# DoodleJump

Advanced Programming Project 2021-2022
Thomas Gueutal
s0195095

The design pattern that exerted the most influence over my general project structure was the model-view-controller pattern. This is evident in the naming of the subfolders of my src directory, which are named after the pattern components. A clear separation of the three subcomponents is paramount to creating a code base that is both extensible and easily understandable.

To that end, the logic library should be entirely standalone. The logic library defines three valid entry points for outside code, that is the view and controller subcomponents or any client code that wishes to run the DoodleJump game with a visual component.
The first and most important entry point is the World class itself. Its interface allows a controller-like object to tie together the unstructured methods into the coherent execution of the DoodleJump game. Such methods include instantiating and resetting the world state, requesting views and other methods. This is essentially the role that my own controller implementation performs.
The second entry point is the abstract entity factory. It provides an interface to the world for entities to be created. As the factory methods defined within this interface only produce entity types accessible within the logic library, all subtypes of the Entity type, the world does not need to know of any view related logic. This logic is separately defined within the view component and utilized by the concrete implementation of the factory.
The third access point is the set of subtypes of the Entity type. These classes provide interfaces that allow the customisation of their internal attributes, which allows for different gameplay experiences based on the implementation of the abstract factory. Though admittedly, this interface is quite minimal; changing how much score a bonus is worth and not much more.

Of special note is that the logic library is actually a fully functional implementation of the DoodleJump game and can be run completely without a visual component. The assignment specified that an entity should be destroyed, clipped, by the camera once it went out of sight. The logic library provides two definitions for being ‘out of sight’. The first relies on the collision rectangle that all entities possess. This allows the camera to still clip objects even if no visual component is present, meaning that the world can run the simulation.

This brings us to the view component. This library relies on a single key feature of the logic library, the Event enum that allows the generalization of the events that the world requires in relation to the used graphics library. The EntityView, the view component attached to an Entity, relies on the CollisionObject logic library class. Other than that, it is standalone.
The view component defines two distinct sections of code. The first pertains to extending the logic library entities into entities that have a view attached. This attachment process is contained within the TemplateView class. As its name suggests, this class template forms a basis for defining the viewable entities and allowing the world to instantiate them. As the

entity hierarchy was derived into many distinct subclasses that all still share the same generalized logic for being displayed, the TemplateView class enforces uniform implementations for all these viewable derivatives of the logic entities. This means that the class facilitates the translation of Entity state into and manipulation of EntityView state. In practice, this means the following. The concrete factory may choose to create TemplateView entities instead. This allows access to view functionality such as attaching a texture or letting the view be a coloured rectangle instead. The controller may then request the world for the views. Because the creation process of the view entities involves attaching the controller as an observer to the view, the view entities register a snapshot of their view state in the controller to be drawn onto the window.

Of special note is that the TemplateView class provides one additional functionality: view clipping. As previously mentioned, a world running without a view component performs clipping on the collision shapes. However, a view also has a separate collision shape attached to it. This allows the separation of collision shape and view shape. The TemplateView class then enforces that any entity with a view attached will utilize their view area instead of their collision area in the clipping process. This clean separation allows for an entity to have a smaller collision shape than their view shape would suggest, which enriches the customizability of the concrete factories in defining interactions between views and collisions.

The second section of the view component pertains to the general implementation of a window manager. The WindowManager class template makes use of the ResourceManager class template to allow easy switching between graphics libraries by simply defining a concrete implementation of the WindowManager class and changing the definition of it in the controller.

Lastly, the controller binds together the view and model components. It makes use of the world interface to define the main game loop. After it requests the world to adjust its internal state based on user input polled from the view component, it requests the view to be delivered to it via a variation to the observer pattern. Namely, it requests the world to signal all of the entities to register their views via display calls. All viewable entities, which define concrete implementations of this observer behavior, will respond to this. The controller, Game, can then push these views to the window manager part of the view component to visualize these entity view states.

Another implementation of the observer pattern is for linking bonuses to the player. The player checks for collisions with its bonus observers and alerts them of any. The bonuses then verify the collision context and respond appropriately, perhaps by modifying the player's movement through the player interface.

Lastly, for the sake of extensionalibility, each entity may override a process method. This method, together with the world being a singleton accessible within these process methods or through Entity class wrapper methods, allows them to influence the game state. They may access the world's interface and rely on camera state, collision info and their own state to make decisions.