To Do
- Final Projects due Jun 9
- PLEASE FILL OUT CAPE EVALUATIONS!!
- KEEP WORKING HARD

Motivation
- Today, create photorealistic computer graphics
  - Complex geometry, lighting, materials, shadows
  - Computer-generated movies/special effects (difficult or impossible to tell real from rendered…)
- CSE 168 images from rendering competition (2011)
- But algorithms were very slow (hours to days)

Real-Time Rendering
- Goal: interactive rendering. Critical in many apps
  - Games, visualization, computer-aided design, …
- Until 15-20 years ago, focus on complex geometry

Evolution of 3D graphics rendering
Interactive 3D graphics pipeline as in OpenGL
- Earliest SGI machines (Clark 82) to today
- Most of focus on more geometry, texture mapping
- Some tweaks for realism (shadow mapping, accum. buffer)

Offline 3D Graphics Rendering
Ray tracing, radiosity, photon mapping
- High realism (global illum, shadows, refraction, lighting…)
- But historically very slow techniques

“So, while you and your children’s children are waiting for ray tracing to take over the world, what do you do in the meantime?” — Real-Time Rendering

Pictures courtesy Henrik Wann Jensen
New Trend: Acquired Data

- Image-Based Rendering: Real/precomputed images as input
- Also, acquire geometry, lighting, materials from real world
- Easy to obtain or precompute lots of high quality data. But how do we represent and reuse this for (real-time) rendering?

20 years ago

- High quality rendering: ray tracing, global illumination
  - Little change in CSE 168 syllabus, from 2003 to today
- Real-Time rendering: Interactive 3D geometry with simple texture mapping, fake shadows (OpenGL, DirectX)
  - Complex environment lighting, real materials (velvet, satin, paints), soft shadows, caustics often omitted in both
  - Realism, interactivity at cross purposes

Today: Real-Time Game Renderings

- Vast increase in CPU power, modern instrs (SSE, Multi-Core)
  - Real-time raytracing techniques are possible (even on hardware: NVIDIA OptiX, RTX Raytracing)
- 4th generation of graphics hardware is programmable
  - (First 3 gens were wireframe, shaded, textured)
  - Modern NVIDIA, ATI cards allow vertex, fragment shaders
- Great deal of current work on acquiring and rendering with realistic lighting, materials… [Especially at UCSD]
  - Focus on quality of rendering, not quantity of polygons, texture

Goals

- Overview of basic techniques for high-quality real-time rendering
- Survey of important concepts and ideas, but do not go into details of writing code
- Some pointers to resources, others on web
- One possibility for final project, will need to think about some ideas on your own

Outline

- Motivation and Demos
- Programmable Graphics Pipeline
- Shadow Maps
- Environment Mapping
High quality real-time rendering

- Photorealism, not just more polygons
- Natural lighting, materials, shadows

Interiors by architect Frank Gehry. Note rich lighting, ranging from localized sources to reflections off vast sheets of glass.

Real materials diverse and not easy to represent by simple parameteric models. Want to support measured reflectance.

Today: Full Global Illumination

- Photorealism, not just more polygons
- Natural lighting, materials, shadows

Agrawala et al. 00
Ng et al. 03

Natural lighting creates a mix of soft diffuse and hard shadows.

Applications

- Entertainment: Lighting design
- Architectural visualization
- Material design: Automobile industry
- Realistic Video games
- Electronic commerce

Programmable Graphics Hardware
Programmable Graphics Hardware

NVIDIA a new dawn demo (may need to type URL)

Precomputation-Based Methods

- Static geometry
- Precomputation
- Real-Time Rendering (relight all-frequency effects)
- Involves sophisticated representations, algorithms

Relit Images

Ng, Ramamoorthi, Hanrahan 04

Video: Real-Time Relighting

Spherical Harmonic Lighting

Avatar 2010, based on Ramamoorthi and Hanrahan 04, Sloan 02

Interactive RayTracing

Advantages
- Very complex scenes relatively easy (hierarchical bbox)
- Complex materials and shading for free
- Easy to add global illumination, specularities etc.

Disadvantages
- Hard to access data in memory-coherent way
- Many samples for complex lighting and materials
- Global illumination possible but expensive

Modern developments: Leverage power of modern CPUs, develop cache-aware, parallel implementations
Recent developments make real-time raytracing mainstream (NVIDIA OptiX 5 in 2017, RTX chips in 2018, denoise, DLSS)

https://www.youtube.com/watch?v=kcP1NzB49zU
Sparse Sampling, Reconstruction

- Same algorithm as offline Monte Carlo rendering
- But with smart sampling and filtering (current work)

NVIDIA RTX Real-Time RayTracing

- Extend AAF, FSF, MAAF: Predict Filter based on Deep Learning (sample and AI-based denoising)
- NVIDIA software (OptiX 2017), hardware (RTX 2018)
- 40-year journey: ray tracing curiosity to every pixel

From SIGGRAPH 18

Outline

- Motivation and Demos
- Programmable Graphics Pipeline
- Shadow Maps
- Environment Mapping
Basic Hardware Pipeline

Application → Geometry → Rasterizer

CPU → GPU

Create geometry, lights, materials, textures, cubemaps, ... as inputs
Transform and lighting calcs.
Apply per-vertex operations
Textures, Cubemaps
Per-pixel (per-fragment) operations

Geometry or Vertex Pipeline

Model, View Transform → Lighting → Projection → Clipping → Screen

These fixed function stages can be replaced by a general per-vertex calculation using vertex shaders in modern programmable hardware

Pixel or Fragment Pipeline

Rasterization (scan conversion) → Texture Mapping → Z-buffering → Framebuffer

These fixed function stages can be replaced by a general per-fragment calculation using fragment shaders in modern programmable hardware

OpenGL Rendering Pipeline

Vertices → Scan Conversion (Rasterize) → Geometry Primitive Operations → Fragment Operations → Programmable in Modern GPUs (Fragment Shader)

Images → Pixel Operations → Texture Memory

Traditional Approach: Fixed function pipeline (state machine)
New Development (2003-): Programmable pipeline

Simplified OpenGL Pipeline

- User specifies vertices (vertex buffer object)
- For each vertex in parallel
  - OpenGL calls user-specified vertex shader:
    - Transform vertex (ModelView, Projection), other ops
- For each primitive, OpenGL rasterizes
  - Generates a fragment for each pixel the fragment covers
- For each fragment in parallel
  - OpenGL calls user-specified fragment shader:
    - Shading and lighting calculations
    - OpenGL handles z-buffer depth test unless overwritten
- Modern OpenGL is “lite” basically just a rasterizer
  - “Real” action in user-defined vertex, fragment shaders

Shading Languages

- Vertex / Fragment shading described by small program
- Written in language similar to C but with restrictions
- Long history. Cook’s paper on Shade Trees, Renderman for offline rendering
- Stanford Real-Time Shading Language, work at SGI
- Cg from NVIDIA, HLSL
- GLSL directly compatible with OpenGL 2.0 (So, you can just read the OpenGL Red Book to get started)
Shader Setup

- Initializing (shader itself discussed later)
- Create shader (Vertex and Fragment)
- Compile shader
- Attach shader to program
- Link program
- Use program
- Shader source is just sequence of strings
- Similar steps to compile a normal program

Shader Initialization Code

```c
GLuint initshaders (GLenum type, const char *filename) {
  GLuint shader = glCreateShader(type) ;
  GLint compiled ;
  string str = textFileRead (filename) ;
  const char * cstr = str.c_str() ;
  glShaderSource (shader, 1, &cstr, NULL) ;
  glCompileShader (shader) ;
  GLint compiled ;
  glGetShaderiv (shader, GL_COMPILE_STATUS, &compiled) ;
  if (!compiled) { shadererrors (shader) ; throw 3 ; } return shader ;
}
```

Linking Shader Program

```c
GLuint initprogram (GLuint vertexshader, GLuint fragmentshader) {
  GLuint program = glCreateProgram() ;
  GLint linked ;
  glAttachShader(program, vertexshader) ;
  glAttachShader(program, fragmentshader) ;
  glLinkProgram(program) ;
  GLint linked ;
  glGetProgramiv(program, GL_LINK_STATUS, &linked) ;
  if (linked) glUseProgram(program) ;
  else { programerrors(program) ; throw 4 ; } return program ;
}
```

Phong Shader: Vertex

```c
void main(void) {
  v = vec3(gl_ModelViewMatrix * gl_Vertex); 
  N = normalize(gl_NormalMatrix * gl_Normal); 
  gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex; 
}
```

Phong Shader: Fragment

```c
void main(void) {
  vec4 lambert = lightcolor * max(dot(N,direction), 0.0); 
  vec4 phong = lightcolor * pow(max(dot(N,halfvec), 0.0), myshininess); 
  vec4 retval = lambert + phong; 
  return retval ;
}
```
Outline

- Motivation and Demos
  - Programmable Graphics Pipeline
  - Shadow Maps
  - Environment Mapping

Shadow and Environment Maps

- Basic methods to add realism to interactive rendering
- Shadow maps: image-based way hard shadows
  - Very old technique. Originally Williams 78
  - Many recent (and older) extensions
  - Widely used even in software rendering (RenderMan)
  - Simple alternative to raytracing for shadows
- Environment maps: image-based complex lighting
  - Again, very old technique. Blinn and Newell 76
  - Huge amount of recent work (some covered in course)
- Together, give most of realistic effects we want
  - But cannot be easily combined!!
  - See Annen 08 [real-time all-frequency shadows dynamic scenes] for one approach: convolution soft shadows

Common Real-time Shadow Techniques

- Projected planar shadows
  - works well only on flat surfaces
- Stenciled shadow volumes
  - determining the shadow volume is hard work
- Light maps
  - totally unsuited for dynamic shadows
- In general, hard to get everything shadowing everything

Problems

- Mostly tricks with lots of limitations
  - Projected planar shadows
  - Stenciled shadow volumes
  - Light maps
  - In general, hard to get everything shadowing everything

Shadow Mapping

- Lance Williams: Brute Force in image space (shadow maps in 1978, but other similar ideas like Z buffer, bump mapping using textures and so on)
- Completely image-space algorithm
  - no knowledge of scene’s geometry is required
  - must deal with aliasing artifacts
- Well known software rendering technique
  - Basic shadowing technique for Toy Story, etc.

Phase 1: Render from Light

- Depth image from light source
Phase 1: Render from Light
- Depth image from light source

Phase 2: Render from Eye
- Standard image (with depth) from eye

Phase 2+: Project to light for shadows
- Project visible points in eye view back to light source

(Reprojected) depths match for light and eye. VISIBLE

(Reprojected) depths from light, eye not the same. BLOCKED!!

Visualizing Shadow Mapping
- A fairly complex scene with shadows

Compare with and without shadows

with shadows

without shadows
Visualizing Shadow Mapping
- The scene from the light’s point-of-view

Visualizing Shadow Mapping
- The depth buffer from the light’s point-of-view

Visualizing Shadow Mapping
- Projecting the depth map onto the eye’s view

Visualizing Shadow Mapping
- Comparing light distance to light depth map

Hardware Shadow Map Filtering
“Percentage Closer” filtering
- Normal texture filtering just averages color components
- Averaging depth values does NOT work
- Solution [Reeves, SIGGARPH 87]
  - Hardware performs comparison for each sample
  - Then, averages results of comparisons
- Provides anti-aliasing at shadow map edges
- Not soft shadows in the umbra/penumbra sense

Notice how specular highlights never appear in shadows.
Notice how curved surfaces cast shadows on each other.

Green is where the light planar distance and the light depth map are approximately equal.
Non-green is where shadows should be.
Hardware Shadow Map Filtering

- GL_NEAREST: blocky
- GL_LINEAR: antialiased edges

Low shadow map resolution used to heighten filtering artifacts

Problems with shadow maps

- Hard shadows (point lights only)
- Quality depends on shadow map resolution (general problem with image-based techniques)
- Involves equality comparison of floating point depth values means issues of scale, bias, tolerance

Reflection Maps

Blinn and Newell, 1976

Environment Maps

Environment Maps

- Miller and Hoffman, 1984

Interface, Chou and Williams (ca. 1985)

Environment Maps

Cylindrical Panoramas

- 180 degree fisheye
- Photo by R. Packo

Cubical Environment Map
Reflectance Maps

- Reflectance Maps (Index by N)
  - Horn, 1977
- Irradiance (N) and Phong (R) Reflection Maps
  - Miller and Hoffman, 1984

Irradiance Environment Maps

Diffuse Reflection

Assumptions

- Diffuse surfaces
- Distant illumination
- No shadowing, interreflection

Hence, Irradiance a function of surface normal

Analytic Irradiance Formula

Ramamoorthi and Hanrahan 01
Basel and Jacobs 01

9 Parameter Approximation

RMS error = 25 %
9 Parameter Approximation

Exact image

Order 1
4 terms

RMS Error = 8%

Y_{lm} (\theta, \varphi)

Order 2
9 terms

RMS Error = 1%

For any illumination, average error < 3% [Basri Jacobs 01]

Real-Time Rendering

\[ E(n) = n^T M n \]

Simple procedural rendering method (no textures)
- Requires only matrix-vector multiply and dot-product
- In software or NVIDIA vertex programming hardware

Widely used in Games (AMFPED for Microsoft Xbox), Movies (Pixar, Framestore CFC, ...)

```c
surface float1 irradmat (matrix4 M, float3 v) {
    float4 n = {v , 1} ;
    return dot(n , M*n) ;
}
```

Environment Map Summary

- Very popular for interactive rendering
- Extensions handle complex materials
- Shadows with precomputed transfer
- But cannot directly combine with shadow maps
- Limited to distant lighting assumption

Resources

- OpenGL red book (latest includes GLSL)
- Older books: OpenGL Shading Language book (Rost), The Cg Tutorial, ...
- [http://www.realtimerendering.com](http://www.realtimerendering.com)
  - Real-Time Rendering by Moller and Haines
  - Links to Miller and Hoffman original, Haeberli/Segal
- [http://www.cs.ucsd.edu/~ravir/papers/envmap](http://www.cs.ucsd.edu/~ravir/papers/envmap)
  - Also papers by Heidrich, Cabral, ...
- Lots of information available on web...
- Look at resources from CSE 274 website (Wi, Fa 15)