Projection and Rendering 1

Version date 2005-05-20

DRAFT: This document has not been reviewed and may contain errors.

Introduction

The real world has three spatial dimensions. It is equally possible to design a world with four or more spatial dimensions (time is not counted amongst the spatial dimensions here). The properties of such a world can easily be defined by mathematics. However it is difficult to imagine or visualise even simple objects and movements in such a world. The contents of this document describe methods to generate still and moving images of the 4D world, as an aid to that process. It describes methods for displaying a four dimensional world on a conventional two or three[1] dimensional display. The aim is to create an interactive world that can be understood and navigated, using current consumer-level computer hardware.

Problem

Objects in four dimensional space must be projected and rendered to a two or three dimensional screen image. In addition to still images, the solution should support real time rendering on current consumer-level hardware.

Analysis

There are many projections that map from four to three dimensions. Discarding one axis will do it, as will any 4×3 matrix where at least three of the columns are linearly independent. We need criteria to choose a particular transformation.

Desirable property	Description
Linear	In this case linear transformations are those that can be represented by a matrix multiplication vector addition[2]. These are simple to handle mathematically.
Physically based (realistic)	There must be an optical model of the four dimensional space. The projection should be derived from that model.
Familiar	For ease of understanding the optical model should be similarly to that of the 3D real world. Light should travel in straight lines, reflect from surfaces, etc.
Dimensionally consistent observer	The observer in the 4D world should be a 4D creature. The author postulates[3] that 3D entities in a 4D world are not *real*.
Retentive	The projection should retain information where possible. There are methods that lead to a rendered image of a 3D slice of a 4D world, but these discard a large part of the 4D world, which is undesirable.
Dimensionally equivalent (homogenous space)	Each dimension should be equivalent to any other, i.e. there should be no 'special' direction or axis. There is one exception: When projecting, the direction in which the viewer is looking is

[1] A current example of a three dimensional display would be displays that present a different picture to each eye, using LCD shutters, parallax or colour filters, to add stereoscopic depth to a scene. Although pseudo-3D, the depth component of each pixel is significant.

[2] The offset vector can be moved into the matrix in the usual way if homogenous coordinates are used.

[3] In full, the postulate is that 'Entities that have a region of influence of zero (hyper)volume are not real'. It is an extension of the nature of the 3D real world, with 'region of influence' accounting for particles like the electron, with potentially zero physical volume but an electric field extending over a volume.

	special, but if the viewer is looking along the *x* axis, the *y*, *z* and *w* axes should be treated equivalently by the projection. In practice this is difficult to achieve, as one dimension must be lost.
Clear and unambiguous	The projection should avoid sources of confusion as far as possible. These include small objects appearing to be large, distant objects appearing to be close, etc.

Table 1: Projection characteristics

For a first attempt, we to extend the concept of the human eye to 4-space. To simplify, I shall treat the eye as a pinhole camera, with a small pinhole at the origin, and a projection screen on the left. Only rays that go through the origin are permitted to reach the screen.

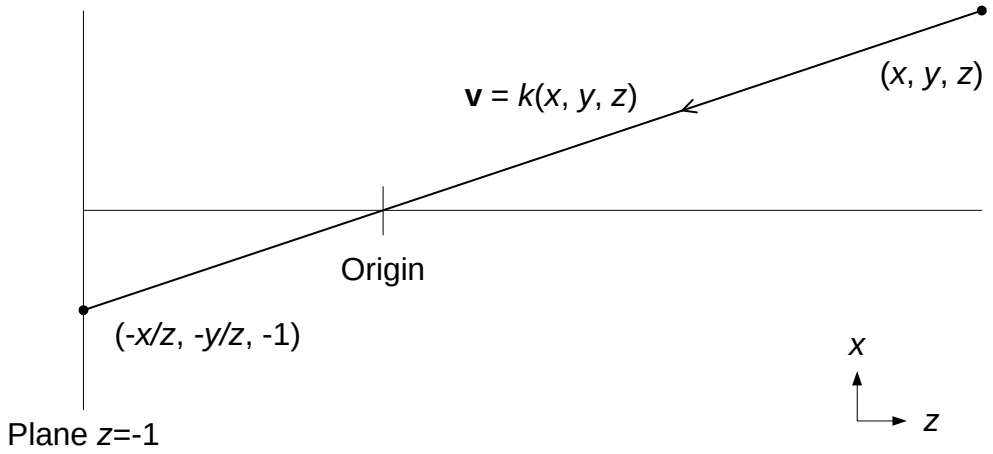

Figure 1: Pinhole camera with planar screen in 3-space

Figure 1 shows a conventional pinhole camera, with all points in $z>0$ projecting onto a plane at $z=-1$.

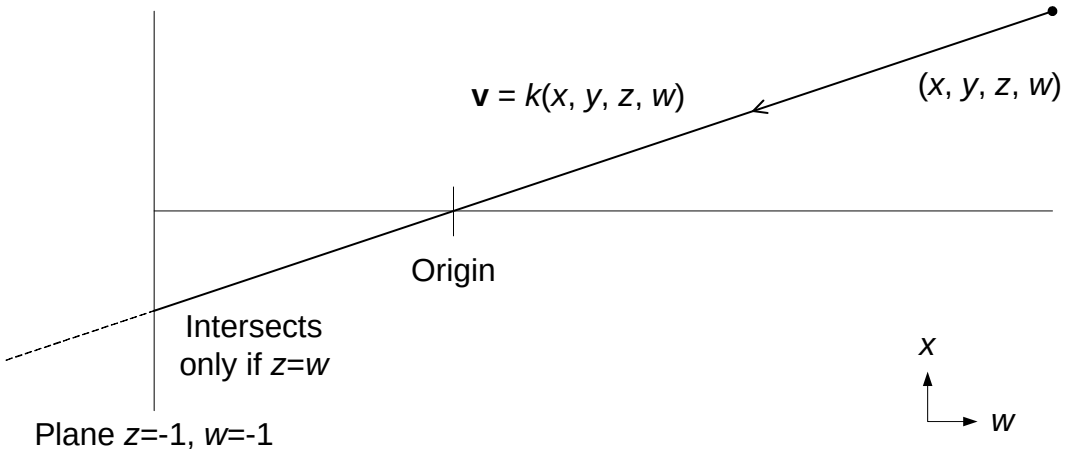

Figure 2: Pinhole camera with planar screen in 4-space

When extending the concept to 4-space in Figure 2, we hit a problem. A plane has two degrees of freedom and two constrained axes in 4-space, shown by the plane equation $z=w=-1$ above. The line equation $\mathbf{v} = k(x, y, z, w)$ will not satisfy the plane equation for any k unless $z=w$. So, almost none of the rays from points in $z>0$, $w>0$ hit the projection screen, and we

don't form a complete image of the scene. We only see parts of objects in the hyperplane $z=w$. This is a three-dimensional, zero-thickness slice of the scene.

This is not surprising, since a plane does not divide a 4-space into two parts, no more than a line divides a 3-space into two parts. A hyperplane has three degrees of freedom and one constrained axis, and does divide a 4-space into two parts, so is a better candidate for our projection screen.

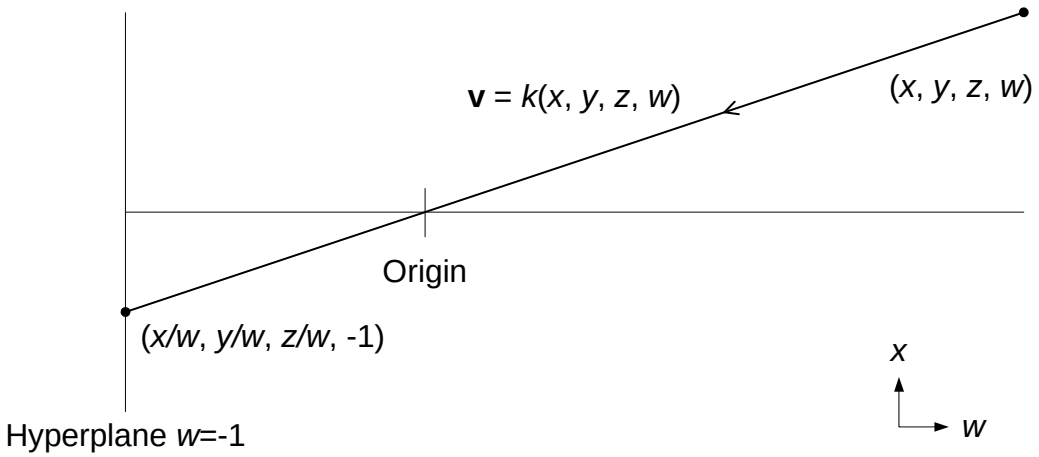

Figure 3: Pinhole camera with hyperplanar screen in 4-space

Using a hyperplane as a screen in Figure 3 generates equations similar to those for the 3-space camera in Figure 1. Each point in $w>0$ projects to a point on the hyperplane at w=-1.

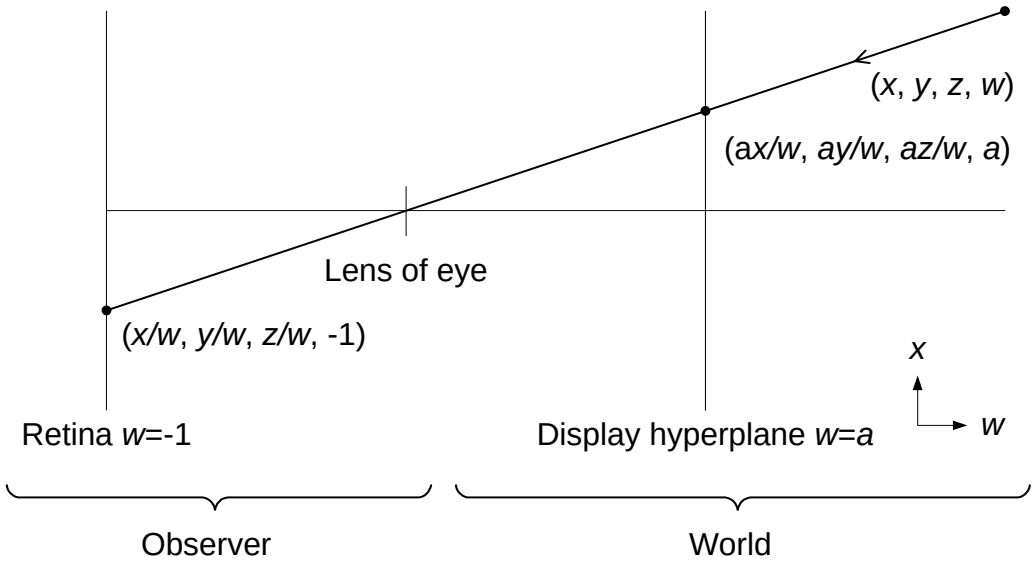

Figure 4: Observer and display in 4-space

Returning to the eye concept, this hyperplane is the retina of our four dimensional creature. Figure 4 shows the division between the observer and the external world. The hyperplanar display at $w=a$ mimics the view of the external world by projecting each point in $w>0$ into the display hyperplane. Both retina and display have extent in three dimensions and are thin in the other.

We can envisage a set of nerves leaving the retina to the left of Figure 4 (along the *w* axis), each from a particular 3D position on the hyperplane (*x*, *y*, *z*). The signals from these nerves are processed by the brain to recreate an internal construct of the 4D space being viewed. Once the signals leave the retina, subsequent processing doesn't necessarily require that the machinery exist in 4D space. A 3D brain similar to ours could theoretically do the same job for 4D space that ours does for 3D space, i.e. perform 4D spatial reasoning and navigation. However this facility is unlikely to arise without any stimulus or evolutionary advantage to be gained. In one sense, the task of the projection is to provide the best stimulus for a brain used to a 3D world to comprehend a 4D one.

Vertex projections

The vertex projection maps 4D vertices in the 4D world to 3D vertices on the display. The projections presented here assume the pinhole camera and hyperplanar retina and display geometry shown in Figure 4. This has implications for the projection:

- Only *w* is used in any perspective scaling and depth tests. We know whether one object is in front of another purely from their respective *w* coordinates[4].
- For a point at position **v**, any point *k***v** in 4-space will map to the same point on the screen.
- The *x*, *y* and *z* coordinates in the hyperplanar retina map to *x* and *y* in the display screen, and *w* in 4-space maps to *z* in the 3 dimensional display.
- For a flat display or retina of finite size, there will be limits of the form $-k < x < k$ or similar on coordinates within the hyperplane *x*, *y* and *z*. The exact form of the limits depends on the shape of the retina or display. Where the limit is exceeded, the associated point in 4D will not be visible[5].

Candidate vertex projections

This section presents two possible vertex projections. Both assume eye coordinates where the viewer is looking from the origin along the *w* axis, so *w* always maps to depth (*z*) on the display. The first projection is chosen for simplicity. It discards the *z* coordinate from 4-space and maps the other three directly to the coordinates of 3-space. The second maps the 4-space axes *x*, *y* and *z* radially into the display plane, and is useful for in situations where the user must aim in three degrees of freedom.

It is convenient to draw 4D-to-3D linear transformations as arrows in 3D representing the vector that each of the 4D axes map to. Figure 5 shows the discard-z projection. The diagram on the left shows the axes as they appear on the display. On the right, the view has been tilted slightly to reveal the *w* axis.

[4] This and the following refer to eye coordinates. The world is rotated and translated so that the camera or eye is sited at the origin looking along the *w* axis before performing this test.

[5] It is possible to design a wraparound retina much like the human eye, where there is no such limit. The equivalent wraparound display could be, for example, a box in which the viewer sits, where each side is a projection display. Only flat-display projections are dealt with here.

Figure 5: Discard-z projection

As the name suggests, this transformation simply discards z from the 4D coordinates. The transformation from eye to screen coordinates is:

$$\begin{pmatrix} x_s \\ y_s \\ z_s \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_e \\ y_e \\ z_e \\ w_e \end{pmatrix}$$

Equation 1: Discard z transformation

Points to note are:

- Equation 1 does not include a perspective transformation; these will be discussed later on.
- In this simple form, the resultant axes in 3D appear as columns of the matrix.
- For a purely 2D display, the resultant z_s is used only for depth tests.

Figure 6 shows the radial projection. Again the diagram of the right is tilted to reveal w. The other axes are coplanar.

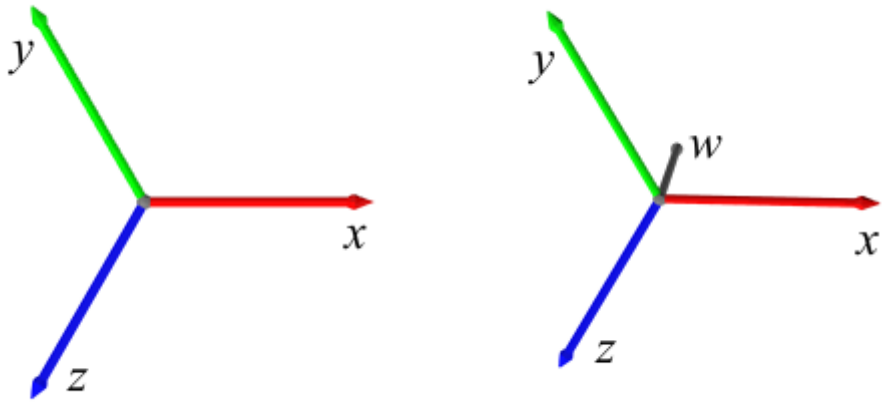

Figure 6: Radial projection

The radial projection attempts to treat all display plane axes (x, y and z) equivalently, by mapping them to the display 120° ($2\pi/3$ radians) apart. It is useful when the viewer must aim using rotations in the xw, yw, and zw planes, as the effect of each individual rotation is visually apparent.

First, we introduce scale factor k for the display plane coordinates.

$$k = \sqrt{\frac{2}{3}}$$

Equation 2: Scale factor *k*

The transformation from eye to screen coordinates is:

$$\begin{pmatrix} x_s \\ y_s \\ z_s \end{pmatrix} = \begin{pmatrix} k & k\cos(2\pi/3) & k\cos(4\pi/3) & 0 \\ 0 & k\sin(2\pi/3) & k\sin(4\pi/3) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_e \\ y_e \\ z_e \\ w_e \end{pmatrix}$$

Equation 3: Radial transformation

It is important to note that this does not solve the dimensional equivalence problem. We are still discarding a dimension. Whereas discard-*z* simply discarded the *z* axis, the radial projection discards an axis in the direction (1,1,1,0). Addition of any proportion of (1,1,1,0) to any vector in eye coordinates will not change the projected vector in screen coordinates.

The radial projection differs only from the discard-z projection by a rotation in 4-space. This means that, instead of using the projection matrix in Equation 3, we could apply a rotation to the eye coordinates first and then apply the discard-*z* projection from Equation 1. This will prove useful later, as it is more straightforward to support the discard-*z* projection using the OpenGL pipeline.

$$\begin{pmatrix} k & k\cos(2\pi/3) & k\cos(4\pi/3) & 0 \\ 0 & k\sin(2\pi/3) & k\sin(4\pi/3) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} k & k\cos(2\pi/3) & k\cos(4\pi/3) & 0 \\ 0 & k\sin(2\pi/3) & k\sin(4\pi/3) & 0 \\ \sqrt{1-k^2} & \sqrt{1-k^2} & \sqrt{1-k^2} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Equation 4: Conversion of discard-z to radial projection using a rotation

The matrix on the right of Equation 4 is a rotation matrix[6]. It allows us to interchange our projections using a rotation about the origin in eye coordinates in 4-space. As expected, it maps our discarded axis in this projection, (1,1,1,0), to (0,0,1,0), the discarded axis of the discard *z* projection.

Reusing the 3D projection

Projection and rendering must be performed using hardware essentially optimised for 3D. Programmable vertex shaders have brought flexibility to the 3D pipeline, but in certain areas the pipeline is still fixed.

In OpenGL the projection matrix projects from eye coordinates to clip coordinates. Both include an additional coordinate to allow perspective projections, translations and rotations to be contained in single matrices. Usually *w* is used to represent the additional coordinate in 3D, but since we have already used *w* for the fourth spatial dimension, *h* will be used. Clip

[6] Fundamentally, rotations preserve the distance between points and do not invert objects, i.e. turn them inside out or change their handedness. For a matrix, this implies that it is orthonormal and has determinant 1. The chosen value of *k* adjusts the determinant to 1 and makes this matrix a true rotation.

coordinates map the extent of the display device to a cube with unit extent along each axis, i.e. $-h \le x \le h$, $-h \le y \le h$, $-h \le z \le h$. The projection transform also takes part in perspective shortening by copying z into h.

$$\begin{pmatrix} x_c \\ y_c \\ z_c \\ h_c \end{pmatrix} = \begin{pmatrix} \frac{f}{aspect} & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & \frac{z_{Far}+z_{Near}}{z_{Near}-z_{Far}} & \frac{2z_{Far}z_{Near}}{z_{Near}-z_{Far}} \\ 0 & 0 & -1 & 0 \end{pmatrix} \begin{pmatrix} x_e \\ y_e \\ z_e \\ h_e \end{pmatrix}$$

Equation 5: The projection matrix in 3D

f is related to the half-angle of the view and z_{Near} and z_{Far} specify the z position of the near and far clipping planes; the area within them is mapped between $-h$ and h. The matrix scales the z axis appropriately for the depth buffer and copies z_e into h_e to perform the perspective transformation[7]. Normalised device coordinates are then generated by dividing thus:

$$\begin{pmatrix} x_{ndc} \\ y_{ndc} \\ z_{ndc} \end{pmatrix} = \begin{pmatrix} x_c/h_c \\ y_c/h_c \\ z_c/h_c \end{pmatrix}$$

Equation 6: Perspective scaling in 3D

Putting these together for z_{ndc} yields:

$$z_{ndc} = \left(\frac{z_{Far}+z_{Near}}{z_{Far}-z_{Near}}\right) + \frac{h_e}{z_e}\left(\frac{2z_{Far}z_{Near}}{z_{Far}-z_{Near}}\right)$$

Equation 7: z value in normalised device coordinates in 3D

i.e. a scaled and offset value mapping the $-z_{Near}$ and $-z_{Far}$ planes to -1 and 1 respectively.

Points to note are:

- The projection matrix and perspective scaling invert z, such that $1/z$ is used for depth tests.
- Although we have support for four coordinates in eye space there are difficulties in using those directly as the coordinates in 4D space. The nature of h_e as the divisor of the z_e coordinates is important to transform z correctly for the depth buffer, clipping and perspective-correct texture mapping. The latter will most likely require the $1/z$ relationship of the 3D pipeline.

The 4D projection matrix

A 4D projection matrix can be constructed by combining the standard projection matrix with the vertex projections of Equation 1 and Equation 3.

[7] OpenGL Technical FAQ 12.050 explains the operation of the depth transformation (http://www.opengl.org/resources/faq/technical/).

$$\begin{pmatrix} x_c \\ y_c \\ z_c \\ h_c \end{pmatrix} = \begin{pmatrix} \frac{f}{aspect} & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & \frac{z_{Far}+z_{Near}}{z_{Near}-z_{Far}} & \frac{2z_{Far}z_{Near}}{z_{Near}-z_{Far}} \\ 0 & 0 & -1 & 0 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_e \\ y_e \\ z_e \\ w_e \\ h_e \end{pmatrix}$$

Equation 8: Projection in 4D

The new matrix introduced is a homogenous version of the discard-*z* matrix. We now have 5 eye coordinates, which has pros and cons. It is useful because a translation vector can be placed in the final column as per 3D OpenGL. The drawback is that hardware is not well optimised for 5 element vectors. If programmable vertex shaders are available, the entire projection can be moved into the hardware using Equation 9.

$$\begin{pmatrix} x_c \\ y_c \\ z_c \\ h_c \end{pmatrix} = \begin{pmatrix} \frac{f}{aspect} & 0 & 0 & 0 & 0 \\ 0 & f & 0 & 0 & 0 \\ 0 & 0 & 0 & \frac{z_{Far}+z_{Near}}{z_{Near}-z_{Far}} & \frac{2z_{Far}z_{Near}}{z_{Near}-z_{Far}} \\ 0 & 0 & 0 & -1 & 0 \end{pmatrix} \begin{pmatrix} x_e \\ y_e \\ z_e \\ w_e \\ h_e \end{pmatrix}$$

Equation 9: Projection in 4D for programmable vertex shaders

Comparing with Equation 5, we have just inserted another coordinate z_e on the right hand side, and a column of zeros in the matrix that discards it. Although not used for projection, z_e will be required for lighting calculation, etc. To avoid 5 element vectors, it is desirable to use h_e=1 for all eye space vertices. This will allow the reuse of the usual 4-value interfaces as h_e will not need to be passed. In the shader, calculation can be performed in the following stages:

$$\begin{pmatrix} x_c \\ y_c \\ z_c \\ h_c \end{pmatrix} = \begin{pmatrix} \frac{f}{aspect} & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & 0 & \frac{z_{Far}+z_{Near}}{z_{Near}-z_{Far}} \\ 0 & 0 & 0 & -1 \end{pmatrix} \begin{pmatrix} x_e \\ y_e \\ z_e \\ w_e \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ \frac{2z_{Far}z_{Near}}{z_{Near}-z_{Far}} \\ 0 \end{pmatrix}$$

Equation 10: OpenGL shader projection calculation for h_e=1

This splitting of single matrices into a matrix and additive vector will be extended throughout the OpenGL pipeline in the following section. For now we note that a further trick can be played by setting z_e to the constant and modifying the matrix.

$$\begin{pmatrix} x_c \\ y_c \\ z_c \\ h_c \end{pmatrix} = \begin{pmatrix} \frac{f}{aspect} & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & \frac{2z_{Far}z_{Near}}{z_{Near}-z_{Far}} & \frac{z_{Far}+z_{Near}}{z_{Near}-z_{Far}} \\ 0 & 0 & 0 & -1 \end{pmatrix} \begin{pmatrix} x_e \\ y_e \\ 1 \\ w_e \end{pmatrix}$$

Equation 11: Modifying the projection matrix and input data to remove the additive vector

The vertex pipeline

The usual pipeline includes the model, view and projection transformations. Normally the model and view matrix are combined into one matrix. The projection matrix cannot be combined as well because the intermediate stage (eye coordinates) is required for lighting calculations, etc. In our 4D pipeline each transformation will be contained in a 4×4 rotation matrix and a 4 element additive vector[8]. To cope with this separation, we separate the modelview matrix into its model and view components.

$\mathbf{v_o}$: A vertex in the definition of an object

$\mathbf{v_w}$: That vertex transformed into world coordinates

$\mathbf{v_e}$: That vertex transformed into eye coordinates

$\mathbf{v_c}$: That vertex transformed into clip coordinates

$\mathbf{M_m}$: The model orientation matrix

$\mathbf{v_m}$: The model additive vector, i.e. the object position

$\mathbf{M_v}$: The view/camera orientation matrix

$\mathbf{v_v}$: The view additive vector, i.e. the camera position

$\mathbf{M_p}$: The projection matrix

$\mathbf{v_p}$: The projection additive vector, i.e. the offset for z_c

Equation 12: Notation

A vertex within an object undergoes the model, view and projection transformations shown below. Combined transformations are also shown.

[8] This is purely for convenience of implementation; a 5×5 matrix combining the two would do just as well.

Model

$$\mathbf{v}_w = \mathbf{M}_m \mathbf{v}_o + \mathbf{v}_m$$

View

$$\mathbf{v}_e = \mathbf{M}_v \mathbf{v}_w + \mathbf{v}_v$$

Projection

$$\mathbf{v}_c = \mathbf{M}_p \mathbf{v}_e + \mathbf{v}_p$$

Model and view

$$\mathbf{v}_e = \mathbf{M}_v(\mathbf{M}_m \mathbf{v}_o + \mathbf{v}_m) + \mathbf{v}_v = \mathbf{M}_v \mathbf{M}_m \mathbf{v}_o + \mathbf{M}_v \mathbf{v}_m + \mathbf{v}_v$$

Model, view and projection

$$\mathbf{v}_c = \mathbf{M}_p(\mathbf{M}_v(\mathbf{M}_m \mathbf{v}_o + \mathbf{v}_m) + \mathbf{v}_v) + \mathbf{v}_p$$

$$= \mathbf{M}_p \mathbf{M}_v \mathbf{M}_m \mathbf{v}_o + \mathbf{M}_p \mathbf{M}_v \mathbf{v}_m + \mathbf{M}_p \mathbf{v}_v + \mathbf{v}_p$$

$$= \mathbf{M}\mathbf{v}_o + \mathbf{v}$$

Equation 13: Vertex transformation pipeline

Equation 13 shows the effect of the model, view and projection matrices. If desirable, these can be combined into a single transformation using **M** and **v** as shown. However both **M** and **v** must be recalculated if any of the model, view or projection transformations change.

Optimisations

Given Equation 13, how much of the OpenGL pipeline can be reused? On hardware with programmable vertex shaders, the entire transformation can be kept within the accelerated hardware. Without them, only the projection matrix can easily be used. The modelview matrix could be used, but the additive vectors still need to be recalculated at each change of view, position or orientation, and this is similar in difficulty to calculating the vertex positions themselves.

Further aspects of projection

Reuse of the OpenGL pipeline

Element	Description
Normal vectors	Cannot be reused directly, as they are limited to 3 coordinates. A 4D unit vector can be stored using only three values (e.g. Euler angles) but this is not convenient for rapid calculations.
Lighting	Without programmable shaders only ambient lighting is reusable, as positions, distances and normals relate to 3D. With programmable shaders, the position interface can be used (4D), but the normal interface cannot (3D). Another interface can be reused for normals, or at least the additional normal coordinate. An interface that supports the glDrawArrays method would be ideal, such as the texture coordinate interface.
Texture mapping	Texture mapping has a rich coordinate interface and presents no problems. However OpenGL 3D textures cannot be used directly to generate final pixel values for the 3D faces of 4D objects in a single pass. A 3D texture returns the value of a pixel at the given three coordinates, whereas a 4D render with 3D textures requires the path integral along a line drawn through the texture.
Clipping	Can be reused. An additional method is required for hidden axis clipping, as user clip planes are inadequate for this purpose.

Table 2: Reuse of the OpenGL pipeline

Hidden axis clipping and hinting

In the discard-*z* projection, the *z* coordinate is discarded and takes no further part in the rendering process. This presents a number of problems:

- No clipping is performed on the *z* axis. This means that everything, regardless of its *z* value, can be drawn. This is in contrast to the *x* and *y* coordinates, for which objects are clipped to the viewing frustum.
- No distance shrinking occurs for the *z* axis. For *x* and *y*, distance shrinking occurs implicitly because the screen is further from the viewer as *x* and *y* increase. Although the effect is small for *x* and *y*, it can become large for *z* if *z* is not clipped. An object far away in *z*, say at (0,0,10000,1), would be drawn the same size as the same objects at (0,0,0,1). Its size will then be a strong function of the view orientation. This would lead to a confusing display.
- There is no way for the viewer to know the position of an object in *z* from its position and size on the display. Additional hinting is required if the view is required to aim at the object. To allow accurate aiming, the hinting must convey the *z* coordinate quantitatively.

Table 3 lists several methods for hidden axis clipping. These can be combined if desired.

Solution	Description
Alpha clipping	In this solution, objects are diminished in alpha as they move away from the view axis in *z*. Objects at distant *z* are transparent so are not seen. Since we are introducing transparency the renderer must also solve the order-dependency problem.
Additional clipping hyperplanes	Objects are clipped against hyperplanes $z=\pm k$. This method treats the *z* axis similarly to *x* and *y* so is consistent. However clipped edges can be visible on the display as hard and unnatural lines truncating visible objects.
Pseudo-perspective	Pseudo-perspective shrinks objects as they move away from the view along the *z* axis. Scaling is applied so that the object is the correct size for its distance from the viewer. The object keeps its place in the depth sort order because, although scaled, it has not moved further away in *w*. The guiding principle is that rotating the view in the *zw* plane (or any plane) should not change the size of the object, as the distance to it has not changed.

Table 3: Hidden axis clipping

Table 4 suggests various methods for hidden axis hinting. Again these can be combined if desired. Ideally these are done on a per-vertex or per-pixel basis, not per-object.

Solution	Description
Ghost image hinting	Objects are rendered with one or more translucent copies at an offset in *x* and *y* from the main object. The degree and/or direction of the offset depends on the *z* value at that point.
Blur hinting	Objects blur as they move away on the *z* axis. Although technically demanding, the concept of objects blurring as they move away in another dimension is appealing. A directional blur could be used to hint the sign of *z*.

Parameter hinting	Various rendering parameters can vary with *z*. Static qualities can be varied, like colour, texture, and illumination, as can dynamic qualities, such as a periodic wobble, with its amplitude dependent on the *z* coordinate. Even size or orientation could be used. However some of these effects may reduce the realism of the scene.
Stereoscopic hinting	The *z* axis could affect the stereoscopic displacement of objects. Even vertical parallax could be introduced, if the eye could make sense of it.
Instrumented gunsight	If hinting is required only to aim at an object, the renderer could add a display to the gunsight (or elsewhere) telling the user where the object within it is on the *z* axis. Control methods can be devised such that knowledge of the *z* axis is not be required to bring the target into the gunsight initially.
Texture blending	Using multiple textures and transparency, the resulting texture of an object can change with its position on the *z* axis.
Rolling or wobbling view	Either a continuous rotation or a wobbling of the view could be introduced in an attempt to make the *z* coordinate apparent. This may confuse the viewer however.

Table 4: Hidden axis hinting

Alpha clipping

Alpha clipping ensures that objects at distant *z* are not drawn, by making them transparent. Alpha is determined from some function of the *z* and *w* coordinates of a vertex or pixel. This is known as the hidden access clipping (HAC) function.

$$\alpha = f\left(\frac{z_e}{w_e}\right)$$

Equation 14: Alpha for alpha clipping

The smoothstep function is one candidate for generating smooth transistions between alpha values.

$$l = \frac{z_e}{kw_e}$$

$$t = \frac{1 + d - l}{2d}$$

$$\alpha = 1 \text{ if } l \le 1 - d$$

$$\alpha = 3t^2 - 2t^3 \text{ if } 1 - d < l < 1 + d$$

$$\alpha = 0 \text{ if } l \ge 1 + d$$

Equation 15: The smoothstep function for alpha

The value of *k* controls the position of the cutoff, and *d* controls its sharpness.

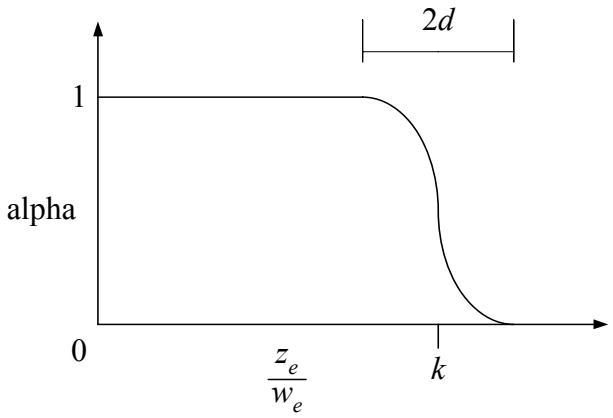

Figure 7: The smoothstep function for alpha

The value of alpha can change in a non-linear way as we move across polygons. If the mesh size, and hence the polygons, are large compared to the rate of change of alpha, artefacts will appear due to linear interpolation of the non-linear function across the polygon. In this case, either further mesh division, larger *d* or per-pixel alpha calculation in the fragment shader can be employed. A less steep or more linear HAC function is a further option.

Pseudo-perspective

Pseudo-perspective can be achieved by replacing the perspective transform. From Figure 3 we see that the perspective transform is:

$$\begin{pmatrix} x_p \\ y_p \\ z_p \\ w_p \end{pmatrix} = \begin{pmatrix} x_e/w_e \\ y_e/w_e \\ z_e/w_e \\ w_e \end{pmatrix}$$

Equation 16: Perspective transform for a 4D pinhole camera

Pseudo-perspective is required because an axis has been discarded, so we must first specify which axis has been discarded. In Equation 17 we use the discard-*z* projection, and the principle that rotation of the view in *zw* should not change the size of the visible object.

$$\begin{pmatrix} x_p \\ y_p \\ z_p \\ w_p \end{pmatrix} = \begin{pmatrix} x_e/\sqrt{z_e^2+w_e^2} \\ y_e/\sqrt{z_e^2+w_e^2} \\ z_e/\sqrt{z_e^2+w_e^2} \\ w_e \end{pmatrix}$$

Equation 17: Pseudo-perspective transform

Although shown above, z_p need not be calculated as the subsequent discard-z transformation will discard it.

$$f = \frac{w_e}{\sqrt{z_e^2+w_e^2}}$$

Equation 18: Pseudo-perspective correction factor

The correction factor in Equation 18 may appear potentially large, but objects that stray too far from the view axis will be clipped. For an object at angle θ off-axis in the zw plane, f=cos θ. Points to note are:

- The pseudo-perspective transform is non-linear.
- The correction is small: $0.85<f<1$ for a view half-angle of $<30^\circ$. If the view angle is small in zw it could be ignored.
- Objects move towards the centre of the screen when f is applied, as well as shrinking.
- When an object is being aimed at, its position will be close to $(0,0,0,k)$ and f will have little effect. Pseudo-perspective does not provide useful hidden axis hinting at the aiming point unless the aim is significantly off.
- The calculation presented is not rigorous; it is based on a guiding principle.

Hinting examples

Figure 8 and Figure 9 show the result of some of the hinting techniques[9]. A curved bar is rotated slightly about its central point by a small angle θ in zw so that at the top, $z_e \approx -\theta$, in the centre $z_e=0$ and at the bottom $z_e \approx \theta$.

Figure 8: Unhinted (left) and ghost hinted images (right)

In the unhinted image the z displacement is not visible. In the ghost hinted image the ghost images move further from the main image as the z displacement increases.

[9] For simplicity, a 3D bar is shown.

Figure 9: Blur hinted (left) and texture hinted images (right)

In the blur hinted image, the ends of the bar are blurred in proportion to their *z* displacement. With texture hinting, the surface texture changes with the *z* displacement.

More complex hinting is possible, including techniques that hint the direction as well as the value of the *z* displacement. For example, the blur could stream off to one side or the other, depending on the direction of the displacement. The pattern of ghost images could appear on different sides of the object for positive and negative *z*. The texture itself could be ghosted, so that another, transparent copy of the surface texture slides over the base texture as *z* changes, and so on. The hinting should depend on the *z* value in eye coordinates, and not just the displacement relative to the centre of the object, i.e. depend on the position as well as the orientation of the object.

Rendering

The above section has covered the pipeline for vertex projection. One major question remains. What should a 4D object look like? How do we render the faces of the object?

Faces of 4D objects

Consider one of the eight faces of the four dimensional hypercube or tesseract. We choose the face for which x=1 and y, z and w range from –1 to +1. This face has a single normal along the x axis (1,0,0,0). Light rays approaching from the region x>1 that impinge on the face undergo specular reflection, diffuse reflection and absorption. No rays impinge on the face from x<1, as those would be coming from inside of the object.

A light ray hits the face at a single point (1,y,z,w). The face has definite material properties at that point and reflects light in a predictable way. In the renderer, those characteristics could be represented as a combination of uniform and varying variables, and parametric and stored 3D textures.

However, we hit a problem. Because the projection to the screen necessarily discards one axis, each pixel on the screen will correspond to a *line* through the 3D texture. By considering a ray model we shall see why.

Ray models

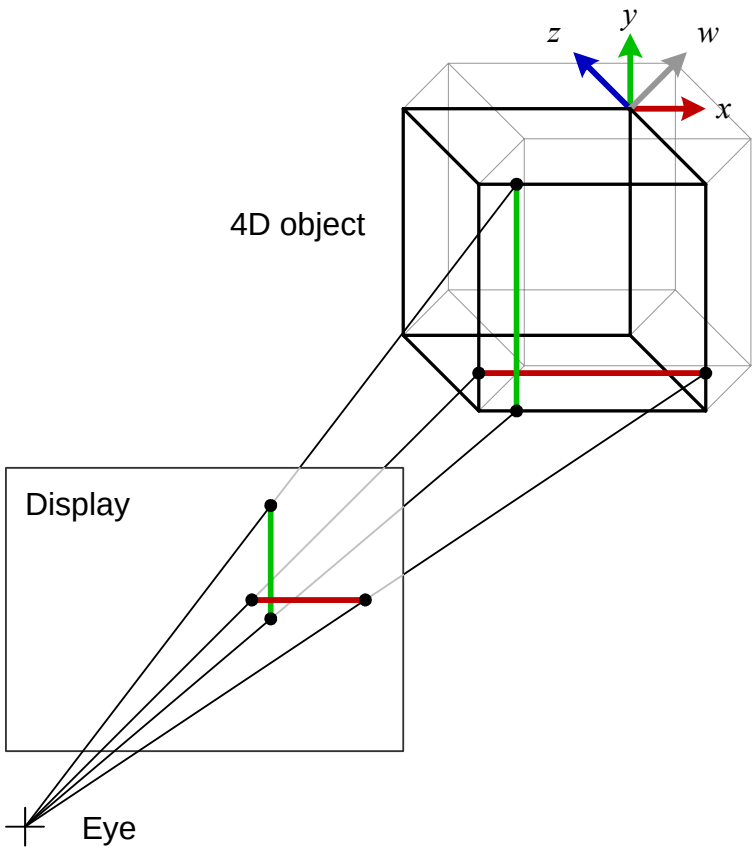

Figure 10: Ray projection of *x* and *y* using the discard-*z* transform

Figure 10 shows a ray projection from a 4D tesseract to a 2D display using the discard-*z* projection. The axes in 4-space have been spread out for clarity. A single face (*w*=-1) is highlighted using bold lines. Projected rays are shown as the *x* and *y* axes are traversed across the face. The tesseract shown has zero rotation, so its axes are the same as eye space. As we traverse the face in *x* and *y*, the projected ray traverses an area in *x* and *y* on the display. Each *x* and *y* value on the face corresponds to a unique *x* and *y* value on the display. This situation is similar to 3D ray projection.

Before continuing, it is important not to be lead into the 3D interpretation of this diagram. Although the face looks like a cube, it is flat and has zero hypervolume. It has zero thickness in the *w* direction. At any orientation, no part of the face can be in front of or behind any other part in a way that would hide one part from the viewer. The face is a boundary between the inside and outside of the object, and not an object itself. It is analogous to a square face of a cube in 3D, and not to the cube. Whereas a cube in 3D has 3D volume, this face has the equivalent (in 4D) of surface area, with three dimensions of extent.

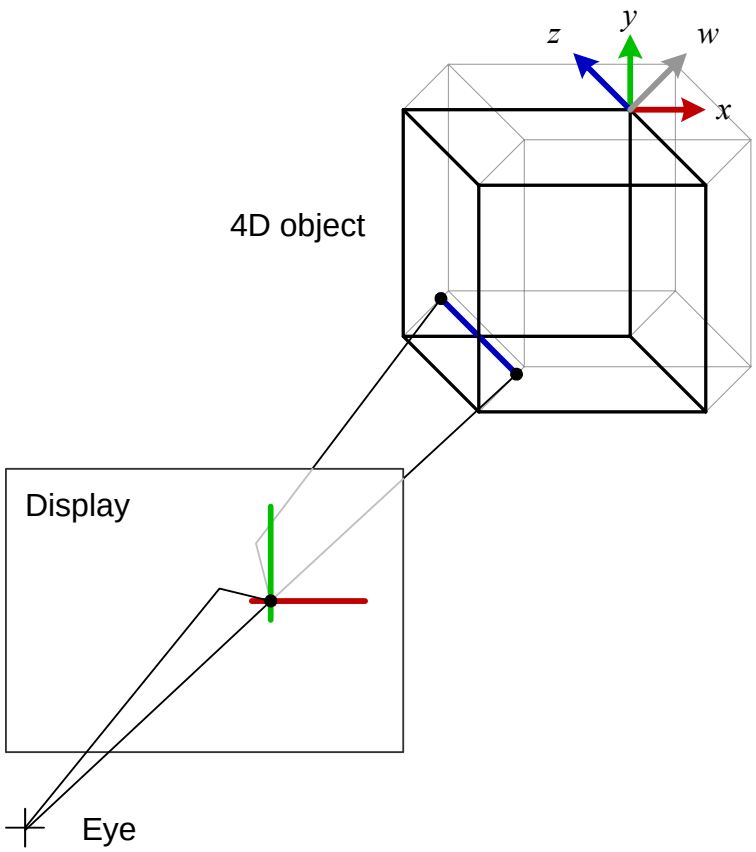

Figure 11: Ray projection of *z* using the discard-z transform

Figure 11 shows the effect of traversing in *z* across the face. Since *z* is discarded by the projection, traversing in *z* across the face has *no effect* on the position mapped to the display. Since no part of the *z* traversal is particularly special compared to any other, each part makes an equal contribution to the colour of that part of the display. Hence the contribution to the display will be the sum of colours present along the path in z across[10] the face.

Now consider the z=1 face. As we traverse z we cross this face, and find it has zero thickness in z. How does this face reflect light? Light arriving from z>1 reflects from all points of the form (x,y,1,w) which are on the face. However none of this light goes back to the viewer at the origin, because the face is of zero thickness when viewed from that direction. It is *edge on*, in the same way that the x=1 and y=1 faces are[11]. The difference is that, whilst x=1 and y=1 map to zero-thickness lines on the display, z=1 covers the entire size of the tesseract on the display. A consequence is that, as the tesseract rotates and one face approaches the z=1 orientation, it will because very large and disappear! This is because its aspect of zero thickness in z is hidden as the z axis is discarded by the projection. The reason it has zero thickness as we traverse it in z is because it is not visible from the viewpoint in the 4D world. Faces of this type will be referred to as *expanded faces*.

[10] Despite what the diagram might suggest, a path *through* the face is the wrong concept.
[11] Again we are ignoring perspective, which would 'tilt' the faces slightly and make them visible due to their extent in w. With perspective, the tesseract could be moved or rotated to restore the zero thickness effect for one face at a time.

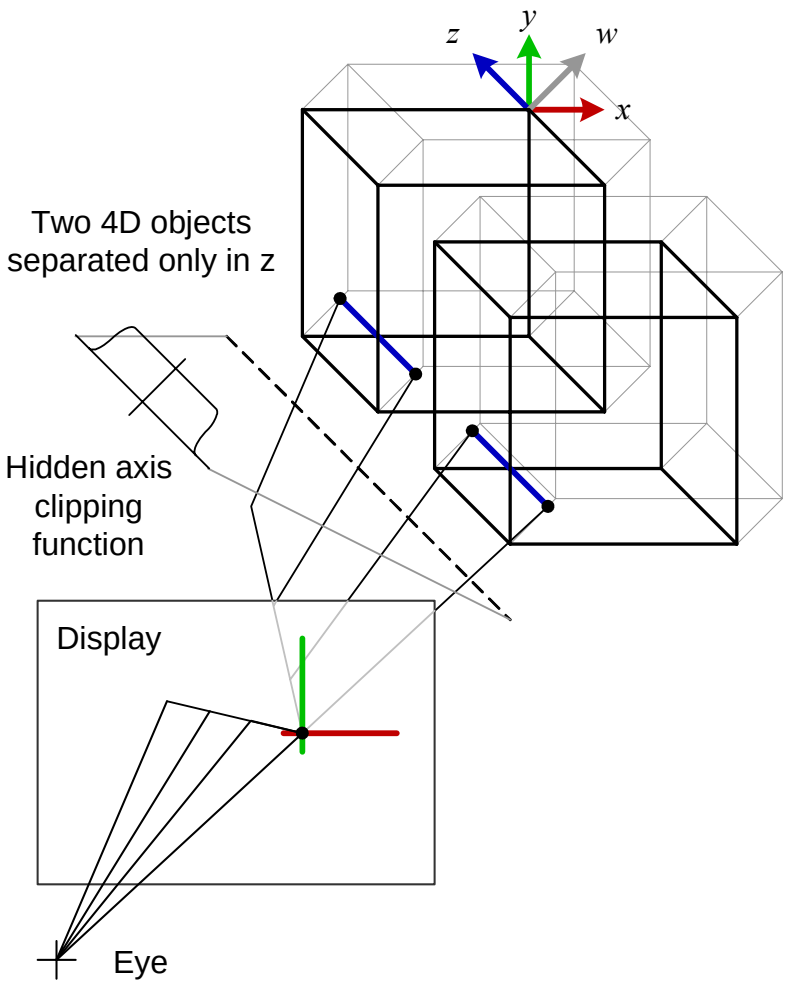

Figure 12: Projection of *z* from multiple objects

When multiple objects are introduced in Figure 12, we find that the colour at a particular display point potentially depends on the contribution of all of the objects that project to that point. The hidden axis clipping (HAC) function limits the *view* in the *z* – a job done by the limits of display size for *x* and *y*. The function modulates the contribution of each part of the *z* traversal. Perspective, depth culling and transparency also modulate the contribution, in the same way that they do for the *x* and *y* axes.

Now we can attempt to form the proper path integral along the *z* axis. It has the following properties:

- The coordinates *x*/*w* and *y*/*w* determine where a coordinate in eye space will map to the display. Regions in eye space for which *x*/*w* and *y*/*w* match can contribute to the same pixel.
- In the *w* direction, pixel values are combined using depth testing and alpha values, as they are in 3D. In the *z* direction, they combine equally unless clipped, where the contribution is attenuated by the HAC function.

It is as if we render a line to the display for the *z* traversal just like we did for the *x* and *y* traversals, modulate using the HAC function, and then squash that line into a single pixel.

Points to note:

- The shape of the HAC function can be altered in order to render a slice or wedge of the scene. If the HAC function becomes a narrow peak, we return to rendering a 3D slice of the 4D world.
- The HAC function can hide just part of an object. The gradual nature of the function prevents hard edges appearing on the display.
- There are corollaries with the retina of our 4D creature. To map the 3D retina of the 4D creature to our 2D retina, lines must be contracted into points. Since there is no criterion for selecting areas of the 3D retina (no part is 'in front' of another) the mean value is taken.

Implementation

We could render the scene much as described, by integrating the paths through the 3D textures relevant to a particular pixel, whilst applying the HAC function, and writing the result to the pixel. However that would be difficult, have performance issues, and besides we would like to draw polygons rather than use a ray tracing technique.

The *y* layered stack

One possible solution has been hinted at already. We rotate about the origin in eye coordinates in *yz* so that the *z* axis maps to the *y* axis and the *y* axis is discarded. We then render a thin[12] slice of the scene at a particular *y* value. This slice is the intersection of the slice hyperplane with the face hyperplane. Except for degenerate cases[13], this intersection will be a 2D plane. Therefore we can use conventional 2D polygons to render this intersection. When rendering 3D textures we can use a single value from the texture, instead of the sum of a path, because the slice is thin.

The HAC function is applied by blending a texture over this rendered image, to diminish the top and bottom edges smoothly towards black. Each vertical column of pixels is then summed and scaled to generate a value for the display pixel for the *y* value we initially chose and the *x* value of the pixel column. The drawback of this method is that it increases the computation requirement per frame by a factor of the display dimension, i.e. ≈1000. Figure 13 shows the pipeline.

[12] In this calculation we use a zero thickness slice to approximate the thin slice of the view.
[13] When the two hyperplanes are coplanar, the face is edge-on and not visible.

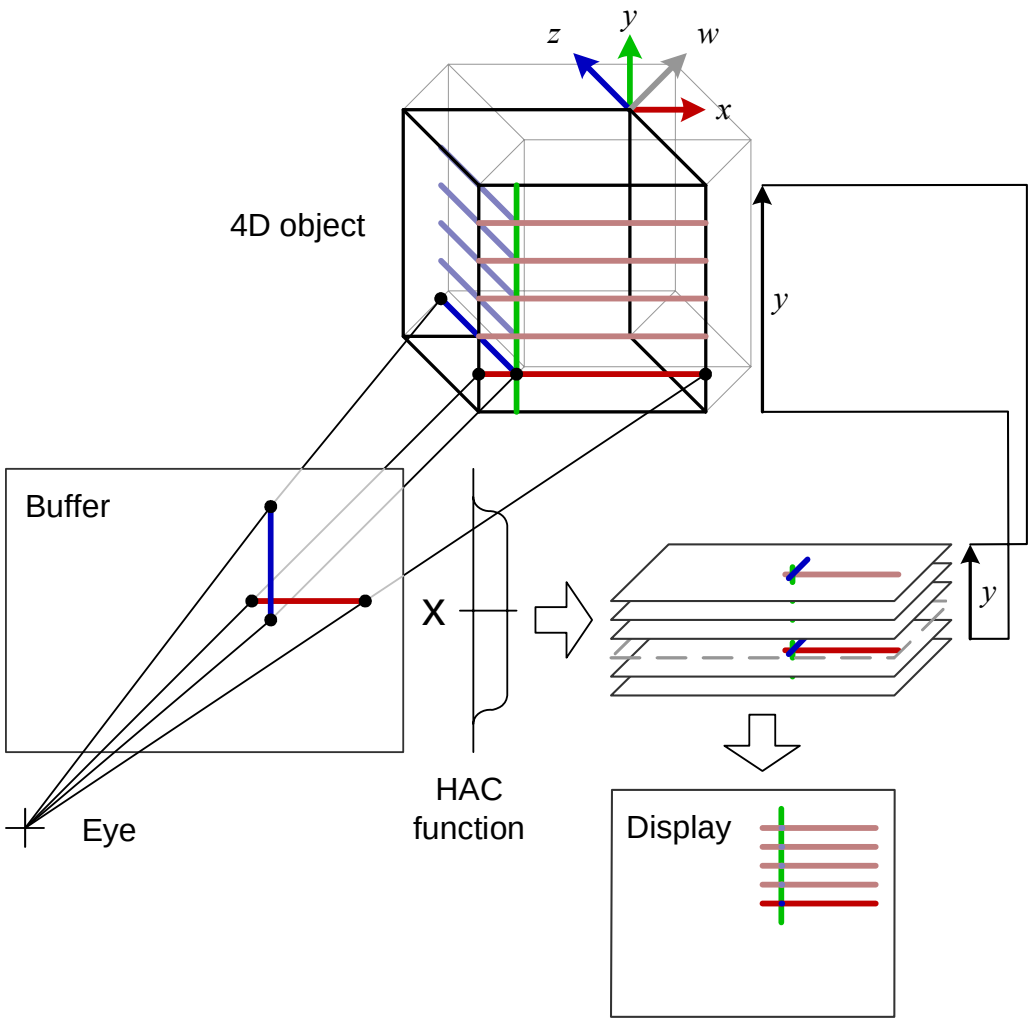

Figure 13: Rendering pipeline for *y* layered stack

What happens to the depth test and perspective effect of *w* in the above pipeline? The depth test is performed when rendering each slice to the buffer. Since depth depends only on *w*, our rotation in *yz* has not changed the relationship of different primitives. Perspective operates as expected, and is applied to the vertex coordinates before they are used. Transparency is also supported, by rendering the 2D polygons using blending in a similar way to 3D rendering. Once transparency is introduced, the render order problem must be resolved, either by depth sorting in *w*, or using an order-independent technique such as depth peeling.

Let us consider how each face orientation is rendered in this scheme. The faces $x=\pm1$, $y=\pm1$ and $z=\pm1$ are *edge on* to the display so are not drawn, so we tilt them slightly to make them visible. A small rotation in *xw*, *yw* or *zw* respectively is suitable. For $x=1$, the face extends in *y*, *z* and *w* and is flat in *x*. The face is sliced in *y* and for each slice a rectangle is rendered into the buffer for that slice, 2 units high (the size of the face in *z*) and $2\sin\theta$ wide (where θ is the angle of our tilt). This rectangle is that generated as the intersection of the face with the hyperplane $y=y_s$ which defines this particular slice. Other orientations of the tesseract will produce differently shaped sections through the face.

At each pixel in the rectangle, the fragment shader chooses a single value from its 3D texture, using the 3 texture coordinates that span the face, performs the depth test on *w* using the

buffer, and writes it to the buffer if it passes. After rendering the whole buffer for this slice, the HAC function is applied by rendering graded transparent rectangles to it, which smoothly darken the top and bottom of the screen towards black. Vertical columns in the buffer are summed, and each sum is written to a pixel on the display. The *x* coordinate on the screen is the *x* coordinate of the column, and the *y* coordinate is the *y* coordinate of the slice. The process then proceeds onto the next slice. The figures below show the results.

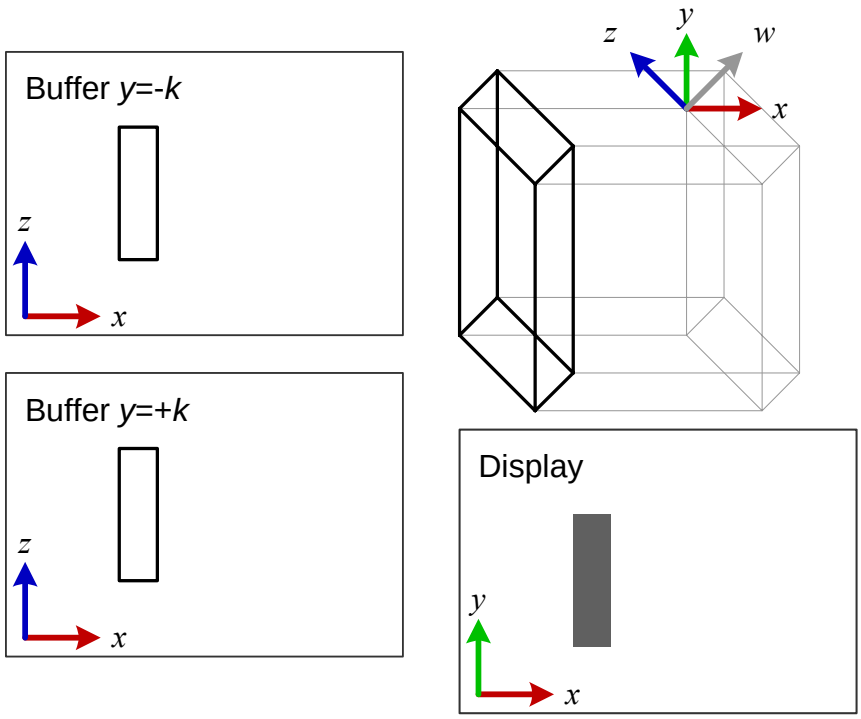

Figure 14: Result of rendering the *x*=-1 face

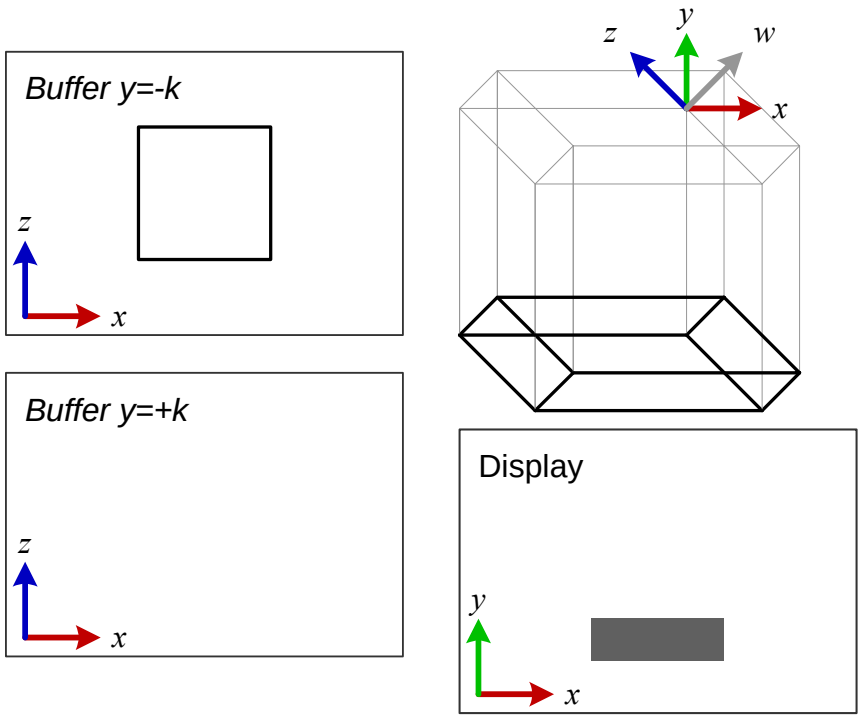

Figure 15: Result of rendering the *y*=-1 face

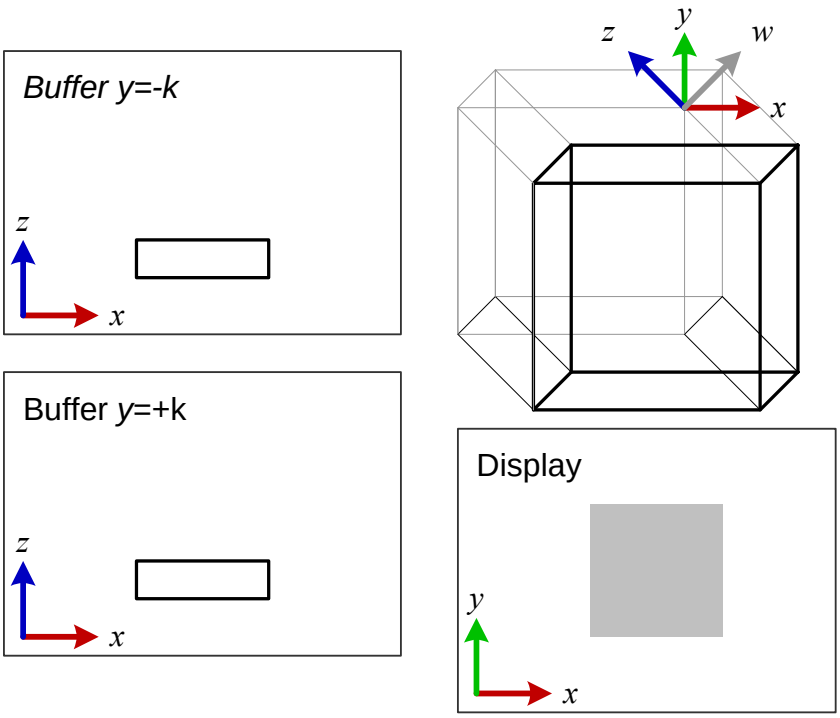

Figure 16: Result of rendering the *z*=-1 face

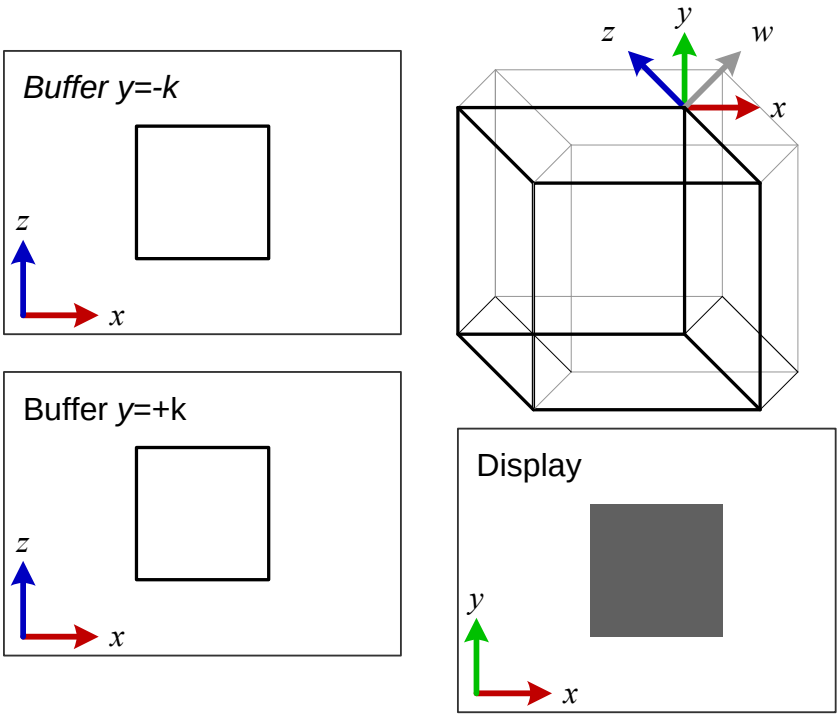

Figure 17: Result of rendering the w=-1 face

The *z* layered stack

It is equally possible to turn the situation round. Instead of rendering slices in *xz*, we render slices in *xy*, as shown in Figure 18.

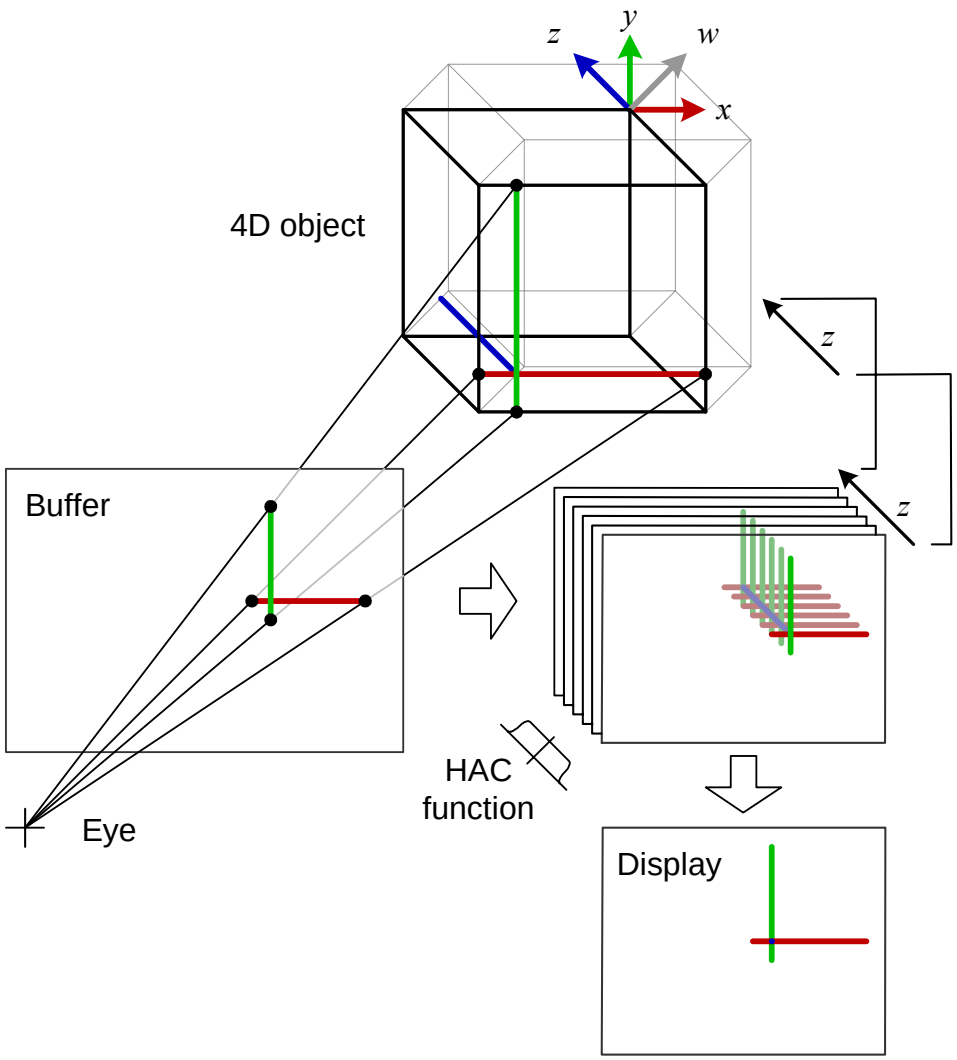

Figure 18: Rendering pipeline for *z* layered stack

Figure 18 shows a *z* layered stack, which is similar to the *y* layered stack of Figure 13. There are pros and cons to each. The *z* layered stack can take advantage of full screen anti aliasing (FSAA) capabilities built into the hardware, as it is rendering a slice that maps directly to the display each time. The *y* layered stack cannot do this because hardware FSAA, designed for 3D, cannot anti-alias between slices in the vertical. The HAC function is now a single value applied to each buffer. Its drawback is that, for the final recombination, each destination on the display pixel requires one source pixel from each layer. It is undesirable to store every buffer until recombination. A high-bit-depth accumulation buffer is ideal. If such a buffer is absent or inefficient, buffers can be rendered and combined tree-wise so that only a few are required in memory at any one time.

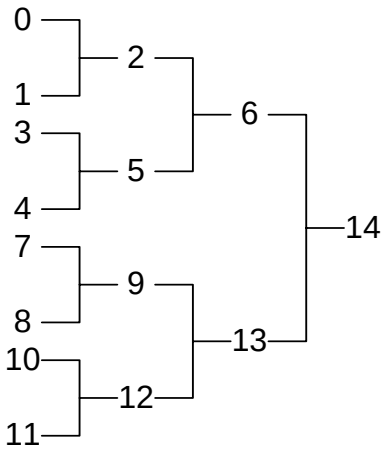

Figure 19: Order for rendering buffers for *z* layered stack

Figure 19 shows a possible rendering order. When two buffers on the left hand side of a junction are ready, the area combined into a buffer that is the average of the two, and the two original buffers are discarded. The maximum of buffers required at any one time is equal to the depth of the tree - four in this case.

Optimising stack rendering

In the stack implementations above, performance can be improved by reducing resolution along any axis, including the hidden axis. In the *y* layered stack (Figure 13) we can reduce the *z* dimension to reduce resolution in *z*. In the *z* layered stack (Figure 18) we can reduce the number of layers in the stack to achieve the same effect. For both of these optimisations, as we reduce resolution in *z*, the final image will progressively appear to be more a composite of slices than a smooth object.

For real-time rendering, further optimisation may be necessary. For this we turn to a different method: one that approximates the real image by rendering slices of each face directly as 2D polygons.

Face slicing

Face slicing approximates the output of the stack renderer by rendering polygons directly to the final display. Such a renderer could conceivably integrate paths through 3D textures to produce an accurate result, but this would be time consuming and place high demand on vertex and fragment shaders. Instead the face slicing technique renders a number of slices of the face as 2D polygons. Using variable transparency, these are combined to approximate the path integral through the 3D face.

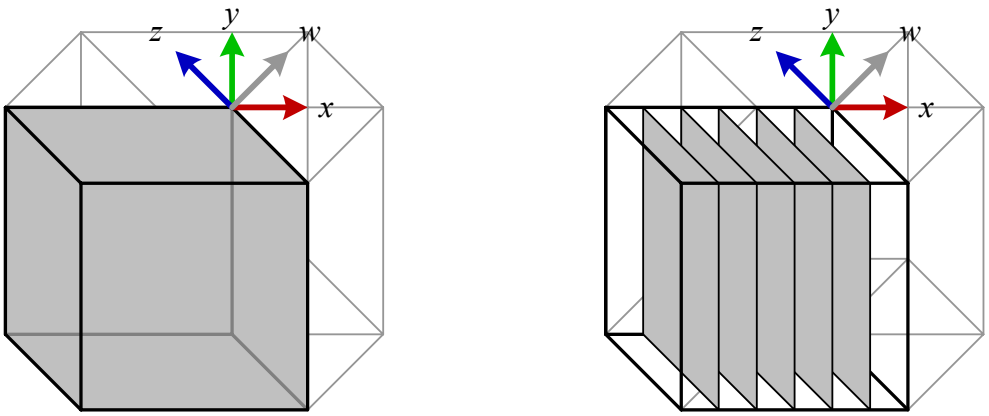

Figure 20: Slice rendering stages 1 and 2: Render solid outline, then slices along *x* axis

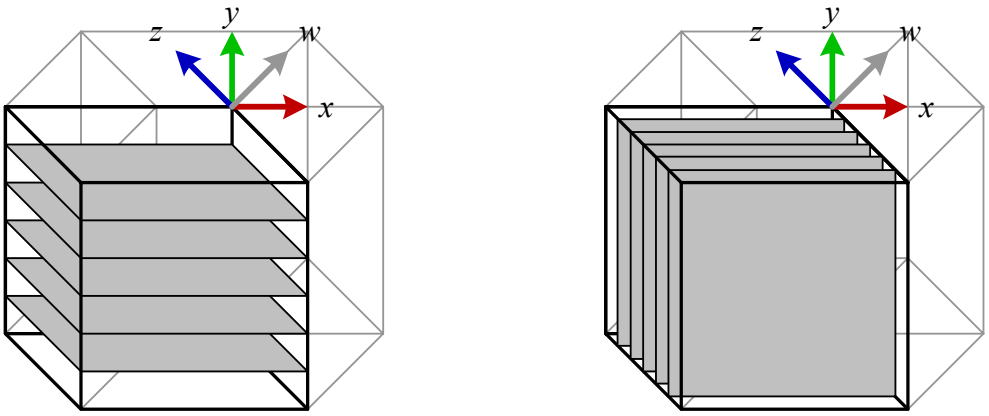

Figure 21: Slice rendering stages 3 and 4: Render slices along *y* and *z* axes

Figure 20 and Figure 21 shows the basic technique. Firstly the solid outline of the face is drawn to write the depth values and initialise the alpha values in the frame buffer. Then, three sets of slices are rendered - one for each extent direction of the face. All stages use the face texture(s), lighting and material properties. Each slice is translucent, so that previously drawn slices show through it. In this way, an approximation of the integrated paths through the 3D texture is built up.

Drawn as shown above using simple transparency, the faces would have jagged sawtooth patterns at their edges and a grid of visible slice edges elsewhere. We can mitigate this to some extent by appropriate choice of blending algorithm. OpenGL provides a number of options for combining source and destination colour and alpha values. However these options are not limitless and, as we shall see, there is no ideal combination for our problem. This area of the pipeline is fixed, and cannot be programmed as part of a vertex or fragment shader[14], so the following applies regardless of their availability.

Our aim is for the resultant display pixel to be the average value of the pixels written to it[15]. As can be seen, different parts of the face on the display are touched by differing numbers of slices. Consider two approaches to this problem:

[14]Programmable shaders cannot read back pixel colours and alpha values from the frame buffer whilst rendering to it.

[15] Again, the diagram can be misleading. No slice is in front of any other. Hidden axis clipping aside, each makes an equal contribution to the colour of the resultant display pixel.

- For each pixel, find out how many slices will touch it. Divide the alpha value by this number before rendering each slice to the pixel.
- Maintain a rolling average of the colour value using destination alpha in the frame buffer.

The first method requires two rendering passes, a programmable fragment shader, another buffer for the pixel count, and is vulnerable to precision problems so we will consider the second.

A rolling average can be maintained using the usual blending for transparent objects. In the following, the six quadrilaterals that make up the boundaries of the face will be known as facets, and suffices *i*, *g*, *s*, *d* and *f* stand for initial, global (i.e. for the entire face), source, destination and final.

Firstly, render a set of facets that cover the area of the face once only. These can be chosen using a method similar to the winding rule used to determine front and back facing faces in 3D. The vertex order is arranged so that, in the 3D subspace of the face hyperplane, it is clockwise for each facet when viewed from the centre of the face. From the eye viewpoint, either the set of facets with clockwise or anticlockwise order will cover the face. We choose one set and render it[16]. We configure the blend equations to use source alpha blending for colour (Equation 19) using derivations of the global alpha value for the face α_g as shown in Equation 20. Destination alpha α_f is written only in preparation for the slice rendering stage.

$$c_f = \alpha_s c_s + (1-\alpha_s)c_d$$

Equation 19: Colour blending equation for transparency using source alpha

$$\alpha_s = \frac{\alpha_g}{2-\alpha_g}$$
$$c_f = \alpha_s c_s^1 + (1-\alpha_s)c_i$$
$$\alpha_f = 1-\frac{\alpha_g}{3}$$

Equation 20: Blend equations for first solid outline facet set

Next, render the other set of facets that cover the area of the face. If initially we rendered the anticlockwise set, render the clockwise set now and *vice versa*. The blending configuration (Equation 20 and Equation 21) is contrived so that the frame buffer contains the values shown in Equation 22 after this stage.

$$c_f = \frac{\alpha_g}{2}c_s^2 + \left(1-\frac{\alpha_g}{2}\right)c_d$$
$$\alpha_f = 1-\frac{\alpha_g}{3}$$

Equation 21: Blend equations for second solid outline facet set

[16] This test must be done after the perspective transformation, but fortunately OpenGL's usual face culling capability can achieve this.

$$c_f = \left(1-\frac{\alpha_g}{2-\alpha_g}\right)\left(\frac{2-\alpha_g}{2}\right)c_i + \left(\frac{\alpha_g}{2-\alpha_g}\right)\left(1-\frac{\alpha_g}{2}\right)c_s^1 + \alpha_g\frac{c_s^2}{2} = (1-\alpha_g)c_i + \alpha_g\frac{c_s^1+c_s^2}{2}$$

$$\alpha_f = 1-\frac{\alpha_g}{3}$$

Equation 22: Resultant colour and alpha values after both facet outline sets are rendered

As a result, each pixel colour has been blended with the average of two values from the facet outline, and α_d is set to 1/3 over the area of the face. If the level of detail is low, we can finish here; if not, we render the slices.

It is important to set the depth buffer and destination alpha values for the face area in this solid outline pass. If slices are to be rendered, only the far set of solid outline facets should write to the depth buffer; otherwise the slices will fail the depth test. The slices themselves should not write to the depth buffer either. For accuracy, a final stage is required to write the depth buffer for the near set of solid outline facets. However, this will worsen depth test artefacts that arise when object separated only in *z* in 4-space intersect after projection, and should generally not be performed. Transparent faces should not write the depth buffer at all; another method of depth control is required (such as depth sorting). Destination alpha should be written by at least the near set of solid outline facets, so that the visible face area is covered.

From now on we blend using destination alpha. The value of α_d stored at a point in the frame buffer is the alpha value used for blending the *next* pixel rendered to the point.

$$c_f = (1-\alpha_d)c_s + \alpha_d c_d$$

Equation 23: Colour blending equation for transparency using destination alpha

The number and configuration of slices can be chosen as appropriate for the level of detail. Now that we have a set of slices that do not cover the area of the face equally, we must use the averaging algorithm. Using the blending in Equation 23 and ignoring α_g for the moment, we can in principle control the alpha value so that the pixel colour is always the mean of pixels written to it so far.

$$c_f = \frac{1}{n}c_s + \frac{(n-1)}{n}c_d,\ n = 1..k$$

Equation 24: Alpha values for rolling average

It can be seen that the final contribution from the first pixel is:

$$c_f^1 = \frac{1}{2}\cdot\frac{2}{3}\cdot\frac{3}{4}\cdot\ldots\cdot\frac{k-1}{k}c^1 = \frac{c^1}{k}$$

Equation 25: Contribution of c^1 to c_f

The result is similar for every other pixel, such that the final colour value is:

$$c_f = \frac{c^1 + c^2 + c^3 + \ldots + c^k}{k}$$

Equation 26: Final pixel colour

From Equation 26 we see that the final colour is the mean of all pixel colours as expected. However this assumes that we can obtain n in Equation 24 on a per-pixel basis for each rendering step and set the alpha value accordingly. In practice this is difficult. We can however approximate n by using the destination alpha value to 'count' the number of pixels rendered to each display pixel. Consider Equation 27, which is achievable using OpenGL[17].

$$c_f = (1-\alpha_d)c_s + \alpha_d c_d$$
$$\alpha_f = (1-\alpha_d)k + \alpha_d$$

Equation 27: Blend equations for approximate averaging

By choice of k we can control the speed at which α_d approaches 1. However we have only limited control over the shape of the curve, as shown in Figure 22.

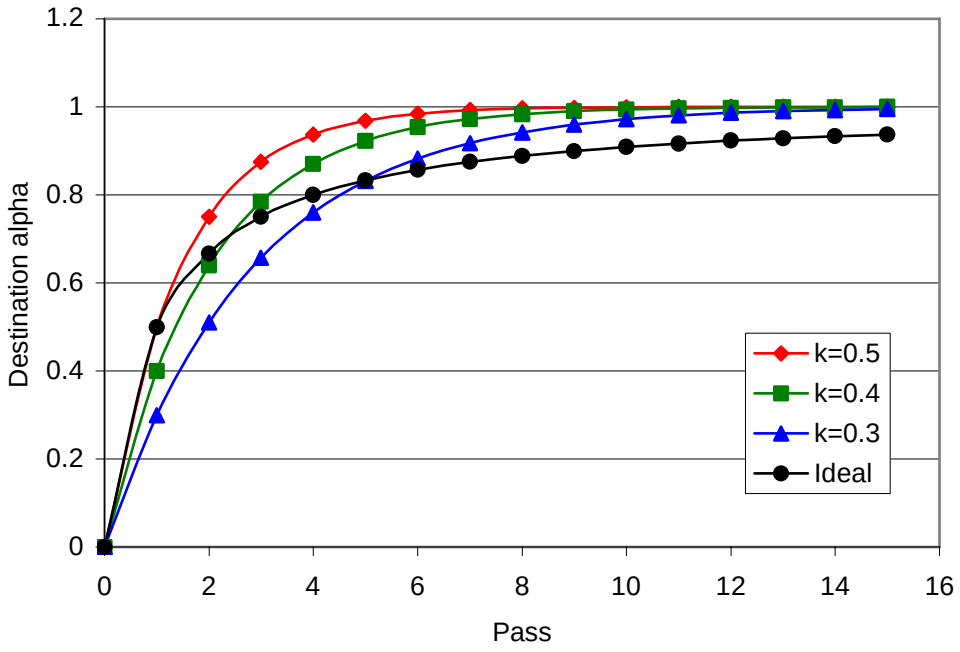

Figure 22: Approximating destination alpha values

Agreement is not particularly good, essentially because we are trying to emulate $1/n$ using $1/n^2$. However we can do a bit better since we know our first two passes have been completed and have already set that destination alpha to 1/3. As we are correct up to that point, we can adjust k to fit the part of the curve beyond that point.

[17] Provided that the blend_equation_separate extension is available. Value k is passed as the alpha value of the source colour.

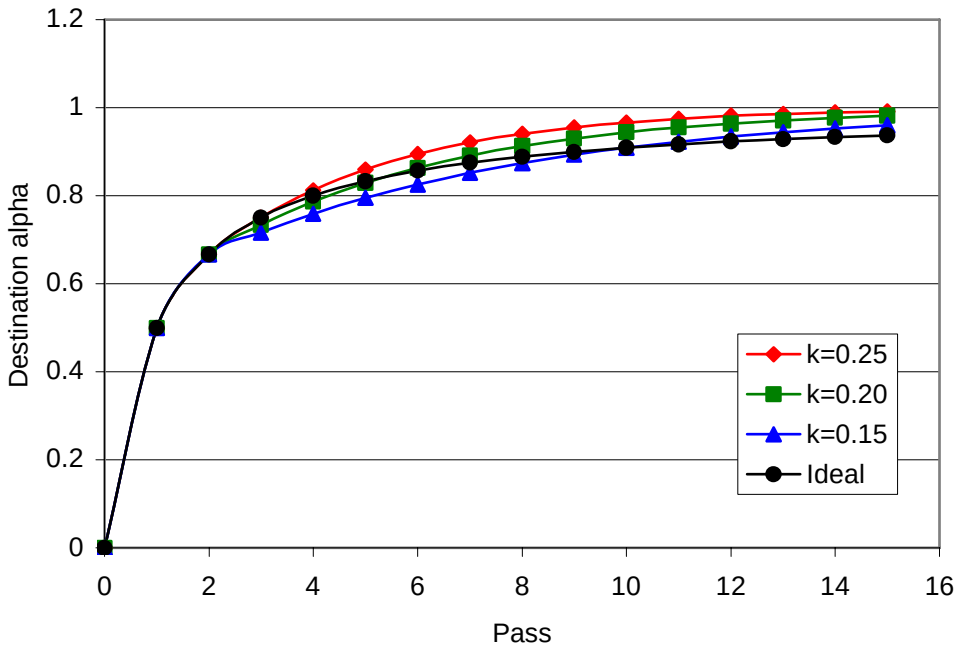

Figure 23: Approximating destination alpha values after pass 2

Figure 23 shows that we can approximate α_d to within a few percent in this way. This is not quite as good as it sounds; if the alpha value is consistently too high or low, errors will tend to accumulate in the frame buffer as subsequent slices are rendered. A full calculation of final colour error would need to take into account the discrete nature of the colour values, and is best achieved by simulation.

Points to note are:

- Each vertex of the face can have a different depth or w value. Depth testing is imperfect when faces that are mostly separated in z in eye coordinates intersect after projection. This is because we no longer have the 3D depth buffer implied by stack rendering. To mitigate the problem, the render order of objects should be consistent. To avoid it, another means of depth determination should be employed, such as depth sorting.
- Further optimisation could be applied to the k value to reflect that, as rendering progresses, slices are more likely to drawn over pixels that are further along the pass curve. As more slices are drawn, k would decrease.
- Since k is supplied as the source alpha, its value could be embedded in the 3D texture, and decrease towards the centre of the texture where more passes are expected.
- The global transparency for the face, α_g, does not act ideally in making the face transparent once slice rendering begins. Errors accumulate with the number of passes, and a typical total loss of transmitted colour might be 30% for many passes. The appearance is similar to a variable transparency across the face. Areas with more passes are less transparent, and the error increases as k is reduced. Some mitigation is possible, but the flaw is fundamental. Blending can be modified to correct the transmitted colour at the expense of the rendered colour if desired.
- The value of α_g can be specified per vertex.
- Other slice configurations are possible. For example, slices which join opposite edges and pass through the centre.

- Slice positions can be randomised or cycled through for each frame so that different parts of the 3D texture are drawn each time. This would build up an impression of the 3D morphology over time at the expense of some flicker.
- Per-pixel source alpha from the 3D texture cannot be used for transparency, as source alpha has been reused to store the k value (α_s does not appear in Equation 27).
- If the slice pattern is fixed, 3D textures may not be the best way to store the texture data. A number of higher resolution 2D textures, one for each slice, may give better results.

3D models as textures for 4D faces

Just as a 2D texture could be replaced by a number of 2D polygons, a 3D texture can be replaced by a number of 3D shapes. Each facet of the 3D model adopts the same 4D normal so is rendered with the same lighting. It is not appropriate to apply 2D textures to the 3D models, as the real representation is an integral through the 3D texture within the object. However the approximation may be acceptable, especially if the models are thin. Ideally the colour and alpha values of the object should be chosen so that the result approximates integration of paths through a 3D texture. The face outline can be rendered before the model if required.

In some ways a set of facet slices can be considered as a 3D model, and similar transparency techniques can be applied to the rendering of 4D faces using 3D models.

Point textures

Point textures are a simple method of adding texture to a face. The 'texture' consists of a number of 4D vertex coordinates that lie within the face. At render time we transform the vertex coordinates and render a small object at that position. The points could be rendered as (multi-coloured) points, billboarded textures, 3D models, etc.

Wireframe

Traditional wireframe rendering can be used to outline 4D faces. However hidden axis clipping and expanded face attenuation should still be applied, and it is difficult to attenuate expanded faces using only the wireframe.

Further aspects of rendering

Expanded face attenuation

Expanded faces are those that, because of their orientation, become thin in the direction of the hidden axis. In the 4D world these appear as thin because they are *edge-on* to the viewer. When the hidden axis is discarded their thinness is lost and they can appear large.

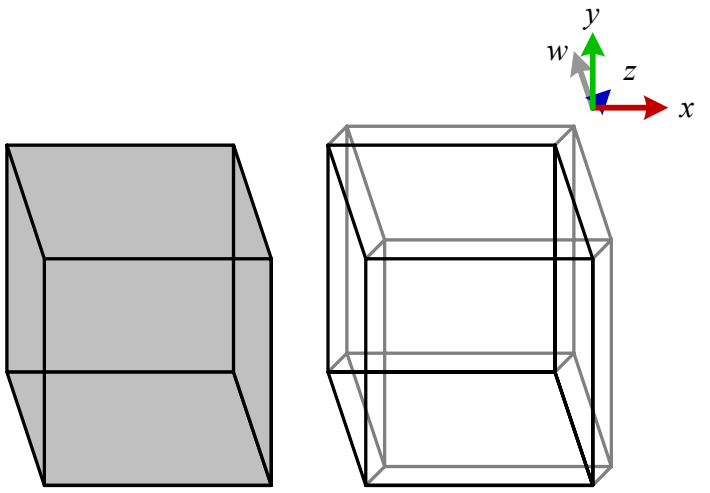

Figure 24: An expanded face, shown on the left removed from object

Figure 24 shows the face z=-1 when the z axis is almost aligned with the hidden axis. It is large and appears to fill the interior of the object. Expanded faces are confusing. For a tesseract the expanded face appears to fill the volume of the object, so it has no clear interior.

Stack rendering automatically attenuates expanded faces as the path integrals through the face diminish towards zero as the face normal aligns with the hidden axis. For other rendering methods we must artificially recreate that effect by, for example, making the face progressively transparent.

Lighting

Lighting techniques remain broadly the same as for 3D, but all of the calculations move into 4-space. The inverse square law (and it's quadratic attenuation) may be replaced with an inverse cube law (and cubic attenuation) if desired, depending on the physical model.

Unfortunately, since the vector calculations for lighting must be performed in 4D space, much of the OpenGL lighting pipeline cannot be reused. The problem is amply suited to programmable shaders however.

Complex objects

Joining simple objects together, and removing the faces at the interface, can create more complex composite objects. When starting with a complex object, complex faces can be handled by breaking them down into cubes, triangular prisms and tetrahedra. In the same way that the faces of 3D objects can always be decomposed into triangles, the faces of 4D objects can always be decomposed into tetrahedra.

Non-hyperplanar faces

In 3D rendering, artefacts appear when faces with 4 or more edges are distorted so that their vertices no longer lie in a plane. In 3D this can be solves by triangulating the faces. In 4D, the stack renderer solves this problem when it generates the intersection for each slice, which is necessarily planar. Slice renderers may need to triangulate facets and slices before drawing, in a similar way to 3D renderers.

Smooth faces and per-vertex normals

Normal vectors can be interpolated across the face before they are used in lighting calculations, etc. Since there are three degrees of freedom within the face, a number of normal vectors can take part in the interpolation, depending on the shape of the face and algorithm chosen.

Hidden face culling

For opaque objects, hidden faces can be discarded. These will be those for which, after final projection, the angle between the face normal and view angle is greater than 90°. In other words, the viewpoint and the normal vector are on different sides of the hyperplane of the face.

Conclusion

We have chosen two projection methods based on the extension of the human eye to a four dimensional creature in a 4D world. Since they are related by a rotation in eye coordinates, the simpler has been chosen for implementation.

A number of methods for the polygonal rendering of 4D environments have been delineated. These have the following characteristics:

Method	Characteristics	Typical Use
Stack rendering	• Accurate • 3D texturing • 3D depth buffer • Transparency • Hidden axis clipping • Global level of detail control • Expanded face attenuation • Slow	• Reference renderer • Still images • Non-real-time rendering
Slice rendering	• Approximate • 3D or 2D texturing • 2D non-ideal depth buffer • Approximate transparency • Hidden axis clipping • Per-object level of detail control • Emulated expanded face attenuation • Complex • Faster	• Real time renderer • Game engine
Face model rendering	• Approximate • Limited texturing • 2D non-ideal depth buffer • Approximate transparency • Hidden axis clipping • Per-object level of detail control • Emulated expanded face attenuation • Fast	• Real time renderer • Game engine

Point texture rendering	• Accurate • Very limited texturing • No depth buffer • Additive transparency • Hidden axis clipping • Per-object level of detail control • Emulated expanded face attenuation • Simple • Fast for limited point count	• Simple real time renderer • Demonstrations • Some elements of a game engine
Wireframe rendering	• Accurate • No texturing • No depth buffer • No transparency • Hidden axis clipping • No level of detail control • No expanded face attenuation • Simple • Fast	• Simple real time renderer • Web applets

Table 5: Rendering method characteristics

Appendix

Example pipeline

The following presents a projection and rendering pipeline for a game engine, using the discard *z* projection and slice rendering.

Stage 1: Gather data. After this stage, all object and view positions and orientations.
Stage 2: Geometric cull. Only objects that can intersect the view are selected for further processing. Only the positions of objects are known so far, so tests are based on object centres and bounding radii. Further culling may take place, depending on requirements.
Stage 3: Eye mesh generation. Using the model and view transformations, a mesh for each object is generated. This mesh will include:

- List of chunks in this object
- List of faces in each chunk
- List of facets in each face
- List of vertices in each facet
- Vertex coordinate values in eye space
- List of texture coordinates in each face
- Texture coordinate values
- List of normal vectors in each face
- Normal vector values in eye space
- Name of material for each face

When generating the eye mesh, two further cull types can be performed. Firstly, chunk culling, which performs the geometric cull test on each chunk. No further eye mesh is generated for chunks that fail. Back facing faces are also culled at this point.

Stage 4: Depth sort. If transparent objects are in use, they are sorted into depth order here, usually at the chunk level.
Stage 5: Lighting. If lighting cannot be calculated using programmable shaders, the work must be done here. Otherwise a set of lights must be prepared for the shaders to use.
Stage 6: Coordinate preparation. If using Equation 11 for projection, the z value of the eye coordinates is set to 1.
Stage 7: Slice selection. Chunks are divided into solid outline and slice facets. For chunks requiring high level of detail, slice facets are switched on.
Stage 8: Slice vertex coordinate generation. Coordinates are generated by interpolation between eye space coordinates already calculated. Colour values, resulting from the lighting calculation, can also be interpolated here.
Stage 9: Chunk despatch. Chunks are passed to OpenGL for rendering, using the glDrawArrays interface.

Version History

1. 2005-04-27: Created with filename project-and-render1-2005-04-27
2. 2005-05-20: First draft