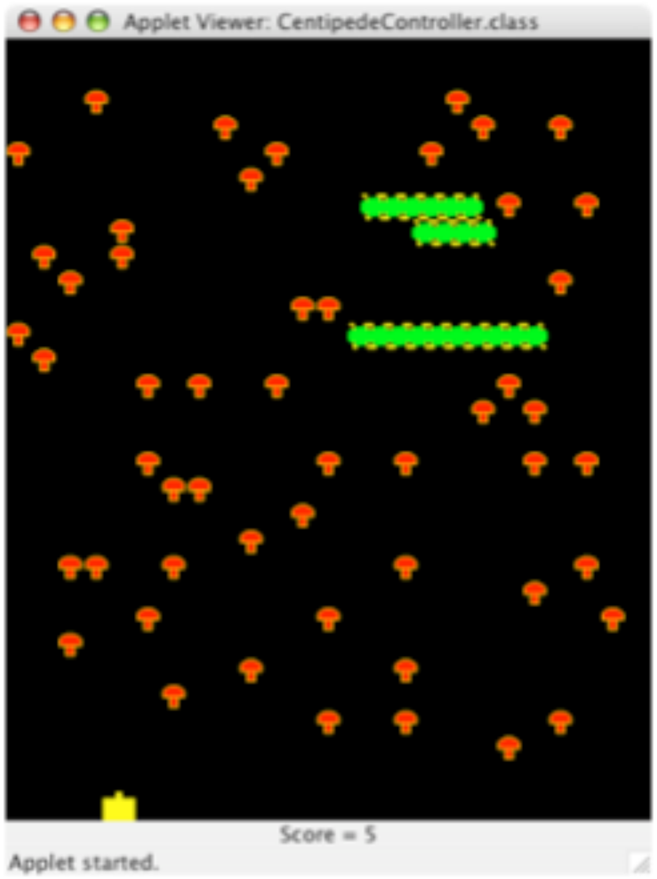
CS 51 Laboratory # 10
Centipede

Due: December 8th at Midnight
(Design due: November 24th, 11am)

A test program is a laboratory that you complete on your own, without the help of others. It is a form of take-home exam. You may consult your text, your notes, your lab work, the lecture notes, our on-line examples, and other documentation directly available from the course web page, but use of any other source for code is forbidden. If you have problems with the assignment, please see the instructor.

Introduction Centipede - with its colorful mushroom patch, tenacious centipedes, bouncing spiders, fleas and scorpions - was one of the earliest and most popular video arcade games. It was brought to life by Atari in 1981 and has remained relatively popular over the years. (One of the 55,000 game units built by Atari recently sold for several thousand dollars on EBay. Sadly, the Dean denied our request for discretionary funds to place a higher bid...) For Test Program 2, you will write a version of "Centipede." We have simplified the game somewhat to make the assignment more manageable.

A demo version of this program is on the on-line version of this handout. You can play with it to see what we have in mind.



Our simplified game begins with a centipede at the top of a field of mushrooms. A bug "zapper" sits at the bottom of the screen. The zapper's job is to defend itself against the attacking centipede.

The centipede is actually made up of a bunch of individual segments. Each segment moves horizontally across the screen, and a segment turns around and moves down a row of mushrooms toward the zapper whenever it runs into the edge of the screen or a mushroom. The segments all move at a fixed speed, but each segment keeps track of whether it is moving to the left or the right. At the start of the game, all of the segments are adjacent to each other and heading in the same direction. Thus, they appear to be a single centipede. As the game progresses, the segments may head in different directions and no longer appear as a single centipede.

The zapper moves left and right in response to the player pressing the left and right arrow keys. The zapper shoots a missile by hitting the space-bar. If a missile hits a mushroom, the mushroom disappears and the player earns one point. If a missile hits a centipede segment, the player earns 5 points. Also, the segment that was shot disappears and a mushroom appears in its place. The centipede moves its remaining segments, as before. If a segment in the middle of the "centipede" is hit, the segments between the new mushroom and the "tail" of the centipede will run into the newly created mushroom and turn around. Thus, the net effect is that there will now appear to be two "smaller centipedes" headed toward the zapper.

The game ends either when the zapper shoots all of the segments or when a segment runs into the zapper. In either case, a message is displayed to the user indicating that the game is over and showing the player's final score.

Implementation Details The game will include a black background, a score-keeping mechanism, a zapper, mushrooms, and all those centipede segments. We provide some suggestions on how to design and implement all of these pieces of the program below *so read through all of this handout before starting anything!*.

The starter folder includes six mushroom images and two segment images. You only need to use one of each. ("`shroom2.gif`" is our personal favorite.) Feel free to use the others if you want to add a little variety or animation to your program. Also, feel free to edit and include your own images, although it will be easiest if you stick to images that are 16x16 pixels.

The scorekeeper should display the score at the bottom of the screen. You must be able to increase the score when a missile hits a segment or a mushroom.

The zapper is an object that will appear at the bottom of the screen. It must provide methods for responding to the player's key strokes. Thus, a zapper should be able to move left, move right, and shoot a missile. If the zapper is hit by a centipede segment, it should stop being able to move or fire. If you want, you can make the zapper disappear in some interesting way.

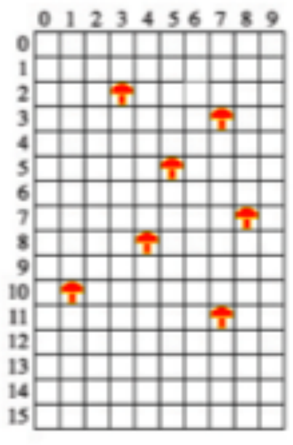
The missiles shot by the zapper are active objects. They should move up the screen and stop when they reach the top, hit a mushroom, or hit a segment.

The centipede has a different representation than our Nibbles snake. In particular, a centipede is an active object that keeps a collection of segments inside it. Each segment can be thought of as a separate entity that has a `VisibleImage` associated with it and it will keep track of where it is and which direction it is moving. The segments are all initially placed next to each other and facing to the right, so that they appear to move as a single, long "centipede". The `Centipede` animates the segments by making each segment take a step. (The segments themselves are NOT active objects.) Think of this like our chain example. The segments are individual objects that are animated using a single active object, the centipede. A segment takes a step by moving a fixed distance in the direction that that segment is heading. The centipede moves all of the segments and then should pause only after all of the segments are moved. *Do not use smooth scrolling or pause between moving individual segments.* If you do, the centipede may not appear to move properly – it will stretch out or shrink, or individual segments may move out of alignment and not look quite right.

If a segment is about to step off the screen or hit a mushroom horizontally, it should move down the screen by one row of mushrooms. The segment should also change its direction so that it will move in the opposite direction the next time it is asked to take a step. Don't worry about the centipede running into segments when it moves down a row (this both makes the implementation more straightforward as well as keep the centipede from getting stuck). When a segment is hit by a missile, it should disappear from the screen.

You need to be careful about how you keep track of the segments in the centipede. We suggest you use an array of `Segment` objects. Most importantly, DO NOT TRY TO DELETE SEGMENTS THAT HAVE BEEN HIT FROM THE ARRAY AND SHIFT OTHER ELEMENTS IN THE ARRAY OVER TO FILL THE HOLE. If you do this, bad things may happen if a missile tries to rearrange your array to delete a segment that has been hit at the same time that the segment is being moved across the screen. Instead, when a segment is hit, just set a variable within the object representing the segment to indicate that it is dead and hide it from the screen.

As you can see from the demo, mushrooms never overlap and are placed inside the squares of a grid laid over the canvas. We suggest using a two dimensional array of `VisibleImage`s to store the mushrooms. Make the array large enough to hold mushrooms laid out over the whole canvas. For reasons similar to those described above for segments, you should fill *all* entries in your array with mushroom visible images. Only show images for mushrooms that should be visible, and NEVER DELETE ELEMENTS FROM THE ARRAY OR SET ENTRIES TO NULL. Intuitively, if there is enough room for a grid of mushrooms with 15 rows and 10 columns in the canvas, you would create a 15x10 array and store a mushroom in each cell. The following shows how the screen would look if that grid were displayed while all but 7 mushrooms were hidden:



To "remove" a mushroom that is hit by a missile, simply use the `hide` method to hide the mushroom visible image. The `isHidden()` method may be handy for determining if a specific mushroom is visible on the screen. When a missile hits a centipede segment, you should make the mushroom in the grid location underneath the segment appear after hiding the segment.

Your program will comprise several classes, corresponding to the objects just described.

CentipedeController The controller will set up the game. It will also accept user input in the form of key strokes. In response to the different key presses, it should invoke methods of the zapper, making it move or shoot. We have provided the skeleton for listening to the user's key presses. You should set up the game in begin and fill in the lines where the `Zapper`'s methods need to be invoked. Be sure to add any GUI components to the window before creating the zapper as otherwise the zapper may lie below the final location of the canvas.

Zapper The Zapper moves in response to each key press of the left and right arrow keys. The game plays more easily if each key press moves the zapper by the width of a mushroom. When the spacebar is pressed, the `CentipedeController` will invoke the `Zapper`'s shoot method to launch a missile.

Segment A `Segment` manages one segment of the centipede, and describes its behavior. It should keep track of the direction it is traveling and record whether or not it is still alive. A segment can take a step, and it should be able to die. The `step` method should determine how far and in which direction the segment should move, and know how to turn around when it runs into the edge of the screen or into a mushroom. The `Segment` class should also have functionality to "kill" the zapper if the segment's image overlaps the zapper.

Centipede A `Centipede` is an `ActiveObject` that keeps an array of `Segment`s. Its `run` method will repeatedly make each segment take a step. The best way to make the centipede move is to have the `step` method in the `Segment` class do all of the work, so the `Centipede` class should be very simple.

Field The `Field` class holds the two-dimensional array of mushrooms. The `Field` constructor should create all of the mushrooms, and the class should provide methods that make it easy to implement the interactions between mushrooms and segments, and mushrooms and missiles.

Missile A `Missile` is an `ActiveObject` that moves up the screen, stopping either when it reaches the top or when it hits something. Note that to achieve this behavior, the missile needs to know about the `Field` and `Centipede`, so that it can determine whether it hits a mushroom or segment, respectively. Life will also be easier if both the gun on the zapper and the missile will be something other than a line, e.g. a FilledRect or FilledOval so you can more easily detect if it overlaps something.

The easiest way to provide information to the `Missile` is to have the `Zapper`'s `shoot` method take the `Field` and `Centipede` as parameters, and then pass them to the `Missile`'s constructor when that method creates the new missile.

(Alternatively, you could store the field and centipede in instance variables in the zapper, but this is a little tricky to set up because the centipede will also need to store the zapper so that its segments will be able to kill the zapper. This circularity between the zapper and the centipede can be resolved in a number of ways, but just passing the necessary information to shoot is the easiest.)

ScoreKeeper The `ScoreKeeper` class displays the score on the screen. Note that the `Segment`s and the `Field` should probably know about the `ScoreKeeper`, as they will likely need to inform it to increase when a segment dies or mushroom disappears.

You may also want to define other classes if you believe they will simplify your design.

The Design As indicated in the heading of this document, you will need to turn in a design plan for your Centipede program well before the program itself. You should include in your design a sketch of each class including the types and names of all instance variables you plan to use, and the headers of all methods you expect to write. You should write a brief description of the purpose/function of each instance variable and method.

In addition, you should provide internal comments for any method whose implementation is at all complicated. In particular, if a method is complicated enough that it will invoke other methods you write (rather than just invoking methods provided by Java or our library), then include comments for the method so that we will see how you expect to use your own methods.

From your design, we should be able to find the answers to questions like the following easily:

1. How and when is the scorekeeper updated?
2. What information is passed to the constructor for `Missiles`, `Segments`, etc.?
3. How do you find the mushroom appearing at a certain location on the canvas in the `Field`'s two-dimensional array? (You will need to use this operation in several places.)
4. How does a `Segment` decide when to turn around? What methods does the Field provide so that the segment can decide whether it is about to step into a mushroom?
5. What operations should the `Centipede` and the `Field` provide to help Missiles determine when they hit and kill segments and mushrooms?

The more time you spend on the design, the easier it will be to complete the program.

Constants The following constants appear in the starter code. Using them will ensure that your segments move at a reasonable speed and are able to turn correctly.

In the `Field` class:

```
// dimensions of the canvas
private static final int CANVAS_WIDTH = 400;
private static final int CANVAS_HEIGHT = 512;

// size of mushroom pictures
private static final int SHROOM_SIZE = 16;

// dimensions of the mushroom array
private static final int NUM_ROWS = CANVAS_HEIGHT / SHROOM_SIZE;
private static final int NUM_COLS = CANVAS_WIDTH / SHROOM_SIZE;
```

In the `Segment` class:

```
// width of segment image
private static final int SEGMENT_WIDTH = 16;

// number of pixels a segment should travel in the X direction
private static final int SEGMENT_STEP_X = 4;

// number of pixels a segment should move in the Y direction
private static final int SEGMENT_STEP_Y = SEGMENT_WIDTH;
```

In the `Centipede` class:

```
// offset between segments when first created.
private static final int SEGMENT_OFFSET_X = 12;

// pause time between moving all of the segments
private static final int CENTIPEDE_PAUSE = 25;
```

In the `Missile` class:

```
// Distance missile should move after each iteration
private static final int DISTANCE_TO_MOVE = 16;
private static final int PAUSETIME = 33;
```

In the `Zapper` class:

```
// width of the zapper
private static final int BODY_WIDTH = 16;
```

Feel free to adjust these constants as you wish. However, `FIELD_WIDTH`, `SHROOM_SIZE`, `SEGMENT_OFFSET_X`, and `DISTANCE_TO_MOVE` should always be multiples of `SEGMENT_STEP_X`. If they are not, your centipedes may appear to separate or bunch up after running into obstacles, because some segments will turn around when they are a pixel or two further away from the obstacle than others.

Implementation Order Begin copying the starter folder as usual. We strongly encourage you to proceed as suggested below to ensure that you can turn in a running program. While a partial program will not receive full credit, a program that does not run at all generally receives a lower grade. Moreover it is easier to debug a program if you know that some parts do run correctly.

- Experiment with the demonstration program.
- Write a program that draws a `Zapper`. Run the program in a window that is 400×512 pixels in size.
- Add code to make the `Zapper` respond to the user's right- and left-arrow key strokes. Do not worry about shooting at this point. (You will need to click in the window when it first opens.)
- The next step is to add the mushrooms by creating a `Field` before creating the `Zapper` in `begin`. Implement the `Field` constructor to create a two-dimensional array of `VisibleImage`s. Initialize all entries with mushrooms, hide them all, and then randomly select perhaps 60 or 70 of them to show. They should not appear in the top few rows or bottom few rows so that they do not interfere with the centipede as it starts to move or with the zapper.
- Next, start working on `Segment` and `Centipede`. Implement a basic `Segment` class that provides a `step` method that moves a segment's image a little to the right. Write a `Centipede` that creates a single segment and makes it step across the screen. (It will work best if you create the centipede in `begin` after creating the field and zapper).
- Change the `Segment`'s `step` method to move left and right properly, moving down the height of a mushroom when it reaches the edge of the canvas. Do not worry about running into mushrooms yet. Change the `Centipede` to create and move a whole array of `Segments`. The x coordinates of consecutive segments should be `SEGMENT_OFFSET_X` pixels apart. Our centipede is created off

the screen and gradually steps into view, but this is not required – you can simply create the segments along the top of the canvas. [*Warning:* If you do create the centipede off of the screen you will have to be careful when you try to determine if the segment is about to hit a mushroom as a location off the screen will result in an illegal subscript of the array in the field.]

- Make sure you are passing the mushroom field into the `Centipede` and `Segment` classes, and modify the `Segment` class to turn around when it is about to step into a visible mushroom. Notice that the centipede only changes direction when it is to hit a mushroom horizontally. Don't try and handle the case of the centipede running in to a mushroom when it steps down after either hitting a wall or another mushroom. The `Segment` will need to ask the `Field` whether a specific (x,y) location on the canvas corresponds to a visible mushroom in your 2D array. We suggest you declare this method in Field as follows:

```
public boolean shroomIsVisible(double x, double y)
```

  You may implement this test in any reasonable way. You may find two helper methods that we have provided in Field for converting screen locations to array indices useful:

```
// Convert a y coordinate in pixels to the corresponding
// row in the mushroom array
private int getRow(double y) {
    return (int)(y / SHROOM_SIZE);
}
// Convert a x coordinate in pixels to the corresponding
// column in the mushroom array
private int getColumn(double x) {
    return (int)(x / SHROOM_SIZE);
}
```

  For example, `getColumn` will convert the x coordinate 50 to the column index 3, and `getRow` will convert the y coordinate 100 to the row index 6. Thus, the mushroom image corresponding to the screen location (50,100) on the screen is stored in your 2D array at row 6 and column 3.

  Note that these two methods may return rows and columns that are not within the bounds of the array if x and y are negative or larger than CANVAS WIDTH and CANVAS HEIGHT. So, be sure to check that the x and y coordinates passed into shroomIsVisible are within bounds before proceeding. If they are not, we can conclude that no mushroom is visible at that location.

- Change the zapper to shoot missiles. Make them move up to the top and disappear. Don't worry about having the missiles hit anything yet.

- Make the missiles hit segments. First, add a method to the centipede class allowing the missile class to check if it's missile has hit any of the segments (and you may also want to take appropriate action for the centipede and segment in that method). Then, make sure to pass the centipede to the `Missile` constructor when you create it. The missile shot should, as it is moving, be continually asking the centipede if it has hit and killed any of the segments that are still alive. If the missile does hit a segment, make it disappear. (Do not worry about hitting mushrooms or making them appear yet.)

- Change the `Missile` class so that, in addition to asking the `Centipede` if it overlaps any living Segment, it asks the Field if it has hit a visible mushroom. If the missile does hit a mushroom, it should disappear. A single missile should not be able to kill both a mushroom and a segment, even if it hits them at the same time. So, be sure to check if a mushroom was hit only if no segments were hit.
- Modify the `Segment` so that the mushroom under it becomes visible when the segment becomes hidden. Note that once you add this feature, killing a segment may make some of the segments overlap or completely cover each other, depending on how you implemented moving and turning. This is perfectly fine, and in fact makes the game a little more interesting to play...
- Change `Segment` to kill the `Zapper` when it runs into it, and set up the score keeper. Getting everything to stop when the zapper is killed or when the centipede is completely destroyed is a bit trickier than it looks. Be sure to allow movement or firing of the zapper only if it is still alive. Be careful to update the score keeper only if everything is still alive. Otherwise the message indicating a win or loss will be replaced by one just updating the score.

There is a great deal of functionality to aim for in this test program. *Do not worry if you cannot implement all of the functionality.* Get as much of it working as you can. As we have done throughout the semester, we will consider both issues of correctness and issues of style when grading your program. It is always best to have full functionality, but you are better off having most of the functionality and a beautifully organized program than all of the functionality with a program that is sloppy, poorly commented, etc.

Extra Credit Since we have deliberately left out many features of the original Centipede game, there are clearly many additional features you could add to your program. If you notice below the total only adds up to 98 points, so you must do some extra credit. In addition, you can earn up to 5 points of additional extra credit for a total of 105 points possible. We will give 1-2 points for each extension.

For each extra credit item that you do, list it in the comments at the top of the `CentipedeController` class or you may not get credit for it!

Some possible extensions are:

- Restrict the zapper to shooting only one missile at a time.
- Make the mouse control the zapper.
- Permit the zapper to move up and down in the bottom few rows of the screen. The Zapper can then avoid segments that reach the bottom. Such segments should "teleport" several rows back up the screen and continue to move.
- Make it require multiple shots to kill a mushroom.
- Reincarnate the centipede by moving it back up to the top when all the segments have all been killed.
- Make the centipede's head look different than the segments. The starter provides additional images for this. If the centipede splits, how do you make both smaller centipedes have heads?

- If you play the original game, fleas drop down the screen creating new mushrooms, spiders run around near the zapper trying to sting it, and scorpions poison some mushrooms. Feel free to implement any of these other creatures. (A very simple addition would be to just randomly select mushrooms to appear occasionally, even if you do not implement the whole flea.)
- Use multiple images for the centipede segment to animate its motion.

Turning it in Your design should be turned in on paper in class or e-mailed to the professor. Keep a copy for yourself since we won't return it to or give you feedback on it until after the program is due. When your work is complete you should deposit in the dropbox folder Before turning it in, please be sure that your folder contains all of the .java files, .class files, etc. as we can't give credit for anything we don't receive.

Before turning it in, make sure the folder name includes your name and the phrase “TP2”. Also make sure to double check your work for correctness, organization and style.

This program is due at midnight on December 8th, the last day of classes. While you may turn your program in late, you will be assessed a 10 point penalty for each day it is late. I.e., if you turn it in by midnight on December 9th, you will receive a 10 point penalty. If you turn it in at midnight on December 10th, you will receive a 20 point penalty, etc.

Table 1: Grading Guidelines

Value	Feature
	Design preparation (20 pts total)
	Syntax Style (14 pts total)
6 pts.	Descriptive and helpful comments
2 pts.	Good names
2 pts.	Good use of constants
2 pts.	Appropriate formatting
2 pts.	Appropriate use of public/private
	Code quality (26 pts total)
5 pts.	Conditionals and loops
5 pts.	General design/Efficiency issues
5 pts.	Parameters, variables, and scoping
5 pts.	Appropriate use of arrays
6 pts.	Misc.
——	**Correctness (38 pts total)**
	Basic setup (6 pts)
3 pts.	general setup (score keeper, etc.)
3 pts.	random mushrooms
	Zapper (6 pts)
2 pts.	drawn correctly
2 pts.	moves left, right
3 pts.	shoots missiles
	Missiles (8 pts)
2 pts.	missile moves up
2 pts.	missile stops at top
2 pts.	missile can hit segments
1 pts.	missile can hit mushrooms
1 pts.	segments turn into mushrooms
	Segments and Centipede (12 points total)
3 pts.	segment moves right or left
3 pts.	segment turns around at edges
3 pts.	segment turns around at mushrooms
3 pts.	segment moves down when it turns
	Ending the Game (6 points total)
1 pts.	Game ends when all segments are hit
1 pts.	Game ends when zapper is hit
4 pts.	Scorekeeper keeps score correctly
2 pts.	*Miscellaneous points (for EC)*
100 pts.	**Total**
5 pts.	**Extra credit**