VGP351 – Week 6

- Agenda:
  - Quiz #3
  - Texture mapping, part 1

16-November-2011

What is texture mapping?

16-November-2011

What is texture mapping?

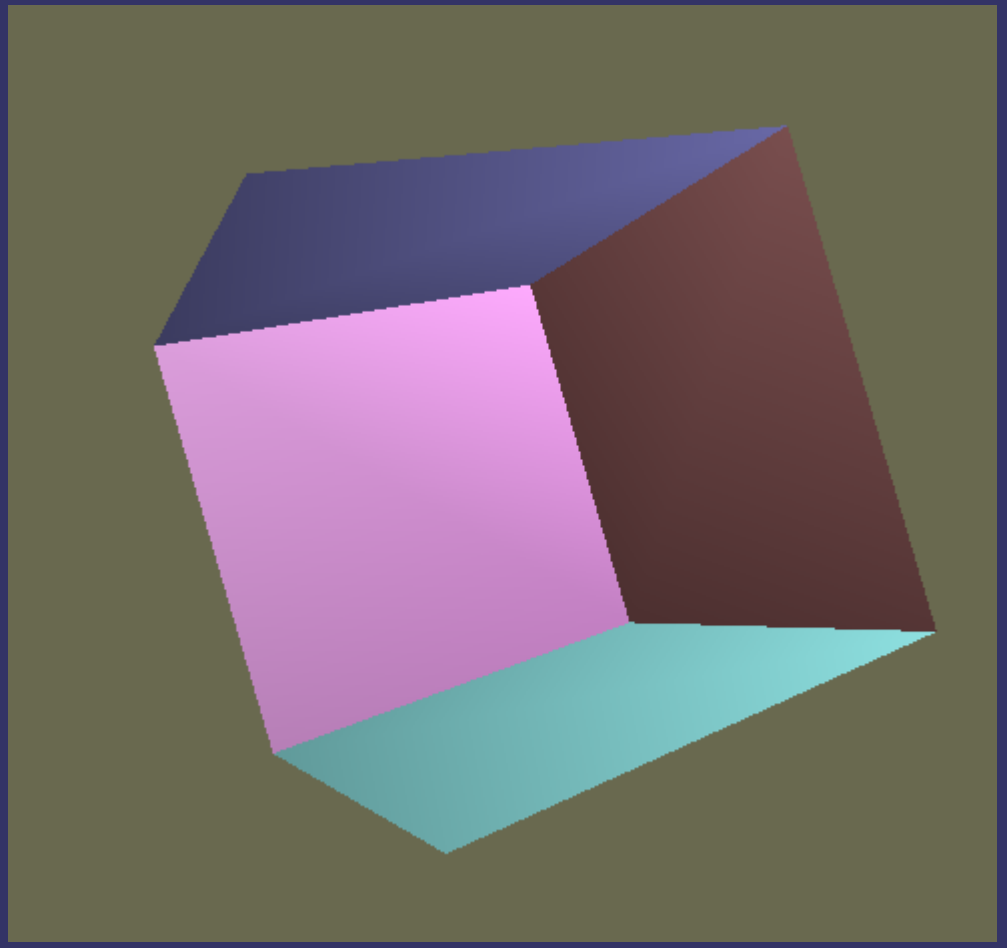
- Classic definition:

  Application of an image to a polygon or 3D model.

Kinds of Images

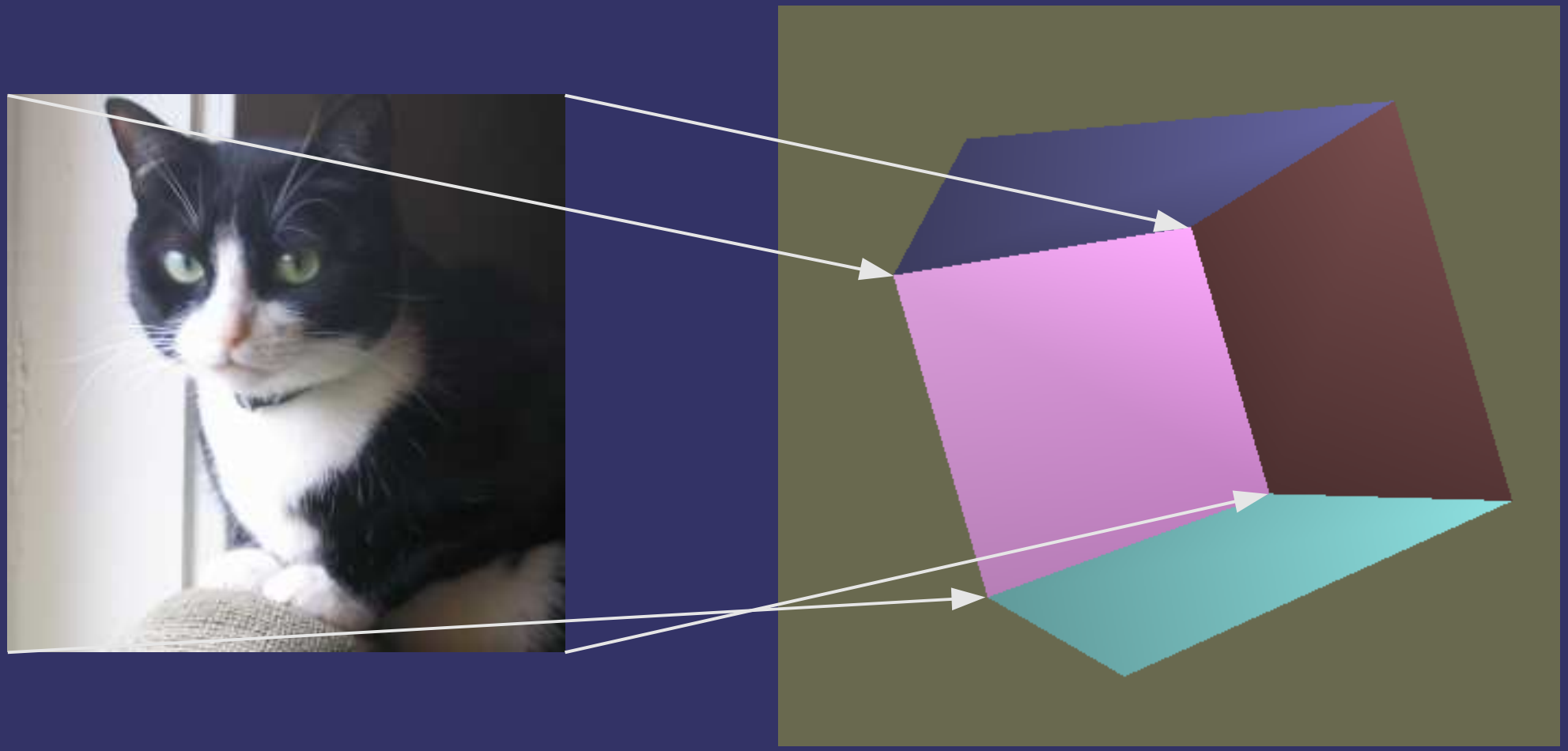
- Several *dimensionalities* are commonly used:
  - 1D – Usually used as large look-up tables or for color space conversions
  - 2D – Rectangular images...what we usually think of as a texture image
  - 3D (volumetric) – May be used to store voxel type data, volumetric light data, etc.
  - Cubemap (cubic) – 6 square, same-sized textures representing faces of a cube. Often used for environment maps

Texture Mapping



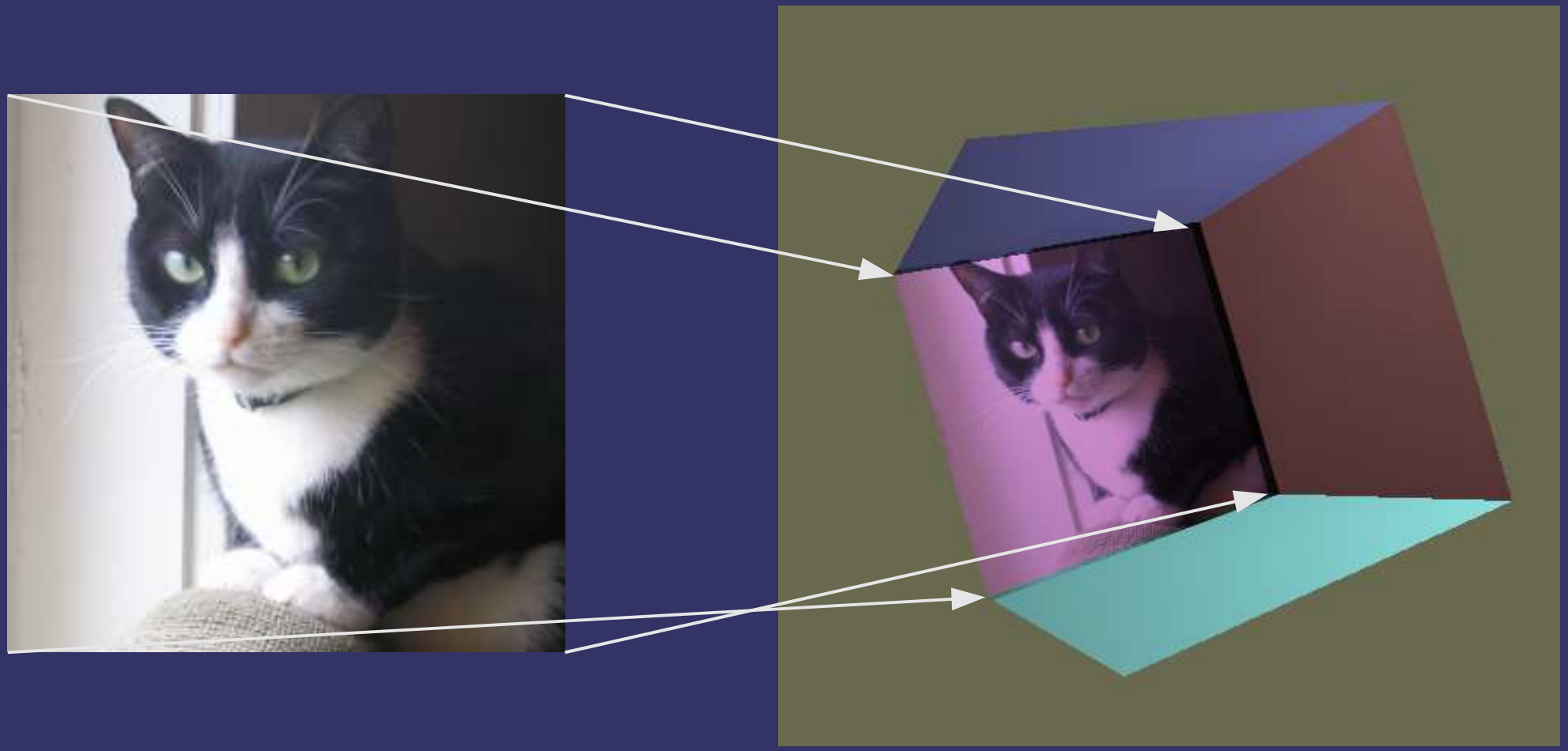
16-November-2011

Texture Mapping



16-November-2011

Texture Mapping



16-November-2011

Texture Mapping

- Where does the *mapping* come from?
  - Numerous types of projections
    - Spherical
    - Cylindrical
    - Planar
  - Reflections
  - “Hand” edited coordinates

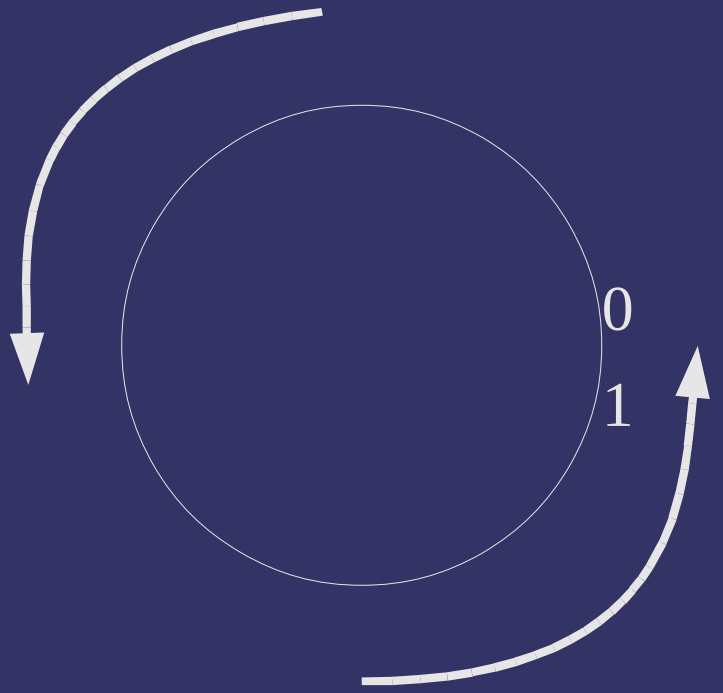
Cylindrical Mapping

16-November-2011

Cylindrical Mapping

1, 0 1, 1

0, 0 1, 0

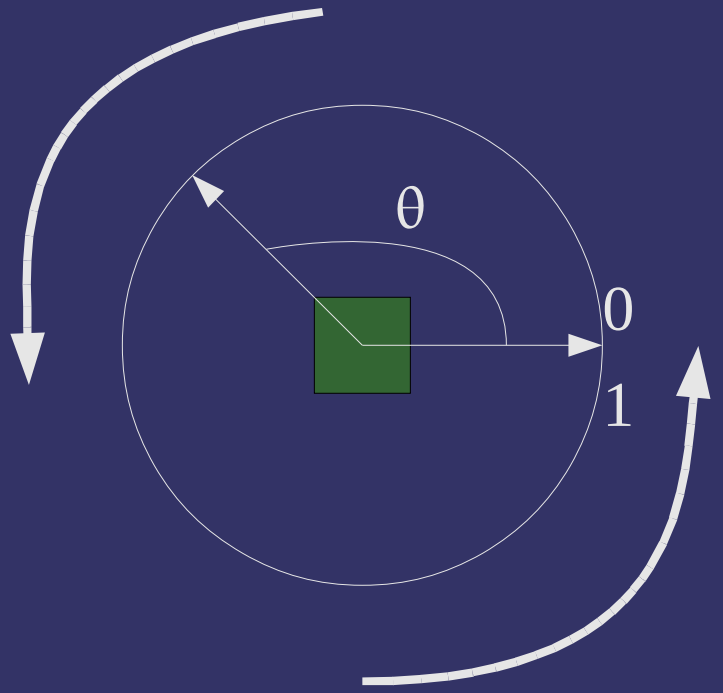


16-November-2011

Cylindrical Mapping

1, 0 1, 1

0, 0 1, 0



$u = \theta / 2\pi$

$v = y$

16-November-2011

Cylindrical Mapping

```
vec2 cylinder_map(vec3 position)
{
    vec2 tc;

    tc.s = atan(position.x, position.z) / (2.0 * PI);
    tc.t = position.y;
    return tc;
}
```

16-November-2011

Explicit Texture Coordinates

- Most commonly, texture coordinates are generated by the 3D modeling package
  - These coordinates are stored in the model file, and supplied, by you, to OpenGL

Explicit Texture Coordinates

- Most commonly, texture coordinates are generated by the 3D modeling package
  - These coordinates are stored in the model file, and supplied, by you, to OpenGL
  - Coordinates are supplied using vertex shader attributes

Point Sprites

- Special mode for rendering points that automatically generates useful texture coordinates
  - Upper left of point gets (0, 0, 0, 0) and lower right gets (1, 1, 0, 0)
  - Enable in GL with:

```
glEnable(GL_POINT_SPRITE);
```

    - Adds the fragment shader varying `gl_PointCoord`
  - `GL_NV_point_sprite` adds some other controls

Coordinate Interpolation

- Linear interpolation:

$$u_\alpha = (1-\alpha)u_0 + \alpha u_1$$

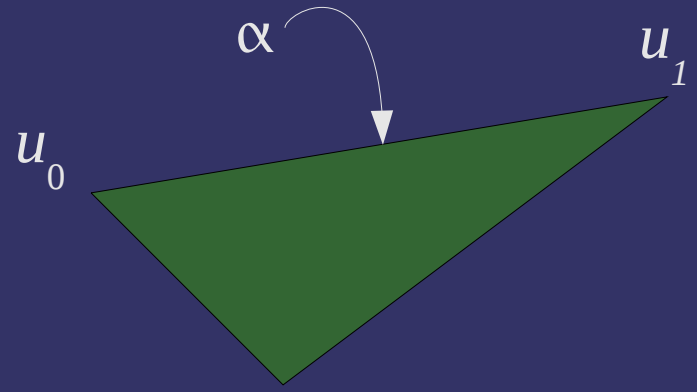


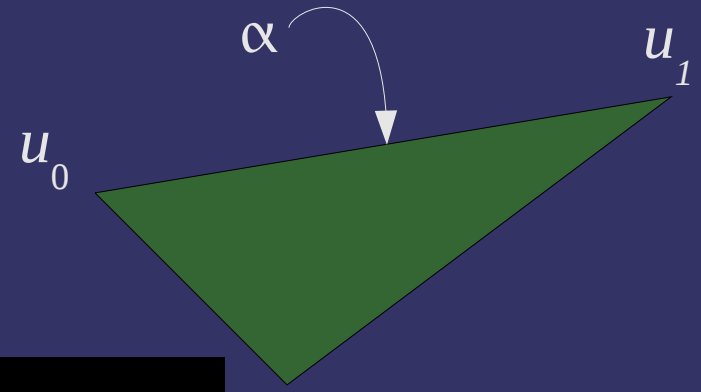
Image from http://wwwx.cs.unc.edu/~sud/courses/236/a6/

16-November-2011

Coordinate Interpolation

- Linear interpolation:

$$u_\alpha = (1-\alpha)u_0 + \alpha u_1$$



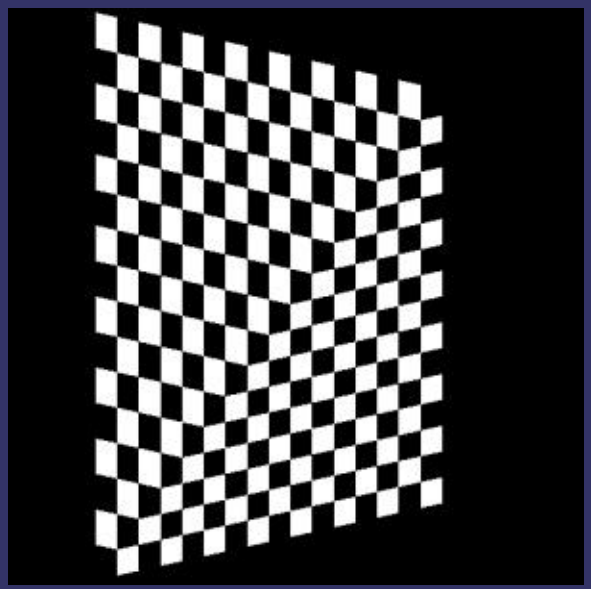


Image from http://wwwx.cs.unc.edu/~sud/courses/236/a6/

16-November-2011

Coordinate Interpolation

- Perspective correct interpolation:

$$u_\alpha = \frac{(1-\alpha)\frac{u_0}{z_0} + \alpha\frac{u_1}{z_1}}{(1-\alpha)\frac{1}{z_0} + \alpha\frac{1}{z_1}}$$

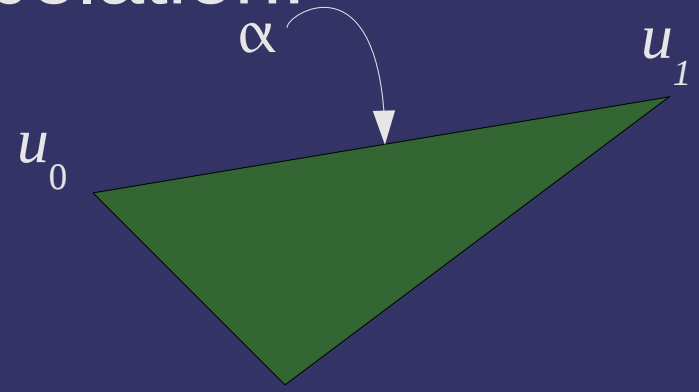


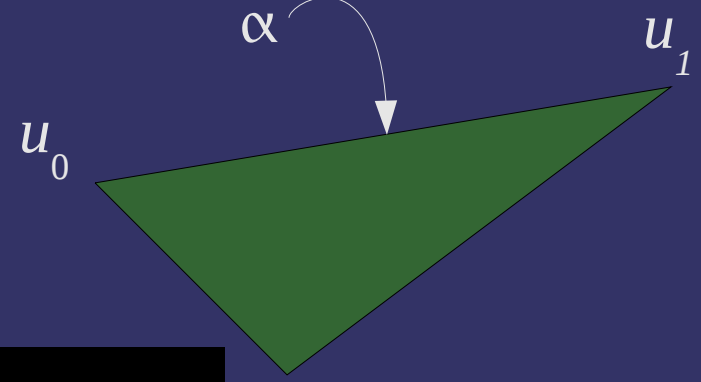
Image from http://wwwx.cs.unc.edu/~sud/courses/236/a6/

16-November-2011

Coordinate Interpolation

- Perspective correct interpolation:

$$u_\alpha = \frac{(1-\alpha)\frac{u_0}{z_0} + \alpha\frac{u_1}{z_1}}{(1-\alpha)\frac{1}{z_0} + \alpha\frac{1}{z_1}}$$



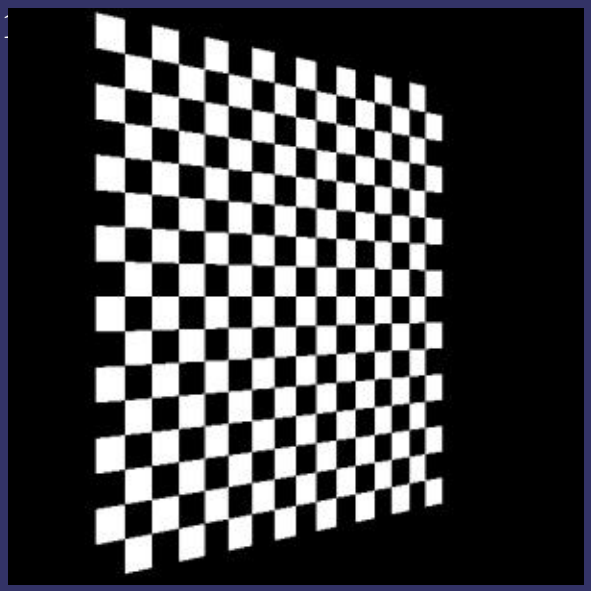


Image from http://wwwx.cs.unc.edu/~sud/courses/236/a6/

16-November-2011

Coordinate Interpolation

- Controllable via interpolation qualifiers

```
#version 130
in vec4 uses_perspective;
noperspective in vec2 uses_linear;
flat in vec3 flat_shaded;
```

  - For `flat`, the value from the last vertex in the primitive is used.

Creating Textures

- In OpenGL, textures are named objects

```
void glGenTextures(GLsizei n,
                   GLuint *textures);

void glDeleteTextures(GLsizei n,
                      const GLuint *textures);
```

- “Bind” a texture for use:

```
void glBindTexture(GLenum target,
                   GLuint texture);
```

  - `target` selects the dimensionality
  - Binding creates the object, but it still has no storage

Creating Textures

- Texture targets:
  - `GL_TEXTURE_1D` – 1D texture
  - `GL_TEXTURE_2D` – 2D texture
  - `GL_TEXTURE_3D` – 3D textures
  - `GL_TEXTURE_RECTANGLE_ARB` – Special kind of 2D texture
  - `GL_TEXTURE_CUBE_MAP` – Cubic texture
    - There are other cubic texture targets. We'll discuss those next week with environment mapping

Creating Textures

- Storage is created and *optionally* initialized with:

```
void glTexImage1D(GLenum target, GLint level,
    GLint internalFormat, GLsizei width,
    GLint border, GLenum format, GLenum type,
    const GLvoid *pixels);
```

  - Variations for 2D and 3D textures also exist
  - 1D textures do not exist in OpenGL ES
- Storage is updated with:

```
void glTexSubImage1D(GLenum target,
    GLint level, GLint xoffset, GLsizei width,
    GLenum format, GLenum type,
    const GLvoid *pixels);
```

Creating Textures

- `format` and `type` describe the source data
  - `format` can be one of: `GL_RED`, `GL_GREEN`, `GL_BLUE`, `GL_ALPHA`, `GL_RGB`, `GL_BGR`, `GL_RGBA`, `GL_BGRA`, `GL_LUMINANCE`, and `GL_LUMINANCE_ALPHA`
    - The last two are removed in OpenGL 3.1
  - With OpenGL 3.0 or `GL_ARB_texture_rg`, `format` can also be one of `GL_RG` or `GL_RED`

Creating Textures

- `format` and `type` describe the source data
  - `type` can be any of the basic type enums (e.g., `GL_UNSIGNED_BYTE`)
  - `type` can be one of the "packed" types:
    `GL_UNSIGNED_SHORT_5_6_5`,
    `GL_UNSIGNED_SHORT_4_4_4_4`,
    `GL_UNSIGNED_SHORT_5_5_5_1`,
    `GL_UNSIGNED_INT_8_8_8_8`
    - There are also `_REV` versions that reverse the ordering of the components
  - A few other uncommon types have been omitted

Creating Textures

- `internalFormat` describes how the texture should be stored
  - `GL_ALPHA`, `GL_ALPHA4`, `GL_ALPHA8`, `GL_ALPHA12`, `GL_ALPHA16`, `GL_LUMINANCE`, `GL_LUMINANCE4`, `GL_LUMINANCE8`, `GL_LUMINANCE12`, `GL_LUMINANCE16`, `GL_LUMINANCE_ALPHA`, `GL_LUMINANCE4_ALPHA4`, `GL_LUMINANCE6_ALPHA2`, `GL_LUMINANCE8_ALPHA8`, `GL_LUMINANCE12_ALPHA4`, `GL_LUMINANCE12_ALPHA12`, `GL_LUMINANCE16_ALPHA16`, `GL_INTENSITY`, `GL_INTENSITY4`, `GL_INTENSITY8`, `GL_INTENSITY12`, `GL_INTENSITY16`, `GL_RGB`, `GL_R3_G3_B2`, `GL_RGB4`, `GL_RGB5`, `GL_RGB8`, `GL_RGB10`, `GL_RGB12`, `GL_RGB16`, `GL_RGBA`, `GL_RGBA2`, `GL_RGBA4`, `GL_RGB5_A1`, `GL_RGBA8`, `GL_RGB10_A2`, `GL_RGBA12`, or `GL_RGBA16`

Creating Textures

- Storage is created and initialized from framebuffer data with:

```
void glCopyTexImage1D(GLenum target,
    GLint level, GLenum internalformat,
    GLint x, GLint y, GLsizei width,
    GLint border);
```

- Storage is updates from framebuffer data with:

```
void glCopyTexSubImage1D(GLenum target,
    GLint level, GLint xoffset,
    GLint x, GLint y, GLsizei width);
```

Texture Units

- A texture unit is the piece of hardware that accesses a texture image
- Many OpenGL texture operations are per-object, but some are per-unit
  - Select the unit with:

```
void glActiveTexture(GLenum texture);
```

Enum is `GL_TEXTURE`*n*, where *n* is unit number

Texture Units

- A texture unit is the piece of hardware that accesses a texture image
- Many OpenGL texture operations are per-object, but some are per-unit
  - Select the unit with:

```
void glActiveTexture(GLenum texture);
```

- Use this API to set per-unit texture objects as well!

Texture Parameters

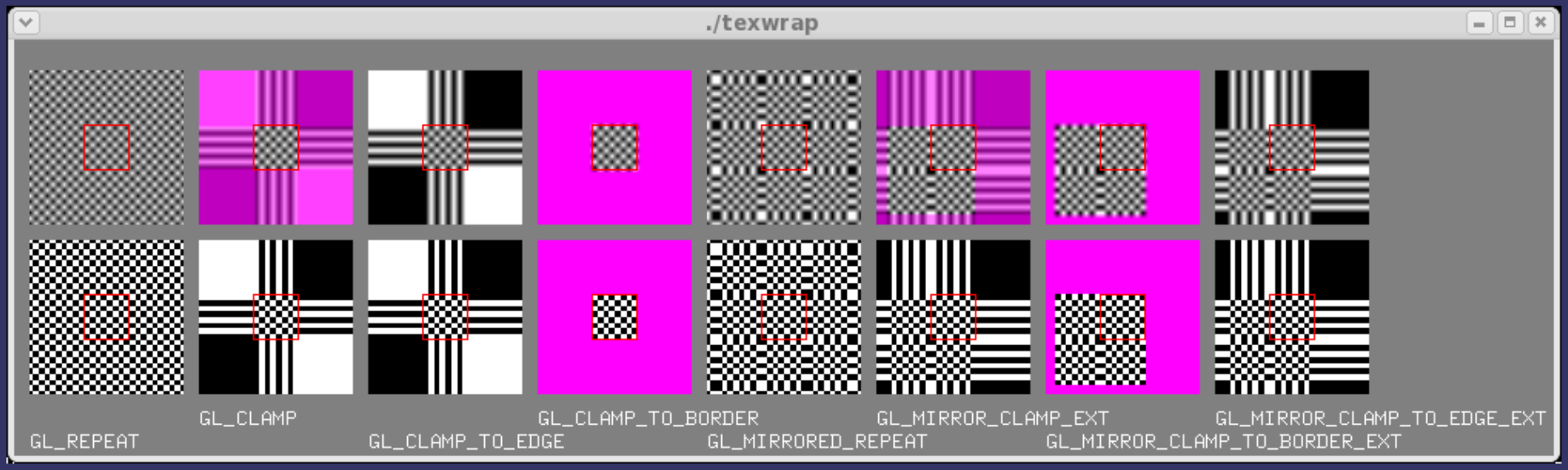
- Set texture object parameters with:

```
void glTexParameteri(GLenum target,
    GLenum pname, GLint param);

void glTexParameteriv(GLenum target,
    GLenum pname, const GLint *params);
```

Texture Wrapping

- Texture images have coordinates on the range [0, 1]
  - What happens if the requested texel coordinate is outside that range?

Texture Wrapping



16-November-2011

Texture Wrapping

- Texture images have coordinates in [0, 1]
  - What happens if the requested texel coordinate is outside that range?
  - It depends on the wrap mode!
- Wrap mode is set independently for each axis
- 8 possible modes
  - Not all implementations support all 8
  - OpenGL 1.5 and later only require 5
  - OpenGL 3.1 and later remove `GL_CLAMP`
  - OpenGL ES 2.0 only has `GL_CLAMP_TO_EDGE`, `GL_REPEAT`, and `GL_MIRRORED_REPEAT`

Texture Wrapping

- Select the wrap mode with `glTexParameter`:

```
glTexParameteri(GL_TEXTURE_2D,
                GL_TEXTURE_WRAP_S,
                GL_CLAMP_TO_BORDER);
glTexParameteri(GL_TEXTURE_2D,
                GL_TEXTURE_WRAP_T,
                GL_REPEAT);
```

Texture Wrapping

- GL_CLAMP, GL_CLAMP_TO_BORDER, and the mirrored versions use a texture “border” color

```
const GLfloat color[4] = {
    0.0, 1.0, 0.0, 1.0
};

glTexParameterfv(GL_TEXTURE_2D,
                 GL_TEXTURE_BORDER_COLOR,
                 color);
```

Samplers

- In GLSL, textures are accessed through special data types called *samplers*
  - There is a sample type for each texture target: `sampler1D`, `sampler2D`, `samplerRect`, `sampler3D`, and `samplerCube`
  - Samplers are uniforms
    - Set the sampler uniform to the number of the texture *unit*

```
sampler_loc = glGetUniformLocation(prog, “tex”);
...
glUniform1i(sampler_loc, 1);
```

Texture Sample Functions

- Textures are accessed using special GLSL functions
  - There is a many variations of these functions
  - The function name must match the sampler type
  - See the GLSL quick reference

http://www.khronos.org/files/opengl-quick-reference-card.pdf

Connecting Textures to Samplers

Texture A

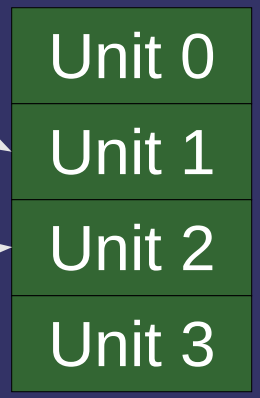
Texture B

Texture C

Texture D

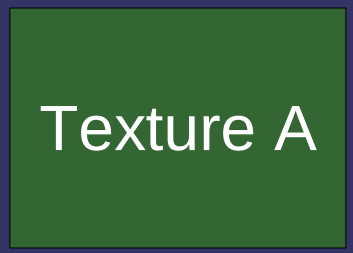
Connect texture with texture unit:

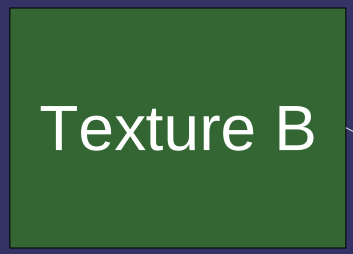
```
glGenTextures(1, &tex_b); glGenTextures(1, &tex_c);
…
glActiveTexture(GL_TEXTURE1);
glBindTexture(GL_TEXTURE_2D, tex_b);
glActiveTexture(GL_TEXTURE2);
glBindTexture(GL_TEXTURE_2D, tex_c);
```

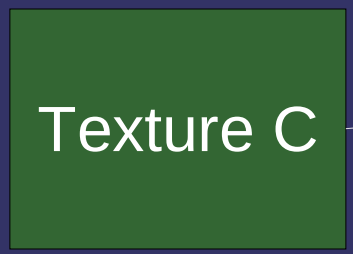


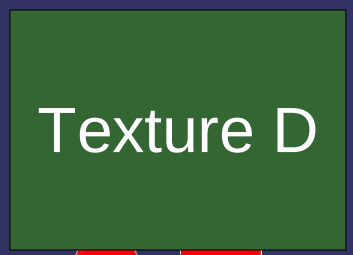

```
#version 130
uniform sampler2D tex;
in vec2 coord;
out vec4 color;

void main()
{
  color = texture(tex, cord);
}
```

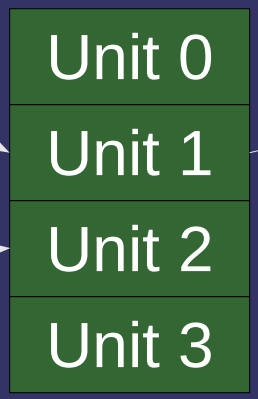
Connecting Textures to Samplers









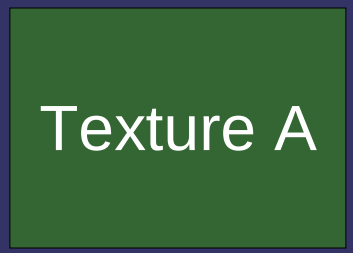


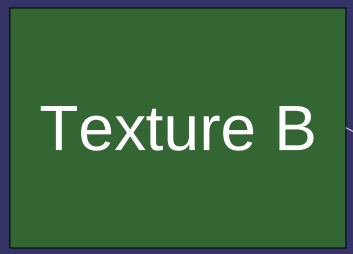

```
#version 130
uniform sampler2D tex;
in vec2 coord;
out vec4 color;

void main()
{
  color = texture(tex, cord);
}
```

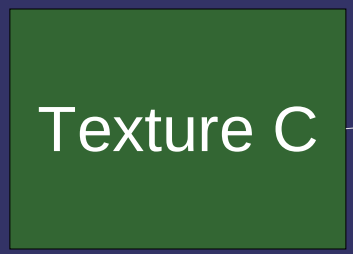
Connect texture unit with sampler:

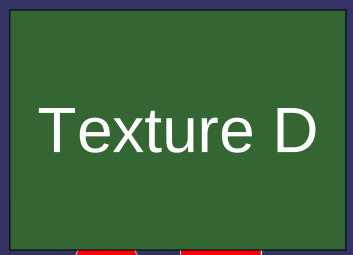
```
tex_loc = glGetUniformLocation(prog, “tex”);
…
glUniform1i(tex_loc, 1);
```

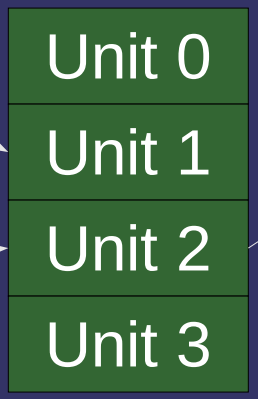
Connecting Textures to Samplers









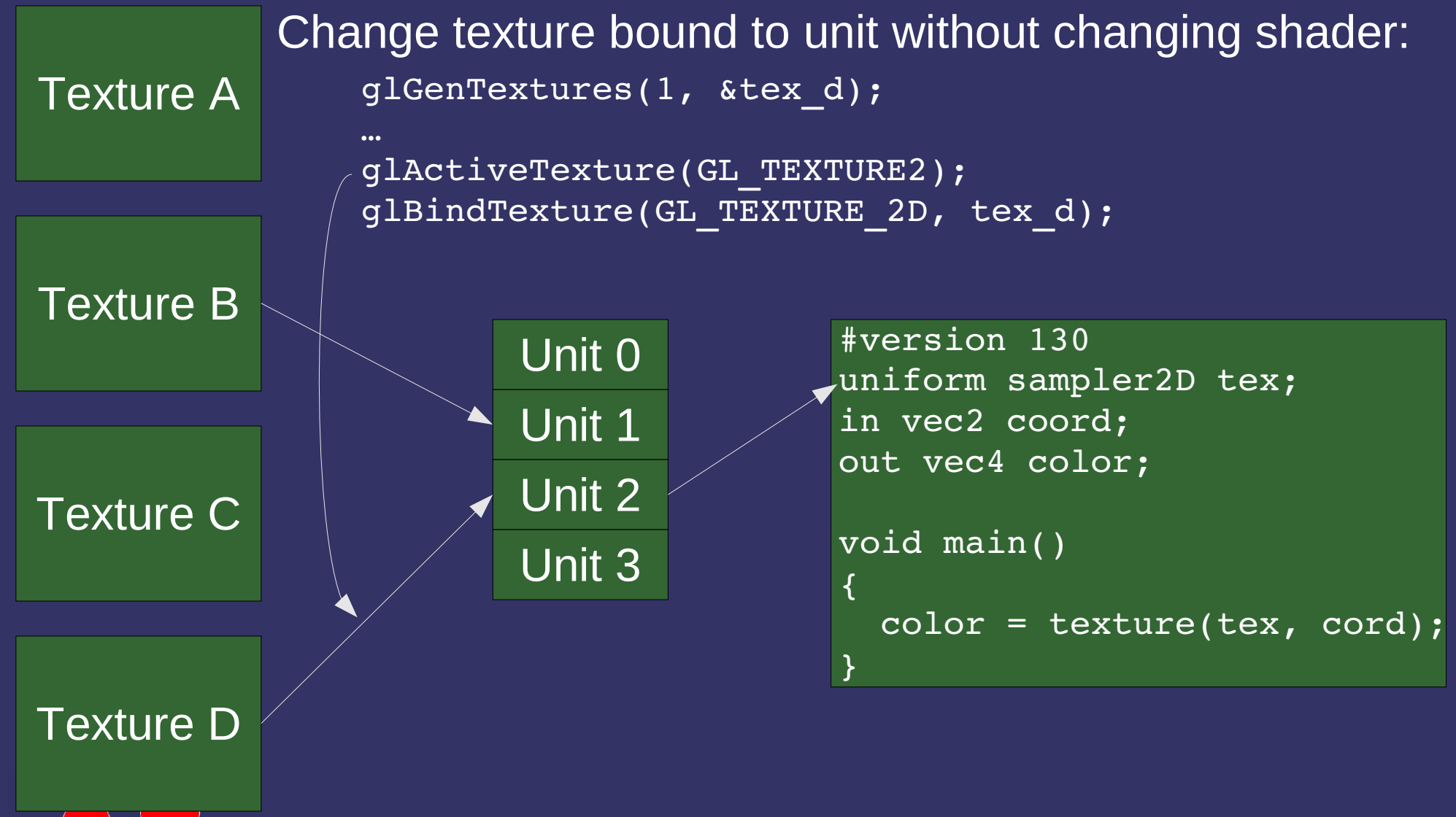



```
#version 130
uniform sampler2D tex;
in vec2 coord;
out vec4 color;

void main()
{
  color = texture(tex, cord);
}
```

Connect different texture unit with sampler:

```
tex_loc = glGetUniformLocation(prog, “tex”);
…
glUniform1i(tex_loc, 2);
```

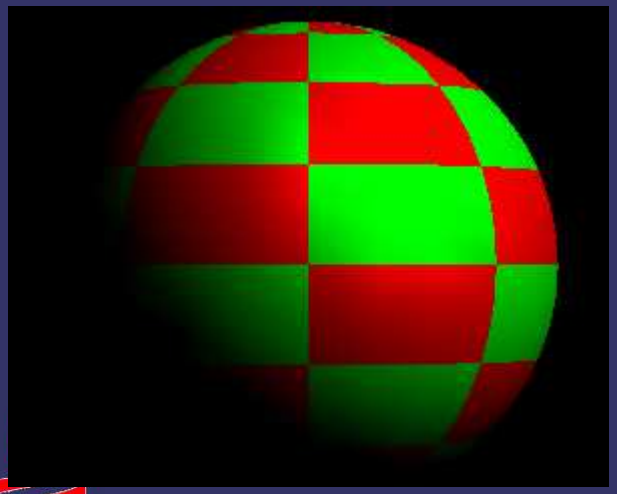
Connecting Textures to Samplers



16-November-2011

Specular Lighting

- We perform lighting in the vertex shader and texturing in the fragment shader
  - VS passes a single color to FS, and FS combines it with the texture color
  - Why is this wrong?



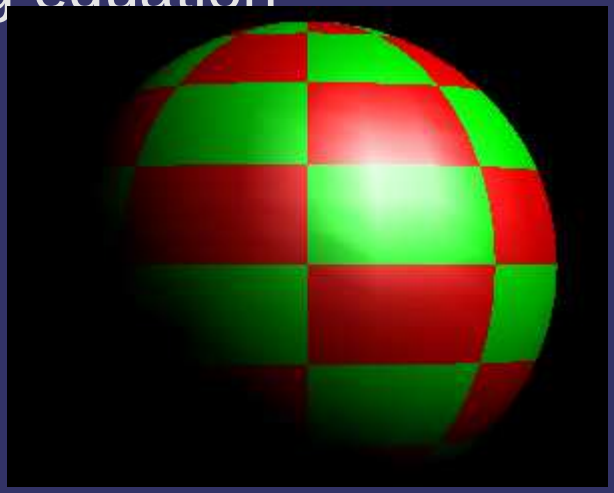
Specular Lighting

```
uniform sampler2D tex;    // sampler set by C code

varying vec2 tex_coord;   // texture coordinate from
                          // vertex shader

varying vec3 lit_color;   // per-vertex lighting from
                          // vertex shader

void main(void)
{
    gl_FragColor = lit_color
        * texture2D(tex, tex_coord);
}
```

16-November-2011

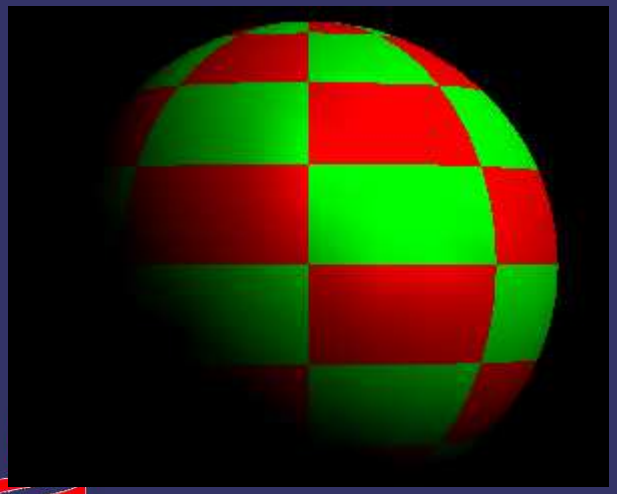
Specular Lighting

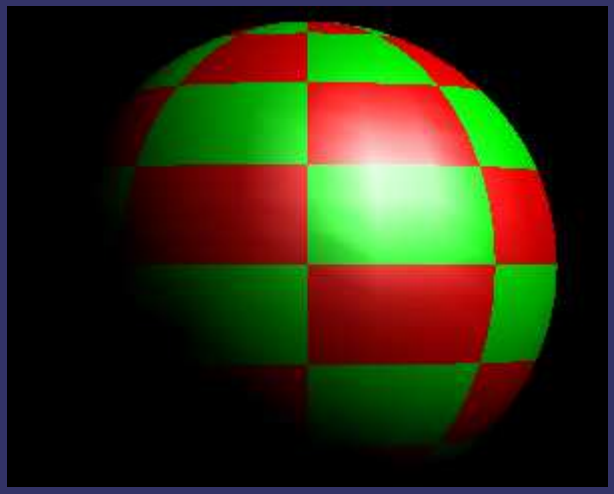
- We perform lighting in the vertex shader and texturing in the fragment shader
  - VS passes a single color to FS, and FS combines it with the texture color
  - Why is this wrong?
  - Texture color is typically a diffuse property
    - It usually supplies C_d in the lighting equation



Specular Lighting

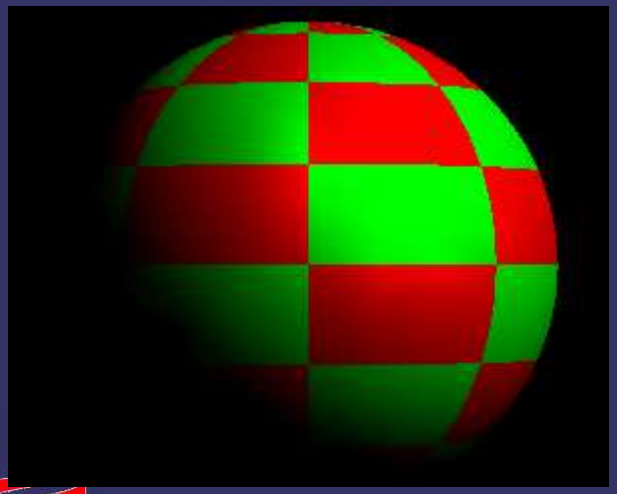
- How can we fix this?

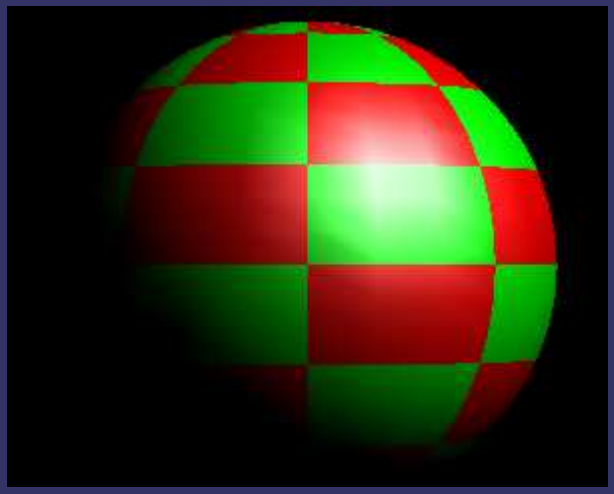




Specular Lighting

- How can we fix this?
  - Perform lighting per-pixel in the fragment shader
  - Send diffuse color and specular color *separately* from the vertex shader to the fragment shader





Specular Lighting

```
uniform sampler2D tex;    // sampler set by C code

varying vec2 tex_coord;   // texture coordinate from
                          // vertex shader

varying vec3 diff_color; // per-vertex diffuse lighting
                          // from vertex shader

varying vec3 spec_color; // per-vertex specular
                          // lighting from vertex shader

void main(void)
{
    gl_FragColor = spec_color
        + (diff_color * texture2D(tex, tex_coord));
}
```

Next week...

- More texture mapping
  - Sampling and filtering

Legal Statement

This work represents the view of the authors and does not necessarily represent the view of Intel or the Art Institute of Portland.

OpenGL is a trademark of Silicon Graphics, Inc. in the United States, other countries, or both.

Khronos and OpenGL ES are trademarks of the Khronos Group.

Other company, product, and service names may be trademarks or service marks of others.

16-November-2011