Lawrence Sheriff School

31260

Rubik’s Cube Solver Documentation

Callum Newlands - 2077

Table of Contents

[**Analysis** 6](#_Toc511566732)

[Problem 6](#_Toc511566733)

[My Client 6](#_Toc511566734)

[Background 6](#_Toc511566735)

[Rubik’s Cube Solving Methods 6](#_Toc511566736)

[Beginner’s Method (Beginner Layer-by-Layer) [6] 6](#_Toc511566737)

[CFOP (Cross, First 2 Layers, Orientate Last Layer, Permute Last Layer) [7] 6](#_Toc511566738)

[Decision 7](#_Toc511566739)

[Stages of the Beginner’s Method [8] 7](#_Toc511566740)

[Client Interview Response 8](#_Toc511566741)

[Conclusion from questionnaire 9](#_Toc511566742)

[Objectives 9](#_Toc511566743)

[Modelling 11](#_Toc511566744)

[Data Required Before Each Stage 11](#_Toc511566745)

[System Diagram 11](#_Toc511566746)

[Current Solutions 12](#_Toc511566747)

[Microsoft Bing Rubik’s Cube Solver 12](#_Toc511566748)

[Ruwix (Rubik’s Cube Wiki) – Rubik’s Cube Solver 13](#_Toc511566749)

[Grubiks – Rubik’s Cube Solver 14](#_Toc511566750)

[Summary of existing applications 14](#_Toc511566751)

[OpenGL 15](#_Toc511566752)

[The 3D pipeline 15](#_Toc511566753)

[Shaders 16](#_Toc511566754)

[Matrices 16](#_Toc511566755)

[Prototyping 17](#_Toc511566756)

[**Design** 18](#_Toc511566757)

[Overall System Summary 18](#_Toc511566758)

[Decomposition of problem 19](#_Toc511566759)

[System Flowchart 20](#_Toc511566760)

[User Interface Design 21](#_Toc511566761)

[OpenGL Design 23](#_Toc511566762)

[Rendering the Initial Cube 23](#_Toc511566763)

[Shaders 23](#_Toc511566764)

[Showing the Instructions 24](#_Toc511566765)

[Data Structures/Classes/Enums 24](#_Toc511566766)

[Cube 25](#_Toc511566767)

[CubeOrientation 26](#_Toc511566768)

[Block 26](#_Toc511566769)

[Vector 27](#_Toc511566770)

[Matrix 28](#_Toc511566771)

[Instruction 28](#_Toc511566772)

[InstructionList 29](#_Toc511566773)

[Enumerated Types 29](#_Toc511566774)

[TopLayerSolver 30](#_Toc511566775)

[MiddleLayerSolver 30](#_Toc511566776)

[BottomLayerSolver 30](#_Toc511566777)

[3DOutput 31](#_Toc511566778)

[Vertex3D 33](#_Toc511566779)

[ShaderProgram 33](#_Toc511566780)

[Camera 33](#_Toc511566781)

[OutputBlock 34](#_Toc511566782)

[RotationStore 34](#_Toc511566783)

[OutputInstructionList 34](#_Toc511566784)

[Class Diagram 35](#_Toc511566785)

[Data Dictionary – Main Variables 36](#_Toc511566786)

[Files and Saving 38](#_Toc511566787)

[Algorithms 38](#_Toc511566788)

[Validating the Cube 38](#_Toc511566789)

[Converting StickerArray to Cube Object 40](#_Toc511566790)

[OrientateStickerArray 44](#_Toc511566791)

[Block.OnFace 47](#_Toc511566792)

[Block.CorrectRotation 47](#_Toc511566793)

[TopLayerSolver 48](#_Toc511566794)

[MiddleLayerSolver 51](#_Toc511566795)

[BottomLayerSolver 53](#_Toc511566796)

[Matrix Multiplication 60](#_Toc511566797)

[Rotating the Entire Cube 60](#_Toc511566798)

[Rotating a Specific Colour Face 61](#_Toc511566799)

[Optimising InstructionList 62](#_Toc511566800)

[GLSL version control 63](#_Toc511566801)

[Getting the Initial Positions for the Output 63](#_Toc511566802)

[Getting the Initial Rotations for the Output 64](#_Toc511566803)

[Rendering the Cube 67](#_Toc511566804)

[Structure diagrams 68](#_Toc511566805)

[Module Level Structure Diagram 68](#_Toc511566806)

[MainMenu 69](#_Toc511566807)

[Input 69](#_Toc511566808)

[Analyse 70](#_Toc511566809)

[Processing 70](#_Toc511566810)

[**Technical Solution** 71](#_Toc511566811)

[Programming Techniques Table 71](#_Toc511566812)

[Index of Code Files 76](#_Toc511566813)

[**Testing** 77](#_Toc511566814)

[Mid-Development Testing 77](#_Toc511566815)

[Final-System Testing 81](#_Toc511566816)

[**Evaluation** 90](#_Toc511566817)

[Overall Summary 90](#_Toc511566818)

[Evaluation of Objectives 90](#_Toc511566819)

[Client Feedback 94](#_Toc511566820)

[**1.** The Input 94](#_Toc511566821)

[**2.** The Output 94](#_Toc511566822)

[**3.** The Method/Instructions to Solve the Cube 94](#_Toc511566823)

[**4.** The Menu System/Program Design 94](#_Toc511566824)

[**5.** The File Saving/Loading System 94](#_Toc511566825)

[**6.** The Program as a Whole 95](#_Toc511566826)

[Analysis of Client Feedback 95](#_Toc511566827)

[Possible Improvements 95](#_Toc511566828)

[Appendix A: *Cube Terminology* 97](#_Toc511566829)

[Appendix B: *Rubik’s Notation* 97](#_Toc511566830)

[Appendix C: *Beginner’s Method Algorithms* 98](#_Toc511566831)

[Appendix D: *Program Code* 99](#_Toc511566832)

[MainMenu.vb 99](#_Toc511566833)

[Input.vb 101](#_Toc511566834)

[Analyse.vb 104](#_Toc511566835)

[CoordinatePairs.vb 110](#_Toc511566836)

[SetCubieProperties.vb 114](#_Toc511566837)

[Processing.vb 122](#_Toc511566838)

[Cube.vb 123](#_Toc511566839)

[Helpers.vb 138](#_Toc511566840)

[Block.vb 143](#_Toc511566841)

[Corner.vb 146](#_Toc511566842)

[Edge.vb 148](#_Toc511566843)

[Middle.vb 150](#_Toc511566844)

[Matrices.vb 151](#_Toc511566845)

[TopLayerSolver.vb 162](#_Toc511566846)

[MiddleLayerSolver.vb 172](#_Toc511566847)

[BottomLayerSolver.vb 175](#_Toc511566848)

[CubeOrientation.vb 185](#_Toc511566849)

[Algorithms.vb 186](#_Toc511566850)

[ArrayMethods.vb 189](#_Toc511566851)

[InstructionList.vb 194](#_Toc511566852)

[Instruction.vb 196](#_Toc511566853)

[FaceInstruction.vb 196](#_Toc511566854)

[CubeInstruction.vb 197](#_Toc511566855)

[3dOutput.vb 198](#_Toc511566856)

[Camera.vb 218](#_Toc511566857)

[Vertex3D.vb 220](#_Toc511566858)

[ShaderProgram.vb 221](#_Toc511566859)

[RotationStore.vb 223](#_Toc511566860)

[OutputBlock.vb 224](#_Toc511566861)

[OutputInstructionList.vb 224](#_Toc511566862)

[Mouse.vb 226](#_Toc511566863)

[Exceptions.vb 227](#_Toc511566864)

[VertexShader.txt 229](#_Toc511566865)

[FragmentShader.txt 230](#_Toc511566866)

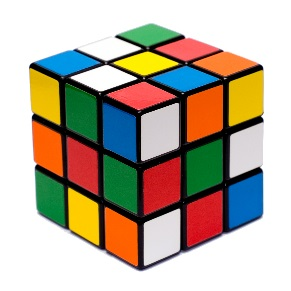
# **Analysis**

## Problem

My client has a Rubik’s Cube and would like an application that can show them the steps needed to solve it from any scrambled state, and help them to learn how to solve the puzzle.

## My Client

My client is a teenage girl who wants to learn how to solve the Rubik’s cube, and wants a program that can show her the steps needed to solve a scrambled cube.



**Figure** **1:** A Scrambled Cube

## Background

“In 1974, a young Professor of architecture in Budapest named Erno Rubik created an object that was not supposed to be possible - [The First Rubik’s Cube].” [1]

A Rubik’s Cube is 3D cube shaped puzzle made of 26 pieces called “cubies” or “cubelets” [2] (see **Appendix A** for Rubik’s cube terminology). The idea of the cube is that it is twisted into a scrambled state (e.g. **figure 1**), and then returned to its solved state (i.e. 6 faces containing each only 1 colour) by twisting its 6 faces (**figure 2**).



**Figure 2:** A Solved Cube

## Rubik’s Cube Solving Methods

The Rubik’s cube can be scrambled into 43,252,003,274,489,856,000[3] different combinations; as such, any method for solving scrambled cubes must be able to generalise steps and cannot be pre-coded for each possible situation. This makes it a challenging problem algorithmically, however there are many methods (at least 30) [5]that humans use to solve the cube. [4] 2 of the main methods used to solve the Rubik’s cube are the Beginner’ Method and CFOP:

### Beginner’s Method (Beginner Layer-by-Layer) [6]

* Number of algorithms: ≈5
* Average Moves per Solve: 80+

Layer-by-Layer (LBL) is a group of methods that solves the cube in layers. The Beginner’s Method is one of these methods which involves only a low number of algorithms and therefore is one of the easiest methods to learn.

### CFOP (Cross, First 2 Layers, Orientate Last Layer, Permute Last Layer) [7]

* Number of algorithms: 78
* Average Moves per Solve: 55

CFOP is an advanced LBL method which employs the use of selecting from many algorithms to solve each layer in fewer moves than the Beginner’s Method. “*CFOP has been the most dominant 3x3 speedcubing method since around 2000, with it and its variants used by the vast majority of the top speedcubers*”. *[7]*

[1] [1] <https://uk.rubiks.com/about/the-history-of-the-rubiks-cube/>

[2] [2] <https://en.wikipedia.org/wiki/Rubik's_Cube>

[3] [3] [http://www.dailymail.co.uk/sciencetech/article-1302414/Study-uncovers-possible-Rubiks-Cube-  
solution-Only-20-moves-needed.html](http://www.dailymail.co.uk/sciencetech/article-1302414/Study-uncovers-possible-Rubiks-Cube-solution-Only-20-moves-needed.html)

[4] <https://ruwix.com/the-rubiks-cube/different-rubiks-cube-solving-methods/>

[5] <https://www.speedsolving.com/wiki/index.php/List_of_methods>

[6] <https://www.speedsolving.com/wiki/index.php/LBL>

[7] <https://www.speedsolving.com/wiki/index.php/CFOP>

### Decision

**Figure 3**: Completed Cross



**Figure 4**: Completed 1st Layer Corners



**Figure 5**: Completed Middle Layer



**Figure 6**: Completed last-layer cross



**Figure 7:** Completed Cube

Because my client wants the solution to aid with learning how to solve the cube, I have decided to use the Beginner’s Method as a basis for my program because it has fewer algorithms to learn and is easier to understand. As a result, it will not necessarily give the most efficient solution, in term of the number of moves, but rather a solution that is easy to understand the stages of, and will therefore aid in the learning of how to solve the cube. Furthermore this will ensure that the program follows similar steps each time so that the client can better learn how the cube is being solved, as opposed to any of 78 algorithms.

### Stages of the Beginner’s Method [8]

1. **Cross** - Make a cross on one face by solving all edges of a given colour. Align the edges with the middle-layer centres, which remain fixed relative to each other (**figure 3**).
2. **1st Layer Corners** – Insert the 4 corners of the first face into the correct positions (**figure 4**).
3. **Middle Layer** – Position the middle row edge pieces in the correct places (**figure 5**).
4. **Last Layer Cross** – Make a cross on the last face (edge permutation), then align the edges with the middle-layer centres (edge orientation) (**figure 6**).

1. **Last Layer Corners** – Position the corners in the correct places (corner permutation), then rotate the corners so they are in the correct orientation (corner orientation) (**figure 7**).

N.B. Figures 3-7 are adapted from <https://www.youcandothecube.com/solve-it/3-x-3-solution>

In order to complete the 5 stages of the beginner’s method, 7 algorithms are needed:

1. Insert middle layer edge piece right *(step 3)*
2. Insert middle layer edge piece left *(step 3)*
3. Orientate bottom layer edge pieces *(step 4)*
4. Permute bottom layer edge pieces *(step 4)*
5. Permute bottom corner pieces right *(step 5)*
6. Permute bottom corner pieces left *(step 5)*

**IMPORTANT NOTE:**

Throughout my project I use the terms Block, Piece and Cubie interchangeably to refer to the 27 individual segments of a Rubik’s cube.

1. Orientate bottom corner pieces *(step 5)*

See **Appendix C** for the algorithms

[8] <https://www.youcandothecube.com/solve-it/3-x-3-solution>

## Client Interview Response

1. **How well do you know how to solve a Rubik’s Cube (1 = No knowledge -> 4 = full knowledge)**

|  |  |  |  |
| --- | --- | --- | --- |
| **1** | **2** | **3** | **4** |
| x |  |  |  |

1. **Which method would you prefer for outputting the instructions?**

|  |  |  |
| --- | --- | --- |
| **Rubik’s Notation** | **3D output** | **Both** |
|  |  | x |

1. **Would you prefer an application that teaches you how to solve the cube alongside showing the steps?**

|  |  |
| --- | --- |
| **Yes** | **No** |
| x |  |

1. **Which cube solving method would you prefer the application to use?**

|  |  |
| --- | --- |
| **Beginner Method** | **CFOP** |
| *A method which solves the cube in layers and involves only a low number of algorithms. It is therefore is one of the easiest methods to learn.* | *Cross, First 2 Layers, Orientate Last Layer, Permute Last Layer (CFOP) is an advanced layer-by-layer method which employs the use of selecting from many algorithms to solve each layer in fewer moves than the Beginner’s Method. It is an advanced method used for speedcubing.* |
| x |  |

1. **Are you likely to stop mid-way through solving, and want to save the current stage of the cube?**

|  |  |
| --- | --- |
| **Yes** | **No** |
| x |  |

1. **Are you likely to want to save/load multiple cubes?**

|  |  |
| --- | --- |
| **Yes** | **No** |
|  | x |

1. **Do you have any comments/ideas about how you would like the state of the cube to be input?**By clicking in the colours on screen – not having to type them. Ideally by a 3d diagram which you click on.
2. **Do you have any comments/ideas about how you would like the instructions to be output?**3D animation with step by step click through
3. **Do you have any comments/ideas about the saving/loading of cubes?**No
4. **Do you have any other comments/suggestions for the application?**No

## Conclusion from questionnaire

|  |  |
| --- | --- |
| Question No. | Conclusion |
| 1 | The client has no knowledge of solving a Rubik’s cube; the application should therefore go through the instructions 1 step at a time. It should also follow the beginner’s method, and should output the steps of the beginner’s method alongside the instructions to solve the cube. |
| 2 | The client wants a 3D animation showing how to solve the cube, alongside the Rubik’s notation for that step. |
| 3 | The client wants the application to teach her how to solve the cube, as well as showing how, so it should output the steps of the beginner’s method algorithm. |
| 4 | The client wants the application to follow the beginner’s method algorithm. |
| 5 | The client wants to be able to save the state of the cube mid-solving. |
| 6 | The client only needs to be able to load one cube at a time, so when saving cubes the program can overwrite other cube files. |
| 7 | The client wants to be able to input the state of their cube via clicking – the system should use a GUI. Ideally a 3d diagram, although this may have to be an optional, extension objective. |
| 8 | The client wants the instructions outputted via 3D animations, one step at a time. |

## Objectives

After a discussion with my client we have agreed upon the following set of objectives for my project. The system should:

1. **Allow the user to input the state of their cube**
   1. Use a GUI to enable easier input
   2. Allow choice from any of the 6 colours
   3. Allow each sticker to be edited after it has been entered
   4. Allow for input via a 3D GUI *(optional extension objective)*
2. **Have a menu system**
   1. Should be able to return to previous screens
      1. Data should be saved or a confirmation message should be displayed
   2. The menu system should be easy to navigate
      1. Clearly labelled buttons
      2. Similar layout and style throughout the solution
3. **Validate if the entered cube is a valid cube**
   1. Check there are 9 of each colour inputted
   2. Check that no 2 opposite colour stickers are on the same block
   3. Check that there are no of the same colour stickers on the same block
4. **Work out the instructions needed to solve the cube from its inputted state**
   1. The instructions should solve the cube fully
   2. The instructions should be fully simplified in terms of consecutive rotations of the same face
      1. 2 quarter turns in a row should be replaced with a half turn
      2. 2 quarter turns in opposite directions in a row should both be removed
      3. 2 half turns in in a row should both be removed
      4. A half turn and quarter turn in a row should be replaced with the opposite quarter turn (e.g. ¼ clockwise + ½ 🡪 ¼ anticlockwise)
   3. The instructions should follow the Beginner’s Method
   4. The instructions should be able to solve the cube from a half-solved state without restarting (e.g. if the green face is correctly solved it should carry on from that point)
5. **Output the instructions to solve the cube**
   1. Output each step one at a time
   2. Allow the user to replay steps
      1. Steps should be able to be undone
      2. Steps should be able to be repeated once undone
      3. Steps should be able to be undone and replayed any number of times
   3. Output the instructions in a 3D graphical format
      1. If 3D is not available on the user’s machine, output a suitable message and switch to text-based instructions
      2. Display the cube turning to clearly show the user the instruction
      3. Allow the user to control the speed of rotation
   4. Also offer the option of text-based instructions using Rubik’s Notation (see **Appendix B**)
6. **Allow the user to store the cube in the middle of being solved**
   1. Save the cube in a file
   2. Allow cubes to be retrieved from the files at the same instruction that the user saved it on
      1. Store the current instruction alongside the cube
      2. If the current instruction file is deleted then output a suitable message to the user

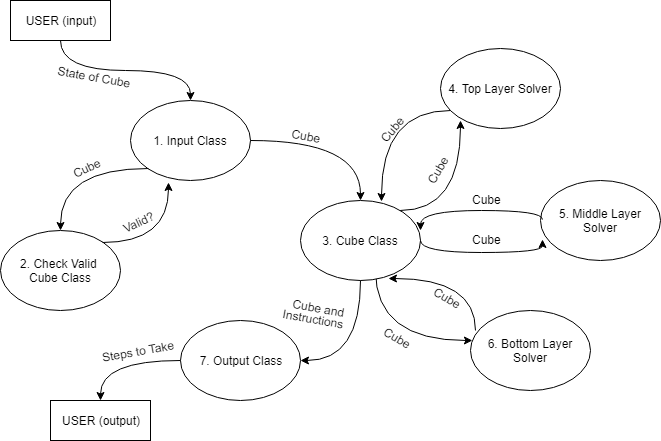
As this is a challenging and time-consuming problem I have discussed with my client the possibility that, as with any project, there may not be time to add all of the requested features. Through discussion, we have decided that objective 1.4 is non-critical and this will be added last, or not added at all, depending on the time left once all of the critical features have been added. To accommodate this and any problems that may arise during development, I will use an agile development method throughout the project.

## Modelling

### Data Required Before Each Stage

|  |  |  |
| --- | --- | --- |
| Input | Processing | Output |
| * + - * Current colours on each face of the cube | * Position of each cubie * Rotation of each cubie | * Steps needed to solve the cube |

### System Diagram



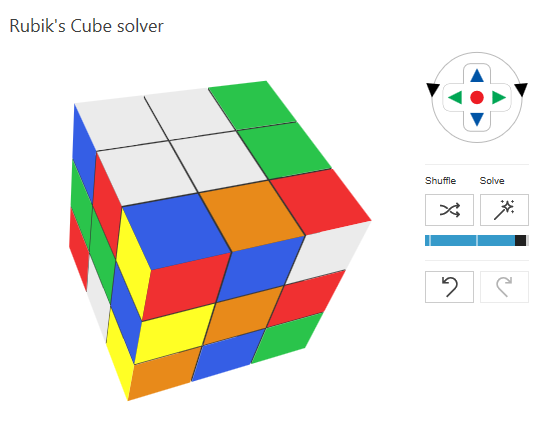
This diagram shows an abstraction of how my system will solve the problem:

It will first take the user’s input of the state of their cube (**1**) then it will check if the inputted cube is valid (**2**) it will then solve the cube and calculate the steps required (**3**). It will solve the top layer (**4**) followed by the middle layer (**5**) and then the bottom layer (**6**). This way it will follow the Beginner’s method to solve the cube. It will then output the stages required (**7**) to the user.

## Current Solutions

### Microsoft Bing Rubik’s Cube Solver

<https://www.bing.com/search?q=rubiks+cube>



|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Objective | Yes | No | Objective | Yes | No |
| 1.1 (Input via GUI) |  | **X** | **4.1** (Fully solved) | **X** |  |
| 1.2 (Input all colours) |  | **X** | **4.2** (Simplified Instructions) |  | **X** |
| 1.3 (Edit colours) |  | **X** | **4.3** (Beginner’s Method) | **X** |  |
| 1.4 (3D GUI input) |  | **X** | **4.4** (Continuing instructions) | **X** |  |
| 2.1 (Menu system) |  | **X** | **5.1** (One step at a time) |  | **X** |
| 2.2 (Easy to navigate) |  | **X** | **5.2** (Re-playable steps) | **X** |  |
| 3 (Validate cube) |  | **X** | **5.3** (3D output) | **X** |  |
|  |  |  | **5.4** (Rubik’s notation) |  | **X** |
|  |  |  | **6** (Saving cube) |  | **X** |

Objectives met: 5/16

#### Additional features

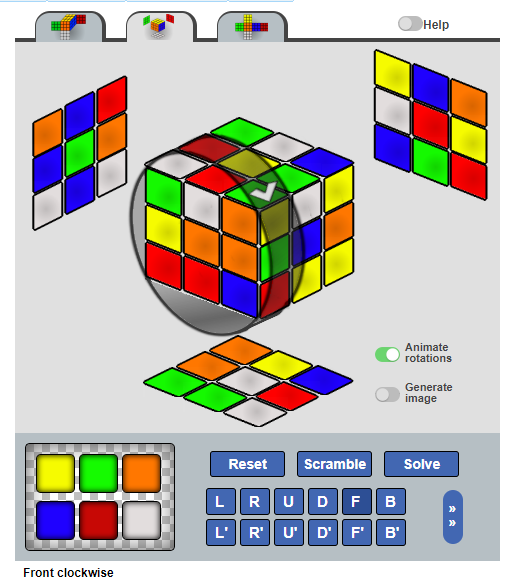
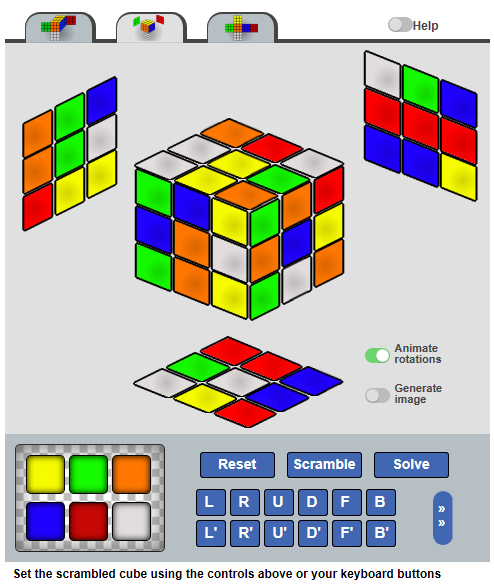
* Ability to manually rotate each face of the cube and then solve the cube from that new state

#### Missing features/criticisms

* Not able to input the state of your cube
* Not able to step through the steps one at a time
* Not able to save the cube
* No menu system

### Ruwix (Rubik’s Cube Wiki) – Rubik’s Cube Solver

<https://ruwix.com/online-rubiks-cube-solver-program/>



|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Objective | Yes | No | Objective | Yes | No |
| 1.1 (Input via GUI) | **X** |  | **4.1** (Fully solved) | **X** |  |
| 1.2 (Input all colours) | **X** |  | **4.2** (Simplified Instructions) | **X** |  |
| 1.3 (Edit colours) | **X** |  | **4.3** (Beginner’s Method) |  | **X** |
| 1.4 (3D GUI input) | **X** |  | **4.4** (Continuing instructions) | **X** |  |
| 2.1 (Menu system) |  | **X** | **5.1** (One step at a time) | **X** |  |
| 2.2 (Easy to navigate) | **X** |  | **5.2** (Re-playable steps) | **X** |  |
| 3 (Validate cube) | **X** |  | **5.3** (3D output) | **X** |  |
|  |  |  | **5.4** (Rubik’s notation) | **X** |  |
|  |  |  | **6** (Saving cube) |  | **X** |

Objectives met: 13/16

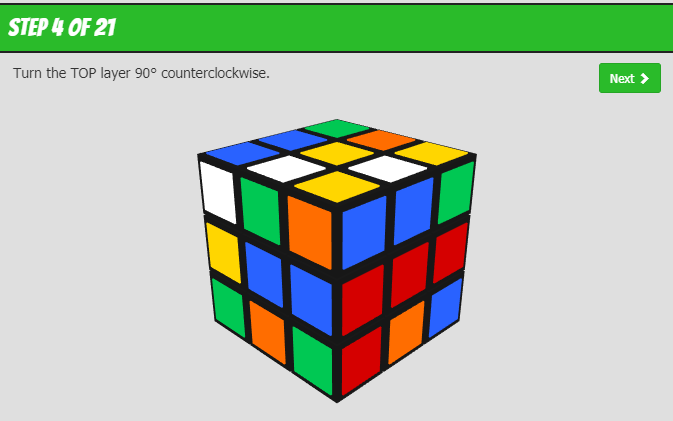
#### Additional features

* Outputs each stage of solving the cube as a 2D diagram as well as an animation.

#### Missing features/criticisms

* The animations don’t show the actual rotation
* Takes a long time to calculate the instructions required

### Grubiks – Rubik’s Cube Solver

<https://www.grubiks.com/rubiks-cube/solver/>

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Objective | Yes | No | Objective | Yes | No |
| 1.1 (Input via GUI) | **X** |  | **4.1** (Fully solved) | **X** |  |
| 1.2 (Input all colours) | **X** |  | **4.2** (Simplified Instructions) | **X** |  |
| 1.3 (Edit colours) | **X** |  | **4.3** (Beginner’s Method) |  | **X** |
| 1.4 (3D GUI input) | **X** |  | **4.4** (Continuing instructions) | **X** |  |
| 2.1 (Menu system) |  | **X** | **5.1** (One step at a time) | **X** |  |
| 2.2 (Easy to navigate) | **X** |  | **5.2** (Re-playable steps) |  | **X** |
| 3 (Validate cube) | **X** |  | **5.3** (3D output) | **X** |  |
|  |  |  | **5.4** (Rubik’s notation) |  | **X** |
|  |  |  | **6** (Saving cube) |  | **X** |

Objectives met: 11/16

#### Additional features

* Minimalistic design

#### Missing features/criticisms

* Doesn’t follow the Beginner’s method
* Steps cannot be replayed

### Summary of existing applications

The existing applications that I reviewed all solved the cube fully, and showed the steps in 3D, however none of them allowed for saving the cube mid-solve, and not all of them allowed for replaying of steps or inputting the state of the cube. Also, most of them didn’t follow the beginner’s method.

From the existing applications I have gained some ideas for how I am going to solve my client’s problem and the features that I am going to include in my design. Such as the input method used in the Ruwix solver of clicking to cycle through the colours, and the 3D style used in the Bing solver with the ability to rotate the cube with the mouse.

## OpenGL

As the system needs to output the instructions with a 3D animation, I have researched into OpenGL/OpenTK and have created some prototype projects to better understand how the libraries and 3D pipeline (**figure 8**) process work.

OpenGL (Open Graphics Library) is a cross-platform graphics API that allows programs to interface with the graphics card in order to render on the screen. OpenTK (Open Toolkit) is a .net wrapper for OpenGL which allows .net languages (such as VisualBasic.net) to use features from OpenGl.

### The 3D pipeline



**Figure 8:** The graphics pipeline – blue squares are areas you can create your own shaders for – from <https://learnopengl.com/Getting-started/Hello-Triangle>

1. The vertex data for the 3D models is passed into the vertex shader, which converts this data into vertices that can be rendered by the graphics card.
2. These vertices are then made into primitives (simple shapes - often triangles) dependent on the shape type specified in the code.
3. The primitives are then passed into the geometry shader, which can form more complex shapes from them.
4. The shapes are then rasterized (mapped onto pixels on the 2D screen).
5. These pixels are then passed into the fragment shader, which deals with colouring the pixels
6. The resulting pixels are then tested for depth (so that closer objects are rendered in front of further ones) and opacity
7. Finally the pixels are displayed on the screen.

### Shaders

Shaders are user-defined programs designed to run on a graphics processor. They are each run at a specific section of the graphics pipeline, and can only communicate via their inputs and outputs. In modern OpenGl, you have to write your own vertex and fragment shaders, but geometry shaders are not needed for all applications. Since I will only be rendering cubes I will not need a geometry shader. Shaders are written in GLSL (OpenGL Shader Language):

Example shaders:

**Vertex Shader**

#version 330 //GLSL version

layout (location = 0) in vec3 position; //vertex position vector from program

uniform mat4 model; //model matrix from program

uniform mat4 view; //view matrix from program

uniform mat4 projection; //projection matrix from program

void main()

{

gl\_Position = projection \* view \* model \* vec4(position.x, position.y, position.z, 1.0); //multiplies the position vector by the projection, view and model matrices then sends the result to the fragment shader

}

**Fragment Shader**

#version 330 //GLSL version

out vec4 FragColor; //used to output the pixels to the graphics card

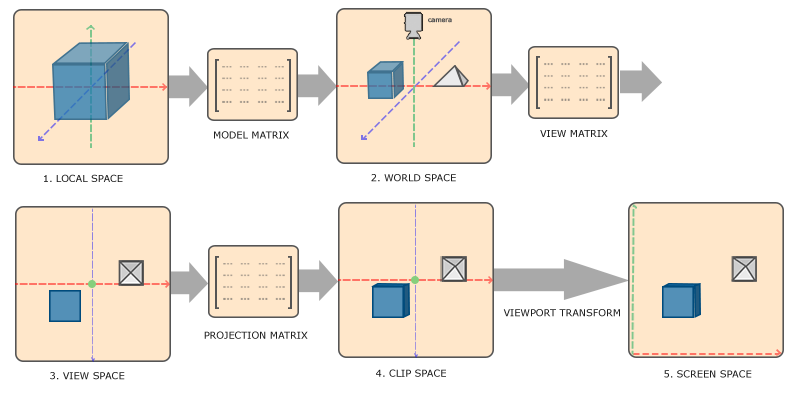
void main()

{

FragColor = vec4(1.0f, 0.5f, 0.2f, 1.0f); //Sets the pixel colour to #FF8033

}

### Matrices

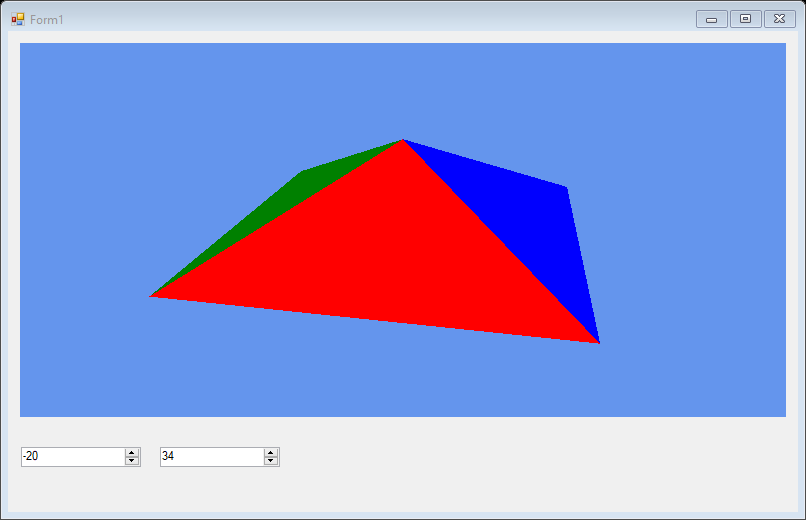
OpenGL uses vectors to represent vertices and matrices to transform these vertices (i.e. rotate, translate and scale them). It then uses various transformation matrices to map the coordinates from local space to screen space (**see figure 9**)

**Figure 9:** Transformation matrices – from <https://learnopengl.com/Getting-started/Coordinate-Systems>

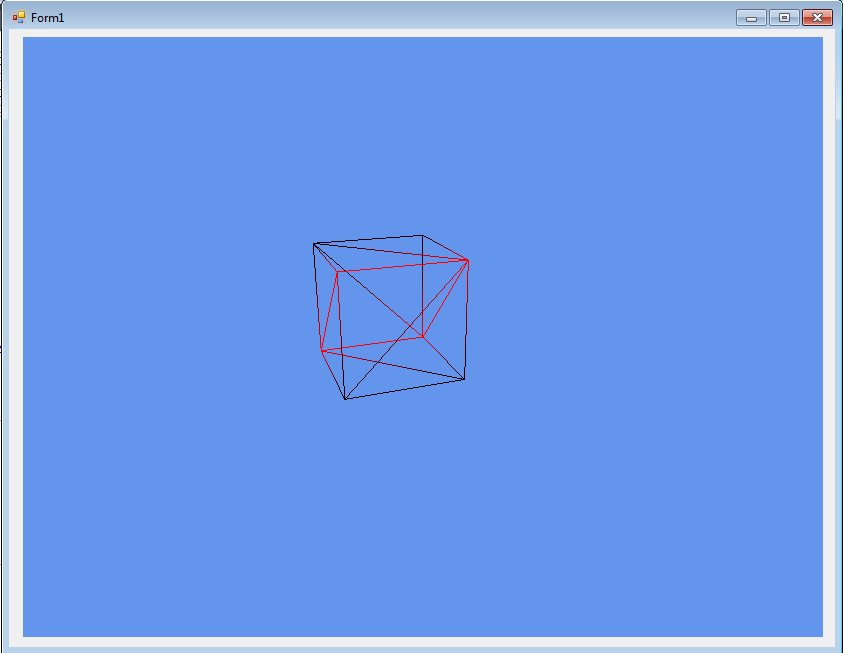
* A model matrix maps an object from its local coordinates (origin at the objects centre) to its correct position in the world (local space to world space). The model matrix consists of any rotation, translation and scaling of the model.
* A view matrix transforms the world coordinates to view-space coordinates so that each coordinate appears to be seen from a single point of view (e.g. a camera).
* A projection matrix then maps each view-space coordinate to a coordinate in clip-space. The projection matrix controls whether the scene is to be viewed with perspective or orthogonally.
* Finally the coordinates are mapped to screen-space pixel coordinates that correspond to the pixels in the window.

### Prototyping

In order to learn more about OpenGL and how the render pipeline works, I created several prototype projects including the following (**figures 10 and 11**)

****

**Figure** **10:** A pyramid to prototype coloring different faces of a shape and controlling the rotation of the whole model



**Figure 11**: A wireframe cube to prototype rotating cubies around a fixed origin and rendering a cube

Through prototyping I gained a better understanding of how OpenGL works, and how to use shaders, vectors and matrices in order to create 3D models on the screen. I have detailed this, and the results of further research in the design section.

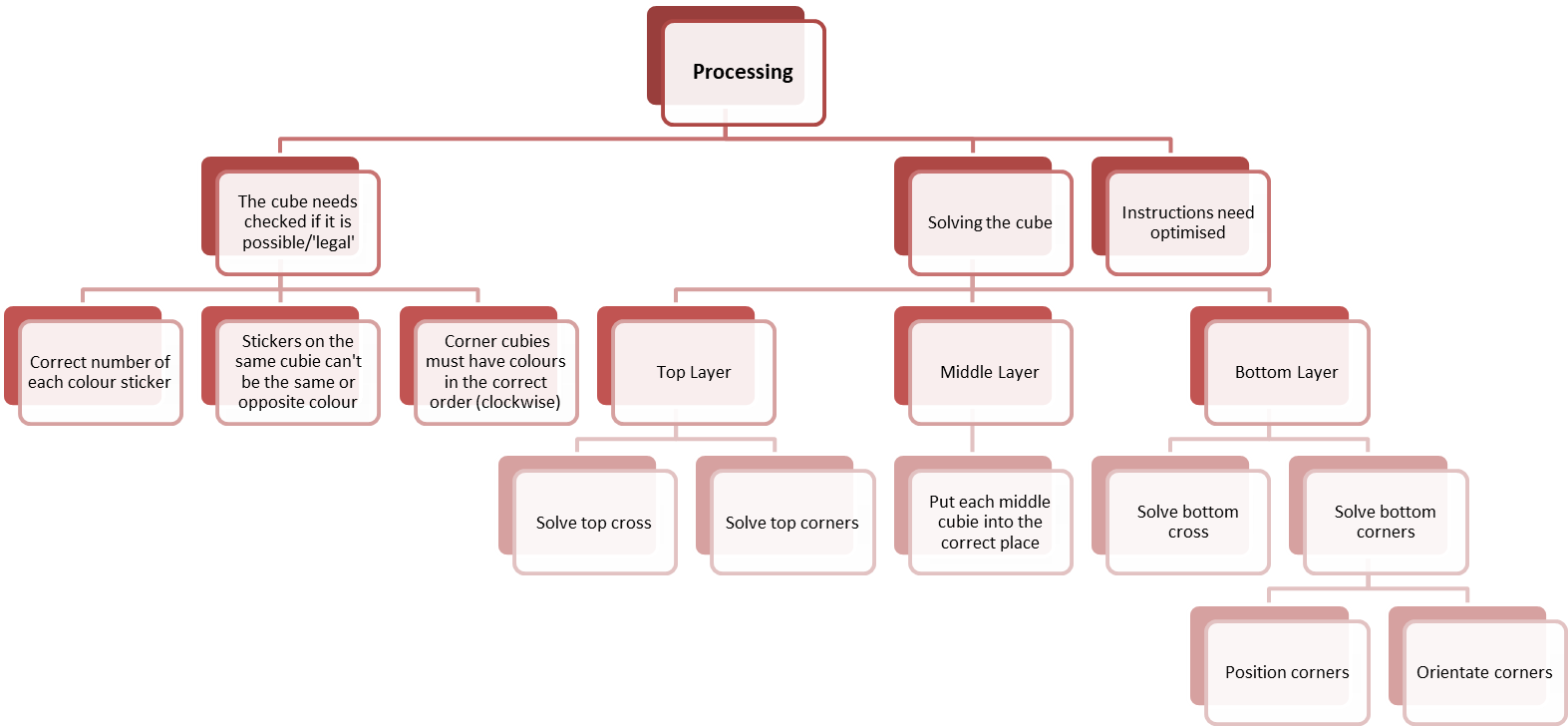
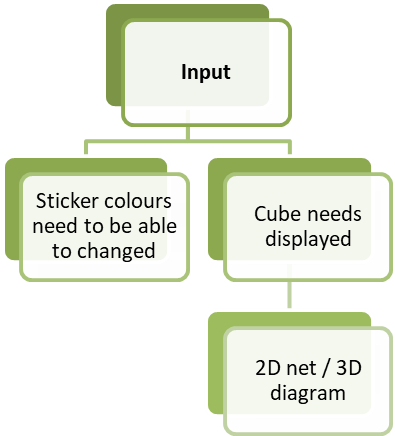
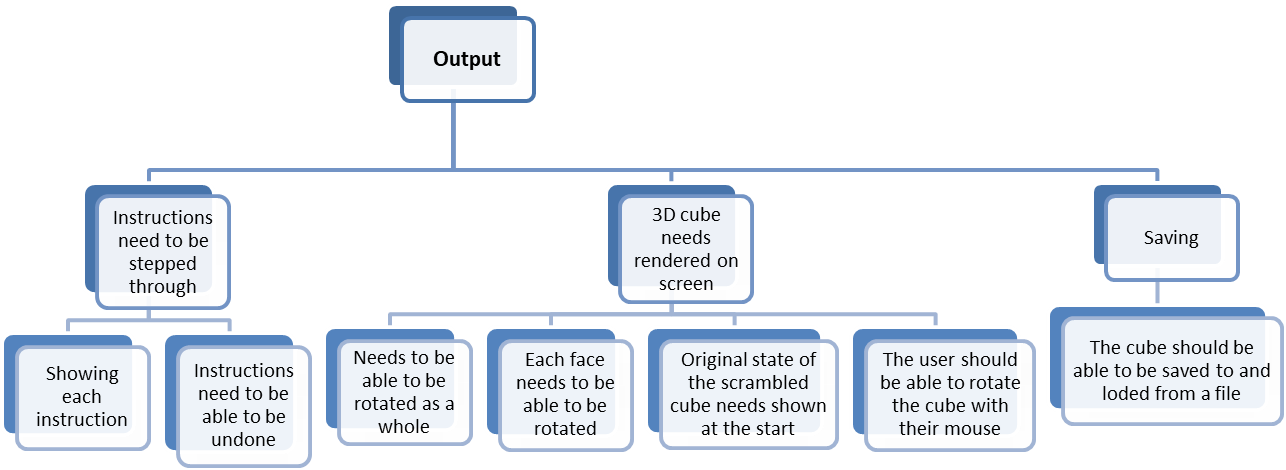
# **Design**

## Overall System Summary

My designed solution can be broken down into 4 main parts:

1. Input: I will use a GUI to allow the user to input the colours of the faces of their cube. This will be laid out in the shape of the net of a cube, and the user will click on each of the stickers to change its colour until it matches the one on their cube that they want to solve. The colours will then be stored in 2D array (1st dimension = the face, 2nd dimension = the position on that face) which will then be passed into the analyse section. (I have taken out the design for 3D input as this was not implemented, as it was an optional objective, howvever I will detail ideas about how it would have been implemented in the evaluation section)
2. Analyse: I will then check that the cube that the user has inputted is valid, this will involve checking that there are exactly 9 stickers for each colour, and that no cubie contains 2 stickers that are of the same colour, or of colours that are on opposite faces of the cube (e.g. white and yellow because the white face is opposite the yellow face). If the cube is ‘legal’ I will then send it to the solve section as an object instantiated from the cube class (see later in the section) which will contain properties and methods related to the cube.
3. Solving: I will solve the cube following the Beginner’s Method, and will use 3 separate classes for each layer. I will pass the cube into each class as an aggregate object so that the layer can be solved and the cube can be used outside of the class. Please see the algorithms section for more detail on how each layer will be solved. Once the cube is solved the steps (instructions) needed to solve it will be optimised so that there are no consecutive steps of the same move and then the steps will be output to the user.
4. Output: I will then output the instructions to the user using a 3D model of a cube which performs the required instruction on the screen. The cube will start of identical to their cube and will be transformed the same as the real-life one as the instructions are executed. I will also have the facility to save and load cubes mid-saving.

## Decomposition of problem



## System Flowchart

This is an overview of how the entire system will work and shows the main program flow.

See later in the design for more detailed breakdowns of each section.

The instructions to solve the cube will be output as the user clicks through them. The user will also be able to undo and replay instructions

The program will check if the inputted cube is valid (possible) and if not will ask the user to check the cube and correct any mistakes when inputting it

The list of instructions will be optimised so that there are no consecutive rotations of the same face

The cube will be passed to a separate module that will solve the bottom layer

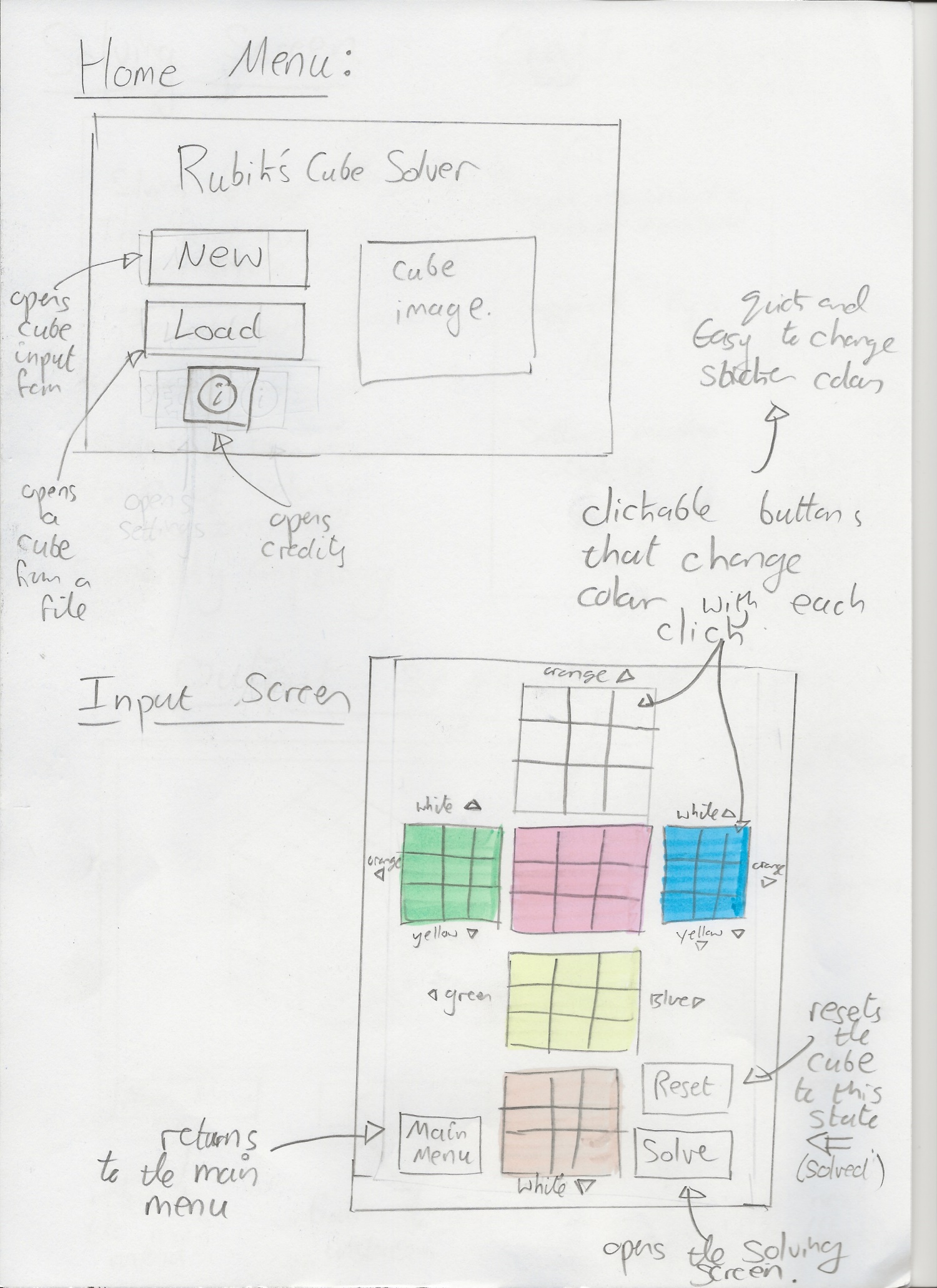
The user will input the state of their cube by clicking on buttons via the input form

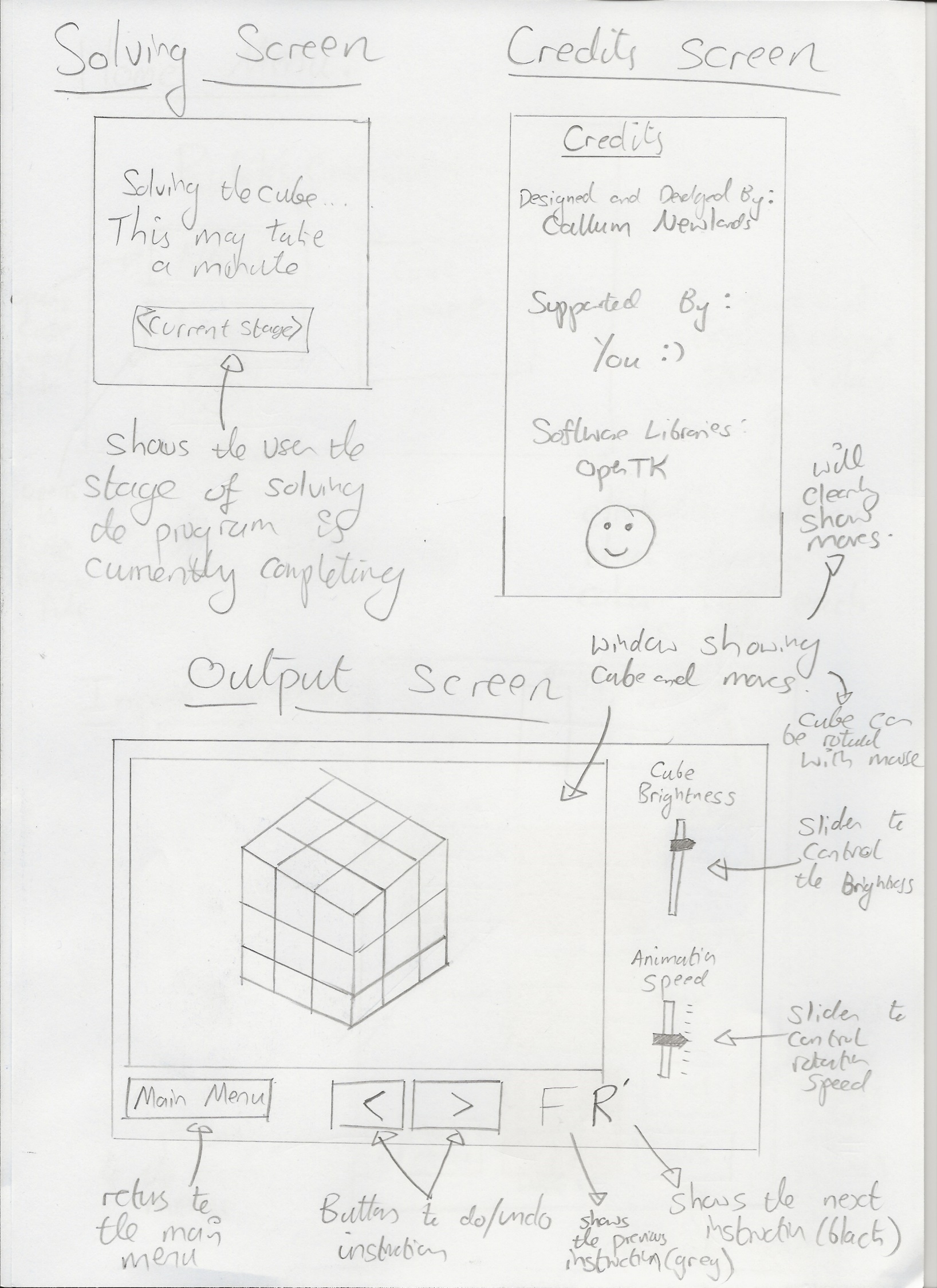
Via the main menu form, the user will choose to either create a new cube (and input the state of their cube), or to load a saved cube from memory.

The cube will be passed to a separate module that will solve the middle layer

The cube will be passed to a separate module that will solve the top layer

## User Interface Design

As I used an agile development process and iteratively designed and programmed the solution in sections I have only included the final designs for each of the sections. Alongside each sketch I have described the features of each window, and explained my choices where it is unclear. I am using buttons to navigate between the different windows as this is a style that is common between programs and this will allow me to create a consistent menu/GUI style throughout. Also, I will use the Microsoft Sans Serif font throughout the program for the same reason.



## OpenGL Design

### Rendering the Initial Cube

In order to render a Rubik’s cube to the screen, I will create each of the 27 cubies individually and then translate them to their correct positions. This will allow me to match up with my planned block classes in the rest of my program, and will also allow me to rotate each of the cubes faces by just rotating the individual cubies. I will centre the cube at the world origin, and therefore any rotations applied after the translations will be centred on the centre of the Rubik’s cube – the same as a real cube.

As each of the 27 cubies is the same cube model, I will only need 1 set of vertex data for a generic cubie (I will go into this data in the data structures section below). I will make each cubie with faces of each of the 6 colours, and I will then use the fragment shader to determine which of the cubie’s faces should be rendered with colour (from the vertices’ positions) i.e. only outward facing faces will be rendered with colour, the internal faces of the cubie will be black.

I will store each of the cubie vertices in an OpenGL Vertex Array Object (VAO), I will then copy them to an OpenGL Vertex Buffer Object (VBO) so that OpenGL can process and render them. In order to specify which vertices make up which faces (so that the colour match up) I will also use an OpenGL Element Buffer Object (EBO) this will allow me to specify which vertices are drawn together as polygons.

Once I have made each of the cubies I will need to rotate them to their correct starting orientation and translate them to their correct places so that the rendered cube matches the inputted cube. The starting positions I can get from the block classes passed from the previous stage in the program. I will detail the algorithm for getting the initial rotations in the algorithms section below.

### Shaders

I will write a vertex shader and a fragment shader for my program. In order to accommodate for the 2 main versions of GLSL (120 and 330) I will use pre-processor directives that will run one of 2 different functions in my shaders. This will allow my program to be used across many more platforms than if I just use a single version of GLSL. Below is the pseudocode for both shaders:

#### Vertex Shader

Because only the vertex shader can access the vertex attributes (position, colour, normal), I will need to pass these into the fragment shader to use them. To represent this in the pseudocode I will use the syntax: Pass(variable) to represent passing a variable to the fragment shader. To represent outputting the vertex’s final position to the graphics card I will use the syntax Output(vector).

main()

{

Output( projectionMatrix \* viewMatrix \* modelMatrix \* vertexPositionVector)

Pass(vertexColour)

Pass(vertexPositionVector)

}

#### Fragment Shader

To represent accessing variables passed in from the vertex shader I will simply use the same names as the variables passed in by the vertex shader. The initialModelMatrix is the matrix applied to the cubie to translate and rotate it to its initial (scrambled) position, and rotation, before any of the instructions are shown on the cube. To represent outputting the vertex’s final colour to the graphics card I will use the syntax Output(colour).

main()

{

worldPositionVector = initialModelMatrix \* vertexPositionVector

IF (worldPositionVector is outside the bounds of the cube) THEN

Output(lightStrength \* lightColour \* vertexColour)

ENDIF

}

### Showing the Instructions

When rendering instructions they will be applied as consecutive multiplications of rotation matrices. Rather than multiplying all the matrices every time for each frame, I will store the most recent rotation matrix (the product of all previous matrices) for each cubie. I will also store the current rotation for each cubie as an axis and max angle, and will apply this rotation with a percentage of the max angle, defined statically in my render loop. This will allow the current rotation to gradually increase by angle – and thus will result in the face appearing to rotate over time. See the data structures section below for the class I will use to store these rotations.

## Data Structures/Classes/Enums

For my solution I will create the following data structures/classes. Please see the algorithms section for explanations of how most of the methods detailed here work. Many of the classes in my project have ToString() functions, these are overrides of the built in VB.net ToString() function and have been excluded from my diagrams for clarity.

Please Note: any properties marked as public are not public variables, but rather are public properties with getter and setter methods. Any public properties marked in blue and marked {readOnly} are read only properties, and as such only have getter methods.

[Cube 25](#_Toc510694590)

[CubeOrientation 26](#_Toc510694591)

[Block 26](#_Toc510694592)

[Vector 27](#_Toc510694593)

[Matrix 28](#_Toc510694594)

[Instruction 28](#_Toc510694595)

[InstructionList 29](#_Toc510694596)

[Enumerated Types 29](#_Toc510694597)

[TopLayerSolver 30](#_Toc510694598)

[MiddleLayerSolver 30](#_Toc510694599)

[BottomLayerSolver 30](#_Toc510694600)

[3DOutput 31](#_Toc510694601)

[Vertex3D 32](#_Toc510694602)

[ShaderProgram 33](#_Toc510694603)

[Camera 33](#_Toc510694604)

[OutputBlock 34](#_Toc510694605)

[RotationStore 34](#_Toc510694606)

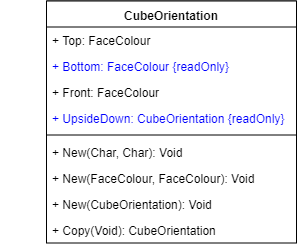
[OutputInstructionList 34](#_Toc510694607)

### Cube

This will be used to store and represent the Rubik’s cube as it is being solved. As such it will need to be able to store each of the cubies, as well as have methods to solve the 3 layers of the cube, and to rotate its faces and the cube as a whole. I will also store the instructions to solve the cube in the cube class as they will need to be accessed only by the cube, when it is being solved. It will be the largest class, and will do all of the solving of the cube.

|  |
| --- |
| ***CubeOrientation*** object to store the current orientation of the cube (determines position vectors and which faces are which (i.e. which colour is the front face), and properties to get/set the orientation. |
| Arrays of ***Block*** objects to store the cubies in the cube and their properties (colours, positions, rotations etc.). |
| ***InstructionList*** object to store the instructions that will be presented to the user once the cube is solved. |
| ReadOnly properties that return groups of ***Blocks*** as an array, for example the edge pieces on the top face going clockwise around the face. These groups are used when solving the cube, for example to check which of the top corner pieces are correctly positioned. |
| A Boolean read only property: returns true if the cube is fully solved. |
| Constructor methods and private methods used by the constructors. The New(Char(,)) method takes in a 2D net of sticker colours from the input form (see UI design) and converts it into ***Block*** objects to be manipulated more easily. |
| The methods which solve the cube in layers. Each of the 3 Solve{*layer*}Layer methods creates an object of the respective LayerSolver classes and passes the cube (itself) in for solving. E.g. the SolveTopLayer() method creates a ***TopLayerSolver*** Object with Me as the constructor argument, then calls TopLayerSolver.Solve() |
| Methods to rotate the entire cube and update its orientation. |
| Method to rotate a specified colour face, and update the position and rotation of the affected ***Blocks*** |
| Methods to serialise the cube and save it to a .cube file |

### CubeOrientation

This class will be used to store the orientation of the cube. This can be defined by just 2 faces (as the colours are in a fixed position relative to one another) so I will store the top and front faces, as these are the faces that are commonly referenced when telling someone what orientation to hold their cube in.

|  |
| --- |
| Property to store the colour of the top face of the cube |
| Returns the opposite colour to the top face |
| Property to store the colour of the front face of the cube |
| Returns a new ***CubeOrientation*** object which has the opposite top face |
| Constructors |
| Returns a new ***CubeOrientation*** object which is a copy of the current one |

### Block

In order to represent the cubies/blocks in the cube I will use a block class which will store the position, rotation and colours of each cubie. It will also have methods to check if the cubies are in the correct place, whether they are rotated correctly, and whether they contain a certain colour or not. As there are 3 different types of cubie (corner, edge and middle) each with a different number of colours, and therefore require different implementations of the methods above, I will use inheritance and polymorphism to achieve this in the following way. Methods outlined in are overridden methods.



|  |
| --- |
| A string used to identify the cubie e.g. the white,red,blue corner piece will be called “WRB” |
| A ***Vector3x1*** object storing the position of the cubie within the cube |
| An enumerated value representing the layer that the cubie is in |
| *See below for explanation of primary and secondary faces and rotations* |
| An array storing the colours of the cubie’s stickers |

|  |
| --- |
| Returns true if the cubie is in its solved position and is rotated correctly |
| Checks that a cubie is on the specified face, and that the cubie is rotated towards that face, doesn’t check the other 2 faces |
| Checks if a cubie is rotated correctly towards a given face (e.g. the cubies blue face is on the blue face of the cube - if blue is the argument face) |
| Checks if a cubie is on a given face |
| Checks if a cubie has a particular colour sticker |
| Sets a cubie’s colour from a string (its name) |

red boxes

#### Cubie rotations

To store the rotation of each cubie I will pick one face for each cubie and call it the ‘primary face’, the rotation for the cubie will then be the face that the primary face is on. For corner pieces I will need a secondary face and secondary rotation, as their rotation is around 2 axes, unlike edges which is only around 1.

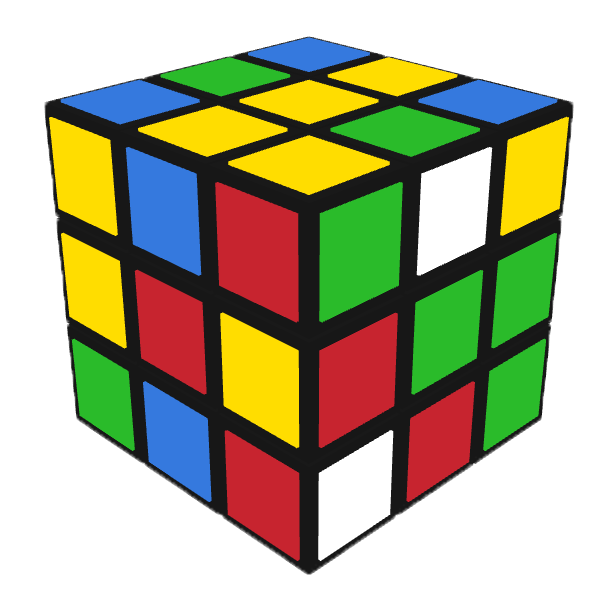


Figure 1: Example Rubik’s Cube

##### Example:

For the yellow, red, green corner (“YRG”) in  
  
**figure 1** opposite:

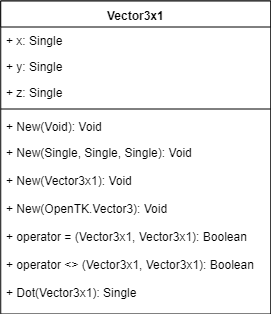
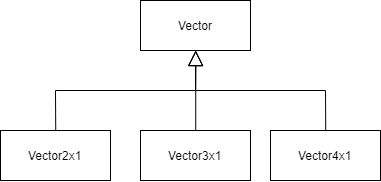
|  |  |
| --- | --- |
| Primary Face: | Y |
| Rotation: | Y |
| Secondary Face: | R |
| Secondary Rotation: | R |

##### List of Primary and Secondary Faces

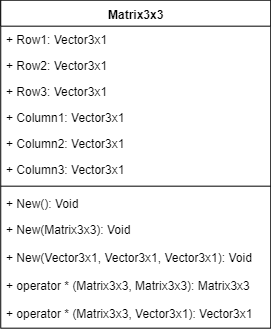
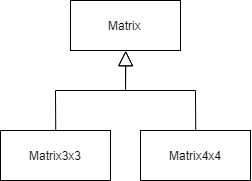
Primary edges are white or yellow where possible, else they are red or orange (except for middle pieces). Secondary faces are red or orange.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Cubie | Type | Primary | Secondary |  | Cubie | Type | Primary | Secondary |
| WRB | Corner | W | R |  | OB | Edge | O | - |
| WOB | Corner | W | O |  | OG | Edge | O | - |
| WOG | Corner | W | O |  | RG | Edge | R | - |
| WRG | Corner | W | R |  | YR | Edge | Y | - |
| YRB | Corner | Y | R |  | YO | Edge | Y | - |
| YRG | Corner | Y | R |  | YB | Edge | Y | - |
| YOG | Corner | Y | O |  | YG | Edge | Y | - |
| YOB | Corner | Y | O |  | W | Middle | W | - |
| WR | Edge | W | - |  | G | Middle | G | - |
| WB | Edge | W | - |  | R | Middle | R | - |
| WO | Edge | W | - |  | B | Middle | B | - |
| WG | Edge | W | - |  | Y | Middle | Y | - |
| RB | Edge | R | - |  | O | Middle | O | - |

### Vector

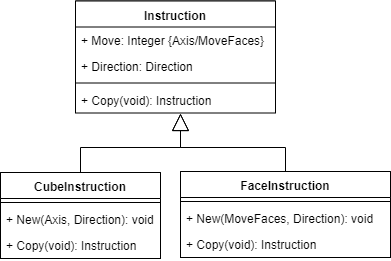
I will define my own vector classes. I will need to define 2D, 3D and 4D vectors so I will use inheritance from a base vector class, as shown below. Each class with have constructors, a dot product method, equality (=) and inequality (<>) operators, and properties to store their values: e.g. a 3D vector will have x, y and z properties. An example definition (3D) is shown on the right:

### Matrix

I will also define 3D and 4D square vectors for use as rotation matrices when rotating the position vectors of the cubies. As these matrices will only be used as rotation matrices they do not need all the general matric functions such as transpose and inverse etc. they will just need multiplication methods for multiplying with other square matrices, and with vectors of the same size as them (e.g. a 3x3 matrix with have methods for Matrix3x3\*Matrix3x3 and Matrix3x3\*Vector3x1). I will store the columns and rows of the matrices as vectors, so that I can use the vector dot product functions in the matrix multiplication functions. An example definition (3D square matrix) is shown on the right:

### Instruction

To store the instructions needed to solve the cube I will use an instruction class which stores the move to be done (e.g. rotate the top face) and the direction (e.g. clockwise). As there are 2 types of instructions: cube rotations and face rotations, I will use 2 more classes which inherit from ***Instruction*** like below. The instruction classes also each have a ToString() function which returns the instruction in Rubik’s Notation

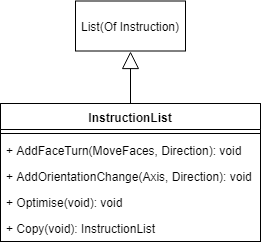


|  |
| --- |
| Stores the move either as an ***Axis*** or ***MoveFaces*** datatype dependant on the datatype of the object (cube/face instruction) |
| Stores the direction of the instruction as a ***Direction*** datatype |
| Returns a copy of the instruction |

### InstructionList

To store the instructions in order I will create an **InstructionList** Class. This will inherit from **System.Collections.Generic.List(Of T)** which is a generic class built into VB.Net. Specifically it will inherit from **List(Of Instruction)** this will give me a list in which to store the instructions in order, and inheriting from it will allow me to add my own methods such as methods to add the 2 different types of instructions into the list, and a method to optimise the instructions (as per objective 4.2). I am using a list rather than an array as it is an inheritable class, and this will allow me to group the data with the methods that act upon it.

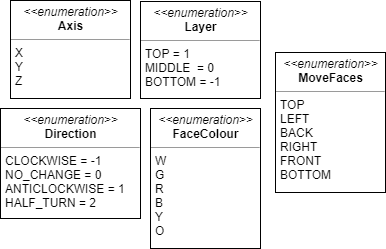
Every time that either the cube is rotated or a face of the cube is rotated, that function will call **InstructionList**.Add() and this will add the move to the end of the list. So as the cube is being solved, the list will automatically fill up with instructions. After the cube is solved, I will call **InstructionList**.Optimise() to remove any consecutive rotations of the same face/axis.



|  |
| --- |
| Adds a new ***FaceInstruction*** to the end of the list |
| Adds a new ***CubeInstruction*** to the end of the list |
| Optimises the instructions so that 2 or more consecutive turns of the same face/axis are merged into 1 |
| Returns a copy of itself |

### Enumerated Types

In my program I will use the following enums with the following members. This is mostly to improve readability of code and datatypes; however some, such as direction, will be used for their values. In the case of direction, it can be multiplied by pi/2 to give the angle for the rotation.



### TopLayerSolver

My LayerSolver classes will be aggregate classes containing the cube object that called it. They will act upon the cube (solve the top layer) and then execution will return to the altered cube so that the next layer can be solved. Below are lists of the main methods within the classes. For explanations please see the algorithms section.

|  |  |
| --- | --- |
| **Solve()** | Solves the top layer of the cube – the only public function. Checks if any face is completed and rotates it to the top if it is. Calls DoTopFace if no face is completed |
| **DoTopFace()** | Checks if any cross is completed and rotates it to the top if it is. Calls DoTopCross if none are completed |
| **DoTopCross()** | Makes a cross on the bottom face then inverts it to the top face |
| **DoTopCorners()** | Puts the corners in the correct place on the top face |
| **MakeWrongEdgedCrossOnBottomFace()** | Makes a cross on the bottom face (e.g. if white is top and yellow is bottom, a white cross is made on the yellow face) |
| **MakeCorrectCrossFromOppositeCross()** | Inverts the cross from the bottom face to the correct place on the top face |
| **PutCornerIntoCorrectPlaceOnTopFace(**Corner**)** | Puts a specified corner into the correct place on the top face |
| **GetAnyCompleteFace() :** FaceColour | Returns the first completed face (if there is one) if it is correctly rotated |
| **GetAnyRotatedCompleteFace() :** FaceColour | Returns the first completed face (if there is one), doesn’t check rotation |
| **GetAnyCrossWithCorrectEdges() :** FaceColour | Returns the first completed cross whose edges are correct (if there is one) |

### MiddleLayerSolver

|  |  |
| --- | --- |
| **Solve()** | Solves the middle layer of the cube – the only public function. Checks if the middle layer is already solved |
| **MiddleRowCorrect() :** Boolean | Returns true if the middle layer is solved |
| **DoMiddleRow()** | Iterates through each edge that should be in the middle and puts it in its correct place if it is not already |
| **PutEdgeInMiddle(**Edge**)** | Puts the specified edge piece into the correct place in the middle row |

### BottomLayerSolver

|  |  |
| --- | --- |
| **Solve()** | Solves the bottom layer of the cube if the cube is not already solved – the only public function. |
| **SolveBottomLayer()** | Solves the bottom cross if it’s not already solved. Then solves the corners. |
| **BottomFaceComplete() :** Boolean | Returns true if the bottom face is complete |
| **DoCross()** | Makes the bottom cross (see algorithms section) |
| **PermuteTopCrossEdges()** | Permutes the bottom cross edges correctly (at this point it is the top cross because the cube is upside down) |
| **DoCorners()** | Solves the bottom corners |
| **PositionBottomCorners()** | Positions the bottom corners in the correct places |
| **OrientateBottomCorners()** | Orientates the bottom corners correctly |

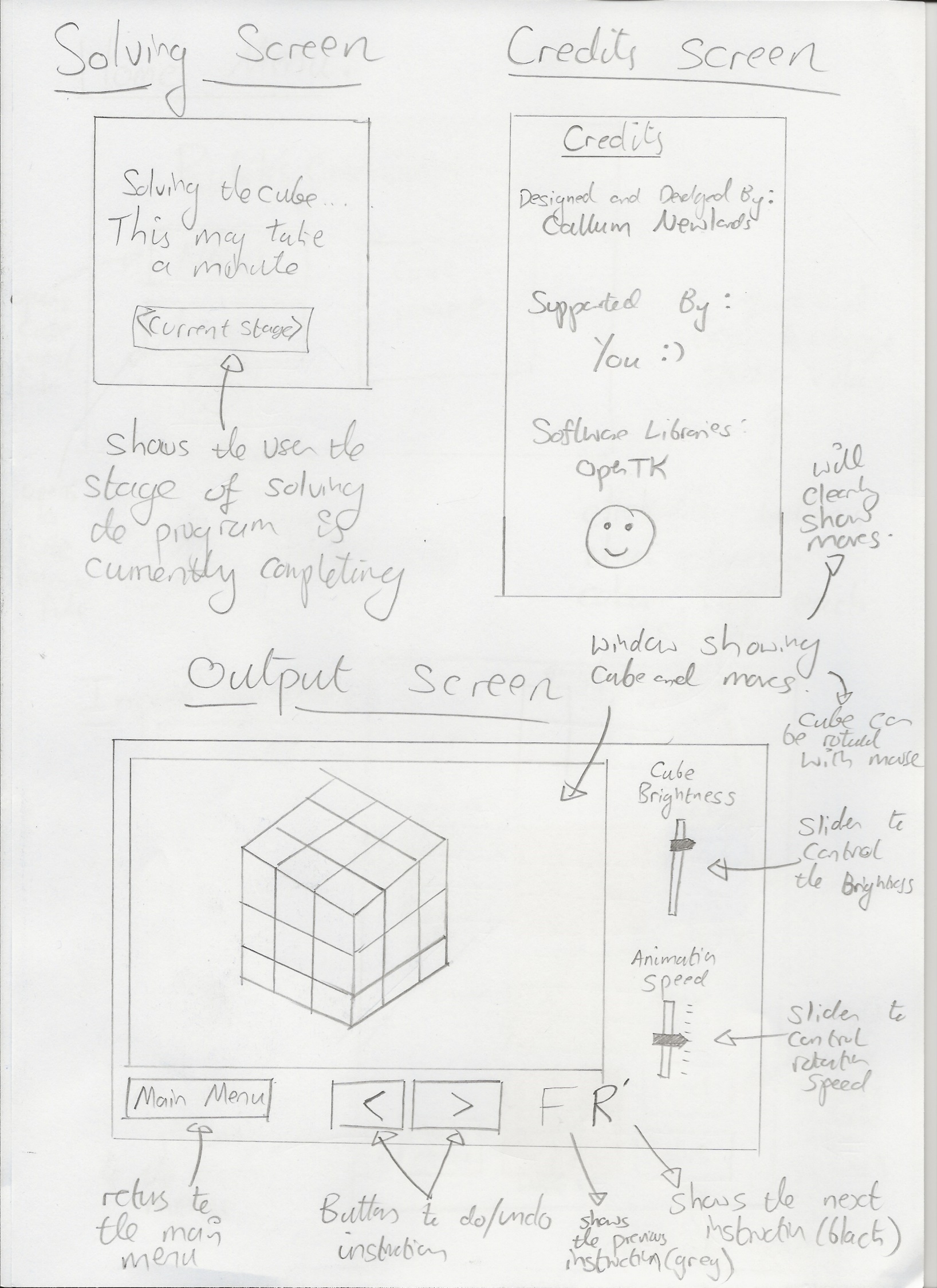
### 3DOutput

This is the form (and the main class) that will handle rendering the 3D cube and displaying the instructions to the user. For the output, instead of using my custom matrix and vector classes, I will use the matrix, vector and quaternion types that are built into OpenTK as these are optimised for 3D graphics, and contain all of the necessary features for use with OpenGL.

As the 3DOutput form will be heavily event driven, there is a need for (private) variables which are in scope throughout the entire class, as they cannot be passed into the event handlers like you would do with normal methods.

Please note that I have not included any event handler methods in my diagram. The following event handlers will be present in my class:

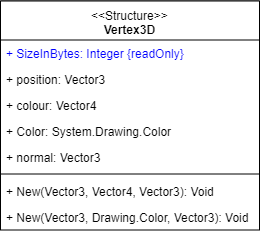
|  |  |  |
| --- | --- | --- |
| **Event** | **Handler** | **Use** |
| When the OpenGL window (GLControl1) first loads | **GlControl1\_Load** | Calls the methods to set up the graphics and geometry for rendering |
| When the OpenGL window is refreshed | **GlControl1\_Paint** | Renders the cube and the current instruction to the screen |
| When the user hovers over the next/undo buttons | **btnUndo\_Hover, btnNext\_Hover** | Colours the next/previous instruction red (so that the user can see the next instruction that will be shown) |
| When the user’s mouse leaves the next/undo buttons | **btnUndo\_MouseLeave, btnNext\_ MouseLeave** | Resets the colours of the labels once the user is no longer hovering over the buttons |
| When the user clicks on the OpenGL window | **GlControl1\_MouseDown, GlControl1\_MouseUp,** | Used to update the positions of the camera (i.e. to rotate the cube model with the mouse) |
| When the user clicks the undo button | **btnUndo\_Click** | Undoes the previous instruction and adds the corresponding rotations to the cubies so that the rotation is displayed on the model |
| When the user clicks the next button | **btnNext\_Click** | Does the next instruction and adds the corresponding rotations to the cubies so that the rotation is displayed on the model |
| When the form is closed | **\_3DOutput\_FormClosed** | Disposes of the shader program |
| When the main menu button is clicked | **btnMain\_Click** | Checks if the user wants to save, then opens a new instance of the mainMenu form and closes this form |



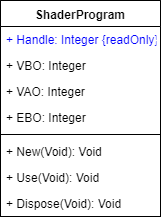
A clipping from the design for the 3DOutput form

|  |
| --- |
| Constant to store the padding between cubes |
| The projection matrix for the scene – readOnly as it will not change |
| Stores the initial positions and rotations of the cubies, and the cube which was inputted into the program |
| Stores the list of instructions and the pointer to the current instruction |
| Stores the handles to the shader program, the VAO, VBO and EBO |
| Stores the current orientation of the camera – used to rotate the cube |
| Stores the difference in time between the current frame and the previous frame – used to ensure current rotation speed, regardless of frame rate |
| Stores the state of the cube being displayed on the screen – used to work out which cubies to apply the rotation matrix to |
| Stores the rotation matrices for each cubie |
| If this is true, the instruction is still being shown to the user |
| Constructors, one from a cube and one from a file path |
| Methods to load the initial state of the window, and to get the initial positions and rotations of the cubies from the initial scrambled cube |
| Renders the cube to the screen with any previous and current rotations applied |
| Updates the labels showing the instructions to show the current and previous instructions |
| Methods to carry out the next/previous instruction |
| Methods to rotate a particular face/the cube by adding rotation vectors/matrices to the correct cubies’ rotationsStores |
| Saves the cube to a .cube and a .cube.ptr files |

### Vertex3D

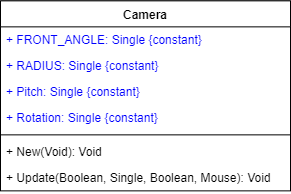
To store the information about each cubie vertex (relative position, colour and normal vector) I will use a structure, as it will just be used more as a datatype than as an object (like a class), and then I will transfer each of the properties on the vertex across to OpenGL via VertexAtrributePointers. In order to use the vertices in an OpenGL buffer I will need to know their size in memory, so I will also need to store their size in bytes as a readOnly property. I will write 2 constructors for them, one which takes the colour as an OpenTK.Vector4 (the default for storing colours in OpenTK) and one which takes it as a VB.NET System.Drawing.Color so that I can use either if needs be in my code, or when adding functionality at a later date.

### ShaderProgram

As I said earlier, in order to render the cubes I will use a Vertex Array Object (VAO), a Vertex Buffer Object (VBO) and Entity Buffer Object (EBO). As I will only need one instance of each of these, and only 1 shader program (which I will compile the shaders to in the setup functions), it makes sense to store the handles (dynamically assigned integers) for the shader program, VAO, VBO and EBO together in a class. I will also need a method to dispose of the buffers when the program finishes.

In the constructor it will handle the compiling of the 2 shaders, as well as binding them to the shader program, and will also handle the GLSL version control (see algorithms section for how).

### Camera

In order to rotate the cube model (or give the appearance of rotating it) I can simply change the view matrix. In order to do this I will create a LookAt camera matrix and set the view matrix equal to this. This will allow me to control the position of the camera what the camera looks at (it’s rotation), as the camera will only ever be looking at the cube (the origin) this will be constant, but in order to store the camera’s rotation and pitch (up and down rotation) I will need a camera class to store and manipulate these values.

To rotate the camera around the cube I will store an angle, and then use trigonometry to get the camera’s X and Z-coordinates. I will then use the stored pitch as the camera’s Y-coordinate. Because the camera will always be oriented towards the cube this will cause the camera to look underneath/above the cube. I.e. it will give the appearance of rotating the cube.

The camera class will have an update function which will be called whenever the user clicks on the OpenGL window (clicks on the cube). It will take the mouse position as an argument, and will update the camera’s pitch and rotation depending on where the users mouse is moved to – allowing the user to rotate the cube with their mouse.

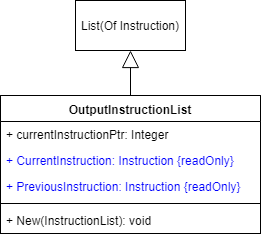
I will also limit the pitch between 2 values (I will have to experiment to find these) and make the cube return to its default pitch when the user’s mouse is not clicked (like a spring being released when the mouse is released).

### OutputBlock

For each cubie rendered on the screen I will need to store its previous and current rotations matrices, and whether it is currently being rotated, so that I can rotate it correctly when rendering it and know whether to show its current rotation or not. I will store these in an output block class, which will be stored as 27 objects in an array. See the OpenGL Design section for a run-through of how I will use this class.

### RotationStore

I will use this class to store each cubie’s previous and current rotations. I will store the end result of all the previous rotations as a rotation matrix (which is the product of all the previous rotation matrices). I will store the current rotation as an axis and an angle. This will allow me to create a new rotation matrix each time, with a different angle (given as a percentage of the max angle) – which will allow the cube to rotate on the screen. I will have a function called GetTotalMatrix that will take a percentage as an argument, and will return the product of the previous rotation matrix, and the newly created rotation matrix from the current axis and angle. It will also check if the rotation is finished (percentage >= 100%) and if so it will multiply this final rotation matrix with the previous ones, and will clear the current axis and max angle.



### OutputInstructionList

This class will be similar to the InstructionList class used in the solving of the cube, and will again inherit from List(Of Instruction), but will not have any methods to add/optimise instructions. It will instead have a constructor that can convert an InstructionList to an OutputInstructionList, and will store a pointer to the current instruction. It will also have readOnly properties that will return the current and previous instructions.

## Class Diagram

## Data Dictionary – Main Variables

Below is a dictionary of the most essential variables that will be used in my program along with their datatypes, any validation performed on them and what they will be used for. I have not included any iterator variables, or variables that will store form objects (i.e. the objects for the windows displayed). I have also not included constants or ReadOnly variables.

As there is little user input, aside from a bit in the input and output forms, most of the variables do not require direct validation as they are only affected by the computer and not the user.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **File** | **Variable Name** | **Data Type** | **Use** | **Validation** |
| MainMenu.vb | fileBrowser | OpenFileDialog | Used to select a file to load. The file path from the fileBrowser is then passed to the 3DOutput form for loading. | A filter is applied to the fileBrowser to only show .cube files. Also the filePath is checked for being blank, and any exceptions with loading the file are caught. |
| Input.vb | stickerColours | Char(,) | Used to store the colours of the user’s cube, inputted by the buttons on the form. | The only method of inputting the colours will be by cycling through the colours with the mouse, so this will limit the inputted data to one of the 6 colours. |
| Input.vb | cube | Cube | Created from stickerColours, and then passed into the Processing form to be solved | Before the cube is created the stickerColours array is sent to the Anaylse module and checked if it is a possible cube (see algorithms section for more details). |
| Processing.vb | \_cube | Cube | The cube which will be solved and have its instructions sent to the 3DOutput form | This is the same cube that is passed in from the Input form, which is checked for legality before it is passed in |
| Processing.vb | scrambledCorners | Corner() | The inputted corners before the cube is solved, to be passed to the 3DOutput form for later use |  |
| Processing.vb | scrambledEdges | Edge() | The inputted edges before the cube is solved, to be passed to the 3DOutput form for later use |  |
| TopLayerSolver.vb/ MiddleLayerSolver.vb/ BottomLayerSolver.vb | cube | Cube | This is the cube that is being solved by the solver class. It is passed in by reference via the constructor |  |
| 3DOutput.vb | shaderProgram | ShaderProgram | Stores the handles to the shader program, the VAO, VBO and EBO, is also used to control the version of GLSL code that is run | If compiling fails (i.e. because of an unsupported OpenGL version,) the handle is set to -1, and an error message is displayed to the user. |
| 3DOutput.vb | camera | Camera | Used to store the current orientation of the camera, which is in turn used to rotate the cube/scene |  |
| 3DOutput.vb | deltaTime | Single | Stores the difference in time between the current frame and the previous frame – used to ensure current rotation speed, regardless of frame rate | Checked that if it greater than 0, if not the program loops until enough time has elapsed |
| 3DOutput.vb | isMouseDown | Boolean | Set to true is the mouse is down, and reset to false when the mouse is released | Boolean values can only be true/false |
| 3DOutput.vb | currentCube | Cube | Stores the current cube being displayed on the screen – used to work out which cubies to apply the rotation matrix to – e.g. if the front face needs rotated, this is used to work out which cubies are on the front face |  |
| 3DOutput.vb | cubieRotations | OutputBlock() | Stores the previous rotation matrix and current rotation for each cubie |  |
| 3DOutput.vb | showingRotation | Boolean | If this is true, the instruction is still being shown to the user | Boolean values can only be true/false |

## Files and Saving

To store the cubes, I will use binary files with a custom (.cube) extension. This is because binary files will allow me to save and load complex data structures (such as vectors and objects). It will also be quicker to write to the file, rather than converting all my data to strings and then writing all these strings to a text file, only to then convert them back to objects when I reload the file. It may also be more space efficient in memory than a text file as numbers etc. can be stored as their pure binary representations rather than ASCII characters.

I will use binary serialisation to store my cube object. In binary serialisation, all of the object’s data (including any objects contained with the object being serialised) are converted to a stream of bytes which can then be written to a binary file. This allows a complete clone of the object to be created from the file when it is deserialised. When an object is serialised in VB.net, the serialisation engine checks object references (of the objects contained within the main object being serialised) to ensure that objects are only serialised once – this reduces file sizes and stops redundant data storage.

Cubes will only be saved once the user is at the output stage of the program, this means that my program will also need some way of saving the current instruction that is being displayed – i.e. the current state of the cube mid-solving. Because the instructions will be stored with a list (inside the *OutputInstructionList* class) to store the current instruction I will just need to save the current pointer that points to the instruction in the list currently being displayed. I will store this also in a binary file with a custom (.cube.ptr) extension. This use of a separate file will enable me to save the cube and the pointer more easily, whilst maintaining separation of modules (i.e. the code to save the pointer will not rely on the code to save the cube). It will also allow me to load the 2 objects (the cube and the pointer) more easily as I will not have to separate them once I have loaded the files. One issue however is that the user may delete the (.cube.ptr) file and then try and load the (.cube) file, this would result in a *FileNotFoundException*, to combat this I will need to check that the (.cube.ptr) file exists and if not output a suitable message to the user.

## Algorithms

Below are my algorithms for the most important/complex methods in my program. For an overview of the whole system algorithm please see my system flowchart at the start of the design section.

### Validating the Cube

Class: Analyse

#### IsCubePossible()

This function is the main function that will check if the cube is possible, rather than using 1 IF statement with 4 AND operators, I will use 3 separate IF statements so that if it fails the 1st test it won’t need to carry out the remaining 3, in order to make my code more efficient.

FUNCTION IsCubePossible(stickerArray) AS BOOLEAN

IF NOT NineOfEachColourSticker(stickerArray) THEN RETURN FALSE

IF NOT EdgesPossible(stickerArray) THEN RETURN FALSE

IF NOT CornersPossible(stickerArray) THEN RETURN FALSE

RETURN CornerRotationsPossible(stickerArray)

END FUNCTION

#### NineOfEachColourSticker()

This function gets the number of each colour sticker as an array, and then checks that each number in that array is 9. I.e. that there are 9 of each colour of sticker. The **NumberOfEachColour** function will iterate through the stickerArray with a nested loop (as it is a 2D array) and will add use the colourChar (e.g. “W”) to get the index of the array for that colour, using the formula:

FUNCTION NineOfEachColourSticker(stickerArray) AS BOOLEAN

IF AnythingElseInArray(NumberOfEachColour(stickers), 9) THEN  
 OUTPUT MESSAGE TO USER   
 RETURN FALSE  
 END IF   
 RETURN TRUE  
END FUNCTION

Index 🡸 InStr(FACE\_COLOURS, stickerArray(i, j)) - 1

Where InStr in a function that returns the position of a character in a string, and FACE\_COLOURS is a constant string = "BGORWY".

The **AnythingElseInArray** function will iterate through the array and return false if the element does not match the 2nd argument (there are one or more elements that are not the 2nd argument).

#### EdgesPossible()

FUNCTION EdgesPossible(stickerArray) AS BOOLEAN

FOR faceNumber 🡸 0 TO 5

FOR edgeNumber 🡸 1 TO 7 STEP 2

adjacentSticker 🡸 AdjacentEdge(faceNumber, edgeNumber)

oppositeColour 🡸 Opposite(stickerColour)

IF (stickerColour = adjacentStickerColour

OR adjacentStickerColour = oppositeColour) THEN

OUTPUT MESSAGE TO USER   
 RETURN FALSE  
 END IF

NEXT edgeNumber

NEXT faceNumber

RETURN TRUE

END FUNCTION

This function iterates through each edge sticker with a nested loop and returns true if any of them are not possible – i.e. that an edge piece contains 2 of the same colour stickers or opposite colour stickers. The outer loop iterates across the 6 faces of the cube and the inner loop iterates across the 4 edge stickers on each face (1, 3, 5, & 7).

The colours of the sticker and adjacent sticker will be gotten from the stickerArray using the formula below. I have excluded this from my algorithm to improve readability and clarity.

stickerColour 🡸 stickerArray(faceNumber, edgeNumber)

The adjacentSticker is the other sticker that makes up the edge piece that the sticker is on. It will be gotten from the **AdjacentEdge** function which will take in 1 sticker and return its matching sticker. And the oppositeColour is the colour that is on the other side of the cube to the stickerColour.

#### CornersPossible

The **CornersPossible** function will work very similarly to the **EdgePossible** one, but instead of checking 1 adjacent sticker colour, it will have to check against the other 2 stickers that make up the corner piece.

#### CornerRotationsPossible

FUNCTION CornerRotationPossible(stickerArray) AS BOOLEAN

FOR faceNumber 🡸 0 TO 5

FOR cornerNumber 🡸 0 TO 8 STEP 2

IF cornerNumber = MIDDLE\_STICKER THEN CONTINUE FOR

cornerStickers 🡸 AdjacentCorners(faceNumber, cornerNumber)

IF 1stStickerColour <> “W” AND <> “Y” THEN NEXT cornerNumber

IF 3rdStickerColour <> getCorrectThirdStickerColour(cornerStickers) THEN

OUTPUT MESSAGE TO USER   
 RETURN FALSE

END IF

NEXT cornerNumber

NEXT faceNumber

RETURN TRUE

END FUNCTION

This function will iterate through each corner sticker until it finds a white or yellow stickers (as every corner will have either a white or yellow face) it then gets the 2 other stickers from the **AdjacentCorners** function, and uses the 1st and 2nd stickers to calculate what the 3rd should be (via the **getCorrectThirdStickerColour** function). It will then compare the actual 3rd sticker colour to what it should be to work out if the corner is valid (i.e. that the 3 stickers are in the correct orientation relative to one another)



Diagram showing the indices of my stickerArray with Top=White and Front=Red

### Converting StickerArray to Cube Object

Class: Cube/SetCubieProperties

This algorithm will be used in the constructor of my cube class, and will take in my array of sticker colours, and convert them into a cube object. It will orientate the sticker array so that the white face is up and the red face is at the front, then will convert the top half of the cube. It will then orientate the sticker array so that the yellow face is up and convert the top half (which was previously the bottom half). It will do it in 2 halves because it will be hard to know the orientation of the bottom face in relation to the other 4, whereas if I only look at 5 faces at a time (the top one and the 4 sides), then I will know easily the orientation of the other 4 faces as they will always be in this arrangement in relation to the top face:

In this algorithm, CreateMiddleBlocks is a subroutine that creates the 6 required **Middle** objects with the correct position vector, colour and rotation (equal to their colour) and stores them in the Cube.Middles() array. This is quite simple because the cube will always have the same middle pieces in the same places as they cannot rotate relative to each other, and the rest of the cube uses them as a reference.

SUBROUTINE ConvertTopCorners(stickerArray)

FOR face 🡸 MoveFaces.TOP to MoveFaces.FRONT # 0 to 4

FOR cornerNo 🡸 0 TO 8 STEP 2

sticker 🡸 New Sticker(face, cornerNo)

IF (sticker is not on top half of cube) OR

(sticker colour is not primary colour) THEN NEXT cornerNo

cornerTriplet 🡸 New CornerTriplet(sticker)

colourString 🡸

GetCornerColoursAndSecondaryFaceAndRotation(CornerTriplet, secondaryFaceColour, secondaryFaceRotation)

#search CORNER\_NAMES to get the index of the current corner then set its properties

FOR index 🡸 0 TO 7

IF CORNER\_NAMES(index) <> colourString THEN NEXT index

Corners(index).Position 🡸 GetCornerPosition(face, cornerNo)

Corners(index).Rotation 🡸 GetFaceOf(cornerNo)

Corners(index).SecondaryFace 🡸 secondaryFaceColour

Corners(index).SecondaryRotation 🡸 secondaryFaceRotation

Corners(cornerNumber).SetColoursFromColourString(cornerString)

NEXT index

NEXT cornerNo

NEXT face

END SUBROUTINE

Cube.CurrentOrientation 🡸 New CubeOrientation(White, Red)

stickerArray 🡸 OrientateStickerArray(CurrentOrientation)

ConvertTopCorners(stickerArray)

ConvertTopEdges(stickerArray)

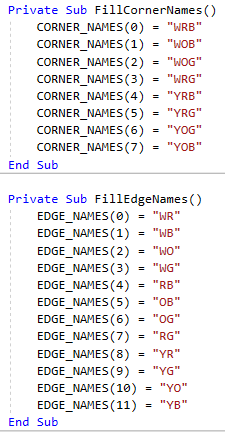
stickerArray 🡸 OrientateStickerArray(CurrentOrientation.UpsideDown)

ConvertTopCorners(stickerArray)

ConvertTopEdges(stickerArray)

CreateMiddleBlocks()

This subroutine uses a nested loop to iterate through the stickerArray. If the current corner sticker is not on the top half of the cube or is not a primary sticker (white or yellow) then it goes to the next corner. When it finds a primary sticker on the top half of the cube it creates a **cornerTriplet** object (a group of 3 stickers that make up a corner) and converts the colours of each sticker into a colour string in the correct order (primary colour, secondary colour and final colour) (see right for colour strings) so that it matches with the names in the CORNER\_NAMES array (a readOnly array of corner colour strings that is used to get the index of a particular corner). Because when creating the corner string it must work out which colour is the secondary colour, I will get the secondaryFaceColour and secondaryRotation at the same time as getting the colour string.



Corner Colour Strings/Names

The GetCornerColoursAndSecondaryFaceAndRotation function will simply iterate through the colours in the corner triplet and use an IF-ELSEIF-ELSE statement to work out which position in the string each colour should be (white/yellow = 0, red/orange = 1, else = 2) and then will create and return the string. In the part of the IF statement that handles red/orange it will also set the secondaryFaceColour and secondaryRotationColour, which will both be passed by reference so that they can be used in the subroutine the called it.

The position vector of the corner will depend on 3 things, whether the sticker array is ‘upside down’ or not, the face that the sticker is on and the position of the corner on that face (its cornerNo in the algorithm above). The table below shows the formulae that I have created to give the x, y and z components of the position vector. Sign() is a function that returns -1 if a number is negative and 1 if a number is positive, and Abs() is a function which returns the absolute (positive) value of a number.

Not upside down (CornerNumber = c):

|  |  |  |  |
| --- | --- | --- | --- |
| Face | X-formula | Y-formula | Z-formula |
| Top | c Mod 6 - 1 | 1 | Sign(c – 4) |
| Left | -1 | 1 | 1 – Abs(c – 2) |
| Back | Abs(c – 2) - 1 | 1 | -1 |
| Right | 1 | 1 | 1 - c |
| Front | c - 1 | 1 | 1 |

Upside down (CornerNumber = c):

|  |  |  |  |
| --- | --- | --- | --- |
| Face | X-formula | Y-formula | Z-formula |
| Top | Abs(c Mod 6 – 2) - 1 | -1 | Sign(c – 4) |
| Left | 1 | -1 | 1 – Abs(c – 2) |
| Back | c - 1 | -1 | -1 |
| Right | -1 | -1 | 1 - c |
| Front | Abs(c – 2) - 1 | -1 | 1 |

As you can see the Y coordinate is known simply from whether the cube is upside down or not, and the Z-formulae do not change between the 2 tables, as the front face of the cube stays the same. Also, for the second table, the X-formulae simply swap between the pairs of opposite faces (e.g. the front and back formula swap between the 2 tables). To work out these formulae, I created a table of faces and corner numbers and the desired X and Z components of the vector and calculated the correct formula that would give these components.

For the **ConvertTopEdges** function, the overall algorithm will be mostly the same, the differences will be that when calculating is the sticker is the primary colour it will have to check whether it is an edge piece with a red/orange primary face as well as a white/yellow, so I will use this function:

Return (

(stickerColour = "W" OR stickerColour = "Y") OR

((stickerColour = "R" OR stickerColour = "O") AND secondStickerColour <> "W" AND secondStickerColour <> "Y")

)

Which returns true if the sticker is white or yellow, or if the sticker is red or orange, as long as the other face of the edge is not yellow or white.

There is also no secondaryFace or secondaryRotation for edge pieces, so that bit of the algorithm won’t be needed for the edges.

The other main difference between the corner and edge algorithms is the formulae for the position vectors, as edges are in different positions, and can also be on the middle row, so they need formulae for the y-component too. Here are my formulae for the edges:

Not upside down (edgeNumber = c):

|  |  |  |  |
| --- | --- | --- | --- |
| Face | X-formula | Y-formula | Z-formula |
| Top | (c Mod 6) Mod 3 - 1 | 1 | Select Case c  Case 1 : z = -1  Case 3 : z = 0  Case 5 : z = 0  Case 7 : z = 1  End Select |
| Left | -1 | Sign(3 - c) | c Mod 3 - 1 |
| Back | Sign(c - 1) | Sign(3 - c) | -1 |
| Right | 1 | Sign(3 - c) | Sign(c - 1) |
| Front | c Mod 3 - 1 | Sign(3 - c) | 1 |

Upside down (edgeNumber = c):

|  |  |  |  |
| --- | --- | --- | --- |
| Face | X-formula | Y-formula | Z-formula |
| Top | -((c Mod 6) Mod 3 – 1) | -1 | Select Case c  Case 1 : z = -1  Case 3 : z = 0  Case 5 : z = 0  Case 7 : z = 1  End Select |
| Left | 1 | -(c Mod 3) | c Mod 3 - 1 |
| Back | c Mod 3 - 1 | -(c Mod 3) | 1 |
| Right | -1 | -(c Mod 3) | Sign(c - 1) |
| Front | Sign(c - 1) | -(c Mod 3) | -1 |

As with the corner formula, the Z-formulae do not change between the 2 tables, and the X-formulae swap between the pairs of opposite faces. I could not find a formula for the Z-component of the top face, so I instead used a select case and hard-coded the 4 possible values.

### OrientateStickerArray



StickerArray in net form, matching the input screen

Class: SetCubieProperties



StickerArray in ‘relational form’ with Top=White and Front=Red

In order to correctly process the stickerArray to convert it into a cube object I need to be able to have the stickers in a certain orientation (see below) based off of a given top and front face. This algorithm will convert a sticker array from net form (left image) to ‘relational form’ (right image) in which 4 faces are given in an orientation relative to the top face. The 1st index of the array will give the faceNumber of the sticker and will match the values of my MoveFaces enum, and the 2nd index will give the sticker position in relation to the top face (see right diagram). Because the cube net I am using has only 1 line of symmetry, in order to map the faces into the relational form, I will have to use different algorithms based on the front and top face colours, and the face that I am converting. Each face can be in 1 of 4 orientations, so I will need 4 difference mapping algorithms. In order to find patterns between the 3 variables (top colour, front colour, and face colour) I used this spreadsheet:



The method numbers correspond to the following situations, and map the sticker numbers from (0, 1, 2, 3, 4, 5, 6, 7, 8) to the order shown in the table. For example, for method 3: (0 -> 6, 1 -> 3, …)

|  |  |  |
| --- | --- | --- |
| Method Number | Situation | Order |
| 1 | Face is in normal orientation | 0, 1, 2, 3, 4, 5, 6, 7, 8 |
| 2 | Face is upside down | 8, 7, 6, 5, 4, 3, 2, 1, 0 |
| 3 | Face is rotated left | 6, 3, 0, 7 , 4, 1, 8, 5, 2 |
| 4 | Face is rotated right | 2, 5, 8, 1, 4, 7, 0, 3, 6 |

Purple cells are ones where the face colour matches the top face colour, and orange cells are ones in which the face colour is orange – in most cases these are the 2 cells which have a different method to the rest.

I will use nested select cases to select which method is correct for each case, because the face colour equalling the top face is often a different method, I will write another function that handles converting the top face, to simplify the readability of my select cases.

Here are the select cases I will use. They follow the patterns that are visible in the spreadsheet.

SELECT CASE TopFace

CASE White

SELECT CASE faceColour

CASE Orange : Method2()

CASE TopFace : ConvertTopFace()

CASE Else : Method1()

END SELECT

CASE Yellow

SELECT CASE faceColour

CASE Orange : Method1()

CASE TopFace : ConvertTopFace()

CASE Else : Method2()

END SELECT

CASE Green

SELECT CASE faceColour

CASE TopFace : ConvertTopFace()

CASE Else : Method3()

END SELECT

CASE Blue

SELECT CASE faceColour

CASE TopFace : ConvertTopFace()

CASE Else : Method4()

END SELECT

CASE Red

SELECT CASE faceColour

CASE Yellow : Method1()

CASE White : Method2()

CASE Blue : Method3()

CASE Green : Method4()

CASE TopFace : ConvertTopFace()

END SELECT

CASE Orange

SELECT CASE faceColour

CASE Yellow : Method2()

CASE White : Method1()

CASE Blue : Method4()

CASE Green : Method3()

CASE TopFace : ConvertTopFace()

END SELECT

END SELECT

And for the top face:

SUBROUTINE ConvertTopFace()

IF FrontFace = Blue Then

Method3()

ELSEIF FrontFace = Green Then

Method4()

ELSEIF FrontFace = White Then

IF TopFace = Orange Then

Method1()

ELSE

Method2()

END IF

ELSEIF FrontFace = Yellow Then

IF TopFace = Orange Then

Method2()

ELSE

Method1()

END IF

ELSEIF FrontFace = Red Then

SELECT CASE TopFace

CASE White : Method1()

CASE Yellow : Method2()

CASE Green : Method3()

CASE Blue : Method4()

END SELECT

ELSEIF FrontFace = Orange Then

SELECT CASE TopFace

CASE White : Method2()

CASE Yellow : Method1()

CASE Green : Method4()

CASE Blue : Method3()

END SELECT

END IF

END SUBROUTINE

The algorithms I will use to map the indices of the array are as follows:

FaceNumber is the number of the face (given by its colour) and faceStickers is an array from 0 to 8, which contains the stickers of the face as they were in net form.

|  |  |
| --- | --- |
| Method | Algorithm |
| 1 | FOR i 🡸 0 TO 8  stickerArray(faceNumber, i) 🡸 faceStickers(i)  NEXT |
| 2 | FOR i 🡸 0 TO 8  stickerArray(faceNumber, i) 🡸 faceStickers(8 - i)  NEXT |
| 3 | count 🡸 6  FOR i 🡸 0 TO 8  stickerArray(faceNumber, i) 🡸 faceStickers(count MOD 10)  count 🡸 count + 7  NEXT |
| 4 | count 🡸 2  FOR i 🡸 0 TO 8  stickerArray(faceNumber, i) 🡸 faceStickers(count MOD 10)  count 🡸 count + 3  NEXT |

### Block.OnFace

Class: Block

FUNCTION OnFace(face) AS BOOLEAN

faces 🡸 cube.FaceColours()

RETURN (face = faces(MoveFaces.TOP) AND Position.y = 1 OR

face = faces(MoveFaces.BOTTOM) AND Position.y = -1 OR

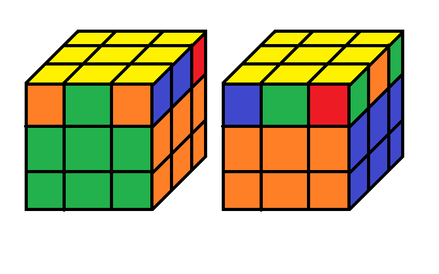
face = faces(MoveFaces.FRONT) AND Position.z = 1 OR

face = faces(MoveFaces.BACK) AND Position.z = -1 OR

face = faces(MoveFaces.LEFT) AND Position.x = -1 OR

face = faces(MoveFaces.RIGHT) AND Position.x = 1)

END FUNCTION

This function (above) will use the block’s position vector to work out whether it is on the correct face. E.g. if the face is front, the block’s z-coordinate will be 1 if it is on the front face. Cube.FaceColours is an array of FaceColours which is dependent on the cube’s orientation and matches up with the values of the MoveFaces enum. So for example, MoveFaces.FRONT = 4, and FaceColours(4) (or FaceColours(MoveFaces.FRONT) is the colour of the front face of the cube.

### Block.CorrectRotation

Class: Block

This function (CorrectRotation) will compare the orientation of one of the cubie’s faces with the cube using the syntax CheckRotation(cubieColour, cubeColour). This will allow me to check that cubies are correctly rotated for 1 face, but ignore the other 2 faces. For example in the image on the right, the Yellow-Orange-Blue corner piece is correctly rotated for the yellow face, but not for the green or orange faces. So Block.CorrectRotation(yellow, yellow) will return true, as will Block.CorrectRotation(orange, green), but Block.CorrectRotation(orange, orange) will not. This will be useful throughout my program in areas such as working out if a face is completed but is rotated incorrectly.

As each of the 3 types of cubie will have different axes of rotation and a different amount of colours, I will use polymorphism and create a different **CorrectRotation** method for each Block class.

#### Corner.CorrectRotation

FUNCTION CorrectRotation(cubieColour, cubeFaceColour) AS BOOLEAN

IF NOT HasColour(cubieColour) THEN RETURN FALSE

SELECT CASE cubieColour

CASE White, Yellow

IF Rotation <> cubeFaceColour THEN RETURN FALSE

CASE Red, Orange

IF Rotation = cubeFaceColour OR SecondaryRotation <> cubeFaceColour THEN RETURN FALSE

CASE Blue, Green

IF Rotation = cubeFaceColour OR SecondaryRotation = cubeFaceColour THEN RETURN FALSE

END SELECT

RETURN TRUE

END FUNCTION

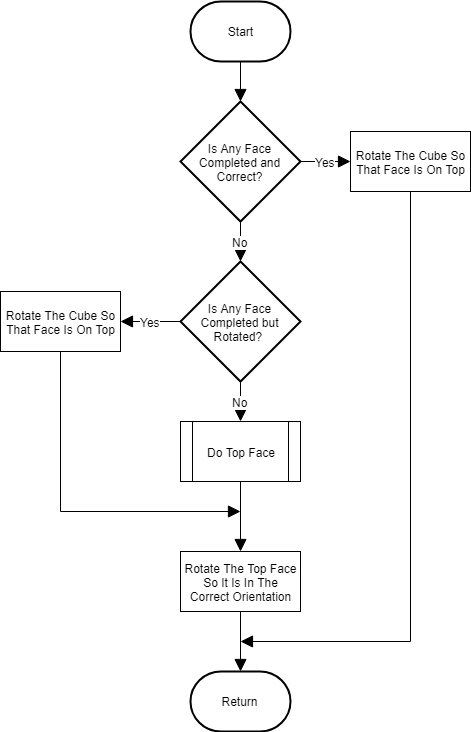
This function (above) first checks that the cubie has the colour sticker that is being checked, and then uses the following properties to work out if the rotation is correct:

* If the cubieColour is the primary face then in order for the rotation to be correct, the cubeFaceColour must be equal to the cubie’s primary rotation
* If cubieColour is the secondary face, then in order for the rotation to be correct, the cubeFaceColour must be equal to the cubie’s secondary rotation, and the rotation must not be equal to the cubie’s primary rotation
* If the cubieColour is neither the primary or secondary face, then the cubeFaceColour must be neither the primary or secondary rotations

#### Edge.CorrectRotation

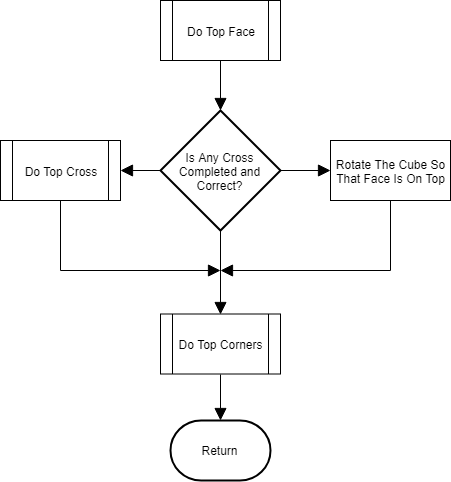
A similar function will be used for the Edge class, with the exception that within the Red/Orange case it must also check whether the Red/Orange (the cubieColour) is the primary face or not, and then from that will determine the checks it will need to make. Also as the Edge pieces have no secondary face or rotation, they only need to check the primary rotation against the cubeFaceColour.

#### Middle.CorrectRotation

The Middle Class **CorrectRotation** function will simply return true always, as the middle piece cannot be rotated away from its face (as the middle pieces are fixed in relation to each other) so its rotation will always be correct.

### TopLayerSolver

Class: TopLayerSolver





SUBROUTINE MakeTopCrossOnBottomFace()

edgePieces 🡸 cube.BlocksByColour(cube.TopFace).Extract(Edges)

cube.RotateFaceToTop(cube.BottomFace)

# Previous TopFace is now BottomFace, and vice versa

FOR EACH edge IN edgePieces

IF (edge is not correctly on opposite face to current bottom face) THEN

PutEdgeIntoEmptySpaceOnTopFace(edge)

END IF

NEXT edge

cube.RotateFaceToTop(BottomFace)

END SUBROUTINE

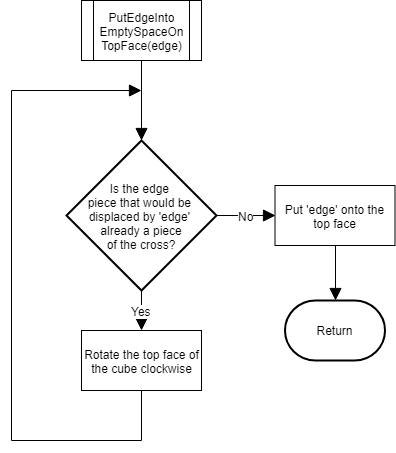


**Extract**() is a function I will define, which separates out the elements of an array with a particular type, in this case it will separate all the **Edge** blocks from within an array of **Blocks**. This (above) is a simple algorithm which iterates through all the edge pieces which should be on the top face, and then if they are not on the opposite face to the one they should be on (i.e. if the top face is white, then this will be true if a white edge piece is not on the yellow face with its yellow Face upwards) it puts them on the current top face (which will be the bottom face overall). The result of this algorithm is shown on the right, with the desired TopFace as white.

This function (left) takes in an edge piece as an argument and the puts it into the current top layer of the cube in an ‘empty’ space, i.e. a space that does not already contain part of the cross. Before moving the piece into the top face, it checks that the piece that would be displaced by the move is not part of the cross, if it is, it rotates the top face 90° and checks again until it finds a valid space.

In order to put an edge piece from its current location to the top layer I have come up with 4 methods dependent on where the edge piece is to begin with. These are detailed on the next page.

To work out which edge piece will be displaced I will work out which face will be rotated (using the same function as the methods that do the rotation), and then check which of the top edge pieces is on that face. The face that will be rotated depends on the position vector of the cubie.



|  |  |
| --- | --- |
| Starting State | Method |
| Edge is in the middle layer | Rotate the face that the cube is on 90° clockwise if the edge is the left hand side of the face being rotated, or 90° anticlockwise if the edge is on the right hand side. |
| Edge is in the top layer but rotated incorrectly (the desired face is forwards not upwards) | Rotate the face that the cube is on 90° clockwise (to put it in the middle layer) then rotate the top face 90° anticlockwise (to put the space in the correct place) and then perform the middle layer method. |
| Edge is in the bottom layer facing downwards | Rotate the face that the cube is on 180°. |
| Edge is in the bottom layer facing forwards | Rotate the top face 90° clockwise (to put the space in the correct place), then rotate the face that the cube is on 90° clockwise (to put the edge in the middle layer) and then perform the middle layer method. |

SUBROUTINE MoveCrossToTopFaceWithCorrectEdges()

edgePieces 🡸 cube.BlocksByColour(cube.TopFace).Extract(Edges)

FOR EACH edge IN edgePieces

sideColour 🡸 edge.SideColour(cube.TopFace)

WHILE NOT edge.OnFace(sideColour)

cube.RotateFace(cube.BottomFace, ANTICLOCKWISE)

END WHILE

cube.RotateFace(sideColour, HALF\_TURN)

NEXT edge

END SUBROUTINE

SideColour(faceColour) is a function that returns the colour of an edge piece that is not the colour passed as an argument, e.g. if you call edge.SideColour(White) for the white-red piece, it would return red.

This algorithm (above) rotates the bottom face of the cube until the edge piece is on the correct face (e.g. if the top face is white, the white-red edge piece needs to be on the red face) and then rotates that face 180° to put the cubie into the correct position in the top layer.

The end result after this algorithm is shown on the right:

SUBROUTINE DoTopCorners()

cornerPieces 🡸 cube.BlocksByColour(cube.TopFace).Extract(Corners)

FOR EACH corner IN cornerPieces

IF (NOT corner.Correct()) THEN

PutCornerIntoCorrectPlaceOnTopFace(corner)

END IF

NEXT corner

END SUBROUTINE

In order to put the corner into its correct place on the top face I have come up with 4 algorithms which are dependent on the starting state of the corner, and which I will detail on the next page. By the direction the corner is ‘facing’ I mean the direction that the face that should be on the top face is pointing.

For some of the methods below I use Rubik’s notation, I will use A to represent the anticlockwise face that the corner is on (when viewed top down) and C to represent the clockwise face that the corner is on; the rest of the notation will follow the standard conventions.

|  |  |
| --- | --- |
| Starting State | Method |
| Corner is on the top layer but rotated incorrectly | Rotate the anticlockwise (when viewed from top) face 90° anticlockwise (to put the corner onto the bottom layer), rotate the bottom face 180° (to move the corner piece out of the way), rotate the 1st face back (90° clockwise), then do the correct corner method to the corner’s current state. |
| Corner is on the bottom layer facing downwards | Rotate the bottom face so the corner is underneath where it should be, then {A’ D A C D2 C’} |
| Corner is on the bottom layer facing left (clockwise when viewed top down) | Rotate the bottom face so the corner is underneath where it should be, then {D’ A’ D A} |
| Corner is on the bottom layer facing right (anticlockwise when viewed top down) | Rotate the bottom face so the corner is underneath where it should be, then {D C D’ C’} |

### MiddleLayerSolver

SUBROUTINE DoMiddleRow()

middlePieces 🡸 (edges that should be in the middle row)

FOR EACH edge IN middlePieces

IF (NOT edge.Correct()) THEN

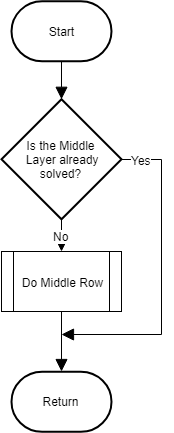
PutEdgeInMiddle(edge)

END IF

NEXT edge

END SUBROUTINE

Class: MiddleLayerSovler



SUBROUTINE PutEdgeInMiddle(edge)

IF edge.Layer MIDDLE Then

cube.RotateFaceToFront(clockwiseFace)

Algorithms.MiddleRightAlgorithm()

PutEdgeFromBottomToMiddle(edge)

ELSEIF edge.Layer = BOTTOM Then

PutEdgeFromBottomToMiddle(edge)

ELSE

THROW ERROR("Top Face Is Not Completed")

END IF

END SUBROUTINE

This algorithm (above) iterates through each of the edges that should be in the middle row, and if they are not in the correct position or rotation (using Edge.Correct()) then it puts them into the middle row using the algorithm below. The algorithm below ensures the edge piece is in the bottom layer before using one of the algorithms in Appendix C to put it into the middle row. If the cubie is already in the bottom row it does nothing, if it is in the middle row it uses AppendixC:Algorithm1 to put another cubie into that place to displace it into the bottom layer. As before, clockwiseFace is the face that the cube is on that is clockwise round the cube when viewed from the top.

SUBROUTINE PutEdgeFromBottomToMiddle(edge)

WHILE NOT edge.Correct()

IF edge.CorrectForFace(anticlockwiseFace) THEN

cube.RotateFaceToFront(anticlockwiseFace)

Algorithms.MiddleRightAlgorithm()

ELSEIF edge.CorrectForFace(clockwiseFace) THEN

cube.RotateFaceToFront(clockwiseFace)

Algorithms.MiddleLeftAlgorithm()

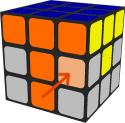
ELSE

cube.RotateFace(cube.BottomFace, Direction.ANTICLOCKWISE)

END IF

END WHILE

END SUBROUTINE

**CorrectForFace**() is a method which checks is a cubie is on a particular face and is rotated correctly for that face only, so it can be used to check whether the orange face of an orange-yellow cubie is on the orange face of the cube (see right). If it is then either AppendixC:Algorithm1 or AppendixC:Algorithm2 can be used to put it into the middle layer of the cube. In the algorithm above, clockwiseFace and anticlockwiseFace are the faces that match the colours on the edge piece, and are clockwise or anticlockwise (viewed from the **bottom** of the cube) in relation to where the edge piece should be. For example, for the orange-yellow cubie in the picture above, the clockwiseFace (right) would be yellow and the anticlockwiseFace (left) would be orange. I am using the bottom of the cube as a reference point in this algorithm as the cube is orientated to face the user and so clockwise will correspond to right and anticlockwise will correspond to left.

The algorithm checks which of the 2 faces the cubie matches up with, if it doesn’t match up then the bottom face of the cube is rotated until it does match up. Once it matches up, the cube is orientated so that the matched face is towards he user, and the correct algorithm (either AppendixC:1 or AppendixC:2) is used, depending on whether the cubie needs inserted into the right or the left. In the picture above it would be inserted into the right.

To get the clockwise and anticlockwise faces I will use the following algorithm:

# 1 to 4 clockwise when viewed from top

FOR face AS MoveFaces 🡸 MoveFaces.LEFT To MoveFaces.FRONT

clockwiseFace 🡸 face Mod 4 + 1 # clockwise when viewed from top

IF (cube.FaceColours(face) = edge.Colours(0) AND

cube.FaceColours(clockwiseFace) = edge.Colours(1)) OR

(cube.FaceColours(face) = edge.Colours(1) AND

cube.FaceColours(clockwiseFace) = edge.Colours(0)) THEN

# ClockwiseFace = cube.FaceColours(face)

# AnticlockwiseFace = cube.FaceColours(clockwiseFace)

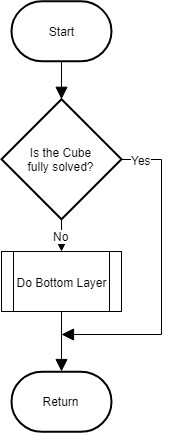
END IF

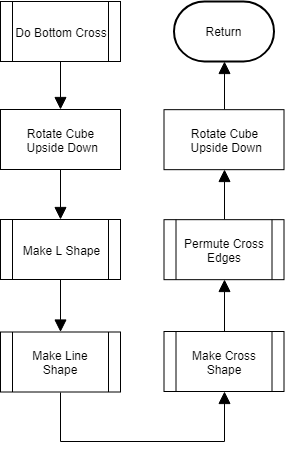
NEXT face

This algorithm iterates through the 4 side faces of the cube going clockwise when viewed from the top, until the colours of the face and the next face (clockwiseFace) match the colours of the cubie. Cube.FaceColours() is a property of the cube which returns the faces of the cube in its current orientation, with indices that match the MoveFaces enum. E.g. MoveFaces.BACK = 2 so cube.FaceColours(2) (or cube.FaceColours(MoveFaces.BACK)) will return the colour of the back face of the cube.

### BottomLayerSolver

Class: BottomLayerSolver





The algorithm to do the bottom cross (left) follows the Beginner’s Method steps to make the cross on the bottom face. At the start of the algorithm the bottom cross will be in 1 of the 4 states shown on the right. AppendixC:Algorithm3 is then used to move through the states until the cross is formed, in the order shown in the algorithm (2 🡪 3 🡪 4 🡪 1). At the start of each of the ‘shape making algorithms’ I will check whether that stage is already completed, and if it is, I will return and execute the next algorithm. That way the program will start from the state the face is in at the start of the algorithm.

It then uses AppendixC:Algorithm4 to permute the edges of the cross so that they are in the correct order and match the order of the side faces of the cube. At the start and end of the algorithm the cube is flipped upside down as we are solving the bottom face, so it is held with that face upwards so we can see what we are doing/what needs to be done.

Below are the 4 algorithms that will be used to transition through the states of the cross, and then permute the edges:

SUBROUTINE MakeL()

IF cube.CorrectTopEdges.Length >= 2 THEN RETURN # The L is done (or the line or cross)

Algorithms.BottomCrossAlgorithm()

END SUBROUTINE

This algorithm simply calls AppendixC:Algorithm3, because the orientation of the cube does not matter for this stage as it is the first one in the sequence. If 2 or more of the edge cubes are in the array cube.CorrectTopEdges() (i.e. there are 2 or more correct edges of the cross) then this stage is already complete and the program should return and run the algorithm for the next stage. It is checking the array of TopEdges() because at this point the cube is upside down, so the edge pieces for the bottom layer cross are currently on the top of the cube.

SUBROUTINE MakeLine()

correctBottomEdges() 🡸 cube.CorrectTopEdges()

IF cube.CorrectTopEdges.Length >= 3 OR

New Vector2x1(correctBottomEdges(0).Position.x, correctBottomEdges(0).Position.z).Dot(

New Vector2x1(correctBottomEdges(1).Position.x, correctBottomEdges(1).Position.z)) <> 0 THEN

RETURN # line (or cross) is done

END IF

WHILE (correctBottomEdges(0).Position.x <> -1 OR correctBottomEdges(1).Position.z <> -1) AND

(correctBottomEdges(1).Position.x <> -1 OR correctBottomEdges(0).Position.z <> -1)

cube.RotateFace(cube.TopFace, ANTICLOCKWISE)

END WHILE

Algorithms.BottomCrossAlgorithm()

END SUBROUTINE

This algorithm again starts with a check to see if the stage has already been completed. The if statement returns true if the number of correct top edges is >= 3 (i.e. the line (plus an extra edge) or cross is already completed), or if the dot product doesn’t equal 0. The dot product is between 2 vectors created from the x and z components of the positions of the 2 correct edges on the top face (if there are 2). The use of just the x and z components, reduces the vectors to 2 dimensional, and effectively gives the positions of the cubie on the top face. If the dot product of these 2 vectors equals 0 then they are perpendicular, if it doesn’t equal 0 then they are parallel i.e. the 2 cubes are in a line (the desired result at the end of this algorithm) and so the algorithm returns to the previous subroutine.

In order to convert the L-shape into a line it must be in the top left corner:   
This means that one of the correct edges must have an x-coordinate of -1,

and the other must have a z-coordinate of -1. The while loop iterates if this is not true, and rotates the top face 90° before checking again. This ensures the top face is in the correct orientation before using AppendixC:3 to convert the L-shape into the line.

SUBROUTINE MakeCross()

IF cube.CorrectTopEdges.Length = 4 Then RETURN # Cross is done

WHILE IsNotLine(cube.CorrectTopEdges())

cube.RotateFace(cube.TopFace, ANTICLOCKWISE)

END WHILE

Algorithms.BottomCrossAlgorithm()

END SUBROUTINE

FUNCTION IsNotLine(ByVal correctCrossEdges()) AS Boolean

SELECT CASE correctCrossEdges.Length

CASE 2

RETURN (correctCrossEdges(0).Position.x <> -1 Or correctCrossEdges(1).Position.x <> 1) AND

(correctCrossEdges(1).Position.x <> -1 Or correctCrossEdges(0).Position.x <> 1)

CASE 3

RETURN (correctCrossEdges(0).Position.x <> -1 Or correctCrossEdges(1).Position.x <> 1) AND

(correctCrossEdges(0).Position.x <> -1 Or correctCrossEdges(2).Position.x <> 1) AND

(correctCrossEdges(1).Position.x <> -1 Or correctCrossEdges(0).Position.x <> 1) AND

(correctCrossEdges(1).Position.x <> -1 Or correctCrossEdges(2).Position.x <> 1) AND

(correctCrossEdges(2).Position.x <> -1 Or correctCrossEdges(0).Position.x <> 1) AND

(correctCrossEdges(2).Position.x <> -1 Or correctCrossEdges(1).Position.x <> 1)

END SELECT

END FUNCTION

This subroutine (top) first checks if the cube already has a cross (4 correct edges) if so then it returns. If not it then uses a while loop to ensure the line is horizontally across the cube (by rotating the top face 90° if it is not) and then uses AppendixC:3 to make the cross.

In order to check if the top face is in the correct orientation for the algorithm I will use the function above called **IsNotLine**(), which returns true if the face is not in the correct orientation. It has 2 different formulae, 1 for if there are 2 correct edges (a line) and one for if there are 3 edges (a T-shape).

The method for 2 returns true if they are not both in one of the 2 correct positions (x = 1 or x = -1).

The method for 3 returns true if any 2 of the edges are not in one of the correct positions (x = 1 or x = -1).

The subroutine on the next page (**PermuteCrossEdges** ) first gets the colours of the edges of the cross clockwise using Edge.SideColour(). Then uses the if-statement labelled **[1]** to check if 2 opposite blocks have the same colour, if they do then it uses the for-loop labelled **[2]** to check 2 adjacent colours to see if they match the faceColour and the next faceColour, if they do then 2 opposite colour and 2 adjacent colours are correct and so all 4 colour are correct so the subroutine returns. If the end of the for-loop is reached without returning, then only 2 of the cross colours are correct, and the following algorithm is used to permute the edges correctly:

AppendixC:4, Rotate TopFace Anticlockwise 90°, AppendixC:4

If 2 opposite edges are not correct then 2 adjacent edges are correct, and the for-loop marked **[3]** is used to get the anticlockwise of the 2 faces, and then this face is rotated to the back, using the formula:

Direction = indexOfAnticlockwiseEdge – 1, {Left(0) 🡪 Clockwise(-1), Back(1) 🡪 NoChange(0), Right(2) 🡪 Anticlockwise(1) and Front(3) 🡪 HalfTurn(2)}

It then uses AppendixC:4 to correct the edges of the cross.

SUBROUTINE PermuteCrossEdges()

FOR edge = 0 TO 3

crossColours(edge) 🡸 cube.TopEdgesClockwise(edge).SideColour(cube.TopFace)

NEXT

IF crossColours(0) = Opposite(crossColours(2)) Or crossColours(1) = Opposite(crossColours(3)) THEN

FOR faceColour = 1 TO 4

IF crossColours(0) = cube.FaceColours(faceColour) AND

crossColours(1) = cube.FaceColours(faceColour MOD 4 + 1) THEN RETURN

NEXT

Algorithms.BottomEdgeAlgorithmRight()

cube.RotateFace(cube.TopFace, Direction.ANTICLOCKWISE)

Algorithms.BottomEdgeAlgorithmRight()

RETURN

END IF

FOR edge = 0 TO 3

FOR faceColour = 1 TO 4

IF crossColours(edge) = cube.FaceColours(faceColour) AND

crossColours((edge + 1) MOD 4) = cube.FaceColours(faceColour MOD 4 + 1) THEN

indexOfAnticlockwiseEdge 🡸 edge

END IF

NEXT

NEXT

cube.RotateFace(cube.TopFace, indexOfAnticlockwiseEdge - 1)

Algorithms.BottomEdgeAlgorithmRight()

END SUBROUTINE



This algorithm (left) finishes the solving of the cube by positioning the 4 corner pieces correctly and then orientating then so that the cube is complete.

#### Position Bottom Corners

To position the corners it will iterate through them until it finds one corner that it in the correct position. It will then check the next corner clockwise, if there are 2 correct corners, then all 4 must be correct (due to the nature of the Rubik's cube) and so the subroutine will return.

If there are no correct corners then AppendixC:Algorithm6 is used in the current orientation, which will put one of the corners into the correct position. The algorithm then recursively calls itself to solve the remaining 3 corners.

If there is one correct corner then permuteCorners will be called, which will rotate the top face so that the correct corner is in the correct place and then apply the right algorithm (either AppendixC:5 or 6) depending on the position of the other corner pieces (see next page).

SUBROUTINE PermuteCorners(nextClockwiseCorner, cornerPositionInde)

# anticlockwise face is the anticlockwise face that the corner is on when viewed from the top

IF nextClockwiseCorner.HasColour(anticlockwiseFace) Then

SELECT CASE cornerPositionIndex

CASE 1 : cube.RotateFace(cube.TopFace, ANTICLOCKWISE)

CASE 2 : cube.RotateFace(cube.TopFace, HALF\_TURN)

CASE 3 : cube.RotateFace(cube.TopFace, CLOCKWISE)

END SELECT

Algorithms.BottomAnticlockwiseCornerAlgorithm()

ELSE

SELECT CASE cornerPositionIndex

CASE 0 : cube.RotateFace(cube.TopFace, ANTICLOCKWISE)

CASE 1 : cube.RotateFace(cube.TopFace, HALF\_TURN)

CASE 2 : cube.RotateFace(cube.TopFace, CLOCKWISE)

END SELECT

Algorithms.BottomClockwiseCornerAlgorithm()

END IF

END SUBROUTINE

If the next clockwise corner after the correct corner contains the colour of the anticlockwise face (I.e. it should be on the anticlockwise face), then the corners need shifted in an anticlockwise direction, so AppendixC:Algorithm5 is used, else they need shifted clockwise and so AppendixC:Algorithm6 is used. Before the algorithm is used the position of the correct corner is used to rotate the top face so that the correct corner is either in the front left position (0) if it is going anticlockwise, or the front right position (3) if it is going clockwise.

OrientateBottomCorners

**OrientateBottomCorners** will be another recursive subroutine, because the process to solve the corners is iterative. The method used depends on the number of corners that are incorrect. I have 3 methods for each of 2, 3 and 4 corners (1 can never be wrong on its own), and these methods then recursively call **OrientateBottomCorners** to either solve the remaining corners, or to finish if all 4 are solved. I will not show the **OrientateBottomCorners** function here as it will just be a simple select case on the number of incorrect top corners, however I will design the 3 algorithms for the varying methods.

Please note, throughout these algorithms there will be various occasions where the cube will have to be rotated upside down. I have excluded these from my algorithms as it does not add to the algorithms, and is just so that methods which claim to act on the bottom face (e.g. **OrientateBottomCorners**) actually do act on the bottom face of the cube, despite the user only seeing the cube in its upside down state. All of this rotating upside down will be cancelled out when the instructions are optimised, and will only be for the sake of consistency with function names.

SUBROUTINE Orientate2Corners()

correctCorners() 🡸 cube.CorrectTopCorners()

IF CornersAreOpposite(correctCorners(0), correctCorners(1)) Then OrientateOppositeCorners()

incorrectCorners() 🡸 cube.IncorrectTopCorners()

facesCommonToBoth() 🡸 GetFacesContainingBothCorners(incorrectCorners(0), incorrectCorners(1))

faceBothRotatedTowards 🡸 GetFaceBothRotatedTowards(incorrectCorners(0), incorrectCorners(1))

cube.RotateFaceToTop(faceBothRotatedTowards)

IF faceBothRotatedTowards = facesCommonToBoth(0) Then

cube.RotateFaceToLeft(facesCommonToBoth(1))

ELSE IF faceBothRotatedTowards = facesCommonToBoth(1) Then

cube.RotateFaceToLeft(facesCommonToBoth(0))

END IF

Algorithms.BottomEdgeAlgorithmRight()

Algorithms.BottomEdgeAlgorithmLeft()

END SUBROUTINE

The subroutine Orientate2Corners (above) is used to orientate the final 2 wrong corners on the face, they can either be adjacent, in which case it will follow the method detailed in that algorithm, or diagonal, in which case it will follow the method in the box below. The use of **BottomEdgeAlgorithmRight** followed by **BottomEdgeAlgorithmLeft** is equivalent to AppendixC: Algorithm7, and is the main algorithm used to orientate a pair of corners in the last layer.

For 2 adjacent corners it rotates the cube so that 2 of the same colour stickers are upwards (using faceBothRotateTowards), then orientates the cube so the 2 corners are on the left and performs AppendixC:7 to rotate them into the correct orientation.

In the algorithm below, IIF is a ternary comparison operator, so if the condition in the first argument is true then it evaluates and returns the 2nd argument, if it is false it returns the 3rd. in this case it is used to get the corner which is on the top face (once the cube has been rotated about the Z-axis. The algorithm rotates the cube until the cubie that is on the top face is orientated such that the colour facing upwards is the colour that should be on the bottom face of the cube (see image on right – bottom face is yellow). It then carries out to AppendixC:7 to create 2 adjacent correct corners, and then returns to Orientate2Corners to orientate those 2 correctly.

SUBROUTINE OrientateOppositeCorners()

bottomFaceColour 🡸 cube.TopFace

incorrectCorners() 🡸 cube.IncorrectTopCorners()

cube.Rotate(Direction.ANTICLOCKWISE, Axis.Z)

DO

cube.Rotate(Direction.ANTICLOCKWISE, Axis.X)

cornerOnTopFace 🡸 IIF(incorrectCorners(0).OnFace(cube.TopFace),

incorrectCorners(0), incorrectCorners(1))

LOOP WHILE NOT cornerOnTopFace.CorrectRotation(bottomFaceColour, cube.TopFace)

Algorithms.BottomEdgeAlgorithmRight()

Algorithms.BottomEdgeAlgorithmLeft()

cube.Rotate(Direction.CLOCKWISE, Axis.Z)

END SUBROUTINE

The method for orientating 3 corners, is to first rotate the cube onto its side then rotate it around the x-axis until neither of the top 2 corners on the left side are solved (done with the do loop in the algorithm). ThenAppendixC:7 is carried out and the number of correct corners is checked. If it isn’t 4 then the subroutine ‘recursively’ calls **OrientateBottomCorners** to orientate the remaining corners – this is not recursion in the sense of a function calling itself, but in the sense of a sub-function calling its own calling function.

SUBROUTINE Orientate3Corners()

cube.Rotate(Direction.ANTICLOCKWISE, Axis.Z)

DO

cube.Rotate(Direction.CLOCKWISE, Axis.X)

count = 0

FOR EACH corner IN cube.Corners

IF corner.Position = Vector3(-1, 1, 1) OR corner.Position = Vector3(-1, 1, -1) Then

leftTopEdgeCorners(count) 🡸 corner

count 🡸 count + 1

END IF

NEXT

LOOP WHILE (leftTopEdgeCorners(0).Correct() Or leftTopEdgeCorners(1).Correct())

Algorithms.BottomEdgeAlgorithmRight()

Algorithms.BottomEdgeAlgorithmLeft()

cube.Rotate(Direction.CLOCKWISE, Axis.Z)

IF cube.CorrectTopCorners.Length <> 4 Then

OrientateBottomCorners()

END IF

END SUBROUTINE

The method for 4 corners involves doing the adjacent part of the 2 corner method, and then calling OrientateBottomCorners to solve the remaining 2. See Orientate2Corners for the explanation of the method.

SUBROUTINE Orientate4Corners()

Dim incorrectCorners() As Corner = cube.IncorrectTopCorners()

Dim facesCommonToBoth() As FaceColour

Dim faceBothRotatedTowards As FaceColour

FOR i 🡸 0 TO 3

facesCommonToBoth 🡸 GetFacesContainingBothCorners(incorrectCorners(i),

incorrectCorners(i + 1))

faceBothRotatedTowards 🡸 GetFaceBothRotatedTowards(incorrectCorners(i),

incorrectCorners(i + 1))

IF faceBothRotatedTowards <> FaceColour.None THEN EXIT FOR

NEXT

cube.RotateFaceToTop(faceBothRotatedTowards)

IF faceBothRotatedTowards = facesCommonToBoth(0) THEN

cube.RotateFaceToLeft(facesCommonToBoth(1))

ELSE IF faceBothRotatedTowards = facesCommonToBoth(1) THEN

cube.RotateFaceToLeft(facesCommonToBoth(0))

END IF

Algorithms.BottomEdgeAlgorithmRight()

Algorithms.BottomEdgeAlgorithmLeft()

ReturnCubeToOriginalOrientation()

OrientateBottomCorners()

END SUBROUTINE

### Matrix Multiplication

Class: Matrix

Because my matrices will be made from vector objects, and all my matrices will be square, this will allow me to easily perform matrix multiplication using the dot products of the vectors . The matrix that is the product of the 2 matrices will be made with rows as shown below (where A is matrix1, B is matrix2, AR1 is matrix1, row1, and AC1 is matrix1, column1, and so on). The example is for 2 3x3 matrices, however the patterns are quite clear and this can easily be scaled up or down square matrices of any dimension.

On the right I have also shown the representation of the product of a matrix A and a vector V.

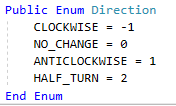
|  |
| --- |
| AR1●V |
| AR2●V |
| AR3●V |

|  |  |  |
| --- | --- | --- |
| AR1●BC1 | AR1●BC2 | AR1●BC3 |
| AR2●BC1 | AR2●BC2 | AR2●BC3 |
| AR3●BC1 | AR3●BC2 | AR3●BC3 |

### Rotating the Entire Cube

Class: Cube

In order to rotate the entire cube I will use rotation matrices. The following matrices (right) are used for rotations around the X, Y and Z axes respectively. I will use the formula:

Angle = Pi/2 \* direction

Where direction is one of the elements in the Direction enum (right):

I will then pass this angle into the correct matrix based on the axis and then multiply the position vector of every cubie by the rotation matrix to get its new position. I will then add the instruction to cube.instructions using the **AddOrientationChange** method. After that I will update the orientation of the cube using the following algorithm:

SUBROUTINE UpdateOrientation(direction, axis)

faces() 🡸 FacesAroundAxisClockwise(axis)

IF axis <> Axis.Y Then TopFace = faces((IndexOf(faces, TopFace) + 4 + direction) MOD 4)

IF axis <> Axis.Z Then FrontFace = faces((IndexOf(faces, FrontFace) + 4 + direction) MOD 4)

END SUBROUTINE

This algorithm (above) calculates the new top and front faces of the cube, by first getting the 4 faces that are around the axis, e.g. for the Z-axis (front to back) it would return {Top, Right, Bottom, Left}. It then gets the index of the current face. E.g. for the top face and the Z-axis it would be 0, and then adds 4 and the direction to it and gets the sum Mod 4. This and the plus 4 ensures that the new index is between 0 and 3, and adding the direction gives the new index, which is then passed back into the array to get the colour of the new face.

Because the entire cube is rotated I will not need to change the rotations of any of the cubies, as they are all relative to the face colours.

### Rotating a Specific Colour Face

Class: Cube

mathematicalDirection 🡸 direction

SELECT CASE face

CASE TOP

axis 🡸 Axis.Y

CASE LEFT

axis 🡸 Axis.X

If direction <> HALF\_TURN THEN

mathematicalDirection 🡸 -mathematicalDirection

END IF

CASE BACK

axis 🡸 Axis.Z

If direction <> HALF\_TURN THEN

mathematicalDirection 🡸 -mathematicalDirection

END IF

CASE RIGHT

axis 🡸 Axis.X

CASE FRONT

axis 🡸 Axis.Z

CASE BOTTOM

axis 🡸 Axis.Y

If direction <> HALF\_TURN THEN

mathematicalDirection 🡸 -mathematicalDirection

END IF

END SELECT

angle 🡸 (PI / 2.0) \* mathematicalDirection

rotationMatrix 🡸 GetRotationMatrix(axis, angle)

orientatedFaces 🡸 GetFaceColoursAsIfRotatedFaceIsTop(face)

FOR EACH cubie IN CornersAndEdgesAndMiddles

IF NOT cubie.OnFace(face) THEN NEXT cubie

cubie.Position 🡸 rotationMatrix \* cubie.Position

cubie.Rotation 🡸 GetNewRotation(direction, cubie.Rotation, orientatedFaces)

IF cubie.GetType() = GetType(Corner) THEN

cubie.SecondaryRotation 🡸 GetNewRotation(direction, cubie.SecondaryRotation,

orientatedFaces)

END IF

NEXT cubie

Because, to get the rotation matrix I need the angle and axis to rotate about, this algorithm first calculates the angle and axis required. It uses a select case on the face to get the axis, and then inverts the direction if it is the left, back or bottom face. It inverts it because these 3 faces are on the negative side of the axis, and so they need to be mathematically rotated the opposite direction to the direction of the instruction.

The algorithm then generates an array of the cube’s face colours, but in an orientation as if the rotated face is on the top. This is then used to calculate the new rotations of the cubies. Once this has been calculated, the subroutine loops through all of the cubies and if the cube is on the face being rotated (cubie.**OnFace**(face) = true) then its rotation is multiplied by the rotation matrix, and it’s new rotations (and secondary rotation is necessary) are calculated with the function **GetNewRotation().**

In order to get the new rotation from the orientated array of face colours, I will use the following formulas (where X = the index of the current rotation) to get the new index for the rotation colour.

|  |  |
| --- | --- |
| Direction | Formula |
| NO\_CHANGE | X |
| CLOCKWISE | X MOD 4 + 1 |
| ANTICLOCKWISE | (X + 2) MOD 4 + 1 |
| HALF\_TURN | (X + 1) MOD 4 = 1 |

And to get the orientated Faces array, I will use the **GetFaceColoursFromOrientation(**orientation**)** method with an orientation of {Top = face, Front = one of the faces listed below}.

|  |  |
| --- | --- |
| Condition | Front Face to Get Above Array |
| Face being rotated = FrontFace | BottomFace |
| face being rotated = BackFace | TopFace |
| Else | FrontFace |

### Optimising InstructionList

Class: InstructionList

Dim anySimplificationsMade As Boolean = False

DO

anySimplificationsMade 🡸 False

ptr <= 0

WHILE ptr < Me.Count - 1

currentInstruction 🡸 Me.ElementAt(ptr)

nextInstruction 🡸 Me.ElementAt(ptr + 1)

IF currentInstruction.Type <> nextInstruction.Type THEN

ptr 🡸 ptr + 1

CONTINUE WHILE

END IF

IF currentInstruction.Move <> nextInstruction.Move THEN

ptr 🡸 ptr + 1

CONTINUE WHILE

END IF

directionSum 🡸 Math.Abs((currentInstruction.Direction + nextInstruction.Direction) Mod 4)

IF directionSum = 3 THEN directionSum <= -1

currentInstruction.Direction 🡸 directionSum

IF currentInstruction.Direction = Direction.NO\_CHANGE THEN

Me.RemoveAt(ptr)

ptr 🡸 ptr - 1

END IF

Me.RemoveAt(ptr + 1)

ptr 🡸 ptr + 1

anySimplificationsMade 🡸 True

END WHILE

LOOP WHILE anySimplificationsMade

This algorithm is an O(n2) algorithm for time efficiency, and is similar to a bubble sort in its operation, however this should not be an issue as there will not be a large number of instructions to optimise, and so an n2 running time will not be too long. The algorithm iterates through the instruction list until no simplifications are made (i.e. there are no more consecutive versions of the same instruction. In each iteration it uses a pointer variable (ptr) to keep track of the current location and then compares the instruction at that index with the next one. If the instructions are the same type and move (i.e. same axis/face) then their directions are added together and they are combined into one instruction at the ptr index, then the next instruction is removed. If the resultant direction is 0 (no change) then the instruction at the pointer is also removed.

### GLSL version control

Class: ShaderProgram

In order to control which version of GLSL is run, I will use the following code. This is code rather than pseudocode as it is library-specific code and would not make sense as pseudocode. To get this code I wrote a prototype program and tested it on 2 different computers running different versions of OpenGL.

Dim versionNo As Integer = CInt(CStr(GL.GetString(StringName.Version)(0) +

GL.GetString(StringName.Version)(2)))

Dim versionString As String = "330"

If versionNo >= 33 Then

versionString = "330"

ElseIf versionNo >= 21 Then

versionString = "120"

Else

MsgBox("3D output is not supported by your graphics library")

Return

End If

This code uses GL.GetString() to get the openGL version. The version is returned as a string e.g. (3.3) so to remove the decimal point I take characters 1 and 3 (indexed 0 and 2), concatenate them and cast them to an integer. I then use an if statement to work out if the version is after 3.3 (GLSL version 330) or between 2.1 and 3.3 (GLSL version 120), if the version is before 2.1 (pre-2006) then it does not support shaders and is not compatible with my program. In this case a message is given to the user and the function returns. If there is a compatible GLSL version, then I set the versionString corresponding to the version, and later prepend this to my shaders with code such as: “#version” + versionString + shader. This works with string concatenation, because shaders are stored and used as strings. The shaders can then use pre-processor directives to check their version and compile the correct version of the code.

### Getting the Initial Positions for the Output

Class: 3DOutput

Before I can render the scrambled cube at the start of the output, I need to work out where to translate each of my rendered cubie objects to. In order to do this I will use the position vectors of the cubies in the scrambled cube, and this formula (see right):

Position 🡸 Vector3 With {

X-component = cubie.Position.x \* (1 + pad) - 0.5,

Y-component = cubie.Position.y \* (1 + pad) - 0.5,

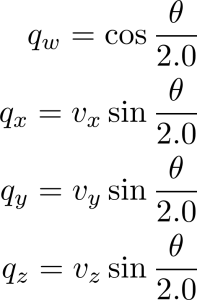
Z-component = cubie .Position.z \* (1 + pad) - 0.5

}

This will allow each cubie to be separated by the pad amount (a constant), and the “- 0.5” accounts for the width of the cubie and the fact that my cubie vertices will have 1 corner at the origin, rather than be centred at the origin.

I will then translate the cubies to these positions each frame after creating then and before applying any of the rotations.

### Getting the Initial Rotations for the Output

Class: 3DOutput

In order to store the initial rotations of each cubie (that is the rotation it is in before the solving starts) I will use quaternions. A quaternion is a 4-D complex number {e.g. 2 + 3i + 6j + k}, which are used to represent transformations in 3D. I will use quaternions because they are quicker and more compact then using matrices and they will be applied to the model every frame, so efficiency factors heavily. Furthermore, the initial rotations will be quite simple, and quaternions can easily be generated from an axis (vector) and an angle, using the formulae on the right (where q is a quaternion and v is the axis vector). I will use this property and the cubie.rotation properties to get the initial quaternions for my cubies. As the rotations will only ever be multiples of 90° around 3 axes (due to the nature of a Rubik’s cube), I can represent them by 2 consecutive rotations about 2 axes. The overall rotation will then be the product of these 2 rotations. To get the first and second rotations I will use the functions **GetRotationFirstAxis** and **GetRotationSecondAxis** respectively. The first rotation will orientate the cube so that its primary face is correct, the second rotation will be around the normal of the primary face, and will therefore set the correct rotation for the other 4 sides of the cubie.

For **GetRotationFirstAxis** I will use the cubie’s rotation to set an amount of rotation around one of the axes. For example if the WRB corner cubie has a rotation of red, then its white face is pointing forwards (towards the user), and so it would need a first rotation of 90° (pi/2 radians) about the X axis.

Because my rotations will only be about one of the 3 major axes, the length of the axis vector will not matter, I can therefore use the rotationVector to not only store the axis of rotation, but also the amount of rotation. For example, for a 180° (pi radians) rotation around the Y-axis, I could store this as the vector {0, 2, 0} which represents a (2\*half-pi) rotation around the Y-axis. I will then use this algorithm to convert this rotation vector to a quaternion:

rotationFirstAxis = New Quaternion With {

.X 🡸 rotationVector.X \* Sin(Abs(rotationVector.X) \* PiOver2 / 2),

.Y 🡸 rotationVector.Y \* Sin(Abs(rotationVector.Y) \* PiOver2 / 2),

.Z 🡸 rotationVector.Z \* Sin(Abs(rotationVector.Z) \* PiOver2 / 2)

}

vecSum 🡸 Abs(rotationVector.X) + Abs(rotationVector.Y) + Abs(rotationVector.Z)

rotationFirstAxis.W 🡸 Cos(vecSum \* PiOver2 / 2)

Below are the rotation vectors I will use for each of the different combinations of primary face colour and rotation.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Primary Face Colour | Rotation | X | Y | Z |  | Primary Face Colour | Rotation | X | Y | Z |
| W | Y | 2 | 0 | 0 |  | R | W | -1 | 0 | 0 |
| W | R | 1 | 0 | 0 |  | R | Y | 1 | 0 | 0 |
| W | O | -1 | 0 | 0 |  | R | O | 2 | 0 | 0 |
| W | G | 0 | 0 | 1 |  | R | G | 0 | -1 | 0 |
| W | B | 0 | 0 | -1 |  | R | B | 0 | 1 | 0 |
| Y | W | 2 | 0 | 0 |  | O | W | 1 | 0 | 0 |
| Y | R | -1 | 0 | 0 |  | O | Y | -1 | 0 | 0 |
| Y | O | 1 | 0 | 0 |  | O | R | 2 | 0 | 0 |
| Y | G | 0 | 0 | -1 |  | O | G | 0 | 1 | 0 |
| Y | B | 0 | 0 | 1 |  | O | B | 0 | -1 | 0 |

In order to get the second rotation, I will need some way of referencing the orientation of the cubie after its first rotation. To do this I will use the normal vectors of the white and red faces of the cubie. I will multiply the white (0,1,0) and red (0, 0, 1) normal by a rotation matrix created from the first quaternion. This will give me the normal vectors after the first rotation. I will then compare them to what the white and red normal should be (which I will get with the function **GetNormalsFromCubie**) and then whichever one matches will be my axis. E.g. if the currentWhiteNormal matches the desiredWhiteNormal, then that normal will be the axis for the second rotation. I will then use the other normal and the function **GetAngleForInitialRotation** to get the angle for the second rotation. This will give me a separate axis and angle, so I will just use the AxisAngle🡪Quaternion formulae I discussed earlier to get the second rotation quaternion. Below are the algorithms for the functions named above.

FUNCTION GetAngleForInitialRotation(axis, currentPosition, desiredPosition) As Single

Dim angle As Single

IF currentPosition = desiredPosition THEN

angle = 0

ELSE IF Abs(Vector3.Dot(currentPosition, desiredPosition)) = 1 THEN

# vectors are parallel

angle = Pi

Else

# vectors are perpendicular

# collapse the vectors onto 1 2D plane, perpendicular to axis of rotation

Dim current2d, correct2d As Vector2

IF axis.X <> 0 THEN

current2d 🡸 New Vector2(-currentPosition.Z, currentPosition.Y)

correct2d 🡸 New Vector2(-desiredPosition.Z, desiredPosition.Y)

ELSE IF axis.Y <> 0 THEN

current2d 🡸 New Vector2(currentPosition.X, -currentPosition.Z)

correct2d 🡸 New Vector2(desiredPosition.X, -desiredPosition.Z)

ELSE IF axis.Z <> 0 THEN

current2d 🡸 New Vector2(currentPosition.X, currentPosition.Y)

correct2d 🡸 New Vector2(desiredPosition.X, desiredPosition.Y)

END IF

IF current2d.Y <> 0 THEN

IF current2d.Y = correct2d.X THEN

angle 🡸 -PiOver2

Else

angle 🡸 PiOver2

END IF

ELSE IF current2d.X <> 0 THEN

IF current2d.X = correct2d.Y THEN

angle 🡸 PiOver2

Else

angle 🡸 -PiOver2

END IF

END IF

END IF

RETURN angle

END Function

SUBROUTINE GetNormalsFromCubie(whiteNormal, redNormal)

whiteSet 🡸 False

redSet 🡸 False

SELECT CASE cubie.PrimaryFace

CASE White, Yellow

SELECT CASE cubie.Rotation

CASE White : whiteNormal = Vector3(0, 1, 0)

CASE Yellow : whiteNormal = Vector3(0, -1, 0)

CASE Red : whiteNormal = Vector3(0, 0, 1)

CASE Orange : whiteNormal = Vector3(0, 0, -1)

CASE Green : whiteNormal = Vector3(-1, 0, 0)

CASE Blue : whiteNormal = Vector3(1, 0, 0)

END SELECT

IF cubie.PrimaryFace = Yellow THEN whiteNormal 🡸 -whiteNormal

whiteSet 🡸 True

CASE Red, Orange

SELECT CASE cubie.Rotation

CASE White : redNormal = Vector3(0, 1, 0)

CASE Yellow : redNormal = Vector3(0, -1, 0)

CASE Red : redNormal = Vector3(0, 0, 1)

CASE Orange : redNormal = Vector3(0, 0, -1)

CASE Green : redNormal = Vector3(-1, 0, 0)

CASE Blue : redNormal = Vector3(1, 0, 0)

END SELECT

IF cubie.PrimaryFace = Orange THEN redNormal 🡸 -redNormal

redSet 🡸 True

END SELECT

SELECT CASE secondaryFace

CASE Red, Orange

SELECT CASE secondaryRotation

CASE White : redNormal = New Vector3(0, 1, 0)

CASE Yellow : redNormal = New Vector3(0, -1, 0)

CASE Red : redNormal = New Vector3(0, 0, 1)

CASE Orange : redNormal = New Vector3(0, 0, -1)

CASE Green : redNormal = New Vector3(-1, 0, 0)

CASE Blue : redNormal = New Vector3(1, 0, 0)

END SELECT

IF secondaryFace = Orange THEN redNormal 🡸 -redNormal

redSet 🡸 True

CASE Green, Blue

Dim greenNormal As Vector3

SELECT CASE secondaryRotation

CASE White : greenNormal = Vector3(0, 1, 0)

CASE Yellow : greenNormal = Vector3(0, -1, 0)

CASE Red : greenNormal = Vector3(0, 0, 1)

CASE Orange : greenNormal = Vector3(0, 0, -1)

CASE Green : greenNormal = Vector3(-1, 0, 0)

CASE Blue : greenNormal = Vector3(1, 0, 0)

END SELECT

IF secondaryFace = Blue THEN greenNormal 🡸 -greenNormal

# use the set vector and green normal to calculate the unset one

# uses the cross product to calculate the perpendicular vector

IF whiteSet THEN

redNormal 🡸 Vector3.Cross(whiteNormal, greenNormal)

ElseIf redSet THEN

whiteNormal 🡸 -Vector3.Cross(redNormal, greenNormal)

END IF

END SELECT

END SUBROUTINE

### Rendering the Cube

Class: 3DOutput

To render the cube I will use the following algorithm which will be called each frame to create the cubie objects and rotate them correctly. DisplayAnglePercent will need to be static variable, as it will need to be referenced and incremented each frame. This subroutine calculates the current percentage of rotation using the algorithm below, and then calls RenderCubie for each index (i) in the array of cubies.

SUBROUTINE RenderCube()

# send view and projection matrices to the shaders

# uses the value of the slider on the form

rotationSpeed 🡸 sldrSpeed.Value / 1000

IF (showingRotation) THEN

# uses delta time to ensure rotation speed is constant

displayAnglePercent 🡸 displayAnglePercent deltaTime \* rotationSpeed

IF displayAnglePercent >= 1 THEN # if rotation is complete (>= 100%)

displayAnglePercent 🡸 1

showingRotation 🡸 False

END IF

ELSE

displayAnglePercent 🡸 0

END IF

FOR i 🡸 0 TO 26

RenderCubie(i, displayAnglePercent)

NEXT

END SUBROUTINE

This subroutine acts on 1 cubie. It first translates it to position 26 (the position of the middle cubie centred around the origin). It then applied the initial rotation for that cubie using initialRotation(index), then translates it back and then uses its initial position vector to position it correctly. It then passes the model matrix at that point to the fragment shader for working out if vertices are within the bounds of the cube. It then uses the cubie’s rotation store to get its current rotation matrix which is also multiplied with the model matrix to form the final matrix which is passed to the shaders. The program will then call GL.DrawElements() to draw the cubie using the matrices passed into the shaders, before moving onto the next cubie.

SUBROUTINE RenderCubie(index, displayAnglePercent)

model 🡸 TranslationMatrixFromVector(initalPositions(26))

model 🡸 model \* RotationMatrixFromQuaternion(initalRotations(index))

model 🡸 model \* TranslationMatrixFromVector(-initalPositions(26))

model 🡸 model \* TranslationMatrixFromVector(initalPositions(index))

(Pass colour matrix to shaders (model matrix before rotations))

model 🡸 model \* cubieRotations(index).Rotations.GetTotalMatrix(displayAnglePercent)

(Pass model matrix to shaders)

(Draw cubie)

END SUBROUTINE

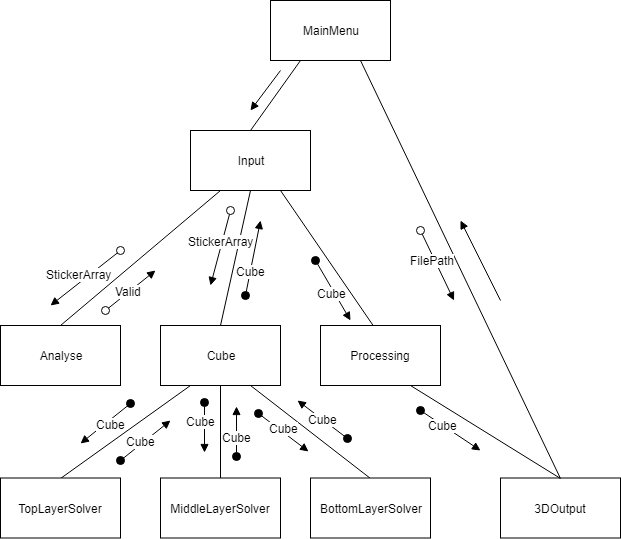
## Structure diagrams

### Module Level Structure Diagram

This diagram is not a conventional structure diagram, but rather is based off of the standards used in structure diagram in order to shows the relations between forms/modules, rather than subroutines (like a standard structure diagram).

The diagram shows how the program control passes between modules/forms. The variables on the arrows between modules are variables that are passed into the classes by their constructors/returned when the class has ‘finished’ its processing. Arrows without variables on them show that the control will pass from one class/form to another without any transfer of data.

One aspect of my program that is not clear from this diagram is that when the input class creates a new cube object (by passing in the array of stickers and getting a cube in return), it will not solve the cube at this point. It will instead pass this cube to the processing form, which will display a message to the user, and then solve the cube at this point. This is signified by the dashed line in the diagram.

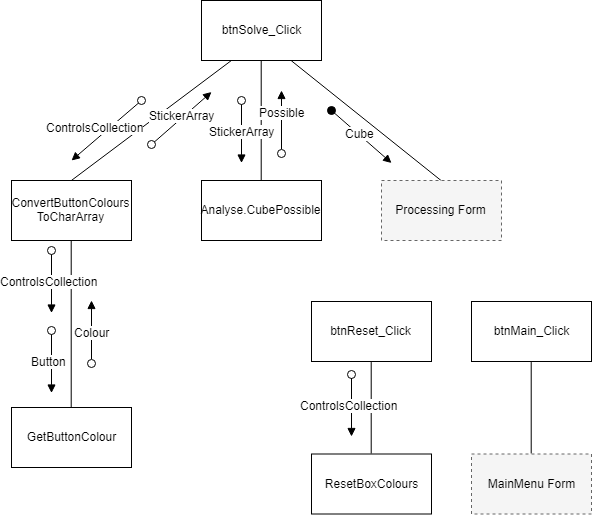


For my form/module-level structure diagrams I will use white boxes with solid edges to represent subroutines and event handlers, and grey boxes with dashed edges to represent opening other forms.

I have not made a structure diagram for my layer solvers or for the output form as I feel that they are represented clearly enough to be coded just by their algorithms, and that their structure diagrams would end up too large, and cluttered to be useful.

### MainMenu

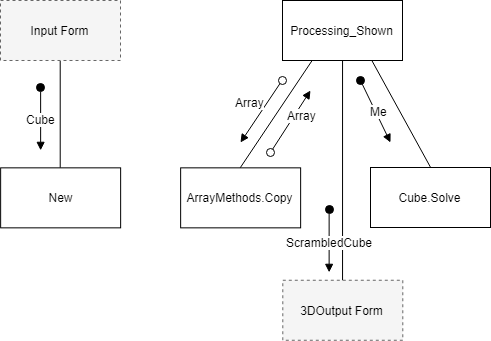
### Input



This is the form that handles the user’s input and creates the cube object to be solved

### Analyse

This is the form that checks if the inputted cube is possible



### Processing

This is the form that calls Cube.Solve() and shows a message to the user whilst it is being solved

# **Technical Solution**

## Programming Techniques Table

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Technique** | | **Group** | | **Example Code Snippet** | **File** |
| Inheritance | | | A | Public Class Corner  Inherits Block  ...  End Class | Corner.vb |
| Composition | | | A | Public MustInherit Class Block  ...  Private \_position As New Vector3x1()  ...  End Class | Block.vb |
| Aggregation | | | A | Public Class TopLayerSolver  Private cube As Cube  Sub New(ByRef cubeToBeSolved As Cube)  cube = cubeToBeSolved  End Sub  ...  End Class | TopLayerSolver.vb |
| Overriding Polymorphism | | | A | Public MustInherit Class Block  ...  Public MustOverride Function HasColour(ByVal colour As FaceColour) As Boolean  ...  End Class  Public Class Corner  Inherits Block  ...  Public Overrides Function HasColour(colour As FaceColour) As Boolean  For col = 0 To 2  If Colours(col) = colour Then Return True  Next  Return False  End Function  ...  End Class | Corner.vb |
| Overloading Polymorphism | | | A | Public Function Copy(Of T)(ByVal array() As T) As T()  Dim copyArray(array.Length - 1) As T  For i = 0 To array.Length - 1  copyArray(i) = array(i)  Next  Return copyArray  End Function  Public Function Copy(Of T)(ByVal queue As Queue(Of T)) As Queue(Of T)  Dim copyQueue As New Queue(Of T)  Dim numberOfElements As Integer = queue.Count  For i = 0 To numberOfElements - 1  copyQueue.Enqueue(queue.ElementAt(i))  Next  Return copyQueue  End Function | Array Methods.vb |
| Binary Serialisation | | | B | <Serializable()> Public Class Cube  ...  Private Sub WriteCubeToFile(ByRef fStream As IO.FileStream)  Dim formatter As New BinaryFormatter()  formatter.Serialize(fStream, Me)  End Sub  End Class | Cube.vb |
| Matrix Operations | | |  | ''' <summary> renders the cubie represented by a given index </summary>  Private Sub RenderCubie(ByVal index As Integer, ByVal displayAnglePercent As Double)  ...  Dim model As Matrix4 = Matrix4.CreateTranslation(initalPositions(26))  model \*= Matrix4.CreateFromQuaternion(initalRotations(index))  model \*= Matrix4.CreateTranslation(-initalPositions(26))  model \*= Matrix4.CreateTranslation(initalPositions(index))    End Sub | 3DOutput.vb |
| Vector Operations | | |  | Public Function Dot(vec As Vector3x1) As Single  Return (Me.x \* vec.x + Me.y \* vec.y + Me.z \* vec.z)  End Function | Matrices.vb |
| Quaternion Arithmetic | | |  | Dim rotationSecondAxis As Quaternion  rotationSecondAxis = Quaternion.Identity  If angle <> 0 Then  rotationSecondAxis.X = Math.Abs(axis.X) \* Math.Sin(angle / 2)  rotationSecondAxis.Y = Math.Abs(axis.Y) \* Math.Sin(angle / 2)  rotationSecondAxis.Z = Math.Abs(axis.Z) \* Math.Sin(angle / 2)  rotationSecondAxis.W = Math.Cos(angle / 2)  Else  rotationSecondAxis = Quaternion.Identity  End If | 3DOutput.vb |
| Shader Based Open GL | | | A | ' Bind the Vertex Array Object  GL.BindVertexArray(shaderProgram.VAO)  ' copy our vertices to a buffer for OpenGL to use  GL.BindBuffer(BufferTarget.ArrayBuffer, shaderProgram.VBO)  GL.BufferData(BufferTarget.ArrayBuffer, CType(Vertex3D.SizeInBytes \* vertices3d.Length,  IntPtr), vertices3d,BufferUsageHint.StaticDraw) | 3DOutput.vb |
| Custom Exception Types | | |  | Public Class StageNotSuccessfulException  Inherits Exception  Public Sub New(ByVal message As String)  MyBase.New(message)  End Sub  Public Sub New()  MyBase.New()  End Sub  End Class | Exceptions.vb |
| Pre-processor Directives | | |  | #if \_\_VERSION\_\_ == 330  ...  #endif  #if \_\_VERSION\_\_ == 120  ...  #endif | vertextShader.txt |
| Generic Functions | | |  | <Extension()>  Public Function Copy(Of T)(ByVal array() As T) As T()  Dim copyArray(array.Length - 1) As T  For i = 0 To array.Length - 1  copyArray(i) = array(i)  Next  Return copyArray  End Function | ArrayMethods.vb |
| Extension Functions to Built-in Datatypes | | |  | <Extension()>  Public Sub Append(Of T)(ByRef array() As T, ByVal item As T)  Try  ReDim Preserve array(array.Length)  array(array.Length - 1) = item  Catch ex As NullReferenceException  array = {item}  End Try  End Sub | ArrayMethods.vb |
| Custom Defined Operators | | |  | Public Shared Operator =(ByVal vec1 As Vector3x1, ByVal vec2 As Vector3x1)  Return vec1.x = vec2.x And vec1.y = vec2.y And vec1.z = vec2.z  End Operator | Matrices.vb |
| Use of lists | | | A | <Serializable()> Public Class InstructionList  Inherits List(Of Instruction)  ...  End Class | InstructionList.vb |
| Recursion | | | A | Private Sub OrientateBottomCorners()  ...  Orientate4Corners()  ...  End Sub  Private Sub Orientate4Corners()  ...  OrientateBottomCorners()  ...  End Sub | BottomLayerSolver.vb |
| Complex User Defined Algorithms: | | | A | (see full code/algorithms section) |  |
| * Top Layer Solver | | | Public Sub Solve()  ...  End Sub | TopLayerSolver.vb |
| * Middle Layer Solver | | | Public Sub Solve()  ...  End Sub | MiddleLayerSolver.vb |
| * Bottom Layer Solver | | | Public Sub Solve()  ...  End Sub | BottomLayerSolver.vb |
| * Rotating the whole cube (when solving) | | | Public Sub Rotate(ByVal direction As Direction, ByVal axis As Axis)  ...  End Sub | Cube.vb |
| * Rotating one face of the cube (when solving) | | | Public Sub RotateFace(ByVal face As FaceColour, ByVal direction As Direction)  ...  End Sub | Cube.vb |
| Simple User Defined Algorithms: | | | B | (see full code/algorithms section) |  |
| * Optimising Instructions | | | Public Sub Optimise()  ...  End Sub | InstructionList.vb |
| * Matrix Multiplication | | | Public Shared Operator \*(ByVal mat1 As Matrix3x3, ByVal mat2 As Matrix3x3) As Matrix3x3  Return New Matrix3x3(New Vector3x1(mat1.Row1.Dot(mat2.Column1),  mat1.Row1.Dot(mat2.Column2),  mat1.Row1.Dot(mat2.Column3)),  New Vector3x1(mat1.Row2.Dot(mat2.Column1),  mat1.Row2.Dot(mat2.Column2),  mat1.Row2.Dot(mat2.Column3)),  New Vector3x1(mat1.Row3.Dot(mat2.Column1),  mat1.Row3.Dot(mat2.Column2),  mat1.Row3.Dot(mat2.Column3)))  End Operator | Matrices.vb |

## Index of Code Files

I have included screenshots of the final windows in the sections containing the code for that form.

[MainMenu.vb 100](#_Toc511390339)

[Input.vb 102](#_Toc511390340)

[Analyse.vb 105](#_Toc511390341)

[CoordinatePairs.vb 111](#_Toc511390342)

[SetCubieProperties.vb 115](#_Toc511390343)

[Processing.vb 123](#_Toc511390344)

[Cube.vb 124](#_Toc511390345)

[Helpers.vb 139](#_Toc511390346)

[Block.vb 144](#_Toc511390347)

[Corner.vb 147](#_Toc511390348)

[Edge.vb 149](#_Toc511390349)

[Middle.vb 151](#_Toc511390350)

[Matrices.vb 152](#_Toc511390351)

[TopLayerSolver.vb 163](#_Toc511390352)

[MiddleLayerSolver.vb 173](#_Toc511390353)

[BottomLayerSolver.vb 176](#_Toc511390354)

[CubeOrientation.vb 186](#_Toc511390355)

[Algorithms.vb 187](#_Toc511390356)

[ArrayMethods.vb 190](#_Toc511390357)

[InstructionList.vb 195](#_Toc511390358)

[Instruction.vb 197](#_Toc511390359)

[FaceInstruction.vb 197](#_Toc511390360)

[CubeInstruction.vb 198](#_Toc511390361)

[3dOutput.vb 199](#_Toc511390362)

[Camera.vb 219](#_Toc511390363)

[Vertex3D.vb 221](#_Toc511390364)

[ShaderProgram.vb 222](#_Toc511390365)

[RotationStore.vb 224](#_Toc511390366)

[OutputBlock.vb 225](#_Toc511390367)

[OutputInstructionList.vb 225](#_Toc511390368)

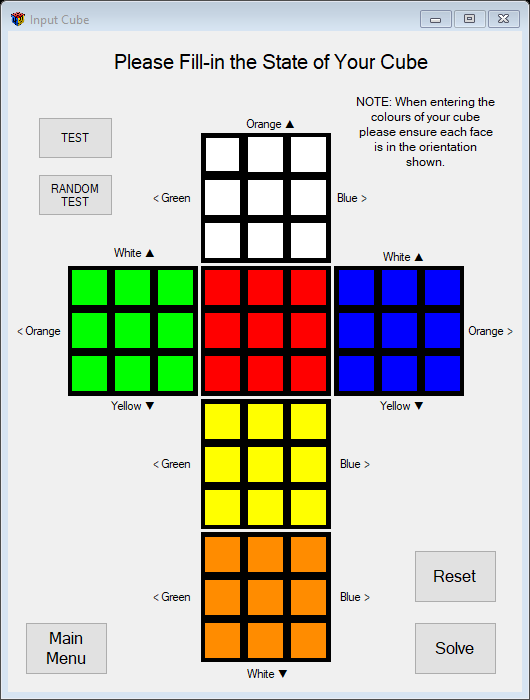
[Mouse.vb 227](#_Toc511390369)

[Exceptions.vb 228](#_Toc511390370)

[VertexShader.txt 230](#_Toc511390371)

[FragmentShader.txt 231](#_Toc511390372)

# **Testing**



## Mid-Development Testing

Throughout the development process I tested all of the functions and sections that I developed as I went along before moving onto the next section. In this section I have documented how I tested many of them.

One of the main ways that I could isolate and test each section, especially within my layer solvers, was by adding the following buttons to my Input form, along with this code:

Private \_stickers(0 To 5, 0 To 8) As Char

Private Sub Input\_Load(sender As Object, e As EventArgs) Handles MyBase.Load

ResetStickersArray(\_stickers)

End Sub

Private Sub btnTest\_Click(sender As Object, e As EventArgs) Handles btnTest.Click

Dim fileName As String = InputBox("Input name of file (without path/extention)")

If fileName = Nothing Then Exit Sub

ReadTestCube("testCubes\" & fileName & ".txt", \_stickers)

btnReset.Enabled = False

If Not CubePossible(\_stickers) Then

MsgBox("Cube Not Possible")

Return

End If

Dim cube As New Cube(\_stickers)

Dim processingWindow As New Processing(cube)

processingWindow.Show()

Me.Close()

End Sub

Private Sub ReadTestCube(ByVal filePath As String, ByRef stickersArray(,) As Char)

Dim fReader As StreamReader

Try

fReader = New StreamReader(filePath)

Catch ex As FileNotFoundException

Dim fileName As String = InputBox("Input name of file (without path/extention)")

If fileName = Nothing Then Exit Sub

ReadTestCube("testCubes\" & fileName & ".txt", stickersArray)

Return

End Try

Dim faceString As String

For face = 0 To 5

faceString = fReader.ReadLine

For sticker = 0 To 8

stickersArray(face, sticker) = faceString(sticker)

Next

Next

fReader.Close()

End Sub

Private Sub btnRndTest\_Click(sender As Object, e As EventArgs) Handles btnRndTest.Click

Randomize()

ReadTestCube("testCubes\completed.txt", \_stickers)

Dim cube As New Cube(\_stickers)

For i = 0 To CInt(Int(Rnd() \* 49)) + 10

cube.RotateFace(CInt(Int(Rnd() \* 6)), CInt(Int(Rnd() \* 4)) - 1)

Next

cube.Instructions.Clear()

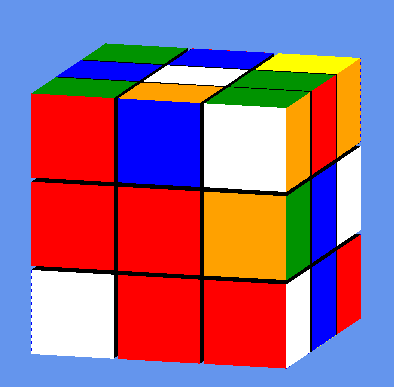
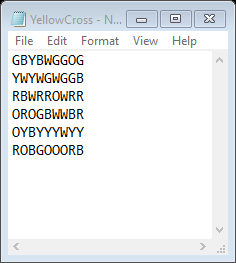
Dim processingWindow As New Processing(cube)

processingWindow.Show()

Me.Close()

End Sub

This code and the 2 buttons allowed me to carry out 2 kinds of tests. The “TEST” button allowed me to read in a text file such as the one on the right (with the rendered version of the cube next to it), where each line corresponds to a face, and the index of each character on that line corresponds to the sticker index. This way, I could represent a cube with a text file, and this allowed me to isolate parts of my program. For example, if I was testing my subroutine that orientated 3 corners, I would create a text file that needed 3 corners orientated, then use the “TEST” button to load that file, and that would test that particular part of my program.



Below is a table of the test cube files I used, and the parts of the program that I used to test/debug. Throughout the development a number of the tests failed, and I used these text files to not only test whether a section was working, but also to debug a failed test. Since my program is fully working, all of the test file now pass all tests and are solved by the program. I used various colour faces as the top face (i.e. have the blue cross already solved to designate blue as the top face) to check whether my program could handle them all, and check that it would not start solving with a different colour face on top.

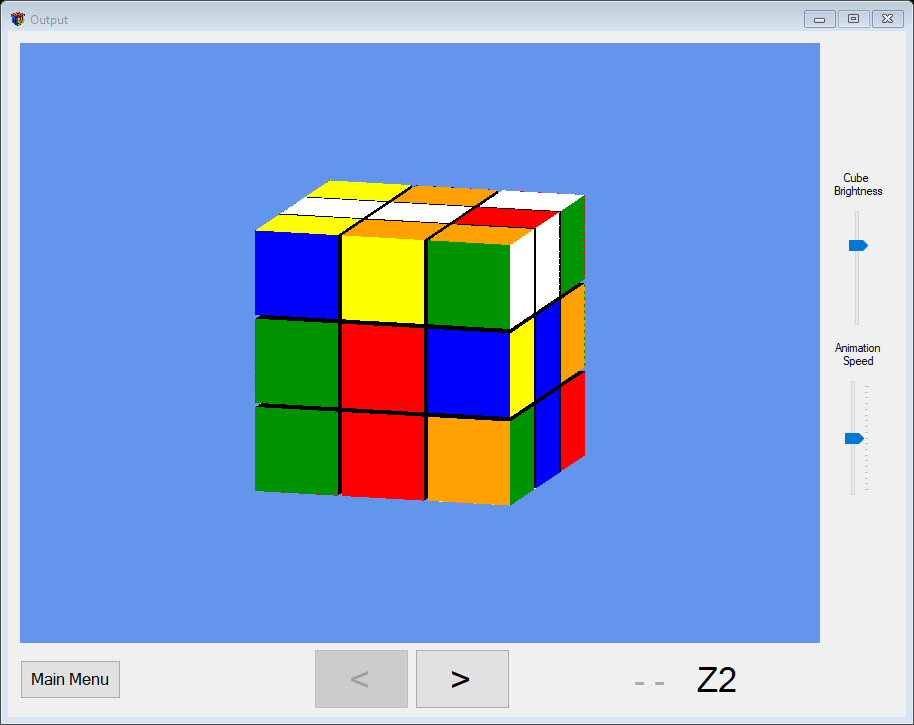
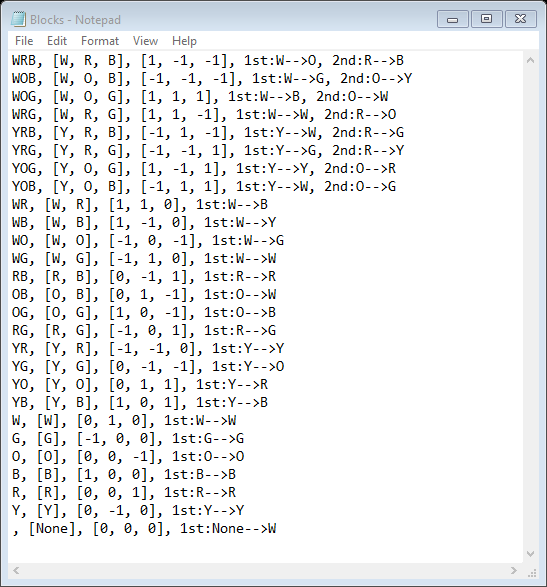
|  |  |
| --- | --- |
| **File Name** | **Reason For Creation** |
| BlueCross.txt | To test whether the program would start with the blue cross solved and continue from there, or whether it would restart the solving from the beginning: **TopLayerSolver.DoTopFace()** |
| BlueFace.txt | To test whether the program would start with the blue face solved and continue from there, or whether it would restart the solving from the beginning: **TopLayerSolver.Solve()** |
| BlueFace2.txt | To test the same as BlueFace.txt, but with a different cube to ensure consistent operation |
| BlueFace3.txt | To test the same as BlueFace.txt, but with a different cube to ensure consistent operation |
| BlueFaceTopRotated.txt | To test whether the program would detect the blue face as being solved even though it is rotated 90°: **TopLayerSolver.GetAnyRotatedCompleteFace()** |
| BlueFaceWrongEdges.txt | To test that the program would recognise the blue face with incorrect edge colours as not being solved: **TopLayerSolver.GetAnyCompleteFace()** |
| BlueFaceWrongRotation.txt | To test that the program would recognise the blue face as not being solved, because the corner pieces are in the correct place, but have the wrong orientation: **TopLayerSolver.GetAnyCompleteFace()/Block.Correct()** |
| Completed.txt | To test that the program would detect the cube as solved and not attempt to solve it further: **Cube.Complete()** |
| InvalidCube1.txt | To test the analyse module, as this is an impossible cube: **Analyse.CubePossible()** |
| OrangeFaceAndBottomCross.txt | To test the permuting of the bottom cross edges: **BottomLayerSolver.PermuteTopCrossEdges()** |
| OrangeFaceAndBottomCrossEdges.txt | To test the solving of the corners once the bottom cross is solved: **BottomLayerSolver.DoCorners()** |
| OrangeFaceAndMiddle.txt | To test the solving of the bottom layer once the middle layer has been solved: **BottomLayerSolver.Solve()** |
| OrangeFaceBottomCorners.txt | To test the solving of the corners once the bottom corners are positioned but not orientated: **BottomLayerSolver.OrientateBottomCorners()** |
| OrangeFaceTopRotated.txt | To test the same as BlueFaceTopRotated.txt but with another colour face |
| RedOppositeCorners.txt | To test **BottomLayerSolver.OrientateOppositeCorners()** |
| Scrambled1.txt | A fully scrambled cube to test the entire solution and highlight any bugs |
| Scrambled2.txt | A fully scrambled cube to test the entire solution and highlight any bugs |
| Scrambled3.txt | A fully scrambled cube to test the entire solution and highlight any bugs |
| Scrambled4.txt | A fully scrambled cube to test the entire solution and highlight any bugs |
| Scrambled5.txt | A fully scrambled cube to test the entire solution and highlight any bugs |
| Scrambled6.txt | A fully scrambled cube to test the entire solution and highlight any bugs |
| Scrambled7.txt | A fully scrambled cube to test the entire solution and highlight any bugs |
| Scrambled8.txt | A fully scrambled cube to test the entire solution and highlight any bugs |
| Scrambled9.txt | A fully scrambled cube to test the entire solution and highlight any bugs |
| Scrambled10.txt | A fully scrambled cube to test the entire solution and highlight any bugs |
| Scrambled11.txt | A fully scrambled cube to test the entire solution and highlight any bugs |
| Scrambled12.txt | A fully scrambled cube to test the entire solution and highlight any bugs |
| WhiteCross2.txt | To test the program would detect a solved cross: **TopLayerSolver.DoTopFace()** |
| WhiteCrossWrongEdges.txt | To test that the program would not detect this as a solved cross due to its incorrect edges: **TopLayerSolver.EdgesOfCrossInRightOrder()** |
| WhiteFace.txt | To test the same as BlueFace.txt but with a different colour face |
| WhiteFaceTopRotated.txt | To test the same as BlueFaceTopRotated.txt but with a different colour face |
| WhiteFaceTopRotated2.txt | To test the same as BlueFaceTopRotated.txt but with a different colour face |
| WhiteFaceYellowCross.txt | To test the same as OrangeFaceAndBottomCross.txt but with a different colour face |
| WhiteFish.txt | To test the solving of the bottom layer when it has 3 incorrect corners (a fish shape): **BottomLayerSolver.Orientate3Corners()** |
| WhiteLine.txt | To test the solving of the bottom cross when it is in a line: **BottomLayerSolver.DoBottomStage\_Cross** |
| YellowCross.txt | To test the same as BlueCross.txt with a different colour cross |
| YellowCrossCornerRotated.txt | To test that the program would output a suitable error message if the user inputted a cube which is possible, but unsolvable due to one of the corner pieces being rotated: |
| YellowCrossWrongEdges.txt | To test the same as BlueFaceWrongEdges.txt with a different colour face |

Once I finished development, I re-tested each of these test cubes and all of them were solved by the program, meaning that all of the tests and methods detailed above work as expected. Each of these tests was also a test of the whole system, as it involved the input, solving and output of each cube.

To test my matrix and vector classes I used the console and created some matrix and vector objects and then performed the dot and multiplication functions on them. I then performed the same operations by hand on a piece of paper with the same vectors and got the same result, thus showing my functions work as expected. Below are some examples of this:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Test Number | Description | Object 1 | Object 2 | Expected Result | Actual Result |
| 1.1 | Vector3x1 ● Vector 3x1 | {2, 1.5, 4} | {1, 6, 10} | 2 + 9 + 40 = 51 |  |
| 1.2 | Matrix3x3 \* Vector 3x1 | |  |  |  | | --- | --- | --- | | 1 | 2 | 0 | | -1 | 0.4 | -1 | | 0 | 0 | 4 | | {3, 4, 3} | {3+8, -3+1.6-3, 12} = {11, -4.4, 12} |  |
| 1.3 | Matrix3x3 \* Matrix3x3 | |  |  |  | | --- | --- | --- | | 1 | 2 | 1 | | 0 | 0 | 0 | | 0.1 | 2 | 0 | | |  |  |  | | --- | --- | --- | | 4 | 3 | 2 | | 0 | 2 | 0 | | 0 | 0 | -1 | | |  |  |  | | --- | --- | --- | | 4 | 7 | 1 | | 0 | 0 | 0 | | 0.4 | 4.3 | 0.2 | |  |

To test the conversion from a stickerArray to a cube object I used the **WriteBlocksToFile()** method in **Helpers.vb** and then scrambled up a real-world Rubik’s cubes and inputted them with both text files and using the input form. I then converted it to a cube object and outputted all of the blocks to a text file and then compared the properties of each cubie with its real-world equivalent on the real-cube in front of me. Here is an example of one of these text files and its rendered cube:



Because each of the files matched up with the physical cubes that I inputted, this showed that my input worked, and that my conversion to a cube object worked.

To test the optimisation of the instruction list, I ran the program with a testCube and put a breakpoint in at the return statement of the **Optimise()** function, this allowed me to see the list that was being returned. I then read through the list, comparing adjacent instruction and making sure no adjacent instructions were for the same move. I then continued to the output and ran through the optimised instructions to ensure that the optimised instructions still solved the cube. I repeated this several times and each time the instructions were fully optimised and the cube was solved. Below is one example of this:

|  |  |
| --- | --- |
|  |  |
| The InstructionList Pre-Optimisation (Count = 204) | The Cube Before Solving |
|  |  |
| The InstructionList Post-Optimisation (Count = 147) | The Cube After All the Instructions Have Been Executed |

## Final-System Testing

To test the entire system I used 3 main techniques.

1. Scrambling a physical real-world cube then inputting it into my program via the input form. Then getting the program to solve it and following along with the outputted instructions with my real cube to ensure that the instructions solve the cube fully
2. Clicking on the “ RANDOM TEST” button (see previous section) and ensuring the inputted cube is solved (if it is not solved an error message is displayed) and then stepping throughout the instructions on the output screen to ensure that the cube is solved
3. Using the “TEST” button to load test files as was detailed in the previous section

Also, to test the loading and saving mechanisms, in the middle of the 2 methods listed above, I would exit and save the cube, and then reload it and compare it to the real cube (if I was doing method 1) and then run all the instructions to the end to ensure the cube was saved and loaded correctly. I would also check that it would save and return to the instruction that I saved at.

Below is a table of some of the tests that I ran to test the entire system. I repeated each of these tests several times with different cubes but did not feel it beneficial to the project to document each test as they were all successful. Please note that the reason there are instructions waiting to be completed after some of the completed cubes (see screenshots of completed cubes in table) is that the program is designed to rotate the cube so that its top face is upwards once it is solved however I have not executed those stages in all cases.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Test No. | Test Style (1=real cube, 2 = random) | Reason For Test/Description | Input/Scrambled Cube | Output/End Of Instructions | Pass/Fail |
| 2.1 | 1 | To test whether the program can correctly take as input a scrambled cube. |  |  | Pass – The cube on the screen exactly matches the one I scrambled and inputted: |
| 2.2 | 1 | To test whether the program can solve the top cross of a real-cube (using the same cube as 2.1) |  |  | Pass – The cube on the screen exactly matches the real-one when I followed the instructions, and the white cross has been formed: |
| 2.3 | 1 | To test whether the program can solve the top layer of a real-cube (using the same cube as 2.1) |  |  | Pass – The cube on the screen exactly matches the real-one when I followed the instructions, and the white layer has been completed: |
| 2.4 | 1 | To test whether the program can solve the middle layer of a real cube (using the same cube as 2.1) |  |  | Pass – The cube on the screen exactly matches the real-one when I followed the instructions, and the middle layer has been completed: |
| 2.5 | 1 | To test whether the program can solve the bottom cross of a real cube (using the same cube as 2.1) |  |  | Pass – The cube on the screen exactly matches the real-one when I followed the instructions, and the bottom cross has been completed: |
| 2.6 | 1 | To test whether the program can position the bottom corners of a real cube (using the same cube as 2.1) |  |  | Pass – The cube on the screen exactly matches the real-one when I followed the instructions, and the bottom corners are correctly positioned: |
| 2.7 | 1 | To test whether the program can fully solve a cube (using the same cube as 2.1) |  |  | Pass – The cube on the screen exactly matches the real-one when I followed the instructions, and is fully solved: |
| 3.1 | 2 | To test the entire system with a random cube |  |  | Pass – The inputted cube was fully solved |
| 3.2 | 2 | To test the entire system with a random cube |  |  | Pass – The inputted cube was fully solved |
| 4.1 | 2 | To test whether the program can successfully save and load a cube and return to the same instruction as before | Before Saving: | After Closing and Re-Loading: | Pass – The cube was saved to a file, and when the file was opened the cube was identical and the instructions were at the same point (see bottom right of screenshots) |
| 4.2 | 2 | To test whether the program can successfully save and load a cube and then that the instructions will still solve the cube fully once it is loaded (using the same save file as 4.1) |  |  | Pass – After the saved file was loaded again, it was able to be fully solved |
| 5.1 | 2 | To test whether instructions can be undone and redone, I undid 20 instructions and the re-did them and compared the state of the cube | Before Undoing: | After Undoing:    After Re-doing | Pass – The instructions were able to be undone and then re-shown and the cube returned to its previous state |
| 6.1 | 3 | To test if the program would show a suitable error message I loaded in a text file with an incorrect number of colours |  |  | Pass – The program detected the incorrect number of blue stickers and displayed an error message |
| 6.2 | 3 | To test if the program would show a suitable error message I loaded in a text file with an invalid edge piece |  |  | Pass – The program detected the incorrect cubie and displayed an error message |
| 6.3 | 3 | To test if the program would show a suitable error message I loaded a text file cube in which one of the corners was taken out and put back in incorrectly – making the cube unsolvable |  |  | Pass – The program detected the incorrect cubie and displayed an error message |
| 7.1 | 2 | To test what would happen if the instruction pointer file (.cube.ptr) got deleted I saved a partially solved cube then deleted the .ptr file and tried to load it again |  |  | Pass – The program caught the error and displayed an error message |

To show another form of evidence, I have recorded a video to show my program taking an input, solving it, and outputting the instructions, and the saving and loading of the file, the video can be found here:

<https://youtu.be/S5cC6rdJwaM>

Below is a list of the tests I carried out in the video, along with the times at which I the test starts

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Test No. | Reason For Test/Description | Start Time | End Time | Pass/Fail |
| 8.1 | To test that the input form functioned by inputting a real-life cube and comparing it to the rendered one | 0.07 | 1.51 | Pass – I was able to input the real cube and have it be rendered by the program |
| 8.2 | To test whether the program could detect the wrong number of yellow stickers when I changed the bottom right sticker from blue (like it is on the real-cube) to yellow | 1.39 | 1.47 | Pass – The program detected it and displayed an error message |
| 8.3 | To test whether the program would solve the inputted cube | 1.51 | 1.54 | Pass – The program progressed throughout the 3 layers in the processing window and onto the output screen |
| 8.4 | To test whether I could rotate the cube within the bounds, and that any vertical rotation was reversed to the default position | 1.55 | 2.11 | Pass – the program allowed me to rotate the cube with my mouse, but only within the desired bounds, and after looking at the top/bottom face, the cube returned to its original position (especially visible at 2.10) |
| 8.5 | To test that the brightness of the cube could be changed with the slider | 2.12 | 2.17 | Pass – moving the slider down made the cube darker and moving it up made it lighter |
| 8.6 | To test that I could control the speed of rotation with the slider | 2.18 | 2.30 | Pass – moving the slider down made it rotate slower, and moving it up made it rotate quicker |
| 8.7 | To test that I could do steps, undo them, and then redo them again to the same point | 3.16 | 4.12 | Pass – I could carry out steps, undo them, and replay them without any missing steps and I could return to the same point in the instructions and the cube would be the same as the first time |
| 8.8 | To test that the cube could be fully solved after undoing instructions | 4.12 | 5.00 | Pass – the cube was fully solved by the program |
| 8.9 | To test that I could save and load a cube | 5.21 | 5.46 | Pass – I saved the cube and then reloaded it and it was identical and in the same state and point in the instructions as when I saved it |
| 8.10 | To test that the cube was still fully solvable after saving and re-loading it | 5.46 | 6.04 | Pass – once I had reloaded the cube it was still solved by the program |

# **Evaluation**

## Overall Summary

Overall I think that my solution solves the client’s problem well, and although there are some areas that could be improved with more time (as with all projects), I think that my program fits the brief, and fulfils its purpose of being a Rubik’s cube solver, as it can fully solve an inputted Rubik’s cube, and show the instructions to the user.

## Evaluation of Objectives

|  |  |  |  |
| --- | --- | --- | --- |
| Objective | Met/Not Met? | Comments | Evidence  (Test 8 = the video) |
| **1.     Allow the user to input the state of their cube** | MET | The user can use the input form to input the state of the cube. The use of the clickable GUI made it quick to enter the colours, however the net of the cube could be a little confusing at times, so if I had had more time I would have liked to implement objective 1.4 | Tests 2.1 and 8.1 |
| 1.1.     Use a GUI to enable easier input | MET | The user can click on the ‘stickers’ on the input screen to cycle through the colours, this was an effective method of input, and was better suited to the program, than say a drop down list/pop-up window of colours | Test 8.1 |
| 1.2.     Allow choice from any of the 6 colours | MET | When cycling throught the colours with their mouse, the user can select any of the 6 colours, and only those 6. This allows the user complete freedom over the colour of that sticker, and also prevents erroneous data entry, as such I think that this was an effective solution to the input. | Test 8.1, and Input.vb: buttonColours() |
| 1.3.     Allow each sticker to be edited after it has been entered | MET | Each sticker can be clicked as many times as needed as I implemeneted the colours in a cyclical pattern, so when the user reaches the end of the buttonColours array, it goes back to the first item in the array. | Test 8.2 |
| 1.4.     Allow for input via a 3D GUI ***(optional extension objective)*** | NOT MET | This was not implemented in time, however it was agreed as an optional extra by the client and myself before devlopment began. I will detail in the possible improvement section below how I would have implemeneted it. | - |
| **2.     Have a menu system** | MET | The program is Windows form based, and is navigated by clicking on buttons to move between the forms – as the client requested – it also has a main menu form which allows the user to access different parts of the program from 1 form | Test 8, and the screenshots of the forms in Appendix D |
| 2.1.     Should be able to return to previous screens | MET | Each of the forms (apart from the processing screen which is only visible for a few seconds whilst the cube is solved) has a button to return to the main menu screen. If I had had more time I could have added a button onto the 3D output screen to return to the input screen to modify the inputted cube, however I deemed this unlikely to be needed once the user has inputted their cube and progressed to the output screen. | Test 8, and the screenshots of the forms in Appendix D |
| *2.1.1.* Data should be saved or a confirmation message should be displayed | MET | Upon returning to the main menu from the output screen the user is asked whether they want to save their cube, not save it, or stay on the output form with a message box. | Test 8.9 |
| 2.2.     The menu system should be easy to navigate | MET | The forms had a similar layout and style throughout the program, and as there were minimal buttons, this aided in the simplicity and easy-of-use of the program. Overall I think that my design of the GUI was simple and effective. | Test 8, and the screenshots of the forms in Appendix D |
| *2.2.1.* Clearly labelled buttons | MET | I used a large, clear font on my buttons, and they were labelled so that they are easy to understand. | Test 8, and the screenshots of the forms in Appendix D |
| *2.2.2.* Similar layout and style throughout the solution | MET | I used the same font and colours throughout my program and positioned the main-menu button in the bottom-left corner of each form. | Test 8, and the screenshots of the forms in Appendix D |
| **3.     Validate if the entered cube is a valid cube** | MET | The program correctly identified each of the test cases that I put to it and outputted an error message to the user telling them what makes the cube invalid. | Tests 6 and 8.2 |
| 3.1.     Check there are 9 of each colour inputted | MET | The program correctly checked the number of each colour. | Test 8.2 |
| 3.2.     Check that no 2 opposite colour stickers are on the same block | MET | The program correctly checked this every time that I tested it. | 6.2 |
| 3.3.     Check that there are no of the same colour stickers on the same block | MET | The program correctly checked this every time that I tested it. | Not evidenced but was tested the same way as test 6.2 |
| **4.     Work out the instructions needed to solve the cube from its inputted state** | MET | The program can correctly calculate the instructions required to fully solve all of the cubes that I give it. I tested this many more times to ensure robustness of my solution, but did not document them all as the procedure and result was the same each time. | Tests 2.7, 3.1, 3.2, 8.3 |
| 4.1.     The instructions should solve the cube fully | MET | The instructions generated each time fully solve the cube. | Tests 2.7, 3.1, 3.2, 8.3 |
| 4.2.     The instructions should be fully simplified in terms of consecutive rotations of the same face | MET | The instructionList is fully simplified each time such that there are no 2 consecutive rotations of the same face/about the same axis | Page 80 (testing the optimise function) and Test 8 |
| *4.2.1.* 2 quarter turns in a row should be replaced with a half turn | MET | This is done each time in the **InstructionList.Optimise()** subroutine. | Page 80 and Test 8 |
| *4.2.2.* 2 quarter turns in opposite directions in a row should both be removed | MET | This is done each time in the **InstructionList.Optimise()** subroutine. | Page 80 and Test 8 |
| *4.2.3.* 2 half turns in in a row should both be removed | MET | This is done each time in the **InstructionList.Optimise()** subroutine. | Page 80 and Test 8 |
| *4.2.4.* A half turn and quarter turn in a row should be replaced with the opposite quarter turn (e.g. ¼ clockwise + ½  ¼ anticlockwise) | MET | This is done each time in the **InstructionList.Optimise()** subroutine. | Page 80 and Test 8 |
| 4.3.     The instructions should follow the Beginner’s Method | MET | The instructions follow the Beginner’s Method, and solve the cube in 3 layers, first with the top cross then the top corners, then the middle layer, then the bottom cross and finally the bottom corners. | Tests 2 and 8 |
| 4.4.     The instructions should be able to solve the cube from a half-solved state without restarting (e.g. if the green face is correctly solved it should carry on from that point) | MET | The program can start solving from a half solved state (e.g. if the blue cross is solved, or if the top 2 layers are solved) without restarting the solving process | Some of the test files on pages 78 and 79. |
| **5.     Output the instructions to solve the cube** | MET | The instructions are shown to the user clearly with a 3D model one at a time. The user is also able to control the speed at which the instruction is carried out (I.e. the rotation speed) | Test 8 |
| 5.1.     Output each step one at a time | MET | Each instruction is shown one at a time as the user needs to click on the next arrow to advance to the next instruction. This also means that the instructions are shown at the right rate for the user. | Test 8 |
| 5.2.     Allow the user to replay steps | MET | The user can undo and replay steps any amount of times correctly without affecting the solve or the cube. | Tests 5.1, 8.7, 8.8 |
| *5.2.1.* Steps should be able to be undone | MET | Steps can be undone with the undo arrow, and a reverse rotation is shown on the 3D model | Test 8.7 |
| *5.2.2.* Steps should be able to be repeated once undone | MET | Steps can then be replayed once they are undone using the next arrow. | Tests 5.1, 8.7 |
| *5.2.3.* Steps should be able to be undone and replayed any number of times | MET | Undoing and redoing steps has no affect on the solving/instructions and so can be carried out any number of times | Tests 5.1, 8.7 |
| 5.3.     Output the instructions in a 3D graphical format | MET | The instructions are shown to the user via the 3D Output form. | Test 8 |
| *5.3.1.* If 3D is not available on the user’s machine, output a suitable message and switch to text-based instructions | MET | If the versionof OpenGL on the user’s machine is not compatible with the program (before 2006) then the user is given a message with a text box and on the form, and the arrows step through the test based instructions only. | See screenshot in AppendixD: 3DOutput |
| *5.3.2.* Display the cube turning to clearly show the user the instruction | MET | The speed can be controlled and the cube is large enough on the screen so that the moves are clear and visible. | Test 8 |
| *5.3.3.* Allow the user to control the speed of rotation | MET | The speed of rotation can be controlled by the user so that the turning is clear. | Test 8.6 |
| 5.4.     Also offer the option of text-based instructions using Rubik’s Notation (see **Appendix B**) | MET | The text-based Rubik’s notation instructions are shown on the bottom right of the window, and match up with the movbes being shown on the 3D model. | Test 8 |
| **6.     Allow the user to store the cube in the middle of being solved** | MET | When returning to the main menu from the 3DOutput form the user can save their cube in its current state and then reload it at a later date. | Tests 4.1 and 8.9 |
| 6.1.     Save the cube in a file | MET | Each cube is saved in one .cube and one .cube.ptr files. | Tests 4.1 and 8.9 |
| 6.2.     Allow cubes to be retrieved from the files at the same instruction that the user saved it on | MET | From the main menu, the user can load a saved cube and can continue from the save instruction. | Tests 4.1, 4.2, 8.9, 8.10 |
| *6.2.1.* Store the current instruction alongside the cube | MET | The current instruction that the cube is at is save in a .cube.ptr file with the same name as the .cube file. | Tests 4.1, 8.9 |
| *6.2.2.* If the current instruction file is deleted then output a suitable message to the user | MET | If the .cube.ptr file is deleted the user is given an error message and the cube is loaded from the first instruction again. | Test 7.1 |

## Client Feedback

**Please score each element from 1 (not what you wanted at all) to 5 (exactly what you wanted)**

### The Input

* 1. **Score**

5

* 1. **Why?**The input matched what I described in the initial interview
  2. **How would like it improved in a later version/if there was more time?**

Despite my initial designs, I found this input method, in the end, to be quite complicated and confusing. In the future maybe a step by step instructional type input system could be created to make it easier to understand. I found it hard to visualise the cube as a net.

### The Output

* 1. **Score**

4

* 1. **Why?**

Most moves were clear, except for when a half-turn was required, this went too fast and I had to replay it to see the move again. The adjustable settings worked very well however there was sometimes a lack of definition between the individual cubes.

* 1. **How would like it improved in a later version/if there was more time?**

In the future the animation of a half-turn could be slowed down so it was the same speed as the other moves, and maybe I could have the ability to change the gap size between the individual cubes.

### The Method/Instructions to Solve the Cube

* 1. **Score**

4

* 1. **Why?**

The method to solve the Rubix cube was relatively quick; however, as there was no sense of how close you were to the final result, it did become a bit tedious at times.

* 1. **How would like it improved in a later version/if there was more time?**

To improve this there could be a counter to show how many steps remained to be completed.

### The Menu System/Program Design

* 1. **Score**

5

* 1. **Why?**

The menu was quite easy to navigate as the layout was very clean and minimal.

* 1. **How would like it improved in a later version/if there was more time?**

None

### The File Saving/Loading System

* 1. **Score**

5

* 1. **Why?**

It was very easy to use, I had no trouble saving or loading cubes, and I liked the use of the pop up windows for selecting folders/files

* 1. **How would like it improved in a later version/if there was more time?**

None

### The Program as a Whole

* 1. **Score**

4.

* 1. **Why?**

Generally the program was very good however there were a few details described above which could be improved to perfect the program

* 1. **How would like it improved in a later version/if there was more time?**

None

### Analysis of Client Feedback

Overall the client gave my project a score of 27/30 for how well it matched their initial brief. This shows that my project matches closely the initial design of the client, and although there are improvements that could be made in a future version, it works fully as desired in its current state and therefore I consider this an effective solution that meets the initial objectives. I will discuss the possible improvemenets for the program that the client suggested in the next section. The client was happy with the input in terms of it matching their design, however upon using the software they have now realised that there are other possible solutions which may work better than the net of the cube. The client was also happy with the design of the ouput being 3D rotations, they did however raise a few issues which I will discuss in the next section. With regards to the solving of the cube, the client was happy with the method, they would have just liked a counter to show how many instructions they had completed/were left. Finally, with the file and menu systems, the client was fully satisfied and said that they both matched her initial design exactly.

## Possible Improvements

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Improvement/ Addition | Section | Objective Number | Client/ Developer suggestion | Possible implementation details |
| Changing the input method to a 3D model of a cube rather than a net | Input | 1.4 | Client & Developer | I could use similar code to my output form to generate the 3D model of the cube, and the code from the input form to cycle through the colours. This however would either require converting the 3D model into an array of stickers to be used by the analyse module, or would require rewriting the analyse module to take a cube object instead of a 2D array. |
| Changing the input method to take you step by step through inputting each face | Input | 1 | Client | This would be fairly simple, although time-consuming, to implement as it would essentially be the same as the current input form, but split into faces. I would make a separate input module that would create a form object for each face and that object would return the colours for that face. |
| Slowing down the rotation speed for half-turns | Output | 5.3.2 | Client | This issue is due to my use of the same percentage increase regardless of the move. I could add a statement to check if the move is a half turn when incremeneting the percentage, and then increase the percentage by half of what it is currently for half-turns |
| Adding the ability to change the padding between cubes | Output | 5.3 | Client | This would require quite a few changes as the pad constant is used at different points in the code and in particular is used in the initialPositions of each cubie, which are readOnly, and so wouldn’t be updated if the pad was. I would have to add the pad to the initial positions in the paint function and then send it to the fragment shader for colouring purposes, as well as changes in other places where the pad constant is used currently. |
| Adding a counter to show the move that you are on/how many are remaining | Output | 5 | Client | I could use the current instruction pointer, and the count from the OuputInstructionList, and then output these values on a label using the UpdateInstructionLabels() subroutine. |
| A button to return to the input screen from the output screen | Menu | 2.1 | Developer | This would reuiqre me to either save the cube after I leave the input form, or to convert the cube back into a stickerArray to be displayed on the input form. It would also require me to add another button to the output form. |

# Appendix A: *Cube Terminology*

|  |  |  |  |
| --- | --- | --- | --- |
| **Term** | **Diagram** | **Term** | **Diagram** |
| Cube  The whole puzzle. | https://www.grubiks.com/images/rc.png | Edge [Block/Cubie/ Cubelet]  One of the 12 pieces on the edge of 2 faces. |  |
| Face  One of the faces of the cube – consisting of 9 cubies. |  | Middle [Block/Cubie/ Cubelet]  One of the 6 pieces in the middle of a face. |  |
| Corner [Block/Cubie/ Cubelet]  One of the 8 pieces in the corners of the puzzle. |  | Sticker  The stickers are on the cubies and dictate the colour of each cubie. Corners have 3, edges 2 and middles 1. |  |

**IMPORTANT NOTE:**

Throughout my project I use the terms Block, Piece and Cubie interchangeably to refer to the 27 individual segments of a Rubik’s cube.

# Appendix B: *Rubik’s Notation*

[TODO]

# Appendix C: *Beginner’s Method Algorithms*

1. **Insert middle layer edge piece right *(step 3)***

D’ R’ D R D F D’ F’

1. **Insert middle layer edge piece left *(step 3)***

D L D’ L’ D’ F’ D F

1. **Orientate bottom layer edge pieces *(step 4)***

F R U R’ U’ F’

1. **Permute bottom layer edge pieces *(step 4)***

R U R’ U R U2 R’

1. **Permute bottom corner pieces right *(step 5)***

R U’ L’ U R’ U’ L U

1. **Permute bottom corner pieces left *(step 5)***

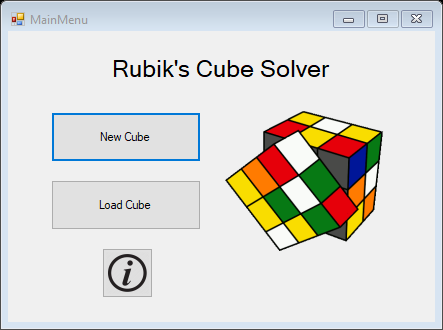
L’ U R U’ L U R’ U’

1. **Orientate bottom corner pieces *(step 5)***

R U R’ U R U2 R’ L’ U’ L U’ L’ U2 L

# Appendix D: *Program Code*

## MainMenu.vb



Public Class MainMenu

Private Sub btnNew\_Click(sender As Object, e As EventArgs) Handles btnNew.Click

Dim inputForm As New Input()

inputForm.Show()

Me.Close()

End Sub

Private Sub btnLoad\_Click(sender As Object, e As EventArgs) Handles btnLoad.Click

LoadCube()

End Sub

Private Sub btnCredits\_Click(sender As Object, e As EventArgs) Handles btnCredits.Click

Dim creditsForm As New Credits()

creditsForm.ShowDialog()

End Sub

Private Sub LoadCube()

Dim fileBrowser As New OpenFileDialog()

fileBrowser.Filter = "cube files (\*.cube)|\*.cube"

fileBrowser.ShowDialog()

Try

Dim filePath As String = fileBrowser.FileName

If filePath = "" Then Return

Dim outputForm As New \_3DOutput(filePath)

outputForm.Show()

Me.Close()

Catch ex As Exception

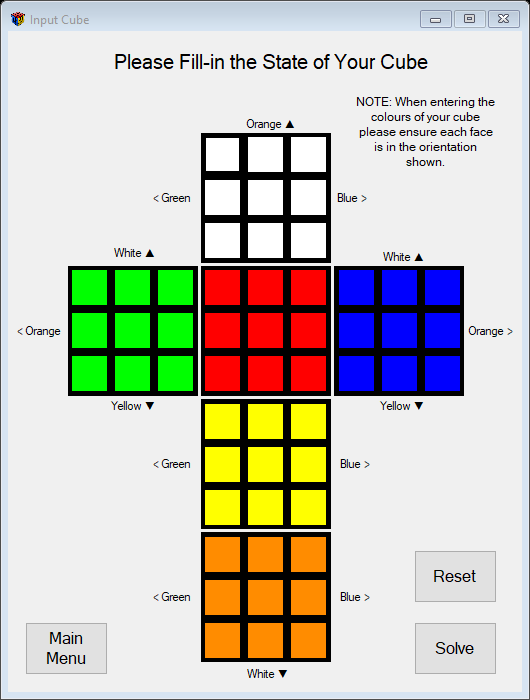
MsgBox("Cannot Read File: " & ex.Message)

End Try

End Sub

End Class

## Input.vb



Imports System.IO

Imports RubiksCubeSolver\_v2\_0.Helpers

Public Class Input

ReadOnly buttonColours As Color() = {Color.White, Color.Lime, Color.Red, Color.Blue, Color.Yellow, Color.DarkOrange}

''' <summary> Handles click events from any of the colour selection boxes </summary>

Private Sub ColourChange\_Click(sender As Object, e As EventArgs) Handles a1.Click, b1.Click, c1.Click, d1.Click,

f1.Click, g1.Click, h1.Click, i1.Click, a2.Click, b2.Click, c2.Click, d2.Click, f2.Click, g2.Click, h2.Click,

i2.Click, a3.Click, b3.Click, c3.Click, d3.Click, f3.Click, g3.Click, h3.Click, i3.Click, a4.Click, b4.Click,

c4.Click, d4.Click, f4.Click, g4.Click, h4.Click, i4.Click, a5.Click, b5.Click, c5.Click, d5.Click, f5.Click,

g5.Click, h5.Click, i5.Click, a6.Click, b6.Click, c6.Click, d6.Click, f6.Click, g6.Click, h6.Click, i6.Click

DirectCast(sender, Button).BackColor = GetNextButtonColour(sender)

End Sub

''' <summary> Returns the next colour in the sequence for a specific button </summary>

Private Function GetNextButtonColour(ByVal button As Button) As Color

Return buttonColours((Array.IndexOf(buttonColours, button.BackColor) + 1) Mod 6)

End Function

Private Sub btnSolve\_Click(sender As Object, e As EventArgs) Handles btnSolve.Click

btnReset.Enabled = False

Dim stickerColours As Char(,) = ConvertButtonColoursToCharArray(Controls)

If Not Analyse.CubePossible(stickerColours) Then

btnReset.Enabled = True

Return

End If

Dim cube As New Cube(stickerColours)

Dim processingWindow As New Processing(cube)

processingWindow.Show()

Me.Close()

End Sub

''' <summary> Reads each of the sticker buttons and returns an array of their colours </summary>

Private Function ConvertButtonColoursToCharArray(ByVal controls As Control.ControlCollection) As Char(,)

Dim stickersArray(5, 8) As Char

For face = 0 To 5

For Each sticker As Char In STICKER\_LETTERS.ToCharArray()

Dim buttonName As String = sticker & (face + 1).ToString() 'e.g. "a2"

Dim button As Button = controls.Find(buttonName, True)(0)

Dim stickerNumber As Integer = InStr(STICKER\_LETTERS, sticker) - 1

'InStr() used to get (position of sticker in "abcdefghi" - 1) == y-coordinate of sticker

stickersArray(face, stickerNumber) = GetButtonColour(button, controls)

Next

Next

Return stickersArray

End Function

'''<summary> Gets the colour character of a specific button </summary>

Private Function GetButtonColour(ByVal box As Button, ByVal controls As Control.ControlCollection) As Char

Dim buttonColour As Char = Mid(box.BackColor.ToString(), 8, 1)

'e.g. if box.BackColor.ToString = "Colour [White]", Mid(8, 1) = "W"

If buttonColour = "D" Then

buttonColour = "O" 'converts DarkOrange to "O"(for Orange)

ElseIf buttonColour = "L" Then

buttonColour = "G" 'converts Lime to "G"(for Green)

End If

Return buttonColour

End Function

Private Sub btnReset\_Click(sender As Object, e As EventArgs) Handles btnReset.Click

ResetBoxColours(Controls)

ResetStickersArray(\_stickers)

End Sub

Private Sub btnMain\_Click(sender As Object, e As EventArgs) Handles btnMain.Click

Dim main As New MainMenu

main.Show()

Me.Close()

End Sub

''' <summary> Resets all the colours of the buttons to their default colours </summary>

Private Sub ResetBoxColours(ByVal controls As Control.ControlCollection)

For Each faceChar As Char In STICKER\_LETTERS\_NO\_MIDDLE.ToCharArray()

For faceNumber = 1 To 6

Dim face As Helpers.FaceColour = faceNumber - 1

Dim buttonName As String = faceChar & faceNumber.ToString() 'e.g. "b5"

Dim button As Button = controls.Find(buttonName, True)(0)

button.BackColor = buttonColours(face)

Next

Next

End Sub

Private Sub ResetStickersArray(ByRef stickersArray(,) As Char)

For i = 0 To 5

Dim face As Helpers.FaceColour = i

For j = 0 To 8

stickersArray(i, j) = face.ToString()

Next

Next

End Sub

End Class

## Analyse.vb

Imports RubiksCubeSolver\_v2\_0.Helpers.PublicFunctions

Imports RubiksCubeSolver\_v2\_0.Helpers.PublicConstants

Public Module Analyse

''' <summary> Returns true if the cube is fully possible </summary>

Public Function CubePossible(ByVal stickers(,) As Char) As Boolean

If Not NineOfEachColourSticker(stickers) Then Return False

If Not EdgesPossible(stickers) Then Return False

If Not CornersPossible(stickers) Then Return False

Return CornerRotationsPossible(stickers)

End Function

''' <summary> Returns true if there are 9 of each colour sticker </summary>

Private Function NineOfEachColourSticker(ByVal stickers(,) As Char) As Boolean

If AnythingElseInArray(NumberOfEachColour(stickers), 9) Then

MsgBox("You have not entered the correct number of each colour. Are you sure you entered all the squares

correctly?")

Return False

End If

Return True

End Function

''' <summary>

''' Returns the number of each letter in a 2d char. array(,) as an array(0 To 5) i.e. ("bgorwy") b:0, g:1 etc...

''' </summary>

Public Function NumberOfEachColour(ByVal chararray(,) As Char) As Integer()

If chararray.Length < 1 Then Return Nothing

Dim number(0 To 5) As Integer

number.Zero()

For i = 0 To UBound(chararray, 1)

For j = 0 To UBound(chararray, 2)

number(InStr(FACE\_COLOURS.ToUpper(), chararray(i, j)) - 1) += 1

Next

Next

Return number

End Function

''' <summary> Returns true if all the edges are possible </summary>

Private Function EdgesPossible(ByVal stickers(,) As Char) As Boolean

For faceNumber = 0 To 5

For edgeNumber = 1 To 7 Step 2

Dim stickerColour As Char = stickers(faceNumber, edgeNumber)

Dim oppColour As Char = Opposite(stickerColour).ToString()

Dim adjacentSticker As Sticker = AdjacentEdge(faceNumber, edgeNumber)

Dim adjacentStickerColour As Char = stickers(adjacentSticker.FaceNumber, adjacentSticker.StickerNumber)

If adjacentStickerColour = stickerColour Or adjacentStickerColour = oppColour Then

DisplayEdgeErrorMessage(stickers, {faceNumber, adjacentSticker.FaceNumber})

Return False

End If

Next edgeNumber

Next faceNumber

Return True

End Function

Private Sub DisplayEdgeErrorMessage(ByVal stickers(,) As Char, ByVal wrongFaceColours() As Integer)

Dim wrongFace1, wrongFace2 As String

wrongFace1 = ColourCharToWord(stickers(wrongFaceColours(0), 4)).ToLower()

wrongFace2 = ColourCharToWord(stickers(wrongFaceColours(1), 4)).ToLower()

MsgBox("The middle cube on the " & wrongFace1 & "/" & wrongFace2 & " edge can't be that. Are you sure you

entered it correctly?")

End Sub

''' <summary> Returns true if all the corners are possible </summary>

Private Function CornersPossible(ByVal stickers(,) As Char) As Boolean

For faceNumber = 0 To 5

For cornerNumber = 0 To 8 Step 2

If cornerNumber = MIDDLE\_STICKER Then Continue For

Dim otherTwoStickers() As Sticker = AdjacentCorners(faceNumber, cornerNumber)

'3 stickers per corner

Dim sticker0Colour As Char = stickers(faceNumber, cornerNumber)

Dim sticker1Colour As Char = stickers(otherTwoStickers(0).FaceNumber, otherTwoStickers(0).StickerNumber)

Dim sticker2Colour As Char = stickers(otherTwoStickers(1).FaceNumber, otherTwoStickers(1).StickerNumber)

If Not CornerPossible({sticker0Colour, sticker1Colour, sticker2Colour}) Then

DisplayCornerErrorMessage(stickers,

{faceNumber, otherTwoStickers(0).FaceNumber, otherTwoStickers(1).FaceNumber})

Return False

End If

Next cornerNumber

Next faceNumber

Return True

End Function

Private Function CornerPossible(ByVal cornerColours() As Char) As Boolean

Dim oppColour As Char = Opposite(cornerColours(0)).ToString()

'if any pair of adjacent corner stickers are the same or opposite colours, return false

Return Not (cornerColours(1) = cornerColours(0) Or

cornerColours(2) = cornerColours(0) Or

cornerColours(1) = oppColour Or

cornerColours(2) = oppColour Or

cornerColours(1) = cornerColours(2))

End Function

Private Sub DisplayCornerErrorMessage(ByVal stickers(,) As Char, ByVal wrongFaces() As Integer)

Dim wrongFace1, wrongFace2, wrongFace3 As String

wrongFace1 = ColourCharToWord(stickers(wrongFaces(0), 4)).ToLower()

wrongFace2 = ColourCharToWord(stickers(wrongFaces(1), 4)).ToLower()

wrongFace3 = ColourCharToWord(stickers(wrongFaces(2), 4)).ToLower()

MsgBox("The cube on the " & wrongFace1 & "/" & wrongFace2 & "/" & wrongFace3 & " corner can't be that.

Are you sure you entered it correctly?")

End Sub

''' <summary> Returns true if all the corners have their stickers in the right order </summary>

Private Function CornerRotationsPossible(ByVal stickers(,) As Char) As Boolean

For faceNumber = 0 To 5

For cornerNumber = 0 To 8 Step 2

If cornerNumber = MIDDLE\_STICKER Then Continue For

Dim cornerStickers() As Sticker = AdjacentCorners(faceNumber, cornerNumber)

Dim firstStickerColour As Char = stickers(faceNumber, cornerNumber)

Dim secondStickerColour As Char = stickers(cornerStickers(0).FaceNumber, cornerStickers(0).StickerNumber)

Dim thirdStickerColour As Char = stickers(cornerStickers(1).FaceNumber, cornerStickers(1).StickerNumber)

If firstStickerColour <> "W" And firstStickerColour <> "Y" Then Continue For

If thirdStickerColour <> getCorrectThirdCornerStickerColour(firstStickerColour, secondStickerColour) Then

DisplayCornerErrorMessage(stickers,

{faceNumber, cornerStickers(0).FaceNumber, cornerStickers(1).FaceNumber})

Return False

End If

Next cornerNumber

Next faceNumber

Return True

End Function

Private Function getCorrectThirdCornerStickerColour(ByVal firstStickerColour As Char, ByVal secondStickerColour As Char) As Char

Dim correctThirdStickerColour As Char

If firstStickerColour = "W" Then

Select Case secondStickerColour

Case "R" : correctThirdStickerColour = "G"

Case "O" : correctThirdStickerColour = "B"

Case "G" : correctThirdStickerColour = "O"

Case "B" : correctThirdStickerColour = "R"

End Select

ElseIf firstStickerColour = "Y" Then

Select Case secondStickerColour

Case "R" : correctThirdStickerColour = "B"

Case "O" : correctThirdStickerColour = "G"

Case "G" : correctThirdStickerColour = "R"

Case "B" : correctThirdStickerColour = "O"

End Select

End If

Return correctThirdStickerColour

End Function

'''<summary> Returns the 2nd sticker that makes up an edge piece </summary>

Public Function AdjacentEdge(ByVal a As Integer, ByVal b As Integer) As Sticker

'Adjacent sides = (0,1)(5,7) (0,3)(1,1) (0,5)(3,1) (0,7)(2,1)

'(a,b)(x,y) (1,1)(0,3) (1,3)(5,3) (1,5)(2,3) (1,7)(4,3)

' (2,1)(0,7) (2,3)(1,5) (2,5)(3,3) (2,7)(4,1)

' (3,1)(0,5) (3,3)(2,5) (3,5)(5,5) (3,7)(4,5)

' (4,1)(2,7) (4,3)(1,7) (4,5)(3,7) (4,7)(5,1)

' (5,1)(4,7) (5,3)(1,3) (5,5)(3,5) (5,7)(0,1)

Dim x, y As Integer

Select Case a

Case 0

Select Case b

Case 1 : x = 5 : y = 7

Case 3 : x = 1 : y = 1

Case 5 : x = 3 : y = 1

Case 7 : x = 2 : y = 1

End Select

Case 1

Select Case b

Case 1 : x = 0 : y = 3

Case 3 : x = 5 : y = 3

Case 5 : x = 2 : y = 3

Case 7 : x = 4 : y = 3

End Select

Case 2

Select Case b

Case 1 : x = 0 : y = 7

Case 3 : x = 1 : y = 5

Case 5 : x = 3 : y = 3

Case 7 : x = 4 : y = 1

End Select

Case 3

Select Case b

Case 1 : x = 0 : y = 5

Case 3 : x = 2 : y = 5

Case 5 : x = 5 : y = 5

Case 7 : x = 4 : y = 5

End Select

Case 4

Select Case b

Case 1 : x = 2 : y = 7

Case 3 : x = 1 : y = 7

Case 5 : x = 3 : y = 7

Case 7 : x = 5 : y = 1

End Select

Case 5

Select Case b

Case 1 : x = 4 : y = 7

Case 3 : x = 1 : y = 3

Case 5 : x = 3 : y = 5

Case 7 : x = 0 : y = 1

End Select

End Select

Return New Sticker(x, y)

End Function

'''<summary> Returns the 2nd and 3rd stickers that makes up a corner piece </summary>

Public Function AdjacentCorners(ByVal a As Integer, ByVal b As Integer) As Sticker()

'Adjacent Coners = (0,0)(1,0)(5,6) (0,2)(5,8)(3,2) (0,6)(2,0)(1,2) (0,8)(3,0)(2,2)

'(a,b)(w,x)(y,z) (1,0)(5,6)(0,0) (1,2)(0,6)(2,0) (1,6)(4,6)(5,0) (1,8)(2,6)(4,0)

' (2,0)(1,2)(0,6) (2,2)(0,8)(3,0) (2,6)(4,0)(1,8) (2,8)(3,6)(4,2)

' (3,0)(2,2)(0,8) (3,2)(0,2)(5,8) (3,6)(4,2)(2,8) (3,8)(5,2)(4,8)

' (4,0)(1,8)(2,6) (4,2)(2,8)(3,6) (4,6)(5,0)(1,6) (4,8)(3,8)(5,2)

' (5,0)(1,6)(4,6) (5,2)(4,8)(3,8) (5,6)(0,0)(1,0) (5,8)(3,2)(0,2)

Dim w, x, y, z As Integer

Select Case a

Case 0

Select Case b

Case 0 : w = 1 : x = 0 : y = 5 : z = 6

Case 2 : w = 5 : x = 8 : y = 3 : z = 2

Case 6 : w = 2 : x = 0 : y = 1 : z = 2

Case 8 : w = 3 : x = 0 : y = 2 : z = 2

End Select

Case 1

Select Case b

Case 0 : w = 5 : x = 6 : y = 0 : z = 0

Case 2 : w = 0 : x = 6 : y = 2 : z = 0

Case 6 : w = 4 : x = 6 : y = 5 : z = 0

Case 8 : w = 2 : x = 6 : y = 4 : z = 0

End Select

Case 2

Select Case b

Case 0 : w = 1 : x = 2 : y = 0 : z = 6

Case 2 : w = 0 : x = 8 : y = 3 : z = 0

Case 6 : w = 4 : x = 0 : y = 1 : z = 8

Case 8 : w = 3 : x = 6 : y = 4 : z = 2

End Select

Case 3

Select Case b

Case 0 : w = 2 : x = 2 : y = 0 : z = 8

Case 2 : w = 0 : x = 2 : y = 5 : z = 8

Case 6 : w = 4 : x = 2 : y = 2 : z = 8

Case 8 : w = 5 : x = 2 : y = 4 : z = 8

End Select

Case 4

Select Case b

Case 0 : w = 1 : x = 8 : y = 2 : z = 6

Case 2 : w = 2 : x = 8 : y = 3 : z = 6

Case 6 : w = 5 : x = 0 : y = 1 : z = 6

Case 8 : w = 3 : x = 8 : y = 5 : z = 2

End Select

Case 5

Select Case b

Case 0 : w = 1 : x = 6 : y = 4 : z = 6

Case 2 : w = 4 : x = 8 : y = 3 : z = 8

Case 6 : w = 0 : x = 0 : y = 1 : z = 0

Case 8 : w = 3 : x = 2 : y = 0 : z = 2

End Select

End Select

Return {New Sticker(w, x), New Sticker(y, z)}

End Function

End Module

## CoordinatePairs.vb

Public Module CoordinatePairs

Public Class Sticker

Public Property FaceNumber As Integer

Public Property StickerNumber As Integer

Public Sub New()

FaceNumber = 0

StickerNumber = 0

End Sub

Public Sub New(ByVal x As Integer, ByVal y As Integer)

Me.FaceNumber = x

Me.StickerNumber = y

End Sub

Public Sub New(ByVal button As Button)

Me.FaceNumber = Mid(button.Name, 2, 1) - 1 'e.g. a5 -> 4

Dim stickerLetter As String = Mid(button.Name, 1, 1) 'e.g. a5 -> a

Me.StickerNumber = Asc(stickerLetter) - 97

End Sub

Public Shared Operator =(ByVal sticker1 As Sticker, ByVal sticker2 As Sticker) As Boolean

Return sticker1.FaceNumber = sticker2.FaceNumber And sticker1.StickerNumber = sticker2.StickerNumber

End Operator

Public Shared Operator <>(ByVal sticker1 As Sticker, ByVal sticker2 As Sticker) As Boolean

Return sticker1.FaceNumber <> sticker2.FaceNumber Or sticker1.StickerNumber <> sticker2.StickerNumber

End Operator

End Class

' A collection of 3 stickers that make up a corner cubie

Public Class CornerTriplet

Private ReadOnly cornerTriples(,) As Sticker = {{New Sticker, New Sticker, New Sticker},

{New Sticker, New Sticker, New Sticker},

{New Sticker, New Sticker, New Sticker},

{New Sticker, New Sticker, New Sticker}}

Private \_corners() As Sticker = {New Sticker, New Sticker, New Sticker}

Public Property Corners(ByVal index As Integer) As Sticker

Get

Return \_corners(index)

End Get

Set(value As Sticker)

\_corners(index) = value

End Set

End Property

Public Sub New(ByVal stickerOne As Sticker)

If cornerTriples(0, 2).FaceNumber = 0 Then SetUpCornerTriplets()

\_corners(0) = stickerOne

Dim pairIndex, argumentCornerIndex As Integer

' Gets the index for the triple containing the sticker passed as an argument

For j = 0 To UBound(cornerTriples, 2)

For i = 0 To UBound(cornerTriples, 1)

If cornerTriples(i, j) <> stickerOne Then Continue For

pairIndex = i

argumentCornerIndex = j

Next

Next

' sets stickers 2 and 3

Dim index As Integer = 1

For i = 0 To UBound(cornerTriples, 2)

If i = argumentCornerIndex Then Continue For

\_corners(index) = cornerTriples(pairIndex, i)

index = 2

Next

End Sub

Private Sub SetUpCornerTriplets()

For i = 0 To 3

cornerTriples(i, 0).FaceNumber = 0

cornerTriples(i, 1).StickerNumber = 0

cornerTriples(i, 2).StickerNumber = 2

Next

cornerTriples(0, 0).StickerNumber = 0 : cornerTriples(0, 1).FaceNumber = 1 : cornerTriples(0, 2).FaceNumber = 2

cornerTriples(1, 0).StickerNumber = 2 : cornerTriples(1, 1).FaceNumber = 2 : cornerTriples(1, 2).FaceNumber = 3

cornerTriples(2, 0).StickerNumber = 6 : cornerTriples(2, 1).FaceNumber = 4 : cornerTriples(2, 2).FaceNumber = 1

cornerTriples(3, 0).StickerNumber = 8 : cornerTriples(3, 1).FaceNumber = 3 : cornerTriples(3, 2).FaceNumber = 4

End Sub

End Class

' A collection of 2 stickers that make up an edge cubie

Public Class EdgePair

Private ReadOnly edgePairs(,) As Sticker = {{New Sticker, New Sticker},

{New Sticker, New Sticker},

{New Sticker, New Sticker},

{New Sticker, New Sticker},

{New Sticker, New Sticker},

{New Sticker, New Sticker},

{New Sticker, New Sticker},

{New Sticker, New Sticker}}

Private \_edges() As Sticker = {New Sticker, New Sticker}

Public Property Edges(ByVal index As Integer) As Sticker

Get

Return \_edges(index)

End Get

Set(value As Sticker)

\_edges(index) = value

End Set

End Property

Public Sub New(ByVal stickerOne As Sticker)

If edgePairs(0, 1).StickerNumber = 0 Then SetUpEdgePairs()

\_edges(0) = stickerOne

Dim pairIndex, argumentEdgeIndex As Integer

' Gets the index for the pair containing the sticker passed as an argument

For j = 0 To UBound(edgePairs, 2)

For i = 0 To UBound(edgePairs, 1)

If edgePairs(i, j) <> \_edges(0) Then Continue For

pairIndex = i

argumentEdgeIndex = j

Next

Next

' Sets sticker 2

For i = 0 To UBound(edgePairs, 2)

If i = argumentEdgeIndex Then Continue For

\_edges(1) = edgePairs(pairIndex, i)

Next

End Sub

Private Sub SetUpEdgePairs()

For i = 0 To 3

edgePairs(i, 0).FaceNumber = 0

edgePairs(i, 1).StickerNumber = 1

Next

For i = 4 To 7

edgePairs(i, 0).StickerNumber = 3

edgePairs(i, 1).StickerNumber = 5

Next

edgePairs(0, 0).StickerNumber = 1 : edgePairs(0, 1).FaceNumber = 2

edgePairs(1, 0).StickerNumber = 3 : edgePairs(1, 1).FaceNumber = 1

edgePairs(2, 0).StickerNumber = 5 : edgePairs(2, 1).FaceNumber = 3

edgePairs(3, 0).StickerNumber = 7 : edgePairs(3, 1).FaceNumber = 4

edgePairs(4, 0).FaceNumber = 1 : edgePairs(4, 1).FaceNumber = 2

edgePairs(5, 0).FaceNumber = 2 : edgePairs(5, 1).FaceNumber = 3

edgePairs(6, 0).FaceNumber = 3 : edgePairs(6, 1).FaceNumber = 4

edgePairs(7, 0).FaceNumber = 4 : edgePairs(7, 1).FaceNumber = 1

End Sub

End Class

End Module

## SetCubieProperties.vb

Imports RubiksCubeSolver\_v2\_0.Helpers.PublicFunctions

Imports RubiksCubeSolver\_v2\_0.Helpers.PublicConstants

Public Module SetCubieProperties

''' <summary>

''' Returns the colourString for a cornerTriplet, and sets secondaryFaceColour and secondaryFaceRotation

''' </summary>

Public Function GetCornerColours(ByVal cube(,) As Char, ByVal cornerTri As CornerTriplet, ByRef secondaryFaceColour As FaceColour, ByRef secondaryFaceRotation As FaceColour) As String

Dim cornerColours(2) As Char

For cornerBlockSticker = 0 To 2

Dim colourStore As Char

colourStore = cube(cornerTri.Corners(cornerBlockSticker).FaceNumber,

cornerTri.Corners(cornerBlockSticker).StickerNumber)

If colourStore = "W" Or colourStore = "Y" Then

cornerColours(0) = colourStore

ElseIf colourStore = "R" Or colourStore = "O" Then

secondaryFaceColour = ColourChar2FaceNumber(colourStore)

secondaryFaceRotation = ColourChar2FaceNumber(cube(cornerTri.Corners(cornerBlockSticker).FaceNumber, 4))

cornerColours(1) = colourStore

ElseIf colourStore = "G" Or colourStore = "B" Then

cornerColours(2) = colourStore

Else

MsgBox("ERROR: " + cornerBlockSticker.ToString() + ", " + colourStore.ToString())

End If

Next cornerBlockSticker

Dim cornerString As String

cornerString = ""

For k = 0 To 2

cornerString += cornerColours(k)

Next k

Return cornerString

End Function

''' <summary> Returns the colourString for a edgePair </summary>

Public Function GetEdgeColours(ByVal cube(,) As Char, ByVal edgePair As EdgePair) As String

Dim edgeString As String = ""

Dim edgeColours(1) As Char

Dim colourStore, colourStore2 As Char

colourStore = cube(edgePair.Edges(0).FaceNumber, edgePair.Edges(0).StickerNumber)

colourStore2 = cube(edgePair.Edges(1).FaceNumber, edgePair.Edges(1).StickerNumber)

If (colourStore = "W" Or colourStore = "Y") Then

edgeColours(0) = colourStore

edgeColours(1) = colourStore2

ElseIf (colourStore2 = "W" Or colourStore2 = "Y") Then

edgeColours(0) = colourStore2

edgeColours(1) = colourStore

ElseIf (colourStore = "R" Or colourStore = "O") Then

edgeColours(0) = colourStore

edgeColours(1) = colourStore2

ElseIf (colourStore2 = "R" Or colourStore2 = "O") Then

edgeColours(0) = colourStore2

edgeColours(1) = colourStore

End If

For k = 0 To 1

edgeString += edgeColours(k)

Next k

Return edgeString

End Function

''' <summary>

''' Returns the position vector for a corner on the top half of the cube.

''' From the face and location of its primary face

''' </summary>

Public Function GetTopCornerPosition(ByVal face As Integer, ByVal sticker As Integer) As Vector3x1

Dim position As New Vector3x1

position.y = 1

Select Case face

Case MoveFaces.TOP

position.x = (sticker Mod 6) - 1

position.z = Math.Sign(sticker - 4)

Case MoveFaces.LEFT

position.x = -1

position.z = 1 - Math.Abs(sticker - 2)

Case MoveFaces.BACK

position.x = Math.Abs(sticker - 2) - 1

position.z = -1

Case MoveFaces.RIGHT

position.x = 1

position.z = 1 - sticker

Case MoveFaces.FRONT

position.x = sticker - 1

position.z = 1

End Select

Return position

End Function

''' <summary>

''' Returns the position vector for a corner on the bottom half of the cube.

''' From the face and location of its primary face

''' </summary>

Public Function GetBottomCornerPosition(ByVal face As Integer, ByVal sticker As Integer) As Vector3x1

Dim position As New Vector3x1

position.y = -1

Select Case face

Case MoveFaces.TOP

position.x = Math.Abs((sticker Mod 6) - 2) - 1

position.z = Math.Sign(sticker - 4)

Case MoveFaces.LEFT

position.x = 1

position.z = 1 - Math.Abs(sticker - 2)

Case MoveFaces.BACK

position.x = sticker - 1

position.z = -1

Case MoveFaces.RIGHT

position.x = -1

position.z = 1 - sticker

Case MoveFaces.FRONT

position.x = Math.Abs(sticker - 2) - 1

position.z = 1

End Select

Return position

End Function

''' <summary>

''' Returns the position vector for an edge on the top half of the cube.

''' From the face and location of its primary face

''' </summary>

Public Function GetTopEdgePosition(ByVal face As Integer, ByVal sticker As Integer) As Vector3x1

Dim position As New Vector3x1

Select Case face

Case MoveFaces.TOP

position.x = ((sticker Mod 6) Mod 3) - 1

position.y = 1

Select Case sticker

Case 1 : position.z = -1

Case 3 : position.z = 0

Case 5 : position.z = 0

Case 7 : position.z = 1

End Select

Case MoveFaces.LEFT

position.x = -1

position.y = Math.Sign(3 - sticker)

position.z = sticker Mod 3 - 1

Case MoveFaces.BACK

position.x = Math.Sign(sticker - 1)

position.y = Math.Sign(3 - sticker)

position.z = -1

Case MoveFaces.RIGHT

position.x = 1

position.y = Math.Sign(3 - sticker)

position.z = Math.Sign(sticker - 1)

Case MoveFaces.FRONT

position.x = sticker Mod 3 - 1

position.y = Math.Sign(3 - sticker)

position.z = 1

End Select

Return position

End Function

''' <summary>

''' Returns the position vector for an edge on the bottom half of the cube.

''' From the face and location of its primary face

''' </summary>

Public Function GetBottomEdgePosition(ByVal face As Integer, ByVal sticker As Integer) As Vector3x1

Dim position As New Vector3x1

Select Case face

Case 0 ' Top

position.x = -(((sticker Mod 3) Mod 6) - 1)

position.y = -1

Select Case sticker

Case 1 : position.z = -1

Case 3 : position.z = 0

Case 5 : position.z = 0

Case 7 : position.z = 1

End Select

Case 1 ' Left

position.x = 1

position.y = -(sticker Mod 3)

position.z = (sticker Mod 3) - 1

Case 2 ' Back

position.x = (sticker Mod 3) - 1

position.y = -(sticker Mod 3)

position.z = -1

Case 3 ' Right

position.x = -1

position.y = -(sticker Mod 3)

position.z = Math.Sign(sticker - 1)

Case 4 ' Front

position.x = Math.Sign(sticker - 1)

position.y = -(sticker Mod 3)

position.z = 1

End Select

Return position

End Function

''' <summary>

''' Orientates the stickerArray to match a given cubeOrientation

''' </summary>

Public Function OrientateStickerArray(ByVal stickerArray(,) As Char, ByVal cubeOrientation As CubeOrientation) As Char(,)

Dim transformedArray(4, 8) As Char

' each face can be in 1 of 4 rotations : 4 methods

Try

If cubeOrientation.Front = Opposite(cubeOrientation.Top) Then

Throw New ArgumentException("The specified top and front face combination is impossible." & vbNewLine &

"Top: " & cubeOrientation.Top & vbNewLine & "Front: " & cubeOrientation.Front)

End If

Catch ex As ArgumentException

MsgBox(ex.Message)

Return Nothing

End Try

Dim faceColours As FaceColour() = GetFaceColoursFromOrientation(cubeOrientation)

For sideFaceNumber = 0 To 4

Dim faceColour As FaceColour = faceColours(sideFaceNumber)

Dim faceStickerStore(9) As Char

For i = 0 To 8

faceStickerStore(i) = stickerArray(faceColour, i)

Next

transformedArray = MapFaceOntoTransformedArray(sideFaceNumber, faceColour, faceStickerStore,

cubeOrientation, transformedArray)

Next

Return transformedArray

End Function

Private Function MapFaceOntoTransformedArray(ByVal sideFaceNumber As Integer, ByVal faceColour As FaceColour, ByVal faceStickerStore() As Char, ByVal cubeOrientation As CubeOrientation, ByVal transformedArray(,) As Char) As Char(,)

Select Case cubeOrientation.Top

Case FaceColour.W

Select Case faceColour

Case FaceColour.O : UpsideDown(transformedArray, sideFaceNumber, faceStickerStore)

Case cubeOrientation.Top : TopFace(transformedArray, sideFaceNumber, faceStickerStore, cubeOrientation)

Case Else : NoRotation(transformedArray, sideFaceNumber, faceStickerStore)

End Select

Case FaceColour.Y

Select Case faceColour

Case FaceColour.O : NoRotation(transformedArray, sideFaceNumber, faceStickerStore)

Case cubeOrientation.Top : TopFace(transformedArray, sideFaceNumber, faceStickerStore, cubeOrientation)

Case Else : UpsideDown(transformedArray, sideFaceNumber, faceStickerStore)

End Select

Case FaceColour.G

Select Case faceColour

Case cubeOrientation.Top : TopFace(transformedArray, sideFaceNumber, faceStickerStore, cubeOrientation)

Case Else : RotateLeft(transformedArray, sideFaceNumber, faceStickerStore)

End Select

Case FaceColour.B

Select Case faceColour

Case cubeOrientation.Top : TopFace(transformedArray, sideFaceNumber, faceStickerStore, cubeOrientation)

Case Else : RotateRight(transformedArray, sideFaceNumber, faceStickerStore)

End Select

Case FaceColour.R

Select Case faceColour

Case FaceColour.Y : NoRotation(transformedArray, sideFaceNumber, faceStickerStore)

Case FaceColour.W : UpsideDown(transformedArray, sideFaceNumber, faceStickerStore)

Case FaceColour.B : RotateLeft(transformedArray, sideFaceNumber, faceStickerStore)

Case FaceColour.G : RotateRight(transformedArray, sideFaceNumber, faceStickerStore)

Case cubeOrientation.Top : TopFace(transformedArray, sideFaceNumber, faceStickerStore, cubeOrientation)

End Select

Case FaceColour.O

Select Case faceColour

Case FaceColour.Y : UpsideDown(transformedArray, sideFaceNumber, faceStickerStore)

Case FaceColour.W : NoRotation(transformedArray, sideFaceNumber, faceStickerStore)

Case FaceColour.B : RotateRight(transformedArray, sideFaceNumber, faceStickerStore)

Case FaceColour.G : RotateLeft(transformedArray, sideFaceNumber, faceStickerStore)

Case cubeOrientation.Top : TopFace(transformedArray, sideFaceNumber, faceStickerStore, cubeOrientation)

End Select

End Select

Return transformedArray

End Function

Private Sub NoRotation(ByRef stickerArray As Char(,), ByVal faceNumber As Integer, ByVal faceStickers As Char())

For i = 0 To 8

stickerArray(faceNumber, i) = faceStickers(i)

Next

End Sub

Private Sub UpsideDown(ByRef stickerArray As Char(,), ByVal faceNumber As Integer, ByVal faceStickers As Char())

For i = 0 To 8

stickerArray(faceNumber, i) = faceStickers(8 - i)

Next

End Sub

Private Sub RotateLeft(ByRef stickerArray As Char(,), ByVal faceNumber As Integer, ByVal faceStickers As Char())

Dim count As Integer = 6

For i = 0 To 8

stickerArray(faceNumber, i) = faceStickers(count Mod 10)

count += 7

Next

End Sub

Private Sub RotateRight(ByRef stickerArray As Char(,), ByVal faceNumber As Integer, ByVal faceStickers As Char())

Dim count As Integer = 2

For i = 0 To 8

stickerArray(faceNumber, i) = faceStickers(count Mod 10)

count += 3

Next

End Sub

Private Sub TopFace(ByRef stickerArray As Char(,), ByVal faceNumber As Integer, ByVal faceStickers As Char(), ByVal currentOrientation As CubeOrientation)

If currentOrientation.Front = FaceColour.B Then

RotateLeft(stickerArray, faceNumber, faceStickers)

ElseIf currentOrientation.Front = FaceColour.G Then

RotateRight(stickerArray, faceNumber, faceStickers)

ElseIf currentOrientation.Front = FaceColour.W Then

If currentOrientation.Top = FaceColour.O Then

NoRotation(stickerArray, faceNumber, faceStickers)

Else

UpsideDown(stickerArray, faceNumber, faceStickers)

End If

ElseIf currentOrientation.Front = FaceColour.Y Then

If currentOrientation.Top = FaceColour.O Then

UpsideDown(stickerArray, faceNumber, faceStickers)

Else

NoRotation(stickerArray, faceNumber, faceStickers)

End If

ElseIf currentOrientation.Front = FaceColour.R Then

Select Case currentOrientation.Top

Case FaceColour.W : NoRotation(stickerArray, faceNumber, faceStickers)

Case FaceColour.Y : UpsideDown(stickerArray, faceNumber, faceStickers)

Case FaceColour.G : RotateLeft(stickerArray, faceNumber, faceStickers)

Case FaceColour.B : RotateRight(stickerArray, faceNumber, faceStickers)

End Select

ElseIf currentOrientation.Front = FaceColour.O Then

Select Case currentOrientation.Top

Case FaceColour.W : UpsideDown(stickerArray, faceNumber, faceStickers)

Case FaceColour.Y : NoRotation(stickerArray, faceNumber, faceStickers)

Case FaceColour.G : RotateRight(stickerArray, faceNumber, faceStickers)

Case FaceColour.B : RotateLeft(stickerArray, faceNumber, faceStickers)

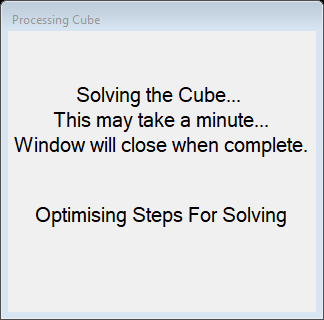
End Select

End If

End Sub

End Module

## Processing.vb



Public Class Processing

Private \_cube As Cube

Public Sub New(ByRef cube As Cube)

InitializeComponent()

Me.\_cube = cube

End Sub

Private Sub Processing\_Shown(sender As Object, e As EventArgs) Handles MyBase.Shown

lblText.Visible = True

lblText.Refresh()

'copy the initial cube before solving it

Dim scrambledCorners() As Corner = Copy(\_cube.Corners)

Dim scrambledEdges() As Edge = Copy(\_cube.Edges)

\_cube.Solve(Me)

Dim outputForm As New \_3DOutput(New Cube(scrambledCorners, scrambledEdges, \_cube.Instructions))

outputForm.Show()

Me.Close()

End Sub

End Class

## Cube.vb

Imports RubiksCubeSolver\_v2\_0.Helpers

Imports System.Runtime.Serialization.Formatters.Binary

<Serializable()> Public Class Cube

Public Property CurrentOrientation As CubeOrientation

Public Property Corners() As Corner()

Public Property Edges() As Edge()

Public Property Middles() As Middle()

Public Property Instructions As InstructionList

Public Property TopFace As FaceColour

Get

Return CurrentOrientation.Top

End Get

Set(value As FaceColour)

CurrentOrientation.Top = value

End Set

End Property

Public ReadOnly Property BottomFace As FaceColour

Get

Return CurrentOrientation.Bottom

End Get

End Property

Public Property FrontFace As FaceColour

Get

Return CurrentOrientation.Front

End Get

Set(value As FaceColour)

CurrentOrientation.Front = value

End Set

End Property

Public ReadOnly Property CornersAndEdgesAndMiddles As Block()

Get

If Me.Corners.Length < 1 Or Me.Edges.Length < 1 Or Me.Middles.Length < 1 Then Return Nothing

Dim \_cornersAndEdgesandmiddles(26) As Block

For i = 0 To 7

\_cornersAndEdgesandmiddles(i) = Corners(i)

Next

For i = 0 To 11

\_cornersAndEdgesandmiddles(i + 8) = Edges(i)

Next

For i = 0 To 6

\_cornersAndEdgesandmiddles(i + 20) = Middles(i)

Next

Return \_cornersAndEdgesandmiddles

End Get

End Property

ReadOnly Property CorrectTopEdges() As Edge()

Get

Dim rightCubies(3) As Edge

Dim index As Integer = 0

Dim topEdges() As Edge = TopEdgesClockwise()

For Each cubie In topEdges

If cubie.CorrectForFace(TopFace, Me) Then

rightCubies(index) = cubie

index += 1

End If

Next

ReDim Preserve rightCubies(index - 1)

Return rightCubies

End Get

End Property

ReadOnly Property IncorrectTopEdges() As Edge()

Get

Dim wrongCubies(3) As Edge

Dim index As Integer = 0

Dim topEdges() As Edge = TopEdgesClockwise()

For Each cubie In topEdges

If Not cubie.CorrectForFace(TopFace, Me) Then

wrongCubies(index) = cubie

index += 1

End If

Next

ReDim Preserve wrongCubies(index - 1)

Return wrongCubies

End Get

End Property

ReadOnly Property CorrectTopCorners() As Corner()

Get

Dim rightCubies(3) As Corner

Dim index As Integer = 0

Dim topEdges() As Corner = TopCornersClockwise()

For Each cubie In topEdges

If cubie.CorrectForFace(TopFace, Me) Then

rightCubies(index) = cubie

index += 1

End If

Next

ReDim Preserve rightCubies(index - 1)

Return rightCubies

End Get

End Property

ReadOnly Property IncorrectTopCorners() As Corner()

Get

Dim wrongCubies(3) As Corner

Dim index As Integer = 0

Dim topEdges() As Corner = TopCornersClockwise()

For Each cubie In topEdges

If Not cubie.CorrectForFace(TopFace, Me) Then

wrongCubies(index) = cubie

index += 1

End If

Next

ReDim Preserve wrongCubies(index - 1)

Return wrongCubies

End Get

End Property

''' <summary>

''' Returns an array of cubies with the correct colour stickers

''' </summary>

Public ReadOnly Property BlocksByColour(ByVal colour As FaceColour) As Block()

Get

Dim cubies(0 To 8) As Block

Dim count As Integer = 0

For Each block In CornersAndEdgesAndMiddles

If Not block.HasColour(colour) Then Continue For

cubies(count) = block

count += 1

Next

Return cubies

End Get

End Property

Public ReadOnly Property TopEdgesClockwise() As Edge()

Get

Return TopBlocksClockwise().Extract(Of Edge)()

End Get

End Property

Public ReadOnly Property TopCornersClockwise() As Corner()

Get

Return TopBlocksClockwise().Extract(Of Corner)()

End Get

End Property

Public ReadOnly Property TopBlocksClockwise() As Block()

Get

Dim topBlocks(8) As Block

For Each block In CornersAndEdgesAndMiddles

If block.Position.y <> 1 Then Continue For

Dim index As Integer

If block.GetType() = GetType(Edge) Then

index = TopEdgePositionToClockwiseIndex(block.Position)

ElseIf block.GetType() = GetType(Corner) Then

index = TopCornerPositionToClockwiseIndex(block.Position)

ElseIf block.GetType() = GetType(Middle) Then

index = 8

End If

topBlocks(index) = block

Next

Return topBlocks

End Get

End Property

''' <summary>

''' Converts a position vector to an index in an 8-element array to store the blocks on the top face

''' </summary>

Private Function TopCornerPositionToClockwiseIndex(pos As Vector3x1) As Integer

Dim index As Integer

Select Case (pos.x + pos.z)

Case -2 'Back left corner

index = 2

Case 0

If pos.x = -1 Then ' Front left corner

index = 0

ElseIf pos.x = 1 Then ' Back right corner

index = 4

End If

Case 2 ' Front right corner

index = 6

End Select

Return index

End Function

''' <summary>

''' Converts a position vector to an index in an 8-element array to store the blocks on the top face

''' </summary>

Private Function TopEdgePositionToClockwiseIndex(pos As Vector3x1) As Integer

Dim index As Integer

Select Case pos.x

Case -1 ' left edge cubie

index = 1

Case 0

If pos.z = -1 Then ' back edge cubie

index = 3

Else ' front edge cubie

index = 7

End If

Case 1 ' right edge cubie

index = 5

End Select

Return index

End Function

Public ReadOnly Property MiddleEdgesClockwise() As Edge()

Get

Dim middleEdges(0 To 3) As Edge

Dim index As Integer

For Each edge In Edges

If edge.Layer <> Layer.MIDDLE Then Continue For

Select Case (edge.Position.x + edge.Position.z)

Case -2 'Back left edge

index = 1

Case 0

If edge.Position.x = -1 Then ' Front left edge

index = 0

ElseIf edge.Position.x = 1 Then ' Back right edge

index = 2

End If

Case 2 ' Front right edge

index = 3

End Select

middleEdges(index) = edge

Next

Return middleEdges

End Get

End Property

ReadOnly Property FaceColours As FaceColour()

Get

Return GetFaceColoursFromOrientation(CurrentOrientation)

End Get

End Property

Public Sub New(ByVal cube As Cube)

Me.CurrentOrientation = cube.CurrentOrientation.Copy()

Me.Corners = cube.Corners.Copy()

Me.Edges = cube.Edges.Copy()

CreateMiddles()

Me.Instructions = cube.Instructions.Copy()

End Sub

Public Sub New(ByVal corners() As Corner, ByVal edges() As Edge, ByVal instructions As InstructionList)

Me.Corners = corners

Me.Edges = edges

CreateMiddles()

Me.Instructions = instructions

End Sub

Public Sub New(ByVal stickers(,) As Char)

CurrentOrientation = New CubeOrientation(CChar("W"), CChar("R"))

Instructions = New InstructionList

InitialiseBlocks()

Dim originalData(,) As Char = Copy(stickers)

'Converts top half of cube

Dim transformedData(,) As Char = OrientateStickerArray(originalData, CurrentOrientation)

ConvertTopCorners(transformedData)

ConvertTopEdges(transformedData)

'Converts bottom half of cube

transformedData = OrientateStickerArray(originalData, CurrentOrientation.UpsideDown)

ConvertTopCorners(transformedData)

ConvertTopEdges(transformedData)

CreateMiddles()

End Sub

Private Sub InitialiseBlocks()

FillNameArrays()

ReDim Corners(7)

ReDim Edges(11)

For i = 0 To 7

Me.Corners(i) = New Corner With {.Name = CORNER\_NAMES(i)}

Next

For i = 0 To 11

Me.Edges(i) = New Edge With {.Name = EDGE\_NAMES(i)}

Next

End Sub

Private Sub CreateMiddles()

ReDim Middles(6)

For i = 0 To 6

Me.Middles(i) = New Middle

Next

Middles(0).Name = "W"

Middles(1).Name = "G"

Middles(2).Name = "O"

Middles(3).Name = "B"

Middles(4).Name = "R"

Middles(5).Name = "Y"

Middles(6).Name = ""

For i = 0 To 5

Middles(i).Colours(0) = Helpers.ColourChar2FaceNumber(CChar(Middles(i).Name))

Middles(i).Rotation = Middles(i).PrimaryFace

Next

Middles(0).Position = New Vector3x1(0, 1, 0)

Middles(1).Position = New Vector3x1(-1, 0, 0)

Middles(2).Position = New Vector3x1(0, 0, -1)

Middles(3).Position = New Vector3x1(1, 0, 0)

Middles(4).Position = New Vector3x1(0, 0, 1)

Middles(5).Position = New Vector3x1(0, -1, 0)

Middles(6).Position = New Vector3x1(0, 0, 0)

End Sub

Private Sub ConvertTopCorners(ByVal stickers(,) As Char)

Dim convertingTopLayer As Boolean

convertingTopLayer = (stickers(0, 4) = FaceNumber2ColourChar(CurrentOrientation.Top))

For faceNumber = MoveFaces.TOP To MoveFaces.FRONT

For cornerStickerNumber = 0 To 8 Step 2

Dim cornerSticker As New Sticker(faceNumber, cornerStickerNumber)

Dim stickerColour As Char = stickers(faceNumber, cornerStickerNumber)

If Not (OnTopHalfOfCube(cornerSticker) And IsPrimaryCornerSticker(stickerColour)) Then Continue For

Dim cornerTri As New CornerTriplet(cornerSticker)

Dim secondaryFaceColour, secondaryFaceRotation As FaceColour

'Gets the colour string for the cubie e.g. "YRG" and also gets the secondary face colour and rotation

Dim cornerString As String = GetCornerColours(stickers, cornerTri, secondaryFaceColour,

secondaryFaceRotation)

For cornerNumber = 0 To UBound(CORNER\_NAMES)

If CORNER\_NAMES(cornerNumber) <> cornerString Then Continue For

If convertingTopLayer Then

Corners(cornerNumber).Position = GetTopCornerPosition(faceNumber, cornerStickerNumber)

Else

Corners(cornerNumber).Position = GetBottomCornerPosition(faceNumber, cornerStickerNumber)

End If

Corners(cornerNumber).Rotation = ColourChar2FaceNumber(stickers(cornerTri.Corners(0).FaceNumber, 4))

Corners(cornerNumber).SecondaryFace = secondaryFaceColour

Corners(cornerNumber).SecondaryRotation = secondaryFaceRotation

Corners(cornerNumber).SetColoursFromColourString(cornerString)

Next cornerNumber

Next cornerStickerNumber

Next faceNumber

End Sub

Private Sub ConvertTopEdges(ByVal stickers(,) As Char)

Dim convertingTopLayer As Boolean

convertingTopLayer = (stickers(0, 4) = FaceNumber2ColourChar(CurrentOrientation.Top))

For faceNumber = MoveFaces.TOP To MoveFaces.FRONT

For edgeStickerNumber = 1 To 7 Step 2

Dim edgeSticker As New Sticker(faceNumber, edgeStickerNumber)

Dim stickerColour As Char = stickers(faceNumber, edgeStickerNumber)

Dim edgePair As New EdgePair(edgeSticker)

Dim secondStickerColour As Char = stickers(edgePair.Edges(1).FaceNumber, edgePair.Edges(1).StickerNumber)

If Not (OnTopHalfOfCube(edgeSticker) And

IsPrimaryEdgeSticker(stickerColour, secondStickerColour)) Then Continue For 'try next sticker

Dim edgeString As String = GetEdgeColours(stickers, edgePair)

For edgeNumber = 0 To UBound(EDGE\_NAMES)

If EDGE\_NAMES(edgeNumber) <> edgeString Then Continue For

If convertingTopLayer Then

Edges(edgeNumber).Position = GetTopEdgePosition(faceNumber, edgeStickerNumber)

Else

Edges(edgeNumber).Position = GetBottomEdgePosition(faceNumber, edgeStickerNumber)

End If

Edges(edgeNumber).Rotation = ColourChar2FaceNumber(stickers(edgePair.Edges(0).FaceNumber, 4))

Edges(edgeNumber).SetColoursFromColourString(edgeString)

Next edgeNumber

Next edgeStickerNumber

Next faceNumber

End Sub

Private Function OnTopHalfOfCube(ByVal position As Sticker) As Boolean

Return position.StickerNumber <> MIDDLE\_STICKER And

(position.FaceNumber = MoveFaces.TOP Or position.StickerNumber < MIDDLE\_STICKER)

End Function

Private Function IsPrimaryCornerSticker(ByVal stickerColour As Char) As Boolean

Return stickerColour = "W" Or stickerColour = "Y"

End Function

Private Function IsPrimaryEdgeSticker(ByVal stickerColour As Char, ByVal secondStickerColour As Char) As Boolean

Return ((stickerColour = "W" Or stickerColour = "Y") Or

((stickerColour = "R" Or stickerColour = "O") And (secondStickerColour <> "W" And secondStickerColour <> "Y")))

End Function

Public Sub Solve(ByRef displayForm As Processing)

If Complete() Then Return

Try

displayForm.lblStatus.Text = "Solving Top Layer"

displayForm.lblStatus.Refresh()

SolveTopLayer()

displayForm.lblStatus.Text = "Solving Middle Layer"

displayForm.lblStatus.Refresh()

SolveMiddleLayer()

displayForm.lblStatus.Text = "Solving Bottom Layer"

displayForm.lblStatus.Refresh()

SolveBottomLayer()

If Not Complete() Then Throw New StageNotSuccessfulException()

Catch ex As StageNotSuccessfulException

Console.WriteLine(ex.Message)

MsgBox("Your cube cannot be solved, this is likely because the stickers have been re-arranged, or one of the pieces has been removed and re-inserted in the wrong orientation. You will need to take the cube apart and re-assemble it in its solved state. If you want you can go throught the instructions the program generated, which will not solve it but may bring it closer to being solved.")

End Try

displayForm.lblStatus.Text = "Optimising Steps For Solving"

displayForm.lblStatus.Refresh()

Instructions.Optimise()

End Sub

Private Sub SolveTopLayer()

Dim topLayerSolver As New TopLayerSolver(Me)

topLayerSolver.Solve()

End Sub

Private Sub SolveMiddleLayer()

Dim middleLayerSolver As New MiddleLayerSolver(Me)

middleLayerSolver.Solve()

End Sub

Private Sub SolveBottomLayer()

Dim bottomLayerSolver As New BottomLayerSolver(Me)

bottomLayerSolver.Solve()

End Sub

Public ReadOnly Property Complete() As Boolean

Get

For face As FaceColour = Helpers.FaceColour.W To Helpers.FaceColour.O

Dim faceBlocks As Block() = BlocksByColour(face)

For Each block In faceBlocks

If Not block.Correct(Me) Then Return False

Next

Next

Return True

End Get

End Property

''' <summary> Rotates the whole cube </summary>

Public Sub Rotate(ByVal direction As Direction, ByVal axis As Axis)

If direction = Direction.NO\_CHANGE Then Return

'used for undoing half-turns when ouputting

If direction = -2 Then direction = Direction.HALF\_TURN

Dim angle As Double = (Math.PI / 2.0) \* direction

Dim rotationMatrix As Matrix3x3 = GetRotationMatrix(axis, angle)

Instructions.AddOrientationChange(axis, direction)

For Each cubie In CornersAndEdgesAndMiddles

cubie.Position = rotationMatrix \* cubie.Position

Next

UpdateOrientation(direction, axis)

End Sub

Private Sub UpdateOrientation(ByVal direction As Direction, ByVal axis As Axis)

Dim faces() As FaceColour = FacesAroundAxisClockwise(axis)

If axis <> Axis.Y Then TopFace = faces((Array.IndexOf(faces, TopFace) + 4 + direction) Mod 4)

If axis <> Axis.Z Then FrontFace = faces((Array.IndexOf(faces, FrontFace) + 4 + direction) Mod 4)

End Sub

Private Function FacesAroundAxisClockwise(ByVal axis As Axis) As FaceColour()

Select Case axis

Case Axis.X

Return {FaceColours(MoveFaces.FRONT), FaceColours(MoveFaces.TOP), FaceColours(MoveFaces.BACK),

FaceColours(MoveFaces.BOTTOM)}

Case Axis.Y

Return {FaceColours(MoveFaces.FRONT), FaceColours(MoveFaces.LEFT), FaceColours(MoveFaces.BACK),

FaceColours(MoveFaces.RIGHT)}

Case Axis.Z

Return {FaceColours(MoveFaces.TOP), FaceColours(MoveFaces.RIGHT), FaceColours(MoveFaces.BOTTOM),

FaceColours(MoveFaces.LEFT)}

Case Else

Throw New ArgumentException("Invalid Axis")

End Select

End Function

''' <summary> Rotates the cube so that a specific face is on top </summary>

Public Sub RotateFaceToTop(ByVal newTop As Helpers.FaceColour)

Dim faces() As FaceColour = FaceColours

Select Case newTop

Case faces(MoveFaces.TOP)

Return

Case faces(MoveFaces.LEFT)

Me.Rotate(Direction.CLOCKWISE, Axis.Z)

Case faces(MoveFaces.BACK)

Me.Rotate(Direction.ANTICLOCKWISE, Axis.X)

FrontFace = faces(MoveFaces.TOP)

Case faces(MoveFaces.RIGHT)

Me.Rotate(Direction.ANTICLOCKWISE, Axis.Z)

Case faces(MoveFaces.FRONT)

Me.Rotate(Direction.CLOCKWISE, Axis.X)

FrontFace = faces(MoveFaces.BOTTOM)

Case faces(MoveFaces.BOTTOM)

Me.Rotate(Direction.HALF\_TURN, Axis.Z)

End Select

End Sub

''' <summary> Rotates the cube so that a specific face is at the front </summary>

Public Sub RotateFaceToFront(ByVal newFront As Helpers.FaceColour)

Dim faces() As FaceColour = FaceColours

Select Case newFront

Case faces(MoveFaces.TOP)

Me.Rotate(Direction.ANTICLOCKWISE, Axis.X)

TopFace = faces(MoveFaces.BACK)

Case faces(MoveFaces.LEFT)

Me.Rotate(Direction.ANTICLOCKWISE, Axis.Y)

Case faces(MoveFaces.BACK)

Me.Rotate(Direction.HALF\_TURN, Axis.Y)

Case faces(MoveFaces.RIGHT)

Me.Rotate(Direction.CLOCKWISE, Axis.Y)

Case faces(MoveFaces.FRONT)

Return

Case faces(MoveFaces.BOTTOM)

Me.Rotate(Direction.CLOCKWISE, Axis.X)

TopFace = faces(MoveFaces.FRONT)

End Select

End Sub

''' <summary> Rotates the cube so that a specific face is on the left </summary>

Public Sub RotateFaceToLeft(ByVal newLeft As Helpers.FaceColour)

Dim faces() As FaceColour = FaceColours

Select Case newLeft

Case faces(MoveFaces.TOP)

Me.Rotate(Direction.ANTICLOCKWISE, Axis.Z)

Case faces(MoveFaces.LEFT)

Return

Case faces(MoveFaces.BACK)

Me.Rotate(Direction.ANTICLOCKWISE, Axis.Y)

Case faces(MoveFaces.RIGHT)

Me.Rotate(Direction.HALF\_TURN, Axis.Y)

Case faces(MoveFaces.FRONT)

Me.Rotate(Direction.CLOCKWISE, Axis.Y)

Case faces(MoveFaces.BOTTOM)

Me.Rotate(Direction.CLOCKWISE, Axis.Z)

End Select

End Sub

Public Sub RotateFace(ByVal face As FaceColour, ByVal direction As Direction)

If direction = Direction.NO\_CHANGE Then Return

If direction = -2 Then direction = Direction.HALF\_TURN

Dim mathematicalDirection As Direction = direction

Dim axis As Axis

Select Case face

Case FaceColours(MoveFaces.TOP)

axis = Axis.Y

Case FaceColours(MoveFaces.LEFT)

axis = Axis.X

If direction <> Direction.HALF\_TURN Then mathematicalDirection = -mathematicalDirection

Case FaceColours(MoveFaces.BACK)

axis = Axis.Z

If direction <> Direction.HALF\_TURN Then mathematicalDirection = -mathematicalDirection

Case FaceColours(MoveFaces.RIGHT)

axis = Axis.X

Case FaceColours(MoveFaces.FRONT)

axis = Axis.Z

Case FaceColours(MoveFaces.BOTTOM)

axis = Axis.Y

If direction <> Direction.HALF\_TURN Then mathematicalDirection = -mathematicalDirection

End Select

Dim angle As Double = (Math.PI / 2.0) \* mathematicalDirection

Dim rotationMatrix As Matrix3x3 = GetRotationMatrix(axis, angle)

Instructions.AddFaceTurn(Array.IndexOf(FaceColours, face), direction)

Dim faceColoursAsIfRotatedFaceIsTop() As FaceColour

If face <> FrontFace And face <> Opposite(FrontFace) Then

faceColoursAsIfRotatedFaceIsTop = Helpers.GetFaceColoursFromOrientation(New CubeOrientation(face, FrontFace))

ElseIf face = FrontFace Then

faceColoursAsIfRotatedFaceIsTop = Helpers.GetFaceColoursFromOrientation(New CubeOrientation(face, BottomFace))

ElseIf face = Opposite(FrontFace) Then

faceColoursAsIfRotatedFaceIsTop = Helpers.GetFaceColoursFromOrientation(New CubeOrientation(face, TopFace))

End If

For Each cubie In CornersAndEdgesAndMiddles

If Not cubie.OnFace(face, Me) Then Continue For

cubie.Position = rotationMatrix \* cubie.Position

cubie.Rotation = GetNewRotation(direction, cubie.Rotation, faceColoursAsIfRotatedFaceIsTop)

If cubie.GetType() = GetType(Corner) Then DirectCast(cubie, Corner).SecondaryRotation =

GetNewRotation(direction, DirectCast(cubie, Corner).SecondaryRotation, faceColoursAsIfRotatedFaceIsTop)

Next

End Sub

''' <summary> Gets the new rotation for a cubie from a given rotation </summary>

Private Function GetNewRotation(ByVal direction As Direction, ByVal currentRotation As FaceColour, ByVal faceColoursAsIfRotatedFaceIsTop() As FaceColour) As FaceColour

Dim currentFaceIndex As Integer = Array.IndexOf(faceColoursAsIfRotatedFaceIsTop, currentRotation)

If currentFaceIndex = 0 Then Return currentRotation

If currentFaceIndex = 5 Then Throw New ArgumentException("The cube is not on the face being rotated")

Select Case direction

Case Direction.CLOCKWISE

Return faceColoursAsIfRotatedFaceIsTop((currentFaceIndex Mod 4) + 1)

Case Direction.ANTICLOCKWISE

Return faceColoursAsIfRotatedFaceIsTop(((currentFaceIndex + 2) Mod 4) + 1)

Case Direction.HALF\_TURN

Return faceColoursAsIfRotatedFaceIsTop(((currentFaceIndex + 1) Mod 4) + 1)

Case Else

Return currentRotation

End Select

End Function

Public Sub Save(ByVal filePath As String)

If filePath.Substring(filePath.Length - 5) <> ".cube" Then filePath += ".cube"

Try

Dim fStream As New IO.FileStream(filePath, IO.FileMode.Create)

WriteCubeToFile(fStream)

fStream.Close()

Catch ex As Exception

MsgBox("Unable to save file, please try a different fileName" & vbNewLine & ex.Message)

Throw New WriteUnsuccessfulException

End Try

End Sub

Private Sub WriteCubeToFile(ByRef fStream As IO.FileStream)

Dim formatter As New BinaryFormatter()

formatter.Serialize(fStream, Me)

End Sub

End Class

## Helpers.vb

Namespace Helpers

Public Module PublicConstants

Public Const STICKER\_LETTERS\_NO\_MIDDLE As String = "abcdfghi"

Public Const STICKER\_LETTERS As String = "abcdefghi"

Public Const FACE\_COLOURS As String = "bgorwy"

Public Const MIDDLE\_STICKER As Integer = 4

Public ReadOnly CORNER\_NAMES(0 To 7) As String

Public ReadOnly EDGE\_NAMES(0 To 11) As String

Public Sub FillNameArrays()

FillCornerNames()

FillEdgeNames()

End Sub

Private Sub FillCornerNames()

CORNER\_NAMES(0) = "WRB"

CORNER\_NAMES(1) = "WOB"

CORNER\_NAMES(2) = "WOG"

CORNER\_NAMES(3) = "WRG"

CORNER\_NAMES(4) = "YRB"

CORNER\_NAMES(5) = "YRG"

CORNER\_NAMES(6) = "YOG"

CORNER\_NAMES(7) = "YOB"

End Sub

Private Sub FillEdgeNames()

EDGE\_NAMES(0) = "WR"

EDGE\_NAMES(1) = "WB"

EDGE\_NAMES(2) = "WO"

EDGE\_NAMES(3) = "WG"

EDGE\_NAMES(4) = "RB"

EDGE\_NAMES(5) = "OB"

EDGE\_NAMES(6) = "OG"

EDGE\_NAMES(7) = "RG"

EDGE\_NAMES(8) = "YR"

EDGE\_NAMES(9) = "YG"

EDGE\_NAMES(10) = "YO"

EDGE\_NAMES(11) = "YB"

End Sub

Public Enum FaceColour

W

G

R

B

Y

O

None

End Enum

Public Enum Axis

X

Y

Z

End Enum

Public Enum MoveFaces

TOP

LEFT

BACK

RIGHT

FRONT

BOTTOM

End Enum

Public Enum Direction

CLOCKWISE = -1

NO\_CHANGE = 0

ANTICLOCKWISE = 1

HALF\_TURN = 2

End Enum

Public Enum Layer

TOP = 1

MIDDLE = 0

BOTTOM = -1

End Enum

End Module

Module PublicFunctions

''' <summary> Converts colour char to the name of the colour </summary>

Public Function ColourCharToWord(ByVal colourchar As Char) As String

Static colours As New Dictionary(Of Char, String)

If colours.Count = 0 Then colours = GetColourDictionary()

Try

Return colours.Item(colourchar)

Catch ex As KeyNotFoundException

Return "Error"

End Try

End Function

Private Function GetColourDictionary() As Dictionary(Of Char, String)

'uses a dictionary of character keys to colour strings for faster lookup

Dim colours As New Dictionary(Of Char, String) From {

{"W", "White"},

{"R", "Red"},

{"B", "Blue"},

{"Y", "Yellow"},

{"O", "Orange"},

{"G", "Green"}

}

Return colours

End Function

''' <summary> Converts colour char to face number </summary>

Public Function ColourChar2FaceNumber(ByVal colourchar As Char) As FaceColour

Return [Enum].Parse(GetType(FaceColour), colourchar, True)

End Function

Public Function ColourChar2FaceNumber(ByVal colourChars() As Char) As FaceColour()

Dim array(colourChars.Length - 1) As FaceColour

For i = 0 To colourChars.Length - 1

array(i) = ColourChar2FaceNumber(colourChars(i))

Next

Return array

End Function

''' <summary> Converts face number to colour char </summary>

Public Function FaceNumber2ColourChar(ByVal number As Integer) As Char

Dim color As FaceColour = number

Return color.ToString()

End Function

Public Sub WriteBlocksToFile(ByVal blocks As Block(), Optional filename As String = "Blocks.txt")

FileOpen(1, filename, OpenMode.Output)

For Each block In blocks

PrintLine(1, block.ToString())

Next

FileClose(1)

End Sub

''' <summary> Returns the opposte coloured face to the colour given as an argument </summary>

Public Function Opposite(ByVal colour As Char) As Char

Select Case colour

Case "W"

Return "Y"

Case "R"

Return "O"

Case "B"

Return "G"

Case "Y"

Return "W"

Case "O"

Return "R"

Case "G"

Return "B"

Case Else

Return "-"

End Select

End Function

Public Function Opposite(ByVal colour As FaceColour) As FaceColour

Return ColourChar2FaceNumber(Opposite(FaceNumber2ColourChar(colour)))

End Function

' (0 = top, 1 = left, 2 = back, 3 = right, 4 = front 5 = bottom)

Public Function GetFaceColoursFromOrientation(ByVal currentOrientation As CubeOrientation) As FaceColour()

Dim faces(0 To 5) As FaceColour

faces(0) = currentOrientation.Top

faces(5) = currentOrientation.Bottom

Dim sideFaceColoursClockwise() As FaceColour = GetSideFaceColours(currentOrientation.Top)

Dim positionOfFront As Integer

positionOfFront = LinearSearch(sideFaceColoursClockwise, currentOrientation.Front)

For i = 1 To 4

faces(i) = sideFaceColoursClockwise((i + positionOfFront) Mod 4)

Next

Return faces

End Function

Private Function GetSideFaceColours(ByVal top As FaceColour) As FaceColour()

Dim sideFaceColoursClockwise(3) As FaceColour

Select Case top

Case FaceColour.W

sideFaceColoursClockwise(0) = FaceColour.R

sideFaceColoursClockwise(1) = FaceColour.G

sideFaceColoursClockwise(2) = FaceColour.O

sideFaceColoursClockwise(3) = FaceColour.B

Case FaceColour.Y

sideFaceColoursClockwise(3) = FaceColour.R

sideFaceColoursClockwise(2) = FaceColour.G

sideFaceColoursClockwise(1) = FaceColour.O

sideFaceColoursClockwise(0) = FaceColour.B

Case FaceColour.R

sideFaceColoursClockwise(0) = FaceColour.B

sideFaceColoursClockwise(1) = FaceColour.Y

sideFaceColoursClockwise(2) = FaceColour.G

sideFaceColoursClockwise(3) = FaceColour.W

Case FaceColour.O

sideFaceColoursClockwise(3) = FaceColour.B

sideFaceColoursClockwise(2) = FaceColour.Y

sideFaceColoursClockwise(1) = FaceColour.G

sideFaceColoursClockwise(0) = FaceColour.W

Case FaceColour.B

sideFaceColoursClockwise(0) = FaceColour.Y

sideFaceColoursClockwise(1) = FaceColour.R

sideFaceColoursClockwise(2) = FaceColour.W

sideFaceColoursClockwise(3) = FaceColour.O

Case FaceColour.G

sideFaceColoursClockwise(3) = FaceColour.Y

sideFaceColoursClockwise(2) = FaceColour.R

sideFaceColoursClockwise(1) = FaceColour.W

sideFaceColoursClockwise(0) = FaceColour.O

End Select

Return sideFaceColoursClockwise

End Function

End Module End Namespace

## Block.vb

Imports RubiksCubeSolver\_v2\_0.Helpers

<Serializable()> Public MustInherit Class Block

Private \_name As String

Public Property Name() As String

Get

Return \_name

End Get

Set(ByVal value As String)

\_name = value

End Set

End Property

Private \_position As New Vector3x1()

Public Property Position As Vector3x1

Get

Return \_position

End Get

Set(value As Vector3x1)

\_position = New Vector3x1(value)

End Set

End Property

Public ReadOnly Property Layer As Layer

Get

Return Position.y

End Get

End Property

''' <summary> The colour of the face that the primary face is on </summary>

Private \_rotation As FaceColour

Public Property Rotation() As FaceColour

Get

Return \_rotation

End Get

Set(value As FaceColour)

\_rotation = value

End Set

End Property

Private \_colours() As FaceColour = {FaceColour.None, FaceColour.None, FaceColour.None}

Public Property Colours(ByVal index As Integer) As FaceColour

Get

Return \_colours(index)

End Get

Set(value As FaceColour)

\_colours(index) = value

End Set

End Property

Public Property Colours() As FaceColour()

Get

Return \_colours

End Get

Set(value As FaceColour())

\_colours = value

End Set

End Property

Public MustOverride Sub SetColoursFromColourString(ByVal colourString As String)

Public ReadOnly Property PrimaryFace() As FaceColour

Get

Return \_colours(0)

End Get

End Property

Public MustOverride Overrides Function ToString() As String

Public MustOverride Function HasColour(ByVal colour As FaceColour) As Boolean

Public Function OnFace(ByVal face As FaceColour, ByVal cube As Cube) As Boolean

Dim faces As FaceColour() = cube.FaceColours

Return face = faces(MoveFaces.TOP) And Position.y = 1 Or

face = faces(MoveFaces.BOTTOM) And Position.y = -1 Or

face = faces(MoveFaces.FRONT) And Position.z = 1 Or

face = faces(MoveFaces.BACK) And Position.z = -1 Or

face = faces(MoveFaces.LEFT) And Position.x = -1 Or

face = faces(MoveFaces.RIGHT) And Position.x = 1

End Function

''' <summary>

''' Checks if the orientation of a given face of a cube matches a given face colour

''' </summary>

Public MustOverride Function CorrectRotation(ByVal cubieFaceColour As FaceColour, ByVal cubeFaceColour As FaceColour) As Boolean

''' <summary>

''' Checks if a cubie is correct for the given face

''' e.g. if face = blue, checks that the cubie is on the blue face,

''' and that the blue side of a blue cubie is on the blue face of the cube

''' </summary>

Public Overridable Function CorrectForFace(ByVal face As FaceColour, ByVal cube As Cube) As Boolean

Return OnFace(face, cube) And CorrectRotation(face, face)

End Function

''' <summary>

''' Checks if an edge is 'correctly' on the opposite face to what it should be

''' (e.g. if edgeFaceColour = White, it checks if the edge's white face is on the yellow face

''' </summary>

Public Overridable Function CorrectlyOnOppositeFace(ByVal cubieFaceColour As FaceColour, ByVal cube As Cube) As Boolean

Dim cubeFaceColour As FaceColour = Opposite(cubieFaceColour)

Return OnFace(cubeFaceColour, cube) And CorrectRotation(cubieFaceColour, cubeFaceColour)

End Function

''' <summary>

''' Checks if a cubie is in its correct position and orientation

''' </summary>

Public MustOverride Function Correct(ByVal cube As Cube) As Boolean

End Class

## Corner.vb

Imports RubiksCubeSolver\_v2\_0.Helpers

<Serializable()> Public Class Corner

Inherits Block

Private \_secondaryFace As FaceColour

Public Property SecondaryFace() As FaceColour

Get

Return \_secondaryFace

End Get

Set(value As FaceColour)

\_secondaryFace = value

End Set

End Property

Private \_secondaryFaceRotation As FaceColour

Public Property SecondaryRotation() As FaceColour

Get

Return \_secondaryFaceRotation

End Get

Set(value As FaceColour)

\_secondaryFaceRotation = value

End Set

End Property

Public ReadOnly Property SideColours(ByVal topColour As FaceColour) As FaceColour()

Get

If Not HasColour(topColour) Then Throw New ArgumentException("corner does not contain topcolour")

Dim sideCols(1) As FaceColour

Dim index As Integer = 0

For Each colour In Colours

If colour = topColour Then Continue For

sideCols(index) = colour

index += 1

Next

Return sideCols

Throw New Exception("edge does not contain the requested colour")

End Get

End Property

Public Overrides Function ToString() As String

Return Name + ", [" + Colours(0).ToString() + ", " + Colours(1).ToString() + ", " + Colours(2).ToString() + "], " + Position.ToString + ", 1st:" + PrimaryFace.ToString() + "-->" + Rotation.ToString() + ", 2nd:" + SecondaryFace.ToString() + "-->" + SecondaryRotation.ToString()

End Function

Public Overrides Function HasColour(colour As FaceColour) As Boolean

For col = 0 To 2

If Colours(col) = colour Then Return True

Next

Return False

End Function

Public Overrides Function CorrectRotation(cubieFaceColour As FaceColour, cubeFaceColour As FaceColour) As Boolean

If Not HasColour(cubieFaceColour) Then Return False

Select Case cubieFaceColour

Case FaceColour.W, FaceColour.Y

If Rotation <> cubeFaceColour Then Return False

Case FaceColour.R, FaceColour.O

If Rotation = cubeFaceColour Or SecondaryRotation <> cubeFaceColour Then Return False

Case FaceColour.B, FaceColour.G

If Rotation = cubeFaceColour Or SecondaryRotation = cubeFaceColour Then Return False

End Select

Return True

End Function

Public Overrides Function Correct(cube As Cube) As Boolean

Return OnFace(Colours(0), cube) And OnFace(Colours(1), cube) And OnFace(Colours(2), cube) And

CorrectRotation(Colours(0), Colours(0)) And CorrectRotation(Colours(1), Colours(1)) And CorrectRotation(Colours(2), Colours(2))

End Function

Public Overrides Sub SetColoursFromColourString(colourString As String)

For i = 0 To 2

Colours(i) = ColourChar2FaceNumber(colourString(i))

Next

End Sub

End Class

## Edge.vb

Imports RubiksCubeSolver\_v2\_0.Helpers

<Serializable()> Public Class Edge

Inherits Block

Public Sub New()

ReDim Preserve Colours(1)

End Sub

Public Overrides Function ToString() As String

Return Name + ", [" + Colours(0).ToString() + ", " + Colours(1).ToString() + "], " + Position.ToString + ", 1st:" + PrimaryFace.ToString() + "-->" + Rotation.ToString()

End Function

Public Overrides Function HasColour(colour As FaceColour) As Boolean

For col = 0 To 1

If Colours(col) = colour Then Return True

Next

Return False

End Function

Public ReadOnly Property SideColour(ByVal topColour As FaceColour) As FaceColour

Get

If Not HasColour(topColour) Then Throw New ArgumentException("edge does not contain topcolour")

For Each colour In Colours

If colour <> topColour Then Return colour

Next

Throw New Exception("edge only contains topcolour == error in assigning colours")

End Get

End Property

Public ReadOnly Property EdgeColour(ByVal cube As Cube) As FaceColour

Get

Dim colourIndex As Integer

If PrimaryFace = cube.TopFace Then

colourIndex = 1

Else

colourIndex = 0

End If

Return Colours(colourIndex)

End Get

End Property

Public Overrides Function CorrectRotation(cubieFaceColour As FaceColour, cubeFaceColour As FaceColour) As Boolean

If Not HasColour(cubieFaceColour) Then Return False

Select Case cubieFaceColour

Case FaceColour.W, FaceColour.Y

If Rotation <> cubeFaceColour Then Return False

Case FaceColour.R, FaceColour.O

If cubieFaceColour = PrimaryFace Then

If Rotation <> cubeFaceColour Then Return False

Else

If Rotation = cubeFaceColour Then Return False

End If

Case FaceColour.B, FaceColour.G

If Rotation = cubeFaceColour Then Return False

End Select

Return True

End Function

Public Overrides Function Correct(cube As Cube) As Boolean

Return OnFace(Colours(0), cube) And OnFace(Colours(1), cube) And

CorrectRotation(Colours(0), Colours(0)) And CorrectRotation(Colours(1), Colours(1))

End Function

Public Overrides Sub SetColoursFromColourString(colourString As String)

For i = 0 To 1

Colours(i) = ColourChar2FaceNumber(colourString(i))

Next

End Sub

End Class

## Middle.vb

Imports RubiksCubeSolver\_v2\_0.Helpers

<Serializable()> Public Class Middle

Inherits Block

Public Sub New()

ReDim Preserve Colours(0)

End Sub

Public Overrides Function ToString() As String

Return Name + ", [" + Colours(0).ToString() + "], " + Position.ToString + ", 1st:" + PrimaryFace.ToString() + "-->" + Rotation.ToString()

End Function

Public Overrides Function HasColour(colour As FaceColour) As Boolean

Return Colours(0) = colour

End Function

Public Overrides Function CorrectRotation(cubieFaceColour As FaceColour, cubeFaceColour As FaceColour) As Boolean

Return True

End Function

Public Overrides Function Correct(cube As Cube) As Boolean

Return True

End Function

Public Overrides Function CorrectForFace(faceColour As FaceColour, cube As Cube) As Boolean

Return True

End Function

Public Overrides Function CorrectlyOnOppositeFace(cubieFaceColour As FaceColour, cube As Cube) As Boolean

Return False

End Function

Public Overrides Sub SetColoursFromColourString(colourString As String)

Throw New NotImplementedException()

End Sub

End Class

## Matrices.vb

Imports RubiksCubeSolver\_v2\_0.Helpers

Public Module Matrices

Public ReadOnly iVector As Vector3x1 = New Vector3x1(1, 0, 0)

Public ReadOnly jVector As Vector3x1 = New Vector3x1(0, 1, 0)

Public ReadOnly kVector As Vector3x1 = New Vector3x1(0, 0, 1)

Public ReadOnly i4Vector As Vector4x1 = New Vector4x1(1, 0, 0, 0)

Public ReadOnly j4Vector As Vector4x1 = New Vector4x1(0, 1, 0, 0)

Public ReadOnly k4Vector As Vector4x1 = New Vector4x1(0, 0, 1, 0)

Public ReadOnly l4Vector As Vector4x1 = New Vector4x1(0, 0, 0, 1)

Public MustInherit Class Matrix

Private \_columns As Integer

Public Property NoOfColumns As Integer

Get

Return \_columns

End Get

Protected Set(ByVal value As Integer)

\_columns = value

End Set

End Property

Private \_rows As Integer

Public Property NoOfRows As Integer

Get

Return \_rows

End Get

Protected Set(ByVal value As Integer)

\_rows = value

End Set

End Property

End Class

<Serializable()> Public MustInherit Class Vector

Private \_rows

Public Property NoOfRows

Get

Return \_rows

End Get

Protected Set(value)

\_rows = value

End Set

End Property

Public MustOverride Overrides Function ToString() As String

End Class

Public Class Matrix3x3

Inherits Matrix

Public Sub New()

Me.NoOfColumns = 3

Me.NoOfRows = 3

Me.Row1 = iVector

Me.Row2 = jVector

Me.Row3 = kVector

End Sub

Public Sub New(ByVal mat As Matrix3x3)

Me.Row1 = mat.Row1

Me.Row2 = mat.Row2

Me.Row3 = mat.Row3

End Sub

Public Sub New(ByVal row1 As Vector3x1, ByVal row2 As Vector3x1, ByVal row3 As Vector3x1)

Me.NoOfColumns = 3

Me.NoOfRows = 3

Me.Row1 = row1

Me.Row2 = row2

Me.Row3 = row3

End Sub

Public Shared Operator \*(ByVal mat1 As Matrix3x3, ByVal mat2 As Matrix3x3) As Matrix3x3

Return New Matrix3x3(New Vector3x1(mat1.Row1.Dot(mat2.Column1),

mat1.Row1.Dot(mat2.Column2),

mat1.Row1.Dot(mat2.Column3)),

New Vector3x1(mat1.Row2.Dot(mat2.Column1),

mat1.Row2.Dot(mat2.Column2),

mat1.Row2.Dot(mat2.Column3)),

New Vector3x1(mat1.Row3.Dot(mat2.Column1),

mat1.Row3.Dot(mat2.Column2),

mat1.Row3.Dot(mat2.Column3)))

End Operator

Public Shared Operator \*(ByVal mat As Matrix3x3, ByVal vec As Vector3x1) As Vector3x1

Return New Vector3x1(mat.Row1.Dot(vec), mat.Row2.Dot(vec), mat.Row3.Dot(vec))

End Operator

Private \_row1 As Vector3x1

Private \_row2 As Vector3x1

Private \_row3 As Vector3x1

Public Property Row1() As Vector3x1

Get

Return \_row1

End Get

Set(ByVal value As Vector3x1)

\_row1 = value

End Set

End Property

Public Property Row2() As Vector3x1

Get

Return \_row2

End Get

Set(ByVal value As Vector3x1)

\_row2 = value

End Set

End Property

Public Property Row3() As Vector3x1

Get

Return \_row3

End Get

Set(ByVal value As Vector3x1)

\_row3 = value

End Set

End Property

Public Property Column1() As Vector3x1

Get

Return New Vector3x1(\_row1.x, \_row2.x, \_row3.x)

End Get

Set(value As Vector3x1)

\_row1.x = value.x

\_row2.x = value.y

\_row3.x = value.z

End Set

End Property

Public Property Column2() As Vector3x1

Get

Return New Vector3x1(\_row1.y, \_row2.y, \_row3.y)

End Get

Set(value As Vector3x1)

\_row1.y = value.x

\_row2.y = value.y

\_row3.y = value.z

End Set

End Property

Public Property Column3() As Vector3x1

Get

Return New Vector3x1(\_row1.z, \_row2.z, \_row3.z)

End Get

Set(value As Vector3x1)

\_row1.z = value.x

\_row2.z = value.y

\_row3.z = value.z

End Set

End Property

Public Overrides Function ToString() As String

Return "{" + Row1.ToString() + vbNewLine + " " + Row2.ToString() + vbNewLine + " " + Row3.ToString() + "}"

End Function

End Class

Public Class Matrix4x4

Inherits Matrix

Public Sub New()

Me.NoOfColumns = 4

Me.NoOfRows = 4

Me.Row1 = i4Vector

Me.Row2 = j4Vector

Me.Row3 = k4Vector

Me.Row4 = l4Vector

End Sub

Public Sub New(ByVal mat As Matrix4x4)

Me.NoOfColumns = 4

Me.NoOfRows = 4

Me.Row1 = mat.Row1

Me.Row2 = mat.Row2

Me.Row3 = mat.Row3

Me.Row4 = mat.Row4

End Sub

Public Sub New(ByVal row1 As Vector4x1, ByVal row2 As Vector4x1, ByVal row3 As Vector4x1, ByVal row4 As Vector4x1)

Me.NoOfColumns = 4

Me.NoOfRows = 4

Me.Row1 = row1

Me.Row2 = row2

Me.Row3 = row3

Me.Row4 = row4

End Sub

Public Sub New(ByVal mat As OpenTK.Matrix4)

Me.NoOfColumns = 4

Me.NoOfRows = 4

Me.Row1 = New Vector4x1(mat.Column0)

Me.Row2 = New Vector4x1(mat.Column1)

Me.Row3 = New Vector4x1(mat.Column2)

Me.Row4 = New Vector4x1(mat.Column3)

End Sub

Public Shared Operator \*(ByVal mat As Matrix4x4, ByVal vec As Vector4x1) As Vector4x1

Return New Vector4x1(mat.Row1.Dot(vec), mat.Row2.Dot(vec), mat.Row3.Dot(vec), mat.Row4.Dot(vec))

End Operator

Private \_row1 As Vector4x1

Private \_row2 As Vector4x1

Private \_row3 As Vector4x1

Private \_row4 As Vector4x1

Public Property Row1() As Vector4x1

Get

Return \_row1

End Get

Set(ByVal value As Vector4x1)

\_row1 = value

End Set

End Property

Public Property Row2() As Vector4x1

Get

Return \_row2

End Get

Set(ByVal value As Vector4x1)

\_row2 = value

End Set

End Property

Public Property Row3() As Vector4x1

Get

Return \_row3

End Get

Set(ByVal value As Vector4x1)

\_row3 = value

End Set

End Property

Public Property Row4() As Vector4x1

Get

Return \_row4

End Get

Set(ByVal value As Vector4x1)

\_row4 = value

End Set

End Property

End Class

Public Class Vector2x1

Inherits Vector

Public Sub New()

Me.NoOfRows = 2

Me.x = 0

Me.y = 0

End Sub

Public Sub New(ByVal x As Single, ByVal y As Single)

Me.NoOfRows = 2

Me.x = x

Me.y = y

End Sub

Public Shared Operator =(ByVal vec1 As Vector2x1, ByVal vec2 As Vector2x1) As Boolean

Return vec1.x = vec2.x And vec1.y = vec2.y

End Operator

Public Shared Operator <>(ByVal vec1 As Vector2x1, ByVal vec2 As Vector2x1) As Boolean

Return Not vec1 = vec2

End Operator

Private \_x As Single

Private \_y As Single

Public Property x As Single

Get

Return \_x

End Get

Set(ByVal value As Single)

\_x = Math.Round(value, 5)

End Set

End Property

Public Property y As Single

Get

Return \_y

End Get

Set(ByVal value As Single)

\_y = Math.Round(value, 5)

End Set

End Property

Public Overrides Function ToString() As String

Throw New NotImplementedException()

End Function

Public Function Dot(vec As Vector2x1) As Single

Return (Me.x \* vec.x + Me.y \* vec.y)

End Function

End Class

<Serializable()> Public Class Vector3x1

Inherits Vector

Public Sub New()

Me.NoOfRows = 3

Me.x = 0

Me.y = 0

Me.z = 0

End Sub

Public Sub New(ByVal x As Single, ByVal y As Single, ByVal z As Single)

Me.NoOfRows = 3

Me.x = x

Me.y = y

Me.z = z

End Sub

Public Sub New(ByVal vec As Vector3x1)

Me.NoOfRows = 3

Me.x = vec.x

Me.y = vec.y

Me.z = vec.z

End Sub

Public Sub New(ByVal vec As OpenTK.Vector3)

Me.NoOfRows = 3

Me.x = vec.X

Me.y = vec.Y

Me.z = vec.Z

End Sub

Private \_x As Single

Private \_y As Single

Private \_z As Single

Public Property x As Single

Get

Return \_x

End Get

Set(ByVal value As Single)

\_x = Math.Round(value, 5)

End Set

End Property

Public Property y As Single

Get

Return \_y

End Get

Set(ByVal value As Single)

\_y = Math.Round(value, 5)

End Set

End Property

Public Property z As Single

Get

Return \_z

End Get

Set(ByVal value As Single)

\_z = Math.Round(value, 5)

End Set

End Property

Public Shared Operator =(ByVal vec1 As Vector3x1, ByVal vec2 As Vector3x1)

Return vec1.x = vec2.x And vec1.y = vec2.y And vec1.z = vec2.z

End Operator

Public Shared Operator <>(ByVal vec1 As Vector3x1, ByVal vec2 As Vector3x1)

Return Not vec1 = vec2

End Operator

Public Overrides Function ToString() As String

Return "[" + x.ToString() + ", " + y.ToString() + ", " + z.ToString() + "]"

End Function

Public Function Dot(vec As Vector3x1) As Single

Return (Me.x \* vec.x + Me.y \* vec.y + Me.z \* vec.z)

End Function

End Class

Public Class Vector4x1

Inherits Vector

Public Sub New()

Me.NoOfRows = 4

Me.x = 0

Me.y = 0

Me.z = 0

Me.w = 0

End Sub

Public Sub New(ByVal x As Single, ByVal y As Single, ByVal z As Single, ByVal w As Single)

Me.NoOfRows = 4

Me.x = x

Me.y = y

Me.z = z

Me.w = w

End Sub

Public Sub New(ByVal vec As Vector4x1)

Me.NoOfRows = 4

Me.x = vec.x

Me.y = vec.y

Me.z = vec.z

Me.w = vec.w

End Sub

Public Sub New(ByVal vec As OpenTK.Vector4)

Me.NoOfRows = 4

Me.x = vec.X

Me.y = vec.Y

Me.z = vec.Z

Me.w = vec.W

End Sub

Private \_x As Single

Private \_y As Single

Private \_z As Single

Private \_w As Single

Public Property x As Single

Get

Return \_x

End Get

Set(ByVal value As Single)

\_x = Math.Round(value, 10)

End Set

End Property

Public Property y As Single

Get

Return \_y

End Get

Set(ByVal value As Single)

\_y = Math.Round(value, 10)

End Set

End Property

Public Property z As Single

Get

Return \_z

End Get

Set(ByVal value As Single)

\_z = Math.Round(value, 10)

End Set

End Property

Public Property w As Single

Get

Return \_w

End Get

Set(ByVal value As Single)

\_w = Math.Round(value, 10)

End Set

End Property

Public Shared Operator =(ByVal vec1 As Vector4x1, ByVal vec2 As Vector4x1)

Return vec1.x = vec2.x And vec1.y = vec2.y And vec1.z = vec2.z And vec1.w = vec2.w

End Operator

Public Shared Operator <>(ByVal vec1 As Vector4x1, ByVal vec2 As Vector4x1)

Return Not vec1 = vec2

End Operator

Public Overrides Function ToString() As String

Throw New NotImplementedException()

End Function

Public Function Dot(vec As Vector4x1) As Single

Return (Me.x \* vec.x + Me.y \* vec.y + Me.z \* vec.z + Me.w \* vec.w)

End Function

End Class

Public Function GetRotationMatrix(ByVal axis As Axis, ByVal angle As Single) As Matrix3x3

Select Case axis

Case axis.X

Return New Matrix3x3(New Vector3x1(1, 0, 0),

New Vector3x1(0, Math.Cos(angle), -Math.Sin(angle)),

New Vector3x1(0, Math.Sin(angle), Math.Cos(angle)))

Case axis.Y

Return New Matrix3x3(New Vector3x1(Math.Cos(angle), 0, Math.Sin(angle)),

New Vector3x1(0, 1, 0),

New Vector3x1(-Math.Sin(angle), 0, Math.Cos(angle)))

Case axis.Z

Return New Matrix3x3(New Vector3x1(Math.Cos(angle), -Math.Sin(angle), 0),

New Vector3x1(Math.Sin(angle), Math.Cos(angle), 0),

New Vector3x1(0, 0, 1))

Case Else

MsgBox("Error with rotation Axis")

Return New Matrix3x3()

End Select

End Function

End Module

## TopLayerSolver.vb

Imports RubiksCubeSolver\_v2\_0.Helpers

Public Class TopLayerSolver

Private cube As Cube

Sub New(ByRef cubeToBeSolved As Cube)

cube = cubeToBeSolved

End Sub

Public Sub Solve()

Dim colourOfCompletedFace As FaceColour = FaceColour.None

colourOfCompletedFace = GetAnyCompleteFace()

If colourOfCompletedFace = FaceColour.None Then

Dim colourOfRotatedCompletedFace As FaceColour = GetAnyRotatedCompleteFace()

If colourOfRotatedCompletedFace = FaceColour.None Then

DoTopFace()

Else

cube.RotateFaceToTop(colourOfRotatedCompletedFace)

End If

RotateTopFaceCorrectly()

Else

cube.RotateFaceToTop(colourOfCompletedFace)

End If

End Sub

Private Sub DoTopFace()

Dim colourOfCross As FaceColour = FaceColour.None

colourOfCross = GetAnyCrossWithCorrectEdges()

If colourOfCross = FaceColour.None Then

DoTopCross()

Else

cube.RotateFaceToTop(colourOfCross)

End If

DoTopCorners()

End Sub

Private Sub DoTopCross()

MakeWrongEdgedCrossOnBottomFace()

MakeCorrectCrossFromOppositeCross()

End Sub

Private Sub DoTopCorners()

Dim topCornersWhenSolved() As Corner = cube.BlocksByColour(cube.TopFace).Extract(Of Corner)()

For Each corner In topCornersWhenSolved

RotateTopFaceCorrectly()

If Not corner.Correct(cube) Then PutCornerIntoCorrectPlaceOnTopFace(corner)

Next

End Sub

Private Sub MakeWrongEdgedCrossOnBottomFace()

Dim topColourEdges() As Edge = cube.BlocksByColour(cube.TopFace).Extract(Of Edge)()

cube.RotateFaceToTop(cube.BottomFace)

For Each edge In topColourEdges

If Not edge.CorrectlyOnOppositeFace(cube.BottomFace, cube) Then PutEdgeIntoEmptySpaceOnTopFace(edge, cube.BottomFace)

Next

cube.RotateFaceToTop(cube.BottomFace)

End Sub

Private Sub PutEdgeIntoEmptySpaceOnTopFace(ByVal edge As Edge, ByVal crossColour As FaceColour)

Dim edgeBeingDisplaced As Edge = GetEdgeBeingDisplaced(edge)

While edgeBeingDisplaced.CorrectlyOnOppositeFace(crossColour, cube) And edgeBeingDisplaced.HasColour(crossColour)

cube.RotateFace(cube.TopFace, Direction.CLOCKWISE)

edgeBeingDisplaced = GetEdgeBeingDisplaced(edge)

End While

Select Case edge.Layer

Case Layer.TOP

RotateForwardsTopLayerEdgeToTop(edge)

Case Layer.MIDDLE

RotateMiddleLayerEdgeToTop(edge)

Case Layer.BOTTOM

If edge.CorrectRotation(crossColour, cube.BottomFace) Then

RotateDownwardsBottomLayerEdgeToTop(edge)

Else

RotateForwardsBottomLayerEdgeToTop(edge)

End If

End Select

End Sub

Private Function GetEdgeBeingDisplaced(ByVal edgeBeingMoved As Edge)

Dim possibleEdges() As Edge = cube.TopEdgesClockwise

For Each edge In possibleEdges

If edge.OnFace(FaceToRotateForCrossEdge(edgeBeingMoved), cube) Then Return edge

Next

Throw New Exception("Didn't return an edge being displaced - code error")

End Function

Private Sub RotateForwardsTopLayerEdgeToTop(edge As Edge)

If edge.Layer <> Layer.TOP Then Throw New ArgumentException("The cubie is not on the top layer.")

cube.RotateFace(FaceToRotateForCrossEdge(edge), Direction.CLOCKWISE)

cube.RotateFace(cube.TopFace, Direction.ANTICLOCKWISE)

RotateMiddleLayerEdgeToTop(edge)

End Sub

Private Sub RotateMiddleLayerEdgeToTop(edge As Edge)

If edge.Layer <> Layer.MIDDLE Then Throw New ArgumentException("The cubie is not on the middle layer.")

Dim faceToRotate As FaceColour = FaceToRotateForCrossEdge(edge)

Dim direction As Direction = DirectionForMiddleLayerEdgeToTop(edge, faceToRotate, cube)

cube.RotateFace(faceToRotate, direction)

End Sub

Private Function DirectionForMiddleLayerEdgeToTop(ByVal cubie As Edge, ByVal faceToRotate As FaceColour, ByVal cube As Cube) As Direction

Dim faces() As FaceColour = cube.FaceColours

Select Case faceToRotate

Case faces(MoveFaces.LEFT)

Return cubie.Position.z

Case faces(MoveFaces.BACK)

Return -cubie.Position.x

Case faces(MoveFaces.RIGHT)

Return -cubie.Position.z

Case faces(MoveFaces.FRONT)

Return cubie.Position.x

Case Else

Return Direction.NO\_CHANGE

End Select

End Function

Private Sub RotateDownwardsBottomLayerEdgeToTop(ByVal edge As Edge)

If edge.Layer <> Layer.BOTTOM Then Throw New ArgumentException("The cubie is not on the bottom layer.")

Dim faceToRotate As FaceColour = FaceToRotateForCrossEdge(edge)

cube.RotateFace(faceToRotate, Direction.HALF\_TURN)

End Sub

Private Sub RotateForwardsBottomLayerEdgeToTop(ByVal edge As Edge)

If edge.Layer <> Layer.BOTTOM Then Throw New ArgumentException("The cubie is not on the bottom layer.")

cube.RotateFace(cube.TopFace, Direction.CLOCKWISE)

Dim originalFace As FaceColour = FaceToRotateForCrossEdge(edge)

cube.RotateFace(originalFace, Direction.CLOCKWISE)

RotateMiddleLayerEdgeToTop(edge)

cube.RotateFace(originalFace, Direction.ANTICLOCKWISE)

End Sub

Private Function FaceToRotateForCrossEdge(ByVal cubie As Edge) As FaceColour

Return IIf(cubie.Layer = Layer.MIDDLE, FaceToRotateForMiddleLayerEdge(cubie, cube.BottomFace),

FaceToRotateForNotMiddleLayerEdge(cubie))

End Function

Private Function FaceToRotateForMiddleLayerEdge(ByVal cubie As Edge, ByVal colourThatWillPointUp As FaceColour) As FaceColour

Dim faces() As FaceColour = cube.FaceColours

If cubie.PrimaryFace <> colourThatWillPointUp Then Return cubie.Rotation

Dim faceOne, faceTwo As FaceColour

Select Case (cubie.Position.x + cubie.Position.z)

Case -2 'Back left edge

faceOne = faces(MoveFaces.LEFT)

faceTwo = faces(MoveFaces.BACK)

Case 0

If cubie.Position.x = -1 Then ' Front left edge

faceOne = faces(MoveFaces.FRONT)

faceTwo = faces(MoveFaces.LEFT)

ElseIf cubie.Position.x = 1 Then ' Back right edge

faceOne = faces(MoveFaces.BACK)

faceTwo = faces(MoveFaces.RIGHT)

End If

Case 2 ' Front right edge

faceOne = faces(MoveFaces.RIGHT)

faceTwo = faces(MoveFaces.FRONT)

End Select

Return IIf(faceOne = cubie.Rotation, faceTwo, faceOne)

End Function

Private Function FaceToRotateForNotMiddleLayerEdge(ByVal cubie As Edge) As FaceColour

Dim faces() As FaceColour = cube.FaceColours

Select Case cubie.Position.x

Case -1 ' left edge cubie

Return faces(MoveFaces.LEFT)

Case 0

If cubie.Position.z = -1 Then ' back edge cubie

Return faces(MoveFaces.BACK)

Else ' front edge cubie

Return faces(MoveFaces.FRONT)

End If

Case 1 ' right edge cubie

Return faces(MoveFaces.RIGHT)

End Select

Throw New ArgumentException("edge.position is invalid")

End Function

Private Sub MakeCorrectCrossFromOppositeCross()

Dim crossEdges() As Edge = cube.BlocksByColour(cube.TopFace).Extract(Of Edge)()

For Each edge In crossEdges

Dim edgeSideColour As FaceColour = edge.SideColour(cube.TopFace)

While Not edge.OnFace(edgeSideColour, cube)

cube.RotateFace(cube.BottomFace, Direction.ANTICLOCKWISE)

End While

cube.RotateFace(edgeSideColour, Direction.HALF\_TURN)

Next

End Sub

Private Sub PutCornerIntoCorrectPlaceOnTopFace(ByVal corner As Corner)

RotateTopFaceCorrectly()

Select Case corner.Layer

Case Layer.TOP

If corner.Correct(cube) Then Return

Dim faceCornerIsOn As FaceColour = FacesEitherSideOfCorner(corner)(0)

cube.RotateFace(faceCornerIsOn, Direction.ANTICLOCKWISE)

cube.RotateFace(cube.BottomFace, Direction.HALF\_TURN)

cube.RotateFace(faceCornerIsOn, Direction.CLOCKWISE)

RotateCornerFromBottomLayerToTop(corner)

Case Layer.BOTTOM

RotateCornerFromBottomLayerToTop(corner)

Case Else

Throw New ArgumentException("corner is not on top or bottom layer")

End Select

End Sub

Private Sub RotateCornerFromBottomLayerToTop(ByVal corner As Corner)

RotateBottomFaceSoCornerLinesUpWithTopCross(corner)

Dim facesToRotate() As FaceColour = FacesEitherSideOfCorner(corner)

'when viewed from topFace

Dim anticlockwiseFace As FaceColour = facesToRotate(0)

Dim clockwiseFace As FaceColour = facesToRotate(1)

If TopCornerFacingDownwards(corner) Then

Algorithms.MoveTopCornerPointingDownwardsFromBottomToTop(clockwiseFace, anticlockwiseFace, cube)

Else

Dim faceRotatedTowards As FaceColour = FaceBottomCornerIsRotatedTowards(corner, cube.TopFace)

If faceRotatedTowards = clockwiseFace Then

Algorithms.MoveTopCornerPointingLeftFromBottomToTop(anticlockwiseFace, cube)

ElseIf faceRotatedTowards = anticlockwiseFace Then

Algorithms.MoveTopCornerPointingRightFromBottomToTop(clockwiseFace, cube)

End If

End If

End Sub

Private Function TopCornerFacingDownwards(ByVal corner As Corner)

Return corner.CorrectlyOnOppositeFace(cube.TopFace, cube)

End Function

Private Sub RotateBottomFaceSoCornerLinesUpWithTopCross(ByVal corner As Corner)

Dim cornerSideFaces() As FaceColour = corner.SideColours(cube.TopFace)

Dim cubeSideFaces() As FaceColour = FacesEitherSideOfCorner(corner)

Do Until (cubeSideFaces(0) = cornerSideFaces(0) And cubeSideFaces(1) = cornerSideFaces(1)) Or

(cubeSideFaces(0) = cornerSideFaces(1) And cubeSideFaces(1) = cornerSideFaces(0))

cube.RotateFace(cube.BottomFace, Direction.ANTICLOCKWISE)

cubeSideFaces = FacesEitherSideOfCorner(corner)

Loop

End Sub

Private Function FaceBottomCornerIsRotatedTowards(ByVal corner As Corner, ByVal mainColour As FaceColour) As FaceColour

If mainColour = corner.PrimaryFace Then

Return corner.Rotation

ElseIf mainColour = corner.SecondaryFace Then

Return corner.SecondaryRotation

Else

Dim sideFaces() As FaceColour = FacesEitherSideOfCorner(corner)

Dim facesAroundCorner() As FaceColour = {sideFaces(0), sideFaces(1), cube.BottomFace}

For Each face In facesAroundCorner

If face.ToString() = corner.Rotation Or face.ToString() = corner.SecondaryRotation Then Continue For

Return face

Next

End If

Throw New ArgumentException("error in corner")

End Function

'returns {anticlockwise, clockwise}

Private Function FacesEitherSideOfCorner(ByVal cubie As Corner) As FaceColour()

Dim faces() As FaceColour = cube.FaceColours

Select Case (cubie.Position.x + cubie.Position.z)

Case -2 'Back left corner

Return {faces(MoveFaces.LEFT), faces(MoveFaces.BACK)}

Case 0

If cubie.Position.x = -1 Then ' Front left corner

Return {faces(MoveFaces.FRONT), faces(MoveFaces.LEFT)}

ElseIf cubie.Position.x = 1 Then ' Back right corner

Return {faces(MoveFaces.BACK), faces(MoveFaces.RIGHT)}

End If

Case 2 ' Front right corner

Return {faces(MoveFaces.RIGHT), faces(MoveFaces.FRONT)}

End Select

Throw New ArgumentException("corner.position is invalid")

End Function

Private Sub RotateTopFaceCorrectly()

Dim edgesClockwise As Edge() = cube.TopEdgesClockwise

Dim offset As Integer = 0

For i As MoveFaces = MoveFaces.LEFT To MoveFaces.FRONT

If edgesClockwise(i - 1).HasColour(cube.FaceColours(MoveFaces.LEFT)) Then

offset = i - 1

Exit For

End If

Next

If offset = 3 Then offset = -1

cube.RotateFace(cube.TopFace, offset)

End Sub

''' <summary>

''' Gets the colour of the first complete face. If non complete, returns FaceColour.None

''' </summary>

Private Function GetAnyCompleteFace() As FaceColour

For face As FaceColour = Helpers.FaceColour.W To Helpers.FaceColour.O

Dim faceBlocks As Block() = cube.BlocksByColour(face)

For Each block In faceBlocks

If Not block.Correct(cube) Then Exit For 'check next face

If faceBlocks.IsLastElement(block) Then Return face

Next

Next

Return FaceColour.None

End Function

''' <summary>

''' Gets the colour of the first complete face regardless of it's rotation. If non complete, returns FaceColour.None

''' </summary>

Private Function GetAnyRotatedCompleteFace() As FaceColour

For face As FaceColour = Helpers.FaceColour.W To Helpers.FaceColour.O

Dim faceBlocks As Block() = cube.BlocksByColour(face)

For Each block In faceBlocks

If Not block.CorrectForFace(face, cube) Or Not EdgesOfFaceInRightOrder(face) Then Exit For 'check next face

If faceBlocks.IsLastElement(block) Then Return face

Next

Next

Return FaceColour.None

End Function

Private Function EdgesOfFaceInRightOrder(ByVal face As FaceColour) As Boolean

Dim originalTop As FaceColour = cube.CurrentOrientation.Top

cube.RotateFaceToTop(face)

Dim faceBlocksClockwise As Block() = cube.TopBlocksClockwise

ReDim Preserve faceBlocksClockwise(7) ' removes middle block

Dim edgesClockwise As Edge() = faceBlocksClockwise.Extract(Of Edge)()

For i As MoveFaces = MoveFaces.LEFT To MoveFaces.FRONT

If edgesClockwise(i - 1).HasColour(cube.FaceColours(MoveFaces.LEFT)) Then

edgesClockwise = edgesClockwise.Rotate(i - 1)

faceBlocksClockwise = faceBlocksClockwise.Rotate(2 \* i - 2)

Exit For

End If

Next

For i = MoveFaces.LEFT To MoveFaces.FRONT

If Not faceBlocksClockwise(2 \* i - 1).HasColour(cube.FaceColours(i)) Or

Not faceBlocksClockwise(2 \* i - 2).HasColour(cube.FaceColours(i)) Or

Not faceBlocksClockwise((2 \* i) Mod 8).HasColour(cube.FaceColours(i)) Then

cube.RotateFaceToTop(originalTop)

Return False

End If

Next

cube.RotateFaceToTop(originalTop)

Return True

End Function

''' <summary>

''' Gets the colour of the first complete cross. If non complete, returns nothing.

''' </summary>

Private Function GetAnyCrossWithCorrectEdges() As FaceColour

For face As FaceColour = Helpers.FaceColour.W To Helpers.FaceColour.O

Dim faceEdges As Edge() = cube.BlocksByColour(face).Extract(Of Edge)()

For Each edgeCubie In faceEdges

If Not edgeCubie.CorrectForFace(face, cube) Or

(Not edgeCubie.Correct(cube) And Not EdgesOfCrossInRightOrder(face)) Then Exit For 'check next face

If faceEdges.IsLastElement(edgeCubie) Then Return face

Next

Next

Return FaceColour.None

End Function

Private Function EdgesOfCrossInRightOrder(ByVal crossFace As FaceColour) As Boolean

Dim originalTop As FaceColour = cube.CurrentOrientation.Top

cube.RotateFaceToTop(crossFace)

Dim edgesClockwise As Edge() = cube.TopEdgesClockwise

For i As MoveFaces = MoveFaces.LEFT To MoveFaces.FRONT

If edgesClockwise(i - 1).HasColour(MoveFaces.LEFT) Then

edgesClockwise = edgesClockwise.Rotate(i - 1)

Exit For

End If

Next

For i = MoveFaces.LEFT To MoveFaces.FRONT

If Not edgesClockwise(i - 1).HasColour(cube.FaceColours(i)) Then

cube.RotateFaceToTop(originalTop)

Return False

End If

Next

cube.RotateFaceToTop(originalTop)

Return True

End Function

End Class

## MiddleLayerSolver.vb

Imports RubiksCubeSolver\_v2\_0.Helpers

Public Class MiddleLayerSolver

Private cube As Cube

Sub New(ByRef cubeToBeSolved As Cube)

cube = cubeToBeSolved

End Sub

Public Sub Solve()

If Not MiddleRowCorrect() Then DoMiddleRow()

End Sub

Private Function MiddleRowCorrect() As Boolean

Dim middleEdges() As Edge = cube.MiddleEdgesClockwise

If Not (middleEdges(0).HasColour(cube.FaceColours(MoveFaces.FRONT)) And

middleEdges(0).HasColour(cube.FaceColours(MoveFaces.LEFT))) Then Return False

For i = 1 To 3

If Not (middleEdges(i).HasColour(cube.FaceColours(i)) And

middleEdges(i).HasColour(cube.FaceColours(i + 1))) Then Return False

Next

Return True

End Function

Private Sub DoMiddleRow()

Dim edgesWhenSolved() As Edge = MiddleEdgesWhenSolved()

For Each edge In edgesWhenSolved

If Not edge.Correct(cube) Then PutEdgeInMiddle(edge)

Next

End Sub

Private Function MiddleEdgesWhenSolved()

Dim edges(3) As Edge

Dim index As Integer = 0

For Each cubie In cube.Edges

If Not (cubie.HasColour(cube.CurrentOrientation.Top) Or

cubie.HasColour(cube.CurrentOrientation.Bottom)) Then

edges(index) = cubie

index += 1

End If

Next

Return edges

End Function

Private Sub PutEdgeInMiddle(ByVal edge As Edge)

If edge.Layer = Layer.MIDDLE Then

Dim leftFace As FaceColour = ClockwiseFace(edge)

cube.RotateFaceToFront(leftFace)

Algorithms.MiddleRightAlgorithm(cube)

PutEdgeFromBottomToMiddle(edge)

ElseIf edge.Layer = Layer.BOTTOM Then

PutEdgeFromBottomToMiddle(edge)

Else

Throw New StageNotSuccessfulException("The top layer is not complete")

End If

End Sub

'Clockwise when viewed top-down

Private Function ClockwiseFace(ByVal edge As Edge) As FaceColour

Dim faces() As FaceColour = cube.FaceColours

Select Case (edge.Position.x + edge.Position.z)

Case -2 'Back left edge

Return faces(MoveFaces.BACK)

Case 0

If edge.Position.x = -1 Then ' Front left edge

Return faces(MoveFaces.LEFT)

ElseIf edge.Position.x = 1 Then ' Back right edge

Return faces(MoveFaces.RIGHT)

End If

Case 2 ' Front right edge

Return faces(MoveFaces.FRONT)

End Select

Throw New ArgumentException("edge has invalid position vector")

End Function

Private Sub PutEdgeFromBottomToMiddle(ByVal edge As Edge)

If edge.Layer <> Layer.BOTTOM Then Throw New ArgumentException("The cubie is not on the bottom layer.")

Dim sideFaces() As FaceColour = GetEdgeFaceColours(edge)

Dim clockwiseFace As FaceColour = sideFaces(0)

Dim anticlockwiseFace As FaceColour = sideFaces(1)

While Not edge.Correct(cube)

If edge.CorrectForFace(anticlockwiseFace, cube) Then

cube.RotateFaceToFront(anticlockwiseFace)

Algorithms.MiddleRightAlgorithm(cube)

ElseIf edge.CorrectForFace(clockwiseFace, cube) Then

cube.RotateFaceToFront(clockwiseFace)

Algorithms.MiddleLeftAlgorithm(cube)

Else

cube.RotateFace(cube.BottomFace, Direction.ANTICLOCKWISE)

End If

End While

End Sub

''' <returns> {clockwise, anticlockwise} viewed from bottom of the cube </returns>

Private Function GetEdgeFaceColours(ByVal edge As Edge) As FaceColour()

Dim cubieColours(1) As FaceColour

For i = 0 To 1

cubieColours(i) = edge.Colours(i)

Next

For face As MoveFaces = MoveFaces.LEFT To MoveFaces.FRONT

Dim clockwiseFace As MoveFaces = face Mod 4 + 1 'Clockwise when viewed from top

If cube.FaceColours(face) = cubieColours(0) And cube.FaceColours(clockwiseFace) = cubieColours(1) Or

cube.FaceColours(face) = cubieColours(1) And cube.FaceColours(clockwiseFace) = cubieColours(0) Then

Return {cube.FaceColours(face), cube.FaceColours(clockwiseFace)}

End If

Next

Throw New ArgumentException("The edge is not a middle piece")

End Function

End Class

## BottomLayerSolver.vb

Imports RubiksCubeSolver\_v2\_0.Helpers

Public Class BottomLayerSolver

Private cube As Cube

Sub New(ByRef cubeToBeSolved As Cube)

cube = cubeToBeSolved

End Sub

Public Sub Solve()

If Not cube.Complete() Then SolveBottomLayer()

End Sub

Private Sub SolveBottomLayer()

If Not BottomFaceComplete() Then

If Not BottomCrossComplete() Then DoCross()

DoCorners()

End If

RotateBottomFaceCorrectly()

End Sub

Private Function BottomFaceComplete() As Boolean

Dim bottomCubies() As Block = cube.BlocksByColour(cube.BottomFace)

For Each cubie In bottomCubies

If Not cubie.CorrectForFace(cube.BottomFace, cube) Then Return False

Next

Return True

End Function

Private Function BottomCrossComplete() As Boolean

Dim faceEdges As Edge() = cube.BlocksByColour(cube.BottomFace).Extract(Of Edge)()

For Each edgeCubie In faceEdges

If Not edgeCubie.CorrectForFace(cube.BottomFace, cube) Or (Not edgeCubie.Correct(cube) And Not EdgesOfBottomCrossInRightOrder()) Then Return False

Next

Return True

End Function

Private Function EdgesOfBottomCrossInRightOrder() As Boolean

cube.RotateFaceToTop(cube.BottomFace)

Dim edgesClockwise As Edge() = cube.TopEdgesClockwise

For i As MoveFaces = MoveFaces.LEFT To MoveFaces.FRONT

If edgesClockwise(i - 1).HasColour(MoveFaces.LEFT) Then

edgesClockwise = edgesClockwise.Rotate(i - 1)

Exit For

End If

Next

For i = MoveFaces.LEFT To MoveFaces.FRONT

If Not edgesClockwise(i - 1).HasColour(cube.FaceColours(i)) Then

cube.RotateFaceToTop(cube.BottomFace)

Return False

End If

Next

cube.RotateFaceToTop(cube.BottomFace)

Return True

End Function

Private Sub DoCross()

cube.RotateFaceToTop(cube.BottomFace)

DoBottomStage\_L()

DoBottomStage\_Line()

DoBottomStage\_Cross()

PermuteTopCrossEdges()

cube.RotateFaceToTop(cube.BottomFace)

End Sub

'cube is upside down

Private Sub DoBottomStage\_L()

If cube.CorrectTopEdges.Length >= 2 Then Return 'L is done

Algorithms.BottomCrossAlgorithm(cube)

End Sub

'cube is upside down

Private Sub DoBottomStage\_Line()

Dim correctBottomEdges() As Edge = cube.CorrectTopEdges()

If cube.CorrectTopEdges.Length >= 3 Or

New Vector2x1(correctBottomEdges(0).Position.x, correctBottomEdges(0).Position.z).Dot(New Vector2x1(correctBottomEdges(1).Position.x, correctBottomEdges(1).Position.z)) <> 0 Then

Return ' line is done

End If

While (correctBottomEdges(0).Position.x <> -1 Or correctBottomEdges(1).Position.z <> -1) And

(correctBottomEdges(1).Position.x <> -1 Or correctBottomEdges(0).Position.z <> -1)

cube.RotateFace(cube.TopFace, Direction.ANTICLOCKWISE)

End While

Algorithms.BottomCrossAlgorithm(cube)

End Sub

'cube is upside down

Private Sub DoBottomStage\_Cross()

If cube.CorrectTopEdges.Length = 4 Then Return ' Cross is done

Dim correctBottomEdges() As Edge = cube.CorrectTopEdges()

While IsNotLine(correctBottomEdges)

cube.RotateFace(cube.TopFace, Direction.ANTICLOCKWISE)

End While

Algorithms.BottomCrossAlgorithm(cube)

End Sub

Private Function IsNotLine(ByVal correctCrossEdges()) As Boolean

Select Case correctCrossEdges.Length

Case 2

Return (correctCrossEdges(0).Position.x <> -1 Or correctCrossEdges(1).Position.x <> 1) And

(correctCrossEdges(1).Position.x <> -1 Or correctCrossEdges(0).Position.x <> 1)

Case 3

Return (correctCrossEdges(0).Position.x <> -1 Or correctCrossEdges(1).Position.x <> 1) And

(correctCrossEdges(0).Position.x <> -1 Or correctCrossEdges(2).Position.x <> 1) And

(correctCrossEdges(1).Position.x <> -1 Or correctCrossEdges(0).Position.x <> 1) And

(correctCrossEdges(1).Position.x <> -1 Or correctCrossEdges(2).Position.x <> 1) And

(correctCrossEdges(2).Position.x <> -1 Or correctCrossEdges(0).Position.x <> 1) And

(correctCrossEdges(2).Position.x <> -1 Or correctCrossEdges(1).Position.x <> 1)

Case Else

Throw New ArgumentException("2 edge faces are not solved - error in BottomLayerSolver")

End Select

End Function

Private Sub PermuteTopCrossEdges()

Dim crossEdges() As Edge = cube.TopEdgesClockwise()

Dim crossColours(3) As FaceColour

For edge = 0 To 3

crossColours(edge) = crossEdges(edge).SideColour(cube.TopFace)

Next

If crossColours(0) = Opposite(crossColours(2)) Or crossColours(1) = Opposite(crossColours(3)) Then

For faceColour = 1 To 4

If crossColours(0) = cube.FaceColours(faceColour) And crossColours(1) = cube.FaceColours(faceColour Mod 4 + 1) Then Return 'edges are correct

Next

Algorithms.BottomEdgeAlgorithmRight(cube)

cube.RotateFace(cube.TopFace, Direction.ANTICLOCKWISE)

Algorithms.BottomEdgeAlgorithmRight(cube)

Return

End If

Dim indexOfAnticlockwiseEdge As Integer

For edge = 0 To 3

For faceColour = 1 To 4

If crossColours(edge) = cube.FaceColours(faceColour) And

crossColours((edge + 1) Mod 4) = cube.FaceColours(faceColour Mod 4 + 1) Then

indexOfAnticlockwiseEdge = edge

End If

Next

Next

cube.RotateFace(cube.TopFace, indexOfAnticlockwiseEdge - 1)

Algorithms.BottomEdgeAlgorithmRight(cube)

End Sub

Private Sub DoCorners()

PositionBottomCorners()

RotateBottomFaceCorrectly()

OrientateBottomCorners()

End Sub

Private Sub PositionBottomCorners()

RotateBottomFaceCorrectly()

cube.RotateFaceToTop(cube.BottomFace)

Dim corners() As Corner = cube.TopCornersClockwise

Dim correctlyPositionedCorner As Corner

Dim nextClockwiseCorner As Corner = Nothing

Dim cornerPositionIndex As Integer

For cornerPositionIndex = 0 To 3

Dim corner As Corner = corners(cornerPositionIndex)

nextClockwiseCorner = corners((cornerPositionIndex + 1) Mod 4)

If Not CornerIsCorrectlyPositioned(corner) Then Continue For

If CornerIsCorrectlyPositioned(nextClockwiseCorner) Then

cube.RotateFaceToTop(cube.BottomFace)

Return

Else

correctlyPositionedCorner = corner

Exit For

End If

Next

If correctlyPositionedCorner Is Nothing Then

Algorithms.BottomClockwiseCornerAlgorithm(cube)

cube.RotateFaceToTop(cube.BottomFace)

PositionBottomCorners()

Return

End If

PermuteCorners(correctlyPositionedCorner, nextClockwiseCorner, cornerPositionIndex)

cube.RotateFaceToTop(cube.BottomFace)

End Sub

Private Function CornerIsCorrectlyPositioned(ByVal corner As Corner) As Boolean

Dim sideFaces() As FaceColour = FacesEitherSideOfCorner(corner)

Return corner.HasColour(sideFaces(0)) And corner.HasColour(sideFaces(1))

End Function

Private Sub PermuteCorners(correctlyPositionedCorner As Corner, nextClockwiseCorner As Corner, cornerPositionIndex As Integer)

Dim anticlockwiseFace As FaceColour = FacesEitherSideOfCorner(correctlyPositionedCorner)(0)

If nextClockwiseCorner.HasColour(anticlockwiseFace) Then

'Rotate anticlockwise

Select Case cornerPositionIndex

Case 1 : cube.RotateFace(cube.TopFace, Direction.ANTICLOCKWISE)

Case 2 : cube.RotateFace(cube.TopFace, Direction.HALF\_TURN)

Case 3 : cube.RotateFace(cube.TopFace, Direction.CLOCKWISE)

End Select

Algorithms.BottomAnticlockwiseCornerAlgorithm(cube)

Else

'Rotate clockwise

Select Case cornerPositionIndex

Case 0 : cube.RotateFace(cube.TopFace, Direction.ANTICLOCKWISE)

Case 1 : cube.RotateFace(cube.TopFace, Direction.HALF\_TURN)

Case 2 : cube.RotateFace(cube.TopFace, Direction.CLOCKWISE)

End Select

Algorithms.BottomClockwiseCornerAlgorithm(cube)

End If

End Sub

'returns {anticlockwise, clockwise} viewed from top

Private Function FacesEitherSideOfCorner(ByVal cubie As Corner) As FaceColour()

Dim faces() As FaceColour = cube.FaceColours

Select Case (cubie.Position.x + cubie.Position.z)

Case -2 'Back left corner

Return {faces(MoveFaces.LEFT), faces(MoveFaces.BACK)}

Case 0

If cubie.Position.x = -1 Then ' Front left corner

Return {faces(MoveFaces.FRONT), faces(MoveFaces.LEFT)}

ElseIf cubie.Position.x = 1 Then ' Back right corner

Return {faces(MoveFaces.BACK), faces(MoveFaces.RIGHT)}

End If

Case 2 ' Front right corner

Return {faces(MoveFaces.RIGHT), faces(MoveFaces.FRONT)}

End Select

Throw New ArgumentException("corner position is invalid")

End Function

Private Sub OrientateBottomCorners()

cube.RotateFaceToTop(cube.BottomFace)

Dim numberOfIncorrectCorners As Integer = cube.IncorrectTopCorners.Length

Select Case numberOfIncorrectCorners

Case 4 : Orientate4Corners()

Case 2 : Orientate2Corners()

Case 3 : Orientate3Corners()

Case 0 : Return

Case Else : Throw New StageNotSuccessfulException("Error in previous stages - invalid cube")

End Select

cube.RotateFaceToTop(cube.BottomFace)

End Sub

Private Sub Orientate4Corners()

Dim originalOrientation As CubeOrientation = New CubeOrientation(cube.CurrentOrientation)

Dim incorrectCorners() As Corner = cube.IncorrectTopCorners()

Dim facesCommonToBoth() As FaceColour

Dim faceBothRotatedTowards As FaceColour

For i = 0 To 3

'will only work if incorrectCorners is clockwise

facesCommonToBoth = GetFacesContainingBothCorners(incorrectCorners(i), incorrectCorners(i + 1))

faceBothRotatedTowards = GetFaceBothRotatedTowards(incorrectCorners(i), incorrectCorners(i + 1))

If faceBothRotatedTowards <> FaceColour.None Then Exit For

Next

cube.RotateFaceToTop(faceBothRotatedTowards)

If faceBothRotatedTowards = facesCommonToBoth(0) Then

cube.RotateFaceToLeft(facesCommonToBoth(1))

ElseIf faceBothRotatedTowards = facesCommonToBoth(1) Then

cube.RotateFaceToLeft(facesCommonToBoth(0))

Else

Throw New StageNotSuccessfulException()

End If

Algorithms.BottomEdgeAlgorithmRight(cube)

Algorithms.BottomEdgeAlgorithmLeft(cube)

cube.RotateFaceToTop(originalOrientation.Top)

cube.RotateFaceToFront(originalOrientation.Front)

cube.RotateFaceToTop(cube.BottomFace)

OrientateBottomCorners()

cube.RotateFaceToTop(cube.BottomFace)

End Sub

Private Sub Orientate2Corners()

Dim originalOrientation As CubeOrientation = New CubeOrientation(cube.CurrentOrientation)

Dim correctCorners() As Corner = cube.CorrectTopCorners()

If correctCorners.Length <> 2 Then Throw New InvalidOperationException

If CornersAreOpposite(correctCorners(0), correctCorners(1)) Then OrientateOppositeCorners()

Dim incorrectCorners() As Corner = cube.IncorrectTopCorners()

Dim facesCommonToBoth() As FaceColour = GetFacesContainingBothCorners(incorrectCorners(0), incorrectCorners(1))

Dim faceBothRotatedTowards As FaceColour = GetFaceBothRotatedTowards(incorrectCorners(0), incorrectCorners(1))

cube.RotateFaceToTop(faceBothRotatedTowards)

If faceBothRotatedTowards = facesCommonToBoth(0) Then

cube.RotateFaceToLeft(facesCommonToBoth(1))

ElseIf faceBothRotatedTowards = facesCommonToBoth(1) Then

cube.RotateFaceToLeft(facesCommonToBoth(0))

Else

Throw New StageNotSuccessfulException()

End If

Algorithms.BottomEdgeAlgorithmRight(cube)

Algorithms.BottomEdgeAlgorithmLeft(cube)

cube.RotateFaceToTop(originalOrientation.Top)

cube.RotateFaceToFront(originalOrientation.Front)

End Sub

Private Function CornersAreOpposite(ByVal corner1 As Corner, ByVal corner2 As Corner) As Boolean

Return ((corner1.Position.x + corner1.Position.z) - (corner2.Position.x + corner2.Position.z)) Mod 4 = 0

End Function

Private Sub OrientateOppositeCorners()

Dim bottomFaceColour As FaceColour = cube.TopFace

Dim incorrectCorners() As Corner = cube.IncorrectTopCorners()

cube.Rotate(Direction.ANTICLOCKWISE, Axis.Z)

Dim cornerOnTopFace As Corner

Do

cube.Rotate(Direction.ANTICLOCKWISE, Axis.X)

cornerOnTopFace = IIf(incorrectCorners(0).OnFace(cube.TopFace, cube), incorrectCorners(0), incorrectCorners(1))

Loop While Not cornerOnTopFace.CorrectRotation(bottomFaceColour, cube.TopFace)

Algorithms.BottomEdgeAlgorithmRight(cube)

Algorithms.BottomEdgeAlgorithmLeft(cube)

cube.Rotate(Direction.CLOCKWISE, Axis.Z)

End Sub

Private Function GetFaceBothRotatedTowards(ByVal corner1 As Corner, ByVal corner2 As Corner) As FaceColour

Dim face As FaceColour = FaceColour.None

If corner1.PrimaryFace = corner2.PrimaryFace And corner1.Rotation = corner2.Rotation Then

face = corner1.Rotation

ElseIf corner1.SecondaryFace = corner2.SecondaryFace And corner1.SecondaryRotation = corner2.SecondaryRotation Then

face = corner1.SecondaryRotation

Else

Dim corner1Faces() As FaceColour = FacesEitherSideOfCorner(corner1)

corner1Faces.Append(cube.TopFace)

'probably needs an except method written

face = corner1Faces.Except({corner1.Rotation, corner1.SecondaryRotation})(0)

End If

If corner1.OnFace(face, cube) And corner2.OnFace(face, cube) Then Return face

Return FaceColour.None

End Function

Private Function GetFacesContainingBothCorners(ByVal corner1 As Corner, ByVal corner2 As Corner) As FaceColour()

Dim corner1Faces() As FaceColour = FacesEitherSideOfCorner(corner1)

Dim corner2Faces() As FaceColour = FacesEitherSideOfCorner(corner2)

Dim commonFaces() As FaceColour = corner1Faces.Intersect(corner2Faces)

commonFaces.Append(cube.TopFace)

Return commonFaces

End Function

Private Sub Orientate3Corners()

cube.Rotate(Direction.ANTICLOCKWISE, Axis.Z)

Dim leftTopEdgeCorners(1) As Corner

Dim count As Integer

Do

cube.Rotate(Direction.CLOCKWISE, Axis.X)

count = 0

For Each corner In cube.Corners

If corner.Position = New Vector3x1(-1, 1, 1) Or corner.Position = New Vector3x1(-1, 1, -1) Then

leftTopEdgeCorners(count) = corner

count += 1

End If

Next

Loop While (leftTopEdgeCorners(0).Correct(cube) Or leftTopEdgeCorners(1).Correct(cube))

Algorithms.BottomEdgeAlgorithmRight(cube)

Algorithms.BottomEdgeAlgorithmLeft(cube)

cube.Rotate(Direction.CLOCKWISE, Axis.Z)

If cube.CorrectTopCorners.Length <> 4 Then

cube.RotateFaceToTop(cube.BottomFace)

OrientateBottomCorners()

End If

cube.RotateFaceToTop(cube.BottomFace)

End Sub

Private Sub RotateBottomFaceCorrectly()

cube.RotateFaceToTop(cube.BottomFace)

Dim edgesClockwise As Edge() = cube.TopEdgesClockwise

Dim offset As Integer = 0

For i As MoveFaces = MoveFaces.LEFT To MoveFaces.FRONT

If edgesClockwise(i - 1).HasColour(cube.FaceColours(MoveFaces.LEFT)) Then

offset = i - 1

Exit For

End If

Next

If offset = 3 Then offset = -1

cube.RotateFace(cube.TopFace, offset)

cube.RotateFaceToTop(cube.BottomFace)

End Sub

End Clas

## CubeOrientation.vb

Imports RubiksCubeSolver\_v2\_0.Helpers

<Serializable()> Public Class CubeOrientation

Public Property Top As FaceColour

Public ReadOnly Property Bottom As FaceColour

Get

Return Opposite(Top)

End Get

End Property

Public Property Front As FaceColour

Public ReadOnly Property UpsideDown As CubeOrientation

Get

Return New CubeOrientation(Bottom, Front)

End Get

End Property

Sub New(ByVal top As Char, ByVal front As Char)

Me.Top = ColourChar2FaceNumber(top)

Me.Front = ColourChar2FaceNumber(front)

End Sub

Sub New(ByVal top As FaceColour, ByVal front As FaceColour)

Me.Top = top

Me.Front = front

End Sub

Sub New(ByVal orientation As CubeOrientation)

Top = orientation.Top

Front = orientation.Front

End Sub

Public Overrides Function ToString() As String

Return "Top: " + Top.ToString() + ", Front: " + Front.ToString()

End Function

Public Function Copy() As CubeOrientation

Return New CubeOrientation(Top, Front)

End Function

End Class

## Algorithms.vb

Imports RubiksCubeSolver\_v2\_0.Helpers.PublicConstants

Imports RubiksCubeSolver\_v2\_0.Helpers.PublicConstants.Movecube.FaceColours

Imports RubiksCubeSolver\_v2\_0.Helpers.PublicConstants.Direction

Module Algorithms

Public Sub MoveTopCornerPointingDownwardsFromBottomToTop(ByVal clockwiseFace As FaceColour, ByVal anticlockwiseFace As FaceColour, ByRef cube As Cube)

cube.RotateFace(anticlockwiseFace, Direction.ANTICLOCKWISE)

cube.RotateFace(cube.BottomFace, Direction.CLOCKWISE)

cube.RotateFace(anticlockwiseFace, Direction.CLOCKWISE)

cube.RotateFace(clockwiseFace, Direction.CLOCKWISE)

cube.RotateFace(cube.BottomFace, Direction.HALF\_TURN)

cube.RotateFace(clockwiseFace, Direction.ANTICLOCKWISE)

End Sub

Public Sub MoveTopCornerPointingLeftFromBottomToTop(ByVal anticlockwiseFace As FaceColour, ByRef cube As Cube)

cube.RotateFace(cube.BottomFace, Direction.ANTICLOCKWISE)

cube.RotateFace(anticlockwiseFace, Direction.ANTICLOCKWISE)

cube.RotateFace(cube.BottomFace, Direction.CLOCKWISE)

cube.RotateFace(anticlockwiseFace, Direction.CLOCKWISE)

End Sub

Public Sub MoveTopCornerPointingRightFromBottomToTop(ByVal clockwiseFace As FaceColour, ByRef cube As Cube)

cube.RotateFace(cube.BottomFace, Direction.CLOCKWISE)

cube.RotateFace(clockwiseFace, Direction.CLOCKWISE)

cube.RotateFace(cube.BottomFace, Direction.ANTICLOCKWISE)

cube.RotateFace(clockwiseFace, Direction.ANTICLOCKWISE)

End Sub

Public Sub MiddleLeftAlgorithm(ByRef cube As Cube)

cube.RotateFace(cube.FaceColours(MoveFaces.BOTTOM), CLOCKWISE)

cube.RotateFace(cube.FaceColours(MoveFaces.LEFT), CLOCKWISE)

cube.RotateFace(cube.FaceColours(MoveFaces.BOTTOM), ANTICLOCKWISE)

cube.RotateFace(cube.FaceColours(MoveFaces.LEFT), ANTICLOCKWISE)

cube.RotateFace(cube.FaceColours(MoveFaces.BOTTOM), ANTICLOCKWISE)

cube.RotateFace(cube.FaceColours(MoveFaces.FRONT), ANTICLOCKWISE)

cube.RotateFace(cube.FaceColours(MoveFaces.BOTTOM), CLOCKWISE)

cube.RotateFace(cube.FaceColours(MoveFaces.FRONT), CLOCKWISE)

End Sub

Public Sub MiddleRightAlgorithm(ByRef cube As Cube)

cube.RotateFace(cube.FaceColours(MoveFaces.BOTTOM), ANTICLOCKWISE)

cube.RotateFace(cube.FaceColours(MoveFaces.RIGHT), ANTICLOCKWISE)

cube.RotateFace(cube.FaceColours(MoveFaces.BOTTOM), CLOCKWISE)

cube.RotateFace(cube.FaceColours(MoveFaces.RIGHT), CLOCKWISE)

cube.RotateFace(cube.FaceColours(MoveFaces.BOTTOM), CLOCKWISE)

cube.RotateFace(cube.FaceColours(MoveFaces.FRONT), CLOCKWISE)

cube.RotateFace(cube.FaceColours(MoveFaces.BOTTOM), ANTICLOCKWISE)

cube.RotateFace(cube.FaceColours(MoveFaces.FRONT), ANTICLOCKWISE)

End Sub

Public Sub BottomCrossAlgorithm(ByRef cube As Cube)

cube.RotateFace(cube.FaceColours(MoveFaces.FRONT), CLOCKWISE)

cube.RotateFace(cube.FaceColours(MoveFaces.RIGHT), CLOCKWISE)

cube.RotateFace(cube.FaceColours(MoveFaces.TOP), CLOCKWISE)

cube.RotateFace(cube.FaceColours(MoveFaces.RIGHT), ANTICLOCKWISE)

cube.RotateFace(cube.FaceColours(MoveFaces.TOP), ANTICLOCKWISE)

cube.RotateFace(cube.FaceColours(MoveFaces.FRONT), ANTICLOCKWISE)

End Sub

Public Sub BottomEdgeAlgorithmRight(ByRef cube As Cube)

cube.RotateFace(cube.FaceColours(MoveFaces.RIGHT), CLOCKWISE)

cube.RotateFace(cube.FaceColours(MoveFaces.TOP), CLOCKWISE)

cube.RotateFace(cube.FaceColours(MoveFaces.RIGHT), ANTICLOCKWISE)

cube.RotateFace(cube.FaceColours(MoveFaces.TOP), CLOCKWISE)

cube.RotateFace(cube.FaceColours(MoveFaces.RIGHT), CLOCKWISE)

cube.RotateFace(cube.FaceColours(MoveFaces.TOP), HALF\_TURN)

cube.RotateFace(cube.FaceColours(MoveFaces.RIGHT), ANTICLOCKWISE)

End Sub

Public Sub BottomEdgeAlgorithmLeft(ByRef cube As Cube)

cube.RotateFace(cube.FaceColours(MoveFaces.LEFT), ANTICLOCKWISE)

cube.RotateFace(cube.FaceColours(MoveFaces.TOP), ANTICLOCKWISE)

cube.RotateFace(cube.FaceColours(MoveFaces.LEFT), CLOCKWISE)

cube.RotateFace(cube.FaceColours(MoveFaces.TOP), ANTICLOCKWISE)

cube.RotateFace(cube.FaceColours(MoveFaces.LEFT), ANTICLOCKWISE)

cube.RotateFace(cube.FaceColours(MoveFaces.TOP), HALF\_TURN)

cube.RotateFace(cube.FaceColours(MoveFaces.LEFT), CLOCKWISE)

End Sub

Public Sub BottomAnticlockwiseCornerAlgorithm(ByRef cube As Cube)

cube.RotateFace(cube.FaceColours(MoveFaces.RIGHT), CLOCKWISE)

cube.RotateFace(cube.FaceColours(MoveFaces.TOP), ANTICLOCKWISE)

cube.RotateFace(cube.FaceColours(MoveFaces.LEFT), ANTICLOCKWISE)

cube.RotateFace(cube.FaceColours(MoveFaces.TOP), CLOCKWISE)

cube.RotateFace(cube.FaceColours(MoveFaces.RIGHT), ANTICLOCKWISE)

cube.RotateFace(cube.FaceColours(MoveFaces.TOP), ANTICLOCKWISE)

cube.RotateFace(cube.FaceColours(MoveFaces.LEFT), CLOCKWISE)

End Sub

Public Sub BottomClockwiseCornerAlgorithm(ByRef cube As Cube)

cube.RotateFace(cube.FaceColours(MoveFaces.LEFT), ANTICLOCKWISE)

cube.RotateFace(cube.FaceColours(MoveFaces.TOP), CLOCKWISE)

cube.RotateFace(cube.FaceColours(MoveFaces.RIGHT), CLOCKWISE)

cube.RotateFace(cube.FaceColours(MoveFaces.TOP), ANTICLOCKWISE)

cube.RotateFace(cube.FaceColours(MoveFaces.LEFT), CLOCKWISE)

cube.RotateFace(cube.FaceColours(MoveFaces.TOP), CLOCKWISE)

cube.RotateFace(cube.FaceColours(MoveFaces.RIGHT), ANTICLOCKWISE)

End Sub

End Module

## ArrayMethods.vb

Imports System.Runtime.CompilerServices

Imports RubiksCubeSolver\_v2\_0.Helpers

Public Module ArrayExtentions

<Extension()>

Public Function Zero(ByRef array() As Integer) As Integer()

If array.Length < 1 Then Return Nothing

For i = 0 To UBound(array)

array(i) = 0

Next

Return array

End Function

<Extension()>

Public Sub Append(Of T)(ByRef array() As T, ByVal item As T)

Try

ReDim Preserve array(array.Length)

array(array.Length - 1) = item

Catch ex As NullReferenceException

array = {item}

End Try

End Sub

<Extension()>

Public Function Intersect(ByVal faces1() As FaceColour, ByVal faces2() As FaceColour) As FaceColour()

If faces1.Length < faces2.Length Then

Dim tmp() As FaceColour = faces1.Copy()

faces1 = faces2.Copy()

faces2 = tmp.Copy()

End If

Dim intersection() As FaceColour

For Each colour In faces1

If faces2.LinearSearch(colour) <> -1 Then intersection.Append(colour)

Next

Return intersection

End Function

<Extension()>

Public Function Extract(Of T As Block)(ByVal blocks() As Block) As T()

Dim tArray(blocks.Length) As T

Dim index As Integer = 0

For Each block In blocks

If block.GetType = GetType(T) Then

tArray(index) = block

index += 1

End If

Next

ReDim Preserve tArray(index - 1)

Return tArray

End Function

<Extension()>

Public Function Copy(ByVal corners() As Corner) As Corner()

Dim cornerCopy(0 To corners.Length - 1) As Corner

For i = 0 To corners.Length - 1

cornerCopy(i) = New Corner With {

.Name = corners(i).Name,

.Rotation = corners(i).Rotation,

.SecondaryFace = corners(i).SecondaryFace,

.SecondaryRotation = corners(i).SecondaryRotation,

.Position = corners(i).Position,

.Colours = corners(i).Colours

}

Next

Return cornerCopy

End Function

<Extension()>

Public Function Copy(ByVal edges() As Edge) As Edge()

Dim edgeCopy(0 To edges.Length - 1) As Edge

For i = 0 To edges.Length - 1

edgeCopy(i) = New Edge With {

.Name = edges(i).Name,

.Rotation = edges(i).Rotation,

.Position = edges(i).Position,

.Colours = edges(i).Colours

}

Next

Return edgeCopy

End Function

<Extension()>

Public Function Copy(Of T)(ByVal array(,) As T) As T(,)

Dim rows As Integer = array.GetLength(0) - 1

Dim columns As Integer = array.GetLength(1) - 1

Dim copyArray(rows, columns) As T

For i = 0 To rows

For j = 0 To columns

copyArray(i, j) = array(i, j)

Next

Next

Return copyArray

End Function

<Extension()>

Public Function Copy(Of T)(ByVal array() As T) As T()

Dim copyArray(array.Length - 1) As T

For i = 0 To array.Length - 1

copyArray(i) = array(i)

Next

Return copyArray

End Function

<Extension()>

Public Function Copy(Of T)(ByVal queue As Queue(Of T)) As Queue(Of T)

Dim copyQueue As New Queue(Of T)

Dim numberOfElements As Integer = queue.Count

For i = 0 To numberOfElements - 1

copyQueue.Enqueue(queue.ElementAt(i))

Next

Return copyQueue

End Function

<Extension()>

Public Function LinearSearch(ByVal array() As Integer, ByVal value As Integer) As Integer

If array.Length < 1 Then Return -1

For i = 0 To UBound(array)

If array(i) = value Then Return i

Next

Return -1

End Function

<Extension()>

Public Function LinearSearch(ByVal array() As FaceColour, ByVal value As FaceColour) As FaceColour

If array.Length < 1 Then Return -1

For i = 0 To UBound(array)

If array(i) = value Then Return i

Next

Return -1

End Function

<Extension()>

Public Function AnythingElseInArray(ByVal array() As Integer, ByVal value As Integer) As Boolean

If array.Length < 1 Then Return True

For i = 0 To UBound(array)

If array(i) <> value Then Return True

Next

Return False

End Function

<Extension()>

Public Function AnythingElseInArray(ByVal array(,) As Boolean, ByVal value As Boolean) As Boolean

If array.Length < 1 Then Return True

For i = 0 To array.GetLength(0) - 1

For j = 0 To array.GetLength(1) - 1

If array(i, j) <> value Then Return True

Next

Next

Return False

End Function

<Extension()>

Public Function CountNotNothing(Of T)(ByVal array() As T) As Integer

Dim count As Integer = 0

For Each item In array

If item IsNot Nothing Then

count += 1

End If

Next

Return count

End Function

<Extension()>

Public Function IsLastElement(Of T)(array As T(), element As T) As Boolean

Return System.Array.IndexOf(array, element) = array.Length - 1

End Function

''' <summary>

''' Cycles an array round so so that the element at newStartIndex is now at index 0

''' </summary>

''' <param name="indexOfNewStart">0 based index of the item to be at the start of the returned array</param>

<Extension()>

Public Function Rotate(Of T)(ByVal array() As T, ByVal indexOfNewStart As Integer) As T()

Dim rotatedArray(array.Length - 1) As T

For i = 0 To array.Length - 1

rotatedArray(i) = array((indexOfNewStart + i) Mod (array.Length))

Next

Return rotatedArray

End Function

End Module

## InstructionList.vb

Imports RubiksCubeSolver\_v2\_0.Helpers

<Serializable()> Public Class InstructionList

Inherits List(Of Instruction)

Public Sub AddFaceTurn(ByVal face As MoveFaces, ByVal direction As Direction)

If direction = Direction.NO\_CHANGE Then Return

Me.Add(New FaceInstruction(face, direction))

End Sub

Public Sub AddOrientationChange(ByVal axis As Axis, ByVal direction As Direction)

If direction = Direction.NO\_CHANGE Then Return

Me.Add(New CubeInstruction(axis, direction))

End Sub

Public Sub Optimise()

Dim anySimplificationsMade As Boolean = False

Do

anySimplificationsMade = False

Dim ptr As Integer = 0

While ptr < Me.Count - 1

Dim currentInstruction As Instruction = Me.ElementAt(ptr)

Dim nextInstruction As Instruction = Me.ElementAt(ptr + 1)

If currentInstruction.GetType() <> nextInstruction.GetType() Then

ptr += 1

Continue While

End If

If currentInstruction.Move <> nextInstruction.Move Then

ptr += 1

Continue While

End If

Dim directionSum As Integer = Math.Abs((currentInstruction.Direction + nextInstruction.Direction) Mod 4)

If directionSum = 3 Then directionSum = -1

currentInstruction.Direction = directionSum

If currentInstruction.Direction = Direction.NO\_CHANGE Then

Me.RemoveAt(ptr)

ptr -= 1

End If

Me.RemoveAt(ptr + 1)

ptr += 1

anySimplificationsMade = True

End While

Loop While anySimplificationsMade

End Sub

Public Function Copy() As InstructionList

Dim copyList As New InstructionList

For i = 0 To Me.Count - 1

copyList.Add(Me.ElementAt(i))

Next

Return copyList

End Function

Public Overrides Function ToString() As String

Dim rtn As String = ""

For i = 0 To Me.Count - 1

rtn += Me.ElementAt(i).ToString + ", "

Next

Return rtn

End Function

End Class

## Instruction.vb

Imports RubiksCubeSolver\_v2\_0.Helpers

<Serializable()> Public MustInherit Class Instruction

Public Property Move As Integer

Public Property Direction As Direction

Public MustOverride Overrides Function ToString() As String

Public MustOverride Function Copy() As Instruction

End Class

## FaceInstruction.vb

Imports RubiksCubeSolver\_v2\_0.Helpers

<Serializable()> Public Class FaceInstruction

Inherits Instruction

Public Sub New(ByVal face As MoveFaces, ByVal direction As Direction)

Move = face

Me.Direction = direction

End Sub

Public Overrides Function ToString() As String

Dim instructionString As String

Select Case Move

Case MoveFaces.TOP : instructionString = "U"

Case MoveFaces.LEFT : instructionString = "L"

Case MoveFaces.BACK : instructionString = "B"

Case MoveFaces.RIGHT : instructionString = "R"

Case MoveFaces.FRONT : instructionString = "F"

Case MoveFaces.BOTTOM : instructionString = "D"

Case Else

Throw New ArgumentException("Invalid face")

End Select

Select Case Direction

Case Direction.CLOCKWISE : instructionString += ""

Case Direction.ANTICLOCKWISE : instructionString += "'"

Case Direction.HALF\_TURN : instructionString += "2"

Case Else

Throw New ArgumentException("Invalid direction")

End Select

Return instructionString

End Function

Public Overrides Function Copy() As Instruction

Return New FaceInstruction(Me.Move, Me.Direction)

End Function

End Class

## CubeInstruction.vb

Imports RubiksCubeSolver\_v2\_0.Helpers

<Serializable()> Public Class CubeInstruction

Inherits Instruction

Public Sub New(ByVal axis As Axis, ByVal direction As Direction)

Move = axis

Me.Direction = direction

End Sub

Public Overrides Function ToString() As String

Dim instructionString As String

Select Case Move

Case Axis.X : instructionString = "X"

Case Axis.Y : instructionString = "Y"

Case Axis.Z : instructionString = "Z"

Case Else

Throw New ArgumentException("Invalid axis")

End Select

Select Case Direction

Case Direction.CLOCKWISE : instructionString += ""

Case Direction.ANTICLOCKWISE : instructionString += "'"

Case Direction.HALF\_TURN : instructionString += "2"

Case Else

Throw New ArgumentException("Invalid direction")

End Select

Return instructionString

End Function

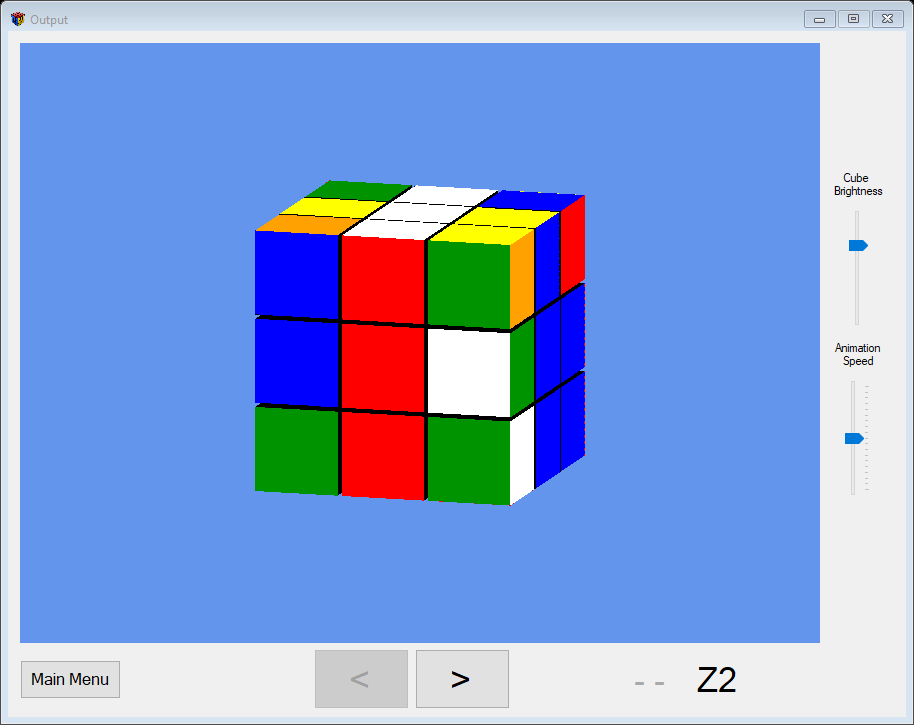
Public Overrides Function Copy() As Instruction

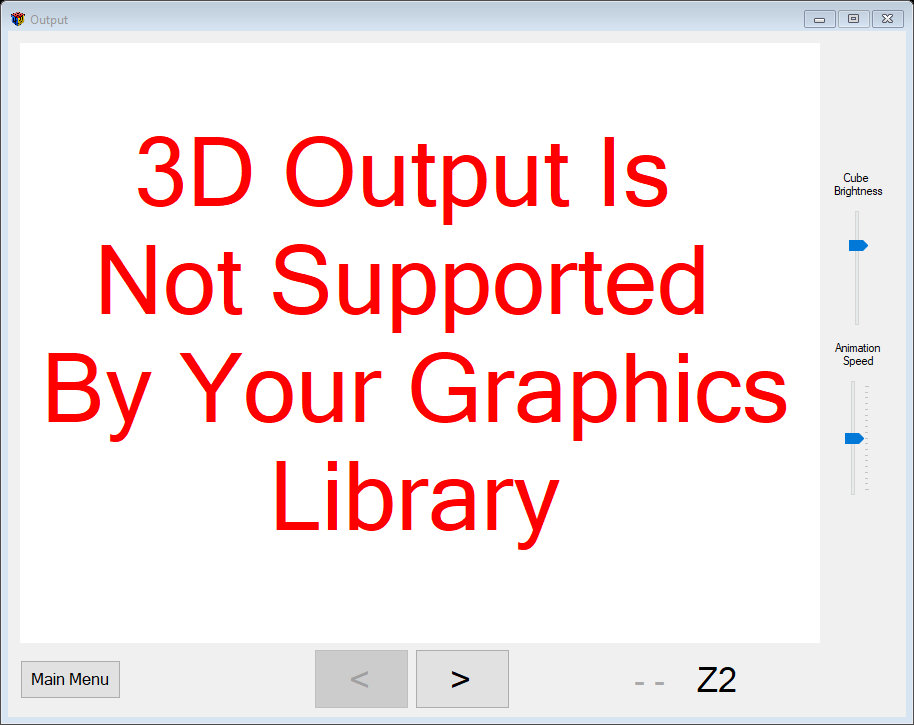
Return New CubeInstruction(Me.Move, Me.Direction)

End Function

End Class

## 3dOutput.vb





Imports OpenTK

Imports OpenTK.Graphics.OpenGL

Imports RubiksCubeSolver\_v2\_0.Helpers.PublicConstants

Imports System.Runtime.Serialization.Formatters.Binary

Public Class \_3DOutput

Const pad As Single = 5 / 100

ReadOnly projection As OpenTK.Matrix4 = Matrix4.CreateOrthographic(4 / 3 \* 7, 7, 1, 1000)

ReadOnly initalPositions() As OpenTK.Vector3

ReadOnly initalRotations() As OpenTK.Quaternion

ReadOnly initalScrambledCube As Cube

ReadOnly instructions As OutputInstructionList

Private shaderProgram As ShaderProgram

Private camera As Camera

Private deltaTime As Single = 0

Private isMouseDown As Boolean

Private currentCube As Cube

Private cubieRotations(26) As OutputBlock

Private showingRotation As Boolean = False

Public Sub New(ByVal cube As Cube)

InitializeComponent()

initalScrambledCube = cube

initalScrambledCube.CurrentOrientation = New CubeOrientation(FaceColour.W, FaceColour.R)

initalPositions = GetInitialPositions()

initalRotations = GetInitialRotations()

camera = New Camera()

instructions = New OutputInstructionList(cube.Instructions)

UpdateInstructionLabels()

currentCube = New Cube(initalScrambledCube)

For i = 0 To 26

cubieRotations(i) = New OutputBlock

Next

End Sub

Public Sub New(ByVal filePath As String)

InitializeComponent()

Dim fStream As New IO.FileStream(filePath, IO.FileMode.Open)

Dim formatter As New BinaryFormatter()

initalScrambledCube = DirectCast(formatter.Deserialize(fStream), Cube)

instructions = New OutputInstructionList(initalScrambledCube.Instructions)

initalPositions = GetInitialPositions()

initalRotations = GetInitialRotations()

camera = New Camera()

UpdateInstructionLabels()

currentCube = New Cube(initalScrambledCube)

For i = 0 To 26

cubieRotations(i) = New OutputBlock

Next

Try

Dim endPtr = GetCurrentInstructionPtr(filePath)

If endPtr = 0 Then Exit Try

For i = 0 To endPtr - 1

If instructions.currentInstructionPtr < instructions.Count Then

DoNextStage()

instructions.currentInstructionPtr += 1

UpdateInstructionLabels()

End If

Next

btnUndo.Enabled = True

Catch ex As ReadUnsuccessfulException

MsgBox(ex.Message)

End Try

End Sub

Private Sub GlControl1\_Load(sender As Object, e As EventArgs) Handles GlControl1.Load

LoadGraphics()

If shaderProgram.Handle = -1 Then

lblError.Visible = True

Return

End If

LoadGeometry()

End Sub

''' <summary> Loads the shaders and sets up the viewport </summary>

Private Sub LoadGraphics()

GL.Viewport(0, 0, GlControl1.Width, GlControl1.Height)

GL.Enable(EnableCap.DepthTest)

GL.DepthFunc(DepthFunction.Lequal)

sldrAmbient.Value = 23

shaderProgram = New ShaderProgram()

End Sub

''' <summary> Loads the lighting for the scene </summary>

Private Sub LoadLighting()

Dim colour As OpenTK.Graphics.Color4 = New OpenTK.Graphics.Color4(255, 255, 255, 255)

Dim ambient As Vector4 = New Vector4(colour.R / 255, colour.G / 255, colour.B / 255, 1)

Dim lightColorLoc As Integer = GL.GetUniformLocation(shaderProgram.Handle, "lightColor")

GL.Uniform4(lightColorLoc, ambient.X, ambient.Y, ambient.Z, ambient.W)

Static ambientStrength As Single = 0.95

ambientStrength = sldrAmbient.Value / 20

Dim ambientStrengthLoc As Integer = GL.GetUniformLocation(shaderProgram.Handle, "ambientStrength")

GL.Uniform1(ambientStrengthLoc, ambientStrength)

End Sub

''' <summary> Loads the vertices and vertexAttributes </summary>

Private Sub LoadGeometry()

Dim vertices3d As Vertex3D() = CubeVertices()

Dim indices As UInteger() = CubeIndices()

shaderProgram.VAO = GL.GenVertexArray()

shaderProgram.VBO = GL.GenBuffer()

shaderProgram.EBO = GL.GenBuffer()

' Bind the Vertex Array Object

GL.BindVertexArray(shaderProgram.VAO)

' copy our vertices to a buffer for OpenGL to use

GL.BindBuffer(BufferTarget.ArrayBuffer, shaderProgram.VBO)

GL.BufferData(BufferTarget.ArrayBuffer, CType(Vertex3D.SizeInBytes \* vertices3d.Length, IntPtr), vertices3d,

BufferUsageHint.StaticDraw)

' copy the indeces to a buffer for OpenGL to use

GL.BindBuffer(BufferTarget.ElementArrayBuffer, shaderProgram.EBO)

GL.BufferData(BufferTarget.ElementArrayBuffer, CType(4 \* indices.Length, IntPtr), indices, BufferUsageHint.StaticDraw)

' set vertex attribute pointers to access them in the shaders

GL.VertexAttribPointer(0, 3, VertexAttribPointerType.Float, False, Vertex3D.SizeInBytes, 0)

GL.EnableVertexAttribArray(0)

GL.VertexAttribPointer(1, 4, VertexAttribPointerType.Float, False, Vertex3D.SizeInBytes, Vector3.SizeInBytes)

GL.EnableVertexAttribArray(1)

GL.VertexAttribPointer(2, 3, VertexAttribPointerType.Float, False, Vertex3D.SizeInBytes,

Vector3.SizeInBytes + Vector4.SizeInBytes)

GL.EnableVertexAttribArray(2)

' Unbind the Vertex Buffer Object

GL.BindBuffer(BufferTarget.ArrayBuffer, 0)

' Unbind the Vertex Array Object

GL.BindVertexArray(0)

GL.PolygonMode(MaterialFace.FrontAndBack, PolygonMode.Fill)

End Sub

Private Function CubeVertices() As Vertex3D()

Return New Vertex3D(0 To 23) {

New Vertex3D(New Vector3(0, 0, 0), Color.DarkOrange, New Vector3(0, 0, -1)),

New Vertex3D(New Vector3(1, 0, 0), Color.DarkOrange, New Vector3(0, 0, -1)),

New Vertex3D(New Vector3(1, 1, 0), Color.DarkOrange, New Vector3(0, 0, -1)),

New Vertex3D(New Vector3(0, 1, 0), Color.DarkOrange, New Vector3(0, 0, -1)),

New Vertex3D(New Vector3(0, 0, 1), Color.Red, New Vector3(0, 0, 1)),

New Vertex3D(New Vector3(1, 0, 1), Color.Red, New Vector3(0, 0, 1)),

New Vertex3D(New Vector3(1, 1, 1), Color.Red, New Vector3(0, 0, 1)),

New Vertex3D(New Vector3(0, 1, 1), Color.Red, New Vector3(0, 0, 1)),

New Vertex3D(New Vector3(0, 0, 0), Color.Green, New Vector3(-1, 0, 0)),

New Vertex3D(New Vector3(1, 0, 0), Color.Blue, New Vector3(1, 0, 0)),

New Vertex3D(New Vector3(1, 1, 0), Color.Blue, New Vector3(1, 0, 0)),

New Vertex3D(New Vector3(0, 1, 0), Color.Green, New Vector3(-1, 0, 0)),

New Vertex3D(New Vector3(0, 0, 1), Color.Green, New Vector3(-1, 0, 0)),

New Vertex3D(New Vector3(1, 0, 1), Color.Blue, New Vector3(1, 0, 0)),

New Vertex3D(New Vector3(1, 1, 1), Color.Blue, New Vector3(1, 0, 0)),

New Vertex3D(New Vector3(0, 1, 1), Color.Green, New Vector3(-1, 0, 0)),

New Vertex3D(New Vector3(0, 0, 0), Color.Yellow, New Vector3(0, -1, 0)),

New Vertex3D(New Vector3(1, 0, 0), Color.Yellow, New Vector3(0, -1, 0)),

New Vertex3D(New Vector3(1, 1, 0), Color.White, New Vector3(0, 1, 0)),

New Vertex3D(New Vector3(0, 1, 0), Color.White, New Vector3(0, 1, 0)),

New Vertex3D(New Vector3(0, 0, 1), Color.Yellow, New Vector3(0, -1, 0)),

New Vertex3D(New Vector3(1, 0, 1), Color.Yellow, New Vector3(0, -1, 0)),

New Vertex3D(New Vector3(1, 1, 1), Color.White, New Vector3(0, 1, 0)),

New Vertex3D(New Vector3(0, 1, 1), Color.White, New Vector3(0, 1, 0))}

End Function

Private Function CubeIndices() As UInteger()

Return New UInteger(0 To 35) {

4, 5, 6, 4, 6, 7,

8, 11, 15, 8, 12, 15,

16, 20, 21, 16, 17, 21,

9, 10, 14, 9, 13, 14,

19, 18, 22, 19, 22, 23,

0, 1, 2, 0, 2, 3}

End Function

''' <summary> gets the inital position vectors for each cubie </summary>

Private Function GetInitialPositions() As Vector3()

Dim corners() As Corner = initalScrambledCube.Corners

Dim edges() As Edge = initalScrambledCube.Edges

Dim Positions(26) As Vector3

For i = 0 To 7

Positions(i) = New Vector3 With {

.X = corners(i).Position.x \* (1 + pad) - 0.5,

.Y = corners(i).Position.y \* (1 + pad) - 0.5,

.Z = corners(i).Position.z \* (1 + pad) - 0.5

}

Next

For i = 0 To 11

Positions(i + 8) = New Vector3 With {

.X = edges(i).Position.x \* (1 + pad) - 0.5,

.Y = edges(i).Position.y \* (1 + pad) - 0.5,

.Z = edges(i).Position.z \* (1 + pad) - 0.5

}

Next

Positions(20) = New Vector3(-0.5, 0.5 + pad, -0.5)

Positions(21) = New Vector3(-1.5 - pad, -0.5, -0.5)

Positions(22) = New Vector3(-0.5, -0.5, -1.5 - pad)

Positions(23) = New Vector3(0.5 + pad, -0.5, -0.5)

Positions(24) = New Vector3(-0.5, -0.5, 0.5 + pad)

Positions(25) = New Vector3(-0.5, -1.5 - pad, -0.5)

Positions(26) = New Vector3(-0.5, -0.5, -0.5)

Return Positions

End Function

''' <summary> gets the inital rotation quaternions for each cubie </summary>

Private Function GetInitialRotations() As Quaternion()

Dim initalBlocks() As Block = initalScrambledCube.CornersAndEdgesAndMiddles.Copy()

Dim initialCubieRotations(26) As Quaternion

Dim cubie As Block

For i = 0 To 19

cubie = initalBlocks(i)

Dim rotationFirstAxis As Quaternion = GetRotationFirstAxis(cubie)

Dim rotationSecondAxis As Quaternion = GetRotationSecondAxis(cubie, rotationFirstAxis)

initialCubieRotations(i) = rotationSecondAxis \* rotationFirstAxis

Next

For i = 20 To 26

initialCubieRotations(i) = Quaternion.Identity

Next

Return initialCubieRotations

End Function

''' <summary> gets the quaternion that puts the primary face correct </summary>

Private Function GetRotationFirstAxis(ByVal cubie As Block) As Quaternion

Dim rotationVector As Vector3 = New Vector3

Select Case cubie.PrimaryFace

Case FaceColour.W

Select Case cubie.Rotation

Case FaceColour.Y : rotationVector.X = 2

Case FaceColour.R : rotationVector.X = 1

Case FaceColour.O : rotationVector.X = -1

Case FaceColour.G : rotationVector.Z = 1

Case FaceColour.B : rotationVector.Z = -1

End Select

Case FaceColour.Y

Select Case cubie.Rotation

Case FaceColour.W : rotationVector.X = 2

Case FaceColour.R : rotationVector.X = -1

Case FaceColour.O : rotationVector.X = 1

Case FaceColour.G : rotationVector.Z = -1

Case FaceColour.B : rotationVector.Z = 1

End Select

Case FaceColour.R

Select Case cubie.Rotation

Case FaceColour.W : rotationVector.X = -1

Case FaceColour.Y : rotationVector.X = 1

Case FaceColour.O : rotationVector.X = 2

Case FaceColour.G : rotationVector.Y = -1

Case FaceColour.B : rotationVector.Y = 1

End Select

Case FaceColour.O

Select Case cubie.Rotation

Case FaceColour.W : rotationVector.X = 1

Case FaceColour.Y : rotationVector.X = -1

Case FaceColour.R : rotationVector.X = 2

Case FaceColour.G : rotationVector.Y = 1

Case FaceColour.B : rotationVector.Y = -1

End Select

End Select

Dim rotationFirstAxis As Quaternion = New Quaternion With {

.X = rotationVector.X \* Math.Sin(Math.Abs(rotationVector.X) \* MathHelper.PiOver2 / 2),

.Y = rotationVector.Y \* Math.Sin(Math.Abs(rotationVector.Y) \* MathHelper.PiOver2 / 2),

.Z = rotationVector.Z \* Math.Sin(Math.Abs(rotationVector.Z) \* MathHelper.PiOver2 / 2)

}

Dim vecSum As Integer = Math.Abs(rotationVector.X) + Math.Abs(rotationVector.Y) + Math.Abs(rotationVector.Z)

rotationFirstAxis.W = Math.Cos(vecSum \* MathHelper.PiOver2 / 2)

If rotationFirstAxis.Length = rotationFirstAxis.W Then

rotationFirstAxis = Quaternion.Identity

End If

Return rotationFirstAxis

End Function

Private Function GetRotationSecondAxis(cubie As Block, rotationFirstAxis As Quaternion) As Quaternion

Dim whiteNormalAfterFirstRotation As Vector3 = New Vector3(0, 1, 0)

Dim redNormalAfterFirstRotation As Vector3 = New Vector3(0, 0, 1)

RecalculateNormalsAfterFirstRotation(whiteNormalAfterFirstRotation, redNormalAfterFirstRotation,

rotationFirstAxis)

Dim desiredWhiteNormal, desiredRedNormal As Vector3

GetNormalsFromCubie(cubie, desiredWhiteNormal, desiredRedNormal)

Dim angle As Single

Dim axis As Vector3

angle = 0

If whiteNormalAfterFirstRotation = desiredWhiteNormal Then

'rotate around white normal

axis = whiteNormalAfterFirstRotation

angle = GetAngleForInitialRotation(axis, redNormalAfterFirstRotation, desiredRedNormal)

ElseIf redNormalAfterFirstRotation = desiredRedNormal Then

'rotate around red normal

axis = redNormalAfterFirstRotation

angle = GetAngleForInitialRotation(axis, whiteNormalAfterFirstRotation, desiredWhiteNormal)

End If

Dim rotationSecondAxis As Quaternion

rotationSecondAxis = Quaternion.Identity

If angle <> 0 Then

rotationSecondAxis.X = Math.Abs(axis.X) \* Math.Sin(angle / 2)

rotationSecondAxis.Y = Math.Abs(axis.Y) \* Math.Sin(angle / 2)

rotationSecondAxis.Z = Math.Abs(axis.Z) \* Math.Sin(angle / 2)

rotationSecondAxis.W = Math.Cos(angle / 2)

Else

rotationSecondAxis = Quaternion.Identity

End If

Return rotationSecondAxis

End Function

Private Sub RecalculateNormalsAfterFirstRotation(ByRef whiteNormal As Vector3, ByRef redNormal As Vector3, ByVal firstRotation As Quaternion)

Dim rotationMatrix As Matrix4

rotationMatrix = Matrix4.CreateTranslation(New Vector3(-0.5, -0.5, -0.5))

rotationMatrix \*= Matrix4.CreateFromQuaternion(firstRotation)

rotationMatrix \*= Matrix4.CreateTranslation(-(New Vector3(-0.5, -0.5, -0.5)))

Dim myRotMatrix As Matrices.Matrix4x4

myRotMatrix = New Matrix4x4(rotationMatrix)

myRotMatrix.Row1.w = 0

myRotMatrix.Row2.w = 0

myRotMatrix.Row3.w = 0

Dim myRedNormal, myWhiteNormal As Matrices.Vector4x1

myWhiteNormal = New Vector4x1(New Vector4(whiteNormal, 1))

myRedNormal = New Vector4x1(New Vector4(redNormal, 1))

myWhiteNormal = myRotMatrix \* myWhiteNormal

myRedNormal = myRotMatrix \* myRedNormal

whiteNormal = New Vector3(Math.Round(myWhiteNormal.x, 4), Math.Round(myWhiteNormal.y, 4),

Math.Round(myWhiteNormal.z, 4))

redNormal = New Vector3(Math.Round(myRedNormal.x, 4), Math.Round(myRedNormal.y, 4),

Math.Round(myRedNormal.z, 4))

End Sub

Private Sub GetNormalsFromCubie(ByVal cubie As Block, ByRef whiteNormal As Vector3, ByRef redNormal As Vector3)

Dim secondaryRotation, secondaryFace As FaceColour

If cubie.GetType() = GetType(Corner) Then

Dim cornerCubie As Corner = cubie

secondaryRotation = cornerCubie.SecondaryRotation

secondaryFace = cornerCubie.SecondaryFace

ElseIf cubie.GetType() = GetType(Edge) Then

For Each colour In cubie.Colours

If colour <> cubie.PrimaryFace Then

secondaryFace = colour

Exit For

End If

Next

For Each face As FaceColour In initalScrambledCube.FaceColours

If cubie.CorrectRotation(secondaryFace, face) And cubie.OnFace(face, initalScrambledCube) Then

secondaryRotation = face

Exit For

End If

Next

End If

Dim whiteSet, redSet As Boolean

whiteSet = False

redSet = False

Select Case cubie.PrimaryFace

Case FaceColour.W, FaceColour.Y

Select Case cubie.Rotation

Case FaceColour.W : whiteNormal = New Vector3(0, 1, 0)

Case FaceColour.Y : whiteNormal = New Vector3(0, -1, 0)

Case FaceColour.R : whiteNormal = New Vector3(0, 0, 1)

Case FaceColour.O : whiteNormal = New Vector3(0, 0, -1)

Case FaceColour.G : whiteNormal = New Vector3(-1, 0, 0)

Case FaceColour.B : whiteNormal = New Vector3(1, 0, 0)

End Select

If cubie.PrimaryFace = FaceColour.Y Then

whiteNormal = -whiteNormal

End If

whiteSet = True

Case FaceColour.R, FaceColour.O

Select Case cubie.Rotation

Case FaceColour.W : redNormal = New Vector3(0, 1, 0)

Case FaceColour.Y : redNormal = New Vector3(0, -1, 0)

Case FaceColour.R : redNormal = New Vector3(0, 0, 1)

Case FaceColour.O : redNormal = New Vector3(0, 0, -1)

Case FaceColour.G : redNormal = New Vector3(-1, 0, 0)

Case FaceColour.B : redNormal = New Vector3(1, 0, 0)

End Select

If cubie.PrimaryFace = FaceColour.O Then

redNormal = -redNormal

End If

redSet = True

End Select

Select Case secondaryFace

Case FaceColour.R, FaceColour.O

Select Case secondaryRotation

Case FaceColour.W : redNormal = New Vector3(0, 1, 0)

Case FaceColour.Y : redNormal = New Vector3(0, -1, 0)

Case FaceColour.R : redNormal = New Vector3(0, 0, 1)

Case FaceColour.O : redNormal = New Vector3(0, 0, -1)

Case FaceColour.G : redNormal = New Vector3(-1, 0, 0)

Case FaceColour.B : redNormal = New Vector3(1, 0, 0)

End Select

If secondaryFace = FaceColour.O Then

redNormal = -redNormal

End If

redSet = True

Case FaceColour.G, FaceColour.B

Dim greenNormal As Vector3

Select Case secondaryRotation

Case FaceColour.W : greenNormal = New Vector3(0, 1, 0)

Case FaceColour.Y : greenNormal = New Vector3(0, -1, 0)

Case FaceColour.R : greenNormal = New Vector3(0, 0, 1)

Case FaceColour.O : greenNormal = New Vector3(0, 0, -1)

Case FaceColour.G : greenNormal = New Vector3(-1, 0, 0)

Case FaceColour.B : greenNormal = New Vector3(1, 0, 0)

End Select

If secondaryFace = FaceColour.B Then

greenNormal = -greenNormal

End If

If whiteSet Then

redNormal = Vector3.Cross(whiteNormal, greenNormal)

ElseIf redSet Then

whiteNormal = -Vector3.Cross(redNormal, greenNormal)

Else

Throw New NotImplementedException("GetNormalsFromCubie: primary normal not set!")

End If

End Select

End Sub

Private Function GetAngleForInitialRotation(axis As Vector3, currentPosition As Vector3, desiredPosition As Vector3) As Single

Dim angle As Single

If currentPosition = desiredPosition Then

angle = 0

ElseIf Math.Abs(Vector3.Dot(currentPosition, desiredPosition)) = 1 Then

angle = MathHelper.Pi

Else

'collapse the vectors onto 1 2D plane, perpendicular to axis of rotation

Dim current2d, correct2d As Vector2

If axis.X <> 0 Then

current2d = New Vector2(-currentPosition.Z, currentPosition.Y)

correct2d = New Vector2(-desiredPosition.Z, desiredPosition.Y)

ElseIf axis.Y <> 0 Then

current2d = New Vector2(currentPosition.X, -currentPosition.Z)

correct2d = New Vector2(desiredPosition.X, -desiredPosition.Z)

ElseIf axis.Z <> 0 Then

current2d = New Vector2(currentPosition.X, currentPosition.Y)

correct2d = New Vector2(desiredPosition.X, desiredPosition.Y)

End If

If current2d.Y <> 0 Then

If current2d.Y = correct2d.X Then

angle = -MathHelper.PiOver2

Else

angle = MathHelper.PiOver2

End If

ElseIf current2d.X <> 0 Then

If current2d.X = correct2d.Y Then

angle = MathHelper.PiOver2

Else

angle = -MathHelper.PiOver2

End If

End If

End If

Return angle

End Function

''' <summary> Called by GlControl1.Invalidate() - updates the GL viewport </summary>

Private Sub GlControl1\_Paint(sender As Object, e As PaintEventArgs) Handles GlControl1.Paint

GL.ClearColor(Color.CornflowerBlue)

GL.Clear(ClearBufferMask.ColorBufferBit Or ClearBufferMask.DepthBufferBit)

If shaderProgram.Handle = -1 Then Return

Static previousFrame As Decimal

UpdateDeltaTime(previousFrame)

Dim camX As Single = Math.Sin(camera.Rotation / 10) \* Camera.RADIUS

Dim camZ As Single = Math.Cos(camera.Rotation / 10) \* Camera.RADIUS

Dim view As Matrix4 = Matrix4.LookAt(New Vector3(camX, camera.Pitch, camZ), New Vector3(0, 0, 0),

New Vector3(0, 1, 0))

'sends view matrix to shaders

GL.Uniform3(GL.GetUniformLocation(shaderProgram.Handle, "viewPos"), camX, camera.Pitch, camZ)

RenderCube(view, projection)

camera.Update(False, deltaTime, isMouseDown)

previousFrame = DateTime.Now.Ticks / 10000

GlControl1.SwapBuffers() 'swaps the buffer being drawn to and the buffer being shown (updates the viewport)

GlControl1.Invalidate() ' calls paint

End Sub

Private Sub UpdateDeltaTime(ByVal previousFrame As Decimal)

Dim currentFrame As Decimal

Static firsttime As Boolean = True

If firsttime Then

deltaTime = 5

firsttime = False

End If

Do

currentFrame = DateTime.Now.Ticks / 10000

deltaTime = currentFrame - previousFrame

Loop While deltaTime = 0

End Sub

''' <summary> Renders a cube onto the secondary buffer </summary>

Private Sub RenderCube(ByRef view As Matrix4, ByRef projection As Matrix4)

shaderProgram.Use()

LoadLighting()

GL.UniformMatrix4(GL.GetUniformLocation(shaderProgram.Handle, "projection"), False, projection)

GL.UniformMatrix4(GL.GetUniformLocation(shaderProgram.Handle, "view"), False, view)

GL.Enable(EnableCap.DepthTest)

GL.DepthFunc(DepthFunction.Less)

Static displayAnglePercent As Double = 0.0

Dim rotationSpeed As Single = sldrSpeed.Value / 1000

If showingRotation Then

displayAnglePercent += deltaTime \* rotationSpeed

If displayAnglePercent >= 1 Then

displayAnglePercent = 1

showingRotation = False

btnNext.Enabled = True

btnUndo.Enabled = True

End If

Else

displayAnglePercent = 0

End If

For i = 0 To 26

RenderCubie(i, displayAnglePercent)

Next

End Sub

''' <summary> renders the cubie represented by a given index </summary>

Private Sub RenderCubie(ByVal index As Integer, ByVal displayAnglePercent As Double)

GL.BindVertexArray(shaderProgram.VAO)

GL.Uniform1(GL.GetUniformLocation(shaderProgram.Handle, "pad"), pad)

Dim model As Matrix4 = Matrix4.CreateTranslation(initalPositions(26))

model \*= Matrix4.CreateFromQuaternion(initalRotations(index))

model \*= Matrix4.CreateTranslation(-initalPositions(26))

model \*= Matrix4.CreateTranslation(initalPositions(index))

'model matrix passed to shaders to set colours in correct places

Dim modelUniformLoc As Integer = GL.GetUniformLocation(shaderProgram.Handle, "colourMat")

GL.UniformMatrix4(modelUniformLoc, False, model)

model \*= cubieRotations(index).Rotations.GetTotalMatrix(displayAnglePercent)

GL.UniformMatrix4(GL.GetUniformLocation(shaderProgram.Handle, "model"), False, model)

GL.DrawElements(PrimitiveType.Triangles, 36, DrawElementsType.UnsignedInt, 0)

End Sub

Private Sub btnUndo\_Hover(sender As Object, e As EventArgs) Handles btnUndo.MouseEnter, btnUndo.MouseHover

lblPrev.ForeColor = Color.Red

End Sub

Private Sub btnNext\_Hover(sender As Object, e As EventArgs) Handles btnNext.MouseEnter, btnNext.MouseHover

lblNext.ForeColor = Color.Red

End Sub

Private Sub btnUndo\_MouseLeave(sender As Object, e As EventArgs) Handles btnUndo.MouseLeave

lblPrev.ForeColor = Color.DarkGray

End Sub

Private Sub btnNext\_MouseLeave(sender As Object, e As EventArgs) Handles btnNext.MouseLeave

lblNext.ForeColor = Color.Black

End Sub

Private Sub UpdateInstructionLabels()

If instructions.PreviousInstruction IsNot Nothing Then

lblPrev.Text = instructions.PreviousInstruction.ToString()

Else

lblPrev.Text = "- -"

btnUndo.Enabled = False

End If

If instructions.CurrentInstruction IsNot Nothing Then

lblNext.Text = instructions.CurrentInstruction.ToString()

Else

lblNext.Text = "- -"

btnNext.Enabled = False

End If

lblPrev.Refresh()

lblNext.Refresh()

End Sub

Private Sub GlControl1\_MouseDown(sender As Object, e As MouseEventArgs) Handles GlControl1.MouseDown

Dim mousePos As New Mouse

If Not isMouseDown Then

mousePos.X = OpenTK.Input.Mouse.GetCursorState.X

mousePos.Y = OpenTK.Input.Mouse.GetCursorState.Y

End If

isMouseDown = True

camera.Update(True, deltaTime, isMouseDown, mousePos)

End Sub

Private Sub GlControl1\_MouseUp(sender As Object, e As MouseEventArgs) Handles GlControl1.MouseUp

isMouseDown = False

End Sub

Private Sub btnUndo\_Click(sender As Object, e As EventArgs) Handles btnUndo.Click

If shaderProgram.Handle <> -1 Then btnUndo.Enabled = False

If instructions.currentInstructionPtr > 0 Then

UndoLastStage()

instructions.currentInstructionPtr -= 1

UpdateInstructionLabels()

End If

If lblNext.Text <> "- -" Then

btnNext.Enabled = True

End If

End Sub

Private Sub btnNext\_Click(sender As Object, e As EventArgs) Handles btnNext.Click

If shaderProgram.Handle <> -1 Then btnNext.Enabled = False

If instructions.currentInstructionPtr < instructions.Count Then

DoNextStage()

instructions.currentInstructionPtr += 1

UpdateInstructionLabels()

End If

If lblPrev.Text <> "- -" Then

btnUndo.Enabled = True

End If

End Sub

Private Sub DoNextStage()

DoInstruction(instructions.CurrentInstruction)

End Sub

Private Sub UndoLastStage()

Dim reverseInstruction As Instruction

reverseInstruction = instructions.PreviousInstruction.Copy()

reverseInstruction.Direction = -reverseInstruction.Direction

DoInstruction(reverseInstruction)

End Sub

Private Sub DoInstruction(ByVal instruction As Instruction)

showingRotation = True

If instruction.GetType() = GetType(FaceInstruction) Then

RotateFace(instruction.Move, instruction.Direction)

ElseIf instruction.GetType = GetType(CubeInstruction) Then

RotateCube(instruction.Move, instruction.Direction)

Else

Throw New ArgumentException("Invalid instruction type")

End If

End Sub

''' <summary> addds the rotation matrices to the correct rotation stores in order to rotate a given face </summary>

Private Sub RotateFace(ByVal face As MoveFaces, ByVal direction As Direction)

Dim axis As Vector3

Select Case face

Case MoveFaces.RIGHT : axis = New Vector3(1, 0, 0)

Case MoveFaces.LEFT : axis = New Vector3(-1, 0, 0)

Case MoveFaces.TOP : axis = New Vector3(0, 1, 0)

Case MoveFaces.BOTTOM : axis = New Vector3(0, -1, 0)

Case MoveFaces.FRONT : axis = New Vector3(0, 0, 1)

Case MoveFaces.BACK : axis = New Vector3(0, 0, -1)

Case Else

Throw New ArgumentException("invalid face")

End Select

For i = 0 To 26

If Not currentCube.CornersAndEdgesAndMiddles(i).OnFace(currentCube.FaceColours(face), currentCube) Then Continue For

cubieRotations(i).Rotations.Add(axis, IIf(Math.Abs(direction) =

Direction.HALF\_TURN, -direction, direction) \* MathHelper.PiOver2)

cubieRotations(i).BeingRotated = True

currentCube.RotateFace(currentCube.FaceColours(face), direction)

Next

End Sub

''' <summary> addds the rotation matrices to the correct rotation stores in order to rotate the cube </summary>

Private Sub RotateCube(ByVal axis As Axis, ByVal direction As Direction)

Dim axisVector As Vector3

Select Case axis

Case Axis.X : axisVector = New Vector3(1, 0, 0)

Case Axis.Y : axisVector = New Vector3(0, 1, 0)

Case Axis.Z : axisVector = New Vector3(0, 0, 1)

Case Else

Throw New ArgumentException("Invalid axis")

End Select

Dim angle As Single = MathHelper.PiOver2 \* IIf(Math.Abs(direction) = Direction.HALF\_TURN, -direction, direction)

For i = 0 To 26

cubieRotations(i).Rotations.Add(axisVector, angle)

Next

currentCube.Rotate(direction, axis)

End Sub

Private Sub UpdateOrientation(ByVal axis As Axis, ByVal direction As Direction)

Dim faces() As FaceColour = FacesAroundAxisClockwise(axis)

If axis <> Axis.Y Then currentCube.TopFace = faces((Array.IndexOf(faces, currentCube.TopFace) + 4 + direction) Mod 4)

If axis <> Axis.Z Then currentCube.FrontFace = faces((Array.IndexOf(faces, currentCube.FrontFace) + 4 + direction) Mod 4)

End Sub

Private Function FacesAroundAxisClockwise(ByVal axis As Axis) As FaceColour()

Dim faceColours() As FaceColour = Helpers.GetFaceColoursFromOrientation(currentCube.CurrentOrientation)

Select Case axis

Case Axis.X

Return {faceColours(MoveFaces.FRONT), faceColours(MoveFaces.TOP), faceColours(MoveFaces.BACK),

faceColours(MoveFaces.BOTTOM)}

Case Axis.Y

Return {faceColours(MoveFaces.FRONT), faceColours(MoveFaces.LEFT), faceColours(MoveFaces.BACK),

faceColours(MoveFaces.RIGHT)}

Case Axis.Z

Return {faceColours(MoveFaces.TOP), faceColours(MoveFaces.RIGHT), faceColours(MoveFaces.BOTTOM),

faceColours(MoveFaces.LEFT)}

Case Else

Throw New ArgumentException("Invalid Axis")

End Select

End Function

Private Sub \_3DOutput\_FormClosed(sender As Object, e As FormClosedEventArgs) Handles MyBase.FormClosed

shaderProgram.Dispose()

End Sub

Private Sub btnMain\_Click(sender As Object, e As EventArgs) Handles btnMain.Click

Try

GlControl1.Visible = False

SaveRoutine()

Catch ex As SaveCancelledException

GlControl1.Visible = True

Return

End Try

Dim main As New MainMenu

main.Show()

Me.Close()

End Sub

Private Sub SaveRoutine()

Dim save As MsgBoxResult = MsgBox("Do you want to save?" & vbNewLine & vbNewLine &

"Yes = save and exit" & vbNewLine &

"No = exit without saving" & vbNewLine &

"Cancel = don't exit", MsgBoxStyle.YesNoCancel, "Save?")

If save = MsgBoxResult.Yes Then

Try

SaveCube()

Catch ex As TaskCanceledException

Throw New SaveCancelledException

Catch ex As WriteUnsuccessfulException

btnMain.PerformClick()

Throw New SaveCancelledException

End Try

ElseIf save = MsgBoxResult.Cancel Then

Throw New SaveCancelledException

End If

End Sub

Private Sub SaveCube()

Dim fileBrowser As New SaveFileDialog()

fileBrowser.Filter = "cube files (\*.cube)|\*.cube"

fileBrowser.ShowDialog()

Dim filePath As String = fileBrowser.FileName

If filePath = Nothing Then Throw New TaskCanceledException

initalScrambledCube.Save(filePath)

WriteCurrentInstructionPtr(filePath)

End Sub

Private Sub WriteCurrentInstructionPtr(ByVal filePath As String)

If filePath.Substring(filePath.Length - 5) <> ".cube" Then filePath += ".cube"

filePath += ".ptr"

Try

Dim fStream As New IO.FileStream(filePath, IO.FileMode.Create)

Dim formatter As New BinaryFormatter()

formatter.Serialize(fStream, instructions.currentInstructionPtr)

fStream.Close()

Catch ex As Exception

Throw New WriteUnsuccessfulException("CurrentInstructionPtr attempted to be written before cube file")

End Try

End Sub

Private Function GetCurrentInstructionPtr(ByVal filePath As String) As Integer

filePath += ".ptr"

Try

Dim fStream As New IO.FileStream(filePath, IO.FileMode.Open)

Dim formatter As New BinaryFormatter()

Return formatter.Deserialize(fStream)

Catch ex As Exception

Throw New ReadUnsuccessfulException(".ptr file not found, cannot load saved instruction")

End Try

End Function

End Class

## Camera.vb

Public Class Camera

Public Const FRONT\_ANGLE As Single = -63.0

Public Const RADIUS As Single = 10

Public Property Pitch As Single

Public Property Rotation As Single

Public Sub New()

Rotation = -60.0

Pitch = 2

End Sub

Public Sub Update(ByVal PositionUpdated As Boolean, ByVal deltaTime As Single, ByVal mouseDown As Boolean, Optional MousePos As Mouse = Nothing)

Static StoredMousePos As New Mouse

If PositionUpdated = True Then

StoredMousePos.X = MousePos.X

StoredMousePos.Y = MousePos.Y

End If

Dim cameraSpeed As Single = 0.07 \* deltaTime

If Not mouseDown Then

If Pitch > 2 + (cameraSpeed \* 2.2) Then

Pitch -= cameraSpeed \* 2.2

ElseIf Pitch < 2 - (cameraSpeed \* 2.2) Then

Pitch += cameraSpeed \* 2.2

ElseIf Pitch >= 2 - (cameraSpeed \* 2.2) And Pitch <= 2 + (cameraSpeed \* 2.2) Then

Pitch = 2

End If

End If

If mouseDown Then

Dim xPosition As Integer = OpenTK.Input.Mouse.GetCursorState.X

Dim yPosition As Integer = OpenTK.Input.Mouse.GetCursorState.Y

Dim xOffset As Integer = xPosition - StoredMousePos.X

Dim yOffset As Integer = yPosition - StoredMousePos.Y

Dim sensitivity As Single = 0.1

StoredMousePos.X = xPosition

StoredMousePos.Y = yPosition

If Pitch <= 25 And Pitch >= -20 Then

Pitch += (yOffset \* sensitivity)

ElseIf Pitch > 25 Then

Pitch = 25

ElseIf Pitch < -20 Then

Pitch = -20

End If

If (Rotation - (xOffset \* sensitivity \* 0.5)) > (FRONT\_ANGLE - 6) And

(Rotation - (xOffset \* sensitivity \* 0.5)) < (FRONT\_ANGLE + 6) Then

Rotation -= (xOffset \* sensitivity \* 0.5)

End If

End If

End Sub

End Class

## Vertex3D.vb

Imports OpenTK

Structure Vertex3D

Public Shared ReadOnly SizeInBytes As Integer = (Vector3.SizeInBytes + Vector4.SizeInBytes + Vector3.SizeInBytes)

Public position As Vector3

Public colour As Vector4

Public normal As Vector3

Public Property Color() As Color

Get

Return Drawing.Color.FromArgb(CInt(colour.W \* 255), CInt(colour.X \* 255), CInt(colour.Y \* 255), CInt(colour.Z \* 255))

End Get

Set(ByVal value As Color)

Me.colour = New Vector4(value.R / 255, value.G / 255, value.B / 255, value.A / 255)

End Set

End Property

Public Sub New(ByVal position As Vector3, ByVal colour As Vector4, ByVal normal As Vector3)

Me.position = position

Me.colour = colour

Me.normal = normal

End Sub

Public Sub New(ByVal position As Vector3, ByVal colour As Color, ByVal normal As Vector3)

Me.position = position

'Converts colour(ARGB) to vector4(R, G, B, A)

Me.colour = New Vector4(colour.R / 255, colour.G / 255, colour.B / 255, colour.A / 255)

Me.normal = normal

End Sub

End Structure

## ShaderProgram.vb

Imports OpenTK.Graphics.OpenGL

Imports System.IO

Public Class ShaderProgram

Private program As Integer

Public ReadOnly Property Handle() As Integer

Get

Return program

End Get

End Property

Public Property VBO As Integer

Public Property VAO As Integer

Public Property EBO As Integer

Sub New()

Dim versionNo As Integer = CInt(CStr(GL.GetString(StringName.Version)(0) + GL.GetString(StringName.Version)(2)))

Dim versionString As String = "330"

If versionNo >= 33 Then

versionString = "330"

ElseIf versionNo >= 21 Then

versionString = "120"

Else

MsgBox("3D output is not supported by your graphics library")

program = -1

versionString = "-1"

Return

End If

Dim vertexShader As Integer

vertexShader = GL.CreateShader(ShaderType.VertexShader)

GL.ShaderSource(vertexShader, "#version " + versionString + My.Resources.vertexShader)

GL.CompileShader(vertexShader)

Console.WriteLine("Vertex Shader Success:" & GL.GetShaderInfoLog(vertexShader).ToString())

Dim fragmentShader As Integer

fragmentShader = GL.CreateShader(ShaderType.FragmentShader)

GL.ShaderSource(fragmentShader, "#version " + versionString + My.Resources.fragmentShader)

GL.CompileShader(fragmentShader)

Console.WriteLine("Fragment Shader Success:" & GL.GetShaderInfoLog(fragmentShader).ToString())

program = GL.CreateProgram()

GL.AttachShader(program, vertexShader)

GL.AttachShader(program, fragmentShader)

GL.LinkProgram(program)

Console.WriteLine("Shader Program Success:" & GL.GetProgramInfoLog(program).ToString())

GL.DeleteShader(vertexShader)

GL.DeleteShader(fragmentShader)

End Sub

Public Sub Use()

GL.UseProgram(program)

End Sub

Public Sub Dispose()

If Handle = -1 Then Return

GL.DeleteVertexArray(VAO)

GL.DeleteBuffer(VBO)

GL.DeleteBuffer(EBO)

End Sub

End Class

## RotationStore.vb

Imports OpenTK

Public Class RotationStore

Public Property PreviousRotations As Matrix4 = Matrix4.Identity

Public Property CurrentAxis As Vector3

Public Property CurrentMaxAngle As Single

Public Sub Add(ByVal axis As Vector3, ByVal maxAngle As Single)

If CurrentAxis <> Nothing Then

PreviousRotations \*= Matrix4.CreateFromAxisAngle(CurrentAxis, CurrentMaxAngle)

End If

CurrentAxis = axis

CurrentMaxAngle = maxAngle

End Sub

Public Function GetTotalMatrix(ByVal displayAnglePercentage As Single) As Matrix4

If displayAnglePercentage >= 1 Then FinishCurrentRotation()

Dim rotationMatrix As Matrix4 = Matrix4.Identity

rotationMatrix \*= PreviousRotations

If CurrentAxis <> Nothing Then

rotationMatrix \*= Matrix4.CreateFromAxisAngle(CurrentAxis, displayAnglePercentage \* CurrentMaxAngle)

End If

Return rotationMatrix

End Function

Private Sub FinishCurrentRotation()

If CurrentAxis <> Nothing Then

PreviousRotations \*= Matrix4.CreateFromAxisAngle(CurrentAxis, CurrentMaxAngle)

End If

CurrentAxis = Nothing

CurrentMaxAngle = Nothing

End Sub

End Class

## OutputBlock.vb

Public Class OutputBlock

Public Property Rotations As New RotationStore

Public Property BeingRotated As Boolean

Public Sub New()

BeingRotated = False

End Sub

End Class

## OutputInstructionList.vb

Public Class OutputInstructionList

Inherits List(Of Instruction)

Public Property currentInstructionPtr As Integer

Public Sub New(ByVal instructions As InstructionList)

For i = 0 To instructions.Count - 1

Me.Add(instructions.ElementAt(i))

Next

End Sub

Public ReadOnly Property CurrentInstruction() As Instruction

Get

Try

Return ElementAt(currentInstructionPtr)

Catch

Return Nothing

End Try

End Get

End Property

Public ReadOnly Property PreviousInstruction() As Instruction

Get

Try

Return ElementAt(currentInstructionPtr - 1)

Catch

Return Nothing

End Try

End Get

End Property

End Class

## Mouse.vb

Public Class Mouse

Private \_x As Integer

Public Property X() As Integer

Get

Return \_x

End Get

Set(ByVal value As Integer)

\_x = value

End Set

End Property

Private \_y As Integer

Public Property Y() As Integer

Get

Return \_y

End Get

Set(ByVal value As Integer)

\_y = value

End Set

End Property

End Class

## Exceptions.vb

Public Class StageNotSuccessfulException

Inherits Exception

Public Sub New(ByVal message As String)

MyBase.New(message)

End Sub

Public Sub New()

MyBase.New()

End Sub

End Class

Public Class WriteUnsuccessfulException

Inherits Exception

Public Sub New(ByVal message As String)

MyBase.New(message)

End Sub

Public Sub New()

MyBase.New()

End Sub

End Class

Public Class ReadUnsuccessfulException

Inherits Exception

Public Sub New(ByVal message As String)

MyBase.New(message)

End Sub

Public Sub New()

MyBase.New()

End Sub

End Class

Public Class SaveCancelledException

Inherits Exception

Public Sub New(ByVal message As String)

MyBase.New(message)

End Sub

Public Sub New()

MyBase.New()

End Sub

End Class

## VertexShader.txt

//

#if \_\_VERSION\_\_ == 330

layout (location = 0) in vec3 position; // Vertex Attribute Pointer 0

layout (location = 1) in vec4 color; // VattPointer 1

layout (location = 2) in vec3 normal; // VAttPointer 2

uniform mat4 model;

uniform mat4 view;

uniform mat4 projection;

out vec4 vertexColor;

out vec3 pos;

void main()

{

gl\_Position = projection \* view \* model \* vec4(position, 1.0f);

vertexColor = color;

pos = position;

}

#endif

#if \_\_VERSION\_\_ == 120

attribute vec3 position; // Vertex Attribute Pointer 0

attribute vec4 color; // VAttPointer 1

attribute vec3 normal; // VAttPointer 2

uniform mat4 model;

uniform mat4 view;

uniform mat4 projection;

varying vec4 vertexColor;

varying vec3 pos;

void main()

{

gl\_Position = projection \* view \* model \* vec4(position, 1.0f);

vertexColor = color;

pos = position;

}

#endif

## FragmentShader.txt

//

#if \_\_VERSION\_\_ == 330

// Input variables from the vertex shader (same name and type)

in vec4 vertexColor;

in vec3 pos;

uniform vec4 lightColor;

uniform float ambientStrength;

uniform float pad;

uniform mat4 colourMat;

out vec4 color;

void main()

{

vec3 worldPosition = vec3(colourMat \* vec4(pos, 1.0f));

if (((worldPosition.x <= -1.499999f - pad) || (worldPosition.x >= 1.499999f + pad)) ||

((worldPosition.y <= -1.499999f - pad) || (worldPosition.y >= 1.499999f + pad)) ||

((worldPosition.z <= -1.499999f - pad) || (worldPosition.z >= 1.499999f + pad))){

vec3 ambient = ambientStrength \* lightColor.xyz;

vec3 result = ambient \* vertexColor.xyz;

color = vec4(result, 1.0f);

}

}

#endif

#if \_\_VERSION\_\_ == 120

// Input variables from the vertex shader (same name and type)

varying vec4 vertexColor;

varying vec3 pos;

uniform vec4 lightColor;

uniform float pad;

uniform float ambientStrength;

uniform mat4 colourMat;

void main()

{

vec3 worldPosition = vec3(colourMat \* vec4(pos, 1.0f));

if (((worldPosition.x <= -1.499999f - pad) || (worldPosition.x >= 1.499999f + pad)) ||

((worldPosition.y <= -1.499999f - pad) || (worldPosition.y >= 1.499999f + pad)) ||

((worldPosition.z <= -1.499999f - pad) || (worldPosition.z >= 1.499999f + pad))){

vec3 ambient = ambientStrength \* lightColor.xyz;

vec3 result = ambient \* vertexColor.xyz ;

gl\_FragColor = vec4(result, 1.0f);

}

}

#endif