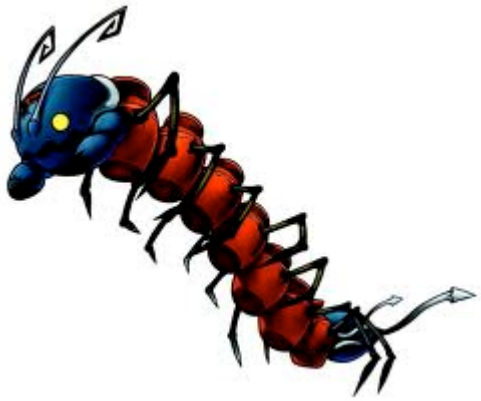
Project #3
Scentipede
DRAFT SPEC 02-17-06:30

Some details of this draft spec may change in the next day or so, but you should get started immediately; further changes will be minor.

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



Time due:

Part 1: 9 PM, Sunday, February 26
Part 2: 9 PM, Thursday, March 1

Note: We understand that Spiders and Scorpions are not Insects – however, for the purposes of this specification we will refer to all bugs, including Spiders and Scorpions as Insects. Please do not tell us that Spiders and Scorpions are not Insects. We know this.

Table of Contents

Introduction

NachenGames corporate spies have learned that SmallSoft is planning to release a new game, called Scentipede, and would like you to program an exact copy so NachenGames can beat SmallSoft to market. To help you, NachenGames corporate spies have managed to steal a prototype Scentipede 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.)

Scentipede is a simplified version of the original Atari Centipede game. In Scentipede, the Player tends to a garden full of Mushrooms and nasty insects. The goal of Scentipede is kill as many centipedes, Spiders, Fleas and Scorpions as possible. If you happen to also destroy some of the garden's Mushrooms in the process, well, that can't really be helped, can it?

Upon starting a new game, the Player's yellow triangular "Ship" is placed in the lower, middle section of the screen, at the bottom of the Mushroom garden. The Player can use the arrow keys to move their Ship left, right, up or down on the screen in order to best position it to battle the invading insects. The Player can also press the space key to shoot a jet of Water Droplets across the garden and kill insects or knock down Mushrooms. All insects are killed by a single shot of water; however, Mushrooms require four shots before they are knocked down.

There are two types of Mushrooms in the garden: normal edible Mushrooms are shown in the cyan (bluish) color, whereas psychedelic, poison Mushrooms are colored yellow.

Scentipedes, shown in white, simply scamper back and forth across the garden in a regular pattern, avoiding Mushrooms when possible. Each Scentipede is composed of multiple independent segments that can survive on their own – therefore if a segment of the Scentipede is shot, it will die, but the remaining Scentipede Segments will continue to live normally. While centipedes ignore regular Mushrooms, they hate Poison Mushrooms and will become sickened temporarily when they run into such a Mushroom, rapidly dropping down to the bottom of the garden before continuing their perpetual march. Centipedes are poisonous and will kill the Player if they ever come into contact with the Player Ship (i.e., move onto the same square as the Ship).

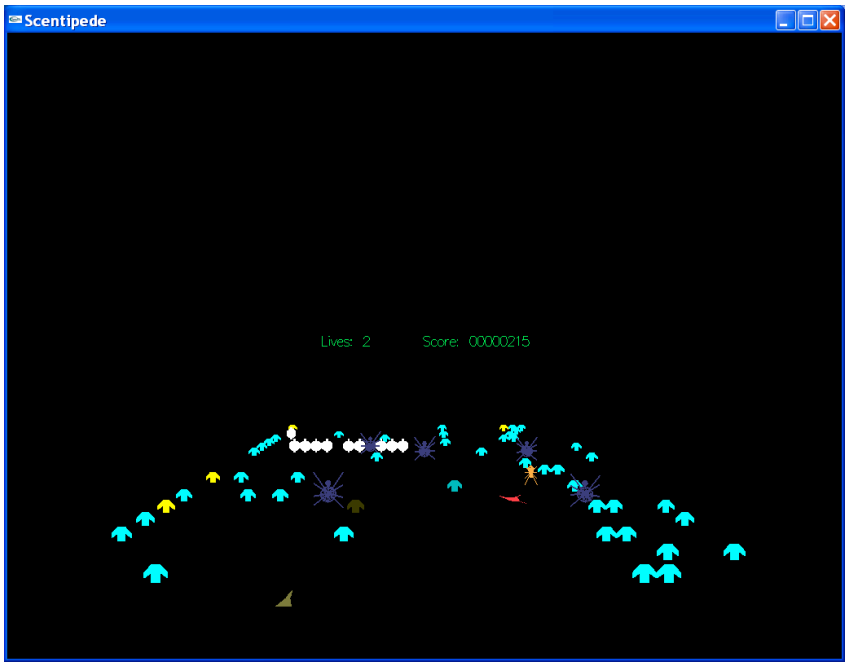
Spiders, shown in purple, like to zigzag horizontally through the garden, eating Mushrooms (both regular and poisonous ones). Spiders will also kill the Player Ship if they move onto the same square as it.

Fleas, drawn in orange, like to run vertically from the top of the garden to the bottom, randomly dropping new Mushrooms as they make their way down through the garden. As

with Spiders and Centipedes, Fleas also kill the Player Ship if the two happen to occupy the same square in the garden.

Finally, red-colored Scorpions occasionally run horizontally through the garden. While Scorpions don't eat Mushrooms, their poison tail occasionally comes into contact with them as the Scorpion moves across the garden, causing such Mushrooms to become poisonous. Scorpions also kill the Player's Ship if the two happen to occupy the same square in the garden.

Here is an example of what the Scentipede display looks like:



Game Details

The Player starts out a new game with 3 lives and can continue to play until all of his/her lives have been exhausted. There are no "levels" in Scentipede – the game just keeps going and going until the Player dies three times and the game is over.

The Scentipede garden is exactly 30 squares wide by 20 squares high. The bottom-leftmost square is numbered 0,0, with the upper-rightmost square is numbered 29,19. You can look in our provided file, GameConstants.h for constants like the garden's width/height.

When a new game begins (or when the game goes on after the Player loses a life), the board is initialized as follows:

1. The screen is populated with randomly-placed normal Mushrooms (not poison Mushrooms). For details on how many Mushrooms to drop and where they may be dropped, read on.
2. The Player Ship is placed in the garden at location x=15, y=0.
3. At the start of the game or when continuing a game after losing a life, NO insects initially start on the screen.

Once a new garden has been prepared and the Player and all of the Mushrooms are in their proper positions, the game play begins. Game play is divided into "ticks," and there are dozens of ticks per second (to provide smooth animation and game play). During each tick, the following occurs:

1. One or more new insects may be added to the garden (see sections below for details on this).
2. The Player has an opportunity to either move their Ship exactly one square horizontally or vertically, or fire a squirt of water.
3. Each of the other objects in the garden, if any, has an opportunity to do something. For example, insects can move one square (left, right or possibly diagonally) according to their built-in movement algorithms (described in the Insect Details section). Each insect may also perform one other action; for example, a Scorpion might also poison a Mushroom it's standing on.

During game play, the user controls the direction of their Player Ship by pushing the directional keys (up '8', down '4', left '4' and right '6' on the numeric keypad). The Player is blocked from moving onto a Mushroom, normal or poisonous. Instead, they may only move onto open squares in the garden. The Player Ship may move anywhere in the bottom 4 rows of the garden (between 0,0 and 29,3 of the garden, inclusive) except, as mentioned, on top of Mushrooms. It must not be allowed to move out of this rectangular area.

The Player may shoot the water cannon by pressing the space key. The water cannon shoots up from the square above the Player and continues until it either hits a Mushroom or insect, or until it reaches the top of the garden (y=19). The Player earns points for shooting the water cannon and hitting either a Mushroom or an insect:

For shooting a Mushroom: 1 point
For shooting a centipede segment: 10 points
For shooting a Spider: 20 points
For shooting a Scorpion: 25 points
For shooting a Flea: 50 points

During each tick, each of the insects and the Player decides how to move. Once each character in the game has updated their coordinates, our provided code (you don't need to worry how this works, unless of course, you are curious) will animate all of the game's actors onto the computer screen for the user to see. Then the next tick begins, each of the

characters decides how to move and updates their coordinates, and then they are again animated to the screen, etc.

If the Player Ship is destroyed by running into an insect, or if an insect runs into the Player Ship, the Player's number of remaining lives is decremented by 1 (of their initial 3 lives). If they still have at least one life left, then the user is prompted to continue and given another chance to play with a fresh garden. Once the user dismisses the prompt, all of the old insects and Mushrooms disappear from the garden. The garden is then re-populated (as described above) with Mushrooms, the Player Ship is returned to its "home" position in the garden, and game play continues. If the Player Ship is killed by an insect and has no lives left, then the game is over.

Each type of insects moves according to a different algorithm. For example, Fleas simply drop from the top of the garden until they reach the bottom of the garden (where they disappear), whereas Spiders zigzag up and down as they move horizontally across the garden. A full listing of all insect behaviors is provided in the sections below.

So how does a video game work?

Fundamentally, a video game is composed of a bunch of objects, like Scentipedes, Fleas, Mushrooms, Water Droplets and even the Player Ship. Each object has its own x,y location, its own internal state (e.g., for an insect, what direction it's moving, its color, etc.) and its own special algorithm to 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 Ship, the algorithm that controls the Player Ship object is the user's own brain, their hand, and the keyboard!

Game time is divided into "ticks." A tick is a unit of time, for example, $1/20^{th}$ of a second. During a given tick, every actor (e.g., Scentipede, insects, and the Player Ship) in the game has an opportunity to do something (e.g., move to an adjacent square in the garden, shoot a water cannon, die, disappear off the edge of the screen, etc.). During each tick, the Player may hit a key to control the Player Ship (e.g., to move left, right, up, or down, or to fire). As each actor runs its algorithm during each tick, in addition to altering the actor's own state (e.g., its location), the actor may also affect other actors within the game. For example, a Scorpion might run into a Regular Mushroom and convert it into a Poisonous Mushroom. So actors not only can affect their own state but the state of the world and of other actors within it.

After the current tick is over and all actors have had a chance to adjust their state, they are then animated onto the screen in their latest configuration.

Then, the next tick occurs, each object again gets a chance to do something, etc.

Assuming the ticks are quick enough (a small fraction of a second), and the actions performed by the objects are subtle enough (i.e., a Scentipede 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 moves fluidly.

A video game can be broken into three different phases:

Initialization: The game world (e.g., the garden with all of its Mushrooms, etc.) is initialized and prepared for play. This involves allocating all of the actors (which are C++ objects) and placing them in the game world.

Game play: During each tick of the series of ticks that constitute game play, 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 have died will be removed from the game.

Cleanup: When the Player loses a life, all of the objects in the garden are freed. If game play is not over (because the user has more lives), then the game proceeds back to the Initialization step, a new game world is created (with new occupants), and then game play continues.

Here is what the main logic of a video game looks like, in pseudocode (we have provided this part of the program for you in GameController.h):

```
void videogame()
{
   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 code

      While (The Player hasn't lost their life)
      {
            // each pass through this loop is a tick (1/50th of a sec)

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

            // we write this code to handle the animation for you
            Animate_all_of_the_alive_actors_to_the_screen();
            Sleep_for_20ms_to_give_the_Player_time_to_react();
      }
      // the Player died - you're going to write this code
      Cleanup_all_game_world_objects();
   }

   Tell_the_user_the_game_is_over();      // "game over dude"
}
```

And here is what the `Ask_all_actors_to_do_something()` function might look like (which is one of the functions you'll have to write):

```
void Ask_all_actors_to_do_something()
{
     for (int i = 0; i < m_actors.size(); i++)
         if (m_actors[i]->isStillAlive())
             m_actors[i]->doSomething();
}
```

As you can see, in the example above, you have a vector of actor-pointers. Each actor has a doSomething method. In this method, each actor (e.g., an insect like a Flea) can decide what to do. For example, here is some pseudo code showing what a Flea might decide to do each time it gets asked to do something:

```
class Flea: public SomeOtherClass
{
  public:
    void doSomething()
    {
      If I'm dead then
            Don't do anything - just return immediately

      If I'm not at the bottom of the screen then
            Adjust my y location down by one
      ...
      If I moved onto the same square as the Player then
            Kill the Player
      Else if I've reached the bottom of the garden then
            Set my status to dead
    }
    ...
};
```

And here's what the Player's doSomething method 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 no Mushroom's there then
            Adjust my y location UP one square
      ...
      If the user pressed the space bar to fire then
            Determine what the Water Droplet hit, and kill
            the first insect/Mushroom it hit

      If the user moved to the same square as an insect then
            Set the Player's status to DEAD
    }
    ...
};
```

What Do You Have to Do?

You must create a number of different classes to implement the Scentipede game. Your classes must work properly with our provided classes, and **you must not modify our classes or our sources 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 (the garden) and all of the objects (insects, Mushrooms, Water Droplets and the Player Ship) that are inside of your garden.
2. You must create a class to represent the Player Ship in the game.
3. You must create classes for the Scentipede, Flea, Scorpion, Spider, and Mushroom, as well as any additional base classes (e.g., an Insect base class if you need one) that are required to implement the game.
4. You must create a class to represent the Water Droplets that the Player Ship shoots (more on this later).

You Have to Create the StudentWorld Class

The StudentWorld class is responsible for orchestrating virtually all game play – it keeps track of the whole game world (the garden and all of its inhabitants such as insects, the Player, Mushrooms, Water Droplets, etc). It is responsible for initializing this game world at the start of the game, asking all of the actors to do something during each tick of the game, destroying all of the actors/Game Objects in the game world when the user loses a life or destroying an actor when it disappears (e.g., an insect flies off the screen).

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

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

The init() method is responsible for initializing the garden (creating objects for all the Mushrooms and the Player Ship, and placing them at their proper locations in the garden). This method is automatically called by our provided code when either the game first starts or after the user loses a life (but has more lives left) and the game play is ready to resume.

Each time the move() method is called, it runs a single tick of the game. This means that it is responsible for asking each of the game actors (e.g., the Player, Scentipede, Spiders, etc.) to try to do something: e.g., move themselves and/or perform their specified behavior. This method also occasionally introduces new insects into the game world – for instance, a Spider might be introduced into the game at a random tick during game play

to cause trouble for the Player. Finally, this method is responsible for disposing of Game Objects (e.g., Mushrooms, insects, etc) that happen to die during a given tick. For example, if the Player shoots a Spider, then the after all of the actors in the game get a chance to do something during the tick, the move() method should remove the Spider from the game world (by deleting its object). The move() method is automatically called once during each tick of the game by our provided game framework.

The cleanup() method is called when the Player loses a life (e.g., the Player Ship is killed by an insect). It is responsible for freeing all Game Objects (e.g., Mushroom objects, Spider objects, Water Droplets, the Player object, etc) that are currently active in the game at the time when the Player lost the life. This includes all actors created during either the init() method or introduced during subsequent game play in one of the many calls to the move() method that have not yet been removed from the game.

You may add as many other public/private methods and private member variables to your StudentWorld class as you like.

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

```
void increaseScore(unsigned int howMuch);
unsigned int getPlayerScore();
bool getKey(int& value);
int getTestParam(int paramID) const;
bool testParamsProvided() const;
```

The increaseScore() method is used by your StudentWorld class (or your other classes) to increase the Player's score upon successfully killing an insect or a Mushroom. When your code calls this method, you must specify how many points the Player gets (e.g., 20 points for a Spider). This means is that the game score is controlled by our GameWorld object – you must not maintain your own score member variable in your Player Ship class.

getPlayerScore() can be used by your StudentWorld class to determine the user's current score.

getKey() can be used to determine if the Player (the user) has hit a key on the keyboard to move the Player Ship. 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 filled in with the key that was pressed by the user (if any was pressed). If the Player does hit a key, the value argument will be set to one of the following values (defined in GameConstants.h):

```
KEY_PRESS_LEFT_ARROW
KEY_PRESS_RIGHT_ARROW
KEY_PRESS_UP_ARROW
KEY_PRESS_DOWN_ARROW
KEY_PRESS_SPACE
```

getTestParam() can be used to get the value of a test parameter provided by the CS32 grader. Our CS32 graders don't have time to play each student's game for hours on end to check for bugs. Therefore, they are providing you with special settings that your code must use in order to help facilitate testing. For example, the grader may want to test your project with 1000 Mushrooms on the screen rather than the usual 25. If so, they will set a test parameter indicating how many Mushrooms should be on the screen when a new game begins. Your code can use the getTestParam() method to obtain these grader-provided parameters. For example, here's how your code could get the initial Mushroom count:

```
int numShroomsToPutIngarden =getTestParam(TEST_PARAM_STARTING_MUSHROOMS);
```

The TEST_PARAM_STARTING_MUSHROOMS constant as well as all other TEST_PARAM constants are defined in GameConstants.h.

Don't worry — the rest of the spec will specify, in detail, what parameters you are responsible for in your code.

The testParamsProvided() method can be used to determine if the grader provided any testing parameters at all. This method returns a bool: False means you can assume that normal game play is taking place (i.e., a regular user is playing your game); true means the grader has specified one or more special settings that must be honored by your code (meaning that your program is being tested and must exhibit slightly different behaviors to facilitate testing).

init() Details

Your init() method must initialize the data structures used to keep track of your game world and then allocate and insert Mushroom objects and the Player Ship object into the game world.

It is recommended that you keep track of all of your Game Objects (e.g., Mushrooms, Fleas, etc) in an STL data structure like a vector or list of pointers to your objects.

You should 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 a game continues after the user has lost a life and has more lives left).

Adding Mushrooms

Your init() method is responsible for populating the garden with normal Mushrooms. The number of normal Mushrooms that you must insert into the garden is as follows:

If the grader provided a starting number of Mushrooms, then you must add that many Mushrooms to the garden at the start of the game and every time game play resumes after

the Player loses a life; otherwise, you must start with exactly 25 Mushroom in the garden. Here's a way to check if the grader wants to specify the number of starting Mushrooms:

```
int numMushrooms;
if (testParamsProvided())
    numMushrooms = getTestParam(TEST_PARAM_STARTING_MUSHROOMS);
else
    numMushrooms = 25;  // default number of Mushrooms to start with
```

Mushrooms must be placed randomly in the garden, and you must not place any Mushrooms in either the top or bottom rows (row 0 or row 19) of the garden. (This means you will not place a Mushroom where the Player Ship is placed initially (at x=15,y=0). Only a single Mushroom is allowed in a given square of the grid, so your placement algorithm must insure that multiple Mushrooms aren't placed in the same square.

Adding the Player Ship to the Game World

Your init() method must place the Player Ship in the garden. The Player Ship must start in location x=15, y=0 at the bottom-middle of the garden.

The 25 Mushrooms and the Player Ship are the only Game Objects that may start in the garden once the init() function completes. Other Game Objects (e.g., Fleas, etc) will be introduced to the game by the move() method (or a method of your choosing called by the move() method).

move() Details

The move() method must perform three different activities:

1. It must determine if it's time to add new insects into the game, and if so, add these to the game world.
2. It must ask each of the Game Objects that is currently in the game world to do something (e.g., ask a Flea to move itself).
3. It must then delete any Game Objects that have died during this tick (e.g., if a Flea was hitt by a Water Droplet, it should be removed from the game world)
4. The method must then determine if the Player Ship died during the current tick (e.g., because a Scentipede or some other insect ran into it, or it ran into one of them)

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

```
GWSTATUS_PLAYER_DIED
GWSTATUS_CONTINUE_GAME
```

The first return value indicates that the Player Ship died during the current tick, and instructs our provided framework code to tell the user that they died and restart the game (if the user has more lives left). The second return value indicates that the tick completed

without the Player dying, and therefore that game play should continue normally for the time being (i.e., in another 20ms, your move() method should be called again).

Here's the pseudo-code for your move() method:

```
int StudentWorld::move()
{
        // Add new insects
      addInsects();  // add any insects to the garden as required
                     // (see below for details)

        // the term "actors" refers to all insects, the Player's
        // Ship, Mushrooms and any Water Droplets that were
        // recently shot by the Player and are still active

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

        // Remove recently-dead Game Objects after each tick
      removeDeadGameObjects(); // "delete" dead bugs/shrooms/etc

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

      return GWSTATUS_CONTINUE_GAME;
}
```

Adding New Insects

You must use the following algorithm to decide whether to add new insects to the game world:

1. Maintain a counter member variable (we'll call it C in this description) in your StudentWorld object that starts at zero and is incremented by 1 during each call to the move() method.
2. If the grader provided test parameters dictating the frequency of appearance of new insects (which you can check using the testParametersProvided() method described above), then you must add new insects as follows:
    a. If C is greater than zero and C is evenly divisible by `getTestParam(TEST_PARAM_CENT_CHANCE_INDEX)` then you must add a new

Scentipede to the garden (details on where the centipede should start are below)

b. If C is greater than zero and C is evenly divisible by `getTestParam(TEST_PARAM_SPIDER_CHANCE_INDEX)` then you must add a new Spider to the garden (details on where the Spider should start are below)
c. If C is greater than zero and C is evenly divisible by `getTestParam(TEST_PARAM_FLEA_CHANCE_INDEX)` then you must add a new Flea to the garden (details on where the Flea should start are below)
d. If C is greater than zero and C is evenly divisible by `getTestParam(TEST_PARAM_SCORPION_CHANCE_INDEX)` then you must add a new Scorpion to the garden (details on where the Scorpion should start are below)
e. Your method that adds new insects must not add any additional insects at all during this tick (*even* if no insects were added in steps a-d above).

3. If the grader did **not** provide test parameters dictating the frequency of appearance of new insects, then you must add new insects as follows:
    a. Determine the current level of the game:
        currentLevel = The-user's-current-score / 500
    b. Count the number of each type of enemy (e.g., the number of Scentipede Segments, Fleas, Spiders, Scorpions) currently in the game world
    c. Your code must add a new Scentipede to the game world only under the following circumstances:
        i. If the current number of Scentipede Segments in the garden is a least min(20,(currentLevel+1) * 3) **or** there is a Scentipede Segment on the top row (y=19) of the garden, you must **not** add a new Scentipede to the garden during this tick; however you must still check to see if you possibly need to add one or more other insects to the garden, so continue with item d below.
        ii. Otherwise, if there are zero Scentipede Segments in the entire garden, then there is a 1 in 20 chance that you should add a new Scentipede to the top row of the garden (y=19) during this tick; in other words, generate a random number between 1 and 20, and if the random number is 1 (a 5% chance) then add a new Scentipede. Details on where and how to add the new Scentipede to the garden are below. Continue with item d below.
        iii. Otherwise, if there are one or more Scentipede Segments already in the garden, then the chance of adding a new Scentipede during the current tick is 1 in max(300 - 30 * currentLevel, 50).
    d. Fleas, Spiders and Scorpions are antisocial; therefore if there are too many of them in the garden already, new ones will stay away. If the total count of Fleas, Spiders, and Scorpions (i.e., #fleas + #spiders + #scorpions) currently in the garden is at least currentLevel/2 + 1, then you must not add any new Fleas, Scorpions or Spiders to the garden during the current tick.
    e. Otherwise (there is a chance of adding a new Flea/Scorpion/Spider)
        i. The chance of adding a new Flea is 1 in

max(300- currentLevel *30,100). If you add a new Flea during the current tick, then you must not add any additional insects during this tick. (i.e., skip ii and iii below)

ii. The chance of adding a new Spider is 1 in max(350- currentLevel *30,100). If you add a new Spider during the current tick, then you must not add any additional insects during this tick (i.e., skip iii below).

iii. The chance of adding a new Scorpion is 1 in chance = max(400- currentLevel *30,100).

Adding New Scentipedes

When adding a new Scentipede to the garden, you must always add it to the top row, in the right-most position of the board. Each new Scentipede is composed of a random number of Scentipede Segments between 6 to 12. The head Segment of the Scentipede must be added first to the garden, and then each following Segment should be added next, until the tail Segment is reached and added to the garden.

So, for example, if you decide that your new Scentipede should be 7 segments long (by generating a random number between 6 and 12, inclusive), then its head segment should be added first at x=6, y=0, the next segment should be added second at x=5, y=0, etc. and its tail segment should be added last at x=0,y=0.

When you add a Scentipede to the garden, you're not really adding a single Scentipede but rather multiple Scentipede Segments. Each Segment of the Scentipede is in reality its own object and operates independently of the other segments (however, because of the design of our Scentipede Segment movement algorithm, all the segments move together in unison like a complete Scentipede).

Adding New Fleas

When adding a new Flea to the garden, it must start at a random x location (between 0 and GARDEN_WIDTH-1, inclusive), and at a y location of 19 (GARDEN_HEIGHT-1), at the top of the garden.

Adding New Scorpions

When adding a new Scorpion to the garden, it must start at an x location of zero (on the left side of the garden), and at a random y location between y=4 and y=19, inclusive.

Adding New Spiders

When adding a new Spider to the garden, there is a 50% chance it will start on the left side of the screen at x=0 (and move right), and a 50% chance it will start at the right side of the screen at x=29 (and move left). Spiders start at a random y location between y=4 and y=19, inclusive.

Give Each Game Object a Chance to Do Something

Game objects include the Player Ship, all Scentipede Segments, Fleas, Spiders, Scorpions, Mushrooms and Water Droplets (fired by the Player Ship). During each tick of the game, after first possibly adding new insects into the garden, each active Game Object must have an opportunity to do something.

Your move() method must loop through each active Game Object in the garden (i.e., held by your StudentWorld object) and ask it to do something by calling a method named doSomething(). In each Game Object's doSomething() method, the object will have a chance to perform some activity: e.g., move, shoot, disappear from the garden, etc.

It is possible that one Game Object (e.g., the Player) may kill another Game Object (e.g., a Flea) during the current tick (e.g., by shooting it). If a Game Object has been killed earlier in the current tick by another Game Object, then the killed Game Object must not have a chance to do something during the current tick (since it's dead).

Note: When animating the Game Objects during a tick, you must make sure that you always animate Scentipede Segments from head to tail. In other words, if a Scentipede has 6 segments {s5,s4,s3,s2,s1,s0}, with the head segment being s0, and the tail segment being s5, then your move() method must make sure to always give s0 a chance to do something before giving s1 a chance to do something, and similarly, it must always give s1 a chance to do something before giving s2 a chance to do something, etc. This requirement ensures that the each Segment of a Scentipede doesn't attempt to run into the Segment in front of it before the Segment in front of it has a chance to move itself.

Remove Dead Game Objects after Each Tick

At the end of each tick your move() method must visit all of your Game Objects (e.g., Mushrooms, Spiders, Water Droplets, etc) to determine which ones are no longer alive, and delete these objects and remove them from your collection of active Game Objects. So if, for example, the Player shoots and kills a Mushroom, then the Mushroom's object should be removed from the collection of active objects and its object should be deleted to free up room in memory for future Game Objects that will be introduced later in the game. (Hint: All of your Game Objects will need to have a member variable indicating whether or not they are still alive!)

cleanUp() Details

When your cleanUp() method is called by our game framework, it means that the user lost a life. In this case, the entire garden (every insect, Mushroom, Water Droplet, the

Player Ship, etc., in the garden) must be deleted, resulting in an empty garden. Assuming the user has more lives, our provided code will subsequently call your init() method to repopulate the garden and the game will the continue with a brand new set of Game Objects.

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

You Have to Create the Classes for All Game Objects

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

- Mushrooms (Regular and Poison ones)
- The Player Ship
- Scentipede Segments (1 or more make up a full Scentipede)
- Fleas
- Spiders
- Scorpions
- Water droplets

Each of these Game Objects can occupy the garden and interact with other Game Objects within the garden.

Now of course, many of your Game Objects will share things in common – for instance, every one of the objects in the game (Mushrooms, the Player, Spiders, etc) has x,y coordinates. They all have the ability to perform an action during each tick of the game. Each of them can be potentially attacked (e.g., by a squirt of water, or by colliding with each other) and could die during a tick. All of them need some attribute that indicates whether or not they are still alive (or have been killed during the current tick), etc., 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 grade on this project will largely depend upon your ability to create a well-designed set of classes that follow object oriented principles. Your classes should never duplicate member functions implementations or data members – if you find yourself writing the same (or largely similar) code across multiple classes than this is an indication that you should define a common baseclass and migrate this common functionality/data to the base class.

You must derive all of your Game Objects from a base class that we provide called GraphObject, e.g.:

```
class GameObject: public GraphObject
{
  public:
```

```
        …
};

class Insect: public GameObject
{
  public:
        …
};

class Spider: public Insect
{
  public:
        …
};
```

A GraphObject is a special class that we have defined that contains a lot of the ugly logic required to display your Game Objects on the screen in 3D. If you don't derive your classes from our GraphObject base class, then you won't see anything displayed on the screen! ☺

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

```
GraphObject(int imageID);          // the constructor
unsigned int getID() const;
void setID(unsigned int imageID);
void displayMe(bool shouldIDisplay);
void setInitialLocation(int x, int y);
void moveTo(int x, int y);
void getLocation(int& x, int& y) const;
void setBrightness(double brightness)
double getBrightness() const
```

You may use any of these methods in your derived classes, but you must not use any other methods found inside of GraphObject in your other classes. You must not redefine any of these methods in your derived classes since they are not defined as virtual in our base class.

GraphObject(int imageID) 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 Spider, a Player Ship, etc.). One of the following IDs, found in GameConstants.h, must be used to initialize each GameObject:

```
IID_CENT
IID_SPIDER
IID_FLEA
IID_SCORPION
IID_PLAYER
IID_MUSHROOM
IID_POISON_MUSHROOM
IID_LASER
```

New GraphObjects always start out with a location of x=0, y=0 unless you specify otherwise by calling setInitialLocation. New GraphObjects start out invisible and are not displayed on the screen until the programmer calls the displayMe() method with a value of true for the parameter. New GraphObjects start out with a brightness level of 1.0 (normal brightness).

getID() can be used to determine a GraphObject's display ID (e.g., IID_CENT). You are welcome to call this method to discern your GameObjects from one another (e.g., is that other object a Scorpion or a Mushroom?).

setID() can be used to change a GraphObject's display ID. You may only use this to change a (Regular) Mushroom to a Poison Mushroom. You must not use this method for any other purpose.

displayMe(bool shouldIDisplay) is used to tell our graphical system whether or not to display a particular GraphObject on the screen. If you call displayMe(true) on a GraphObject, then your object will be displayed on screen automatically by our framework (e.g., a Flea image will be drawn to the screen at the GraphObject's specified x,y coordinates if the object's ID is IID_FLEA). If you call displayMe(false) then your GraphObject will not be displayed on the screen. Once you create a new GameObject (e.g., a Flea, a Mushroom, Water Droplet or the Player Ship, always remember to call the displayMe() method with a value of true or the GameObject won't display on screen!)

setInitialLocation(int x, int y) is used to specify the initial location of a GraphObject in the garden. You may specify an x value between 0 and GARDEN_WIDTH (29), and a y value between 0 and GARDEN _HEIGHT (19). You must not use this method to move a GraphObject (like a Flea) from one spot on the screen to another spot on the screen. Instead, you must use this only to set the initial coordinates of a GraphObject.

moveTo(int x, int y) is used to update the location of a GraphObject within the garden to an adjacent square. For example, if a Flea's movement logic dictates that it should move to the right, you could do the following:

```
int x,y;
getLocation(x,y);
moveTo(x+1,y);      // 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. Do not use the setInitialLocation() to update the location of an object during game play as this will not result in the smooth animation of the Game Object on the screen.

getLocation(int &x, int &y) is used to determine the current location of a GraphObject in the garden. Since each GraphObject maintains an x,y location, this means that your derived classes must not also have x,y member variables, but instead rely upon those in the GraphObject base class.

setBrightness(double brightness) is used to specify the relative brightness of a GraphObject on the screen. A passed-in value of 1.0 indicates that the object is at full brightness, while a passed-in value of 0.0 indicates that the object is totally blackened out. A value of .5 would therefore indicate that the object is half its usual brightness. You should use this method to adjust the brightness of your Mushroom and Poison Mushroom objects as they are shot by the Player (they get darker as they are more damaged from shots, until they finally disappear).

getBrightness() can be used to determine the current brightness of a GraphObject.

Mushrooms

Here are the requirements you must meet when implementing Mushrooms (likely inside a Mushroom class, derived in some way from our GraphObject class):

There are two types of Mushrooms: regular Mushrooms (with an image ID of IID_MUSHROOM) and poison Mushrooms (with an image ID of IID_POISON_MUSHROOM).

All Mushrooms start out with exactly 4 "lives."

Mushrooms can be attacked (shot) by the Player's water cannon up to 4 times before they must die/disappear from the game. Mushrooms can also be attacked by Scentipedes and Spiders, who can kill Mushrooms completely in a single attack. (Hint: Since there are multiple ways for a Mushroom to get attacked – by the Player, by Spiders, etc., it might make sense to have a method that can be used to tell a Mushroom that it's been attacked – hey, wait a second – almost all objects in the game can be attacked, so maybe this method should be in a base class. Hmmmm...)

All Mushrooms have an x,y coordinate.

Any time a Player shoots a Mushroom of either type with his/her water cannon, the Player gets 1 point. Therefore, if a Player destroys a Mushroom by shooting it 4 times, they would get 4 total points. (Hint: When a Mushroom is hit by the Player, it must make sure that the increaseScore() method of our GameWorld class is called to tell the game that the Player has received a point.[1])

Every time a Mushroom is shot its brightness must decrease by ¼ of its initial brightness; so a Mushroom would start out with a brightness of 1.0, and after being shot once, decrease to a brightness of .75, then .50, etc.

[1] Hint: Your Mushroom class, or more correctly, one of its super classes, must have a pointer to its containing StudentWorld class (which is derived from GameWorld) so the Mushroom can call the increaseScore() method when it's shot by the Player. You can pass the pointer to StudentWorld in to your GameObjects when you construct them, for example.

Once a Mushroom is shot four times, it must die and disappear from the garden.

The Player Ship

Here are the requirements you must meet when implementing the class for the Player Ship (likely inside a PlayerShip class, derived in some way from our GraphObject class):

The Player Ship must have an image ID of IID_PLAYER.

The Player loses a life any time an insect (e.g., Spiders, Fleas, Centipede Segments, Scorpions) moves onto the same square as the Player, or any time the Player moves onto the same square as any of these insects.

What the Player Ship Must Do During a Tick

The Player Ship must be given an opportunity to do something during every tick. When given an opportunity to do something, the Player Ship must do the following:

1. See if the user has recently pressed a key (e.g., to move or to fire) – you can use the getKey() method of our GameWorld class for this[2].
2. If the user pressed a directional key (left, right, up, down) then your class must move the Player Ship in the specified direction, subject to the following constraints.
    a. The Player Ship must never be allowed to move out its designated area of the garden; specifically, the Player Ship is confined to x >=0, x < GARDEN_WIDTH, y >= 0, y <= 3.
    b. The Player Ship must never be allowed to move on top of a live Mushroom. They will fail to move at all if they try to move onto of a live Mushroom of any type.
    c. If the Player Ship tries to move onto an insect, the Player Ship must set its status to "dead" (the user may have more lives and get to continue playing, but this round of the game will be over).
    d. If both an insect and a Mushroom occupy the same square, and the Player Ship tries to move onto that square, then the Player must set its status to "dead" since it tried to move into an insect, and this is deadly for the player (the Player may have more lives and get to continue playing, but this round of the game will be over).

[2] Hint: In order for your PlayerShip class to call the getKey() method of the GameWorld class, it needs to have a pointer to your StudentWorld object (which is derived from the GameWorld object and therefore inherits all of its methods) – therefore, it makes sense for your Player Ship or, more correctly, one of its super classes to hold a pointer to its StudentWorld object. You can pass this pointer to StudentWorld in when you construct a PlayerShip object.

e. Your Player class (and its base class(es)) must NOT use the resetLocation() method in GraphObject to update the location of the Player.

3. If the user pressed the fire key (the space bar), then your class must fire the Player's water cannon subject to the following requirements:
   a. The Player Ship may only fire every other tick. Therefore if a Player fires their cannon during the current tick, and they also hit the space bar to fire during the next tick, nothing will happen during the next tick. This prevents the Player from rapid-firing during every tick and cheating. You must therefore add logic to this class to prevent the user from firing during every trick.
   b. Assuming the Player is allowed to fire their water cannon during the current tick, your class must use the following algorithm to fire:

      Compute the starting coordinate of the Water Droplets that will fire from the Player Ship:

      dropletX = PlayersCurrentX coordinate
      dropletY = PlayersCurrentY coordinate + 1

      While (dropletX, dropletY is still within the confines of the garden)

      If the square (dropletX, dropletY) is occupied by another Game Object (e.g., a Mushroom or an Insect) then
      The PlayerShip should directly attack that object (e.g., call a method in the target object to indicate that the target object has been hit by a Water Droplet from the Player's cannon); this will, for example, cause a Mushroom to lose one of its four lives, or kill an insect (a single shot kills an insect).
      The loop must immediately end and no further droplets may be added to the garden if/when another GameObject is hit.

      Otherwise, if the garden square (dropletX, dropletY) is empty then
      Your Player must introduce a new Water Droplet object into the garden at that dropletX, dropletY location and make this droplet visible. This ensures that the user can see where they're firing.

      Increase the current dropletY by one, and continue the loop.

4. The Player may only perform one action per tick. In other words, they can either do nothing during a tick, move one square horizontally/vertically in a tick, OR fire during at tick. But they can't, for example, fire and move left during the same tick.

<u>The Player Class Must Have a Method that can be Called if He/She is Attacked</u>

Any time the Player Ship is attacked (e.g., another insect moves onto the same square as the Player Ship), the insect must call a method in the Player Ship class telling the Player Ship that it has died. In this method, the Player object should record the fact that it has transitioned from an alive state to a dead state.

Getting Input From the User

Since Scentipede is a "real-time" game, you can't use the typical getline and cin commands to prompt the user for input within the Player Ship's doSomething() method. These commands would stop your program and wait until the user types in the proper data and hits the enter key. This would make for a really boring Scentipede game (requiring the user to hit a directional key then hit enter, then hit a direction key, then hit enter, etc). Instead, you will need to use a special function that we provide in our GameWorld class (which your StudentWorld class is derived from) called `getKey()` to get input from the user[3]. This function rapidly checks to see if the Player hit a key and then returns the result to you (whether or not they hit a key). If the Player hit a key, then the key they typed is sent back to the caller of the getKey() function. Otherwise, the function immediately returns a result informing the caller that no key was hit. This function could be used as follows:

```
#include "myio.h"
…
void Player::DoSomething(void)
{
     bool bGotAKey;
     char ch;
     StudentWorld *sw;

     sw = geyMyWorld(); // get's the world the Player
                        // is located in

    bGotAKey =  sw->getKey(ch);

    if (bGotAKey == true)
    {
       switch (ch)
       {
         Case KEY_PRESS_LEFT_ARROW:
           MovePlayer(left);
           break;
         case KEY_PRESS_RIGHT_ARROW:
```

[3] Hint: Since your Player Ship class will need to access the getKey() method in the GameWorld class (which is the base class for your StudentWorld class), your Player Ship class (or more correctly, one of its base classes) will need to have a pointer to the StudentWorld class. If you look at the code example below, you'll see how the Player's doSomething() method first gets a pointer to its world via a call to getMyWorld() (a method in one of its base classes that returns a pointer to StudentWorld), and then uses this pointer to call the getKey() method.

```
                MovePlayer(right);
                break;

               // etc…
            }
        }

        ...
    }
```

The Water Droplet

Here are the requirements you must meet when implementing the Water Droplet (likely inside a WaterDroplet class, derived in some way from our GraphObject class):

Water droplets are fired by the Player Ship to kill Mushrooms and insects.

The WaterDroplet class is very simple – it simply is used to display a Water Droplet in the garden for two ticks, then the Water Droplet disappears (kills itself) and is removed from the Garden. When the Player Ship fires their water cannon, that code must insert a series of 0 or more Water Droplets into the Garden to show the player where they fired. The Water Droplets don't actually attack anything (the Player Ship is responsible for attacking whatever Game Object would have been hit by the Water Droplets); instead, the Water Droplets simply need to display themselves and then go away.

Each Water Droplet object must have an image ID of IID_WATER_DROPLET.

When a Water Droplet is created, it is initialized with a lifetime of exactly two ticks. Water Droplets have a lifetime of two ticks so that they remain displayed on the screen long enough to be seen by the Player.

Each time the Water Droplet is asked to do something (during a tick), it should decrement its remaining lifetime count by one.

When a Water Droplet reaches a lifetime of zero, it should kill itself, hence causing it to disappear from the garden at the end of the current tick.

Scentipede Segments

A single Scentipede is made up of multiple Scentipede Segments. A Scentipede, made up of 6 segments, for example, does NOT operate as a single cohesive unit. Instead, each Scentipede Segment operates as if it were its own unique insect. The collective behavior of all of a Scentipede's segments causes the Scentipede as a whole to act as a single, cohesive unit (this is called "emergent behavior" – even though each segment is unaware of the other segments, their individual algorithm causes their collective behavior to

appear to appear organized). Therefore, rather than defining a Scentipede class that manages an entire Scentipede you will define a class that manages an individual Scentipede Segment.

Here are the requirements you must meet when implementing the Scentipede Segment (likely inside a ScentipedeSegment class, derived in some way from our GraphObject class):

All Scentipede Segments must have an image ID of IID_SCENTIPEDE_SEGMENT.

All Scentipede Segments must maintain at least two movement states: what horizontal direction they're currently moving in, and what vertical direction they'll move in if/when they hit the edge of the screen or a Mushroom.

All Scentipede Segments must start out with a horizontal movement direction of right and a vertical movement direction of down when they are first created. This means, practically, that all Scentipede Segments will start out by moving rightward, and when they hit first the right edge of the screen, will the move down one square before switching their horizontal direction and moving leftward back across the screen, etc. Details of their movement algorithm are described below.

All Scentipede Segments start out in an un-poisoned state.

<u>What the Scentipede Segment Must Do During a Tick</u>

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

1. If the Scentipede Segment is currently in a poisoned state (because it ran into a poison Mushroom during an earlier tick), then it must:
    a. Check if its current y location is 0 (indicating it has reached the bottom of the garden). If so, it should switch back into an un-poisoned state and immediately continue with step 2 below. This segment is no longer poisoned.
    b. Otherwise if the y location is > 0, the Scentipede Segment should move down one square.
    c. If the Scentipede Segment moves down onto a square containing a Mushroom (of either Normal or Poison type), then the Scentipede Segment must tell the Mushroom that it has been killed (the Player gets no points for this) and, as such, the Mushroom must disappear from the garden at the end of the tick.
    d. If the Scentipede Segment moves down onto a square containing the Player, then the Scentipede Segment must kill the Player (tell the Player he/she's been hit).
    e. If the Scentipede Segment moves down onto a square containing another Scentipede Segment, then the moving Scentipede Segment must kill the

other Scentipede Segment (tell the moved-onto segment it's been hit). The segment doing the moving will not die, but the moved-onto segment will die.

   f. Note: If the Scentipede Segment moves onto a square with any other type of insect (e.g., Flea, Spider, etc.), then the segment just moves and ignores the fact that it's on the same square as another insect.
   g. Assuming the segment is still poisoned, after moving, the Scentipede Segment should now return and perform no other actions during the current tick. If the segment is no-longer poisoned, continue to step #2.
2. The Scentipede Segment computes a target square (to move to) based on its current location and its current horizontal movement direction. Call this target square newx, newy. For example, if the segment were currently at location curx=4,cury=9 and the segment had a horizontal movement direction of right, then newx,newy would be newx=5,newy=9.
3. If newx is less than zero, then that means the segment has reached the left side of the screen and is ready to start moving in the opposite direction. Do not update the segment's actual x,y location. Instead, change the segment's horizontal movement direction to right and continue with step 6.
4. If newx is equal to GARDEN_WIDTH, then that means the segment has reached the right side of the screen and is ready to start moving in the opposite direction. Do not update the segment's actual location. Change the segment's horizontal movement direction to left and continue with step 6.
5. Otherwise this means that newx is within the horizontal confines of the garden (newx >=0 and newx < GARDEN_WIDTH); do the following:
   a. If the target square described by newx, newy does not contain either a Mushroom or the Player or another Scentipede Segment, then the Scentipede Segment must set its coordinates to newx, newy, effectively moving onto this target square, and then immediately return (doing nothing more in this tick).
   b. If the Player Ship is at location newx, newy, then the Scentipede Segment must kill the Player (tell the Player Ship's object its been hit) and immediately return.
   c. If the target square at newx,newy contains a Poisoned Mushroom, then
      i. The Scentipede Segment will change into a poisoned state.
      ii. The Scentipede Segment will change its vertical direction to down.
      iii. The segment must NOT move onto the target square (newx,newy) holding the poisoned Mushroom.
      iv. Your code must continue with step 6.
   d. If the target square at newx,newy contains a regular Mushroom, then the segment must NOT move onto the target square, and you should proceed to step 6.
   e. If the target square at newx,newy contains another Scentipede Segment, then the segment must NOT move onto the target square, and you should proceed to step 6.
6. If you get to this step, it means that your segment has either run into a Mushroom or reached the left/right horizontal edge of the screen. At this point, your segment

will either move up or down, depending on its current vertical movement direction (during the next tick, it will start trying to move horizontally again). You first must determine a new potential coordinate newx,newy based on the segment's current location curx,cury and the segment's current vertical movement direction. For example, if the segment were currently at curx=4,cury=9 and it were currently moving down, then newx,newy would be newx=4,newy=8.

7. If newy < 0 then this means that the segment has reached the bottom of the garden and now will start moving up towards the top of the garden. The segment should set its vertical movement direction to up, and its newy variable to 1, then continue with the step 9 below. (Note: the segment should not update its actual location, just its newy variable, which will be used below)
8. Otherwise, if newy == GARDEN_HEIGHT then this means that the segment has reached the top of the garden and now will start moving down towards the bottom of the garden. The segment should set its vertical movement direction to down, and its newy variable to GARDEN_HEIGHT-2, then continue with the step 9 below. (Note: the segment should not update its actual location, just its newy variable, which will be used below)
9. The segment should now determine the contents of square newx,newy in the garden.
10. If this new square does not contain the Player Ship and it doesn't contain a Mushroom (of either type) and it doesn't contain another Scentipede Segment, then the segment must move onto this new square (update its location to newx, newy) and then immediately return.
11. If the new target square instead contains a Poison Mushroom, then
    a. The segment should immediately change into a poisoned state.
    b. The segment should change its current vertical direction to down.
    c. The segment should update its newx,newy location to be down exactly one square from its current x,y location.
    d. Now continue with step 12.
12. If the target square newx,newy (which may have changed due to step 11c above, meaning you may have to re-check what is at square newx,newy) contains a Mushroom (of either Normal or Poison type) or the Player Ship or another Scentipede Segment, then the moving Scentipede Segment must kill that object (the Player gets no points for this) and the killed object must disappear from the garden at the end of the current tick. If the target square is occupied, this will therefore either result in a Mushroom disappearing, the Player Ship being killed, or another Scentipede Segment being killed.
13. Finally, the segment should update its coordinates to newx,newy, officially moving onto this square, then return.

Note: Scentipede Segments ignore all other non-Scentipede insects (e.g., Fleas, Spiders or Scorpions), and in fact, may reside in the same square of the garden as another non-Scentipede insect without any issues!

<u>What a Scentipede Segment Must Do When Its Attacked</u>

Each Scentipede Segment may be attacked by the Player when they shoot their water cannon. If a Scentipede Segment is hit by the Player, then:

1. The Player must receive 10 points for killing the segment.
2. If the Scentipede Segment's y location is less than GARDEN_HEIGHT -1 (i.e., its not in the very top row of the garden), then there is a 33% chance (1 in 3) that the Scentipede Segment will drop a new Regular Mushroom in its current x,y location when it dies.
3. The Scentipede Segment should die and be removed from the garden at the end of the current tick.

Each Scentipede Segment may also be destroyed by another Scentipede Segment that moves onto the same square as them. If a Scentipede Segment is destroyed in this manner, it will simply disappear from the garden at the end of the current tick, but will not drop a Mushroom into the garden. The Player will receive no points if a Scentipede Segment dies in this manner.

Fleas

Here are the requirements you must meet when implementing the Flea (likely inside a Flea class, derived in some way from our GraphObject class):

All Fleas must have an image ID of IID_FLEA.

To determine the x,y location where a Flea should start out in the garden when first created, see the section Adding New Fleas. The Flea must start out with a vertical movement direction of down, and no horizontal movement direction (since Fleas ONLY move downward through the garden).

What a Flea Must Do During a Tick

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

1. If a Flea is on a square that does not contain any type of Mushroom, and the Flea's current y coordinate is strictly greater than 0 and strictly less than GARDEN_HEIGHT-1, then there is a 25% (1 in 4) chance that the Flea will drop a Regular Mushroom into the garden at it's current square.
2. The Flea determines its new coordinate (where it would like to move), which is always one down from its current x,y location. For instance, if the Flea is currently at x=10,y=5, the Flea's new coordinate will be x=10,y=4.
3. If the Flea's new location would place it past the bottom of the garden (to y=-1), then it should "die" and be removed from the garden at the end of the current tick (the user doesn't get any points for this).
4. Otherwise, the Flea should move onto the target square.

5. If the Flea moved onto the same square as the Player Ship, then the Flea should attack (and kill) the Player Ship.

What a Flea Must Do When Its Attacked

Each Flea may be attacked by the Player Ship with their water cannon. If a Flea is hit by the Player Ship, then the user must receive 50 points for killing the Flea and the Flea must die and disappear from the game during the current tick.

Fleas are never attacked by any other means (i.e., other insects can't hurt Fleas) and therefore can only die if they are shot by the Player Ship or they leave the bottom edge of the screen.

Spiders

Here are the requirements you must meet when implementing the Spider (likely inside a Spider class, derived in some way from our GraphObject class):

All Spiders must have an image ID of IID_SPIDER.

To determine the x,y location where a Spider should start out in the garden when first created, see the section Adding New Spiders.

The Spider must start out with a vertical movement direction of down. A Spider has a 50% chance of starting out on the left side of the screen (x=0) with a horizontal movement direction of right. Following from this, a Spider has a 50% chance of starting out on the right side of the screen (x=GARDEN_WIDTH-1) with a horizontal movement direction of left.

Spiders like to zig-zag up and down as they move horizontally across the screen. In other words, each time a Spider moves one square horizontally during a tick it will also move either up/down one square as well. The result is that Spiders always move diagonally (up+right, down+right, up+left, or down+left) during each tick.

Each Spider initially (during construction) selects a random number R (R must be between 1 and 4). We call R the vertical movement distance.

What a Spider Must Do During a Tick

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

1. A Spider will choose to rest every other tick (and do nothing). If it is the proper tick for a Spider to rest, it must simply return immediately.

2. If the Spider's current remaining vertical movement distance (R) is zero, it will flip its vertical direction and select a new value for R (see Creating a New Travel Plan below for details)
3. The Spider determines its new x,y coordinates based on its current horizontal movement direction and the current vertical movement direction. This new location will always be diagonal from the Spider's current position.
4. If the new x,y coordinate contains a Mushroom of either type, the Spider will eat (i.e., kill) the Mushroom (without the user getting any points) and the Mushroom will disappear from the garden at the end of the current tick.
5. If the Spider's new x,y coordinates would place it past the edge of the garden (to x=-1 or to x=GARDEN_WIDTH), then it must "die" and be removed from the garden at the end of the current tick (the user doesn't get any points for this).
6. Otherwise, the Spider should move onto its target diagonal square.
7. If the Spider moved onto the same square as the Player Ship, then the Spider must attack (and kill) the Player Ship. Spiders ignore all other insects including Scentipedes, Fleas, etc. and may occupy the same square as them without any interaction occurring.

Creating a New Travel Plan

This algorithm flips the Spider's current vertical direction and picks a new R value which determines how long the Spider will move in its new vertical direction before switching to the opposite vertical direction. To determine the travel plan:

1. If the Spider's current vertical direction is up, then
    a. Set its new vertical direction to down.
    b. Determine the Spider's y location.
    c. Set its vertical movement distance R to a random value between 1 and y-1, inclusive.
2. Otherwise, if the Spider's current vertical direction is down, then
    a. Set its new vertical direction to up.
    b. Determine the Spider's y location.
    c. Set its vertical movement distance to a random value between 1 and (GARDEN_HEIGHT-y)-1, inclusive.

<u>What a Spider Must Do When Its Attacked</u>

Each Spider may be attacked by the Player Ship with their water cannon. If a Spider is hit by the Player Ship, then the Player must receive 20 points for killing the Spider and the Spider must die and disappear from the garden at the end of the current tick.

Spiders are never attacked by any other means (i.e., other insects can't hurt Spiders by colliding with them).

Scorpions

Here are the requirements you must meet when implementing the Scorpion (likely inside a Scorpion class, derived in some way from our GraphObject class):

All Scorpion must have an image ID of IID_SCORPION.

To determine the x,y location where a Scorpion should start out in the garden when first created, see the section Adding New Scorpions. The Scorpion must always start out with a horizontal movement direction of right, and no vertical movement direction (since Scorpions ONLY move rightward through the garden).

What a Scorpion Must Do During a Tick

Scorpions have very simple behavior when asked to do something. During each tick, a Scorpion will get its current location in the garden and check to see if it is standing on top of a Regular Mushroom. If so, there is a 33% (1 in 3) chance that the Scorpion will convert the Regular Mushroom into a Poison Mushroom. The Scorpion will then move one square to the right, until it has moved past the right-hand side of the screen (at which point it should be removed from the garden at the end of the current tick). If, at any point when moving, the Scorpion moves onto the same square as the Player Ship, then the Scorpion must kill the Player Ship. Scorpions don't interact with any of the other insects in the game and may occupy the same square as another insect without issue.

What a Scorpion Must Do When It's Attacked

Each Scorpion may be attacked by the Player with their water cannon. If a Scorpion is hit by the Player, then the Player must receive 25 points for killing the Scorpion and the Scorpion must die and disappear at the end of the current tick.

Scorpions are never attacked by any other means (i.e., other insects can't hurt Scorpions by colliding with them).

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 iteratively almost always fail CS32's project #3, so don't do it!

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

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

   ```
   class foo
   {
   public:
       int getMyID() { return -1; }    // 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, re-compile 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.**

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 or thousands of errors and just get frustrated! So don't do it.

Compiling the Game

To compile the game, follow these steps:

For Windows

Unzip the proj3-windows.zip archive into a folder on your hard drive. You can then open the scentipede.sln file in Visual Studio 2010 and build your program.

For OS X

Unzip proj3-mac.zip archive into a folder on your hard drive.
Use our provided Scentipede.xcodeproj to build your program.

What To Turn In

Part #1 (30%)

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 Scentipede 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., Mushrooms, Scentipede Segments, Scorpions, the Player Ship, etc):
    i. It must have a simple constructor and destructor.
    ii. It must be derived from our GraphObject class.
    iii. It must have a single virtual method called doSomething() that can be called by the world to get one of the game's actors to do something.
    iv. You may add other public/private methods and private member variables to this base class, as you see fit.
2. 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 simple constructor and destructor that initializes the Player.
    ii. It must have an Image ID of IID_PLAYER.
    iii. It must have a limited version of a doSomething() method that lets the user pick a direction by hitting a directional ('2','4','6','8') key. If the Player hits a key during the current tick, and there is no Mushroom in specified direction and the target square is within the limits of where the Player is allowed to move (see the Player section), it updates the Player's location to the target square. All your doSomething() method has to do is properly adjust the Player's X,Y coordinates and our graphics system will automatically animate its movement it around the garden!
    iv. Hint: Your Player will have to ask the StudentWorld class that holds it (see below) whether or not a given square of the garden is occupied by a Mushroom. The Player Ship should only be able to move to an adjacent square if it is not occupied by a Mushroom.
    v. You may add any public/private methods and private member variables to your Player class as you see fit, so long as you use good object oriented programming style (e.g., don't duplicate functionality across classes).
3. Create a limited version of your Mushroom class, derived in some way from the base class described in #1 just above:

i. It must have a simple constructor and destructor that initializes a new Mushroom.
ii. It must have an Image ID of IID_MUSHROOM.
iii. It must have a limited version of a doSomething() method. Of course, it might not do too much since Mushrooms aren't too smart.
iv. You may add any set of public/private methods and private member variables to your Mushroom class as you see fit, so long as you use good object oriented programming style (e.g., don't duplicate functionality across classes).

4. Create a limited version of the StudentWorld class.
    i. Add any private member variables to this class required to keep track of exactly 25 Mushroom objects and the Player Ship object.
    ii. Implement a constructor for this class that initializes appropriate member variables.
    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 and distribute 25 Mushrooms randomly in the garden (as specified in the Adding Mushrooms section) – don't worry about addressing TA/grader parameters at this point.
    v. Implement the move() method in your StudentWorld class. You may ignore, for part 1 of this assignment, all specifications that refer to insects. You should program enough of the move() method so that it can ask your Player and all of the Mushrooms to do something during each tick and then return. Your move() method need not check to see if the Player Ship has died or not.
    vi. Implement a cleanup() method that frees any dynamically allocated data that was allocated during calls to the init() method (e.g., it should delete all your allocated Mushrooms and the Player).

Once you've implemented these classes, compile your program – you'll probably start out with lots of errors… Relax and try to remove all of the compile errors and get your program to run. (A historical note: When he was still a student taking CS131, Professor Nachenberg once got 1,800 compiler errors when compiling a 900-line program)

You'll know you're done with part 1 when your program compiles and does the following: When it runs and the user hits Enter to begin playing, it should display a garden filled with 25 randomly-distributed Mushrooms and with the Player Ship in its proper starting position. If your base class(es) and Player class work properly, you should be able to move the Player around the garden (except on top of Mushrooms) using the '2', '4', '6' and '8' keys, respectively.

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

Note, the Part #1 specification above doesn't require you to implement insects (unless you want to). You may do these unmentioned items if you like but they're not required for Part 1. HOWEVER – IF YOU ADD ADDITIONAL FUNCTIONALITY, MAKE SURE THAT YOUR PLAYER, MUSHROOM AND STUDENTWORLD CLASSES STILL WORK PROPERLY AND THAT YOUR PROGRAM STILL COMPILES AND MEETS THE REQUIREMENTS STATED ABOVE FOR PART #1!

If you can get this simple version working, you'll have done most 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 **compiles without any errors in Visual Studio**. It must consist of these **exact** files and no other source files:

```
actor.h            // contains base, Mushroom and Player class declarations
                   // as well as constants or #defines required by these classes
actor.cpp          // contains the implementation of these classes
StudentWorld.h     // contains your StudentWorld class declaration
StudentWorld.cpp   // contains your StudentWorld class implementation
```

You will not be turning in any other files – we'll test your code with our versions of the the other .cpp and .h files. Therefore, your solution may NOT modify any of our files or you will receive zero credit!

Part #2 (70%)

After you have turned in your work for Part #1 of Project 3, your TA 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 provided by the TA**.

In Part #2, your goal is to implement a fully working version of the Scentipede 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 base, Mushroom and Player class declarations
                 // as well as constants or #defines 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       // your report (10% of your grade)
```

You must turn in a report. The report should contain 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 blah() function in my base class because all objects in Scentipede have a blah function, and all of them define their own special version of it."
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 shooting of Water Droplets." or "My Scorpion doesn't work correctly yet so I just treat it like a Spider right now."
3. A list of other design decisions and assumptions you made, e.g.:
    i. It was ambiguous 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 ambiguous. What should I do?
A: Play with our sample program and do what it does. Use our program as a reference. If the specification is ambiguous and our program is ambiguous, do whatever you like and document it in your report. **If the specification is ambiguous, 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 compiles! If we can sort of play your game, but it's not perfect, that's better than it not even compiling!

Q: Where can I go for help?
A: Try Eta Kappa Nu – 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!