Implementation

Design Decisions

In order, to implement the entity manager as described in the concrete architecture, we chose to use hash tables to associate entities with their components. The data type that is used is: `HashMap<Class<? extends Component>, HashMap<Integer, ? extends Component>>` The outer hash map has keys that are classes (e.g. `PositionComponent.class`), and the inner hash map is from entity IDs as the keys to instances of that specific component as the value. However, we wrote and tested many helper functions that abstract this detail into an easy-to-use interface. for the collections of entities that these functions return, we use sets because they, by definition, remove duplicate entries and also provide us the ability to intersect the set of entities using e.g. PositionComponent and the set with CollisionComponent instances so that our collision systems will only work on the entities that have both.

We took a similar approach with our event manager; its key property is `HashMap<Integer, HashMap<String, IEvent>>`. The outer hash map's keys are the entities, and the inner maps from names such as "COLLISION" to some function that gets executed when the event is triggered. This way, we can group all the events belonging to an entity, and assign each event a name (defined as a static constant in EventName) so that retrieval of the event to fire is simple.

The IEvent interface just defines one function whose signature is `public boolean fire(EntityEngine entityEngine, int sender, int receiver, Object data)`. The receiver ID is the ID of the entity this event is currently running on (because multiple entities could share the same event), and the sender ID is who sent the message to the receiver. The data parameter is of type object, and could also be null. The point of this parameter is to send additional data to the event, such as collision handlers which take the intersection instance. Because different event handlers require different types of data, the parameter is of the general type Object, and then checked for validity using the instanceof operator.

The return value of the fire function is a boolean to indicate whether the receiver entity should be removed. Removal of entities is managed by the EntityEngine class which stores a set of entities that have been flagged for removal. At the end of the frame, DuckGame.update calls EntityEngine.removeAllFlaggedEntities which iterates through this set, and tells the entity manager, tag manager and event manager to remove the entity. The reason why we have to wait until the end of the frame is to prevent null pointer exceptions; for example, when a collision between 2 entities occurs, entity A sends a message to entity B, and then B sends a message to A. If at the end of the first message, B gets removed, but then the second message requires referencing B, it won't be able to because it no longer exists.

Because multiple entities can share the same events, e.g. all the enemy geese have a collision event, we decided to let them point to the same instance of EnemyCollision. To do this, we implemented the singleton pattern for our classes that implement IEvent.

Another key part of the game is the input system. This is implemented as a hash map from an action enum to a binding, stored in the ControlMap class. An “action” in the game is something that the player can do, such as go left, reload weapon or sprint, and a “binding” is the input required for the action. By using this method instead of hardcoding this, it allows us to have rebindable controls, and potentially could be extended to a local multiplayer system where each player has a different ControlMap.

The Binding class groups together a KeyBinding and ControllerBinding instance, with helper functions for setting the keys/controller inputs required for the binding. The KeyBinding consists of a primary key and secondary key value - this allows us to bind walking to both WASD and the arrow keys. The ControllerBinding needs to be able to support the multiple types of input that controllers can give: axes, buttons and the d-pad, so it uses an enum to tell which type of binding this is for, and an index for which specific axis/button/d-pad is providing the input. The helper functions in all of these classes are also designed to be able to handle the idea of “null” - ie. they never throw null pointer exceptions. Instead, if you try to pass it a null, it creates an instance of a default key/controller binding (either using LibGDX’s Keys.UNKNOWN or ControllerBindingType.NONE).

Retrieving input from the control map was also designed so that the caller doesn’t need to know if the input came from the keyboard or the controller. Instead of using just booleans to represent the state for an action (i.e. true if the key is pressed, else false), we had to switch an analogue system due to controller axes returning a continuous value between -1 and 1. Our solution was to use floats and represent a button pressed as either 0 or 1, then query the binding for whether it’s state was greater than 0. We also had to split axes up into a positive axis and a negative axis so that our controller querying functions returned in the range of 0 to 1. Deadzone support was also added to each binding so that axis bindings will only return a non-zero result if the position reported from the controller is greater than the deadzone.

A big part of the game is the enemy geese AI. This was initially designed as a state machine before being turned into code. There are 3 main states that the enemy could be in: patrolling, chasing or searching.

- Only enemies that have a patrol route attached to them will be patrolling - this is where they follow a predefined path. They have a viewcone attached to them and if the player enters this cone (detected using the length of the vector between the enemy and player, a dot product calculation to calculate the angle, and raycasting to check for line of sight), they switch to the chasing state.
- When in the chasing state, an enemy has a target who, as long as they can still see them, they’ll constantly walk towards them - at this stage, pathfinding has not yet been implemented so they just walk in the direction of the vector between the 2 entities. If they lose sight of their target, they’ll enter the search state.
- The searching state is the default state for all the enemies that don’t have a patrol route. If an enemy is searching, then they either pick a random direction to walk in and a time (approximately 2 to 6 seconds), or follow their last known vector to their target if they’ve come from the chase state. If they see the player whilst searching, they’ll swap back to the chase state, however if they don’t after about 5 seconds, and

they are able to forget the player (i.e. they have a patrol route to return to), then they'll forget about the player and return to their patrol route.

Still Required to Implement

One of our functional requirements, FR4, is to allow the user to save and load their game. It references UR1.3 and 1.3.1 that state there should be options on the menu to allow the player to do this. We have implemented the buttons that call the function save(), but the code to actually save and load the user's game has not been written. Adding this functionality should be straightforward because of the GameProgress class. This gets passed between game screens via the win screen and contains the current level number and cumulative score for the player - it can be edited to include more such as abilities as stated in FR 5.3 and 5.4.

We also intend for the game to have multiple types of obstacles (FR9). Currently it just has the standard goose, however with the use of our entity-component system, adding more entities to meet these requirements isn't difficult as much of their logic has already been created. For example, the guard turret would require a PositionComponent, SpriteComponent, CollisionComponent, WeaponComponent and just one new component that stores the data of this turret (such as who it is currently targeting, the quality of its aim etc.): TurretComponent. For the logic of the turret, it would need a new system, but due to the separation of concerns in ECS, this system would be easy to implement as it stands by itself without worrying about the rest of the game logic.

In FR3.5, we say that the game should also include sound. Currently there is no code for playing sounds in the game, however the framework we are using, LibGDX, provides support for playing sounds. Our event system would also make it easy to add sounds into the game - e.g. when a collision between the player and an enemy happens, the triggered event can play the damage sound.

As our game is a stealth game based around the light produced from a torch, shadows are a necessary feature to add. The light right now appears to travel through the walls which isn't their intended effect - instead they should stop at the wall. A possible implementation for this is to use an OpenGL triangle fan with the centre vertex at the position of the light emitter, and then the other vertices sit either along the edges of tiles that stop the light, or at the edge of the cone if there aren't any tiles. Rendering this shape, with varying alpha values at the vertices, onto the frame buffer would produce the effect of the walls actually stopping the light, however it is more difficult to implement than the current solution of just rendering a cone texture onto the frame buffer.