Project 3
Super Peach Sisters

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

Time due:
Part 1: 11 PM, Sunday, February 20
Part 2: 11 PM, Sunday, February 27

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

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

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

Table of Contents

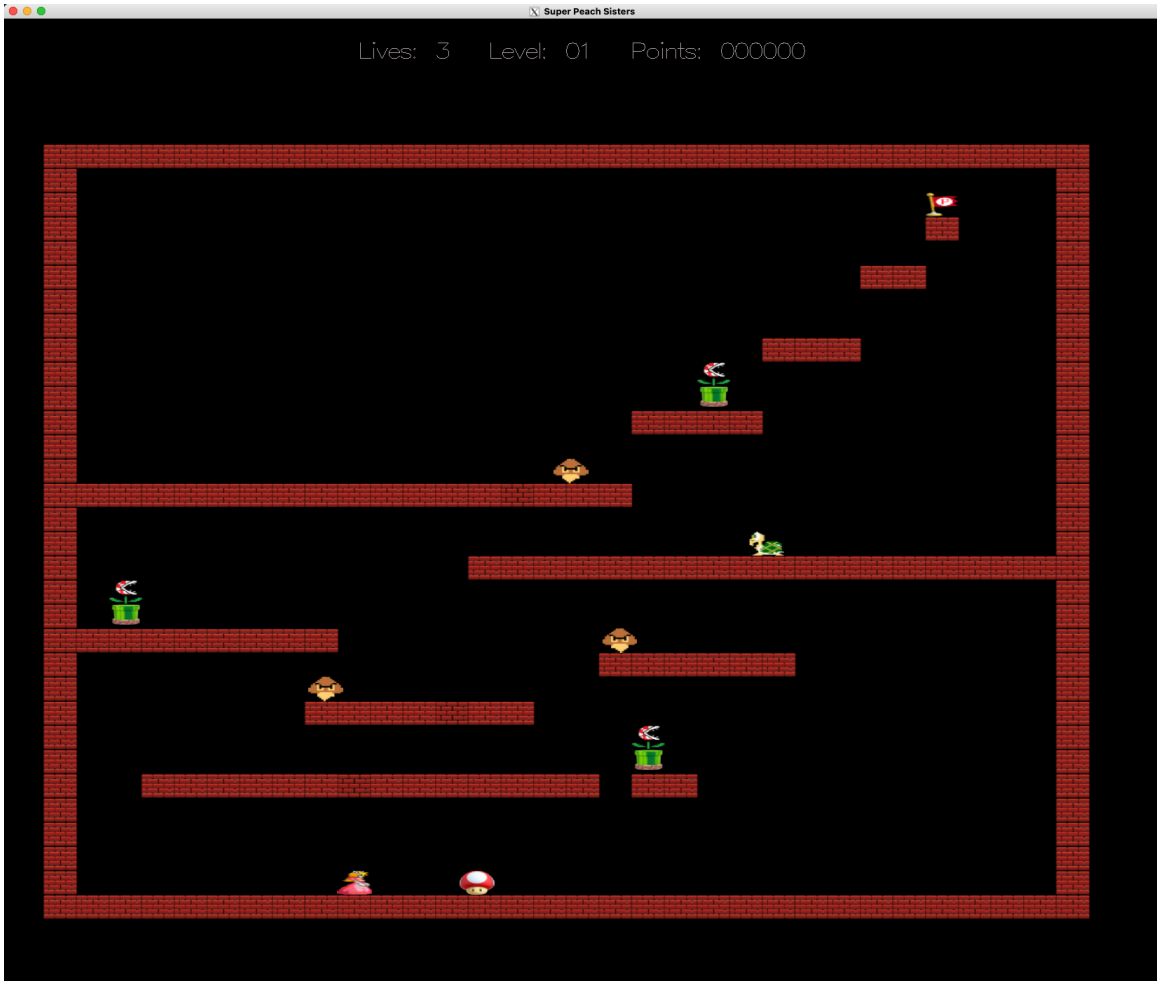
Introduction

NachenGames corporate spies have learned that SmallSoft is planning to release a new video game called Super Peach Sisters, a sequel to Super Mario Brothers, 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 Super Peach Sisters 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.

Super Peach Sisters is set in the Mushroom Queendom, fictional land of crazy creatures like fire-breathing piranhas, koopas and goombas. Unfortunately, last week Mario was taken hostage by the evil greenhouse gas producer named Yerac Grebnehcan (who is well known for producing huge amounts of methane), and Peach must face a number of increasingly difficult levels to rescue him.

Each level contains a number of challenges, which include dangerous koopas, goombas, and fire-breathing Piranhas that grow from pipes. Peach must avoid or destroy these creatures to reach the flag on each level, which takes her to the next level. Once Peach reaches the final level, she must then get to Mario to save him and win the game. Each level contains a number of goodies that Peach can reveal by bonking her head against blocks. These goodies include Stars (which grant invincibility to Peach), Mushrooms (which enable Peach to double her jumping height), and Flowers which enables Peach to shoot fireballs at her opponents.

Here's a screenshot from the game:



In the image above, we can see Peach in the bottom left of the screen. A mushroom (which grants Peach extra-high jumping skill) is near her to the right. We also see three piranhas sitting atop their pipes. Be careful, because they shoot fire from their mouths if Peach gets too close. The brown pointy-headed characters are goombas, and the turtles are koopas, both types of monster that cause Peach to shrivel up if she comes into contact with them (unless she has a special power due to picking up a star, mushroom or flower). At the top right, we see a flag which Peach must reach to complete the current level and advance to the next level.

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

- Left arrow key or the ‘a’ key: Moves Peach toward the left
- Right arrow key or the ‘d’ key: Moves Peach toward the right
- Up arrow key or the ‘w’ key: Causes peach to jump up
- Space bar: Fires a fireball, if Peach has any
- The ‘q’ key: Quits the game
- The ‘f’ and 'r' keys: Freeze and resume the game to aid with debugging

Game Details

In Super Peach Sisters, Peach starts out a new game with three lives and continues to play until all of her lives have been exhausted. There are multiple levels in Super Peach Sisters, beginning with level 1 (NOT zero), and during each level except for the last Peach must reach a flag to continue to the next level. On the last level, Peach must reach (and thus save) Mario in order to win the entire game (there is no flag on the last level).

The Super Peach Sisters 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., Peach, koopas, stars, shells, etc.) will have an x coordinate in the range 0 to VIEW_WIDTH−1 inclusive, and a y coordinate in the range 0 to VIEW_HEIGHT−1 inclusive. These constants also may be found in the same header file.

Each level has its layout defined in a data file, such as level01.txt or level02.txt. You can find example level files in the Assets folder that we provide, and you may modify these data files or add additional ones to create new and exciting levels. In every level except for the last level, there must be at least one flag that Peach must reach in order to advance to the next level. The last level must not contain a flag, but must contain Mario, who Peach must reach to win the game. For more details on the format of the level data files, please see the Level Data File section.

Peach may move left or right, and may also jump. Because of the strange physics of the Mushroom Queendom, Peach can actually move left and right while jumping in the air as well, which makes it easier for her to jump from platform to platform.

As Peach works through each level, she must avoid coming into contact with any of her enemies. Coming into contact with an enemy (unless she currently has a superpower due to picking up a goodie) will cause her to die, either resetting the level, or ending the game if she has no lives left. Similarly, Peach must avoid all fireballs shot by Piranhas or she will die. If Peach dies three times, the game is over.

At the beginning of each level and when Peach restarts the current level because she has died, the level will be reset to an initial state. That is, all actors will be placed in their initial positions and start with their initial states. All blocks will be reset, so they again release any goodies when bonked by Peach, etc. Peach also starts out with no special powers (i.e., no invincibility, jump power or shooting power).

Some of the blocks in each level may hold special goodies, and if Peach bonks them with her head by jumping into them, she can cause them to release their goodie, which Peach

can pick up. These special blocks look just like all of the other blocks, so you will need to learn where they are in each level by trying to bonk blocks here and there.

There are three different types of goodies: star goodies, mushroom goodies and flower goodies. Star goodies give Peach invincibility for a limited amount of time, so she can't be injured by coming into contact with enemies like koopas, goombas, piranhas, or piranha fireballs. Mushroom goodies give Peach the ability to jump extra high, allowing her to reach platforms that would otherwise be out of reach. Finally, flower goodies enable Peach to shoot fireballs to destroy her enemies.

The effect of a mushroom or a flower goodie lasts until either Peach comes into contact with an enemy (in which case she loses the goodie's power, but doesn't die), or she completes the current level. Similarly, invincibility from the star goodie lasts about 10 seconds or until Peach completes the current level, whichever comes first.

Assuming Peach has Fire Power (granted to her by picking up a flower goodie), the player may hit the spacebar to have peach shoot a fireball. A fireball will destroy the first enemy it contacts, and then disappears from the game. A fireball will fly forward until either it hits an enemy, or it hits a block or a pipe in a horizontal direction (simply falling down onto a block or pipe will not cause the fireball to dissipate). Beyond damaging enemies, fireballs will otherwise pass over all other objects in the game. For example, fireballs will pass over flags, Mario, mushrooms, flowers, stars and shells without damaging them. Fireballs fall to the ground as they move forward, so for example, if one flies past the edge of the current platform of blocks it will start falling down.

Each level may have up to three different types of enemies: koopas, goombas and piranhas. Koopas and goombas simply move left and right on their platforms, stopping before they fall off the edge of their platform or if they run into a block or pipe, and then turning to move in the opposite direction. Piranhas don't move, but will turn to face Peach if she gets close enough. Piranhas will also shoot fireballs at Peach if she gets too close. A fireball fired by a piranha will hurt Peach just as if it were an enemy coming into contact with Peach. A fireball fired by a piranha will fly forward until it hits either a block or a pipe, but beyond damaging Peach, it will otherwise pass over all other objects in the game (goombas, koopas, other piranhas, flags, Mario, mushrooms, flowers, stars, and shells) without damaging them. A fireballs fired by piranha falls to the ground as it moves forward, so for example, if one flies past the edge of a platform of blocks, it will start falling down to the level below.

If Peach destroys an enemy (either by contacting them while invincible, or by shooting them with a fireball), the enemy will disappear from the level and Peach will get points. When Peach destroys a koopa, it will produce a shell which will travel like a projectile until it hits an block, pipe or an enemy, at which point it will dissipate (A shell behaves identically to a fireball fired by Peach, but looks different). Shells float right past Peach, but will destroy all enemies (koopas, goombas, or piranhas) they come into contact with.

As Peach plays, if she comes into contact with an enemy (e.g., goomba, koopa or piranha) and she does not have any type of power (e.g., Star Power) she will lose a life, and the level will reset (or the game will be over if she has no more lives). If Peach has Star Power (invincibility) then coming into contact with an enemy will not hurt Peach, and will immediately destroy the enemy. If Peach has Jump Power and/or Shoot Power (but not Star Power) and comes into contact with an enemy, then she will immediately lose those power(s) but she will not lose a life, and her enemy will not be hurt either. In this case, Peach will have a short amount of temporary invincibility so she does not immediately die on the next tick as the enemy continues to maintain contact with her (as it moves). If, after losing a life, Peach has one or more remaining lives left, the level and all of the actors (Peach, goombas, etc.) are reset to their initial state and Peach must again play the level from scratch. If Peach dies and has no lives left, then the game is over.

If Peach reaches the flag on the level, then the level ends and Peach advances to the next level. If Peach reaches Mario on the final level, then Peach wins the entire game and the game is over.

Points are awarded as follows:

- When Peach disposes of a koopa, goomba or piranha: 100 points
- When Peach picks up a mushroom: 75 points
- When Peach picks up a flower: 50 points
- When Peach picks up a star: 100 points
- When Peach reaches a flag: 1000 points
- When Peach finally reaches (and saves) Mario: 1000 points

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

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

- You must give each object – including Peach, goombas, koopas, piranhas, blocks, projectiles, goodies, etc. - a chance to do something – e.g., move, shoot, die, etc.
- You must check to see if Peach has died. If so, you must indicate this to our game framework (we'll tell you how later) so the level can end and potentially restart fresh, if Peach has more lives.
- You must delete/remove all dead objects from the game – this includes goombas, koopas and piranhas that have been destroyed by Peach, fireballs fired by Peach or Piranhas that have dissipated, goodies that have been picked up, etc.
- Your code may also need to introduce one or more new objects into the game – for instance, a new flower, mushroom or star may be introduced when Peach bonks a block, or a piranha might introduce a fireball when shooting at Peach.
- You must update the game statistics line at the top of the screen, including the number of remaining lives Peach has, the player's current score, the current level number, and any powers Peach has (e.g., Star Power).

- You must check to see if Peach has completed the current level (by reaching a flag or Mario), and if so ending the current level so Peach may advance to the next level or win the game.

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

Lives: 2 Level: 5 Points: 500 StarPower! ShootPower! JumpPower!

Where the items in red are ALWAYS required, and the items in blue are required only if Peach has a particular power. For example, if Peach just has Jump Power, the line might look like this:

Lives: 2 Level: 5 Points: 500 JumpPower!

Each of the first three stats of the status line must be separated from each other by exactly two spaces. For example, between "Lives: 2" and "Level:" there must be two spaces. The powers (e.g., JumpPower!), if present, must each be separated by a single space, and are separated from the point count by a single space as well. You may find the *Stringstreams* writeup on the main class web site to be helpful.

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

```
// Make a sound effect when Peach shoots a fireball
pointerToMyWorld->playSound(SOUND_PLAYER_FIRE);
```

- You must play a SOUND_PLAYER_FIRE sound any time Peach successfully shoots a fireball.
- You must play a SOUND_PLAYER_JUMP sound any time Peach jumps.
- You must play a SOUND_PLAYER_POWERUP sound any time Peach picks up a goodie like a star, mushroom or flower goodie.
- You must play a SOUND_PLAYER_BONK sound any time Peach bonks a block with her head by jumping up into it.
- You must play a SOUND_PLAYER_KICK sound any time Peach destroys an enemy by coming into contact with them while invincible with Star Power.
- You must play a SOUND_PLAYER_HURT sound any time Peach is hurt (but does not die) by contacting an enemy or being hit by a fireball fired by piranha. This would happen if Peach has Jump Power or Shoot Power.
- You must play a SOUND_PLAYER_DIE sound any time Peach loses a life by contacting an enemy or being hit by a fireball fired by piranha (while not protected by a power).
- You must play a SOUND_PIRANHA_FIRE sound any time a piranha shoots a fireball.
- You must play a SOUND_GAME_OVER sound any time Peach runs out of lives and the game is over.
- You must play a SOUND_FINISHED_LEVEL sound any time Peach finishes the current level.

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

Determining Object Overlap

In a video game, it's often important to determine if two game objects come into contact with each other (are they close enough that they touch/overlap and therefore interact). For example, if Peach shoots a fireball, does it come into contact with a nearby goomba, koopa or piranha? Or when Peach moves near a goodie, does the goodie get close enough to grant her its special power?

For the purposes of this project, to determine if two objects A and B overlap, simply check to see if their bounding squares overlap. Each object in the game (e.g., a block, koopa, Peach, goodies, pipes, fireballs) is represented by a rectangle that is SPRITE_WIDTH pixels wide and SPRITE_HEIGHT pixels high. Each object's location is denoted by the coordinates of the bottom-left corner of that rectangle. So if an object is at location (x,y), it will extend to (x+SPRITE_WIDTH−1, y+SPRITE_HEIGHT−1). It is therefore a pretty trivial problem to check if two boxes have any overlap with a few if statements. These constants are defined in *GameConstants.h*.

You must use this approach to detect overlap between two different objects any time this specification requires you to detect overlap conditions.

So how does a video game work?

Fundamentally, a video game is composed of a bunch of game objects; in Super Peach Sisters, those objects include Peach, enemies (e.g., goombas, koopas and piranhas), goodies (e.g., stars, mushrooms, flowers), projectiles (e.g., fireballs fired by peach and piranhas), shells, blocks and pipes, flags and Mario. 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., an enemy 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 Peach, the algorithm that controls the Peach object is the player's own brain and hand, and the keyboard! In the case of other actors (e.g., a goomba), each object has an internal autonomous algorithm and state that dictates how the object behaves in the game world.

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

During a given tick, the game calls upon each object's behavioral algorithm and asks the object to perform its behavior. When asked to perform its behavior, each object's behavioral algorithm must decide what to do and then make a change to the object's state (e.g., move the object one pixel to the left, and one pixel down), or change another

objects' state (e.g., when a flower goodie detects that it overlaps with Peach, it will grant Peach the Fire Power). Typically the behavior exhibited by an object during a single tick is limited in order to ensure that the gameplay is smooth and that things don't move too quickly and confuse the player. For example, a goomba will move just a few pixels forward, rather than moving ten or more pixels per tick; a goomba moving, say, 20 pixels in a single tick would confuse the user, because humans are used to seeing smooth movement in video games, not jerky shifts.

After the current tick is over and all actors have had a chance to adjust their state (and possibly adjust other actors' states), the graphical framework that we provide animates the actors onto the screen in their new configuration. So if a goomba changed its location from (x=10, y=50) to (x=9, y=50) (i.e., moved one pixel left), then our game framework would erase the graphic of the goomba from location (10, 50) on the screen and draw the goomba's graphic at (9, 50) 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 goomba doesn't move 3 inches away from where it was during the last tick, but instead moves a few millimeters 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 level.

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

Cleanup: The player has lost a life (but has more lives left), or the player has completed the current level, or the player has lost all of their lives and the game is over. This phase frees all of the objects in the world (e.g., Peach, goombas, koopas, piranhas, goodies, fireballs, blocks, pipes, flags, Mario, shells, 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 with initial states, and game play starts from scratch for the level.

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

```
while (the player has lives left)
{
   Prompt_the_user_to_start_playing    // "press any key to start"
   Initialize_the_game_world

   while (Peach 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
        Delete_any_dead_actors_from_the_world

          // we write this code to handle the animation for you
        Animate_each_actor_to_the_screen;
        Sleep for 50ms to give the user time to react
   }
     // Peach died
   Clean up_all_game_world_objects
   if (Peach still has lives left)
      Prompt_the_player_to_continue
}

Tell_the_player_the_game_is_over
```

And here is what a `Tell_all_actors_to_do_something` function might look like:

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

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

```
class Goomba: public SomeOtherClass
{
   public:
       virtual void doSomething()
       {
           If Peach overlaps with me, then
                  Attempt to damage Peach
           Else if my current movement direction is right:
                  If I can move to the right without hitting a block/pipe and without
                     falling off a ledge, then:
                     Move one pixel right
                  Else
                     Switch my direction to face left
           Else if my current movement direction is left:
```

```
                    If I can move to the left without hitting a block/pipe and without
                        falling off a ledge, then:
                        Move one pixel left
                    Else
                        Switch my direction to face right
        }
        ...
};
```

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

```
class Peach: public …
{
   public:
        virtual void doSomething()
        {
             Try to get user input (if any is available)
               If the user pressed the LEFT arrow key and the
                  position to the left is not blocked, then
                   Set Peach's direction to face left
                   Move Peach 4 pixels to the left
               If the user pressed the RIGHT arrow key and the
                  position to the right is not blocked, then
                   Set Peach's direction to face right
                   Move Peach 4 pixels to the right
               ...
               If user pressed space and Peach has Shoot Power, then
                  Introduce a new fireball object in front of
                         Peach, with an initial direction matching Peach's
                         current direction
        }
        ...
};
```

What Do You Have to Do?

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

1. You must create a class called *StudentWorld* that is responsible for keeping track of your game world and all of the actors/objects (e.g., Peach, goombas, koopas, piranhas, fireballs, stars, flowers and mushrooms, shells, blocks, pipes, flags, and Mario) that are inside the game.
2. You must create a class to represent Peach in the game.
3. You must create classes for goombas, koopas, piranha, fireballs, stars, flowers and mushrooms, shells, blocks, pipes, flags, and Mario, etc., as well as any additional base classes (e.g., a goodie base class if you find it convenient, since goodies share many things in common) that help you implement your actors.

You Have to Create the StudentWorld Class

Your *StudentWorld* class is responsible for orchestrating virtually all gameplay – it keeps track of the entire game world (each level and all of its inhabitants such as goombas, blocks, piranhas, Peach, goodies, 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 koopa dies, a fireball flies into a pipe and dissipates, etc.), and destroying *all* of the actors in the game world when the player loses a life or advances to the next level.

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

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

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

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

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

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

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

Once a level has been prepared with a call to the *init()* method, our game framework will repeatedly call the *StudentWorld's move()* method, at a rate of roughly 20 times per second. Each time the *move()* method is called, it must run a single tick of the game. This means that it is responsible for asking each of the game actors (e.g., Peach, each goomba, koopa, piranha, each goodie, each fireball, each shell, etc.) to try to do something: e.g., move themselves and/or perform their specified behavior. This method might also introduce new actors into the game, for instance adding a new flower goodie above a block that Peach bonked with her head. Finally, this method is responsible for disposing

of (i.e., deleting) actors that need to disappear during a given tick (e.g., fireball that runs into an enemy and disappears, a dead enemy, etc.). For example, if a goomba is shot by Peach's fireball, then its state should be set to dead, and then after all of the alive actors in the game get a chance to do something during the tick, the *move()* method should remove that goomba from the game world (by deleting its object and removing any pointers 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 Peach completes the current level or loses a life (e.g., by being damaged due to contact with an enemy or piranha-fired fireball). The *cleanup()* method is responsible for freeing all actors (e.g., all enemies, blocks, pipes, fireballs, shells, flags, Mario, etc.) that are currently in the game. This includes all actors created during either the *init()* method or introduced during subsequent game play by the actors in the game (e.g., a fireball that was added to the screen by a piranha, a goodie that was added after Peach bonked a block) that have not yet been removed from the game.

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

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

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

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

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

decLives() reduces the number of Peach lives by one.

incLives() increases the number of Peach lives by one.

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

increaseScore() is used by a *StudentWorld* object (or your other classes) to increase the user's score upon successfully destroying enemies, picking up a goodie of some sort or finishing a level. When your code calls this method, you must specify how many points

the user gets (e.g., 100 points for destroying a goomba). 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.:

Lives: 2 Level: 5 Points: 500 StarPower! JumpPower!

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

```
KEY_PRESS_LEFT
KEY_PRESS_RIGHT
KEY_PRESS_UP
KEY_PRESS_DOWN
KEY_PRESS_SPACE
```

The *playSound()* method can be used to play a sound effect when an important event happens during the game (e.g., Peach shoots a fireball or picks up a goodie). You can find constants (e.g., SOUND_PLAYER_FIRE) that describes 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 Peach attacks an enemy

if (peachOverlapsWithAnEnemy() && peachHasStarPower())
    studentWorldObject->playSound(SOUND_PLAYER_KICK);
```

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 Peach object into the game world. Every time a level starts or restarts, Peach starts out fully initialized (with the no special powers active, etc.) in her initial location as specified by the current level data file.
3. Allocate and insert all of the blocks, pipes, flags, enemies and Mario into the game world as described below.

Your *init()* method must construct a representation of your world and store this in a *StudentWorld* object. It is **required** that you keep track of all of the actors (Peach, enemies, blocks, flags, goodies, etc.) in a **single** STL collection such as a *list, map* or *vector*. (To do so, we recommend using a container of pointers to the actors). If you like, your *StudentWorld* object may keep a separate pointer to the Peach object rather than keeping a pointer to that object in the container with the other actor pointers; Peach 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 actors that need to be destroyed on this level before Peach can advance to the next level.

The *init()* method must load information from a level data file for the current level into a Level object, and use this to populate the current level with objects at the proper locations. (Details appear later in the Level Data File section.) The following types of objects must be populated:

- Blocks and pipes, including special blocks that produce goodies when bonked
- Peach
- Flags
- Mario
- Goombas, koopas and piranhas

A level of the game is a 32x32 grid, each spot possibly occupied by an object. If the Level object specifies that a given object is at position lx,ly in the grid (where 0 <= lx < 32, and 0 <= ly < 32), then the actual object must be placed at location (x,y) on the screen, where x = lx * SPRITE_WIDTH and y = ly * SPRITE_HEIGHT. (Notice that 32 comes from VIEW_WIDTH (i.e., 256) / SPRITE_WIDTH (i.e., 8), and similarly for the heights.)

The *init()* method returns `GWSTATUS_LEVEL_ERROR` if no level data file exists for the current level or if the file is improperly formatted. Otherwise, *init()* returns `GWSTATUS_CONTINUE_GAME`. These constants are defined in *GameConstants.h*.

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 active in the game world to do something (e.g., ask a goomba to move itself, ask a goodie to check if it overlaps with Peach, and if so, grant her its special power, give Peach a chance to move, jump, or shoot a fireball, etc.).

    a. If an actor does something that causes Peach to die (e.g., a fireball overlaps with Peach while she doesn't have a Power), then the *move()* method should:
        1. Play a SOUND_PLAYER_DIE sound using playSound().
        2. Immediately return with a value of `GWSTATUS_PLAYER_DIED`.
2. If Peach has reached a flag (overlaps with them), then it's time to advance to the next level. In this case, the *move()* method must:
    a. Play a SOUND_FINISHED_LEVEL sound using playSound().
    b. Immediately return with a value of `GWSTATUS_FINISHED_LEVEL`.
3. If Peach has reached Mario (overlaps with him), then the player has won and the game is over. In this case, the *move()* method must:
    a. Play a SOUND_GAME_OVER sound using playSound().
    b. Immediately return with a value of `GWSTATUS_PLAYER_WON`.
4. It must then delete any actors that have died during this tick (e.g., a goomba that was hit by a fireball shot by Peach, or who overlaps with Peach while she's got Star Power should be removed from the game world, as should a goodie that disappeared because it overlapped with Peach and activated, a fireball that has contacted something and needs to disappear, etc.).
5. It must update the status text on the top of the screen with the latest information (e.g., the user's current score, their number of lives, the Powers they have, etc.).
6. If the level is not over and Peach has not died, then the function must return `GWSTATUS_CONTINUE_GAME`.

(All of these constants are defined in *GameConstants.h*)

The return value `GWSTATUS_PLAYER_DIED` indicates that Peach died during the current tick, and instructs our provided framework code to tell the user the bad news and restart the level if the player has more lives left (or end the game if they are out of lives). If your *move()* method returns this value and Peach 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 return value `GWSTATUS_CONTINUE_GAME` indicates that the tick completed without Peach dying and that Peach has not yet completed the current level. Therefore, the game play 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 return values `GWSTATUS_FINISHED_LEVEL and GWSTATUS_WON_GAME` indicate that Peach has completed the current level or won the game (that is, she successfully reached a flag or Mario). If your *move()* method returns one of these values, then the current level is over, and our framework will call your *cleanup()* method to destroy the level. If the game has not been won, our framework will then advance to the next level, call your *init()* method to prepare that level for play, etc..

IMPORTANT NOTE: The skeleton code that we provide to you is hard-coded to return a `GWSTATUS_PLAYER_DIED` status value from our dummy version of the *move()* method.

Unless you implement something that returns `GWSTATUS_CONTINUE_GAME` your game will not display any objects on the screen! So if the screen just immediately tells you that you lost a life once you start playing, you'll know why!

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

```
int StudentWorld::move()
{
    // The term "actors" refers to all actors, e.g., Peach, goodies,
    // enemies, flags, blocks, pipes, fireballs, etc.

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

          if (Peach died during this tick) {
                play dying sound
                return GWSTATUS_PLAYER_DIED;
          }

          if (Peach reached Mario) {
                play game over sound
                return GWSTATUS_WON_GAME;
          }

          if (Peach competed the current level) {
                play completed level sound
                return GWSTATUS_FINISHED_LEVEL;
          }
        }
    }

    // Remove newly-dead actors after each tick
    remove 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 Peach, enemies like goombas, koopas and piranhas, fireballs (fired by Peach or a piranha), shells, blocks, flags, Mario, etc.

Your *move()* method must iterate over every actor that's active in the game (i.e., held by your *StudentWorld* object) and ask it to do something by calling a member function in the actor's object named *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 goomba might move four pixels left
- A piranha might shoot a fireball
- Peach might jump up
- A previously-fired fireball may disappear due to smacking into a block or hitting an enemy

It is possible that one actor (e.g., a fireball) may cause the death of another actor (e.g., a goomba or koopa) during the current tick. An actor that has died earlier in the current tick 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., Peach must not be injured by a koopa that was just hit by a fireball, but also overlaps with her).

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 any key other than 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 game play, 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 goomba is killed by a fireball, then it should be noted as dead, and at the end of the tick, its *pointer* should be removed from the StudentWorld's container of active objects, and the goomba 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 fireball has impacted an enemy or a block/pipe, it must disappear from the screen and its object needs to be deleted as well. (Hint: Each of your actors could maintain a dead/alive status data member.)

cleanUp() Details

When your *cleanUp()* method is called by our game framework, it means that Peach lost a life (e.g., she came into contact with a fireball or enemy without having a Power) or has completed the current level. In either case, every actor in the entire game (Peach and every enemy, goodie, projectile, block, flag, 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 Peach dies. Instead, this method will be called by our code when *move()* returns an appropriate status.

Level Data File

As mentioned, every level of Super Peach Sisters has a different layout. The layout for each level is stored in a plain text data file that you can edit with Windows Notepad, macOS's textedit, vi/vim, emacs, nano, etc. The file "level01.txt" holds the details for the first level, "level02.txt" holds the details for the second level, etc. The numbers in the names are two digits; we will never test your program with Mario never appearing somewhere in the first 99 levels. These level data files are stored in the Assets directory along with all images and sound files.

An example data file is shown below — you can modify our data files to create wacky new levels, or add your own new level data files to add new levels, if you like.

level01.txt:

```
################################
#                              #
#                            F #
#                            # #
#                              #
#                          ##  #
#                              #
#                              #
#                        ###   #
#                      P       #
#                      I       #
#                    ####      #
#                              #
#    G                         #
###############%###            #
#                              #
#                             K #
#             #####*############
# P                            #
# I                            #
#########            G         #
#                  ######      #
#          G                   #
#        ####%##               #
#                    p         #
#                    I         #
#  ######^####### ##           #
#                              #
#                              #
#                              #
# @                            #
################################
```

As you can see, the data file contains a 32x32 grid of different characters that represent the different actors/things in the level. The file *GameConstants.h* defines `GRID_WIDTH` and `GRID_HEIGHT` to each be 32. Valid characters for your level data file are:

The **@** character specifies the starting location of Peach when she starts a level. Peach should also restart at this location if she dies and must replay the current level. Each level must have exactly one Peach.

The **#** character represents a regular block. The perimeter of each level MUST be surrounded completely by blocks or our Level class will refuse to load it.

The **I** character represents a pipe. Pipes are identical to blocks in that they block movement (they just look different).

The ^ character represents a special block that holds a mushroom goodie in it. The first time Peach bonks such a block with her head, the block will release the goodie.

The **%** character represents a special block that holds a flower goodie in it. The first time Peach bonks such a block with her head, the block will release the goodie.

The ***** character represents a special block that holds a star goodie in it. The first time Peach bonks such a block with her head, the block will release the goodie.

The **F** character represents a flag. Each level except for the last level must have at least one flag. The last level must have no flags.

The **M** character represents Mario. The last level must have exactly one Mario.

The **G** character represents a goomba monster.

The **K** character represents a koopa monster.

The **P** character represents a piranha monster.

All **space** characters represent empty locations where Peach, goombas and koopas may move within the level.

You must not have tab characters in your level data files, so make sure your text editor inserts only spaces, not tabs.

The Level Class

We are providing you with a class that can load level data files for you. The class is called *Level* and may be found in our provided *Level.h* file. Here's how you might use this class:

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

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

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

                Level::GridEntry ge;

                ge = lev.getContentsOf(5, 10);        // x=5, y=10
                switch (ge)
```

```
            {
                case Level::empty:
                    cout << "Location 5,10 is empty" << endl;
                    break;
                case Level::koopa:
                    cout << "Location 5,10 starts with a koopa" << endl;
                    break;
                case Level::goomba:
                    cout << "Location 5,10 starts with a goomba" << endl;
                    break;
                case Level::peach:
                    cout << "Location 5,10 is where Peach starts" << endl;
                    break;
                case Level::flag:
                    cout << "Location 5,10 is where a flag is" << endl;
                    break;
                case Level::block:
                    cout << "Location 5,10 holds a regular block" << endl;
                    break;
                case Level::star_goodie_block:
                    cout << "Location 5,10 has a star goodie block" << 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 your *StudentWorld* class inherit from *GameWorld* might also be useful too! You may also find the *Stringstreams* writeup on the main class web site to be helpful, since unlike this example that hard-codes a file name "level01.txt", you will generate a file name from a level number.

You Have to Create the Classes for All Actors

Peach has a number of different actors, including:

- Peach
- Blocks
- Pipes
- Flags
- Mario
- Flowers
- Mushrooms
- Stars
- Piranha-fired fireballs
- Peach-fired fireballs
- Shells
- Goomba
- Koopa
- Piranha

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

Now of course, many of your game actors will share things in common – for instance, every one of the actors in the game (goombas, koopas, piranha, Peach, goodies, etc.) will need to have an alive/dead status. Many of them can potentially be damaged or bonked and will react in response. Certain objects like Stars, Flowers and Mushrooms as well as fireballs and shells "activate" when they come into contact with a proper target, etc.

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

Your grade on this project will depend upon your ability to intelligently create a set of classes that follow good object-oriented design principles. **Your classes must never duplicate non-trivial code or a data member** – 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 Star section and the Mushroom section, or in the koopa and goomba sections) you must make sure to identify common behaviors and move these into proper base classes. NEVER duplicate non-trivial behaviors (aka methods and member variables) across classes that can be moved into a base class! A non-trivial behavior is one that is more than a single statement long.

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 Goomba: 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, int startX, int startY,
            int startDirection = 0, int depth = 0, double size = 1.0);
double getX() const;             // in pixels (0-255)
double getY() const;             // in pixels (0-255)
void moveTo(double x, double y); // in pixels (0-255)
int getDirection() const;        // in degrees (0-359)
void setDirection(int d);        // in degrees (0-359)
void increaseAnimationNumber();  // forces a sprite to animate frames
```

You may use any of these member functions in your derived classes, but you **must not** use any other member functions found inside of *GraphObject* in your other classes (even if they are public in our class). You **must not** redefine any of these methods in your derived classes **unless** they are marked as virtual in our base class.

GraphObject(int imageID,
int startX, *// column first - x*
int startY, *// then row - y*
int startDirection,
int depth = 0,
double size = 1.0);

is the constructor for a new *GraphObject*. When you construct a new *GraphObject*, you must specify an image ID that indicates how the *GraphObject* should be displayed on screen (e.g., as a koopa, goomba, piranha, Peach, mushroom, star, block, flag, etc.). You must also specify the initial (x, y) location of the object. The x value may range from 0 to VIEW_WIDTH-1 inclusive, and the y value may range from 0 to VIEW_HEIGHT-1 inclusive (these constants are defined in our provided header file GameConstants.h). Notice that you pass the coordinates as x, y (i.e., column, row starting from bottom left, and ***not*** row, column). You may also specify the initial direction an object is facing as an angle between 0-359 degrees (though you'll only use values of 0 or 180 for this project).

The *imageID* is one of the following IDs, found in *GameConstants.h*:

IID_PEACH
IID_KOOPA
IID_GOOMBA
IID_SHELL
IID_PIRANHA
IID_MARIO
IID_BLOCK
IID_PIPE
IID_STAR

IID_FLOWER
IID_MUSHROOM
IID_FLAG
IID_PIRANHA_FIRE
IID_PEACH_FIRE

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

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. That includes strings, ints, enums, or any other clever data elements you can think of. Only our GraphObject class may store the imageId or related value.

You MUST not use the imageID to identify object types, e.g., determine that a particular actor is a goomba by checking its image ID is IID_GOOMBA). Nor may you store other similar data (e.g., a string "goomba", enum, etc.) based on the image ID to identify your object types. For hints on how to distinguish between different actors, see the Object Oriented Programming Tips section later in this document!

getX() and *getY()* are used to determine a *GraphObject's* current location in the level. Since each *GraphObject* maintains its own (x, y) location, this means that your derived classes **must NOT** also have x or y 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 and also updates our sprite frames if the image has multiple animated frames. For example, if a goomba'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 *0-359* (though for this project, the direction would be only 0 or 180).

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

The use of *increaseAnimationNumber()* is specified in the requirements for Piranha.

Peach

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

What a Peach Object Must Do When It Is Created

When it is first created:

1. A Peach object must have an image ID of IID_PEACH.
2. A Peach object starts out alive.
3. A Peach object starts out with 1 hit point (health point).
4. A Peach object has a starting (x,y) position based on the current level data file. Your *StudentWorld* object can pass in that position when constructing this object.
5. A Peach object has a direction of 0 degrees.
6. A Peach object has a graphical depth of 0.
7. A Peach object has a size of 1.0.
8. A Peach object starts out with no temporary invincibility.
9. A Peach object starts with no special powers (i.e., no Star Power, Shoot Power, or Jump Power).

What a Peach Object Must Do During a Tick

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

1. Peach must check to see if she is currently alive. If not, then Peach's *doSomething()* method must return immediately – none of the following steps should be performed.
2. Peach must check if she is currently invincible (Star Power), and if so, decrement the number of remaining game ticks before she loses this invincibility power. If this tick count reaches zero, Peach must set her invincibility status to off.
3. Peach must check if she is currently temporarily invincible, and if so, decrement the number of remaining game ticks before she loses temporary invincibility. If this tick count reaches zero, Peach must set her temporary invincibility status to false. (Peach gains temporary invincibility if she overlaps with an enemy while she has Jump Power or Fire Power.)
4. Peach must check if she is currently in "recharge" mode before she can fire again. If the number of time_to_recharge_before_next_fire ticks is greater than zero, she must decrement this tick count by one. If the tick count reaches zero, then Peach may again shoot a fireball (if she has Shoot Power).

5. Peach must check to see if she currently overlaps with any other game object[1] (e.g., an enemy, a fireball, a flag, etc.) and if so, she must "bonk" the other object. What happens when you bonk another object? It depends on what's being bonked; each class should have its own unique bonk() method that reacts appropriately.
6. If Peach had previously initiated a jump and her remaining_jump_distance is > 0, then she will try to move upward by four pixels during the current tick:
    a. Peach will calculate her target x,y position first (in this case, four pixels greater than her current y position)
    b. Peach will check to see if there is an object that blocks movement at this destination position (before moving there). If so:
        i. Peach will bonk the target object that is blocking her way (e.g., cause a bonk() method in the target object to be called)
        ii. Peach will abort trying to move to the destination square since it is blocked
        iii. Peach will update her remaining_jump_distance to zero such that the jump will be aborted and she will no longer try to move upward on the next tick.
    c. Otherwise if there is not a blocking object above Peach:
        i. Peach will use the *moveTo()* function from *GraphObject* to update her location 4 pixels upward.
        ii. Peach must decrement her remaining_jump_distance by 1 to indicate that she is now one step closer to reaching the top of her jump.
7. Otherwise, if Peach was not actively jumping during the current tick, then she must check to see if she is falling:
    a. Peach must check if there is an object that blocks movement between 0 and 3 (inclusive) pixels directly below her.
    b. If not, then Peach must update her y position by -4 pixels (so she is falling downward) using *GraphObject*'s *moveTo()* function.
8. Next, Peach must check to see if the player pressed a keystroke using the getKey() function.
9. If the user pressed a key:
    a. If the pressed key was KEY_PRESS_LEFT then Peach must:
        i. Set her direction to 180 degrees
        ii. Peach will calculate a target x,y position first (4 pixels less than her current x position)
        iii. Peach will check to see if there is an object that blocks movement at this destination position (before moving there). If so:
            1. Peach will bonk the target object that is blocking her way (e.g., cause a bonk() method in the target object to be called)
            2. Peach will abort trying to move to the destination square since it is blocked

[1] Hint: Since your StudentWorld class holds all of your game's actors, it makes sense to add a public method to it that can be used by all of your actors to determine if a given slot is occupied by a blocking object. For instance: if (studentWorldPtr->isBlockingObjectAt(x,y)) { … }

iv. Otherwise, Peach will update her location 4 pixels leftward.

b. If the pressed key was KEY_PRESS_RIGHT then Peach must:
   i. Set her direction to 0 degrees
   ii. Peach will calculate a target x,y position first (4 pixels greater than her current x position)
   iii. Peach will check to see if there is an object that blocks movement at this destination position (before moving there). If so:
      1. Peach will bonk the target object that is blocking her way (e.g., cause a bonk() method in the target object to be called)
      2. Peach will abort trying to move to the destination square since it is blocked
   iv. Otherwise, Peach will update her location 4 pixels rightward.
c. If the pressed key was KEY_PRESS_UP then Peach must:
   i. Check to see if there is an object that would block movement one pixel below her. (Such an object gives her support to jump; she doesn't actually move downward.) If so:
      1. Peach must set her remaining_jump_distance to the appropriate value:
         a. If Peach does NOT have Jump Power, then set remaining_jump_distance to 8.
         b. If Peach DOES have Jump Power, then set remaining_jump_distance to 12.
      2. Peach must play the sound SOUND_PLAYER_JUMP using the playSound() method in the GameWorld class.
d. If the pressed key was the KEY_PRESS_SPACE bar key, then:
   i. If Peach doesn't have Shoot Power, then do nothing
   ii. Otherwise, if the time_to_recharge_before_next_fire is greater than zero, then do nothing.
   iii. Otherwise:
      1. Play the sound SOUND_PLAYER_FIRE using the playSound() method in the GameWorld class.
      2. Set time_to_recharge_before_next_fire to 8, meaning that Peach may not fire again for another 8 game ticks
      3. Determine the x,y position directly in front of Peach that is 4 pixels away in the direction she's facing.
      4. Introduce a new fireball object at this location into your StudentWorld. The fireball must have its direction set to the same direction that Peach was facing when she fired.

<u>What Peach Must Do In Other Circumstances</u>

Hint: The following actions can be performed on Peach by other actors in the game (e.g., a fireball fired by a piranha might damage Peach). Each of the items below might be

implemented by a separate method inside the Peach class (or perhaps one of her super-classes).

- Peach does NOT block other actors from moving onto the same spot as her, and will indicate this if asked by a method call.
- Peach CAN be bonked (e.g., by a goomba, koopa, or piranha that overlaps with her and bonks her) and this will cause different results, depending on whether or not Peach has one or more special Powers at the time. Here's what you must do if another object attempts to bonk() Peach:
    - If Peach either has Star Power (invincibility) or temporary invincibility, then do nothing.
    - Otherwise:
        - Decrement Peach's hit points by one
        - Set Peach's temporary invincibility to 10 ticks
        - If Peach had Shoot Power, turn it off
        - If Peach had Jump Power, turn it off
        - If Peach has at least one hit point left, she must play the SOUND_PLAYER_HURT sound effect using GameWorld's playSound() method.
        - If Peach hit points reach zero (or below), the Peach object must:
            - Immediately set its status set to not-alive.
            - (The *StudentWorld* class should later detect Peach's death and the current level ends)
- Peach can be damaged by another object (e.g., a piranha-fired fireball that overlaps with her), and will indicate this if asked by a method call.
- When she is damaged, Peach must do the same thing as if she was bonked.
- Peach can be given Star Power (made invincible for a specified number of ticks).
- Peach can be given Jump Power (able to jump 50% higher until Peach is bonked by an enemy or piranha-fired fireball).
- Peach can be given Shoot Power (able to shoot fireballs until Peach is bonked by an enemy or piranha-fired fireball).
- Peach can have her hit points increased.

<u>Getting Input From the User</u>

Since Super Peach Sisters is a *real-time* game, you can't use the typical *getline* or *cin* approach to get a user's key press within Peach's *doSomething()* method— that would stop your program and wait for the user to type something and then hit the Enter key. This would make the game awkward to play, requiring the user to hit a directional key then hit Enter, then hit a directional key, then hit Enter, etc. Instead of this approach, you will use a function called *getKey()* that we provide in our *GameWorld* class (from which

your *StudentWorld* class is derived) to get input from the player[2]. 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 Peach::doSomething()
{
    ...
    int ch;
    if (getWorld()->getKey(ch))
    {
        // user hit a key during this tick!
        switch (ch)
        {
          case KEY_PRESS_LEFT:
            ... move Peach left ...;
            break;
          case KEY_PRESS_RIGHT:
            ... move Peach right...;
            break;
          case KEY_PRESS_SPACE:
            ... add fireball in front of Peach...;
            break;
          // etc…
        }
    }
    ...
}
```

Block

Blocks don't really do much… unless they're bonked, that is.

What a Block Must Do When It Is Created

When it is first created:

1. A Block object must have an image ID of IID_BLOCK.
2. A Block object has a starting (x,y) position based on the current level data file. Your *StudentWorld* object can pass in that position when constructing this object.
3. A Block object has a direction of 0 degrees.
4. A Block object has a graphical depth of 2.
5. A Block object has a default size of 1.
6. A Block object may be configured to release a specific type of goodie when it is first bonked. Options include: release no goodie at all, release a Star goodie, release a Flower goodie, or release a Mushroom goodie.
7. A Block starts out in a mode where it has not yet released a goodie.
8. A Block object starts out in the alive state.

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

A Block 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 Block must do nothing :).

What a Block Must Do In Other Circumstances

- A Block DOES block/prevent other actors from moving onto the same spot as it, and will indicate this if asked by a method call.
- A Block can be bonked (e.g., by Peach bonking her head into it). Here's what a block must do if another object attempts to bonk() it:
    - If the Block does not hold a goodie or has already released its goodie, then:
        - Play the SOUND_PLAYER_BONK sound using GameWorld's playSound() method
        - Do nothing else
    - If the Block does hold a goodie and the Block has not yet been bonked during the current level, then:
        - Play the SOUND_POWERUP_APPEARS sound using GameWorld's playSound() method
        - Introduce a goodie object of the appropriate type (Flower, Star or Mushroom) exactly 8 pixels above the block that was bonked (the same x coordinate as the block, but y+8 from the block)
        - Update the block's state to indicate that is has been bonked during the current level (so it doesn't produce another goodie if bonked again)
- A Block object is not damageable and will indicate this if asked by a method call.
- Attempts to damage a Block will do nothing.

Pipe

Pipe don't really do much… at all! Hint: Pipes are really similar to Blocks!

What a Pipe Must Do When It Is Created

When it is first created:

1. A Pipe object must have an image ID of IID_PIPE
2. A Pipe object has a starting (x,y) position based on the current level data file.
3. A Pipe object has a direction of 0 degrees.

4. A Pipe object has a graphical depth of 2.
5. A Pipe object has a default size of 1.
6. A Pipe object starts out in the alive state.

What a Pipe Must Do During a Tick

A Pipe 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 Pipe must do nothing :).

What a Pipe Must Do In Other Circumstances

- A Pipe DOES block/prevent other actors from moving onto the same spot as it, and will indicate this if asked by a method call.
- A Pipe can be bonked, but nothing happens if it is bonked.
- A Pipe object is not damageable and will indicate this if asked by a method call.
- Attempts to damage a Pipe will do nothing.

Flags

A Flag is a portal to the next level of the game. If a Flag object detects that Peach overlaps with it, then the Flag object will indicate this to the StudentWorld such that the StudentWorld can advance to the next level of the game.

What a Flag Must Do When It Is Created

When it is first created:

1. A Flag object must have an image ID of IID_FLAG.
2. A Flag object has a starting (x,y) position based on the current level data file.
3. A Flag object has a direction of 0 degrees.
4. A Flag object has a graphical depth of 1.
5. A Flag object has a default size of 1.
6. A Flag object starts out in the alive state.

What a Flag Must Do During a Tick

A Flag 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 Flag must do the following:

1. The Flag must check to see if it is alive, and if not, it must do nothing.
2. Otherwise, the Flag must see if it currently overlaps with Peach. If so:
    a. It will increase the player's score by 1000 points

b. It will immediately set its state to not-alive
c. It will inform the StudentWorld object that the current level was completed (so the StudentWorld can advance to the next level)

What a Flag Must Do In Other Circumstances

- A Flag does NOT block/prevent other actors from moving onto the same spot as it, and will indicate this if asked by a method call.
- A Flag does nothing when bonked.
- A Flag object is not damageable and will indicate this if asked by a method call.
- Attempts to damage a flag will do nothing.

Mario

Mario must be rescued (Peach must overlap with him) for her to win the game. If a Mario object detects that Peach overlaps with it, then the Mario object will indicate this to the StudentWorld such that the StudentWorld can tell the player that they have won the game. Hint: Mario objects are very similar to Flag objects!

What Mario Must Do When It Is Created

When it is first created:

1. A Mario object must have an image ID of IID_MARIO
2. A Mario object has a starting (x,y) position based on the current level data file.
3. A Mario object has a direction of 0 degrees.
4. A Mario object has a graphical depth of 1.
5. A Mario object has a default size of 1.
6. A Mario object starts out in the alive state.

What Mario Must Do During a Tick

Mario 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 Mario object must do the following:

1. Mario must check to see if it is alive, and if not, it must do nothing.
2. Otherwise, the Mario object must see if it currently overlaps with Peach. If so:
    d. It will increase the player's score by 1000 points
    e. It will immediately set its state to not-alive
    f. It will inform the StudentWorld object that the player has won the game (so the StudentWorld can inform the player and end the game)

What Mario Must Do In Other Circumstances

- A Mario object does NOT block/prevent other actors from moving onto the same spot as it, will indicate this if asked by a method call.
- A Mario object does nothing when bonked.
- A Mario object is not damageable and will indicate this if asked by a method call.
- Attempts to damage Mario will do nothing.

Flower

Flower goodies are responsible for detecting when they overlap with Peach, and when they do, giving her the Shoot Power, setting her hit points to 2, and then disappearing from the game.

What a Flower Must Do When It Is Created

When it is first created:

1. A Flower object must have an image ID of IID_FLOWER
2. A Flower object has its (x, y) position specified based on where it was created by the Block that generated it.
3. A Flower object has a direction of 0 degrees.
4. A Flower object has a graphical depth of 1.
5. A Flower object has a default size of 1.
6. A Flower object starts out in the alive state.

What a Flower Must Do During a Tick

A Flower 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 Flower object must do the following:

1. The Flower object must see if it currently overlaps with Peach. If so:
    g. It will increase the player's score by 50 points
    h. It will inform the Peach object that it now has the Shoot Power
    i. It will set Peach's hit points to 2
    j. It will immediately set its state to not-alive
    k. It will play a sound of SOUND_PLAYER_POWERUP using GameWorld's playSound() method
    l. It will do nothing else and immediately return
3. Otherwise, the Flower object must determine if there is an object just beneath it that would block it from falling two pixels downward. If there is no such blocking object beneath the Flower, it will:
    a. Use the moveTo() method to move downward 2 pixels.

4. The Flower object will then determine what direction it is facing (0 or 180 degrees) and try to move in that direction by 2 pixels:
    a. The Flower will calculate a target x,y position first (2 pixels greater or less than its current x position)
    b. The Flower will check to see if there is an object that would block movement to this destination position. If so:
        i. The Flower will reverse its direction (from 0 to 180, or vice versa)
        ii. The Flower will do nothing else and immediately return
    c. Otherwise, the Flower will update its location 2 pixels leftward or rightward depending on the direction it's facing.

What a Flower Must Do In Other Circumstances

- A Flower object does NOT block/prevent other actors from moving onto the same spot as it, and will indicate this if asked by a method call.
- A Flower object does nothing when bonked.
- A Flower object is not damageable and will indicate this if asked by a method call.
- If another object attempts to damage a Flower it has no effect.

Mushroom

Mushroom goodies are responsible for detecting when they overlap with Peach, and when they do, giving her the Jump Power, setting her hit points to 2, and then disappearing from the game.

What a Mushroom Must Do When It Is Created

When it is first created:

1. A Mushroom object must have an image ID of IID_MUSHROOM
2. A Mushroom object has its (x, y) position specified based on where it was created by the Block that generated it.
3. A Mushroom object has a direction of 0 degrees.
4. A Mushroom object has a graphical depth of 1.
5. A Mushroom object has a default size of 1.
6. A Mushroom object starts out in the alive state.

What a Mushroom Must Do During a Tick

A Mushroom 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 Mushroom object must do the following:

1. The Mushroom object must see if it currently overlaps with Peach. If so:

    d. It will increase the player's score by 75 points
    e. It will inform the Peach object that it now has the Jump Power
    f. It will set Peach's hit points to 2
    g. It will immediately set its state to not-alive
    h. It will play a sound of SOUND_PLAYER_POWERUP using GameWorld's playSound() method
    i. It will do nothing else and immediately return
2. Otherwise, the Mushroom object must determine if there is an object just beneath it that would block it from falling two pixels downward. If there is no such blocking object beneath the Mushroom, it will:
    a. Use the moveTo() method to move downward 2 pixels.
3. The Mushroom object will then determine what direction it is facing (0 or 180 degrees) and try to move in that direction by 2 pixels:
    a. The Mushroom will calculate a target x,y position first (2 pixels greater or less than its current x position)
    b. The Mushroom will check to see if there is an object that would block movement to this destination position. If so:
        i. The Mushroom will reverse its direction (from 0 to 180, or vice versa)
        ii. The Mushroom will do nothing else and immediately return
    c. Otherwise, the Mushroom will update its location 2 pixels leftward or rightward depending on the direction it's facing.

What a Mushroom Must Do In Other Circumstances

- A Mushroom object does NOT block/prevent other actors from moving onto the same spot as it, and will indicate this if asked by a method call.
- A Mushroom object does nothing when bonked.
- A Mushroom object is not damageable and will indicate this if asked by a method call.
- If another object attempts to damage a Mushroom it has no effect.

Star

Star goodies are responsible for detecting when they overlap with Peach, and when they do, giving her the Star Power and then disappearing from the game.

What a Star Must Do When It Is Created

When it is first created:

1. A Star object must have an image ID of IID_STAR
2. A Star object has its (x, y) position specified based on where it was created by the Block that generated it.
3. A Star object has a direction of 0 degrees.

4. A Star object has a graphical depth of 1.
5. A Star object has a default size of 1.
6. A Star object starts out in the alive state.

What a Star Must Do During a Tick

A Star 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 Star object must do the following:

1. The Star object must see if it currently overlaps with Peach. If so:
    d. It will increase the player's score by 100 points
    e. It will inform the Peach object that it now has the Star Power for 150 game ticks.
    f. It will immediately set its state to not-alive
    g. It will play a sound of SOUND_PLAYER_POWERUP using GameWorld's playSound() method
    h. It will do nothing else and immediately return
2. Otherwise, the Star object must determine if there is an object just beneath it that would block it from falling two pixels downward. If there is no such blocking object beneath the Star, it will:
    a. Use the moveTo() method to move downward 2 pixels.
3. The Star object will then determine what direction it is facing (0 or 180 degrees) and try to move in that direction by 2 pixels:
    a. The Star will calculate a target x,y position first (2 pixels greater or less than its current x position)
    b. The Star will check to see if there is an object that would block movement to this destination position. If so:
        i. The Star will reverse its direction (from 0 to 180, or vice versa)
        ii. The Star will do nothing else and immediately return
    c. Otherwise, the Star will update its location 2 pixels leftward or rightward depending on the direction it's facing.

What a Star Must Do In Other Circumstances

- A Star object does NOT block/prevent other actors from moving onto the same spot as it, and will indicate this if asked by a method call.
- A Star object does nothing when bonked.
- A Star object is not damageable and will indicate this if asked by a method call.
- If another object attempts to damage a Star it has no effect.

Piranha Fireball

A Piranha Fireball is responsible for detecting when it overlaps with Peach, and when it does so, damaging her and then disappearing from the game.

What a Piranha Fireball Must Do When It Is Created

When it is first created:

1. A Piranha Fireball object must have an image ID of IID_PIRANHA_FIRE
2. A Piranha Fireball object has its (x, y) position specified based on where it was created by the Piranha that generated it.
3. A Piranha Fireball object has a direction that is specified by the piranha that introduces the fireball into the game.
4. A Piranha Fireball object has a graphical depth of 1.
5. A Piranha Fireball object has a default size of 1.
6. A Piranha Fireball object starts out in the alive state.

What a Piranha Fireball Must Do During a Tick

A Piranha Fireball 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 Piranha Fireball object must do the following:

1. The Piranha Fireball object must see if it currently overlaps with Peach. If so:
    i. It will inform the Peach object that she has been damaged.
    j. It will immediately set its own state to not-alive
    k. It will do nothing else and immediately return
2. Otherwise, the Piranha Fireball object must determine if there is an object just beneath it that would block it from falling two pixels downward. If there is no such blocking object beneath the Piranha Fireball, it will:
    a. Use the moveTo() method to move downward 2 pixels.
3. The Piranha Fireball object will then determine what direction it is facing (0 or 180 degrees) and try to move in that direction by 2 pixels:
    a. The Piranha Fireball will calculate a target x,y position first (2 pixels greater or less than its current x position)
    b. The Piranha Fireball will check to see if there is an object that would block movement to this destination position. If so:
        i. The Piranha Fireball will immediately set its state to not-alive
        ii. The Piranha Fireball will do nothing else and immediately return
    c. Otherwise, the Piranha Fireball will update its location 2 pixels leftward or rightward depending on the direction it's facing.

What a Piranha Fireball Must Do In Other Circumstances

- A Piranha Fireball object does NOT block/prevent other actors from moving onto the same spot as it, and will indicate this if asked by a method call.
- A Piranha Fireball object does nothing when bonked.
- A Piranha Fireball object is not damageable and will indicate this if asked by a method call.
- If another object attempts to damage a Piranha Fireball it has no effect.

Peach Fireball

A Peach Fireball is responsible for detecting when it overlaps with an enemy (goomba, koopa, or piranha), and when it does so, damaging them and then disappearing from the game.

What a Peach Fireball Must Do When It Is Created

When it is first created:

1. A Peach Fireball object must have an image ID of IID_PEACH_FIRE
2. A Peach Fireball object has its (x, y) position specified based on where it was created by Peach.
3. A Peach Fireball object has a direction that is specified by Peach
4. A Peach Fireball object has a graphical depth of 1.
5. A Peach Fireball object has a default size of 1.
6. A Peach Fireball object starts out in the alive state.

What a Peach Fireball Must Do During a Tick

A Peach Fireball 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 Peach Fireball object must do the following:

1. The Peach Fireball object must see if it currently overlaps with a damageable object (other than Peach). If so:
    a. It will inform the object that it has been damaged.
    b. It will immediately set its own state to not-alive
    c. It will do nothing else and immediately return
2. Otherwise, the Peach Fireball object must determine if there is an object just beneath it that would block it from falling two pixels downward. If there is no such blocking object beneath the Peach Fireball, it will:
    a. Use the moveTo() method to move downward 2 pixels.
3. The Peach Fireball object will then determine what direction it is facing (0 or 180 degrees) and try to move in that direction by 2 pixels:
    a. The Peach Fireball will calculate a target x,y position first (2 pixels greater or less than its current x position)

b. The Peach Fireball will check to see if there is an object that would block movement to this destination position. If so:
    i. The Peach Fireball will immediately set its state to not-alive
    ii. The Peach Fireball will do nothing else and immediately return
c. Otherwise, the Peach Fireball will update its location 2 pixels leftward or rightward depending on what direction it's facing.

What a Peach Fireball Must Do In Other Circumstances

- A Peach Fireball object does NOT block/prevent other actors from moving onto the same spot as it, and will indicate this if asked by a method call.
- A Peach Fireball object does nothing when bonked.
- A Peach Fireball object is not damageable and will indicate this if asked by a method call.
- If another object attempts to damage a Peach Fireball it has no effect.

Shell

A Shell is responsible for detecting when it overlaps with an enemy (goomba, koopa, or piranha), and when it does so, damaging them and then disappearing from the game.

What a Shell Must Do When It Is Created

When it is first created:

1. A Shell object must have an image ID of IID_SHELL
2. A Shell object has its (x, y) position specified based on where the koopa that creates it died.
3. A Shell object has a direction that is specified by the koopa that creates it
4. A Shell object has a graphical depth of 1.
5. A Shell object has a default size of 1.
6. A Shell object starts out in the alive state.

What a Shell Must Do During a Tick

A Shell 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 Shell object must do the following:

1. The Shell object must see if it currently overlaps with a damageable object (other than Peach). If so:
    a. It will inform the object that it has been damaged.
    b. It will immediately set its own state to not-alive
    c. It will do nothing else and immediately return

2. Otherwise, the Shell object must determine if there is an object just beneath it that would block it from falling two pixels downward. If there is no such blocking object beneath the Shell, it will:
    a. Use the moveTo() method to move downward 2 pixels.
3. The Shell object will then determine what direction it is facing (0 or 180 degrees) and try to move in that direction by 2 pixels:
    a. The Shell will calculate a target x,y position first (2 pixels greater or less than its current x position)
    b. The Shell will check to see if there is an object that would block movement to this destination position. If so:
        i. The Shell will immediately set its state to not-alive
        ii. The Shell will do nothing else and immediately return
    c. Otherwise, the Shell will update its location 2 pixels leftward or rightward depending on the direction it's facing.

What a Shell Must Do In Other Circumstances

- A Shell object does NOT block/prevent other actors from moving onto the same spot as it, and will indicate this if asked by a method call.
- A Shell object does nothing when bonked.
- A Shell object is not damageable and will indicate this if asked by a method call.
- If another object attempts to damage a Shell it has no effect.

Goomba

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

What a Goomba Must Do When It Is Created

When it is first created:

1. A Goomba object must have an image ID of IID_GOOMBA.
2. A Goomba has a starting (x,y) position based on the current level data file.
3. A Goomba has a direction of either 0 or 180 degrees, chosen randomly.
4. A Goomba has a size of 1.
5. A Goomba has a depth of 0.
6. A Goomba starts out in the alive state.

What a Goomba Must Do During a Tick

Each time a Goomba is asked to do something (during a tick):

1. The Goomba 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 Goomba must check to see if it overlaps with Peach at its current location. If so:
    a. The Goomba will attempt to bonk[3] Peach
    b. The Goomba will immediately return
3. The Goomba will then determine if it can move 1 pixel in its current direction without running into an object that blocks movement. If it CANNOT move in this direction because there is an object blocking its way, it must switch to face the opposite direction
4. Otherwise the Goomba will determine if it can move 1 pixel in its current direction without stepping partly or fully off of the edge of its current platform of blocks and/or pipes. If it CANNOT move 1 pixel without extending partly or fully over empty space, it must switch to face the opposite direction
5. The Goomba will then determine what direction it is now facing (0 or 180 degrees) and try to move in that direction by 1 pixel:
    a. The Goomba will calculate a target x,y position first (1 pixel greater or less than its current x position)
    b. The Goomba will check to see if there is an object that would block movement to this destination position. If so:
        i. The Goomba will do nothing else and immediately return
    c. Otherwise, the Goomba will update its location 1 pixel leftward or rightward depending on the direction it's facing.

<u>What Goomba Must Do In Other Circumstances</u>

- Goomba does NOT block other actors from moving onto the same spot as it, and will indicate this if asked by a method call.
- Goomba can be bonked (e.g., by Peach when she's got Star Power). Here's what you must do if another object attempts to bonk Goomba:
    - If the bonker is not Peach, then ignore the bonk
    - Otherwise the bonker is Peach. If Peach has Star Power (invincibility), then:
        - Play the sound SOUND_PLAYER_KICK using GameWorld's playSound() method
        - Increase the player's score by 100 points
        - Set Goomba's state to not-alive (The *StudentWorld* class should later detect Goomba's death and remove it from the game at the end of the tick)
- Goomba can be damaged by another object (e.g., a fireball fired by Peach overlaps with her), and will indicate this if asked by a method call.
- When a Goomba is damaged, it must:
    - Increase the player's score by 100 points

[3] Note: Bonking is different than Damaging, and the two operations should be dealt with distinctly.

- Set its state to not-alive (The *StudentWorld* class should later detect Goomba's death and remove it from the game at the end of the tick)

Koopa

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

What a Koopa Must Do When It Is Created

When it is first created:

1. A Koopa object must have an image ID of IID_KOOPA.
2. A Koopa has a starting (x,y) position based on the current level data file.
3. A Koopa has a direction of either 0 or 180 degrees, chosen randomly.
4. A Koopa has a size of 1.
5. A Koopa has a depth of 0.
6. A Koopa starts out in the alive state.

What a Koopa Must Do During a Tick

Each time a Koopa is asked to do something (during a tick):

1. The Koopa 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 Koopa must check to see if it overlaps with Peach at its current location. If so:
    a. The Koopa will attempt to bonk Peach
    b. The Koopa will immediately return
3. The Koopa will then determine if it can move 1 pixel in its current direction without running into an object that blocks movement. If it CANNOT move in this direction because that is an object blocking its way, it must switch to face the opposite direction
4. Otherwise the Koopa will determine if it can move 1 pixel in its current direction without stepping partly or fully off of the edge of its current platform of blocks and/or pipes. If it CANNOT move 1 pixel without extending partly or fully over empty space, it must switch to face the opposite direction
5. The Koopa will then determine what direction it is now facing (0 or 180 degrees) and try to move in that direction by 1 pixel:
    a. The Koopa will calculate a target x,y position first (1 pixel greater or less than its current x position)
    b. The Koopa will check to see if there is an object that would block movement to this destination position. If so:
        i. The Koopa will do nothing else and immediately return

c. Otherwise, the Koopa will update its location 1 pixel leftward or rightward depending on the direction it's facing.

What Koopa Must Do In Other Circumstances

- Koopa does NOT block other actors from moving onto the same spot as it, and will indicate this if asked by a method call.
- Koopa can be bonked (e.g., by Peach when she's got Star Power). Here's what you must do if another object attempts to bonk Koopa:
    - If the bonker is not Peach, then ignore the bonk
    - Otherwise the bonker is Peach. If Peach has Star Power (invincibility), then:
        - Play the sound SOUND_PLAYER_KICK using GameWorld's playSound() method
        - Increase the player's score by 100 points
        - Set Koopa's state to not-alive (The *StudentWorld* class should later detect Koopa's death and remove it from the game at the end of the tick)
        - Introduce a new Shell object at the same location as Koopa, facing the same direction as Koopa.
- Koopa can be damaged by another object (e.g., a fireball fired by Peach overlaps with her), and will indicate this if asked by a method call.
- When a Koopa is damaged, it must:
    - Increase the player's score by 100 points
    - Set its state to not-alive (The *StudentWorld* class should later detect Koopa's death and remove it from the game at the end of the tick)
    - Introduce a new Shell object at the same location as Koopa, facing the same direction as Koopa.

Piranha

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

What a Piranha Must Do When It Is Created

When it is first created:

1. A Piranha object must have an image ID of IID_PIRANHA.
2. A Piranha has a starting (x,y) position based on the current level data file.
3. A Piranha has a direction of either 0 or 180 degrees, chosen randomly.
4. A Piranha has a size of 1.
5. A Piranha has a depth of 0.
6. A Piranha starts with a firing delay of 0 (meaning it can fire immediately if Peach gets too close).

7. A Piranha starts out in the alive state.

What a Piranha Must Do During a Tick

Each time a Piranha is asked to do something (during a tick):

1. The Piranha 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 Piranha must cycle its sprite image using *GraphObject*'s *increaseAnimationNumber*() method. This will cause Piranha's jaws to open and close even though Piranha doesn't move (which is what usually updates the sprite images).
3. The Piranha must check to see if it overlaps with Peach at its current location. If so:
    a. The Piranha will attempt to bonk Peach
    b. The Piranha will immediately return
4. The Piranha will then determine if Peach's y coordinate is within 1.5*SPRITE_HEIGHT of Piranha's y coordinate. If not, Piranha will immediately return and do nothing else
5. Otherwise the Piranha knows that Peach is on the same level.
6. Piranha will determine if Peach is to its left (or right), and if so, set its direction to 180 (or 0), so it is facing Peach.
7. Piranha will check if it has a firing delay, and if so:
    a. It will decrease the firing delay by one.
    b. It will immediately return.
8. If there is no firing delay, then Piranha may choose to fire:
    a. It will compute the distance between itself and Peach. If Peach's x coordinate is strictly less than 8*SPRITE_WIDTH pixels away from Piranha's x coordinate, then Piranha will:
        i. Add a new Piranha Fireball object to StudentWorld at its current x,y position, with an initial direction facing in the same direction that Piranha is facing
        ii. Play the sound SOUND_PIRANHA_FIRE using GameWorld's playSound() method.
        iii. Set its firing delay to 40, so Piranha won't fire at Peach for another 40 ticks

What Piranha Must Do In Other Circumstances

- Piranha does NOT block other actors from moving onto the same spot as it, and will indicate this if asked by a method call.
- Piranha can be bonked (e.g., by Peach when she's got Star Power). Here's what you must do if another object attempts to bonk() Piranha:
    - If the bonker is not Peach, then ignore the bonk
    - Otherwise the bonker is Peach. If Peach has Star Power (invincibility), then:

        - Play the sound SOUND_PLAYER_KICK using GameWorld's playSound() method
        - Increase the player's score by 100 points
        - Set Piranha's state to not-alive (The *StudentWorld* class should later detect Piranha's death and remove it from the game at the end of the tick)
- Piranha can be damaged by another object (e.g., a fireball fired by Peach overlaps with her), and will indicate this if asked by a method call.
- When a Piranha is damaged, it must:
    - Increase the player's score by 100 points
    - Set its state to not-alive (The *StudentWorld* class should later detect Piranha's death and remove it from the game at the end of the tick)

Object Oriented Programming Tips

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

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

Here we go!

1. **You MUST NEVER use the imageID (e.g., IID_GOOMBA, IID_PIRANHA, IID_FLAG, etc.) to determine the type of an object or store the imageID inside any of your objects as a member variable. Doing so will result in a score of ZERO for this project. You must also not use any equivalent, e.g., adding strings like "IID_KOOPA", enums, ints or Morse code to your classes.**

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

Don't do this:

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

```
}
```

Do this instead:

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

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

Don't do this:

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

Do this instead:

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

4. If two related subclasses (e.g., SmellyRobot and GoofyRobot) each directly define a 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 doSomething()
    {
      doCommonThingA();
      passStinkyGas();
      pickNose();
      doCommonThingB();
    }
};

class ShinyRobot: public Robot
{
      …
  public:
    virtual void doSomething()
    {
      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, ONLINE STORAGE, OR A PRIVATE GITHUB REOSITORY 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 SuperPeachSisters-skeleton-windows.zip archive into a folder on your hard drive. Double-click on SuperPeachSisters.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 SuperPeachSisters-skeleton-mac.zip archive into a folder on your hard drive. Double-click on our provided SuperPeachSisters.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/SuperPeachSisters/DerivedData/SuperPeachSisters/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 Super Peach Sisters game that implements maybe 15% of the overall project. You must program:

1. A class that can serve as the base class for all of your game's actors (e.g., a base class that accommodates Peach, all types of enemies, goodies, projectiles, etc.):
    i. It must have a constructor that initializes the object appropriately.
    ii. It must be derived from our *GraphObject* class.
    iii. It must have a member function named *doSomething()* that can be called to cause the actor to do something.
    iv. You may add other public/private member functions and private data members to this base class, as you see fit.
2. A Block class, derived in some way from the base class described in 1 above:

i. It must implement the specifications described in the Block section above, except that you need not implement the functionality of holding or releasing goodies; plain blocks suffice.
ii. You may add any public/private member functions and private data members to your Block class as you see fit, so long as you use good object oriented programming style (e.g., you **must NOT** duplicate non-trivial functionality across classes).

3. A limited version of your Peach 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 Peach – see Peach section for more details on how to initialize Peach.
    ii. It must have a limited version of a *doSomething()* method that lets the user move Peach left and right by hitting a directional key. If the player hits a directional key during the current tick, your code must update Peach's direction and position appropriately if there is no Block preventing Peach from moving in the specified direction (see the spec above). All this *doSomething()* method has to do is properly adjust Peach's direction and (x) coordinates using the *GraphObject* class's *setDirection()* and *moveTo()* method, and our graphics system will automatically animate her movement! You do NOT need to address cases like where Peach falls vertically if she steps off a block, meaning that for Part 1 Peach may move left/right even if there are not supporting blocks/pipes underneath her (and thus float in the air).
    iii. You may add other public/private member functions and private data members to your Peach class as you see fit, so long as you use good object oriented programming style (e.g., you must not duplicate functionality across classes). But you need not implement firing, jumping, falling, etc. for this part of the project.
4. A limited version of the *StudentWorld* class.
    i. Add any private data members to this class required to keep track of all game objects/actors. Right now all of those game objects will just be Blocks and Peach, but eventually you'll need to include all of your actors.
    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 destructed.
    iv. Implement the *init()* method in this class. It must load up the first level data file using our provided Level class, and then create Block objects and a Peach object and insert them into its data structures. The positions of each of the Blocks and Peach must be set based on the contents of the Level data file. Your *init()* method may ignore any other objects like koopas, goombas, piranhas, pipes, flags, Mario, etc. - it must only deal with Peach and Blocks.

v. Implement the *move()* method in your *StudentWorld* class. During each tick, it must ask Peach and other actors (just Blocks for now) to do something. Your *move()* method need not check to see if Peach has died or not; you may assume for Part #1 that Peach cannot die. Your *move()* method does not have to deal with any actors other than Peach and the Blocks.
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 Blocks and Peach). 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 the first level with Peach in her proper starting position, and blocks in their proper positions as specified in the first level data file. If your classes work properly, you should be able to move Peach left and right until she runs into a block.

Your Part #1 solution may actually do more than what is specified above; for example, if you are making good progress, try to add a Flag class 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 pipes, koopas, goombas, piranhas, goodies, flags, Mario, 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 Peach, Block, 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, Peach, and Block 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 the Super Peach Sisters 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 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 Flower class." or "My koopa doesn't work correctly yet so I treat it like a goomba right now."
3. A list of other design decisions and assumptions you made; e.g., "It was not specified what to do in situation X, so this is what I decided to do."

FAQ

Q: Why does my video game run slower/faster than yours?
A: It could be your choice of data structures and algorithms, graphics cards, etc. It's OK if your game is faster or a little slower than ours. If your game runs MUCH slower, then you probably have a Big-O problem and could choose better data structures. If your game runs faster and you'd like to slow down gameplay, update line 52 in GameController.cpp with a larger value, like 20 or 50:

```
static const int MS_PER_FRAME = 5;
```

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 if it won't even build!

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

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

GOOD LUCK!