
CSE 5542 - Real Time Rendering

Week 11, 12, 13

Texture Mapping

Courtesy: Ed Angel



DEPARTMENT OF
COMPUTER SCIENCE
AND ENGINEERING

Limits of Geometric Modeling

Millions of Polygons/Second



The image cannot be displayed. Your computer may not have enough memory to open the image, or the image may have been corrupted. Restart your computer, and then open the file again. If the red x still appears, you may have to delete the image and then insert it again.

Cannot Do



DEPARTMENT OF
COMPUTER SCIENCE
AND ENGINEERING

Use Textures

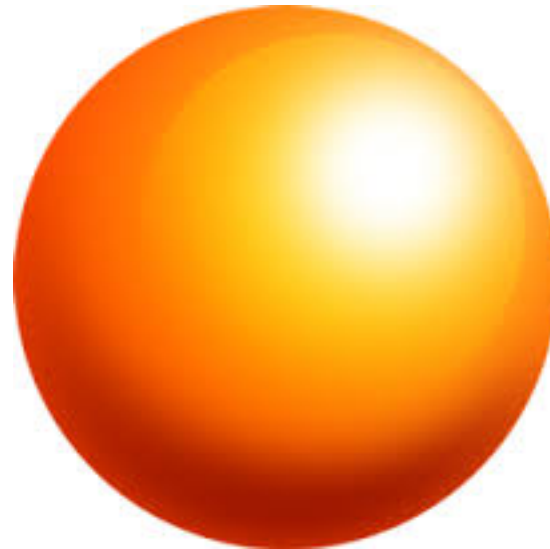


Orange



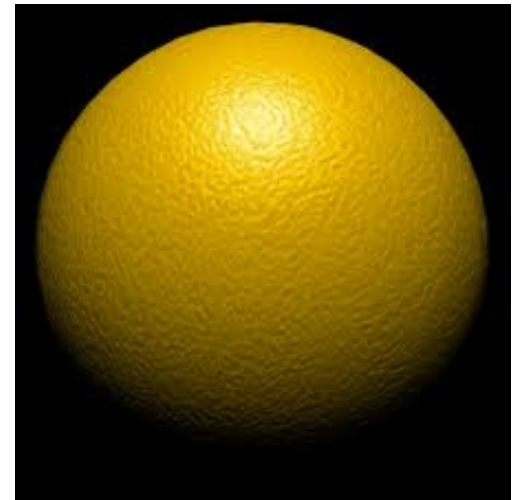
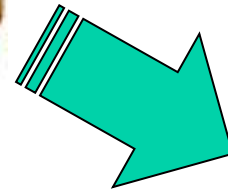
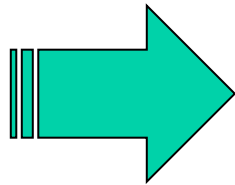
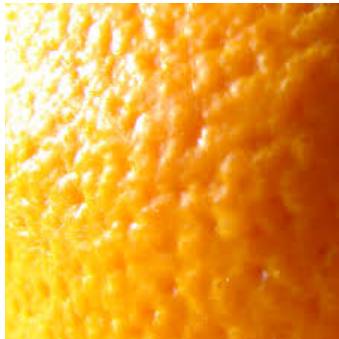
DEPARTMENT OF
COMPUTER SCIENCE
AND ENGINEERING

Orange Spheres



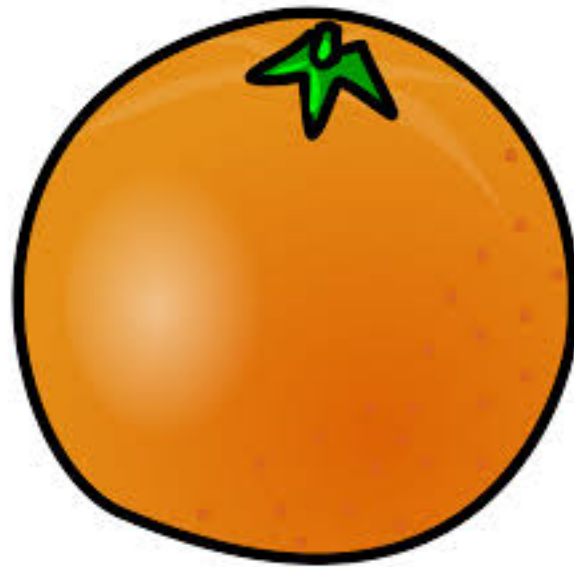
DEPARTMENT OF
COMPUTER SCIENCE
AND ENGINEERING

Texture Mapping



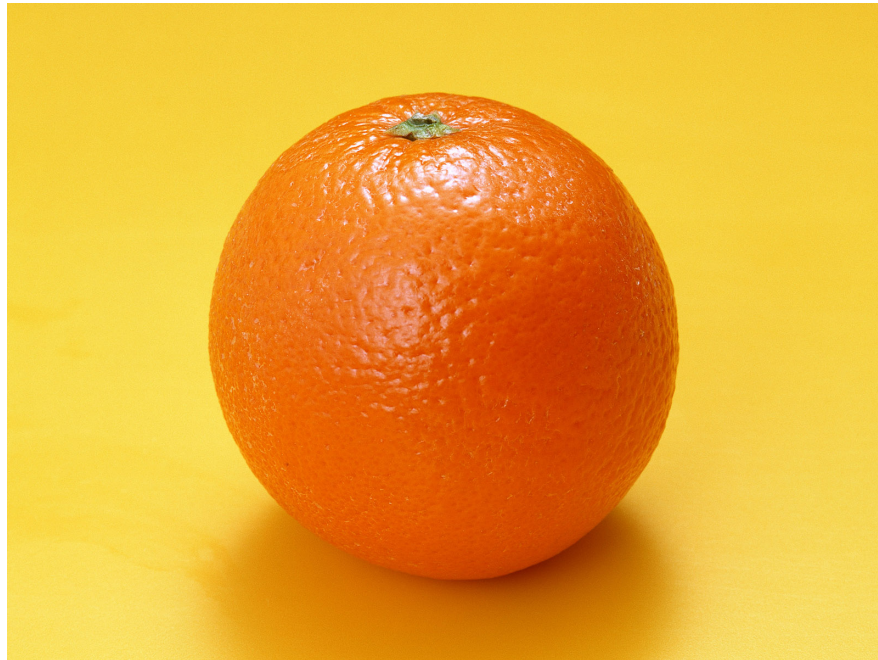
DEPARTMENT OF
COMPUTER SCIENCE
AND ENGINEERING

Looking Better

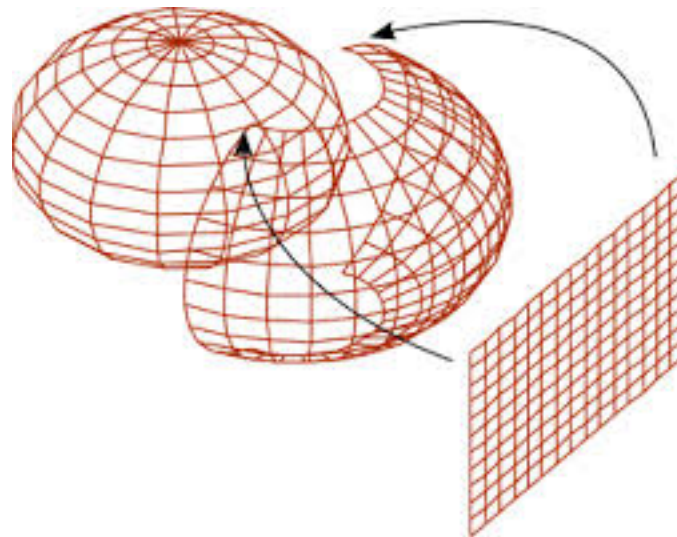
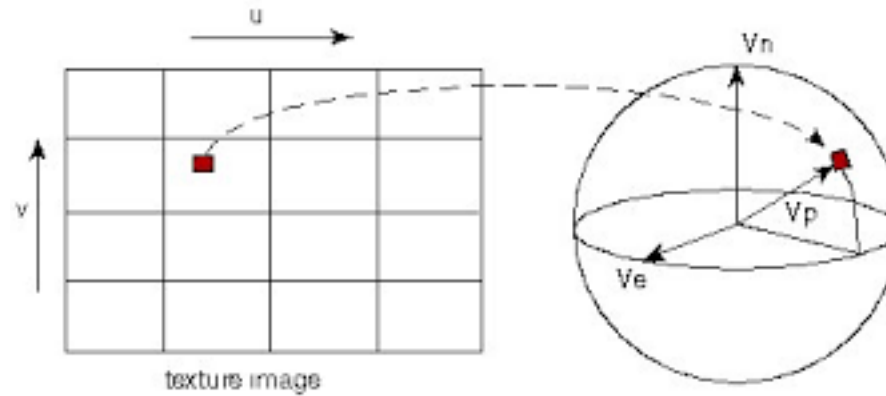


Still Not Enough

Local Variation



Texture Mapping



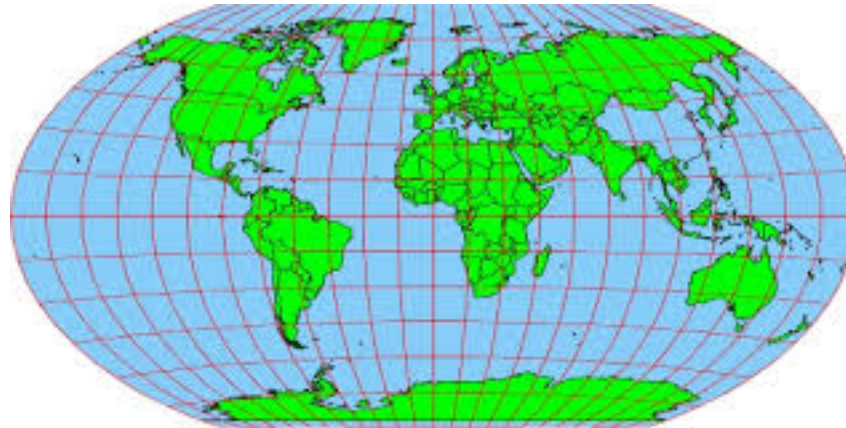
DEPARTMENT OF
COMPUTER SCIENCE
AND ENGINEERING

Globe



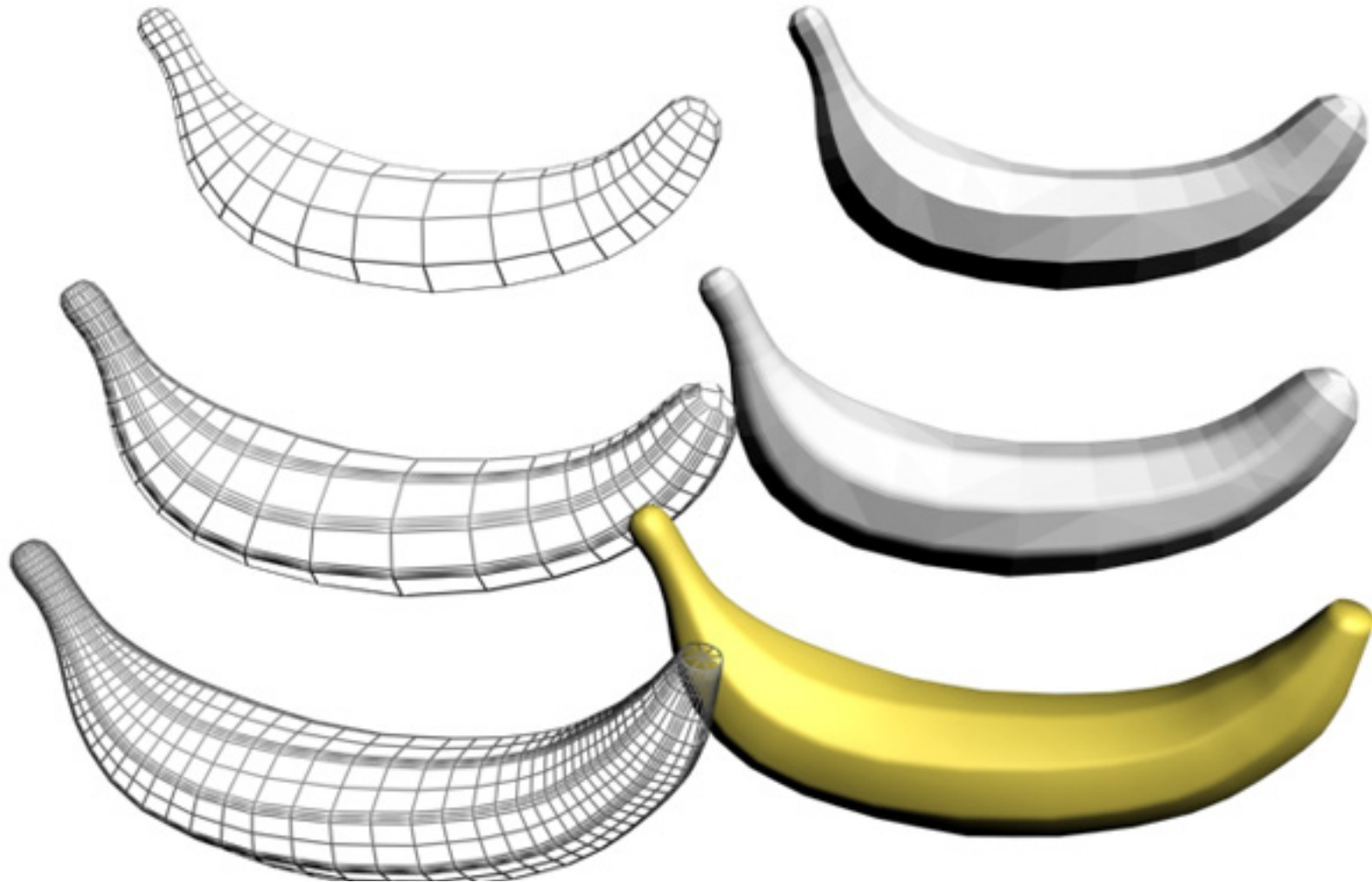
DEPARTMENT OF
COMPUTER SCIENCE
AND ENGINEERING

Not Mercator



DEPARTMENT OF
COMPUTER SCIENCE
AND ENGINEERING

Yet Another Fruit



DEPARTMENT OF
COMPUTER SCIENCE
AND ENGINEERING

Three Types of Mapping



17

DEPARTMENT OF
COMPUTER SCIENCE
AND ENGINEERING

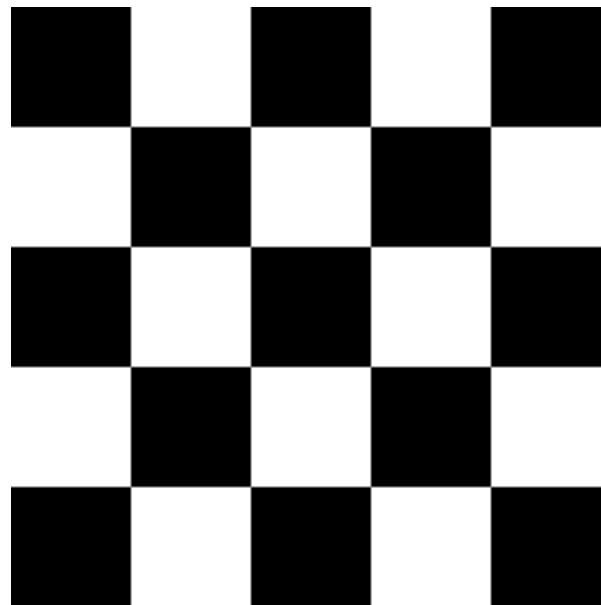
Generating Textures

Pictures



DEPARTMENT OF
COMPUTER SCIENCE
AND ENGINEERING

Algorithms



DEPARTMENT OF
COMPUTER SCIENCE
AND ENGINEERING

Checkerboard Texture

```
GLubyte image[64][64][3];
```

```
// Create a 64 x 64 checkerboard pattern
```

```
for ( int i = 0; i < 64; i++ ) {
```

```
    for ( int j = 0; j < 64; j++ ) {
```

```
        GLubyte c = (((i & 0x8) == 0) ^ ((j & 0x8) == 0)) * 255;
```

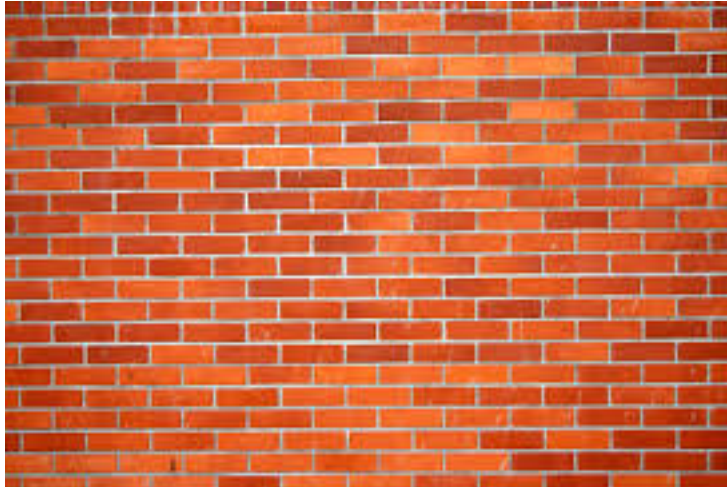
```
        image[i][j][0] = c;
```

```
        image[i][j][1] = c;
```

```
        image[i][j][2] = c;
```

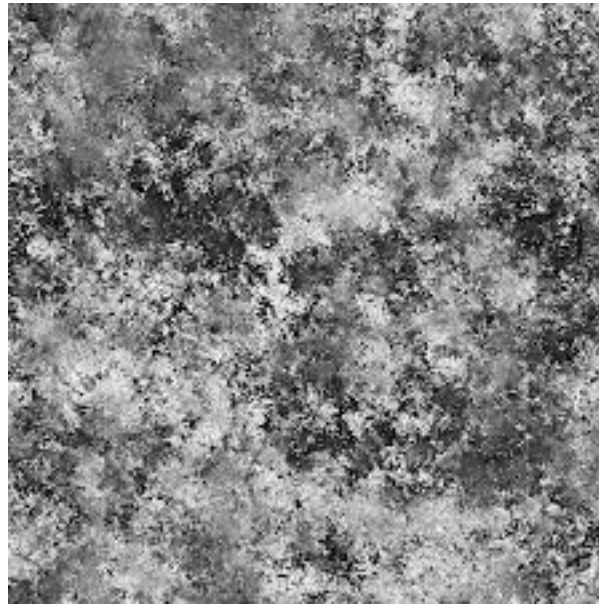


Brick Wall

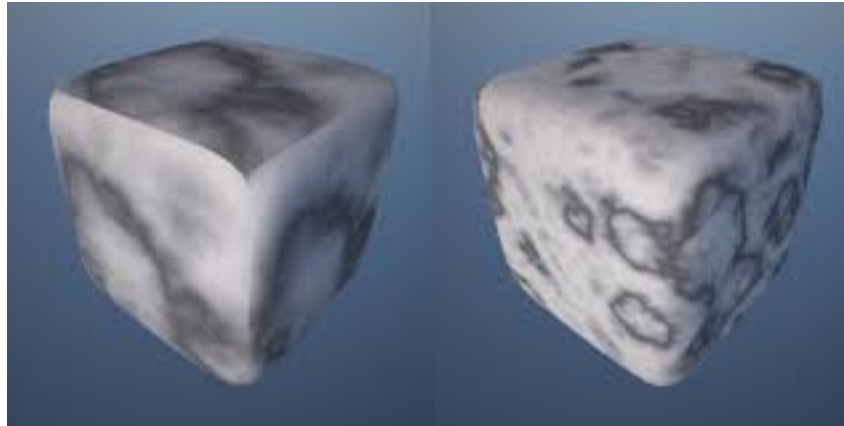


DEPARTMENT OF
COMPUTER SCIENCE
AND ENGINEERING

Noise

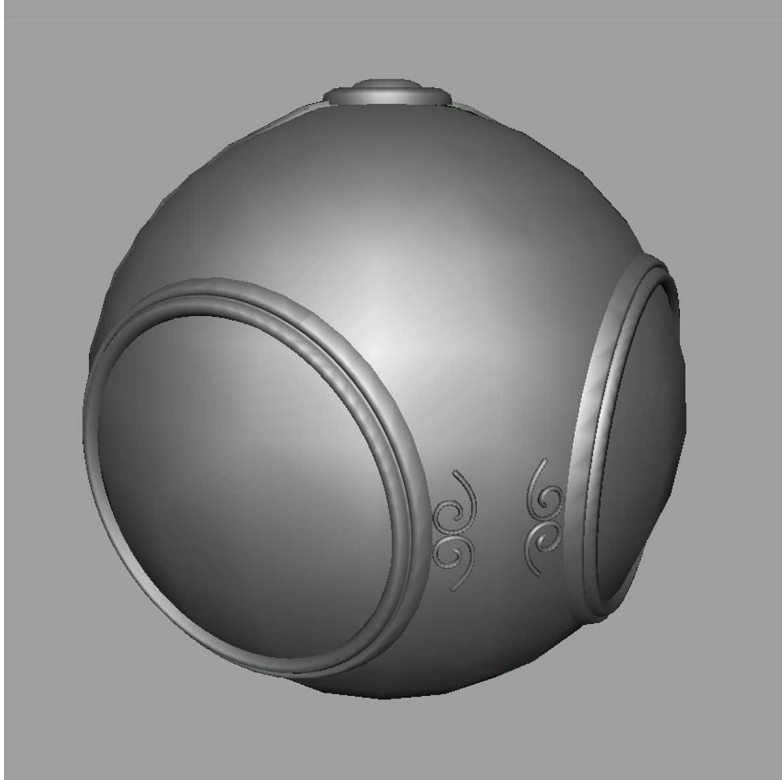


Marble



DEPARTMENT OF
COMPUTER SCIENCE
AND ENGINEERING

Texture Mapping



geometric model



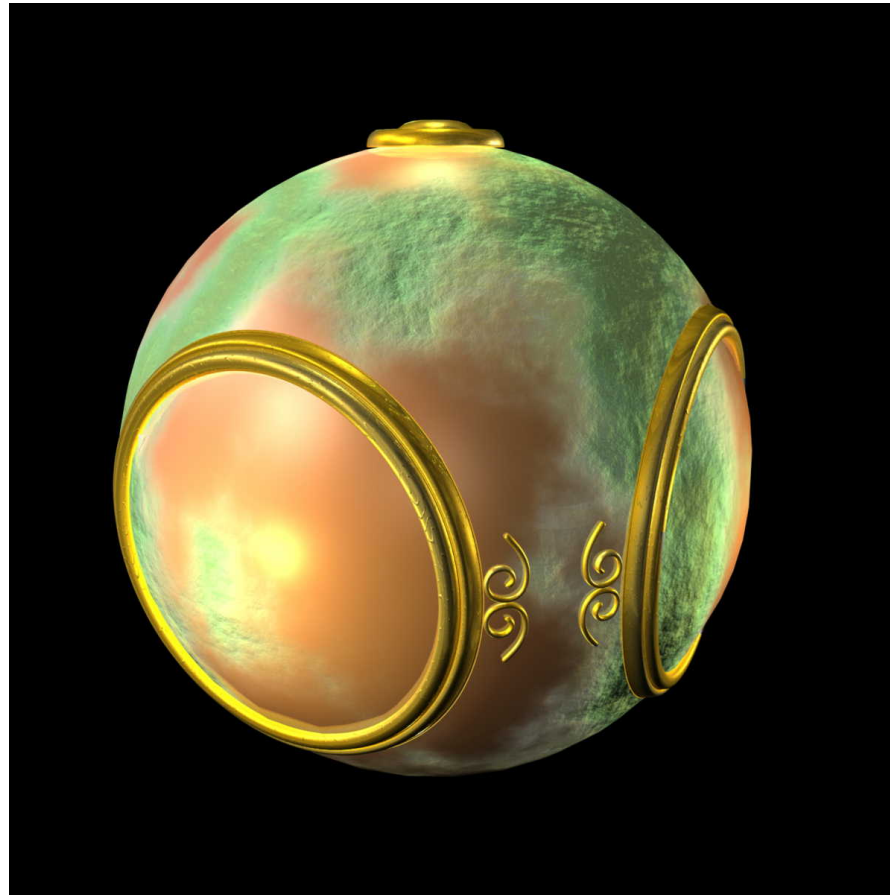
texture mapped



Environment Mapping

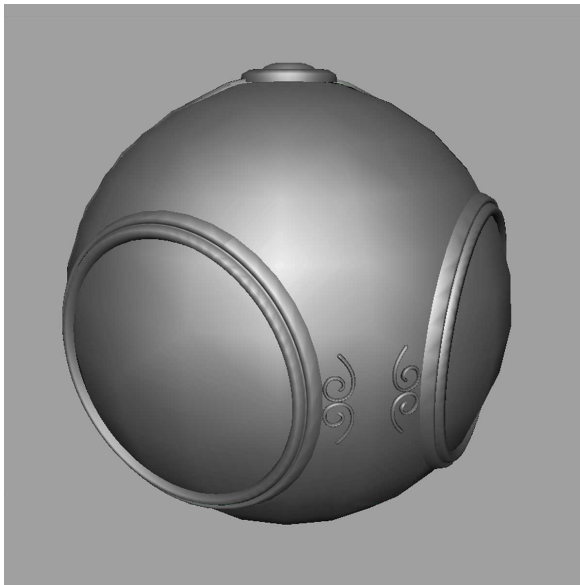


Bump Mapping



Three Types

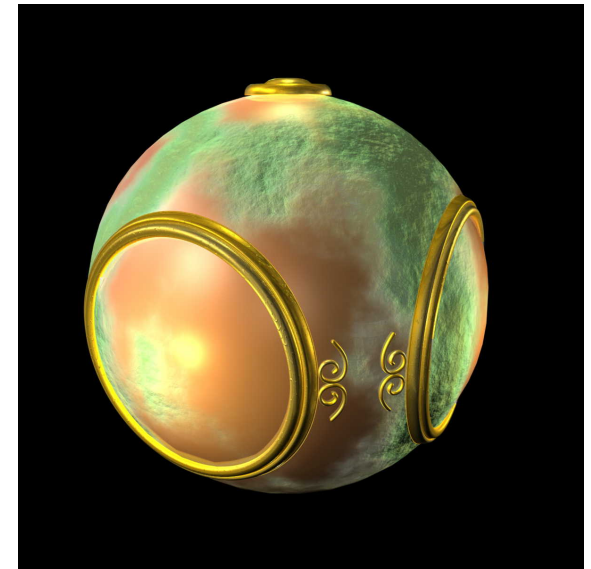
Texture mapping



smooth shading



environment
mapping



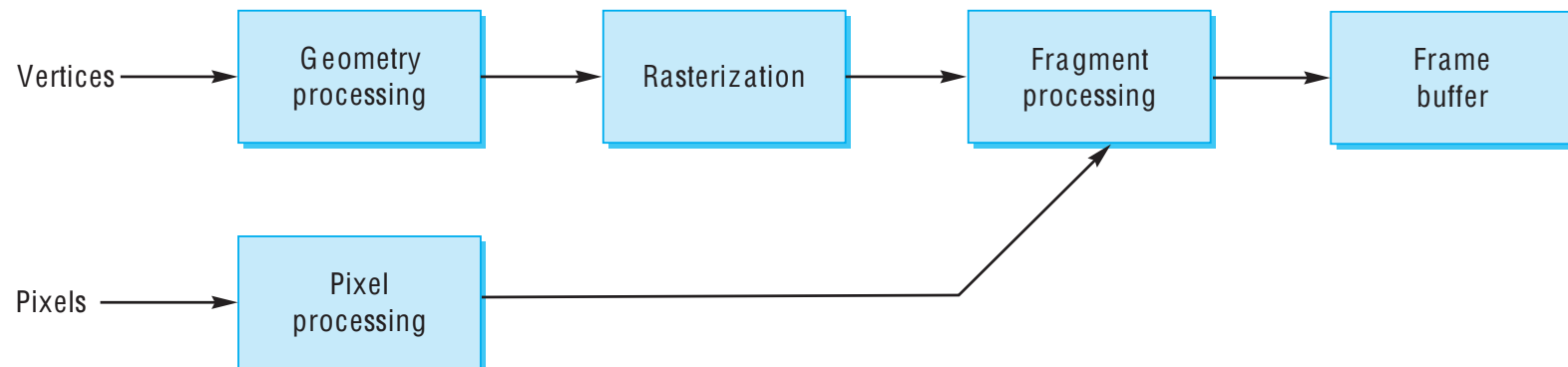
bump mapping



Texture Mapping - Pipeline

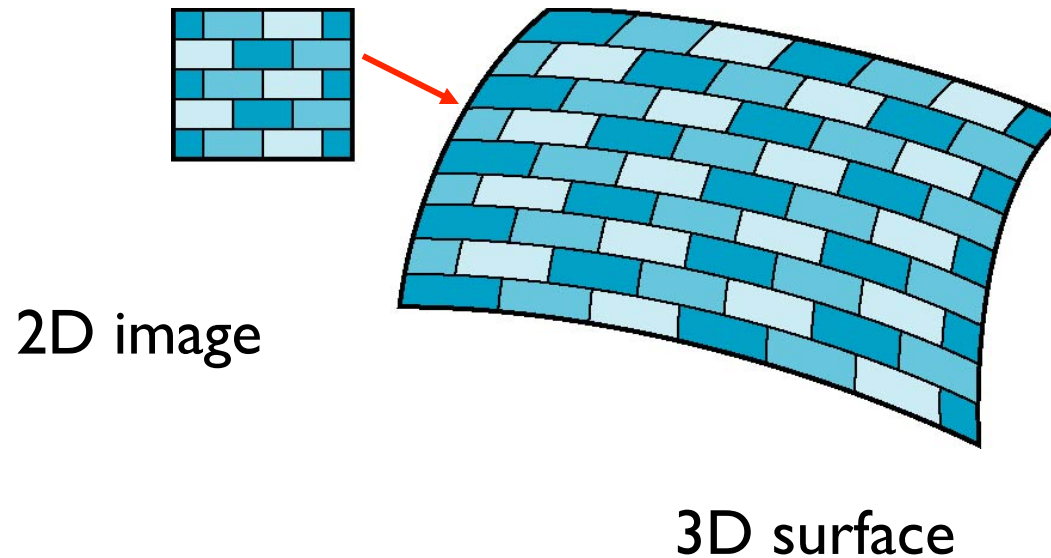
Mapping techniques are implemented at the end of the rendering pipeline

- Very efficient because few polygons make it past the clipper

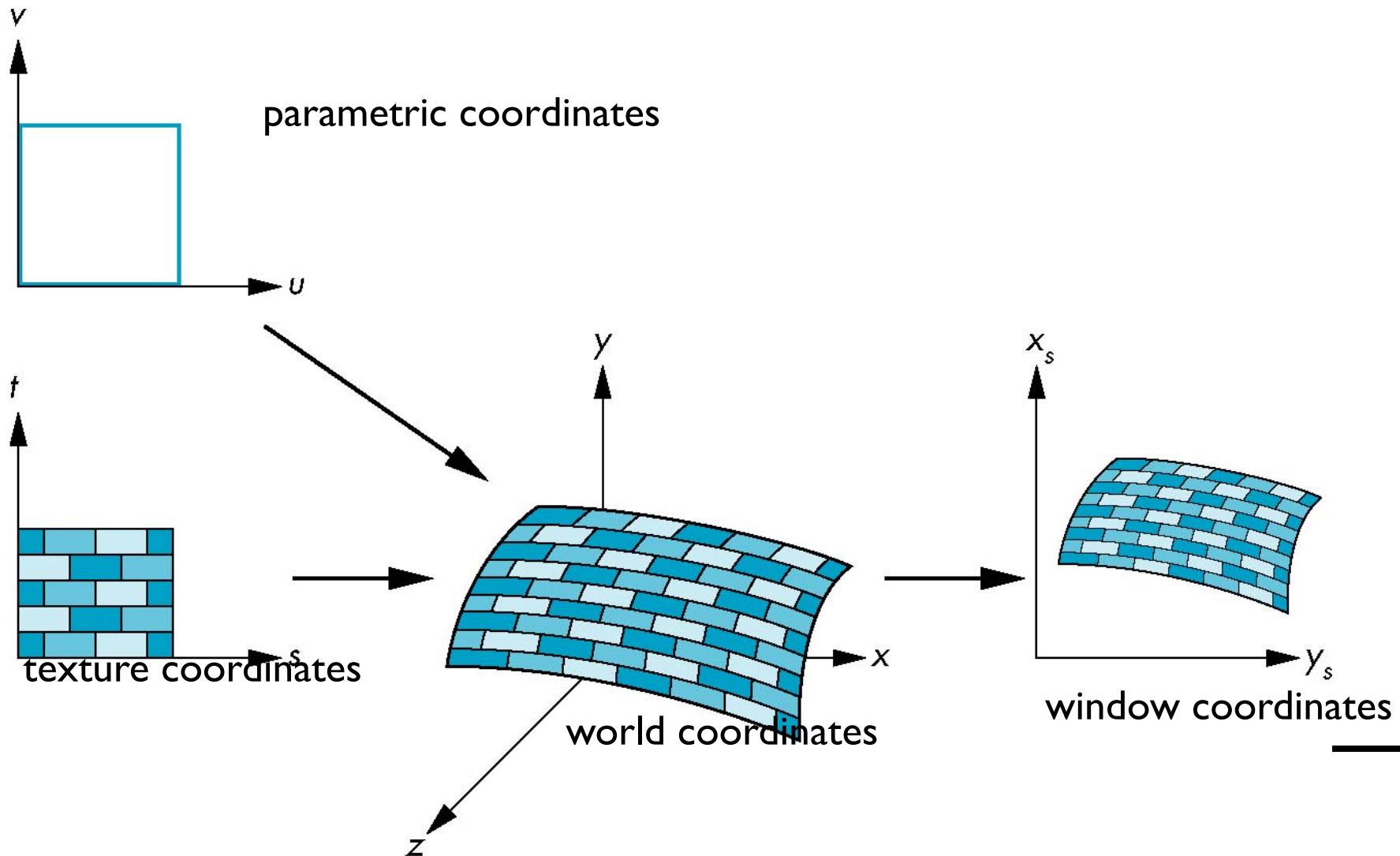


Mapping Mechanics

3 or 4 coordinate systems involved



Texture Mapping



Coordinate Systems

- Parametric coordinates
 - Model curves and surfaces
- Texture coordinates
 - Identify points in image to be mapped
- Object or World Coordinates
 - Conceptually, where the mapping takes place
- Screen Coordinates
 - Where the final image is really produced



Mapping Functions

Mapping from texture coords to point on surface

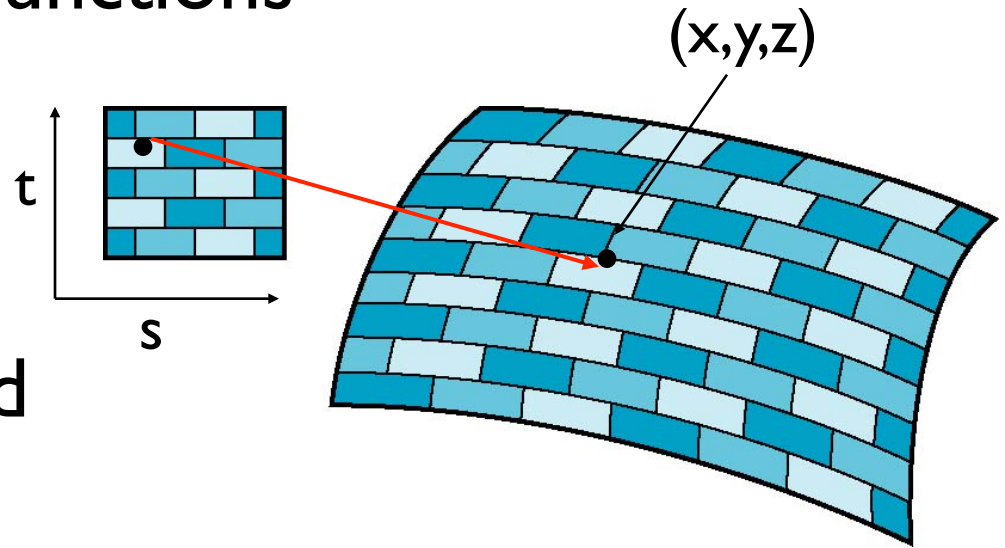
- Appear to need three functions

$$x = x(s,t)$$

$$y = y(s,t)$$

$$z = z(s,t)$$

- Other direction needed



Backward Mapping

Mechanics

- Given a pixel want point on object it corresponds
- Given point on object want point in the texture it corresponds

Need a map of the form

$$s = s(x,y,z)$$

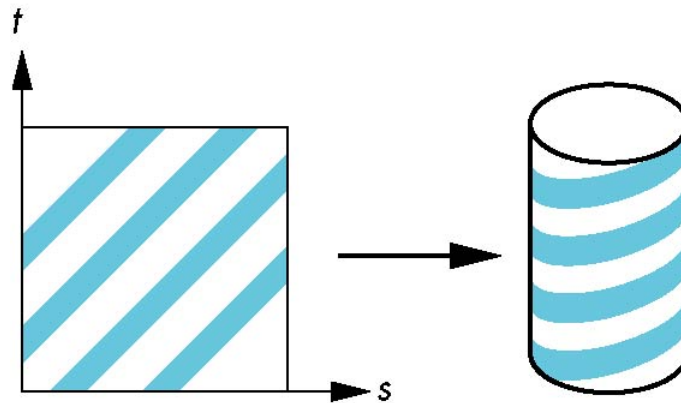
$$t = t(x,y,z)$$

Such functions are difficult to find in general



Two-part mapping

- First map texture to a simple intermediate surface
- Map to cylinder



Cylindrical Mapping

parametric cylinder

$$x = r \cos 2\pi u$$

$$y = r \sin 2\pi u$$

$$z = v/h$$

maps rectangle in u,v space to cylinder
of radius r and height h in world coordinates

$$s = u$$

$$t = v$$

maps from texture space



Spherical Map

We can use a parametric sphere

$$x = r \cos 2\pi u$$

$$y = r \sin 2\pi u \cos 2\pi v$$

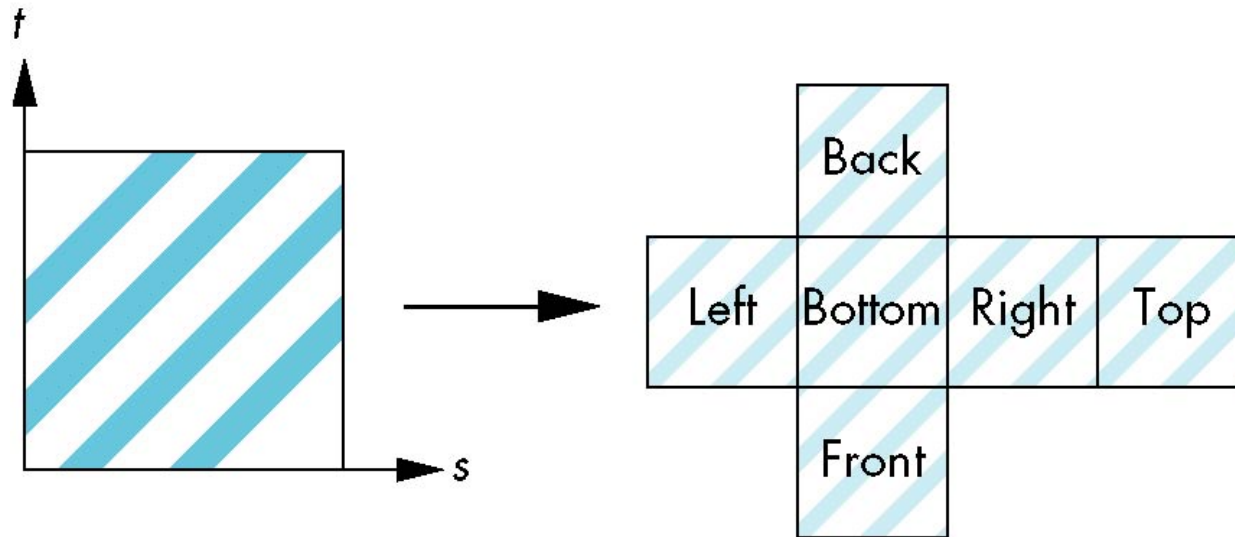
$$z = r \sin 2\pi u \sin 2\pi v$$

in a similar manner to the cylinder
but have to decide where to put
the distortion

Spheres are used in environmental maps

Box Mapping

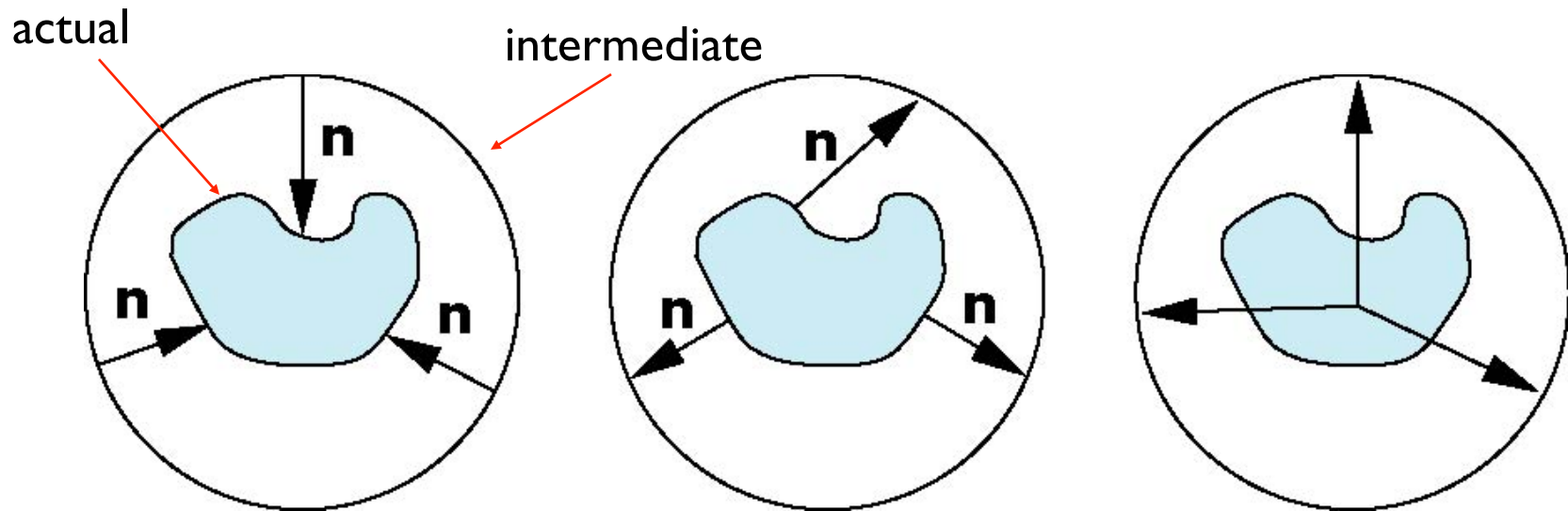
- Easy to use with simple orthographic projection
- Also used in environment maps



Second Mapping

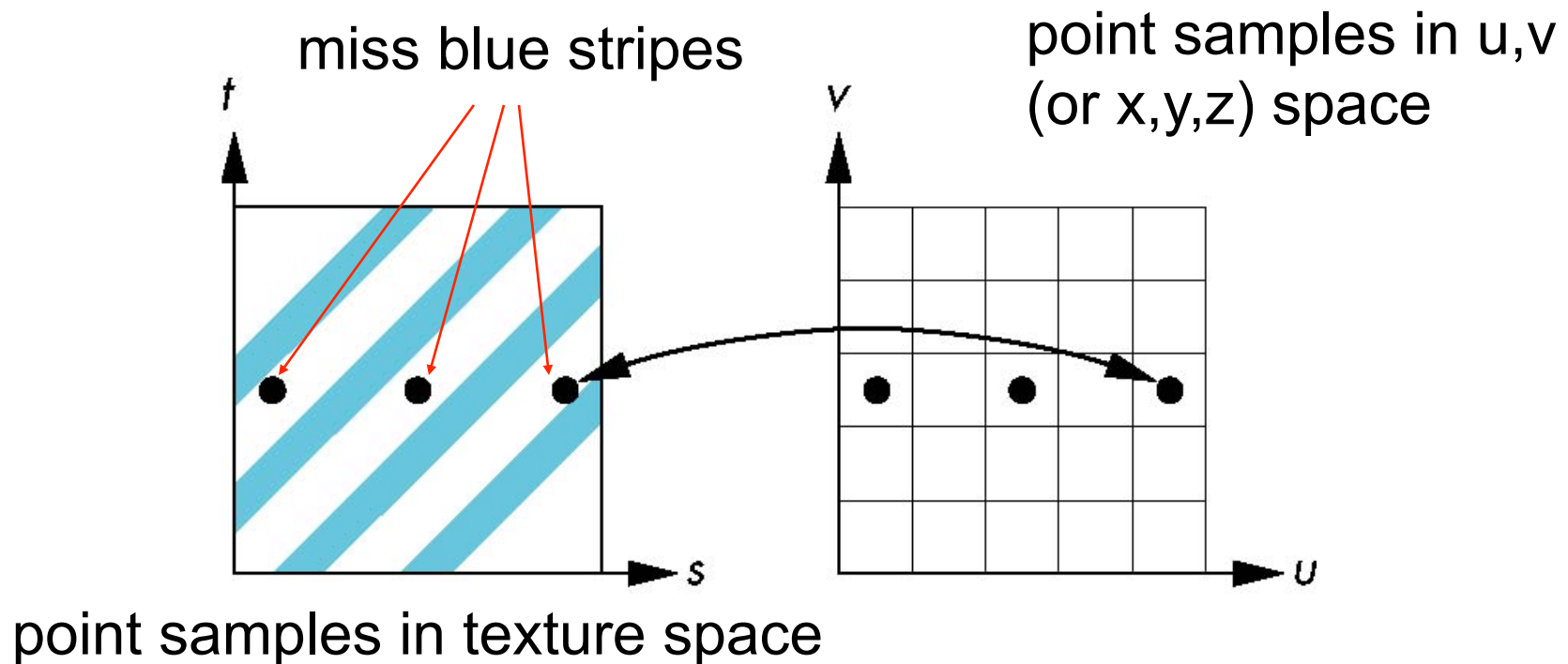
Map from intermediate object to actual object

- Normals from intermediate to actual
- Normals from actual to intermediate
- Vectors from center of intermediate



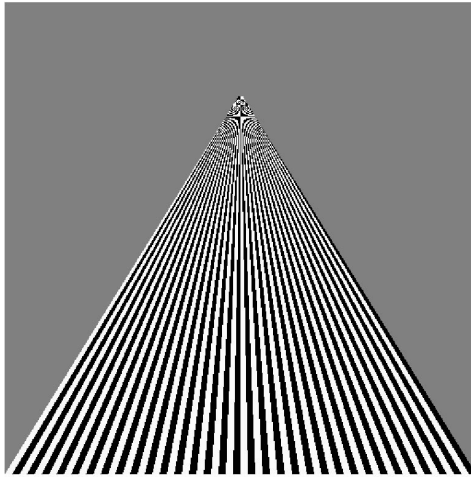
Aliasing

Point sampling of texture leads to aliasing errors

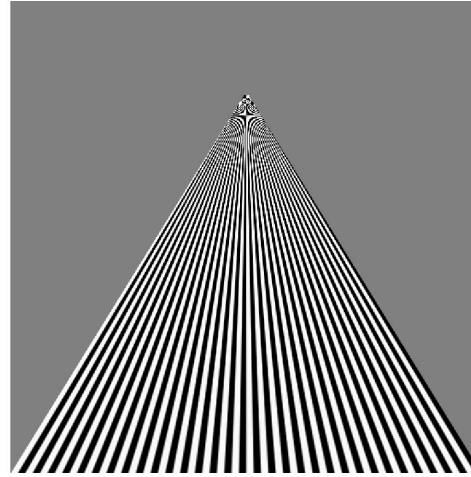


Anti-Aliasing in Textures

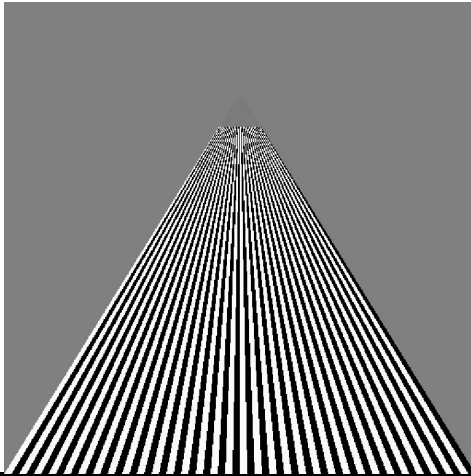
point
sampling



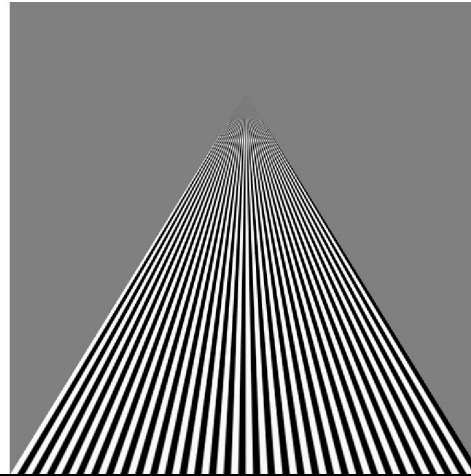
linear
filtering



mipmapped
point
sampling

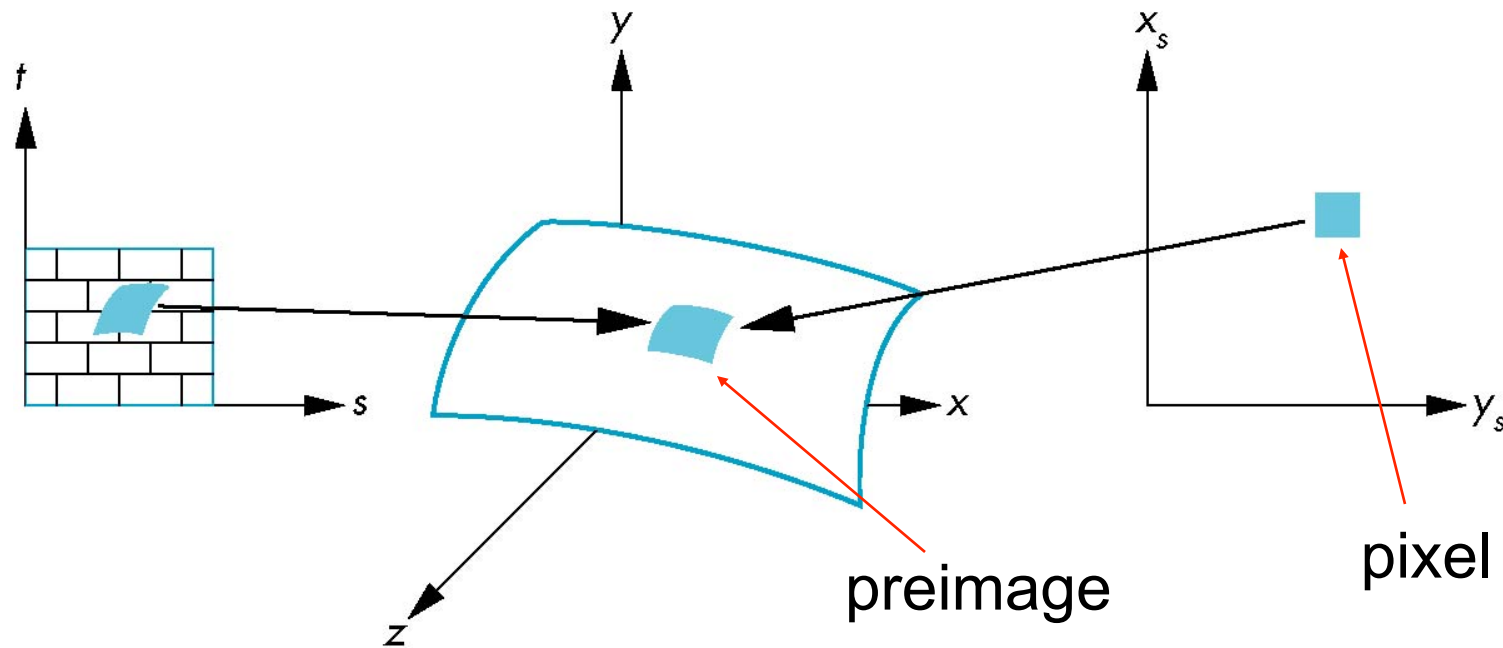


mipmapped
linear
filtering



Area Averaging

A better but slower option is to use *area averaging*



Note that *preimage* of pixel is curved

OpenGL Texture

Basic Strategy

Three steps

1. Specify texture

- read or generate image
- assign to texture
- enable texturing

2. Assign texture coordinates to vertices

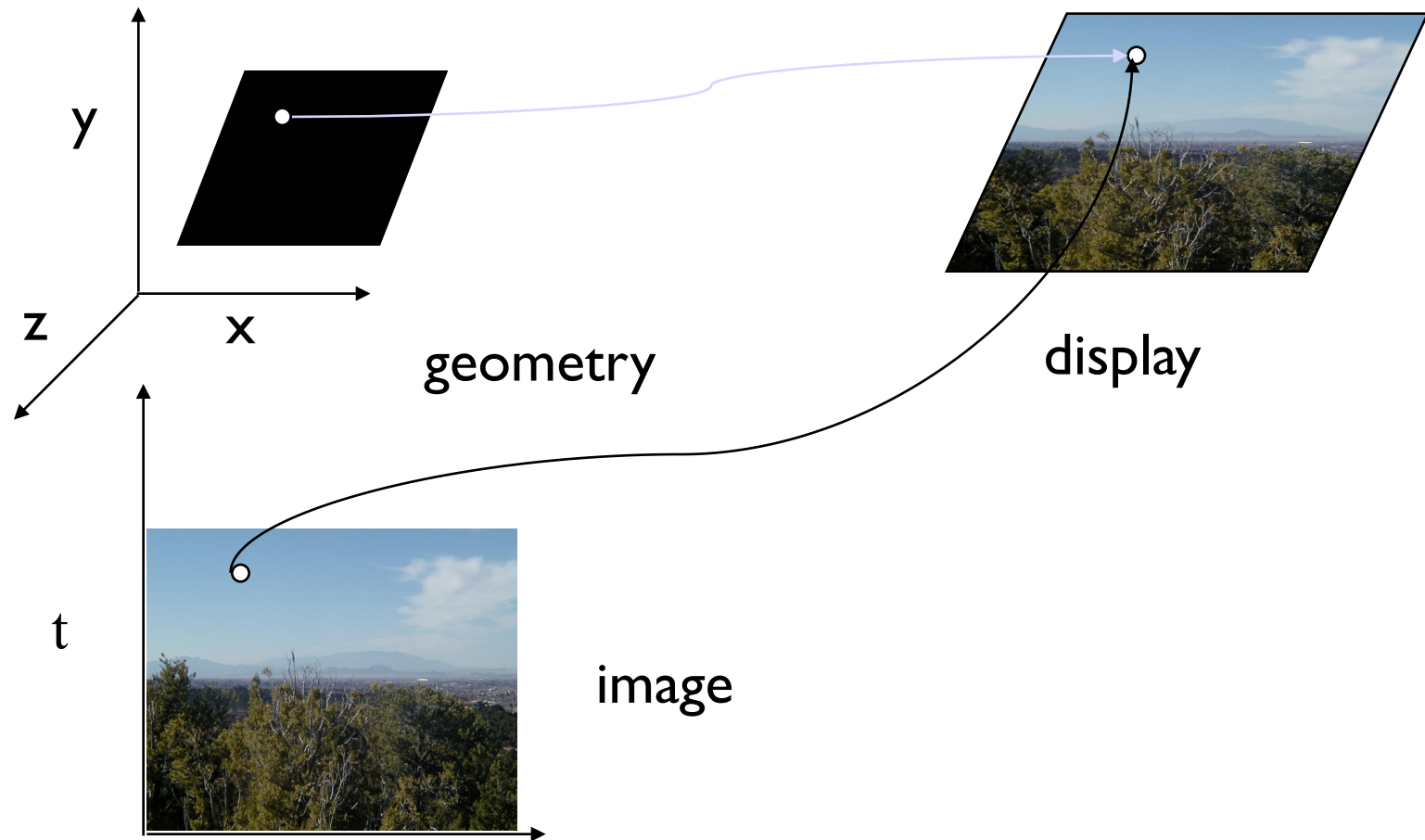
- Proper mapping function is left to application

3. Specify texture parameters

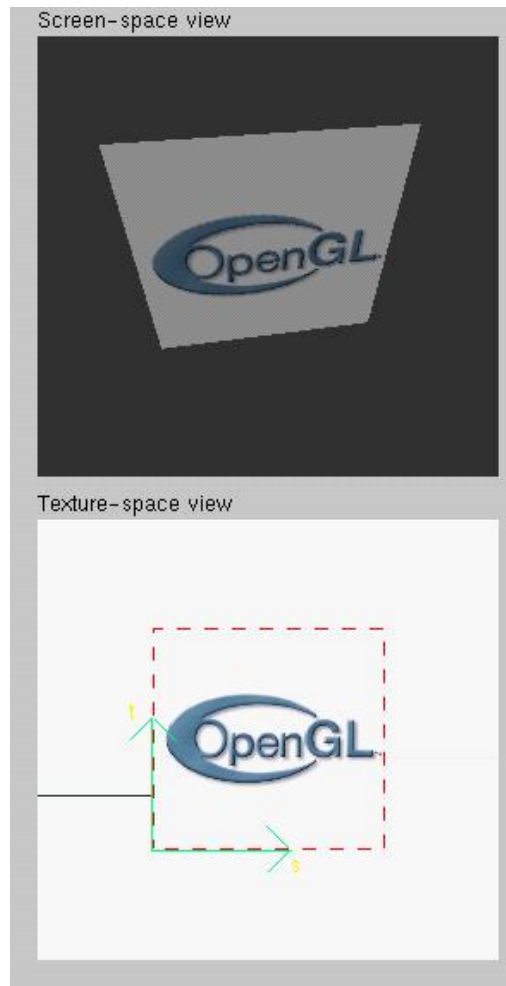
- wrapping, filtering



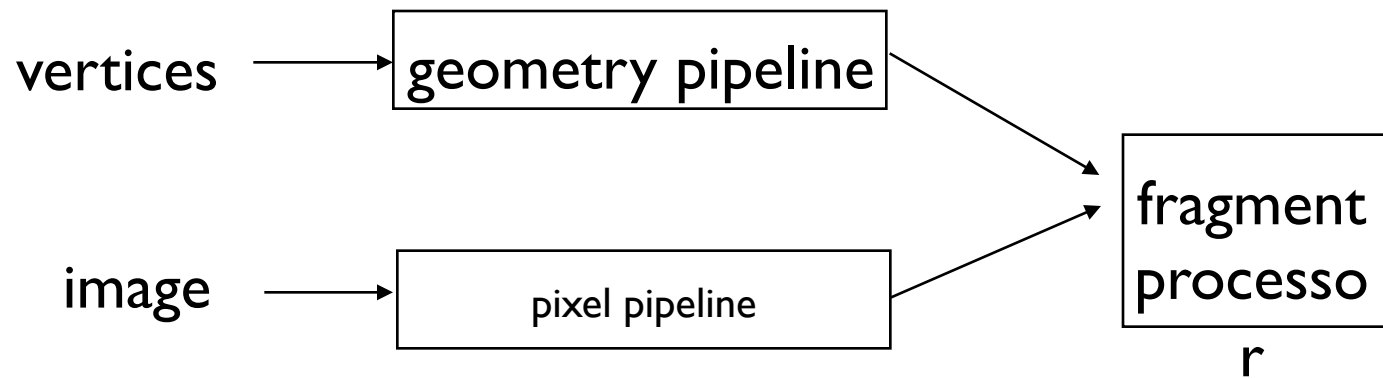
Texture Mapping



Texture Example



Texture Mapping in OpenGL



Specifying a Texture Image

- Define a texture image from an array of *texels* (texture elements) in CPU memory
Glubyte my_texels[512][512];
- Define as any other pixel map
 - Scanned image
 - Generate by application code
- Enable texture mapping
 - **glEnable(GL_TEXTURE_2D)**
 - OpenGL supports 1-4 dimensional texture maps

Defining a Texture Image

```
glTexImage2D( target, level, components, w, h, border, format, type, texels );
```

target: type of texture, e.g. GL_TEXTURE_2D

level: used for mipmapping

components: elements per texel

w, h: width and height of texels in pixels

border: used for smoothing

format and type: describe texels

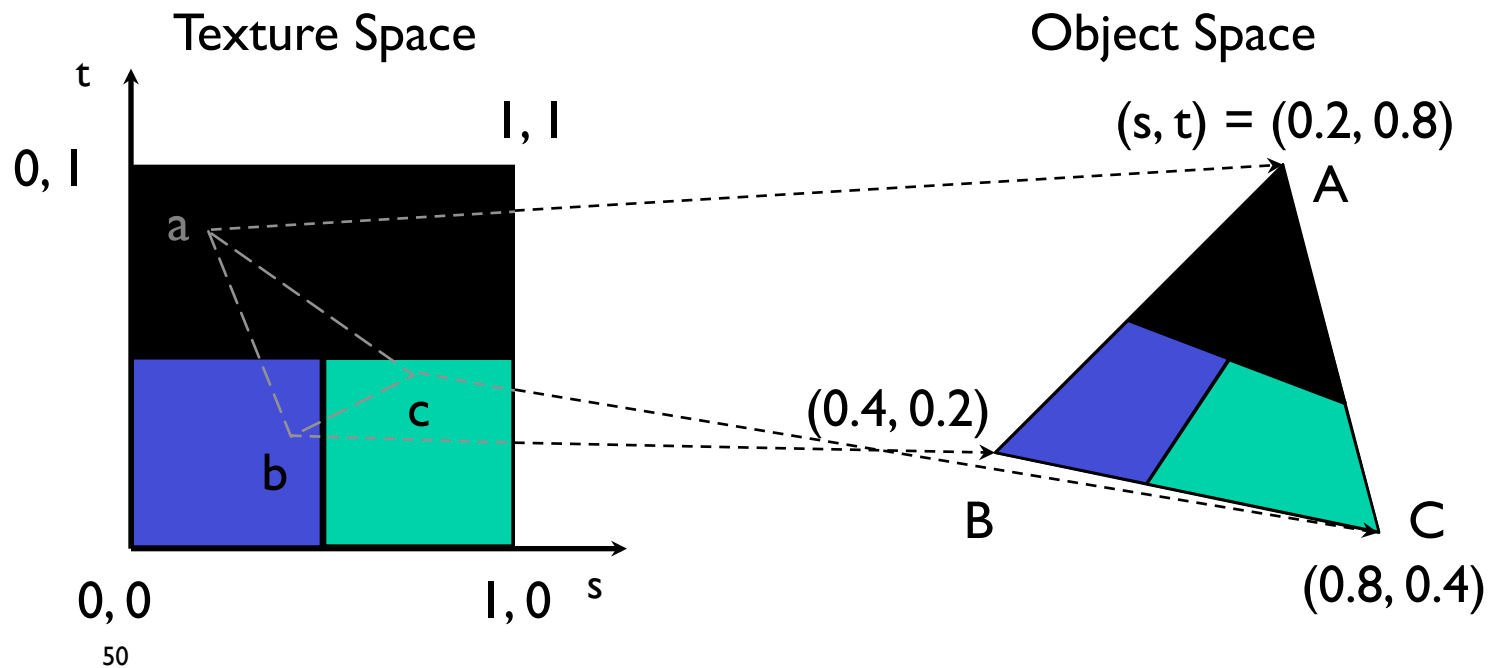
texels: pointer to texel array

```
glTexImage2D(GL_TEXTURE_2D, 0, 3, 512, 512, 0, GL_RGB,  
GL_UNSIGNED_BYTE, my_texels);
```



Mapping a Texture

- Based on parametric texture coordinates
- **glTexCoord*()** specified at each vertex



50



GLSL - Typical Code

```
offset = 0;
GLuint vPosition = glGetAttribLocation( program, "vPosition" );
glEnableVertexAttribArray( vPosition );
glVertexAttribPointer( vPosition, 4, GL_FLOAT, GL_FALSE,
    0, BUFFER_OFFSET(offset) );

offset += sizeof(points);
GLuint vTexCoord = glGetAttribLocation( program, "vTexCoord" );
glEnableVertexAttribArray( vTexCoord );
glVertexAttribPointer( vTexCoord, 2, GL_FLOAT,
    GL_FALSE, 0, BUFFER_OFFSET(offset) );
```



Adding Texture Coordinates

```
void quad( int a, int b, int c, int d )
{
    quad_colors[Index] = colors[a];
    points[Index] = vertices[a];
    tex_coords[Index] = vec2( 0.0, 0.0 );
    index++;
    quad_colors[Index] = colors[a];
    points[Index] = vertices[b];
    tex_coords[Index] = vec2( 0.0, 1.0 );
    Index++;

    // other vertices
}
```

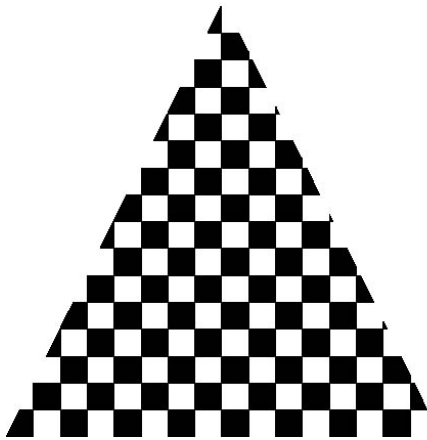
Role of Interpolation

Interpolation

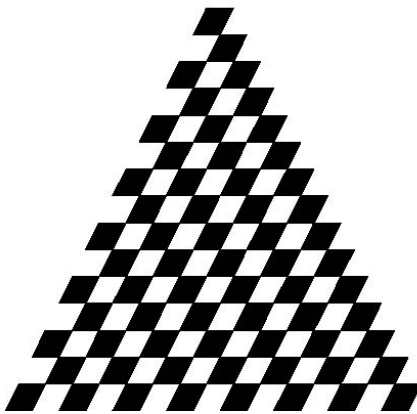
OpenGL uses interpolation to find proper texels from specified texture coordinates

Can be distorted

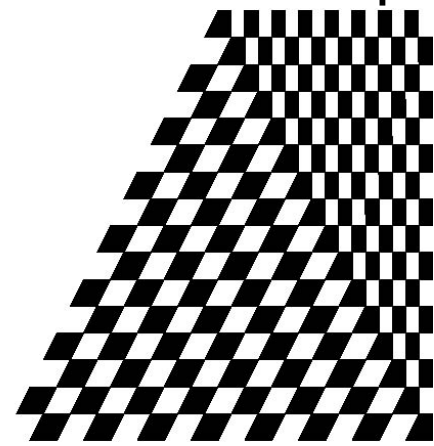
good selection
of tex coordinates



poor selection
of tex coordinates

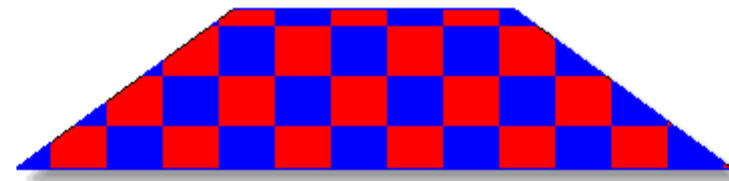
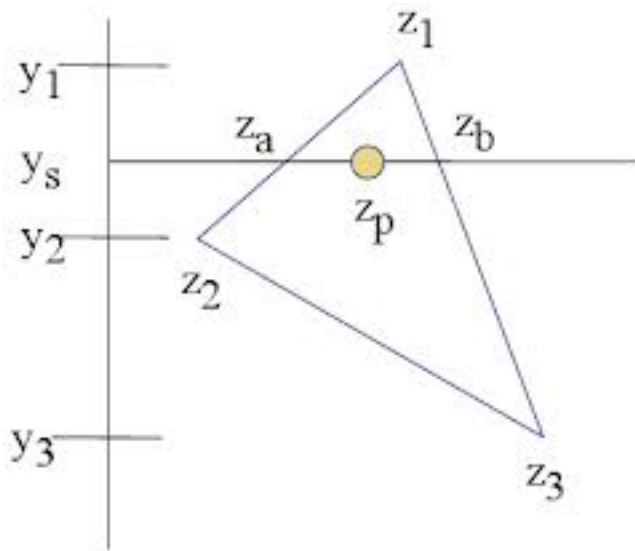


texture stretched
over trapezoid
showing effects of
bilinear interpolation

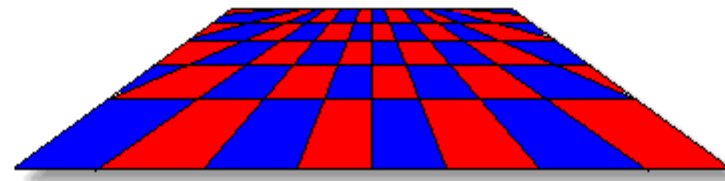


Interpolation

Figure 1.0 - Affine and perspective texture mapped polygons.



a. Affine texture mapping - notice no perspective cues.



b. Perspective texture mapping - notice 3D perspective both near and far.



DEPARTMENT OF
COMPUTER SCIENCE
AND ENGINEERING

Control of Texture Mapping



DEPARTMENT OF
COMPUTER SCIENCE
AND ENGINEERING

Texture Parameters

OpenGL has a variety of parameters that determine how texture is applied

- Wrapping parameters determine what happens if s and t are outside the $(0,1)$ range
- Filter modes allow us to use area averaging instead of point samples
- Mipmapping allows us to use textures at multiple resolutions
- Environment parameters determine how texture mapping interacts with shading

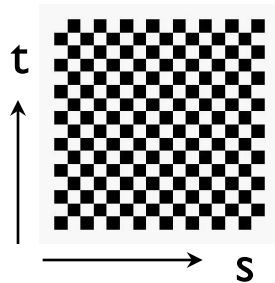
Wrapping Mode

Clamping: if $s, t > 1$ use 1, if $s, t < 0$ use 0

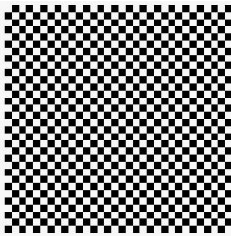
Wrapping: use s, t modulo 1

```
glTexParameteri( GL_TEXTURE_2D,  
                  GL_TEXTURE_WRAP_S, GL_CLAMP )
```

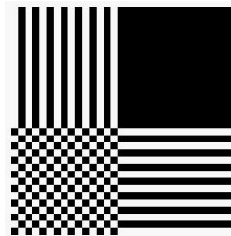
```
glTexParameteri( GL_TEXTURE_2D,  
                  GL_TEXTURE_WRAP_T, GL_REPEAT )
```



texture



GL_REPEAT
wrapping



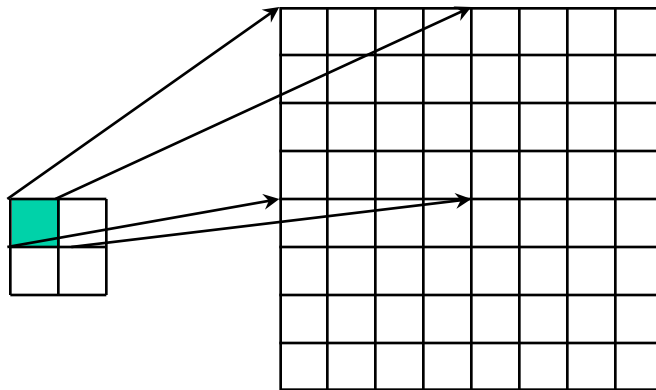
GL_CLAMP
wrapping



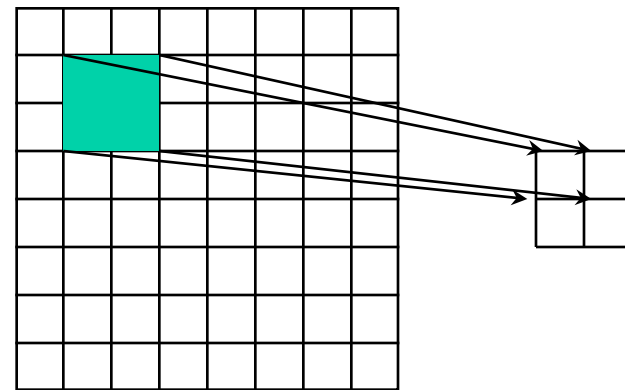
Magnification/Minification

More than one texel can cover a pixel (*minification*) or more than one pixel can cover a texel (*magnification*)

Can use point sampling (nearest texel) or linear filtering (2 x 2 filter) to obtain texture values



Texture Polygon
Magnification



Texture Polygon
Minification

Filter Modes

Modes determined by

– **glTexParameteri(target, type, mode)**

**glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
GL_NEAREST);**

**glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
GL_LINEAR);**

Note that linear filtering requires a border of an extra texel for filtering at edges (border = 1)

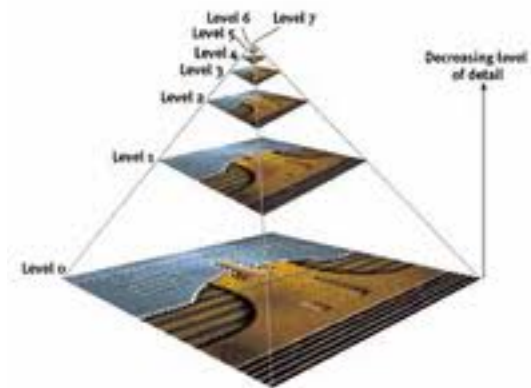
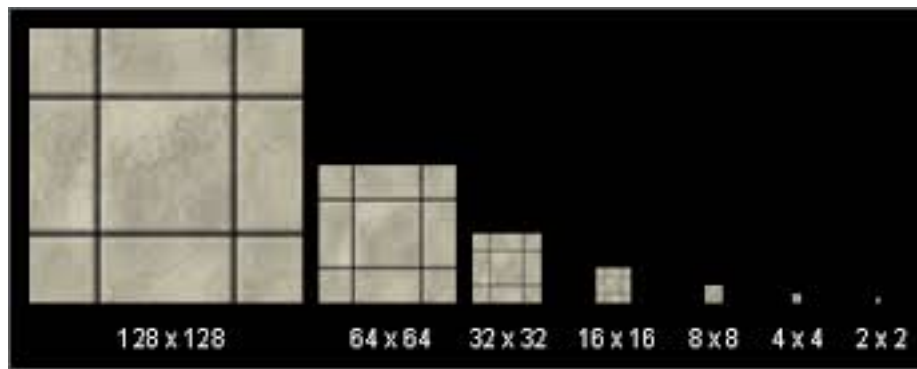


Mipmapped Textures

- *Mipmapping* allows for prefiltered texture maps of decreasing resolutions
- Lessens interpolation errors for smaller textured objects
- Declare mipmap level during texture definition
glTexImage2D(GL_TEXTURE_*D, level, ...)

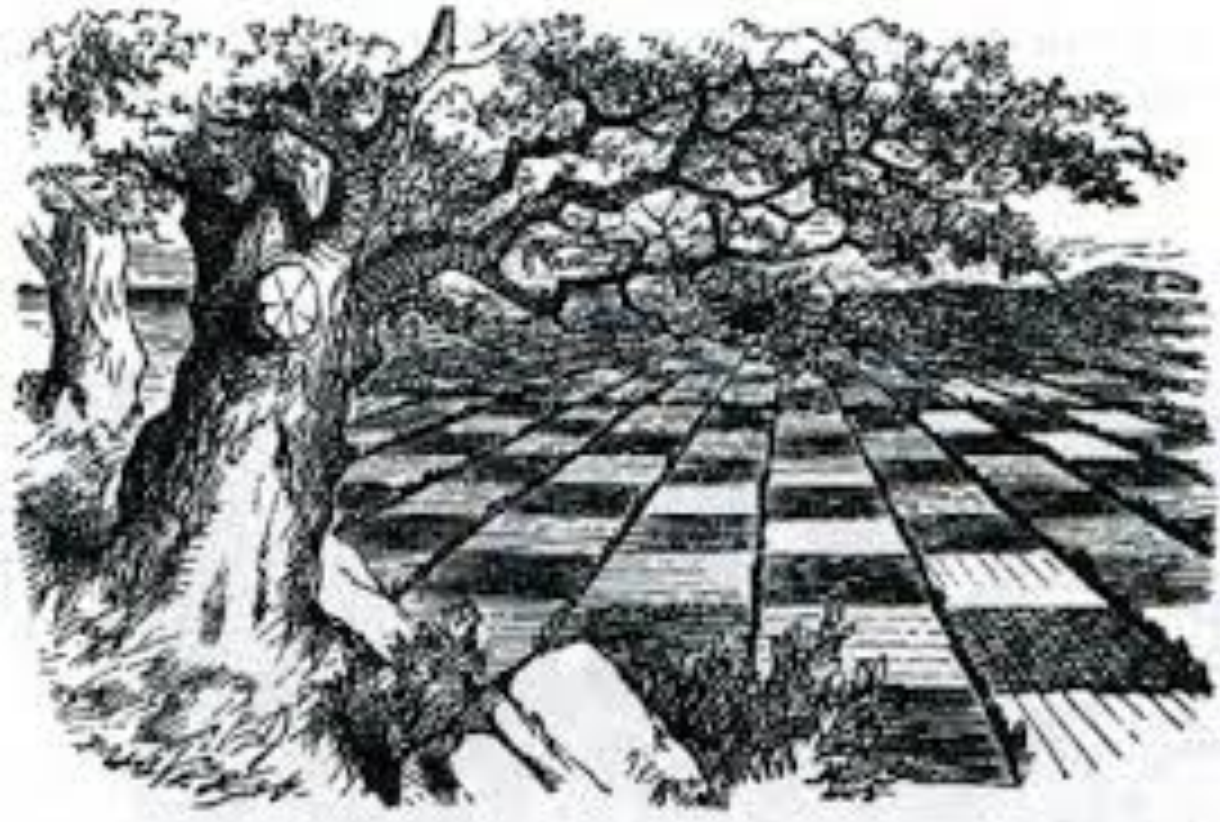


MipMaps



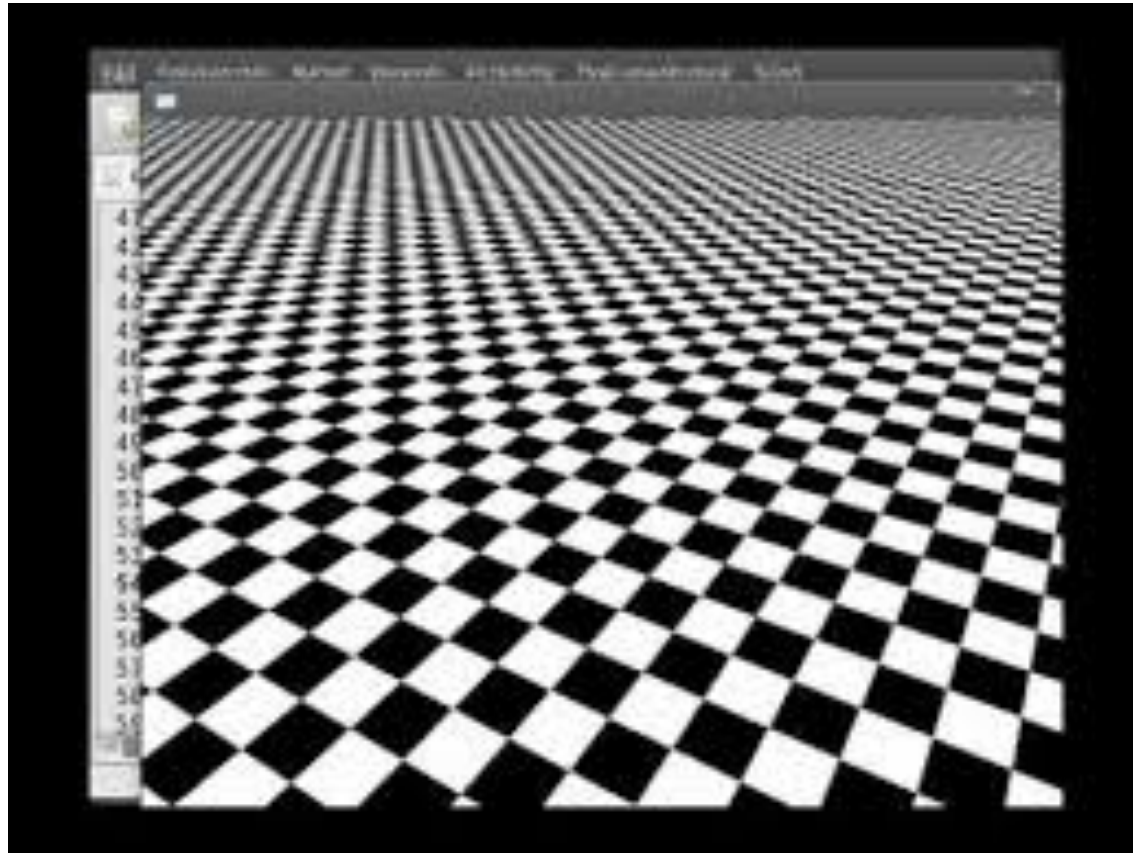
DEPARTMENT OF
COMPUTER SCIENCE
AND ENGINEERING

Mip-Mapping



DEPARTMENT OF
COMPUTER SCIENCE
AND ENGINEERING

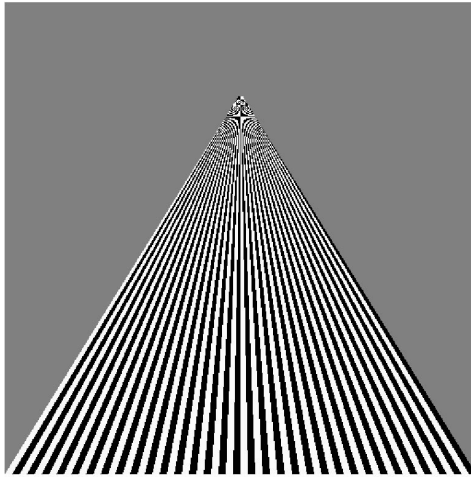
Mip-Mapping



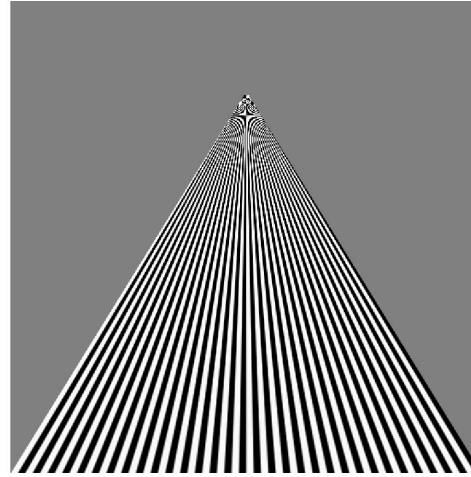
DEPARTMENT OF
COMPUTER SCIENCE
AND ENGINEERING

Example

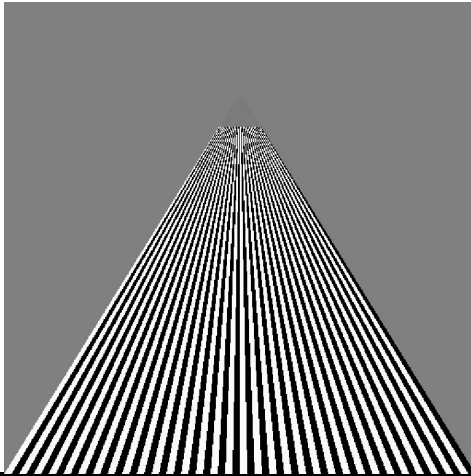
point
sampling



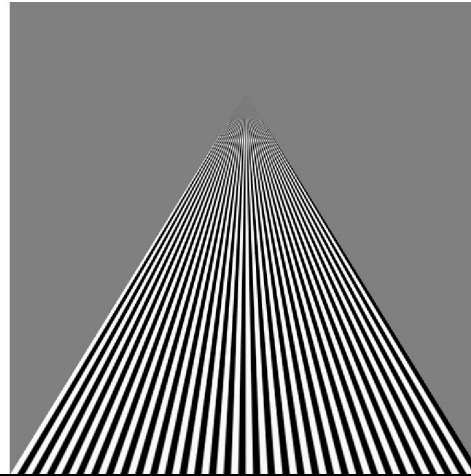
linear
filtering



mipmapped
point
sampling



mipmapped
linear
filtering



Texture Functions

- Controls how texture is applied
 - **glTexEnv{fi}[v](GL_TEXTURE_ENV, prop, param)**
- **GL_TEXTURE_ENV_MODE** modes
 - **GL_MODULATE**: modulates with computed shade
 - **GL_BLEND**: blends with an environmental color
 - **GL_REPLACE**: use only texture color
 - **GL(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);**
- Set blend color with
GL_TEXTURE_ENV_COLOR



Using Texture Objects

1. specify textures in texture objects
2. set texture filter
3. set texture function
4. set texture wrap mode
5. set optional perspective correction hint
6. bind texture object
7. enable texturing
8. supply texture coordinates for vertex
 - coordinates can also be generated



Other Texture Features

- Environment Maps
 - Start with image of environment through a wide angle lens
 - Can be either a real scanned image or an image created in OpenGL
 - Use this texture to generate a spherical map
 - Alternative is to use a cube map
- Multitexturing
 - Apply a sequence of textures through cascaded texture units

GLSL

Samplers

[https://www.opengl.org/wiki/Sampler_\(GLSL\)](https://www.opengl.org/wiki/Sampler_(GLSL))

Applying Textures

- Textures are applied during fragment shading by a **sampler**
- Samplers return a texture color from a texture object

in vec4 color; //color from rasterizer

in vec2 texCoord; //texture coordinate from rasterizer

uniform sampler2D texture; //texture object from application

```
void main() {  
    gl_FragColor = color * texture2D( texture, texCoord );  
}
```

Vertex Shader

- Usually vertex shader will output texture coordinates to be rasterized
- Must do all other standard tasks too

- Compute vertex position

- Compute vertex color if needed

in vec4 vPosition; //vertex position in object coordinates

in vec4 vColor; //vertex color from application

in vec2 vTexCoord; //texture coordinate from application

out vec4 color; //output color to be interpolated

out vec2 texCoord; //output tex coordinate to be interpolated

Adding Texture Coordinates

```
void quad( int a, int b, int c, int d )
{
    quad_colors[Index] = colors[a];
    points[Index] = vertices[a];
    tex_coords[Index] = vec2( 0.0, 0.0 );
    index++;
    quad_colors[Index] = colors[a];
    points[Index] = vertices[b];
    tex_coords[Index] = vec2( 0.0, 1.0 );
    Index++;

    // other vertices
}
```

Texture Object

```
GLuint textures[1];  
glGenTextures( 1, textures );  
  
glBindTexture( GL_TEXTURE_2D, textures[0] );  
glTexImage2D( GL_TEXTURE_2D, 0, GL_RGB, TextureSize,  
    TextureSize, 0, GL_RGB, GL_UNSIGNED_BYTE, image );  
glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_WRAP_S,  
    GL_REPEAT );  
glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_WRAP_T,  
    GL_REPEAT );  
glTexParameteri( GL_TEXTURE_2D,  
    GL_TEXTURE_MAG_FILTER, GL_NEAREST );  
glTexParameteri( GL_TEXTURE_2D,  
    GL_TEXTURE_MIN_FILTER, GL_NEAREST );  
glActiveTexture( GL_TEXTURE0 );
```

Linking with Shaders

```
GLuint vTexCoord = glGetAttribLocation( program, "vTexCoord" );
glEnableVertexAttribArray( vTexCoord );
glVertexAttribPointer( vTexCoord, 2, GL_FLOAT, GL_FALSE, 0,
                      BUFFER_OFFSET(offset) );
```

```
// Set the value of the fragment shader texture sampler variable
// ("texture") to the the appropriate texture unit. In this case,
// zero, for GL_TEXTURE0 which was previously set by calling
// glActiveTexture().
glUniform1i( glGetUniformLocation(program, "texture"), 0 );
```



Vertex Shader Applications

- Moving vertices
 - Morphing
 - Wave motion
 - Fractals
- Lighting
 - More realistic models
 - Cartoon shaders

Wave Motion Vertex Shader

```
uniform float time;
uniform float xs, zs, // frequencies
uniform float h; // height scale
uniform mat4 ModelView, Projection;
in vec4 vPosition;

void main() {
    vec4 t = vPosition;
    t.y = vPosition.y
        + h*sin(time + xs*vPosition.x)
        + h*sin(time + zs*vPosition.z);
    gl_Position = Projection*ModelView*t;
}
```

Particle System

```
uniform vec3 init_vel;
uniform float g, m, t;
uniform mat4 Projection, ModelView;
in vPosition;
void main(){
    vec3 object_pos;
    object_pos.x = vPosition.x + vel.x*t;
    object_pos.y = vPosition.y + vel.y*t
        + g/(2.0*m)*t*t;
    object_pos.z = vPosition.z + vel.z*t;
    gl_Position = Projection*
        ModelView*vec4(object_pos,1);
}
```



Example

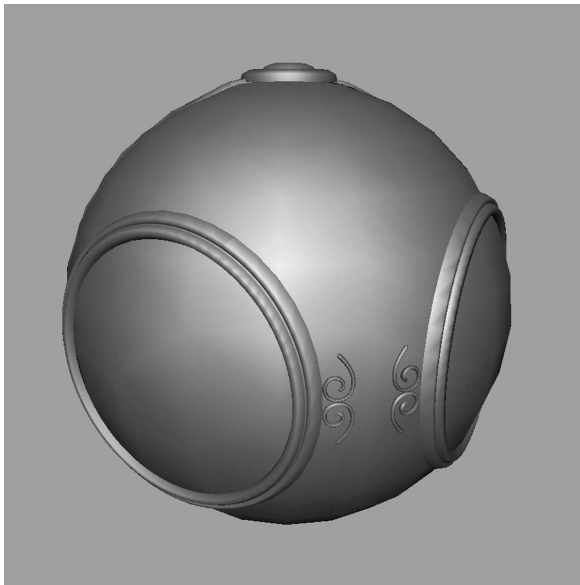
<http://www.lighthouse3d.com/tutorials/glsl-core-tutorial/glsl-core-tutorial-texturing-with-images/>

Example

<http://www.lighthouse3d.com/tutorials/glsl-tutorial/simple-texture/>

Fragment Shader

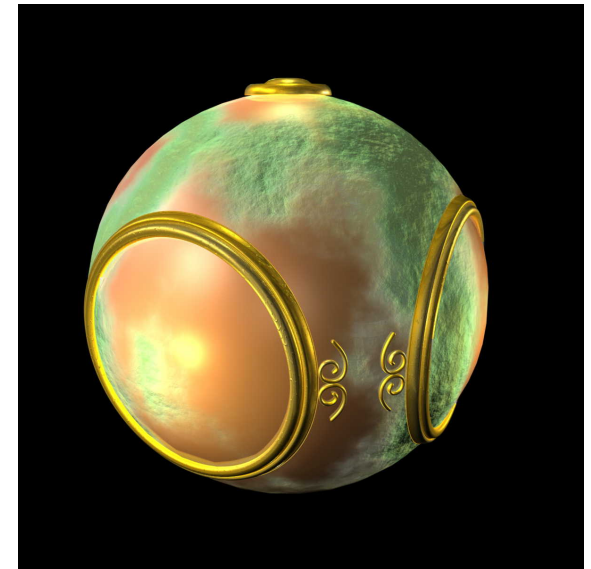
Texture mapping



smooth shading



environment
mapping



bump mapping



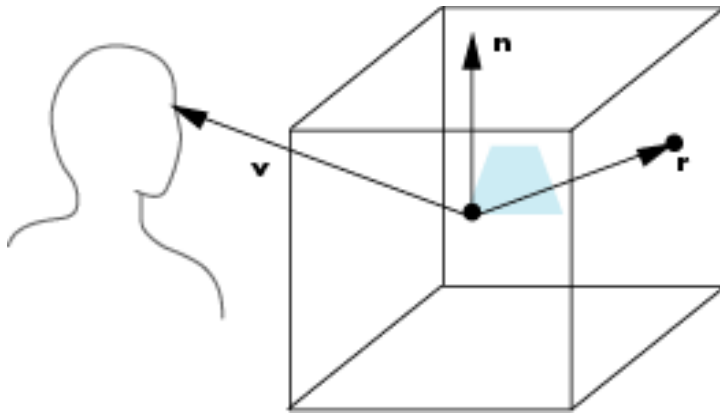
Cube Maps

- We can form a cube map texture by defining six 2D texture maps that correspond to the sides of a box
- Supported by OpenGL
- Also supported in GLSL through cubemap sampler
`vec4 texColor = textureCube(mycube, texcoord);`
- Texture coordinates must be 3D



Environment Map

Use reflection vector to locate texture in cube map



Environment Maps with Shaders

- Computed in world coordinates
 - keep track of modeling matrix & pass as a uniform variable
- Use reflection map or refraction map
- Simulate water

Reflection Map Vertex Shader

```
uniform mat4 Projection, ModelView, NormalMatrix;  
in vec4 vPosition;  
in vec4 normal;  
out vec3 R;
```

```
void main(void)  
{  
    gl_Position = Projection*ModelView*vPosition;  
    vec3 N = normalize(NormalMatrix*normal);  
    vec4 eyePos = ModelView*gvPosition;  
    R = reflect(-eyePos.xyz, N);  
}
```



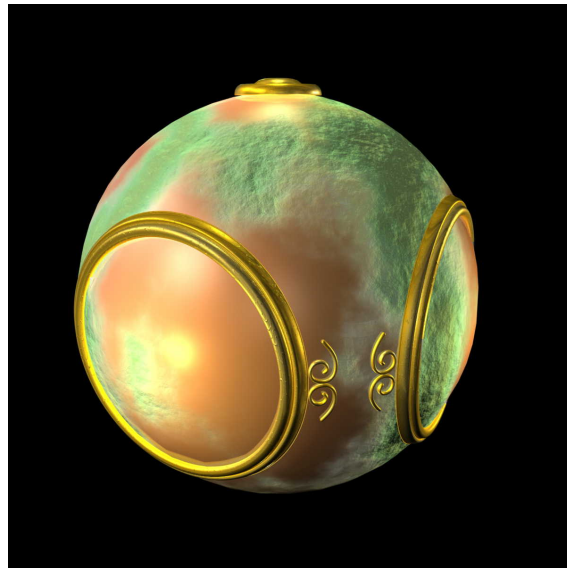
Reflection Map Fragment Shader

```
in vec3 R;  
uniform samplerCube texMap;  
  
void main(void)  
{  
    gl_FragColor = textureCube(texMap, R);  
}
```

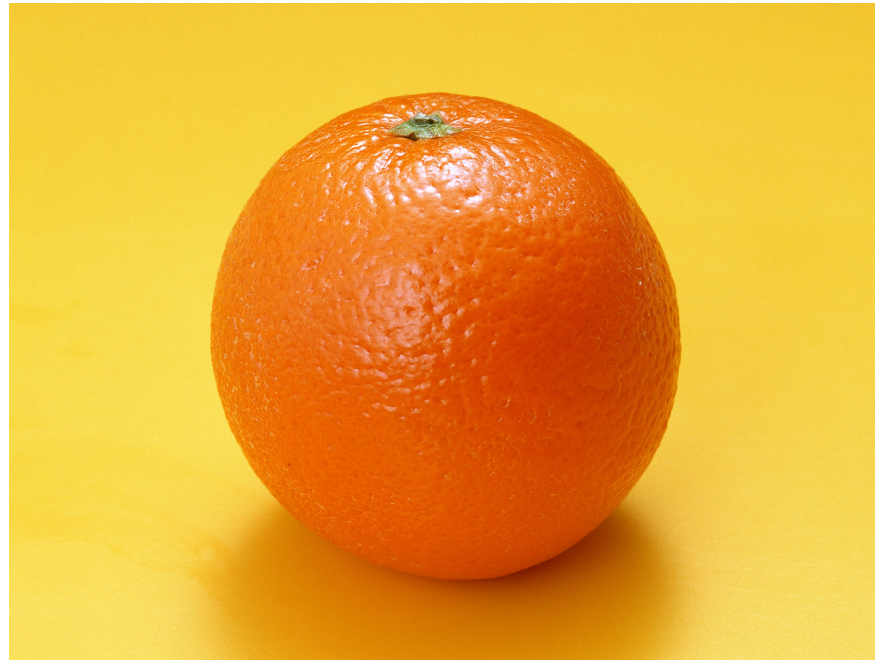


Bump Mapping

- Perturb normal for each fragment
- Store perturbation as textures



Back 2 Orange



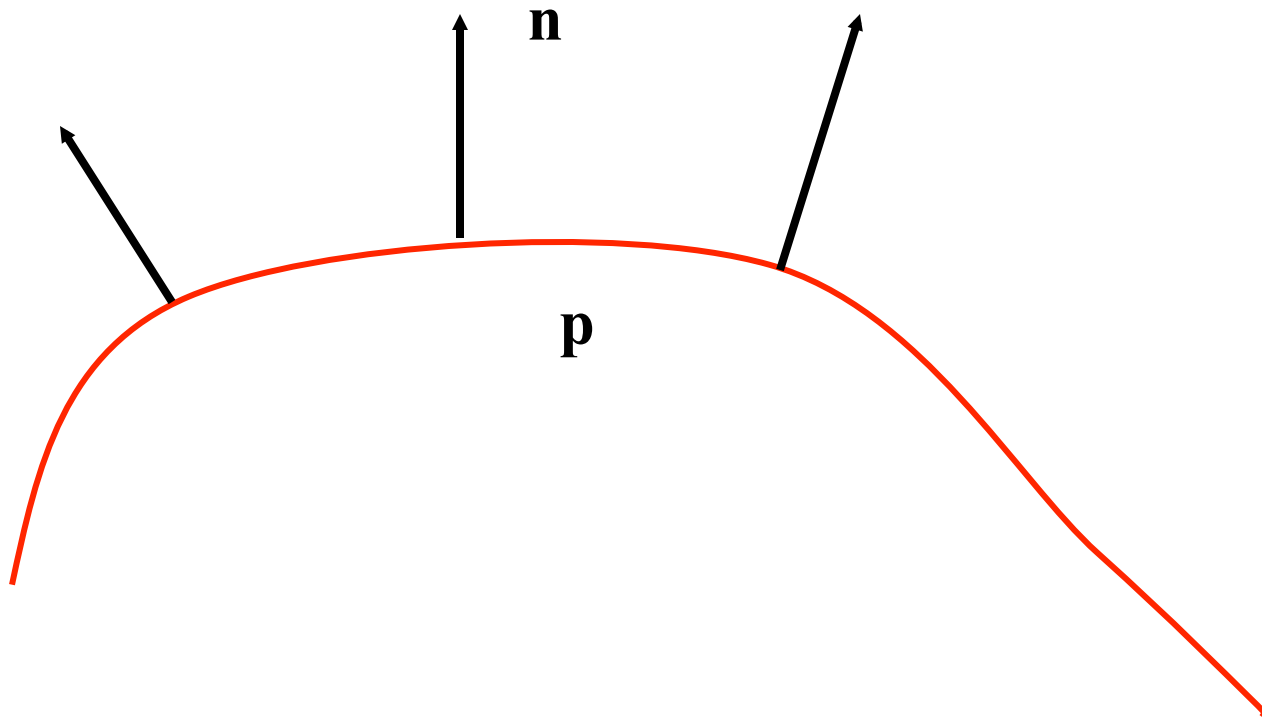
The Orange

- Texture map a photo of an orange onto a surface
 - Captures dimples
 - Will not be correct if we move viewer or light
 - We have shades of dimples rather than their correct orientation
- Ideally perturb normal across surface of object and compute a new color at each interior point

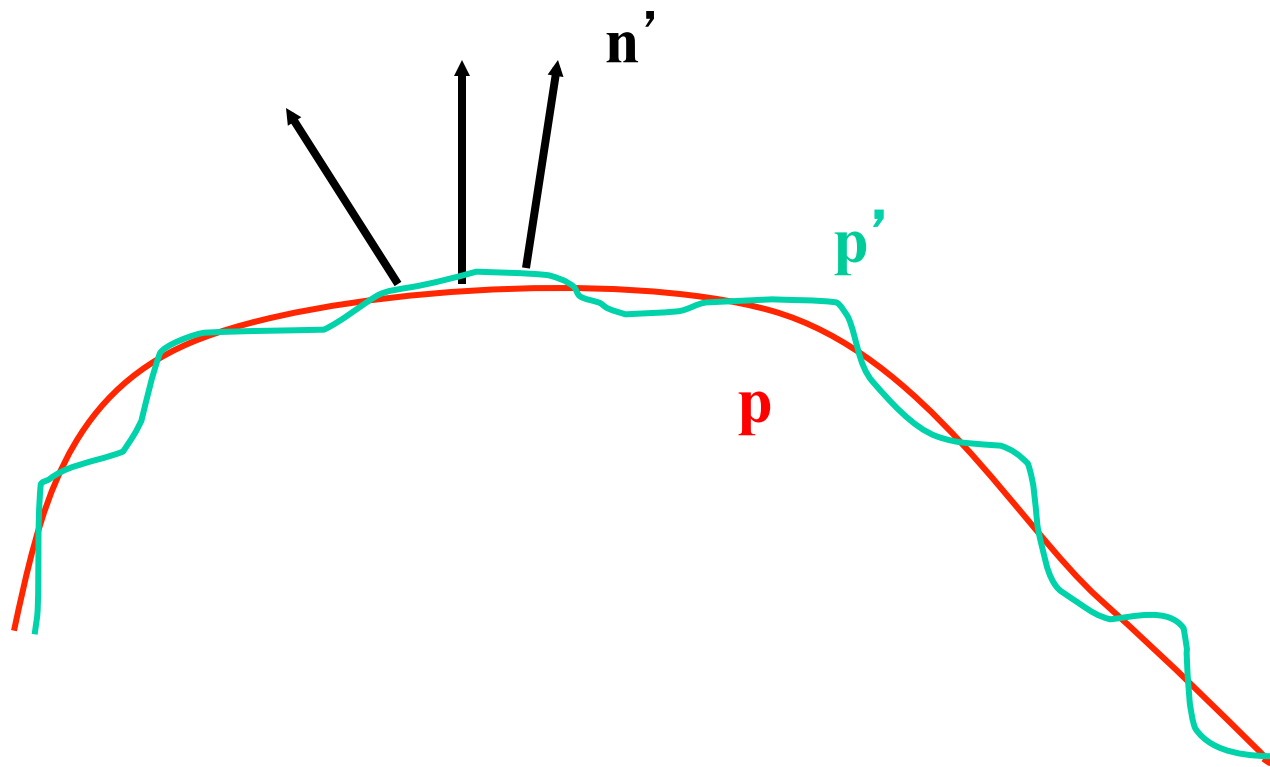


Bump Mapping (Blinn)

Consider a smooth surface



Rougher Version



Equations

$$\mathbf{p}(u,v) = [x(u,v), y(u,v), z(u,v)]^T$$

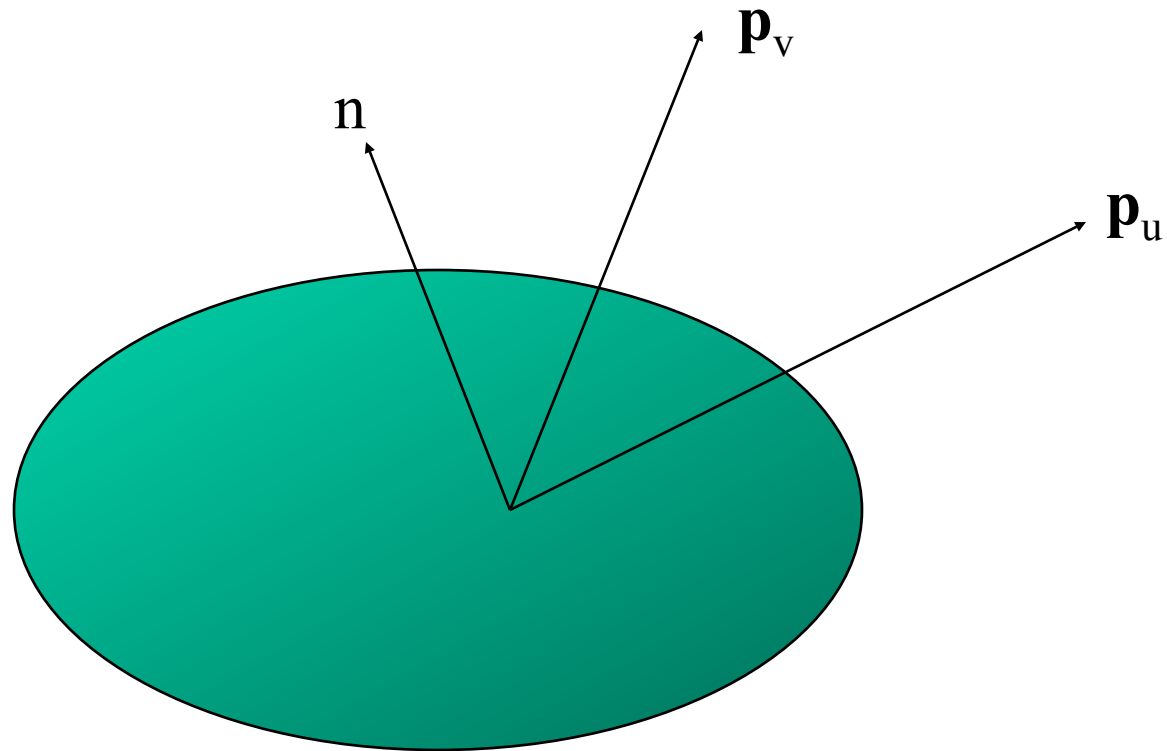
$$\mathbf{p}_u = [\partial x / \partial u, \partial y / \partial u, \partial z / \partial u]^T$$

$$\mathbf{p}_v = [\partial x / \partial v, \partial y / \partial v, \partial z / \partial v]^T$$

$$\mathbf{n} = (\mathbf{p}_u \times \mathbf{p}_v) / |\mathbf{p}_u \times \mathbf{p}_v|$$



Tangent Plane



Displacement Function

$$\mathbf{p}' = \mathbf{p} + d(u,v) \mathbf{n}$$

$d(u,v)$ is the bump or displacement function

$$|d(u,v)| \ll 1$$



Perturbed Normal

$$\mathbf{n}' = \mathbf{p}'_u \times \mathbf{p}'_v$$

$$\mathbf{p}'_u = \mathbf{p}_u + \left(\frac{\partial d}{\partial u} \right) \mathbf{n} + d(u,v) \mathbf{n}_u$$

$$\mathbf{p}'_v = \mathbf{p}_v + \left(\frac{\partial d}{\partial v} \right) \mathbf{n} + d(u,v) \mathbf{n}_v$$

If d is small, we can neglect last term



Approximating the Normal

$$\mathbf{n}' = \mathbf{p}'_u \times \mathbf{p}'_v$$

$$\approx \mathbf{n} + (\partial d / \partial u) \mathbf{n} \times \mathbf{p}_v + (\partial d / \partial v) \mathbf{n} \times \mathbf{p}_u$$

The vectors $\mathbf{n} \times \mathbf{p}_v$ and $\mathbf{n} \times \mathbf{p}_u$ lie
in the tangent plane

Hence the normal is displaced in the tangent plane

Must precompute the arrays $\partial d / \partial u$ and $\partial d / \partial v$

Finally, we perturb the normal during shading



Compositing & Blending



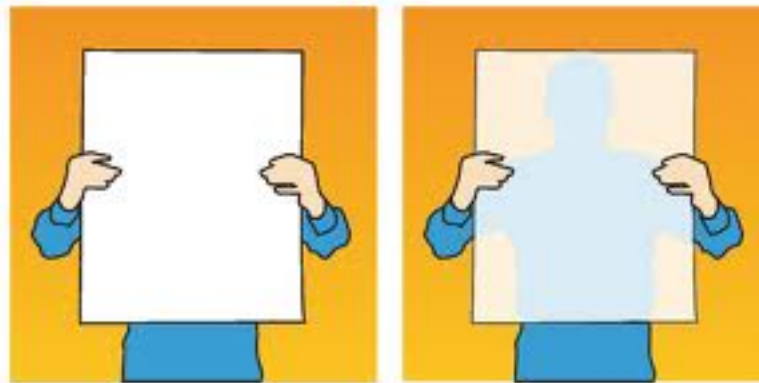
102

DEPARTMENT OF
COMPUTER SCIENCE
AND ENGINEERING

A

- Blending for translucent surfaces
- Compositing images
- Antialiasing

A



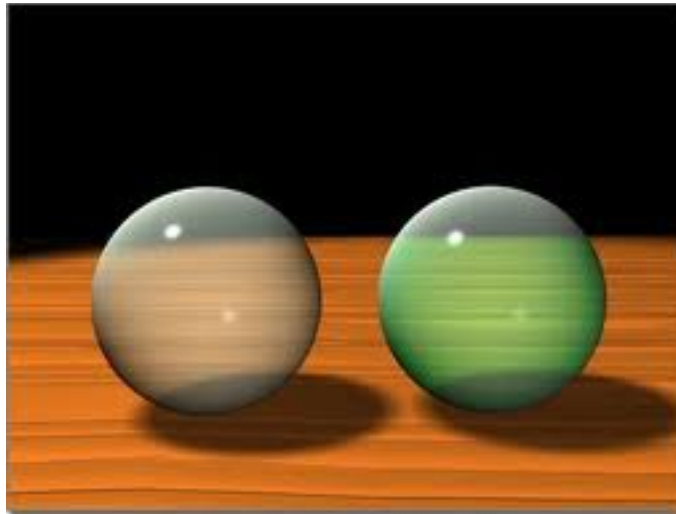
High opacity

Low opacity



DEPARTMENT OF
COMPUTER SCIENCE
AND ENGINEERING

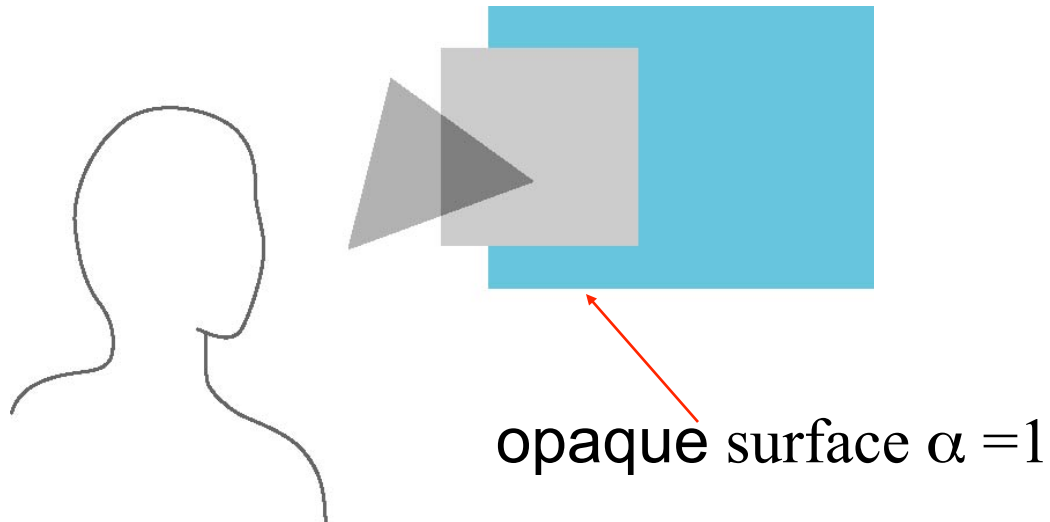
A



DEPARTMENT OF
COMPUTER SCIENCE
AND ENGINEERING

A

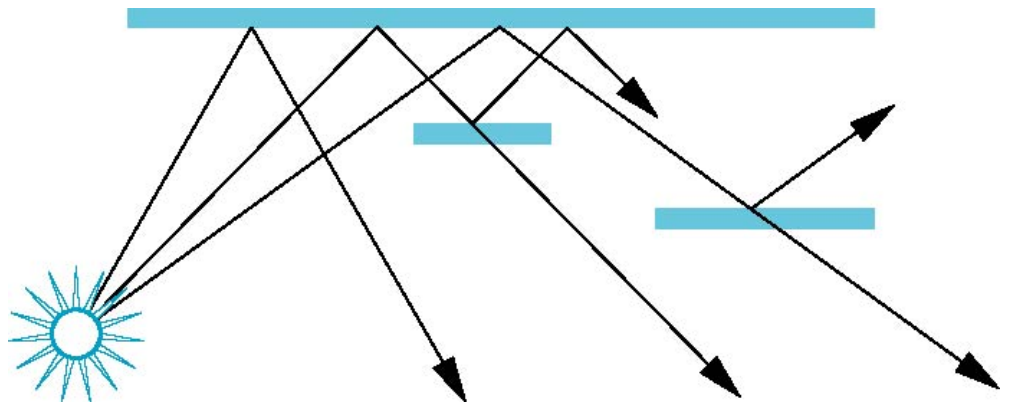
- Opaque surfaces permit no light to pass through
 - Transparent surfaces permit all light to pass
 - Translucent surfaces pass some light
- translucency = $1 - \text{opacity } (\alpha)$



Physical Models

Translucency in a physically correct manner is difficult

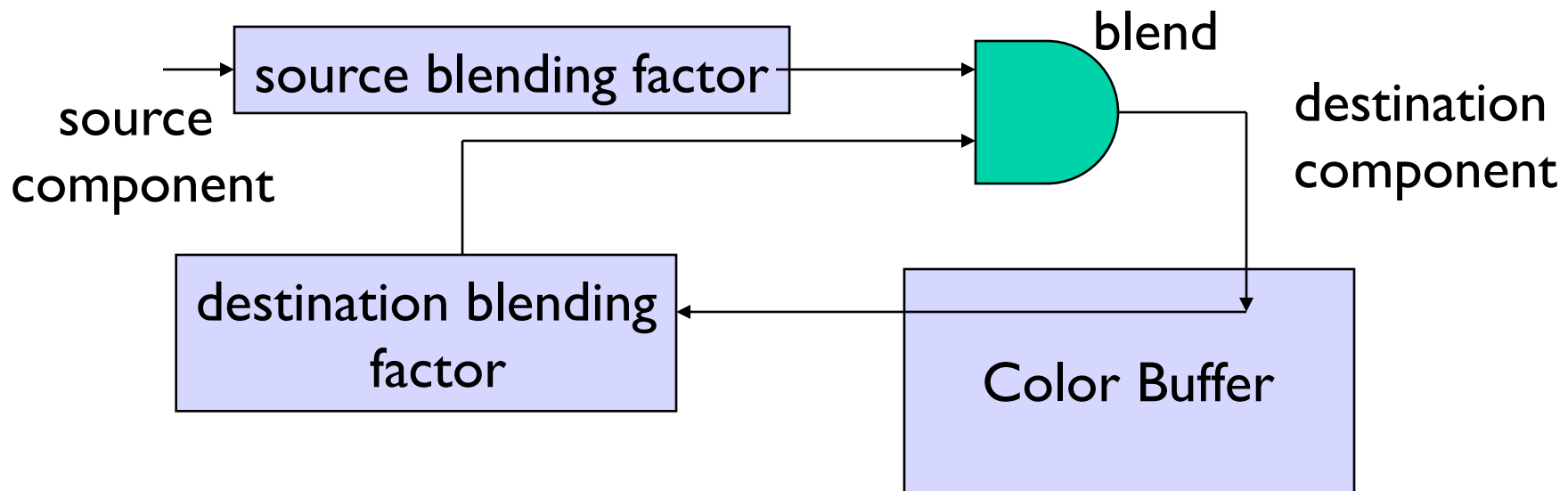
- the complexity of the internal interactions of light and matter
- Using a pipeline renderer



Compositing Operation

Rendering Model

- Use A component of RGBA (or RGBa) color for opacity
- During rendering expand to use RGBA values

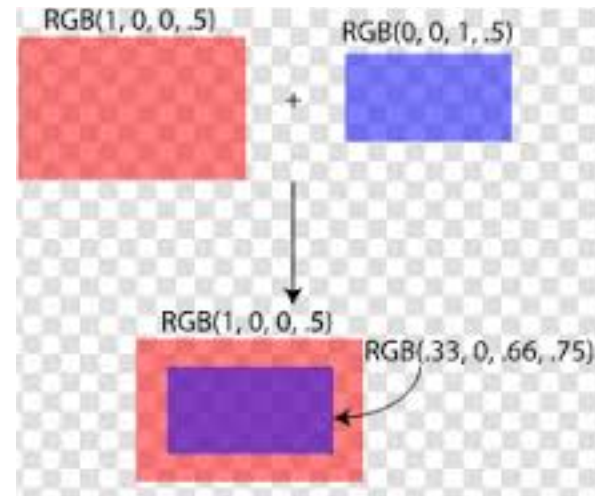


Examples



DEPARTMENT OF
COMPUTER SCIENCE
AND ENGINEERING

One Method



Blending Equation

We can define source and destination blending factors for each RGBA component

$$\mathbf{s} = [s_r, s_g, s_b, s_a]$$

$$\mathbf{d} = [d_r, d_g, d_b, d_a]$$

Suppose that the source and destination colors are

$$\mathbf{b} = [b_r, b_g, b_b, b_a]$$

$$\mathbf{c} = [c_r, c_g, c_b, c_a]$$

Blend as

$$\mathbf{c}' = [b_r s_r + c_r d_r, b_g s_g + c_g d_g, b_b s_b + c_b d_b, b_a s_a + c_a d_a]$$



OpenGL

Must enable blending and pick source and destination factors

glEnable(GL_BLEND)

**glBlendFunc(source_factor,
destination_factor)**

Only certain factors supported

GL_ZERO, GL_ONE

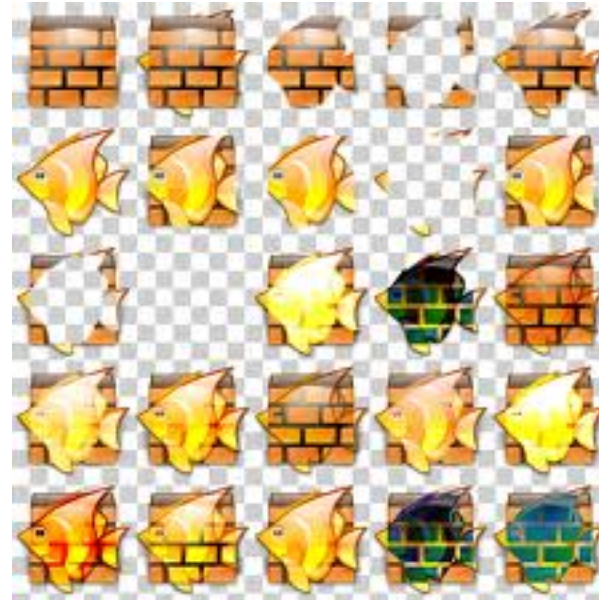
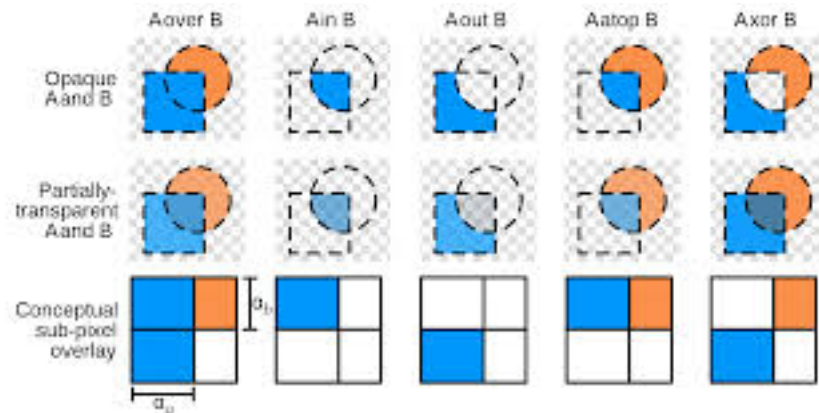
GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA

GL_DST_ALPHA, GL_ONE_MINUS_DST_ALPHA

See Redbook for complete list



Operator



DEPARTMENT OF
COMPUTER SCIENCE
AND ENGINEERING

Example

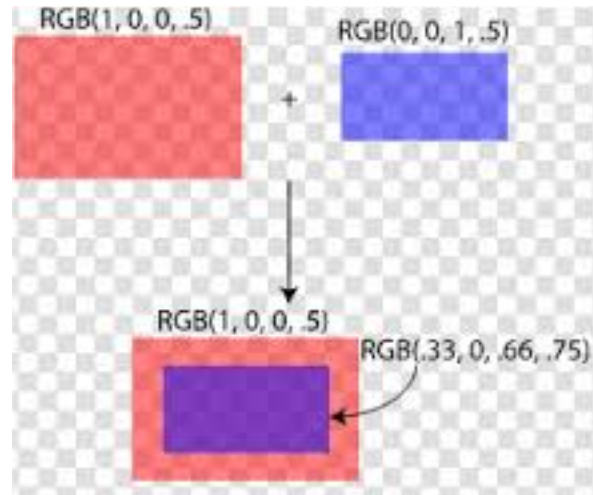
- Start with the opaque background color (R_0, G_0, B_0, I)
 - Initial destination color
- Blend in a translucent polygon with color (R_1, G_1, B_1, a_1)
- Select **GL_SRC_ALPHA** and **GL_ONE_MINUS_SRC_ALPHA** as the source and destination blending factors

$$R'_1 = a_1 R_1 + (1 - a_1) R_0, \dots\dots$$

- Note this formula is correct if polygon is either opaque or transparent



Works Here Too...



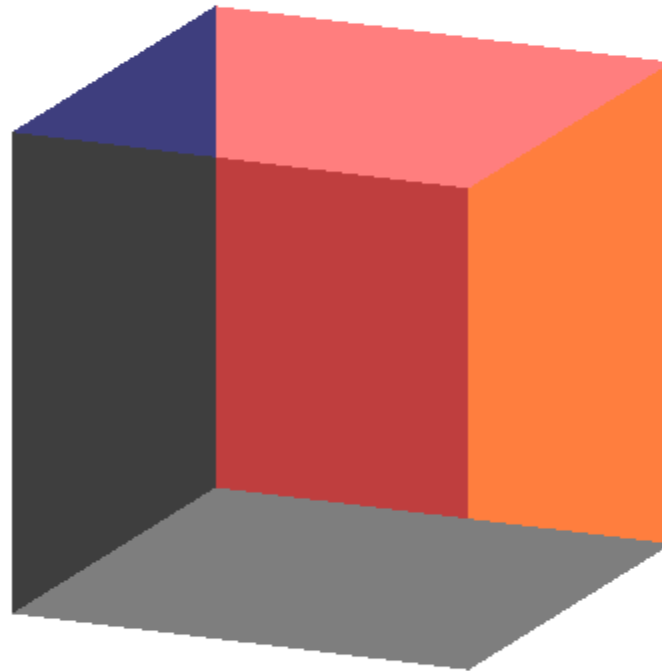
DEPARTMENT OF
COMPUTER SCIENCE
AND ENGINEERING

Clamping and Accuracy

- All RGBA are clamped to the range (0,1)
- RGBA values 8 bits !
 - Loose accuracy after much components together
 - Example: add together n images
 - Divide all color components by n to avoid clamping
 - Blend with source factor = 1, destination factor = 1
 - But division by n loses bits

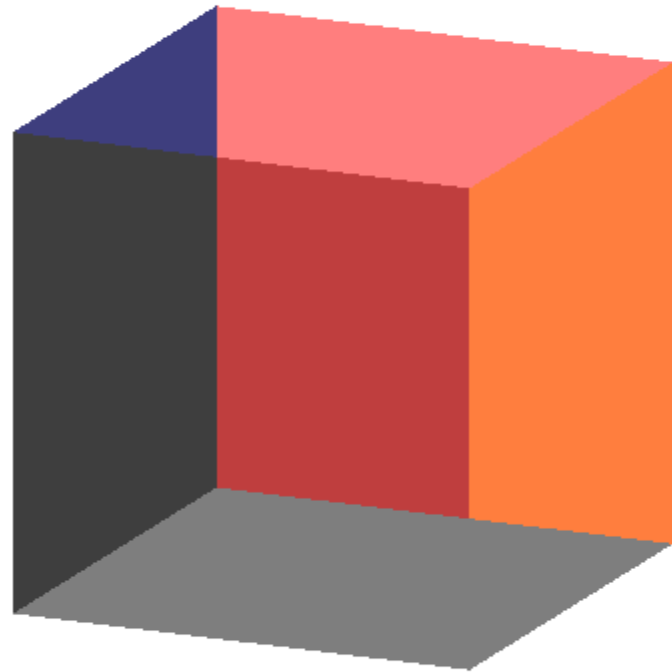


Order Dependency



Order Dependency

- Is this image correct?
 - Probably not
 - Polygons are rendered in the order they pass down the pipeline
 - Blending functions are order dependent



HSR with A

- Polygons which are opaque & translucent
- Opaque polygons block all polygons behind & affect depth buffer
- Translucent polygons should not affect depth buffer
 - Render with **glDepthMask(GL_FALSE)** which makes depth buffer read-only
- Sort polygons first to remove order dependency



Fog

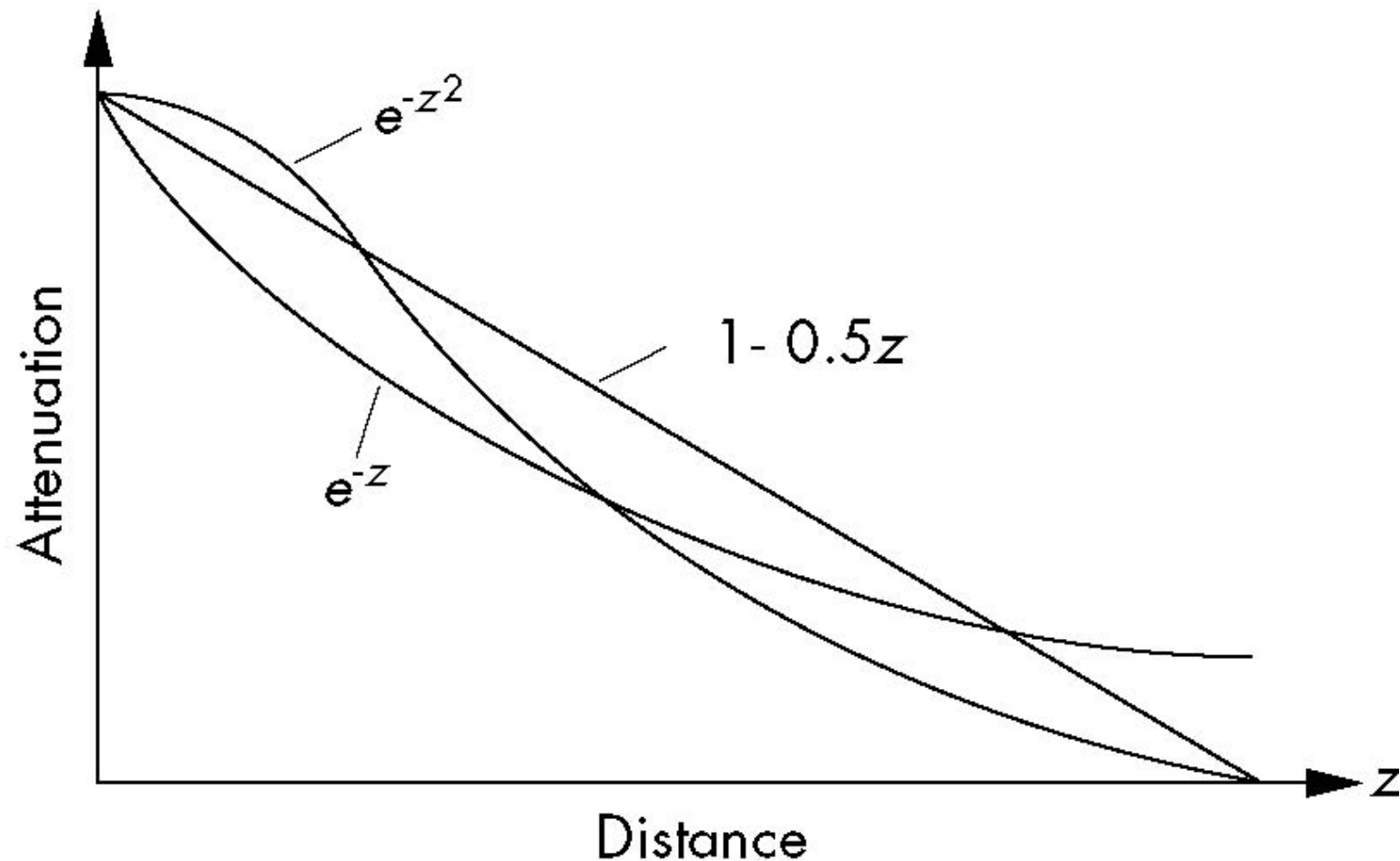


Simulate Fog

- Composite with fixed color and have blending factors depend on depth
 - Simulates a fog effect
- Blend source color C_s and fog color C_f by
$$C_s' = f C_s + (1-f) C_f$$
- f is the *fog factor*
 - Exponential
 - Gaussian
 - Linear (depth cueing)



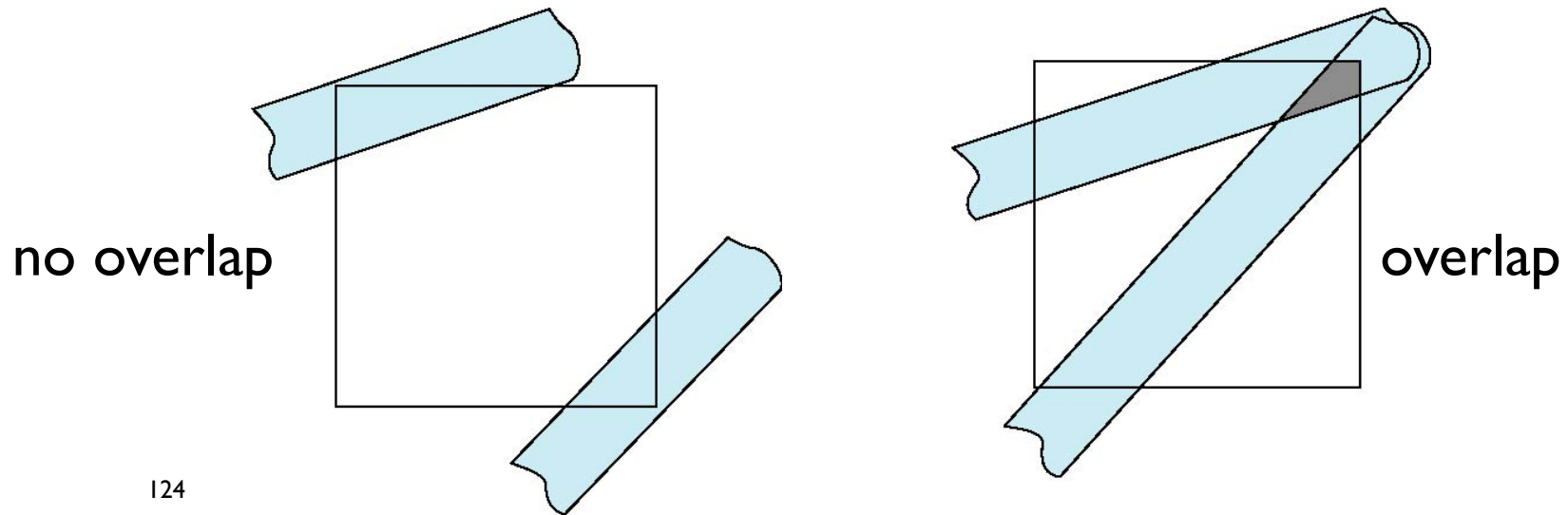
F - Fog Functions



Antialiasing

Color a pixel by adding fraction of color to frame buffer

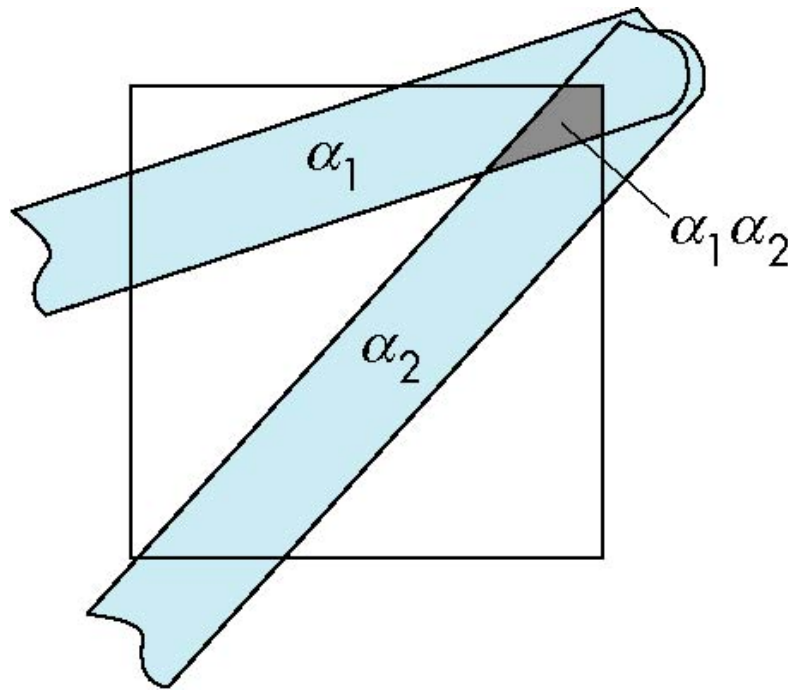
- Fraction depends on percentage of pixel covered by fragment
- Fraction depends on whether there is overlap



124

Area Averaging

Use average area $a_1 + a_2 - a_1 a_2$ as blending factor



OpenGL Antialiasing

Enable separately for points, lines, or polygons

```
glEnable(GL_POINT_SMOOTH);
```

```
glEnable(GL_LINE_SMOOTH);
```

```
glEnable(GL_POLYGON_SMOOTH);
```

```
glEnable(GL_BLEND);
```

```
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```



Accumulation

- Compositing/blending limited by resolution of frame buffer
 - Typically 8 bits per color component
- *Accumulation buffer* was a high resolution buffer (16 or more bits per component) that avoided this problem
- Could write into it or read from it with a scale factor
- Slower than direct compositing into the frame buffer

Particle Systems



Many Uses

- Used to model
 - Natural phenomena
 - Clouds
 - Terrain
 - Plants
 - Crowd Scenes
 - Real physical processes

Newtonian Particle

- Particle system is a set of particles
- Each particle is an ideal point mass
- Six degrees of freedom
 - Position
 - Velocity
- Each particle obeys Newtons' law

$$f = ma$$



Particle Equations

$$\mathbf{p}_i = (x_i, y_i, z_i)$$

$$\mathbf{v}_i = d\mathbf{p}_i / dt = \mathbf{p}_i' = (dx_i / dt, dy_i / dt, z_i / dt)$$

$$m \mathbf{v}_i' = \mathbf{f}_i$$

Hard part is defining force vector



Force Vector

- Independent Particles
 - Gravity
 - Wind forces
 - $O(n)$ calculation
- Coupled Particles $O(n)$
 - Meshes
 - Spring-Mass Systems
- Coupled Particles $O(n^2)$
 - Attractive and repulsive forces

Solution of Particle Systems

```
float time, delta state[6n], force[3n];  
state = initial_state();  
for(time = t0; time<final_time, time+=delta) {  
    force = force_function(state, time);  
    state = ode(force, state, time, delta);  
    render(state, time)  
}
```



Simple Forces

- Consider force on particle i

$$\mathbf{f}_i = \mathbf{f}_i(\mathbf{p}_i, \mathbf{v}_i)$$

- Gravity $\mathbf{f}_i = \mathbf{g}$

$$\mathbf{g} = (0, -g, 0)$$

- Wind forces
- Drag

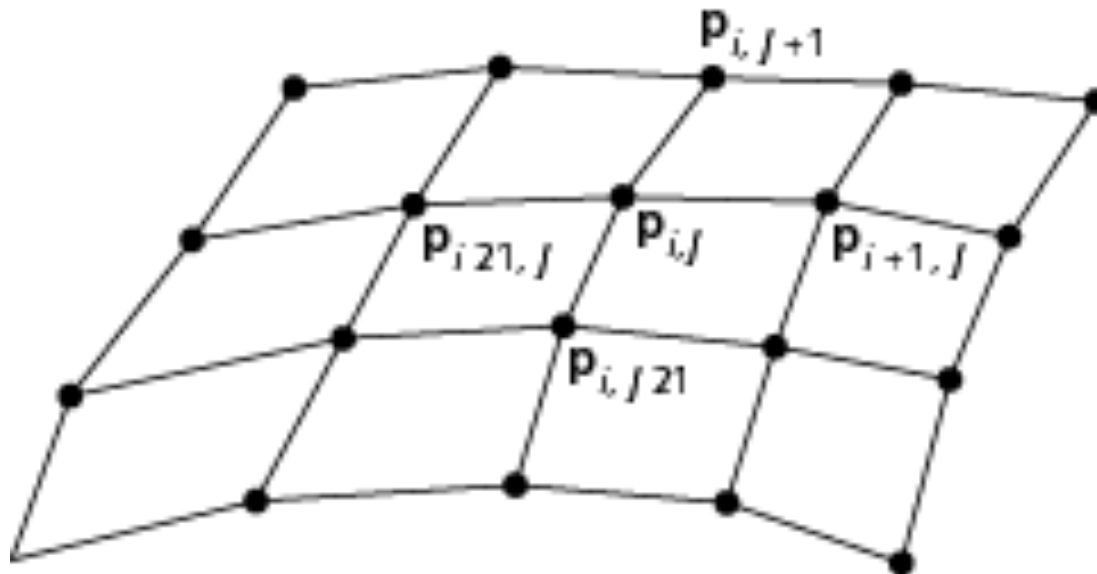


$\mathbf{p}_i(t_0), \mathbf{v}_i(t_0)$



Meshes

- Connect each particle to its closest neighbors
 - $O(n)$ force calculation
- Use spring-mass system



Spring Forces

- Assume each particle has unit mass and is connected to its neighbor(s) by a spring
- Hooke's law: force proportional to distance ($d = \|\mathbf{p} - \mathbf{q}\|$) between the points



Hooke's Law

Let s be the distance when there is no force

$$\mathbf{f} = -k_s(|\mathbf{d}| - s) \mathbf{d}/|\mathbf{d}|$$

k_s is the spring constant

$\mathbf{d}/|\mathbf{d}|$ is a unit vector pointed from \mathbf{p} to \mathbf{q}

Each interior point in mesh has four forces applied to it



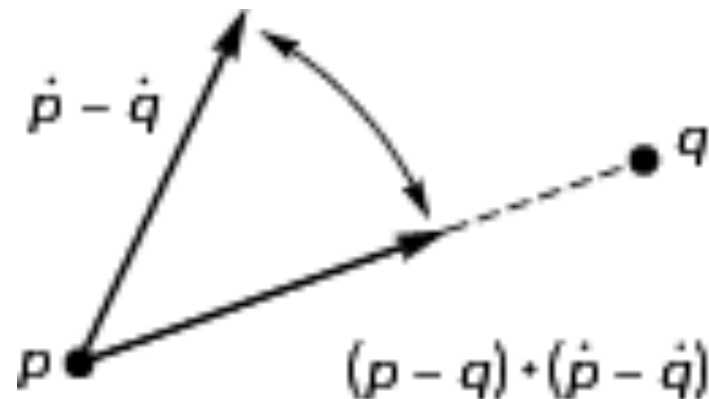
DEPARTMENT OF
COMPUTER SCIENCE
AND ENGINEERING

Spring Damping

- A pure spring-mass will oscillate forever
- Must add a damping term

$$\mathbf{f} = -(k_s(|\mathbf{d}| - s) + k_d \mathbf{d} \cdot \dot{\mathbf{d}} / |\mathbf{d}|) \mathbf{d} / |\mathbf{d}|$$

- Must project velocity



Attraction and Repulsion

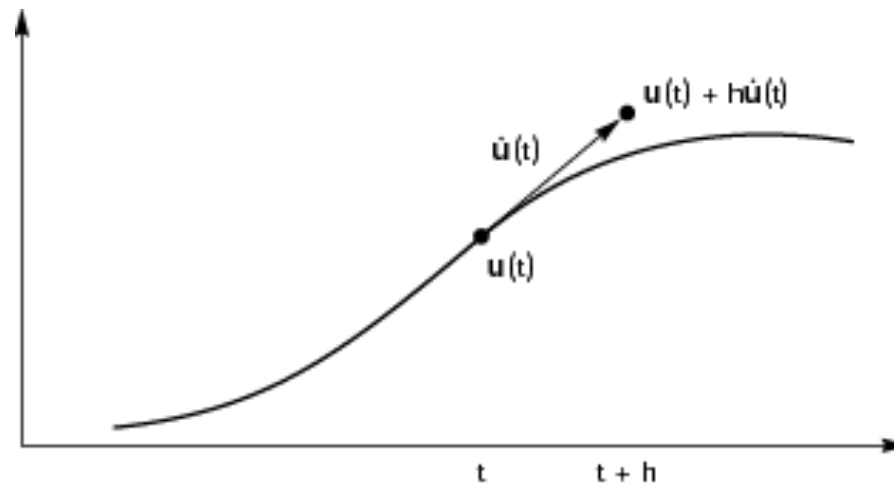
- Inverse square law

$$\mathbf{f} = -k_r \mathbf{d} / |\mathbf{d}|^3$$

- General case requires $O(n^2)$ calculation
- In most problems, the drop off is such that not many particles contribute to the forces on any given particle
- Sorting problem: is it $O(n \log n)$?

Solution of ODEs

- Particle system has $6n$ ordinary differential equations
- Write set as $d\mathbf{u}/dt = g(\mathbf{u}, t)$
- Solve by approximations using Taylor's Thm



Euler's Method

$$\mathbf{u}(t + h) \approx \mathbf{u}(t) + h \, d\mathbf{u}/dt = \mathbf{u}(t) + h\mathbf{g}(\mathbf{u}, t)$$

Per step error is $O(h^2)$

Require one force evaluation per time step

Problem is numerical instability
depends on step size



Improved Euler

$$\mathbf{u}(t + h) \approx \mathbf{u}(t) + h/2(\mathbf{g}(\mathbf{u}, t) + \mathbf{g}(\mathbf{u}, t+h))$$

Per step error is $O(h^3)$

Also allows for larger step sizes

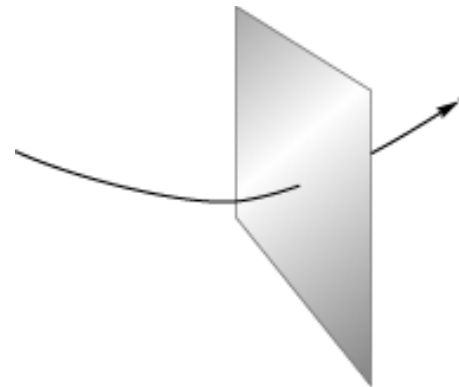
But requires two function evaluations per step

Also known as Runge-Kutta method of order 2



Constraints

- Easy in computer graphics to ignore physical reality
- Surfaces are virtual
- Must detect collisions separately if we want exact solution
- Can approximate with repulsive forces



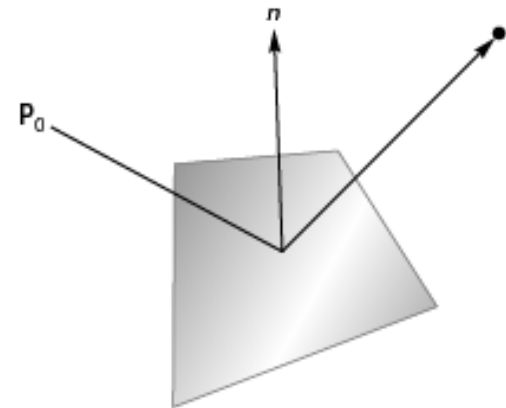
Collisions

Once we detect a collision, we can calculate
new path

Use coefficient of resitution

Reflect vertical component

May have to use partial time step

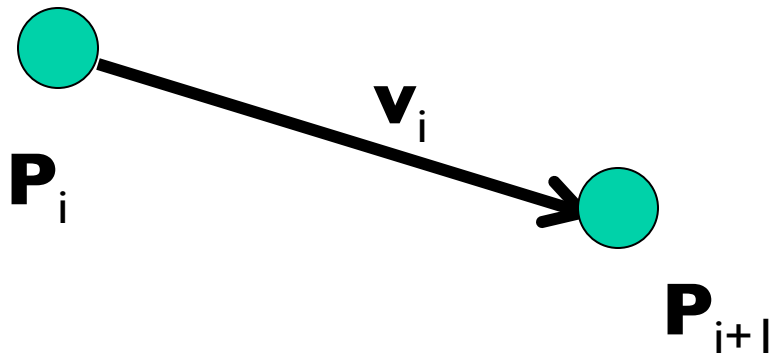


Example

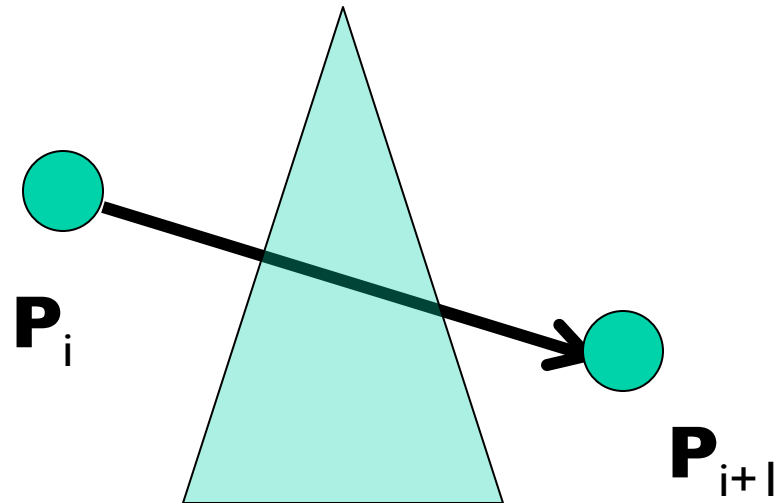
$$\mathbf{p}_i = (x_i, y_i, z_i)$$

$$\mathbf{v}_i = d\mathbf{p}_i / dt = \mathbf{p}_i' = (dx_i / dt, dy_i / dt, dz_i / dt)$$

$$m \mathbf{v}_i' = \mathbf{f}_i$$



Collision ?



DEPARTMENT OF
COMPUTER SCIENCE
AND ENGINEERING

Problem:

Triangle & Ray Distinct Objects

Ray/Triangle Intersection

Fast, Minimum Storage Ray/Triangle Intersection

Tomas Möller

Prosolvia Clarus AB

Chalmers University of Technology

E-mail: tompa@clarus.se

Ben Trumbore

Program of Computer Graphics

Cornell University

E-mail: wbt@graphics.cornell.edu



DEPARTMENT OF
COMPUTER SCIENCE
AND ENGINEERING

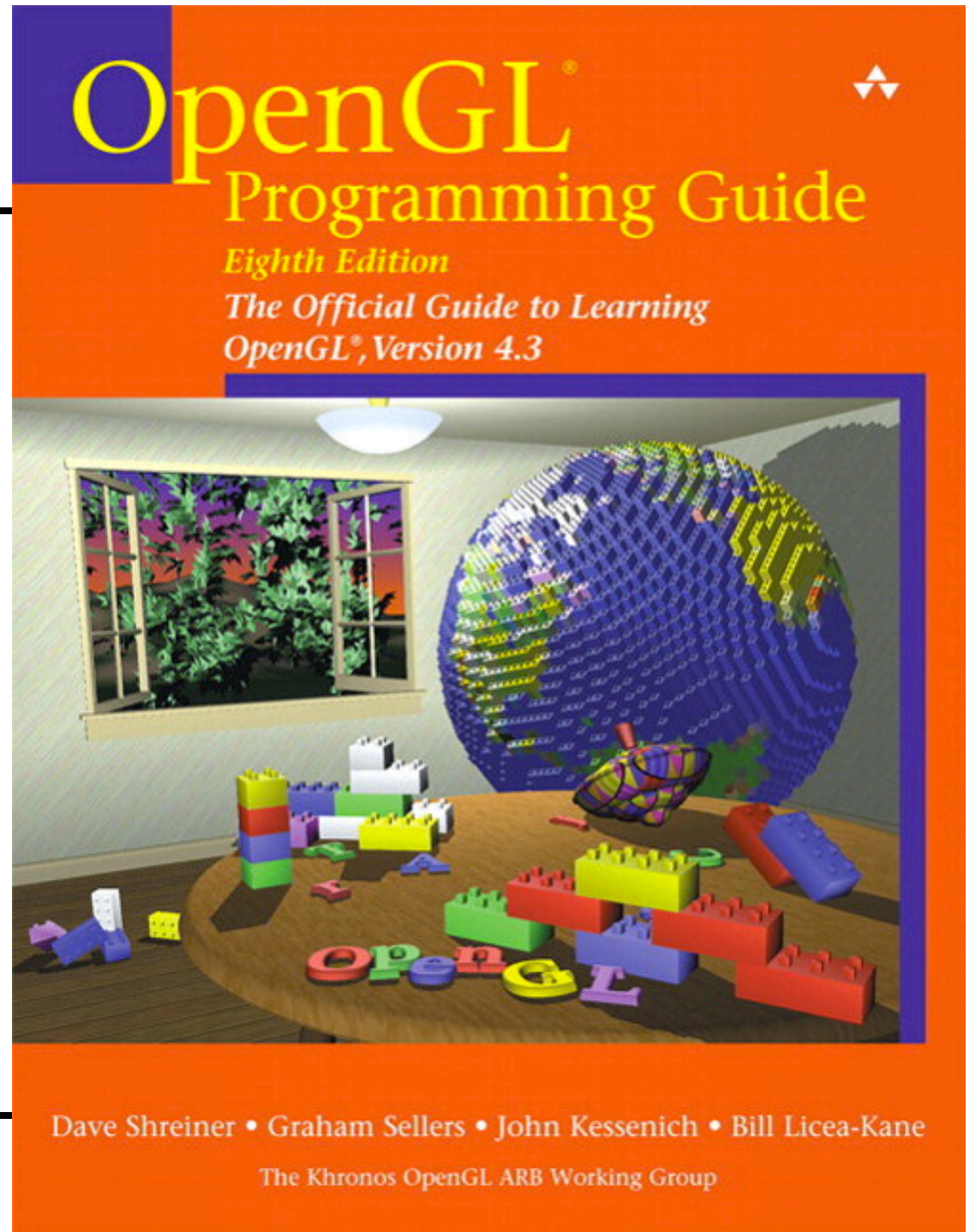
Advanced Features of GLSL

TF - Transform Feedback

TBO – Texture Buffer Object

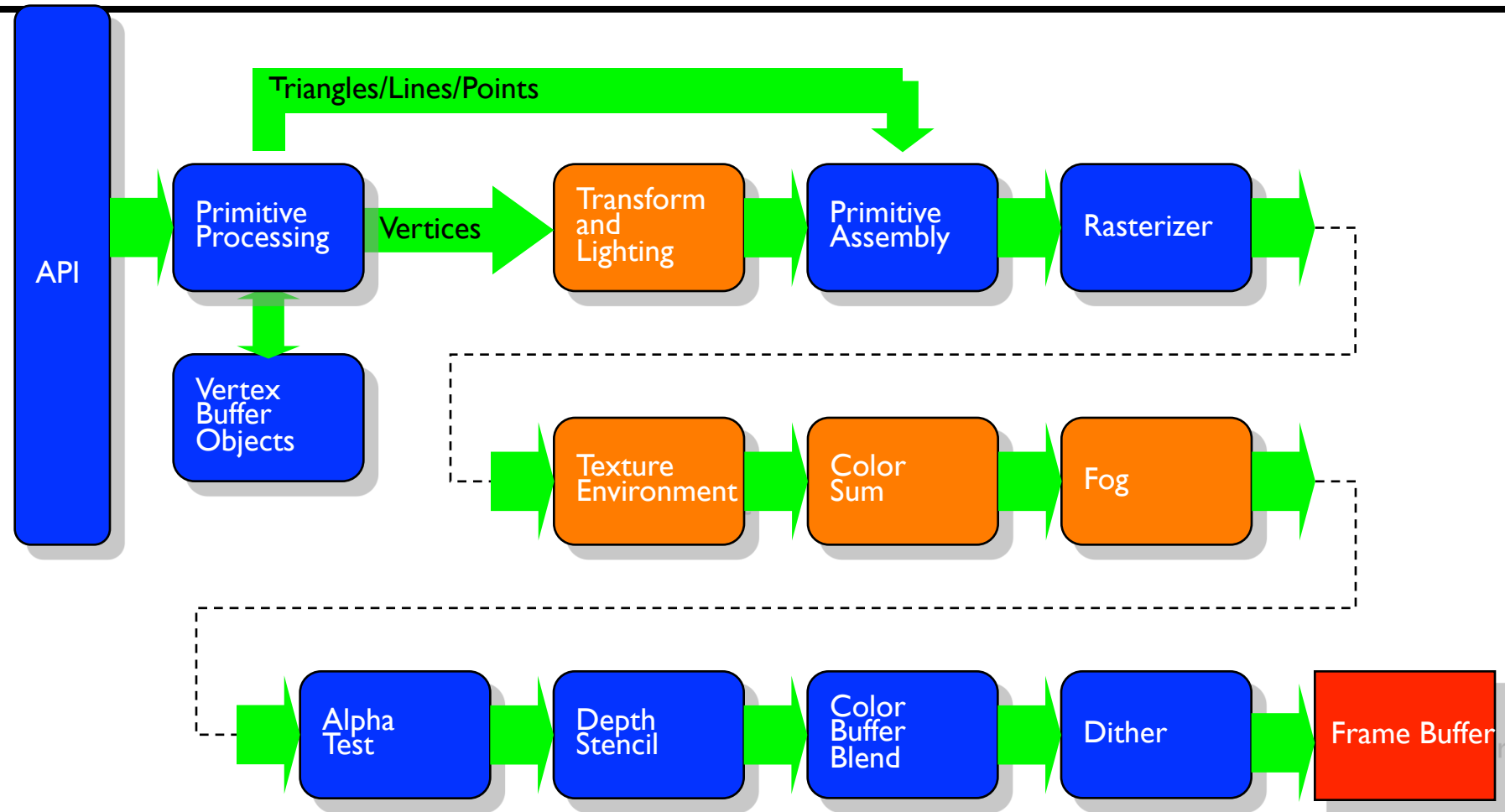


Chapter 5

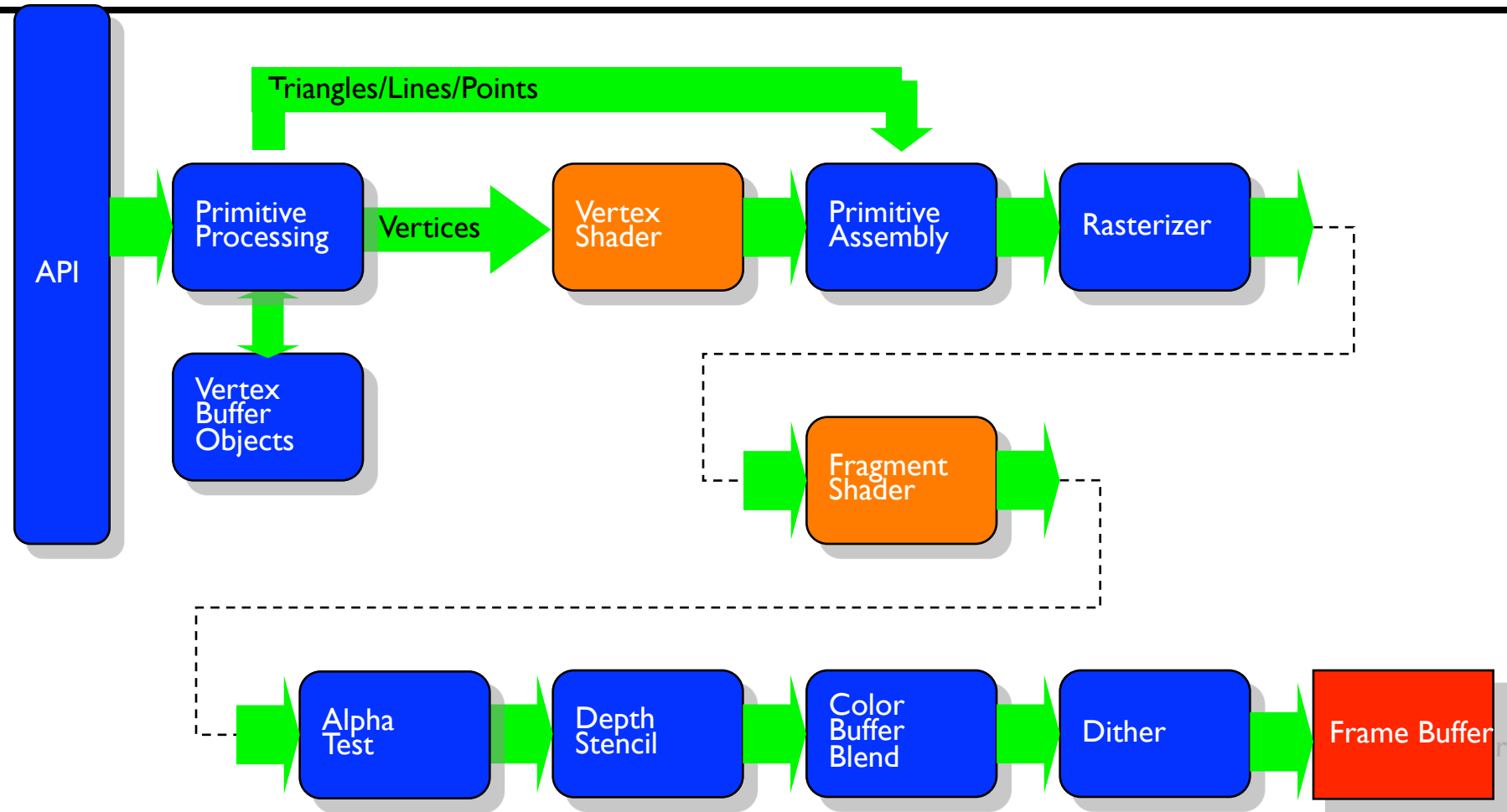


DEPARTMENT OF
COMPUTER SCIENCE
AND ENGINEERING

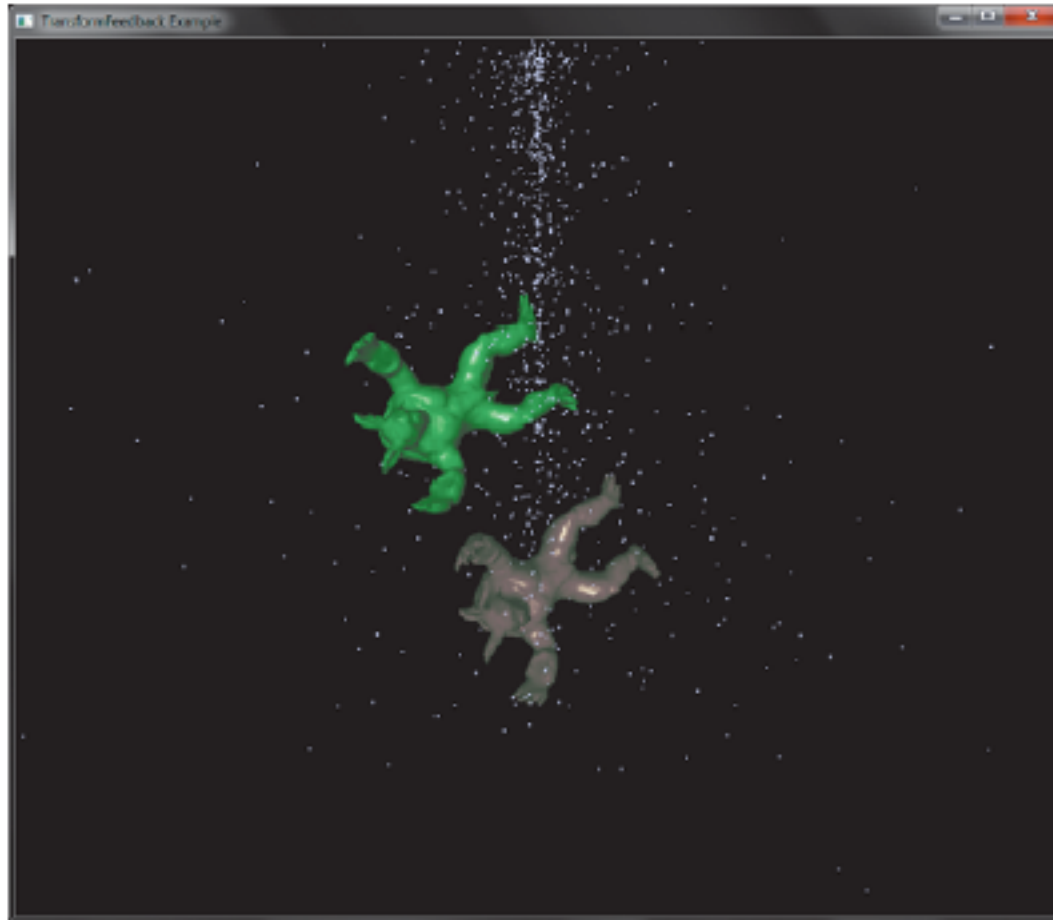
Fixed Functionality Pipeline



Programmable Shader Pipeline



Back2Particles



DEPARTMENT OF
COMPUTER SCIENCE
AND ENGINEERING

Schema

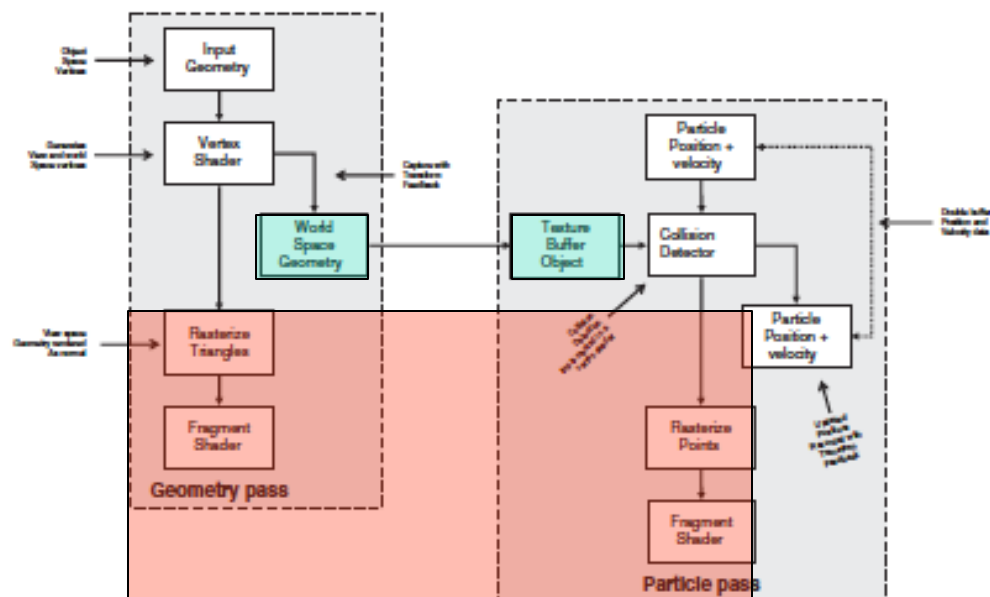


Figure 5.19 Schematic of the particle system simulator

Geometry Pass

Example 5.8 Vertex Shader Used in Geometry Pass of Particle System Simulator

```
#version 420 core

uniform mat4 model_matrix;
uniform mat4 projection_matrix;

layout (location = 0) in vec4 position;
layout (location = 1) in vec3 normal;

out vec4 world_space_position;
out vec3 vs_fs_normal;

void main(void)
{
    vec4 pos = (model_matrix * (position * vec4(1.0, 1.0, 1.0, 1.0)));
    world_space_position = pos;
    vs_fs_normal = normalize((model_matrix * vec4(normal, 0.0)).xyz);
    gl_Position = projection_matrix * pos;
};
```



Storing Geometry

Example 5.9 Configuring the Geometry Pass of the Particle System Simulator

```
static const char * varyings2[] =
{
    "world_space_position"
};

glTransformFeedbackVaryings(render_prog, 1, varyings2,
                             GL_INTERLEAVED_ATTRIBS);
glLinkProgram(render_prog);
```

TBO writing



DEPARTMENT OF
COMPUTER SCIENCE
AND ENGINEERING

Transform Feedback

```
void glTransformFeedbackVaryings(GLuint program,  
                                GLsizei count,  
                                const GLchar ** varyings,  
                                GLenum bufferMode);
```

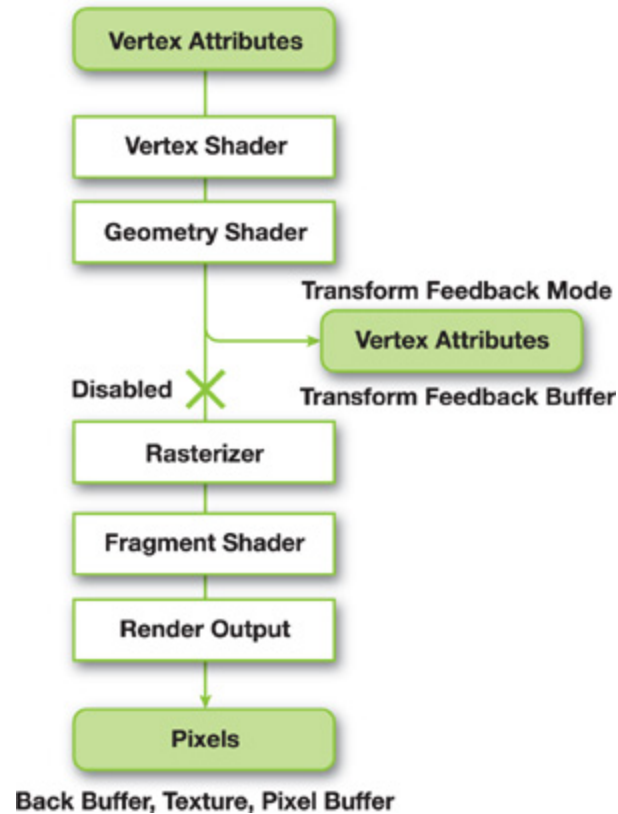
Sets the varyings to be recorded by transform feedback for the program specified by *program*. *count* specifies the number of strings contained in the array *varyings*, which contains the names of the varyings to be captured. *bufferMode* is the mode in which the varyings will be captured—either separate mode (specified by GL_SEPARATE_ATTRIBS) or interleaved mode (specified by GL_INTERLEAVED_ATTRIBS).



Transform feedback?

RedBook says: “Transform Feedback is the process of altering the rendering pipeline so that primitives processed by a Vertex Shader and optionally a Geometry Shader will be written to buffer objects. This allows one to preserve the post-transform rendering state of an object and resubmit this data multiple times.”

Transform Feedback diagram



Absence of Transform Feedback

To update Vertex Buffer Object's attributes:

1. OpenGL copies VBO from GPU memory to CPU memory
2. Update in CPU and send back
3. Consumes time and bandwidth

Role of TF

1. All computations are now conducted in GPU
2. A special buffer after shaders and send transformations

CPU not needed and little application involvement

Transform Feedback Examples

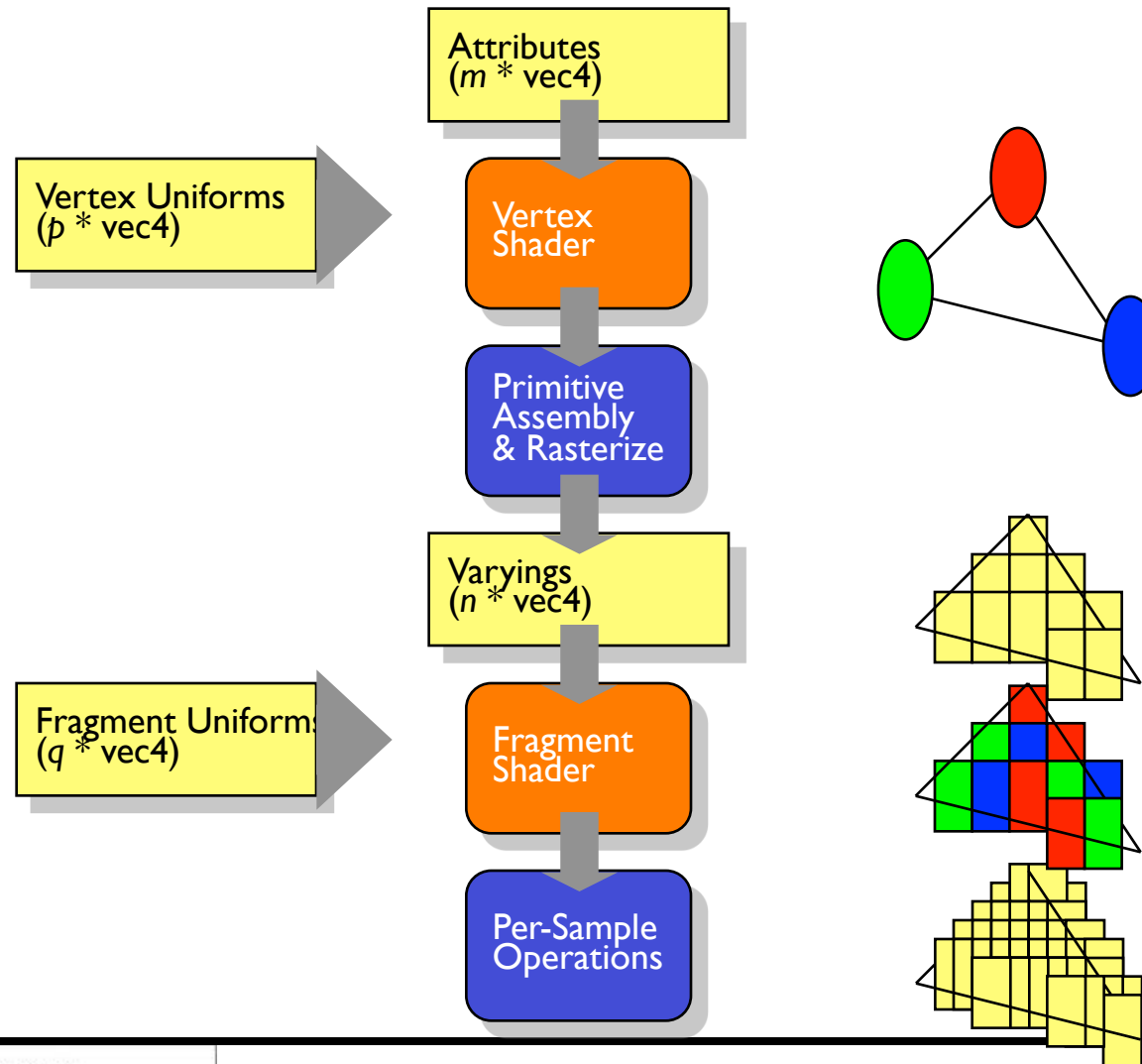
<http://www.youtube.com/watch?v=SiCq8ETTqRk>

- Uses TF to render a particle smoke system with fire spreading

<http://www.youtube.com/watch?v=E636tYOxoVI>

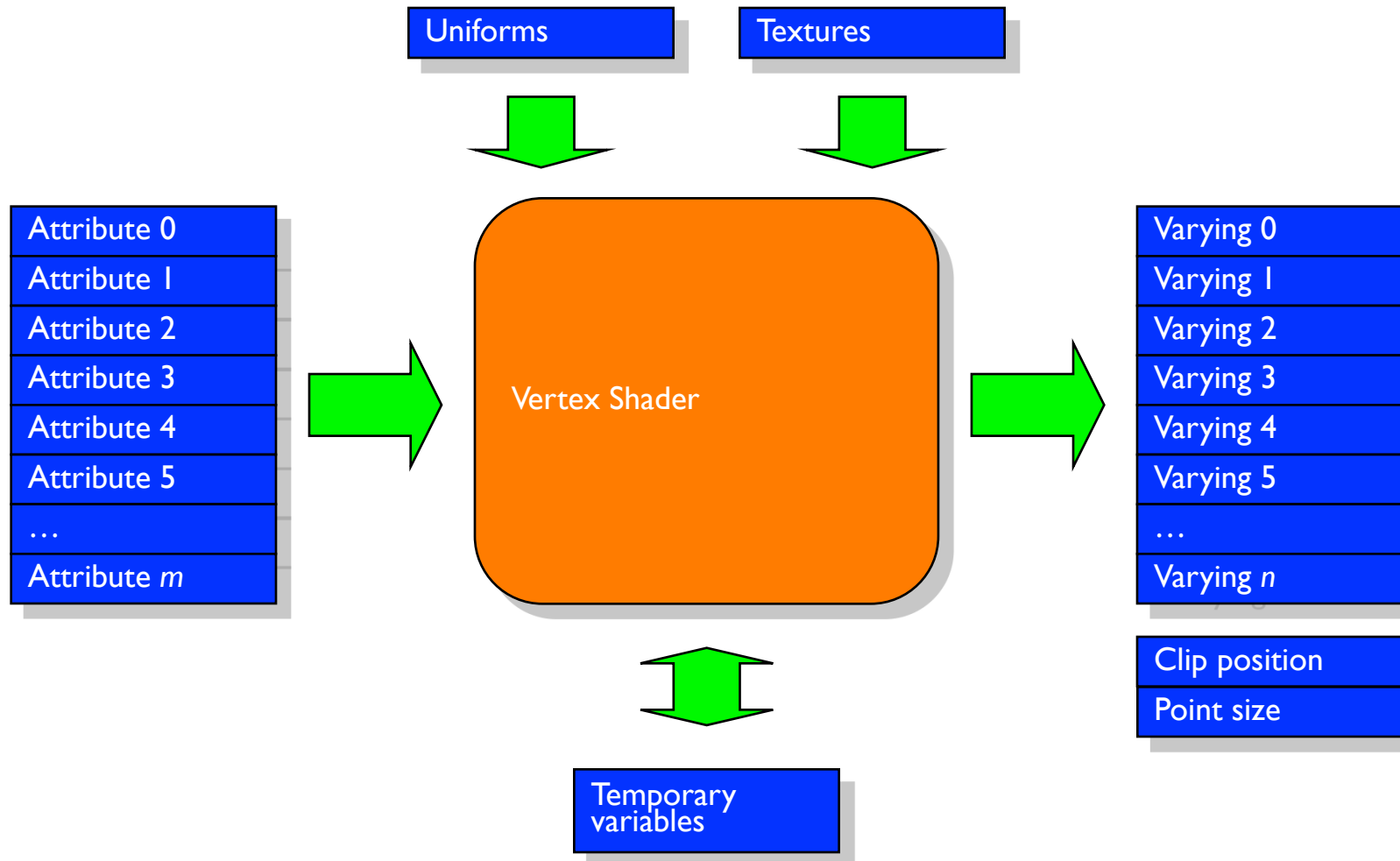
Attain good performance can be by using TF. It controls all of the particles in this on the GPU.

Programmer's Model

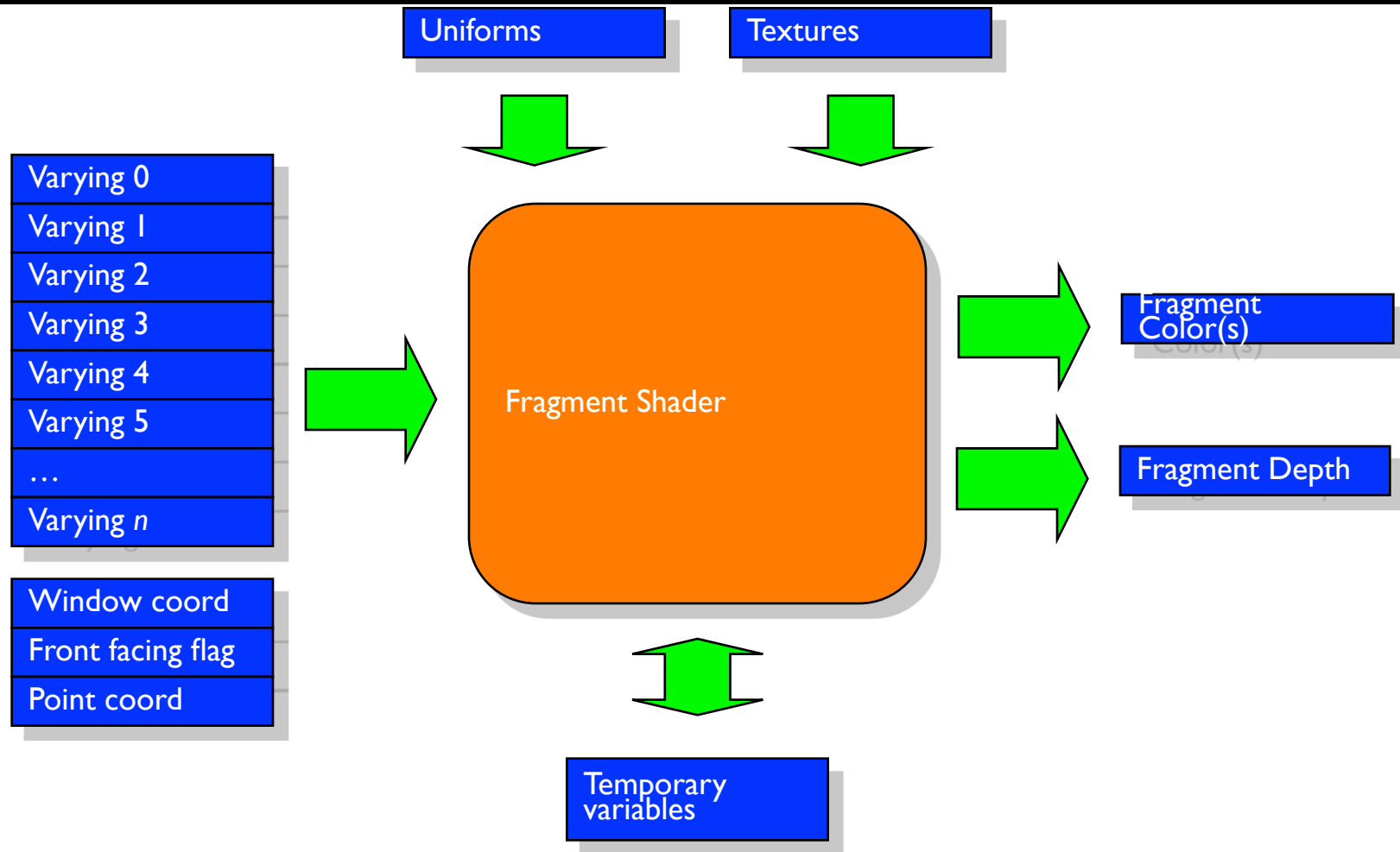


DEPARTMENT OF
COMPUTER SCIENCE
AND ENGINEERING

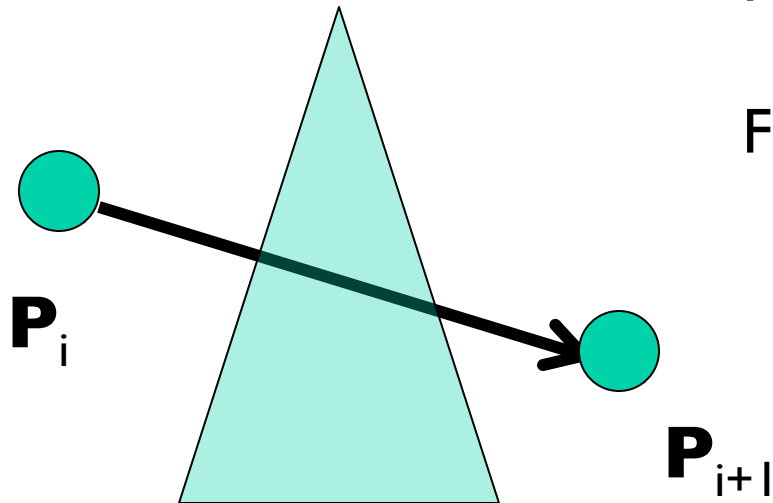
Vertex Shader Environment



Fragment Shader Environment



Collision Detection



Find intersection of ray with plane

Find actual intersection



DEPARTMENT OF
COMPUTER SCIENCE
AND ENGINEERING

Ray/Triangle Intersection

Fast, Minimum Storage Ray/Triangle Intersection

Tomas Möller

Prosolvia Clarus AB

Chalmers University of Technology

E-mail: tompa@clarus.se

Ben Trumbore

Program of Computer Graphics

Cornell University

E-mail: wbt@graphics.cornell.edu



DEPARTMENT OF
COMPUTER SCIENCE
AND ENGINEERING

Some Math

A ray $R(t)$ with origin O and normalized direction D is defined as

$$R(t) = O + tD \quad (1)$$

A point, $T(u, v)$, on a triangle is given by

$$T(u, v) = (1 - u - v)V_0 + uV_1 + vV_2, \quad (2)$$



DEPARTMENT OF
COMPUTER SCIENCE
AND ENGINEERING

Some Math

interpolation, color interpolation etc. Computing the intersection between the ray, $R(t)$, and the triangle, $T(u, v)$, is equivalent to $R(t) = T(u, v)$, which yields:

$$O + tD = (1 - u - v)V_0 + uV_1 + vV_2 \quad (3)$$

Rearranging the terms gives:

$$\begin{bmatrix} -D, & V_1 - V_0, & V_2 - V_0 \end{bmatrix} \begin{bmatrix} t \\ u \\ v \end{bmatrix} = O - V_0 \quad (4)$$

This means the barycentric coordinates (u, v) and the distance, t , from the ray origin to the intersection point can be found by solving the linear system of equations above.

Fast Ray-Triangle Intersection

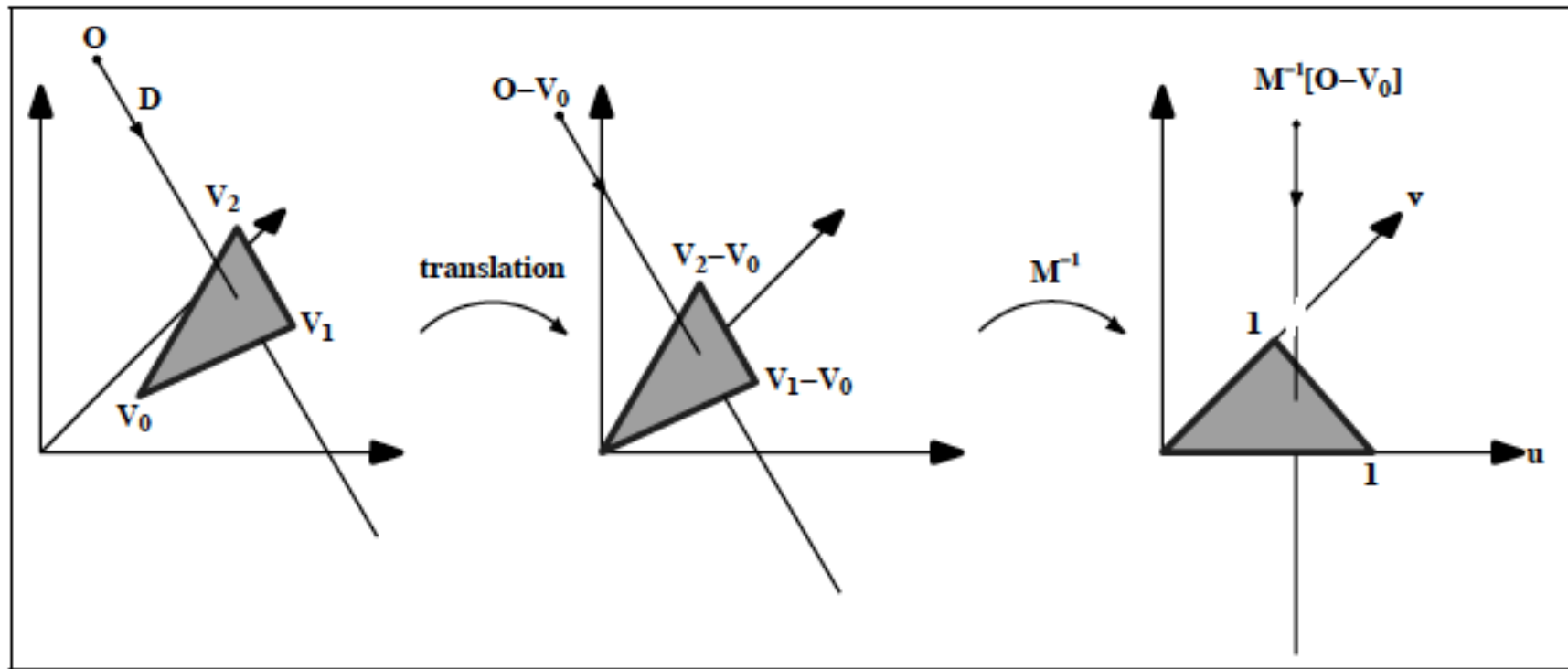


Figure 1: Translation and change of base of the ray origin.



Final Computations

$$\begin{bmatrix} t \\ u \\ v \end{bmatrix} = \frac{1}{(D \times E_2) \cdot E_1} \begin{bmatrix} (T \times E_1) \cdot E_2 \\ (D \times E_2) \cdot T \\ (T \times E_1) \cdot D \end{bmatrix} = \frac{1}{P \cdot E_1} \begin{bmatrix} Q \cdot E_2 \\ P \cdot T \\ Q \cdot D \end{bmatrix}, \quad (6)$$

where $P = (D \times E_2)$ and $Q = T \times E_1$. In our implementation we reuse these



Geometry Pass

Vertex Shader

Example 5.8 Vertex Shader Used in Geometry Pass of Particle System Simulator

```
#version 420 core

uniform mat4 model_matrix;
uniform mat4 projection_matrix;

layout (location = 0) in vec4 position;
layout (location = 1) in vec3 normal;

out vec4 world_space_position;

out vec3 vs_fs_normal;

void main(void)
{
    vec4 pos = (model_matrix * (position * vec4(1.0, 1.0, 1.0, 1.0)));
    world_space_position = pos;
    vs_fs_normal = normalize((model_matrix * vec4(normal, 0.0)).xyz);
    gl_Position = projection_matrix * pos;
};
```



Configuring Geometry Pass

Example 5.9 Configuring the Geometry Pass of the Particle System Simulator

```
static const char * varyings2[] =
{
    "world_space_position"
};

glTransformFeedbackVaryings(render_prog, 1, varyings2,
                             GL_INTERLEAVED_ATTRIBS);
glLinkProgram(render_prog);
```

TBO writing



DEPARTMENT OF
COMPUTER SCIENCE
AND ENGINEERING

Particle Pass

Example 5.10 Vertex Shader Used in Simulation Pass of Particle System Simulator

```
#version 420 core

uniform mat4 model_matrix;
uniform mat4 projection_matrix;
uniform int triangle_count;

layout (location = 0) in vec4 position;
layout (location = 1) in vec3 velocity;

out vec4 position_out;
out vec3 velocity_out;

uniform samplerBuffer geometry_tbo;
uniform float time_step = 0.02;

bool intersect(vec3 origin, vec3 direction, vec3 v0, vec3 v1, vec3 v2,
              out vec3 point)
{
    vec3 u, v, n;
    vec3 w0, w;
    float r, a, b;

    u = (v1 - v0);
    v = (v2 - v0);
    n = cross(u, v);

    w0 = origin - v0;
    a = -dot(n, w0);
    b = dot(n, direction);

    r = a / b;
    if (r < 0.0 || r > 1.0)
        return false;

    point = origin + r * direction;

    float uu, uv, vv, wu, wv, D;

    uu = dot(u, u);
    uv = dot(u, v);
    vv = dot(v, v);
    w = point - v0;
```

Find intersection of ray and plane with triangle
http://en.wikipedia.org/wiki/Line%E2%80%93plane_intersection

Find actual intersection

```

wu = dot(w, u);
wv = dot(w, v);
D = uv * uv - uu * vv;

float s, t;

s = (uv * wv - vv * wu) / D;
if (s < 0.0 || s > 1.0)
    return false;
t = (uv * wu - uu * wv) / D;
if (t < 0.0 || (s + t) > 1.0)
    return false;

```

$$\begin{bmatrix} t \\ u \\ v \end{bmatrix} = \frac{1}{(D \times E_2) \cdot E_1} \begin{bmatrix} (T \times E_1) \cdot E_2 \\ (D \times E_2) \cdot T \\ (T \times E_1) \cdot D \end{bmatrix} = \frac{1}{P \cdot E_1} \begin{bmatrix} Q \cdot E_2 \\ P \cdot T \\ Q \cdot D \end{bmatrix}, \quad (6)$$

where $P = (D \times E_2)$ and $Q = T \times E_1$. In our implementation we reuse these

```

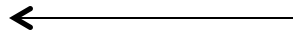
return true;
}

```

```

vec3 reflect_vector(vec3 v, vec3 n)
{
    return v - 2.0 * dot(v, n) * n;
}

```



```

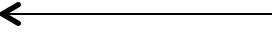
void main(void)
{

```

```

    vec3 acceleration = vec3(0.0, -0.3, 0.0);
    vec3 new_velocity = velocity + acceleration * time_step;
    vec4 new_position = position + vec4(new_velocity * time_step, 0.0);

```



```

    vec3 v0, v1, v2;
    vec3 point;
    int i;
    for (i = 0; i < triangle_count; i++)
    {
        v0 = texelFetch(geometry_tbo, i * 3).xyz;
        v1 = texelFetch(geometry_tbo, i * 3 + 1).xyz;
        v2 = texelFetch(geometry_tbo, i * 3 + 2).xyz;
        if (intersect(position.xyz, position.xyz - new_position.xyz,
            v0, v1, v2, point))

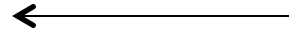
```

<https://www.opengl.org/sdk/docs/man/html/texelFetch.xhtml>

```

    {
        vec3 n = normalize(cross(v1 - v0, v2 - v0));
        new_position = vec4(point
            + reflect_vector(new_position.xyz -
                point, n), 1.0);
        new_velocity = 0.8 * reflect_vector(new_velocity, n);
    }
}

```



```

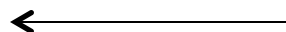
if (new_position.y < -40.0)
{
    new_position = vec4(-new_position.x * 0.3, position.y + 80.0,
        0.0, 1.0);
    new_velocity *= vec3(0.2, 0.1, -0.3);
}

```

```

velocity_out = new_velocity * 0.9999;
position_out = new_position;
gl_Position = projection_matrix * (model_matrix * position);

```



```

};

```


Configuring Particle Pass

Example 5.11 Configuring the Simulation Pass of the Particle System Simulator


```
static const char * varyings[] =
{
    "position_out", "velocity_out"
};

glTransformFeedbackVaryings(update_prog, 2, varyings,
                             GL_INTERLEAVED_ATTRIBS);

glLinkProgram(update_prog);
```




Example 5.12 Main Rendering Loop of the Particle System Simulator

```
glUseProgram(render_prog); 
glUniformMatrix4fv(render_model_matrix_loc, 1, GL_FALSE, model_matrix);
glUniformMatrix4fv(render_projection_matrix_loc, 1, GL_FALSE,
                    projection_matrix);

glBindVertexArray(render_vao);

glBindBufferBase(GL_TRANSFORM_FEEDBACK_BUFFER, 0, geometry_vbo);
glBeginTransformFeedback(GL_TRIANGLES); 
object.Render();
glEndTransformFeedback(); 

glUseProgram(update_prog); 
glUniformMatrix4fv(model_matrix_loc, 1, GL_FALSE, model_matrix);
glUniformMatrix4fv(projection_matrix_loc, 1, GL_FALSE,
                    projection_matrix);
glUniformli(triangle_count_loc, object.GetVertexCount() / 3);

if ((frame_count & 1) != 0)
{
    glBindVertexArray(vao[1]);
    glBindBufferBase(GL_TRANSFORM_FEEDBACK_BUFFER, 0, vbo[0]);
}
else
{
    glBindVertexArray(vao[0]);
    glBindBufferBase(GL_TRANSFORM_FEEDBACK_BUFFER, 0, vbo[1]);
}

glBeginTransformFeedback(GL_POINTS); 
glDrawArrays(GL_POINTS, 0, min(point_count, (frame_count >> 3)));
glEndTransformFeedback();

glBindVertexArray(0);

frame_count++;
```

Shadows

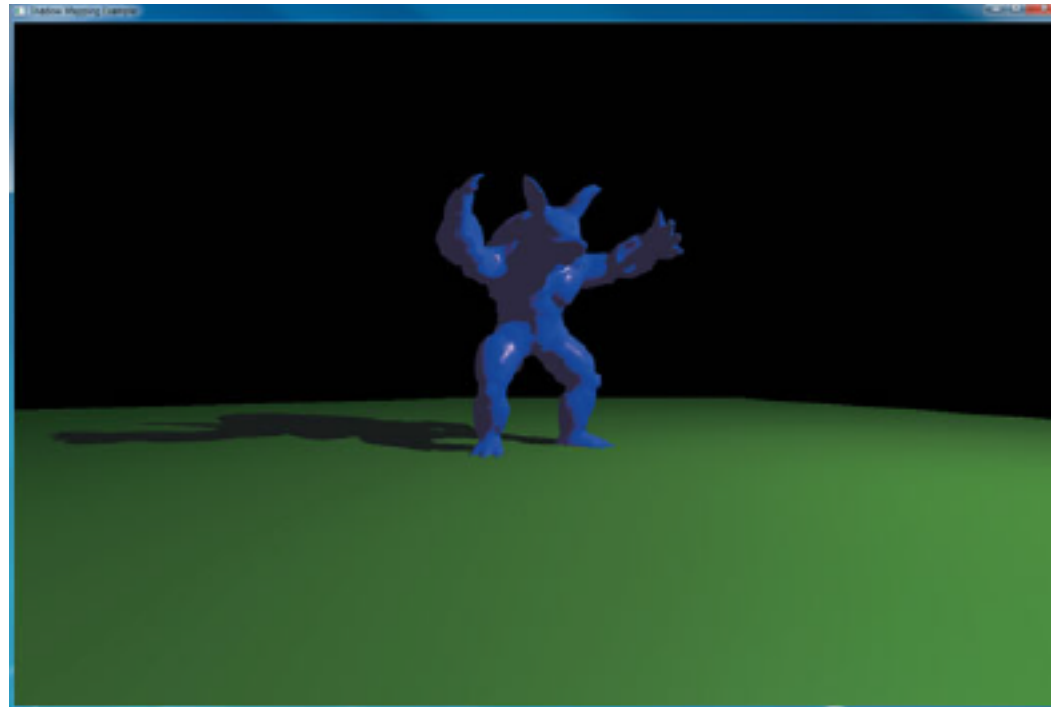
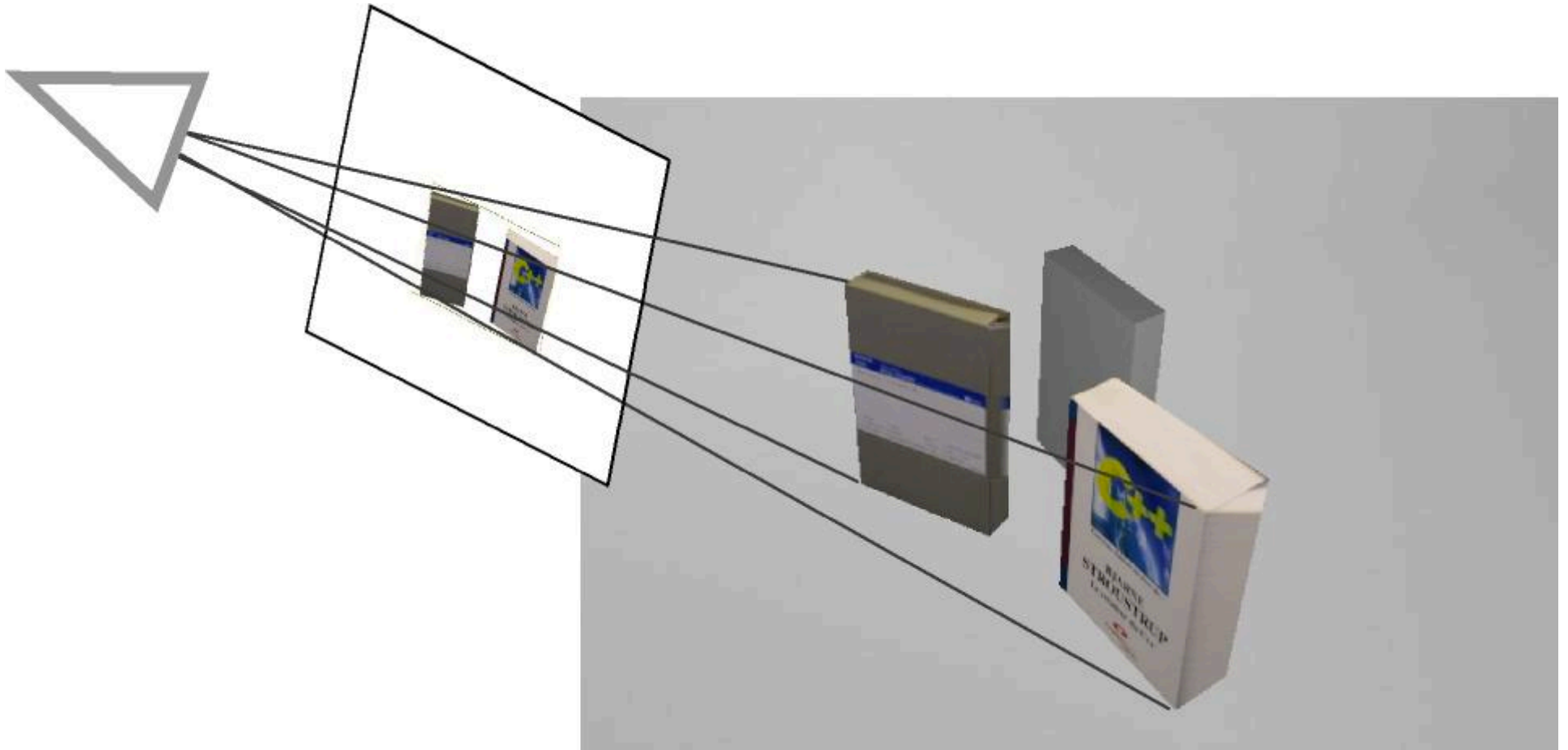


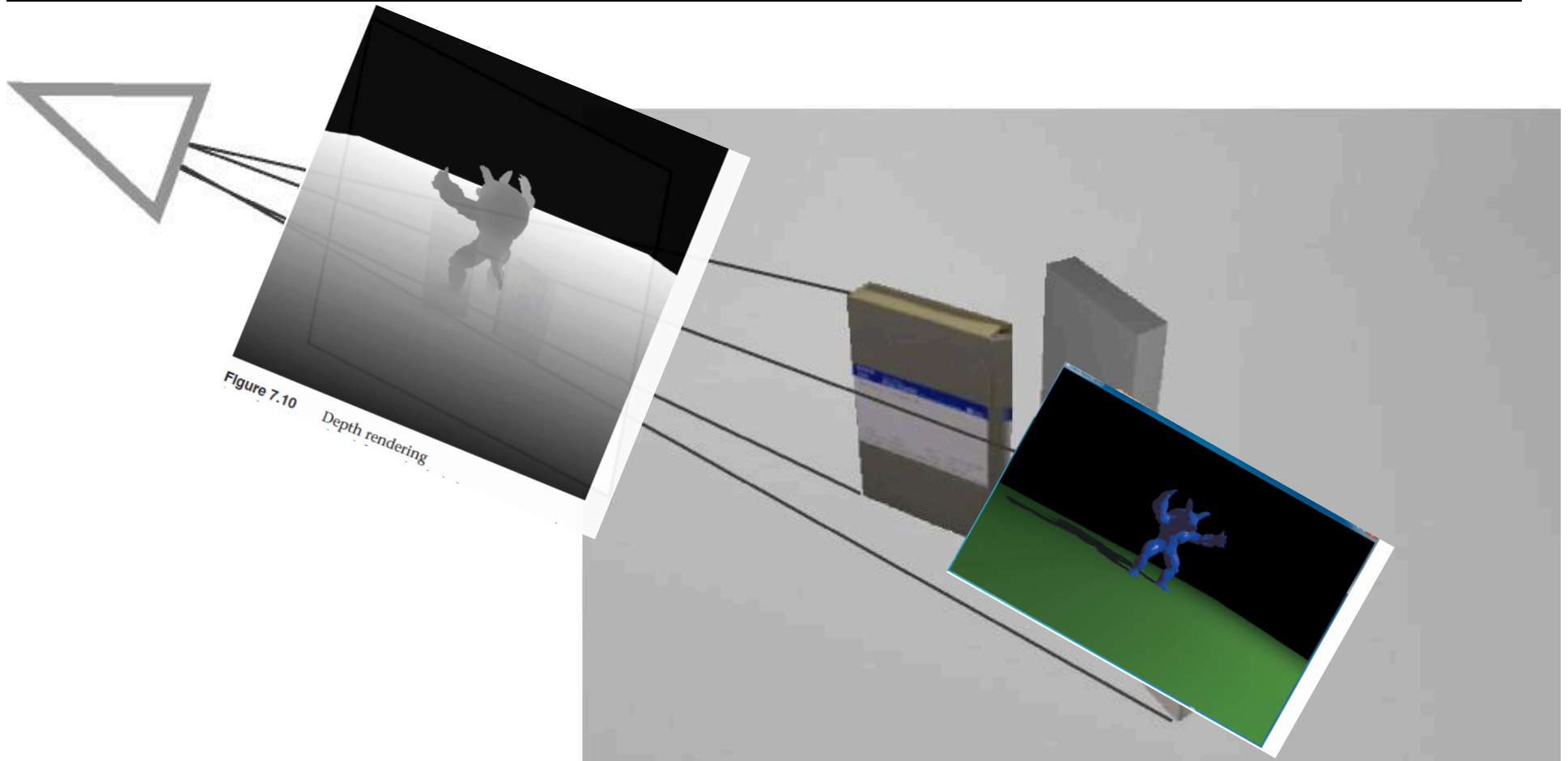
Figure 7.11 Final rendering of shadow map



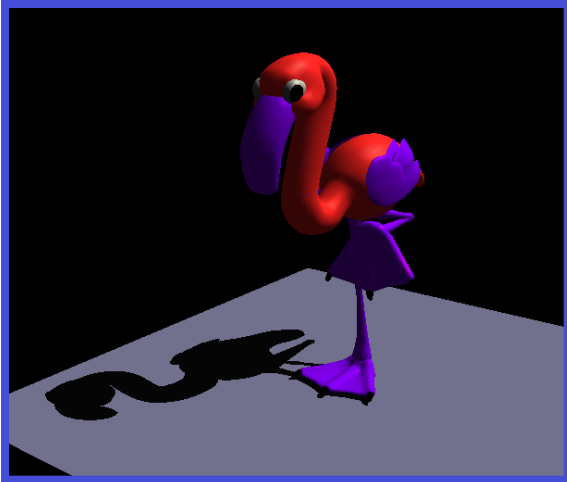
Shadows & Textures ?



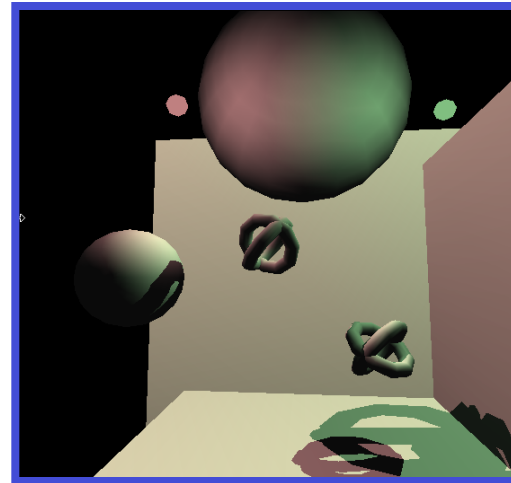
Shadows & Textures ?



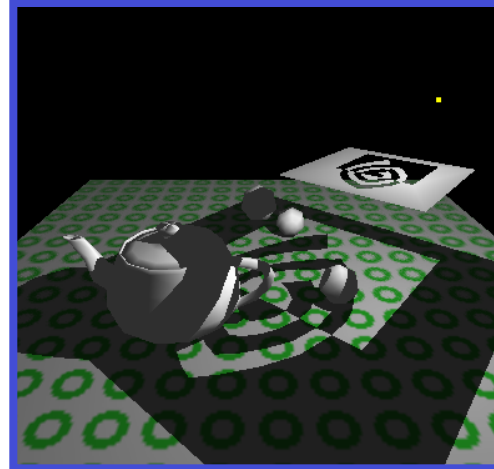
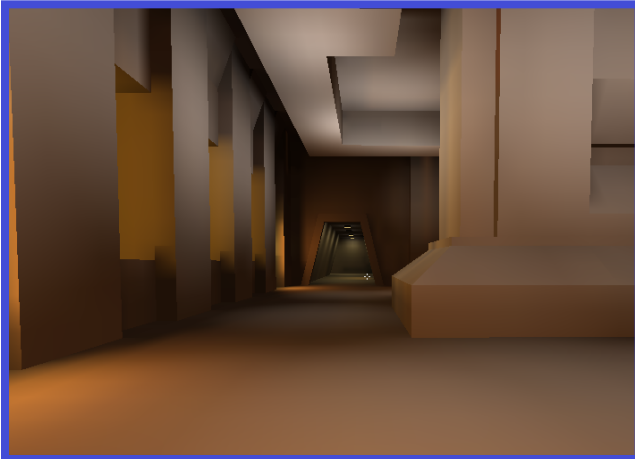
Real-time Shadow Techniques



*Projected
planar
shadows*



*Shadow
volumes*



*Hybrid
approaches*



Light maps



DEPARTMENT OF
COMPUTER SCIENCE
AND ENGINEERING

Luxo Jr. – The Famous One

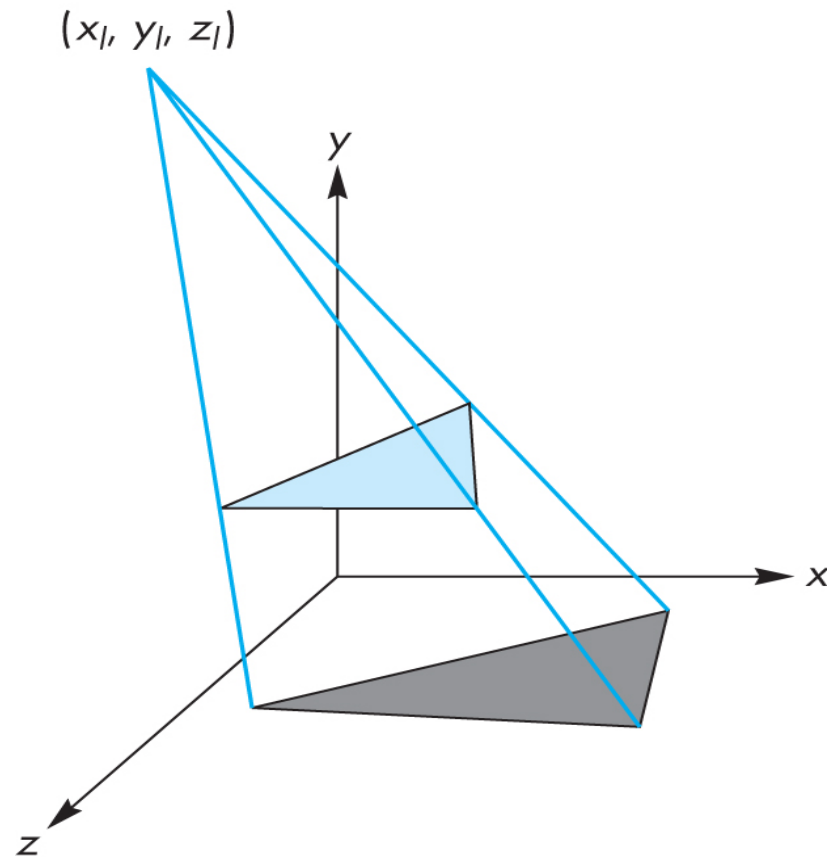
- Luxo Jr. has two animated lights and one overhead light
 - Three shadow maps *dynamically* generated per frame
- Complex geometry (cords and lamp arms) all correctly shadowed
- User controls the view, shadowing just works



(Sorry, no demo. Images are from web cast video of Apple's MacWorld Japan announcement.)



Shadow Mapping



186

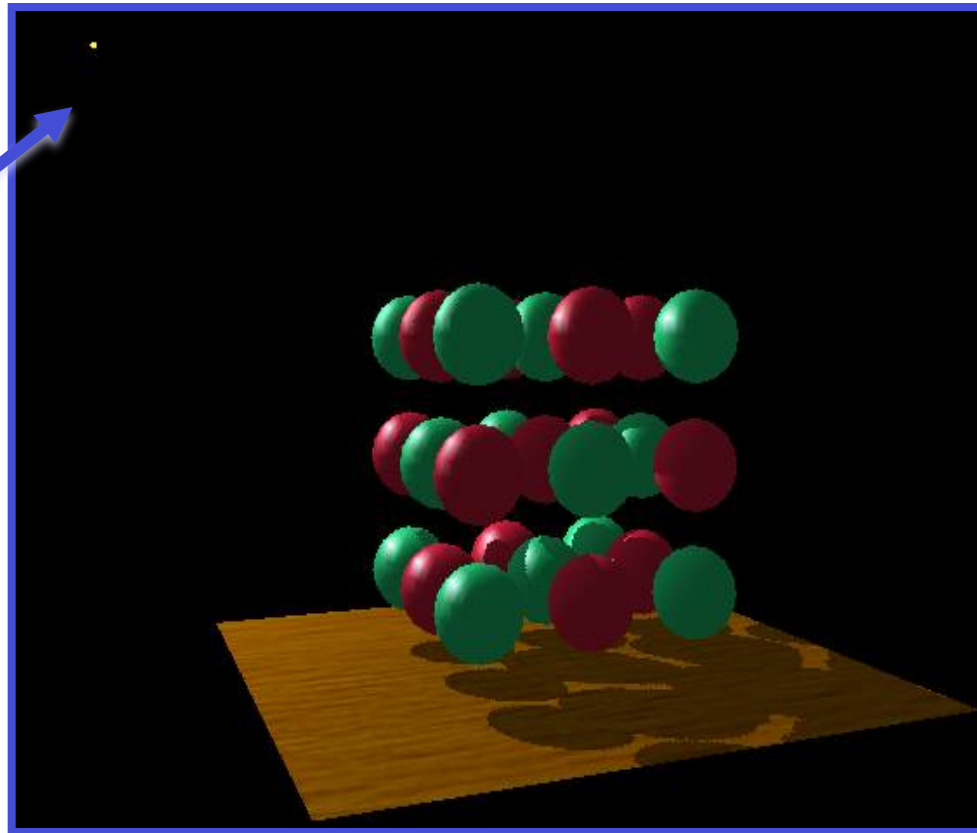
DEPARTMENT OF
COMPUTER SCIENCE
AND ENGINEERING

Projective Shadows

- Projection of a polygon is a polygon called a shadow polygon
- Given a point light source and a polygon, the vertices of the shadow polygon are the projections of the original polygon's vertices from a point source onto a surface

Visualizing Shadow Mapping

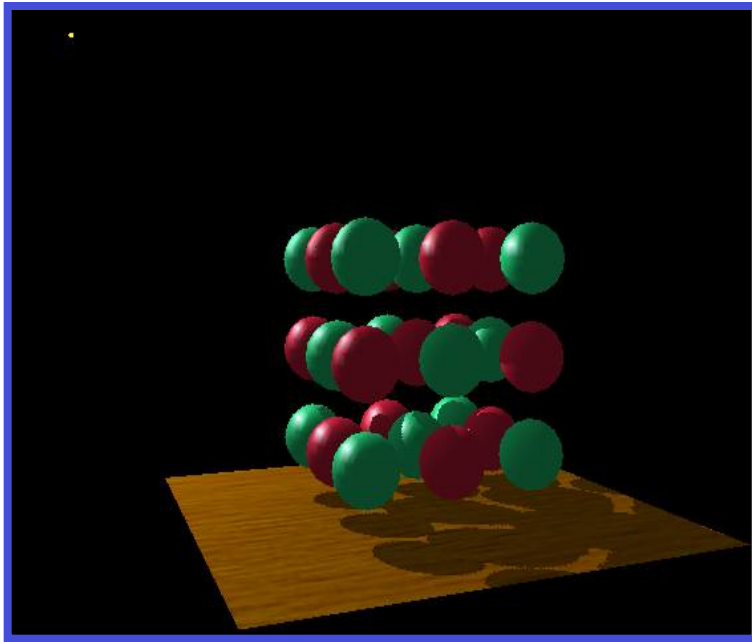
*the point
light source*



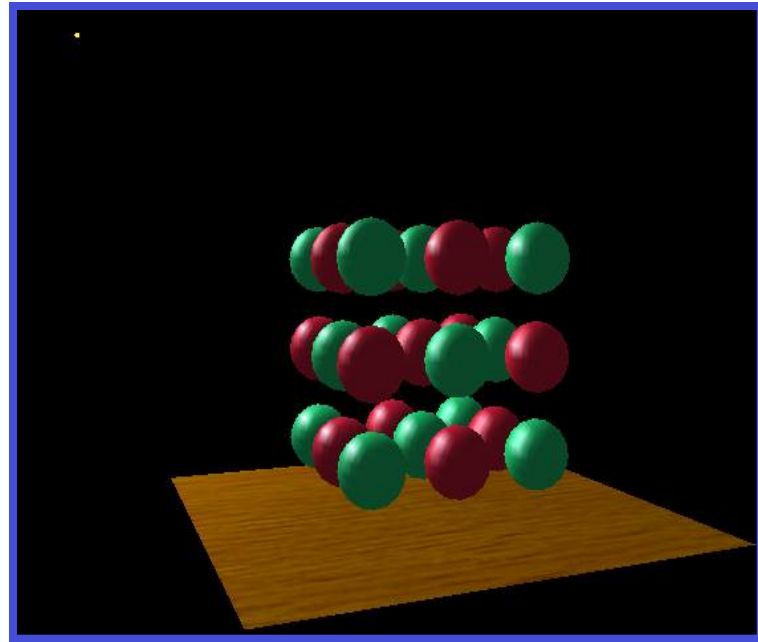
DEPARTMENT OF
COMPUTER SCIENCE
AND ENGINEERING

Visualizing Shadow Mapping

Compare with and without shadows



with shadows



without shadows

Shadow Process

1. Put two identical triangles and their colors on GPU (black for shadow triangle)
2. Compute two model view matrices as uniforms
3. Send model view matrix for original triangle
4. Render original triangle
5. Send second model view matrix
6. Render shadow triangle
 - Note shadow triangle undergoes two transformations
 - Note hidden surface removal takes care of depth issues



Shadow Map Matrices

1. Source at (x_l, y_l, z_l)
2. Vertex at (x, y, z)
3. Consider simple case of shadow projected onto ground at $(x_p, 0, z_p)$
4. Translate source to origin with $T(-x_l, -y_l, -z_l)$
5. Perspective projection

6. Translate back

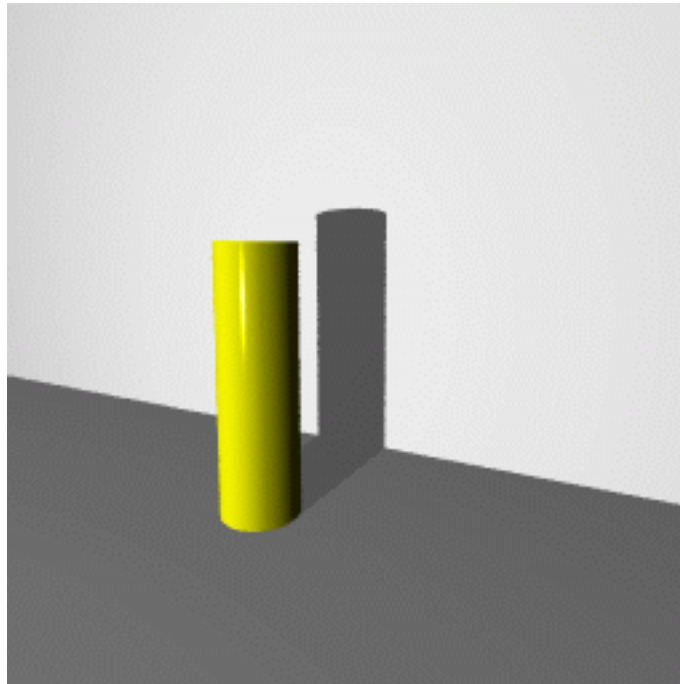
$$M = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & \frac{1}{-y_l} & 0 & 0 \end{bmatrix}$$



Shadow Maps

- Render a scene from a light source; depth buffer will contain the distances from the source to each fragment.
- Store depths in texture called **depth/shadow map**
- Render image in shadow map with light - anything lit is not in shadow.
- Form a shadow map for each source

Example



Shadow Map



Figure 7.10 Depth rendering



DEPARTMENT OF
COMPUTER SCIENCE
AND ENGINEERING

Final Rendering

- Compare distance from fragment to light source with distance in the shadow map
- If depth in shadow map is less than distance from fragment to source, fragment is in shadow (from this source)
- Otherwise we use rendered color

Shadows

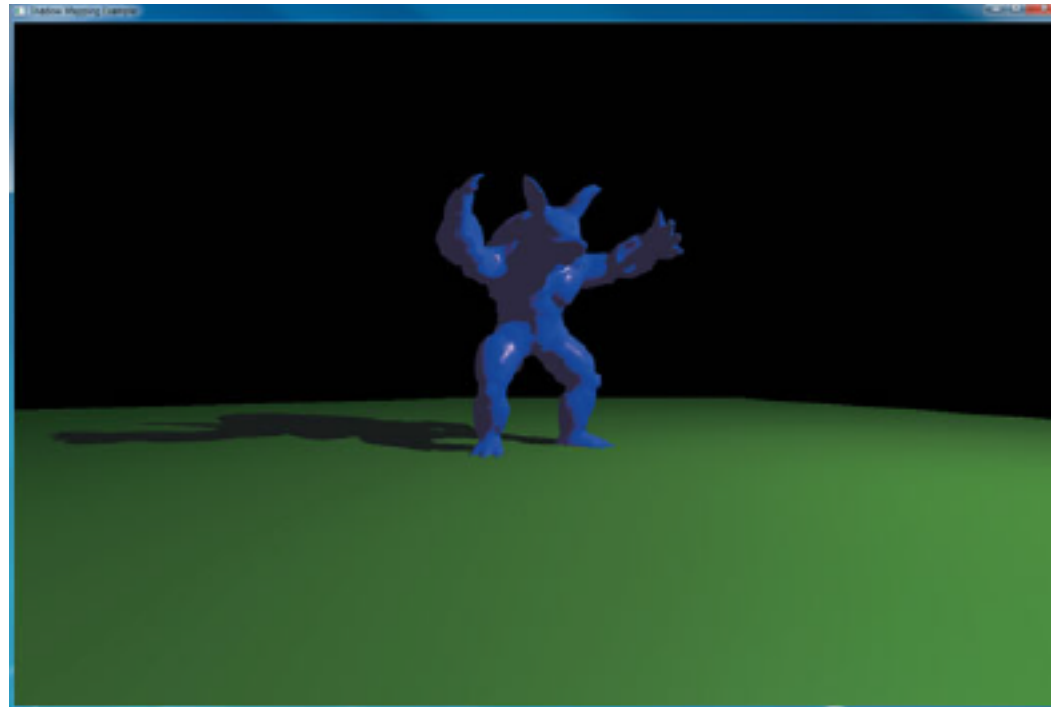


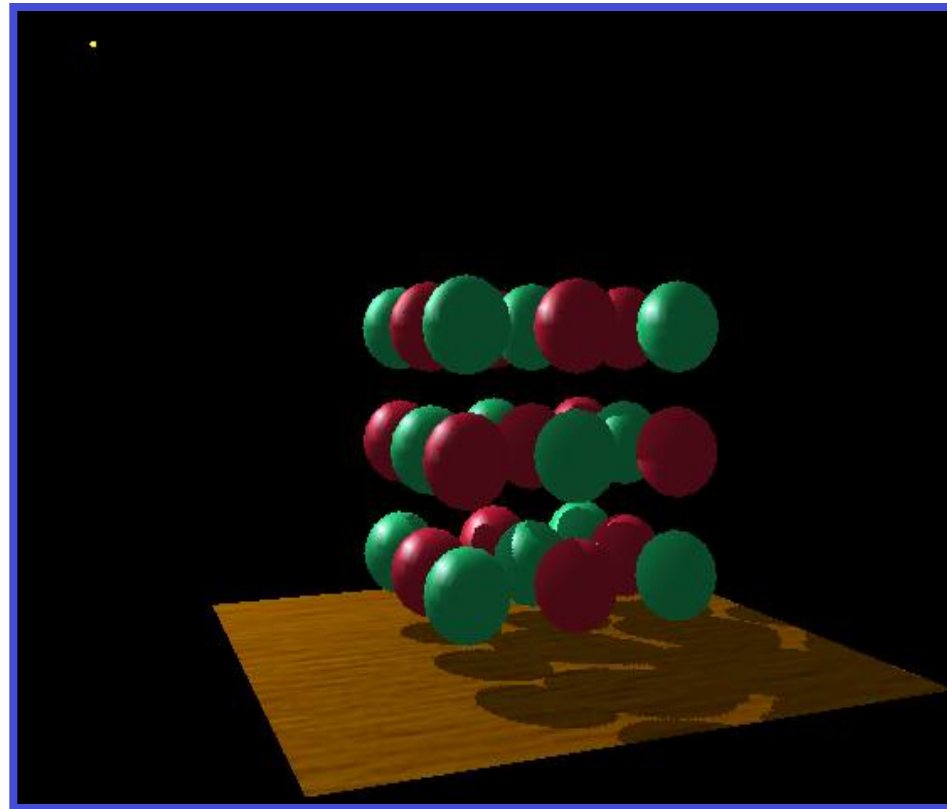
Figure 7.11 Final rendering of shadow map



Visualizing Shadow Mapping

Scene with shadows

*Notice how
specular
highlights never
appear in
shadows*



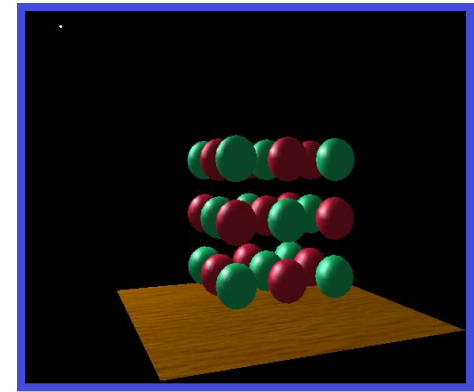
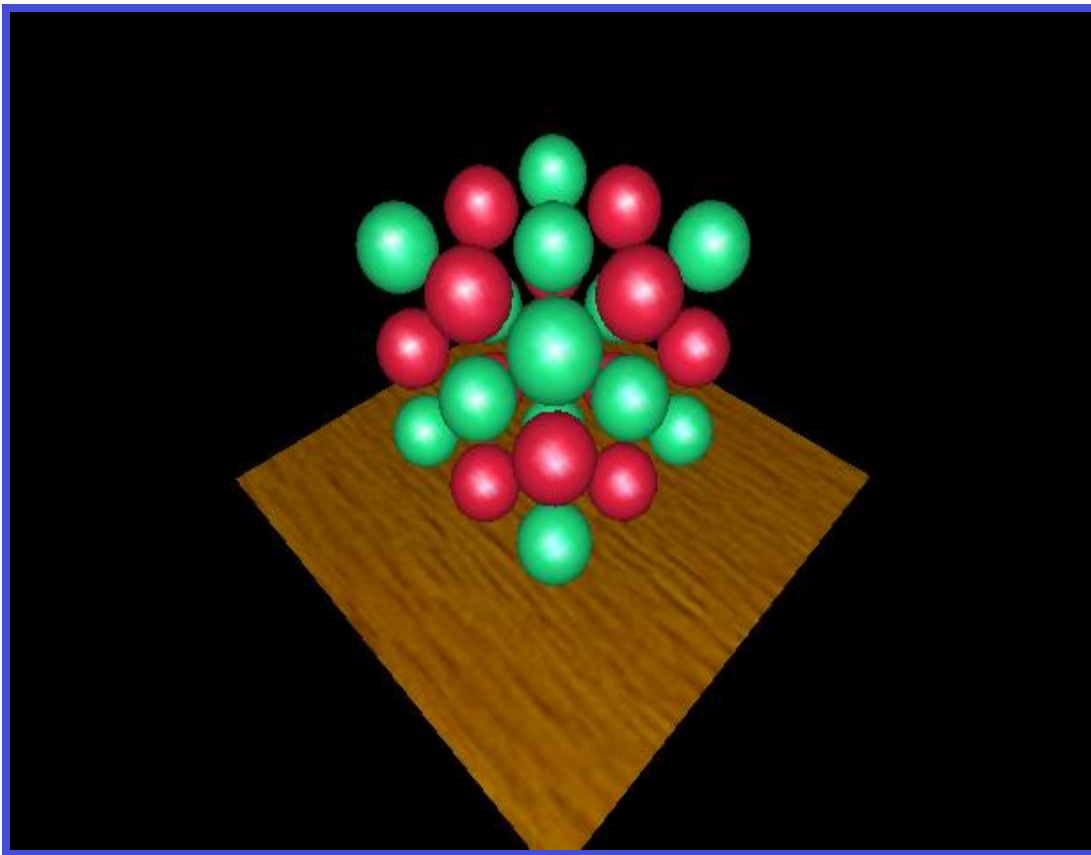
*Notice how
curved surfaces
cast shadows
on each other*

Applications Side

- Start with vertex in object coordinates
- Want to convert representation to texture coordinates
- Form LookAt matrix from light source to origin in object coordinates (MVL)
- From projection matrix for light source (PL)
- From a matrix to convert from $[-1, 1]$ clip coordinates to $[0, 1]$ texture coordinates
- Concatenate to form object to texture coordinate matrix (OTC)

Visualizing Shadow Mapping

The scene from the light's point-of-view



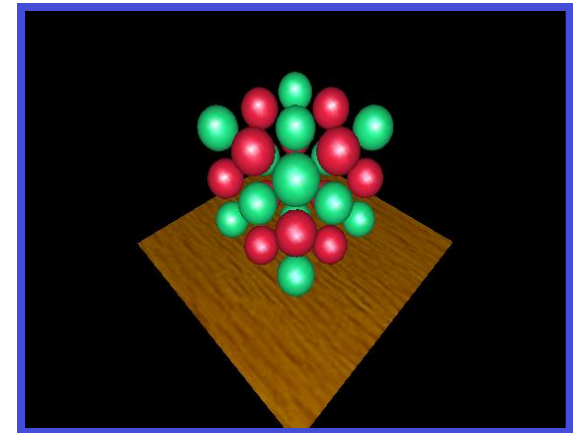
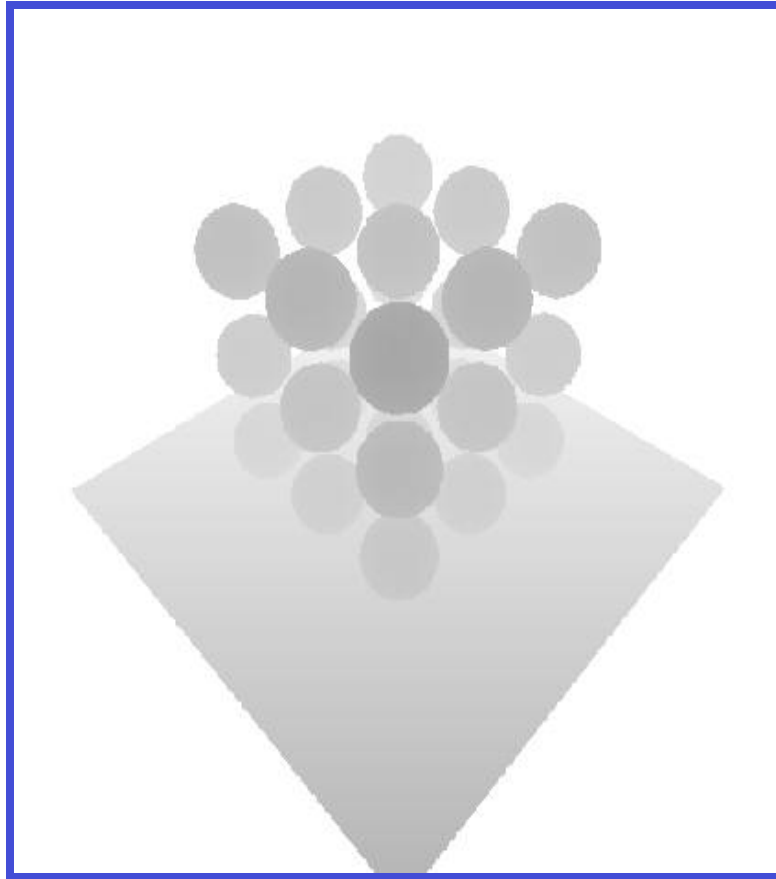
*FYI: from the
eye's point-of-view
again*



DEPARTMENT OF
COMPUTER SCIENCE
AND ENGINEERING

Visualizing Shadow Mapping

The depth buffer from the light's point-of-view



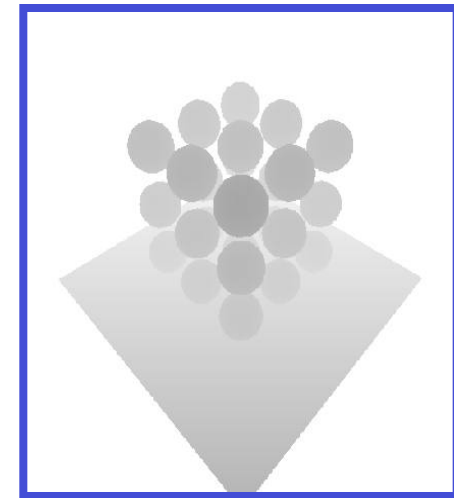
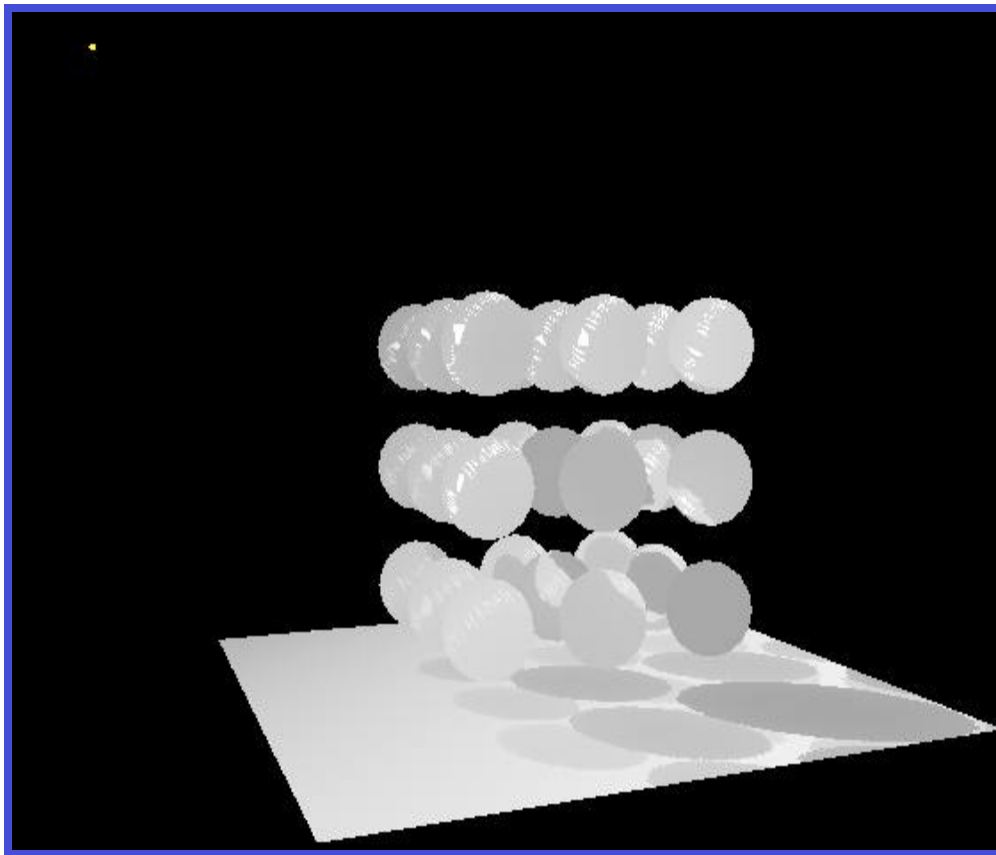
*FYI: from the
light's point-of-view
again*



DEPARTMENT OF
COMPUTER SCIENCE
AND ENGINEERING

Visualizing Shadow Mapping

Projecting the depth map onto the eye's view



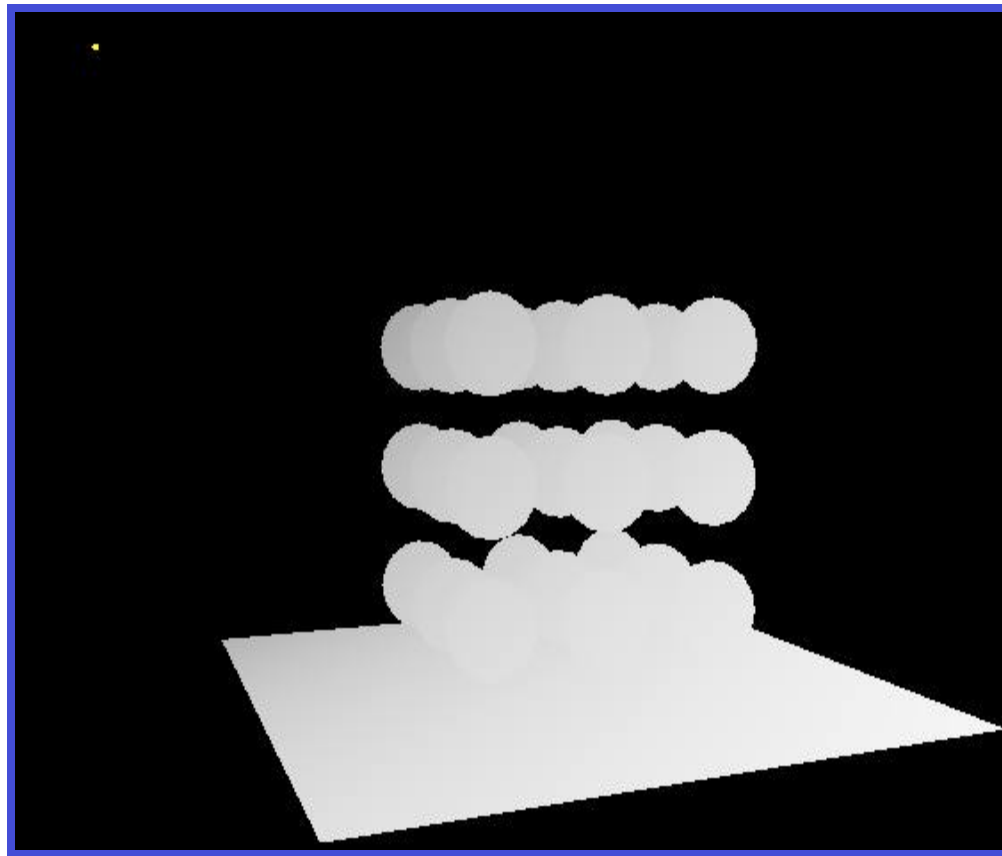
*FYI: depth map for
light's point-of-view
again*



DEPARTMENT OF
COMPUTER SCIENCE
AND ENGINEERING

Visualizing Shadow Mapping

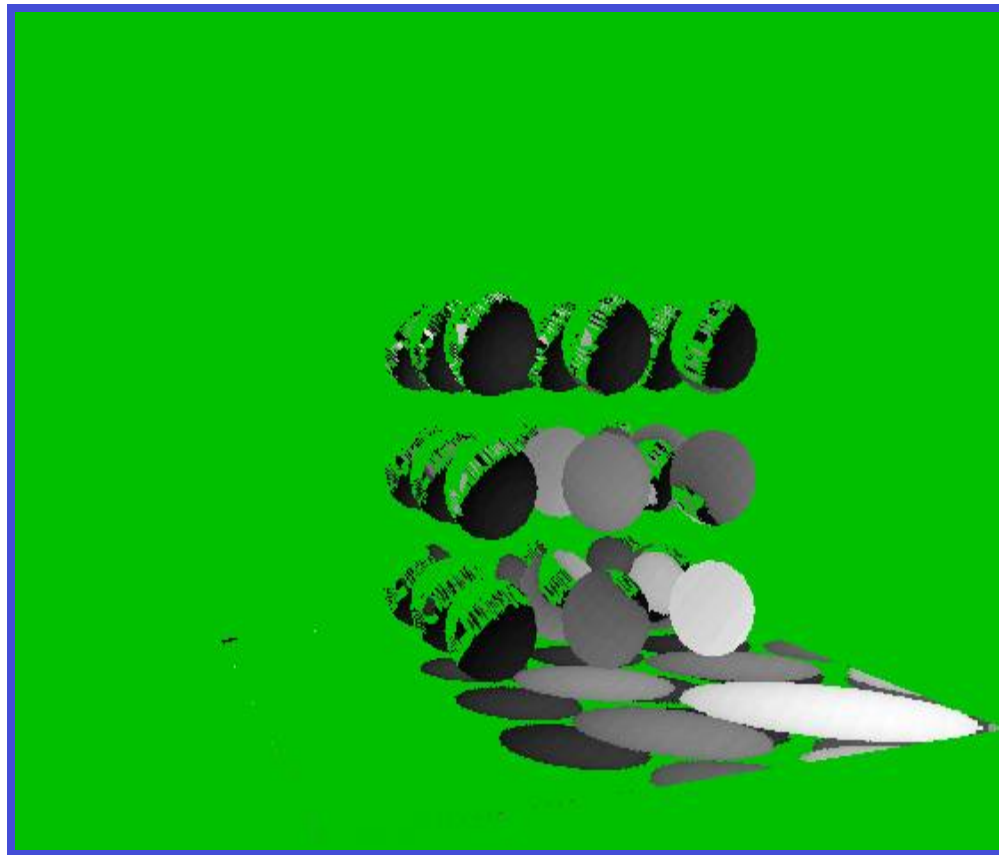
Projecting light's planar distance onto eye's view



Visualizing Shadow Mapping

Comparing light distance to light depth map

*Green is where
the light planar
distance and
the light depth
map are
approximately
equal*



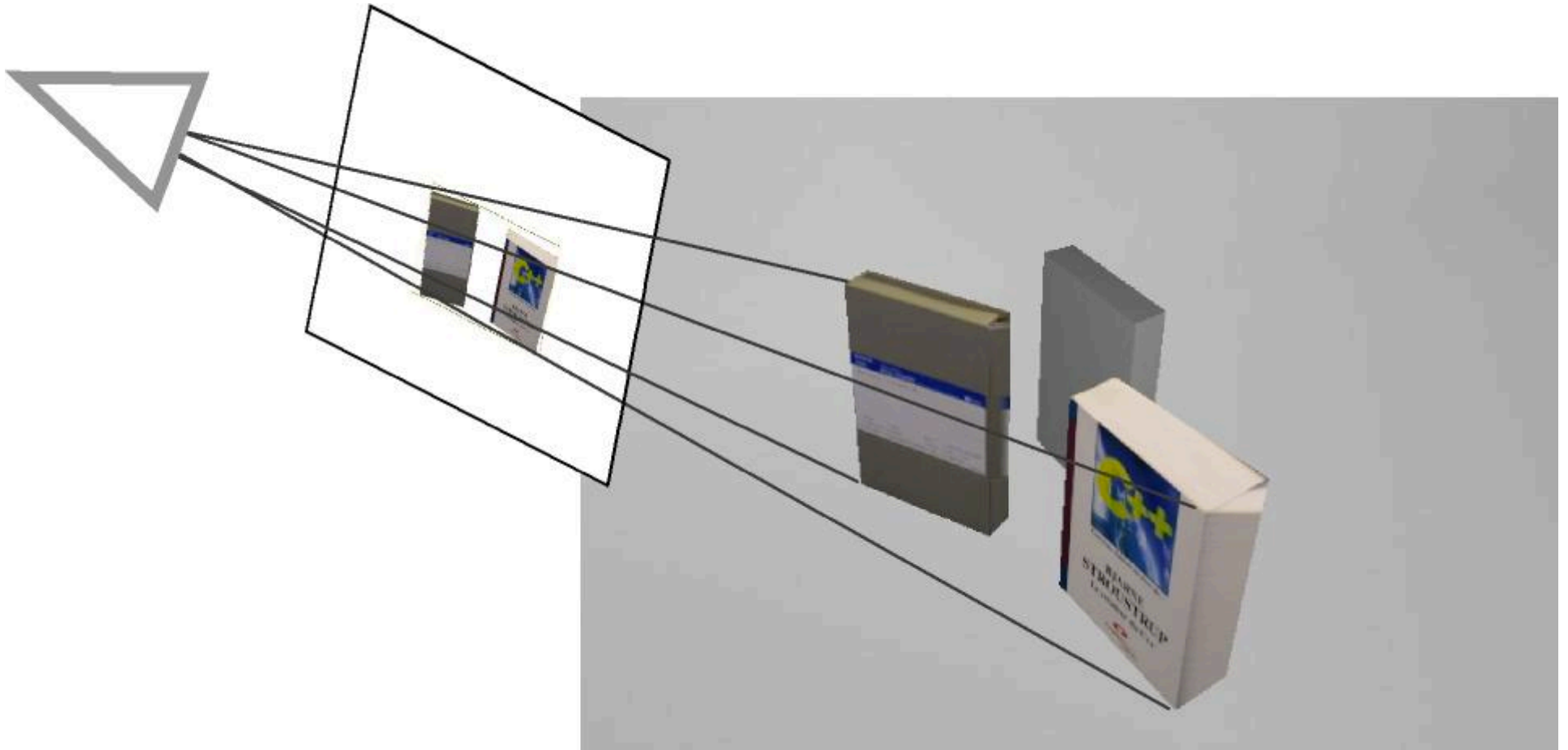
*Non-green is
where shadows
should be*



Generalized Shadows

- Approach was OK for shadows on a single flat surface
- Cannot handle shadows on general objects

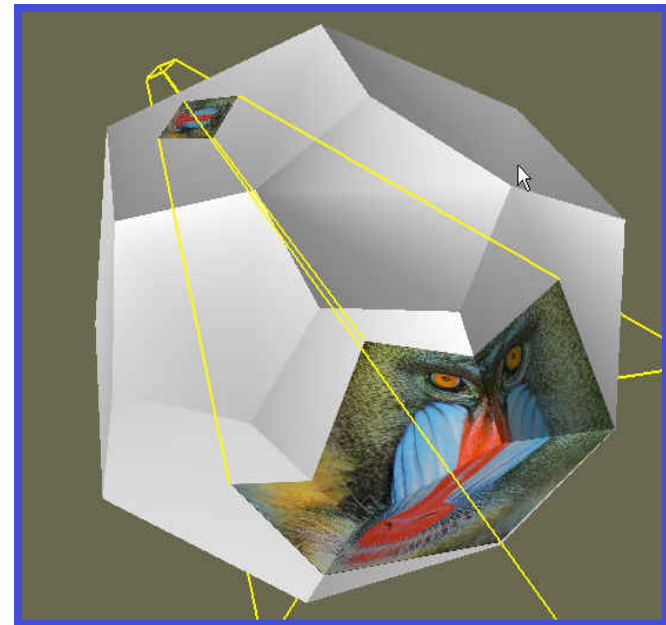
Projective Textures



Projective Texturing?

An intuition for projective texturing

- The slide projector analogy



Source: Wolfgang Heidrich [99]

Image Based Lighting

- Project texture onto surface; treat texture as “slide projector”
- Projective textures and image based lighting
- OpenGL/GLSL – 4D texture coordinates



DEPARTMENT OF
COMPUTER SCIENCE
AND ENGINEERING

Projective Texturing

Key - perspective-correct texturing?

- Normal 2D texture mapping uses (s, t) coordinates
- 2D perspective-correct texture mapping
 - (s, t) should be interpolated linearly in eye-space
 - compute per-vertex s/w , t/w , and $1/w$
 - linearly interpolate these three parameters over polygon
 - per-fragment compute $s' = (s/w) / (1/w)$ and $t' = (t/w) / (1/w)$
 - results in per-fragment perspective correct (s', t')

Projective Texturing

- Consider homogeneous texture coordinates
 - $(s, t, r, q) \rightarrow (s/q, t/q, r/q)$
 - Similar to homogeneous clip coordinates where $(x, y, z, w) = (x/w, y/w, z/w)$
- Project $(s/q, t/q, r/q)$ per-fragment

Projective Texturing

Tricking hardware into doing projective textures

- By interpolating q/w , hardware computes per-fragment
 - $(s/w) / (q/w) = s/q$
 - $(t/w) / (q/w) = t/q$
- Net result: projective texturing

4D Textures Coordinates

- Texture coordinates (s, t, r, q) affected by perspective division; actual coordinates (s/q, t/q, r/q) or (s/q, t/q) for 2D textures
- GLSL – *textureProj* uses the 2D/3D texture coordinate obtained by a perspective division of a 4D texture coordinate a texture value from a sampler

`color = textureProj(my_sampler, tex_coord)`



Shadow Map Generation

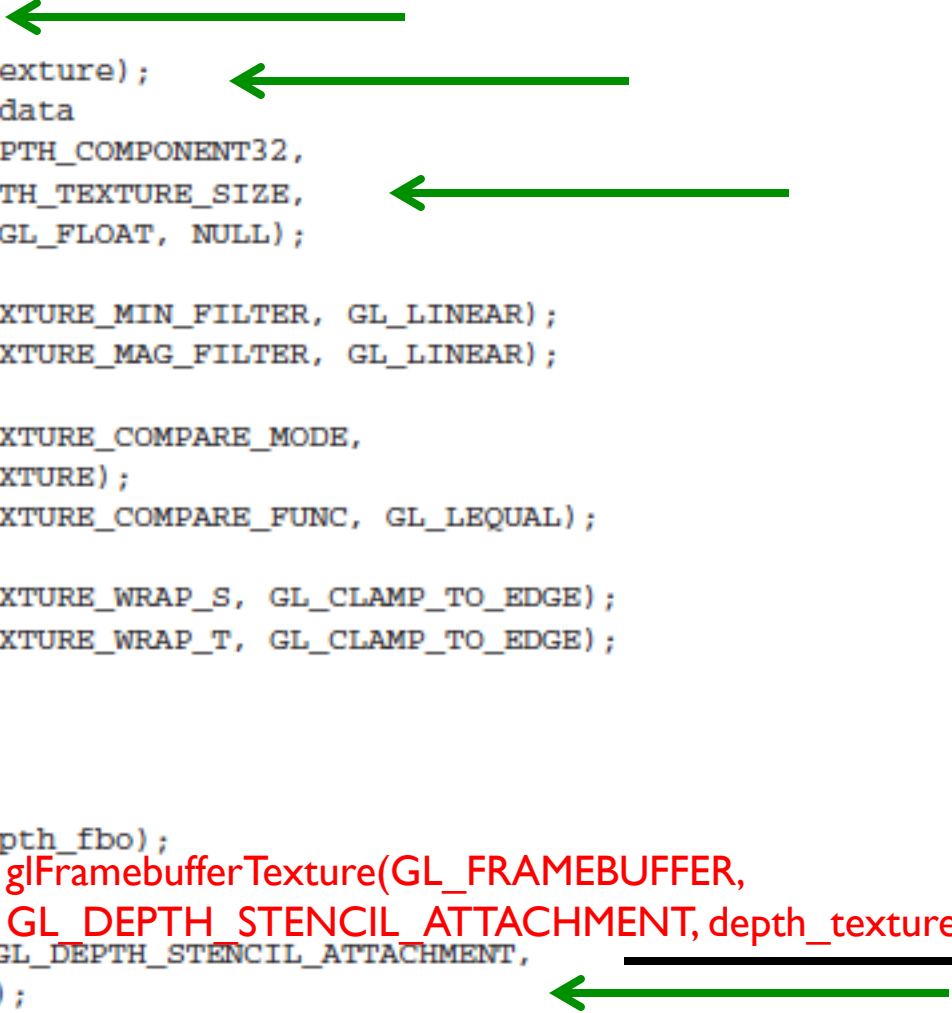
Matrices

Texture Parameters - OpenGL

- Example 7.15 Creating a Framebuffer Object with a Depth Attachment

```
// Create a depth texture
glGenTextures(1, &depth_texture);
glBindTexture(GL_TEXTURE_2D, depth_texture);
// Allocate storage for the texture data
glTexImage2D(GL_TEXTURE_2D, 0, GL_DEPTH_COMPONENT32,
             DEPTH_TEXTURE_SIZE, DEPTH_TEXTURE_SIZE,
             0, GL_DEPTH_COMPONENT, GL_FLOAT, NULL);
// Set the default filtering modes
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
// Set up depth comparison mode
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_COMPARE_MODE,
                GL_COMPARE_REF_TO_TEXTURE);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_COMPARE_FUNC, GL_LEQUAL);
// Set up wrapping modes
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
glBindTexture(GL_TEXTURE_2D, 0);

// Create FBO to render depth into
glGenFramebuffers(1, &depth_fbo);
glBindFramebuffer(GL_FRAMEBUFFER, depth_fbo);
glFramebufferTexture(GL_FRAMEBUFFER, GL_DEPTH_STENCIL_ATTACHMENT,
                    depth_texture, 0);
// Disable color rendering as there are no color attachments
glDrawBuffer(GL_NONE);
```



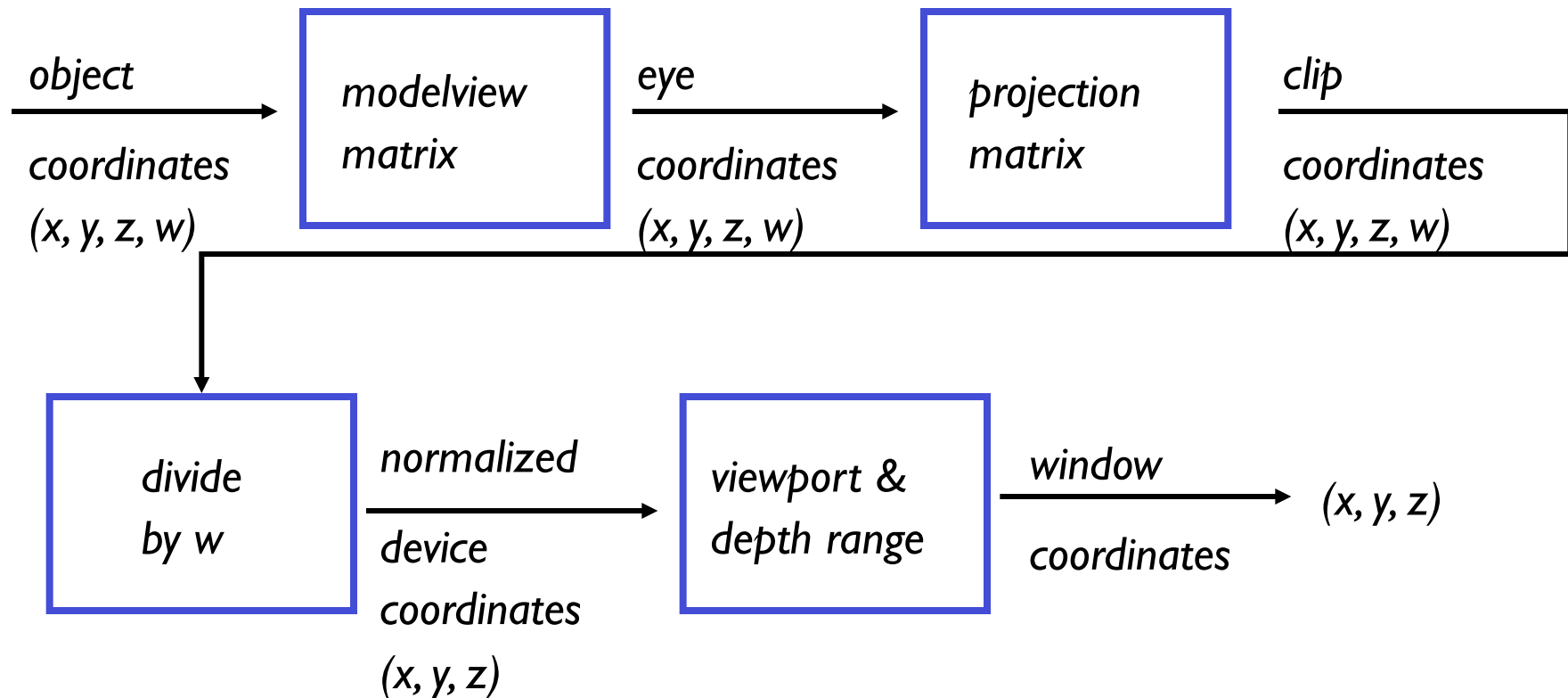
Check

<http://openme.gl/opengl-4-tutorial-code/>

```
glFramebufferTexture(GL_FRAMEBUFFER,  
GL_DEPTH_STENCIL_ATTACHMENT, depth_texture, 0);
```

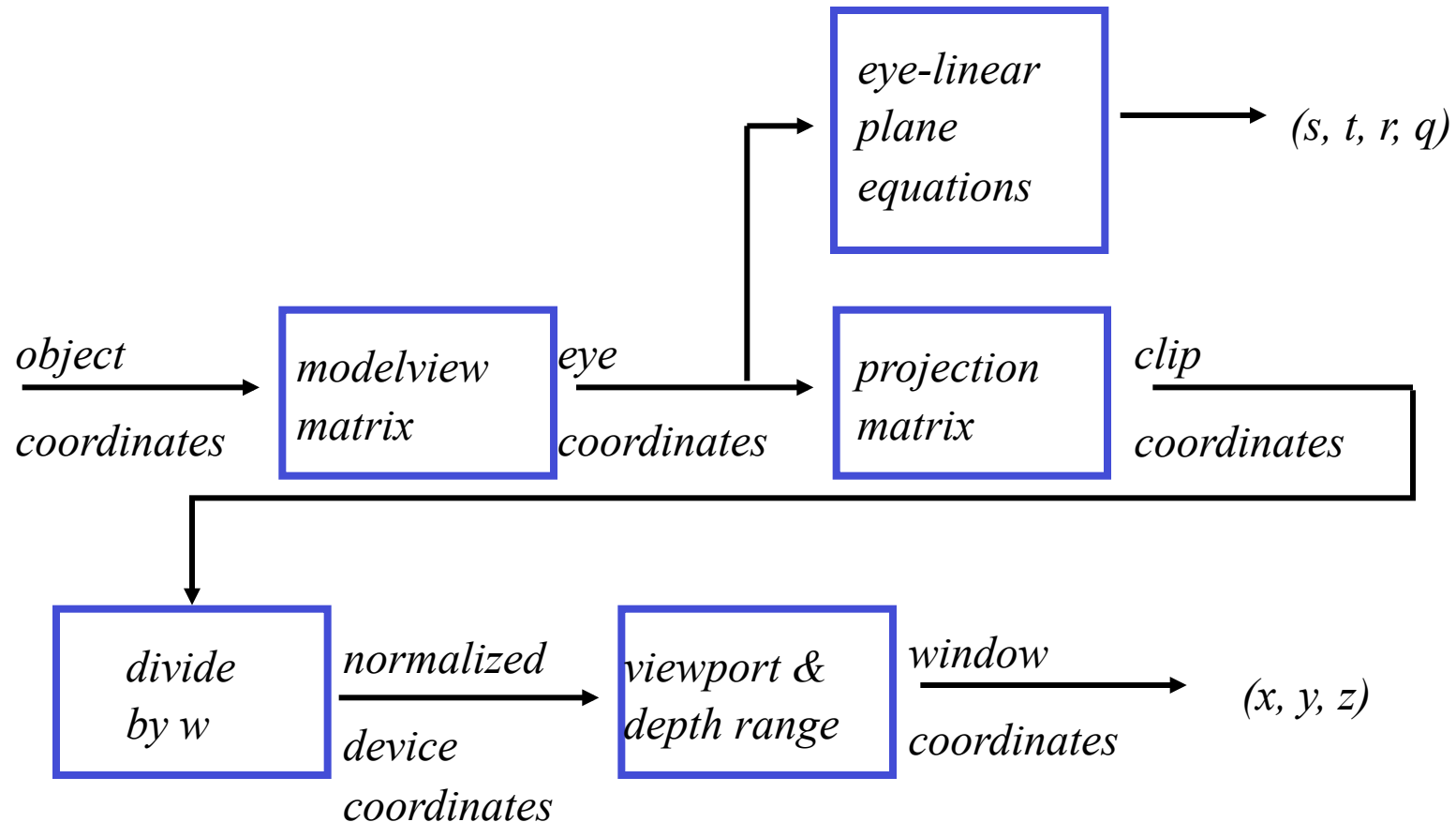
Vertex Coordinate Transform

From object to window coordinates



Eye Linear Texture Coordinate

Generating texture coordinates from eye-space



Transforms

$$\begin{bmatrix} x_e \\ y_e \\ z_e \\ w_e \end{bmatrix} = \begin{bmatrix} \text{Eye} \\ \text{view} \\ \text{(look at)} \\ \text{matrix} \end{bmatrix} \begin{bmatrix} \text{Modeling} \\ \text{matrix} \end{bmatrix} \begin{bmatrix} x_o \\ y_o \\ z_o \\ w_o \end{bmatrix}$$

Map Use

$$\begin{bmatrix} s \\ t \\ r \\ q \end{bmatrix} = \begin{bmatrix} 1/2 & 1/2 \\ & 1/2 & 1/2 \\ & & 1/2 & 1/2 \\ & & & 1 \end{bmatrix} \begin{bmatrix} \text{Light} \\ \text{frustum} \\ \text{(projection)} \\ \text{matrix} \end{bmatrix} \begin{bmatrix} \text{Light} \\ \text{view} \\ \text{(look at)} \\ \text{matrix} \end{bmatrix} \begin{bmatrix} \text{Inverse} \\ \text{eye} \\ \text{view} \\ \text{(look at)} \\ \text{matrix} \end{bmatrix} \begin{bmatrix} x_e \\ y_e \\ z_e \\ w_e \end{bmatrix}$$

Setting Up Matrices

Example 7.16 Setting up the Matrices for Shadow Map Generation

```
// Time varying light position
vec3 light_position = vec3(
    sinf(t * 6.0f * 3.141592f) * 300.0f,
    200.0f,
    cosf(t * 4.0f * 3.141592f) * 100.0f + 250.0f);

// Matrices for rendering the scene
mat4 scene_model_matrix = rotate(t * 720.0f, Y);

// Matrices used when rendering from the light's position
mat4 light_view_matrix = lookat(light_position, vec3(0.0f), Y);
mat4 light_projection_matrix(frustum(-1.0f, 1.0f, -1.0f, 1.0f,
                                     1.0f, FRUSTUM_DEPTH));

// Now we render from the light's position into the depth buffer.
// Select the appropriate program
glUseProgram(render_light_prog);

glUniformMatrix4fv(render_light_uniforms.MVPMatrix,
                  1, GL_FALSE,
                  light_projection_matrix *
                  light_view_matrix *
                  scene_model_matrix);
```



Simple Shaders

Example 7.17 Simple Shader for Shadow Map Generation

```
----- Vertex Shader -----  
// Vertex shader for shadow map generation  
#version 330 core  
uniform mat4 MVPMatrix;  
layout (location = 0) in vec4 position;  
void main(void)  
{  
    gl_Position = MVPMatrix * position; ←  
}  
  
----- Fragment Shader -----  
// Fragment shader for shadow map generation  
#version 330 core  
layout (location = 0) out vec4 color;  
void main(void)  
{  
    color = vec4(1.0); ←  
}
```



Depth Rendering

Example 7.18 Rendering the Scene From the Light's Point of View

```
// Bind the "depth only" FBO and set the viewport to the size
// of the depth texture
glBindFramebuffer(GL_FRAMEBUFFER, depth_fbo);
glViewport(0, 0, DEPTH_TEXTURE_SIZE, DEPTH_TEXTURE_SIZE);

// Clear
glClearDepth(1.0f);
glClear(GL_DEPTH_BUFFER_BIT);      ←

// Enable polygon offset to resolve depth-fighting issues
glEnable(GL_POLYGON_OFFSET_FILL);      ←
glPolygonOffset(2.0f, 4.0f);
// Draw from the light's point of view
DrawScene(true);
glDisable(GL_POLYGON_OFFSET_FILL);      ←
```



In Practice

- Two Issues
 - Constructing the depth map
 - use existing hardware depth buffer
 - use `glPolygonOffset` to offset depth value back
 - read back the depth buffer contents
 - Depth map can be copied to a 2D texture
 - unfortunately, depth values tend to require more precision than 8-bit typical for textures
 - depth precision typically 16-bit or 24-bit

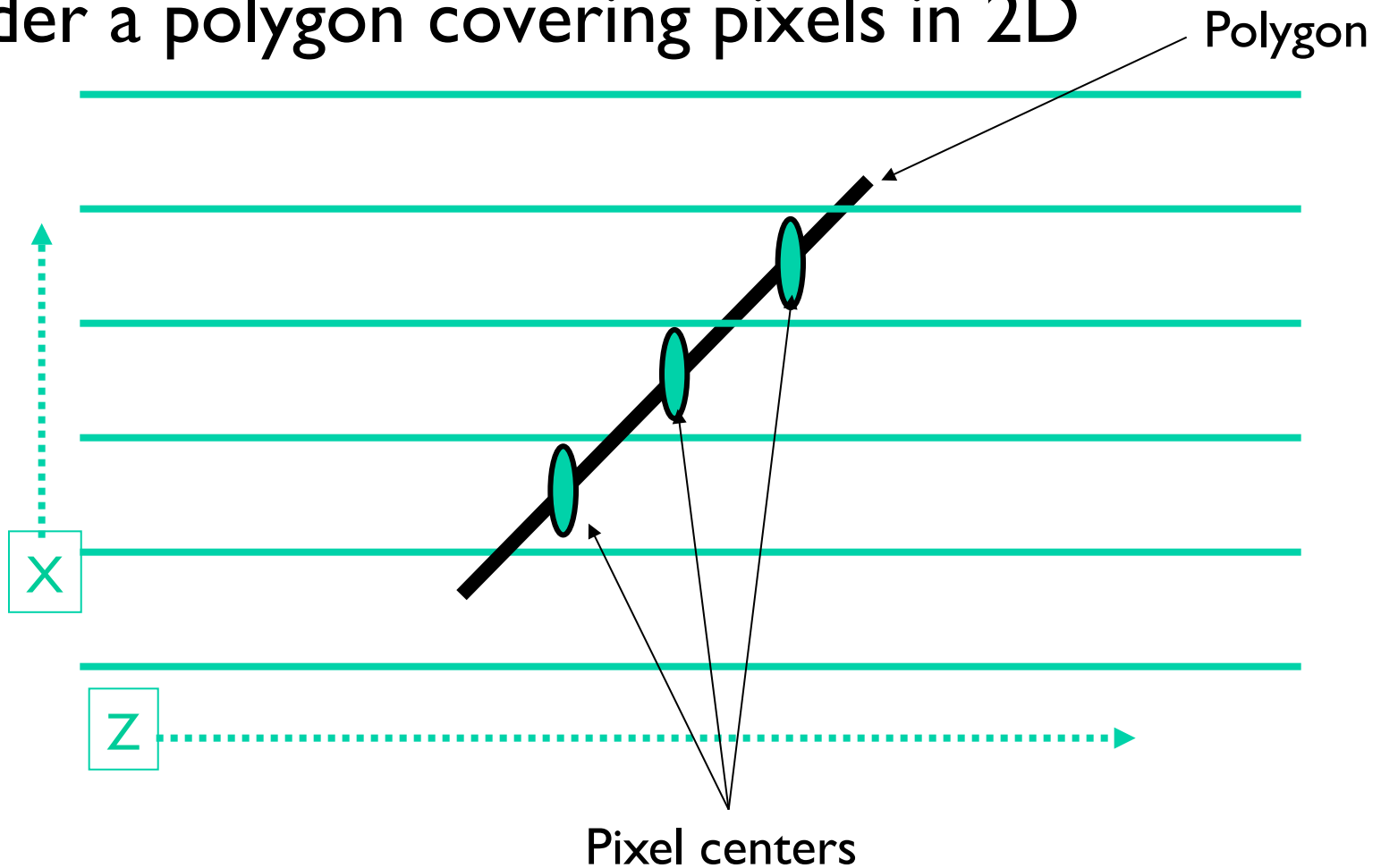


glPolygonOffset

- Depth buffer contains “window space” depth values
 - Post-perspective divide means non-linear distribution
 - glPolygonOffset is guaranteed to be a window space offset
- Doing a “clip space” glTranslatef is not sufficient
 - Common shadow mapping implementation mistake
 - Actual bias in depth buffer units will vary over the frustum
 - No way to account for slope of polygon

In Pictures - Pixel Centers

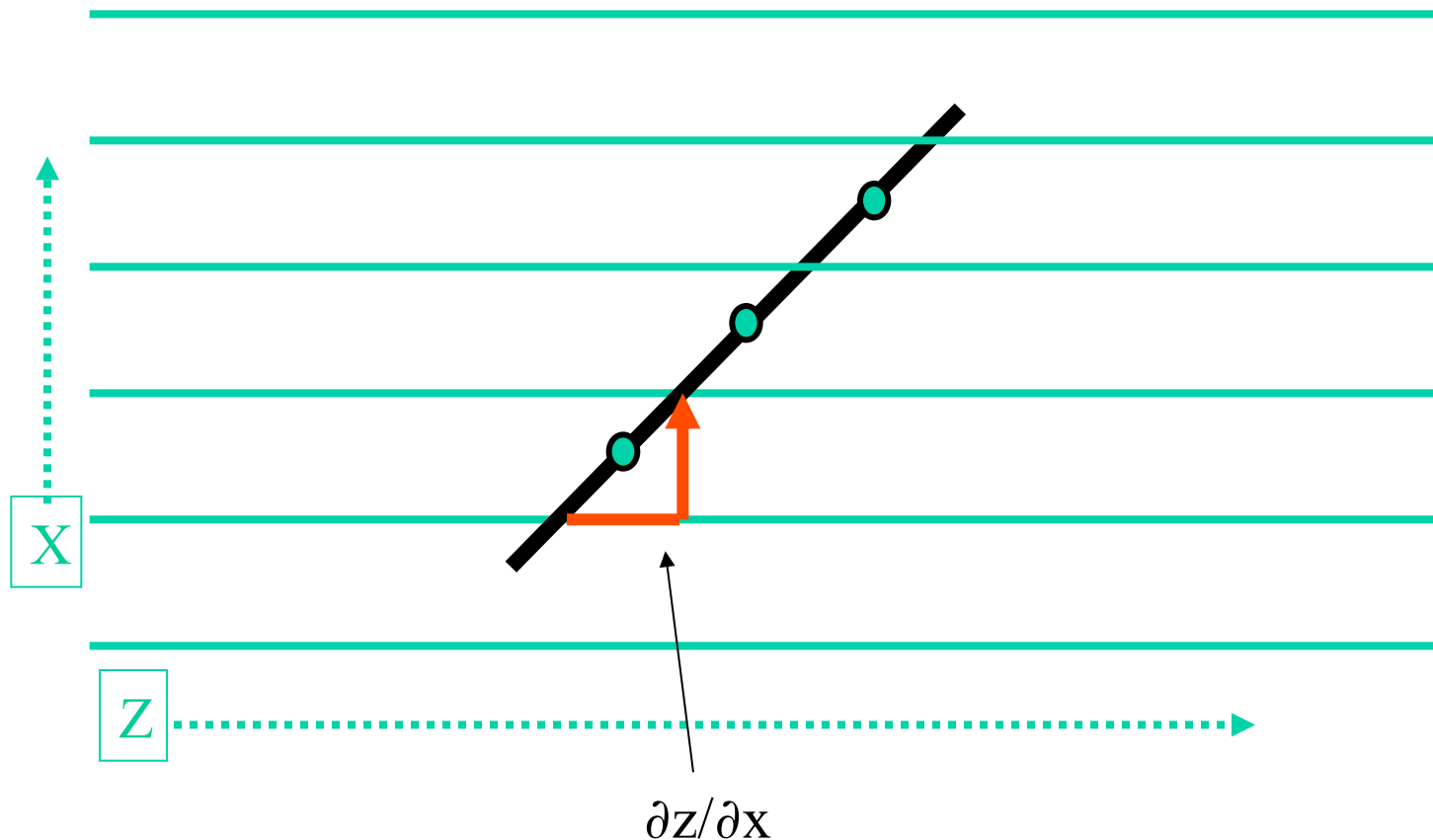
Consider a polygon covering pixels in 2D



DEPARTMENT OF
COMPUTER SCIENCE
AND ENGINEERING

In Pictures - Pixel Centers

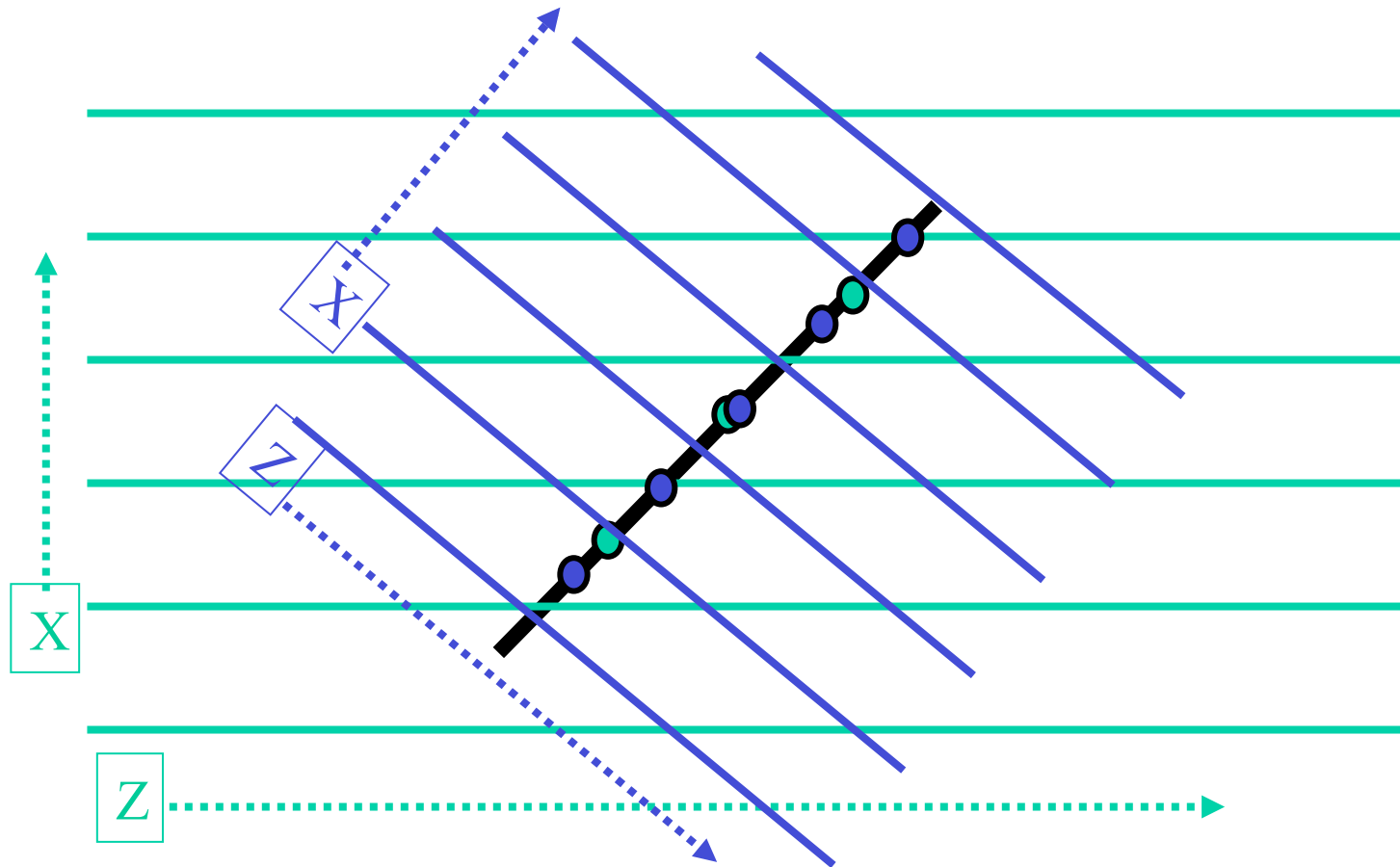
Change of Z with respect to X



DEPARTMENT OF
COMPUTER SCIENCE
AND ENGINEERING

In Pictures - Pixel Centers

Consider a 2nd grid for the polygon covering pixels in 2D



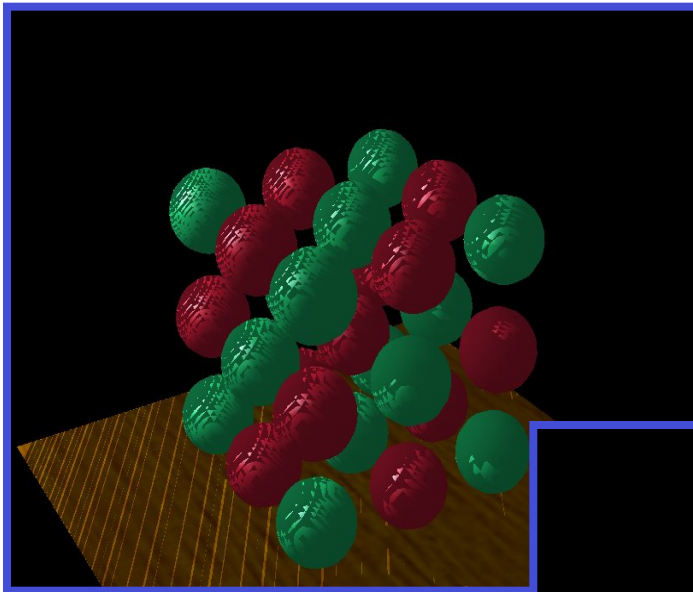
DEPARTMENT OF
COMPUTER SCIENCE
AND ENGINEERING

glPolygonOffset's Slope

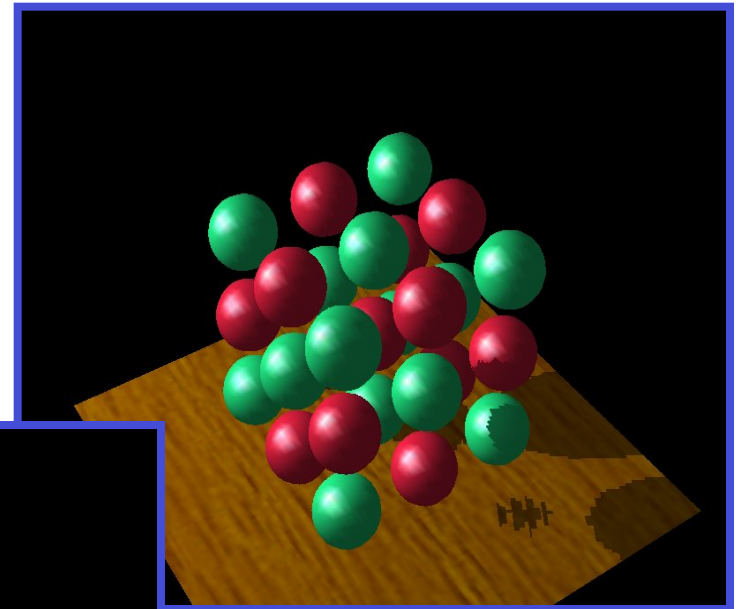
- Pixel center is re-sampled to another grid
 - For example, the shadow map texture's grid!
- The re-sampled depth could be off by $\pm 0.5 \frac{\partial z}{\partial x}$ and $\pm 0.5 \frac{\partial z}{\partial y}$
- The maximum absolute error would be $|0.5 \frac{\partial z}{\partial x}| + |0.5 \frac{\partial z}{\partial y}| \approx \max(|\frac{\partial z}{\partial x}|, |\frac{\partial z}{\partial y}|)$
 - This assumes the two grids have pixel footprint area ratios of 1.0
 - Otherwise, we might need to scale by the ratio
- Exactly what polygon offset's "slope" depth bias does

Results

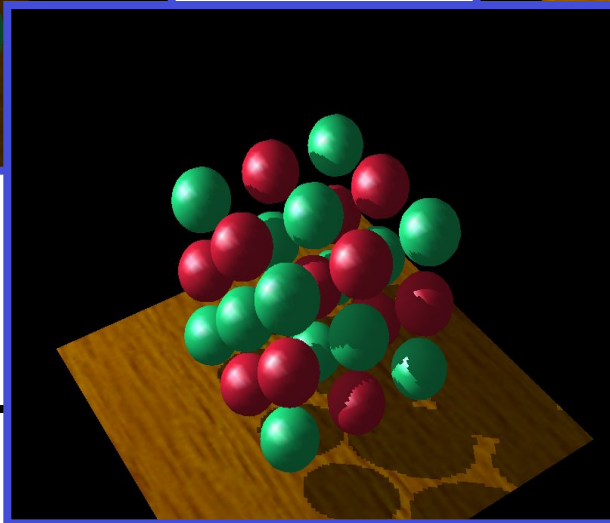
How much polygon offset bias depends



*Too little bias,
everything begins to
shadow*



*Too much bias, shadow
starts too far back*



Selecting Depth Map Bias

- Not that hard
 - Usually the following works well
 - `glPolygonOffset(scale = 1.1, bias = 4.0)`
 - Usually better to error on the side of too much bias
 - adjust to suit the shadow issues in your scene
 - Depends somewhat on shadow map precision
 - more precision requires less of a bias
 - When the shadow map is being magnified, a larger scale is often required

Result



Figure 7.10 Depth rendering



DEPARTMENT OF
COMPUTER SCIENCE
AND ENGINEERING

Using Shadow Map

Matrices

Example 7.19 Matrix Calculations for Shadow Map Rendering

```
mat4 scene_model_matrix = rotate(t * 720.0f, Y);
mat4 scene_view_matrix = translate(0.0f, 0.0f, -300.0f);
mat4 scene_projection_matrix = frustum(-1.0f, 1.0f, -aspect, aspect,
                                       1.0f, FRUSTUM_DEPTH);

mat4 scale_bias_matrix = mat4(vec4(0.5f, 0.0f, 0.0f, 0.0f),
                              vec4(0.0f, 0.5f, 0.0f, 0.0f),
                              vec4(0.0f, 0.0f, 0.5f, 0.0f),
                              vec4(0.5f, 0.5f, 0.5f, 1.0f));

mat4 shadow_matrix = scale_bias_matrix *
                    light_projection_matrix *
                    light_view_matrix;
```



Vertex Shader

Example 7.20 Vertex Shader for Rendering from Shadow Maps

```
#version 330 core

uniform mat4 model_matrix;
uniform mat4 view_matrix;
uniform mat4 projection_matrix;

uniform mat4 shadow_matrix;

layout (location = 0) in vec4 position;
layout (location = 1) in vec3 normal;

out VS_FS_INTERFACE
{
    vec4 shadow_coord;
    vec3 world_coord;
    vec3 eye_coord;
    vec3 normal;
} vertex;

void main(void)
{
    vec4 world_pos = model_matrix * position;
    vec4 eye_pos = view_matrix * world_pos;
    vec4 clip_pos = projection_matrix * eye_pos;

    vertex.world_coord = world_pos.xyz;
    vertex.eye_coord = eye_pos.xyz;
    vertex.shadow_coord = shadow_matrix * world_pos;

    vertex.normal = mat3(view_matrix * model_matrix) * normal;

    gl_Position = clip_pos;
}
```



Transforms

$$\begin{bmatrix} x_e \\ y_e \\ z_e \\ w_e \end{bmatrix} = \begin{bmatrix} \text{Eye} \\ \text{view} \\ \text{(look at)} \\ \text{matrix} \end{bmatrix} \begin{bmatrix} \text{Modeling} \\ \text{matrix} \end{bmatrix} \begin{bmatrix} x_o \\ y_o \\ z_o \\ w_o \end{bmatrix}$$

Map Use

$$\begin{bmatrix} s \\ t \\ r \\ q \end{bmatrix} = \begin{bmatrix} 1/2 & 1/2 \\ & 1/2 & 1/2 \\ & & 1/2 & 1/2 \\ & & & 1 \end{bmatrix} \begin{bmatrix} \text{Light} \\ \text{frustum} \\ \text{(projection)} \\ \text{matrix} \end{bmatrix} \begin{bmatrix} \text{Light} \\ \text{view} \\ \text{(look at)} \\ \text{matrix} \end{bmatrix} \begin{bmatrix} \text{Inverse} \\ \text{eye} \\ \text{view} \\ \text{(look at)} \\ \text{matrix} \end{bmatrix} \begin{bmatrix} x_e \\ y_e \\ z_e \\ w_e \end{bmatrix}$$



DEPARTMENT OF
COMPUTER SCIENCE
AND ENGINEERING

Map Generation

Fragment Shader

Example 7.21 Fragment Shader for Rendering from Shadow Maps

```
#version 330 core

uniform sampler2DShadow depth_texture;
uniform vec3 light_position;

uniform vec3 material_ambient;
uniform vec3 material_diffuse;
uniform vec3 material_specular;
uniform float material_specular_power;

layout (location = 0) out vec4 color;

in VS_FS_INTERFACE
{
    vec4 shadow_coord;
    vec3 world_coord;
    vec3 eye_coord;
    vec3 normal;
} fragment;

void main(void)
{
    vec3 N = fragment.normal;
    vec3 L = normalize(light_position - fragment.world_coord);
    vec3 R = reflect(-L, N);
    vec3 E = normalize(fragment.eye_coord);
    float NdotL = dot(N, L);
    float EdotR = dot(-E, R);

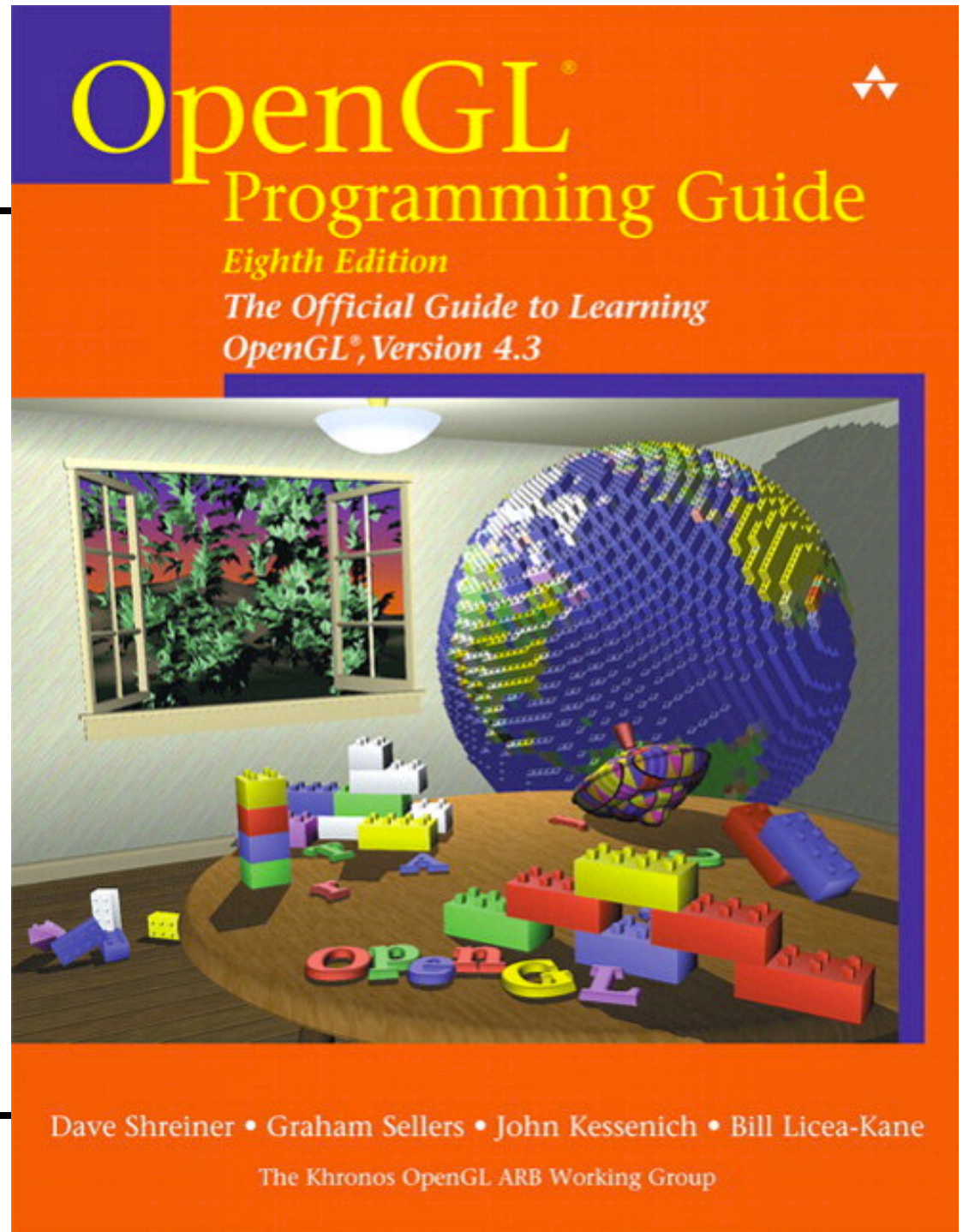
    float diffuse = max(NdotL, 0.0);
    float specular = max(pow(EdotR, material_specular_power), 0.0);

    float f = textureProj(depth_texture, fragment.shadow_coord);

    color = vec4(material_ambient +
                  f * (material_diffuse * diffuse +
                      material_specular * specular), 1.0);
}
```



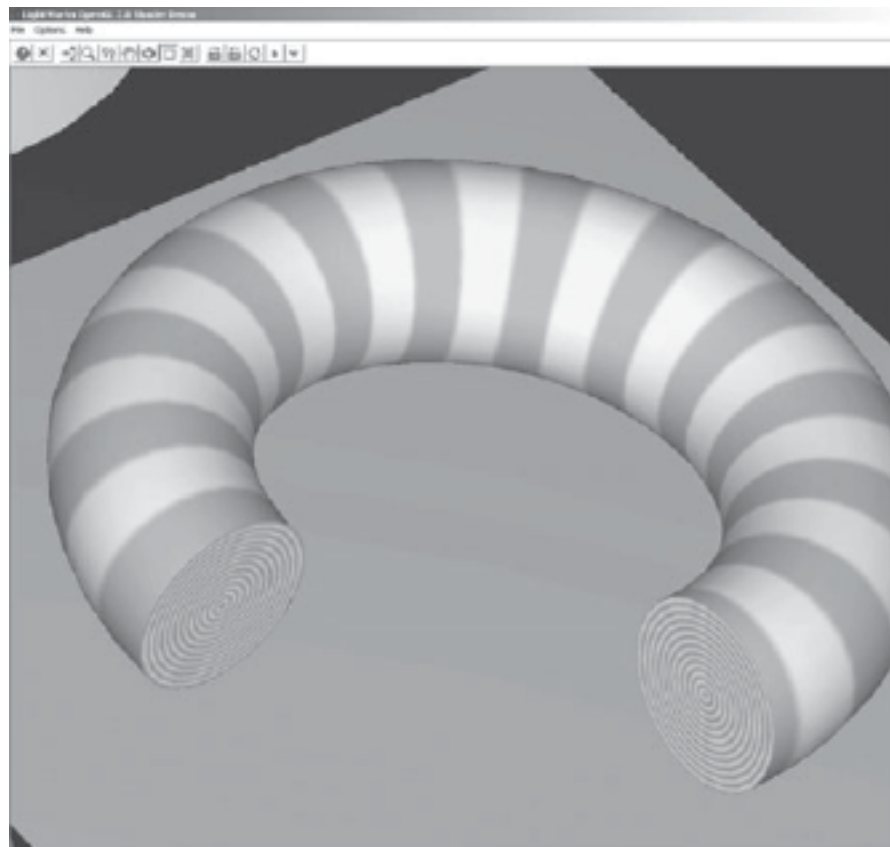
Chapter 8



DEPARTMENT OF
COMPUTER SCIENCE
AND ENGINEERING

Procedural Texturing

Regular Patterns



Vertex Shader

Example 8.1 Vertex Shader for Drawing Stripes

```
#version 330 core

uniform vec3 LightPosition;
uniform vec3 LightColor;
uniform vec3 EyePosition;
uniform vec3 Specular;
uniform vec3 Ambient;

uniform float Kd;
uniform mat4 MVMMatrix;
uniform mat4 MVPMatrix;
uniform mat3 NormalMatrix;

in vec4    MCVertex;
in vec3    MCNormal;
in vec2    TexCoord0;

out vec3    DiffuseColor;
out vec3    SpecularColor;
out float   TexCoord;

void main()
{
    vec3 ecPosition = vec3(MVMMatrix * MCVertex);
    vec3 tnorm      = normalize(NormalMatrix * MCNormal);
    vec3 lightVec    = normalize(LightPosition - ecPosition);
    vec3 viewVec     = normalize(EyePosition - ecPosition);
    vec3 hvec        = normalize(viewVec + lightVec);

    float spec = clamp(dot(hvec, tnorm), 0.0, 1.0);
    spec = pow(spec, 16.0);

    DiffuseColor    = LightColor * vec3(Kd * dot(lightVec, tnorm));
    DiffuseColor    = clamp(Ambient + DiffuseColor, 0.0, 1.0);
    SpecularColor   = clamp((LightColor * Specular * spec), 0.0, 1.0);
    TexCoord        = TexCoord0.t;
    gl_Position     = MVPMatrix * MCVertex;
}
```



Anti-aliasing



Figure 8.2 Stripes close-up

(Extreme close-up view of one of the stripes that shows the effect of the "fuzz" calculation from the stripe shader (courtesy of LightWork Design).)



DEPARTMENT OF
COMPUTER SCIENCE
AND ENGINEERING

Fragment Shader

Example 8.2 Fragment Shader for Drawing Stripes

```
#version 330 core

uniform vec3  StripeColor;
uniform vec3  BackColor;

uniform float Width;
uniform float Fuzz;
uniform float Scale;

in vec3  DiffuseColor;
in vec3  SpecularColor;
in float TexCoord;

out vec4 FragColor;

void main()
{
    float scaledT = fract(TexCoord * Scale);
    float frac1 = clamp(scaledT / Fuzz, 0.0, 1.0);
    float frac2 = clamp((scaledT - Width) / Fuzz, 0.0, 1.0);

    frac1 = frac1 * (1.0 - frac2);
    frac1 = frac1 * frac1 * (3.0 - (2.0 * frac1));

    vec3 finalColor = mix(BackColor, StripeColor, frac1);
    finalColor = finalColor * DiffuseColor + SpecularColor;
    FragColor = vec4(finalColor, 1.0);
}
```

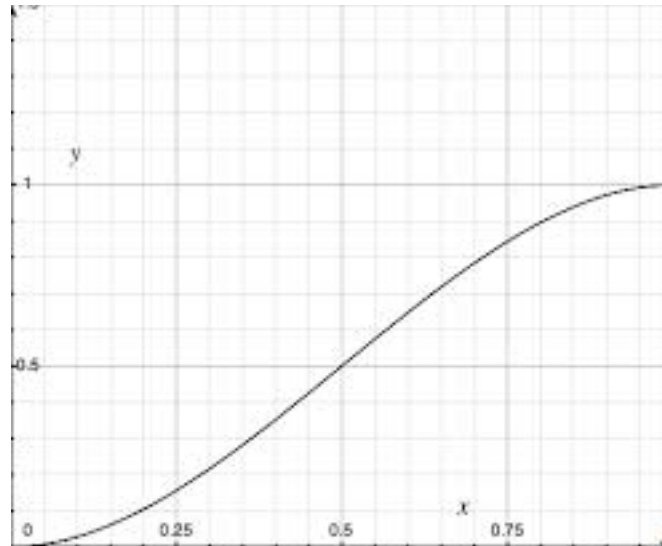


Hermite
Interpolation



DEPARTMENT
COMPUTER SCIENCE
AND ENGINEERING

Hermite



$$2t^3 - 3t^2 + 1$$



DEPARTMENT OF
COMPUTER SCIENCE
AND ENGINEERING

The Brick Wall

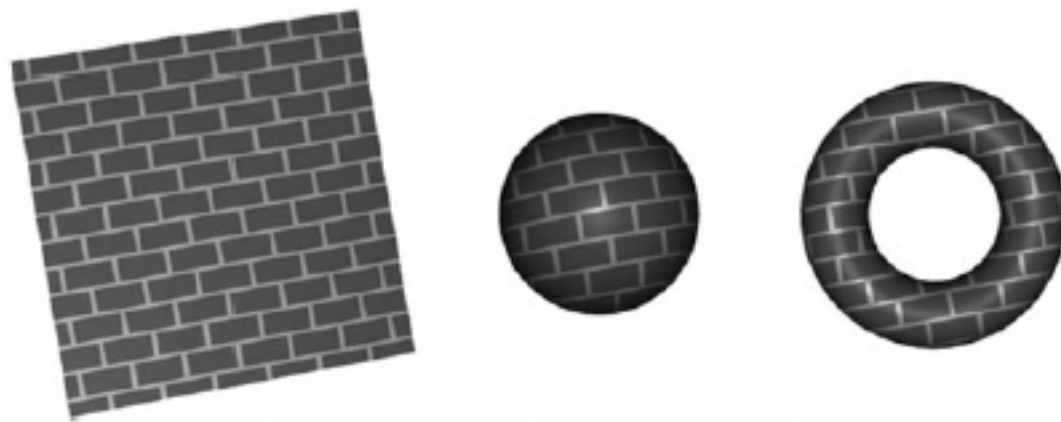


Figure 8.3 Brick patterns
(A flat polygon, a sphere, and a torus rendered with the brick shaders.)



Example 8.3 Vertex Shader for Drawing Bricks

```
#version 330 core

in vec4      MCvertex;
in vec3      MCnormal;

uniform mat4  MVMatrix;
uniform mat4  MVPMatrix;
uniform mat3  NormalMatrix;
uniform vec3  LightPosition;

const float  SpecularContribution = 0.3;
const float  DiffuseContribution  = 1.0 - SpecularContribution;

out float     LightIntensity;
out vec2      MCposition;

void main()
{
    vec3 ecPosition = vec3(MVMatrix * MCvertex);
    vec3 tnorm      = normalize(NormalMatrix * MCnormal);
    vec3 lightVec    = normalize(LightPosition - ecPosition);
    vec3 reflectVec  = reflect(-lightVec, tnorm);
    vec3 viewVec     = normalize(-ecPosition);
    float diffuse    = max(dot(lightVec, tnorm), 0.0);
```

Bricks – Vertex Shader

```
float spec      = 0.0;

if (diffuse > 0.0)
{
    spec = max(dot(reflectVec, viewVec), 0.0);
    spec = pow(spec, 16.0);
}

LightIntensity = DiffuseContribution * diffuse +
                SpecularContribution * spec;

MCposition      = MCvertex.xy;
gl_Position     = MVPMatrix * MCvertex;
}
```



Example 8.4 Fragment Shader for Drawing Bricks

```
#version 330 core

uniform vec3 BrickColor, MortarColor;
uniform vec2 BrickSize;
uniform vec2 BrickPct;

in vec2   MCposition;
in float  LightIntensity;

out vec4  FragColor;

void main()
{
    vec3 color;
    vec2 position, useBrick;

    position = MCposition / BrickSize;

    if (fract(position.y * 0.5) > 0.5)
        position.x += 0.5;

    position = fract(position);
    useBrick = step(position, BrickPct);

    color = mix(MortarColor, BrickColor, useBrick.x * useBrick.y);
    color *= LightIntensity;

    FragColor = vec4(color, 1.0);
}
```



Lattice

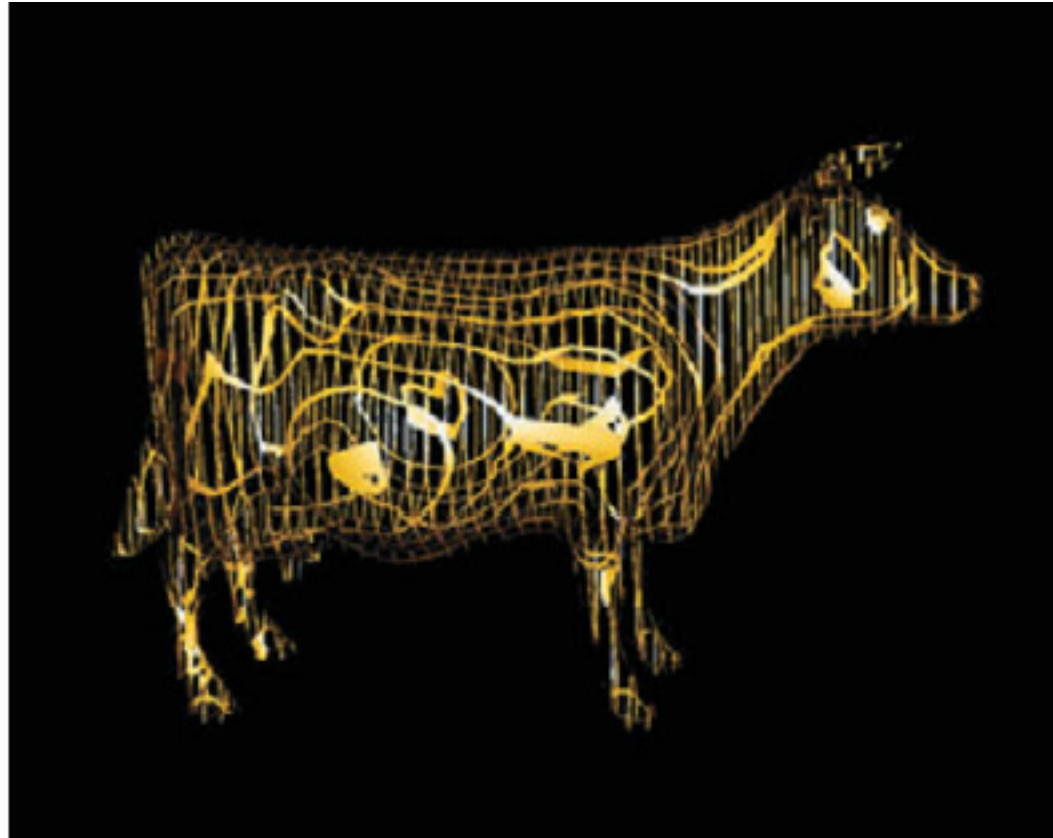


Figure 8.7 The lattice shader applied to the cow model
(3Dlabs, Inc.)



DEPARTMENT OF
COMPUTER SCIENCE
AND ENGINEERING

Discarding Cow Parts

Example 8.8 Fragment Shader for Procedurally Discarding Part of an Object

```
in vec3 DiffuseColor;
in vec3 SpecularColor;
in vec2 TexCoord

out vec3 FragColor;

uniform vec2  Scale;
uniform vec2  Threshold;
uniform vec3  SurfaceColor;

void main()
{
    float ss = fract(TexCoord.s * Scale.s);
    float tt = fract(TexCoord.t * Scale.t);

    if ((ss > Threshold.s) && (tt > Threshold.t))
        discard;

    vec3 finalColor = SurfaceColor * DiffuseColor + SpecularColor;
    FragColor = vec4(finalColor, 1.0);
}
```



Noise



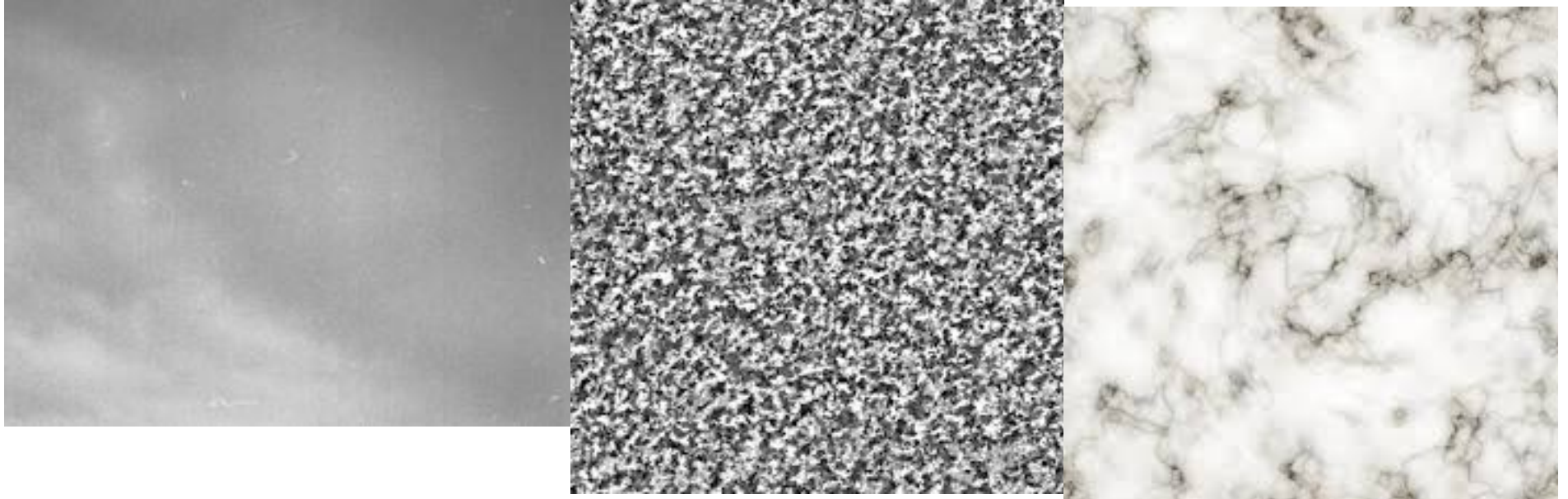
Noise



Tone



Noise Textures

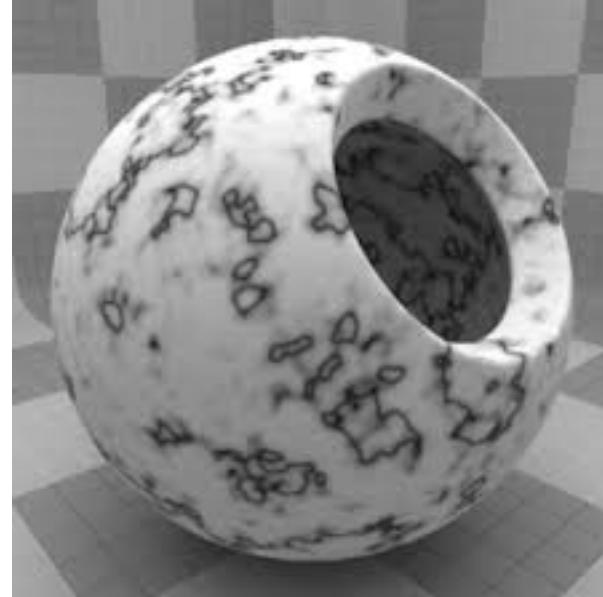


Noise is Seasoning

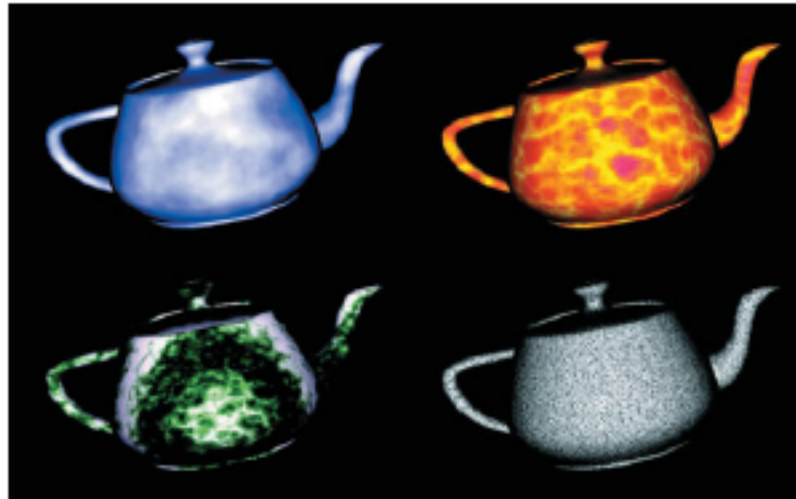
- It does not show any obvious regular or repeated patterns.
- It is a continuous function, and its derivative is also continuous. That is, there are no sudden steps or sharp bends, only smooth variation, and zooming in to smaller and smaller scales still shows only smooth variation.
- It is a function that is repeatable across time (i.e., it generates the same value each time it is presented with the same input).
- It has a well-defined range of output values (usually the range is $[-1, 1]$ or $[0, 1]$).
- It is a function whose small-scale form is roughly independent of large-scale position (there is an underlying frequency to variation, or statistical character, that is the same everywhere).
- It is a function that is isotropic (its statistical character is the same in all directions).
- It can be defined for 1, 2, 3, 4, or even more dimensions.
- It is fast to compute for any given input.



Noisy Texture on Surfaces



RedBook



Discrete Noise



Figure 8.21 A discrete 1D noise function



Continuous Noise

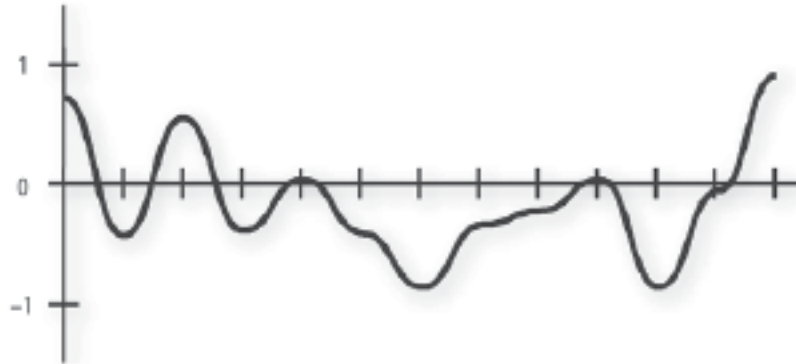


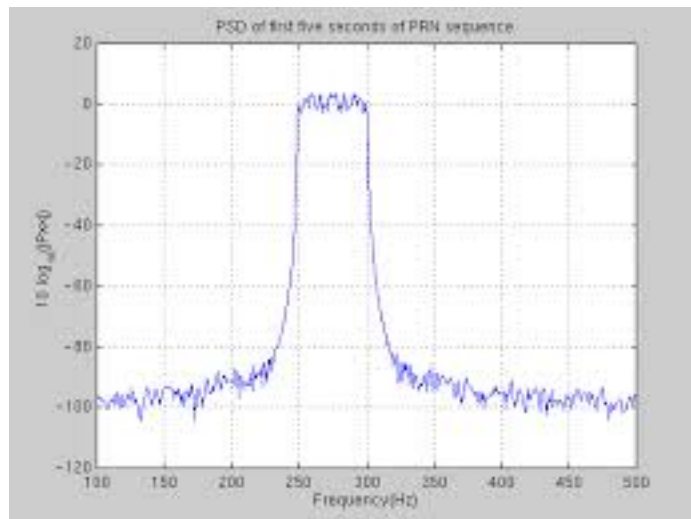
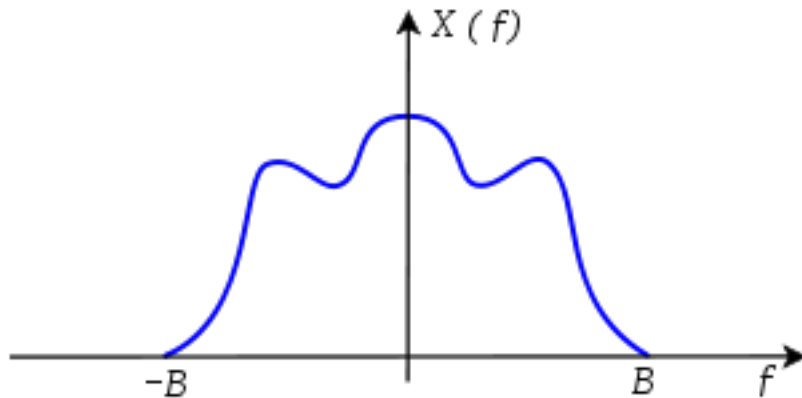
Figure 8.22 A continuous 1D noise function

Interpolation



DEPARTMENT OF
COMPUTER SCIENCE
AND ENGINEERING

Bandlimited Noise



frequency = 4
amplitude = 1.0



frequency = 8
amplitude = 0.5



frequency = 16
amplitude = 0.25



frequency = 32
amplitude = 0.125



frequency = 64
amplitude = 0.0625



Figure 8.23 Varying the frequency and the amplitude of the noise function



DEPARTMENT OF
COMPUTER SCIENCE
AND ENGINEERING

Making Synthetic Noise

Perlin Noise Generator

$$fnoise(x) = \sum_{i=0}^{octaves-1} \alpha^i \cdot noise(2^i \cdot x)$$

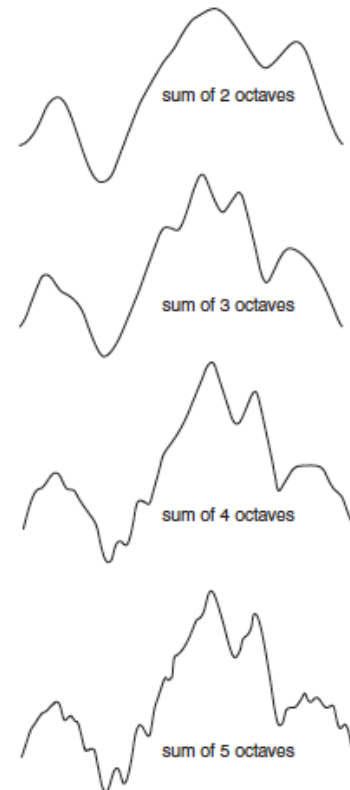


Figure 8.24 Summing noise functions
(Shows the result of summing noise functions of different amplitude and frequency.)



DEPARTMENT OF
COMPUTER SCIENCE
AND ENGINEERING

2D

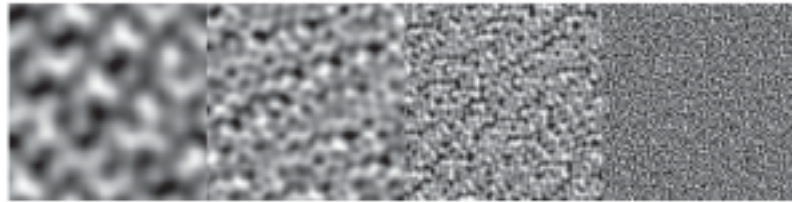


Figure 8.25 Basic 2D noise, at frequencies 4, 8, 16, and 32 (contrast enhanced)

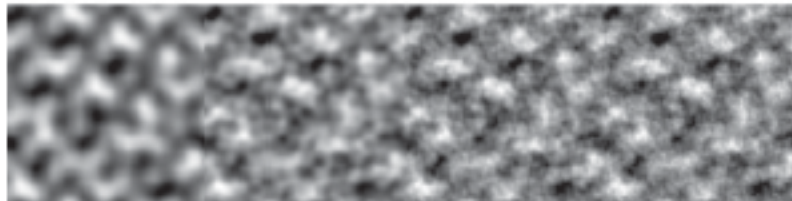


Figure 8.26 Summed noise, at 1, 2, 3, and 4 octaves (contrast enhanced)



3D, nD

$$fnoise(x) = \sum_{i=0}^{octaves-1} \alpha^i \cdot noise(2^i \cdot x)$$

Example 8.14 C function to Generate a 3D Noise Texture

```
int noise3DTexSize = 128;

GLuint noise3DTexName = 0;
GLubyte *noise3DTexPtr;

void make3DNoiseTexture(void)
{
    int f, i, j, k, inc;
    int startFrequency = 4;
    int numOctaves = 4;
    double ni[3];
    double inci, incj, inck;
    int frequency = startFrequency;
    GLubyte *ptr;
    double amp = 0.5;

    if ((noise3DTexPtr = (GLubyte *) malloc(noise3DTexSize *
                                           noise3DTexSize *
                                           noise3DTexSize * 4))
        == NULL)
    {
        fprintf(stderr,
            "ERROR: Could not allocate 3D noise texture\n");
        exit(1);
    }

    for (f = 0, inc = 0; f < numOctaves;
        ++f, frequency *= 2, ++inc, amp *= 0.5)
    {
        setNoiseFrequency(frequency);
        ptr = noise3DTexPtr;
        ni[0] = ni[1] = ni[2] = 0;

        inci = 1.0 / (noise3DTexSize / frequency);
        for (i = 0; i < noise3DTexSize; ++i, ni[0] += inci)
        {
            incj = 1.0 / (noise3DTexSize / frequency);
            for (j = 0; j < noise3DTexSize; ++j, ni[1] += incj)
            {
                inck = 1.0 / (noise3DTexSize / frequency);
                for (k = 0; k < noise3DTexSize;
                    ++k, ni[2] += inck, ptr += 4)
                {
                    *(ptr+inc) = (GLubyte)((noise3(ni)+1.0) * amp)
                        + 128.0;
                }
            }
        }
    }
}
```



Repeatability

Example 8.15 A Function for Activating the 3D Noise Texture

```
void init3DNoiseTexture()
{
    glGenTextures(1, & noise3DTexName);

    glActiveTexture(GL_TEXTURE6);
    glBindTexture(GL_TEXTURE_3D, noise3DTexName);
    glTexParameterf(GL_TEXTURE_3D, GL_TEXTURE_WRAP_S, GL_REPEAT);
    glTexParameterf(GL_TEXTURE_3D, GL_TEXTURE_WRAP_T, GL_REPEAT);
    glTexParameterf(GL_TEXTURE_3D, GL_TEXTURE_WRAP_R, GL_REPEAT);
    glTexParameterf(GL_TEXTURE_3D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
    glTexParameterf(GL_TEXTURE_3D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);

    glTexImage3D(GL_TEXTURE_3D, 0, GL_RGBA, noise3DTexSize,
                noise3DTexSize, noise3DTexSize, 0, GL_RGBA,
                GL_UNSIGNED_BYTE, noise3DTexPtr);
}
```



Noise in GLSL

Example 8.16 Cloud Vertex Shader

```
#version 330 core

uniform mat4 MVMatrix;
uniform mat4 MVPMatrix;
uniform mat3 NormalMatrix;

uniform vec3 LightPos;
uniform float Scale;

in  vec4 MCVertex;
in  vec3 MCNormal;

out float LightIntensity;
out vec3 MCposition;

void main()
{
    vec3 ECposition = vec3(MVMatrix * MCVertex);
    MCposition      = vec3(MCVertex) * Scale;
    vec3 tnorm      = normalize(vec3(NormalMatrix * MCNormal));
    LightIntensity  = dot(normalize(LightPos - ECposition), tnorm);
    LightIntensity *= 1.5;
    gl_Position     = MVPMatrix * MCVertex;
}
```



Fragment Shader

Example 8.17 Fragment Shader for Cloudy Sky Effect

```
#version 330 core

uniform sampler3D Noise;
uniform vec3 SkyColor;      // (0.0, 0.0, 0.8)
uniform vec3 CloudColor;    // (0.8, 0.8, 0.8)

in float LightIntensity;
in vec3 MCposition;

out vec4 FragColor;
```

Chapter 8: Procedural Texturing



```
void main()
{
    vec4 noisevec = texture(Noise, MCposition);

    float intensity = (noisevec[0] + noisevec[1] +
                      noisevec[2] + noisevec[3] + 0.03125) * 1.5;

    vec3 color = mix(SkyColor, CloudColor, intensity) *
                 LightIntensity;
    FragColor = vec4(color, 1.0);
}
```



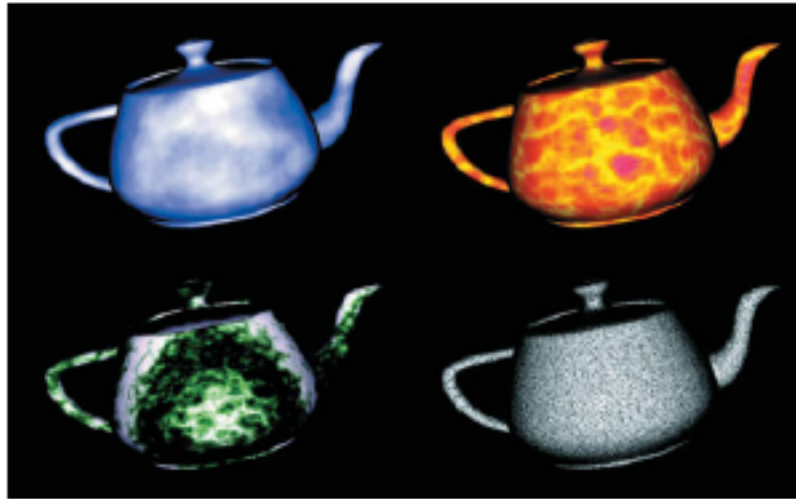


Figure 8.27 Teapots rendered with noise shaders
(Clockwise from upper left: a cloud shader that sums four octaves of noise and uses a blue-to-white color gradient to code the result; a sun surface shader that uses the absolute value function to introduce discontinuities (turbulence); a granite shader that uses a single high-frequency noise value to modulate between white and black; a marble shader that uses noise to modulate a sine function to produce alternating “veins” of color. (3Dlabs, Inc.))



Turbulence

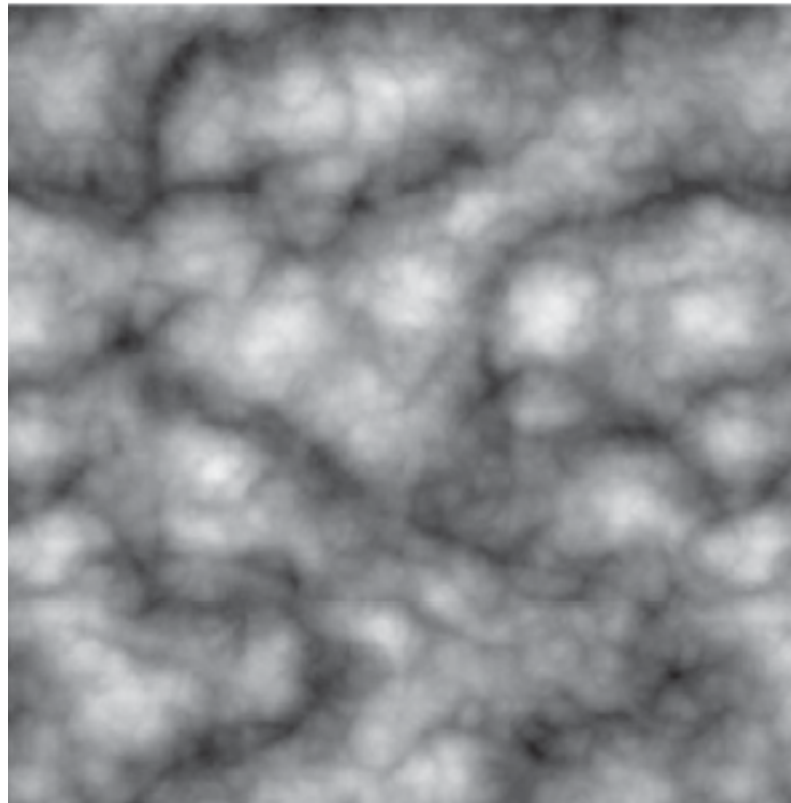


Figure 8.28 Absolute value noise or “turbulence”



DEPARTMENT OF
COMPUTER SCIENCE
AND ENGINEERING

Sun Surface

Example 8.18 Sun Surface Fragment Shader

```
#version 330 core

in float LightIntensity;
in vec3 MCposition;

uniform sampler3D Noise;
uniform vec3 Color1;      // (0.8, 0.7, 0.0)
uniform vec3 Color2;      // (0.6, 0.1, 0.0)
uniform float NoiseScale; // 1.2

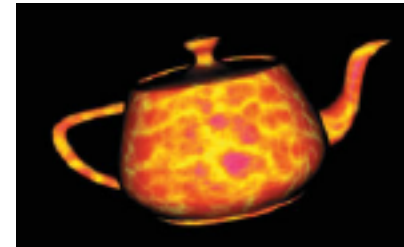
out vec4 FragColor;

void main()
{
    vec4 noisevec = texture(Noise, MCposition * NoiseScale);

    float intensity = abs(noisevec[0] - 0.25) +
                     abs(noisevec[1] - 0.125) +
                     abs(noisevec[2] - 0.0625) +
                     abs(noisevec[3] - 0.03125);

    intensity = clamp(intensity * 6.0, 0.0, 1.0);
    vec3 color = mix(Color1, Color2, intensity) * LightIntensity;
    FragColor = vec4(color, 1.0);
}
```

We can achieve an effect that looks like a pit of hot molten lava or the surface of the sun by using the same vertex shader as the cloud shader and a slightly different fragment shader. The main difference is that we scale each noise value and shift it over so that it is centered at 0; then we take its absolute value. After summing the values, we scale the result again to occupy nearly the full range of [0, 1]. We clamp this value and use it to mix between yellow and red to get the result shown in Figure 8.27 (see Example 8.18). This technique can be extended to change the results over time, using another dimension of noise for time, resulting in animation of the effect.



Granite

With noise, it's also easy just to try to make stuff up. In this example, we want to simulate a grayish rocky material with small black specks. To generate a relatively high-frequency noise texture, we use only the fourth component (the highest frequency one). We scale it by an arbitrary amount to provide an appropriate intensity level and then use this value for each of the red, green, and blue components. The shader in Example 8.20 generates an appearance similar to granite, as shown in Figure 8.27.

Example 8.20 Granite Fragment Shader

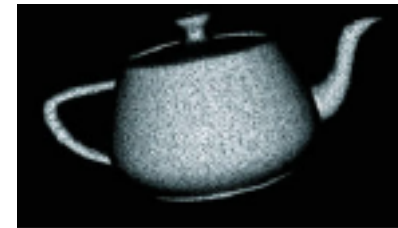
```
#version 330 core

uniform sampler3D Noise;
uniform float NoiseScale;

in float LightIntensity;
in vec3 MCposition;

out vec4 FragColor;

void main()
{
    vec4 noisevec = texture(Noise, NoiseScale * MCposition);
    float intensity = min(1.0, noisevec[3] * 18.0);
    vec3 color = vec3(intensity * LightIntensity);
    FragColor = vec4(color, 1.0);
}
```



Marble

Example 8.19 Fragment Shader for Marble

```
#version 330 core

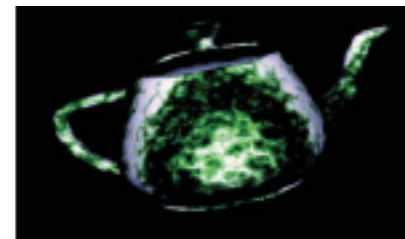
uniform sampler3D Noise;
uniform vec3 MarbleColor;
uniform vec3 VeinColor;

in float LightIntensity;
in vec3 MCposition;

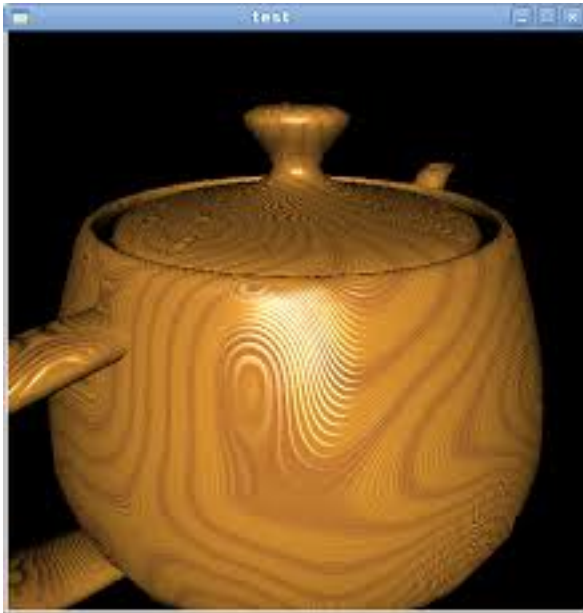
out vec4 FragColor;

void main()
{
    vec4 noisevec = texture(Noise, MCposition);
    float intensity = abs(noisevec[0] - 0.25) +
                     abs(noisevec[1] - 0.125) +
                     abs(noisevec[2] - 0.0625) +
                     abs(noisevec[3] - 0.03125);
    float sineval = sin(MCposition.y * 6.0 + intensity * 12.0)
                  * 0.5 + 0.5;
    vec3 color = mix(VeinColor, MarbleColor, sineval)
               * LightIntensity;
    FragColor = vec4(color, 1.0);
}
```

Yet another variation on the noise function is to use it as part of a periodic function such as sine. By adding noise to the input value for the sine function, we get a “noisy” oscillating function. We use this to create a look similar to the alternating color veins of some types of marble. Example 8.19 shows the fragment shader to do it. Again, we use the same vertex shader. Results of this shader are also shown in Figure 8.27.



Wood



Wood

LightPos	0.0, 0.0, 4.0
Scale	2.0
LightWood	0.6, 0.3, 0.1
DarkWood	0.4, 0.2, 0.07
RingFreq	4.0
LightGrains	1.0
DarkGrains	0.0
GrainThreshold	0.5
NoiseScale	0.5, 0.1, 0.1
Noisiness	3.0
GrainScale	27.0

- Wood is composed of light and dark areas alternating in concentric cylinders surrounding a central axis.
- Noise is added to warp the cylinders to create a more natural-looking pattern.
- The center of the “tree” is taken to be the y axis.
- Throughout the wood, a high-frequency grain pattern gives the appearance of wood that has been sawed, exposing the open grain nature of the wood.



Wood

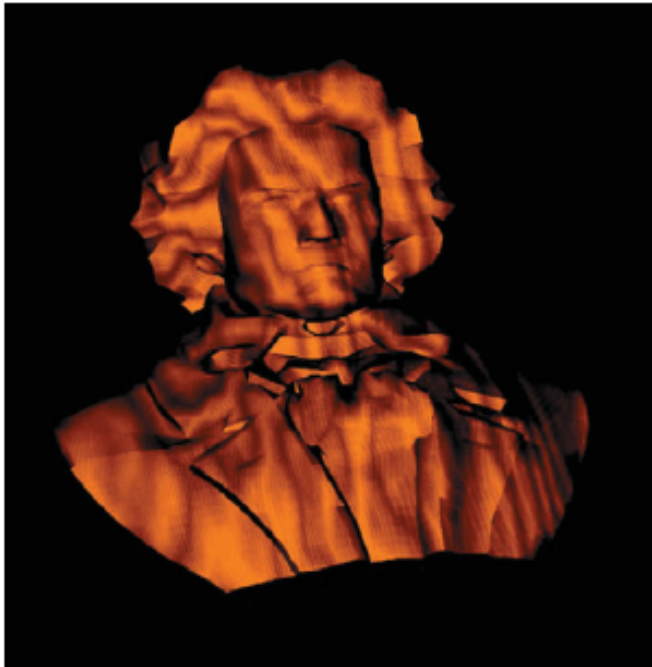


Figure 8.29 A bust of Beethoven rendered with the wood shader (3Dlabs, Inc.)

Example 8.21 Fragment Shader for Wood

```
#version 330 core

uniform sampler3D Noise;
uniform vec3 LightWood;
uniform vec3 DarkWood;
uniform float RingFreq;
uniform float LightGrains;
uniform float DarkGrains;
uniform float GrainThreshold;
uniform vec3 NoiseScale;
uniform float Noisiness;
uniform float GrainScale;

in float LightIntensity;
in vec3 MCposition;

out vec4 FragColor;

void main()
{
    vec3 noisevec = vec3(texture(Noise, MCposition * NoiseScale) *
                          Noisiness);

    vec3 location = MCposition + noisevec;
    float dist = sqrt(location.x * location.x + location.z * location.z);
    dist *= RingFreq;
    float r = fract(dist + noisevec[0] + noisevec[1] + noisevec[2])
              * 2.0;

    if (r > 1.0)
        r = 2.0 - r;
    vec3 color = mix(LightWood, DarkWood, r);
    r = fract((MCposition.x + MCposition.z) * GrainScale + 0.5);
    noisevec[2] *= r;
    if (r < GrainThreshold)
        color += LightWood * LightGrains * noisevec[2];
    else
        color -= LightWood * DarkGrains * noisevec[2];
    color *= LightIntensity;
    FragColor = vec4(color, 1.0);
}
```



DEPARTMENT OF
COMPUTER SCIENCE
AND ENGINEERING

Wood

Example 8.21 Fragment Shader for Wood

```
#version 330 core

uniform sampler3D Noise;
uniform vec3 LightWood;
uniform vec3 DarkWood;
uniform float RingFreq;
uniform float LightGrains;
uniform float DarkGrains;
uniform float GrainThreshold;
uniform vec3 NoiseScale;
uniform float Noisiness;
uniform float GrainScale;

in float LightIntensity;
in vec3 MCposition;

out vec4 FragColor;

void main()
{
    vec3 noisevec = vec3(texture(Noise, MCposition + NoiseScale) *
                          Noisiness);

    vec3 location = MCposition + noisevec;
    float dist = sqrt(location.x * location.x + location.z * location.z);
    dist *= RingFreq;
    float r = fract(dist + noisevec[0] + noisevec[1] + noisevec[2])
              + 2.0;
    if (r > 1.0)
        r = 2.0 - r;
    vec3 color = mix(LightWood, DarkWood, r);
    r = fract((MCposition.x + MCposition.z) * GrainScale + 0.5);
    noisevec[2] *= r;
    if (r < GrainThreshold)
        color += LightWood * LightGrains * noisevec[2];
    else
        color -= LightWood * DarkGrains * noisevec[2];
    color *= LightIntensity;
    FragColor = vec4(color, 1.0);
}
```



ToyBall



(A)



(B)



(C)



(D)

Fragment Shader - TB

Example 8.7 Fragment Shader for Drawing a Toy Ball

```
#version 330 core

uniform vec4  HalfSpace[5]; // half-spaces used to define star pattern
uniform float StripeWidth;
uniform float InOrOutInit;  // -3.0
uniform float FWidth;       // = 0.005

uniform vec4  StarColor;
uniform vec4  StripeColor;
uniform vec4  BaseColor;

uniform vec4  LightDir;      // light direction, should be normalized
uniform vec4  HVector;       // reflection vector for infinite light
```

Procedural Texturing 429



DEPARTMENT OF
COMPUTER SCIENCE
AND ENGINEERING

ToyBall

Example 8.7 Fragment Shader for Drawing a Toy Ball

```
#version 330 core

uniform vec4  HalfSpace[5]; // half-spaces used to define star pattern
uniform float StripeWidth;
uniform float InOrOutInit;  // -3.0
uniform float FWidth;       // = 0.005

uniform vec4  StarColor;
uniform vec4  StripeColor;
uniform vec4  BaseColor;

uniform vec4  LightDir;      // light direction, should be normalized
uniform vec4  HVector;       // reflection vector for infinite light
```

Procedural Texturing 429

Fragment Shader

```
uniform vec4 SpecularColor;
uniform float SpecularExponent;

uniform float Ka;
uniform float Kd;
uniform float Ks;

    in vec4 ECPosition;    // surface position in eye coordinates
    in vec3 OCPosition;    // surface position in object coordinates
flat in vec4 ECBallCenter; // ball center in eye coordinates

out vec4 FragColor;

void main()
{
    vec3 normal;           // Analytically computed normal
    vec4 pShade;           // Point in shader space
    vec4 surfColor;        // Computed color of the surface
    float intensity;       // Computed light intensity
    vec4 distance;         // Computed distance values
    float inorout;         // Counter for classifying star pattern

    pShade.xyz = normalize(OCPosition.xyz);
    pShade.w = 1.0;

    inorout = InOrOutInit;    // initialize inorout to -3.0

    distance[0] = dot(pShade, HalfSpace[0]);
    distance[1] = dot(pShade, HalfSpace[1]);
    distance[2] = dot(pShade, HalfSpace[2]);
    distance[3] = dot(pShade, HalfSpace[3]);

    //float FWidth = fwidth(pShade);
    distance = smoothstep(-FWidth, FWidth, distance);

    inorout += dot(distance, vec4(1.0));

    distance.x = dot(pShade, HalfSpace[4]);
    distance.y = StripeWidth - abs(pShade.z);
    distance.xy = smoothstep(-FWidth, FWidth, distance.xy);
    inorout += distance.x;

    inorout = clamp(inorout, 0.0, 1.0);

    surfColor = mix(BaseColor, StarColor, inorout);
    surfColor = mix(surfColor, StripeColor, distance.y);

    // Calculate analytic normal of a sphere
    normal = normalize(ECPosition.xyz-ECBallCenter.xyz);

    // Per-fragment diffuse lighting
    intensity = Ka; // ambient
    intensity += Kd * clamp(dot(LightDir.xyz, normal), 0.0, 1.0);
```



Fragment Shader

```
surfColor *= intensity;

// Per-fragment specular lighting
intensity = clamp(dot(HVector.xyz, normal), 0.0, 1.0);
intensity = Ks * pow(intensity, SpecularExponent);
surfColor.rgb += SpecularColor.rgb * intensity;
FragColor = surfColor;
}
```



DEPARTMENT OF
COMPUTER SCIENCE
AND ENGINEERING