Aditya Pimplaskar

UID: 804900470

CS32 WINTER 2019 – Project 3

1. **High level descriptions of what my functions do:**

**STUDENT WORLD**

|  |  |
| --- | --- |
| **FUNCTION** | **WHAT IT DOES** |
| StudentWorld(std::stringAssetDirectory) | Constructor for the StudentWorld, should construct based on where we store the auxiliary asset files needed to build the project |
| ~StudentWorld() | Since StudentWorld is derived from GameWorld, we want to ensure proper destruction of our objects |
| init | Init was called at the start of every level to initialize the level if it was available. If incorrectly formatted, it returned the appropriate error message. If there were no more levels, it alerted the player that the game is over. It called buildLevel |
| buildLevel(int level, Level::LoadResult &res) | This was the function in which I created the objects that would go into the game based on parsing the level files. I pass in a variable by reference to store the result as it is this result that will tell init whether the game builds, if there is an error, or if the game is complete. |
| Move() | Move is called on every actor and calls their doSomething function leading to every actor doing something that they should do during that tick. |
| cleanup() | Loops over the list of pointers that stores our actors to prevent memory leaks. |
| clearDeadItems() | Similar to clean up, but will delete memory allocated for items that have died during a level. |
| addActor() | This was a simpler way to push a new actor onto the list of actors that we had. |
| recordCitizenGone() | This was a way to decrement the counter for number of citizens left (we want to keep track of number left because the player can only exit if all the citizens have exited or died). |
| activateOnAppropriateActors(Actor \*a) | This function was important because it delegated activity of each class back to the actor.cpp file. If an actor overlapped with the actor that was passed in, the activating object involved would perform its specific function (specificity came from virtual functions in actor class). |
| isAgentMovementBlockedAt(double x, double y, Actor\* ptr) | This function handled checking for collisions between actors by checking all the corners of the sprite boxes. Looped over elements of m\_actors and checked if they blocked agent movement and were in the specified location. Returned a Boolean accordingly. The ptr was passed in to ensure that elements would not block themselves (a check was conducted when looping over the list of actors). |
| isFlameBlockedAt(double x, double y, Actor\* ptr) | This function was very similar to checking if agent movement was blocked. The discrepancy was in the Boolean that it checked as this checked if an object blocked flames. |
| isZombieVomitTriggerAt(double x, double y, Actor\* ptr) | This function as similar to checking if agent movement or flame movement was blocked. It just involved looping over the list of actors and scanning x and y coordinates accordingly for humans. |
| locateClosestNearbyVomitTrigger(double x, double y, double otherX, double& otherY) | This function was used mainly in smartZombie as a way to check distance from a human that could potentially be poisoned. It made use of the Euclidian distance calculator and found a minimum across all humans in m\_actors. The variables passed by reference would mark target locations for these smart Zombies. It is supposed to return a Boolean but should not return false ever (because so long as Penelope is alive there is a vomit trigger) |
| locateNearestCitizenTrigger(double x, double y, double& otherX, double& otherY, double& distance, bool& isThreat) | This function was used on the citizen class as a way to determine whether or not citizens were close enough to a trigger (Penelope or zombie) to warrant movement. isThreat allowed us to distinguish between Penelope and zombies and allowed for the division of functionality in citizen’s doSomething depending on which trigger was closer by. It operated similarly to the previous function. It is supposed to return a Boolean but should not return false ever (because so long as Penelope is alive there is a citizen trigger) |
| locateNearestCitizenThreat(double x, double y, double& otherX, double& otherY) | This function works similarly to locateNearestCitizenTrigger but is solely for zombies. It will return false If no zombies are left, and true if there are zombies left (in which case the variables passed by reference will hold the coordinates). |
| Getters: getCitizensLeft(), getPlayerLife(), getPlayer() | Accessing the private members to view them for comparisons |
| Setters: playerDied(), playerNowAlive(), finishLevel(), | To manipulate private members according to the events that transpire in the game. |
| writeToSS() | Kept track of the stringstream. Used getters from this and from the actor class to update scores every tick. Called at the end of move. |
| Euclidian(Actor\* a1, Actor\* a2) and euclidian2(double x1, double y1, double x2, double y2) | Two different implementations of the Euclidian distance calculator that cater to different types of input. |
| doesOverlap(Actor\* a1, Actor\* a2) | Calls the Euclidian distance calculator and determines if that Euclidian distance is within the threshold that we have set to dictate that something overlaps/does not. |
| levelJumpForTest(int level) | A function that was purely used for testing the game. It allowed me to jump around between levels without having to play the game all the way to a particular level to test functionality on that level. |

**ACTOR**

|  |  |
| --- | --- |
| **FUNCTION** | **WHAT IT DOES/WHY** |
| Constructors | Every class had a constructor that ensured that it constructed itself as well as its base class appropriately. |
| Destructors | Base class had virtual destructor to ensure appropriate destruction of derived objects |
| doSomething() | The doSomething() function was the crux of all actions that any actor did. This function was used across all actors, and in turn was virtual in the actor class (though some implementations did not do anything like wall’s). doSomething() for activating objects simply called the activateOnAppropriateActors function from StudentWord (which called activateIfAppropriate). This was just to check if overlap occurred with activating objects, which would lead to some response. For agents, since they had to move around, doSomething() handled object movement, life status, poison status, etc. For Penelope, doSomething() handled input keystrokes and corresponding actions as well as poison status, life status, etc. |
| Getters: getWorld() | Returns a pointer to studentWorld. This was critical in calling StudentWorld functions like increaseScore, playSound, checking for blocking movement, and creating new objects (because they needed a pointer back to the world). |
| Booleans to check nature of object:  blocksMovement(), blocksFlame(), triggersOnlyActiveLandmines(), threatensCitizen(), triggersCitizen() | We used these Booleans as ways to classify our objects and determine what they were. BlocksMovement() was walls, and all agents. blocksFlame() was only walls. triggersOnlyActiveLandmines() was all agents. threatensCitizens() was all zombies. triggersCitizens() was all zombies and Penelope. |
| activateIfAppropriate(Actor\* a) | This function was the crux of the activating objects subclass. It was called by activateOnAppropriateObjects from the StudentWorld. Ultimately, this function was virtual as the functionality varied depending on which activating object was used (if it was a goodie, it would get picked up; if it were a landmine, it would explode; if it were a pit/flame it would kill the overlapping object, etc.). The variety in its functionality made it a clear candidate to make it virtual as it could diversify its effects across different object types. |
| useExitIfAppropriate() | Humans (Penelope and citizens) had the shared functionality of using exits. Instead of checking for overlap with the exit in both Penelope and citizen classes, we could do it once, simply determine whether or not the overlapping character can exit, and call the appropriate version of this function. This allowed for a general implementation whose function would be distinguished for the Penelope and citizen. |
| pickUpGoodieIfAppropriate(Goodie\* g) | Penelope was the only one who could pickUpGoodies, but delegating this functionality to the Goodie class as opposed to Penelope’s class allowed to avoid using keys (or some kind of tag) for each type of goodie. Instead, the goodie that it is called on will be the one whose behavior is called. This allows for a more general implementation. This function would go on to call the appropriate goodie’s pickUp(Penelope\* p) function. |
| dieByFallOrBurnIfAppropriate() | If an actor that could be burned/could fall down a pit and die overlapped with a flame or a pit, this function was called. This was called on goodies as well as all agents. This allowed me to generalize the functionality of pits and flames. |
| beVomitedOnIfAppropriate() | If any actor could be vomited on (all human objects), this function was called within each Zombie’s doSomething() to shoot vomit at the human (provided the human object was adjacent to the zombie). This function allowed me to generalize vomit triggers to humans, not to specific classes. |
| **LANDMINE:** getSafetyTicks(), getActive() | These functions allowed me to check whether or not the landmine was active (i.e. it had been 30 ticks since its conception). getActive() was a checker that allowed to ensure that the landmine was active so that if someone stepped on it, it would only explode WHEN ACTIVE. getSafetyTicks() simply kept track of the number of ticks. |
| **LANDMINE:** setActive() | Activated the landmine after 30 ticks so that it could explode. |
| **ALL GOODIES:** pickup(Penelope\* p) | Each goodie had its own version of this function. This function was virtual as picking up different goodies would increment different goodie counters in the Penelope object. If this functionality were delegated to Penelope, she would need to track which type of goodie was getting picked up. Since I delegate this functionality to each goodie type, I was able to avoid using explicit keys that specified the goodie. Instead, we were able to use the virtual nature of the function (its redefinition by goodie type) to be the determining factor for Penelope’s action. |
| **AGENTS:** getParalysisStatus(), swapParalysisStatus() | Paralysis was a shared functionality across citizens and both zombies. In turn, I added a virtual function that kept track of the paralysis status of the agent. getParalyisStatus() simply told us whether or not the agent was paralyzed. At the end of every doSomething() for zombies and citizens, swapParalysisStatus() was called to freeze them for the next turn. |
| **HUMAN:** makePoison(), curePoison(), isPoisoned(), getPoisonCt(), addPoison(), | Getting poisoned was a common feature for Penelope and citizens, so functions that handled poison were created as virtual in the human superclass (Penelope and citizens derive from here). makePoison() sets the Boolean poison status to true and addPoison() will increment the poisonCount() every tick if the individual is poisoned (until 500 ticks where the citizen would turn to a zombie, or the player would die). getPoisonCt() and isPoisoned() were ways to access the private data. curePoison() was called only in Penelope’s doSomething when she used a vaccine. |
| **HUMAN:** setExit(), getExit() | Citizens and Penelope also had the shared functionality of using exits. setExit() and getExit() allowed each of these individuals to use the useExitIfAppropriate() function. Citizens simply had to check for overlap, while Penelope had to check for overlap and had to ensure that no live citizens remained. |
| **PENELOPE:** addNVacc(), addNFlame(), addNMines(), getNVacc(), getNFlame(), getNMines() | Penelope was the only actor who could pick up goodies and store them. These functions represented ways to check the number of each goodie and ways to increment the number of each goodie if it was picked up. |

1. **Functionality that was skipped:**

I did not, on any large scale, skip any major functionality in the project.

There were a few edge cases that did not present solutions; for instance, at times the game temporarily did not permit citizens to step on the exit. Additionally, there were case instances where citizens would not move despite proximity to zombies.

1. **Design Assumptions that I made:**

Randomly selecting movement when the citizen/smart zombie did not share an X or a Y coordinate with its trigger was a 50-50 shot between movement in X or in Y. It handled the random motion accordingly, as it would act as though the X direction or Y direction were fixed depending on the randomly generated number between 1 and 2.

1. **How I tested each class:**

**PENELOPE**

I initially developed Penelope in part one of the project. The goal was simply to make sure that she could take in user input and move in directions that were not blocked off by walls. To test this, I played through the first level of the game many times, attempting different maneuvers, walking along walls, as well as rapidly changing directions. I had to ensure that Penelope was responsive to movement keystrokes, not only for moving but also for changing direction.

The next step was testing Penelope’s mortality. I did this by leading Penelope to overlap with Pits (which were tested on their own) as well as by allowing her to overlap with fire (which was tested in flames/landmines). This should have returned player dead, and ended the level. I made sure to overlap with a pit from all possible directions, and to experience flame from the landmines.

The next step was testing Penelope’s ability to use goodies. I had to ensure that when Penelope overlapped with goodies, her count of the goodies was incremented. This was done by stepping on the goodies in the playing field and checking for changes in the private variables (as displayed on the stringstream above the game arena). In addition to picking up Goodies, she had to be able to use them effectively, which also meant decrementing the count of the goodies she possessed. Testing the actual effects of goodies is described in the goodies section of the write-up.

The next step was testing Penelope’s interaction with other agents. I did so by implementing citizens and checking if they could track and follow Penelope’s movements. This required functions that looked for the nearest citizen trigger and allowed citizens to pursue that trigger (Penelope). I then implemented dumb zombies. The main interaction to test for here was the vomit; I would make Penelope stand next to zombies and experience what would happen if the zombie were to vomit on Penelope. This would trigger the poison counter, which appeared on the stringstream (I was able to track it here to make sure it got to 500 prior to killing Penelope). Finally, I tested smart zombies by implementing similar tracking functions to those in citizen. These tracking functions allow smart zombies to move in the general direction of Penelope (in the absence of citizens and provided the player is sufficiently close). Testing these cases required a great deal of playing the game over and over again (and at times omitting confounding objects to check for object interactions).

**WALL**

I first tested the wall in part one of the project. The main idea in this case was to ensure that the wall would block the movement of Penelope. At first, the wall did not block Penelope’s movement entirely; she would be able to enter the wall partially. This, by extension, led to a modification in my function that checked collisions. Ultimately, Penelope was blocked off from the wall. Yet, Penelope is not the only actor that had to be blocked by the wall.

My next tests were checking if flames fired by Penelope and landmines were blocked by walls. I did this hand in hand with the flame and landmine classes. I would fire flames at the wall (from distances corresponding to no flames appearing on the screen, one flame, and two flames) to ensure that the flames would stop if they collided with the wall. For landmines, I planted landmines next to the wall and fired flames at them to see if the flames were blocked on the side against the wall.

The next step was checking if moving elements like zombies and citizens were blocked off by walls. This was done by leading citizens and smart zombies towards walls and observing collisions. When testing the game as a whole, I was able to see dumb zombies colliding with walls, but also saw it if I planted a dumb zombie in a level against a wall.

**PIT**

I tested pits by first checking if Penelope would fall through them upon overlapping with a Euclidian distance of about 10 pixels (used ActivateOnAppropriateActors) and lose a life. Once that was implemented, I had to ensure that if citizens or zombies entered pits, they would also lose lives.

Finally, I had to make sure that flames could be shot across pits. This was done in level 4, where the gas can is above the pits. This made it conducive to picking up the gas can goodie and firing flames across the pits at zombies. Also a landmine is available to check if flames from landmines can cross (though it would work the same way, given the landmine dynamically creates flame objects just as Penelope does).

Another issue that I encountered was destruction of Penelope if the user pressed q on the welcome screen. This was fixed by the private Penelope member of StudentWorld being set to nullptr prior to initialization when the level was built.

**FIRE**

I tested fire by first implementing Penelope’ s ability to shoot flames after picking up a Gas Can goodie object. Once I was able to create new flame objects in the for loop, I tried firing them at walls and at Goodie objects. Walls were meant to block any flame objects that collided with them (see Wall test methods). On the other hand, firing flames at goodies should call a method that sets them as dead (as should happen when flames overlap with members of the Agent subclass too). The goal was to make sure that when flames overlapped with goodies, the goodies disappeared and were no longer usable.

After firing flames at stationary objects, I was able to implement landmines. I planted landmines after establishing methods to pick up Landmine goodies. I would step on landmines to check if flames created by landmines were able to kill the player. After that, I would plant landmines near walls and near goodie objects and fire flames at them to see if these flames would trigger an explosion of the landmine.

Also, I had to ensure that the duration of the flame was only two ticks. I did this by using cerr statements to keep track of the duration member variable that I had in each fire object.

Finally, once objects of the Agent subclass (zombies and citizens) were implemented, I had to ensure that firing flames at these objects would lead to their death (and corresponding change in score). I did this by, both firing flames at citizens and zombies, as well as planting landmines and allowing citizens and zombies to tread on landmines, triggering fire release.

**GOODIES**

In a general sense, I had to ensure that when the player overlapped with the goodies, the goodies were marked as dead (and removed from the screen) while the member variables that held the number of goodies were incremented. I made sure of this by trying to overlap with each type of goodie from every possible direction, and checking values of the member variables that held goodie counts immediately afterwards.

*VACCINES*

After ensuring that vaccines were picked up, I had to ensure that hitting enter and activating the vaccine decremented the number of vaccines, dropped the poison tick counter to 0, and reset the Boolean status of whether or not Penelope was poisoned. I did this by artificially setting the Boolean to state the Penelope was poisoned and setting the poison counter to a random number. After ensuring that the enter key worked in this case, vaccines had to be tested again in real time during the game to see if activation within 500 ticks led to the same results.

*GAS CAN*

After ensuring that gas cans were picked up, I had to ensure that Penelope had 5 flames charged up. This was checked in a cerr statement but was later transferred to the stringstream for checking. I fired the flames 5 times to ensure that the count for flames remaining decremented appropriately. The location of the flames from the gas can were checked according to my description of how the flame class was tested.

*LANDMINE (GOODIES + ACTIVATING OBJECT)*

After ensuring that landmine goodies were picked up, I had to ensure that the count of landmines possessed would increase by 1. Whenever the landmine was dropped using the tab key, the count had to decrement and a landmine had to be placed under the getX(), getY() position of the player.

An important facet to consider was the activation time for landmines. I had to make sure that landmines were inactive for a short period of time (30 ticks) before they were active (meaning if they were stepped on they would explode). Additionally, I had to ensure that firing flames at a landmine would cause an explosion despite tick count. All of these criteria had to be considered when firing flames at landmines or stepping on landmines. Testing landmines, in turn, was a two-fold process. Step one was ensuring that the goodie was picked up, and step two was ensuring that the landmine activating object was placed on the level field and did the right thing in terms of releasing flames upon explosion.

**VOMIT**

When testing vomit, I had to ensure that a zombie object was implemented (even if the distinction between dumb and smart zombies had not been tested yet). I had to first ensure that zombies would create the vomit object appropriately (taking into consideration location of the human (player/citizen) with respect to the zombie (more on this in testing the zombie classes). After allowing zombies to fire vomit at humans, I tested on Penelope and on the citizens, allowing both to get hit by vomit and get infected.

I used cerr statements to check on the status of the infection (poison count) for citizens, and used the stringstream to monitor it for Penelope. I had to ensure that in 500 ticks of being poisoned, Penelope would die, or in the case of citizens, they would be replaced by a zombie. Testing vomit was a matter of keeping track of the effects of vomit on the poison count and poison status of the human class.

**DUMB ZOMBIE**

Testing dumb zombie mainly revolved around determining if new movement plans were being created and if vomit was being fired. I tested this by using cerr to write out the movement plans for each dumb zombie. I tested vomit by moving Penelope and citizen objects adjacent to dumb zombies, trying to evoke a vomit response.

I also had to ensure that previous activating object functionality worked with dumb zombies. This meant testing flames (from Penelope and landmines) and pits to ensure that they killed zombies. Also, I had to make sure that zombies did not pick up goodies.

Finally, I had to check the added perk that dumb zombies could drop vaccine goodies. A snag that I hit was that after dumb zombies died, they would drop vaccines; however, since zombies were killed by fire, these flames also killed the vaccine goodie that was dropped. I fixed this by providing a buffer time for dropped vaccines such that they would not die if fire overlapped with them upon conception (this buffer was NOT applied to vaccines that were initialized with the level).

**SMART ZOMBIE**

After implementing smart zombie, I had to ensure that it differed from dumb zombie in its movement plan. I used cerr statements to check the plan for the smart zombie handling cases like resetting the plan if movement was blocked and checking for distance to triggers like Penelope and citizens. I had to test the nature shared by zombies like vomiting on humans who are adjacent to them; I did this by making Penelope, as well as citizens move adjacent to the zombies.

I tested the motion towards citizens and zombies by moving Penelope around near the smart zombie objects. This was especially do-able on level 4, as the zombies were on the other side of a wall, but would still follow Penelope’s movement.

I also tested flame and pit functionality on these zombies in level 4, shooting flames at them and leading them into pits to ensure that their death was functional.

**CITIZEN**

Implementing citizen was tricky. The preliminary tests I did were to check if they died when overlapping with flames or pits. After that, I tested functionality that allowed citizens to exit the level; I had to be sure to account for decrementing the number of live citizens as this had bearing on Penelope’s ability to exit the level.

After testing for the functionality of citizen on its own, I had to test citizen with respect to other actors. The first step to this was implementing the locateNearestCitizenTrigger with just Penelope. This allowed me to check if the citizens would gravitate towards Penelope when she was closeby. I used cerr statements to check the distance from Penelope.

After this, I implemented zombies which allowed me to test citizen’s actions when interacting with zombies. I tested if citizens could get vomited on, and if they would turn into zombies. I killed these zombies with fire and used the resulting score change as a gauge as to whether the zombie generated was a dumb or smart one. I also was able to then gauge whether or not citizens would flee zombies (again, I used cerr statements to check distances).

I played through the game many times and monitored these interactions.

**LEVEL INITIALIZER**

I first tried to build each level, jumping to each level using a function that I had in StudentWorld. I then played through every level to make sure that I could make it to the last level. Then, I went in and added errors that would cause levels to not build into level files to detect if it would return an error. I played through the game various times to make sure levels built properly.