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

Reading for Today
- "Rendering Fake Soft Shadows with Smoothies", Chan & Durand, EGSR 2003
Reading for Today

• "Deep Opacity Maps",
  Yuksel and Keyser, Eurographics 2008

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
  - 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)
  - 5536 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
- NVIDIA GeForce 10 (2016)
  - Almost 4,000 (shaders, texture map, render output)

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

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

// a shader for a black & white checkerboard
void main(void) {
    position_worldspace = 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)

```glsl
// Input vertex data, different for all executions of this shader.
layout(location = 0) in vec3 vertexPosition, multipass;
layout(location = 1) in vec3 vertexNormal, multipass;
layout(location = 2) in vec3 vertexColor, multipass;

// Output data
out vec3 vertexPosition_worldspace;
out vec3 vertexNormal_worldspace;
out vec3 vertexColor_worldspace;
out vec3 vertexProjection_worldspace;
out vec3 vertexEyespace;

// Values that stay constant for the whole mesh.
uniform mat4 MVP; uniform mat4 V; uniform mat4 M;

void main() {
    // Output position of the vertex, in clip space : MVP * position
    gl_Position = MVP * vec4(vertexPosition, multipass[11], 1.0);
    // Position of the vertex, in worldspace : M * position
    vertexPosition_worldspace = vec4(M * vec4(vertexPosition, multipass[11], 1.0), 1.0);
    // 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 vertexPosition_worldspace = (Y * vec4(vertexPosition_worldspace, 1.0));
    // Eye space coordinate is vec3(0.0, 0.0) -> vertexPosition_worldspace;
    vertexEyespace_worldspace = normalize(X * vec4(vertexPosition_worldspace, 1.0));
    // pass color to the fragment shader
    multipass[0] = vertexColor_worldspace;
}
```

GLSL example: checkerboard.fs (GLUT)

```glsl
varying vec3 normal;
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,1.1) > 0.15) count++;
    if (mod(position_worldspace.y,1.1) > 0.15) count++;
    if (count == 1) count = 0;
    color = vec3(1,0,0);
    else {
        color = vec3(1,1,1);
    }
    // direction to the light
    vec3 light = normalize(gl_LightSource[1].position.xyz - position_worldspace);
    // 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);
}
```

GLSL example: hw4_shader.fs (GLFW)

```glsl
in vec3 vertexPosition_worldspace;
in vec3 vertexNormal_worldspace;

// Values that stay constant for the whole mesh.
uniform lightPosition_worldspace;
uniform lightColor;
uniform mat4 MVP;
uniform mat4 V;
uniform mat4 M;

void main() {
    // a shader for a black & white checkerboard
    vec3 checkerboard(vec3 pos) {
        // determine the parity of this point in the 3D checkerboard
        int count = 0;
        if (mod(pos.x,1.1) > 0.15) count++;
        if (mod(pos.y,1.1) > 0.15) count++;
        if (count == 1) count = 0;
        return vec3(1,0,0);
    }
    // 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);
}
```

The above code snippets represent GLSL shaders for the checkerboard and hw4_shader examples. The first snippet is for glut, and the second is for glfw. Each snippet demonstrates how to create a shader for a black and white checkerboard or a general shader for hw4 that can be used with different lighting and material properties.
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
Bump Mapping

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

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

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
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*

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

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?

Environment Mapping Example

Terminator II
Texture Maps for Illumination

- Also called "Light Maps"

Questions?

Image by Henrik Wann Jensen
Environment map by Paul Debevec

Reading for Friday


Reading for Friday

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