COMP3421

Texturing

The graphics pipeline
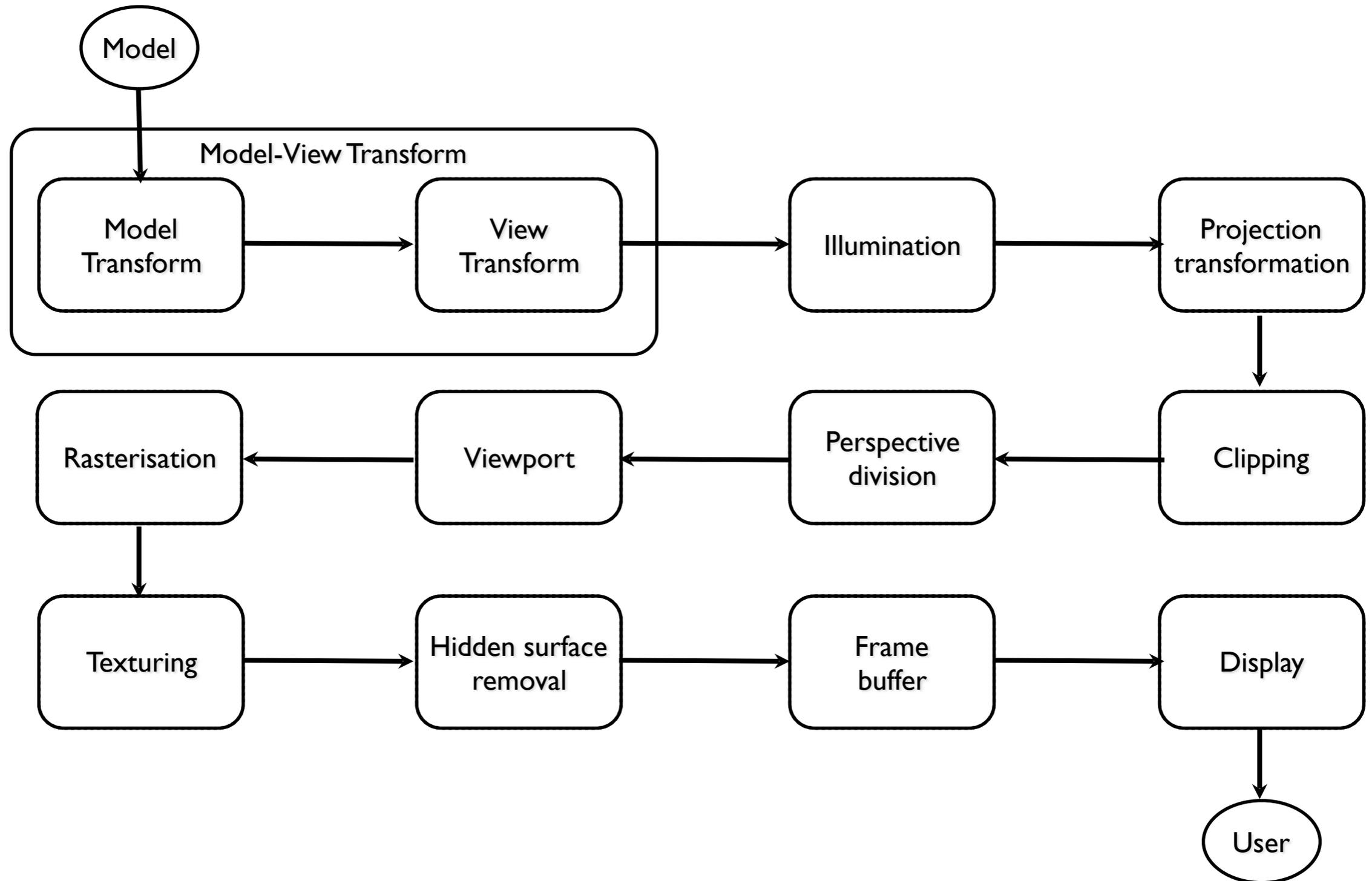
Texturing

Textures are a way to add detail to our models without requiring too many polygons.

Textures are used to add:

- Colour
- Reflections
- Shadows
- Bumps
- Lighting effects
- etc...

Textures

A texture is basically a function that maps texture coordinates to pixel values.

$$T(s,t) = \begin{pmatrix} r \\ g \\ b \\ a \end{pmatrix}$$

Texture coordinates are usually in the range (0,1).

Textures

A texture is basically a function that maps texture coordinates to pixel values.

$$T(s, t) = \begin{pmatrix} r \\ g \\ b \\ a \end{pmatrix}$$

texture coords
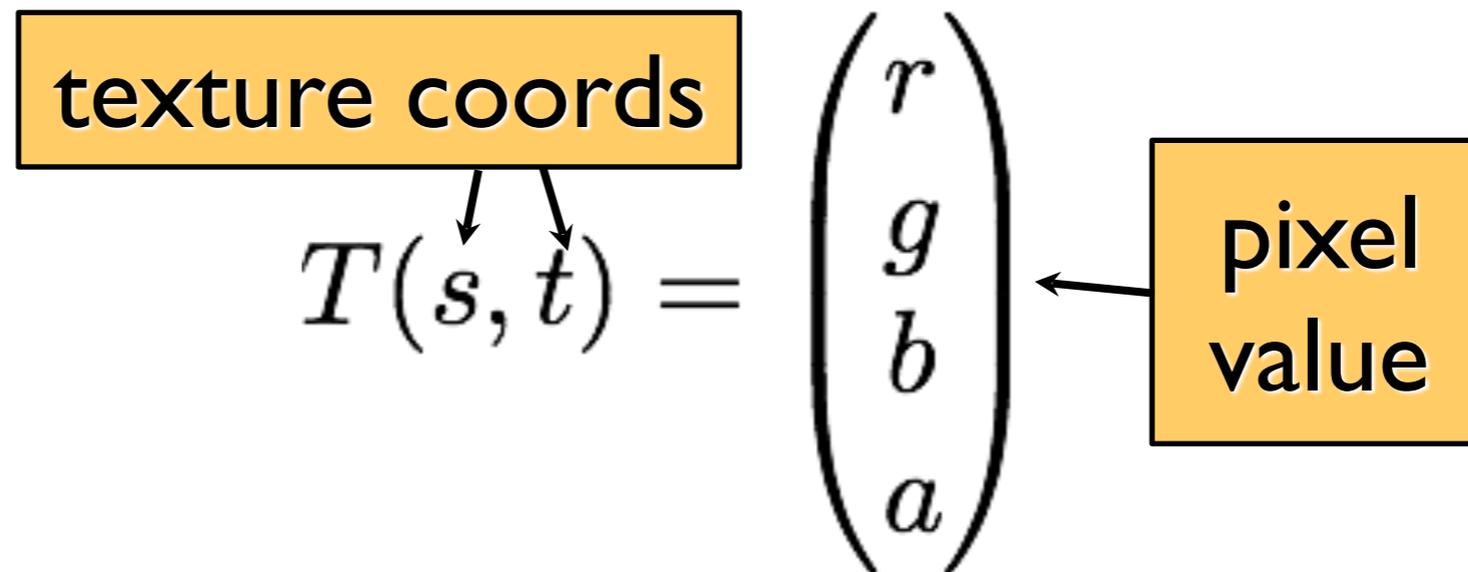
pixel value

Texture coordinates are usually in the range (0,1).

Textures

Textures are most commonly represented by bitmaps

i.e. 2D image files

$T(s,t)$ = texel value at (s,t)

texel = pixel on a texture

$(1,1)$

$(0,0)$

t

s
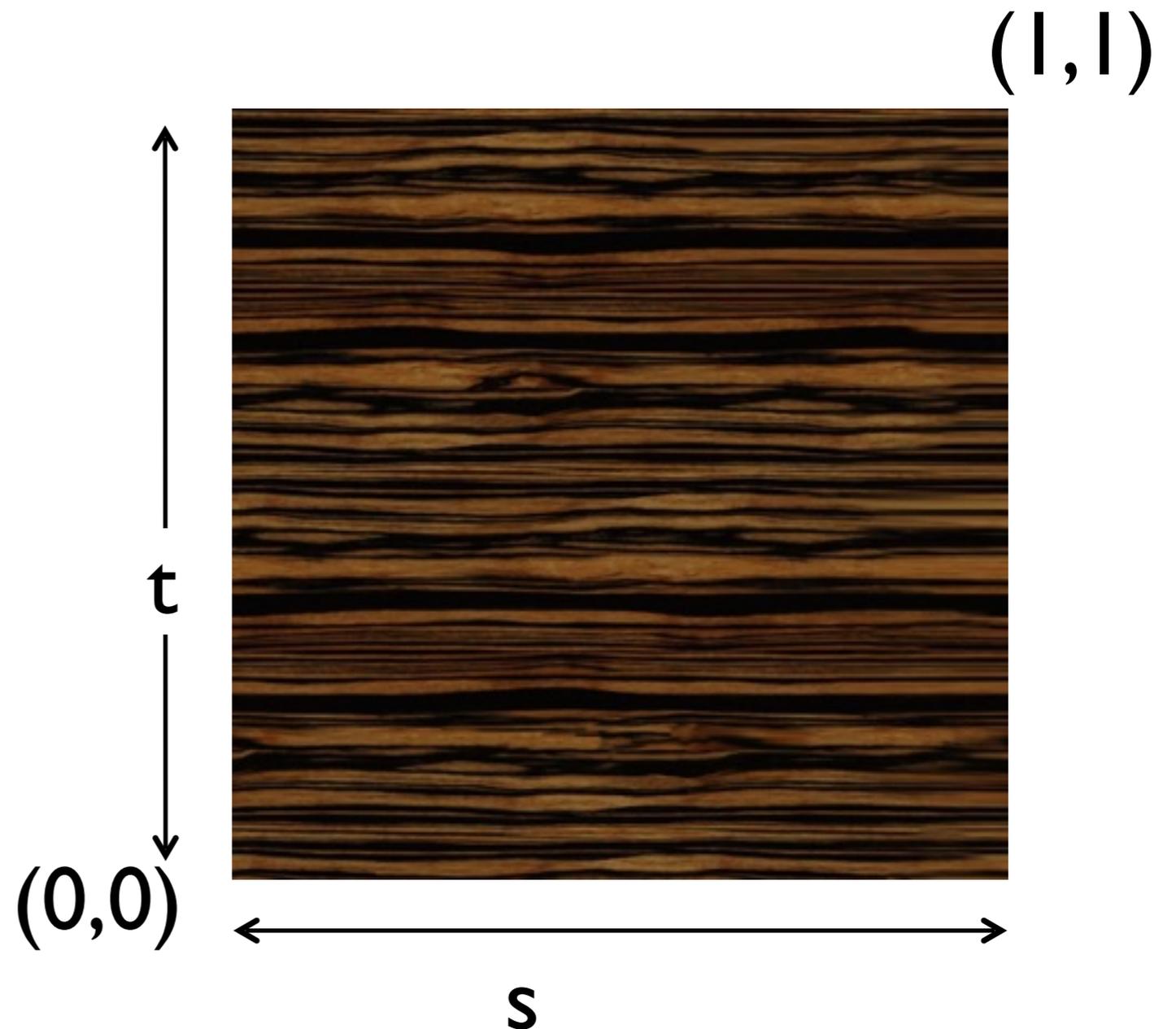
Procedural textures

It is also possible to write code to compute the texture value at a point.

This can be good to generate materials like marble or woodgrain.

Using Textures

1. Load or create textures

2. Turn on texturing
   `gl.glEnable(GL2.GL_TEXTURE_2D);`

3. Set the current texture

   `gl.glBindTexture(GL2.GL_TEXTURE_2D,texId);`

4. Map texture co-ordinates to vertices

Texture mapping

To add textures to surfaces in on our model, we set texture coordinates for each vertex.
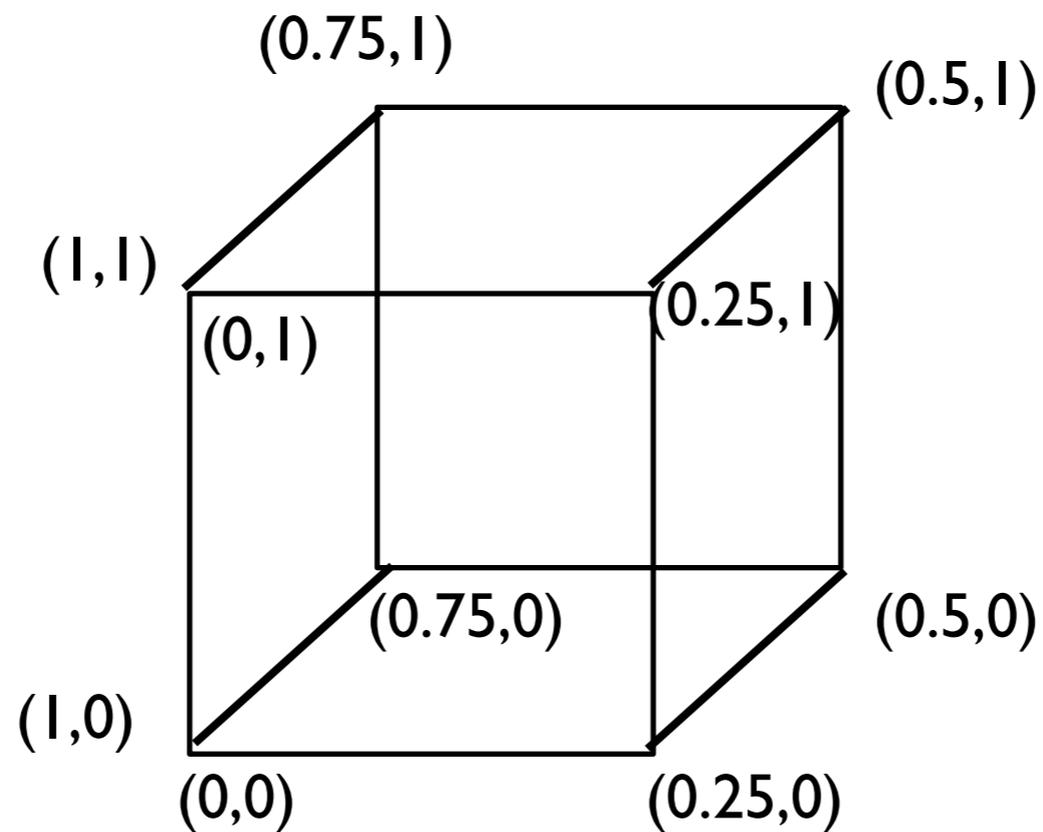
Texture Co-ordinates

gl.glNormal3d(0,0,1);

gl.glTexCoord2d(0,0);

gl.glVertex3d(-5,-5,0);

gl.glTexCoord2d(1,0);

gl.glVertex3d(5,-5,0);

//etc

Exercise: Circle

(1,1)

$x = \cos(\text{theta})$
$y = \sin(\text{theta})$

t

(0,0)

s
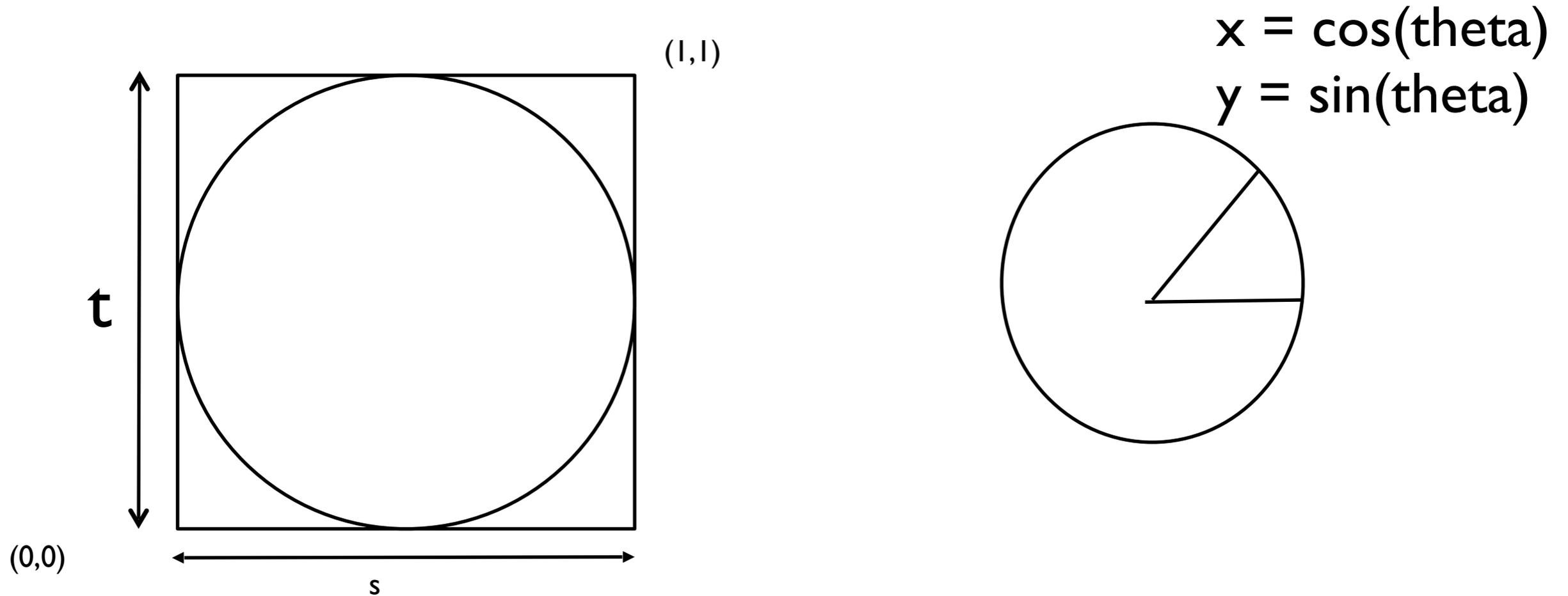
```
gl.glTexCoord2d(0.5+0.5*Math.cos(theta),
                0.5+0.5*Math.sin(theta));
```

Mapping a Cylinder

Cylinder: s is an angle coordinate, t is a height coordinate.
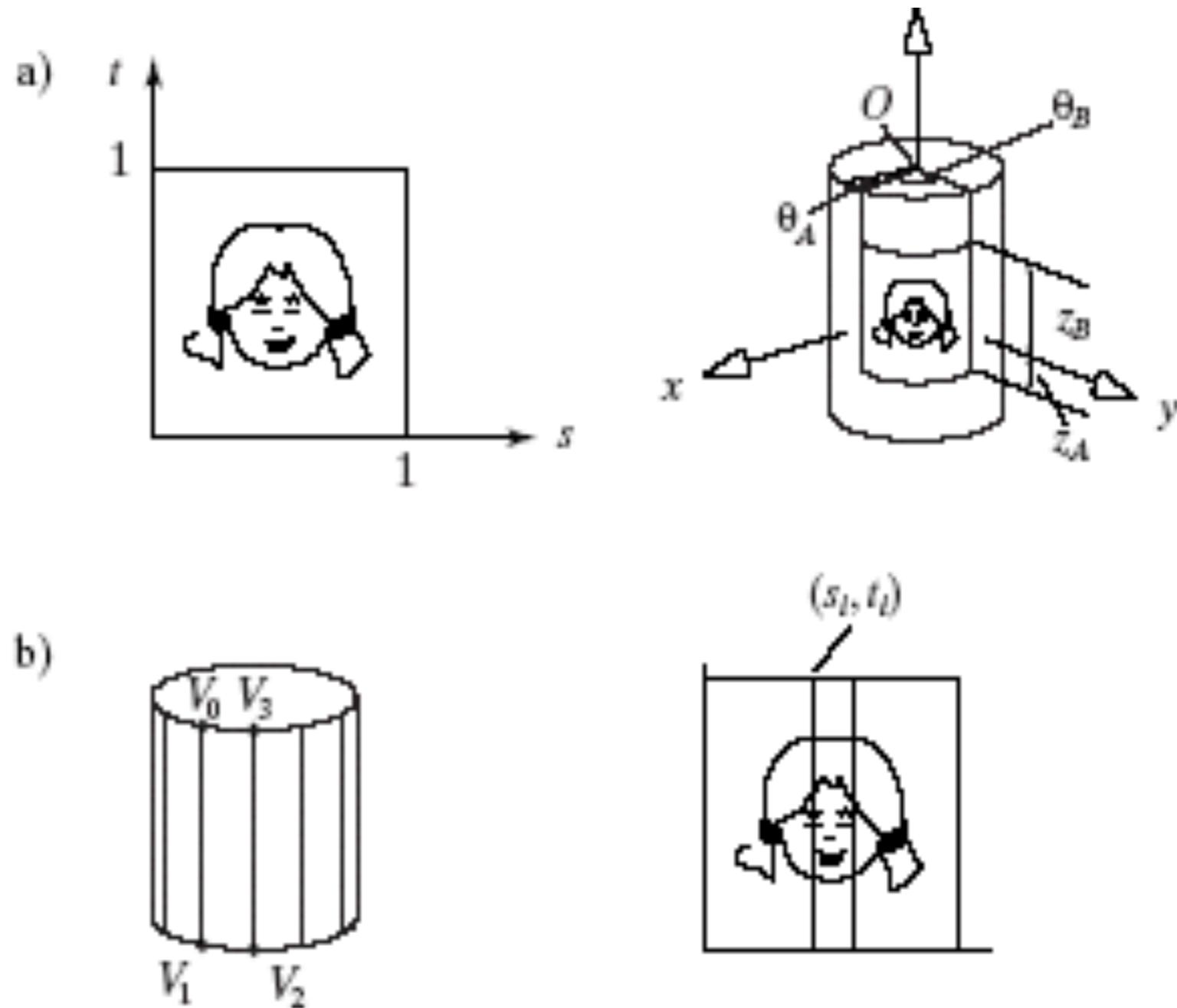
Model Texture Mapping

We can assign texture coordinates to vertices however we want. Complex models often have weird flattened textures.

Texture Wrap

You can assign texture coordinates outside the range [0,1] and set the texture wrap to

GL2.GL_REPEAT (default)

GL2.GL_MIRRORED_REPEAT

GL2.GL_CLAMP_TO_EDGE

Texture WRAP

For example, setting to GL2.GL_REPEAT in both s and t dimensions:

```
gl.glTexParameteri( GL2.GL_TEXTURE_
2D,   GL2.GL_TEXTURE_WRAP_S,
GL2.GL_REPEAT);
```

```
gl.glTexParameteri( GL2.GL_TEXTURE_
2D, GL2.GL_TEXTURE_WRAP_T,
GL2.GL_REPEAT);
```

Repeating a Texture

For example this shows the use of texture coordinates outside [0,1] that **repeat** the texture, if the setting is GL_REPEAT
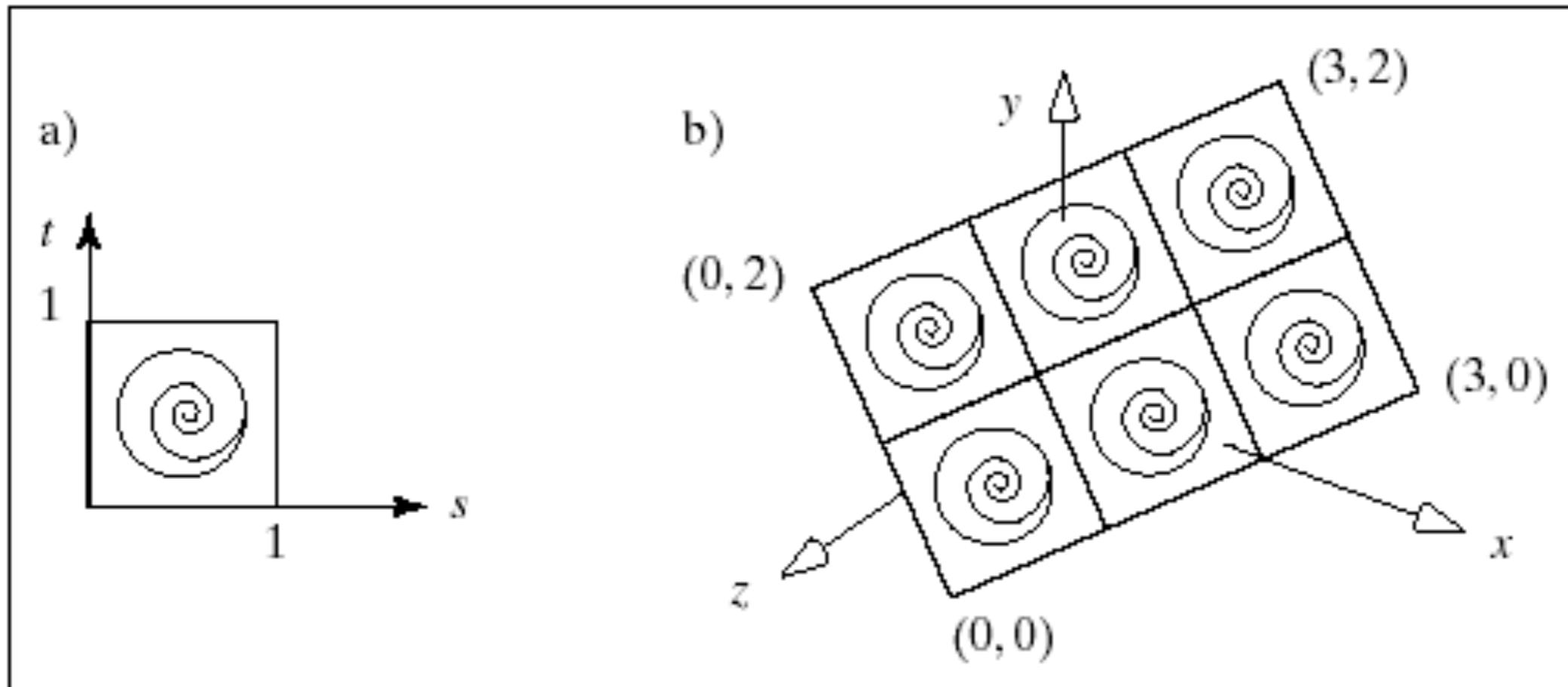
Textures and shading

How do textures interact with shading?

The simplest approach is to replace illumination calculations with a texture look-up.

$$I(P) = T(s(P), t(P))$$

This produces objects which are not affected by lights or color.

Textures and shading

A more common solution is to use the texture to modulate the ambient and diffuse reflection coefficients:

$$I(P) = T(s,t)\left[I_a\rho_a + I_d\rho_d(\hat{\mathbf{s}} \cdot \hat{\mathbf{m}})\right] + I_s\rho_s(\hat{\mathbf{r}} \cdot \hat{\mathbf{v}})^f$$

We usually leave the specular term unaffected because it is unusual for the material colour to affect specular reflections.

OpenGL

```
// to use without lighting

gl.glTexEnvf(GL2.GL_TEXTURE_ENV,
             GL2.GL_TEXTURE_ENV_MODE,
             GL2.GL_REPLACE);

// to use with lighting

gl.glTexEnvf(GL2.GL_TEXTURE_ENV,
GL2.GL_TEXTURE_ENV_MODE,
             GL2.GL_MODULATE);
```

Specular Highlights

```
// to make specular highlights
// be unaffected by the
// texture set

gl.glLightModeli(

GL2.GL_LIGHT_MODEL_COLOR_CONTROL,
GL2.GL_SEPARATE_SPECULAR_COLOR);
```

Loading textures in JOGL

```
int nTex = 1;
int[] textures = new int[nTex];

//get texture id - release when finished

gl.glGenTextures(nTex, textures, 0);

// Use this texture - set current
texture

gl.glBindTexture(
    GL.GL_TEXTURE_2D, textures[0]);
```

Loading textures in JOGL

```java
// Loading data from a file -
// make sure width and height of
// file are a power of 2

GLProfile glp =
        GLProfile.getDefault();

TextureData data =
    TextureIO.newTextureData(
        glp, file, false, "png");
```

Loading textures in JOGL

```java
// Setting data to current texture
gl.glTexImage2D(
    GL2.GL_TEXTURE_2D,
    0,// level of detail: 0 = base
    data.getInternalFormat(),
    data.getWidth(),
    data.getHeight(),
    0,  // border (must be 0)
    data.getPixelFormat(),
    data.getPixelType(),
    data.getBuffer());
```

Texture mapping

When we rasterise an image, we colour each pixel in a polygon by interpolating the texture coordinates of its vertices.

Standard bilinear interpolation does not work because it fails to take into account foreshortening effects in tilted polygons.

Luckily this is done by OpenGL for us

Foreshortening

Unequally spaced
in world space

Equally spaced pixels
in screen space

perspective
camera
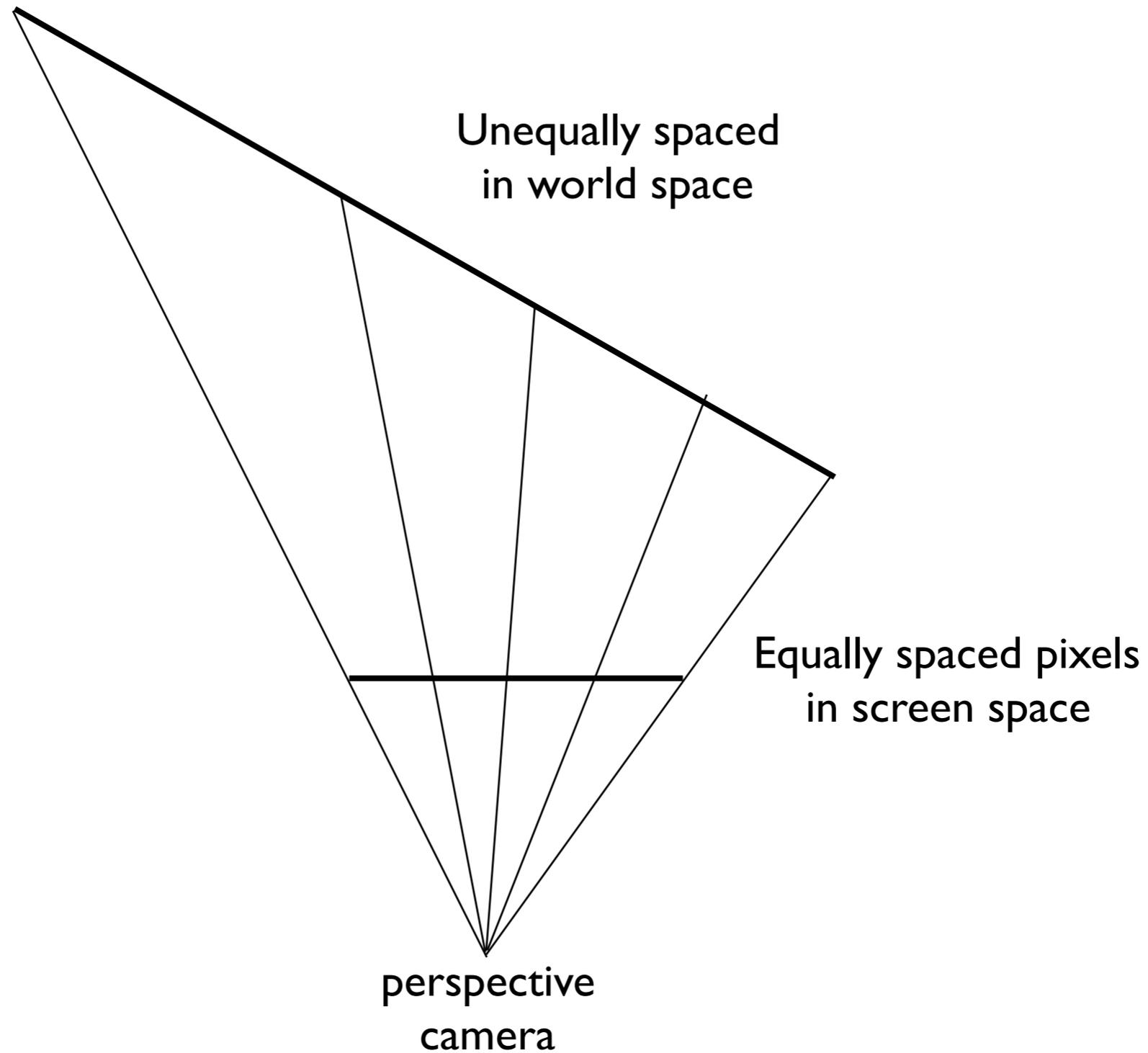
Rendering the Texture

Linear vs. correct interpolation example:

a) linear interpolation

b) correct interpolation
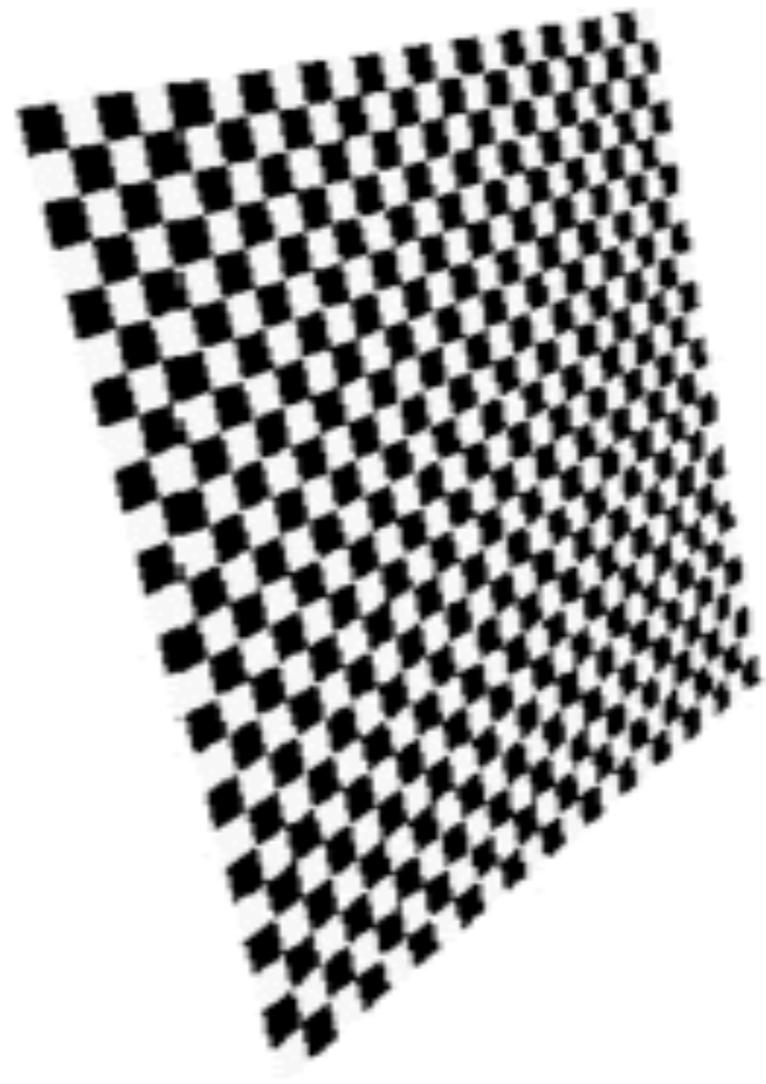
Hyperbolic interpolation

We want texture coordinates to interpolate linearly in world space.

But the perspective projection distorts the depth coordinate so that

linear interpolation $\neq$ linear interpolation
in screen space in world space

Hyperbolic interpolation fixes this.

Mathematical details in textbook if desired.

OpenGL Hints

`gl.glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST)`

`gl.glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_FASTEST)`

If perspective-corrected parameter interpolation is not efficiently supported by the hinting GL_FASTEST can result in simple linear interpolation

3D textures

We can also make 3D textures by adding an extra texture coordinate.

Imagine a volume of space with different colours at each point, e.g. a block of wood.

This eliminates weird seams and distortions when a 2D texture is wrapped on a curve 3D surface.

Magnification

Normal bitmap textures have finite detail.

If we zoom in close we can see individual texture pixels (texels).

If the camera is close enough to a textured polygon multiple screen pixels may map to the same texel.

This results in "pixelated" effects.

Magnification

Magnification

The alignment is probably not exact.

texels

pixels
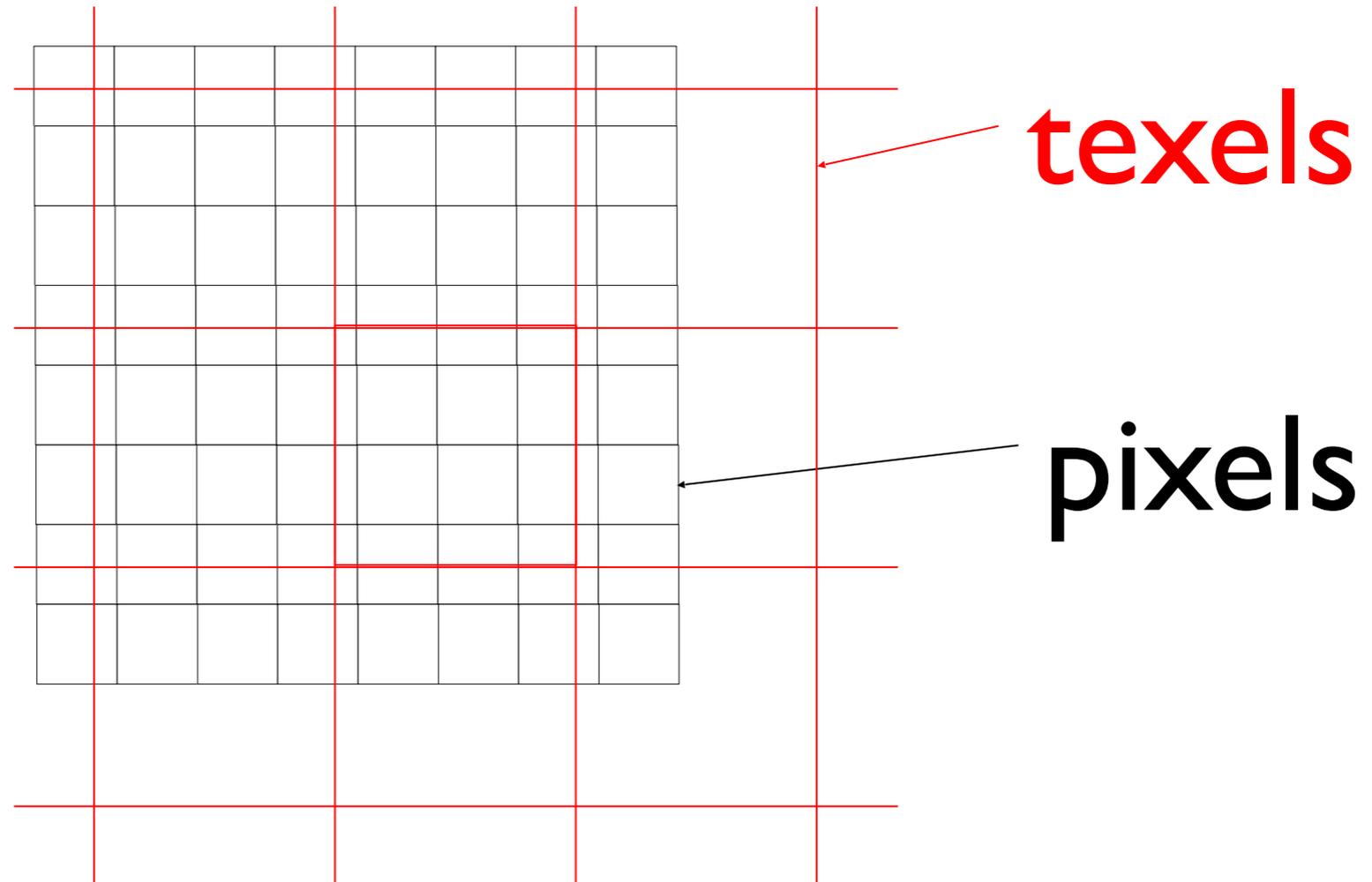
Nearest Texel

Find the nearest texel.

texels

pixels
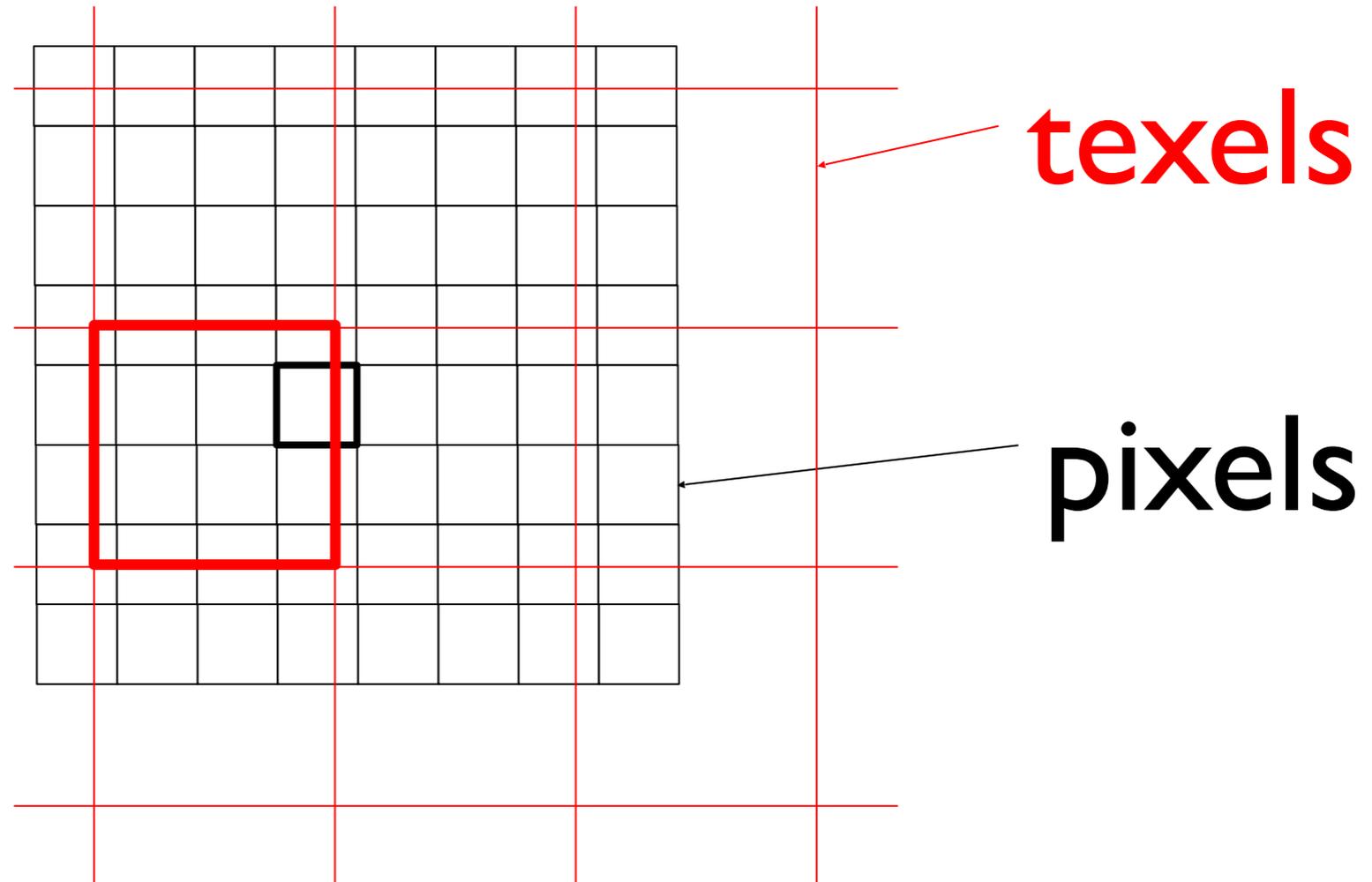
Bilinear Filtering

Find the nearest four texels and use bilinear interpolation over them

texels

pixels
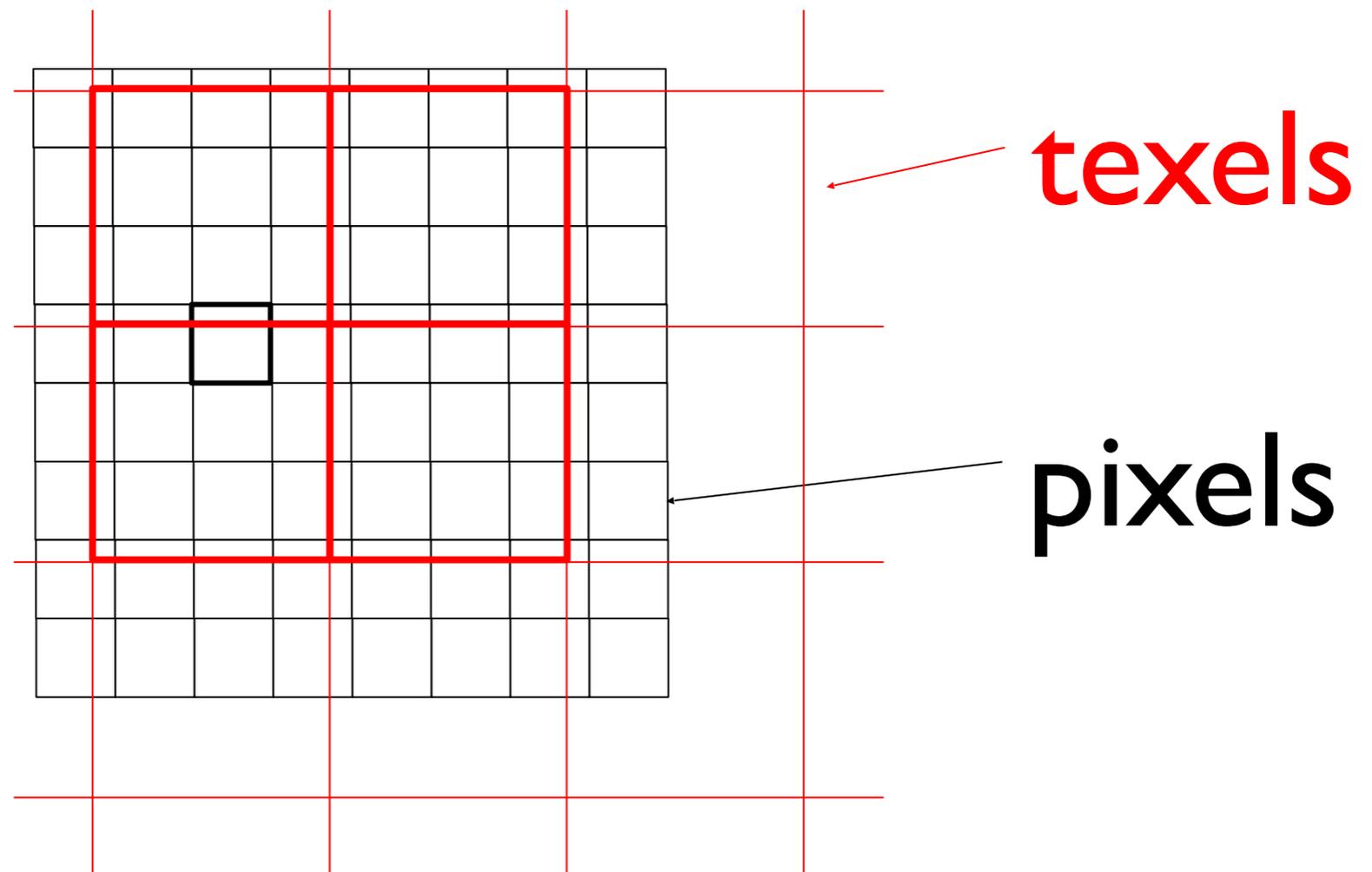
Bilinear Filtering

No filtering
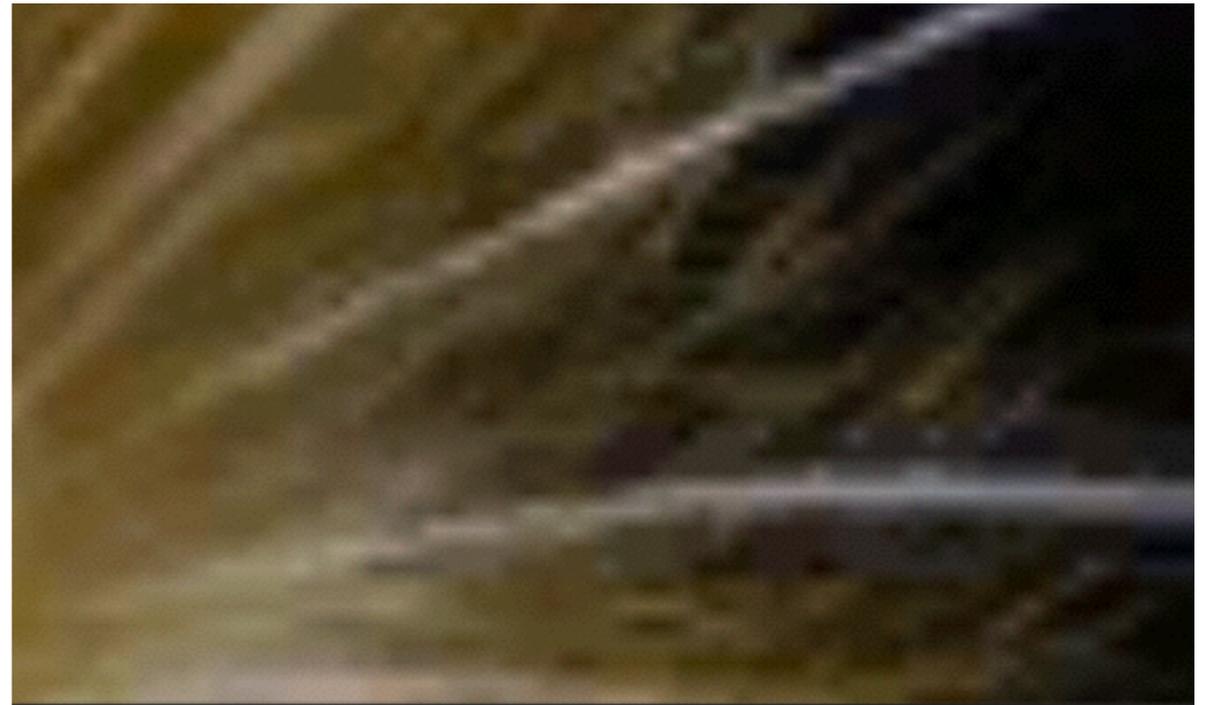
Filtering

Magnification Filtering

```
//bilinear filtering
gl.glTexParameteri(
GL.GL_TEXTURE_2D,
GL.GL_TEXTURE_MAG_FILTER,
GL.GL_LINEAR);
// no bilinear filtering
gl.glTexParameteri(
GL.GL_TEXTURE_2D,
GL.GL_TEXTURE_MAG_FILTER,
GL.GL_NEAREST);
```

Minification

Problems also occur when we zoom out too far from a texture.

We can have more than one texel mapping to a pixel.

If image pixels line up with regularities in the texture, strange artefacts appear in the output such as moire patterns or shimmering in an animation

Minification

Again, the alignment is not exact.
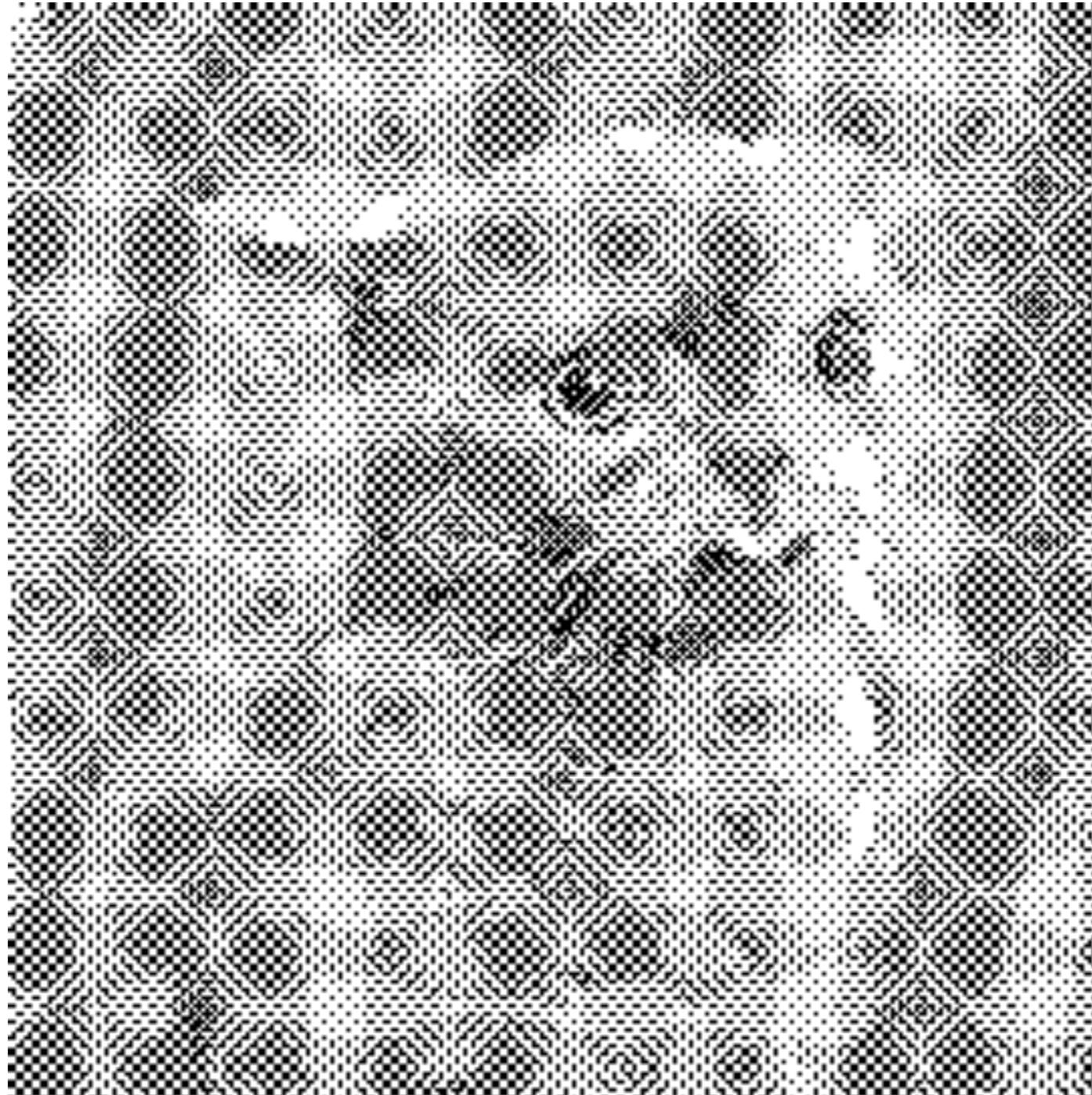
texels

one pixel

Minification

Aliasing

This effect is called aliasing. It occurs when samples are taken from an image at a lower resolution than repeating detail in the image.

texels
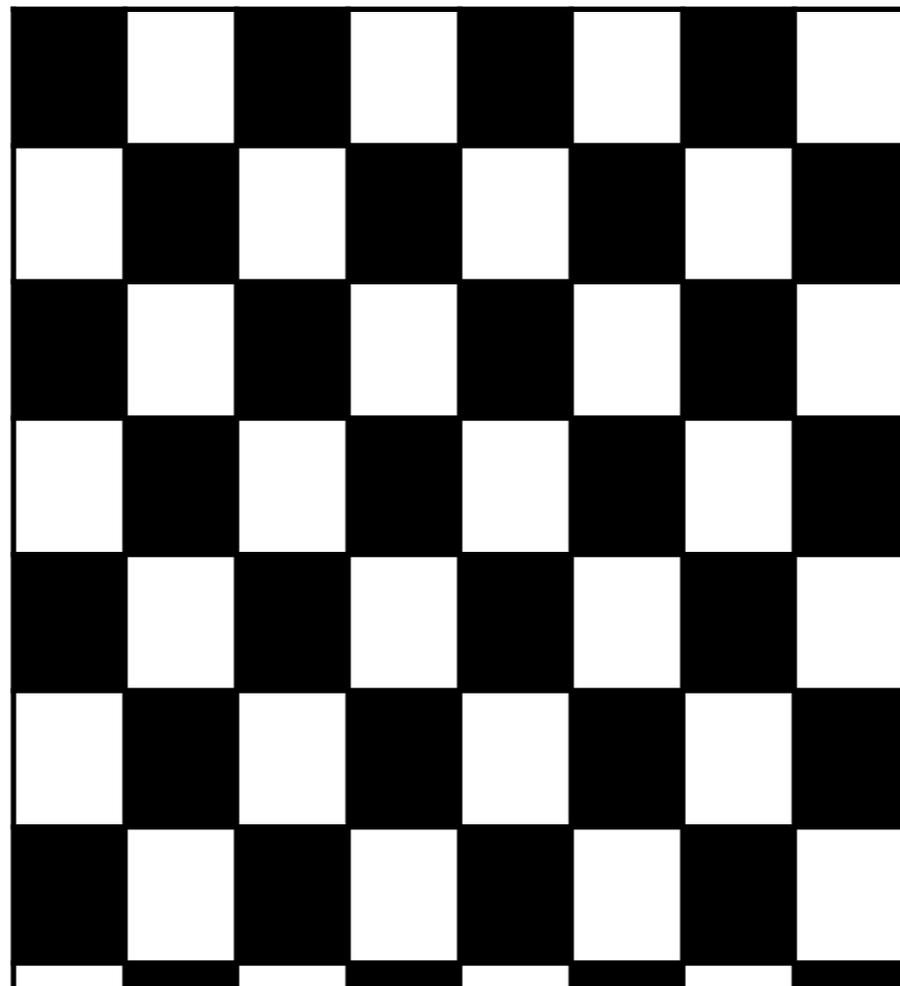
Aliasing

This effect is called aliasing. It occurs when samples are taken from an image at a lower resolution than repeating detail in the image.

pixels
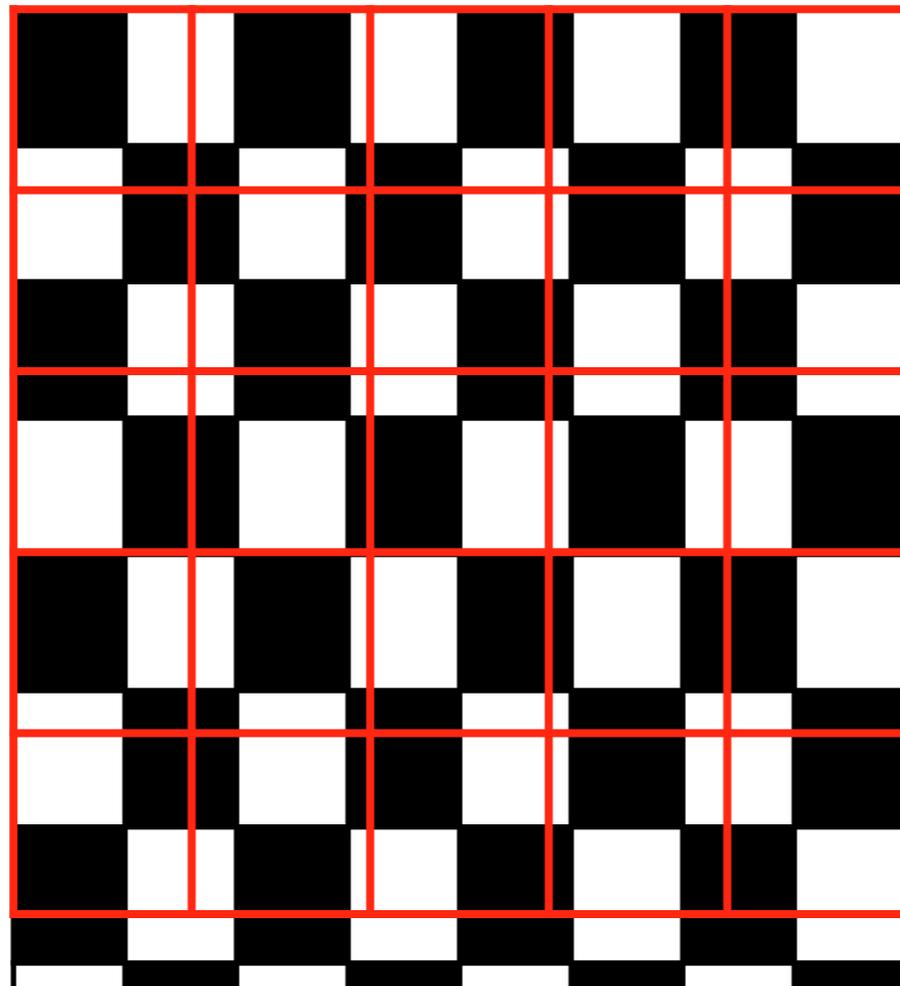
Aliasing

This effect is called aliasing. It occurs when samples are taken from an image at a lower resolution than repeating detail in the image.

samples
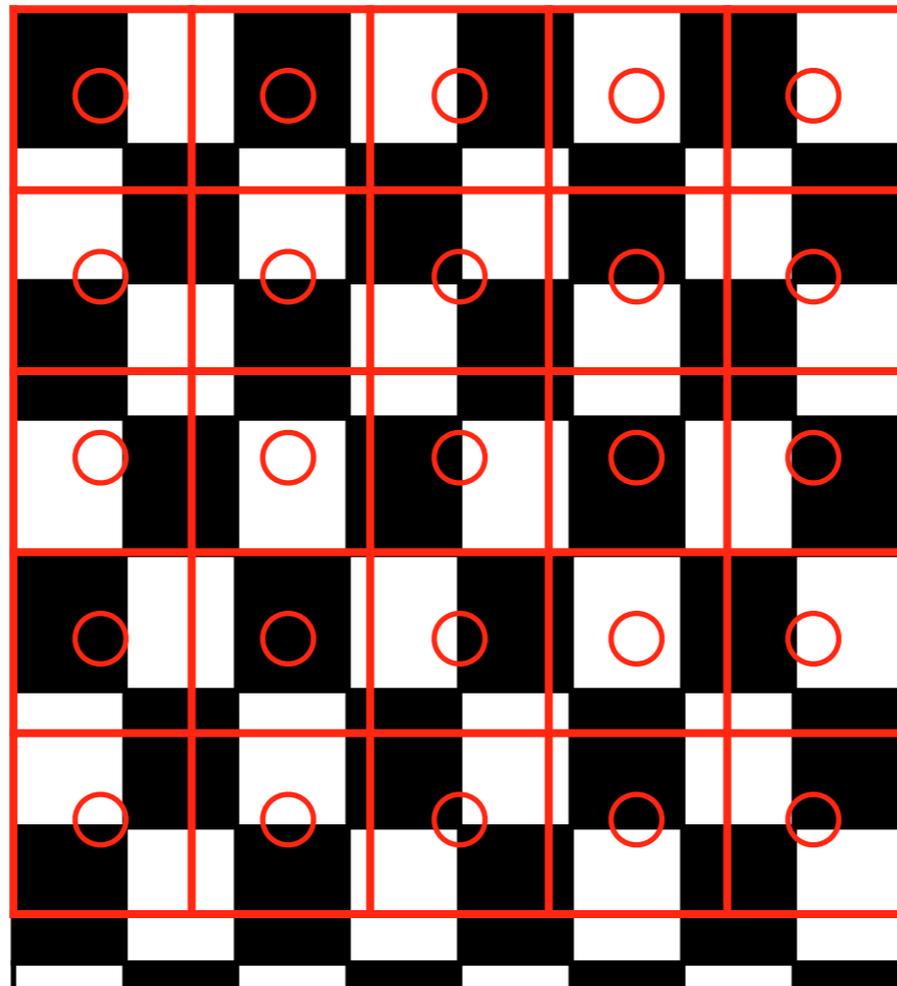
Aliasing

This effect is called aliasing. It occurs when samples are taken from an image at a lower resolution than repeating detail in the image.

result
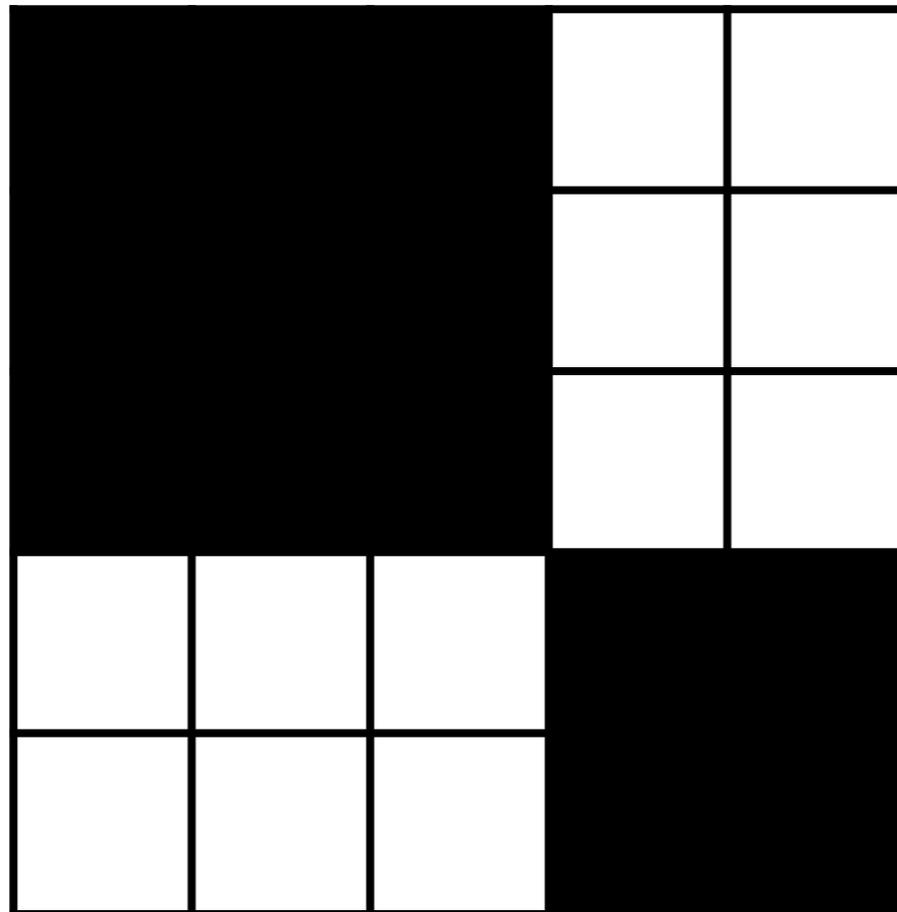
Filtering

The problem is that one screen pixel overlaps multiple texels but is taking its value from only one of those texels.

A better approach is to average the texels that contribute to that pixel.

Doing this on the fly is expensive.

MIP mapping

Mipmaps are precomputed low-res versions of a texture.

Starting with a 512x512 texture we compute and store 256x256, 128x128, 64x64, 32x32, 16x16, 8x8, 4x4, 2x2 and 1x1 versions.

This takes total memory = 4/3 original.

Using mipmaps

The simplest approach is to use the next smallest mipmap for the required resolution.

E.g. To render a 40x40 pixel image, use the 32x32 pixel mipmap and magnify using magnification filter

Trilinear filtering

A more costly approach is trilinear filtering:
- Use bilinear filtering to compute pixel values based on the next highest and the next lowest mipmap resolutions.
- Interpolate between these values depending on the desired resolution.

Minification Filtering

```
//bilinear filtering with no mipmaps
gl.glTexParameteri( GL.GL_TEXTURE_2D,
    GL.GL_TEXTURE_MIN_FILTER,
GL.GL_LINEAR);
// no bilinear filtering with no
mipmaps
gl.glTexParameteri( GL.GL_TEXTURE_2D,
    GL.GL_TEXTURE_MIN_FILTER,
GL.GL_NEAREST);
```

Generating Mip-Maps

```
//get opengl to auto-generate

//mip-maps.
gl.glGenerateMipmap(GL2.GL_TEXTURE_2D);
// Must make sure you set the

// appropriate min filters

// once you have done this
```

MipMap Minification Filtering

```
// use nearest mipmap
gl.glTexParameteri(
GL.GL_TEXTURE_2D,
GL.GL_TEXTURE_MIN_FILTER,
    GL.GL_NEAREST_MIPMAP_NEAREST);
// use trilinear filtering
gl.glTexParameteri(
GL.GL_TEXTURE_2D,
    GL.GL_TEXTURE_MIN_FILTER,
    GL.GL_LINEAR_MIPMAP_LINEAR);
```

Aniso Filtering

If a polygon is on an oblique angle away from the camera, then minification may occur much more strongly in one dimension than the other.
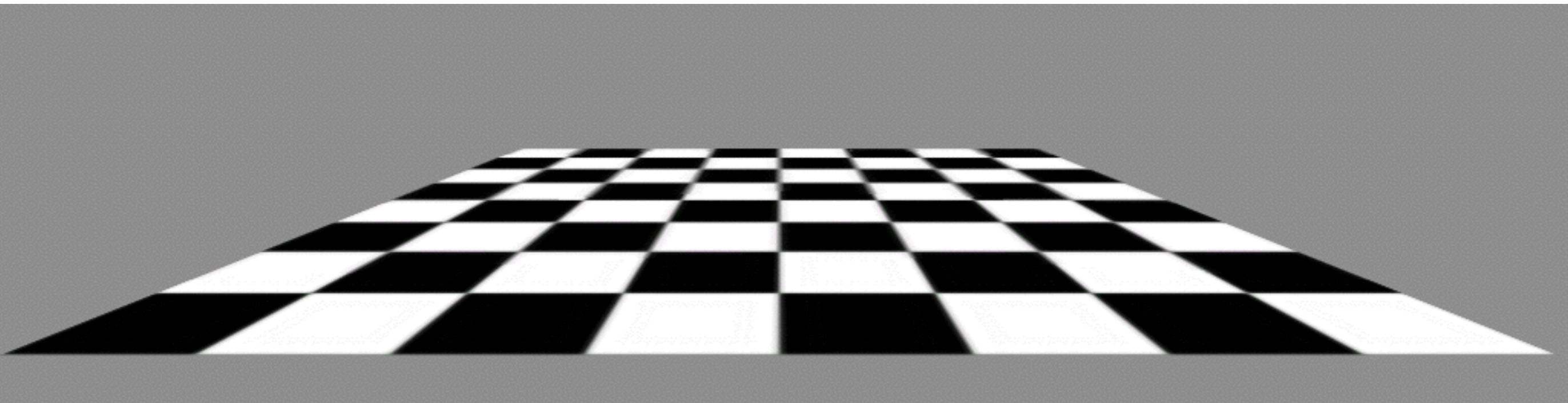
Aniso filtering

Anisotropic filtering is filtering which treats the two axes independently.

```
float fLargest[] = new float[1];
gl.glGetFloatv(GL.GL_MAX_TEXTURE_MAX_ANIS
OTROPY_EXT, fLargest,0);

gl.glTexParameterf(GL.GL_TEXTURE_2D,
GL.GL_TEXTURE_MAX_ANISOTROPY_EXT,
fLargest[0]);
```

Aniso Filtering
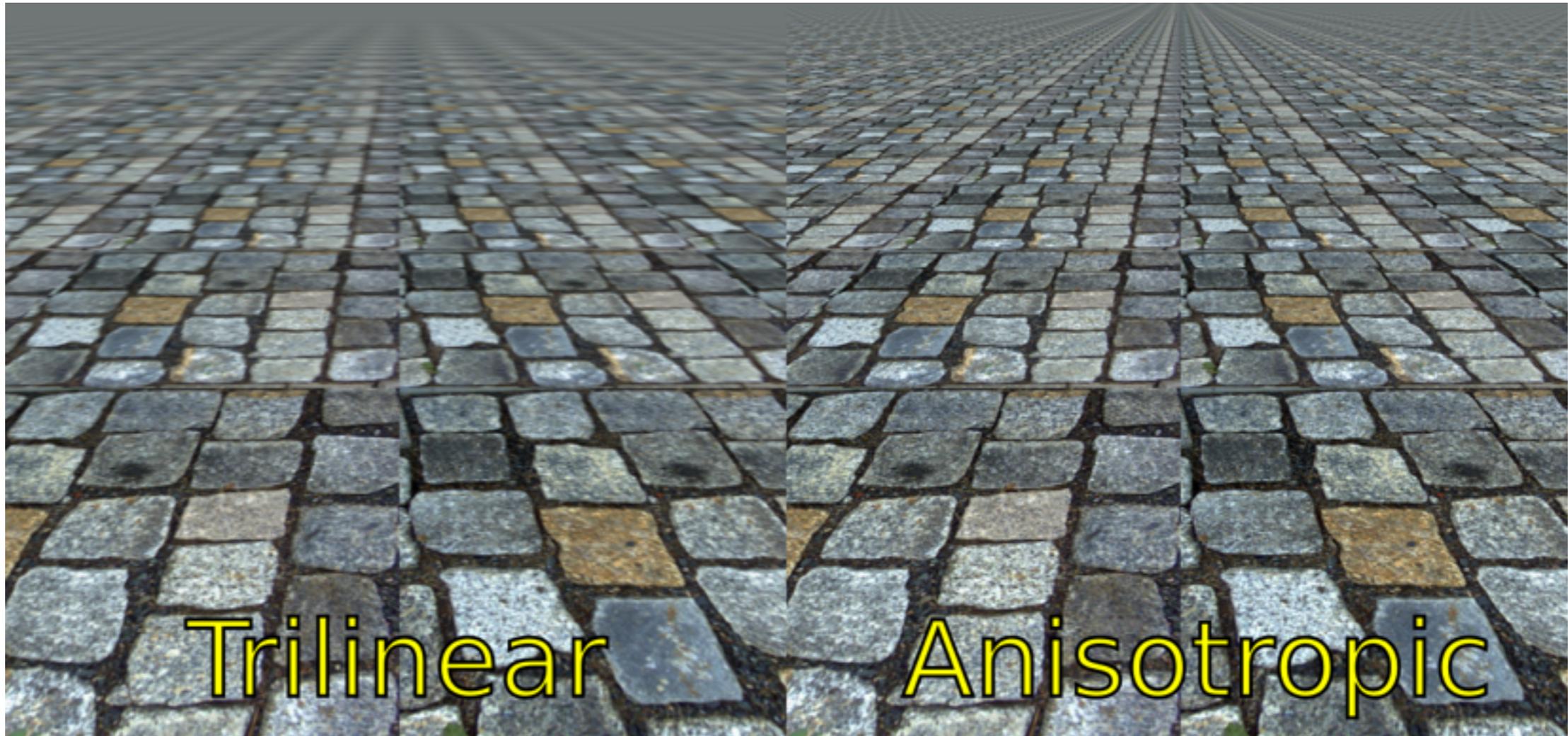
RIP Mapping

RIP mapping is an extension of MIP mapping which down-samples each axis and is an approach to anisotropic filtering

So a 256x256 image has copies at:

256x128, 256x64, 256x32, 256x16, ...,
128x256, 128x128, 128x64, ...
64x256, 64x128, etc.

RIP Mapping

Limitations of RIP Mapping:

- Does not handle diagonal anisotropy.
- More memory required for RIP maps (4 times as much).
- Not implemented in OpenGL

Multi-texturing

Can use more than one texture on the same fragment.

```
gl.glActiveTexture(GL2.GL_TEXTURE0);
gl.glBindTexture(GL2.GL_TEXTURE_2D, texId1);


gl.glActiveTexture(GL2.GL_TEXTURE1);
gl.glBindTexture(GL2.GL_TEXTURE_2D, texId2);

gl.glTexEnvi(GL2.GL_TEXTURE_ENV,
GL2.GL_TEXTURE_ENV_MODE, GL2.GL_COMBINE);
```

Multi-texturing

GL_COMBINE, instead of default GL_REPLACE, indicates that the first texture unit combines with the zeroth by application of a texture combiner function.

An example texture combiner function is

```
gl.glTexEnvi(GL TEXTURE ENV, GL COMBINE RGB,
             GL INTERPOLATE);

//Uses Arg0 * Arg2 + Arg1 * (1-Arg2)
//See code for setting up the Arg0,Arg1,Arg2 values
```

Multi-texturing

Need to define different sets of texture coordinates

```
gl.glMultiTexCoord2d(GL2.GL_TEXTURE0,
0.5, 1.0);
gl.glMultiTexCoord2d(GL2.GL_TEXTURE1,
0.5, 1.0);

gl.glVertex3d(….);
```

Animated textures

Animated textures can be achieved by loading multiple textures and using a different one on each frame.
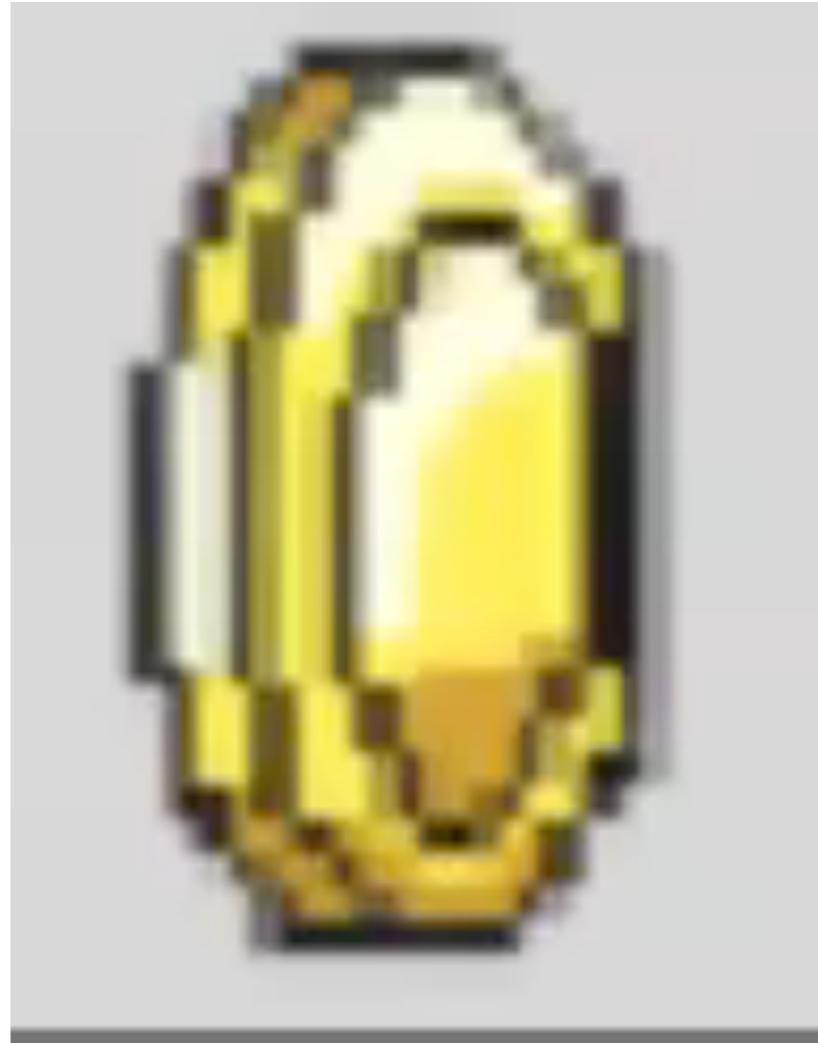
Textures and Shaders

Vertex Shader

Simply pass through the texture coords to the fragment shader (they will be interpolated).

```
out vec2 texCoord;

void main(void){
 //gl_MultiTexCoord0 has texture coords
 texCoord = vec2(gl_MultiTexCoord0);
}
```

Textures and Shaders

Fragment Shader

```
//passed in by vertex shader

in vec2 texCoord;

//texture variable passed in by jogl program

uniform sampler2D texUnit1;

//This would implement replace mode

gl_FragColor = texture(texUnit1,texCoord);
```

Textures and Shaders

Fragment Shader

```
//For modulate with simple coloured vertices

gl_FragColor = texture(texUnit1,texCoord) *
gl_Color;

//For modulate with separate specular with

//lighting

gl_FragColor =

texture(texUnit1,texCoord) * (emissive + ambient +
diffuse) + specular
```

Textures and Shaders

```
//In your jogl program, link texture variables

texUnitLoc =

gl.glGetUniformLocation(shaderprog,"texUnit1");

// By default unless we are using mult-texturing

// we always bind to texture0

gl.glUniform1i(texUnitLoc , 0);
```

Exercises

How can we modify our texturing example to use VBOs instead of immediate mode?

Rendering to a texture

A common trick is to set up a camera in a scene, render the scene into an offscreen buffer, then copy the image into a texture to use as part of another scene.

E.g. Implementing a security camera in a game.

```
//In openGL you can use

gl.glCopyTexImage2D(…);
```

lostandtaken.com

opengameart.org/textures

textures.com

Reflection

To do better quality reflections we need to compute where the reflected light is coming from in the scene.

Object seen

Reflective object

Eye
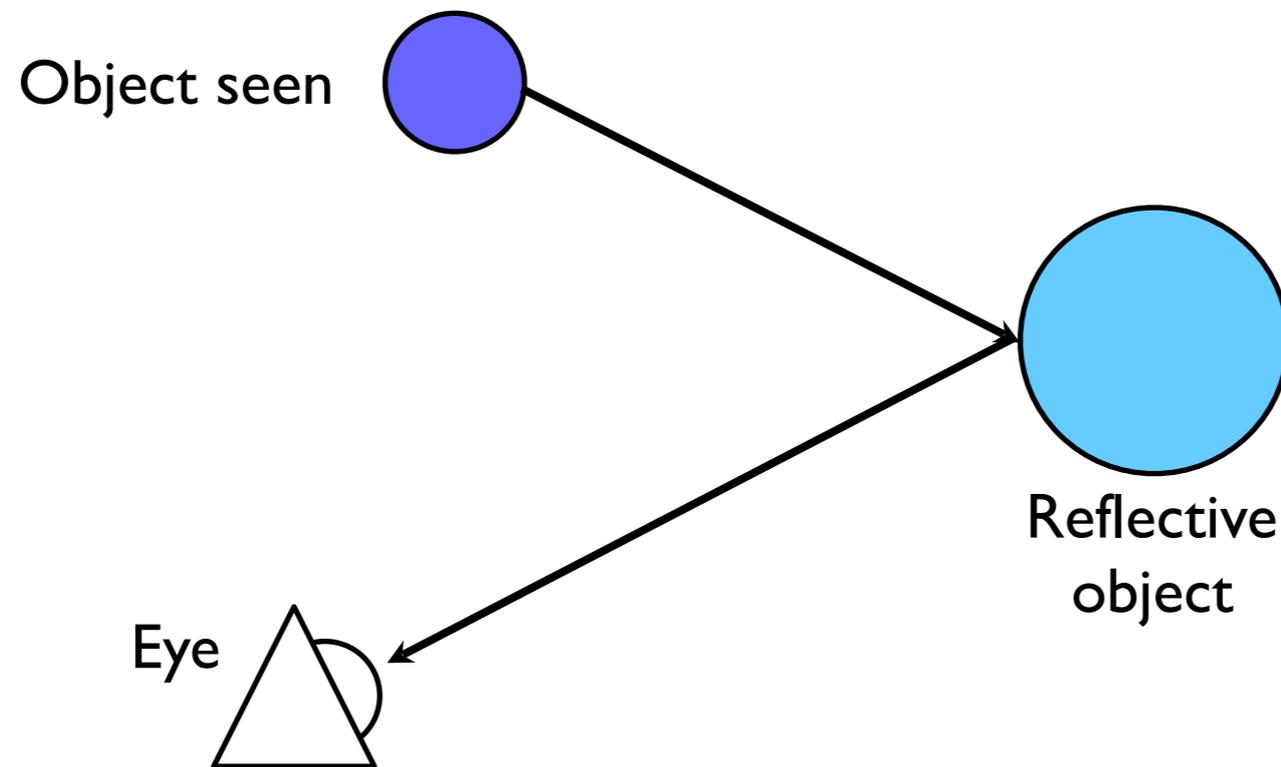
Reflection mapping

Doing this in general is expensive, but we can do a reasonable approximation with textures:

- Generate a cube that encloses the reflective object.
- Place a camera at the centre of the cube and render the outside world onto the faces of the cube.
- Use this image to texture the object

Reflection mapping

Reflective
object

Reflection mapping

Cube

Reflection mapping

Render scene onto cube

Camera

Reflection mapping

Repeat for each face

Camera

Reflection mapping

Repeat for
each face

Camera

Reflection mapping

Repeat for each face
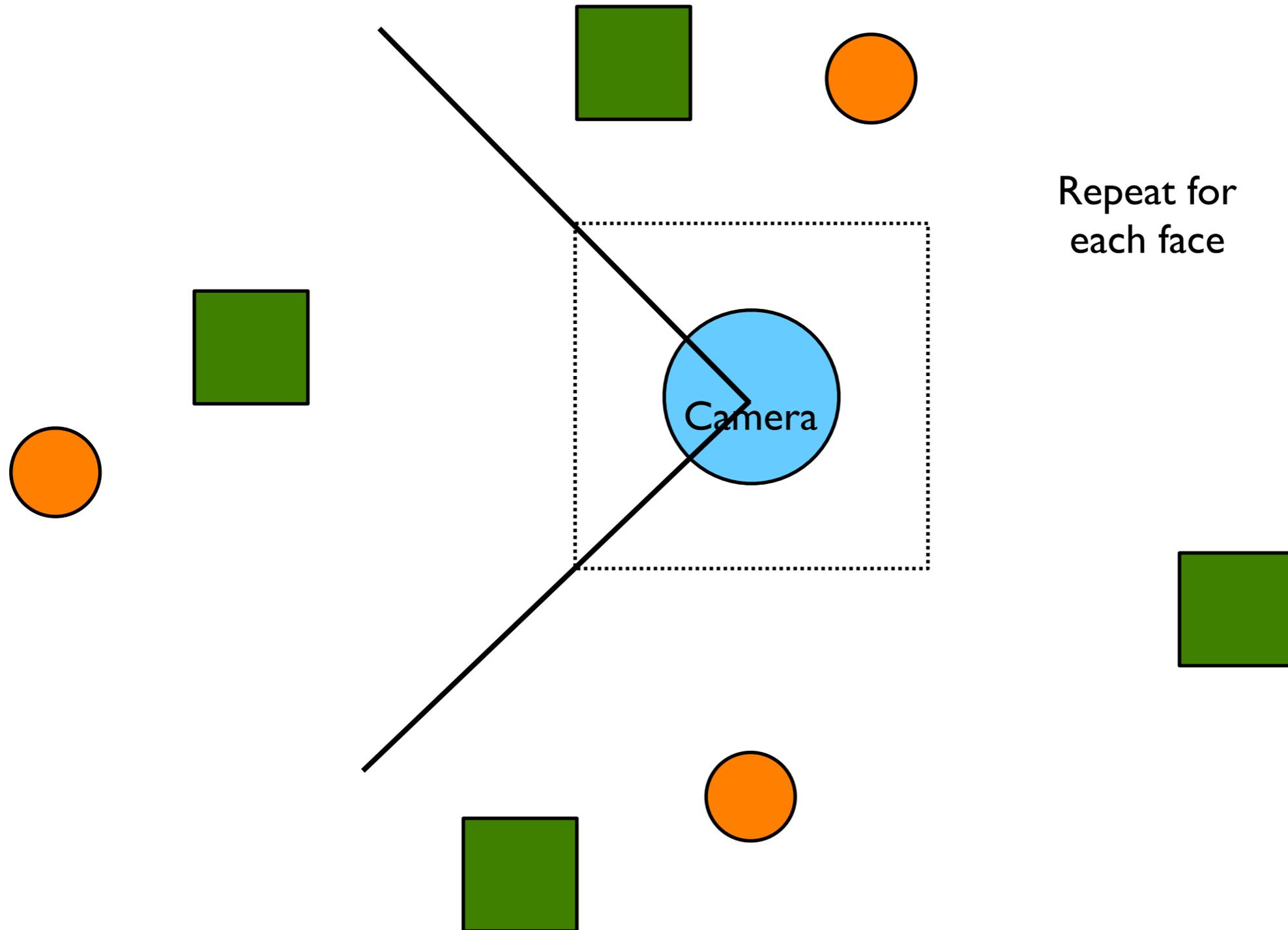
Camera

Reflection mapping

To apply the reflection-mapped texture to the object we need to calculate appropriate texture coordinates.

We do this by tracing a ray from the camera, reflecting it off the object and then calculating where it intersects the cube.

Reflection mapping

point

normal

Eye

View
vector

Cube
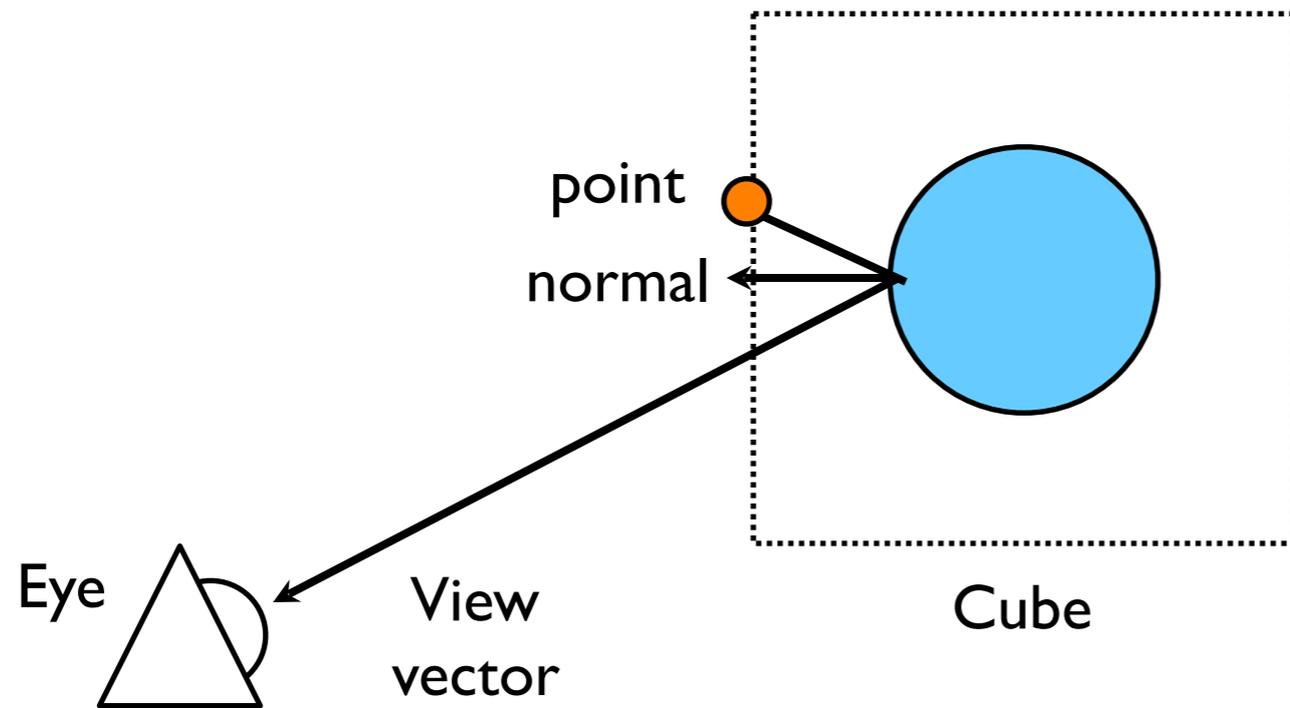
Reflection mapping

Pros:

- Produces reasonably convincing polished metal surfaces and mirrors

Reflection mapping

Cons:

- Expensive: Requires 6 additional render passes per object
- Angles to near objects are wrong.
- Does not handle self-reflections or recursive reflections.

OpenGL

 OpenGL has built in support for fast approximate reflection mapping (cube mapping).

http://www.nvidia.com/object/cube_map_ogl_tutorial.html

OpenGL also has sphere mapping support, although this usually produces more distortion and is not as effective as cube mapping.

Shadows

Our lighting model does not currently produce shadows.

We need to take into account whether the light source is occluded by another object.
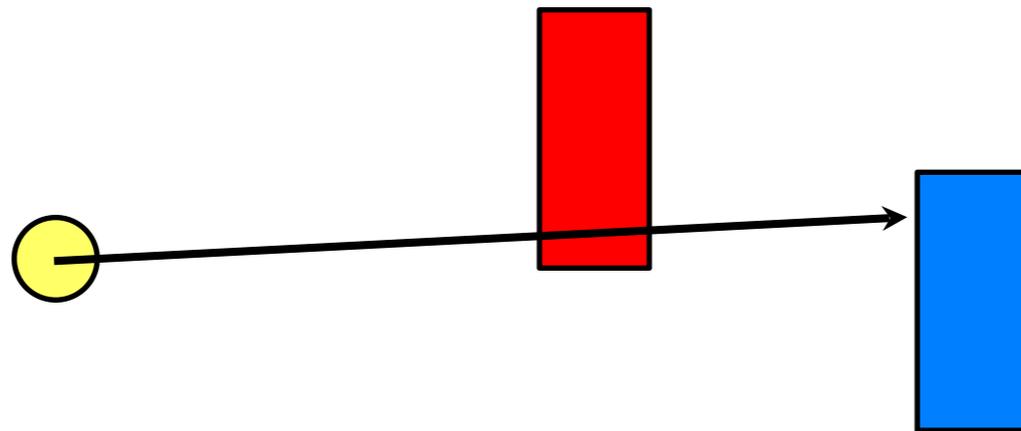
Shadow buffering

One solution is to keep a shadow buffer for each light source.

The shadow buffer is like the depth buffer, it records the distance from the light source to the closest object in each direction.
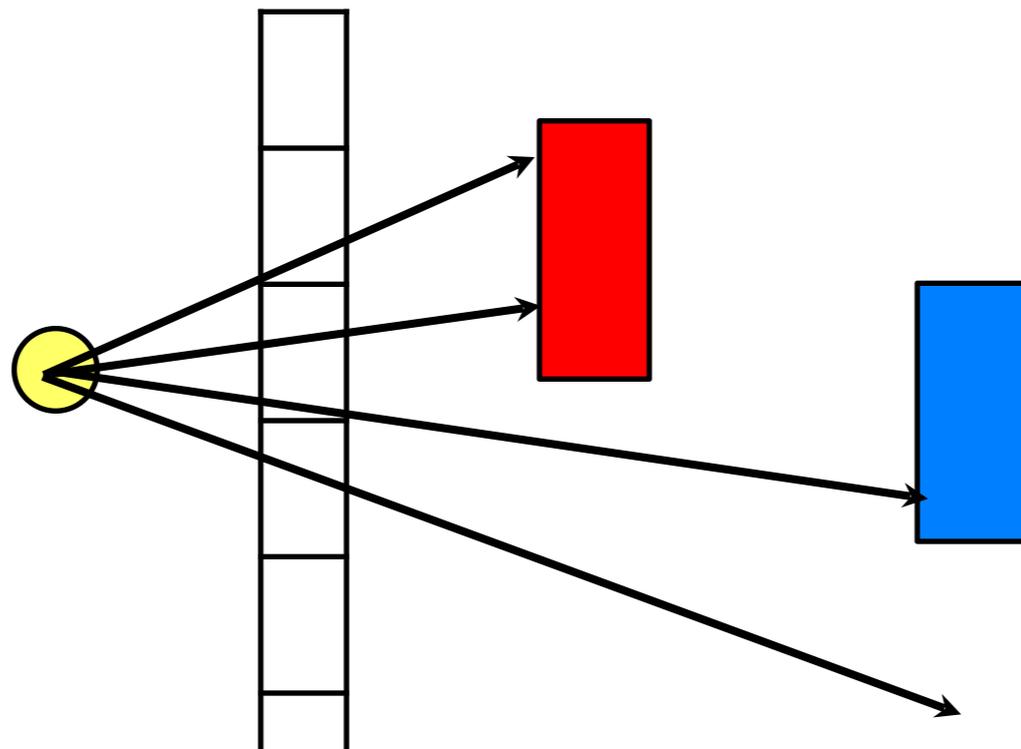
Shadow buffer

Shadow rendering is usually done in multiple passes:

1. Render the scene from the light's viewpoint capturing only z-info in shadow buffer (color buffer turned off)

2. Render the scene from camera's viewpoint in ambient light first

3. Render the scene from camera's point of view for each light and add the pixel values for lit objects

Shadow buffer

When rendering a point P:

- Project the point into the light's clip space.
- Calculate the index (i,j) for P in the shadow buffer
- Calculate the pseudodepth d relative to the light source
- If shadow[i,j] < d then P is in the shadow

Shadow buffer

Pros:

- Provides realistic shadows
- No knowledge or processing of the scene geometry is required

Shadow buffer

Cons:

- More computation
- Shadow quality is limited by precision of shadow buffer. This may cause some aliasing artefacts.
- Shadow edges are hard.
- The scene geometry must be rendered once per light in order to generate the shadow map for a spotlight, and more times for an omnidirectional point light.

OpenGL

http://www.paulsprojects.net/tutorials/smt/smt.html

Light Mapping

If our light sources and large portions of the geometry are static then we can precompute the lighting equations and store the results in textures called light maps.

This process is known as baked lighting.

Light Mapping

Pros:

- Sophisticated lighting effects can be computed at compile time, where speed is less of an issue.

Light mapping

Cons:

- Memory and loading times for many individual light maps.
- Not suitable for dynamic lights or moving objects.
- Potential aliasing effects depending on the resolution of the light maps.

Normal mapping

When we interpolate normals in a Phong shader we are assuming that the surface of the polygon is smoothly curved.

What if the surface is actually rough with many small deformities?

Putting a rough texture on a smooth flat surface looks wrong.

Normal mapping

One solution would be to increase the number of polygons to represent all the deformities, but this is computationally unfeasible for most applications.

Instead we use textures called normal maps to simulate minor perturbations in the surface normal.

Normal maps

Rather than arrays of colours, normal maps can be considered as arrays of vectors. These vectors are added to the interpolated normals to give the appearance of roughness.

Vertex normals

Normal maps

Rather than arrays of colours, normal maps can be considered as arrays of vectors. These vectors are added to the interpolated normals to give the appearance of roughness.

Interpolated
fragment normals
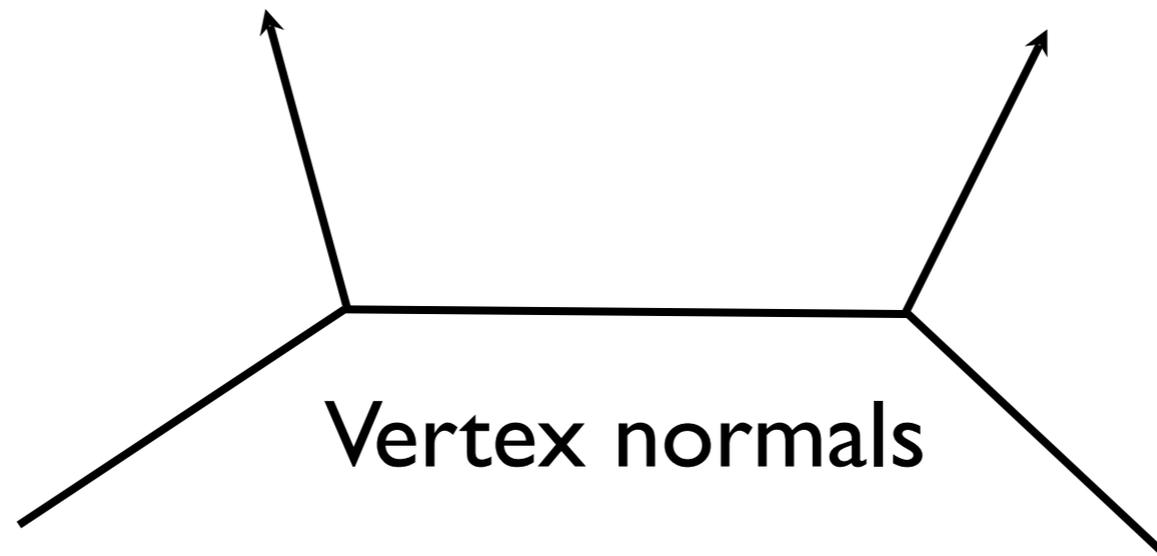
Normal maps

Rather than arrays of colours, normal maps can be considered as arrays of vectors. These vectors are added to the interpolated normals to give the appearance of roughness.
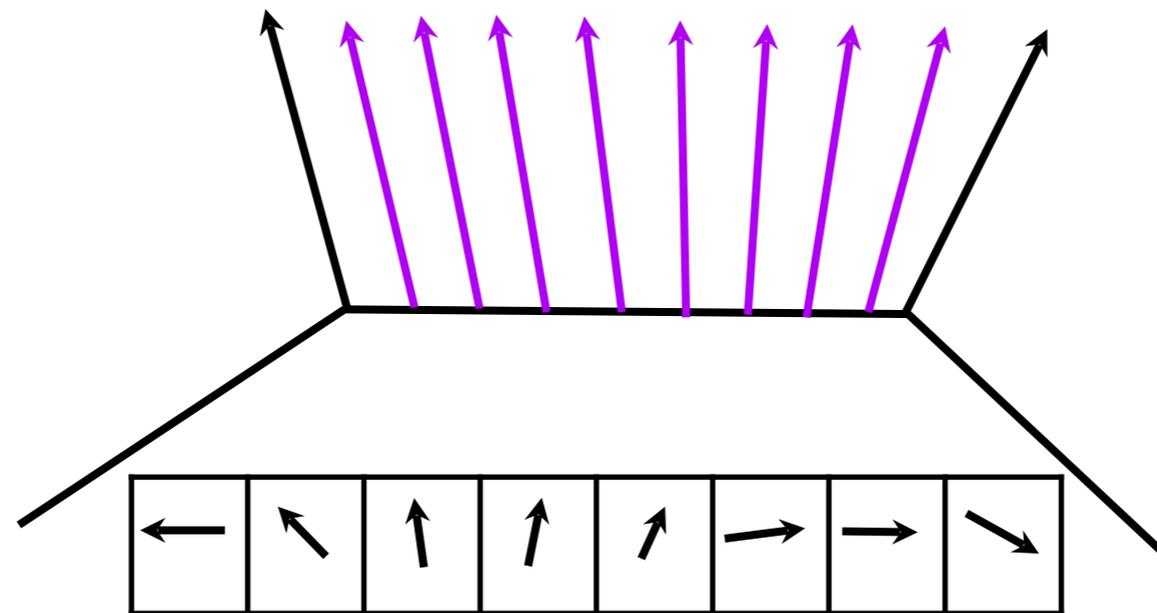
Normal map

Normal maps

Rather than arrays of colours, normal maps can be considered as arrays of vectors. These vectors are added to the interpolated normals to give the appearance of roughness.

Perturbed normals
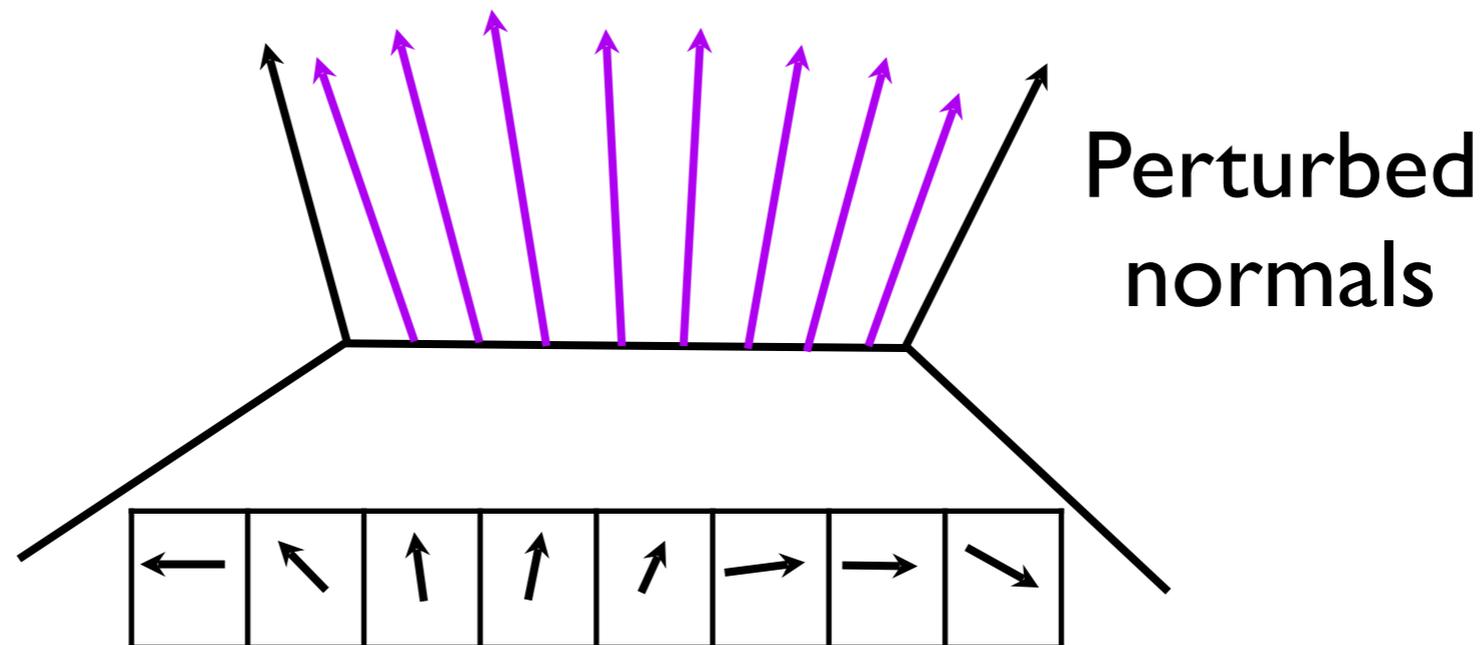
Normal maps

Pros:

- Provide the illusion of surface texture

Cons:

- Does not affect silhouette
- Does not affect occlusion calculation

Normal Mapping

OpenGL

http://www.opengl-tutorial.org/intermediate-tutorials/tutorial-13-normal-mapping/

http://www.terathon.com/code/tangent.html