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.

One naïve way to implement the camera is to have a separate class for each special camera type. The developer could simply choose which one he or she wants for his game for various parts of the game. However, this would mean that the camera mode is stuck at runtime, which prevents potential user choice, and also does not allow for camera response to sprite interactions (knockout punch as mentioned, zooming in on a powerup when it spawns). This would also be harder for the developer to add his own camera modes if desired, since he would need to go into any of the classes that had a camera class and change them in the code as necessary. Essentially this would be a very straightforward but inflexible way to approach this problem.

My idea for the design is to have one camera superclass that implements basic camera functionality, and subclasses have the ability to override the default behavior such as scroll or zoom before calling the superclass's update and render methods. Each CombatInstance class will have an instance of the superclass Camera, whose subclass functionality can be determined at and during runtime based on properties such as level default and options. This allows the developer to create new subclasses of camera at will which have any number of scroll behavior, and have them available to be changed as he or she is making the game and also during the game itself for the player. Additionally, the camera will have some kind of event listening, which when a certain type of collision event is detected, a special camera function will override the default camera function for a small period of time; this will allow the developer to not only include special camera actions in the game triggered by any type of event, but is also easy for the developer to implement any custom camera actions and have them trigger in the game based on a mapping of collision event to special camera action. Essentially this design solves the problem that there are many types of cameras and many instances where a short alteration in camera behavior is necessary to react to a game event.

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 being able to have different combinations of camera techniques that will allow developers to distinguish a SSB-type camera from a SF- or KOF-type camera. There are 8 big types of camera types, fixed point, rotating, scrolling, movable, floating, tracking, pushable, and first person. Some of these such as first person will not be relevant but most of them will be. Currently, we can take advantage of inheritance to solve this problem. There is a Camera superclass which performs functions that all cameras would need, but individual subclasses can implement zooming or scrolling based on their desired effect. The current implementation also allows the game developer to create new custom camera modes, which can be easily added into the system without changing existing code. The way that the Camera system controls scrolling and zoom is that the Camera class keeps track of its center and bounds based on the position of each of the sprites in the CombatInstance, which control the scrolling of the camera as well as the zoom ratio. The zoom ratio is then used by a wrapper class of each sprite to scale and transform them on the canvas so it seems like they are bigger; this essentially creates the zoomed-in effect, where if the only 2 sprites on the map are close to each other, the camera will be focused on that part of the canvas in terms of the background, non-player sprites, and fighter sprites, and should stretch each out to fit the game screen. Basically all of the sprites change their position and states based on resolving collisions with other sprites and platforms, and then the camera updates to only render a specific portion of the game canvas.

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. We can do this by utilizing the framework’s Option utilities, where the developer will have the choice to add an option menu for the player to choose different camera modes, among other options. If we use the Strategy pattern, we can simply have one declaration of a Camera in the CombatInstance class (and any other types of game state that would require a camera) and then on runtime instantiate that Camera to be a specific subtype, based on either the level default camera, or the developer/player choice.

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).