4D Visualiser

Technical Guide

Student Name:	Liam Tuite	**Student Number:**	13389266
Supervisor:	Liam Tuohey	**Date:**	22/05/17

Abstract

4D Visualiser is a tool that provides a graphical display of 3D and 4D regular geometric objects. The user can choose from a list of 3D or 4D shapes to looks at and has a number of customisation options that change the appearance of the shape shown, such as its colour and rotation. The main use of this program is to provide a currently non-existent interface for displaying higher-dimensional geometry. This can be used by researchers in many branches of mathematics. The project also investigates the potential of abstract functionality in perspective projection in higher dimensions.

This document covers the technical work that has gone into the project, including the design and implementation of the software as well as some of the problems encountered and conclusions drawn from the overall process. It is not a guide on how to use the program.

Contents

1. Background

There have been many steps involved in producing a working program based on the project's concept over the course of 7 months. A significant portion of this work was research. Linking the general idea of the project to a planned system architecture could not be done without sufficient reading and practice in certain areas. I took on the project idea with very little knowledge in some of the disciplines involved in the work, so I knew from the beginning that there would be a very steep learning curve. This section of the report will give an outline the problem I attempted to solve as well as an overview of some of the research involved through all stages of the project.

1.1 Motivation

I started off considering a few projects on applied mathematics, and I found higher-dimensional shapes (more than 3 dimensions) very interesting to look at. Though it is easy to find a .gif or some other form of image of a 4D shape, I couldn't find any resources online that allow for user interaction. I could look at shapes rotating around a set rotation plane, of set colouring and at a set location relative to the viewpoint. I thought that if there are so many different images of these shapes made by different people then I could surely animate them myself and allow other people to view them and interact with the shapes to some degree.

I had no real knowledge about computer graphics or what mathematics are involved, nor was I experienced in designing or building user interfaces in Java. I thought that this project would be a good opportunity to expand on my knowledge in computer programming and to learn about graphics. So I set about finding out about what mathematics were needed for displaying 4D imagery on a screen and seeing if there were parts of the process that I could abstract and simplify.

1.2 Research

The relevant knowledge I had preceding this project was so limited that I first had to figure out what it was that I didn't know. I had no clue about the idea of projection involved in displaying imagery in 3-space on a 2D screen. I had to find out as much as I could, so I researched specific areas I deemed necessary for the project.

1.2.1 GUI Building in Java

I spent some time following online tutorials and practicing simple Java GUI programs. These involved adding user controls like labels, text fields, buttons, checkboxes, selection menus and sliders. I designed a very simple GUI in the beginning that displayed a menu for selecting shapes, a JPanel that would display the selected shape (in the beginning I left this panel empty), a checkbox for rotating and a “Customise” button that would bring up a separate customisation window which would contain more options. This preliminary interface was necessary to make sure that I could test functionality as it was being implemented by seeing what was being displayed as a result of my code. I left the button/checkbox functionality and most of the user controls to be decided on at a later stage in the project.

When designing the final GUI, I had tried various different layouts. I found Java's GridBagLayout to be confusing and difficult to use at first compared to GridLayout, but after spending several hours experimenting on the interface with it I found GridBagLayout to be capable of much more than other layouts. It took me some time to figure out just how to change the size of each grid cell so I could have certain components scaled to my preferred size without impeding on the positioning of other components.

1.2.2 Linear Algebra

When looking into projection methods, it was apparent that I would need a more solid foundation of mathematical knowledge to know how to approach some problems. I spent much time reading up on concepts involved projection before attempting to look at the maths and try it for myself. This meant I had to wrap my head around some concepts involved in linear algebra such as vertices and their positioning relative to different potential co-ordinate systems.

I gradually moved through an online tutorial practicing exercises on various operations on vertices, vectors and matrices, such as addition, subtraction, multiplication, dot product, cross product and so on (I have a link to this tutorial on my project blog). I sketched out many examples of problems by hand and worked on the process of projecting a set of 3D vertices to 2D, employing the mathematical concepts I had been learning about.

1.2.3 3D to 2D Projection

I followed an example of projecting a prism in 3-space to 2-space, by multiplying each of the prism’s vertices by a transformation matrix, which removes the z co-ordinate. The resulting 2D co-ordinates were different depending on the parameters of the transformation matrix

(i.e. the viewpoint and viewplane). The process here involved following a straightforward formula to get each 2D vertex, so implementing this projection was relatively simple.

Later in the project, when researching 4D projection, I could find no resources online on extending this 3D projection method to 4D, so I had to abandon it for a different method which involved a vector from the viewpoint to a 'looking' point (the point we are looking at). Though this meant redesigning work that I had already done, this new method added depth to the imagery, so the sides of the shape furthest from the viewpoint actually appear smaller.

1.2.4 4D to 3D Projection

My supervisor pointed me towards a very helpful resource on 4-space visualisation on computers. This was a thesis published in 1991 (link in project blog) that outlined the whole process from conceptual 4D co-ordinates to 2D screen co-ordinates, as well as the addition of raytracing methods to make shapes appear nicer and easier to understand.

This thesis provided me with just the formulae needed to transform 4D co-ordinates to 3D co-ordinates. However, it made no sense for me to continue using my existing 3D projection method as this functionality could be abstracted and applied to both 4D and 3D with differing parameters (and also because the method lacked depth perception).

The concepts involved in 4D to 3D projection were mostly covered by the tutorial I had worked on, but the addition of a function for finding the cross product of three 4D vectors was quite different to the 3D cross product formula.

1.2.5 JUnit

I looked into different methods of testing for the project and found that unit testing made the most sense for my needs. Since I was using the Eclipse IDE for coding and Java as my coding language, I decided to look into JUnit.

I had never used JUnit but found it quick to learn. The basic idea is that you write a number of tests (usually one for each method) that, when run, assert either a success or a failure. If the test fails, you can see exactly where your code has gone wrong. JUnit provides a number of handy tools, such as Suites, which work as a collection of JUnit test cases. So you can write 10 JUnit classes, each corresponding to a class in your project code, and write one test suite containing each test class. When this suite is run, it calls all 10 of the test cases, saving you time and allowing you to effectively test all (testable) aspects of the program with a single click.

JUnit can also work with testing for failure, so you can write code that is expected to produce an Exception and if it does so, the test evaluates as a success.

```
@Test(expected=ArrayIndexOutOfBoundsException.class)
public void testGetColumnIndexOutOfBounds(){

    Matrix m = new Matrix(new double[][]{
        new double[]{1, 0, 2}
    });
    m.getColumn(3);
}
```

The above test will assert true if and only if an ArrayIndexOutOfBoundsException is produced by the code.

2. Design

The research-oriented nature of the project has made a fully fleshed-out design from the very beginning an unrealistic goal. In this respect, I put together a very simple 'architecture' at the beginning of the project.

2.1 High-Level System Architecture

The general plan for the software was that the GUI components would be managed in the "visualiser" package. The "graphics" package would see all the behind the scenes work (i.e. the mathematics involved in producing the 4D images). The "shapes3D" and "shapes4D" packages are locations to store each specific 3D and 4D shape, respectively. This is just to prevent cluttering of the graphics package.

The visualisation package calls on the graphics package in order to produce the image on the GraphicsPanel in the MainWindow. The graphics package imports shapes from the other 2 packages as they are needed.

2.2 Graphics

The graphics package contains the bulk of the code. The classes within it interact with eachother to produce whichever shape is selected by the user, and, if the user chooses, the package will cover the shape's rotation about the selected axes/planes. The package contains 10 classes: Constants, GraphicsPanel, Matrices, Matrix, Polygon2D, Polygon3D, Polygon4D, Vector, Vertex and ViewSettings.

2.3 3D Shapes

The shapes3D package contains one class for each of the 5 convex regular 3-polytopes. They are: Tetrahedron, Cube, Octahedron, Dodecahedron and Icosahedron. Each class is an extension of Polygon3D, and the role each shape is to specify the values of each vertex and which vertices join together to make faces (2D polygons).

2.4 4D Shapes

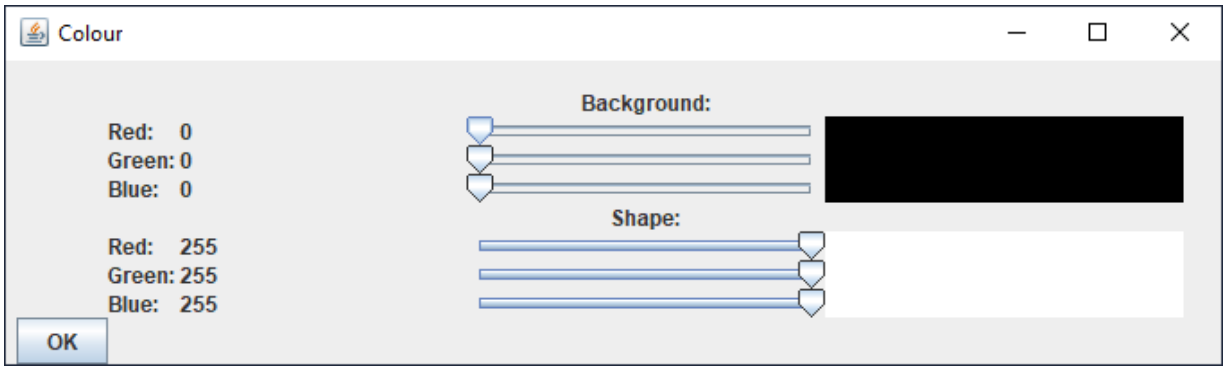
The shapes4D package works in the same way as the shapes3D package, but relative to 4D. It contains one class for each of the 4 convex regular 4-polytopes that the project covers. Each shape has its own specification of vertex locations and 2D faces.

The intuitive ideal design for 4D polyhedra would be to have each one composed of cells (3D polygons). For example, the Tesseract (or 8-cell) is composed of 8 cubes. However, the specification of each 3D polygon seemed too meticulous a task to be of any practical use. Some cells would share common edges with one another, so resulting 4D polygons would be drawing the same lines multiple times. This is unnecessary complexity, so I felt it best to specify each 4D shape's 2D faces directly within the shape's class definition.

2.5 GUI

The visualisation package groups together all components of the software's GUI. The MainWindow contains most of what the user sees. It directly controls the shape and dimension selection menus, as well as the "Colour" and "Reset" buttons. The MainWindow contains a RotationPanel and ViewpointPanel. The RotationPanel itself is broken down into a RotationAxesPanel and RotationPlanesPanel. These both contain relevant checkboxes. When a checkbox on one of these sub-components is checked, the ViewSettings rotation matrix is changed and the static GraphicsPanel on the MainWindow is updated. The ViewpointPanel contains buttons for incrementing or decrementing the value of each axis of the viewpoint.

The "Colour" button brings up a separate ColourWindow which is composed of two ColourPanels. Each panel contains 3 JSliders, one for each 'rgb' value of the relevant colour. The user is able to change both the background and foreground colours of the GraphicsPanel here.



3. Implementation

3.1 GraphicsPanel

This is the panel that displays the shapes. It contains a Polygon3D and Polygon4D, a Vertex for the viewpoint, a boolean for rotation and 2 transformation matrices, one for 3D and one for 4D. The transformation matrices are stored here and only calculated in the constructor to prevent unnecessarily re-calculating the same matrix several times as different shapes are selected. The matrices are static and accessed through 'getter' methods by class Vertex.

The "setShape" method sets the value of the 3D and 4D polygons (one of which will be set to null, depending on which dimension is active) and refreshes the panel.

3.2 ViewSettings

This class contains a number of public static final objects accessible to all other classes. This class allows the developer to change things like rotation speed, orientation vectors and the scale of the shape in order to test certain functionality. The class also contains rotation matrices that are updated according to user-selected axes or planes.

3.3 Vertex

This class represents a vertex. Most of the operations here are quite intuitive. Each vertex has a double value for its x, y, z and w co-ordinates, though not all vertices use a z or w co-ordinate, so there are constructors to allow for this and default these values to zero.

The steps involved in projecting a vertex are all here, including transformation (converting real-world co-ordinates to co-ordinates relative to the viewpoint), perspective projection (removing the z or w co-ordinate) and mapping to the viewbox (intermediate region in 3-space) or straight to the screen.

3.4 Vector

There is not much about this object that requires explanation. A vector has four double values: a, b, c and d. The operations involved are fairly straight forward.

3.5 Matrix

A matrix is represented here with a two-dimensional array of double values.

```
// Adds new element to the next available slot in the Matrix
public void add(double d){

    data[rowIndex][columnIndex] = d;
    columnIndex++;

    if(columnIndex == data[0].length){
        // Move on to next row
        rowIndex++;
        columnIndex = 0;
    }
}
```

The above code shows how values are added to the array. The object keeps track of the indices available for adding items to the array, so rows are filled out one at a time and when a row is full, it moves on to the next one.

```
// This is necessary to get all values in a column from the multi-dimensional array
public double[] getColumn(int index){

    double[] column = new double[numRows];

    for(int i = 0; i < numRows; i++){
        column[i] = data[i][index];
    }

    return column;
}
```

When dealing with multi-dimensional arrays, the data is stored in multiple rows. There is no functionality provided by the Java API for easily getting a column value, so I found the above code to be necessary.

3.6 Matrices

This class contains static rotation matrices for 3D and 4D. These matrices are accessed by ViewSettings in order to determine the rotation matrix that may be a combination of multiple rotation matrices. Each vertex of the shape is multiplied by the rotation matrix to

move it about a certain axis/plane. Here is an example of one rotation matrix:

```
public static Matrix anticlockwiseRotationX(double theta){
    return new Matrix(new double[][]{

        new double[]{1, 0, 0},
        new double[]{0, Math.cos(theta), -Math.sin(theta)},
        new double[]{0, Math.sin(theta), Math.cos(theta)}
    });
}
```

3.7 Polygon2D

A polygon2D is made up of any number of vertices which are joined together to form edges in the sequence in which they are specified. The first and last Vertex are always joined together to complete the shape.

```
// Used to join two vertices with a line
private void drawLine(Graphics2D graphics, Vertex v1, Vertex v2){
    graphics.draw(new Line2D.Double(v1.getX(), v1.getY(), v2.getX(), v2.getY()));
}
```

A problem I encountered here is that the standard Java Graphics API draws lines (and other 2D objects) with their specified screen co-ordinates explicitly as integers, so in order to pass the double co-ordinates of vertices I had to import the Graphics2D library. This saved me the work of storing each co-ordinate's converted values as int variables.

3.8 Polygon3D

3D shapes are extensions of this abstract class. The class provides some abstract functionality common to all 3D shapes, such as abstract methods setFaces and project. Each extension of this class (Tetrahedron, Cube, Octahedron, Dodecahedron, and Icosahedron) defines their faces uniquely. The project method is abstract because an abstract class cannot be instantiated, and this method must return a new object, so that the conceptual 3-space co-ordinates remain intact.

This class also contains a getProjectedVertices method, which loops through the list of vertices and transforms and projects each one before returning the new list of projected vertices.

```
protected static ArrayList<Vertex> getProjectedVertices3(ArrayList<Vertex> vertices){

    ArrayList<Vertex> projectedVertices = new ArrayList<>(); // To store the projection of each vertex

    // Transform each vertex from world coordinates to eye coordinates
    for(Vertex v : vertices){
        projectedVertices.add(v.transform3());
    }

    for(Vertex v : projectedVertices){

        // Get perspective projection onto viewplane
        Vertex projection = v.perspectiveProjection3();
        v.setX(projection.getX());
        v.setY(projection.getY());

        // Map to screen
        projection = v.mapToScreen();
        v.setX(projection.getX());
        v.setY(projection.getY());
    }

    return projectedVertices;
}
```

Also contained in this class is the rotate method, which does actually change each vertex. Below are class Tetrahedron's vertex and face specifications:

```
public static final Vertex[] TETRAHEDRON = { // Specify each original Vertex, before shape is scaled
    new Vertex(-1, 0, -Constants.ONE_OVER_ROOT_2),// 0
    new Vertex(1, 0, -Constants.ONE_OVER_ROOT_2), // 1
    new Vertex(0, -1, Constants.ONE_OVER_ROOT_2), // 2
    new Vertex(0, 1, Constants.ONE_OVER_ROOT_2)   // 3
};

protected void setFaces(){

    faces = new ArrayList<>();

    faces.add(new Polygon2D(new Vertex[]{vertices.get(0), vertices.get(1), vertices.get(2)}));
    faces.add(new Polygon2D(new Vertex[]{vertices.get(0), vertices.get(1), vertices.get(3)}));
    faces.add(new Polygon2D(new Vertex[]{vertices.get(0), vertices.get(2), vertices.get(3)}));
    faces.add(new Polygon2D(new Vertex[]{vertices.get(1), vertices.get(2), vertices.get(3)}));
}
```

3.9 Polygon4D

This class applies exactly the same functionality provided by Polygon3D but to 4D shapes.

3.10 Constants

Constants is the simplest of all implemented classes (aside from MainClass). The class only contains mathematical constants that are used to plot the starting vertices for some shapes centred on the origin.

```java
package graphics;

public class Constants {

    // List of constant values used to define the vertices of some shapes

    public static final double ONE_OVER_ROOT_2 = 1 / Math.sqrt(2);
    public static final double PHI = (1 + Math.sqrt(5)) / 2;
    public static final double ONE_OVER_PHI = 1 / PHI;
    public static final double ONE_OVER_ROOT_10 = 1 / Math.sqrt(10);
    public static final double ONE_OVER_ROOT_6 = 1 / Math.sqrt(6);
    public static final double ONE_OVER_ROOT_3 = 1 / Math.sqrt(3);
    public static final double ROOT_3_OVER_2 = Math.sqrt((double) 3 / (double) 2);
    public static final double TWO_OVER_ROOT_3 = 2 * ONE_OVER_ROOT_3;
    public static final double TWO_ROOT_2_OVER_5 = 2 * Math.sqrt((double) 2 / (double) 5);
}
```

3.11 MainWindow

The layout of this window is set using a GridBagConstraints variable ‘gb’. Below shows the padding of the grid. By adding horizontal and vertical struts, the width and height of each row and column is determined. I have applied this same solution to figuring out the layout of other windows.

```java
// set the width/height of each column/row of the grid

gb.gridx = 2;
gb.gridy = 1;
pane.add(Box.createHorizontalStrut(250), gb);

gb.gridx = 3;
gb.gridy = 1;
pane.add(Box.createHorizontalStrut(125), gb);

gb.gridx = 4;
gb.gridy = 1;
pane.add(Box.createHorizontalStrut(125), gb);

gb.gridx = 1;
gb.gridy = 2;
pane.add(Box.createVerticalStrut(50), gb);

gb.gridx = 1;
gb.gridy = 3;
pane.add(Box.createVerticalStrut(500), gb);
```

Another Java object I’ve learned about through GUI practice is the DefaultComboBoxModel<T>, which allows you to update the selections available in a menu without revalidating and repainting the component each time.

3.12 RotationPanel

The RotationAxesPanel and RotationPlanesPanel within this panel employ simple GridLayouts. These classes contain sub-classes which implement ItemListener, so that when a checkbox is ticked, the MainWindow updates its GraphicsPanel.

3.13 ColourWindow

The key functionality within the ColourPanels is their ability to update the displayed colour in real time once a slider is moved. This is detailed below:

```
private class Slide implements ChangeListener{

    @Override
    public void stateChanged(ChangeEvent e){

        if(e.getSource().equals(rValue)){
            rLabel.setText("Red:     " + rValue.getValue());
        }
        else if(e.getSource().equals(gValue)){
            gLabel.setText("Green: " + gValue.getValue());
        }
        else if(e.getSource().equals(bValue)){
            bLabel.setText("Blue:    " + bValue.getValue());
        }

        colour.setBackground(new Color(rValue.getValue(), gValue.getValue(), bValue.getValue()));
    }
}
```

5. Project Conclusions and Future Work

The body of work that this project entailed involved a lot of challenges not only for my programming skills but also my ability to handle some complex mathematics. Though I did not meet every original goal that I set out to achieve with this project, I learned a substantial amount where I overcame problems and even more about things that I failed to do. When I started out, I had no knowledge in graphics, and now I feel that were I to take on another graphics-oriented project I would hit the ground running, and if there are areas that I lack knowledge in, I at least have a better idea about what it is I don't know and would be quicker at solving problems.

The finished product is not the same as the initial planned program. I did not get all 6 convex regular polyhedra. As well as this, I feel that the visualisation capabilities of the final product are limited. The shapes are exclusively given in a 'wireframe' style. This is because the project lacks ray tracing functionality. However, I don't believe the design of the project was at fault for this aspect. Starting out, I didn't know what raytracing was so I could not have anticipated its significance from the beginning. I was more focused on the core of mathematics involved, and I believe I am justified in having focused on this. Without carefully laying out the foundation of graphics mechanics, there would be no place for the sort of 'icing' that things like raytracing would provide.

Having said that, I'm sure I could have been more diligent in project work in the first semester and if I had been then I could very well have found time for raytracing and other additions to the project. I have detailed some potential room for improvement below.

5.1 More Shapes

Of course, there are the final 2 convex regular 4-polytopes that I didn't get round to implementing. I believe that doing so would require a shape construction algorithm that links each vertex to its connecting vertices to form the shapes edges. I made some attempts at such an algorithm with pen and paper but to no avail. In theory, if I found such a method, there would be no need for a class corresponding to each shape, only a parameter corresponding to the shape in the form of that shape's Schläfli Symbol, which tells enough about the shape to determine its pattern of vertices and edges. This is the kind of abstract functionality I hoped I would achieve with the project.

5.2 Added User Control

A number of factors of the shapes' display could be adjusted with more user controls on screen, such as rotation speed, the size of the shape as well as orientation options (looking point, up and over vectors). With raytracing incorporated, the lighting could also be changed and the user could switch between solid and wireframe view.

5.3 Higher Dimensions

I have not looked into any form of projection of co-ordinates in more than 4 dimensions, but there are some images online of 5D shapes. In speculation, I imagine the processes involved in such projections are similar but not identical to 4D projection, just as 4D projection is similar but not identical to 3D projection. It seems a great task to implement such complex shapes, but that is exactly what I thought about this project in the beginning, so it is not a matter of whether it is achievable but rather what is necessary to achieve it.