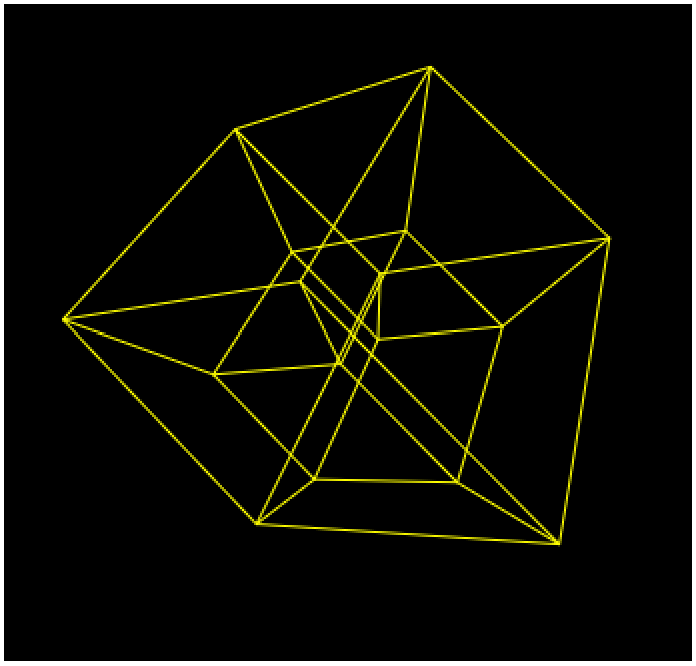
Hypercube Rotations via Quaternion Arc-balls



Ethan Rowe & Aaron Paterson

November 26, 2020

Introduction

This project started from the boredom of quarantine. With both my friend and I interested in matrices and linear algebra. In combination with this curiosity was the linear algebra class I was taking and the art class my friend was taking. In a true combination of efforts, this project was an effort to create a visualization for a hypercube. The end result of this project includes linear algebra, century old math, html, CSS, Java script, and higher order number systems. And if I am being honest, looks really cool.

Linear Algebra

The whole project relies on matrices, taking mouse input, and converting those into rotations for each of the four degrees of freedom of the cube. These were a couple different blocks of code that we used, that were also taught in class:

```
function transpose(m) { <!--Linear transpose-->
    var t = [];
    for(var i=0; i<Math.max(...(m.map(a => a.length))); i+=1) {
        t[i] = [];
        for(var j=0; j<m.length; j+=1) {
            t[i][j] = m[j][i];
        }
    }
    return t;
}
```

The transpose function implements the concept of a matrix transpose that we learned in class. It takes a matrix as a parameter and procedurally flips the contents of the matrix around its main diagonal.

```
function dot(l, r) { <!--Dot product matrix-->
    var d = 0.0;
    for(var i=0; i<Math.min(l.length,r.length); i+=1) {
        d += l[i]*r[i];
    }
    return d;
}
```

This dot function implements the dot product calculation. It takes two vectors as inputs, then applies the dot product calculation for each index.

```
function compose(l, r) { <!-- Matrix-Matrix multiplication-->
    var c = [];
    var t = transpose(r);
    for(var i=0; i<l.length; i+=1) {
        c[i] = [];
        for(var j=0; j<t.length; j+=1) {
            c[i][j] = dot(l[i],t[j]);
        }
    }
    return c;
}
```

The compose function uses the transpose and dot functions above to multiply two matrices. The transpose function is needed because the input matrices are an n by m and m by n matrix. After the right-hand matrix is transposed, this function can apply the dot product to combinations of the rows of these matrices, and find each entry in the matrix product.

```
function cross(l, r) { <!--Cross Product-->
    return [
        l[1]*r[2] - l[2]*r[1],
        l[2]*r[0] - l[0]*r[2],
        l[0]*r[1] - l[1]*r[0]
    ];
}
```

This piece of code implements the cross product. The quaternion multiplication uses this and the dot product to help measure rotations in response to mouse input.

Quaternions

This section of the project was almost completely new to me. They exist as an extension of imaginary numbers. Where imaginary numbers give additional dimensions to a number line, quaternions give additional dimensions to three-dimensional space. A quaternion is written in the form:

$$A + Bi + Cj + Dk$$

A, B, C, & D represent real numbers, and the i, j, k represents the complex components. These complex components represent the 3 spatial axis and are used as unit vectors. In the case of this program, two quaternions are used to track the orientation of the cube. The mouse input slides a plane tangent to two arc balls, whose orientations are measured by the quaternions

```
    function twist(x, y, c, s) {
        var l = Math.hypot(x, y);
        if(0===l) {
            var t = (tock/12000);
            return twist(
                s*Math.cos(c*t),
                c*Math.sin(s*t)
            );
        }
        else {
            var t = l/1024.0;
            var s = Math.sin(t/2);
            return [s*x/l, s*y/l, 0, Math.cos(t/2)];
        }
        return [0,0,0,1];
    }
```

This function measures the orientation of an arcball that is rolling tangent to the plane the mouse. It alters one quaternion when the mouse moves towards the center of the screen, and the other turns when it moves away.

```
function qcompose(q, p) { <!--quaternion-quaternion mult--> <!--
Special Rotations-->
    var v = q.slice(0,3);
    var u = p.slice(0,3);
    return [ v.map(e => e*p[3]), u.map(e => e*q[3]), cross(v,u) ]
        .reduce(ttt).concat(q[3]*p[3] - dot(v,u));
}
```

This function is used to multiply the orientation quaternions with the ones measured from the mouse input. It finds a third quaternion that represents the result of applying the first rotation before the second. Quaternions are not commutative, and there are two unit quaternions for each 3d orientation.

```
function qpfour(q,p) { <!--Van Elfrionkhof Formula-->
    return compose(
        [   [+q[3],-q[2],+q[1],+q[0]],
            [+q[2],+q[3],-q[0],+q[1]],
            [-q[1],+q[0],+q[3],+q[2]],
            [-q[0],-q[1],-q[2],+q[3]]
        ],
        [   [+p[3],+p[2],-p[1],+p[0]],
            [-p[2],+p[3],+p[0],+p[1]],
            [+p[1],-p[0],+p[3],+p[2]],
            [-p[0],-p[1],-p[2],+p[3]]
        ],
    );
}
```

This section of code, qpfour, is where the two quaternions are taken in, and using the Van Elfrinkhof Formula, a single 4d rotation matrix is formed. 3d rotations are the product of a vector between a unit quaternion and its conjugate, equivalent to the product of that vector and a rotation matrix. Similarly, 4d rotations are the product **qvp** where both q=a+b**i**+c**j**+d**k** and p=e+f**i**+g**j**+h**k** are independent unit quaternions, equivalent to the product of **v** and the following matrices:

Van Elfrinkhof Formula

$$
\begin{aligned}
A &= \begin{pmatrix}
ap-bq-cr-ds & -aq-bp+cs-dr & -ar-bs-cp+dq & -as+br-cq-dp \\
bp+aq-dr+cs & -bq+ap+ds+cr & -br+as-dp-cq & -bs-ar-dq+cp \\
cp+dq+ar-bs & -cq+dp-as-br & -cr+ds+ap+bq & -cs-dr+aq-bp \\
dp-cq+br+as & -dq-cp-bs+ar & -dr-cs+bp-aq & -ds+cr+bq+ap
\end{pmatrix} \\
&= \begin{pmatrix}
a & -b & -c & -d \\
b & a & -d & c \\
c & d & a & -b \\
d & -c & b & a
\end{pmatrix}
\begin{pmatrix}
p & -q & -r & -s \\
q & p & s & -r \\
r & -s & p & q \\
s & r & -q & p
\end{pmatrix}.
\end{aligned}
$$

This formula is from a Dutch physics and medicine journal from 1897.

L. van Elfrinkhof: Eene eigenschap van de orthogonale substitutie van de vierde orde. Handelingen van het 6e Nederlandsch Natuurkundig en Geneeskundig Congres, Delft, 1897.

The formula was slightly modified so that the order of the real and imaginary components matches our implementation with JavaScript arrays, and so that our implementations of vector operations can be used with homogenous coordinates and quaternions.

In the program, variable four is the perspective matrix, responsible for what is drawn on the screen, variable five is responsible for the hyper-rotation matrix. Five gets the result of the Van

Elfrinkhof formula, four gets a simple one-point perspective matrix, and then the hypercube is plotted to the screen using this code:

```
ctx.beginPath();
    [0,1,9,11,10,8,9,13,15,11,3,7,15,14,10,2,6,14,12,8,0,4,12,13,5,7,6,4,5,1,3,2]
        .map(i => [0,1,2,3].map(j => (Math.sign(i&(1<<j))<<1)-1))
        .map(p => transform(five, p))
        .map(p => p.slice(0,3).concat(3-p[3])) // 5d -> 4d
        .map(project) // 4d -> 3d -> 2d
        .forEach(p => ctx.lineTo(...p));
    ctx.closePath();
    ctx.stroke();
}
```

Also, in this code segment you can see the reduction from five dimensions (only four are using pure matrix multiplications) down into two on the screen.

Normally rendering 3d points involves translation, like a camera that moves with WASD, or props that move independently of it. There is no 3x3 matrix that translates (x, y, z) into (x+a, y+b, z+c), but by augmenting 3D vectors into 4D homogenous coordinates, (x, y, z,1), you can make a 4D translation matrix that produces (x+a, y+b, z+c,1). We don't actually really use a translation matrix in the cube projection, we just rotate the points and scale the screen. But we still use the extra component in a similar way in the perspective matrix, by translating the z coordinate so that points further out in the 4th dimension appear further away in the 3rd dimension. The hypercube needs four dimensions for its vertex positions anyway, so there is no reason to avoid it. Finally, when it is time to draw lines on the screen, we divide the 3d points by their z value to get (x, y, z) -> (x/z, y/z). This is a dirt simple projection, that just moves points further away on the z axis towards the center of the screen, where your eyes naturally expect distant lines to converge in an image.

Conclusion

This project got way deep very quickly, delving into things we both had never heard of. Quaternions were the largest learning curve I had experienced; they were something that did not make much sense to me at the time when we had first started experimenting with them. I am certainly no expert on any of it, but I have a much greater understanding of them. Having experienced the class this semester I have realized a lot of what we did was covered in class. Transpose/matrix multiplication make much more sense in their context here, than they did during the creation of this project. In addition, I started seeing more similarities to things we learned later during the year. For example, using the quaternions by themselves was prone to error, so an error catching method very similar to least square approximations was added to ensure they were accurate. The values were periodically normalized before being multiplied, a simpler equation than the one we used in class. Overall, this project helped lay the groundwork for stuff we learned in class, as well as letting me reach out to further understand more material that is related to matrices.

Link to the final interactive animation: https://ethansrowe.github.io/Linear_Mini_Proejct.html