Alfred Lucero

CS 32 Project 3 Report

ID: 604251044

February 26, 2014

1. ACTOR.H // ACTOR.CPP

For the overall object-oriented design of my Bug Blast project, I had a base class Actor derived from Graph Object from which other derived classes would follow such as Exit, Bug Spray, Bug Sprayer, Player, Brick, Zumi, and Goodie. From the Brick class, two more classes are derived named Permanent Brick and Destroyable Brick; from the Zumi class, two classes are derived named Complex Zumi and Simple Zumi; and from the Goodie class, three classes are derived named Extra Life Goodie, Increase Simultaneous Sprayer Goodie, and Walk Through Walls Goodie. The Actor base class has a constructor that takes in the starting x and y coordinates, the proper image ID, and the corresponding Student World that the actor currently resides in. It has a virtual destructor that does “nothing” (~Actor(){}), and I created a doSomething() pure virtual function because all derived actors will perform a certain action, but an Actor itself will never be instantiated. I have public helper functions such as setDead() and isDead() because every derived Actor will have an alive or dead status to be eventually removed from the game during every turn of Student World’s move() function. Also, Actor contains a getWorld() function that returns a pointer to the Student World the derived Actor resides in, allowing derived classes to use public functions within the Student World.

Within the Zumi class derived from Actor, I had two pure virtual functions of doSomething() and damage() because a Simple Zumi and a Complex Zumi varies in the manner by which each dies or moves, and a Zumi base class will never be instantiated. There is also a general attemptMove() function that checks if a derived Zumi can move in a certain random direction, and there are public helper functions that handle the time intervals and randomly generated directions by which a derived Zumi moves such as increaseTicks(), resetTicks(), getTicksPerMove(), getTicks(), setCurrentDirection(), and getCurrentDirection(). The Simple Zumi’s implementation of doSomething() only attempts to move if its current ticks equal the time interval and then resets to 0 to always have a specific time interval between moves (also accounts for whether or not the Simple Zumi can actually move in its currentDirection). Once a Simple Zumi dies, it receives “damage()” in which the Simple Zumi is set to dead and may drop a Goodie depending on the level’s probabilities. In addition, the Complex Zumi has its own versions of doSomething() and damage(). In its doSomething(), the Complex Zumi carries out a queue-based (breadth-first) search within its smellDistance, utilizing structs of Coords within the queue to figure out if a path exists. There is also 15 x 15 integer array that keeps track of the level to handle explored Coords as reference. After checking for a path, the array is reset to contain all 0s to signify unexplored (explored is 1). If there is a path to the Player, the queue based search will keep track of the best possible first move to make towards the Player; otherwise if the Complex Zumi is in range but is blocked by Bricks, it will behave as if it were a Simple Zumi. Its damage() function differs from the Simple Zumi’s implementation by simply increasing the game score by a higher value.

In the Brick class, its virtual doSomething() function is shared by the Permanent and Destroyable Brick, and it does “nothing” (doSomething(){}), and it has a pure virtual isPermanent() function so the user cannot create an instance of a Brick. The Permanent Brick and Destroyable Brick differ by their implementations of isPermanent(), which either return true or false.

In the Goodie base class, it has a pure virtual doSomething() because each derived Goodie will have a varying implementation of it, and no Goodie itself will be instantiated. The Goodie base class has public helper functions that manage the Goodie’s lifetime in ticks such as decLifetime() and getLifetime(). The WalkThroughWallsGoodie, ExtraLifeGoodie, and IncreaseSimultaneousSprayer Goodie vary only in their implementations of doSomething(). The ExtraLifeGoodie ‘s doSomething() checks if the Player is on it before setting itself to dead and increasing the Player’s overall life in the game; otherwise, decrement its lifetime until it hits 0 when it is set to dead for removal. The WalkThroughWallsGoodie carries out a similar process except it grants the Player the ability to walk through walls for a short period of time, and the IncreaseSimultaneousSprayerGoodie increases the number of BugSprayers a Player can drop at a time for a certain number of ticks.

Within the BugSprayer class, its doSomething() function decrements its lifetime as long as it is not dead, but once it dies, it adds BugSprays in a cross formation from where it was originally dropped by the Player. It also has a public helper function setChainReaction() that sets its lifetime to 0 if a BugSpray pops on a BugSprayer before its lifetime runs out, thus expediting its dead process to unleash more BugSprays. Moreover, BugSpray’s doSomething() function also decrements its lifetime while it is alive, and it destroys possible targets that it may be on such as a Destroyable Brick, a Player, a Zumi, or a BugSprayer.

In the Player class, there is a doSomething() function that receives key input from the user and then checks to see if the action can be carried out, whether it is a move in a certain direction or an attempt to drop a BugSprayer in a certain spot. If the Player is on the same spot as a Zumi or on the same spot as a Destroyable Brick when it does not have the special effect of a WalkThroughWallsGoodie, the Player will be set to dead and lose a life. The Player also has two functions that can activate the special effects of the WalkThroughWallsGoodie and IncreaseSimultaneousSprayerGoodie: activateWalkThroughWalls() and increaseSimultaneousSprayers(). Other public helper functions aid in setting its status or turning off the effects of a Goodie after the effects have cooled down such as getWalkThroughStatus(), getBoostedSprayerStatus(), setWalkThroughStatus(), setBoostedSprayerStatus(), turnOffWalkThroughStatus(), and turnOffBoostedSprayerStatus(); these will usually be called by the Student World object to keep track of the Goodie effects and their remaining ticks.

Furthermore, within the Exit class, it has a doSomething() function that checks if it is exposed and a Player is on it to signify to the Student World object that the level is completed to move onto the next one. The other public helper function exposeExit() sets its image to visible after all the Zumis on the level have been destroyed; it is called only once.

STUDENTWORLD.H // STUDENTWORLD.CPP

The StudentWorld class derived from GameWorld utilized functions such as those that aided in loading the level, handling the location checks among actors, reacting to completion, dynamically allocating new actors in the level, controlling Spray/Sprayer related tasks, managing Goodie effects, updating the display text, and removing the dead game objects. It uses a vector of Actor pointers to keep track of all the actors in the game, and it uses a separate pointer to specifically point to the Player in the game. In addition, there is a Level pointer that is dynamically initialized upon loading a level and initializing all the pertinent data members for the StudentWorld object to keep track of during the game.

First of all, I followed the general recommended designs in implementing the public init(), move(), cleanup(), and destructor functions. For the init() function, the StudentWorld object would attempt to load a file, returning -1 if it failed to load successfully, but if its level is greater than 0 when the loading error occurs, that means that the Player beat the game. Otherwise, the init() function would continue the game normally.

For the move() function, it calls upon a function that updates the upper display text that shows the current level, bonus, score, and number of lives; this function makes extensive use of #include <sstream> and #include <iomanip> to format the display text easily as recommended by the spec. Then, the move() function iterates through the vector of Actors, calling upon the Player and Actors in the map to doSomething() before checking if the Player is dead or the if the level is completed. Next, after all the Actors attempted to doSomething(), all dead objects are removed by the removeDeadGameObjects() function that iterates through the vector, deleting and erasing the Actors that are dead. The move() function later checks for special Goodie effects and decrements the number of ticks accordingly, decrements the bonus, checks if all Zumis are killed in the level to expose the Exit, and checks once again if the level is completed; otherwise, it continues on while the Player is still alive and has not died or completed the game.

For the cleanup() function and ~StudentWorld(), both would remove the remaining dynamically allocated objects by iterating through the vector from the end while the vector was not empty. It deletes the last item in the vector before popping off the last item in the vector. The process repeats until there are no more dynamically allocated actors, and then the dynamically allocated Player and Level are deleted to finish the cleanup.

In addition, for all public helper functions that handled location checks such as getAnActorAtTheProposedLocation(), playerCanMoveTo(), zumiCanMoveTo(), isOnPlayer(), isOnZumiOrDestroyableBrick(), canDropSprayer(), and canDropGoodie(), a common technique was used that involved dynamic casting and iterating through the vector of Actors to check for locations. If the dynamic casting of a certain Actor type led to a null assignment, this means that either there is no Actor at that specified location or there is a different type of Actor at that location than the one that was attempting to be dynamically casted from.

In order to dynamically allocate new Actors such as Goodie types, BugSprayers, or BugSprays, I created public functions that would push back a dynamically allocated Actor of the specified type onto the vector of Actor pointers. Also, to handle the Level, Exit, Goodie, BugSpray, and BugSprayer effects, I implemented public helper functions that can be called within other Actor type classes by using getWorld()->someFunction(). This technique became the basis by which Actors can interact with other Actors or retain knowledge about the level.

2.

After much deliberation and countless hours of hard work during the night, I feel that I completed the BugBlast game with the proper functionality and without any perceivable bugs.

Pivotal Fixes to Ensure Proper Functionality:

At first I was unable to fix a bug that wouldn’t allow me to inform the Player that it has stepped on a Goodie to receive its effects. I strenuously searched for hours for common resolutions to the problem while wondering why my Player was not receiving the necessary information of ticks from the level for the effect duration. It turned out I accidentally implemented a function with a slight spelling error and called a function with the proper spelling. The compiler did not catch the error or pinpointed any problems because it saw the call to the function of correct spelling as an entirely different entity, leading the program to compile but with undefined behavior. In addition, I initially could not figure out why my newly dynamically allocated objects such as BugSprays or BugSprayers would lead my program to crash. I later realized I was calling getWorld() without actually initializing the StudentWorld those objects were in within their constructors. As a result, I added another parameter that would take a pointer to the StudentWorld object the Actor was in within each constructor of the Actors (I initially only set Player’s world, none of the other objects).

3.

While I was implementing how to load a level and initialize its content, I realized that it was unspecified as to how a Player would move onto the next level automatically, retrieving the next successive file appropriately after completing and finishing a level. I cleverly decided and assumed that we had to use stringstream and input/output manipulation. Similar to updating the display text that would provide a formatted string of the file, for example, “level00.dat”, I formatted the string to load “level” + leading ‘0’s if any + getLevel() + “.dat” every time a level is about to be loaded. In addition, before seeing the general object-oriented design displayed by Professor Smallberg, I settled on my own arrangement/inheritance pattern of base classes and derived classes as well as public helper functions that would reside in each of Actor’s derived classes or StudentWorld’s class and private helper functions within StudentWorld’s class that would handle level initialization and removal of dead game objects. Furthermore, I decided to focus the interactions among Actors and the level through calls to getWorld()->somePublicFunctionWithinStudentWorld() that would dynamically allocate new objects during run-time such as the Goodies, BugSprayers, or BugSprays, find locations of certain Actors, or manage Actor lifetimes/effects.

4.

For the Actor base class, I verified that any class derived from Actor can use its public member functions and that its constructor worked correctly in conjunction with its base class of GraphObject. I would ensure that all Actors were correctly placed on the level’s map and set to visible at the proper coordinates with the exception of the Exit class. I tested by comparing the “level00.dat” file’s level layout with the image displayed during the debugging process. If at first the objects were not displaying, I would ensure that the image IDs matched properly and the StudentWorld’s load() function would correctly dynamically allocate the objects at specified locations with pointers to the StudentWorld object they were in. I would keep checking until all Actors were visible on the map, though without allowing any doSomething() functionality yet.

Later, in order to verify the Zumi class, I would test to see if an actual Zumi would be displayed on the map, whether it is a Complex Zumi or a Simple Zumi, with the proper imageID. In addition, I would test to see if the public helper functions relating to the ticks per move were working by checking to see if the Zumi did not continuously move randomly in quick succession but in the interval given by the Level’s option values. In addition, I analyzed the attemptMove() function and viewed whether or not the Zumi would occasionally stay put if the direction was pointing to a Brick or a BugSpray and would keep on choosing a new direction away from those objects the Zumi cannot move to.

In relation the Zumi class, I checked whether or not the Simple Zumi can move at all in the map and in a random direction usually away from Actors such as BugSprayers or Bricks. Then, I tested whether or not a Simple Zumi will die to a BugSpray; for this, I simply implemented the code and played through the game manually to ensure the correct behavior occurs with the proper sounds being projected as the Simple Zumi dies. Later, I observed whether or not the Goodie drop rates were working according to plan by looking into the “level00.dat” file and seeing whether or not Goodies such as a WalkThroughWallsGoodie should drop when a Simple Zumi dies.

Also, I inspected the Complex Zumi class in a similar fashion by checking to see whether or not the Complex Zumi can move at all or show up in the map. I then observed whether or not it should act like a Simple Zumi if there was no possible path to the Player by simply having my Player stand still in a location where the Complex Zumi in question would be blocked off without a possible route to the Player. Next, I would break Destroyable Bricks around the Player to allow a route for the Complex Zumi to come to the Player, and once again, I would leave my Player to stand still and see if the Complex Zumi would go towards the Player and eventually killing the Player. Furthermore, I would check to see if the Complex Zumi would die to a BugSpray or if the Complex Zumi would drop a Goodie when it is killed and removed from the level.

For the Brick, Permanent Brick, and Destroyable Brick classes, I would verify whether or not the Actors would display properly with distinguishable images for Permanent Bricks and Destroyable Bricks. Then, I would observe whether or not the Bricks would do “nothing” and stay constant as long as BugSprays were not affecting them. I made sure to notice that only Destroyable Bricks were able to be destroyed by BugSprays and removed from the level, and I analyzed whether or not the Player, once a WalkThroughWallsGoodie effect is on, can walk through only Destroyable Bricks and eventually die to a Destroyable Brick if the Player remains standing on the Destroyable Brick and the Goodie effect runs out.

For the Goodie class, I would check to see if its public helper functions concerning the lifetime of a Goodie would work appropriately and whether or not the actual image of a derived Goodie class (ExtraLifeGoodie, IncreaseSimultaneousSprayerGoodie, WalkThroughWallsGoodie) would appear and be allocated after a Zumi dies. I tested by not letting the Player pick up the Goodie and wait for its lifetime to expire to run out, and later, I observed whether or not after a Player steps on a Goodie that the effect would be granted appropriately to the Player. For example, I tested to see that the ExtraLifeGoodie worked correctly by checking that the number of lives was incremented on the upper display text. I also verified that the WalkThroughWallsGoodie enabled the Player to walk through walls for a brief period of time and kept on running through Destroyable Bricks until it can no longer pass through them, confirming that the effects can run out accordingly. Finally, I did a similar case for the IncreaseSimultaneousSprayerGoodie by checking to see whether or not the Player can drop more BugSprayers and by seeing if the effect will eventually run out and return back to the old maximum for the number of active BugSprayers out at a time.

For the Player class, I ensured that first the Player would spawn at the correct starting point and that the Player can move anywhere around the level with correct key inputs. After implementing restrictions, I tested to see that the Player cannot move to spaces where there are Bricks unless the Player has the WalkThroughWallsGoodie effect for a brief period of time. Moreover, I observed whether or not the Player can drop more BugSprayers with the IncreaseSimultaneousSprayerGoodie effect for a short interval as well and verified that the number of BugSprayers would be reduced after the effect was gone. Finally, I checked to see that the Player can die by moving onto a Zumi or onto a BugSpray by manually testing the game through the scenarios.

For the Exit class, I would first verify that the proper imageID would show up if I set its visibility to true. Then, I set its visibility to false and ran the game to make sure that the Exit does not show up. After playing the game and killing all the Zumi on the map, I would observe whether or not the Exit appears to see if it is working correctly. Finally, I would move my Player onto the Exit and make sure that the level is finished and moves onto the next one to play the subsequent level.

In order to test the StudentWorld class, I would gradually check certain aspects of the StudentWorld by first checking to see if I can get its init() and load() functionality down so that the level can be loaded onto the screen. I would verify that all parts are dynamically allocated and accounted for within the vectors, and I would run checks in my head to validate the iterations through the vectors to make sure that no elements out of bounds are being accessed, an act which would lead to undefined behavior, compiler errors, or crashes. Also, for the public helper functions that dynamically allocate run-time Actors, I would play the game and make sure that Goodies, BugSprayers, and BugSprays would appear on the level, and I would observe that they properly go away after being stepped on by a catalyzing Actor or after a certain lifetime runs out. I would also check that all data members are initialized at some point, and I would later check to see that the upper display text is refreshed continuously with the proper scores, lives, levels, and bonuses. In addition, I would verify the move() function by evaluating during debugging simulation that all the Actors, if possible to move such as a Zumi, were moving the way they were supposed to. Other aspects that I tested included failures to load levels, Player deaths in a game, level advancement, and overall experiences in Bug Blast. Often, I would view the example Bug Blast game as a reference and compare it to how my own version of the game played throughout the whole process of implementing and testing certain scenarios while consistently perusing back and forth between the code and the lengthy specifications for Project 3.