Tarah Peltz, Audrey Randall, Chance Roberts, Peter Gutenko

We apologize for the size of our diagrams and the complexity of our code, but there's no way they will fit into an 8.5x11 PDF. Please see our Original Class Diagram and New Class Diagram on Github. Excerpts are available below.

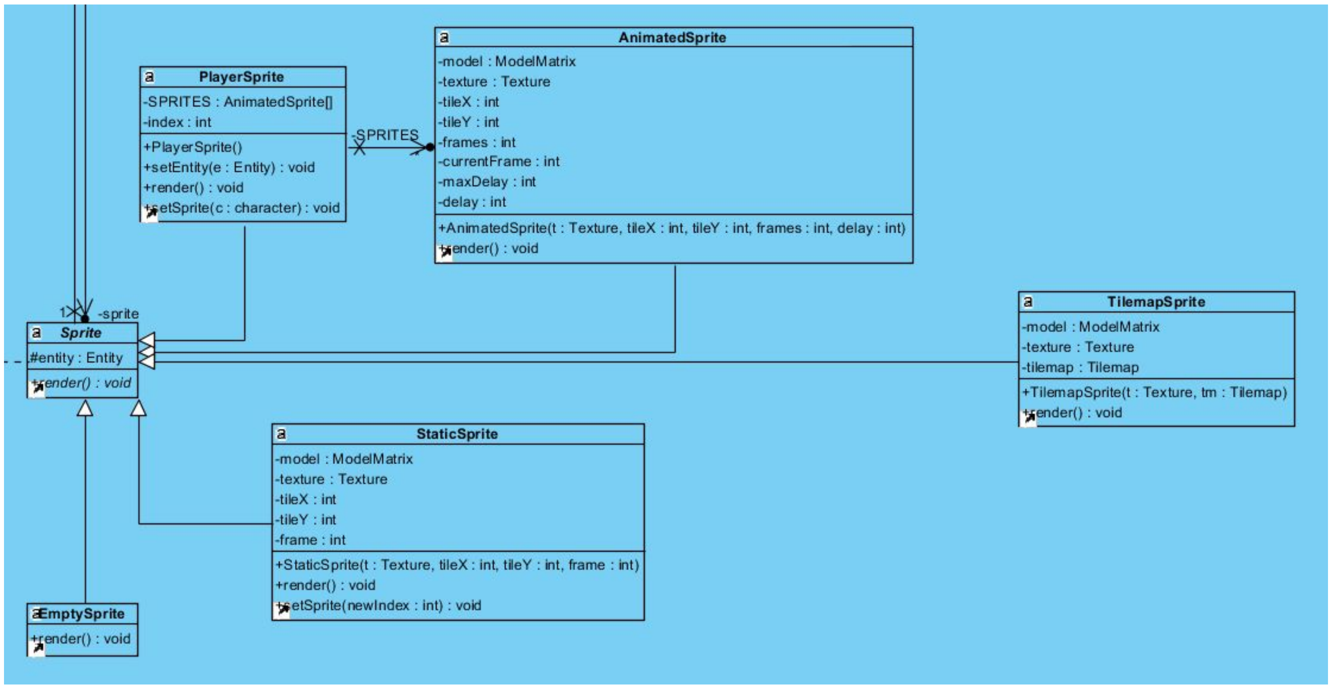
General Refactoring

- Component was changed to an interface instead of a class.
- GameWorld had functionality added to store constants such as the number of enemies, as well as information needed to create a camera (which OpenGL, our chosen library, does not do for us). It was also converted to a Singleton.
- EntityFactory was added to create whichever sort of Entity is needed whenever it is needed.
- Functionality to keep track of whether an Entity is active was added to the Entity class, in order to recycle Entities efficiently.
- Movement is now controlled by Entity rather than PlayerBehavior so that it can be used by anything that needs to move.
- Sprite has been expanded into a package that contains classes PlayerSprite, AnimatedSprite, StaticSprite, TilemapSprite, and EmptySprite, in order to keep it from becoming a blob.
- Collider has also been expanded into a package containing TilemapCollider, EmptyCollider, and BoxCollider.
- Subclasses of Behavior that have been added include CosmeticBehavior (for the collectible items) and HatBehavior (specifically for hats).
- PlayerBehavior has been significantly expanded. We debated at length before each addition about whether it was becoming a blob, and ultimately concluded that all of the functionality included in it belonged there. CharacterInfo and character are a class and an enumeration that we concluded were not behavior-oriented enough to belong in PlayerBehavior, and so were separated. We also pulled out the special behaviors for the Rat, Possum, and Raccoon characters and had them not inherit from Behavior, which might seem counter-intuitive, but they're really sub-behaviors and don't need to implement the Behavior interface. PlayerBehavior now has an instance of each of these.
- Food Behavior was expanded so that it could be consumed, and the enumeration FoodType was added in order to keep track of which characters gain health from eating which food.
- EmptyBehavior was added as a default Behavior.
- Map was deleted since it only handled tasks GameWorld should be keeping track of anyway.

Design Patterns

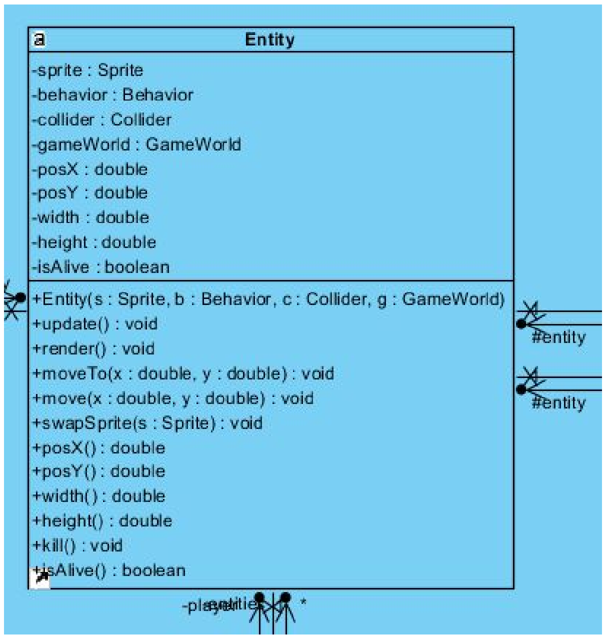
Flyweight

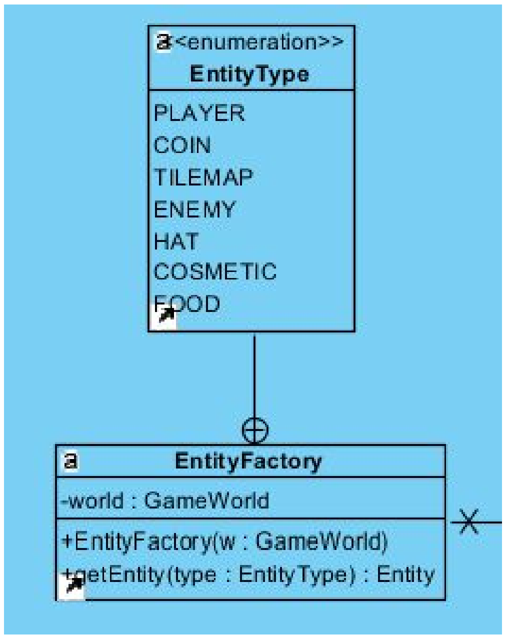
In order to prevent unnecessary re-loading of textures, we used the Flyweight pattern to implement our sprites. The extrinsic portion of our sprites (classes AnimatedSprite and StaticSprite) is their position in the gameworld, represented by tileX and tileY. The intrinsic portions are the texture each one uses, variables related to their animation behavior, their size, and so on.



Factory

All objects that the player interacts with or sees rendered on the screen in “The Trash Panda Dilemma” are represented as abstract Entities. To facilitate the ability to choose what sort of Entity to create at runtime, we used the Factory pattern to create EntityFactory. This can return an Entity of any type. The classes that inherit from Entity are listed in the EntityType enum so that viewers of the diagram can easily see what types EntityFactory can produce.



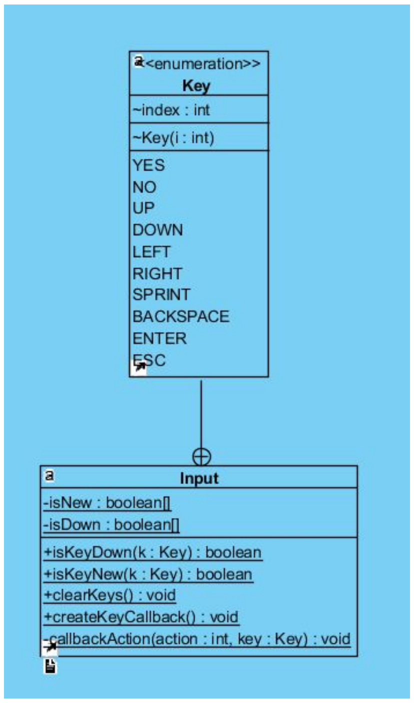


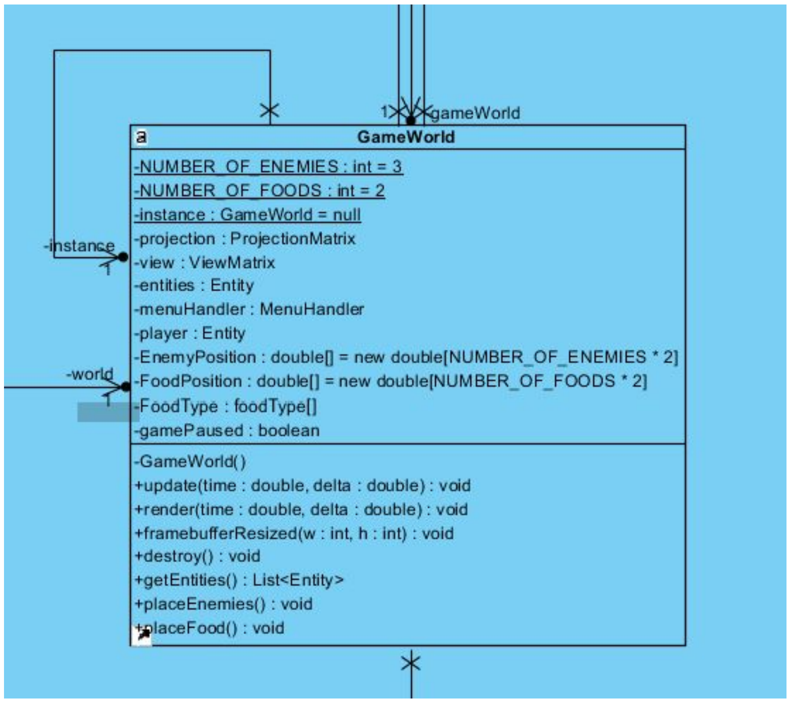
Strategy

In order to encapsulate the variation in the behavior of Entities, each Entity has an instance of a Behavior class. Behavior is an abstract class that implements Component and is inherited from by PlayerBehavior, FoodBehavior, CoinBehavior, and several others. The algorithm that determines an Entity’s behavior is selected at the time the Entity is created, and could be changed dynamically at runtime if necessary. Please see the Behavior package in the full class diagram; it is too large to put in this PDF at a visible font size.

Singleton

Two of our classes only makes sense if they are instantiated exactly once: the Input class, which handles key presses, and GameWorld, which knows things like how many enemies the game contains, camera positions, etc.. Most classes in the game need to be able to access these. In the case of Input, in order to avoid race conditions, only one Input handler should be created.





1
gameWorld
-instance
1
-world
1
GameWorld
-NUMBER_OF_ENEMIES : int = 3
-NUMBER_OF_FOODS : int = 2
-instance : GameWorld = null
-projection : ProjectionMatrix
-view : ViewMatrix
-entities : Entity
-menuHandler : MenuHandler
-player : Entity
-EnemyPosition : double[] = new double[NUMBER_OF_ENEMIES * 2]
-FoodPosition : double[] = new double[NUMBER_OF_FOODS * 2]
-FoodType : foodType[]
-gamePaused : boolean
-GameWorld()
+update(time : double, delta : double) : void
+render(time : double, delta : double) : void
+framebufferResized(w : int, h : int) : void
+destroy() : void
+getEntities() : List<Entity>
+placeEnemies() : void
+placeFood() : void

State

In the end, we didn't quite end up using a State pattern. We never have enough states that it made sense to have a class whose purpose was just to control which state the game is in. We have a "menu" state and a "gameplay" state, but a boolean in Input is a lot simpler than a whole class to tell Input how to handle key presses based on state. We also considered implementing State to keep track of which character the player is using. However, the solution we have now (an array of function pointers) seemed elegant, clean, easy to read, and simpler than having a "CharacterHandler" class, so we didn't use the State pattern there either in the end.