Comp 410/510

Computer Graphics

Spring 2023

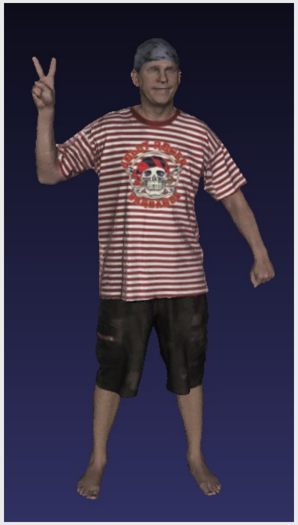
Texture mapping

Texture Mapping - Example

Wireframe

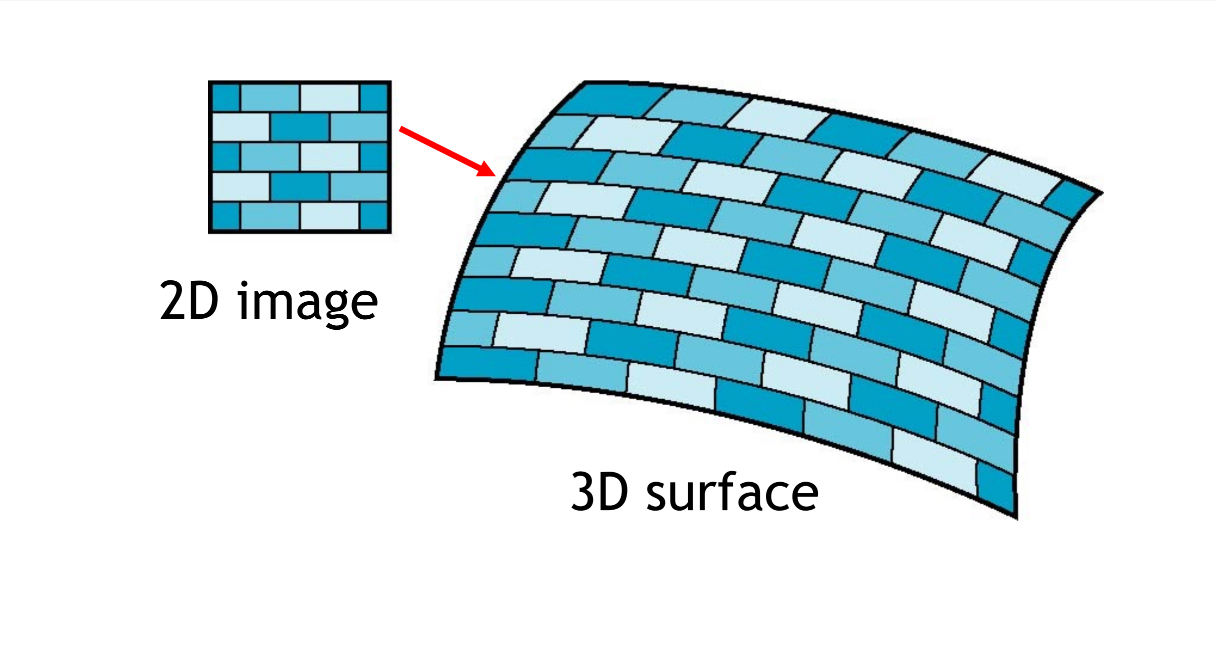
Shading

Texture mapping



Texture mapping: Is it simple?

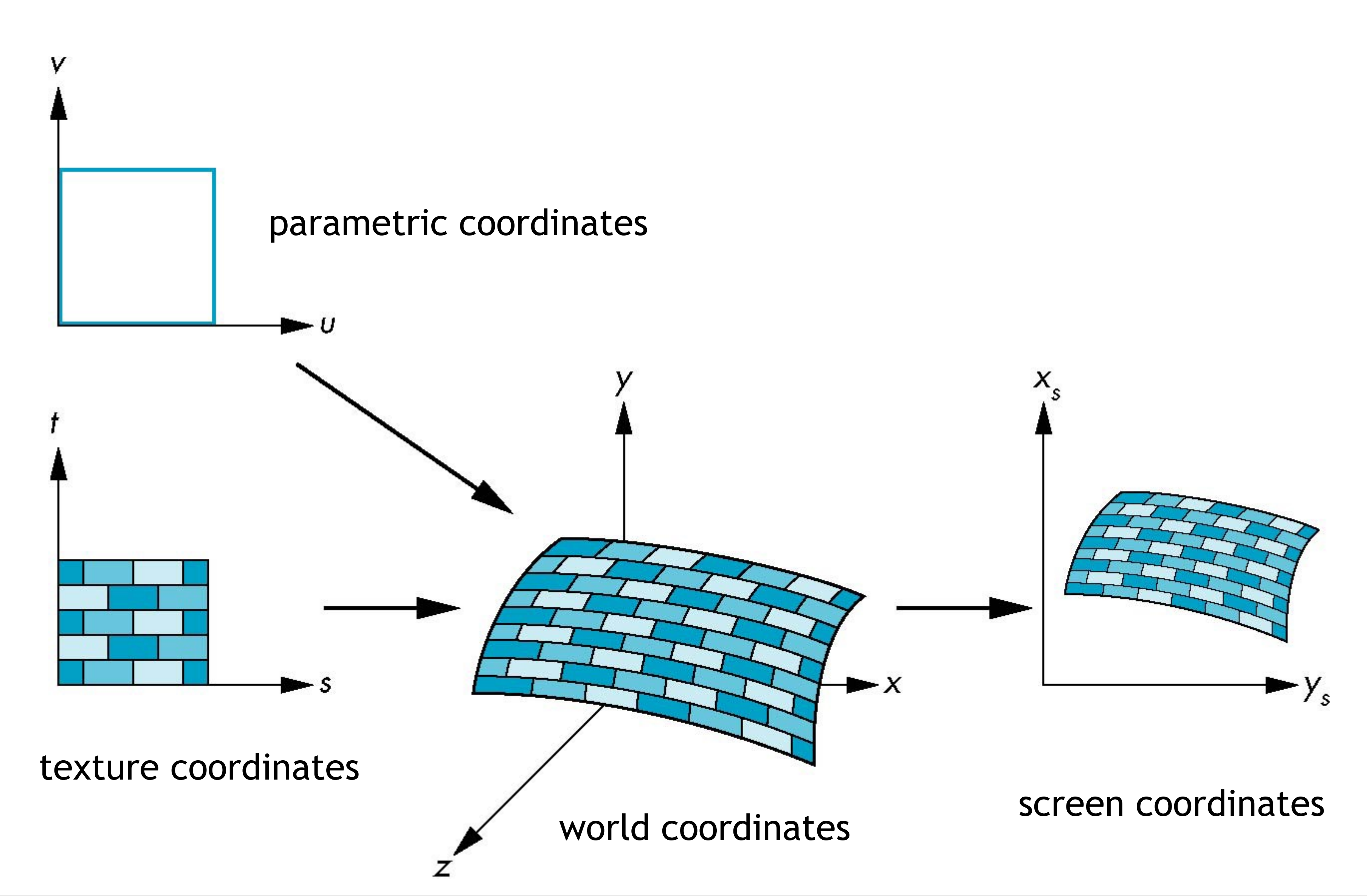
- Although the idea is simple --- map an image to a surface ---, there are 3 or 4 coordinate systems involved



Coordinate Systems

- Parametric coordinates (u,v)
  - Can be used to model curved surfaces
- Texture coordinates (s,t)
  - Used to identify points in the image to be mapped
- Object or World Coordinates (x,y,z)
  - Conceptually, where the mapping takes place
- Screen or Window Coordinates (x_s,y_s)
  - Where the final image is really produced

Coordinate Systems

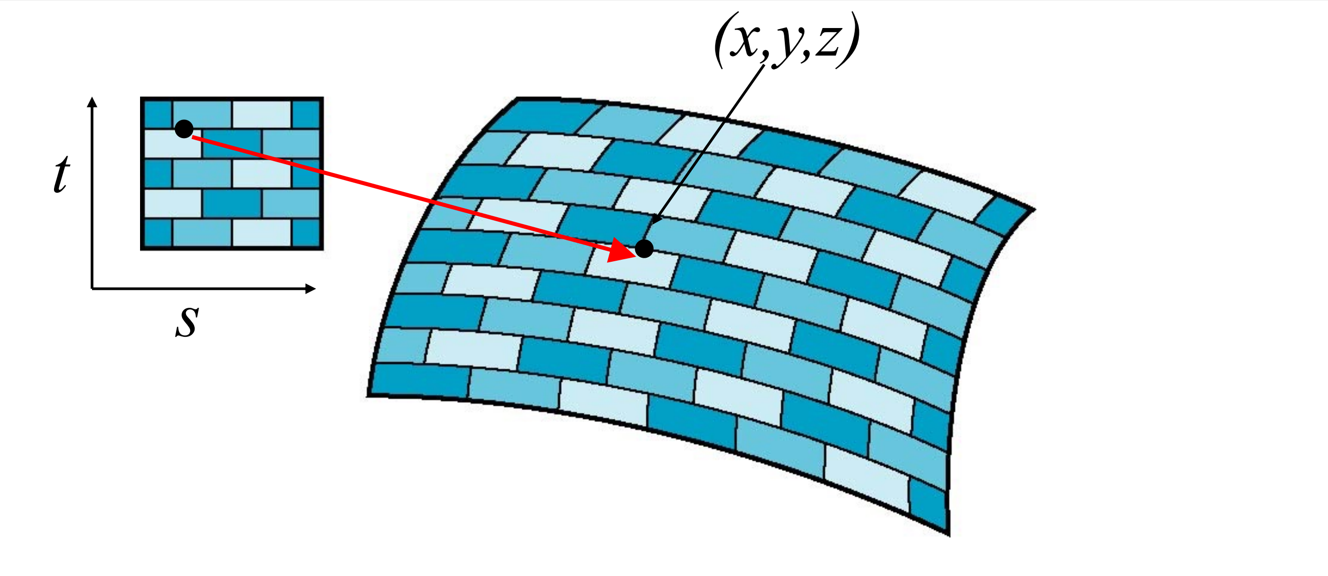


Mapping Functions

- Basic problem is how to find the map
- Consider mapping from texture coordinates to a point on the surface
- Need three functions

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

- But we actually want to go the other way

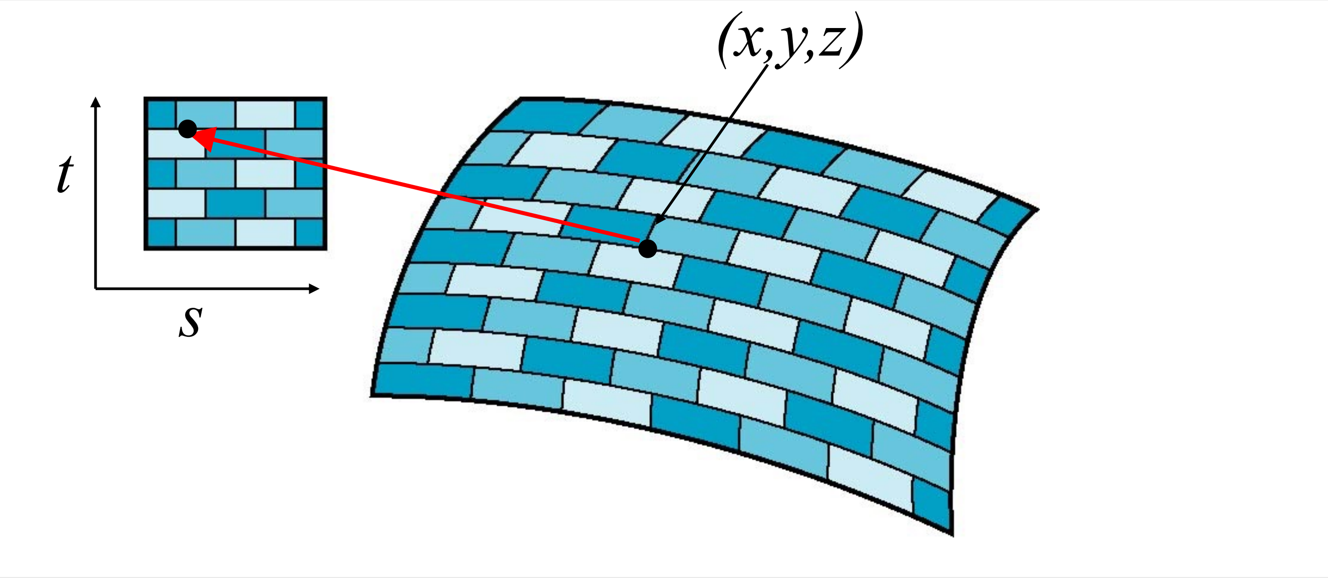


Backward Mapping

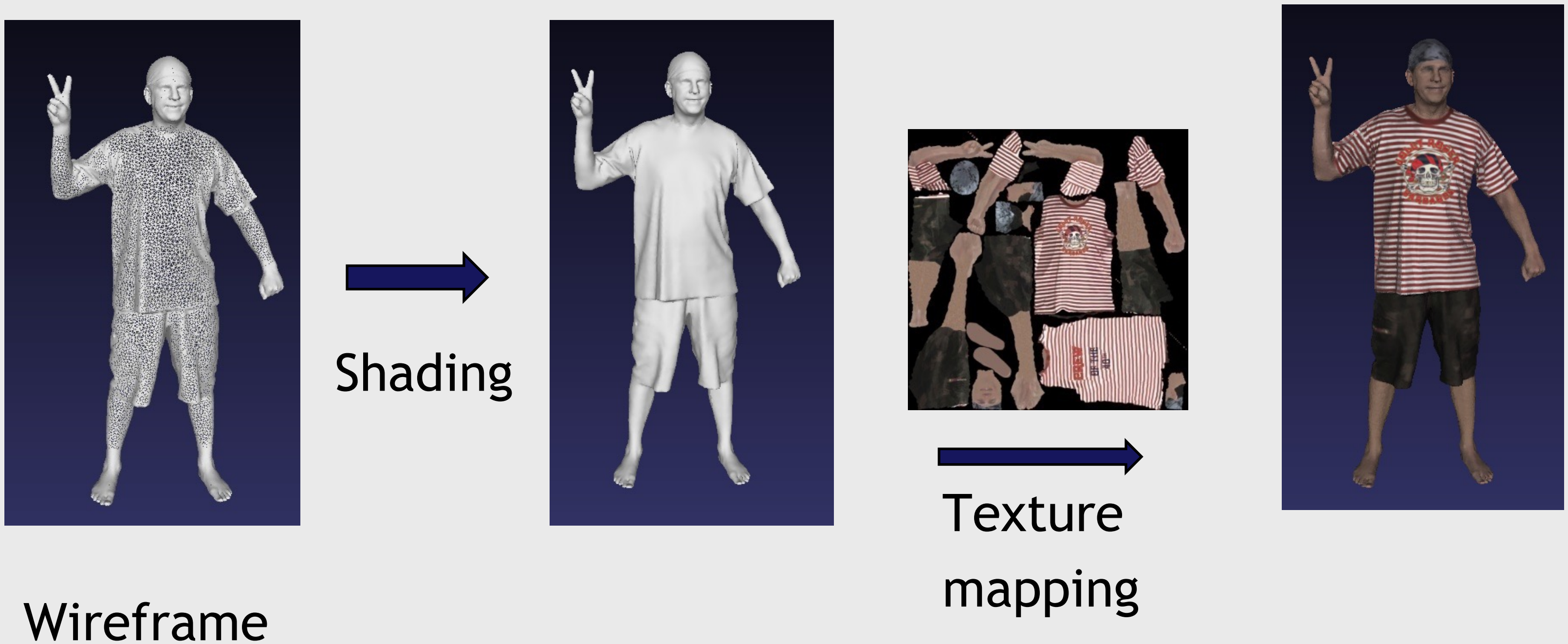
- We actually want to go backwards:
- Given a point on an object, we'd rather like to know to which point in the texture it corresponds (backward)
- Need a map of the form
  - $s = s(x,y,z)$
  - $t = t(x,y,z)$

$s,\ t$: texture coordinates $\quad 0 \le s,t \le 1$

- Such functions are difficult to find in general, if not given explicitly

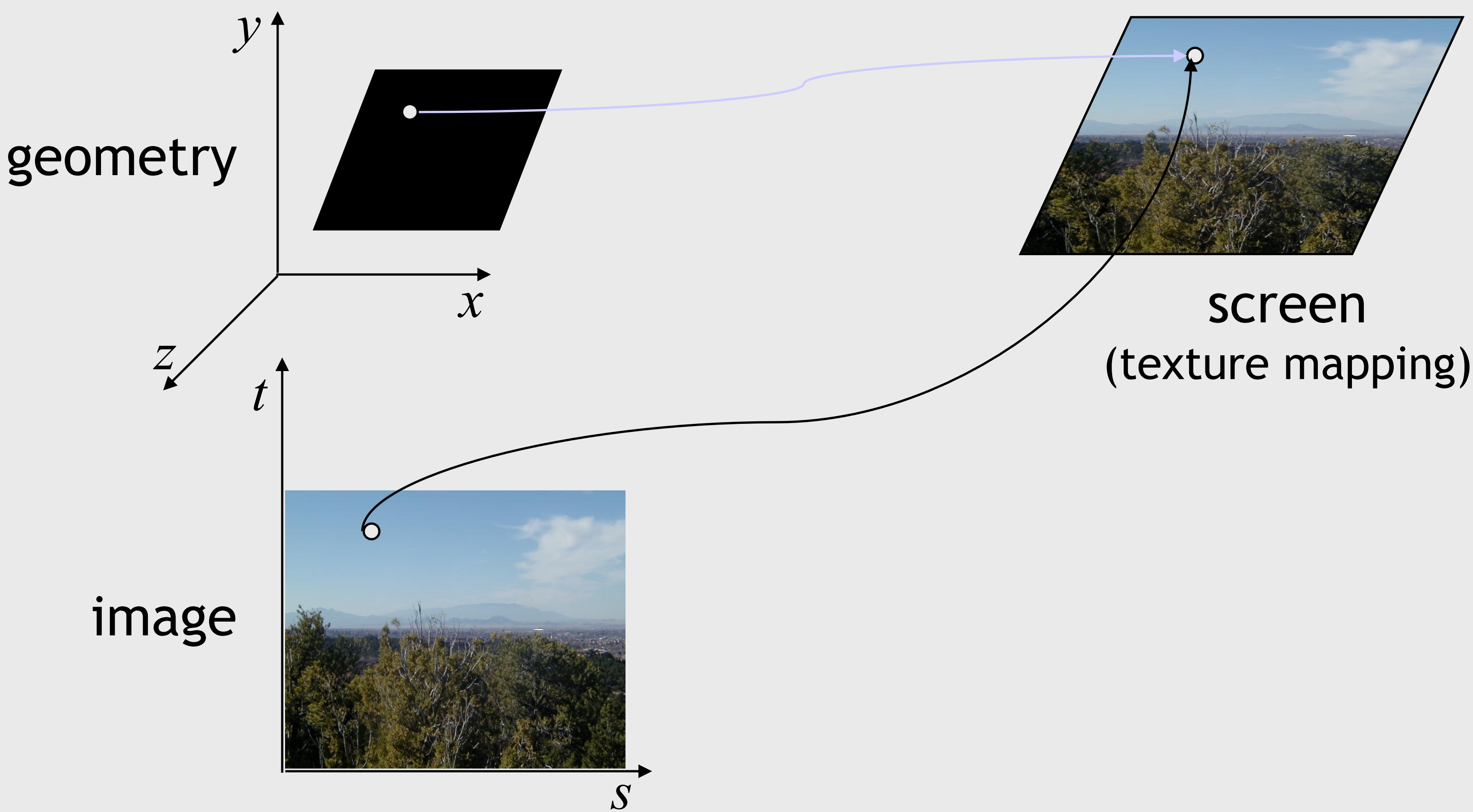


Texture Mapping - Example



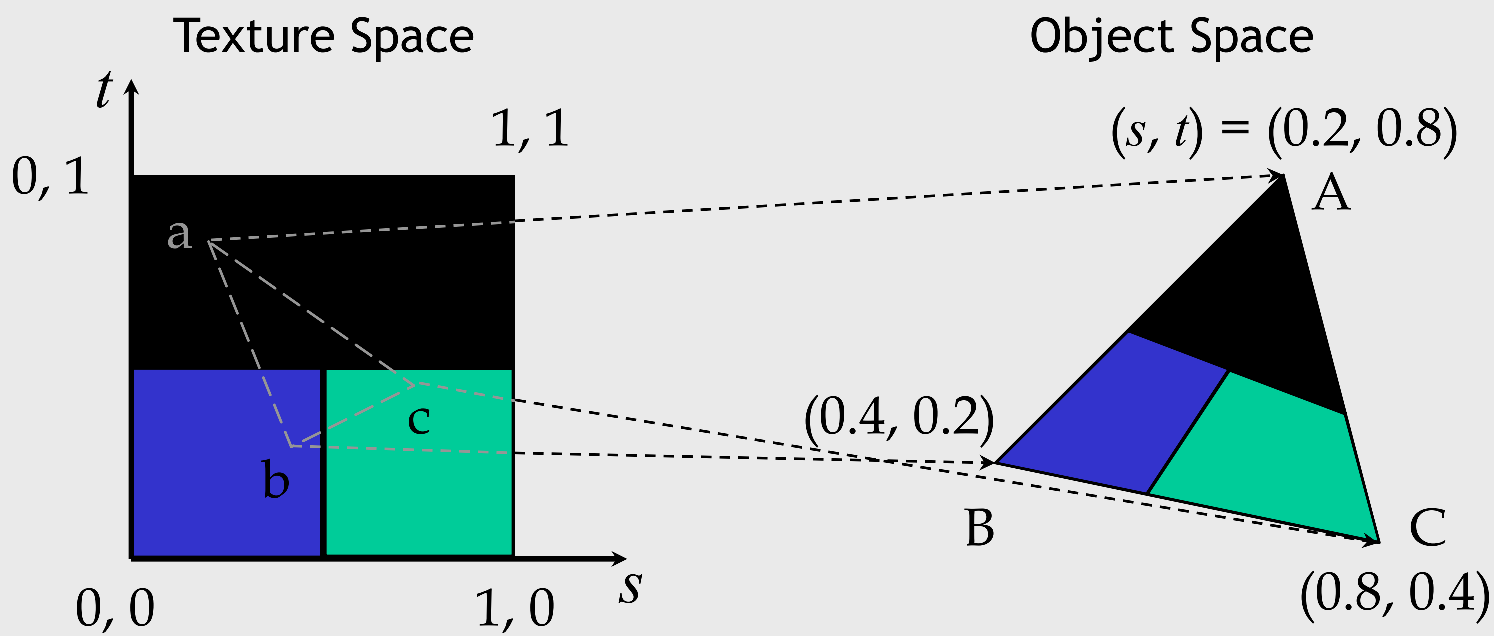
In this example, the mapping function, i.e., texture coordinates (s,t) associated with each vertex, are given explicitly.

Example



Mapping a Texture

- Based on parametric texture coordinates (s,t)
  - specified at each vertex as a vertex attribute

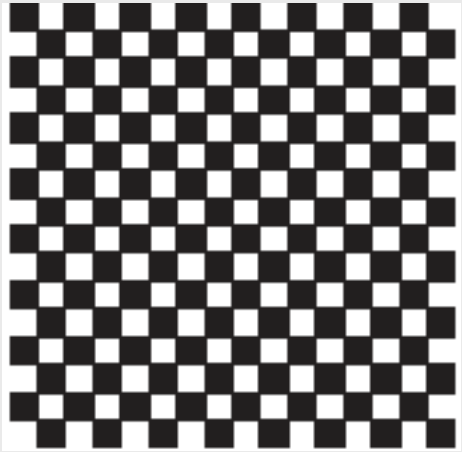


OpenGL uses bilinear interpolation to find the texture coordinates for the interior points of a polygon.

Interpolation

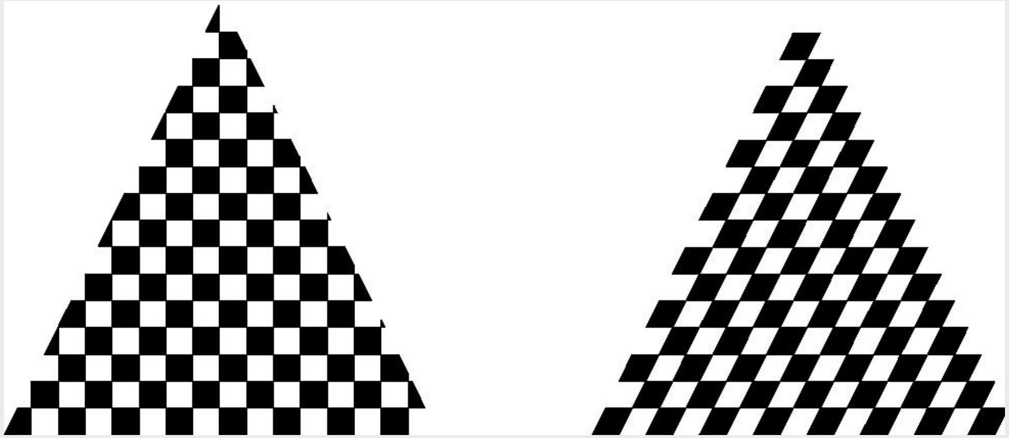
Bilinear interpolation may cause distortions when texture is mapped onto a triangle:

Texture image



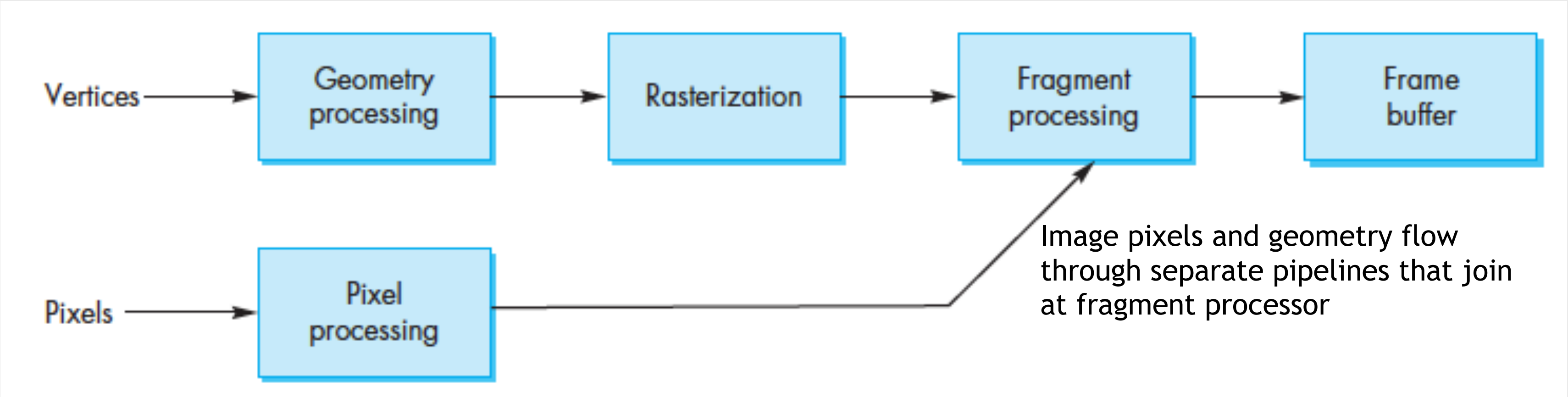
"good" selection
of texture coordinates

"poor" selection
of texture coordinates



Where does texture mapping take place?

- Texture mapping is carried out at the end of the rendering pipeline as part of fragment processing
  - More efficient because not all polygons make it past the clipper



Texture Mapping
Basic Strategy (OpenGL)

Texture mapping requires interaction among the application program, the vertex shader, and the fragment shader.

Three steps to apply a texture:

1. Specify the texture
   - read or generate the texture image
   - place it as texture object on GPU
2. Assign texture coordinates to vertices (which are then interpolated through fragments)
   - Proper mapping (assignment) function is left to application
3. Specify texture parameters
   - wrapping, filtering, etc

Texture Object

- First create texture object(s):
  - Generate ids and bind them

```
GLuint mytex[4];
glGenTextures(4, mytex);
glBindTexture(GL_TEXTURE_2D, mytex[0]);
```

- Any other call to `glBindTexture` either starts a new texture object, or switches to an existing texture object.

- Then specify the texture image and its parameters, which become part of the texture object.

- OpenGL supports 1-3 dimensional texture maps

Specify Texture Image

- Create a texture image from an array of *texels* (texture elements)

```
Glubyte my_texels[512][512][3];
```

- Specify that this array is to be used as a 2D texture after a call to `glBindTexture` function:

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

Function prototype:

- `glTexImage2D(GLenum target, GLint level, GLint iformat, GLsizei width, GLsizei height, GLint border, GLenum format, GLenum type, GLvoid *tarray)`

How to Specify an Image as Texture?

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

target: type of texture, e.g., **`GL_TEXTURE_2D`**
level: used for mipmapping (will be discussed soon)
components: elements per texel
w, h: width and height of **texels** in pixels
border: must be zero (no longer used)
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);
```

Texture Parameters

- OpenGL has a variety of parameters that determine how texture is applied, such as
  - Wrapping parameters determine what happens when texture coordinates s and t are outside $[0,1]$ range
  - Filtering modes allow us to use area averaging instead of point samples
  - Mipmapping allows us to use textures at multiple resolutions

Wrapping Mode

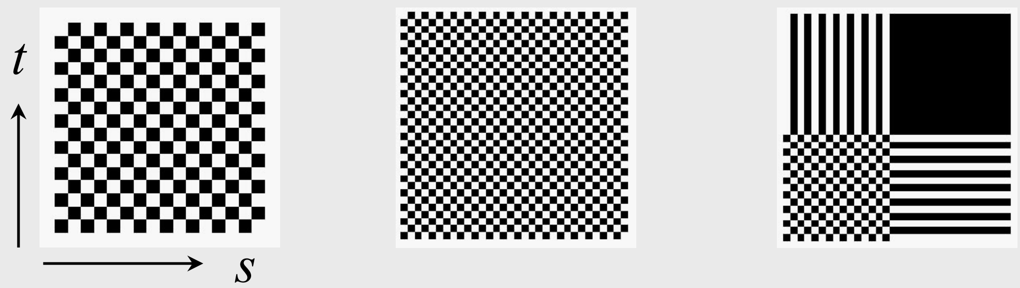
Determines what to do when texture coordinates are out of the range [0,1]:

Clamping: if $s, t > 1$ use 1, if $s, t < 0$ use 0

Wrapping (repeat): use s, t modulo 1

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP)

glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT)
```



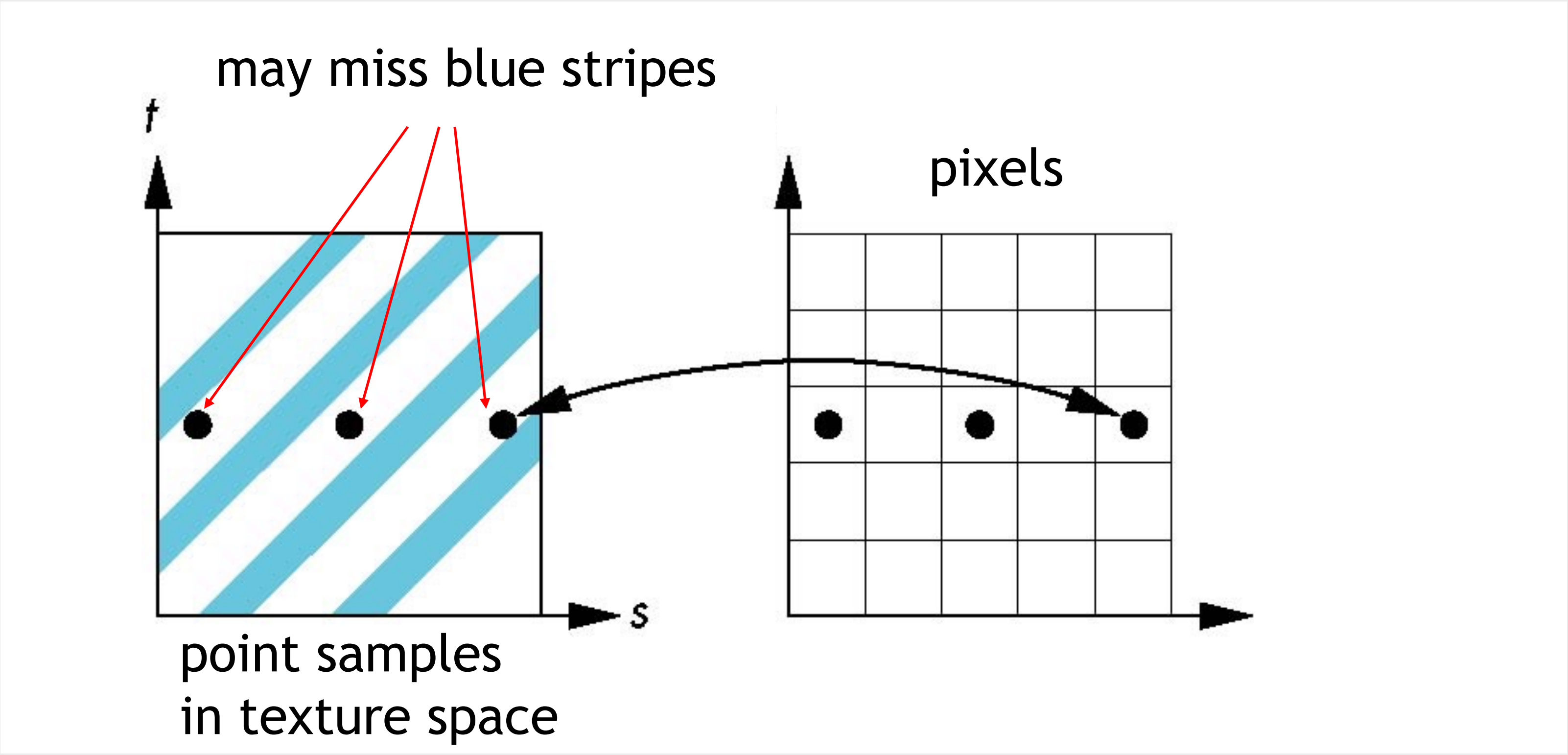
texture

GL_REPEAT wrapping

GL_CLAMP clamping

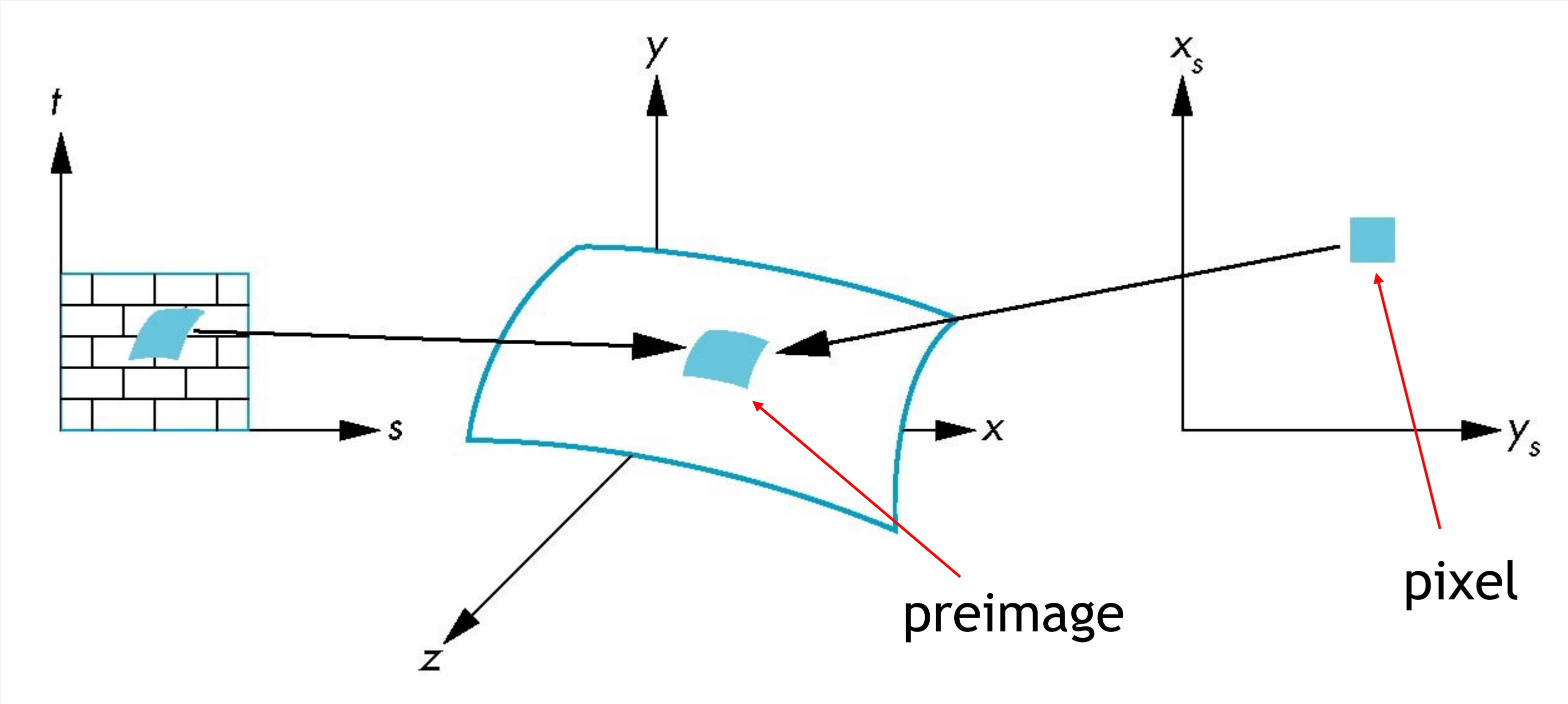
Aliasing

Point sampling of the texture can lead to aliasing errors:



Area Averaging

A better but slower option is to use area averaging



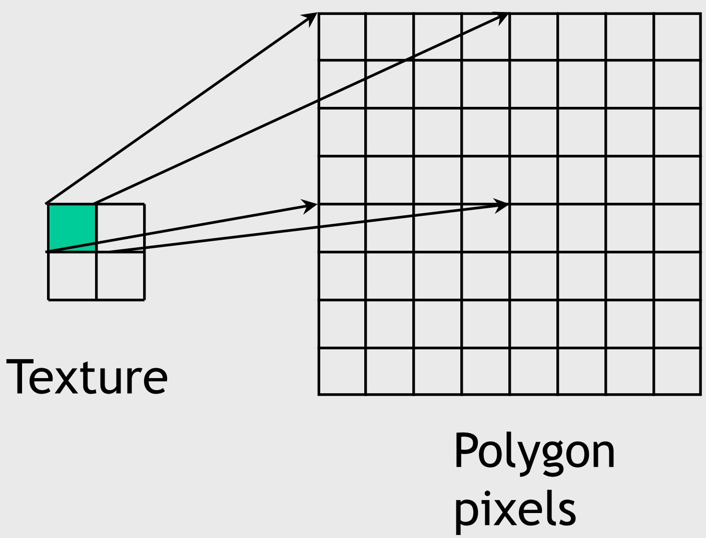
Note that the preimage of a pixel is curved

OpenGL - Texture Aliasing

Let’s see how OpenGL handles this problem…

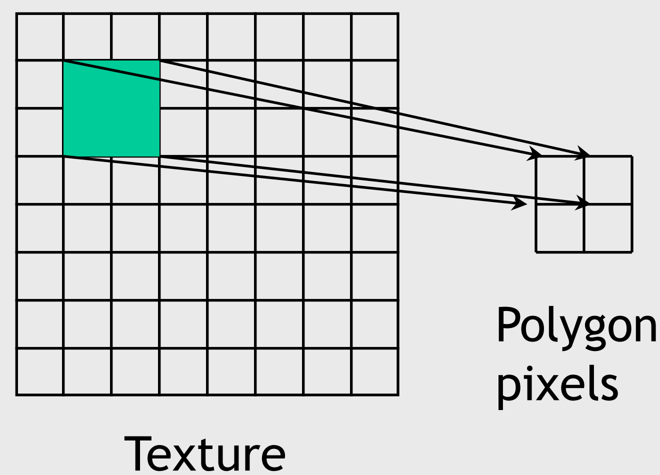
Texture Aliasing Problems

- **Magnification:** More than one pixel may cover a texel
- **Minification:** More than one texel may cover a pixel



Magnification

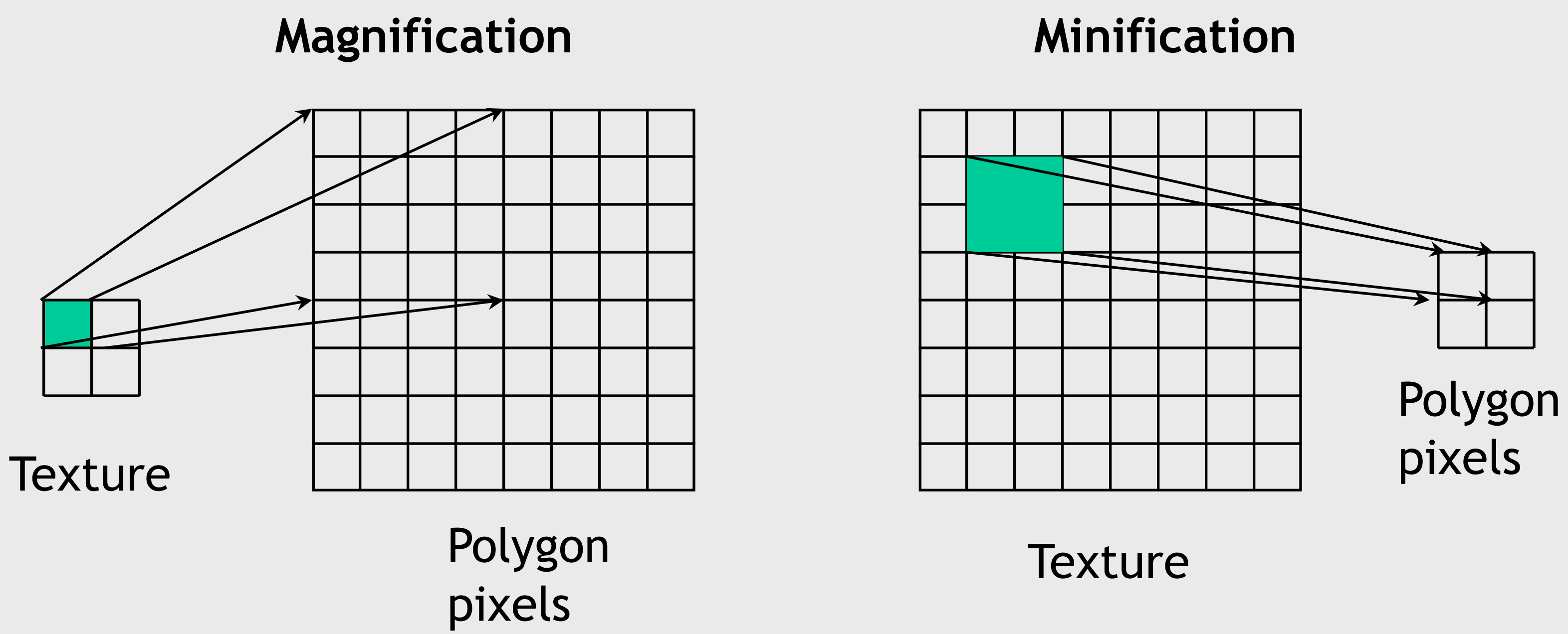
e.g., when zooming in



Minification

e.g., when zooming out

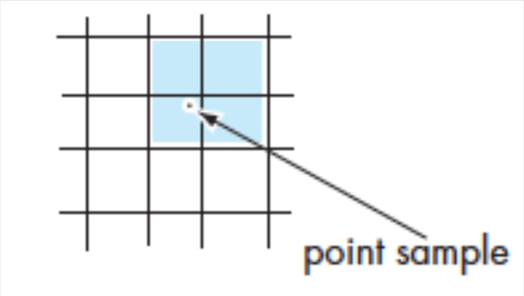
How to deal with magnification and minification?



To obtain texture values, we can use

- point sampling (use nearest texel)
- linear filtering (use the average in 2×2 neighborhood)

Filter Modes (OpenGL)



Point sampling (`GL_NEAREST`) vs Linear filtering (`GL_LINEAR`)

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);
```

Mipmapped Textures

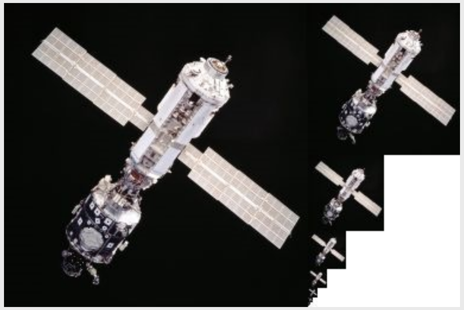
- Mipmapping allows for prefiltered texture maps of decreasing resolutions
- OpenGL can create a series of texture arrays at reduced sizes, and then automatically uses the appropriate size.
- For a 64 × 64 original array, we can set up 32 × 32, 16 × 16, 8 × 8, 4 × 4, 2 × 2, and 1 × 1 arrays for the current texture object by executing the function call:

```
glGenerateMipmap(GL_TEXTURE_2D);
```

- These mipmaps are invoked automatically by

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

```
GL_NEAREST_MIPMAP_LINEAR: Do point-sampling on the best two mipmap
images and then compute a weighted average (linear-sampling).
```



Mipmapped texture images

See Ch. 7.6.5 in the textbook

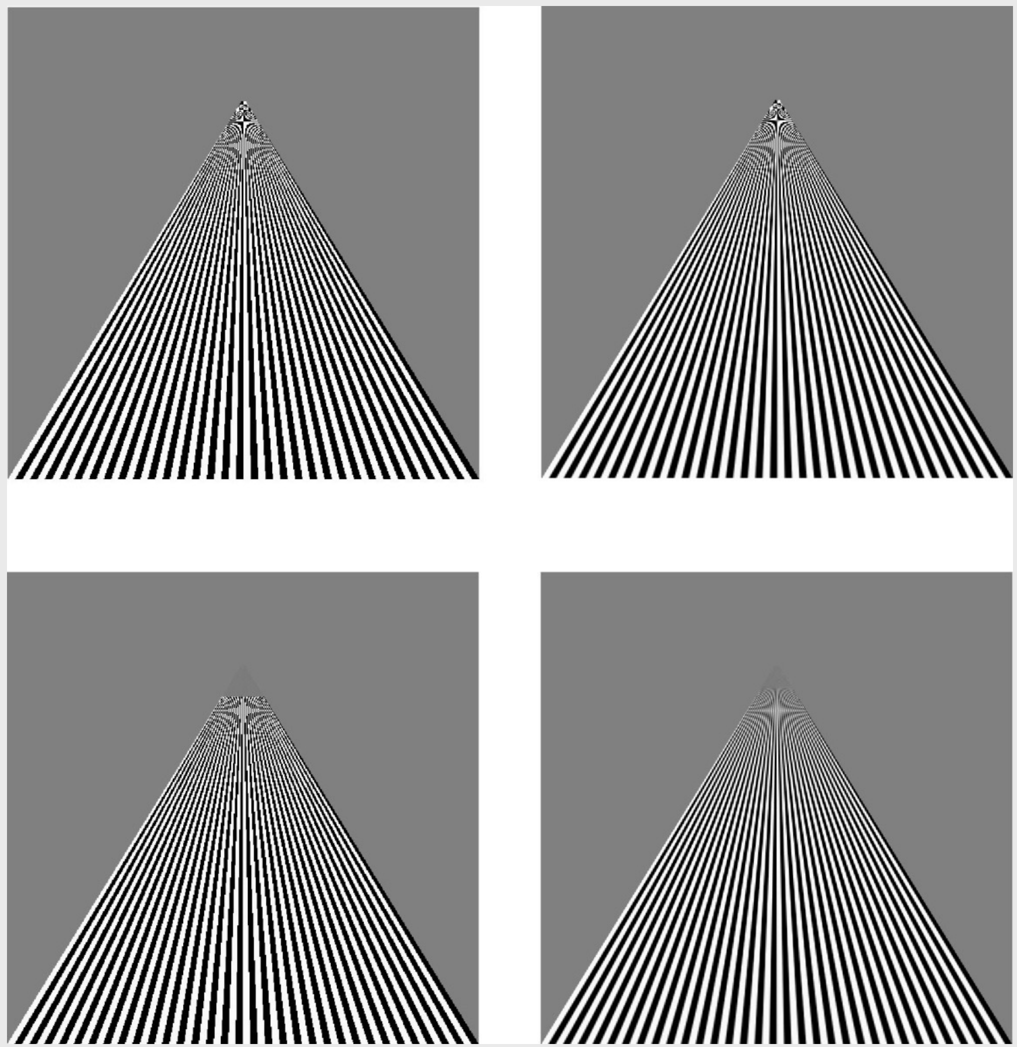
Example - Minification

point sampling

GL_NEAREST

linear filtering

GL_LINEAR



mipmapped point sampling

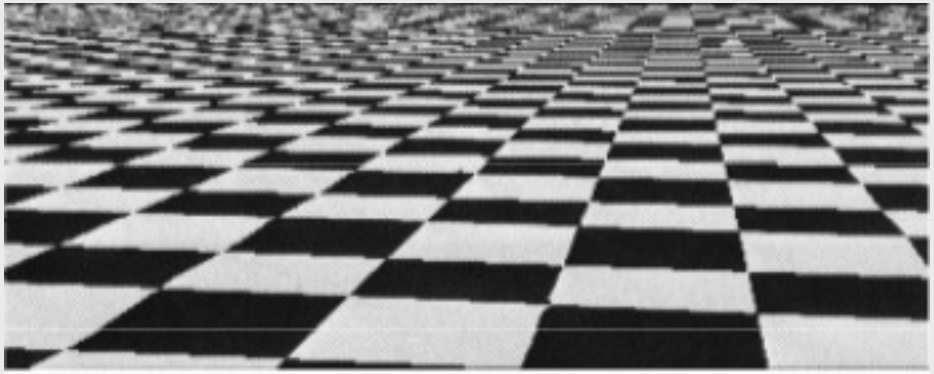
GL_NEAREST_MIPMAP_LINEAR

mipmapped linear filtering

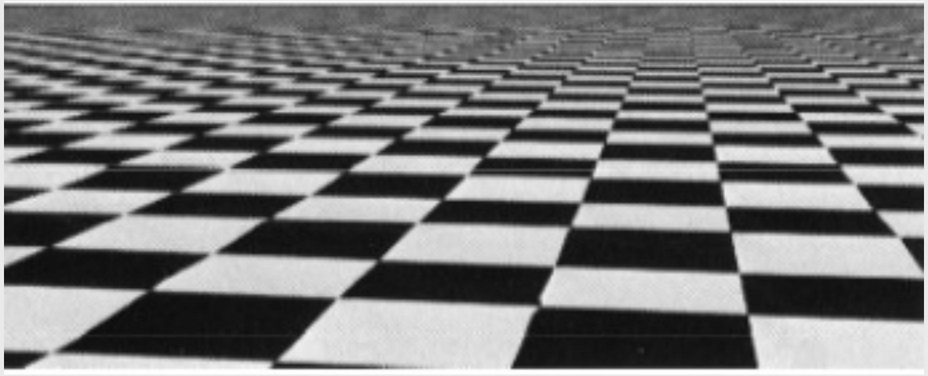
GL_LINEAR_MIPMAP_LINEAR

Example - Minification

point sampling



mipmapped linear filtering



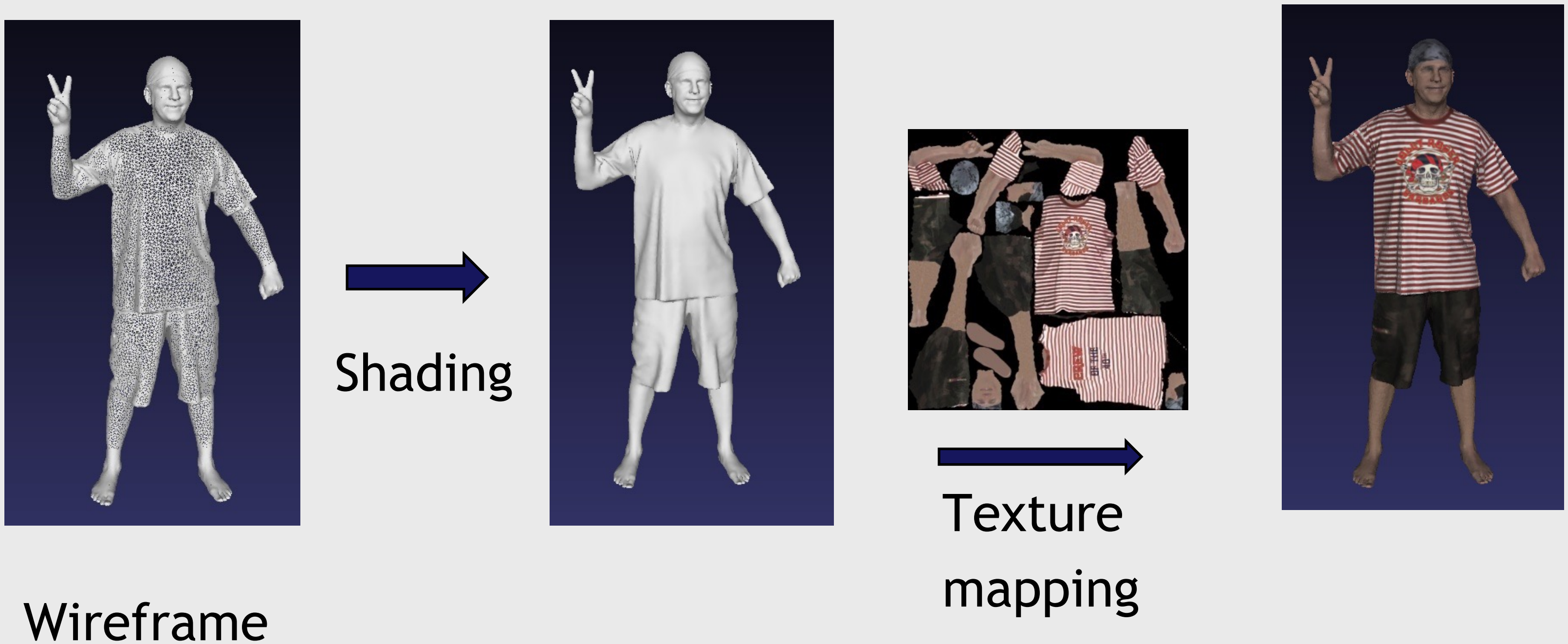
Applying Textures

- Textures are applied during fragment processing by using a sampler function `texture`
- Sampler function returns a texture color from the texture object `tex`, given the interpolated texture coordinates `texCoord`

```
in vec2 texCoord; //texture coordinate from rasterizer
uniform sampler2D tex; //texture object from application
out color;

void main()  {
    color = texture( tex, texCoord );
}
```

Texture Mapping - Example

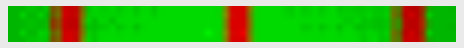


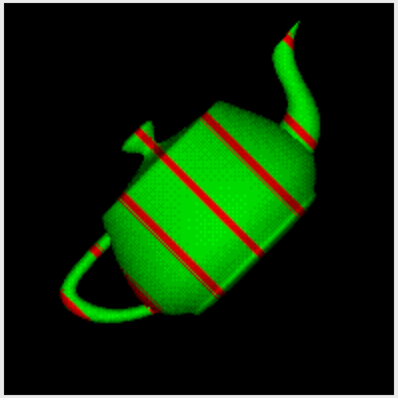
In this example, the mapping function, i.e., texture coordinates (s,t) associated with each vertex, are given explicitly.

Generating Texture Coordinates

- We have assumed so far that texture coordinates are explicitly provided with data.
- If not, application should somehow automatically generate them. There are various ways of doing that.
- **One option:** Specify a plane, and then generate texture coordinates based upon distance from that plane.
- A **1D texture** mapping example:

1D texture:



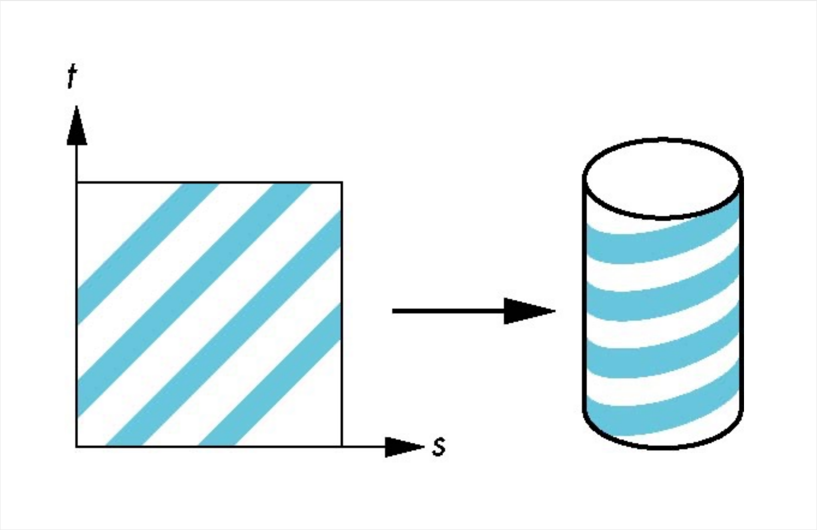


See Ch. 7.7.6 in the textbook

- **Another option:** Use two-part mapping approach described next.

Two-part Mapping Method

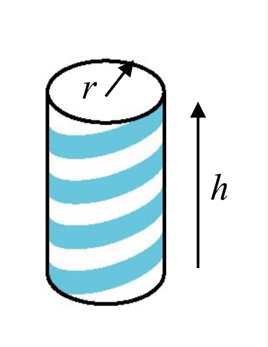
- First map the texture onto a simple intermediate surface, and then to the actual object surface
- Example: Map first to a **cylinder**



Cylindrical Mapping

Parametric cylinder

$$x = r\cos(2\pi u)$$
$$y = r\sin(2\pi u) \qquad\qquad 0 \le u,v \le 1$$
$$z = v \cdot h$$

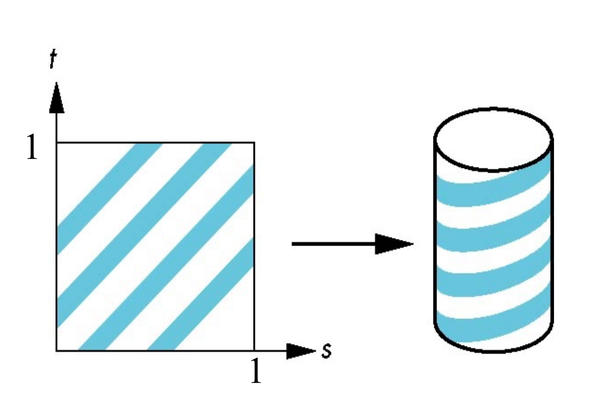


maps the unit rectangle in u,v space to a cylinder of radius r and height h in world coordinates.

Hence

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

gives a mapping to generate texture coordinates.

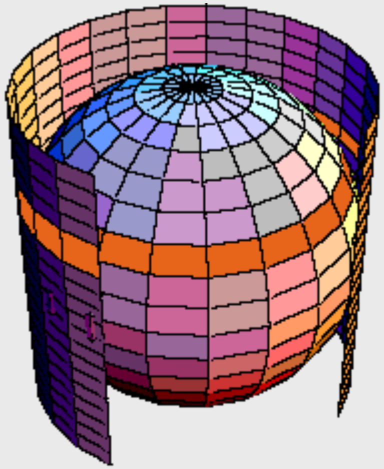


We should then write u and v in terms of x, y, z so that we can associate a texture coordinate with each vertex

Spherical Mapping

We could also use a parametric sphere:

$$x = r\cos(\pi u)$$
$$y = r\sin(\pi u)\cos(2\pi v) \qquad 0 \le u, v \le 1$$
$$z = r\sin(\pi u)\sin(2\pi v)$$

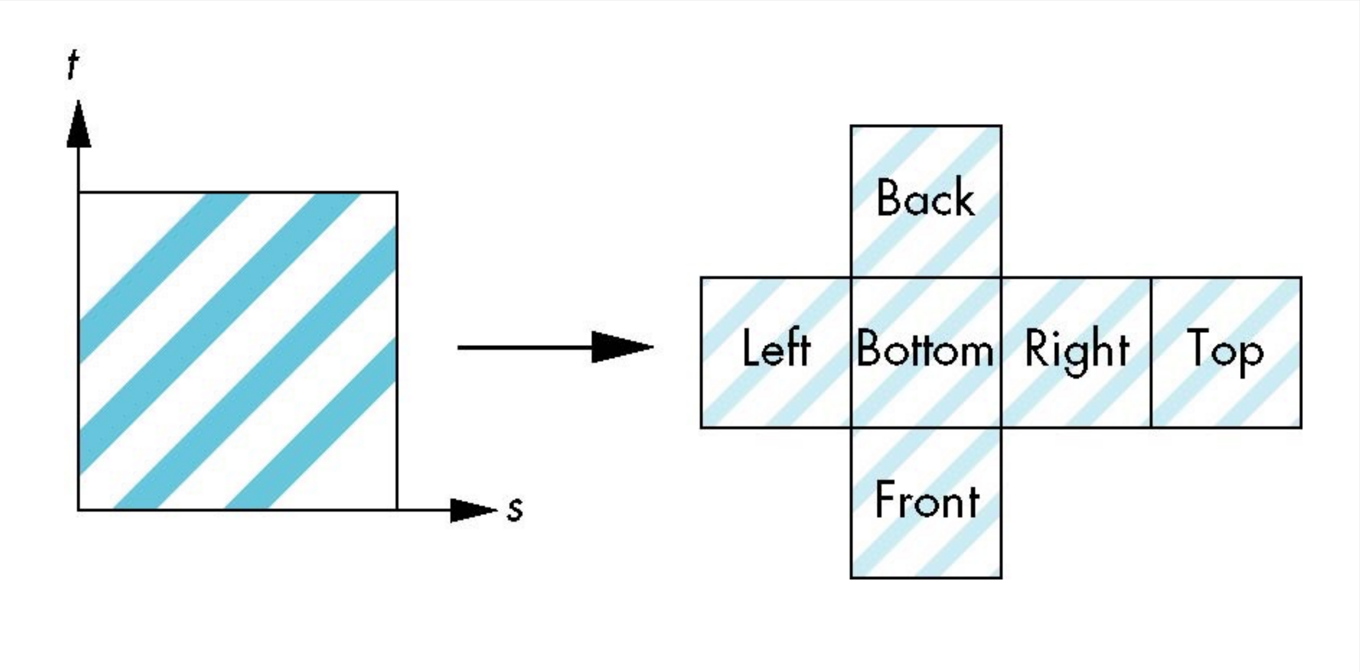


in a similar manner to the cylinder.

But we then have to decide where to put the distortion
(at the poles of the sphere for instance).

Spherical maps are used especially in **environmental mapping**.

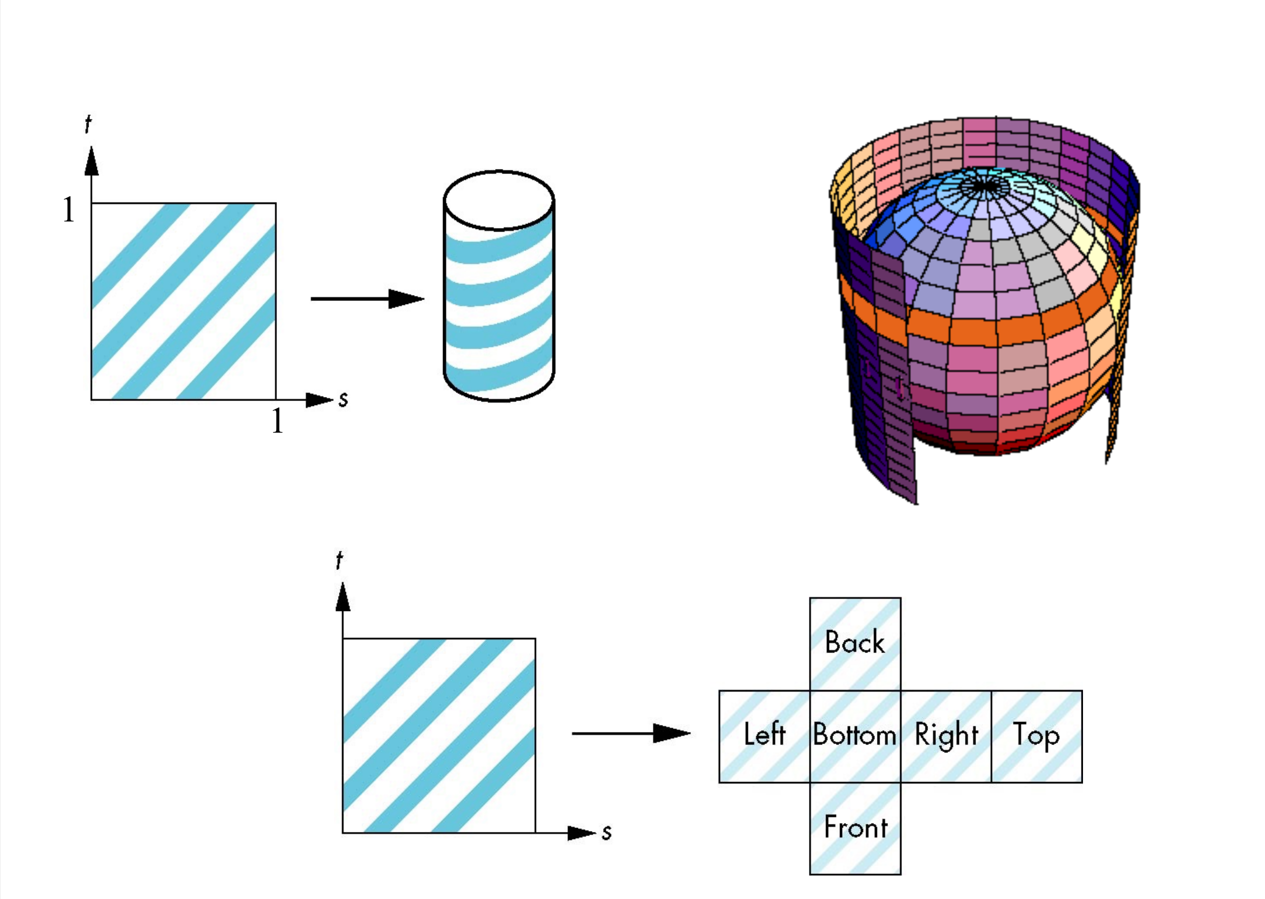
Box Mapping



Can also be used in environmental maps

Second-part Mapping

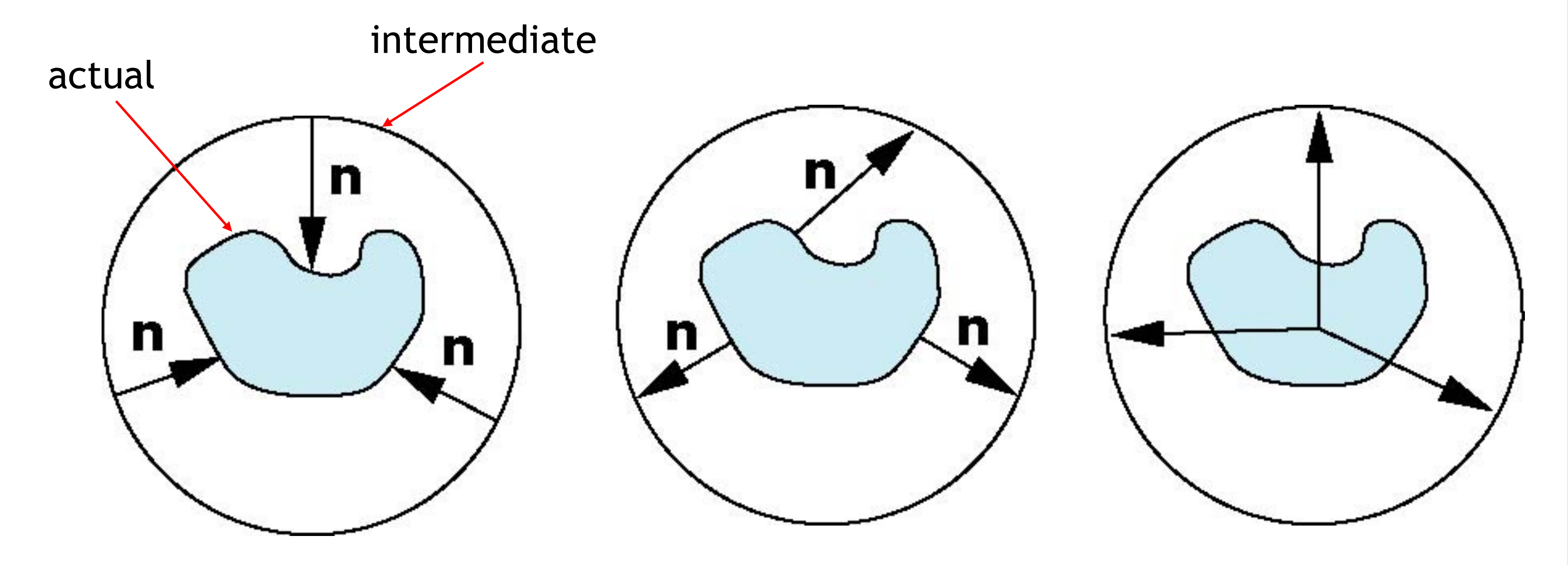
But what if the actual object is not a sphere, cylinder or a box?



Second-part Mapping

Mapping between *intermediate* object and *actual* object:

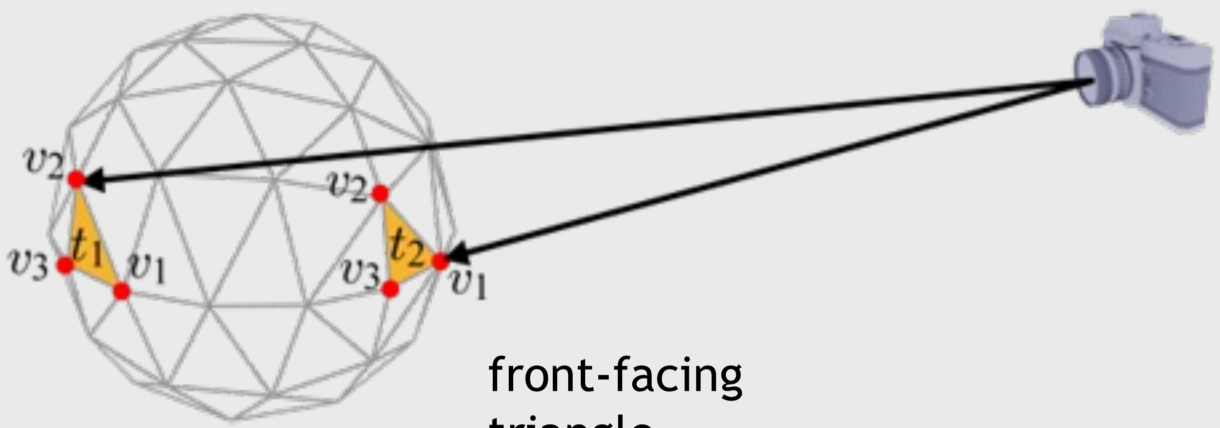
- Use **normals** from intermediate to actual
- Use **normals** from actual to intermediate
- Use **vectors** from the center of intermediate



Back-Face Removal (Culling)

- We can avoid rendering triangles facing away from the viewer
- This is called culling
- Triangles can be culled prior to clipping (i.e., discarded without rendering), based on the triangle's facing in camera space

back-facing triangle



front-facing triangle

Image by tizen.org

Back-Face Removal (Culling)

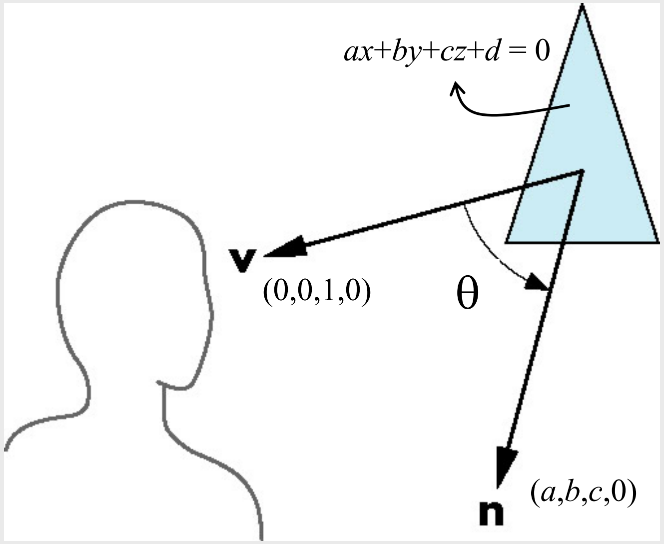
- A face is visible if $-90 \leq \theta \leq 90$
- or equivalently if $0 \leq \boldsymbol{v} \cdot \boldsymbol{n} = \cos\theta$

- If the plane of the face is $ax+by+cz+d = 0$ in normalized window coordinates (hence after view normalization), we have $\boldsymbol{v} = (0,0,1,0)$ and need only to test the sign of c.

Note that the normal of the plane can be written as $\boldsymbol{n} = (a,b,c,0)$.

- In OpenGL, we can simply enable culling with:

```
glEnable(GL_CULL_FACE)
```



in normalized window coordinates

Note: Using `glCull(.)`, you can also choose which faces to cull, front-facing or back-facing, that might be useful when the camera gets inside a closed object for rendering.