IMT 3601 Game programming

“Final Exam” project report

Group members:

Ondrej Benes

Gytis Apinys

2016

Table of contents

Repository

Link (there's an installation guide at the overview page).

Game description / overview

"Final Exam" game was intended to be a multi-player, battle royale style, 3 phased game where players would play against each other for the title of the winner. After some time in development the game genre changed to survival RPG single player game where the goal changed from killing other players to killing the final boss. The game still has multiplayer, but only a "spectator mode".

The thing that makes this game unique is the player heath pool. It is not possible to heal yourself in the game so player needs to play around how much he stays in combat to have as much health as he can when he faces the final boss of the game. So in this game, players health is his surviving resource and can determine between winning and losing.

Game mechanics and balance

"Final exam" is a 2.5D RPG style survival game with a goal to kill the last boss. To kill the boss you need to slay few other enemies to obtain keys leading to the final arena and weapons to improve your chances of killing them. Player moves using arrow keys and selects items with mouse from their inventory. The main unique mechanic of the Final exam game is player's hitpoints pool. Players start with a very high number of health that is shared across all the fighting in the game. Because there's no way to replenish your health player needs to think about how much he can sacrifice his health on weak enemies or minibosses so he would have enough health to fight the final boss at the end of the game. Combat in game is made automatic and it takes in account the weapon you're equipped to determine the range of damage that you can do and the attack rate in milliseconds. Other than those controls, player doesn't have that much input on the game, unfortunately.

There isn’t that much balance done in the game. Our original idea was to make player decisions play a very important part in determining on how well he’ll succeed and how hard will the boss fight become. Unfortunately, currently player choice doesn’t make such a big change. All enemies have their health, damage and attack speed made unique so it would feel different from one another (some are fast but weak in damage while others do heavy damage but attack very rarely).

Game design

At the start of the project we devoted 3 weeks to game design. We had one more person in our group so we assigned each group member to area they would be responsible and in charge of. Ondra was in charge of technical part, Gytis - game mechanics and design while the last person was in charge of graphics and music. We had weekly meetings where we all proposed our ideas on how our game should look. By the first week we had decided on a game idea, second week we agreed on mechanics and by last week of design development we come up with the style of the game.

Our game design has changed a few times over the time we were making it. We made a lot of changes and cuts because of our groups resize and time that we could have spend on this project.

Original design

The game idea was to make a 3 phase multiplayer game with each phase having different feel and style. The main goal was to be the last one standing at the last phase. The unique aspect of this game is that you start with a fixed amount of health and there’s no way to replenish it, only to decrease. So your goal is to save as much as possible of it till the last phase.

First phase would have been gathering phase where player explores area and caves to find equipment, weapon scrolls, resources needed for surviving till last phase.

Players could also find map pieces that could help them pass 2nd phase much safer. In this phase, there would also be PvP enabled, so some players could try killing others for items they had gathered.

After some time the gates to 2nd phase would open and everyone would go there with as much items they could have found. This phase would have been more of a puzzle game. Players would appear inside of the labyrinth and would need to find a way out. Labyrinth would be filled with fightable creatures that would damage the players thus making them weaker at the last phase. If the player had any map pieces from the first phase they would unlock a part of a minimap and letting them pass it without any problems. Like in the first phase, after certain time there would be message poping up saying that you should hurry up. If you're still in this phase after time runs out, you'll get periodical damage done to your character.

Finally there's the last phase. It's a simple battle royale where the winner is the last player that's standing alive. Everyone would fight with every resource they have gathered and because of the time limit in 1st phase, everyone should have different resources so the battles should be random for most of the time. The fighting would be done in one to one combat (There would always be 6 players in the game, bots if needed). The winner would be the one who wins the last fight.

Final design

Current game is changed from multiplayer to single player and we changed the game to have only 2 phases. We completely scratched second phase.

Player starts at the island's dock and is greeted by a friendly NPC "Shiny Donkey". He explains that to the player that to win the game he needs to collect bronze, silver and gold keys and open the gates to the final fight. After killing the first enemy, player will get stronger weapon that he can use for other enemies. Following the path, player will have to kill mini bosses that drop keys and weapons needed to progress. After killing the last mini boss, player will go to gates in the middle of the

island where after opening them and stepping on the entrance he will be teleported to the final boss battle.

At the final phase player will see a giant boss that is guarded by 2 demon dogs. The idea here is that you should kill them before attacking the boss. After killing the dogs you are left to kill the boss that has the highest stats of all the enemies in the game and because your health does not heal after previous battles you fight him with the amount you have left. So basically, all your gameplay till the last boss is your preparation for the final fight.

Scratched ideas

For the first few weeks, we followed our weekly work pace according to plan so we did a mistake of making plans based on our current workflow. After our group resize, time and difficulty overestimate we noticed that we won't be able to do everything that we planned so we had to remove a lot of things that we wanted to implement.

Game mechanics we removed:

- 2nd game phase
- Weapon enchantments
- Player stats
- Timed gameplay
- Cave exploring
- PvP multiplayer
- Player controlled combat
- Farmable enemies for items
- "Map piece" item mechanic

Technical options we wanted but didn't use:

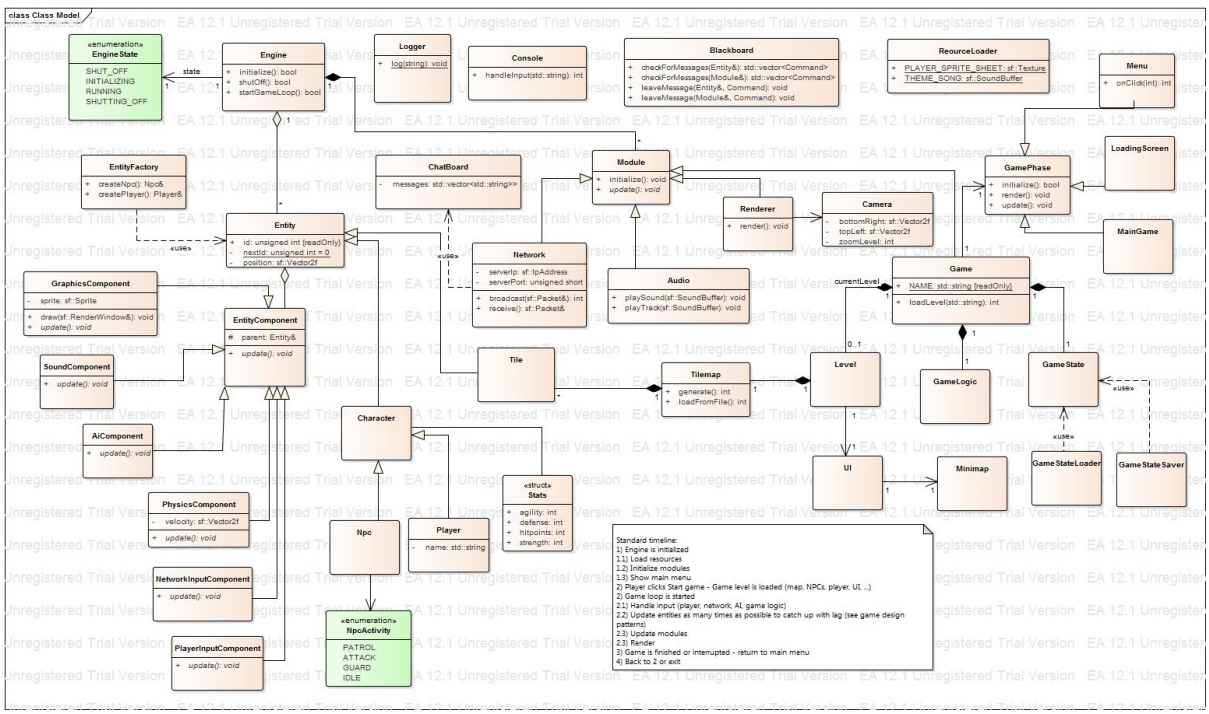
- Lua integration
- Ai pathfinding

Technologies used

- SFML - we used this library for making graphics, networking, vectors, sounds and other basic things needed for game.
- Tiled - program we used for making our games map.
- tinyXML2 - very simple to use open source library for reading/writing data into XML files
- Windows API - was used for getting system time and reading / writing to config files

Implementation

Before we started with development, we've created a UML class diagram, which captured the intended engine architecture. A mistake that was made was that we intended to create an engine and build a game on top of it later on. Considering our experience with game programming (or lack thereof), we should have just focused on developing a game. There have been many changes to the architecture, but mostly adding new classes. The core remained more or less as shown below.



Engine and Modules

The core of the engine is the Engine class. It is responsible for initializing the Modules, running the game loop and exiting the game.

We are using five Modules - Audio, Game, Network, Renderer and Scheduler. All of these are described in further detail below. The modules have an update method, which gets called in the game loop.

Audio

Since we are using just two sounds (“Weapon clash” and “Death cry”) and one background track, played in a loop, we have only a very basic support for audio. We are using SFML extensively for Audio handling. There are two kinds of audio that can be played - sounds and music. We’ve created a Sound class, which is a wrapper around sf::Sound and sf::SoundBuffer. Sounds are placed in a cache (a std::map) in the Audio Module. The audio volume is loaded from the config file (separate values for sounds and music).

Game

The Game module is responsible for creating the initial GamePhase (the Main menu). In its update method, it calls the current GamePhase update method. It’s also responsible for breaking out of the game loop.

Network

The Network Module handles the networking. We are using SFML and TCP. The Module can operate in two “modes” - client or server. The player runs the game in the server mode while the spectators run it in client mode. The behaviour is also different depending on the current GamePhase:

- Server mode
    - Create multiplayer game menu

        - If a new connection is registered, a sf::Socket is added to the sf::SocketSelector and to the vector of clients.
    - Main game
        - Check the client sockets, if there are recieved packets, they get resend to other clients
        - The message is decoded and if it is a Chat message (the only type of message the server can parse), it gets displayed in the chat window
- Client mode
    - Join multiplayer game menu
        - Establish connection with the server
        - If a “Game start” message has been received, start the MainGame phase.
    - Main game
        - If a packet has been received, decode it. If it is a known message type, execute the relevant commands

The Module has a broadcast method, which is used to send packets over the net. In server mode, the packet gets sent to all clients. In client mode, it is sent to the server. The packets (sf::Packet) are created in the PacketFactory. The message has the following syntax: MESSAGE_TYPE[[;MESSAGE ARGUMENT]*]. For example, a “VELOCITY_CHANGE” message might look like this: 2;0;150;0 (where 2 is the message code, 0 is entity id and 150 and 0 are the velocities in the direction of x and y axes, respectively.

Renderer

The Renderer keeps a pointer to the main window (sf::RenderWindow). The rendering is done in the render method (called from the game loop). First, the current GamePhase render method is called. In case of the MainGame phase, tiles and entities are rendered (by calling the draw method of their GraphicsComponents). GUI elements are also drawn by the GamePhase. Then, the console is drawn (if the visible flag is set to true), the view is centered on the player and zoom is applied.

The renderer has support for fade in and fade out effects. A sf::Clock is used to determine the opacity of a black rectangle that is drawn over the window to achieve the desired effect.

Scheduler

The Scheduler is a workaround that was needed because we are not using multithreading. It has a priority queue of Tasks. Task has a callback and a std time_point. The Task's priority is determined by the value of its time_point. The Tasks are first pushed to the Blackboard, then the Scheduler places them to the prio queue. The callback gets called when the current time is greater or equal to the time_point.

Game Phase

GamePhases are classes responsible for game mechanics. They are created by GamePhaseFactory and are stored in GamePhaseManager in a stack. This allows us to for example push the MainGame phase on top of the MainMenu phase. When it is needed to return to the menu, the MainGame is popped off the stack.

The GamePhase class is subclassed by two classes - Menu and MainGame. These classes have different responses to the player's input. The MainGame class has methods that handle the game mechanics (updating the Entities, teleporting the player to the Arena, handling player's death etc.). It is also (indirectly) responsible for creating the world map (it delegates to the LevelLoader class, which creates the level based on a XML definition).

Blackboard

We decided to use the Blackboard pattern for communication between Modules. The Blackboard has vectors of callbacks for each of the Modules. The first thing that a Module does in its update method is checking the queue for callbacks and if it finds any, calling them.

The Blackboard also has a queue of SFML Events. Events are enqueued in the main loop and are dequeued by the current GamePhase. That way we can handle key presses, mouse clicks etc.

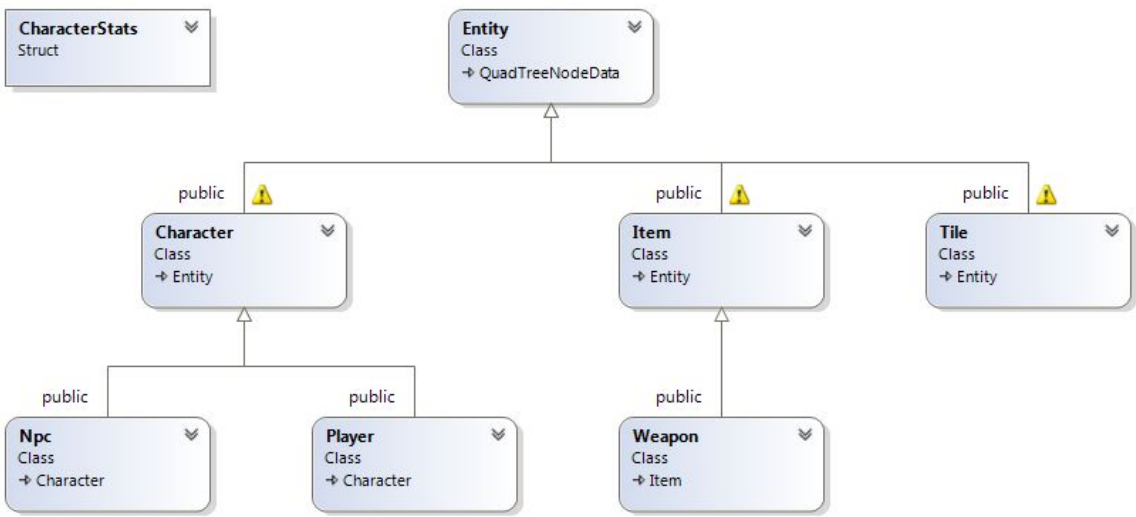
Component system

We are using a component system in order to improve the decoupling in our architecture.

Entity

The Entity is the base class of all the game objects, such as Characters and NPCs. It is derived from the QuadTreeNodeData, so that it can be placed as an element of Quadtree. It has id, name, position and a collection of Components. It's update method calls the update of each of its components. The components can be accessed using a template method called getComponent.

The Entities are created in the EntityFactory class and are stored in the EntityManager.



Components

The concrete components are derived from the EntityComponent class. This abstract class declares the update method and the link to the parent Entity.

Ai

The AiComponent encapsulates the artificial inteligence of NPCs. We are using the Strategy pattern to simulate different behaviours. The AiComponent has an AiState, which can be either AiIdle, AiPatrol or AiAttack. NPC in Idle state scans for the player in a defined attack radius and switches to AiAttack when the player is in range. In the AiAttack, the NPC gets to weapon range and starts combat. The player may disengage by leaving the attack radius. In AiPatrol, the NPC moves between randomly generated points inside its patrol radius. Similarly to the AiIdle, if the player enters the patrol radius, it switches to AiAttack.

Very simple pathfinding is used to guide the NPC's movement. The velocity of the Entity is set so that it will move towards the move point's y coordinate. After the entity's y position is approximately equal to the point's y, the process is repeated for the x coordinate. This means that the Ai may get stuck by blocking tiles.

Animation

The Animation component changes the cell (the texture rectangle) of the entity's sprite sheet. The changes on the x axis happen in constant intervals. The y axis value changes if the entity's movement direction changed. The sf::FloatRects (which represent the cells) are cached by the GraphicsComponent.

Combat

This component is above all responsible for doing and taking damage. The damage is based on the stats of the equipped weapon. The weapon's stats also include the attack speed (the interval between successive attacks). In case that the Entity's health reaches zero, it's death is handled (loot drop, death cry sound, removal from EntityManager). Handling of the player's death is delegated to the MainGame class.

Graphics

The GraphicsComponent keeps a map of SpriteWrappers. SpriteWrapper includes the sprite itself (sf::Sprite), the sprite name, it's offset relative to the parent Entity's position, the number of cells in both x and y axes and the cache mentioned above.

The draw method simply draws the active sprite on the main window. It may also draw the Entity's collider (if the relevant flag is set to true), which is useful for debugging.

Physics

This component is responsible for moving the Entity in the game world and handling potential collisions and activations of triggers. The Entity velocity is a sf::Vector2f. During each update call, a new position is calculated based on the velocity and the time that has passed since the previous update. Before moving the Entity, a collision check is run. It consist in checking for nearby tiles and characters. If any of their

colliders would intersect the Entity's collider, the Entity is not moved and it's velocity is set to zero.

In case that there was no collision, the nearby tiles and characters are checked again. If the Entity entered any of their triggers, the relevant callbacks get called. An example of such a trigger may be the teleportation of the player to the Arena.

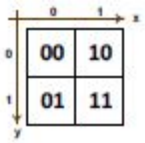
Spatial indexing

A Quadtree is used for spatial indexing of Entities. The EntityManager has two trees - one for tiles and one for characters. The Quadtree uses an insertion path which determines in which node an entity will be placed. The x and y coordinate are divided by the width (or height, respectively) of the tree. This value is converted to a binary number. The path is then made up of a sequence of pairs, n-th pair being the n-th digit of the x and y binary value. This pair gets translated to direction, so 00 is NorthWest, etc. For example, if an entity with coordinates 25, 55 was inserted to a tree of size 55, 70, the path would look like this:

$$p\,(25;55) \approx (25/50;55/70)_{10} = (0{,}1000000000;0{,}1100100100\ldots)_2$$

$$\Downarrow$$

$$11|01|00|00|01|00|00|01|00|00\ldots$$



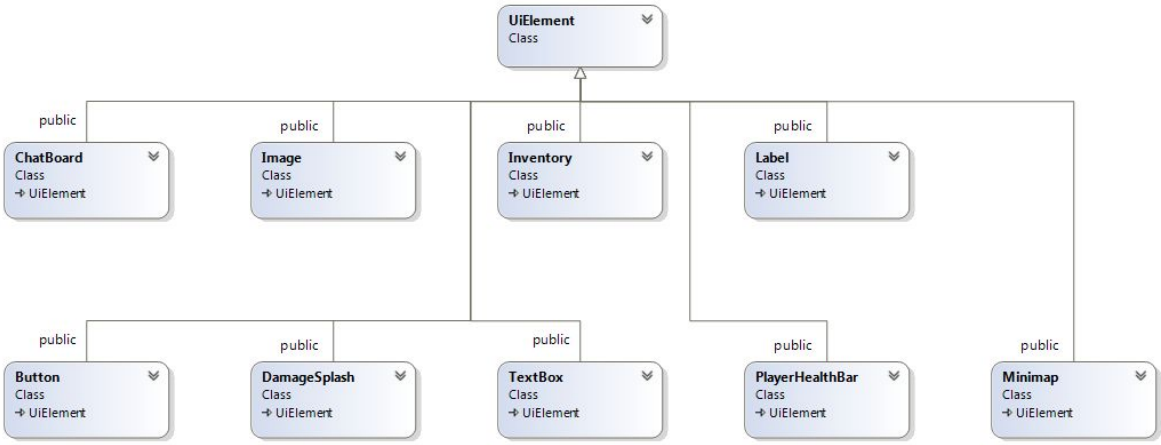
The implementation is based on an assignment done by one of the group members at his home university.

GUI

GamePhases have an instance of the Ui class. Ui class has a vector of UiElements and a pointer to the focused element. UiElements have a method for drawing them on a sf::RenderWindow. They also have a boundary, a name and may have callbacks (onClick, onKeyPressed, onTextEntered).

The current GamePhase checks the player's input, and calls the relevant callbacks (for example, if a mouse click was registered inside an UiElement's boundary).

The following subclasses of UiElement are available:



Utility classes

A number of utility classes was implemented. Some of the more prominent are described in more detail.

Console

The Console proved to be quite useful for debugging. It attempts to parse the given input based on regular expressions, and if it succeeds, it calls the callback paired to the regex. Currently, the following commands are supported:

- help - shows the list of commands and the syntax of each command
- list - outputs a list of all the Characters (shows the class, id and position)
- move - moves an entity with the provided id to the provided position
- zoom - sets the zoom of the main window's view

ConfigIO

A class with static methods that can read and write to .ini files. It uses Windows API. There is one main config file, which stores values such as screen resolution, player's name etc. There are also ini files with definitions of Ui elements for each of the GamePhases.

Logger

The logger saves messages to a log file, and if the game is run in debug, it also outputs to the console. The date, time, severity, class, method, line and message get saved. A number of convenient macros has been created (for each severity level), so all that needs to be passed is the log message.

ResourceLoader

This class is used for loading sf::Fonts and sf::Textures. Using a central repository for these resources means we don't need to worry about duplicate loading and memory leaks.

Discussion

For this course our group made a game that we called "Final Exam" to learn how to do game programming and how to work on it for a long period of time. All of our group members are international students who never did any game programming, so this was a perfect time for us to learn about it. We wanted to get the most of this opportunity that we had so we decided to put our focus on the technical side, learn how game loops work, how does every component work together and make game mechanics second thing.

For game development we used C++ and Visual Studio with some additional libraries. Most used one was SFML. We had a discussion about maybe instead using easier and more advanced development engines like Unreal or Unity but in the end we decided to use lower level engines to get more knowledge from the project.

In the earlier stages of development we had support for saving and loading the game, but we have not maintained it as the development progressed, so it does not work in the current build. We wanted to reimplement it so that it would use XML, but did not have the time.

What we would do differently

Bigger team

At the beginning of the course when we needed to form groups, we should have made a group of more than 3 people or ask other game programming students to join our group so we had a better starting project. It would probably have been useful to have some Norwegian students in our group, because they could provide knowledge from the first two years of the bachelor programme.

Focusing more on simple player input

We should have thought of easier ways of how players could have interacted with the game and made their decisions make a visible change to the game. Our original ideas had a lot of them but most of it were too big for us to implement. Simple combat improvements would do the trick.

Lua integration

We haven't encountered any serious obstacles or dead ends, so we were not forced to abandon an idea or a feature just because we did not find a way to implement it. An exception might be the failure to include Lua.

Towards the end of the development, when we started to implement game mechanics, the originally decoupled EntityComponents became more and more coupled.

We believe that if we managed to integrate Lua, we could have had a ScriptComponent, which would hold Lua scripts, that would eliminate (or at least considerably reduce) the coupling of EntityComponents. Lua might have also been useful for debugging (running a script from the Console for example) and for Ai scripts.

GUI library

A lot of time was spent on implementation of GUI elements. While it was interesting to make a GUI system from scratch, the time may have been better spent on working on the Ai or the Combat system. It would have probably been better if we used a GUI library instead (something similar to Java's Swing or C# WPF), although that would require some amount of research and learning.

Config API

The Windows API for writing and reading .ini files is not very easy to use, especially due to the need to convert between std::string (used everywhere in our project) and std::wstring (used by Win API).

We should have used XML instead, because we needed it for the level definitions anyway. Nevertheless, it was interesting to learn this small part of Win API and it might be useful in the future.

Ondrej's view

Responsibilities

- Most of the code
- Engine architecture
- Mentoring and project lead

Lessons learned

I had very limited experience with game development, the only game I ever made before this course was done in JavaFX, so most of the code was not directly applicable in C++ environment.

I spent the first month learning about design patterns used in game development. I knew some from my Java background, but most were new to me. I wish I had more time to go deeper into this area and make our architecture "cleaner" as a result.

I had to refresh on my C++ skills, since I had not written in C++ code in about 2 years. At the beginning of the semester I had quite negative attitude towards C++, especially due to the steep learning curve and the “archaistic” way in which it was taught back at my home university. I have to say that my attitude changed and I now enjoy writing C++ code, especially after learning the modern C++ concepts. Using Resharper (VS addon) was also helpful in making me feel like I’m working with a modern language and IDE.

I consider the C++11/14 stuff to be the most valuable outcome of this course, because they are also applicable outside game development. I find lambda expressions extremely useful (and I made extensive use of them in our code). The smart pointers are also very handy and I regret not knowing about them at the beginning of the project. It would have saved me time that had to be spent on refactoring later on. I also got used to using the std collections, iterators and algorithms whenever it made sense. The auto keyword is also amazingly useful.

It was interesting to learn about game AI. The AI courses I had taken focused mostly on searching trees and optimization. Same goes for networking - I had never done anything not related to databases. The SFML library was fun to learn and to use.

Personal reflections

The course was at times quite frustrating, mainly because I was responsible for writing most of the code. The dropout of one of our team members at the beginning was quite a heavy blow.

I hoped to be able to focus on the AI, which is the part of game programming that interests me most, but due to time and priority concerns, our game ended up only with a basic AI. But I learned a lot about game programming patterns and modern C++.

In the end I am quite satisfied with the final build of our game, espcially because we managed to cover all the requirements (Ai, Networking, Physics etc.), but it's a bit sad to see it not reach the potential that I believe it has, due to the factors that were out of my control.

Gytis's view

Responsibilities

Because of my lack of technical skills I wasn't able to take responsibilities for full parts of any sort, so I tried my best to help with everything I could that required that little coding skills.

Lessons learned

Everything I did here was new to me. Understanding how real game needs to function was the biggest learning experience for me on this course. Just making the character move to where you want him to move might require a lot more work then I had thought.

I started looking at programming overall from other point of view to see the bigger picture. Seeing how to micromanage every single thing that happens at the same time was also a really great learning experience. I got a lot of help from Ondrej regarding this work and basic help so that helped for me to learn more and quicker.

Personal reflections

Working in a group for a big project was really awesome. Brainstorming for ideas, working together to find an error and such makes working much easier and faster.

I liked the weekly review sessions of this course. It made us want to show something in front of everyone and so we tried to make things eye catching.

At first, it actually made me worry a bit because I had no real idea where to start when you want to make a game so it was a bit scary, but it turned out quite well.

References

The following resources were used during development:

How to make your own game engine - Module architecture

SFML made easy - SFML tutorial

Tutorial: Basic Game Engine - Game Phases

Game programming patterns - Game loop, Component system

A C++ Config File Parser - Reading and writing to .ini files