CS32 Project 3 Report: Super Peach Sisters

By Arsh Malik

**Description of public member functions**

1. Actor.h

**Actor Class**

Actor(StudentWorld\* p, int x, int y, int ID, int startDir, int depth, double size, int healthStatus);

The constructor for the base class needs to be public for derived classes to use it as a part of their constructor. The constructor is never virtual (as told by professor Carey).

virtual ~Actor()

The destructor of a base class is always a public member function. It needs to be virtual.

virtual void doSomething() = 0;

This function defines what each actor needs to do in every tick of the game. Since every actor has different actions, I made the function pure virtual, ensuring that Actor is an Abstract Base Class (ABC).

virtual void bonk();

This function defines what an actor will do if it is bonked by another actor. Since every actor performed differently when bonked, I made this function virtual.

virtual void canBlock();

This function is used to inform if an actor blocks other objects or not. Since this was different for sub category of actors, I defined it as a virtual function.

virtual bool isDamageable();

This functions returns true if an Actor can be damaged in any way. Otherwise, it returns false. Since each object could be/ couldn’t be damaged, this function would be redefined by the derived classes. Hence I defined it virtual.

virtual void addShell();

I defined this function to support the shell functionality of the koopa actor. Since this function is redefined by a sub-class, I made it virtual.

int getHealth();

This function is used to get the health(hit points) of an actor. Since this isused for every actor and won’t be redefined by a sub-class, I made it non-virtual in the base class.

bool hitStatus();

This function returns true if the actor has been hit. Otherwise, it returns false. Since this isused for every actor and won’t be redefined by a sub-class, I made it non-virtual in the base class.

void setHealth(int x);

This function is used to set the health of each actor. Since this isused for every actor and won’t be redefined by a sub-class, I made it non-virtual in the base class.

void setHitStatus(bool x);

This function sets a boolean that tells if an object is hit or not. Since this isused for every actor and won’t be redefined by a sub-class, I made it non-virtual in the base class.

StudentWorld\* getPtrToWorld();

This is a very useful function as it connects the actors to the game world and helps call the functions implemented in StudentWorld.cpp as well as functions given to us that belong to the game world.

**Goodie (Sub-class of Actor)**

Goodie(StudentWorld\* s, int x, int y, int idnum);

This is the contructor of our base class that takes in parameters that our common to the different types of goodies. Constructors are not virtual.

void goodieMove(StudentWorld\* &ptrToWorld, int curX, int curY);

This function defines how a goodie object moves in the world. This functionality is common to all the goodies, so I didn’t make it virtual.

void bonk();

This function is being overwritten for the goodies compared to what we had for Actor as each goodie behaves in the same fashion. Hence, no need to make it virtual as it won’t be redefined.

***Flower (sub-class of goodie)***

Flower(StudentWorld\* s, int x, int y);

The flower constructor takes in a pointer to StudentWorld and its initial coordinates. It is used to construct the flower object. Constructors are not virtual.

virtual void doSomething();

This function defines the criteria for a flower object to behave in the world. It is marked virtual just for clarity of thought that it is redefining a base class member function.

***Mushroom (sub-class of goodie)***

Mushroom(StudentWorld\* s, int x, int y);

*same description as constructor for Flower goodie.*

virtual void doSomething();

*same description as doSomething for Flower goodie.*

***Star (sub-class of goodie)***

Star(StudentWorld\* s, int x, int y);

*same description as constructor for Flower goodie.*

virtual void doSomething();

*same description as doSomething for Flower goodie.*

**Projectile (sub class of Actor)**

Projectile(StudentWorld\* s, int x, int y, int idnum, int dir);

This constructor for a projectile accounts for the common details among different types of projectiles and is used by the constructors of the derived classes. Constructors aren’t virtual.

virtual void doSomething() = 0;

This function is required to be redefined for each type of projectile since each one acts in a slightly different manner.

It accounts for the

Defining it as pure virtual makes the class ABC and ensures non-definition a Projectile object.

virtual void bonk();

This function details the steps that a projectile should perform if it is bonked. When bonked, a projectile does nothing. It is marked virtual just for clarity of thought that it is redefining a base class member function.

void doSomethingCommon(StudentWorld\* &ptrToWorld, int curX, int curY);

This function ensures that we don’t re write the code that is common among the projectiles. For e.g. all projectiles have same motion in the world. Since this function won’t be redefined, I made it non-virtual.

***class piranhaFireball (derived from Projectile)***

piranhaFireball(StudentWorld\* s, int x, int y, int dir);

Used to construct the object. Constructors aren’t defined virtual.

virtual void doSomething();

Lays down the steps a piranha fireball takes when it interacts with peach. It also uses the super-class’s doSomethingCommon function for movement. It is marked virtual just for clarity of thought that it is redefining a base class member function.

***class Shell (derived from Projectile)***

Shell(StudentWorld\* s, int x, int y, int peachDir);

Used to construct the object. Constructors aren’t defined virtual.

virtual void doSomething()

Lays down the steps a shell takes when it interacts with koopas and goombas. It also uses the super-class’s doSomethingCommon function for movement. It is marked virtual just for clarity of thought that it is redefining a base class member function.

***class fireball(derived from Projectile)***

Fireball(StudentWorld\* s, int x, int y, int peachDir);

Used to construct the object. Constructors aren’t defined.

virtual void doSomething();

Lays down the steps a peach fireball takes when it interacts with koopas and goombas (killing them). It also uses the super-class’s doSomethingCommon function for movement. It is marked virtual just for clarity of thought that it is redefining a base class member function.

**Enemy Class (derived from Actor)**

Enemy(StudentWorld\* s, int x, int y, int idnum, int d, double size);

This constructor for an enemy accounts for the common details among different types of enemies and is used by the constructors of the derived classes. Constructors aren’t virtual.

void moveEnemy();

This member function defines the movement of an enemy in the game world. Since all enemies (except Piranha) have the same movement, it is defined as a super-class member function. It is not redefined in the subclasses, hence it is not required to be virtual.

bool checkPeachOverlap();

This function checks the common behaviour of an enemy overlapping with a peach object in the game world. It is not redefined in the subclasses, hence it is not required to be virtual.

virtual void doSomething() = 0;

Since each enemy behaves slightly different from the other, this method always needs to be redefined. I made this function pure virtual, also ensuring it’s an ABC.

bool canBlock();

This function tells it user that an enemy doesn’t block other objects. This behaviour is common among all the enemies, hence it is in base class and not defined as virtual.

virtual bool isDamageable();

This function returns true as enemies are damageable. It is marked virtual just for clarity of thought that it is redefining a base class member function.

virtual void addShell();

This function will be redefined for the sub classes of enemies as each enemy doesn’t produce a shell. Hence, it is defined virtual. It adds shell to our Actor’s list and our game world.

***class Piranha (derived from enemy)***

Piranha(StudentWorld\* s, int x, int y);

This function creates a Piranha object. Constructors aren’t virtual.

virtual void doSomething();

This function details the steps that a piranha performs when it is interacting with peach and otherwise. It is marked virtual just for clarity of thought that it is redefining a base class member function.

***class Goomba (derived from enemy)***

Goomba(StudentWorld\* s, int x, int y);

This function creates a Goomba object. Constructors aren’t

made virtual.

virtual void doSomething();

This function instructs a goomba object to move around in the world and bonk the peach if it comes in contact. It is marked virtual just for clarity of thought that it is redefining a base class member function.

***class Koopa (derived from enemy)***

Koopa(StudentWorld\* s, int x, int y);

This function creates a Koopa object. Constructors aren’t

made virtual.

Both functions below are marked virtual just for clarity of thought that they are redefining base class member functions.

virtual void doSomething();

This function describes the motion of a koopa during a tick of the game. If it encounters the peach, it bonks it. If it is hit by peach fireball, then turns into a shell.

virtual void addShell();

This functions helps us add a shell to the game.

***Class LevelEnder (derived from Actor)***

LevelEnder(StudentWorld\* s, int x, int y, int idnum);

Constructs a level ender object, that is a flag or mario, based on common details. Constructors aren’t virtual.

virtual void doSomething() = 0;

This function lays down the steps a flag or mario performs when interacting with peach. Needs to be pure virtual to make the class ABC and for redefinition.

Both functions below are marked virtual just for clarity of thought that they are redefining base class member functions.

virtual bool canBlock();

Returns false as level ender objects don’t block other objects.

virtual void bonk();

Flags and mario can’t be bonked.

bool possiblePeachOverlap();

Checks for peach overlap with flag or mario. Common behaviour. Therefore added to the superclass and not made virtual.

***class Mario (derived from level ender)***

Mario(StudentWorld\*s , int x, int y);

Constructs a mario object in the game. Constructor not made virtual.

virtual void doSomething();

If mario overlaps with peach, then it informs the game world that the game has been won. Described as virtual for clarity purposes.

***class Flag (derived from level ender)***

Flag(StudentWorld\*s , int x, int y);

Constructs a flag object in the game. Constructor not made virtual.

virtual void doSomething();

If flag overlaps with peach, then it informs the game world that the level has been completed. Described as virtual for clarity purposes.

**Class Object (derived from Actor)**

Object(StudentWorld\* s, int x, int y, int idnum, int d, double size);

Constructs a pipe or a block based on common details. Defined not virtual as it is a constructor.

virtual void doSomething() = 0;

Lays down the steps a pipe or block performs in the game. Needs to be pure virtual to make the class ABC and needs to be redefined for specificity.

virtual void bonk() = 0;

Defines the steps performed when an object is bonked. Marked as pure virtual in order to enforce redefinition.

bool cannotBeDamaged();

Returns true as pipe and block cannot be damaged. As it is a common behaviour, function not made virtual.

virtual bool canBlock();

returns true as both the objects block. As it is a common behaviour, function not made virtual.

***class Block (derived from Object)***

Block(StudentWorld\* s, int x, int y, int goodieType);

Constructor for the block. Therefore, not marked virtual.

virtual void doSomething();

Does nothing. Marked virtual for clarity.

virtual void bonk();

Makes a sound when bonked. Marked virtual for clarity.

***class Pipe (derived from object)***

(same functionality as block function)

Pipe(StudentWorld\* s, int x, int y);

virtual void doSomething();

virtual void bonk();

**Class Peach (derived from Actor)**

Peach(StudentWorld\* s, int x, int y);

Constructs a peach object for the game world. Not marked as virtual as constructors aren’t virtual.

virtual void doSomething();

This is the main function that defines the movement of the peach in the game with reference to user inputs and the steps the peach takes when it interacts with other objects. Made virtual just for clarity.

virtual void bonk();

When the peach is bonked, it’s hit points are decreased and it makes a sound of being hurt. It can also die if the hit point is 1 and is bonked. Made virtual just for clarity.

Below functions are not required to be virtual because there would be no redefinition.

void getCurrentCoord(int &x, int &y);

This function gets the current coordinates of peach.

bool fireballPowerStatus();

int getRechargeTime();

returns the recharge time for peach

bool getStarPowerStatus();

returns the status of star power

bool getJumpPowerStatus();

returns the status of jump power

bool getShootPowerStatus();

returns the status of shoot power

void setStarPowerTime(int time);

Sets the star power as per the parameter.

void setRechargeTime(int t);

Sets the recharge time as per the parameter.

void setDead();

Marks peach as dead.

void setShootingPower(bool b);

Sets the shooting power as per the parameter.

void setJumpPower(bool b);

Sets the Jump power as per the parameter.

void setStarPowerStatus(bool b);

Sets the star power status as per the parameter.

1. StudentWorld.h

**StudentWorld Class**

Following functions aren’t made virtual as the implementations are specific to StudentWorld.

StudentWorld(std::string assetPath);

Constructs the game world. Constructers aren’t virtual.

~StudentWorld();

Destructor for our game world. Calls cleanup to clear dynamically allocated memory.

void addActor(Actor\* p);

This function is used to add a new actor to our Actor’s list.

bool isBlockingObject(int x, int y, Actor\* &p);

This function checks for object overlap with an object that blocks other objects in the game.

bool peachOverlap(int x, int y, Actor\* &p);

This function checks for object overlap with peach.

bool koopagoombaoverlap(int x, int y, Actor\* &p);

This function checks for object overlap with a koopa or goomba.

Peach\* getPeach();

This function returns a a ptr to peach object.

void changeLevelStatus(bool x);

Changes the level status if peach reaches the flag.

void changeGameStatus(bool x);

changes the game status as peach reaches Mario.

The definitions for the below functions were provided to us.

They are defined virtual for readability. They’re defined as virtual in the gameWorld class from which StudentWorld class is derived.

virtual int init();

Initializes the game world by reading the level file and adds actors.

virtual int move();

Calls each actors’ doSomething method in a game tick and updates the game statistics.

virtual void cleanUp();

Iterates through the list and deletes dynamically created objects.

**Assumptions**

There was a case where the shell doesn’t fall in the 1 block gap in level 1 in the sample game provided. However, the spec says that it should fall.

I implemented shell’s behavior as per the sample game provided.

*Thank You*