Architecture

Description

The initial class diagram, Fig 1.1, shows how the concrete backbone of our entity-component system architecture is structured. Fig 1.2, 1.3 and 1.4 are on separate diagrams as it was difficult to fit them into Fig 1.1 whilst keeping the diagram legible and clear; however, they are still connected to the main architecture, as can be seen through the 'Component', '*IEvent*' and 'GameSystem' superclasses.

These diagrams are UML standard class diagrams, and were created with the jsUML2 editor.

The architecture consists of an entity-component system as detailed in the previous assessment's architecture section. The explanation of our architecture has not changed much from our more abstract architecture, apart from being more firmly grounded in the implementation – for example, individual component and system classes are shown.

However, the links between classes have been established by the diagrams in this document. There was almost no indication of relationships and associations between classes shown in our previous architecture document – instead, mere inheritance was shown. This did not give an accurate enough picture of our architecture, which, at its core (that is, the entity managers and actual game state update) attempts not to rely a great deal on inheritance.

Some (not all) attributes and methods of classes in **Fig 1.1** (appendix) have been shown. This is because they help to illustrate the relationships between the classes. For example, EntityEngine contains 1 instance of each EntityManager, EntityTagManager, and EventManager, so the diagram shows that the class has an attribute of type EntityManager. It is therefore clearer to imagine how the EntityEngine, which is a convenient collection of the EntityManager, EntityTagManager and EventManager, can associate to 1 instance of each of these whereas they in turn do not associate with an EntityEngine (as the body of these three classes does not refer to or know about the EntityEngine). Through these shown attributes, methods and associations, the method we have implemented of managing the game data is represented.

The EntityEngine, therefore, contains an instance of each EntityManager, EntityTagManager and EventManager, and therefore encompasses the entity handling. Then the classes IEvent, which handles events, and GameSystem, which is the baseline class upon which 'systems' (different logics, such as MovementSystem) inherit from. SystemManager is a unique, ordered list of GameSystem classes, used to apply the logic with priority whilst the game is running – therefore has an aggregate relationship with this class.

Additionally, the World class also contains the current instance of entityEngine.

Entity manager contains the hash map 'entities', which contains the unique identifiers for each entity present in the game. As explained in our previous architecture documents, an 'entity' is simply and integer identification number. Components are then mapped to this number to determine its behaviour; the entity in turn maps to its personal value of each component. This is shown in Fig 1.1 by the EntityManager containing one instance of

‘entities’: the entity hash map. Many components can relate to each entity in the hash map, and each entity can relate to multiple components – but there is only 1 entity hash map.

Fig 1.2, **1.3** and **1.4** (appendix) show the representation of classes which inherit from the Component, GameSystem and IEvent class respectively. These diagrams are still connected to the main architecture through the EntityEngine, GameSystem and SystemManager associations, but are on separate figures to allow a clearer, more visible representation. Classes such as WeaponComponent and CollisionComponent inherit from Component, the base component class. The same is true with classes such as MovementSystem and InputLogicSystem for the baseline GameSystem class, and BulletCollision for the IEvent interface. Creating baseline classes in this way allows us to organise classes based on whether they are of type Component, System or Event.

System classes then handle the logic of how components are applied to entities. This creates the operation of the game.

Justification

Our choice of using an entity-component system is based on the ease of understanding and simplicity of the model when compared with deep, far-reaching inheritance models. Entity-component systems are widely used in the industry for this reason; it follows the ‘composition over inheritance’ model. Classes define behaviours the system should exhibit, which are then applied to entities, rather than behaviour being created through a complex mass of super-classes and subclasses.

Apart from the human understanding advantage, the entity-component model also offers a great deal of efficient code reusability. Logic classes are created once; they then apply to all entities with the concerned components. To illustrate this, look at the example of the ‘player’ entity and ‘enemy’ entities.

In our entity-component architecture, the movement system and collision system are reused for the player entity and the enemy entities. Movement and Collision components are assigned to the entities and, with no extra code, both can behave with the same movement logic – the difference being, of course, that the player is input-movable, whereas the enemies are moved automatically.

The inheritance model would require a different, less code-efficient approach. In this example, it would not be logical for an Enemy class to inherit from a Player class – no doubt unwanted attributes and methods would be transferred. It would also be strange to have both classes inherit from the same abstract class or superclass.

To explain further, imagine in this example that the enemy was not movable, but simply a solid (can be collided with) guard turret. The player and the turret are clearly not the same ‘thing’, and it would be confusing to introduce inheritance into that relationship. However, the same collision logic could apply to both. There would then be a problem of how to apply collision to each entity without duplicating code. The entity-component system’s approach of adding logic to individual entities is therefore a better choice to use to create our game.

To summarise, the system means it is possible to create concrete classes while duplicating next to no code. In this example, we have the Player class, represented by an entity - it is

solid, movable and user controlled. By contrast we have the enemy entity, which is solid, movable but not user controlled. Another example would be a health pickup entity, which is neither solid, movable or user controlled, but has its own components and related logic systems which handle how it is picked up, and its effect on the player entity state (increase health).

The entity-component system is a good fit for our requirements. The player entity, aka the duck, is required by FR5.1 and FR5.2 to have a health bar and stamina bar. The health bar is already implemented through use of a health component, and the stamina bar can easily be implemented in a similar way.

The duck's health is required in FR5.1 to reduce when it is hit by an enemy or enemy projectile. This is achieved through giving each entity a 'team component' – anything which is not on the player's team can reduce its health, whereas this also ensures that entities which are on the player's team – e.g. its own projectiles – will not reduce the player's health. This allows implementation of a wide range of obstacles. Should the requirements change to require it, damaging 'lasers' which switch on/off in a pattern, or randomly generated spikes in the floor, etc., could easily and quickly be added using this existing logic in combination with out tile-based design.

FR5.3 lists the player's innate abilities. The architecture of our system is ideal for implementing these, as different speeds of movement can be quickly added by changing the integer value of the velocity component attached to the player entity. Similarly, acquired abilities (FR5.4) can be implemented by adding new components to the entity as they are acquired throughout the game. The projectile-throwing ability is created through a weapon component and the bullet component class, and the effects of the weapon (damage, collision) is done through the team component class, as previously touched on.

Enemies are implemented as entities. They may or may not have movement components and projectile components, and all of them have team components which are the opposite team to the player's. Most of the remaining logic is still the same.

Requirement FR10 describes the point system, which awards the player points when they kill enemies, perform 'combos' (killing more than one enemy in quick succession), and completing a quest/objective. This is achieved through a score component attached to the player, which attaches an integer 'score' to the entity and also handles the combo calculations.

The idea of an entity-component system and most of the basic classes have been lifted from our initial architecture model. Several more system and component classes have been added as it became clear they were needed in implementation from our agile method and requirements updates. However, the most major change in our representation is the inclusion of the links between classes. This is an aspect of our architecture that was largely missing, and under-represented, in our previous architecture document. Our current diagrammatic representations illustrate the relationships, rather than only the classes themselves and their attributes/methods, as outlined in the description. All associations are shown, whereas previously, only inheritance could be seen in the diagrams. Therefore our current representation is a much more accurate, concrete picture of our implemented architecture.