Learning to See in 4D Through Three(.js) Dimensional Slices

Omar Shehata, Joseph Peterson, Tianyu Pang, Justin Pacholec
MSCS Deptartment - St. Olaf College

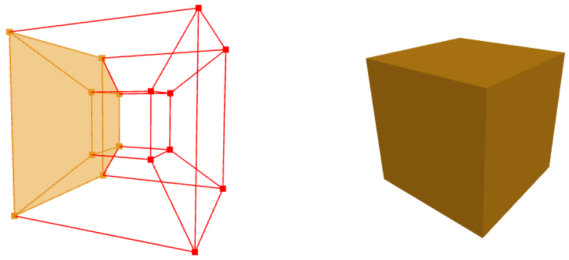

Figure 1: *Projected 4D cube with highlighted slice (left) The same 3D slice (right)*

Abstract

Higher dimensional spaces are applicable to many problems in science and engineering. To help others develop an intuition for these spaces, we have built an online tool to visualize four-dimensional geometry. While there are several existing tools to visualize the fourth dimension, we distinguish ourselves by focusing on dimensional analogy. The user can view projections and cross sections of any 2D, 3D, or 4D object represented by a cartesian equation, a parametric equation, or a convex hull. This research provides solutions to the technical challenges of real-time slicing of objects in 2D and 3D, as well as the construction and projection of 4D convex hulls, with a focus on pedagogical value for the visualizations.

Keywords: four dimensional, real time graphics, web, educational, geometric algebra

1 Introduction

Four dimensional space is a topic that easily piques many people's interests, and there have been numerous attempts to visualize such worlds [Mie] [4DT][Dra][Mat]. We wanted to create a tool that would be accessible to students with minimal mathematical background in an undergraduate course on 4D geometry.

In the same way the popular novel *Flatland* [Abbott et al. 1885] describes the struggle of 2D beings trying to understand a 3D world, we build up geometric intuition by encouraging students to try to understand 3D objects by looking at 2D slices (in addition to looking at 1D slices of 2D objects). We believe this makes the subject more approachable and allows students to easily generalize concepts to 4D.

An important goal was to encourage students to explore objects they could input themselves. There were several challenges in doing this real-time rendering on the web. In this paper, we present these problems and our solutions to them. We also close by highlighting the kinds of insights the app has so far been able to make clear, and where it falls short.

1.1 Using the Online App

The tool we developed is a web application written in Javascript and built on top of Three.js. It is open source and can be accessed at:

`https://github.com/StoDevX/humke-4d-geometry`

The user selects 2D, 3D or 4D mode to begin.

Each mode has an N-dimensional left view, and an (N-1)-dimensional right view. The left view renders the object itself, and the right view renders lower dimensional slices.

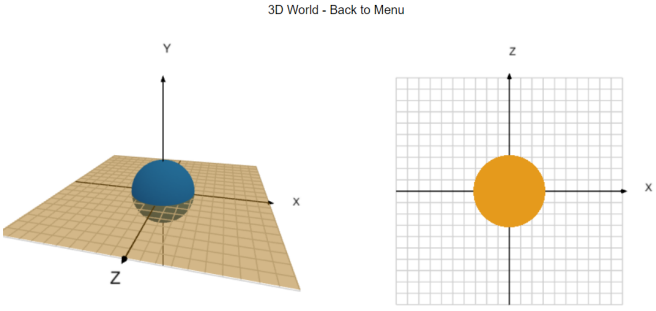

Figure 2: *A sphere and its 2D slice*

The slicing plane is controlled by a slider (not shown in the above figure). Users can input any cartesian equation, parametric equation, or a set of points describing a convex hull.

2 Lines of Consistent Thickness in 2D

To render arbitrary cartesian equations in 2D, we used a fragment shader. Given an equation as a string such as $y = sin(x)$, we first

split the left-hand side and the right-hand side. We then generate a GLSL function that evaluates LHS - RHS.

Listing 1: *Generated GLSL function for* $y = sin(x)$

```
float eq(float x,float y){
    return y - sin(x);
}
```

This allows us to evaluate the function at all pixels and then choose whether to render $y = sin(x)$, $y < sin(x)$ or $y > sin(x)$. This way we can support drawing lines as well as regions instantly as the user types.

The issue with rendering lines this way is that we are forced to choose a tolerance value. This is because there's a finite number of pixels on the screen, and it's unlikely that a pixel's position will correspond exactly with an x and y value that sets LHS = RHS.

Listing 2: *Evaluating the equation with fixed tolerance*

```
// p is a vec2 of the pixel's world position
if( abs(eq(p.x,p.y)) < 0.01){
 // Draw a point
} else {
 discard;
}
```

This works, but will produce lines of uneven thickness as seen in Fig 3.

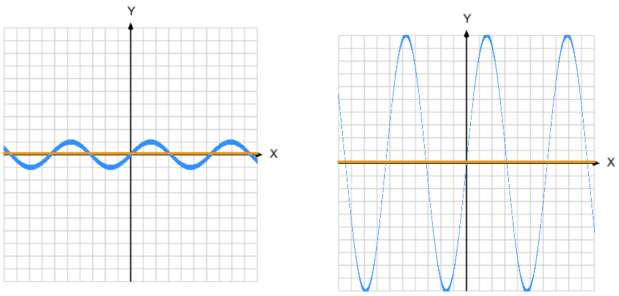

Figure 3: *Different graphs will vary in thickness.* $y = sin(x)$ *and* $y = 10 \cdot sin(x)$

To solve this, first notice that the rate of change of the function is directly proportional to the produced thickness. We can verify this visually by rendering the rate of change at each pixel. Notice how the white areas (high rates of change) in Fig 4 correspond to the areas with the thinnest lines in Fig 3.

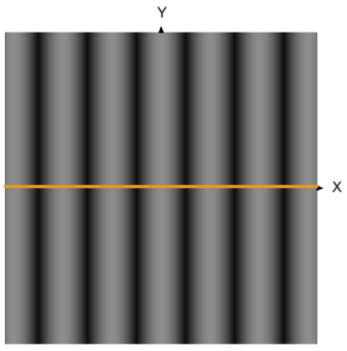

Figure 4: *Gradient field for* $y = 10 \cdot sin(x)$

This rate of change is computed for each pixel by choosing 4 neighbouring points within some small delta (we used 0.01) and evaluating the function at each one of those points. The difference is then taken and averaged to produce an approximate rate of change.

Finally, we use that that rate of change as our threshold value, effectively creating a dynamic thickness that produces lines with consistent thickness as seen in Fig 5.

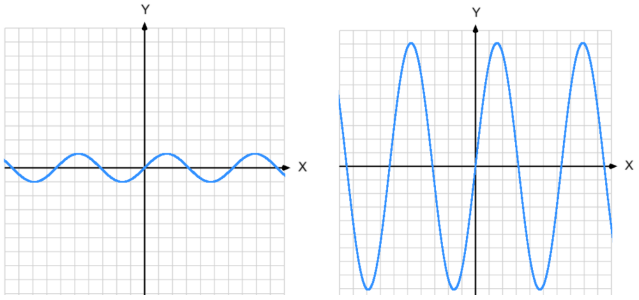

Figure 5: *A dynamic threshold produces lines of even thickness*

This rate of change can be multiplied by a final constant factor to control the final thickness of the lines.

3 3D Cartesian Equations

There is no equivalent hardware-accelerated way to iterate over all voxels in 3D space like we did in 2D, so we resort to the marching cubes algorithm [Bou] to approximate a function with an isosurface.

The marching cubes algorithm divides the space into a set of cubes (Fig 6). The corners of each cube are then tested for intersections. An intersection point exists between two corners that are on opposite sides of the equation. The intersection points are collected to build a polygonal approximation of the continuous surface.

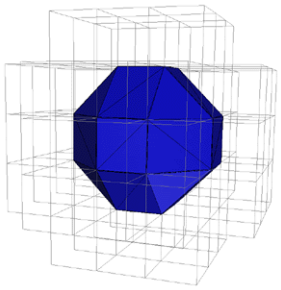

Figure 6: *A rough approximation of a sphere from [Bou]*

The real strength of the marching cubes algorithm comes from the use of lookup tables. The tables are used for the different possible ways a cube may be intersected by the surface and the correct triangulation of the intersection points. This efficiency makes the marching cubes algorithm ideal for interactive applications.

We define the resolution of the approximation to be the number of cubes per axis. We found 20 to be a good number for a "low" resolution, which is fast enough to enable users to type an equation and have the shape re-render on each keystroke. While 60 was a good "medium" that removed most artifacts. Finally 112 was used for "high" and usually takes a few seconds to compute. This was mainly used when taking screenshots of the app.

4 Slicing Shapes

We used several methods for slicing shapes in real-time.

4.1 2D

All shapes in 2D mode are sliced by passing them through a fragment shader which simply discards pixels with coordinates that are above or below the slicing line.

4.2 3D

All shapes in 3D are made up of a set of triangles that cover the surface. So the problem reduces to finding the line of intersection given a triangle and a plane.

But that can be reduces even further: A triangle is composed of 3 lines. Just check the intersection of each line with the plane. The total number of intersection points for a triangle and a plane will always be 2 (ignoring the case when one of the vertices is on the plane). The resulting 2 points describe a line.

Since all of our planes of intersection are aligned with one of the axes, this is a very simple check.

The connection of all the line segments resulting from the intersection of all the triangles that bound the shape forms the boundary of the intersection, which can then be filled or not depending on whether the shape is hollow or not.

4.3 4D

The problem in 4D is surprisingly not any more complicated. A 4D surface is bounded by 3D tetrahedra. Each tetrahedron is composed of 6 edges. We can thus check the intersection of each of those edges with the hyperplane (which is again simplified because our hyperplane is axis-aligned).

In 3D, each triangle's intersection with a plane always produced a line. This is because the only unique case is when one point is above and two are below.

In 4D, we have two separate cases: Either 1 point is above the hyperplane, and 3 are below, or 2 are above and 2 are below. The first case produces 3 points of intersection that form a triangle. The second cases produces 4 points of intersection that form a square.

The collection of these 2D faces when rendered will form the bound of the 3D slice. We are fortunate that we do not need to fill in the inside of the computed slice since the user cannot see inside bounded 3D volumes.

4.4 Cartesian

Cartesian equations are a special case since to produce the slice, we can substitute for a variable and get an equation that describes the slice in one less dimension.

For example, to slice the sphere $x^2 + y^2 + z^2 = 10$ along the plane $y = 5$ we simply need to render $x^2 + (5)^2 + z^2 = 10$. We can conveniently use the tools from the lower dimensional mode to render this.

5 4D Projections

In order to view 4 dimensional points on screen, we first have to project them onto some intermediate 3 dimensional screen, which can then be projected and rendered onto our 2D screen using the traditional graphics pipeline. This projection is described more thoroughly in other work [Hollasch 1991].

To apply this projection in Three.js, we had to modify the default "position" attribute to be of type **vec4** instead of **vec3**, adding in that extra coordinate. We then had to write a modified vertex shader to perform the projection to a 3D position before passing it to the default model, view and projection matrices.

5.1 Rendering Solids

Transforming points and edges using a vertex shader did not cause any problems. However it did cause issues when transforming a cube with triangular faces. This is because the normals are used to compute lighting as well as culling. Since moving a 3D cube in 4D space can result in it being inside out, the normals will all be pointing the wrong way.

To solve this, we simply abandoned using the normals. We rendered the faces with a solid "unlit" color, and then rendered the vertices and edges separately to delineate the faces. The result is a 3D solid that we can freely distort and still recognize all of its faces as seen in Fig 7.

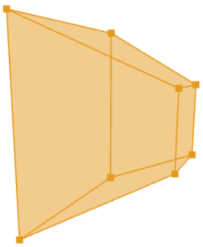

Figure 7: *A distorted 3D cube in 4D space is still recognizable.*

6 4D Convex Hull

Our viewer enables users to visualize the convex hull of a set of 4-dimensional points. We compute the convex hull using the quickhull algorithm, as described in *The Quickhull Algorithm For Convex Hulls* written by Barber, C. Bradford and Dobkin, David P. and Huhdanpaa, Hannu. Quickhull uses two geometric operations: *oriented hyperplane through d points, where d is the dimension of the points* and *signed distance to hyperplane*. These operations are important for checking properties of points with respect to a hull's *facet*, which is a (*d*-1)-dimensional face. In particular, they are necessary for finding the hyperplane determined by a facet of the convex hull, testing if a point is above or below a facet, and for finding the furthest point from a facet. Quickhull starts with an initial simplex and iteratively expands until it reaches the final convex hull. These operations are performed at each iteration, so we will discuss our approach to implementing them.

6.1 Oriented Hyperplanes of 4D Facets

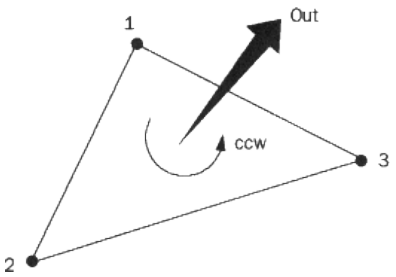

Figure 8: *A facet of a three-dimensional object with a counter-clockwise orientation and its corresponding facet normal, which points in the outside direction. Taken from www.fabbers.com/tech/STL_Format*

Quickhull relies on the oriented hyperplane determined by a facet to do much of its work. A hyperplane is said to be poperly oriented if its normal points outside of the hull.

Orienting facets is a common technique in computational geometry and has well-known solutions in three-space. The facets of a 3-dimensional object are 2-dimensional triangles. In turn, these 2-dimensional facets are represented by directed edges. A triangle can have one of two orientations: clockwise or counterclockwise. The direction of its edges determine its orientation (Figure 8). Hence, the usual approach to facet winding in three-space is to consistently direct the edges of a facet in a clockwise or counterclockwise fashion.

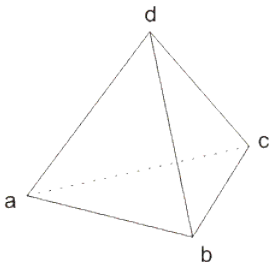

Figure 9: *A facet of a four-dimensional object. Taken from pyva.net/eng/articles/tetrahedra/*

Four-dimensional objects can be represented in a manner analogous to three-dimensional objects. Facets of four-dimensional objects are three-dimensional tetrahedra (Figure 9). Just as before, the normal of this facet is determined by the orientation of its edges. There are two possible orientations: $U = \vec{(ab)}$, $V = \vec{(bc)}$, and $W = \vec{(cd)}$, or $U = \vec{(ba)}$, $V = \vec{(bc)}$, and $W = \vec{(dc)}$. The normal of this facet is then calculated using the four-dimensional cross-product:

$$\times_4(U,V,W) = \begin{vmatrix} U_0 & U_1 & U_2 & U_3 \\ V_0 & V_1 & V_2 & V_3 \\ W_0 & W_1 & W_2 & W_3 \\ i & j & k & l \end{vmatrix} \quad (1)$$

By consistently orienting facet normals to point outside the convex hull, we can use them to determine if a point is outide the hull. Determining a point's position and distance with respect to a hyperplane extends trivially from lower dimensions [Hollasch 1991].

6.2 Computing the Initial Simplex

The quickhull algorithm starts with an initial 4-simplex, which is a total of five points. Suppose we want the convex hull for a set of points P. We begin computing the initial 4-simplex by selecting the pair of points in P with the greatest distance between them. These two points then determine a line in 4-space, and are the first two points of the simplex. We then select a third point in P which is the greatest distance from that line, and add that point to our simplex. Our simplex is now a triangle in 4-space. We proceed by adding a fourth point that is the greatest distance from the plane determined by that triangle, forming a tetrahedral simplex. Finally, we add a fifth point which is the greatest distance from the hyperplane determined by the tetrahedral simplex, Thereby completing the desired 4-simplex.

Once we have constructed our initial 4-simplex, we need to ensure that the facets of this simplex are properly oriented. This can be accomplished by checking the normal of each facet in the simplex against the point of the simplex not contained in that facet. This process is outlined in Algorithm 1.

Algorithm 1 Orienting The Initial Simplex

```
1: procedure ORIENTSIMPLEX
2:     F ← facets of the initial simplex
3:     P ← points of the initial simplex
4:     for all facets F' in F do
5:         P' ← the point in P not in F'
6:         if P' is above F' then
7:             Reverse the orientation of F'
```

6.3 Adding New Facets

One step of the quickhull algorithm is adding new facets after computing the horizon of some point. After these new facets are constructed, it is necessary to ensure that they are properly oriented. Observe that the visible set of some point is necessarily below the facets constructed using that point and its horizon (Figure 10). Hence, we can orient each new facet constructed with some point with respect to a point in its visible set. This procedure is outlined in Algorithm 2.

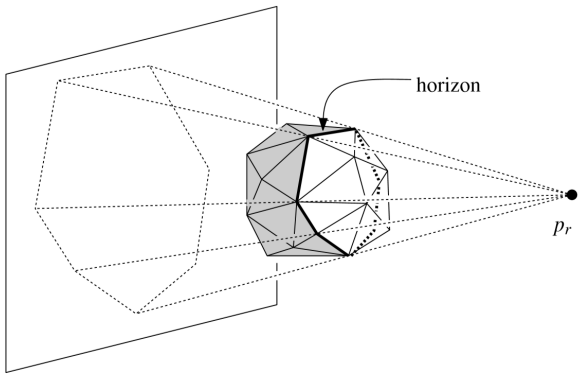

Figure 10: *The horizon of a point P_r. The set of facets bounded y the horizon is called the visible set. Taken from de Berg, Mark.*

Algorithm 2 Orienting New Facets Constructed From the Horizon

```
procedure ORIENTNEWFACETS
    F ← facets constructed from the horizon and some point p
    c ← the centroid of some facet in the visible set of p
    for all facets F' in F do
        if c is above F' then
            Reverse the orientation of F'
```

7 4D Rotations

There are 6 planes of rotation in 4D: XY, XZ, YZ, XW, YW, ZW. Our first attempt to compute rotations was to use 6 rotation matrices and compose them together in a vertex shader, essentially extending the 3 Euler angles from 3 dimensions to 6. This led to the dreaded gimbal lock.

While it is still possible to use quaternions to solve this rotation problem in 4D as in 3D, we opted instead to implement it using Rotors from Geometric Algebra [Vince 2008], which had the promise of being more generalizable to higher dimensions.

A rotor R is defined as a *multivector* which has a scalar part and a *bivector*. To rotate by some angle θ, define:

$$R = cos(\frac{\theta}{2}) + sin(\frac{\theta}{2}) \cdot A$$

Where A is a bivector representing the plane of rotation.

We used a Javascript port of the Versor geometric algebra library (versor.js) to implement this. Once the Rotor object has been constructed, we can apply it to a vector to get a new rotated vector.

To integrate this with Three.js, we first initialized the four basis vectors. Every time we applied a rotation, we apply them to all 4 vectors. Finally, on every frame, we create a 4×4 matrix using the mutated basis vectors as the columns. This can then be passed to a vertex shader that applies the rotation before the projection step.

This allowed us to rotate on all 6 planes easily and without running into gimbal lock. The Rotors were also easy to express in Javascript using the versor library.

7.1 User Input

We chose to use keyboard input for 4D rotations. We mapped the following keys to the 6 planes:

Keys	Plane
A,D	XW
W,S	YW
Q,E	ZW
J,L	XY
I,K	XZ
U,O	YZ

The keys were chosen to feel like arrow keys on a QWERTY keyboard, where each pair moves back and forth along the plane.

8 Classroom Use

Each mode contains a variety of built-in examples to show the user what's possible and encourage exploration. The greatest value of this tool comes not from any one of the modes but from looking at phenomena across dimensions.

Here we'd like to present a few examples where the app can present interesting insights to students in a classroom setting.

8.1 Slicing a Tube

If you take a cartesian equation of a sphere in 3D $x^2+y^2+z^2 = 10$ and remove the z variable, you get a tube:

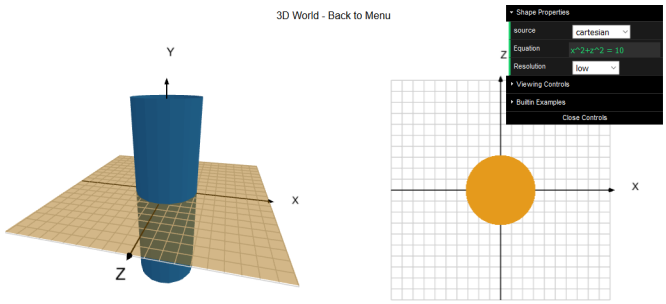

Figure 11: *A 2D circle might be a slice of a sphere, or a tube*

What's interesting here is that the the 2D slice is still a circle. If that's all we were looking at, we might think this object is a sphere. It's only until you slice along the other axes do you realize the true nature of this shape.

Similarly, if we graph $x^2 + y^2 + z^2$ in 4D, we get a slice that's a sphere. But we know this shape should be a hyper-cylinder, and we can verify this by slicing through the other axes.

We hope that by being able to switch back and forth like this that students will find the process illuminating and that seemingly obscure phenomena in 4D can be reasoned about through analogies like this.

9 Future Work

We've been able to put together a cohesive set of tools for exploring custom objects in 2D, 3D and 4D, but our app leaves much to be desired, especially in 4D mode.

9.1 Marching Tesseracts

One important missing feature is the ability to see a projection of 4D cartesian equations. To do this, we would have to generalize the marching cubes algorithm to a marching "tesseracts". With marching cubes, we could inspect the $2^3 = 256$ possibilities of intersections and reduce them down to 15 unique cases after discarding symmetries.

In 4D, we have $2^16 = 65,536$ cases to consider, and it's not obvious what the isosurface should look like.

9.2 Filling in Slices of Concave Shapes

Our slicing method correctly computes the boundary of any shape, but we do not have an algorithm for filling in these shapes correctly n 3D.

9.3 Embedding Slices in Projection View

It is very useful to be able to see where the lower dimensional slice lives in the higher dimensional space. We know any slice of a sphere is a circle, but being able to highlight where that circle lives on that sphere would be particularly illuminating, especially in 4 dimensions.

9.4 Rotated Slicing Planes

It would be nice to be able to extend our algorithms to work for planes in arbitrary rotations, which would allow students to explore even more interesting shapes and slices.

9.5 Sharing Discoveries

As this is mainly an educational tool, we would like to develop a protocol within the app to be able to save, share, and build on other students' creations.

Conclusion

The aim of our viewer has been to allow users to grasp higher-dimensional space via dimensional analogy. We hope that we have convinced readers of the power of this pedagogical tool. Our viewer offers real-time viewing of projection and slicing of any 2D, 3D, or 4D object represented by a cartesian equation, a parametric equation, or a convex hull. It is our intention that students may leverage this application to draw mental connections between the 2D and 3D worlds they are familiar with and the 4D world that is just out of reach.

Our source code and documentation is publicly available on GitHub, open for any developer to examine and use. We hope other students will find our implementation useful both as a reference and as a sandbox to further extend and develop 4D algorithms and visualization tools.

References

4D Visualization Document. `http://eusebeia.dyndns.org/4d/vis/vis`. Accessed: 2017-11-29.

4D Toys. `http://4dtoys.com/`. Accessed: 2017-11-29.

ABBOTT, E., ROBERTS BROTHERS (BOSTON, M., WILSON, J., SON, AND UNIVERSITY PRESS (CAMBRIDGE, M. 1885. *Flatland: Romance of Many Dimensions*. Roberts Brothers.

BARBER, C. B., DOBKIN, D. P., AND HUHDANPAA, H. 1996. The quickhull algorithm for convex hulls. *ACM Transactions on Mathematical Software 22*, 4 (Jan), 469483.

Polygonising a scalar field. `http://paulbourke.net/geometry/polygonise/`. Accessed: 2017-11-29.

DE BERG, M. 2008. *Computational Geometry: Algorithms and Applications*. Berlin : Springer.

4D Draw. `http://geometrygames.org/Draw4D/index.html`. Accessed: 2017-11-29.

HOLLASCH, S. R. 1991. *Four-Space Visualization of 4D Objects*. Master's thesis, Arizona State University.

MathMod. `https://sourceforge.net/projects/mathmod/`. Accessed: 2017-11-29.

Miegakure. `http://miegakure.com/`. Accessed: 2017-11-29.

NOLL, A. M. 1967. A computer technique for displaying n-dimensional hyperobjects. *Communications of the ACM 10*, 8 (Jan), 469473.

VINCE, J. J. A. 2008. *Geometric algebra for computer graphics*. Springer, London.

ZHOU, J. 1991. *Visualization of Four Dimensional Space and Its Applications*. PhD thesis, Purdue University.