Implementing Bump-Mapping using Register Combiners

Chris Wynn
NVIDIA Corporation
cwynn@nvidia.com
Overview

• Motivation
  • Goals
  • Required GPU Features
• Overview of Bump-Mapping Technique
  • Lighting Equation
  • Rendering Strategy
• Normal Maps
  • Construction from Height Map
  • Relationship to Texture Space
• Per-Pixel Lighting Setup
  • Light and Half-angle Vector Calculations
  • Normalization Cube-Maps
  • Surface Local Space
• Register Combiner Configurations
  • Diffuse, Diffuse + Specular, Self-Shadowing
Why Bump-Mapping?

- Offers Accurate Lighting at the Pixel Level
- Provides Increased Realism
  - Surface detail
  - Surface irregularities
- Simulates Complex Geometry
  - Reduces geometric complexity required to capture a certain level of detail
    - Memory and Performance benefits
    - Form of lossy geometry compression
- Looks Great!
Why Bump-Mapping?

- Simulate surface detail on an object by computing accurate lighting on a per-pixel basis.

- Vertex lighting would require significantly more polygons to capture same amount of detail.
Why Bump-Mapping?

- Particularly compelling in dynamic scenes
  - Moving objects with respect to light(s)
  - Animated characters

- Demo…
What GPU features required?

- Register Combiners – for per-pixel dot-products and additional lighting equation math
  - `NV_register_combiners`
    
    *`ARB_texture_env_dot3` can be used instead, but with substantially less flexibility in the lighting equation*

- Dual or Quad Texture – for normal map, diffuse map, gloss map, normalization cube-map, etc.
  - `ARB_multitexture`

- Cube Maps – (optional) for per-pixel normalization
  - `ARB_texture_cube_map`

- Vertex Program – (optional) for offloading per-vertex setup code
  - `NV_vertex_program`
Overview of Bump-Mapping Technique

• Basic Idea:
  • Start with a model.
  • Provide per-pixel normals.
  • Provide other required per-pixel lighting parameters (light vector, half-angle vector, etc.).
  • For each pixel:
    Evaluate “some” lighting model using per-pixel normal and other lighting parameters.

Typically use a variant of Blinn’s version of the Phong lighting model:

\[
\text{out}_{\text{col}} = \text{diffuse}_{\text{col}} \times (\mathbf{N} \cdot \mathbf{L}) + \text{spec}_{\text{col}} \times (\mathbf{N} \cdot \mathbf{H})^m
\]
Overview of Bump-Mapping Technique

- How this is done in real-time:
  - Encode normals into a texture.
  - Map the “normal map” texture onto a model using standard 2D texture mapping.
  - Compute L and/or H vectors on a per-vertex basis and interpolate these across a triangle.
  - Compute the necessary dot-products using texture combining hardware (ex. register combiners)

Requires custom vertex processing AND some pixel processing – ideal for GeForce-class GPUs.
Overview of Bump-Mapping Technique

- Other lighting models possible but we’ll focus on Blinn-Phong for simplicity:
  \[ \text{out}_\text{col} = \text{diffuse}_\text{col} \times (N' \cdot L) + \text{spec}_\text{col} \times (N' \cdot H)^m \]

- Ignore (for now)
  - ambient, spotlighting, shadowing, distance atten.

“Key” in implementing bump-mapping is understanding:

1. How to provide the GPU with the per-pixel parameters (vectors N’, L, and H)
2. How to compute the per-pixel dot-products
Providing Per-Pixel Parameters: The Normal Map

- Per-pixel normal vectors specified using a “Normal Map”
- Normal Map
  - 2D texture map that encodes (x,y,z) unit-length normal vectors.
  - **GL_RGB:**
    - Signed \((R,G,B) = (x,y,z)\)
    - Unsigned \((R,G,B) = .5 \times (x,y,z) + (.5, .5, .5)\)
  - **GL_HILO_NV:**
    - Signed \((HI,LO) = (x,y)\)
    - Unsigned N/A
Providing Per-Pixel Parameters: The Normal Map

- Normal Map constructed from a Height Map
  - Convert height-field to normal map using finite differencing (\( \text{d}i/\text{d}s, \text{d}i/\text{d}t, \text{scale} \))

The mostly chalk blue appearance is because the “straight up” normal is \([0.5 0.5 1.0]\)
Providing Per-Pixel Parameters: The Normal Map

- Texture the model with the Normal Map

... Per-pixel normals

In order to compute meaningful per-pixel dot-products, the L, H, and N' vectors must be defined in the same coordinate space:

- World space
- Eye space
- Any other space

Must understand what space the per-pixel normals are in...
Understanding the Normal Map

- Recall how we constructed the normal map
- Finite differencing

\[
( \frac{d}{ds}, \frac{d}{dt}, \text{scale} ) = (1, 0, -\frac{d}{ds}) \times (0, 1, -\frac{d}{dt})
\]

\[
\frac{d}{ds} = \text{change in height when moving along s axis}
\]

\[
\frac{d}{dt} = \text{change in height when moving along t axis}
\]

In a local region of constant height

\[
(0, 0, \text{scale} ) = (1, 0, 0) \times (0, 1, 0)
\]

and the normal points “straight up (or down)”
Understanding the Normal Map

- Finite differencing
  \[( \frac{di}{ds}, \frac{di}{dt}, \text{scale} ) = (1, 0, -\frac{di}{ds}) \times (0, 1, -\frac{di}{dt}) \]

- When mapped onto a model in 3-space, \(\frac{di}{ds}\) and \(\frac{di}{dt}\) correspond to the change in height when moving along \(S\) and \(T\) direction vectors defined in 3-space.
  - \(S\) and \(T\) indicate the direction in which the texture is mapped or “wrapped” onto the model.
  - \(S\), \(T\), and \(S\times T\) form a basis called “Texture Space” – this is the space per-pixel normals are defined in
Understanding the Normal Map

- So...

  Normals in the normal map are defined in Texture Space (S,T, SxT)
  AND
  Texture Space is defined by how a 2D texture is mapped onto a 3D model (more on this later)

- For correct per-pixel lighting we must either:
  - Compute dot-products in Texture Space
  - Compute dot products in some other space
    - Would require transforming each N’ to the correct space before using it
Providing Per-Pixel Parameters: The Light and Halfangle Vectors

- In order to compute lighting, we need to specify per-pixel L and H vectors.
- Since N’ is already in texture space, it’s convenient to provide vectors in the same space.

Overview

- Compute vectors at each vertex of the model
- Interpolate (and renormalize) across a poly
  - Unit-length vectors per-pixel

- Specifying L and H vectors is pretty much the same
  - For simplicity, we’ll just consider L...
Providing Per-Pixel Parameters: The Light and Halfangle Vectors

- For each vertex...
  - compute unit-length L vector
  - transform into Texture Space

- Specify the Texture Space L vector as a vertex parameter and allow it to be interpolated

Two ways:

1. Specify L as an RGB color
   - Colors clamped to [0,1] so must “range compress” the L vector
     (i.e. `glColor3f( .5(Lx+1), .5(Ly+1), .5(Lz+1) )` )
   - Each R, G, B component interpolated independently
     - Renormalize using the register combiners

2. Specify L as (s,t,r) texture coordinates and use a “Normalization” Cube-Map to produce unit-length vectors
Providing Per-Pixel Parameters: The Normalization Cube-Map

- Cube-Maps not only useful for Environment Mapping
- Useful for looking up ANY function of direction.

Think: Cube-Map = F(V) where V is a direction vector

- Normalization Cube-Map encodes the function:
  $F(V) = \text{normalize}(V)$

- Each texel of cube-map stores RGB representing range-compressed normalized vector from origin to the texel

- Magnitude does not alter cube-map texture fetch
  - Valid way to get normalized version of (s,t,r)
  - 32x32x6 often sufficient (GL_NEAREST filtering)
Providing Per-Pixel Parameters: The Normalization Cube-Map

- Normalization Cube-Map (unsigned RGB)

- What happens if you don’t re-normalize? (highlights lost across poly!)
Why Interpolation Works…

- Bump-Mapping based on an assumption:
  distance from actual surface to light $>>$
  distance from simulated surface to actual surface

- This is a reasonable assumption for small scale detail.
Computing Dot-Products using the Combiners

Diffuse Lighting

tex0: normal map (N’)
tex1: normalization cube-map (L)

!!RC1.0
{
  rgb {
    spare0 = expand(tex0) . expand(tex1); // NdotL
  }
}
out.rgb = spare0; // auto clamped to [0,1]

“expand” mapping assumes tex0 and tex1 are unsigned RGB
- not required for signed RGB formats
Computing Dot-Products using the Combiners

**Diffuse w/ Decal Modulation**

- tex0: normal map (N’)
- tex1: normalization cube-map (L)
- tex2: decal texture

```plaintext
!!RC1.0
{
   rgb {
      rgb {
         spare0 = expand(tex0) . expand(tex1);  // NdotL
      }
   }
}
out.rgb = spare0 * tex2;
```

Single pass on GeForce3 (Two-pass on GeForce)
Computing Dot-Products using the Combiners

Add Ambient w/ const. color

- tex0: normal map (N’)
- tex1: normalization cube-map (L)
- tex2: decal texture

```cpp
const0 = (0.2, 0.2, 0.2, 0); // Ambient
{
    rgb {
        spare0 = expand(tex0) . expand(tex1); // NdotL
    }
}
out.rgb = spare0 * tex2 + const0;
```
Computing Dot-Products using the Combiners

Specular Lighting (N’ • H)

tex0: normal map (N’)
tex1: normalization cube-map (H)

```glsl
!!RC1.0
{
    rgb {
        spare0 = expand(tex0) . expand(tex1);  // NdotH
    }
}
out.rgb = spare0;
```

\( (N\cdot H)^m \) where \( m = 1 \)
What about higher powers of \( m \)?
Computing Dot-Products using the Combiners

Specular Lighting \((N' \cdot H)^4\)

```c
!!RC1.0
{
  rgb {
    spare0 = expand(tex0) . expand(tex1); // NdotH
  }
}
{
  rgb {
    spare0 = unsigned(spare0) * unsigned(spare0);  
  }
}
final_product = spare0 * spare0;
out.rgb = final_product;
```

Clamp to \([0,1]\) before squaring
Computing Dot-Products using the Combiners

Diffuse + Specular

decal_{col} * (N'\cdot L) + spec_{col} * (N'\cdot H)^4

tex0: normal map (N')
tex1: normalization cube-map (L)
tex2: normalization cube-map (H)
tex3: decal texture
Computing Dot-Products using the Combiners

**Diffuse + Specular:** $\text{decal}_{\text{col}} \cdot (N' \cdot \text{L}) + \text{spec}_{\text{col}} \cdot (N' \cdot \text{H})^4$

```c
!!RC1.0
const0 = ( 0.2, 0.2, 0.2, 0 ); // Spec. color
{
    rgb {
        spare0 = expand(tex0) . expand(tex1); // NdotLspare1 = expand(tex0) . expand(tex2); // NdotH
    }
}
{
    rgb {
        spare0 = tex3 * unsigned(spare0); // decal*NdotL
        spare1 = unsigned(spare1) * spare1; // NdotH^2
    }
}
final_product = spare1 * spare1; // NdotH^4
out.rgb = const0 * final_product + spare0;
```
Surface Self-Shadowing

- Previous examples do not self-shadow correctly
- Two kinds of self-shadowing
  - $\max(0, L \cdot N')$ based on the perturbed normal
  - Also should clamp when $L \cdot N$ goes negative!
Self-Shadowing Computation

- Modulate specular and diffuse by: \((L_z < 0) \ ? \ 0 : 1\)
  - Use `mux()` in the combiners
  - Simple, but may result in "hard" shadow boundary
    - Possible winking and popping of highlights.
  - Requires alpha portions of two combiners

```c
{ alpha {
    spare0 = tex1.b; // spare0.a = Lz (unexpanded);
} }
{ alpha {
    discard = zero;
    discard = one;
    spare0 = mux(); // spare0.a = (Lz < 0) ? 0 : 1
} }
out.rgb = ... * spare0.a;
```
Self-Shadowing Computation

- Better, modulate by \( \min(8\times\max(L_z,0), 1) \)
  - Steep ramp eliminates popping
  - Requires alpha portion of only one combiner

```c
{ alpha {
    discard = expand(tex1.b);
    discard = expand(tex1.b);
    spare0 = sum();
    scale_by_four(); // spare0.a = 8 * Lz
}
``` out.rgb = \( \ldots \times \text{unsigned(spare0.a)} \);

- In either case, illumination does not appear on geometric “back side”.

Normalization in the Combiners

- Previous examples used Normalization Cube-Map
  - Not necessary on GeForce3

- By using an approximation technique, can normalize in the combiners

- It can be shown (by numerical means) that...

\[
\text{Normalize}( V ) \approx \frac{V}{2} * (3 - V \cdot V) \quad \text{when}
\]

1. V is a vector derived from the interpolation of unit-length vectors across a polygon AND
2. The angle between all pairs of the original per-vertex vectors is no more than 40º (or so).

For models of reasonable tessellation (and/or reasonable distance to the light and viewer) #2 holds.
Normalization in the Combiners

- Simplifying the approx.
  \[ \frac{V}{2} \ast (3 - V \cdot V) = 1.5V - 0.5V \ast (V \cdot V) \]
  \[ = V + 0.5V - 0.5V \ast (V \cdot V) \]
  \[ = V + 0.5V \ast (1 - (V \cdot V)) \]

- Compute simplified approx. in 2 general combiner stages…
Normalization in the Combiners

Suppose col0 contains interpolated (de-normalized) vector compressed into [0..1] range

```c
{ // normalize V (step 1.)
    rgb {
        spare0 = expand(col0) . expand(col0); // VdotV
    }
}

{ // normalize V (step 2.)
    rgb {
        discard = expand(col0); // V in [-1..1]
        discard = half_bias(col0) * unsigned_invert(spare0);
        col0 = sum();
    }
}
```

\[
\text{col0} = V + 0.5V(1 - V \cdot V)
\]

\[
V \text{ in } [-0.5..0.5] \equiv 0.5V \quad 1 - V \cdot V
\]
Normalization in the Combiners

- Normalization of one vector requires 2 general combiners, but two vectors can be normalized in 3.
- Combiner normalization faster than using cube-maps!
For More Information…

- **NVIDIA OpenGL SDK**
  - Technical Demos
  - Bump-Mapping Lab Exercises
    - Vertex Programming of Setup Code
    - Register combiner configuration

- **Additional Bump-Mapping Presentations**
  - Per-Pixel Lighting Mathematical background
  - Texture Space
  - Register Combiners
  - Bump-Mapping of Animated Models

- **Available at NVIDIA Developer Website:**
Acknowledgements

- **Scott Cutler**
  - Newton-Raphson fast “combiner normalization” technique.

- **Cass Everitt**
  - Earth demo.

- **Mark Kilgard**
  - Slide content.
  - “A Practical and Robust Bump-mapping Technique for Today’s GPUs” white paper.
Questions, comments, feedback

- Chris Wynn, cwynn@nvidia.com
- www.nvidia.com/developer