Foundations *of* Game Engine Development

Foundations *of* Game Engine Development

VOLUME 2

RENDERING

by Eric Lengyel

Terathon Software LLC
Lincoln, California

Foundations of Game Engine Development
Volume 2: Rendering

ISBN-13: 978-0-9858117-5-4

Eighth printing

Published by Terathon Software LLC
www.terathon.com

Series website: foundationsofgameenginedev.com

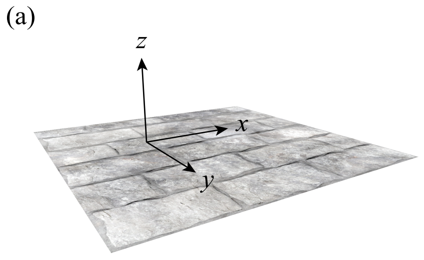
About the cover: The cover image is a scene from a game entitled *The 31st*, developed with the Tombstone Engine. Artwork by Javier Moya Pérez.

7.5 Tangent Space

Up to this point, our shading calculations have been carried out in object space. More advanced techniques, some of which have become standard fixtures in game engines, are able to make use of texture maps that store finely detailed geometric information instead of colors. The most common example is the normal mapping technique in which a texture map contains vector data, and this is introduced in the next section. The numerical values stored in this kind of texture map are expressed in the coordinate system of the texture map itself so that the geometric details are independent of any particular model. This allows a geometric texture map to be applied to any triangle mesh without having to account for the object-space coordinate system used by its vertices.

In the coordinate system of a texture map, the x and y axes are aligned to the horizontal and vertical directions in the 2D image, and the z axis points upward out of the image plane, as shown in Figure 7.11(a). If the origin of the texture map is located in the upper-left corner, then this constitutes a left-handed coordinate system. It is also possible to flip the texture upside down and put the origin in the lower-left corner to create a right-handed coordinate system. Either choice works fine because we will need to account for the handedness inherent in the mapping of the texture map to a surface anyway.

In order to perform shading calculations that use geometric information stored in a texture map, we need a way to transform between the coordinate system of the



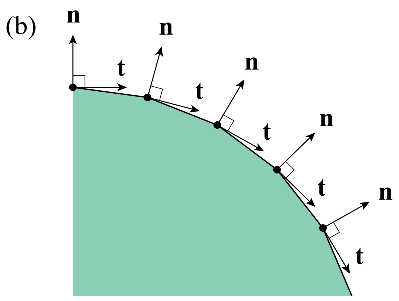


Figure 7.11. (a) In the coordinate system of a texture map, the x and y axes are aligned to the texel image, and the z axis points upward out of the image plane. (b) Each vertex in a triangle mesh has a normal vector **n** and a perpendicular tangent vector **t**, and both vectors form a smooth field over the entire model. The direction that the tangent vector points within the tangent plane is determined by the orientation of the texture map at each vertex.

texture map and object space. This is done by identifying the directions in object space that correspond to the coordinate axes of the texture map. These object-space directions are not constant, but vary from triangle to triangle. For a single triangle, we can think of the texture map as lying in the triangle's plane with its x and y axes oriented in the directions that are aligned to the (u, v) texture coordinates assigned to the triangle's vertices. The z axis of the texture map points directly out of the plane, so it is aligned with the triangle's normal vector in object space. The x and y axes of the texture map point along directions that are tangent to the surface in object space, and at least one of these vectors needs to be calculated ahead of time.

As with normal vectors, we calculate an average unit-length tangent vector $\mathbf{t}$ for each vertex in a triangle mesh. This lets us create a smooth tangent field on the surface of a model, as shown in Figure 7.11(b). Although it may not be strictly true for the specific texture mapping applied to a model, we assume that the two tangent directions are perpendicular to each other, so a second tangent direction $\mathbf{b}$ called the *bitangent vector* can be calculated with a cross product. The three vectors $\mathbf{t}$, $\mathbf{b}$, and $\mathbf{n}$ form the basis of the *tangent frame* at each vertex, and the coordinate space in which the x, y, and z axes are aligned to these directions is called *tangent space*. We can transform vectors from tangent space to object space using the 3×3 matrix $\mathbf{M}_{\text{tangent}}$ given by

$$\mathbf{M}_{\text{tangent}} = \begin{bmatrix} \uparrow & \uparrow & \uparrow \\ \mathbf{t} & \mathbf{b} & \mathbf{n} \\ \downarrow & \downarrow & \downarrow \end{bmatrix}, \tag{7.31}$$

which has the vectors $\mathbf{t}$, $\mathbf{b}$, and $\mathbf{n}$ as its columns. Since this matrix is orthogonal, the reverse transformation from object space to tangent space is the transpose

$$\mathbf{M}_{\text{tangent}}^{\mathrm{T}} = \begin{bmatrix} \leftarrow & \mathbf{t}^{\mathrm{T}} & \rightarrow \\ \leftarrow & \mathbf{b}^{\mathrm{T}} & \rightarrow \\ \leftarrow & \mathbf{n}^{\mathrm{T}} & \rightarrow \end{bmatrix}, \tag{7.32}$$

where the vectors $\mathbf{t}$, $\mathbf{b}$, and $\mathbf{n}$ form the rows. The name *TBN matrix* is often used to refer to either one of these matrices.

Applying a little linear algebra to the vertex positions and their associated texture coordinates lets us calculate the tangent field for a triangle mesh. Let $\boldsymbol{p}_0$, $\boldsymbol{p}_1$, and $\boldsymbol{p}_2$ be the three vertices of a triangle, wound in counterclockwise order, and let (u_i, v_i) represent the texture coordinates associated with the vertex $\boldsymbol{p}_i$. The values of u and v correspond to distances along the axes $\mathbf{t}$ and $\mathbf{b}$ that are aligned to the x

and y directions of the texture map. This means that we can express the difference between two points with known texture coordinates as

$$\boldsymbol{p}_i - \boldsymbol{p}_j = (u_i - u_j)\,\mathbf{t} + (v_i - v_j)\,\mathbf{b}. \tag{7.33}$$

To determine what the vectors **t** and **b** are, we can form a system of equations using differences between the vertices on two of the triangle's edges. After making the definitions

$$\begin{aligned} \mathbf{e}_1 &= \boldsymbol{p}_1 - \boldsymbol{p}_0, \quad (x_1, y_1) = (u_1 - u_0, v_1 - v_0), \\ \mathbf{e}_2 &= \boldsymbol{p}_2 - \boldsymbol{p}_0, \quad (x_2, y_2) = (u_2 - u_0, v_2 - v_0), \end{aligned} \tag{7.34}$$

we can write this system very compactly as

$$\begin{aligned} \mathbf{e}_1 &= x_1\mathbf{t} + y_1\mathbf{b} \\ \mathbf{e}_2 &= x_2\mathbf{t} + y_2\mathbf{b}. \end{aligned} \tag{7.35}$$

An equivalent matrix equation is

$$\begin{bmatrix} \uparrow & \uparrow \\ \mathbf{e}_1 & \mathbf{e}_2 \\ \downarrow & \downarrow \end{bmatrix} = \begin{bmatrix} \uparrow & \uparrow \\ \mathbf{t} & \mathbf{b} \\ \downarrow & \downarrow \end{bmatrix} \begin{bmatrix} x_1 & x_2 \\ y_1 & y_2 \end{bmatrix}, \tag{7.36}$$

where $\mathbf{e}_1$, $\mathbf{e}_2$, **t** and **b** are all column vectors. This equation is readily solved by inverting the 2×2 matrix of coefficients on the right side to produce

$$\boxed{\begin{bmatrix} \uparrow & \uparrow \\ \mathbf{t} & \mathbf{b} \\ \downarrow & \downarrow \end{bmatrix} = \frac{1}{x_1 y_2 - x_2 y_1} \begin{bmatrix} \uparrow & \uparrow \\ \mathbf{e}_1 & \mathbf{e}_2 \\ \downarrow & \downarrow \end{bmatrix} \begin{bmatrix} y_2 & -x_2 \\ -y_1 & x_1 \end{bmatrix}.} \tag{7.37}$$

To calculate an average tangent vector and bitangent vector at each vertex, we maintain sums of the vectors produced for each triangle and later normalize them. When values of **t** and **b** are calculated with Equation (7.37), they are added to the sums for the three vertices referenced by the triangle. The results are usually not exactly perpendicular, but unless the texture mapping is skewed to a significant degree, they should be *close* to perpendicular. We can nudge them the rest of the way by applying Gram-Schmidt orthonormalization. First, assuming the vertex normal vector **n** has unit length, we make sure the vertex tangent vector **t** is perpendicular to **n** by replacing it with

$$\mathbf{t}_\perp = \operatorname{nrm}(\mathbf{t} - (\mathbf{t} \cdot \mathbf{n})\,\mathbf{n}) \tag{7.38}$$

using the rejection operation described in Section 1.6. (We use the subscript ⊥ simply to mean that the vector has been orthonormalized.) The vertex bitangent vector **b** is then made perpendicular to both **t** and **n** by calculating

$$\mathbf{b}_\perp = \operatorname{nrm}(\mathbf{b} - (\mathbf{b} \cdot \mathbf{n})\,\mathbf{n} - (\mathbf{b} \cdot \mathbf{t}_\perp)\,\mathbf{t}_\perp). \tag{7.39}$$

The vectors $\mathbf{t}_\perp$, $\mathbf{b}_\perp$, and **n** now form a set of unit-length orthogonal axes for the tangent frame at a vertex. Code that implements this entire process is provided in Listing 7.4.

Since the vectors are orthogonal, it is not necessary to store all three of the vectors $\mathbf{t}_\perp$, $\mathbf{b}_\perp$, and **n** for each vertex. Just the normal vector and the tangent vector will always suffice, but we do need one additional bit of information. The tangent frame can form either a right-handed or left-handed coordinate system, and which one is given by the sign of $\det(\mathbf{M}_{\text{tangent}})$. Calling the sign of this determinant σ, we can reconstitute the bitangent with the cross product

$$\boxed{\mathbf{b}_\perp = \sigma(\mathbf{n} \times \mathbf{t}_\perp),} \tag{7.40}$$

and then only the normal and tangent need to be supplied as vertex attributes. An example showing the normal field and tangent field for a character model is provided in Figure 7.12. One possible way to communicate the value of σ to the vertex shader is by extending the tangent to a four-component vertex attribute and storing σ in the w coordinate. This is the method used in Listing 7.4, but a more clever approach might encode σ in the least significant bit of one of the x, y, or z coordinates of the tangent to avoid increasing the size of the vertex data.

It is common for there to be discontinuities in a model's texture mapping, and this is in fact unavoidable for anything topologically equivalent to a sphere because a continuous nonvanishing tangent field is impossible. In these cases, vertices are duplicated along the triangle edges where the discontinuity occurs. The additional vertices have the same positions, but they could have different texture coordinates. Because they are indexed separately, their tangent vectors are not averaged, and this can lead to a visible boundary where an abrupt change in shading is visible. To avoid this, duplicates need to be identified so that their tangents can be averaged and set equal to each other, but only if the tangent frames have the same handedness and the tangents are pointing in similar directions.

Listing 7.4. This function calculates the per-vertex tangent vectors for the triangle mesh having `triangleCount` triangles with indices specified by `triangleArray` and `vertexCount` vertices with positions specified by `vertexArray`. The per-vertex normal vectors and texture coordinates are given by `normalArray` and `texcoordArray`. Tangents are written to `tangentArray`, which must be large enough to hold `vertexCount` elements. The determinant of the matrix $\mathbf{M}_{\text{tangent}}$ at each vertex is stored in the w coordinate of each tangent vector.

```
void CalculateTangents(int32 triangleCount, const Triangle *triangleArray,
             int32 vertexCount, const Point3D *vertexArray, const Vector3D *normalArray,
             const Point2D *texcoordArray, Vector4D *tangentArray)
{
   // Allocate temporary storage for tangents and bitangents and initialize to zeros.
   Vector3D *tangent = new Vector3D[vertexCount * 2];
   Vector3D *bitangent = tangent + vertexCount;
   for (int32 i = 0; i < vertexCount; i++)
   {
      tangent[i].Set(0.0F, 0.0F, 0.0F);
      bitangent[i].Set(0.0F, 0.0F, 0.0F);
   }

   // Calculate tangent and bitangent for each triangle and add to all three vertices.
   for (int32 k = 0; k < triangleCount; k++)
   {
      int32 i0 = triangleArray[k].index[0];
      int32 i1 = triangleArray[k].index[1];
      int32 i2 = triangleArray[k].index[2];
      const Point3D& p0 = vertexArray[i0];
      const Point3D& p1 = vertexArray[i1];
      const Point3D& p2 = vertexArray[i2];
      const Point2D& w0 = texcoordArray[i0];
      const Point2D& w1 = texcoordArray[i1];
      const Point2D& w2 = texcoordArray[i2];

      Vector3D e1 = p1 - p0, e2 = p2 - p0;
      float x1 = w1.x - w0.x, x2 = w2.x - w0.x;
      float y1 = w1.y - w0.y, y2 = w2.y - w0.y;

      float r = 1.0F / (x1 * y2 - x2 * y1);
      Vector3D t = (e1 * y2 - e2 * y1) * r;
      Vector3D b = (e2 * x1 - e1 * x2) * r;

      tangent[i0] += t;
      tangent[i1] += t;
      tangent[i2] += t;
      bitangent[i0] += b;
      bitangent[i1] += b;
      bitangent[i2] += b;
   }

   // Orthonormalize each tangent and calculate the handedness.
```

```
    for (int32 i = 0; i < vertexCount; i++)
    {
        const Vector3D& t = tangent[i];
        const Vector3D& b = bitangent[i];
        const Vector3D& n = normalArray[i];
        tangentArray[i].xyz() = Normalize(Reject(t, n));
        tangentArray[i].w = (Dot(Cross(t, b), n) > 0.0F) ? 1.0F : -1.0F;
    }

    delete[] tangent;
}
```

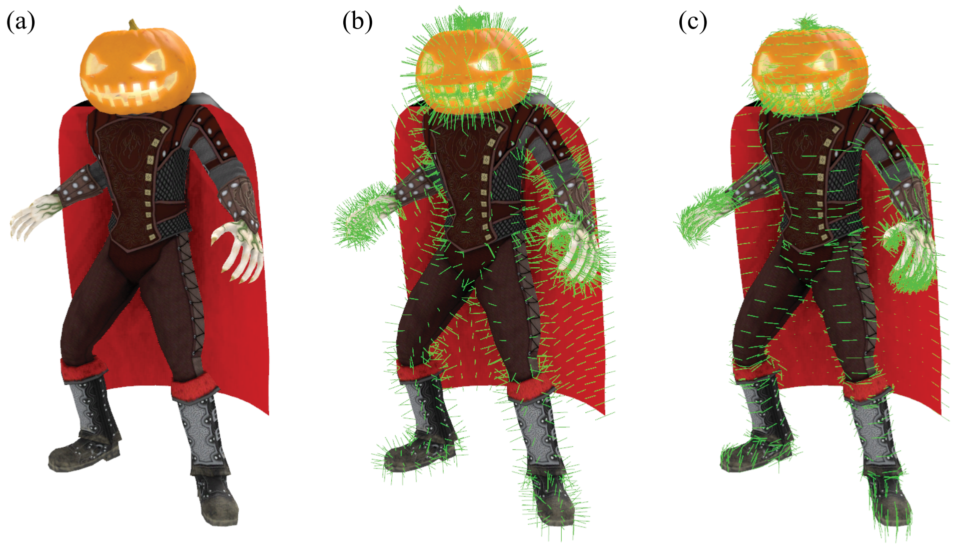


Figure 7.12. (a) A character model uses texture mapping techniques that require a tangent frame. (b) The normal vector corresponding to each vertex is shown as a green line starting at the vertex's position. (c) The tangent vector is shown for each vertex, and it is aligned to the x direction of the texture map at the vertex position.

7.6 Bump Mapping

Bump mapping is a technique that gives the surface of a model the appearance of having much greater geometric detail than is actually present in the triangle mesh. It works by using a special type of texture map call a *normal map* to assign a different normal vector to each point on a surface. When shading calculations account for these finely detailed normal vectors in addition to the smoothly interpolated geometric normal, it produces the illusion that the surface varies in height even though the triangles we render are still perfectly flat. This happens because the brightness of the reflected light changes at a high frequency, causing the surface to appear as if it were bumpy, and this is where the term bump mapping comes from. Because it uses a normal map, bump mapping is often called *normal mapping*, and the two terms can be used interchangeably. Figure 7.13 shows an example in which a wall is rendered as a flat surface having a constant normal vector and is rendered again with the varying normal vectors produced by bump mapping. The difference is already pretty remarkable, but we will be able to extend the concept even further with the addition of parallax and horizon mapping in the next two sections.

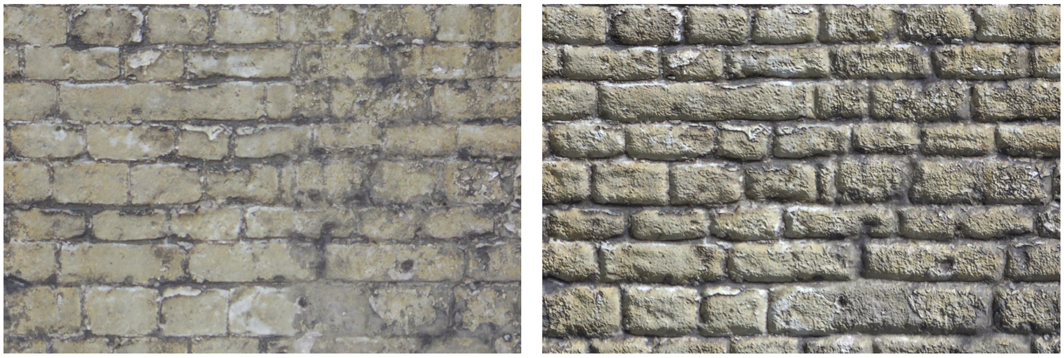


Figure 7.13. (a) A flat wall is rendered without bump mapping, so it has a constant normal vector across its surface. (b) The same wall is rendered with bump mapping, and the normal vector is modified by the vectors stored in a normal map. In both images, the direction to the light points toward the upper-right corner.

7.6.1 Normal Map Construction

Each texel in a normal map contains a unit-length normal vector whose coordinates are expressed in tangent space. The normal vector $(0,0,1)$ corresponds to a smooth surface because it is parallel to the direction pointing directly along the interpolated normal vector $\mathbf{n}$ in object space. Any other tangent-space normal vector represents a deviation from the smooth surface due to the presence of geometric detail. Though it is possible to generate a normal map based on some mathematical description of a surface, most normal maps are created by calculating slopes in a height field. A single-channel *height map* is typically supplied, and the values it contains correspond to the heights of the detailed geometry above a flat surface. For example, the height map that was used to create the normal map for the wall in Figure 7.13 is shown as a grayscale image in Figure 7.14(a).

To calculate the normal vector for a single texel in the normal map, we first apply central differencing to calculate slopes in the x and y directions. The values in the height map are usually interpreted as numbers in the range $[0,1]$, so they must be scaled by some constant factor to stretch them out into the full range of heights that are intended to be covered. Using a scale factor of s means that the maximum height is s times the width of a single texel. Let the function $h(i,j)$ represent the value, in the range $[0,1]$, stored in the height map at the coordinates (i,j). The two slopes d_x and d_y are then given by

$$\begin{aligned} d_x &= \frac{\Delta z}{\Delta x} = \frac{s}{2}[h(i+1,j) - h(i-1,j)] \\ d_y &= \frac{\Delta z}{\Delta y} = \frac{s}{2}[h(i,j+1) - h(i,j-1)]. \end{aligned} \tag{7.41}$$

At the edges of the height map, care must be taken to either clamp the coordinates or wrap them around to the opposite side of the image, depending on the intended wrapping mode of the resulting normal map.

Once the slopes have been determined, we can express directions $\mathbf{u}_x$ and $\mathbf{u}_y$ that are tangent to the height field along the x and y axes as

$$\mathbf{u}_x = (1,0,d_x) \quad \text{and} \quad \mathbf{u}_y = (0,1,d_y). \tag{7.42}$$

Since these are independent directions in the height map's tangent plane, the normal vector $\mathbf{m}$ can be calculated with the cross product

$$\boxed{\mathbf{m} = \operatorname{nrm}(\mathbf{u}_x \times \mathbf{u}_y) = \frac{(-d_x, -d_y, 1)}{\sqrt{d_x^2 + d_y^2 + 1}}.} \tag{7.43}$$

This is the value that gets stored in the normal map. We use the letter $\mathbf{m}$ to avoid confusion with the object-space normal vector $\mathbf{n}$ defined at each vertex. The code shown in Listing 7.5 uses Equation (7.43) to construct a normal map from a height map that has already been scaled by the factor of s appearing in Equation (7.41). At the edges of the height map, this code calculates differences by wrapping around to the opposite side.

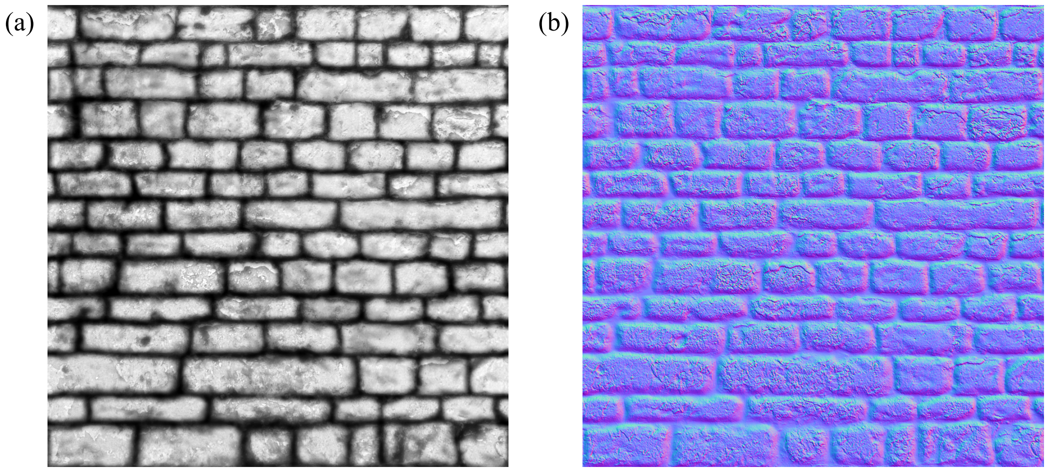


Figure 7.14. (a) A single-channel image contains a height map for a stone wall. Brighter values correspond to greater heights. (b) The corresponding normal map contains the vectors calculated by Equation (7.43) with a scale of $s = 24$. The components have been remapped to the range $[0,1]$.

In the early days of GPU bump mapping, the three components of the normal vector had to be stored in a texture map whose color channels could hold values in the range $[0,1]$. Since the x and y components of $\mathbf{m}$ can be any values in the range $[-1,+1]$, they had to be remapped to the range $[0,1]$ by calculating $r = \frac{1}{2}x + \frac{1}{2}$ for the red channel and $g = \frac{1}{2}y + \frac{1}{2}$ for the green channel. Even though the z component of $\mathbf{m}$ is always positive (because normal vectors always point out of the surface), the same mapping was applied to it as well for the blue channel. When a normal vector encoded in this way is fetched from a texture map, the GPU performs the reverse mapping back to the range $[-1,+1]$ by multiplying by two and subtracting one. Because the normal vectors are shortened a little by linear interpolation when they are fetched from a texture map, they should be renormalized in the pixel shader. The normal map shown in Figure 7.14(b) uses this encoding scheme. The fact that the blue channel always contains values in the range $[\frac{1}{2},1]$ gives it the characteristic purple tint that normal maps are known for having.

With the widespread availability of a much larger set of texture formats, we have better options. A texture map can have signed channels that store values in the range $[-1,+1]$, so we no longer need to remap to the range $[0,1]$. To save space, we can take advantage of two-channel formats by storing only the x and y components of a normal vector $\mathbf{m}$. After these are fetched from a texture map in the pixel shader, we can reconstitute the z component by calculating

$$m_z = \sqrt{1 - m_x^2 - m_y^2} \tag{7.44}$$

under the assumption that **m** has unit length and m_z is positive. This method for storing normal vectors works well with compressed texture formats that support two uncorrelated channels. Equation (7.44) is implemented in Listing 7.6.

Listing 7.5. This function constructs a normal map corresponding to the scaled height map having power-of-two dimensions `width` × `height` specified by `heightMap`. Normal vectors are written to the buffer supplied by `normalMap`, which must be large enough to hold `width` × `height` values.

```
void ConstructNormalMap(const float *heightMap, Vector3D *normalMap,
                        int32 width, int32 height)
{
   for (int32 y = 0; y < height; y++)
   {
      int32 ym1 = (y - 1) & (height - 1), yp1 = (y + 1) & (height - 1);

      const float *centerRow = heightMap + y * width;
      const float *upperRow = heightMap + ym1 * width;
      const float *lowerRow = heightMap + yp1 * width;

      for (int32 x = 0; x < width; x++)
      {
         int32 xm1 = (x - 1) & (width - 1), xp1 = (x + 1) & (width - 1);

         // Calculate slopes.
         float dx = (centerRow[xp1] - centerRow[xm1]) * 0.5F;
         float dy = (lowerRow[x] - upperRow[x]) * 0.5F;

         // Normalize and clamp.
         float nz = 1.0F / sqrt(dx * dx + dy * dy + 1.0F);
         float nx = fmin(fmax(-dx * nz, -1.0F), 1.0F);
         float ny = fmin(fmax(-dy * nz, -1.0F), 1.0F);
         normalMap[x].Set(nx, ny, nz);
      }

      normalMap += width;
   }
}
```

Listing 7.6. This pixel shader function fetches the x and y components of a normal vector from the 2D texture map specified by `normalMap` at the texture coordinates given by `texcoord`. The z component of the normal vector is reconstituted with Equation (7.44).

```
uniform Texture2D      normalMap;

float3 FetchNormalVector(float2 texcoord)
{
   float2 m = texture(normalMap, texcoord).xy;
   return (float3(m, sqrt(1.0 - m.x * m.x - m.y * m.y)));
}
```

7.6.2 Rendering with Normal Maps

To shade a surface that has a normal map applied to it, we perform the same calculations that we would perform without a normal map involving the view direction $\mathbf{v}$, light direction $\mathbf{l}$, and halfway vector $\mathbf{h}$. The difference is that we no longer take any dot products with the interpolated normal vector $\mathbf{n}$ because it is replaced by a normal vector $\mathbf{m}$ fetched from a normal map. Before we can take dot products with $\mathbf{m}$, though, we have to do something about the fact that it is not expressed in the same coordinate system as $\mathbf{v}$ and $\mathbf{l}$. We need to either transform $\mathbf{v}$ and $\mathbf{l}$ into tangent space to match $\mathbf{m}$ or transform $\mathbf{m}$ into object space to match $\mathbf{v}$ and $\mathbf{l}$.

To transform any vector $\mathbf{u}$ from object space to tangent space, we multiply it by the matrix $\mathbf{M}_{\text{tangent}}^{\text{T}}$ given by Equation (7.32). The rows of this matrix are the per-vertex tangent $\mathbf{t}$, bitangent $\mathbf{b}$, and normal $\mathbf{n}$, so the components of $\mathbf{u}$ simply become $\mathbf{t} \cdot \mathbf{u}$, $\mathbf{b} \cdot \mathbf{u}$, and $\mathbf{n} \cdot \mathbf{u}$ in tangent space. This is applied to the object-space view direction $\mathbf{v}$ and light direction $\mathbf{l}$ by the vertex shader function shown in Listing 7.7 after it calculates the bitangent vector with Equation (7.40). The resulting tangent-space view direction $\mathbf{v}_{\text{tangent}}$ and light direction $\mathbf{l}_{\text{tangent}}$ should then be output by the vertex shader so their interpolated values can be used in the pixel shader.

Shading can also be performed in object space by fetching the vector $\mathbf{m}$ from a normal map and then multiplying it by the matrix $\mathbf{M}_{\text{tangent}}$ in the pixel shader. The columns of $\mathbf{M}_{\text{tangent}}$ are the vectors $\mathbf{t}$, $\mathbf{b}$, and $\mathbf{n}$, so the transformed normal vector $\mathbf{m}_{\text{object}}$ is given by

$$\mathbf{m}_{\text{object}} = m_x\mathbf{t} + m_y\mathbf{b} + m_z\mathbf{n}. \tag{7.45}$$

To perform this operation in the pixel shader, we need to interpolate the per-vertex normal and tangent vectors in addition to the view direction $\mathbf{v}$ and the light direction $\mathbf{l}$, which now remain in object space. The handedness σ of the tangent frame

Listing 7.7. This vertex shader function calculates the tangent-space view direction $\mathbf{v}_{\text{tangent}}$ and light direction $\mathbf{l}_{\text{tangent}}$ for a vertex having object-space attributes `position`, `normal`, and `tangent` and returns them in `vtan` and `ltan`. The w coordinate of the tangent contains the handedness σ of the tangent frame used in the calculation of the bitangent vector.

```
uniform float3 cameraPosition;      // Object-space camera position.
uniform float3 lightPosition;       // Object-space light position.

void CalculateTangentSpaceVL(float3 position, float3 normal, float4 tangent,
                             out float3 vtan, out float3 ltan)
{
   float3 bitangent = cross(normal, tangent.xyz) * tangent.w;
   float3 v = cameraPosition - position;
   float3 l = lightPosition - position;
   vtan = float3(dot(tangent, v), dot(bitangent, v), dot(normal, v));
   ltan = float3(dot(tangent, l), dot(bitangent, l), dot(normal, l));
}
```

also needs to be passed from the vertex shader to the pixel shader. Since handedness is always constant over a triangle, it can be flat interpolated to save some computation. After interpolation, the normal and tangent vectors may not have unit length, and it's possible that they are no longer perpendicular. To correct for this, we have to orthonormalize them in the pixel shader before we calculate the bitangent vector with Equation (7.40). The pixel shader function shown in Listing 7.8 carries out these steps to construct the tangent frame. It then applies Equation (7.45) to transform a normal vector **m** fetched from a normal map into object space.

Listing 7.8. This pixel shader function fetches a normal vector from the 2D texture map specified by `normalMap` at the texture coordinates given by `texcoord` using the function in Listing 7.6. It then transforms it into object space by multiplying it by the matrix $\mathbf{M}_{\text{tangent}}$. The interpolated object-space normal, tangent, and handedness values are given by `normal`, `tangent`, and `sigma`.

```
uniform Texture2D      normalMap;

float3 FetchObjectNormalVector(float2 texcoord, float3 normal, float3 tangent,
                               float sigma)
{
   float3 m = FetchNormalVector(texcoord);
   float3 n = normalize(normal);
   float3 t = normalize(tangent - n * dot(tangent, n));
   float3 b = cross(normal, tangent) * sigma;
   return (t * m.x + b * m.y + n * m.z);
}
```

7.6.3 Blending Normal Maps

There are times when we might want to combine two or more normal maps in the same material. For example, a second normal map could add finer details to a base normal map in a high-quality version of the material. It's also possible that texture coordinate animation is causing two normal maps to move in different directions, as commonly used to produce interacting ripples on a water surface. We might also want to smoothly transition between two different normal maps. In general, we would like to have a function $f_{\text{blend}}(\mathbf{m}_1, \mathbf{m}_2, a, b)$ that calculates the weighted sum of two normal vectors $\mathbf{m}_1$ and $\mathbf{m}_2$ with the weights a and b. The sum must behave as if the original height maps from which $\mathbf{m}_1$ and $\mathbf{m}_2$ were derived had been added together with the same weights a and b, allowing a new normal vector to be calculated with Equation (7.43). We cannot simply calculate $a\mathbf{m}_1 + b\mathbf{m}_2$ because it does not satisfy this requirement. In particular, if one height map contains all zeros, then it should have no effect on the sum, but blending the normal vectors directly would cause the results to be skewed toward the vector $(0, 0, 1)$.

Fortunately, we can easily recover the slopes d_x and d_y to which a normal vector corresponds. All we have to do is scale a normal vector by the reciprocal of its z coordinate to match the unnormalized vector in the numerator of Equation (7.43), effectively undoing the previous normalization step. These slopes are nothing more than scaled differences of heights, so they are values that we *can* blend directly. This leads us to the blending function

$$\boxed{f_{\text{blend}}(\mathbf{m}_1, \mathbf{m}_2, a, b) = \operatorname{nrm}\left(a\frac{x_1}{z_1} + b\frac{x_2}{z_2}, a\frac{y_1}{z_1} + b\frac{y_2}{z_2}, 1\right),} \tag{7.46}$$

where $\mathbf{m}_1 = (x_1, y_1, z_1)$ and $\mathbf{m}_2 = (x_2, y_2, z_2)$.

By setting $a = 1 - t$ and $b = t$, we can smoothly transition from one normal map to another as the parameter t goes from zero to one. To combine two normal maps in such a way that they have an additive effect without diminishing the apparent size of the bumps encoded in either one, we apply Equation (7.46) with the weights $a = b = 1$. Since the result is normalized, we can multiply all three components by $z_1 z_2$, in which case the additive blending function is given by

$$f_{\text{add}}(\mathbf{m}_1, \mathbf{m}_2) = \operatorname{nrm}(z_2 x_1 + z_1 x_2, z_2 y_1 + z_1 y_2, z_2 z_1). \tag{7.47}$$

The weights a and b do not need to be positive. Using a negative weight for one normal map causes it to be subtracted from the other so that its bumps appear to be inverted.

7.7 Parallax Mapping

Plain bump mapping has its limitations. While it often looks good when a surface is viewed from a nearly perpendicular direction, the illusion of bumpiness quickly breaks down as the angle between the surface and the viewing direction gets smaller. This is due to the fact that the same texels are still rendered at the same locations on the surface from all viewing directions, betraying the flatness of the underlying geometry. If the surface features encoded in the normal map actually had real height, then some parts of the color texture would be hidden from view by the bumps, and other parts would be more exposed, depending on the perspective. A technique called *parallax mapping* shifts the texels around a little bit to greatly improve the illusion that a surface has varying height.

Parallax mapping works by first considering the height h and normal vector $\mathbf{n}$ mapped to each point on a surface. As shown in Figure 7.15, these values can be used to establish a plane $[\mathbf{n} \mid d]$ that is tangent to the bumpy surface at that point. This plane serves as a local approximation to the surface that can be used to calculate a texture coordinate offset that accounts for the viewing direction and produces the appearance of parallax. Since we need only the offset, we can assume that the original texture coordinates are $(0,0)$ for simplicity. The value of d is then determined by requiring that the point $(0,0,h)$ lies on the plane, from which we obtain

$$d = -n_z h. \tag{7.48}$$

For a particular tangent-space view direction $\mathbf{v}$, the point where the ray $\boldsymbol{o} + t\mathbf{v}$ intersects the plane $[\mathbf{n} \mid d]$ is approximately the point $\boldsymbol{p}$ that would be visible from the direction $\mathbf{v}$ if the surface actually had the height h at the point sampled on the flat geometry. The parameter t is calculated by solving the equation

$$[\mathbf{n} \mid d] \cdot (\boldsymbol{o} + t\mathbf{v}) = 0, \tag{7.49}$$

and the point $\boldsymbol{p}$ is thus given by

$$\boldsymbol{p} = \boldsymbol{o} + \frac{n_z h}{\mathbf{n} \cdot \mathbf{v}} \mathbf{v}. \tag{7.50}$$

The x and y coordinates of $\boldsymbol{p}$ provide the offset that should be added to the original texture coordinates. All texture maps used by the pixel shader, including the normal map, are then resampled at these new coordinates that include the parallax shift.

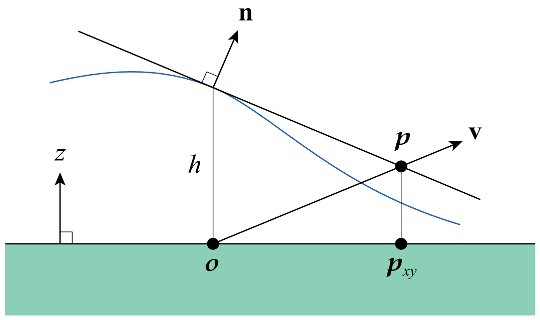


Figure 7.15. The height h and normal vector $\mathbf{n}$ sampled at the point $\boldsymbol{o}$ are used to construct a plane that approximates the bumpy surface shown by the blue line. For a tangent-space view vector $\mathbf{v}$, the parallax offset is given by the x and y coordinates of the point $\boldsymbol{p}$ where the ray $\boldsymbol{o} + t\mathbf{v}$ intersects the plane.

In practice, the offset given by Equation (7.50) is problematic because the dot product $\mathbf{n} \cdot \mathbf{v}$ can be close to zero, or it can even be negative. This means that the offset can produce an arbitrarily large parallax shift toward or away from the viewer when $\mathbf{n}$ and $\mathbf{v}$ are nearly perpendicular. The usual solution to this problem, even though it has little geometric significance, is to gradually reduce the offset as $\mathbf{n} \cdot \mathbf{v}$ becomes small by simply multiplying by $\mathbf{n} \cdot \mathbf{v}$! This effectively drops the division from Equation (7.50) and gives us the new offset formula

$$\boxed{\boldsymbol{p}_{xy} = n_z h \mathbf{v}_{xy}.} \tag{7.51}$$

This offset generally produces good results, like those shown in Figure 7.16, and it is very cheap to calculate. Because we are no longer dividing out their magnitudes, however, we must ensure that both $\mathbf{n}$ and $\mathbf{v}$ have unit length before applying this formula.

The value of $n_z h$ in Equation (7.51) is precomputed for every texel in the normal map and stored in a separate *parallax map* having a single channel. To minimize storage requirements, an 8-bit signed format can be used in which texel values fall in the range $[-1, +1]$. Unsigned values h from the original height map are remapped to this range by calculating $2h - 1$ before multiplying by n_z and storing the results in the parallax map. A signed format is chosen so that texture coordinates are shifted both toward and away from the viewer when the full range of heights is well utilized. An original height of $h = 1/2$ corresponds to no offset, larger

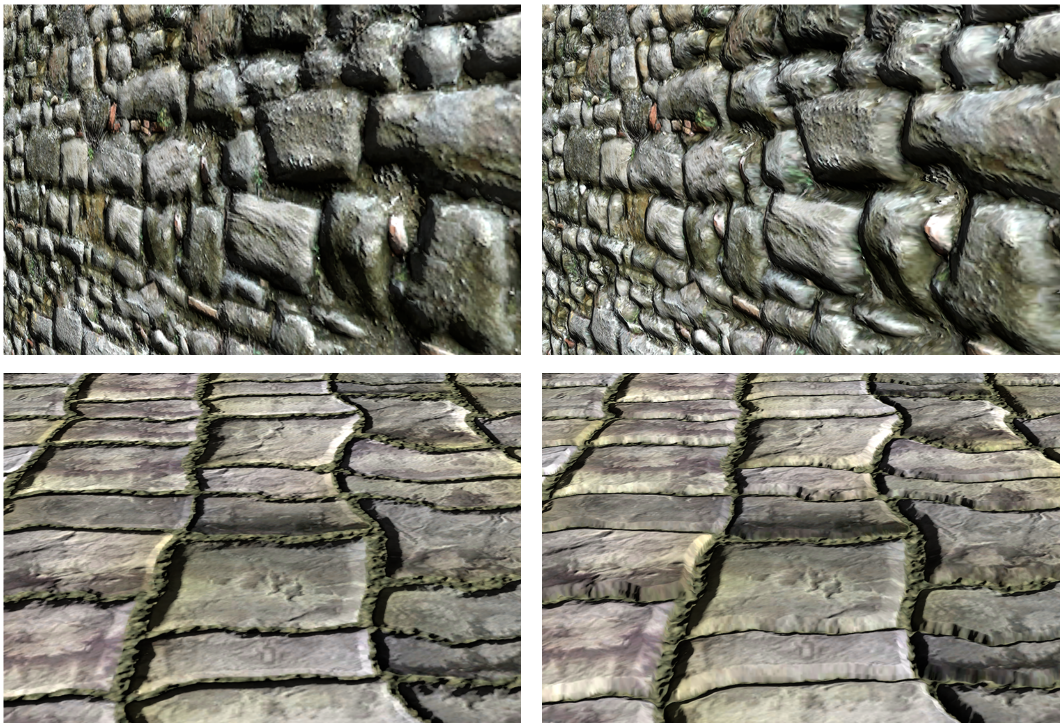


Figure 7.16. Two flat surfaces are rendered with only normal mapping in the left column. A parallax shift has been applied to the same surfaces in the right column. Texture coordinate offsets are calculated using Equation (7.51) with four iterations.

heights cause a surface to appear raised, and smaller values cause a surface to appear depressed. In the pixel shader, the sampled values of $n_z h$ are multiplied by $1/2$ to account for the doubling when they were converted to the signed format. They must also be multiplied by the same scale used when the normal map was generated so that the heights used in parallax mapping are the same as those that were used to calculate the per-texel normal vectors.

For bump maps containing steep changes in height, offsets given by Equation (7.51) can still be too large because the tangent plane becomes a poorer approximation as the size of the offset increases. Shifted color samples taken too far away from areas having a steep slope often produce visible artifacts. This problem can be eliminated in most cases by using k iterations and multiplying the offset by $1/k$ each time. A new value of $n_z h$ is fetched from the parallax map for each iteration, allowing each incremental parallax shift to be based on a different approximating plane.

The pixel shader code shown in Listing 7.9 implements parallax mapping with Equation (7.51), and it uses four iterations to mitigate the appearance of artifacts produced by steep slopes. The two-component scale value **u** passed to the function is given by

$$\mathbf{u} = \left(\frac{s}{2kr_x}, \frac{s}{2kr_y} \right), \tag{7.52}$$

and it accounts for several things that can be incorporated into a precalculated product. First, it includes the height scale s that was originally used to construct the normal map. Second, since the heights are measured in units of texels, the parallax offsets must be normalized to the actual dimensions (r_x, r_y) of the height map, so the scale is multiplied by the reciprocals of those dimensions. (This is the only reason why there are two components.) Third, the scale includes a factor of $1/2$ to account for the multiplication by two when the heights were converted from unsigned to signed values. Fourth, the scale includes a factor of $1/k$, where k is the number of iterations, so that each iteration contributes its proper share of the final result. Finally, the scale may include an extra factor not shown in Equation (7.52) that exaggerates the parallax effect.

Listing 7.9. This pixel shader function applies parallax mapping to the texture coordinates given by `texcoord` using four iterations and returns the final result. The texture specified by `parallaxMap` holds the signed values $n_z h$ belonging to the parallax map. The `vdir` parameter contains the tangent-space view direction, which must be normalized to unit length. The value of `scale` is given by Equation (7.52), where $k = 4$ in this code.

```
uniform Texture2D      parallaxMap;

float2 ApplyParallaxOffset(float2 texcoord, float3 vdir, float2 scale)
{
    float2 pdir = vdir.xy * scale;
    for (int i = 0; i < 4; i++)
    {
        // Fetch n.z * h from the parallax map.
        float parallax = texture(parallaxMap, texcoord).x;
        texcoord += pdir * parallax;
    }

    return (texcoord);
}
```