ECS7014P - Lab Exercises
Advanced Game Development
Lab Weeks 11-12 - Assignment 2 (40%)

Objective: The objective of this assignment is to enhance the Entity-Component-System architecture you implemented in Assignment 1. In this assignment, you are offered several features to complete this design, and with each one of them the number of marks associated to each feature. The amount of marks per feature are indicative of the difficulty of the implementation. There is no limit in the number of features that you can implement (but the mark will be capped at 100%).

Previous code: You can re-use your code submitted to previous assignments. The mini-game must be **fully functional** for you to complete this assignment (in other words, you should've completed, successfully, *at least* up to assignment 1-B). A non-fully functional game will incur penalties. You **don't need** to have an EC (assignment 1-C) or an ECS (assignment 1-D) architecture completed, but you're welcome to do so[1].

Submission (deadline 26^{th} May): As in previous submissions, you must fill and upload a PDF form to QM+. Check the document in QM+ to see what do you need to include in this form, which also includes the marking criteria. As before, the code is "submitted" as a github repository link (at `https://github.qmul.ac.uk/`). We are again asking you to publish a GitHub ***private release***. Note that, once a release has been created, further changes to the code will not be captured by that release. As usual, make sure the accounts **eex516** and **acw634** have write access to the repository.

Features: The following list shows all features (and marks, up to 100%) that you can implement for this assignment. As a rule of thumb, it's better to program fewer features correctly than many features incorrectly (i.e. you will not get high marks for features that are not correctly implemented).

- **Entity Component System Architectures:** If you implement multiple ECS architectures, include a new parameter in the constructor of Game that indicates which of the implemented ECS architectures the game should run with, so it's easy to change from one to the other. You're recommended to create a new *enum class* for the type of this parameter, and also to extract all ECS data and logic from game to a separate class, for clarity.
  - **Big Array (10%):** If you completed Assignment 1-D, you've already got this. If not, complete 1-D.
  - **Archetypes (15%):** First, identify the archetypes out of the entity types we have. Each archetype should be a vector of entities. Then, identify which systems must operate with which archetypes (see slides from Week 4), which in turn iterate through their entities at every frame.
  - **Packed Arrays (25%):** You don't need to implement *groups* and you can implement a simpler version of a Packed Array that only has a sparse array and 1 dense array of entities (i.e. you won't need to have an extra dense array for components).
- **Game Programming Patterns:**
  - **Flyweight (10%):** Implement the Flyweight pattern for the tiles of the board. The class Tile should be separated in two: i) the intrinsic data (the tile's texture) must be put in a separate class and only one instance (per tile type) should be created in the game. You can store the instances of this new class in Board. ii) the extrinsic data (sprite, position, type) should stay in Tile and a shared pointer to the new intrinsic object substitute the Texture member variable in Tile.

[1] Not only it'll be easier to you to implement the features here (that's the whole point of an ECS), but you'll complete the Big Array ECS by doing so.

- **Observer (10%):** Implement two achievements using the Observer pattern: one for collecting all potions in the game and another for shouting 5 times. When these achievements are granted, simply output a message in the console indicating it. When working on the *Subject* of this pattern, you may not need to have a collection of observers, so your *notify()* method would not need to iterate through a series of observers to implement this functionality.
- **Service Locator (25%):** Implement an Audio Manager using the Service Locator pattern. Sounds must be played in three situations:
  1. When a potion is picked up.
  2. When fire is shot.
  3. When the player attacks with the axe, but only when the animation is "in action".

  A few tips about implementing sound with SFML:
  * See SFML documentation: `https://www.sfml-dev.org/tutorials/2.5/audio-sounds.php`.
  * You will need to add one or more of the following libraries to the Input configuration parameter in Visual Studio: *sfml-audio-s-d.lib* (for static debug linking), *openal32.lib*, *vorbisenc.lib*, *vorbisfile.lib*, *vorbis.lib*, *ogg.lib* and *flac.lib*.
  * Copy the *openal32.dll* file from SFML (located at SFML-2.5.1/bin/openal32.dll) to the directory where your .exe is generated after building the solution.
  * SFML sound works similarly to textures and sprites: there are two classes (Sound and SoundBuffer) which are required to play a sound. Keep a pair of variables (Sound and SoundBuffer) as members of the class that plays the sound.

  The *quality* of the audio is not important nor is evaluated. You can simply pick any sound from an available free asset site, like `https://freesound.org/browse/tags/game-sound/`. **Attention:** Do **NOT** use any audio you don't have permission to use, or you'll incur in severe penalties for plagiarism.
- **Object Pool (35%):** Implement *three* separate object pools for potions, logs and fire objects (respectively). All objects in these pools must be created on start-up. All potions and logs should be *in-use*, while fire objects should be all *not-in-use* at start-up. No potions, logs or fire objects can be created or destroyed during the game, after start-up. They must instead toggle between *in-use* and *not-in-use* in their respective object pools.

- **Misc Features:**
  - **Tuning the Command pattern (15%):** Implement a swap functionality for the player's input. There should be two input modes for movement: ASDW keys and cursor keys. Define a separate key (for example, *Enter*) that, when pressed, switches from one input mode to the other. Note that both set of input controls must not be enabled at the same time (i.e. at any give time, you should either be able to move right pressing the key 'D', *or* the right cursor key, but not both).
  - **Collisions (15%):** Add a collisions manager that handles collisions using function pointers (concretely, the std::function<> type of callbacks). See Slides 13-14 of Week 5 (lecture *Entity Updates and Dynamic Memory Allocation*). This can be implemented in different ways, but here's an example:
    * Create a map in Game to store all the collision callbacks for the player (one for potions, one for logs).
    * Create two functions in Player to deal with both types of collisions (potions and logs). These functions must have the same *type* (return and parameters) as those of the callback.
    * Instead of dealing with the collisions in Game::update(), call the respective callbacks.
    * Callbacks need to be registered to the player object using std::bind. Note that this should be done once the player object has been initialized, not before.