KTH Royal Institute of Technology

DH2323

Project Report

Zhang, Tianli (tianli@kth.se)

Project Report : 4D Rendering

Abstract

This report describes the design and details in implementation of a 4D rendering system based on Unity. The 4D rendering system provides two ways of viewing a 4D model in 3D object space, including projection and cross section, which can be normally shown on a 2D screen or other suitable devices. Supporting features including limited 4D rotation and translation are provided, which allows different viewing angle of a 4D model.

Introduction

As all of the known creatures lives in the 3D world, 4D space is always an interesting direction to explore in visualization. Normally, there are no differences in visualization between whatever dimensional spaces -- the fourth dimension W just acquire an additional unit vector in orthogonal projections or an additional disappearing point in perspective projection -- the only thing matters is to convey the information of additional dimension to viewers effectively.

Cross section is a good way to achieve this. As what we can define a position and angle of a cross section of a 3D model: the position and the normal vector of a plane, in 4D space the way of determining the properties of a cross section is similar: the 4D position and the 4D normal vector of a hyperplane. As a result, the hyperplane, whose direction is determined by three individual vectors that is perpendicular to each other, and also perpendicular to the normal vector, can be easily projected into 3D space because of the loss of information of one dimension.

As a good example, the computer game Miegakure[1] show us how a 4D modeled scene in computer games would looks like. According to the developers, they modeled all objects in 4D, that is to have all geometric elements presented in 4D perceptions, such as a 4D vector for a vertex; they adapted the method of cross section to actually show part of the world to viewers. As seen in the introduction video, the cross section method works well in application. However, to fully understand or take overall control of a 4D model, there show be ways to acknowledge the full information about one 4D model at once.

For a full view of a 4D model, we need to project the model directly into whatever space we want to view it in. As we ordinarily percept objects in 3D spaces, it's beneficial to generate projected 3D information for universal use, such as presenting the information onto 2D screen using normal processing ways of computer graphics, or give the viewer a direct view of the 3D space using VR techniques, but that will be afterwords, and are also showing the potential of the 4D rendering system.

For the projection from 4D to 3D, there are two choices: **perspective projection** and **orthogonal projection**.

Perspective projection works by scaling the positions (x_i, y_i, z_i, w_i) according to the amount of offset w_i on W axis. This shares similarity with perspective projection in 3D, as the transformed positions is projected keeping their 2D positions but scaled according to the amount of offset on Z axis. As a result, tesseracts is a good example of perspective projection in 4D, as a tesseract is a possible view in perspective projection of a hypercube, which shows the 8 vertices with different offset on W axis with the other 8 vertices scaled with the scaling origin at the center of projected cube-like 3D model.

Orthogonal projection in 4D works just like orthogonal projection in 3D. As a 4D vector **v** can be decomposed into a summation of scaled unit vectors x**i** + y**j** + z**k** + w**l**, all of the four unit vectors would have its projected vector on screen calculated. For consistency with cross section methods mentioned above, we would have projected vectors of X, Y and Z axis kept ordinary with current 3D space definition, and have the vector of W axis affected by different factors other than the camera angle. A simple and wise choice is to always keep the projected vector of W axis perpendicular to the direction of camera, which means motions along W axis will always appears moving "forwards" or "backwards" in camera space.

Finally, for a quick mention, 4D rotations shares similarity in Algebra terms, as basic rotation matrices have 4 entries changed based on sine or cosine of rotated angle. However, 4D rotations doesn't have apparent "axes", because a rotation typically shifts two axes in appropriate space but keeps all other axes in higher dimensional spaces, which means a plane of points stays still during the rotation. Similar to Euler angles in 3D, one need 6 Euler angles in order to generate all possible rotations in 4D space.

For this project, I especially choosed **cross section** and **orthogonal projection** (referred as "projection" below) for 4D rendering, along with limited 4D rotation and translation to affect the direction of cross section and projection. For 4D rotation, only three of the Euler angles is chosen, as mentioned as "limited": WX, WY and WZ angles (represented with shifted axes, as points on positive part of the first axis is rotated towards the second axis).

For a direct sense of all features presented in this project, please refer to the online interactive demo in my blog[2].

Implementation & Results

1 Data Structure for 4D Models

A 4D model, in addition to what usually a 3D model consists of, vertices and faces to be spoken out, to generate proper 3D model by cross-section, has an extra basic geometric element called cell. A cell is typically a 3D model, but with all vertices in 4D space. A cell is similar to a face to 3D models, as a face of a 3D model is typically stored as a submesh of triangles, and a cell of a 4D model can be stored in a similar manner.

I decided to use these fields as the properties of the Mesh4D class based on Unity's geometry classes (noted in programming language C#, same below):

```
public class Mesh4D {
    public Vector4[] vertices;
    public Vector3Int[] triangles;
    public int[][] cells;
}
```

In this implementation, all vertices are 4D vectors, triangles is still stored with indices of each three vertices, and cells is stored with lists of triangle indices which form the cells. For example, a 5-cell contains 5 cells with each cell consists 4 triangles as a typical tetrahedron with 4D vertices; a hypercube or 4-cube contains 8 cells for each direction of 4 axes with each cell consists 12 triangles for 6 rectangular faces as a typical cube with 4D vertices.

2 4D Model Generation

For examination of the data structure and later-on rendering systems, I created two preset of 4D models together with their generation algorithms. These 4D models are pentachoron (5-Cell) and hypercube (4-Cube).

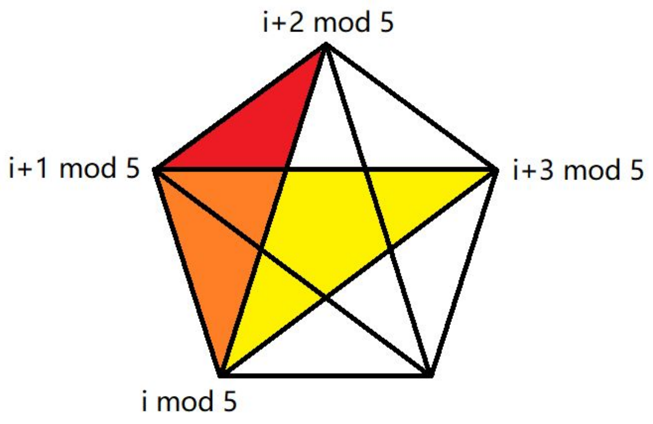
2.1 Pentachoron

A pentachoron, or formally a 5-cell, is a 4D object formed by 5 vertices in which every pair of vertices has the same distance in between.

For a easier setup, I put a tetrahedron in xyz-hyperplane and added a 5th vertex on top of the 4 vertices in w direction. According to Wikipedia[3], the vertices of a 5-cell with edge length $2\sqrt{2}$ are:

$$\begin{gathered}(1, 1, 1, -1/\sqrt{5}) \\ (1, -1, -1, -1/\sqrt{5}) \\ (-1, 1, -1, -1/\sqrt{5}) \\ (-1, -1, 1, -1/\sqrt{5}) \\ (0, 0, 0, \sqrt{5} - 1/\sqrt{5})\end{gathered}$$

For triangles and cells, I mapped all the indices within patterns according to this type of 2D projection of the 5-cell:



With this projection, I divided the triangles into 2 types with their vertex indices determined by triangle index #i:

- Side triangles #i, (i mod 5, i+1 mod 5, i+2 mod 5) for i ∊ [0, 4]
- Middle triangles #i, (i mod 5, i+1 mod 5, i+3 mod 5) for i ∊ [5, 9]

And also each cell with cell index #j contains 2 triangles of each kind, 4 triangles in total:

- Side triangles #j, #j+1 mod 5
- Middle triangles #j+5, #(j+2 mod 5)+5

With these informations, a 5-cell is complete as a Mesh4D.

2.2 Hypercube

To actually determine the mapping between indices, functional programming is acquired and several lambda expressions is used for index mapping.

In the first place, indices of axes is defined with binary codes:

```
const int x = 1, y = 2, z = 4, w = 8;
```

This will benefit us in representing constants for axes, and also in combining multiple axes or binary values on axes together without order.

After preparations, first of all we need a mapping from indices of rectangle faces to vertices:

```
Func<int, int[]> rect2vert = (int rect) => {
    int[] dim; // dim = { dimMain1, dimMain2, dimSub1, dimSub2 }
    switch (rect / 4 % 6) {
        case 0: dim = new int[] { x, y, z, w }; break; // xy plane
        case 1: dim = new int[] { x, z, y, w }; break; // xz plane
        case 2: dim = new int[] { y, z, x, w }; break; // yz plane
        case 3: dim = new int[] { x, w, y, z }; break; // xw plane
        case 4: dim = new int[] { y, w, x, z }; break; // yw plane
        case 5: dim = new int[] { z, w, x, y }; break; // zw plane
        default: return null;
    }
    rect = rect % 4;
    int mainIndex = dim[2] * (rect & 1) + dim[3] * ((rect & 2) >> 1);
    return new int[] { mainIndex, mainIndex + dim[0], mainIndex + dim[1],
                       mainIndex + dim[0] + dim[1] };
};
```

This function receives indices of 24 rectangular faces of a hypercube. The orientation, which is determined by the two main-dimensions, of the rects is changed every 4 indices, as there are exactly 4 rects with the same orientation. For each orientation, I also indicates which the other two sub-dimensions (that is vertical to the rect) are. By combining sub-indices, which is indices of rects with the same orientation from 0 to 3, with the two sub-dimensions, we got the indices of vertices that the rectangle is formed; by adding offset determined by two main dimensions, we got the exact four numbers of indices.

In response to rect2vert, there is a reversed function that converts main-dimensions and variables on sub-dimensions into indices of rects:

```
Func<int, int, int> dim2rect = (int dimsPlane, int dimsVar) => {
    switch (dimsPlane) {
        case x + y: return 0 + dim2idx(z, w, dimsVar); // xy plane
        case x + z: return 4 + dim2idx(y, w, dimsVar); // xz plane
        case y + z: return 8 + dim2idx(x, w, dimsVar); // yz plane
        case x + w: return 12 + dim2idx(y, z, dimsVar); // xw plane
        case y + w: return 16 + dim2idx(x, z, dimsVar); // yw plane
        case z + w: return 20 + dim2idx(x, y, dimsVar); // zw plane
    }
    return 0;
};
Func<int, int, int, int> dim2idx = (int dim1, int dim2, int dimsVar) =>
    ((dim1 & dimsVar) == 0 ? 0 : 1) + ((dim2 & dimsVar) == 0 ? 0 : 2);
```

It’s clear that every possible orientations of the faces is matched to a main-index (i.e. 0, 4, 8 … 20), and with parameter for the two other dimensions, the sub-indices is also determined.

With dim2rect, we can easy build up a 3D cell with 2D rects for the 4D hypercube. With the three dimensions that a cell lying on and the one parameter for the other dimension, the 6 rects is determined by switching out each of the three dimensions in order and also iterate the parameter for each switched out parameter. Also don't forget that we need triangle indices here, so every index #i of rectangle is replicated as #i*2 and #i*2+1:

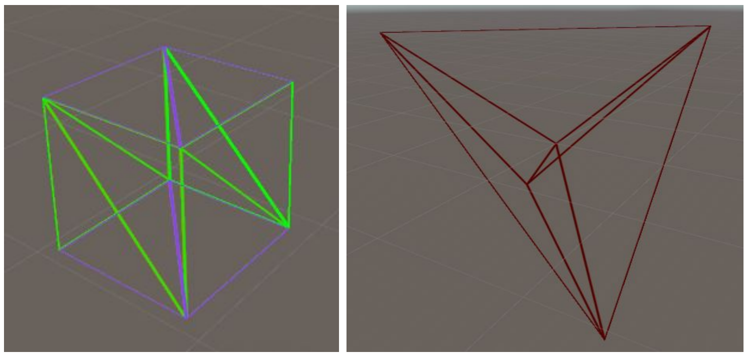
```
Func<int, int, int, int, int[]> dim2cell =
    (int dim1, int dim2, int dim3, int dimVar) => new int[12] {
        // Orientation dim1-dim2
        dim2rect(dim1 + dim2, dimVar) * 2,
        dim2rect(dim1 + dim2, dimVar) * 2 + 1,
        dim2rect(dim1 + dim2, dim3 + dimVar) * 2,
        dim2rect(dim1 + dim2, dim3 + dimVar) * 2 + 1,
        // Orientation dim1-dim3
        dim2rect(dim1 + dim3, dimVar) * 2,
        dim2rect(dim1 + dim3, dimVar) * 2 + 1,
        dim2rect(dim1 + dim3, dim2 + dimVar) * 2,
        dim2rect(dim1 + dim3, dim2 + dimVar) * 2 + 1,
        // Orientation dim2-dim3
        dim2rect(dim2 + dim3, dimVar) * 2,
        dim2rect(dim2 + dim3, dimVar) * 2 + 1,
        dim2rect(dim2 + dim3, dim1 + dimVar) * 2,
        dim2rect(dim2 + dim3, dim1 + dimVar) * 2 + 1,
    };
};

var cells = new Mesh4D.TriangleArray[8];
cells[0] = new Mesh4D.TriangleArray(dim2cell(x, y, z, 0)); // xyz0 cube
cells[1] = new Mesh4D.TriangleArray(dim2cell(x, y, z, w)); // xyz1 cube
cells[2] = new Mesh4D.TriangleArray(dim2cell(x, y, w, 0)); // xyw0 cube
// ...
```

3 Projection

3.1 Model-Local Projection

For a quick view of 4D models for sanity check of the data structure and model generation algorithm mentioned above, I created a model-local projection algorithm that uses different disappearing point for each individual models. In short, this is not a good projection algorithm which should keep consistency among all models in the game scene, but this creates basic feedback of additional information on the fourth dimension.

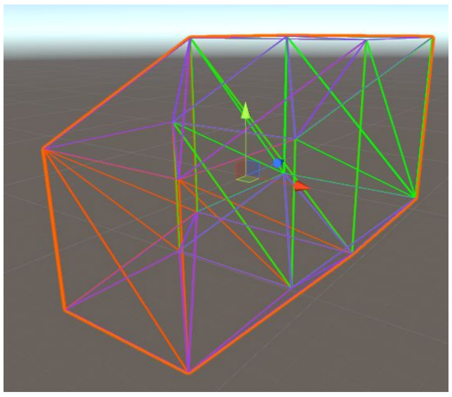
By ignoring the 4th dimension and show all vertices with XYZ info, an axis-aligned hypercube looks like a normal cube, and the 5-cell I've built above looks like a tetrahedron with a center vertex:



By simply applying an offset to a direction based on the distance on W axis:

```
v3d = new Vector3(v4d.x, v4d.y, v4d.z) + Vector3.one * 0.57735f * v4d.w
```

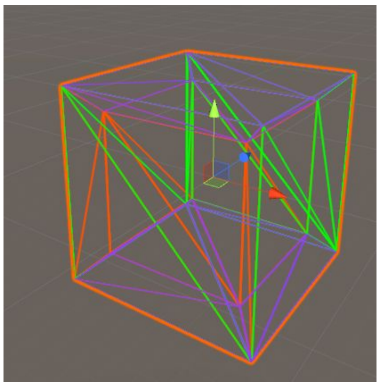
we will create this:



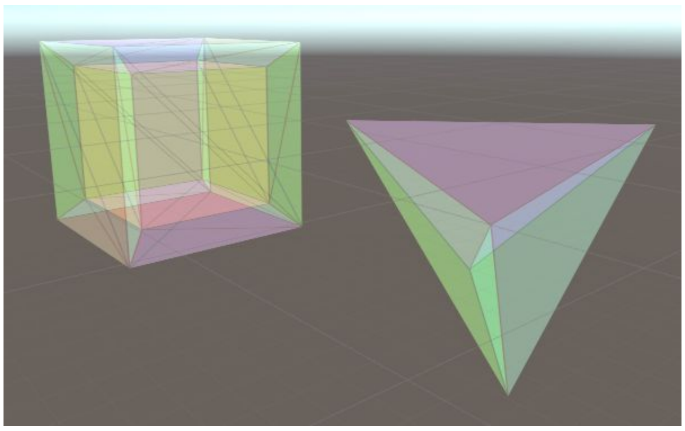
Instead of the equation above, I used a especially designed perspective projection equation:

```
v3d = new Vector3(v4d.x, v4d.y, v4d.z) * Mathf.Pow(1.35f, v4d.w)
```

I mapped w axis into some depth of view, but I used power instead of inverse so to match all possible w values to (0, +Inf). This would create a familiar view of a hypercube, which is called a tesseract:



With adequate shader, it would looks convincing that the 4D data structure and model generation is working correctly:

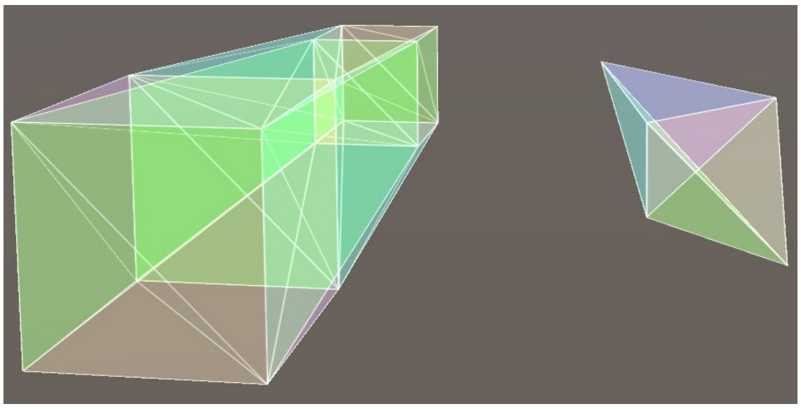


3.2 Scene-Space Orthogonal Projection

As expected, I created a projection method that keeps consistency in scene space for all models. The consistency is kept by using a uniform vector for all model to project their unit vector of W axis on. For example, if we choose the direction of main camera as the vector, we get equation:

```
v3d = new Vector3(v4d.x, v4d.y, v4d.z) +
      camera.forward * projectionFactor * v4d.w
```

With a high projection factor, a timelapse effect, in which the W axis is precepted as time, is created:



3.3 Cross Section Algorithm in 4D

Compared with my test of creating cross sections of 3D models [4], the procedure of creating a cross section of a 4D model appears similar in the procedure, as both of them are following the principle that by doing cross section any elements of original model generates new elements with its dimension decreased by one.

In 3D, it's easy to find out that every basic elements of a 3D object that intersects with the cross section plane generates a responding lower dimensional element, e.g. a line generates a point, and a facet generates a line segment. Assuming every facets to be triangles, we would infer that every facet that is not on the cross-section plane would have either 0 or 2 edges that intersect with the plane. As a closed mesh would have each edge shared between exactly two facets, by graph theories, the line segments from cross section would only creates loops in graph, which is typically polygons.

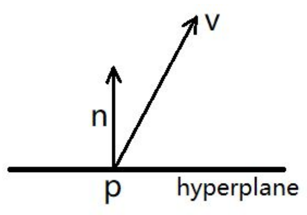
By extending this to 4D, a cell of a 4D model has similarity when creating a cross section with a hyperplane. Because connectivity of all edges and facets is still valid, the conclusion of loops in graph still make sense, but the polygons would have all vertices in 3D, not necessarily coplanar.

As a natural result, the cells of a 4D model creates polygons in 3D in cross sections. As the volume of the cells matters, the polygons can be treated as faces by filling them up with triangle facets with any triangulation algorithm. With all the triangle facets, the necessary information for a 3D model, typically a mesh, is complete.

The detailed procedure of generating a 3D model of cross section is listed below:

(1) Classify Vertex Sides

For this part, we calculate whether a vertex is on the positive or negative side of the hyperplane of cross section. It is determined by signature of dot product of the normal vector **n** of the hyperplane with the distance vector from the center of the hyperplane **p** to the given vertex **v**. A figure of 2D substitutes below:



(2) Find Intersecting Edges

With side information of vertices, we can determine which edges are intersecting with the hyperplane of cross section, i.e. the two vertices of the edge are on different side of the hyperplane.

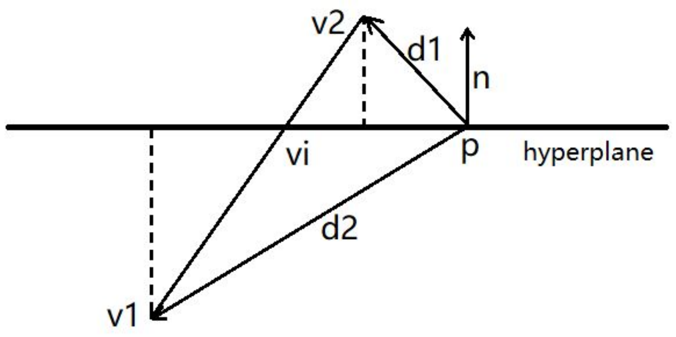
For every triangle facet, we know that there could be at maximum 2 edges that could intersects with a hyperplane at the same time, we record these edges in pairs:

```
// key & value for two pair of indices of edge vertices
var clippedEdgePairs = new List<KeyValuePair<Vector2Int, Vector2Int>>();
```

(3) Calculate Intersection Points

Before we fill out informations of a 3D model, we have to determine the positions of all intersection points of edges.

For any edge that has a intersection point, the ratio of length of two parts of the edge is calculated by projecting equivalent vectors $\mathbf{d}_1$ and $\mathbf{d}_2$ to the normal vector $\mathbf{n}$ of the hyperplane and measure the length of projected vectors. A figure of 2D substitutes below:



Calculation in C#:

```
// calculate the ratio of intersection point by measuring the ratio
// of equivalent vectors projected on plane's normal vector
float k = Vector3.Dot(planeCenter - v1, planeNormal) /
          Vector3.Dot(v2 - v1, planeNormal);
// then simply use the ratio to determine the exact intersection point
intersections[i] = v1 + (v2 - v1) * k;
```

(4) Generate Connectivity Graph

For this task, we are going to fill out a adjacency list as the connectivity graph, where the edges are treated as vertices of the graph and whether two edges are in pair determines the connection of two vertices, so that we can iterate all intersection points in a polygon in the following steps. Other methods such as adjacency table is also applicable.

To distinguish edges (in model term) with vertices in reversed order, we should also maintain a look-up table that maps both edges to a same symbol; a self-incremental id is recommended.

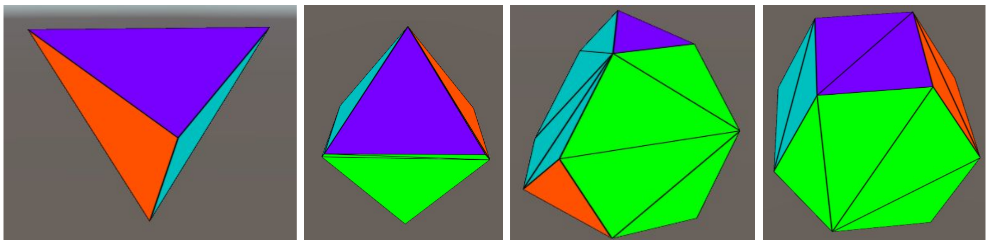
(5) Build Up Submeshes

First of all, start a single-trip travel on the connectivity graph beginning from any unvisited vertex to find a valid path. As the graph consist of only loops, every path we could find represents a individual loop.

For simplicity of the generated mesh, all vertices that is collinear with the neighbor vertices in the loop can be removed, as long as the loop still consists more that 3 vertices.

To properly fill the triangle information for the generated mesh, triangulation is required for every loop of vertices. For better delivery of which cells of a 4D model corresponding to which faces, we can arrange the submeshes to be exactly the same number and order as the list of cells.

With all steps above, the cross section of 4D models is created. For example, various valid cross sections of a hypercube can be created:



Future Works

In generating the faces of the 3D cross section models, there is no way to determine the normal vector based on give information about the 4D model. This is because no “direction” information can be generated for cells of a 4D model. Better research on the directions in 4D should be done, or there could be heuristic ways to determine the normal vectors of all faces, otherwise illumination will not work for the models generated by cross section.

Either these methods is beneficial for everyone to understand 4D geometries is not clear yet. Perceptual studies can be conducted to have a better understand of the effect of 4D rendering method among average people.

Another idea or improvement to this project could be extending the rendering onto VR platforms. With a native feeling of 3D space, the delivery of 4D informations is not limited by the bottleneck of a 2D screen.

References

[1] Youtube: Miegakure [Hide&Reveal] a true 4D puzzle-platforming game, https://www.youtube.com/watch?v=KhbUvoxjxIg

[2] 4D Rendering Demo, https://luicat.github.io/2018/05/23/4D-rendering-demo.html

[3] Wikipedia: 5-cell, https://en.wikipedia.org/wiki/5-cell

[4] 2D Cross-Section in Unity, https://luicat.github.io/2018/05/19/2D-clipping-in-unity.html