Junho Choi

305329873

Project 3 Report

Hierarchy :

* Actor
  + BorderLine
  + OilSlick
  + HolyWaterProjectile
* Goodie
  + HealingGoodie
  + HolyWaterGoodie
  + SoulGoodie
* HitPointActor
  + Ghost Racer
  + MovingNPC
    - HumanPedestrian
    - ZombiePedestrian
    - ZombieCab

**Public Member Function Descriptions :**

**Actor :**

* Accessor Member Functions
  + bool isColAvoidWorthy() const
    - This function was defined in the Actor class because all derived class Actors are either collision avoidance worthy or not. It is not virtual because it is a simple getter function that returns the value of a private boolean data member in Actor. The reason why it is public is because the information is required by StudentWorld.
  + bool isAffectedByHolyWater() const
    - This function was defined in the Actor class because all derived class Actors are either affected or unaffected by the Holy Water Projectile. It is not virtual because it is a simple getter function that returns the value of a private boolean data member in Actor. The reason why it is public is because the information is required by StudentWorld.
  + virtual bool isAlive() const
    - This function was defined in the Actor class because derived classes need access to whether or not they are alive so they can respond accordingly. It is virtual because a derived class (HitPointActor) determines if an Actor is alive using a different process. The reason why it is public is because StudentWorld member functions require it in order to know when to erase the Actor off its list and free the dynamically allocated memory.
  + int getVertSpeed() const
    - This function was defined in the Actor class because many derived classes needed access to its vertical speed. It is not virtual because vertical speed works the same way in every single derived Actor class. The reason why it is public is because some StudentWorld member functions require it in order to initialize derived class Actors.
  + int gerHorizSpeed() const
    - This function was defined in the Actor class because many derived classes needed access to its horizontal speed. It is not virtual because horizontal speed works the same way in every single derived Actor class. The reason why it is public is because some StudentWorld member functions require it in order to initialize derived class Actors.
* Mutator Member Functions
  + virtual void hitByHolyWater()
    - This function was defined in the Actor class because every derived class must be able to use it. It was not made purely virtual because only certain derived class Actors actually do something particular when hit by water. In its implementation in the Actor class, it simply returns as that is what many derived class Actors do when it collides with a HolyWaterProjectile. Additionally, my implementation of checking for Actors that overlap with a HolyWaterProjectile and acting accordingly only calls hitByHolyWater if that specific derived class Actor is affected by it, which would make redefining hitByHolyWater for every derived class unnecessarily repetitive.
* Pure Virtual Functions
  + virtual void doSomething()
    - This function was defined in the Actor class because every derived class Actor “does something.” It is virtual because each Actor “did something” in a different way. There were commonalities among all derived base class doSomething implementations such as checking if they were alive (if not, the function would return) or how they moved. For the former, the lines of code that checked for the alive condition were repeated in an effort to make the Actor class an Abstract Base Class. For the latter, a separate protected method was made for more reusability.

**HitPointActor :**

* Accessor Member Functions
  + bool isAlive()
    - This function was redefined in the HitPointActor class because derived classes need access to whether or not they are alive so they can respond accordingly. It is not virtual because it does not have to be redefined further--all HitPointActor derived classes are able to know whether they are alive or not by the number of Hit Points it has left. The reason why it is public is because StudentWorld member functions require it in order to know when to erase the Actor off its list and free the dynamically allocated memory.
  + int getHealth() const
    - This function was defined in the HitPointActor class because derived classes need access to how much health they have left so they can respond accordingly. It is not virtual because it is a simple getter function that returns the value of a private int data member.
* Mutator Member Functions
  + void damageActor(int damageBy=1)
    - This function was defined in the HitPointActor class because all derived classes can get damaged in one way or another. It is not virtual because it is simply mutating a private int data member It is public because other non-derived functions may need to damage Actors.
  + void killActor()
    - This function was redefined in the HitPointActor class (it was a protected member function in the Actor class) because the way that HitPointActors are killed is different (based off of Hit Points). It is not virtual because all derived functions are killed in the same way, and does not to be further refined.

**GhostRacer :**

* Accessor Member Functions
  + int getSpraysLeft() const
    - This function was defined in the GhostRacer class because other functions needed to use it to get how many Holy Water sprays GhostRacer has left. It is not virtual because there are no GhostRacer derived classes, and therefore does not need to be redefined.
* Mutator Member Functions
  + void doSomething()
    - This function was redefined in the GhostRacer class because other functions need to call it in order to make GhostRacer do something. In this case, it needed to move based on its angle and read inputs from the user. It is not virtual because there are no GhostRacer derived classes, and therefore does not need to be redefined.
  + void moveActor()
    - This function was redefined in the GhostRacer class because the way that the GhostRacer moves is different from all other classes. It only moves horizontally (because it technically is not moving in the y-direction), and moves based off of the angle that it is facing. Using trigonometry, GhostRacer moves accordingly. It is not virtual because there are no GhostRacer derived classes, and therefore does not need to be redefined.
  + void spinCar()
    - This function was defined in the GhostRacer class in order to react accordingly to overlapping with OilSlicks. It generates a random value (within certain parameters of GhostRacer’s directional range) to spin GhostRacer by while it is overlapping an OilSlick. It is not virtual because there are no GhostRacer derived classes, and therefore does not need to be redefined.
  + void addHolyWater(int amt)
    - This function was defined in the GhostRacer class in order to react accordingly to overlapping with HolyWaterGoodies. It will refill GhostRacer’s holy water spray amount (which is a private int data member) by an amount equal to the argument. It is not virtual because there are no GhostRacer derived classes, and therefore does not need to be redefined.
  + void healActor(int amt)
    - This function was defined in the GhostRacer class in order to react accordingly to overlapping with HolyWaterGoodies. It will increase GhostRacer’s Hit Point amount (which is a private int data member in the HitPointActor class) by an amount equal to the argument. It utilizes a derived member function from the HitPointActor class to do this. It is not virtual because there are no GhostRacer derived classes, and therefore does not need to be redefined.

**BorderLine :**

* Mutator Member Functions
  + void doSomething()
    - This function was redefined in the GhostRacer class because other functions need to call it in order to make BorderLine do something. In this case, it is simply called the moveActor function which moves the BorderLine according to the BorderLine’s horizontal speed and GhostRacer’s horizontal speed. It also kills itself if it is no longer in the game frame. It is not virtual because there are no BorderLine derived classes, and therefore does not need to be redefined.

**HumanPedestrian :**

* Mutator Member Functions
  + virtual void doSomething()
    - This function was redefined in the GhostRacer class because other functions need to call it in order to make HumanPedestrian do something. In this case, it checks if it overlaps with GhostRacer (in this case the game is over), moves according to its own speed and Ghost Racer’s speed, and adjusts its horizontal speed randomly if the move plan distance calls for it. The move plan may also be adjusted randomly. It also kills itself if it is no longer in the game frame. It is not virtual because there are no HumanPedestrian derived classes, and therefore does not need to be further redefined.

**ZombiePedestrian :**

* Mutator Member Functions
  + virtual void doSomething()
    - This function was redefined in the ZombiePedestrian class because other functions need to call it in order to make HumanPedestrian do something. In this case, it checks if it overlaps with GhostRacer (in this case GhostRacer is damaged and ZombiePedestrian is killed), grunts if appropriate, attempts to follow GhostRacer if GhostRacer is in front of ZombiePedestrian and close, moves according to its own speed and GhostRacer’s speed, and adjusts its horizontal speed randomly (and angle accordingly) if the move plan distance calls for it. The move plan may also be adjusted randomly. It also kills itself if it is no longer in the game frame. It is not virtual because there are no ZombiePedestrian derived classes, and therefore does not need to be further redefined.

**ZombieCab :**

* Mutator Member Functions
  + void doSomething()
    - This function was redefined in the ZombieCab class because other functions need to call it in order to make ZombieCab do something. In this case, it checks if it overlaps with GhostRacer (in this case GhostRacer is damaged and ZombieCab is swerved off of the road), slows down or speeds up if it is faster or slower (respectively) than GhostRacer and there is an actor 96px in front of or behind (respectively) in the same lane, moves according to its own speed and GhostRacer’s speed, and adjusts its horizontal speed randomly if the move plan distance calls for it. The move plan may also be adjusted randomly. It also kills itself if it is no longer in the game frame. It is not virtual because there are no ZombieCab derived classes, and therefore does not need to be further redefined.

**OilSlick**

* Mutator Member Functions
  + void doSomething()
    - This function was redefined in the OilSlick class because other functions need to call it in order to make OilSlick do something. In this case, it moves according to its own speed and GhostRacer’s speed and it checks if it overlaps with GhostRacer and makes it spin if it does. It also kills itself if it is no longer in the game frame. It is not virtual because there are no OilSlick derived classes, and therefore does not need to be further redefined.

**HealingGoodie**

* Mutator Member Functions
  + void doSomething()
    - This function was redefined in the HealingGoodie class because other functions need to call it in order to make HealingGoodie do something. In this case, it moves according to its own speed and GhostRacer’s speed and it checks if it overlaps with GhostRacer and heals it by an appropriate amount if it does. It also self-destructs if it overlaps. It also kills itself if it is no longer in the game frame. It is not virtual because there are no HealingGoodie derived classes, and therefore does not need to be further redefined.

**HolyWaterGoodie**

* Mutator Member Functions
  + void doSomething()
    - This function was redefined in the HolyWaterGoodie class because other functions need to call it in order to make HealingGoodie do something. In this case, it moves according to its own speed and GhostRacer’s speed and it checks if it overlaps with GhostRacer and gives holy water to it by an appropriate amount if it does. It also self-destructs if it overlaps. It also kills itself if it is no longer in the game frame. It is not virtual because there are no HolyWaterGoodie derived classes, and therefore does not need to be further redefined.

**SoulGoodie**

* Mutator Member Functions
  + void doSomething()
    - This function was redefined in the HolyWaterGoodie class because other functions need to call it in order to make HealingGoodie do something. In this case, it moves according to its own speed and GhostRacer’s speed and it checks if it overlaps with GhostRacer and gives holy water to it by an appropriate amount if it does. It also self-destructs if it overlaps. It also kills itself if it is no longer in the game frame. It is not virtual because there are no HolyWaterGoodie derived classes, and therefore does not need to be further redefined.

**HolyWaterProjectile**

* Mutator Member Functions
  + void doSomething()
    - This function was redefined in the HolyWaterProjectile class because other functions need to call it in order to make HealingGoodie do something. In this case, it moves according to its own speed and direction, and it checks if it overlaps with all other Actors in the world. If it does, it will see if that Actor is affected by HolyWaterProjectile and react accordingly. Details on how it is able to figure it out is defined below as it is implemented in StudentWorld. It is not virtual because there are no HolyWaterProjectile derived classes, and therefore does not need to be further redefined.

**StudentWorld**

* Accessor Member Functions
  + GhostRacer\* getAisha() const
    - This function was defined in the StudentWorld class because functions in Actor or its derived classes (aside from GhostRacer) need it in order to communicate with GhostRacer. It is not virtual because there are no StudentWorld derived classes, and therefore does not need to be further redefined.
  + bool closeActorInSameLane(const Actor\* actor, int yDistance) const
    - This function was defined in the StudentWorld class because ZombieCab needs it in order to determine if/how to adjust its speed. It checks if there is an actor yDistance from the passed in Actor, with a negative yDistance representing another actor behind the passed in actor (i.g. yDistance is one-sided). It is not virtual because there are no StudentWorld derived classes, and therefore does not need to be further redefined. Pseudocode is below.

bool StudentWorld::closeActorInSameLane(const Actor\* actor, int yDistance) const{

for each Actor in the world (not including itself) :

if it is in the same lane :

if the Actor’s y coordinate is yDistance + passed in actor’s y coordinate :

if the above statement is true, return true.

return false.

}

* + bool overlapsWithAisha(const Actor\* actor) const
    - This function was defined in the StudentWorld class because multiple Actors (not including Aisha) must know if it overlaps with GhostRacer. It uses the algorithm that was given within the specifications. It is not virtual because there are no StudentWorld derived classes, and therefore does not need to be further redefined.
  + Actor\* getOverlappingActorAffectedByHolyWater(const Actor\* actor) const
    - This function was defined in the StudentWorld class because HolyWaterProjectile needs it in order to determine if it is overlapping with an Actor that is affected by it. It goes through every Actor (excluding itself) in the world, and checks if it is overlapping and if that Actor is affected by HolyWaterProjectile. If it is, a pointer to that Actor is returned. It is not virtual because there are no StudentWorld derived classes, and therefore does not need to be further redefined. Pseudocode is below.

Actor\* getOverlappingActorAffectedByHolyWater(const Actor\* actor) const{

for each Actor in the world (not including itself) :

if it is overlapping :

if it is affected by HolyWaterProjectile :

return pointer to that Actor

return nothing (nullptr)

}

* Mutator Member Functions
  + virtual int init()
    - This function was defined in the StudentWorld class because it is used to initialize the World when the game begins. It first dynamically allocates GhostRacer, and then the initial BorderLines. It then adds them to a single STL linked list, and returns a signal for the game to start. It is virtual because this is how it was given to us in the skeleton code--it does not need to be virtual because there are no derived classes.
  + virtual int move()
    - This function was defined in the StudentWorld class because it is called every tick to appropriate move Actors, remove Actors, add Actors, and update game texts. It also checks to make sure if the game is over because GhostRacer died, or if the user has finished the level by collecting enough SoulGoodies. It is virtual because this is how it was given to us in the skeleton code--it does not need to be virtual because there are no derived classes.
  + virtual void cleanUp()
    - This function was defined in the StudentWorld class because it is called after GhostRacer either loses a life, the game is over, or the user progressed to the next level. It is virtual because this is how it was given to us in the skeleton code--it does not need to be virtual because there are no derived classes. Pseudocode is below.

virtual void cleanUp(){

for each Actor in the STL list :

delete the Actor

clear the STL list

set the Aisha pointer to nullptr

set number of souls saved to none

reset bonus points to 5000

}

* + void addOilSlick(int xCoord, int yCoord)
    - This function was defined in the StudentWorld class because ZombieCab has a potential to create an OilSlick at the place it died if it died due to a HolyWaterProjectile. It will dynamically allocate an OilSlick object using the passed in arguments, and add it to the STL list. It is not virtual because there are no StudentWorld derived classes, and therefore does not need to be further redefined.
  + void addHolyWaterProjectile(int xCoord, int yCoord)
    - This function was defined in the StudentWorld class because GhostRacer may add a HolyWaterProjectile to the World if the user presses SPACE. It will dynamically allocate an HolyWaterProjectile object using the passed in arguments, and add it to the STL list. It is not virtual because there are no StudentWorld derived classes, and therefore does not need to be further redefined.
  + void addHealingGoodie(int xCoord, int yCoord)
    - This function was defined in the StudentWorld class because ZombiePedestrian may add a HealingGoodie at the place it died if it died due to HolyWaterProjectile. It will dynamically allocate an HealingGoodie object using the passed in arguments, and add it to the STL list. It is not virtual because there are no StudentWorld derived classes, and therefore does not need to be further redefined.
  + void incrementSoulsSaved()
    - This function was defined in the StudentWorld class because GhostRacer may pick up a SouleGoodie which is used to progress in the level. It is a simple setter function that increments a private int data member by one whenever it is called. It is not virtual because there are no StudentWorld derived classes, and therefore does not need to be further redefined.

**Functionalities Failed to Finish :**

Although I did this project in a rush due to a big event that is happening the second week we have to do this project, I do not believe I missed out on any functionalities.

**Assumptions :**

* I assumed that Actor should be an abstract base class and therefore made doSomething a purely virtual function despite there being some commonalities in all redefined doSomething implementations. (Check if dead and return immediately if dead). This is because otherwise, an Actor (as well as HitPointActor, MovingNPC, and Goodie objects) object can be created.
* I assumed that pointers to functions weren’t allowed to be used, which made some parts of my source code require some unnecessary repetition.
* I assumed that the slight variation in wording of the specifications regarding how to handle move plan distance made reusability for that portion of the code not possible. This is not a big deal as I was able to simplify other parts of MovingNPC derived class’ doSomething method.

**Design Choices :**

* I decided to use an STL linked list as it would be the fastest to iterate through and delete/add Actors to. The order of the Actors wouldn’t matter, which made it an especially appealing choice.
* I decided to handle all aspects of checking for overlaps in StudentWorld. Although it was possible to have a member function that returns the STL linked list from StudentWorld, this would have a large overhead (though it wouldn’t be too bad if I returned a reference to the list). The downside of my approach was that there were a few methods which served only one specific purpose within StudentWorld.
* I decided to use a non-member function in StudentWorld.cpp to handle whether two Actors are overlapping and another one to get the boundaries of a lane. Neither required access to any private member functions of any classes, which made it seem logical to keep them out of any class. They were also reused multiple times throughout member function implementations.