Group Design Document

Vimeo Link: https://vimeo.com/41137497

The Game Loop

The framework uses a GameEngine object (borrowed from Golden Tee) to begin the main game. The main method resides here. The MainGame provides a sample state engine system that can be used to begin the game. A Title screen and a CombatEngine (the actual gameplay engine) are generated and returned as needed in the getGame method.

Adding Multiple Game Play Screens

To add more screens, you need to set the flow of the screens, ensure that they flow from one to another. To do this, make each screen extend the GameState class (this class extends GameObject). The screen to be transitioned to is contained in the instance variable *nextstate* and the screen transitioned from is in *laststate*. It can be set with the appropriate setters. The transitionState method, abstract in GameState, should take care of the logic for determining and switching to the next state. A sample implementation is below.

```
public void transitionState() {
            if (nextState != null)
                  myEngine.nextGame = this.nextState;
            else
                  myEngine.nextGame = this.lastState;
            super.finish();
}
```

Creating Combat Instances

Overview

The main game loop of CombatInstance runs according the Golden Tee Game Framework. The framework calls the update method in the GameObject (CombatInstance) that is running. After update, the render method is called. The CombatInstance is initialized using a LevelObjectsFactory, which initializes the CombatInstance and parses from XML.

To see this without writing code, simply instantiate a new CombatInstance, and point it to the XML file with information regarding the properties of the game.

To add an instance of combat (analogous to a level) to a game, you must create a CombatInstance and add it as a GameState that is reached by some transition method. CombatInstances control the all of the gameplay mechanics of a particular level and are highly extensible. CombatInstances are repositories for all of the information contained in a level, including the fighters, non-player sprites, the physics engines and the camera. It is possible to extend and then hardcode all of the necessary variables, but a LevelObjectsFactory has been provided to instantiate the level from an XML file.

The LevelObjectsFactory is where parsing of the XML happens. To parse a new tag from the XML, create a method to instantiate the actual objects, and then call it from the *initResources* method in CombatInstance.

LevelObjectsFactory.java

```
public ArrayList<SpriteTemplate> createPowerUps () throws JDOMException
    {
        List<Element> b = findAllInstancesOfElement("Power");
        ArrayList<SpriteTemplate> fs = new ArrayList<SpriteTemplate>();
        for (Element e : b)
        {
             System.out.println("create power");
             HealthSprite s =
             new HealthSprite(c.getImage(e.getChildText("img")),
                         GroupID.getIdFromString(e.getChildText("id")),
                         Integer.parseInt(e.getChildText("damage")));
             s.setLocation(Double.parseDouble(e.getChildText("x")),
                         Double.parseDouble(e.getChildText("y")));
        }
        return fs;
      }
```

CombatInstance.java :: initResources

```
try {
       LevelObjectsFactory lof = new LevelObjectsFactory(this);
             …
             …
       powerUps = lof.createPowerups();
    } catch (JDOMException e) {
       e.printStackTrace();
    }
```

Input Handling

Keyboard input is the form of input that is currently supported by our framework. Keyboard input is handled by the InputHandler, a necessary instance variable in all GameStates with keyboard input. The InputHandler plugs directly into the Golden T loop with its update method. On each update call, the InputHandler iterates through its map, checking if any of the keys in its map are pressed. If they are pressed, the action that is mapped to the key is triggered by calling the *performAction* method.

```
InputHandler h = c.getMyHandler();
h.addKey(map[1], MotionAction.DOWN(s,myPhysicsEngine));
```

To add support for a key to a screen, call the addKey method to add the key that triggers the input and the corresponding Action object. All CombatInstances have an InputHandler instance variable that can be accessed using the *getMyHandler* method.

The InputHandler contains default mappings for Fighters that are instantiated. These can be changed as needed by editing the default mappings method.

Actions

Actions are generic objects that allow for actions to be executed whenever their perform action method is called. Actions are used primarily in pair with the InputHandler but also as a way of effecting change for an AI agent. Actions can also be paired with InputHandlers on non-combat screens to enable easy option selection.

In order to enable multi-action groups, an ActionSeries object that implements Action is used. The ActionSeries object tree maps an action to a Long key. On each update, the time in the action series will increment. The action with the key that is closest to the current time being tracked by the action series is selected. As the key finding is done with a TreeMap, we maintain efficiency. The ActionSeries can also be reset and reused, so new objects need not be instantiated for each frame. ActionSeries require that a TreeMap with times mapped to Actions are entered in order to construct the object.

To implement variable time actions, a Goal object is used. The GoalSeries object ultimately determines the action performed. The GoalSeries knows whether it has completed successfully or unsuccessfully. As long as success states continue without the goal being achieved, the GoalSeries should continuously perform an Action. Once the goal is achieved or the cutoff time is reached, the goal ends. To extend a Goal, the condition that the goal ends on must be specified by implementing *updateGoalState*.

These ActionSeries and Goal objects can be instantiated and used as follows.

```
TreeMap<Long, Action> actions = new TreeMap<Long,Action>();
actions.put((long) 5000, MotionAction.UP(myFighter, myPhysicsEngine));
ActionSeries a = new ActionSeries(actions);

Goal a = new Goal((Action) action, (long) cutoff);
```

The ActionSeries and Goals will then update their time whenever the performAction method is called. The isDone method can be used to check when they have completed running.

Some actions can only be performed in certain states, and cannot be repeated indefinitely (jump, or weapons). To accomodate this, the ActionTimer is used. FighterBody objects have a list of ActionTimers, *myTimers*, that should be checked for such Action types.

To add a timer:

```
myTimers.add(new ActionTimer(500));
```

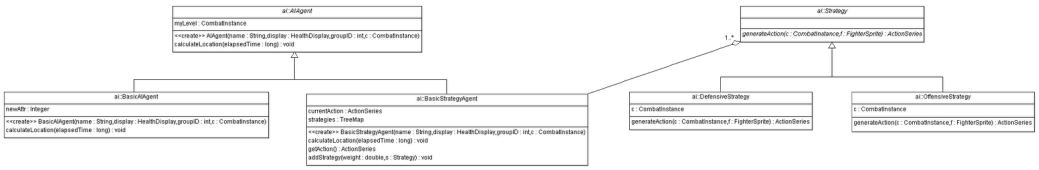
To update the timer and check if the action can be performed (if the timer is available and there is time left):

```
if(myFighter.getMyTimer(0).action(elapsed_time)){
            myPhysicsEngine.process(this, elapsed_time);
      }
```

To make a timer available when an event occurs:

```
if(this.getCollisionStatus().getStandOnSth()){
            myTimers.get(0).makeAvailable();
            }
```

Timers are generally used to filter a performAction for an Action, determining whether that action can be performed.



Sprite/Animation

Sprite Definition:

In developing this framework, our goal was to allow a fighter game developer to create sprites and define their animation without having to draw extra sprites. Contrary to the conventional way of defining fighter sprites where a fighter is made up of numerous pre-drawn images, each sprite is defined by a tree of limbs, each has the flexibility of rotating about its joint linked to the parent node.

Additionally, the developer may create a basic sprite and pass in various properties and reactions to collisions, which allows for greater diversity in sprites than having to subclass sprites every time a new functionality is needed.

Creating a fighter body tree is as follows:

****Note that creating a root node requires a different constructor as its limb nodes**

```
//below is a constructor for root, takes the name of the root node, a BufferedImage image
visually represents the root, a groupID (an identifier to a fighter, all limbs share the same
groupID, see GroupID.java for more details), the x and y coordinates of the sprite's location,
and the amount of damage this node deals:

NodeSprite torso = new NodeSprite("torso", imgTorso, GroupID.PLAYER_1,
this.getWidth()/2, this.getHeight()/2, -5);
//note: damage field is always a negative number

//below is a constructor for limb, takes the name of the limb, a BufferedImage, a pointer to
its parent node, dx and dy from the top left corner of the parent node, damage, and an angle
theta the limb should rotate by (about its center):

NodeSprite LeftArm = new NodeSprite("LeftArm", torso,imgLA,-(torso.getWidth()/2),0,-
3, 45);

//myFighter is an instance of FighterBody, which takes a pointer to the root of the tree, the
name of the fighter (shown with health display), and the health display bar:

torso.addChild(LeftArm);
myFighter = new FighterBody(torso, "fighter1", myDisplay);
```

Dynamically attach/detach a node:

```
//add a node, takes the name of the parent node, and the new node to be added:

BufferedImage newLimb = GraphicsTest.loadImage("src...");
NodeSprite NewLimb = new NodeSprite("newLimb",myFighter.getNode("LeftArm"), newLimb,
0, 40, 0);

myFighter.add("LeftArm", NewLimb)

//detach a node, simply pass the name of the node to be removed:

myFighter.removeChild("newLimb");
```

Animation Definition:

In this framework, animation is defined in terms of a timeline of motions (Motion.java) and each motion's start time (in terms of elapsed time). Each FighterBody holds its own list of animations, each animation denotes a possible set of movements that a fighter may perform. (i.e. walking, punch, jump, etc.) A Motion object is constructed as follows:

```
//Each Motion is defined by passing the name of the limb to be rotated, the expected angle,the
FighterBody associated with the motion, and the duration of the animation.

Motion m1 = new Motion("RightLeg", -80, myFighter, 500);

//Each Animation is constructed with a HashMap<Long, Motion> that holds all of the motions to
be performed in the animation sequence, and the FighterBody associated with the sequence
```

```
HashMap<Long, Motion> sequence = new HashMap<Long, Motion>();

sequence.put((long) 1000, m1);
// 1000 is the start time of the animation, i.e. after 1000 milliseconds, start this motion

animation = new Animation(sequence, myFighter);
```

Activate an animation:

```
//to activate and perform an animation:

animation.activateAnimation();
…
if(this.animation.getStatus()==true){
       animation.update(elapsedTime);
}
```

Artificial Intelligence Agents

The AI system was designed to allow the developers to easily create and extend AI agents tha execute certain behaviors. At the most basic level, an AI agent can be created and used by initializing the AI agent and adding it to the list of fightersprites.

```
NodeSprite body = new NodeSprite("ai1body",
             c.getImage("resources/flame.png"), GroupID.PLAYER_AI, 400, 500,
             0);
BasicAIAgent ai = new BasicAIAgent("ai1main", body, new HealthDisplay(
             50, 50, c.getWidth() / 2 - 30), 0, c);

ai.setRoot(body);
ai.setDefaultSpeed(1.5);
fs.add(ai)
```

AI must implement the calculateLocation method to function.

A more complicated agent can be easily instantiated with the BasicAIStrategyAgent.java. These AI agents, instantiated in the same way, perform a strategy from their map of strategies, and then select a new Strategy randomly. The TreeMap should contain doubles that map a random number weight to a strategy. Take care to make the weights cumulative and ensure that they reach 1.

```
NodeSprite body = new NodeSprite("ai1body",
             c.getImage("resources/flame.png"), GroupID.PLAYER_AI, 400, 500,
             0);
BasicAIAgent ai = new BasicAIAgent("ai1main", body, new HealthDisplay(
             50, 50, c.getWidth() / 2 - 30), 0, c);
ai.addStrategy();
```

```
        ai.setRoot(body);
        ai.setDefaultSpeed(1.5);
ai.addStrategy(1,new OffensiveStrategy(ai, c));
ai.addStrategy(.5,new DefensiveStrategy(ai, c));
```

The most advanced Strategy agent is the SituationalStrategyAgent, which utilizes Situation classes (e.g. DefaultSituation.java) to define the Strategy set and how a Strategy is selected.

```
public class DefaultSituation extends Situation
{

    public DefaultSituation(FighterBody myFighter, CombatInstance c){
        strategies.put(1.0, new OffensiveStrategy(myFighter, c));
        strategies.put(.5, new DefensiveStrategy(myFighter, c));

    }
    public boolean isOccurring ()
    {
        return true;
    }

}
```

Strategies are selected in the Situation class by if they are occurring, and then their order. To add a new Situation like critical health, do something similar to the code below.

```
public class CriticalHealthSituation extends Situation
{

    public CriticalHealthSituation(FighterBody myFighter, CombatInstance c){
        strategies.put(0.1, new OffensiveStrategy(myFighter, c));
        strategies.put(.7, new DefensiveStrategy(myFighter, c));
        strategies.put(1, new HealthStrategy(myFighter, c));

    }
    public boolean isOccurring ()
    {
        return myFighter.health<20;
    }

}
```

The Strategy class is an abstract class that can be extended to implement specific strategies, and get new Goal objects based on the strategy selected. These strategies can be used in the selection of the next action for a StrategyAgent. To make a new strategy, the Strategy class must be subclassed. The Strategy agent must declare the specific list of goals it must accomplish in the constructor. Specific goals were defined as inner classes to better organize the goals associated with a Strategy.

To subclass a Strategy, you must implement the initialization of the Goals to specify which goals to use. For example, see below.

```
public void initializeGoals ()
    {
```

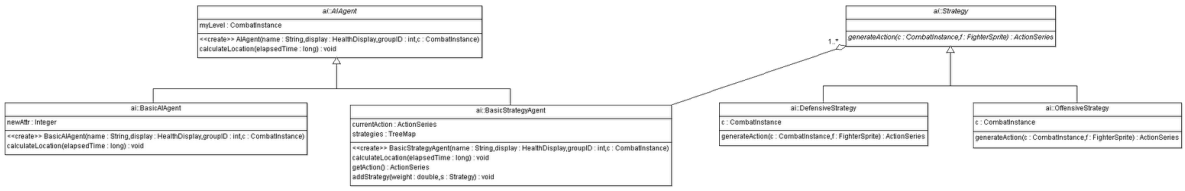
```
        goals.add(new FollowGoal(myFighter, c.getFighters().get(0), this.c));
        goals.add(new AttackGoal(myFighter, c.getFighters().get(0), this.c));
      }
```

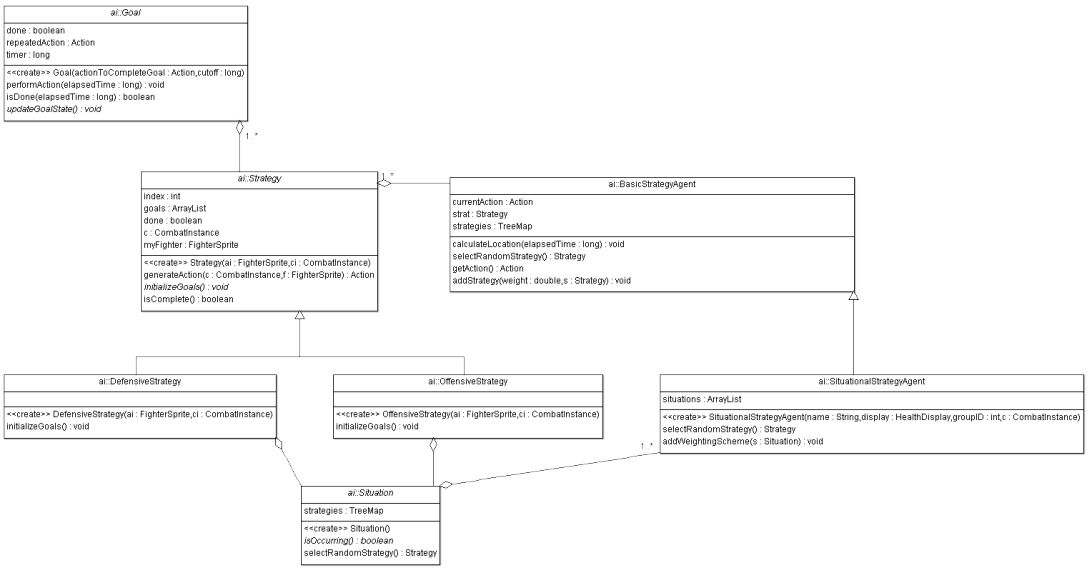
Goals can be used to specify variable-length repeatable Action sets that accomplish a goal.

```
private class AttackGoal extends Goal
    {
        FighterBody myFighter;
        FighterBody myEnemy;
        public AttackGoal (FighterBody me, FighterBody enemy, CombatInstance ci)
        {
              super(new WeaponAction(me, 0, ci), 10000);
              myFighter = me;
              myEnemy = enemy;
        }
        @Override
        protected
        void updateGoalState ()
        {
              if(this.repeatedAction.isDone(0))
              done = true;
        }

      }
```

AI Agents





Additional Sprite creation:
The SpriteTemplate allows for compositing new sprites by passing in PropertyObjects and CollisionEvents, and creating new subclasses of PropertyObjects and CollisionEvents.
Example health power-up:

```
SpriteTemplate s=new SpriteTemplate(BufferedImage b, GroupID g);
        s.setLocation(x,y);
        s.addProperty(HealthProperty.getName(),new HealthProperty(50));
```

Level Editor

The Level Editor facilitates the development of a new level by enabling the game designer to customize various features of a level using a Graphical User Interface (GUI) rather than simply hard-coding code from scratch.

More specifically, the Level Editor GUI provides the options to assign to each created class a level name, unique background image, and different non-Fighter Sprites. Fighters themselves are designed through the separate Character Editor, although the Level Editor compasses the ability to place an AI sprite in the level, effectively determining the starting point of the enemy.

The Level Editor essentially is broken down into two stages:
1. Designing level with GUI
2. Saving to XML

1. The developer can easily modify the physical design (visual layout) and functionality of the

GUI. Rather than including all the GUI code in one large class, each component of the GUI is contained in its own class; for example, the Save button and the Background Image each represent a distinct class. A couple of classes are used to group together related components, similar to a Composite pattern, but the primary **View** class, which subclasses JFrame, contains references to each of these smaller individual components and also sets the layout for all of them in the overall GUI window.

```
add (new LevelNameComponent(myController));
add (new BackgroundImage());
add (new SpriteEditor(myController));
add (new AIEditor(myController));
add (new SaveLevelComponent(myController).create());
```

Currently, the View uses a BoxLayout, hence items are displayed vertically in a top-down pattern depending on the order in which they are added. However, it is possible to edit or even subclass View and change the layout and the positions of individual components, as well as their sizes. Components can be removed or added as needed to suit the needs of the developer who determines what options should be available on the GUI.

2. In order to facilitate the transfer of information between the View and the XMLWriter object, three intermediate layers are used: Controller, Model, and LevelObject. The first two work alongside the View as part of the standard MVC pattern. A LevelObject stores all the information or objects associated with a particular level, and these properties are then written out using the XMLWriter.

The LevelObject stores a list of *SpriteEditable* objects. Since there is no strong need to fully load and instantiate classes at runtime as they are dynamically added, the SpriteEditable object contains a Hashmap that maps names of properties to their corresponding values. These values are then stored with each SpriteEditable and their values are extracted from the map and written out to the text file with the XMLWriter which simply iterates through the map.

Character Editor

Character Editor let the developer to create a fighter using GUI. The GUI has three parts: the ComponentPanel from which the developer can choose a body part, the MainPane, where drag and put happens, and the AttributePane, where developer can set properties of each body parts and their fighter.
To create a fighter, the develop should first select the body part by click one of the rectangle on the left side. Then he can put that part in the MainPane by click anywhere in the MainPane. After that, a rectangle should appear where the developer just clicked.
The developer could continue add body parts in the MainPane, or he can start to edit the body parts in the MainPane.

Here are what a developer can do to edit a part

a) You can change a body part's location by, first select the body part you want to change location. The selected body part should be colored orange, and the rest unselected component are green. Then drag it to wherever you want.

b) You can change a body part's size by, first select the body part you want to change size. And move the cursor to its lower right corner, then cursor should change its appearance to indicate ready to change size. Finally, drag the corner to change the size.
c) Add properties to a certain body part by, first select the body part. Then select the properties you want to add from the ComboBox. Click add button to the left of the ComboBox. Then input the value of that property.
d) Remove a properties to a certain body part. Exactly the same as c) unless do not enter anything in the pop up dialog. That property will be canceled.
e) Give a certain body part a name by, first select the body part. Then input its name in the name TextField and then press Enter. Do not name two body parts the same name.
f) Add a Image to a body part by, first select the body part. Then click the add img button and chose a image. The Image should be provide by the developer. The GUI cannot create a picture.
g) Delete a body part in the MainPane. Double click the body part you want to delete.
h) Connect two body part by, first select the parent. Then press ctrl and hold it and click the second body part as child. The child body part should colored blue. Right click you mouse and select connect. Then the two body part connect together. A arrow point from parent to its child indicate the relationship between the two body part.
i) Manipulate a group of body parts by, click anywhere other than a body part in the MainPane and drag a black rectangle. All the body parts contained by the rectangle will be selected and colored pink. Then manipulate them at the same time. For example, delete them or change their location.
j) Save a fighter to a file or load a fighter from a file. Just click the button Save or Load. And choose a file to save or load. All the component in the MainPane will be saved.

The body parts in the MainPane or in a File is objects of MyComponent. Not any objects in a game framework. Developer could write the converter to convert a MyComponent to the objects they need.

For example, convert from MyComponent to LimbNode. It happens in FrameWorkLoader.java

```
MyComponent root = null;
for (MyComponent m : list)
        if (m.isRoot())
                root = m;
return buildBodyTree(root, null);
```

where list is an ArrayList of MyComponent. And you can write your own function: buildBodyTree(root,null).

Should the developer want to add more function of GUI. For example, add more Button. He should first subclass JButton and implement interface Update.

```
public class LoadButton extends JButton  implements Update{
```

instantiate in its container and pass the container as a parameter:

```
public LoadButton myLoadButton = new LoadButton(this);
```

in the subclass LoadButton, add its self in container and register itself in container.

```
AttributePane outer;
outer.register(this);
```

outer.add(this);

keep a reference of Controller simply by

```
private Controller myController = Controller.Instance();
```

Implement update() function. In this function, you should ask the data you need from the controller.

```
@Override
public void update() {
	// TODO Auto-generated method stub
}
```

Update others, tell the controller to update.

```
private  void updateOther() {
	myController.updateAttributPane();
	myController.updateFigherBuilder();
}
```

The developer could add their own function by subclass the interface State:

```
public interface State{
	abstract void create();
	abstract State getState();
	abstract void action();
}
```

For example If you want to add new reaction as mouse dragged. MouseDraggedSate implement State and keep a list of it self's subclass.

```
public class MouseDraggedState implements State {
	private HashSet<MouseDraggedState> allStates = new HashSet<MouseDraggedState>();
	MainPaneModel model;
	MouseEvent e;
	private static MouseDraggedState instance;

	public static MouseDraggedState Instance(MainPaneModel model, MouseEvent e) {
		if (instance == null)
		instance = new MouseDraggedState(model, e);
		instance.e = e;
		return instance;
	}

	MouseDraggedState(MainPaneModel model, MouseEvent e) {
		this.model = model;
		this.e = e;
	}

	public void create() {
		allStates.add(isDraggingState.Instance(model, e));
		allStates.add(isDragingSizeState.Instance(model, e));
		allStates.add(isDragingRectangleState.Instance(model, e));
	}

	public MouseDraggedState getState() {
		for (MouseDraggedState state : allStates) {
		if (state.getState() != null)
```

```
                return state.getState();
            }
            return new PseudoState(model, e);
        }

        public void action() {
        }

}
```

And the developer could override the getState() and action() in a subclass of MouseDraggedState

```
public class isDragingSizeState extends MouseDraggedState

        public isDragingSizeState getState() {
            if (model.isDragingSize())
            return instance;
            return null;
        }

        public void action() {
            model.dragSize(e.getPoint());
        };
```

Camera

The Camera is a huge part of what can be done with a fighter game. Depending on how the developer sets up the camera, you can have an array of different fighter games ranging from Street Fighter style (side to side scrolling) to Super Smash Brothers style (universal floating camera to capture all of the action) . The main Camera class is subclassed by a variety of different Camera subclasses and also SpecialCamera subclasses that allow the user to have custom camera actions that respond to events during the game (i.e. shake the screen when hit, or zoom into the boss on a finishing blow). Not only can the developer customize his own levels, he can choose to allow his players to make changes to the camera. The camera was designed to be as flexible as possible to allow easy access to changes during the game as well as addition of new camera subclasses that the developer might want to implement.

The Camera, as well as a CameraUtility class, are contained within each instance of CombatObject. The CameraUtility class allows the Camera type and properties to be changed on runtime or beforehand, allowing for any type of Camera to be associated with each CombatInstance. The CameraUtility class contains a map of keywords to Camera subclasses, which allows the developer to add his or her custom camera subclasses using code like:

```
CombatInstance.getCameraUtility().addCameraType("Floating", new FloatingCamera);
```

The developer also has the option of getting all the possible camera types, for example if he

wanted to display them to a player to choose:

```
ArrayList<Camera> cameraTypes = CombatInstance.getCameraUtility().getCameraTypes();
```

If the developer wanted to change the default Camera type for a certain level, he or she would use:

```
CombatInstance.getCameraUtility().changeCamera(“Scrolling”);
```

However, if the developer decides to allow the user to choose the camera type in an options screen, then the same code could be used in some kind of ActionListener:

```
actionPerformed(ActionEvent e) {
	CombatInstance.getCameraUtility().changeCamera(“Scrolling”);
}
```

The Camera class itself has default render() and update() methods that are used by Golden Tee’s game loop to properly render and update on each tick of the game. Since Camera can be subclassed, it is easy to create custom camera behavior by overriding the default render and update methods. For example, the Camera’s default update() method consists of the following code:

```
public void update(ArrayList<FighterBody> playerSprites, CameraBackground bg) {
	calculateNewCenter(playerSprites);
	calculateNewBounds(playerSprites);
	changeZoom(bg.getX());
	… // code to handle special camera stuff
 }
```

If we wanted to override the default behavior with a camera that follows around a sprite, we can override the default behavior in a subclass:

```
@Override
public void update(ArrayList<FighterBody> playerSprites, CameraBackground bg) {
	bg.setToCenter(this.getX(), this.getY(),
			this.getHeight(), this.getWidth());
	super.update(playerSprites, bg);
}
```

The same can be done with the render() method, the default which is shown below:

```
public void render (Graphics g1, CameraBackground bg) {
	Rectangle r = new Rectangle(5, 5, getWidth() - 10, getHeight() - 10);
	Graphics2D g = (Graphics2D) g1;
	g.setColor(Color.BLACK);
	g.draw(r);
	super.paintComponent(g1);
}
```

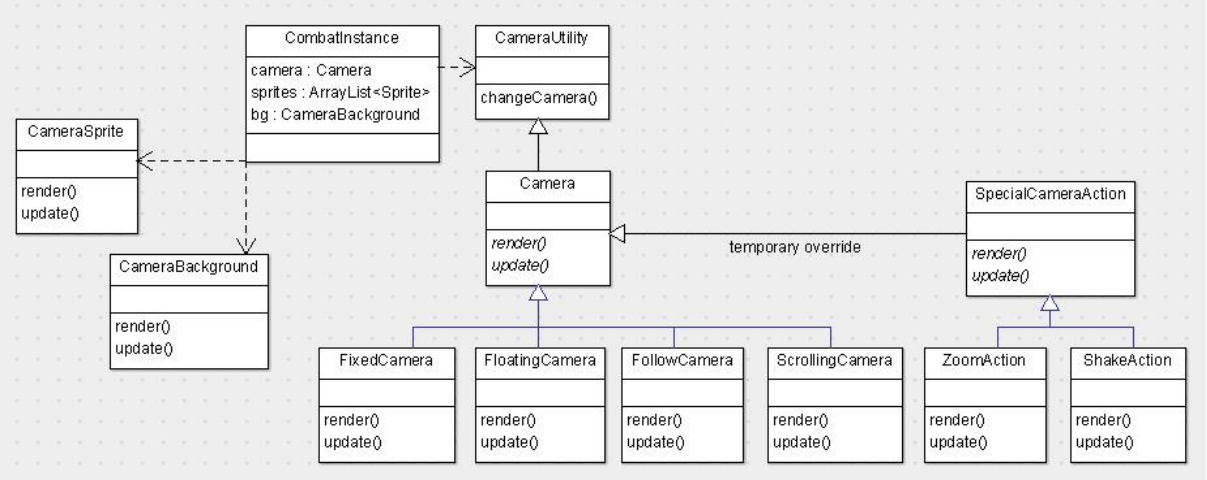
Then we can override this behavior to be able to scale the background in the subclass:

```
public void render(Graphics2D g, CameraBackground bg, Camera camera) {
	AffineTransform old = g.getTransform();
	AffineTransform tr2 = new AffineTransform(old);
	tr2.translate(camera.getX(), camera.getY());
	tr2.scale((double)Camera.CANVAS_WIDTH/camera.getWidth(),
	(double)Camera.CANVAS_HEIGHT/camera.getHeight());
	g.setTransform(tr2);
	super.render(g, bg);
	g.setTransform(old);
}
```

The developer will be able to add his or her own special camera effects that can react to collisions or events in the CombatInstance. The CollisionDetector, on each iteration of the update, will essentially check for collisions between sprites, and if there is a collision, the CollisionStatus for those sprites will be marked as true for the remainder of the update (before they are reset to false). That means we can check for collision events right after they update and temporarily override the update() and render() methods of the Camera. If the Camera mode is active on one of these special modes, then the state of the Camera will change from normal to *inSpecialMode* (governed by a boolean). As long as this mode is active, the *elapsedTime* will continue counting until it hits the duration of the special mode (given in the constructor), at which the mode reverts to the normal Camera behavior. Again, we can simply override the Camera methods with what we want the camera to do for the duration of the special effect:

```
public void update(ArrayList<FighterBody> playerSprites, CameraBackground bg, Camera
	camera, double duration) {
	Point oldPoint = camera.getCenter();
	Point newPoint = new Point((int)(camera.getX() * (1.3 * Math.sin(duration))),
	camera.getY());
	Rectangle oldBounds = camera.getBounds();
	Rectangle newBounds = new Rectangle((int)(oldBounds.x * (1.3 *
	Math.sin(duration))), oldBounds.y, oldBounds.width, oldBounds.height);
	camera.setCenter(newPoint);
	camera.setBounds(newBounds);
	super.update(playerSprites, bg);
}
```

Essentially, it is very easy for the developer to add new Camera types, implement these Camera types into his or her game, and also make it flexible for the user to customize his or her camera options. The types of custom cameras that can be implemented is up to the imagination of the developer; simple, horizontal scrolling cameras allow for a King of Fighters feel, whereas a more complex rendering will allow for 3d-esque gameplay.



Physics

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);
```

Also, the physics engine is allowed to developers to change some features:

- `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.

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. Every CollisionKind has some CollisionReactions in it.
- 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);
	…
}
```

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

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

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

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

They are the specific reactions we need these two sprites to do. Here are six common reactions.

- 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.

- 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.
- 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.
- Reaction Push: This class works for the case that one pushes the other and these two go together.
- 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.
- 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.
- The developers are allowed to write their reaction inheriting Reaction.

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

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