Nathan Tung
Com Sci 32 Dis 2C
February 28, 2012

CS32 Project 3 Report

a. ***A high-level description of each of your public member functions in each of your classes, and why you chose to define each member function in its host class; also explain why (or why not) you decided to make each function virtual or pure virtual.***

StudentWorld:

`StudentWorld::StudentWorld();`
- Called when the StudentWorld is constructed
- Initializes the member counter variable to 0 (for adding insects)

`~StudentWorld();`
- Deletes all GameObject pointers in the vector to prevent memory leak; clears vector

`virtual void init();`
- Initializes a new round of the game
- Creates the player and the specified number of mushrooms in random positions without overlap
- Store all those GameObject pointers into the vector
- Made virtual just in case a different kind of world is to be designed later

`virtual int move();`
- Tells all current objects in the world that are still alive to do something
- If any object died during that last change, delete them from the vector
- Return an integer indicating whether the game should continue
- Made virtual just in case a different kind of world is to be designed later

`virtual void cleanUp();`
- Deletes all GameObject pointers in the vector to prevent memory leak; clears vector
- Made virtual just in case a different kind of world is to be designed later

`bool poisonMushroomThere(int x, int y);`
- Return whether there is a poison mushroom at the coordinates x, y

`bool mushroomThere(int x, int y);`
- Return whether there is any kind of mushroom at the coordinates x, y

`bool insectThere(int x, int y);`
- Return whether there is any insect at the coordinates x, y

`bool playerThere(int x, int y);`
- Return whether there is a player at the coordinates x, y

`bool scentipedeSegmentThere(int x, int y);`
- Return whether there is a scentipede at the coordinates x, y

`void storeObject(GameObject* temp);`
- Add the GameObject pointer to the private member vector in StudentWorld

`void attackObjects(int x, int y);`
- Set any mushroom or insect at coordinates x, y to dead
- Give the player the corresponding amount of points for each insect

`void dropMushroom(int x, int y);`

- Create a new normal mushroom at coordinates x, y
- Add its pointer into the vector

`void removeMushroom(int x, int y);`

- If there's any kind of mushroom at coordinates x, y, set it to dead

`void makeMushroomPoisonous(int x, int y);`

- If there's a normal mushroom at coordinates x, y, change its ID to that of a poison mushroom

`void removeSegment(int x, int y);`

- If there's a scentipede segment at coordinates x, y, set it to dead

`void killPlayer(int x, int y);`

- If the player is at coordinates x, y, set it to dead

`void addInsects();`

- Add each type of the four insects if test parameters are provided and counter is divisible by it
- If no test parameters specified, count the number of each insect type and determine player level
- Determine if there's no scentipede segment in the top row, add it
- Calculate the conditions and probabilities for the other insects and add them if necessary

`void addScentipede();`

- Randomly determine the number of scentipede segments to add (between 6 and 12)
- Create that number of new scentipedes, starting from the top left of the world
- Add all the pointers into the vector

`void addFlea();`

- Randomly determine the x position of the flea to add
- Create that flea at the top of the world and add its pointer into the vector

`void addScorpion();`

- Randomly determine the y position of the scorpion to add
- Create that scorpion at the left of the world and add its pointer into the vector

`void addSpider();`

- Randomly determine the y position of the spider to add; randomly determine whether it is to be added to the left or the right side
- Create that spider and add its pointer into the vector

<u>GameObject:</u>

`GameObject(int id, StudentWorld* world);`

- Create a game object with a world pointer, ID, and default coordinates (0, 0)
- Set that object's display to true

`GameObject(int id, StudentWorld* world, int x, int y);`

- Create a game object with a world pointer, ID, and specified coordinates x, y
- Set that object's display to true

`virtual ~GameObject();`

- Set that object's display to false; virtual because GameObject is a base class `virtual void doSomething() = 0;`

- Pure virtual, since no GameObject will be created to move in such a way; derived classes all do something differently

`virtual bool isStillAlive();`

- Return whether that object is still alive (returns a private boolean)

`virtual void setDead();`

- Changes that object to dead state (changes private boolean)

`StudentWorld* getWorld() const;`

- Fetch and return the pointer to the world it's in; the world doesn't change, so it's constant

`int randInt(int lowest, int highest);`

- Generate a random integer, inclusively, from lowest to highest parameter integers
- Used to determine insect movements

<u>Player:</u>

`Player(StudentWorld* world);`

- Create a player with a world pointer and set coordinates (15, 0)

`~Player();`

- Player destructor, which essentially has nothing to free

`virtual void doSomething();`

- Get the user's input, which might be to move the player, shoot, or do nothing
- If it is an arrow key, pass that into movePlayer function
- If it is to shoot, create WaterDroplets in a vertical line until it hits something or leaves bounds
- If something is "hit," call the attackObject() function on that position
- Change "can shoot" boolean so that the player can only shoot every other tick
- If the player moves onto an insect, set the player to dead

`void movePlayer(int dir);`

- Check the player's new potential position for mushrooms and insects
- Kill the player if there's an insect
- Move if the player won't run into a mushroom

<u>Mushroom:</u>

`Mushroom(StudentWorld* world, int x, int y);`

- Create a mushroom with a world pointer and specified coordinates x, y
- Initialize its number of health to 4

`~Mushroom();`

- Mushroom destructor, which essentially has nothing to free

`virtual void doSomething();`

- If mushroom has no health, set it to dead

`int getHealth() const;`

- Return mushroom's health without changing it

`void lowerHealth();`

- Decrement mushroom's health by 1

<u>WaterDroplet:</u>

`WaterDroplet(StudentWorld* world, int x, int y);`

- Create a WaterDroplet with a world pointer and specified coordinates x, y
- Initialize its number of ticks to 2

`~WaterDroplet();`

- WaterDroplet destructor, which essentially has nothing to free

`virtual void doSomething();`

- Decrement number of ticks
- If it has no ticks, set it to dead

<u>Insect:</u>

`Insect(int id, StudentWorld* world, int x, int y);`

- Create an insect with an ID, world pointer and specified coordinates x, y

`virtual ~Insect();`

- Destructor for insect, which essentially has nothing to free; virtual because it's a base class

`virtual void doSomething() = 0;`

- Like GameObject, no Insect will ever be made to doSomething; pure virtual, since all Insect derived classes also have their own way of doing something

<u>Scentipede:</u>

`Scentipede(StudentWorld* world, int x, int y);`

- Create a Scentipede with a world pointer and specified coordinates x, y
- Set its state to move down, move right, and not poisoned

`~Scentipede();`

- Scentipede destructor, which essentially has nothing to free

`virtual void doSomething();`

- If the Scentipede is poisoned, make it move down; reaching the bottom makes it normal again
- Kill other scentipedes, mushrooms, or the player if it moves down onto them while poisoned
- If it's not poisoned, calculate its next position; change direction if it reaches bounds
- It can't move to the new position if there is a mushroom, player, or other scentipede
- If only a player is there, kill the player
- If there is a poisoned mushroom, set the scentipede to poisoned state
- Change its direction to moving down
- If it goes onto a mushroom, player, or other scentipede, set it to dead, then move there

`virtual void setDead();`

- Call the GameObject's version of setDead
- In addition, randomly spawn with 33% chance a mushroom where the Scentipede is

<u>Flea:</u>

`Flea(StudentWorld* world, int x, int y);`

- Create a Flea with a world pointer and specified coordinates x, y

`~Flea();`

- Flea destructor, which essentially has nothing to free

```
virtual void doSomething();
```
- If the flea is in bounds but not on a mushroom, spawn a mushroom there with 25% chance
- Move down one position if possible; if it's out of bounds, set the flea to dead
- If it moves onto a player, kill the player

<u>Spider:</u>

```
Spider(StudentWorld* world, int x, int y);
```
- Create a Spider with a world pointer and specified coordinates x, y
- Give it a random distance integer between 1 and 4, inclusively; set its initial moving directions
- Make its rest variable false; it will determine which ticks the spider should rest

```
~Spider();
```
- Spider destructor, which essentially has nothing to free

```
virtual void doSomething();
```
- If the distance is not positive, change its vertical distance and reset its distance
- If not resting, calculate its new diagonal position, move there, and decrement distance
- If it moves onto a mushroom, kill the mushroom
- If it moves onto a player, kill the player; set Spider to dead if it leaves bounds
- If it moved, make it rest next tick; otherwise, make it move the next tick

<u>Scorpion:</u>

```
Scorpion(StudentWorld* world, int x, int y);
```
- Create a Scorpion with a world pointer and specified coordinates x, y

```
~Scorpion();
```
- Scorpion destructor, which essentially has nothing to free

```
virtual void doSomething();
```
- If it is on a mushroom, there is 33% chance of making that mushroom poisonous
- The scorpion tries to move right; if it will leave bounds, set it to dead
- If it moves onto a player, kill the player

I made many public functions to use in the StudentWorld class, since they can be called in the GameObject classes with the help of the StudentWorld pointer. Without these public functions, we could not be able to access or update the private member variables, such as the vector holding all the GameObject pointers. My GameObject base class has (in addition to constructors and a virtual destructor) many functions that are basically used by all derived objects. For example, all objects can do something during a tick, die, etc. The function doSomething is pure virtual because a GameObject type object will never have its own base way to act in such a method. However, setDead is virtual since most objects implement it regularly, but few classes might want to change it around. Likewise, my Insect base class copies that same structure, and essentially it is only a cookie cutter for its derived classes. I did not group WaterDroplets and Mushrooms together with a "health-based" superclass, since they operate differently. I thought it would be helpful to distinguish between a user-caused health decrement (as in Mushrooms) and an automatic tick counter decrement (as in WaterDroplets).

Nathan Tung
Com Sci 32 Dis 2C
February 28, 2012

b. ***A list of all functionality that you failed to finish as well as known bugs in your classes.***

To my knowledge, I have finished all functionality that is listed in the project specifications. I originally did have some bugs with how the scentipede would behave (especially under the influence of poison), but that should have been sorted out.

c. ***A list of other design decisions and assumptions you made, e.g.: It was ambiguous what to do in situation X, and this is what I decided to do.***

In the project specifications, the player is not allowed to shoot WaterDroplets every single tick. The Spider must choose to rest every other tick as well. I decided to initialize the boolean in such a way that the player is allowed to shoot during the first tick (but not the one after that, etc.) and the Spider will not rest during the first tick (but will during the second tick, fourth tick, etc.). Furthermore, I since some insects (like Scentipedes) might move "multiple times" in one tick, in some cases I might have checked multiple times to see if the player should be deemed "killed." Lastly, a counter variable is supposed to be incremented in the move function, but since further specifications are lacking, I decided to increment the counter at the very beginning of the move function, prior to calling insects (which is where the counter matters).

In short: the player can start out shooting WaterDroplets, the Spider will not rest during the first tick, and the member counter variable (for testing parameters) is incremented in the very front of the move function.

d. ***A description of how you tested each of your classes.***

Player

In order to test the player, I spawned the player by itself in the world to test its boundaries. Then I added mushrooms to make sure that the player could not move onto an occupied space. Lastly, I added all types of insects onto the map to see whether the player would be set dead correctly when encountering the same spot as another object. It was difficult to test specific functions except by playing the game, and the player's movements seem to be fluid and working as specified. To verify that the correct number of points was added per kill, I froze all the objects by disabling their move method, and then killed them off from there.

Mushroom

The mushroom class essentially does nothing except block the pathways of scentipedes and the player. To test this, I spawned more mushrooms than normal at the beginning of the game, then spawned the insects. As predicted, the player and scentipedes (that were unpoisoned) could not move through them. The other insects when performing their actions could, however, remove mushrooms in one go, as intended. In order to test whether they had the correct amount of health and brightness, I shot them in an isolated world without other objects and saw that they indeed dimmed in brightness after each shot, and were completely removed from the world after 4 hits.

WaterDroplet

WaterDroplets are specified in the instructions to have a lifetime of two ticks. It lasts for one tick after it's fired so that the player can actually see the trajectory of the bullet fired. Two things had to be tested for the WaterDroplet: whether it would stop appearing once it hit an object or bounds, and whether it would only fire only if it didn't fire the last tick. For the first point, I simply shot at insects and mushrooms to make sure no WaterDroplets were displayed beyond the object, then outputted the WaterDroplets' coordinates for good measure. For the latter, I outputted a message in the command prompt every time a WaterDroplet could not be fired to make sure it had a correct firing restriction.

Scentipede

The scentipede segments are arguably the most complicated Insect subclass. However, testing it is still quite similar to testing with the other objects. There are two states for the scentipede that we have to account for: a poisoned state and a normal state. A normal scentipede must avoid mushrooms by going around them, then change horizontal directions when it hits left or right bounds. Then it starts moving upward when it hits the bottom, and vice versa. Simply create a row of scentipede segments in a row of only mushrooms and confirm that it loops around the mushrooms and goes from top to bottom in an emergent and zigzag fashion. For the next state, insert a poisonous mushroom right in front of the scentipede's path. Verify that it will go all the way down, removing any mushrooms in between, until it is at the bottom and is no longer poisoned. Regular behavior should then commence.

Flea

Fleas are simpler Insects in comparison to some others. In essence, it starts from the top at a random x coordinate, and then drops straight down with a chance to drop mushrooms wherever it lands. It is somewhat easy to see whether the fleas are working correctly. I created many fleas during the initialization part of an empty world then watched them drop down and create mushrooms. They will drop a mushroom about a third of the time, which is correct.

Spider

The spiders are a little complicated as Insects. They have to keep recalculating their vertical move distance as well as their newest diagonal direction. To test this part, we can simply put spiders in an empty word and call their move function. Each time they move, output their coordinate positions, and optionally, their distance measure, to see that they are moving correctly as well as resting every other turn (as the specifications mention). Since the world is in a sort of 3D shape, it can be difficult to tell where the spider is being removed; has it really gone out of bounds? The coordinate outputs will also enable us to check for this.

Scorpion

Scorpions are also slightly simpler than the other Insects. Since they always spawn on the left side of the world, all we have to is place a row of mushrooms in front of them and make a world where only spiders and mushrooms exist. When the scorpions run through the mushrooms, they don't collide, but rather have a 1/3 chance of converting any passed-over mushroom into a poisonous one. After it runs through that coordinate line, it will be removed. We can simply return its x-coordinate before it dies to make sure it has reached the edge correctly.

The easiest way to test how the actors react to each other collectively is to simply play the game. It might help to eliminate all insects but two, etc. just to see what happens when they occupy the same position. In all cases, whenever the player touches any insect, the player must die. Whenever a poisoned scentipede descends onto another segment below it, the non-poisoned scentipede must die. For the most part, however, insects ignore each other. The StudentWorld itself can be tested by creating objects on all its boundaries and see if they are displayed correctly and that memory is correctly released.