Kieran Gallagher

S1313540

Computer Games (Software Development)

Code Explanation – GP1 CourseWork 2015

cSprite Class

This was one of the classes that was provided in the labs, it has been modified since then though to include my implementation of per pixel collision detection. It deals with generating, updating and rendering the sprites I used in the game. It is the parent class to cEnemyCar, cPlayerCar and cBkGround. Because this is a parent class, methods like update(), render() and renderCollisionBox() are specified with the keyword 'virtual' as they are likely to be overridden in the child classes to serve their specific purposes. Many of the functions actually remain untouched by me during development as I felt they were satisfactory for the purposes I needed them for such as setting up the textures as well as their dimensions or dealing with sprite translations along with sprite rotations and so on. I only added functionality towards the end of development when implementing per pixel collision.

My implementation of per pixel collision detection was aided by some psuedocode from an answer on gamedev.stackexchange.com(see References for more details.) The method PixelCollision() is used to determine whether or not 2 'solid' pixels have collided from the sprites passed in as parameters (thisSprite and otherSprite). It generates a world matrix for both sprites as well as the inverse world matrix that the otherSprite (the sprite that is being checked against for the collision), The texture sizes of the textures used by the sprites is also retrieved as that will be needed when iterating through the pixels for the collision. The reason for using world matrix is in order to get a point(pixel) in one sprites local space and convert it's location in relation to the other sprites local space. The function also checks to see if a pixel from one sprite falls out of bounds of the other sprite and if it returns false, the pixel is then checked to see if it has collided with. After checking to see if the both pixels in the collision are 'solid'(not clear), the function will then return true and if not, false later in the method which in turns determines whether or not the 'solid' pixels in a texture have collided. It is worth noting that this method is set up to detect the collisions of sprites that are centered properly(half width and half height, in the middle.) in hindsight this could have also been made a virtual method, like render(), as other types of sprites may require slightly different implementations of this.

Another method I have added to the cSprite class was GetWorldMatrix() which exists purely to serve the needs of per pixel collision detection functionality. It gets the world matrix of a sprite by multiplying other matrices that are related to the sprite together such as translation, scale and rotation as well as the origin. It then returns the matrix.

cPlayerCar Class

This class, which is a child to the cSprite class, deals with the player controlled car featured in the main game mode. the sprite has acess to all of the functions that can be found in cSprite as well as implementing some of it's own. The update function deals with many things that make the player controlled car work as intended.

In the game, the player can move the car left or right by pressing left or right in the d-pad or use the left or right keys on the keyboard, to do this the update method will check the input manager for any input that occurs from the keyboard or the game pad and if they return true during the update, the sprite will move according to the direction pushed by the player, this movement will be multiplied by the elapsed time since the previous update in order to keep the translation of the car sprite consistent over time. The update function also keeps tack of the health attribute, if it is less than 1, the player car is deactivated an appropriate sound effect is also played. the update method also takes note of the vibration strength in the game pad's motors and will reset it to zero after a short period of time if it is not already zero. The update method also prevents the player from progressing past certain boundaries along the x-axis by locking off particular inputs.

The render() method works in very much the same way as it does in cSprite with one key difference in that it is rendered with the origin being in the middle of the sprite allowing for rotation of sprite around the middle instead of the top left hand corner.

Because the players health can be decreased by other classes, it made sense to have a dedicated public function that dealt with that, it simply reduces the players health variable by one and plays an appropriate sound effect.

Other methods for cPlayer car act as appropriate setters and getters for the appropriate attributes like the health, XBoundaries and speedX variables.

cEnemyCar Class

This child of cSprite is simple when compared to cPlayerCar as all it has to do it move downwards and that is it. It renders the sprite much the same way as the player car (see cPlayerCar Class - code explanation). It updates it's position along the y-axis each frame and calculates how much it should move by multiplying it's speed attribute with the elapsed time since last update.

The class also contains a function that returns true if the sprite position has 'left the screen' via the bottom of it. This was useful when determining when it is safe to delete the car object in another class as it is off screen and out of view of the user.

cBkGround Class

This code supplies the game with both static and scrolling visual elements to the game, it used for UI elements and for the road tiles that the player's car is driving on. To give the illusion that the player's car is moving, the road tiles scroll underneath the player car sprite. This was done by simply manipulating the texture coordinate data(essentially offsetting the texture) over time. The render() method is virtually unchanged from the one use in the parent class, cSpite, as the uses for this class are limited to static visual elements that don't rotate.

cEnemySpawner Class

This class is what keeps track of the enemy cars it spawns as well as checking for collisions between the enemy cars and the player car. It's constructor initialises the the classes variables such as the timers it uses to keep track of when it should spawn cars or increase the difficulty. it also attaches parameters like the player car to it along with the boundaries the player car(see cPlayerCar Class) uses among other things.

SpawnEnemy() instantiates a new enemy, set its speed to that of the spawner's speedY attribute and adds it to the vector of enemy cars the class iterates through in the update() function. In the update() function the class iterates through vector of cars calling their update and render functions while also checking if the  player car is colliding with any of them, if so then the car is erased from the vector of car pointers, the player cars ReducePlayerHealth() function is also called. The update function will call the enemy car in the beginning of the vector to see if it has left the screen, if so, the enemy car from the vector. A Random position that takes into account the previous spawn position of the previous enemy car and the x-boundaries that are set in the level is also set for the new enemy car.

The difficulty timers and spawn timers are also incremented on each time the update function is called, if certain limits are reached, variable within the class are modified which affects its behaviour through out the main game such as the speed increase of enemies being spawned of the decrease in the intervals in between EnemySpawn() calls. This class could be thought of the AI that the player is playing against during the runtime of the game.

GameScene Class

I needed a way to elegantly manage screens, or scenes as I like to call them, during runtime. I also want to reduce clutter in the main.cpp file so that is the purpose of the GameScene class. It's constructor pretty much loads everything the game needs to function while also instantiating/initialsing the game objects that will be used in the scene. Fonts and textures are created, sounds are loaded and so on. in the update method the class manages the scenes by checking an enum variable of type scenes (i.e. "if (scenes = = Main\_Menu){//Update.Draw main menu}”.)

I felt that by doing things this way, I could easily keep track what scene is to be drawn at any one time. Once it has been decided what scene is supposed to be drawn, the class checks if the scene has been initialised, if not then it initialises it, in the case of the main game mode scene this could include setting the players position and health while also instantiating the enemy's spawner. after the initialisation process has occurred, the scene elements would be rendered and update as normally until a condition is met that will switch the scene(like the player dying), this is done by altering the enum mentioned previously.

Xcontroller Class & cInputMgr

the Xcontroller class is a class that contains code that was taken from somewhere else on the internet(see References). It deals with player input on the game pad with functions that can retrieve the current state of the game pad which can be used to interpret what the player is doing with it, a class that checks if the controller is connected and a function that changes the vibration strength of the motors with in the game pad. I needed to install the xinput api in order to utilise these functions correctly.

The cInputMgr is a heavily modified class that was taken from the labs with mouse functionality removed. It has a buffer that stores keys pressed and keys that are held down, there are other functions that can manipulate the state of keys in the buffers. I added a function that directly sets a specific key down state for a key to 'up'. I have added functionality to the class that allows it to store a collection of  Xcontrollers that makes it easier for other classes to access the game pads connected with the AddController()function and GetController() functionality.

Other Classes

Most of the classes I discuss here were taken from the labs and feature  very little modification from me.

The cSound and cSoundMgr classes deal with loading, storing and playing the sounds that I featured in the game, they are exactly the same as the ones feature in lab 8.

The cFontMgr which stores the collection of fonts to be used in the game remains unchanged from lab 8. The cFont class prints text out onto the screen. This actually has been modified slightly to fix the issue where text would display upside down, it was fixed by simply inverting the y value of the scaling(which in turn caused me to have to set the y value of the position where the text would display to a negative value if I wanted the text to display on the screen.) I have also added a function that makes it easier to edit the colour of a particular font.

The windowsOGL this deals with the creation and management of the open gl window as well as destroying it when the program closes. cWndManager has recieved a slight alteration that calls the keydown function of the input manager when certain keys are pressed and the keyup function when the keys are released.

cTexture class includes some additions such as returning a value that can be interpreted as whether or not a specific pixel on a texture is clear or now. Apart from that it is mostly unchanged.

References

Here is a list of all the articles, tutorials and books I have read and relied on throughout the module when working on the coursework:

* [Per Pixel Collision psuedocode/tutorial](http://gamedev.stackexchange.com/questions/23603/how-to-handle-pixel-perfect-collision-detection-with-rotation)
* [How to query performance counter](http://stackoverflow.com/questions/1739259/how-to-use-queryperformancecounter)
* [Lynda.com c++ essential course](http://www.lynda.com/C-tutorials/Arrays-strings/182674/366532-4.html?autoplay=true)
* [Game Coding Complete Book by Mike McShaffry](http://www.amazon.co.uk/Game-Coding-Complete-Mike-McShaffry/dp/1133776574/ref=sr_1_5?ie=UTF8&qid=1429016067&sr=8-5&keywords=games+programming)
* [How to convert int to LPCSTR/String](http://stackoverflow.com/questions/17639202/how-to-convert-type-int-to-type-lpcstr-in-win32-c)
* [This helped when I was figuring out how to spawn enemies](http://www.cplusplus.com/forum/beginner/69766/)
* [This helped give me an idea how to switch scenes](http://stackoverflow.com/questions/20677023/change-scenes-in-game-opengl)
* [XInput controller code](http://fisnikhasani.com/xbox-360-controller-input-with-c-using-xinput-h-api/)

Class Diagram and Code map can be found in Visual Studio Project.