5

Smooth Horizon Mapping

Eric Lengyel
Terathon Software

5.1 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 5.1. A conventional diffuse color map and normal map are used to render a cube having flat sides in Figure 5.1(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 5.1(f).

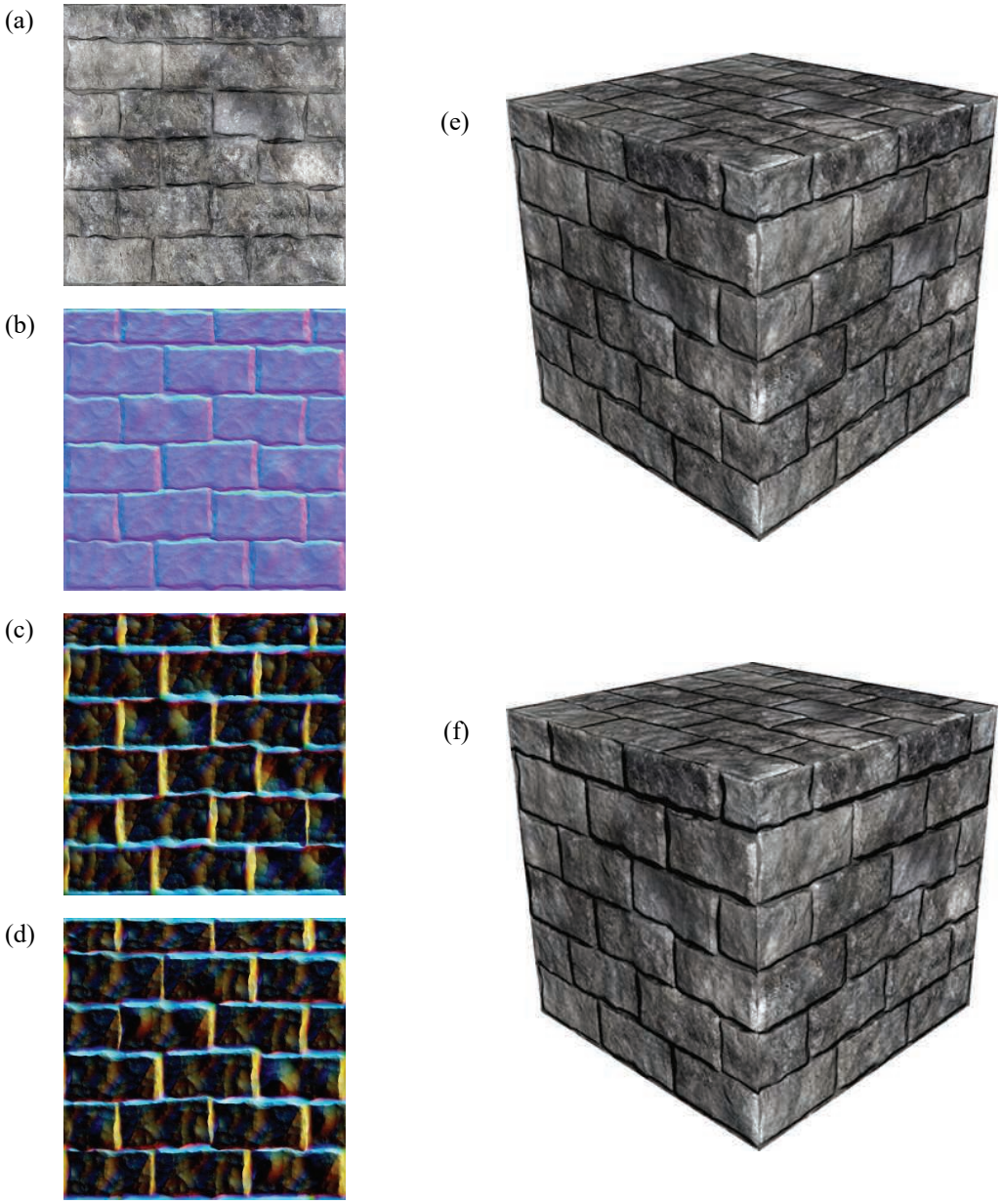

Figure 5.1. The conventional diffuse color map (a) and normal map (b) are augmented by eight channels of horizon mapping information in (c) and (d). A cube rendered only with ordinary normal mapping is shown in (e). Horizon mapping has been applied in (f).

A variety of materials rendered with and without shadows generated by horizon maps are shown in Figure 5.2. 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 5.3, the same material is rendered nine times as the direction to the light source rotates about the surface normal by 15-degree increments.

5.2 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 5.1, 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 5.4, 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.

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 source code accompanying this chapter on the book's website. For each texel in the output

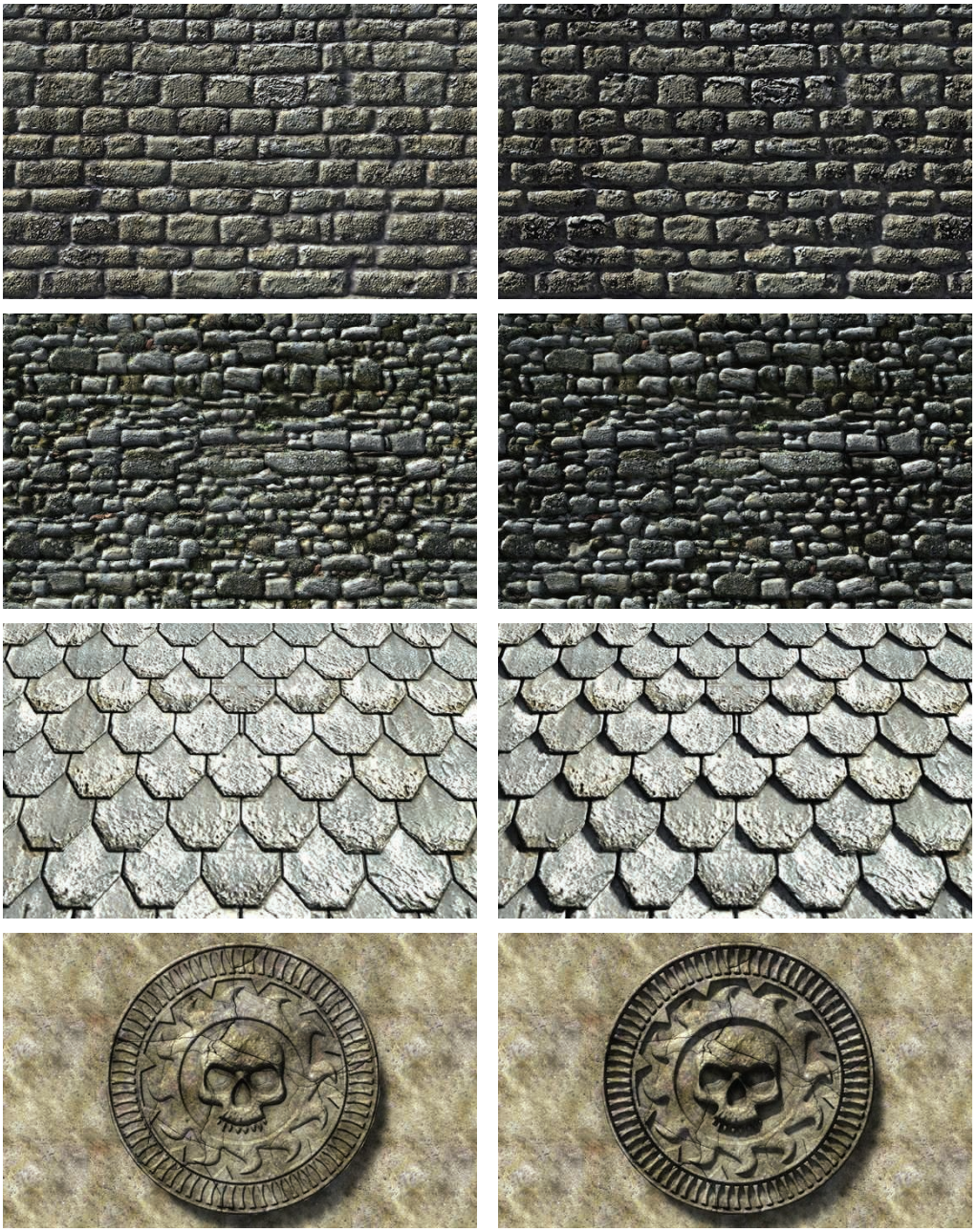

Figure 5.2. A variety of horizon mapping examples. In the left column, only ordinary normal mapping is applied. In the right column, shadows are added by horizon mapping.

Figure 5.3. This flat disk is illuminated from nine different angles ranging from –60° to +60° in 15-degree increments. Shadows are dynamically rendered using information stored in the eight channels of the horizon map.

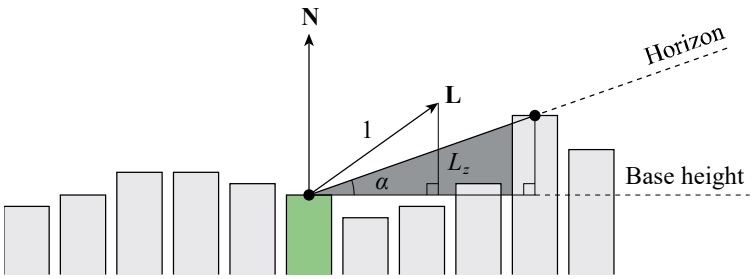

Figure 5.4. The horizon information for the central texel (green) in the direction to the right is given by the maximum angle α determined by the heights of other texels in a local neighborhood. The value stored in the horizon map is $\sin\alpha$, and this is compared against L_z when rendering to determine whether the pixel being shaded is illuminated. If $L_z > \sin\alpha$, then light reaches the pixel; otherwise, the pixel is in shadow.

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 α made with the central texel as

$$\tan^2\alpha = \frac{(h-h_0)^2}{(x-x_0)^2+(y-y_0)^2}, \tag{5.1}$$

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

$$\sin\alpha = \frac{1}{\sqrt{\frac{1}{\tan^2\alpha}+1}}. \tag{5.2}$$

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°, 45°, 90°, and 135° 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°, 225°, 270°, and 315° 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.

5.3 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

$$s = \max(2\mathbf{f} - 1, 0) \cdot \mathbf{h}_0 + \max(-2\mathbf{f} + 1, 0) \cdot \mathbf{h}_1, \tag{5.3}$$

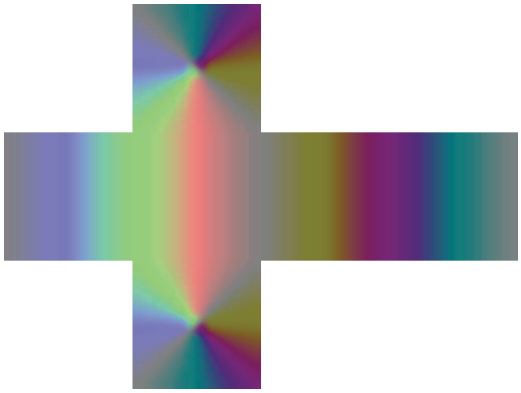

Figure 5.5. When sampled with coordinates given by the tangent-space direction to light vector, this cube texture map returns the channel factors to be applied to the information stored in the horizon map.

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 5.5. This texture is included in the materials accompanying this chapter on the book's website along with code for generating it. We have found that a cube texture having a resolution of only 16×16 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 (5.3) 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 5.6(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 5.6(b). This can be accomplished by calculating

$$F = k(L_z - s) + 1 \tag{5.4}$$

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

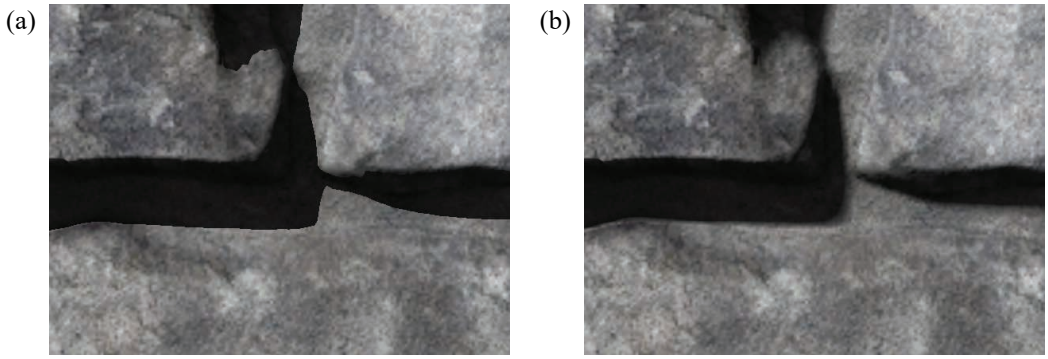

Figure 5.6. This close-up comparison shows the difference between a hard shadow and a soft shadow for a stone wall. (a) The illumination factor F is exactly one or zero, depending on whether $L_z \geq s$. (b) The illumination factor F is given by Equation (5.4) with $k = 8.0$.

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 5.1 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 (5.3) and multiplies the RGB components of `color` by the illumination factor F given by Equation (5.4). (Instead of using the max function that appears in Equation (5.3), 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.

Listing 5.1. 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 (5.4).

```
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.