Four-Dimensional Brickout

Matt Jordan

Abstract

This project examines the problem of visualizing the 4th dimension through the use of a classic 2-D game being implemented in higher dimensions. The project begins by recreating the original 2-D Brickout game, then developing a 3-D version before finally moving onto a 4-D implementation. This will be accomplished using OpenGL with C++.

1 Mathematics

1.1 Three to two dimensions

Before looking at four dimensions, it is easier to first look at the transition from three dimensions to two dimensions. By placing a three dimensional object on a two dimensional plane, a dimension is lost. To accurately visualize such an object, one or more depth cues need to take the place of the dimension. A depth cue is some manipulation of the object or the environment that aid a viewer in perceiving the object's lost dimension. Some examples of typical depth cues are:

- Shadows
- Rotation
- Lighting

If two objects are directly behind one another and the viewer is facing them, they will only see one object. A shadow allows the viewer to look at the plane below the objects and see that there are in fact two shadows at different locations, showing the second object's depth relative to the first one. Rotation accomplishes the same goal. By either rotating a view automatically or giving the viewer control of the rotation, the viewer is able to see how the different objects appear in different planes of view. If one were to shine a light directly at a set of objects, the differently positioned objects would reflect the light in different ways, giving highlights that serve as clues for the viewer.

1.2 Four dimensions to two

When trying to visualize a four dimensional object in a two dimensional plane, two dimensions have been lost. Multiple depth cues have to be invoked in order to accurately represent the object, or alternative methods of display are required. This project looks at two such alternative methods.

Dimension truncation

The first method explored is dimension truncation. This is the simplest method of bringing a four dimensional object into three dimensions. By ignoring one of the dimensions of the object, the object can be represented like any other three dimensional object and as such normal depth cues can be used. The three dimensional analogy would be to draw only the front face of of a cube when drawing it in two dimensions. However there is a serious flaw with this method. In the three dimensional analogy, one could pass two cubes past each other at vastly different depths, but in the two dimensional representation it would look as if two squares were colliding and passing through one another instead of stopping. The same problems occur in the four dimension to three dimension transition, causing objects to appear to pass through each other without repercussion. This can be overcome to some extent by allowing the viewer to switch between which three dimensions are used to represent the object. This method is more of an interesting experiment than a useful tool for visualizing the fourth dimension.

Four-dimensional rotations and stereographic projection

A typical three dimensional rotation around the z-axis is done by multiplying a rotation matrix by a matrix holding the coordinates of a point. What this is actually doing is rotating the x-axis into the y-axis at an angle θ, called an XY rotation. The four dimensional analogue of this is to rotate two axes into each other about a plane. Thus the possible rotations in four dimensions are: XY, XZ, XW, YZ, YW, and ZW. These rotations are performed by multiplying different 4x4 matrices by a 4x1 matrix representing the point to be rotated. These matrices are as follows (note that each θ is unique to its rotation matrix:

$$R_{XY} = \begin{bmatrix} \cos(\theta) & \sin(\theta) & 0 & 0 \\ -\sin(\theta) & \cos(\theta) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} R_{XZ} = \begin{bmatrix} \cos(\theta) & 0 & -\sin(\theta) & 0 \\ 0 & 1 & 0 & 0 \\ \sin(\theta) & 0 & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$R_{XW} = \begin{bmatrix} \cos(\theta) & 0 & 0 & -\sin(\theta) \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ \sin(\theta) & 0 & 0 & cos(\theta) \end{bmatrix} R_{YZ} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\theta) & \sin(\theta) & 0 \\ 0 & -\sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$R_{YW} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\theta) & 0 & -\sin(\theta) \\ 0 & 0 & 1 & 0 \\ 0 & \sin(\theta) & 0 & \cos(\theta) \end{bmatrix} R_{ZW} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \cos(\theta) & -\sin(\theta) \\ 0 & 0 & \sin(\theta) & \cos(\theta) \end{bmatrix}$$

To simplify the process, the matrix containing the coordinates of the point to be rotated are only multiplied by one matrix, R_{Tot}. This matrix is given by: $R_{Tot} = R_{XY}R_{XZ}R_{XW}R_{YZ}R_{YW}R_{ZW}$

Were one to place a flashlight somewhere on the z-axis and move a cube about three dimensional space in front of a wall, its shadow would be projected back onto the wall. This is the basic idea behind stereographic projection. By placing a "flashlight" on the w-axis and shining it on a four dimensional object, it is able to be projected onto the three dimensional space. Given points (x_0, y_0, z_0, w_0) and the location of the "flashlight" on the w-axis f, this is represented by the formula $(x, y, z) = (\frac{x_0}{f-w_0}, \frac{y_0}{f-w_0}, \frac{z_0}{f-w_0})$

1.3 The Paddle and the Ball

In most implementations of the Brickout game, the ball deflects off the paddle according to the law of reflection, that is to say the x-component of the ball's velocity remains the same and the y-component of velocity is multiplied by -1. In the implementation presented here, the ball deflects off the paddle according to the following equations:

$$V_x = -\sin(\frac{2*x_b - x_p*\pi*\tau}{width_p})V_{x0}$$

$$V_y = \sqrt{1 - V_x^2}$$

Where V_x is the x-component of the ball's velocity, V_y is the y-component of its velocity, x_b is the x-coordinate of its center, x_p is the x-coordinate of the center of the paddle, $width_p$ is the width of the paddle and τ is some constant.

The implication of this is that the further away from the center of the paddle the ball hits, the greater the magnitude of the x-component of its velocity. The ball will always deflect in the same direction as the side of the paddle it deflects off of (i.e. if the ball deflects off the right side of the

paddle, it will go to the right). Additionally, the y-component is such that the magnitude of the total velocity of the ball remains constant.

2 The Game

Brickout is a game that first appeared in arcades in 1976. The goal of the game is to break all of the breaks that appear in the top section of the field of play by hitting them with a ball. If the ball reaches the bottom of the field of play, the player loses the game. The player interacts with the ball by moving a paddle around that deflects the ball back towards the top of the field of play. In the original version of the game the paddle would deflect the ball according to law of reflection derived from the Fresnel equations. In the 2D implementation presented in this project, the ball is reflected according to the rules discussed in section 1.3.

3 The Application

3.1 2DBrickout

This implementation serves as the groundwork for all further implementations of the Brickout game. It is composed of several classes. The playing field is defined by the GameZone class, which holds all of the different objects that make up the game, such as the ball and the paddle. These objects are all based off the Object class, which defines basic functionality shared by all objects such as collision detection and sets virtual methods that all inherited classes must overload, such as drawing. From there each object is given its own unique class inheriting methods from the Object class. Basic square collision detection is used to detect collisions. Each object is generalized to a rectangle, and the ball compares its left side with each objects right side, its top side with the objects bottom side, etc. If the ball detects a collision it then calculates which side it is colliding with and deflects off of the object in the appropriate manner.

The player controls the paddle using the "a" and "d" keys to move it left and right. Some functionality shared by all implementations of the Brickout game are as follows:

- Press "r" to reset
- Press "l" to generate a new set of bricks

- Press "g" to enable the paddle to move on its own, automatically tracking the ball
- Press the "Esc" key to exit the program.

3.2 3DBrickout

The introduction of a third dimension represents the first step towards four dimensional Brickout. In this implementation, All objects are given a z position and a depth. Collision detection is now cube based, but is functionally the same as it was in two dimensions. Drop shadows have been added in order to give a better understanding of where the ball and the bricks are located.

3.3 4DBrickout - dimension truncating

This is the first implementation of Brickout in 4 dimensions. Each object is given a w position as well as another depth component, called "wDepth". Only three axis are shown at a time, with the fourth dimension being ignored entirely as far as visuals are concerned. The player can switch between which three axis are being viewed by pressing the 'v' key. Again, collision detection is increased by dimension to 4-cube collision detection, but becomes no more difficult than in two or three dimensions. Besides the choice of axis, this implementation appears to be visually identical to the three dimensional implementation.

3.4 4DBrickout - stereographic projection

This implementation of Brickout is made possible by the Matrix4D class. The name "Matrix4D" is actually something of a misnomer, as the class is generalized to matrices of any size. This class has several key methods, namely the Multiply method. The Multiply method takes the input of a another Matrix4D and multiplies the first Matrix4D by the input, outputting another Matrix4D.

In order to display objects, the Object class has the generateVertices method. This method takes an input of a Matrix4D, which is expected to be R_{Tot}. The method first creates a vector of Matrix4Ds. Each Matrix4D is a 4x1 matrix containing the coordinates of a vertex of the object. It then multiplies R_{Tot} by each Matrix4D in this vector, completing the rotation.

This is then passed to the object's draw function which draws the object according to the formula discussed at the end of section 1.2.

4 Bibliography

Banks, David. Interacting With Surfaces In Four Dimensions Using Computer Graphics. Rep. no. TR93-011. Chapel HIll: n.p., 1993. Univeristy of North Carolina Computer Science. University of North Carolina. Web. 17 Oct. 2013. (http://www.cs.unc.edu/techreports/93-011.pdf).

"Rotations in Three Dimensions: 3D Rotation Matrices." 3D Rotation Matrices with OpenGL. N.p., n.d. Web. 15 Dec. 2013.

"Matrix Multiplication." Wikipedia. Wikimedia Foundation, 12 Oct. 2013. Web. 15 Dec. 2013.

"How to Overlay Text in OpenGL." StackOverflow. N.p., n.d. Web. 15 Dec. 2013. (http://stackoverflow.com/questions/20082576/how-to-overlay-text-in-opengl).