Towards a set of techniques to implement Bump Mapping

Márcio da Silva Camilo, Bernardo Nogueira S. Hodge, Rodrigo Pereira Martins, Alexandre Sztajnberg
Departamento de Informática e Ciências da Computação
Universidade Estadual do Rio de Janeiro
{pmacstronger, bernardohodge, rodrigomartins, alexszt}@ime.uerj.br

Abstract.

The revolution of three dimensional video games lead to an intense development of graphical techniques and hardware. Texture-mapping hardware is now able to generate interactive computer-generated imagery with high levels of per-pixel detail. Nevertheless, traditional single texture techniques are not able to simulate bumped surfaces decently. More traditional bump mapping techniques, such as Emboss bump mapping, most times deploy an undesirable or 'fake' appearance to wrinkles in surfaces. Bump mapping can be applied to different types of applications varying form computer games, 3d environment simulations, and architectural projects, among others. In this paper we will examine a method that can be applied in 3d game engines, which uses texture-maps to generate bump surfaces in a flat polygon, called Dot3 bump mapping. This method is based on normal-perturbation technique and a vectorial operation performed at each pixel which results on a correct light calculation, and therefore great visual quality. We will present the foundations of the bump mapping method, and a technique proposal to apply it in a systematic form. We have used this technique in the implementation of a game engine that will be used as a case study.

1. Introduction

Irregular surfaces are a problem in real-time rendering for they have much more geometrical complexity than a flat surface, leading to a performance decrease of the application.

Photorealistic rendering is a general aim in most games today but real surfaces present a lot of bumps, geometry addition is an overhead so great that most present day hardware is not able to deal with. Single texture mapping is not able to simulate wrinkles in a surface, in the best case, a photo-real wrinkled texture can be assigned to a polygon, but these textures won't have a different light calculation for the wrinkled part of it, as it should for a realistic result.

Blinn [3] invented the bump mapping towards solving this problem. Bump mapping is a normal-perturbation rendering technique for simulating lighting effects caused by irregularities on smooth surfaces. It can be observed that bump mapping simulates a surface's irregular lighting appearance without modeling its irregular patterns as true geometric perturbations, increasing the model's polygon count, hence decreasing applications' performance.

There are several other techniques for implementing bump mapping. Some of them, such as Emboss Mapping, do not result in a good appearance for some surfaces, in general because they use a too simplistic approximation of light calculation [4]. Others, such as Blinn's original bump mapping idea, are computationally expensive to calculate in real-time [7]. Dot3 bump mapping deploys a great final result on surface appearance and is feasible in today's hardware in a single rendering pass. It is based on a mathematical model of lighting intensity and reflection calculated for each pixel on a surface to be rendered.

Our Dot3 bump mapping implementation is based on normal perturbation encoded as RGB values on a texture-map. This texture-map then can be fetched from an image file and applied to a polygon using multi-textures technique. For correct pixel color calculation, a Dot3 product is performed between the encoded normal and the light vector. This operation is a very decent approximation of real-world light calculations and gives us a great final result.

We are going to present our implementation of Dot3 bump mapping applied to a 3D game engine renderpath. Our Dot3 implementation is based on OpenGL fixed pipeline programming, OpenGL extensions, and Tangent space transformation, which will also be discussed in this paper. Advantages and disadvantages of our Dot3 bump mapping implementation compared to others will be discussed along the sections.

In the next section, Section 2, we present an introduction to bump mapping and show how to store, fetch and use normals for bump surfaces simulation. In Section 3 we discuss the mathematics of Dot3 bump mapping. In Section 4 we discuss Tangent space calculation, a very important tool for dealing with vectorial space issue in Dot3 bump mapping. Section 5 is dedicated to a case study of our Dot3 bump mapping implementation in a 3D first person shooter game engine. Our conclusions are presented on section 6.

2. Dot3 Bump Mapping and normals

Games nowadays demand very high detail in graphics, trying to simulate real world environments. The problem is that real world surfaces are bumped, irregular and share not much similarity with flat planes.

In a first attempt, we have the idea of creating these bumps and irregularities by modeling them with traditional model creation. This approach can increase geometry count of the models so much that is practically impossible to be implemented in today's hardware.

Dot3 Bump Mapping principle is based on the fact that in the real world our perception of a bump in an irregular surface if given by the different lightning compared to a flat surfaces. This perception is proportional to the normal vector and intensity of the light reflected by the surface. These parameters can be variable depending on the surface irregularity.

Light calculation is, thus, defined by normals. In a graphic application, normals can used to simulate bumped surfaces in a flat plane. Figure 1 describes the situation.

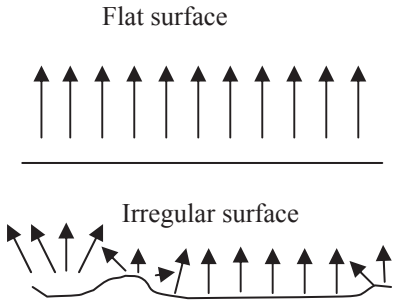


Figure 1 - normals on flat and bumped surfaces

Additionally it is straight forward the possibility to simulate bumps in a flat surface if we can artificially introduce perturbation to the normals, as in Figure 2.

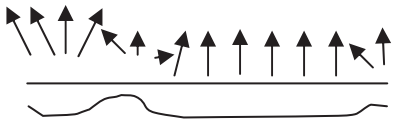


Figure 2 - This Figure summarizes what Bump mapping technique is

The classic formulation of bump mapping developed by Blinn [3], based on the normal perturbation technique, computes perturbed surface normals for a surface as if a height field was displaced beneath the unperturbed surface. The surface is then rendered and illuminated based on the perturbed surface normals. These computations are performed at each and every visible pixel on the surface.

3. Mathematics of Dot3 Bump mapping

The mathematical foundation of Dot3 bump mapping is based upon linear algebra operation dot product between three-dimensional vectors.

Dot3 bump mapping is based on two tasks, the calculation of a perturbed normal for a polygon and then the light calculation with that normal. Each of these operations must be performed at a fragment level (in our case, the fragment level is a pixel), in a way that the light will be calculated in a per-pixel basis. The pixel normal information is obtained from the texture map, that can be generated by a graphical software such as Adobe Photoshop™. In this file the normals are encoded as the RGB values of the texture itself.

For the light calculation in Dot3 bump mapping we use the Phong [1] model. Light intensity of a given fragment in Phong model is given by equation (1):

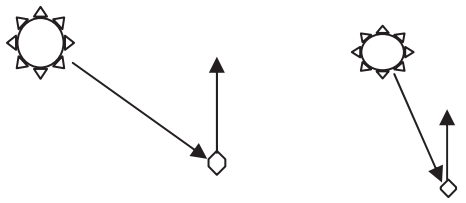
$$Intensity = Ambient + Diffuse * (N \bullet L) + Specular * (R \bullet V)\ ^{\wedge}\ n \quad (1)$$

Where:
L is the light vector
N is the normal vector of a surface
R is the reflection vector
V is view position vector
N is the intensity of specular contribution.
Ambient is the ambient component light intensity
Diffuse is the diffuse component light intensity
Specular is the specular component light intensity

The basic operation we must perform is the Dot3 operation between the normal of the surface and the light vector. Dot3 is a mathematical tool that allows us to calculate the angle between two vectors. Dot3 product is defined as (2):

Being s and v two three dimensional vectors with coordinates (x,y,z). Let Theta be the angle between these vectors,

$$s \bullet v = (s.x*v.x + s.y*v.y + s.z*v.z) \quad (2)$$
$$\cos Theta = (s \bullet v) / |s|\ |v|$$



(a)Open angle light incision (b) Narrow angle incision

Figure 3 - Light and normal vector disposition

It is simple to realize that in real world, greater the angle between the light source and the normal of target

point to be lit, smaller will be the light intensity on this point. This is the mathematical foundation for choosing Dot3 operation as the light calculation operation. Figure 3 gives us a better idea.
The Dot3 operation perfectly fits in this description since if the angle between the two vectors increases, their cosine decreases (considering angles in the same quadrant).
Using the light model described above, is easy to realize that if the Dot3 product results in a near zero or zero value, for great angle values, fragment intensity will depend on ambient contribution, which is a global illumination component that affects all objects equally on a scene.

The second part of the equation deals with specular component calculation. This term adds some specular lighting that is some shininess when the light is directly reflected in direction of the camera. This can be easily explained: If light reflects in direction of the camera the (R•V) term will result a high value that will be added to the fragment's intensity value. *n* is a constant that defines how much specular component will affect fragment's intensity. Blinn´s original formulation uses V angle for specular highlights, our implementation uses the Half-angle vector because it is easier to calculate and achieves similar results.

Dot3 is the operation that allows Dot3 bump mapping method to achieve great visual quality. Other forms of bump mapping, as Emboss bump mapping, are based on different light calculations, most of times not as much accurate for a good final result.

4. Tangent Space

Dot3 bump mapping is based on a vectorial operation between light vector and an encoded normal. As every vectorial operation, the vectorial space issue is fundamental since it can lead to errors on calculations and have a strong influence on applications performance.

Although Tangent space is not a necessary part of dot3 bump mapping process, its implementation can lead to significantly performance enhancement as well as codification simplicity.

4.1 Vectorial Spaces

Dot3 bump mapping is based on vectorial manipulation of normal and light vector. A very important issue when dealing with vectors is the vectorial space they belong. Traditional Dot3 operation can be applied to every vector composed by three coordinates, but unless they are under the same vectorial space, the operation result will be wrong. To understand this better, consider this example:

Let s = (1,2,6) be a vector on a canonical (1,1,1) space. Considering the W = (x,2y,3z) vectorial space,

(1,0,0) = (x,2y,3z)
(0,2,0) = (x,2y,3z)
(0,0,6) = (x,2y,3z)

x=1 ; y = 1 ; z = 2

s will be written as (1,1,2) under W vectorial space.

Performing any vectorial operation between *s* and a vector that belongs to *W* vectorial space should be performed as *sw = (1,1,3)*, and is very easy to see that if the operation is performed with *s* under canonical coordinates the result will be incorrect as inequation (3) states.

$$(1,2,6) \bullet (\alpha,\beta,\chi) <> (1,1,2) \bullet (\alpha,\beta,\chi) \quad (3)$$

OpenGL rendering is based on vectorial spaces for rendering; two of the most important are Eye space and Object space. The camera or 'eye' is under the Eye space. Polygons to be rendered lie on Object space. As mentioned above, a vector transformation is necessary to correctly compute them in one vectorial space. A matrix multiplication for writing a vector in different vectorial spaces (each space has an associated matrix) can be used for this transformation. OpenGL uses the Modelview-Projection (MVP) matrix for correctly computing space coordinates and placing them on screen [6].

Both spaces can be used for Dot3 bump mapping. As long as we are working with textures for generating bumps, it becomes natural to use a texture based vectorial space, since embedded normal vectors belong to this space. Next subsection introduces some Tangent Space concepts.

4.2 Tangent Space

Tangent space is also known as texture space, for it matches the homogeneous (UV) coordinates on a texture.

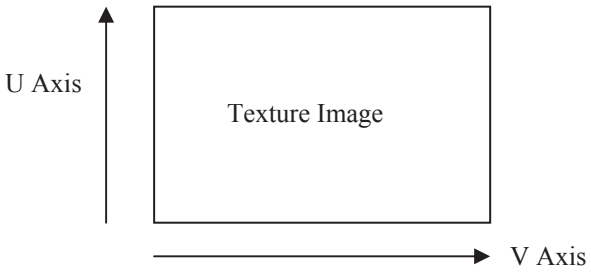


Figure 4 - Homogeneous coordinates on a texture image, also the coordinates of Tangent space

Several are the reasons for the choice of this vectorial space. In Eye space, for instance, every time the camera changes its position we need to recalculate and renormalize vectors. In Object space, by its turn, it is not necessary calculating Tangent and Binormal vectors, but we cannot re-use textures that share UV mapping coordinates and, besides, every time we rotate an object we need to change the texture map that encode the normals, since they will be pointing to a wrong direction [5]. Tangent space in particular is well suited to be used with techniques that use several model rotations and

translations such as Skeletal Animation for the reasons explained above.

There are several ways for calculating Tangent Space. Our implementation of Dot3 is based on Binormal and Tangent vectors. The Binormal vector follows the V coordinate increasing direction. Tangent vector is U's coordinate correspondent vector. Tangent space calculation can be divided in two steps: the computation of Tangent and Binormal vectors for each vertex and the calculation of a normal for these vertices. The procedure for Tangent space calculation is described in the sequence.

Algorithm 1

Considering X the cross product operation between two vectors. This operation results on a vector orthogonal to the other two vectors. Considering triangle-based models composed by 3 vertices v1,v2,v3 with x,y,z coordinates as well as UV mapping coordinates. Binormal and Tangent vectors for each vertex also composed by (x,y,z) coordinates.

```
for each triangle on a scene to be rendered {

    //Cross product
   (x, y, z) =
      (v2.x - v1.x, v2.u - v1.u, v2.v - v1.v) X
      (v3.x - v1.x, v3.u - v1.u, v3.v - v1.v)
   if (x!=0) {
     // binormal is a unit length vector
     NORMALIZEVECTOR(x,y,z)

    //Writing the x component of Tangent and
    //Binormal vectors based on partial
    //derivative theory definition
     v1->Tangent.x += -y/x;
     v1->Binormal.x += -z/x;
     v2->Tangent.x += -y/x;
     v2->Binormal.x += -z/x;
     v3->Tangent.x += -y/x;
     v3->Binormal.x += -z/x;
   }

    //repeating the procedure for y coordinate
   (x, y, z) =
       (v2.y - v1.y, v2.u - v1.u, v2.v - v1.v) X
       (v3.y - v1.y, v3.u - v1.u, v3.v - v1.v)
   if (x!=0) {
     NORMALIZEVECTOR(x,y,z)
     v1->Tangent.y += -y/x;
     v1->Binormal.y += -z/x;
     v2->Tangent.y += -y/x;
     v2->Binormal.y += -z/x;
     v3->Tangent.y += -y/x;
     v3->Binormal.y += -z/x;
   }

    //repeating the procedure for z coordinate
   (x, y, z) =
       (v2.z - v1.z, v2.u - v1.u, v2.v - v1.v) X
       (v3.z - v1.z, v3.u - v1.u, v3.v - v1.v)
   if (x!=0) {
     NORMALIZEVECTOR(x,y,z)
     v1->Tangent.z += -y/x;
     v1->Binormal.z += -z/x;
     v2->Tangent.z += -y/x;
     v2->Binormal.z += -z/x;
     v3->Tangent.z += -y/x;
     v3->Binormal.z += -z/x;
   }
}
```

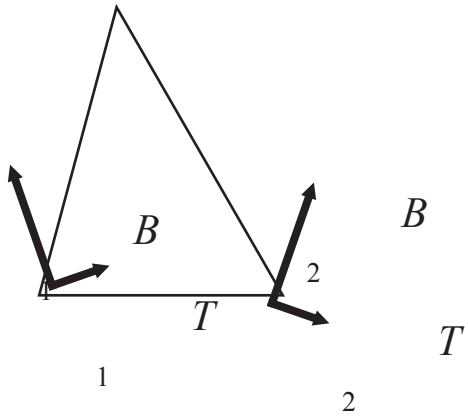


Figure 5 - Example of Tangent and Binormal vectors

Algorithm 1 computes Binormal and Tangent vectors for each vertex of each triangle on scene to be rendered. It is based on partial derivative theory that defines Tangent and Binormal vectors[2]. Figure 5 exemplifies the final result of this procedure. The next procedure is calculating the normal vector for each vertex. By definition, Binormal and Tangent vectors follow the same direction of the triangle surface. This way the normal vector is orthogonal to these vectors as it is orthogonal to the triangle itself. Algorithm 2 computes the normal vector. After normal vector creation we have the 3 linearly independent vectors used as a basis for Tangent space.

Algorithm 2

Consider N the orthogonal vector to Tangent and Binormal, '• ' is a dot product operation, T is the Tangent vector, B the Binormal vector, and Normal the Object space vertex normal.

```
for each vertex
{
    // vector normalization
    NORMALIZEVECTOR(B);
    NORMALIZEVECTOR(T);

    //cross product between tangent and binormal
    N= Tangent X Binormal;

    //correct orientation check
    if N • Normal < 0
       N = - N;
}
```

The cross product on algorithm 2 can result on any of the two orthogonal vectors to Tangent and Binormal. To discover its actual orientation we use the Object space normal vector associated with each vertex and check if they have the same orientation by performing a dot product. Using an incorrect orientation will result in wrong light calculation as explained on section 3.

Finalizing this process, the next step is to convert light vector used in Dot3 to Tangent space. The procedure below performs this operation.

Being Lo, light on Object space, Light_tangent a three dimensional vector:

```
Light_tangent = (Lo • T, Lo • B, Lo • N);
```

After these steps Tangent space calculation is done. It is important to note that Tangent space transformation is the most expensive part of Dot3 bump mapping implementation, as previously explained the generated overhead is counter balanced by the performance enhancement in some situations. This calculation can be done in the GPU (Graphics Processing Unit) using vertex programs, but this requires programmable pipeline compatible cards [5].

5. Dot3 bump mapping implementation

As previously mentioned Dot3 bump mapping can be applied to different types of computer graphics applications. Our implementation focused a 3D first person shooter game engine. It was based on fixed pipeline OpenGL programming for texture unit setup. Also we used OpenGL extensions widely available on 3D accelerator cards up to GeForce2 and compatible cards. This implementation of Dot3 bump mapping can be divided in 3 major steps, each one of them executing a specific task, that, combined, perform the two operations needed for method completion, computation of a perturbed normal and the dot3 product operation between the encoded normal and the light vector.

The first specific task is the transformation of light vector, whether under Object, Eye, Tangent or any vectorial space, into RGB values for later use as the primary color in texture combining stage of the rendering process. In this stage we will perform the dot3 operation between normal and light vector. The conversion of light vector (in our implementation, written under Tangent space) can be coded as follows:

Being Light_tangent the light vector composed by three coordinates (x, y, z). RGB are the Red, Green and Blue color primitive values:

```
{
  R= 0,5 + Light_tangent.x * 0,5;
  G= 0,5 + Light_tangent.y * 0,5;
  B= 0,5 + Light_tangent.z * 0,5;
}
```

Rescaling these values is necessary since OpenGL color values range from 0 to 1.

The second task deals with normal fetching. For this implementation, encoded normals can be fetched and applied as any texture map. One of the most common ways of encoding normal values is storing them in a texture-map, encoded as RGB values. This way we can easily fetch normals as they can be loaded as a texture. For instance, in standard OpenGL texture assignment can be coded as:

Considering that an image (in any desired format) has been read by a procedure and stored in a g_Texture variable

```
glBindTexture(GL_TEXTURE_2D,g_Texture);
glEnable(GL_TEXTURE_2D);
```

Other techniques, as some Emboss bump mapping implementations, require a special texture map reading. Dot3 bump mapping does not require that: a standard image reading procedure can be used.

In the third task we have to perform the Dot3 operation between light vector and the normals in a per-pixel basis. OpenGL provides the DOT3_RGB_EXT extension, which performs a Dot3 operation in every pixel of a textured surface to be rendered. DOT3_RGB_EXT is a constant sent to OpenGL pipeline to configure the operation that the GPU will perform. This configuration is set on texture environment setup. The dot3 operation will use as operands the primary color for each vertex, which encodes the light vector and a texture map that encode normals. Our implementation configures texture environment as:

```
{
  //This tells OpenGL to use texture combining
  //(pixel value multiplication)
  glTexEnvi(GL_TEXTURE_ENV,GL_TEXTURE_ENV_MODE,
            GL_COMBINE_EXT);

  //The combine operation to be performed
  //is a Dot3
   glTexEnvi(GL_TEXTURE_ENV,GL_COMBINE_RGB_EXT,
              GL_DOT3_RGB_EXT);

  //A texture containing encoded normals is
  //one of the sources to be used
  glTexEnvi(GL_TEXTURE_ENV,GL_SOURCE1_RGB_EXT,
            GL_TEXTURE);

   //The other is the light vector in Tangent
   //space, stored as primary color for
   //each vertex.
  glTexEnvi(GL_TEXTURE_ENV,GL_SOURCE0_RGB_EXT,
            GL_PRIMARY_COLOR_EXT);

   //Operates on RGB values.
  glTexEnvi(GL_TEXTURE_ENV,GL_OPERAND1_RGB_EXT,
            GL_SRC_COLOR);
  glTexEnvi(GL_TEXTURE_ENV,GL_OPERAND0_RGB_EXT,
            GL_SRC_COLOR);
}
```

After that Dot3 bump mapping process is completed and all operations needed for normal perturbation simulation will be executed.

Dot3 bump mapping combined with per pixel lightning also can replace some effects of the traditional lightmapping technique, used for years in game engines to illuminate surfaces, without actually implemented lights. One problem with this new approach is that now it is necessary to correctly orient light vectors and their specific components to achieve the desired illumination effect.

As said in Section 4, the operations performed for calculating Tangent space add an overhead to the traditional renderpath. But, fortunately, regarding the other steps of the implementation, hardware today is able to perform these operations on a single rendering pass.

6. Conclusion

In this paper we investigated Dot3 bump mapping as a method for simulating bumped surfaces, which, associated to per-pixel lighting calculation, leads to high quality scenes for game engines. Along the investigation we developed a set of systematic techniques to apply this method using OpenGL fixed pipeline and Tangent space transformation.

We could use the proposed techniques in simple, but comprehensive, examples. The code seams to be quite reusable and the results were very satisfactory. Nevertheless, during the implementation of the method applied to the game engine we faced several problems specially in debugging Tangent space transformation, due to the large number of vertices and complex vectorial operations performed in this algorithm. Other important issue was defining a correct light vector orientation, in order to achieve a correct illumination for scenes, which was a matter of design, but in our case a wrong configuration incurred in many days of debugging (the scene was dark due to an incorrect light source orientation).

As ongoing work we are implementing a Dot3 bump mapping in OpenGL programmable pipeline using the vertex and fragment programs since this approach allows much more control of the transformation and lighting pipeline functions.

Comparing this technique to other bump mapping techniques, Dot3 can result on much more realistic scene result and, implemented using OpenGL extensions, does not need special texture image reading procedures.

Computational overhead concerns on Dot3 bump mapping, especially the Tangent space transformation, can be easily attenuated by the increasing computational power of today's GPUs and their transformation and lighting pipeline. In this way the results regarding graphical enhancement of the Dot3 bump mapping is worthy for OpenGL compliant cards that support the necessary extensions.

Acknowledgements. The authors would like to acknowledge the partial support from CNPq under process number 552192/2002-3.

References

1. Bui Tuong Phong, "Illumination for Computer Generated Pictures", *Communications of the ACM*, 18(6), June 1975, pp. 311-317.
2. Eric Desrosiers, "Vulgarisation of Tangent Space calculation for triangle based mesh", Available at http://members.rogers.com/deseric/tangentspace.htm, July, 2003.
3. James Blinn, "Simulation of Wrinkled Surfaces,'' *Computer Graphics (Proc. Siggraph '78)*, August 1978, pp. 286-292. Also in *Tutorial: ComputerGraphics: Image Synthesis*, pp. 307-313.
4. Jeff Molofee, "OpenGL Windows: Emboss bump mapping tutorial", Available at http://www.gamedev.net, July, 2003.
5. Jim Dietrich, "Texture Space Bump Maps', NVIDIA Corporation.
6. M. Woo; J. Neider, T. Davis and D. Shreiner, "OpenGL Programming Guide", Third Edition, Addison-Wesley, 1999.
7. Mark J. Kilgard, "A practical and robust bump-mapping technique for today's gpus", In *GDC 2000: Advanced OpenGL Game Development*, July 2000.