Project 3
Ghost Racer

For questions about this project, first consult your TA.
If your TA can't help, ask Professor Nachenberg.

Time due:

Part 1: 11 PM, Saturday, February 20
Part 2: 11 PM, Saturday, February 27

WHEN IN DOUBT ABOUT A REQUIREMENT, YOU WILL NEVER LOSE CREDIT IF YOUR SOLUTION WORKS THE SAME AS OUR POSTED SOLUTION.
SO PLEASE DO NOT ASK ABOUT ITEMS WHERE YOU CAN DETERMINE THE PROPER BEHAVIOR ON YOUR OWN FROM OUR SOLUTION!

BACK UP YOUR SOLUTION EVERY 30 MINUTES TO THE CLOUD OR A THUMB DRIVE. WE WILL
NOT ACCEPT "MY COMPUTER CRASHED" EXCUSES FOR LATE WORK.

PLEASE THROTTLE THE RATE YOU ASK QUESTIONS
TO 1 EMAIL PER DAY! IF YOU'RE SOMEONE WITH
LOTS OF QUESTIONS, SAVE THEM UP AND ASK ONCE.

Table of Contents

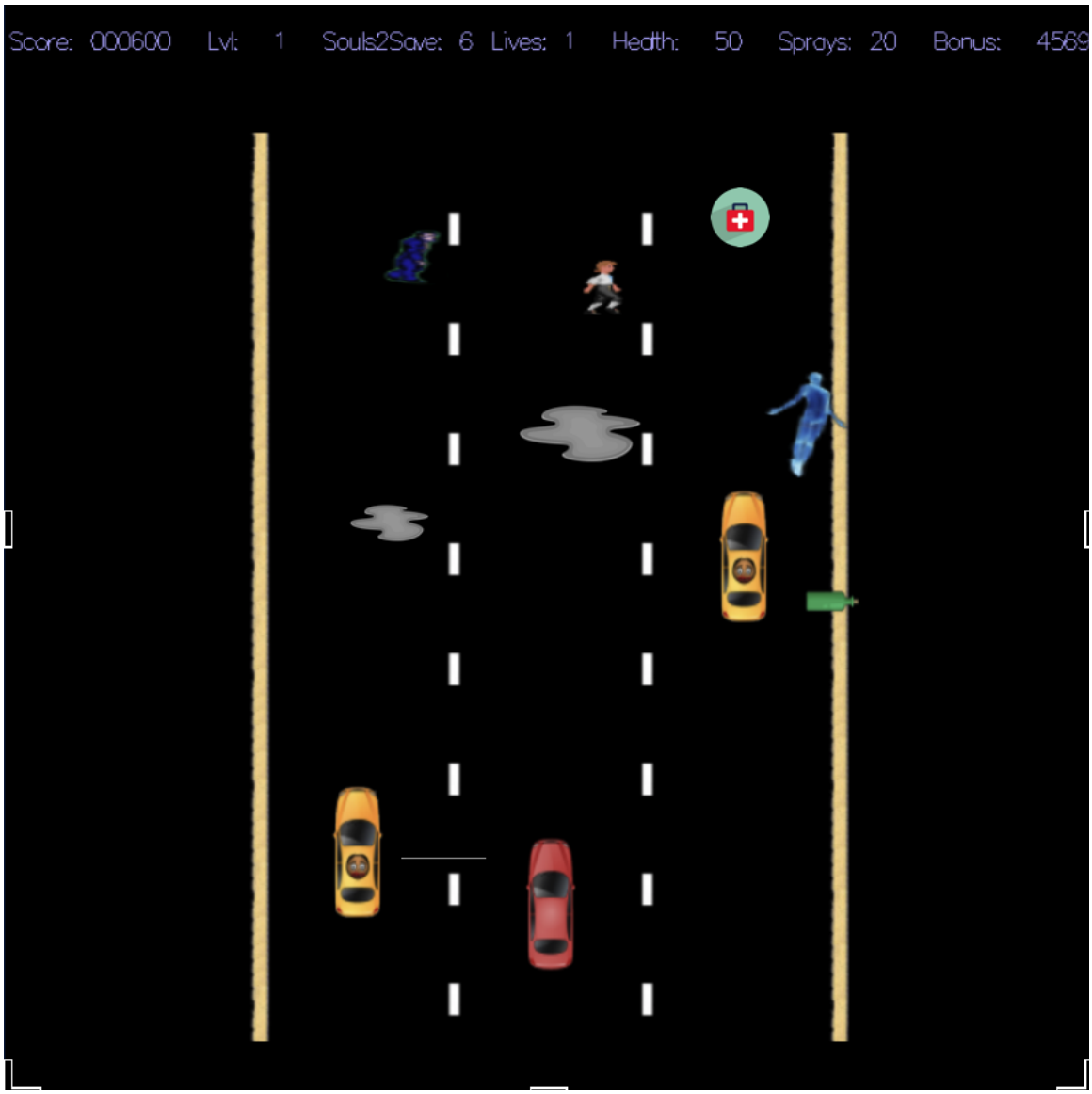
Introduction

NachenGames corporate spies have learned that SmallSoft is planning to release a new driving game called Ghost Racer, and would like you to program an exact copy so NachenGames can beat SmallSoft to the market. To help you, NachenGames corporate spies have managed to steal a prototype Ghost Racer executable file and several source files from the SmallSoft headquarters, so you can see exactly how your version of the game must work (see posted executable file) and even get a head start on the programming. Of course, such behavior would never be appropriate in real life, but for this project, you'll be a programming villain.

Ghost Racer is set in the year 2022 in post-apocalyptic Los Angeles. Zombies roam around unhindered, and have even learned how to drive taxi cabs (better than most Los Angeles taxi drivers, but not better than Lyft drivers). The few humans that are left wander aimlessly around searching for long lost friends. And the souls of the recently deceased have for some strange reason convened on the 405 freeway, waiting for someone to free them to be sent to the afterlife. Our hero is Aisha Washington, a recently-certified zombie hunter, amateur Formula 1 driver, and former Google Principal Engineer (the world has no need for Google anymore). Aisha must drive her Ghost Racer (a modified Subaru Forester, like Prof. Nachenberg drives) north on the 405 freeway, freeing trapped souls so they can go to the afterlife. While doing so, Aisha has to dodge wandering human pedestrians (hitting a human ped instantly ends the level), shoot zombies with holy water, run zombie cabs off the road, and avoid dangerous oil slicks (which cause Aisha's Ghost Racer to slip and slide). Aisha can also pick up goodies like healing kits and holy water recharges that she finds on the road to help in her quest. Once Aisha has freed all of the souls from the current stretch of the 405, she advances to the next, more infested stretch of the freeway, where she continues the good fight. The 405 goes on and on[1], so Aisha will be busy for a long time.

Here's a screenshot of the game:

[1] Well, to be accurate, until Sylmar.



As you can see, the game is set upon the 405 freeway (the apocalypse left only three lanes usable). Aisha's vehicle is the candy-red Ghost Racer at the bottom of the screen. We also see an innocent human pedestrian (white shirt and grey pants) in the upper middle of the screen, a zombie pedestrian (dark blue) in the upper left, two zombie-driven taxi cabs (shown in yellow), and a soul waiting to be saved (in the mid-upper-right). Also visible is a bottle of holy water (green, mid-right), which Aisha needs to pick up to refill her Ghost Racer's holy water cannon. Shooting holy water at the zombie peds and cabs causes them to dissolve into sludge, but fortunately does nothing but annoy the human pedestrians. Also shown are several grey oil slicks (that cause Aisha's Ghost Racer to skid) and a healing goodie (upper right) that repairs the Ghost Racer when it gets injured by running into cabs or zombie peds.

You, the player, will play the role of Aisha and use the following keystrokes to control her Ghost Racer:

- Left arrow key or the ‘a’ key: Turns the Ghost Racer steering wheel left
- Right arrow key or the ‘d’ key: Turns the Ghost Racer steering wheel right
- Up arrow key or the ‘w’ key: Speeds the Ghost Racer up
- Down arrow key or the ‘s’ key: Slows the Ghost Racer down
- Space bar: Fires holy water, if Ghost Racer has any left
- The ‘q’ key: Quits the game
- The ‘f’ key: Causes the game to run one tick at a time for debugging

Points are awarded as follows:

- When Ghost Racer disposes of a zombie pedestrian: 150 points
- When Ghost Racer disposes of a zombie cab: 200 points
- When Ghost Racer saves a soul: 100 points
- When Ghost Racer picks up a holy water refill goodie: 50 points
- When Ghost Racer picks up a healing goodie: 250 points
- Bonus points for finishing the level quickly: max of 5000 (goes down as the game progresses)

Game Details

In Ghost Racer, Aisha Washington's Ghost Racer starts out a new game with three lives and continues to play until all of its lives have been exhausted. There are multiple levels in Ghost Racer, beginning with level 1 (NOT zero), and during each level the player, playing the role of Aisha, must drive down the 405 saving all of the lost souls. To complete level L, the Aisha must save 2 * L + 5 lost souls by driving her Ghost Racer over them.

The Ghost Racer screen is exactly 256 pixels wide by 256 pixels high. The bottom-leftmost pixel has coordinates x=0, y=0, while the upper-rightmost pixel has coordinate x=255,y=255, where x increases to the right and y increases upward toward the top of the screen. The *GameConstants.h* file we provide defines constants that represent the game’s width and height (VIEW_WIDTH and VIEW_HEIGHT), which you must use in your code instead of hard-coding the integers. Every object in the game (e.g., Ghost Racer, human pedestrians, healing goodies, etc.) will have an x coordinate in the range [0, VIEW_WIDTH), and a y coordinate in the range [0, VIEW_HEIGHT). That file also defines constants ROAD_WIDTH and ROAD_CENTER. The 405 roadway runs upward in the middle of the screen and is ROAD_WIDTH pixels wide, with each of the three lanes being ROAD_WIDTH/3 pixels wide. The middle lane’s center is at x=ROAD_CENTER, its left lane’s center is at x=ROAD_CENTER - ROAD_WIDTH/3, and its right lane's center is at x=ROAD_CENTER + ROAD_WIDTH/3.

Each level has a random set of pedestrians, enemies (zombie peds and zombie cabs), obstacles (like oil slicks), goodies (like holy water refills or healing kits) and souls to save. The density of oil slicks, human pedestrians, zombie pedestrians, zombie cabs, and

souls you need to save varies based on the level to make the game more difficult as the player solves each level. For instance, later levels will have more zombie peds and cabs to dodge, and more souls to save. Details will be described in the sections below.

At the beginning of each level, and when the player restarts the current level because the Ghost Racer was destroyed, the level must be reset to an initial state in which the road starts out empty, with none of the previous baddies or goodies present. At the beginning of each level or when the player restarts a level because the Ghost Racer was destroyed, Ghost Racer starts out with ten holy water sprays and 100 hit points (health points).

Once a level begins, it is divided into small time periods called *ticks*. There are dozens of ticks per second (to provide smooth animation and gameplay).

During each tick of the game, your program must do the following:

- You must give each object – including the Ghost Racer, human peds, zombie peds, zombie cabs, goodies, oil slicks, souls, road border lines, etc. - a chance to do something – e.g., move, die, etc.
- You must check to see if Ghost Racer has been destroyed. If so, you must indicate this to our game framework (we'll tell you how later) so the level can end and potentially restart fresh, if Ghost Racer has more lives.
- You must delete/remove all dead objects from the game – this includes zombie peds and zombie cabs that have been destroyed by holy water, holy water shots fired by the Ghost Racer that have dissipated, souls that has been saved, goodies that have been picked up, and all objects that have gone off the screen (e.g., a pedestrian gets passed by Ghost Racer and disappears off the screen, a cab that veers off the road, or road border lines that have flowed off the bottom of the screen).
- Your code may also need to introduce one or more new objects into the game – for instance, a new zombie cab may be introduced at either the top or bottom of the road, a human or zombie ped might be introduced at the top of the road, an oil slick, holy water refill or soul might be introduced at the top of the road. White and yellow border lines may need to be added to the road. A healing goodie may be generated occasionally by destroyed zombie peds, and oil slicks may be generated by zombie cabs when they're destroyed.
- You must update the game statistics line at the top of the screen, including the number of remaining lives Aisha's Ghost Racer has, the player's current score, the current level number, the number of holy water sprays Ghost Racer currently holds, Ghost Racer's current health level (between 0 and 100), as well as the current level bonus.
- Check to see if Ghost Racer has completed the current level (by saving the required number of souls), and if so, add any remaining bonus points to the player's score, and end the current level so Ghost Racer may advance to the next level.

The status line at the top of the screen must have the following components:

Score: 2100 Lvl: 1 Souls2Save: 5 Lives: 3 Health: 95 Sprays: 22 Bonus: 4321

Each labeled value of the status line must be separated from the next by exactly two spaces. For example, the 3 between "Lives: 3" and "Health:" must have two spaces after it. You may find the *Stringstreams* writeup on the main class web site to be helpful.

There are three major types of "goodies" that Ghost Racer will want to pick up: healing goodies, holy water bottle goodies, and soul goodies (the souls that need to be saved). Goodies "trigger" when the Ghost Racer drives on top of them; at this time, they update Ghost Racer's stats in some positive way (e.g., increasing its amount of holy water and points), then disappear. Their respective behaviors are described in the sections on goodies below. Similar to a goodie (but bad for Ghost Racer) are oil slicks. Like a goodie, oil slicks trigger when they come into contact with Ghost Racer, but instead of doing something good, they cause the Ghost Racer vehicle to swerve uncontrollably. These types of objects just move down the road (at a rate that depends on Ghost Racer's speed), and if they come into contact with the Ghost Racer, they affect the Ghost Racer in their good or bad way and disappear.

Assuming Ghost Racer has holy water in its tanks, the player may spray Ghost Racer's holy water forward by pressing the spacebar. The player can get more holy water by picking up holy water bottle goodies, shown as green bottles on the screen. A holy water shot will do 1 hit point of damage to a zombie ped (which is destroyed after 2 shots, because zombie peds have 2 hit points) or a zombie cab (which is destroyed after 3 holy water shots, since zombie cabs have 3 hit points). If a squirt of holy water comes into contact with holy water refill goodies or healing goodies, it will instantly destroy them. Holy water shots will stop when they hit human peds, but do no damage to humans. Holy water will simply pass over lost souls, oil slicks, and road border lines.

As the Ghost Racer speeds down the 405, if it comes into contact with a zombie ped, this does 5 hit points of damage to the Ghost Racer and instantly kills the zombie. If Ghost Racer comes into contact with a zombie cab, it will do 20 hit points of damage to the Ghost Racer, and cause the zombie cab to veer off the road. If the Ghost Racer bumps into either the left or right sides of the freeway, it suffers 10 points of damage. If Ghost Racer is destroyed (its hit points reach zero or below due to hitting zombie peds or zombie cabs or road borders) the player loses one "life." If, after losing a life, the player has one or more remaining lives left, they are placed back in a newly-generated Ghost Racer at the current level and must again save all of the souls from scratch. If Ghost Racer is destroyed and the player has no lives left, then the game is over.

There are three types of "intelligent agents" in Ghost Racer: human pedestrians, zombie pedestrians, and zombie cabs. Human pedestrians wander around the road aimlessly looking for lost friends. Zombie peds similarly wander around aimlessly along the road,

but will turn to run at the Ghost Racer if they wander in front of it. Zombie cabs drive down the road at various speeds, getting in Aisha's way as she drives the Ghost Racer. Their exact behaviors are described in the sections below.

Your game implementation must play various sounds when certain events occur, using the *playSound()* method provided by our *GameWorld* class, e.g.:

```
// Make a sound effect when Ghost Racer sprays its holy water squirter
pointerToWorld->playSound(SOUND_PLAYER_SPRAY);
```

- You must play a SOUND_PLAYER_SPRAY sound any time Ghost Racer successfully shoots a squirt of holy water.
- You must play a SOUND_OIL_SLICK sound any time Ghost Racer runs over an oil-slick and skids.
- You must play a SOUND_PED_HURT sound any time a zombie pedestrian is injured.
- You must play a SOUND_PED_DIE sound any time a zombie pedestrian is destroyed.
- You must play a SOUND_VEHICLE_HURT sound any time a zombie cab is injured.
- You must play a SOUND_VEHICLE_DIE sound any time a zombie cab is destroyed.
- You must play a SOUND_VEHICLE_CRASH sound any time the ghost racer runs into a zombie cab or into one of the left or right sides of the road.
- You must play a SOUND_PLAYER_DIE sound any time Ghost Racer is destroyed.
- You must play a SOUND_GOT_GOODIE sound any time the player successfully picks up a goodie (other than a soul goodie).
- You must play a SOUND_GOT_SOUL sound any time the player successfully saves a soul.
- You must play a SOUND_FINISHED_LEVEL sound every time Ghost Racer successfully completes a level.

Constants for each specific sound, e.g., SOUND_GOT_SOUL, may be found in our *GameConstants.h* file.

Determining Object Overlap

In a video game, it's often important to determine if two game objects come into contact with each other (are they close enough that they touch/overlap and therefore interact with one another). For example, if Ghost Racer shoots a holy water spray, does the spray come into contact with a nearby zombie ped as it flies upward? Or when Ghost Racer drives near a goodie, did it get close enough to pick it up?

For the purposes of Ghost Racer, use the following algorithm to determine if two objects A and B overlap:

1. Compute the delta_x between A and B (which is the absolute value of the distance between A's center x coordinate and B's center x coordinate).
2. Compute the delta_y between A and B (which is the absolute value of the distance between A's center y coordinate and B's center y coordinate).
3. Compute the sum of the radiuses of A and B. While most of our graphics are rectangular, you can treat them a bit like a circle to simplify things.
4. If delta_x < radius_sum*.25 AND delta_y < radius_sum*.6 then the two objects are said to overlap[2].

You must use this approach to detect overlap between two different objects any time this specification requires you to detect overlap conditions. Our framework's class *GraphObject* from which all your game objects' types must derive (detailed below) provides methods *getX()*, *getY()*, and *getRadius()* so that you can determine an object's center coordinates and radius.

So how does a video game work?

Fundamentally, a video game is composed of a bunch of game objects; in Ghost Racer, those objects include the Ghost Racer, pedestrians (e.g., human and zombies), zombie cars, goodies (e.g., healing goodies, holy water refill goodies), oil slicks, projectiles (e.g., holy water spray), road border lines, and lost souls. Let's call these objects "actors," since each object is an actor in our video game. Each actor has its own (x, y) location in space, its own internal state (e.g., a zombie knows its location, what direction it's moving, etc.) and its own special algorithms that control its actions in the game based on its own state and the state of the other objects in the world. In the case of Ghost Racer, the algorithm that controls the Ghost Racer object is the user's own brain and hand, and the keyboard! In the case of other actors (e.g., a pedestrian), each object has an internal autonomous algorithm and state that dictates how the object behaves in the game world.

Once a game begins, gameplay is divided into *ticks*. A tick is a unit of time, for example, 50 milliseconds (that's 20 ticks per second).

During a given tick, the game calls upon each object's behavioral algorithm and asks the object to perform its behavior. When asked to perform its behavior, each object's behavioral algorithm must decide what to do and then make a change to the object's state (e.g., move the object one pixel to the left, and one pixel down), or change another objects' state (e.g., when a holy water refill goodie detects that it overlaps with the Ghost Racer, it might increase the Ghost Racer's quantity of holy water). Typically, the behavior exhibited by an object during a single tick is limited in order to ensure that the gameplay is smooth and that things don't move too quickly and confuse the player. For example, a pedestrian will move just a few pixels forward, rather than moving ten or

[2] If you think this approach for detecting overlap seems a bit like a hack, it is. In our game framework, all of our object images are represented by NxN squares, but as in real life, vehicles tend to be longer than they are wide. So this hack accounts for this asymmetry. It's not perfect, but good enough to detect collisions for the purposes of a CS 32 project.

more pixels per tick; a pedestrian moving, say, 20 pixels in a single tick would confuse the user, because humans are used to seeing smooth movement in video games, not jerky shifts.

After the current tick is over and all actors have had a chance to adjust their state (and possibly adjust other actors' states), the graphical framework that we provide animates the actors onto the screen in their new configuration. So if a pedestrian changed its location from (10, 50) to (9, 51) (i.e., moved one pixel left, and one pixel up), then our game framework would erase the graphic of the pedestrian from location (10, 50) on the screen and draw the Pedestrian's graphic at (9, 51) instead. Since this process (asking actors to do something, then animating them to the screen) happens 20 times per second, the user will see somewhat smooth animation.

Then, the next tick occurs, and each actor's algorithm is again allowed to do something, our framework displays the updated actors on-screen, etc.

Assuming the ticks are quick enough (a fraction of a second), and the actions performed by the objects are subtle enough (i.e., a pedestrian doesn't move 3 inches away from where it was during the last tick, but instead moves 1 millimeter away), when you display each of the objects on the screen after each tick, it looks as if each object is performing a continuous series of fluid motions.

A video game can be broken into three different phases:

Initialization: The Game World is initialized and prepared for play. This involves allocating one or more actors (which are C++ objects) and placing them in the game world so that they will appear when our framework first shows the level on the screen.

Gameplay: Gameplay is broken down into a bunch of ticks. During each tick, all of the actors in the game have a chance to do something, and perhaps die. During a tick, new actors may be added to the game and actors who die must be removed from the game world and deleted.

Cleanup: The player has lost a life (but has more lives left), or the player has completed the current level, or the player has lost all of their lives and the game is over. This phase frees all of the objects in the world (e.g., Ghost Racer, peds, oil slicks, holy water sprays (that were fired by the Ghost Racer), border lines, goodies, lost souls, etc.) since the level has ended. If the game is not over (i.e., the player has more lives), then the game proceeds back to the *Initialization* step, where the level is repopulated with new occupants, and gameplay starts from scratch for the level.

Here is what the main logic of a video game looks like, in pseudocode (The *GameController.cpp* we provide for you has some similar code):

```
while (Ghost Racer has lives left)
{
   Prompt the user to start playing    // "press any key to start"
   Initialize the game world           // you're going to write this

   while (Ghost Racer is still alive)
   {
         // each pass through this loop is a tick (1/20th of a sec)

           // you're going to write code to do the following
         Tell all actors to do something
         Remove any dead actors from the world

           // we write this code to handle the animation for you
         Animate each actor to the screen
         Sleep for one tick to give the user time to react
   }
     // Ghost Racer died – you're going to write this code
   Clean up all game world objects     // you're going to write this
   if (Ghost Racer still has lives left)
      Prompt the player to continue
}

Tell the player the game is over      // we provide this
```

And here is what `Tell all actors to do something` might do:

```
for each actor on the level:
    if (the actor is still alive)
        tell the actor to doSomething()
```

You will typically use a container (an array, vector, or list) to hold pointers to each of your live actors. Each actor (a C++ object) has a *doSomething( )* member function in which the actor decides what to do. For example, here is some pseudocode showing what a (simplified) zombie pedestrian might decide to do each time it gets asked to do something:

```
class ZombiePedestrian: public SomeOtherClass
{
   public:
       virtual void doSomething()
       {
           If the Ghost Racer overlaps with me on the road, then
                  Damage the player by 5 hit points
                  Set my alive state to false
           Else if I still want to wander in the same direction
                  Move one pixel in my chosen wandering direction
           Else if I'm done wandering in the same direction
                  Pick a new direction to wander
                  Pick a new number of ticks to wander in that direction
           Else if I have fallen off the bottom of the screen
                  Set my alive state to false
           Else ...
       }
       ...
};
```

And here's what Ghost Racer's *doSomething( )* member function might look like:

```
class GhostRacer: public …
{
   public:
        virtual void doSomething()
        {
             Try to get user input (if any is available)
               If the user pressed the LEFT key then
                   Rotate me 8 degrees counter-clockwise
               If the user pressed the RIGHT key then
                   Rotate me 8 degrees clockwise
               ...
               If user pressed space and I have holy water, then
                  Introduce a new holy water spray object in front of
                         me, with an angle matching my angle
             If my angle > 90 degrees, then
                  Reduce my X location by 1 pixel
             If my angle < 90 degrees, then
                  Increase my X location by 1 pixel
        }
        ...
};
```

Whenever the spec calls for selecting a random integer, you may use the *randInt()* function defined in *GameConstants.h*. Calling *randInt(min, max)* returns a uniformly distributed random integer between *min* and *max*, inclusive.

What Do You Have to Do?

You must create a number of different classes to implement the Ghost Racer game. Your classes must work properly with our provided classes, and **you MUST NOT modify our provided classes or our source files in any way to get your classes to work properly (doing so will result in a score of zero on the entire project!).** Here are the specific classes that you must create:

1. You must create a class called *StudentWorld* that is responsible for keeping track of your game world and all of the actors/objects (Ghost Racer, zombie pedestrians, human pedestrians, zombie cabs, holy water projectiles, healing goodies, holy water refill goodies, yellow and white road border lines, oil slicks, and lost souls) that are inside the game.
2. You must create a class to represent Ghost Racer in the game.
3. You must create classes for zombie pedestrians, human pedestrians, zombie cabs, holy water projectiles, healing goodies, holy water refill goodies, border lines, oil slicks, lost souls, etc., as well as any additional base classes (e.g., a goodie base class if you find it convenient, since goodies share many things in common) that help you implement your actors.

You Have to Create the StudentWorld Class

Your *StudentWorld* class is responsible for orchestrating virtually all gameplay – it keeps track of the entire game world (each level and all of its inhabitants such as pedestrians, road border lines, the Ghost Racer, goodies, holy water projectiles, oil slicks, etc.). It is responsible for initializing the game world at the start of the game, asking all the actors to do something during each tick of the game, destroying an actor when it disappears (e.g., a zombie ped dies, a zombie cab disappears off the bottom of the screen because Ghost Racer passed it up, Ghost Racer shoots holy water and it hits a healing goodie and unfortunately destroys it, a holy water projectile dissipates after flying 160 pixels through the air, etc.), and destroying **all** of the actors in the game world when the user loses a life or advances to the next level.

Your *StudentWorld* class **must** be derived from our *GameWorld* class (found in *GameWorld.h*) and **must** implement at least these three methods (which are defined as pure virtual in our *GameWorld* class):

```
virtual int init() = 0;
virtual int move() = 0;
virtual void cleanUp() = 0;
```

The code that you write must *never* call any of these three functions (except that *StudentWorld*'s destructor may call *cleanUp()*). Instead, our provided game framework will call these functions for you. So you have to implement them correctly, but you won't ever call them yourself in your code (except in the one place noted above).

Each time a level starts, our game framework will call the *init()* method that you defined in your *StudentWorld* class. You don't call this function; instead, our provided framework code calls it for you.

The *init()* method is responsible for constructing a representation of the current level in your *StudentWorld* object and populating it with initial objects (e.g., road border lines and the Ghost Racer), using one or more data structures that you come up with.

The *init()* method is automatically called by our provided code either (a) when the game first starts, (b) when the player completes the current level and advances to a new level (that needs to be initialized), or (c) when the user loses a life (but has more lives left) and the game is ready to restart at the current level.

After the *init()* method finishes initializing your data structures/objects for the current level, it **must** return `GWSTATUS_CONTINUE_GAME`.

Once a level has been prepared with a call to the *init()* method, our game framework will repeatedly call the *StudentWorld's move()* method, at a rate of roughly 20 times per second. Each time the *move()* method is called, it must run a single tick of the game. This means that it is responsible for asking each of the game actors (e.g., Ghost Racer, each road border line, each pedestrian, each zombie cab, each goodie, each holy water

projectile, each lost soul, each oil slick, etc.) to try to do something: e.g., move themselves and/or perform their specified behavior. This method might also introduce new actors into the game, for instance adding a new holy water refill goodie to the top of the roadway. Finally, this method is responsible for disposing of (i.e., deleting) actors that need to disappear during a given tick (e.g., holy water spray that falls to the ground and disappears, a dead zombie pedestrian, an actor that disappears off the bottom of the screen, etc.). For example, if a zombie pedestrian is shot twice by the Ghost Racer's holy water, then its state should be set to dead, and then after all of the alive actors in the game get a chance to do something during the tick, the *move()* method should remove that zombie pedestrian from the game world (by deleting its object and removing any reference to the object from the *StudentWorld's* data structures). The *move()* method will automatically be called once during each tick of the game by our provided game framework. You will never call the *move()* method yourself.

The *cleanUp()* method is called by our framework when Ghost Racer completes the current level or loses a life (e.g., because of too much damage from collisions with zombies, or by running into a human pedestrian). The *cleanUp()* method is responsible for freeing all actors (e.g., all ped objects, all vehicle objects, all oil slick objects, all projectile objects, all goodie objects, the Ghost Racer object, lost soul objects, road border line objects, etc.) that are currently in the game. This includes all actors created during either the *init()* method or introduced during subsequent gameplay by the actors in the game (e.g., a holy water spray that was added to the screen by the Ghost Racer when the user hit the space bar, a goodie that was added after a few minutes of play) that have not yet been removed from the game.

You may add as many other public/private member functions or private data members to your *StudentWorld* class as you like (in addition to the above three member functions, which you *must* implement). You must **not** add any public data members.

Your *StudentWorld* class must be derived from our *GameWorld* class. Our *GameWorld* class provides the following methods for your use:

```
int getLevel() const;
int getLives() const;
void decLives();
void incLives();
int getScore() const;
void increaseScore(int howMuch);
void setGameStatText(string text);
bool getKey(int& value);
void playSound(int soundID);
```

getLevel() can be used to determine the current level number.

getLives() can be used to determine how many lives Ghost Racer has left.

decLives() reduces the number of Ghost Racer lives by one.

incLives() increases the number of Ghost Racer lives by one.

getScore() can be used to determine Ghost Racer's current score.

increaseScore() is used by a *StudentWorld* object (or your other classes) to increase the user's score upon successfully disposing of a zombie cab, picking up a goodie of some sort, saving a soul, etc. When your code calls this method, you must specify how many points the user gets (e.g., 150 points for destroying a zombie pedestrian). This means that the game score is controlled by our *GameWorld* object – you *must not* maintain your own score data member in your own classes.

The *setGameStatText()* method is used to specify what text is displayed at the top of the game screen, e.g.:

Score: 2100 Lvl: 1 Souls2Save: 5 Lives: 3 Health: 95 Sprays: 22 Bonus: 4321

getKey() can be used to determine if the user has hit a key on the keyboard to move Ghost Racer or to fire a squirt of holy water. This method returns true if the user hit a key during the current tick, and false otherwise (i.e., if the user did not hit any key during this tick). The only argument to this method is a variable that will be set to the key that was pressed by the user (if any key was pressed). If the function returns true, the argument will be set to one of the following values (defined in *GameConstants.h*):

```
KEY_PRESS_LEFT
KEY_PRESS_RIGHT
KEY_PRESS_UP
KEY_PRESS_DOWN
KEY_PRESS_SPACE
```

The *playSound()* method can be used to play a sound effect when an important event happens during the game (e.g., a zombie ped dies or Ghost Racer picks up a goodie). You can find constants (e.g., SOUND_PLAYER_SPRAY) that describe what noise to make in the *GameConstants.h* file. The *playSound()* method is defined in our *GameWorld* class, which you will use as the base class for your *StudentWorld* class. Here's how this method might be used:

```
// if a zombie pedestrian dies, make a dying sound

if (theZombiePedestrianHasDied())
    pointerToWorld->playSound(SOUND_PED_DIE);
```

init() Details

Your *StudentWorld*'s *init()* member function must:

1. Initialize the data structures used to keep track of your game's world.

2. Allocate and insert a Ghost Racer object into the game world. Every time a level starts or restarts, Ghost Racer starts out fully initialized (with the baseline number of holy water sprays, hit points, etc.).
3. Allocate and insert all of the road border lines (white dashed lines and continuous yellow lines) into the game world as described below.

Your *init()* method must construct a representation of your world and store this in a *StudentWorld* object. It is **required** that you keep track of all of the actors (e.g., actors like pedestrians and vehicles, oil slicks, lost souls, projectiles like spray, goodies, etc.) in a **single** STL collection such as a *list*, *set*, or *vector*. (To do so, we recommend using a container of pointers to the actors). If you like, your *StudentWorld* object may keep a separate pointer to Ghost Racer object rather than keeping a pointer to that object in the container with the other actor pointers; Ghost Racer is the **only** actor pointer allowed to not be stored in the single actor container. The *init()* method may also initialize any other *StudentWorld* data members it needs, such as the number of remaining actors that need to be destroyed on this level before Ghost Racer can advance to the next level.

You must not call the *init()* method yourself. Instead, this method will be called by our framework code when it's time for a new game to start (or when the player completes a level or needs to restart a level).

Here is how the *init()* method **must** add objects to the roadway at the start of each level:

1. Create and add a new Ghost Racer/player object at location (x=128, y=32) to the roadway; this is in the bottom-middle of the roadway.
2. Add N yellow border line objects (each object shows a short yellow border line segment) on each side of the roadway to mark the left and right boundaries of the 405, where N is equal to VIEW_HEIGHT / SPRITE_HEIGHT:
    a. Let LEFT_EDGE = ROAD_CENTER - ROAD_WIDTH/2
    Let RIGHT_EDGE = ROAD_CENTER + ROAD_WIDTH/2
    b. Each left yellow border line must have an X value of LEFT_EDGE
    c. Each right yellow border line must have an X value of RIGHT_EDGE
    d. For each of the N total border line objects on each side (j goes from 0 to N-1, inclusive), its Y value must be equal to j * SPRITE_HEIGHT
3. Add M white border line objects (each object shows a short white border line segment) to separate the three lanes of the 405, where M is equal to VIEW_HEIGHT / (4*SPRITE_HEIGHT):
    a. Let LEFT_EDGE = ROAD_CENTER - ROAD_WIDTH/2
    Let RIGHT_EDGE = ROAD_CENTER + ROAD_WIDTH/2
    b. The white border line that separates the left-most and middle lanes must have an X value of LEFT_EDGE + ROAD_WIDTH/3
    c. The white border line that separates the middle and right-most lanes must have an X value of RIGHT_EDGE - ROAD_WIDTH/3
    d. For each of the M total border line objects separating the three lanes (j goes from 0 to M-1, inclusive), its Y value must be equal to j * (4*SPRITE_HEIGHT)

move() Details

The *move()* method must perform the following activities:

1. It must tell all of the actors that are currently active in the game world to do something (e.g., tell a pedestrian to move itself, tell a goodie to check if it overlaps with Ghost Racer (and if so, grant it a special power), give Ghost Racer a chance to move or shoot holy water, etc.).
    a. If an actor does something that causes Ghost Racer to die (e.g., a zombie cab bumps into Ghost Racer such that its hit points reach zero), then the *move()* method should immediately return `GWSTATUS_PLAYER_DIED`.
    b. Otherwise, if Ghost Racer has saved the required number of souls for the current level (for level L, that's L*2 + 5), then it's time to advance to the next level. In this case, the *move()* method must award any remaining bonus points to the player and return a value of `GWSTATUS_FINISHED_LEVEL`.
2. It must then delete any actors that have died during this tick (e.g., a zombie pedestrian that was dissolved by holy water and so should be removed from the game world, or a goodie that disappeared because it overlapped with Ghost Racer and activated, or an actor that disappeared off the bottom or sides of the screen because Ghost Racer drove past it, etc.).
3. It must then add any new objects to the game (e.g., a new goodie, oil slick, zombie or human ped, yellow or white road border line, or vehicle) as required for proper gameplay.
4. It must update the status text on the top of the screen with the latest information (e.g., the user's current score, the number of lost souls that need to be saved, the number of holy water sprays that Ghost Racer has, the current level number, etc.).

The *move()* method must return one of three different values when it returns at the end of each tick (all are defined in *GameConstants.h*):

```
GWSTATUS_PLAYER_DIED
GWSTATUS_CONTINUE_GAME
GWSTATUS_FINISHED_LEVEL
```

The first return value, `GWSTATUS_PLAYER_DIED`, indicates that Ghost Racer died during the current tick, and instructs our provided framework code to tell the user the bad news and restart the level if the player has more lives left (or end the game if they are out of lives). If your *move()* method returns this value and Ghost Racer has more lives left, then our framework will prompt the player to continue the game, call your *cleanUp()* method to destroy the level, call your *init()* method to re-initialize the roadway from scratch, and then begin calling your *move()* method over and over, once per tick, to let the user play the level again.

The second return value, `GWSTATUS_CONTINUE_GAME`, indicates that the tick completed without Ghost Racer dying BUT Ghost Racer has not yet completed the current level.

Therefore, the gameplay should continue normally for the time being. In this case, the framework will advance to the next tick and call your *move()* method again.

The final return value, `GWSTATUS_FINISHED_LEVEL`, indicates that the player has completed the current level (that is, Aisha successfully saved all of the required lost souls). If your *move()* method returns this value, then the current level is over, and our framework will call your *cleanUp()* method to destroy the level, our framework will then advance to the next level, then call your *init()* method to prepare that level for play, etc…

IMPORTANT NOTE: The skeleton code that we provide to you is hard-coded to return a `GWSTATUS_PLAYER_DIED` status value from our dummy version of the *move()* method. Unless you implement something that returns `GWSTATUS_CONTINUE_GAME` your game will not display any objects on the screen! So if the screen just immediately tells you that you lost a life once you start playing, you'll know why!

Here's pseudocode for how the move() method might be implemented:

```
int StudentWorld::move()
{
      // The term "actors" refers to all actors, Ghost Racer, pedestrians,
      // vehicles, goodies, oil slicks, holy water, spray, lost souls, etc.

     // Give each actor a chance to do something, including Ghost Racer
    for each of the actors in the game world
    {
        if (the actor is still active/alive)
        {
            // tell that actor to do something (e.g. move)
          the actor->doSomething();

          if (Ghost Racer was destroyed during this tick)
                return GWSTATUS_PLAYER_DIED;

          if (Ghost Racer completed the currentLevel)
          {
                add bonus points to the score
                return GWSTATUS_FINISHED_LEVEL;
          }
        }
    }

     // Remove newly-dead actors after each tick
    Remove and delete dead game objects

     // Potentially add new actors to the game
     // (e.g., oil slicks or goodies or border lines)
    Add new actors

     // Update the Game Status Line
    Update display text   // update the score/lives/level text at screen top

     // the player hasn't completed the current level and hasn't died, so
     // continue playing the current level
    return GWSTATUS_CONTINUE_GAME;
}
```

Give Each Actor a Chance to Do Something

During each tick of the game each active actor must have an opportunity to do something (e.g., move around, shoot, etc.). Actors include Ghost Racer, pedestrians, vehicles, projectiles like holy water (which are added to the game when Ghost Racer shoots), lost souls, goodies like healing goodies, oil slicks, and border lines (both yellow and white).

Your *move()* method must iterate over every actor that's active in the game (i.e., held by your *StudentWorld* object) and ask it to do something by calling a member function in the actor's object named perhaps *doSomething()*. In each actor's *doSomething()* method, the object will have a chance to perform some activity based on the nature of the actor and its current state: e.g.:

- A pedestrian might move four pixels left,
- A border line might move eight pixels down the screen,
- Ghost Racer might shoot holy water,
- A previously-sprayed holy water projectile may damage a nearby zombie ped, then disappear

It is possible that one actor (e.g., a holy water squirt) may destroy another actor (e.g., a zombie pedestrian or healing goodie) during the current tick. If an actor has died earlier in the current tick, then the dead actor must NOT have a chance to do something during the current tick (since it's dead). Also, other live actors processed during the tick must not interact with an actor after it has died (e.g., Ghost Racer should not get a bonus if it drives on top of a goodie that disappeared earlier during the same tick.).

To help you with testing, if you press the `f` key during the course of the game, our game controller will stop calling move() every tick; it will call move() only when you hit a key (except the `r` key). Freezing the activity this way gives you time to examine the screen, and stepping one move at a time when you're ready helps you see if your actors are moving properly. To resume regular gameplay, press the `r` key.

Remove Dead Actors After Each Tick

At the end of each tick, your *move()* method must determine which of your actors are no longer alive, remove them from your container of active actors, and use a C++ delete expression to free their objects (so you don't have a memory leak). So if, for example, a zombie pedestrian is killed by a spray of holy water and it dies, then it should be noted as dead, and at the end of the tick, its *pointer* should be removed from the *StudentWorld*'s container of active objects, and the pedestrian object should be deleted (using a C++ delete expression) to free up memory for future actors that will be introduced later in the game. Or, for example, after a holy water projectile has been fired and has travelled 160 pixels and is ready to dissipate, it must disappear from the screen and its object needs to be deleted as well. (Hint: Each of your actors could maintain a dead/alive status data member.)

Add New Actors As Required During Each Tick

During each tick of the game, after allowing any of your existing actors to do something, you may need to introduce one or more new actors into the game for Ghost Racer to interact with. Here are the rules for doing so on level L of the game:

Border Lines

1. Let new_border_y = VIEW_HEIGHT - SPRITE_HEIGHT
2. Let delta_y = new_border_y - the y coordinate of the last white border line that was added to the screen.
3. If delta_y >= SPRITE_HEIGHT then you must add new yellow border lines on either side of the road (at x=ROAD_CENTER - ROAD_WIDTH/2, and x= ROAD_CENTER + ROAD_WIDTH/2), both with a y value of new_border_y.
4. If delta_y >= 4*SPRITE_HEIGHT then you must add two new white border lines between the three lanes (at x = ROAD_CENTER - ROAD_WIDTH / 2 + ROAD_WIDTH/3, and x = ROAD_CENTER + ROAD_WIDTH / 2 - ROAD_WIDTH/3), both with a y value of new_border_y.

Zombie Cabs

1. ChanceVehicle = max(100 – L * 10, 20)
2. Generate a random integer between [0, ChanceVehicle)
3. If the random integer is 0, then try to add a new zombie cab to the road, as described just below:

Determining a Starting Position and Initial Movement Plan of a Zombie Cab

You must use the following algorithm to determine the starting location and initial vertical speed of a zombie cab, and for that matter, whether it should ultimately be added during the current tick:

1. Let cur_lane = a random candidate lane to start evaluating (left, middle, or right, chosen with equal probability) in which to start our new zombie cab.
2. Repeat the following up to three times (once for each lane):
    a. Determine the closest "collision avoidance-worthy actor" to the BOTTOM of the screen in candidate lane[3]. A collision avoidance-worthy actor is a pedestrian or vehicle of any type.
    b. If there is no such actor in the candidate lane, or there is such an actor and it has a Y coordinate that is greater than (VIEW_HEIGHT / 3) then:
        i. cur_lane is the chosen lane for the new vehicle
        ii. The start Y coordinate for this new vehicle will be SPRITE_HEIGHT / 2

[3] For this project, an actor is considered to be in a particular lane K if the center of the actor is on or to the right of the left boundary of lane K, and to the left of (but not on) the right boundary of lane K.

iii. The initial vertical speed of the new vehicle will be the Ghost Racer's vertical speed PLUS a random integer between 2 and 4 inclusive.
iv. Break out of the loop and proceed to step 3

c. Determine the closest "collision avoidance-worthy actor" to the TOP of the screen in the candidate lane
d. If there is no such actor in the candidate lane, or there is such an actor and it has a Y coordinate that is less than (VIEW_HEIGHT * 2 / 3) then:
   i. cur_lane is the chosen lane for the new vehicle
   ii. The start Y coordinate for this new vehicle will be VIEW_HEIGHT - SPRITE_HEIGHT / 2
   iii. The initial vertical speed of the new vehicle will be the Ghost Racer's vertical speed MINUS a random integer between 2 and 4 inclusive.
   iv. Break out of the loop and proceed to step 3
e. Otherwise, the current lane is too dangerous to add a new zombie cab (it would probably quickly result in a collision with Ghost Racer or some other actor that should be avoided), so set cur_lane to the next of the three lanes to check (you may check the three lanes in any order, so long as you check each lane just once)

3. If no viable lane was identified as the "chosen lane" to introduce our new zombie cab (because there are too many cabs/peds in every lane already), then we will not introduce a zombie cab during this tick. You can simply avoid adding a cab during this tick.
4. Otherwise, it means we found a lane and a position (top or bottom of the lane) to safely add a new zombie cab.
5. The start X coordinate for this new vehicle is the center of the chosen lane, which will be at ROAD_CENTER for the middle lane, or ROAD_CENTER - ROAD_WIDTH/3 for the left lane, or ROAD_CENTER + ROAD_WIDTH/3 for the right lane.
6. Create your new zombie cab with the appropriate starting x, y and vertical speed components computed above.

Oil Slicks

1. ChanceOilSlick = max(150 – L * 10, 40)
2. Generate a random integer between [0, ChanceOilSlick)
3. If the random integer is 0, then add a new oil slick at a random x position on the surface of the road, and at a y location of VIEW_HEIGHT.

Zombie Peds

1. ChanceZombiePed = max(100 – L * 10, 20)
2. Generate a random integer between [0, ChanceZombiePed)

3. If the random integer is 0, then add a new zombie ped to the road at a random x position between 0 and VIEW_WIDTH, and at a y location of VIEW_HEIGHT.

Human Peds

1. ChanceHumanPed = max(200 – L * 10, 30)
2. Generate a random integer between [0, ChanceHumanPed)
3. If the random integer is 0, then add a new human ped to the road at a random x position between 0 and VIEW_WIDTH, and at a y location of VIEW_HEIGHT.

Holy Water Refill Goodies

1. ChanceOfHolyWater = 100 + 10 * L
2. Generate a random integer between [0, ChanceOfHolyWater)
3. If the random integer is 0, then add a new bottle of holy water at a random x position on the surface of the road, and at a y location of VIEW_HEIGHT.

Lost Soul Goodies

1. ChanceOfLostSoul = 100
2. Generate a random integer between [0, ChanceOfLostSoul)
3. If the random integer is 0, then add a new lost soul at a random x position on the surface of the road, and at a y location of VIEW_HEIGHT.

cleanUp() Details

When your *cleanUp()* method is called by our game framework, it means that Ghost Racer lost a life (e.g., its hit points went to zero or below due to hitting too many zombie pedestrians and/or zombie cabs, by bumping into the sides of the freeway, or ran into a human pedestrian) or has completed the current level. In this case, every actor in the entire game (Ghost Racer and every ped, vehicle, goodie, projectile, soul, oil slick, border line, etc.) must be deleted and removed from the *StudentWorld's* container of active objects, resulting in an empty level. If the user has more lives left, our provided code will subsequently call your *init()* method to reload and repopulate the level with a new set of actors, and the level will then continue from scratch.

You must not call the *cleanUp()* method yourself when Ghost Racer dies. Instead, this method will be called by our code when *init()* returns an appropriate status.

You Have to Create the Classes for All Actors

Ghost Racer has a number of different actors, including:

- Ghost Racer
- Human pedestrians
- Zombie pedestrians
- Zombie cabs
- Oil slicks
- Holy water projectiles (fired by Aisha from Ghost Racer)
- Border lines (yellow and white)
- Lost souls
- Healing goodies
- Holy water bottle (refill) goodies

Each of these actor types can occupy your various levels and interact with other game actors within the visible screen view.

Now of course, many of your game actors will share things in common – for instance, every one of the actors in the game (pedestrians of both types, zombie cabs, Ghost Racer, oil slicks, holy water, etc.) has (x, y) coordinates. Many game actors have the ability to perform an action (e.g., move, damage another actor) during each tick of the game. Many of them can potentially be damaged (e.g., Ghost Racer, zombie peds, zombie cabs, holy water goodies, healing goodies) and could "die" during a tick. Certain objects like holy water, oil slicks, and goodies "activate" when they come into contact with a proper target:

- Goodies activate when they overlap with Ghost Racer and give it some special benefit,
- Holy water projectiles activate when they overlap with zombies and human peds,
- Oil slicks activate when Ghost Racer drives over them,
- Etc...

It is therefore your job to determine the commonalities between your different actor classes and make sure to factor out common behaviors and traits and move these into appropriate base classes, rather than duplicate these items across your derived classes – this is in fact one of the tenets of object-oriented programming.

Your grade on this project will depend upon your ability to intelligently create a set of classes that follow good object-oriented design principles. **Your classes must never duplicate code or a data member** – if you find yourself writing the same (or largely similar) code or duplicating data members across multiple classes, then this is an indication that you should define a common base class and migrate this common functionality/data to the base class. Duplication of code is a so-called *code smell*, a weakness in a design that often leads to bugs, inconsistencies, code bloat, etc.

Hint: When you notice this specification repeating the same text nearly identically in the following sections (e.g., in the healing goodie section and the holy water refill goodie section, or in the human and zombie pedestrian sections) you must make sure to identify common behaviors and move these into proper base classes. NEVER duplicate behaviors (aka methods and data members) across classes that can be moved into a base class!

You MUST derive all of your game objects directly or indirectly from a base class that we provide called *GraphObject*, e.g.:

```
class Actor: public GraphObject
{
public:
	…
};

class Pedestrian: public Actor
{
public:
	…
};

class Goodie: public Actor
{
public:
	…
};
```

GraphObject is a class that we have defined that helps hide the ugly logic required to graphically display your actors on the screen. If you don't derive your classes from our *GraphObject* base class, then you won't see anything displayed on the screen! ☺

The *GraphObject* class provides the following methods that you may use:

```
GraphObject(int imageID, double startX, double startY,
            int startDirection = 0, double size = 1.0, int depth = 0);
double getX() const;             // in pixels (0-255)
double getY() const;             // in pixels (0-255)
void moveTo(double x, double y); // in pixels (0-255)
  // moveForward() moves the actor the specified number of units in the
  // direction it is facing.
void moveForward(int units = 1);
  // getPositionInThisDirection() returns a new (x, y) location in the
  // specified direction and distance, based on the passed-in angle and the
  // GraphObject's current (x, y) location.
void getPositionInThisDirection(int angle, int units,
                                                double& dx, double& dy);
int getDirection() const;        // in degrees (0-359)
void setDirection(int d);        // in degrees (0-359)
double getRadius() const;        // in pixels
```

You may use any of these member functions in your derived classes, but you **must not** use any other member functions found inside of *GraphObject* in your other classes (even

if they are public in our class). You **must not** redefine any of these methods in your derived classes **unless** they are marked as virtual in our base class.

```
GraphObject(int imageID,
            int startX,          // column first
            int startY,          // then row!
            int startDirection,
            double size,
            int depth = 0)
```

is the constructor for a new *GraphObject*. When you construct a new *GraphObject*, you must specify an image ID that indicates how the *GraphObject* should be displayed on screen (e.g., as a human pedestrian, zombie pedestrian, zombie cab, Ghost Racer, an oil slick, a holy water refill goodie, etc.). You must also specify the initial (x, y) location of the object. The x value may range from 0 to VIEW_WIDTH-1 inclusive, and the y value may range from 0 to VIEW_HEIGHT-1 inclusive (these constants are defined in our provided header file *GameConstants.h*). Notice that you pass the coordinates as x, y (i.e., column, row starting from bottom left, and ***not*** row, column). You may also specify the initial direction an object is facing as an angle between 0-359 degrees. Finally, you may specify the depth of the object (which defaults to 0 if you don't). An object of depth 0 is in the foreground, whereas objects with increasing depths are drawn further in the background. Thus, an object of depth zero always covers object with a depth of one or greater, and an object with a depth of one always covers objects of depth two or greater, etc. In Ghost Racer, all road border lines and goodies are at depth 2, holy water projectiles are at depth 1, and all other game actors are at depth 0, ensuring our active characters are in the foreground.

One of the following IDs, found in *GameConstants.h*, must be passed in for the *imageID* value:

IID_GHOST_RACER
IID_YELLOW_BORDER_LINE
IID_WHITE_BORDER_LINE
IID_OIL_SLICK
IID_HUMAN_PED
IID_ZOMBIE_PED
IID_ZOMBIE_CAB
IID_HOLY_WATER_PROJECTILE
IID_HEAL_GOODIE
IID_SOUL_GOODIE
IID_HOLY_WATER_GOODIE

If you derive your game objects from our *GraphObject* class, they will be displayed on screen automatically by our framework (e.g., a human pedestrian image will be drawn to the screen at the *GraphObject's* specified x,y coordinates if the object's Image ID is IID_HUMAN_PED).

The classes you write MUST NOT store an imageId value or any value somehow related/derived from the imageID value in any way or you will get a Zero on this project. Only our GraphObject class may store the imageId value.

You MUST NOT use the imageID to identify object types, e.g., determine that a particular actor is a zombie pedestrian by checking its image ID is IID_ZOMBIE_PED). Nor may you store other similar data (e.g., a string "zombie_ped") based on the image ID to identify your object types. For hints on how to distinguish between different actors, see the hints section later in this document!

getX() and *getY()* are used to determine a *GraphObject's* current location in the level. Since each *GraphObject* maintains its own (x, y) location, this means that your derived classes **must NOT** also have x or y data members, but instead use these functions and *moveTo()* from the *GraphObject* base class.

moveTo(double x, double y) is used to update the location of a *GraphObject* within the level. For example, if a Pedestrian's movement logic dictates that it should move one pixel to the left, you could do the following:

```
moveTo(getX()-1, y);       // move one pixel to the left
```

moveForward(int units) is used to update the location of a *GraphObject* by moving it the specified number of pixels in the object's current direction. For example, if a holy water squirt movement logic dictates that it should move 6 pixels forward, you could do the following:

```
moveForward(6);            // move 6 pixels in squirt's current direction
```

You ***must*** use the *moveTo()* or *moveForward()* methods to adjust the location of a game object if you want that object to be properly animated. As with the *GraphObject* constructor, note that the order of the parameters to *moveTo* is x,y (col, row) and NOT y, x (row,col).

getDirection() is used to determine the direction a *GraphObject* is facing, and returns a value of *0-359*.

setDirection(`int` *d)* is used to change the direction a *GraphObject* is facing and takes a value between 0 and 359. For example, you could use this method and *getDirection()* to adjust the direction of an actor when it decides to move in a new direction. Note that an actor can be facing in one direction, but move in a totally different direction using *moveTo()*. The angle of movement and the angle the actor is facing are allowed to be totally different.

getRadius() is used to determine the radius of a *GraphObject*.

Ghost Racer

Here are the requirements you must meet when implementing the Ghost Racer class.

What a Ghost Racer Object Must Do When It Is Created

When it is first created:

1. A Ghost Racer object must have an image ID of IID_GHOST_RACER.
2. A Ghost Racer object starts out alive.
3. A Ghost Racer object has a direction (that it faces) of 90 degrees (upward).
4. A Ghost Racer object has a starting position on the roadway at x=128, y=32.
5. A Ghost Racer object has a size of 4.0.
6. A Ghost Racer object has a graphical depth of 0.
7. A Ghost Racer object starts out with a vertical speed of 0.
8. A Ghost Racer object starts with 10 units of holy water spray.
9. A Ghost Racer object starts out with 100 hit points (health).
10. A Ghost Racer object is a "collision avoidance-worthy actor".

What a Ghost Racer Object Must Do During a Tick

Ghost Racer must be given an opportunity to do something during every tick (in its *doSomething()* method). When given an opportunity to do something, Ghost Racer must do the following:

1. If the Ghost Racer is not currently alive (i.e., its hit points are zero or less), its *doSomething()* method must return immediately – none of the following steps should be performed.
2. If the Ghost Racer's center X coordinate is less than or equal to the left road boundary (meaning it's swerving off the road), it must:
    a. Check if the GhostRacer is still facing toward the left side of the road (has an angle > 90 degrees) and if so, *damage* itself by 10 hit points. See the What a Ghost Racer Object Must Do In Other Circumstances section below for details on what it means to "damage" the Ghost Racer.
    b. Change its direction to 82 degrees (pointing away from the left road boundary)
    c. Play a sound of SOUND_VEHICLE_CRASH
    d. Go to step 5.
3. If the Ghost Racer's center X coordinate is greater than or equal to the right road boundary, it must:
    a. Check if the GhostRacer is still facing toward the right side of the road (has an angle < 90 degrees) and if so, *damage* itself by 10 hit points. See the What Ghost Racer Must Do In Other Circumstances section below for details on what it means to "damage" the Ghost Racer.
    b. Change its direction to 98 degrees (pointing away from the right road boundary)

c. Play a sound of SOUND_VEHICLE_CRASH
d. Go to step 5.

4. The Ghost Racer must check to see if the player pressed a key (the section below shows how to check this). If the player pressed a key:
    a. If the player pressed the Space key and Ghost Racer has at least one unit of holy water left, then Ghost Racer will:
        i. Add a new holy water spray object SPRITE_HEIGHT pixels directly in front of the Ghost Racer (in the same direction Ghost Racer is facing) into their *StudentWorld* object. Hint: The cos() and sin() functions can be used to determine the proper delta_x and delta_y in front of Ghost Racer where to place the new holy water projectile.
        ii. Ghost Racer must play the SOUND_PLAYER_SPRAY sound effect (see the *StudentWorld* section of this document for details on how to play a sound).
        iii. Ghost Racer's spray count must decrease by 1.
        iv. Go to step 5.
    b. If the user asks to move left by pressing a directional key AND Ghost Racer's current angle is less than 114 degrees:
        i. Increase Ghost Racer's current directional angle 8 degrees.
        ii. Go to step 5.
    c. If the user asks to move right by pressing a directional key AND Ghost Racer's current angle is greater than 66 degrees:
        i. Decrease Ghost Racer's current directional angle 8 degrees.
        ii. Go to step 5.
    d. If the user asks to move faster by pressing an "up" directional key, and Ghost Racer's vertical speed is less than 5, then increase Ghost Racer's vertical speed by 1 and go to step 5.
    e. If the user asks to move slower by pressing a "down" directional key, and Ghost Racer's vertical speed is greater than -1, then decrease Ghost Racer's vertical speed by 1 and go to step 5.
5. Ghost Racer must attempt to move using the *Ghost Racer Movement Algorithm* described in the section below.

<u>Ghost Racer Movement Algorithm</u>

When attempting to move as described in the section above, the Ghost Racer must use this algorithm:

1. Let max_shift_per_tick = 4.0
2. Let direction = Ghost Racer's current direction (from getDirection())
3. Let delta_x = cos(direction) * max_shift_per_tick
4. Let cur_x = Ghost Racer's current X position
5. Let cur_y = Ghost Racer's current Y position
6. moveTo(cur_x + delta_x, cur_y)

Notice that this algorithm only changes Ghost Racer's X position, leaving its Y position the same. This ensures that the player always sees the Ghost Racer at the bottom-middle of the screen, making gameplay easier. The X position changes in proportion to Ghost Racer's angle, just like a real car. If its angle is close to 90 degrees, then its delta_x will be close to zero, causing it not to shift too far left or right. Conversely if its angle is closer to 0 or 180 degrees, this will cause it to shift up to 4 pixels right or left respectively (per tick).

What Ghost Racer Must Do In Other Circumstances

Hint: The following actions can be performed on Ghost Racer by other actors in the game (e.g., an oil slick might spin Ghost Racer around). Each of these bullets might be implemented by a separate method inside the Ghost Racer class (or perhaps one of its base classes).

- Ghost Racer is not affected by projectiles like holy water
- Ghost Racer can be damaged (e.g., by overlapping with a zombie pedestrian or cab, or by bumping into the left or right edges of the roadway), reducing Ghost Racer's hit points. Here's what you must do in this case:
    - Ghost Racer must lower its hit points based on the amount of damage specified in the function call to damage Ghost Racer.
    - If Ghost Racer hit points reach zero (or below), the Ghost Racer object must:
        - Immediately have its status set to not-alive.
        - It must play a SOUND_PLAYER_DIE sound effect.
        - (The *StudentWorld* class should later detect Ghost Racer's death and the current level ends)
- Ghost Racer can be spun around (by driving over an oil slick). If an oil slick tries to spin Ghost Racer, Ghost Racer must adjust its direction by a random integer between [5, 20] degrees clockwise or counterclockwise of its current direction. Ghost Racer's angle must never go below 60 degrees or above 120 degrees.
- Ghost Racer can be healed by a specified number of hit points, up to its maximum hit points of 100 (it cannot exceed 100 hit points).
- Ghost Racer can have holy water added to its holy water tank. There is no limit to the amount of holy water the Ghost Racer can hold.

Getting Input From the User

Since Ghost Racer is a *real-time* game, you can't use the typical *getline* or *cin* approach to get a user's key press within Ghost Racer's *doSomething()* method— that would stop your program and wait for the user to type something and then hit the Enter key. This would make the game awkward to play, requiring the user to hit a directional key then hit Enter, then hit a directional key, then hit Enter, etc. Instead of this approach, you will use a function called *getKey()* that we provide in our *GameWorld* class (from which your *StudentWorld* class is derived) to get input from the player[4]. This function rapidly checks to see if the user has hit a key. If so, the function returns true and the int variable passed to it is set to the code for the key. Otherwise, the function immediately returns false, meaning that no key was hit. This function could be used as follows:

```
void Ghost Racer::doSomething()
{
    ...
    int ch;
    if (getWorld()->getKey(ch))
    {
        // user hit a key during this tick!
        switch (ch)
        {
          case KEY_PRESS_LEFT:
            ... move Ghost Racer counterclockwise ...;
            break;
          case KEY_PRESS_RIGHT:
            ... move Ghost Racer clockwise...;
            break;
          case KEY_PRESS_SPACE:
            ... add spray in front of Ghost Racer...;
            break;

          // etc…
        }
    }
    ...
}
```

Border Line

Border lines don't really do much. They just move down the screen at a speed that's dictated by our Ghost Racer's vertical speed. Since Ghost Racer doesn't move vertically (it always stays at the same Y=32 position), what gives the appearance of Ghost Racer's movement is the yellow and white border lines moving down the screen. The faster Ghost Racer's "speed," the faster the road border lines must move down the screen to create an illusion of faster driving.

[4] Hint: Since your Ghost Racer class will need to access the *getKey()* method in the *GameWorld* class (which is the base class for your *StudentWorld* class), your Ghost Racer class (or more likely, one of its base classes) will need a way to obtain a pointer to the *StudentWorld* object it's playing in. If you look at our code example, you'll see how Ghost Racer's *doSomething()* method first gets a pointer to its world via a call to *getWorld()* (a method in one of its base classes that returns a pointer to a *StudentWorld*), and then uses this pointer to call the *getKey()* method.

What a Border Line Must Do When It Is Created

When it is first created:

1. A border line object must have an image ID of either IID_YELLOW_BORDER_LINE or IID_WHITE_BORDER_LINE depending on whether it's yellow or white.
2. A border line object must have its (x, y) location specified for it as detailed in the *StudentWorld::init()* section of this document. *Hint: A StudentWorld object can pass in this (x, y) location when constructing a border line object.*
3. A border line object has a direction of 0 degrees.
4. A border line object has a graphical depth of 2 (ensuring border lines are rendered behind other objects like the Ghost Racer).
5. A border line object has a size of 2.0.
6. A border line has a vertical speed of -4 (meaning they travel four pixels down the screen during every tick, until they reach the bottom of the screen).
7. A border line has a horizontal speed of zero.
8. A border line object starts out in the alive state.
9. A border line object has no hit points; it is either fully alive or fully dead.
10. A border line is not a "collision avoidance-worthy actor".[5]

What a Border Line Must Do During a Tick

A border line must be given an opportunity to do something during every tick (in its *doSomething()* method). When given an opportunity to do something during a tick, the border line must move itself (that's all that border lines do, they move down the screen as the Ghost Racer drives forward). You must use this algorithm to move the border line:

1. Let vert_speed = the border line's current vertical speed - Ghost Racer's current vertical speed (yes, you'll need to somehow ask the Ghost Racer object for its vertical speed, and use this to compute vert_speed)
2. Let horiz_speed = the border line's horizontal speed
3. Let new_y = border line's current y + vert_speed
4. Let new_x = border line's current x + horiz_speed
5. Adjust the border line's location to new_x, new_y using the GraphObject::moveto() method.
6. If the border line has gone off of the screen (either its X or Y coordinate is less than zero, or its X coordinate is > VIEW_WIDTH, or its Y coordinate > VIEW_HEIGHT), it must set its status to not-alive, so it will be removed by *StudentWorld* later in this tick. It must then immediately return.

[5] As you'll learn zombie cabs will only drive onto a lane if there are no collision avoidance-worthy actors close by in that lane. Obviously, zombie cabs don't need to check if there are any border lines nearby, since you can't collide with a border line.

What a Border Line Must Do In Other Circumstances

- A border line is not affected by projectiles like holy water (the projectile will just pass over a border line).
- A border line cannot be healed by a specified number of hit points.
- A border line cannot be spun around by an oil slick.

Human Pedestrians

You must create a class to represent human pedestrian actors. Here are the requirements you must meet when implementing the human pedestrian class.

What a Human Pedestrian Must Do When It Is Created

When it is first created:

1. A human pedestrian object must have an image ID of IID_HUMAN_PED.
2. A human pedestrian must always start at the proper location as passed to its constructor.
3. A human pedestrian has a direction of 0 degrees.
4. A human pedestrian has a size of 2.0.
5. A human pedestrian has a depth of 0.
6. A human pedestrian starts with a *movement plan distance* of 0, meaning it starts with no plan to move in its current direction.
7. A human pedestrian has a starting vertical speed of -4.
8. A human pedestrian has a starting horizontal speed of 0.
9. A human pedestrian starts out in an "alive" state.
10. A human pedestrian starts with 2 hit points.
11. A human pedestrian is always a "collision avoidance-worthy actor".

What a Human Pedestrian Must Do During a Tick

A human pedestrian must be given an opportunity to do something during every tick (in its *doSomething()* method). When given an opportunity to do something during a tick, the human pedestrian must:

1. If the human pedestrian is not currently alive, its *doSomething()* method must return immediately – none of the following steps should be performed.
2. If the human pedestrian overlaps with the Ghost Racer, then the player loses a life and the level ends (you must communicate this somehow to your *StudentWorld*). The human pedestrian's *doSomething()* method must then immediately return.
3. The human pedestrian must move, using the following algorithm:
    a. Let vert_speed = the human ped's current vertical speed - Ghost Racer's current vertical speed
    b. Let horiz_speed = the human ped's horizontal speed

    c. Let new_y = human ped's current y + vert_speed
    d. Let new_x = human ped's current x + horiz_speed
    e. Adjust the human ped's location to new_x, new_y using the GraphObject::moveTo() method.
    f. If the human pedestrian has gone off of the screen (either its X or Y coordinate is less than zero, or its X coordinate is > VIEW_WIDTH, or its Y coordinate > VIEW_HEIGHT), it must set its status to not-alive, so it will be removed by *StudentWorld* later in this tick. It must then immediately return.
4. Decrement the human pedestrian's movement plan distance.
5. If the distance is greater than zero, then immediately return.
6. Otherwise, it's time to pick a new movement plan for the human pedestrian:
    a. Set the human pedestrian's horizontal speed to a random integer from -3 to 3, inclusive, NOT including zero.
    b. Set the length of the movement plan to a random integer between 4 and 32, inclusive.
    c. Set the human pedestrian's direction to 180 degrees if its new horizontal speed is less than zero. Otherwise set its direction to 0 degrees if its new horizontal speed is greater than zero.

What Human Pedestrian Must Do In Other Circumstances

- A human pedestrian is affected by holy water. When damaged by holy water the human pedestrian will reverse its direction:
    - Ignore all hit point damage (holy water doesn't actually injure the pedestrian)
    - Change its current horizontal speed by multiplying it by -1 (e.g., if the ped was going left at 2 pixels/tick, it must change its direction to right at 2 pixels/tick).
    - Change the direction the human is facing (e.g., from 0 to 180 degrees, or 180 degrees to 0) as appropriate.
    - The human pedestrian must play a sound of SOUND_PED_HURT.
- A human pedestrian cannot be damaged by any collisions with zombie peds or zombie cabs. The zombies will simply pass over them.
- A human pedestrian cannot be spun around by an oil slick.
- A human pedestrian cannot be healed by a specified number of hit points.

Zombie Pedestrian

You must create a class to represent zombie pedestrian actors. Zombie pedestrians behave much like human pedestrians, in that they mostly wander aimlessly. However, if they ever wander in front of the Ghost Racer, they will turn to face it and walk toward it, grunting. See the details below. Here are the requirements you must meet when implementing the zombie pedestrian class.

What a Zombie Pedestrian Must Do When It Is Created

When it is first created:

1. A zombie pedestrian object must have an image ID of IID_ZOMBIE_PED.
2. A zombie pedestrian must always start at the proper location as passed into its constructor.
3. A zombie pedestrian has a direction of 0 degrees.
4. A zombie pedestrian has a size of 3.0.
5. A zombie pedestrian has a depth of 0.
6. A zombie pedestrian starts with a *movement plan distance* of 0, meaning it has no plan to move in its current direction.
7. A zombie pedestrian has a starting vertical speed of -4.
8. A zombie pedestrian has a starting horizontal speed of 0.
9. A zombie pedestrian starts out in an "alive" state.
10. A zombie pedestrian starts with 2 hit points.
11. A zombie pedestrian starts with zero ticks until it grunts next.
12. A zombie pedestrian is always a "collision avoidance-worthy actor".

What a Zombie Pedestrian Must Do During a Tick

A zombie pedestrian must be given an opportunity to do something during every tick (in its *doSomething()* method). When given an opportunity to do something during a tick, the zombie pedestrian must:

1. If the zombie pedestrian is not currently alive, its *doSomething()* method must return immediately – none of the following steps should be performed.
2. If the zombie pedestrian overlaps with the Ghost Racer:
    a. The Ghost Racer must receive 5 points of damage.
    b. The zombie pedestrian will be damaged[6] with 2 hit points of damage. See the Other Circumstances section below for what it means to damage a zombie pedestrian.
    c. The zombie pedestrian must immediately return and do nothing else.
3. If the zombie pedestrian's X coordinate is within 30 pixels of Ghost Racer's X coordinate, either left or right, AND the zombie pedestrian is in front of the Ghost Racer on the road then:
    a. The zombie pedestrian will set its direction to 270 degrees (facing down).
    b. If the zombie ped is to the left of Ghost Racer, it will set its horizontal speed to 1, else if the zombie ped is to the right of Ghost Racer, it will set its horizontal speed to -1. Otherwise (they have the same X coordinate), it will set its horizontal speed to zero.
    c. The zombie ped decreases the number of ticks before it grunts next.

[6] It's your choice which way you handle this interaction: You could have the zombie pedestrian damage itself if it detects a collision with Ghost Racer, or have the Ghost Racer damage the zombie pedestrian if it detects the collision. We have no preference.

    d. If the ticks until its next grunt is less than or equal to zero, the zombie ped must:
        i. Play a sound of SOUND_ZOMBIE_ATTACK.
        ii. Reset the number of ticks before its next grunt to 20.
4. The zombie pedestrian must then move, using the following algorithm:
    a. Let vert_speed = the zombie ped's current vertical speed - Ghost Racer's current vertical speed
    b. Let horiz_speed = the zombie ped's horizontal speed
    c. Let new_y = zombie ped's current y + vert_speed
    d. Let new_x = zombie ped's current x + horiz_speed
    e. Adjust the zombie ped's location to new_x, new_y using the GraphObject::moveTo() method.
    f. If the zombie pedestrian has gone off of the screen (either its X or Y coordinate is less than zero, or its X coordinate is > VIEW_WIDTH, or its Y coordinate > VIEW_HEIGHT), it must set its status to not-alive, so it will be removed by *StudentWorld* later in this tick. It must then immediately return.
5. If the zombie pedestrian's movement plan distance is greater than zero, then decrement it by one and immediately return.
6. Otherwise, it's time to pick a new movement plan for the zombie pedestrian:
    a. Set the zombie pedestrian's horizontal speed to a random integer from -3 to 3 inclusive, NOT including zero.
    b. Set the length of the movement plan to a random integer between 4 and 32, inclusive.
    c. Set the zombie pedestrian's direction to 180 degrees if its new horizontal speed is less than zero. Otherwise, set its direction to 0 degrees if its new horizontal speed is greater than zero.

<u>What Zombie Pedestrian Must Do In Other Circumstances</u>

- A zombie pedestrian is affected by projectiles like holy water (the projectile will stop when it hits a zombie pedestrian).
- When damaged (e.g., by holy water sprays), a zombie pedestrian must do the following:
    - It must reduce its hit points by the specified amount of damage hit points.
    - If its hit points reach zero or below, the zombie pedestrian must:
        - Set its status to not-alive, so it will be removed by *StudentWorld* later in this tick.
        - Play a sound of SOUND_PED_DIE.
        - If the zombie ped does not currently overlap with Ghost Racer (i.e., it didn't die due to Ghost Racer colliding with it), then there is a 1 in 5 chance that the zombie ped will add a new healing goodie at its current position.
        - Ensure the player receives 150 points.
    - Otherwise (the zombie ped still has hit points left), play a sound of SOUND_PED_HURT.

- A zombie pedestrian cannot be damaged by collisions with any actors other than the Ghost Racer. The zombie pedestrian will simply pass all other actors collision-free.
- A zombie pedestrian cannot be spun around by an oil slick.
- A zombie pedestrian cannot be healed by a specified number of hit points.

Zombie Cab

You must create a class to represent zombie cab actors. Here are the requirements you must meet when implementing the zombie cab class.

What a Zombie Cab Must Do When It Is Created

When it is first created:

1. A zombie cab object must have an image ID of IID_ZOMBIE_CAB.
2. A zombie cab must always start at the proper location as passed into its constructor.
3. A zombie cab has an initial direction of 90 degrees.
4. A zombie cab has a size of 4.0.
5. A zombie cab has a depth of 0.
6. A zombie cab has a horizontal speed of 0.
7. A zombie cab starts with 3 hit points.
8. A zombie cab starts with a plan length of 0.
9. A zombie cab starts out in a state where it has not yet damaged the Ghost Racer (it must track whether or not it has damaged the Ghost Racer yet).
10. A zombie cab is always a "collision avoidance-worthy actor".

What a Zombie Cab Must Do During a Tick

A zombie cab must be given an opportunity to do something during every tick (in its *doSomething()* method). When given an opportunity to do something during a tick, the zombie cab must:

1. If the zombie cab is not currently alive, its *doSomething()* method must return immediately – none of the following steps should be performed.
2. If the zombie cab overlaps with the Ghost Racer:
    a. If the zombie cab has already damaged Ghost Racer, it must immediately skip to step 3.
    b. Otherwise…
    c. Play a sound of SOUND_VEHICLE_CRASH.
    d. The zombie cab must do 20 points of damage to the Ghost Racer (see Ghost Racer's section on what it does when it's damaged).

    e. If the zombie cab is to the left of the Ghost Racer or has the same X coordinate as Ghost Racer then the zombie cab must:
        i. Set its horizontal speed to -5.
        ii. Set its direction equal to: 120 degrees plus a random integer between [0,20).
    f. If the zombie cab is right of the Ghost Racer then it must:
        i. Set its horizontal speed to 5.
        ii. Set its direction equal to: 60 degrees minus a random integer between [0,20).
    g. The zombie cab must remember that it has now damaged Ghost Racer (so it doesn't repeat the above steps for this zombie cab during subsequent ticks, even if the cab still overlaps with the Ghost Racer).
3. The zombie cab must then move, using the following algorithm:
    a. Let vert_speed = the zombie cab's current vertical speed - Ghost Racer's current vertical speed
    b. Let horiz_speed = the zombie cab's horizontal speed
    c. Let new_y = zombie cab's current y + vert_speed
    d. Let new_x = zombie cab's current x + horiz_speed
    e. Adjust the zombie cab's location to new_x, new_y using the GraphObject::moveTo() method.
    f. If the zombie cab has gone off of the screen (either its X or Y coordinate is less than zero, or its X coordinate is > VIEW_WIDTH, or its Y coordinate > VIEW_HEIGHT), it must set its status to not-alive, so it will be removed by *StudentWorld* later in this tick. It must then immediately return.
4. If the zombie cab's vertical speed is greater than Ghost Racer's vertical speed (so the cab is moving up the screen) and there is a "collision-avoidance worthy" actor in the zombie cab's lane that is in front of that zombie cab:
    a. If the closest such actor is less than 96 vertical pixels in front of the zombie cab, decrease the zombie cab's vertical speed by .5 and immediately return.
5. If the zombie cab's vertical speed is the same as or slower than Ghost Racer's vertical speed (so the cab is moving down the screen or holding steady with Ghost Racer) and there is a "collision-avoidance worthy" actor in the zombie cab's lane that is behind that zombie cab:
    a. If the closest such actor is less than 96 vertical pixels behind the zombie cab and is not Ghost Racer, increase the zombie cab's vertical speed by .5 and immediately return.
6. Decrement the zombie cab's movement plan distance by one.
7. If the zombie cab's movement plan distance is greater than zero, then immediately return.
8. Otherwise, it's time to pick a new movement plan for the zombie cab:
    a. Set the length of the zombie cab's movement plan to a random integer between 4 and 32, inclusive.
    b. Set the zombie cab's vertical speed to its vertical speed + a random integer between -2 and 2, inclusive.

What Zombie Cab Must Do In Other Circumstances

- A zombie cabs is affected by projectiles like holy water (the projectile will stop when it hits a zombie cab).
- When damaged (e.g., by a squirt of holy water), a zombie cab does the following:
    - It must reduce its hit points by the specified amount of hit points.
    - If its hit points reach zero or below, the zombie cab must:
        - Set its status to not-alive, so it will be removed by *StudentWorld* later in this tick.
        - Play a sound of SOUND_VEHICLE_DIE.
        - There is a 1 in 5 chance that the zombie cab will add a new oil slick at its current position.
        - Ensure the player receives 200 points.
        - Immediately return.
    - Otherwise (the zombie cab still has hit points left), play a sound of SOUND_VEHICLE_HURT.
- A zombie cabs cannot be damaged by collisions. The zombie cab will simply pass by all other actors collision-free.
- A zombie cabs cannot be spun around by an oil slick.
- A zombie cabs cannot be healed by a specified number of hit points.

Oil Slick

You must create a class to represent an oil slick. An oil slick will cause the Ghost Racer to skid if it drives on it. Oil slicks are introduced by *StudentWorld* randomly on the road, and also sometimes when zombie cabs are destroyed by holy water.

What an Oil Slick Must Do When It Is Created

When it is first created:

1. An oil slick object must have an image ID of IID_OIL_SLICK.
2. An oil slick's starting location will be specified during its construction.
3. An oil slick has a direction of 0 degrees.
4. An oil slick has a random size between 2 and 5, inclusive.
5. An oil slick has a depth of 2.
6. An oil slick has a horizontal speed of 0.
7. An oil slick has a vertical speed of -4.
8. An oil slick is NOT a "collision avoidance-worthy actor".

What an Oil Slick Must Do During a Tick

An oil slick must be given an opportunity to do something during every tick (in its *doSomething()* method). When given an opportunity to do something during a tick, the oil

slick must move itself (that's mostly what oil slicks do, they move down the screen as the Ghost Racer drives forward) and also check to see if it overlaps with Ghost Racer, and if so, cause it to skid. Since an oil skid may overlap with Ghost Racer for more than one tick, every tick where there's an overlap with Ghost Racer, the oil skid must cause it to skid. You must use this algorithm to move the oil slick:

1. Let vert_speed = the oil slick's vertical speed - Ghost Racer's current vertical speed
2. Let horiz_speed = the oil slick's horizontal speed (always zero)
3. Let new_y = oil slick's current y + vert_speed
4. Let new_x = oil slick's current x + horiz_speed
5. Adjust the oil slick's location to new_x, new_y using the GraphObject::moveTo() method
6. If the oil slick has gone off of the screen (either its X or Y coordinate is less than zero, or its X coordinate is > VIEW_WIDTH, or its Y coordinate > VIEW_HEIGHT), it must set its status to not-alive, so it will be removed by *StudentWorld* later in this tick. It must then immediately return.
7. If the oils slick overlaps with the Ghost Racer, the oil slick must:
    a. Play a sound of SOUND_OIL_SLICK.
    b. Tell the Ghost Racer to spin itself (see Ghost Racer section for details on how it should spin itself).

What an Oil Slick Must Do In Other Circumstances

- An oil slick is NOT affected by projectiles like holy water (the projectile just passes right over an oil slick).
- An oil slick cannot be damaged.
- An oil slick cannot be spun around by an oil slick.
- An oil slick cannot be healed by a specified number of hit points.

Healing Goodie

You must create a class to represent a healing goodie. A healing goodie will cause the Ghost Racer to be healed and receive points, if it overlaps with it. Healing goodies are introduced sometimes when zombie pedestrians are destroyed by holy water.

What a Healing Goodie Must Do When It Is Created

When it is first created:

1. A healing goodie object must have an image ID of IID_HEAL_GOODIE.
2. A healing goodie's starting location will be specified during its construction.
3. A healing goodie has a direction of 0 degrees.
4. A healing goodie has a size of 1.0.
5. A healing goodie has a depth of 2.

6. A healing goodie has a horizontal speed of 0.
7. A healing goodie has a vertical speed of -4.
8. A healing goodie is NOT a "collision avoidance-worthy actor".

What a Healing Goodie Must Do During a Tick

A healing goodie must be given an opportunity to do something during every tick (in its *doSomething()* method). When given an opportunity to do something during a tick, the healing goodie must move itself (that's mostly what healing goodies do, they move down the screen as the Ghost Racer drives forward) and also check to see if it overlaps with Ghost Racer, and if so, increase its hit points and disappear. You must use this algorithm to move the healing goodie:

1. Let vert_speed = the healing goodie's vertical speed - Ghost Racer's current vertical speed
2. Let horiz_speed = the healing goodie's horizontal speed (always zero)
3. Let new_y = healing goodie's current y + vert_speed
4. Let new_x = healing goodie's current x + horiz_speed
5. Adjust the healing goodie's location to new_x, new_y using the GraphObject::moveTo() method.
6. If the healing goodie has gone off of the screen (either its X or Y coordinate is less than zero, or its X coordinate is > VIEW_WIDTH, or its Y coordinate > VIEW_HEIGHT), it must set its status to not-alive, so it will be removed by *StudentWorld* later in this tick. It must then immediately return.
7. If the healing goodie overlaps with the Ghost Racer, the healing goodie must:
    a. Tell the Ghost Racer to heal itself by 10 hit points.
    b. Set its status to not-alive, so it will be removed by *StudentWorld* later in this tick.
    c. Play a sound of SOUND_GOT_GOODIE.
    d. Increase the player's score by 250 points.

What a Healing Goodie Must Do In Other Circumstances

- A healing goodie is affected by projectiles like holy water (the spray will stop when it overlaps with the healing goodie and damages it).
- A holy water spray shot by the Ghost Racer that overlaps with a healing goodie will damage it, causing it to be destroyed.
- A healing goodie cannot be spun around by an oil slick.
- A healing goodie cannot be healed by a specified number of hit points.

Holy Water Goodie

You must create a class to represent a bottle of holy water that the Ghost Racer can pick up (by driving over it). to replenish its holy water tank. A holy water goodie will

replenish the Ghost Racer's holy water tank. These goodies are introduced by *StudentWorld* randomly during ticks. See the *StudentWorld* section for more details.

What a Holy Water Goodie Must Do When It Is Created

When it is first created:

1. A holy water goodie object must have an image ID of IID_HOLY_WATER_GOODIE.
2. A holy water goodie's starting location will be specified during its construction.
3. A holy water goodie has a direction of 90 degrees.
4. A holy water goodie has a size of 2.0.
5. A holy water goodie has a depth of 2.
6. A holy water goodie has a horizontal speed of 0.
7. A holy water goodie has a vertical speed of -4.
8. A holy water goodie is NOT a "collision avoidance-worthy actor".

What a Holy Water Goodie Must Do During a Tick

A holy water goodie must be given an opportunity to do something during every tick (in its *doSomething()* method). When given an opportunity to do something during a tick, the holy water goodie must move itself (that's mostly what holy water goodies do, they move down the screen as the Ghost Racer drives forward) and also check to see if it overlaps with Ghost Racer, and if so, add holy water to its tank and disappear. You must use this algorithm to move the holy water goodie:

1. Let vert_speed = the holy water goodie's vertical speed - Ghost Racer's current vertical speed
2. Let horiz_speed = the holy water goodie's horizontal speed (always zero)
3. Let new_y = holy water goodie's current y + vert_speed
4. Let new_x = holy water goodie's current x + horiz_speed
5. Adjust the holy water goodie's location to new_x, new_y using the GraphObject::moveTo() method.
6. If the holy water goodie has gone off of the screen (either its X or Y coordinate is less than zero, or its X coordinate is > VIEW_WIDTH, or its Y coordinate > VIEW_HEIGHT), it must set its status to not-alive, so it will be removed by *StudentWorld* later in this tick. It must then immediately return.
7. If the holy water goodie overlaps with the Ghost Racer, the holy water goodie must:
    a. Tell the Ghost Racer to increase its holy water charges by 10 charges.
    b. Set its status to not-alive, so it will be removed by *StudentWorld* later in this tick.
    c. Play a sound of SOUND_GOT_GOODIE
    d. Increase the player's score by 50 points.

What a Holy Water Goodie Must Do In Other Circumstances

- A holy water goodie is affected by projectiles like holy water (the spray will stop when it overlaps with the holy water goodie and damages it).
- A holy water spray shot by the Ghost Racer that overlaps with a holy water goodie will damage it, causing it to be destroyed.
- A holy water goodie cannot be spun around by an oil slick.
- A holy water goodie cannot be healed by a specified number of hit points.

Soul Goodie

You must create a class to represent a lost soul that needs to be rescued by Ghost Racer. A specified number of soul goodies must be picked up by Ghost Racer in order to complete each level. Picking up such a goodie also gives the player points. These goodies are introduced by *StudentWorld* randomly during ticks. See the *StudentWorld* section for more details.

What a Soul Goodie Must Do When It Is Created

When it is first created:

1. A soul goodie object must have an image ID of IID_SOUL_GOODIE.
2. A soul goodie's starting location will be specified during its construction.
3. A soul goodie has a direction of 0 degrees.
4. A soul goodie has a size of 4.0.
5. A soul goodie has a depth of 2.
6. A soul goodie has a horizontal speed of 0.
7. A soul goodie has a vertical speed of -4.
8. A soul goodie is NOT a "collision avoidance-worthy actor".

What a Soul Goodie Must Do During a Tick

A soul goodie must be given an opportunity to do something during every tick (in its *doSomething()* method). When given an opportunity to do something during a tick, the soul goodie must move itself (that's mostly what soul goodies do, they move down the screen as the Ghost Racer drives forward) and also check to see if it overlaps with Ghost Racer, and if so, disappear, counting as one of the souls saved for the current game level level. You must use this algorithm to move the soul goodie:

1. Let vert_speed = the soul goodie's vertical speed - Ghost Racer's current vertical speed
2. Let horiz_speed = the soul goodie's horizontal speed (always zero)
3. Let new_y = soul goodie's current y + vert_speed
4. Let new_x = soul goodie's current x + horiz_speed

5. Adjust the soul goodie's location to new_x, new_y using the GraphObject::moveTo() method.
6. If the soul goodie has gone off of the screen (either its X or Y coordinate is less than zero, or its X coordinate is > VIEW_WIDTH, or its Y coordinate > VIEW_HEIGHT), it must set its status to not-alive, so it will be removed by *StudentWorld* later in this tick. It must then immediately return.
7. If a soul goodie overlaps with the Ghost Racer, the soul goodie must:
    a. Increase the number of lost souls that have been saved by telling *StudentWorld* that a lost soul was just saved; this may result in the level being finished, which *StudentWorld* must communicate to our framework at the end of the tick).
    b. Set its status to not-alive, so it will be removed by *StudentWorld* later in this tick.
    c. Play a sound of SOUND_GOT_SOUL.
    d. Increase the player's score by 100 points.
8. The soul goodie must rotate itself by 10 degrees clockwise.

What a Soul Goodie Must Do In Other Circumstances

- A soul goodie is not affected by projectiles like holy water (the projectile just passes over a soul goodie).
- A soul goodie is not damageable.
- A soul goodie cannot be spun around by an oil slick.
- A soul goodie cannot be healed by a specified number of hit points.

Holy Water Projectile

You must create a class to represent sprayed holy water "projectiles" which Ghost Racer can use to attack zombie pedestrians and zombie cabs. Holy water projectile objects are produced from Ghost Racer's holy water cannon (when the player presses the spacebar) and travel up the screen. Here are the requirements you must meet when implementing the holy water projectile class.

What Holy Water Projectile Must Do When It Is Created

When it is first created:

1. A holy water projectile object must have an image ID of IID_HOLY_WATER_PROJECTILE.
2. The starting location of a holy water projectile must be specified during construction.
3. The starting direction of a holy water projectile must be specified during construction.
4. A holy water projectile starts out with a maximum travel distance of 160 pixels.
5. A holy water projectile has a size of 1.0.

6. A holy water projectile has a depth of 1.
7. A holy water projectile starts in the alive state.
8. A holy water projectile is NOT a “collision avoidance-worthy actor”.

What a Holy Water Projectile Must Do During a Tick

A holy water projectile must be given an opportunity to do something during every tick (in its *doSomething()* method). When given an opportunity to do something during a tick, the holy water projectile must:

1. If the holy water projectile is not currently alive, its *doSomething()* method must return immediately – none of the following steps should be performed.
2. Check to see if it has activated. A holy water projectile activates if an object that is affected by projectiles overlaps with it (e.g., zombie peds, zombie cabs, and some types of goodies). If multiple objects overlap with a holy water projectile, then you may pick any one and ignore the others. If the holy water projectile overlaps with an affected object, the holy water spray must:
    a. Attempt to damage the other object with 1 hit point of damage.
    b. Set its own status to not-alive, so it will be removed by *StudentWorld* later in this tick.
    c. Return immediately.
3. Otherwise, the holy water projectile will move forward in its current direction by SPRITE_HEIGHT pixels. You can use the *GraphObject::moveForward()* function to move the holy water based on its current directional angle.
4. If the holy water projectile has gone off of the screen (either its X or Y coordinate is less than zero, or its X coordinate is > VIEW_WIDTH, or its Y coordinate > VIEW_HEIGHT), it must set its status to not-alive, so it will be removed by *StudentWorld* later in this tick. It must then immediately return.
5. If the holy water projectile has moved a total of 160 pixels, then it immediately sets its status to not-alive (it dissipates) and will be removed by *StudentWorld* later in this tick.

What a Holy Water Must Do In Other Circumstances

- A holy water projectile is not affected by other projectiles.
- A holy water projectile cannot be damaged.
- A holy water projectile cannot be spun around.
- A holy water projectile cannot be healed by a specified number of hit points.

Object Oriented Programming Tips

Before designing your base and derived classes for Project 3 (or for that matter, any other school or work project), make sure to consider the following best practices. These tips will help you not only write a better object-oriented program, but also help you get a better grade on P3!

Try your best to leverage the following best practices in your program, but don't be overly obsessive – it's rarely possible to make a set of perfect classes. That's often a waste of time. Remember, the best is the enemy of the good (enough).

Here we go!

1. **You MUST NEVER use the imageID (e.g., IID_PLAYER, IID_BORDER, IID_PEDESTRIAN, etc.) to determine the type of an object or store the imageID inside any of your objects as a data member. Doing so will result in a score of ZERO for this project.**

2. **Avoid using dynamic cast to identify common types of objects. Instead add methods to check for various <u>classes of behaviors</u>:**

   Don't do this:

```
void decideWhetherToAddOil(Actor* p)
{
    if (dynamic_cast<BadRobot*>(p) != nullptr ||
         dynamic_cast<GoodRobot*>(p) != nullptr ||
         dynamic_cast<ReallyBadRobot*>(p) != nullptr ||
         dynamic_cast<StinkyRobot*>(p) != nullptr)
         p->addOil();
}
```

   Do this instead:

```
void decideWhetherToAddOil(Actor* p)
{
      // define a common method, have all Robots return true, all
      // biological organisms return false
    if (p->requiresOilToOperate())
         p->addOil();
}
```

3. **Always avoid defining specific isParticularClass() methods for each type of object. Instead add methods to check for various common behaviors that span multiple classes:**

   Don't do this:

```
void decideWhetherToAddOil(Actor* p)
{
       if (p->isGoodRobot() || p->isBadRobot() || p->isStinkyRobot())
              p->addOil();
}
```

   Do this instead:

```
void decideWhetherToAddOil(Actor* p)
{
```

```
    // define a common method, have all Robots return true, all
    // biological organisms return false
    if (p->requiresOilToOperate())
      p->addOil();
}
```

4. **If two related subclasses (e.g., SmellyRobot and GoofyRobot) each directly define a data member that serves the same purpose in both classes (e.g., m_amountOfOil), then move that data member to the common base class and add accessor and mutator methods for it to the base class. So the Robot base class should have the m_amountOfOil data member defined once, with getOil() and addOil()functions, rather than defining this variable directly in both SmellyRobot and GoofyRobot.**

Don't do this:

```
class SmellyRobot: public Robot
{
      …
  private:
    int m_oilLeft;
};

class GoofyRobot: public Robot
{
      …
  private:
    int m_oilLeft;
};
```

Do this instead:

```
class Robot
{
  public:
    void addOil(int oil) { m_oilLeft += oil; }
    int getOil() const { return m_oilLeft; }
  private:
    int m_oilLeft;
};
```

5. **Never make any class's data members public or protected. You may make class constants public, protected, or private.**

6. **If a method is used only by other methods in same class as that method, make it private. If a method is used only by other methods in same class as that method or in classes derived from that class, make it protected.**

7. **Your StudentWorld public methods should never return a collection of the objects StudentWorld maintains or a pointer to such a collection. Only StudentWorld should know about all of its game objects and where they are. If an action requires traversing StudentWorld's collection, then a StudentWorld method should do it.**

Don’t do this:

```
class StudentWorld
{
  public:
    vector<Actor*> getActorsThatCanBeZapped(int x, int y)
    {
       …      // create a vector with a actor pointers and return it
    }
};

class NastyRobot
{
  public:
    virtual void doSomething()
    {
              …
      vector<Actor*> v;
      vector<Actor*>::iterator p;

      v = studentWorldPtr->getActorsThatCanBeZapped(getX(), getY());
      for (p = actors.begin(); p != actors.end(); p++)
        p->zap();

       }
};
```

Do this instead:

```
class StudentWorld
{
  public:
    void zapAllZappableActors(int x, int y)
    {
      for (p = actors.begin(); p != actors.end(); p++)
        if (p->isAt(x,y) && p->isZappable())
          p->zap();
    }
};

class NastyRobot
{
  public:
    virtual void doSomething()
    {
              …
      studentWorldPtr->zapAllZappableActors(getX(), getY());
    }
};
```

Notice that it is fine, however, for a StudentWorld method to return a pointer to one actor:

```
class StudentWorld
{
  public:
    Actor* findClosestZappableActor(int x, int y);
       …
```

It's possible to design a solution in which StudentWorld's interfaces make no mention of the names of any particular type of actor except Ghost Rider.

8. If two subclasses have a method that shares some common functionality, but also has some differing functionality, use an auxiliary method to factor out the differences:

Don’t do this:

```
class StinkyRobot: public Robot
{
      …
  public:
    virtual void doDifferentiatedStuff()
    {
      doCommonThingA();
      passStinkyGas();
      pickNose();
      doCommonThingB();
    }
};

class ShinyRobot: public Robot
{
      …
  public:
    virtual void doDifferentiatedStuff()
    {
      doCommonThingA();
      polishMyChrome();
      wipeMyDisplayPanel();
      doCommonThingB();
    }
};
```

Do this instead:

```
class Robot
{
  public:
    virtual void doSomething()
    {
        // first do the common thing that all robots do
      doCommonThingA();

        // then call a virtual function to do the differentiated stuff
      doDifferentiatedStuff();

        // then do the common final thing that all robots do
      doCommonThingB();
    }

  private:
    virtual void doDifferentiatedStuff() = 0;
};

class StinkyRobot: public Robot
{
      …
```

```
  private:
      // define StinkyRobot's version of the differentiated function
    virtual void doDifferentiatedStuff()
    {
        // only Stinky robots do these things
      passStinkyGas();
      pickNose();
    }
};

class ShinyRobot: public Robot
{
      …
  private:
      // define ShinyRobot's version of the differentiated function
    virtual void doDifferentiatedStuff()
    {
        // only Shiny robots do these things
      polishMyChrome();
      wipeMyDisplayPanel();
    }
};
```

Yes, it is legal for a derived class to override a virtual function that was declared private in the base class. (It's not trying to *use* the private member function; it's just defining a new function.)

Don't know how or where to start? Read this!

When working on your first large object-oriented program, you're likely to feel overwhelmed and have no idea where to start; in fact, it's likely that many students won't be able to finish their entire program. Therefore, it's important to attack your program piece by piece rather than trying to program everything at once.

Students who try to program everything at once rather than program incrementally almost always fail to solve CS32's project 3, so don't do it!

Instead, try to get one thing working at a time. here are some hints:

1. When you define a new class, try to figure out what public member functions it should have. Then write dummy "stub" code for each of the functions that you'll fix later:

```
class Foo
{
  public:
      int chooseACourseOfAction() { return 0; }      // dummy version
};
```

Try to get your project compiling with these dummy functions first, then you can worry about filling in the real code later.

2. Once you've got your program compiling with dummy functions, then start by replacing one dummy function at a time. Update the function, rebuild your program, test your new function, and once you've got it working, proceed to the next function.
3. **Make backups of your working code frequently. Any time you get a new feature working, make a backup of all your .cpp and .h files just in case you screw something up later.**

BACK UP YOUR .CPP AND .H FILES TO A REMOVABLE DEVICE OR TO ONLINE STORAGE EVERY TIME YOU MAKE A MEANINGFUL CHANGE!

WE WILL NOT ACCEPT EXCUSES THAT YOUR HARD DRIVE/COMPUTER CRASHED OR THAT YOUR CODE USED TO WORK UNTIL YOU MADE THAT ONE CHANGE (AND DON'T KNOW WHAT CAUSED IT TO BREAK).

If you use this approach, you'll always have something working that you can test and improve upon. If you write everything at once, you'll end up with hundreds of errors and just get frustrated! So don't do it.

In particular, for Part #1 (see below), start with this task: Re-read the spec, but when it mentions game objects, pay attention only to things related to Ghost Racer and border lines. Implement a Ghost Racer class whose *doSomething()* method does nothing. Have *StudentWorld*'s *init()* create a new Ghost Racer, have *move()* tell it to do something, and have *cleanUp()* delete it. This can be done in about five lines of *StudentWorld* code and five lines of *Actor* code. Once you do this, you'll be surprised at what a psychological boost it is to see the graphics on the screen. (You can type *q* to quit, but will have a memory leak if the *StudentWorld* destructor doesn't call *cleanUp()*.)

Building the Game

The game assets (i.e., image and sound files) are in a folder named *Assets*. The way we've written the main routine, your program will look for this folder in a standard place (described below for Windows and macOS). A few students may find that their environment is set up in a way that prevents the program from finding the folder. If that happens to you, change the string literal `"Assets"` in *main.cpp* to the full path name of wherever you choose to put the folder (e.g., `"Z:/CS32Project3/Assets"` or `"/Users/fred/CS32Project3/Assets"`).

To build the game, follow these steps:

For Windows

Unzip GhostRacer-skeleton-windows.zip archive into a folder on your hard drive. Double-click on GhostRacer.sln to start Visual Studio.

If you build and run your program from within Visual Studio, the Assets folder should be in the same folder as your *.cpp* and *.h* files. On the other hand, if you launch the program by double-clicking on the executable file, the Assets folder should be in the same folder as the executable.

For macOS

Unzip GhostRacer-skeleton-mac.zip archive into a folder on your hard drive.
Double-click on our provided GhostRacer.xcodeproj to start Xcode.

If you build and run your program from within Xcode, the Assets directory should be in the directory yourProjectDir*/DerivedData/*yourProjectName*/Build/Products/Debug* (e.g., */Users/fred/GhostRacer/DerivedData/GhostRacer/Build/Products/Debug*). On the other hand, if you launch the program by double-clicking on the executable file, the Assets directory should be in your home directory (e.g., */Users/fred*).

What to Turn In

Part #1 (20%)

Ok, so we know you're scared to death about this project and don't know where to start. So, we're going to incentivize you to work incrementally rather than try to do everything all at once. For the first part of Project 3, your job is to build a really simple version of the Ghost Racer game that implements maybe 15% of the overall project. You must program:

1. A class that can serve as the base class for all of your game's actors (e.g., a base class that accommodates Ghost Racer, all types of actors, goodies, projectiles, etc.):
    i. It must have a constructor that initializes the object appropriately.
    ii. It must be derived from our *GraphObject* class.
    iii. It must have a member function named *doSomething()* that can be called to cause the actor to do something.
    iv. You may add other public/private member functions and private data members to this base class, as you see fit.
2. A BorderLine class, derived in some way from the base class described in 1 above:
    i. It must implement the specifications described in the Border Line section above.
    ii. You may add any public/private member functions and private data members to your BorderLine class as you see fit, so long as you use good object-oriented programming style (e.g., you **must NOT** duplicate functionality across classes).

3. A limited version of your Ghost Racer class, derived in some way from the base class described in 1 above (either directly derived from the base class, or derived from some other class that is somehow derived from the base class):
    i. It must have a constructor that initializes Ghost Racer – see Ghost Racer section for more details on where to initialize Ghost Racer.
    ii. It must have a limited version of a *doSomething()* method that lets the user pick a direction by hitting a directional key. If the player hits a directional key during the current tick, your code must update Ghost Racer's angle and location appropriately (see the spec above). All this *doSomething()* method has to do is properly adjust Ghost Racer's direction and (x, y) coordinates using the *GraphObject* class's *setDirection()* and *moveTo()* method, and our graphics system will automatically animate its movement down the roadway!
    iii. You may add other public/private member functions and private data members to your Ghost Racer class as you see fit, so long as you use good object-oriented programming style (e.g., you must not duplicate functionality across classes). But you need not implement spraying holy water, skidding on an oil slick, etc., for this part of the project.
4. A limited version of the *StudentWorld* class.
    i. Add any private data members to this class required to keep track of all game objects (right now all of those game objects will just be BorderLines, but eventually it'll also include actors, oil slicks, holy water spray, goodies, souls, etc.) as well as your Ghost Racer object. You may ignore all other items in the game such as human and zombie pedestrians, holy water spray, goodies, etc. for Part #1.
    ii. Implement a constructor for this class that initializes your data members.
    iii. Implement a destructor for this class that frees any remaining dynamically allocated data, if any, that has not yet been freed at the time the *StudentWorld* object is about to destroyed.
    iv. Implement the *init()* method in this class. It must create Ghost Racer and insert it into the roadway at the proper starting location. It must also create all of the border lines and add them to the roadway as specified in this spec. Your *init()* method may ignore any other objects like lost souls, oil slicks, etc., but it must deal with Ghost Racer and border lines.
    v. Implement the *move()* method in your *StudentWorld* class. During each tick, it must ask Ghost Racer and other actors (just border lines for now) to do something. Your *move()* method need not check to see if Ghost Racer has died or not; you may assume for Part #1 that Ghost Racer cannot die. Your *move()* method does not have to deal with any actors other than Ghost Racer and the border lines.
    vi. Implement a *cleanUp()* method that frees any dynamically allocated data that was allocated during calls to the *init()* method or the *move()* method (i.e., it should delete all your allocated border lines and Ghost Racer). Note: Your *StudentWorld* class must have both a destructor

and the *cleanUp()* method even though they likely do the same thing (in which case the destructor could just call *cleanUp()*).

As you implement these classes, repeatedly build your program – you'll probably start out with lots of errors… Relax and try to remove them and get your program to run. (Historical note: A UCLA student taking CS 131 once got 1,800 compilation errors when compiling a 900-line class project written in the Ada programming language. His name was Carey Nachenberg. Somehow he survived and has lived a happy life since then.)

You'll know you're done with Part #1 when your program builds and does the following: When it runs and the user hits Enter to begin playing, it displays a roadway with Ghost Racer in its proper starting position, yellow border lines along the sides, and white border lines separating the three lanes of the roadway. If your classes work properly, you should be able to move Ghost Racer left and right, and accelerate/decelerate using the directional keys. The border lines should move faster or slower down the screen depending on Ghost Racer's speed.

Your Part #1 solution may actually do more than what is specified above; for example, if you are making good progress, try to add lost soul objects to your program. Just make sure that what you have builds and has at least as much functionality as what's described above, and you may turn that in instead.

Note, the Part #1 specification above doesn't require you to implement any human or zombie pedestrians, zombie cabs, goodies, holy water spray, oil slicks, etc. (unless you want to). You may do these unmentioned items if you like but they're not required for Part #1. **However, if you add additional functionality, make sure that your Ghost Racer, border lines, and *StudentWorld* classes still work properly and that your program still builds and meets the requirements stated above for Part #1!**

If you can get this simple version working, you'll have done a bunch of the hard design work. You'll probably still have to change your classes a lot to implement the full project, but you'll have done most of the hard thinking.

What to Turn In For Part #1

You must turn in your source code for the simple version of your game, which **must build without errors** under either Visual Studio or Xcode. We may also devise a simple test framework that runs under g32; if we do, your code **must build without errors** in that framework. If it does not also run without errors, that indicates some fundamental problem that will probably cost you a lot of points. You will turn in a zip file containing nothing more than these four files:

```
Actor.h          // contains base, Ghost Racer, and BorderLine class declarations
                 //    as well as constants required by these classes
Actor.cpp        // contains the implementation of these classes
StudentWorld.h   // contains your StudentWorld class declaration
```

StudentWorld.cpp // contains your *StudentWorld* class implementation

You will not be turning in any other files – we'll test your code with our versions of the other .cpp and .h files. **Therefore, your solution must NOT modify any of our files or you will receive zero credit!** (Exception: You may modify the string literal `"Assets"` in *main.cpp*.) You will not turn in a report for Part #1; we will not be evaluating Part #1 for program comments, documentation, or test cases; all that matters for Part #1 is correct behavior for the specified subset of the requirements.

Part #2 (80%)

After you have turned in your work for Part #1 of Project 3, we will discuss one possible design for this assignment. For the rest of this project, you are welcome to continue to improve the design that you came up with for Part #1, **or you can use the design we provide**.

In Part #2, your goal is to implement a fully working version of Ghost Racer game, which adheres exactly to the functional specification provided in this document.

What to Turn In For Part #2

You must turn in your source code for your game, which **must build without errors** under either Visual Studio or Xcode. We may also devise a simple test framework that runs under g32; if we do, your code **must build without errors** in that framework. If it does not also run without errors, that indicates some fundamental problem that will probably cost you a lot of points. You will turn in a zip file containing nothing more than these five files:

```
Actor.h             // contains declarations of your actor classes
                    //   as well as constants required by these classes
Actor.cpp           // contains the implementation of these classes
StudentWorld.h      // contains your StudentWorld class declaration
StudentWorld.cpp    // contains your StudentWorld class implementation

report.docx, report.doc, or report.txt  // your report (10% of your grade)
```

You will not be turning in any other files – we'll test your code with our versions of the other .cpp and .h files. **Therefore, your solution must NOT modify any of our files or you will receive zero credit!** (Exception: You may modify the string literal `"Assets"` in *main.cpp*.)

You must turn in a report that contains the following:

1. A high-level description of each of your public member functions in each of your classes, and why you chose to define each member function in its host class; also explain why (or why not) you decided to make each function virtual or pure virtual. For example, "I chose to define a pure virtual version of the sneeze() function in my base Actor class because all actors in the Pedestrian class are able to sneeze, and each type of actor sneezes in a different way." There's no need to write pseudocode except for the comparatively few more complex functions.
2. A list of all functionality that you failed to finish as well as known bugs in your classes, e.g. "I didn't implement the holy water goodie class." or "My zombie pedestrian doesn't work correctly yet so I treat it like a human pedestrian right now."
3. A list of other design decisions and assumptions you made; e.g., "It was not specified what to do in situation X, so this is what I decided to do."

FAQ

Q: The specification is silent about what to do in a certain situation. What should I do?
A: Play with our sample program and do what it does. Use our program as a reference. If neither the specification nor our program makes it clear what to do, do whatever seems reasonable and document it in your report. **If the specification is unclear, but your program behaves like our demonstration program, YOU WILL NOT LOSE POINTS!**

Q: What should I do if I can't finish the project?!
A: Do as much as you can, and whatever you do, make sure your code builds! If we can sort of play your game, but it's not complete or perfect, that's better than it not even building!

Q: Where can I go for help?
A: Try TBP/HKN/UPE – they provide free tutoring and can help you with your project!

Q: Can I work with my classmates on this?
A: You can discuss general ideas about the project, but don't share source code with your classmates. Also don't help them write their source code.

GOOD LUCK!