Advanced Deferred Shading

Efficient and Scalable Shading for Many Lights

Outline…

1. GPU Overview
2. Shading recap
3. Forward Shading
4. Deferred Shading
5. Tiled Deferred Shading
6. And more!

The Graphics Processing Unit
A Brief History

- NVIDIA GeForce 256, 1999
  - Fixed function
- Programmable Shaders, 2001-2003
  - Increasing Programmability
- Unified Shaders – AMD Xbox360, 2005
  - General Programmable Cores
- CUDA, 2007
  - C/C++, Integers, Memory Access
- Open CL, 2009

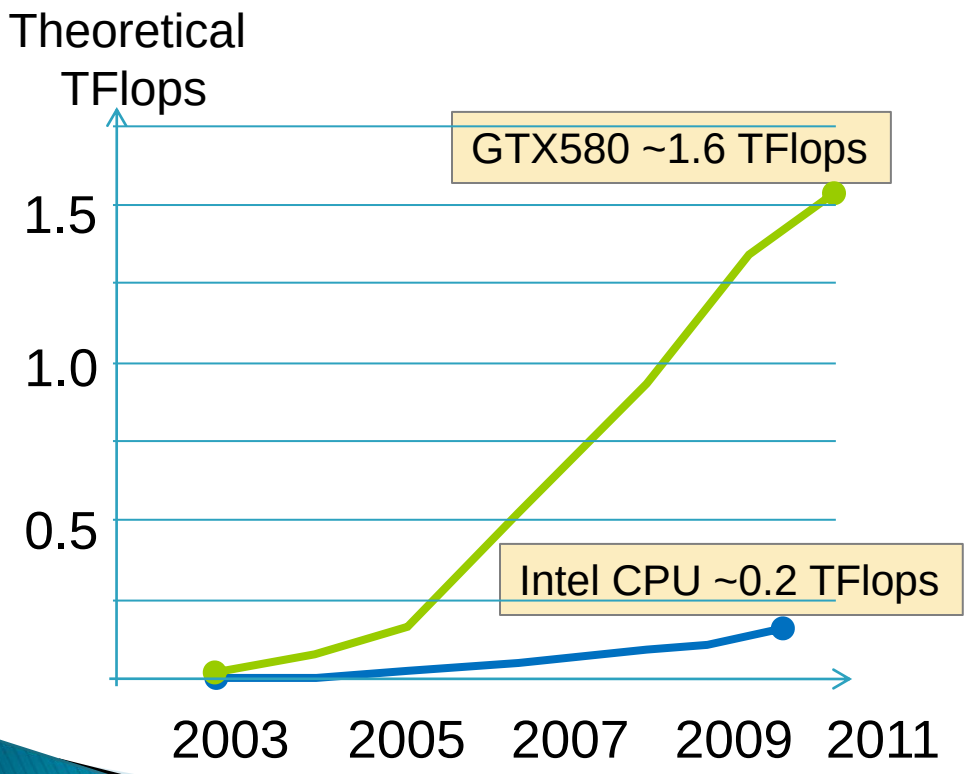
The GPU: Now and Future

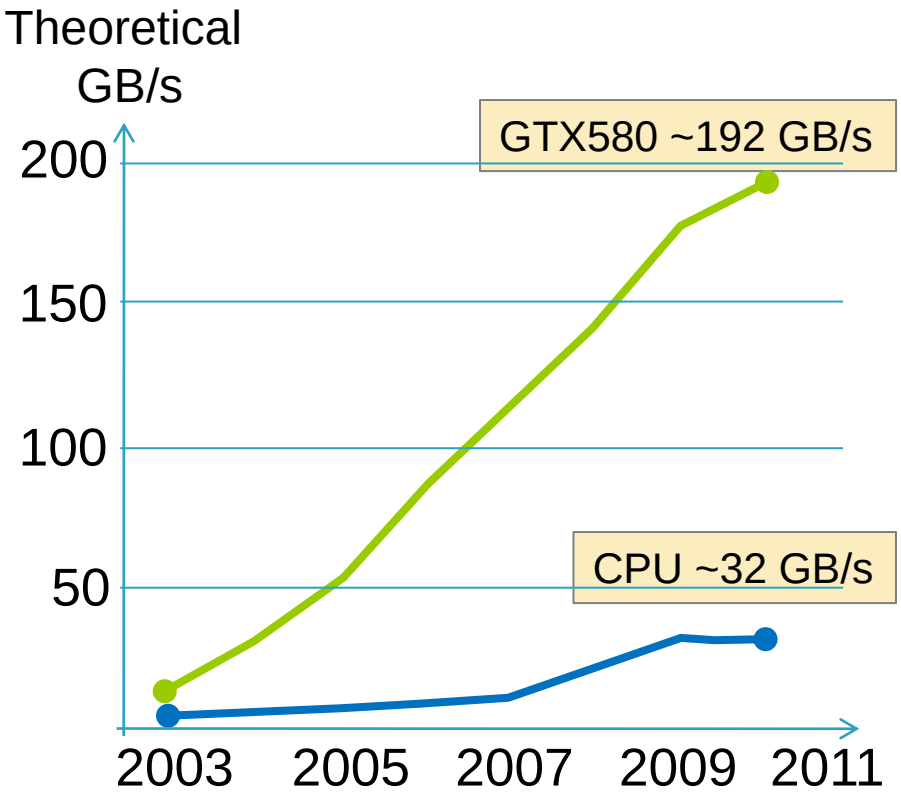
- NVIDIA Geforce 400 series
  - Fermi, 2010
  - Virtual Functions
  - Unified Memory Addressing
- AMD Radeon 7900 series
  - Graphics Core Next, 2012
  - Much the same actually.
- CPU/GPU integration
  - AMD Fusion APU
  - Intel Sandy Bridge

GPU Performance

- Around 8x flops
- Growing fast!

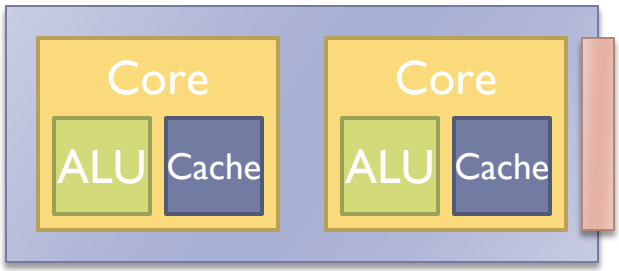
- Around 6x bandwidth
- Not growing fast enough!





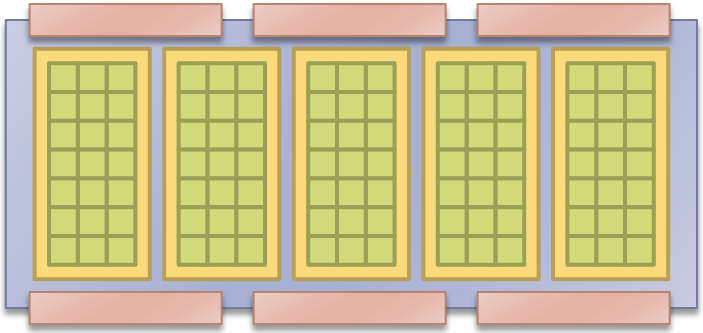
GPU Architecture –How do they do it?

CPU



- Few complex tasks
  - 1–2 thread/core
- Clever instruction exec
  - Branch prediction etc
- Lots of cache
- Narrow Memory Bus

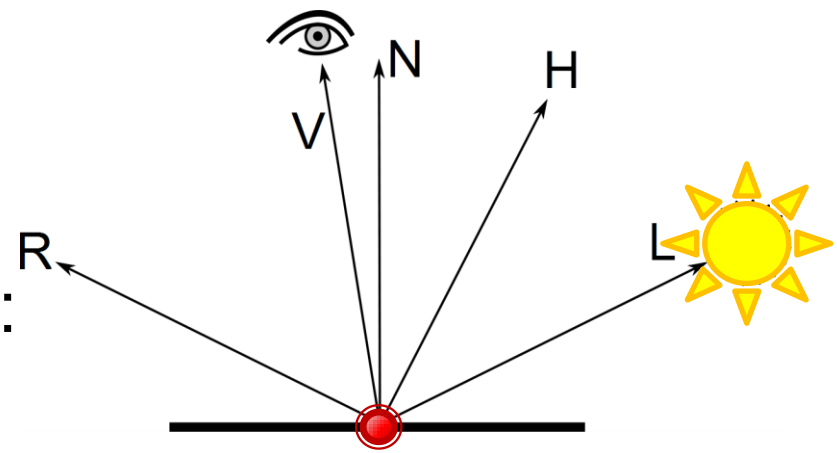
GPU



- Many simple tasks
  - ~48 threads/core
  - Wide SIMD (Many ALUs)
- Task switching
  - Latency hiding
- Wide Memory Bus

Shading

- Color and intensity at sample point
  - Here: sample point = fragment
- Diffuse contribution needs:
  - Light direction
  - Normal
  - Material diffuse color
- Specular shading needs:
  - Light direction
  - Normal
  - Eye direction
  - Material shininess
  - Material specular color



Fragment Shader Example (1/2)

- Only One Light!
  - Position
  - Color
- Material
  - Reflectances
  - Untextured
- Vertex data
  - Interpolated
- Result color
  - Output to frame buffer

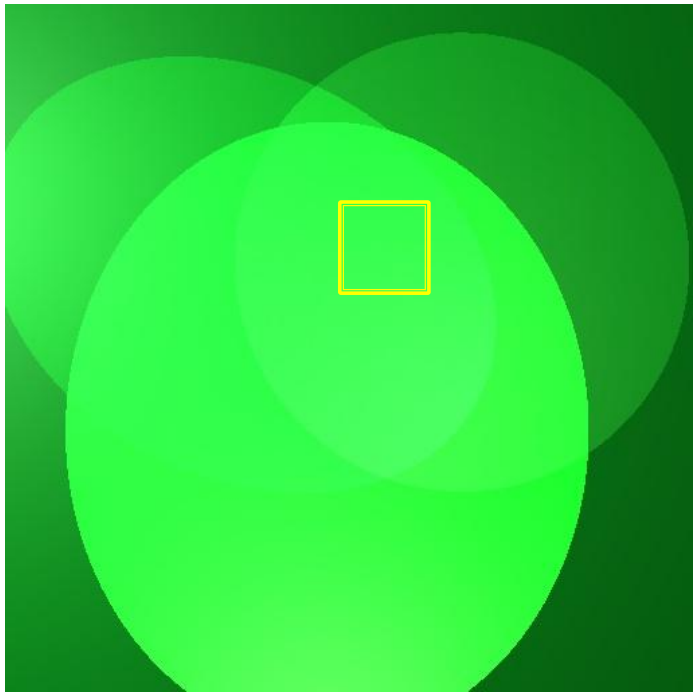
```
uniform vec3 lightPos;
uniform vec3 lightColor;

uniform vec3 materialDiffuse;
uniform vec3 materialSpecular;
uniform float materialShininess;

in vec3 samplePos;
in vec3 sampleNormal;

out vec4 resultColor;

void main()
{
  // see next slide...
}
```

Fragment Shader Example (2/2)

```
out vec4 resultColor;

void main()
{
  vec3 lightDir = normalize(lightPos - samplePos);
  vec3 diffuse = dot(sampleNormal, lightDir)
                   * materialDiffuse * lightColor;
  vec3 viewDir = normalize(-samplePos);
  vec3 h = normalize(lightDir + viewDir);
  vec3 specular = pow(dot(h, normal), materialShininess)
                   * materialSpecular * lightColor;
  resultColor = vec4(diffuse + specular, 1.0);
}
```

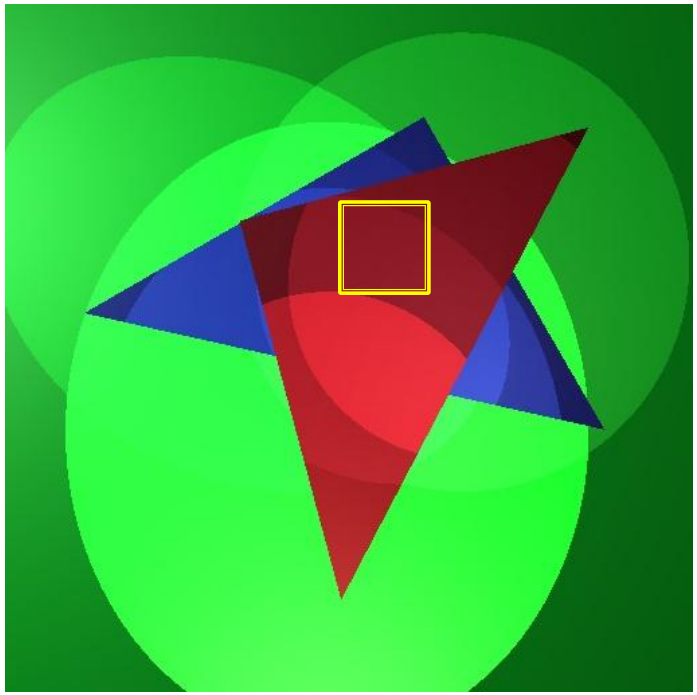
Forward Shading

- The traditional approach
  - Still most common for real time?
    - Shifting now…
  - Single pass
  - Shading in fragment shaders
- Geometry generates fragments
  - Each fragment is shaded
  - Result is merged into frame buffer
    - E.g. overwrite if nearest.



Forward Shading – Problems

- Overdraw
  - New fragments replaces old
  - Shading re-computed
  - And again…
  - Lots of waste
    - Especially for many lights.
- Light Assignment
  - What lights to use?
  - Usually per 'object'
    - E.g. A few thousand triangles
- Shader management
  - Must support varying numbers of lights

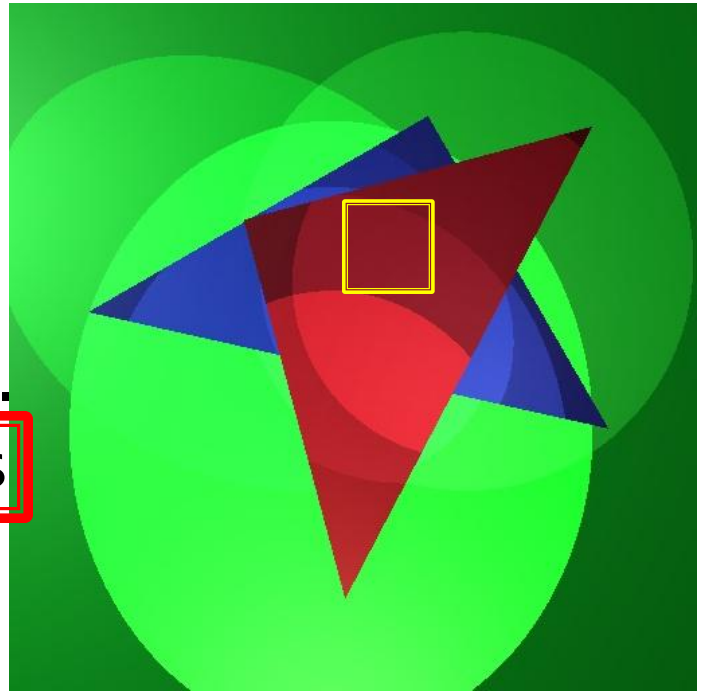


Overdraw

- Brighter = more overdraw
  - Lots of waste

Light Assignment

- Which lights to use?
  - Lights vs geometry
  - Complex problem
    - 1000s of objects/batches.
    - 1000s of lights
- Want minimal set of lights.
  - (Geometrically) Small batches
- Want quick drawing
  - Large batches!
  - Good batching for GPU/API.
  - Few material changes



Summary: Forward Shading

The good

- Single Pass
- Low storage overhead
  - Single Frame Buffer
- Simple if only single / few lights
- MSAA works

The Bad

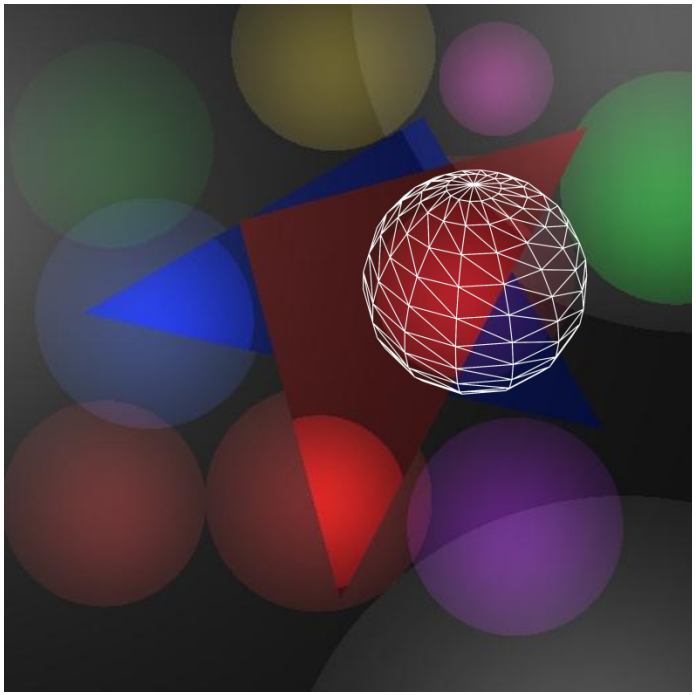
- Overdraw
- Light management
  - Difficult for many lights
- Shader management
- Batching coupled with lighting

Deferred Shading

- Key idea
  - Defer (i.e. wait with) light computations
- First do a geometry pass
  - Sample geometry as per normal
  - But, no lighting
  - Store geometry attributes per pixel
- Then do a shading pass
  - Use attributes from first pass.
  - Process lights one at a time.

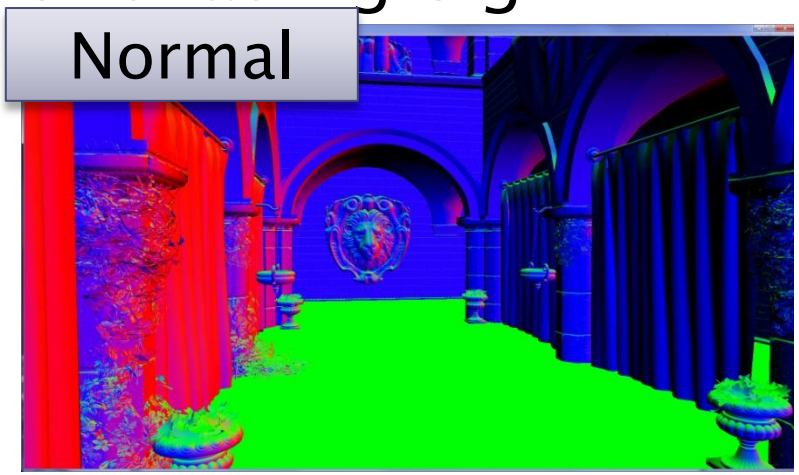
Deferred Shading

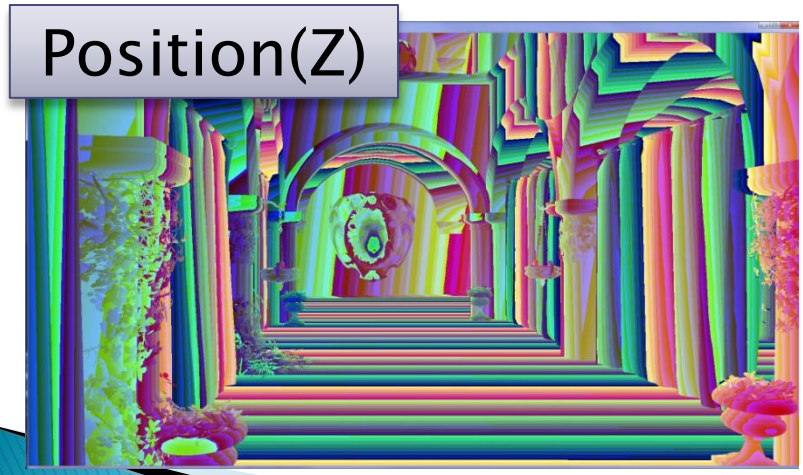
- Render Geometry to G-Buffers
  - G-Buffers store attributes,
  - e.g. color, normal, position.
- For each light
  - Draw Light Bounds
    - For each fragment
      - Read G-Buffers
      - Compute Shading
      - Add to frame buffer
- No overdraw
- Trivial light management

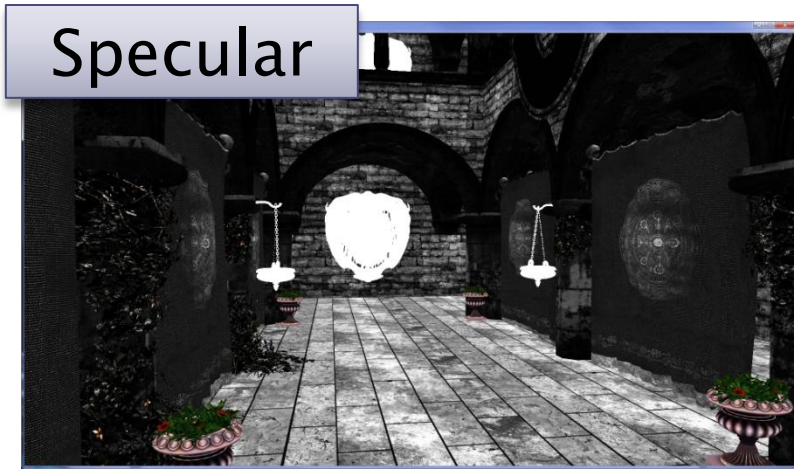


Render G-Buffers

- All attributes needed for shading e.g







G-Buffer Fragment Shader

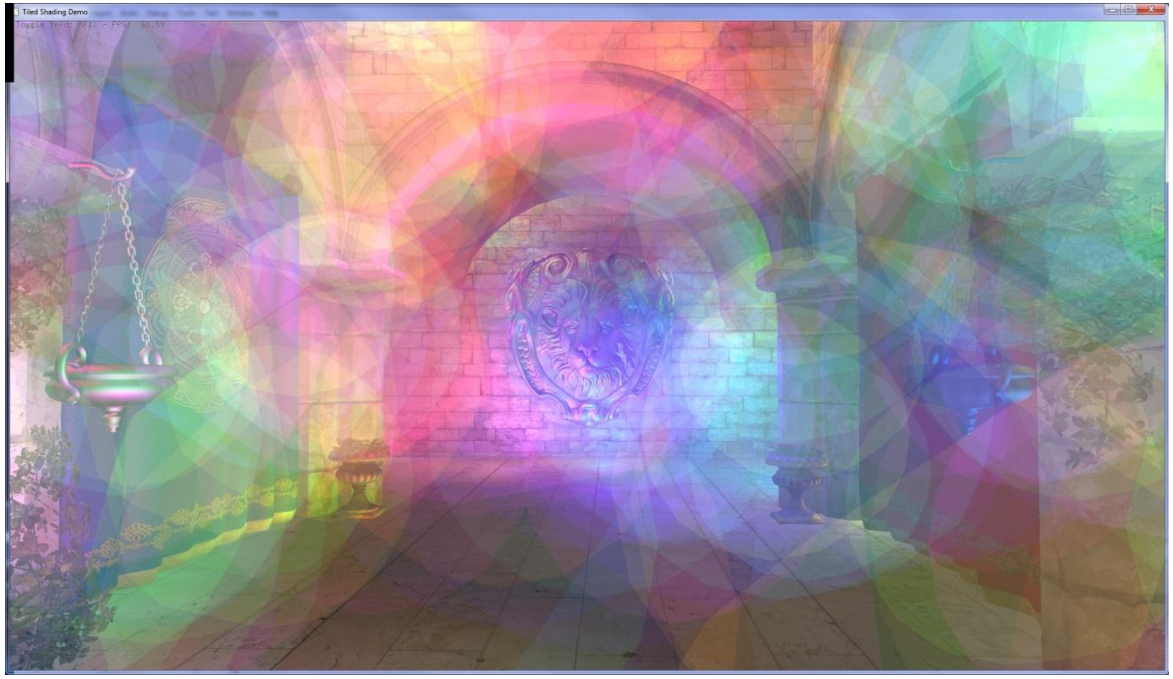
- No Lights!
  - Just the geometry
- More Outputs
  - to G-buffer
  - 4 Screen sized textures
- Position not stored
  - Use depth buffer.
  - Reconstruct using inverse projection

```
uniform vec3 materialDiffuse;
uniform vec3 materialSpecular;
uniform float materialShininess;

in vec3 samplePos;
in vec3 sampleNormal;

out vec3 gBufferNormal;
out vec3 gBufferDiffuse;
out vec3 gBufferSpecular;
out float gBufferShininess;

void main()
{
  gBufferNormal = normal;
  gBufferDiffuse = materialDiffuse;
  gBufferSpecular =
            materialSpecular;
  gBufferShininess =
            materialShininess);
}
```

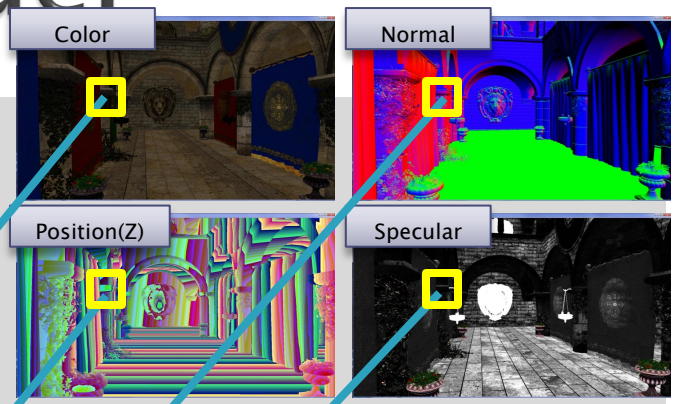
Draw Light Bounds

- For each light
  - Draw Light Bounds
    - For each fragment

- Read G-Buffers
- Compute Shading
- Add to frame buffer



Deferred Light Shader




```
uniform vec3 lightPosition;
uniform vec3 lightColor;
uniform float lightRange;

void main()
{
  vec3 color = texelFetch(colorTex, gl_FragCoord.xy);
  vec3 specular= texelFetch(specularTex, gl_FragCoord.xy);
  vec3 normal = texelFetch(normalTex, gl_FragCoord.xy);
  vec3 position = fetchPosition(gl_FragCoord.xy);

  vec3 shading = doLight(position, normal, color,
                         specular, lightPosition,
                         lightColor, lightRange);

  resultColor = vec4(shading, 1.0);
}
```

Summary: Deferred Shading

The good	The Bad
Enables many lights Trivial light management Simple shader management No overdraw	Difficult to support transparency Large frame buffer ◦ Especially with MSAA High memory bandwidth usage

Tiled Deferred Shading

- Key goal
  - Solve bandwidth issue
- Read G-Buffers once

The Bandwidth problem (1/3)

- New type of overdraw
  - Light overdraw
- N lights cover a certain pixel ->
  - N reads from the **same** G-Buffer location
- Deferred Shading

```
for each light
  for each covered pixel
    read G-Buffer
    compute shading
    write frame buffer
```

The Bandwidth problem (2/3)

- Deferred Shading, Light Loop

```
for each light
  for each covered pixel
    read G-Buffer
    compute shading
    write frame buffer
```

- We would like to re-arrange

```
for each pixel
 read G-Buffer
 for each affecting light
   compute shading
 write frame buffer
```

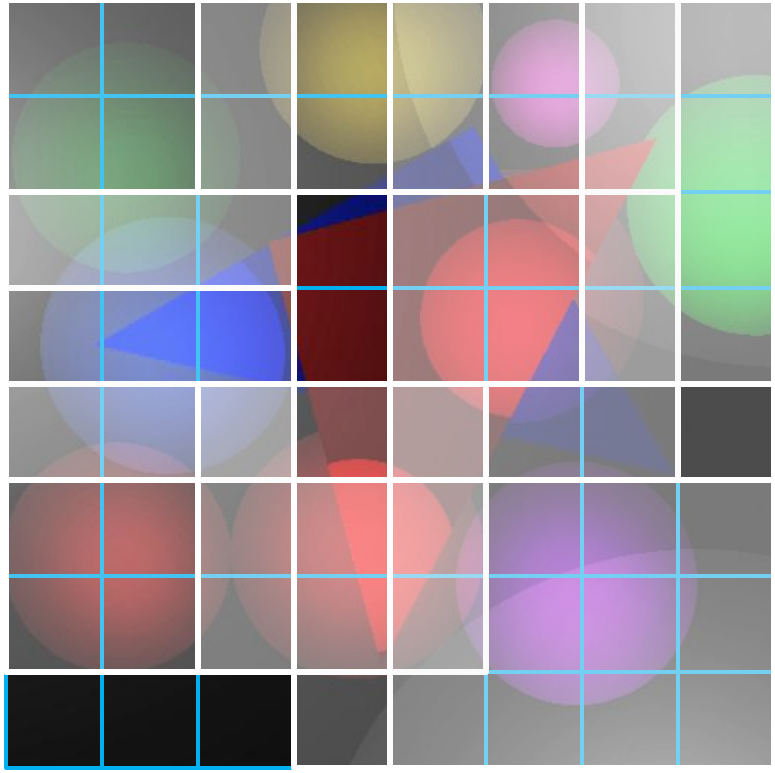
The Bandwidth problem (3/3)

- Requires
  - sequential access to lights for each pixel
- List per pixel prohibitive
  - Lots of data
  - Slow construction
- Share list in screen space tiles
  - E.g. 32 x 32 pixels
  - Simple construction
  - Little storage
  - Coherent access

Tiled Deferred Shading

- The Light Grid

1	1	2	1	2	2	3	2
1	1	2	1	2	2	3	2
2	2	2		2	2	3	2
1	1	1		2	2	3	2
2	2	3	1	3	2	2	1
1	1	2	2	3	2	2	2
1	1	2	2	3	2	2	2
			1	2	2	2	2



Demo Time

- Lets look at those tiles...

Tiled Deferred Shading

- 1. Render Scene to G-Buffers
- 2. Build Light Grid
- 3. Full Screen Quad (or CUDA, or Compute Shaders)
  - For each pixel
    - Fetch G-Buffer Data
    - Find Tile
    - Loop over lights and accumulate shading
    - Write shading

Deferred Light Computation

```
vec3 computeLight(vec3 position, vec3 normal, vec3 albedo,
                  vec3 specular, float shininess)
{
  vec3 viewDir = -normalize(position);

  ivec2 l = ivec2(gl_FragCoord.xy) / CELL_DIM);

  int count = lightGrid[l.x][l.y].x;
  int offset = lightGrid[l.x][l.y].y;

  vec3 shading = vec3(0.0, 0.0, 0.0);

  for (int i = 0; i < count; ++i)
  {
    int lightId = texelFetch(tileDataTex, offset + i);
    shading += doLight(position, normal, albedo, specular,
                       shininess, viewDir, lightId);
  }
  return shading;
}
```

Deferred Light Computation

```
void main()
{
  ivec2 fragPos = ivec2(gl_FragCoord.xy);
  vec3 albedo = texelFetch(albedoTex, fragPos).xyz;
  vec4 specShine =
            texelFetch(specularShininessTex, fragPos);
  vec3 position = unProject(gl_FragCoord.xy,
                    texelFetch(depthTex, fragPos));
  vec3 normal = texelFetch(normalTex, fragPos).xyz;

  vec3 viewDir = -normalize(position);

  gl_fragColor = computeLight(position, normal, albedo,
                              specShine.xyz, viewDir,
                              specShine.w, fragPos)
}
```

Summary: Tiled Deferred Shading

The good

- Low bandwidth usage
- Enables many lights
- Trivial light management
- Simple shader management
- No overdraw

The Bad

- Difficult to support transparency
- Large frame buffer
  - Especially with MSAA

Tiled Forward Shading

- Key Idea
  - Use tiles to solve light assignment
  - For forward shading
- Forward as usual
  - Shading in fragment shader
  - Access grid for list of lights
- Benefits
  - No G-Buffer
  - Transparency
- Can easily combine!
  - Tiled Deferred for opaque geometry
  - Tiled Forward for transparent

Tiled Forward Shading

- 1. Build Light Grid
- 2. Render Scene
  - For each fragment
    - Use geometry attributes
    - Find Tile
    - Loop over lights and accumulate shading
    - Write shading
- Notes:
  - + FSAA
  - + Transparency
  - + No G-Buffers
  - - Overdraw

Forward Light Computation

```
in vec3 normal;
in vec3 position;
in vec2 texCoord;
uniform vec3 specular;
uniform float shininess;
uniform float diffuse;

void main()
{
  ivec2 fragPos = ivec2(gl_FragCoord.xy);
  vec3 albedo = texture2D(albedoTex, texCoord) * diffuse;
  vec3 viewDir = -normalize(position);

  gl_fragColor = computeLight(position, normal, albedo,
                              specular, viewDir, shininess,
                              fragPos);
}
```

Summary: Tiled Forward Shading

The good

- Low bandwidth usage
- Enables many lights
- Trivial light management
- Simpler shader management
- MSAA
- Transparency
- No G-Buffers

The Bad

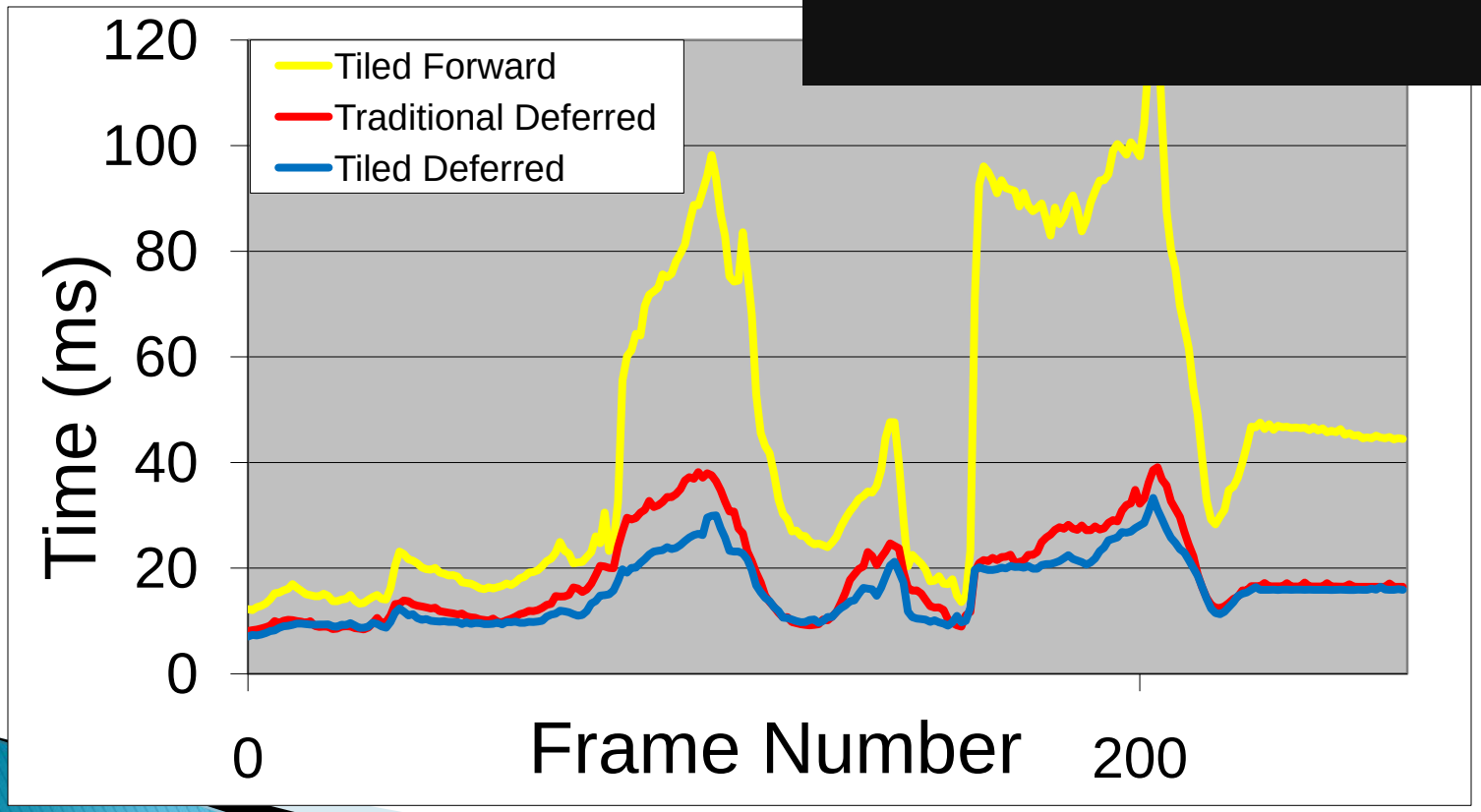
- Overdraw is back
- Coherency worse
  - Looong shaders

Technique Comparison

	Deferred	Tiled Deferred	Tiled Forward
Innermost loop	Pixels	Lights	Lights
Light data access pattern	Sequential	Random	Random
Pixel data access pattern	Random	Sequential	Sequential
Re-use Shadow Maps	Yes	No	No
Shading Pass	Deferred	Deferred	Geometry
G-Buffers	Yes	Yes	No
Overdraw of shading	No	No	Yes
Transparency	Difficult	Simple[a]	Simple
Supporting FSAA	Difficult	Difficult	Trivial
Bandwidth Usage	High	Low	Low
Light volume intersection	Per Pixel	Per Tile	Per Tile

Performance Comparison

- Frame times, GTX 280



Performance: Scaling

GPU	GTX280		GTX480	
G-Buffer Depth	16-bit	32-bit	16-bit	32-bit
TiledDeferred	15.7	17.2	7.87	10.4
min / max	7.05 / 33.3	8.82 / 34.8	4.19 / 15.9	6.91 / 18.2
TiledFwd	148		43.0	
min / max	26.5 / 410		11.0 / 107	
TiledFwd-PreZ	45.5		30.1	
min / max	12.1 / 125		9.31 / 72.4	
Deferred	38.3	82.1	26.8	51.3
min / max	14.0 / 90.0	27.7 / 197	9.36 / 64.3	18.2 / 121
DeferredStencil	18.4	28.3	12.2	20.2
min / max	8.1 / 39.1	12.4 / 64.8	4.98 / 26.6	9.55 / 43.8

Other Variants

- Deferred Lighting
  - Factor out specular and diffuse color
  - G-Buffer only store normal and shininess
  - Output diffuse and specular shading
  - Second geometry pass which multiplies colors
- Light PrePass
  - Much the same
  - But with monochromatic specular highlight
- Similar performance as deferred
  - Only improves constant factors.
- Limits shading model even further

Optimizations

- Exclude Small Lights (on screen)
  - E.g. small = radius < a tile
  - Use standard deferred

Further Reading

- *Tiled Shading* – Olsson and Assarsson, JGT 2011
- *Deferred Rendering for Current and Future Rendering Pipelines* – Lauritzen, SIGGRAPH 2010 Course.
- *Bending the Graphics Pipeline*, Andersson, SIGGRAPH 2010 Course
- *Improved Tiled Shading* – Olsson and Billeter Somewhere 2012 ;)

Cheers

- You may now ask them questions.