Multi-pass Rendering

CSGV3 – Real-time Rendering
An example of a high quality image generated at 60 Hz using multi-pass rendering techniques

Multi-pass Rendering

• Render the same scene multiple times and combine the results
• This is the predominant approach for realistic imagery in real-time applications
• Many advanced rendering techniques fall into this category
  • E.g. shadow, light map, etc.
Multi-pass Rendering

- E.g. Quake III engine uses 10 passes
  - Accumulate bump map
  - Diffuse lighting
  - Base texture
  - Specular lighting
  - Emissive lighting
  - Volumetric/atmospheric effects
  - Screen flashes
Multipass Techniques

• Many cool effects are easier to achieve by rendering the scene multiple times:
  • Render the scene
  • Read the scene into a texture
  • Render that texture on the screen in some interesting way

• Main Benefits:
  • Speed and lower algorithm complexity:
    • Each pass is a specialised simplification of the rendering problem
    • Combine passes for apparent generality
Planar Reflections

• Simulate mirror reflection using multi-pass rendering
  • Reflect the eye about the plane of the mirror (OR reflect the scene)
  • Render a view from that point into texture
  • Render the scene as normal, placing that texture over the mirror
Sample Algorithm

• Render the scene excluding the mirror object as normal (stencil test off, depth test enabled)
  • Clear the stencil buffer and render the mirror polygon into the stencil and into Z buffer
    • Stencil buffer is set to draw only when depth test passes
    • Create a stencil mirror mask with visible mirror polygon tagged
  • Draw the mirror polygon into Z buffer using stencil mask so that visible mirror pixels are reset to maximum depth
• Render the reflected scene using stencil testing
  • Objects that are behind the reflector plane should not be reflected
Multi-texture

- Two or more textures are applied at once on an object
  - Hardware supported to render in a single pass
    - Fixed function or Fragment shader
  - Draw object once with several textures
Multi-texturing

• Some of which can be dynamically modified
  • Light map may be updated by a dynamic light and the fog map may by the moving camera.
  • The texture coordinate and vertex correspondence can be different for each texture map

\[ + \times = \text{Gloss Map} \]
Light Mapping

• Stores the result of the diffuse component computation on static lighting (view independent)
• By multiplying the map with the underlying surface’s material, brightness or global intensity decreases
• Basically per-pixel lighting
Light Mapping
Light Mapping
Dynamic Lightmaps
Light map example
Shadow Mapping

• E.g. Shadow buffers:
  • Render scene from light’s viewpoint
    • Store depth values
  • Render scene from eye
    • Compare depths with shadow buffer

• More on Shadows in next lecture
Dynamic Environment Mapping

• Generate environment map on the fly by rendering objects onto six faces of cube map

• Problems with multiple reflecting objects:
  • Cubemap for each centre OR approximate from centre of scene
  • Reflecting objects reflecting each other
Cube Maps

• Really good collection of real world cube maps at high res:
  • http://www.humus.name/index.php?page=Textures
Image Based Effects

Depth-of-field

Bloom

Motion Blur
Motion Blur

1. T = empty texture
2. Render a slightly dimmed T
3. Render the scene
4. Copy the framebuffer into T
5. Goto 2
Image Convolution

1. Render the scene
2. Save it as a texture
3. Render the texture, using a shader to convolve the image

• Convolution:

\[ H(x, y) = \sum_{j=0}^{\text{height}-1} \sum_{i=0}^{\text{width}-1} F(x + i, y + j) \cdot G(i, j) \]

• F is the base image
• G is the convolution kernel
Image Convolution

const int MaxKernelSize = 25;
uniform vec2 Offset[MaxKernelSize];
// size of kernel (width * height) for this execution

uniform int KernelSize; // final scaling value
uniform vec4 ScaleFactor; // image to be convolved
uniform sampler2D BaseImage;

void main(void)
{
    int i;
    vec4 sum = vec4 (0.0);

    for (i = 0; i < KernelSize; i++)
        sum += texture2D(BaseImage, gl_TexCoord[0].st + Offset[i]);

    gl_FragColor = sum * ScaleFactor;
}

<table>
<thead>
<tr>
<th>Edge Detection</th>
<th>Sharpening</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0</td>
<td>0 -1 0</td>
</tr>
<tr>
<td>1 -4 1</td>
<td>-1 4 -1</td>
</tr>
<tr>
<td>0 1 0</td>
<td>0 -1 0</td>
</tr>
</tbody>
</table>