Texture Mapping

Frame Buffer

- Figure 7.2 shows the OpenGL frame buffer and some of its constituent parts. When we work with the frame buffer, we usually work with one constituent buffer at a time. Thus, we shall use the term *buffer* in what follows to mean a particular buffer within the frame buffer. Each of these buffers is $n \times m$ and is k bits deep. However, k can be different for each buffer. For a color buffer, its k is determined by how many colors the system can display, usually 24 for RGB displays and 32 for RGBA displays.

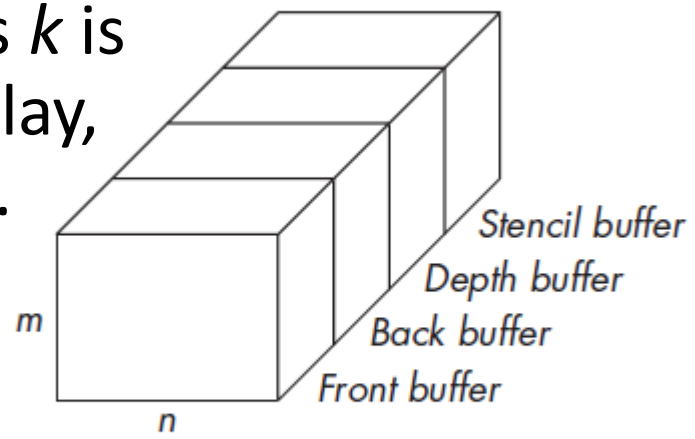


FIGURE 7.2 OpenGL frame buffer.

Color Palette

- Framebuffers have traditionally supported a wide variety of color modes. Due to the expense of memory, most early framebuffers used 1-bit (2-color), 2-bit (4-color), 4-bit (16-color) or 8-bit (256-color) color depths.
- Here is a typical indexed 256-color image and its own palette

Digital Images

- if we are working with RGB images, we usually represent each of the color components with 1 byte whose values range from 0 to 255. Thus, we might declare a 512 × 512 image in our application program as
- GLubyte myimage[512][512][3];
- or, if we are using a floating-point representation,
- typedef vec3 color3;
- color3 myimage[512][512];

Digital Images

- For example, suppose that we want to create a 512 × 512 image that consists of an 8 × 8 checkerboard of alternating red and black squares, such as we might use for a game. The following code will work:

```
class mRGB
{
public:
        uchar r,g,b,a;
        mRGB(){r = g = b = 0,a=255;}
};
```

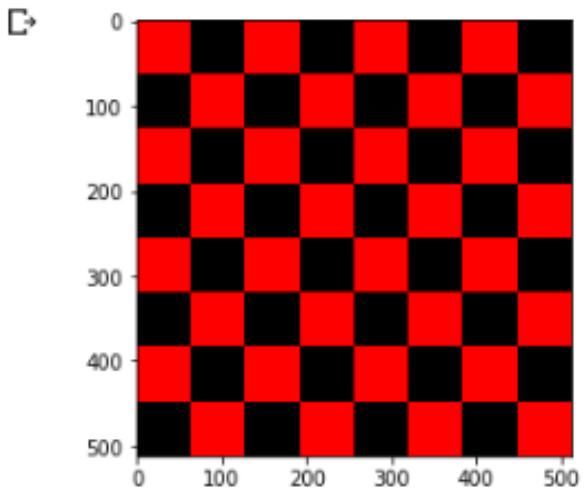
Digital Images

```
nRows=nCols=64;
pixel = new mRGB[3*nRows*nCols];
long count=0;
for(int i=0;i<nRows;i++)
        for(int j=0;j<nCols;j++)
        {
                int c=(( (i/8)+(j/8) ) %2)*255;
                pixel[count].r=c;        //red
                pixel[count].g=c;        //green
                pixel[count++].b=0;      //blue
        }
```

Digital Images

```
[50] import numpy as np
     img = np.zeros([512,512,3])
     for i in range(512):
       for j in range(512):
         c= int((int(i/64)+int(j/64)) %2);
         if c == 0:
           img[i][j][0] = 1
         else:
           img[i][j][0] = 0


     import matplotlib.pyplot as plt
     plt.imshow(img)
     plt.show()
```



Mapping Methods

- Consider, for example, the task of creating a virtual orange by computer. Our first attempt might be to start with a sphere. Although it might have the correct overall properties, such as shape and color, it would lack the fine surface detail of the real orange. If we attempt to add this detail by adding more polygons to our model, even with hardware capable of rendering tens of millions of polygons per second, we can still overwhelm the pipeline. as the implementation renders a surface—be it a polygon or a curved surface—it generates sets of fragments, each of which corresponds to a pixel in the frame buffer. Fragments carry color, depth, and other information that can be used to determine how they contribute to the pixels to which they correspond. As part of the rasterization process, we must assign a shade or color to each fragment.

Mapping Methods

- An alternative is not to attempt to build increasingly more complex models, but rather to build a simple model and to add detail as part of the rendering process. There are three major techniques:
- Texture mapping
- Bump mapping
- Environment mapping

Texture Mapping

- **Texture mapping** uses an image (or texture) to influence the color of a fragment. Textures can be specified using a fixed pattern, such as the regular patterns often used to fill polygons; by a procedural texture-generation method; or through a digitized image. In all cases, we can characterize the resulting image as the mapping of a texture to a surface, as shown in Figure 7.8, as part of the rendering of the surface.

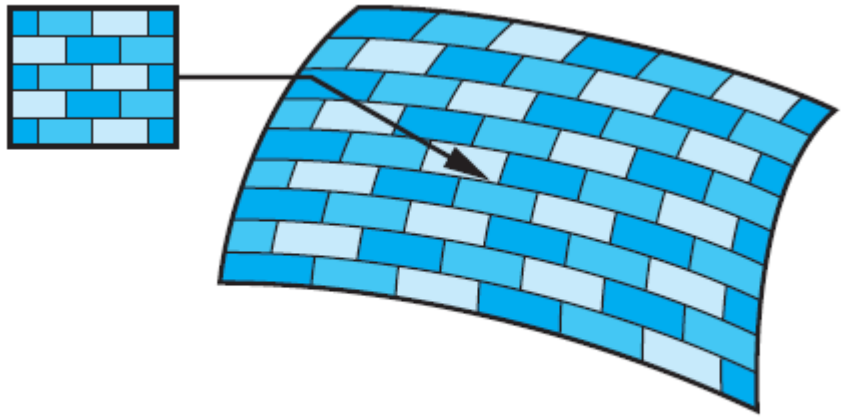
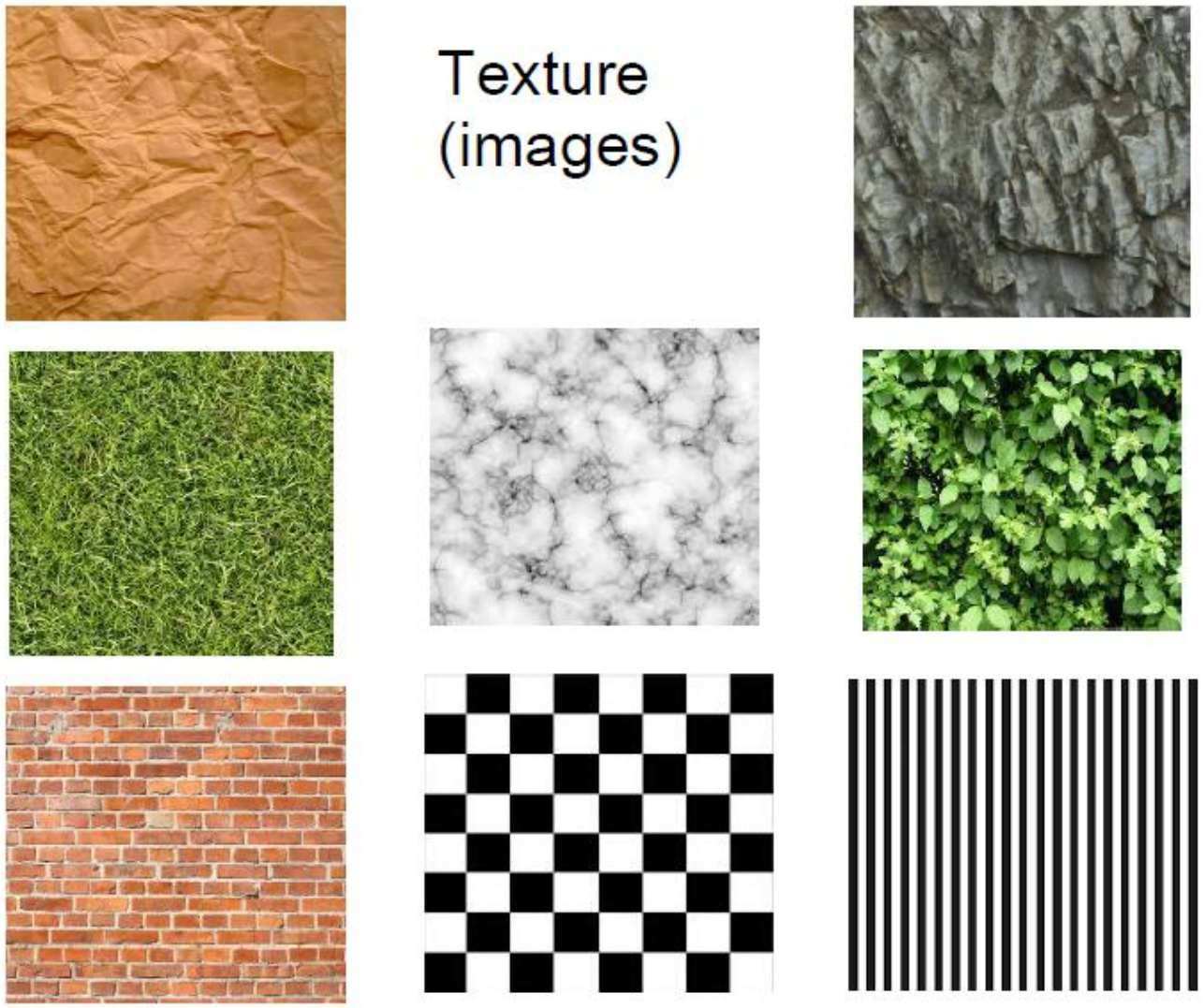


FIGURE 7.8 Texture mapping a pattern to a surface.

Bump Mapping & Environment Mapping

- **bump maps** distort the normal vectors during the shading process to make the surface appear to have small variations in shape, such as the bumps on a real orange.
- **Reflection maps**, or **environment maps**, allow us to create images that have the appearance of reflected materials without our having to trace reflected rays. In this technique, an image of the environment is painted onto the surface as that surface is being rendered.
- Color Plate 15 uses a texture map for the surface of the table; Color Plate 10 uses texture mapping to create a brick pattern.

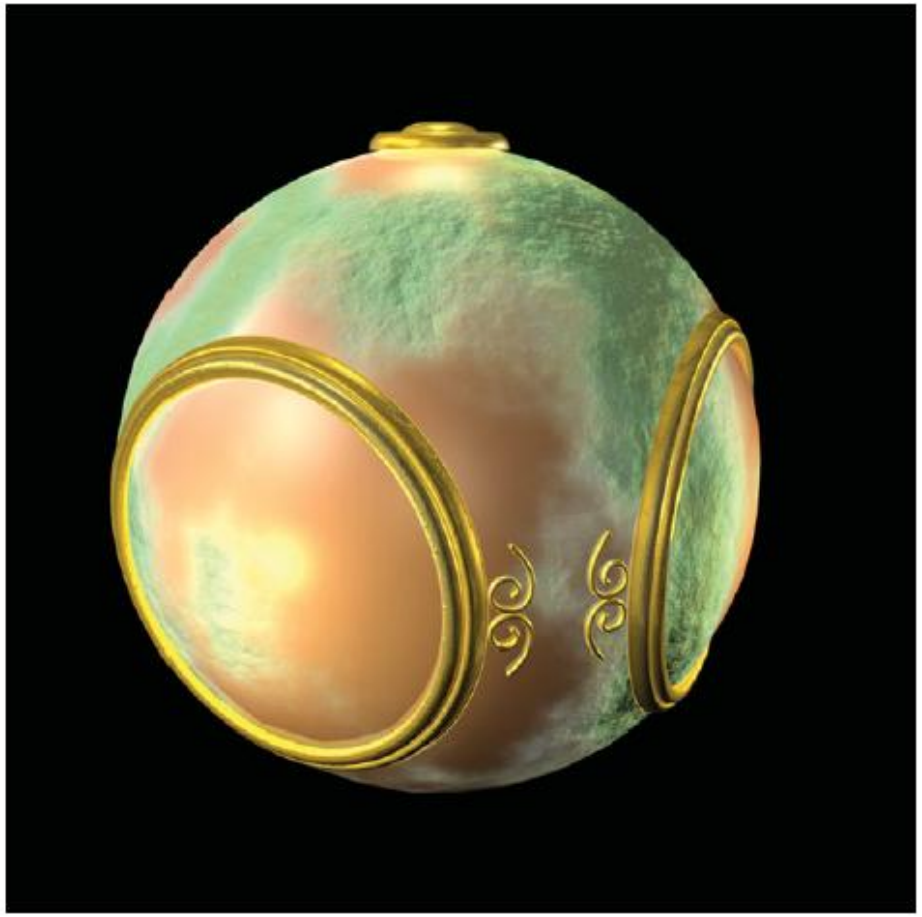
Example of Texture



Example of Bump Mapping

Color Plate 6 Rendering of sun object showing bump map.

(Courtesy of Fulldome Project, University of New Mexico.)



Example of Environment Mapping

Color Plate 7 Rendering of sun object with an environment map.

(Courtesy of Fulldome Project, University of New Mexico.)

Example of Texture Mapping

Color Plate 10 Perspective view of interior of temple.

(Courtesy of Richard Nordhaus, Architect, Albuquerque, NM.)

Example of Texture Mapping

Color Plate 15 Rendering using ray tracer.

(Courtesy of Patrick McCormick.)

Texture Mapping

- Textures are patterns. They can range from regular patterns, such as stripes and checkerboards, to the complex patterns that characterize natural materials. In the real world, we can distinguish among objects of similar size and shape by their textures.

Two Dimensional Texture Mapping

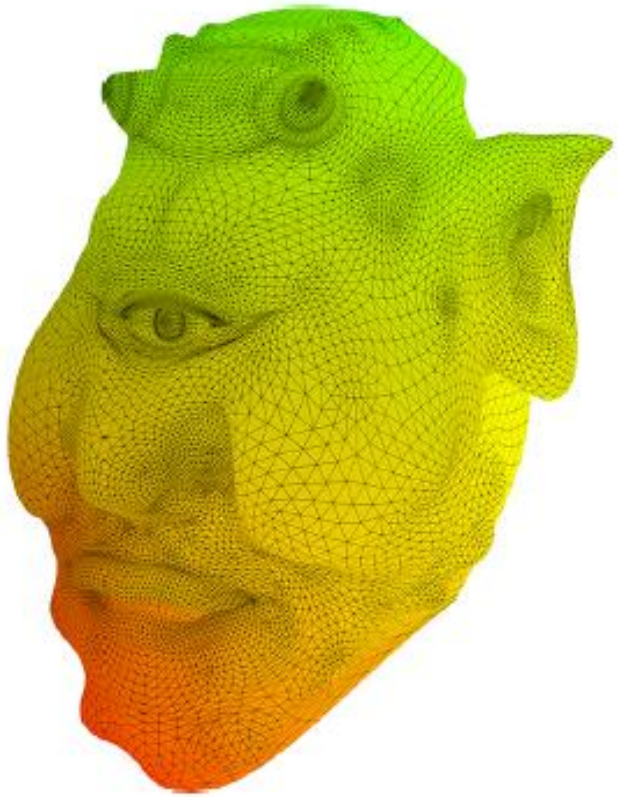
- Although there are multiple approaches to texture mapping, all require a sequence of steps that involve mappings among three or four different coordinate systems. At various stages in the process, we shall be working with:
- screen coordinates, where the final image is produced;
- object coordinates, where we describe the objects upon which the textures will be mapped;
- texture coordinates, which we use to locate positions in the texture;
- and parametric coordinates, which we use to help us define curved surfaces.

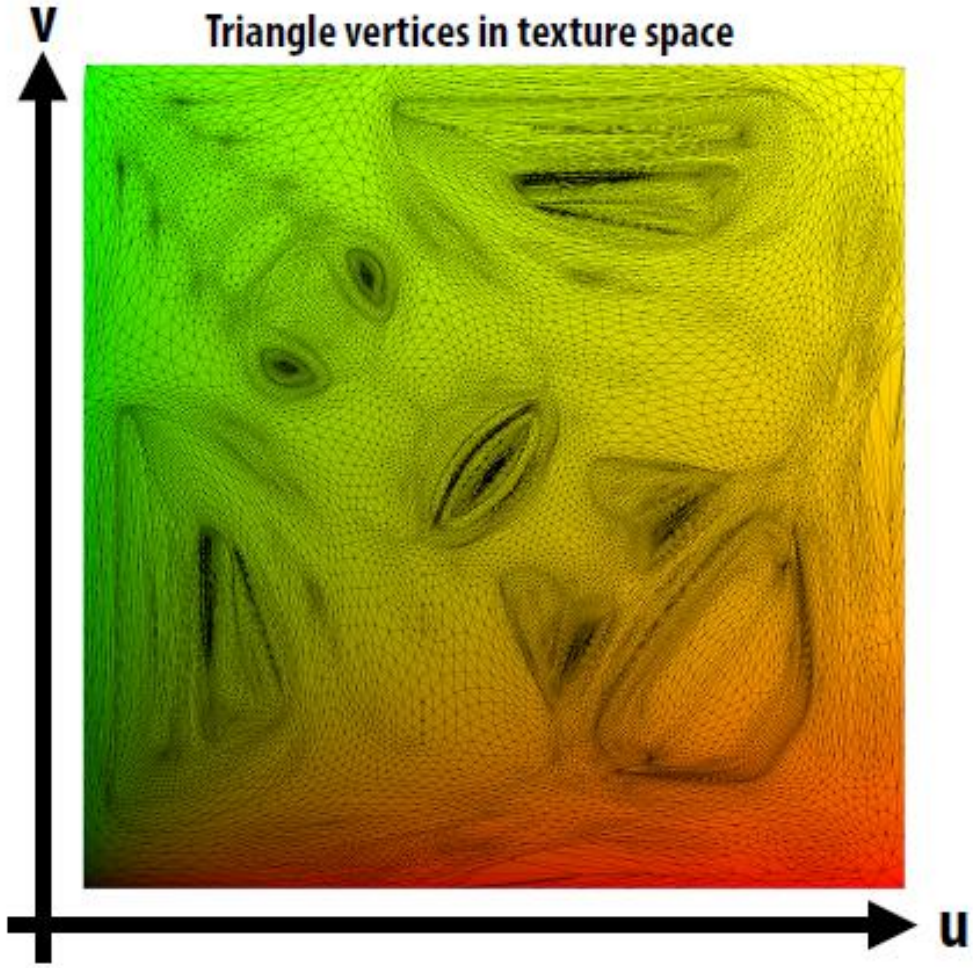
What is a texture map?

- Practical: “A way to slap an image on a model.”
- Better: “A mapping from any function onto a surface in three dimensions.”
- Most general: “The mapping of any image into multidimensional space.”

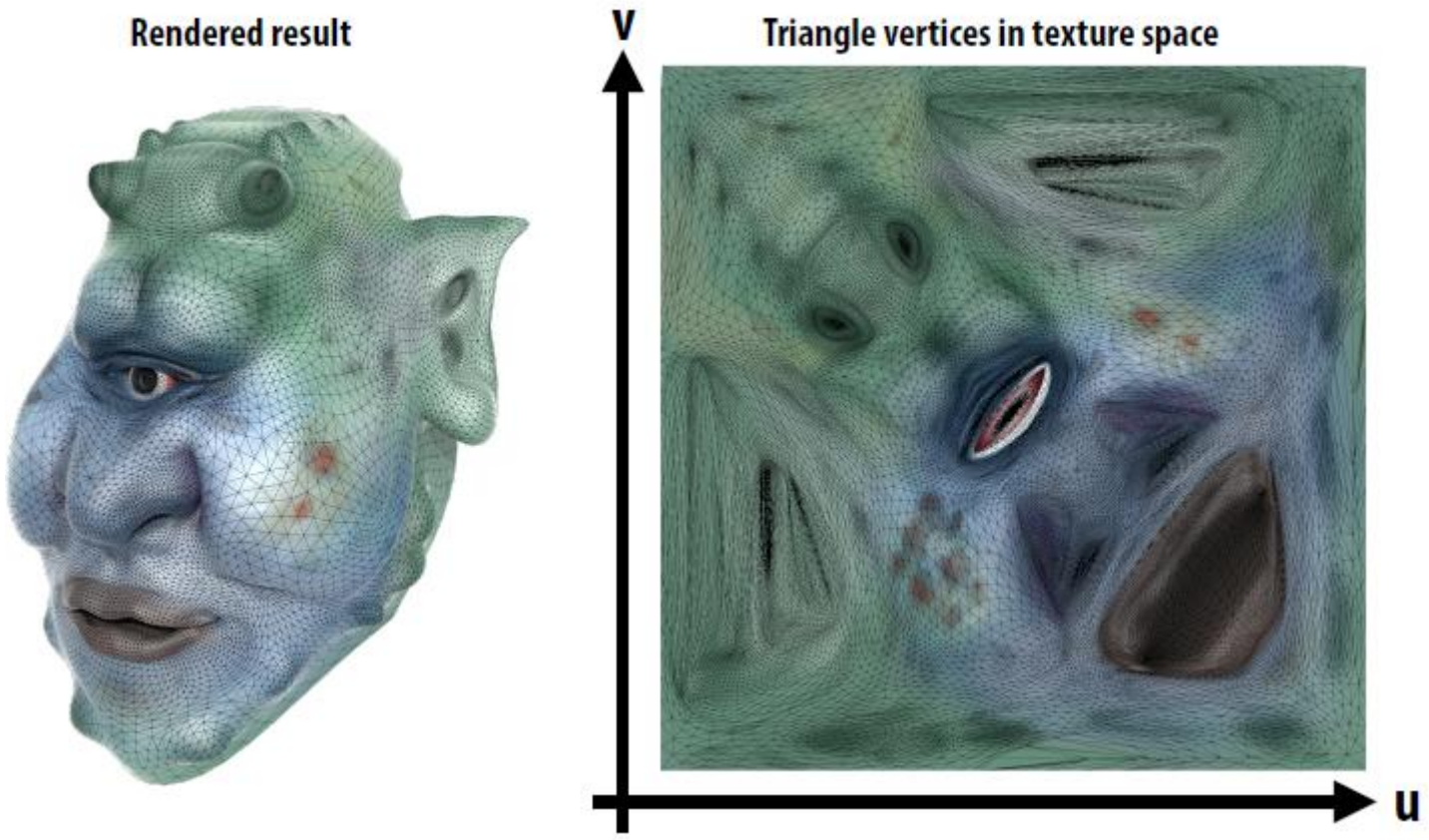
Texture Mapping

Visualization of texture coordinates





Texture Mapping



Two Dimensional Texture Mapping

- In most applications, textures start out as two-dimensional images of the sorts we introduced in page-12 of this slide. Thus, they might be formed by application programs or scanned in from a photograph, but, regardless of their origin, they are eventually brought into processor memory as arrays. We call the elements of these arrays **texels**, or texture elements, rather than pixels to emphasize how they will be used. However, at this point, we prefer to think of this array as a continuous rectangular two-dimensional texture pattern *T(s, t)*. The independent variables *s* and *t* are known as **texture coordinates**. With no loss of generality, we can scale our texture coordinates to vary over the interval [0,1].

Two Dimensional Texture Mapping

- A **texture map** associates a texel with each point on a geometric object that is itself mapped to screen coordinates for display. If the object is represented in homogeneous or *(x, y, z, w)* coordinates, then there are functions such that
- $x = x(s, t)$,
- $y = y(s, t)$,
- $z = z(s, t)$,
- $w = w(s, t)$.

Two Dimensional Texture Mapping

- One of the difficulties we must confront is that although these functions exist conceptually, finding them may not be possible in practice. In addition, we are worried about the inverse problem: Having been given a point (x, y, z) or (x, y, z , w) on an object, how do we find the corresponding texture coordinates, or equivalently, how do we find the "inverse" functions
- $s = s(x, y, z , w)$,
- $t = t(x, y, z , w)$
- to use to find the texel $T(s, t)$?

Two Dimensional Texture Mapping

- If we define the geometric object using parametric (*u*, *v*) surfaces, there is an additional mapping function that gives object coordinate values, *(x, y, z)* or *(x, y, z , w)* in terms of *u* and *v*.
- we also need the mapping from parametric coordinates *(u, v)* to texture coordinates and sometimes the inverse mapping from texture coordinates to parametric coordinates.

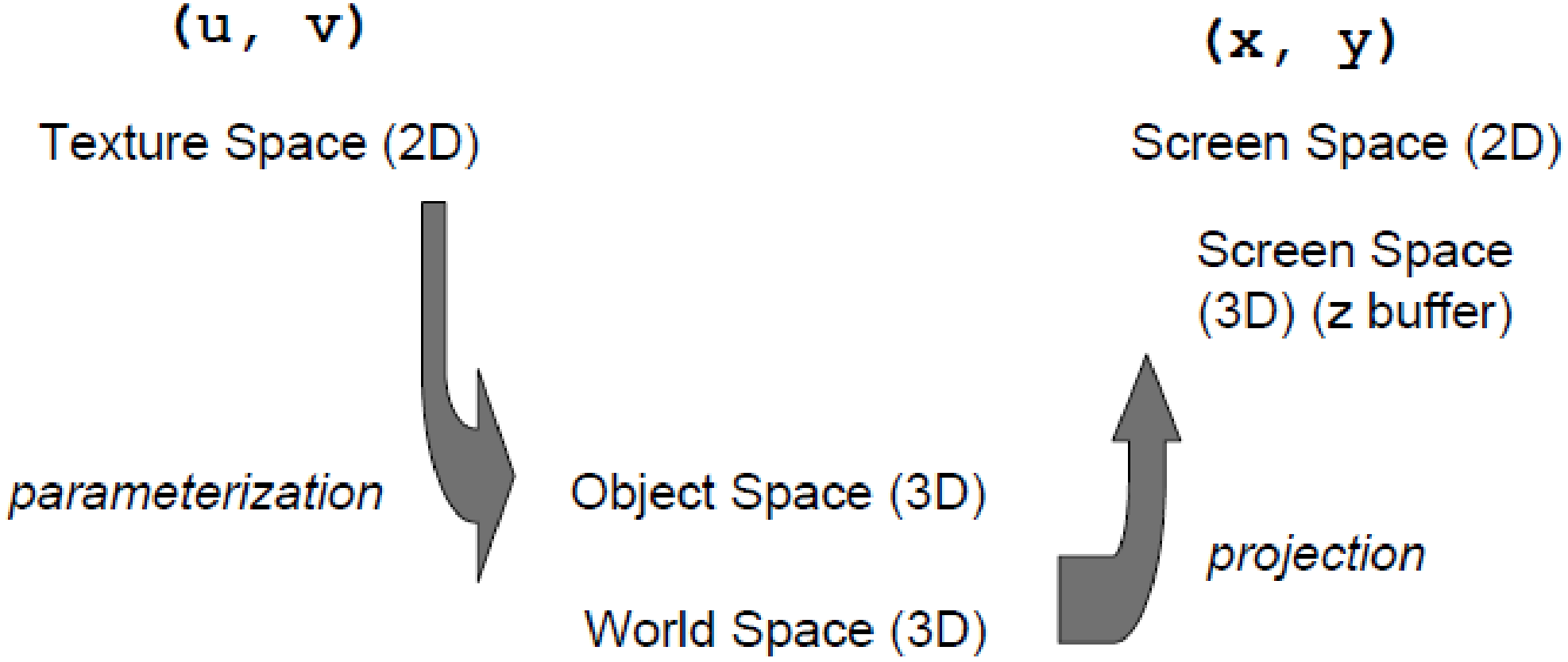
Two Dimensional Texture Mapping

- We also have to consider the projection process that take us from object coordinates to screen coordinates, going through eye coordinates, clip coordinates, and window coordinates along the way. We can abstract this process through a function that takes a texture coordinate pair *(s, t)* and tells us where in the color buffer the corresponding value of *T(s, t)* will make its contribution to the final image. Thus, there is a mapping of the form
- $x_s = x_s(s, t)$,
- $y_s = y_s(s, t)$
- into coordinates, where (x_s , y_s) is a location in the color buffer.

Two Dimensional Texture Mapping

- One way to think about texture mapping is in terms of two concurrent mappings: the first from texture coordinates to parametric coordinates, and the second from parametric coordinates to object coordinates. A third mapping takes us from object coordinates to screen coordinates.

Two Dimensional Texture Mapping



Two Dimensional Texture Mapping

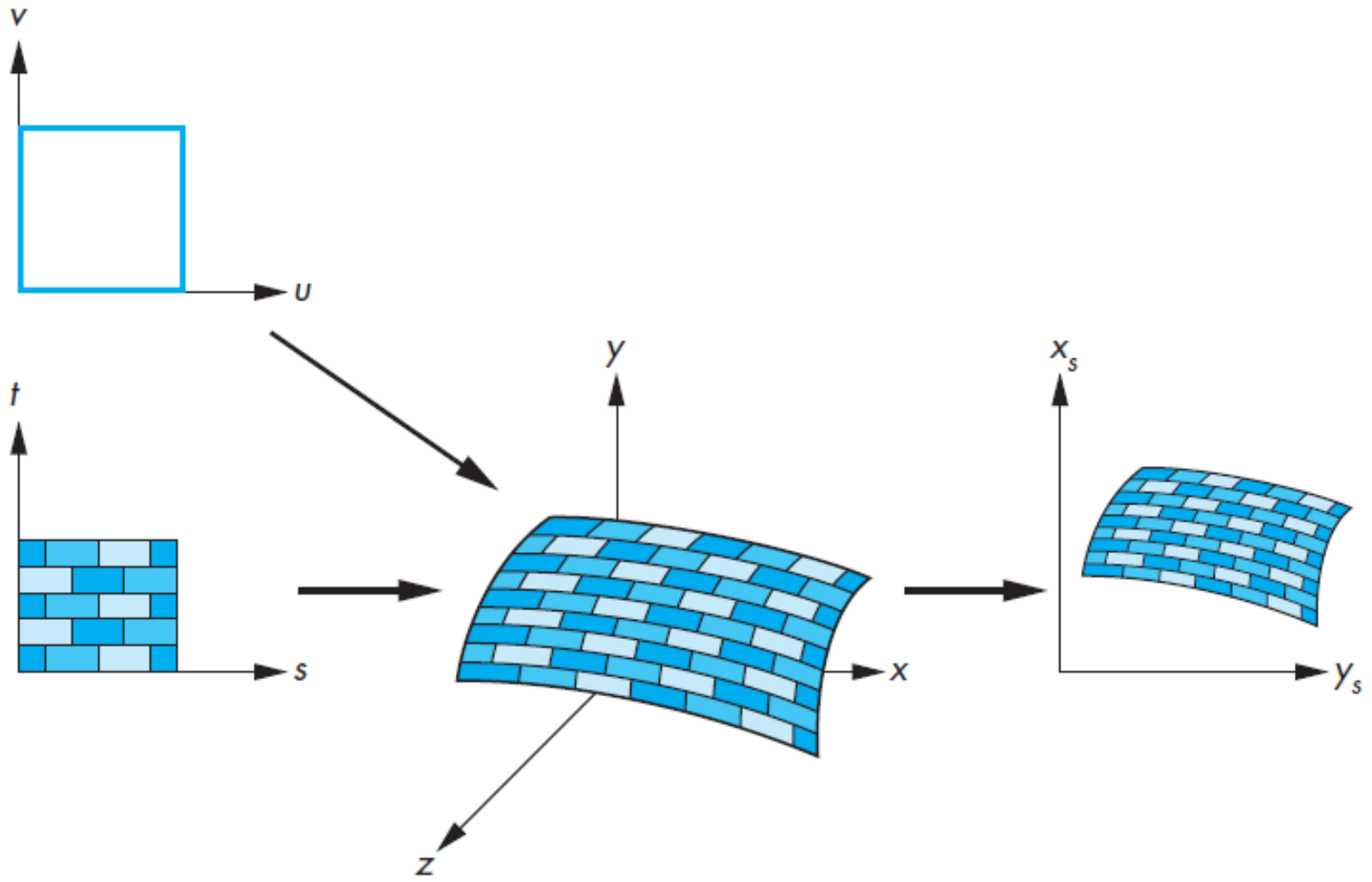
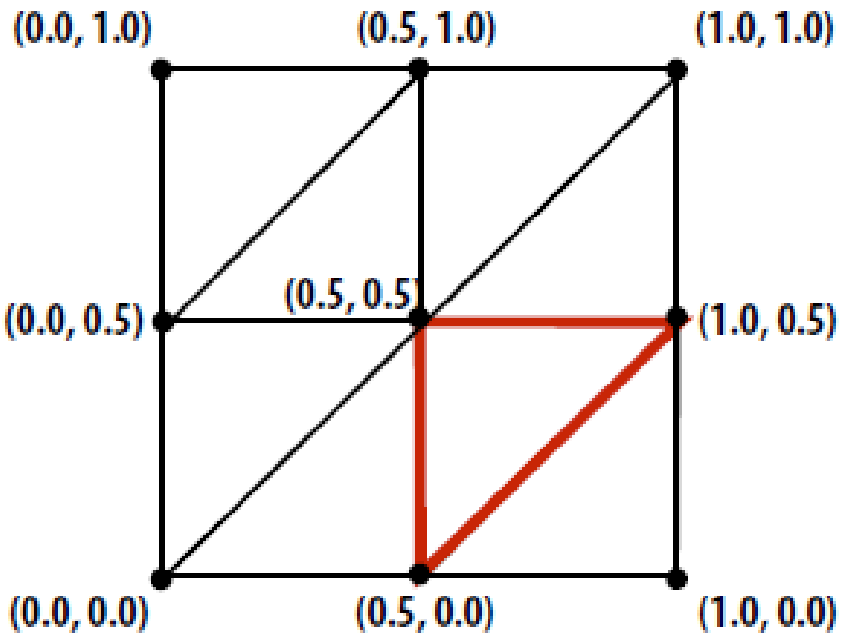


FIGURE 7.9 Texture maps for a parametric surface.

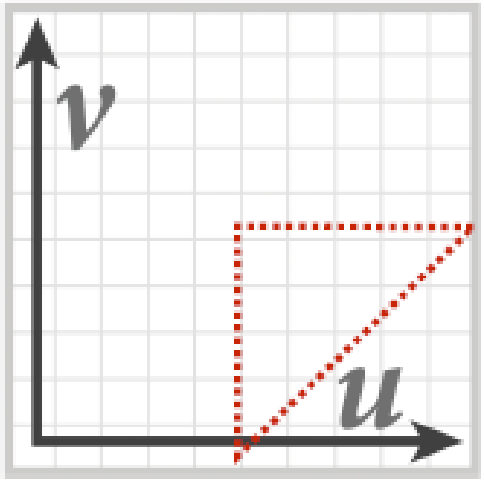
Two Dimensional Texture Mapping

- If we assume that the values of *T* are RGB color values, we can use these values either to modify the color of the surface that might have been determined by a lighting model or to assign a color to the surface based on only the texture value. This color assignment is carried out as part of the assignment of fragment colors.

Two Dimensional Texture Mapping

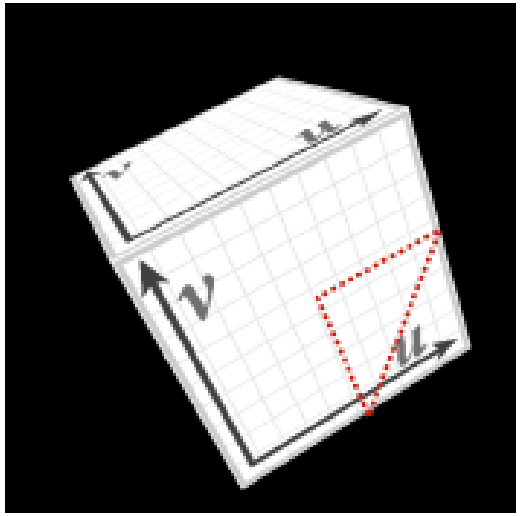


Eight triangles (one face of cube) with surface parameterization provided as per-vertex texture coordinates.



myTex(u,v) is a function defined on the $[0,1]^2$ domain (represented by 2048x2048 image)

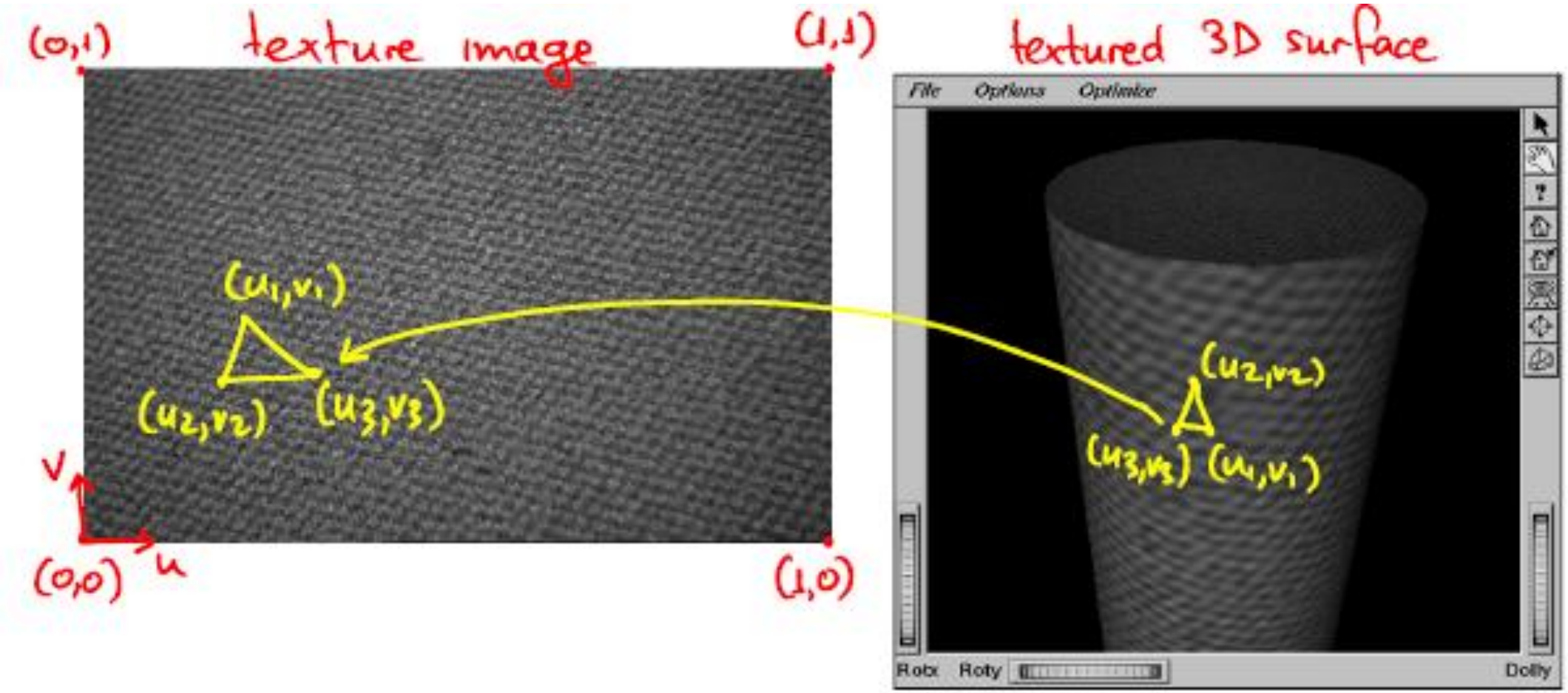
Location of highlighted triangle in texture space shown in red.



Final rendered result (entire cube shown).

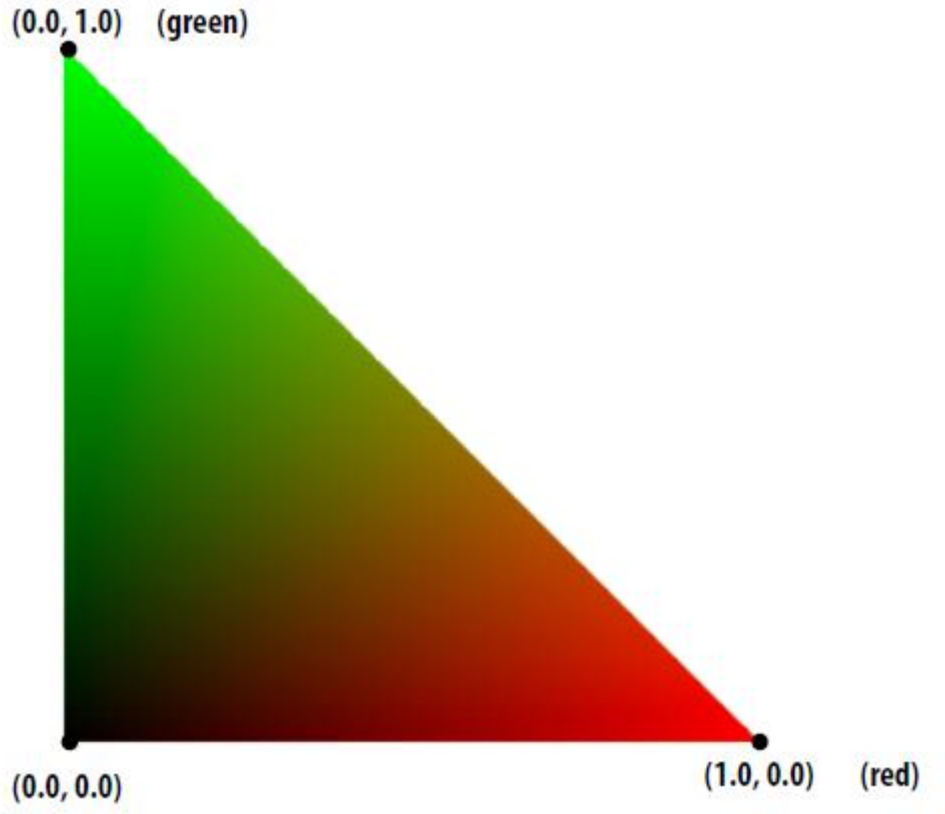
Location of triangle after projection onto screen shown in red.

Two Dimensional Texture Mapping

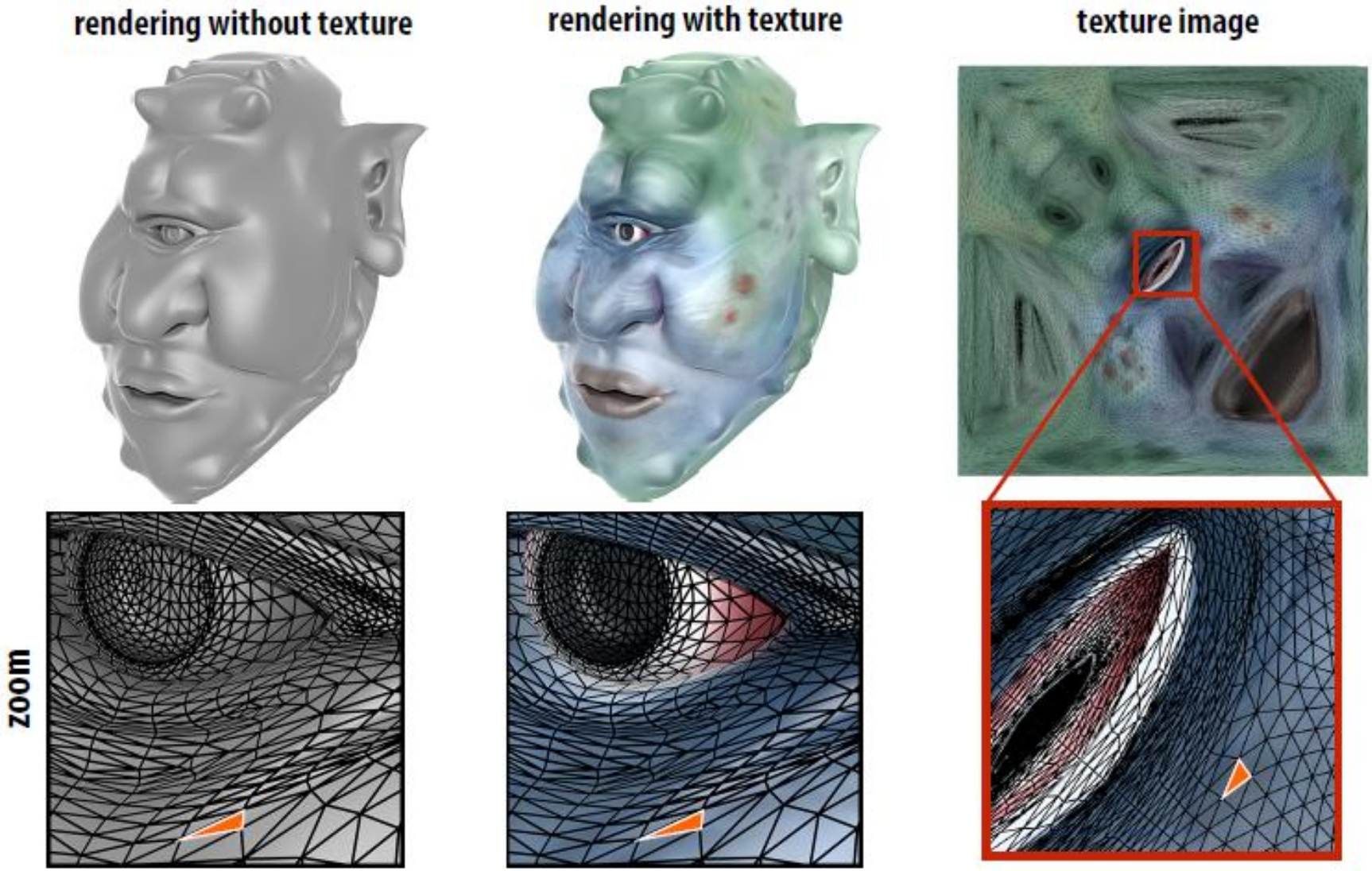


Visualization of texture coordinates

- **Texture coordinates linearly interpolated over triangle**

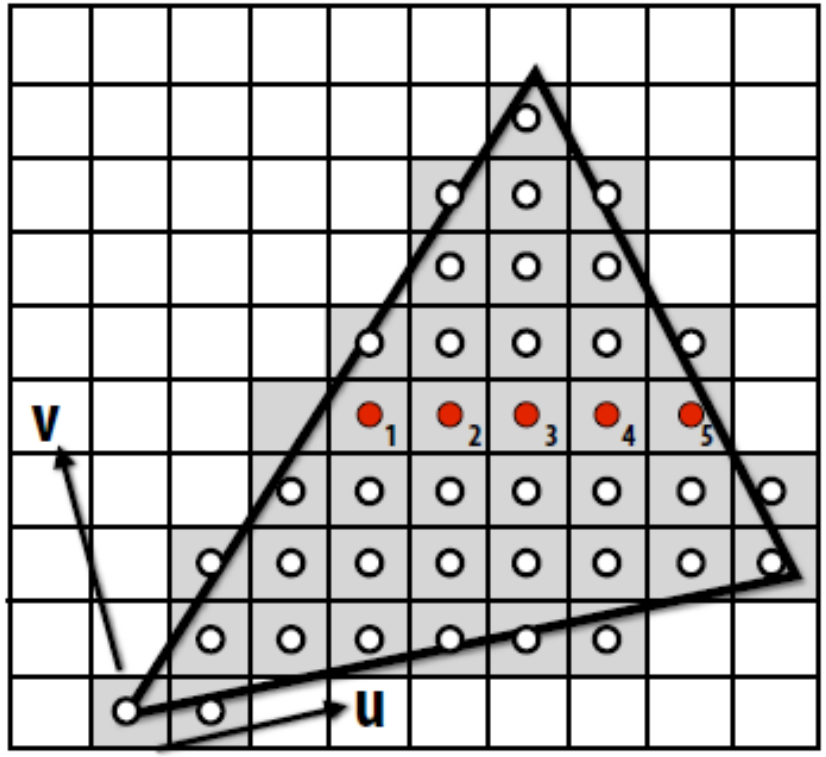


Texture mapping



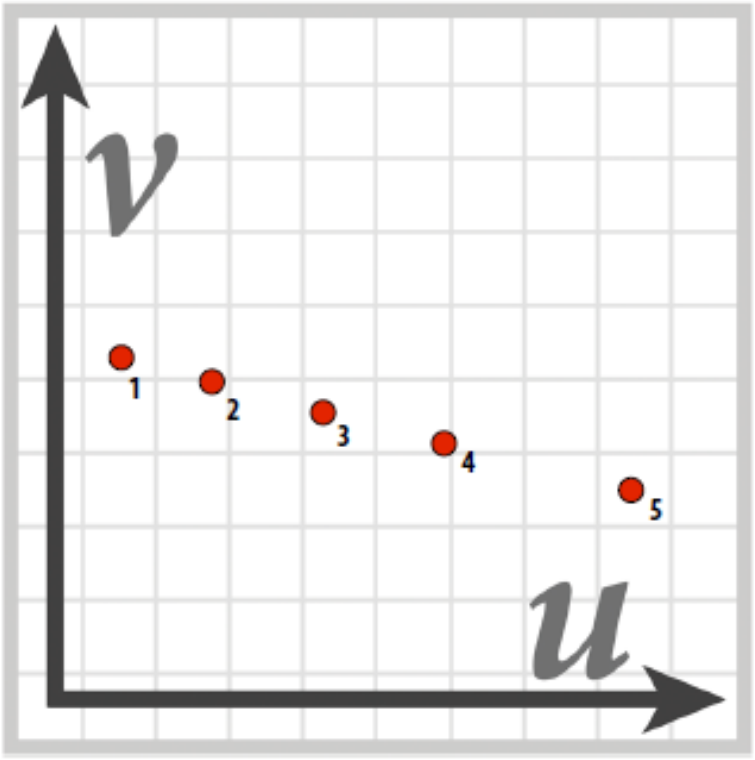
Texture Mapping

Sample positions in XY screen space



Sample positions are uniformly distributed in screen space (rasterizer samples triangle's appearance at these locations)

Sample positions in texture space



Texture sample positions in texture space (texture function is sampled at these locations)

Linear Texture Mapping

- Do a direct mapping of a block of texture to a surface patch:

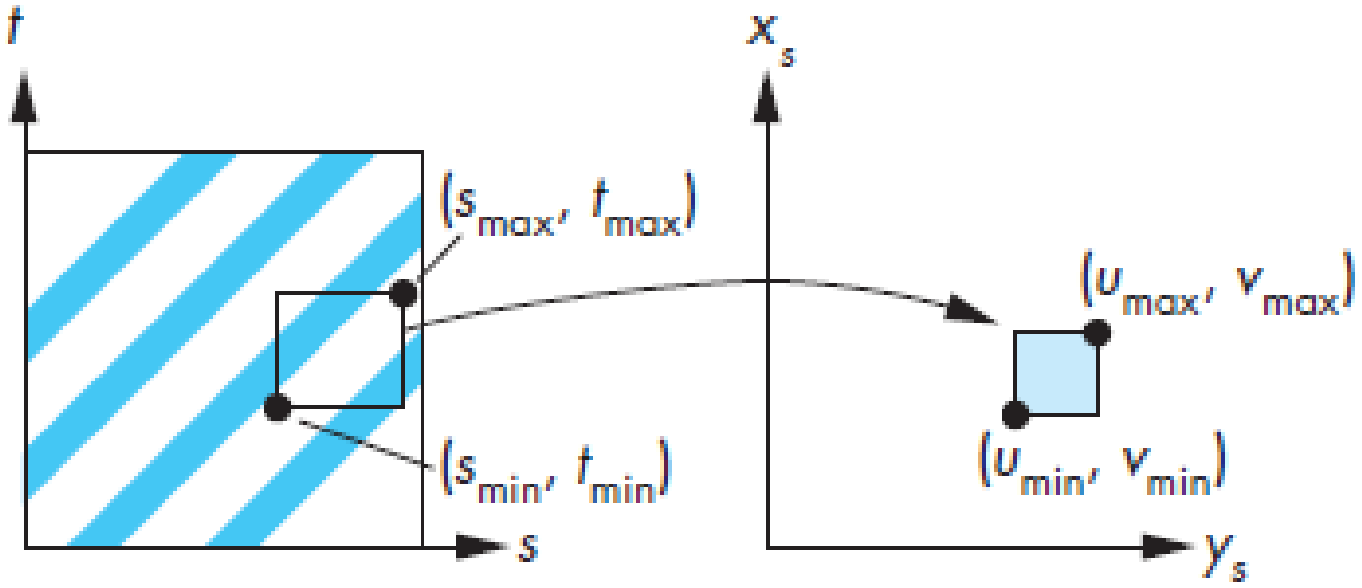


FIGURE 7.12 Linear texture mapping.

Linear Texture Mapping

- A point **p** on the surface is a function of two parameters *u* and *v*. For each pair of values, we generate the point:
- $p(u,v) = \begin{bmatrix} x\,(u,v) \\ y\,(u,v) \\ z\,(u,v) \end{bmatrix}$
- In Figure 7.12, the patch determined by the corners *(s*min, *t*min*)* and *(s*max, *t*max*)* corresponds to the surface patch with corners *(u*min, *v*min*)* and *(u*max, *v*max*)*, then the mapping is
- $u = u_{min} + \frac{s - s_{min}}{s_{max} - s_{min}}\,(u_{max} - u_{min}),$
- $v = v_{min} + \frac{t - t_{min}}{t_{max} - t_{min}}\,(v_{max} - v_{min})$

Cube Mapping

- “Unwrap” cube and map texture over the cube.

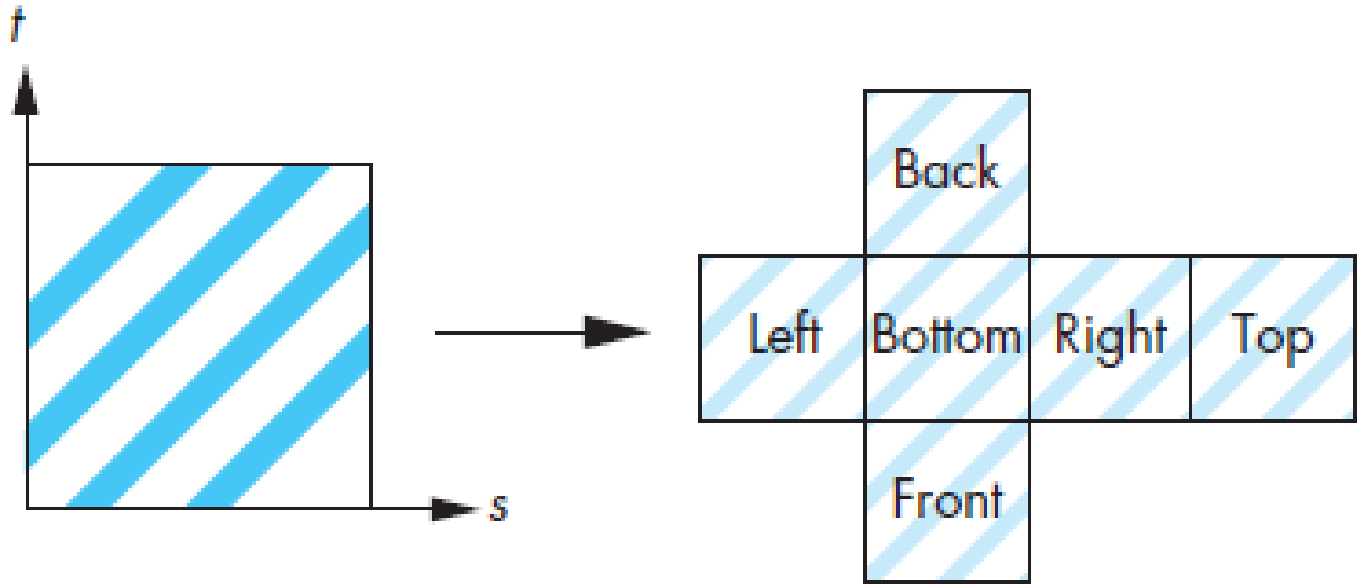


FIGURE 7.14 Texture mapping with a box.

Cylinder Mapping

- Wrap texture along outside of cylinder, not top and bottom
- This stops texture from being distorted

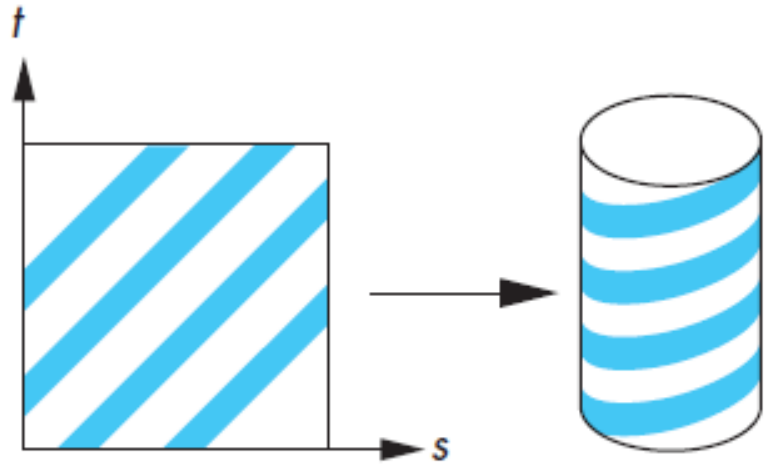


FIGURE 7.13 Texture mapping with a cylinder.

Cylinder Mapping

- in Figure 7.13. Points on the cylinder are given by the parametric equations
- $x = r\cos(2\pi u)$,
- $y = r\sin(2\pi u)$,
- $z = v/h$,
- as u and v vary over (0,1). Hence, we can use the mapping
- $s = u$,
- $t = v$.

Two-part Mapping

- To simplify the problem of mapping from an image to an arbitrary model, use an object we already have a map for as an intermediary!
- Texture -> Intermediate object -> Final model
- Common intermediate objects:
- Cylinder
- Cube
- Sphere

Intermediate Object to Model

- This step can be done in many ways:
- Normal from intermediate surface
- Normal from object surface
- Use center of object

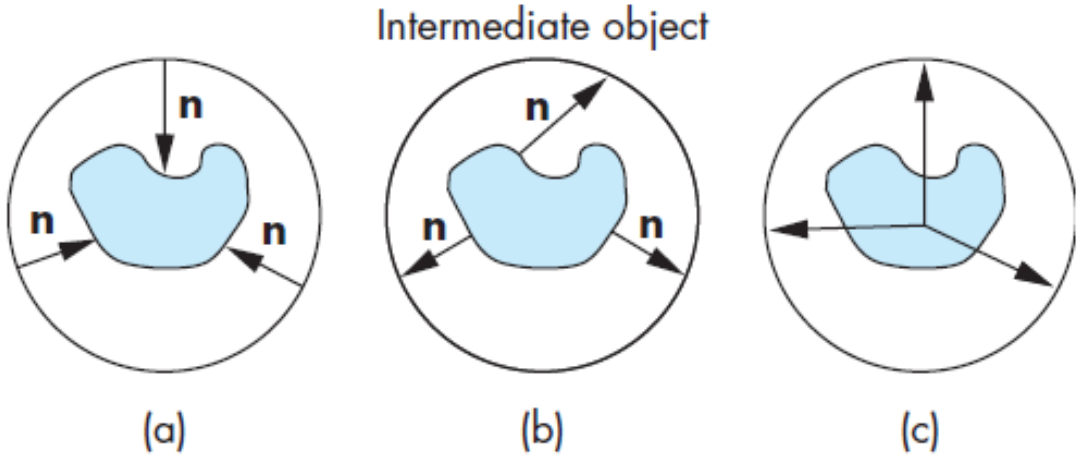


FIGURE 7.15 Second mapping. (a) Using the normal from the intermediate surface. (b) Using the normal from the object surface. (c) Using the center of the object.

Difficulties in Texture Mapping

- First, we must determine the map from texture coordinates to object coordinates. A two-dimensional texture usually is defined over a rectangular region in texture space. The mapping from this rectangle to an arbitrary region in three-dimensional space may be a complex function or may have undesirable properties. For example, if we wish to map a rectangle to a sphere, we cannot do so without distortion of shapes and distances.
- Second, owing to the nature of the rendering process, which works on a pixel-by-pixel basis, we are more interested in the inverse map from screen coordinates to texture coordinates.

Difficulties in Texture Mapping

- Third, because each pixel corresponds to a small rectangle on the display, we are interested in mapping not points to points, but rather areas to areas. Here again is a potential aliasing problem that we must treat carefully if we are to avoid artifacts, such as wavy sinusoidal or moire’ patterns.

What is aliasing?

- An on-screen pixel does not always map neatly to a texel. Particularly severe problems in regular textures.

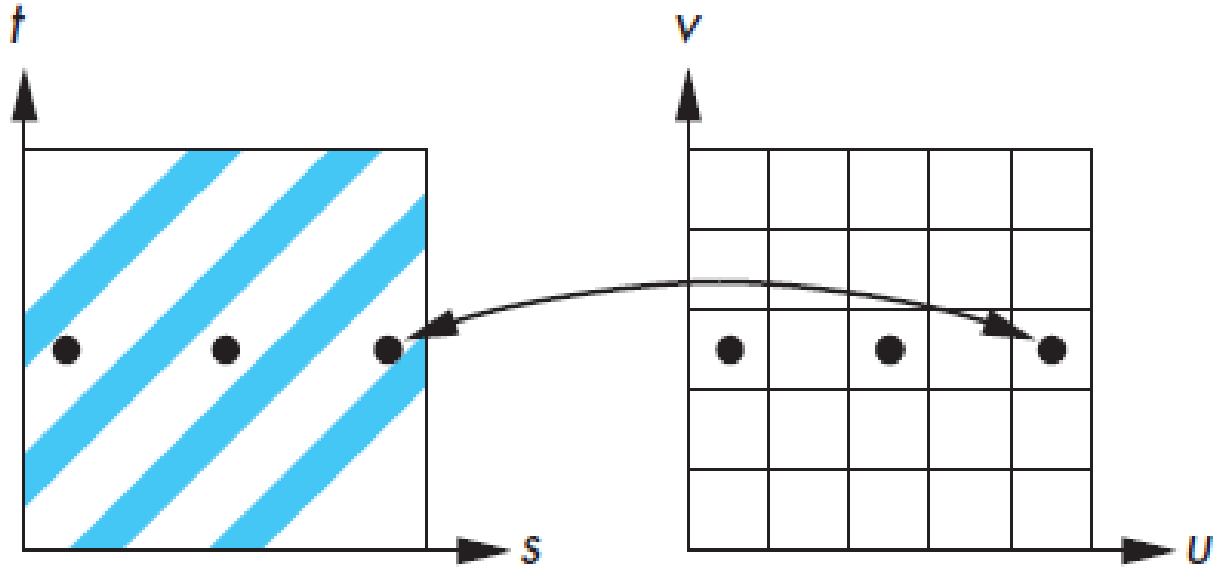
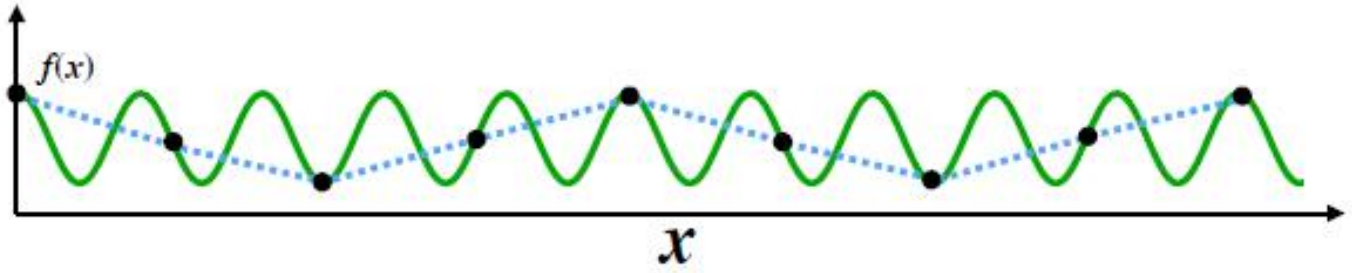


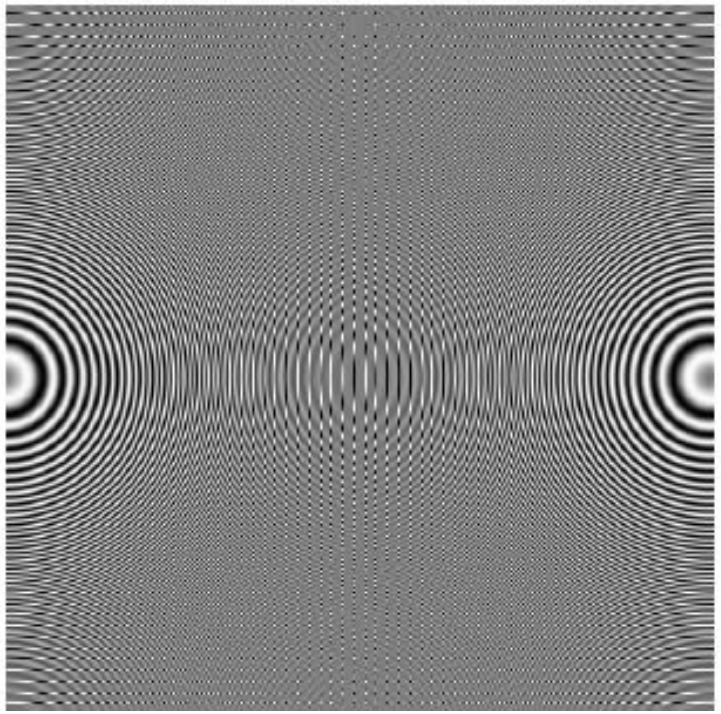
FIGURE 7.11 Aliasing in texture generation.

Aliasing

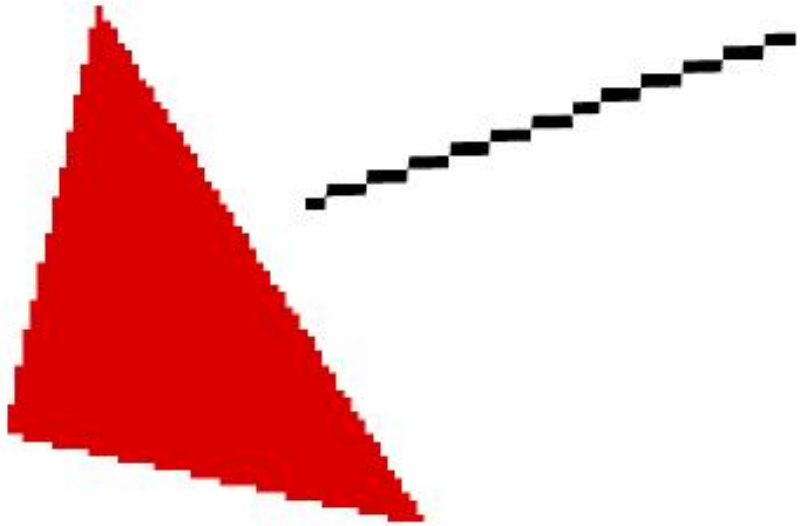
Undersampling a high-frequency signal can result in aliasing



1D example



2D examples:
Moiré patterns, jaggies

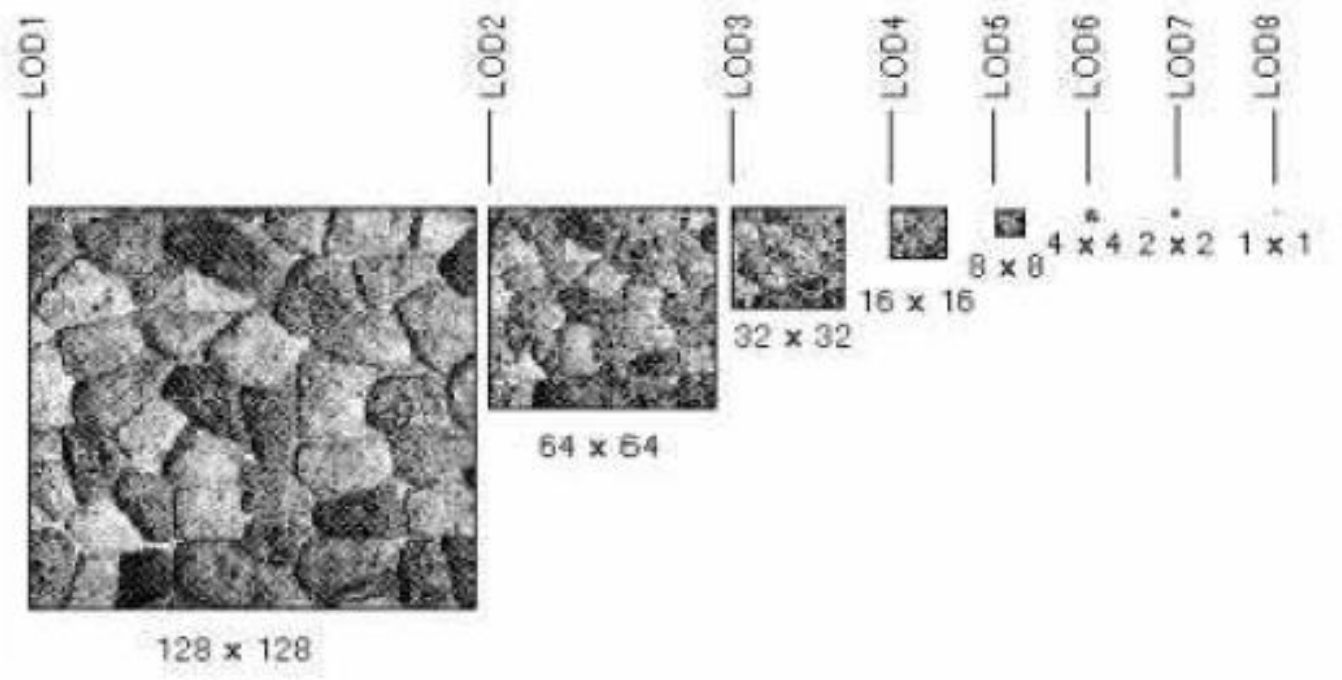


Moire Pattern

Caused by camera pixel frequency being higher than that of the grid pattern on the big central door (minification).

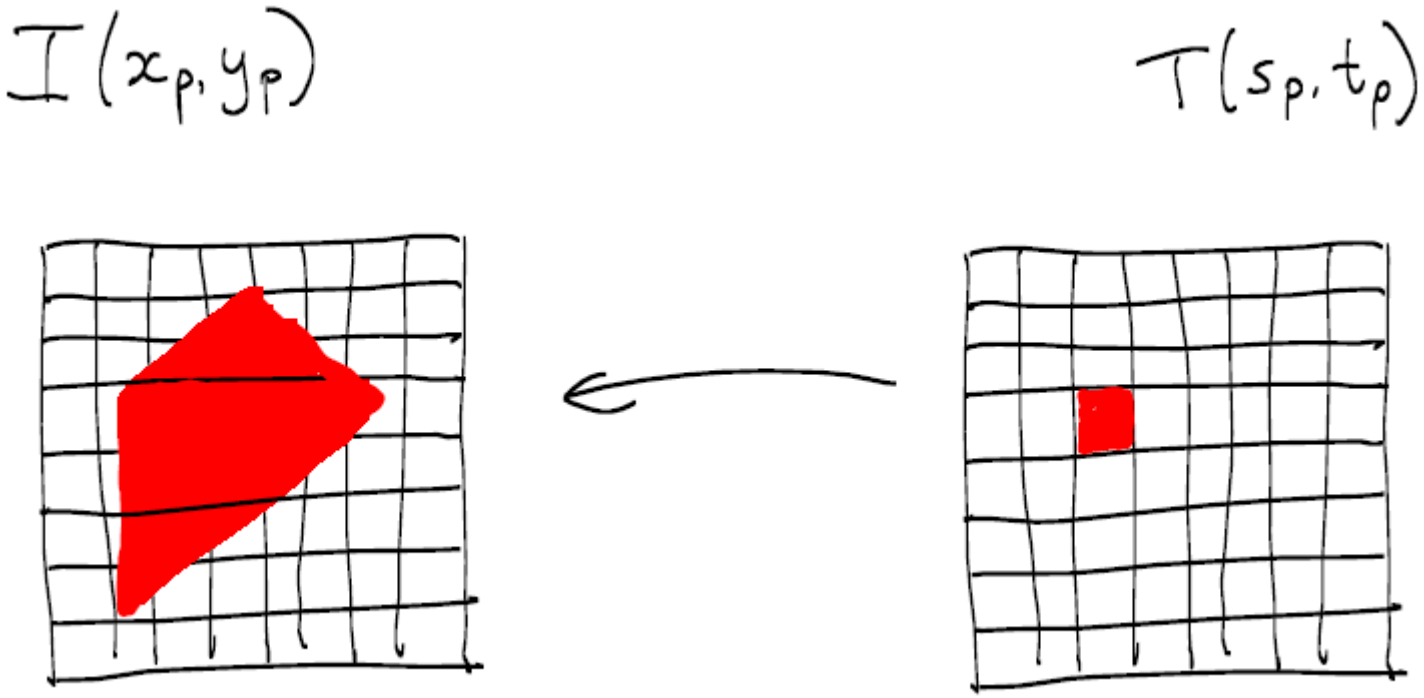
Anti-Aliasing

- Pre-calculate how the texture should look at various distances, then use the appropriate texture at each distance. This is called *mipmapping*.



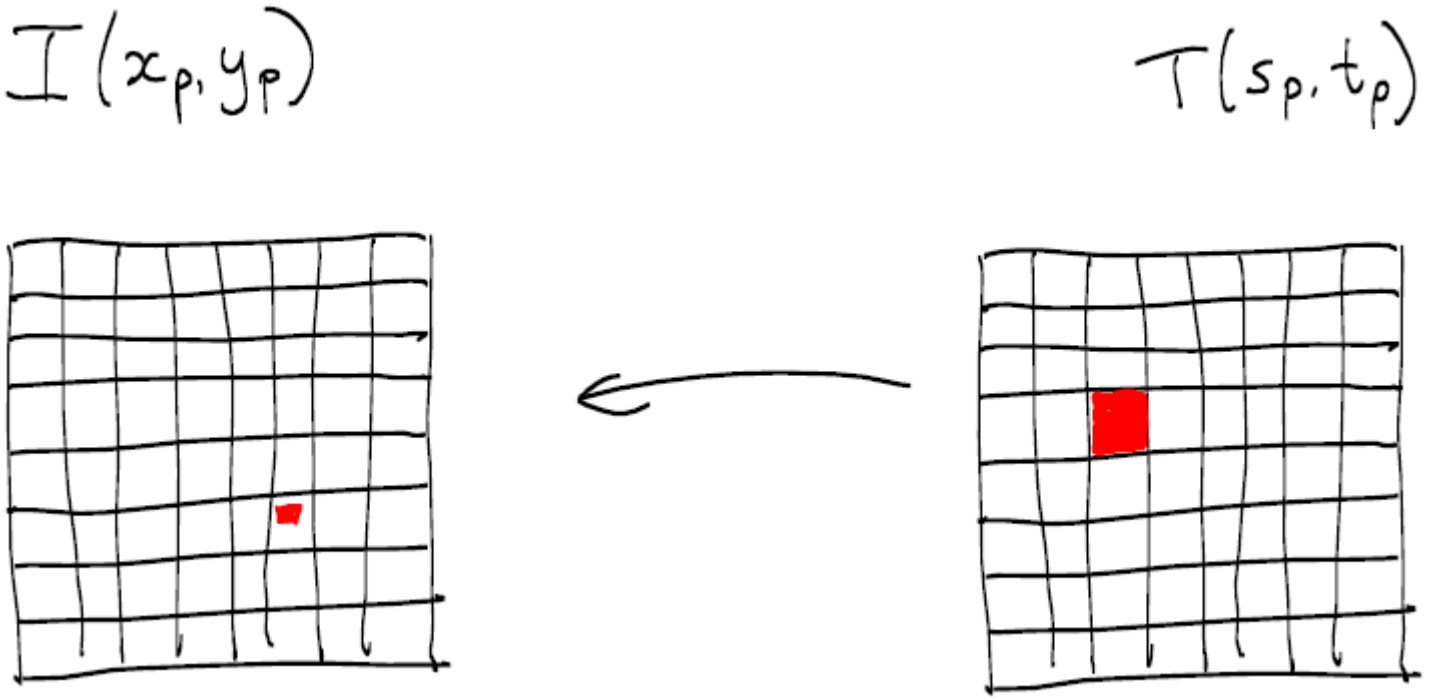
Texture magnification

- a pixel in texture image ('texel') maps to an area larger than one pixel in image



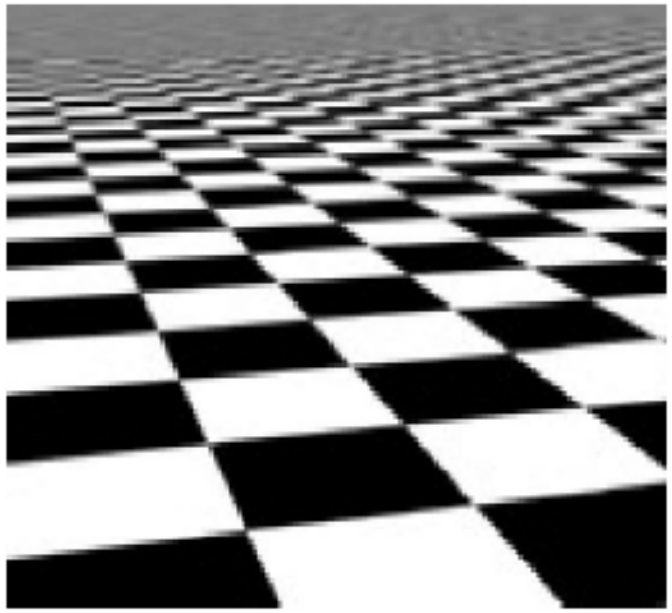
Texture minification

- a pixel in texture image('texel') maps to an area smaller than a pixel in image:



Mipmapping

$I(x_p, y_p)$



e.g. 300 x 300

minification

magnification

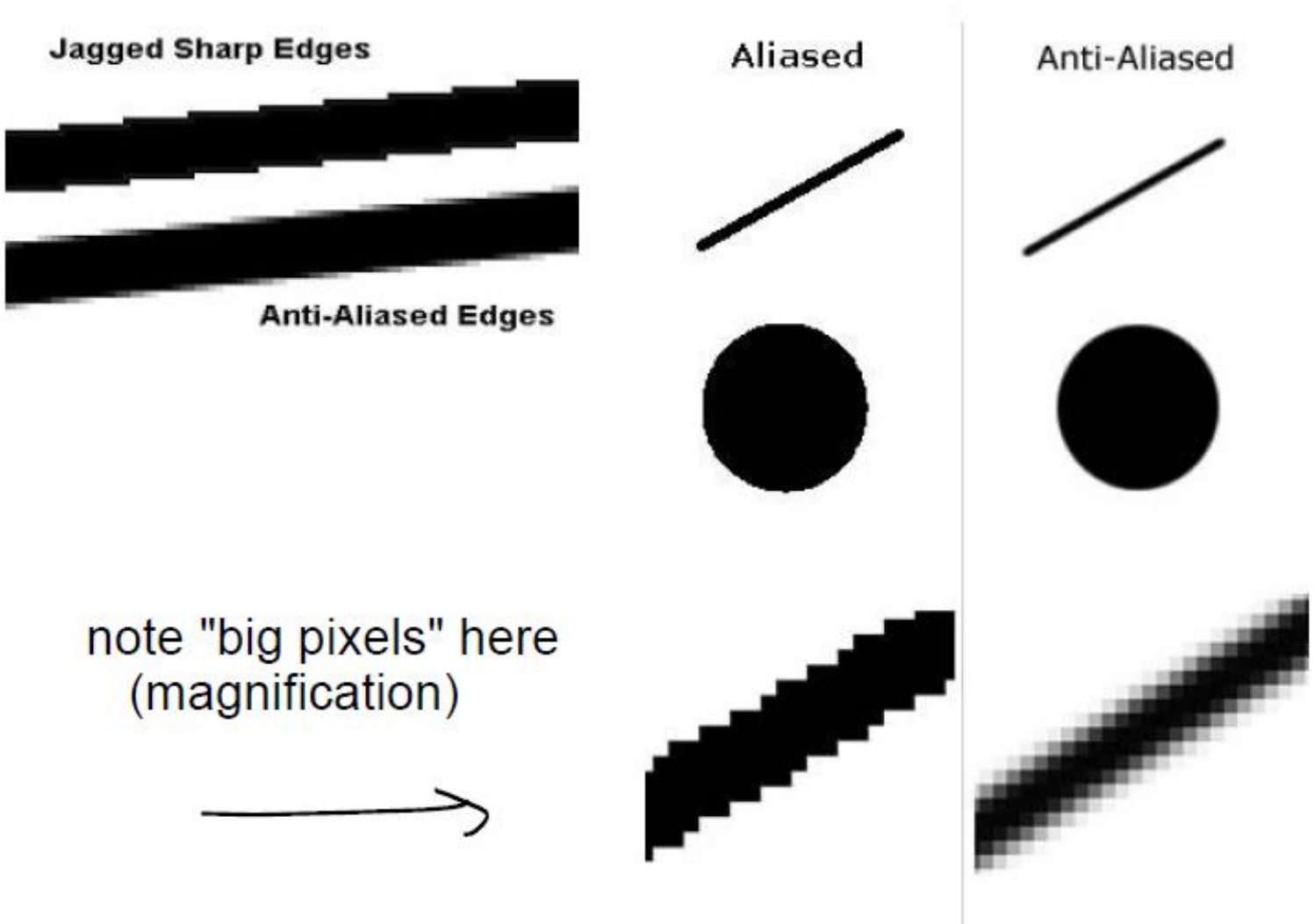
$T(s_p, t_p)$

e.g. 8 x 8

Anti-Aliasing

- Another approach: Filtering-
  1. Bi Linear Filtering
  2. Tri Linear Filtering
  3. Anisotropic Filtering

Aliasing and Anti-aliasing



OpenGL Texture Mapping

- OpenGL's texture maps rely on its pipeline architecture. We have seen that there are actually two parallel pipelines: the geometric pipeline and the pixel pipeline. For texture mapping, the pixel pipeline merges with fragment processing after rasterization, as shown in Figure 7.16. This architecture determines the type of texture mapping that is supported. In particular, texture mapping is done as part of fragment processing. Each fragment that is generated is then tested for visibility with the *z*-buffer. We can think of texture mapping as a part of the shading process, but a part that is done on a fragment-by-fragment basis.

Texture Mapping Pipeline

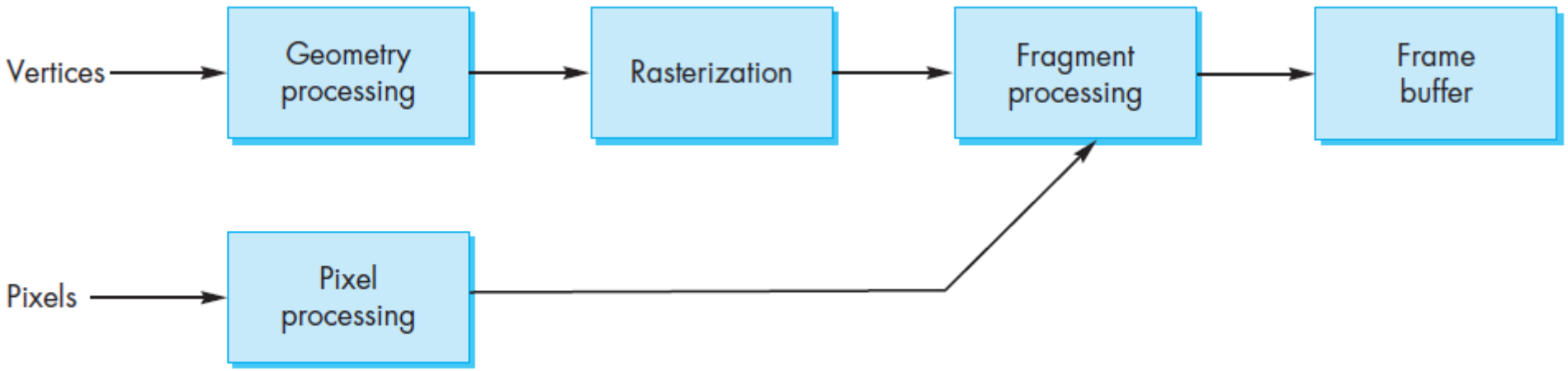


FIGURE 7.16 Pixel and geometry pipelines.

OpenGL Texture Mapping

- Texture mapping requires interaction among the application program, the vertex shader, and the fragment shader. There are three basic steps. First, we must form a texture image and place it in texture memory on the GPU. Second, we must assign texture coordinates to each fragment. Finally, we must apply the texture to each fragment.

glBindTexture

- glBindTexture(GL_TEXTURE_2D,textureName);
- GL_TEXTURE_2D: Specify that it is a 2D texture
- textureName: Name of the texture

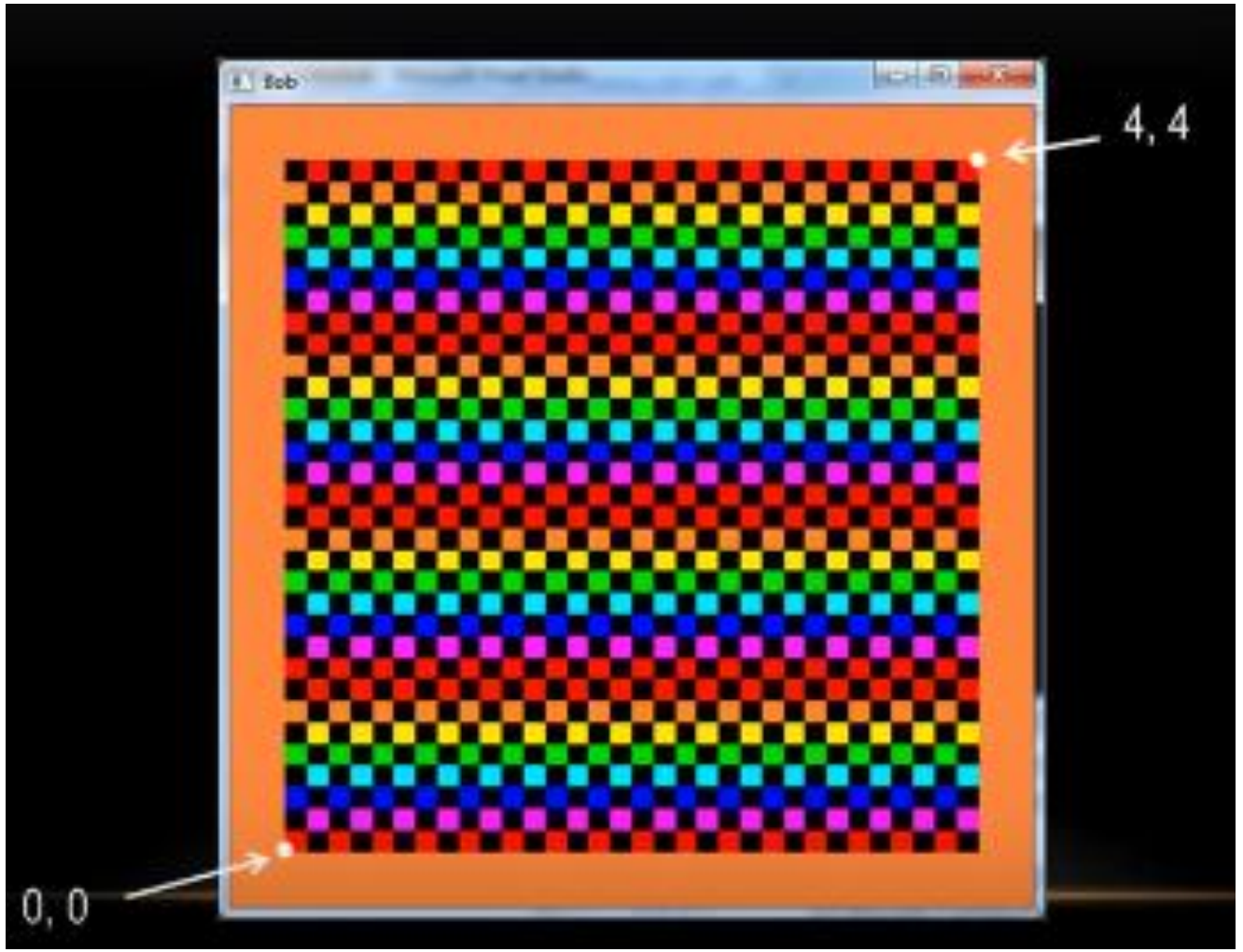
glTexImage2D

- glTexImage2D(GL_TEXTURE_2D, level, components, width, height, border, format, type, tarray)
- GL_TEXTURE_2D: Specify that it is a 2D texture
- Level: Used for specifying levels of detail for mipmapping
- Components: Generally is 0 which means GL_RGB, Represents components and resolution of components
- Width, Height: The size of the texture must be powers of 2
- Border Format: Specify what the data is (GL_RGB, GL_RGBA, …)
- Type: Specify data type (GL_UNSIGNED_BYTE, GL_BYTE, …)

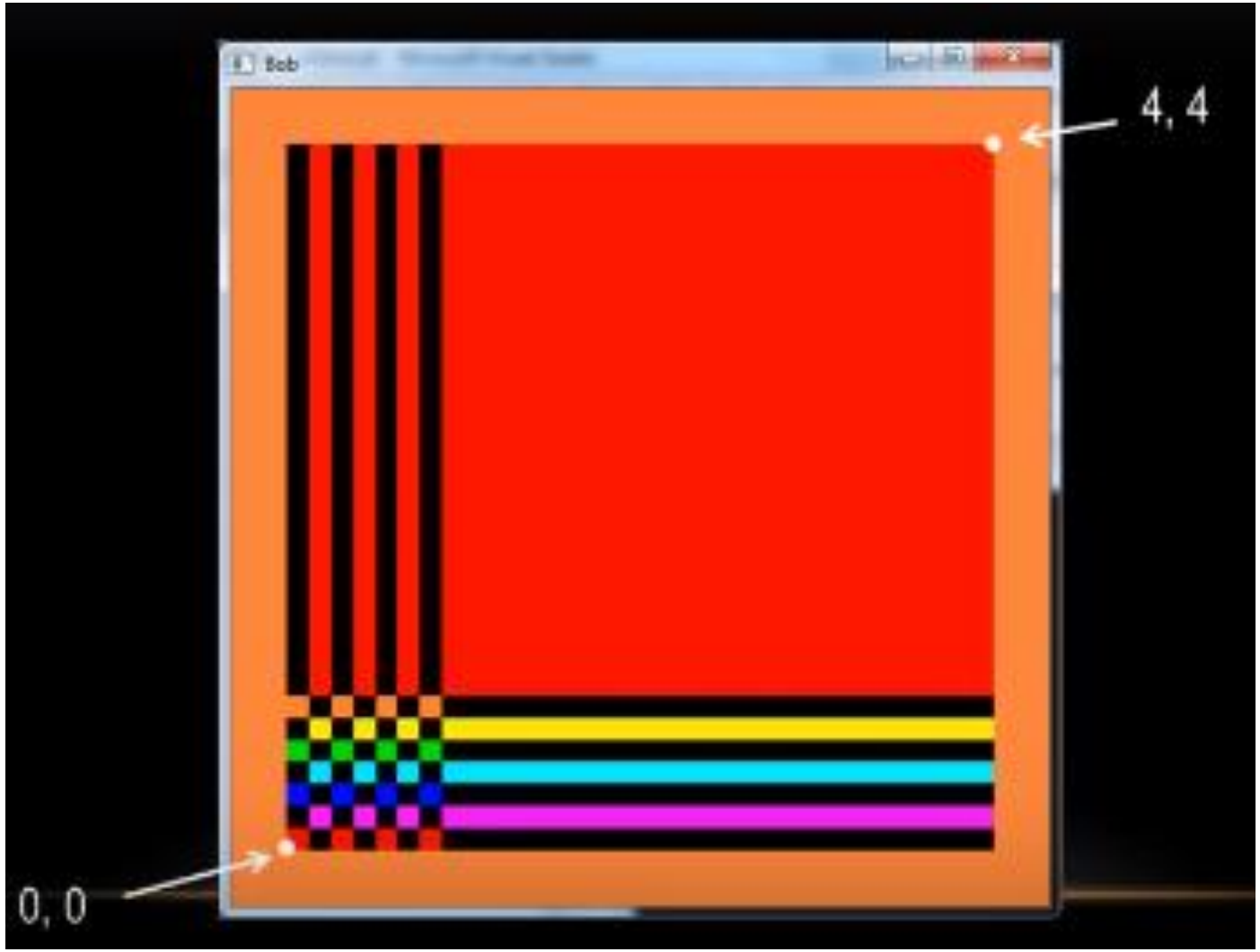
glTexParameteri

- glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
- This function sets several texture mapping parameters. These parameters are bound to the current texture state that can be made current with glBindTexture.
- parameters:
- P1: GLenum: The texture target for which this parameter applies. Must be one of GL_TEXTURE_1D, GL_TEXTURE_2D, GL_TEXTURE_3D, or GL_TEXTURE_CUBE_MAP.
- P2: GLenum: The texturing parameter to set. GL_TEXTURE_MAG_FILTER Returns the texture magnification filter value
- P3: GLfloat or GLfloat* or GLint or GLint*: Value of the parameter specified by pname.

GL_REPEAT Instead of GL_LINEAR



GL_CLAMP



GL_NEAREST

GL_LINEAR

glTexGen

- void glTexGeni(GLenum coord, GLenum pname, GLint param);
- glTexGen selects a texture-coordinate generation function or supplies coefficients for one of the functions. coord names one of the (s, t, r, q) texture coordinates; it must be one of the symbols GL_S, GL_T, GL_R, or GL_Q.
- Coord: Specifies a texture coordinate. Must be one of GL_S, GL_T, GL_R, or GL_Q.
- Pname: Specifies the symbolic name of the texture-coordinate generation function or function parameters. Must be GL_TEXTURE_GEN_MODE, GL_OBJECT_PLANE, or GL_EYE_PLANE.

glTexGen

- Params: Specifies a pointer to an array of texture generation parameters. If pname is GL_TEXTURE_GEN_MODE, then the array must contain a single symbolic constant, one of GL_OBJECT_LINEAR, GL_EYE_LINEAR, GL_SPHERE_MAP, GL_NORMAL_MAP, or GL_REFLECTION_MAP. Otherwise, params holds the coefficients for the texture-coordinate generation function specified by pname.