Hui Gao
CS108
Vooga

Fighting Game Camera: A Design Document

Introduction

The camera of a game is an aspect that is present in virtually all games (barring Zork) but is usually taken for granted. It should be implemented carefully in a game framework to allow for maximum flexibility. Not only are there multiple types of passive cameras, the developer may want custom camera actions for certain events in-game, such as a camera shake on a knockout punch. There are many ways of tackling this problem, some which are better than others.

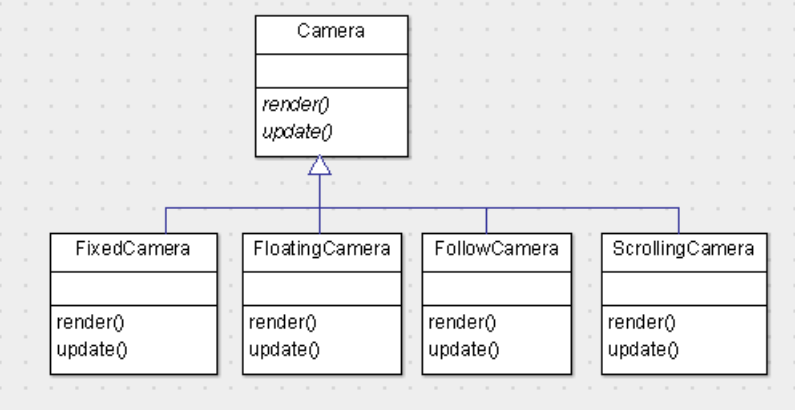
Our design for the camera should be able to handle both easy and hard use cases. One particularly easy use case is if the developer chooses the default camera for his fighting game, and does not allow the player to change the default camera, or have special camera effects that react to in-game collisions and events. For example, the default camera is the fixed camera, in which the background and platforms do not scale or translate in any way, and the sprites move in this static space. The developer decides to keep this default camera mode and develops his fighting game around it, and decides not to add any special camera effects. The user is also not given the option to change the camera mode. All in all this is a very boring route to take.

The hardest use case would be if the developer decides to change the default fixed camera for a certain fighter level. He changes it to use a horizontal scrolling camera and also gives the players the option to change the camera mode before the game actually starts. The player picks the floating camera mode and starts the game. After the game begins, there are all kinds of special camera effects that respond to the collisions and events of the actual gameplay. The player is punched, and the screen shakes a little bit. He then picks up a damage powerup, and the camera quickly zooms into the player and zooms back out. As the camera zooms in to emphasize the powerup, the player is hit again, causing the screen to shake and zoom at the same time.

Design Details

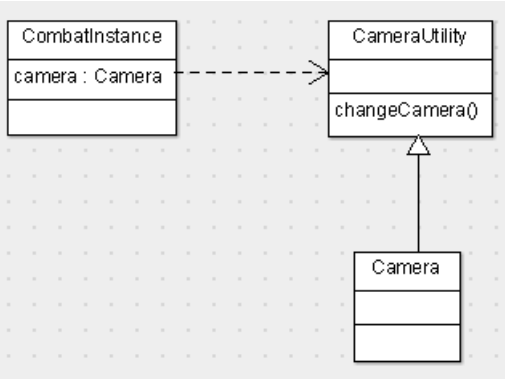
The overarching interesting problem that we will need to face in our fighting game is how the camera will work. There can be many orientations, movements, and configurations. For example, Street Fighter type games have a side to side scrolling camera that does not allow the characters to move off the screen at all (with a background that follows in the scrolling), whereas a game like Super Smash Brothers will balance the camera out to capture everyone on the screen as well as the terrain by zooming in and out, and panning. Since we want the developers to be able to pick which kind of camera they want to use in their game, as well as allow the users to have the option to change these settings right before a game or when they create a level, and also allow different combinations of panning, zooming, scrolling, etc., an elegant design will be more complicated than having a simple config file. Not only is there normal camera behavior, there may be special camera actions such as a shake when the player is hit, or a quick zoom in when a player picks up a powerup.

The first aspect we should consider is to allow the developer to add his or her own camera modes without violating the open-closed principle. Based on the rest of the design, this problem is simply solved by subclassing any new camera modes with the Camera superclass. Since all cameras will have the same basic underlying principles and methods, this will prevent code duplication by using the superclass methods, but will also allow unique camera behavior by letting the developer create his own camera render and update methods (which is the crux of the camera; after all collisions and events are resolved, the camera handles the rendering of all the sprites/background to fit its camera behavior). This simple inheritance structure works because of the way the rest of the design is set up, which is outlined beneath the UML representation:



The way we can have the developer choose the camera type must be considered as well. Each level that the developer chooses must have a default camera type, which can simply be declared by the developer. However, we also want to give the developer the option to let

the player choose their own camera of choice before playing a level or storyline. By having a CameraUtility class that utilizes a Strategy-method-like pattern, we create methods to change the camera mode that the developer can either define in his or her own game, or bind to certain key presses to allow the player to change the camera mode right before they play a level. This also allows the developer to easily include any custom camera modes he or she creates in the games that he or she develops. The UML diagram below shows this design:



The developer could use code such as

```
CombatInstance.getCameraUtility().changeCamera("Scrolling");
```

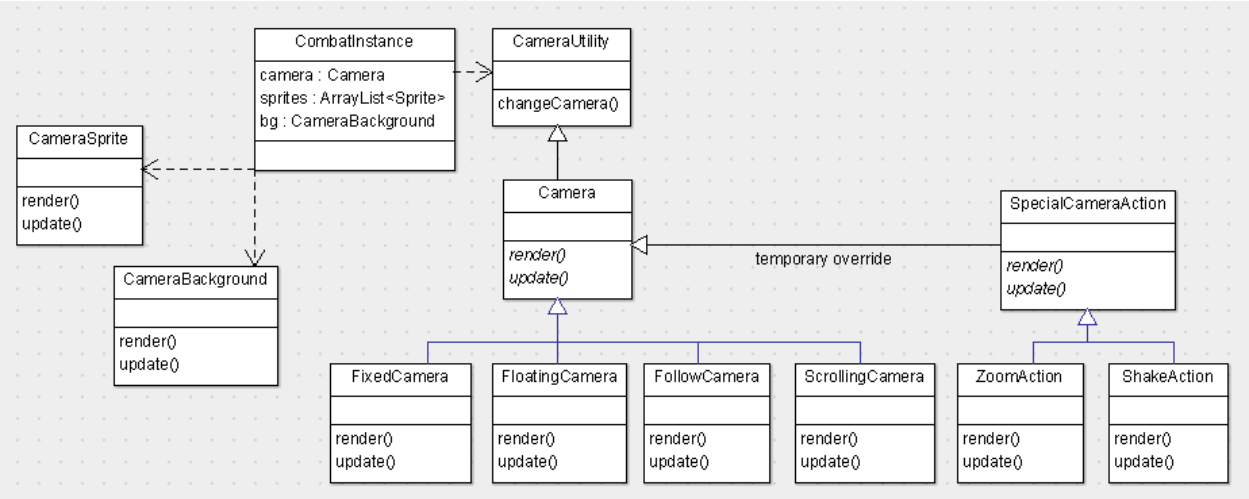
to set the camera of a specific level, and use

```
actionPerformed(ActionEvent e) {
    CombatInstance.getCameraUtility().changeCamera("Floating");
}
```

to let the player change the camera mode before starting a game.

The last major aspect of the camera that needs to be considered is the custom in-game camera effects that can happen. Since these effects react to certain collisions between sprites and other fighter sprites or non-player sprites, we need some kind of event listener that listens for specific kinds of output from the collision detection system. In the special camera effects class, we can use these listeners as well a map of different collision events to corresponding camera effects; whenever a collision event occurs that is in the map, the camera effect class's update and render methods will override the current camera mode's methods for a specific amount of time, which can be kept track of with timers in the system. The implementation of each custom camera effect is also relatively straightforward, and would simply require an algorithmic way of calculating the camera's center and bounds (for example, if we wanted a quick camera shake, we could use the center of the camera as normal, except adding on

something like a sin(time) term to the x axis movement for the specified time duration). The UML diagram shows the entire design of the camera system:



The CameraSprite and CameraBackground control the zoom of the backgrounds and sprites; after collisions have been resolved, the background and sprites are scaled and translated based on the camera type, and if a special camera action is happening, essentially by decorating the background and sprites in CombatInstance.