

# **SELECTED PAPERS**

by

Eric Lengyel, Ph.D.

1999–2025





# Contents

Tweaking a Vertex's Projected Depth Value	5
Computing Tangent Space Basis Vectors for an Arbitrary Mesh	9
Oblique View Frustum Depth Projection and Clipping	13
Unified Distance Formulas for Halfspace Fog	23
Smooth Horizon Mapping	31
A Graphical Scripting Language for Interactive Virtual Environments	39
Motion Blur and the Velocity-Depth-Gradient Buffer	45
Transition Cells for Dynamic Multiresolution Marching Cubes	55
Fog with a Linear Density Function	75
GPU-Centered Font Rendering Directly from Glyph Outlines	87
Space-Antispace Transform Correspondence in Projective Geometric Algebra	101
Relativistic Quaternions	109
The Transwedge Product	115



# Tweaking a Vertex's Projected Depth Value

December 1999.

Published in *Game Programming Gems*, 2000.

## 1 Introduction

Many games need to render special effects such as scorch marks on a wall or foot prints on the ground that are not an original part of a scene, but are created during gameplay. These types of decorative additions are usually decaled onto an existing surface and thus consist of polygons which are coplanar with other polygons in a scene. The problem is that pixels rendered as part of one polygon rarely have the exactly the same depth value as pixels rendered as part of a coplanar polygon. The result is an undesired pattern in which parts of the original surface show through the decaled polygons.

The goal is to find a way to offset a polygon's depth in a scene without changing its projected screen coordinates or altering its texture mapping perspective. Most 3D graphics libraries contain some kind of polygon offset function to help achieve this goal. However, these solutions generally lack fine control and usually incur a per-vertex performance cost. This paper presents an alternative method which modifies the projection matrix to achieve the depth offset effect.

## 2 Examining the Projection Matrix

Let us first examine the effect of the standard OpenGL perspective projection matrix on an eye space point  $\mathbf{P} = (P_x, P_y, P_z, 1)$ . To simplify our matrix, we assume that the view frustum is centered about the z-axis in eye space (i.e., the rectangle on the near clipping plane carved out by the four side planes has the property that *left* = *-right* and *bottom* = *-top*). Calling the distance to the near clipping plane  $n$  and the distance to the far clipping plane  $f$ , we have

$$\begin{bmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & -\frac{2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix} \cdot \begin{bmatrix} P_x \\ P_y \\ P_z \\ 1 \end{bmatrix} = \begin{bmatrix} nP_x \\ nP_y \\ -\frac{f+n}{f-n}P_z - \frac{2fn}{f-n} \\ -P_z \end{bmatrix}. \quad (1)$$

To finish the projection, we need to divide this result by its  $w$ -coordinate, which has the value  $-P_z$ . This division gives us the following projected 3D point, which we will call  $\mathbf{P}'$ .

$$\mathbf{P}' = \begin{bmatrix} -\frac{nP_x}{P_z} \\ -\frac{nP_y}{P_z} \\ \frac{f+n}{f-n} + \frac{2fn}{P_z(f-n)} \\ 1 \end{bmatrix} \quad (2)$$

Recall that the near clipping plane lies at  $z = -n$  and the far clipping plane lies at  $z = -f$  since the camera points in the negative  $z$  direction. Thus, plugging  $-n$  and  $-f$  into Equation (2) for  $P_z$  gives us the expected  $z$  values of  $-1$  and  $1$  bounding the normalized clipping volume. Also recall that this mapping from  $(-n, -f)$  to  $(-1, 1)$  is a function of inverse  $z$ . This is necessary so that linear interpolation by the 3D hardware of values in the depth buffer remain perspective correct.

### 3 Tweaking the Depth Value

It is clear from Equation (2) that preserving the value of  $-P_z$  for the  $w$ -coordinate will guarantee the preservation of the projected  $x$ - and  $y$ -coordinates as well. From this point forward, we shall only concern ourselves with the lower-right  $2 \times 2$  portion of the projection matrix, since this is the only part which affects the  $z$ - and  $w$ -coordinates. The projected  $z$ -coordinate may be altered without disturbing the  $w$ -coordinate by introducing a factor of  $1 + \varepsilon$ , for some small  $\varepsilon$ , as follows.

$$\begin{bmatrix} -(1 + \varepsilon)\frac{f+n}{f-n} & -\frac{2fn}{f-n} \\ -1 & 0 \end{bmatrix} \cdot \begin{bmatrix} P_z \\ 1 \end{bmatrix} = \begin{bmatrix} -(1 + \varepsilon)\frac{f+n}{f-n}P_z - \frac{2fn}{f-n} \\ -P_z \end{bmatrix} \quad (3)$$

After division by  $w$ , we arrive at the following value for the projected  $z$ -coordinate.

$$\begin{aligned} P'_z &= (1 + \varepsilon)\frac{f+n}{f-n} + \frac{2fn}{P_z(f-n)} \\ &= \frac{f+n}{f-n} + \frac{2fn}{P_z(f-n)} + \varepsilon\frac{f+n}{f-n} \end{aligned} \quad (4)$$

Comparing this to the  $z$ -coordinate in Equation (2), we see that we have found a way to offset projected depth values by a constant  $\varepsilon(f+n)/(f-n)$ .

### 4 Choosing an Appropriate Epsilon

Due to the nonlinear nature of the  $z$ -buffer, the constant offset given in Equation (4) corresponds to a larger eye space difference far from the camera than it does near the camera. While this constant offset may work well for some applications, there is no single solution that works for every application at all depths. The best we can do is choose an appropriate  $\varepsilon$  given an eye space offset  $\delta$  and a depth value  $P_z$  which collectively represents the object that we are offsetting. To determine a formula for  $\varepsilon$ , let us examine the result of applying the standard projection matrix from Equation (1) to a point whose  $z$ -coordinate has been offset by some small  $\delta$ .

$$\begin{bmatrix} -\frac{f+n}{f-n} & -\frac{2fn}{f-n} \\ -1 & 0 \end{bmatrix} \cdot \begin{bmatrix} P_z + \delta \\ 1 \end{bmatrix} = \begin{bmatrix} -\frac{f+n}{f-n}(P_z + \delta) - \frac{2fn}{f-n} \\ -(P_z + \delta) \end{bmatrix} \quad (5)$$

Dividing by  $w$ , we have the following value for the projected  $z$ -coordinate.

$$\begin{aligned} P'_z &= \frac{f+n}{f-n} + \frac{2fn}{(P_z + \delta)(f-n)} \\ &= \frac{f+n}{f-n} + \frac{2fn}{P_z(f-n)} + \frac{2fn}{f-n} \left( \frac{1}{P_z + \delta} - \frac{1}{P_z} \right) \end{aligned} \quad (6)$$

Equating this result to Equation (4) and simplifying a bit, we end up with

$$\varepsilon = -\frac{2fn}{f+n} \left( \frac{\delta}{P_z(P_z + \delta)} \right). \quad (7)$$

A good value of  $\delta$  for a particular application can be found with a little experimentation. In should be kept in mind that  $\delta$  is an eye space offset, and thus becomes less effective as  $P_z$  gets larger. For an  $m$ -bit integer depth buffer, we want to make sure that

$$|\varepsilon| \geq \frac{1}{2^m - 1} \left( \frac{f-n}{f+n} \right) \quad (8)$$

since smaller values of  $\varepsilon$  will not yield an offset significant enough to alter the integer depth value. Substituting the right side of Equation (7) for  $\varepsilon$  and solving for  $\delta$  gives us

$$\delta \geq \frac{kP_z^2}{1 - kP_z} \quad (9)$$

or

$$\delta \leq \frac{-kP_z^2}{1 + kP_z} \quad (10)$$

where the constant  $k$  is given by

$$k = \frac{f-n}{2fn(2^m - 1)}. \quad (11)$$

Equation (9) gives us the minimum effective value for  $\delta$  when offsetting a polygon towards the camera (the usual case) and Equation (10) gives us the maximum effective value for  $\delta$  when offsetting a polygon away from the camera.

## 5 Implementation

The sample code below demonstrates how the projection matrix shown in Equation (3) may be implemented under OpenGL. The function `LoadOffsetMatrix` takes the same six values which are passed to the OpenGL function `glFrustum`. It also takes values for  $\delta$  and  $P_z$  which are used to calculate  $\varepsilon$ .

```

#include <gl.h>

void LoadOffsetMatrix(GLdouble l, GLdouble r, GLdouble b, GLdouble t,
    GLdouble n, GLdouble f, GLfloat delta, GLfloat pz)
{
    GLfloat    matrix[16];

    // Set up standard perspective projection
    glMatrixMode(GL_PROJECTION);
    glFrustum(l, r, b, t, n, f);

    // Retrieve the projection matrix
    glGetFloatv(GL_PROJECTION_MATRIX, matrix);

    // Calculate epsilon with Equation (7)
    GLfloat epsilon = -2.0F * f * n * delta / ((f + n) * pz * (pz + delta));

    // Modify entry (3,3) of the projection matrix
    matrix[10] *= 1.0F + epsilon;

    // Send the projection matrix back to OpenGL
    glLoadMatrix(matrix);
}

```

# Computing Tangent Space Basis Vectors for an Arbitrary Mesh

March 2001.

Published in *Mathematics for 3D Game Programming & Computer Graphics*, 2001.

**Abstract.** Modern bump mapping (also known as normal mapping) requires that tangent plane basis vectors be calculated for each vertex in a mesh. This article presents the theory behind the computation of per-vertex tangent spaces for an arbitrary triangle mesh and provides source code that implements the proper mathematics.

## 1 Mathematical Derivation

We want our tangent space to be aligned such that the  $x$  axis corresponds to the  $\mathbf{u}$  direction in the bump map and the  $y$  axis corresponds to the  $\mathbf{v}$  direction in the bump map. That is, if  $\mathbf{Q}$  represents a point inside the triangle, we would like to be able to write

$$\mathbf{Q} - \mathbf{P}_0 = (u - u_0)\mathbf{T} + (v - v_0)\mathbf{B}$$

where  $\mathbf{P}_0$  is the position of one of the vertices of the triangle, and  $(u_0, v_0)$  are the texture coordinates at that vertex. The vectors  $\mathbf{T}$  and  $\mathbf{B}$  are the *tangent* and *bitangent* vectors aligned to the texture map, and these are what we'd like to calculate.

Suppose that we have a triangle whose vertex positions are given by the points  $\mathbf{P}_0$ ,  $\mathbf{P}_1$ , and  $\mathbf{P}_2$  and whose corresponding texture coordinates are given by  $(u_0, v_0)$ ,  $(u_1, v_1)$ , and  $(u_2, v_2)$ . Our calculations can be made much simpler by working relative to the vertex  $\mathbf{P}_0$ , so we let

$$\mathbf{Q}_1 = \mathbf{P}_1 - \mathbf{P}_0$$

$$\mathbf{Q}_2 = \mathbf{P}_2 - \mathbf{P}_0$$

and

$$(s_1, t_1) = (u_1 - u_0, v_1 - v_0)$$

$$(s_2, t_2) = (u_2 - u_0, v_2 - v_0).$$

We need to solve the following equations for  $\mathbf{T}$  and  $\mathbf{B}$ .

$$\mathbf{Q}_1 = s_1\mathbf{T} + t_1\mathbf{B}$$

$$\mathbf{Q}_2 = s_2\mathbf{T} + t_2\mathbf{B}$$

This is a linear system with six unknowns (three for each  $\mathbf{T}$  and  $\mathbf{B}$ ) and six equations (the  $x$ ,  $y$ , and  $z$  components of the two vector equations). We can write this in matrix form as follows.

$$\begin{bmatrix} (\mathbf{Q}_1)_x & (\mathbf{Q}_1)_y & (\mathbf{Q}_1)_z \\ (\mathbf{Q}_2)_x & (\mathbf{Q}_2)_y & (\mathbf{Q}_2)_z \end{bmatrix} = \begin{bmatrix} s_1 & t_1 \\ s_2 & t_2 \end{bmatrix} \begin{bmatrix} T_x & T_y & T_z \\ B_x & B_y & B_z \end{bmatrix}$$

Multiplying both sides by the inverse of the  $(s, t)$  matrix, we have

$$\begin{bmatrix} T_x & T_y & T_z \\ B_x & B_y & B_z \end{bmatrix} = \frac{1}{s_1 t_2 - s_2 t_1} \begin{bmatrix} t_2 & -t_1 \\ -s_2 & s_1 \end{bmatrix} \begin{bmatrix} (\mathbf{Q}_1)_x & (\mathbf{Q}_1)_y & (\mathbf{Q}_1)_z \\ (\mathbf{Q}_2)_x & (\mathbf{Q}_2)_y & (\mathbf{Q}_2)_z \end{bmatrix}.$$

This gives us the (unnormalized)  $\mathbf{T}$  and  $\mathbf{B}$  vectors for the triangle whose vertices are  $\mathbf{P}_0$ ,  $\mathbf{P}_1$ , and  $\mathbf{P}_2$ . To find the tangent vectors for a single vertex, we average the tangents for all triangles sharing that vertex in a manner similar to the way in which vertex normals are commonly calculated. In the case that neighboring triangles have discontinuous texture mapping, vertices along the border are generally already duplicated since they have different mapping coordinates anyway. We do not average tangents from such triangles because the result would not accurately represent the orientation of the bump map for either triangle.

Once we have the normal vector  $\mathbf{N}$  and the tangent vectors  $\mathbf{T}$  and  $\mathbf{B}$  for a vertex, we can transform from tangent space into object space using the matrix

$$\begin{bmatrix} T_x & B_x & N_x \\ T_y & B_y & N_y \\ T_z & B_z & N_z \end{bmatrix}.$$

To transform in the opposite direction (from object space to tangent space—what we want to do to the light direction), we can simply use the inverse of this matrix. It is not necessarily true that the tangent vectors are perpendicular to each other or to the normal vector, so the inverse of this matrix is not generally equal to its transpose. It is safe to assume, however, that the three vectors will at least be close to orthogonal, so using the Gram-Schmidt algorithm to orthogonalize them should not cause any unacceptable distortions. Using this process, new (still unnormalized) tangent vectors  $\mathbf{T}'$  and  $\mathbf{B}'$  are given by

$$\begin{aligned} \mathbf{T}' &= \mathbf{T} - (\mathbf{N} \cdot \mathbf{T})\mathbf{N} \\ \mathbf{B}' &= \mathbf{B} - (\mathbf{N} \cdot \mathbf{B})\mathbf{N} - (\mathbf{T}' \cdot \mathbf{B})\mathbf{T}'/\mathbf{T}'^2. \end{aligned}$$

Normalizing these vectors and storing them as the tangent and bitangent for a vertex lets us use the matrix

$$\begin{bmatrix} T'_x & T'_y & T'_z \\ B'_x & B'_y & B'_z \\ N'_x & N'_y & N'_z \end{bmatrix} \quad (*)$$

to transform the direction to light from object space into tangent space. Taking the dot product of the transformed light direction with a sample from the bump map then produces the correct Lambertian diffuse lighting value.

It is not necessary to store an extra array containing the per-vertex bitangent since the cross product  $\mathbf{N} \times \mathbf{T}'$  can be used to obtain  $m\mathbf{B}'$  where  $m = \pm 1$  represents the handedness of the tangent space. The handedness value must be stored per-vertex since the bitangent  $\mathbf{B}'$  obtained from  $\mathbf{N} \times \mathbf{T}'$



may point in the wrong direction. The value of  $m$  is equal to the determinant of the matrix in Equation (\*). You might find it convenient to store the per-vertex tangent vector  $\mathbf{T}'$  as a four-dimensional entity whose  $w$  coordinate holds the value of  $m$ . Then the bitangent  $\mathbf{B}'$  can be computed using the formula

$$\mathbf{B}' = T'_w(\mathbf{N} \times \mathbf{T}'),$$

where the cross product ignores the  $w$ -coordinate. This works nicely for vertex shaders by avoiding the need to specify an additional array containing the per-vertex  $m$  values.

## 2 Bitangent versus Binormal

The term *binormal* is commonly used as the name of the second tangent direction (that is perpendicular to the surface normal and  $u$ -aligned tangent direction). This is a misnomer. The term binormal pops up in the study of **curves** and completes what is known as a Frenet frame about a particular point on a curve. Curves have a single tangent direction and two orthogonal normal directions, hence the terms normal and binormal. When discussing a coordinate frame at a point on a **surface**, there is one normal direction and two tangent directions, which should be called the tangent and *bitangent*.

## 3 Source Code

The code below generates a four-component tangent  $\mathbf{T}$  in which the handedness of the local coordinate system is stored as  $\pm 1$  in the  $w$ -coordinate. The bitangent vector  $\mathbf{B}$  is then given by  $\mathbf{B} = T_w(\mathbf{N} \times \mathbf{T})$ .

```
#include "TSVector4D.h"

struct Triangle
{
    unsigned short  index[3];
};

void CalculateTangentArray(long vertexCount, const Point3D *vertex,
                          const Vector3D *normal, const Point2D *texcoord, long triangleCount,
                          const Triangle *triangle, Vector4D *tangent)
{
    Vector3D *tan1 = new Vector3D[vertexCount * 2];
    Vector3D *tan2 = tan1 + vertexCount;
    ZeroMemory(tan1, vertexCount * sizeof(Vector3D) * 2);

    for (long a = 0; a < triangleCount; a++)
    {
        long i1 = triangle->index[0];
        long i2 = triangle->index[1];
        long i3 = triangle->index[2];

        const Point3D& v1 = vertex[i1];
        const Point3D& v2 = vertex[i2];
        const Point3D& v3 = vertex[i3];
```

```

const Point2D& w1 = texcoord[i1];
const Point2D& w2 = texcoord[i2];
const Point2D& w3 = texcoord[i3];

float x1 = v2.x - v1.x;
float x2 = v3.x - v1.x;
float y1 = v2.y - v1.y;
float y2 = v3.y - v1.y;
float z1 = v2.z - v1.z;
float z2 = v3.z - v1.z;

float s1 = w2.x - w1.x;
float s2 = w3.x - w1.x;
float t1 = w2.y - w1.y;
float t2 = w3.y - w1.y;

float r = 1.0F / (s1 * t2 - s2 * t1);
Vector3D sdir((t2 * x1 - t1 * x2) * r, (t2 * y1 - t1 * y2) * r,
              (t2 * z1 - t1 * z2) * r);
Vector3D tdir((s1 * x2 - s2 * x1) * r, (s1 * y2 - s2 * y1) * r,
              (s1 * z2 - s2 * z1) * r);

tan1[i1] += sdir;
tan1[i2] += sdir;
tan1[i3] += sdir;

tan2[i1] += tdir;
tan2[i2] += tdir;
tan2[i3] += tdir;

triangle++;
}

for (long a = 0; a < vertexCount; a++)
{
    const Vector3D& n = normal[a];
    const Vector3D& t = tan1[a];

    // Gram-Schmidt orthogonalize
    tangent[a] = (t - n * Dot(n, t)).Normalize();

    // Calculate handedness
    tangent[a].w = (Dot(Cross(n, t), tan2[a]) < 0.0F) ? -1.0F : 1.0F;
}

delete[] tan1;
}

```

# Oblique View Frustum Depth Projection and Clipping

May 2004.

Published in *Journal of Game Development*, Vol. 1, No. 2, 2005.

**Abstract.** Several 3D rendering techniques have been developed in which part of the final image is the result of rendering from a virtual camera whose position in the scene differs from that of the primary camera. In these situations, there is usually a planar surface, such as the reflecting plane of a mirror, that can be considered the physical boundary of the recursively rendered image. In order to avoid artifacts that can arise when rendered geometry penetrates the boundary plane from the perspective of the virtual camera, an additional clipping plane must be added to the standard six-sided view frustum. However, many 3D graphics processors cannot support an extra clipping plane natively, or require that vertex and fragment shaders be augmented to explicitly perform the additional clipping operation.

This paper discusses a technique that modifies the projection matrix in such a way that the conventional near plane of the view frustum is repositioned to serve as the generally oblique boundary clipping plane. Doing so avoids the performance penalty and burden of developing multiple shaders associated with user-defined clipping planes by keeping the total number of clipping planes at six. The near plane is moved without affecting the four side planes, but the conventional far plane is inescapably destroyed. We analyze the effect on the far plane as well as the related impact on depth buffer precision and present a method for constructing the optimal oblique view frustum.

## 1 Introduction

Several techniques have been developed to render 3D images containing elements that are inherently recursive in nature. Some examples are mirrors that reflect their immediate surroundings, portals through which a remote region of the scene can be viewed, and water surfaces through which refractive transparency is applied. Each of these situations requires that part of the scene be rendered from the perspective of some virtual camera whose position and orientation are calculated using certain rules that take into account the position of the primary camera through which the user is looking. For example, the image visible in a mirror is rendered using a camera that is the reflection of the primary camera through the plane of the mirror.

Once such a component of an image is rendered through a virtual camera, it is usually treated as a geometrically planar object when rendering from the perspective of the primary camera. The plane chosen to represent the image is simply the plane that naturally separates the image from the rest of the environment, such as the plane of a mirror, portal, or water surface. In the process of rendering from a virtual camera, it is possible that geometry lies closer to the camera than the plane

representing the surface of the mirror, portal, etc. If such geometry is rendered, it can lead to unwanted artifacts in the final image.

The simplest solution to this problem is to enable a user-defined clipping plane to truncate all geometry at the surface. Unfortunately, older GPUs do not support user-defined clipping planes in hardware and must resort to a software-based vertex processing path when they are enabled. Later GPUs do support generalized user-defined clipping operations, but using them requires that the vertex or fragment programs be modified—a task that may not be convenient since it necessitates two versions of each program be kept around to render a particular geometry.

This article presents an alternative solution that exploits the clipping planes that already exist for every rendered scene. Normally, every geometric primitive is clipped to the six sides of the view frustum: four side planes, a near plane, and a far plane. Adding a seventh clipping plane that represents the surface through which we are looking almost always results in a redundancy with the near plane, since we are now clipping against a plane that slices through the view frustum further away from the camera. Thus, our strategy is to modify the projection matrix in such a way that the conventional near plane is repositioned to coincide with the additional clipping plane, which is generally oblique to the ordinary view frustum. Since we are still clipping only against six planes, such a modification gives us our desired result at absolutely no performance cost. Furthermore, this technique can be applied to any projection matrix, including the conventional perspective and orthographic projections as well as the infinite projection matrix used by stencil shadow volume algorithms.

Moving the near plane does not affect the four side planes of the view frustum, but such an action generally has an unintuitive and detrimental effect on the conventional far plane which is not addressed by an earlier ad hoc implementation of the technique described herein [1]. The associated effect on the mapping of points to depth values is equally detrimental. Remedies to this undesirable consequence are limited, but we analyze the problem in this paper and provide a method for constructing the optimal oblique view frustum for any given near plane.

## 2 Background

The technique described in this paper is presented in the context of the OpenGL architecture, and we thus adopt the mathematical representations and coordinate systems used by the OpenGL library. For other graphics systems, such as Direct3D, in which the camera-space and clip-space coordinate systems differ slightly, the derivation in the next section can be reapplied to obtain an equivalent result.

A plane  $C$  is mathematically represented by a four-dimensional vector of the form

$$C = \langle N_x, N_y, N_z, -\mathbf{N} \cdot \mathbf{Q} \rangle, \quad (1)$$

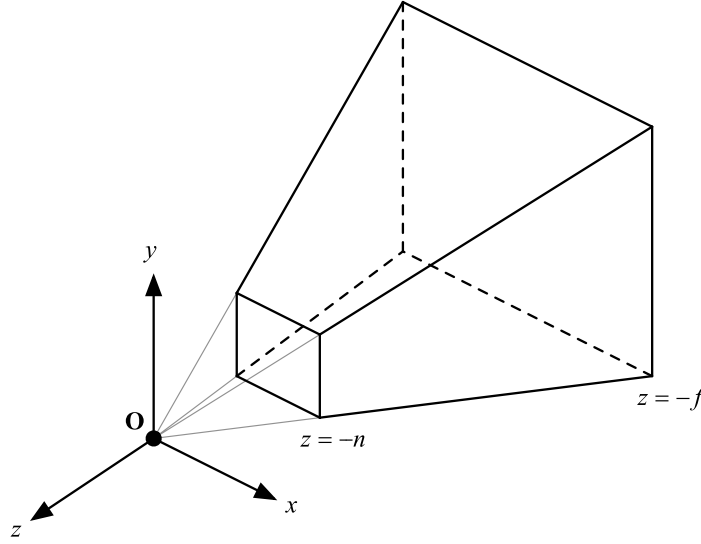
where  $\mathbf{N}$  is the normal vector pointing away from the front side of the plane, and  $\mathbf{Q}$  is any point lying in the plane itself. A homogeneous point  $\mathbf{P} = \langle P_x, P_y, P_z, P_w \rangle$  lies in the plane if and only if the four-dimensional dot product  $C \cdot \mathbf{P}$  is zero. If the normal vector  $\mathbf{N}$  of a plane  $C$  is unit length and  $P_w = 1$ , then the dot product  $C \cdot \mathbf{P}$  measures the signed perpendicular distance from the point  $\mathbf{P}$  to the plane  $C$ .

A plane is a covariant vector, and therefore must be transformed from one coordinate system to another using the inverse transpose of the matrix that transforms ordinary points (which are contravariant vectors). This is particularly important when transforming planes with the projection matrix since it is generally nonorthogonal. Given a camera-space point  $\mathbf{P}$  and a camera-space plane  $C$  represented by four-component column vectors, the projection matrix  $\mathbf{M}$  produces a clip-space point  $\mathbf{P}'$  and a clip-space plane  $C'$  as follows.

$$\begin{aligned}\mathbf{P}' &= \mathbf{M}\mathbf{P} \\ \mathbf{C}' &= (\mathbf{M}^{-1})^T \mathbf{C}\end{aligned}\tag{2}$$

Inverting these equations allows us to transform from clip space to camera space.

Recall that in OpenGL camera space, the camera lies at the origin and points in the  $-z$  direction, as shown in Figure 1. To complete a right-handed coordinate system, the  $x$ -axis points to the right, and the  $y$ -axis points upward. Vertices are normally transformed from whatever space in which they are specified into camera space by the model-view matrix. In this paper, we do not worry about the model-view matrix and assume that vertex positions are specified in camera space.



**Figure 1.** Camera space and the standard view frustum. The near and far planes are perpendicular to the  $z$ -axis and lie at the distances  $n$  and  $f$  from the camera, respectively.

The standard view frustum is the six-sided truncated pyramid that encloses the volume of space visible to the camera. As shown in Figure 1, it is bounded by four side planes representing the four edges of the viewport, a near plane at  $z = -n$ , and a far plane at  $z = -f$ . The near and far planes are normally perpendicular to the camera's viewing direction, but our modifications to the projection matrix will move these two planes and change the fundamental shape of the view frustum.

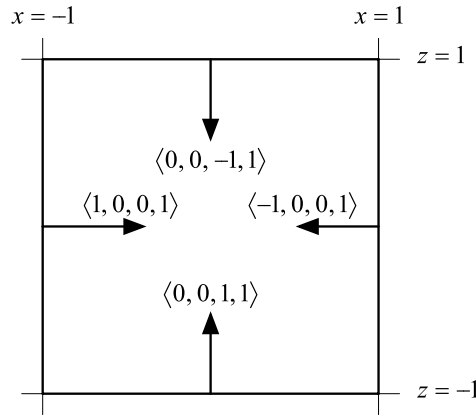
The projection matrix transforms vertices from camera space to homogeneous clip space. In homogeneous clip space, a four-dimensional point  $\langle x, y, z, w \rangle$  lies inside the projection of the camera-space view frustum if the following conditions are satisfied.

$$\begin{aligned}-w &\leq x \leq w \\ -w &\leq y \leq w \\ -w &\leq z \leq w\end{aligned}\tag{3}$$

Performing the perspective division by the  $w$ -coordinate moves points into normalized device coordinates, where each of the coordinates of a point in the view frustum lies in the interval  $[-1, +1]$ . Our goal is to modify the projection matrix so that points lying on a given arbitrary plane have a

$z$ -coordinate of  $-1$  in normalized device coordinates. We avoid examining any particular form of the projection matrix and require only that it be invertible. This allows our results to be applied to arbitrary projection matrices which may have already undergone some kind of modification from the standard forms.

Figure 2 shows the  $x$  and  $z$  components of a three-dimensional slice of the four-dimensional homogeneous clip space. Within this slice, the  $w$ -coordinate of every point is 1, and the projection of the view frustum described by Equation (3) is bounded by six planes forming a cube. The  $w$ -coordinate of each plane is 1, and exactly one of the  $x$ -,  $y$ -, and  $z$ -coordinates is  $\pm 1$ , as shown in Table 1. Given an arbitrary projection matrix  $\mathbf{M}$ , the inverse of Equation (2) can be used to map these planes into camera space. This produces the remarkably simple formulas listed in Table 1, in which each camera-space plane is expressed as a sum or difference of two rows of the projection matrix [2].



**Figure 2.** The  $x$  and  $z$  components of a three-dimensional slice of homogeneous clip space at  $w = 1$  and four of the six clipping planes that form a cube in this space.

Frustum Plane	Clip-space Coordinates	Camera-space Coordinates
Near	$\langle 0, 0, 1, 1 \rangle$	$\mathbf{M}_4 + \mathbf{M}_3$
Far	$\langle 0, 0, -1, 1 \rangle$	$\mathbf{M}_4 - \mathbf{M}_3$
Left	$\langle 1, 0, 0, 1 \rangle$	$\mathbf{M}_4 + \mathbf{M}_1$
Right	$\langle -1, 0, 0, 1 \rangle$	$\mathbf{M}_4 - \mathbf{M}_1$
Bottom	$\langle 0, 1, 0, 1 \rangle$	$\mathbf{M}_4 + \mathbf{M}_2$
Top	$\langle 0, -1, 0, 1 \rangle$	$\mathbf{M}_4 - \mathbf{M}_2$

**Table 1.** The six view frustum planes in clip space and camera space. The matrix  $\mathbf{M}$  represents the projection matrix that transforms points from camera space to clip space. The notation  $\mathbf{M}_i$  represents the  $i$ -th row of the matrix  $\mathbf{M}$ .

### 3 Clipping Plane Modification

Let  $\mathbf{C} = \langle C_x, C_y, C_z, C_w \rangle$  be the plane shown in Figure 3, having coordinates specified in camera space, to which we would like to clip our geometry. The camera should lie on the negative side of the plane, so we can assume that  $C_w < 0$ . The plane  $\mathbf{C}$  will replace the ordinary near plane of the view frustum. As shown in Table 1, the camera-space near plane is given by the sum of the last two rows of the projection matrix  $\mathbf{M}$ , so we must somehow satisfy

$$\mathbf{C} = \mathbf{M}_4 + \mathbf{M}_3. \quad (4)$$

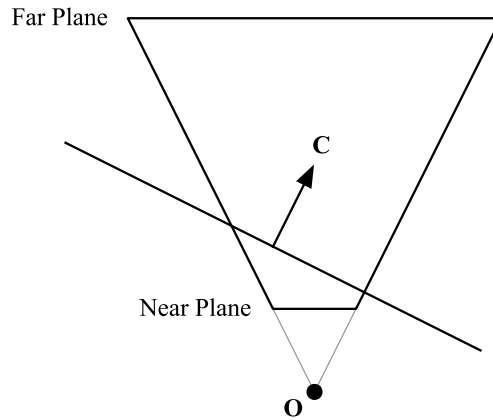
We cannot modify the fourth row of the projection matrix because perspective projections use it to move the negation of the  $z$ -coordinate into the  $w$ -coordinate, and this is necessary for perspective-correct interpolation of vertex attributes such as texture coordinates. Thus, we are left with no choice but to replace the third row of the projection matrix with

$$\mathbf{M}'_3 = \mathbf{C} - \mathbf{M}_4. \quad (5)$$

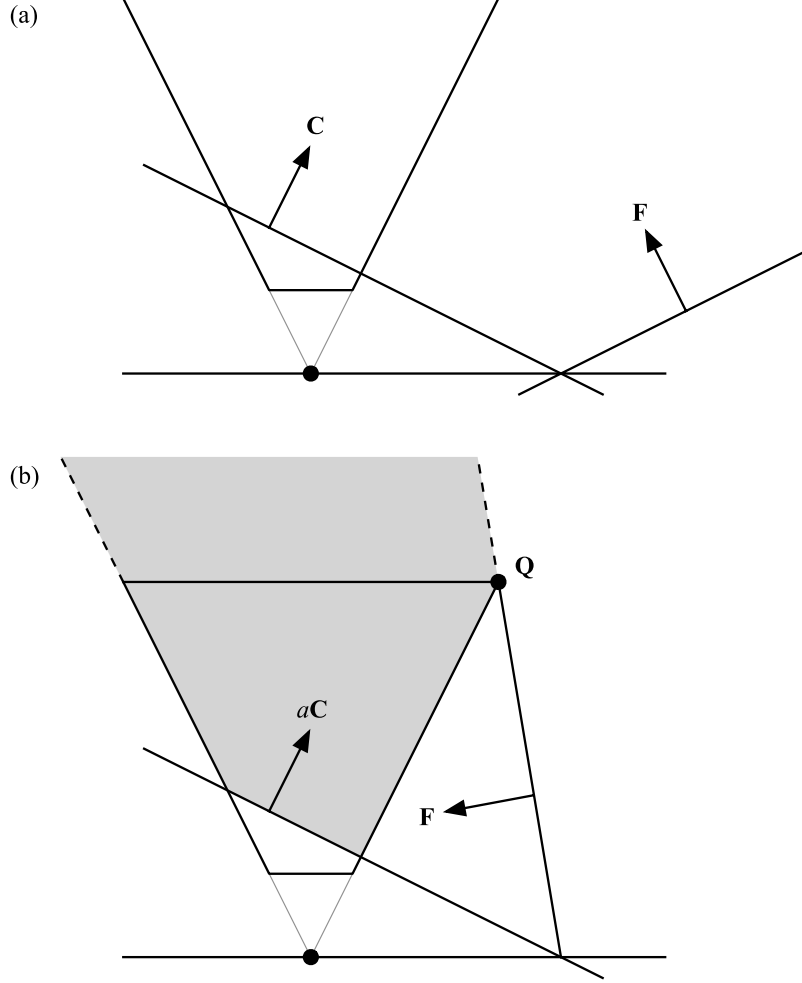
After making the replacement shown in Equation (5), the far plane  $\mathbf{F}$  of the view frustum becomes

$$\begin{aligned} \mathbf{F} &= \mathbf{M}_4 - \mathbf{M}'_3 \\ &= 2\mathbf{M}_4 - \mathbf{C}. \end{aligned} \quad (6)$$

This fact presents a significant problem for perspective projections. A perspective projection matrix must have a fourth row given by  $\mathbf{M}_4 = \langle 0, 0, -1, 0 \rangle$  so that the clip-space  $w$ -coordinate receives the negation of the camera-space  $z$ -coordinate. As a consequence, the near plane and far plane are no longer parallel if either  $C_x$  or  $C_y$  is nonzero. This is extremely unintuitive and results in a view frustum having a very undesirable shape. By observing that any point  $\mathbf{P} = \langle x, y, 0, w \rangle$  for which  $\mathbf{C} \cdot \mathbf{P} = 0$  implies that we also have  $\mathbf{F} \cdot \mathbf{P} = 0$ , we can conclude that the intersection of the near and far planes occurs in the  $x$ - $y$  plane, as shown in Figure 4(a).



**Figure 3.** The near plane of the view frustum is replaced with the arbitrary plane  $\mathbf{C}$ .



**Figure 4.** (a) The modified far plane  $F$  given by Equation (6) intersects the modified near plane  $C$  in the  $x$ - $y$  plane. (b) Scaling the near plane  $C$  by the value  $a$  given by Equation (9) adjusts the far plane so that the angle between the near and far planes is as small as possible without clipping any part of the original view frustum. The shaded area represents the volume of space that is not clipped by the modified view frustum.

Since the maximum projected depth of a point is achieved at the far plane, projected depth no longer represents the distance along the  $z$ -axis, but rather a value corresponding to the position between the new near and far planes. This has a severe impact on depth-buffer precision along different directions in the view frustum. Fortunately, we have a recourse for minimizing this effect, and it is to make the angle between the near and far planes as small as possible. The plane  $C$  possesses an implicit scale factor that we have not yet restricted in any way. Changing the scale of  $C$  causes the orientation of the far plane  $F$  to change, so we need to calculate the appropriate scale that minimizes the angle between  $C$  and  $F$  without clipping any part of the original view frustum, as shown in Figure 4(b).

Let  $C' = (\mathbf{M}^{-1})^T C$  be the projection of the new near plane into clip space (using the original projection matrix  $\mathbf{M}$ ). The corner  $Q'$  of the view frustum lying opposite the plane  $C'$  is given by

$$Q' = \langle \text{sgn}(C'_x), \text{sgn}(C'_y), 1, 1 \rangle. \quad (7)$$



(For most perspective projections, it is safe to assume that the signs of  $C'_x$  and  $C'_y$  are the same as  $C_x$  and  $C_y$ , so the projection of  $\mathbf{C}$  into clip space can be avoided.) Once we have determined the components of  $\mathbf{Q}'$ , we obtain its camera-space counterpart  $\mathbf{Q}$  by computing  $\mathbf{Q} = \mathbf{M}^{-1}\mathbf{Q}'$ . For a standard view frustum,  $\mathbf{Q}$  coincides with the point furthest from the plane  $\mathbf{C}$  where two side planes meet the far plane.

To force the far plane to contain the point  $\mathbf{Q}$ , we must require that  $\mathbf{F} \cdot \mathbf{Q} = 0$ . The only part of Equation (6) that we can modify is the scale of the plane  $\mathbf{C}$ , so we introduce a factor  $a$  as follows.

$$\mathbf{F} = 2\mathbf{M}_4 - a\mathbf{C}. \quad (8)$$

Solving the equation  $\mathbf{F} \cdot \mathbf{Q} = 0$  for  $a$  yields

$$a = \frac{2\mathbf{M}_4 \cdot \mathbf{Q}}{\mathbf{C} \cdot \mathbf{Q}}. \quad (9)$$

Replacing  $\mathbf{C}$  with  $a\mathbf{C}$  in Equation (5) gives us

$$\mathbf{M}'_3 = a\mathbf{C} - \mathbf{M}_4, \quad (10)$$

and this produces the optimal far plane orientation shown in Figure 4(b). It should be noted that this technique will also work correctly in the case that  $\mathbf{M}$  is an infinite projection matrix (i.e., one that places the conventional far plane at infinity) by forcing the new far plane to be parallel to one of the edges of the view frustum where two side planes meet.

## 4 Effect on the Standard View Frustum

Given the theory presented in the previous section, we now examine its application to the standard perspective projection matrix. We also analyze the effect that that an oblique near clipping plane has on depth buffer precision. The standard OpenGL perspective projection matrix  $\mathbf{M}$  is given by

$$\mathbf{M} = \begin{bmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & -\frac{2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix}, \quad (11)$$

where  $n$  is the distance to the near plane,  $f$  is the distance to the far plane, and the values  $l$ ,  $r$ ,  $b$ , and  $t$  represent the left, right, bottom, and top edges of the rectangle carved out of the near plane by the four side planes of the view frustum [3]. The inverse of  $\mathbf{M}$  is given by

$$\mathbf{M}^{-1} = \begin{bmatrix} \frac{r-l}{2n} & 0 & 0 & \frac{r+l}{2n} \\ 0 & \frac{t-b}{2n} & 0 & \frac{t+b}{2n} \\ 0 & 0 & 0 & -1 \\ 0 & 0 & -\frac{f-n}{2fn} & \frac{f+n}{2fn} \end{bmatrix}. \quad (12)$$

Given a clipping plane  $\mathbf{C}$ , Equations (9) and (10) give us the following formula for the third row of the modified projection matrix  $\mathbf{M}'$ .

$$\mathbf{M}'_3 = \frac{2\mathbf{M}_4 \cdot \mathbf{Q}}{\mathbf{C} \cdot \mathbf{Q}} \mathbf{C} - \mathbf{M}_4 \quad (13)$$

Since  $\mathbf{M}_4 = \langle 0, 0, -1, 0 \rangle$ , this simplifies to

$$\mathbf{M}'_3 = \frac{-2Q_z}{\mathbf{C} \cdot \mathbf{Q}} \mathbf{C} + \langle 0, 0, 1, 0 \rangle. \quad (14)$$

Applying the value of  $\mathbf{M}^{-1}$  given by Equation (12) to the point  $\mathbf{Q}'$  given by Equation (7), we have

$$\mathbf{Q} = \begin{bmatrix} \text{sgn}(C_x) \frac{r-l}{2n} + \frac{r+l}{2n} \\ \text{sgn}(C_y) \frac{t-b}{2n} + \frac{t+b}{2n} \\ -1 \\ 1/f \end{bmatrix}. \quad (15)$$

To test our procedure for modifying the projection matrix in the case that the clipping plane  $\mathbf{C}$  is perpendicular to the  $z$ -axis (i.e., parallel to the conventional near plane), we assign  $\mathbf{C} = \langle 0, 0, -1, -d \rangle$  for some positive distance  $d$ . The best that we can hope for is a new projection matrix in which the near plane has been moved to a distance  $d$  from the camera, but the far plane remains in its original position. Using the value of  $\mathbf{Q}$  given by Equation (15), the dot product  $\mathbf{C} \cdot \mathbf{Q}$  is simply  $1 - d/f$ . Evaluating Equation (14) yields the new third row of the projection matrix:

$$\begin{aligned} \mathbf{M}'_3 &= \frac{2}{1 - d/f} \langle 0, 0, -1, -d \rangle + \langle 0, 0, 1, 0 \rangle \\ &= \left\langle 0, 0, -\frac{f+d}{f-d}, -\frac{2fd}{f-d} \right\rangle. \end{aligned} \quad (16)$$

We have thus achieved the desired result. If  $d = n$ , then we recover the original projection matrix shown in Equation (11).

As mentioned earlier, modifying the view frustum to perform clipping against an arbitrary plane impacts depth-buffer precision because the full range of depth values may not be used along different directions in camera space. Given an arbitrary camera-space direction vector  $\mathbf{V} = \langle V_x, V_y, V_z, 0 \rangle$  with  $V_z < 0$ , a camera-space point  $\langle 0, 0, 0, 1 \rangle + s\mathbf{V}$ , with  $s > 0$ , produces the normalized device  $z$ -coordinate given by

$$\begin{aligned}
z(s) &= \frac{(\mathbf{M}' \langle sV_x, sV_y, sV_z, 1 \rangle)_z}{(\mathbf{M}' \langle sV_x, sV_y, sV_z, 1 \rangle)_w} \\
&= \frac{(a\mathbf{C} - \mathbf{M}_4) \cdot \langle sV_x, sV_y, sV_z, 1 \rangle}{\mathbf{M}_4 \cdot \langle sV_x, sV_y, sV_z, 1 \rangle},
\end{aligned} \tag{17}$$

where  $a$  is the scale factor given by Equation (9). For  $\mathbf{M}_4 = \langle 0, 0, -1, 0 \rangle$ , Equation (17) becomes

$$\begin{aligned}
z(s) &= \frac{asC_xV_x + asC_yV_y + asC_zV_z + sV_z + aC_w}{-sV_z} \\
&= \frac{as(\mathbf{C} \cdot \mathbf{V}) + sV_z + aC_w}{-sV_z}.
\end{aligned} \tag{18}$$

We assume that  $\mathbf{C} \cdot \mathbf{V} \geq 0$ , since otherwise no point  $\langle 0, 0, 0, 1 \rangle + s\mathbf{V}$  would fall within the viewable volume of space. Letting  $s$  tend to infinity yields the limit

$$\lim_{s \rightarrow \infty} z(s) = -\frac{a(\mathbf{C} \cdot \mathbf{V}) + V_z}{V_z}, \tag{19}$$

giving the maximum attainable normalized device  $z$ -coordinate in the direction  $\mathbf{V}$ .

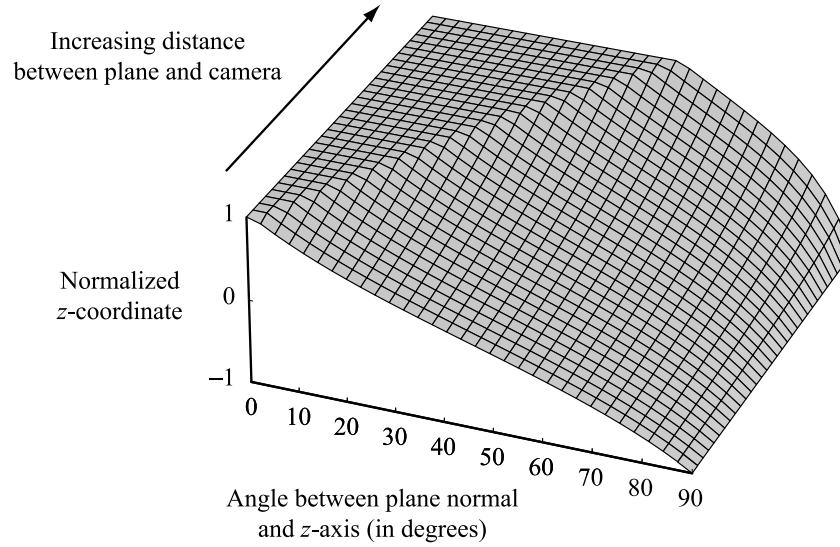
Let us consider the direction  $\mathbf{V} = \langle 0, 0, -1, 0 \rangle$  that looks straight ahead from the camera position. The limiting value given by Equation (19) is less than one when the condition  $aC_z > -2$  is satisfied. In this case, the  $z$ -coordinate of the far plane  $\mathbf{F}$  given by Equation (8) is less than zero, and the far plane never enters the volume of space bounded by the four side planes of the view frustum. Since the far plane can never be reached along the direction  $\mathbf{V}$ , the range of normalized depth values that can be attained is smaller than that of the ordinary view frustum. Figure 5 illustrates the effect of the clipping plane's orientation on the utilization of the depth buffer and demonstrates that the loss of precision can be considerable. In general, the precision decreases as the angle between the normal direction of the clipping plane  $\mathbf{C}$  and the  $z$ -axis increases and as the distance from the camera to the clipping plane increases.

## 5 Conclusion

An arbitrary invertible projection matrix  $\mathbf{M}$  can be modified so that the near plane is replaced by any camera-space clipping plane  $\mathbf{C}$ , with  $C_w < 0$ , by constructing a new projection matrix  $\mathbf{M}'$  as follows,

$$\mathbf{M}' = \begin{bmatrix} \mathbf{M}_1 \\ \mathbf{M}_2 \\ \frac{2\mathbf{M}_4 \cdot \mathbf{Q}}{\mathbf{C} \cdot \mathbf{Q}} \mathbf{C} - \mathbf{M}_4 \\ \mathbf{M}_4 \end{bmatrix}, \tag{20}$$

where  $\mathbf{Q} = \mathbf{M}^{-1}\mathbf{Q}'$ , and  $\mathbf{Q}'$  is given by Equation (7). If  $\mathbf{M}$  is a perspective projection, then  $\mathbf{M}_4 = \langle 0, 0, -1, 0 \rangle$  and  $\mathbf{M}'_3$  can be simplified to Equation (14). This procedure constructs the optimal view frustum by maximizing usage of the available values in the depth buffer. Although depth buffer precision is diminished, we have found that enough discrete depth values are still utilized in a standard 24-bit depth buffer to render images without artifacts in almost all real-world situations.



**Figure 5.** This surface represents the maximum attainable normalized device  $z$ -coordinate, as given by Equation (19), in the camera-space direction  $\mathbf{V} = \langle 0, 0, -1, 0 \rangle$ . The surface is plotted as a function of the angle between the near clipping plane's normal direction and the negative  $z$ -axis and the distance from the camera to the near plane. The area in which the surface is clamped to one represents plane angles for which the modified far plane intersects the  $z$ -axis in front of the camera.

## References

- [1] [http://developer.nvidia.com/object/oblique\\_frustum\\_clipping.html](http://developer.nvidia.com/object/oblique_frustum_clipping.html)
- [2] Eric Lengyel, *Mathematics for 3D Game Programming and Computer Graphics*. Charles River Media, 2002, p. 103.
- [3] OpenGL Architecture Review Board, *The OpenGL Graphics System: A Specification*. Silicon Graphics, Inc., 2004, p. 45.

# Unified Distance Formulas for Halfspace Fog

April 2006.

Published in *Journal of Graphics Tools*, Vol. 12, No. 2, 2007.

**Abstract.** In many real-time rendering applications, it is necessary to model a fog volume that is bounded by a single plane but is otherwise infinite in extent. This paper presents unified formulas that provide the correct distance traveled through a fog halfspace for all possible camera and surface point locations. Such formulas effectively remove the need to code for multiple cases separately, thereby achieving optimal fragment shading performance on modern rendering hardware.

## 1 Introduction

A fog volume bounded by a single plane, but otherwise infinite in extent, has many uses in interactive rendering applications. For example, the bounding plane could represent the maximum height that a fog bank reaches in an outdoor environment, it could coincide with a water plane to restrict fog application to underwater surfaces, or it could serve as the boundary between fogged and unfogged indoor areas. In such cases, the partial distance within the fog volume through which light travels between a surface point and the camera must be determined for each pixel rendered. Managing this calculation can be cumbersome, however, due to the different possible configurations of a surface point and the camera position with respect to the bounding plane. Furthermore, an implementation that handles distinct configurations as separate cases can suffer from poor performance caused by fragment shader code containing branches or using some kind of conditional execution. We therefore derive unified formulas that provide the correct distance traveled through a fog halfspace for all surface points and camera positions, allowing a single fragment shader to be used in all cases. One formula is derived for a volume having a constant fog density, and a second formula is derived for a volume having a fog density that increases linearly with distance from the bounding plane.

## 2 Background

As summarized in the following table, OpenGL defines three different fog modes that each calculate a fog fraction  $f$  in terms of a fog density  $\rho$  and the distance  $d$  that light travels through the fog [Leech 04]. (In the GL\_LINEAR case, an additional constant  $e$  represents the distance at which the fog fraction is zero, corresponding to fully saturated fog.)

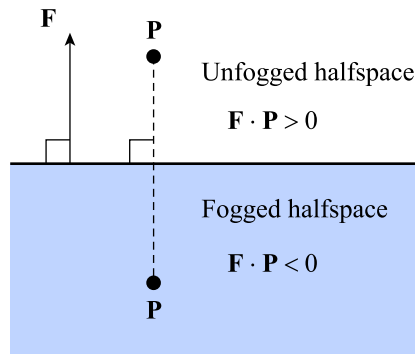
Fog Mode	Fog Fraction
GL_LINEAR	$f = \rho(e - d)$
GL_EXP	$f = \exp(-\rho d)$
GL_EXP2	$f = \exp(-\rho^2 d^2)$

Once  $f$  has been calculated for a fragment being rasterized, it is clamped to the range  $[0, 1]$ , and the fragment's final color  $C_r$  is blended with a constant fog color  $C_f$  to produce the color  $C$  which is written to the frame buffer using the formula

$$C = fC_r + (1 - f)C_f. \quad (1)$$

Ordinarily, the distance  $d$  is taken to be the total distance between the camera and the center of a fragment being rasterized.<sup>1</sup> In this paper, we consider fog volumes that are bounded by a plane dividing space into a fogged halfspace and an unfogged halfspace. To calculate the appropriate fog fraction  $f$  in this case for an arbitrary camera position, we need to determine the portion of the distance between the camera and a fragment's center that lies within the fogged halfspace to use as the value of  $d$ . Note that once the correct distance  $d$  has been calculated, any formula may be used to calculate the fog fraction  $f$ , not just the conventional OpenGL formulas.

We represent the bounding plane between fogged and unfogged halfspaces by the four-dimensional vector  $\mathbf{F} = \langle F_x, F_y, F_z, F_w \rangle$ . We shall assume that the plane's normal (described by the  $x$ ,  $y$ , and  $z$  components of  $\mathbf{F}$ ) has unit length and, as a convention, that the normal points away from the fogged halfspace. Recall that the dot product between the plane  $\mathbf{F}$  and a homogeneous point  $\mathbf{P}$  having a  $w$  coordinate of one gives the signed distance between the plane and the point. As illustrated in Figure 1, all points having a negative distance to the plane lie in the fogged halfspace, and all points having a positive distance to the plane lie in the unfogged halfspace. As explained later, whether points lying in the plane itself are considered fogged is unimportant and may be chosen one way or the other.

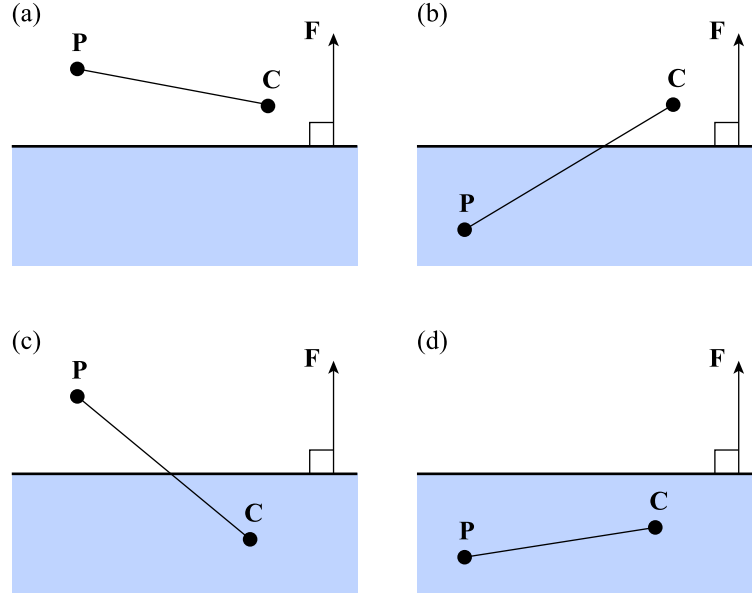


**Figure 1.** A point  $\mathbf{P}$  in the fogged halfspace forms a negative dot product with the plane vector  $\mathbf{F}$ , and a point in the unfogged halfspace forms a positive dot product with the plane vector.

<sup>1</sup>The OpenGL specification allows an implementation to approximate this distance with the fragment's camera-space depth.

### 3 Constant Density Fog Halfspace

We first consider a volume in which the fog density is given by a constant  $\rho$ . Suppose that the point  $\mathbf{P}$  represents a fragment inside a triangle being rasterized and that the point  $\mathbf{C}$  represents the camera position. When determining the partial distance along the direction from  $\mathbf{P}$  to  $\mathbf{C}$  lying within the fogged halfspace, there are four cases to consider based on the signs of  $\mathbf{F} \cdot \mathbf{P}$  and  $\mathbf{F} \cdot \mathbf{C}$ , as shown in Figure 2.



**Figure 2.** There are four distinct configurations to consider when calculating the partial distance along the direction from a surface point  $\mathbf{P}$  to the camera position  $\mathbf{C}$  that lies within the fogged halfspace.

In the case that  $\mathbf{P}$  and  $\mathbf{C}$  both lie on the positive side of the fog plane, no part of the line segment connecting the two points intersects the fogged halfspace, and thus no fog should be applied to the fragment corresponding to the point  $\mathbf{P}$ . In the case that  $\mathbf{P}$  and  $\mathbf{C}$  both lie on the negative side of the fog plane, the entire distance  $\|\mathbf{C} - \mathbf{P}\|$  should be used to calculate the amount of fog that should be applied.

The remaining two cases, in which  $\mathbf{F} \cdot \mathbf{P}$  and  $\mathbf{F} \cdot \mathbf{C}$  have opposite signs, require us to calculate the point at which the line segment connecting  $\mathbf{P}$  and  $\mathbf{C}$  intersects the plane  $\mathbf{F}$ . If we define the function  $\mathbf{Q}(t) = \mathbf{P} + t\mathbf{V}$ , where we have introduced  $\mathbf{V} = \mathbf{C} - \mathbf{P}$  as the traditional “view vector”, then a quick calculation yields

$$t = -\frac{\mathbf{F} \cdot \mathbf{P}}{\mathbf{F} \cdot \mathbf{V}} \quad (2)$$

as the parameter value for which  $\mathbf{Q}(t)$  lies in the plane. (Note that the  $w$  coordinate of  $\mathbf{V}$  is zero.) In the case that  $\mathbf{F} \cdot \mathbf{C} > 0$ , the distance traveled through the fog between  $\mathbf{P}$  and  $\mathbf{C}$  is  $t\|\mathbf{V}\|$ . In the case that  $\mathbf{F} \cdot \mathbf{C} < 0$ , the fogged distance is  $(1 - t)\|\mathbf{V}\|$ .

The following table summarizes the distances traveled through the fogged halfspace for the four cases illustrated in Figure 2.

Case	$\mathbf{F} \cdot \mathbf{P}$	$\mathbf{F} \cdot \mathbf{C}$	Distance $d$
(a)	Positive	Positive	0
(b)	Negative	Positive	$-\frac{\mathbf{F} \cdot \mathbf{P}}{\mathbf{F} \cdot \mathbf{V}} \ \mathbf{V}\ $
(c)	Positive	Negative	$\left(1 + \frac{\mathbf{F} \cdot \mathbf{P}}{\mathbf{F} \cdot \mathbf{V}}\right) \ \mathbf{V}\ $
(d)	Negative	Negative	$\ \mathbf{V}\ $

To satisfy the goal of optimal real-time rendering performance, we seek a single formula that gives the correct distance  $d$  for all cases simultaneously. Such a formula avoids expensive branches or other conditional code in the fragment shader.

First, we can make use of the fact that  $\mathbf{F} \cdot \mathbf{V}$  is always positive in case (b) and always negative in case (c). By applying an absolute value to  $\mathbf{F} \cdot \mathbf{V}$ , we can merge cases (b) and (c) into one formula and write  $d$  as

$$d = \left( k - \frac{\mathbf{F} \cdot \mathbf{P}}{|\mathbf{F} \cdot \mathbf{V}|} \right) \|\mathbf{V}\|, \quad (3)$$

where the constant  $k$  is defined as

$$k = \begin{cases} 1, & \text{if } \mathbf{F} \cdot \mathbf{C} \leq 0; \\ 0, & \text{otherwise.} \end{cases} \quad (4)$$

If Equation (3) were applied to case (a), then we would always obtain a negative distance when we want a distance of zero. If it were applied to case (d), then we would always obtain a distance greater than  $\|\mathbf{V}\|$  when we want the distance  $\|\mathbf{V}\|$  itself. Fortunately, this means that we can clamp the scale value to the range  $[0, 1]$  as follows to obtain the correct distance in all four cases.

$$d = \text{sat} \left( k - \frac{\mathbf{F} \cdot \mathbf{P}}{|\mathbf{F} \cdot \mathbf{V}|} \right) \|\mathbf{V}\| \quad (5)$$

The sat function represents the saturate operation available on all modern graphics processors that clamps values to the range  $[0, 1]$ .

If the direction between the points  $\mathbf{P}$  and  $\mathbf{C}$  is perpendicular to the fog boundary plane's normal, then  $\mathbf{F} \cdot \mathbf{V} = 0$  and we encounter a division-by-zero hazard in Equation (5). In this case,  $\mathbf{F} \cdot \mathbf{P}$  and  $\mathbf{F} \cdot \mathbf{C}$  must have the same sign, so the distance  $d$  becomes

$$d = \text{sat}(-\text{sgn}(\mathbf{F} \cdot \mathbf{C})\infty) \|\mathbf{V}\|. \quad (6)$$

The saturate operation maps the infinity to one if the camera lies inside the fog volume and zero otherwise, giving the correct results.

If the points  $\mathbf{P}$  and  $\mathbf{C}$  both lie in the plane  $\mathbf{F}$ , then  $\mathbf{F} \cdot \mathbf{P} = 0$  and  $\mathbf{F} \cdot \mathbf{V} = 0$ , leading to a zero-divided-by-zero expression in Equation (5) that produces a floating-point NaN (Not a Number) value. Whether a NaN is mapped to zero or one by the saturate operation is unspecified, but either case is acceptable because the pixel corresponding to the point  $\mathbf{P}$  is guaranteed to have neighbors for which  $d = 0$  and neighbors for which  $d = \|\mathbf{V}\|$ .



## 4 Linear Density Fog Halfspace

Multiplying the distance given by Equation (5) by a constant density value produces a fog volume that can appear too thick near the boundary plane. A constant density can also produce the appearance of a harsh transition between fogged and unfogged pixels when the camera is positioned near the boundary plane. A more natural result can be achieved by utilizing a density function  $\rho(\mathbf{P})$  that is zero on the bounding plane and increases linearly with distance into the fog volume. A comparison between constant density and linear density is shown in Figure 3.

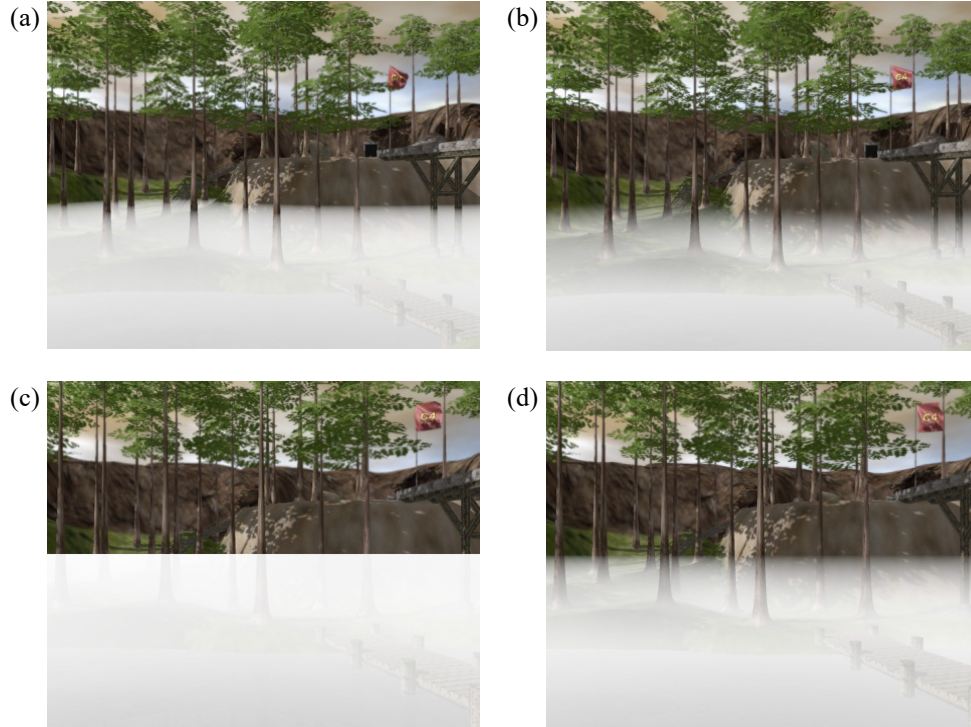
Let the density function  $\rho(\mathbf{P})$  be defined as

$$\rho(\mathbf{P}) = -a(\mathbf{F} \cdot \mathbf{P}) \quad (7)$$

for some positive constant  $a$ , and let  $dg$  represent the amount of fog applied along a differential length  $ds$ , given by the product

$$dg = \rho(\mathbf{P}) ds. \quad (8)$$

By integrating along the partial path between the points  $\mathbf{P}$  and  $\mathbf{C}$  that lies inside the fog volume, we obtain a function  $g(\mathbf{P})$  that can be substituted for the product  $\rho d$  in any of the fog fraction formulas.



**Figure 3.** Images (a) and (c) show a fog volume rendered with constant density  $\rho = 0.04$ , and images (b) and (d) show a fog volume rendered with a linear density function  $\rho(\mathbf{P}) = -0.008(\mathbf{F} \cdot \mathbf{P})$ . The camera is placed 5 meters above the fog volume's boundary plane in images (a) and (b), and the camera lies in the boundary plane in images (c) and (d).

Let the function  $\mathbf{Q}(u) = \mathbf{P} + u\mathbf{V}$  represent the line segment connecting  $\mathbf{P}$  and  $\mathbf{C}$  with  $u \in [0, 1]$ , where we continue to use  $\mathbf{V} = \mathbf{C} - \mathbf{P}$ . The differential distance  $ds$  can be expressed in terms of  $u$  as  $ds = \|\mathbf{V}\| du$ . We again need to consider the four possible configurations of the points  $\mathbf{P}$  and  $\mathbf{C}$  with respect to the boundary plane. Of course, if  $\mathbf{F} \cdot \mathbf{P}$  and  $\mathbf{F} \cdot \mathbf{C}$  are both positive, then  $g(\mathbf{P}) = 0$ . In the case that  $\mathbf{F} \cdot \mathbf{P} < 0$  and  $\mathbf{F} \cdot \mathbf{C} < 0$ , we integrate over the entire distance between  $\mathbf{P}$  and  $\mathbf{C}$  to obtain

$$\begin{aligned} g(\mathbf{P}) &= \int_{\mathbf{P}}^{\mathbf{C}} dg = \int_0^1 \rho(\mathbf{Q}(u)) \|\mathbf{V}\| du \\ &= -\frac{a}{2} \|\mathbf{V}\| (\mathbf{F} \cdot \mathbf{P} + \mathbf{F} \cdot \mathbf{C}). \end{aligned} \quad (9)$$

In the remaining two cases, for which  $\mathbf{F} \cdot \mathbf{P}$  and  $\mathbf{F} \cdot \mathbf{C}$  have opposite signs, the parameter  $t$  given by Equation (2) is one of the limits of integration. If only  $\mathbf{F} \cdot \mathbf{P} < 0$ , we have

$$g(\mathbf{P}) = \int_0^t \rho(\mathbf{Q}(u)) \|\mathbf{V}\| du = \frac{a}{2} \|\mathbf{V}\| \frac{(\mathbf{F} \cdot \mathbf{P})^2}{\mathbf{F} \cdot \mathbf{V}}, \quad (10)$$

and if only  $\mathbf{F} \cdot \mathbf{C} < 0$ , we have

$$g(\mathbf{P}) = \int_t^1 \rho(\mathbf{Q}(u)) \|\mathbf{V}\| du = -\frac{a}{2} \|\mathbf{V}\| \left( \mathbf{F} \cdot \mathbf{P} + \mathbf{F} \cdot \mathbf{C} + \frac{(\mathbf{F} \cdot \mathbf{P})^2}{\mathbf{F} \cdot \mathbf{V}} \right). \quad (11)$$

The fog functions  $g(\mathbf{P})$  for all four cases shown in Figure 2 are summarized in the following table.

Case	$\mathbf{F} \cdot \mathbf{P}$	$\mathbf{F} \cdot \mathbf{C}$	Fog Function $g(\mathbf{P})$
(a)	Positive	Positive	0
(b)	Negative	Positive	$\frac{a}{2} \ \mathbf{V}\  \frac{(\mathbf{F} \cdot \mathbf{P})^2}{\mathbf{F} \cdot \mathbf{V}}$
(c)	Positive	Negative	$-\frac{a}{2} \ \mathbf{V}\  \left( \mathbf{F} \cdot \mathbf{P} + \mathbf{F} \cdot \mathbf{C} + \frac{(\mathbf{F} \cdot \mathbf{P})^2}{\mathbf{F} \cdot \mathbf{V}} \right)$
(d)	Negative	Negative	$-\frac{a}{2} \ \mathbf{V}\  (\mathbf{F} \cdot \mathbf{P} + \mathbf{F} \cdot \mathbf{C})$

As in the constant density formula, we can merge cases (b) and (c) by applying an absolute value to the quantity  $\mathbf{F} \cdot \mathbf{V}$  and adjusting the sign of the term containing it. This yields the unified function

$$g(\mathbf{P}) = -\frac{a}{2} \|\mathbf{V}\| \left[ k(\mathbf{F} \cdot \mathbf{P} + \mathbf{F} \cdot \mathbf{C}) - \frac{(\mathbf{F} \cdot \mathbf{P})^2}{|\mathbf{F} \cdot \mathbf{V}|} \right] \quad (12)$$

for these two cases, where  $k$  is still defined by Equation (4). In order to incorporate case (d) into this formula, we need to eliminate the last term inside the brackets whenever  $\mathbf{F} \cdot \mathbf{P}$  and  $\mathbf{F} \cdot \mathbf{C}$  have

the same sign. This can be accomplished by replacing  $\mathbf{F} \cdot \mathbf{P}$  with  $\min((1 - 2k)(\mathbf{F} \cdot \mathbf{P}), 0)$  in the last term to arrive at the formula

$$g(\mathbf{P}) = -\frac{a}{2}\|\mathbf{V}\| \left[ k(\mathbf{F} \cdot \mathbf{P} + \mathbf{F} \cdot \mathbf{C}) - \frac{[\min((1 - 2k)(\mathbf{F} \cdot \mathbf{P}), 0)]^2}{|\mathbf{F} \cdot \mathbf{V}|} \right]. \quad (13)$$

Note that this formula also works for case (a) since both terms are eliminated if  $\mathbf{F} \cdot \mathbf{P}$  and  $\mathbf{F} \cdot \mathbf{C}$  are both positive.

In Equation (13), if the quantity  $\mathbf{F} \cdot \mathbf{V}$  is zero, then it is always true that the numerator of the last term is also zero. Although in practice a zero-divided-by-zero situation is rare enough to be ignored, a small positive  $\varepsilon$  can be added to  $|\mathbf{F} \cdot \mathbf{V}|$  without affecting the fog function's value significantly to guarantee that a NaN is not produced.

## 5 Implementation

We finish with a discussion of the implementation of Equations (5) and (13) using OpenGL fragment shaders. Example code is given in the GL high-level shading language defined by the GL\_fragment\_shader extension, now a core feature of the OpenGL 2.0 standard.

For a fog volume of constant density, the quantities  $\rho\mathbf{V}$ ,  $\mathbf{F} \cdot \mathbf{P}$ , and  $\mathbf{F} \cdot \mathbf{V}$  are calculated in a vertex program and interpolated across triangles during rasterization. The value of  $k$  only needs to be calculated once for a particular camera position, and it is stored in a constant register. The following fragment shader can be used to calculate the product  $\rho d$  and then apply the GL\_EXP fog fraction formula. Note that to match the fog fraction that would be produced by OpenGL, the density  $\rho$  should be multiplied by  $1/\ln 2$  because the `exp2` function exponentiates using the base 2 instead of the base  $e$ . (This function was chosen over the `exp` function to match the underlying hardware functionality of typical GPUs.)

```
uniform float k;           // (F dot C <= 0.0)
varying float3 rhoV;
varying float F_dot_V;
varying float F_dot_P;

void main()
{
    float4 color = ...;    // final color

    // Calculate distance * rho using Equation (5)
    float d = saturate(k - F_dot_P / abs(F_dot_V));
    d *= length(rhoV);

    // Calculate fog fraction and apply
    float f = saturate(exp2(-d));

    gl_FragColor.rgb = color.rgb * f + gl_FogParameters.color.rgb * (1.0 - f);
    gl_FragColor.a = color.a;
}
```

For a fog volume of linearly varying density, we can calculate the quantities  $(a/2)\mathbf{V}$ ,  $k(\mathbf{F} \cdot \mathbf{P} + \mathbf{F} \cdot \mathbf{C})$ , and  $(1 - 2k)(\mathbf{F} \cdot \mathbf{P})$  at each vertex and interpolate them during triangle rasterization. The fragment shader implementation of the linear density function is shown in the following code listing, which calculates  $g(\mathbf{P})$  and then applies the GL\_EXP fog fraction formula.

```
uniform float3 aV;           // (a / 2) * V
varying float c1;           // k * (F dot P + F dot C)
varying float c2;           // (1 - 2k) * (F dot P)
varying float F_dot_V;

void main()
{
    float4 color = ...;      // final color

    // Calculate g(P) using Equation (13)
    float g = min(c2, 0.0);
    g = -length(aV) * (c1 - g * g / abs(F_dot_V));

    // Calculate fog fraction and apply
    float f = saturate(exp2(-g));

    gl_FragColor.rgb = color.rgb * f + gl_FogParameters.color.rgb * (1.0 - f);
    gl_FragColor.a = color.a;
}
```

## References

- [Leech 04] Jon Leech and Pat Brown, eds. “The OpenGL Graphics System, Version 2.0.” Technical Specification, Silicon Graphics, Inc., 2004. Online at <http://www.opengl.org/documentation/specs/version2.0/glspec20.pdf>.
- [Legakis 98] Justin Legakis. “Fast Multi-Layer Fog.” *Conference Abstracts and Applications (SIGGRAPH 98)*, p. 266, 1998.

# Smooth Horizon Mapping

November 2006.

Published in *Game Engine Gems 3*, 2016.

## 1 Introduction

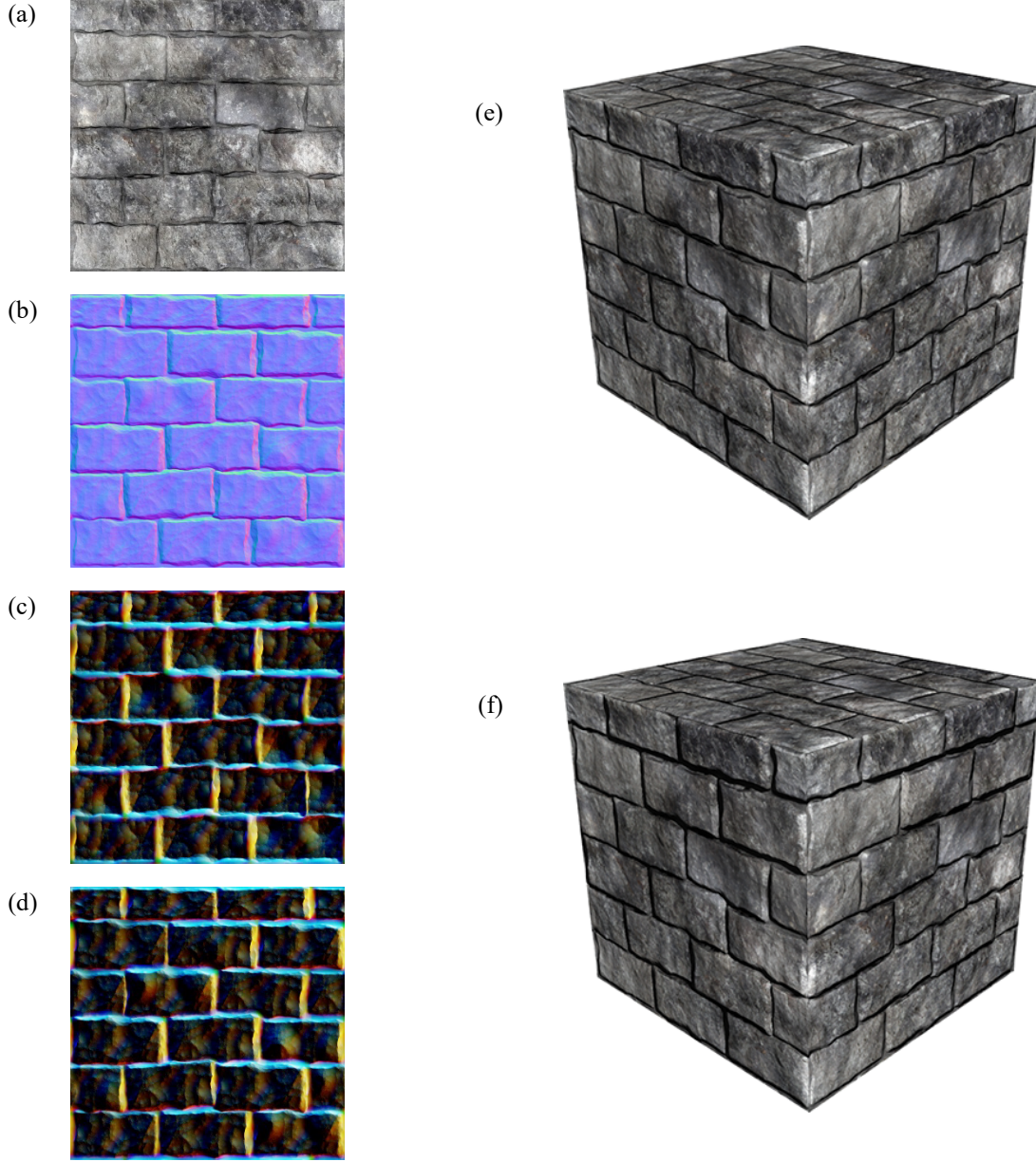
Normal mapping has been a staple of real-time rendering ever since the first graphics processors with programmable fragment pipelines became available. The technique varies the normal direction at each texel to make it appear that a smooth surface actually has a more complex geometric shape. Because the normal direction changes, the intensities of the diffuse and specular reflections also change, and this produces an effective illusion. But it is an illusion that could be taken further by accounting for the fact that some of the incoming light would be occluded if the more complex geometry really existed.

*Horizon mapping* is a technique, first proposed by [Max 1988] for offline rendering, that uses an additional texture map to store information about the height of the normal-mapped geometry in several directions within a neighborhood around each texel. Given the direction to the light source at each texel, the information in the horizon map can be used to cast convincing dynamic shadows for the high-resolution geometric detail encoded in the normal map.

This chapter describes the process of creating a horizon map and provides the details for a highly efficient real-time rendering method that adds soft shadows to normal-mapped surfaces. The rendering method works in tangent space and thus fits well into existing normal mapping shaders. Furthermore, it does not require newer hardware features, so it can be used across a wide range of GPUs.

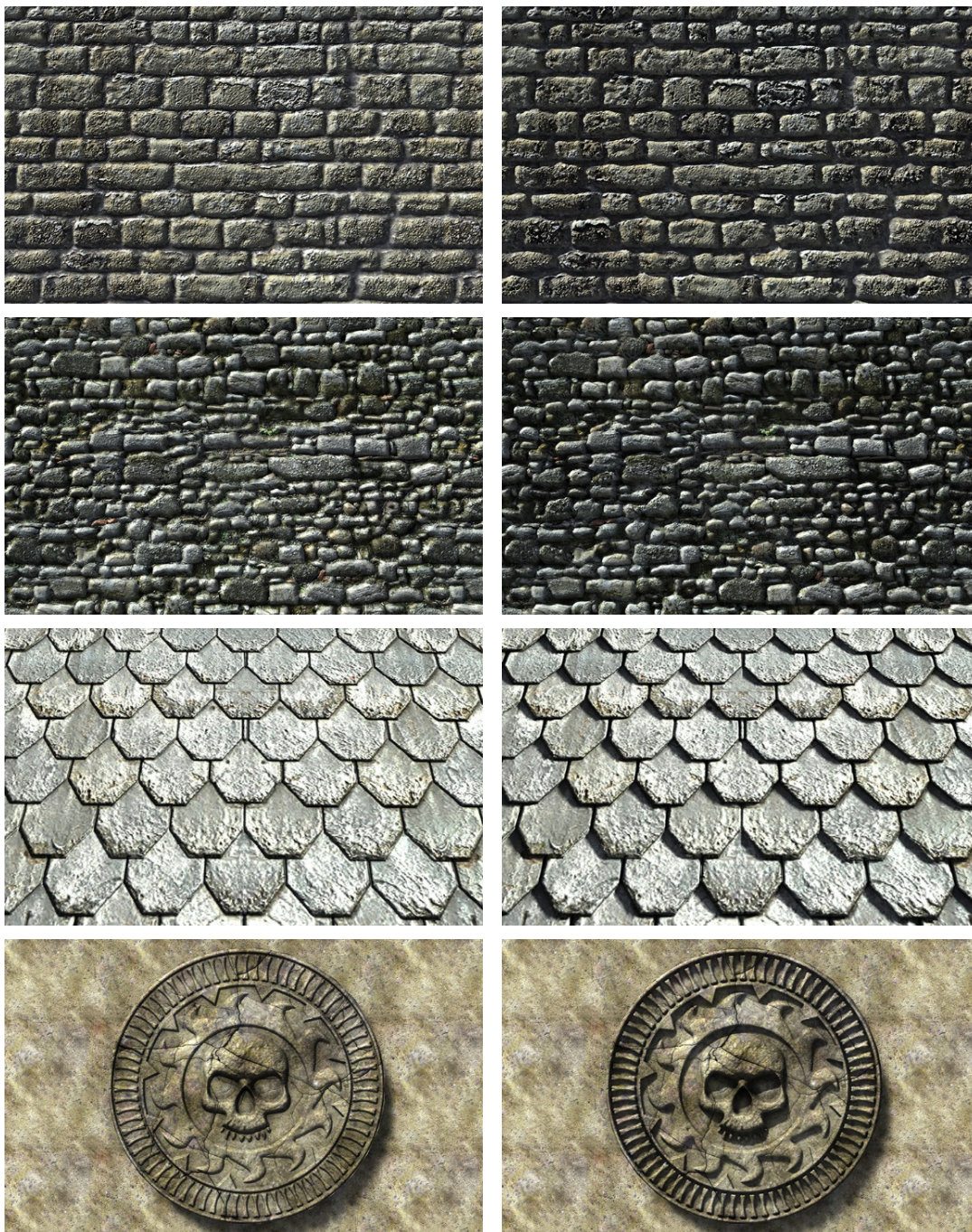
An example application of horizon mapping on a stone wall is shown in Figure 1. A conventional diffuse color map and normal map are used to render a cube having flat sides in Figure 1(e). A four-channel horizon map with two layers encodes horizon information for eight light directions, and this horizon map is used to render shadows on the same cube in Figure 1(f).

A variety of materials rendered with and without shadows generated by horizon maps are shown in Figure 2. The shadows make a significant contribution to the illusion that the surfaces are anything but flat. This is especially true if the light source is in motion and the shadows are changing dynamically. Even though the horizon map contains height information for only eight distinct tangent directions (every 45 degrees about the normal), linearly interpolating that information for other directions is quite sufficient for rendering convincing shadows. In Figure 3, the same material is rendered nine times as the direction to the light source rotates about the surface normal by 15-degree increments.



**Figure 1.** The conventional diffuse color map (a) and normal map (b) are augmented by eight channels of horizon mapping information in (c) and (d). A cube rendered only with ordinary normal mapping is shown in (e). Horizon mapping has been applied in (f).





**Figure 2.** A variety of horizon mapping examples. In the left column, only ordinary normal mapping is applied. In the right column, shadows are added by horizon mapping.



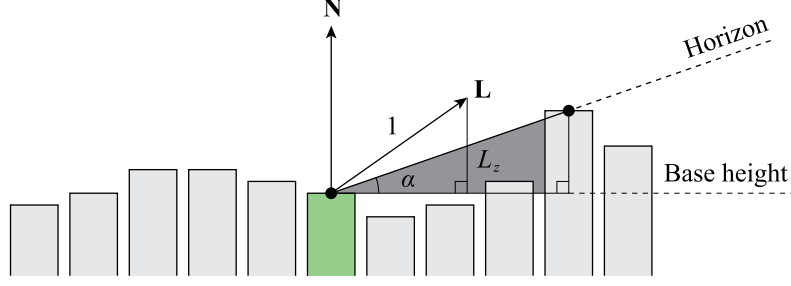


**Figure 3.** This flat disk is illuminated from nine different angles ranging from  $-60^\circ$  to  $+60^\circ$  in 15-degree increments. Shadows are dynamically rendered using information stored in the eight channels of the horizon map.

## 2 Horizon Map Generation

A horizon map contains eight channels of information, each corresponding to a tangent direction beginning with the positive  $x$  axis of the normal map and continuing at 45-degree increments counterclockwise. (In Figure 1, the alpha channels of the two layers of the horizon map are not visible, so a total of six channels can be seen as RGB color.) The intensity value in each channel is equal to the sine of the maximum angle made with the tangent plane for which a light source would be occluded in the direction corresponding to that channel. This is the horizon. As illustrated in Figure 4, a light source is unoccluded precisely when the direction to the light makes an angle having a larger sine. Otherwise, the pixel being shaded with the particular texel from the horizon map is in shadow. Given the tangent-space unit-length direction vector  $\mathbf{L}$  to the light source in a shader, determining whether a pixel is in shadow is simply a matter of comparing  $L_z$  to the value in the horizon map. The details about calculating this value for any light direction are discussed in the next section.





**Figure 4.** The horizon information for the central texel (green) in the direction to the right is given by the maximum angle  $\alpha$  determined by the heights of other texels in a local neighborhood. The value stored in the horizon map is  $\sin \alpha$ , and this is compared against  $L_z$  when rendering to determine whether the pixel being shaded is illuminated. If  $L_z > \sin \alpha$ , then light reaches the pixel; otherwise, the pixel is in shadow.

To construct a horizon map, we must calculate appropriate sine values for the horizon in eight directions. If the graphics hardware supports array textures, then the horizon map can be stored as a 2D texture image array with two layers. Otherwise, the horizon map must be stored as two separate 2D texture images at the tiny expense of needing to bind an additional texture map when rendering. In both cases, the texture images contain RGBA data with 8-bit channels. When the texture maps are sampled, the data is returned by the hardware as floating-point values in the range  $[0, 1]$ , which is a perfect match for the range of sine values that we need to store.

Using the raw height map as input, there are many valid ways of generating a horizon map. What follows is a description of the method used in the code accompanying this paper. For each texel in the output horizon map, we examine a neighborhood having a 16-texel radius in the height map and look for heights larger than that of the central texel under consideration. Each time a larger height is found, we calculate the squared tangent of the angle  $\alpha$  made with the central texel as

$$\tan^2 \alpha = \frac{(h - h_0)^2}{(x - x_0)^2 + (y - y_0)^2}, \quad (1)$$

where  $h$  is the height of the texel found at the location  $(x, y)$  in the height map, and  $h_0$  is the height of the central texel at the location  $(x_0, y_0)$ .

The maximum squared tangent is recorded for a array of 32 directions around the central texel, and after all of the texels in the neighborhood have been processed, the sine value for each direction is computed with

$$\sin \alpha = \frac{1}{\sqrt{\frac{1}{\tan^2 \alpha} + 1}}. \quad (2)$$

The affected directions in the array are determined by the position of the texel and its angular size relative to the central texel. It is possible for texels near the center to affect multiple entries in the array due to their larger angular sizes.

For each of the eight directions for which sine values are stored in the horizon map, the nearest five sine values in the array of 32 directions are simply averaged together. The sine values corresponding to the directions making angles  $0^\circ$ ,  $45^\circ$ ,  $90^\circ$ , and  $135^\circ$  with the  $x$  axis are stored in the red, green, blue, and alpha channels of the first layer of the horizon map, respectively. The sine values corresponding to the angles  $180^\circ$ ,  $225^\circ$ ,  $270^\circ$ , and  $315^\circ$  are stored in the red, green, blue, and alpha channels of the second layer.

Because the calculations are independent for each texel, the horizon map generation process is highly parallelizable. The horizon map can be sliced into subrectangles that are processed by different threads on the CPU, or each texel can be assigned to a thread to be processed by a GPU compute shader.

### 3 Rendering with Horizon Maps

Horizon mapping is applied in a shader by first calculating the color due to the contribution of a light source in the ordinary manner and then multiplying this color by an illumination factor derived from the information in the horizon map. The illumination factor is a value that is zero for pixels that are in shadow and one for pixels that are fully illuminated. We use a small trick to make the illumination factor change smoothly near the edge of the shadow to give it a soft appearance.

The horizon map stores the sine value corresponding to the horizon angle in eight tangent directions. For an arbitrary tangent-space direction vector  $\mathbf{L}$  to the light source, we interpolate between the horizon map's sine values for the two tangent directions nearest the projected light direction  $(L_x, L_y)$  and compare the interpolated value to  $L_z$  in order to determine whether a pixel is in shadow.

It would be expensive to decide which channels of the horizon map contribute to the sine value and to calculate the interpolation factors in the fragment shader. However, it is possible and quite convenient to store the interpolation factors for all eight channels of the horizon map in a special cube texture map that is accessed directly with the vector  $\mathbf{L}$ . We can encode factors for eight directions in a four-channel cube map by taking advantage of the fact that if a factor is nonzero for one direction, then it must be zero for the opposite direction. This allows us to use positive factors when referring to channels in the first layer of the horizon map and negative factors when referring to channels in the second layer. (We remap factors in the  $[-1, 1]$  range to  $[0, 255]$  when constructing the cube map so that we can use ordinary 8-bit RGBA color.) For a value  $\mathbf{f}$  sampled from the cube map with four components returned by the hardware in the range  $[0, 1]$ , we can compute the horizon sine value  $s$  as

$$s = \max(2\mathbf{f} - 1, 0) \cdot \mathbf{h}_0 + \max(-2\mathbf{f} + 1, 0) \cdot \mathbf{h}_1, \quad (3)$$

where  $\mathbf{h}_0$  and  $\mathbf{h}_1$  are the four-channel sine values read from layers 0 and 1 of the horizon map, and the max function is applied componentwise.

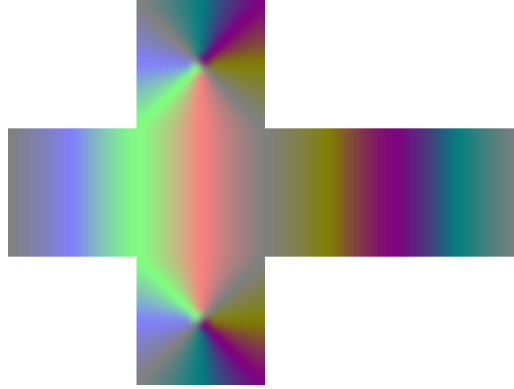
The cube texture map containing the channel factors has the appearance shown in Figure 5. This texture is included in the materials accompanying this paper along with code for generating it. We have found that a cube texture having a resolution of only  $16 \times 16$  texels per side is sufficient, and it requires a mere six kilobytes of storage.

Once the interpolated sine value  $s$  has been calculated with Equation (3) using the information sampled from the horizon map, we can compare it to the sine of the angle that the direction to light  $\mathbf{L}$  makes with the tangent plane, which is simply given by  $L_z$  when  $\mathbf{L}$  is normalized. If  $L_z \geq s$ , then the light source is above the horizon, and the pixel being shaded is therefore illuminated. If we were to compute an illumination factor  $F$  that is one when  $L_z \geq s$  and zero otherwise, then the shadow would be correct, but the shadow's edge would be hard and jagged as shown in Figure 6(a). We would instead like to have  $F$  smoothly transition from one to zero at the shadow's edge to produce the soft appearance shown in Figure 6(b). This can be accomplished by calculating

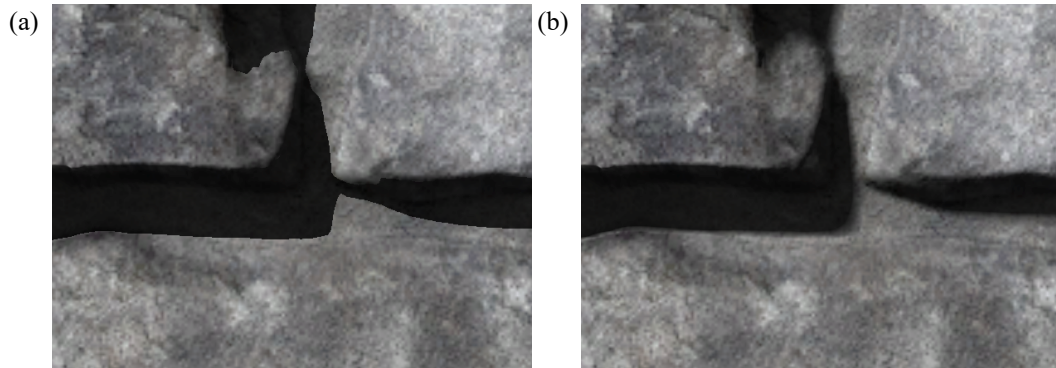
$$F = k(L_z - s) + 1 \quad (4)$$

and clamping it to the range  $[0, 1]$ . The value of  $F$  is always one, corresponding to a fully illuminated pixel, when  $L_z \geq s$ . The value of  $F$  is always zero, corresponding to a fully shadowed pixel, when  $L_z \leq s - 1/k$ . The constant  $k$  is a positive number that determines how gradual the transition from light to shadow is. The transition takes place over a sine value range of  $1/k$ , so smaller values of  $k$  produce softer shadows.

The fragment shader code in Listing 1 implements the horizon mapping technique described in this chapter. It assumes that the shading contribution from a particular light source has already been computed, possibly with larger-scale shadowing applied, and stored in the variable `color`. The code calculates the interpolated sine value for the horizon using Equation (3) and multiplies the RGB components of `color` by the illumination factor  $F$  given by Equation (4). (Instead of using the `max` function that appears in Equation (3), the code clamps to the range  $[0, 1]$  because many GPUs are able to saturate the result of multiply and multiply-add operations at no additional cost.) An ambient contribution would typically be added to `color` before it is output by the shader.



**Figure 5.** When sampled with coordinates given by the tangent-space direction to light vector, this cube texture map returns the channel factors to be applied to the information stored in the horizon map.



**Figure 6.** This close-up comparison shows the difference between a hard shadow and a soft shadow for a stone wall. (a) The illumination factor  $F$  is exactly one or zero, depending on whether  $L_z \geq s$ . (b) The illumination factor  $F$  is given by Equation (4) with  $k = 8.0$ .

**Listing 1.** This GLSL fragment shader code implements the horizon mapping technique. The texture `horizonMap` is a two-layer 2D array texture map that contains the eight channels of the horizon map. The texture `factorCube` is the special cube texture map that contains the channel factors for every light direction **L**. The interpolant `texcoord` contains the ordinary 2D texture coordinates used to sample the diffuse color map, normal map, etc. The interpolant `ldir` contains the tangent-space direction to the light source, which needs to be normalized before its *z* component can be used in Equation (4).

```
const float kShadowHardness = 8.0;

uniform sampler2DArray horizonMap;
uniform samplerCube factorCube;

in vec2 texcoord;    // 2D texture coordinates.
in vec3 ldir;        // Tangent-space direction to light.

void main()
{
    // The direction to light must be normalized.
    vec3 L = normalize(ldir);

    vec4 color = ...;    // Shading contribution from light source.

    // Read horizon channel factors from cube map.
    float4 factors = texture(factorCube, L);

    // Extract positive factors for horizon map layer 0.
    float4 f0 = clamp(factors * 2.0 - 1.0, 0.0, 1.0);

    // Extract negative factors for horizon map layer 1.
    float4 f1 = clamp(factors * -2.0 + 1.0, 0.0, 1.0);

    // Sample the horizon map and multiply by the factors for each layer.
    float s0 = dot(texture(horizonMap, vec3(texcoord, 0.0)), f0);
    float s1 = dot(texture(horizonMap, vec3(texcoord, 1.0)), f1);

    // Finally, multiply color by the illumination factor based on the
    // difference between Lz and the value derived from the horizon map.
    color.xyz *= clamp((L.z - (s0 + s1)) * kShadowHardness + 1.0, 0.0, 1.0);
}
```

## References

[Max 1988] Nelson L. Max. “Horizon mapping: shadows for bump-mapped surfaces”. *The Visual Computer*, Vol. 4, No. 2 (March 1988), pp. 109–117.

# A Graphical Scripting Language for Interactive Virtual Environments

February 2008.

**Abstract.** This paper describes a graphical scripting language that is suitable for encoding sequences of actions to be executed in an interactive virtual environment such as a computer game. A program is represented in this language primarily by a control flow graph, but a special type of node in such a graph may also encode a textual expression. Conditional execution and loops are encoded through properties assigned to the edges of the program graph. The user interface for construction of graphical scripts is also discussed.

## 1 Introduction

Most modern computer game engines, whether used for entertainment or other purposes, include some kind of high-level scripting capabilities that allow level designers to implement a sequence of actions without having to compile C++ code. A scripting system is usually accessible only by the developers of a product and is used to define a large range of functionality within each environment. For example, a script could control a sequence of events that occur in response to a character pulling a level inside a game.

A scripting language is typically implemented as a custom text-based language of some kind that is interpreted at run-time. This requires that a level designer learn the custom language and write code in a text editor. This process differs from writing C++ code in that the interpreted language usually has fewer features and greater type safety to prevent programming errors. However, designers are still writing code and are still required to understand that code in order to determine what a script actually does.

In this paper, we present the design and implementation details for an alternative type of scripting language that is largely graphical in nature. This makes the process of encoding action sequences more accessible to level designers who may not have a coding background, and it limits the types of errors that can occur. Our approach differs from previous implementations of graphical languages by providing named variables and general expression evaluation.

## 2 Implementation

### Program Structure

In our scripting language, a program is represented by a general directed graph. The nodes in the graph are called *methods* and correspond to fundamental atomic actions that can be performed by the underlying engine code. The edges in the graph are called *fibers* and each corresponds to the

flow of execution from one method to another. Each method may have any number of incoming fibers and any number of outgoing fibers. Cycles in the graph are allowed as this enables loop structures.

Aside from the control flow graph, any number of variables may be declared for use in a script. We explicitly do not include data flow edges in the graphical representation of the program. Including data dependencies often introduces a large amount of path redundancy in the graph, making it at very least visually unappealing. In our implementation, all variables are simply referenced by name whenever they are used as an input to a method or an output from a method.

Each type of method can expose a set of configurable parameters to the user. These parameters may be filled with constant values or configured so that they obtain their values at run time from script variables. Each method may also output a single value to a script variable. A separate Boolean result is generated by each method for use in conditional execution. If a method outputs a value of some kind, then it is customary for the Boolean result to reflect whether the output value is nonzero. Conditional execution is implemented by assigning one of three execution modes to each fiber in the graph. A fiber may execute always, execute on result true, or execute on result false. This allows different paths in the graph to be followed based on the Boolean results produced by the methods that are executed.

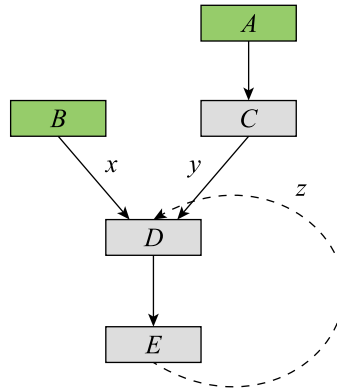
A fiber is identified as a looping edge in the graph if it satisfies the two conditions (a) its finish node is a predecessor of its start node in the graph, and (b) its finish node has a lower maximum depth in the graph with respect to a depth-first traversal starting at the nodes having no incoming edges.

## Execution Model

When a script begins execution, the methods having no incoming fibers are visited first and executed. From an abstract high-level perspective, these methods are executed in parallel since they have no control dependencies among themselves, but the actual execution takes place serially at lower levels. The serial order in which the initially executed methods run is purposely not defined. When a method finishes executing, each of its outgoing fibers is signaled as either live or dead depending on the Boolean result of the method and the execution mode of the fiber. Any fibers that are not dead looping fibers are signaled as ready. A subsequent method is ready for execution whenever (a) *all* of its non-looping incoming fibers are ready or (b) *any* of its looping incoming fibers are ready. If any of the incoming fibers are live, then the next method is executed normally. If all of the fibers are dead, then the next method is skipped, but its non-looping outgoing fibers are still signaled to be in the ready and dead state. This model allows execution to continue past points where different conditionally executed paths join back together. When a method begins execution, all of its incoming fibers are returned to a non-ready state.

In the example graph illustrated in Figure 1, methods *A* and *B* are executed first, and method *C* executes once method *A* has completed. Only when both fibers *x* and *y* are ready does method *D* execute in the first iteration through that point in the program. Later, after method *E* completes, the looping fiber *z* becomes ready, allowing method *D* to execute again without fibers *x* and *y* being ready.

As was the case with the initially executed methods, new methods visited through multiple fibers leaving a completed method can be thought of as executing in parallel from a high-level perspective. Since they are actually executed serially at the low level in a cooperatively scheduled fashion, the flow of execution is reminiscent of operating system fibers, and that is why the term fiber was chosen to describe the edges in the graph. The order in which ready methods immediately succeeding a completed method are executed is again undefined.



**Figure 1.** A program graph with a loop.

## Variables

A script may declare any number of variables that are accessible by every method. Each variable possesses a name, a type, a scope, and an initial value. Variables are statically typed as one of those in the following list.

- A Boolean value that is either true or false.
- A 32-bit signed integer value.
- A 32-bit floating-point value.
- A character string.
- A four-component RGBA color.
- A three-dimensional vector.

There are two possible scopes that a variable may possess. If a variable has *script scope*, then it resets to its initial value before each invocation of the script in which it is declared. If a variable has *object scope*, then it retains its value across multiple invocations of the script in a manner analogous to static local variables in C++. (We use the term object scope because in our engine, script data is stored in a generic container that we call an *object*.)

## Expressions

In order to support complex mathematical operations, we included a special type of method that encapsulates a text-based expression. When designing a script, the user is able to enter an expression that employs operations from the following list.

- Unary: + - ~
- Multiplicative: \* / %
- Additive: + -
- Shifts: << >>
- Relational: < > <= >= == !=
- Bit logic: & ^ |

An expression is parsed using the same syntactic rules that are defined by the C++ Standard, and thus the order of operator precedence is also the same. Parentheses may be used to group subexpressions. After an expression has been parsed, the resulting syntax tree is stored in the method. At run time, the syntax tree is traversed and evaluated using the current variable state.

Each operand appearing in an expression may be a script variable name or a literal value of type integer, float, or string. In the case that there is a type mismatch between the operands of a binary operator, the type of one of the operands is implicitly promoted to the type of the other, if possible. The legal promotions are those in the following list.

- Boolean can be promoted to integer, float, or string.
- Integer can be promoted to float or string.
- Float can be promoted to string.

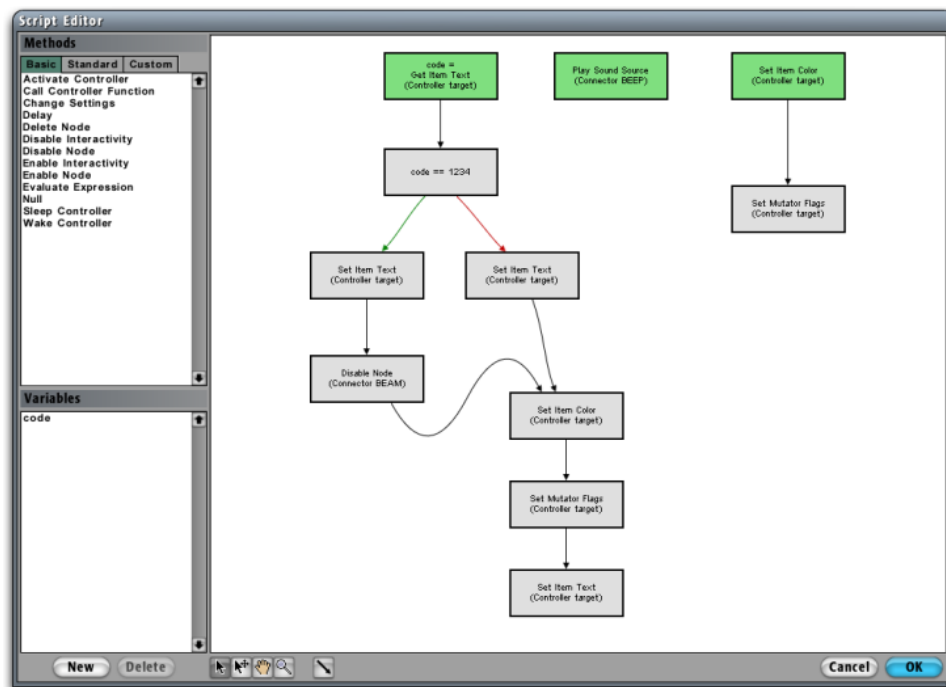
The type returned by the binary operator is equivalent to that of the non-promoted operand for all but the relational operators, which always return Boolean values.

If multiplication or division occurs between a multi-component value (color or vector) and a scalar value, then the scalar value is promoted to float if necessary, and the multiplicative operation is distributed to each component.

To eliminate the possibility of a run-time exception, we define the result of an integer division or modulo operation with a right-hand operand of zero to be zero.

## User Interface

We implemented a script editor that allows users to construct a program graph by placing methods in a viewport and connecting them with fibers. As shown in Figure 2, methods are represented by boxes containing some text indicating what actions they perform, and fibers are represented by curves with arrowheads pointing in the direction of execution.



**Figure 2.** The user interface for script editing.



The methods that are first to execute when a script is invoked are colored green to give the user a clear indication of how the program begins. Methods are colored red in the case that the user creates a closed loop having no incoming fibers from methods not belonging to a cycle in the graph. Methods in such a loop can never be executed, so the red color gives an indication of dead code. We included a null method that performs no operation to serve as both a sometimes-necessary predecessor node for loops and a node at which multiple paths of execution could be joined together before proceeding.

Fibers are drawn in different colors depending on their execution modes. A fiber set to execute always is colored black, a fiber set to execute on result true is colored green, and a fiber set to execute on result false is colored red. Independently from their colors, fibers may also be drawn as solid curves or dashed curves. A dashed curve represents a fiber that is a looping edge in the graph, while a solid curve represents an ordinary forward edge.

New methods are placed in the program graph by selecting a method type from a list on the left side of the editor window and then clicking in the graph viewport. A new fiber is drawn by selecting a tool at the bottom of the editor and dragging a line between two methods. Methods may be moved around after being placed in the graph, and the fibers connected to them recalculate their shapes dynamically. Of course, the physical location of a method has no relevance to program structure, but only serves to give the user a way to organize the program in a way that allows intuitive cognition of the flow of execution.

Variables are declared by adding them to a list on the left side of the editor window. When the user creates a new variable, he is able to specify its name, type, scope, and initial value. These can be edited later by double-clicking on an existing variable.

The input parameters and output variable for a method can be edited by double-clicking on a method in the graph viewport. When editing a method, a subwindow is displayed showing a list of input parameters for the method and, if a value is produced by the method, its output variable. The user may assign a constant value to each input parameter or may indicate that the input parameter be read from a variable.

It is allowable for an input variable to have a type different from that expected by the method for a particular input parameter. In such a case, the value of the input variable is implicitly converted at run time to the type of the parameter to which it is assigned, if possible. (In some cases, such as a conversion from an integer value to a Boolean value, information may be lost in this process.) The output variable for a method may also have a different type than that of the value produced by the method, and similar conversions are applied when necessary.

Our editor also includes support for standard cut, copy, and paste operations, as well as multiple levels of undo.

### 3 Related Work

The concept of a graphical programming language dates back to research begun in 1982 on what became the Prograph language [1]. Prograph and the many similar languages following it use directed graphs to represent programs just as in our scripting language, but the edges in the graph represent data flow as opposed to mere control flow. Each of these data flow languages requires that edges be mapped to specific input ports on the execution nodes, complicating the appearance of the graph. Furthermore, each piece of data is labeled separately for each node for which it is used as an input, leading to inconsistent naming of a single value throughout a program. We chose not to include data dependencies in our program graphs, but to instead simply assign global names to a set of variables that can be used anywhere in the program.

The only known use of a graphical language in a commercial game engine predating our work is the recently developed Kismet language that is now a part of the Unreal Engine [2]. Kismet also uses a data flow design with labeled input ports, and the program graph is further cluttered by special nodes required for representing constant input values. Kismet does not contain support for general expressions, but it does provide nodes that perform simple comparisons between two input values. Conditional execution is accomplished by attaching edges to separate true and false output ports on such nodes. Not enough documentation is publicly available for us to determine the full execution model of Kismet. In particular, the way in which loops and join nodes is handled is not known.

## 4 Conclusion

Our implementation of a graphical scripting language is now being used successfully in a production environment. We have conducted limited usability testing, and the results have shown that level designers welcome our simplified approach to graphical program representation wherein only control dependencies are considered. A key observation has been that our choice to use named variables in conjunction with a method for general expression evaluation provides a powerful tool that allows advanced level designers to implement programs more easily than they could in similar software lacking these capabilities.

## References

- [1] Steinman, Scott B., *Visual Programming with Prograph CPX*, Manning Publications, 1995.
- [2] Epic Games, *UnrealKismet*, <http://www.unrealtechnology.com/features.php?ref=kismet>.

# Motion Blur and the Velocity-Depth-Gradient Buffer

December 2009.

Published in *Game Engine Gems*, 2011.

Motion blur adds a significant amount of realism to a rendered scene since our eyes are accustomed to seeing it when we look at moving objects in the real world. There are several techniques for producing motion blur in computer graphics, and they vary widely in both rendering speed and image quality. Temporal supersampling, in which multiple frames are rendered and then combined to form one image, can produce very accurate results, but its extreme rendering expense makes it impractical for real-time applications like games. Techniques using an accumulation buffer of some kind to store previous frames to be combined with the current frame are fast, but produce terrible results in terms of image quality.

There is a class of motion blur techniques that are based on calculating per-pixel velocities and using them to collect many samples from the color buffer along the direction of motion. These techniques generally produce good results, but many of them produce a particular artifact that manifests itself as a fuzzy halo around foreground objects when the pixels behind them are moving quickly. The technique described in [1] makes no attempt to eliminate or reduce the appearance of this artifact. The method presented in [2] eliminates the artifact, but also eliminates some cases of correct motion blur, and it comes with some significant limitations.

There is but one method that is both fast and capable of producing high-quality images, and it involves the use of a velocity buffer in conjunction with a post-processing shader to render a directional blur for pixels belonging to moving objects. A basic implementation of this concept produces adequate results for some applications, but it also produces the fuzzy halo artifact. This paper discusses an improvement to this motion blur technique that eliminates halo artifacts without also affecting cases where motion blur would be correctly rendered, producing images of much higher quality than is possible with other techniques.

## 1 Technique Overview

The technique described in this paper requires that a dedicated four-channel velocity buffer be allocated by the rendering system. For each frame we render, we fill this velocity buffer with information about the two-dimensional screen-space velocity of pixels belonging to each object to which we want to apply the motion blur effect. This is typically done early in the rendering process for a particular frame, and it happens independently of any previously rendered frames.

When rendering to the velocity buffer, there are three sources of motion that we need to consider. First, we must take the motion of the camera into account, and this motion affects all objects in the scene. Second, we must consider the motion that individual objects have as a whole. An object may be moving through space or rotating, and this motion can be captured by considering

the object's transformation matrix for both the current frame and the preceding frame. Third, it's possible that the vertices composing an object's triangle mesh are themselves in motion. This frequently occurs with skinned character models, soft bodies, and cloth. Examples of motion blur arising from camera movement and object movement are shown in Figure 1. An example of motion blur due to vertex movement within a single mesh is shown in Figure 2.



**Figure 1.** (Left) Motion blur resulting only from camera movement. Notice how the ground and trees closer to the camera are blurred much more than distant objects. (Right) Motion blur resulting from rigid objects moving in the scene. Both translational and rotational motion is visible in this still image.



**Figure 2.** Motion blur resulting from vertex movement on a skinned character model.

At the end of the rendering process for a single frame, we apply to the entire screen a post-processing shader that generates the motion blur effect using the data stored in the velocity buffer. This shader can usually be combined with other post-processing effects such as glow, distortion, and color matrix application. The shader that generates the motion blur reads the screen-space velocity for each pixel from the velocity buffer and uses it to choose a set of sample points at which the color buffer is then read. These color samples are distributed along the direction of the velocity, and they are spread over a larger distance for higher velocities. After the color samples are collected, they are combined to generate the final color for each pixel.

It is not always the case that we want to use all of the color samples for a particular pixel. In particular, if a fast-moving object passes behind a slow-moving or stationary object with respect to the camera, then the motion blur applied to pixels belonging to the background object should not sample pixels belonging to the foreground object. To prevent this from occurring, we must be able to determine when pixels belong to the same object and when they don't while the post-processing shader is collecting color samples. This can be accomplished by storing additional information about the depth and the slope of surfaces in the velocity buffer.

In the two remaining available channels of the velocity buffer, we store the depth  $z$  of each pixel in camera space, and we store the magnitude of the two-dimensional gradient of the depth. These values give us the ability to calculate the minimum depth  $z_{\min}$  that a sample location must have in order to be considered part of the same surface as the pixel being blurred. The formula is

$$z_{\min} = z - r|\nabla z|, \quad (1)$$

where  $r$  is the distance from the sample location to the pixel being blurred. Color samples lying closer to the camera than this minimum depth are discarded. Figure 3 demonstrates how this technique eliminates artifacts that appear if the depth and gradient are not considered.



**Figure 3.** In these two images, the camera is rotating around the character, causing the ground to move across the screen while the character is almost completely still. In the left image, the depth and gradient information in the velocity buffer is not considered, and all color samples along the direction of the velocity are used. Notice the ghosting of the glowing parts of the character's armor and the fuzzy halo surrounding his legs. In the right image, the depth and gradient information in the velocity buffer is considered, and the rejection of the appropriate color samples eliminates the artifacts.

## 2 Rendering to the Velocity-Depth-Gradient Buffer

At some point in time before post-processing is performed, we must fill a dedicated velocity-depth-gradient buffer with the information that will be used to generate the motion blur effect. Many modern engines render a depth-only pass at the beginning of a frame in order to maximize the effectiveness of hierarchical depth buffering capabilities built into the graphics hardware. This gives us a convenient place to also render our velocity information without having to pass vertex data through the rendering pipeline a second time. Since many engines also require a linear depth value to be generated and stored early in the frame in order to render some types of special effects, it is doubly convenient that such a depth is one of the values we must calculate and store in the velocity buffer.

We render out velocity, depth, and gradient values into a floating-point buffer having 16 bits per channel. It is possible to implement the technique described in this paper using a conventional integer buffer with 8 bits per channel, but the small amount of available precision for the depth value in that case limits the practical range for which we can eliminate motion blur artifacts to an unacceptably short distance in front of the camera.

To calculate a two-dimensional screen-space velocity, we determine the screen-space positions for each vertex over two consecutive frames, subtract them, and then multiply by a normalizing scale factor. The homogeneous screen-space position  $\mathbf{P}_{\text{screen}}$  of a vertex is given by

$$\mathbf{P}_{\text{screen}} = \mathbf{M}_{\text{viewport}} \mathbf{M}_{\text{project}} \mathbf{M}_{\text{camera}} \mathbf{M}_{\text{model}} \mathbf{P}_{\text{model}}, \quad (2)$$

where  $\mathbf{P}_{\text{model}}$  is the model-space vertex position,  $\mathbf{M}_{\text{model}}$  is the matrix that transforms model-space points into world space,  $\mathbf{M}_{\text{camera}}$  is the matrix that transform world-space points into camera space,  $\mathbf{M}_{\text{project}}$  is the projection matrix for the camera, and  $\mathbf{M}_{\text{viewport}}$  is the viewport transformation. The values of  $\mathbf{M}_{\text{viewport}}$  and  $\mathbf{M}_{\text{project}}$  are ordinarily constant from one frame to the next, but the values of  $\mathbf{M}_{\text{camera}}$ ,  $\mathbf{M}_{\text{model}}$ , and  $\mathbf{P}_{\text{model}}$  can change. Thus, it is necessary to store the  $\mathbf{M}_{\text{camera}}$  matrix used by the camera during the preceding frame, and it is necessary to store the  $\mathbf{M}_{\text{model}}$  matrix used by each model during the preceding frame. If the model-space vertex positions can change for a particular model (for example, on a skinned character), then the entire array of vertex positions used by that model during the preceding frame must also be stored.

When rendering an object into the velocity buffer, we calculate the product of the four matrices in Equation (2) to construct the matrix  $\mathbf{M}_{\text{motion}}$  for both the preceding frame and the current frame as follows:

$$\begin{aligned} \mathbf{M}_{\text{motion}}^A &= \mathbf{M}_{\text{viewport}} \mathbf{M}_{\text{project}} \mathbf{M}_{\text{camera}}^A \mathbf{M}_{\text{model}}^A \\ \mathbf{M}_{\text{motion}}^B &= \mathbf{M}_{\text{viewport}} \mathbf{M}_{\text{project}} \mathbf{M}_{\text{camera}}^B \mathbf{M}_{\text{model}}^B. \end{aligned} \quad (3)$$

The superscript A indicates a matrix belonging to the preceding frame, and the superscript B indicates a matrix belonging to the current frame. The two products  $\mathbf{M}_{\text{motion}}^A$  and  $\mathbf{M}_{\text{motion}}^B$  are sent to the GPU as parameters that can be accessed by the vertex shader. As shown in Listing 1, these matrices are used in the vertex shader to calculate two homogeneous screen-space positions for each vertex and store them in texture coordinates that are interpolated as triangles are rasterized. The vertex shader shows the same position being transformed for both frames A and B, but in the case that  $\mathbf{P}_{\text{model}}$  is not constant, a second vertex position array must be specified and used when calculating the position for frame A.

The matrix multiplications performed by the vertex shader produce two four-dimensional homogeneous screen-space positions. It is important that these positions be interpolated in homo-

**Listing 1.** This GLSL vertex shader first transforms the vertex position for the current frame to homogeneous clip-space coordinates in the ordinary manner using the model-view-projection (MVP) matrix. The shader then transforms the vertex position into screen space for both the preceding frame using the matrix *motionA* and the current frame using the matrix *motionB*. The resulting positions are output as texture coordinates that will be read by the fragment shader.

```
uniform mat4    mvp, motionA, motionB;

void main()
{
    // Transform the position using the ordinary MVP matrix.
    gl_Position = mvp * gl_Vertex;

    // Transform the position into screen space using the motion matrix
    // from the preceding frame (A) and the current frame (B).
    gl_TexCoord[0] = motionA * gl_Vertex;
    gl_TexCoord[1] = motionB * gl_Vertex;
}
```

geneous form and that the perspective divide by the  $w$ -coordinate occurs in the fragment shader. Otherwise, the interpolated positions in the interiors of triangles would be incorrect, especially for triangles having vertices that lie behind the camera.

In the fragment shader used when rendering to the velocity buffer, we obtain two-dimensional screen-space positions for frames A and B by dividing the homogeneous interpolated positions  $\mathbf{P}_{\text{screen}}^A$  and  $\mathbf{P}_{\text{screen}}^B$  by their  $w$ -coordinates, as shown in Listing 2. Subtracting these positions then produces a screen-space velocity  $\mathbf{V}$  through the formula

$$\mathbf{V} = \frac{\mathbf{P}_{\text{screen}}^B}{(\mathbf{P}_{\text{screen}}^B)_w} - \frac{\mathbf{P}_{\text{screen}}^A}{(\mathbf{P}_{\text{screen}}^A)_w}, \quad (4)$$

where only the  $x$  and  $y$  components of the velocity are calculated.

The magnitude of the velocity  $\mathbf{V}$  is unbounded, but we can only read a limited number of color samples per pixel in the post-processing phase, and we don't want them to be too far away from the pixel being processed. Thus, to ensure a smooth blur, we need to impose some kind of bounds on the velocity's size. We first divide the velocity by the maximum distance  $r_{\text{max}}$  that we want to allow between a pixel's location and any color sample used to blur it. The value  $1/r_{\text{max}}$  is a constant that is passed to the fragment shader as a parameter by which the velocity is multiplied. We can also include in this parameter a normalization factor  $s$  that accounts for the time in between two frames and adjusts the overall intensity of the motion blur effect. We define  $s$  as

$$s = \frac{t_0}{\Delta t} m, \quad (5)$$

where  $t_0$  is the target time interval between frames,  $\Delta t$  is the actual time between the preceding frame and the current frame, and  $m$  is an adjustable factor that controls the motion blur intensity. The scaled screen-space velocity  $\mathbf{V}'$  is given by

$$\mathbf{V}' = \frac{s}{r_{\text{max}}} \mathbf{V}. \quad (6)$$

After applying this scale factor, we clamp the velocity's magnitude to the range  $[0, 1]$  using the following formula to preserve its direction:

$$\mathbf{V}_{\text{final}} = \frac{\mathbf{V}'}{\max\{|V'_x|, |V'_y|, 1\}}. \quad (7)$$

The  $x$  and  $y$  components of the velocity  $\mathbf{V}_{\text{final}}$  are stored in the red and green channels of the color output to the velocity buffer.

What remains is to write the depth and gradient information to the blue and alpha channels of the velocity buffer. The linear camera-space depth is supplied by the  $w$ -coordinate of the position for the current frame, and it is simply copied to the blue channel of the output color. To obtain the gradient of the depth, we query the hardware for the derivatives of the position's  $w$ -coordinate in both the  $x$  and  $y$  screen directions. To achieve slightly higher performance, we output the larger absolute value of the two derivatives in the alpha channel instead of computing the actual magnitude of the gradient. (That is, we compute the maximum norm instead of the Euclidean norm.) In the fragment shader shown in Listing 2, the gradient magnitude  $g$  stored in the alpha channel is given by

$$g = \max\left\{\left|\frac{\partial z}{\partial x}\right|, \left|\frac{\partial z}{\partial y}\right|\right\}. \quad (8)$$

**Listing 2.** This GLSL fragment shader calculates the screen-space velocity and writes it to the red and green components of the output color. The `velocityScale` parameter holds the value of  $s/r_{\text{max}}$  shown in Equation (6). The depth is taken directly from the  $w$ -coordinate of the current position and is written to the blue component of the output color. The gradient of the depth is calculated using the hardware derivative functions, and the larger of its components is written to the alpha component of the output color.

```
uniform vec2    velocityScale;

void main()
{
    // Divide by the w-coordinates to get 3D positions.
    vec2 posA = gl_TexCoord[0].xy / gl_TexCoord[0].w;
    vec2 posB = gl_TexCoord[1].xy / gl_TexCoord[1].w;

    // Subtract the positions and scale to get velocity.
    vec2 veloc = (posB.xy - posA.xy) * velocityScale;

    // Clamp the velocity to a max magnitude of 1.0.
    float vmax = max(abs(veloc.x), abs(veloc.y));
    gl_FragColor.xy = veloc / max(vmax, 1.0);

    // Pass the current depth through.
    gl_FragColor.z = gl_TexCoord[1].w;

    // Calculate the max component of the depth gradient.
    vec2 grad = vec2(ddx(gl_TexCoord[1].w), ddy(gl_TexCoord[1].w));
    gl_FragColor.w = max(abs(grad.x), abs(grad.y));
}
```



### 3 Rendering the Post-Processing Effect

At the end of a frame, the motion blur effect is generated for the final image by rendering a post-processing shader over the entire screen. This shader uses the information in the velocity-depth-gradient buffer to select a set of sample locations from which the color buffer is read. All of the color samples that satisfy the minimum depth requirement are averaged together to produce the final color for each pixel. Since the sample locations are derived from the magnitude and direction of the velocity that a pixel possesses, the result is an image containing convincing motion blur.

The fragment shader shown in Listing 3 performs the motion blur operation. It starts by sampling the color buffer at the pixel location being rendered, which we call the “center pixel”, and initializing the number of valid samples to one. The number of valid samples is stored in the  $w$  component of the color, and the sample at the center pixel is always valid. This particular implementation takes eight additional equally weighted samples from the color buffer at equally spaced intervals in the direction parallel to the velocity. There is some freedom in choosing the number of samples and their weights, and some implementations may even decide to take a variable number of color samples based on the magnitude of the velocity.

The velocity-depth-gradient buffer is read at the location of the center pixel, and the minimum depth required for all color samples is calculated as

$$z_{\min} = z - r_{\max} \max\{g, 0.001\}, \quad (7)$$

where  $z$  is the depth stored in the blue channel of the buffer,  $g$  is the depth gradient given by Equation (8) stored in the alpha channel of the buffer, and  $r_{\max}$  is the largest distance between the center pixel and a sample location. This is a little different from Equation (1) because we use the maximum sample distance  $r_{\max}$  to compute a single minimum depth  $z_{\min}$  that is used for all sample values. We clamp the minimum value of the gradient to 0.001 so that precision errors don’t prevent the motion blur effect from working on surfaces that are nearly perpendicular to the view direction.

The value of  $r_{\max}$  is 7.0 in the fragment shader shown in Listing 3, and color samples are taken at offsets given by the velocity vector multiplied by the values 1.75, 3.5, 5.25, and 7.0. At each sample location, the velocity buffer is read, but only to fetch the depth at that sample location and compare it to  $z_{\min}$ . (Velocity and gradient information is only used at the center pixel.) For any sample satisfying  $z \geq z_{\min}$ , we add the color sample to the final color and add one to the number of valid samples (in the  $w$  component of the color). After all samples have been taken, we divide the final color by the number of valid samples that have been accumulated and output the result.

**Listing 3.** This GLSL fragment shader applies the motion blur effect in the post-processing pass. The nine color samples are accumulated in the  $x$ ,  $y$ , and  $z$  components of the color vector, and the number of valid samples is stored in the  $w$  component of the color vector. The value of `minDepth` is calculated using Equation (7), and only samples having a depth at least this far from the camera plane are used to generate the final blurred pixel.

```
#extension GL_ARB_texture_rectangle : require

uniform sampler2DRect    colorTexture;
uniform sampler2DRect    velocityTexture;

void main()
{
    vec4    color, sample;
```

```

// Read the center sample from the color buffer.
color.xyz = texRECT(colorTexture, gl_FragCoord.xy).xyz;
color.w = 1.0;

// Read the velocity buffer at the current pixel.
float4 velo = texRECT(velocityTexture, gl_FragCoord.xy);

// Calculate the minimum depth for other color samples.
float minDepth = velo.z - max(velo.w, 0.001) * 7.0;

// Initialize constant sample weight.
sample.w = 1.0;

// Calculate coordinates for first sample on either side.
vec4 coord = velo.xyxy * vec4(1.75, 1.75, -1.75, -1.75) + gl_FragCoord.xyxy;

// Read a color and depth at the sample location.
sample.xyz = texRECT(colorTexture, coord.xy).xyz;
float depth = texRECT(velocityTexture, coord.xy).z;

// Add the sample to the final color if it's depth is great enough.
if (depth >= minDepth) color += sample;

// Grab the sample on the opposite side of the center pixel.
sample.xyz = texRECT(colorTexture, coord.zw).xyz;
depth = texRECT(velocityTexture, coord.zw).z;
if (depth >= minDepth) color += sample;

// Calculate coordinates for second pair of samples.
coord = velo.xyxy * vec4(3.5, 3.5, -3.5, -3.5) + gl_FragCoord.xyxy;
sample.xyz = texRECT(colorTexture, coord.xy).xyz;
depth = texRECT(velocityTexture, coord.xy).z;
if (depth >= minDepth) color += sample;

sample.xyz = texRECT(colorTexture, coord.zw).xyz;
depth = texRECT(velocityTexture, coord.zw).z;
if (depth >= minDepth) color += sample;

// Calculate coordinates for third pair of samples.
coord = velo.xyxy * vec4(5.25, 5.25, -5.25, -5.25) +
        gl_FragCoord.xyxy;
sample.xyz = texRECT(colorTexture, coord.xy).xyz;
depth = texRECT(velocityTexture, coord.xy).z;
if (depth >= minDepth) color += sample;

sample.xyz = texRECT(colorTexture, coord.zw).xyz;
depth = texRECT(velocityTexture, coord.zw).z;
if (depth >= minDepth) color += sample;

// Calculate coordinates for fourth pair of samples.
coord = velo.xyxy * vec4(7.0, 7.0, -7.0, -7.0) + gl_FragCoord.xyxy;
sample.xyz = texRECT(colorTexture, coord.xy).xyz;

```

```

depth = texRECT(velocityTexture, coord.xy).z;
if (depth >= minDepth) color += sample;

sample.xyz = texRECT(colorTexture, coord.zw).xyz;
depth = texRECT(velocityTexture, coord.zw).z;
if (depth >= minDepth) color += sample;

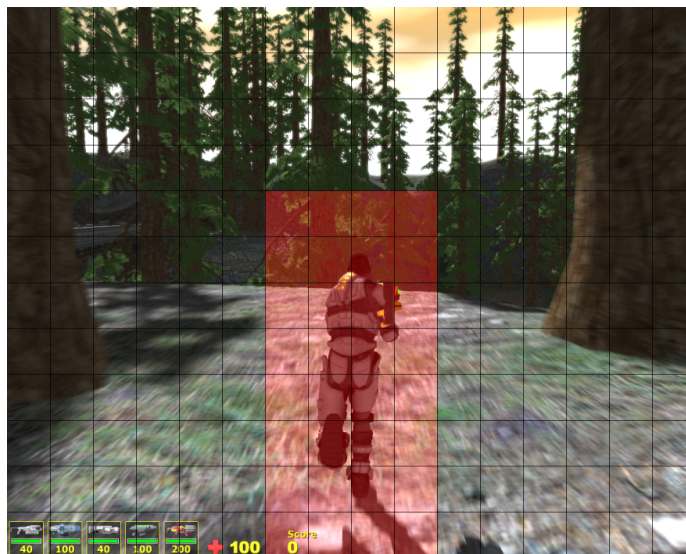
// Total weight of used color samples is in the w-coordinate.
// Divide by it to get the final averaged color.
gl_FragColor.xyz = color.xyz / color.w;
}

```

## 4 Grid Optimization

The fragment shader presented in Listing 3 produces very precise results, but it can be unnecessarily expensive for large parts of the scene. When it is known that a region of the screen contains pixels that are all moving a similar speeds relative to the camera, a simpler shader that does not consider depth can be used in order to increase overall performance. The use of the full implementation can be limited to those areas of the screen in which slow-moving foreground objects are expected to appear.

In Figure 4, we have divided the screen into a  $16 \times 12$  grid of rectangular cells. Before rendering the post-processing shader, we determine whether any foreground objects might be slow moving relative to the background and mark cells covered by those objects as requiring the full-blown shader. This would typically be done when objects are being rendered into the velocity buffer near the beginning of the frame. In the post-processing phase, we apply the simpler, faster shader to cells that have not been marked.



**Figure 4.** In this image, the viewport is partitioned into a grid of  $16 \times 12$  cells. The depth and gradient information is only used in the highlighted cells surrounding the character since that is where foreground objects are likely to be moving slowly relative to the background.

## References

- [1] Gilberto Rosado. “Motion Blur as a Post-Processing Effect”. *GPU Gems 3*, Addison-Wesley, 2008.
- [2] Ben Padget. “Efficient Real-Time Motion Blur for Multiple Rigid Objects”. *ShaderX7*. Charles River Media, 2009.

# Transition Cells for Dynamic Multiresolution Marching Cubes

March 2010.

Published in *Journal of Graphics, GPU, and Game Tools*, Vol. 15, No. 2, 2011.

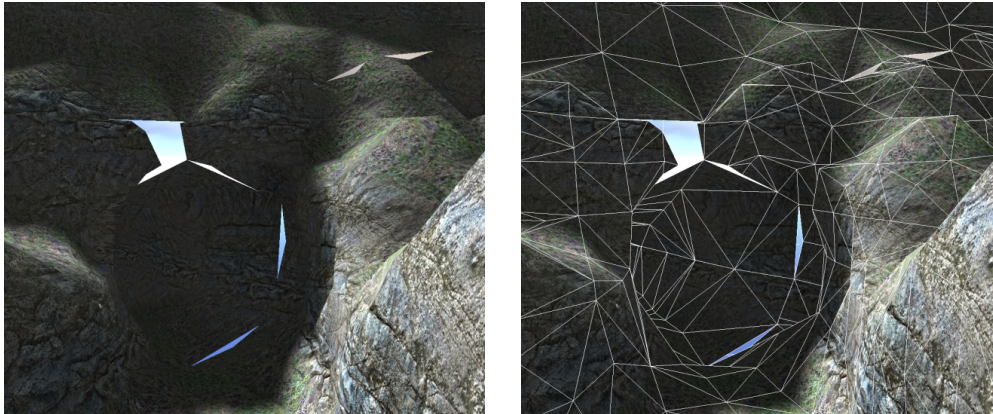
**Abstract.** This paper presents a method for quickly and seamlessly stitching together triangle meshes generated from multiresolution voxel data with the marching cubes algorithm. A layer of “transition cells” is inserted between volumes of differing voxel resolutions, and triangles are generated for these cells using a new algorithm that is based on a concept similar to marching cubes, but operates on voxel data at two different resolutions. To enable high performance for dynamically changing voxel data, our stitching algorithm requires access only to local data in the voxel map.

## 1 Introduction

Many games and other real-time simulations are turning toward a voxel representation of large complex surfaces due to the flexibility in the types of topographies they are able to describe. For example, a voxel representation of terrain can describe formations such as overhangs, arches, and truly vertical cliffs that would be impossible to achieve using a conventional height-based representation. The voxel representations are not rendered directly, but triangle meshes are instead generated using an isosurface extraction algorithm such as marching cubes [Lorensen and Cline 87]. The high vertex and triangle densities resulting from the application of marching cubes raises the importance of a level-of-detail system for high-performance rendering on the GPU.

Large voxel-based surfaces are rarely rendered as a monolithic mesh, but are instead partitioned into many components based on a coarse two- or three-dimensional grid. This is done for a variety of reasons that include the need to perform visibility determination at a reasonable granularity and the ability to dynamically load and unload meshes as the camera moves through the scene. Such an organization also provides a natural set of boundaries at which the detail level can be chosen for different blocks of voxel data. A natural choice for creating different levels of detail is to simply generate triangle meshes for voxel data at half the resolution in each dimension, one-quarter the resolution, and so on. Meshes corresponding to the highest resolution voxel data are rendered near the camera, and meshes corresponding to voxel data of discretely decreasing resolutions are rendered at distances further from the camera.

The primary problem that arises is the formation of a variety of prominent artifacts like those shown in Figure 1 along the boundaries between meshes representing different detail levels. In this paper, we introduce a method for patching the seams, cracks, and holes that appear due to the mismatches between edges lying in the boundary planes. Our algorithm follows in the footsteps of marching cubes, but it operates on a different type of spatial structure called a “transition cell”.



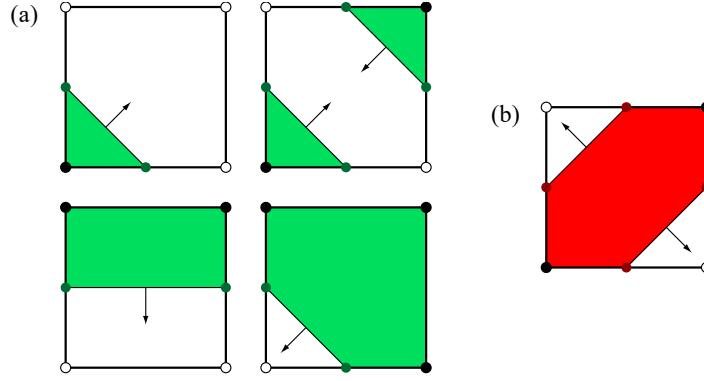
**Figure 1.** Triangle meshes for adjacent voxel terrain blocks are rendered at two different resolutions near a cave entrance. (a) Cracks are visible along the boundary between the blocks, allowing the sky in the background to show through, and a large hole appears where the terrain surface is nearly tangent to the boundary plane between the blocks. (b) A wireframe is overlaid to show the triangles generated by marching cubes.

Transition cells are inserted between ordinary cells where the voxel map sampling frequencies differ by a factor of two. Our method provides the means to efficiently generate triangles that smoothly and seamlessly connect mesh blocks rendered at different levels of detail, producing an artifact-free multiresolution rendition of large voxel-based data sets.



















Previous algorithms [Shu et al. 95, Shekhar et al. 96, Kazhdan et al. 07] either suffer from robustness issues, such as the creation of T-junctions, or require access to arbitrarily large subsets of the voxel data in order to construct a watertight stitching mesh between two different levels of detail, resulting in an open-ended running time. Our method handles all possible cases robustly, runs in a constant per-cell time, and requires access only to the voxel data immediately surrounding each transition cell. This final property is essential for fast retriangulation in a dynamic environment where the voxel data can change at run-time (e.g., when an explosion blows a crater into a chunk of terrain).

## 2 Ambiguous Faces

A well-known defect in the marching cubes algorithm concerns the so-called ambiguous faces arising when two diagonally opposing corners of a cell face have the same inside/outside classification and the other two corners have the inverse classification [Dürst 88]. The literature addressing this issue is extensive, and solutions to the problem vary in complexity. Because simplicity and speed are among our goals, and because it is often possible to easily change the input voxel data to eliminate problem areas in an output mesh, we dispense with the issues of ambiguous faces and topology preservation entirely and choose the simplest solution, which is enforcing a fixed edge polarity on ambiguous faces [Ning and Bloomenthal 93]. For any face of a cell, we only allow the edge placements shown in Figure 2(a) when running any variant of the marching cubes algorithm, and the edge placement shown in Figure 2(b) is never used. This induces the set of 18 equivalence classes shown in Figure 3 that we use in what we call our “modified” marching cubes algorithm. In addition to simplifying the conventional triangulation procedure, the decision to use a fixed edge polarity turns out to be necessary for connecting faces of different resolutions within transition cells. Some of the cases we discuss in this paper would be greatly complicated if we allowed a choice of edge placement on ambiguous faces, and proper triangulations for some other cases would become impossible.



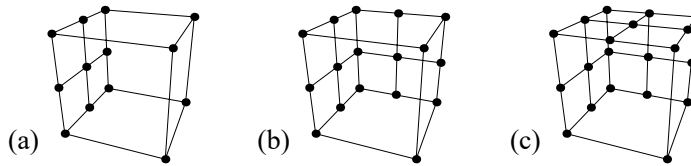
**Figure 2.** (a) These are the only four nontrivial edge configurations allowed on a cell face modulo rotation. (b) This edge configuration is not allowed for the ambiguous case. Corners with solid dots are classified as inside, and corners with open dots are classified as outside. The filled region represents solid space, and the arrows represent the outward surface normal direction along the edges.

 #0 (2)	 #1 (16)	 #2 (12)
 #3 (24)	 #4 (8)	 #5 (48)
 #6 (24)	 #7 (8)	 #8 (24)
 #9 (6)	 #10 (2)	 #11 (6)
 #12 (12)	 #13 (12)	 #14 (8)
 #15 (12)	 #16 (24)	 #17 (8)

**Figure 3.** These are the 18 equivalence classes identified by our modified marching cubes algorithm. The number in the lower-left corner of each entry is the class index, and the number in the lower-right corner is the number of cases belonging to the equivalence class out of the 256 total cases. A black dot indicates a corner that is inside the solid volume, and corners without a dot are outside. Green triangles are front-facing, and red triangles are back-facing.

### 3 Transition Cells

Let us consider two adjacent blocks of voxel data such that one block is sampled at half the resolution of the other block. We call a cell inside the half-resolution block that lies along the border with the full-resolution block a *transition cell*, and we see that a triangulation (compatible with the marching cubes algorithm) for any such cell must account for a total of 13 sample values lying on its boundary. As shown in Figure 4(a), nine of these samples come from the full-resolution data, and the remaining four come from the half-resolution data. Classifying each of these sample values as either inside or outside the solid space produces  $2^{13} = 8192$  possible cases to be triangulated. This is quite large compared to the 256 cases arising in a marching cubes implementation for a single resolution, but there is no reason to believe that an analogous algorithm using a 13-bit lookup table would not produce a satisfactory and robust result.



**Figure 4.** The points shown on the boundaries of these cubes illustrate the sample positions for a single cell in a half-resolution block when it is bordered by a full-resolution block on (a) one side, (b) two sides, and (c) three sides.

However, the half-resolution block can be bordered on more than one side by full-resolution blocks. If we further consider a single cell in the half-resolution block that is bordered by full-resolution cells on two adjacent faces, then a triangulation of that cell must account for the 17 sample values shown in Figure 4(b), giving rise to  $2^{17} = 131,072$  distinct cases. Finally, if a half-resolution cell is bordered on three adjacent faces by full-resolution cells, a triangulation must account for the 20 sample values shown in Figure 4(c), resulting in a whopping  $2^{20} = 1,048,576$  distinct cases. To implement an algorithm that would connect the full-resolution volume to the half-resolution volume by triangulating all possible cases arising within individual transition cells along the border between them, we would have to account for all  $2^{13} + 2^{17} + 2^{20} = 1,187,840$  configurations.

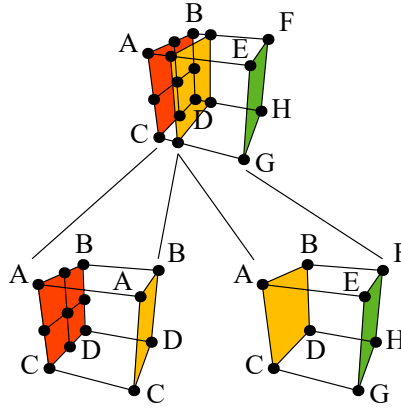
From an engineering perspective, determining the set of topological equivalence classes for the nearly 1.2 million cases and then generating a lookup table containing triangulation data for all of those cases is a monumentally tedious job. Furthermore, even if such a lookup table could be easily constructed, it would require dozens of megabytes of storage space. This size by itself could be prohibitively large on some platforms, but a more important issue is the high number of data cache misses that an algorithm using this lookup table would experience on any platform, severely affecting performance. In light of these disadvantages to the implementation of a direct analog of marching cubes for transition cells, we are compelled to search for some way of simplifying the fundamental problem at hand.

Ideally, we would like the number of distinct triangulation cases for a transition cell to be on the same order of magnitude as the number of cases arising in the conventional marching cubes algorithm, and we would prefer that the number of possible cases be the same for all transition cells. Both of these properties can be achieved by dividing each transition cell into two smaller

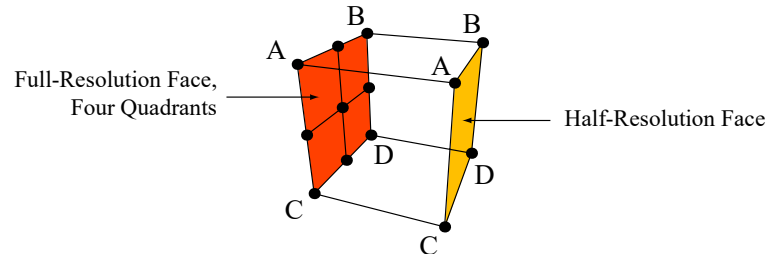


cells in the manner illustrated by Figure 5. The left cell is triangulated using sample values only from the face bordering the full-resolution block, and the four corner values labeled A, B, C, and D in the figure are duplicated on the opposite face of the cell. The transition from full resolution to half resolution takes place entirely inside this cell, for which we now have a very manageable nine distinct sample values to consider. The right cell is triangulated in the conventional manner using only the eight sample values from the half-resolution data.

We call cells (at any resolution) that are influenced by eight corner voxels and triangulated using the marching cubes algorithm *regular* cells. We redefine the meaning of a *transition* cell to be one of the form shown in Figure 6, which is influenced by nine voxels lying on a single face. In a transition cell, we call the face on which nine sample values appear the *full-resolution* face. The face opposite the full-resolution face is called the *half-resolution* face, and the sample values at its four corners are identical to those of the corresponding corners of the full-resolution face. The remaining four faces are called *lateral* faces.



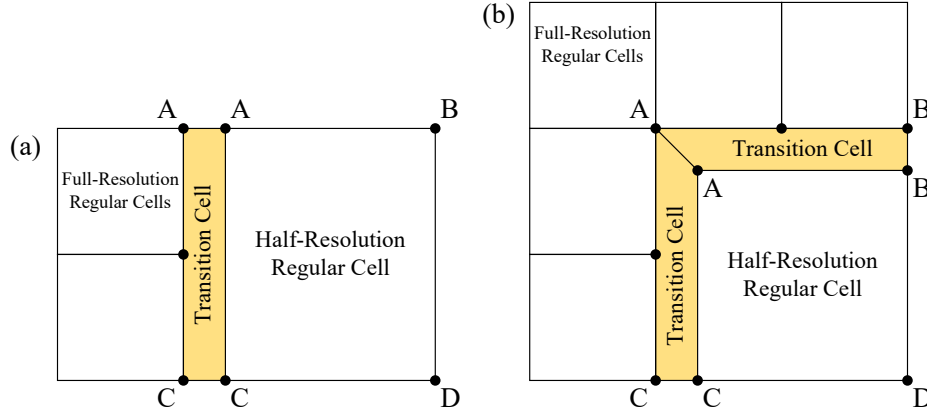
**Figure 5.** A transition cell is divided into two parts along a plane parallel to the boundary face between full-resolution and half-resolution blocks. Our new algorithm triangulates the left part using sample values only appearing on the full-resolution face. The right part is triangulated with the conventional marching cubes algorithm.



**Figure 6.** This cube represents the topological layout of a transition cell. It has a full-resolution face (shown in red) that is divided into four quadrants corresponding to four full-resolution regular cells. Opposite the full-resolution face is the half-resolution face (shown in orange), and its corner voxel sample values are the same as the corresponding corners of the full-resolution face. The remaining four faces of the cube are the lateral faces.

The width of a transition cell (i.e., the distance between the full-resolution and half-resolution faces) is a parameter that can be freely adjusted at the global level. The triangles that we construct inside a transition cell to connect voxel data of different resolutions are simply scaled to the proper width. The transition cell width is normally chosen to be a large fraction of the size of a full-resolution cell. It is possible to use a width of zero and still produce results that seamlessly stitch multiresolution meshes together, but this width leads to severe shading problems.

Transition cells always have the same configuration with respect to the number and location of their voxel samples, and this is true even when a half-resolution block is bordered by full-resolution blocks on two or three sides. Figure 7(a) shows a two-dimensional slice of a transition cell connecting full-resolution regular cells to a half-resolution regular cell on one face, and Figure 7(b) shows two transition cells connecting full-resolution regular cells to a half-resolution regular cell on two adjacent faces. In the case of a half-resolution regular cell connected full-resolution regular cells on three adjacent faces, the configuration is an analogous extension of that shown for two faces. The transition cells like those shown in Figure 7(b) are no longer boxes whose edges always meet at right angles, but they are still topologically equivalent to the cube shown in Figure 6.



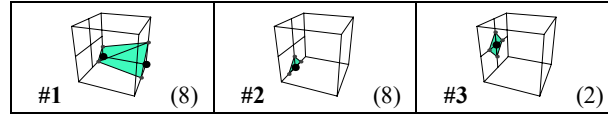
**Figure 7.** These are two-dimensional slices showing what is done in the cases in which a half-resolution cell is bordered by full-resolution cells on (a) one side and (b) two sides. All regular cells are triangulated using the conventional marching cubes algorithm, and all transition cells have the same data point configuration.

The transition cell triangulation problem has now been reduced to one for which there are  $2^9 = 512$  possible cases to handle. It is natural for us to attempt to identify a set of equivalence classes whose members are related through an orthogonal transform in the same spirit as the marching cubes algorithm. Since the nine voxel sample values influencing a transition cell lie in a square region of a plane, we determine the set of equivalence classes by considering the action of the dihedral group  $D_8$  on the sample locations of the individual cases. Given a particular case  $x$ , the other members of the equivalence class containing  $x$  are precisely the cases belonging to the orbit of  $x$  under the group action. The size of the equivalence class containing  $x$  is given by the index of the stabilizer subgroup of  $x$  in  $D_8$ . (Consequently, the size of each equivalence class must divide 8.) Because  $D_8$  includes reflections, the equivalence relation induced by the group action equates cases that are mirror images of each other.

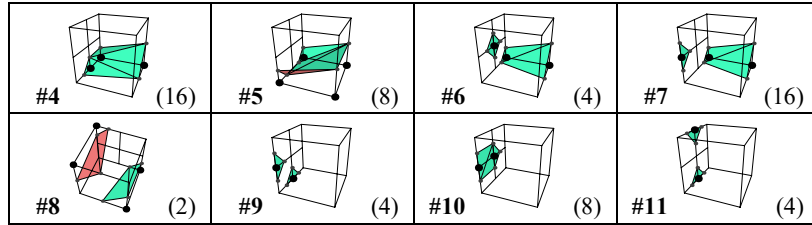
If a case  $x$  does not have an ambiguous half-resolution face and does not have any ambiguous quadrants on its full-resolution face, then we also add an inversion operation to the group acting on

$x$  that reverses the inside/outside state of all nine sample values. For these cases, the group inducing our equivalence classes is isomorphic to  $D_8 \times \mathbb{Z}_2$ , and the net effect is that the sizes of the equivalence classes are doubled.

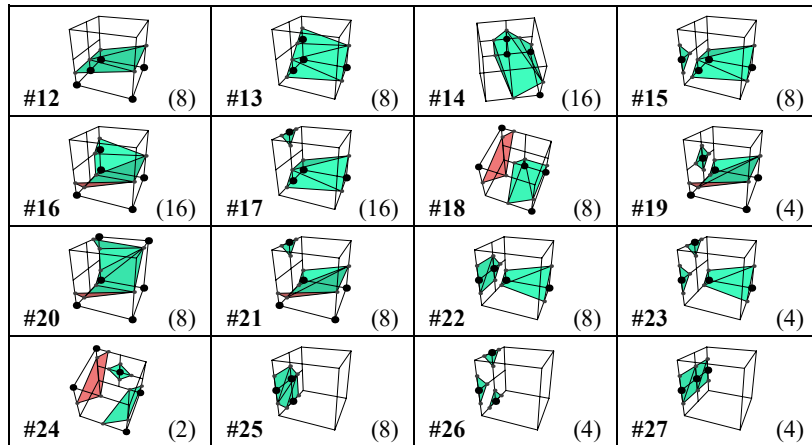
Observing the orbits of each of the 512 cases under the action of the group  $D_8$  or  $D_8 \times \mathbb{Z}_2$ , as appropriate, yields exactly 73 distinct equivalence classes. One equivalence class is the trivial class containing the two cases for which the inside/outside state of all nine sample values is the same. The 72 nontrivial classes are enumerated in Figures 8 through 13, where all of the cases appearing in a particular table are related in some way. For each table entry, the number in the lower-left corner is the class index (that we have assigned somewhat arbitrarily), and the number in the lower-right corner is the number of cases belonging to the equivalence class. A black dot indicates a corner that is inside solid space, and corners without a dot are outside. Green triangles are front-facing, and red triangles are back-facing.



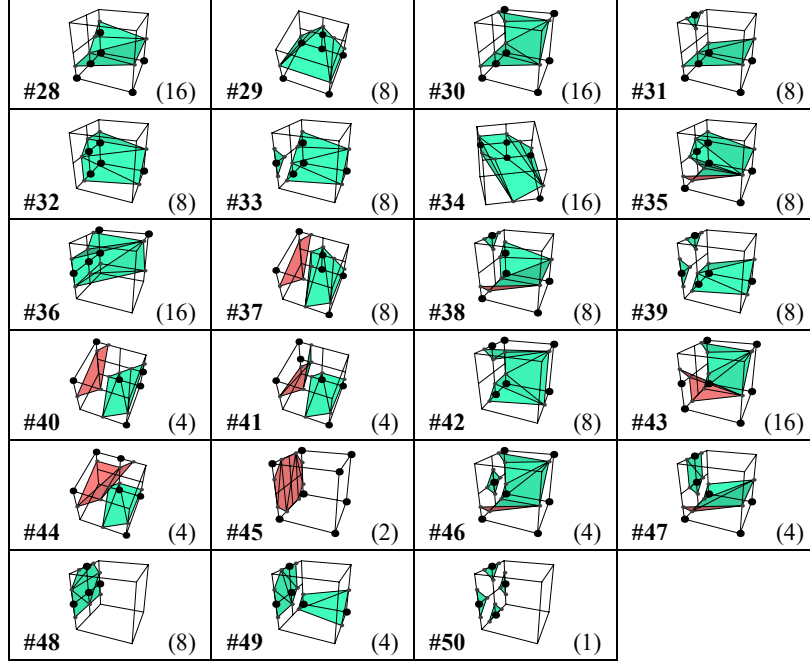
**Figure 8.** These are the 3 transition cell equivalence classes for which exactly one sample value is inside solid space. These classes represent 18 of the 512 distinct cases.



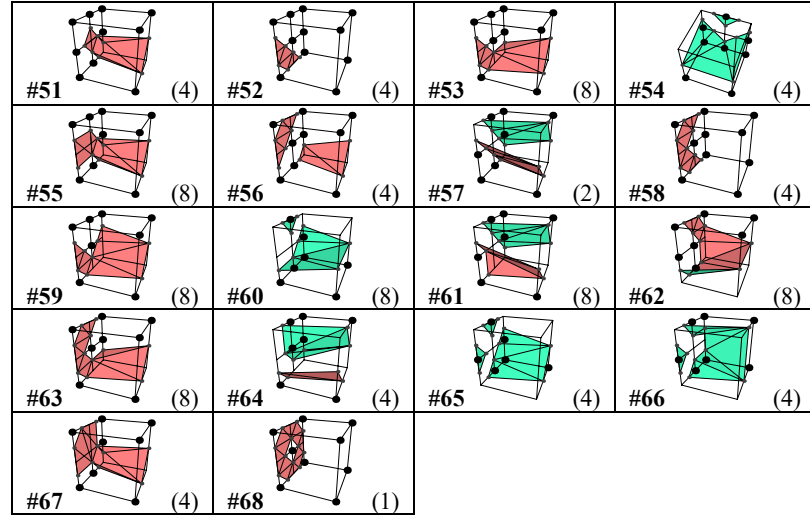
**Figure 9.** These are the 8 transition cell equivalence classes for which exactly two sample values are inside solid space. These classes represent 62 of the 512 distinct cases.



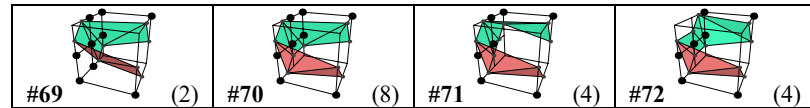
**Figure 10.** These are the 16 transition cell equivalence classes for which exactly three sample values are inside solid space. These classes represent 130 of the 512 distinct cases.



**Figure 11.** These are the 23 transition cell equivalence classes for which exactly four sample values are inside solid space. These classes represent 187 of the 512 distinct cases.



**Figure 12.** These are the 18 transition cell equivalence classes corresponding to the inverses of classes appearing in Figures 9 through 11 that have at least one ambiguous quadrant on the full-resolution face. These classes represent 95 of the 512 distinct cases.



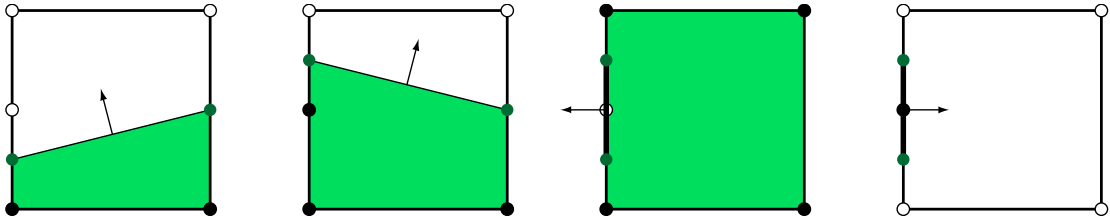
**Figure 13.** These are the 4 transition cell equivalence classes corresponding to the inverses of classes appearing in Figures 9 through 11 for which the half-resolution face is ambiguous, but no full-resolution quadrants are ambiguous. These classes represent 18 of the 512 distinct cases.

Table 1 lists the 51 transition cell equivalence classes having four or fewer sample values classified as inside solid space. For equivalence classes that do not include geometric inverses due to the presence of an ambiguous face, the inverse equivalence class is listed, and this accounts for the remaining 22 classes. For the representative case  $x$  shown for each equivalence class in Figures 8 through 13, Table 1 also identifies the stabilizer subgroup  $H = G_x$ , where  $G \cong D_8 \times \mathbb{Z}_2$  for classes that include geometric inverses, and  $G = D_8$  for those that do not. The stabilizer subgroup is expressed in terms of the generators  $r$  and  $f$  of the group  $D_8$ , where we have chosen  $r$  to be the 90-degree counterclockwise rotation and  $f$  to be the flip about the horizontal axis. Equivalence classes containing cases that exhibit no symmetry are the largest because the stabilizers are simply the subgroup  $\{e\}$  containing only the identity element  $e$ .

The edge placements that we use for transition cells follow a consistent set of rules. Clearly, we must match edges occurring on the regular cells adjacent to a transition cell on both the full-resolution and half-resolution faces in order to ensure a continuous mesh. For any nontrivial quadrant of the full-resolution face, open edges must be placed within the quadrant at the locations specified by Figure 2. The same is true for the half-resolution face when it is nontrivial.

What remains are the lateral faces of a transition cell. Any open edges placed on a lateral face must be matched by an edge having the same endpoints on the coincident lateral face of an adjacent transition cell, and this edge must have the opposite winding direction. Figure 14 shows the four possible nontrivial configurations of a lateral face in terms of the inside/outside state of the three voxel sample values affecting it. In the two configurations for which the sample states alternate, either inside-outside-inside or outside-inside-outside, a mesh edge is placed on the boundary edge between the lateral face and the full-resolution face, and it is thus not connected to the half-resolution face.

Unlike typical triangulations of regular cells, many of the triangulations for transition cells contain internal edges that lie entirely on the full-resolution face, and they can span multiple quadrants. The placement of these edges on the boundary of the transition cell cannot be avoided without introducing additional vertices. In most cases, these edges connect two triangles that both lie in the full-resolution face and form part of a cap for features in the full-resolution voxel map that exceed the sampling frequency in the half-resolution voxel map. The four classes shown in Figure 13 are the only exceptions to this pattern, and they each contain internal edges connecting triangles lying in the full-resolution face with triangles that span the width of the transition cell. These triangulations form a saddle-like surface in the local region of the overall mesh.



**Figure 14.** These are the only four nontrivial edge configurations allowed on a lateral face of a transition cell modulo a vertical flip. The left edge represents the full-resolution face, and the right edge represents the half-resolution face. Sample locations with solid dots are classified as inside, and sample locations with open dots are classified as outside. The green region represents solid space, and the arrows represent the outward surface normal direction along the edges. In the two rightmost configurations, the edge lies on the boundary between the lateral face and the full-resolution face.

Class	Inverse class	Stabilizer subgroup	Class size
#0		$D_8$	2
#1		$\langle r^3 f \rangle$	8
#2		$\langle r^2 f \rangle$	8
#3		$D_8$	2
#4		$\{e\}$	16
#5		$\langle r^2 f \rangle$	8
#6	#51	$\langle r^3 f \rangle$	4
#7		$\{e\}$	16
#8	#69	$\langle r^2, rf \rangle$	2
#9	#52	$\langle rf \rangle$	4
#10		$\langle f \rangle$	8
#11		$\langle r^2 f \rangle$	4
#12		$\langle r^2 f \rangle$	8
#13		$\langle r^3 f \rangle$	8
#14		$\{e\}$	16
#15	#53	$\{e\}$	8
#16		$\{e\}$	16
#17		$\{e\}$	16
#18	#70	$\{e\}$	8
#19	#54	$\langle r^2 f \rangle$	4
#20		$\langle r^3 f \rangle$	8
#21		$\langle r^2 f \rangle$	8
#22	#55	$\{e\}$	8
#23	#56	$\langle r^3 f \rangle$	4
#24	#57	$\langle r^2, rf \rangle$	2
#25		$\langle rf \rangle$	8

Class	Inverse class	Stabilizer subgroup	Class size
#26	#58	$\langle f \rangle$	4
#27		$\langle r^2, f \rangle$	4
#28		$\{e\}$	16
#29		$\langle r^2 f \rangle$	8
#30		$\{e\}$	16
#31		$\langle r^2 f \rangle$	8
#32		$\langle r^3 f \rangle$	8
#33	#59	$\{e\}$	8
#34		$\{e\}$	16
#35	#60	$\{e\}$	8
#36		$\{e\}$	16
#37	#61	$\{e\}$	8
#38	#62	$\{e\}$	8
#39	#63	$\{e\}$	8
#40	#71	$\langle r^3 f \rangle$	4
#41	#64	$\langle rf \rangle$	4
#42		$\langle f \rangle$	8
#43		$\{e\}$	16
#44	#72	$\langle r^2 \rangle$	4
#45		$D_8$	2
#46	#65	$\langle r^3 f \rangle$	4
#47	#66	$\langle r^2 f \rangle$	4
#48		$\langle r^2 f \rangle$	8
#49	#67	$\langle r^3 f \rangle$	4
#50	#68	$D_8$	1

**Table 1.** This table lists all 73 transition cell equivalence classes and identifies the stabilizer subgroup  $H$  for the representative element previously shown for each class. For classes in Figures 8 through 11 that do not include geometric inverses, the associated inverse equivalence class from Figure 12 or 13 is indicated. The size of each equivalence class is given by  $|G : H|$ , where  $G \cong D_8 \times \mathbb{Z}_2$  for classes that include geometric inverses, and  $G = D_8$  for those that do not.

## 4 Rendering

In our implementation, a large voxel data set is composed of a hierarchy of blocks measuring  $16 \times 16 \times 16$  cells, and these blocks are arranged in an octree for which each level corresponds to a separate mesh resolution (for details, see Lengyel [2010]). Each block not of the highest level of detail neatly contains exactly eight blocks of the next-highest resolution. The octree is traversed when the surface is rendered, and any block not intersecting the view frustum is culled along with its entire subtree. When a block is visited in the octree and determined to be visible to the camera, its projected size in the viewport is calculated. If the size falls below a threshold value, then that block is rendered, and its subtree is skipped. This process efficiently selects the proper levels of detail for the entire surface, and the threshold value can be adjusted to control at what distances LOD transitions occur.

The triangles belonging to transition cells should only be rendered between adjacent blocks of differing resolutions. Each block selected for rendering needs to be aware of which of its siblings in the octree have also been selected for rendering, and the transition cells occurring along the boundaries with those blocks must be disabled. Thus, for all but the highest-resolution blocks, the mesh generated for a block is composed of up to seven distinct parts: the primary mesh for the block and a transition mesh for each of the six faces of the block on which nontrivial transition cells occur. The primary mesh is always rendered, but the transition meshes are only rendered under specific conditions.

Regular cells that occur along the boundary of a low-resolution block can be rendered in one of two states. If an adjacent block is rendered at the same level of detail, then the regular cells along the boundary are rendered normally with no transition cells in between. If adjacent blocks are rendered at different levels of detail, however, then regular cells in the block for the lower level of detail must be scaled in one or more directions to make space for transition cells as illustrated in Figure 7. In order to avoid extra computation in the vertex program, we store two positions for each vertex belonging to regular cells on the boundary of a low-resolution block, a primary position used when transition cells are not rendered, and a secondary position used when transition cells are rendered.

Secondary vertex positions can be calculated by linearly transforming positions inside boundary cells so that the full-size cell is scaled to a smaller size that allows space for between one and three transition cells, as necessary, depending on the location with respect to the edges and corners of the entire block. This can be accomplished by computing offsets  $(\Delta x, \Delta y, \Delta z)$  for the coordinates  $(x, y, z)$  in any boundary cell using the formula

$$\Delta x = \begin{cases} (1 - 2^{-k}x)w(k), & \text{if } x < 2^k; \\ 0, & \text{if } 2^k \leq x \leq 2^k(s-1); \\ (2^k - 1 - x)w(k), & \text{if } x > 2^k(s-1), \end{cases} \quad (1)$$

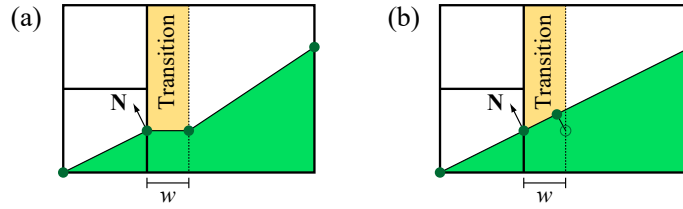
where  $k$  is the zero-based LOD index, and  $s$  is the size of a block in one dimension when measured in cells (e.g., if a block measures  $16 \times 16 \times 16$  cells, then  $s = 16$ ). The function  $w(k)$  specifies the width of transition cells for LOD index  $k$ , and it is defined as  $w(k) = 2^{k-2}$  in our implementation. Formulas for  $\Delta y$  and  $\Delta z$  are similar to Equation (1).

Along a block face on which a transition is occurring between levels of detail, we apply Equation (1) to all vertices generated for the regular cells and to any vertices generated on the half-resolution faces of transition cells. Vertices generated on the full-resolution faces of transition cells are not moved. Adding the offsets directly to the original vertex positions has the effect of creating

a flattened strip of triangles in the transition region as shown in Figure 15(a), and it deforms the surface in the regular cell such that unwanted concavities can be created. These problems can be eliminated by projecting the offset vector for a vertex onto the tangent plane passing through the vertex's original position to obtain the new vertex position  $(x', y', z')$  using the formula

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \end{bmatrix} + \begin{bmatrix} 1 - N_x^2 & -N_x N_y & -N_x N_z \\ -N_x N_y & 1 - N_y^2 & -N_y N_z \\ -N_x N_z & -N_y N_z & 1 - N_z^2 \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta y \\ \Delta z \end{bmatrix}, \quad (2)$$

where  $\mathbf{N}$  is the unit-length vertex normal. The effect of the projection given by Equation (2) is shown in Figure 15(b).

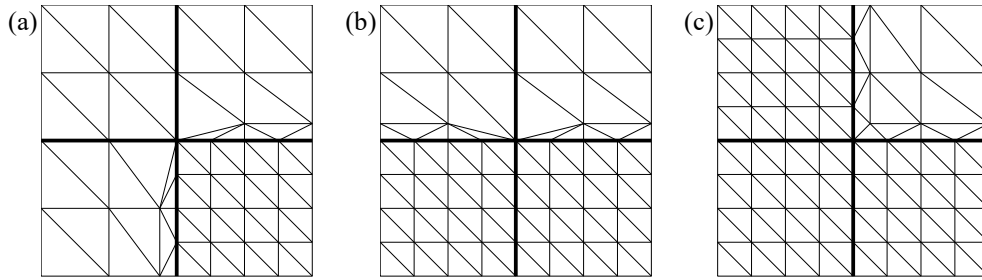


**Figure 15.** (a) The linear transformation given by Equation (1) is applied to vertices in the low-resolution cell to make space for a transition cell, but this has the unwanted effects of creating a flattened region and a concavity. (b) Using Equation (2) to project the difference in vertex positions onto the tangent plane with respect to the vertex normal  $\mathbf{N}$  eliminates these problems.

In addition to the primary and secondary positions, we also store a three-component value for each vertex that indicates which faces of the block the vertex is near. The value stored for each component corresponds to one of three states possible for each coordinate of the primary vertex position: near the positive face, near the negative face, or near neither of those faces. When a block is selected for rendering, six boolean values are calculated to indicate whether the adjacent block at the same level of detail is also being rendered, and these values are made available to the vertex program. If the vertex program determines that none of the blocks are being rendered at the same level of detail adjacent to faces for which a vertex is near, then it selects the secondary vertex position because all transition cells affecting the vertex are being rendered. If any of the adjacent blocks adjacent to faces for which a vertex is near are being rendered at the same level of detail as the block owning the vertex, then the primary position is selected.

This procedure for selecting between the primary and secondary vertex positions causes the primary vertex position to be used at corners where blocks meet and more than one block is rendered at the lower level of detail. Figure 16 shows examples in the case where four blocks meet and are rendered at different levels of detail. In images (a) and (b), the primary vertex position is always selected at the corner location because not all adjacent blocks are being rendered at the higher resolution. In image (c), the secondary vertex position is selected since all three adjacent blocks are rendered at the higher resolution. The pinching that occurs in images (a) and (b) prevents edge mismatches on the lateral faces of transition cells when they are adjacent to low-resolution regular cells.





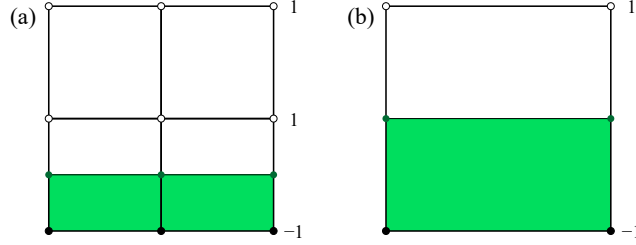
**Figure 16.** These are the three possible transitions that can take place at the corner where four blocks meet and are not all rendered at the same level of detail. All of the transition cells shown here use equivalence class #12 from Figure 10. (a) Three blocks are rendered at low resolution, and one is rendered at high resolution. (b) Two adjacent blocks are rendered at low resolution, and the other two are rendered at high resolution. (c) One block is rendered at low resolution, and three are rendered at high resolution.

## 5 Multiresolution Meshes

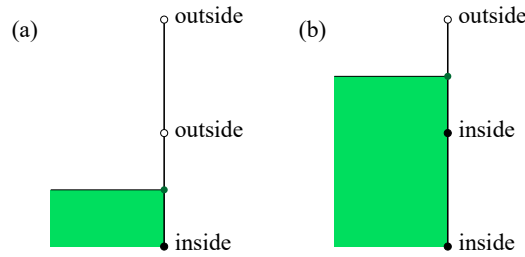
When triangulating a lower-resolution sampling of a voxel map, running the same marching cubes algorithm used for the highest level of detail tends to produce meshes that are smoothed out and a poor match for their higher-resolution counterparts. The result is a very noticeable shift in the surface position whenever there is a change in the level of detail rendered for part of a surface. As illustrated in Figure 17, this effect can be particularly bad even for something like flat ground because the surface can shift by as much as half of a cell width. Any structures placed on the surface would suddenly appear sunken into the ground when the detail level changed, which is not acceptable.

To eliminate the surface shifting problem, we modify the way in which vertex coordinates are chosen when running marching cubes over low-resolution cells so that vertices belonging to low-resolution meshes coincide exactly with vertices belonging to the highest-resolution mesh. This is accomplished by first examining the voxel sample at the midpoint of each active edge of a low-resolution cell. As shown in Figure 18, if a transition from inside to outside occurs on an edge of a cell, then such a transition occurs inside one of the halves of the edge at double the resolution, but it cannot occur inside both halves. So we choose the sub-edge for which the transition does occur and continue recursively until the highest resolution is reached. We know a vertex is placed on the edge to which we ultimately converge in the highest-resolution mesh, so we calculate the vertex position for the low-resolution mesh by finding the location of the isovalue between the endpoints of the highest-resolution sub-edge. This process is deterministic even when multiple inside/outside transitions occur along an edge at one quarter or lower resolution, and it guarantees that all vertices in lower levels of detail coincide exactly with vertices of the highest-resolution mesh. Note that the search described here only needs to be performed for vertices lying in the interior of an edge, and not for vertices placed at cell corners.

The vertices in the high-resolution mesh that lie in the interior of low-resolution cells can still be far enough away from the low-resolution triangle surface to cause a perceptible pop when the level of detail abruptly changes. One way to alleviate this problem is to precalculate a secondary position for each vertex in the high-resolution mesh that is equal to the projection of the primary position onto the surface in the triangulation of the low-resolution cell. For some distance before the point at which the detail level changes, a smooth interpolation from the primary position to the secondary position can be employed to morph the high-resolution mesh into a surface matching the low-resolution mesh. A secondary normal vector can also be precalculated in order to eliminate shading pops due to an abrupt change in the Lambertian factor.



**Figure 17.** These are side views showing where ground triangles would be placed for cells of two different resolutions occupying the same space when the voxel values transition from  $-1$  to  $1$  along the vertical edges. In both (a) high-resolution cells and (b) low-resolution cells, the ground vertices are placed halfway between the voxel map sample locations by the marching cubes algorithm. The misalignment of these halfway points causes an undesirable shift in the surface position when detail levels change.



**Figure 18.** If a transition from inside to outside occurs along an edge in a half-resolution sampling of the voxel map, represented by the top and bottom sample locations, then such a transition can only occur in one half of the edge at double the resolution where the sample at the midpoint is considered. (a) The transition occurs in the bottom half of the edge because the center sample's inside/outside state does not match the bottom sample's state. (b) The transition occurs in the top half because the center sample's state does not match the top sample's state.

Although not particularly important for the highest level of detail, the specific placement of internal edges can also make a significant difference when triangulating cells for lower levels of detail. For most of the equivalence classes arising in the conventional marching cubes algorithm, there are multiple triangulations that could be legally generated for the same set of vertices. The triangulation for each class is composed of one to four connected components that are topologically equivalent to convex polygons having between three and seven vertices, inclusive. For components having three vertices, we have no choice but to generate a single triangle. However, for components having a larger number of vertices, we do have some choice in the matter, and it may not be the case that some arbitrary default triangulation is the best option because it is a poor match for higher-resolution voxel data.

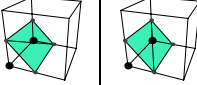
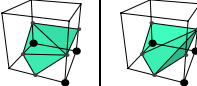
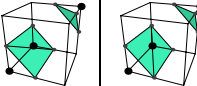
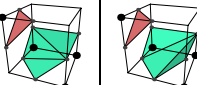
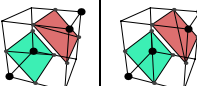
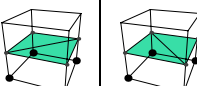
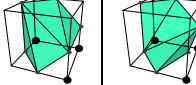
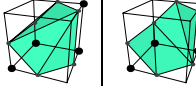
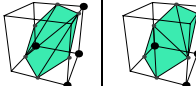
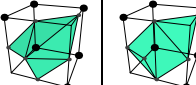
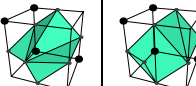
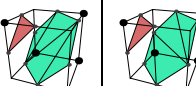
Even though they don't participate in the selection of the equivalence class to which a low-resolution cell belongs, we still have access to the highest-resolution voxel samples inside each cell when generating a low-resolution mesh. Each of these voxel samples represents a signed distance to the isosurface, so our goal is to choose the triangulation that minimizes the error between these values and the actual distances to the isosurface created by the triangulation. Since the interior edges are the only ones that can be changed, we only consider voxels that lie in the interior of a cell, and not those lying on a cell's boundary.

In the case that a triangulation component has four vertices, there are obviously two possible ways to connect vertices to form a triangulation. As the number of vertices increases, however, the number of possible triangulations grows very rapidly. In general, the number of ways to triangulate a convex polygon having  $n$  vertices is given by the Catalan number  $C_{n-2}$ , where

$$C_n = \frac{1}{n+1} \binom{2n}{n}.$$

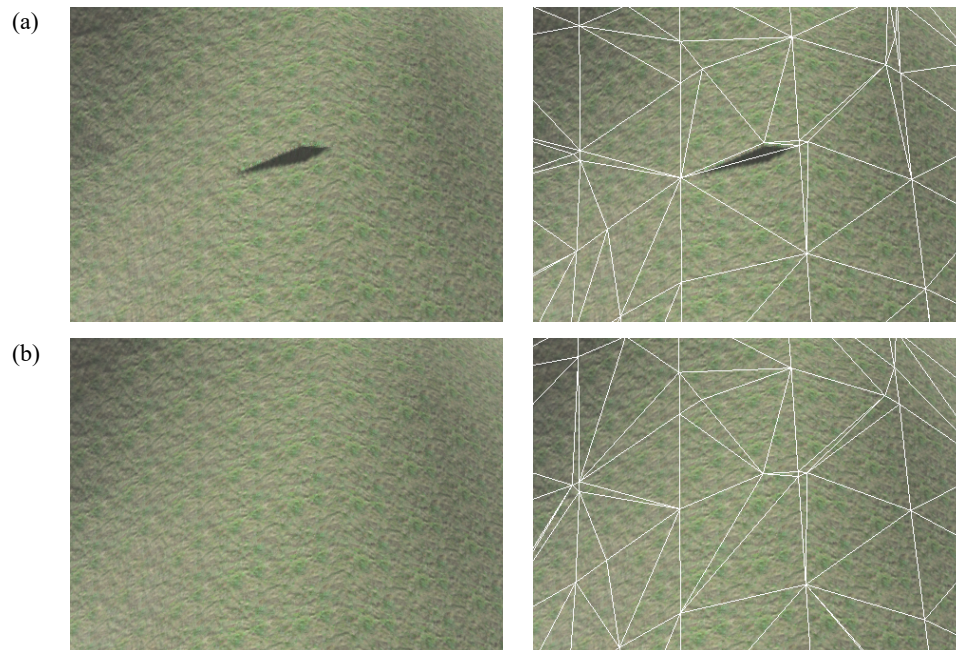
The values of  $C_n$  for  $n = 1, 2, 3, 4, 5$  are 1, 2, 5, 14, and 42, corresponding to the number of possible triangulations of polygons having 3, 4, 5, 6, and 7 vertices.

There are two equivalence classes for which 5 triangulations are possible, five classes for which 14 triangulations are possible, and one class for which 42 triangulations are possible. Searching for the best triangulation out of all the possibilities in these cases would have a negative impact on performance, so we instead make a compromise in our implementation and limit the number of triangulations tested per cell to two. For components having five or more vertices, we choose the two triangulations to be as dissimilar as possible in terms of their principal directions of curvature. Table 2 lists the pairs of triangulations that we test for the 12 equivalence classes having multiple possibilities.

Class Index	Triangulations		Number Possible
#3			$C_2 = 2$
#5			$C_3 = 5$
#6			$C_2 = 2$
#8			$C_3 = 5$
#9			$C_2^2 = 4$
#11			$C_2 = 2$
#12			$C_4 = 14$
#13			$C_4 = 14$
#14			$C_4 = 14$
#15			$C_4 = 14$
#16			$C_5 = 42$
#17			$C_4 = 14$

**Table 2.** These are the 12 equivalence classes in the modified marching cubes algorithm, identified by class indexes corresponding to Figure 3, for which multiple triangulations are possible. The two triangulations shown are the only ones that we test for a best match to the high-resolution voxel data, and they are chosen to be as dissimilar as possible in terms of their principal directions of curvature. The last column shows how many total triangulations are possible for each equivalence class.

Because lower levels of detail are typically rendered only when the camera is somewhat far away, the triangulation assigned to any particular cell only makes a subtle directly-observable difference in the mesh. However, there are two indirect ways in which the ability to choose between triangulations does make a more prominent difference. First, the magnitude of the pop that occurs when detail levels change (or the distance by which vertices move if morphing is used) is reduced when the low-resolution triangulation is a better match for the high-resolution surface. Second, as shown in Figure 19(a), a very noticeable artifact can appear when a poor choice of triangulation is made because it can create a concavity that produces an unwanted shadow when aligned in an unfavorable manner with the light source. Choosing the better triangulation from Table 2 for each cell helps eliminate such artifacts by creating a smoother low-resolution mesh, as shown in Figure 19(b).

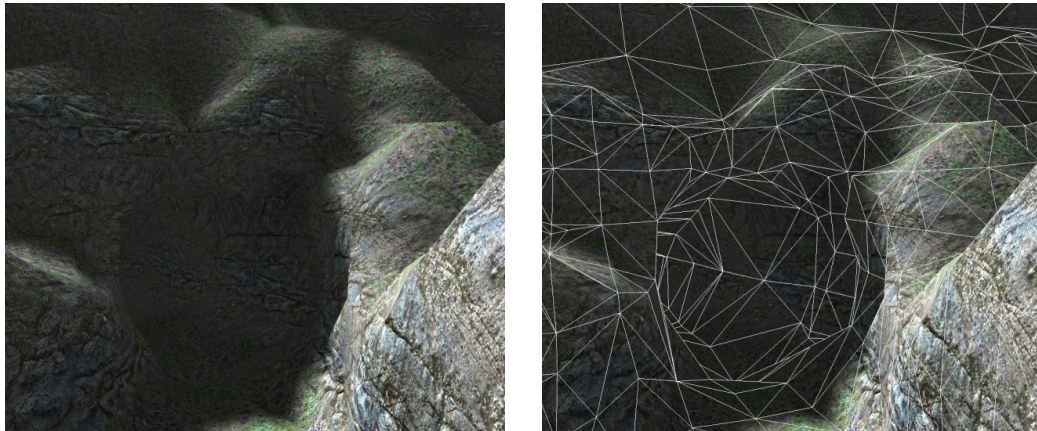


**Figure 19.** (a) A prominent shadowing artifact appears in a low-resolution mesh, shown with and without wireframe, because a poor choice of triangulation creates a concavity that blocks light. (b) Choosing the better triangulation from Table 2 in order to match the high-resolution curvature eliminates the artifact.

## 6 Results

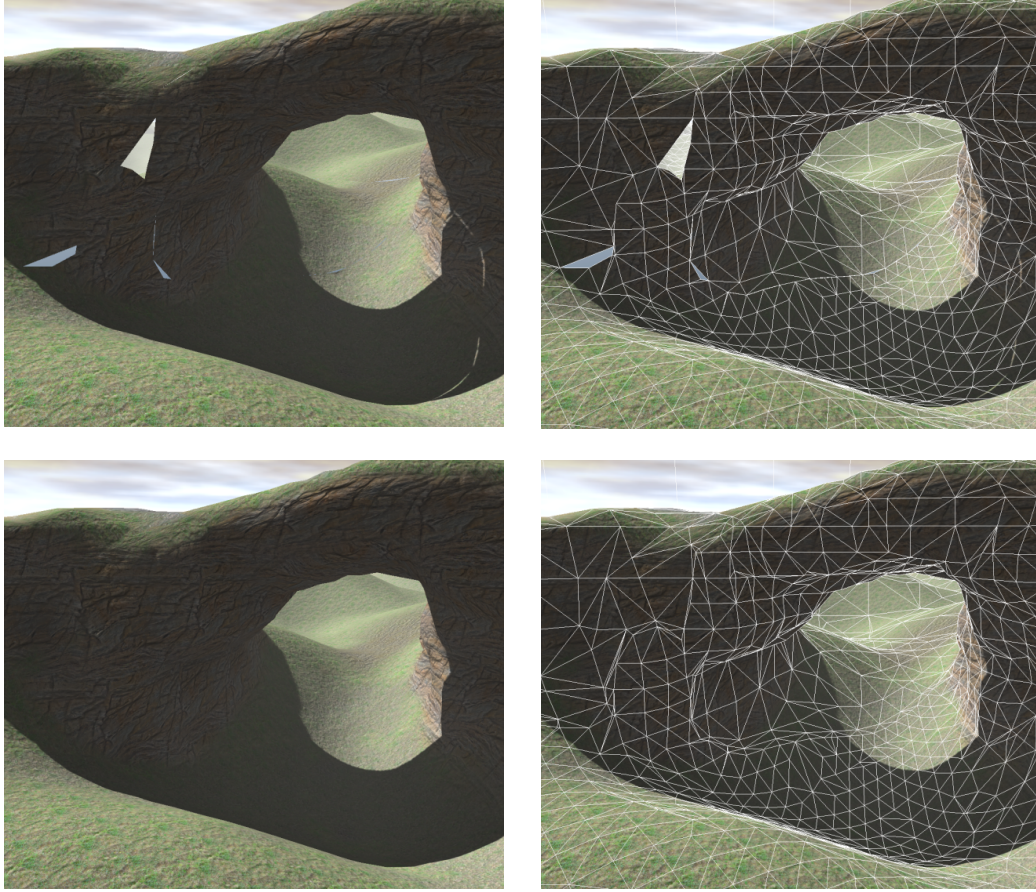
Adding transition cells to the terrain shown in Figure 1 and following the rendering procedure described in Section 4 produces the result shown in Figure 20. Another example is shown in Figure 21. The triangles generated by transition cells stitch together all the cracks, seams, and holes arising from mismatched edges between blocks rendered at differing resolutions. Furthermore, these patch triangles are created in such a way that a smooth surface is maintained, avoiding potential texturing and shading artifacts.

The motivation for constructing multiresolution meshes from voxel data in the first place is to increase rendering performance. When rendering blocks of triangulated voxels with multiple levels of detail, the bulk of the savings in GPU processing time happens in the vertex shader, but there is also a slight performance increase in fragment processing due to the fact that larger triangles are rendered for distant parts of the scene. Figure 22 shows a voxel-based 4 km<sup>2</sup> island terrain rendered with and without the use of multiresolution meshes and transition cells. In the case that the terrain is rendered with three levels of detail, the rendering time is roughly five to eight milliseconds shorter compared to the same scene rendered with only the highest-resolution meshes across a range of DirectX 10-class GPUs on which we tested. In this particular scene, 191,500 triangles are rendered when only the highest-resolution meshes are used. This number is reduced to 63,800 triangles when multiresolution meshes are used, and 3,200 of those triangles are generated by transition cells.

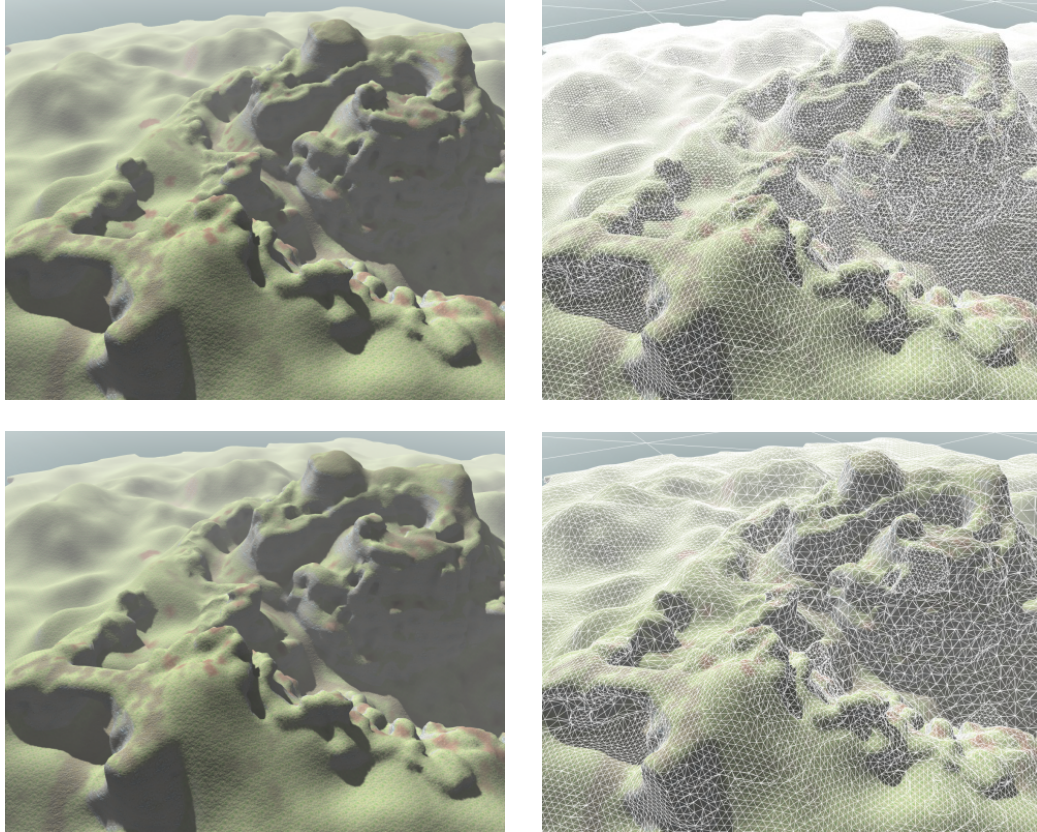


**Figure 20.** The artifacts shown in Figure 1 are eliminated by the addition of transition cells, which can be identified in the wireframe overlay.





**Figure 21.** The detail level changes at the block boundary near the left side of this arch, shown with and without a wireframe overlay. No transition cells are rendered in the top row, and large artifacts can be seen both directly and in the shadow cast by the terrain mesh on the right side of the image. In the bottom row, transition cells are rendered to stitch together the different levels of detail, and all artifacts are eliminated.



**Figure 22.** A large voxel-based terrain is rendered in a  $1280 \times 1024$  viewport to compare performance with and without multiresolution meshes. In the top row, the highest-resolution meshes are used everywhere. In the bottom row, three levels of detail and transition cells are used, and the rendering time is reduced by five to eight milliseconds. Only one-third the number of triangles are rendered in the multiresolution case, and five percent of those are due to transition cells.

## References

- [Dürst 88] M. J. Dürst. Letters: additional reference to marching cubes. *Computer Graphics* 22:2, pp. 72–73, 1988.
- [Kazhdan et al. 07] Michael Kazhdan, Allison Klein, Ketan Dalal, and Hugues Hoppe. “Unconstrained Isosurface Extraction on Arbitrary Octrees.” In *Proceedings of the Fifth Eurographics Symposium on Geometry Processing*, pp. 125–133, 2007.
- [Lengyel 2010] Eric Lengyel. “Voxel-Based Terrain for Real-Time Virtual Simulations.” PhD diss., University of California at Davis, 2010.
- [Lorensen and Cline 87] William E. Lorensen and Harvey E. Cline. “Marching cubes: A High Resolution 3D Surface Construction Algorithm.” In *Computer Graphics (Proceedings of SIGGRAPH 87)*, Vol. 21, ACM, pp. 163–169, 1987.
- [Ning and Bloomenthal 93] Paul Ning and Jules Bloomenthal. “An Evaluation of Implicit Surface Tilers.” *IEEE Computer Graphics and Applications* 13:6 (1993), pp. 33–34.
- [Shekhar et al. 96] Raj Shekhar, Elias Fayyad, Roni Yagel, and J. Fredrick Cornhill. “Octree-Based Decimation of marching cubes Surfaces.” In *Proceedings of the 7th Conference on Visualization '96*, pp. 335–342, 1996.
- [Shu et al. 95] Renben Shu, Chen Zhou, and Mohan S. Kankanhalli. “Adaptive marching cubes.” *The Visual Computer* 11 (1995), pp. 202–217.



# Fog with a Linear Density Function

February 2015.

Published in *Game Engine Gems 3*, 2016.

## 1 Introduction

In this chapter, we consider a fog volume inside which the density of the fog is a linear function of distance along some given direction. This naturally gives rise to a halfspace bounded by a plane where the density function is zero. On one side of the plane, the fog grows thicker as the distance from the plane increases, and on the other side of the plane, there is no fog at all. Such a fog volume has many uses in games ranging from heavy mist trapped in a valley to murky waters in a swamp. An example outdoor scene is shown in Figure 1.

We first derive a unified formula that allows us to render halfspace fog for all possible configurations in which the camera position and point being shaded are inside or outside the fog volume. Then, we examine the problem of determining which objects in the scene can be skipped when rendering because they are either too deep or too far away inside the fog volume to be visible.

The mathematics in this chapter are written in terms of four-dimensional Grassmann algebra. This means that we are using homogeneous coordinates and making a distinction between vectors and points. A vector  $\mathbf{V} = (V_x, V_y, V_z, 0)$  is written in bold style and always has a  $w$  coordinate of zero. A point  $\mathcal{P} = (P_x, P_y, P_z, 1)$  is written in script style and always has a  $w$  coordinate of one. A plane is represented by a trivector  $\mathbf{F} = (F_x, F_y, F_z, F_w)$  that we also write in the bold style. A plane  $\mathbf{F}$  can be multiplied by a vector  $\mathbf{V}$  or a point  $\mathcal{P}$  using the wedge product to produce a scalar as follows:

$$\begin{aligned}\mathbf{F} \wedge \mathbf{V} &= F_x V_x + F_y V_y + F_z V_z \\ \mathbf{F} \wedge \mathcal{P} &= F_x P_x + F_y P_y + F_z P_z + F_w.\end{aligned}\tag{1}$$

If the plane  $\mathbf{F}$  is normalized, meaning that  $F_x^2 + F_y^2 + F_z^2 = 1$ , then the product  $\mathbf{F} \wedge \mathcal{P}$  gives the signed perpendicular distance between the plane  $\mathbf{F}$  and the point  $\mathcal{P}$ . As illustrated in Figure 2, the normal direction of our fog plane points outward from the fog volume, so  $\mathbf{F} \wedge \mathcal{P} < 0$  for points  $\mathcal{P}$  below the fog plane in the fogged halfspace, and  $\mathbf{F} \wedge \mathcal{P} > 0$  for points  $\mathcal{P}$  above the fog plane in the unfogged halfspace.

## 2 Fog Factor Calculation

The fog factor  $f$  determines how the shaded color calculated on a surface is mixed with the fog color before it is finally output from a fragment shader. For fog having a constant density  $\rho$ , the fog factor is typically calculated with the formula

$$f = e^{-\rho d},\tag{2}$$

where  $d$  is the distance between the point being shaded and the camera position. Once the fog factor has been calculated, it is clamped to the range  $[0, 1]$ , and the fragment's final color  $K_{\text{final}}$  is blended

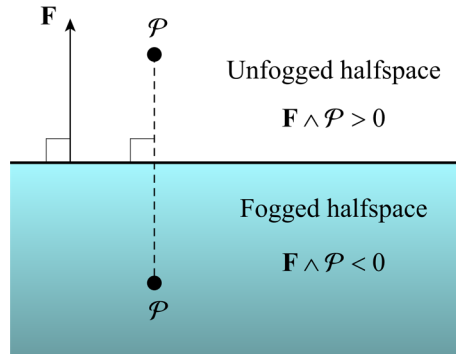
with a constant fog color  $K_{\text{fog}}$  to produce the color  $K$  that is written the frame buffer using the formula

$$K = fK_{\text{final}} + (1 - f)K_{\text{fog}}. \quad (3)$$

What follows is the derivation for the case that a linear density function  $\rho(\mathcal{P})$ , depending on the point  $\mathcal{P}$  being shaded, is used instead of a constant density [Lengyel 2007]. We account for the fact that the distance  $d$  is no longer necessarily equal to the total distance to the camera because the point  $\mathcal{P}$  and the camera position may lie on opposite sides of the fog plane.



**Figure 1.** This is an example of a fog volume that uses a linear density function. The fog plane (at which the density is zero) lies at a small distance above the camera position.



**Figure 2.** A point  $\mathcal{P}$  in the fogged halfspace forms a negative wedge product with the fog plane  $\mathbf{F}$ , and a point  $\mathcal{P}$  in the unfogged halfspace forms a positive wedge product.

## Derivation

Let the density function  $\rho(\mathcal{P})$  be defined as

$$\rho(\mathcal{P}) = -a(\mathbf{F} \wedge \mathcal{P}) \quad (4)$$

for some positive constant  $a$ , and let  $dg$  represent the contribution to the fog factor exponent along a differential length  $ds$ , given by the product

$$dg = \rho(\mathcal{P}) ds. \quad (5)$$

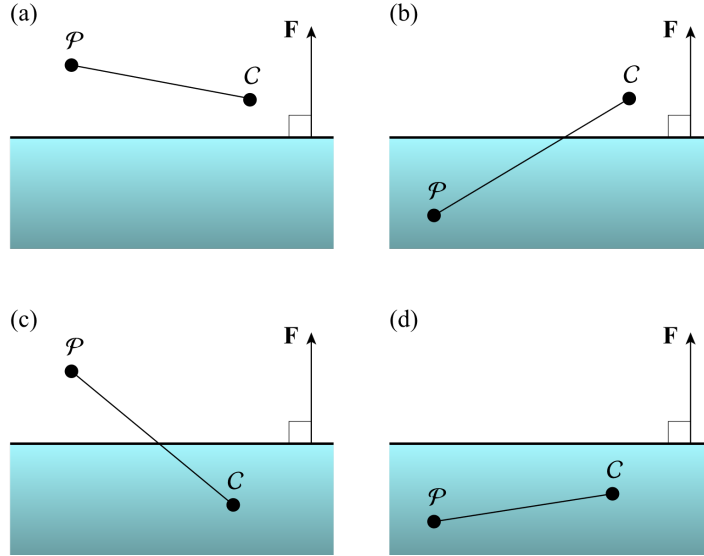
By integrating over the portion of the path that lies beneath the fog plane between the shaded point  $\mathcal{P}$  and the camera position  $C$ , we obtain a function  $g(\mathcal{P})$  that can be substituted for the product  $\rho d$  in Equation (2).

Let the function

$$\mathbf{Q}(t) = \mathcal{P} + t\mathbf{V} \quad (6)$$

represent the line segment connecting the shaded point  $\mathcal{P}$  with the camera position  $C$ , where  $t \in [0, 1]$  and the traditional view direction  $\mathbf{V}$  is defined as  $\mathbf{V} = C - \mathcal{P}$ . The differential distance  $ds$  can be expressed in terms of  $t$  as  $ds = \|\mathbf{V}\| dt$ . We need to consider the four possible configurations of the points  $\mathcal{P}$  and  $C$  with respect to the boundary plane, as illustrated in Figure 3. Of course, if  $\mathbf{F} \wedge \mathcal{P}$  and  $\mathbf{F} \wedge C$  are both positive, then no part of the line segment travels through the fog volume, and  $g(\mathcal{P}) = 0$ . In the case that  $\mathbf{F} \wedge \mathcal{P} < 0$  and  $\mathbf{F} \wedge C < 0$ , we integrate over the entire distance between  $\mathcal{P}$  and  $C$  to obtain

$$\begin{aligned} g(\mathcal{P}) &= \int_{\mathcal{P}}^C dg = \int_0^1 \rho(\mathbf{Q}(t)) \|\mathbf{V}\| dt \\ &= -\frac{a}{2} \|\mathbf{V}\| (\mathbf{F} \wedge \mathcal{P} + \mathbf{F} \wedge C). \end{aligned} \quad (7)$$



**Figure 3.** For a surface point  $\mathcal{P}$  and a camera position  $C$ , there are four distinct configurations to consider when integrating over the portion of the path between them that lies within the fogged halfspace.

In the two remaining cases, in which  $\mathbf{F} \wedge \mathcal{P}$  and  $\mathbf{F} \wedge \mathcal{C}$  have opposite signs, we must integrate over only the part of the path that lies inside the fog volume. The parameter  $t_{\text{plane}}$  at which the line segment  $\mathbf{Q}(t)$  passes through the fog plane is given by

$$t_{\text{plane}} = -\frac{\mathbf{F} \wedge \mathcal{P}}{\mathbf{F} \wedge \mathbf{V}}, \quad (8)$$

and this becomes one of the limits of integration. In the case that only  $\mathbf{F} \wedge \mathcal{P} < 0$ , we have

$$\begin{aligned} g(\mathcal{P}) &= \int_0^{t_{\text{plane}}} \rho(\mathbf{Q}(t)) \|\mathbf{V}\| dt \\ &= \frac{a}{2} \|\mathbf{V}\| \frac{(\mathbf{F} \wedge \mathcal{P})^2}{\mathbf{F} \wedge \mathbf{V}}, \end{aligned} \quad (9)$$

and in the case that only  $\mathbf{F} \wedge \mathcal{C} < 0$ , we have

$$\begin{aligned} g(\mathcal{P}) &= \int_{t_{\text{plane}}}^1 \rho(\mathbf{Q}(t)) \|\mathbf{V}\| dt \\ &= -\frac{a}{2} \|\mathbf{V}\| \left( \mathbf{F} \wedge \mathcal{P} + \mathbf{F} \wedge \mathcal{C} + \frac{(\mathbf{F} \wedge \mathcal{P})^2}{\mathbf{F} \wedge \mathbf{V}} \right). \end{aligned} \quad (10)$$

The following table summarizes the fog functions  $g(\mathcal{P})$  for the four possible cases shown in Figure 3. Our goal is to combine these into a single formula that produces the correct fog function in all cases without expensive branches or other conditional code in a fragment shader.

Case	$\mathbf{F} \wedge \mathcal{P}$	$\mathbf{F} \wedge \mathcal{C}$	Fog Function $g(\mathcal{P})$
(a)	Positive	Positive	0
(b)	Negative	Positive	$\frac{a}{2} \ \mathbf{V}\  \frac{(\mathbf{F} \wedge \mathcal{P})^2}{\mathbf{F} \wedge \mathbf{V}}$
(c)	Positive	Negative	$-\frac{a}{2} \ \mathbf{V}\  \left( \mathbf{F} \wedge \mathcal{P} + \mathbf{F} \wedge \mathcal{C} + \frac{(\mathbf{F} \wedge \mathcal{P})^2}{\mathbf{F} \wedge \mathbf{V}} \right)$
(d)	Negative	Negative	$-\frac{a}{2} \ \mathbf{V}\  (\mathbf{F} \wedge \mathcal{P} + \mathbf{F} \wedge \mathcal{C})$

First, we can make use of the fact that  $\mathbf{F} \wedge \mathbf{V}$  is always positive in case (b) and always negative in case (c). By applying an absolute value to  $\mathbf{F} \wedge \mathbf{V}$ , we can merge cases (b) and (c) into one formula and write  $g(\mathcal{P})$  as

$$g(\mathcal{P}) = -\frac{a}{2} \|\mathbf{V}\| \left( k(\mathbf{F} \wedge \mathcal{P} + \mathbf{F} \wedge \mathcal{C}) - \frac{(\mathbf{F} \wedge \mathcal{P})^2}{|\mathbf{F} \wedge \mathbf{V}|} \right), \quad (11)$$

where the constant  $k$  is defined as

$$k = \begin{cases} 1, & \text{if } \mathbf{F} \wedge \mathcal{C} \leq 0; \\ 0, & \text{otherwise.} \end{cases} \quad (12)$$

In order to incorporate case (d) into this formula, we need to eliminate the last term inside the brackets whenever  $\mathbf{F} \wedge \mathcal{P}$  and  $\mathbf{F} \wedge \mathbf{C}$  have the same sign. This can be accomplished by replacing  $\mathbf{F} \wedge \mathcal{P}$  with  $\min((\mathbf{F} \wedge \mathcal{P}) \operatorname{sgn}(\mathbf{F} \wedge \mathbf{C}), 0)$  in the last term to arrive at the formula

$$g(\mathcal{P}) = -\frac{a}{2} \|\mathbf{V}\| \left[ k(\mathbf{F} \wedge \mathcal{P} + \mathbf{F} \wedge \mathbf{C}) - \frac{[\min((\mathbf{F} \wedge \mathcal{P}) \operatorname{sgn}(\mathbf{F} \wedge \mathbf{C}), 0)]^2}{|\mathbf{F} \wedge \mathbf{V}|} \right]. \quad (13)$$

This formula also works for case (a) because both terms are eliminated when  $\mathbf{F} \wedge \mathcal{P}$  and  $\mathbf{F} \wedge \mathbf{C}$  are both positive, so we have found a single unified fog function that can be used in all cases. It may look complicated, but it is inexpensive to evaluate in a shader because  $\mathcal{P}$  and  $\mathbf{V}$  are the only values that vary.

Note that in Equation (13), if the quantity  $\mathbf{F} \wedge \mathbf{V}$  is zero, then it is always true that the numerator of the last term is also zero. Although in practice a zero-divided-by-zero situation is rare enough to be ignored, a small positive  $\varepsilon$  can be added to  $|\mathbf{F} \wedge \mathbf{V}|$  in the denominator to guarantee that a NaN is not produced without affecting the fog function's value significantly.

## Implementation

A fragment shader implementation of the unified fog function  $g(\mathcal{P})$  given by Equation (13) is shown in Listing 1. We can calculate the quantities  $\mathbf{V}$ ,  $\mathbf{F} \wedge \mathbf{V}$ ,  $k(\mathbf{F} \wedge \mathcal{P} + \mathbf{F} \wedge \mathbf{C})$ , and  $(\mathbf{F} \wedge \mathcal{P}) \operatorname{sgn}(\mathbf{F} \wedge \mathbf{C})$  in the vertex shader and interpolate them during triangle rasterization. This leaves only a somewhat small amount of computation to be performed in the fragment shader. The fog factor  $f$  is calculated as

$$f = \operatorname{sat}(2^{-g(\mathcal{P})/\ln 2}), \quad (14)$$

where the  $\operatorname{sat}$  function clamps to the range  $[0, 1]$ . The negative signs appearing in front of the right side of Equation (13) and in the exponent of Equation (14) cancel each other, so neither appears in the code.

**Listing 1.** This GLSL fragment shader code implements the unified fog function given by Equation (13), calculates the fog factor using Equation (14), and uses the fog factor to interpolate between the shader's final color and the fog color.

```
uniform float density;      // a / (2 ln 2)
uniform vec3 fogColor;

in vec3 V;                  // V = C - P
in float FV;                // F ^ V
in float c1;                // k * (F ^ P + F ^ C)
in float c2;                // (F ^ P) * sgn(F ^ C)

void main()
{
    vec4 color = ...;        // Final shaded surface color.
```

```

// Calculate g(P) using Equation (13).
float g = min(c2, 0.0);
g = length(V) * (c1 - g * g / abs(FV)) * density;

// Calculate fog factor and apply.
float f = clamp(exp2(g), 0.0, 1.0);

color.rgb = color.rgb * f + fogColor * (1.0 - f);
...
}

```

## Infinite Geometry

For geometry rendered at infinity, such as a skybox, vertices are no longer represented by homogeneous points  $\mathcal{P}$  having an implicit one in the  $w$  coordinate, but are instead given by direction vectors  $\mathbf{D}$  having an implicit zero in the  $w$  coordinate. In this case, a fragment is always fully fogged whenever  $\mathbf{F} \wedge \mathbf{D} < 0$ , regardless of the value of  $\mathbf{F} \wedge \mathbf{C}$ , because the ray given by

$$\mathcal{R}(t) = \mathbf{C} + t\mathbf{D} \quad (15)$$

travels an infinite distance through the fog volume. The only nontrivial case is the one in which  $\mathbf{F} \wedge \mathbf{D} > 0$  and  $\mathbf{F} \wedge \mathbf{C} < 0$ , where the camera is inside the fog volume, and the direction  $\mathbf{D}$  to the fragment being rendered points upward out of the fog volume.

To formulate the function  $g(\mathcal{P})$  for infinite geometry, we need to integrate from the camera position  $\mathbf{C}$  to the point where  $\mathcal{R}(t)$  crosses the fog plane. This point is given by the parameter

$$t_{\text{plane}} = -\frac{\mathbf{F} \wedge \mathbf{C}}{\mathbf{F} \wedge \mathbf{D}}, \quad (16)$$

and thus, the integral defining our fog function is

$$\begin{aligned}
g(\mathcal{P}) &= \int_0^{t_{\text{plane}}} \rho(\mathcal{R}(t)) \|\mathbf{D}\| dt \\
&= \frac{a}{2} \|\mathbf{D}\| \frac{(\mathbf{F} \wedge \mathbf{C})^2}{\mathbf{F} \wedge \mathbf{D}}.
\end{aligned} \quad (17)$$

In order to use this function in all cases, we can clamp the fog factor calculation using bounds derived from  $\mathbf{F} \wedge \mathbf{C}$  and  $\mathbf{F} \wedge \mathbf{D}$  to obtain

$$f = \text{clamp}\left(2^{-g(\mathcal{P})/\ln 2}, u, v\right), \quad (18)$$

where  $u$  and  $v$  are defined as

$$\begin{aligned}
u &= \begin{cases} 1, & \text{if } \mathbf{F} \wedge \mathbf{C} > 0 \text{ and } \mathbf{F} \wedge \mathbf{D} > 0; \\ 0, & \text{otherwise;} \end{cases} \\
v &= \begin{cases} 1, & \text{if } \mathbf{F} \wedge \mathbf{D} > 0; \\ 0, & \text{otherwise.} \end{cases}
\end{aligned} \quad (19)$$

If  $\mathbf{F} \wedge \mathbf{D} \leq 0$ , in which case the ray  $\mathcal{R}(t)$  must enter the fog volume at some point, then both  $u$  and  $v$  are zero, and the pixel being shaded is always fully fogged. If  $\mathbf{F} \wedge \mathbf{D} > 0$ , then the fog factor can be less than one only if  $\mathbf{F} \wedge \mathbf{C} \leq 0$ , corresponding to the case that the camera is inside the fog volume.

### 3 Visibility Culling

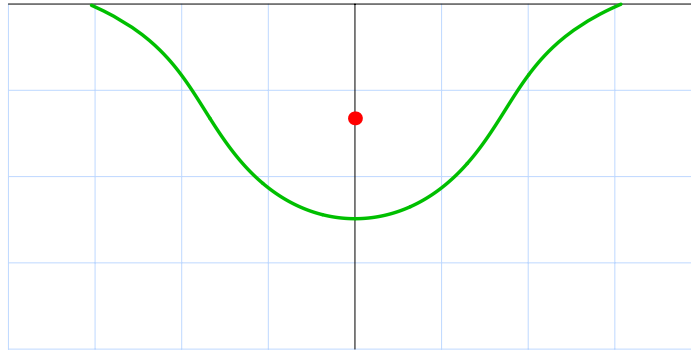
When rendering a scene with fog, objects can be far enough away from the camera to become imperceptible even though the fog factor  $f$  never reaches zero in a strictly mathematical sense. For example, a fog factor of  $1/256$  is too small for the light reflected from an object to make any difference when the display uses 8-bit color channels, and the rendered output would be 100% fog color. If we can calculate the distance at which the fog factor becomes this small, then we can cull any objects lying beyond that distance because they won't be visible. Due to the way the human visual system works, a significantly larger factor can be used to calculate the cutoff distance when the fog color has a bright intensity.

If we calculate the exact surface at which the fog factor reaches some small constant for a given camera position, then we get a shape such as the one shown in Figure 4, with some variation depending on the camera depth in the fog volume. (The formula for this curve is derived below.) Culling objects directly against this surface would be impractical, so we instead calculate the maximum distances at which objects would be visible in a couple different directions and place culling planes at those locations relative to the camera. First, we determine the vertical depth beneath the camera position within the fog volume at which objects become fully fogged and place a culling plane there that is parallel to the fog plane. This involves a simple, straightforward calculation. Second, we determine the horizontal distance (parallel to the fog plane) at which objects would be culled at *all* depths and place a culling plane at that distance from the camera aligned to be perpendicular to the camera's view direction. This calculation turns out to be much more complicated, but it has an elegant solution [Lengyel 2015].

In both cases, we determine the distance to the culling plane by setting the fog factor  $f = e^{-g(\mathcal{P})}$  to a constant  $c$  small enough to be considered the cutoff value for perceptibility (such as  $c = 1/256$ ). This means that  $g(\mathcal{P})$  can be treated as the constant value

$$g(\mathcal{P}) = -\ln c \quad (20)$$

in any of the fog function equations in the previous section.



**Figure 4.** The green curve represents the exact surface at which the fog factor reaches a small constant value given the camera position (red dot) beneath the fog plane at the top of the graph.

## Vertical Depth

To cull objects by vertical depth, we need to find the distance  $d$  directly beneath the camera, opposite the normal direction of the fog plane, at which an object becomes completely fogged within the tolerance set by Equation (20). We assume that the camera is inside the fog volume, which means  $\mathbf{F} \wedge \mathbf{C} \leq 0$ . In the case that the camera is above the fog plane, we can just set  $\mathbf{F} \wedge \mathbf{C} = 0$  because it wouldn't change the amount of fog through which a ray pointing straight down from the camera would pass.

Starting with Equation (7) and recognizing that  $\|\mathbf{V}\| = d$ , we have

$$g(\mathcal{P}) = -\frac{a}{2}d(\mathbf{F} \wedge \mathcal{P} + \mathbf{F} \wedge \mathbf{C}). \quad (21)$$

For a point  $\mathcal{P}$  lying directly below the camera position such that the vector  $\mathbf{C} - \mathcal{P}$  is perpendicular to the fog plane, we have  $d = \mathbf{F} \wedge \mathbf{C} - \mathbf{F} \wedge \mathcal{P}$ , and we can thus make the substitution

$$\mathbf{F} \wedge \mathcal{P} + \mathbf{F} \wedge \mathbf{C} = 2(\mathbf{F} \wedge \mathbf{C}) - d. \quad (22)$$

This leads us to the quadratic equation

$$d^2 - 2d(\mathbf{F} \wedge \mathbf{C}) - m = 0, \quad (23)$$

where we have set  $m = 2g(\mathcal{P})/a$ . The solution providing a positive distance  $d$  is then given by

$$d = \mathbf{F} \wedge \mathbf{C} + \sqrt{(\mathbf{F} \wedge \mathbf{C})^2 + m}. \quad (24)$$

A plane parallel to the fog plane can be placed at this distance below the camera position to safely cull objects that are too deep in the fog volume to be visible.

## Horizontal Distance

To cull objects in the horizontal direction, we need to find the minimum distance parallel to the fog plane beyond which objects are completely fogged (again, within tolerance) at all depths in the fog volume. This requires that we somehow find the maximum horizontal distance  $d$  for which Equation (7) holds true for a given constant  $g(\mathcal{P})$ , where  $d$  is the length of the projection of  $\mathbf{V}$  onto the fog plane. Some examples of the distance  $d$  are shown in Figure 5, which also illustrates how the culling surfaces can be classified into three general varieties that are discussed below.

The problem of finding the correct culling distance  $d$  becomes much easier to tackle if we write  $\mathbf{F} \wedge \mathcal{P}$  as a multiple of  $\mathbf{F} \wedge \mathbf{C}$  so we can make the substitution

$$\mathbf{F} \wedge \mathcal{P} = t(\mathbf{F} \wedge \mathbf{C}). \quad (25)$$

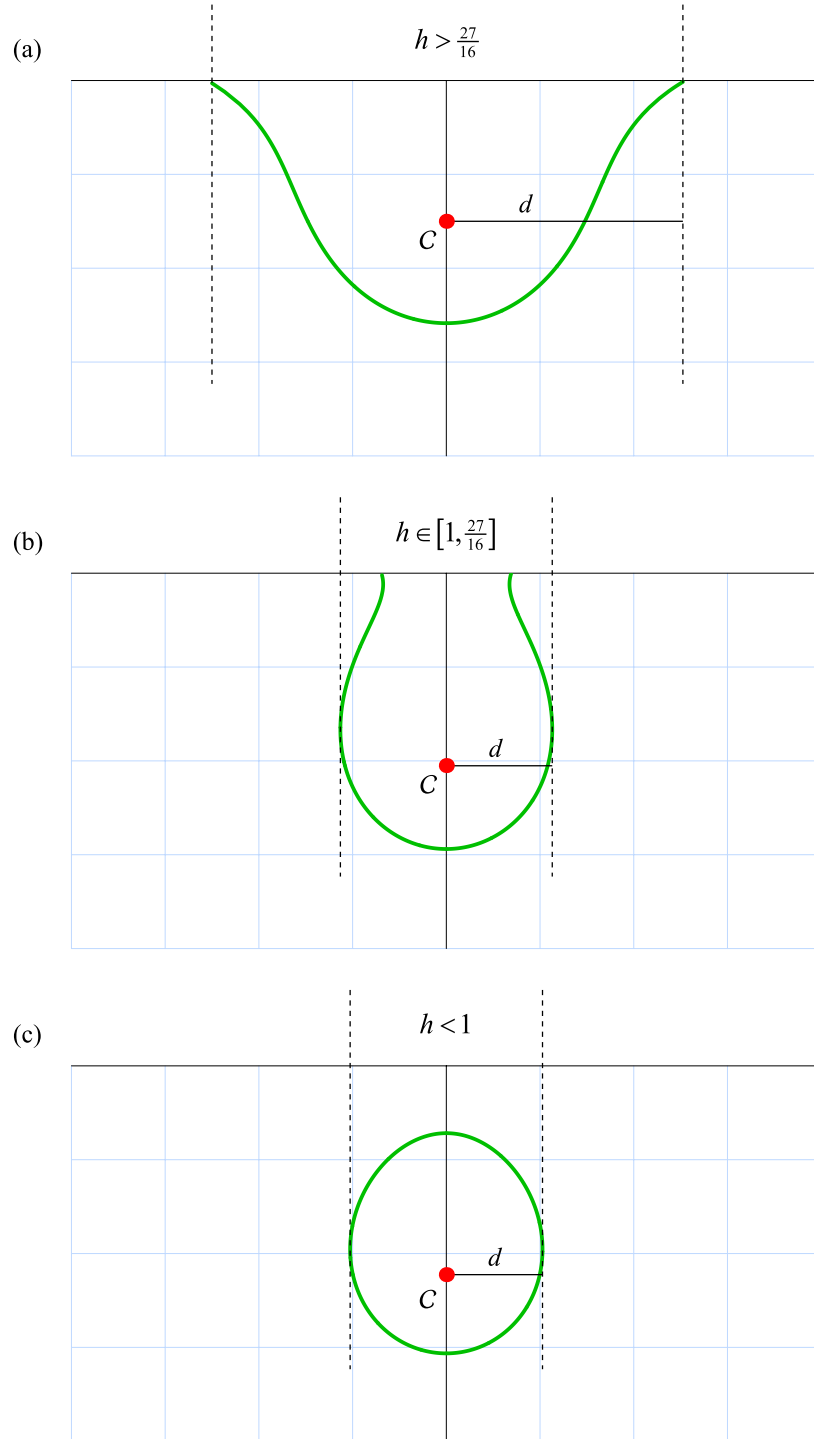
Equation (7) can then be written as

$$g(\mathcal{P}) = -\frac{a}{2}\|\mathbf{V}\|(t+1)(\mathbf{F} \wedge \mathbf{C}). \quad (26)$$

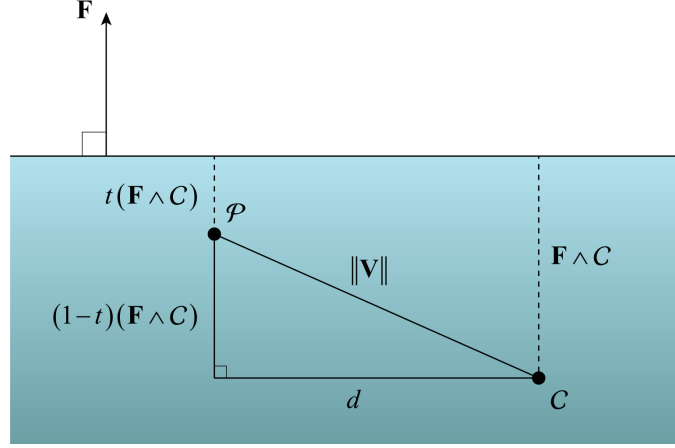
The vector  $\mathbf{V}$  still depends on the point  $\mathcal{P}$ , but its length can be expressed in terms of the horizontal distance  $d$  and the depth of the camera position  $\mathbf{C}$  by considering the right triangle shown in Figure 6, giving us

$$\|\mathbf{V}\|^2 = d^2 + (1-t)^2(\mathbf{F} \wedge \mathbf{C})^2. \quad (27)$$





**Figure 5.** The horizontal culling distance  $d$  is equal to the maximum distance parallel to the fog plane beyond which objects are completely fogged. The green curves represent the exact culling surfaces that can be classified into three cases depending on the value of  $h$ , defined by Equation (31).



**Figure 6.** The length of the vector  $\mathbf{V}$  can be expressed in terms of the distance  $d$  and the depths of the points  $\mathcal{C}$  and  $\mathcal{P}$ .

We can now rewrite Equation (26) as

$$g(\mathcal{P}) = -\frac{a}{2}(t+1)(\mathbf{F} \wedge \mathbf{C})\sqrt{d^2 + (1-t)^2(\mathbf{F} \wedge \mathbf{C})^2}, \quad (28)$$

and this depends only on the known quantities  $g(\mathcal{P})$  and  $\mathbf{F} \wedge \mathbf{C}$ .

Solving Equation (28) for  $d^2$  produces the function

$$d^2 = \frac{m^2}{(t+1)^2(\mathbf{F} \wedge \mathbf{C})^2} - (1-t)^2(\mathbf{F} \wedge \mathbf{C})^2 \quad (29)$$

that returns the squared culling distance for any input  $t$  representing the depth of a point  $\mathcal{P}$ , where we have again set  $m = 2g(\mathcal{P})/a$ . We can find the local maxima for the squared distance by taking a derivative of this function with respect to  $t$  and setting it equal to zero. After some simplification, we end up with the quartic equation

$$q(t) = t^4 + 2t^3 - 2t + h - 1 = 0, \quad (30)$$

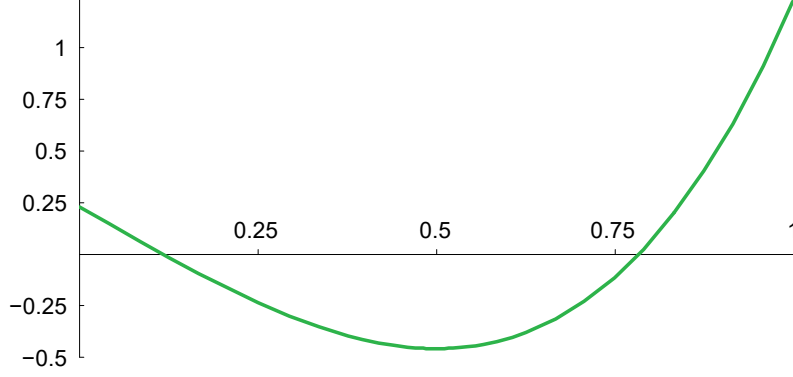
where we have named the polynomial  $q(t)$  and defined

$$h = \frac{m^2}{(\mathbf{F} \wedge \mathbf{C})^4}. \quad (31)$$

A typical plot of the quartic function  $q(t)$  in Equation (30) is shown in Figure 7. There are a couple of important properties that we can identify about this function. First, it is always the case that  $q(1) = h$ , and  $h$  is a positive number. Second, if we take a derivative to get

$$q'(t) = 4t^3 + 6t^2 - 2, \quad (32)$$

then we see that  $q'(\frac{1}{2}) = 0$ , which means that  $q(t)$  always has a local minimum at  $t = \frac{1}{2}$ . If the actual value of  $q(t)$  is negative at  $t = \frac{1}{2}$ , then  $q(t)$  must have a root in the range  $(\frac{1}{2}, 1)$ , and this is the value of  $t$  that we are looking for. By evaluating  $q(\frac{1}{2})$ , we see that this root exists precisely when  $h < 27/16$ , and this corresponds to cases (b) and (c) in Figure 5.



**Figure 7.** This is a plot of the function  $q(t) = t^4 + 2t^3 - 2t + h - 1$  with  $g(\mathcal{P}) = \ln 256$ ,  $a = 0.1$ , and  $\mathbf{F} \wedge \mathbf{C} = -10$ , giving an approximate value of  $h = 1.23$ .

It may be tempting to find the roots of  $q(t)$  by using an analytical solution, but in this case, the root in which we are interested can be found much more efficiently and accurately by using Newton's method with an initial value of  $t_0 = 1$ . Recall that Newton's method refines an approximation  $t_i$  of a root using the formula

$$t_{i+1} = t_i - \frac{q(t_i)}{q'(t_i)}. \quad (33)$$

We know right at the beginning that  $q(1) = h$  and  $q'(1) = 8$ , so the first iteration of Newton's method can be explicitly calculated with ease to obtain

$$t_1 = 1 - \frac{h}{8}. \quad (34)$$

One or two more iterations are all that are needed to produce an extremely accurate result that can be plugged back into Equation (29) to calculate the culling distance  $d$ .

If  $h > 1$ , then  $d^2 > 0$  when  $t = 0$ , and it may be the case that the largest horizontal distance to the culling surface occurs on the fog plane itself. This corresponds to cases (a) and (b) in Figure 5. When  $t = 0$ , the distance  $d$  is given by

$$d = -(\mathbf{F} \wedge \mathbf{C})\sqrt{h - 1}, \quad (35)$$

where we have been careful to negate  $\mathbf{F} \wedge \mathbf{C}$  when factoring it out of the radical after taking square roots of both sides in Equation (29).

If  $h \in [1, \frac{27}{16}]$ , then the distance given by the root of Equation (30) and the distance given directly by Equation (35) are both valid, and the larger must be chosen as the horizontal culling distance. This corresponds to case (b) in Figure 5. Otherwise, only one of the distances can be calculated and is thus the only choice for the horizontal culling distance. Listing 2 implements the distance calculations and handles all three of the possible cases.

**Listing 2.** This C++ code implements the horizontal culling distance calculations for a given value of  $F \wedge C$  passed in the `F_wedge_C` parameter. The constant `kFogCutoff` is the value of  $c$  in Equation (20), and the constant `kFogDensity` is the value of  $a$  in Equation (4).

```
float CalculateHorizontalCullingDistance(float F_wedge_C)
{
    // Calculate m using Equation (20) for g(P).
    float m = -2.0F * log(kFogCutoff) / kFogDensity;
    float m2 = m * m;

    float d = 0.0F;
    float z2 = F_wedge_C * F_wedge_C;
    float zinv = 1.0F / F_wedge_C;
    float zinv2 = zinv * zinv;

    // Calculate h using Equation (31).
    float h = m2 * zinv2 * zinv2;
    if (h < 1.6875F)
    {
        // Here, h < 27/16, so a root to q(t) exists.
        // Explicitly calculate first iteration of Newton's method.
        float t = 1.0F - h * 0.125F;
        float t2 = t * t;

        // Apply Newton's method one more time.
        t -= (((t + 2.0F) * t2 - 2.0F) * t + (h - 1.0F)) /
            ((t * 4.0F + 6.0F) * t2 - 2.0F);

        // Plug root back into Equation (29).
        float tp = t + 1.0F; float tm = t - 1.0F;
        d = sqrt(m2 * zinv2 / (tp * tp) - tm * tm * z2);
    }

    if (h > 1.0F)
    {
        // Calculate the distance on the fog plane using Equation (35).
        // If both solutions exist, take the larger distance.
        d = max(-F_wedge_C * sqrt(h - 1.0F), d);
    }

    return (d);
}
```

## References

- [Lengyel 2007] Eric Lengyel. “Unified Distance Formulas for Halfspace Fog”. *Journal of Graphics Tools*, Vol. 12, No. 2 (2007), pp. 23–32.
- [Lengyel 2015] Eric Lengyel. “Game Math Case Studies”. Game Developers Conference, 2015. Available at [http://www.terathon.com/gdc15\\_lengyel.pdf](http://www.terathon.com/gdc15_lengyel.pdf).

# GPU-Centered Font Rendering Directly from Glyph Outlines

November 2016.

Published in *Journal of Computer Graphics Techniques*, Vol. 6, No. 2, 2017.

Described by United States patent 10,373,352, granted August 6, 2019.

**Abstract.** This paper describes a method for rendering antialiased text directly from glyph outline data on the GPU without the use of any precomputed texture images or distance fields. This capability is valuable for text displayed inside a 3D scene because, in addition to a perspective projection, the transform applied to the text is constantly changing with a dynamic camera view. Our method overcomes numerical precision problems that produced artifacts in previously published techniques and promotes high GPU utilization with an implementation that naturally avoids divergent branching.

## 1 Introduction

Games and other real-time 3D applications often have a need to render text on various surfaces inside a virtual environment, as shown in Figure 1. Because the camera is almost always moving in some way, the glyphs that compose such text are drawn with continuously changing transforms, and thus the size of the glyphs in the viewport are almost never the same from one frame to the next. Furthermore, the glyphs are usually drawn with perspective distortion because the camera isn't pointed straight at the surface to which text is applied. This creates a demand for the ability to dynamically render text with high quality no matter what affine and projective transformations have been applied to it.

Prerendered glyphs stored in a texture atlas have traditionally been used to apply text to in-game surfaces. Despite being simple and extremely fast, this technique suffers from unsightly blurring due to bilinear filtering once the displayed sizes of the glyphs exceed the resolution at which they're stored in the texture atlas. This problem was addressed by the introduction of signed distance field (SDF) methods [Green 2007], which produce crisp boundaries at all scales by storing distances to the glyph boundaries in the texture atlas instead of the final appearance of each glyph at a particular size. However, these methods tend to round off sharp corners and thus do not preserve the true outlines of the glyphs. Multichannel signed distance fields [Chlumský 2015] corrected the corner rounding problem, but required a complicated analysis step in the preparation of the texture atlas and created a new class of difficult-to-avoid artifacts for complex glyphs.

All of the techniques that store data in a texture atlas are inherently using a discrete sampling of what is actually an infinitely precise description of a glyph outline. This inescapably leads to limitations that can be extended by increasing the resolution of the texture atlas, but can never be completely removed. For applications that need to render a wide range of characters at potentially



**Figure 1.** Glyphs are rendered in a game level directly from Bézier curve data extracted from a TrueType font. Because no precomputed images or distance fields are utilized, the results are pixel accurate under any affine or projective transformation, including all scales, rotations, and perspective distortions.

large font sizes, a texture atlas capable of producing glyphs at an acceptable level of quality may have prohibitively large storage requirements.

The limitations of sampling can be avoided altogether by rendering glyphs directly from the mathematical curves that define their shapes. No longer does the source data for each glyph have an intrinsic resolution because the exact positions of the outline’s control points are utilized throughout the rendering process without any prior sampling. The Loop-Blinn method [2005] renders text directly from outline data by constructing a triangle mesh for each glyph that incorporates the control points as vertex positions. Each triangle corresponds to at most one component of a glyph’s outline, and a short calculation in the pixel shader determines whether each pixel is inside or outside the boundary with respect to that one component. This method effectively renders precise glyph shapes at all scales, but it requires a complicated triangulation step in which the number of vertices is tied to the number of control points defining each glyph. At small font sizes, these triangles can become very tiny and decrease thread group occupancy on the GPU, reducing performance. The variable and font-dependent numbers of vertices per glyph also make it somewhat difficult to perform text layout in general, and greater difficulty arises when attempting to apply heavily triangulated glyphs to curved surfaces.

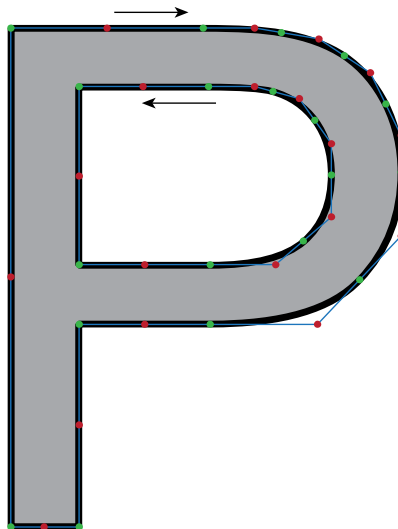
A more versatile solution renders each glyph using only two triangles to cover the glyph’s bounding box. The pixel shader then accesses a subset of *all* the components of the glyph’s outline to determine whether each pixel is inside or outside the entire boundary. This requires that we have a robust way of dynamically calculating either a signed distance value or a winding number at each pixel. The winding number corresponds to the absolute difference of the number of closed contours wound clockwise around the pixel and the number of closed contours wound counterclockwise around the pixel. Previous attempts at implementing such methods [Esfahbod 2015; Dobbie 2016] have suffered from numerical precision issues that produce a variety of rendering artifacts. These artifacts often begin to appear upon modest magnification and manifest themselves as dropped pixels inside a glyph or incorrectly drawn pixels outside a glyph. Because the artifacts are usually due to round-off errors, they tend to be very position sensitive and thus “sparkle” as the location or scale of a glyph changes. These sparkles also tend to occur along straight lines, producing “streaking” artifacts at various angles inside or outside a glyph.

We present a new technique in this paper that solves the precision problem and completely eliminates all artifacts by taking a different approach. Once robustness is guaranteed, we focus on ways to minimize the number of outline components examined at each pixel for best performance. Our method requires only widely available GPU features and can be implemented on OpenGL 3.x / DX10 hardware.

## 2 Winding Number Calculation

A glyph is defined by a set of closed contours that are each composed of a continuous piecewise sequence of Bézier curves, as shown in Figure 2. Although cubic curves are supported by other formats, we restrict ourselves to the quadratic curves used by TrueType fonts to keep rendering calculations as simple as possible. A particular point is considered to be inside the glyph outline if the sum of its winding numbers with respect to all of the contours is nonzero. The winding number for each contour is an integer that reflects the number of complete loops the contour makes around a point. A positive number is assigned to one winding direction, clockwise or counterclockwise, and a negative number is assigned to the opposite winding direction.

The winding number is calculated by firing a ray in any arbitrary direction from the point being rendered and looking for intersections with each of the contours belonging to a glyph. The winding number is initialized to zero. When a contour crosses the ray from left to right, one is added to the winding number, and when a contour crosses the ray from right to left, one is subtracted from the winding number (or vice-versa, as long as the choice is consistent). If the final winding number is zero, then the point lies in empty space. Otherwise, the point lies inside the glyph outline, and the



**Figure 2.** The shape of a glyph is defined by one or more closed contours each composed of a continuous sequence of quadratic Bézier curves. The green dots represent on-curve control points, the red dots represent off-curve control points, and the blue lines are tangent to the outline. The clockwise winding convention is used in this example, meaning that contours wound in a clockwise direction contribute a positive winding number, and contours wound in a counterclockwise direction, like the interior loop of the letter P, contribute a negative winding number.

fact that it can be positive or negative allows contours to employ either a clockwise or counter-clockwise convention for defining interior regions of the glyph.

Since the ray directions don't matter, we choose directions that are parallel to the coordinate axes for convenience. A single component of a contour is defined by the parametric function

$$\mathbf{C}(t) = (1-t)^2 \mathbf{p}_1 + 2t(1-t) \mathbf{p}_2 + t^2 \mathbf{p}_3, \quad (1)$$

which constitutes a quadratic Bézier curve having the 2D control points  $\mathbf{p}_1$ ,  $\mathbf{p}_2$ , and  $\mathbf{p}_3$ . The parameter  $t$  varies over the range  $[0, 1]$ . For a ray pointing in the direction of the positive  $x$  axis in a coordinate system in which the point being rendered has been translated to the origin, we can solve for the values of  $t$  at which  $C_y(t) = 0$  by finding the roots of the polynomial

$$(y_1 - 2y_2 + y_3)t^2 - 2(y_1 - y_2)t + y_1, \quad (2)$$

where we have set  $\mathbf{p}_i = (x_i, y_i)$ . The roots  $t_1$  and  $t_2$  are then

$$t_1 = \frac{b - \sqrt{b^2 - ac}}{a} \quad \text{and} \quad t_2 = \frac{b + \sqrt{b^2 - ac}}{a}, \quad (3)$$

where  $a = y_1 - 2y_2 + y_3$ ,  $b = y_1 - y_2$ , and  $c = y_1$ . In the case that  $a$  is near zero, we instead compute  $t_{1,2} = c/2b$ . For any values of  $t$  in the range  $[0, 1)$  such that  $C_x(t_i) \geq 0$ , we have found an intersection between the ray and a contour. The value  $t_i = 1$  is specifically disallowed because it corresponds to an intersection at  $t_i = 0$  for the succeeding component in the contour, and we don't want to count an intersection at a shared control point twice.

To determine whether a ray intersection corresponds to a positive or negative change in the winding number, we examine the values of  $C_y(t)$  before or after a root  $t_i$ , but only in the range  $[0, 1)$  and only between the two roots if both fall in that range. If  $C_y(t) > 0$  for  $t < t_i$  or  $C_y(t) < 0$  for  $t > t_i$ , then we add one to the winding number. Conversely, if  $C_y(t) < 0$  for  $t < t_i$  or  $C_y(t) > 0$  for  $t > t_i$ , then we subtract one from the winding number. Note that these conditions exclude straight lines parallel to the ray from making any contribution to the winding number.

While sound from a purely mathematical standpoint, the method just described is plagued by numerical precision errors in any practical implementation whenever  $y_1$  or  $y_3$  is near zero. The problem is that the finite number of bits in a floating-point value are incapable of producing the exactness needed in calculating  $t_1$  and  $t_2$  where either could be close to zero or one. The result is that a ray passing too close to the point where two contour components are connected may end up counting two intersections or missing both curves altogether, leading to the sparkle and streak artifacts described in the previous section. The situation is especially bad if the Bézier curve is tangent to the ray at its first or last control point. Typical hacks such as the use of epsilons or coordinate perturbation may eliminate the problem in some cases, but these measures are not generally effective and do not lead to a robust solution.

We now introduce a different approach that achieves absolute, unconditional robustness over the entire space of finite inputs (i.e., no coordinate value is infinity or NaN). Our new method ignores the values of  $t_i$  inasmuch as whether they satisfy  $t_i \in [0, 1)$  and instead calculates winding numbers based solely on a binary classification of the values  $y_1$ ,  $y_2$ , and  $y_3$ , specifically whether each is positive or not positive. Every quadratic Bézier curve then has a three-bit state that reduces the problem domain to exactly eight distinct equivalence classes. For all of the cases belonging to each equivalence class, contributions to the winding number arising from the two roots at  $t_1$  and  $t_2$  are handled in exactly the same way. Furthermore, whenever a contribution is made for the root at  $t_1$ , we always add one to the winding number, and whenever a contribution is made for the root at



$t_2$ , we always subtract one from the winding number. Thus, all we have to do is turn a three-bit input into a two-bit output, and we have all of the information necessary to properly handle all possible configurations of a quadratic Bézier curve, including degenerate cases, with respect to a ray pointing in the positive  $x$  direction.

The eight equivalence classes are illustrated in Table 1. In the columns labelled  $y_i$ , a one indicates that  $y_i > 0$ . In the columns labelled  $t_i$ , a one indicates that the root at  $t_i$  should make a contribution to the winding number if  $C_x(t_i) \geq 0$ . The contribution is always positive one for  $t_1$ , and it is always negative one for  $t_2$ , regardless of the order of  $C_x(t_1)$  and  $C_x(t_2)$ . The representative curves shown in the table for each equivalence class cover all 27 cases in which  $y_i < 0$ ,  $y_i = 0$ , and  $y_i > 0$  in order to make it clear what happens in the important instances in which the ray passes directly through a control point. A change is made to the winding number whenever the curve transitions from positive to not positive or vice-versa, and these changes are indicated by green and red dots in the table. A green dot corresponds to a change of positive one occurring when the curve transitions from positive to not positive at the root  $t_1$ , and a red dot corresponds to a change of negative one occurring when the curve transitions from not positive to positive at the root  $t_2$ .

In equivalence classes A and H, no transitions between positive and not positive ever occur, and thus no change is made to the winding number. In each of the remaining six equivalence classes, the potential for a contribution to the winding number exists. The historically difficult case in which a contour is tangent to the ray at an endpoint shared by two consecutive curves is handled without explicit detection or special code. Equivalence class A covers all cases for which a contour is tangent to the ray at an endpoint but is otherwise *negative*, ensuring that the winding number is unaffected. In the similar case that a contour is tangent to the ray at an endpoint but is otherwise *positive*, two equal and opposite contributions are always made to the winding number, and they cancel each other out exactly. This is exemplified by the many combinations of tangent curves shown in the table in which a green dot and red dot would coincide when the curves are connected to each other. (Note that there is no requirement that the curves have a continuous derivative at the endpoint where they are joined.) In equivalence classes C and F, there is a special case in which  $y_1$  and  $y_3$  have the same state but  $y_2$  has the opposite state, and it's possible that  $C_y(t)$  has no real roots. In order to handle this case with uniformity, we clamp  $b^2 - ac$  to zero, which has the effect of setting  $t_1 = t_2 = b/a$ . If one root makes a contribution, then the other one does as well in this case because  $C_x(t_1) = C_x(t_2)$ , so they cancel each other out. This combination of a positive and negative contribution is represented by the yellow dot shown in the table for class F.

The values in the columns labelled  $t_i$  in Table 1 form a 16-bit lookup table that can simply be expressed as the number  $0x2E74$ , with row A corresponding to the two least significant bits and row H corresponding to the two most significant bits. The values in the columns labelled  $y_i$  form a shift code such that when the lookup table is shifted right by twice that amount, then the output state for the corresponding equivalence class appears in the lowest two bits. Thus, given  $y_1$ ,  $y_2$ , and  $y_3$  for an arbitrary quadratic Bézier curve that has been translated so that the point being rendered is at the origin, all we have to do is calculate the value

$$((y_1 > 0) ? 1 : 0) + ((y_2 > 0) ? 2 : 0) + ((y_3 > 0) ? 4 : 0) \quad (4)$$

and use it to shift the number  $0x2E74$  rightward. If the lowest bit of the result is set, then one is added to the winding number when  $C_x(t_1) \geq 0$ . If the ninth lowest bit of the result is set, then one is subtracted from the winding number when  $C_x(t_2) \geq 0$ .

Because the precision-sensitive range checks on the values of  $t_i$  have been eliminated, it is no longer possible to miscount the number of intersections that a ray makes with a contour. The shift code is an exact calculation based on the translated  $y$  coordinates of the input control points, which

Class	$y_3$	$y_2$	$y_1$	$t_2$	$t_1$	Representative curves
<b>A</b>	0	0	0	0	0	
<b>B</b>	0	0	1	0	1	
<b>C</b>	0	1	0	1	1	
<b>D</b>	0	1	1	0	1	
<b>E</b>	1	0	0	1	0	
<b>F</b>	1	0	1	1	1	
<b>G</b>	1	1	0	1	0	
<b>H</b>	1	1	1	0	0	

**Table 1.** The three bits in the columns labelled  $y_i$  constitute an input code based on whether each  $y_i$  is positive, and they partition the set of all quadratic Bézier curves into eight equivalence classes. The two bits in the columns labelled  $t_i$  constitute an output code specifying whether each intersection with the  $x$  axis should make a contribution to the winding number of the contour to which the curve belongs. Green dots indicate roots at which one is added to the winding number, and red dots indicate roots at which one is subtracted from the winding number. The shaded areas represent the interior of a glyph when a clockwise winding convention is followed.

are invariant along any horizontal ray. The quadratic and linear terms of  $C_x(t)$  are also invariant, leaving only the constant term equal to the translated  $x$  coordinate of the control point  $\mathbf{p}_1$  as the quantity that changes as the ray origin is moved left or right. This guarantees that there exists a value  $x_0$  such that for all  $x \leq x_0$ , a particular contour intersection is counted, and for all  $x > x_0$ , the same intersection is not counted.

Of course, calculating a discrete inside/outside state at each point being rendered produces a pixelated, black and white image. Instead of calculating an integral winding number, we can accumulate coverage values that reflect how close each ray intersection is to the center of the pixel being rendered. The fraction  $f$  of a pixel crossed by a ray from left to right before an intersection occurs is given by

$$f = \text{sat}\left(mC_x(t_i) + \frac{1}{2}\right), \quad (5)$$

where  $m$  is the number of pixels in one em, which corresponds to the font size, and  $\text{sat}$  is the saturate function that clamps to the range  $[0, 1]$ . Adding and subtracting these fractions from the winding number has the effect of antialiasing in the direction of the rays. Averaging the final coverages calculated for multiple ray directions antialiases with greater isotropy, but at a performance cost. Considering only rays parallel to the coordinates axes is a good compromise, especially when combined with supersampling, as discussed later.

### 3 Performance Optimization

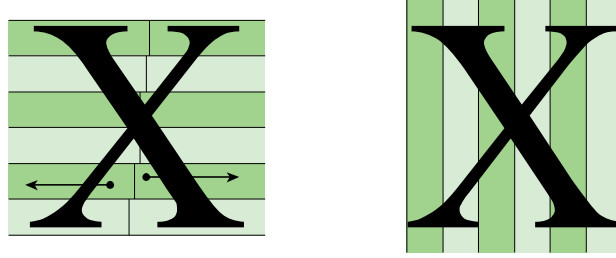
A glyph is correctly rendered when we consider every contour component for each pixel intersecting the glyph's bounding box and accumulate the coverage values. Fortunately, no branching is necessary to calculate a coverage value, so a pixel shader runs with high thread group coherence on the GPU. However, many of the quadratic Bézier curves make no contribution because they never cross a horizontal or vertical ray fired from a particular pixel center, and the best performance is achieved by minimizing the number of components that need to be processed.

We divide each glyph into a number of equal-width horizontal and vertical bands, as shown in Figure 3. The number of bands is proportional to the total number of Bézier curves composing the glyph's outline up to a limit of 16 in each direction. For each band, we create a list of the curves that intersect the band and sort them in descending order by their maximum coordinates in the band's direction ( $x$  for horizontal and  $y$  for vertical). When a pixel is rendered, we first determine which horizontal band contains it and loop over the Bézier curves that are known to intersect that band. As soon as we encounter a Bézier curve for which

$$\max\{x_1, x_2, x_3\}m < -\frac{1}{2}, \quad (6)$$

we can break out of the loop because Equation (5) produces a value of zero from that point onward. The process is repeated for the vertical band containing the pixel using a ray that points in the positive  $y$  direction and checking the maximum  $y$  coordinate of each curve for the early out condition.

For rays pointing in the positive direction along an axis, more curves need to be processed for pixels near the left or bottom sides of a glyph than for pixels near the right or top sides. To reduce the costs of rendering these pixels and make the whole process more symmetric, we split each band into two parts at a location roughly corresponding to the median position of the Bézier curves in that band. As shown in Figure 3, pixels falling on the negative side of the split location fire rays in the negative direction instead of the positive direction. In these cases, the winding number contributions must be negated, so Equation (5) is replaced with



**Figure 3.** A glyph is divided into equal-width horizontal and vertical bands. Two lists of Bézier curves intersecting each band are created, one sorted in descending order by maximum  $x$  or  $y$  coordinate (for horizontal and vertical bands, respectively), and another sorted in ascending order by minimum  $x$  or  $y$  coordinate. Bands are split along a median position, and rays originating on either side point away from this median to reduce the number of curves that need to be processed.

$$f = \text{sat}\left(\frac{1}{2} - mC_x(t_i)\right). \quad (7)$$

Because the ray is pointing in the opposite direction, we also need to sort the curves in the opposite order by their minimum coordinates and replace the early out condition given by Equation (6) with

$$\min\{x_1, x_2, x_3\}m > \frac{1}{2}. \quad (8)$$

Due to the divergence that it introduces in the pixel shader, the band split optimization can have a negative performance impact at small font sizes, so we make it an option for text that is known ahead of time to be rendered at larger sizes.

The most straightforward geometry to render for a glyph is a single quad corresponding to its bounding box. In order to capture all of the pixels for which Equations (5) and (7) could generate fractional values, the box needs to be expanded by half the width of a single pixel. The bounding boxes for uppercase letters are shown in the top row of Figure 4 to demonstrate how many glyphs contain nothing but empty space near the corners. We don't want to run the pixel shader for all of those pixels that end up being completely transparent, so we eliminate some of this empty space by clipping the corners of the bounding boxes where possible, as shown in the bottom row of Figure 4. Where to clip is determined by considering several normal directions at each corner, calculating the support plane for each normal direction with respect to all of the glyph's control points, and choosing the plane that clips off the triangle having the greatest area above some minimum threshold. As with the band split optimization, this geometry clipping optimization is more effective at larger font sizes. The tiny triangles that it can produce at small font sizes reduce occupancy on the GPU, which can result in slightly worse performance.

## 4 Implementation

Our implementation stores the control points for every glyph in a four-channel 16-bit floating-point texture map. The first and second control points belonging to each Bézier curve are stored in the  $(x, y)$  and  $(z, w)$  components of a single texel. The third control point is stored in the  $(x, y)$  components of the next texel in the same row. Since the third control point of one curve is identical to the first control point of the next curve in the same contour, it is usually the case that the second texel is shared by two curves, and thus the total storage requirements for the control point data is slightly larger than eight bytes per curve.

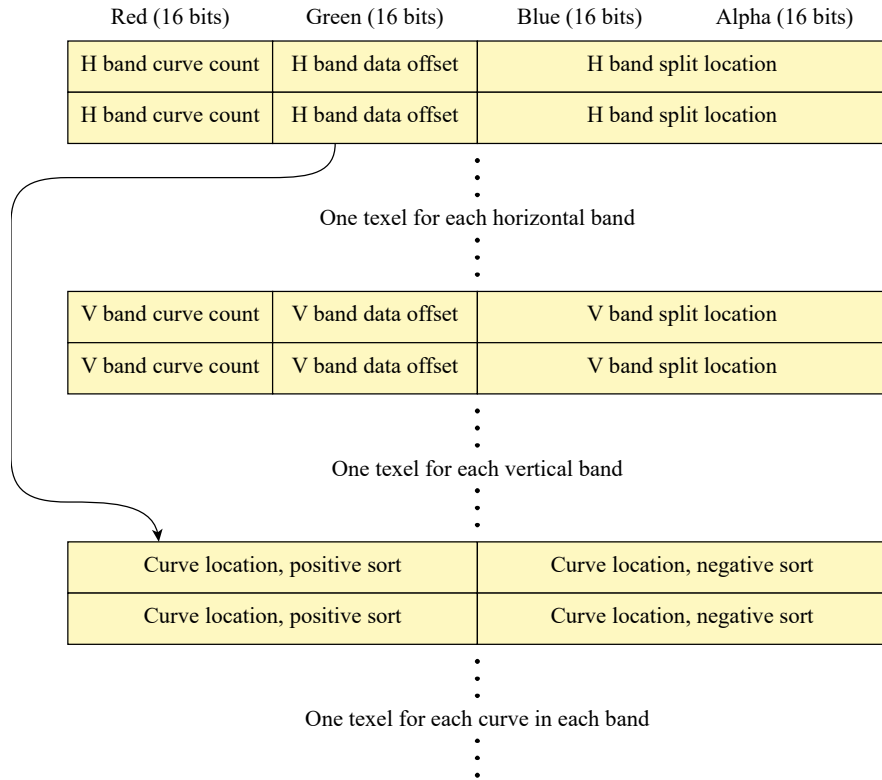


**Figure 4.** To reduce the number of pixels rendered for most glyphs, simple quads are replaced with polygons having up to eight sides after the corners of the bounding boxes are clipped.

A second texture map containing four-channel 16-bit integer data holds the location of every Bézier curve intersecting the horizontal and vertical bands belonging to each glyph. The layout of this data is shown in Figure 5. The data for a glyph begins with a table of band headers for all of the horizontal bands followed by all of the vertical bands. The header fits into one texel and contains the number of curves intersecting the band, the offset to the list of curve locations, and the coordinate value at which the band is split between negative rays and positive rays. The list of curve locations is actually two lists that occupy different channels in the same set of texels. One set of  $(x, y)$  coordinates holding the location of a Bézier curve in the control point texture is stored in the red and green channels, and another set of  $(x, y)$  coordinates is stored in the blue and alpha channels. The list of curves is sorted in descending order by maximum coordinate in the red and green channels for positive rays, and it is sorted in ascending order by minimum coordinate in the blue and alpha channels for negative rays.

The pixel shader that renders a glyph can be found in the supplemental materials. For the sake of brevity, the band split optimization is omitted, and rays are always fired in the positive direction along each of the  $x$  and  $y$  axes. The pixel's position in em-square coordinates is interpolated and passed into the pixel shader through the `texcoord` input. The starting location of the band data is passed to the pixel shader from the vertex shader in the lower 12 bits of the  $x$  and  $y$  components of the `glyphParam` input. (This limits the band data texture to  $4096 \times 4096$  texels.) The upper four bits contain the maximum band indexes for vertical bands in the  $x$  component and horizontal bands in the  $y$  component. Scale and offset parameters are passed in through the `bandParam` input, and they are used to calculate band indexes for each pixel. All bands, both horizontal and vertical, have the same width, so a single scale value is passed in through the  $z$  component of `bandParam`. The  $x$  and  $y$  components contain separate offsets for vertical and horizontal bands, respectively. Once the scale and offsets have been applied, the resulting band indexes are clamped to the maximum values specified in the `glyphParam` input.

After the band indexes have been determined, the shader reads the headers from the band data texture, locates the per-band curve lists, and calculates a coverage value for each curve until the early-out condition is satisfied or all the curves have been processed. For each curve, Equation (4) is used to calculate a shift code that is then applied to the lookup table `0x2E74` to move the root contribution code into the lowest two bits. Although not strictly necessary, the shader then tests whether these two bits are nonzero before continuing with the coverage calculation because we have found that doing so provides a small speed increase. If a contribution could be made, then the roots of the curve are calculated with Equation (3), and the cumulative coverage value is increased or decreased by the value given by Equation (5) as directed by the two-bit contribution code.



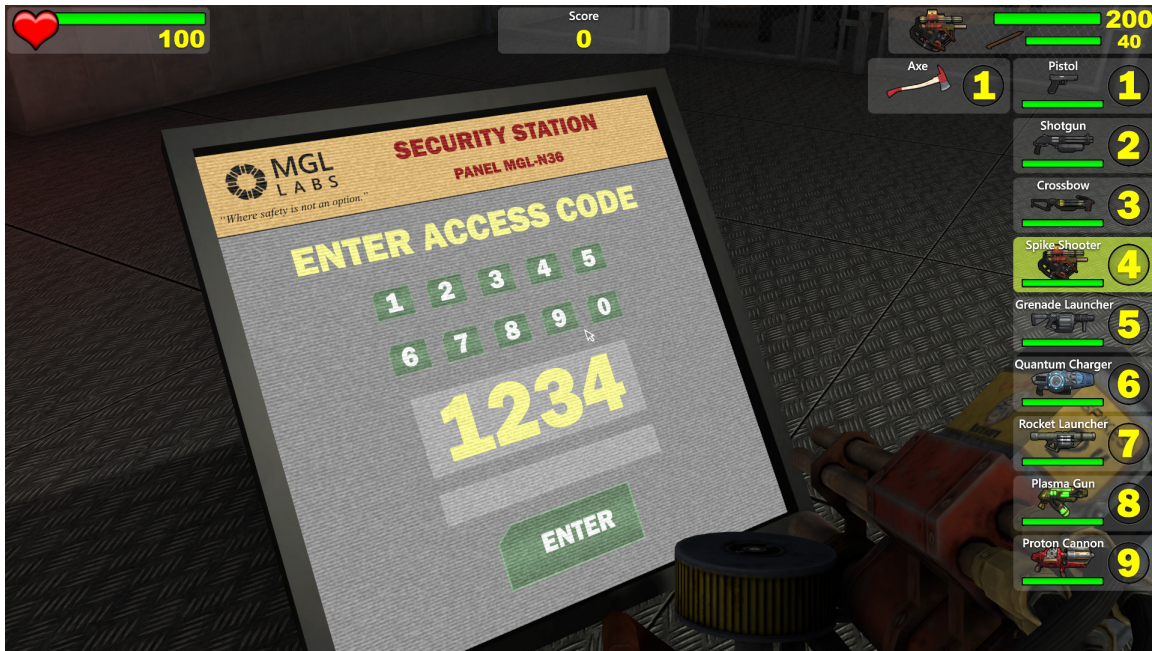
**Figure 5.** For every glyph, the band data texture includes a header texel for each horizontal and vertical band, and those are followed by the locations of the curves belonging to each band in the control point texture.

## 5 Results

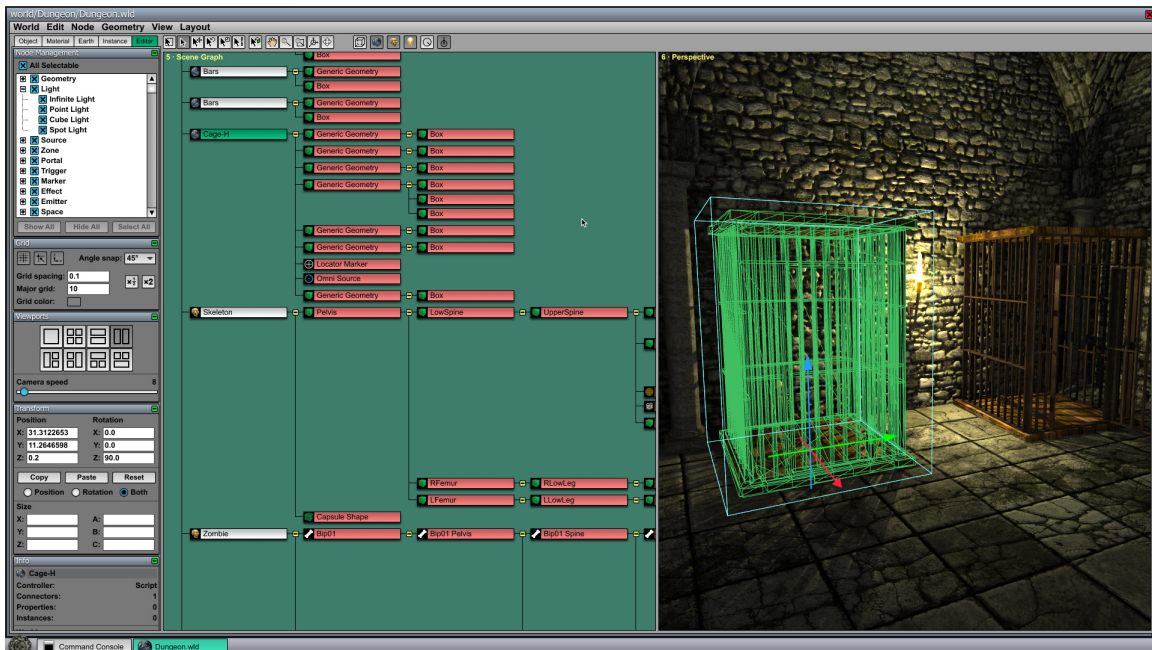
The glyph rendering method described above has been integrated into our professional-grade game engine [Lengyel 2016], and it is used to handle all text drawing needs, including user interface widgets, heads-up displays, debugging facilities, and interactive panels rendered inside the game world. Samples of these usages are shown in Figures 6 and 7. The resolution independence of our method allows text to be crisply rendered at a constant physical size in DPI-aware applications across monitors having different pixel densities. It also provides a way to render crisp text when the pixel density may not even be constant over a single glyph, as is the case for text drawn inside a game world where the camera may be viewing it at an oblique angle.

Compared to a basic text shader that does nothing more than sample glyphs from a prerendered texture map, containing either final coverage values or a signed distance field, it should be clear that our method requires considerably more computation. The cost of the flexibility and scalability provided by rendering directly from outline data was measured by filling a two-megapixel area with text drawn in a variety of fonts and recording the time needed to render it on a GeForce GTX 1060 graphics chip. In the best-performing case, a conventional shader sampling prerendered glyph images (without an SDF) requires 26  $\mu$ s to render the text, and the time required by our method is 1.1 ms, roughly 40 times as long. The performance of our method strongly depends on the complexity of the font, as determined by the typical number of Bézier curves composing a glyph, so the best case is achieved using a font like Arial that has simple outlines and no serifs. Timings for more complex fonts are listed in Table 2.





**Figure 6.** Text is rendered with our method for the heads-up display, showing information about health, score, and weapons, as well as the interactive panel embedded in the game world itself and viewed at an oblique angle.



**Figure 7.** Our method is used to render text in a world editor comprising a complex user interface that contains windows, menus, lists, check boxes, push buttons, and a variety of additional widgets. Resolution independence allows the glyphs to cleanly scale to properly match system DPI settings.

The Dobbie method [2016], which is the previously published method closest to ours in terms of algorithm design, requires 5.2 ms to render the same text in the same area using the Centaur font. Its measured time of 10.4 ms was cut in half to account for the fact that it evaluates four rays, although not a requirement of the algorithm, instead of the two used by our method. Even after this adjustment, our implementation is four times as fast, requiring only 1.3 ms.

A TrueType font needs to be preprocessed in order to generate the data format that is consumed by the glyph shader. The glyph outlines contained in a TrueType font usually have many implicit control points that require no storage. (An implicit on-curve or off-curve control point is one that falls exactly halfway between two explicit control points of the opposite type.) Because the glyph shader must be able to fetch all three controls points belonging to any quadratic Bézier curve, every control point must be included in the final data, which increases storage requirements. Furthermore, the band data needed for efficient rendering adds considerable storage requirements beyond what is found in a TrueType font. In general, we have found that the preprocessed data needed by our method is roughly twice as large to several times as large as the TrueType font from which it is derived. The size differences for a variety of fonts containing a wide range of characters are listed in Table 3 along with the dimensions of the texture maps that were generated to hold the final data.

Font	Sample	Complexity	Time
Arial	ABCDEFGFG	20	1.1 ms
Centaur	ABCDEFGFG	48	1.3 ms
Halloween	ABCDEFGFG	72	3.8 ms
Wildwood	ABCDEFGFG	546	13.3 ms

**Table 2.** A two-megapixel area was filled with 50 lines of text rendered at 32 pixels per em using a variety of fonts and timed on a GeForce GTX 1060 graphics chip. The complexity value represents the typical number of Bézier curves composing an uppercase letter in each font.

## 6 Extensions

There are a number of ways in which our glyph rendering method can be extended. In particular, it is straightforward to implement techniques that utilize multiple samples per pixel. As the ray origin is moved perpendicular to the direction in which the ray points, the values of  $a$  and  $b$  used to calculate roots in Equation (3) are invariant, so a significant amount of work can be shared over multiple samples. A simple box filter can be implemented by shifting the ray origin up and down within a pixel’s footprint for horizontal rays, or right and left for vertical rays, and averaging the coverage values calculated for each Bézier curve. Figure 8 shows the result of supersampling in this manner. Because the pixel size grows larger in em space as a font is rendered at smaller sizes, care must be taken to expand the width of the bands by half of the largest pixel size, equal to the reciprocal of



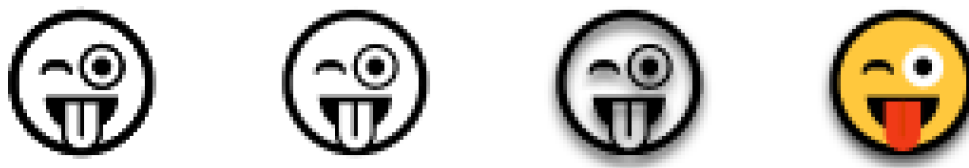
Font	Glyph Count	TTF Size (kB)	Data Size (kB)	Curve Texture Size	Band Texture Size
Wildwood	64	118	631	4096 × 6	4096 × 23
Halloween	131	57	294	4096 × 2	4096 × 17
Centaur	241	81	201	4096 × 2	4096 × 9
Arial	3223	894	1549	4096 × 11	4096 × 55
Times	3502	1085	2201	4096 × 16	4096 × 85
JhengHei	29,386	20,650	52,842	4096 × 507	4096 × 2350

**Table 3.** This table lists the number of glyphs contained in a variety of TrueType fonts and the sizes of the original .ttf files. The Data Size column lists the storage requirements of the glyph data after processing to generate the curve and band texture maps. This size also includes a small amount of per-glyph data and information about kerning, ligatures, and combining diacritical marks. The last two columns give the dimensions of the curve and band texture maps that were generated.

the smallest font size, when collecting the curves that intersect each band. Otherwise, the lists of curves belonging to each band may not be valid for all sample positions within a pixel.

By inflating the pixel footprint and applying a more sophisticated filter, effects such as a glow or drop shadow can be implemented. As with the supersampling technique, sample positions are always arranged on a line perpendicular to the ray direction. An increase in the pixel size corresponds to a decrease in the value of  $m$  used by Equations (5) and (7), and this causes the coverage gradient to be spread out over a longer distance for a softer appearance, as shown in the drop shadow in Figure 8. Again, the bands must be expanded to account for the largest effect radius so that the lists of curves are valid for each sample point.

An extension to the TrueType format facilitates multicolor glyphs by defining outlines for multiple layers that are each rendered in a different color and stacked on top of each other. (This data appears in 'COLR' and 'CPAL' tables inside a font.) This can be implemented by adding an outer loop to our pixel shader and including some extra color data in our texture maps. The result is the ability to render multicolor emoji and pictographs, as shown in Figure 8.



**Figure 8.** Emoji glyph with Unicode value U+01F61C rendered at 34 pixels per em. The leftmost image is rendered with the average coverage calculated for a horizontal ray and a vertical ray. The second image applies supersampling with three samples in each direction. The third image adds a drop shadow, and the fourth image renders six color layers.

## References

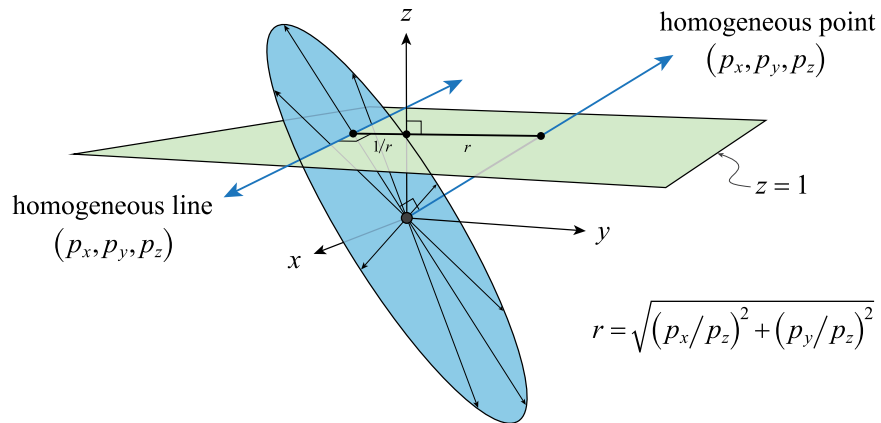
- GREEN, C. 2007. Improved Alpha-Tested Magnification for Vector Textures and Special Effects. In *SIGGRAPH 2007 Courses*, 9–18. URL: [http://www.valvesoftware.com/publications/2007/SIGGRAPH2007\\_AlphaTestedMagnification.pdf](http://www.valvesoftware.com/publications/2007/SIGGRAPH2007_AlphaTestedMagnification.pdf)
- CHLUMSKÝ, V. 2015. Shape Decomposition for Multi-channel Distance Fields. Master's Thesis, Czech Technical University. URL: <https://dspace.cvut.cz/bitstream/handle/10467/62770/F8-DP-2015-Chlumsky-Viktor-thesis.pdf>
- LOOP, C. AND BLINN, J. 2005. Resolution Independent Curve Rendering Using Programmable Graphics Hardware. In *Proceedings of ACM SIGGRAPH 2005*, ACM, 1000–1009. URL: <https://www.microsoft.com/en-us/research/wp-content/uploads/2005/01/p1000-loop.pdf>
- ESFAHBOD, B. 2012. Glyphy (software library). URL: <https://github.com/behdad/glyphy>
- DOBBIE, W. 2016. GPU Text Rendering with Vector Textures (posted on website). URL: <http://wdobbie.com/post/gpu-text-rendering-with-vector-textures/>
- LENGYEL, E. 2005. C4 Engine (software game engine). URL: <http://c4engine.com/>

# Space-Antispace Transform Correspondence in Projective Geometric Algebra

May 2022.

The concept of duality can be understood geometrically in an  $n$ -dimensional projective setting by considering both the subspace that an object occupies and the complementary subspace that the object concurrently does not occupy. The dimensionalities of these two components always sum to  $n$ , and they represent the space and antispace associated with the object. (Antispace is also known as negative space or counterspace.) The example shown in Figure 1 demonstrates the duality between homogeneous points and lines in a three-dimensional projective space. The triplet of coordinates  $(p_x, p_y, p_z)$  can be interpreted as a vector pointing from the origin toward a specific location on the projection plane  $z = 1$ . This vector corresponds to the one-dimensional space of the point that it represents. The dual of a point materializes when we consider all of the directions of space that are orthogonal to the single direction  $(p_x, p_y, p_z)$ . As illustrated by the figure, these directions span an  $(n - 1)$ -dimensional subspace that intersects the projection plane at a line when  $n = 3$ . In this way, the coordinates  $(p_x, p_y, p_z)$  can be interpreted as both a point and a line, and they are geometric duals of each other.

When we express the coordinates  $(p_x, p_y, p_z)$  on the vector basis as  $p_x \mathbf{e}_1 + p_y \mathbf{e}_2 + p_z \mathbf{e}_3$ , it explicitly states that we are working with a single spatial dimension representing a point, and the ambiguity is removed. Similarly, if we express the coordinates on the bivector basis as  $p_x \mathbf{e}_{23} + p_y \mathbf{e}_{31} + p_z \mathbf{e}_{12}$ , then we are working with the two orthogonal spatial dimensions representing a line.



**Figure 1.** The coordinates  $(p_x, p_y, p_z)$  can be interpreted as the one-dimensional span of a single vector representing a homogeneous point or as the  $(n - 1)$ -dimensional span of all orthogonal vectors representing a homogeneous hyperplane, which is a line when  $n = 3$ . Geometrically, these two interpretations are dual to each other, and their distances to the origin are reciprocals of each other.

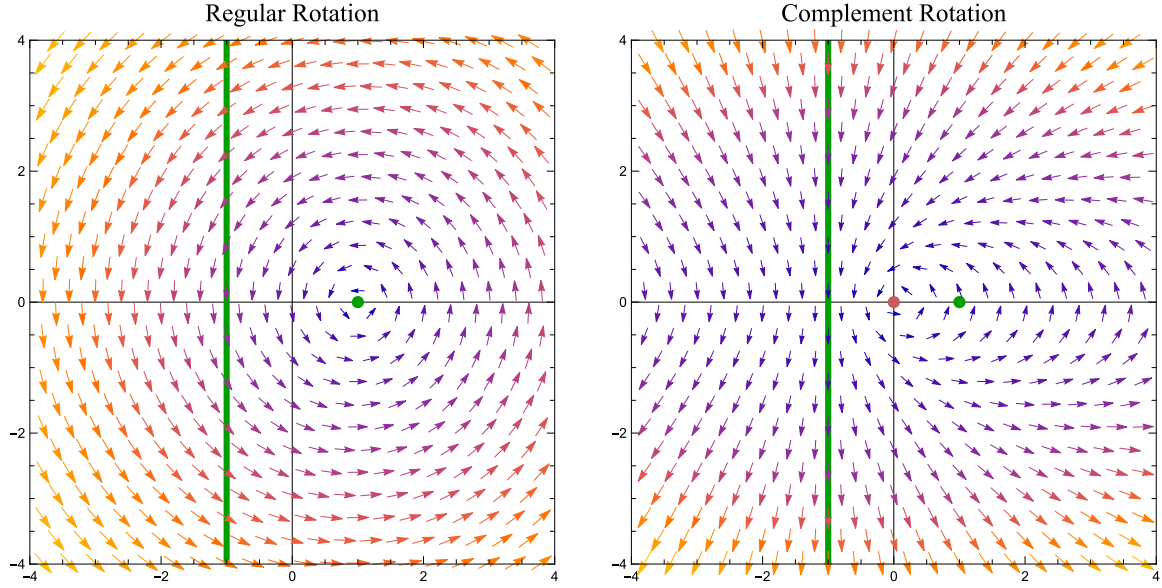
In each case, the subscripts of the basis elements tell us which basis vectors are present in the representation, and this defines the space of the object. The subscripts also tell us which basis vectors are absent in the representation, and this defines the antispace of the object. Acknowledging the existence of both the space and the antispace of any object and assigning equal meaningfulness to them allows us to explore the nature of duality to its fullest. A vector  $p_x \mathbf{e}_1 + p_y \mathbf{e}_2 + p_z \mathbf{e}_3$  is never only a point, but both a point and a line simultaneously, where the point exists in space, and the line exists in antispace. Likewise, a bivector  $p_x \mathbf{e}_{23} + p_y \mathbf{e}_{31} + p_z \mathbf{e}_{12}$  is never only a line, but both a line and a point simultaneously, where the line exists in space, and the point exists in antispace. If we study only the spatial facet of these objects and their higher-dimensional counterparts, then we are missing half of a bigger picture.

It is particularly interesting to consider the Euclidean isometries that map n-dimensional space onto itself while preserving distances and angles. We know how each isometry transforms the space of a point, line, plane, etc., but for a complete understanding of the geometry, we must ask ourselves what happens to the antispace of those objects at the same time. Equivalently, when an object is transformed by an isometry, we would like to know how its dual is transformed. The answer requires that we first look at the invariants associated with each Euclidean isometry.

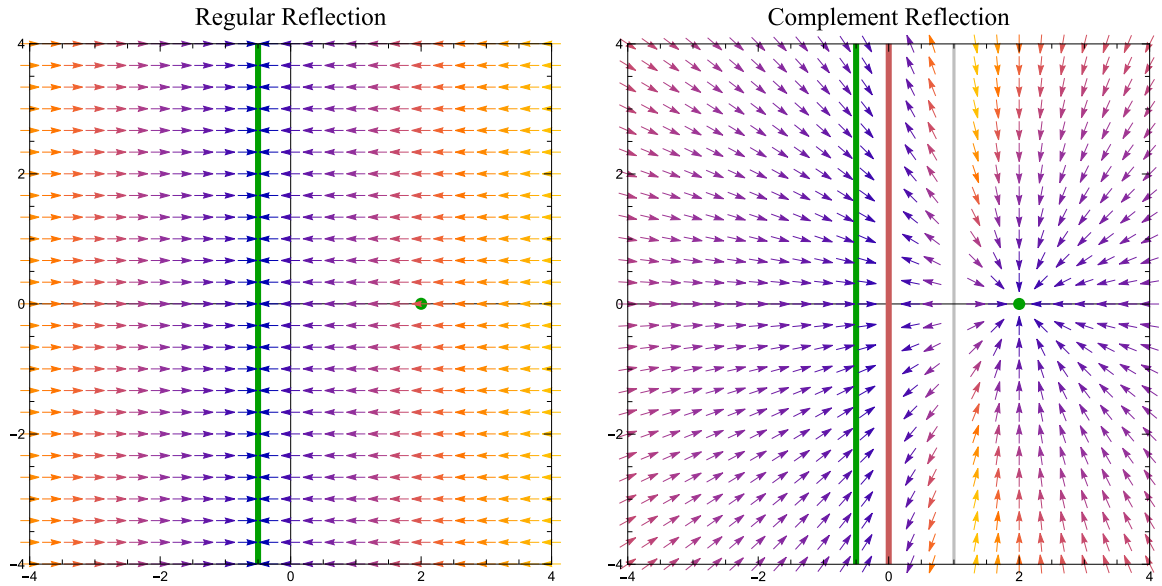
In the two-dimensional plane, the Euclidean isometries consist of a rotation about point, a translation in a specific direction, and a transfection with respect to a specific mirroring line. A reflection is a special case of transfection in which there is no motion parallel to the line, and translation is a special case of rotation in which the center lies in the horizon. Naturally, the invariant of a rotation is its center point, and the invariant of a transfection is its mirroring line. These objects are mapped onto themselves by their associated transforms, and this necessitates that their duals also be mapped onto themselves by whatever corresponding transforms occur in antispace. Because a rotation fixes its center point, the corresponding transform in antispace must fix the line that is dual to that point. And because a transfection fixes its mirroring line, the corresponding transform in antispace must fix the point that is dual to that line. These transforms occurring in antispace are reciprocal analogs of the transforms occurring in space. There is a direct correspondence between the two transforms, and they are inextricably linked. Whenever one transform is applied in space, the other is applied in antispace, and vice-versa.

Figure 2 shows a two-dimensional rotation transform and its reciprocal analog. The green point represents the center of rotation, and the green line is the dual of that point. Under the regular rotation, the center point is fixed, and under the reciprocal rotation, the line dual to the center point is fixed. These are not, however, the only fixed geometries. A regular rotation also fixes the horizon line, and thus its dual, the origin, must be fixed in the reciprocal rotation. This is illustrated by the red point in the figure, which is the focus of the various conic-section orbits. Here, the green line is the directrix.

Figure 3 shows a two-dimensional reflection transform and its reciprocal analog. The green line represents the mirroring plane of the reflection, and the green point is the dual of that line. Under the regular reflection, the mirroring line is fixed, and under the reciprocal reflection, the point dual to the mirroring line is fixed. As with rotation, there are additional fixed geometries under these transforms. A regular reflection fixes the point in the horizon in the direction perpendicular to the mirroring plane. The dual of this point is a line parallel to the mirroring line and containing the origin that remains fixed by the reciprocal reflection. This is illustrated by the red line in the figure, which is clearly a reflection boundary in a sense.



**Figure 2.** (Left) A regular rotation fixes the green center point at  $(1, 0)$  and the horizon. (Right) The corresponding complement rotation fixes the green line at  $x = -1$  dual to the center point and the origin.



**Figure 3.** (Left) A regular reflection fixes the green mirroring line at  $x = -1/2$  and the point in the horizon in the perpendicular direction. (Right) The corresponding complement reflection fixes the green point  $(2, 0)$  dual to the mirroring line and the line through the origin parallel to the mirroring line.

Finally, Figure 4 shows a two-dimensional translation transform and its reciprocal analog. As mentioned above, a translation is a special case of rotation in which the center lies in the horizon. As such, there is no finite fixed geometry that can be shown in the figure for a regular translation. However, the dual of the center in the horizon must be a line containing the origin that is fixed by the reciprocal translation, and that is illustrated by the red line in the figure. Reciprocal translation is especially important because it is the one to which we can most easily assign some practical meaning. It is a perspective projection onto the line through the origin perpendicular to the direction of translation.

In the three-dimensional projective geometric algebra  $\mathbb{R}_{2,0,1}$ , a homogeneous representation of two-dimensional space, a regular rotation about a center point  $\mathbf{c}$  is given by

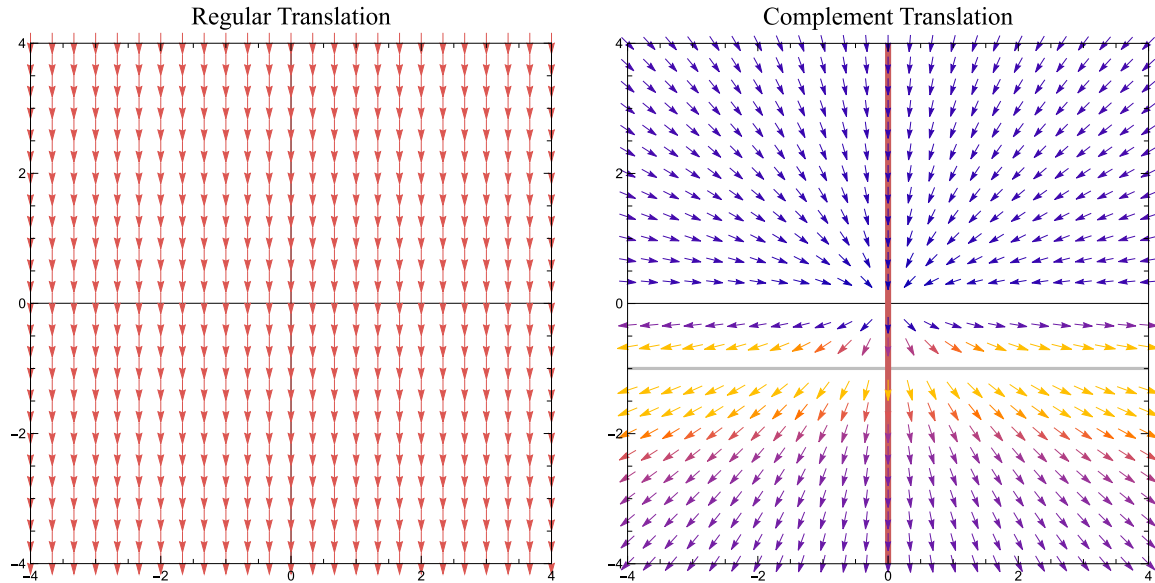
$$\mathbf{Q} = c_x \mathbf{e}_1 + c_y \mathbf{e}_2 + c_z \mathbf{e}_3 + r \mathbb{1},$$

and this becomes a translation when  $c_z = 0$ . A regular transfection across the line  $\mathbf{h}$  is given by

$$\mathbf{F} = s + h_x \mathbf{e}_{23} + h_y \mathbf{e}_{31} + h_z \mathbf{e}_{12},$$

and this becomes a pure reflection when  $s = 0$ . Together, these operators include all possible Euclidean isometries in the two-dimensional plane. Under the geometric antiproduct  $\vee$ , arbitrary products of these operators form the group  $E(2)$  with  $\mathbb{1}$  as the identity, and they covariantly transform any object  $\mathbf{a}$  in the algebra through the sandwich products

$$\mathbf{a}' = \mathbf{Q} \vee \mathbf{a} \vee \mathbf{Q} \quad \text{and} \quad \mathbf{a}' = \mathbf{F} \vee \mathbf{a} \vee \mathbf{F}.$$



**Figure 4.** (Left) A regular translation fixes the point in the horizon perpendicular to the direction of translation and every line parallel to the direction of translation. (Right) The corresponding complement translation (a perspective projection) fixes the line parallel to the direction of translation through the origin and every point in the line through the origin perpendicular to the direction of translation.

Symmetrically, a complement rotation about the line  $\mathbf{c}$  is given by

$$\mathbf{Q} = c_x \mathbf{e}_{23} + c_y \mathbf{e}_{31} + c_z \mathbf{e}_{12} - r,$$

and a complement transfection across the point  $\mathbf{h}$  is given by

$$\mathcal{F} = h_x \mathbf{e}_1 + h_y \mathbf{e}_2 + h_z \mathbf{e}_3 - s \mathbb{1}.$$

These two operators generate a different group of transformations under the geometric product  $\wedge$ . Arbitrary products of these operators form the complement Euclidean group  $\bar{\mathbf{E}}(2)$  with  $\mathbf{1}$  as the identity, and they covariantly transform any object  $\mathbf{a}$  in the algebra through the sandwich products

$$\mathbf{a}' = \mathbf{Q} \wedge \mathbf{a} \wedge \tilde{\mathbf{Q}} \quad \text{and} \quad \mathbf{a}' = \mathcal{F} \wedge \mathbf{a} \wedge \tilde{\mathcal{F}}.$$

The groups  $\mathbf{E}(2)$  and  $\bar{\mathbf{E}}(2)$  are isomorphic, and they each contain the orthogonal group  $\mathbf{O}(2)$  as a common subgroup. The complement operation provides a two-way mapping between transforms associated with members of  $\mathbf{E}(2)$  and  $\bar{\mathbf{E}}(2)$ .

The invariant geometries of the four types of transforms described above are summarized in Table 1. The Euclidean isometries always fix a coinvariant contained in the horizon, and the corresponding complement transforms always fix a coinvariant containing the origin. In general, if  $\mathbf{x}$  is the primary invariant of a Euclidean isometry, then the weight dual of  $\mathbf{x}$  gives the coinvariant. Symmetrically, if  $\mathbf{x}$  is the primary invariant of a complement transform, then the bulk dual of  $\mathbf{x}$  gives the coinvariant. When the primary invariant of a Euclidean isometry contains the origin, there is a corresponding complement transform that performs the same operation. Symmetrically, when the primary invariant of a complement transform is contained in the horizon, there is a corresponding Euclidean isometry that performs the same operation. These are where  $\mathbf{E}(2)$  and  $\bar{\mathbf{E}}(2)$  intersect at  $\mathbf{O}(2)$ .

Transform	Primary Invariant	Coinvariant
Regular rotation $\mathbf{Q} = c_x \mathbf{e}_1 + c_y \mathbf{e}_2 + c_z \mathbf{e}_3 + r \mathbb{1}$	Point $c_x \mathbf{e}_1 + c_y \mathbf{e}_2 + c_z \mathbf{e}_3$	Horizon line $\mathbf{e}_{23}$
Regular transfection $\mathbf{F} = s + h_x \mathbf{e}_{23} + h_y \mathbf{e}_{31} + h_z \mathbf{e}_{12}$	Line $h_x \mathbf{e}_{23} + h_y \mathbf{e}_{31} + h_z \mathbf{e}_{12}$	Point in horizon $h_x \mathbf{e}_1 + h_y \mathbf{e}_2$
Complement rotation $\mathbf{Q} = c_x \mathbf{e}_{23} + c_y \mathbf{e}_{31} + c_z \mathbf{e}_{12} - r$	Line $c_x \mathbf{e}_{23} + c_y \mathbf{e}_{31} + c_z \mathbf{e}_{12}$	Origin point $\mathbf{e}_3$
Complement transfection $\mathcal{F} = h_x \mathbf{e}_1 + h_y \mathbf{e}_2 + h_z \mathbf{e}_3 - s \mathbb{1}$	Point $h_x \mathbf{e}_1 + h_y \mathbf{e}_2 + h_z \mathbf{e}_3$	Line through origin $h_x \mathbf{e}_{23} + h_y \mathbf{e}_{31}$

**Table 1.** These are the invariants of transforms occurring in the 3D projective geometric algebra representing the 2D plane. The primary invariant of any regular transform (a Euclidean isometry) or complement transform is given by the vector or bivector components of the operator itself. The coinvariant is given by the weight dual of the primary invariant in the case of regular transforms and by the bulk dual of the primary invariant in the case of complement transforms.

In the four-dimensional projective geometric algebra  $\mathbb{R}_{3,0,1}$ , representing three-dimensional space, every Euclidean isometry is either a screw transform  $\mathbf{Q}$  or a rotoreflection  $\mathbf{F}$ . (The only geometrical difference between these is that the displacement along the rotation axis in a screw transform is replaced by a reflection in a plane perpendicular to the rotation axis in a rotoreflection.) The primary invariant of a screw transform is its bivector components, which corresponds to the line about which a rotation is taking place. A rotoreflection can have two primary invariants, one associated with its vector components and a second associated with its trivector components. These invariants, the invariants of the corresponding complement transforms, and the coinvariants for each are summarized in Table 2.

Transform	Primary Invariants	Coinvariants
Regular screw transform $\mathbf{Q} = r_x \mathbf{e}_{41} + r_y \mathbf{e}_{42} + r_z \mathbf{e}_{43} + r_w \mathbb{1}$ $+ u_x \mathbf{e}_{23} + u_y \mathbf{e}_{31} + u_z \mathbf{e}_{12} + u_w$	Line $r_x \mathbf{e}_{41} + r_y \mathbf{e}_{42} + r_z \mathbf{e}_{43} + u_x \mathbf{e}_{23} + u_y \mathbf{e}_{31} + u_z \mathbf{e}_{12}$	Line in horizon $r_x \mathbf{e}_{23} + r_y \mathbf{e}_{31} + r_z \mathbf{e}_{12}$
Regular rotoreflection $\mathbf{F} = s_x \mathbf{e}_1 + s_y \mathbf{e}_2 + s_z \mathbf{e}_3 + s_w \mathbf{e}_4$ $+ h_x \mathbf{e}_{423} + h_y \mathbf{e}_{431} + h_z \mathbf{e}_{412} + h_w \mathbf{e}_{321}$	Plane $h_x \mathbf{e}_{423} + h_y \mathbf{e}_{431} + h_z \mathbf{e}_{412} + h_w \mathbf{e}_{321}$ , Point $s_x \mathbf{e}_1 + s_y \mathbf{e}_2 + s_z \mathbf{e}_3 + s_w \mathbf{e}_4$	Point in horizon $h_x \mathbf{e}_1 + h_y \mathbf{e}_2 + h_z \mathbf{e}_3$ , Horizon $\mathbf{e}_{321}$
Complement screw transform $\mathbf{Q} = u_x \mathbf{e}_{41} + u_y \mathbf{e}_{42} + u_z \mathbf{e}_{43} - u_w \mathbb{1}$ $+ r_x \mathbf{e}_{23} + r_y \mathbf{e}_{31} + r_z \mathbf{e}_{12} - r_w$	Line $u_x \mathbf{e}_{41} + u_y \mathbf{e}_{42} + u_z \mathbf{e}_{43} + r_x \mathbf{e}_{23} + r_y \mathbf{e}_{31} + r_z \mathbf{e}_{12}$	Line through origin $r_x \mathbf{e}_{41} + r_y \mathbf{e}_{42} + r_z \mathbf{e}_{43}$
Complement rotoreflection $\mathcal{F} = h_x \mathbf{e}_1 + h_y \mathbf{e}_2 + h_z \mathbf{e}_3 + h_w \mathbf{e}_4$ $+ s_x \mathbf{e}_{423} + s_y \mathbf{e}_{431} + s_z \mathbf{e}_{412} + s_w \mathbf{e}_{321}$	Point $h_x \mathbf{e}_1 + h_y \mathbf{e}_2 + h_z \mathbf{e}_3 + h_w \mathbf{e}_4$ , Plane $s_x \mathbf{e}_{423} + s_y \mathbf{e}_{431} + s_z \mathbf{e}_{412} + s_w \mathbf{e}_{321}$	Plane through origin $h_x \mathbf{e}_{423} + h_y \mathbf{e}_{431} + h_z \mathbf{e}_{412}$ , Origin $\mathbf{e}_4$

**Table 2.** These are the invariants of transforms occurring in the 4D projective geometric algebra representing 3D space. The primary invariant of any regular transform (a Euclidean isometry) or complement transform is given by the vector, bivector, or trivector components of the operator itself. The coinvariants are given by the weight dual of each primary invariant in the case of regular transforms and by the bulk dual of each primary invariant in the case of complement transforms.

The unit-weight invertible elements of the  $(n+1)$ -dimensional projective geometric algebra  $\mathbb{R}_{n,0,1}$  constitute a double cover of both the groups  $E(n)$  and  $\bar{E}(n)$ . The geometric product corresponds to transform composition in the group  $\bar{E}(n)$ , and the geometric antiproduct corresponds to transform composition in the group  $E(n)$ . Regular reflections across planes are represented by antivectors (having antigrade one), and they meet at lower-dimensional invariants under the geometric antiproduct. Symmetrically, complement reflections across points are represented by vectors (having grade one), and they join at higher-dimensional invariants under the geometric product. A sandwich product

$$\mathbf{Q} \wedge \mathbf{a} \wedge \tilde{\mathbf{Q}}$$

transforms the space of  $\mathbf{a}$  with an element of  $\bar{E}(n)$ , and it transforms the antispaces of  $\mathbf{a}$  with the complementary element of  $E(n)$ . Symmetrically, a sandwich product



$$\mathbf{Q} \vee \mathbf{a} \vee \mathbf{Q}$$

transforms the space of  $\mathbf{a}$  with an element of  $E(n)$ , and it transforms the antispace of  $\mathbf{a}$  with the complementary element of  $\bar{E}(n)$ .

The groups  $E(n)$  and  $\bar{E}(n)$  have a number of subgroups, and the hierarchical relationships among them are shown in Figure 5. In particular, the Euclidean group  $E(n)$  contains the special Euclidean subgroup  $SE(n)$  consisting of all combinations of regular rotations, which are covered by the antigrade 2 elements of  $\mathbb{R}_{n,0,1}$ . Correspondingly, the complement Euclidean group  $\bar{E}(n)$  contains the special complement Euclidean subgroup  $S\bar{E}(n)$  consisting of all combinations of reciprocal rotations, which are covered by the grade 2 elements of  $\mathbb{R}_{n,0,1}$ . The subgroups  $SE(n)$  and  $S\bar{E}(n)$  further contain translation subgroups  $T(n)$  and  $\bar{T}(n)$  respectively.

Transforms about invariants containing the origin are the same in both  $E(n)$  and  $\bar{E}(n)$ , and they constitute the common subgroup  $O(n)$ . Every member of  $O(n)$  has a representation in  $\mathbb{R}_{n,0,1}$  that transforms elements with the geometric product and a complementary representation that transforms elements with the geometric antiproduct. For example, in  $\mathbb{R}_{3,0,1}$ , a conventional quaternion  $\mathbf{q}$  can be expressed as

$$\mathbf{q} = x\mathbf{e}_{23} + y\mathbf{e}_{31} + z\mathbf{e}_{12} - w\mathbf{1},$$

which covariantly transforms any object  $\mathbf{a}$  with the sandwich product  $\mathbf{q} \wedge \mathbf{a} \wedge \tilde{\mathbf{q}}$ , and it can be expressed as

$$\mathbf{q} = x\mathbf{e}_{41} + y\mathbf{e}_{42} + z\mathbf{e}_{43} + w\mathbf{1},$$

which covariantly transforms any object  $\mathbf{a}$  with the sandwich product  $\mathbf{q} \vee \mathbf{a} \vee \mathbf{q}$ .

In terms of matrix multiplication, a general element of the group  $E(n)$  transforms a point by multiplying on the left by an  $(n+1) \times (n+1)$  matrix of the form

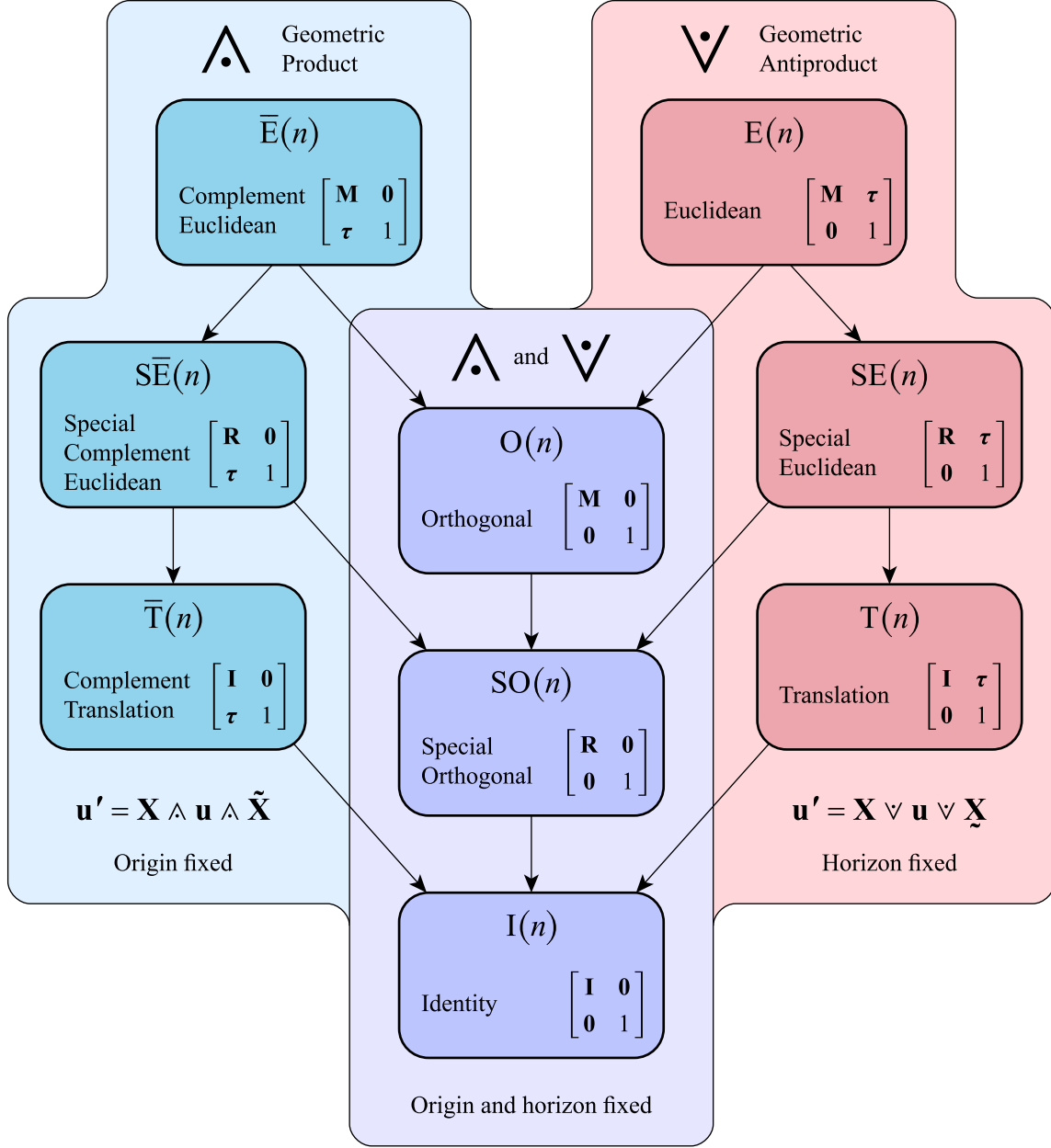
$$\begin{bmatrix} \mathbf{M}_{n \times n} & \boldsymbol{\tau}_{n \times 1} \\ \mathbf{0}_{1 \times n} & 1 \end{bmatrix},$$

where the  $n \times n$  submatrix  $\mathbf{M}$  is orthogonal. A general element of the corresponding group  $\bar{E}(n)$  transforms points with matrices of the form

$$\begin{bmatrix} \mathbf{M}_{n \times n} & \mathbf{0}_{n \times 1} \\ \boldsymbol{\tau}_{1 \times n} & 1 \end{bmatrix}.$$

In the special subgroups of  $E(n)$  and  $\bar{E}(n)$ , the submatrix  $\mathbf{M}$  has a determinant of +1. In the translation subgroups  $T(n)$  and  $\bar{T}(n)$ ,  $\mathbf{M}$  is the identity matrix. Finally, when  $\boldsymbol{\tau} = \mathbf{0}$ , both matrices have the same form and belong to  $O(n)$ .

The isomorphic mapping between  $E(n)$  and  $\bar{E}(n)$  is given by the inverse transpose operation on the matrix representatives. That is, if  $\mathbf{H}$  is an  $(n+1) \times (n+1)$  matrix representing an element of  $E(n)$ , then the corresponding element of  $\bar{E}(n)$  is given by  $(\mathbf{H}^{-1})^T$ . Of course, this operation is an involution, and the mapping works both ways.



**Figure 5.** The unit-weight invertible elements of the projective geometric algebra  $\mathbb{R}_{n,0,1}$  are a double cover of both the Euclidean group  $E(n)$  and the complement Euclidean group  $\bar{E}(n)$ . Elements corresponding to transforms in  $E(n)$  are composed with the geometric antiproduct, and elements corresponding to transforms in  $\bar{E}(n)$  are composed with the geometric product. Transforms belonging to the common subgroup  $O(n)$  have two representations in  $\mathbb{R}_{n,0,1}$ , one associated with the geometric product, and one associated with the geometric antiproduct.

# Relativistic Quaternions

December 2024.

**Abstract.** This note discusses the ways in which quaternions and dual quaternions fit as motion operators into the five-dimensional projective spacetime algebra. Euclidean isometries in three-dimensional space are combined with a temporal translation to produce physically meaningful motions in a relativistic setting. Real Lorentz invariants arise from the canonical norms of operators representing motions at subluminal velocities.

In relativity, physical quantities in 3D space like position and velocity pick up an extra coordinate in time. A position  $(x, y, z)$  in space becomes  $(ct, x, y, z)$  in spacetime, where  $c$  is the speed of light. A velocity  $(u_x, u_y, u_z)$  in space becomes  $(\gamma c, \gamma u_x, \gamma u_y, \gamma u_z)$  in spacetime, where  $\gamma$  is given by

$$\gamma = \frac{dt}{d\tau} = \frac{1}{\sqrt{1 - u^2/c^2}}$$

and accounts for things like time dilation between coordinate time  $t$  and proper time  $\tau$ . There are other types of quantities like angular momentum and the electromagnetic field that become six-component quantities in spacetime, but we're not going to talk about those right now. The goal of this note is to explore the correct way to apply quaternions and dual quaternions in a relativistic setting.

A proper understanding of quaternions in relativity, even for nothing more than basic rotations about the origin, requires that we work in the five-dimensional projective geometric algebra  $\mathbb{R}(3, 1, 1)$ . This algebra takes the 4D rigid geometric algebra  $\mathbb{R}(3, 0, 1)$  and adds a time dimension. There are five vector basis elements  $\mathbf{e}_0, \mathbf{e}_1, \mathbf{e}_2, \mathbf{e}_3$ , and  $\mathbf{e}_4$ , where the vectors  $\mathbf{e}_1, \mathbf{e}_2$ , and  $\mathbf{e}_3$  correspond to 3D space with  $\mathbf{e}_i \cdot \mathbf{e}_i = +1$  for  $i = 1, 2, 3$ , the vector  $\mathbf{e}_0$  corresponds to time with  $\mathbf{e}_0 \cdot \mathbf{e}_0 = -1$ , and vector  $\mathbf{e}_4$  is the projective dimension with  $\mathbf{e}_4 \cdot \mathbf{e}_4 = 0$ . This means that our metric tensor  $\mathbf{g}$  looks like

$$\mathbf{g} = \begin{bmatrix} -1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}.$$

There are 32 total basis elements in the exterior algebra of  $\mathbb{R}(3, 1, 1)$ . Half of them are the 16 basis elements from  $\mathbb{R}(3, 0, 1)$ , and the other half are those 16 basis elements multiplied by the time direction  $\mathbf{e}_0$ . All of the operators in  $\mathbb{R}(3, 0, 1)$  are multiplied by  $\mathbf{e}_0$  when we transfer them to  $\mathbb{R}(3, 1, 1)$  because time is an invariant of any rigid spatial motion.

Using the basis vectors of  $\mathbb{R}(3, 1, 1)$ , the spacetime position  $\mathbf{r}$  of a particle is expressed as

$$\mathbf{r} = ct\mathbf{e}_0 + x\mathbf{e}_1 + y\mathbf{e}_2 + z\mathbf{e}_3 + \mathbf{e}_4,$$

and its velocity  $\mathbf{u}$  is expressed as the derivative with respect to proper time  $\tau$ , given by

$$\mathbf{u} = \frac{d\mathbf{r}}{d\tau} = \gamma c\mathbf{e}_0 + \gamma\dot{x}\mathbf{e}_1 + \gamma\dot{y}\mathbf{e}_2 + \gamma\dot{z}\mathbf{e}_3,$$

where a dot above a variable means derivative with respect to coordinate time  $t$ .

In the projective spacetime algebra  $\mathbb{R}(3, 1, 1)$ , a conventional *quaternion*  $\mathbf{q} = q_x\mathbf{i} + q_y\mathbf{j} + q_z\mathbf{k} + q_w$  is represented by the motion operator<sup>1</sup>

$$\mathbf{q} = q_x\mathbf{e}_{410} + q_y\mathbf{e}_{420} + q_z\mathbf{e}_{430} + q_w\mathbb{1},$$

and a position  $\mathbf{r}$  (or any other quantity) is transformed by the sandwich product  $\mathbf{q} \vee \mathbf{r} \vee \mathbf{q}$ , using the geometric antiproduct just as it would be in  $\mathbb{R}(3, 0, 1)$ .<sup>2</sup> Such a quaternion can also be expressed as<sup>3</sup>

$$\mathbf{q}(\tau) = (a_x\mathbf{e}_{410} + a_y\mathbf{e}_{420} + a_z\mathbf{e}_{430})\sin\left(\frac{1}{2}\gamma\tau\dot{\phi}\right) + \mathbb{1}\cos\left(\frac{1}{2}\gamma\tau\dot{\phi}\right),$$

where  $\mathbf{a} = (a_x, a_y, a_z)$  is the unit-length axis of rotation, and  $\dot{\phi} = d\phi/dt$  is the angular velocity.

The quaternion  $\mathbf{q}(\tau)$  performs a rotation about the axis  $\mathbf{a}$  in space, but it has no effect on time coordinates, so the rotation happens instantaneously. This can't represent a physical motion because the object being rotated must arrive at its new orientation at some point in the future. To make this physical, we need to combine the quaternion with a temporal translation.

A general translation through spacetime is performed by the motion operator

$$\mathbf{T}(\tau) = \frac{1}{2}\gamma\tau(-c\mathbf{e}_{321} + \dot{x}\mathbf{e}_{230} + \dot{y}\mathbf{e}_{310} + \dot{z}\mathbf{e}_{120}) + \mathbb{1}.$$

Notice that the trivector part of this operator is  $\frac{1}{2}\tau(\mathbf{e}_4 \wedge \mathbf{u})^\star$  for a four-velocity  $\mathbf{u}$ , where the dualization  $\star$  applies the metric and causes the temporal component to be negated. To translate through time but not space, we need only the  $\mathbf{e}_{321}$  term, so the operator for a purely temporal translation is simply

$$\mathbf{S}(\tau) = \mathbb{1} - \frac{1}{2}\gamma c\tau\mathbf{e}_{321}.$$

When we multiply this by the quaternion  $\mathbf{q}(\tau)$ , we obtain a motion operator that performs both a rotation in space and a translation in time. (The two operators commute, so the order of multiplication doesn't matter.) The combined operator is given by

$$\mathbf{Q}(\tau) = \mathbf{q}(\tau) \vee \left( \mathbb{1} - \frac{1}{2}\gamma c\tau\mathbf{e}_{321} \right),$$

<sup>1</sup> The quantity  $\mathbb{1}$  is the volume element or *antiscalar* unit of the algebra given by  $\mathbb{1} = \mathbf{e}_0 \wedge \mathbf{e}_1 \wedge \mathbf{e}_2 \wedge \mathbf{e}_3 \wedge \mathbf{e}_4$ .

<sup>2</sup> As discussed in *Projective Geometric Algebra Illuminated*, Section 3.9.3, quaternions have two representations in projective geometric algebra. In the case of  $\mathbb{R}(3, 1, 1)$ , we could also express a quaternion as  $\mathbf{q} = q_w\mathbb{1} - q_x\mathbf{e}_{23} - q_y\mathbf{e}_{31} - q_z\mathbf{e}_{12}$ , and it would transform a position  $\mathbf{r}$  with the sandwich product  $\mathbf{q} \wedge \mathbf{r} \wedge \tilde{\mathbf{q}}$ . However, this form of a quaternion does not belong to the Poincaré group containing the temporal translations that we need.

<sup>3</sup> This is the exponential  $\mathbf{q}(\tau) = \exp_{\vee}\left(\frac{1}{2}\gamma\tau\dot{\phi}\mathbf{I}\right)$ , where  $\mathbf{I} = a_x\mathbf{e}_{410} + a_y\mathbf{e}_{420} + a_z\mathbf{e}_{430}$  is a line containing the origin.

and this expands to

$$\begin{aligned}\mathbf{Q}(\tau) = & (a_x \mathbf{e}_{410} + a_y \mathbf{e}_{420} + a_z \mathbf{e}_{430}) \sin\left(\frac{1}{2} \gamma \tau \dot{\phi}\right) + \mathbb{1} \cos\left(\frac{1}{2} \gamma \tau \dot{\phi}\right) \\ & - \frac{1}{2} \gamma c \tau (a_x \mathbf{e}_1 + a_y \mathbf{e}_2 + a_z \mathbf{e}_3) \sin\left(\frac{1}{2} \gamma \tau \dot{\phi}\right) - \frac{1}{2} \gamma c \tau \mathbf{e}_{321} \cos\left(\frac{1}{2} \gamma \tau \dot{\phi}\right).\end{aligned}$$

If we set  $\tau = t/\gamma$  for some time  $t$ , then the sandwich product  $\mathbf{Q}(\tau) \vee \mathbf{r} \vee \mathbf{Q}(\tau)$  rotates the position  $\mathbf{r}$  through the angle  $\dot{\phi}t$  in space and adds  $ct$  to its time coordinate. This is our relativistic quaternion, and it's an operator with eight components that we can write generically as

$$\mathbf{Q} = q_x \mathbf{e}_{410} + q_y \mathbf{e}_{420} + q_z \mathbf{e}_{430} + q_w \mathbb{1} + s_x \mathbf{e}_1 + s_y \mathbf{e}_2 + s_z \mathbf{e}_3 + s_w \mathbf{e}_{321}.$$

These eight components are not independent because each of the four  $s$  coordinates is just a copy of a corresponding  $q$  coordinate multiplied by  $-\frac{1}{2}\gamma c\tau$ , so we haven't added any new degrees of freedom.

The operator  $\mathbf{Q}$  is equivalent to the  $5 \times 5$  matrix  $\mathbf{m}$  given by

$$\mathbf{m} = \begin{bmatrix} 1 & 0 & 0 & 0 & -2(q_x s_x + q_y s_y + q_z s_z + q_w s_w) \\ 0 & 1 - 2q_y^2 - 2q_z^2 & 2(q_x q_y - q_w q_z) & 2(q_z q_x + q_w q_y) & 0 \\ 0 & 2(q_x q_y + q_w q_z) & 1 - 2q_z^2 - 2q_x^2 & 2(q_y q_z - q_w q_x) & 0 \\ 0 & 2(q_z q_x - q_w q_y) & 2(q_y q_z + q_w q_x) & 1 - 2q_x^2 - 2q_y^2 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix},$$

and the sandwich product  $\mathbf{Q}(\tau) \vee \mathbf{r} \vee \mathbf{Q}(\tau)$  is equivalent to the matrix product  $\mathbf{m}\mathbf{r}$ , treating  $\mathbf{r}$  as a column vector.<sup>4</sup> The middle  $3 \times 3$  portion of this matrix performs the conventional quaternion motion in space, and the upper-right entry performs the temporal translation. The sandwich product  $\mathbf{Q}(\tau) \vee \mathbf{x} \vee \mathbf{Q}(\tau)$  not only transforms position vectors but any multivector  $\mathbf{x}$  in the entire algebra. The matrix equivalents that transform bivectors, trivectors, and quadrivectors are given by the second, third, and fourth compound matrices of  $\mathbf{m}$ .

A *dual quaternion* performs a general rigid motion in 3D space, and it has an eight-component representation in  $\mathbb{R}(3, 0, 1)$ . In the projective spacetime algebra  $\mathbb{R}(3, 1, 1)$ , a dual quaternion becomes the motion operator<sup>5</sup>

$$\mathbf{d}(\tau) = \mathbf{l} \sin\left(\frac{1}{2} \gamma \tau \dot{\phi}\right) + \mathbb{1} \cos\left(\frac{1}{2} \gamma \tau \dot{\phi}\right) - \left(\frac{1}{2} \gamma \tau \dot{\delta} \mathbf{l}^* \wedge \mathbf{e}_0\right) \cos\left(\frac{1}{2} \gamma \tau \dot{\phi}\right) - \frac{1}{2} \gamma \tau \dot{\delta} \mathbf{e}_0 \sin\left(\frac{1}{2} \gamma \tau \dot{\phi}\right).$$

Here,  $\mathbf{l}$  is an arbitrary unitized line that may not contain the origin,  $\dot{\phi}$  continues to be the angular velocity, and  $\dot{\delta}$  is the rate of displacement along the axis of rotation. When  $\dot{\delta} = 0$ , the last two terms vanish, and when the line  $\mathbf{l}$  also passes through the origin, the dual quaternion  $\mathbf{d}(\tau)$  reduces to the quaternion  $\mathbf{q}(\tau)$ .

<sup>4</sup> If you're familiar with how 4D homogeneous coordinates represent points in 3D space, then you can think of this matrix as operating on 5D homogeneous coordinates representing points in 4D spacetime.

<sup>5</sup> This is the exponential  $\mathbf{d}(\tau) = \exp_{\vee} \left[ \frac{1}{2} \gamma \tau (\dot{\delta} \mathbf{e}_0 + \dot{\phi} \mathbb{1}) \vee \mathbf{l} \right]$ , where  $\mathbf{l}$  is an arbitrary unitized line. The expanded operator is equivalent to Equation (3.93) in *Projective Geometric Algebra Illuminated*, multiplied by  $\mathbf{e}_0$ , where  $\phi$  and  $\delta$  have been replaced by rates  $\frac{1}{2} \gamma \tau \dot{\phi}$  and  $\frac{1}{2} \gamma \tau \dot{\delta}$ .

As before, the dual quaternion  $\mathbf{d}(\tau)$  performs an instantaneous motion under the sandwich product  $\mathbf{d} \vee \mathbf{r} \vee \mathbf{d}$ , so we again need to combine it with the temporal translation  $\mathbf{S}(\tau)$  to produce a physical motion given by

$$\mathbf{D}(\tau) = \mathbf{d}(\tau) \vee \left( \mathbb{1} - \frac{1}{2} \gamma c \tau \mathbf{e}_{321} \right).$$

Expanding this product, we can write

$$\begin{aligned} \mathbf{D}(\tau) = & \mathbf{I} \sin\left(\frac{1}{2} \gamma \tau \dot{\phi}\right) + \mathbb{1} \cos\left(\frac{1}{2} \gamma \tau \dot{\phi}\right) - \left(\frac{1}{2} \gamma \tau \dot{\phi} \mathbf{I}^* \wedge \mathbf{e}_0\right) \cos\left(\frac{1}{2} \gamma \tau \dot{\phi}\right) - \frac{1}{2} \gamma \tau \dot{\phi} \mathbf{e}_0 \sin\left(\frac{1}{2} \gamma \tau \dot{\phi}\right) \\ & - \frac{1}{2} \gamma c \tau \left[ \mathbf{I} \sin\left(\frac{1}{2} \gamma \tau \dot{\phi}\right) + \mathbb{1} \cos\left(\frac{1}{2} \gamma \tau \dot{\phi}\right) \right] \vee \mathbf{e}_{321}. \end{aligned}$$

This is our relativistic dual quaternion, and it has picked up the same additional four components that we got for the quaternion  $\mathbf{Q}(\tau)$ . The 12 components can be written generically as

$$\begin{aligned} \mathbf{D} = & q_x \mathbf{e}_{410} + q_y \mathbf{e}_{420} + q_z \mathbf{e}_{430} + q_w \mathbb{1} + m_x \mathbf{e}_{230} + m_y \mathbf{e}_{310} + m_z \mathbf{e}_{120} + m_w \mathbf{e}_0 \\ & + s_x \mathbf{e}_1 + s_y \mathbf{e}_2 + s_z \mathbf{e}_3 + s_w \mathbf{e}_{321}. \end{aligned}$$

Each of the four  $s$  coordinates is a copy of a corresponding  $q$  coordinate multiplied by  $-\frac{1}{2} \gamma c \tau$  in the same way that it was for the quaternion  $\mathbf{Q}(\tau)$ .<sup>6</sup>

The bulk norm of a relativistic dual quaternion is given by

$$\|\mathbf{D}\|_{\bullet} = \sqrt{\mathbf{D} \bullet \mathbf{D}} = \sqrt{s_x^2 + s_y^2 + s_z^2 + s_w^2 - m_x^2 - m_y^2 - m_z^2 - m_w^2},$$

where  $\bullet$  is the inner product induced by the metric tensor  $\mathbf{g}$ . Assuming that  $\mathbf{D}$  is unitized, which means  $q_x^2 + q_y^2 + q_z^2 + q_w^2 = 1$  in this case, we know that  $m_x^2 + m_y^2 + m_z^2 + m_w^2$  is the square of half the distance that a particle starting at the origin gets moved through space when it's transformed by  $\mathbf{D}$ .<sup>7</sup> Since the  $s$  coordinates are just multiples of the  $q$  coordinates, we have

$$s_x^2 + s_y^2 + s_z^2 + s_w^2 = \frac{1}{4} \gamma^2 c^2 \tau^2,$$

which is the square of half the distance that a particle gets moved through time. For a physically possible motion in which a particle starting at the origin moves to a new location at less than the speed of light, the distance it is moved through time by a relativistic dual quaternion  $\mathbf{D}$  must be greater than the distance it is moved through space. This is nicely captured by the requirement that the bulk norm  $\|\mathbf{D}\|_{\bullet}$  has a real value.

When  $\dot{\phi} = 0$ , there is no rotation, and the dual quaternion  $\mathbf{D}(\tau)$  reduces to a translation operator having exactly the form of the operator  $\mathbf{T}(\tau)$  introduced earlier in this note. Taking the bulk norm, we have

$$\|\mathbf{T}(\tau)\|_{\bullet} = \frac{1}{2} \gamma \tau \sqrt{c^2 - \dot{x}^2 - \dot{y}^2 - \dot{z}^2},$$

<sup>6</sup> The set of 12-component relativistic dual quaternions  $\mathbf{D}(\tau)$  is closed under the geometric antiproduct. We get the complete set of 16-component motion operators in  $\mathbb{R}(3, 1, 1)$  when we add Lorentz boosts to the mix, but that's a story for another time.

<sup>7</sup> See *Projective Geometric Algebra Illuminated*, Section 3.6.1.

and this makes it a lot more obvious that a real bulk norm corresponds to a subluminal velocity. Setting  $dt = \gamma\tau$ ,  $dx = \dot{x} dt$ ,  $dy = \dot{y} dt$ , and  $dz = \dot{z} dt$ , we can rewrite this as

$$\|\mathbf{T}(\tau)\|_{\bullet} = \frac{1}{2} \sqrt{c^2 dt^2 - dx^2 - dy^2 - dz^2}.$$

This spacetime interval is a Lorentz invariant having the same value for all inertial observers. Interestingly, the  $(+, -, -, -)$  signature has appeared in the bulk norm despite the fact that we are using the  $(-, +, +, +)$  signature in our metric tensor  $\mathbf{g}$ . The appropriate change in signature arises naturally within the algebra to give us real values for physically meaningful quantities.





# The Transwedge Product

May 2025.

Introductory texts on geometric algebra often begin by showing how the geometric product is a combination of the wedge product and the dot product, giving us the formula<sup>1</sup>

$$\mathbf{a} \mathbf{b} = \mathbf{a} \wedge \mathbf{b} + \mathbf{a} \cdot \mathbf{b}. \quad (1)$$

However, the above formula holds only for *vectors*  $\mathbf{a}$  and  $\mathbf{b}$ . When  $\mathbf{a}$  and  $\mathbf{b}$  are allowed to assume values of higher grade, the geometric product generally yields more terms, especially in higher dimensions. When the operands have grades  $g$  and  $h$ , the geometric product can generate a result containing terms having grades  $m$  ranging from  $|g - h|$  to  $g + h$  such that the difference between each grade  $m$  and  $g + h$  is an even number. For example, the geometric product between two *bivectors*  $\mathbf{A}$  and  $\mathbf{B}$  in a 4D algebra generates components having grades 0, 2, and 4. We can decompose this product as

$$\mathbf{A} \mathbf{B} = \mathbf{A} \wedge \mathbf{B} + \mathbf{A} \times \mathbf{B} + \mathbf{A} \cdot \tilde{\mathbf{B}}, \quad (2)$$

where the  $\mathbf{A} \times \mathbf{B}$  term is called the *commutator product*, defined as  $\mathbf{A} \times \mathbf{B} = \frac{1}{2}(\mathbf{A} \wedge \mathbf{B} - \mathbf{B} \wedge \mathbf{A})$ . The commutator product corresponds to the grade-2 part of the geometric product in this case, but it generally produces more terms when the operands have higher grades. Furthermore, because the commutator product is defined in terms of the geometric product, it can't help us decompose the geometric product into independent operations. That would be circular.

When one of the operands of the geometric product is a vector  $\mathbf{a}$ , and the other operand is an arbitrary multivector  $\mathbf{B}$ , we can generalize Equation (1) a little bit as

$$\mathbf{a} \mathbf{B} = \mathbf{a} \wedge \mathbf{B} + \mathbf{B} \lrcorner \mathbf{a}. \quad (3)$$

Here, the expression  $\mathbf{B} \lrcorner \mathbf{a}$  is the *right contraction*, which is one of the interior products in Grassmann algebra. The right contraction is defined as  $\mathbf{B} \lrcorner \mathbf{a} = \mathbf{B} \vee \mathbf{a}^\star$ , where the superscript  $\star$  denotes the right dual.<sup>2</sup> When the operands of the contraction have the same grade, it reduces to the inner product, so we end up right back where we started if we insert a vector for  $\mathbf{B}$ .

What would be nice to have is a general formula that gives us each possible term generated by the geometric product using only the fundamental operations of Grassmann algebra. These operations are the exterior (wedge) product, the left and right complements, and an application of the metric. All other operations in exterior algebra can be derived from these primitives, and it is my intention to demonstrate that the geometric product can be as well. The general formula that we're looking for would need to generate a spectrum of products that include the exterior product on one

<sup>1</sup> The symbol  $\wedge$  denotes the geometric product here instead of the traditional juxtaposition because modern geometric algebra also has a geometric antiproduct denoted by  $\vee$ .

<sup>2</sup> The right dual, or more specifically the right *bulk* dual, is equivalent to the Hodge dual. We use the term right bulk dual because there is also a left bulk dual, and when we have a degenerate metric, there are separate right and left *weight* duals.

end and the interior product (contraction) on the other end, as well as anything that could arise in between. This formula would not only be able to replace the commutator product above but also give us every term of every geometric product between operands of higher grades.

I would like to introduce the *transwedge product*, which is the result of my search for a way to completely decompose the geometric product.<sup>3</sup> The transwedge product is parameterized by an order  $k$ , and it is denoted by the upward double wedge symbol  $\mathbb{A}$  with a subscript or underscript indicating that order.<sup>4</sup> The transwedge product is defined for arbitrary multivectors  $\mathbf{a}$  and  $\mathbf{b}$  as

$$\mathbf{a} \mathbb{A}_k \mathbf{b} = \sum_{\mathbf{c} \in \mathcal{B}_k} (\underline{\mathbf{c}} \vee \mathbf{a}) \wedge (\mathbf{b} \vee \mathbf{c}^\star). \quad (4)$$

It may look like there's a lot going on in this formula, but it's actually pretty straightforward stuff. First,  $\mathcal{B}_k$  is the set of all basis elements having grade  $k$ . For example, in three dimensions,  $\mathcal{B}_0 = \{\mathbf{1}\}$ ,  $\mathcal{B}_1 = \{\mathbf{e}_1, \mathbf{e}_2, \mathbf{e}_3\}$ ,  $\mathcal{B}_2 = \{\mathbf{e}_{23}, \mathbf{e}_{31}, \mathbf{e}_{12}\}$ , and  $\mathcal{B}_3 = \{\mathbf{e}_{123}\}$ . The sum is taken over all basis elements in the set  $\mathcal{B}_k$ , and each summand is a wedge product of something involving  $\mathbf{a}$  on the left and something involving  $\mathbf{b}$  on the right. The left operand,  $\underline{\mathbf{c}} \vee \mathbf{a}$ , is the left complement of  $\mathbf{c}$  within the subspace spanned by the factors of  $\mathbf{a}$ , with the sign set so that  $(\underline{\mathbf{c}} \vee \mathbf{a}) \wedge \mathbf{c} = \mathbf{a}$ . This means the left operand contains the factors of  $\mathbf{a}$  that are *not* in  $\mathbf{c}$ . If  $\mathbf{c}$  is not a factor of  $\mathbf{a}$ , then the result is zero, so many of the summands vanish. The right operand,  $\mathbf{b} \vee \mathbf{c}^\star$ , is the right contraction of  $\mathbf{b}$  with  $\mathbf{c}$ , which we could write as  $\mathbf{b} \lrcorner \mathbf{c}$ . This has the effect of removing the factors of  $\mathbf{c}$  from  $\mathbf{b}$  or producing zero if that's not possible. The dual operation in this part is where the metric enters the calculation.

When  $k = 0$ , the transwedge product reduces to the exterior product. In this case, there is only one summand, and it has  $\mathbf{c} = \mathbf{1}$ . The left complement and right dual of  $\mathbf{1}$  are both the volume element (or antiscalar)  $\mathbf{1}$ , which is the identity of the antiwedge product. Thus, the summand becomes  $\mathbf{a} \wedge \mathbf{b}$ .

Now suppose that  $\mathbf{a}$  is a simple  $m$ -vector. When  $k = m$ , the transwedge product reduces to the interior product. There is only one basis element of grade  $m$  that matches the basis element by which  $\mathbf{a}$  is multiplied, and the summand is nonzero only when  $\mathbf{c}$  is set to this particular basis element. In this case, the expression  $\underline{\mathbf{c}} \vee \mathbf{a}$  reduces to the scalar coefficient of this basis element in  $\mathbf{a}$ , which we can transfer to the expression  $\mathbf{b} \vee \mathbf{c}^\star$ . This makes the entire summand  $\mathbf{b} \lrcorner \mathbf{a}$ , which is the right contraction of  $\mathbf{b}$  with  $\mathbf{a}$ , often written as  $\mathbf{b} \lrcorner \mathbf{a}$ . When  $\mathbf{a}$  is non-simple, multiple summands produce nonzero results, one for each component of  $\mathbf{a}$ .

When  $0 < k < m$ , the transwedge product  $\mathbb{A}$  is a liminal product belonging to a transitional sequence of products between the exterior product and the interior product. For operands  $\mathbf{a}$  and  $\mathbf{b}$  having grades  $g$  and  $h$ , the transwedge product  $\mathbf{a} \mathbb{A}_k \mathbf{b}$  generates a result having grade  $g + h - 2k$ , assuming it's nonzero. Do these liminal products have any geometric meaning? We'll get back to that in a moment, but first, let's talk about the geometric product.

With the transwedge product in hand, we can now state that the geometric product is given by the sum of the transwedge products of all orders ranging from zero to the dimension  $n$  of the algebra, where we negate for orders 2 and 3 modulo 4. We can express this as

$$\mathbf{a} \mathbb{A} \mathbf{b} = \sum_{k=0}^n (-1)^{k(k-1)/2} \mathbf{a} \mathbb{A}_k \mathbf{b}. \quad (5)$$

In the case that  $\mathbf{a}$  and  $\mathbf{b}$  are vectors in a three-dimensional algebra, we can now rewrite the geometric product in Equation (1) as follows.

<sup>3</sup> John Browne pursued similar ideas in his book *Grassmann Algebra, Volume 2: Extensions*. The formulation presented here has significant differences, but I have adopted his use of the term "order".

<sup>4</sup> The symbol  $\mathbb{A}$  is the Unicode character U+2A53, and the symbol  $\mathbb{V}$  is the Unicode character U+2A54.

$$\begin{aligned}
\mathbf{a} \underset{0}{\mathbb{A}} \mathbf{b} &= \mathbf{a} \wedge \mathbf{b} \\
\mathbf{a} \underset{1}{\mathbb{A}} \mathbf{b} &= \mathbf{a} \vee \mathbf{b}^\star = \mathbf{a} \cdot \mathbf{b} \\
\mathbf{a} \wedge \mathbf{b} &= \mathbf{a} \underset{0}{\mathbb{A}} \mathbf{b} + \mathbf{a} \underset{1}{\mathbb{A}} \mathbf{b} = \mathbf{a} \wedge \mathbf{b} + \mathbf{a} \cdot \mathbf{b}
\end{aligned} \tag{6}$$

In the case that  $\mathbf{A}$  and  $\mathbf{B}$  are bivectors in a four-dimensional algebra, we can now rewrite the geometric product in Equation (2) as follows.

$$\begin{aligned}
\mathbf{A} \underset{0}{\mathbb{A}} \mathbf{B} &= \mathbf{A} \wedge \mathbf{B} \\
\mathbf{A} \underset{2}{\mathbb{A}} \mathbf{B} &= \mathbf{A} \vee \mathbf{B}^\star = \mathbf{A} \cdot \mathbf{B} \\
\mathbf{A} \wedge \mathbf{B} &= \mathbf{A} \underset{0}{\mathbb{A}} \mathbf{B} + \mathbf{A} \underset{1}{\mathbb{A}} \mathbf{B} - \mathbf{A} \underset{2}{\mathbb{A}} \mathbf{B} = \mathbf{A} \wedge \mathbf{B} + \mathbf{A} \underset{1}{\mathbb{A}} \mathbf{B} + \mathbf{A} \cdot \tilde{\mathbf{B}}
\end{aligned} \tag{7}$$

Here, the liminal product  $\mathbf{A} \underset{1}{\mathbb{A}} \mathbf{B}$  replaces the commutator product in Equation (2), and it properly generalizes to higher dimensions.<sup>5</sup>

The formula given by Equation (5) for the geometric product applies universally to *all* geometric algebras. It works in any dimension and with any metric signature, including ordinary Euclidean geometric algebra, projective geometric algebra, spacetime geometric algebra, and conformal geometric algebra. This demonstrates that exterior algebra and geometric algebra are inextricably tied together, and it's thus valid to consider them as different aspects of the same algebraic structure.

When the metric is diagonal, the geometric product between each pair of basis elements is given by exactly one of the transwedge products. This is illustrated in Table 1 for the 4D projective geometric algebra  $\mathbb{R}(3, 0, 1)$ , which shows the geometric product and color codes the corresponding transwedge products of different orders.

So what about the geometric significance of the liminal products that are neither exterior products nor interior products? Let's take a look at what we can already do in  $\mathbb{R}(3, 0, 1)$  with the wedge and antiwedge products to calculate joins, meets, and weight expansions. These are summarized in the Table 2, which draws from Tables 2.7 and 2.22 in *Projective Geometric Algebra Illuminated*. This table includes almost all of the ways that we can combine two objects that could each be a point, line, or plane in 3D space to create a new related geometry, but there is one operation that's conspicuously absent. How do we combine two skew lines  $\mathbf{l}$  and  $\mathbf{k}$  to create a new line  $\mathbf{f}$  that's perpendicular to the first two? The situation is shown in Figure 1. In *Projective Geometric Algebra Illuminated*, Section 3.6.3, the solution is to extract the axis of rotation from the motor given by the geometric antiproduct  $\mathbf{l} \vee \mathbf{k}$  and enforce the geometric constraint with Equation (3.128). Extracting the axis means selecting the grade-2 part of the geometric antiproduct, an operation that I always felt was artificial because we're overcalculating with the geometric antiproduct and throwing away some of the information in the result.

Now that we have the transwedge product, we can calculate the line  $\mathbf{f}$  directly. Like all operations in geometric algebra, the transwedge product has a corresponding antiproduct given by a

---

<sup>5</sup> It may seem like we're using a circular expression for the geometric product if you're used to defining a dual with multiplication by the volume element, as in  $\mathbf{a}^\star = \mathbf{a} \wedge \mathbf{I}$ . However, the (Hodge) dual is more properly defined using more fundamental operations as  $\mathbf{a}^\star = \overline{\mathbf{G}\mathbf{a}}$ , where the matrix product  $\mathbf{G}\mathbf{a}$  applies the metric to the multivector  $\mathbf{a}$  (which is a  $2^n \times 2^n$  matrix multiplying the  $2^n$ -dimensional column vector  $\mathbf{a}$ ), and the overline takes the right complement. The correct identity involving the geometric product that can be derived from this definition of dual is then  $\mathbf{a}^\star = \tilde{\mathbf{a}} \wedge \mathbf{I}$ , with a reversal applied to  $\mathbf{a}$ .

Transwedge Products

   $a \wedge b$ 
   $a \mathbb{A}_1 b$ 
   $-a \mathbb{A}_2 b$ 
   $-a \mathbb{A}_3 b$ 
   $a \mathbb{A}_4 b$

$a \backslash b$	1	$e_1$	$e_2$	$e_3$	$e_4$	$e_{41}$	$e_{42}$	$e_{43}$	$e_{23}$	$e_{31}$	$e_{12}$	$e_{423}$	$e_{431}$	$e_{412}$	$e_{321}$	1
1	1	$e_1$	$e_2$	$e_3$	$e_4$	$e_{41}$	$e_{42}$	$e_{43}$	$e_{23}$	$e_{31}$	$e_{12}$	$e_{423}$	$e_{431}$	$e_{412}$	$e_{321}$	1
$e_1$	$e_1$	1	$e_{12}$	$-e_{31}$	$-e_{41}$	$-e_4$	$-e_{412}$	$e_{431}$	$-e_{321}$	$-e_3$	$e_2$	1	$e_{43}$	$-e_{42}$	$-e_{23}$	$e_{423}$
$e_2$	$e_2$	$-e_{12}$	1	$e_{23}$	$-e_{42}$	$e_{412}$	$-e_4$	$-e_{423}$	$e_3$	$-e_{321}$	$-e_1$	$-e_{43}$	1	$e_{41}$	$-e_{31}$	$e_{431}$
$e_3$	$e_3$	$e_{31}$	$-e_{23}$	1	$-e_{43}$	$-e_{431}$	$e_{423}$	$-e_4$	$-e_2$	$e_1$	$-e_{321}$	$e_{42}$	$-e_{41}$	1	$-e_{12}$	$e_{412}$
$e_4$	$e_4$	$e_{41}$	$e_{42}$	$e_{43}$	0	0	0	0	$e_{423}$	$e_{431}$	$e_{412}$	0	0	0	1	0
$e_{41}$	$e_{41}$	$e_4$	$e_{412}$	$-e_{431}$	0	0	0	0	$-1$	$-e_{43}$	$e_{42}$	0	0	0	$-e_{423}$	0
$e_{42}$	$e_{42}$	$-e_{412}$	$e_4$	$e_{423}$	0	0	0	0	$e_{43}$	$-1$	$-e_{41}$	0	0	0	$-e_{431}$	0
$e_{43}$	$e_{43}$	$e_{431}$	$-e_{423}$	$e_4$	0	0	0	0	$-e_{42}$	$e_{41}$	$-1$	0	0	0	$-e_{412}$	0
$e_{23}$	$e_{23}$	$-e_{321}$	$-e_3$	$e_2$	$e_{423}$	$-1$	$-e_{43}$	$e_{42}$	$-1$	$-e_{12}$	$e_{31}$	$-e_4$	$-e_{412}$	$e_{431}$	$e_1$	$e_{41}$
$e_{31}$	$e_{31}$	$e_3$	$-e_{321}$	$-e_1$	$e_{431}$	$e_{43}$	$-1$	$-e_{41}$	$e_{12}$	$-1$	$-e_{23}$	$e_{412}$	$-e_4$	$-e_{423}$	$e_2$	$e_{42}$
$e_{12}$	$e_{12}$	$-e_2$	$e_1$	$-e_{321}$	$e_{412}$	$-e_{42}$	$e_{41}$	$-1$	$-e_{31}$	$e_{23}$	$-1$	$-e_{431}$	$e_{423}$	$-e_4$	$e_3$	$e_{43}$
$e_{423}$	$e_{423}$	$-1$	$-e_{43}$	$e_{42}$	0	0	0	0	$-e_4$	$-e_{412}$	$e_{431}$	0	0	0	$e_{41}$	0
$e_{431}$	$e_{431}$	$e_{43}$	$-1$	$-e_{41}$	0	0	0	0	$e_{412}$	$-e_4$	$-e_{423}$	0	0	0	$e_{42}$	0
$e_{412}$	$e_{412}$	$-e_{42}$	$e_{41}$	$-1$	0	0	0	0	$-e_{431}$	$e_{423}$	$-e_4$	0	0	0	$e_{43}$	0
$e_{321}$	$e_{321}$	$-e_{23}$	$-e_{31}$	$-e_{12}$	$-1$	$e_{423}$	$e_{431}$	$e_{412}$	$e_1$	$e_2$	$e_3$	$-e_{41}$	$-e_{42}$	$-e_{43}$	$-1$	$e_4$
1	1	$-e_{423}$	$-e_{431}$	$-e_{412}$	0	0	0	0	$e_{41}$	$e_{42}$	$e_{43}$	0	0	0	$-e_4$	0

**Table 1.** The geometric product between all pairs of basis elements in the 4D projective geometric algebra. Cells are color coded to identify which transwedge product contributes to the geometric product in each case.

simple De Morgan relationship. The *transwedge antiproduct* is denoted by a downward double wedge, and it is defined as

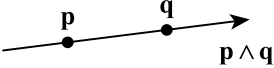
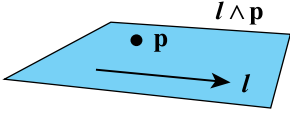
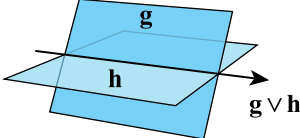
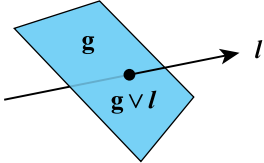
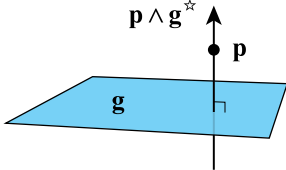
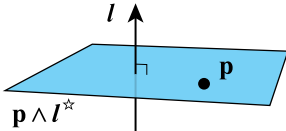
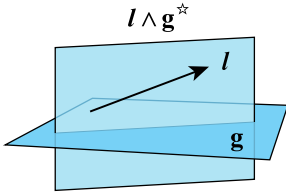
$$\mathbf{a} \Downarrow_k \mathbf{b} = \overline{\mathbf{a} \mathbb{A}_k \mathbf{b}}. \quad (8)$$

The line  $\mathbf{f}$  that is perpendicular to two skew lines  $\mathbf{l}$  and  $\mathbf{k}$  is simply given by their order 1 transwedge antiproduct, as expressed by

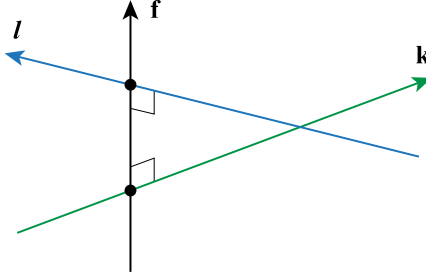
$$\mathbf{f} = \mathbf{l} \Downarrow_1 \mathbf{k}. \quad (9)$$

The line  $\mathbf{f}$  is the same as the one extracted from the geometric antiproduct, but this time we don't have to throw anything away. Keep in mind that  $\mathbf{f}$  generally needs to be adjusted with Equation (3.78) in the book in order to orthogonalize the direction and moment and make it satisfy the geometric constraint. This essentially divides by the norm of the line with respect to the ring of dual numbers instead of the reals (which is something I'd like to study further).

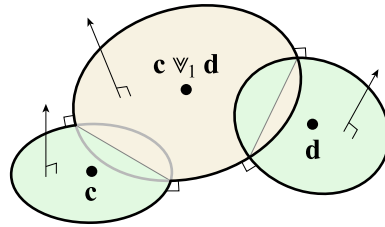
In conformal geometric algebra, Equation (9) generalizes to circles, and lines are special cases of circles that include the point at infinity. Suppose we have two circles represented by trivectors  $\mathbf{c}$  and  $\mathbf{d}$ , which can each have any arbitrary position, attitude, and radius. When we multiply them together with the transwedge antiproduct of order 1, we get a new circle  $\mathbf{c} \Downarrow_1 \mathbf{d}$  that intersects both of the circles  $\mathbf{c}$  and  $\mathbf{d}$  at right angles, as shown in Figure 2. The new circle also needs to be adjusted to satisfy the geometric constraint, but in CGA, that involves dividing by a six-component norm produced by the resulting trivector. An understanding of the exact nature of this norm will require further study.

Operation	Illustration
$\mathbf{p} \wedge \mathbf{q}$ Line containing points $\mathbf{p}$ and $\mathbf{q}$ .	
$\mathbf{l} \wedge \mathbf{p}$ Plane containing line $\mathbf{l}$ and point $\mathbf{p}$ .	
$\mathbf{g} \vee \mathbf{h}$ Line where planes $\mathbf{g}$ and $\mathbf{h}$ intersect.	
$\mathbf{g} \vee \mathbf{l}$ Point where plane $\mathbf{g}$ and line $\mathbf{l}$ intersect.	
$\mathbf{p} \wedge \mathbf{g}^\star$ Line containing point $\mathbf{p}$ and orthogonal to plane $\mathbf{g}$ .	
$\mathbf{p} \wedge \mathbf{l}^\star$ Plane containing point $\mathbf{p}$ and orthogonal to line $\mathbf{l}$ .	
$\mathbf{l} \wedge \mathbf{g}^\star$ Plane containing line $\mathbf{l}$ and orthogonal to plane $\mathbf{g}$ .	

**Table 2.** These are the ways in which two geometries can be combined in  $\mathbb{R}(3, 0, 1)$  using the join, meet, and weight expansion operations.



**Figure 1.** Two skew lines  $l$  and  $k$  are connected by a line  $f$  that meets both  $l$  and  $k$  at right angles.



**Figure 2.** Two circles  $c$  and  $d$  are multiplied together with the transwedge antiproduct of order 1. The result is a third circle that intersects both  $c$  and  $d$  at right angles.

One more practical application of the transwedge product I'd like to mention involves the computation of the geometric product in conformal geometric algebra. It's usually convenient to work in a basis for which the metric is not diagonal in CGA, but this can make it difficult to implement a geometric product for general calculations. The advice I give in *Projective Geometric Algebra Illuminated* is to transform to the basis where the metric is diagonal, compute the geometric product in that setting, and then transform back to the basis where the metric is not diagonal. Other textbooks have given this advice as well, and it's the method I used in my Mathematica packages, but it's somewhat tedious and unwieldy. The transwedge product gives us an alternative method, once we've implemented the primitive operations of the exterior algebra, that can be used to compute the geometric product explicitly without having to change back and forth between bases. After updating the Mathematica packages to use this method, it became clear that it's a vastly cleaner solution that also reduces the size of the code considerably.

### Author contact information

Eric Lengyel  
Terathon Software  
[lengyel@terathon.com](mailto:lengyel@terathon.com)



All mathematical expressions in this document were typeset with the [Radical Pie](#) equation editor.

© 2025 Eric Lengyel (the Author).

The Author provides this document (the Work) under the Creative Commons CC BY-ND 3.0 license available online at <https://creativecommons.org/licenses/by-nd/3.0/>. The Author further grants permission for reuse of images and text from the first page of the Work, provided that the reuse is for the purpose of promoting and/or summarizing the Work in scholarly venues and that any reuse is accompanied by a scientific citation to the Work.

