Kelly Yu

Dr. Smallberg

2/27/22

Project 3 Report

Part 1:

|  |
| --- |
| **class** Actors : **public** GraphObject{  **public**:  Actors(StudentWorld\* stud, **int** imageID, **int** startX, **int** startY, **int** dir = 0, **int** depth = 1, **double** size = 1.0);  **virtual** ~Actors();  //virtual  **virtual** **void** doSomething()=0; //similarities: some check isAlive, some don't  **virtual** **void** bonk();  **virtual** **bool** doesBlock() **const**;  **virtual** **bool** isDamageable() **const**; //only Peach and enemies damageable  **virtual** **void** getDamaged(); //only Peach and enemies can get damaged    //regular  StudentWorld\* getWorld() **const**;  **bool** isAlive() **const**;  **void** setDead();  **int** getNewX(**int** steps) **const**; //returns new X after taking # of steps in current dir    **private**:  **bool** m\_isAlive;  StudentWorld\* m\_stud;  }; |

1. I placed Actors(…) as a regular function because only the Actor class needs the Actor constructor to initialize actors.
2. I placed ~Actors() as a virtual function because if any classes derive from Actors in the future and dynamically allocate memory, the compiler will call the right destructor to deallocate memory. Additionally, having a non-virtual base class destructor can cause undefined behavior.
3. I placed doSomething() as a pure virtual function. Since all actors have a doSomething() and they inherit indirectly from Actor, having doSomething() as a pure virtual function allows all actors or groups of actors define a unique version of doSomething() to do their specific tasks. This also blocks users from creating a general Actor object.
4. I placed bonk() as a virtual function. All actors should be able to be bonked by other things, but only Peach, blocks, and Enemies should react at all. The basest version of bonk() is empty, so all actors but Peach, blocks, and Enemies who inherit bonk() from Actor will correctly do nothing when bonked upon. Setting bonk() to be virtual will allow Peach, blocks, and Enemies to override the bonk() so that it allows them to react correctly when bonked upon.
5. I placed doesBlock() as a virtual function. All actors should be able to notify actors or other things that check if they are blocking actors or not. All actors but blocks and pipes are blocking actors. The basest version of doesBlock() returns false, so all actors but blocks and pipes who inherit doesBlock() from Actor will correctly indicate that they are not blocking objects. Setting doesBlock() to be virtual allows blocks and pipes to specialize the function and make it return true, to correctly indicate that blocks and pipes are blocking actors.
6. I placed isDamageable() as a virtual function. Similar to doesBlock(), all actors should be able to notify other things that check if they are damageable or not. All actors but Peach and Enemies are damageable Actor. The basest version of isDamageable() returns false, so all actors but Peach and Enemies who inherit isDamageable() from actors will correctly indicate that they are not damageable objects. Setting isDamageable() to be virtual allows Peach and Enemies to specialize the function and make it return true, to correctly indicate that they are damageable actors.
7. I placed getDamaged() as a virtual function. Similar to getDamaged(), all actors can be damaged, but only Peach and Enemies should react when others try to damage them. The basest version of getDamaged() is empty, so all actors but Peach and Enemies who inherit getDamaged() from Actor will correctly perform nothing when others try to damage them. Setting getDamaged() to be virtual allows Peach and Enemies to specialize the function and to react correctly when others damage them.
8. I placed getWorld() as a regular function since all actors may need and can use the same getter function to utilize the pointer to their StudentWorld object. Such uses include calling playSound() or interacting with other actors. All interactions are handled by the StudentWorld object.
9. I placed isAlive() as a regular function since all actors’ alive status will be checked when they are asked to do something during each tick.
10. I placed setDead() as a regular function since all actors may die during the game and they must change their status to dead upon dying.
11. I placed getNewX() as a regular function since the function does not need to be specific to any actor for the actor to use it correctly. Additionally, majority of actors (Goodies, Enemies, and Weapons) who inherit indirectly from Actor need to determine a new location to move based on their direction and their number of steps vary. This common function allows any actor who needs to compute new locations based on their directions and desired steps.

|  |
| --- |
| **class** Stationary : **public** Actor{  **public**:  Stationary(StudentWorld\* stud, **int** imageID, **int** startX, **int** startY);  **virtual** **bool** doesBlock() **const**;  **virtual** **void** doSomething();  };  **class** Block : **public** Stationary{  **public**:  Block(StudentWorld\* stud, **int** startX, **int** startY, std::string goodieType);  **virtual** **void** bonk();    **private**:  **bool** m\_hasGoodie;  std::string m\_goodieType;  };  **class** Pipe : **public** Stationary{  **public**:  Pipe(StudentWorld\* stud, **int** startX, **int** startY);  }; |

1. I placed Stationary(…) as a regular function because only the Stationary class needs the Stationary constructor to initialize Stationary objects.
2. I placed doesBlock() as a virtual function because I override Actor’s version of doesBlock(). Adding the virtual keyword is a convention. The overridden version of doesBlock() will return true. Since only blocks and pipes inherit from Stationary, and they are both blocking objects, overriding doesBlock() in Stationary makes doesBlock() useful for both blocks and pipes.
3. I placed doSomething() as a virtual function because I defined Actor’s version of doSomething(). Adding the virtual keyword is a convention. Since the only two actors who inherit from Stationary class are blocks and pipes, and the two actors both do nothing during each tick, the doSomething() is empty and can be a common function for both blocks and pipes to inherit from Stationary class.
4. I placed Block(…) as a regular function because only the Block class needs the Block constructor to initialize Block objects.
5. I placed bonk() as a virtual function because I override Actor’s version of bonk() and specialize it for Block objects. Adding the virtual keyword is a convention.
6. I placed Pipe(…) as a regular function because only the Pipe class needs the Pipe constructor to initialize Pipe objects.

|  |
| --- |
| **class** Goal : **public** Actor{  **public**:  Goal(StudentWorld\* stud, **int** imageID, **int** startX, **int** startY);  **virtual** **void** doSomething();  **virtual** **void** changeLvlStatus()=0;  };  **class** Flag : **public** Goal{  **public**:  Flag(StudentWorld\* stud, **int** startX, **int** startY);  **virtual** **void** changeLvlStatus();  };  **class** Mario : **public** Goal{  **public**:  Mario(StudentWorld\* stud, **int** startX, **int** startY);  **virtual** **void** changeLvlStatus();  }; |

1. I placed Goal(…) as a regular function because only the Goal class needs the Goal constructor to initialize Goal objects.
2. I placed doSomething() as a virtual function because I defined Actor’s version of doSomething(). Adding the virtual keyword is a convention. Since only Flags and Mario inherit from Goal, and they both do similar things during each tick, I defined doSomething() for both Flag and Mario in Goal.
3. I placed changeLvlStatus() as a pure virtual function because Flag and Mario will change the level status differently when Peach overlaps with them. Flag and Mario must uniquely define changeLvlStatus(). This also prevents users from creating a general Goal object.
4. I placed Flag(…) as a regular function because only the Flag class needs the Flag constructor to initialize Flag objects.
5. I placed changeLvlStatus() as a virtual function because Flag will specialize the function to indicate to StudentWorld that the level has ended.
6. I placed Mario(…) as a regular function because only the Mario class needs the Mario constructor to initialize Mario objects.
7. I placed changeLvlStatus() as a virtual function because Mario will specialize the function to indicate to StudentWorld that the game has been won.

----------------------------­

|  |
| --- |
| **class** MovingActors : **public** Actor{  **public**:  MovingActors(StudentWorld\* stud, **int** imageID, **int** startX, **int** startY, **int** dir, **int** depth, **double** size);  **void** reverseDirection(); //reverses direction of Actor  }; |

1. I placed Goal(…) as a regular function because only the Goal class needs the Goal constructor to initialize Goal objects.
2. I placed reverseDirection() as a regular function in MovingActors because Goodies, Weapons, and Enemies who indirectly inherit from MovingActors will potentially reverse direction upon some condition, such as when the actors reach the edge of a platform or reach a blocking actor. The function is regular as no actors need to specialize the function.

|  |
| --- |
| **class** Items : **public** MovingActors{  **public**:  Items(StudentWorld\* stud, **int** imageID, **int** startX, **int** startY, **int** dir, **int** depth, **double** size);  **virtual** **void** doSomething();  **virtual** **bool** passedInteractionCheck();  **virtual** **void** interactWithActor()=0;  **virtual** **void** reactToBlockingObject()=0;  }; |

1. I placed Items(…) as a regular function because only the Items class needs the Items constructor to initialize Items objects.
2. I placed doSomething() as a virtual function in Items because Goodies and Weapons who inherit from Items will perform actions that are similar and actions that structured in similar order.
3. I placed passedInteractionCheck() as a virtual function and defined it to return true as long as the actor who calls the function is overlapping Peach. Since Goodies and Piranha Fireball who inherit from Items require this check before the first part of their doSomething(), this function is common to Goodies and Piranha Fireball. Setting the function virtual allows Peach Fireball and Shell to specialize the function to check if they are overlapping Enemies rather than Peach.
4. I placed interactWithActor() as a pure virtual function. Goodies and Weapons interact in unique ways with unique actors they check if they are overlapping with. This requires interactWithActor() to be specialized in Goodies and Weapons.
5. I placed reactToBlockingObject() as a pure virtual function. When Goodies and Weapons run into blocking objects, they react to blocking objects in unique ways. This requires reactToBlockingObject() to be specialized in Goodie and Weapons.

|  |
| --- |
| **class** Goodies : **public** Items{  **public**:  Goodies(StudentWorld\* stud, **int** imageID, **int** startX, **int** startY);  **virtual** **void** interactWithActor();  **virtual** **void** reactToBlockingObject();  **virtual** **void** addPointsAndPower() **const** =0;  };  **class** Flower : **public** Goodies{  **public**:  Flower(StudentWorld\* stud, **int** startX, **int** startY);  **virtual** **void** addPointsAndPower() **const**;  };  **class** Mushroom : **public** Goodies{  **public**:  Mushroom(StudentWorld\* stud, **int** startX, **int** startY);  **virtual** **void** addPointsAndPower() **const**;  };  **class** Star : **public** Goodies{  **public**:  Star(StudentWorld\* stud, **int** startX, **int** startY);  **virtual** **void** addPointsAndPower() **const**;  }; |

1. I placed Goodies(…) as a regular function because only the Goodies class needs the Goodies constructor to initialize Goodies objects.
2. I placed interactWithActor() as a virtual function because Flower, Mushroom, and Star who inherit from Goodies all interact with the actor they overlap (Peach) in similar ways. I can specialize this function that can be common for all Goodies. Virtual keyword added due to convention.
3. I placed reactToBlockingObject() as a virtual function because Flower, Mushroom, and Star who inherit from Goodies all react blocking objects the same way. I can specialize this function that can be common for all Goodies. Virtual keyword added due to convention.
4. I placed addPointsAndPower() as a pure virtual function because Flower, Mushroom, and Star who inherit from Goodies increase the player’s score by different amounts and gives Peach different powers.
5. I placed Flower(…) as a regular function because only the Flower class needs the Flower constructor to initialize Flower objects.
6. I placed addPointsAndPower() as a virtual function because Flower must specialize how it increases the player’s score and what unique power to give Peach.
7. I placed Mushroom(…) as a regular function because only the Mushroom class needs the Mushroom constructor to initialize Mushroom objects.
8. I placed addPointsAndPower() as a virtual function because Mushroom must specialize how it increases the player’s score and what unique power to give Peach.
9. I placed Star(…) as a regular function because only the Star class needs the Star constructor to initialize Star objects.
10. I placed addPointsAndPower() as a virtual function because Star must specialize how it increases the player’s score and what unique power to give Peach.

|  |
| --- |
| **class** Weapons : **public** Items{  **public**:  Weapons(StudentWorld\* stud, **int** imageID, **int** startX, **int** startY, **int** dir);  **virtual** **void** reactToBlockingObject();  }; |

1. I placed Weapons(…) as a regular function because only the Weapons class needs the Weapons constructor to initialize Weapons objects.
2. I placed reactToBlockingObject() as a virtual function because Piranha Fireball, Peach Fireball, and Shell who inherit from Weapons all react the same way when they encounter a blocking object. I can specialize this function as a common function for Piranha Fireball, Peach Fireball, and Shell.

|  |
| --- |
| **class** PiranhaFireball : **public** Weapons{  **public**:  PiranhaFireball(StudentWorld\* stud, **int** startX, **int** startY, **int** dir);  **virtual** **void** interactWithActor();  }; |

1. I placed PiranhaFireball(…) as a regular function because only the PiranhaFireball class needs the PiranhaFireball constructor to initialize PiranhaFireball objects.
2. I placed interactWithActor() as a virtual function because unlike the other two weapons, only Piranha Fireball interacts with Peach. The function must be specialized and is only useful for Piranha Fireball. Virtual keyword added due to convention.

|  |
| --- |
| **class** PeachWeapons : **public** Weapons{  **public**:  PeachWeapons(StudentWorld\* stud, **int** imageID, **int** startX, **int** startY, **int** dir);  **virtual** **bool** passedInteractionCheck();  **virtual** **void** interactWithActor();  };  **class** PeachFireball : **public** PeachWeapons{  **public**:  PeachFireball(StudentWorld\* stud, **int** startX, **int** startY, **int** dir);  };  **class** Shell : **public** PeachWeapons{  **public**:  Shell(StudentWorld\* stud, **int** startX, **int** startY, **int** dir);  }; |

1. I placed PeachWeapons(…) as a regular function because only the PeachWeapons class needs the PeachWeapons constructor to initialize PeachWeapons objects.
2. I placed passedInteractionCheck() as a virtual function because unlike Goodies and Piranha Fireballs that inherit from Items, Peach Fireballs and Shells must check if they are overlapping with Enemies, not if they are overlapping with Peach.
3. I placed interactWithActor() as a virtual function because unlike Piranha Fireball, Peach Fireball and Shell interact with Enemies. The function can be specialized as a common function Peach Fireball and Shell. Virtual keyword added due to convention.
4. I placed PeachFireball(…) as a regular function because only the PeachFireball class needs the PeachFireball constructor to initialize PeachFireball objects.
5. I placed Shell(…) as a regular function because only the Shell class needs the Shell constructor to initialize Shell objects.

|  |
| --- |
| **class** Enemies : **public** MovingActors{  **public**:  Enemies(StudentWorld\* stud, **int** imageID, **int** startX, **int** startY, **int** dir);  **virtual** **bool** isDamageable() **const**;  **virtual** **void** getDamaged();  **virtual** **void** doSomething();  **virtual** **void** doDifferentThing1()=0;  **virtual** **void** doDifferentThing2()=0;  **virtual** **void** bonk(); //implement  }; |

1. I placed Enemies(…) as a regular function because only the Enemies class needs the Enemies constructor to initialize Enemies objects.
2. I placed isDamageable() as virtual function in Enemies because the function must be specialized so that Goomba, Koopa, and Piranha who inherit from Enemies all indicate that they are damageable.
3. I placed getDamaged() as virtual function in Enemies because the function must be specialized so that Goomba, Koopa, and Piranha who inherit from Enemies all react when they are damaged by Peach’s weapons. Since Goomba, Koopa, and Piranha react in exactly the same way, except that Koopa also adds a Shell, the function can be specialized and made common for Goomba, Koopa, and Piranha who inherit from Enemies.
4. I placed doSomething() as a pure virtual function because Goomba, Koopa, and Piranha who inherit from Enemies all do actions that are similar and actions that are structured in the same order. The function can be specialized and made common for Goomba, Koopa, and Piranha who inherit from Enemies.
5. I placed doDifferentThing1() as a pure virtual function because Goomba and Koopa, and Piranha who inherit from Enemies do something different in between their check if they are alive or not and if they should bonk Peach. Goomba and Koopa, and Piranha must specialize this function.
6. I placed doDifferentThing2() as a pure virtual function because Goomba and Koopa, and Piranha who inherit from Enemies do something different after they check if they should bonk Peach. Goomba and Koopa, and Piranha must specialize this function.
7. I placed bonk() as a virtual function because Goomba and Koopa, and Piranha who inherit from Enemies all react the same way when they are bonked, except that Koopa also adds a Shell. This function can be specialized and made common to Goomba and Koopa, and Piranha who inherit from Enemies.

|  |
| --- |
| **class** Piranha : **public** Enemies{  **public**:  Piranha(StudentWorld\* stud, **int** startX, **int** startY, **int** dir);  **virtual** **void** doDifferentThing1(); //implement  **virtual** **void** doDifferentThing2(); //implement  **private**:  **int** firing\_delay;  }; |

1. I placed Piranha(…) as a regular function because only the Piranha class needs the Piranha constructor to initialize Piranha objects.
2. I placed doDifferentThing1() as a virtual function because Piranha does something unique in between its check if it is alive or not and if it should bonk Peach. This function must be specialized for Piranha.
3. I placed doDifferentThing2() as a virtual function because Piranha does something unique after it checks if it should bonk Peach. This function must be specialized for Piranha.

|  |
| --- |
| **class** MovingEnemies : **public** Enemies{  **public**:  MovingEnemies(StudentWorld\* stud, **int** imageID, **int** startX, **int** startY, **int** dir);  **virtual** **void** doDifferentThing1();  **virtual** **void** doDifferentThing2(); //implement  };  **class** Goomba : **public** MovingEnemies{  **public**:  Goomba(StudentWorld\* stud, **int** startX, **int** startY, **int** dir);  };  **class** Koopa : **public** MovingEnemies{  **public**:  Koopa(StudentWorld\* stud, **int** startX, **int** startY, **int** dir);  **virtual** **void** getDamaged();  **virtual** **void** bonk();  }; |

1. I placed MovingEnemies(…) as a regular function because only the MovingEnemies class needs the MovingEnemies constructor to initialize MovingEnemies objects.
2. I placed doDifferentThing1() as a virtual function because Goomba and Koopa both do nothing in between its check if it is alive or not and if it should bonk Peach. This function can be specialized and made common for both Goomba and Koopa.
3. I placed doDifferentThing2() as a virtual function because Goomba and Koopa both do the same thing after they check if they should bonk Peach. This function can be specialized and made common for both Goomba and Koopa.
4. I placed Goomba(…) as a regular function because only the Goomba class needs the Goomba constructor to initialize Goomba objects.
5. I placed Koopa(…) as a regular function because only the Koopa class needs the Koopa constructor to initialize Koopa objects.
6. I placed getDamaged() as a virtual function because in addition to doing what all Enemies do when they get damaged, Koopas also add a Shell. This function extends Enemies’s base version of getDamaged().
7. I placed bonk() as a virtual function because in addition to doing what all Enemies do when they get bonked, Koopas also add a Shell. This function extends Enemies’s base version of bonk().

|  |
| --- |
| **class** Peach : **public** Actor{  **public**:  Peach(StudentWorld\* stud, **int** startX, **int** startY);    **virtual** **void** doSomething();  **virtual** **void** bonk();  **virtual** **void** getDamaged();  **virtual** **bool** isDamageable() **const**;  //getters; needed for status line in StudentWorld  **bool** hasShootPower() **const**;  **bool** hasJumpPower() **const**;  **bool** hasStarPower() **const**;  //setters  **void** setShootPower(**bool** set); //flower  **void** setJumpPower(**bool** set); //mushroom  **void** setStarPower(**bool** set); //star  **void** setHitPoints(**int** hp);  **void** setTicksLeftWithStarPower(**int** ticks);    **private**:  **int** m\_hitPoints;  **bool** m\_isTempInvincible; //temp invincibility  **bool** m\_hasShootPower;  **bool** m\_hasJumpPower;  **bool** m\_hasStarPower; //'permanant' invincibility  **int** ticks\_left\_with\_star\_power;  **int** ticks\_left\_with\_temp\_invincibility;  **double** remaining\_jump\_distance;  **int** time\_to\_recharge\_before\_next\_fire;  }; |

1. I placed Peach(…) as a regular function because only the Peach class needs the Peach constructor to initialize Peach objects.
2. I placed doSomething() as a virtual function because Peach does something unique compared to all actors during each tick of the game. This function must be specialized for Peach’s use.
3. I placed bonk() as a virtual function because Peach reacts in a unique way compared to all actors when Peach is bonked by Enemies. This function must be specialized for Peach’s use.
4. I placed getDamaged() as a virtual function because Peach reacts in a unique way compared to all actors when Peach is bonked by Enemies. Only Peach reacts the same way as it is bonked when Peach is damaged. This function must be specialized for Peach’s use.
5. I placed isDamageable() as a virtual function because Peach is damageable and must override Actor’s default version of the function that returns false. Since Peach only had minimal similarities with Enemies, I overrode this function the same way separately for Peach and Enemies.
6. I placed hasShootPower() as a regular function because this returns an indicator about whether or not Peach has the Shoot Power. This function is unique to Peach and not need be specialized.
7. I placed hasJumpPower() as a regular function because this returns an indicator about whether or not Peach has the Jump Power. This function is unique to Peach and not need be specialized.
8. I placed hasStarPower() as a regular function because this returns an indicator about whether or not Peach has the Star Power. This function is unique to Peach and not need be specialized.
9. I placed setShootPower() as a regular function because this lets others give and remove Peach’s Shoot Power. This function is unique to Peach and not need be specialized.
10. I placed setJumpPower() as a regular function because this lets others give and remove Peach’s Jump Power. This function is unique to Peach and not need be specialized.
11. I placed setStarPower() as a regular function because this lets others give and remove Peach’s Star Power. This function is unique to Peach and not need be specialized.
12. I placed setHitPoints() as a regular function because this lets others set Peach’s amount of hit points. This function is unique to Peach and not need be specialized.
13. I placed setTicksLeftWithStarPower() as a regular function because this lets others set how long Peach has the Star Power. This function is unique to Peach and not need be specialized.

|  |
| --- |
| **class** Actor;  **class** Peach;  **class** StudentWorld : **public** GameWorld{  **public**:  StudentWorld(std::string assetPath);  ~StudentWorld();  **virtual** **int** init();  **virtual** **int** move();  **virtual** **void** cleanUp();    **void** setBonkAtCoords(**double** x, **double** y);  **void** setDamageAtCoords(**double** x, **double** y);    **bool** isBlockingOrDamageableObjectAt(**double** tryX, **double** tryY, Actor\* a, **char** type); //overlap  **void** bonkBlockingObjectAt();  **void** bonkEnemies();  **bool** peachHasStarPower() **const**;  **bool** isOverlappingPeach(**double** x, **double** y) **const**; //overlap  **void** setLvlStatus(std::string s);  **void** addPower(std::string power);  **void** damagePeach() **const**;  **void** damageEnemies();  **void** addPeachFireball(**int** startX, **int** startY, **int** dir);  **void** addShell(Actor\* a);  **void** addGoodie(std::string goodieType, **int** startX, **int** startY);    **void** bonkPeach();  **bool** platformExists(**double** tryX, **double** tryY, Actor\* a) **const**;    **bool** isPeachWithinThisHeight(**double** y);  **bool** isPeachToLeft(**double** x);  **bool** isPeachToRight(**double** x);  **bool** isPeachWithinThisWidth(**double** x);  **void** addPiranhaFireball(**double** x, **double** y, **int** dir);  **private**:  std::list<Actor\*> actorContainer;  Peach\* m\_peach;  **double** bonkAtX, bonkAtY;  **double** damageAtX, damageAtY;  std::string m\_lvlStatus;  }; |

1. I placed StudentWorld(…) as a regular function because only the StudentWorld class needs the StudentWorld constructor to initialize StudentWorld objects.
2. I placed ~StudentWorld() as a regular function because only the StudentWorld class needs the StudentWorld destruct StudentWorld objects and deallocate any memory that StudentWorld objects dynamically allocated.
3. I placed init() as a virtual function because it defines the pure virtual function init() in GameWorld. It loads the level files and places all the objects in their correct spots at the start of each level.
4. I placed move() as a virtual function because it defines the pure virtual function move() in GameWorld. It asks all active actors to doSomething() during each tick, removes dead actors from the game, indicates to end the level or game if player finishes the level, loses the level, wins the game, or loses the game, and updates the status line.
5. I placed cleanUp() as a virtual function because it defines the pure virtual function cleanUp() in GameWorld. It frees all the memory that was used to dynamically allocate actors for each level.
6. I placed setBonkAtCoords(..) as a regular function because it allows me assign 2 member variables with the coordinates of the objects I want actors to bonk.
7. I placed setDamageAtCoords(..) as a regular function because it allows me assign 2 member variables with the coordinates of the objects I want actors to damage.
8. isBlockingOrDamageableObjectAt

Traverses through the actor container to see if an objects span the location that is passed into the function. If the object does block or is damageable, the function returns true and sets the corresponding member variables to the x,y coordinates of the object to to be damaged or bonked.

1. bonkBlockingObjectAt

Peach uses this to bonk the blocks.

1. bonkEnemies

Peach uses this to bonk Goombas, Koopas, and Piranhas.

1. peachHasStarPower

Returns true or false depending on whether or not Peach has the Star Power.

1. isOverlappingPeach

Actors other than peach pass in their coordinates or desired coordinates to see if they overlap with Peach.

1. setLvlStatus

This sets the string member variable to the correct level status so that Student World will know when to start or restart the level and when to end the level or end the game. This function is used by Flag and Mario.

1. addPower

Goodies use this function to indicate to the StudentWorld object to add particular powers to Peach.

1. damagePeach

Enemies use this function to indicate to the StudentWorld object to add damage Peach.

1. damageEnemies

Peach uses this function to damage alive Goombas, Koopas, and Piranhas that Peach overlaps. Only after Peach checks if it overlaps with Enemies does this function successfully damages Enemies.

1. addPeachFireball

Peach uses this function to add a Peach Fireball. The pointer to the Peach Fireball is stored in the actor container so that we can find the address again to delete the actor when it should be gone.

1. addShell

Koopa uses this function to add a Shell. The pointer to the Shell is stored in the actor container so that we can find the address again to delete the actor when it should be gone.

1. addGoodie

Blocks use this function to add Flower, Mushrooms, and Stars. The pointers to the actors are stored in the actor container so that we can find the address again to delete the actors when they should be gone.

1. bonkPeach

Enemies use this function to indicate to the StudentWorld object to add bonk Peach

1. platformExists

Enemies use this function to check if they should turn around to avoid moving off the edge of the platform partially or fully.

1. isPeachWithinThisHeight

Piranhas use this function to check if Peach is within 1.5\*SPRITE\_HEIGHT in the y-direction.

1. isPeachToLeft

Piranhas use this function to check if Peach is to the left of Piranhas so they know which way to face.

1. isPeachToRight

Piranhas use this function to check if Peach is to the right so they know which way to face.

1. isPeachWithinThisWidth

Piranhas use this function to check if Peach is within the right distance in the x-direction for them to shoot Piranha Fireballs at Peach.

1. addPiranhaFireball

Piranhas use this function to add a Piranha Fireball if Peach is within the right distance in the x-direction. The pointer to the Piranha Fireball is stored in the actor container so that we can find the address again to delete the actor when it should be gone.

Part 2:

N/A

Part 3:

I designed the function in a way that for classes do majority of the same thing will stem from the same base class. The class hierarchy did grow more abstract as I rooted more classes to the same derived classes, but this allowed me to repeat only a few trivial functionalities such as check if there’s a blocking object at the desired coordinates the actor wants to move to. If there is only 1 simple similarity among a bunch of complex differences, I left the classes a separated rather than stemming them from the same root. For instance, Peach and Enemies are both damageable. This similarity is so minor that if I were to stem Enemies and Peach from the same class who overrides the isDamageable() to return true, it would only complicate how I would prevent duplicate functionality between the Enemies and the Goodies and the Weapons who all share more similarities compared to the similarities shared between Enemies and Peach.

Below is a photo of my class hierarchy for all the Actors.

Diagram, schematic

Description automatically generated

Additionally, I left all interactions between the actors in StudentWorld, so that I could avoid misplacing and mis-accessing pointers within actors’ functions. I also only dynamically allocated actors in StudentWorld as actors who die but who dynamically allocate other actors will cause me to lose the address of the dynamically allocated actors, and make me unable to free up the memory.