

# Week 9

Vertex Normal and Light Sources	▼
Material Properties	▼
Phong Shading	▼
Gouraud Shading Code	▼
Phong Shading Code	▼
Mapping Techniques	▼
Backwards Texture Mapping	▼
Area to Area Mapping	▼

# Vertex Normal and Light Sources

Wednesday, November 20, 2019 3:19 PM

**Shading** is the *computation of reflected light* at every point

To compute the shading values, we need to know:

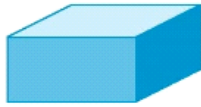
- Normal vectors (denoted by  $\mathbf{n}$ )
- Material properties
- Light vectors (denoted by  $\mathbf{l}$ )

The cosine terms in lighting calculations (see the previous lecture) can be computed using dot product.

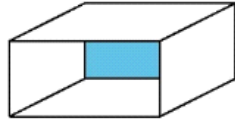
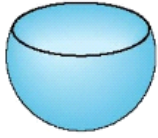
Using unit length vectors simplifies calculation.

GLSL has a **built-in normalization function** for normalizing vectors to unit length.

## Front and Back Faces



back faces not visible



back faces visible

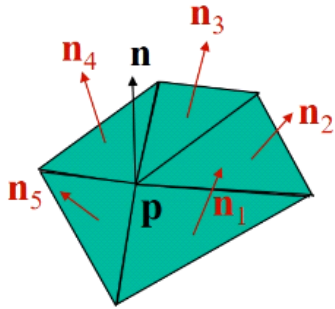
Every surface has a front and back face

For many objects, we never see the back face so we don't care how or if it is rendered (e.g. Closed spheres/prisms)

If it matters, we can handle that in the shader

## Vertex Normals

Complex surfaces are often represented by a triangular mesh.

	<p>Consider 5 adjacent triangles sharing a common vertex <math>p</math>.</p> <ol style="list-style-type: none"><li>1. We can compute the normal <math>\mathbf{n}_1 \dots \mathbf{n}_5</math> of the 5 triangles</li><li>2. We can average these normals to get the normal <math>\mathbf{n}</math> at <math>p</math>. We then divide by the magnitude of the resulting vector to get a unit vector.</li></ol> $\mathbf{n} = \sum_{i=1}^5 \mathbf{n}_i \quad \text{then} \quad \mathbf{n} \leftarrow \mathbf{n} / \ \mathbf{n}\ $ <ol style="list-style-type: none"><li>3. This allows us to have normals defined at vertices</li><li>4. We call <math>\mathbf{n}</math> the <b>vertex normal</b> at <math>p</math>. The normal points away from "front" faces</li></ol>
---	--

## Moving Light Sources

Light sources are geometric objects whose positions or directions are *affected by the model-view matrix*

Depending on where we place the position/direction setting function, we can

- Move the light source(s) with the object(s)
- Fix the object(s) and move the light source(s)
- Fix the light source(s) and move the object(s)
- Move the light source(s) and object(s) independently

In interactive graphics, users should be able to do all of the above.

## Specifying a Point Light Source

For each light source, we can **set the RGBA values** for the diffuse, specular, and ambient components, and for the **light source position**:

```
vec4 light0_pos = vec4(1.0, 2.0, 3.0, 1.0);  
vec4 diffuse0 = vec4(1.0, 0.0, 0.0, 1.0);  
vec4 ambient0 = vec4(1.0, 0.0, 0.0, 1.0);  
vec4 specular0 = vec4(1.0, 0.0, 0.0, 1.0);
```

Red (R)      Green (G)      Blue (B)      Opacity (A or alpha)

This will produce red light

## Distance and Direction

The light source color is specified in RGBA

The light source position is given in homogeneous coordinates:

- If  $w = 1.0$ , we are specifying a finite location (interpreted as a point)
- If  $w = 0.0$ , we are specifying a parallel source with the given direction vector

Recall from lecture 15, we can have a distance-attenuation coefficient so objects further from the light source receive less light.

The distance-attenuation coefficient is usually inversely proportional to the square of the distance  $d$ :  $1/(a + bd + cd^2)$ , where  $a, b$  and  $c$  are user-defined constants.

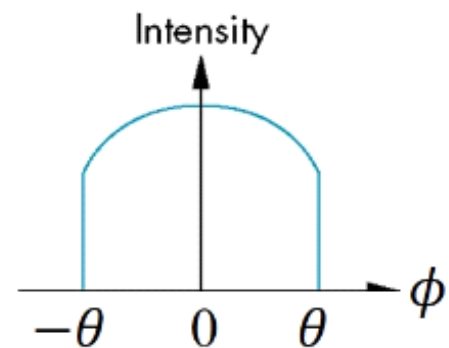
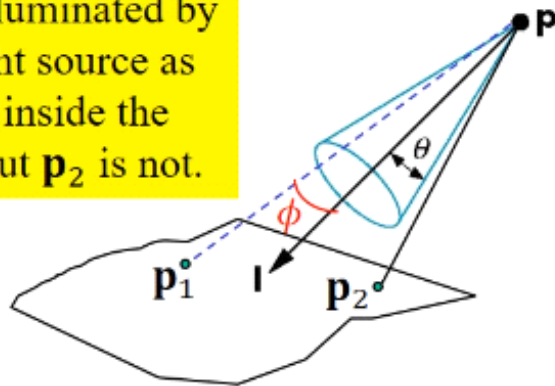


## Spotlights

Spotlights are *derived from point sources*, and have:

- A **distance** (like points)
- A **colour** (like points)
- A **direction**
- A **cutoff value** ( $\theta$ ) (The light always stays between  $-\theta$  and  $\theta$ )
- An **attenuation function**, usually defined by  $\cos^e \phi$ 
  - *Cos phi raised to power (e)*
  - *Large  $e$  values  $\rightarrow$  attenuate faster*
  - *Small  $e$  values  $\rightarrow$  attenuate slower*

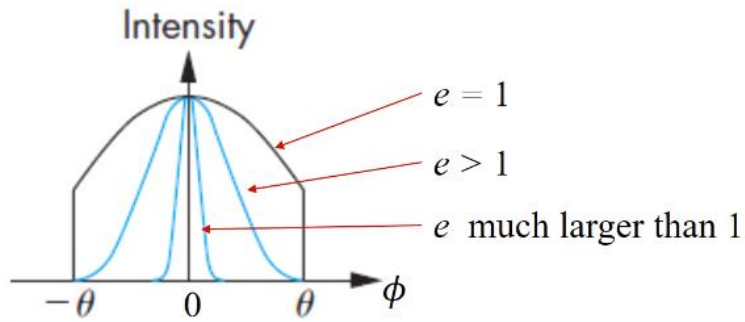
$p_1$  is illuminated by the light source as it falls inside the cone but  $p_2$  is not.



If  $\theta = 180^\circ$  then the spotlight becomes a point source

## Light Intensity

The exponent term  $\alpha$  in  $\cos^e \phi$  determines how rapidly the light intensity drops off.



Note: the term  $e$  here is only a coefficient. It is not equal to  $\exp(1)$ . It is a poor choice of symbol from Angel & Shreiner.

When  $e=1$ , we get slow attenuation

When  $e$  is large, we get dramatic decreases in intensity

### Material properties should match the terms in the light source model

- Material properties are specified via the ambient, diffuse, and specular reflectivity coefficients ( $k_a$ ,  $k_d$  and  $k_s$ )
- Material properties are specified as RGBA values.
- The **A** value gives opacity, and can be used to make the surface translucent.
  - The default is that all surfaces are opaque.

```
vec4 ambient = vec4(0.2, 0.2, 0.2, 1.0);  
vec4 diffuse = vec4(0.8, 0.8, 0.0, 1.0);  
vec4 specular = vec4(1.0, 1.0, 1.0, 1.0);  
GLfloat shine = 100.0;
```

R G B A (opacity)

This is a yellow (not fully saturated though) object.  
Due to shine value of 100, it will be smooth and shiny

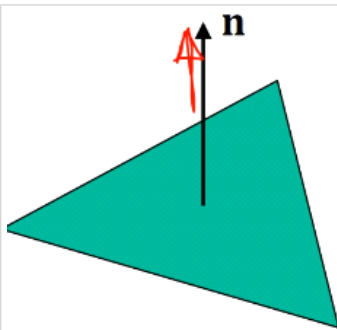
## Polygonal Shading Methods

There are 3 methods for shading a polygonal mesh:

1. Flat shading
2. Smooth shading
  - Gouraud shading
  - Phong shading

Recall that OpenGL handles only triangles. So the terms "polygon" and "polygonal" from here onward should be interpreted as "triangle" and "triangular".

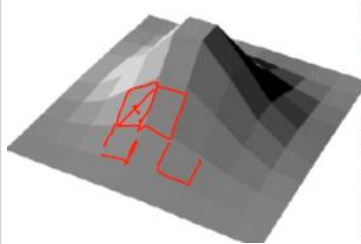
## Flat/Constant Shading



Recall that to compute the shading value at a point, we need to know the light vector  $l$ , the normal vector  $n$ , and the view vector  $v$ .

As we move from one point to the next point inside the polygon,  $l$  and  $v$  should vary, but in the flat shading method, they **are assumed to be constant**.

- A single shading calculation is performed for each polygon.
- The entire polygon has a single shading value.



An example of rendering a polygonal mesh using flat shading

**Advantage:** Computationally cheap

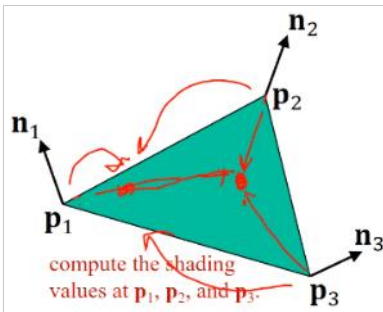
**Disadvantage:** Boundary edges of polygons may show up in the rendered output. , Edges are distinct, not smoothed over

This shading method is suitable when the viewer and/or light source is far away from the polygon.

In OpenGL, we specify flat shading as follows:

```
glShadeModel(GL_FLAT);
```

## Smooth/Gouraud Shading



In the Gouraud shading method:

- The normal vector at each vertex of the polygon is firstly computed
- The shading value at each vertex is then computed using the light vector  $l$ , the normal vector  $n$ , and the view vector  $v$  at that vertex
- Shading values of interior points of the polygon are obtained by interpolating the shading values at the vertices (we combine the value of each vertex, taking into account the distance from each vertex)

As one shading calculation is required for each vertex, it is also known as per-vertex shading.

**Advantage:** Compared to flat shading, the smooth shading method gives much better, smoother rendering results

**Disadvantage:** Compared to flat shading, smooth shading is more computationally intensive;

Smooth shading (or Gouraud shading) is the **default implementation** in OpenGL (it will be set if you do not set anything)

You can also explicitly set the shading mode as follows:

```
glShadeMode(GL_SMOOTH);
```

## Phong Shading

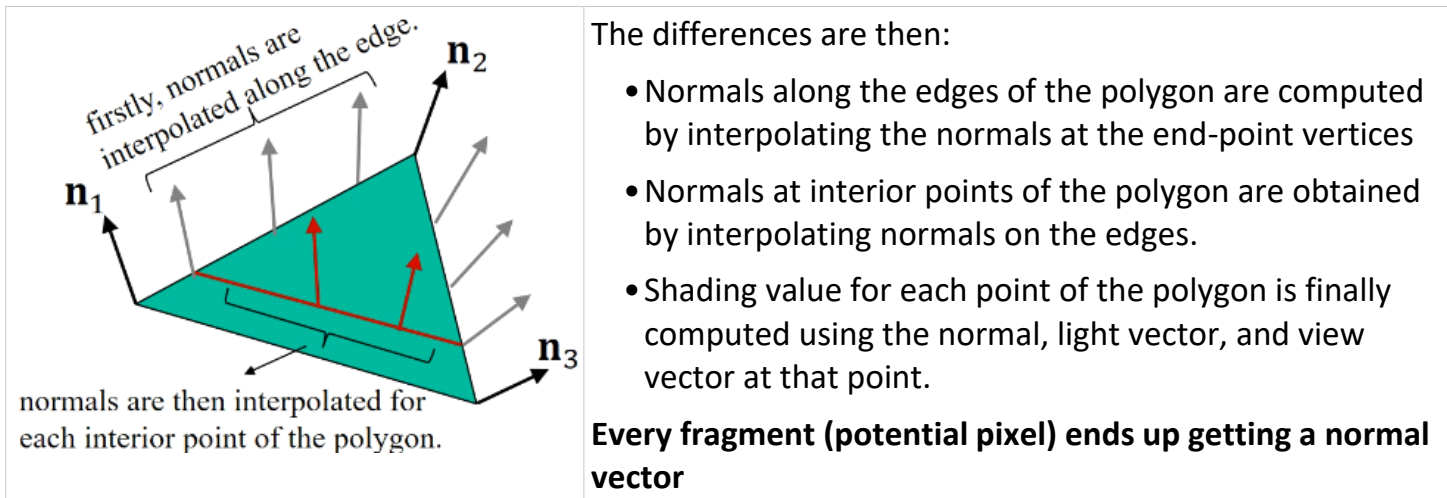
Although **Gouraud shading** is smooth, the appearance of **Mach bands** (exaggerated contrast between edges) may still be found, especially at specular highlight regions of the scene

Phong proposed:

1. Instead of interpolating vertex intensities to get the intensities of the interior of each polygon, **we interpolate the normals of the vertices of each polygon** to get the normal across the interior of each polygon.
2. *We then use the interpolated normal at each fragment to compute the intensity of the fragment.*

## Process

Similar to Gouraud shading, the Phong shading method firstly computes the normal vector at each vertex of the polygon.





## Analysis

As 1 shading calculation is performed for each fragment, Phong shading is also known as **per-fragment** shading/lighting.

Recall that a fragment is a potential pixel which carries additional information such as depth value, colour value, opacity, etc.

**Advantage:** Very good rendering results, especially at the highlight regions of highly specular (shiny) surfaces

**Disadvantage:** Too computationally expensive. Only done in off-line processing. Not suitable for interactive graphics. However, it is reported that the latest graphics cards perform Phong shading in real-time.

## Comparison



If the polygon mesh approximates surfaces with a high curvatures, Phong shading may look smooth while Gouraud shading may show edges.

Both need data structures to represent meshes so we can obtain vertex normals.

Note the difference in the highlight region of the teapot. Gouraud shading gives a flat impression in the region.

# Gouraud Shading Code

Wednesday, November 20, 2019 3:19 PM

# Vertex Shader

Wednesday, November 20, 2019 3:19 PM

## Declarations

The vertex shade/colour is computed in the vertex shader.

This is Gouraud shading or per-vertex shading.

<pre>// vertex shader in vec4 vPosition; in vec3 vNormal; out vec4 color; //vertex shade  // light and material properties uniform vec4 AmbientProduct, DiffuseProduct, SpecularProduct; uniform mat4 ModelView; uniform mat4 Projection; uniform vec4 LightPosition; uniform float Shininess;</pre>	<p><b>In variables</b> = Vertex position and normal</p> <p><b>Out variable</b> = Color = <i>passed down the pipeline where the rasterizer carries out the interpolation on the color</i></p> <p>All of the <u>light material properties</u> have been <i>computed in the application</i> and passed to the vertex shader as uniform variables.</p>
--	--

## Code 1

```
void main()
{
    // Transform vertex position into eye coordinates
    vec3 pos = (ModelView * vPosition).xyz;

    vec3 L = normalize( LightPosition.xyz - pos );
    vec3 V = normalize( -pos ); // since the camera is at 0,0,0
    vec3 H = normalize( L + V ); // the half-way vector

    // Transform vertex normal into eye coordinates
    vec3 N = normalize( ModelView*vec4(vNormal, 0.0) ).xyz;
```

We extract the xyz components of the model view matrix multiplied by the vertex position

Normalize = Ensures that the vectors are **unit vectors**. Built-in function that can be used in the vertex and fragment shaders.

L = Light source SUBTRACT vertex position  
V = 0,0,0 position SUBTRACT vertex position  
H = L + V

## Code 2

// Compute terms in the illumination equation

```
vec4 ambient = AmbientProduct;
```

```
float Kd = max( dot(L, N), 0.0 );
```

```
vec4 diffuse = Kd*DiffuseProduct;
```

```
float Ks = pow( max(dot(N, H), 0.0), Shininess );
```

```
vec4 specular = Ks * SpecularProduct;
```

```
if( dot(L, N) < 0.0 ) specular = vec4(0.0, 0.0, 0.0, 1.0);
```

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

```
color = ambient + diffuse + specular;
```

```
color.a = 1.0;
```

```
}
```

Recall that the total shading is (see previous lecture):

$$I = k_d I_d (\mathbf{l} \cdot \mathbf{n}) + k_s I_s (\mathbf{n} \cdot \mathbf{h})^\beta + k_a I_a$$

Kd = Maximum of dot product and 0 = Cannot be less than 0

We give color an alpha value of 1

## Fragment Shader

<pre>in vec4 color;  void main() {     gl_FragColor = color; }</pre>	<p>All the hard work has been done by the application code and the vertex shader.</p> <p>The fragment shader simply takes the interpolated colour and assigns it to the fragment.</p>
--	---



# Phong Shading Code

Wednesday, November 20, 2019 3:19 PM

# Vertex Shader

Wednesday, November 20, 2019 3:19 PM

## Declarations

The **fragment colour is computed in the fragment shader** using the interpolated normal, light, and view vectors.

This is Phong shading or **per-fragment shading**.

<pre>// vertex shader in vec4 vPosition; in vec3 vNormal;  // output values that will be interpolated per-fragment out vec3 fN; out vec3 fV; out vec3 fL;  uniform mat4 ModelView; uniform vec4 LightPosition; uniform mat4 Projection;</pre>	<ol style="list-style-type: none"><li>1. The <i>vertex shader</i> calculates the <b>out</b> fN, fV, and fL variables<ul style="list-style-type: none"><li>• fN = Normal vector</li><li>• fV = Viewer location</li><li>• fL = Light location</li></ul></li><li>2. The vertex shader then passes these down the pipeline, for the rasterizer to interpolate</li></ol>
---	---

## Code

```
void main()
{
    fN = (ModelView * vNormal).xyz;
    fV = - (ModelView * vPosition).xyz;
    fL = LightPosition.xyz - (ModelView * vPosition).xyz;

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

We do not normalize the vectors

We extract vec3's only

fV is negative because it is being subtracted from 0,0,0 Camera

# Fragment Shader

Friday, 10 April 2020 1:54 PM

## Declarations

```
// per-fragment interpolated values from the vertex shader
```

```
in vec3 fN;
```

```
in vec3 fL;
```

```
in vec3 fV;
```

```
uniform vec4 AmbientProduct, DiffuseProduct, SpecularProduct;
```

```
uniform mat4 ModelView;
```

```
uniform vec4 LightPosition;
```

```
uniform float Shininess;
```

We receive three variables from the vertex shader

We receive the uniform variables from the application

## Code 1

```
void main()
{
    // Normalize the input lighting vectors

    vec3 N = normalize(fN);
    vec3 V = normalize(fV);
    vec3 L = normalize(fL);

    vec3 H = normalize( L + V );
    vec4 ambient = AmbientProduct;
```

We normalize the vectors

## Code 2

```
float Kd = max(dot(L, N), 0.0);  
vec4 diffuse = Kd*DiffuseProduct;
```

```
float Ks = pow(max(dot(N, H), 0.0), Shininess);  
vec4 specular = Ks*SpecularProduct;
```

```
// discard the specular highlight if the light's behind the vertex  
if( dot(L, N) < 0.0 )  
    specular = vec4(0.0, 0.0, 0.0, 1.0);
```

```
gl_FragColor = ambient + diffuse + specular;  
gl_FragColor.a = 1.0;  
}
```

Negative values become zero when max() is used

Recall that the total shading is (see previous lecture):

$$I = k_d I_d (\mathbf{l} \cdot \mathbf{n}) + k_s I_s (\mathbf{n} \cdot \mathbf{h})^\beta + k_a I_a$$

First part of equation = first two lines

Second part of equation = second two lines

Beta is shininess

We correct for the specular term being negative

We add up light terms and set alpha value to 1.0



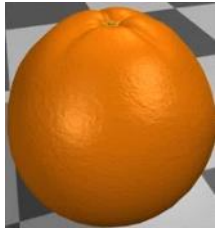
# Mapping Techniques

## Limits of Geometric Modelling

Although graphics cards can render over 10 million triangles per second, that number is insufficient for many phenomena

- Clouds
- Grass
- Terrain
- Skin

## Modelling Example



Consider the problem of modelling an orange

We can model it as an orange-colored sphere, but this would be unrealistic. Unfortunately, the rendering would be too regular to look like an orange.



Alternatively, we can replace the sphere with a more complex shape

Unfortunately, the rendering still does not capture some fine surface characteristics (e.g. small dimples on the orange)

Increasing the no. of polygons to capture the surface details would overwhelm the pipeline.

## The Solution

We can use the **process of texture mapping**: Take a picture of a real orange and *"paste" the image of the orange skin onto a sphere*

However, this still might not be sufficient because the resulting surface will be smooth.

To make the surface look a bit rough, we need to change local shape, i.e., we need **bump mapping**

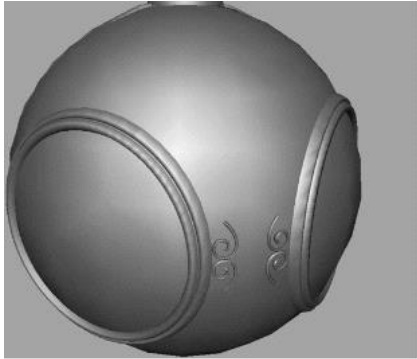
# Three Types

The three types of mapping are:

1. Texture Mapping
2. Environment Mapping
3. Bump Mapping

## Texture Mapping

**Texture Mapping** uses an image/texture map to *influence the colour of a fragment*  
(We paint patterns onto smooth surfaces)



geometric model

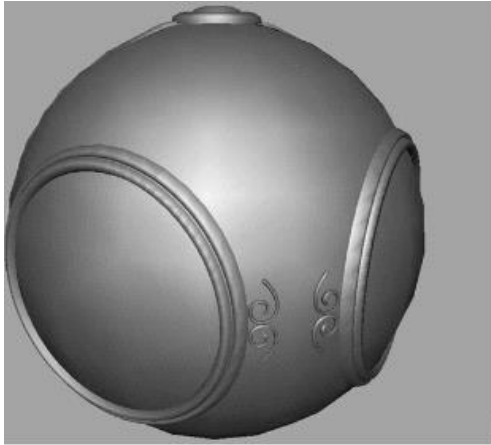


texture mapping output

## Environment Mapping

### Environment/Reflection Mapping

- Uses a picture of the environment as the texture map
- Allows us to *simulate highly specular surfaces* (e.g. mirrors)



geometric model

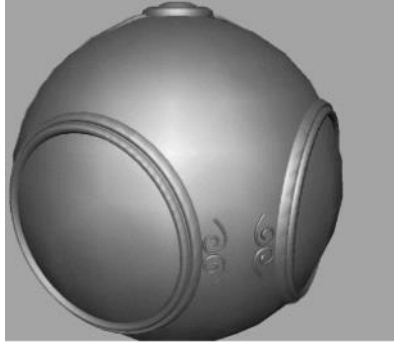


Environment mapping output

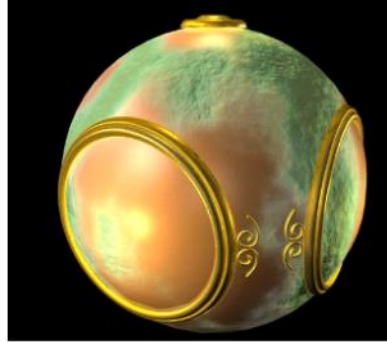
## Bump Mapping

### Bump Mapping

- Adds small distortions to the surface normals before mapping the texture during the rendering process.
- This allows us to simulate small variations in shape, such as the bumps on a real orange.



geometric model



Bump mapping output

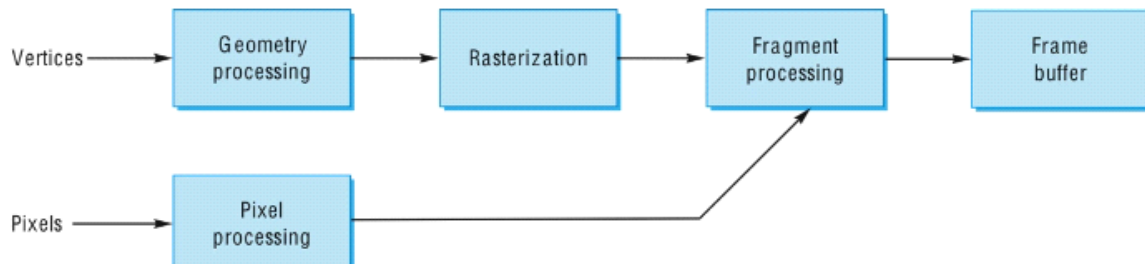


# Backwards Texture Mapping

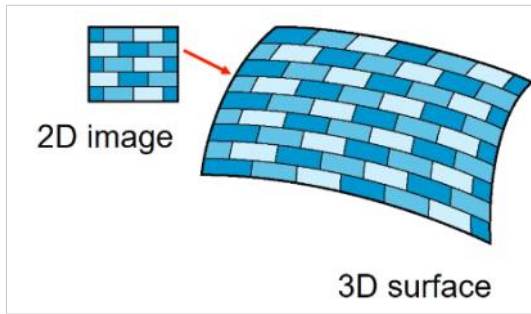
## Mapping in Pipeline

Mapping techniques are implemented at the **end of the rendering pipeline**

- Very efficient because *few triangles make it past the clipper*
- This is good as we have less triangles to map the texture onto



## Process

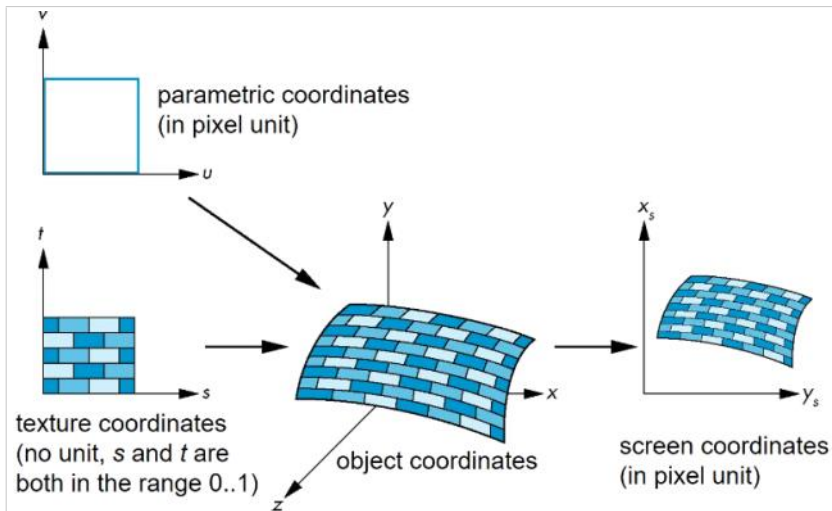


Although the idea of mapping a texture (usually stored as an image in a file) onto a 3D surface is simple, there are 3 or 4 coordinate systems involved.

## Mapping Coordinate Systems

Type	Description
<i>Parametric Coordinates</i>	May be used to model curves and surfaces
<i>Texture Coordinates</i>	Used to identify points in the image to be mapped
<i>Object or World Coordinates</i>	Conceptually, where the mapping takes place
<i>Window or Screen Coordinates</i>	Where the final image is really produced

## Diagram

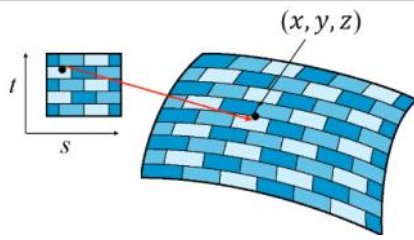


Some have different units (e.g. parametric and screen are in pixels while texture has no units)

We use the parametric and texture coordinates to find out a mapping from a pixel to a point on the 3D surface of the object

Lastly, we map from the object coordinates to the screen coordinates

## Mapping Functions



Consider mapping from texture coordinates to a point on a surface

We need three parametric functions to describe the surface

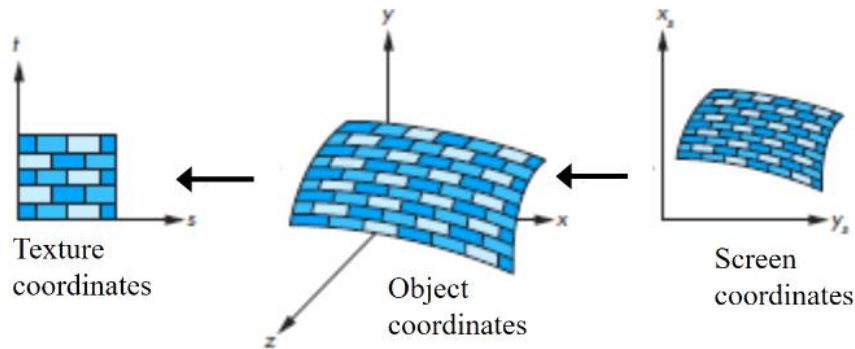
- $x(s, t)$
- $y(s, t)$
- $z(s, t)$

The texture image pixel coordinates are represented by  $(s, t)$ . We want to find out which point on the surface it corresponds to  $(x, y \text{ or } z)$

But we want to go the other way - we want to see what texture a pixel needs in the final image, as other points may be clipped out anyway

## Backward Mapping

We want to map screen coordinates to texture coordinates.



Given a pixel on the screen, we want to find the corresponding 3D point  $(x, y, z)$  on the object and then the  $(s, t)$  coordinates in the texture-map that the pixel corresponds to.

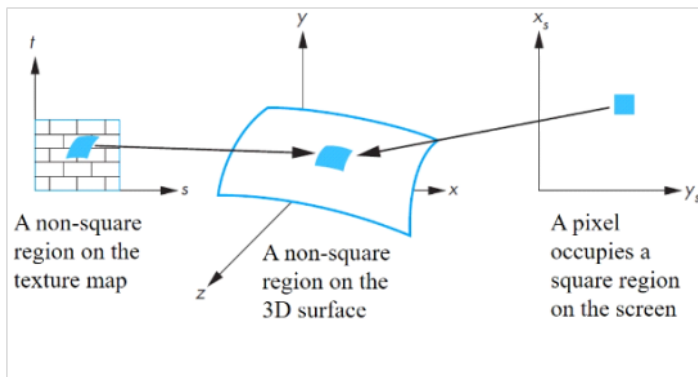
For the mapping from object coordinates  $(x, y, z)$  to texture coordinates  $(s, t)$ , we need a mapping the form:

- $s(x, y, z)$
- $t(x, y, z)$

(given  $x, y$  and  $z$ , what are the values for the parameters  $s$  and  $t$  to index into the texture map)

Such mapping functions are difficult to find in general.

## Area to Area Mapping



Because each pixel corresponds to a small rectangle on the display, when doing the backward mapping, we cannot just map **points to points** in the different coordinate systems.

We need to map **areas to areas** to prevent shapes being warped

In the diagram below, the non-square area of the texture should contribute to the shading of the pixel



The image shows the word "Aefgoy" in a smooth, black, sans-serif font. A thin blue horizontal line is positioned directly beneath the text.

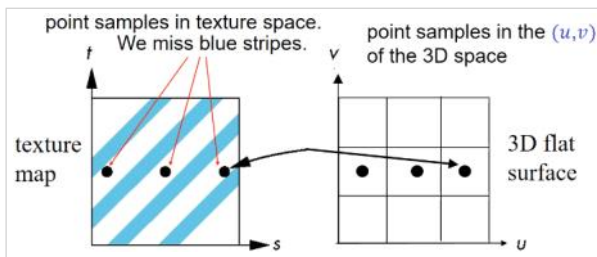
We *need to map areas to areas* as **mapping points to points** can cause **aliasing** even for simple flat surfaces.

The image shows the word "Aefgoy" in a pixelated, black, sans-serif font. A thin blue horizontal line is positioned directly beneath the text.

**Aliasing** refers to the process by which *smooth curves or lines become jagged* because of the resolution of the graphics device or there is insufficient data to represent the curves.

Aliasing occurs in texture mapping too.

## Aliasing and Mapping



Suppose that we have a periodic looking texture and we map points to points. Then even for flat surfaces, we can still encounter the aliasing problem. Especially when the camera zooms out of the scene, this results in each pixel covering a large region of the 3D surface.

Mapping areas to areas means that we assign the **average shading value of the texture region** to the pixel. This helps to reduce the aliasing effect. However, we would still miss the blue stripes when the resolution of both the frame buffer and the texture map drops.

## Two Part Mapping

Backward mapping functions are difficult to find in general...

One solution is to use a **two-part mapping**:

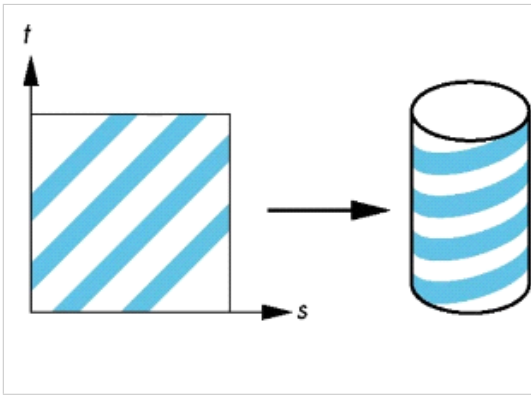
**First Part Mapping =**

Map the texture onto a similar intermediate object

**Second Part Mapping =**

Map the texture from the intermediate object to the actual object.

## First Part = Intermediate Objects



Common intermediate objects:

- Cylinder
- Sphere
- Box

If the intermediate surface is a cylinder, the first-part mapping involves mapping the texture onto the cylinder.

## Cylinder

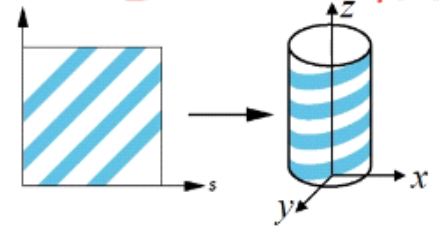
Parametric equation of a cylinder with radius= $r$  and height= $h$ :

$$\circ x = r \cos 2\pi u$$

$$\circ y = r \sin 2\pi u$$

$$\circ z = v/h$$

The parameters are  $u$  and  $v$



Note that  $x$  and  $y$  take values in the range  $-r \cdots r$ ,  $z$  takes values from  $0$  to  $h$ .

Given a 3D point  $(x, y, z)$  on the cylinder, do the following:

- Find out the parameter values  $u$  and  $v$
- Since  $u$  and  $v$  vary from  $0$  to  $1$ , we can simply let

$$s = u$$

$$t = v$$

- retrieve the colour at coordinate  $(s, t)$  from the texture map.

## Sphere

We can also use a sphere as the intermediate object. The parametric equation (with parameters  $u$  and  $v$ ) of a sphere of radius  $r$  is given by:

$$x' = r \cos 2\pi u$$

$$y = r \sin 2\pi u \cos 2\pi v$$

$$z = r \sin 2\pi u \sin 2\pi v$$

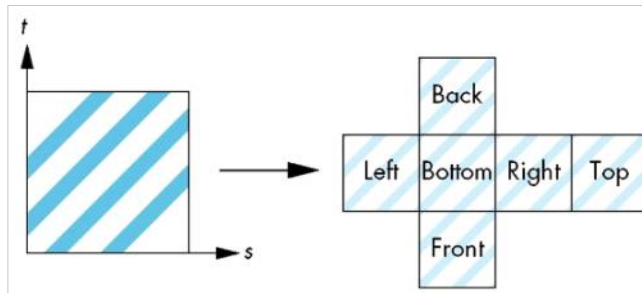
In a similar manner to the cylinder:

- Give  $(x, y, z)$  compute  $u$  and  $v$

- Let  $s = u$  and  $t = v$

Obviously there is distortion. We have to decide where to put the distortion. Spheres are used in environment mapping.

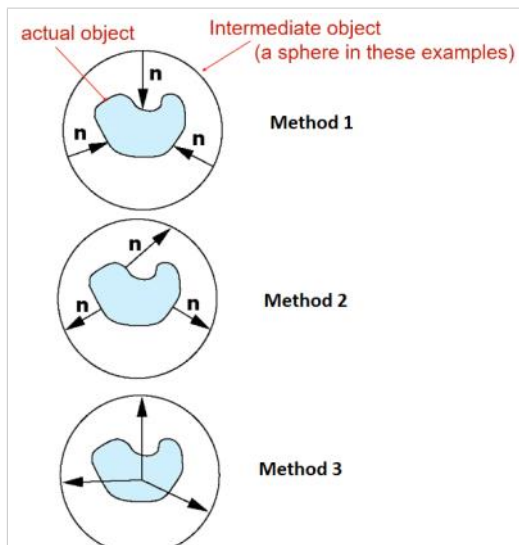
## Box



Easy to use with simple orthographic projection

Boxes are also used as intermediate objects in environment mapping

## Second Part



The second-part mapping is to map the texture from the intermediate object to the actual object.

There are 3 ways to map texture from an intermediate object to the actual object:

1. Normals from intermediate to actual
2. Normals from actual to intermediate
3. Vectors from center of intermediate