TEXTURE MAPPING

Lecturer: Asst. Prof. Ufuk Çelikcan

Based on the slides by: E. Angel and D. Shreiner

The Limits of Geometric Modeling

- Although graphics cards can render over billions of polygons per second (2017: according to the tests Nvidia GTX 1080 processes 11 billion triangles per second), that number is still not sufficient to render many phenomena together
  - Clouds
  - Grass
  - Terrain
  - Skin
  - …

Modeling an Orange

- Consider the problem of modeling an orange (the fruit)

- Start with an orange-colored sphere?
  - Too simple

- Replace sphere with a more complex shape?
  - Does not capture surface characteristics (small dimples)
  - Takes too many polygons to model all the dimples

Modeling an Orange (2)

- Take a picture of a real orange, scan it, and “paste” onto simple geometric model?
  - This process is known as **texture mapping**

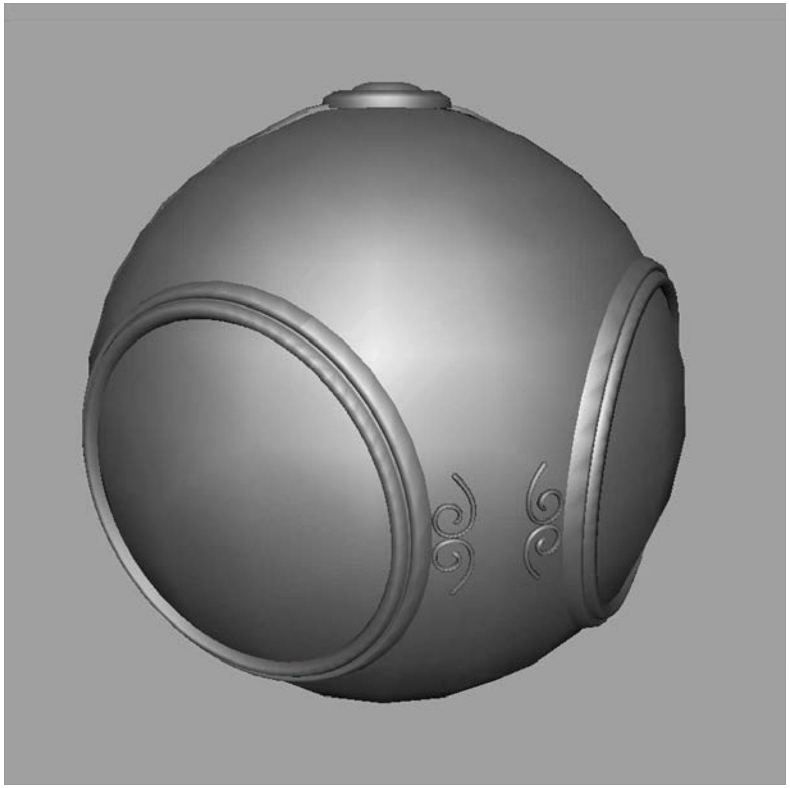
- Still might not be sufficient because resulting surface will be smooth
  - Need to change local shape

    + Bump mapping

Three Types of Mapping

- Texture Mapping
  - Uses images to fill inside of polygons
- Environment (reflection mapping)
  - Uses a picture of the environment for texture maps
  - Allows simulation of highly specular surfaces
- Bump mapping
  - Emulates altering normal vectors during the rendering process

Texture Mapping

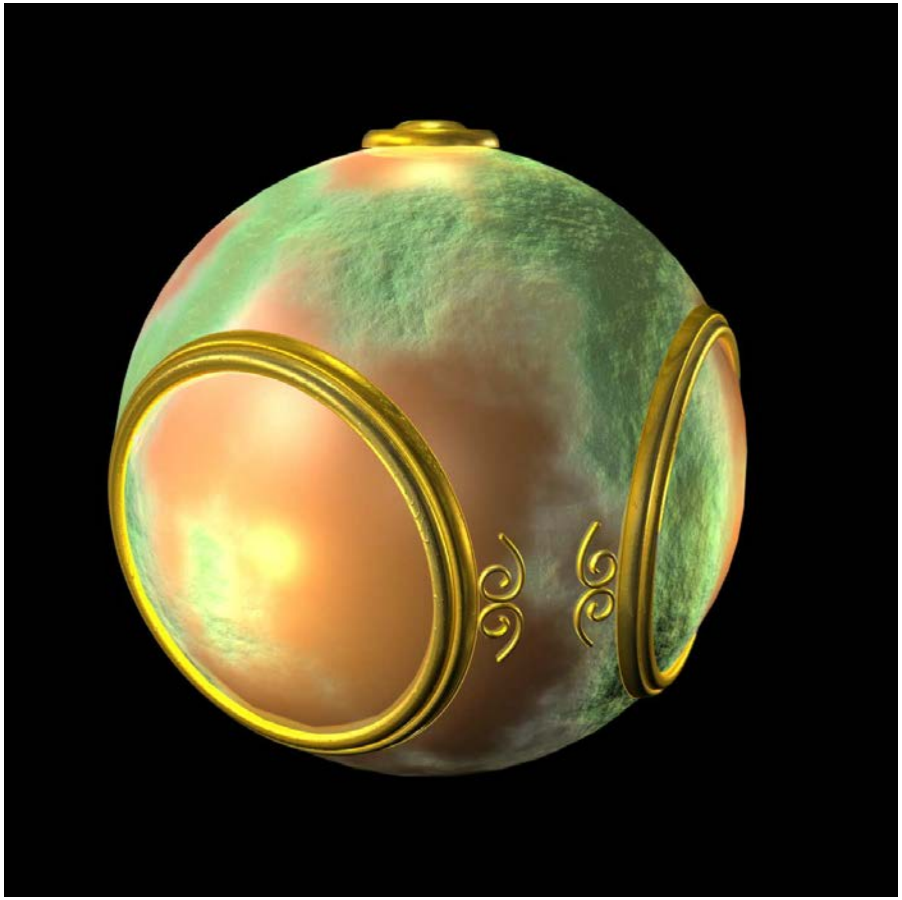


geometric model

texture mapped

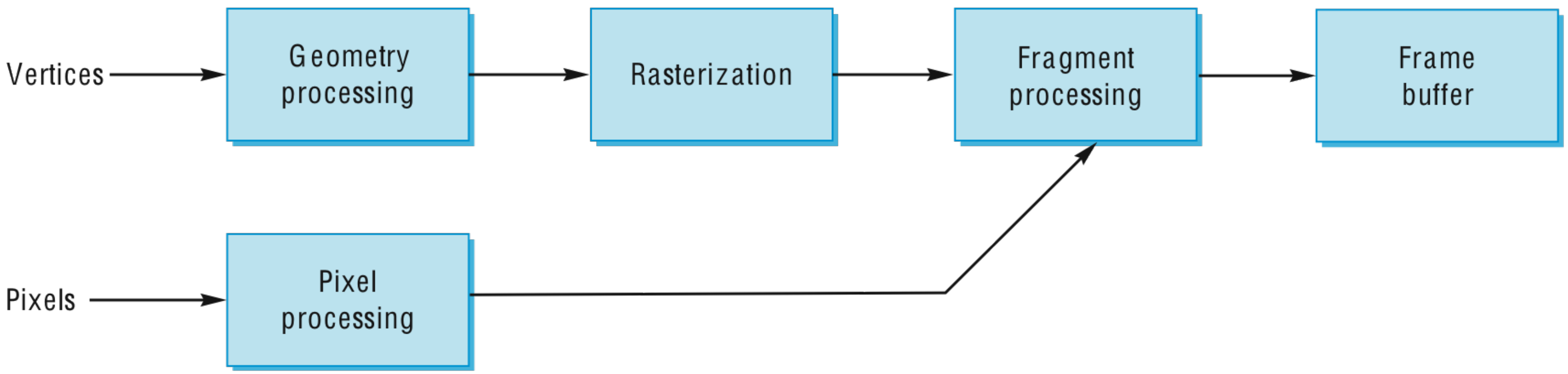
Environment Mapping

Bump Mapping



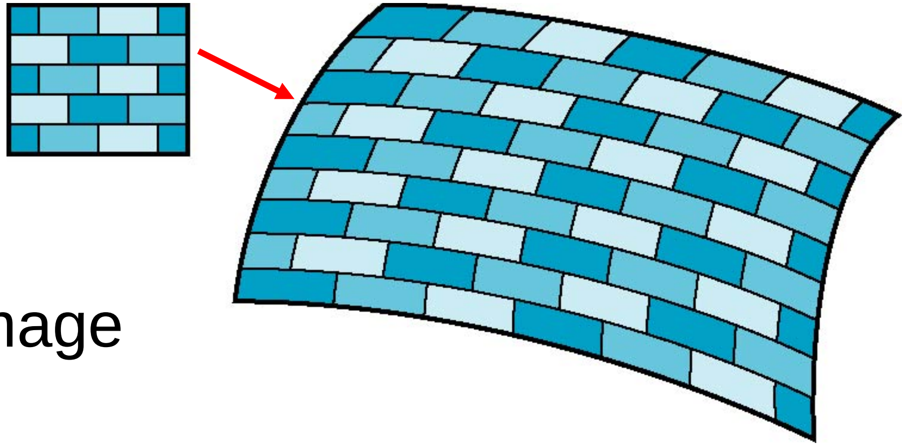
Where does mapping take place?

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



Is it simple?

- Although the idea is simple: map an image to a surface
- there are 3 or 4 coordinate systems involved



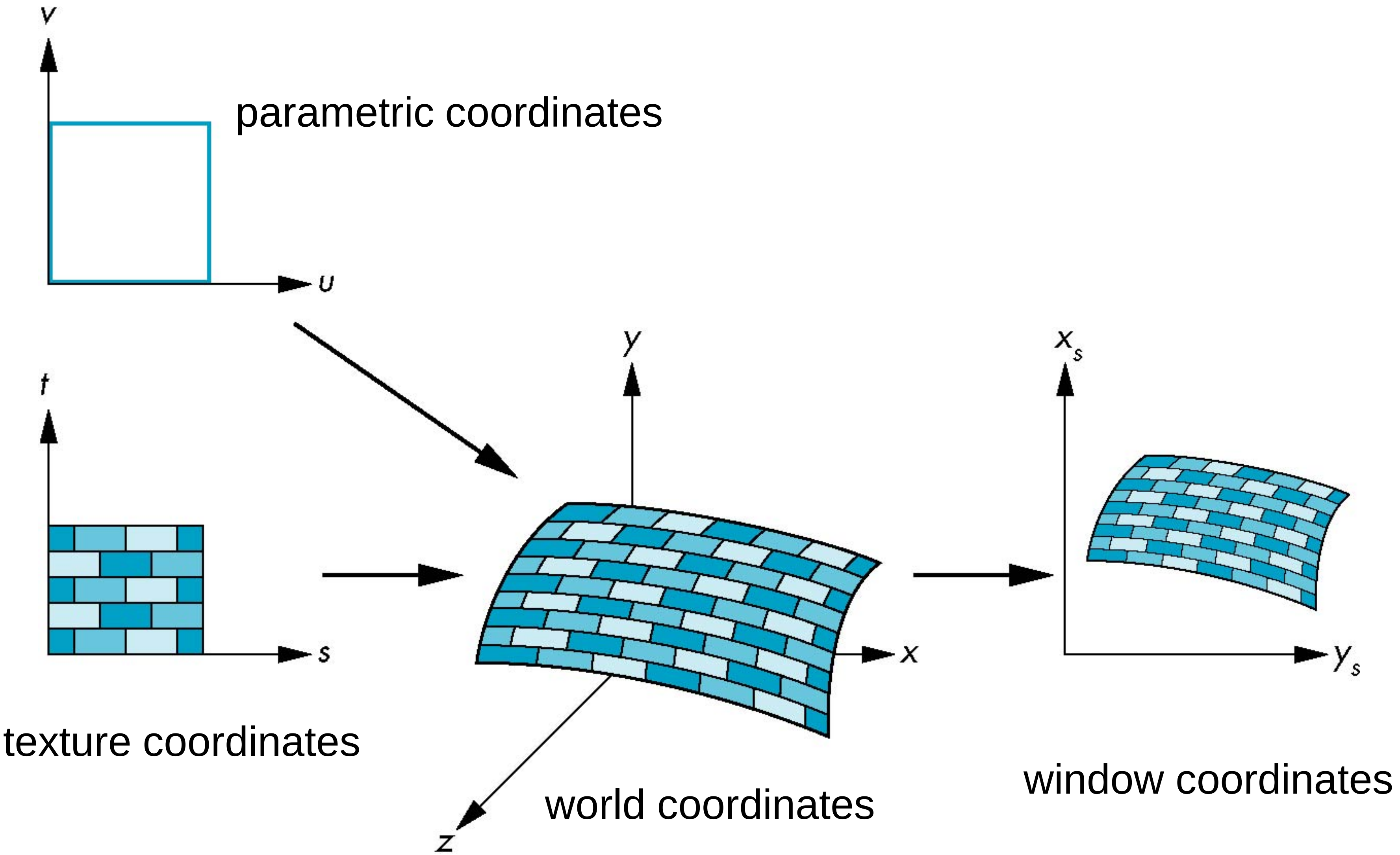
2D image

3D surface

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 World Coordinates
  - Conceptually, where the mapping takes place
- Window Coordinates
  - Where the final image is really produced

Texture Mapping



Mapping Functions

- Basic problem is **how to find the mappings**

- Consider mapping from texture coordinates to a point on a surface: "Given a texture pixel, which point over the object it covers?"
- Appears to need three functions

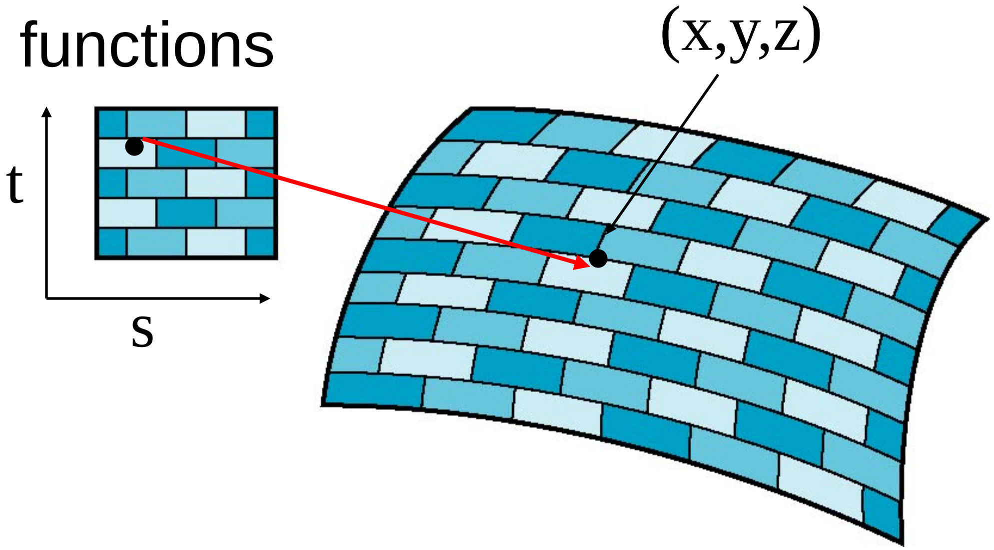
  x = x(s,t)

  y = y(s,t)

  z = z(s,t)

- **But we really want**

to go the other way



Backward Mapping

- We really want to go backwards
  - **Given a point on an object, we want to know to which point in the texture it corresponds**
- Need a map of the form

$$s = s(x,y,z)$$

$$t = t(x,y,z)$$

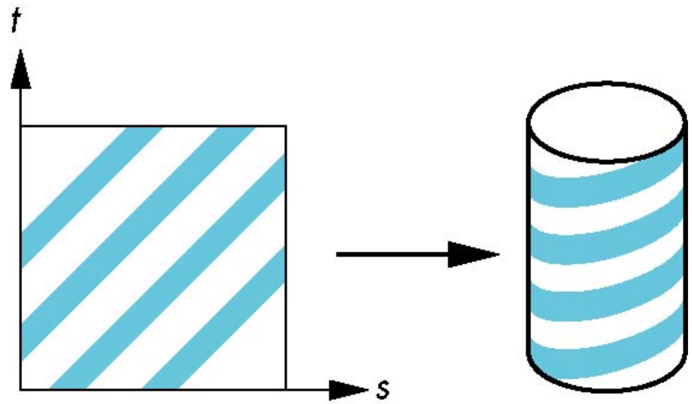
- Such functions are difficult to find in general

Two-part mapping

- One solution to the mapping problem is :

first map the texture to a simpler **intermediate surface**

- Example: map to cylinder



Texturing: Parameterization (Object Mesh)

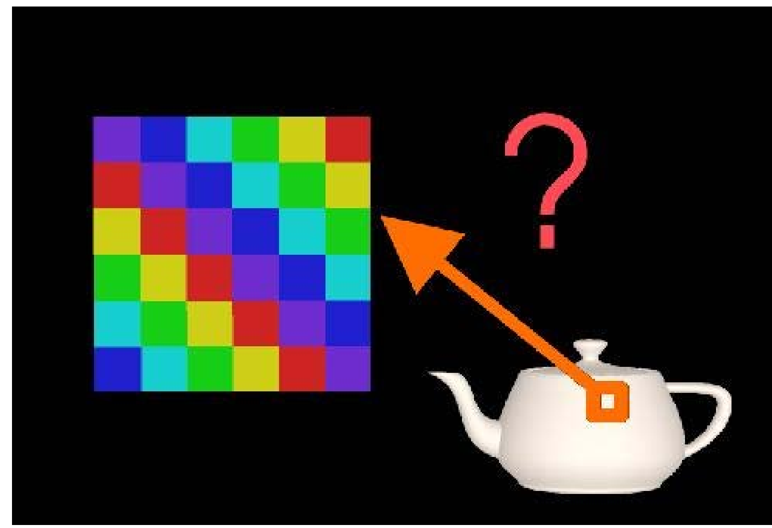
- How do we assign texture coordinates to objects?
  - Problem: Map from 3D to 2D
  - Idea: Map (x, y, z) to an intermediate space (u, v)

- **Projector** function to obtain object surface coordinates (u, v)

- **Corresponder** function to find **texel** coordinates (s, t)

- Filter texel at (s, t)
- Modify pixel (i, j)

Rasterization



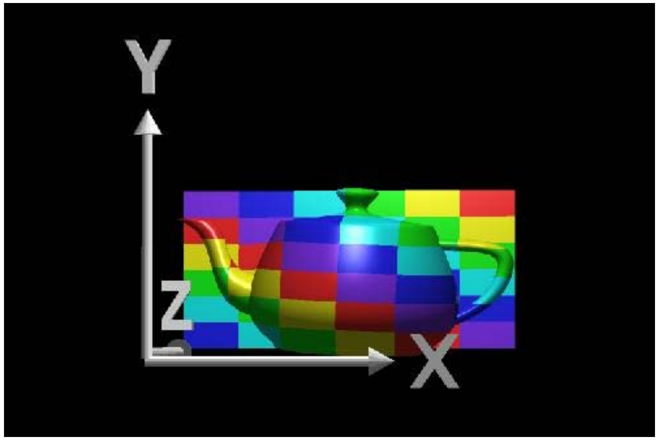
courtesy of R. Wolfe

Projector Functions

- How do we map the texture onto a arbitrary (complex) object?
  - Construct a mapping between the 3-D point to an intermediate surface
  - Why?
    - The intermediate surface is simple $\Rightarrow$ we know its characteristics
    - Still a 3D surface, but easier to map to texture space (2D)
    - Easy to parameterize the intermediate surface in 2D, i.e. (u, v) space

- Idea: Project each object point to the intermediate surface with a parallel or perspective projection
  - The focal point is usually placed inside the object

  - Plane
  - Cylinder
  - Sphere
  - Cube
  - Mesh: piece-wise planar

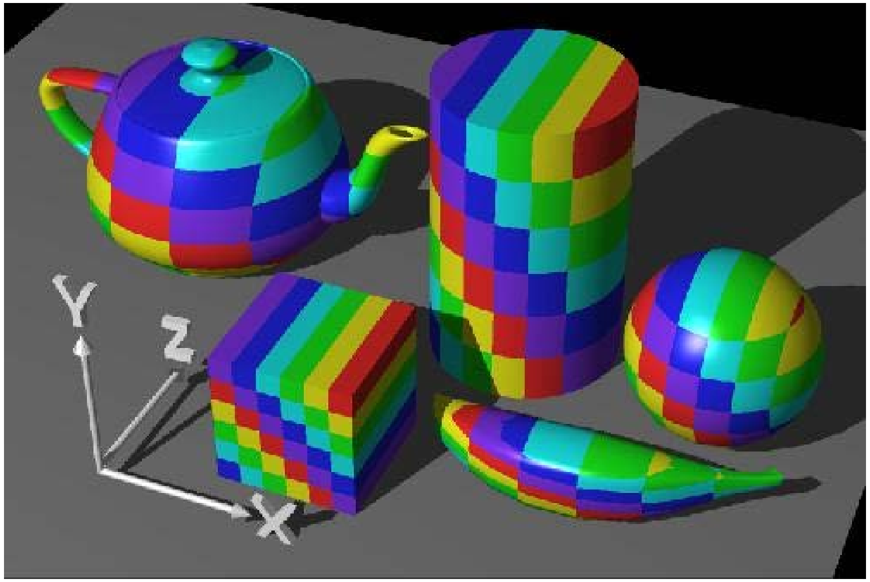


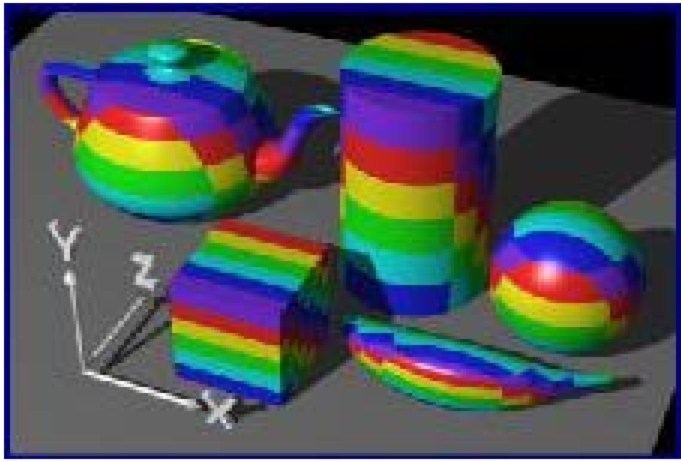
courtesy of R. Wolfe

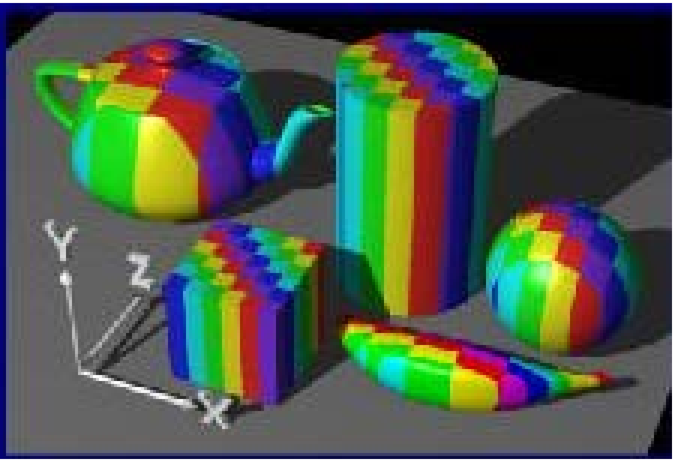
Planar projector

Planar Projector

Orthographic projection onto XY plane:
$u = x, \ v = y$







courtesy of
R. Wolfe

…onto YZ plane

…onto XZ plane

Cylindrical Mapping

parametric cylinder

$$x = r\cos(2\pi u)$$
$$y = r\sin(2\pi u)$$
$$z = v/h$$

Projector function

maps

- from a rectangle in **u,v** space
- to a cylinder of radius **r** and height **h** in world coordinates

$$s = u$$
$$t = v$$

Corresponder function

maps from texture space to the rectangle

Cylindrical Projector

courtesy of
R. Wolfe

Spherical Map

In a similar way, we can use a parametric sphere such as:

$$x = r\cos(2\pi u)$$
$$y = r\sin(2\pi u)\cos(2\pi v)$$
$$z = r\sin(2\pi u)\sin(2\pi v)$$

Projector function

Then once again

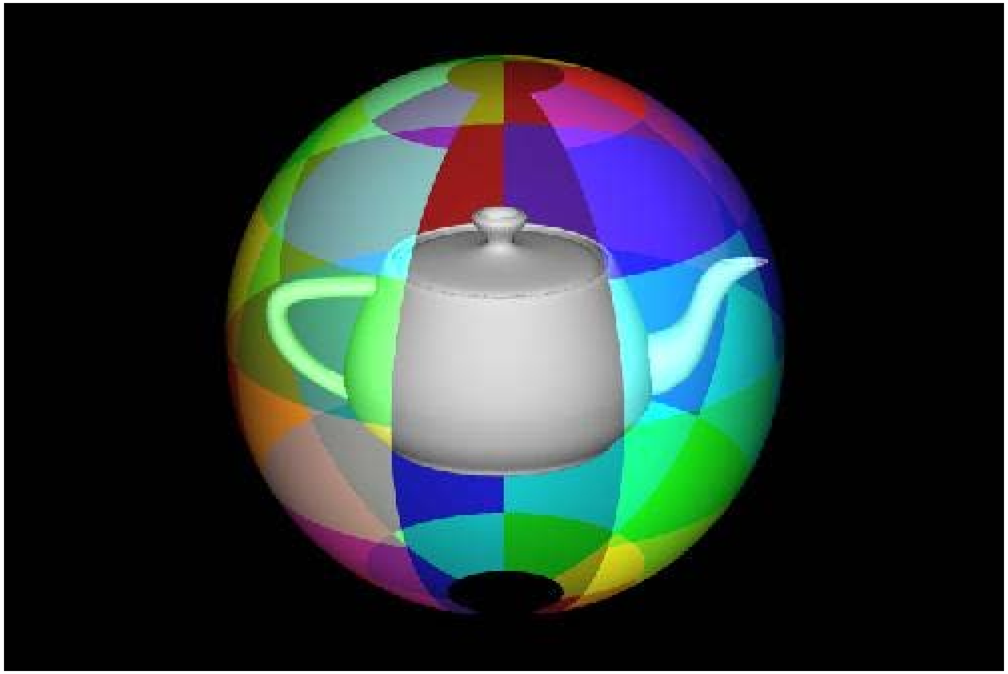
$$s = u$$
$$t = v$$

Corresponder function

maps from texture space to the rectangle

- but have to decide where to put the distortion.
- **Spheres are used in environmental maps**

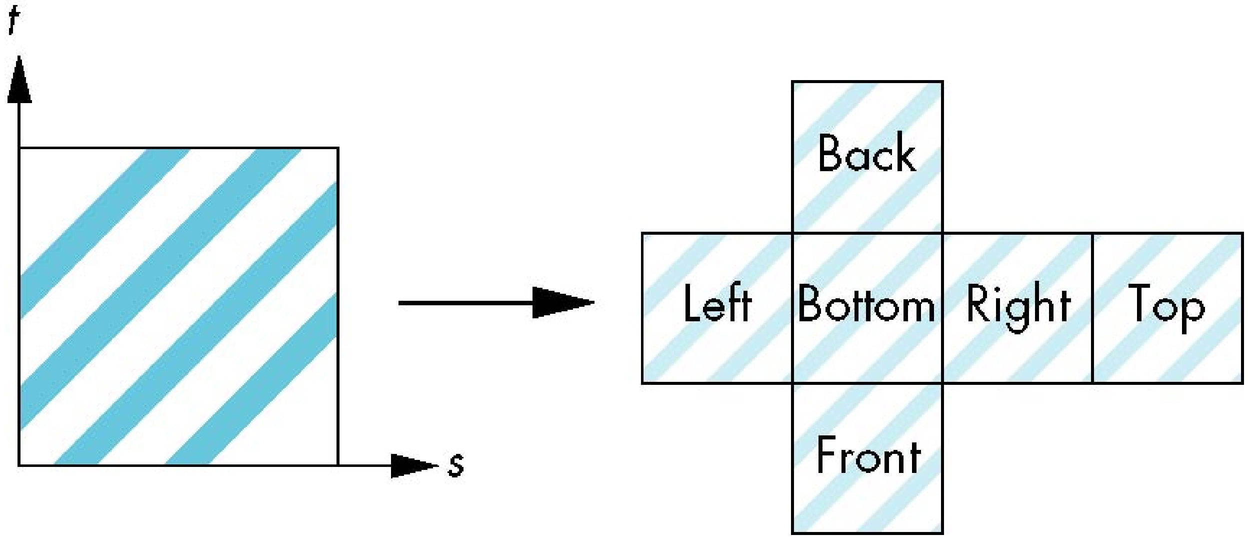
Spherical Projector



courtesy of R. Wolfe

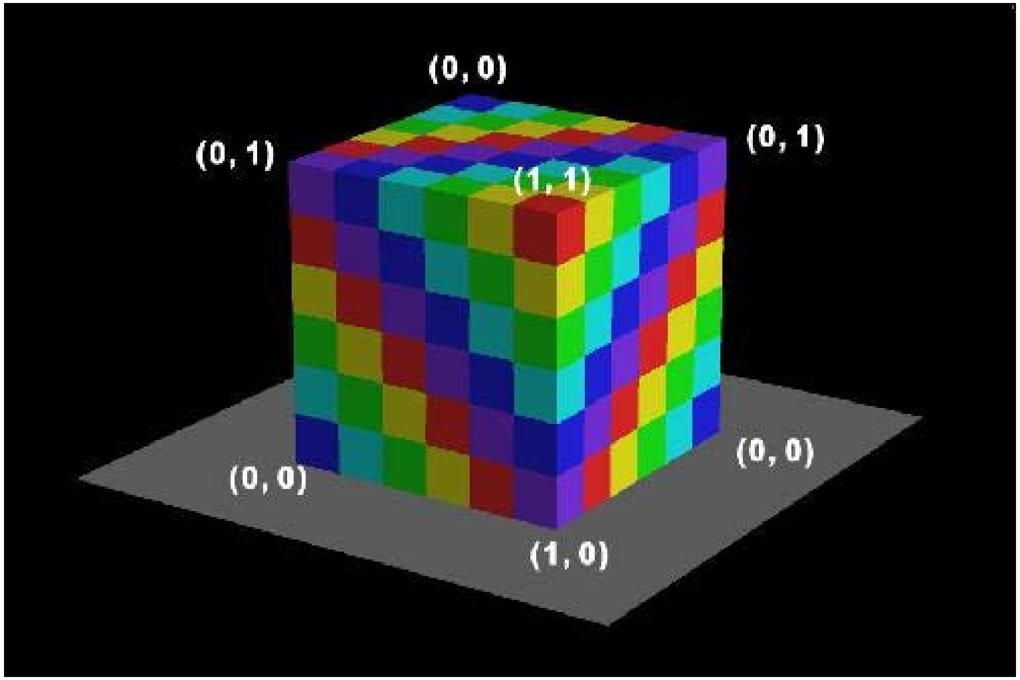
Box Mapping

- Easy to use with simple orthographic projection
- Also used in environment maps



Surface Patches

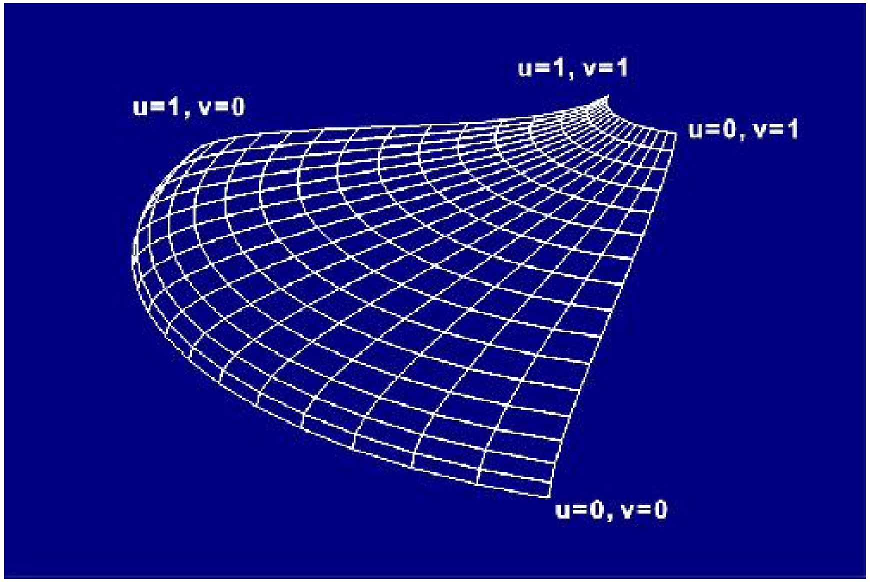
- A polygon or mesh of polygons defining a surface
  - Map four corners of a quad to (*u*, *v*) values



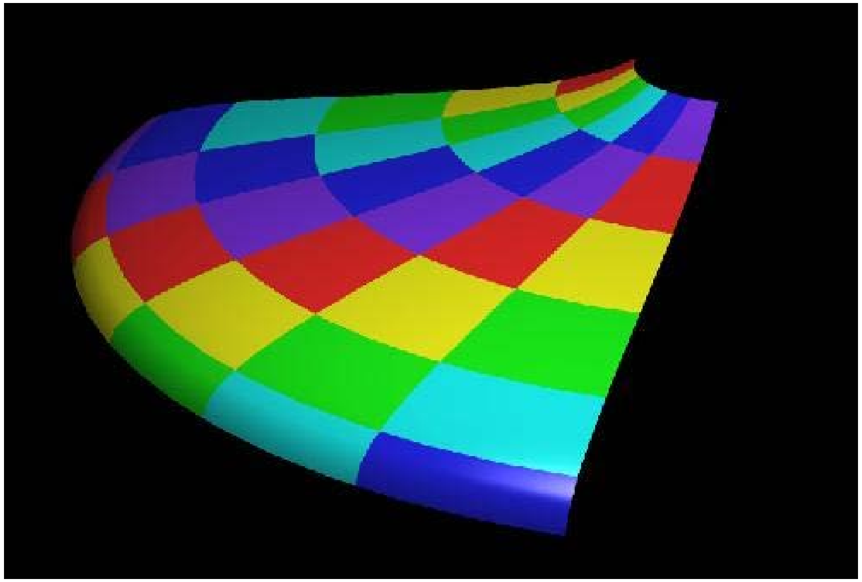
courtesy of
R. Wolfe

Parametric Surfaces

- A parameterized surface patch
  - $x = f(u, v)$, $y = g(u, v)$, $z = h(u, v)$

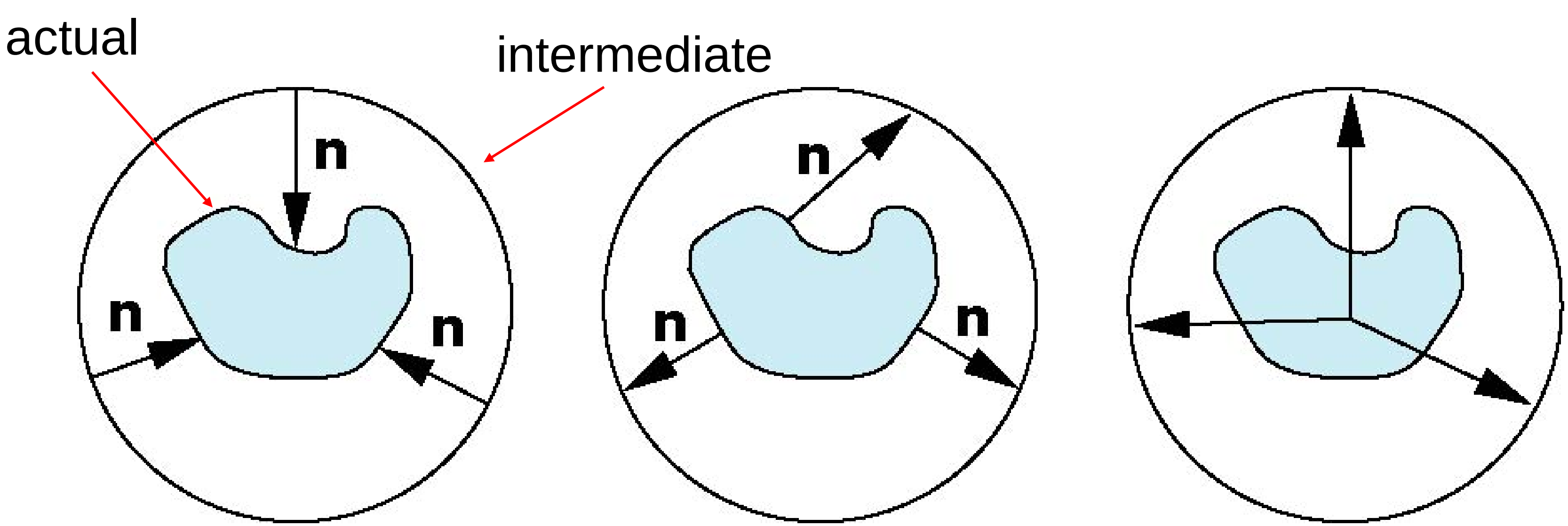


courtesy of R. Wolfe

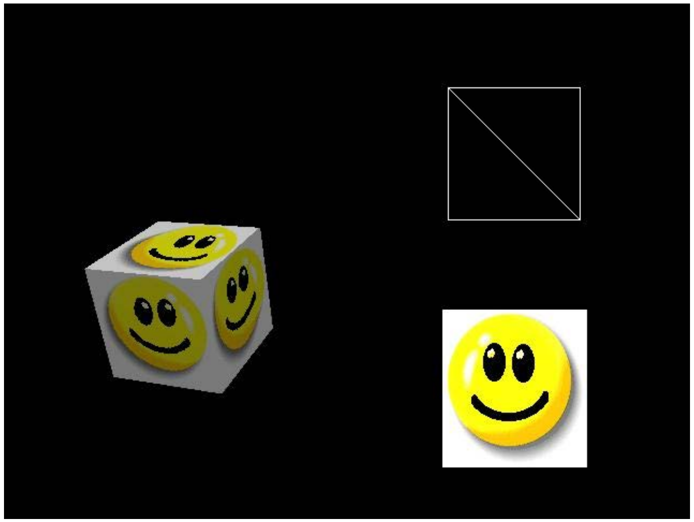


Second Mapping

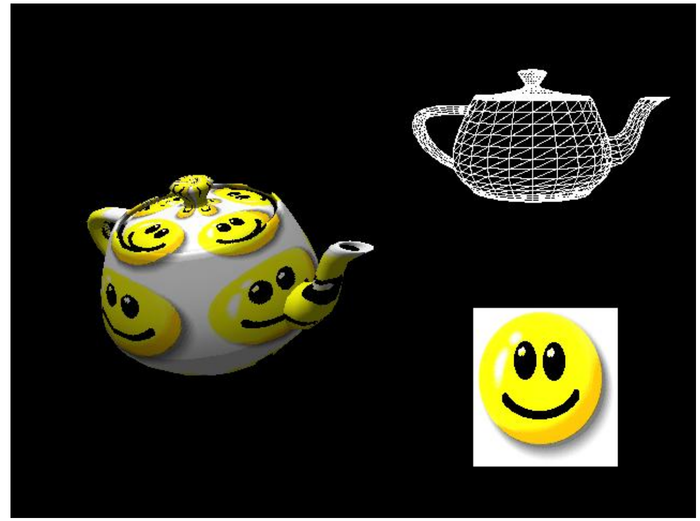
- Map from intermediate object to actual object by using
  1. Normals from intermediate to actual
  2. Normals from actual to intermediate
  3. Vectors from center of intermediate



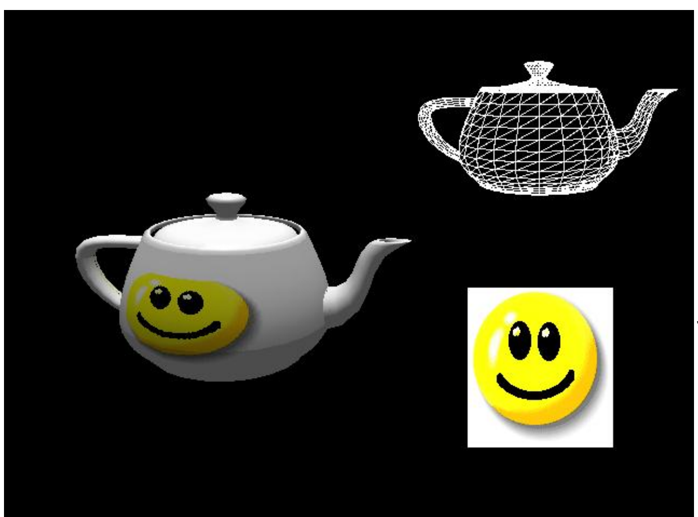
Examples courtesy of Jason Bryan



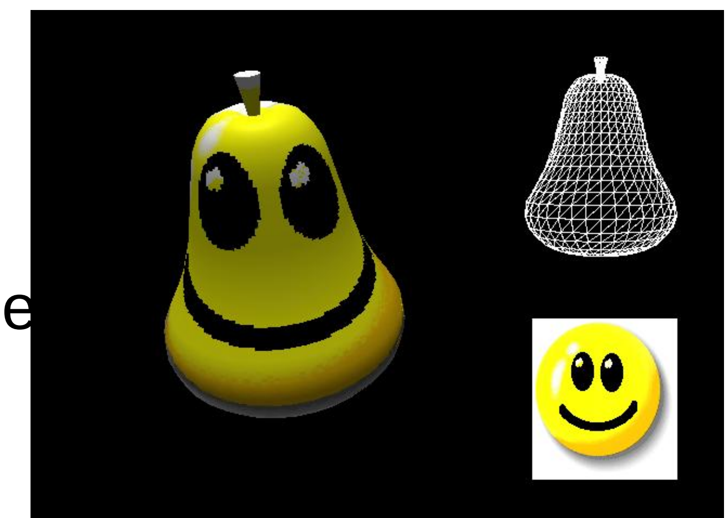
planar



spherical



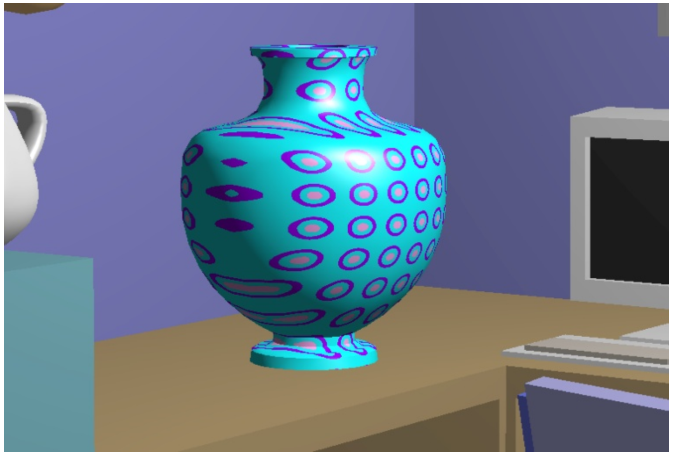
surface patch



cylindrical

Notice Distortions Due To Object Shape

Watt



planar

cylindrical

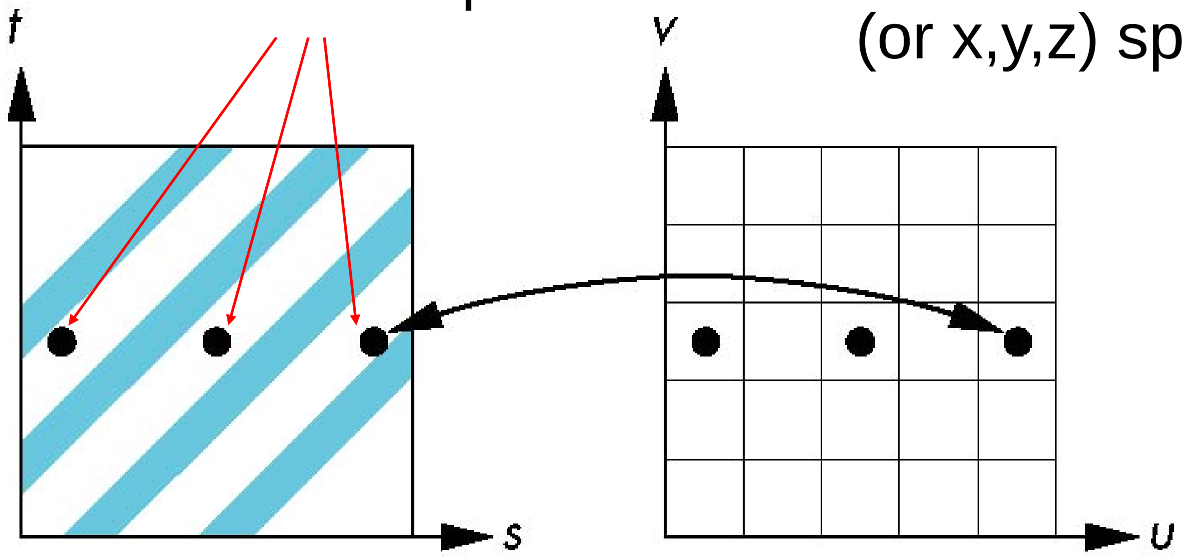
spherical

Aliasing

- Point sampling of the texture can lead to aliasing errors

miss blue stripes

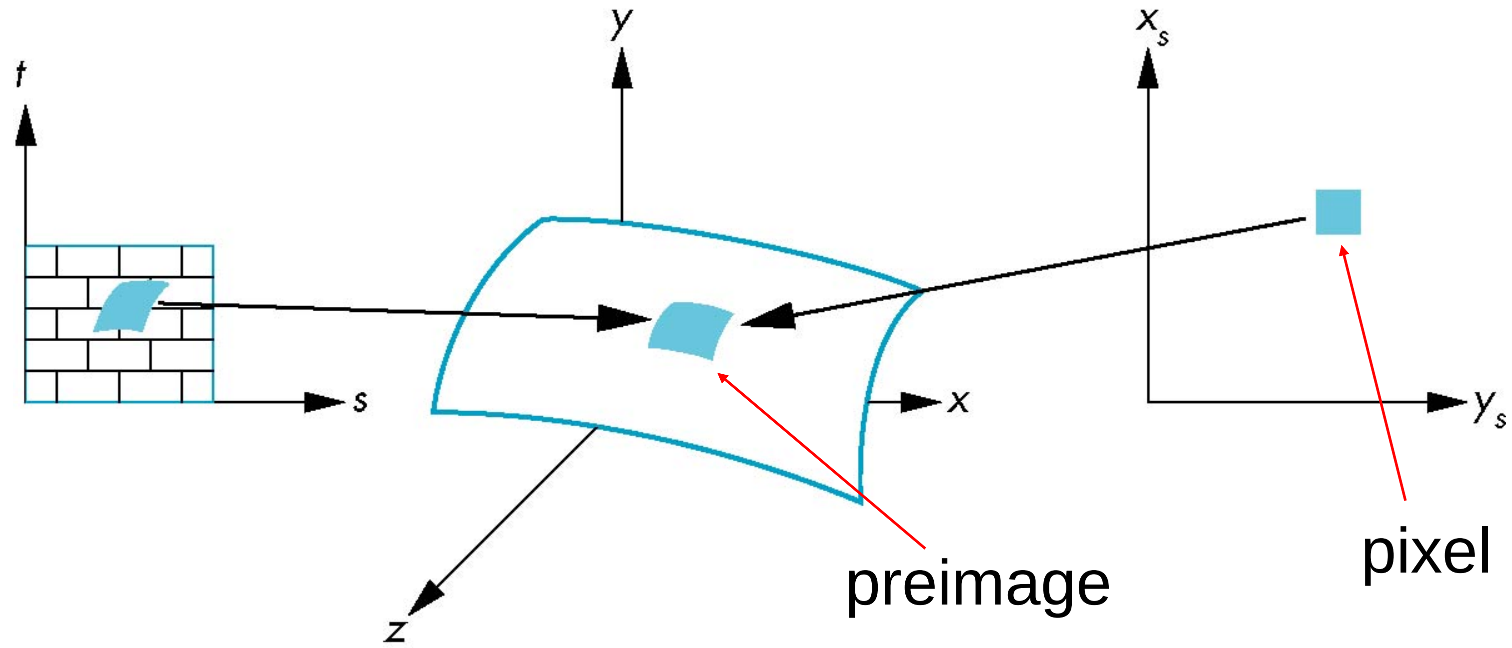
point samples in u,v
(or x,y,z) space



point samples in texture space

Area Averaging

A better but slower option is to use *area averaging*



Note that *preimage* of pixel is curved

OPENGL TEXTURE MAPPING

Lecturer: Asst. Prof. Ufuk Çelikcan

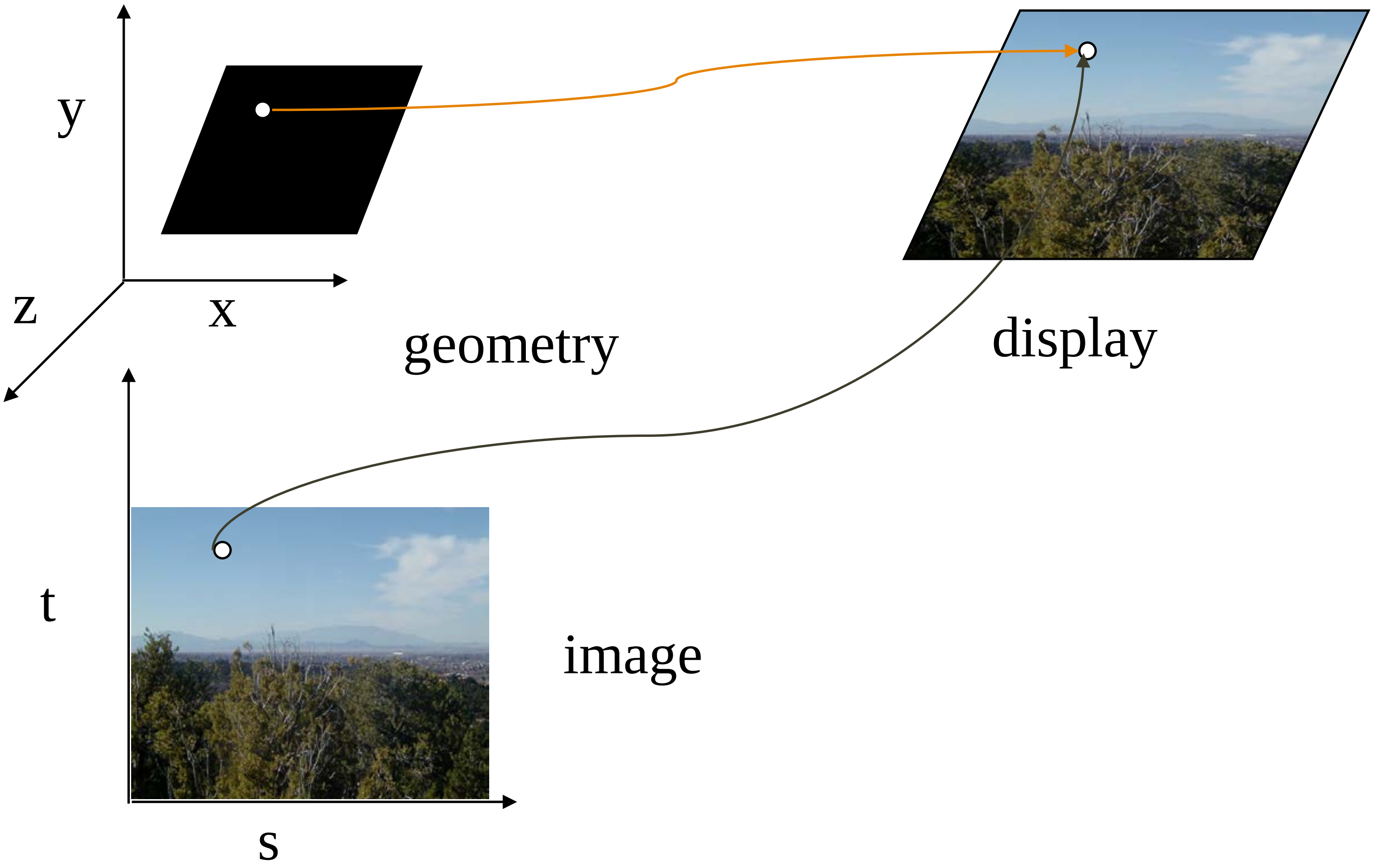
Based on the slides by: E. Angel and D. Shreiner

Basic Stragegy

Three steps to applying a texture

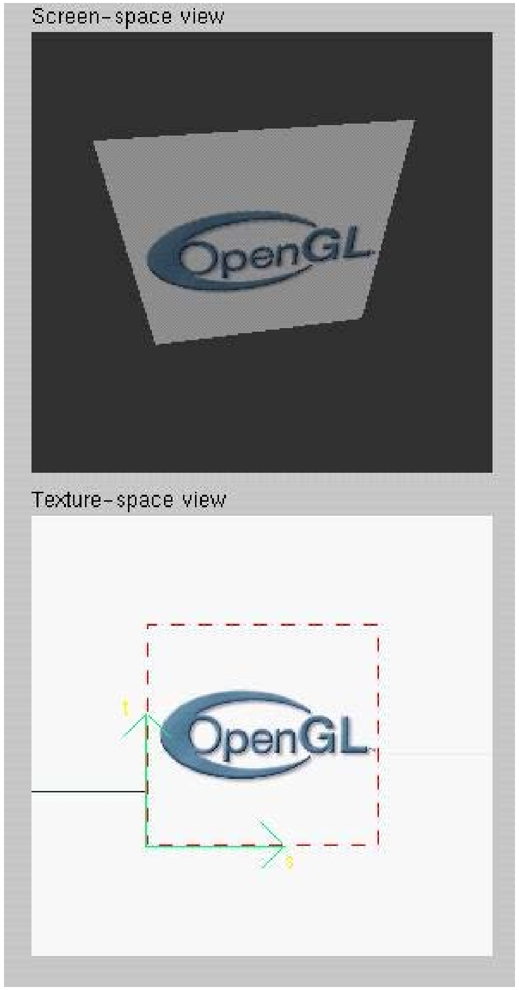
1. specify the texture
    - read or generate image
    - assign to texture
    - enable texturing
2. assign texture coordinates to vertices
    - proper mapping function is left to application >> you
3. specify texture parameters
    - wrapping, filtering ..

Texture Mapping



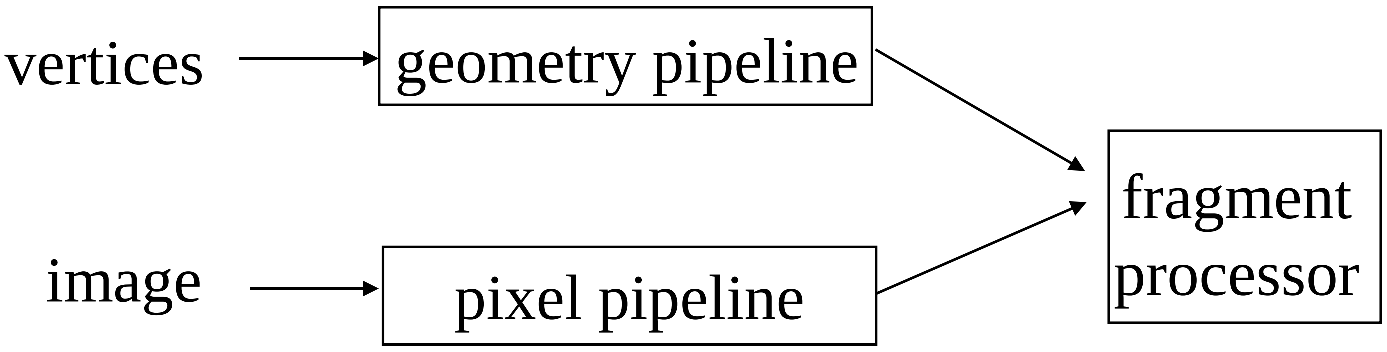
Texture Example

- The texture (below) is a 256 x 256 image that has been mapped to a rectangular polygon which is viewed in perspective



Texture Mapping and the OpenGL Pipeline

- Images and geometry flow through separate pipelines that unite during fragment processing
  - this way “complex” textures do not affect geometric complexity



Specifying a Texture Image

- Define a texture image from an array of *texels* (texture elements) in CPU memory

```
Glubyte my_texels[512][512];
```

- Define as any other pixel map
  - Scanned image
  - Generate by application code
- Enable texture mapping
  - **`glEnable(GL_TEXTURE_2D)`**
  - OpenGL supports 1-4 dimensional texture maps

Define Image as a Texture

```
glTexImage2D( target, level, components,
              w, h, border, format, type, texels );
```

- Specifying the texels for a texture is done using the `glTexImage{123}D()` call.
- This will transfer the texels in CPU memory to OpenGL, where they will be processed and converted into an internal format.

Define Image as a Texture

```
glTexImage2D( target, level, components,
              w, h, border, format, type, texels );
```

`target:` type of texture, e.g. **`GL_TEXTURE_2D`**

`level:` used for mipmapping

`components:` elements per texel, for rgb: 3

`w, h:` width and height of **`texels`** in pixels

`border:` used for smoothing (usually discarded : 0)

`format and type:` describe texels

`texels:` pointer to texel array

```
glTexImage2D(GL_TEXTURE_2D, 0, 3, 512, 512, 0, GL_RGB,
             GL_UNSIGNED_BYTE, my_texels);
```

Define Image as a Texture

```
glTexImage2D( target, level, components,
              w, h, border, format, type, texels );
```

- The level parameter is used for defining how OpenGL should use this image when mapping texels to pixels.
  - Generally, you’ll set the level to 0, unless you are using a texturing technique called mipmapping, which we will discuss later.

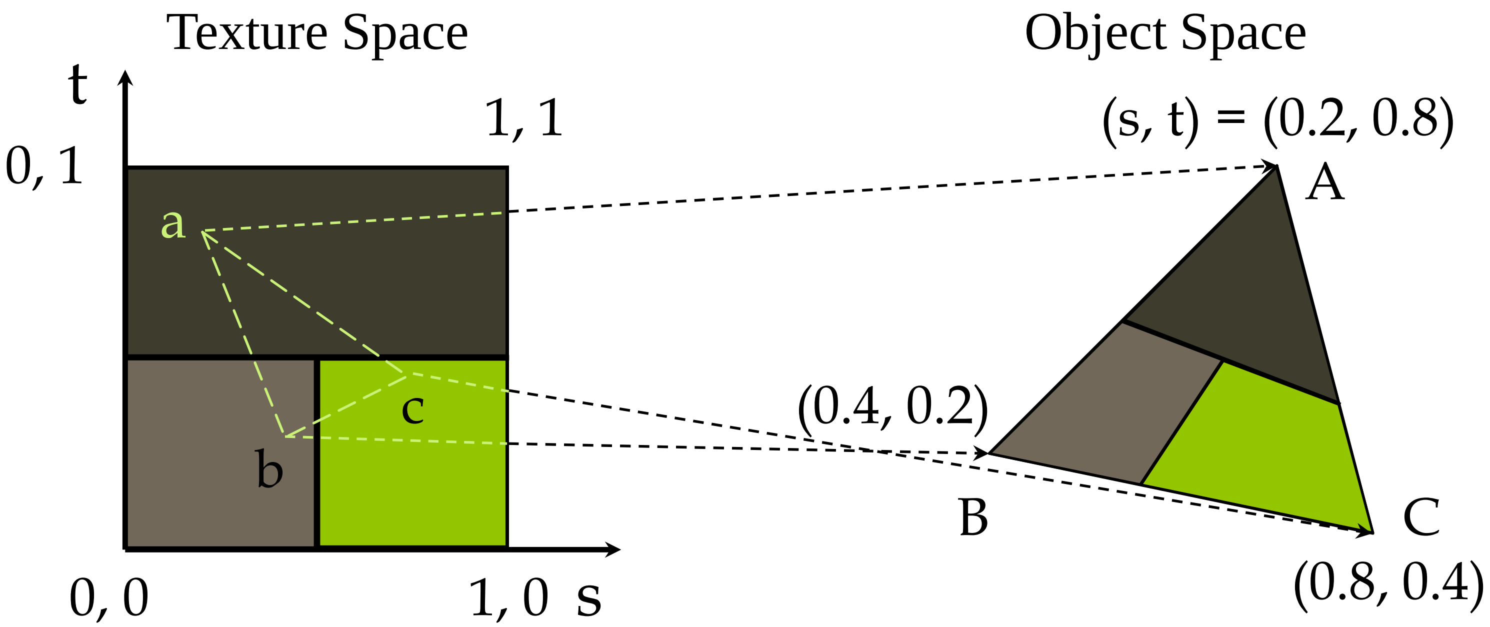
Mapping a Texture

- When you want to map a texture onto a geometric primitive, you need to provide texture coordinates.

- Valid texture coordinates are between 0 and 1, for each texture dimension, and usually manifest themselves in shaders as vertex attributes.
  - But we see how to deal with texture coordinates outside the range [0, 1] in a moment.

Mapping a Texture

- Based on parametric texture coordinates
- coordinates need to be specified at each vertex



Typical Code

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

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

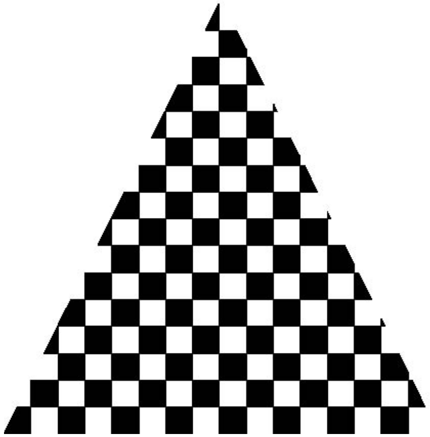
void **glVertexAttribPointer**(GLuint *index*, GLint *size*, GLenum *type*,
GLboolean *normalized*, GLsizei *stride*,
const GLvoid * *pointer*);

- *index* Specifies the index of the generic vertex attribute to be modified.
- *size* Specifies the number of components per generic vertex attribute. Must be 1, 2, 3, or 4. The initial value is 4.
- *type* Specifies the data type of each component in the array. Symbolic constants GL_BYTE, GL_UNSIGNED_BYTE, GL_SHORT, GL_UNSIGNED_SHORT, GL_FIXED, or GL_FLOAT are accepted. The initial value is GL_FLOAT.
- *normalized* Specifies whether fixed-point data values should be normalized (GL_TRUE) or converted directly as fixed-point values (GL_FALSE) when they are accessed.
- *stride* Specifies the byte offset between consecutive generic vertex attributes. If *stride* is 0, the generic vertex attributes are understood to be tightly packed in the array. The initial value is 0.
- *pointer* Specifies a pointer to the first component of the first generic vertex attribute in the array. The initial value is 0.

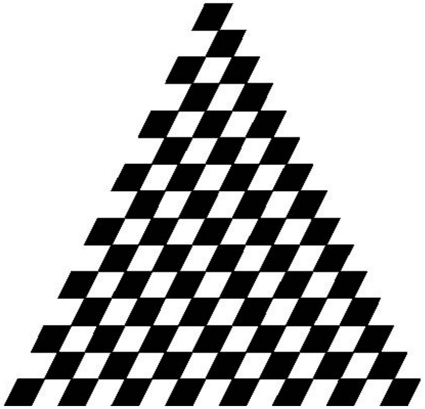
Interpolation

- OpenGL uses interpolation to find proper texels from specified texture coordinates
- Need to be careful. Can be distortions

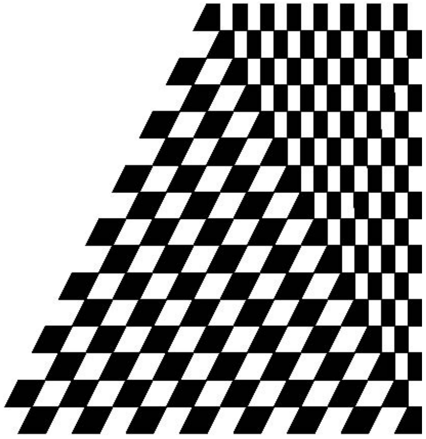
good selection of tex coordinates



poor selection of tex coordinates



texture stretched over trapezoid showing effects of bilinear interpolation



Texture Parameters

- OpenGL has a variety of parameters that determine how texture is applied
  - **Wrapping parameters** determine what happens if texture coordinates *s* and *t* are outside the (0,1) range
  - **Filter modes** allow 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 )
```

- Repeat: use s,t modulo 1

```
glTexParameteri( GL_TEXTURE_2D,
                 GL_TEXTURE_WRAP_T, GL_REPEAT )
```

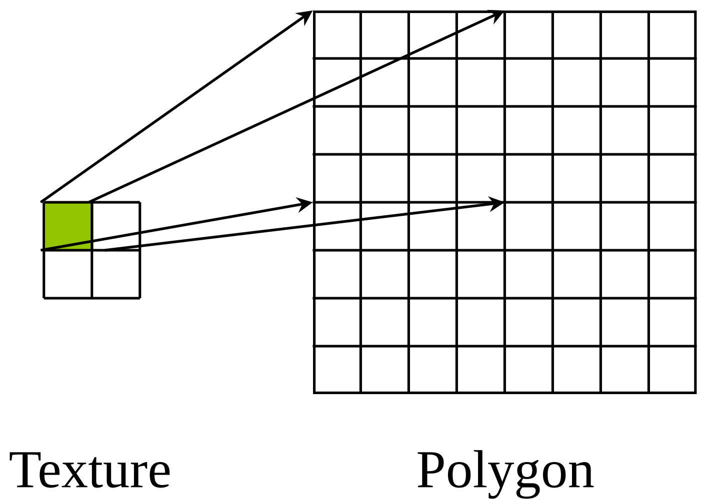
Magnification and Minification

- A single texel can cover more than one pixel : ***magnification***

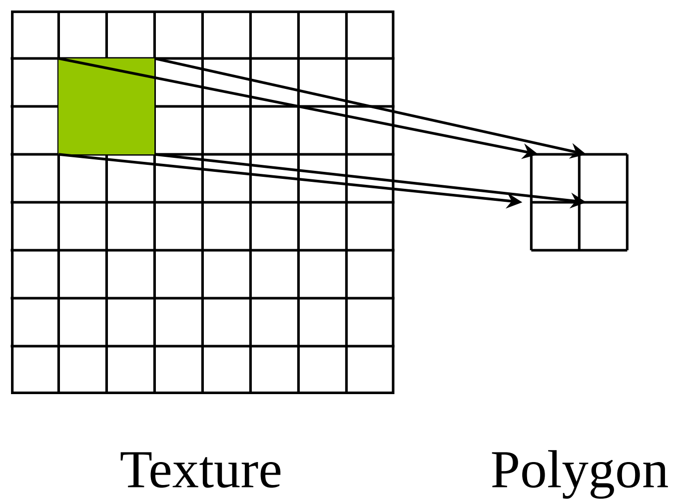
or

- More than one texel can cover a single pixel : ***minification***

Can use point sampling (nearest texel) or linear filtering
(2 x 2 filter) to obtain texture values



Magnification



Minification

Filter Modes

determined by

- **`glTexParameteri( target, type, mode )`**

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
                GL_NEAREST);

glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
                GL_LINEAR);
```

Note that linear filtering requires a border of an extra texel for filtering at edges >> border = 1

Mipmapped Textures

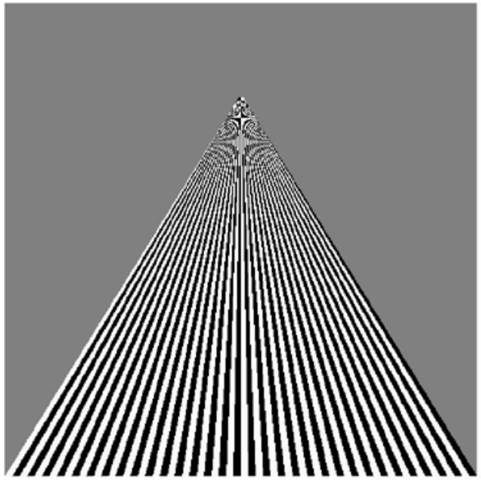
- *Mipmapping* allows for prefiltered texture maps of decreasing resolutions
- Lessens interpolation errors for smaller textured objects
- Declare mipmap level during texture definition

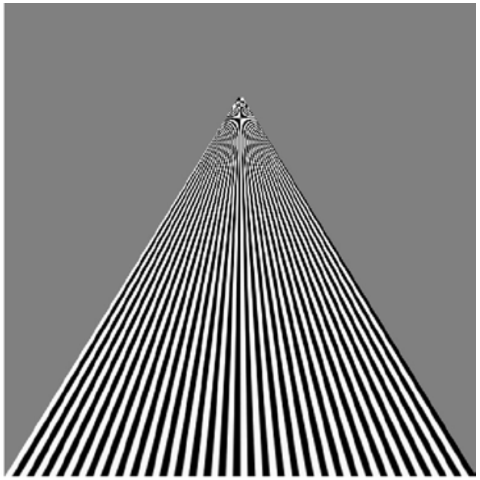
```
glTexImage2D( GL_TEXTURE_*D, level, … )
```

Example

https://www.cs.unm.edu/~angel/BOOK/INTERACTIVE_COMPUTER_GRAPHICS/SIXTH_EDITION/CODE/WebGL/CODE/07/textureSquare.html

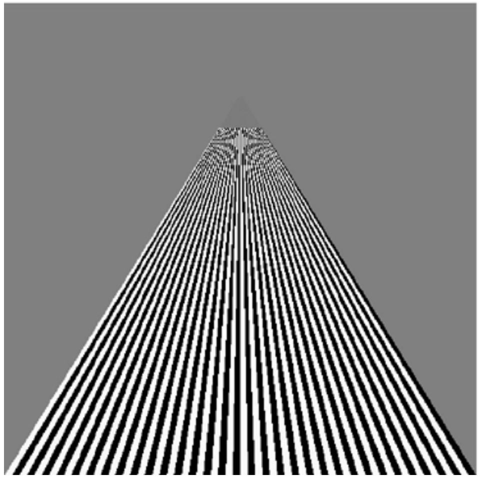
point sampling

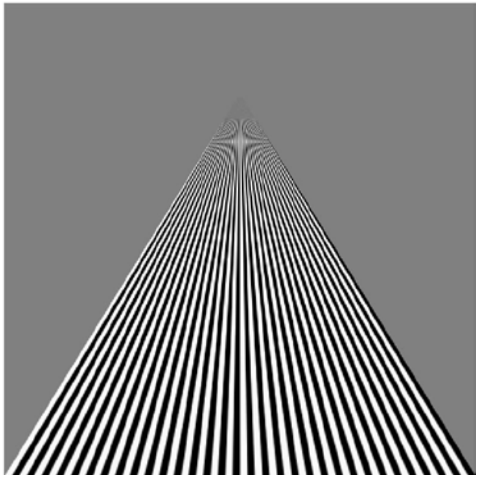




linear filtering

mipmapped point sampling





mipmapped linear filtering

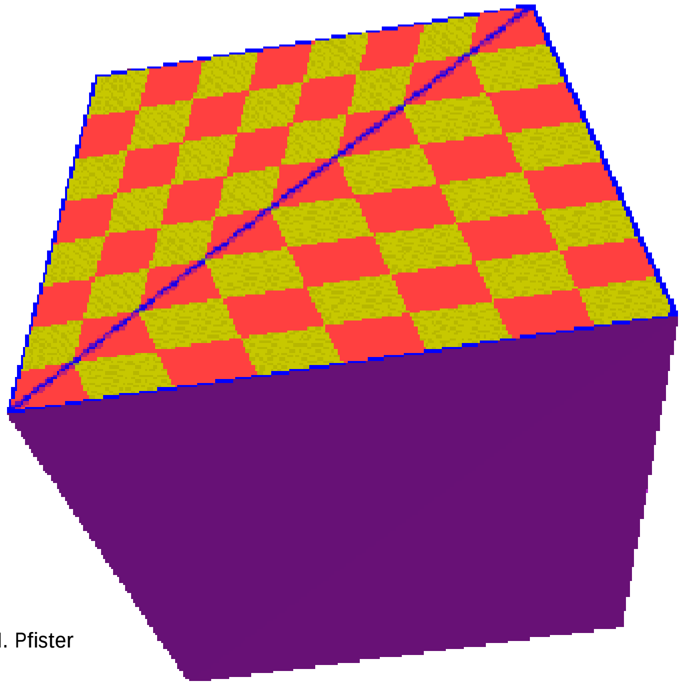
Perspective Correction Hint

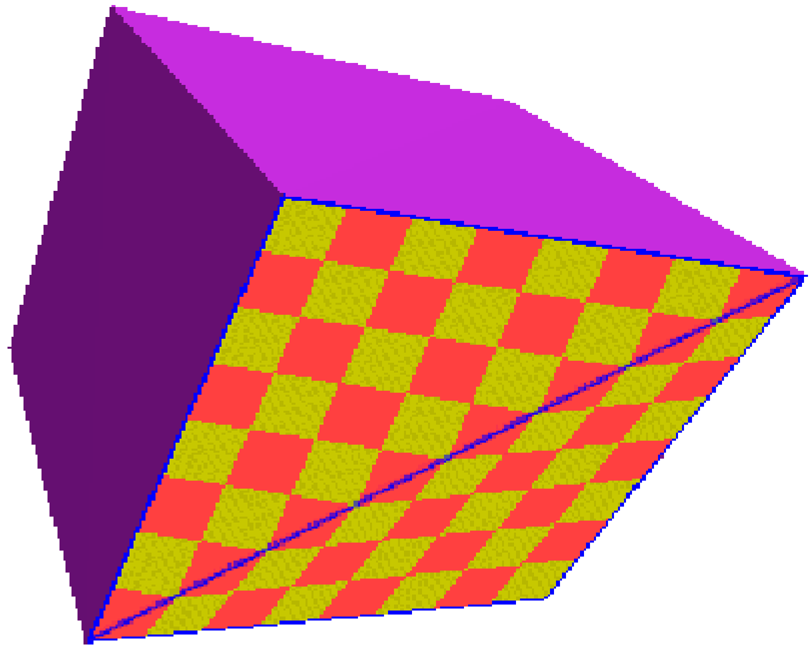
- Indicates the quality of color and texture coordinate interpolation
  - either linearly in screen space (wrong)
  - or using depth/perspective values (slower)

- Noticeable for polygons “on edge”

Linear Texture Coordinate Interpolation

- This doesn't work in perspective projection!
- The textures look warped along the diagonal
- Noticeable during an animation





courtesy of H. Pfister

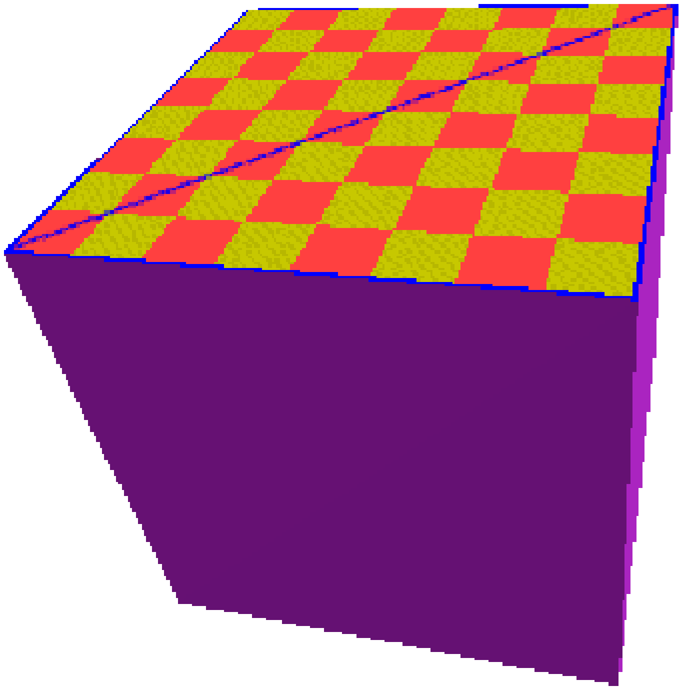
Perspective Correction Hint

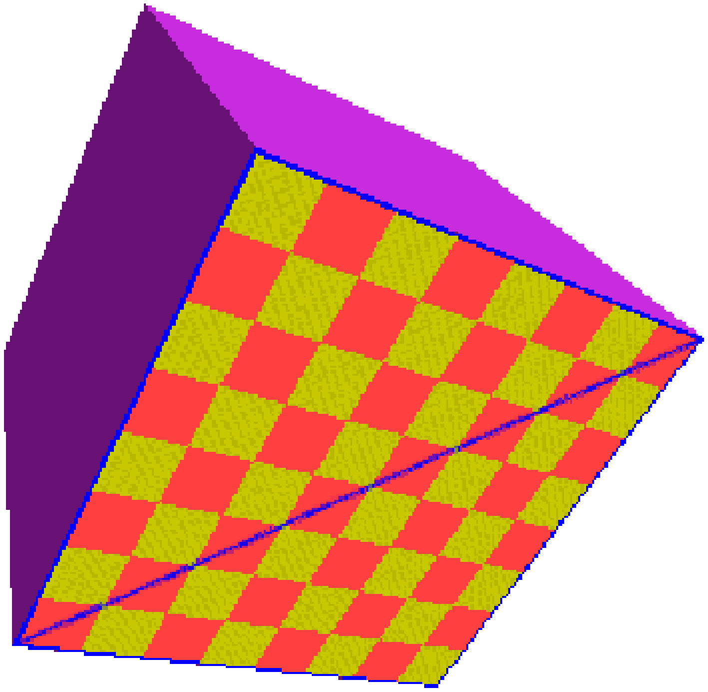
- GL_PERSPECTIVE_CORRECTION_HINT indicates the quality of color and texture coordinate interpolation.
  - **`glHint( GL_PERSPECTIVE_CORRECTION_HINT, hint )`**

  where **`hint`** is one of
    - **`GL_NICEST:`** The most correct, or highest quality, option should be chosen
    - **`GL_FASTEST:`** The most efficient option should be chosen.
    - **`GL_DONT_CARE:`** No preference.

Perspective-Correct Interpolation

- That fixed it!





Texture Objects

- OpenGL have *texture objects*
  - one image per texture object
  - Texture memory can hold multiple texture objects

Applying Textures

1. specify textures in texture objects
2. set texture filter
3. set texture function
4. set texture wrap mode
5. optional: set perspective correction hint
6. bind texture object
7. enable texturing
8. supply texture coordinates for vertex

Texture Objects

- Have OpenGL store your images
  - one image per texture object
  - may be shared by several graphics contexts
- The first step in creating texture objects is to have OpenGL reserve some indices for your objects

```
glGenTextures( n, *texIds );
```

  - *glGenTextures()* will request *n* texture *id*s and return those values back to you in *texIds*.

Texture Objects (cont'd.)

- Bind textures before using.
- To have OpenGL use a particular texture object, call glBindTexture() with the target and id of the object you want to make active.

```
glBindTexture( target, id );
```

  - The target is one of GL_TEXTURE_{123}D()

- To delete texture objects, use

```
glDeleteTextures( n, *texIds );
```

where texIds is an array of texture object identifiers to be deleted.

Applying a Texture in a Shader

- Just like vertex attributes are associated with data in the application, so too with textures.
- In particular, you access a texture defined in your application using a ***texture sampler*** in your shader.
- The type of the sampler needs to match the type of the associated texture.
- For example, you would use a sampler2D to work with a two-dimensional texture created with glTexImage2D(GL_TEXTURE_2D, …);

```
in vec4 texCoord;

// Declare the sampler
uniform float     intensity;
uniform sampler2D diffuseMaterialTexture;

// Apply the material color
vec3 diffuseColor = intensity *
   texture(diffuseMaterialTexture, texCoord).rgb;
```

Applying a Texture in a Shader

- Within the shader, you use the ***texture()*** function to retrieve data values from the texture associated with your sampler.
- To the texture() function, you pass the sampler as well as the texture coordinates where you want to pull the data from.
- the overloaded texture() method was added into GLSL version 3.30. Prior to that release, there were special texture functions for each type of texture sampler (e.g., there was a texture2D() call for use with the sampler2D).

```
in vec4 texCoord;


// Declare the sampler
uniform float     intensity;
uniform sampler2D diffuseMaterialTexture;

// Apply the material color
vec3 diffuseColor = intensity *
   texture(diffuseMaterialTexture, texCoord).rgb;
```

Now let's see our cube example with texturing

Applying Texture to Cube

- Similar to our first cube example, if we want to texture our cube, we need to provide texture coordinates for use in our shaders.

- Following our previous example, we merely add an additional vertex attribute that contains our texture coordinates. We do this for each vertex.

- We will also need to update VBOs and shaders to take this new attribute into account.

```
// add texture coordinate attribute  to
quad function

quad( int a, int b, int c, int d )
{
    vColors[Index] = colors[a];
    vPositions[Index] = positions[a];
    vTexCoords[Index] = vec2( 0.0, 0.0 );
    Index++;

    vColors[Index] = colors[b];
    vPositions[Index] = positions[b];
    vTexCoords[Index] = vec2( 1.0, 0.0 );
    Index++;
   … // rest of vertices
}
```

Creating a Texture Image

```
// Procedurally create two 64*64 checkerboard patterns
for ( int i = 0; i < 64; i++ ) {
    for ( int j = 0; j < 64; j++ ) {
        GLubyte c;
        c = ((i & 0x8 == 0) ^ (j & 0x8 == 0)) * 255;
        image[i][j][0]  = c;
        image[i][j][1]  = c;
        image[i][j][2]  = c;
        image2[i][j][0] = c;
        image2[i][j][1] = 0;
        image2[i][j][2] = c;
    }
}
```

You probably won’t procedurally generate your textures, but rather read them from image files

Texture Object

- Below code completely specifies a texture object.
- The code creates a texture id by calling **glGenTextures()**
- It then binds the texture using **glBindTexture()** to enable the object for use, and loading in the texture by calling **glTexImage2D()**
- After that, numerous sampler characteristics are set, including the texture wrap modes, and texel filtering.

```
GLuint textures[1];
glGenTextures( 1, textures );

glActiveTexture( GL_TEXTURE0 );
glBindTexture( GL_TEXTURE_2D, textures[0] );

glTexImage2D( GL_TEXTURE_2D, 0, GL_RGB, TextureSize,
              TextureSize, GL_RGB, GL_UNSIGNED_BYTE, image );

glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT );
glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT );
glTexParameteri( GL_TEXTURE_2D,
                 GL_TEXTURE_MAG_FILTER, GL_NEAREST );
glTexParameteri( GL_TEXTURE_2D,
                 GL_TEXTURE_MIN_FILTER, GL_NEAREST );
glUniform1i(glGetUniformLocation(shaderProgram, "texCheckerboard"), 0);
```

Typical Code

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

offset += sizeof(points);
GLuint vTexCoord = glGetAttribLocation(shaderProgram,
 "vTexCoord" );
glEnableVertexAttribArray( vTexCoord );
glVertexAttribPointer( vTexCoord, 2,GL_FLOAT,
    GL_FALSE, 0, BUFFER_OFFSET(offset) );
```

Vertex Shader

```
#version 150
in vec4 vPosition;
in vec4 vColor;
in vec2 vTexCoord;

out vec4 color;
out vec2 texCoord;

void main()
{
    color       = vColor;
    texCoord    = vTexCoord;
    gl_Position = vPosition;
}
```

- In order to apply textures to our geometry, we need to modify both the vertex shader and the fragment shader.
- we add some simple logic to pass-thru the texture coordinates from an attribute into data for the rasterizer.

Fragment Shader

- Continuing to update our shaders, we add some simple code to modify our fragment shader to include sampling a texture.
- How the texture is sampled (e.g., coordinate wrap modes, texel filtering, etc.) is configured in the application using the glTexParameter*() call.

```
#version 150
in vec4 color;
in vec2 texCoord;

out vec4 fColor;

uniform sampler2D texCheckerboard;

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

Let’s see it in action

https://www.cs.unm.edu/~angel/BOOK/INTERACTIVE_COMPUTER_GRAPHICS/SIXTH_EDITION/CODE/WebGL/CODE/07/textureCube1.html

take a look at the page source of above page for shaders’ implementation and

https://www.cs.unm.edu/~angel/BOOK/INTERACTIVE_COMPUTER_GRAPHICS/SIXTH_EDITION/CODE/WebGL/CODE/07/textureCube1.js

for OpenGL side (yes it is a WebGL implementation but you already know how to relate it to OpenGL code and, as always, Google is you best friend for this type of thing)

You can also take a look at the shader-based OpenGL texturing code examples of the book at

https://www.cs.unm.edu/~angel/BOOK/INTERACTIVE_COMPUTER_GRAPHICS/SIXTH_EDITION/CODE/CHAPTER07/WINDOWS_VERSIONS/

SHADER APPLICATIONS

Lecturer: Asst. Prof. Ufuk Çelikcan

Based on the slides by: E. Angel and D. Shreiner

Vertex Shader Applications

- Moving vertices
  - Morphing
  - Wave motion
  - Fractals

- Lighting
  - More realistic models
  - Cartoon shaders

Simple **Wave Motion** Vertex Shader

```
uniform float time;
uniform float xs, zs, // frequencies
uniform float h; // height scale
uniform mat4 ModelView, Projection;
in vec4 vPosition;

void main() {
   vec4 t = vPosition;
   t.y = vPosition.y
         + h*sin(time + xs*vPosition.x)
         + h*sin(time + zs*vPosition.z);
   gl_Position = Projection*ModelView*t;
}
```

Simple **Particle System** Vertex Shader

```
uniform vec3 init_vel;
uniform float g, t;
uniform mat4 Projection, ModelView;
in vec4 vPosition;

void main()
{
    vec3 object_pos;

    object_pos.x = vPosition.x + init_vel.x*t;
    object_pos.y = vPosition.y + init_vel.y*t + g*t*t / 2.0;
    object_pos.z = vPosition.z + init_vel.z*t;

    gl_Position = Projection*ModelView*vec4(object_pos, 1);
}
```

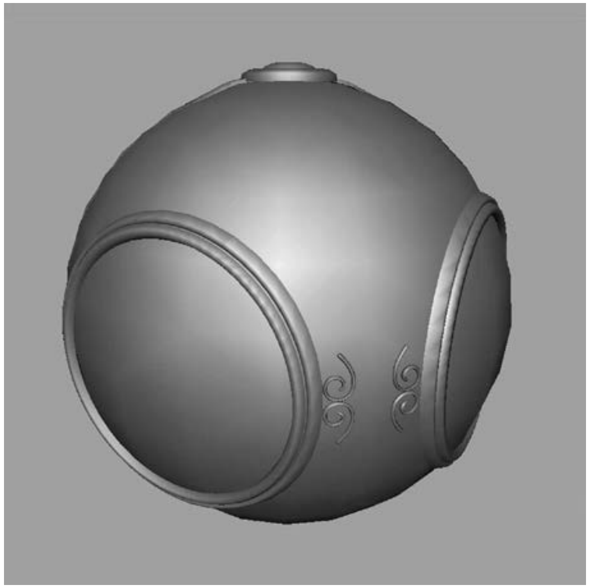
free fall:

$y = gt^2/2 + v_i t$

The only force on a particle in the air is the gravitational force which is in –y direction

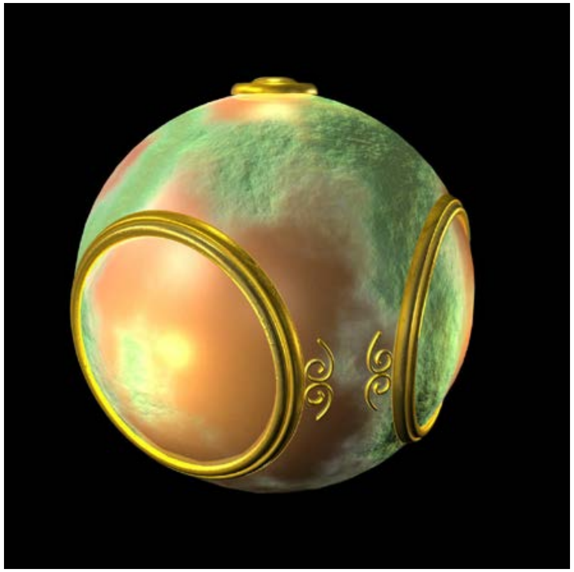
Fragment Shader Applications

Texture mapping



smooth shading

environment mapping

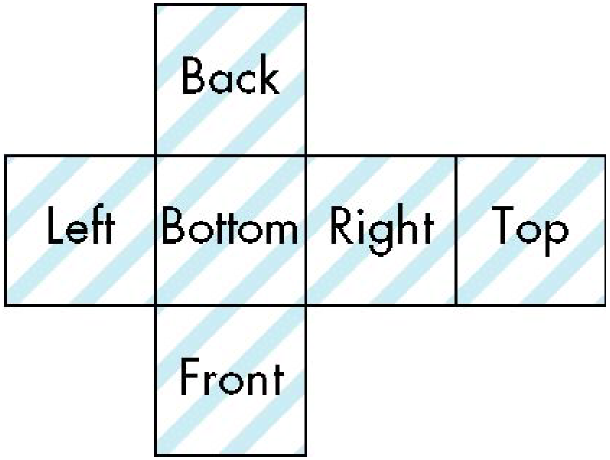


bump mapping

Cube Maps

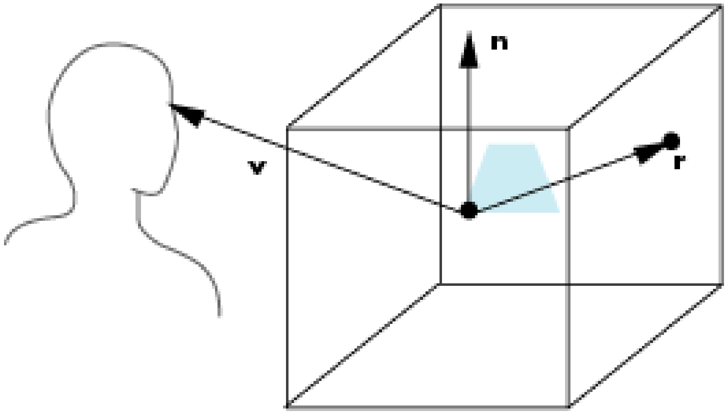
- We can form a cube map texture by defining six 2D texture maps that correspond to the sides of a box
- Supported by OpenGL
- Also supported in GLSL through cubemap sampler
  - vec4 texColor = textureCube(mycube, texcoord);
    - **textureCube >> texture in new GLSL versions**

- Texture coordinates must be 3D



Environment Map

Use the reflection vector to locate texture in cube map



Environment Maps with Shaders

- Environment maps are usually computed in world coordinates which can differ from object coordinates because of the modeling matrix
  - May have to keep track of modeling matrix and pass it to the shader as a uniform variable

- Can also use reflection map or refraction map (for example: to simulate water)

Reflection Map Vertex Shader

```
uniform mat4 Projection, ModelView,
NormalMatrix;
in vec4 vPosition;
in vec4 normal;
out vec3 R;

void main(void)
{
    gl_Position = Projection*ModelView
                  *vPosition;


    vec3 N = normalize(NormalMatrix*normal);


    vec4 eyePos = ModelView*vPosition;


    R = reflect(-eyePos.xyz, N);
}
```

- We can compute the reflection vector at each vertex in our vertex shader and then let the fragment shader interpolate these values over the given primitive.
- However, to compute the reflection vector, we need the normal to each side of the rotating cube.
- Compute normal for each vertex in the application and send them to the vertex shader as a vertex attribute.
- The normals must then be rotated in vertex shader before we can use the reflect function to compute the direction of reflection.
- It computes the reflection vector in eye coordinates as a varying variable.
- We assume that the rotation to the cube is applied in the application and its effect is is incorporated in the model-view matrix. We also assume that the camera location is fixed.

reflect — calculate the reflection direction for an incident vector

Declaration

genType **reflect**(genType *I*, genType *N*);

Parameters

I
Specifies the incident vector.

N
Specifies the normal vector.

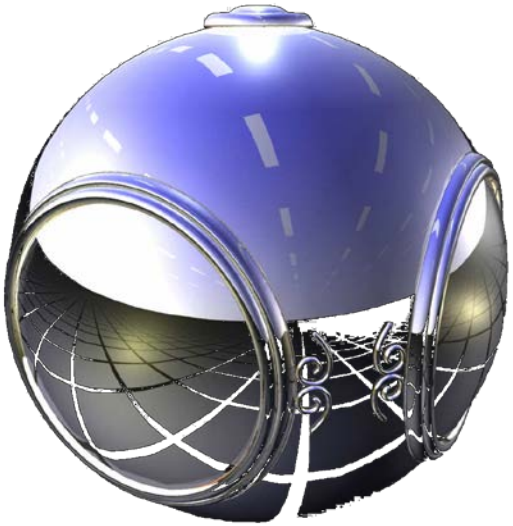
Description

For a given incident vector *I* and surface normal *N* **reflect** returns the reflection direction calculated as *I* - 2.0 * **dot**(*N*, *I*) * *N*.

N should be normalized in order to achieve the desired result.

Reflection Map Fragment Shader

```
uniform samplerCube texMap;
in vec3 R;
out fColor

void main(void)
{
   fColor = textureCube(texMap, R);
}
```



Function names may have
changed in newer GLSL versions.
Check before use!

Toon Shading

- Toon shading is probably the simplest non-photorealistic shader we can write.
- It uses very few colors, usually tones, hence it **changes abruptly from tone to tone**, yet it provides a sense of 3D to the model. The following image shows what we're trying to achieve.

- The tones in the teapot above are selected **based on the angle**, actually on the **cosine of the angle**, <u>between the light's direction and the normal of the surface.</u>
- >> So if we have a normal that is close to the light's direction, then we'll use the brightest tone. As the angle between the normal and the light's direction increases darker tones will be used.
- In other words, the cosine of the angle provides an intensity for the tone.

Toon Shading Vertex Shader

```
uniform mat4 ModelView, Projection, NormalMatrix;
uniform vec4 LightPosition;
in vec4 vPosition, normal;
out vec3 normal, lightDir;

void main()
{
    lightDir = normalize(LightPosition.xyz);
    normal = normalize((NormalMatrix*normal).xyz);

    gl_Position = Projection*ModelView*vPosition;
}
```

Toon Shading Fragment Shader

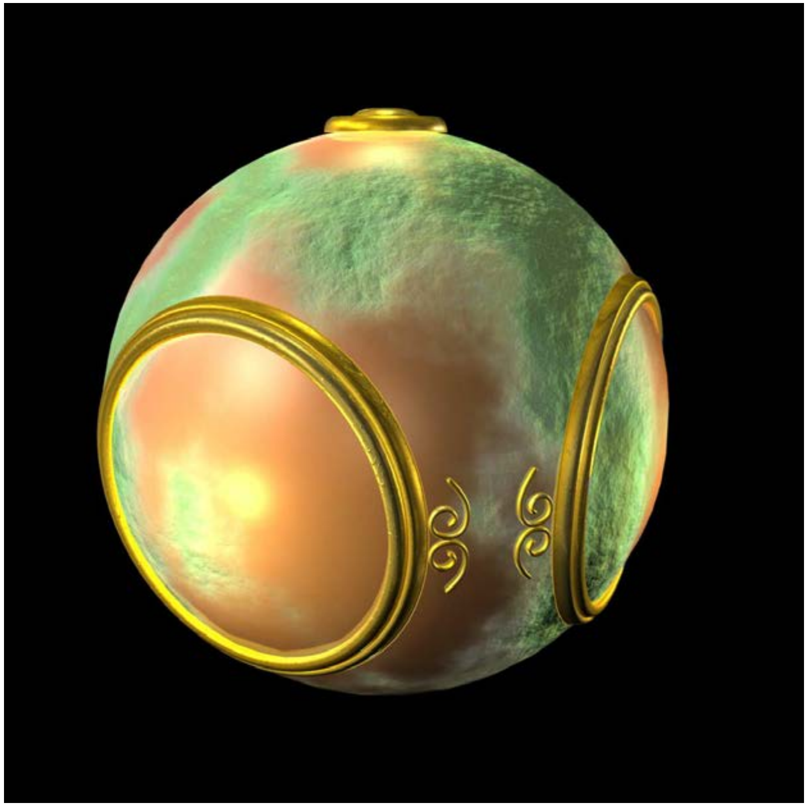
```
in vec3 normal, lightDir;
out vec4 fColor;

void main() {
    float intensity;
    vec3 n;
    vec4 color;
    n = normalize(normal);
    intensity = max(dot(lightDir, n), 0.0);

    if (intensity > 0.98)
       color = vec4(0.8, 0.8, 0.8, 1.0);
    else if (intensity > 0.5)
       color = vec4(0.4, 0.4, 0.8, 1.0);
    else if (intensity > 0.25)
       color = vec4(0.2, 0.2, 0.4, 1.0);
    else
       color = vec4(0.1, 0.1, 0.1, 1.0);

    fColor = color;
}
```

Silhouette?

- A simple way is to find the points where $|v \cdot n|$ is small, i.e., where the view vector is almost perpendicular to the surface normal

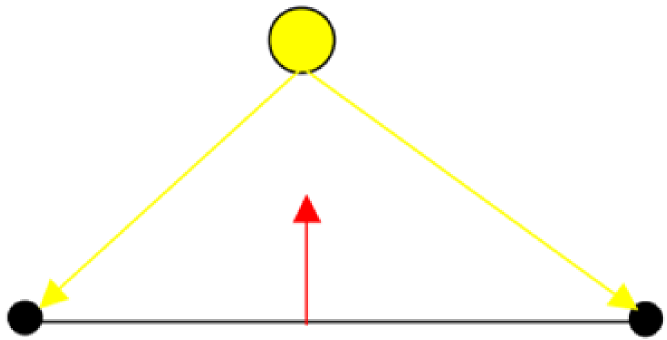
Bump Mapping

- Solves flatness problem of texture mapping
- Perturb normal for each fragment
- Store perturbation as textures



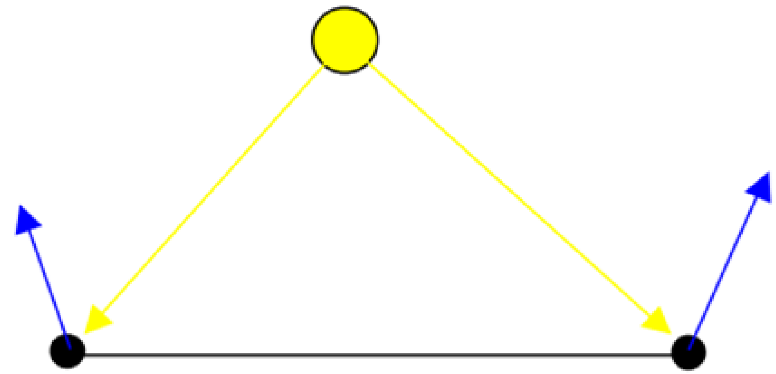
Vertex: Light: Normal:

Flat shading



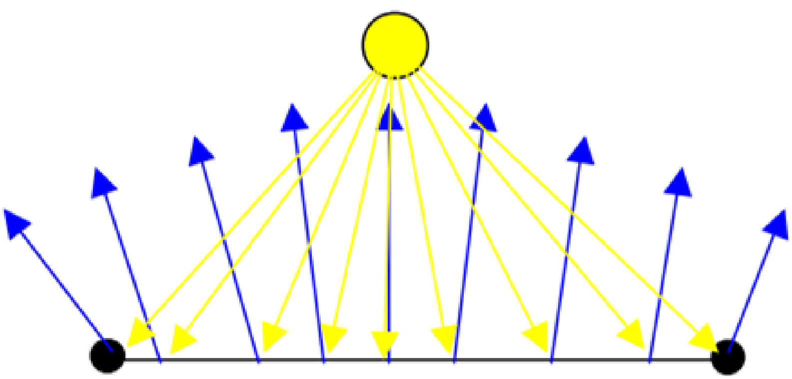
Only the first normal of the triangle is used to compute lighting in the entire triangle.

Goraud shading



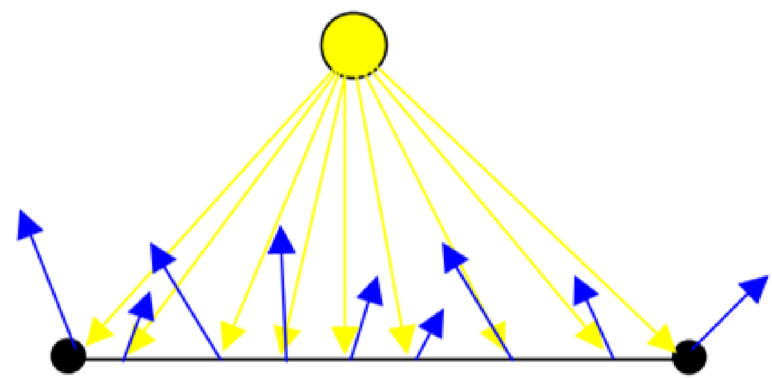
The light intensity is computed at each vertex and interpolated across the surface.

Phong shading



Normals are interpolated across the surface, and the light is computed at each fragment.

Bump mapping



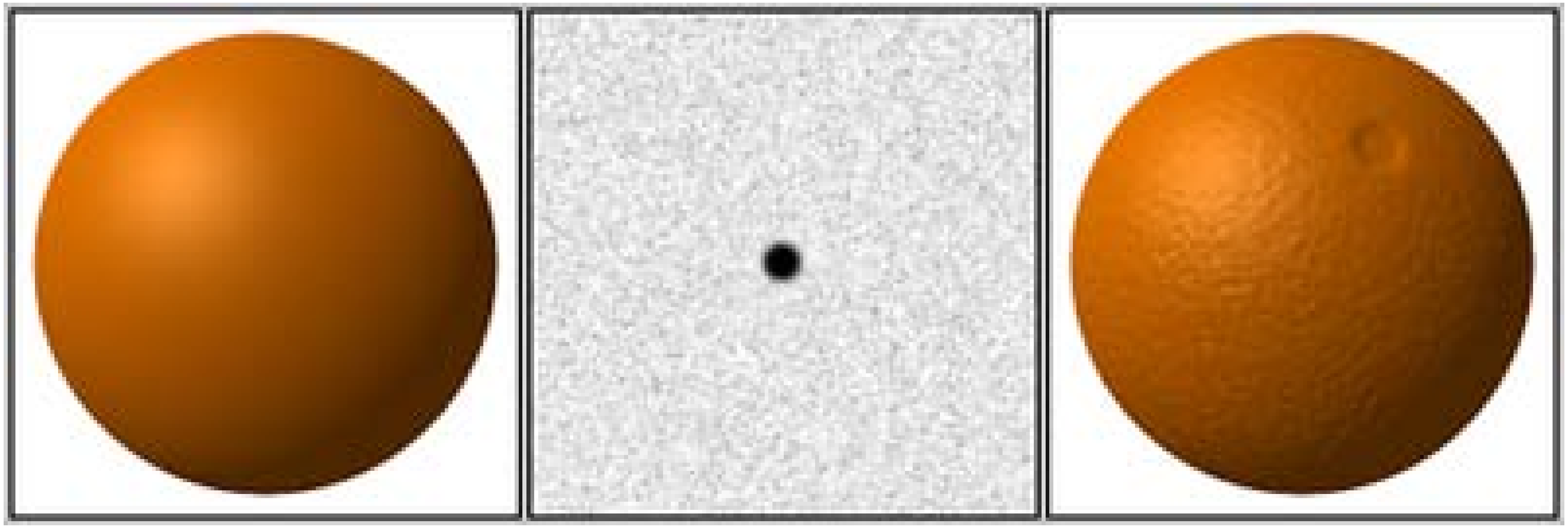
Normals are stored in a bumpmap texture, and used instead of Phong normals.

Modeling an Orange

- Consider modeling an orange

- Texture map a photo of an orange onto a 3D surface
  - Captures dimples
  - Will not be correct if we move viewer or light
  - We have shades of dimples rather than their correct orientation

- Ideally we need to perturb normals across the surface of the object and compute a new color at each interior point

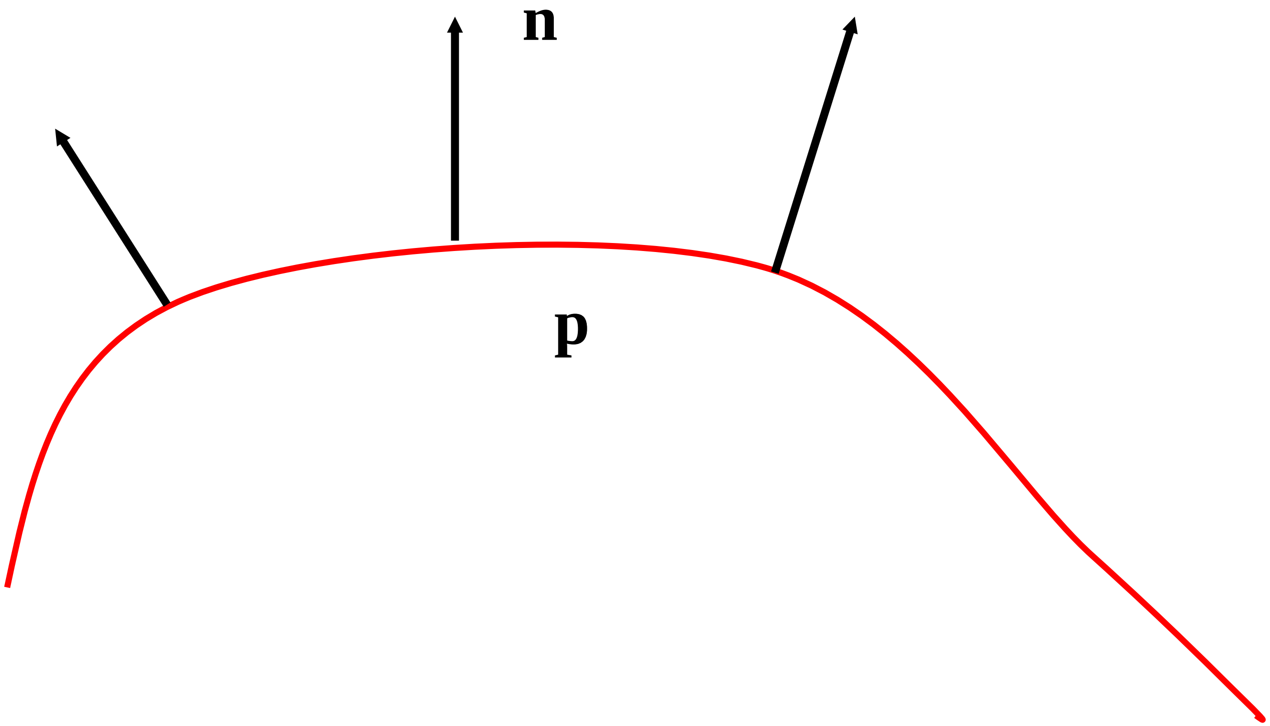


- A sphere without bump mapping (left).
- A bump map to be applied to the sphere (middle).
- The sphere with the bump map applied (right) appears to have a mottled surface resembling an orange.

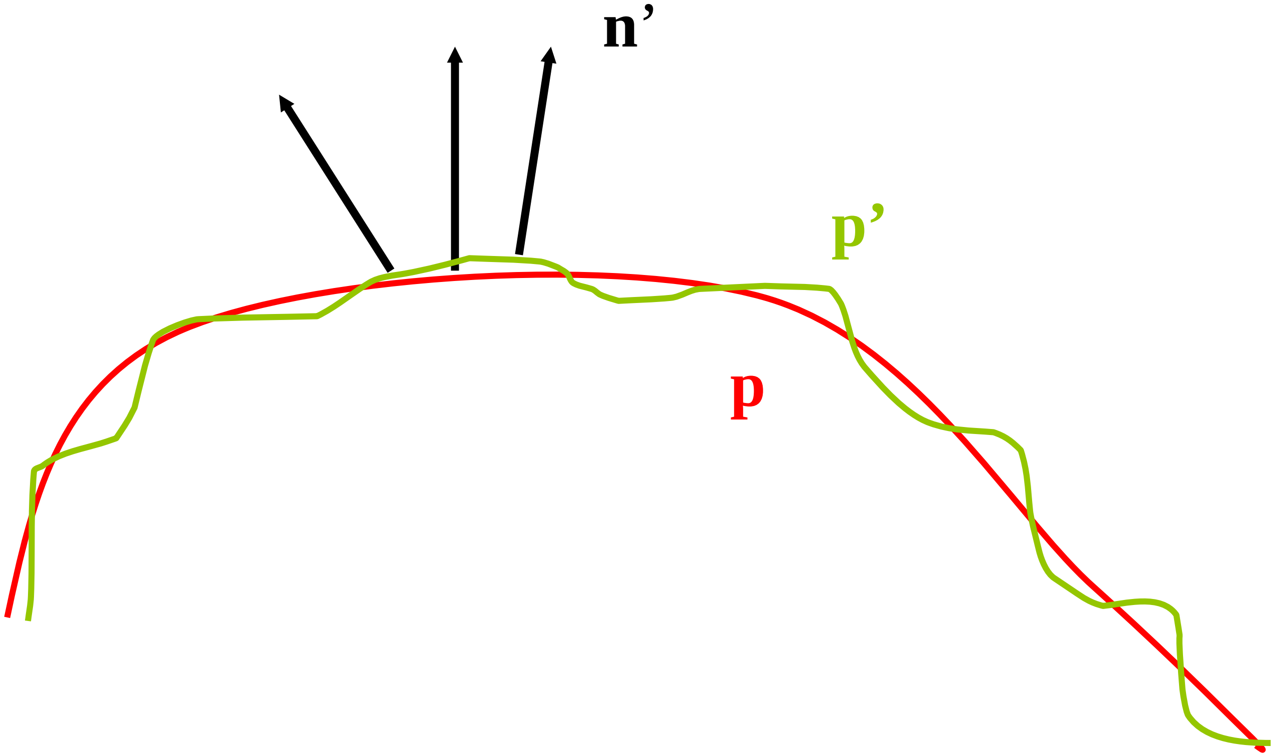
Bump maps achieve this effect by changing how an illuminated surface reacts to light without actually modifying the size or shape of the surface.

Bump Mapping (Blinn)

- Consider a smooth surface



We want to realize this **rougher** look with Bump Mapping



Equations

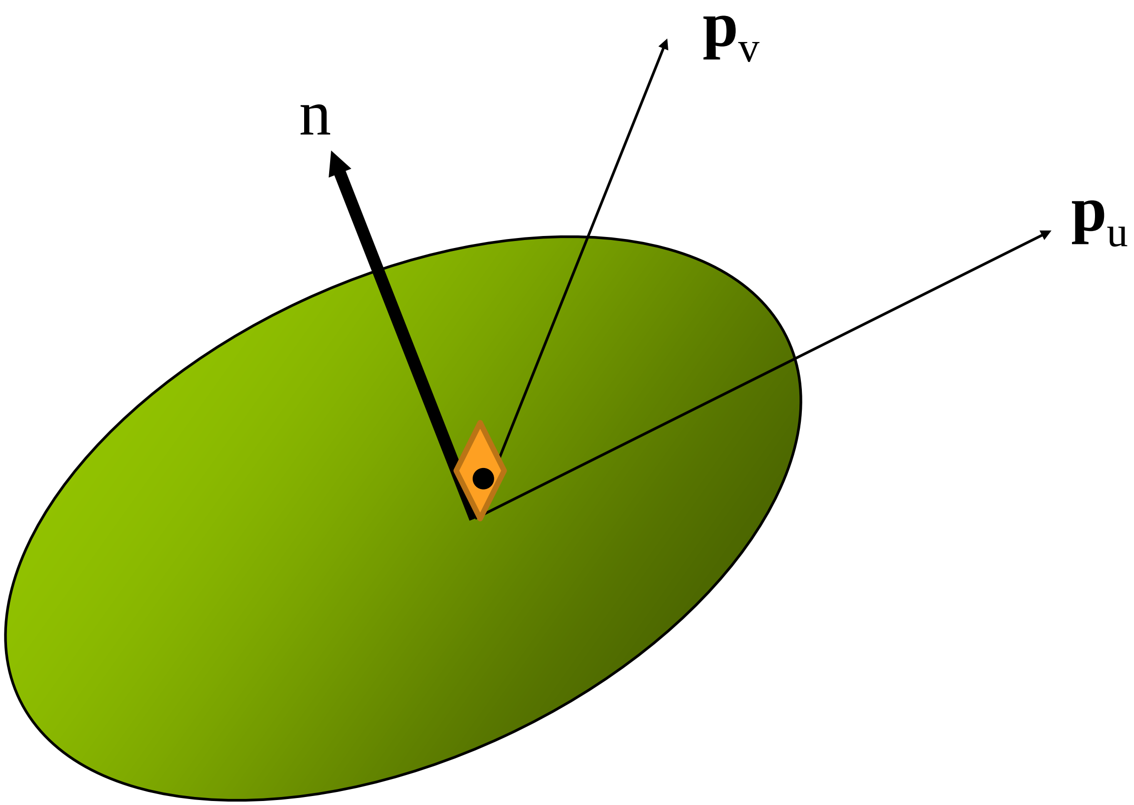
$$\mathbf{p}(u,v) = [x(u,v), y(u,v), z(u,v)]^T$$

$$\mathbf{p}_u = [\, \partial x/\partial u, \partial y/\partial u, \partial z/\partial u]^T$$

$$\mathbf{p}_v = [\, \partial x/\partial v, \partial y/\partial v, \partial z/\partial v]^T$$

$$\mathbf{n} = (\mathbf{p}_u \times \mathbf{p}_v) \,/\, |\,\mathbf{p}_u \times \mathbf{p}_v\,|$$

Tangent Plane



Displacement Function

$$\mathbf{p}' = \mathbf{p} + d(u,v)\,\mathbf{n}$$

$d(u,v)$ is the bump (displacement) function

$|d(u,v)| \ll 1$

Perturbed Normal

$$\mathbf{n'} = \mathbf{p'}_u \times \mathbf{p'}_v$$

$$\mathbf{p'}_u = \mathbf{p}_u + (\partial d/\partial u)\mathbf{n} + d(u,v)\mathbf{n}_u$$

$$\mathbf{p'}_v = \mathbf{p}_v + (\partial d/\partial v)\mathbf{n} + d(u,v)\mathbf{n}_v$$

If d (displacement) is small, we can neglect last term

Approximating the Normal

$$\mathbf{n'} = \mathbf{p'}_u \times \mathbf{p'}_v$$

$$\approx \mathbf{n} + (\partial d/\partial u)\mathbf{n} \times \mathbf{p}_v + (\partial d/\partial v)\mathbf{n} \times \mathbf{p}_u$$

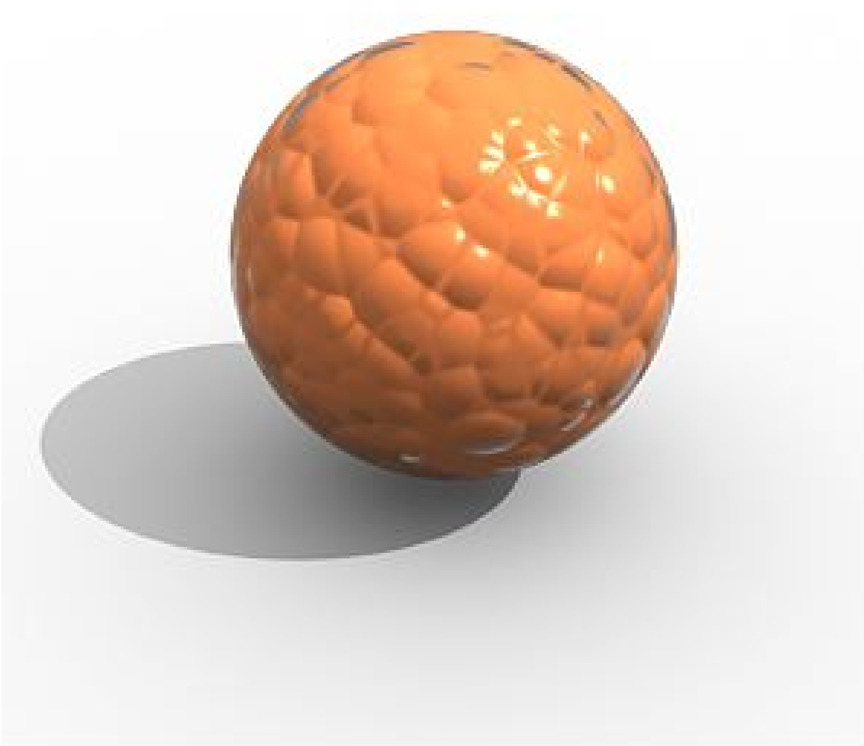
- The vectors $\mathbf{n} \times \mathbf{p}_v$ and $\mathbf{n} \times \mathbf{p}_u$ lie in the tangent plane
- Hence the normal is displaced in the tangent plane
- Must precompute the arrays $\partial d/ \partial u$ and $\partial d/ \partial v$
- Finally, we perturb the normal during shading

Image Processing

- Suppose that we start with a function (bump map) $d(u,v)$
- We can sample it to form a lookup table $D=[\ d_{ij}\ ]$
- Then $\partial d/\partial u \approx [\ d_{ij} - d_{i-1,j}\ ]$

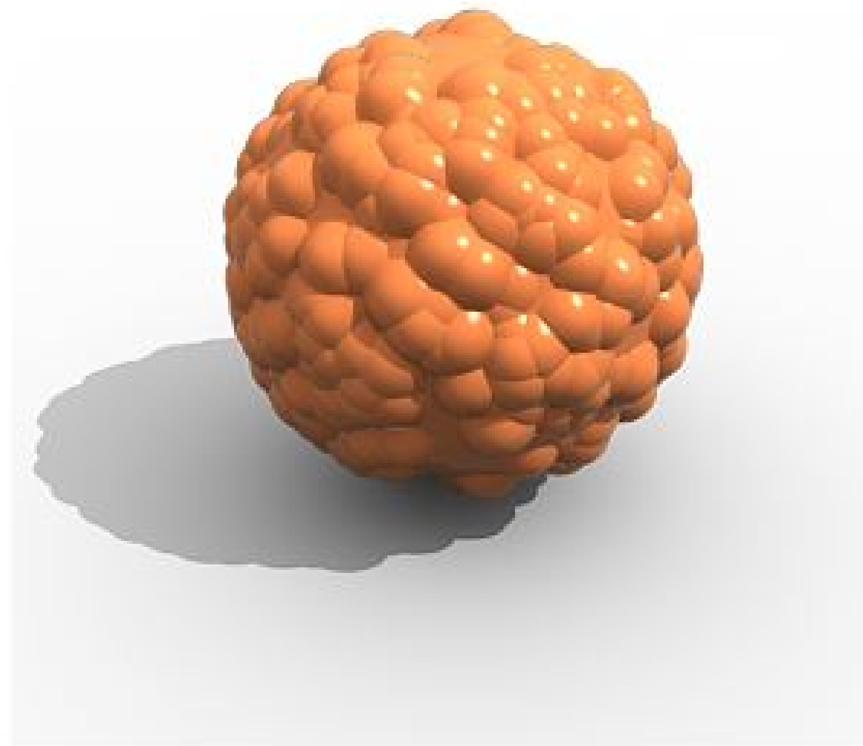
  and $\partial d/\partial v \approx [\ d_{ij} - d_{i,j-1}\ ]$

- **Embossing**: multipass approach using floating point buffer



Bump mapping

- perturbs normals according to value range in image map
- alters shading calculations to give the illusion of variations in surface geometry



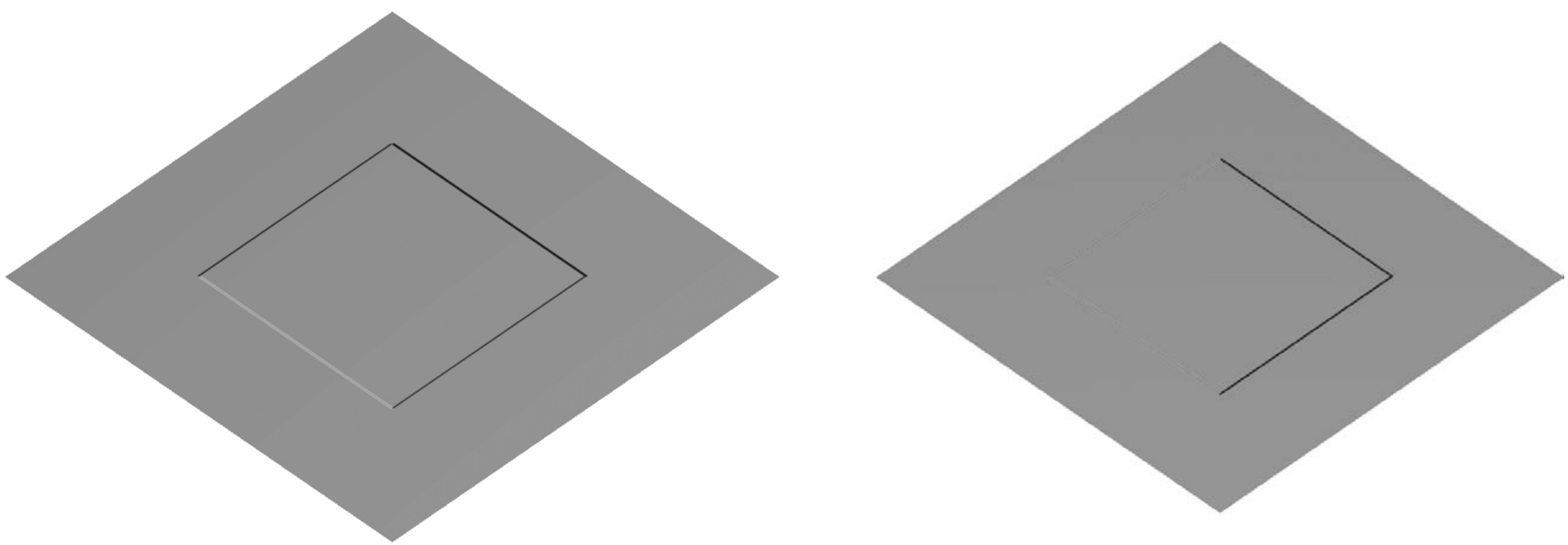
Displacement mapping

- displaces and retessellates geometry along normal (perpendicular to surface) according to value range in image map

Bump Mapping Example

http://www.cs.unm.edu/~angel/WebGL/7E/07/bumpMap.html

Single Polygon and a Rotating Light Source



How to do this?

- The problem is that we want to apply the perturbation at all points on the surface
- Cannot solve by vertex lighting (unless polygons are very small)
- >> Really want to apply to every fragment
  - Couldn’t do that in fixed function pipeline
- But can do using programmable pipeline with a fragment shader

Resources

- http://www.cs.unm.edu/~angel/WebGL/7E/07/
- http://www.webglplayground.net/
- http://glslsandbox.com/