COSC 414/519I: Computer Graphics

2023W2

Shan Du

Generated Texture Image

- Two-dimensional texture mapping starts with an array of texels, which is a two-dimensional pixel rectangle. Suppose that we have a 64×64 RGBA image `myTexels` that was generated by the code

```
var texSize = 64;
var numRows = 8;
var numCols = 8;
var numComponents = 4; // RGBA texels

var myTexels = new Uint8Array(numComponents*texSize*texSize);

for (var i = 0; i < texSize; ++i) {
  var patchx = Math.floor(i/(texSize/numRows));

  for (var j = 0; j < texSize; ++j) {
    var patchy = Math.floor(j/(texSize/numCols));

    var c = (patchx%2 != patchy%2 ? 255 : 0);

    var texel = numComponents*(i*texSize + j);

    myTexels[texel+0] = c;
    myTexels[texel+1] = c;
    myTexels[texel+2] = c;
    myTexels[texel+3] = 255;
  }
}
```

Generated Texture Image

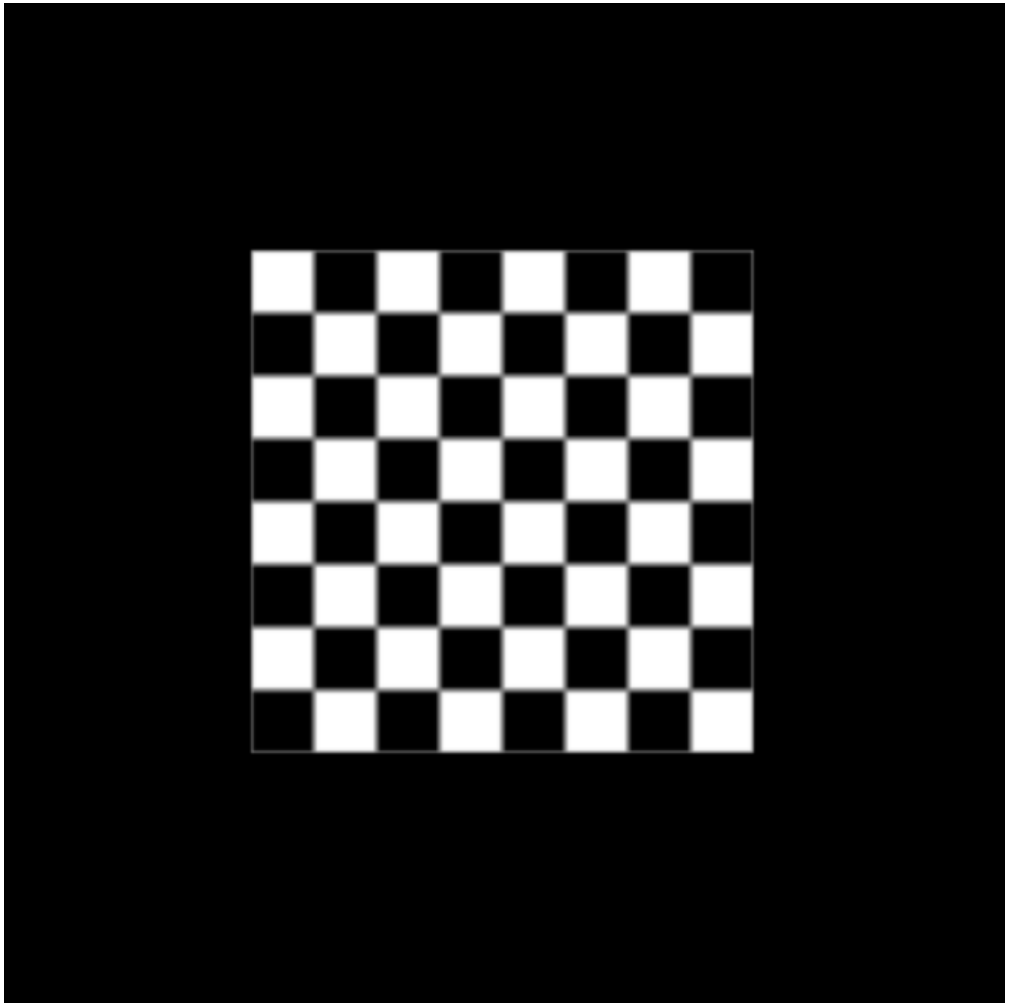
- Thus, the colors in `myTexels` form an 8×8 black-and-white checkerboard. We specify that this array is to be used as a two-dimensional texture after the call to `gl.bindTexture` by

```
gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGBA, texSize, texSize, 0, gl.RGBA,
              gl.UNSIGNED_BYTE, myTexels);
```

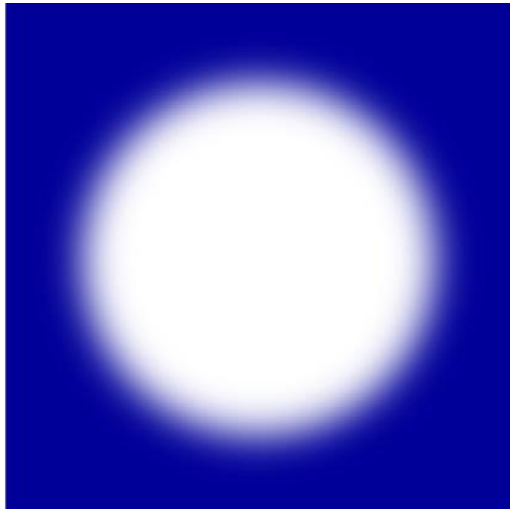
- More generally, two-dimensional textures are specified through the function

```
gl.texImage2D(target, level, iformat,
              width, height, border, format, type, texelArray)
```

Generated Texture Image



Pasting Multiple Textures to a Shape



Sample Program (MultiTexture.js)

- Three key differences:

(1) the fragment shader accesses two textures,

(2) the final fragment color is calculated from the two texels from both textures,

and (3) initTextures() creates two texture objects.

Sample Program (MultiTexture.js)

```
13 var FSHADER_SOURCE =
     ...
 17   'uniform sampler2D u_Sampler0;\n' +
 18   'uniform sampler2D u_Sampler1;\n' +
 19   'varying vec2 v_TexCoord;\n' +
 20   'void main() {\n' +
 21   '  vec4 color0 = texture2D(u_Sampler0, v_TexCoord);\n' +          <-(1)
 22   '  vec4 color1 = texture2D(u_Sampler1, v_TexCoord);\n' +
 23   '  gl_FragColor = color0 * color1;\n' +                          <-(2)
 24   '}\n';

103 function initTextures(gl, n) {
104   // Create a texture object
105   var texture0 = gl.createTexture();                               <-(3)
106   var texture1 = gl.createTexture();
    ...
112   // Get the storage locations of u_Sampler1 and u_Sampler2
113   var u_Sampler0 = gl.getUniformLocation(gl.program, 'u_Sampler0');
114   var u_Sampler1 = gl.getUniformLocation(gl.program, 'u_Sampler1');
    ...
```

Sample Program (MultiTexture.js)

```
121   var image0 = new Image();
122   var image1 = new Image();
    ...
127   // Register the event handler to be called when image loading is completed
128   image0.onload = function(){ loadTexture(gl, n, texture0, u_Sampler0,
                                                          ➥image0, 0); };
129   image1.onload = function(){ loadTexture(gl, n, texture1, u_Sampler1,
                                                          ➥image1, 1); };
130   // Tell the browser to load an Image
131   image0.src = '../resources/redflower.jpg';
132   image1.src = '../resources/circle.gif';
```

Sample Program (MultiTexture.js)

```
136 // Specify whether the texture unit is ready to use
137 var g_texUnit0 = false, g_texUnit1 = false;
138 function loadTexture(gl, n, texture, u_Sampler, image, texUnit) {
139   gl.pixelStorei(gl.UNPACK_FLIP_Y_WEBGL, 1);// Flip the image's y-axis
140   // Make the texture unit active
141   if (texUnit == 0) {
142     gl.activeTexture(gl.TEXTURE0);
143     g_texUnit0 = true;
144   } else {
145     gl.activeTexture(gl.TEXTURE1);
146     g_texUnit1 = true;
147   }
148   // Bind the texture object to the target
149   gl.bindTexture(gl.TEXTURE_2D, texture);
```

Sample Program (MultiTexture.js)

```
151   // Set texture parameters
152   gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.LINEAR);
153   // Set the texture image
154   gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGBA, gl.RGBA, gl.UNSIGNED_BYTE, image);
155   // Set the texture unit number to the sampler
156   gl.uniform1i(u_Sampler, texUnit);
    ...
161   if (g_texUnit0 && g_texUnit1) {
162     gl.drawArrays(gl.TRIANGLE_STRIP, 0, n); // Draw a rectangle
163   }
164 }
```

Sample Program (MultiTexture.js)

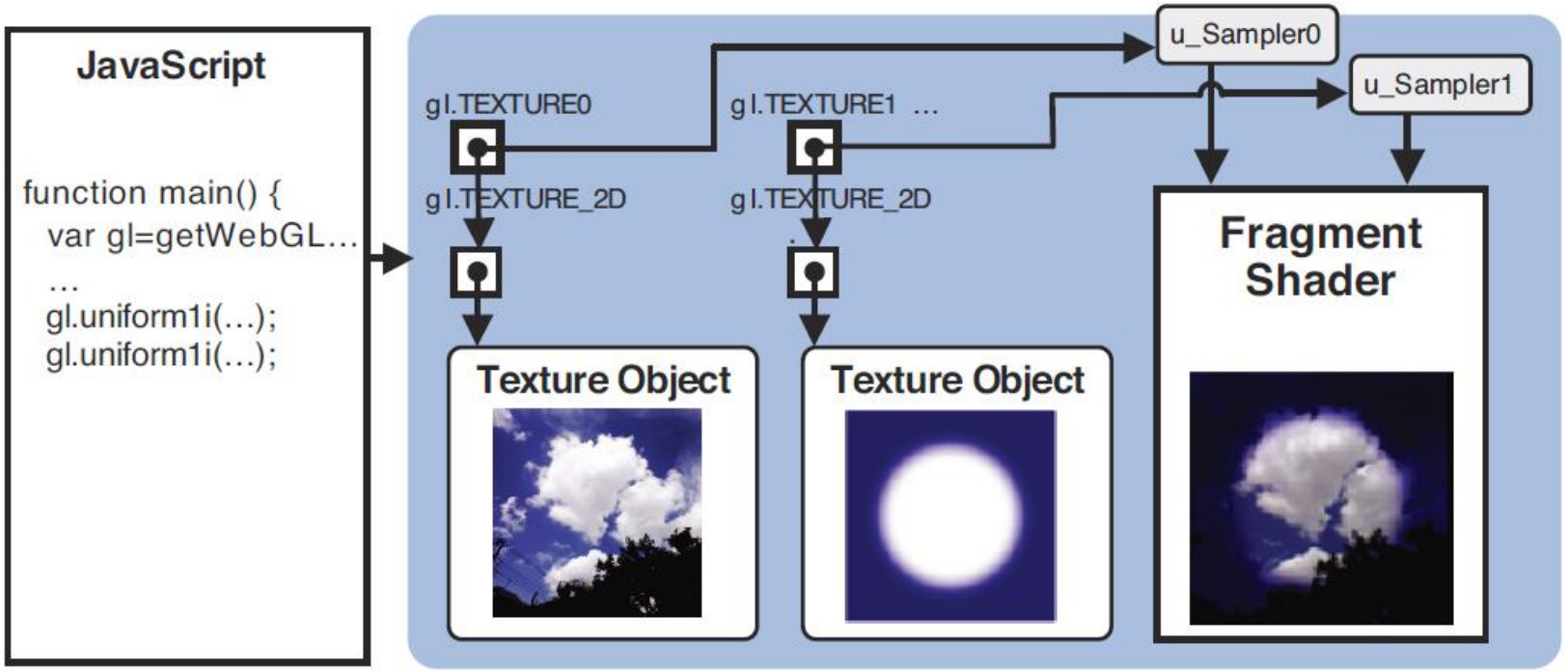


Figure 5.37 Internal state of WebGL when handling two texture images

Texture Coordinates and Samplers

- It is up to the application and the shaders to determine the appropriate texture coordinates for a fragment. The most common method is to treat texture coordinates as a vertex attribute. Thus, we could provide texture coordinates just as we provide vertex colors in the application. We then would pass these coordinates to the vertex shader and let the rasterizer interpolate the vertex texture coordinates to fragment texture coordinates.

Texture Coordinates and Samplers

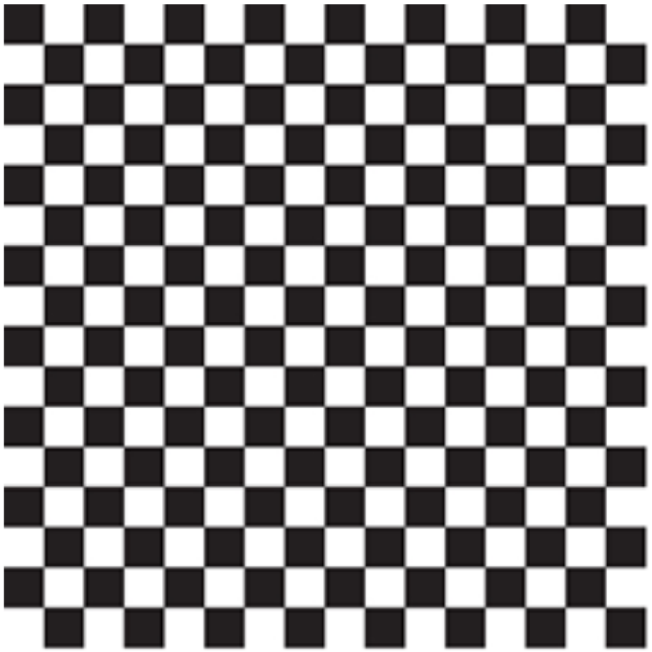
- Note that the vertex shader is only concerned with the texture coordinates and has nothing to do with the texture object we created earlier. We should not be surprised because the texture itself is not needed until we are ready to assign a color to a fragment, which occurs in the fragment shader. Note also that many of the parameters that determine how we can apply the texture are inside the texture object and thus will allow us to use very simple fragment shaders.

Texture Coordinates and Samplers

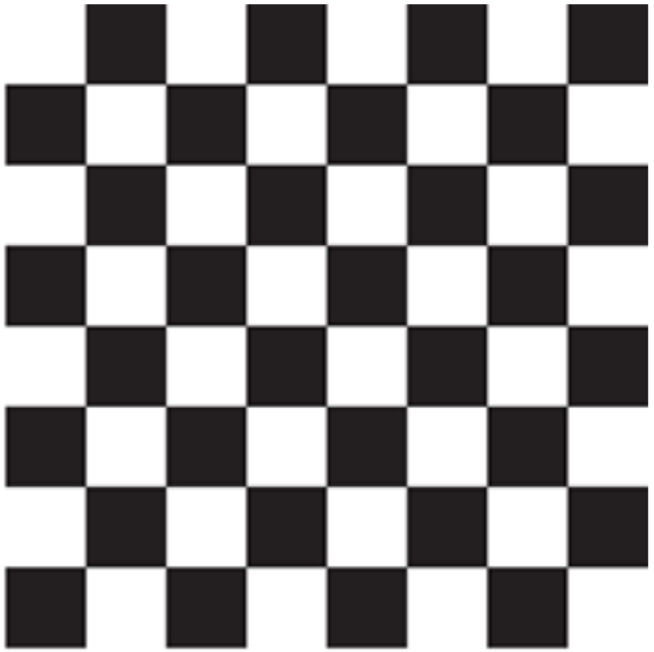
- The key to putting everything together is a new type of variable called a **sampler**, which we usually use only in a fragment shader. A sampler variable provides access to a texture object, including all its parameters. What a sampler does is return a value or sample of the texture image for the input texture coordinates. How this value is determined depends on parameters associated with the texture object.

Texture Coordinates and Samplers

Figure 7.14 Mapping of a checkerboard texture to a quadrilateral. (a) Using the entire texel array. (b) Using part of the texel array.



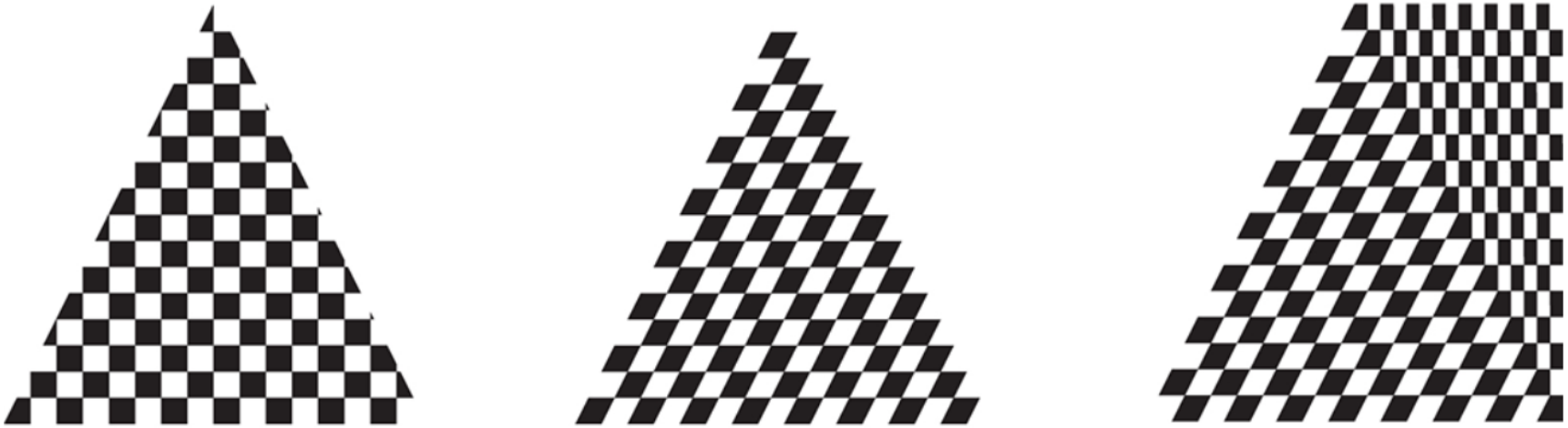
(a)



(b)

Texture Coordinates and Samplers

Figure 7.15 Mapping of texture to polygons. (a and b) Mapping of a checkerboard texture to a triangle. (c) Mapping of a checkerboard texture to a trapezoid.

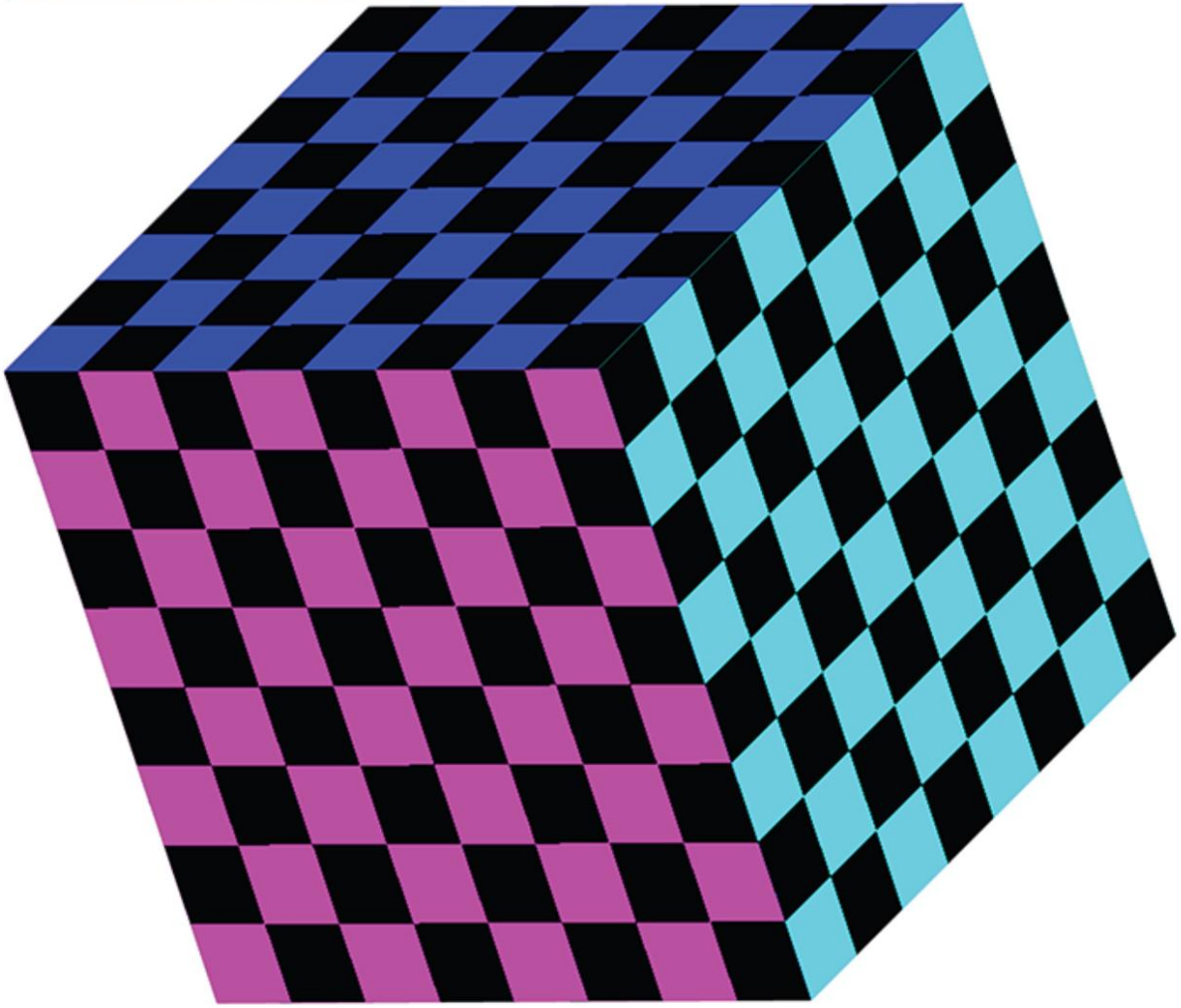


Texture Coordinates and Samplers

Figure 7.16 Color Cube with a texture image on each face.
(http://www.interactivecomputergraphics.com/Code/07/textureCube1.html)

Texture Coordinates and Samplers

Figure 7.17 Color Cube with checkerboard texture image on each face.
(http://www.interactivecomputergraphics.com/Code/07/textureCube2.html)



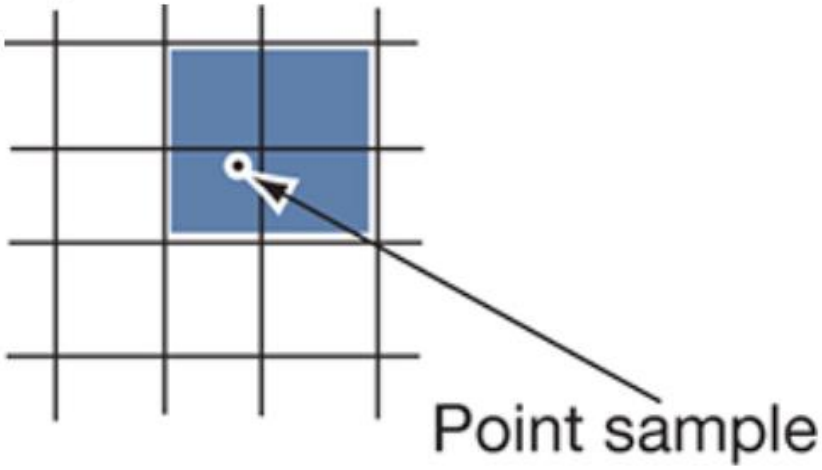
Texture Sampling

- Aliasing of textures is a major problem. When we map texture coordinates to the array of texels, we rarely get a point that corresponds to the center of a texel. One option is to use the value of the texel that is closest to the texture coordinate output by the rasterizer. This option, known as **point sampling**, is the one most subject to visible aliasing errors. A better strategy, although one that requires more work, is to use a weighted average of a group of texels in the neighborhood of the texel determined by point sampling. This option is known as **linear filtering**.

Texture Sampling

The location within a texel is given by bilinear interpolation from the texture coordinates at the vertices and the four texels would be used to obtain a smoother value. If we are using linear filtering, there is a problem at the edges of the texel array because we need additional texel values outside the array.

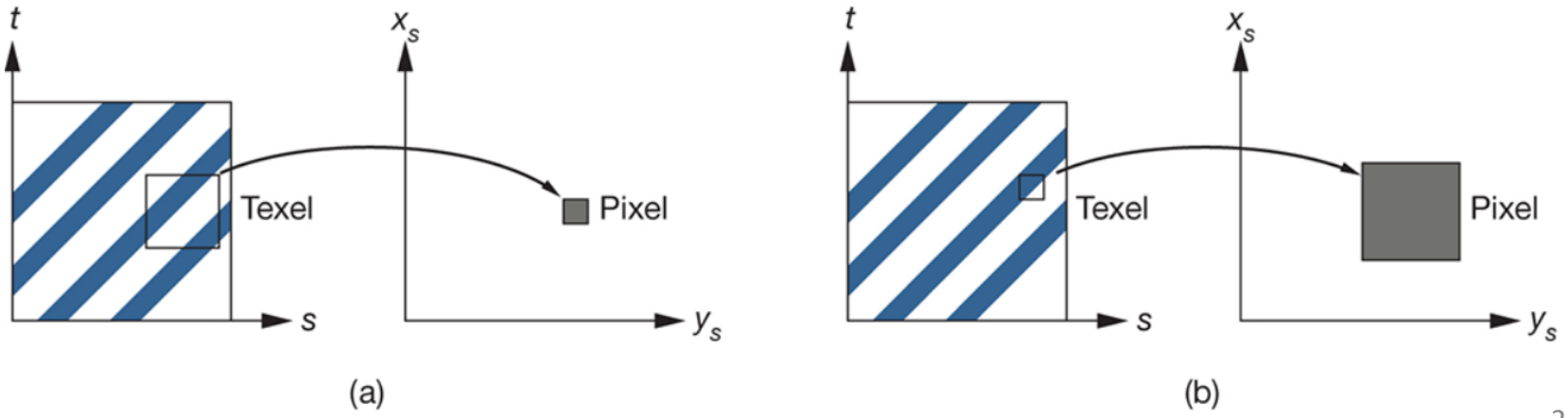
Figure 7.18 Texels used with linear filtering.



Texture Sampling

- There is a further complication, however, in deciding how to use the texel values to obtain a texture value. The size of the pixel that we are trying to color on the screen may be smaller or larger than one texel.

Figure 7.19 Mapping texels to pixels. (a) Minification. (b) Magnification.



Texture Sampling

- In the first case, the texel is larger than one pixel (**minification**); in the second, it is smaller (**magnification**). In both cases, the fastest strategy is to use the value of the nearest point sampling. We can specify this option for both magnification and minification of textures as follows:

```
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER, gl.NEAREST);
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.NEAREST);
```

Texture Sampling

- Alternatively, we can use filtering to obtain a smoother, less aliased image if we specify `gl.LINEAR` instead of `gl.NEAREST`.

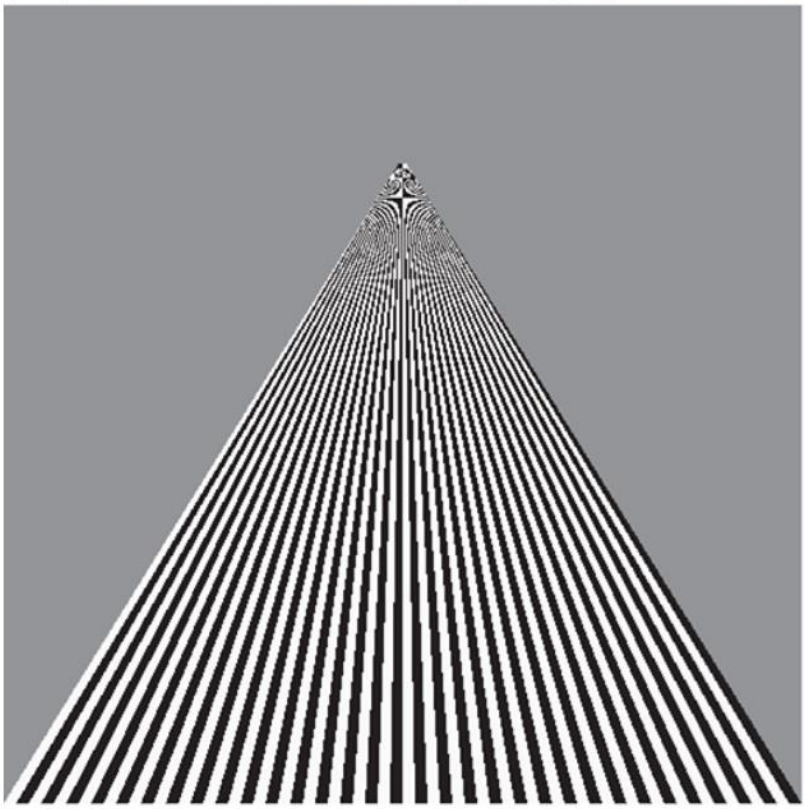
Texture Sampling

- WebGL has another way to deal with the minification problem: **mipmapping**. For objects that project to an area of screen space that is small compared with the size of the texel array, we do not need the resolution of the original texel array. WebGL allows us to create a series of texel arrays at reduced sizes; it will then automatically use the appropriately sized texture in this pyramid of textures, the one for which the size of the texel is approximately the same size of a pixel. For a 64×64 original array, we can set up 32×32,16×16,8×8,4×4,2×2, and 1×1 arrays for the current texture object by executing the function call

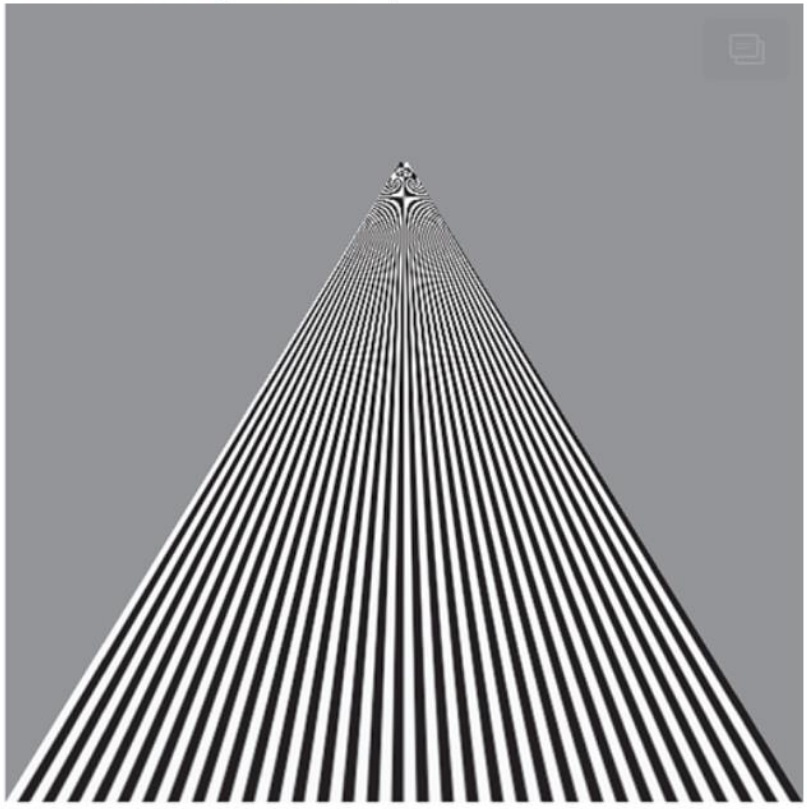
```
gl.generateMipmap(gl.TEXTURE_2D);
```

Texture Sampling

Figure 7.20 Texture mapping to a quadrilateral. (a) Point sampling. (b) Linear filtering. (c) Mipmapping point sampling. (d) Mipmapping linear filtering. (http://www.interactivecomputergraphics.com/Code/07/textureSquare.html)

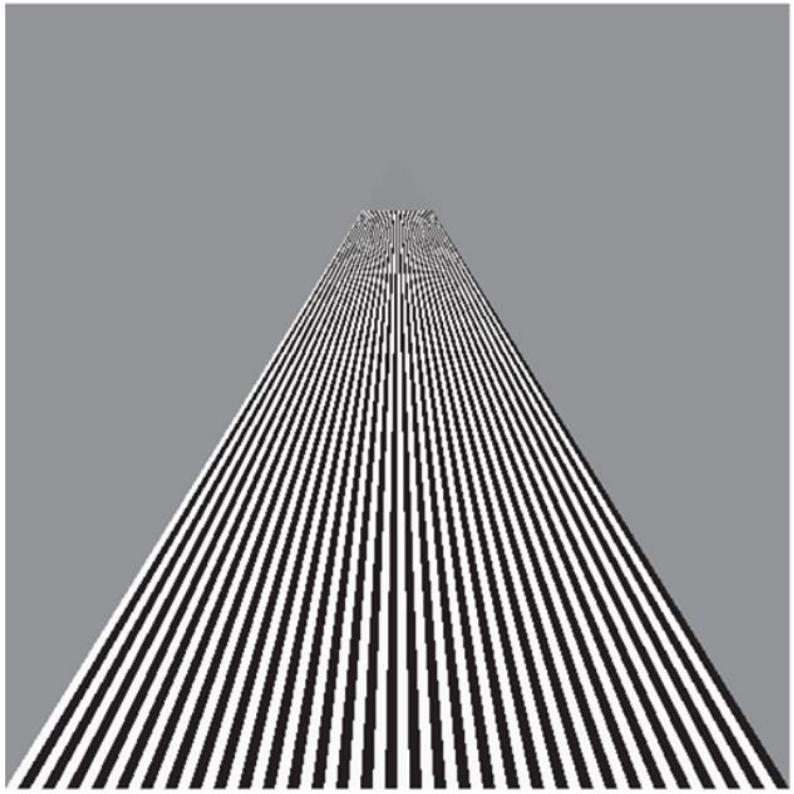


(a)

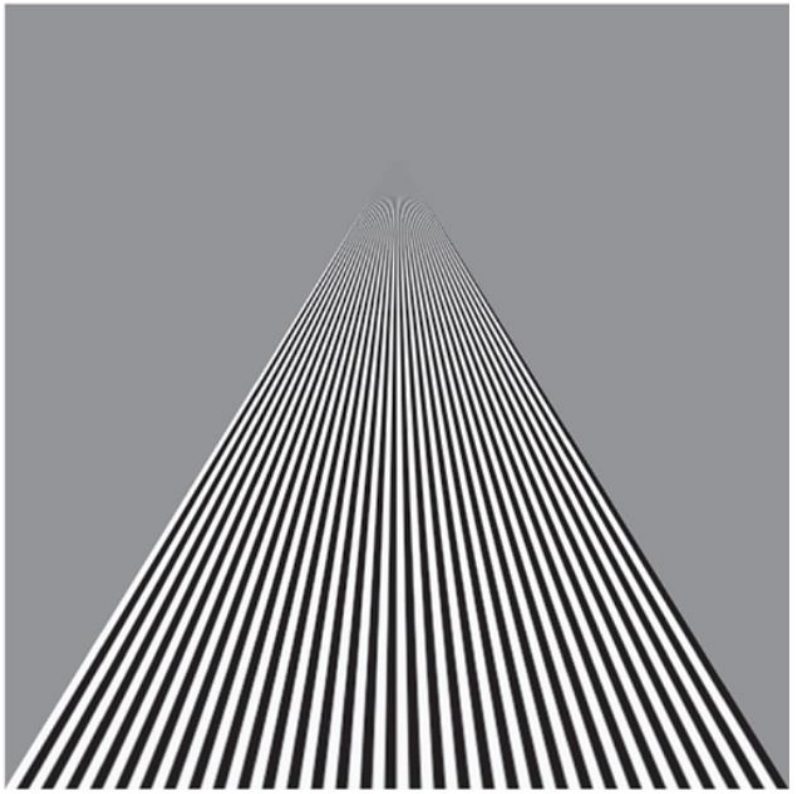


(b)

Texture Sampling



(c)



(d)

Working With Texture Coordinates

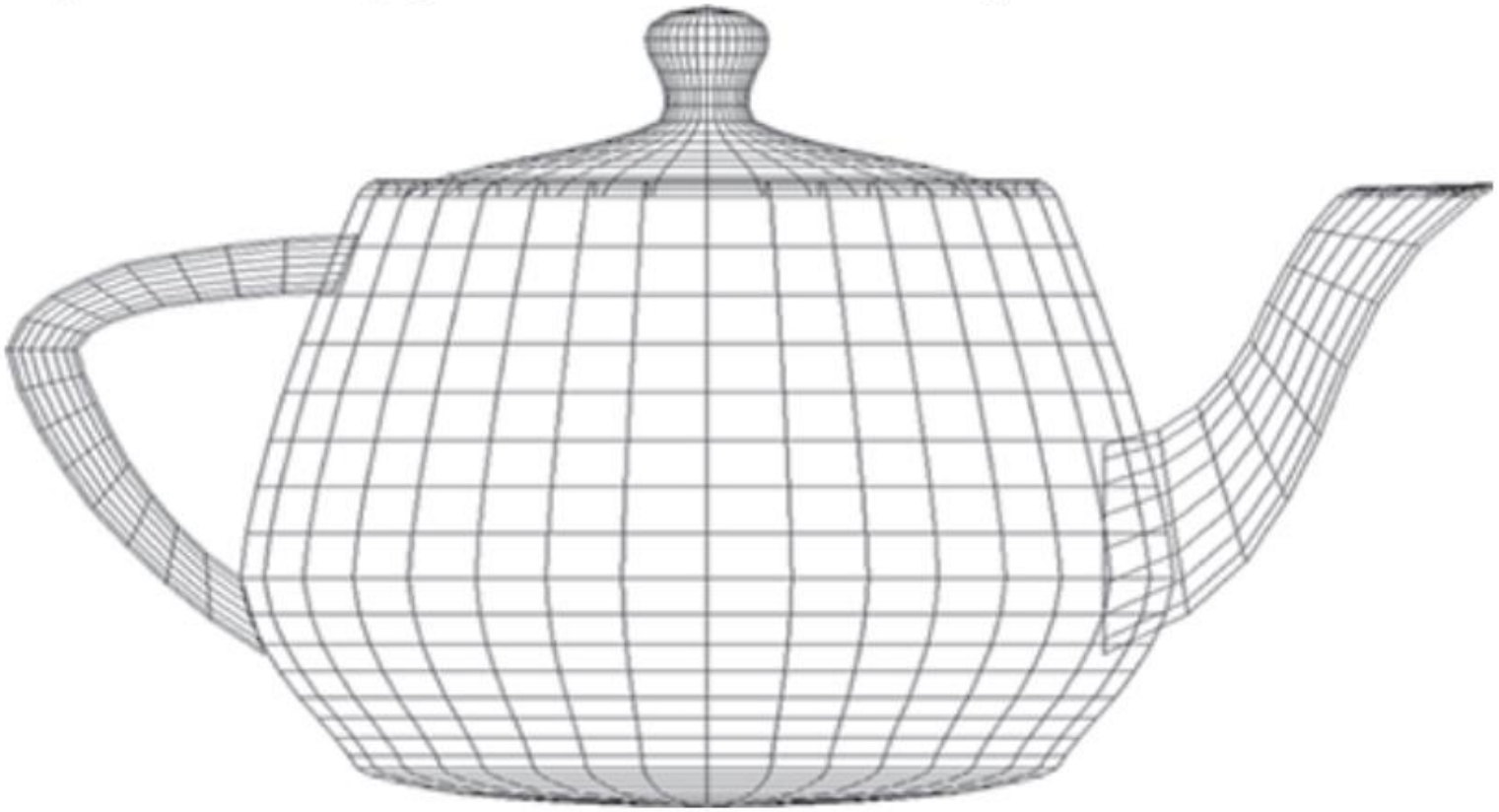
- If we work with rectangular polygons of the same size, then it is fairly easy to assign coordinates. We can also use the fact that texture coordinates can be stored as one-, two-, three-, or four-dimensional arrays, just as vertices are. Thus, texture coordinates can be transformed by matrices and manipulated in the same manner as we transformed positions with the model-view and projection matrices. We can create a texture matrix to scale and orient texture coordinates and to create effects in which the texture moves with the object, the camera, or the lights.

Working With Texture Coordinates

- However, if the set of polygons is an approximation to a curved object, then assigning texture coordinates is far more difficult. Consider the polygonal approximation of the Utah teapot. The model is built from data that describe small surface patches. The patches are of different sizes, with smaller patches in areas of high curvature. When we use the same number of line strips to display each patch, we see different-size quadrilaterals.

Working With Texture Coordinates

Figure 7.21 Polygonal model of Utah teapot.

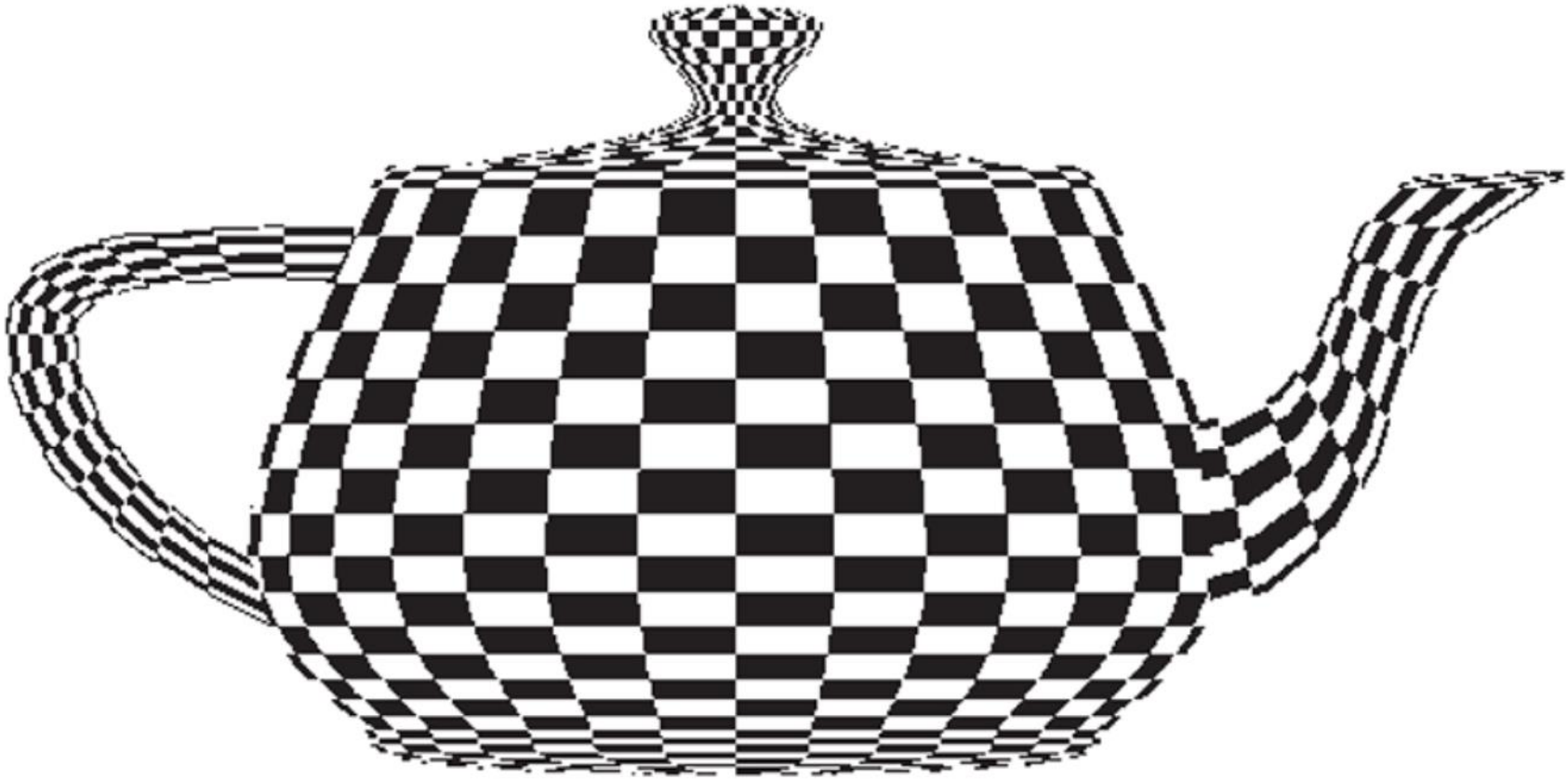


Working With Texture Coordinates

- By assigning the same set of texture coordinates to each patch, the texture-mapping process adjusts to the individual sizes of the triangles we use for rendering by scaling the texture map as needed. Hence, in areas such as the handle, where many small triangles are needed to give a good approximation to the curved surface, the black-and-white boxes are small compared to those on the body of the teapot. In some applications, these patterns are acceptable. However, if all surfaces of the teapot were made from the same material, we would expect to see the same pattern on all its parts. In principle, we could use the texture matrix to scale texture coordinates to achieve the desired display. However, in practice, it is almost impossible to determine the necessary information from the model to form the matrix.

Working With Texture Coordinates

Figure 7.22 Texture-mapped Utah teapot.



Working With Texture Coordinates

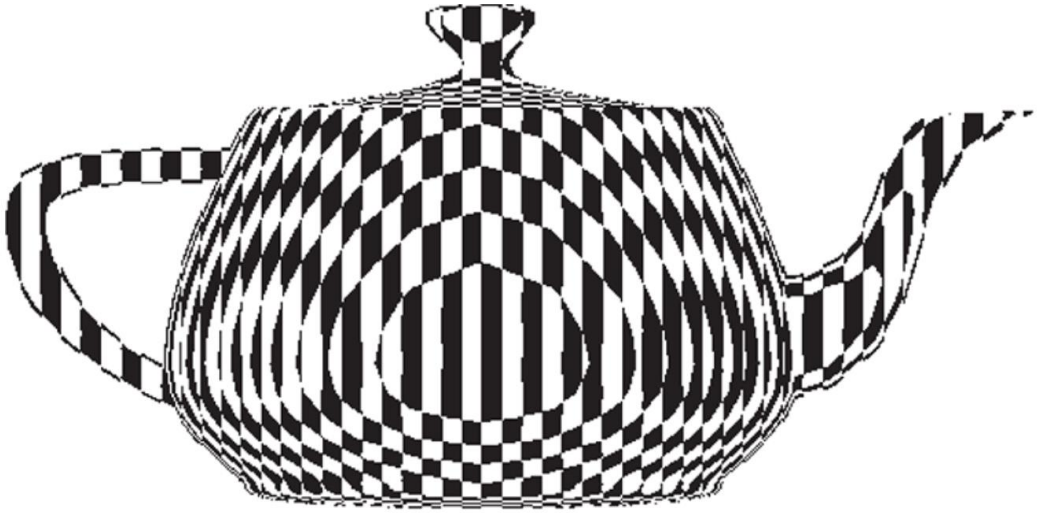
- One solution to this problem is to generate texture coordinates for each vertex in terms of the distance from a plane in either eye coordinates or object coordinates. Mathematically, each texture coordinate is given as a linear combination of the homogeneous-coordinate values. Thus, for *s* and *t*,

$$\begin{aligned} s &= a_s x + b_s y + c_s z + d_s w \\ t &= a_t x + b_t y + c_t z + d_t w. \end{aligned}$$

Working With Texture Coordinates

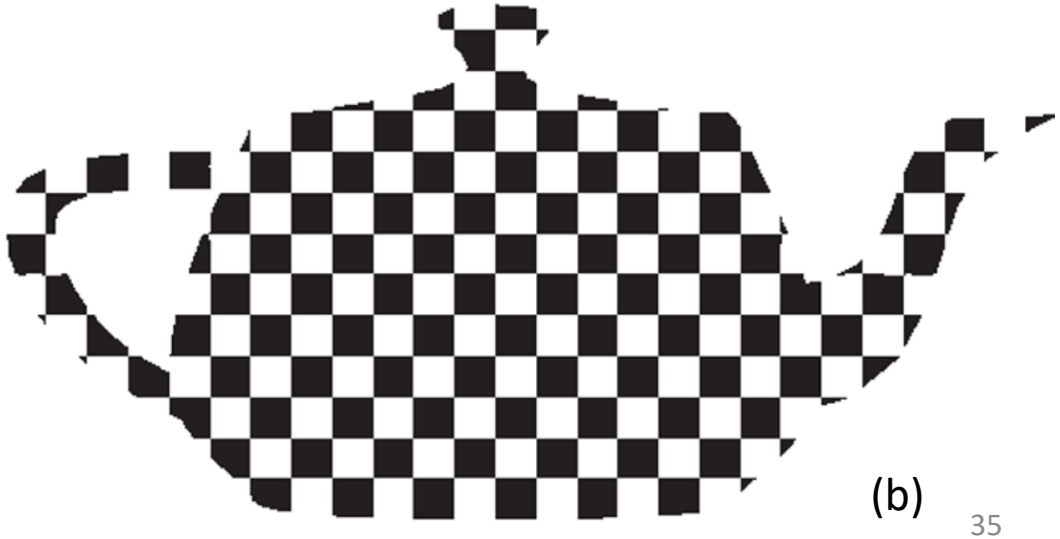
- (a) shows the teapot with texture-coordinate generation in object space.
- (b) uses the same equations but with the calculations in eye space.
- By doing the calculation in object space, the texture is fixed to the object and thus will rotate with the object. Using eye space, the texture pattern changes as we apply transformations to the object and give the illusion of the object moving through a texture field. One of the important applications of this technique is in terrain generation and mapping. We can map surface features as textures directly onto a three-dimensional mesh.

Working With Texture Coordinates



(a)

Figure 7.23 Teapot using texture coordinate generation. (a) In object coordinates. (b) In eye coordinates.



(b)