Kalin Khemka

30433696

Project 3 Report

1. 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. For example, “I chose to define a pure virtual version of the sneeze() function in my base Actor class because all actors in Bug Blast are able to sneeze, and each type of actor sneezes in a different way.” 50

1.

**STUDENTWORLD:**

StudentWorld() – The constructor, does nothing really

virtual ~StudentWorld() – The destructor, frees all the dynamic memory allocated

virtual int init() – Creates the level set up every time a new game is started, the player dies, or reaches a new level. Loads the level and puts in all the objects within the grid. Given function to program.

void setDisplayText() – Basically creates the text that is inputted on the screen. Using OSStream to create the text with the proper formatting, it passes the value on to the gameWorld function to set the text on the screen.

virtual int move() – moves all the objects on the grid, determines if the player is dead or not and based on that takes action. It also updates the text at the top of the screen every time, updates the bonus, and deletes dead objects.

virtual void cleanUp() – deletes all the dynamically allocated memory from the studentWorld

Actor\* objectAtSpot(int x, int y) const – returns a pointer to the actor located at that x and y location. The studentworld is the only one that knows what object is where.

Actor\* objectAtSpotZumi(int x, int y) const – returns a pointer to the actor located at that x and y location except if it is a Zumi, in which case it returns either any other object on that spot or nullptr. This is for the Zumi function to check for other objects on the location on which it is standing. The studentworld is the only one that knows what object is where.

void getPlayerLocation(int& x, int& y) const – gives the X and y location of the player on the grid. The studentworld is the only one that knows where the player is

Player\* getCollidingPlayer(Actor\* a) const – if there is a player on the same location as another actor, it will give a pointer to that player, otherwise it will give a nullptr reference. The studentworld is the only one that knows where the player is

void activatePlayerWalkThroughWalls() – Tells the player object that it has a powerup and that it needs to activate its ability to walk through walls. The student world has to tell the player this because it won’t know if it is on the same spot as a powerup or not.

bool playerCanWalkThroughWalls() const – checks whether the player is in a situation where it can walk through walls, this is important for the movement of the player object.

void increasePlayerSimultaneousSprayers() – tells the player that it received a powerup that lets it put down more sprayers. Again, only the world can tell the player that is has eaten a powerup.

bool anyLiveZumi() const – checks whether there are any live Zumis left in the grid and then returns a bool saying whether there are or not – only the student world knows whether there are zumis left or not

void setLevelFinished() – Sets a Boolean private data member within the studentWorld to true indicating that the level is done. Then, in the move() function, when it checks the value of the variable, it will know whether the level is over or not

void addActor(Actor\* a) – allows other classes to add actors to the studentWorld, such as a player dropping a bugsprayer or a zumi dropping a goodie. You pass in what actor you would like to add and it is put in the list with the other actors.

unsigned int getProbOfGoodieOverall() const – gives the probability of a goodie dropping

unsigned int getProbOfExtraLifeGoodie() const – gives the probability of an extra life goodie

unsigned int getProbOfWalkThruGoodie() const – probability of a walkthroughwalls goodie

unsigned int getProbOfMoreSprayersGoodie() const – probability of a increasesprayers goodie

unsigned int getTicksPerSimpleZumiMove() const – number of ticks that a simpleZumi has to wait to move

unsigned int getTicksPerComplexZumiMove() const – number of ticks a complexZumi has to wait to move

unsigned int getGoodieLifetimeInTicks() const – the number of ticks a goodie stays alive when dropped

unsigned int getMaxBoostedSprayers() const – number of max sprayers when given powerup

unsigned int getWalkThruLifetimeTicks() const – number of ticks that walkthroughwalls power stays

unsigned int getBoostedSprayerLifetimeTicks() const – number of ticks that increasedsprayer power stays

unsigned int getComplexZumiSearchDistance() const – distance of grid spots that a complex sumi searches for when trying to find the player

**ACTOR:**

Actor(StudentWorld\* world, int imageID, int startX, int startY) :GraphObject(imageID, startX, startY) – Constructor, creates the GraphObject for the base Actor, the first parameter for making any object is passing the studentWorld that it is in so that it can have access to the functions within the studentWorld class

~Actor() – sets the actor invisible in case it wasn’t already and deletes the object

virtual void doSomething() = 0; - the doSomething function was set as a pure virtual function because every actor does something different within the doSomething. There was no point in making it do something in the base class

virtual void setDead() – this function basically sets the private data member m\_isDead to true and makes the object invisible. This is virtual because in some classes, like the player, the function needs to do other things as well

bool isDead() const – returns a Boolean saying whether the actor is dead or alive

StudentWorld\* getWorld() const – This returns a pointer to the studentWorld that the actor is in

virtual void applyBugSpray() - This function takes action on the actor if it is hit by BugSpray. In the actor class it is left a virtual function that is empty because for some actors, nothing happens when bug spray hits them. For the classes where something happens, the function’s actions are changed.

virtual bool blocksPlayer() const{return true;} – returns whether the actor blocks the player from walking onto that spot, this is also virtual because some actors do not block the player, like Zumi and Goodies

virtual bool colocationKillsPlayer() const{return false;} - This tells whether the player will die if he stands on the same spot as the actor. This is also virtual because it changes, such as with zumis it will true

virtual bool blocksZumiAndSprayers() const{return false;} – this function says whether zumis and sprayers can move/be placed on those spots – this is also virtual because for objects like bricks it is true and sprayers, but everything else is different.

virtual bool allowsSprayToBeDropped() const{return true;} – this function returns whether or not the spray from the bugsprayers can be paced on the actor, this is also virtual because it can change to false if there are bricks

virtual bool allowsSprayToPass() const{return true;} – this function returns whether or not the spray can continue on in the direction that it is going. This is virtual because it changes for objects like bricks because if there is a brick, the spray will not go through in and on the other side of the brick, it will just stop

virtual bool isZumi() const {return false;} – this function returns whether the object is a Zumi or not, it is default false, but within the Zumi class this is changed to true, thus it is virtual

**PLAYER:**

Player(StudentWorld\* world, int startX, int startY):Actor(world, IID\_PLAYER, startX, startY) - - The constructor for Player which initializes all the private data members and also creates an actor with the imageID of the player.

~Player(){} – the destructor – does nothing really

virtual void setDead(); - The virtual function setDead is changed here because it adds the ability to play the sound when the player dies. This is not needed for the other actors so the virtual function is changed only here like that

virtual void doSomething(); - the doSomething function for the player checks whether the player isDead, then takes in user input for moves and executes them.

virtual void applyBugSpray() – this basically sets the player to dead to end the current game/level – virtual because it changes with actors

void activateWalkThroughWalls(unsigned int lifetime) – changes the private data members to allow the player to walk through walls for the amount of time inputted

bool canWalkThroughWalls() const – returns whether the player currently has the ability to walk through walls or not

void increaseSimultaneousSprayers(unsigned int max, unsigned int lifetime) – changes the private data members to allow the player to place “max” amount of sprayers at a time for “lifetime” amount of time

void reduceActiveSprayers() – this function basically updates the number of current sprayers on the map every time it dies (called by the bugSprayer’s setDead() command)

**BRICK:**

Brick(StudentWorld\* world, int imageID, int startX, int startY):Actor(world, imageID, startX, startY) {} – constructor, does nothing as this is yet another base class

virtual ~Brick() {} – destructor that does nothing really

virtual void doSomething(){} – the doSomething for bricks does absolutely nothing

virtual bool blocksZumiAndSprayers() const{return true;} – changes the base class’s function’s return values to block Zumis and Sprayers

virtual bool allowsSprayToPass() const{return false;} – changes the base class’s function’s return values to block Spray from passing

**PERMANENTBRICK:**

PermaBrick(StudentWorld\* world, int startX, int startY) :Brick(world, IID\_PERMA\_BRICK, startX, startY) {} – constructor that creates the permaBrick with the correct imageID

~PermaBrick() {} – destructor that does nothing

virtual bool allowsSprayToBeDropped() const{return false;} – Changes the value of the base function’s return because nothing that kill permanent bricks. This is the only class that returns false for this function

**DESTROYABLEBRICK:**

DestBrick(StudentWorld\* world, int startX, int startY):Brick(world, IID\_DESTROYABLE\_BRICK, startX, startY) {} – constructor that makes the destBrick with the proper imageID

~DestBrick() {} – destructor that does nothing

virtual void applyBugSpray() – because the brick can be destroyed, the function sets the brick to dead and makes it invisible

virtual bool colocationKillsPlayer() const; - this function returns true if the player is not in walkThroughWalls mode and returns false if he has that ability at the current Tick.

virtual bool blocksPlayer() const; - this function returns true if the player is not in walkThroughWalls mode and returns false if he has that ability at the current Tick.

**EXIT:**

Exit(StudentWorld\* world, int startX, int startY):Actor(world, IID\_EXIT, startX, startY) {setVisible(false);} – constructor that sets the exit to invisible. Has to do this because the base class contructor sets everything to visible when an object is created and this is the only class that needs it to be invisible when created

virtual ~Exit() {} – destructor that does nothing

virtual void doSomething(); - the doSomething function checks whether the player is on the spot of the exit and whether or not the exit is visible. If both the conditions are met, then the studentWorld gets it level set to finished.

virtual bool blocksPlayer() const{return false;} – the exit does not block the player from walking onto it

**TIMEDLIFEACTOR:**

TimedLifetimeActor(StudentWorld\* world, int imageID, int startX, int startY, unsigned int lifetime) :Actor(world, imageID, startX, startY) – base constructor that sets the private data member for the length of life of the object

virtual bool blocksPlayer() const{return false;} – these timedActors do not block the player from walking on them

unsigned int getLife() const – This returns the number of ticks left before the timed Actor dies

void decreaseLife() – this function decrements the length of life left for the actor before it dies

void expireImmediately() – this function sets the length of life of the actor to 0 immediately so that on the next tick it dies first

**BUGSPRAYER:**

BugSprayer(StudentWorld\* world, int startX, int startY) :TimedLifetimeActor(world, IID\_BUGSPRAYER, startX, startY, 40) {} – constructor that gives the proper imageID and sets the life of the sprayer to 40 ticks.

virtual void doSomething(); - the doSomething function checks for the length of life left in the bugsprayer and based on that takes action. If there is time left, it will decrement the life. If there is no time left, then it will unleash the bugspray in all 4 directions as far as it can, with a max of 2 grid slots.

virtual void applyBugSpray() – this function expires the time immediately on the bugsprayer and sets it off by calling doSomething again for the object

virtual void setDead(); - this function sets the object dead and then calls upon the player function to reduce the number of active bugsprayers on the map

virtual bool blocksZumiAndSprayers() const{return true;}

**BUGSPRAY:**

BugSpray(StudentWorld\* world, int startX, int startY) :TimedLifetimeActor(world, IID\_BUGSPRAY, startX, startY, 3) {} – the bugSpray constructor that gives it the proper imageID and gives it a life length of 3 ticks

virtual void doSomething(); - the doSomething function checks for the length of life left and if there is none then calls the applyBugspray on all objects that share the space with the bugspray and then sets the object dead

GOODIE:

Goodie(StudentWorld\* world, int imageID, int startX, int startY, int lifetime) :TimedLifetimeActor(world, imageID, startX, startY, lifetime) {} – constructor that simply makes a timedLifeActor

virtual ~Goodie() {} destructor that does nothing

WALKTHRUGOODIE:

WalkThruGoodie(StudentWorld\* world, int startX, int startY, int lifetime):Goodie(world, IID\_WALK\_THRU\_GOODIE, startX, startY, lifetime) {} – constructor that gives the proper imageID and lifetime from the input

~WalkThruGoodie() {} – destructor that does nothing

virtual void doSomething(); - the doSomething basically decrements the life of the object until either a player steps on it or it dies first. If the player steps on it, it activated the walkthroughwall ability of the player using the studentWorld function

EXTRALIFEGOODIE:

ExtraLifeGoodie(StudentWorld\* world, int startX, int startY, int lifetime):Goodie(world, IID\_EXTRA\_LIFE\_GOODIE, startX, startY, lifetime) {} - constructor that gives the proper imageID and lifetime from the input

~ExtraLifeGoodie() {}– destructor that does nothing

virtual void doSomething();- the doSomething basically decrements the life of the object until either a player steps on it or it dies first. If the player steps on it, it increases the lives of the player by 1 using the gameWorld function

INCREASESPRAYERGOODIE:

IncreaseSprayersGoodie(StudentWorld\* world, int startX, int startY, int lifetime):Goodie(world, IID\_INCREASE\_SIMULTANEOUS\_SPRAYER\_GOODIE, startX, startY, lifetime) {} – constructor that gives the proper imageID and lifetime from the input

~IncreaseSprayersGoodie() {}– destructor that does nothing

virtual void doSomething();- the doSomething basically decrements the life of the object until either a player steps on it or it dies first. If the player steps on it, it activates the ability of the player to place more than 2 sprayers on the grid using the studentWorld function

**ZUMI:**

Zumi(StudentWorld\* world, int imageID, int startX, int startY); - the base constructor which creates an Actor object

virtual ~Zumi() {} – destructor that does nothing

virtual void applyBugSpray(); - if bugSpray is applied, then it sets the Zumi dead and then it does the calculations to see if the zumi will drop a goodie or not and then it will either put a goodie on the grid or not

virtual bool blocksPlayer() const{return false;} – Zumis do not block the player from walking on them

virtual bool colocationKillsPlayer() const{return true;} – if a player is on the spot with a zumi, it is dead

virtual bool isZumi() const {return true;} – the only place where this virtual function is changed, a zumi is in fact a zumi

virtual void setDirection(int direction = -1); - this function randomly sets the direction if not given a direction as input (basically if direction == -1).

int getDirection() const {return m\_currentDirection;} – returns the int value of the direction the zumi is facing

bool attemptMoveInDirection(); - this attempts to move the zumi in the direction that it is already facing. Should it not be able to move, it changes it direction and waits for the next opportunity to move

SIMPLE ZUMI:

SimpleZumi(StudentWorld\* world, int startX, int startY); - constructor that sets the private data members and gives it the correct imageID

~SimpleZumi() {} destructor that does nothing

virtual void doSomething(); - the do something function checks if the zumi is able to move (determined by every x ticks given by the level info). If it can move, then it calls the attemptMove function, otherwise it simply decrements the waiting time.

**COORD:**

Coord(int cc, int rr) : m\_c(cc), m\_r(rr) {} – constructor that creates the coord object

int c() const { return m\_c; } – returns the column (x value);

int r() const { return m\_r; } – returns the row (y value);

**COMPLEX ZUMI:**

ComplexZumi(StudentWorld\* world, int startX, int startY); - constructor that sets the private data members and gives it the correct imageID

~ComplexZumi() {} – destructor that does nothing

virtual void doSomething(); - the do something function checks if the zumi is able to move (determined by every x ticks given by the level info). If it can move, then it calls the attemptMove function, otherwise it simply decrements the waiting time. (I also implemented a way to go after the player but it was not working properly: basically it checks through the grid to see if there is a path to the player using queues and if it finds one and it is within the searchDistance, then the zumi will move in the direction necessary to get to that spot of the player).

virtual void applyBugSpray(); - same as the applyBugSpray for zumis except that it gives 400 more points for death (yielding 500 total points)

2.

My complex Zumi wasn’t able to function correctly. The program crashes whenever the player comes within searchDistance of the ComplexZumi so I simply treated it like a SimpleZumi right now.

3.

For one, if there are two Zumis within the range of a Bug Sprayer, they can both be killed.

4. A description of how you tested each of your classes (1-2 paragraphs per

class)

**StudentWorld**

To test StudentWorld, I simply ran the program over and over again, testing first that everything I wanted on the map showed up, then testing whether the object would move and whether the text on the screen would update every time. Eventually, I started to see whether the objects were interacting properly on the screen, like the bugsprayers and the bricks and that those were in fact making everything disappear. Then when the exit was designed, if the player stood on the exit, did that work? And also, do the actors disappear off the map only or do they actually get deleted as well was a problem I encountered while trying to debug everything. Also, getting all the functions to return the right object was another debugging step taken.

**Actor**

Actor was difficult to test because it was a base class, but basically if every sub class was able to create itself on the grid with the right image, I knew that the Actor class was working. Also, if when I called the other functions that tell me whether the player can move there and it told me what it should have, then I knew that the base class of Actor was working fine.

**Player**

Getting player to work was a hard one. Making sure the keys were working was tough because I had to try out all the keys over and over again and make sure that there was no glitch that would let the player walk through walls because it was checking the wrong grid spot or something. Then I had to make sure that it could only put down the 2 bugsprayers and that it would die if it stood on the bugspray. Also, I had to test each powerup individually on the player by creating one manually on the map and having the player “eat” it to test if it worked.

**Brick**

If the bricks showed up on the screen, I knew the base class was working perfectly fine.

**PermaBrick**

If the correct image showed, it was working. Also, I tested to make sure that there was no way the player could walk through them or that the bugSprayers could not destroy the PermaBricks on accident. Finaly I had to check that Zumis and sprays couldn’t go through these bricks no matter what.

**DestBrick**

The brick had to first show up. Then, I had to make sure, that normally nothing could pass through it, like a player or Zumi or anything. I had to make sure that nothing else could be built on it, so I manually put stuff on it, like the player, who died immediately. Then, I had to test that the bricks could be destroyed and would dissapeaer by breaking them with bugSpray. The final testing was making sure that when the powerup was on for the player, that the bricks would not kill him or block him from going through it

**Exit**

The exit had to start invisible and it was the only object to do so. Then, I had to make sure that if there Zumis it was invisible and if there were none, then it became visible. Also, if the player stepped on it when invisible, he should not pass the level. If the player stepped on it when it was visible though, then the level would be finished.

**TimedLifeActor**

The base class could not really be checked because it was hard to create an object of this type, but if the subclasses showed up with the right images and the correct length of life, then this was working for sure.

**BugSprayer**

The bugSprayer object was a pain to make work correctly. First I had to make sure that the objects actually showed up when I pressed the space button. Then, when it did show up, it couldn’t be placed on a brick or any object that you can’t put it on. Also, I had to make sure that Zumi’s couldn’t go through it but the player could. Finally, I had to make sure that when the time elapsed, it would release the Bug Spray in all the directions and that it would stop at bricks and not go through them.

**BugSpray**

The bugSpray testing involved manually putting bugSpray on certain objects and seeing what would happen when the game started. For example, if I put it on the player, he should die and same with the zumis and destroyable bricks. Also, the permanent bricks should not be affected by the spray at all and the other bugSprayer should be set off by the spray landing on the same spot as it. Most of this was tested by manual placement on the grid.

**Goodie**

The base class again could not really be tested, but if all the goodies showed up on the screen, then the base class was working.

**WalkThruGoodie**

I simply placed this goodie manually on the screen and made the player take the goodie to test it. If the player was able to walk thourhg walls for a short period of time, then this was a success.

**ExtraLifeGoodie**

I simply placed this goodie manually on the screen and made the player take the goodie to test it. If the player’s lives increased, then this was a success.

**IncreaseSprayersGoodie**

I simply placed this goodie manually on the screen and made the player take the goodie to test it. If the player was able to place more sprayers for a short period of time, then this was a success.

**Zumi**

The base class again could not really be tested, but if all the functions worked, then this was working. I had to test the movement function multiple times to make sure that it was really getting random numbers and moving in those directions.

**SimpleZumi**

SimpleZumi testing involed just watching the zumi run around. As long as it was going in a straight line and only turning when it hit a wall, then it was working properly. Also, when a zumi died, I had to make sure that there was a chance of dropping a goodie. For that, I simply used level one where there was a 100% chance of getting a goodie and checked if the goodie dropped properly.

**Coord**

This was simply for making the queue function work in the complexZumi.

**ComplexZumi**

For the complexZumi, most of the testing involved the same deal as the simpleZumi, because the full functionality of the complexZumi did not work in my project. For some reason whenever I would come within the search range of the complexZumi, the program would crash. Otherwise, I would have tested the program by placing myself in a straight line with the zumi and seeing if it came toward me and then I would try by putting a wall between us and seeing if the complexZumi turned in the direction of the path toward the player.