Project Video Game

Due: 3 Feb 2009

1 Overview

For this project, you’ll implement a video game of your choice—one of the three I’ve pre-approved and described below, or a different one that you can get me to approve (details in section 2.4). It’ll be a real system with a real interface; you’ll set up your Scheme file so that once it’s loaded into DrScheme, clicking “Run” will open the game window and start the game running and interacting with the keyboard.

Impress me!

2 The games

2.1 Space Invaders

The setup: the player is a defender at the bottom of the screen, and can move back and forth and shoot lasers upward. A phalanx of space invaders march mercilessly towards you, shooting at you, and you have to dodge their lasers (possibly hiding behind your shields) and shoot them before they land or kill you.

Details: The player is moved with the left and right arrow keys, and the spacebar fires a shot. Each laser shot ascends straight up the screen from wherever the defender was when it shot it, and the defender can’t fire again until the shot either leaves the screen or hits an alien. The aliens are arranged in a grid, initially 11 wide and 5 high; the whole phalanx of aliens moves as a unit, continuing to the left until the leftmost remaining alien is at the left edge of the screen, then descending by the height of one alien and moving en masse to the right. The aliens’ shots drop straight down, and there is no limit to the number of alien shots that can be on the screen. When a defender shot hits an alien, the alien and the shot both go away. When an alien shot hits the defender, the defender dies, and if that’s their last life, the game ends. If the aliens get to the bottom of the screen, the game ends even if the defender has lives left. If the defender clears the
board, they move on to the next level.

Extra: shields!

2.2 Snake

The setup: The voracious and venomous snake wanders around trying to feed itself, and as it feeds it grows. But it needs to be careful: the poisonous mushrooms sprouting up will kill it if it eats any, and if it runs into itself, its own venom will kill it too.

Details (version 1, “Snake”): The snake starts out very short, in the middle of the screen, heading in a random cardinal direction. It is controlled by the arrow keys, which start it off going up, down, left, or right; it continues in that direction until another key is pressed. Pressing a key opposite to its current direction has no effect. If a snake runs across a dead mouse, it eats it, and three things happen: another dead mouse shows up in a random, unoccupied location, a poisonous mushroom sprouts in a random, unoccupied location, and the snake grows (from the front). If a snake runs across a poisonous mushroom or any portion of its own body, it dies, and if that’s its last life, the game ends. After the snake has eaten twenty mice, the board clears and it moves on to the next level.

Details (version 2, “Voratio”): As with “Snake”, except that the snake starts out heading in any direction, and the left and right arrows make it turn a certain number of degrees to the left or right of its current heading. (The number of degrees should be empirically determined according to what makes for good gameplay, but it will be relatively small.)

Extra: live mice!

2.3 Pac-Man

The setup: A strange and oddly globular man runs around eating as many food pellets as he can find; when he eats one of the four “magic pills” he is filled with the power to eat the ghosts that relentlessly pursue him, haunting him in this waking nightmare—only to find that they will return to pursue him again.

Details: Pac-Man is controlled by the arrow keys, and continues moving in a given direction until countermanded by a new arrow or until he is stopped
by a wall. He cannot go through walls. When he passes over a pellet or a
magic pill, he eats it. Inky, Blinky, Pinky, and Clyde (the ghosts) wander
around the maze randomly (also restricted by the walls); under normal
circumstances, if they run into Pac-Man, he dies, and if that’s his last life,
the game ends. However, after he eats a magic pill, for a brief period the
ghosts turn blue, and if they run into Pac-Man then, he eats them and they
regenerate at their home in the middle of the screen. When all of the pellets
(and all four pills) have been cleared, the board resets and Pac-Man moves
on to the next level.

Extra: smart ghosts!

2.4 Or...

Propose another video game to implement! Your proposal should include
a “Details” section at the same level of specificity as the ones above. An
appropriate video game should have:

- At least three kinds of things
- Some kind of thing that responds to the keyboard\(^1\)
- Some kind of thing that responds to time passing
- Some kind of thing that has many instances on the board at any given
time
- Some kind of collision/intersection detection

Even with all that, I won’t guarantee approval (but if you’ve got a good idea,
try me). Past projects have included Asteroids, Frogger, Yar’s Revenge,
among others.

3 How to think about this

Above all, think functionally. Even if you’ve done something like this in a
procedural or OO language, you’ll be inclined to imagine yourself stepping
through a sequence of actions. Here, your world is a structure that represents

\(^1\) or mouse, as long as it’s moderately complicated
one moment of game state; to “update”, you write a function that can take one state and produce a new similar state based on what updated (i.e. time tick, which key was pressed, etc). To “draw”, you write a function that can take a state and produce a scene (which the library then takes care of rendering). And so on.

The drawing in particular is a bit different than you’re used to. Rather than the very procedural draw-on-a-canvas methods, you’ll take simple images and compose them into increasingly complex images and place them in a scene, and finally hand that scene off to someone else to actually render. That means, and this is important, the functions in image.ss are much more like constructors for some platonic Circle or Square class than like paint methods; they produce values, which in the end you’ll hand off to the DrScheme renderer to draw for you. If you call

\[
\text{(overlay (overlay (star 7 30 50 'solid 'red) } \\
\hspace{1cm} (circle 25 'solid 'yellow)) } \\
\hspace{1cm} \text{(text "boom" 10 'black))}
\]

nothing is actually drawn by the overlay functions themselves—but place this little explosion icon in a scene and return it when called by on-redraw, and *that* will cause it to be drawn on a canvas.

Part of thinking functionally is decomposing the problem into short input/output units. You will, or should, have a lot of relatively short functions when you get done.

THE POINT of this project is to A) get you thinking functionally about data interactions, and B) to get you more practice manipulating lists of stuff (laser shots, snake segments, etc) in non-contrived ways. You are sure to use \texttt{map} and \texttt{filter} on several occasions, possibly also \texttt{foldl} depending on the game. Have lots of fun with the graphics, but don’t let the graphics consume your time!

### 4 Assorted reminders

Don’t forget to load the \texttt{world.ss} teachpack, via “Add Teachpack...” in the Language menu.

Docs for the \texttt{world.ss} functions can be found in the Help Desk (Under “Languages” click “Teachpacks”) The functions for making Images are in
image.ss, and the functions for opening the canvas and registering callbacks are in world.ss. They’re also linked from the course webpage.

The World type referred to in the world.ss pages is whatever you define it to be; the library never actually interacts with it (it just hands it off to the various callbacks you’ve registered). Store as much or as little in it as is appropriate.

All the logic of your game will be testable without resorting to graphics, by writing test cases on examples of your internal World representation. You should probably take advantage of this.

Using overlay and overlay/xy, you can compose multiple Images into a single Image before ever placing them into a Scene. Make use of this to build your alien/pacman/mushroom/whatever image in advance, and store it—then you don’t have to programmatically redraw it every time it moves!

## 5 Timeline

**25 Jan** By Sunday at 7pm you need to have picked the game you will implement and emailed your choice to me. If you are proposing a different game, that needs to be done by tomorrow so that I have a chance to respond before the Sunday deadline.

**29 Jan** Demo day: next Thursday, part of the class period will be taken up with you showing off your system to the class. It does not need to be complete at this point, but you should at least have two or three things you can show to the class that you’ve done. This will be 5% of the grade, but you won’t be graded on how much of your project works—as long as you’re not just phoning it in, you’ll get the full 5% just for having something to show.

**3 Feb** The project is due a week from Tuesday, on the 3rd, at the start of class. Plan ahead.