Overview

As specified in the project document, the AGE Engine employs the MVC design architecture to separate the logic, display, and user input. Derived from the abstract Model, View, and Controller classes, the AGE Engine provides concrete implementations of the Game, GameView, and Keyboard class.

GameView manages two Windows (which are wrapper classes for ncurses windows) that are responsible for displaying the 80 x 22 game screen and 3 status bars respectively. Keyboard takes in input from the user and maps it to an Action before flushing the input stream. In addition to the default key mappings, a constructor is provided which allows clients to create their own key mappings for games created with the AGE Engine. Finally, Game manages the game loop, which fetches input, processes game logic, and updates graphics on a regular clock cycle. The AGE Engine provides two clock classes, HalfSecondClock and CustomClock. Constructing a Game with a HalfSecondClock will make each game tick last 500ms, while using CustomClock will allow the client to specify a custom game tick rate. Calling the *go* method will start the game, and calling *stop* will stop the game at the end of the current game loop.

Game owns a World that supplies Game with the game logic that should be run in each game tick. The client will be able to specify game logic by deriving from the World class (as well as the Entity class, but that will be discussed later in the document) and implementing the *initialize* and *process* methods. *initialize* is run at the start of the game (i.e when *go* is called in Game) and *process* is called in each iteration of the game loop. As such, it is natural to put all of the logic for placing elements onto the game board in *initialize* and the actual logic of the game into *process*.

World also manages a number of Entities, which are automatically manipulated by a Physics class owned by World. World can switch between two concrete implementations of Physics, which are SolidPhysics (Entities can collide with the border) and ViewPhysics (Entities pass through the border). This is done by calling the *toggleSolidBorder* method. This means the behavior of the border can change during the course of the game, should the client choose to.

The Entity class is what all the game objects the client creates should be derived from. Entity itself is derived from a Body, which contains all the physics properties of the Entity, such as position, gravity, velocity, push (which is temporary velocity), player controls, bounding box, collision layer/mask… etc. This choice was motivated by a need to decouple Physics and Entity, and will be discussed later in the document.

The Entity class also owns a number of Sprites, which can either be StaticSprites or AnimatedSprites. Sprites manage Bitmaps, and provide the Entities graphics. In the World's *doProcess* method, in each iteration of the game loop, the World pushes every Entity's current sprite frame onto a render queue, which Game sends to the GameView to be displayed. The client can add many Sprites to a single entity, and freely switch between forms in their game logic.

The client should create their own game objects by deriving from the Entity class. Since Entities often need to have their own logic (e.g an enemy may move in a certain way depending on the location of the player character), which is what motivates Entities to have their own virtual *initialize* and *process* methods that need to be implemented by the client. These work in nearly the same way as the *initialize* and *process* methods in World, and are guaranteed to run every game tick (this is implemented by extensively using the template method pattern, and will be discussed later in the document).

Apart from these two methods, clients must also implement *collideX*, *collideY*, *collide*, and *pass* in their custom Entities, which are virtual methods inherited from Body. These implementations will define the logic of how their Entities behave during collisions with other entities or the border, and are called by the Physics classes when they detect specific conditions while manipulating the positions of Bodies. When two Bodies collide in the X or Y directions, *collideX*, *collideY* are called respectively (a pointer to the collided Entity is also passed in; it can be typechecked, and logic can be defined for collisions between specific Entity types). When a Body collides with a border, *collide* is called, and the border in question is passed in. Finally, when two Bodies pass through each other, *pass* is called (similarly to the *collideX/Y* methods, a pointer to the collided Entity is also passed in). The template method pattern is also used here so that certain checks are guaranteed (e.g a Body cannot pass through another Body twice in the same game tick).

Entities can be spawned and destroyed from the World with *queueSpawn* and *queueDestroy* (and are template methods so that all classes that Entity is a base of can be added to the World). They can be called from Entity as well, so that Entities can be easily spawned by other Entities. The way Entities are spawned is not by adding them directly into the World, as this would cause multiple inconsistencies with how Entities interact depending on what phase they were introduced. For example, suppose Entity A was added into the World during World's *process*, and Entity B was added by a different Entity (e.g an enemy splits in two). Since World's *process* is called after all of the Entities processes, Entity B would exist before Entity A, and that may have serious consequences on the game logic and introduce ambiguity. Therefore, when *queueSpawn* is called, the spawned Entity is pushed onto a spawn queue, and all spawns are added to the World at once after all calls to game logic. Similarly, when an Entity is destroyed with *queueDestroy,* what actually happens is that it is flagged and only removed from the World after all calls to game logic.

This is the basic structure of the AGE Engine. The engine tries to handle all aspects unrelated to game logic, and tries to provide templates for the client wherever game logic is needed. As a final wrap up to the overview section, the steps of how a game is run from start to finish in the AGE Engine will be described:

1. *go* is called
2. World is initialized (by calling World's *doInitialize* method)
    a. World's *initialize* method is called (client implemented)
    b. Entities are spawned into World
        i. Each Entity's *initialize* method is called (client implemented)
    c. World pushes all Entity frames onto a render queue in order of height
3. Graphics are updated in GameView
4. Game loop begins
5. Fetch input from Keyboard
6. Game logic is processed (by calling World's *doProcess* method)
    a. Physics steps through each Entity in World
    b. Each Entity's *process* method is called (client implemented)
    c. World's *process* method is called (client implemented)
    d. Entities are spawned into World
        i. Each Entity's *initialize* method is called (client implemented)
    e. Entities are destroyed from World
    f. World pushes all Entity frames onto a render queue in order of height
7. Graphics are updated in GameView
8. If stop() was not called during the game loop, go back to step 5 after the game tick is over

Design

The template method pattern was extensively used in the AGE Engine. All methods that the client is required to implement (i.e game logic methods), are template methods. This is because in any given game tick, numerous things besides client defined game logic needs to occur. For example, *process* is a template method that is used in *doProcess*, because besides game logic, the World also needs to allow Physics to step through Entities at each height level, call the *process* methods in each Entity, spawn/destroy Entities, and add Bitmap and Points to the render queue during each iteration of the game loop.

One design challenge was allowing the client to specify how two Entities on the same height would interact. For example, how could you allow the client to decide which Entities would collide and which would pass through each other? Initially, ideas involved using the visitor pattern, but as the number of types of Entities increased, that would blow up the number of methods that needed to be implemented. The solution for me was to bypass this entirely and give each Entity a collision layer and mask and use the bitwise AND of two Entities' layer and mask to determine if they would collide. For example an entity with mask 0b001 would collide with all entities on layer 0bXX1 (where X could be either 0 or 1) since 0b001 & 0bXX1 != 0. On the other hand, it would pass through all entities on layer 0bXX0, since 0b0001 & 0bXX0 == 0. Since layer and mask are both unsigned integers, it should provide enough combinations of collisions for any game the AGE Engine may be used to implement.

Another problem I encountered during the design of the AGE Engine was restricting where certain methods could be called. For example, *doProcess* in World should only be called from the Game class since it defines the logic of the game in a single game tick. Calling *doProcess* from anywhere else could cause serious problems. For example, if *doProcess* is called in *process*, the methods may just recurse infinitely. One solution could be to make *doProcess* private and make World a friend of Game, but that would allow Game access to too much of World's interface and weaken encapsulation. Instead, we create a Key<Game> class that is a friend of Game's and has a private constructor so that it can only be constructed inside Game. Then we make *doProcess* take a Key<Game> as an argument so that it can only be called if the caller has access to a Key<Game> object. Since Key<Game> can only be constructed inside Game, only it can create the object needed to call *doProcess*. This way, *doProcess* can be called without exposing the entire interface of World to Game. This is apparently called the Passkey idiom and is used in the AGE Engine somewhat frequently (e.g in Entity's *doProcess* and *doInitialize* methods).

Deviations from original design

The largest change made to the AGE Engine was the addition of the Body class that Entity is derived from. Originally, Physics stepped through Entities themselves, rather than Bodies. However, this was a coupling issue since Physics was exposed to the graphical interface (e.g Sprites), to the spawning interface, (e.g *queueSpawn*), and many other aspects of the Entity class unnecessary for physics handling. Now, the Body class contains all the physics characteristics of an Entity, which are the only things Physics needs to do its job, and as such, the only interface it should be exposed to.

The Body class also replaces the Movement hierarchy which originally used the decorator pattern to modify movement characteristics. This idea was scrapped because, while a good idea in theory, in practice, for the client, movement was difficult to define, and difficult to modify in the game logic (e.g, it was incredibly difficult to change the movement direction without adding more and more decorators). Body removes all the decorators and gives them each their own field (e.g velocity field, gravity field, player controls field). While this in a way may make movement less open for extension (a violation of the open-closed principle), this is a good tradeoff, since it ultimately makes movement much more usable.

During the development of the first game is also when many of the inconsistencies with spawning/destroying entities that were described earlier in the design document were discovered, and the choice was made to make spawning and destroying occur all in one phase after the game logic.

Besides some name changes for classes to make them more readable, not much else changed.

Extra Features

Memory management is done entirely using RAII

\

What would I do differently?

There's two major things I would redo:

1) While the collision layer/mask system works very well to determine if objects collide, determining what happens after the collision/pass through is an issue. For example, a client may want projectiles to collide with enemies, but do more damage to some enemies than others. To implement this without typecasting (e.g dynamic_cast) using the AGE Engine in its current form would be challenging. This means that along with the collision layer/mask system, I would like to implement some kind of visitor pattern that could perform this kind of double dispatch for determining what happens after collisions.

2) Currently the movement of everything is based on integers (using Point, which are (x,y) pairs), so you cannot get fractions of a pixel. Since so many classes rely on Points, changes to them would require changes to large parts of the project. I would like to shift away from this kind of system and implement more precise points that work at fractions of a pixel. Additionally, velocities are currently stored as an x,y Point. Instead, I would like to store velocities as a scalar speed and a unit vector so that, in conjunction with more precise points, the AGE Engine could simulate more realistic movements like acceleration, deceleration, gradual turning… etc. This is currently impossible since the only unit vectors that could be stored would be (1,0), (-1,0), (0,1), and (0,-1). This would obviously look janky with a 80 x 23 resolution, which is why I would also want to support higher resolutions.