CS 537 – Interactive Computer Graphics

Lecture 7 – Part 1

Mapping Basics

Texture Mapping in OpenGL

Topics

- Mapping Basics
- Texture Mapping in OpenGL

Reading

- Angel
  - Chapter 7
- Red Book
  - Chapter 6
  - Chapter 8

Limitations of Geometric Modeling

- Although graphics cards can render over 10 million polygons per second, that is still insufficient for a lot of things:
  - Clouds
  - Grass
  - Terrain
  - Skin

Modeling and Orange

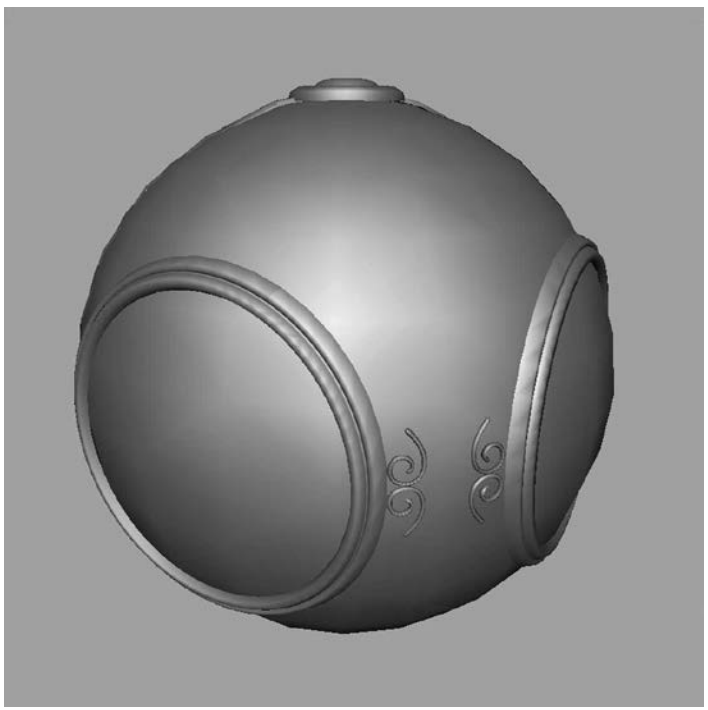
- Considering the problem of modeling an orange (the fruit)
- Start with an orange-colored sphere
  - Too simple
- Replace sphere with a more complex shape
  - Doesn't capture surface characteristics (small dimples)
  - Too many polygons to model all the dimples
- Take a picture of a real orange, scan it, and "paste" onto simple geometric model
  - This is the concept of texture mapping
- May still not be sufficient because the underlying model is smooth
  - Need to change the local shape
  - Bump mapping

Types of Mapping

- Texture mapping
  - Using images to fill inside of polygons
- Environmental mapping (reflection mapping)
  - Use a picture of the environment for the texture map
- Bump mapping
  - Emulates altering normal vectors during the rendering process to give the illusion of bumps

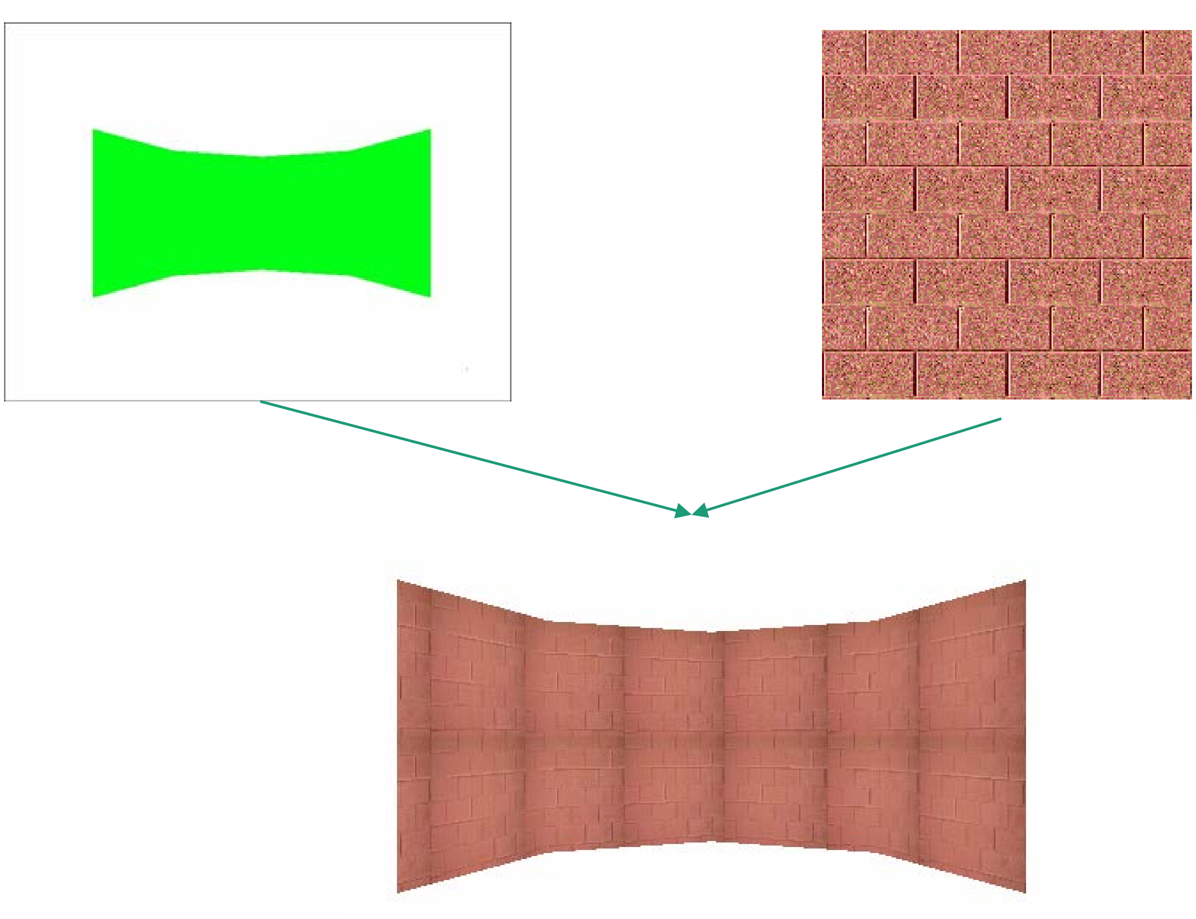
Texture Mapping

Geometric Model

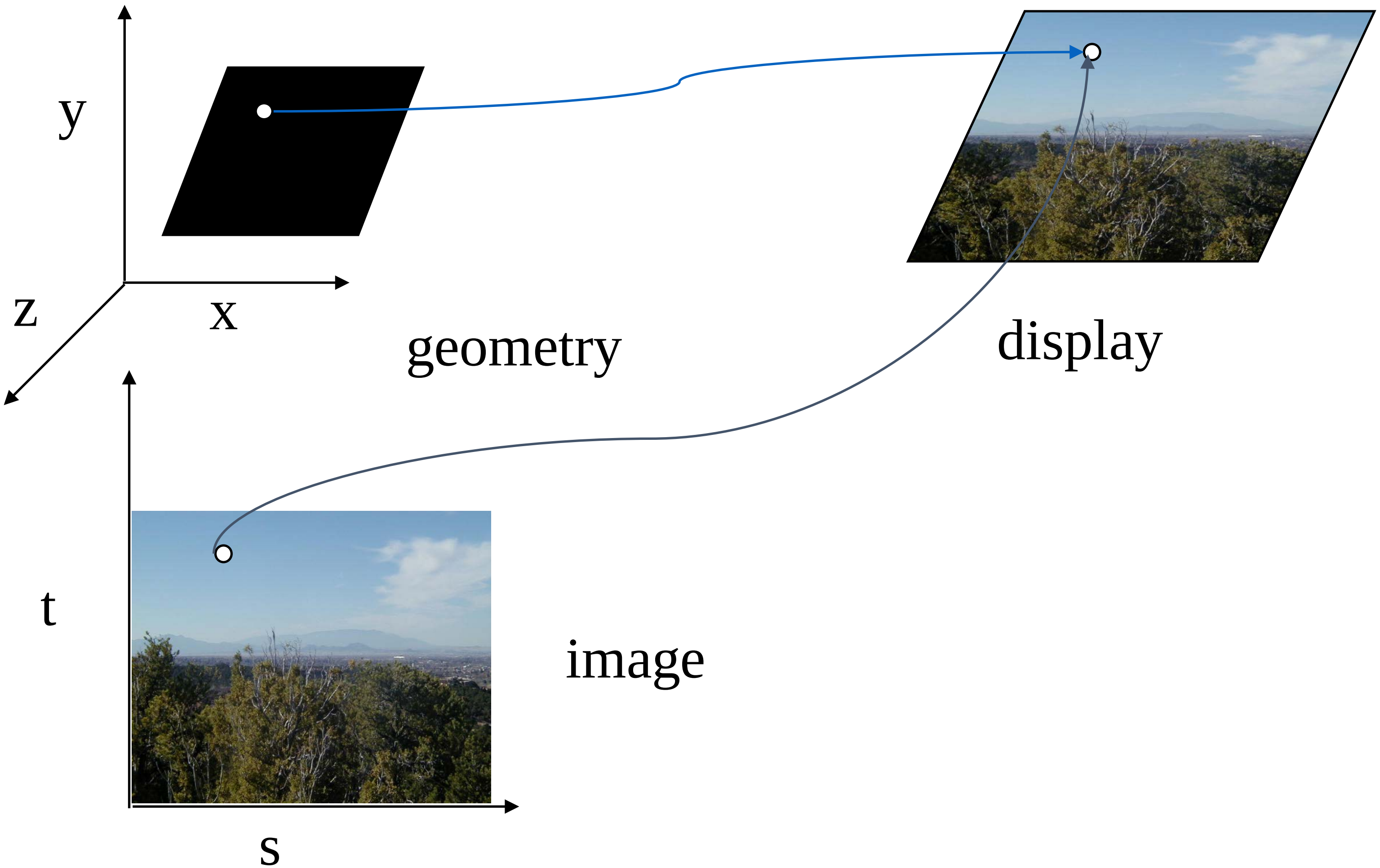


Texture mapped

Texture Mapping

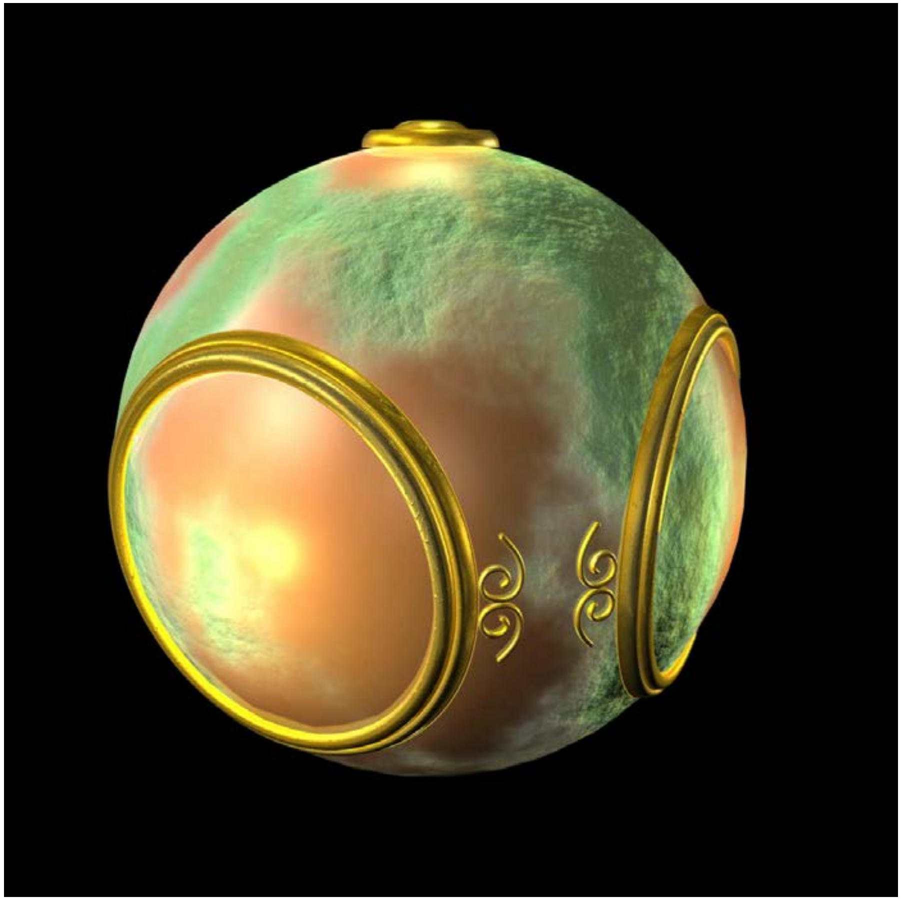


Texture Mapping



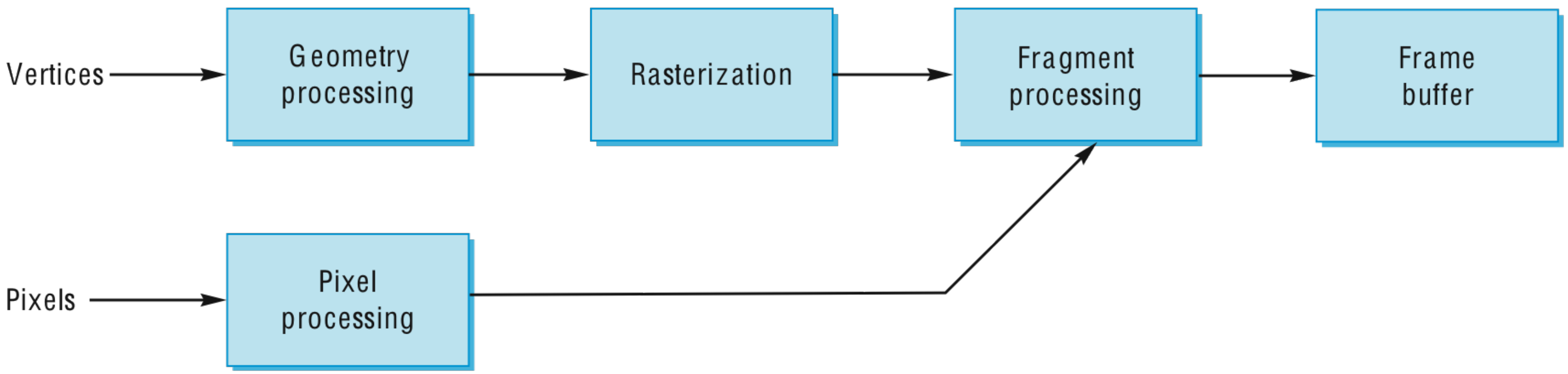
Environment Mapping

Bump Mapping



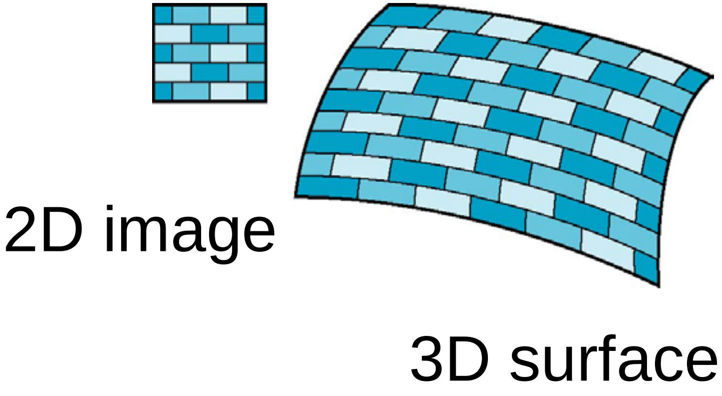
Where does mapping take place?

- Mapping techniques are implemented at the end of the rendering pipeline
  - Very efficient because few polygons make it past the clipper



Mapping

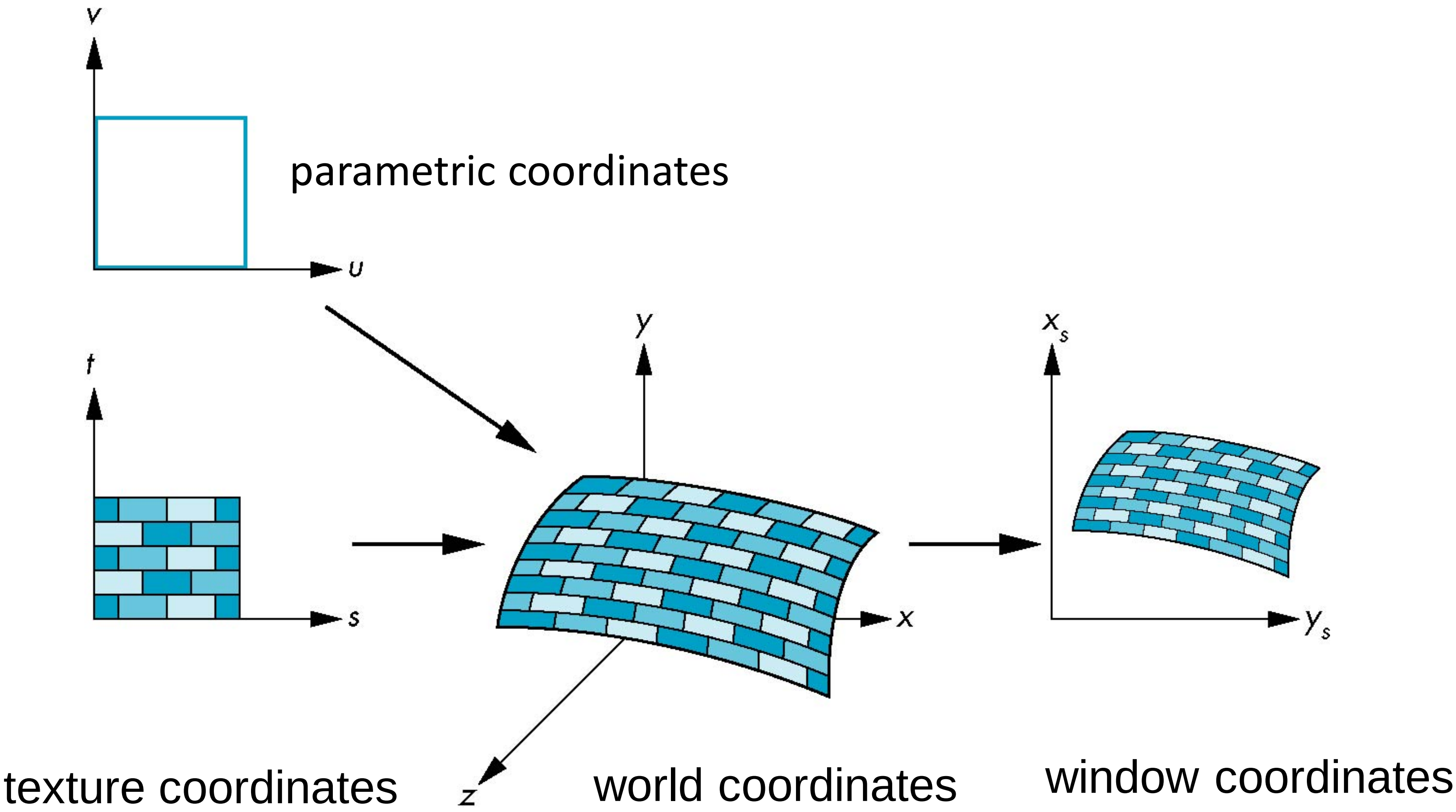
- To do mapping we're essentially assigning locations in one coordinate to another
- In particular, since images are usually 2D we're mapping a 2D surface to a 3D surface
- The whole processes involves quite a few coordinate systems



Coordinate Systems

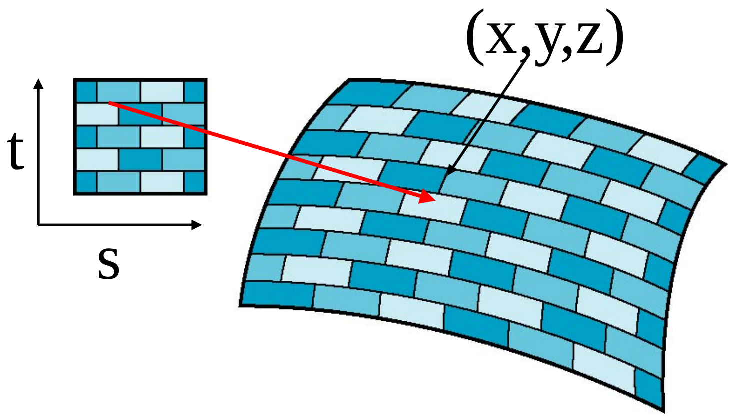
- Parametric Coordinates
  - May be used to model curves and surfaces
- Texture coordinates
  - Used to identify points in the image to be mapped
- Object or Model Coordinates
  - This is where the mapping can happen

Texture Mapping



Mapping Functions

- The most difficult problem is finding the mapping functions from one coordinate system to another
- Consider mapping from texture coordinates to a point on a surface
  - We need 3 functions
    - $x = f_x(s, t);$
    - $y = f_y(s, t)$
    - $z = f_z(s, t)$



- Or we can go the other way, from object/model coordinates to texture coordinates

Backwards Mapping

- To map from object/world coordinates back to texel:
  - Given a point on an object, we want to know to which point in the texture it corresponds
- Need mapping of the form
  - $s = fs(x, y, z)$
  - $t = f_t(x, y, z)$
- But these are generally difficult to come up with ☹

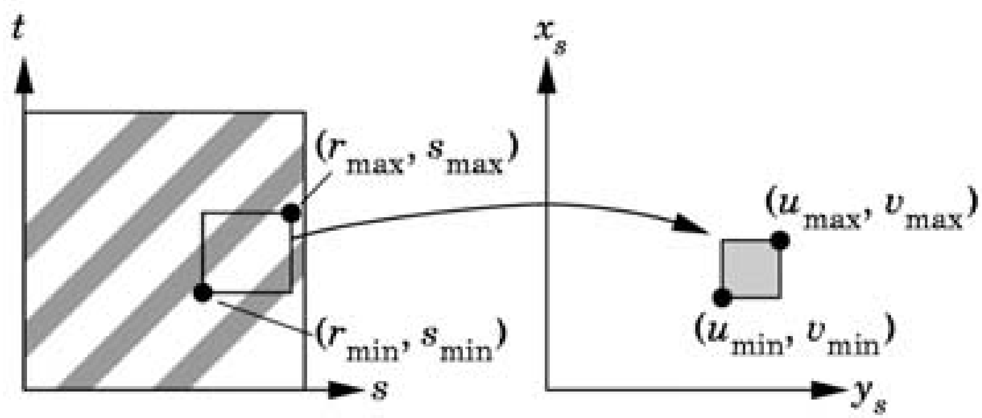
Common Mappings

- Most mappings can fit into one of three categories
  1. Map to plane
  2. Map to parametric surface
  3. Map to polyhedron (closed surface)

- Each have their own common approaches

Mapping a Plane

- For each vertex we just need to specify the location on the texture $0 \leq (s, t) \leq 1$
- We can also call this "direct mapping"



Mapping a Parametric Surface

- If we can specify a surface parametrically $f(u, v)$ then we can find functions that go from
  - Object space (x, y, z) to
  - Parametric space (u, v) to
  - Texture space (s, t) the last of which is a planar/linear mapping

Cylindrical Mapping

- We can map a rectangle in parametric $0 \le u, v \le 1$ space to cylinder of radius r and height h in world coordinates as :
  - $x = r\cos(2\pi\, u)$
  - $y \ = \ r\sin(2\pi\, u)$
  - $z \ = \ v/h$
- So given a point on the cylinder in 3D space (x, y, z), the radius r, and the height h, we can find its location in parametric space as:
  - $u = \frac{\cos^{-1}\left(\frac{x}{r}\right)}{2\pi}$
  - $v \ = \ z * h$
- Then to map from parameter space to texture space we can assign
  - $s = u$
  - $t = v$

Sphere Map

- Recall the parametric form of a sphere
  - $x(\theta, \Phi) \ = \ r\cos\theta\sin\Phi$
  - $y(\theta, \Phi) \ = \ r\sin\theta\sin\Phi$
  - $z(\theta, \Phi) \ = \ r\cos\Phi$
- Using some basic algebra and trig functions:
  - $\frac{y}{x} = \frac{rsin\theta sin\varphi}{rcos\theta sin\varphi} = tan\theta$
- Therefore:
  - $\theta = tan^{-1}\left(\frac{y}{x}\right)$ //NOTE: Use the `tan2(y,x)` function to handle cases when $x = 0$
  - $\varphi = cos^{-1}\left(\frac{z}{r}\right)$
- Since θ and Φ range from $-\pi$ to π we must shift them by π and then divide by 2π to get our range in [0 1]
  - $s = (\theta + \pi)/(2\pi)$
  - $t = (\varphi + \pi)/(2\pi)$

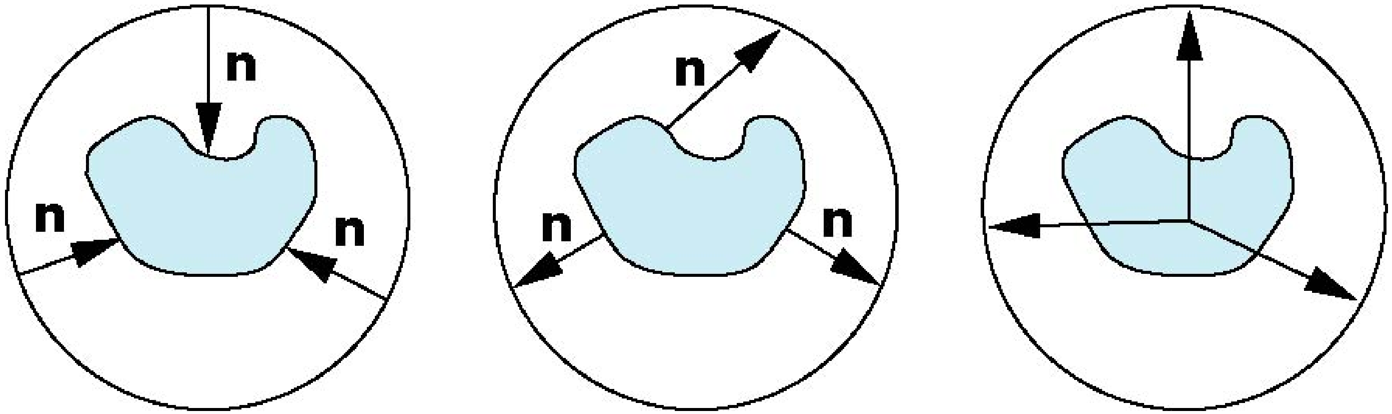
Mapping an Enclosed Polyhedron

- An approach to mapping an enclosed polyhedron is a two-step process

1. Map the texture to an intermediate surface
    - Commonly a sphere or box
2. Project points on one surface to the other
    - Generally using the surface normals
    - For the 2nd stage, there are a few approaches

Second Mapping

- Map from intermediate object to actual object
  - Normals from intermediate to actual (find where intersect actual object)
  - Normals from actual to intermediate (find where intersect intermediate object). Probably the best
  - Vectors from center of intermediate (find where interesects both objects)



Texture Mapping in OpenGL

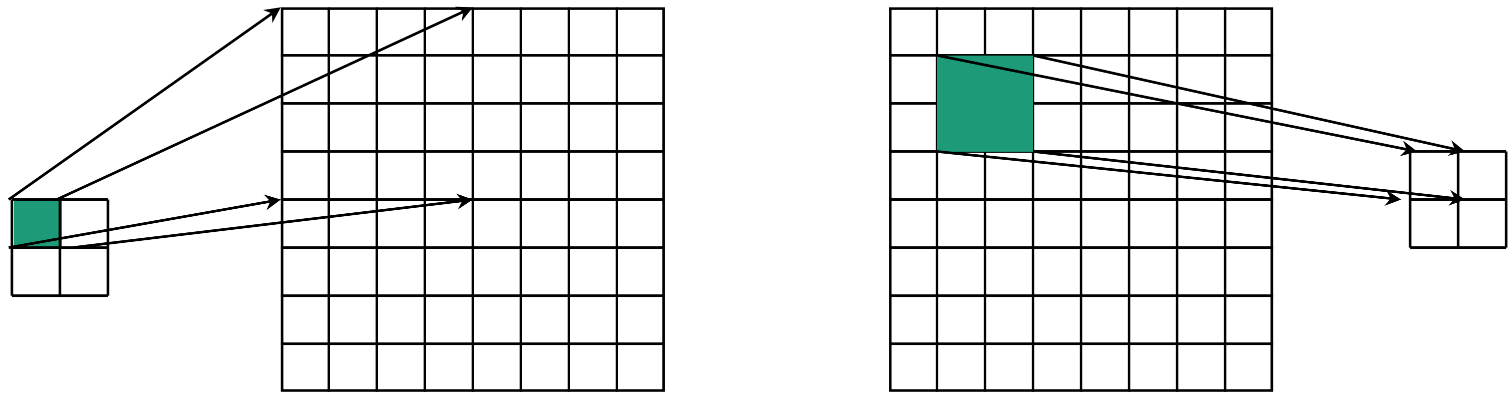
- Basic Strategy:
  1. Get the texture data
     - Read or generate image
  2. Make a texture active (bind it) and move data to that texture (put on GPU)
  3. Specify texture parameters
     - Wrapping, filtering
  4. Assign texture coordinates to vertices
     - Proper mapping function is left to application
  5. Draw the scene, making texture units active and associating textures with texture units as necessary.

Design Ideas

- You may want to include with the object (static or not?)
  - Texture coordinates
  - Texture

Create Texture Object

- Texel values can be up to 4D (RGBA)
- Loading textures is **expensive**
- Faster to bind (`glBindTexture`) then to load (`glTexImage*D`)
  - So load once, and reuse/bin when needed
- Textures are similar to VBOs and VAOs
  - Create texture names:
    - ```
      GLuint texName[4];
      glGenTextures(4, texName);
      ```
  - Enable/activate a texture object
    - `glBindTexture(GL_TEXTURE_2D, texName[0]);`
      ```

Multi-Texturing

- We can use *multi-texturing* to allow for several textures per object.
- This is done by assigning textures to *texture units*
- Think of texture locations a matrix:

GL_TEXTURE0	**GL_TEXTURE1**	**…**	**GL_TEXTUREN**
GL_TEXTURE_2D	GL_TEXTURE_2D	…	GL_TEXTURE_2D
GL_TEXTURE_CUBE_MAP	GL_TEXTURE_CUBE_MAP	…	GL_TEXTURE_CUBE_MAP

Multi-Texturing

- When drawing, we do the following:
  - We must first select/activate a texture unit:
    - `glActiveTexture(GL_TEXTURE0);`
  - Next we must make sure that the texture type we want within that unit is active
    - `glEnable(GL_TEXTURE_2D)`
  - Finally we just associate a texture with the texture type in that texture unit
    - `glBindTexture(GL_TEXTURE_2D, textures[0]);`

Step 1: Get Texture Data

- We define a texture image from an array of *texels* (texture elements) in CPU memory
  - `GLubyte my_texels[512][512][3]`
- We may populate this texture image by
  - Manually/systematically providing values
  - Loading from image

Step 1: Create Data

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


//Create a 64x64 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;
      }
}
```

Step 1: Load Data

- For simplicity in this class to load texture from images let's
  1. Convert an image to a portable pix map (PPM) using an online utility:
     - http://ziin.pl/en/utilities/convert/
  2. Load the PPM into an unsigned byte array. Code provided for you in the Drawable class in the sample code:

```
static unsigned char* ppmRead(char* filename,
        int* width, int* height);
```

Step 2: Bind Data to Texture Unit

- We must first request from GPU a list of available textures
  - `getGenTextures(3,textures)`
- Then we must make active the texture we want
  - `glBindTexture(GL_TEXTURE_2D, textures[2]);`
- Next we must move the data into the texture
  - `glTexImage2D(target, level, components, w, h, border, format, type, texels)`
    - `target`: Type of texture, e.g. `GL_TEXTURE_2D`
    - `level`: Used for mipmapping (discussed later)
    - `components`: # of elements per texel (RGB, etc..)
    - `w`, `h`: Width and Height of texture in pixels
    - `border`: Use for smoothing (typically 0 or 1)
    - `format` and `type`: Describe texels
    - `texels`: Pointer to texel array
  - EX:

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

Step 3: 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 mode allows 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
  - `glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP;`
- Repeating: Use $s, t \; modulo \; 1$
  - `glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);`

Magnification and Minification

- One texel can cover more than one pixel (magnification) or one pixel can be covered by more than one texel (minification).
- To specify how to deal with this, for each situation (OpenGL detects if either situation occurs) we can specify:
  - Use point sampling (nearest texel)
  - Use linear filtering (2x2 filter) to obtain texture values



Texture Polygon Texture Polygon

Magnification Minification

Filter Modes

- Modes determined by
  - `glTexParameteri(target, type, mode)`
- Examples:
  - `glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);`
  - `glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);`
- Note: 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
  - Probably want higher resolution images for closer objects, lower resolution for further objects
- Declare mipmap level during texture definition
  - `glTexImage2D(GL_TEXTURE_*D, level, …)`
- Alternatively allow OpenGL to make the mipmaps
  - `glGenerateMipmaps(GL_TEXTURE_2D);`
  - ```
    glTexParameterf(GL_TEXTURE_2D,
                    GL_TEXTURE_MAG_FILTER,
                    GL_NEAREST_MIPMAP_NEAREST);
    ```
    ```

Step 4: Assign Texture Coordinates

- We talked about different ways in Part 1
- Let's look at the simple plane->plane mapping:

```
void Cube::quad2Triangles(vec4 v1, vec4 v2, vec4 v3, vec4 v4){
    //triangle 1
    vec3 N = normalize(cross(v2-v1,v3-v1));
    vertices[index]=v1;  normals[index] = N;    textures[index] = vec2(0.0,0.0);    index++;
    vertices[index]=v2;  normals[index] = N;    textures[index] = vec2(1.0,0.0);    index++;
    vertices[index]=v3;  normals[index] = N;    textures[index] = vec2(1.0,1.0);    index++;

    N = normalize(cross(v3-v1,v4-v1));
    vertices[index]=v1;  normals[index] = N;    textures[index] = vec2(0.0,0.0);    index++;
    vertices[index]=v3;  normals[index] = N;    textures[index] = vec2(1.0,1.0);    index++;
    vertices[index]=v4;  normals[index] = N;    textures[index] = vec2(0.0,1.0);    index++;
}
```

Step 5: Rendering with Textures

- Just need to
  - Make sure texture location attributes are linked and enabled
    - `vTexture = glGetAttribLocation(program, “vTexCoord”);`
    - `glEnableVertexAttribArray(vTexture);`
    - `glVertexAttribPointer(vTexture,2,GL_FLOAT, GL_FALSE,0, BUFFER_OFFSET(sizeof(vertices)+ sizeof(normals)));`
  - Make sure desired texture unit is active (for drawing)
    - `glActiveTexture(GL_TEXTURE0);`
    - `glEnable(GL_TEXTURE_2D)`
    - `glBindTexture(GL_TEXTURE_2D,texture);`
    - `glUniform1i(glGetUniformLocation(program,”texture”),0);`

Shaders

- Vertex Shader
  - Often just pass the texture attribute through to fragment shader
- Fragment Shader
  - Use a texture “sampler” to get the color from the current texture at the specified texture location

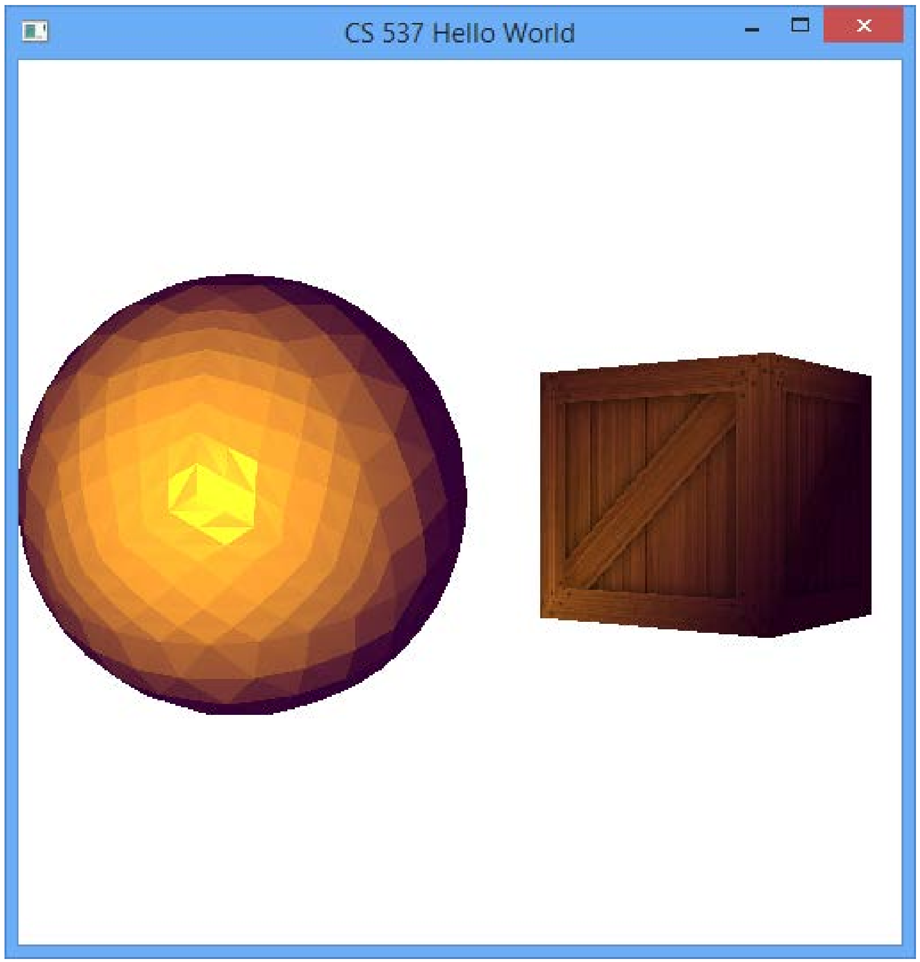
Vertex Shader

```
in vec4 vPosition;  //vertex position in object cords
in vec4 vColor; //vertex color from application
in vec2 vTexCoord; //texture coord from app

out vec4 color;  //output color to be interpolated
out vec2 texCoord; //output tex coordinate to be interpolated
```

Fragment Shader

```
in vec4 color;  //color from rasterizer

in vec2 texCoord; //texture coordinate from rasterizer

uniform sampler2D textureID; //texture object from application;

out vec4 fColor;


void main(){
        fColor = color*texture(textureID,texCoord);
}
```

Example: Crate = Lights + Texture



Crate Example: Building Object

```
CubeTextured::CubeTextured() {
    glGenBuffers(1, &VBO);
    glGenVertexArrays(1, &VAO);
    program = InitShader("../vshader09_texture_v150.glsl", "../fshader09_texture_v150.glsl");

    index = 0;
    TextureSize = 512;

    build();

    glBindVertexArray(VAO);
    glBindBuffer(GL_ARRAY_BUFFER, VBO);
    glBufferData(GL_ARRAY_BUFFER, sizeof(vertexLocations) + sizeof(vertexNormals) + sizeof(vertexTextureCoords),
        NULL, GL_STATIC_DRAW);
    glBufferSubData(GL_ARRAY_BUFFER, 0, sizeof(vertexLocations), vertexLocations);
    glBufferSubData(GL_ARRAY_BUFFER, sizeof(vertexLocations), sizeof(vertexNormals), vertexNormals);
    glBufferSubData(GL_ARRAY_BUFFER, sizeof(vertexLocations) + sizeof(vertexNormals), sizeof(vertexTextureCoords),
        vertexTextureCoords);

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

    GLuint vNormal = glGetAttribLocation(program, "vNormal");
    glEnableVertexAttribArray(vNormal);
    glVertexAttribPointer(vNormal, 3, GL_FLOAT, GL_FALSE, 0, BUFFER_OFFSET(sizeof(vertexLocations)));

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

Crate Example : Building (cont)

- Continued…

```
    glGenTextures(1, &texture);
    GLubyte *image0 = ppmRead("../crate_texture.ppm", &TextureSize, &TextureSize);
    glBindTexture(GL_TEXTURE_2D, texture);
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, TextureSize, TextureSize, 0, GL_RGB, GL_UNSIGNED_BYTE, image0);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);

}
```

Crate Example: Coords

```
void CubeTextured::build(){
        makeQuad(1,0,3,2);  //front
        makeQuad(2,3,7,6);  //right
        makeQuad(3,0,4,7);  //bottom
        makeQuad(6,5,1,2);  //top
        makeQuad(4,5,6,7);  //back
        makeQuad(5,4,0,1);  //left
}
```

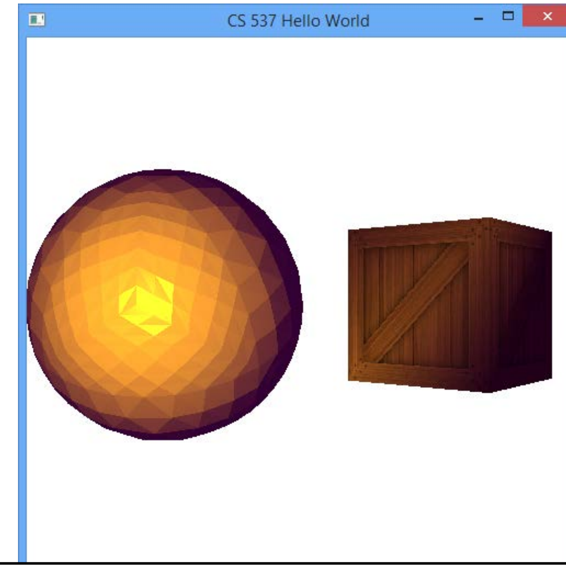
```
void CubeTextured::makeQuad(GLuint ind1, GLuint ind2, GLuint ind3, GLuint ind4){  //assume vertices in counter-clockwise order from top-left
    vec4 data[] = {vec4(-0.5,-0.5,0.5,1.0),vec4(-0.5,0.5,0.5,1.0),vec4(0.5,0.5,0.5,1.0),vec4(0.5,-0.5,0.5,1.0),
                    vec4(-0.5,-0.5,-0.5,1.0),vec4(-0.5,0.5,-0.5,1.0),vec4(0.5,0.5,-0.5,1.0),vec4(0.5,-0.5,-0.5,1.0)};

    //first triangle
    vec3 N = normalize(cross(data[ind2]-data[ind1],data[ind3]-data[ind1]));
    vertexLocations[index] = data[ind1]; vertexNormals[index] = N; vertexTextureCoords[index] = vec2(0,0); index++;
    vertexLocations[index] = data[ind2]; vertexNormals[index] = N; vertexTextureCoords[index] = vec2(1,0); index++;
    vertexLocations[index] = data[ind3]; vertexNormals[index] = N; vertexTextureCoords[index] = vec2(1,1); index++;

    //second triangle
    N = normalize(cross(data[ind3] - data[ind1],data[ind4]-data[ind1]));
    vertexLocations[index] = data[ind3]; vertexNormals[index] = N; vertexTextureCoords[index] = vec2(0,0); index++;
    vertexLocations[index] = data[ind4]; vertexNormals[index] = N; vertexTextureCoords[index] = vec2(1,1); index++;
    vertexLocations[index] = data[ind1]; vertexNormals[index] = N; vertexTextureCoords[index] = vec2(0,1); index++;

}
```

Crate Example: Draw Cube

```
void CubeTextured::draw(Camera cam, vector<Light> lights){
    glBindVertexArray(VAO);
    glUseProgram(program);

    GLuint light_loc = glGetUniformLocation(program, "lightPos");
    glUniform4fv(light_loc, 1, cam.getEye());

    GLuint diffuse_loc = glGetUniformLocation(program,"diffuseProduct");
    glUniform4fv(diffuse_loc,1,diffuse*lights[0].getDiffuse());

    GLuint spec_loc = glGetUniformLocation(program,"specularProduct");
    glUniform4fv(spec_loc,1,specular*lights[0].getSpecular());

    GLuint ambient_loc = glGetUniformLocation(program,"ambientProduct");
    glUniform4fv(ambient_loc,1,ambient*lights[0].getAmbient());

    GLuint alpha_loc = glGetUniformLocation(program,"alpha");
    glUniform1f(alpha_loc,shininess);

    GLuint model_loc = glGetUniformLocation(program,"model_matrix");
    glUniformMatrix4fv(model_loc,1,GL_TRUE,modelmatrix);

    GLuint view_loc = glGetUniformLocation(program, "camera_matrix");
    glUniformMatrix4fv(view_loc, 1, GL_TRUE, cam.getViewMatrix());

    GLuint proj_loc = glGetUniformLocation(program, "proj_matrix");
    glUniformMatrix4fv(proj_loc, 1, GL_TRUE, cam.getProjectionMatrix());

    glEnable(GL_TEXTURE_2D);
    glActiveTexture(GL_TEXTURE0);
    glBindTexture(GL_TEXTURE_2D, texture);
    glUniform1i(glGetUniformLocation(program,"textureID"),0);

    glDrawArrays(GL_TRIANGLES,0,numVertices);
}
```

Crate Example: Shaders

```
#version 150

in vec4 vPosition;
in vec3 vNormal;
in vec2 vTexCoord;

out vec4 color;
out vec2 texCoord;

uniform mat4 model_matrix;
uniform mat4 camera_matrix;
uniform mat4 proj_matrix;


uniform vec4 lightPos;
uniform vec4 ambientProduct,
      diffuseProduct, specularProduct;
uniform float alpha;


void main()
{
      texCoord = vTexCoord;
      //Then all the lighting effects
      // to compute output color…..
      //….
}
```




```
#version 150

in vec4 color;
in vec2 texCoord;

uniform sampler2D textureID;

out vec4 fColor;

void main()
{
    fColor = texture(textureID,texCoord)*color;

}
```