**Actor Class**

**Actor(StudentWorld\* w, int imageID, int initialX, int initialY, int initialDirection, int depth, double size);**

Constructor

**bool isAlive();**

Returns if the actor is alive or not and tells the system if the actor needs to be deleted. This is put into the Actor class because all objects need to know if they are alive or not since that is how the system(StudentWorld) knows if the actor needs to be removed from the game and the screen for smooth gameplay. Not a virtual function since the functionality is the same for all objects.

**void setDead();**

Sets an actor as not alive. Since all actors are defaulted to alive, we only need to set it as dead for the system to know to delete the actor. Again this is put in Actor since all objects will need to be deleted and this allows the system to do this during each tick. Not a virtual function since the functionality is the same for all objects.

**StudentWorld\* getWorld();**

This will return a pointer to the StudentWorld class that is controlling and holding all objects and all the game control commands. It is useful for all objects to have access to the studentWorld and thus this is made in the Actor class. Not a virtual function since the functionality is the same for all objects.

**virtual bool getBlocking();**

Returns TRUE if the actor should block movement and FALSE if the actor does not. This function is very useful in order for all moving objects to see if it can keep moving despite overlapping with another actor. In Actor class since all actors will need to tell moving objects if they block movement or not. Virtual function since this will change for Obstacles but not pure virtual since most objects will return false for this function and it will be more code efficient to default this to false and change it for obstacles than to redefine as false for all other Actors.

**virtual bool isDamageable();**

Returns TRUE if the actor can be damaged and FALSE if not. Useful for projectiles to see if they should be trying to damage something and then disappearing or if they should be moving past overlapping objects that don’t matter. In Actor class since all actors will need to tell projectiles if they are damageable or not. Virtual function since this will change for different actors but not a pure virtual function since more objects are not damageable and thus this is defaulted to false and then changed for classes.

**virtual void bonk(Actor\* bonker) {}**

This will differ for different types of actors but what will happen is a number of actions will occur when two objects overlap and want to bonk each other. This function is to tell the object what to do when it is getting bonked. This function is in the Actor class since all large number of actors will want to have action when bonked. This is a virtual function since it will differ for different actors but is not pure virtual since there are still actors who will do nothing when bonked and thus defaulting this to nothing makes this an efficient way to take care of those instead of having to define it as 0.

**virtual void doSomething() = 0;**

This is what each object will do during each tick. It is in the Actor class since all actors will want to know what to do during the tick. This is a pure virtual function since every actor or each type of actor will do something different and thus making it a pure virtual function is much better.

**Obstacle Class**

**Obstacle(StudentWorld\* w, int imageID, int initialX, int initialY, int initialDirection, int depth, double size);**

Constructor

**bool getBlocking();**

Obstacles are meant to block movement and thus this function is redefined to overwrite the default false for most objects and returns TRUE for blocking movement so that the system knows that moving objects cannot move past or through this one. This was defined in the obstacle class since all obstacles(Pipes and Blocks) block movement and thus this will help reduce the amount of repetitive code. Not a virtual function since it is the same for both pipes and blocks.

**void doSomething() {}**

This is set to nothing since Blocks and Pipes do nothing on each tick, they just sit there. In Obstacle since same for both Blocks and Pipes and this is why it's not a virtual function either.

**Block Class**

**Block(StudentWorld\* w, int initialX, int initialY, int contains);**

Constructor

**void bonk(Actor\* bonker);**

Define the Block's bonk to deal with secret Blocks and Goodies. If the Block doesn’t contain anything, it will just play the SOUND\_PLAYER\_BONK sound. If it contains a goodie, depending on the goodie, it will create that above the Block and then play the SOUND\_POWERUP\_APPEARS sound. It will then set the status of the Block to a normal Block so that if it is bonked again it will only play the SOUND\_PLAYER\_BONK sound. This is unique to Block and thus is why its not a virtual function and only in the Block class.

**Pipe Class**

**Pipe(StudentWorld\* w, int initialX, int initialY);**

Constructor

**Peach Class**

**Peach(StudentWorld\* w, int initialX, int initialY);**

Constructor

**void doSomething();**

Move based on the input keys of UP, LEFT, or RIGHT and then shoot a fireball if the Peach has the shoot power. During this time, Peach will also decrement her times in terms of invincibility through star or temporary and will also be where the gravity is implemented to make her fall if there is nothing to block her downwards.

**bool isDamageable();**

Let the program know that Peach is damageable. In this class since need to overwrite the default false that is for most of the actors. Also why not virtual since it is changing the previous virtual.

**bool shootPowerStatus();**

Return TRUE if Peach has shoot power and FALSE if not. In Peach class because only relevant to Peach and not a virtual function since Peach doesn’t have any classes made from her.

**void updateShootPower(bool status);**

Change the status of the Shoot Power. Peach class because only relevant to Peach and not a virtual function since Peach doesn’t have any classes made from her.

**bool jumpPowerStatus();**

Return TRUE if Peach has jump power and FALSE if not. Peach class because only relevant to Peach and not a virtual function since Peach doesn’t have any classes made from her.

**void updateJumpPower(bool status);**

Change the status of the Jump Power. Peach class because only relevant to Peach and not a virtual function since Peach doesn’t have any classes made from her.

**bool starPowerStatus();**

Return TRUE if Peach has star power and FALSE if not. Peach class because only relevant to Peach and not a virtual function since Peach doesn’t have any classes made from her.

**void updateStarPower(int time);**

Add 150 ticks to Star Power if she gets the Star Power. Peach class because only relevant to Peach and not a virtual function since Peach doesn’t have any classes made from her.

**void bonk(Actor\* bonker);**

If Peach gets bonked by an enemy or an enemy projectile she will take damage. This entails a couple cases. If she has invincibility, nothing will happen, if she has a powerup such as Jump or Shoot power, she will just lose the power but won’t die. If she has nothing, Peach will die.

**PointAdders Class**

**PointAdders(StudentWorld\* w, int imageID, int initialX, int initialY, int initialDirection, int depth, double size, int pointsWorth);**

Constructor

**int getPoints();**

Return the number of points a PointAdder is worth. In the PointAdders class since all objects with points are in the pointAdders class and will need to be able to access their point worth. Not virtual since it is the same for all PointAdders.

**Goodies Class**

**Goodies(StudentWorld\* w, int imageID, int initialX, int initialY, int initialDirection, int depth, double size, int pointsWorth);**

Constructor

**virtual void powerUp() = 0;**

Tells Peach which type of goodie she ran into and what power up to give her. This is in the goodie class since only the three goodies(Star, Mushroom, Flower) need to deal with this functionalbility and it is a pure virtual function since each one will tell Peach something different and thus all of the three classes within goodie will need to change the function powerUp.

**void doSomething();**

Move left or right and switch directions when running into a blocking object. This function is placed here in the Goodie class since all three of the goodies, Mushroom, Star, and Flower will do the same thing during their tick which is to move and check if they are overlapping Peaches. Not a virtual function since it is the same for all subsequent classes.

**Flower Class**

**Flower(StudentWorld\* w, int initialX, int initialY);**

Constructor

**void powerUp();**

Give Peach the shooting power. Flower class because only relevant to flower and not a virtual function since no subsequent classes.

**Mushroom Class**

**Mushroom(StudentWorld\* w, int initialX, int initialY);**

Constructor

**void powerUp();**

Give Peach the jump power. Mushroom class because only relevant to mushroom and not a virtual function since no subsequent classes.

**Star Class**

**Star(StudentWorld\* w, int initialX, int initialY);**

Constructor

**void powerUp();**

Give Peach 10 seconds of star power. Star class because only relevant to star and not a virtual function since no subsequent classes.

**Portals Class**

**Portals(StudentWorld\* w, int imageID, int initialX, int initialY);**

Constructor

**virtual int statusChanger() = 0;**

Complete the level or the game depending on flag or Mario. In the Portal class since it is only relevant to flag or Mario which are subsequent classes to the portals class but is a pure virtual function since it is different for both flag and Mario and will be changed for both but is a function both needs.

**void doSomething();**

Check if it overlaps with Peach and change the status according to the function above. This is just to tell the flag or Mario to continuously check for Peach and if she has reached the end of the level and if so they must either indicate that the level is beat or that the game is beat. The reason this is in the Portals class is because it is the same functionality for both Flag and Mario but is not a virtual or a pure virtual function since it calls the statusChanger to change the specifics and thus is exactly the same for both Flag and Mario.

**Flag Class**

**Flag(StudentWorld\* w, int initialX, int initialY);**

Constructor

**int statusChanger();**

Change the statusChanger function to complete the level. In the Flag class because only the flag will complete the level and not a virtual function since no subsequent class.

**Mario Class**

**Mario(StudentWorld\* w, int initialX, int initialY);**

Constructor

**int statusChanger();**

Change the statusChanger function to complete the game. In the Mario class because only Mario will complete the game and not a virtual function since no subsequent class.

**Projectiles Class**

**Projectiles(StudentWorld\* w, int imageID, int initialX, int initialY, int initialDirection);**

Constructor

**virtual void damage();**

Damage damageable characters depending on if it's friendly or not. For Shells and Peach Fireballs, this will only damage enemies, for Piranha Fireballs, this will only damage Peach. Since Shells and Peach Fireballs both damage the same thing, this is defaulted for damaging enemies and thus is not a pure virtual because it is more code efficient to define it here for both rather than individually defining it for the other two. However, it is a virtual function since Piranha Fireballs will want to change this to damage peach. In the Projectiles Class since the projecticles are the only thing that wants to check what they are colliding with before trying to damage and removing itself from the game. (if it bonks anything that it doesn’t want to hit, without checking, it will just be removed from the game for no reason)

**void doSomething();**

This will move the projectile left or right until it hits an object that it wants to damage or a blocking object. The projectiles will also check for a blocking object below it and will proceed to fall if there isn’t anything under it to support the projectile. This is in the Projectiles class since all projectiles have the same movement and thus is not a virtual function since neither of the three will need to change this function.

**PeachFire Class**

**PeachFire(StudentWorld\* w, int initialX, int initialY, int initialDirection);**

Constructor

**Shell Class**

**Shell(StudentWorld\* w, int initialX, int initialY, int initialDirection);**

Constructor

**PiranhaFire Class**

**PiranhaFire(StudentWorld\* w, int initialX, int initialY, int initialDirection);**

Constructor

**void damage();**

Change what the projectile damages to only Peach. In the PiranhaFire class since it is the only object that will want to check if its bonking peach before bonk it and is not a virtual function since no subsequent class.

**Enemies Class**

**Enemies(StudentWorld\* w, int imageID, int initialX, int initialY);**

Constructor

**bool isDamageable();**

Enemies are damageable and thus change the base virtual Actor function to overwrite it and tell the system that enemies are damageable. In the Enemies class since all three of the enemies, Koopa, Goomba, and Piranha will want to be damageable and not a virtual function since neither of the three will want to change or overwrite this function.

**void bonk(Actor\* bonker);**

If an enemy is bonked by a peach with star power or a projectile such as a Shell or Peach Fireball, it will add the appropriate number of points and die. This is the same for all Enemies and thus is why it's not a virtual function and in the Enemies class.

**virtual void takeDamage();**

This is the function that actually deals with killing the actor. The reason for the function is to make it a virtual one so that I can just overwrite this to add functionality to Koopa to release a Shell on death in a way where I only rewrite this function rather than the whole bonk function from above. In the Enemies class since I can default this to what Gooma and Piranha want which is the same.

**virtual void doSomething();**

This function will differ for Piranhas since they don’t move back and forth, but since Goomas and Koopas have the same movement, I put this in the Enemies class so that I don’t have to rewrite it for both of them which is just repetitive code. What this function is defaulted to is the Goomba and Koopa movement which is back and forth until it hits a ledge or an Obstacle. A virtual function since Prianahs will need to change this since they don’t move.

**Goomba Class**

**Goomba(StudentWorld\*w, int initialX, int initialY);**

Constructor

**Koopa Class**

**Koopa(StudentWorld\*w, int initialX, int initialY);**

Constructor

**void takeDamage();**

Changes the takeDamage function a little to allow Koopa to release a Shell on death. This is in the Koopa class since this is unique to Koopa and not a virtual function since no subsequent class.

**Piranha Class**

**Piranha(StudentWorld\*w, int initialX, int initialY);**

Constructor

**void doSomething();**

Piranhas will change animations and then will shoot at Peach if she gets within a certain distance from them, and also changes directions to face Peach once she is the same y as them. This function is in the Piranha Class since it is unique to Piranha and is not a virtual function since there are no subsequent classes to Prianaha.

**StudentWorld Class**

**StudentWorld(std::string assetPath);**

Constructor

**~StudentWorld();**

Destuctor, calls the cleanUp() function

**bool overlap(int x, int y, Actor\* actor1, Actor\*& actor2);**

This function is used to check if an actor at a certain location is overlapping another actor. Passes in x and y coordinates to check for spaces in front or below, and passes actor1 to make sure that it doesn’t overlap itself when looping through the vector of all the actors. Actor2 is useful to see what it is colliding with and seeing if that object is important in collision or not. In the StudentWorld class since it is the class with the Actor vector and thus is the only class that is able to loop through and check for this. Not a virtual function since no subsequent class.

**void add(Actor\*);**

Adds an actor to the actors vector. This is in the StudentWorld class since it is the only class with access to the Actor vector and thus is the only class able to pushback another actor when needed. Not a virtual function since no subsequent class.

**void setStageCompleteStatus(int status);**

Set stage to complete if stage is complete or set the game is complete if the game is beat. THis is in the StudentWorld class since it is the class that is able to return the GWSTATUS\_FINISHED\_LEVEL or the GWSTATUS\_PLAYER\_WON commands. Not a virtual function since no subsequent class.

**bool isPeach(Actor\* actor);**

Check if an actor is Peach. This si in the StudentWorld class since it is the only class that holds the pointer to peach and thus the only class able to check if any actor is Peach. Not a virtual function since no subsequent class.

**void powerUpPeach(int power);**

Give Peach the appropriate power up depending on the input. If int power is 1, give peach the shoot power, if int power is 2, give peach the jump power, if int power is 3, give peach 150 ticks of star power. In the StudentWorld class since it is the only class with access to the peach actor directly and is able to call her functions in giving her the correct powerup. Not a virtual function since no subsequent class.

**bool starPeach();**

Check if Peach has star power. In the StudentWorld class since it is the only class with access to the peach actor directly and is able to call her functions in checking her powerup status. Not a virtual function since no subsequent class.

**int getPeachY();**

Get Peach’s y coordinate. In the StudentWorld class since it is the only class with access to the peach actor directly and is able to get her coordinates. Not a virtual function since no subsequent class.

**int getPeachX();**

Get Peach’s y coordinate. In the StudentWorld class since it is the only class with access to the peach actor directly and is able to get her coordinates. Not a virtual function since no subsequent class.

**virtual int init();**

Checks which level the player is on and will try to get the level file that matches the appropriate level. It will then load in all the actors that are specified in the level file or return a bad level file if there is no file for the requested level. In the StudentWorld class since it is the only class with access to the actors vector and thus is the only class that is allowed to add these actors into it to start the game.

**virtual int move();**

This function is in charge of telling all actors to their doSomething function whatever that entails. It is also in charge of checking if the player has finished the level or the game as well as deleting dead actors so that they are removed from the screen and don’t take up storage in the system. This function will also update the status text so that the player knows which level they are on, how many lives they have, which powers they have, as well as how many points they have. This is in the StudentWorld since it is the only class with access to the actors vector and thus is the only class that can make them call their do something function.

**virtual void cleanUp();**

This function is in charge of looping through all the actors in the actors vector and deleting everything that is in there so that there are no memory leaks and hta the game can be completely deleted and closed without issues. This is in the StudentWorld since it is the only class with access to the actors vector and thus is the only class that can loop through and delete all actors.

2. I believe that I finished implementing everything that was wanted of me.

3. Something I did that was not specified in the spec was to make it so that the score filled 6 spaces no matter what. Just like how the provided example did, I filled this space with 0 so that even if the player has 0 score, it will still display 000000.