Tarnowskie Góry, 10.02.2018

Semester: 3

Group: 1

Computer Programming Laboratory

Knighto 2D – Platformer Game

Author: Marek Paśnik

E-mail: [marepas035@student.polsl.pl](mailto:marepas035@student.polsl.pl)

Tutor: Roman Starosolski

1. Task topic

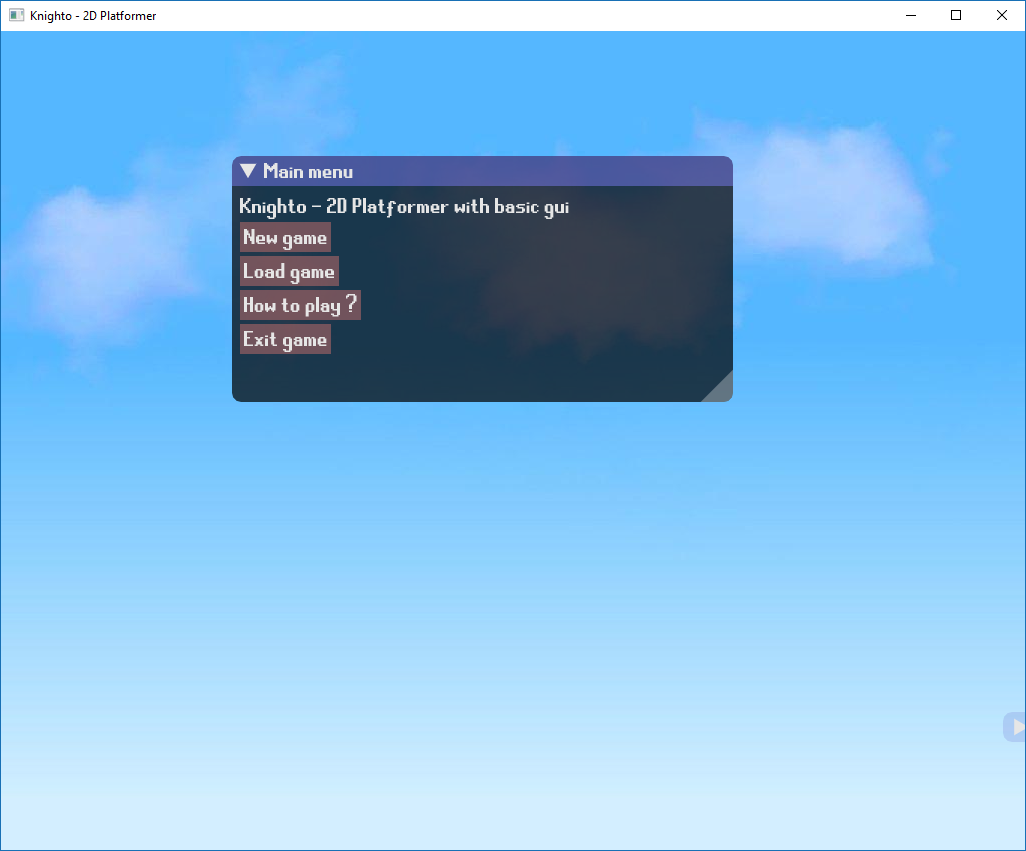
My laboratory task was to create a two dimensional platformer game like Mario. Inspired on Mario game in my game will be main character named Knighto, which has to fight his way through the countless enemies like Orcs, Shamans to rescue his Princess. At the start Knighto will fight with his bare hands, collecting equipment and consumables and defeating enemies that are standing in his way.

1. Project analysis

Before starting the implementation I collected information from many game developing sources and consulted about which library I should choose. My choices were:

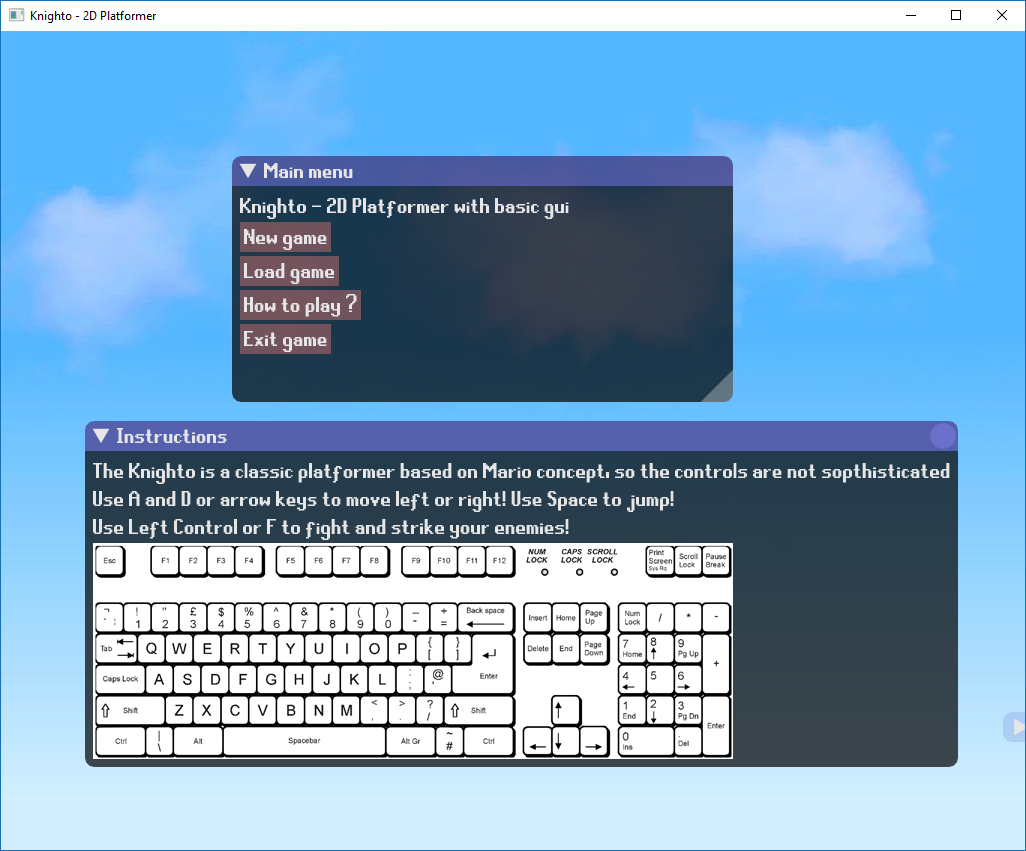
* SFML Library – because C++ language lacks any graphical implementation like drawing sprites and GUI. SFML Library gives possibility to easily implement any sound, fonts management.
* ImGUI – which is a binding to SFML library, which helps the developer with creating GUI for his projects. It wasn’t mandatory but helped a lot.
* Assets from GameDevMarket – because I lack skill in graphical design, my project is using free assets like sprites, fonts and sounds which are available on this webpage.

1. External specification
   1. Main menu



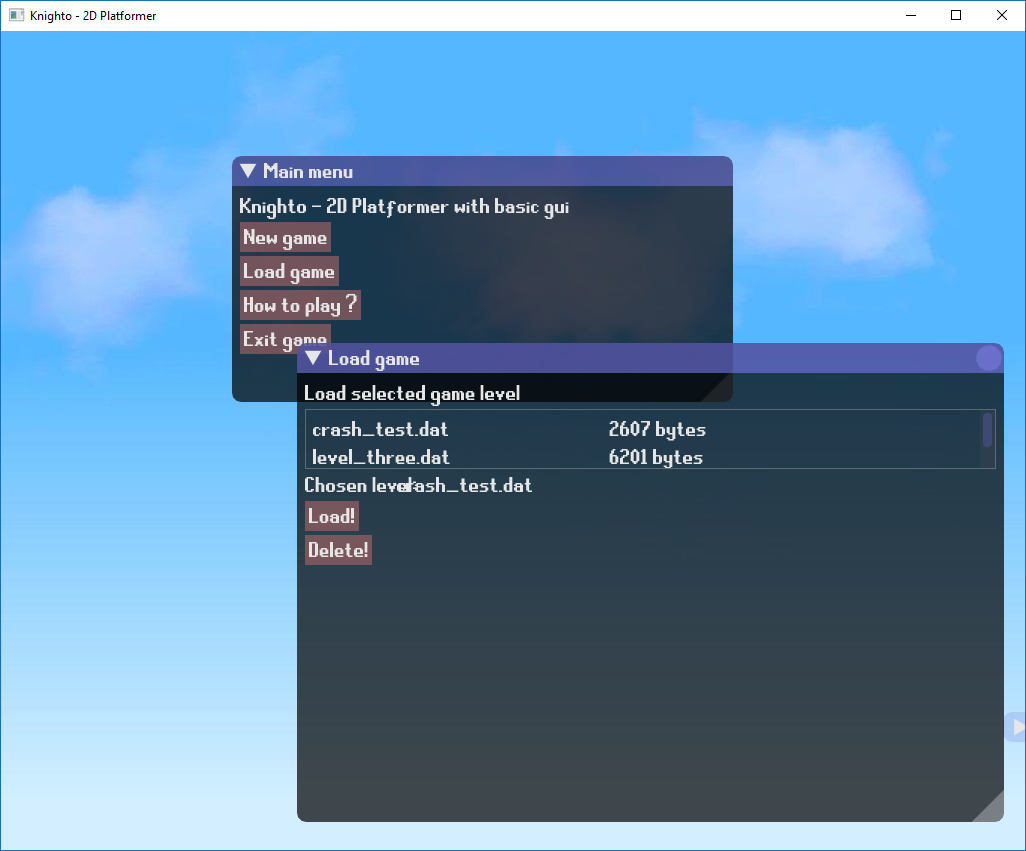
After launching the game using game executable Knighto.exe, the main menu appears. As shown in the screenshot the menu contains few options which will be discussed later. The options are:

* New game – in which we can create new empty level, with chosen name and size. The new level will contain only borders with brick tiles!
* Load game – in which we can choose on which level we want to play. The newly created level should also appear here. **Levels have .dat extensions.**
* How to play? – opens another window with basic instructions about how to play and control our character. (picture below).
* Exit game – closes the game window

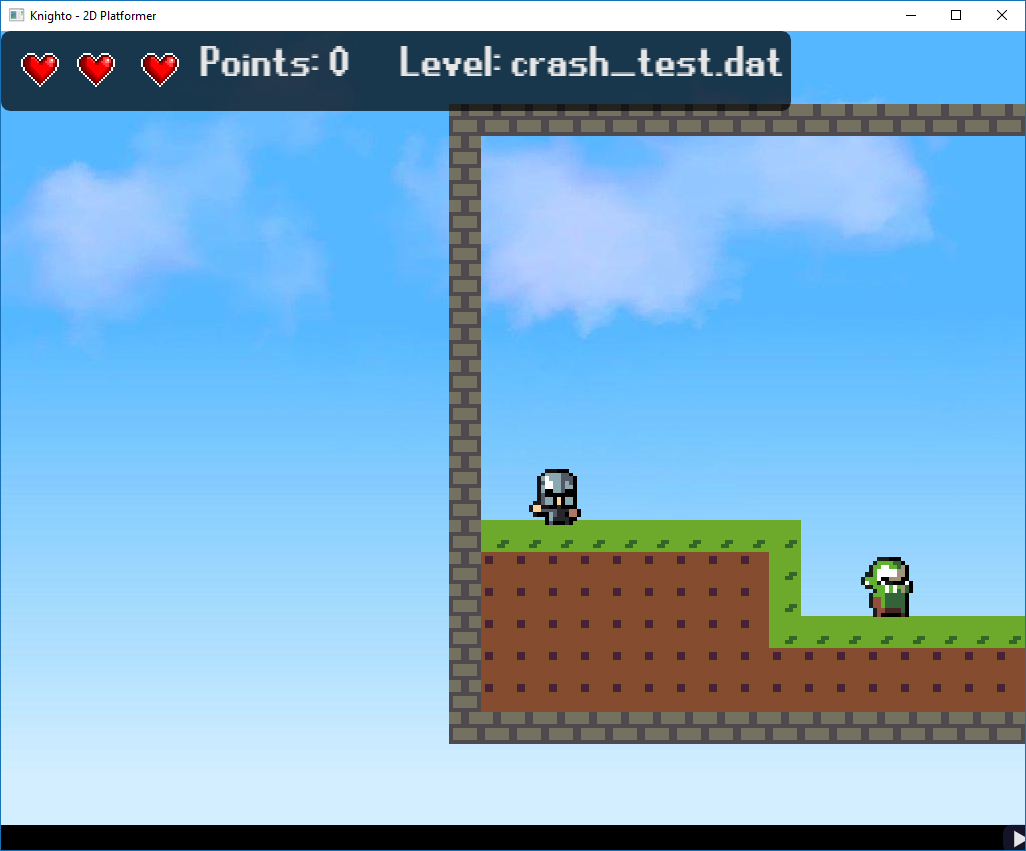


* 1. Load Game

Here as show below we have the load game window from we can choose which level we want to play, just click the level name and click the **Load!** Button.



* 1. Gameplay



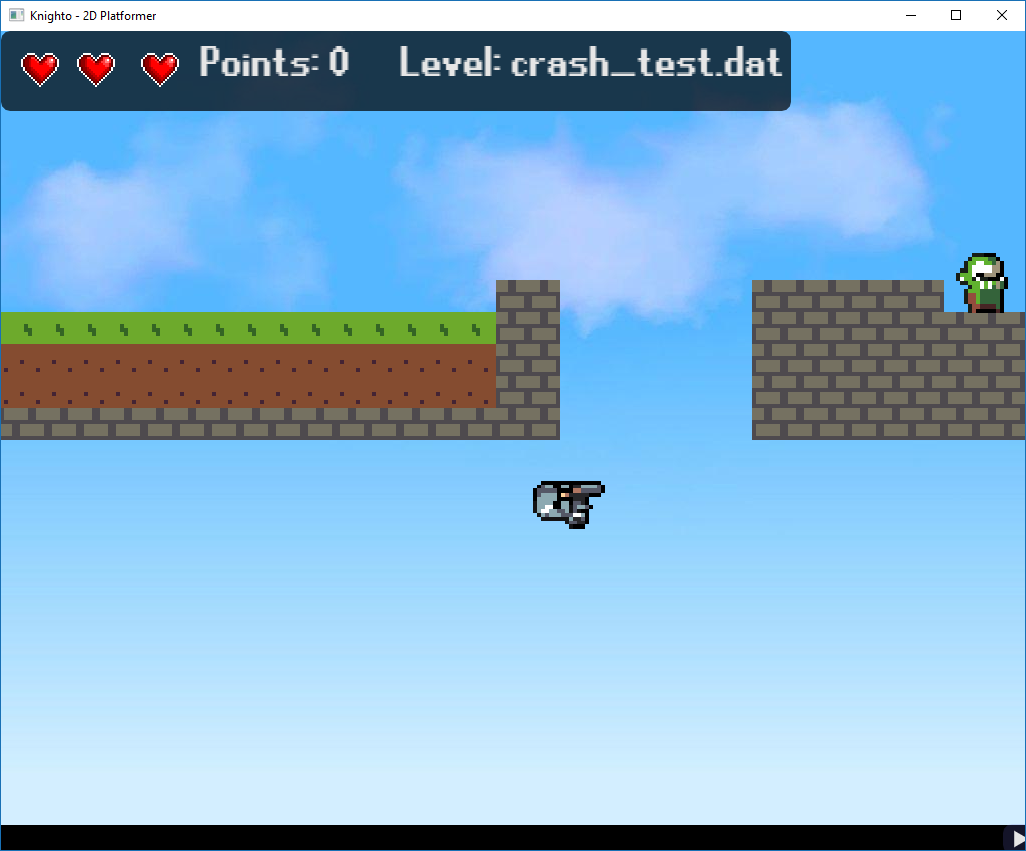
After loading the game, we now have our character, the brave knight Knighto! If u skipped the instruction window earlier, the controls for our character are:

Arrow buttons or A and D keys to move left and right

Space to Jump

Left Control or F to attack!

Our main goal is to get to the end of the level! On our road, are many obstacles like enemies Orc Shamans and the most dangerous one **gravity!** There are some gaps which lead us to easy death…



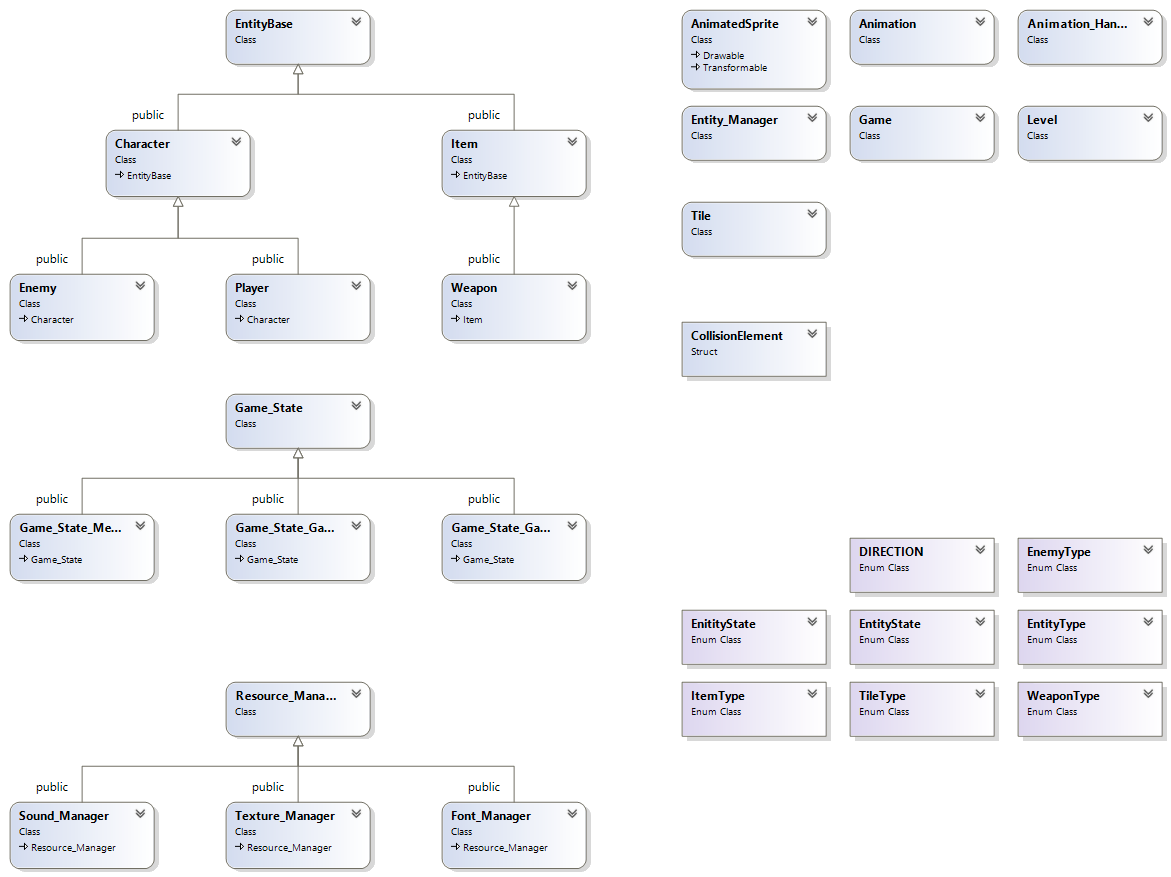
1.Poor knight and his jumps!

On other side we have some help from various chests and coins, chest can contain weapons, potions and other stuff for our knight! Coin increases our score which will be presented in the High Score table (to be implemented).

1. Internal specification

Project was designed using C++ programming language and coded in Visual Studio 2017 Enterprise Edition and compiled for 32 bit platform. As many classes was used in the project the internal specification will not explain them all.

* 1. Class diagram



* 1. Classed used:
* Resource Manager – this is a virtual class with load method, which takes two std::string arguments, which are the name of our resource for ex. knighto and the file name like knighto.png
  + Derived classes method getRef() – which returns reference to our asset
  + Derived classes
    - Texture Manager – manages sprites
    - Sound Manager – manages sounds and music
    - Font Manager – manages fonts
* Tile class – this class represents all the object from which the Level in game is built, tile is a square with its own ID, sprite, position and/or animation. In addition the tile has its properties like friction (which is needed if Player collides with it), visibility (spawns are also tiles but we don’t need to see them), solidity (air in the game is not solid).
  + Structures:
    - enum class TileType – each tiles has its type ex. GRASS , DIRT
  + Methods:
    - void draw(RenderWindow, float) – method needed to draw the tile using graphical library
    - void update() – method used to update the tile logic (for example if its destroyed or changed)
* Level class – this class contains our level in game, it holds all the tiles, monster spawns , chest spawns, item spawns. The level has its own name and size. Level class also contains the player position! If its not specified in game level file it will assume that the player starting position is (0, 0).
  + Methods:
    - void loadByText(std::string, tileAtlas) – this is text edition function for loading level, takes file name for arguments and the tileAtlas (its collection of all types of tiles), from file given it load the level size and all the tiles to the vector of tiles
    - void create(std::string) – it’s the function used in New Level menu, it takes name of newly created file as argument and creates it with specified size. The created level is easy to edit and use.
    - void update(float) – method used to update level logic
    - void draw(RenderWindow, float) – method used to draw the level (basically its just for executing draw() function for each tile)
    - Tile\* GetTile(unsigned int, unsigned int) – method that returns pointer to the tile at specified coordinates (x,y)
    - Tile\* GetDefaultTile() – method that should return pointer to default tile (friction only on X axis)
* Game State – this is class holds the pointer to actual game, it has virtual methods needed for the graphical library and gameplay.
  + Virtual methods
    - void draw(float dt) – needed for drawing all objects in our current game state
    - void update(float dt) – needed for updating the game logic (like movement of sprites)
    - handleInput() – needed for handling the player movement (keyboard support)
    - show\_debug() – shows debug window (not enabled in user mode)
  + Derived classes
    - Game State Menu – main menu game state
    - Game State Gameplay – actual gameplay game state
    - Game State Game Over – game over game state

To create new Game State all what is need is to create new derived class and push the object of this new game state to the game state stack.

* Entity Base - every object apart from the tiles is an entity. The Entity Base class holds all methods and variables needed for the exemplary entity, position, velocity, collision box. An Entity has its own EntityState, ID and many more. The derived classes are created to give the entity more specific parameters and functions. For example weapon or coin doesn’t need health like Player or Enemy. Each entity is assigned to the global Entity Manager.
  + Methods:
    - Getters and Setters because all variables are private, just use specific one for example getPosition()
  + Structures:
    - enum class EntityType - type of entity if none chosen the type will be Base
    - enum class EntityState – state in which entity is now if none chosen the state will be Idle
    - struct CollisionElement – structure needed for managing collisions of our entity and game world
  + Derived classes:
    - Character – class for the interactive objects like Monsters, Player
    - Item – class for non-interactive objects like Weapons, Potions, Coins
* Entity Manager – as the name states it manages all the entities in our game, the pointers to the entities are stored in a vector container.
  + Methods:
    - add(EntityBase\* e) – method for adding new entites
    - getById(unsigned int id) – returns the pointer to our entity by id
    - getByName(std::string name) – returns the pointer to our entity by name
    - enCount() – returns the number of entities in cointainer
    - remove(unsigned id) – removes the entity from the container
    - void draw(sf::RenderWindow, float) – method for drawing each entity
    - void update(float) – updates the logic of entities (for example change of position)
* Game class – main class in our game, it holds all the logic that is needed in our game.

The Game class is above all classes in the game, it manages the game loop, loads all the assets and stores it in specialized managers. All the managers and game states are stored here, that’s why we need to keep track of the game class pointer in Game State class in order not to lose the contents.

* + Methods:
    - loadTextures(), loadTiles() ,loadFonts(), loadSounds() – all these functions loads the assets to the specific manager respectively
    - pushState(\*state) – used to transit the game to another state, for example from Main Menu to Gameplay
    - popState() – used to go back to previous state or just to delete states
    - changeState(\*state) – change state to another by deleting the previous (with checking if empty) and pushing the new one
    - state\* peekState() – used to check the actual state or if its none (at the start of the application)
    - void run() – main game loop function discussed more specific below
  + Game Loop – void run() function

According to the pattern in game developing the game loop has its own clock and works in its own pace. That’s because if we don’t lock the speed of the game loop, the game would work faster on the better machine and slower on the worse which is not acceptable. By locking the game loop to specified time, we managed to keep the same pace of gameplay on both computers.

The game loop in my game works as long as the game window is open and follows this pattern (clock that measures our time for each cycle is created before the loop):

1. Source code

According to our tutor directions the source code of the project should be presented and available on the Github platform, link below:

<https://github.com/Etherion33/Knighto---2D-Platformer>

1. Testing

The project was run with series of tests, it has some issues listed below:

* Collision boxes may be inaccurate, shifted
* Main character have some problems with jumping naturally (movement handling)
* Orcs and other Monsters are too dumb (the don’t jump if there is an obstacle before them)
* Main character actions are not executed immediately

1. Conclusions

The project gave me opportunity to improve myself and motivate me to pursue my education in computer programming. The game itself it very simple and I had a lot of ideas how to improve it ~~add more bugs,~~ it helped to understand some concepts and discover new ones. I’m thankful to my tutor Roman Starosolski for giving me this chance and choose my own project. I’m now more aware of my errors and I will try to not do them again in the next project which I already have on my mind☺.