Project 3
Zombie Dash

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

Time due:

Part 1: 11 PM, Thursday, February 21
Part 2: 11 PM, Thursday, February 28

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.

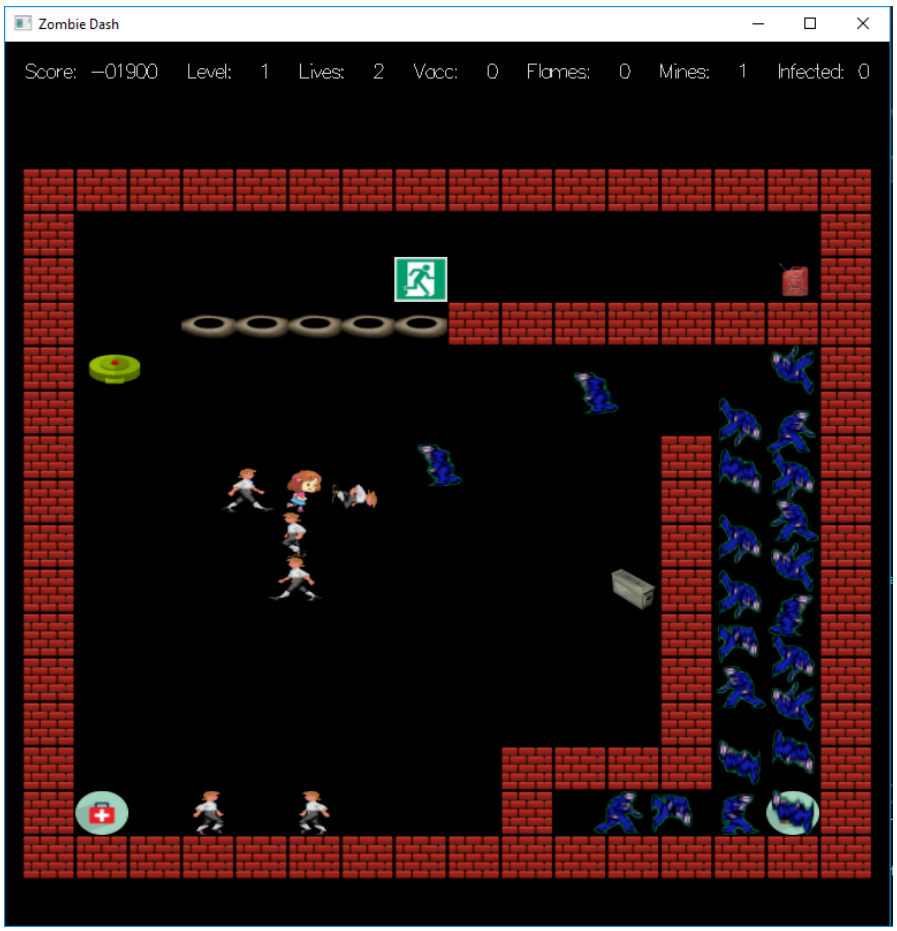
Table of Contents

Introduction

NachenGames corporate spies have learned that SmallSoft is planning to release a new game called Zombie Dash, 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 Zombie Dash 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.

In Zombie Dash, you play the role of Penelope Dolittle, an amateur zombie hunter and professional StarCraft player. Your job is to trek through an abandoned building, rescuing frightened citizens before they are turned into brainless, drooling zombies. On each level, you must lead all of the citizens safely to the level's exit(s), then head through an exit yourself so you can advance to the next level of the building to save more citizens. If you kill some zombies in the process, that's even better. Once you have saved all of the citizens from every level of the building, the game is over.

Here's a screenshot of the game:



In the center left of the screen, you can see Penelope with her shocking red hair, surrounded by four frightened citizens who need to be rescued. You can also see an army of zombies dressed in blue rags. While all of the zombies look the same, some are pretty dumb and just wander around randomly, while others are smart and will attempt to move toward Penelope or citizens if they get too close. Even worse, any time either type of zombie gets next to a person and faces them, the zombie may attempt to vomit on the person. Should the vomit hit the victim, the person will soon become a brainless, vomiting zombie, unless before it's too late, they can get to the exit or use a vaccine. However, only Penelope is clear-headed enough to use a vaccine on herself; a citizens' thinking is too muddled by fear.

On the lower left and right corners of the screen, you see vaccination kits. If Penelope picks one up, she gets a single-use vaccine that she can use to cure herself (by the player's pressing the enter key) should she have been previously infected by zombie vomit. Unfortunately these vaccines aren't protective against future infection; they can only cure a previously-acquired infection should Penelope get vomited on.

In the center right of the screen, you'll notice a metallic box. Penelope can pick up the box in order to obtain two landmines. Once Penelope has picked up the box, she can hit the tab key to deploy a landmine. You can see an active landmine in green in the upper-left corner of the screen. Be careful: Once Penelope deploys a landmine it becomes active almost immediately, and if she or anyone else steps on it, it will explode into flames taking out zombies and people nearby. When a landmine explodes, it will also create a pit in the floor. You can see a row of these pits in the top middle of the screen. A pit is deadly to people and zombies alike; if they step into one, they'll fall through the floor and die.

In the upper right corner, you'll notice a gas can which holds fuel. Fuel you say? When Penelope picks up the gas can, she'll instantly get enough fuel to shoot five searing flames from her not-Boring flamethrower ☺. Just hit the space key to fire. Be careful: although the flames are deadly to zombies, they are also lethal to the citizens you're trying to save, and accidentally taking out a citizen will cost you dearly (you'll lose lots of points!). Flames will also destroy goodies (like vaccines, gas cans, boxes containing landmines, etc.) and cause deployed landmines to explode violently, killing nearby people and zombies alike.

Some dumb zombies hold vaccines (that they presumably somehow acquired in their past). When these zombies die (e.g., from being consumed by a flame, being blown up by a landmine, falling into a pit, etc.), they will sometimes drop a vaccine goodie onto the square where they were standing. Penelope can pick up these vaccines to heal herself from the effects of zombie vomit.

The citizens that Penelope needs to rescue are petrified by fear and generally remain still unless either Penelope or a zombie comes nearby. A citizen will follow Penelope automatically if close enough to her (hoping to be led to an exit), and will attempt to run

away from a zombies that gets too close. Because of their panicked state, citizens aren't attentive enough to watch out for pits in the floor or landmines. So be careful: If Penelope leads a citizen near a landmine or a pit, the citizen might trigger the mine or fall through the pit. Both will cost you many points and of course, the loss of a simulated human life ☹.

In the upper middle of the screen, you see the exit (there may be more than one exit on a level). To complete a level, Penelope must lead all of the living citizens to the exit to save them, then use the exit herself.

Penelope is not superhuman, and she can die if she comes into contact with a flame (hanging in the air after being fired from her flamethrower or a landmine), gets too close to a pit and falls through, or gets infected by a zombie's vomit and turns into a zombie herself. The citizens can also die (or be converted to zombies) in the same ways.

Points are awarded or taken away as follows:

- When Penelope picks up a goodie (a vaccine, gas can, or landmine box): 50 points
- When a citizen makes it safely to the exit: 500 points
- When a dumb zombie is destroyed: 1000 points
- When a a smart zombie is destroyed: 2000 points
- When a citizen becomes a zombie or dies for any reason: −1000 points

You control Penelope with the arrow keys, or for lefties and others for whom the arrow key placement is awkward, WASD or the numeric keypad: up is *w* or *8*, left is *a* or *4*, down is *s* or *2*, right is *d* or *6*. Use the space key to fire Penelope's flamethrower (if she's picked up fuel), the tab key to deploy a landmine (if she's picked up landmines), and the enter key to use a vaccine (if she's picked up a vaccine kit). To quit the game at any time, press the 'q' key.

Game Details

In Zombie Dash, Penelope starts out a new game with three lives and continues to play until all of her lives have been exhausted or until there are no more levels. There are multiple levels in Zombie Dash, beginning with level 1 (NOT zero), and during each level Penelope (controlled by the player) must deliver all living citizens (if any) to an exit, and then use an exit herself, in order to advance to the next level. Since it is possible for citizens to die or be converted to zombies before Penelope has a chance to save them, it is not required to deliver to the exit all the citizens that existed at the start of the level, just those that remain alive. If there are no living citizens on a level, Penelope may complete the current level by going directly to an exit. Of course, the more citizens that she saves, the more points the player gets.

The Zombie Dash 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., Penelope, a zombie, a landmine, etc.) will have an x coordinate in the half-open range [0,VIEW_WIDTH), and a y coordinate in the half-open range [0, VIEW_HEIGHT).

Each level has a fixed layout that is specified in a data file found in your Assets directory (e.g., level01.txt, level02.txt, etc.). Each level data file contains a specification for the layout of the current level, the initial locations of all the smart and dumb zombies and Penelope, as well as the initial locations of all walls, pits, goodies, and exits. For more information on these level data files, please see the Level Data File section below. You may define your own level data files in order to customize your game (*and more importantly, to test your game*).

At the beginning of each level, and when the player restarts a level because Penelope died or was turned into a zombie, the level must be reset to its initial state (as shown in the data file). That is, all old objects should be discarded, and all zombies, people, goodies, pits, etc., should start in their initial position and state for that level (i.e., be freshly constructed).

At the beginning of each level (or when the player restarts a level because Penelope died or turned into a zombie), Penelope starts out with zero flamethrower charges, zero vaccines, and zero landmines regardless of how many she had previously. Regardless of whether she was uninfected, infected by zombie vomit (but not yet turned into a zombie), or turned into a zombie, she is restored to her original healthy uninfected state every time she starts a new level or restarts the current level.

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 Penelope, zombies, citizens, landmines, flames, vomit, goodies, etc. - a chance to do something – e.g., fire, move, die, vomit, etc.
- You must check to see if Penelope has died or changed into a zombie. If so, you must indicate this to our game framework (we'll tell you how later) so the level can end and, if Penelope has more lives, restart fresh.
- You must delete/remove all dead objects from the game – this includes zombies that have been destroyed by flames or by falling through a pit, citizens that have been killed or converted into zombies, flames or vomit that have dissipated, landmines that have exploded, goodies that have been picked up, etc.
- You must update the game statistics line at the top of the screen, including the number of remaining lives Penelope has, the player's current score, the current level number, the number of flamethrower charges, landmines and vaccines

Penelope current holds, as well as Penelope's current infection level (0 to 499 – at 500 she becomes a zombie).
- Check to see if the player has completed the current level, and if so, end the current level so Penelope may advance to the next level.

During each tick, the game may also need to introduce one or more new objects into the game – for instance, a new zombie (created when a citizen has changed into a zombie), a new flame (generated by a flamethrower or a landmine), or a new vaccine goodie (dropped by a dumb zombie that just died).

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

```
Score: 004500  Level: 27  Lives: 3  Vaccines: 2  Flames: 16  Mines: 1  Infected: 0
```

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

There are three major types of "goodies" in Zombie Dash that Penelope will want to pick up: vaccine goodies, gas can goodies, and landmine goodies. Their respective behaviors are described in the sections on goodies below.

There are two major types of projectiles in Zombie Dash: flames and vomit. Their respective behaviors are described in the sections on projectile behaviors below.

If Penelope dies or is converted into a zombie she loses one "life." If, after losing a life, Penelope has one or more remaining lives left, she is placed back on the current level and must again complete the entire level from scratch (with the level starting as it was at the beginning of the first time it was attempted). If Penelope dies and has no lives left, then the game is over.

The player may fire Penelope's flamethrower by pressing the space bar. If Penelope has a fuel charge (obtained by having earlier picked up a gas can goodie), firing the flamethrower will place three flames directly in front of Penelope in the direction she is facing. Flames fired by Penelope will instantly destroy both people and zombies touched by the flame (including Penelope herself if she walks into a flame that was just fired!). Flames will also destroy all goodies they touch. Finally if a flame hits a landmine, it will cause the mine to explode just as if it were stepped upon.

The player may deploy a landmine, if Penelope has any, by pressing the tab key. A landmine deployed by Penelope will produce a grid of flames all around it if it is stepped on by Penelope, a citizen, or a zombie while it's active (or activated by a flame from another landmine or Penelope's flamethrower). These flames, like those fired from Penelope's flamethrower, instantly kill all peoples and zombies that are close enough, including Penelope. When a landmine explodes, it will also introduce a pit in the ground at the location of the landmine.

A person (Penelope or a citizen) or a zombie who steps on a pit will fall in and die.

There are two types of zombies in Zombie Dash: dumb zombies and smart zombies. A dumb zombie wanders around randomly and, if it happens to face a person near it, will attempt to vomit on that person. A smart zombie also wanders around randomly until it gets soewhat near Penelope or a citizen. At this point, it will attempt to move toward the person, and vomit on that person when it's close enough. The exact behaviors are described in the sections on zombie behaviors below.

Just as Penelope can fire flames, zombies can shoot projectile vomit from their mouths when they are next to a person. A person will turn into a zombie exactly 500 ticks after being infected by zombie vomit. The only exception is if Penelope gets infected and subsequently uses a vaccine to cure herself.

When Penelope is turned into a zombie, the player's number of remaining lives is decremented by 1. If the player still has at least one life left, then the user is prompted to continue and given another chance by restarting the current level from scratch.

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 Penelope fires her flamethrower
pointerToGameWorld->playSound(SOUND_PLAYER_FIRE);
```

- You must play a SOUND_PLAYER_FIRE sound any time Penelope successfully fires her flamethrower.
- You must play a SOUND_ZOMBIE_VOMIT sound any time a zombie attempts to vomit on a person.
- You must play a SOUND_PLAYER_DIE sound any time Penelope dies by being damaged by a flame, falling into a pit, or turning into a zombie.
- You must play a SOUND_ZOMBIE_DIE sound any time a zombie dies by being damaged by a flame or falling into a pit.
- You must play a SOUND_CITIZEN_DIE sound any time a citizen dies by being damaged by a flame or falling into a pit.
- You must play a SOUND_LANDMINE_EXPLODE sound any time a landmine explodes.
- You must play a SOUND_GOT_GOODIE sound any time the player successfully picks up a goodie.
- You must play a SOUND_CITIZEN_INFECTED sound any time a citizen is successfully infected by a zombie's vomit. Note: This sound is played only when the citizen is first covered in vomit, NOT when they become a zombie.
- You must play a SOUND_ZOMBIE_BORN sound any time a citizen converts from an infected citizen into a full-fledged zombie
- You must play a SOUND_CITIZEN_SAVED sound any time a citizen uses an exit on the level to reach safety

- You must play a SOUND_LEVEL_FINISHED sound every time Penelope successfully completes a level.

Constants for each specific sound, e.g., SOUND_ZOMBIE_BORN, 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. For example, if a zombie shoots vomit out of its mouth, did this vomit come into contact with a nearby person? Or when Penelope walks near a goodie, did she get close enough to pick it up?

In Zombie Dash, two objects are said to overlap if the Euclidean distance between their (x,y) centers is less than or equal to 10 pixels (i.e., if $(\Delta x)^2 + (\Delta y)^2 \leq 10^2$). So, for example, in this spec we may say:

- Vomit will infect a person if it overlaps with the person.
- A flame will kill a person if it overlaps with the person.
- A person will fall into a pit if they overlap with the pit.
- Penelope will pick up a goodie if she overlaps with the goodie.
- A flame will activate a landmine if it overlaps with the landmine.
- A flame will destroy a goodie if it overlaps with the goodie.
- A person will use an exit if they overlap with the exit.
- A flame will be blocked from being fired by a wall or an exit.
- Vomit will be blocked by running into a wall or an exit.

This means that the center (x,y) positions of each of the two objects must be within 10 pixels of each other for the action to take place/be blocked.

Determining Blocking of Movement

In Zombie Dash, citizens, zombies, and Penelope can move anywhere on the level, with the following exceptions:

- They must not move onto any wall.
- They must not move onto any citizen, zombie, or Penelope.

When we say "P must not move onto Q" we mean the following: Moving P must not result in P's bounding box intersecting AT ALL with Q's bounding box. If an actor has its lower left corner at location (x,y), then the actor's bounding box is the rectangle with lower left corner (x,y) and upper right corner

(x+SPRITE_WIDTH−1, y+SPRITE_HEIGHT−1)

where SPRITE_WIDTH and SPRITE_HEIGHT are the width and height of all game objects (16x16).

There are additional game details that you must address in your implementation – these will be described in the sections below.

So how does a video game work?

Fundamentally, a video game is composed of a bunch of game objects; in Zombie Dash, those objects include Penelope, zombies (dumb and smart), goodies (e.g., vaccine goodies, gas can goodies, and landmine goodies), projectiles (e.g., flames, vomit), landmines, pits, and exits. 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 smart 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 Penelope, the algorithm that controls the Penelope object is the user's own brain and hand and the keyboard! In the case of other actors (e.g., a dumb zombie), 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 1 pixel to the left), or change other objects' states (e.g., when a smart zombie's algorithm is called by the game, it may determine that Penelope has moved in front of it, and it may eject vomit at her). Typically the behavior exhibited by an object during a single tick is limited in order to ensure that the gameplay appears smooth and that things don't move too quickly and confuse the player. For example, a dumb zombie will move just a pixel left/right/up/down, rather than moving ten or more pixels per tick; a dumb zombie moving, say, 20 pixels in a single tick would be annoying to the player, because we 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 dumb zombie changed its location from (10,5) to (9,5) (moved one pixel left), then our game framework would erase the graphic of the dumb zombie from location (10,5) on the screen and draw the dumb zombie's graphic at (9,5) 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 dumb zombie 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 in the maze.

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), 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., Penelope, citizens, zombies, pits, flames, vomit, exits, landmines, 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 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 (Penelope has lives left)
{
   Prompt the player to start playing  // "press a key to start"
   Initialize the game world           // you're going to write this

   while (Penelope 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
   }
     // Penelope died - you're going to write this code
   Cleanup all game world objects      // you're going to write this
```

```
    if (Penelope 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 (a vector or a 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) dumb zombie might decide to do each time it gets asked to do something:

```
class DumbZombie : public SomeOtherClass
{
   public:
        virtual void doSomething()
        {
            If the player is in front of me and close by, then
                   Vomit in the direction of the player
            Else if I still want to continue moving in the current direction
                   Move one pixel in my current direction
                   Decrement the number of remaining ticks to move in this direction
            Else if I want to choose a new direction
                   Pick a new direction to move, and pick how many ticks to move
                   in that direction.
        }
        ...
};
```

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

```
class Penelope : public …
{
   public:
        virtual void doSomething()
        {
             Try to get user input (if any is available)
               If the user pressed the UP key then
                   Increase my y location by one
               If the user pressed the DOWN key then
                   Decrease my y location by one
               ...
               If the user pressed the space bar to fire and I have
                         flamethrower charges left, then
                  Introduce three new flame objects into the game in front
                         of me
               ...
        }
        ...
};
```

What Do You Have to Do?

You must create a number of different classes to implement Zombie Dash game. Your classes must work properly with our provided classes, and **you must not modify our 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 (Penelope, citizens, zombies, projectiles, goodies, landmines, pits, flames, walls, exits, etc.) in the game.
2. You must create a class to represent Penelope in the game.
3. You must create classes for dumb zombies, smart zombies, vomit, flames, vaccine goodies, gas can goodies, landmine goodies, walls, pits, landmines, etc., as well as any additional base classes (e.g., a zombie base class if you find it convenient) 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 citizens, dumb zombies, smart zombies, Penelope, goodies, projectiles, walls, 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 dies, the user shoots a flame at a goodie and destroys it, a flame dissipates, etc.), and destroying ALL of the actors in the game world when the user loses a life.

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, not you, will call the *init()* method that you defined in your *StudentWorld* class. The *init()* method is responsible for constructing a representation of the current level in a StudentWorld object and populating it with initial objects (e.g., walls, zombies, goodies, exits, Penelope), 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 (which 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., Penelope, each citizen, zombie, goodie, projectile, landmine, exit, pit, etc.) to try to do something: move themselves and/or perform their specified behavior. Finally, this method is responsible for disposing of (i.e., deleting) actors that need to disappear during a given tick (e.g., vomit that falls to the ground and disappears, a dead dumb zombie, etc.). For example, if a dumb zombie is hit by a flame, then its state should be set to dead, and then after all of the live actors in the game get a chance to do something during the tick, the *move()* method should remove that dumb zombie 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 Penelope completes the current level or loses a life (e.g., she falls into a pit, gets singed by flames, or turns into a zombie). The *cleanup()* method is responsible for freeing all actors (e.g., all zombie objects, all wall objects, all projectiles, all goodie objects, the Penelope object, landmine objects, exit 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 flame that was added to the screen by an exploding landmine) that have not yet been removed from the game.

You may add as many other public/protected/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* or *protected* 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);
```

```
string assetPath() const;
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 Penelope has left.

decLives() reduces the number of lives Penelope has by one.

incLives() increases the number of lives Penelope has by one.

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

increaseScore() is used by a *StudentWorld* object (or your other classes) to increase or decrease the user's score upon successfully destroying a zombie or picking up a goodie of some sort. When your code calls this method, you must specify how many points the user gets (e.g., 1000 points for destroying a dumb zombie, −1000 points if a citizen dies or is converted to a zombie). 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: 1200  Level: 14  Lives: 2  Vacc:  4  Flames: 19  Mines:  3  Infected: 0
```

assetPath() returns the path to the directory that contains the game assets (images, sounds, and the level data files).

getKey() can be used to determine if the user has hit a key on the keyboard to move Penelope or to fire a projectile. This method returns true if the user hit a key during the current tick, and false otherwise (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
KEY_PRESS_TAB
KEY_PRESS_ENTER
```

The *playSound()* method can be used to play a sound effect when an important event happens during the game (e.g., a zombie dies or Penelope picks up a goodie). You can find constants (e.g., SOUND_PLAYER_FIRE) 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 dumb zombie dies, make a dying sound

if (the Zombie Has Died)
    studentWorldPtr->playSound(SOUND_ZOMBIE_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 Penelope object into the game world as specified in the current level's data file.
3. Allocate and insert various wall, pit, goodie, zombie, and exit objects into the game world as specified in the current level's data file.

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., zombies like smart zombies, pits, landmines, projectiles like vomit, goodies, etc.) in a **single** STL collection such as a *list* or *vector*. (To do so, we recommend using a container of pointers to the actors). If you like, a *StudentWorld* object may keep a separate pointer to the Penelope object rather than keeping a pointer to that object in the container with the other actor pointers; Penelope is the **only** actor pointer allowed to not be stored in the single actor container. The *init()* method may also initialize any other *StudentWorld* member variables it needs, such as the number of remaining citizens that need to be saved on this level before Penelope can use the exit.

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).

move() Details

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

1. It must ask all of the actors that are currently alive in the game world to do something (e.g., ask a dumb zombie to move itself, ask a goodie to check if it overlaps with Penelope, and if so, grant her something, give Penelope a chance to move up, down, left or right, etc.).
    a. If an actor does something that causes Penelope to die (e.g., a flame overlaps with Penelope, she falls into a pit, she turns into a zombie), then the *move()* method should immediately return `GWSTATUS_PLAYER_DIED`.

   b. Otherwise, if the all remaining citizens and Penelope have used the exit and it's time to advance to the next level, then the *move()* method must return a value of `GWSTATUS_FINISHED_LEVEL`.
2. If *move()* hasn't returned as a result of the above, it must then delete any actors that have died during this tick (e.g., a dumb zombie that was killed by a flame or a goodie that disappeared because it overlapped with Penelope and was therefore picked up) and remove them from the collection of actors.
3. 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 landmines Penelope has, the current level, 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 indicates that Penelope died during the current tick, and instructs our provided framework code to tell the user the bad news and restart the level if Penelope has more lives left (or end the game if she's out of lives). If your *move()* method returns this value and Penelope 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 level 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 indicates that the tick completed without Penelope dying BUT Penelope 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 indicates that Penelope has completed the current level (that is, she successfully escorted all of the citizens to an exit and used an exit herself). 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 (if one exists), 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 zombies, Penelope, goodies,
    // pits, flames, vomit, landmines, etc.

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

          if (Penelope died during this tick)
                return GWSTATUS_PLAYER_DIED;

          if (Penelope completed the current level)
                return GWSTATUS_FINISHED_LEVEL;
        }
    }

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

    // 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 Penelope, zombies, projectiles like flames and vomit (which are added to the game when Penelope or a zombie attacks or when a landmine goes off and the flames it produces must linger on the screen for a second or two, then disappear), landmines, goodies like vaccine goodies, pits, and walls.

Your *move()* method must iterate over every actor that's in the game (i.e., held by a *StudentWorld* object) and ask it to do something by calling a member function in the actor's object named *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 dumb zombie might move one pixel left, Penelope might shoot a flame or drop a landmine, a flame may singe a nearby actor, etc.

It is possible that one actor (e.g., a flame) may destroy another actor (e.g., a dumb zombie) 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., a zombie that died during the tick should NOT block a person from moving into a location that intersects with its former bounding box).

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 dumb zombie is killed by a flame and it dies, then it should be noted as dead (so that other actors don't interact with it during the current tick), and at the end of the tick, its *pointer* should be removed from the StudentWorld's container of active objects, and the dumb zombie 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 flame has been fired and has burned nearby actors for exactly two ticks, 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.)

cleanUp() Details

When your *cleanUp()* method is called by our game framework, it means that Penelope lost a life (e.g., she fell into a pit, turned into a zombie after being infected, or walked into a flame she fired herself) or has completed the current level. In this case, every actor in the entire game (Penelope and every zombie, goodie, projectile, landmine, wall, pit, 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 Penelope dies. Instead, this method will be called by our code when *init()* returns an appropriate status.

The *StudentWorld* destructor will be called by our game framework when the game is over. If the game ends prematurely because the player pressed the `q` key, *cleanUp()* will NOT have been called by our framework, so your destructor should call it to make sure the game shuts down cleanly. In normal gameplay, Penelope may lose her last life or finish the last level, resulting in *cleanUp()* being called as for any level ending; a little later, the *StudentWorld* destructor is called, which would call *cleanUp()* again. **Make sure that two consecutive calls to *cleanUp()* won't do anything undefined.** For example, if *cleanUp()* deletes an object and leaves a dangling pointer, it could be disastrous if the second call to *cleanUp()* tries use that pointer in a delete expression.

Level Data File

As mentioned, every level of Zombie Dash has a different layout. The layout for each level is stored in a text data file that you can edit with Windows Notepad or macOS's textedit. The file "level01.txt" holds the details for the first level's maze, "level02.txt" holds the details for the second level's maze, etc. If the next numbered level file does not exist, Penelope has completed all the levels and the player wins.

Here's an example level data file (you can modify our level data files to create wacky new levels, or add your own new level data files to add new levels, if you like):

level01.txt:

```
################
#L     @       #
#      X      G#
#  OOOOO#######
#              #
#             ss#
#  C         #ss#
#      C     #ss#
#            #ss#
#           L#ss#
#      C     #ss#
#            #ss#
#  C         #dd#
#         ####dd#
#V C    C#ddddV#
################
```

As you can see, the data file contains a 16x16 grid of different characters that represent the different actors/things in the level. Valid characters for your level data file are as follows; upper and lower case version of a letter are treated the same way:

The **#** character represents a wall. The perimeter of each maze MUST be surrounded completely by walls.

The **@** character specifies the starting location of Penelope, the player's avatar, when she starts a level. If Penelope must replay the current level because she dies or gets converted to a zombie, she must restart at this location.

The **C** character represents a citizen that Penelope must save – this specifies that a citizen must start at this location when the player starts (or replays) the current level.

The **O** (oh, not zero) character represents a pit that people and zombies can fall into.

The **V** character represents a vaccine goodie that Penelope can pick up and later use to cure herself if she's been vomited upon and is infected by zombie vomit.

The **G** character represents a gas can goodie that gives Penelope 5 flamethrower charges.

The **L** character represents a landmine goodie that grants Penelope two landmines.

The **X** character represents the level's exit. Penelope must lead all living citizens to exits, and then go into one herself to complete a level.

The **D** character represents a dumb zombie – this specifies that a dumb zombie must start at this location when the player starts (or replays) the current level.

The **S** character represents a smart zombie – this specifies that a smart zombie must start at this location when the player starts (or replays) the current level.

All **space** characters represent empty locations where Penelope, citizens, and zombies may walk within the level.

The x and y values in the level coordinate system lie in the half-open ranges [0,LEVEL_WIDTH) and [0,LEVEL_HEIGHT) respectively, where LEVEL_WIDTH is 16 and LEVEL_HEIGHT is 16. Since each object in the game is SPRITE_WIDTH pixels wide and SPRITE_HEIGHT pixels tall (i.e., 16x16), an object like the bottom-leftmost citizen (C) in the level shown above, at level x coordinate 3 and level y coordinate 1 in the level file, has (48, 16) as its (x,y) coordinates in the game.

The Level Class

We have graciously ☺ decided to provide you with a class that can load level data files for you. The class is named *Level* and may be found in our provided *Level.h* file. Here's how this class might be used:

```
#include "Level.h"    // required to use our provided class

void StudentWorld::someFunc()
{
        Level lev(assetPath());

        string levelFile = "level01.txt";
        Level::LoadResult result = lev.loadLevel(levelFile);
        if (result == Level::load_fail_file_not_found)
                cerr << "Cannot find level01.txt data file" << endl;
        else if (result == Level::load_fail_bad_format)
                cerr << "Your level was improperly formatted" << endl;
        else if (result == Level::load_success)
        {
                cerr << "Successfully loaded level" << endl;

                Level::MazeEntry ge = lev.getContentsOf(5,10); // level_x=5, level_y=10
                switch (ge)                                    // so x=80 and y=160
                {
                    case Level::empty:
                        cout << "Location 80,160 is empty" << endl;
                        break;
                    case Level::smart_zombie:
                        cout << "Location 80,160 starts with a smart zombie" << endl;
```

```
                    break;
                case Level::dumb_zombie:
                    cout << "Location 80,160 starts with a dumb zombie" << endl;
                    break;
                case Level::player:
                    cout << "Location 80,160 is where Penelope starts" << endl;
                    break;
                case Level::exit:
                    cout << "Location 80,160 is where an exit is" << endl;
                    break;
                case Level::wall:
                    cout << "Location 80,160 holds a Wall" << endl;
                    break;
                case Level::pit:
                    cout << "Location 80,160 has a pit in the ground" << endl;
                    break;
                // etc…
            }
        }
}
```

Hint: You will likely want to use our *Level* class to load the current level specification in your *StudentWorld* class's *init()* method. The *assetPath()* and *getLevel()* methods that your *StudentWorld* class inherits from *GameWorld* might also be useful, along with the *Stringstreams* writeup on the class web site!

You Have to Create the Classes for All Actors

Zombie Dash has a number of different game objects, including:

- Penelope
- Dumb zombies
- Smart zombies
- Citizens
- Landmines
- Pits
- Flames
- Vomit
- Vaccine Goodies
- Gas can Goodies
- Landmine Goodies
- Walls
- Exits

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

Now of course, many of your game objects will share things in common – for instance, every object in the game (dumb zombies, citizens, Penelope, pits, flames, exits, etc.) has x,y coordinates. Many game objects can perform an action (e.g., move or vomit) during each tick of the game. Many of them can be attacked (e.g., Penelope, citizens, and

zombies) and could "die" during a tick, including goodies and landmines if they're burned by a flame. Some objects like flames, vomit, pits, landmines, and goodies "activate" when they come into contact with an appropriate target (e.g., goodies activate when they overlap with Penelope (and give her some special power), flames activate when they overlap with Penelope, zombies and citizens, etc.

It is therefore your job to determine the commonalities between your different game objects 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 avoid duplicating code or data members** – if you find yourself writing the same (or largely similar) code or duplicating member variables 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 vaccine goodie section and the gas can goodie section, or in the dumb zombie and smart zombie sections) you must make sure to identify common behaviors and move these into proper base classes. DO NOT duplicate behaviors 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 SmartZombie: 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, int depth = 0);
double getX() const;                        // in pixels (0-255)
double getY() const;                        // in pixels (0-255)
virtual void moveTo(double x, double y); // in pixels (0-255)
int getDirection() const;                   // in degrees (0-359)
void setDirection(Direction d);             // {up, down, left, right}
```

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 since they are not defined as virtual in our base class.

GraphObject(int imageID,
int startX, // column first
int startY, // then row!
Direction startDirection,
int depth)

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 dumb zombie, citizen, Penelope, a pit, a flame, 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: **up**, **down**, **left** or **right**. Finally, you may specify the depth of the object. 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.

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

```
IID_PLAYER (for Penelope)
IID_ZOMBIE (for both smart and dumb zombies)
IID_CITIZEN
IID_FLAME
IID_VOMIT
IID_PIT
IID_LANDMINE (for a deployed landmine)
IID_VACCINE_GOODIE
IID_GAS_CAN_GOODIE
IID_LANDMINE_GOODIE
IID_EXIT
IID_WALL
```

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

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.

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 member variables, 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 dumb zombie'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
```

You ***must*** use the *moveTo()* method 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 *up*, *down*, *left* or *right*.

setDirection(Direction d) is used to change the direction a *GraphObject* is facing. For example, you could use this method and *getDirection()* to adjust the direction a person or zombie faces when it decides to move in a new direction.

Penelope

Here are the requirements you must meet when implementing Penelope class.

What a Penelope Object Must Do When It Is Created

When it is first created:

1. A Penelope object must have an image ID of IID_PLAYER.
2. The object must start at the location on the level as specified in the current level's data file. The object's starting location in the level must be equal to (SPRITE_WIDTH * level_x, SPRITE_HEIGHT * level_y); its starting level_x and level_y can be obtained using our provided *Level* class. *Hint: The StudentWorld object can pass in this (x,y) location when constructing the object.*
3. A Penelope object starts out alive.
4. A Penelope object has a direction of *right*.

5. A Penelope object has a depth of 0.
6. A Penelope object has no landmines, flamethrower charges, or vaccines.
7. A Penelope object has an infection status of false.
8. A Penelope object has an infection count of 0.

What a Penelope Object Must Do During a Tick

A Penelope object must be given an opportunity to do something during every tick (in her *doSomething()* method). When given an opportunity to do something, Penelope must do the following:

1. Penelope must check to see if she is currently alive. If not, then Penelope's *doSomething()* method must return immediately – none of the following steps should be performed.
2. The *doSomething()* method must check to see if Penelope is infected (because of previously being vomited on by a zombie). If so, she must increase her infection count by one. If Penelope's infection count reaches 500, she becomes a zombie and:
    a. She must immediately set her status to dead.
    b. The game must play a SOUND_PLAYER_DIE sound effect
    c. The *doSomething()* method must return immediately, doing nothing more during this tick.
    d. (The *StudentWorld* object should then detect that she's dead and the current level ends)
3. The *doSomething()* method 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 Penelope has at least one flamethrower charge, then Penelope will attempt to fire three flames into the three slots directly in front of her:
        i. Penelope's flamethrower charge count must decrease by 1.
        ii. Penelope must play the SOUND_PLAYER_FIRE sound effect (see the *StudentWorld* section of this document for details on how to play a sound).
        iii. Penelope will add up to three new flame objects to the game. If Penelope is at (px, py) this is where the new flame objects will go: For i = 1, 2, and 3,
            - If Penelope is facing up: pos_i = (px, py + i *SPRITE_HEIGHT)
            - If she is facing down: pos_i = (px, py − i *SPRITE_HEIGHT)
            - If she is facing left: pos_i = (px − i *SPRITE_WIDTH, py)
            - If she is facing right: pos_i = (px + i *SPRITE_WIDTH, py)

        You must use something like the following algorithm to add the flames to the game: For each of the i=1,2, and 3 slots in front of Penelope
        i. Compute the position pos_i where the next flame will go.

ii. If a flame at pos_i would overlap[1] with a **wall** or **exit** object, then immediately stop the loop.
iii. Otherwise add a new flame object[2] at pos_i with a starting direction that is the same as the direction Penelope is facing.

b. If the user pressed the tab key and if Penelope has any landmines in her inventory, Penelope will introduce a new landmine object at her current (x,y) location into the game and her landmine count will decrease by 1.
c. If the user pressed the enter key and if Penelope has any vaccines in her inventory, Penelope will set her infected status to false and reduce her vaccine count by 1. (She wasted that vaccine if she was not infected.)
d. If the user asks to move up, down, left or right by pressing a directional key:
    i. Set Penelope's direction to the specified movement direction.
    ii. Determine Penelope's destination location (dest_x, dest_y) which will be exactly 4 pixels in the direction Penelope is facing. So for example, if Penelope is at (x=16, y=16) and is facing down, her destination would be (dest_x=16, dest_y=12).
    iii. If the movement to (dest_x, dest_y) would not cause Penelope's bounding box to intersect with the bounding box[3] of any **wall**, **citizen** or **zombie** objects, then update Penelope's location to the specified location with the *GraphObject* class's *moveTo()* method.

What Penelope Must Do In Other Circumstances

- Penelope can be infected by vomit. When vomit overlaps with Penelope, her infection status becomes true.
- Penelope can be damaged. If a flame object overlaps with Penelope it will kill her. When killed:
    - She must immediately has her status set to dead,
    - The game must play a SOUND_PLAYER_DIE sound effect
    - (The *StudentWorld* object should detect her death and the current level ends)
- Penelope blocks other objects from moving nearby/onto her. Penelope's bounding box must never intersect with that of any citizen, zombie, or wall.

Getting Input From the User

Since Zombie Dash is a *real-time* game, you can't use the typical *getline* or *cin* approach to get a user's key press within Penelope'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

[1] See the discussion of overlap in the **Determining Object Overlap** section.
[2] Hint: When you create a new flame object at a particular location, give it to the *StudentWorld* object to manage (e.g., animate) along with the other game objects.
[3] See the discussion of bounding boxes in the **Determining Blocking of Movement** section.

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 Penelope::doSomething()
{
    ...
    int ch;
    if (getWorld()->getKey(ch))
    {
        // user hit a key during this tick!
        switch (ch)
        {
          case KEY_PRESS_LEFT:
            ... move Penelope to the left ...;
            break;
          case KEY_PRESS_RIGHT:
            ... move Penelope to the right ...;
            break;
          case KEY_PRESS_SPACE:
            ... add flames in front of Penelope...;
           break;

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

Wall

Walls don't really do much. They just sit there. Here are the requirements you must meet when implementing the wall class.

What a Wall Must Do When It Is Created

When it is first created:

1. A wall object must have an image ID of IID_WALL.
2. The object must start at the location on the level as specified in the current level's data file. The object's starting location in the level must be equal to (SPRITE_WIDTH * level_x, SPRITE_HEIGHT * level_y); its starting level_x and level_y can be obtained using our provided *Level* class. *Hint: The StudentWorld object can pass in this (x,y) location when constructing the object.*
3. A wall object has a direction of right.
4. A wall object has a depth of 0.

[4] Hint: Since your Penelope object will need to access the *getKey()* method in the *GameWorld* class (which is the base class for your *StudentWorld* class), your Penelope object (or more likely, one of its base classes) will need a way to obtain a pointer to the *StudentWorld* object it belongs to. If you look at our code example, you'll see how Penelope'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 Wall Must Do During a Tick

A wall 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 wall must do nothing. After all, it's just a wall!

What a Wall Must Do In Other Circumstances

- A wall cannot be damaged by a flame.
- A wall cannot be infected by vomit.
- A wall blocks the movement of citizens, zombies, and Penelope (its bounding box must never intersect with any actor's bounding box)[5]
- A wall blocks flames (i.e., a flame fired by Penelope or emitted by a landmine cannot overlap[6] or be created past a wall).

Exit

You must create a class to represent an exit that the citizens and Penelope can use to exit the current level. When a citizen overlaps with an exit, they will immediately leave the current level and the player will get points. When Penelope overlaps with an exit and all citizens have exited the level (or died), she will advance to the next level. Here are the requirements you must meet when implementing the exit class.

What an Exit Must Do When It Is Created

When it is first created:

1. An exit object must have an image ID of IID_EXIT.
2. The object must start at the location on the level as specified in the current level's data file. The object's starting location in the level must be equal to (SPRITE_WIDTH * level_x, SPRITE_HEIGHT * level_y); its starting level_x and level_y can be obtained using our provided *Level* class. *Hint: The StudentWorld object can pass in this (x,y) location when constructing the object.*
3. An exit has a direction of *right.*
4. An exit has a depth of 1.

What an Exit Must Do During a Tick

An exit 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 exit must do the following:

[5] See the **Determining Blocking of Movement** section.
[6] See the **Determining Object Overlap** section.

1. The exit must determine if it overlaps with a citizen (not Penelope!). If so, then the exit must:
   a. Inform the *StudentWorld* object that the user is to receive 500 points.
   b. Set the citizen object's state to dead (so that it will be removed from the game by the *StudentWorld* object at the end of the current tick) – **note, this must not kill the citizen in a way that deducts points from the player as if the citizen died due to a zombie infection, a flame, or a pit**.
   c. Play a sound effect to indicate that the citizen was saved by using the exit: SOUND_CITIZEN_SAVED.
2. The exit must determine if it overlaps[7] with Penelope. If so *and* all citizens have either exited the level or died, then:
   a. Inform the *StudentWorld* object that Penelope has finished the current level.

What an Exit Must Do In Other Circumstances

- An exit cannot be damaged by a flame.
- An exit cannot be infected by vomit.
- An exit do not block other objects from moving nearby/onto them.
- An exit DOES block flames (i.e., a flame fired by Penelope or emitted by a landmine cannot overlap or be created past an exit).

Pit

Pits don't really do much. They just sit there waiting for actors to fall into them. Here are the requirements you must meet when implementing the pit class.

What a Pit Must Do When It Is Created

When it is first created:

1. A pit object must have an image ID of IID_PIT.
2. The object must start at the location on the level as specified in the current level's data file. The object's starting location in the level must be equal to (SPRITE_WIDTH * level_x, SPRITE_HEIGHT * level_y); its starting level_x and level_y can be obtained using our provided *Level* class. *Hint: The StudentWorld object can pass in this (x,y) location when constructing the object.*
3. A pit object has a direction of right.
4. A pit object has a depth of 0.

What a Pit Must Do During a Tick

[7] See the **Determining Object Overlap** section.

A pit 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 pit will cause any person or zombie that overlaps with it to be destroyed (they fall into the pit). When the person/zombie is destroyed, it must behave just as it were damaged by a flame (e.g., if a dumb zombie falls into a pit, the player gets 1000 points, the game plays a dying noise, etc.; if Penelope falls into a pit the current level will end; a citizen falling into a pit dies, and the player loses 1000 points, etc.).

What a Pit Must Do In Other Circumstances

- A pit cannot be damaged by a flame.
- A pit cannot be infected by vomit.
- A pit does not block the movement of actors.
- A pit does not block vomit or flames.

Flame

You must create a class to represent a flame. Flame objects are produced from two different sources: by Penelope's flamethrower and by landmines. Here are the requirements you must meet when implementing the flame class.

What a Flame Must Do When It Is Created

When it is first created:

1. A flame object must have an image ID of IID_FLAME.
2. The starting location of a flame must be specified during construction.
3. The starting direction of a flame must be specified during construction.
4. A flame object has a depth of 0.
5. A flame object starts in an "alive" state.

What a Flame Must Do During a Tick

A flame 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 flame must do the following:

1. It must check to see if it is currently alive. If not, then *doSomething()* must return immediately – none of the following steps should be performed.
2. After exactly two ticks from its creation, the flame must set its state to dead so it can be destroyed and removed from the level by the *StudentWorld* object. The *doSomething()* method must return immediately, doing nothing more during this tick.

3. Otherwise, the flame will damage all damageable objects that overlap[8] with the flame. The following objects are all damageable and must react to being damaged in the appropriate way: Penelope, citizens, all types of goodies, landmines, and all types of zombies.

What a Flame Must Do In Other Circumstances

- A flame cannot be damaged by a flame.
- A flame cannot be infected by vomit.
- A flame does not block other objects from moving nearby/onto them.

Vomit

You must create a class to represent zombie vomit. Vomit objects are produced by both dumb and smart zombies when the zombie is next to a person (Penelope or a citizen) and the zombie faces in their direction. Here are the requirements you must meet when implementing the vomit class.

What Vomit Must Do When It Is Created

When it is first created:

1. An vomit object must have an image ID of IID_VOMIT.
2. The starting location of vomit must be specified during construction.
3. The starting direction of vomit must be specified during construction.
4. A vomit object has a depth of 0.
5. A vomit object starts in an "alive" state.

What Vomit Must Do During a Tick

Vomit 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 vomit must do the following:

1. It must check to see if it is currently alive. If not, then *doSomething()* must return immediately – none of the following steps should be performed.
2. After exactly two ticks from its creation, the vomit must set its state to dead so it can be destroyed and removed from the level by the *StudentWorld* object. The *doSomething()* method must return immediately, doing nothing more during this tick.

[8] See the **Determining Object Overlap** section.

3. Otherwise, the vomit will infect any infectable object that overlaps[9] with the vomit. The following objects are infectable and must react to being infected in the appropriate way: Penelope and citizens.

What Vomit Must Do In Other Circumstances

- Vomit cannot be damaged by a flame.
- Vomit cannot be infected by vomit.
- Vomit does not block other objects from moving nearby/onto it.

Vaccine Goodie

You must create a class to represent a vaccine goodie that Penelope can pick up. When Penelope overlaps with (picks up) this goodie, it adds one vaccine to Penelope's inventory. She may later use the vaccine by pressing the enter key to cure herself from a zombie infection (caused by a previous encounter with zombie vomit). Here are the requirements you must meet when implementing the vaccine goodie class.

What a Vaccine Goodie Must Do When It Is Created

When it is first created:

1. A vaccine goodie object must have an image ID of IID_VACCINE_GOODIE.
2. The object must start at the location on the level as specified in the current level's data file. The object's starting location in the level must be equal to (SPRITE_WIDTH * level_x, SPRITE_HEIGHT * level_y); its starting level_x and level_y can be obtained using our provided *Level* class. *Hint: The StudentWorld object can pass in this (x,y) location when constructing the object.*
3. A vaccine goodie has a direction of *right.*
4. A vaccine goodie has a depth of 1.
5. A vaccine goodie starts in an "alive" state.

What a Vaccine Goodie Must Do During a Tick

A vaccine 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 vaccine goodie must do the following:

1. It must check to see if it is currently alive. If not, then *doSomething()* must return immediately – none of the following steps should be performed.
2. The vaccine goodie must determine if it overlaps with Penelope. If so, then the vaccine goodie must:
    a. Inform the *StudentWorld* object that the user is to receive 50 points.

[9] See the **Determining Object Overlap** section.

b. Set its state to dead (so that it will be removed from the game by the *StudentWorld* object at the end of the current tick).
c. Play a sound effect to indicate that Penelope picked up the goodie: SOUND_GOT_GOODIE.
d. Inform the *StudentWorld* object that Penelope is to receive one dose of vaccine.

What a Vaccine Goodie Must Do In Other Circumstances

- A vaccine goodie can be damaged by a flame. When damaged, it must set its status to dead.
- A vaccine goodie cannot be infected by vomit.
- A vaccine goodie does not block other objects from moving nearby/onto it.

Gas Can Goodie

You must create a class to represent a gas can goodie that Penelope can pick up. When Penelope overlaps[10] with (picks up) this goodie, it adds 5 flamethrower charges to her inventory. She may later use the flamethrower charges by pressing the space key to fire her flamethrower. Here are the requirements you must meet when implementing the Gas Can Goodie class.

What a Gas Can Goodie Must Do When It Is Created

When it is first created:

1. A gas can goodie object must have an image ID of IID_GAS_CAN_GOODIE.
2. The object must start at the location on the level as specified in the current level's data file. The object's starting location in the level must be equal to (SPRITE_WIDTH * level_x, SPRITE_HEIGHT * level_y); its starting level_x and level_y can be obtained using our provided *Level* class. *Hint: The StudentWorld object can pass in this (x,y) location when constructing the object.*
3. A gas can goodie has a direction of *right.*
4. A gas can goodie has a depth of 1.
5. A gas can goodie starts in an "alive" state.

What a Gas Can Goodie Must Do During a Tick

A gas can 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 gas can goodie must do the following:

[10] See the **Determining Object Overlap** section.

1. It must check to see if it is currently alive. If not, then *doSomething()* must return immediately – none of the following steps should be performed.
2. The gas can goodie must determine if it overlaps with Penelope. If so, then the gas can goodie must:
   a. Inform the *StudentWorld* object that the user is to receive 50 points.
   b. Set its state to dead (so that it will be removed from the game by the *StudentWorld* object at the end of the current tick).
   c. Play a sound effect to indicate that Penelope picked up the goodie: SOUND_GOT_GOODIE.
   d. Inform the *StudentWorld* object that Penelope is to receive 5 charges for her flamethrower.

What a Gas Can Goodie Must Do In Other Circumstances

- A gas can goodie can be damaged by a flame. When damaged it must set its status to dead.
- A gas can goodie cannot be infected by vomit.
- A gas can goodie does not block other objects from moving nearby/onto it.

Landmine Goodie

You must create a class to represent a landmine goodie that Penelope can pick up. When Penelope overlaps[11] with (picks up) this goodie, it adds two landmines to Penelope's inventory. She may later use the landmines by pressing the tab key to deploy a landmine. Here are the requirements you must meet when implementing the landmine goodie class.

What a Landmine Goodie Must Do When It Is Created

When it is first created:

1. A landmine goodie object must have an image ID of IID_LANDMINE_GOODIE.
2. The object must start at the location on the level as specified in the current level's data file. The object's starting location in the level must be equal to (SPRITE_WIDTH * level_x, SPRITE_HEIGHT * level_y); its starting level_x and level_y can be obtained using our provided *Level* class. *Hint: The StudentWorld object can pass in this (x,y) location when constructing the object.*
3. A landmine goodie has a direction of *right.*
4. A landmine goodie has a depth of 1.
5. A landmine goodie starts in an "alive" state.

[11] See the **Determining Object Overlap** section.

What a Landmine Goodie Must Do During a Tick

A landmine 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 landmine goodie must do the following:

1. It must check to see if it is currently alive. If not, then *doSomething()* must return immediately – none of the following steps should be performed.
2. The landmine goodie must determine if it overlaps with Penelope. If so, then the landmine goodie must:
   a. Inform the *StudentWorld* object that the user is to receive 50 points.
   b. Set its state to dead (so that it will be removed from the game by the *StudentWorld* object at the end of the current tick).
   c. Play a sound effect to indicate that Penelope picked up the goodie: SOUND_GOT_GOODIE.
   d. Inform Inform the *StudentWorld* object that Penelope is to receive 2 landmines.

What a Landmine Goodie Must Do In Other Circumstances

- A landmine goodie can be damaged by flames. When damaged it must set its status to dead.
- A landmine goodie cannot be infected by vomit.
- A landmine goodie does not block other objects from moving nearby/onto them.

Landmine

You must create a class to represent a landmine. When a person (including Penelope) or zombie overlaps[12] with a landmine it will trigger the landmine and cause it to introduce flames all around the landmine (the flames will then damage anything they touch; the landmine itself does **not** damage anything directly). When another flame overlaps with a landmine it will trigger the landmine and cause it to introduce flames all around the landmine. Once triggered, a landmine is replaced by a pit. Here are the requirements you must meet when implementing the landmine class.

What a Landmine Must Do When It Is Created

When it is first created:

1. A landmine object must have an image ID of IID_LANDMINE.
2. The location of a landmine must be specified when it is constructed.
3. A landmine has a direction of *right.*
4. A landmine has a depth of 1.

[12] See the **Determining Object Overlap** section.

5. A landmine starts with 30 safety ticks (before it becomes active).
6. A landmine starts in an inactive state.
7. A landmine starts in an "alive" state.

What a Landmine Must Do During a Tick

A landmine 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 landmine must do the following:

1. It must check to see if it is currently alive. If not, then *doSomething()* must return immediately – none of the following steps should be performed.
2. If the landmine is not yet active then:
    a. It must decrement the number of safety ticks left.
    b. If the number of safety ticks is zero, the landmine becomes active.
    c. The *doSomething()* method must return immediately, doing nothing more during this tick.
3. The landmine must determine if it overlaps[13] with Penelope, a citizen, or a zombie. If so, then the landmine must:
    a. Set its state to dead (so that it will be removed from the game by the *StudentWorld* object at the end of the current tick).
    b. Play a sound effect to indicate that the landmine exploded: SOUND_LANDMINE_EXPLODE.
    c. Introduce a flame object at the same (x,y) location as the landmine .
    d. Introduce flame objects in the eight adjacent slots around the landmine (north, northeast, east, southeast, south, southwest, west, northwest). Each such adjacent spot must exactly SPRITE_WIDTH pixels away horizontally and/or SPRITE_HEIGHT pixels away vertically. (SPRITE_WIDTH and SPRITE_HEIGHT are both 16.) So if the landmine goodie were at position (100, 100), the northwest flame would be added at (84, 116), the east goodie at (116, 100), the southeast goodie at (116, 84), etc.
    e. Introduce a pit object at the same (x,y) location as the landmine.

What a Landmine Must Do In Other Circumstances

- A landmines can be damaged by flames. If a flame overlaps with a landmine (whether or not the landmine is active), then it must trigger the landmine just as if a person stepped on it. When damaged, the landmine must:
    - Set its state to dead (so that it will be removed from the game by the *StudentWorld* object at the end of the current tick).
    - Play a sound effect to indicate that the landmine exploded: SOUND_LANDMINE_EXPLODE.
    - Introduce a flame object in the same (x,y) location as the landmine .
    - Introduce flame objects in the eight adjacent slots around the landmine (north, northeast, east, southeast, south, southwest, west, northwest). Each

[13] See the **Determining Object Overlap** section.

such adjacent spot must exactly SPRITE_WIDTH pixels away horizontally and/or SPRITE_HEIGHT pixels away vertically. (SPRITE_WIDTH and SPRITE_HEIGHT are both 16.) So if the landmine goodie were at position (100, 100), the northwest flame would be added at (84, 116), the east goodie at (116, 100), the southeast goodie at (116, 84), etc.
    - Introduce a pit object at the same (x,y) location as the landmine.
- A landmine cannot be infected by vomit.
- A landmine does not block other objects from moving nearby/onto them.
- A landmine does not block flames.

Dumb Zombie

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

What a Dumb Zombie Must Do When It Is Created

When it is first created:

1. A dumb zombie object must have an image ID of IID_ZOMBIE.
2. If the object was created because an infected person became a zombie, it must start at the location passed to its constructor. Otherwise, the object must start at the location on the level as specified in the current level's data file. The object's starting location in the level must be equal to (SPRITE_WIDTH * level_x, SPRITE_HEIGHT * level_y); its starting level_x and level_y can be obtained using our provided *Level* class. *Hint: The StudentWorld object can pass in this (x,y) location when constructing the object.*
3. A dumb zombie has a direction of *right*.
4. A dumb zombie has a depth of 0.
5. A dumb zombie starts with a movement plan distance of 0.
6. A dumb zombie starts out in an "alive" state.

What a Dumb Zombie Must Do During a Tick

A dumb zombie 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 dumb zombie must do the following:

1. The dumb zombie must check to see if it is currently alive. If not, then its *doSomething()* method must return immediately – none of the following steps should be performed.
2. The dumb zombie will become paralyzed every other tick trying to figure out what to do. The 2nd, 4th, 6th, etc., calls to *doSomething()* for a dumb zombie are the

"paralysis" ticks for which *doSomething()* must return immediately – none of the following steps should be performed.

3. The dumb zombie must check to see if a person (either Penelope or one of the citizens on the level) is in front of it in the direction it is facing:
    a. It will compute vomit coordinates in the direction it is facing, SPRITE_WIDTH pixels away if it is facing left or right, or SPRITE_HEIGHT pixels away if it is facing up or down. So if the dumb zombie is at position (x,y) facing left, it would compute the vomit coordinates (x−SPRITE_WIDTH, y), i.e., (x−16, y).
    b. If there is a person whose Euclidean distance from the vomit coordinates is less than or equal to 10 pixels, then there is a 1 in 3 chance that the dumb zombie will vomit. If the zombie choses to vomit, it will:
        i. Introduce a vomit object into the game at the vomit coordinates.
        ii. Play the sound SOUND_ZOMBIE_VOMIT.
        iii. Immediately return and do nothing more this tick.
4. The dumb zombie will check to see if it needs a new movement plan because its current movement plan distance has reached zero. If so, the dumb zombie will:
    a. Pick a new random movement plan distance in the range 3 through 10 inclusive.
    b. Set its direction to a random direction (*up*, *down*, *left*, or *right*).
5. The dumb zombie will then determine a destination coordinate (dest_x, dest_y) that is 1 pixel in front of it in the direction it is facing.
6. If the movement to (dest_x, dest_y) would not cause the dumb zombie's bounding box to intersect with the bounding box[14] of any **wall**, **person** or other **zombie** objects, then:
    a. Update the dumb zombie's location to (dest_x, dest_y) using the *GraphObject* class's *moveTo()* method.
    b. Decrease the movement plan distance by 1.
7. Otherwise, the dumb zombie was blocked from moving by another wall, person or zombie, so set the movement plan distance to 0 (which will cause the dumb zombie to pick a new direction to move during the next tick).

What a Dumb Zombie Must Do In Other Circumstances

- A dumb zombie can be damaged by flames. If a flame overlaps with a dumb zombie, it will kill the dumb zombie. The dumb zombie must:
    - Set its own state to dead (so that it will be removed from the game by the *StudentWorld* object at the end of the current tick).
    - Play a sound effect to indicate that the dumb zombie died: SOUND_ZOMBIE_DIE.
    - Increase the player's score by 1000 points.
    - 1 in 10 dumb zombies are mindlessly carrying a vaccine goodie that they'll drop when they die. If this dumb zombie has a vaccine goodie, it will introduce a new vaccine goodie at its (x,y) coordinate by adding it to the *StudentWorld* object.

[14] See the **Determining Blocking of Movement** section.

- A dumb zombie cannot be infected by vomit.
- A dumb zombie blocks other objects from moving nearby/onto it. A dumb zombie's bounding box must never intersect with that of any other dumb zombie, smart zombie, person, or wall).
- A dumb zombie does not block flames.

Smart Zombie

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

What a Smart Zombie Must Do When It Is Created

When it is first created:

1. A smart zombie object must have an image ID of IID_ZOMBIE.
2. If the object was created because an infected person became a zombie, it must start at the location passed to its constructor. Otherwise, the object must start at the location on the level as specified in the current level's data file. The object's starting location in the level must be equal to (SPRITE_WIDTH * level_x, SPRITE_HEIGHT * level_y); its starting level_x and level_y can be obtained using our provided *Level* class. *Hint: The StudentWorld object can pass in this (x,y) location when constructing the object.*
3. A smart zombie has a direction of *right*.
4. A smart zombie has a depth of 0.
5. A smart zombie starts with a movement plan distance of 0.
6. A smart zombie starts out in an "alive" state.

What a Smart Zombie Must Do During a Tick

A smart zombie 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 smart zombie must do the following:

1. The smart zombie must check to see if it is currently alive. If not, then its *doSomething()* method must return immediately – none of the following steps should be performed.
2. The smart zombie will become paralyzed every other tick trying to figure out what to do. The 2nd, 4th, 6th, etc., calls to *doSomething()* for a smart zombie are the "paralysis" ticks for which *doSomething()* must return immediately – none of the following steps should be performed.
3. The smart zombie must check to see if a person (either Penelope or one of the citizens on the level) is in front of it in the direction it is facing:
    a. It will compute vomit coordinates in the direction it is facing, SPRITE_WIDTH pixels away if it is facing left or right, or

SPRITE_HEIGHT pixels away if it is facing up or down. So if the smart zombie is at position (x,y) facing left, it would compute the vomit coordinates (x−SPRITE_WIDTH, y), i.e., (x−16, y).

   a. If there is a person whose Euclidean distance from the vomit coordinates is less than or equal to 10 pixels, then there is a 1 in 3 chance that the smart zombie will vomit. If the zombie choses to vomit, it will:
      i. Introduce a vomit object into the game at the vomit coordinates.
      ii. Play the sound SOUND_ZOMBIE_VOMIT.
      iii. Immediately return and do nothing more this tick.

4. The smart zombie will then check to see if it needs a new movement plan because its current movement plan distance has reached zero. If so, the smart zombie will:
   a. Pick a new random movement plan distance in the range 3 through 10 inclusive.
   b. Select the person (Penelope or a citizen) closest to the smart zombie, i.e., the one whose Euclidean distance from the zombie is the smallest. If more than one person is the same smallest distance away, select one of them.
   c. Set its direction to a random direction:
      i. If the distance to the selected nearest person is more than 80 pixels away, the direction is chosen from *up*, *down*, *left*, and *right*.
      ii. Otherwise, the nearest person is less than or equal to 80 pixels away, the direction is chosen to be one that would cause the zombie to get closer to the person:
         1. If the zombie is on the same row or column as the person, choose the (only) direction that gets the zombie closer.
         2. Otherwise, choose randomly between the two directions (one horizontal and one vertical) that get the zombie closer.
5. The smart zombie will then determine a destination coordinate (dest_x, dest_y) that is 1 pixel in front of it in the direction it is facing.
6. If the movement to (dest_x, dest_y) would not cause the smart zombie's bounding box to intersect with the bounding box[15] of any **wall**, **person** or other **zombie** objects, then:
   b. Update the smart zombie's location to (dest_x, dest_y) using the *GraphObject* class's *moveTo()* method.
   c. Decrease the movement plan distance by 1.
7. Otherwise, the smart zombie was blocked from moving by another wall, person or zombie, so set the movement plan distance to 0 (which will cause the smart zombie to pick a new direction to move during the next tick).

What a Smart Zombie Must Do In Other Circumstances

- A smart zombie can be damaged by flames. If a flame overlaps with a smart zombie, it will kill the smart zombie. The smart zombie must:
  - Set its own state to dead (so that it will be removed from the game by the *StudentWorld* object at the end of the current tick).

[15] See the **Determining Blocking of Movement** section of this document.

    - Play a sound effect to indicate that the smart zombie died: SOUND_ZOMBIE_DIE.
    - Increase the player's score by 2000 points.
- A smart zombie cannot be infected by vomit.
- A smart zombie blocks other objects from moving nearby/onto it. A smart zombie's bounding box must never intersect with that of any other smart zombie, dumb zombie, person, or wall).
- A smart zombie does not block flames.

Citizen

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

What a Citizen Must Do When It Is Created

When it is first created:

1. A Citizen object must have an image ID of IID_CITIZEN.
8. The object must start at the location on the level as specified in the current level's data file. The object's starting location in the level must be equal to (SPRITE_WIDTH * level_x, SPRITE_HEIGHT * level_y); its starting level_x and level_y can be obtained using our provided *Level* class. *Hint: The StudentWorld object can pass in this (x,y) location when constructing the object.*
2. A Citizen has a direction of *right*.
3. A Citizen has a depth of 0.
4. A Citizen has an infection status of false.
5. A Citizen has an infection count of 0.
6. A Citizen starts out alive.

What a Citizen Must Do During a Tick

A citizen 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 citizen must do the following:

1. The citizen must check to see if it is currently alive. If not, then its *doSomething()* method must return immediately – none of the following steps should be performed.
2. If a citizen is infected (because of previously being vomited on by a zombie), it must increase its infection count by one. If the citizen's infection count reaches 500, it must:
    a. Set its state to dead (so that it will be removed from the game by the *StudentWorld* object at the end of the current tick).
    b. Play a SOUND_ZOMBIE_BORN sound effect.

    c. Decrease the player's score by 1000 for failing to save this citizen (the player's score could go negative!)
    d. Introduce a new zombie object into the same (x,y) coordinate as the former citizen by adding it to the *StudentWorld* object:
        i. There's a 70% chance a dumb zombie object will be introduced.
        ii. There's a 30% chance a smart zombie object will be introduced.
    e. Immediately return (since the citizen is now dead!)
3. Otherwise, the citizen will become paralyzed every other tick trying to figure out what to do. The 2nd, 4th, 6th, etc., calls to *doSomething()* for a citizen are the "paralysis" ticks for which *doSomething()* must return immediately – none of the following steps should be performed.
4. The citizen must determine its distance to Penelope: dist_p
5. The citizen must determine its distance to the nearest zombie: dist_z
6. If dist_p < dist_z or no zombies exist in the level (so dist_z is irrelevant), and the citizen's Euclidean distance from Penelope is less than or equal to 80 pixels (so the citizen wants to follow Penelope):
    a. If the citizen is on the same row or column as Penelope:
        i. If the citizen can move 2 pixels in the direction toward Penelope without being blocked[16] (by another citizen, Penelope, a zombie, or a wall), the citizen will
            1. Set its direction to be facing toward Penelope.
            2. Move 2 pixels in that direction using the *GraphObject* class's *moveTo()* method.
            3. Immediately return and do nothing more during the current tick.
        ii. Otherwise, the citizen would be blocked. Skip to step 7.
    b. If the citizen is not on the same row or column as Penelope, determine the one horizontal and the one vertical direction that would get the citizen closer to Penelope. Then
        i. Choose one of those two directions at random. If the the citizen can move 2 pixels in that direction without being blocked (by another citizen, Penelope, a zombie, or a wall), the citizen will
            1. Set its direction to that chosen direction.
            2. Move 2 pixels in that direction using the *GraphObject* class's *moveTo()* method.
            3. Immediately return and do nothing more during the current tick.
        ii. If the citizen would be blocked by moving in that chosen direction, then choose the other of the two directions that get the citizen closer to Penelope. If the the citizen can move 2 pixels in that other direction without being blocked (by another citizen, Penelope, a zombie, or a wall), the citizen will
            1. Set its direction to that other direction.
            2. Move 2 pixels in that direction using the *GraphObject* class's *moveTo()* method.

[16] See the **Determining Blocking of Movement** section of this document.

3. Immediately return and do nothing more during the current tick.

iii. If the citizen would be blocked in both of the two directions, continue to step 7.

7. If there is a zombie whose Euclidean distance from the citizen is less than or equal to 80 pixels, the citizen wants to run away, so:
    a. For each of the four directions *up*, *down*, *left* and *right*, the citizen must:
        i. Determine if the citizen is blocked from moving 2 pixels in that direction. If the citizen is blocked from moving there, it ignores this direction.
        ii. Otherwise, the citizen determines the distance to the nearest zombie of any type if it were to move to this new location.
    b. If none of the movement options would allow the citizen to get further away from the nearest zombie (e.g., right now it's 20 pixels away from the nearest zombie, but if it were to move in any of the four directions it would get even closer to some zombie), then *doSomething()* immediately returns – there is no better place for the citizen to move to than where it is now.
    c. Otherwise:
        i. Set the citizen's direction to the direction that will take it furthest away from the nearest zombie.
        ii. Move 2 pixels in that direction using the *GraphObject* class's *moveTo()* method.
        iii. Immediately return and do nothing more during the current tick.
8. At this point, there are no zombies whose Euclidean distance from the citizen is less than or equal to 80 pixels. The citizen does nothing this tick.

What a Citizen Must Do In Other Circumstances

- A citizen can be damaged by flames. If a flame overlaps[17] with a citizen, it will kill the citizen. The citizen must:
    - Set its own state to dead (so that it will be removed from the game by the *StudentWorld* object at the end of the current tick).
    - Play a sound effect to indicate that the citizen died: SOUND_CITIZEN_DIE.
    - Decrease the player's score by 1000 points.
- A citizen can be infected by vomit. When vomit overlaps with a citizen, the citizen's infection status becomes true.
- A citizen blocks other objects from moving nearby/onto it. A citizen's bounding box must never intersect with that of any other citizen, Penelope, dumb zombie, or smart zombie).
- A Citizen does not block flames.

[17] See the discussion of overlap in the **Determining Object Overlap** section.

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 Project 3!

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 NOT use the imageID (e.g., IID_ZOMBIE, IID_PLAYER, IID_WALL, etc.) or any value somehow related/derived from the imageID to determine the type of an object, or store such a value inside any of your objects as a member variable. 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 classes of behaviors:**

   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 member variable that serves the same purpose in both classes (e.g., m_amountOfOil), then move that member variable 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 member variable 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. **Never make a method public if it is only used directly by other methods within the same class that holds it. Make it private or protected instead.**

7. **Your StudentWorld methods should never return a vector, list or iterator to StudentWorld's private game objects or pointers to those objects. Only StudentWorld should know about all of its game objects and where they are. Instead StudentWorld should do all of the processing itself if an action needs to be taken on one or more game objects that it tracks.**

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());
    }
};
```

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.

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 the ZombieDash-skeleton-windows.zip archive into a folder on your hard drive. Double-click on ZombieDash.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 the ZombieDash-skeleton-mac.zip archive into a folder on your hard drive. Double-click on ZombieDash.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/ZombieDash/DerivedData/ZombieDash/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 Zombie Dash 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., Penelope, all types of zombies, goodies, projectiles, etc.):
    i. It must have a simple constructor.
    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 wall class, derived in some way from the base class described in 1 above:
    i. It must implement the specifications described in the Wall section above.
    ii. You may add any public/private member functions and private data members to your wall 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 Penelope 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 Penelope – see Penelope section for more details on where to initialize Penelope.
    ii. It must have an Image ID of IID_PLAYER.
    iii. 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 and this will not cause Penelope to move to a location that is blocked (by a wall), it updates Penelope's location appropriately. All this *doSomething()* method has to do is properly adjust Penelope's x,y coordinates using the *GraphObject*

class's *moveTo()* method, and our graphics system will automatically animate its movement it around the level!

iv. You may add other public/private member functions and private data members to your Penelope class as you see fit, so long as you use good object oriented programming style (e.g., you must not duplicate functionality across classes).

4. A limited version of the *StudentWorld* object.
    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 walls, but eventually it'll also include zombies, pits, projectiles like vomit/flames, etc.) as well as your Penelope object. You may ignore all other items in the game such as dumb zombies, projectiles, 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 Penelope and insert her into the level at the proper starting location (as specified by the level 01 data file). It must also create all of the walls and add them to the level (as specified by the level 01 data file). Your *init()* method may ignore any exits and other objects in the level - it must only deal with Penelope and walls.
    v. Implement the *move()* method in your *StudentWorld* class. During each tick, it must ask Penelope and other actors (just walls for now) to do something. Your *move()* method need not check to see if Penelope has died or not; you may assume for Part #1 that Penelope cannot die. Your *move()* method does not have to deal with any actors other than Penelope and the walls.
    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 walls and Penelope). 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 level with Penelope in its proper starting position and a bunch of walls surrounding the level. If your classes

work properly, you should be able to move Penelope around the level using the directional keys so long as she doesn't move into a wall.

Your Part #1 solution may actually do more than what is specified above; for example, if you are making good progress, try to add exit 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 dumb zombies, smart zombies, goodies, projectiles like vomit or flames, pits, exits, 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 Penelope, wall, 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, Penelope, and Wall 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 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 Zombie Dash 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 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 Zombie Dash are able to sneeze, and each type of actor sneezes in a different way."
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 Flame class." or "My smart zombie doesn't work correctly yet so I treat it like a dumb zombie 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."
4. A description of how you tested each of your classes (1-2 paragraphs per class).

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 your 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!