



SOCLAB

SOCIETIES OF COMPUTATION LABORATORY

# OPEN RENDER BUMP

---

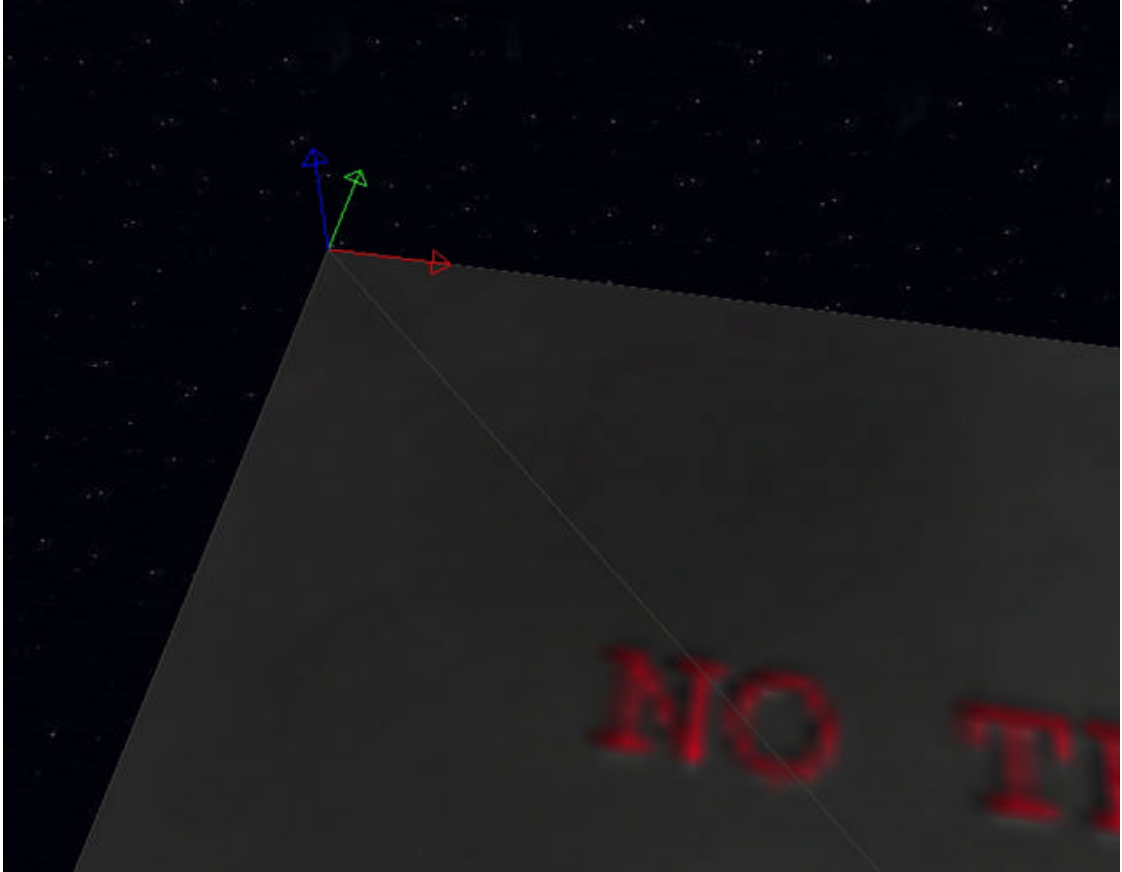
TANGENT SPACE VECTOR CALCULATION

---

## 1. TANGENT SPACE

---

Bump mapping is usually calculated in tangent space. Tangent space, just like object space, is an orthonormal basis defined with three vectors X, Y, Z (see Figure 1). The X-vector corresponds to the U/S direction in texture coordinates (**red**), the Y-vector corresponds to the V/T direction (**green**), and the Z-vector corresponds to the smoothed vertex normal (**blue**). *Tangent space vectors is often called tangent, bi-normal, and normal, however, in this document they're named as they are in object or world space.*



**Figure 1. Tangent space at a vertex.**

---

## 2. SMOOTHING NORMALS

---

- The first thing we need is the face normal. This can either be read from model-file or calculated at load time.
- Secondly, for each vertex in the mesh, do this
  - Average the face-normals of each face that the current vertex is connected to, and that share the same smoothing group(s). The result becomes the (smoothed) vertex normal.
- This process can be very slow in a naïve implementation, but can easily be optimized. Before the smoothing begins, build a face-list for each vertex, i.e. the vertex has a list of all the faces it is connected to. Then the order of the algorithm becomes a single loop-thru of the n vertices.
- Averaging normals at sharp edges may give poor results, therefore it is recommended to put a threshold on the averaging. I.e. only average the face-normals if their difference in degrees is less than 90-120. *This degree setting is also the way smoothing works in the modeling package Lightwave 3D.* In some tests a setting of 120 proved to give the best result in a variety of cases. ORB uses 120 degrees if not an angle is supplied from a Lightwave model, in that case, the imported value is used.

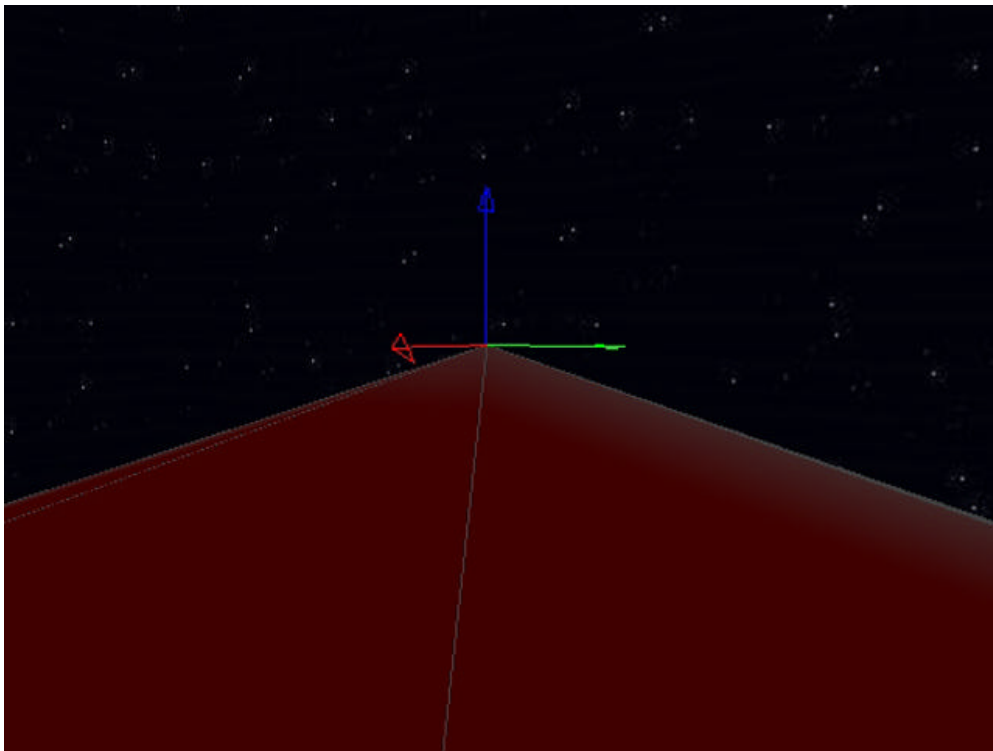


Figure 2. Smoothed vertex normal (blue).

---

### 3. TANGENT VECTORS CALCULATION

---

The **Z** vector of the vertex tangent space was calculated during the normal smoothing (2.). Now we need the **X**, and the **Y** vectors at each vertex. To calculate the tangent X and Y vectors, you will need three functions: `CalculateTriangleBasis()`, `Orthogonalize()`, and `ClosestPointOnLine()`. The code for these functions is included in the Appendix.

For each triangle **tri** in the mesh, do this:

- Get the vertices V0, V1, V2 (type VECTOR, float[3])
- Get the vertex tex-coords (v0s, v0t), (v1s, v1t), (v2s, v2t) (type float)
- Have two vectors (facetangentX, and facetangentY) to store the result in
- Send the data to the `CalculateTriangleBasis()` function:

```
o CalculateTriangleBasis(V0, V1, V2, v0s, v0t, v1s, v1t, v2s, v2t,
    &facetangentX, &facetangentY);
```

- Now we have the tangent vectors (facetangentX, and facetangentY) **for the face!**

In order to get the tangents **for a vertex**, we need to average the result of the tangent spaces of all the faces that is connected to that vertex (and share smoothing group). For each vertex **E** do:

- **Example:** a vertex **E** has 6 faces connected to it. We calculated the tangent vectors for each of these faces in the previous step (facetangentX, facetangentY) (see Figure 3). We then average the tangents of all the 6 faces to get the tangent space for vertex **E**:

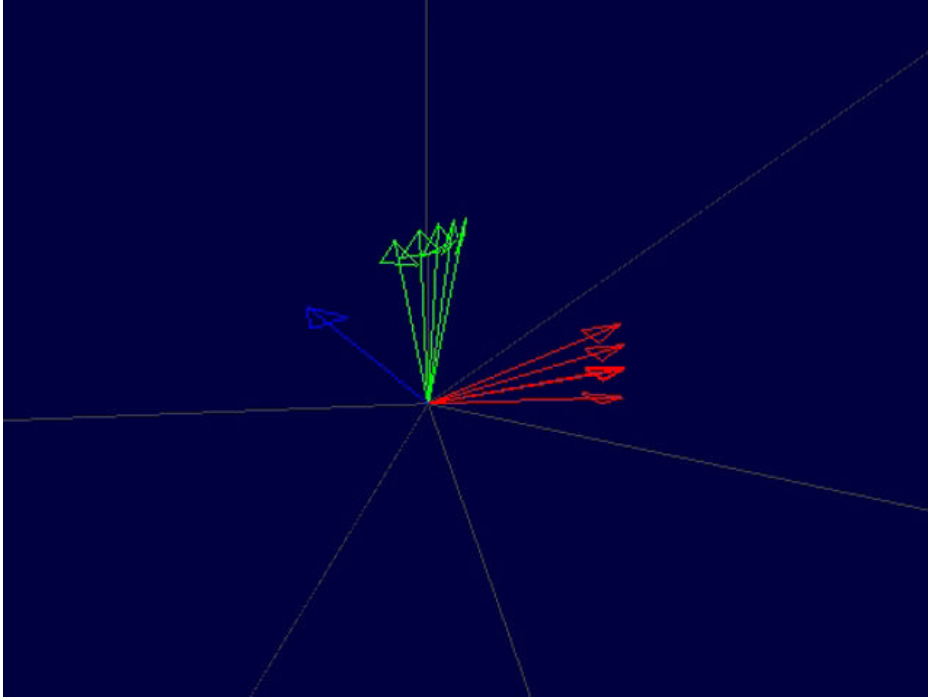
- o **E's** tangentX = (face0.facetangentX + face1.facetangentX + face2.facetangentX + face3.facetangentX + face4.facetangentX + face5.facetangentX) / 6; (see Figure 4)

- o But, only average if the tangents are less than 90 degrees apart, otherwise, there is discontinuous texture mapping, and the vertex should be split anyway (this depends on texture coords is supplied by face or by vertex). The same goes for discontinuities in smoothing groups.

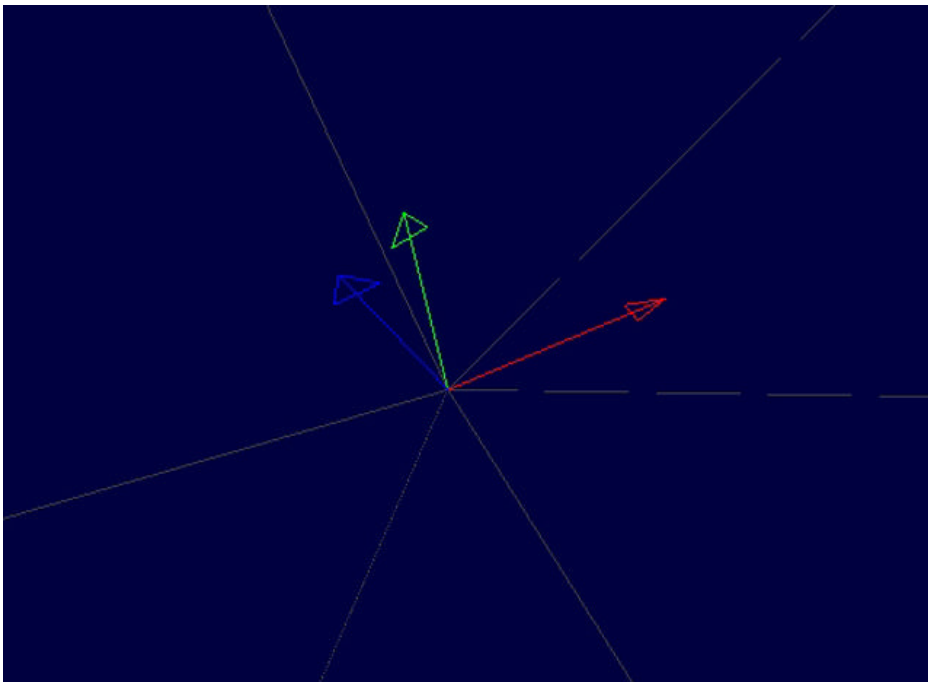
- o Finally, we have to correct the tangents so they make an orthonormal basis that fit the smoothed vertex normal. We do this with Gram-Schmidt Orthogonalization [1]. This is what the `Orthogonalize()` function does.

```
o vertE->tangentX = Orthogonalize(vertE->normal, vertE->tangentX);
```

```
o vertE->tangentY = Orthogonalize(vertE->normal, vertE->tangentY);
```



**Figure 3. Tangents X and Y (red and green) calculated for the six faces that this vertex is connected to.**



**Figure 4. Tangents X,Y, are now averaged /“smoothed” and orthogonalized.**

---

## APPENDIX A. CALCULATE TRIANGLE BASIS FUNCTION

---

This function was derived from the technique presented in “Mathematics for 3D Game Programming & Computer Graphics” by E. Lengyel [1].

```
void CalcTriangleBasis(VECTOR& E, VECTOR& F, VECTOR& G, float sE, float tE, float sF,
float tF, float sG, float tG, VECTOR * tangentX, VECTOR * tangentY)
{
    VECTOR P = F - E;

    VECTOR Q = G - E;

    float s1 = sF - sE;

    float t1 = tF - tE;

    float s2 = sG - sE;

    float t2 = tG - tE;

    float pqMatrix[2][3];

    pqMatrix[0][0] = P.vector[0];
    pqMatrix[0][1] = P.vector[1];
    pqMatrix[0][2] = P.vector[2];

    pqMatrix[1][0] = Q.vector[0];
    pqMatrix[1][1] = Q.vector[1];
    pqMatrix[1][2] = Q.vector[2];

    float temp = 1 / (s1*t2 - s2*t1);

    float stMatrix[2][2];

    stMatrix[0][0] = t2 * temp;

    stMatrix[0][1] = -t1 * temp;

    stMatrix[1][0] = -s2 * temp;
```

```

stMatrix[1][1] = s1 * temp;

float tbMatrix[2][3];

// stMatrix * pqMatrix

tbMatrix[0][0] = stMatrix[0][0]*pqMatrix[0][0] + stMatrix[0][1]*pqMatrix[1][0];
tbMatrix[0][1] = stMatrix[0][0]*pqMatrix[0][1] + stMatrix[0][1]*pqMatrix[1][1];
tbMatrix[0][2] = stMatrix[0][0]*pqMatrix[0][2] + stMatrix[0][1]*pqMatrix[1][2];
tbMatrix[1][0] = stMatrix[1][0]*pqMatrix[0][0] + stMatrix[1][1]*pqMatrix[1][0];
tbMatrix[1][1] = stMatrix[1][0]*pqMatrix[0][1] + stMatrix[1][1]*pqMatrix[1][1];
tbMatrix[1][2] = stMatrix[1][0]*pqMatrix[0][2] + stMatrix[1][1]*pqMatrix[1][2];

tangentX->Set(tbMatrix[0][0], tbMatrix[0][1], tbMatrix[0][2]);

tangentY->Set(tbMatrix[1][0], tbMatrix[1][1], tbMatrix[1][2]);

tangentX->Normalize();

tangentY->Normalize();

}

```

---

## APPENDIX B. ORTHOGONALIZE FUNCTION

---

```
VECTOR Orthogonalize(VECTOR v1, VECTOR v2)

{

    VECTOR v2ProjV1 = ClosestPointOnLine(v1, -v1, v2);

    VECTOR res = v2 - v2ProjV1;

    res.Normalize();

    return res;

}


VECTOR ClosestPointOnLine(VECTOR & a, VECTOR & b, VECTOR & p)

{

    // a-b is the line, p the point in question

    VECTOR c = p-a;

    VECTOR V = b-a;

    float d = V.Length();

    V = V | 1.0f;      // normalize V

    float t = V % c;    // V dot c

    // Check to see if t is beyond the extents of the line segment

    if (t < 0.0f)

    {

        VECTOR res = a;

        return res;

    }

    if (t > d)

    {
```



```

        VECTOR res = b;

        return res;
    }

    // Return the point between a and b

    V = V | t;      //set length of V to t.

    VECTOR res = a + V;

    return res;
}

```

---

## REFERENCES

---

[1] Lengyel, "Mathematics for 3D Game Programming & Computer Graphics", Charles River Media, 2002.

---

## CONTACT

---

Christian Seger

Societies of Computation Laboratory (SOCLAB),

Department of Software Engineering and Computer Science,

Blekinge Institute of Technology,

Box 520, Ronneby - Sweden.

[christian.seger@bth.se](mailto:christian.seger@bth.se)