Goal Of This Talk

- The new features of Dx8 and the next generation of HW make huge strides in the area of Per-Pixel Shading

- Most developers have yet to adopt Per-Pixel Shading techniques, and rely on simple multi-texturing

- The goal of this talk is to introduce you to Per-Pixel Shading concepts necessary to understand and implement these techniques in your app
What is Per-Pixel Shading?

- Dynamic Shading computed on a per-pixel basis

- Per Pixel shading includes a variety of dynamic shading techniques
  - Bump Mapping
  - Per-Pixel Directional Lights
  - Per-Pixel Spot Lights
  - Per-Pixel Point Lights
A Single Quad Lit Per-Pixel
What this Talk will Cover

• Bump Mapping Overview
  • EMBM, Embossing, DOT3
  • Bump Mapping as Per-Pixel Shading

• Texture Space
  • A Vertex-Local Coordinate System
  • Animation And Per-Pixel Shading

• Per-Pixel
  • Directional Lights
  • Point Lights
    • Distance Attenuation
  • Spot Lights
What This Talk Will Cover

- DX8 Pixel Shaders
  - How they improve upon Dx7-Style Per-Pixel Shading
  - Integrating Pixel Shaders
Bump Mapping Overview

- Bump Mapping is a subset of Per-Pixel Lighting

- Bump Mapping examples such as Embossing simulate diffuse directional lighting

- Dx6 Environmental Bump Mapping (EMBM) simulates planar specular reflections

- However, DOT3 Per Pixel Lighting can be used to achieve diffuse and/or specular point lights, spotlights and volumetric lights as well
This talk will primarily cover D3DTOP_DOTPRODUCT3, or DOT3 effects.

DOT3 is more generally useful for per-pixel lighting than either Embossing or EMBM.

DOT3 is a texture blending operation that performs a dot product between two vectors:
  - It yields a value from 0 to 1.
Performing a Per-Pixel Dot Product

- Directional Diffuse Lighting is computed by
  \[ \text{LightColor} \times ( \text{LightVector} \cdot \text{SurfaceNormal} ) \]

- Directional Specular Lighting is computed via
  \[ \text{LightColor} \times ( \text{HalfAngle} \cdot \text{SurfaceNormal} ) \]

- The DOT3 per-pixel operation allows us to perform \( L \cdot N \) or \( H \cdot N \) on a per-pixel basis
Per-Pixel Dot Products

• To perform a per-pixel dot product, we require three things

• A Light Vector L

• A Surface Normal N

• A Common Coordinate System for L and N
The Light Vector L

- We can store the Light Vector L on a per-vertex basis in an iterated color

- We can use either the Diffuse or Specular iterated color

- This allows the L vector to be interpolated across a triangle, just like a color
  - perspective correction is important for this
The Surface Normal N

- We also want a per-pixel surface normal $N$
- This is stored in a “Normal Map” which is typically derived from a Height Map
Normal Maps

- Normal Maps are just an array of normalized surface vectors encoded as RGB colors.

- They are defined such that:
  - X maps to Red
  - Y maps to Green
  - Z maps to Blue
  - Alpha is available for gloss or other scalar values
Single Quad Lit With Per-Pixel Directional and Point Lights
What About a Common Coordinate System?

- The Surface Normals are expressed in their own coordinate system, such that
  - +X points to the Right
  - +Y points Up
  - +Z points Out of the screen

- The Light Vector is expressed in World Space

- Either N must be transformed into World Space with L, or we must transform L into this Normal Map coordinate system
We can generate Normal Maps in World Space, but then our world needs to be more or less uniquely textured, wasting texture memory.

This also only works for static geometry – what would happen if the geometry rotated?

Here is the underlying problem: Texels Don’t Rotate.
Help, My Texel Won’t Rotate!

- When we rotate Bump-Mapped geometry, the texels within the Normal Map don’t rotate along with the geometry.

- Even if we could rotate each texel, it would be an expensive proposition to perform a per-texel rotation matrix.

- This implies we need a way to move a light vector $L$ from World Space into the Normal Map’s coordinate Space.
Enter Texture Space

- Texture Space is a per-vertex coordinate system that expresses how to go from Model Space into the Normal Map Coordinate System, where:
  - +X Axis points to the Right
  - +Y Axis points Up
  - +Z Axis points Out of the screen
Texture Space Diagram

Normal Map – A flat plane in S,T direction

L and N are expressed in different coordinate systems

Solution = Rotate Light position into S,T,SxT space.

Result: New light position for each vertex.
How to Author for Texture Space

• The best method for generating Texture Space for your geometry is as follows:
  • Have the artist apply bump maps in their authoring tool – or just use the same mapping as the decal
    • Don’t let them use texture mirroring
    • Don’t use degenerate projections (ie stretched textures)
  • When loading in a model, create an extra set of 3 3D vectors per vertex
    • These will store the axes of the Texture Space basis
    • Generate the Texture Space vectors from the vertex positions and bump map texture coordinates
How to Generate Texture Space?

• For each triangle in the model:
  • Use the x,y,z position and the s,t bump map texture coordinates
  • Create plane equations of the form:
    • \( Ax + Bs + Ct + D = 0 \)
    • \( Ay + Bs + Ct + D = 0 \)
    • \( Az + Bs + Ct + D = 0 \)
  • Solve for the texture gradients \( \text{d}sx, \text{d}sy, \text{d}sz, \text{etc.} \)
Generating Texture Space

- Now treat the \( \text{dsdx}, \text{dsdy}, \text{dsdz} \) as a 3D vector representing the S axis \(< \text{dsdx}, \text{dsdy}, \text{dsdz} >\)
- Do the same to generate the T axis
- Now cross the two to generate the \( \text{SxT} \) axis – this is the ‘Z’ or up axis of Texture Space, and is typically close to parallel with the triangle’s normal
- If your \( \text{SxT} \) and the triangle normal point in opposite directions, the artist applied the texture backwards – have the artist fix this, or negate the SxT axis
Generating Texture Space

- These 3 Axes together make up a 3x3 rotation/scale matrix

\[
\begin{bmatrix}
\text{dsdx} & \text{dtdx} & \text{SxTx} \\
\text{dsdy} & \text{dtdy} & \text{SxTy} \\
\text{dsdz} & \text{dtdz} & \text{SxTz}
\end{bmatrix}
\]

Putting an XYZ model-space vector through this 3x3 matrix produces a vector expressed in local Texture Space
Per-Triangle Bases
Per-Triangle Bases

- We now have a coordinate basis for each triangle
- We need them on a per-vertex basis so they can vary smoothly across our geometry
- The solution:
  - For each vertex, sum up the S vectors from each face that shares this vertex
  - Do the same for all T and SxT vectors
  - Normalize each sum vector
  - Optionally scale by the average original magnitude of S,T or SxT if your texture map is applied anisotropically
  - The result is per-vertex Texture Space
  - This is analogous to calculating vertex normals for lighting
Per-Vertex Texture Space

• Now we have what we need to move a light into a local space defined at each vertex via the Texture Space Basis Matrix

• For each per-pixel light, we move it’s L or H vector into local Texture Space
  - On the CPU with C code
  - Or on the GPU with a vertex program
The Resulting Per-Vertex Texture Space

Per Vertex Texture Space is derived From Shared Faces
Texture Space In Practice

- The L or H vector is linearly interpolated across the polygon in Texture Space:
  - *In the diffuse or specular color*
    - It must be normalized before storing in the iterated color
    - It will get de-normalized across large polygons
    - Doesn’t handle anisotropy well
  - *Or in a set of 3D texture coordinates*
    - Use a Cube Map to renormalize the vector
      - Able to support scaling on textures
      - Can avoid CPU or GPU work
    - The L or H vector can be renormalized per-pixel via a texture, such as a Cube Map, Volume Map or Projected Texture
What about Animation?

• When triangles distort, so do their texture gradients, invalidating the Model Space->Texture Space matrix
• When triangles rotate relative to Model Space, the Model Space->Texture Space matrix is invalid
• Therefore, the Texture Space will need to be updated or recomputed during animation
• The obvious approach, and one practical for simple models, is to simply go through the previous steps for each animation frame
  • Regenerate Texture Space for each triangle, then each vertex
A Better Way – Update the Bases

- The two most popular animation techniques both work with Texture Space bump mapping WITHOUT requiring recalculating the entire basis
- Bone-Based Skinning (Indexed or Not)
- Keyframe Interpolation
Bone-Based Skinning

• For each axis of the Texture Space – S, T and SxT, “skin” the axis by putting it through the same matrix as the vertex normals

• Alternatively, skip the SxT axis and perform S cross T instead – can be cheaper if you have many bones
Keyframe Interpolation

- Create keyframes for the S, T and SxT axes as well.
- Linearly interpolate between the S(0) and S(1) using the keyframe weight from 0 to 1:
  \[(1 - \text{Weight}) \times S0 + (\text{Weight}) \times S1\]
- Now Normalize the result.
- To handle scaled or stretched textures:
  - Rescale by the linearly interpolated length of the two keyframe vectors.
  - \[\text{NormalizedVector} \times= \]
  \[(1 - \text{Weight}) \times \text{LengthOf}(S0) + (\text{Weight}) \times \text{LengthOf}(S1)\]
Keyframe Interpolation

- The normalizing of the vector approximates a spherical interpolation, or SLERP

- The rescaling ensures that any stretching or scaling in the textures is preserved
  - especially important if morphing
Texture Space Calculations

- These calculations are a prerequisite for practical Per-Pixel Lighting

- The cost of computing and updating Texture Space for moving models can seem large

- Keep it in perspective:
  - For a certain amount of per-vertex work, you are getting tremendous per-pixel detail

- All of the previous techniques for moving lights into Texture Space and updating the Texture Space vectors for moving objects can be handled with Dx8 Vertex Shaders
Texture Space Overview

• Texture Space is necessary to handle arbitrary, animating bump mapped geometry correctly

• It allows us to setup the per-pixel dot product

• For each vertex, we rotate the L or H vector into local Texture Space, where it is interpolated across the polygon as a color

• Now, we can perform L dot N or H dot N per pixel and achieve per-pixel lighting
Implementing Per-Pixel Light Types

- Directional Lights
- Spot Lights
  - Attenuation
- Point Lights
  - Attenuation
Directional Lights

- Directional Lights are the simplest – just perform $L \cdot N$ or $H \cdot N$, then multiply by the light color

  - $\text{LightColor} \times (L \cdot N)$

  or

  - $\text{LightColor} \times (H \cdot N)$
Spot Lights

- Spot lights are a little harder
- We need to use a projective texture to represent the spotlight’s cosine attenuation from the umbra and penumbra
- This is pre-generated as a circular texture map, like so:
Spot Light Directional Falloff

Penumbra

Umbra

Direction of light (-L)
Spot Light Texture

- The interior should hold the cosine falloff term from 1 (everywhere in the umbra) ramping down to 0 (at the edge of the penumbra)

- Ensure that the edges of the texture are black

- Set up a ‘Light Plane’ at the spotlight, pointing in the same direction as the light

- Set up texture coordinate generation and the texture matrix to so that the vertex positions are projected onto the Light Plane
Getting Clever

- Since it is a scalar value, one can place the cosine falloff term in the texture alpha only.
  
- That frees up the RGB values to hold the L vector.
  
- This has two benefits:
  - Per Pixel Spotlights with a single 2D texture
    - Handles self-shadowing automatically.
  - Put the distance attenuation in the diffuse color or alpha.
  
- \(( \text{SpotLight.RGB DOT3 Normal.RGB} ) * ( \text{SpotLight.Alpha * Diffuse.RGB} )\)
Point Lights

- Point Lights are similar to SpotLights in complexity
- They require a per-pixel distance attenuation value
- There are four basic ways to achieve this...
Four Attenuation Techniques

- 3D Texture holds Attenuation function
  - + Can be an arbitrary function
  - - Not all cards have 3D Textures
  - - Lots of Texture Memory

Use 2 2D Textures to compute $1 - d^2$

- $\left[ 1 - x^2 - y^2 \right] - \left[ z^2 \right]$

- Or Use 1 texture, and compute $z^2$ in texture blender via Diffuse or Specular
  - + Works on all cards
  - - Not very flexible for attenuation
Alternate 2 Texture Attenuation

- Use 2 2D Textures to compute $e^{d^*d}$
  - $[e^{(-x^2 - y^2)}] * [e^{(-z^2)}]$
  - + Works on All Cards
  - + Smoother Attenuation Function
  - + Can use other factors other than $e$
  - - Must use 2 Textures
Last Attenuation Technique

- 3D Texture - Store 3D L vector in RGB of texture, put Attenuation Function in Alpha only
  - + Less Textures used, easier to reduce passes
  - - Not all cards have 3D Textures
  - - Lots of Texture Memory
  - - May have to Point Sample if close to the Light
Attenuation Tips

• Always keep the edge of the Attenuation textures black if using scalars, or use the zero vector if encoding vectors, and use CLAMP mode.

• Use Alpha Test to eliminate pixels that map to the border of the Attenuation Map.

• Set up the texture coordinate generation / texture matrix to offset to the light’s position and scale by 1 / LightRange.

• You can use Destination Alpha to hold Attenuation for multi-pass effects.
<table>
<thead>
<tr>
<th>1-d*d</th>
<th>e^{-d*d}</th>
</tr>
</thead>
</table>

The table above shows two expressions: $1-d^2$ and $e^{-d^2}$. These are commonly used in various scientific and engineering applications, particularly in the context of image processing and computer graphics.
Bringing It All Together

- **At Author Time**
  - Apply Bump Maps (Actually Normal Maps) to your models
  - NEVER Pre-Light Textures
    - You can’t combine pre-lit and real-time lights
  - Storing global ambient light/shadows in a *separate* light map is OK

- **At Load Time**
  - Per Triangle
    - Generate Per-Triangle Texture Space
  - Per Vertex
    - Generate Texture Space Matrices from Per Triangle Bases
Bringing It All Together - Runtime

- Per Vertex
  - Move L or H into Local Texture Space

- Per Pixel
  - Perform Dot Product
  - Apply Attenuation
  - Apply Light Color
Multiple Per-Pixel Point Lights
What About Dx8 / Pixel Shaders?

• All of the preceding material applies directly to Dx8 and more advanced Pixel Shading

• Understanding the preceding sections is extremely helpful when investigating Dx8-Level Pixel Shading

• Dx8 Pixel Shaders are mostly extensions of the same ideas behind DOT3
What’s New With Dx8 Pixel Shaders?

• Math is performed in floating point instead of fixed point

• Can perform dependent textures
  • Texture1.S = ( Texture0.AR )
  • Texture1.T = ( Texture1.GB )

• Can perform a per-pixel reflection vector lookup into a CubeMap
  • True per-pixel bumpy reflections
Pixel Shaders Allow Per-Pixel Bumpy Reflections
Further Topics

- Using Cube Maps To Normalize Light Vectors
  - + Keeps Vectors Normalized
  - - Takes up a Texture
  - See my GDC 2000 presentation on Cube Maps

- Creating Normal Maps from Height Values or Other Textures
  - See my GDC 2000 presentation on Per-Pixel Lighting

- Both of these are employed in the Bump Maker tool on NVIDIA’s public developer website
Credits:
Where I First Learned of These Techniques

- Texture Space Generation Idea
  - Sim Dietrich
- Texture Space Generation Details
  - Sim Dietrich and Doug Rogers
- 3D Texture Attenuation
  - John Carmack
- SpotLight Attenuation w/ Normals
  - Sim Dietrich
- 1 – d*d Attenuation w/ 1 or 2 Textures
  - Sim Dietrich
- e ^ (-d*d) Attenuation w/ 2 Textures
  - Cass Everitt
- 3D Texture w/ Normals & Attenuation
  - Sim Dietrich