Programmable GPUS

Last Time?

- Planar Shadows
- Projective Texture Shadows
- Shadow Maps
- Shadow Volumes
  - Stencil Buffer
Homework 4

- Create some geometry
  - Reflected object & floor
  - Silhouette edges
  - Shadow polygons
    - Make sure your polygons aren’t doubled up
    - Make sure your polygons are oriented consistently

- Mess with the stencil buffer
  - Don’t just blindly copy code from the tutorial
  - Use the web to read the man page for each instruction & its parameters

- Be creative with shaders
  - Hopefully everyone can get the examples to compile & run

Questions?

- From a previous quiz: Check the boxes to indicate the features & limitations of each technique

<table>
<thead>
<tr>
<th>Features / Limitations</th>
<th>Planar Fake Shadows</th>
<th>Projective Texture Shadows</th>
<th>Shadow Maps</th>
<th>Shadow Volumes</th>
<th>Ray Casting Shadows</th>
</tr>
</thead>
<tbody>
<tr>
<td>Allows objects to cast shadows on themselves (self shadowing)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Permits shadows on arbitrary surfaces (i.e. curved)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Renders geometry from the viewpoint of the light</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Generates extra geometric primitives</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Limited resolution of intermediate representation can result in jaggy shadow artifacts</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Reading for Tuesday:


- Shadow volumes win for dynamic scenes
- Value in considering all algorithms: simple but expensive, complex but fast. Don’t eliminate options from consideration too early.
- When were ray tracing shadows introduced?
- Combining contour edges optimization
- Can much of this can be done on the GPU now?
- Debugging this code at the time (slower computers and no internet for reference!) would be difficult
- Good comparison of algorithms
- Strange how there was very little math
  Easy to read but not necessarily easy to understand
- Awful font choice
Reading for HW4:

• “Improving Shadows and Reflections via the Stencil Buffer”, Mark Kilgard, NVIDIA

• Long but well-written, tutorial rather than academic paper
• Would like more explanation for some of the functions (expects reader to know them already)
• How different are OpenGL vs. DirectX?
• Grr to see classic immediate mode rendering (glBegin, glVertex, instead of VBOs)
• Maybe ray tracing be more efficient for multiple mirror reflections?
• Good at presenting algorithm + limitations + revised algorithm to fix those problems
Today

- Modern Graphics Hardware
- Shader Programming Languages
- Gouraud Shading vs. Phong Normal Interpolation
- Many “Mapping” techniques

Modern Graphics Hardware

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

![Diagram showing data parallelism and task parallelism in modern graphics hardware](image)
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
  - 4-6 geometry units, 16 fragment units
  - Deep pipeline (~800 stages)
- NVIDIA GeForce 9 (Feb 2008)
  - 32/64 cores, 512 MB/1GB memory
- ATI Radeon R700 (2008)
  - 480 stream processing units
- NVIDIA GeForce GTX 480 (2010)
  - 480 cores, 1536 MB memory
  - 2560x1600 resolution
- ATI Radeon HD 7900 (2012)
  - 2048 processors, 3GB memory
- NVIDIA GeForce GTX 680 (2012)
  - 1536 cores, 2040 MB memory
- NVIDIA GeForce GTX 980 (2014)
  - 2048 core, 4 GB RAM
- AMD Radeon R9 275x2 (2014)
  - Water cooling, 1200 power supply
  - 5,632 processors, 8 GB RAM,
Today

- Modern Graphics Hardware
- **Shader Programming Languages**
  - Cg design goals
  - GLSL examples
- Gouraud Shading vs. Phong Normal Interpolation
- Many “Mapping” techniques

Emerging & Evolving Languages

- Inspired by Shade Trees [Cook 1984] & Renderman Shading Language [1980’s]:
  - RTSL [Stanford 2001] – real-time shading language
  - Cg [NVIDIA 2003] – “C for graphics”
  - GLSL [OpenGL ARB 2004] – OpenGL 2.0
  - Optix [NVIDIA 2009] – Real time ray tracing engine for CUDA
- General Purpose GPU computing
  - CUDA [NVIDIA 2007]
  - OpenCL (Open Computing Language) [Apple 2008] for heterogeneous platforms of CPUs & GPUs
Cg Design Goals

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

Cg Design

- Hardware is changing rapidly [2003]… 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…)
Today

- Modern Graphics Hardware
- Shader Programming Languages
  - Cg design goals
  - GLSL examples
- Gouraud Shading vs. Phong Normal Interpolation
- Many “Mapping” techniques

GLSL example: checkerboard.vs (GLUT)

```cpp
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;
}
```
GLSL example: hw4_shader.vs (GLFW)

```cpp
void main()
{
  // Output position of the vertex, in clip space: M * position
  gl_Position = MVP * vec4(position_worldspace, 1.0);
  // Position of the vertex, in workspace: M * position
  vertexPosition_worldspace = (M * vec4(position_worldspace, 1.0)).xyz;
  // Vector that goes from the vertex to the camera, in camera space.
  // In camera space, the camera is at the origin (0,0,0).
  vec3 viewDirection_worldspace = (V * M * vec4(vertexPosition_worldspace, 1.0)).xyz;
  vertexNormal_worldspace = normalize(V * vec4(normal_worldspace, 0.0)).xyz;
  // Pass color to the fragment shader.
  myColor = vec4(color);
}
```

GLSL example: checkerboard.fs (GLUT)

```cpp
// 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, 1.0) < 0.5) count++;
  if (mod(position_worldspace.y, 1.0) < 0.5) count++;
  if (mod(position_worldspace.z, 1.0) < 0.5) count++;
  if (count % 2 == 1) color = vec4(0.0, 0.0, 0.0, 1.0);
  else color = vec4(1.0, 1.0, 1.0, 1.0);
  // direction to the light
  vec3 light = normalize(gl_LightSource[1].position_worldspace);
  // basic diffuse
  float ambient = 0.3;
  float diffuse = 0.7 * max(dot(normal_worldspace, light), 0.0);
  color = color * diffuse + color;
  gl_FragColor = vec4(color, 1.0);
}
```
GLSL example: hw4_shader.fs (GLFW)

```glsl
ivec3 vertNormal_worldspace;

// Output data
out vec3 color;

// Values that stay constant for the whole mesh.
uniform vec3 LightPosition_worldspace;
uniform int whichshader;


// a shader for a black & white checkerboard
vec3 checkerboard(ivec3 xy) {
  // determine the parity of this point in the 3D checkerboard
  int count = 0;
  if (mod(xy.x, 8) >= 8) count++;
  if (mod(xy.y, 8) >= 8) count++;
  if (mod(xy.z, 8) >= 8) count++;
  if (count == 1 || count == 3) {
    return vec3(0, 1, 0);
  } else {
    return vec3(1, 1, 1);
  }
}

// Material properties
vec3 MaterialDiffuseColor = myColor;
if (whichshader == 1) {
  MaterialDiffuseColor = checkerboard(vertPosition_worldspace);
} else if (whichshader == 2) {
  vec3 normal2;
  MaterialDiffuseColor = orange(vertPosition_worldspace, surface_normal);
} else if (whichshader == 3) {
  MaterialDiffuseColor = wood(vertPosition_worldspace, surface_normal);
}
```

Remember Gouraud Shading?

- Instead of shading with the normal of the triangle, we’ll shade the vertices with the average normal and interpolate the shaded color across each face
  - This gives the 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
  - Normals should be re-normalized (ensure length=1)

Before shaders, per-pixel shading was not possible in hardware (Gouraud shading is actually a decent substitute!)

Today

- Modern Graphics Hardware
- Shader Programming Languages
- Gouraud Shading vs. Phong Normal Interpolation
- Many “Mapping” techniques
  - Bump Mapping
  - Displacement Mapping
  - Environment Mapping
  - Light Mapping
  - Normal Mapping
  - Parallax Mapping
  - Parallax Occlusion Mapping
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

Another GLSL example: orange.vs

```glsl
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 highlights)
    position_eyespace = vec3(gl_ModelViewMatrix * gl_Vertex);
    position_worldspace = gl_Vertex.xyz;

    // pass along the normal
    normal = normalise(gl_NormalMatrix * gl_Normal);
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```
Another GLSL example: orange.fs

```glsl
varying vec3 normal;
varying vec3 position_worldspace;

// a shader that looks like orange peel
void main (void) {
    // the base color is orange!
    vec3 color = vec3(1.0, 0.0, 0.1);
    // high frequency noise added to the normal for the bump map
    vec3 normal2 = normalize(normal+0.4*noise[70.0*position_worldspace]);
    // direction to the light
    vec3 light = normalize(gl_LightSource[1].position_worldspace - position_worldspace);
    // direction to the viewer
    vec3 eye_vector = normalize(-position_worldspace);
    // ideal specular reflection
    vec3 reflected_vector = normalize(-reflect(light, normal));
    // basic phong lighting
    float ambient = 0.6;
    float diffuse = 0.4*max(dot(normal2, light), 0.0);
    float specular = 2.0 * 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);
}
```

Bump Mapping

- Treat a greyscale 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

Normal Mapping

• Variation on Bump Mapping:
  Use an RGB texture to directly encode the normal

original mesh
4M triangles

simplified mesh
500 triangles

simplified mesh
and normal mapping
500 triangles

http://en.wikipedia.org/wiki/File:Normal_map_example.png
What's Missing?

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

Today

- Modern Graphics Hardware
- Shader Programming Languages
- Gouraud Shading vs. Phong Normal Interpolation
- Many “Mapping” techniques
  - Bump Mapping
  - Displacement Mapping
  - Environment Mapping
  - Light Mapping
  - Normal Mapping
  - Parallax Mapping
  - Parallax Occlusion Mapping
Displacement Mapping

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

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

- Displace the texture coordinates for each pixel based on view angle and value of the height map at that point
- At steeper view-angles, texture coordinates are displaced more, giving illusion of depth due to parallax effects

Parallax Mapping  

a.k.a. Offset Mapping or Virtual Displacement Mapping

- Displace the texture coordinates for each pixel based on view angle and value of the height map at that point
- At steeper view-angles, texture coordinates are displaced more, giving illusion of depth due to parallax effects

“Detailed shape representation with parallax mapping”, Kaneko et al. ICAT 2001
Parallax Occlusion Mapping

- Brawley & Tatarchuk 2004
- Per pixel ray tracing of the heightfield geometry
- Occlusions & soft shadows

http://developer.amd.com/media/gpu_assets/
Tatarchuk-ParallaxOcclusionMapping-Sketch-print.pdf

Today

- Modern Graphics Hardware
- Shader Programming Languages
- Gouraud Shading vs. Phong Normal Interpolation
- Many “Mapping” techniques
  - Bump Mapping
  - Displacement Mapping
  - Environment Mapping
  - Light Mapping
  - Normal Mapping
  - Parallax Mapping
  - Parallax Occlusion Mapping
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?

- Box Map
- Latitude Map
- GL Map
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 Friday:

• Chris Wyman, "An Approximate Image-Space Approach for Interactive Refraction”, SIGGRAPH 2005
Readings for Tuesday:

Choose: