Project 3
Bug Blast

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

Time due:

Part 1: 9 PM, Saturday, February 22
Part 2: 9 PM, Thursday, February 27

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

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

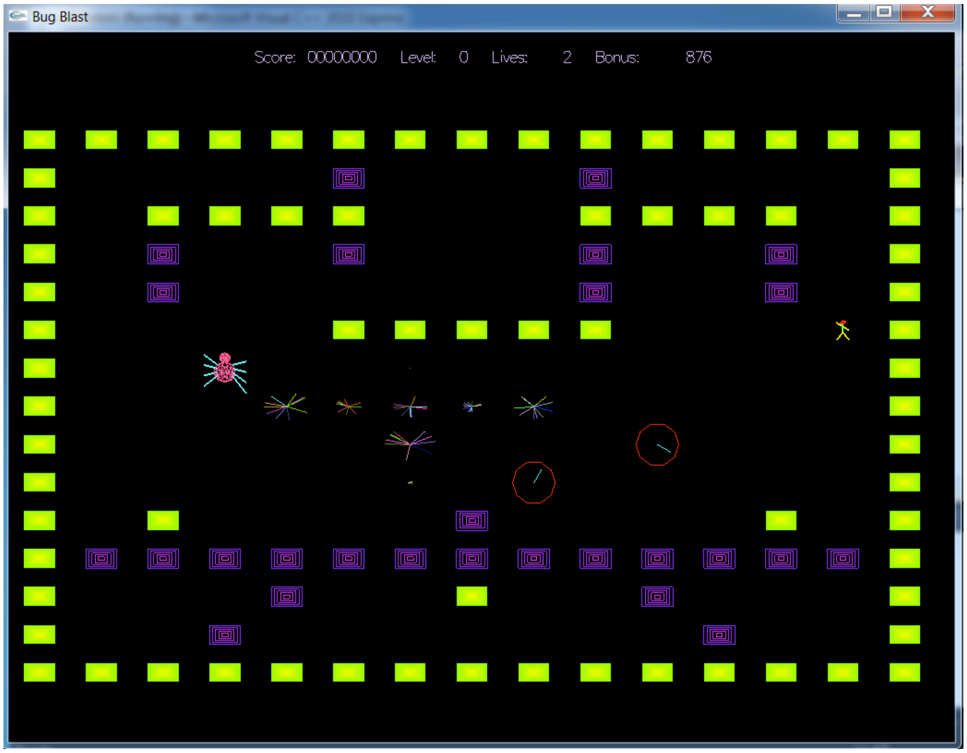
Table of Contents

Introduction

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

Bug Blast is a simplified version of the original Bomberman game created by Hudson Soft (based in Japan) in 1983. In Bug Blast, the player has to navigate through a series of mazes exterminating nasty bugs called Zumi using bug spray. After exterminating all of the Zumi within a given maze, the player may head to the Exit square on that level in order to advance to the next maze. The player wins by completing all of the mazes.

Here is an example of what the Bug Blast game looks like:



```
Score: 00000000   Level: 0   Lives: 2   Bonus: 876
```

Figure #1: A screenshot of the Bug Blast game. You can see the Player (a stick figure), a Simple Zumi (the spider-like creature), two Bug Sprayers (that look like clocks), and the mist from a recently-triggered sprayer (which looks like fireworks)

Game Details

In Bug Blast, the Player starts out a new game with three lives and continues to play until all of his/her lives have been exhausted. There are multiple levels in Bug Blast, beginning with level 0, and each level has its own maze. During each level, the Player must exterminate all of the Zumi within the current maze before they may head to the exit and move on to the next level's maze.

Upon starting each level, the Player's avatar is placed in a maze filled with zero or more Zumi bugs. The player may use the arrow keys to move their avatar (the stick figure that represents the human player) left, right, up, or down through the maze of Bricks. They may walk on any square so long as it doesn't have a Brick on it (and may even walk on some Bricks in certain circumstances – see below). When the avatar gets near a Zumi they'd like to exterminate, they can drop a timer-based Bug Sprayer on their current square in the maze. After a specified duration, the timer will expire and the Bug Sprayer will spray Bug Spray (i.e., poison mist) into the squares surrounding the sprayer. This mist, which dissipates after several seconds, is instantly deadly to both Zumi as well as the Player, and in fact, it's so corrosive that it's capable of destroying certain Bricks (Destroyable Bricks) within each maze. When a Bug Sprayer triggers and releases its Bug Spray, this Bug Spray will also trigger other Bug Sprayers that are within range of the released spray (this can potentially cause chain reactions).

The Player's avatar is allowed to drop up to two Bug Sprayers within the maze at a time. Once the Player has dropped two sprayers (which have not yet triggered and released their mist), they are no longer allowed to drop additional sprayers until one or more of the existing sprayers have triggered and released their mist.

Zumi live indefinitely and only die if they come into contact with Bug Spray (aka poison mist) from a Bug Sprayer. When a Zumi dies, there is a chance (which varies level by level) that it will randomly drop a "Goodie" (a special object that helps the Player) into the maze. There are three different types of Goodies that a Zumi can drop: Extra Life Goodies, Increase Simultaneous Sprayer Goodies and Walk Through Walls Goodies. If the Player's avatar steps upon the same square as an Extra Life Goodie, the Goodie instantly gives the avatar an extra life. If the avatar steps onto the same square as an Increase Simultaneous Sprayer Goodie, the Goodie will temporarily allow the avatar to simultaneously drop up to six Bug Sprayers at a time rather than the usual two. Finally, if the avatar steps onto the same square as a Walk Through Walls Goodie, the Goodie will temporarily allow the avatar to walk through Destroyable Bricks within the maze (but not permanent Bricks). Goodies quickly disappear after being dropped by Zumi, so it's important for the Player to grab them quickly.

There are two different types of Zumi bugs in Bug Blast: Simple Zumi and Complex Zumi. Simple Zumi are a primitive species with limited intelligence. These bugs simply walk in straight lines until they run into a Brick, at which point they may choose a new direction to move. Complex Zumi are an offshoot species that are much more intelligent – if they are near the avatar, they will move aggressively toward her/him. However, when they are not near the avatar, they exhibit Simple Zumi-like behavior and move in straight lines until they hit Bricks.

Once the Player has exterminated all Zumi within the current maze (or if there are no Zumi within a maze), an Exit will appear. The Exit is invisible and unusable until all Zumi have been exterminated from the level. The Player must direct their avatar to the exit in order to move onto the next level. The game is complete once the Player has killed all of the Zumi on all of the levels of the game, and used the Exit on the last level.

If the Player's avatar comes into contact with a Zumi (i.e., they move onto the same square as a Zumi, or a Zumi moves onto the same square as their avatar), or the avatar inhales the deadly Bug Spray (i.e., they move onto a square that contains poison mist), they will die and lose one of their lives. If, after losing a life, they have one or more remaining lives left, they are placed back on their current level and they must again solve the entire level from scratch (with all of the Zumi alive again, all Bricks as they were initially, etc.). If the avatar dies and has no lives left, then the game is over.

The Bug Blast maze is exactly 15 squares wide by 15 squares high, and the Player's avatar may occupy any square that doesn't contain some type of Brick (in some circumstances, described later, it can even occupy squares that contain Destroyable Bricks). Zumi may occupy any square so long as the square doesn't contain a Brick or a Bug Sprayer. The bottom-leftmost square has coordinates x=0,y=0, while the upper-rightmost square has coordinate x=14,y=14, where x increases to the right and y increases upward toward the top of the screen. You can look in our provided file, GameConstants.h, for constants that represent the maze's width and height.

In Bug Blast, each level's maze layout as well as parameters used to govern gameplay on the level are stored in a different data file. For example, the first level's maze and parameters are stored in a file called level00.dat. The second level's maze and parameters are stored in a file called level01.dat, and so on. Each time the Player is about to start a new level, your code must load the data from the appropriate data file (using a class called *Level* that we provide) and then use this data to determine the nature of the current level.

Each level data file contains a specification for the layout of the maze, the initial locations of all the Zumis and the Player's avatar, as well as a set of parameters that specify items such as the likelihood that a Zumi will drop a Goodie when it dies, or how quickly Zumi should move. These parameters allow the game designer to adjust the difficulty of each level and make the game get progressively harder as the Player advances levels. For more information on the level data files, please see the Level Data File section below. You may define your own level data files in order to customize your game (and test your game, for that matter).

Once a new maze has been prepared and the Player's avatar and all of the Zumi are in their proper positions, the game play begins. Game play is divided into *ticks*, and there are twenty ticks per second (to provide smooth animation and gameplay). During each tick, the following occurs:

1. The Player has an opportunity to move their avatar exactly one square horizontally or vertically, or drop a Bug Sprayer into the maze.
2. Every other object in the maze (e.g., Zumi, Bug Sprayers, Bug Spray, Goodies, etc.) is given an opportunity to do something. For example, when given the opportunity to do something, a Zumi can move one square (left, right, up or down) according to its built-in movement algorithm. (Zumi movement algorithms are described in detail in the various Zumi sections below.)

The player controls the direction of their avatar with the arrow keys, or for lefties and others for whom the arrow key placement is awkward, WASD or the numeric keypad: up is *w* or *8*, left is *a* or *4*, down is *s* or *2*, right is *d* or *6*.

The Player's avatar may drop a new Bug Sprayer into their current position in the maze (and activate it) by pressing the space bar. Once activated, a Bug Sprayer counts down over multiple ticks until its timer expires – at this point it "triggers." Once a Bug Sprayer triggers it will spray Bug Spray into the maze (in a 'plus' pattern around the sprayer - for more details on this, see the Bug Sprayer section below). If either the avatar or a Zumi steps onto the same square as the Bug Spray (before it has dissipated), they will die. If a Zumi comes into contact with the Bug Spray and is exterminated, the Player earns points:

For destroying a Simple Zumi:	100 points
For destroying a Complex Zumi:	500 points

The Player also earns 1000 points (and special benefits) by picking up (i.e., moving onto the same square) a Goodie that a Zumi might drop when it dies.

Players also earn bonus points for completing a level quickly. Each level's maze data file specifies a bonus score (e.g., 1000 points). During each tick of the game, the bonus score is reduced by one point until the bonus reaches zero (the bonus never goes below zero). If and when the Player completes the current level, whatever bonus remains is added onto their score. This incentivizes the Player to complete each level as quickly as possible since the Player wants to maximize their score in the game.

The Player starts with three lives. The Player will lose a life under the following three scenarios:

1. The Player occupies the same square as a Zumi.
2. The Player occupies the same square as the Bug Spray from a Bug Sprayer before the spray/mist has dissipated.
3. The Player occupies the same square as a Destroyable Brick when their ability to walk through Bricks expires. (This special ability to walk through Destroyable Bricks is obtained by picking up one of the Goodies dropped by Zumi.)

When a Player dies in one of these three ways, the Player's number of remaining lives is decremented by 1. If the Player still has at least one life left, then the user is prompted to continue and given another chance by restarting the current maze/level from scratch. All of the Zumis that were initially on the level will again be alive and returned to their starting positions, the Player's avatar will be returned to their starting position, and the maze will revert back to its original state (with all Bricks intact). In addition, if the Exit was exposed in the maze prior to the avatar's death, then this too will be removed from the maze until such time that the Player exterminates all of the Zumi on the level. Then game play restarts. If the Player is killed and has no lives left, then the game is over. Pressing the *q* key lets you quit the game prematurely.

So how does a video game work?

Fundamentally, a video game is composed of a bunch of objects; in Bug Blast, those objects include the Player's avatar, Zumi (Simple and Complex), Goodies (e.g., Extra Life Goodies), Bug Sprayers, Bug Spray mist, Bricks (Permanent and Destroyable), and the Exit. Let's call these objects "actors," since each object is an actor in our video game. Each actor has its own x,y location in the maze, its own internal state (e.g., a Zumi knows its location, what direction it's moving, etc.) and its own special algorithms that control its actions in the game based on its own state and the state of the other objects in the world. In the case of the Player's avatar, the algorithm that controls the avatar actor object is the user's own brain and hand, and the keyboard! In the case of other actors (e.g., Zumi), each object has an internal autonomous algorithm and state that dictates how the object behaves in the game world.

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

During a given tick, the video game calls upon each object's behavioral algorithm and asks the object to perform its behavior. When asked to perform its behavior, each object's behavioral algorithm must decide what to do and then make a change to the object's state (e.g., move the object 1 square to the left), or change other objects' states (e.g., when a Zumi's algorithm is called by the game, it may determine that it has moved onto the same square as the Player's avatar and set the avatar's state to indicate it is dead). Typically the behavior exhibited by an object during a single tick is limited in order to ensure that the gameplay is smooth and that things don't move too quickly and confuse the player. For example, a Zumi will move just one square left/right/up/down, rather than moving two or more squares; if a Zumi moved 5 squares in a single tick, for example, the user would be confused because humans are used to seeing smooth movement in video games, not jerky shifts.

After the current tick is over and all actors have had a chance to adjust their state (and possibly adjust other actor's states), our game framework (that we provide) animates the actors onto the screen in their new configuration. So if a Zumi changed its location from 10,5 to 11,5 (moved one square right), then our game framework would erase the graphic of the Zumi from location 10,5 on the screen and draw the Zumi'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 smooth animation.

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

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

A video game can be broken into three different phases:

Initialization: The game World is initialized and prepared for play. This involves allocating one or more the actors (which are C++ objects) and placing them in the game world so that they will appear in the maze.

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

Cleanup: The Player has lost a life (but has more lives left) or the game is over. This phase frees all of the objects in the World (e.g., Simple Zumi, Bricks, Bug Sprayers, Bug Spray, the Player's avatar, etc.) since the level has ended. If game is not over (i.e., the user has more lives), then the game proceeds back to the *Initialization* step, where the maze is repopulated with new occupants and game play restarts at the current level.

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

```
while (The Player has lives left)
{
   Prompt_the_user_to_start_playing(); // "press a key to start"
   Initialize_the_game_world();        // you're going to write this

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

           // you're going to write code to do the following
         Ask_all_actors_to_do_something();
         Delete_any_dead_actors_from_the_world();

           // we write this code to handle the animation for you
         Animate_all_of_the_actors_to_the_screen();
         Sleep_for_50ms_to_give_the_user_time_to_react();
   }
```

```
     // the Player died - you're going to write this code
   Cleanup_all_game_world_objects();   // you're going to write this
}

Tell_the_user_the_game_is_over();      // we provide this
```

And here is what the `Ask_all_actors_to_do_something()` function might look like:

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

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

```
class SimpleZumi: public SomeOtherClass
{
   public:
       void doSomething()
       {
           If I'm on the same square as the Player's avatar,
                  Set the Player avatar state to dead
           Else if I'm can move in my current direction w/o hitting a Brick,
                  Move one square in my current direction
           Else if I'm about to run into a Brick
                  Pick a new direction, but don't move during this tick
       }
       ...
};
```

And here's what the Player's *doSomething* member function might look like:

```
class Player: public …
{
   public:
        void doSomething()
        {
             Try to get user input (if any is available)
               If the user pressed the UP key and that square is open then
                   Increase my y location by one
               If the user pressed the DOWN key and that square is open then
                   Decrease my y location by one
               ...
               If the user pressed the space bar to drop a sprayer, then
                   Add a new Bug Sprayer into the maze on the same square as
                     the Player.

               If the Player moves into the same square as a Zumi, then
                   Set the Player's state to dead
        }
        ...
};
```

What Do You Have to Do?

You must create a number of different classes to implement the Bug Blast 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 maze) and all of the actors/objects (Zumi, Bug Sprayers, Bug Spray, Goodies, and the Player's avatar) that are inside the maze.
2. You must create a class to represent the Player in the game.
3. You must create classes for Simple Zumi, Complex Zumi, Bug Sprayers, Bug Spray, Extra Life Goodies, Walk Through Walls Goodies, Increase Simultaneous Bug Sprayer Goodies, Bricks, and the Exit, as well as any additional base classes (e.g., a Zumi base class if you need one) that are required to 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 whole game world (the maze and all of its inhabitants such as Zumi, the Player, Bug Sprayers, Bricks, the Exit, Goodies, etc.). It is responsible for initializing the game world at the start of the game, asking all of the actors to do something during each tick of the game, and destroying all of the actors in the game world when the user loses a life or when actors disappear (e.g., a Zumi dies due to being exterminated by Bug Spray).

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

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

The code that you write must *never* call any of these three functions. Instead, our provided game framework will call these functions for you. Thus, you have to implement them correctly, but you won't ever call them yourself in your code.

When a new level starts (e.g., at the start of a game, 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 maze from a data file (we'll show you how below), and constructing a representation of the current level in your StudentWorld class, 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.

Once a new level has been loaded/initialized with a call to your *init()* method, our game framework will repeatedly call your *StudentWorld's move()* method, at a rate of roughly 20 times per second. Each time your *move()* method is called, it must run a single tick of the game. This means that it is responsible for asking each of the game actors (e.g., the Player's avatar, each Zumi, 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 (e.g., Bug Spray, a dead Zumi, etc.) that need to disappear during a given tick. For example, if a Complex Zumi steps onto the same square as a Bug Spray object, then its state should be set to dead, and the after all of the actors in the game get a chance to do something during the tick, the *move()* method should remove that Complex Zumi from the game world (by deleting its object and removing any reference to the object from the *StudentWorld's* data structures). Your *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 (e.g., by stepping onto the same square as a Bug Spray object, or having a Zumi move onto the same square as the Player). The *cleanup()* method is responsible for freeing all actors (e.g., all Zumi objects, all Brick objects, the Player object, the Exit object, all Goodie objects, etc.) that are currently in the game. This includes all actors created during the *init()* method or introduced during subsequent game play by the actors in the game (e.g., a Bug Sprayer that was added to the maze when the Player hit the spacebar during a tick) that have not yet been removed from the game.

You may add as many other private data members or public/private member functions to your *StudentWorld* class 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* class provides the following member functions for your use:

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

```
	void playSound(int soundID);
```

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

getLives() can be used to determine how many lives 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 your *StudentWorld* class (or your other classes) to increase the player's score upon successfully destroying a Zumi, picking up a Goodie of some sort, or completing a level (to give the Player their remaining level bonus). When your code calls this member function, you must specify how many points the player gets (e.g., 100 points for destroying a Simple Zumi). This means is that the game score is controlled by our *GameWorld* object – you must *not* maintain your own score data member in your own classes.

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

```
    Score: 00321000  Level: 05  Lives: 003  Bonus:   742
```

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

```
	KEY_PRESS_LEFT
	KEY_PRESS_RIGHT
	KEY_PRESS_UP
	KEY_PRESS_DOWN
	KEY_PRESS_SPACE
```

The *playSound()* method can be used to play a sound effect when an important event happens during the game (e.g., a Zumi avatar dies or a Bug Sprayer triggers and sprays Bug Spray). 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 poison mist is at the same location as a Zumi, then
	// then make a Zumi dying sound
	if (spray_x == zumi_x && spray_y == zumi_y)
		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 world.
2. Load the current maze details from a level data file (each level has its own data file that specifies what the maze looks like for that level, as well as all the game parameters for the level – such as what the bonus is for completing the level quickly).
    a. If the current level is 0 and the first level data file *level00.dat* can not be found, immediately return `GWSTATUS_NO_FIRST_LEVEL`.
    b. If the current level is not 0 and the level data file for the current level can not be found, this means the Player has completed all levels and won, so immediately return `GWSTATUS_PLAYER_WON`.
    c. If the level data file exists but is improperly formatted, immediately return `GWSTATUS_LEVEL_ERROR`.
3. Allocate and insert a valid Player object into the game world.
4. Allocate and insert any other Simple and Complex Zumi objects, Brick objects, or Exit objects into the game world, as required by the specification in the current level’s data file.
5. Return `GWSTATUS_CONTINUE_GAME`.

The constants `GWSTATUS_NO_FIRST_LEVEL`, `GWSTATUS_PLAYER_WON`, etc. are defined in *GameConstants.h*.

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. Briefly, here’s an example of using the Level class to load a level data file:

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

int someFunctionYouWriteToLoadALevel()
{
       string curLevel = "level03.dat";
       Level lev;
       Level::LoadResult result = lev.loadLevel(curLevel);

       if (result == Level::load_fail_file_not_found ||
           result == Level::load_fail_bad_format)
           return -1;              // something bad happened!

       // 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::perma_brick)
               cout << "There should be a perma brick at 10,7 in the maze\n":

        // you can also access the level options/parameters

        unsigned int bonusForThisLevel =
                      lev.getOptionValue(optionLevelBonus);
        cout << "The bonus for this level is: " << bonusForThisLevel;

        unsigned int howManyTicksBeforeAGoodieDisappears =
                      lev.getOptionValue(optionGoodieLifetimeInTicks);
        cout << "The player has " << howManyTicksBeforeAGoodieDisappears <<
               " ticks to pick up a goodie before it disappears.\n";
        … // etc
}
```

You can examine the *Level.h* file for a full list of options and 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 a *StudentWorld* object. It is **required** that you keep track of all of the actors (e.g., Simple Zumi, Bricks, Extra Life Goodies, the Exit, etc.) in a **single** STL collection like a list or vector. (To do so, we recommend using a container of pointers to game objects.) If you like, your *StudentWorld* class 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 Simple Zumi to move itself, ask a Bug Sprayer to count down and possibly release Bug Spray into the maze, give the Player a chance to move up, down, left or right, or drop a Bug Sprayer in the maze, etc.).
    a. If an actor does something that causes the Player to die, then the *move()* method should immediately return `GWSTATUS_PLAYER_DIED`.
    b. If the Player steps onto the same square as an Exit (after first clearing the level of all Zumi) and completes the current level, then the *move()* method should immediately:
        i. Increase the Player's score by the remaining bonus for the level.
        ii. Return a value of `GWSTATUS_FINISHED_LEVEL`.

2. It must then delete any actors that have died during this tick (e.g., a Simple Zumi that was destroyed by Bug Spray and so should be removed from the game world, or a Goodie that disappeared because the Player picked it up).
3. It must reduce the level's bonus points by one during each tick. So, for example, if level #0 has a starting bonus of 1000, then after the first tick, the bonus would drop to 999. On the second tick it would drop to 998, etc. This declining bonus value incentivizes the Player to complete the level as quickly as possible to get the biggest bonus score. The level bonus may not go below a value of zero.
4. It should expose/activate the Exit on the current level once the Player has exterminated all of the Zumi on the level (alternatively, your Exit class can do this instead of your *StudentWorld* class). This enables the Player to move over to the Exit and complete the level, advancing to the next level.
5. It must update the status text on the top of the screen with the latest information (e.g., the user's updated score, the remaining bonus score for the level, etc.).

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

```
GWSTATUS_PLAYER_DIED
GWSTATUS_CONTINUE_GAME
GWSTATUS_FINISHED_LEVEL
```

The first return value indicates that 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. If your *move()* method returns this value, then our framework will call your *cleanup()* method to destroy the level, then call your *init()* method to re-initialize the level from scratch. Our framework will then begin calling your *move()* method over and over, one per tick, to let the user play the level again.

The second return value indicates that the tick completed without the Player dying BUT the Player has not yet completed the current level. Therefore the game play should continue normally, so our framework will call *move()* again.

The final return value indicates that the Player has completed the current level (that is, has killed all of the Zumi on the level and stepped onto the same square as the Exit). 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…

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 text at screen top

   // The term "actors" refers to all Zumi, the Player, Bricks, Bug
   // Sprayers, Bug Spray, 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 the actor to do something (e.g. move)
            actor[i]->doSomething();

           if (playerDiedDuringThisTick())
                 return GWSTATUS_PLAYER_DIED;

           if (playerCompletedCurrentLevel())
           {
                 increaseScoreByBonus();
                 return GWSTATUS_FINISHED_LEVEL;
           }
        }
    }

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

    // Reduce the current bonus for the Level by one
    reduceLevelBonusByOne();

    // If the Player has killed all the Zumi on the level, then we
    // must expose the exit so the Player can advance to the next level
    if (playerHasKilledAllZumiOnTheLevel())
           exposeTheExitInTheMaze();  // make the exit Active

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

    if (playerCompletedCurrentLevel())
    {
           increaseScoreByBonus();
           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, release poison mist, etc.). Actors include the Player's avatar, Simple Zumi, Complex Zumi, the two types of Bricks, Bug Sprayers, Bug Spray, the three kinds of Goodies, and the Exit.

Your *move()* method must enumerate each active actor in the maze (i.e., held by your *StudentWorld* object) and ask it to do something by calling a member function in the actor's object named *doSomething()*. In each actor's *doSomething()* method, the object will have a chance to perform some activity based on the nature of the actor and its current state: e.g., a Complex Zumi might move toward the Player, the Player might drop a Bug Sprayer into the maze, a Bug Sprayer might trigger and release Bug Spray into the maze, etc.

It is possible that one actor (e.g., Bug Spray) may destroy another actor (e.g., a Simple Zumi) 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 any 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 Simple Zumi walks into some Bug Spray and dies, then it should be noted as dead, and at the end of the tick, its *pointer* should be removed from your StudentWorld's container of active objects, and the Simple Zumi object should be deleted (using the C++ delete expression) to free up room in 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();
   unsigned int bonus = getCurrentLevelBonus();
   int livesLeft = getNumberOfLivesThePlayerHasLeft();

   // Next, create a string from your statistics, of the form:
   // "Score: 00000100  Level: 09  Lives: 003  Bonus:   345"

   string s = someFunctionToFormatItemsNicely(score,level,lives,bonus);

   // 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:

1. Each field must be exactly as wide as shown in the example above:

    a. The score must be exactly 8 digits long, with leading zeros.
    b. The Level field must be 2 digits long, with leading zeros.
    c. The Lives field should be 3 digits long, with leading zeros.
    d. The Bonus field must be 6 characters wide, with spaces filling the the left-most positions and no leading zeros, e.g. "Bonus: _ _ _345" instead of "Bonus: 345", where _ in this sentence represents a space.

2. Each statistic must be separated from the previous statistic by exactly two spaces. For example, between the "00000100" 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.

cleanUp() Details

When your *cleanUp()* method is called by our game framework, it means that the Player lost a life (e.g., they got killed by a Zumi or by stepping onto Bug Spray). In this case, every actor in the entire maze (the Player and every Zumi, Goodies, Bug Sprayers, Bug Spray, Bricks, the Exit, etc.) must be deleted and removed from your StudentWorld's container of active objects, resulting in an empty maze. If the user has more lives left, our provided code will subsequently call your *init()* method to reload and repopulate the maze and the game will then continue with a brand new set of actors.

You must not call the *cleanUp()* method yourself when the Player dies. Instead, this method will be called by our code.

The Level Class and Level Data File

As mentioned, every level of Bug Blast has a different maze and different game parameters for that maze. These parameters and the maze layout are stored in data files, with the file "level00.dat" holding the details for the first level's maze, and "level01.dat" holding the details for the second level's maze, etc.

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

level00.dat:

```
probOfGoodieOverall=100
probOfExtraLifeGoodie=0
probOfWalkThruGoodie=100
probOfMoreSprayersGoodie=0
ticksPerSimpleZumiMove=5
ticksPerComplexZumiMove=10
goodieLifetimeInTicks=40
```

```
levelBonus=1000
walkThruLifetimeTicks=200
boostedSprayerLifetimeTicks=200
maxBoostedSprayers=6

###############
#    * @ *    #
# ####   #### #
# * s*   *c * #
# *  *   *  * #
#    #####    #
#             #
#             #
#             #
#             #
# #    *    # #
#*************#
#   *  #  *   #
#  *   e   *  #
###############
```

As you can see, the data file contains a bunch of parameters for the level as well as the layout of the maze. These will be described in detail below.

Maze Parameters

Here is a list of the parameters that must be specified in a level data file:

```
probOfGoodieOverall=###
```

This value specifies the probability of a Zumi dropping a goodie when it dies. So if the ### were 40, there would be a 40% chance that a Zumi would drop a Goodie when it dies.

```
probOfExtraLifeGoodie=###
```

Assuming a Zumi does decide to drop a Goodie when it dies, this value specifies the probability of that Goodie being an Extra Life Goodie (rather than some other Goodie). So if the ### were 30, there would be a 30% chance that when a Zumi drops a Goodie, it will be an Extra Life Goodie.

```
probOfWalkThruGoodie=###
```

Assuming a Zumi does decide to drop a Goodie when it dies, this value specifies the probability of that Goodie being a Walk Through Walls Goodie (rather than some other

Goodie). So if the ### were 50, then there would be a 50% chance that when a Zumi drops a Goodie, it will be a Walk Through Walls Goodie.

```
probOfMoreSprayersGoodie=###
```

Assuming a Zumi does decide to drop a Goodie when it dies, this value specifies the probability of that Goodie being an Increase Simultaneous Sprayers Goodie. So if the ### were 20, then there would be a 20% chance that when a Zumi drops a Goodie, it will be an Increase Simultaneous Sprayers Goodie.

Note: On each level, the above three parameters must add up to 100 (100%).

```
ticksPerSimpleZumiMove=##
```

This parameter specifies how frequently a Simple Zumi is allowed to move. If a Simple Zumi were allowed to make a move during each and every tick of the game (e.g., 20 times per second), the game would be really hard to play and the Player would die very quickly, since humans have trouble thinking at this speed (20x/second). As such, to make the game easier to play, you will often make your characters (e.g., the Zumi) move more slowly. For example, on level #0 (an easy level) you might want your Simple Zumi only move once every 5 ticks rather than once per tick, to let the player's brain have time to react and make a move. So, using this parameter, you can specify that Simple Zumi should sleep for X ticks between each move that it makes. For example, if the ## were 5, then each of your Simple Zumi would be required to sleep for 4 ticks, then move, then sleep another 4 ticks, then move, then sleep another 4 ticks, etc. You will likely lower this value for higher levels to make gameplay more difficult.

```
ticksPerComplexZumiMove=##
```

This parameter specifies how frequently a Complex Zumi is allowed to move and is similar to the previous parameter.

```
goodieLifetimeInTicks=##
```

All Goodies, after being dropped by a Zumi, have a limited lifetime before they disappear from the game. This parameter specifies how long a Goodie should hang around after it is dropped in the maze. For example, specifying a value of 40 would indicate that a Goodie should hang around for about 40 ticks (i.e., a couple of seconds) waiting for the Player to pick it up before disappearing from the game world.

```
levelBonus=####
```

This parameter specifies the maximum bonus points that the Player will receive for the level. This value incentivizes the Player to complete the level quickly, as the bonus value decreases by 1 during each tick of the game (until it reaches zero). The level bonus is reset to its full value every time a Player replays a level (e.g., because they die and have to try to complete the level again).

```
walkThruLifetimeTicks=####
```

When a Player picks up a Walk Through Walls Goodie, their avatar is temporarily allowed to walk onto squares in the game world that contain Destroyable Bricks or Bug Sprayers – these squares would normally be off-limits to both the Player's avatar and the Zumi bugs. This parameter specifies how many ticks the Player is allowed to walk through Destroyable Bricks before this temporary power expires. If a Player's avatar is on the same square as a Brick when this period ends, they will die.

```
boostedSprayerLifetimeTicks=####
maxBoostedSprayers=####
```

Normally, the Player's avatar may drop up to two Bug Sprayers at a time. A third Bug Sprayer may not be dropped until one or more of the previously-dropped bug-sprayers emits its poison mist and disappears from the screen. However, when the Player picks up an Increase Simultaneous Sprayers goodie, (dropped by a Zumi) they will temporarily be able to drop more than two sprayers at a time. The `maxBoostedSprayers` option specifies how many sprayers the Player may drop at a time (e.g., 4 instead of 2). The `boostedSprayerLifetimeTicks` option specifies how many ticks the Player will have this power. For example, they may be able to drop up to 4 sprayers at a time for up to 200 ticks.

The Maze Layout

You may specify the layout of each level's maze by editing the 15x15 grid in each level data file. Here's an example of such a level layout specification:

```
###############
#    * @ *    #
# ####   #### #
# * s*   *c * #
# *  *   *  * #
#    #####    #
#             #
#             #
#             #
#             #
# #    *    # #
#*************#
#   *  #  *   #
#  *   e   *  #
###############
```

A # character represents a Permanent Brick. The perimeter of each maze MUST be surrounded completely by Permanent Bricks.

A * character represents a Destroyable Brick that may be destroyed. The Player may also walk through these Destroyable Bricks if they pick up a Walk Through Walls Goodie.

The @ character specifies the starting location of the Player's avatar when he/she starts the level. The Player's avatar should also restart at this location if the Player dies and must replay the current level.

A **lower case s** character represents a Simple Zumi – this specifies that a Simple Zumi must start at this location in the maze when the Player starts (or replays) the current level.

A **lower case c** character represents a Complex Zumi – this specifies that a Complex Zumi must start at this location in the maze when the Player starts (or replays) the current level.

The **lower case e** character represents the level's exit. The level's exit will be made visible/active only when the Player has exterminated all of the Zumi on the level.

All **space** characters represent locations where the Player's avatar and Zumi may walk within the maze.

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 is an example of its use:

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

void someFunc()
{
	Level lev;

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

		Level::MazeEntry ge = lev.getContentsOf(5,10);  // x=5, y=10
		switch (ge)
		{
			case Level::empty:
				cout << "Location 5,10 is empty\n";
				break;
			case Level::simple_zumi:
```

```
                cout << "Location 5,10 starts with a Simple Zumi\n";
                break;
            case Level::complex_zumi:
                cout << "Location 5,10 starts with a Complex Zumi\n";
                break;
            case Level::player:
                cout << "Location 5,10 is where the player starts\n";
                break;
            case Level::exit:
                cout << "Location 5,10 is where the exit is\n";
                break;
            case Level::perma_brick:
                cout << "Location 5,10 holds a perma-brick\n";
                break;
            case Level::destroyable_brick:
                cout << "Location 5,10 holds a destroyable brick\n";
                break;
        }

        unsigned int val = lev.getOptionValue(optionProbOfGoodieOverall);
        cout << "The probability of a Zumi dropping a goodie is: ";
        cout << val << " percent. " << endl;

    }
}
```

The following are valid parameter values that may be passed to *getOptionValue()*:

```
optionProbOfGoodieOverall
optionProbOfExtraLifeGoodie
optionProbOfWalkThruGoodie
optionProbOfMoreSprayersGoodie
optionTicksPerSimpleZumiMove
optionTicksPerComplexZumiMove
optionGoodieLifetimeInTicks
optionLevelBonus
optionWalkThruLifetimeTicks
optionBoostedSprayerLifetimeTicks
optionMaxBoostedSprayers
```

These option values directly reflect the options found in your level data file.

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

If the result of a call like

```
    Level::LoadResult result = lev.loadLevel("level00.dat");
```

is indicating that the file is not found, and you can't figure out what folder to place the file in, you can supply a second argument that explicitly indicates the folder the file is in, e.g.,

```
    Level::LoadResult result = lev.loadLevel("level00.dat", "Z:/Proj3/Data");
```

When we test your program, we'll use a version of *Level.h* that ignores any such second argument and instead looks in the folder in which we'll place our test level data files.

You Have to Create the Classes for All Actors

The Bug Blast game has a number of different Game Objects, including:

- The Player’s avatar
- Simple Zumi bugs
- Complex Zumi bugs
- Bricks (permanent and destroyable)
- The Exit for the level
- Bug Sprayers
- Bug Spray (sprayed onto a square by a Bug Sprayer)
- Extra Life Goodies
- Walk Through Walls Goodies
- Increase Simultaneous Bug Sprayer Goodies

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 (Simple Zumi, the Player, Permanent Bricks, Bug Spray, etc.) has x,y coordinates. Many game objects have the ability to perform an action (e.g., move, or deploy Bug Spray) during each tick of the game. Many of them can be potentially attacked (e.g., a Zumi can be attacked by Bug Spray; the Player will die if they move onto the same square as a Zumi; a Destroyable Brick will be destroyed by Bug Spray, etc.) and could "die" during a tick. All of them need some attribute that indicates whether or 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 behavior 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 score on this project will depend upon your ability to intelligently create an set of classes that follow good object-oriented design principles. Your classes must never duplicate 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 the same text nearly identically in the following sections (e.g., in the Extra Life Goodie section and the Walk Through Walls Goodie section, or in the Simple Zumi and Complex Zumi sections) 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 Zumi: public Actor
{
public:
       …
};

class SimpleZumi: public Zumi
{
public:
       …
};
```

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

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

```
GraphObject(int imageID, int startX, int startY);
void setVisible(bool shouldIDisplay);
void getX() const;
void getY() const;
void moveTo(int x, int y);
```

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

GraphObject(int imageID, int startX, int startY) is the constructor for a new *GraphObject*. When you construct a new *GraphObject*, you must specify an image ID which indicates how the *GraphObject* should be displayed on screen (e.g., as a Simple Zumi, a Player, a Permanent Brick, 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. Notice that you pass the coordinates as x,y (i.e., column, row starting from bottom left, and *not* row, column.) One of the following IDs, found in *GameConstants.h*, must be passed in for the imageID value:

```
IID_PLAYER
IID_SIMPLE_ZUMI
IID_COMPLEX_ZUMI
IID_EXIT
```

```
IID_BUGSPRAYER
IID_BUGSPRAY
IID_EXTRA_LIFE_GOODIE
IID_WALK_THRU_GOODIE
IID_INCREASE_SIMULTANEOUS_SPRAYER_GOODIE
IID_PERMA_BRICK
IID_DESTROYABLE_BRICK
```

New *GraphObjects* start out invisible and are **NOT** displayed on the screen until the programmer calls the *setVisible()* method with a value of true for the parameter.

setVisible(bool shouldIDisplay) is used to tell our graphical system whether or not to display a particular *GraphObject* on the screen. If you call *setVisible(true)* on a *GraphObject*, then your object will be displayed on screen automatically by our framework (e.g., a Simple Zumi image will be drawn to the screen at the *GraphObject's* specified x,y coordinates if the object's image ID is IID_SIMPLE_ZUMI). If you call *setVisible(false)* then your GraphObject will not be displayed on the screen. When you create a new game object, always remember to call the *setVisible()* method with a value of true or the actor won't display on screen!

getX() and *getY()* 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 data members, but instead should use these functions and *moveTo()* from the GraphObject base class.

moveTo(int x, int y) is used to update the location of a GraphObject within the maze. For example, if a Simple Zumi'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 GameObject in the game 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).

The Player

Here are the requirements you must meet when implementing the Player class:

<u>What a Player Object Must Do When It Is Created</u>

When it is created:

1. The Player 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 default state:
    a. Is able to drop up to only two active Bug Sprayers at a time (they may drop additional Bug Sprayers once one or more of their Bug Sprayers have activated and deployed Bug Spray).
    b. Is not allowed to walk onto squares with Bricks of any type. In other words, a Player starts out in a state where it may not walk through Bricks, not even Destroyable Bricks.

In addition to any other initialization that you decide to do in your Player class, a Player object must make itself visible using the *GraphObject* class's *setVisible()* method, perhaps by calling *setVisible(true)*.

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:

1. The Player must check to see if it is currently alive. If not, then the Player's *doSomething()* method must return immediately – none of the following steps should be performed.
2. The Player must check to see if it has died due to being in the same square of the maze as a forbidden object (see details below) – if they have died, their *doSomething()* method must set the Player's state to dead, play a *SOUND_PLAYER_DIE* sound effect, and then return immediately. Here are the cases where the Player will die:
    a. If the Player is on the same square of the maze with a living Zumi of any kind, they will instantly die.
    b. If the Player is on the same square of the maze as a Brick and is not currently in a walk-through-Bricks state (by having recently picked up a Walk Through Walls Goodie), then they will instantly die.
    c. Note: The Player need not check to see if it is on the same square as Bug Spray – instead, each Bug Spray object will check for this (although your Player *may* check for this if it likes as well).
3. If the Player is currently in a state where they can walk through Bricks (because they recently picked up a Walk Through Walls Goodie), then the number of remaining ticks for their walk-through-Bricks state must be decremented by one (until it reaches zero and the Player ceases to be in this state).
4. If the Player is currently in a state where they can drop a larger number of simultaneous Bug Sprayers (because they recently picked up an Increase Simultaneous Bug Sprayers Goodie) then the number of remaining ticks for their increase-simultaneous-bug-sprayers state must be decremented by one (until it reaches zero and the Player ceases to be in this state). While the Player is in this state, they may drop more Bug Sprayers simultaneously than the two sprayers that are usually permitted. The number of Bug Sprayers that they may drop is

specified in the current level data file, and may be retrieved by using the *Level* class's *getOptionValue()* method with the *optionMaxBoostedSprayers* value.

5. Otherwise, the *doSomething()* method must check to see if the user pressed a key (the section below shows how to check this). If the user pressed a key:
    a. If the user asked to move either up, down, left or right by pressing a directional key and the Player is allowed to move from their current square to the selected adjacent target square, then the Player's location should be updated with the GraphObject's *moveTo()* method. A Player may move to an adjacent target square under the following circumstances:
        i. If the Player is not in a walk-through-Bricks state then they may move onto an adjacent target square that does not contain a Brick of any type.
        ii. If the Player is in a walk-through-Bricks state (because they recently picked up a Walk Through Walls Goodie), then they may move onto any adjacent square that does not contain a Permanent Brick.

       Note: The Player object is not responsible for checking to see if the target square where the Player moves contains a Zumi or Bug Spray. If the Player moves onto the same Square as a Zumi or Bug Spray during the current tick, this will be checked for either by each Zumi or Bug Spray object during the next call to its *doSomething()* method, or by the Player's *doSomething()* method when it is called during the next tick of the game.
    b. If the user asks to drop a Bug Sprayer by pressing the space bar and is allowed to drop a Bug Sprayer in the current location (they are not on the same square as a Brick or an existing Bug Sprayer AND the current number of active, non-exploded Bug Sprayers currently in the maze is less than the maximum number of active Bug Sprayers allowed), then a Bug Sprayer object must be added to the Player's current square in the maze. Note: In the Player's default state, they may drop up to only two Bug Sprayers at a time. They may not drop additional Bug Sprayers until at least one of the currently active sprayers has exploded and is no longer active. When the Player is in a state where they may drop an increased number of simultaneous Bug Sprayers (because they recently picked up an Increase Simultaneous Bug Sprayers Goodie), then they may drop more sprayers at the same time. The number of simultaneous sprayers allowed during such a state may be retrieved from the *Level* class using its *getOptionValue()* method with a parameter of *optionMaxBoostedSprayers*.

What the Player Must Do When It Dies

If a Player is killed for any reason (e.g., by being poisoned by Bug Spray, running into a Zumi, etc.), it must note its state as dead, and then use *GameWorld*'s *playSound()* method to play the SOUND_PLAYER_DIE sound effect.

Getting Input From the User

Since Bug Blast 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 type something and then hit the Enter key. This would make the game awkward to play, requiring the user to hit a directional key then hit Enter, then hit a direction 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[1]. 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:
            ... move Player to the left ...;
            break;
          case KEY_PRESS_RIGHT:
            ... move Player to the left ...;
            break;
          case KEY_PRESS_SPACE:
           ... drop a Bug Sprayer in the current square ...;
           break;

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

Bricks

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

Bricks are of two different types – permanent bricks and destroyable bricks. You may define either one, two, or three different classes to represent the types of bricks, as you see fit.

[1] 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 Brick Must Do When It Is Created

When it is created:

1. A Permanent Brick object must have an image ID of IID_PERMA_BRICK.
2. A Destroyable Brick object must have an image ID of IID_DESTROYABLE_BRICK.
3. Each Brick must be created 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 the x,y locations to pass for each call to the constructors for the Permanent and Destroyable Bricks.

In addition to any other initialization that you decide to do in your Brick class, a Brick object must make itself visible using the *GraphObject* class's *setVisible()* method, perhaps by calling *setVisible(true)*.

What a Brick Must Do During a Tick

A Brick must be given an opportunity to do something during every tick (in its *doSomething()* method). When given an opportunity to do something, the Brick should do nothing. After all, it's just a brick! (What did you think it would do?)

The Exit

Here are the requirements you must meet when implementing the Exit class (which represents an Exit doorway that the Player must step upon to complete the current level once all of the Zumi on the level have died).

What an Exit object Must Do When It Is Created

When it is created:

1. An Exit object must have an image ID of IID_EXIT.
2. An Exit must be created at the proper location as specified by the current level's data file.
3. The Exit must start out *invisible* when it is first created, since it is only revealed on-screen to the Player once the Player clears the level of all Zumi. At that point the Exit object must be made visible, and then if the Player steps on it, they will complete the level.

What the Exit object Must Do During a Tick

Each time the Exit object is asked to do something (during a tick), it should:

1. Determine if the Player is currently on the same square as the Exit object. If so, and if the Exit is visible/active (because the Player has cleared the level of Zumi), then the Exit must:
    a. Use *GameWorld*'s *playSound()* method to play the SOUND_FINISHED_LEVEL sound effect.
    b. Communicate to the *StudentWorld* object that the user has completed the current level.
    c. The *StudentWorld* object can grant the Player the bonus points for completing the level (if there are any bonus points left). If you like, you can have the Exit object grant the Player the bonus points instead.

What the Exit object Must Do When Revealed

If the Player exterminates all of the Zumi on the current level of the game, then your *StudentWorld's move()* method is responsible for activating the Exit object on the current level so it becomes visible and may be used by the Player to exit the current level and advance to the next level. Therefore, your Exit object should have some method that can be called that can be used to transition it from the invisible/inactive state it starts in to the visible/active state. When this method is called by your *StudentWorld's move()* method, it should:

1. Set the Exit object to be visible/active.
2. Use *GameWorld*'s *playSound()* method to play the SOUND_REVEAL_EXIT sound effect.

Note: This transition of the Exit from invisible/inactive to visible/active must happen only once, at the time when the current level is completely cleared of Zumi.

Bug Sprayers

You must create a class to represent a Bug Sprayer with a timer. When the timer reaches zero, then the Bug Sprayer emits poison mist (Bug Spray objects) into the maze, as described below. Here are the requirements you must meet when implementing the Bug Sprayer class.

What a Bug Sprayer object Must Do When It Is Created

When it is created:

1. The Bug Sprayer object must have an image ID of IID_BUGSPRAYER.
2. The Bug Sprayer must be created at the Player's x,y location when the Player hits the space bar to add a new sprayer into the maze.
3. When a Bug Sprayer is created, it must give itself a lifetime of 40 ticks, which is is the number of ticks of the game that the sprayer will last from the time of its creation until it self-destructs and releases Bug Spray into the maze.

In addition to any other initialization that you decide to do in your Bug Sprayer class, a Bug Sprayer object must make itself visible using the *GraphObject* class's *setVisible()* method, perhaps by calling *setVisible(true)*.

What the Bug Sprayer Object Must Do During a Tick

Each time the Bug Sprayer object is asked to do something (during a tick), it should:

1. 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. Decrement its lifetime by one.
3. If the Bug Sprayer's lifetime reaches zero, then it must activate:
    a. The Bug Sprayer adds a Bug Spray object (i.e., poison mist) onto its own square (bx,by).
    b. The Bug Sprayer adds Bug Spray objects at up to two squares in each of the up, down, left and right directions from itself in an attempt to form a plus (+) shape. The poison mist can not spread past any Brick, but it can settle on a Destroyable Brick and eventually dissolve it (as described below in the Bug Spray section). The spraying algorithm is something like the following (the pseudocode below is for the positive x direction; you must implement something with its effect for all four directions):

```
for i = 1 to 2 inclusive
{
      if square (bx+i,y) has no Permanent Brick
            Add a Bug Spray object onto that square
      if square (bx+i,y) has a Brick
            Immediately break out of the loop
}
```

    c. The Bug Sprayer must use *GameWorld*'s *playSound()* method to play the SOUND_SPRAY sound effect.
    d. The Bug Sprayer must then set its state to dead (so that it will be removed from the game by the *StudentWorld* object at the end of the current tick).

Bug Spray

You must create a class to represent deadly Bug Spray (aka poison mist). Here are the requirements you must meet when implementing the Bug Spray class.

What a Bug Spray object Must Do When It Is Created

When it is created:

1. The Bug Spray object must have an image ID of IID_BUGSPRAY.

2. A Bug Spray object must have its x,y location specified for it; a Bug Sprayer object can pass an x,y location to the Bug Spray constructor.
3. When a Bug Spray object is created, it must give itself a lifetime of 3 ticks, which is is the number of ticks of the game that the spray will last from the time of its creation until it dissipates.

In addition to any other initialization that you decide to do in your Bug Spray class, a Bug Spray object must make itself visible using the *GraphObject* class's *setVisible()* method, perhaps by calling *setVisible(true)*.

What the Bug Spray Object Must Do During a Tick

Each time the Bug Spray object is asked to do something (during a tick), it should:

1. 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. Decrement its lifetime by one.
3. If the Bug Spray's lifetime reaches zero, it must set its state to dead (so that it will be removed from the game by the *StudentWorld* object at the end of the current tick).
4. Otherwise, the Bug Spray must check to see if any other objects are at its location:
    a. If the Bug Spray is on the same square as a Destroyable Brick, then the Bug Spray will set that Brick's state to dead (so that it will be removed from the game by the *StudentWorld* object at the end of the current tick).
    b. If the Bug Spray is on the same square as the Player or a Simple or Complex Zumi, then the Bug Spray will set that object's state to dead (so that it will be removed from the game by the *StudentWorld* object at the end of the current tick). Note: When either the Player or a Zumi has its state set to dead, it will need to make a sound effect and potentially change the game's score.
    c. If the Bug Spray is on the same square as a Bug Sprayer (a different one than the one that released this Bug Spray object), then it will immediately trigger that Bug Sprayer (by setting its lifetime to zero), causing that Bug Sprayer to immediately release its Bug Spray into the maze as well, as described in the Bug Sprayer section. For all intents and purposes, this means that Bug Sprayers that are close enough together will chain-react – when one Bug Sprayer goes off, if another Bug Sprayer is within range of the Bug Spray of the first, it will activate, and so on.
    d. If the Bug Spray is on the same square as any object other than the ones mentioned above, it will have no impact on that object.

Extra Life Goodie

You must create a class to represent an Extra Life Goodie. 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 object Must Do When It Is Created

When it is created:

1. The Extra Life Goodie object must have an image ID of IID_EXTRA_LIFE_GOODIE.
2. An Extra Life Goodie object must have its x,y location specified for it; a Simple/Complex Zumi object can pass an x,y location to the Extra Life Goodie constructor (since a Zumi might drop this goodie when it dies).
3. When an Extra Life Goodie object is first created, it must give itself a "lifetime" of N ticks, where N must be determined from the *goodieLifetimeInTicks* value in the current level's data file. This is the number of ticks from the time of its creation that the goodie will hang around before it disappears from the game. The Extra Life Goodie constructor can ask the *StudentWorld* object for this value (which it presumably would determine by asking its *Level* object, passing the argument *optionGoodieLifetimeInTicks* to *getOptionValue()*).

In addition to any other initialization that you decide to do in your Extra Life Goodie class, an Extra Life Goodie object must make itself visible using the *GraphObject* class's *setVisible()* method, perhaps by calling *setVisible(true)*.

What the Extra Life Goodie Object Must Do During a Tick

Each time the Extra Life Goodie object is asked to do something (during a tick), it should:

1. 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. Decrement its lifetime by one.
3. If the Extra Life Goodie's lifetime reaches zero, it must set its state to dead (so that it will be removed from the game by the *StudentWorld* object at the end of the current tick).
4. Otherwise, the Extra Life Goodie must check to see if any other objects are on its square:
   a. If the Extra Life Goodie is on the same square as any object other than the Player, it will have no impact on that object.
   b. If the Extra Life Goodie is on the same square as the Player, then:
      i. The Extra Life Goodie must inform the *StudentWorld* object that the Player just got one extra life.
      ii. The Extra Life Goodie must inform the *StudentWorld* object that the user is to receive 1000 extra points. It can do this by using *GameWorld's increaseScore()* method.
      iii. The Extra Life Goodie must set its state to dead (so that it will be removed from the game by the *StudentWorld* object at the end of the current tick).

iv. The Extra Life Goodie must use *GameWorld*'s *playSound()* method to play the SOUND_GOT_GOODIE sound effect.

Walk Through Walls Goodie

You must create a class to represent a Walk Through Walls Goodie. When the Player picks up this Goodie (by moving onto the same square as it), it is allowed to temporarily walk through Destroyable Bricks. Here are the requirements you must meet when implementing the Walk Through Walls Goodie class.

What a Walk Through Walls Goodie object Must Do When It Is Created

When it is created:

1. The Walk Through Walls Goodie object must have an image ID of IID_WALK_THRU_GOODIE.
2. A Walk Through Walls Goodie object must have its x,y location specified for it; a Simple/Complex Zumi object can pass an x,y location to the Walk Through Walls Goodie constructor (since a Zumi might drop this goodie when it dies).
3. When a Walk Through Walls Goodie object is first created, it must give itself a "lifetime" of N ticks, where N must be determined from the *goodieLifetimeInTicks* value in the current level's data file. This is the number of ticks from the time of its creation that the goodie will hang around before it disappears from the game. The Walk Through Walls Goodie constructor can ask the *StudentWorld* object for this value (which it presumably would determine by asking its *Level* object, passing the argument *optionGoodieLifetimeInTicks* to *getOptionValue()*).

In addition to any other initialization that you decide to do in your Walk Through Walls Goodie class, a Walk Through Walls Goodie object must make itself visible using the *GraphObject* class's *setVisible()* method, perhaps by calling *setVisible(true)*.

What the Walk Through Walls Goodie Object Must Do During a Tick

Each time the Walk Through Walls Goodie object is asked to do something (during a tick), it should:

1. 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. Decrement its lifetime by one.
3. If the Walk Through Walls Goodie's lifetime reaches zero, it must set its state to dead (so that it will be removed from the game by the *StudentWorld* object at the end of the current tick).
4. Otherwise, the Walk Through Walls Goodie must check to see if any other objects are on its square:

a. If the Walk Through Walls Goodie is on the same square as any object other than the Player, it will have no impact on that object.

b. If the Walk Through Walls Goodie is on the same square as the Player, then:

    i. The Walk Through Walls Goodie must inform your Player object that the Player temporarily may walk through Destroyable Bricks, for a configurable number of ticks as specified by the *walkThruLifetimeTicks* value in the current level's data file.

    ii. The Walk Through Walls Goodie must inform the *StudentWorld* object that the user is to receive 1000 extra points. It can do this by using *GameWorld's increaseScore()* method.

    iii. The Walk Through Walls Goodie must set its state to dead (so that it will be removed from the game by the *StudentWorld* object at the end of the current tick).

    iv. The Walk Through Walls Goodie must use *GameWorld*'s *playSound()* method to play the SOUND_GOT_GOODIE sound effect.

Increase Simultaneous Sprayers Goodie

You must create a class to represent an Increase Simultaneous Sprayers Goodie. When the Player picks up this Goodie (by moving onto the same square as it), it temporarily allows the Player to drop a larger number of active Bug Sprayers at the same time instead of the usual two sprayers. Here are the requirements you must meet when implementing the Increase Simultaneous Sprayers Goodie class.

What an Increase Simultaneous Sprayers Goodie object Must Do When It Is Created

When it is created:

1. The Increase Simultaneous Sprayers Goodie object must have an image ID of IID_INCREASE_SIMULTANEOUS_SPRAYER_GOODIE.
2. An Increase Simultaneous Sprayers Goodie object must have its x,y location specified for it; a Simple/Complex Zumi object can pass an x,y location to the Increase Simultaneous Sprayers Goodie constructor (since a Zumi might drop this goodie when it dies).
3. When an Increase Simultaneous Sprayers Goodie object is first created, it must give itself a "lifetime" of N ticks, where N must be determined from the *goodieLifetimeInTicks* value in the current level's data file. This is the number of ticks from the time of its creation that the goodie will hang around before it disappears from the game. The Increase Simultaneous Sprayers Goodie constructor can ask the *StudentWorld* object for this value (which it presumably would determine by asking its *Level* object, passing the argument *optionGoodieLifetimeInTicks* to *getOptionValue()*).

In addition to any other initialization that you decide to do in your Increase Simultaneous Sprayers Goodie class, an Increase Simultaneous Sprayers Goodie object must make itself visible using the *GraphObject* class's *setVisible()* method, perhaps by calling *setVisible(true)*.

<u>What the Increase Simultaneous Sprayers Goodie Object Must Do During a Tick</u>

Each time the Increase Simultaneous Sprayers Goodie object is asked to do something (during a tick), it should:

1. 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. Decrement its lifetime by one.
3. If the Increase Simultaneous Sprayers Goodie's lifetime reaches zero, it must set its state to dead (so that it will be removed from the game by the *StudentWorld* object at the end of the current tick).
4. Otherwise, the Increase Simultaneous Sprayers Goodie must check to see if any other objects are on its square:
    a. If the Increase Simultaneous Sprayers Goodie is on the same square as any object other than the Player, it will have no impact on that object.
    b. If the Increase Simultaneous Sprayers Goodie is on the same square as the Player, then:
        i. The Increase Simultaneous Sprayers Goodie must inform your Player object that the Player temporarily may drop up to a larger number of Bug Sprayers at a time, for a configurable number of ticks as specified by the *boostedSprayerLifetimeTicks* value in the current level's data file. The temporary maximum number of simultaneous sprayers allowed is specified by the *maxBoosterSprayers* value in that file.
        ii. The Increase Simultaneous Sprayers Goodie must inform the *StudentWorld* object that the user is to receive 1000 extra points. It can do this by using *GameWorld's increaseScore()* method.
        iii. The Increase Simultaneous Sprayers Goodie must set its state to dead (so that it will be removed from the game by the *StudentWorld* object at the end of the current tick).
        iv. The Increase Simultaneous Sprayers Goodie must use *GameWorld*'s *playSound()* method to play the SOUND_GOT_GOODIE sound effect.

Simple Zumi

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

What a Simple Zumi object Must Do When It Is Created

When it is created:

1. A Simple Zumi must have an image ID of IID_SIMPLE_ZUMI.
2. Each Simple Zumi must be created at the proper location as specified by the current level's data file.
3. The Simple Zumi must initially select a random direction for movement – up, down, left or right, chosen with equal probability. Call this its currentDirection.
4. Since unlike the Player, a Simple Zumi doesn't necessarily get to take an action during every tick of the game, the Simple Zumi constructor must have a parameter that indicates how often the Simple Zumi is allowed to take an action (once every N ticks) when asked to do something. (This is to make the game easier to play, since if Zumi moved once every tick, they'd move much faster than the typical player can think and hit the keys on the keyboard). The value of N is specified in each level's data file, and may be different for different levels. Your *StudentWorld* object should get this value of N from its current Level by calling *Level*'s *getOptionValue()* method with *optionTicksPerSimpleZumiMove* as its argument. It can then pass the value to the Simple Zumi constructor.

In addition to any other initialization that you decide to do in your Simple Zumi class, a Simple Zumi object must make itself visible using the *GraphObject* class's *setVisible()* method, perhaps by calling *setVisible(true)*.

What a Simple Zumi Must Do During a Tick

Each Simple Zumi must be given an opportunity to do something during every tick. When given an opportunity to do something, the Simple Zumi must do the following:

1. The object 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. Next, the Simple Zumi must check to see if it is on the same square as the Player's avatar. If so, it should inform the Player that the Player has died. (See the Player description for details).
3. The Simple Zumi must then decide if it should move during the current tick. If the current level's data file specified that the value of *ticksPerSimpleZumiMove* were 10, for example, then the Simple Zumi would be allowed to move once every 10 ticks of gameplay (9 idle, 1 move, 9 idle, 1 move, etc).
4. Assuming a Simple Zumi has decided to move during the current tick, it must use the following algorithm:
    a. The Simple Zumi will try to move one square in its currentDirection if the destination square is not occupied by a Bug Sprayer or some type of Brick.
    b. If the destination square in the currentDirection contains a Bug Sprayer or some type of Brick, then the Simple Zumi must not move, but instead must randomly pick a new currentDirection (up, down, left, or right,

chosen with equal probability) and then immediately return. Note: The Simple Zumi must not try to move in this direction during this tick; it must wait *ticksPerSimpleZumiMove* ticks before it can move again.

What a Simple Zumi Must Do When It Is Killed by Bug Spray

A Simple Zumi will die if it occupies the same square as Bug Spray in the maze. Simple Zumi should therefore have a member function called something like *damage* that can be called by a Bug Spray object to tell the Simple Zumi that it has been exterminated due to being on the same square as the deadly Bug Spray.

If a Simple Zumi has been notified that it has been killed by Bug Spray, it must do the following:

1. Set its own state to dead/inactive, so that it can be removed from the game world at the end of the current tick.
2. Use *GameWorld*'s *playSound()* method to play the SOUND_ENEMY_DIE effect.
3. Use *GameWorld's increaseScore()* method to increase the Player's score by 100 points.
4. Determine the chance that it will drop a Goodie into the maze at its current position. The probability that a Zumi (of any type) will drop a Goodie is specified in each level's data file. A Zumi can ask the *StudentWorld* object for this probability (which it presumably would determine by asking its *Level* object, passing the argument *optionProbOfGoodieOverall* to *getOptionValue()*).
5. Generate a random number between 0 and 99 inclusive, and if this number is less than the probability determined in the previous step, then the Simple Zumi will drop a Goodie at its current location.
6. You must then decide what type of Goodie to drop, again consulting the *StudentWorld* object for the probabilities of dropping an Extra Life Goodie, a Walk Through Walls Goodie, or an Increase Simultaneous Sprayers Goodie.
7. Add the new Goodie of the proper type into the maze at the Simple Zumi's current location.

Complex Zumi

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

What a Complex Zumi object Must Do When It Is Created

When it is created:

1. A Complex Zumi must have an image ID of IID_COMPLEX_ZUMI.
2. Each Complex Zumi must be created at the proper location as specified by the current level's data file.
3. The Complex Zumi must initially select a random direction for movement – up, down, left or right, chosen with equal probability. Call this its currentDirection.
4. Since unlike the Player, a Complex Zumi doesn't necessarily get to take an action during every tick of the game, the Complex Zumi constructor must have a parameter that indicates how often the Complex Zumi is allowed to take an action (once every N ticks) when asked to do something. (This is to make the game easier to play, since if Zumi moved once every tick, they'd move much faster than the typical player can think and hit the keys on the keyboard). The value of N is specified in each level's data file, and may be different for different levels. Your *StudentWorld* object should get this value of N from its current Level by calling *Level*'s *getOptionValue()* method with *optionTicksPerComplexZumiMove* as its argument. It can then pass the value to the Complex Zumi constructor.

In addition to any other initialization that you decide to do in your Complex Zumi class, a Complex Zumi object must make itself visible using the *GraphObject* class's *setVisible()* method, perhaps by calling *setVisible(true)*.

What a Complex Zumi Must Do During a Tick

Each Complex Zumi must be given an opportunity to do something during every tick. When given an opportunity to do something, the Complex Zumi must do the following:

1. The object 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. Next, the Complex Zumi must check to see if it is on the same square as the Player's avatar. If so, it should inform the Player that the Player has died.
3. The Complex Zumi must then decide if it should move during the current tick. If the current level's data file specified that the value of *ticksPerComplexZumiMove* were 10, for example, then the Complex Zumi would be allowed to move once every 10 ticks of gameplay (9 idle, 1 move, 9 idle, 1 move, etc).
4. Assuming a Complex Zumi has decided to move during the current tick, it must use the following algorithm:
    a. The Complex Zumi must compute the horizontal distance in maze squares between itself and the Player's avatar, as well as the vertical distance in maze squares between itself and the Player's avatar, e.g., horizDist = absolute_value(playerX-complexZumiX). This computation is not affected by the presence or absence of obstacles like Bricks.
    b. The Complex Zumi must determine the value of *optionComplexZumiSearchDistance* from the current *Level's* options. Let's call this value smellDistance. This value of smellDistance indicates how far the Complex Zumi can smell on the current level.

c. If the Complex Zumi is within smellDistance horizontal squares *and* within smellDistance vertical squares of the Player’s avatar, it can smell the Player. In this case, it will try to move in the direction (up, down, left or right) that will enable it to reach the Player most quickly. To do so, you must perform a breadth-first search of the maze, starting at the Complex Zumi's x,y location, looking for the Player’s avatar.

    i. If the breadth-first search indicates that the Complex Zumi can somehow reach the Player (e.g., the Complex Zumi is not trapped in a pen of Bricks and Bug Sprayers), it must move a single square in the direction indicated by the breadth-first search. Notes:

        1. The direction that the Complex Zumi must move will not always be directly toward the Player – the Complex Zumi might have to initially move away from the Player if, for example, the shortest path requires it to work its way around a Brick to get to the Player. For more information on breadth-first search, please see slide 32 in Professor Nachenberg’s lecture5-updated.pptx (entitled *Solving a Maze With a Queue*). Hint: The breadth-first search in that slide deck simply determines if a path exists from a start point to an end point. You’ll have to modify the algorithm to determine the direction of the first step of such a path.
        2. The shortest path may include squares that would be outside of the Complex Zumi's smell range of the Player's current position. That's fine; the smell range is used each tick only to decide whether or not to conduct a search during that tick, but plays no role in the search itself.
        3. If there is more than one path with the shortest distance, you may use any the first step of any one of those paths (e.g., that of the first such path your algorithm happens to discover).

    ii. If the breadth-first search indicates that the Complex Zumi cannot reach the Player (e.g., the Comple Zumi is trapped in a pen of Bricks and Bug Sprayers so that no possible move could help it eventually reach the Player’s avatar), then the Complex Zumi must behave like a Simple Zumi, that is:

        1. The Complex Zumi will try to move one square in its currentDirection if the destination square is not occupied by a Bug Sprayer or some type of Brick.
        2. If the destination square in the currentDirection contains a Bug Sprayer or some type of Brick, then the Complex Zumi must not move, but instead must randomly pick a new currentDirection (up, down, left, or right, chosen with equal probability) and then immediately return. Note: The Complex Zumi must not try to move in this direction during this tick; it must wait *ticksPerComplexZumiMove* ticks before it can move again.

d. If *either* the horizontal distance or vertical distance is greater than smellDistance, then the Complex Zumi will behave like a Simple Zumi in the manner of 4c(ii)(1) and 4c(ii)(2) above.

What a Complex Zumi Must Do When It Is Killed by Bug Spray

A Complex Zumi will die if it occupies the same square as Bug Spray in the maze. Complex Zumi should therefore have a member function called something like *damage* that can be called by a Bug Spray object to tell the Complex Zumi that it has been exterminated due to being on the same square as the deadly Bug Spray.

If a Complex Zumi has been notified that it has been killed by Bug Spray, it must do the following:

1. Set its own state to dead/inactive, so that it can be removed from the game world at the end of the current tick.
2. Use *GameWorld*'s *playSound()* method to play the SOUND_ENEMY_DIE effect.
3. Use *GameWorld's increaseScore()* method to increase the Player's score by 500 points.
4. Determine the chance that it will drop a Goodie into the maze at its current position. The probability that a Zumi (of any type) will drop a Goodie is specified in each level's data file. A Zumi can ask the *StudentWorld* object for this probability (which it presumably would determine by asking its *Level* object, passing the argument *optionProbOfGoodieOverall* to *getOptionValue()*).
5. Generate a random number between 0 and 99 inclusive, and if this number is less than the probability determined in the previous step, then the Complex Zumi will drop a Goodie at its current location.
6. You must then decide what type of Goodie to drop, again consulting the *StudentWorld* object for the probabilities of dropping an Extra Life Goodie, a Walk Through Walls Goodie, or an Increase Simultaneous Sprayers Goodie.
7. Add the new Goodie of the proper type into the maze at the Simple Zumi's current location.

How to Tell Who's Who

Depending on how you design your classes, you may find that you need to determine what type of object one of your pointers points to. For example, suppose the Player is directed to move right and needs to decide whether that's prevented by a Brick:

```
Actor* ap = getAnActorAtTheProposedLocation(…);
if (ap != nullptr)
{
    Determine if ap points to a Brick
    ...
}
```

In the code above, the Player calls a function *getAnActorAtTheProposedLocation* that you might write, which returns a pointer to an actor, if any, that occupies the destination square. The Player is allowed to occupy the same square as a Goodie or a Bug Sprayer, for example, but not a Brick in most circumstances (it's allowed when the Player can walk through bricks and the Brick is not permanent). But how can we determine whether *ap* points to some kind of Brick or not? Here's a possible way:

```
Actor* ap = getAnActorAtTheProposedLocation(…);
if (ap != nullptr)
{
    Brick* bp = dynamic_cast<Brick*>(ap);
    if (bp != nullptr)
    {
        cerr << "bp points to some kind of Brick" << endl;
        ...
    }
}
```

A C++ *dynamic_cast* expression can be used to determine whether the object pointed to by a pointer of a general type may also be pointed to by an pointer of a more specific type (e.g., whether a Brick pointed to by an Actor pointer can in fact be pointed to by a Brick pointer, or whether a Complex Zumi pointed to by an Actor pointer can in fact be pointed to by a Zumi pointer). In the example above, if *ap* pointed to a Brick object, then dynamic casting *ap* to a Brick pointer (the target type in the angle brackets) succeeds, and the returned pointer that is used to initialize *bp* will point to that Brick object. If *ap* pointed to a Complex Zumi object, then since a Brick pointer could not point to such an object, the dynamic cast yields a null pointer.

Here's another example in a different problem domain:

```
class Person { ... };
class Faculty : public Person { ... };
class Student : public Person { ... };
class GradStudent : public Student { ... };
class Undergrad : public Student { ... };
class ExchangeStudent : public Undergrad { ... };

Person* p = new Undergrad(...);         // The object is an Undergrad
...
Faculty* f = dynamic_cast<Faculty*>(p); // f is nullptr; an Undergrad is
                                        // not a kind of Faculty
Student* s = dynamic_cast<Student*>(p); // s is not nullptr; an Undergrad
                                        // is a kind of Student
Undergrad* u = dynamic_cast<Undergrad*>(p);  // u is not nullptr; an
                                        // Undergrad is an Undergrad
ExchangeStudent* e = dynamic_cast<ExchangeStudent*>(p);
                                        // e is nullptr; an Undergrad is
                                        // not a kind of ExchangeStudent
```

Note: dynamic_cast works only for classes with at least one virtual function. A base class should always have a virtual destructor, so that's an easy requirement to meet.

Stylistic note: Uses of dynamic_cast should be rare. In most cases, when you have a base pointer and you want to do different things depending on which kind of derived object it points to, you'd just call a virtual function declared in the base class and implemented in different ways in the derived classes.

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 that!

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 that.

Building the Game

To build the game, follow these steps:

For Windows

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

If you build and run your program from within Visual Studio, your level data files and the sound files (ending in *.wav*) must 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 level data files and the sound files must be in the same folder as the executable.

For Mac OS X

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

If you build and run your program from within Xcode, your level data files and the sound files (ending in *.wav*) must be in the directory yourProjectDir*/DerivedData/*yourProjectName*/BuildProducts/Debug* (for example, */Users/fred/BugBlast/DerivedData/BugBlast/Build/Products/Debug*). On the other hand, if you launch the program by double-clicking on the executable file, the level data files and the sound files must 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 Bug Blast 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 objects (e.g., the Player, Simple Zumis, Complex Zumis, Goodies, Bricks, etc.):
   i. It must have a constructor.
   ii. It must be derived from our GraphObject class.
   iii. You may add other public/private member functions and private member variables to this base class, as you see fit.

2. A Brick class, derived in some way from the base class described in 1 above:
    i. It must have a constructor that initializes a Brick.
    ii. It must have an Image ID of IID_PERMA_BRICK or IID_DESTROYABLE_BRICK.
    iii. You may add any set of public/private member functions and private member variables to your Brick 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 1 just above (either directly derived from the base class, or derived from some other class that is somehow derived from the base class):
    i. It must have a constructor that initializes the Player – see the Player section for more details on where to initialize the Player.
    ii. It must have an Image ID of IID_PLAYER.
    iii. It (or its base class) must make itself visible via a call to *setVisible(true);*
    iv. It must have a limited version of a *doSomething()* member function that lets the user pick a direction by hitting a directional key. If the Player hits a directional key during the current tick and the target square does not contain a Brick, it updates the Player's location to the target square. All this *doSomething()* member function has to do is properly adjust the Player's X,Y coordinates and our graphics system will automatically animate its movement around the maze!
    v. You may add any public/private member functions and private data member 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.
    i. Add any private data members to this class required to keep track of Bricks as well as the Player object. You may ignore all other items in the maze such as Zumi, the Exit, etc. for part #1.
    ii. Implement a constructor for this class that initializes your data members.
    iii. Implement a destructor for this class that frees any remaining dynamically allocated data that has not yet been freed at the time the class is destroyed.
    iv. Implement the *init()* method in this class. It must create the Player and insert it into the maze at the right starting location (see the Level class section of this document for details on the starting location). It must also create all of the Bricks and add them to the maze as specified in the current Level's data file.
    v. Implement the *move()* method in your *StudentWorld* class. During each tick, it must ask your Player to do something. Your *move()* method need not check to see if the Player has died or not; you

may assume at this point that the Player cannot die. Nor need your *move()* method deal with any Zumi or other actors (e.g., Goodies or Bug Sprayers) at this point – just the Player.

vi. 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 Bricks and the Player). Note: Your StudentWorld class must have both a destructor and the *cleanUp()* method even though they likely do the same thing.

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

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 maze with the Player in its proper starting position. If your classes work properly, you should be able to move the Player around the maze using the directional keys, without the Player walking through any bricks.

Your Part #1 solution may actually 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 any Zumi, Bug Sprayers, Bug Spray, Goodies or the Exit (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 additonal functionality, make sure that your Player, Brick, and StudentWorld classes still work properly and that your program still builds and meets the requirements stated above for Part #1!**

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

What to Turn In For Part #1

You must turn in your source code for the simple version of your game, which **must build without errors** under either Visual Studio or Xcode. You do **not** have to get it to run under more than one compiler. You will turn in a zip file containing nothing more than these four files:

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

You will not be turning in any other files – we'll test your code with our versions of the other .cpp and .h files. Therefore, your solution must NOT modify any of our files or you will receive zero credit! 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 Bug Blast game, which adheres exactly to the functional specification provided in this document.

What to Turn In For Part #2

You must turn in the following files, and ONLY the following files. If you name your source files with other names, you will be docked points, so be careful!

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

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

You must turn in a report that contains the following:

1. A high-level description of each of your public member functions in each of your classes, and why you chose to define each member function in its host class; also explain why (or why not) you decided to make each function virtual or pure virtual. For example, "I chose to define a pure virtual version of the sneeze() function in my base Actor class because all actors in Bug Blast are able to sneeze, and each type of actor sneezes in a different way."

2. A list of all functionality that you failed to finish as well as known bugs in your classes, e.g., “I wasn’t able to implement the Exit class.” or “My Complex Zumi doesn’t work correctly yet so I just treat it like a Simple Zumi right now.”
3. A list of other design decisions and assumptions you made, e.g., “It was unspecified what to do in situation X, and this is what I decided to do.”
4. A description of how you tested each of your classes (1-2 paragraphs per class)

FAQ

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

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

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

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

GOOD LUCK!