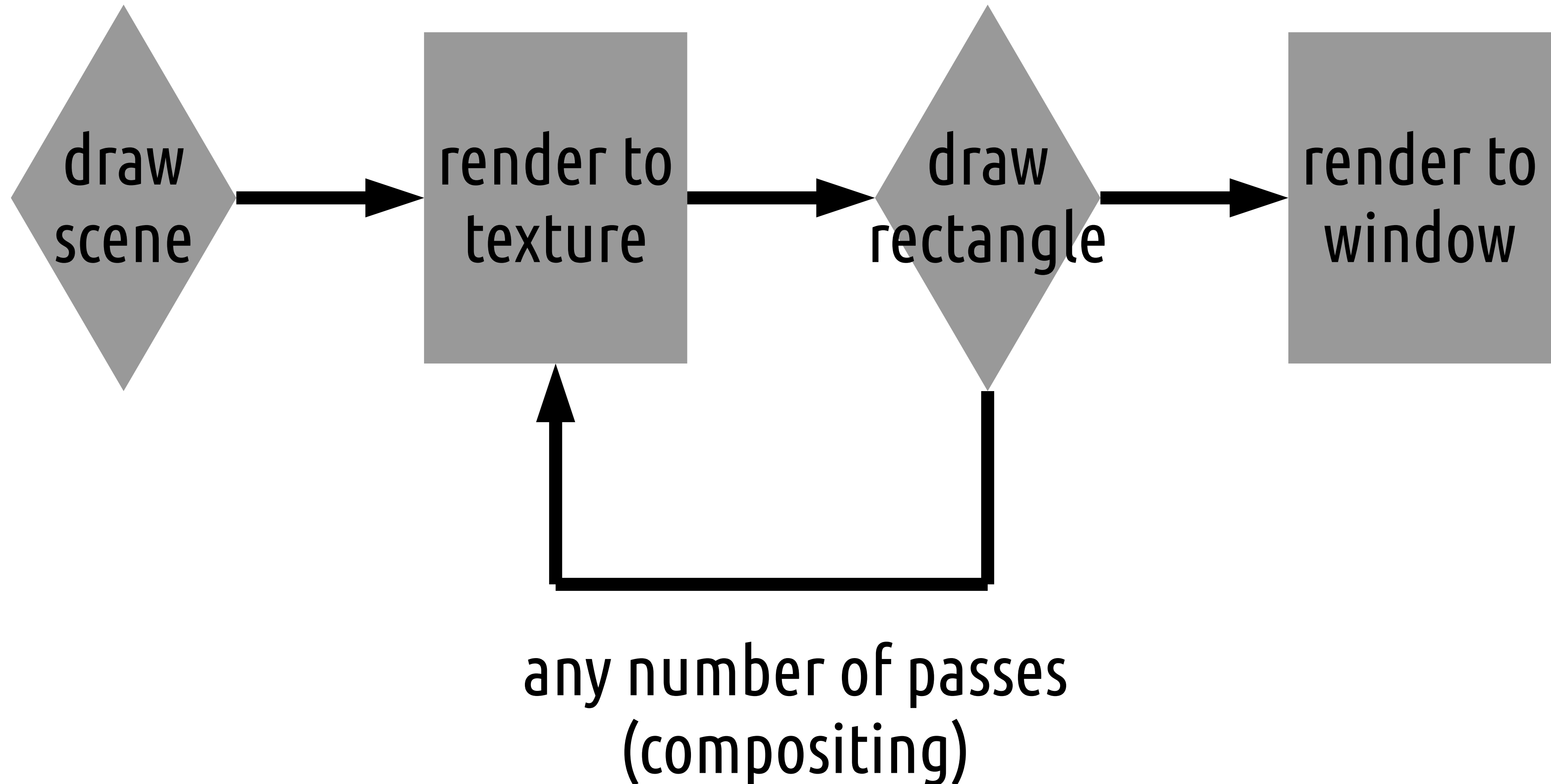
Multi-Pass

Anton Gerdelan gerdela@scss.tcd.ie
Trinity College Dublin

Fragment Shader Limitations

- Painful to add a global effect that modifies ALL shader outputs
  - Gamma correction
  - Whole-screen flashes / palette change
- Would be nice to access neighbouring fragments
  - Blur
  - Edge-detection
  - More advanced effects

Multi-Pass Rendering Concept



How to render to texture in GL

- Default **framebuffer** renders to the window
- Create
  - additional framebuffer
  - texture for colour buffer (RGBA == out vec4 frag_colour)
  - texture for depth buffer (special depth format for 24-bit depth buffer)
- Attach textures to new framebuffer
- **Bind new framebuffer**
- **Draw normal scene**
- **Bind back to default**
- You have now rendered to a texture

C Code

- Basic setup demo and simple ex. (#34) https://github.com/capnramses/antons_opengl_tutorials_book
- I walk through these in my book (later chapters)
- Just more tedious GL state machine code (similarish to VAO)
  - `bool init_fb ()` on main.cpp: line 34
  - Check out how I swap the framebuffers in the `while()` loop

Shaders that draw to a texture

- No changes
- Remember the fragment shader's output?
  - `out vec4 frag_colour;`
- When we set up the new framebuffer, we re-direct this (colour output 0) to our new attached colour texture
- Shaders write to a depth-buffer automatically
- We redirected this to use our attached depth texture

Shaders – Full-Screen Quad

- Could create a new VAO with 2 triangles that cover the screen, and draw that
- Or…

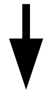
Attribute-less Rendering

```
#version 420

vec2 data[4] = vec2[]
(
	vec2 (-1.0,  1.0),
	vec2 (-1.0, -1.0),
	vec2 ( 1.0,  1.0),
	vec2 ( 1.0, -1.0)
);

out vec2 st;

void main () {
	gl_Position = vec4 (data[gl_VertexID], 0.0, 1.0);
	st.s = gl_Position.x * 0.5 + 0.5;
	st.t = 1.0 - (gl_Position.y * 0.5 + 0.5);
}
```

Note: horrible array format in GLSL. Stick to this or your code only works on **some** machines.

Attribute-less Rendering

- Vertices hard-coded in array in shader -1 to 1
- Built-in **`gl_VertexID`** integer to index
- Texture coordinates inferred from vertices
- No need to create or bind a VAO, buffers, etc.
- BTW This is your second rendering pass



```
glBindFramebuffer (GL_FRAMEBUFFER, 0);

glActiveTexture (GL_TEXTURE0);

glBindTexture (
  GL_TEXTURE_2D, our_colour_output_tex);

glDrawArrays (GL_TRIANGLE_STRIP, 0, 4);
```

Draw to main display this time

Sample texture that we drew to in first pass

Fragment Shader

- Sample the texture
- frag_colour = texel → draws the original scene
- **Q. how can we make a grey-scale version of the original?**
- **PS remind me I want to do this now**

Palette Effects, Basic Filters

"Beserk" mode in Doom (Id Software, 1993)

- Greyscale/sepia filter with weights
- Flashes over the whole screen
- "Palette" effects (swap certain colours)
- Gamma correction (just make the attached colour image sRGB format)

Kernel-Based Algorithms

- Computer vision and image processing algorithms
- Sliding window (“kernel”) of neighbouring pixels
  - Blurs
  - Bloom
  - Edge detection

1	2	1
2	0	2
1	2	1

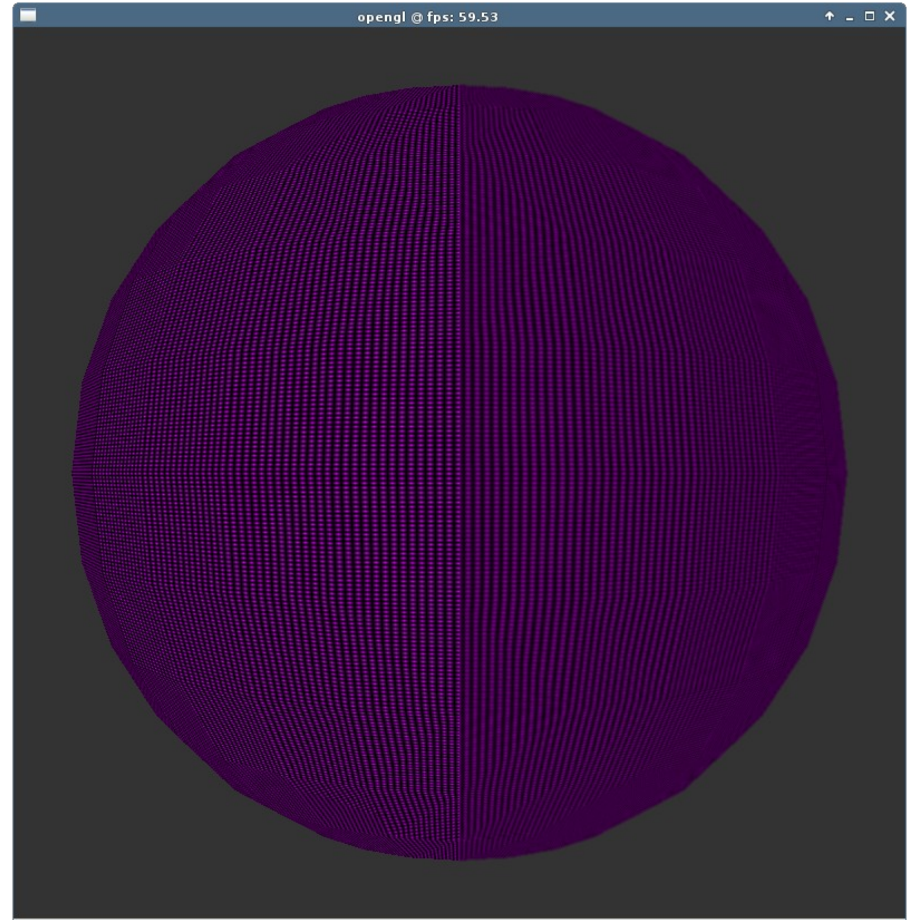
- **Q. How can we work out the texture coordinates of neighbouring pixels?**
- **Q. Are there any invalid neighbours?**

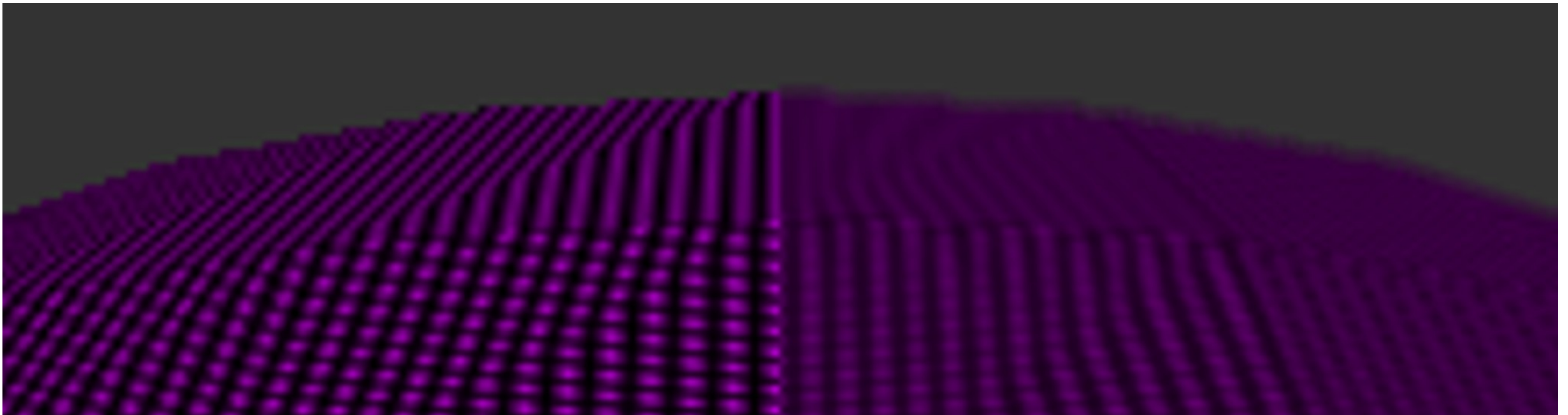
sample neighbouring texels

- Texcoords are 0.0...1.0 but pixels are 0...1024 etc.
- 1 pixel distance in texture coordinates is 1.0 / width or 1.0 / height
- Use known width or height or send as uniforms
- If useful `gl_FragCoord` will give you the current fragment's pixel coords i.e. in range of 0...1024
- **Q. now how can we do a blur?**

3x3 kernel averaging blur

1	2	1
2	0	2
1	2	1





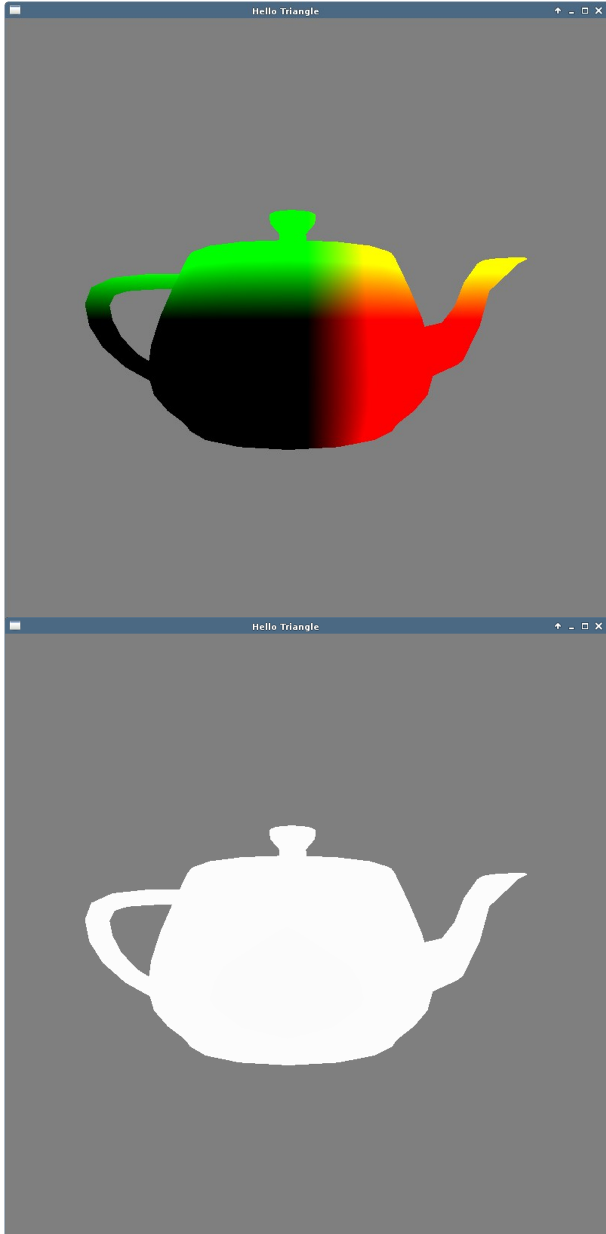
Compositing

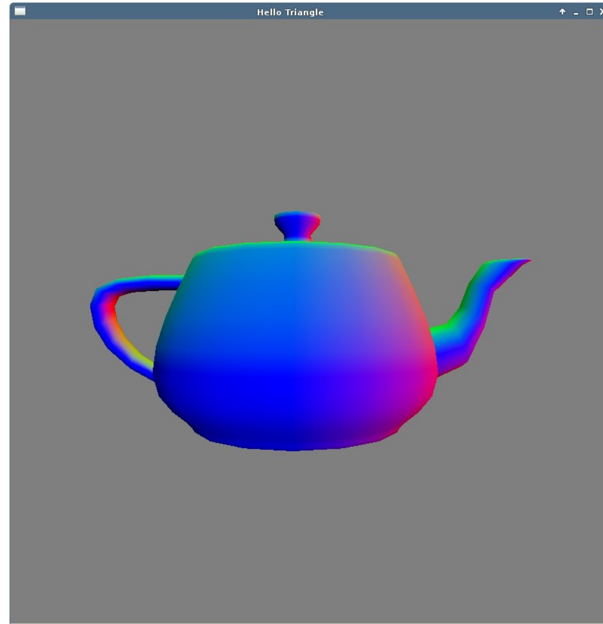
- When drawing your second pass, instead of binding the default framebuffer
- Bind yet another framebuffer with new textures
- Easy to chain together many post-processing passes

Deferred Rendering

- Textures can hold **any data**, not just visual images
- Can output **more than one** colour and texture from frag shader
- Exploit this
- Put other per-pixel variables in the texture
  - Normals
  - Positions
  - Material properties
- This texture collection is called a **geometry buffer** (G-Buffer)
- Can then **defer** calculations to screen-space

G-Buffer for Deferred Lighting

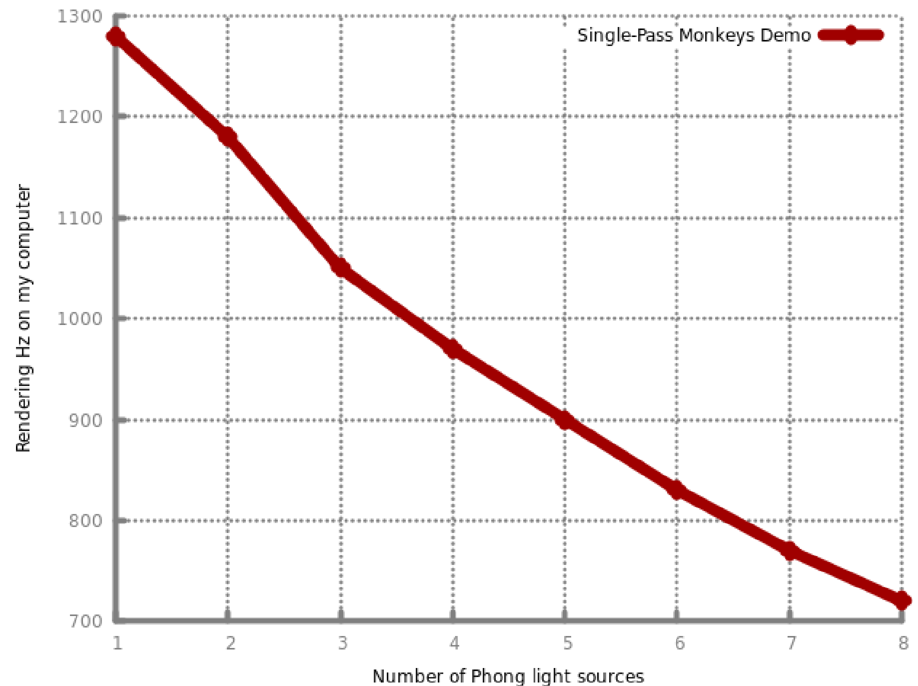




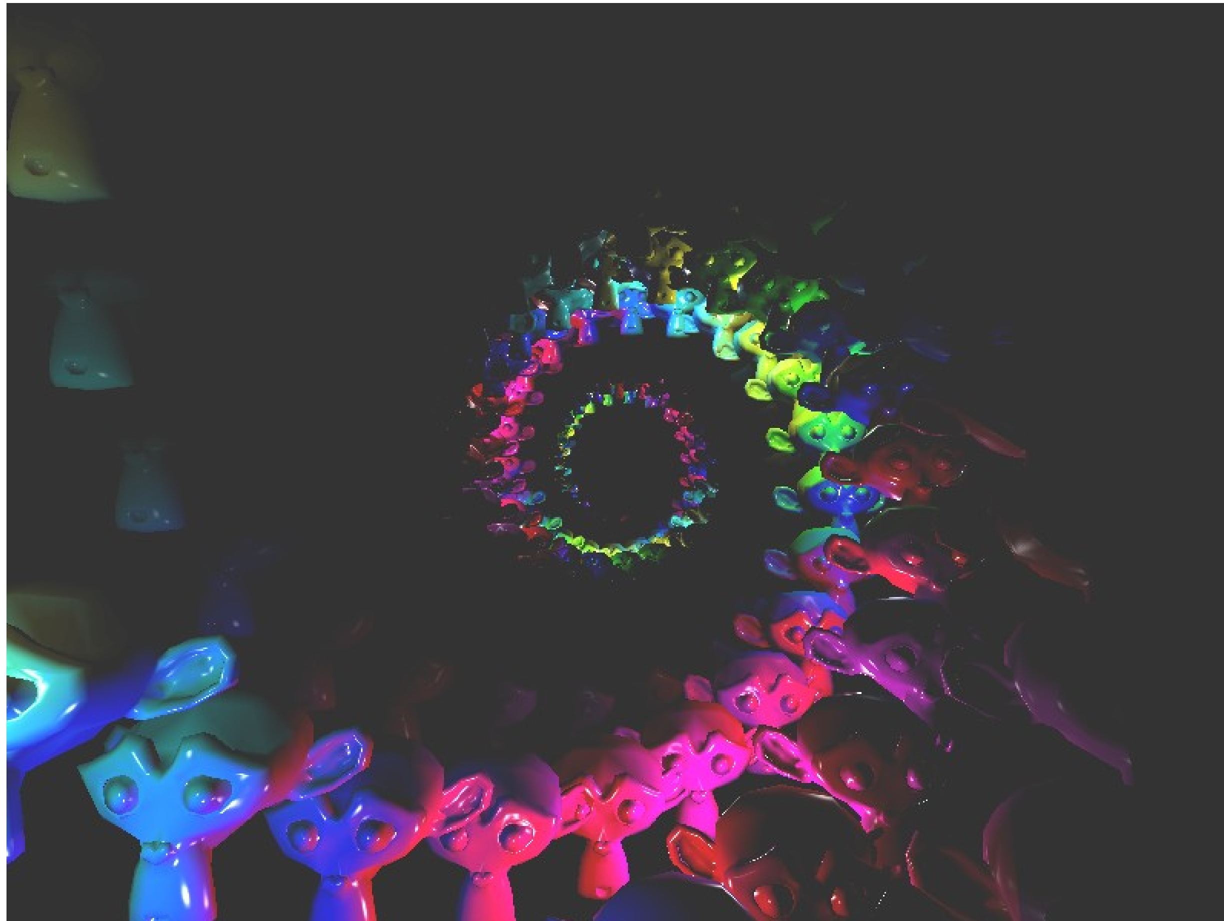
- Each variable that we need for diffuse lighting
  - Position
  - Normal
  - Diffuse reflection coefficient
  - And z-buffer (depth map) for sorting
- Can now do lighting in second pass .frag
- **Q. Advantage?**

Advantages

- Eliminate **overdraw** of fragments (not that useful in reality because **early-Z** rejection)
- Lights to use are specified per-fragment instead of per-mesh
- Managing lights to use in the 3d scene based on 2d screen coverage
- Can use more lights in view overall
- Beware! 2nd pass, lots of sampling, big G-buffer memory = slow



Deferred Lighting



- 64 lights in a scene
- Trick is to limit range of lights to reduce overlap

Other Post-Processing Techniques

- Tile-based rendering
  - split view into 16x16px viewports
- Screen-space ambient occlusion (SSAO)
  - G-buffer normals and depths to check for enclosed bits
- Stylised rendering
  - Cartoon, oil painting, other NPR, edges
  - Bloom (blur the light bits)
  - High dynamic range (HDR) – render bigger size, and downsize
  - Motion blur, depth-of-field blur (cinematic camera-like)

Disadvantages to Multi-Pass Rendering

- 2nd pass has high overhead cost
- Sampling very large textures is expensive, and kernels require >1 sampling per pixel
- You lose built-in anti-aliasing of polygon edges
- Some techniques are harder to implement in s.space
- Memory cost of G-Buffer. Reducing size is involved.

References and Reading

- OpenGL Superbible – deferring shading example with memory optimisations
- OpenGL Insights, chap. "*Performance Tuning for Tile-Based Architectures*" - Bruce Merry
- Any image processing or computer vision source – ideas for kernel and colour filters