Description of each class:

class Actor

**Actor():** constructor of this class which receive all the parameters.

**virtual ~Actor()**: a destructor basically does nothing since there is not dynamic memory allocation. It has to be virtual since the Actor class has to be the abstract base class for other classes.

**virtual void doSomething() = 0**: all the actors derived from this class have to do something in a different way and this class itself does not need to and does not need to be constructed as a variable, so it is pure virtual.

**virtual void m\_move():** nearly all actors use the same moving logistic, except for Ghost Racer. So even though it has only one exception, it has to be virtual.

**bool is\_alive() const:** an accessor which returns the alive private member to indicate whether the actor is alive. All actors require the same function so it doesn’t have to be virtual.

**void set\_alive(bool set):** an mutator to change the alive private member. All actors require the same function so it doesn’t have to be virtual.

**int get\_v\_speed() const:** an accessor to get the actor’s vertical speed. All actors have to get its horizontal speed so it doesn’t have to be virtual.

**void set\_v\_speed(int v\_speed):** a mutator to set the actor’s vertical speed. All actors have to set its own vertical speed so it doesn’t have to be virtual.

**int get\_h\_speed() const:** an accessor to get the actor’s horizontal speed. All actors have to get its horizontal speed so it doesn’t have to be virtual.

**void set\_h\_speed(int h\_speed):** a mutator to set the actor’s horizontal speed. All actors have to set its own horizontal speed so it doesn’t have to be virtual.

**int get\_hit\_point() const:** an accessor to get the actor’s hit point. Nearly all the actors have hit point (but in different uses) and get it in same way, so it doesn’t have to be virtual.

**void set\_hit\_point(int hitpoint):** a mutator to set the actor’s hit point. All actors change its hit point in the same way so it doesn’t have to be virtual.

**bool get\_cawa() const:** an accessor to get whether the actor is a collision avoidance warning actor. All actors get it in a same way so it doesn’t need to be virtual.

**bool is\_overlap(Actor\* ac) const:** to check whether the actor is overlap with ac. All actors use the same logistic to check whether it is overlap with others, so it doesn’t have to be virtual.

**int in\_which\_line() const:** to find which line the actor is in. All actors use the same logistic to determine which line it is in, so it doesn’t need to be virtual.

**StudentWorld\* get\_studentworld() const:** an accessor to get the StudentWorld pointer. All actors get it in the same way, so it doesn’t have to be virtual.

class GhostRacer : public Actor

**GhostRacer():** constructor, passing some parameters by initialized list to the Actor class since it is derived from it.

**virtual ~GhostRacer():** virtual destructor basically does nothing since there is no dynamic memory allocation. It is better to be virtual since the GhostRacer class is derived from Actor class.

**virtual void doSomething():** GhostRacer has to do something in its own way different from others’ during every tick, so it has to be virtual.

**virtual void m\_move():** GhostRacer is the only actor that has unique moving logistic, so even with only one exception, this function also has to be virtual.

**void set\_holy\_water\_spray(int hws):** a mutator sets the GhostRacer’s holy water spray. It does not have to be virtual since only the GhostRacer has holy water spray object.

**int get\_holy\_water\_spray() const:** an accessor gets the GhostRacer’s holy water spray. It does not have to be virtual since only GhostRacer has holy water spray object.

**bool check\_hit\_point():** to check whether the GhostRacer’s hit point is equal to zero. It does not have to be virtual since only GhostRacer’s hit points affect the process of the game. If it is equal to zero, the game must stop.

class BorderLine : public Actor

**BorderLine():** a constructor passes some parameters to Actor class by initialized list since it is derived from the Actor class.

**virtual ~BorderLine():** a destructor basically does nothing since there is no dynamic memory allocation in this class. It is better to be virtual since it is derived from Actor class.

**virtual void doSomething():** BorderLine has to do something in its own way during every tick, so it has to be virtual.

class MovingWalkingActor : public Actor

**MovingWalkingActor():** a constructor passes something parameters to Actor class by initialized list since it is derived from the Actor class.

**virtual ~MovingWalkingActor():** a virtual destructor basically does nothing since there is no dynamic memory allocation in this class. It is better to be virtual since this class is derived from Actor class.

**void set\_mpd(int n):** a mutator that sets the MovingWalkingActor’s moving plan distance. It does not have to be virtual since only three MovingWalkingActors have moving plan distance and set it in the same way. The parameter n is used to distinguish different MovingWalkingActors.

**double get\_mpd() const:** an accessor to get the moving plan distance. It does not have to be virtual since all MovingWalkingActors get it in the same way.

**bool dec\_mpd():** decrease moving plan distance by one and check if its less then or equal to zero. It does not have to be vritual since all MovingWalkingActors check it in the same way.

**virtual void doSomething():** all MovingWalkingActors follow the same procedure and skeleton of doing something in particular time and situation even though they have differences in what they actually do. It has to be virtual since the MovingWalkingActor is a base class that is derived from the actor class.

**virtual void OverlapDoSomething() = 0:** when the MovingWalkingActor is overlap with the GhostRacer, it has to do something. It has to be pure virtual to make the MovingWalkingActor class abstract, since every MovingWalkingActor does different things when it is overlap and the base class MovingWalkingActor itself is hard to tell what to do.

**virtual void CheckZpClose():** only zombie pedestrian has to do something when it is close to the GhostRacer, so it is virtual. It is nor pure virtual even though this function does nothing in the MovingWalkingActor class because if it is, all other derived classes have to include this function but does nothing to avoid being abstract, which is more complicated. Thus, I just make it a “dummy function” which does nothing in the base class and only ZombiePedestrian class has to include it.

**virtual bool CheckZcClose():** only zombie cab has to notice collision avoidance warning actors when it is moving, so it is virtual. It is not pure virtual even though it does nothing in the MovingWalkingActor class for the same reason as the CheckZpClose() function.

class HumanPedestrian : public MovingWalkingActor

**HumanPedestrian():** a constructor that passes some parameters to MovingWalkingActor class by initialized list since it is derived from that class.

**virtual ~HumanPedestrian():** a destructor basically does nothing since there is no dynamic memory allocation in this class. It is better to be virtual since it is derived from MovingWalkingActro class.

**virtual bool OverlapDoSomething():** human pedestrian has to get crashed when it is overlap with the GhostRacer. It has to be virtual since it is derived from the MovingWalkingActor class and other derived classes do something when they are overlap with the GhostRacer in different ways.

Class ZombiePedestrian : public MovingWalkingActor()

**ZombiePedestrian():** a constructor passes some parameters to MovingWalkingActor by initialized list since it is derived from that class.

**virtual ~ZombiePedestrian():** a destructor basically does nothing since there is no dynamic memory allocation in this class. It is better to be virtual since it is derived from MovingWalkingActor class.

**void set\_ticks(int tick):** a mutator that sets the ZombiePedestrian’s tick. It does not have to be virtual since only ZombiePedestrian has tick.

**int get\_ticks() const:** an accessor to get the tick. It does not have to be virtual since only ZombiePedestrian has tick.

**virtual bool OverlapDoSomething():** when ZombiePedestrian is overlap with the GhostRacer, it has to get attacked. It is virtual since it is derived from MovingWalkingActor class and every other derived class does something in a different way.

**virtual void CheckZpClose():** when ZombiePedestrian is close to the GhostRacer, it has to do get close to it. It is virtual since it is derived from MovingWalingActor class.

class ZombieCab : public MovingWalkingActor

**ZombieCab():** a constructor passes some parameters to MovingWalkingActor by initialized list since it is derived from that class.

**virtual ~ZombieCab():** a destructor basically does nothing since there is no dynamic memory allocation in this class. It is better to be virtual since it is derived from MovingWalkingActor class.

**void set\_damaged\_ghostracer(bool d\_g):** a mutator to set the damaged ghostracer parameters. It does not have to be virtual since only ZombieCab needs this functionality.

**bool get\_damaged\_ghostracer() const:** an accessor to get whether the ZombieCab has damaged the GhostRacer. It does not have to be virtual since only ZombieCab needs this functionality.

**virtual bool OverlapDoSomething():** when ZombieCab is overlap with the GhostRacer, it has to get crushed. It is virtual since it is derived from MovingWalkingActor class and every other derived class does something in a different way.

**virtual bool CheckZcClose():** when ZombieCab is moving, it has to pay attention to other collision avoidance warning actors to adjust its speed. It is virtual since it is derived from MovingWalingActor class.

class Goodie : public Actor

**Goodie():** a constructor passes some parameters to Actor class by initialized list since it is derived from that class.

**virtual ~Goodie():** a destructor basically does nothing since there is no dynamic memory allocation in this class. It is better to be virtual since it is derived from Actor class.

**virtual void doSomething():** every goodie does something follow a same sequence and skeleton in which time and situation, even though they are different from what they actually do. It is virtual since it is derived from Actor class.

**virtual void GoodieDoSomething() = 0:** every goodie has to do something when it is overlap with the GhostRacer. It is pure virtual to make the Goodie class abstract since every Goodie behaves differently and it is hard to tell what the Goodie class itself does.

class OilSlick : public Goodie

**OilSlick():** a constructor passes some parameters to Goodie class by initialized list since it is derived from that class.

**virtual ~OilSlick():** a destructor basically does nothing since there is no dynamic memory allocation in this class. It is better to be virtual since it is derived from Goodie class.

**virtual void GoodieDoSomething():** OilSlick has to slick when it is overlap whit the GhostRacer. It is virtual since it is derived from the Goodie class and other goodies behave in different ways.

class HealingGoodie : public Goodie

**HealingGoodie():** a constructor passes some parameters to Goodie class by initialized list since it is derived from that class.

**virtual ~HealingGoodie():** a destructor basically does nothing since there is no dynamic memory allocation in this class. It is better to be virtual since it is derived from Goodie class.

**virtual void GoodieDoSomething():** HealingGoodie has to heal the GhostRacer when it is overlap whit the GhostRacer. It is virtual since it is derived from the Goodie class and other goodies behave in different ways.

class HolyWaterGoodie : public Goodie

**HolyWaterGoodie():** a constructor passes some parameters to Goodie class by initialized list since it is derived from that class.

**virtual ~HolyWaterGoodie():** a destructor basically does nothing since there is no dynamic memory allocation in this class. It is better to be virtual since it is derived from Goodie class.

**virtual void GoodiDoSomething():** HolyWaterGoodie has to refill the holy water spray when it is overlap whit the GhostRacer. It is virtual since it is derived from the Goodie class and other goodies behave in different ways.

class SoulGoodie : public Goodie

**SoulGoodie():** a constructor passes some parameters to Goodie class by initialized list since it is derived from that class.

**virtual ~SoulGoodie():** a destructor basically does nothing since there is no dynamic memory allocation in this class. It is better to be virtual since it is derived from Goodie class.

**virtual void GoodiDoSomething():** SoulGooide has to be saved when it is overlap whit the GhostRacer. It is virtual since it is derived from the Goodie class and other goodies behave in different ways.

class HolyWaterProjectile : public Actor

**HolyWaterProjectile():** a constructor passes some parameters to Actor class by initialized list since it is derived from that class.

**virtual ~HolyWaterProjectile():** a destructor basically does nothing since there is no dynamic memory allocation in this class. It is better to be virtual since it is derived from Actor class.

**virtual void doSomething():** HolyWaterProjectile has to react when it is overlap whit the zombies and some goodies, either attack it or just destroy it. It is virtual since it is derived from the Actor class and other actorsbehave in different ways.

class StudentWorld : public GameWorld

**StudentWorld():** the constructor to start the game world

**~StudentWorld():** the destructor basically calls the cleanup() function to free all the memory.

**virtual int init():** initializes the game and fills it with basic actors. It is virtual since it is derived from GameWorld.

**virtual int move():** gives every actor a chance to move and clean all the not alive actors. Finally add more actors to the game. It is virtual since it is derived from GameWorld.

**virtual void cleanUp():** free all the memory that actors occupy. It is virtual since it is derived from GameWorld.

**GhostRacer\* getghostracer() const:** return the pointer to GhostRacer actor.

**int get\_soul\_saved() const:** return the number of souls that has been saved.

**void save\_a\_soul():** increase the souls saved by one.

**int get\_bonus\_score() const:** get the current bonus score.

**void set\_bonus\_score(int score):** set the bonus score to a particular value.

**std::vector<Actor\*> get\_m\_actor():** return the stl container that contains all the other actors except ghostracer.

**void add\_healing\_goodie(double x\_cor, double y\_cor):** add a healing goodie at that particular coordinates.

**void add\_oil\_slick(double x\_cor, double y\_cor):** add a oil slick at that particular coordinates

**void add\_holy\_water\_projectile(double x\_cor, double y\_cor, int dir):** add a holy water projectile at that particular coordinate and toward that direction.

A list of all functionality that I failed to finish as well as known bugs in my classes:

So far, I haven’t discovered any yet.

A list of other design decisions and assumptions I made:

When the oilslick has to slide the GhostRacer, the GhostRacer will turn a random degree between 5 and 20. However, when we adjust the GhostRacer’s direction by keyboard, it is adjusted 8 degrees per time. This means that if the random slide is not the multiple of 8, it is possible to have the GhostRacer not moving vertically. Therefore, I rewrite the code at that part to make sure that the last turn toward 90 degree will ensure that the GhostRacer moves vertically.