**Rohith Venkatesh**

**Project 3 Writeup**

**Part 1 – Explanation of all public functions:**

**The class hierarchy for actors is as follows:**

Actor

Peach

IdleActors

Block

Pipe

Finish

Flag

Mario

Projectile

Peach Fireball

Piranha Fireball

Shell

Enemy

Goomba

Koopa

Piranha

**StudentWorld Class**

* virtual int init()
  + This function is an implementation of the same function in the GameWorld class. The function loads in all the actors from the level text file using a for loop and a switch statement. The output stringstream is used to load in the file.
* virtual int move()
  + This function is an implementation of the same function in the GameWorld class. The function uses a for loop to make all the actors do something during the tick and frees up dynamically allocated actors if they are dead. The function also returns a const int variable that corresponds to whether the game has to continue, end, or
* virtual void cleanup()
  + This function frees all dynamically allocated data, clears the vector of actor pointers, and resets all Boolean varables
* void addActor(Actor\* newActor)
  + This function adds a new actor to the vector of actors in the Student World class. This function was necessary because some actions of actors in “Actors.H” and “Actors.cpp” cause a new actor to be added to the game
* void setLevelComplete()
  + This function sets the Boolean variable “completedLevel” to true. This indicates that the current level was completed so the move() function can return the proper value. I chose to inline this function since it is a trivial function.
* void setGameComplete()
  + This function sets the Boolean variable “completedGame” to true. This indicates that the current level was completed so the move() function can return the proper value. I chose to inline this function since it is a trivial function.
* bool overlaps(double x, double y, Actor\* stationary)
  + This function checks to see if the inputted x and y coordinates overlap with the actor that the stationary parameter points to. The function returns true if there is an overlap and false otherwise.
* bool objectBlocking(double x, double y, Actor\* moving, Actor\*& blocking)
  + This function checks to see if the inputted x and y coordinates overlap with any actor in the game apart from the actor that the moving parameter points to. The function returns true if there is an actor blocking movement to (x,y) and false otherwise. If there is an actor in the way, the parameter pointer reference, blocking, is set to point to the actor in the way.
* Peach\* gamePeach()
  + Method returns a pointer to the peach object in the game

**Actor (base) class**

* virtual void doSomething() = 0
  + Each actor that is derived from the Actor (base) class has to do something on each tick. Thus, each Actor will have its own doSomething() function. For this reason, I decided to make the function pure virtual.
* virtual void bonk(Actor\* bonker) = 0
  + This function specifies what the actor must do when it is bonked. It is declared pure virtual since all actors do something when it is bonked and it is meant to force the programmer to implement the proper implementation of the bonk method for the class. The bonker parameter specifies the Actor that is bonking the object.
* virtual void damage()
  + The damage() function damages the actor that the function is called by. I decided to make this function non-pure virtual since most of the actors don’t do anything when they are damaged (only 4 actors are damageable). This made the implementation of the other classes easier as repetition was prevented
* virtual bool isPeach()
  + This is a non-pure virtual identifier function that returns false for all actors but Peach and only the Peach class will return true for the function
* Virtual bool canBlockActors()
  + This is a non-pure identifier function that returns false for all actors but peach, but the IdleActors class (Pipe and Block) for which the function returns true to indicate that they can block actors. This is important when deciding whether an object can move to a location
* virtual bool canBeDamaged()
  + This is a non-pure identifier function that returns false for all actors but peach, but the enemy classes (Koopa, Goomba, Piranha), for which the function returns true to indicate that they can be damaged. This is important when deciding whether an object should be damaged
* virtual bool isProjectile()
  + This is a non-pure identifier function that returns false for all actors but peach, but the Projectile classes (Peach Fireball, Piranha Fireball, Shell), for which the function returns true to indicate that they are Projectiles.
* bool isAlive()
  + This is a non-virtual accessor function that returns the value of the Boolean data variable specifying if the actor is alive or not
* StudentWorld\* getWorld()
  + This is an accessor function that returns the StudentWorld object that the Actor class points to
* void SetDead()
  + This is a non-virtual mutator function that sets the Boolean data variable specifying if the actor is alive to false and it is used when an actor dies
* void move(int pix, bool down)
  + This is a non-virtual function that moves an actor a specified number of pixels sideways and down (if specified by the bool parameter “down”. I created this function because code was duplicated in the doSomething() functions of the Projectile and Enemy classes so this function consolidates the code.

**Peach Class (Derived from Actor Class)**

* virtual void doSomething() = 0
  + This function specifies what Peach will do during a tick. This function implements the pure virtual function of the same name in the Actor class
* virtual void bonk(Actor\* bonker) = 0
  + This function specifies what Peach will do when it is bonked. This function implements the pure virtual function of the same name in the Actor class
* virtual void damage()
  + The damage() function overrides the damage function in the Actor class. The function calls Peach’s bonk function
* virtual bool isPeach()
  + This is a non-pure virtual identifier function that returns true to indicate actor is peach
* Virtual bool canBeDamaged()
  + This is a non-pure virtual identifier function that returns true to indicate actor can be damaged
* void setHitpts(int pts)
  + Mutator function that sets Peach’s data member that records its remaining hit points to the value specified
* void setStarTicks(int ticks)
  + Mutator function that sets Peach’s data member that records its remaining star power ticks to the value specified
* void giveShootPower()
  + Mutator function that gives Peach Shooting Power by setting a bool data member that states whether Peach has shooting power to true
* void giveJumpPower()
  + Mutator function that gives Peach Shooting Power by setting a bool data member that states whether Peach has jump power to true
* bool hasStarPower()
  + Accessor function that returns true if Peach has Star Power (Peach’s remaining star ticks are greater than 0)
* bool hasShootPower()
  + Accessor function that returns true if Peach has Shoot Power
* bool hasJumpPower()
  + Accessor function that returns true if Peach has Jump Power

**IdleActors Class (Derived from Actor Class)**

* virtual void doSomething()
  + This is a virtual function derived from the Actor class. The IdleActors (Pipe and Block) will do nothing so the doSomething() function is empty.
* virtual void bonk(Actor\* bonker)
  + This is a pure virtual function derived from the Actor class. The function is also declared pure virtual in the IdleActors class to force the programmer to program unique bonk functions for Pipes and Blocks
* virtual bool canBlockActors()
  + Function returns true to indicate that IdleActors can block actors from moving to their location. This function overrides the canBlockActors() function in the base class to indicate that The IdleActors Class and all classes derived from it can block actors

**Pipe Class (Derived from IdleActors Class)**

* virtual void bonk(Actor\* bonker)
  + Function overrides the virtual function of the same declaration in the IdleActors class (declared non-pure virtual). Function has an empty body as a Pipe does nothing when it is bonked.

**Block Class (Derived from IdleActors Class)**

* virtual void bonk(Actor\* bonker)
  + Function overrides the virtual function of the same declaration in the IdleActors class (declared non-pure virtual). The bonk function for the Block class has a unique implementation as the Block has to bonk out a goodie if it is a goodie-loaded block

**Finish Class (Derived from Actor Class)**

* virtual void doSomething()
  + Function overrides virtual function of same name in the Actor class. The function will increase the player’s score by 1000 if Peach reaches the Finish (Flag or Mario), remove the Finish object from the game, and either end the game or the level depending on the call of the pure virtual function complete() (defined below)
* virtual void bonk(Actor\* bonker)
  + Function overrides the pure virtual function of the same name in the Actor class and does nothing as a Finish object (Flag or Mario) does nothing when bonked

**Flag Class (Derived from Flag Class)**

* virtual void complete()
  + Function overrides the pure-virtual function of the same name declared in the Finish Class. The function will set the level complete as a flag is the end of the current level.

**Mario Class**

* virtual void complete()
  + Function overrides the pure-virtual function of the same name declared in the Finish Class. The function will set the game complete as Mario is the end of the game.

**Goodie Class**

* virtual void doSomething()
  + The doSomething() function overrides the same pure-virtual function of the same name in the Actor class. The function will increase the Score by a specified point value based on the pointsAdded() function (defined below), give Peach a power based on the type of goodie, set the hit points to 2, play a powerup sound, and remove the goodie from the game IF the goodie overlaps with Peach. Otherwise, the goodie will try to move down and to the side.
* virtual void bonk(Actor\* bonker)
  + Function overrides the pure virtual function of the same name in the Actor class and does nothing as a Goodie (Flower, Mushroom, or Star) does nothing when bonked
* virtual int pointsAdded() = 0
  + A pure-virtual function that specifies the number of points that the Goodie adds to the score when Peach obtains the goodie. Each Goodie has a specified number of points that the programmer specifies so it made sense to make this function pure virtual.
* virtual void givePeachPower() = 0
  + A pure-virtual function that specifies the power that the Goodie gives Peach when Peach obtains the goodie. Each Goodie has a specified power that is given to peach so it made sense to make this function pure virtual.

**Flower Class**

* virtual int pointsAdded()
  + Function implements pure-virtual function of same name in Goodie class by specifying that 50 points have to be added to the score when Peach obtains the goodie
* Virtual void givePeachPower()
  + Function implements pure-virtual function of same name in Goodie class by specifying that Peach should be assigned the power to shoot fireballs

**Mushroom Class**

* virtual int pointsAdded()
  + Function implements pure-virtual function of same name in Goodie class by specifying that 75 points have to be added to the score when Peach obtains the goodie
* virtual void givePeachPower()
  + Function implements pure-virtual function of same name in Goodie class by specifying that Peach should be assigned the power to jump higher

**Star Class**

* virtual int pointsAdded()
  + Function implements pure-virtual function of same name in Goodie class by specifying that 100 points have to be added to the score when Peach obtains the goodie
* virtual void givePeachPower()
  + Function implements pure-virtual function of same name in Goodie class by specifying that Peach should be assigned invincibility powers

**Projectile Class**

* virtual void doSomething()
  + Implements doSomething() method to move projectile and strike target
* virtual void isProjectile()
  + Overrides non-pure virtual function of same name in Actor class to indicate that the Actor is a projectile (returns true)
* virtual bool conditionToDamage(Actor\* damagedObject)
  + Non-pure virtual function that returns true if the damagedObject is damageable and is not Peach. I chose to make this function non-pure virtual as the Peach Fireball and Shell have the same implementation for the function, while the Piranha Fireball has a different implementation.
* virtual void bonk(Actor\* bonker)
  + Implementation of pure-virtual function of same name in Actor class. Funtion does nothing as a projectile does nothing when bonked

**Piranha Fireball Class**

* virtual bool conditionToDamage(Actor\*damagedObject)
  + Overrides non-pure virtual function of same name in projectile class. It returns true if the damaged object is Peach. The function is used in the Projectile implementation of doSomething() to indicate if the projectile should damage an object

**Peach Fireball Class**

* CLASS HAS NO PUBLIC FUNCTIONS

**Shell Class**

* CLASS HAS NO PUBLIC FUNCTIONS

**Enemy Class**

* virtual void doSomething()
  + Implementation of pure-virtual function of same name in Actor class that tells the enemy object to kill peach if peach touches it and makes the enemy move sideways and turn as needed.
* virtual void bonk()
  + Implementation of pure-virtual function of same name in Actor class that tells the game to increase the score by 100 points and remove the enemy if Peach touches the object with star power
* virtual void damage()
  + Implementation of non-pure-virtual function of same name in Actor class that tells the game to increase the score by 100 points and remove the enemy if damaged and to create a new Shell object if the Enemy is a Koopa
* virtual void canBeDamaged()
  + Implementation of non-pure virtual function in Actor class that returns true to indicate that Enemy can be damaged
* virtual bool isKoopa()
  + Non-pure virtual function that returns false for all enemies but Koopa to indicate Koopa. Function is used in the damage() function to indicate if a Shell object should be created

**Goomba Class**

* NO PUBLIC FUNCTIONS (INHERITED FROM ENEMY CLASS)

**Koopa Class**

* virtual bool isKoopa()
  + Returns true to identify Koopa. This function is used in the implementation of the damage() function in the Enemy class to indicate if a new Shell object should be created

**Piranha Class**

* virtual void doSomething()
  + Overrides doSomething() implementation in Enemy class. I decided to override the Enemy class doSomething() since Koopa and Goomba have the same implementation, while the doSomething() implementation of a Piranha object is different. Animates Piranha and fires fireballs when necessary

**Part 2 – Explanation of behavior that was not implemented:**

I was able to implement all the behaviors of the actors with no known bugs.

**Part 3 – Design Explanation List:**

I chose to use a vector of Actor pointers to store all the actors dynamically. I used a vector of actor pointers instead of Actor objects because the Actor class was declared as abstract so pointers were needed. I felt a vector was the best STL container for the Actor pointers since I didn’t know the exact number of actors that would be in the game. Additionally, vectors are the easiest to search, remove from, and add to. I also created a private Peach pointer variable in the StudentWorld class that points to the Peach object in the vector of actors. This made it easier to alter the Peach object and check for Peach’s attributes.

The specification specified that the initial direction of all Enemies was random (either 0 or 180) to choose a random direction, I created a private function called “random” in the StudentWorld classes to choose a random direction between 0 and 180.

I used output stringstreams to load both the file name and to output the text that specifies the number of lives, the score, and the powers Peach has at the top of the game screen. I felt that using an output stringstream was the best way to do this as the values of the level file and at the top of the game screen fluctuate thorough the game.

I made the Actor, IdleActors, and Finish classes abstract as there were functions that each of the base classes had to implement independently. I did not make the Projectile and Enemy classes abstract as there were no functions that made sense to be pure virtual.