

## \* Assignment 5: Space Invaders \*

Instructor: Dr. Roberto A. Flores

### Goal

Practice GUI and event-driven programming.

### Introduction

Space Invaders is an video game whose purpose is to avoid the landing of a wave of alien invaders. As shown in Figure 1, the game begins with a wave of 50 invaders (5 rows of 10 invaders each), and a laser base at the bottom of the screen. Invaders move horizontally, initially to the right, until one of them reaches the edge of the screen. At that point all descend one level, invert their horizontal direction, and increase their speed.

The player controls the laser base, which can only move horizontally and shoot one missile at a time. Since invaders also fire missiles, players should avoid getting the laser base hit. The game is over when the invaders land or when the base is destroyed by an alien missile.

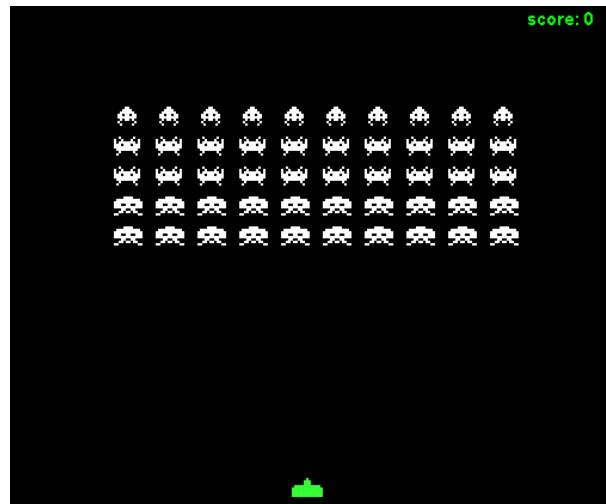


Figure 1: Initial state of a Space Invaders game.

### Playing the Game

#### Objective

From the player's perspective, the objective of the game is to prevent an invasion by shooting all invaders from a laser base while avoiding being shot. Reaching a high score (labeled "A" in Figure 2) is another objective prioritized against survival.

#### Fundamentals

The game starts with five rows of 10 invaders.

Aliens at the bottom 2 rows (labeled "E") are worth 10 points each; aliens at the middle 2 rows (labeled "D") are worth 20 points each; and aliens at the top row (labeled "C") are worth 30 points each. Thus a wave of aliens is worth 900 points.

Players can also shoot "mystery" ships (labeled "B"), which occasionally fly across the screen. Its

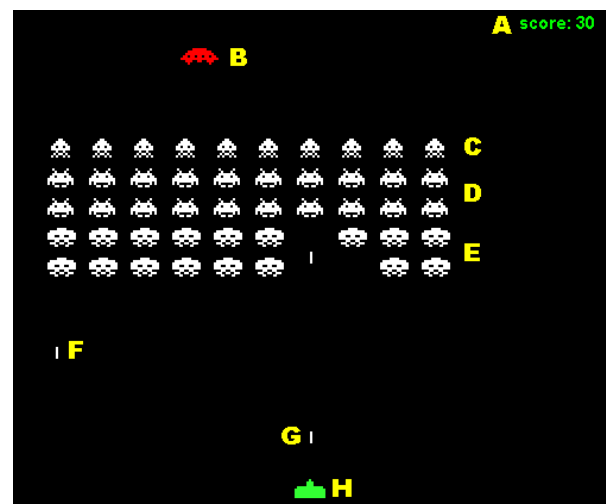


Figure 2: Elements in a Space Invaders game.

value is randomly chosen among 50, 100, 150 or 300 points.

Players have a laser base (labeled “H”) that moves sideways and shoots missiles (labeled “G”). Invaders can also fire missiles (such as the one labeled “F”). The game ends if the base is destroyed or if an invader reaches the level of the laser base.

Invaders fire missiles, although there can only be a maximum of 3 missiles on screen at any one time. Likewise, the base can have only 1 missile on the screen at any one time. If the player's missile does not hit any invaders, it reaches the top of the screen and disappears; at that point the base could fire another missile. Invader missiles that miss the laser base disappear at the bottom of the screen.



Figure 3: Initial state of the game.

## Program Description

The objective of this assignment is to develop a simplified Space Invaders program.

Figure 3 shows a snapshot of the program, which has a panel displaying the game and a menu bar. Figure 4 shows the “Game” menu options. Choosing “New Game” pauses the game (if one is been played) and displays a dialog requesting to start a new game (Figure 6). Choosing “Yes” ends the current game and starts a new one. “No” returns to the current game (if one is been played).

“Pause” pauses the game and “Resume” continues a paused game. “Pause” is only enabled if a game is been played, and that “Resume” is enabled only if the game is paused. Both options are disabled if no game is been played.

“Exit” displays a dialog (Figure 7) asking to confirm quitting the game. Choosing “Yes” ends the program. “No” keeps the program open. This dialog is also shown when using the red close window icon on the title bar.

The “Game” menu defines accelerator keys to access its menu options using the keyboard.

The “Help” menu has a single “About” option that (as shown in Figure 8) displays a dialog with the

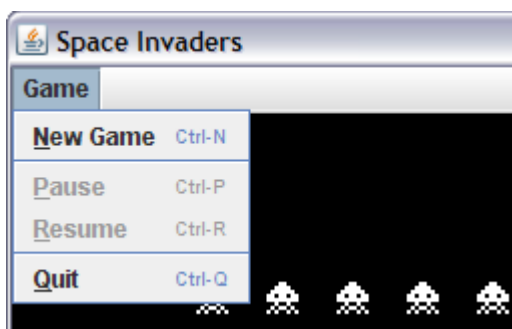


Figure 4: Game menu options.

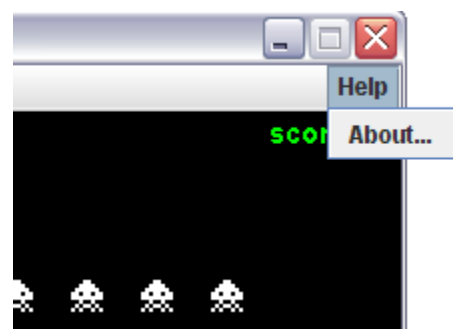


Figure 5: About menu option.

name of the programmers—that is, with your name not mine!

## Program Implementation

The program is made out of several classes (see Figure 9). These classes are described below.

### The SI class

This class (which stands for Space Invaders) is a subclass of *JFrame* that has a “main” method where execution of the program begins. It implements the menus used by the player to interact with the game.

The frame is titled “Space Invaders”, is not resizable and has a dimension of 500 by 450 pixels (width and height). The frame is displayed centered on the screen. The frame invokes methods in the *SIPanel* class as a result of menu selections.

Choosing to start a new game, to exit the program or to display the *About* dialog pauses the game (if not already paused) and resumes after the dialog is closed (but only if the game was not paused before; if it was, it remains paused).

### The SIPanel class

This class is a subclass of *JPanel* and is central to the player’s interaction with the game. This class is the most complex of all the classes in this assignment. It single-handedly keeps track of invaders, missiles, the base, mystery ship, score, and (if it wasn't enough) it receives events from the keyboard to move the base and to fire missiles, and from a *Timer* to advance the game.

### Timer and Action Events

*Sipanel* creates a *Timer* object that generates *ActionEvent* events every 10 milliseconds. Each time one is generated the state of the game will be updated; that is, ships and missiles move and get destroyed (more on how this happens is covered later in the section *Dynamics of the Game*).

The timer is created when the *Sipanel* is constructed. The advantage of using a timer to drive the state of the game is that stopping the timer pauses the game (that is, if there are no events then the state of the game is not updated) and restarting the timer continues executing the game.

### Keyboard Events

In addition to *ActionListener* (to handle *Timer* events), *Sipanel* implements *KeyListener* to receive keyboard events. This way the panel recognizes when the space bar, the left arrow or the right arrow keys are pressed to shoot a missile, and move the base to the left or to the right, respectively. Be aware that *Sipanel* must be able to “get the focus” before it can receive any keyboard events. To make it “focusable”, invoke the method *setFocusable(true)* in its constructor.

More about this class is mentioned later in the section on game dynamics.

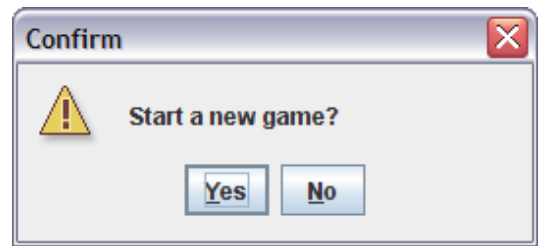


Figure 6: New game dialog.

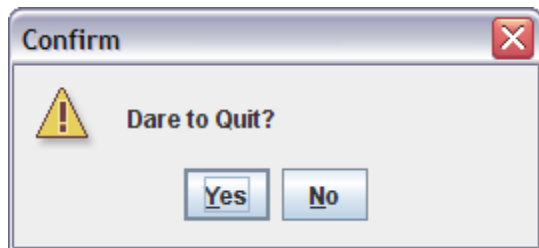


Figure 7: Exit dialog.

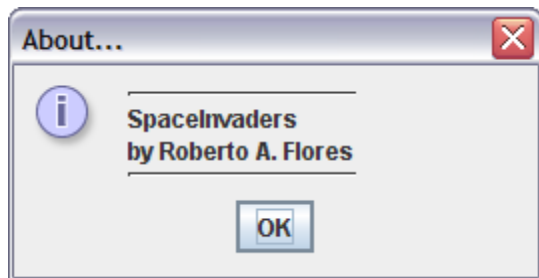
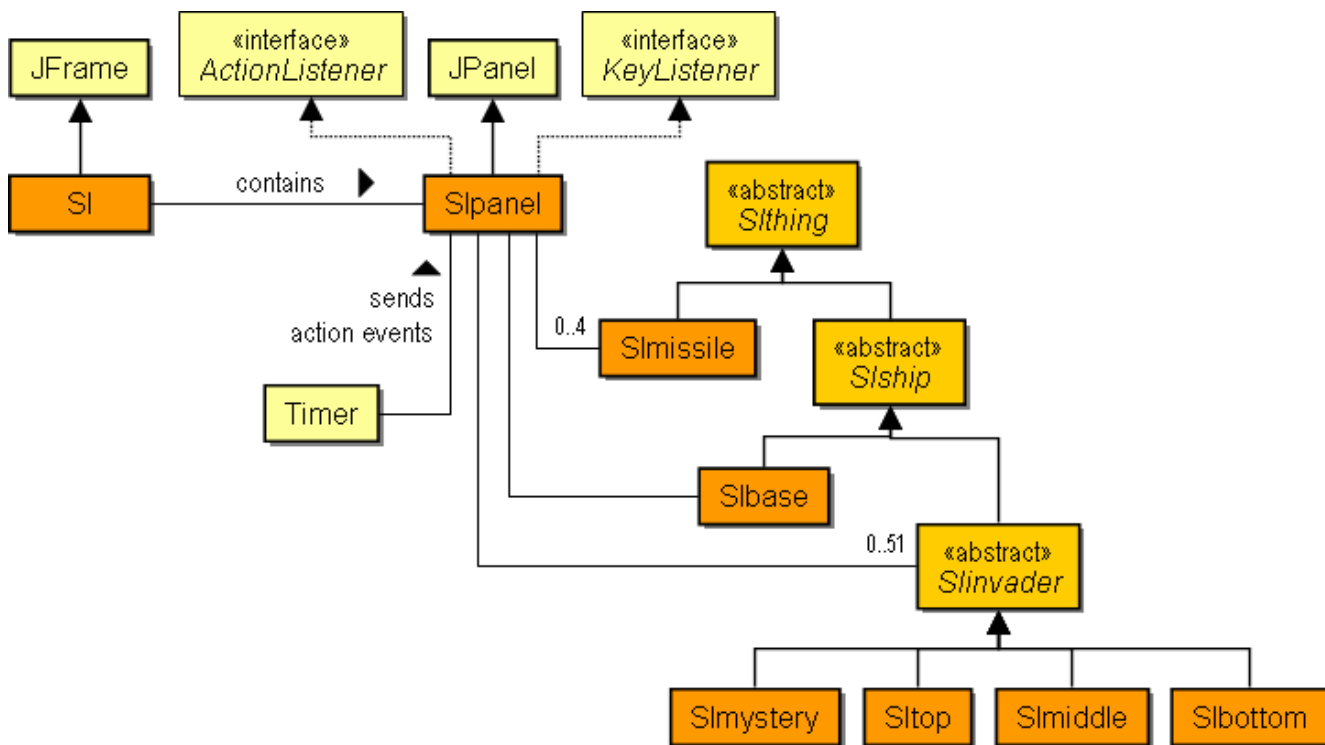


Figure 8: About dialog.



## Invaders, Missiles and Laser Base classes

The objective of the game is to destroy invaders with missiles shot from a base before invaders can land, while avoiding the missiles they fire back. These objects, whether invaders, missiles or the base, must be positioned, displayed and moved on the screen as the game advances.

The class *Sithing* is the class that encompasses the common characteristics of these objects. As shown in Figure 9, space invader “things” can be either missiles or ships (represented by the classes *Slmissile* and *Slship*), where ships are subdivided into bases or invaders (*Slbase* and *Slinvader*), and where invaders are further subdivided into mystery ships (*Slmystery*), and top, middle and bottom invaders (*Sltop*, *Slmiddle* and *Slbottom*). These classes are detailed below.

## The Slthing class

In the absence of a better name, anything that gets drawn on the panel (except the score) is a “thing”, or more precisely, a subclass of *Sthing*. This class defines fields keeping the x and y position of the “thing”, as well as its width and height.

This class also defines getter and setter methods, an abstract method for getting painted given a *Graphics* context, and concrete methods for moving this “thing” a number of pixels in a certain direction (either up, down, left or right), and saying whether this move does not reach a certain limit (e.g., any of the boundaries of the screen).

## The SImissile class

Missiles are the simplest kind of displayable objects in the game. They are implemented in the *Smissile* class, and drawn as white-filled rectangles of dimensions 2-by-10 pixels wide and high.

## The Sship class

There are two types of ships in the game: invaders and the base. Their common functionality is implemented in the *Sship* abstract class.

This class defines several fields: one indicating whether the ship has been hit by a missile, one keeping track of an audio clip with the sound that the ship makes when hit, and another one keeping an image showing what the ships looks like when it is hit.

This class includes a method to find out whether a given missile has hit the ship, and another method changing the state of the ship to “hit” (at which point it plays the appropriate sound).

All ships (whether invaders or the base) can use the same booming sound when hit. This sound is found as an audio clip named “SshipHit.wav” provided with this assignment (see the Resources section at the end of this assignment description).

The image that is displayed when the ship is hit is initialized independently by subclasses.

## The Sibase class

This simple class defines fields holding the image of the base when it is “alive” (“Sibase.gif”) and when it has been hit (“SibaseBlast.gif”). It defines a field to hold the audio clip “SibaseShoot.wav”, which is used when the laser base shoots a missile, and a method for playing this clip.

## The Slinvader class

The abstract class *Slinvader* is the super-class of all invader ships, either mystery ships or any of the three different types of invaders in the wave. Invaders share several characteristics, among which are that they are worth points, and that they have the same image when hit.

Figure 10 shows an example of an invader (fourth column, fourth row from left to right, top to bottom) exploding after being hit by a missile from the base. This image is found in the file “SlinvaderBlast.gif”, loaded in *Slinvader*'s constructor, stored in the image field defined in *Sship*, and drawn when the invader is shot.

Also, *Slinvader* defines a field indicating how many points an invader is worth. Note though that this field is initialized by individual subclasses, as each invader is worth different number of points. In the case of the mystery ship, its value is randomly selected among four equally-distributed values. Figure 11 shows the points each invader is worth.



Figure 10: Space Invader explosion.

## The Sltop, Slmiddle & Sibottom classes

The classes *Sltop*, *Slmiddle* and *Sibottom* define the invaders in the wave. As you can imagine, invaders at the top row are instances of *Sltop*, invaders on the second and third row from the top are instances of *Slmiddle*, and invaders on the bottom two rows are instances of *Sibottom*.

When playing the game, invaders alternate displaying an image each time they move.





	10 points
	20 points
	30 points
	One of 50, 100, 150 or 300 points

Figure 11: Points each Space Invader is worth.

These images are “Sltop0.gif”, “Sltop1.gif”, “Slmiddle0.gif”, “Slmiddle1.gif”, “Slbottom0.gif” and “Slbottom1.gif”; where “Sl<name>0.gif” and “Sl<name>1.gif” are the alternating images for invaders of type Sl<name>.

These images are shown in Figure 12.

## The SImystery class

The mystery ship is an invader that crosses the top of the screen and never descends. This ship could travel either from left to right or from right to left, and could be worth 50, 100, 150 or 300 points. The direction and points it is worth are randomly selected when the ship is created. Differently from other invaders, this ship does not have an alternating image, and continuously plays a sound when traveling across the screen. The image and audio clip for the mystery ship are “SImystery.gif” and “SImystery.wav”, respectively. If there were a mystery ship on the screen when the “New Game” button is pressed its sound will stop, and will resume when (and if) the game resumes. No more than one mystery ship can exist at the same time.

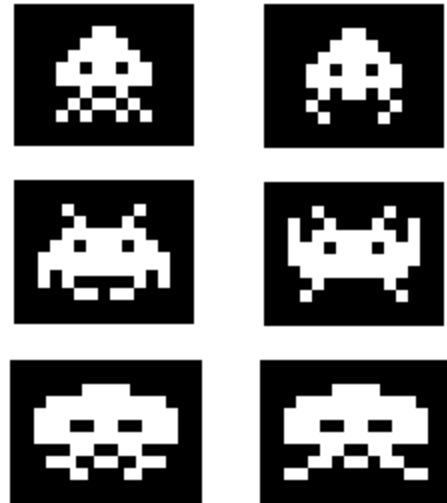


Figure 12: Alternating Space Invader images.

## Dynamics of the Game

This section describes what happens when the game is played. For the most part, it focuses on the *Spanel* class. This class keeps track of the state of the game, which is made out of a set of invaders, missiles, the base and the score.

### Pulses and Pace

When the *Spanel* is constructed, it sets the score to 0, initializes the base and the wave of invaders (in the positions shown in Figure 1), and creates a *Timer* generating *ActionEvent* events every 10 milliseconds. These events, which can be seen as the “**pulse**” of the game, invoke the *actionPerformed* method in *Spanel*. On each call, this method checks whether any ships have moved or have been hit, and whether missiles have been fired or have gone outside the screen.

Even though all events run at the same “pulse” frequency, different components of the game react at different “**paces**”.

We can conceptualize “pace” as the number of “pulses” that a component will let pass until it reacts. For example, the base moves faster than invaders because invaders have a lower pace than the base: invaders have an initial rate of 1 pace per 40 pulses, whereas the base reacts to every pulse, i.e., its rate is 1 pace per pulse. Invader missiles and mystery ships react at every 2 pulses. Base missiles react with every pulse.

To make the wave of invaders descend at a higher rate as the game advances, the program reduces the number of pulses they react to by 20 percent each time that an invader reaches the edge of the screen. For example, when the game starts, invaders react once at every 40 pulses; but when they reach the edge of the screen their reaction rate is reduced to one pace every 32 pulses. (Hint: To keep track of these different reaction pulses your program could keep two variables per component, one (counter) counting the number of events elapsed since last reaction, and another

one (limit) with the number of events that should pass before reacting. Once that counter becomes higher than limit then a reaction occurs and the counter gets reset).

## **The Wave of Invaders**

At the beginning of the game, 50 invaders are drawn in 5 rows of 10 invaders each. The wave is positioned at the center of the screen, 80 pixels from the top. Invaders are horizontally separated by 35 pixels (which is not the gap between them, but the difference between the x position of one invader and the next). Likewise, the difference between their y positions from one row of invaders to the next is 25 pixels.

Initially, invaders move toward the right edge of the screen at a rate of 40 pulses per pace. Each time they move, they shift 5 pixels from their current position. Once they reach the edge of the screen, they move down 12 pixels and change direction toward the other side of the screen at a rate of 32 pulses per move (i.e., the previous number of pulses per move minus 20 percent). When they reach the other edge of the screen they move down 12 pixels and reduce 20 percent of the pulse threshold at that time.

In addition to moving, invaders can fire missiles, which can be done 80 percent of the time when there are less than 3 missiles on the screen. This means that each time that the invaders react to the timer, there is an 80 percent chance that one invader will shoot a missile. However, only the invader that is lower in a column of invaders can shoot. For example, of the four invaders on the leftmost column in Figure 10, only the fourth from top to bottom can shoot. The second column has only one invader, located at the top row, so this invader could also be selected to fire a missile (which it does in the figure); and so on. In practice, once a shot is to happen then all lower level invaders have an equal probability of being randomly chosen as the one to shoot.

All missiles (including base missiles) move 5 pixels per pace. Even though invader and base missiles move the same amount of pixels, they travel at different speeds. The difference happens because invader missiles move at a rate of 1 pace every 2 pulses, and laser base missiles move with every pulse, thus making base missiles twice as fast as invader missiles.

If all invaders are destroyed before landing, a new wave is created at the same place and with the same paces they had at the beginning of the game. Note that the display of a new wave is postponed if there are either invader missiles or a mystery ship still on the screen. This means that the new wave is deployed once both the mystery ship (if there was one) has gone out of the screen (or you have shot it) and all missiles have passed the base location (beware: your base could be destroyed by any of these missiles, even though the invaders that released them have been destroyed!). More on the mystery ship below.

## **The Mystery Ship**

Once every so often a red mystery ship will cross horizontally the top of the screen. Mystery ships are created at a random rate of 0.003, and only if there is no mystery ship already on the screen. A new mystery ship has a 50 percent chance of moving either left-to-right or right-to-left. Mystery ships do not fire missiles, and are content to continuously play its sound while swiftly traveling from one side of the screen to the other. The reaction rate of mystery ships is one pace every 2 pulses, and moves 5 pixels per pace. Its vertical position is 50 pixels from the top of the screen.

## The Base

Since the base is the ship controlled by the player, it responds to keyboard events. If either the left or right arrow key is pressed, the base moves 5 pixels in that direction. If the space bar is pressed, then a new missile is fired, but only if there is no other base missile already on the screen. This missile moves at a rate of 1 pace per pulse (it travels twice as fast as invader missiles) and of course Missiles are able to hit any type of invader (whether a mystery ship or an invader on the wave). If they do not hit an invader, they disappear at the top of the screen, and the base is able to fire another missile. (Hint: There is no need to repaint the game each time that a key is pressed. Rather, you could modify the location of the base and expect that, when a new

*Timer* event triggers the *actionPerformed* method, the state of the game advances and the screen gets repainted with the new position of the base).

## Keeping the Score

Each time that an invader is destroyed, the score on the screen increases by the number of points that that invader is worth. The score is located at the top of the screen, and there is a gap of 10 pixels between the number of points and the right edge of the screen. As the number of points gets higher, their value gets updated. This means that the horizontal position of the score will shift to the left to accommodate the score and keep its distance from the right side of the screen to a constant 10 pixels (Hint: To find the horizontal position where the score is drawn each time that it needs to be drawn, you will need to calculate the length of the string “score: *number*”, add 10 (pixels) to this length, and subtract this result from the width of the screen). The score is drawn using a 15-point, green, bold, “SansSerif” font.

Scores are kept between waves of invaders; that is, the score retains earned points from one wave to the next, for as long as the game is not over. The score is reset if a new game begins.

## Ending the Game

The game ends if the invaders land or if the base is destroyed (sorry, this means that invaders never lose!) When the base is destroyed, the game stops and a “Game Over” message like the one shown in Figure 13 is displayed. This message is drawn centered on the screen, and is formatted using a 35-point, green, bold, “SansSerif” font.



Figure 13: Game over: Base destroyed.



## Resources

The initial resource files are in a zipped file available on Scholar. This file contains the audio clips and images shown in this assignment description.

## Groups

The assignment will be done individually or in groups of two students indicated by the instructor. Each person (or group) will submit one copy of the assignment with the name of the student(s) and their percentage effort in the assignment.

## Submission

Submit your `/*.java` files to Scholar.

At the top of your file, include the percentage effort made by partners (if you have one); for example,

- `// John Doe 50%, Jane Doe 50%` indicates that both partners made equal effort.
- `// Jack 100%, Jill 0%`, indicates that Jack did the assignment with no help from Jill.

## Grading

Your program will be tested according to the functionality specified in this assignment.

A perfect assignment would be one in which the player could play several rounds of the game without noticeable failures.

•|•