In-Game Special Effects and Lighting
Introduction

- Tomas Arce

Special Thanks
- Matthias Wloka
- Craig Galley
- Stephen Broumley
- Cyrus Lum
- Sumie Arce
- Inevitable
- nVidia
- Bungy

www.inevitable.com
What Is Per-Pixel Lighting

- Bad name
  - Texel-lighting is more accurate
- Texture resolution matters -- tiling helps
- Complex operation done per-pixel basis
  - Texture lookups, dp3, and pixel shaders
- Use colors as data
What is Per-Pixel Lighting

- Encode pixel-normals in texture, fill RGB as:
  \[ R = \text{Normal}.X, \ G = \text{Normal}.Y, \ B = \text{Normal}.Z \]

- Light dir is also encoded, fill vertex color as:
  \[ R = \text{LightDir}.X, \ G = \text{LightDir}.Y, \ B = \text{LightDir}.Z \]

- Now we can do:
  \[ (\text{Texture.Texel}) \cdot (\text{Vertex.Color}) \]
Why Is Per-Pixel Better?

- Per-vertex light is faster and more flexible but lacks resolution

- Light-maps have pit-falls
  - No real specular
  - Low resolution
  - Doesn’t work for dynamic objects

- Projected textures don’t give much detail for the surface
Why Is Per-Pixel Better?

- Detail. Detail. Detail.
- ppLighting needs few polygons per-mesh
  - Simplifies collision, stencil shadows, and memory
- Bump mapping is a subset of the ppLighting
  - "Normal maps"
- Normal maps handle different types of lights and surfaces well
Why is Per-Pixel Better?
What space?
- Local-Space vs Texture Space

Local Space Normals (LS)
- Use the origin of the object to extract normals

Texture Space Normals (TS)
- Store normals in generic space, e.g., Tangent Space
- Store a matrix per vertex that takes the light from local-space to texture space
How to Store Normals in Texels

Local Space                         Texture Space
How to Store Normals in Texels

- LS is simple to work with and can be very fast
  - But cannot be compressed
  - Good for characters and objects such as cars

- TS is more complex to work with but has 2 advantages
  - Tileable maps
  - Palletized textures
  - Good for big things like terrain with detail textures
Different Types of Lights

- 3 typical light types
  - Directional
  - Point
  - Spot

- Point and spot lighting usually have attenuation coefficients

- Store spot lighting attenuation function in a texture
  - X is a function of the distance
  - Y is a function of the angle
    The dot-product direction of the light with the vertex
Different Types of Lights

- Implementing attenuation function \( \frac{1}{(k_1 + d^*k_2 + d^*d^*k_3)} \) directly in the vertex shader
  - Takes few instructions, but has issues going towards zero

- Light vector normalization is usually not needed but...
  - Use cube-map lookup (32x32 or so)
  - Or use Newton-Raphson approximation in pixel-shader:
    ```
    mul r0, LightDir, 0.5
    dp3 r1, LightDir, LightDir
    mad r0, 1-r1, r0, LightDir
    ```
; LS Directional Light
; Transform position to clip space and output it
dp4 oPos.x, V_POSITION, c[CV_WORLDVIEWPROJ_0]
dp4 oPos.y, V_POSITION, c[CV_WORLDVIEWPROJ_1]
dp4 oPos.z, V_POSITION, c[CV_WORLDVIEWPROJ_2]
dp4 oPos.w, V_POSITION, c[CV_WORLDVIEWPROJ_3]

; Output tex coords
mov oT0, V_TEXTURE
mov oT1, V_TEXTURE
; TS Directional Light
; Transform position to clip space and output it
dp4 oPos.x, V_POSITION, c[CV_WORLDVIEWPROJ_0]
dp4 oPos.y, V_POSITION, c[CV_WORLDVIEWPROJ_1]
dp4 oPos.z, V_POSITION, c[CV_WORLDVIEWPROJ_2]
dp4 oPos.w, V_POSITION, c[CV_WORLDVIEWPROJ_3]

; Transform local space light by basis vectors to put it into texture space
dp3 LIGHT_T.x, V_S, c[L_DIR_LOCAL]
dp3 LIGHT_T.y, V_T, c[L_DIR_LOCAL]
dp3 LIGHT_T.z, V_Q, c[L_DIR_LOCAL]

; Scale to 0-1
add LIGHT_T, LIGHT_T, c[CV_ONE]
mul oD0, LIGHT_T, c[CV_HALF]

; Output tex coords
mov oT0, V_TEXTURE
mov oT1, V_TEXTURE
This example is mostly for rigid geometry
- Optimized TS: 11 instructions
- Optimizes LS: 6 instructions

Note that the LS could do up to 5 lights
- All light directions in local space are loaded into the pixel-shader constants

The TS uses light direction in local space
- Faster than transforming the basis vectors
LS vs TS Soft-Skin
// Transform pos with Weight 1
mov a0.x, V_INDICES.x

dp4 r1.x, V_POSITION, c[a0.x + CV_BONESTART + 0]
dp4 r1.y, V_POSITION, c[a0.x + CV_BONESTART + 1]
dp4 r1.z, V_POSITION, c[a0.x + CV_BONESTART + 2]

// Weight the light part 1
mul r7, c[a0.x + CV_LDIR_LOCALSPACE], V_WEIGHT0.x

// Transform pos with Weight 2
mov a0.x, V_INDICES.y

dp4 r2.x, V_POSITION, c[a0.x + CV_BONESTART + 0]
dp4 r2.y, V_POSITION, c[a0.x + CV_BONESTART + 1]
dp4 r2.z, V_POSITION, c[a0.x + CV_BONESTART + 2]

// Weight the 2 part of the light
mad r8, c[a0.x + CV_LDIR_LOCALSPACE], V_WEIGHT1.x, r7
// Blend between r1 and r2
mul r1.xyz, r1.xyz, V_WEIGHT0.x
mad r2, r2.xyz, V_WEIGHT1.x, r1.xyz

mov r2.w, c[CV_CONSTANTS].z //set w to one

// r2 now contains final position

; Do the texture compression by multiplying
; 1 or -1 as needed
mul r1.x, r1.x, V_MIRROR_FLAG

// Normalize light vector
dp3 r1.w, r1, r1
rsq r1.w, r1.w
mul r1, r1, r1.w
// Transform to clip space
dp4 oPos.x, r2, c[CV_WORLDVIEWPROJ_0]
dp4 oPos.y, r2, c[CV_WORLDVIEWPROJ_1]
dp4 oPos.z, r2, c[CV_WORLDVIEWPROJ_2]
dp4 oPos.w, r2, c[CV_WORLDVIEWPROJ_3]

// Scale to 0-1

// [-1, 1] --> [0, 1]
add r1, r1, c[CV_CONSTANTS].z
mul oD0, r1, c[CV_CONSTANTS].y

// Pass through texcoords
mov oT0.xy, V_TEX
mov oT1.xy, V_TEX
LS vs TS Soft-Skin  (T.S.) 4 of 7

// Transform pos with Weight 1
mov a0.x, V_INDICES.x

dp4 r1.x, V_POSITION, c[a0.x + CV_BONESTART + 0]
dp4 r1.y, V_POSITION, c[a0.x + CV_BONESTART + 1]
dp4 r1.z, V_POSITION, c[a0.x + CV_BONESTART + 2]

// Transform the light from the bone local space to the TS
dp3 r7.x, V_S,   c[a0.x + CV_LDIR_LOCALSPACE]
dp3 r7.y, V_T,   c[a0.x + CV_LDIR_LOCALSPACE]
dp3 r7.z, V_SxT, c[a0.x + CV_LDIR_LOCALSPACE]

// Transform pos with Weight 2
mov a0.x, V_INDICES.y

dp4 r2.x, V_POSITION, c[a0.x + CV_BONESTART + 0]
dp4 r2.y, V_POSITION, c[a0.x + CV_BONESTART + 1]
dp4 r2.z, V_POSITION, c[a0.x + CV_BONESTART + 2]
// Transform the light from the bone local space to the // texture space
dp3 r8.x, V_S, c[a0.x + CV_LDIR_LOCALSPACE]  
dp3 r8.y, V_T, c[a0.x + CV_LDIR_LOCALSPACE]  
dp3 r8.z, V_SXT, c[a0.x + CV_LDIR_LOCALSPACE]

// Blend between r1 and r2, r2 now contains final position  
mul r1.xyz, r1.xyz, V_WEIGHT0.x  
mad r2, r2.xyz, V_WEIGHT1.x, r1.xyz  
mov r2.w, c[CV_CONSTANTS].z // set w to one

// Blend the light  
mul r7, r7.xyz, V_WEIGHT0.x  
mad r8, r8.xyz, V_WEIGHT1.x, r7

// Normalize light vector  
dp3 r1.w, r1, r1  
rsq r1.w, r1.w  
mul r1, r1, r1.w
// Transform to clip space
dp4 oPos.x, r2, c[CV_WORLDVIEWPROJ_0]
dp4 oPos.y, r2, c[CV_WORLDVIEWPROJ_1]
dp4 oPos.z, r2, c[CV_WORLDVIEWPROJ_2]
dp4 oPos.w, r2, c[CV_WORLDVIEWPROJ_3]

// [-1, 1] --> [0, 1]
add r1, r1, c[CV_CONSTANTS].z
mul oD0, r1, c[CV_CONSTANTS].y

// Pass through texcoords
mov oT0.xy, V_TEX
mov oT1.xy, V_TEX
Optimized LS: 25 instructions
Optimized TS: 30 instructions

Skinning the light direction is as good as skinning a normal

The LS technique works well for characters
  - Unique pixel per polygon and symmetrical

Passing the light in the local space for each bone for the TS is a good idea
  - Standard TS technique is about 50 instructions
Evolution demo shader is 42 instructions
- Vertex-shader was the first bottle neck that we hit

Geforce3 Ti500 runs a 240Mhz
- 1 clock per instruction = 240M instruction/sec
- @ 60fps is 4M so 4M/25 = 160K Peak

The Geforce4 can do 600M instruction/sec
- Step in the right direction
1 to \(n\) lights and different types of lights
- Compile and cache vertex/pixel shaders on the fly
- Layout code so they can be combined
- Doom\textsc{III} does 1 light per pass

Geforce\textsc{3} has max. 8 instructions in the pixel-shader
- But can use an off-screen texture as a temporary register
- Then project texture in consequent passes
- Allows for unlimited length of pixel shaders (with extra cost of course)
“Anisotropic” textures are a good way to do complex lighting equations based on the specular and diffuse angles.
Use UVs to set the lightDir and half-vector
  - Then use “texp3x2pad” and “texp3x2tex” to compute the look up UVs

Pre-scale the lightDir and the half-vector by 0.5 to get the full range -1 to 1 of the light equation
  - Useful for back-lighting

Be creative with your lighting equation
  - “Diffuse + Specular” pretty much sucks
  - Try: “Diffuse + X * Specular * Diffuse”
Use multiple layers to achieve complex lighting
- Evolution demo had 3: Diffuse, Anisotropic, and ppSpecular

Make sure to use all channels in your textures
- Don’t forget about the alpha channel

Compress textures (DXT1, DXT3)

Post-Effects are becoming part of the shading technology; don’t miss that (check out Wreckless)
Wreckless

Nice job guys!
Two main techniques projected textures (nVidia/ATI/plain) and stencil volumes

Projected textures are easier to implement
  - Self-shadow
  - Can do 4 lights/shadows at a time
  - But can only do spot and directional lights

Hint: try to use orthogonal projections
  - Do per-object projected textures
- Stencils are a more generic solution
  - But fill rate is an issue

- Worth looking at
  - SVBSP and order tree structures
  - As well as Cut and Continue type of techniques

- Make sure to cache shadow volumes

- For perfect lighting, use stencil to *not* write lit pixels
  - vs darkening them
Massive over-draw!
In the near future new versions of vertex/pixel shaders will make another big leap.

Expect to see lighting independent of geometry complexity, and a final generalized lighting solution.

Shadows will still be a thing to try to achieve efficiently.
- Although it may be helped via fill-rate increases etc.

Post-effects will be very important as they start to became part of the final shader render.
We still live in very exciting times, but it would be more exciting if they do a real time ray-tracer!