DigiPen INSTITUTE OF TECHNOLOGY

CS300
Misc. Topics 1

Render to Texture & Projective Texturing

DigiPen
INSTITUTE OF TECHNOLOGY

CS300
Render to Texture

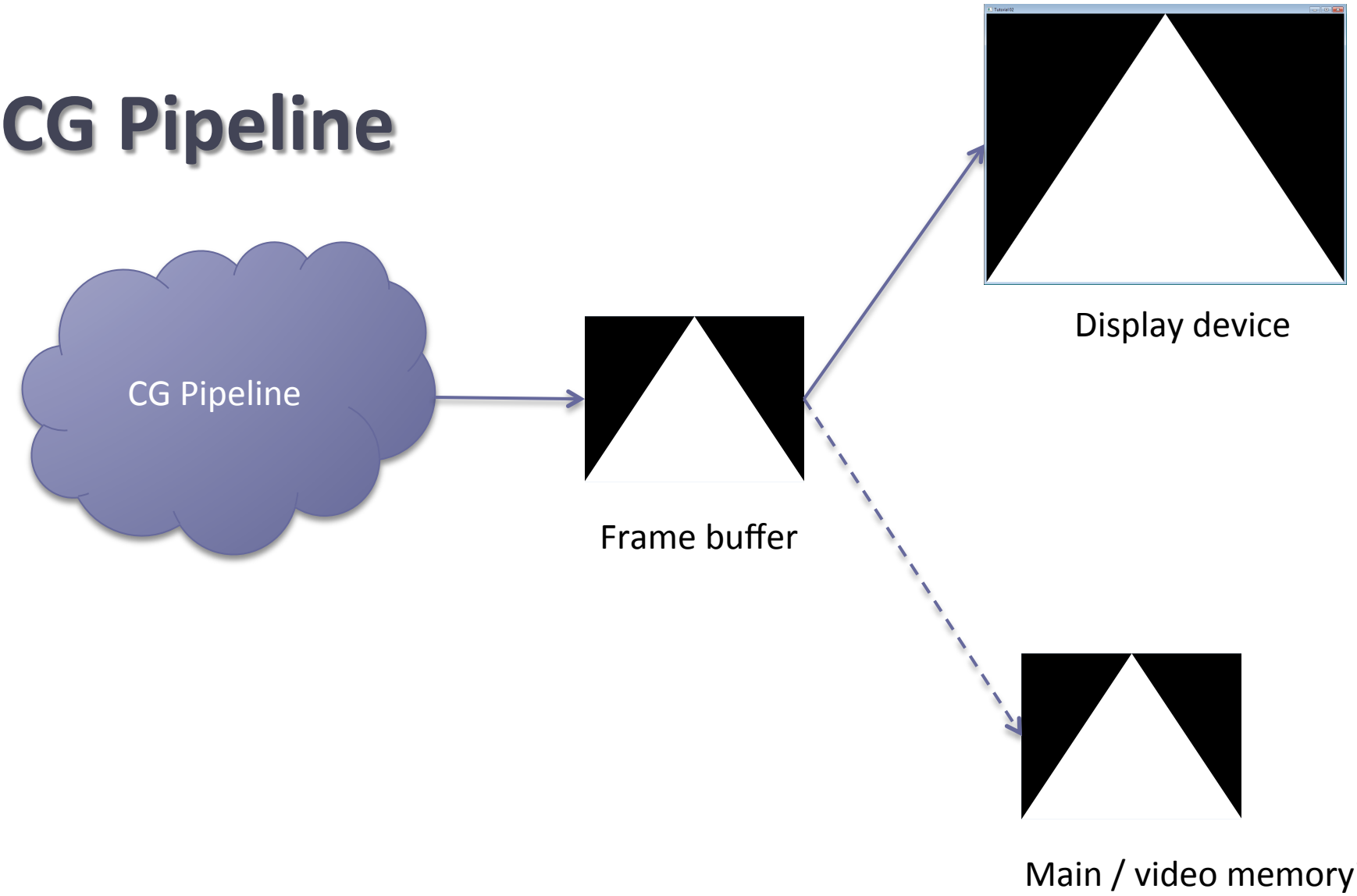
A Swiss-army knife for advanced rendering

CG Rendering pipeline

- Aim
  - Create a 2D representation of a 3D scene
- Final output
  - Special area in video memory – “Frame Buffer”

- Frame Buffer – A block of memory that is polled by hardware for updates
  - Transparent operation as far as the programmer is concerned

CG Pipeline



Using custom frame buffer

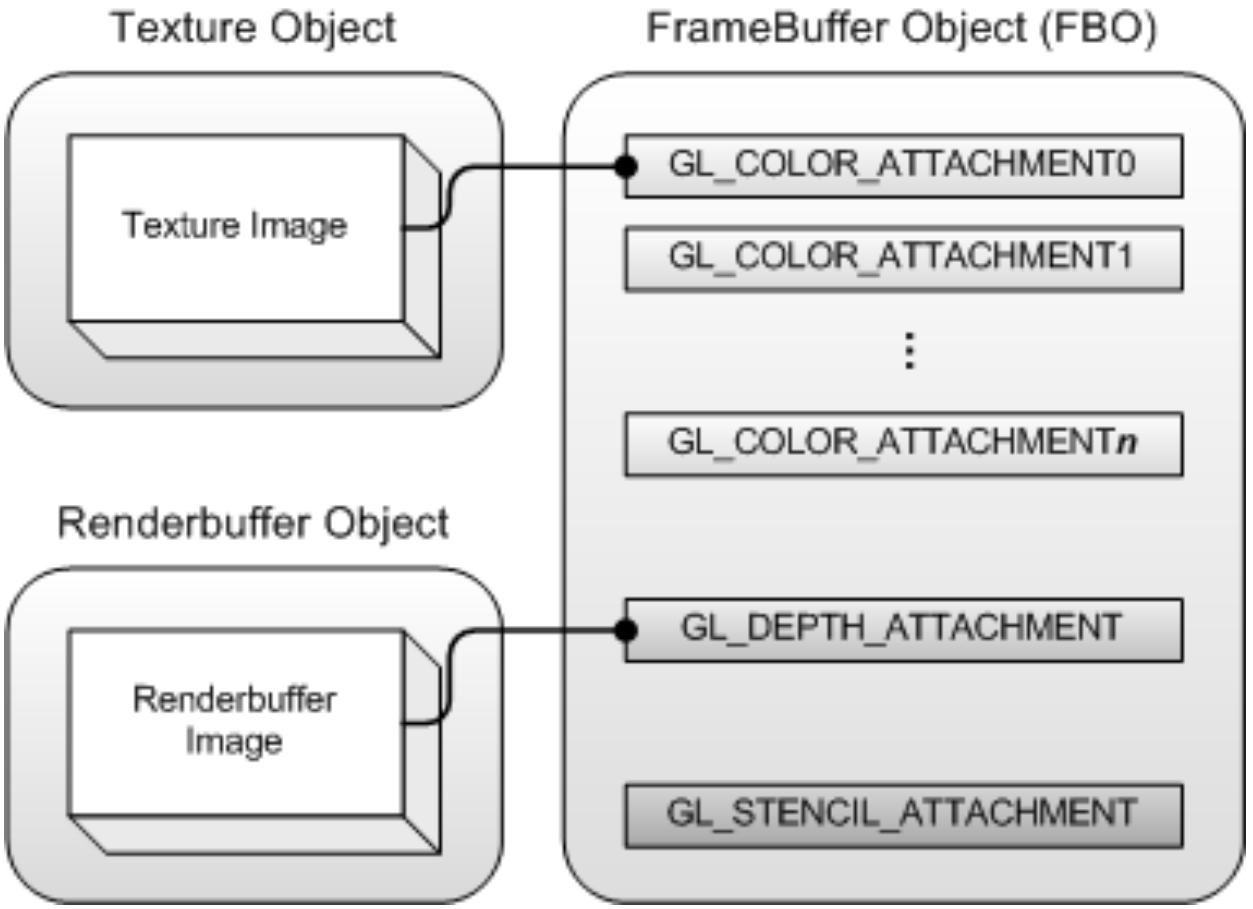
- The FB is not passed on to the display device, but stored in-core (CPU / GPU)
- Can be treated as
  - buffer of data (unstructured array)
  - stream of information (logically aligned data records)
  - visual output (image/texture)

- If the output is interpreted as an image and used in further processing as input → Render-to-texture.
- If output is used as renderbuffer → Offscreen rendering

Applications of Custom FBs

- Off-screen rendering
  - Shadow mapping
- Updating texture data procedurally
  - Video streaming on a 3D billboard
  - Visibility maps
  - Numerical solutions
- Buffer ping-pong
  - Custom pool of buffers other than FRONT and BACK
- And more …

Overall schematic



Frame Buffer Object

- OpenGL data structure that encapsulates what information is passed from OpenGL to underlying windowing system
- Color, stencil and depth values
- Bypass the window-system pipeline
  - Application **MUST** provide all the data storage for applicable slots

Step 1: Get handle to the FB Object

- FrameBuffer – output of our CG pipeline
- No “pointers” on the GPU
  - so create a handle to the FB object

```
GLuint FramebufferName = 0;                    // handle to framebuffer
glGenFramebuffers(1, &FramebufferName);  //  create the FB object handle
glBindFramebuffer(GL_FRAMEBUFFER, FramebufferName); // assign the handle to an
                                                    // object of type GL_FRAMEBUFFER,
                                                    // GL_DRAW_FRAMEBUFFER, or
                                                    // GL_READ_FRAMEBUFFER
```

FBO

- Binding to current handle overwrites any previous binding
  - To revert back to the default FB, use '0'
    - glBindFrameBuffer(GL_FRAMEBUFFER, 0);
- Immediately after binding
  - GL_COLOR_ATTACHMENT0 = GL_NONE
  - GL_DEPTH_ATTACHMENT = GL_NONE
  - GL_STENCIL_ATTACHMENT = GL_NONE
  - No Accumulation Buffer provided

Step 2: Create a texture

- Feedback loop
  - Output of step 1 will be used as input in subsequent passes
- Create standard OpenGL texture
  - width, height = viewport width and height
  - no actual data pointer
    - last argument = '0'
  - Data format – GL_RGB
    - Can customize!
    - Use floating point for HDR Rendering

```
// The texture we're going to render to
GLuint renderedTexture;
glGenTextures(1, &renderedTexture);

// "Bind" the newly created texture
// all future texture functions will modify this texture
glBindTexture(GL_TEXTURE_2D, renderedTexture);

// Give an empty image to OpenGL
glTexImage2D(GL_TEXTURE_2D, 0,GL_RGB, 1024, 768, 0,GL_RGB, GL_UNSIGNED_BYTE, 0);
```

Step 2.5 : Add Depth buffer

- Generate the handle to depth buffer
- Bind it as an object of type GL_RENDERBUFFER

- Set memory storage for width and height of the FB
- Choose data type of this buffer

- Crucial step
  - "Attach" the render buffer as a "depth buffer" to receive depth values from Opengl

```
// The depth buffer
GLuint depthrenderbuffer;

glGenRenderbuffers(1,
&depthrenderbuffer);

glBindRenderbuffer(GL_RENDERBUFFER,
depthrenderbuffer);

glRenderbufferStorage(GL_RENDERBUFFER,
GL_DEPTH_COMPONENT, 1024, 768);

glFramebufferRenderbuffer(
    GL_FRAMEBUFFER,
    GL_DEPTH_ATTACHMENT,
    GL_RENDERBUFFER,
    depthrenderbuffer);
```

Step 3: Now add the “render target”

- Texture pointed to by “renderedTexture” (Step 2) is attached as output of the OpenGL pipeline

```
glGenTextures(1, renderedTexture);
```

```
// Set "renderedTexture" as our colour attachment #0
glFramebufferTexture(
    GL_FRAMEBUFFER,
    GL_COLOR_ATTACHMENT0,
    renderedTexture,
    0);

// Set the list of draw buffers.
GLenum DrawBuffers[1] = {GL_COLOR_ATTACHMENT0};

Gluint numBuffers = 1;
glDrawBuffers(numBuffers, DrawBuffers);
```

Important Issues to consider

- Size and format of FBO is completely controlled by the OpenGL application
- FBOs are not affected by system events
  - window resizing, display mode changes
- The FBOs always return a valid pixel ownership test
- No concept of front and back buffers
  - Have to explicitly set GL_BACK after reverting to default FBO
    - glBindBuffer(GL_DRAW_FRAMEBUFFER,0)
    - glDrawBuffer(GL_BACK_LEFT)
- No multisample buffer (anti aliasing) support

Shader code

- Use layout specification on out variable

```
layout(location=0) out vec3 color
```

location = n, where n is the index of the attached buffer in the glDrawBuffers call

If we add the depth attachment as a render-target, then we can also add the following code:

```
Glenum DrBuffers={GL_COLOR_ATTACHMENT0,
GL_DEPTH_ATTACHMENT, GL_COLOR_ATTACHMENT1}

glDrawBuffers( numBuffers, DrBuffers );
```

Shader code (contd.)

CPU Code

```
Glenum DrBuffers={GL_COLOR_ATTACHMENT0,
GL_DEPTH_ATTACHMENT, GL_COLOR_ATTACHMENT1}
```

GPU Code

```
layout(location = 0) out vec3 color;
layout(location = 1) out float depth;
layout(location = 2) out vec3 normal;
```

Note: The index is the location in the array, NOT the index referred in GL_COLOR_ATTACHMENT*i*

Switching FBOs

- Very expensive – avoid it
- Switch attachment objects on same FBO
  - glFrameBufferTexture2D (for render-to-textures)
  - glFrameBufferRenderbuffer (for renderbuffer objects)

Resources

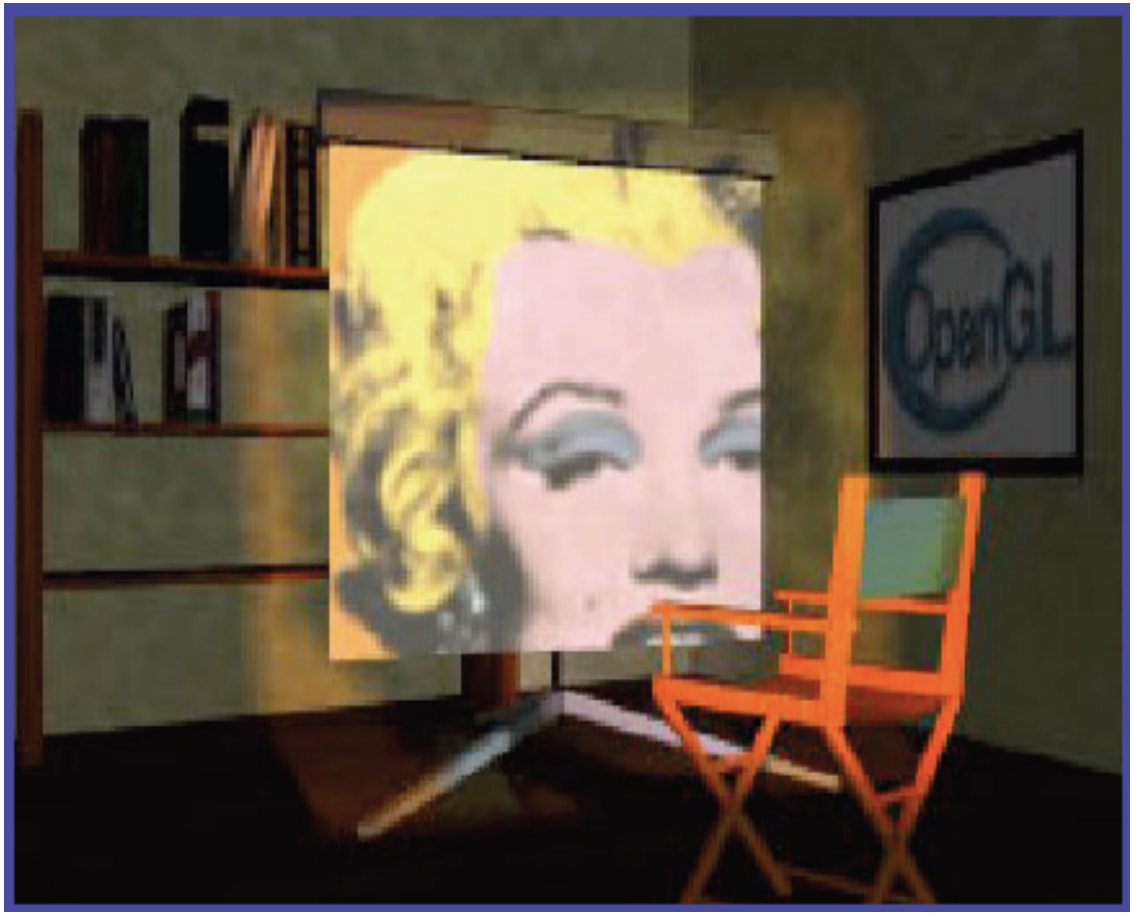
- OpenGL Programming Guide (Red Book), Chapter 4 “Colors, Pixels and Framebuffers”
- http://www.opengl-tutorial.org/
- www.khronos.org, OpenGL Documentation

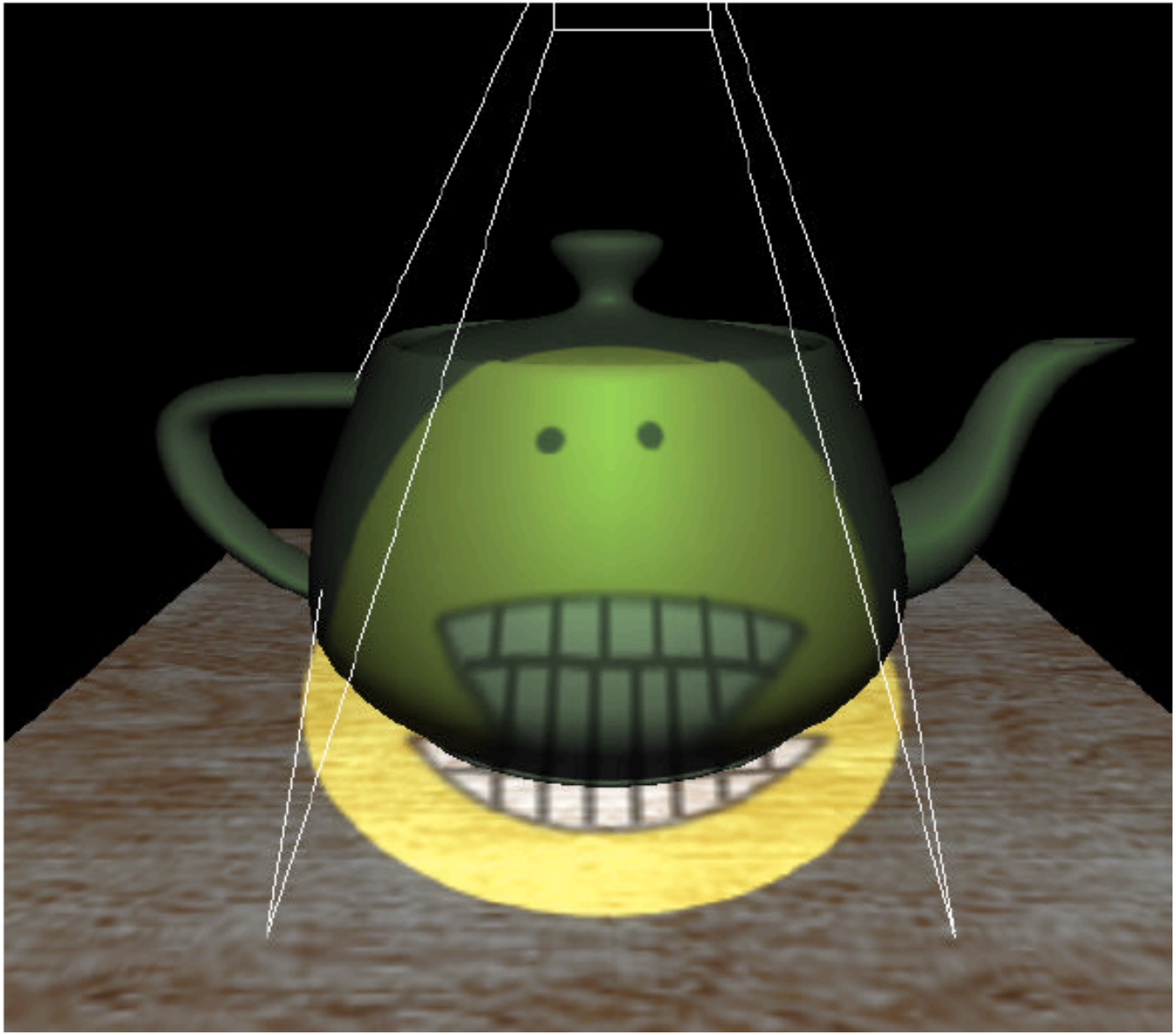
DigiPen
INSTITUTE OF TECHNOLOGY

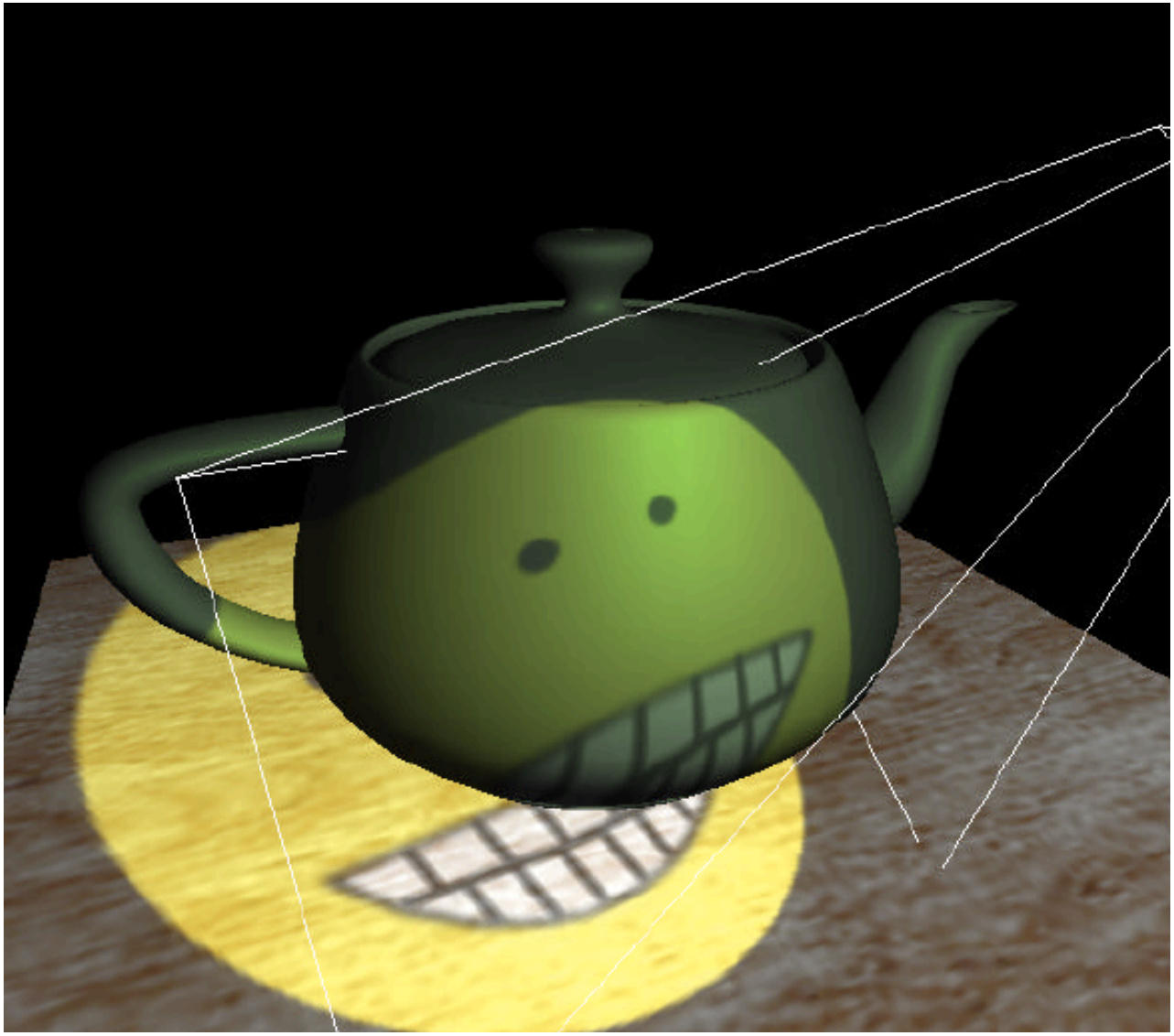
CS300
Projective Texturing

Projective Texturing

Example







What is Projective Texturing?

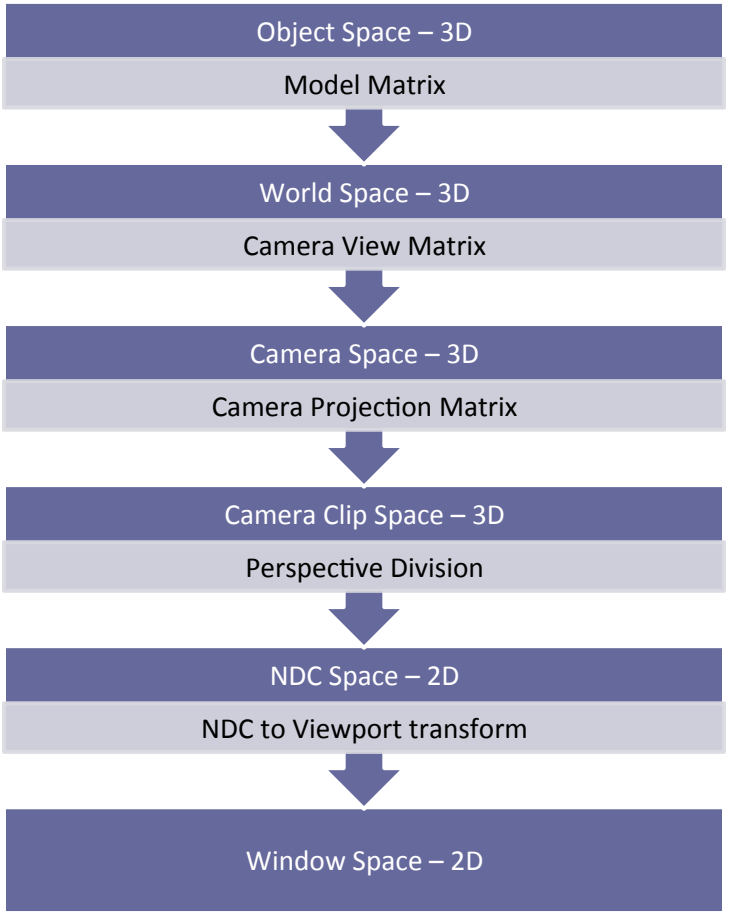
- Projective texturing is a method that projects texture map onto scenes as if it is projected by a slide projector.
- It computes texture coordinate for each vertex and use it to index colors inside the projected texture.
- It refers to how texture coordinates are calculated and how it is interpolated during rasterization.
- It can be used to do unusual lighting or shadowing.

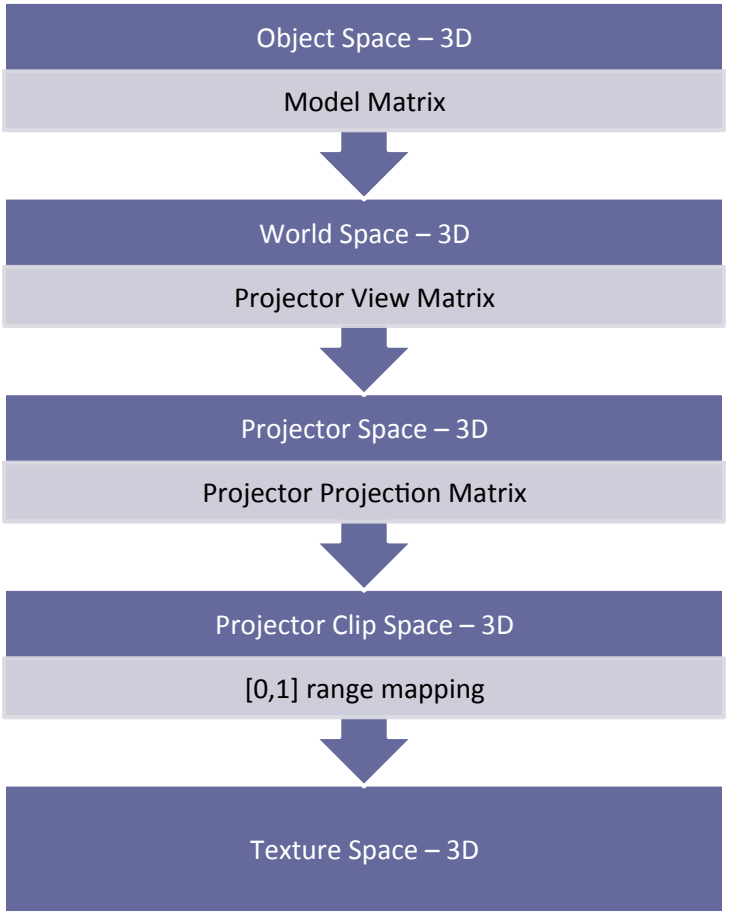
What is Projective Texturing? *(cont'd)*

- In order to do projective texturing, we need:
  - Projector
  - Texture
  - Object(s)

- Projector
  - It has most of the properties of the camera.
  - It has position, direction, field of view, etc.
  - Like the camera matrix, its matrix is used to transform the object vertices to the projector's space.

Camera vs. Projector Transform





- Texture
  - Depends on what it is you are trying to do.
  - Typically, it is one of the following:
    - Light texture
    - Shadow texture
    - Normal RGB texture
- Object(s)
  - The object(s) where the texture will be projected upon.
  - Need the world space transformation matrix.

Texture Coordinates Interpolation

- Use homogeneous texture coordinates
  - OpenGL supports (s, t, p, q) vector of texture coordinates
- The 3 (s, t, p) components of the texture coordinates are interpolated across the primitive
  - 'q' acts as the homogeneous space component 'w' in the graphics rendering pipeline
- For each fragment, the texture coordinates are projected to 2D texture coordinates by dividing by q (the last component) (s/q, t/q, p/q, q/q).
- The 'p' component is not used.

OpenGL behavior

- Texture coordinate values
  - (s,t) supplied by the user (parameter space values)
  - (p,q) have default values (0,1) if not explicitly supplied by the user
- Addition of the 'q' value adds minimal overhead to the fixed-function pipeline
  - Instead of a (1/w) divide for the texture coordinates, OpenGL employs a divide by (q/w)

Texture Coordinates Generation

- Transform the vertices in world space to projector's homogeneous space.

$$T = \begin{bmatrix} 0.5 & 0 & 0 & 0.5 \\ 0 & 0.5 & 0 & 0.5 \\ 0 & 0 & 0.5 & 0.5 \\ 0 & 0 & 0 & 1 \end{bmatrix} * P_p * V_p$$

- Where:
  - T is the matrix to transform vertices in world space to texture coordinates in homogeneous space
  - P_P is the projector's projection matrix
  - V_P is the projector's viewing transform

Texture Coordinates Generation *(cont'd)*

- Notice that the input vertices need to be in world space.
- So, if the source vertices are not in world space, a necessary transformation must be done to take them to world space.
- Basically:
  - if the source vertices are in model space, apply the model-to-world transform.
  - if the source vertices are in camera space , apply the inverse of world-to-camera transform.

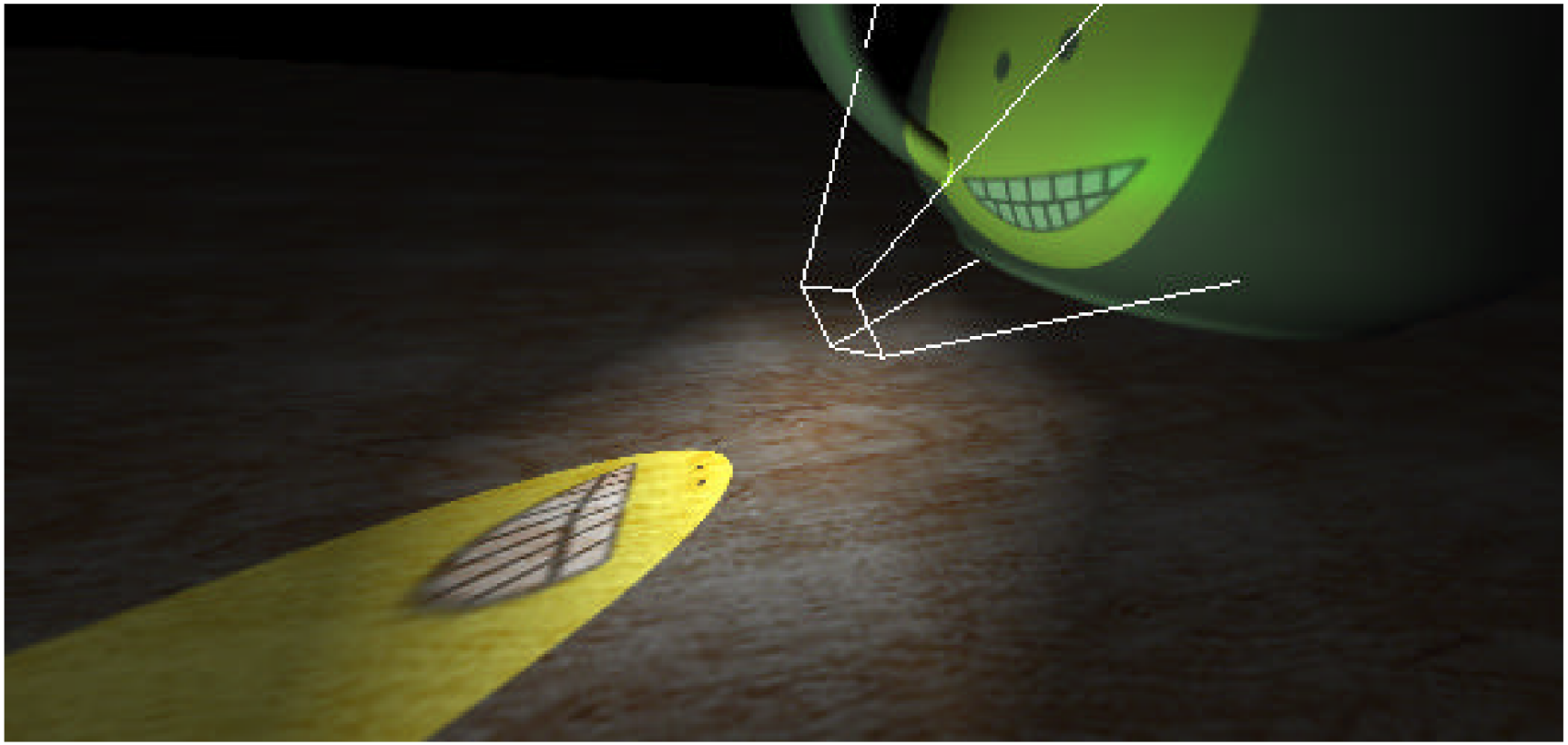
Vertex Shader

```
uniform mat4 TexGenMat;
uniform mat4 InvViewMat;
varying vec3 normal, lightDir, eyeVec;

void main()
{
        normal = gl_NormalMatrix * gl_Normal;

        vec4 posEye =  gl_ModelViewMatrix * gl_Vertex;
        vec4 posWorld = InvViewMat * posEye;
        gl_TexCoord[0] = TexGenMat * posWorld;

        lightDir = vec3(gl_LightSource[0].position.xyz - posEye.xyz);
        eyeVec = -posEye.xyz;

        gl_Position = ftransform();
}
```

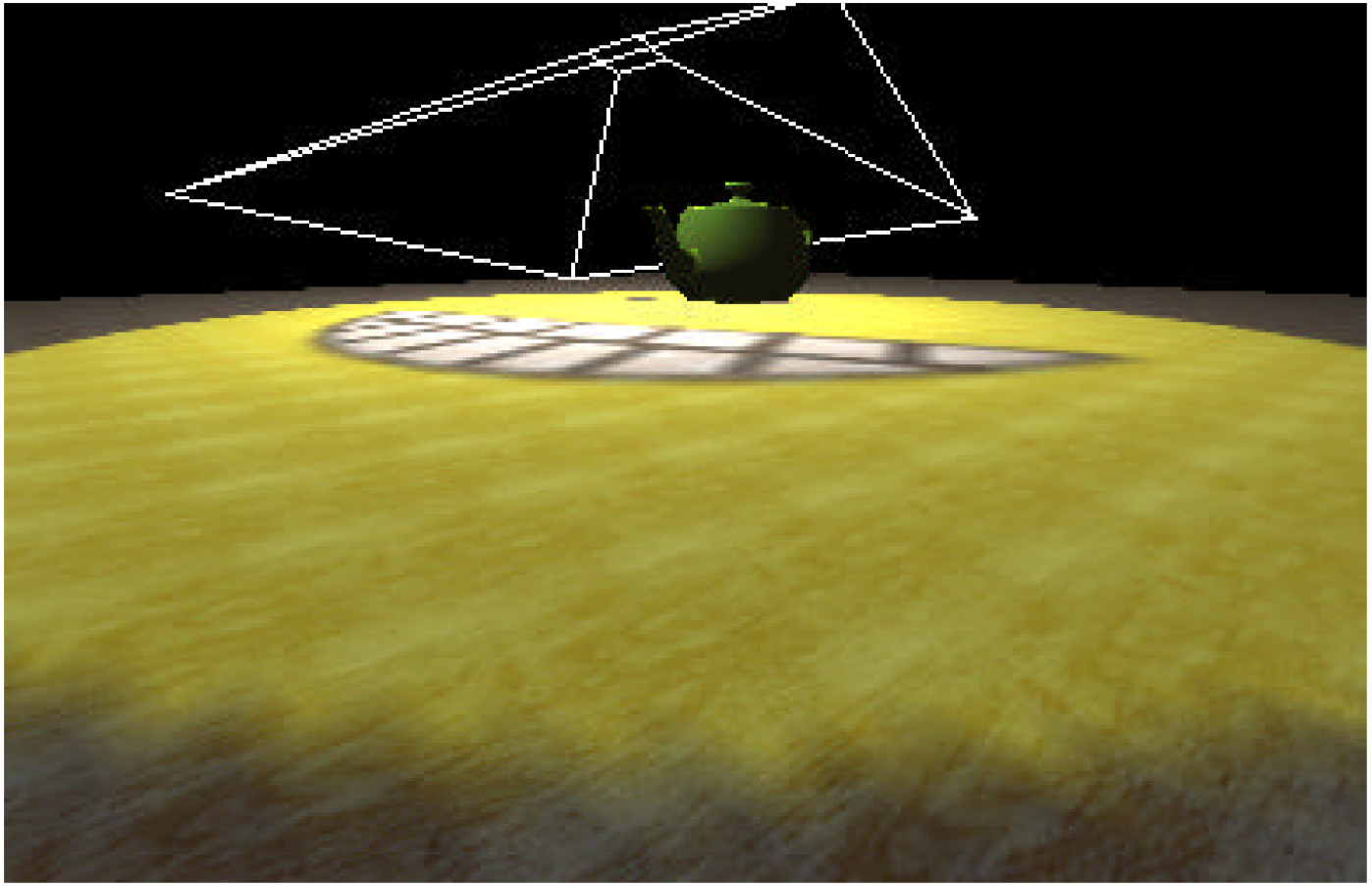
Fragment Shader (Partial)

```
uniform sampler2D projMap;
varying vec3 normal, lightDir, eyeVec;

void main (void)
{

  // Lighting calculations …

  float lambertTerm = dot(N,L);

  if(lambertTerm > 0.0)
  {
          // More lighting …

          // Suppress the reverse projection.
          if( gl_TexCoord[0].q>0.0 )
          {
           vec4 ProjMapColor = texture2DProj(projMap, gl_TexCoord[0]);
           final_color += ProjMapColor*lambertTerm;
          }
  }

  gl_FragColor = final_color;
}
```

Pitfalls: Reverse Projection



Pitfalls: Aliasing



texture2DProj

```
if( gl_TexCoord[0].q>0.0 )
{
  // Perform divide by 'q' for correct interpolation
  vec2 projCoords = gl_TexCoord[0].st / gl_TexCoord[0].q;

  // Look up using projected coordinates
  vec4 ProjMapColor = texture2D(projMap, projCoords);

  final_color += ProjMapColor*lambertTerm;
}
```

Applications

- Typical application of projective texturing:
  - Lighting the scene with unusual light shape, i.e. non-circular light
  - Fast shadowing
  - Projecting texture on object(s)

Reference

- http://www.ozone3d.net/tutorials/glsl_texturing_p08.php
- http://www.opengl.org/resources/code/samples/sig99/advanced99/notes/node79.html
- http://www.opengl.org/wiki/Mathematics_of_glTexGen
- OpenGL Programming Guide (Red Book) 8th edition, Chapter 6 “Textures”