**CS 32 Project 3 Report**

**Margaret Capetz**

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 are able to sneeze, and each type of actor sneezes in a different way.”*

Public Member Functions:

Actor:

Actor(StudentWorld\* mg, int imageID, int startX, int startY, int dir = 0, int depth = 0, double size = 1.0);

Constructor constructs actor according to specification, using base class Graph Object constructor.

virtual void doSomething();

Not pure virtual because while all actors may do different things, some do the same thing: nothing.

This is defined in the base class and as virtual because all actors must doSomething() and can

override the default.

virtual void bonk();

Not pure virtual because some classes’ bonk() should do nothing. Those derived classes would just use this base class public function instead of overriding. This is defined in the base class and as virtual because all actors must bonk() and can override the default.

bool isAlive() const;

Const promise because it will not alter anything, it just returns a bool. Used to check the alive/dead status of each actor. All actors must keep track if they are alive or not with this by returning the m\_isAlive Boolean.

void setDead();

Sets the actor’s status to not alive. All actors must have the ability to set their status to dead.

StudentWorld\* getWorld() const;

Returns a pointer to the StudentWorld object. All actors have access to their world.

virtual bool isStatic();

Returns a boolean, true if the Actor is static, false otherwise. This is virtual because it can be overridden. Initially all actors are set to false, as in not static. However, obstacles are static and this is used to block movement.

virtual bool isEnemy();

Returns a Boolean, true if the Actor is an enemy, false otherwise. All actors are default not enemies, but the enemy class is an enemy. Necessary for Peach/Projectiles to see if they are overlapping enemies because they will attempt to damage enemies.

virtual bool isPlayer();

Returns a Boolean, true if the Actor is an player, false otherwise. All actors are default not player, but the Peach class is a player. Necessary for Enemies/Projectiles to see if they are overlapping Peach because they will attempt to damage Peach.

bool overlappingPeach() const;

Returns a boolean, true if the actor is overlapping peach, false otherwise. The actor accesses the player (peach) pointer by through its access to the student world pointer. Necessary for Enemies/Projectiles/Goodies/etc to see if they overlap with Peach.

bool reachedFlagOrMario();

Returns a Boolean, true if the player reached a flag or mario, false otherwise. If true, the function sets the actor to dead and increases the player’s score by 1000. Necessary for notifying the world that the level should be ended since Peach has reached a levelEnder.

Obstacle:

Obstacle(StudentWorld\* mg, int imageID, int startX, int startY);

Obstacle constructor takes in an imageID because both blocks and pipes are obstacles but have different imageIDs.

virtual bool isStatic();

Returns true since obstacles are static. Overrides base class implementation. Necessary because actors that return true for isStatic() may block movement, so for example Peach cannot pass over an obstacle.

Pipe:

Pipe(StudentWorld\* mg, int startX, int startY);

Constructor for pipe using base class obstacle constructor. (There is not doSomething() or bonk() because the Pipe class just uses the default functions from the base class that do nothing.)

Block:

Block(StudentWorld\* mg, int startX, int startY);

Constructor constructs block according to specification. All blocks start out with their holdGoodie Boolean set to false.

virtual void bonk();

Accesses the student world pointer to play the sound SOUND\_PLAYER\_BONK.

bool isHoldingGoodie() const;

Returns a Boolean, true if the block is holding a goodie, false otherwise. I made this to track whether or not the block can return a goodie or not, as a block can only hold 1 goodie, all subsequent bonks will not release another goodie.

void setIsHoldingGoodie(bool b);

Setter for the holdingGoodie Boolean. I made this so that the holdGoodie Boolean can be set to false if a block that holds a goodie has been bonked.

bool powerUp();

Returns a Boolean, true if the block is holding a goodie, false otherwise. If block is holding a goodie, then the sound powerup appears plays and the block no longer holds goodie. This function is used by all goodie blocks in their doSomething() function.

Star block:

starBlock(StudentWorld\* mg, int startX, int startY);

Constructor constructs star block using block constructor, but holdingGoodie is set to true initially.

virtual void bonk();

If base class powerup() returns true, adds a new star object to the world. This utilizes object oriented programming because the main code for the bonk function has been factored to the base class in the powerup function.

Flower block:

flowerBlock(StudentWorld\* mg, int startX, int startY);

Constructor constructs flower block using block constructor, but holdingGoodie is set to true initially.

virtual void bonk();

If base class powerup() returns true, adds a new flower object to the world.

Mushroom block:

mushroomBlock(StudentWorld\* mg, int startX, int startY);

Constructor constructs mushroom block using block constructor, but holdingGoodie is set to true initially.

virtual void bonk();

If base class powerup() returns true, adds a new mushroom object to the world.

Goodie:

Goodie(StudentWorld \*mg, int imageID, int startX, int startY);

Constructor constructs goodie using actor base class constructor.

bool powerUpPeach(int score);

If goodie overlaps peach, increase score, set player score, set goodie status to dead, play sound, and return. Else return false. Used in derived classes: Star, Mushroom, Flower.

void doGoodie();

Handles goodie movement according to spec. Used in all goodie derived classes. I implemented this to factor out the common things to the base class since since goodies all move in the same way.

Star:

Star(StudentWorld \*mg, int startX, int startY);

Constructor constructs star based on goodie constructor.

virtual void doSomething();

Virtual to override base class doSomething(). If powerUpPeach(), increase score by 100, set player’s star power, set star invincibility to 150 ticks, return. Else doGoodie() (move according to spec).

Mushroom:

Mushroom(StudentWorld \*mg, int startX, int startY);

Constructor constructs star based on goodie constructor.

virtual void doSomething();

Virtual to override base class doSomething(). If powerUpPeach(), increase score by 75, set player’s jump power to true, return. Else doGoodie() (move according to spec).

Flower:

Flower(StudentWorld \*mg, int startX, int startY);

Constructor constructs star based on goodie constructor.

virtual void doSomething();

Virtual to override base class doSomething(). If powerUpPeach(), increase score by 50, set player’s shoot power to true, return. Else doGoodie() (move according to spec).

Peach:

Peach(StudentWorld\* mg, int startX, int startY);

Constructor constructs Peach according to spec, with Actor constructor. Defaults health to 1, powers to false, invincibility to zero, remaining jump distance to zero, time remaining to fire to 0.

virtual void doSomething();

Handles cases including checking for invincibility, check overlap with other actors, check jumping, check falling, then process user key input to move left, right, jump, or fire.

bool isStarPower() const;

Returns Boolean, true if star power is on, false otherwise.

bool isShootPower() const;

Returns Boolean, true if shoot power is on, false otherwise.

bool isJumpPower() const;

Returns Boolean, true if jump power is on, false otherwise.

void setStarPower(bool b);

Setter for star power.

void setShootPower(bool b);

Setter for shoot power.

void setJumpPower(bool b);

Setter for jump power.

virtual bool isPlayer();

Overrides base class isPlayer() and returns true since Peach is a player.

virtual void bonk();

If invincible, immediately return. Otherwise decrement health, set temp invincibility to 10 because overlapping with enemy, set powers to false, either play sound hurt or die based on the number of hit points that peach has left.

void setHealth(int x);

Setter for m\_health.

void setTempInvincibility(int x);

Setter for temp\_invincibility.

void setStarInvincibility(int x);

Setter for star\_invincibility.

LevelEnder:

levelEnder(StudentWorld\* mg, int imageID, int startX, int startY);

Constructor using base class actor constructor. Takes in imageID because the different level enders (Mario, Flag) have different imageIDs.

virtual void endLevel() = 0;

Pure virtual because the derived classes must implement this themselves because they handle cases of ending a level uniquely.

Flag:

Flag(StudentWorld\* mg, int startX, int startY);

Constructor using base class levelender constructor, inputs Flag imageID.

virtual void endLevel();

Tells the world that the flag has been reached by calling reachedFlag() from the student world pointer. Fulfills the requirement of the base class since in the base class this is a pure virtual function.

Mario:

Mario(StudentWorld\* mg, int startX, int startY);

Constructor using base class levelender constructor, inputs Mario imageID.

virtual void endLevel();

Tells the world that the Mario has been reached by calling reachedMario() from the student world pointer. Fulfills the requirement of the base class since in the base class this is a pure virtual function.

Enemy:

Enemy(StudentWorld\* mg, int imageID, int startX, int startY);

Constructor using base class Actor constructor, also takes in imageID as a parameter because its derived classes have unique imageIDs. Direction is a random choice between 0 and 180 degrees.

Peach\* getPlayer() const;

Returns a pointer to a player by accessing it from the getWorld() function.

virtual void doEnemy();

Does common things that enemies (Goombas, Koopas) do: check if overlapping peach, check if moving off block, check if obstacle is blocking it, attempt to move. I implemented this to factor out common things to the base class for good object oriented programming style.

virtual bool isEnemy();

Overrides base class function and returns true.

virtual void enemyBonk();

Does common things that an enemy does when bonked: if the player has star power, play sound kick, then increase player score and set state to dead. I implemented this to factor out common things to the base class for good object oriented programming style.

void enemyOverlappingPeach();

If enemy is overlapping peach, bonk peach and return. All enemies have this functionality so it has been factored to the enemy base class.

Goomba:

Goomba(StudentWorld\* mg, int startX, int startY);

Constructor using base class Enemy constructor, passing in the Goomba imageID.

virtual void doSomething();

Calls base class doEnemy().

virtual void bonk();

Calls base class enemyBonk().

Koopa:

Koopa(StudentWorld\* mg, int startX, int startY);

Constructor using base class Enemy constructor, passing in the Koopa imageID.

virtual void doSomething();

Calls base class doEnemy().

virtual void bonk();

Calls base class enemyBonk(). Also introduces a new Shell actor to the world.

Piranha:

Piranha(StudentWorld\* mg, int startX, int startY);

Constructor using base class Enemy constructor, passing in the Piranha imageID.

virtual void doSomething();

Does not call the base class doEnemy(). Instead checks if overlapping peach by enemyOverlappingPeach(), checking if Peach is on the same row as Piranha, checking if Peach is near enough to fire, checking if firing delay allows firing.

virtual void bonk();

Calls base class enemyBonk().

Projectile:

Projectile(StudentWorld\* mg, int imageID, int startX, int startY, int dir);

Constructor using base class Actor constructor, takes in

void moveWithFalling();

Common movement for all projectiles (Shells, Fireballs): check if alive, try to move down two pixels if no obstacle is blocking it, then try to move left or right if there are no obstacles, if there is an obstacle in the direction then setDead(). I implemented this to factor out common things to the base class for good object oriented programming style since all projectiles move the same.

Shell:

Shell(StudentWorld\* mg, int startX, int startY, int dir);

Constructor using base class Projectile constructor, passing in Shell imageID.

virtual void doSomething();

If overlapping with an enemy, bonk the enemy, setDead(), return. Else moveWithFalling().

Peach Fireball:

PeachFireball(StudentWorld\* mg, int startX, int startY, int dir);

Constructor using base class Projectile constructor, passing in PeachFireball imageID.

virtual void doSomething();

If overlapping with an enemy, bonk the enemy and return. Else moveWithFalling().

Piranha Fireball:

PeachFireball(StudentWorld\* mg, int startX, int startY, int dir);

Constructor using base class Projectile constructor, passing in PiranhaFireball imageID.

virtual void doSomething();

If overlapping with Peach, bonk Peach, setDead(), and return. Else moveWithFalling().

StudentWorld:

StudentWorld(std::string assetPath);

Constructor using base class GameWorld constructor.

~StudentWorld();

Call cleanup() to empty the data structures that hold the actors.

virtual int init();

Set isFlagReached and isMarioReached to false. Attempt to load level using string streams to build the level file text name. Return error messages (GW\_LEVEL\_ERROR) if necessary. In a switch statement, populate the vector data structure of actors and Peach pointer with associated actors/Peach. Return GW\_CONTINUE\_GAME.

virtual int move();

If the player is alive, ask it to do something. For all actors that are alive, ask them to do something. Check if peach is not alive, check if player has reached flag/mario. Delete dead actors. Update the status at the top of the screen using string streams. Return GW\_CONTINUE\_GAME.

virtual void cleanUp();

Delete the m\_player pointer. Iterate through the m\_actors vector and delete all actor pointers. Clear the vector.

bool isBlockingObjectAt(int x, int y) const;

Iterates through m\_actors vector and returns true if an obstacle (Actor that isStatic() returns true) is overlapping at the x and y coordinates. I created this so the actors can move around on the grid without overlapping with obstacles that are supposed to block movement. This is defined in the StudentWorld class and may be accessed by all actors since they all have a StudentWorld pointer accessible by getWorld() in the Actor class. The check for overlap is defined in StudentWorld because StudentWorld holds the vector of actors.

Actor\* ActorBlockingObjectAt(double x, double y) const;

Iterates through m\_actors vector and returns an Actor pointer to the actor that overlaps at the x and y coordinates. If there is no actor that overlaps, returns nullptr. Does not check if m\_player overlaps. I created this function for actors to see if they were overlapping with others. I created this so the actors can move around on the grid and see if they are overlapping with other actors. This is defined in the StudentWorld class and may be accessed by all actors since they all have a StudentWorld pointer accessible by getWorld() in the Actor class. The check for overlap is defined in StudentWorld because StudentWorld holds the vector of actors.

Actor\* ActorBlockingObjectAtAND(double x, double y) const;

Iterates through m\_actors vector and returns an Actor pointer to the actor that overlaps at the x and y coordinates. If there is no actor that overlaps, returns nullptr. DOES check if m\_player overlaps. This function is different than the above function because it also checks for Peach (the player) to see if the player overlaps with the actor that calls the function. I created this so the actors can move around on the grid and see if they are overlapping with other actors AND the player. This is defined in the StudentWorld class and may be accessed by all actors since they all have a StudentWorld pointer accessible by getWorld() in the Actor class. The check for overlap is defined in StudentWorld because StudentWorld holds the vector of actors and the Peach pointer.

Peach\* getPlayer() const;

Returns the Peach pointer m\_player that is associated with the student world.

void addActor(Actor\* a);

Takes in an actor pointer and pushes it to the back of the m\_actors vector.

void reachedFlag();

Sets isFlagReached to true to indicate that the player has reached the flag to end the level.

void reachedMario();

Sets isMarioReached to true to indicate that the player has reached the flag to end the level.

2. *A list of all functionality that you failed to finish as well as known bugs in your classes, e.g. “I didn’t implement the Flower class.” or “My koopa doesn’t work correctly yet so I treat it like a goomba right now.”*

N/A

3. *A list of other design decisions and assumptions you made; e.g., “It was not specified what to do in situation X, so this is what I decided to do.”*

Inheritance Structure:

Actor

Peach

Enemy

Goomba

Koopa

Piranha

Projectile

Shell

PeachFireball

Piranha

Obstacle

Pipe

Block

FlowerBlock

StarBlock

MushroomBlock

Goodie

Star

Flower

Mushroom

LevelEnder

Mario

Flag

Assumptions:

Projectile movement: In the sample game, the shells’ movement on the first level is inconsistent. Sometimes the shell passes over the hole in the center, sometimes it dies when it hits the hole it dies.

Two different invincibility’s: In the spec, both temporary and star invincibility are mentioned. So I created two separate Booleans to track whether or not the invincibility is on or off as well as two separate ints to track the time (ticks) left for each invincibility. The Booleans are set to false and the ints are set to 0 to initialize.