TERATHON SOFTWARE
3D Graphics Library

AdChoices
linode Cloud Hosting for You. Start Now

Transvoxel Algorithm
Tangent Basis
Oblique Frustum
Scissor Rectangle
Polygon Clipping
Edge List
Vector2D Class
Vector3D Class
Vector4D Class

Computing Tangent Space Basis Vectors for an Arbitrary Mesh (Lengyel's Method)

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.

Mathematical Derivation

[This derivation also appears in *Mathematics for 3D Game Programming and Computer Graphics, 3rd ed.*, Section 7.8 (or in Section 6.8 of the second edition).]

We want our tangent space to be aligned such that the x axis corresponds to the u direction in the bump map and the y axis corresponds to the 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 $\mathbf{T}$ and $\mathbf{B}$ are tangent vectors aligned to the texture map, $\mathbf{P}_0$ is the position of one of the vertices of the triangle, and (u_0, v_0) are the texture coordinates at that vertex. The letter $\mathbf{B}$ stands for *bitangent*, but in many places it is stilled called *binormal* because of a mix-up in terms when tangent-space bump mapping first became widespread. (See "Bitangent versus Binormal" below.)

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 $\mathbf{B}$ tangent 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

$$\begin{bmatrix} \\ \\ \end{bmatrix}.$$

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

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

$$\mathbf{T}' = \mathbf{T} - (\mathbf{N} \cdot \mathbf{T})\mathbf{N}$$

$$\mathbf{B}' = \mathbf{B} - (\mathbf{N} \cdot \mathbf{B})\mathbf{N} - (\mathbf{T}' \cdot \mathbf{B})\mathbf{T}' / T'^2$$

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

$$\begin{bmatrix} T'_x & T'_y & T'_z \\ B'_x & B'_y & B'_z \\ N_x & N_y & N_z \end{bmatrix} \qquad (*)$$

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 (*). One may 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 programs by avoiding the need to specify an additional array containing the per-vertex m values.

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*.

Source Code

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

```
#include "Vector4D.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;
}
```

How to cite this article

Lengyel, Eric. “Computing Tangent Space Basis Vectors for an Arbitrary Mesh”. Terathon Software 3D Graphics Library, 2001. http://www.terathon.com/code/tangent.html

Copyright © 2001 - 2015, Terathon Software LLC