CPS 108 VOOGA Individual Design

Name: Donghe Zhao
Date: April 26th
Genre: Fighting Game
Teammate: Hareesh, Helena, Chen, Peggy, Wendy, Hui
UTA: Nathan, Tanner
GitLink: https://github.com/hareeshganesan/Vooga/tree/master/src/PhysicsEngine

1. Introduction

The Physics Engine is made up by two aspects, Physics part and Collision Part. The Physics part deals with all Motion input, one of the most important functions is to calculate the next location increment. Collision part checks whether there is collision, and sets detailed collision status to sprites, and does some reaction.

In a certain game, we can create a PhysicsEngine by passing in the gameEngine. Like the example below:

```
private PhysicsEngine myPhysicsEngine;
myPhysicsEngine = new FightPhysicsEngine(myEngine);
```

We also need to initialize the collision by three steps in CombatInstance with a following example below:

1. Create an instance viable Collision and another instance viable SpriteTemplateGroup in CombatInstance, like our demo game below:

```
// Collision
private Collision myCollision;
private SpriteGroupTemplate groupSprite;
```

2. Set the Collision in “initResources” method . The Collision needs three things to pass in.

    - The SpriteTemplateGroup: contains some teams and each team has some sprites in it. We only check the collision between sprites from different teams.
    - A list of CollisionKinds or a CollisionKind: the kind here means the requirement of these two sprites. CollisionKindFriends works for two FighterBodies with the same GroupID; CollisionKindEnemy works for two FighterBodies with different GroupIDs; CollisionKindNeutral works for one sprite and one platformBlock. Also, CollisionKindCustom can help developers to create other CollisionKind easily.
    - The specific PhysicsEngine

```
@Override
public void initResources() {
    …
    groupSprite = new SpriteGroupTemplate();
    groupSprite.addPlatformBlockArray(platform);
    groupSprite.addFighterSpriteArray(playerSprites);
    groupSprite.addSpriteArray(nonplayers);
    ArrayList<CollisionKind> CollisionkindList = new
    ArrayList<CollisionKind>();
    CollisionkindList.add(new CollisionKindFriends(new
    ReactionMomentumConservation()));
    CollisionkindList.add(new CollisionKindEnemy(new ReactionPush()));
    CollisionkindList.add(new CollisionKindNeutral(new ReactionRebound()));
    myCollision = new Collision(groupSprite, CollisionkindList,
    myPhysicsEngine);
    …
}
```

3. In the update of CombatInstance, check the group collisions every updated time.

```
@Override
public void update(long elapsedTime) {
    …
    myCollision.checkGroupCollision();
    …
}
```

2. Deatailed Description

1. Physics Part

PhysicsEngine is the super class of physics engine. If developers want to create their own engine, then just make a subclass of that. There are two abstract methods need to be implement in sub class.

- ```
  public abstract void process(MotionAction motionAction, long elapsedTime);
  ```

  Deal with any MotionAction from inputHandler, calculate new location

- ```
  public abstract void setNextLocationIncrement(SpriteTemplate sprite,
              double dx, double dy);
  ```

  Set this sprite's location increment.

FightPhysicsEngine is an implement for physics engine. It has the following features, all of which can be set.

- `myBackgroundFactor`

  All the movement will multiply this factor. Its default value is 1.0. For example, if we develop this game in the water, maybe we need to set it to 0.8.

- `myOutBoundDistance`

  When sprites hit the up bound, left bound or right bound, we make them a rebound with this certain distance.

- `mySpeedFactor`

  This factor works for the input MotionAction. After getting the speed from Motion, it will multiply this factor as a final speed. In other word, it can control speed from MotionAction independently.

2. Collision Part

This is a basic class for collision issue. When creating it for a game, it needs to pass in

three arguments as the introduction part above. Outside this class, we can do some simple change to the three arguments, including getCollisionGroup, setCollisonGroup, addCollisonKind, removeCollisionKind.

2.1.Collision Kind

This is a super class for all kinds of collision. Every collision kind has one or more reactions associated with it.

- `public abstract boolean isThisKind(SpriteTemplate spriteOne,SpriteTemplate spriteTwo);`
  Implement this in sub class to check what kind it belongs to. For example, two FighterBody with the same GroupID.
- `public void doThisReaction(SpriteTemplate spriteOne,SpriteTemplate spriteTwo, PhysicsEngine physicsEngine)`
  Every Collision Kind has its own reactions. Here we do it one by one.

Also, the reaction associated can be changed, including addReaction, removeReaction, setReaction.

2.1.1. Collision Kind Enemy

The Collision between two bodyFighter with different GroupID.

2.1.2. Collision Kind Friend

The Collision between two bodyFighter with the same GroupID.

2.1.3. Collision Kind Neutral

The Collision between some sprite and a block.

2.1.4. Collision Kind Custom

This class helps developers to create more kinds of collision. After creating this class with its reactions passing in, it needs to setType:

```
public void setType(Class<?> classOne, CollisionKindCustom.Type typeOne,
CollisionKindCustom.Relation relation, Class<?> classTwo,
CollisionKindCustom.Type typeTwo)
```

Here are five arguments. The type means what relation between our certain class and the sprites we need to check (is, super class or sub class). The relation means whether these two sprites have to meet the requirements at the same time.

2.2.Collision Reaction

This is the super class of all collision reaction. It needs sub class to implement:

```
public abstract void act(SpriteTemplate spriteOne, SpriteTemplate spriteTwo,
PhysicsEngine physicsEngine);
```

It is just the specific reactions we need these two sprites to do. 2.2.1 to 2.2.6 are six common reactions. The developers are allowed to write their reaction inheriting Reaction.

2.2.1. Reaction Force

This reaction works in the case that sprites have a power to others. It has four instance variables, two sprites' power in horizontal and vertical direction. The location increment is the other's power over its own mass. For example,

```
dx1 = forceX2 / massOne;
```

It allows developers to set these four forces.

2.2.2. Reaction Momentum Conservation

This reaction is for the case that the collision under the law of momentum conservation and the law of energy conservation at the same time. This should happen totally under idea physics environment. So nothing can be changed.

The location increment is like below:

```
private double getIncrement(double m1, double m2, double speed1, double
speed2) {
    return (m1 - m2) * speed1 / (m1 + m2) + 2 * m2 * speed2 / (m1 + m2);
}
```

2.2.3. Reaction Punch

This works for the case that one sprite punches another to a certain position. For example, punch someone to the wall. It allows developers to set the bound in left, right, up and down.

2.2.4. Reaction Push

This class works for the case that one pushes the other and these two go together

2.2.5. Reaction Rebound

This class works for the collision between sprite and a block. It has four instance variables, which can be set, to control the activity of rebound. If some bound is active, when sprite collided to the block at this bound, it will have a rebound distance which can also be set.

2.2.6. Reaction Stop

This class works for the case that two sprites collide and then we want them both stop with a certain distance between them. The certain distance can be set.

3. Design.

1. System Level

For the system level, CombatInstance has an instance variable PhysicsEngine. During each system updated time period, we have the following things happening.

InputHandler can get this PhysicsEngine. InputHandler deals with different keyboards to specific MotionAction. In each MotionAction, it will call PhysicsEngine, in which the sprites collisionStatusOnGound would be set, to calculate new location. With the new locations of all sprites, the system checks collisions. If there is any collision, we'll set CollisionStatus to these two sprites with detailed collided edges and find what type the collision it is by CollisionKind, which has several Reactions in it. Then do its associated Reaction, in which we also need PhysicsEngine to calculate the new location for this reaction. In the system update, it will see the CollisionStatus for every sprite, if it is collided it will update the new location from collision reaction, else it will update the new location from the input MotionAction. Then set all sprites' CollisionStatus as default, all false.

The most challenging problem here is the CollisionStatus part. If we don't have this

status for sprites, every time system update, the new location will be the sum of MotionAction and Collison Reaction. To solve this problem, I give each sprite an instance variable that keeps its CollisonStatus. In this way, PhysicsEngine sets next location increment is a “soft” setting. While in the system update, use CollisonStatus to check either MotionAction or Collision Reaction needs to be really set. Furthermore, I give detailed CollisionStatus for sprites to keep the exact collided edge, up down left and right, and whether it is standing on sth. These five status help a lot when we develop reactions. For example, if one is on the ground or blocks, the other is colliding downwards to it, they are supposed to stay there however if we don’t have collision detailed information it will be hard to implement it.

2. Collision Level

For the lower level, I would like to talk about the collision design. The collision needs three arguments. One is the group of sprite. I define the group is made up by teams, and teams are made up by sprites. I only check collisions between sprites from different team. In this way, we can check collision easily and efficiently. After collision triggered, the CollisonKindList will go through and find which CollisionKind it is, if we find it we do its associated reactions. The reactions in the CollisionKind could be set. I was inspired by Factory Method Design Pattern, collision kind is like what we create, but let the sub classes to decide which kind it should be. I also wrote some common reactions for fighter games. If developers want to create new kind of collision, just create a CollisonKindCustom and then setType. If

developers want to create a new reaction, they might want to create a new class inheriting Reaction.

The shortcoming of this collision design is that it needs developers to define teams carefully. And it is not convenient to modify some sprites in a certain team. For example, one collided sprite divides itself to two parts, in this design we have to find the team index in the whole group, then remove this sprite from this team, then add the two new sprites in this team or other teams if we want.

Here I give an alternative design. The group is just made up by all sprites and then is passed into the Collision. Then we check any pair in this group to determine which kind of collision it is and do reactions. This design would be easier to develop, including the initialization and changing during games. However, this design would not be as efficient as the design I took. If the number of sprites is large, it might be slow.

3. The UML is below.