ECS7014P - Lab Exercises

The Game Loop

Week 3 - Assignment 1-A (10%)

Lab Objectives

The objective of this lab is to continue with the work from the previous lab to implement different variants of the Game Loop pattern. The application resultant from this exercise will consist of a window where many "mushroom" objects will move independently and bounce off the limits of the screen.

In particular, in this lab you will:

- Implement different types of game loops.
- Get practical experience with C++ features, such as standard library vectors, raw pointers and passing objects to functions.
- Learn SFML utilities for drawing text on screen and keeping track of time.
- Calculate the Frames Per Second (FPS) at which an application runs and some simple ways of keeping it constant.
- Understand the difference between updating per frame and per second.

The code you produce by completing this lab corresponds to **Assignment 1-A**. Please read the assignment instructions 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 **17**th **February**. 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 **10**th **March**. The draft submission will **never** be evaluated as a final submission.

Before you start

Make sure you have completed the exercises from the previous lab. The instructions in this sheet assume you continue to work on the same codebase.

1 Factoring out an Entity class:

Our "mushroom" code logic is at the moment integrated with the game class, but they are meant to be independent entities. Especially, considering we plan to add many of them to this application. Extract the logic about the mushroom to a separate class *Entity*.

- First, add all member variables needed to draw and move the entity to the new Entity class.
- Add a constructor (which receives the necessary parameters to initialize the member variables of Entity) and a destructor to the Entity class.
- Declare and define the methods *move* and *draw*, so the entity can be moved and drawn on the screen, respectively.

Then, our class *Game* must now hold one member variable of type Entity. Do the following in the class *Game*:

- Create a **raw pointer** of type Entity in the class Game[1].
- Of course, we no longer need the texture, sprite and movement vector of the mushroom in the Game class; remove them now.
- Crete an entity for the mushroom in the constructor, and destroy it in the destructor.
- The movement and drawing logic for the mushroom are now in Entity functions. Re-direct these method calls to the respective Entity functions.
- You may have more functions in Game that were directly retrieving variables from the mushroom. Like in *move* and *draw*, create the appropriate methods in the Entity and re-direct the calls to keep the needed functionality.

After following all these steps, you should be able to build and run your code and obtained the same behaviour that you got at the end of the previous lab.

2 Randomizing the movement of the Entity

Now that the logic of an entity is encapsulated in its own class, we can make its behaviour more interesting without making the Game class more complex. Let's do the following:

- Randomize the starting position of the Entity (by default, this was (0,0)). Random numbers on C++ can be used from the standard library **srand**[2]. Follow these steps:
  1. Initialize the random seed at the start of the *main* function. Use as seed the value returned by the function time, as indicated in the CPP documentation. Note that you'll need to make a casting to solve a warning derived from simply using `srand(time(NULL));`.
  2. Set the initial position of the entity in the Entity constructor at random[3], always within the boundaries of the window. You will need to pass the Window object to the constructor of Entity to access its dimensions. To avoid requiring the constructor to copy the window object (as it would do if passed by value), pass **a pointer** to the window as a parameter instead.
  3. Build and play-test that the mushroom now appears at different positions at the start of the execution.
  4. Given the speed at which the mushroom moves, you're likely not able to notice anything. Try any (or both) of these methods to verify your code works:
     - Use the standard output to print to the console the starting position of the entity.

[1] In modern C++ , you would create a smart pointer (unique or shared, depending on the needs). In this lab, we will use raw pointers, but we'll work with smart pointers in future labs.

[2] `https://www.cplusplus.com/reference/cstdlib/srand/`

[3] Hint: use the function *rand()* (`https://www.cplusplus.com/reference/cstdlib/rand/`) and the macro RAND_MAX (`https://www.cplusplus.com/reference/cstdlib/RAND_MAX/`)

  - Use breakpoints and the debugger to inspect that the initial position of the entity is different at every execution.

- Not only we are able to initialize the position at random, also the orientation in which it moves. If you haven't changed anything, your mushroom is likely moving still with a velocity of $x = 4.0$ and $y = 4.0$ pixels per frame. Let's now give it an initial orientation with a random unit vector[4].

- The good thing about defining a unit vector is that we separate direction (the unit vector) and speed (the vector's magnitude). Let's then create a new member variable for Entity that captures the speed of the entity. Before, this speed was the magnitude of the vector $(4, 4)$.[5] Now, initialize the speed in the constructor of Entity with a value passed by parameter. As usual, remember to use initializer lists in constructors when possible.

- Define this speed value to be also passed to the Game's constructor as a parameter (we'll have to modify it later on). Play-test your code, passing different values of this speed to see the different effects.

3 Moving to multiple entities

Rather than having one single entity bouncing around, we are now going to change our code so we can hold many of them. In order to do this, we are going to use a data structure from the standard library: the std::vector:[6]

- Add a std::vector member to the Game class that holds (raw) pointers to Entity objects. There must no longer be a separate pointer for an entity object in this class.

- We will be adding multiple entities to the game. To do this, add a new parameter to the Game's constructor that indicates how many entities should be added. Then, in this constructor, create as many entity pointers as this parameter indicates and add them to the entity vector.[7]

- The destructor must now release the memory held by the pointers contained in the vector. Change the destructor so it iterates through all elements in the array, deleting the pointers and clearing the vector at the end.

- The function in Game that called the entity's function for moving needs to change, as we no longer have an entity pointer to call this function on. Instead, we need to iterate over all entity pointers in our vector to make entities move. Also, there is little need now to keep having a separate function for moving all game entities in Game. The loop that iterates through all entities can be directly placed in the Game function that updates the game state.

- You've probably realized that our move function in Entity is essentially the update function in the Update Pattern. Let's call things by their right name, and change Entity::move(...) to Entity::update(...).

- The other essential function of the Game Loop pattern is the one that renders all elements of the game. In this case, we also need to change our Game render function to call the function *draw* on every entity of the vector.

- Finally, we still have a function that returns the position of the mushroom. This doesn't make sense given we have now n entities. Change this method so it receives an integer, which will correspond to an index in the vector. Then, return the position (x,y) of the entity in the position indicated by the index in the vector. Make sure to check for invalid values for this index in this function, and raise an exception in case the index is not valid.[8]

Once these changes are made, you should be able to build and launch your application and see as many entities as you indicate to Game bouncing around the window.

[4] Given a random number n between 0 and 1, a random unit vector will be formed by $x = cos(n \times 2 \times \pi)$ and $y = sin(n \times 2 \times \pi)$

[5] **Q?** Freshen up your math: what value is that?

[6] `https://en.cppreference.com/w/cpp/container/vector`

[7] Note for the future: with this implementation, your code will be loading and storing as many separate copies of the same texture (the mushroom PNG) as entities we create. This is highly inefficient, and we'll get back to this later in the module, when we see the Flyweight pattern.

[8] Use the construct *throw std::runtime_error(std::string)*, see `https://en.cppreference.com/w/cpp/language/throw`. You'll need to *#include < exception >*

4 Pausing the game

It may be rather hard to visualize how many entities are *really* bouncing around, unless you are using a very low speed. Let's add a pause functionality to stop *time* and count, to make sure. Do the following:

- Game should know if it's paused or not, so it requires a member variable to indicate this. Add it to this class.
- Give this variable a default value (originally, the game is *not* paused) in the constructor. Our last reminder about this: use the intializers list.
- Let's handle the input in the function in Game that handles input (which should be empty at the moment). Capture the *Space* key event in this function and toggle the value of the pause variable when this event happens.
- When the game is paused, what do you think should happen in the *update* and *render* functions of Game? Implement this.

If everything was implemented correctly, you'll see that you're now able to pause the game when pressing the *Space* key and, when pressing the same key again, the entities will continue moving from the position they stopped.[9]

5 Determining and printing the game's framerate

The next task is to measure the framerate (Frames per second; FPS) of our application. In order to do this, we are going to first provide our game with a global timer.

- Create a variable of type **sf::Clock**[10] in the class Game. This object allows us to measure elapsed time. The clock starts automatically after being constructed.
- In order to query the time of the clock, we need to call the function *sf::Clock::getElapsedTime()*. This will return the elapsed time in the most precise time that the underlying OS can achieve (generally microseconds or nanoseconds) since the object was constructed or the function sf::Clock::restart() was called. Add a function to Game that returns this elapsed time (note the return type is sf::Time).

Let's first obtain the FPS of our simple game loop:

- As we are going to be implementing different types of game loops, refactor the three calls (handleInput, update and render) into a single function called *simpleLoop* in main.cpp. Call the *simpleLoop* method during every iteration of the game loop from the `main` function.
- In the *simpleLoop* function, calculate the FPS. You'll need to get two timestamps as seconds: the time before the call to *handleInput()* and the time after the call to *render()*. The elapsed time t is the difference between the two timestamps, and the FPS will be $1/t$. FPS is normally indicated as an integer (rather than a floating point number), so cast the result of $1/t$ to 'int' after calculating the FPS.
- Print these two amounts (elapsed time and FPS) to console, so it displays them at every iteration of the game loop.
- Add a return of type `int` to this function, and return the FPS value. Use this value to calculate the average FPS resultant for all frames run in the *main* function[11] and print this to console once (after the game loop has finished).
- Finally, try running the game with different amount of entities (i.e 1, 10, 100, 1000, 10000, etc...) and observe how the FPS goes down as you increase the number of entities on screen. Observe also the average FPS printed for each case.

[9] You may also notice that it's way too easy for your code to capture the event in more than one consecutive frame, making it pretty unreliable. A possible and elegant way of solving this is using the combination of two other patterns we see in this module: the Command and Event Queue Patterns.

[10] `https://www.sfml-dev.org/documentation/2.5.1/classsf_1_1Clock.php`

[11] Use the running/moving average to keep this updated at every game loop iteration: $avg_{t+1} = avg_t + \frac{FPS_{t+1} - avg_t}{n}$, where n is the number of frames executed.

6 Printing the FPS on the window

It's useful to see the FPS in the console, as you can see how the values fluctuate over time. However, printing things to the game window is often more convenient. Let's add the FPS to the window, so it doesn't only appear in the execution console.

- We will use sf::Text and sf::Font from SFML to write text on the screen. Check the documentation of these two classes at the SFML website.[12]
- Let's add the following members to the Window class to support this text drawing:
  1. A (private) sf::Font member variable to hold the font to use to draw text.
  2. A (private) sf::Text member variable to hold the text drawn on the screen.
  3. A (private, constant) integer that determines the size of the font to use (a good value is, for instance, 50).
- If you try to build the application now, it's possible that you get linker errors. The reason for this is that we need to add another library (*freetype.lib*) to our linker. Do so now for both *Debug* and *Release* configurations.

First, we need to setup the font for our text:

- Download the provided materials for this lab from QM+ and unzip it in your solution directory. This contains three files: regular and bold fonts, and the font license file. You could use any other font if you wish, this one is just provided for convenience.
- We need a function in Window that receives a string with the filename of the font (make it a const reference string object) and loads the font into the sf::Font object. Declare and define such function, which you can name *loadFont*, so that:
  - The function throws a runtime error if the file is not found.
  - The font, once loaded, is set as font to the sf::Text member variable.
  - The sf::Text variable must set the character size (using the constant you've recently defined in this class) and a color for the text (choose any sf::Color you like).
  - You can set a default text for sf::Text at this point, using the sf::Text::setString() method. This will be rewritten at every frame, but it's useful for testing that we are drawing correctly.
- This function that loads the font must be called from somewhere. We have a good candidate on the Window setup function. Call *loadFont* from the setup function passing the filename (including path) of one of the .ttf files provided.
- Finally, you need to actually ask the window to draw that text on the screen. sf::Text is a sf::Drawable object, so it can be passed to the sf::RenderWindow::draw() object as we do with sprites. Do this in the Window render method and verify that the default text is being drawn on the screen.

The final step is to draw the actual FPS on the screen, rather than a default text. For this:

- Implement a function (i.e. *getGUIText()*) in *Window* that returns a non-constant reference to the *sf::Text* object.
- We are calculating the FPS in our main file, where we (rightfully) don't have access to the window of the game. Add a function to Game that receives an int (the value of the FPS) as a parameter and returns `void`. Define this function so it create a string with this value (i.e. "FPS: <value>"), and sets it as the text the window should print, calling the function *getGUIText()* you've just added to Window.
- Finally, call this function from the *simpleLoop* method in main.cpp so the FPS value printed to the window is updated every frame.

[12] Also this post should come in handy: `https://www.sfml-dev.org/tutorials/2.5/graphics-text.php`.

7 A Game Loop with an FPS target

The next step in this lab is to implement a game loop that aims at a particular FPS target.

- Implement another function (e.g. *targetTimedLoop()*) in main.cpp that receives a target elapsed time as one of its parameters. This target elapsed time should be of type float, so if for instance we want to set up a target of 60 FPS, the value to pass would be $1/60 = 0.016$.
- This function should run an iteration of the game loop as normal but, if the elapsed time spent in this iteration is lower than the target elapsed time, the code would sleep until completing the target time. You can use the function *sf::sleep()*[13] from SFML.
- As you did in the *simpleLoop* function, print to console FPS and elapsed time, and add the time spent in the sleep phase of the iteration. The game window should still print the FPS of the game as in the previous case and also return the FPS of each iteration so the average FPS is still calculated at the *main* function.
- Try with different loads (different number of entities created and target times) to see what happens. Aim for testing FPS values of 15, 30, 60 and 120; your code should be able to keep these frame-rates using at least up to 100 simultaneous entities. Observe the average FPS and see how close they are to the target.

As you can see, this type of game loop makes sure the game doesn't run too fast, but it doesn't do anything if the game runs too slow. Because our game is too simple, it's hard to appreciate this, but we can simulate it by adding an artificial delay to the entities in our game:

- In the Entity's update function, add the following code that would simulate CPU cycles spent doing some heavy and non-deterministic computation:

```
1  //Artificial delay (80% time is low delay, 20% is high delay):
2  float delay = (((float)rand() / RAND_MAX) > 0.8) ? 0.075f : 0.01f;
3  sf::sleep(sf::seconds(delay));
```

- Play-test this version of the code with 5 entities and a target FPS of 60. Observe the results and how the movement of the entities is not continuous.

Make sure to comment these two lines of code to continue with the exercise.

8 An adaptive Game Loop

In the previous game loops, the game receives updates without being aware of the elapsed time between two consecutive frames. This has a big disadvantage: even if the speed of our entities is constant, their actual movement will be variable if the calls to the update methods is not periodic. At the moment, our entities move in *pixels per frame* units. Ideally, we would like to move our entities in *pixels per second*, independently of the potentially variable rate at which frames are processed. We'll address this next:

- Implement a new function, *adaptiveLoop()*, in main.cpp.
- This function must receive a float variable that indicates the elapsed time as seconds since the start of the game as captured at the previous iteration of the game loop. The goal is to use this value to calculate the elapsed time between frames.
- We need another update function in Game that receives a float parameter that indicates the elapsed time since the execution of the previous iteration of the game loop.
- The movement of the entities must now depend on the value of this elapsed time. Therefore, it needs to be passed to the Entity update's function.[14] Update the way the position of the entity is updated on each Entity update function, so it changes the position of the sprite in increments per second, rather than per frame.

[13] `https://www.sfml-dev.org/documentation/2.5.1/group__system.php#gab8c0d1f966b4e5110fd370b662d8c11b`

[14] Note that, in order to keep the Game's update method compatible with the previous game loop implementations, this parameter should be now added as an optional argument to the update function. See an example here for setting optional arguments and default values: `https://www.w3schools.com/cpp/cpp_function_default.asp`.

- Implement the adaptive loop in the function *adaptiveLoop()*, so the game receives the elapsed time since the execution of the previous iteration of the game loop in the update function. As in the previous functions, return the last FPS value so the *main* function can calculate the average FPS.
- Note that the unit of the speed has changed now. If you were moving the entities at, say, 4 pixels per frame, this same value 4 now means 4 pixels per second. At 60 FPS, this is **60 times slower**. So, you'll likely need to change the value of this speed to adjust it to the new units and keep the same speed you had previously.
- Finally, add the possibility of setting up a target FPS, as we did in Section 7, to the adaptive loop case.
- Play-test again with different settings for the target FPS and number of entities in the game. You can also bring back the artificial delay code to study how longer computation update times affect performance.

Congratulations! You've reached the end of the lab.

9 Extra

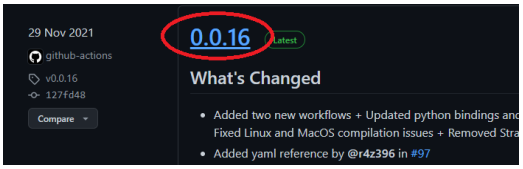
If you want an extra challenge (to be rewarded with extra marks in the assignment), you can attempt the following ideas:

- Add a time-to-live variable to the Entity class, so each entity disappears after X frames or seconds.
- Add another input to the game so new entities can be added dynamically when another key is pressed.
- The "Pause" functionality is a bit unstable, as it can capture and process the event in several consecutive frames. Fix this issue without using any other pattern seen in the lectures. Yes, we are asking you to find a *hack*.

Preparing your assignment 1-A submission

For the assignment submission, you are asked to fill and upload a 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.

The code is "submitted" as a github repository link. Rather than asking you for the link to the repository, we require that you create a GitHub *release* (check the GitHub documentation[15] 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. You can obtain the link to the release, which should be included in the form, from the name of the release in github:



The date on the release **must be** the same as the one in QM+. We will mark the state of the code release by the time of your submission to QM+. Note the release must be published (don't leave it as Draft!) so we can see it.

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.research.its.qmul.ac.uk/`[16], 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 fill a table with the average FPS obtained with different settings, and write a short answer (max 300 words) for a question about this exercise.

[15] `https://docs.github.com/en/repositories/releasing-projects-on-github/managing-releases-in-a-repository`
[16] If you are not, you should. Any problems, let us know **as soon as possible!**