CS106L
Fall 2008

Handout #24a
November 20, 2008

Assignment 2a: MyPhoton

Due December 7, 11:59 PM

Introduction

It's time to pull together the results of your `grid` assignment and your newfound command of inheritance techniques to create a puzzle game: *MyPhoton*. Once you've completed the assignment, you'll have a pretty impressive piece of software that will showcase the full extent of your newfound C++ skills, especially with regards to inheritance.

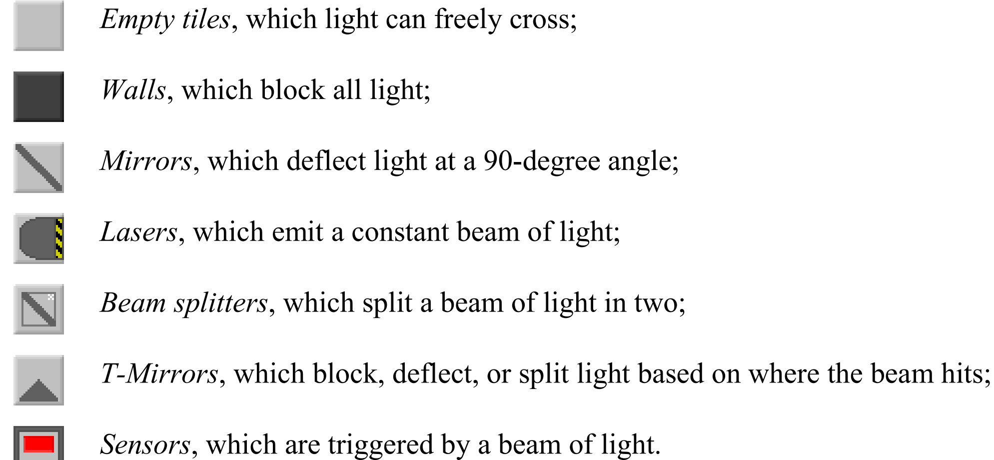
A Word of Warning to Mac Users

As of November 20, 2008, the graphics package I've developed for this assignment does not work on Macintosh computers – apparently the CS106 graphics libraries aren't as portable as advertized. I'm terribly sorry about this and am currently trying to get a replacement graphics package ready. In the meantime, if you have a Mac and want to work on this assignment, you'll need to work on one of the Stanford cluster computers.

Overview

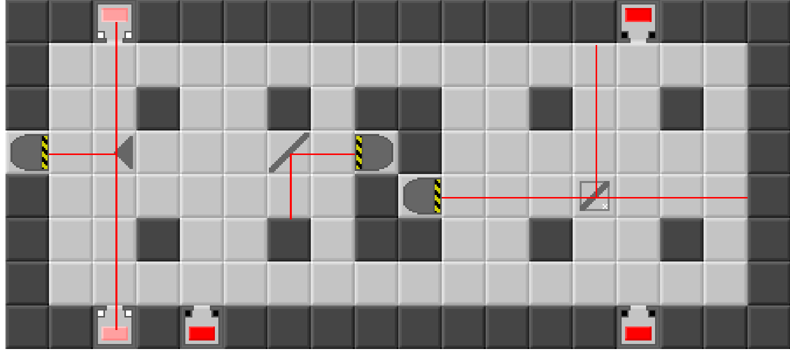
MyPhoton is a puzzle game combining lasers and logic. Each MyPhoton puzzle is a grid containing some number of lasers and light sensors, along with an assortment of mirrors. To solve the puzzle, the player must trigger all of the light sensors by directing the laser light appropriately.

Each grid tile is one of seven different types:



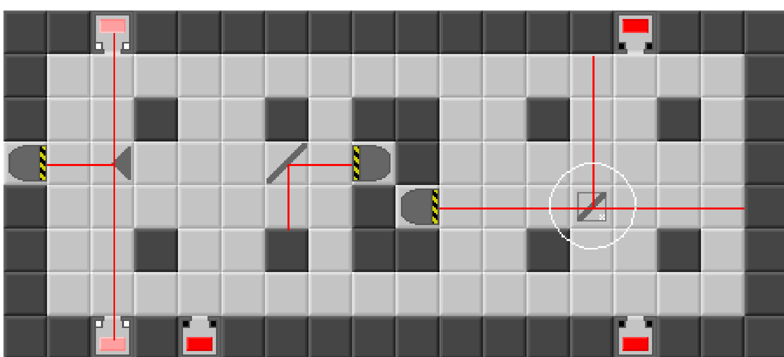
- *Empty tiles*, which light can freely cross;
- *Walls*, which block all light;
- *Mirrors*, which deflect light at a 90-degree angle;
- *Lasers*, which emit a constant beam of light;
- *Beam splitters*, which split a beam of light in two;
- *T-Mirrors*, which block, deflect, or split light based on where the beam hits;
- *Sensors*, which are triggered by a beam of light.

Here is a sample MyPhoton level demonstrating each of these components:

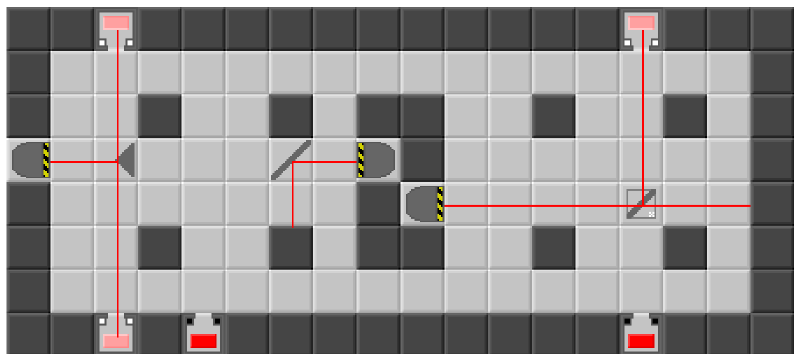


This level contains three lasers – one on the left wall and two in the center – as well as a single mirror, beam splitter, and T-mirror. Currently, only two of the five light sensors have been triggered, and it's up to the player to figure out how to trigger them all.

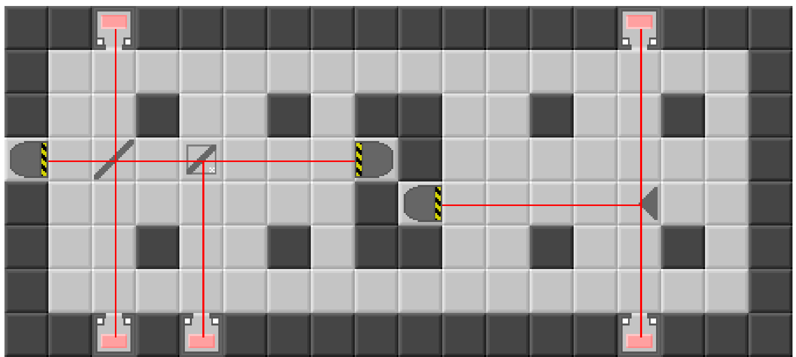
In MyPhoton, the position and orientation of each of the level's lasers, walls, and sensors is fixed. The player can only change the position (but not the orientation) of the level's *hardware* – that is, mirrors, T-mirrors, and beam splitters. To change the path the laser takes through the level, the user can reposition hardware by moving a piece of hardware into an empty square or by swapping the positions of two pieces of hardware. For example, in the above example, suppose that the user clicks on the beam splitter piece in the right half of the puzzle. This highlights the piece, as shown here:



Now, if the user wants to move this piece one square to the right, she can do so by clicking that empty square. This results in the puzzle looking as follows:



Now, three of the five sensors are lit. The player continues to swap tile positions until she has successfully triggered all of the light sensors. To solve this puzzle, the user will need to get the pieces into the following configuration:

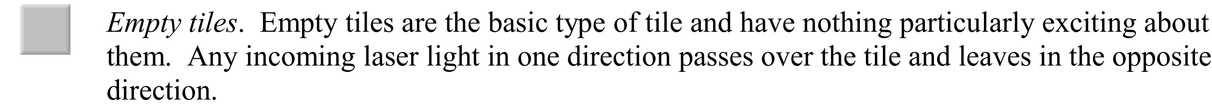
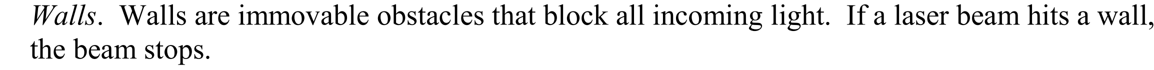
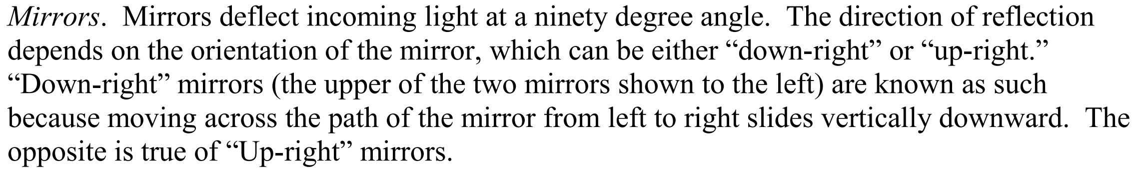
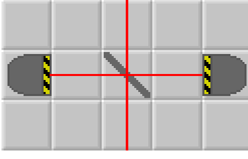


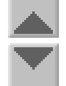
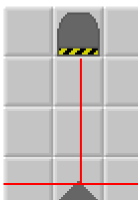
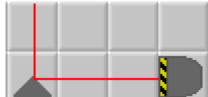
This setup is rather complicated, but manages to trigger all of the sensors.

In this assignment, you will implement the back-end of the MyPhoton game – the code that maintains the state of the board, propagates laser light through the world, and updates the world in response to user input. While this may seem a bit tricky, the amount of code you actually need to write for this assignment is not particularly great and by choosing a good design the resulting code is straightforward.

Types of Tiles

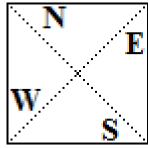
The seven tile types mentioned above are explained in more detail below:

	*Empty tiles*. Empty tiles are the basic type of tile and have nothing particularly exciting about them. Any incoming laser light in one direction passes over the tile and leaves in the opposite direction.
	*Walls*. Walls are immovable obstacles that block all incoming light. If a laser beam hits a wall, the beam stops.
	*Mirrors*. Mirrors deflect incoming light at a ninety degree angle. The direction of reflection depends on the orientation of the mirror, which can be either "down-right" or "up-right." "Down-right" mirrors (the upper of the two mirrors shown to the left) are known as such because moving across the path of the mirror from left to right slides vertically downward. The opposite is true of "Up-right" mirrors. An interesting interaction of lasers and mirrors arises when two different lasers hit opposite sides of the laser. In this case, the two beams will deflect in different directions, as shown here: 

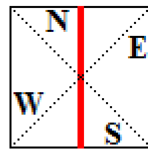
<table>
<tr><td> </td><td>Lasers. Lasers emit a beam in the direction in which they are facing. However, they are otherwise opaque, and any beam that hits a laser in any direction is blocked.</td></tr>
<tr><td> </td><td>Beam splitters. Beam splitters split a beam into two parts – one beam traveling in the same direction as the original beam, and one deflected at a 90 degree angle from the first. As with regular mirrors, the direction that the beam is deflected depends on the orientation of the mirror. Beam splitters can be thought of as a mirror that also lets light continue normally.</td></tr>
<tr><td></td><td>T-Mirrors. T-mirrors are the most complicated of the components. The behavior of light as it hits a T-mirror depends on which direction the beam hits from. If the beam hits the point of the T-mirror directly, the beam is split into two beams, each moving at 90 degree angles from the original. This is shown below:

If the beam hits the T-mirror from either of its two shorter sides, then the beam is deflected in the direction of the point:

Finally, if the beam hits the back side of the T-mirror, the beam is blocked entirely.</td></tr>
<tr><td> </td><td>Sensors. Sensors act as laser receivers. If a laser hits a sensor in any direction except for the direction that contains an opening, the laser is blocked. Otherwise, if the laser hits in the direction of the opening, the sensor is “triggered” and changes is picture to an “on” state. The sensor will turn off as soon as the laser source is removed.</td></tr>
</table>

Implementing the Lasers

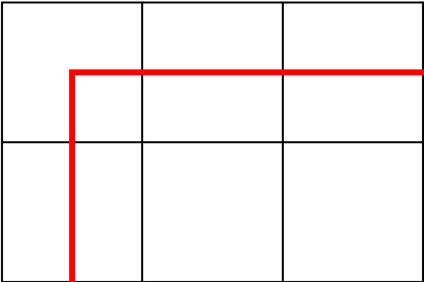
The first challenge that you will have to face when implementing the MyPhoton back end is how to represent the lasers. While there are several ways that you can go about doing this, one method stands out as the simplest to implement. The trick is to break down laser beams into smaller laser segments called *laserlets* which store a position and direction. For each tile on the board, we can divide that tile into four regions – north, south, east, and west – as shown here:



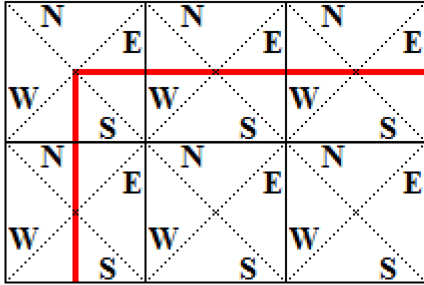
Now, suppose that this tile has a laser beam running through it from north to south, as shown here:



Then we can segment this laser beam into two laserlets, one in the north of the tile and one in the south. Now, if we have a laser beam that spans multiple tiles, we can describe that laser as a collection of laserlets occurring in different directions in those different tiles. For example, consider the following laser beam:

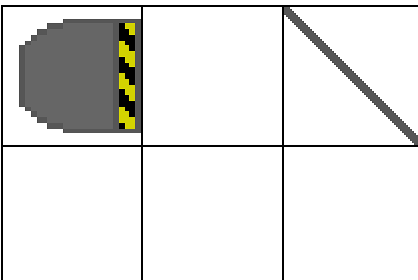


Here, the beam abruptly turns 90 degrees in the upper-left corner. This could be from some sort of interaction with the contents of the tile there – perhaps it hits a mirror, for example. Now, if we overlay this image with our segmentation diagram, we see that this laser beam can be segmented into the following pieces:

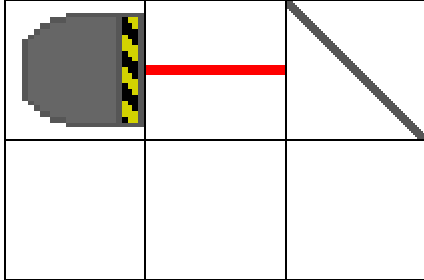


We can keep track of this laser by simply storing the position and orientation of each of these laserlets.

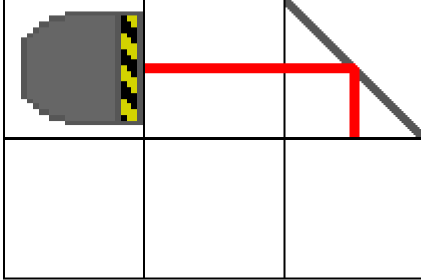
Representing lasers as a collection of laserlets greatly simplifies the rest of the implementation of the backend because it allows us to generate the path of a laser in a series of simple discrete steps. For example, consider the following setup:



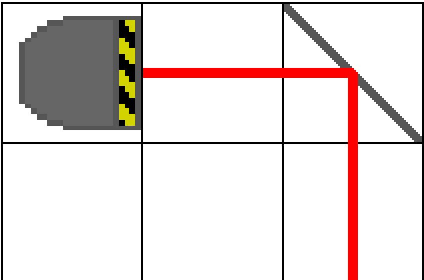
Here, the object in the top-left corner is a laser source, and the object in the upper-right corner is a mirror. Suppose that we want to compute the path the the laser light will take as at leaves the laser source. We begin by considering the laserlet emitted from the laser source in the upper corner. Since the laser source is facing east, the generated laserlet will attempt to enter the tile one square to the right of the source while moving easterly. Now, since the tile to the right is an empty tile, the laser will pass through the square unhindered, and we will have this result:



If you'll notice, at this point the head of the laser beam is a laserlet (not pictured above) that attempts to enter the top-right square while moving east. Consequently, we compute the effect of the laser beam entering the mirror tile while moving toward the east. Since this light will bounce off the mirror, the light will enter from the west and leave from the south. This yields:

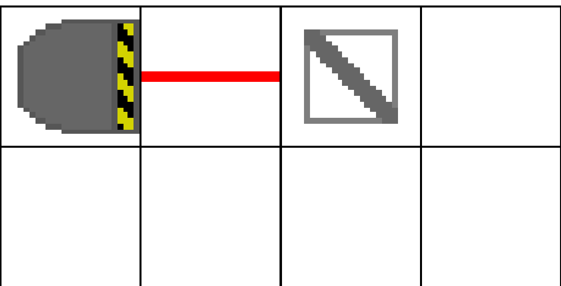


Finally, we determine what the effect of shining a laser from the north is on an empty tile. This simply causes the laser to pass through the square, yielding the following:

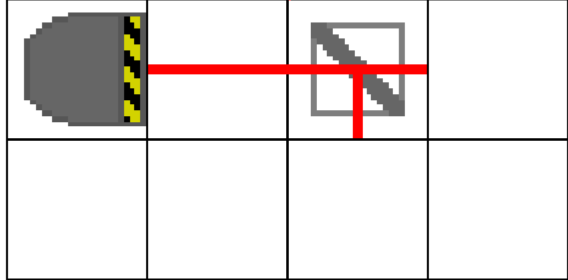


This is exactly the path we'd expect the laser to take. By now the advantages of the laserlet approach should seem more obvious. By quantizing the laser into discrete segments, we can compute the path of the laser by simply asking the square the laser is about to enter to continue computing the path. If each tile is capable of determining the effect of laser light entering it, then given only information about where the laser sources are, we can construct the entire path of that laser.

If you'll notice, at each step in this process, we only needed to keep track of a single laserlet – the laserlet at the front of the beam – in order to determine what path the rest of the laser beam would take. We will refer to this laserlet as the *head* of the beam. In the above example, the laser had a unique head at all times, but this is not always the case. Suppose, for example, that the head of our current laser is about to travel through a beam splitter, as shown below:

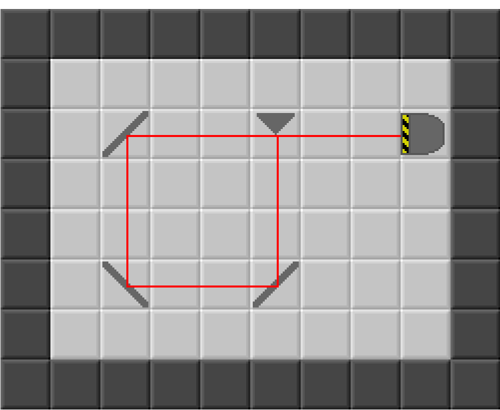


Then in this case, the laser will enter the square containing the beam splitter and will bifurcate, as shown here:



Notice that our laser now has two heads – one moving south and one moving east. This will not cause any problems with our process for generating the path of the laser, but will require us to keep track of a set of many heads, rather than just a single head.

An important detail to note is that when working with this approach, you must be very careful to check for cycles in the path of the laser. For example, consider the following layout:



Here, the initial beam from the laser hits the T-mirror on a side, deflecting it downward. The beam then reflects off of the the three mirrors and hits the other side of the T-mirror, causing it to deflect downward once again. Using the laserlet approach illustrated above, we will be caught in an infinite loop shining the

laser. To resolve the problem, we can keep track of the laserlet heads in a "used heads" set. If at any point we find a laserlet head that's in this set, we can safely ignore it.
The pseudocode for the algorithm outlined above looks like this:

```
Construct a set of all heads (e.g. find all lasers and create heads for them)
Repeat while more heads exist:
    Choose one of the unchosen heads.
    If the head is in not in the used set:
        Add the head to the used heads list.
        Compute the effects of the head moving through its square.
```

The last step in this process depends on the type of tile that the laser is moving through, but thanks to C++ inheritance, if you model the different tile types as a class hierarchy such that each tile can respond to a laserlet entering that tile, it shouldn't be particularly difficult.

The Starter Code

Unlike previous assignments, MyPhoton comes with a substantial amount of starter code to handle graphics, sound, and user input. Because C++ does not have a standard graphics or sound package, the starter code is written using the CS106 graphics libraries. For those of you not in CS106X, you can obtain a copy of these libraries from the CS106X course website (cs106x.stanford.edu).

Note that while the starter code uses the CS106 libraries for graphics, sound, and input, in your code for this assignment you <u>must not</u> use any of the CS106X library code. Your submission should consist solely of standard C++.

The provided code is broken down into several main components, each of which are described below:

Direction Support: Because you will need to keep track of the directions of various objects (lasers, tiles, etc.), I've provided a pair of types, `directionT` and `diagonalDirectionT`, which are enumerations representing the cardinal directions `North`, `South`, `East`, and `West`, as well as the two diagonals of the square. `directionT` is useful for represent laserlet positions, while `diagonalDirectionT` is ideal for storing the direction of mirror tiles. There are several supporting functions that take care of a good number of common operations on `directionT`s, such as rotation, reflection off of mirrors, etc.

The `grid` Class: Since MyPhoton puzzles are two-dimensional, to complete this assignment you will need a working `grid` class. If you completed Assignment 1, feel free to use your own `grid` implementation. Otherwise, I've provided a working version of the `grid` class that meets the specifications of the previous assignment.

World Loading Code: MyPhoton levels are stored in a special format that are somewhat easy to read and write by hand (the format is detailed in the end of the handout). Because some of the loading code is tedious, I've provided a skeleton implementation of a world loading function called `LoadWorldSkeleton`, which parses the file and does error checking but which does not actually do anything with the data it reads. You should modify this code so that the rest of your program can read the puzzle files. Note that the world-reading code uses exception-handling for error reporting, so you will need to check that your program doesn't leak any resources if the function throws an exception.

`PhotonController`: `PhotonController` is an interface for objects that can interact with the starter code's graphics and input package. The `PhotonGraphics` module (described below) will send messages to a `PhotonController` object in order to translate from user clicks into actual in-game events. The

interface for `PhotonController` is as follows:

(All member functions declared here are pure virtual unless indicated otherwise)

`void onGameStart()`	Called when the game first starts. This member function should perform any needed initialization.
`void onGameEnd()`	Called when the game shuts down. This member function should perform any needed cleanup.
`void drawBoard()`	The `PhotonGraphics` object will call the `drawBoard` function when the puzzle needs to be redrawn. This function should redraw the entire puzzle. While this is not particularly efficient, it's simple to implement.
`bool trySelectTile(int x, int y, bool firstTile)`	Called whenever the user clicks on a tile. The function will be passed as parameters the x and y coordinate of the click in game coordinates, as well as whether or not this was the first or second tile selected by the user. These coordinates are guaranteed to be in bounds. The function should return true if the selection is valid and false otherwise.
`void swapTiles(int x1, int y1, int x2, int y2)`	After the user selects the second tile successfully, the `PhotonGraphics` object will call `swapTiles`, passing in the x and y coordinates of the source location and destination location. The coordinates are guaranteed to have been the coordinates approved of by `trySelectTile`. The `PhotonController` object should then exchange the positions of the tiles in those two positions.
`bool isPuzzleSolved()`	This function should return whether or not the player has solved the puzzle.

In this assignment, you should not modify `PhotonController`. Instead, to create a `PhotonController`, create a new class that derives from `PhotonController`.

`tileT`: When working with the graphics package, you will need to be able to specify what tiles to draw in which locations. To specify which tile picture to use, I've provided a `tileT` enumerated type along with utility functions that can return the proper `tileT` for a given scenario. Consult `tile.h` for more information.

`PhotonGraphics`: The `PhotonGraphics` class is a fully-functional graphics module for MyPhoton that also acts as the interface between your program and the outside world. `PhotonGraphics` is a singleton class, meaning that you cannot instantiate your own copies of the `PhotonGraphics` class. To get the single instance of `PhotonGraphics`, use the `PhotonGraphics::getInstance()` function.

The member functions of `PhotonGraphics` are as follows:

`static PhotonGraphics* getInstance()`	Returns a pointer to the unique instance of the `PhotonGraphics` object.
`void setController(PhotonController *c)`	Sets the `PhotonController` object that will be responsible for driving the game. The `PhotonGraphics` object queries this object for information and will instruct it to perform various functions needed for the game to work properly.
`void resizeWorld(int width, int height)`	Informs the `PhotonGraphics` module how large the world is. These dimensions should be greater than zero and less than the constant `PhotonGraphics::MAX_DIMENSIONS`. Provided that you use the starter code's world loading routines, you should not need to worry about breaking these invariants.
`void drawTile(tileT tile, int x, int y, const set<directionT> &lsr)`	Draws the tile with game coordinates (`x`, `y`) using the picture specified by `tile`. The tile will have laserlets drawn on top of it based on the contents of the `lsr` set.
`void runGame()`	Given the controller specified in a previous call to `setController` and a world whose dimensions were specified in `resizeWorld`, begins listening to user input and playing the puzzle. The `PhotonController` specified in the call to `setController` will receive messages from the `PhotonController` object about how what events are going on in-game.

Task Breakdown

This assignment has a surprisingly large amount of supporting code, and consequently you do not need to write very much code for this assignment. However, there are many tasks that you will need to perform in order to get this program up and running. Below I've listed a set of smaller tasks – several of them purely conceptual – which you can use as guidelines for getting started on the project. This is by no means the only way to write this program, but I believe it is the easiest approach.

1. **Determine what operations you need to perform on tiles**. MyPhoton is a tile-based game and part of your goal will be to implement those tiles. Think about what sorts of functionality you will need those tiles to have. As a friendly hint, here's a summary of the sorts of things tiles will need to be able to do:
    1. *Determine the effects of lasers* – If a laser hits this tile, will it trigger a sensor? What directions, if any, will the lasers leave this tile?
    2. *Determine mobility*. The player can move some tiles but not others. Can this tile be moved? Is this a tile that another piece can be moved into?
    3. *Display graphics*. You will need to draw the tiles to the screen. Which picture should this tile use? What laserlets are present at this tile?
    4. *Detect victory*. Some tiles (sensors) need to have light shining on them for the game to be over. Is this tile contributing to a victory state? Does it prevent a victory state?
2. **Determine how to model those operations**. Because you will be working with a collection objects with a common set of operations but different actions associated with those operations, I strongly encourage you to model the tiles using inheritance. See if you can figure out how to translate the above operations into a set of classes and virtual functions.
3. **Implement a dummy `PhotonController`**. Write a stub implementation of a class deriving from `PhotonController` and see if you can get it plugged into the `PhotonGraphics` library. This will give you some familiarity playing around with the starter code and will make testing the rest of the program easier.
4. **Implement the basic tiles**. Some tiles like the empty tile and wall tile can be implemented in almost no time. Try implementing them completely and plugging them into the file loading code. Once you've gotten this under control, you can practice loading some of the sample puzzle files included with the project.
5. **Implement the laser propagation algorithm**. Implement the laser source tile and see if you can get the algorithm for propagating laser beams working correctly. If everything seems to be working fine, start adding more and more tiles to the mix.
6. **Implement the rest of the tiles**. Once you're at this point, you can do pretty much everything except for responding to user input.
7. **Implement the full `PhotonController`**. Allow the user to move tiles around and solve puzzles correctly. At this point, you're golden and just need to do some final tests to see if everything's working correctly.

Advice, Tips, and Tricks

This assignment is slightly more complicated than previous assignments because of the number of different pieces that all have to work together in order for the final program to work correctly. To help get you on the right track, here are some tips and tricks that might be useful:

- Don't worry about handling the cases where lasers fall off the edge of the world. You can handle this case if you'd like, but none of the world files will test this case.
- You can factor most of the behavior of the tiles into a single base class that keeps track of attributes like picture, mobility, etc. This will make writing the other tile classes much easier. Protected constructors will be your friend.
- The starter code that reads in the world file will throw exceptions if it encounters a problem in the source file. Make sure that if you allocate any resources in the code that reads in files, those resources aren't leaked or corrupted if the loader throws an exception. You are free to modify the code to have it not use exceptions if you feel most comfortable working that way.
- A beam splitter acts exactly like a mirror, except that the beam continues through the mirror in addition to deflecting. If you're clever, you can make the beam splitter implementation extremely simple by defining it in terms of the mirror.

Submission Instructions

Email any files you've created or modified to htiek@cs.stanford.edu. Then congratulate yourself – you've just made it through the final assignment of CS106L!

Appendix A: World File Format

In case you want to create your own MyPhoton puzzles, I've attempted to make the puzzle format as easy to read and write as possible. A sample puzzle is shown below:

```
15 15
W W W W SvW W W W W W W W W W
W . . . . . . . . . . . . . W
W Lv. . . . . . . . L>. . T>W
W . . . . . . . . . . . . . W
W . . . . . . . . . . . . . W
W . . . T>. . . . . L<. . . W
W . . . . . . . . . . . . . W
W . . . W L>. . . Tv. . . . W
W . . . . . . . . . . . . . W
W . . . . . . . . . . . M\. W
L>. . . T^. . . . . . . . . W
W . . . . . . . . W . . . . W
W . . . . . . B/. . . . . . W
W . . . . . . . . . . . . . W
W W W W W W W W W W W W W W W
```

The first line of the file contains two integers that correspond to the width and height of the world, including the walls that ring the world. The rest of the file consists of the individual tiles that compose the file. They are interpreted as follows:

- A dot (`.`) represents an empty tile.
- A wall is reprsented by a capital `W`.
- Lasers are represented by a capital `L` followed by the direction the laser points. The direction is indicated using one of the four characters `< > ^ v`, which look like arrows. That is, east and west are represented using angle brackets, north by a carat, and south by a lower-case `v`.
- Sensors are represented by a capital `S` followed by a direction character, much in the same way as the laser.
- T-Mirrors are represented by a capital `T` followed by a direction character that indicates which direction the point of the T-mirror faces.
- Mirrors are represented by a capital `M` followed by either `/` or `\` to indicate which direction the mirror faces.
- Beam-splitters are represented by a capital `B` followed by either `/` or `\` to indicate which direction the beam-splitter is oriented.

If you create any fun puzzles, send them my way and I'll be glad to look over them!

Appendix B: MyPhoton Puzzle Files

To help test your solution, I've provided a handful of prewritten MyPhoton puzzles. They are stored in the `Puzzles` subdirectory.

`empty`	A world consisting only of empty tiles.
`walls`	A world consisting only of empty walls.
`simple-laser`	A world consisting of a single laser.
`many-lasers`	A world consisting of several lasers.
`simple-mirror`	A world consisting of a single laser and mirror.
`simple-splitter`	A world consisting of a single laser and beam-splitter.
`simple-tmirror`	A world consisting of several lasers and T-mirrors.

`broken1`
`broken2`
`broken3`
`broken4`
	These are test files that contained malformed MyPhoton worlds. While all of the error-checking and error handling for file reading is handled in the starter code, you should probably test that your program doesn't crash when reading a malformed world. If it does, don't worry about it too much, but do attempt to fix it if possible.

`splitters`	A world consisting of a large number of beam splitters. Good for checking that you handle cycles.
`sensors`	A puzzle with a large number of sensors. Good for checking that you correctly test that all sensors have to be lit before the puzzle is solved.
`mirrors`	A puzzle with a large number of mirrors. Great for checking that your mirror code works correctly.
`tmirrors`	A puzzle that tests all four orientations of T-mirrors with respect to lasers.
`example`	The puzzle used as an example at the top of the handout. This tests all of the pieces and is a great way to check that you've gotten things working.
`shared-mirror`	A puzzle where laser light hits two different sides of a mirror.
`cycle`	A puzzle that contains a laser cycle. Your program should not crash on this input.
`izaaks`	A puzzle that's a bit of challenge – see if you can figure it out! Thanks to Izaak Rubin for the puzzle design.
`threeway`	A fun puzzle that can only be solved by splitting apart a single beam.