

Braynzar Soft, Direct3D 11 Bounding Volume Collision Detection, Last accessed May 2014, http://www.braynzarsoft.net/index.php?p=D3D11BVCD

Da button factory, Button graphics generator, Last accessed May 2014, http://dabuttonfactory.com/

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

Miguel Casillas, 3D Collision Detection (C++), Last accessed May 2014, http://www.miguelcasillas.com/?mcportfolio=collision-detection-c

Peter Soxberger, Dead Reckoning Template, Last accessed May 2014, http://anet-plugin.com/workshops/temp_workshop1_eng.htm

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

Yunhong Gu, UDT: Breaking the Data Transfer Bottleneck, Last accessed May 2014, http://udt.sourceforge.net/index.html

Xander Deseyn, Simple & Fast Game Engine, Last accessed May 2014, https://code.google.com/p/sfgengine/