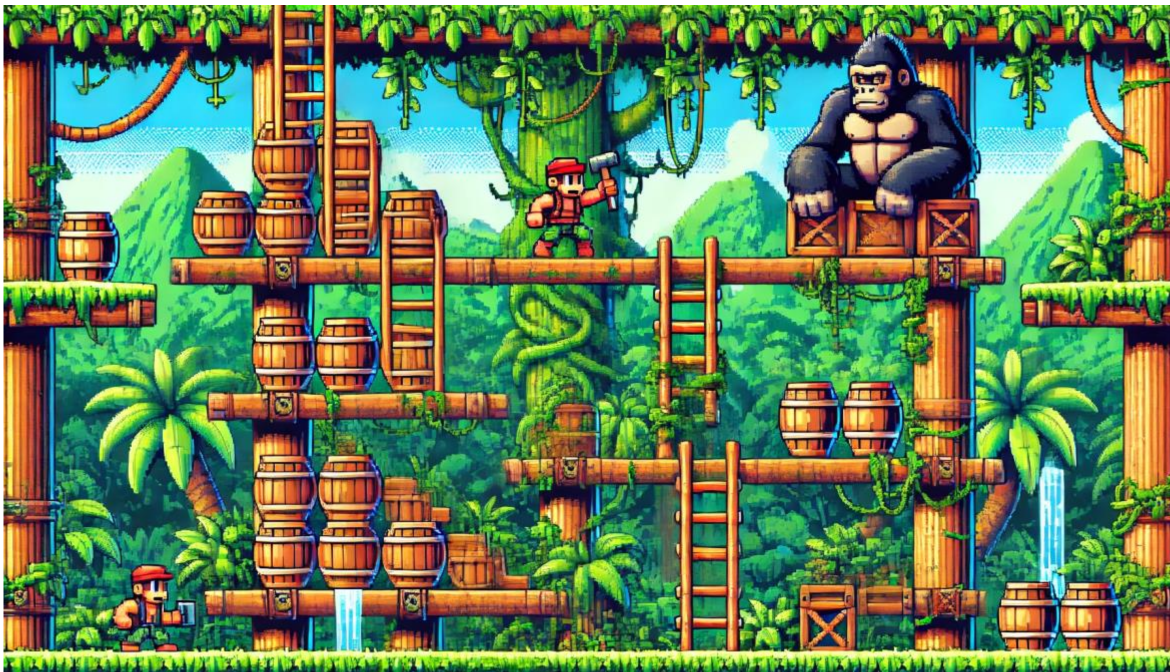
Project 3

Wonky Kong

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 23
Part 2: 11 PM Sunday, March 2

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!

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

Table of Contents 2
Introduction 3
Game Details 4
How Does a Video Game Work? 6
  Phases of the Game 7
    Initialization 7
    Game Play 8
    Cleanup 8
    Putting it All Together 8
  Actors and Their Behaviors 9
What Do You Have to Do? 10
  You Have to Create the StudentWorld Class 11
    `init()` Details 14
    `move()` Details 15
    `cleanUp()` Details 19
    The Level Class and Level Data File 19
  You Have to Create the Classes for All Actors 22
    The Player 26
    Floor 29
    Ladder 29
    Burp 30
    Bonfire 30
    Extra Life Goodie 31
    Garlic Goodie 32
    Fireball 33
    Koopa 34
    Barrel 35
    Kong 37
Object Oriented Programming Best Practices 38
Don’t know how or where to start? Read this! 44
Building the Game 45
  For Windows 45
  For Mac OS X 45
What to Turn In 46
  Part #1 (20%) 46
    What to Turn In For Part #1 48
  Part #2 (80%) 48
    What to Turn In For Part #2 48
FAQ 49

Introduction

NachenGames has discovered that Small.ai intends to use ChatGPT to create a retro version of the famous ‘80s Donkey Kong video game. Being super competitive, the evil NachenGames CEO Yerac Nachengreb wants to beat Small.ai to the punch by releasing his own human-coded version called Wonky Kong. You've been hired by NachenGames for an unpaid internship to implement Wonky Kong so that it plays and behaves exactly like the early prototype Yerac stole from Small.ai.

In Wonky Kong, the Player’s objective is to navigate a series of platforms and ladders, all while carefully dodging falling barrels thrown by Kong and avoiding dangerous Fireballs and Koopas. The Player can jump, climb up and down ladders, and blast enemies by blowing noxious burp gas on them. Upon reaching Kong, the Player progresses to the next level. The game continues until the Player runs out of lives or completes all levels.

Below is a screenshot of Wonky Kong:

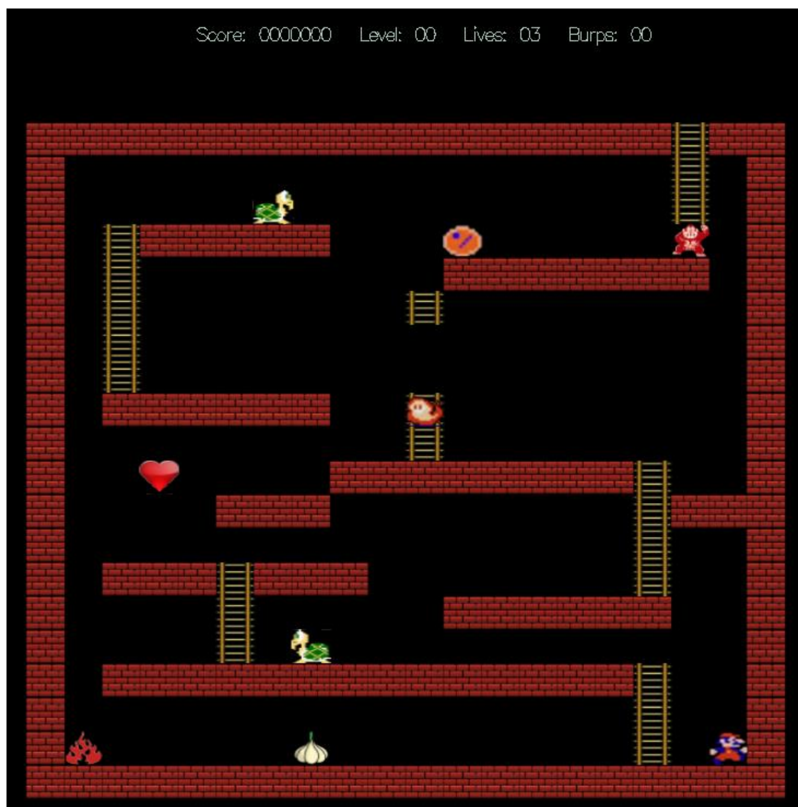


Figure #1: A screenshot of Wonky Kong gameplay. You can see the Player in the lower-right, platforms, ladders, a bonfire in the lower-left, a garlic goodie, an extra-life goodie, a fireball, a barrel, two koopas, and Kong himself looming on a ledge in the upper-right.

Game Details

In Wonky Kong, each level is a 20×20 grid consisting of the following:

1. The Player (the plumber wearing overalls)
2. Floors/walls that you can walk on
3. Ladders that can be climbed up and down
4. Barrels that roll over platforms and fall off edges in classic Donkey Kong style
5. Fireballs that move back and forth on platforms and can climb up or down ladders
6. Koopas (turtle-like creatures) that move back and forth on platforms
7. Kong, who throws barrels, and flees up off the game screen once the Player gets too close
8. Extra-life Goodies (shown as a heart) that give the Player an additional life
9. Garlic Goodies (shown as a garlic bulb) that give the Player bad breath and the ability to blast 5 burps at enemies
10. Bonfires that burn up Barrels and the Player

The Player, who starts with 3 lives and zero Burps, has a goal during play to reach Kong by navigating the various platforms and ladders. To do so, the Player can move left or right (using arrow keys or A and D) and climb ladders (using arrow keys or W and S). Gravity takes effect if there is no floor or ladder underfoot, causing the Player to fall until hitting a floor or ladder. The Player can also jump (by pressing the Spacebar), but only under certain conditions (e.g., while standing on top of a floor or ladder, or while climbing a ladder). If the Player eats a Garlic Goodie they get 5 Burps that they can expel one at a time (by pressing the Tab key) to attack enemies. The Player can collect goodies in order to gain burp power or an extra life and aid them in their quest.

If the Player comes into contact with a Barrel, Fireball, or Bonfire they lose a life, and assuming they have additional lives left, must restart the current level from scratch. If the Player comes into contact with a Koopa, they are temporarily frozen in place, making them vulnerable to attack.

If and when the Player reaches Kong, Kong will flee off the screen and the Player is awarded 1000 bonus points. The Player then advances to the next level. If the Player completes the final level, the game ends in victory; if the Player runs out of lives, the game is over.

The following is a list of objects that may appear on any given level, along with their basic interactions.

Floor

- A standard platform tile.
- The Player and the other actors may safely stand or move on it.
- Blocks movement if trying to move through it from the side.

Ladder

- The Player and Fireballs can climb upward or downward on a ladder if the next square in that direction is not blocked.
- The Player and other actors may move on a ladder and may stay on it, except that if a Barrel is on a ladder with no floor underneath it, it will fall.
- The Player may jump onto a ladder if one is within reach.

Bonfire

- A stationary hazard that incinerates Players and Barrels.
- If the Player steps onto the same square as a Bonfire, it immediately causes the Player to lose a life.
- If a Barrel rolls onto the same square as a Bonfire, it immediately destroys the Barrel.

Player

- The Player moves around trying to avoid hazards (Bonfires, Barrels, Fireballs, Koopas, etc.).
- The Player has a limited number of burps (initially zero), which can be unleashed to destroy or damage an adjacent enemy in front of them.
- The Player may be “frozen” (by a Koopa), which prevents burping, movement and jumping for a limited number of ticks.
- The Player can jump over small obstacles. Jumps follow a specific short sequence, and if the space is obstructed, the jump aborts.

Kong

- Wonky Kong’s namesake character, perched somewhere in the level.
- Periodically throws Barrels in the direction it faces.
- Kong “freaks out” when the Player gets close enough, triggering a flee sequence: Kong will escape upward out of the level. Once Kong fully flees, the level is considered complete.
- Points: Once Kong flees the Player is awarded 1000 points.

Barrel

- A rolling Barrel that Kong hurls to stop the Player.
- Barrels continue rolling horizontally unless they reach a square where there is no floor underneath, at which point they drop down.
- Once a Barrel has fallen at least one step, it reverses direction so when it lands it rolls the other way.
- Points: Destroying a Barrel with a Burp awards the Player 100 points.

Fireball

- An enemy that wanders back and forth on platforms and that can occasionally climb up and down ladders.
- When blasted with a Burp, there is a 1 in 3 chance that a Fireball drops a Garlic Goodie.
- Points: Destroying a Fireball with a Burp awards the Player 100 points.

Koopa

- An enemy that wanders back and forth on platforms.
- Koopa will “freeze” the Player upon contact, stopping the Player's movement for a short duration.
- After using its freezing attack, Koopa must wait through a cooldown before it can freeze the Player again.
- When blasted with a Burp, there is a 1 in 3 chance that the Koopa drops an Extra Life Goodie.
- Points: Destroying a Koopa with a Burp awards the Player 100 points.

Extra Life Goodie

- When picked up, grants the Player an additional life.
- Upon being collected, the Goodie disappears from the level.
- Points: Picking up an Extra Life Goodie awards the Player 50 points.

Garlic Goodie

- When picked up, adds 5 Burps to the Player’s inventory.
- Upon being collected, the Goodie disappears from the level.
- Points: Picking up a Garlic Goodie awards the Player 25 points.

Burp

- A stationary cloud of burp gas emitted by a player.
- Blasts a Barrel, Fireball, or Koopa that moves to its square.
- Disappears after a certain number of ticks.

How Does a Video Game Work?

So how does a video game like Wonky Kong work? Fundamentally, a video game is composed of many objects; in Wonky Kong, these objects (actors) include:

- The Player (the user-controlled avatar)
- Kong (the big, boss-like character hurling barrels)
- Barrels
- Fireballs
- Koopas
- Floors
- Ladders
- Bonfires
- Extra Life Goodies
- Garlic Goodies

Just as in the classic Donkey Kong arcade game, each actor in Wonky Kong has its own x,y location in the game world, its own internal state (e.g., the direction the actor is currently

moving, how many burps the Player has left), and its own special behaviors (e.g., Barrels roll until they fall off a ledge, then switch directions, Fireballs climb up or down a ladder). The Player's behavior is, of course, controlled by human input (the arrow keys, spacebar, tab, etc.), while other actors have autonomous algorithms for how they move or interact with the Player.

Once a Wonky Kong level begins, gameplay is divided into ticks. Each tick is a single unit of time (for example, 1/20 of a second). In a given tick:

1. The game asks each live actor to do something by calling the actor's doSomething() method.
2. During doSomething(), the actor may move, emit a burp, throw a barrel, cause the Player to lose a life, freeze the Player, fall, climb up or down, add a new actor to the game (e.g., when Kong throws a new Barrel) or get destroyed.
3. At the end of the tick, dead actors are removed from the game, and the remaining actors are then animated on the screen in their new positions.

Typically, the behavior exhibited by an actor 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 person playing the game. For example, a Fireball moves just one square left/right/up/down, rather than moving two or more squares; a Fireball moving, say, 5 squares in a single tick would confuse a human, because people 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 game framework that we provide animates the actors onto the screen in their new configuration. So, if a Fireball changed its location from 10,5 to 11,5 (moved one square right), then our game framework would erase the graphic of the Fireball from location 10,5 on the screen and draw the Fireball's graphic at 11,5 instead. Since this process (asking actors to do something, then animating them to the screen) happens 20 times per second, the user will see somewhat smooth animation. This process repeats every tick.

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

Phases of the Game

Initialization

- The level's details are loaded from a level file (e.g., level00.txt) that specifies where floors, ladders, enemies (Kong, Koopas, Fireballs, etc.), goodies, and the Player start out.
- This data is used to decide what actor objects to create and their starting locations and directions in the current level.

- All of these actors are allocated and added to your data structures to prepare for the level to be played.

Game Play

- The game enters a loop where, during each iteration, you ask all of your actors to perform some action.
- During each tick, an actor might:
    - Move in the direction it's facing (if no obstacle prevents movement).
    - Climb up or down a ladder.
    - Jump.
    - Switch directions.
    - Fall if there is nothing to support it.
    - Attack or interact with another actor (e.g., freeze the Player, cause the Player to lose a life, give the player an extra life).
    - Add a new actor to the world (e.g., Kong hurling a new Barrel).
    - Die (and thus get removed) under certain conditions (e.g., when a Barrel rolls into a Bonfire and is destroyed, or an enemy is blasted by a Burp).

Cleanup

- If the Player loses a life or the level finishes (e.g., Kong flees off the top of the screen), we delete all live actors.
- If the Player still has lives left, we begin the next iteration of initialization for either the same level or the next level.
- If the Player is out of lives, the game ends.

Putting it All Together

This is what the overall video game flow looks like:

```
while (the Player still has lives left)
{
    Prompt the user to press a key to start the level

    // For you to implement
    Initialize the game world with the current level

    while (the Player's still alive and has not finished the level)
    {
        // Each pass is a tick (e.g., 1/20th of a second).

        // For you to implement:
```

```
        Ask all actors to doSomething()
        Remove any dead actors from the world

        // Provided for you:
        Animate actors to the screen
        Pause for a short time (e.g., 50 ms)
    }

    // For you to implement
    Clean up all actors from this level

    If the Player still has lives remaining and the level is finished:
        try to load the next level
}

Tell the user the game is over
```

Actors and Their Behaviors

In Wonky Kong, each game object is an actor, and each actor implements a doSomething() method to define its unique behavior. You will maintain a collection (e.g., std::vector or std::list) of all active actors in the world. Below is a (simplified) look at what some of the actors do, without revealing the code. You will implement your own versions of these.

For instance, here's what a Fireball's doSomething() might look like in pseudocode:

```
void Fireball:: doSomething() {
  If Fireball is not alive:
    Return from the function

  If there is a climbable object at (currentX, currentY):
    Generate a random number r between 1 and 3
    If Fireball is not currently climbing down AND
        (Fireball is climbing up OR r is 1):
      Move Fireball up by one unit
      Set climbing direction to up
      Return from the function

  If there is a climbable object at (currentX, currentY - 1):
    Generate a random number r between 1 and 3
    If Fireball is not currently climbing up AND
       (Fireball is climbing down OR r is 1):
```

```
      Move Fireball down by one unit
      Set climbing direction to down
      Return from the function

  // Perform standard horizontal movement
  ...
}
```

A sample doSomething() for the Player might look like this (pseudocode):

```
void Player::doSomething() {
  If Player is dead, return.
  If Player is currently jumping:
    Execute the current step of the jump and return.
  Check gravity/fall if there's no floor or ladder beneath.
  If Player is frozen:
    Decrement freeze count and return.
  Else, read user input:
    if left arrow pressed, attempt to move left
    if right arrow pressed, attempt to move right
    if up arrow pressed, attempt to climb up
    if down arrow pressed, attempt to climb/fall
    if space pressed, attempt to start a new jump
    if tab pressed, attempt to add a burp in front of the Player
}
```

What Do You Have to Do?

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

1. You must create a class called StudentWorld which is responsible for keeping track of your game world (including the platforms/floor layout) and all of the actors/objects (Kong, Koopa, Fireball, Barrel, Bonfire, Floor, Ladder, ExtraLifeGoodie, GarlicGoodie, Burp, the Player's avatar, etc.) that are in the game.
2. You must create a class to represent the Player in the game.
3. You must create classes for all other actors (e.g., Kong, Koopa, Fireball, Barrel, Bonfire, Floor, Ladder, ExtraLifeGoodie, GarlicGoodie, Burp, etc.), as well as any additional base classes (e.g., an Enemy base class if you find it convenient) that help you implement the game.

You Have to Create the StudentWorld Class

Your StudentWorld class is responsible for orchestrating virtually all game play – it keeps track of the game world (the layout and all of its inhabitants, such as Kong, Koopas, Fireballs, Barrels, Bonfires, Floors, Ladders, Goodies, Burps, and the Player, etc.). It is responsible for:

- Initializing the game world at the start of each level
- Asking all the actors to do something during each tick of the game
- Destroying actor(s) when they disappear (e.g., an enemy dies)
- Destroying all of the actors in the game world when the user loses a life or completes a level

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

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

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

When a new level starts (e.g., at the start of a game or when the Player dies and needs to replay the level, or when the Player completes a level and advances to the next level), 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 loading the current level's layout from a data file (we'll show you how below), and constructing a representation of the current level in your StudentWorld object, 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 loaded/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.

When the Player has finished the level loaded from, for example, level00.txt, our game framework will call your cleanUp() method to clean things up, and then call your init() method again to load the next level data file (e.g., level01.txt).

If there is no level data file with the next number, or if the level just completed is level 99, the init() method must return GWSTATUS_PLAYER_WON. If the next level file exists but is not in the proper format for a level data file, the init() method must return GWSTATUS_LEVEL_ERROR. Otherwise, the init() method must return GWSTATUS_CONTINUE_GAME.

Once a new level has been loaded/initialized with a call to the init() method, our game framework will repeatedly call your StudentWorld's move() method, at a rate of roughly 20 times per second until the Player either finishes the level or dies.

Each time the move() method is called, it must orchestrate a single tick of the game. This means that it is responsible for asking each of the game actors (e.g., the Player's avatar, each enemy, goodies, etc.) to try to do something: e.g., move themselves and/or perform their specified behavior. Finally, this method is responsible for disposing of (i.e., deleting) actors that need to disappear during a given tick (e.g., a Barrel that has rolled into a Bonfire, a dead enemy, etc.).

For example, if a Koopa is blasted by a Burp, then its state should be set to dead, and then after all of the actors in the game get a chance to do something during the tick, the move() method should remove that Koopa from the game world (by using a C++ delete expression to free its object, AND removing any reference to the Koopa object's pointer 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 the Player completes the current level or loses a life. The cleanUp() method is responsible for freeing all actor objects (e.g., all Koopa objects, all Fireball objects, all Floor objects, the Player object, goodies, 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 Barrel added to the game by Kong) that have not yet been removed from the game.

You may add to your StudentWorld class as many private data members or public or private member functions as you like (in addition to the above three member functions, which you must implement).

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

```
int getLevel() const;
int getLives() const;
void decLives();
void incLives();
int getScore() const;
```

```
void increaseScore(int howMuch);
void setGameStatText(string text);
string assetPath() const;
bool getKey(int& value);
void playSound(int soundID);
```

Here's what they do:

- getLevel() can be used to determine the current level number to help you load the right level data file.
- getLives() can be used to determine how many lives the Player has left.
- decLives() reduces the number of Player lives by one.
- incLives() increases the number of Player lives by one.
- getScore() can be used to determine the Player's current score.
- increaseScore() is used by a StudentWorld object (or your other classes) to increase the user's score upon successfully destroying an enemy, picking up a goodie, or completing a level. When your code calls this method, you must specify how many points the user gets. 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.
- setGameStatText() method is used to specify what text is displayed at the top of the game screen, e.g.:

  Score: 0321000 Level: 05 Lives: 03 Burps: 05

- assetPath() returns the name of the directory that contains the game assets (image, sound, and level data files).
- getKey() can be used to determine if the user has hit a key on the keyboard to move the Player, to jump, etc. This method returns true if the user hit a key during the current tick, and false if the user did not hit any key during this tick. The only argument to this method is a variable that will be set to the key that was pressed by the user (if any key was pressed). If the function returns true, the argument will be set to one of the following values (defined in GameConstants.h):

```
KEY_PRESS_LEFT
KEY_PRESS_RIGHT
KEY_PRESS_UP
KEY_PRESS_DOWN
KEY_PRESS_SPACE
KEY_PRESS_TAB
```

- playSound() can be used to play a sound effect when an important event happens during the game (e.g., an enemy dies or the Player picks up a goodie). You can find constants (e.g., SOUND_ENEMY_DIE) that describe what noise to make in the

GameConstants.h file. Here's how this method might be used:

```
// if a Koopa reaches zero hit points and dies, make a dying sound
if (theEnemyHasDied())
    studentWorldObject->playSound(SOUND_ENEMY_DIE);
```

`init()` Details

Your StudentWorld's init() member function must:

1. Initialize the data structures used to keep track of your game's level and actors.
2. Load the current level's details from the appropriate level data file.
3. Allocate and insert a valid Player object into the game world.
4. Allocate and insert any Kong objects, Floor objects, Ladder objects, Bonfire objects, Fireball objects, Koopa objects, Goodie objects, or other relevant objects into the game world, as required by the specification in the current level's data file.

To load the details of the current level from a level data file, you can use the Level class (described later) that we wrote for you, which can be found in the provided Level.h header file. Here's a brief example that uses the Level class to load a level data file:

```
#include "Level.h"  // you must include this file to use our Level class

void StudentWorld::someFunctionYouWriteToLoadALevel()
{
    string curLevel = "level02.txt";
    Level lev(assetPath());
    Level::LoadResult result = lev.loadLevel(curLevel);
    if (result == Level::load_fail_file_not_found ||
        result == Level::load_fail_bad_format)
    {
        cout << "Something bad happened\n";
        return;
    }

    // otherwise the load was successful and you can access the
    // contents of the level - here's an example
    int x = 0;
    int y = 5;
    Level::MazeEntry item = lev.getContentsOf(x, y);
    if (item == Level::Player)
        cout << "The Player should be placed at 0,5 in the maze\n";
```

```
    x = 10;
    y = 7;
    item = lev.getContentsOf(x, y);
    if (item == Level::floor)
        cout << "There should be a floor placed at 10,7 in the maze\n";
    ... // etc
}
```

Notice that the getContentsOf() method takes the column parameter (x) first, then the row parameter (y) second. This is different from the order one normally uses when indexing a 2-dimensional array, which would be array[row][col]. Be careful!

You can examine the Level.h file for a full list of functions that you can use to access each level.

Once you load a level's layout and details using our Level class, your init() method must then construct a representation of your world and store this in your StudentWorld object. It is REQUIRED that you keep track of all of the actors (e.g., Kong, Koopa, Fireball, Barrel, Ladder, Floor, Goodies, Bonfire, etc.) in a single STL collection like a list 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 Player object rather than keeping a pointer to that object in the container with the other actor pointers; the Player is the **only** actor allowed to not be stored in the single actor container.

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 Fireball to roll, ask a Bonfire to damage the Player, give the Player a chance to move left or right, etc.).
    - If an actor does something that causes the Player to die, then the move() method should immediately return GWSTATUS_PLAYER_DIED.
    - Once Kong flees off the screen, completing the current level, then the move() method should immediately:
        - Increase the Player's score appropriately.
        - Return a value of GWSTATUS_FINISHED_LEVEL.
2. It must then delete any actors that have died during this tick (e.g., a Barrel that was attacked by a burp so it should be removed from the game world, or a goodie that disappeared because the Player picked it up).

3. It must update the status text on the top of the screen with the latest information (e.g., the user's current score, the number of burps the Player has left, etc.).

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

GWSTATUS_PLAYER_DIED
GWSTATUS_CONTINUE_GAME
GWSTATUS_FINISHED_LEVEL

The first return value indicates that the Player 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 your move() method returns this value and the Player has more lives left, then our framework will prompt the Player to continue the game, call your cleanUp() method to destroy the level, call your init() method to re-initialize the level from scratch, and then begin calling your move() method over and over, once per tick, to let the user play the level again.

The second return value indicates that the current tick completed without the Player dying and the Player has not yet completed the current level. Therefore, the game play should continue normally for the time being. In this case, the framework will display all of the actors on the screen in their updated locations and then advance to the next tick and call your move() method again.

The final return value indicates that Kong had fled off the screen because the Player has completed the current level by reaching Kong. If your move() method returns this value, then the current level is over, and our framework will call your cleanUp() method to destroy the level, advance to the next level, then call your init() method to prepare that level for play, etc.

IMPORTANT NOTE: The skeleton code that we provide to you is hard-coded to return a GWSTATUS_CONTINUE_GAME status value from our dummy version of the move() method. Until you implement code that can return one of the other statuses, the game will never end normally with your winning or losing; the only way to get out of the game is to quit abruptly by typing q.

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

```
int StudentWorld::move()
{
    // Update the Game Status Line
    updateDisplayText();   // update the score/lives/level/burps text at
screen top
```

```
    // The term "actors" refers to all enemies, the Player, goodies,
    // floors, ladders, barrels, etc.

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

    // Remove newly-dead actors after each tick
    removeDeadGameObjects(); // delete dead game objects

    // return the proper result
    if (thePlayerDiedDuringThisTick())
        return GWSTATUS_PLAYER_DIED;

    if (thePlayerCompletedTheCurrentLevel())
    {
        increaseScoreAppropriately();
        return GWSTATUS_FINISHED_LEVEL;
    }

    // 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, emit a burp, etc.).

Your move() method must, for each active actor in the game world (i.e., held by your StudentWorld object), 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.

It is possible that one actor (e.g., a Burp) may destroy another actor (e.g., a Barrel) during the current tick. If an actor has died earlier in the current tick, then the dead actor must NOT have a chance to do something during the current tick (since it's dead).

To help you with testing, if you press the f key during the course of the game, our game controller will stop calling move() every tick; it will call move() only when you hit a key (except the r key). Freezing the activity this way gives you time to examine the screen, and stepping one move at a time when you're ready helps you see if your actors are moving properly. To resume regular 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 delete their objects (so you don't have a memory leak). So if, for example, a Fireball's is hit by a Player's Burp and dies, then it should be marked as dead, and at the end of the tick, its pointer should be removed from the StudentWorld's container of active objects, and the Fireball object should be deleted (using a C++ delete expression) to free up memory for future actors that will be introduced later in the game. (Hint: Each of your actors could have a data member indicating whether or not it is still alive.)

Updating the Display Text

Your move() method must update the game statistics at the top of the screen during every tick by calling the setGameStatText() method that we provide in our GameWorld class. You could do this by calling a function like the one below from the move() method:

```
void setDisplayText()
{
    int score = getCurrentScore();
    int level = getCurrentGameLevel();
    int livesLeft = getNumberOfLivesThePlayerHasLeft();
    unsigned int burps = getCurrentPlayerBurps();

    // Next, create a string from your statistics, of the form:
    // Score: 0000100  Level: 03  Lives: 03  Burps: 08

    string s = generate_stats(score, level, livesLeft, burps);

    // Finally, update the display text at the top of the screen with your
    // newly created stats
    setGameStatText(s); // calls our provided GameWorld::setGameStatText
}
```

Your status line must meet the following requirements:

- Each field’s label is followed by a colon and one space.
- Each field’s value after the colon and space must be exactly as wide as shown in the example above:
  - The Score field must be 7 digits long, with leading zeros.
  - The Level field must be 2 digits long, with leading zeroes.
  - The Lives field must be 2 digits long, with leading zeros.
  - The Burp field must be 2 digits long, with leading zeros.
- Each statistic must be separated from the previous statistic by two spaces. For example, between the “0000100” of the score and the “L” in “Level” there must be exactly two spaces.

You may find the stringstreams writeup on the class web site to be helpful for creating a formatted string like the one above.

`cleanUp()` Details

When your cleanUp() method is called by our game framework, it means that the Player lost a life or has completed the current level. In this case, every actor in the maze (the Player and every Koopa, Fireball, Floor, Bonfire, Barrel, Ladder, Burp, Kong, goodie, etc.) must be deleted and removed from the StudentWorld’s container of active objects, resulting in an empty world. 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 the Player dies. Instead, this method will be called by our code.

IMPORTANT NOTE: The StudentWorld destructor will be called by our game framework when the game is over. If the game ends prematurely because the user pressed the q key, cleanUp() will NOT have been called by our framework, so your destructor should call it to make sure the game shuts down cleanly. In normal gameplay, the Player may have no more lives or may finish the last level, resulting in cleanUp() being called as for any level ending; a little later, the StudentWorld destructor is called, which would call cleanUp() again. MAKE SURE two consecutive calls to cleanUp() won't do anything undefined. For example, if cleanUp() deletes an object and leaves a dangling pointer, it could be disastrous if the second call to cleanUp() tries to use that pointer in a delete expression.

The Level Class and Level Data File

As mentioned, every level of Wonky Kong may have a different layout. The layout for each level is stored in a data file, with the file level00.txt holding the details for the first level’s layout, level01.txt holding the details for the second level’s layout, etc.

You may modify our layout data files to create 🔥new levels, or add your own new data files to add additional levels, if you like. During program development, when you add code to implement a new feature, you'll probably want to set up a simple level00.txt file that has just enough to let you test that feature right when the game starts so that you don't have to spend time playing for a while until you reach a situation in the game to test that feature.

Here's an example layout data file:

`level00.txt:`

```
@@@@@@@@@@@@@@@@@#@@
@                # @
@    K           # @
@ #@@@@@         < @
@ #        @@@@@@@ @
@ #       #        @
@ #                @
@ #                @
@ @@@@@@  #        @
@         #        @
@       @@@@@@@@#  @
@  @@@@@        #@@@
@               #  @
@ @@@#@@@      F#  @
@    #     @@@@@@  @
@ K  #             @
@ @@@@@@@@@@@@@@#  @
@               #  @
@B     G        # P@
@@@@@@@@@@@@@@@@@@@@
```

As you can see, the data file contains a 20×20 grid of different characters that represent the different actors in the level. Valid characters for your layout data file are:

- '@' (the "floor" character). Indicates a floor (or side wall) is at this position in the layout.
- '#' (the "ladder" character). Indicates that a ladder is at this position in the layout.
- 'P' indicates the location of the Player's avatar when starting the level.
- '<' Indicates a left-facing Kong starts at this location when the Player starts the level.
- '>' Indicates a right-facing Kong starts at this location when the Player starts the level.
- 'B' indicates a Bonfire starts at this location.
- 'F' indicates a Fireball starts at this location.
- 'K' indicates a Koopa starts at this location.
- 'E' indicates an Extra Life Goodie is placed here.
- 'G' indicates a Garlic Goodie is placed here.
- ' ' (space character) indicates an empty cell that can be walked through.

All levels must also fulfill the requirement that the bottom row and the left and right edges of the grid must be floors ('@'), with the exception that the top row can contain ladder characters. (This is enforced by our class that loads the file.)

The Level Class

We have graciously decided to provide 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());

    Level::LoadResult result = lev.loadLevel("level00.txt");
    if (result == Level::load_fail_file_not_found)
        cerr << "Could not find level00.txt data file\n";
    else if (result == Level::load_fail_bad_format)
        cerr << "Your level was improperly formatted\n";
    else if (result == Level::load_success)
    {
        cerr << "Successfully loaded level\n";

        Level::MazeEntry me = lev.getContentsOf(5,10);  // x=5, y=10
        switch (me)
        {
            case Level::floor:
                cout << "5,10 is a Floor\n";
                break;
            case Level::ladder:
                cout << "5,10 is a Ladder\n";
                break;
            case Level::left_kong:
                cout << "5,10 is a left-facing Kong\n";
                break;
            case Level::right_kong:
                cout << "5,10 is a right-facing Kong\n";
                break;
            case Level::fireball:
                cout << "5,10 is a Fireball\n";
                break;
```

```
            case Level::koopa:
                cout << "5,10 is a Koopa\n";
                break;
            case Level::bonfire:
                cout << "5,10 is a Bonfire\n";
                break;
            case Level::extra_life:
                cout << "5,10 is an Extra Life Goodie\n";
                break;
            case Level::garlic:
                cout << "5,10 is a Garlic Goodie\n";
                break;
            case Level::Player:
                cout << "5,10 is where the Player starts\n";
                break;
            case Level::empty:
                cout << "5,10 is empty\n";
                break;
        }
    }
}
```

Hint: You will presumably want to use our Level class when loading the current level specification in your StudentWorld's init() method.

You Have to Create the Classes for All Actors

The Wonky Kong game has a number of different game objects, including:

- The Player's Avatar
- Kong
- Barrels
- Fireballs
- Koopas
- Bonfire
- Ladders
- Floors
- Extra Life Goodies
- Garlic Goodies
- Burps (fired by the Player)

Each of these game objects can occupy the maze and interact with other game objects within the maze.

Now of course, many of your game objects will share things in common – for instance, every one of the objects in the game (Kong, the Player, Barrels, etc.) has x,y coordinates. Many game objects have the ability to perform an action (e.g., move, climb, fall, etc.) during each tick of the game.

Many of them can potentially be attacked (e.g., Barrels, Koopas and Fireballs can be attacked by a Burp, the Player can be attacked by a Fireball and Bonfire, etc.) and could "die" during a tick. All of them need some attribute that indicates whether or not they are still alive or they died during the current tick, etc.

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

Your grade on this project will depend upon your ability to intelligently create a set of classes that follow good object-oriented design principles. Your classes must avoid duplicating code or data members – if you find yourself writing the same (or largely similar) code 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 text nearly identically in the following sections (e.g., in the Extra Life Goodie and the Garlic Goodie sections, or in the Enemy-based classes), you must make sure to identify common behaviors and move these into proper base classes. NEVER duplicate behaviors across classes that can be moved into a base class!

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

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

class Koopa: public Actor
{
public:
    ...
};
```

If you don't derive a class from our GraphObject base class, then you won't see any objects of that type displayed on the screen!

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

```
GraphObject(int imageID, int startX, int startY, int startDirection = none);
int getX() const;
int getY() const;
void moveTo(int x, int y);
int getDirection() const;  // Directions: none, up, down, left, right
void setDirection(int dir);  // Directions: none, up, down, left, right
void getPositionInThisDirection(int dir, int units, int& newX, int& newY) const;
void increaseAnimationNumber();
```

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 provide overriding implementations of any of these methods in your derived classes since they are not declared as virtual in our base class.

GraphObject(int imageID, int startX, int startY, int startDirection)

This 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 Kong, a Player, a Floor, 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. (VIEW_WIDTH and VIEW_HEIGHT are defined in GameConstants.h.) Notice that you pass the coordinates as x,y (i.e., column, row starting from bottom left, and not row, column). The bottom-left corner is at x=0,y=0; the top-right corner is at x=VIEW_WIDTH-1,y=VIEW_HEIGHT-1. For those objects for which the concept of a direction doesn't apply (e.g., Floors), you may leave off the final constructor argument or may specify *none*; For those objects for which a direction applies (i.e., the Player, enemies, Burps), you must specify the initial direction the object is facing (i.e., *left*, *right*, *up*, or *down* – these constants are defined in the GraphObject.h file).

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

```
IID_PLAYER
IID_KONG
IID_BARREL
IID_FIREBALL
IID_KOOPA
IID_FLOOR
IID_LADDER
IID_EXTRA_LIFE_GOODIE
```

```
IID_GARLIC_GOODIE
IID_BONFIRE
IID_BURP
```

getX() and getY()

These are used to determine a GraphObject's current location in the maze. Since each GraphObject maintains its x,y location, this means that your derived classes must not also have x,y member variables, but instead use these functions and moveTo() from the GraphObject base class.

moveTo(int x, int y)

This is used to update the location of a GraphObject within the maze. For example, if a Koopa's movement logic dictates that it should move to the right, you could do the following:

```
moveTo(getX()+1, getY()); // move one square to the right
```

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()

This is used to determine the direction a GraphObject is facing. For example, the Player can face either left or right, so you can use this method to learn that direction.

setDirection(int dir)

This is used to change the direction a GraphObject is facing. For example, when the user presses the left arrow, you can use this method to cause the Player's avatar to be displayed facing left.

getPositionInThisDirection(int dir, int units, int& newX, int& newY)

This calculates the coordinates of a square located a certain number of squares away from the current GraphObject in a specified direction. This is particularly useful for determining the starting position of a projectile, such as a Burp fired by the Player or a Barrel thrown by Kong, relative to their originating actor.

increaseAnimationNumber()

This is used by some types of non-moving objects to make them appear more lively on the display. The descriptions of the behavior of a Bonfire and a Kong tell you when you'll use this.

The Player

Here are the requirements you must meet when implementing the Player class for Wonky Kong.

What the Player Must Do When It Is Created

When it is first created:

1. The Player object must have an image ID of IID_PLAYER.
2. The Player must always start at the proper location as specified by the current level's data file. Hint: Since your StudentWorld's init() function loads the level, it knows this (x,y) location to pass when constructing the Player object.
3. The Player in its initial state:
    1. Has 3 lives.
    2. Has 0 burps.
    3. Faces right.

What the Player Must Do During a Tick

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

- If the Player is not alive:
    - Do nothing for this tick and return from the method.
- If the Player is currently performing a jump sequence:
    - Execute the next step of the jump sequence (see below)
    - Do nothing else for this tick and immediately return.
- If the Player is frozen (i.e., cannot perform actions for a certain number of ticks):
    - Decrement the freeze timer by one tick.
    - Do nothing else for this tick and immediately return.
- Check if there is no solid floor or climbable object directly beneath the Player's current position.
    - If the space below is empty and does not contain a Floor or Ladder:
        - Attempt to move the Player one square downward (simulate falling).
        - Do nothing else for this tick and immediately return.
- Check if the user has pressed any key during this tick. If a key is pressed:
    - Left/Right Arrow Key Pressed:
        - If the Player is not already facing in this direction, change the Player's direction to the new direction.
        - Else if the Player is not blocked by a floor from moving in the specified direction, move the Player one square to the specified direction.
    - Up/Down Arrow Key Pressed:
        - If the Player is in a position to climb (e.g., on a ladder) in the specified direction and there is no floor blocking movement, update the Player's position accordingly.
    - Space Bar Pressed:

        - If the Player is allowed to initiate a jump (i.e., they not already jumping and they are either standing one square above a floor or ladder or they are currently on the same square as a ladder), then:
            - Perform the first step of a new jump sequence (see below).
            - Play the jump sound effect.
    - Tab Key Pressed:
        - Check if the Player has any burps remaining. If burps are available:
            - Play the Burp sound effect (SOUND_BURP)
            - Create a new Burp object in the square immediately adjacent to the Player in the direction the Player is currently facing and add it to the game.
            - Decrease the Player's Burp count by one.

Jumping

When the user initiates a jump by pressing the spacebar, the Player object must move one position during the current tick T and each of the subsequent 4 ticks. Assuming the Player is currently facing direction D (left or right) when they initiate the jump, the Player must move as specified bythis jump sequence unless interrupted by a floor or ladder as described below:

- Tick T: Move up
- Tick T+1: Move in direction D
- Tick T+2: Move in direction D
- Tick T+3: Move in direction D
- Tick T+4: Move down

Here is what it means to perform a step of the jump sequence:

1. Compute the new target x,y location for the next step in the jump sequence
2. Check if the new location is a valid location that is free of obstructions:
    - If movement is blocked by a floor or the jump would take the Player off the screen:
        - Do not move the Player
        - Terminate the jump sequence.
    - Otherwise, if movement to the new position is possible:
        - Move the Player to the new position.
        - If the new position is on a climbable surface, terminate the jump sequence (the Player has grabbed onto the ladder).
        - Otherwise, the next jump step will be done on the next tick
2. If the Player has completed all steps in the current jump sequence then the jump has ended.

Getting Input From the User

Since Wonky Kong is a real-time game, you can’t use the typical getline or cin approach to get a user's key press within the Player’s doSomething() method—that would stop your program and wait for the user to hit a key, then hit Enter, then hit a 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 user. 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 Player::doSomething()
{
    ...
    int ch;
    if (getWorld()->getKey(ch))
    {
        // user hit a key this tick!
        switch (ch)
        {
          case KEY_PRESS_LEFT:
            ... turn or move left ...
            break;
          case KEY_PRESS_RIGHT:
            ... turn or move right ...
            break;
          case KEY_PRESS_SPACE:
            ... initiate jump ...
            break;
          case KEY_PRESS_TAB:
            ... burp if you have burps left ...
            break;
          // etc…
        }
    }
    ...
}
```

Hint: Since your Player class will need to access the getKey() method in the GameWorld class (which is the base class for your StudentWorld class), your Player 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 the Player'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 Player Must Do When It Is Attacked

1. The Player must play a SOUND_PLAYER_DIE sound.
2. The Player must set their status to dead.

What a Player Must Do When It Is Frozen

1. The Player must be frozen for 50 ticks.

Floor

Floors don’t really do much. They just sit still in place. Here are the requirements you must meet when implementing the Floor class in Wonky Kong.

What a Floor Must Do When It Is Created

When it is first created:
1. A floor object must have an image ID of IID_FLOOR.
2. A floor must always start at the proper location as specified by the current level’s data file.
3. A floor has no direction (Hint: Its ancestor GraphObject base object always has the direction none).

What a Floor Must Do During a Tick

A floor must be given an opportunity to do something during every tick (in its doSomething() method). When given an opportunity to do something, the floor must do nothing. After all, it’s just a floor!

What a Floor Must Do When It Is Attacked

When a floor is attacked, nothing happens to the floor.

Ladder

Ladders don’t really do much. They just sit still in place. Here are the requirements you must meet when implementing the Ladder class in Wonky Kong.

What a Ladder Must Do When It Is Created

When it is first created:
1. A ladder object must have an image ID of IID_LADDER.
2. A ladder must always start at the proper location as specified by the current level’s data file.
3. A ladder has no direction.

What a Ladder Must Do During a Tick

A ladder must be given an opportunity to do something during every tick (in its doSomething() method). When given an opportunity to do something, the ladder must do nothing.

What a Ladder Must Do When It Is Attacked

When a ladder is attacked, nothing happens to the ladder.

Burp

You must create a class to represent a Burp. Here are the requirements you must meet when implementing the Burp class for Wonky Kong.

What a Burp Must Do When It Is Created

When it is first created:

1. A Burp object must have an image ID of IID_BURP.
2. A burp must have its x,y location specified for it — the actor that emits the Burp must pass in this x,y location when constructing the object.
3. A Burp must have its direction specified for it — the actor that emits the burp must pass in this direction during construction.
4. A Burp must track how long it will remain in the world before it disappears. This duration must be initialized to 5 ticks when the burp is created.

What a Burp Must Do During a Tick

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

1. The Burp 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 Burp must decrement its remaining lifetime by 1 tick. If its lifetime has reached zero, the Burp must:
   - Set its own state to dead (so that it will be removed from the game by the StudentWorld object at the end of the current tick).
   - Do nothing more during this tick.
3. If the Burp is still alive after step #2, it must attack any actor on its current square that can be blasted (e.g., an enemy).

Note: Since Burp gas hangs around for a while, a Burp can potentially attack multiple enemies.

What a Burp Must Do When It Is Attacked

Burps can’t be attacked, silly. When a burp is attacked, nothing happens to the burp.

Bonfire

You must create a class to represent a Bonfire. When the Player is on the same square as a Bonfire, the Bonfire attacks the Player, causing them to lose a life. In addition, if a Barrel moves onto the same square as the Bonfire, that barrel is attacked and destroyed. Other enemies are

immune to Bonfires. Here are the requirements you must meet when implementing the Bonfire class.

What a Bonfire Must Do When It Is Created

When it is first created:
1. A Bonfire object must have an image ID of IID_BONFIRE.
2. A Bonfire must always start at the proper location as specified by the current level’s data file.
3. A Bonfire has no direction.

What a Bonfire Must Do During a Tick

Each time a Bonfire object is asked to do something (during a tick):
1. The Bonfire increments its animation frame each tick by calling increaseAnimationNumber() (inherited from GraphObject) so that its animation looks lively on every frame. This is required because the Bonfire doesn't move between squares, which normally advances the graphic animations, so we must update its animation explicitly.
2. If the Player is on the same square as a Bonfire, then the Bonfire must attack the Player causing them to lose a life.
3. If a Barrel is on the same square as the Bonfire, the Barrel must be attacked and destroyed (the Bonfire remains alive).
4. All other enemies are unaffected by the Bonfire.

What a Bonfire Must Do When It Is Attacked

When a bonfire is attacked, nothing happens to the bonfire.

Extra Life Goodie

You must create a class to represent an Extra Life Goodie for Wonky Kong. When the Player picks up this goodie (by moving onto the same square as it), it gives the Player an extra life! Here are the requirements you must meet when implementing the Extra Life Goodie class.

What an Extra Life Goodie Must Do When It Is Created

When it is first created:
1. An Extra Life Goodie object must have an image ID of IID_EXTRA_LIFE_GOODIE.
2. An Extra Life Goodie must always start at the proper location as specified by the current level’s data file.
3. Extra Life Goodies have no direction.

What an Extra Life Goodie Must Do During a Tick

Each time an Extra Life Goodie is asked to do something (during a tick):

1. The Extra Life Goodie must check to see if it is currently alive. If it is not alive, then its doSomething() method must return immediately – none of the following steps should be performed.
2. If the Player is on the same square as the Extra Life Goodie, then the Extra Life Goodie must:
    1. Inform the StudentWorld object that the user is to receive 50 more points.
    2. Set its own state to dead (so that it will be removed from the game by the StudentWorld object at the end of the current tick).
    3. Play a sound effect to indicate that the Player picked up the goodie: SOUND_GOT_GOODIE.
    4. Give the Player one extra life.

What an Extra Life Goodie Must Do When It Is Attacked

When an extra life goodie is attacked, nothing happens to the extra life goodie.

.

Garlic Goodie

You must create a class to represent a garlic goodie for Wonky Kong. When the Player picks up this goodie (by moving onto the same square as it), it gives the Player additional burps! Here are the requirements you must meet when implementing the Garlic Goodie class.

What a Garlic Goodie Must Do When It Is Created

When it is first created:

1. A Garlic Goodie object must have an image ID of IID_GARLIC_GOODIE.
2. A Garlic Goodie must always start at the proper location as specified by the current level's data file.
3. Garlic Goodies have no direction.

What a Garlic Goodie Must Do During a Tick

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

1. The garlic goodie must check to see if it is currently alive. If it is not alive, then its doSomething() method must return immediately – none of the following steps should be performed.
2. If the Player is on the same square as the garlic goodie, then the garlic goodie must:
    1. Inform the StudentWorld object that the user is to receive 25 more points.
    2. Set its own state to dead (so that it will be removed from the game by the StudentWorld object at the end of the current tick).

3. Play a sound effect to indicate that the Player picked up the goodie: SOUND_GOT_GOODIE.
4. Give the Player five additional burps.

What a Garlic Goodie Must Do When It Is Attacked

When a garlic goodie is attacked, nothing happens to the garlic goodie.

Fireball

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

What a Fireball Must Do When It Is Created

When it is first created:
1. A Fireball object must have an image ID of IID_FIREBALL.
2. A Fireball must always start at the proper location as specified by the current level's data file.
3. A Fireball must start out facing either left or right, with an equal probability of either (this may be determined via random number generation; GameConstants.h has a randInt function you may use).
4. A Fireball must start out in a non-climbing state.

What a Fireball Must Do During a Tick

Each time a Fireball is asked to do something (during a tick):
1. If the Fireball is not currently alive, then its doSomething() method must return immediately. None of the following steps should be performed.
2. If the Fireball is on the same square as the Player then:
    a. The Fireball must attack the Player.
    b. Immediately return
3. Once every 10 ticks (i.e., once every 10 calls to the Fireball's doSomething() method), the Fireball must also do the following, unless step 2 returned:
    a. If the Fireball's current square is climbable, there is no solid object above the Fireball, and the Fireball is not in a climbing down state:
        - If the Fireball is already in a climbing up state, or if a randomly chosen 1 out of 3 chance occurs, it must:
            - Set its state to climbing up
            - Move one square up
            - Once it moves, the Fireball should skip to step e
    b. Otherwise, if the square below the Fireball is climbable and the Fireball is not in a climbing up state, it must decide whether to climb down.
        - If the Fireball is already in a climbing down state, or if a randomly chosen 1 out of 3 chance occurs, it must:

- Set its state to climbing down
- Move one square down.
- Once it moves, the Fireball should skip to step e

c. If while in a climbing state, the Fireball cannot climb any higher or lower in its climbing state direction (e.g., runs into a floor above/below or runs out of ladder above/below it), the Fireball transitions into a non-climbing state.

d. The Fireball will attempt to move horizontally in the direction it is currently facing:
   - If the square in front of the Fireball is solid (i.e., a floor) OR there is no floor/ladder beneath the square where the Fireball wishes to move, the Fireball does not move to that square, but instead reverses its direction (left ↔ right).
   - Otherwise, the Fireball moves to the square in the direction it is currently facing.

e. If the Fireball is on the same square as the Player then:
   - The Fireball must attack the Player.
   - Immediately return

What a Fireball Must Do When It Is Attacked

A Fireball must be able to handle being attacked by a Burp. When a Fireball is attacked:

1. The Fireball must set itself to a dead state so that it will be removed from the game.
2. The Fireball must play the appropriate enemy death sound (SOUND_ENEMY_DIE).
3. The Player must receive +100 points for defeating the Fireball.
4. There is a random 1 out of 3 chance that the Fireball will drop a GarlicGoodie at its current location after having been defeated.

Koopa

You must create a class to represent a Koopa. 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 must have an image ID of IID_KOOPA.
2. A Koopa must always start at the proper location as specified by the current level's data file.
3. A Koopa must start out facing either left or right, with an equal probability of either (this may be determined via random number generation).
4. A Koopa must keep track of a freeze cooldown timer (an integer). When created, this timer must be set to 0, indicating it is ready to perform its freeze action if it comes into contact with the Player.

What a Koopa Must Do During a Tick

Each time a Koopa is asked to do something (i.e., each game tick):

1. If the Koopa is not currently alive, then its doSomething() method must return immediately. None of the following steps should be performed.
2. If the Koopa is on the same square as the Player and the Koopa's freeze cooldown value is 0, the Koopa will activate:
    - The Koopa must cause the Player to become frozen.
    - The Koopa sets its own freeze cooldown timer to 50 ticks, preventing it from repeatedly freezing the Player for at least another 50 ticks.
    - Immediately return.
3. If the Koopa's freeze cooldown timer is greater than 0, it is decremented by 1.
4. Once every 10 ticks (i.e., once every 10 calls to the Koopa's doSomething() method), the Koopa must also do the following, unless step 2 returned:
    a. The Koopa will attempt to move horizontally in the direction it is currently facing:
        - If the square in front of the Koopa is solid (i.e., a floor) OR there is no floor/ladder beneath the square where the Koopa wishes to move, the Koopa does not move to that square, but instead reverses its direction (left ↔ right).
        - Otherwise, the Koopa moves to the square in the direction it is currently facing.

    b. If the Koopa is on the same square as the Player and the Koopa's freeze cooldown value is 0, the Koopa will activate:
        - The Koopa must cause the Player to become frozen for 50 ticks.
        - The Koopa sets its own freeze cooldown timer to 50 ticks, preventing it from repeatedly freezing the Player for another 50 ticks.
        - Immediately return.

What a Koopa Must Do When It Is Attacked

A Koopa must be able to handle being attacked by a Burp. When a Koopa is attacked:

1. The Koopa must set itself to a dead state so that it will be removed from the game.
2. The Koopa must play the appropriate enemy death sound (SOUND_ENEMY_DIE).
3. The Player must receive +100 points for defeating the Koopa.
4. There is a random 1 out of 3 chance that the Koopa will drop an ExtraLifeGoodie at its current location after having been defeated.

Barrel

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

What a Barrel Must Do When It Is Created

When it is first created:

1. A Barrel object must have an image ID of IID_BARREL.
2. A Barrel must have its direction (left or right) specified for it — the actor that creates the barrel must pass in this direction during construction.

What a Barrel Must Do During a Tick

Each time a Barrel is asked to do something (i.e., each game tick):

1. If the Barrel is not currently alive, then its doSomething() method must return immediately. None of the following steps should be performed.
2. If the Barrel is on the same square as the Player then:
    - The Barrel must attack the Player.
    - Immediately return.
3. If there is a destructive entity (e.g., a Bonfire) in the same square as the Barrel, the Barrel:
    - Must set its status to dead.
    - Immediately return.
4. If there is no solid floor directly beneath the Barrel:
    - It must move exactly one square down.
    - Once the Barrel finally lands on a Floor, it must reverse the direction it is facing (left ↔ right) such that it will move in the opposite direction it was moving before it fell.
5. Once every 10 ticks (i.e. once every 10 calls to Barrel's doSomething() method), the Barrel must also do the following:
    a. The Barrel will attempt to move horizontally in the direction it is currently facing:
        - If the square in front of the Barrel is solid (i.e., a floor) then the Barrel does not move to that square and instead reverses its direction (left ↔ right).
        - Otherwise, the Barrel moves to the square in the direction it is currently facing.
    b. If the Barrel is on the same square as the Player then:
        - The Barrel must attack the Player.
        - Immediately return.

What a Barrel Must Do When It Is Attacked

A Barrel must be able to handle being attacked by a Burp. When a Barrel is attacked

1. The Barrel must set itself to a dead state so that it will be removed from the game.
2. The Barrel must play the appropriate enemy death sound (SOUND_ENEMY_DIE).
3. The Player must receive +100 points for defeating the Barrel.

Kong

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

What a Kong Must Do When It Is Created

When it is first created:

- A Kong must have an image ID of IID_KONG.
- A Kong must always start at the proper location as specified by the current level's data file.
- A Kong must start out facing either left or right as specified by the current level's data file.
- A Kong must maintain a "flee" state, which starts as false (meaning it is not fleeing initially).

What a Kong Must Do During Each Tick

Each time a Kong is asked to do something (i.e., each game tick):

1. If the Kong is not alive, then its doSomething() must end immediately. No further actions should be taken.
2. The Kong must increment its animation frame each tick by calling increaseAnimationNumber() (inherited from GraphObject) so that its animation looks lively on every frame. This is required because a Kong doesn't move between squares, which normally advances the graphic animations, so we must update its animation explicitly.
3. If the Euclidean distance between corresponding points of the squares occupied by Kong and the Player is less than or equal to 2, then:
   - Kong must transition into a "flee" state.

   (Examples: The distance between (12,16) and (14,16) is 2; the distance between (13,15) and (14,16) is less than 2; the distance between (12,15) and (14,16) is greater than 2.)
4. If Kong is not in the "flee" state AND N ticks have elapsed since Kong last threw a Barrel, where N = max(200 - 50 * StudentWorld::getLevel(), 50), then:
   - Create a new Barrel object in the square immediately adjacent to Kong in the direction it is currently facing and add it to the game.
5. Once every 5 ticks (i.e. once every 5 calls to Kong's doSomething() method), Kong must also do the following:
   a. If Kong is in the "flee" state:
      - It must attempt to move up by one square
      - If moving up one square reaches the top of the screen:
        - The Player should immediately earn 1000 points.
        - The Kong must play the appropriate celebration sound (SOUND_FINISHED_LEVEL).
        - Kong must tell StudentWorld that the current level is finished (so the game proceeds to the next level or ends if it was the last level).

Object Oriented Programming Best Practices

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 NOT use the imageID (e.g., IID_KOOPA, IID_PLAYER, IID_BARREL, etc.) to determine the type of an object or store the imageID inside any of your objects as a member variable. You may also not use any other similar approach (e.g., a member string with the object's name, an enumerated type, etc.). Creative approaches that try to accomplish the same thing are also banned. Doing so will result in a score of ZERO for this project.

2. Avoid using dynamic_cast to identify common types of objects. Instead add methods to check for various classes of behaviors:

Don't do this:

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

Do this instead:

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

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

Don't do this:

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

Do this instead:

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

4. If two related subclasses (e.g., SmellyRobot and GoofyRobot) each define a data member that serves the same purpose in both classes (e.g., m_OilLeft), then move that data member to the common base class and add accessor and mutator methods for it to the base class. So the Robot base class should have the m_OilLeft member variable defined once, with getOil() and addOil()functions, rather than defining this data member 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, except that 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 collection of game objects or a collection of pointers to game objects, or a pointer or reference to such a collection, or an iterator pointing into such a collection. 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 requires looking at several game objects that it keeps track of.

Don't do this:

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

class NastyRobot
{
  public:
    virtual void doSomething()
    {
```

```
      ...
      vector<Actor*> v =
               studentWorldPtr->getActorsThatCanBeZapped(getX(), getY());
      vector<Actor*>::iterator it;
      for (it = actors.begin(); it != actors.end(); it++)
        (*it)->zap();

    }
};
```

Do this instead:

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

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

Note that it's acceptable to return a pointer to single game object:

```
class StudentWorld
{
  public:
```

```
    Actor* findOneZappableActor(int x, int y)
    {
      vector<Actor*>::iterator it;
      for (it = actors.begin(); it != actors.end(); it++)
      {
        Actor* p = *it;
        if (p->isAt(x,y) && p->isZappable())
         return p;
      }
      return nullptr;
    }
};
```

8. If two subclasses have a method that shares some common functionality, but also has some differing functionality, use an auxiliary method to factor out the differences:
Don't do this:

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

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

Do this instead:

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

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

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

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

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

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

```
      polishMyChrome();
      wipeMyDisplayPanel();
    }
};
```

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

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

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

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

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

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

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

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

2. Once you’ve got your program compiling with dummy functions, then start by replacing one dummy function at a time. Update the function, rebuild your program, test your new function, and once you’ve got it working, proceed to the next function.

3. Make backups of your working code frequently. Any time you get a new feature working, make a backup of all your .cpp and .h files just in case you screw something up later.

BACK UP YOUR .CPP AND .H FILES TO A REMOVABLE DEVICE OR TO ONLINE STORAGE JUST IN CASE YOUR COMPUTER CRASHES!

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, sound, and level data 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 Mac OS X). 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:/WonkyKong/Assets" or "/Users/fred/WonkyKong/Assets").

To build the game, follow these steps:

For Windows

Unzip the WonkyKong-skeleton-windows.zip archive into a folder on your hard drive. Double-click on WonkyKong.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 Mac OS X

Unzip the WonkyKong-skeleton-mac.zip archive into a folder on your hard drive.
Double-click on our provided WonkyKong.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/WonkyKong/DerivedData/WonkyKong/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 **Wonky Kong** 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., the Player, Kong, Barrels, Koopas, Fireballs, Bonfires, goodies, Floors, Ladders, etc.):
    - It must have a simple constructor.
    - It must be derived from our `GraphObject` class.
    - It must have a member function named `doSomething()` that can be called to cause the actor to do something.
    - You may add other public/private member functions and private data members to this base class, as you see fit.
2. **A Floor class**, derived in some way from the base class described in item 1 above:
    - It must have a simple constructor.
    - It must have an Image ID of `IID_FLOOR`.
    - You may add other public/private member functions and private data members to your Floor class as you see fit, so long as you use good object-oriented programming style (e.g., you must not duplicate functionality across classes).
3. **A limited version of your Player class**, derived in some way from the base class described in item 1 above (either directly derived from that base class, or derived from some other class that is itself derived from that base class):
    - It must have a constructor that initializes the **Player**.
    - It must have an Image ID of `IID_PLAYER`.
    - It must have a **limited version** of `doSomething()` that reacts to horizontal user input (e.g., left/right movements). If the user tries to move the Player in some direction and the target square is not blocked, update the Player's location and direction accordingly. Do not worry about implementing falling behavior. All this `doSomething()` method has to do is properly adjust the Player's x,y coordinates and direction; our graphics system will automatically show the Player moving around the level!
    - You may add other public/private member functions and private data members to your Player class as you see fit, so long as you use good object-oriented programming style (e.g., you must not duplicate functionality across classes).
4. **A limited version of the StudentWorld class**:
    - Add any private data members required to keep track of Floors as well as the **Player** object. You may ignore all other actors (e.g., Barrels, Fireballs, Bonfires, goodies, etc.) for Part #1.

- Implement a constructor for this class that initializes your data members.
- Implement a destructor for this class that frees any remaining dynamically allocated data that has not yet been freed at the time the `StudentWorld` object is destroyed.
- Implement the `init()` method in this class. It must create the Player actor and place it into the level at the correct starting location (see the Level class section of this document for details). It must also create all of the Floor objects and add them into the level as specified in the current Level's data file.
- Implement the `move()` method in your `StudentWorld` class. During each tick, it must ask your Player and Floors to do something. Your `move()` method need not check to see if Player has died or not; you may assume at this point that the Player cannot die. Your `move()` method does not have to deal with any actors other than the Player and Floors for Part #1.
- Implement a `cleanUp()` method that frees any dynamically allocated data that was allocated during calls to the `init()` method or the `move()` method (e.g., it should delete all your allocated Floors and the Player). 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, try to remove them, and get your program to run.

You'll know you're done with **Part #1** when your program builds and does the following: when it runs and the user hits Enter to begin playing, it displays a level with the Player in the proper starting position. If your classes work properly, you should be able to move Player left and right using the directional keys, without passing through any Floors.

Your **Part #1** solution may do more than what is specified above; for example, if you are further along in the project, and what you have builds and has at least as much functionality as what's described above, then you may turn that in instead.

Note, the **Part #1** specification above **doesn't require** you to implement Barrels, Fireballs, Koopas, Bonfires, Kong, goodies, or finishing the level (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 Player, Floor, 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 **Wonky Kong** game, which must build without errors under either Visual Studio or Xcode. You do not have to get it to run under more than one compiler. You will turn in a zip file containing exactly these four files:

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

Part #2 (80%)

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

In **Part #2**, your goal is to implement a **fully working version** of the **Wonky Kong** 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 **Wonky Kong** game, which must build without errors under either Visual Studio or Xcode. You do not have to get it to run under more than one compiler. You will turn in a zip file containing exactly these five files:

1. `Actor.h` – contains declarations of your actor classes (including any base classes, Player, enemies, goodies, etc.) as well as constants required by these classes
2. `Actor.cpp` – contains the implementation of these classes
3. `StudentWorld.h` – contains your `StudentWorld` class declaration
4. `StudentWorld.cpp` – contains your `StudentWorld` class implementation
5. `report.docx` or `report.txt` – your report (5% of your grade)

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

You must turn in a **report** that contains the following:

1. A description of the control flow for the interaction of the Player and a goodie. Where in the code is the co-location of the two objects detected, and what happens from that point until the interaction is finished? Which functions of which objects are called and what do they do during the handling of this situation?
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 Koopa's freeze logic." or "My Bonfire code doesn't actually remove Barrels from the maze."
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."

That's it! After finishing Part #2, you will have a complete **Wonky Kong** game with floors, ladders, barrels, enemies, goodies, a working Kong or Player actor, and more. Good luck, and have fun!

FAQ

Q: Why does my 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 14 in main.cpp with a larger value, like 20 or 50:

```
const int msPerTick = 10;
```

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

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

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

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

GOOD LUCK!