ECS7014P - Lab Exercises
An Entity-Component-System Architecture

Lab Weeks 9-10 - Assignment 1-D (10%)

Lab Objectives

The objective of this lab is to implement an Entity-Component-System architecture on the mini-game we've implemented in the previous weeks.

In particular, in this lab you will:

- Modify the current code to incorporate systems and their interaction with entities and components.
- Separate the data and logic, so the former is encapsulated in components while the latter is managed by systems.
- Provide components with IDs and entities with bitmask, so systems can operate with them.
- Implement a big-array ECS architecture.

The code you produce by completing this lab corresponds to **Assignment 1-D**. Please read the instruction 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 **31st 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 **21th April**. The draft submission will **never** be evaluated as a final submission.

The code produced following these instructions will be the base for **Assignment 2**. In fact, the Big Array implementation already counts for 10% in Assignment 2, so you'll be directly contributing to the final assignment by completing this lab.

Before you start

In this case, you don't need to create any new Visual Studio 2022 solution. Keep using the same one you used in the previous lab, committing your changes to your repository. For this lab, you will will need to download a single file provided in QM+: **Bitmask.h**.

You can only start this lab once you've completed **at least** the steps from previous one referring to the TTL, Health, Input, Position and Velocity components.

An overview: what are we doing?

The objective of this lab is turn the EC architecture that you implemented in the previous lab into a full ECS. First, we'll modify our components to provide them with an ID. Then, we'll give our entities bitmasks so they can summarize what components they have.

We'll then implement *systems*, which will operate with a big-array architecture. We will start with probably the simplest system: the Time-to-Live (or TTL). Instructions for implementing this system are given in detail. After this, you're asked to implement the rest of the systems required for the mini-game. Finally, there is an option for you to implement further functionality for extra marks.

1 Adding Component identifiers

The first step for implementing our Big Array-based Entity Component System (ECS) is to add the bit mask functionality[1]. For this, we need a Bit Mask and provide our components with identifiers.

We'll start with the component identifiers. We will need to create a new `enum class` to define the IDs that we'll have for each type of component. Then, we'll have to add a member variable to all components implemented. In order to avoid having to repeat code through all our component classes, we'll use simple inheritance to add this feature. Do the following:

- Create a new file, Components.h, in include/components/.
- Define a new enum class with name ComponentID, with one value for each component that you have implemented. Assign consecutive integers to the different values of this type. For instance:

```
enum class ComponentID
{
 UNDEFINED = -1, //Good to have a default value.
 INPUT = 0,
 POSITION = 1,
 VELOCITY = 2,
 // ... Complete with all components you've implemented. If your components
 // use inheritance, pick the base class as the type (i.e. Logic, Graphics).
}
```

- Then, in the same file, define an interface[2] called *Component*. This class must have only one (public) member function called *getID()*, with a return type *ComponentID*.
- Each one of the components classes that you have implemented[3] must now:
  - Derive from the new class *Component*.
  - Implement the function getID(), which must return the value of the corresponding component type. For instance, *InputComponent::getID()* should only have the following statement:

    ```
    return ComponentID::INPUT;
    ```

This is all we need to be able to identify components by using the *getID()* function. We are going to use these IDs to reference components in our Systems and Entities. We'll start with the entities, in the next section.

2 A Bitmask for entities

Now, we'll add a bit mask to our entities that will identify which components they own. Take the provided BitMask.h file and add it to the project (a good place for this is in *include/utils/*). This file contains an implementation of the class *Bitmask*, a similar implementation to the one included in the book *SFML by Example (Raimondas Pupius)*. Bitmask offers functionality for working with bit-sets, so you can retrieve and modify individual bits from a 32-bit type. These bit sets will be used by the entities to indicate which components they own. The size of the bit mask (32 bits) imposes a limit on the number of components our entities may have (32)[4].

2.1 Bit masks

Let's give Entity a bitmask to indicate which components these entities have:

- The first thing to do is to provide Entity with a member variable of type Bitmask (name it e.g. *componentSet*).

[1]Checking the lecture slides about Big Array ECS along this lab is a good idea.
[2]Remember: an interface is a class where all its functions are pure virtual.
[3]Depending how far you got on the previous lab, you'll have a different components: TTL, Health, Input, Position, etc.
[4]Don't worry, we won't reach that limit. Also, 32 should also be a good limit for any real game.

- Add a getter function to Entity that returns this component set (call it *getComponentSet()*).
- Every time a new component is initialized, we need to turn on the bit that corresponds to its ID in the mask of the entity that owns the component. This must be done for every component in the Entity class and derived subclasses (Player, Fire, etc). As this needs to be repeated in several places, it's better to encapsulate this piece of functionality in a separate method. Do the following:
  1. Create a new function in Entity, *addComponent()*, which returns void and receives a shared pointer to a Component type.
  2. In this function:
     – Retrieve the id of the component by calling its function *getID()*.
     – Modify the bitmask owned by the entity by calling the function *Bitmask::turnOnBit()*, which must receive the ID of the component to add. Note you need to do a cast from *ComponentID* to *unsigned int* when making this call.
  3. Having this encapsulation means a small price to pay: all smart pointers to components in our code must be **shared**, not unique. We can't call *Entity::addComponent()* with a unique pointer as an argument. Do these changes now in all Component members that you have in the Entity class, and subclasses (Fire and Player). Note this will also require you to change their initialization (from *make_unique* to *make_shared*) where appropriate.
  4. Once all our smart pointers to components are of the type *shared*, we can call *Entity::addComponent()*. Do so every time a component is initialized in any of these entity classes, passing the component you've just initialized. For instance, in Fire's constructor you would do something like this to add the TTL and Velocity components:

     ```
     ttl = std::make_shared<TTLComponent>(startTimeToLive);
     addComponent(ttl);

     velocity = std::make_shared<VelocityComponent>();
     addComponent(velocity);
     ```

     Call *addComponent()* in the Entity and Player classes[5].

2.2 Comparing masks

Finally, we need a function that indicates if the entity has a set of components indicated by a bitmask.

- We'll need a function that indicates if the entity has a set of components summarized by a bitmask (passed by parameter). Declare a function with the following signature in Entity that returns a bool type:

  ```
  bool hasComponent(Bitmask mask) const;
  ```

  Then, give a definition to this function so it calls the function *Bitmask::contains()* called upon the member variable *Entity::componentSet*, passing the mask received by parameter. *Entity::hasComponent()* must return the same value returned by *Bitmask::contains()*.

  The objective of this function is to provide a way to the Systems to evaluate if the entity owns the components the system needs to work with. Systems will have their own masks, and by calling *Entity::hasComponent* we can validate if a particular System (e.g. MovementSystem) can work with a particular entity (which should have, in our example, the Position and Velocity components). We'll build this functionality in the following steps.

Before closing this chapter, make sure your code builds and runs correctly. Next, we'll add Systems.

3 Adding Systems

Our next step is to add systems to our architecture. In this section, we'll add one system (for Time to Live, or TTL) and the infrastructure needed to support it.

[5]If you implemented the graphics component, you may be receiving it via parameter in the init() method. You'll have to call addComponent() from Entity::init() in that case.

3.1 The System interface

As shown in the lectures, our infrastructure for systems requires a generic interface and subclasses for each system. We'll start with the System interface:

- Create a new file *Systems.h* in the directory include/systems/. The declarations of System and subclasses will be pretty small, so we can use this single file for all declarations.
- Declare in this file a new class *System*, which must have the following:
  - Declare a protected member variable, *componentMask*, of type *Bitmask*. This bit mask will indicate which components the system will operate with.
  - Declare a public pure virtual function *update*, which returns void and receives three parameters: a pointer to a Game, a pointer to an Entity and a *float* type (for the game loop's elapsed time).
  - Declare **and define** a public member function *validate*, which returns a *bool* type and has one single parameter: a pointer to an Entity type. This function must call and return the value obtained by *Entity::hasComponent()*, called on the entity received by parameter and passing the *componentMask* member of the System class. The objective of this validate function is to verify if this system is compatible with a given entity.

3.2 The TTL System

It's easier to understand and visualize how are we going to use systems by implementing one first, before it's fully integrated into the architecture. Let's implement one of the simplest ones first, the TTL system:

- In Systems.h, declare the class *TTLSystem*, which inherits from *System*.
- Add to this class two declarations: a public (default) constructor and the overriden *update* function.
- Create a new file *TTLSystem.cpp* in source/systems/. In this file, create the definition of the following two functions:
  - The **update** function. Leave it empty for now.
  - The default constructor. The only thing to do in this constructor is to turn on the bit of the system's mask that corresponds to the components needed for the TTL System to operate. In this case, we only need for an entity to have the TTL component[6]. You can add this bit to the component's mask of the system with the following code:

```
componentMask.turnOnBit(static_cast<int>(ComponentID::TTL));
```

The *TTLSystem::update()* function will take care of the logic behind this system. In this case, of decreasing the time to live variable and deleting the entity when this gets to 0. To make this work, we need a few functions in our classes to give access and functionality to the involved members in this business.

The first of our concerns is that we don't have a way of retrieving *any* component from an Entity pointer. Our TTLSystem::update() function receives a pointer to *Entity*, hence we need a way to retrieving the TTL component from the Entity object. Note that Entity objects do **not** have a TTL Component[7]. We are going to use polymorphism to solve this issue[8].

Add the following:

- In Entity.h, declare a virtual function called *getTTLComponent()* that returns a shared pointer to a TTLComponent.

[6]A TTL system does not need the entity to have a position, or a velocity, to operate. It does not matter that an entity has other components. Our TTL system will be able to operate with an entity if it has *at least* the TTL component. That is what the function *validate()*, which you implemented in the previous section, will do when deciding if a system must be run.

[7]Why would they, not all entities require TTL functionality. And adding this component to Entity would be bad practice (remember the Bubble-up effect).

[8]Later, in section 5.1, we propose a better and cleaner way for doing this, for extra marks.

- In Entity.cpp, add a definition for this function for the class Entity. This function must return, simply, the null pointer (*nullptr*). This is how Entity tells to whoever calls this function in an Entity object that this object does **not** have a TTL component.
- In Fire.h, add and override the Entity's virtual function *getTTLComponent()* you've just created. This function simply returns the (for this class existing) *TTLComponnent::ttl* shared pointer. This is how Fire tells to whoever calls this function in a Fire object that this object **does have** a TTL component.

  Why does this work? TTLSystem::update() receives a pointer to Entity. Remember that the compiler will resolve the call to *the most derived* possible class. If the argument is an actual Fire object, we'll retrieve the TTLComponent from the Fire. If it's a basic Entity, we'll get a null pointer from this call.

Now we can complete the definition of TTLSystem::update():

- First, retrieve the shared pointer of the TTL Component by calling the function *Entity::getTTLComponent()*.
- For safety and to avoid following a null pointer, add a condition that verifies if the returned pointer is not null. If it is null, the function should throw an exception[9].
- Once we are sure the component retrieved is not a null pointer, we need to execute the logic of the TLL component update. As we are removing the logic from the component (to implement a proper ECS architecture), we need to bring that logic to the system class. Therefore, add the following to TTLSystem::update():
  1. Decrease the TTL: *ttl* in TTLComponent is a private *int*, so we need a function that decreases its value by one every time is called. Create this function in TTLComponent and call it from TTLSystem::update().
  2. Check if the value of the TTL counter (TTLComponent::getTTL()) is less or equal than 0 and, in that case, delete the entity (calling the function Entity::deleteEntity()).

Finally, the next step is to remove the logic from the TTL component. In fact, we won't even need to have an *update()* function: this component doesn't need to be "updated" once per frame, as they are just containers for data. Their only function and structure now will be to hold this data and provide functions to access and modify it. Do the following:

- Remove the function TTLComponent::update().
- This function was called from Fire::update(). Not only that, Fire::update() was also checking the value of TTL and flagging the entity for deletion. This is now done by the TTLSystem, so remove the TTL related functionality from Fire::update().

You should now be able to build and run the game. If you play-test, you'll realize that the Fire objects do not disappear with time. This makes sense, as we've moved the TTL functionality to the system, but we have not setup Game to use this (or any) system. That's what we'll do in the next subsection.

3.3 Handling the control of the systems to the game

Our first step is to actually bring systems to game. As we'll have quite a few, rather than having separate pointers to systems in Game, we'll organize them into a vector:

- Declare a STD vector containing shared pointers of type System, as a private member of the class Game. We'll refer to this vector as "systems" in these instructions, and you may call it like that as well.
- In the Game's constructor, initialize a shared pointer of type TTLSystem. Add this to the systems vector.

Now, we need a function that implements the big array ECS architecture. Do the following:

- In Game, declare a function bigArray, which returns type void and receives a single parameter: a float for the elapsed time.

[9]We still haven't seen how the function update() is going to be called, but preceding that call with a check on validate() will make sure that we never try to access a null pointer in this function. If an exception is thrown, we know that we're not properly doing our job on validating which entities can be used by which systems.

- Add the definition of Game::bitArray() in Game.cpp. This function must iterate through all systems and entities, and call the respective System::update() functions **only** in those entities that match the component mask of the system (which we can verify with the System::validate() function we've already implemented). Let's do it as follows:

  1. Retrieve an iterator from the systems vector and use it (with a *while* loop) to iterate through all systems defined in this vector.
  2. For every system (i.e. for every iteration of the previous loop), retrieve *another* iterator for all the entities of the game. Again, iterate through all existing entities in the entities vector.
  3. For every system-entity pair (i.e. for every iteration of the inner loop), we need to check two conditions:
     - That the entity is not deleted.
     - That the system is valid for the entity, calling *System::validate()* and passing the entity as the only argument of this function.
  4. If both conditions above are `true`, call the function Systems::update(), passing the three arguments this method receives: a pointer to this game, a pointer to the entity we are considering in this iteration, and the elapsed time variable passed by parameter to our Game::bigArray() function.

- Finally, we need to call our bigArray ECS implementation function. We will do this in *Game::udpate()*, so go to this function and find the loop that iterates and updates all entities in the game. If this was the end of our work, we would *substitute* this loop for our call to *Game::bigArray()*. However, if we do this, we would essentially stop all logic from working, except the TTL functionality - the only system we've implemented so far. Therefore, simply add a call to *Game::bigArray()*, passing the elapsed timer as an argument, before the entity update loop. We'll delete this loop once we're done with all systems.

Now build your game again and play-test. In this situation, you should now see the effects again of TTL, making fire disappear a few seconds after our character spawns it.

4 Adding all the other systems

The bulk of this assignment is to implement the rest of the systems for the game, following the same procedure as done for the TTLSystem. For each system (described below), do the following:

1. Declare a new "XSystem" as a derived class of *System*, which must have an *update* function (which overrides the parent's class update) and a default constructor.
2. The Entity needs to add a function that returns the component (as for TTLComponent, using polymorphism to return the component in the most derived class, returning *nullptr* in the parent ones that do not have that component).
3. The constructors of the new system classes need to turn on the necessary bits of the component mask that refer to the components the systems operate with.
4. The update functions of the new system classes must now execute the logic of their respective components. This requires removing the code from the component classes and likely adding new functions to access or modify their data. These functions will be used by the implemented systems.
5. Systems must be added to the vector of systems in Game, initialized and inserted in the vector in the Game's constructor.

Roughly, the same procedure described above should be used for all systems of the game. A few notes on each one of them:[10]:

- **InputSystem:** requires the Input and Velocity components. The system must reset the velocity, retrieve the player input and execute the commands (from the Command pattern). Note that Game will no longer need to call the player's handleInput function: it's the system what will be executed to retrieve and apply the input, not *Game::handleInput()* anymore. In fact, *Player::handleInput()* can be removed. The game's input will still be managed from *Game::handleInput()*.

[10]As you can see, the list of systems does not include a *HealthSystem*. Not all components need to have their own system, and a Health system is actually not necessary (we don't need to update anything in Health every frame).

- **MovementSystem:** requires the Position and Velocity components. The system must set the position of the entity using the current velocity and speed of the entity. Note this system will be able to move *any* entity that has a position and a velocity component defined (i.e. Player **and** Fire objects).
- **GraphicsSystem:** requires the Graphics and the Position components. The latter is used to retrieve the position of the entity to be drawn, while the former is used to retrieve the sprite to be drawn.
- **ColliderSystem:** requires the Collider and the Position components. The latter is used to retrieve the position of the entity, so the bounding box of the collider can be updated to the new position. This system also draws the collider to the screen.
- **GameplaySystem:** requires one single component: Logic Component. The update function of this system simply redirects the call to the update function of the logic component[11].

As seen in the lectures, it's important to note the existing system inter-dependencies. The order in which systems need to be executed matters. For instance, the InputSystem must run before the MovementSystem, so the player reacts to the input in the same frame it's provided. You can control this by providing the right order of insertion to the game's vector of systems in the constructor of Game.

Finally, remember to remove the entities update loop, in Game, once **all** systems have been implemented. However, you'll need to leave this update loop *as is* in case you don't implement all systems, as otherwise you'd lose functionality.

5 Extra

We propose two different modifications that can be attempted for extra marks. All these ideas derive in a more correct, cleaner, more general and simpler code for our ECS.

5.1 Organizing components in Entity

The use of multiple getComponentX() functions in Entity is not as neat as it could be. True, we are avoiding bubbling up components to Entity, but we still have to add a bunch of functions in many classes every time we want to add a new component. There is a better design for this, which consists of using a map of components:

- In the class Entity, add a map (from the standard library, std::map) where the key is the ComponentID and the value a shared pointer to a *Component*.
- The class Entity must also implement the following functions for adding and returning components to and from this map:

```cpp
Component* getComponent(ComponentID id);
void addComponent(std::shared_ptr <Component> c);
```

- Modify Entity::addComponent() so:
  - It gets the ID of the component received by parameter and add the component to the Entity's map of components, using the retrieved ID as a key.
  - Turns *on* the bit that corresponds to the Component ID in the Entity's component mask (this should be there already).
- Rather than using individual components in Entity and derived classes, we'll use this new map we've created. We need to update the following:
  - Remove the member shared pointers to individual components in Entity and derived classes, as well as the getComponentX() functions.
  - Substitute the initialization of these removed individual components for a call to the new *Entity::addComponent()* method. This way, we are intializaing all components in the same way, adding them to the Entity's map, and setting the component bit mask in one single call.

[11]Player should have, at this point, a shared pointer to a PlayerState component, which subclasses LogicComponent. When GameplaySystem retrieves the logic component from any entity, it will retrieve the PlayerState from the Player. This will automatically call the gameplay logic from the player. If other logic was to be added to different entities, this would work without modifying the gameplay system at all.

5.2 Separating Graphics and Logic Systems

There is one aspect that the previous instructions don't get completely right: one system is dedicated to graphics, but its operations take place during the update phase of the game loop. Drawing must happen in the rendering phase of the game loop. One way to fix this is as follows:

- Rather than having one vector for systems in Game, have **two**: one for logic systems and another one for graphical systems (in our case, so far, only the GraphicsSystem). Rename the current *systems* vector to *logicSystems* and create a new one (*graphicsSystems*) for graphics systems.
- Add the GraphicsSystem to the *graphicsSystems* vector in the Game's constructor (instead of *logicSystems*).
- This will make our bigArray method not to execute the graphics system. We could replicate the bigArray method so it uses the graphics vector, but code duplication is not great. Instead:
  - Add a parameter to the bigArray function, which is a vector of shared pointers to System.
  - In *Game::update()*, call bigArray passing the *logicSystems* vector to it.
  - In *Game::render()*, after drawing the board, call the bigArray function passing the *graphicsSystems* vector.

The second part of our code that is not correctly placed is the drawing of the collision boundaries of the game entities. Additionally, this is debug information: if we wanted to eliminate this drawing, we had to go to the Collider system and comment or delete the drawing functionality. We can fix these two issues at once:

- Create a **new system**: PrintDebugSystem. This system requires the collider component. In the update function, the system must retrieve the bounding box from the collider component and draw it.
- Add this new system to the *graphicsSystems* vector in Game that you've just added above.

Now, this debug information is rendered at the correct time of the game loop. Additionally, it's much easier now to stop debugging this information. For instance, you can add a *bool* parameter to the Game constructor that indicates if debug printing should be enabled, and only add the new PrintDebugSystem to the *graphicsSystems* vector if this parameter is `true`.

Preparing your assignment 1-D 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. 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. We should already have access to your repository from the previous submission, but double check that the accounts **eex516** and **acw634** do indeed have access to it.