ECS7014P - Lab Exercises

Building a Simple Mini-Game

Lab Weeks 5-6 - Assignment 1-B (10%)

Lab Objectives

The objective of this lab is to improve your C++ skills and to implement two of the most important patterns in game development: Update and Command. You will also use the Game Loop pattern as implemented in previous labs.

In particular, in this lab you will:

- Implement a fully functioning mini-game where you control an animated 2-D sprite character.
- This game includes collision management, spawning of entities, timers and animations control, among others.
- You'll implement two versions of the Command pattern for handling the game's and the player's input.
- You'll implement the Update pattern with special attention put at the addition and removal of entities from a vector that holds all objects in the game.
- You'll implement all this using smart pointers, references, virtual functions and other characteristics of C++ such as `const`, `auto`, vectors and iterators from the Standard Template Library.

The code you produce by completing this lab corresponds to **Assignment 1-B**. Assignments 1-C and 1-D will build on the outcome of this lab, so by completing these exercises you will also be indirectly working towards those two assignments as well. Please read the instructions of the assignment to understand how this fits in the submission. You can also find some help for preparing this submission at the end of this document. Remember that you have the opportunity of submitting this assignment for a feedback round by **3**rd **March**. No late submissions for the feedback round are allowed.

Independently from you submitting a draft submission for feedback, you must submit the final version of the assignment by **24**th **March**. The draft submission will **never** be evaluated as a final submission.

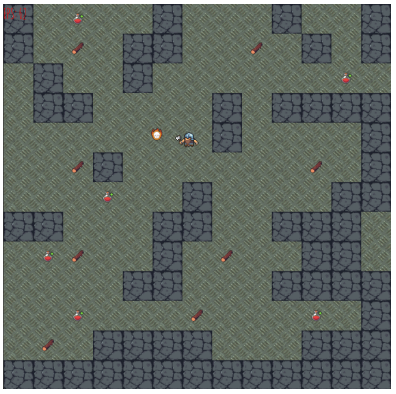
1 Before you start

The first step is to create a new Visual Studio 2022 solution with a visual studio project for the game. Configure your project to use SFML as we did for the previous exercise, including linking the libraries needed for using SFML and fonts (*freetype.lib*). Upload this solution to a new github repository, as you did for the previous exercise.

For this exercise, we provide several resource and code classes in QM+, which you should download now. Copy the subfolders and files from code.zip to the directory of the main code project in the Visual Studio 2022 solution you've just created, add the included files to the code project by type. This project should build with the code provided (it should run as well, but nothing will happen).

2 An overview: what's in the code and what's to be done

The objective of this lab is to build a mini game where you will control an animated 2D character and interact with several entities in the world. At the end of the lab, our mini-game will look something like the following:



The focus of the lab is on the gameplay logic, which includes input handling, updating game objects and managing their collisions. In order not to make this lab and assignment too long, several pieces of functionality are already given. These are accessory but necessary tools such as vectors, sprite sheet and 2D animation helper classes, etc. It's important to get familizarised with the codebase, therefore read this section carefully, and refer to the code itself when reading about the different pieces.

The following classes are complete and will need no modification through the exercise. Do take a look at them to see how some pieces of functionality have been implemented, and ask if you have any question:

- utils/Vector2.h contains a Vector2f class 2 float-type coordinates. This is actually quite similar to what you implemented in lab 1.
- There are 4 files in the graphics (include and src) folders:
  - Window: Provides a window for the game. This is also very similar to the window you worked on in previous exercises, with the only difference that adds an on-screen message to indicate that the game is paused.
  - SpriteSheet: reads and loads a sprite sheet asset resource specified in a particular format. You can take a look at the files img/DwarfSpriteSheet.png to see the sprite sheet and img/DwarfSpriteSheet_data.txt for the information on this spreadsheet and how to load it[1]
  - AnimBase and AnimDirectional are two classes used by the spritesheet to play and control animations.
- core/Board and core/Tile implement the level for the game. The level is a 2 dimensional grid (implemented in Board) of cells (implemented in Tile). Tiles can be of two different types, have a position in the world and can be drawn. The Board class holds the tiles allowing setting them in the grid and drawing them all on the screen.

The following list of classes are part of the code that you will be working on. They are partially provided, with gaps that you will be filling in the next sections. This is what they (will) do:

- utils/Rectangle.h: An API for a rectangle class, formed by two vectors (top left and bottom right). You will implement the arithmetic operations that determine overlapping and inclusion of points and vectors.
- There are 5 classes that represent the different entities that can be present in the game:
  - Entity (Entity.h and Entity.cpp): it's the base class for all entities. Defines a class enum with all entity types. Each entity has several class members and functions to access and modify these (such as position, velocity, size, sprite, etc). In this class, you will mainly work on the update and drawing of the entities.

[1] This code and format is adapted from the one in the book "SFML Game Development By Example (Raimondas Pupius)". You can take a look at this book to learn more about this implementation.

  - Player (Player.h and Player.cpp) implements the entity controlled by the player of the game. This player has certain member variables and constants that customize the player's behaviour. In this class, you will mainly work on the update of the entity and handling the input given by the player, implementing a **Command pattern**.
  - Potion and Log (both in StaticEntities.h) are two types of entities that do not move during the game. In this classes, you'll take care of their initialization.
  - Fire (Fire.h and Fire.cpp): This is a type of entity spawned by the player when sufficient logs have been chopped. Fire travels in a straight distance with a time-to-live timestamp that makes it disappear after a few frames. In this class, you will work on the update of the fire object, which includes determining its movement and when it must be destroyed.

- core/Game.h (and cpp): this is the core class for this game, which holds its entities, a reference to the Window object and the classical functions to update, handle input and render the game. In this class, you will implement the code that handles user input (again, another instance of the Component pattern), bits of the game initialization, managing entities, updating and rendering them.
- main.cpp: the entry point of the game. The `main` function loads the file that contains the level to be loaded, creates the game and runs it. You will implement here an adaptive game loop (which you can bring from the previous lab).

After you've completed this exercise, you will have a working mini-game as described above. This game does not follow any particular Entity-Component(-System) architecture. In fact, it will not be very flexible for further modifications. In the following labs, we'll introduce the concepts and modifications that follow the Entity-Component and Entity-Component-System architecture principles (as seen in the lectures) to improve the quality of the code and bring it closer to industry standards.

But, for the moment, let's make a game. Shall we start?

3 Our game loop

For this and all subsequent sections, tasks are indicated accompanied by comments in the code with the format *RomanNumber.Letter*. This way, you can easily match the instructions indicated here and the place in the code where the required functionality needs to be implemented[2].

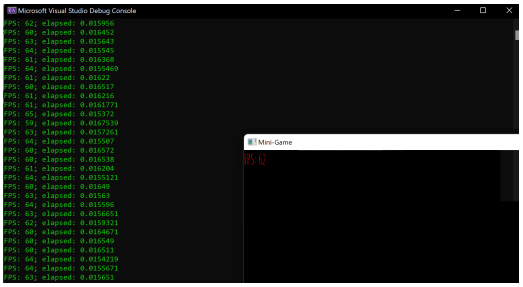
The first part of the exercise takes place in the main.cpp file, and it's pretty simple: the objective is to write a framerate adaptive game loop that calls the main functions of the game on each iteration (input, update and render).

As you can see, `main.cpp` gives you the skeleton for this. It first loads the level file and then creates and calls the function that initializes the game. Then, it calls the function *adaptiveLoop*, which is the one that you need to fill. In this function, look for the comments I.A, I.B, I.C and I.D, which indicate what needs to be done for this part.

Do what the comments say now. You can bring the code from the game loop that you implemented in the previous assignment.

Once this is implemented, you should be able to build and run your code. You should then see an empty window being displayed, with an FPS text (which should give a value ≥ 60). The console should also print the FPS and elapsed time on each frame. Something like the following:

[2]You can search all documents for a text by pressing 'Control' + 'Shift' + 'F'.



4 Drawing the Scene

The first thing we need to do to draw our scene is to build the board to be drawn. Take a look at the function `Game::init`, which receives a vector of lines that have been read from the *levels/lvl0.txt* file:

```
w.p..w...w..w
w.x.ww..x.w.w
.w..w...x..p.
.ww..*.w.wwww
.......w....w
..xw......x.w
...p..w....ww
ww...ww.xwww.
.px..w.x..ww.
....ww...wwww
..p...x...p.w
.x.wwww...www
wwwwwwwwwwww
```

The `Game::init` function runs through the lines of text adding tiles and entities to the board. The switch statement in this function shows what does each character correspond to. There are three types of entities defined in the board (potions, logs and the player) and two types of tiles (walls and corridors)[3]. Note that objects ('*','x' and 'p') automatically add a tile of type 'corridor' underneath them.

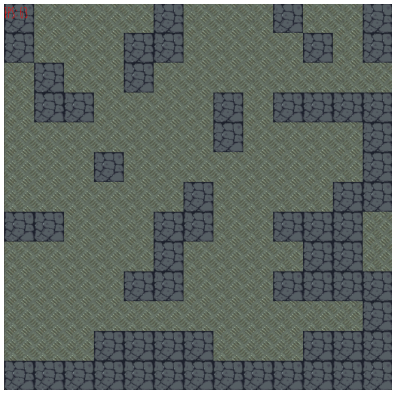
4.1 Drawing the board

Let's start by creating and drawing the board, only processing the background tiles:

- II.A: We need an object of type Board in the Game class. Start by declaring a unique pointer of type Board as a private member of the class Game. The recommendation is that this member is called 'board' (we'll refer to it as such during the rest of the exercise).
- II.B: This object needs to be instantiated once we know the dimensions of the board. You can see that, in the `Game::init` function, we obtain this information on the first iteration of our parsing loop. Create this pointer after width and height have been determined (find comment II.B).
- II.C (×5): Each 'case' of the switch statement must add a tile to the board, so there's a tile on every position. Use the utility function *addTile()* from Board to add a tile of each corresponding type (*wall* for case 'w', a *corridor* for the rest)
- II.D Finally, we need to actually tell the game to draw the board. In *Game::render()*, add the call to the *draw* method of the Board class, passing a pointer to the Window object.

After adding this code, if you build and execute, you should see that the window now shows an empty board showing only the tiles:

[3]In this implementation, there's actually no functionality associated to these tiles. They are both simply background.



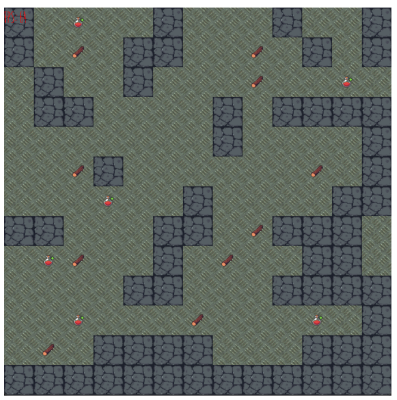
Pay special attention to where these tiles are placed, as they should resemble the figure above. If they're mirrored, you've likely mixed up columns and rows!

4.2 Adding game entities

The next step is to add the different entities to the game:

- We'll start with logs and potions, which correspond to the cases 'x' and 'p'. For doing this you need to call the *templated* function `buildEntityAt` in the class Game. Follow this for logs (point III.A) and potions (III.B).
- Note that the function `buildEntityAt` calls the **init** class of the Entity created. As Log and Potion (in StaticEntities.h) derive from Entity and the function init is overriden in the subclasses, these functions at the respective classes will be called first. Log and Potion will be able to do their own initialization in these classes, and call the init class on the parent Entity class for common initialization operations for all entities. Do the latter in point III.C ($\times 2$) for both classes.
- Next, we'll create the data structure needed to manage these entities. We need a vector of Entity shared pointers (point III.D, Game.h) and a counter to give entities an ID (III.E). This member variable to count game entities needs to be initialized to 0 in the Game constructor (III.F). We also need to implement getter functions for the entity count (point III.G) and retrieving an entity from the vector given its index (III.H);
- The function addEntity in Game must be implemented now (point III.I).
- We now need to render the entities on the window (III.J): add a loop to the *render* function of Game that iterates through all entities in the vector and calls the *draw* method on each entity.

Once this is all working, executing the mini-game should now show the following window, which includes logs and potions:



4.3 Adding the player

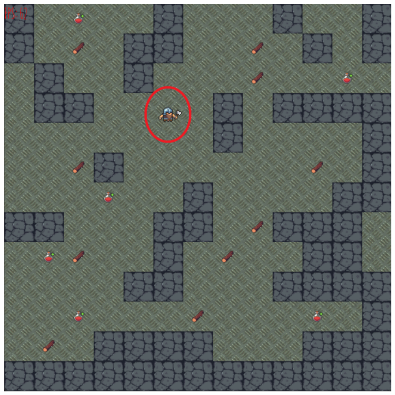
The next step is to add the player to the game. Follow the next steps associated with the corresponding points in the code (IV):

- IV.A (×2): Declare a pointer to a Player object in the Game class and add a public member function to Game that returns the shared pointer of the player object.
- IV.B (×4): Initialize the player in the Game::init() case '*'. First create the pointer (1/4), then call the function that builds the sprite sheet (2/4), positions the sprite in the board (3/4) and adds the entity to the vector of entities in the game (4/4). Note that, different to other entities in the game, we are going to have a pointer to the player in two places: directly in the Game class (as a member) and also in the vector of entities (as well a member of Game). It's convenient to do so as this player is a member we need to access very often, and having to look for it every time in the vector of entities would be a computational waste. Note these pointers point to the **same** object.

If you build and run the code now, you should see that the player has been added to the (approximately) center of the board. The reason it's being drawn is because it's part of the entities vector, and we are already rendering all the elements in that container. However, the player uses an animated SpriteSheet, and as you can appreciate the sprite is not currently playing any animation. We should be playing the default animation ("Idle") for the player.

- IV.C: In order to animate the player, write a **while** loop in the update function in Game that runs through all objects in the entities vector and calls the method update on each one of them. For this loop, use a standard template library (STL) **iterator**, rather than a "traditional" *for* loop (which uses initialization, condition, increment of an iteration counter variable).
- IV.D (×2): Next, we implement the update function in Player.cpp to update the spritesheet. First, as done for the static entities, add a call to the base class of Player in Update (IV.D 1/2). Then, update the spreadsheet in the Entity class (which is the one that holds the SpriteSheet object), in the update function (IV.D 2/2). Although we are only concerned with the player's spritesheet, we must also add here the case for when the entity uses a sprite, rather than a spritesheet, at this point.

Once this is all working, executing the mini-game should now show the following window, which now includes also the player, **animated**:



5 Game Input

We are now going to implement an instance of the **Command** pattern to manage the input for the game. We'll start adding one instance of this pattern to control the pausing feature.

- Create the infrastructure for the command pattern. As seen in the lecture, we need a class in charge of retrieving the input from the user (*InputHandler*), a class to manage commands (*Command*) and derived classes that handle each different command (in this case, only one for pausing the game, *PauseCommand*).

- *Command*. Create the file core/Command.h. In it, declare the **new** class Command, which will be a pure virtual class that serves as a base to all game commands. The class must have a virtual default destructor and a pure virtual function to *execute* the command, which returns `void` and receives a reference to a Game object.
- *PauseCommand*. Declare this **new** class in core/Command.h, which inherits from Command and overrides the *execute* function. The definition of the *execute* function for PauseCommand should be in a new cpp file: core/GameCommand.cpp. Every time this function is called, the value of the game's pause variable is toggled from *true* to *false* and vice-versa. Implement this in the *PauseCommand::execute(Game& game)* function, calling the appropriate method from Game.
- *InputHandler*. Create two new files: core/InputHandler.h and core/InputHandler.cpp. Then, declare a **new** class with the following members (write the member declarations in the header file and the definitions in the source .cpp file):
  1. A private member variable of type shared pointer to a Command object (for pausing the game). You will need to add a forward class declaration to the class Command.
  2. A public default constructor. Note that this constructor needs to initalize the pointer to the *PauseCommand* object.
  3. A function that handles the game input and returns a (shared) pointer to a command object. For now and for the sake of simplicity, simply hard-code the 'Escape' key-press event in this function, so the pause command is returned when this key is pressed.

- V.A: Our Game class needs a pointer to input handler. Create it as a private unique (as it doesn't need to be shared with any other class) pointer of the Game class. Note that you will have to add an #include directive in order for the compiler to know where this InputHandler class is.
- V.B: This pointer needs to be created in the constructor of Game. Add it now.
- V.C: The function handleInput in Game is where our Game class handles the user input and executes the respective commands. Add this functionality to this method. Note that again you may have to add an include directive to Game.cpp so you can use Command objects in this file.
- V.D: We will handle (in the next section) the input of the player in the function *Player::handleInput()*, so let's add a call to that function from *Game::handleInput()* now.
- V.E: In our implementation, game entities should only be updated when the game is not paused. Add this condition to the *update* method in Game.

You should be able to build and run the application now, and pause it (you'll see a blue "Paused" message as shown in the next image) when you press the "Escape" key. When the game is paused, the player's sprite should not be playing any animation at this point[4].

6 Player Input

Our next task is to implement another version of the Command pattern for handling the player input. The fact that we use the Command pattern doesn't mean that we have only to use it *once*, nor that they need to share the same code/implementation. In this case, we want to have a different implementation for the input that supports multiple keys to be pressed at the same time, something that the implementation of the pattern

[4]We'll actually find a better way to do this when we build an Entity Component System later on.

we used for the game's input does not let us do.

Let's start by creating an input handler for the player:

- Start by creating a new subclass of Command to capture a movement (for instance, *MoveRightCommand*). Declare this class in core/Command.h.
- Create a new file (core/PlayerCommand.cpp) where you will define the functions for the player commands.
- Define the execute method of *MoveRightCommand*, which will simply set the players velocity X coordinate (using *Player::setVelocityX()*) to $1.0f$.
- In core/InputHandler.h, create a new class for the player's input handler (possible name: *PlayerInputHandler*). For now, create this class like you did with the game input handler: a constructor (that initializes the move right command), a member variable for a the command object that moves the player to the right, and a function to handle the input and return the appropriate command object. This function must return the *MoveRightCommand* pointer when the key "D" has been pressed[5].
- VI.A (×2): Declare a smart pointer to a *PlayerInputHandler* object in the Player class (1/2). Again, this pointer can be of type unique, as we don't need to share it with any other object. In order for the compiler to understand what *PlayerInputHandler* is in this class, you must add a forward declaration to the class PlayerInputHandler (2/2).
- VI.B: Create the unique pointer to the *PlayerInputHandler* object.
- Now we need to handle the input and make the player move.
  1. VI.C: The place to do the first step is in the function *handleInput* from the Player class. In here, similar to how you did for the Game input handling, call the input handler method to retrieve a command to run, and execute it. You'll also need to add two include directives to Player.cpp so the PlayerInputHandler and Command classes are visible here.
  2. VI.D: This will set the velocity X component of the player to 1.0 when pressing the "D" key, but we are still not using this to move the player. Movement is not something that only players do, so the functionality for this should be placed at the Entity level (remember we're already calling *Entity::update()* from *Player::udpate()*). Go to the Entity's update function and use this variable to change the position of the entity.

Now you should be able to build and run the application. If you press "D", you'll see that the player moves to the right ... however, when you release the key, the player keeps moving to the right. Additionally, when the player moves, the sprite is still playing the "Idle" animation, rather than the "Walk" one. Let's fix this:

- VI.E: The velocity member in Player must be reset at every frame. One possible place to do this is at the top of the function *handleInput* in Player.
- VI.F (×2): In *Player::update()*, set the animation of the spritesheet to "Walk" when the x component of the velocity vector is positive, as well as the direction the sprite should be facing (1/2). Also, the player should go back to the "Idle" animation when the player is not moving (2/2).

At this point, the player should be able to move to the right when pressing the "D" key, When moving, the sprite should be playing the "Walk" animation, and be back to playing "Idle" when it stops.

The next step is to repeat what you've done for the input "Right" for the other three directional movements: "Left", "Up" and "Down":

- Create the Command classes for the other three movements, as subclasses of the "Command" base class. These classes must implement, in their respective *execute()* methods, the appropriate changes in the player's velocity vector, as *MoveRightCommand* does.
- Add commands for moving left, up and down in the PlayerInputHandler class.

[5] You've probably guessed now that we are defining a W-A-S-D movement for the player.

- Add the necessary code to the function in *PlayerInputHandler* that creates command objects so the appropriate commands are created and returned when the keys "A", "W" and "S" are pressed (for "Left", "Up" and "Down" movements, respectively).
- The function *Player::handleInput()* should **not** need any modification at this point to accommodate for new commands, as it's calling the execute function for any Command received from the *PlayerInputHandler* object.
- VI.G: Update Player::update() so the changes in the velocity vector translate into the proper animations that must be played with each movement.

At this point, you should be able to move in all directions, keeping the orientation of the sprite updated to the left/right depending on the direction of movement. If the player stops, the animation played by the sprite will be "Idle".

7 Simultaneous and other action commands

At the moment, *PlayerInputHandler* returns only one command. This imposes a limitation: we can't move the player in diagonals when pressing a combination of inputs for the two movement axis (for instance, when pressing "W" and "D" simultaneously). The following steps are a possible way of addressing this:

- Change the function in *PlayerInputHandler* that handles the input and returns a pointer to a Command object, so it now returns **a reference to a vector of Command shared pointers**. Note the following:
  - We return a reference to a vector (rather than a vector) to avoid copying this structure when returning from the function. Returning a reference is more efficient.
  - We must never return a reference to a local variable, as that would incur in an undefined behaviour. Therefore, the vector of commands should be a **member variable** of the *PlayerInputHandler* class. Create this member variable in this class now.
  - The input handling function must clear this vector at the start of the method.
  - In this same function, rather than returning a command pointer, these pointers must now be added to the vector. Finally, the vector should be returned at the end of the function.
- VII.A: The Player's *handleInput* function now must handle a vector of Command pointers, rather than a single pointer. Modify this so the code iterates through the vector of commands and executes them all.

This should allow you now to move in diagonals by simultaneously pressing keys in the two directional axis.

Finally, we want our player to be able to do two more things: collect wood by chopping it with the axe, and to shoot fire only when a certain amount of wood has been collected[6].

- Add another command to the input management system for the player so it plays the animation "Attack" when the player presses the "Space" key.
- Similarly, add a final command for the player so it plays the "Shout" animation (which in our case will mimic a shooting action) when the *Left Shift* key is pressed.
- Note a few differences with the previous commands:
  - Rather than modifying the velocity coordinates, the new Command classes must modify the `bool` flags for *attacking* and *shouting* in Player. These should be set to `true` (unless they are already `true`).
  - These two animations are one-shot (i.e. they should not *loop*). This not only affects the way they are played, but also we need to set these "attacking" and "shouting" flags back to false when the animation has finished playing. You can know this by accessing $spriteSheet.getCurrentAnim() \rightarrow isPlaying()$ from Player. Set these flags back to **false** in the Player's update function (point VII.B).
  - In Player::update(), play the respective animations when the bool variables *attacking* or *shouting* are `true`. Think what should be the order in which the Player's member variables should be checked in Player::update(), so the shouting and attacking animations take preference over the walking and idle ones.

[6] Nobody said that this game should actually make sense ...

8 Setting some colliders for our entities

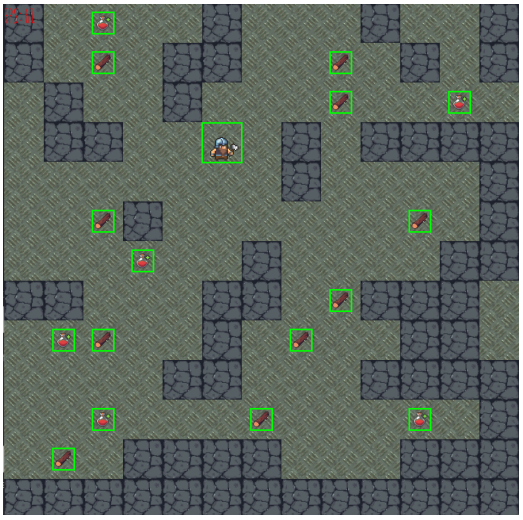
Our next task is to implement a collision system that allows the player to pick up potions and (eventually) collect resources from logs. The first thing we need to have a collision system is count on **collision boundaries**. In our simple mini-game, we'll use 2D boxes (rectangles):

- VIII.A: The boundary box is an object of type Rectangle, owned by the Entity class. We just need to update its position (2 vectors: top-left and bottom-right corners of the rectangle) in the *Entity::update()* function.
- VIII.B: We can actually draw the bounding box on the screen to visualize collisions better. In *Entity::draw()*, draw the boundary box.

If you build and play-test the game, you should now be able to move the player around and see the bounding box in a green outline, which also moves together with the player around the level. We just need to add now the bounding box to the static entities:

- VIII.C ($\times 2$): Given that static entities don't move, we don't need to update the position of the bounding box on the update method. Instead, set the top left and bottom right corners of the bounding box in the Init method of the classes Log and Potion.
- The code for drawing these colliders is located at the Entity level, so we don't need to add it for these boundaries to be shown on screen.

Play-testing again should lead you to seeing the bounding boxes for all visible entities: Player, Logs and Potions. This is how it should look like at this point:



9 Simple collision management

Our next step is to make collisions actually happen. We'll implement this entirely on the Game class, in the update() function. However, before doing the collision management, we must complete a couple of methods in Rectangle that will help us determine if two rectangles (i.e. bounding boxes) overlap.

- IX.A: In *Rectangle.cpp*, implement the function *inside()*.
- IX.B: Finally, implement the function *intersects()* in *Rectangle.cpp*. Note that you can use the inside() function to compute this.

Now that the utility functions are implemented, we can implement the collision management. Given that, in this game, the player is the only entity that collides with anything else, rather than compare the bounding boxes of all entities among themselves, we'll just check collisions of the player's bounding box with those of the static entities.

- IX.C: Retrieve a reference to the player's bounding box and run through all *other* entities in the game.
- IX.D: Check for the intersection of the player bounding box and that from the other entity.
- IX.E: Write a **switch** statement that determines the type of the object we are colliding with, in case there is an intersection between the two bounding boxes. You should have two case statements, one for *EntityType::POTION* and another one for *EntityType::LOG*.
- In the case we are colliding with a potion, simply add the following text for now:

```
1 std::cout << "Collide with potion" << std::endl;
```

  Similarly, if you are colliding with a log, add the following:

```
1 std::cout << "Collide with log" << std::endl;
```

This should be enough for now to move around and collide with the items in the game. Play-test this and verify that the messages are printed to console at the appropriate times.

Let's now make the collision with the simplest of the objects: the Potion. As you can see, potions are able to restore some characteristic "health" of the Player. Potions have a default restore value (*Potion::potionHealth*), which is a constant integer set to 10. You can access this value with the function *Potion::getHealth()*. Player also has a member variable *int health*, and the Player class provides two methods for retrieving and adding values to this health (respecting a maximum value: another constant *Player::maxHealth* set to 100). The goal now is to allow the player to collide with a potion, let it restore health and remove the potion from the list of entities.

In IX.F (inside the case *EntityType::POTION* in the collision switch):

- In order to add health to the player, we need to call *Potion::getHealth()*. However, your vector of entities contains pointers to Entity objects (which don't have a member function *getHealth()*). Therefore, you need to do a *dynamic_cast* from `Entity*` to `Potion*` before you can call this method. Do this cast[7] now and retrieve the health value of the potion.
- Add the health restore value of the potion to the player using the *Player::addHealth()* method.
- To make our console output message more informative, modify the "Collide with potion" text so it displays the health restore value of the potion and the player's health value after it has been modified.

Build and play-test. You should now get the new message with the values of the player's health at 100. Let's now take care of chopping logs[8]. Note that we are going to react to the collision only when the player is playing the animation "Attack" and the animation is "in action". The implemented sprite sheet animation system allows you to identify specific frames of a given animation as being "in action", during which the function *AnimBase::isInAction()* returns `true`. In our case, the "Attack" animation has 7 frames, and the "action" frames (when the ax *hits*) are frames 4 and 5. See, from *DwarfSpriteSheet_data.txt*:

$\#|Type|Name|StartFrame|EndFrame|Row|FrameTime|$**FrameActionStart**|**End**|
Animation Attack 0 6 2 0.1 **4 5**

Detecting specific frames in an animation allows us to create effects at particular points of the animation. We want the collision between Player and Log to take effect **exactly** when the ax drops in the "Attack" animation, not when the animation starts.

In IX.G (inside the case *EntityType::LOG* in the collision switch):

- Repeat the steps taken for potions for the case of the log: Cast the `Entity*` to a `Log*`, retrieve the wood value of the log (*Log::getWood()*) and add wood to the player (Player::addLog()). Change also the console output so it prints the wood collected

[7] Note that you need to cast the raw pointer from the shared pointer, for which you need to use the method *get()*.
[8] As with potions and health, logs allow the player to collect "wood".

- Wrap the above functionality with an **if** statement that checks if the player is currently playing the "Attack" animation and this animation is "in action".

Build and play-test. You should now also get the log printouts but only when the player "attacks" the log and, in particular, when the ax drops during the animation.

10 Removing entities from the game

Of course, we are not deleting the potions and the logs yet, which means that we are effectively "collecting" health and wood from them every frame. As seen in the lecture, removing elements from a vector while iterating it is normally not a good idea. We'll implement this by adding a *deleted* flag to the entities.

- X.A: First, let's add the entity flag for deletion. Add a private member variable to the class Entity of type *bool*.
- X.B (×2): Initialize it to `false` in the **two** constructors of the Entity class (in the initializers list).
- X.C: Add two helper functions to the Entity class for returning the value of this flag (i.e. *isDeleted()*) and another one for setting the flag to `true` (i.e. *deleteEntity()*).
- Back to IX.F and IX.G in Game::update(), call the new function that "deletes" the entity in case of a collision with either a Potion or a Log.

This should be enough to "mark" entities as *deleted*, but we are not factually removing the entities from the vector of entities yet. Let's do that:

- X.D: Write a loop that iterates through all entities and removes them from the vector of entities if their "deleted" flag is `true`. To remove an element from a vector, you can use the function *std::vector<T>::erase()*, which receives an iterator as a single parameter.

Once this is implemented, you should see after building and play-testing that potions and logs disappear on collision (logs, particularly, when playing the "Attack" command, in the "action" frames).

11 Spawning entities to the game

Our last bit of functionality consists of shooting fire when executing one of the commands we've partially implemented so far: Shouting. Our player will shoot fire when playing the "Shout" animation, only if the amount of collected wood is above a certain shooting cost. We'll also implement a cooling down system so we don't create multiple fire objects in consecutive frames. The idea for this Fire object is that it moves in a straight line until, after certain time has passed, it disappears.

We'll start this in the *Player::update()* function, in point XI.A:

- The function *Player::createFire()* builds and returns a Fire object. Call this function to create a Fire object here and add it to the game by calling the function *Game::addEntity()*.
- After this, we need to reduce the wood used to creating this entity. The "cost" of firing is set in a constant member of Player called *shootingCost*. Subtract this shootingCost from the amount of wood the player has at this point.
- Wrap the above functionality in an if statement that only executes this fire creation if the following three conditions apply:
  1. The player is shouting: we should have the Player::shouting variable set to `true` (we did this earlier in Section 7) if we are pressing the 'Left Shift' key.
  2. The player is playing the "Shout" animation and the animation is "in action". Like with chopping logs, we are shouting at an exact frame of the "Shout" animation.
  3. There is enough wood to create one Fire object (i.e. *Player::wood* $\geq$ *Player::shootingCost*).

Finally, we'll add cooling down system so there's no consecutive Fire creation with only one key pressed. Three steps in XI.B (×2):

- (1/2) Every time a Fire is created, set the member variable *Player::shootCooldown* to the cooldown time, defined as a const member of Player called *shootCooldownTime*. Then, add another condition to the shooting **if** statement that only allows shooting when the *Player::shootCooldown* value is ≤ 0.
- (2/2) At every frame (independently if we are shooting or not), the *Player::shootCooldown* needs to be reduced by the elapsed time since the last game frame, if this cooldown time is positive.

If you play-test and try to shoot (after you've chopped some logs), you'll see that a fire object is created, but it's not moving yet. Let's fix this:

- XI.C: (in Fire::Update()): call Entity::update() for its regular update call, same as we did for Player::update() in point IV.D (1/2). This should be enough for shooting fire that moves until it goes off screen (play-test this!).
- XI.D: Fire objects have a time to live member variable (*Fire::ttl*) that it's intialized to a constant value (*const int Fire::startTimeToLive*) upon construction. Fire::update must reduce *ttl* by 1 at every frame, and delete it (set the *deleted* flag to `true`) when *ttl* is reduced to 0. Note that our previous implementation to delete entities from the game should automatically pick this up and remove if from the vector of entities in Game at the end of the frame when this is done.

Build and play-test. Now you should be able to chop some logs, shoot some shouted fire and observe the fire traveling in a straight line direction (with its collision) until it disappears after certain number of frames has passed. Note that you shouldn't be able to infinitely shoot (due to *ammunition*, which you collect chopping more logs), nor shooting twice without a certain cooldown time.

If you can get this behaviour, congratulations! You've reached the end of the lab.

Preparing your assignment 1-B submission

For the assignment submission, you are asked to fill and upload a form to QM+, as you did for the previous assignment. Check the document in QM+ to see what do you need to include in this form, which also includes the marking criteria.

The code is "submitted" as a github repository link. Rather than asking you for the link to the repository, we require that you publish a GitHub ***private release*** (check the GitHub documentation[9] for instructions on how to create one). Note that, once a release has been created, further changes to the code will not be captured by that release.

In order for us to be able to give you feedback, you need to **give us access** to your repository. We are assuming you are using `https://github.qmul.ac.uk/`[10], therefore you'll have to give access to the following accounts: **eex516** and **acw634**.

As you can see in the assignment form, the code is evaluated attending two categories: implementation and C++ correctness. Additionally, you are asked to a couple of questions about this exercise.

[9] `https://docs.github.com/en/repositories/releasing-projects-on-github/managing-releases-in-a-repository`

[10] You may also be using `https://github.research.its.qmul.ac.uk/`, which is ok. If you are not using any of these two, you should. Any problems, let us know **as soon as possible!**