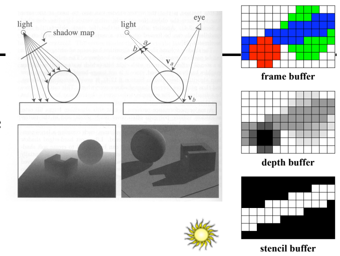
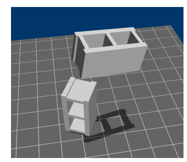
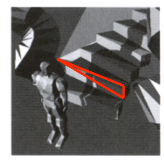
Programmable GPUS

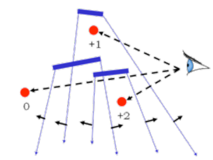
Last Time?

- Planar Shadows
- Projective Texture Shadows
- Shadow Maps
- Shadow Volumes
  - Stencil Buffer







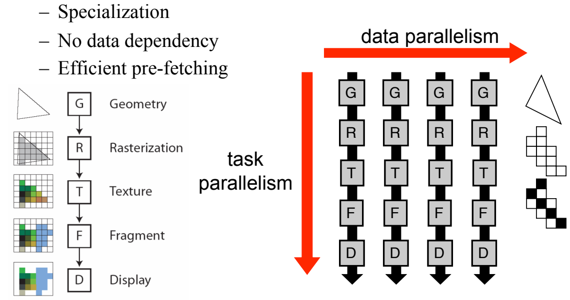


Today

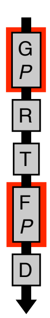
- Modern Graphics Hardware
- Shader Programming Languages
- Gouraud Shading vs. Phong Normal Interpolation
- Bump, Displacement, & Environment Mapping

Modern Graphics Hardware

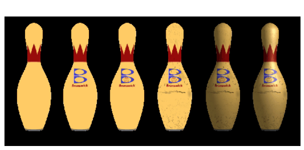
- High performance through
  - Parallelism
  - Specialization
  - No data dependency
  - Efficient pre-fetching



Programmable Graphics Hardware



- Geometry and pixel (fragment) stage become programmable
  - Elaborate appearance
  - More and more general-purpose computation (GPU hacking)



Misc. Stats on Graphics Hardware

- 2005
  - About 4-6 geometry units
  - About 16 fragment units
  - Deep pipeline (~800 stages)
  - 600 million vertices/second
  - 6 billion texels/second
- NVIDIA GeForce 9 (Feb 2008)
  - ~1 TFLOPS
  - 32/64 stream processors
  - 512 MB/1GB memory
- ATI Radeon R700 (2008)
  - 480 stream processing units
- NVIDIA® GeForce® GTX 480 (2010)
  - 480 cores
  - 2560x1600 resolution
  - 1536 MB memory

Today

- Modern Graphics Hardware
- Shader Programming Languages
- Gouraud Shading vs. Phong Normal Interpolation
- Bump, Displacement, & Environment Mapping

Emerging Languages

- Inspired by Shade Trees [Cook 1984] & Renderman Shading Language:
  - RTSL [Stanford 2001] – real-time shading language
  - Cg [NVIDIA 2003] – C for graphics
  - HLSL [Microsoft 2003] – Direct X
  - GLSL [OpenGL ARB 2004] – OpenGL 2.0
- General Purpose GPU computing
  - CUDA [NVIDIA 2007]
  - OpenCL (Open Computing Language) [Apple 2008] for heterogeneous platforms of CPUs & GPUs

Cg Design Goals

"Cg: A system for programming graphics hardware in a C-like language"
Mark et al. SIGGRAPH 2003

- Ease of programming
- Portability
- Complete support for hardware functionality
- Performance
- Minimal interference with application data
- Ease of adoption
- Extensibility for future hardware
- Support for non-shading uses of the GPU

Cg Design

- Hardware is changing rapidly… no single standard
- Specify "profile" for each hardware
  - May omit support of some language capabilities (e.g., texture lookup in vertex processor)
- Use hardware virtualization or emulation?
  - "Performance would be so poor it would be worthless for most applications"
  - Well, it might be ok for general purpose programming (not real-time graphics)

Cg compiler vs. GPU assembly

- Can inspect the assembly language produced by Cg compiler and perform additional optimizations by hand
  - Generally once development is complete (& output is correct)
  - Using Cg is easier than writing GPU assembly from scratch

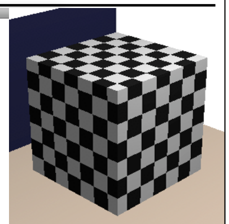
(Typical) Language Design Issues

- Parameter binding
- Call by reference vs. call by value
- Data types: 32 bit float, 16 bit float, 12 bit fixed & type-promotion (aim for performance)
- Specialized arrays or general-purpose arrays
  - `float4 x` vs. `float x[4]`
- Indirect addressing/pointers (not allowed…)
- Recursion (not allowed…)

GLSL example: checkerboard.vs

```
varying vec3 normal;
varying vec3 position_eyespace;
varying vec3 position_worldspace;

// a shader for a black & white checkerboard

void main(void) {
  position_eyespace = vec3(gl_ModelViewMatrix * gl_Vertex);
  position_worldspace = gl_Vertex.xyz;
  normal = normalize(gl_NormalMatrix * gl_Normal);
  gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

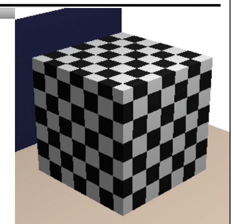
```
--:---  checkerboard.vs   All L13   CVS-1.1  (C/l Abbrev)---------------------------
```



GLSL example: checkerboard.fs

```
varying vec3 normal;
varying vec3 position_eyespace;
varying vec3 position_worldspace;

// a shader for a black & white checkerboard

void main (void) {

  vec3 color;

  // determine the parity of this point in the 3D checkerboard
  int count = 0;
  if (mod(position_worldspace.x,0.3)> 0.15) count++;
  if (mod(position_worldspace.y,0.3)> 0.15) count++;
  if (mod(position_worldspace.z,0.3)> 0.15) count++;
  if (count == 1 || count == 3) {
    color = vec3(0.1,0.1,0.1);
  } else {
    color = vec3(1,1,1);
  }

  // direction to the light
  vec3 light = normalize(gl_LightSource[1].position.xyz - position_eyespace);

  // basic diffuse
  float ambient = 0.3;
  float diffuse = 0.7*max(dot(normal,light),0.0);
  color = ambient*color + diffuse*color;
  gl_FragColor = vec4 (color, 1.0);
}
```

```
--:---  checkerboard.fs   All L1    CVS-1.1  (C/l Abbrev)---------------------------
ESC <mouse-1> is undefined
```

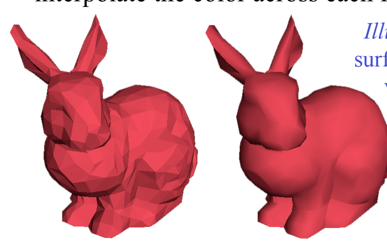


Today

- Modern Graphics Hardware
- Shader Programming Languages
- Gouraud Shading vs. Phong Normal Interpolation
- Bump, Displacement, & Environment Mapping

Remember Gouraud Shading?

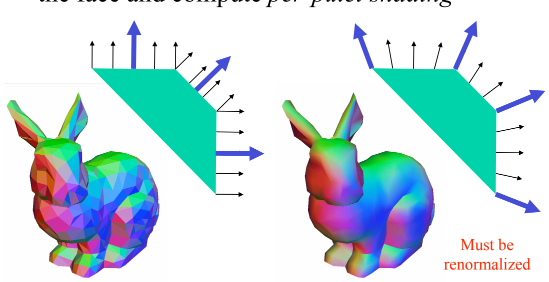
- Instead of shading with the normal of the triangle, shade the vertices with the *average normal* and interpolate the color across each face



Illusion of a smooth surface with smoothly varying normals

Phong Normal Interpolation (Not Phong *Shading*)

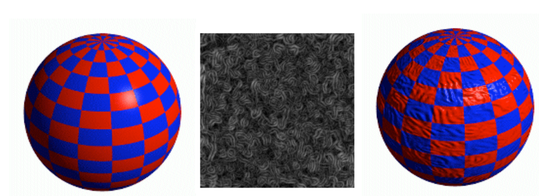
- Interpolate the average vertex normals across the face and compute *per-pixel shading*



Must be renormalized

Bump Mapping

- Use textures to alter the surface normal
  - Does not change the actual shape of the surface
  - Just shaded as if it were a different shape

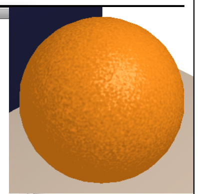


Sphere w/Diffuse Texture | Swirly Bump Map | Sphere w/Diffuse Texture & Bump Map

Another GLSL example: orange.vs

```
varying vec3 normal;
varying vec3 position_eyespace;
varying vec3 position_worldspace;

// a shader that looks like orange peel

void main(void) {

  // the fragment shader requires both the world space position (for
  // consistent bump mapping) & eyespace position (for the phong
  // specular highlight)
  position_eyespace = vec3(gl_ModelViewMatrix * gl_Vertex);
  position_worldspace = gl_Vertex.xyz;

  // pass along the normal
  normal = normalize(gl_NormalMatrix * gl_Normal);

  gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
 }
```

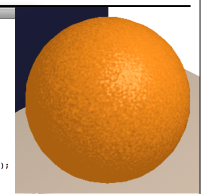
`--:---  orange.vs      All L1     CVS-1.1   (C/l Abbrev)----`



Another GLSL example: orange.fs

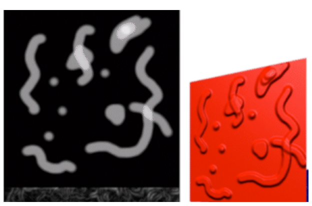
```
varying vec3 normal;
varying vec3 position_eyespace;
varying vec3 position_worldspace;

// a shader that looks like orange peel

void main (void) {

  // the base color is orange!
  vec3 color = vec3(1.0,0.5,0.1);

  // high frequency noise added to the normal for the bump map
  vec3 normal2 = normalize(normal+0.4*noise3(70.0*position_worldspace));

  // direction to the light
  vec3 light = normalize(gl_LightSource[1].position.xyz - position_eyespace);
  // direction to the viewer
  vec3 eye_vector = normalize(-position_eyespace);
  // ideal specular reflection
  vec3 reflected_vector = normalize(-reflect(light,normal2));

  // basic phong lighting
  float ambient = 0.6;
  float diffuse = 0.4*max(dot(normal2,light),0.0);
  float specular = 0.2 * pow(max(dot(reflected_vector,eye_vector),0.0),10.0);
  vec3 white = vec3(1.0,1.0,1.0);
  color = ambient*color + diffuse*color + specular*white;
  gl_FragColor = vec4 (color, 1.0);
}
```

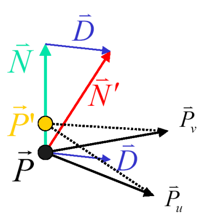
`--:---  orange.fs      All L1     CVS-1.1   (C/l Abbrev)----`



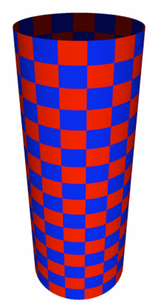
Bump Mapping

- Treat the texture as a single-valued height function
- Compute the normal from the partial derivatives in the texture

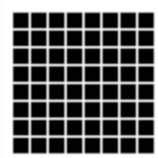




Another Bump Map Example



Cylinder w/Diffuse Texture Map



Bump Map

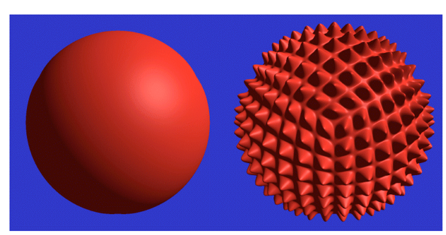
Cylinder w/Texture Map & Bump Map

What's Missing?

- There are no bumps on the silhouette of a bump-mapped object
- Bump maps don’t allow self-occlusion or self-shadowing

Displacement Mapping

- Use the texture map to actually move the surface point
- The geometry must be displaced before visibility is determined



Displacement Mapping

Image from:

Geometry Caching for
Ray-Tracing Displacement Maps
EGRW 1996
Matt Pharr and Pat Hanrahan

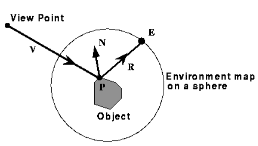
note the detailed shadows cast by the stones

Displacement Mapping

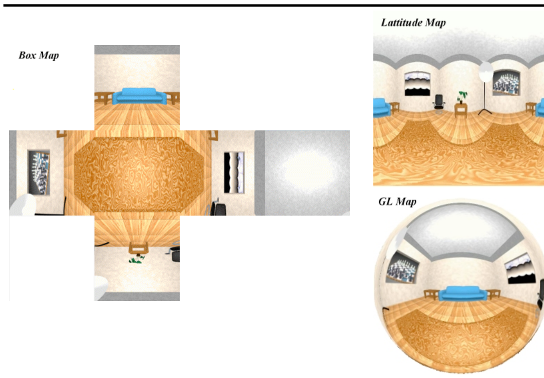
Ken Musgrave

Environment Maps

- We can simulate reflections by using the direction of the reflected ray to index a spherical texture map at "infinity".
- Assumes that all reflected rays begin from the same point.



What's the Best Chart?

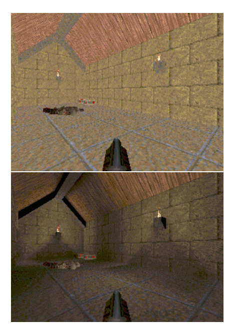


Environment Mapping Example

Terminator II

Texture Maps for Illumination

- Also called "Light Maps"



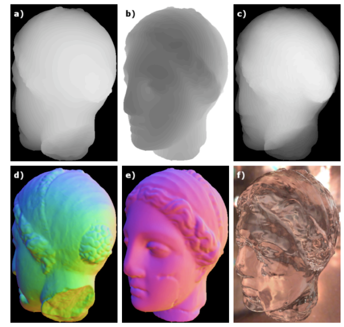
Quake

Questions?

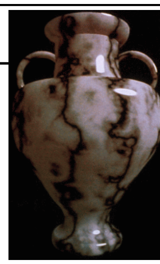
Image by Henrik Wann Jensen
Environment map by Paul Debevec

Reading for Today:

- Chris Wyman, "An Approximate Image-Space Approach for Interactive Refraction", SIGGRAPH 2005



Readings for Friday:



Choose:

- “An Image Synthesizer”, Perlin, SIGGRAPH 1985 & “Improving Noise”, Perlin, SIGGRAPH 2002
- “Parallel White Noise Generation on a GPU via Cryptographic Hash”, Tzeng & Wei, I3D 2008.