Functional Programming Assignment — Fixing The World

Edwin Blake and Sunrise Wang

25th March, 2015

Introduction

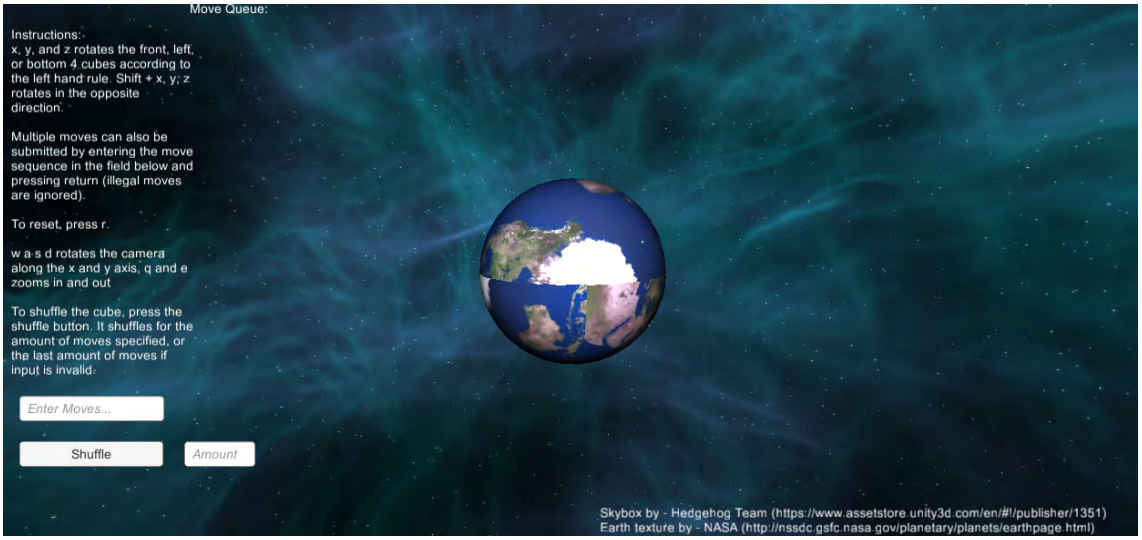

Figure 1: *Rubik's World simulator.*

A Rubik's cube is a 3D puzzle where the player is required to rotate cubes in a 3x3x3 grid so that each face of the grid has a single colour. In fact the only rotations that we shall be considering are through 90 degrees and we'll call those "twists".

In this assignment, we will be writing a solver for a simplified version of a Rubik's cube, Rubik's world. A solver in this context is a program that takes a shuffled Rubik's world and outputs the moves required in order to 'solve the world', i.e. rearrange the faces so that they're all in the correct orientation with respect to each other.

Rubik's World is a variation of the pocket cube, a 2x2x2 version of the original Rubik's Cube. In the Rubik's World, the earth is divided up into eight octants that allow for twists along the x, y, and z axis without changing the shape of the earth. Our aim is to write a simple brute-force solver for this, utilizing breadth-first search.

As it may be difficult to visualize and verify your outputs, a simulator is provided. Take some time to play around with the simulator in order to understand its functions and how it performs its twists, as your code will need to reflect these twists in order for you to correctly verify your outputs.

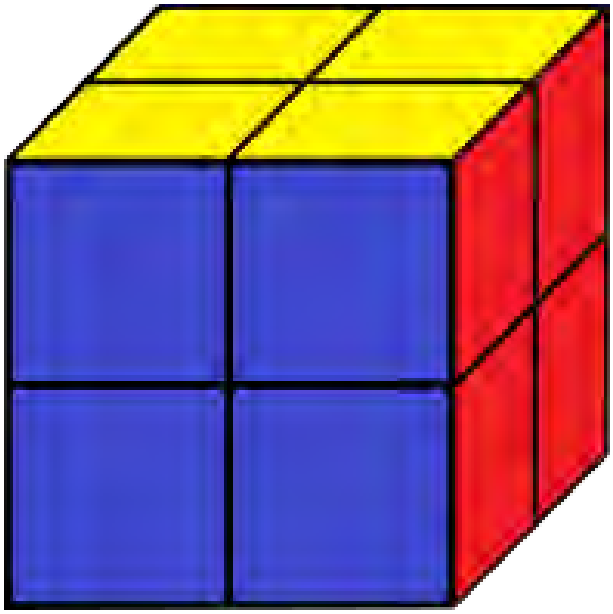

Figure 2: *A Pocket Cube.*

In order to run the simulator for Windows:
Browse to C:\rubiks_world and execute rubiks_world.exe
In order to run the simulator for Linux:
"Rubiks World" can be accessed via the Ubuntu Unity search dialogue, alternatively it can be run from the console with: /usr/local/bin/rubiks_world.86_64

The simulator is also on Vula for download.

Pocket Cube

As mentioned, the Pocket Cube is a 2x2x2 variation of the classic Rubik's cube puzzle (Figure 2). The puzzle has 8 sub-cubes, of which 4 can be twisted along the x, y, or z axis by 90 degrees at a time. The goal of the puzzle is to perform a series of twists on a scrambled cube state so that the overall cube has a uniform colour for each face.

We are going to aim to solve a variation of the Pocket Cube, namely *Rubik's World*. Rubik's world is almost identical to the Pocket Cube, with the only exception being that the sub-cubes now represent different pieces of the earth instead of octants. The goal is to thus move the pieces until the earth is in its correct position rather than to have each face be the same colour. If you are having trouble visualising Rubik's world as a Pocket Cube, consider the following: there are 8 Octants in Rubik's world as there are 8 cubes that make up a Pocket Cube. Each cube in the Pocket Cube corresponds to exactly one octant in Rubik's world. In Figure 3, we overlay 4 cubes of a pocket cube onto 4 quadrants of Rubik's world to indicate what we mean. Basically, Rubik's world is a Pocket Cube with the corners smoothed out.

Rotating the octants

The rotational rules for Rubik's World are the same as the Pocket Cube. This means that at any given time, you are able to twist 4 octants around the x, y, or z axis.

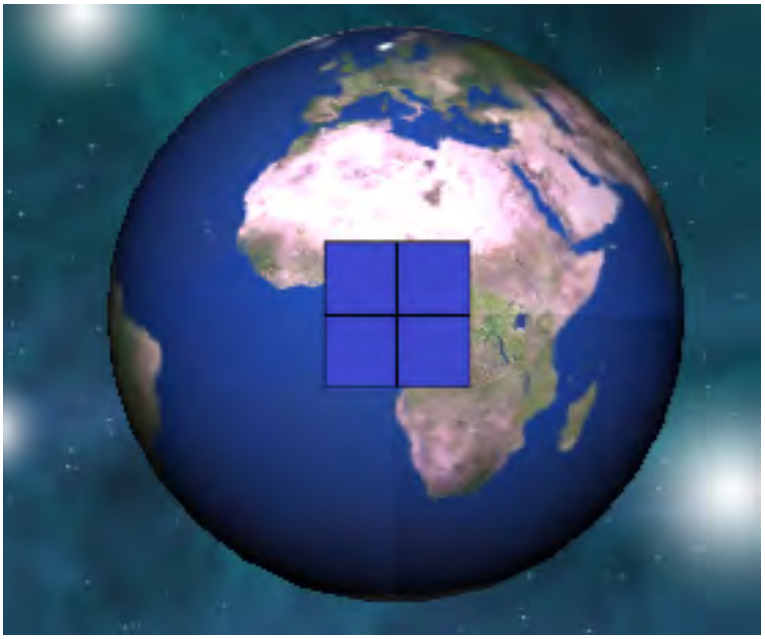

Figure 3: *Rubik's World can be converted to a Pocket cube by fitting a face at each corner of the octants.*

If you have taken time to play around with the simulator, you will have noticed that you can only twist the front, left, or bottom 4 octants along their respective axis. The reason for this is that a negative twist of 4 cubes is equivalent to a positive twist of the other 4 cubes and vice versa, as the **orientation of the puzzle is irrelevant**.

In the simulator, we treat a lower case axis name as the positive twist, and an upper case axis name as the negative twist. We determine the positive and negative directions of rotation by referring to the simulator's coordinate system, which is **left-handed** (unlike the normal mathematical coordinate systems you have used up to now — such a system is common in computer graphics). Figure 4 explains how this coordinate system works.

What to Hand In

Please hand in your solution as an archive named with your name and student number.

Your Code

Your code has to be able to run on one of the two interpreters: Gambit or Racket.

All your code must comply with acceptable Scheme style rules (if in doubt refer to your slides). Failure to choose appropriate names, create useful comments, etc. can lead to a loss of up to 10% per question. It is vital that you ensure that your code works. Code that does not work can get at most 40% and most likely will get much less.

As part of the brief, we have provided a template named 'assignment3.scm'. This contains skeleton methods that correspond to each question in the assignment. Please complete your assignment in this template, completing the code for each question in the methods provided.

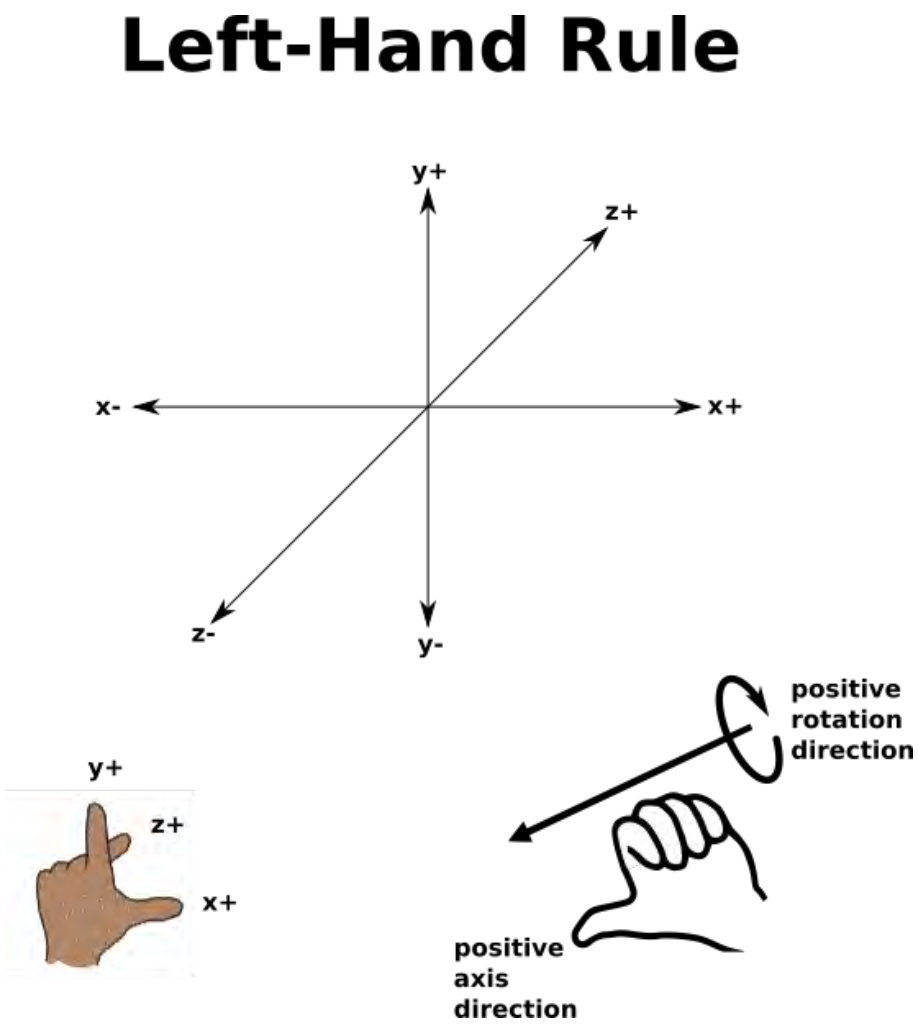

Figure 4: *The left handed coordinate system is shown, where the x axis is positive to the right, the y axis is positive in the up direction, and the z axis is positive in the forward direction. In order to figure out the positive rotation direction, you stick the thumb of your left hand in the positive direction of the axis. The way the rest of your fingers wind is the positive rotational direction*

We have done this to make it possible to automatically mark your code for correctness. Our automarker is as grumpy as it is strict — if you rename the file, or do not use the required methods it will simply mark you as wrong.

Evidence of Unit Testing

In addition to your code you need to give evidence of unit testing of the code. In other words for every function there must be test cases and resultant outputs. You will be writing some recursive functions and if possible you need to make them tail recursive. If you claim that a function is tail recursive then say so in the comment then supply a short trace output that shows this.

If you choose to have a separate file for test results, transcripts, and traces, the format should be either plain text, pdf, or word file (with headings for the various parts).

Theoretical Questions

There are a number of theoretical (i.e., not coding) questions in this assignment, questions 2.2, 3.2 and 4. Your answers should be in a pdf file, and submitted alongside your code and unit tests. Both your name and student number must also be in the pdf.

README

Please pay attention when writing your README. There are always a number of different ways of running and interpreting code. You cannot assume that the way you have chosen to complete this assignment is obvious to tutors. Please include all information that is relevant to the successful execution of your solution. This includes, but is not necessarily limited to

- operating system
- flavour of Scheme (Gambit or Racket)
- version of interpreter

You May Not Copy or Share Any Code

Please note that while you may discuss ideas with others, you may not copy or share any code.

Questions

The whole assignment will count out of 100 marks and that is the maximum you can get. The marks for the questions add up to 110 to make things slightly easier.

Tackling the problem

As the Rubik's World is a combinatorial problem, we should first understand how many possible states there are (assuming that only 90 degree twists are possible). There are

$$\frac{8! \times 3^7}{24} = 3\,674\,160$$

combinations. A good explanation can be found in http://en.wikipedia.org/wiki/Pocket_Cube. Briefly: Each of the octants can be in any position of the sphere, and has 3 different orientations

(however only 7 cubes can be twisted independently). As we are only taking into account 90 degree twists, there are 24 possible orientations in space for the solved Rubik's World.

We also need to agree on a way of encoding the problem. The state of the world can be encoded as
`((1 1) (2 1) (3 1) (4 1) (5 3) (6 3) (7 3) (8 3))` where we have 8 sublists that specify which octant is in which position and what each orientation is. So in the example in position zero we have octant 1 and its orientation is 1. If we now perform an 'x' twist then we get the following arrangement
`((5 4) (2 1) (1 2) (4 1) (7 4) (6 3) (3 2) (8 3))` where the first position (zero!) is now occupied by octant 5 in orientation 4 and the 1st octant has moved to the third position (2!). Please see the "Representation Guide" on Vula for more details.

Question 1 [30]

1.1 [15]

Write functions to twist your world state along the x, y, z, X, Y, and Z directions. In this case, the upper case letters denote a 90 degree twist in the negative direction, whereas the lower case letters denote a 90 degree twist in the positive direction.
hint: As we are only dealing with a 2x2x2 world, it is perfectly ok to hard-code this. However, should you wish to attempt to make this more general, have a look at 3D rotational matrices to get an idea of how to calculate the new positions.

1.2 [15]

Write a function that can generate all immediately succeeding states from a current given state. Additionally, this function should be able to take in a history of moves, and generate a list of moves with an extended history for each of the generated succeeding states. The moves are the ones that the simulator understands but in this case they are presented as a list (see skeleton code): `'(("x") ("X") ("y") ("Y") ("z") ("Z"))`.

Question 2 [30]

2.1 [25]

Write a function *genStates* that takes in a state and an integer n. This function should generate all the states reachable by the world in n twists.

2.2 [5]

Write a mathematical function that defines how many states (not necessarily distinct) can be generated in n moves.

Question 3 [40]

3.1 [30]

Use the *genStates* function that you wrote in Question 2 to implement a breadth-first search function, *solveCube* which searches for the moves required to move from a shuffled state to a solved state. This function should work for any shuffles of 6 moves or fewer. Use the simulator to verify your function. If the world is in a solved state, there will be white particles coming out of it.

3.2 **[10]**

Try your search function on shuffles of more than 6 moves. Keep an eye out on memory usage, as you may need to terminate the program before your system thrashes. What do you notice with regards to both memory usage and processing speeds? Why do you think this is?

Question 4 [10]

One way to alleviate some of the computational and memory burden is to reduce the amount of immediate successor states to 5, instead of the original 6. This can be done by checking the last move made, and then eliminating the twist that undoes that move (e.g if the last move is X, we do not generate the successor state that twists by x). How much will this reduce the computational cost by if we were solving worlds with shuffles of 10 moves (A rough estimate is sufficient)?

Note that although pruning away the untwist operation is a start, it is still not enough for worlds that require a large number of moves to solve. There other viable strategies which you will be looking at in the theory of algorithms course.

General Hints

Scheme functions that you may find helpful include

- list-ref
- map
- apply
- filter
- append
- list
- equal?
- null?

Acknowledgments

Many people contributed to setting this assignment. Firstly Peter Wentworth from Rhodes University for the idea. Then Sunrise Wang for turning the Scheme version into reality and writing the simulator. Also Steven Rybicki and Jared Norman for ideas and finalizing the current version.