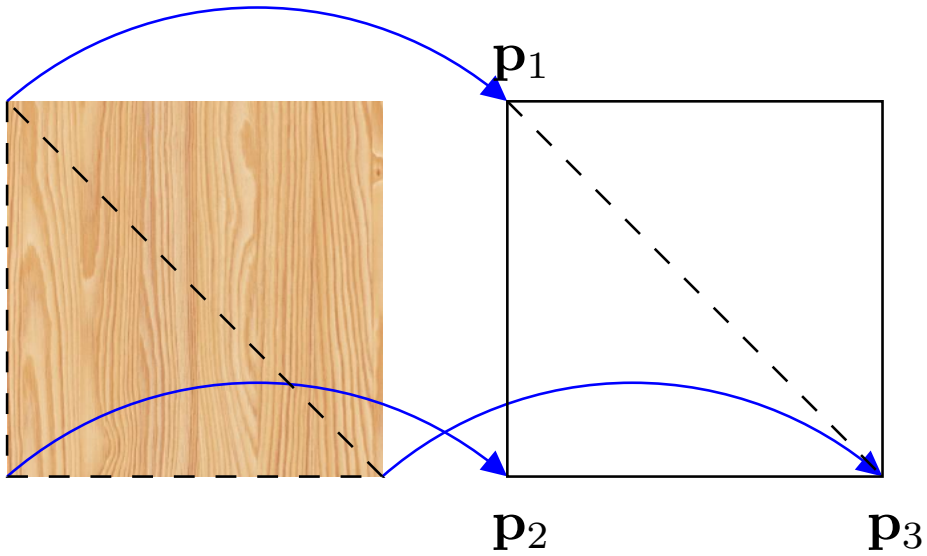
Texture Mapping

Thumrongsak Kosiyatrakul
tkosiyat@cs.pitt.edu

Texture Mapping

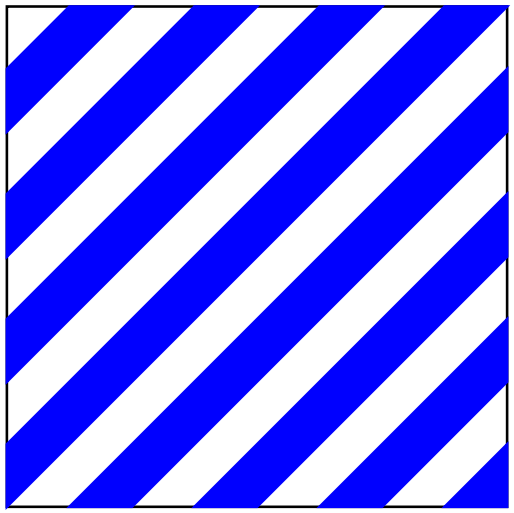
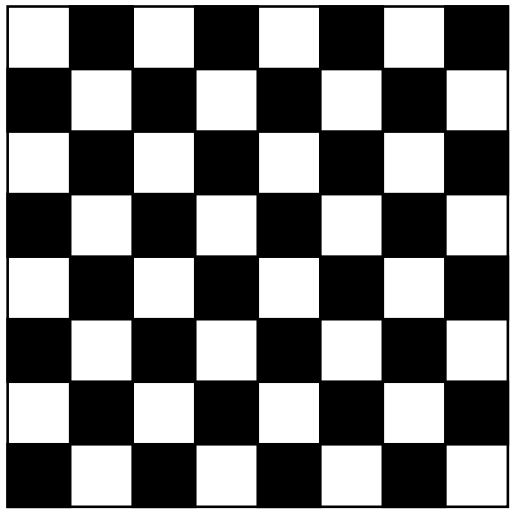
- It is complicate to create a surface with a texture manually
- For example, a cube with wood grain texture
- A simple way is to map a two-dimensional image (texture) to object's surface:



- This method can be easily done in OpenGL

Two-Dimensional Texture Mapping

- A texture is a two-dimensional image
  - Computer generated
  - Image from a camera

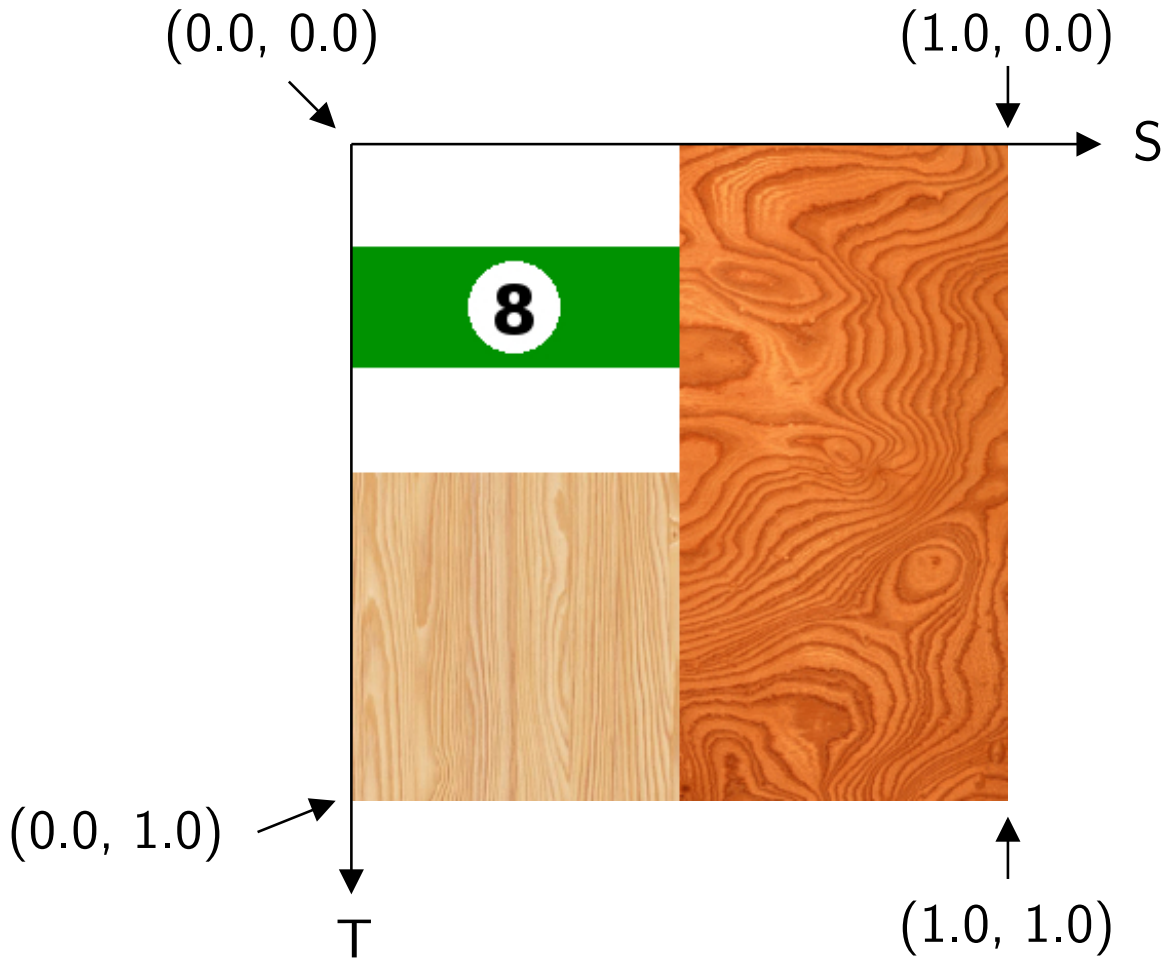

- A texture must be loaded into an application first (pixel by pixel)
  - Each pixel is called a **texel**
  - Each texel consists of three color elements, red, green, and blue
  - Each color element is an unsigned byte `[0..255]`

Two-Dimensional Texture Mapping

- Since the array of texels is in the form of pixel by pixel, it is a discrete type of data
- In OpenGL, we prefer to think of it as a continuous
- A texel, can be referred by a function $T(s,t)$ where $0.0 \leq s \leq 1.0$ and $0.0 \leq t \leq 1.0$
- The coordinate (s,t) is called **texture coordinate**

Two-Dimensional Texture Mapping

- Example



Texture Mapping in OpenGL

- There are three basic steps:
  1. Create a texture image and put it into the GPU's memory
     - The goal is to create an array of texels
     - Can be either computer generated or from an image
  2. Assign texture coordinate to each vertex
     - Manually assigned for a very simple object such as a cube
     - Computationally assigned for a complex object such as a sphere
  3. Apply the texture to each fragment
     - This is done in the fragment shader

Texture Object

- Similar to the vertex array object, we need to generate a texture object:

```
GLuint mytex;

glGenTextures(1, &mytex);
```

- As usual, we need to bind it before we can transfer the data:

```
glBindTexture(GL_TEXTURE_2D, mytex);
```

Texture Array

- A two-dimensional texture is simply a two-dimensional array.
- However, each element should consists of three values for red, green, and blue
  - Traditionally, red, green, and blue values should be in between 0 and 255
  - Suitable for copying from a regular image (RGB)
- For a texture array of 512×512, it can be declared by

```
GLubyte my_texels[512][512][3];
```

- Next is to fill this array of texels with colors
  - computer generated, or
  - read from an image file

Texture from an Image

- A texels can come from an image
- This requires your application to open an image file and transfer all red, green, and blue values for each pixel into your texels array
- It is pretty complicate to write a program to open an image file like `jpeg`
  - However, it is quite easy for a Bit Map (`bmp`) file format
- A free program like GIMP can be used to export raw RGB data to a file
  - Open an image
  - Export As $\rightarrow$ By Extension $\rightarrow$ Raw
  - Select RGB
- Then simply read the file byte by byte RGBRGBRGB... and put them into your texel array.

Read from an Image File

- Suppose you have a raw image file organized as RGBRGBRGB... where the image width and height are known:

```
int width = ...;
int height = ...;

GLubyte my_texels[width][height][3];

FILE *fp;

fp = fopen("image.raw", "r");
if(fp == NULL) {
    printf("Unable to open file\n");
    exit(0);
}

fread(my_texels, width * height * 3, 1, fp);

fclose(fp);
```

Texture Array

- Then we have to specify that the above array will be used as two-dimensional texture:

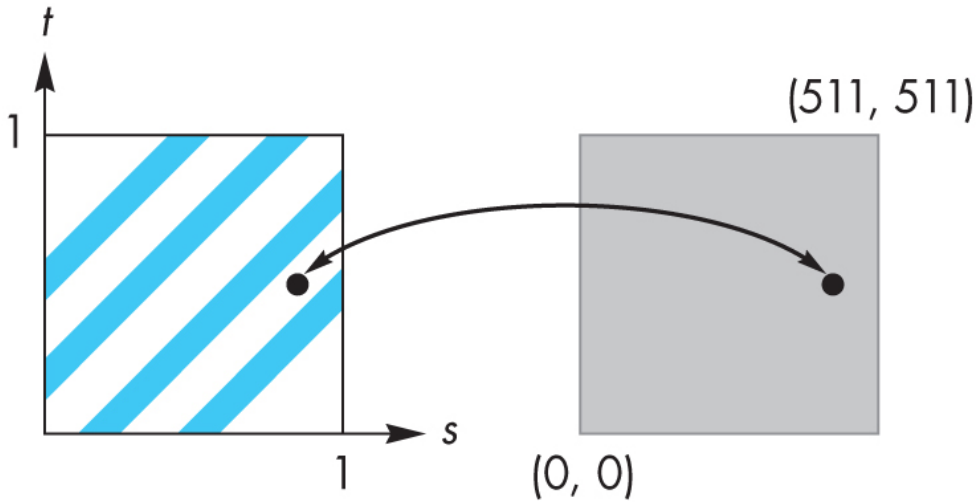
```
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, 512, 512, 0,
             GL_RGB, GL_UNSIGNED_BYTE, my_texels);
```

- Arguments of the `GLTexImage2D()` function are as follows:

```
glTexImage2D(GLenum target,
             GLint level,
             GLint iformat,     // format to store in texture memory
             GLsizei width,     // width of the texture
             GLsizei height,    // height of the texture
             GLint border,      // always 0 (no longer used)
             GLenum format,     // format of texels
             GLenum type,       // type of texels
             GLvoid *tarray)    // pointer to texels
```

Texture Coordinate

- Each vertex must be associated with a texture coordinate
  - A two-dimension coordinate requires two elements s and t (coordinate (s, t))
  - Note that $0.0 \leq s \leq 1.0$ and $0.0 \leq t \leq 1.0$



(x, y) and (s, t) coordinates

- A 2D image generally comes with width and hight in pixels
- But a texture coordinate is in (s, t) where $0.0 \leq s \leq 1.0$ and $0.0 \leq t \leq 1.0$
- Relation between x and s can be easily calculate:
  - Suppose an image width is 512 $(0 \leq x \leq 511)$
  - $x = 0$ maps to $s = 0.0$ and $x = 511$ maps to $s = 1.0$
  - Thus we have

$$s = \frac{x}{511.0} \rightsquigarrow x = 511.0 \times s$$

- Similarly for an image height of 512 pixels:

$$t = \frac{y}{511.0} \rightsquigarrow y = 511.0 \times t$$

(x, y) and (s, t) coordinates

- In other words, given an image of size width $\times$ height, we have

$$s = \frac{x}{\text{width} - 1} \rightsquigarrow x = (\text{width} - 1) \times s$$

and

$$t = \frac{y}{\text{height} - 1} \rightsquigarrow y = (\text{height} - 1) \times t$$

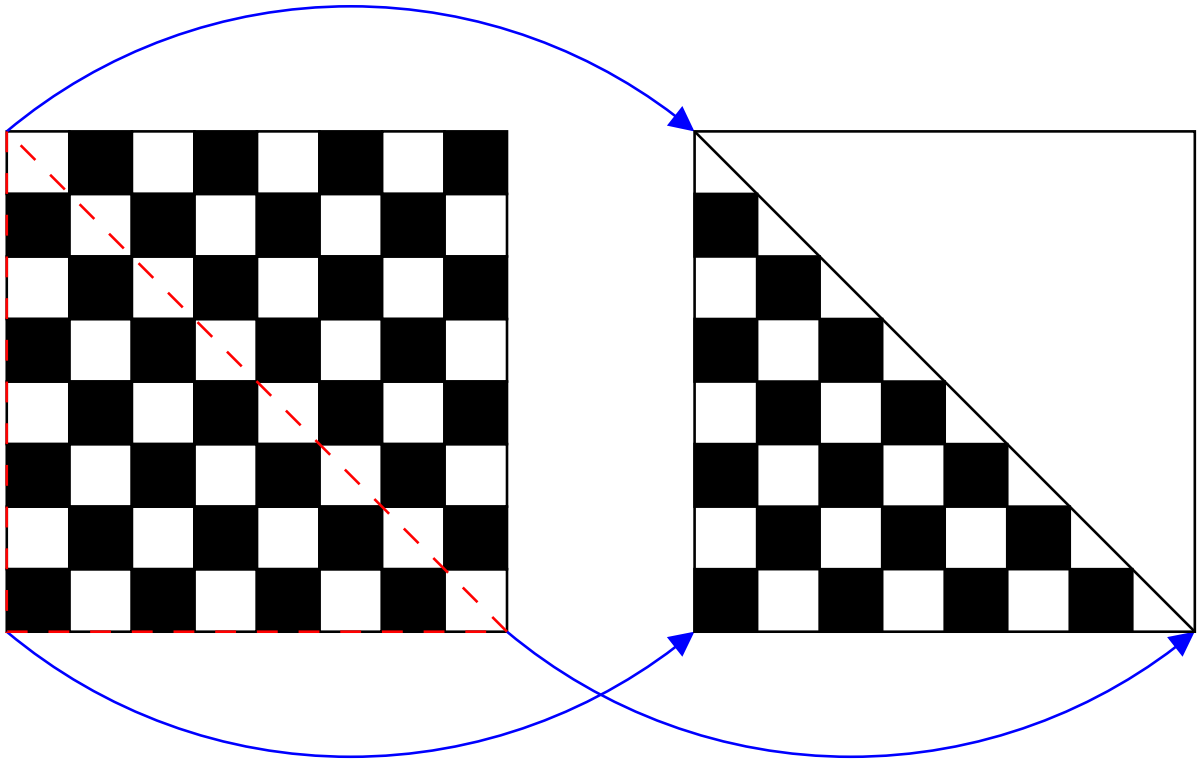
- **Note** that a texture coordinate (s, t) is not always mapped to a center of a pixel
- For example, suppose an image size is 512×512, from the above formula, the texture coordinate $(0.3, 0.7)$ maps to

$$x = 511.0 * 0.3 = 153.3 \qquad y = 511.0 * 0.7 = 357.7$$

- Note that the image coordinate $(153.3, 357.7)$ is not an exact coordinate
- We need to tell OpenGL what to do (e.g., use the nearest pixel $(153, 358)$)

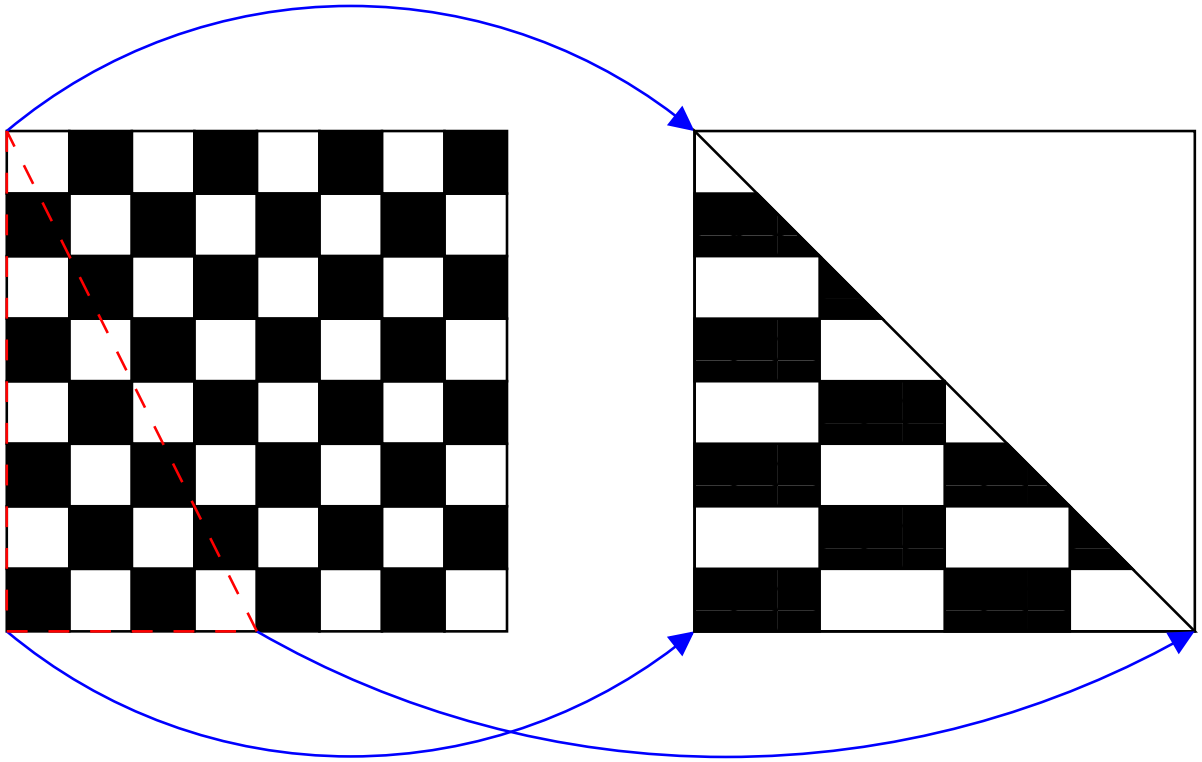
Ratios

- The ratio of the texture and the surface that it mapped to do not have to be the same
- Same ratio:



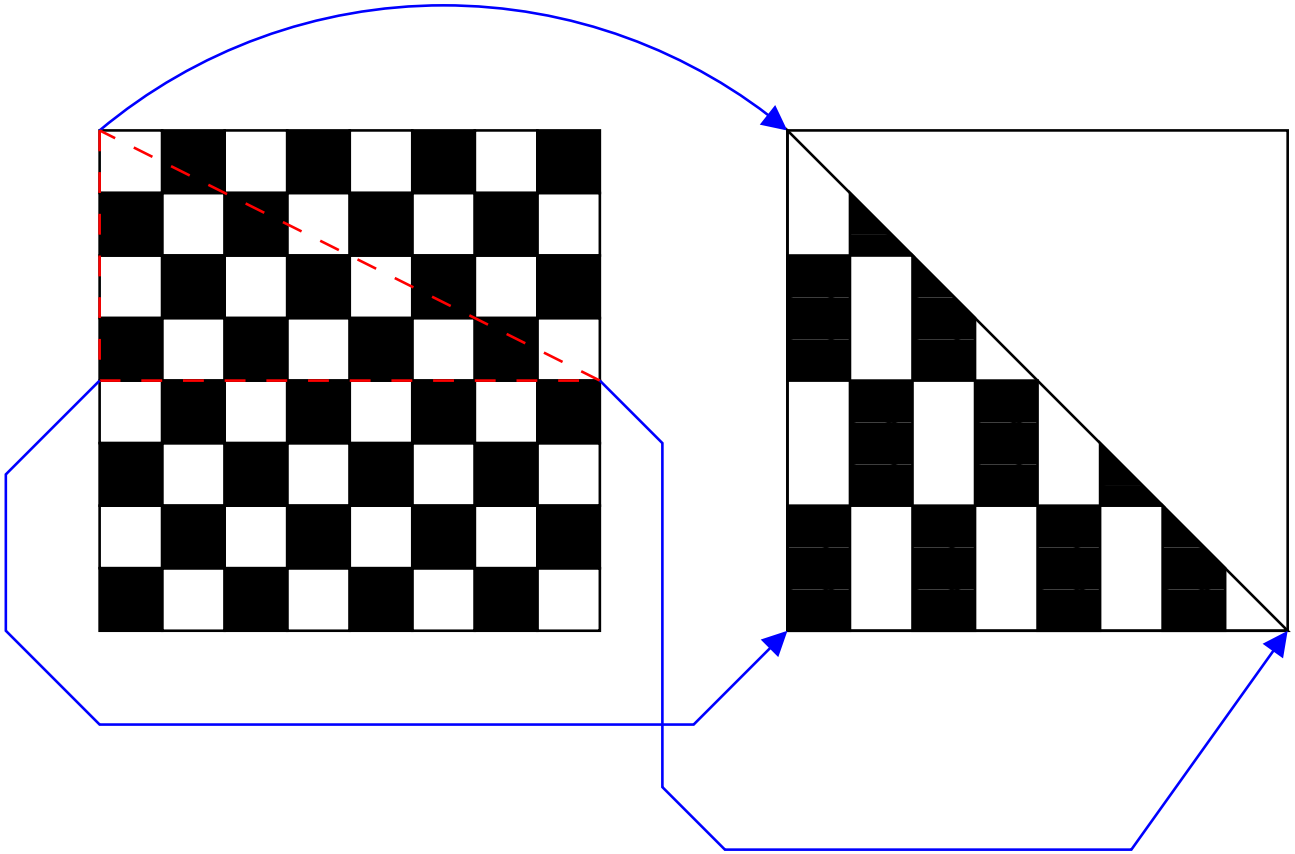
Ratios

- The ratio of the texture and the surface that it mapped to do not have to be the same
- Stretched:



Ratios

- The ratio of the texture and the surface that it mapped to do not have to be the same
- Stretched:



Multiple Textures

- We generally put multiple textures into one texel array
- For example, texture for each side of a cube

Texture Coordinate

- The texture coordinates is simply a two-dimension array of size $n \times 2$ where n is the number of vertices

```
GLfloat tex_coord[36][2];    // for a cube
```

- Next, simply assign a texture coordinate to each vertex:

```
tex_coord[0][0] = 0.0;
tex_coord[0][1] = 0.0;

tex_coord[1][0] = 0.0;
tex_coord[1][1] = 1.0;

tex_coord[2][0] = 1.0;
tex_coord[2][1] = 1.0;
:
```

- This can be done manually or by an algorithm

Texture Coordinate

- Once we have the coordinates for vertices, we simply need to send it to the graphics pipeline just like the array of vertices:

```
GLuint vTexCoord = glGetAttribLocation(program, "vTexCoord");
glEnableVertexAttribArray(vTexCoord);
glVertexAttribPointer(vTexCoord, 2, GL_FLOAT, GL_FALSE, 0,
                      (GLvoid *) 0 + verticesSize);
```

- **Note** that the offset value (the last argument) depends on what you send before this array
  - In this case, only two arrays will be sent, array of vertices, and array of coordinates

Texture Coordinate

- Suppose `verticesSize` is the size of the array of vertices in bytes and `texcoordSize` is the size of the array of texture coordinate in bytes:

```
GLuint vao;
glGenVertexArrays(1, &vao);
glBindVertexArray(vao);

GLuint buffer;
glGenBuffers(1, &buffer);
glBindBuffer(GL_ARRAY_BUFFER, buffer);
glBufferData(GL_ARRAY_BUFFER, verticesSize + texcoordSize, NULL, GL_STATIC_DRAW);
glBufferSubData(GL_ARRAY_BUFFER, 0, verticesSize, vertices);
glBufferSubData(GL_ARRAY_BUFFER, verticesSize, texcoordSize, tex_coords);

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

GLuint vTexCoord = glGetAttribLocation(program, "vTexCoord");
glEnableVertexAttribArray(vTexCoord);
glVertexAttribPointer(vTexCoord, 2, GL_FLOAT, GL_FALSE, 0, (GLvoid *) 0 + verticesSize);
```

Vertex Shader

- The vertex shader simply pass the texture coordinates to be interpolated (similar to color)

```
#version 120

attribute vec4 vPosition;
attribute vec2 vTexCoord;

varying vec2 texCoord;

uniform mat4 ctm;

void main()
{
    texCoord = vTexCoord;
    gl_Position = ctm * vPosition;
}
```

Fragment Shader

- The fragment shader needs the texels **location** which can be past as a uniform variable

```
#version 120

varying vec2 texCoord;

uniform sampler2D texture;

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

Linking the Texture Object

- We need to link the texture object (`mytex`) with the fragment shader

```
GLuint tex_loc;

tex_loc = glGetUniformLocation(program, "texture");

glUniform1i(tex_loc, 0);
```

  which is done in the `init()` function
- The second parameter (0) of the `glUniform1i()` function, indicates the first default texture.

More about Texture Coordinate

- It is possible to assign the coordinate outside the range (0.0, 1.0)
- If we want to wrap around for both s and t, set the texture parameters as shown below:

```
glTexParameterf( GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT );
glTexParameterf( GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT );
```

- The above functions should be called right after `glBindTexture()` function.

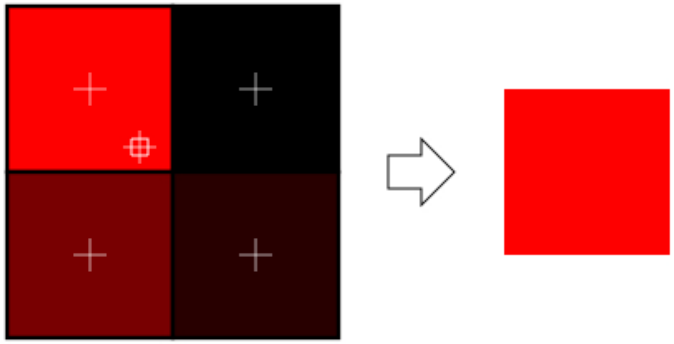
Texture Sampling

- A texture coordinate may not map directly to the array of texels
  - The mapped texel may be larger than one pixel, or
  - the mapped texel may be smaller than one pixel.
- Zoom in or enlarge an object (**Magnification**)
  - Multiple screen pixels map to a single texel
- Zoom out or shrink an object (**Minification**)
  - Single screen pixel maps to multiple texels
- We can simply tell OpenGL to use the nearest point of sampling:

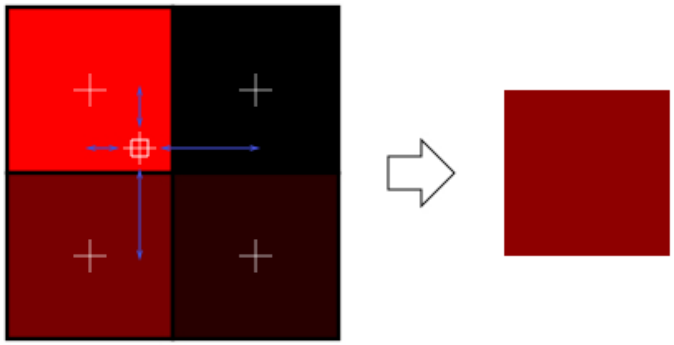
```
glTexParameterf( GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glTexParameterf( GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
```

Texture Sampling

- Nearest



- Linear



The init() Function

```
GLubyte my_texels[...][...][3];
:
GLfloat tex_coords[...][2];
:
GLuint program = initShader("vshader.glsl", "fshader.glsl");
glUseProgram(program);

GLuint mytex;
glGenTextures(1, &mytex);
glBindTexture(GL_TEXTURE_2D, mytex);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0, GL_RGB, GL_UNSIGNED_BYTE, my_texels);
glTexParameterf( GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT );
glTexParameterf( GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT );
glTexParameterf( GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST );
glTexParameterf( GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST );

GLuint vao;
glGenVertexArrays(1, &vao);
glBindVertexArray(vao);

GLuint buffer;
glGenBuffers(1, &buffer);
glBindBuffer(GL_ARRAY_BUFFER, buffer);
glBufferData(GL_ARRAY_BUFFER, verticesSize + texcoordSize, NULL, GL_STATIC_DRAW);
glBufferSubData(GL_ARRAY_BUFFER, 0, verticesSize, vertices);
glBufferSubData(GL_ARRAY_BUFFER, verticesSize, texcoordSize, tex_coords);

GLuint vPosition = glGetAttribLocation(program, "vPosition");
glEnableVertexAttribArray(vPosition);
glVertexAttribPointer(vPosition, 4, GL_FLOAT, GL_FALSE, 0, BUFFER_OFFSET(0));
GLuint vTexCoord = glGetAttribLocation(program, "vTexCoord");
glEnableVertexAttribArray(vTexCoord);
glVertexAttribPointer(vTexCoord, 2, GL_FLOAT, GL_FALSE, 0, (GLvoid *) 0 + verticesSize);
```