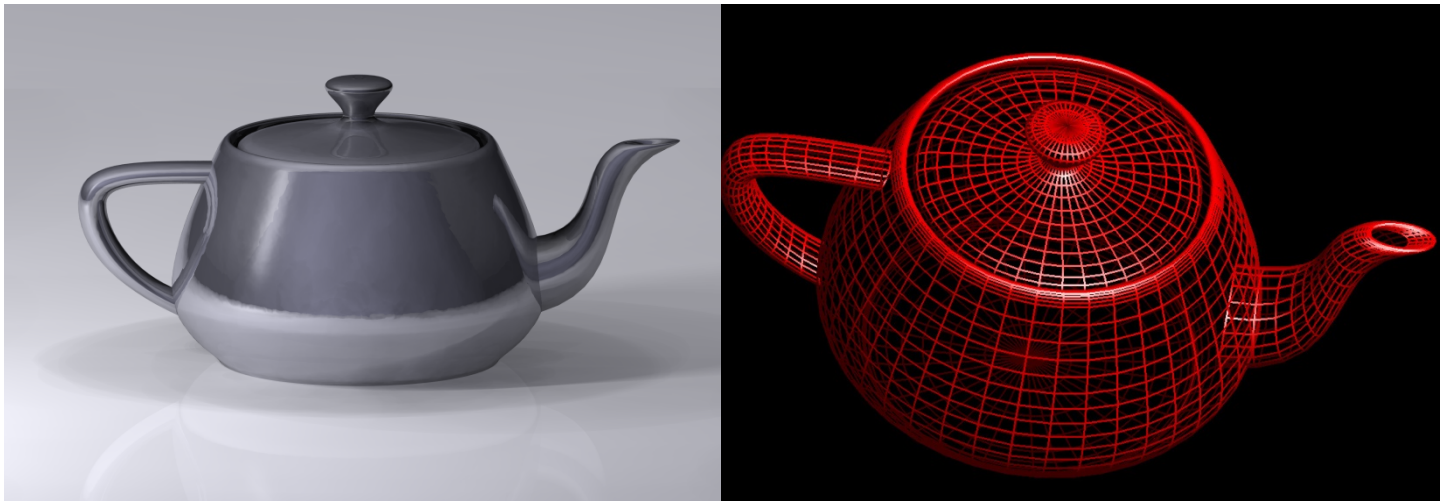


Foundations of Computer Graphics

Saurabh Ray

Read sections 7.3 - 7.9 of the textbook.

Tuesday, April 7 instead of April 2.

Model File Format

Various formats are used for representing 3D models.

Common formats: 3D Studio Max (.max, .3ds), Blender (.blend), Collada (.dae), Wavefront (.obj)

Wavefront OBJ example: `cube.obj`

```
# Blender v2.60 (sub 0) OBJ File: ''
# www.blender.org
mtllib cube.mtl
o Cube
v 1.000000 -1.000000 -1.000000
v 1.000000 -1.000000 1.000000
v -1.000000 -1.000000 1.000000
v -1.000000 -1.000000 -1.000000
v 1.000000 1.000000 -1.000000
```

Lines starting with # are comments.

Materials are described in the file `cube.mtl`

Name of the object is 'Cube'.

Each line starting with a 'v' describes a vertex.

The vertices are numbered in order starting from 1.

Model File Format

```
v 1.000000 1.000000 1.000001
v -1.000000 1.000000 1.000000
v -1.000000 1.000000 -1.000000
usemtl Material
f 1 2 3 4
f 5 8 7 6
f 2 6 7 3
f 3 7 8 4
f 5 1 4 8
usemtl Material.001
f 1 5 6 2
```

Use the material called 'Material' for the following

Each line starting with an 'f' describes a face.

The numbers are vertex numbers.

In this example each face is a quad.

Use the material called 'Material.001' for the following

Model File Format

`cube.mtl`:

```
# Blender MTL File: ''
# Material Count: 2
newmtl Material
Ka 0.000000 0.000000 0.000000
Kd 1.000000 0.000000 0.000000
Ks 0.000000 0.000000 0.000000
Ns 96.078431
Ni 1.000000
d 1.000000
illum 0
newmtl Material.001
Ka 0.000000 0.000000 0.000000
Kd 1.000000 0.450000 0.000000
Ks 0.000000 0.000000 0.000000
Ns 96.078431
Ni 1.000000
d 1.000000
illum 0
```

Ambient color

Diffuse color

Specular color

Model File Format

More details:

```
# List of geometric vertices, with (x,y,z[,w])
# coordinates, w is optional and defaults to 1.0.
 v 0.123 0.234 0.345 1.0
 v ...


# List of texture coordinates, in (u, v [,w])
# coordinates, these will vary between 0 and 1,
# w is optional and defaults to 0.
 vt 0.500 1
 vt ...


# List of vertex normals in (x,y,z) form;
# normals might not be unit vectors.
 vn 0.707 0.000 0.707
 vn ...
```

Model File Format

More details:

```
# Polygonal face element.
# Format: vertex/texture/normal indices.
  f 1 2 3
  f 3/1 4/2 5/3
  f 6/4/1 3/5/3 7/6/5
  f ...
```

Format: `f v1/vt1/vn1 v2/vt2/vn2 v3/vt3/vn3 ...`

`f v1//vn1 v2//vn2 v3//vn3 ...` *vertex and normal coordinates without texture coordinates*

Obj to JSON

We will use a simple script to convert Wavefront Obj files to JSON (Javascript Object Notation) format:

The script is not complete. It ignores several things, including material descriptions but it suffices for our purposes.

It outputs an object in the following format:

```
var obj = {
    positions:  [ ... ],
    normals:    [ ... ],
    texCoords:  [ ... ],
    triangles:  [ ... ]
}
```

Obj to JSON

Output for `cube.obj`:

```
var obj = {
        positions : [
                [ 1, -1, -1 ],
                [ 1, -1, 1 ],
                [ -1, -1, 1 ],
                [ -1, -1, -1 ],
                [ 1, 1, -1 ],
                [ -1, 1, -1 ],
                [ -1, 1, 1 ],
                [ 1, 1, 1 ],
        ],
        triangles : [
                [ 0, 1, 2 ],
                [ 0, 2, 3 ],
                [ 4, 5, 6 ],
                [ 4, 6, 7 ],
                [ 1, 7, 6 ],
                [ 1, 6, 2 ],
                [ 2, 6, 5 ],
                [ 2, 5, 3 ],
                [ 4, 0, 3 ],
                [ 4, 3, 5 ],
                [ 0, 4, 7 ],
                [ 0, 7, 1 ],
        ],
}
```

Obj to JSON

What should the script output for this input?

```
v -1 -1 0
v 1  -1 0
v 1 1 0
v -1 1 0
f 1 2 3 4
```

The script outputs:

```
var obj = {
        positions : [
                [ -1, -1, 0 ],
                [ 1, -1, 0 ],
                [ 1, 1, 0 ],
                [ -1, 1, 0 ],
        ],
        triangles : [
                [ 0, 1, 2 ],
                [ 0, 2, 3 ],
        ],
}
```

Obj to JSON

What should the script output for this input?

```
v -1 -1 0
v 1  -1 0
v 1 1 0
v -1 1 0
vn 0 0 1
vn 0 0 -1
f 1//1 2//1 3//1
f 1//2 3//2 4//2
```

The script outputs:

```
var obj = {
        positions : [
                [ -1, -1, 0 ],
                [ 1, -1, 0 ],
                [ 1, 1, 0 ],
                [ -1, -1, 0 ],
                [ 1, 1, 0 ],
                [ -1, 1, 0 ],
        ],
        normals : [
                [ 0, 0, 1 ],
                [ 0, 0, 1 ],
                [ 0, 0, 1 ],
                [ 0, 0, -1 ],
                [ 0, 0, -1 ],
                [ 0, 0, -1 ],
        ],
        triangles : [
                [ 0, 1, 2 ],
                [ 3, 4, 5 ],
        ],
}
```

Obj to JSON

In the output:

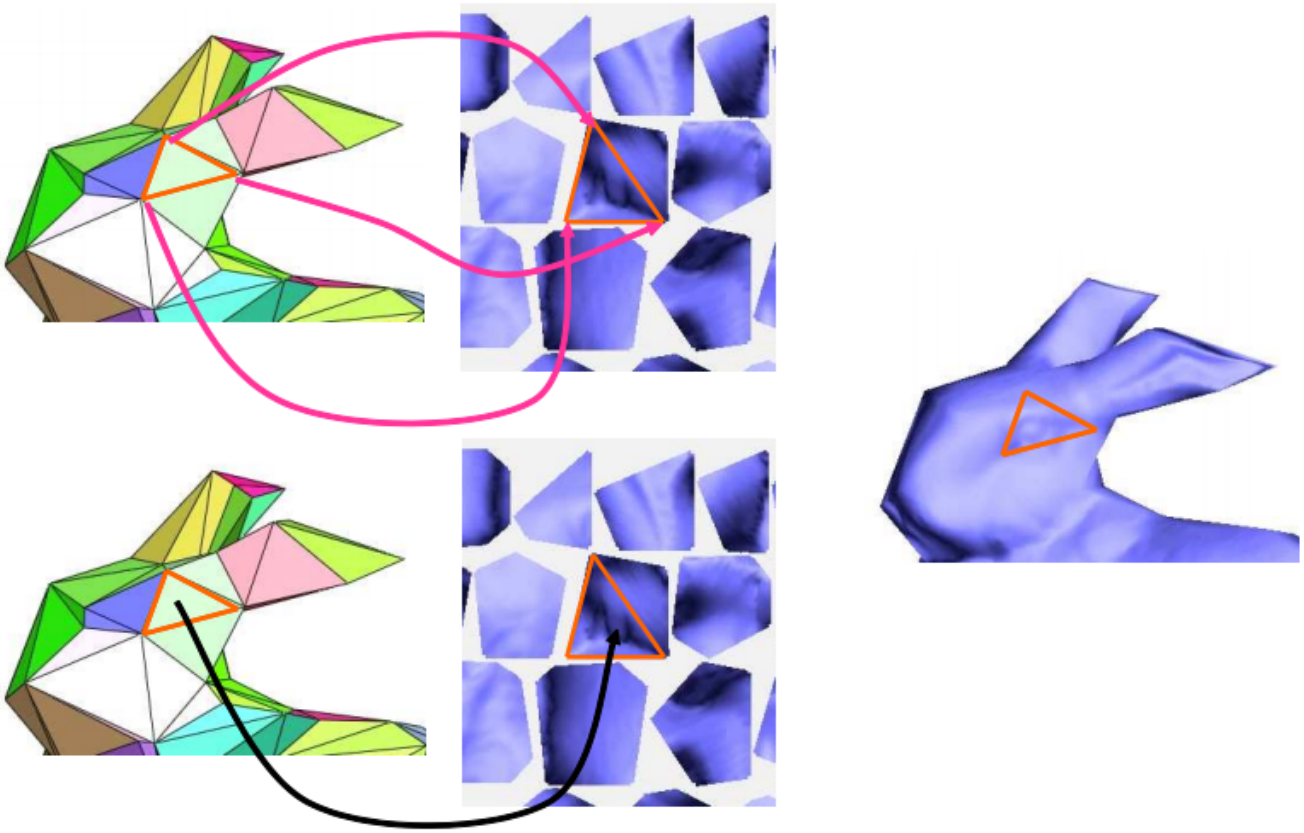
`normals[i]` and `texCoords[i]` correspond to `position[i]`

We can draw the mesh using `gl.drawElements(...)`

Texture Mapping

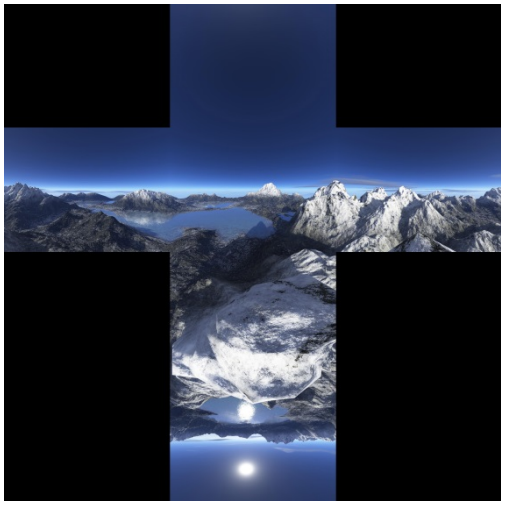
Idea: Glue an image into a surface.

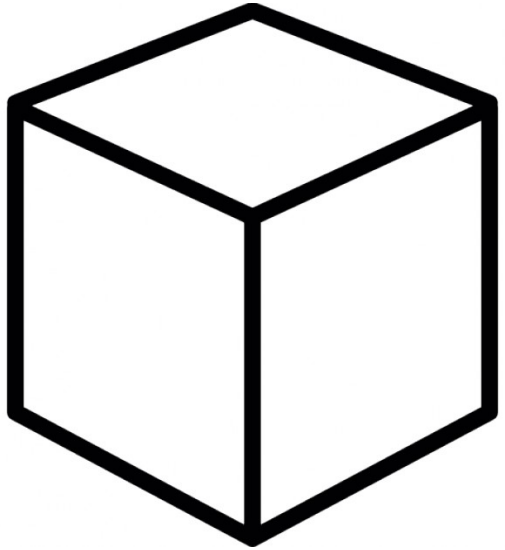
For each triangle in the mesh, we specify textute coordinates at the three vertices.



The texture coordinates of any point in the interior of a face is found by interpolation.

Cube Map





What are the texture coordinates of the corners?

How would you describe the object in Wavefront Obj format?

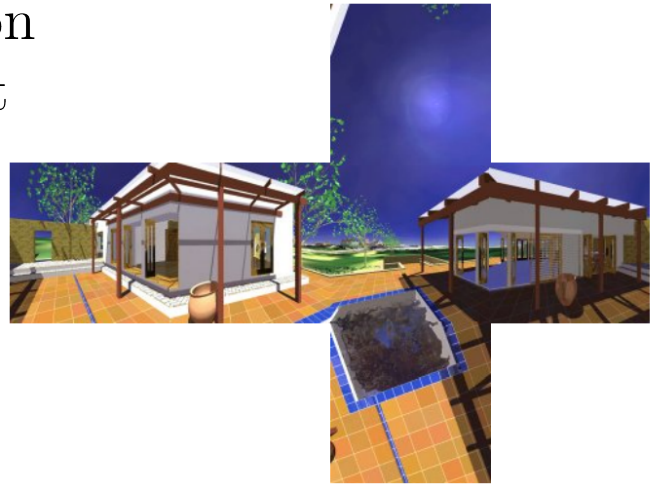
Environment / Reflection Mapping

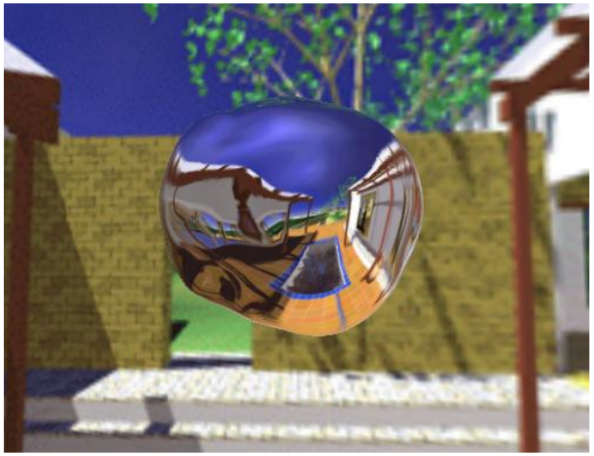
First we set up a cube map for the environment.

For each point on the surface, we know the normal and the direction of the ray towards the eye/camera.

From this we can figure out the direction of the incoming ray.

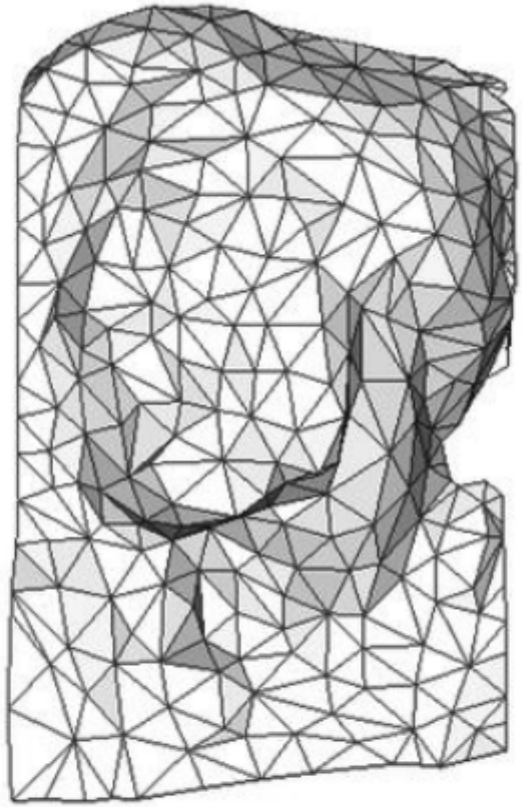
We query the cube map using this direction to get the color at the point.



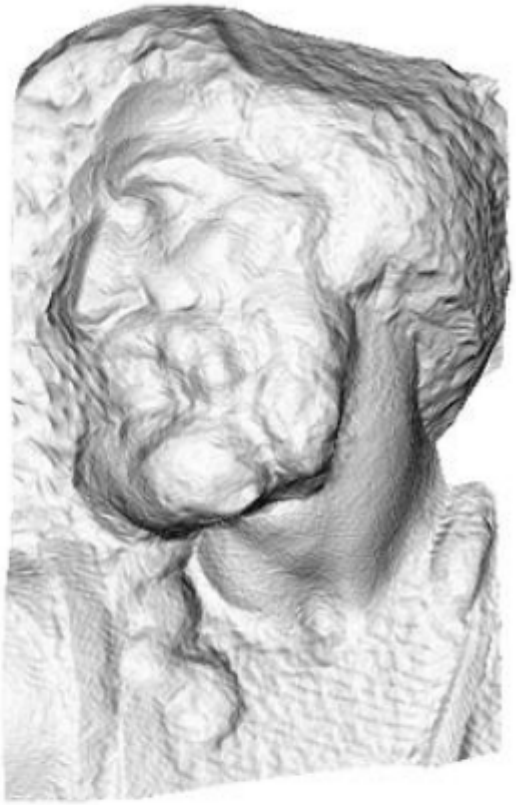


Normal Mapping

original mesh
4M triangles



simplified mesh
500 triangles



simplified mesh
and normal mapping
500 triangles

Bump Mapping vs Normal Mapping

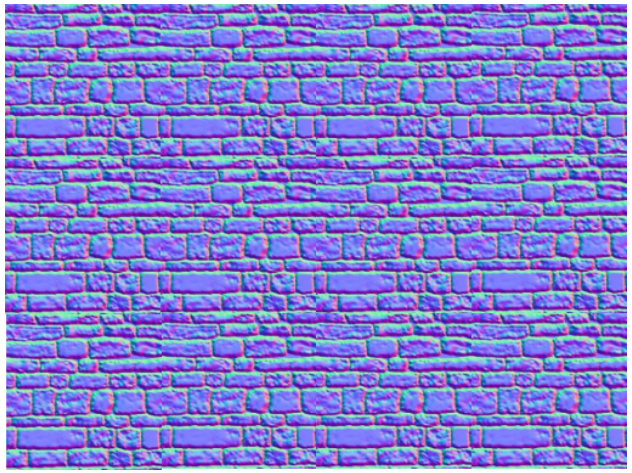
Normal Mapping: The color in the texture encodes a normal vector.

(also known as Dot3 bump mapping)

Bump Mapping: Each texture element stores an intensity encoding the height at that point.

Bump Map

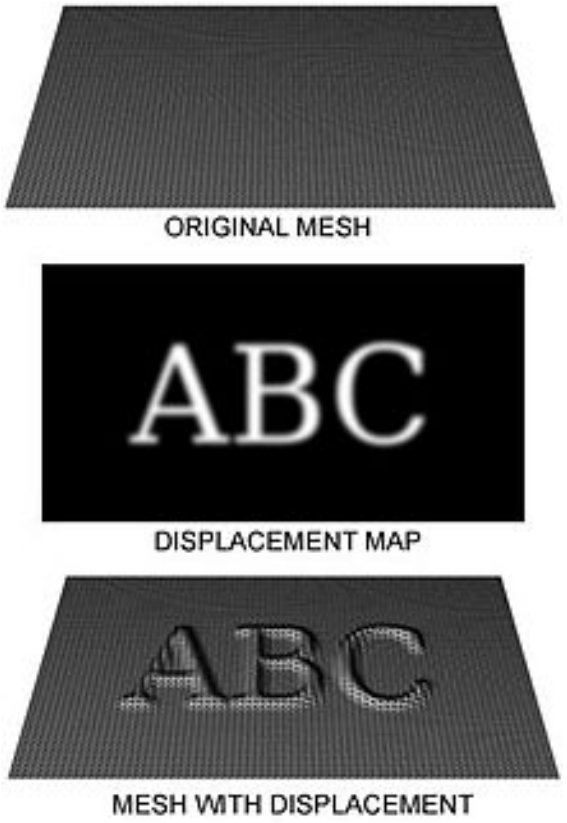
(also called a height field)



Normal Map

Displacement Mapping

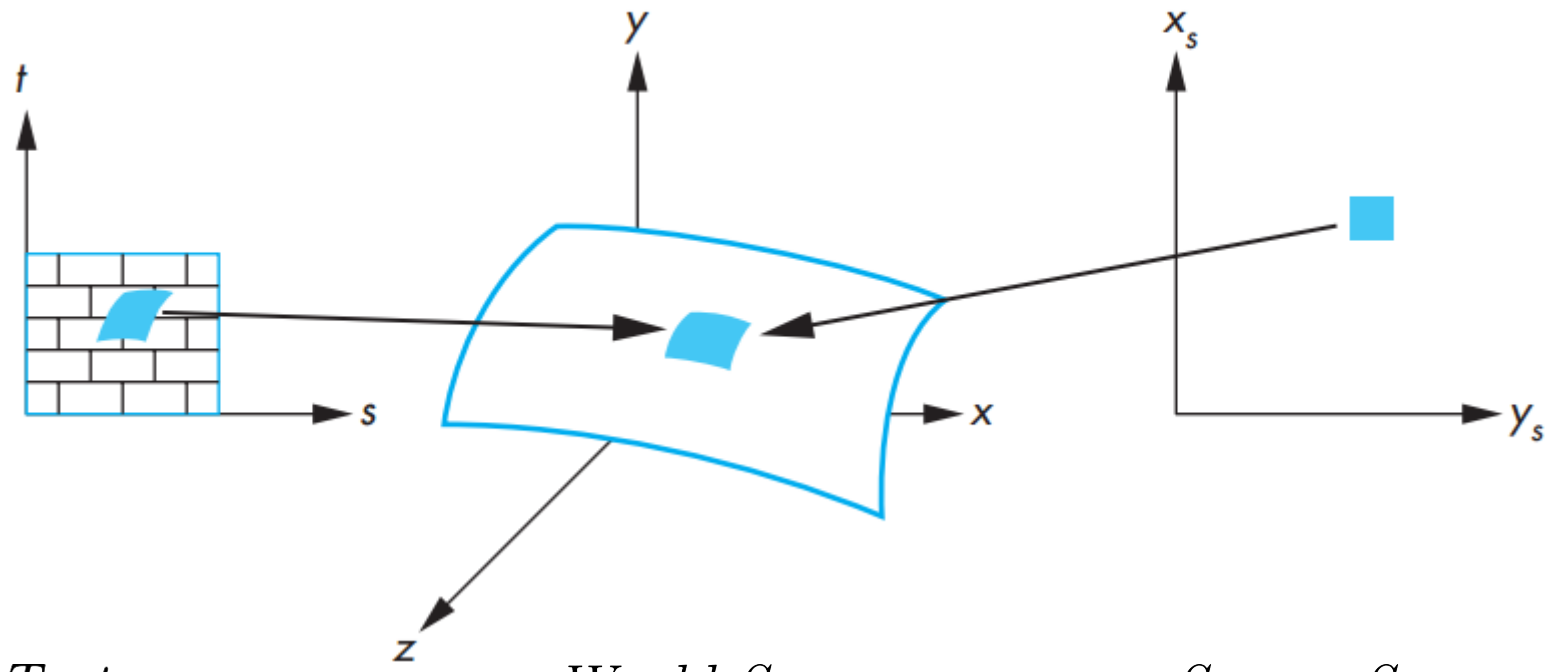
Actual geometric position of the points in the mesh is altered according to the texture.



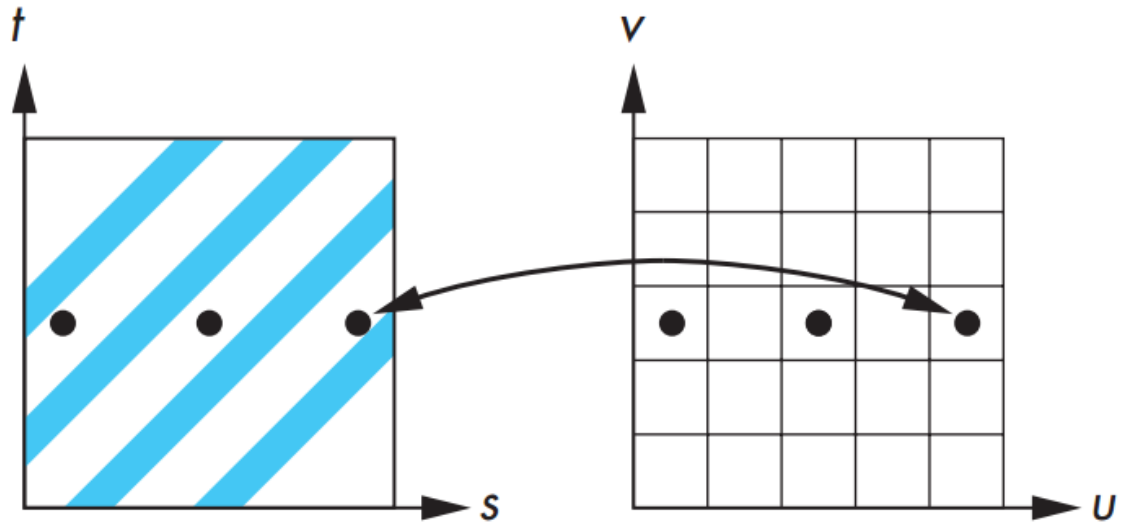
Texture Mapping

The idea of texture mapping is really simple. But there are several technical difficulties:

A pixel is not a point, its a rectangle, and it can have a **curved preimage** in the texture.



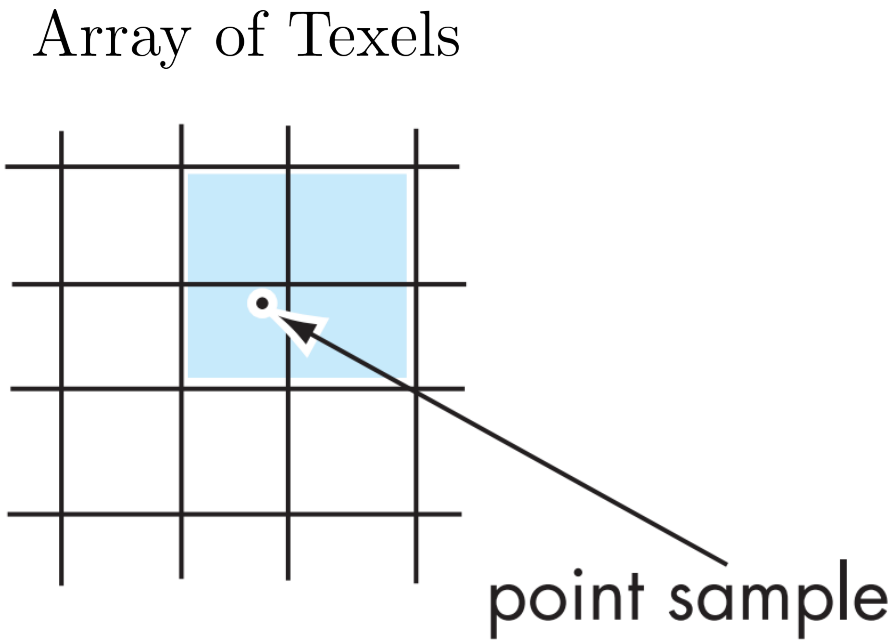
Texture Mapping



If we just look at the preimage of the pixel center, then we have aliasing errors.

A more difficult option: average over the preimage of the pixel on the texture.

Still not accurate: you get an average shade instead of the striped pattern.



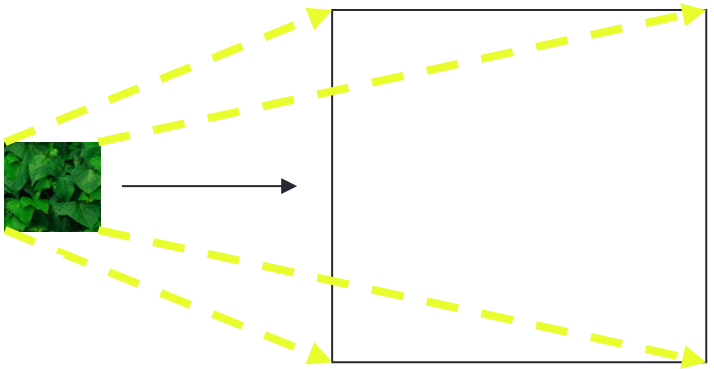
point sampling: Choose the texel whose center is closest to the texture coordinates.

linear filtering: Use a weighted average of the neighboring texels.

Texture Mapping

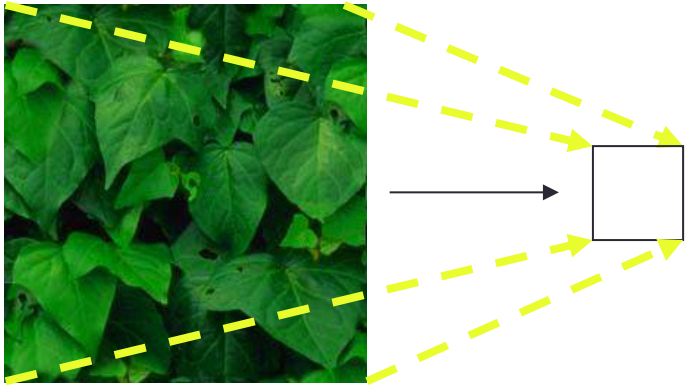
Magnification:

a few texels map to many pixels



Minification:

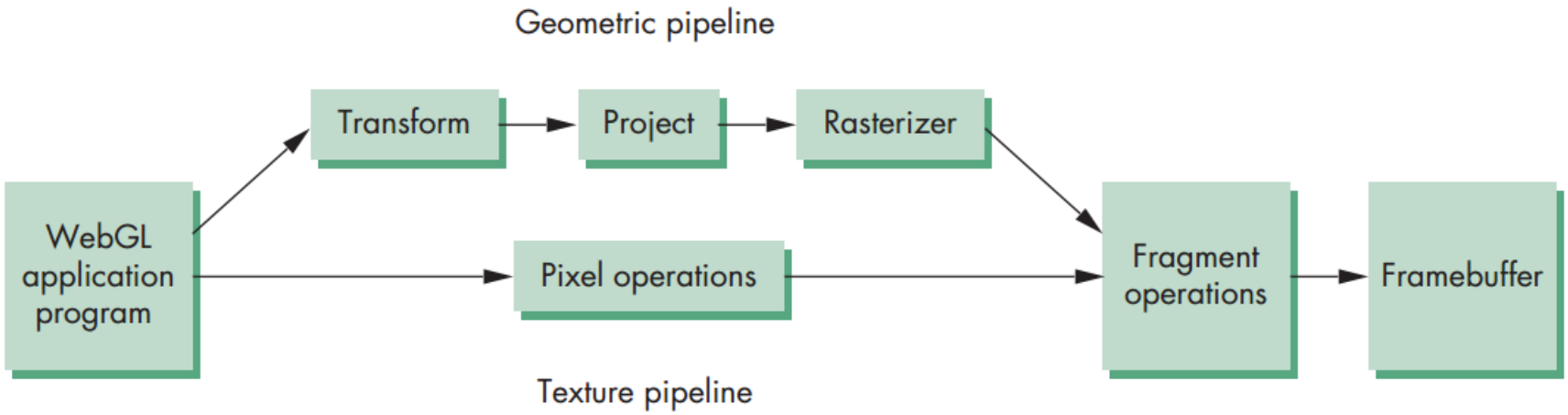
many texels map to a few pixels



In this case, we can keep images at lower resolutions and use the appropriate resolution for mapping.

This is known as **mipmapping**.

Texture Mapping in WebGL



There are actually two pipelines that meet at the Fragment Shader.

Texture Mapping in WebGL

Multiple texture objects can be created and stored in the GPU memory.

We create a texture object using: `var texture = gl.createTexture();`

And we bind it using: `gl.bindTexture(gl.TEXTURE_2D, texture);`

We then need an array of texels.

We can either create it ourselves or load it from an image.

Suppose that we create the following array and fill it in with data:

```
var texSize = 64;
var numRows = 8;
var numCols = 8;

var myTexels = new Uint8Array(4*texSize*texSize);
```

Texture Mapping in WebGL

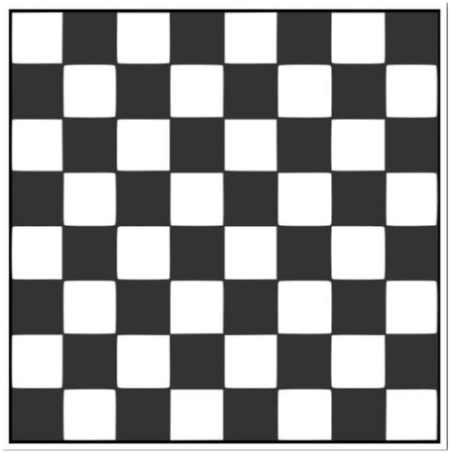
For instance we can fill it up as follows:

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

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

    myTexels[4*i*texSize+4*j]   = c;
    myTexels[4*i*texSize+4*j+1] = c;
    myTexels[4*i*texSize+4*j+2] = c;
    myTexels[4*i*texSize+4*j+3] = 255;
  }
}
```

This creates a black and white checkerboard image.



Texture Mapping in WebGL

Specify that we need to use that image as the texture:

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

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

`target`: specifies `gl.TEXTURE_2D` or `gl.TEXTURE_CUBE_MAP`

`level`: used for mipmapping, level 0 is highest resolution

`iformat`: internal format to be used in texture memory

`width, height`: width and height of the image

`border`: no longer used, should be set to 0

`format`: format used in image

`type`: the type of texture data

`texelArray`: array in which the image is stored

Texture Mapping in WebGL

We can also load the data from an image:

```
var myTexels = new Image();              create a new Javascript Image object
myTexels.src = "my_image.gif";           specify the source
gl.pixelStorei(gl.UNPACK_FLIP_Y_WEBGL, true);      invert vertically
gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGB, gl.RGB,
              gl.UNSIGNED_BYTE, myTexels);
```

We need to invert vertically because images use a coordinate system where the top left corner is the origin and y-axis runs downwards.

Texture Mapping in WebGL

We also need to set up texture coordinates.

We associate with each vertex a 2D texture coordinate.

These can be passed to the vertex shader as attributes and can be interpolated in the fragment shader via varying variables.

In the fragment shader, we need a new type of variable called **sampler**.

```
uniform sampler2D texMap;
```

We connect this object to the texture object we created:

```
gl.activeTexture(gl.TEXTURE0);
gl.bindTexture(gl.TEXTURE_2D, texture);
gl.uniform1i(gl.getUniformLocation(program, "Tex0"), 0);
```

associate sampler `Tex0` *to texture unit* 0

Texture Mapping in WebGL

The fragment shader looks something like this:

```
varying vec2 fTexCoord;          interpolated texture coordinate
                                 of the fragment
uniform sampler2D texMap;

void main(){
      gl_FragColor = texture2D(texMap,fTexCoord);
}
```

Some more parameters to be set:

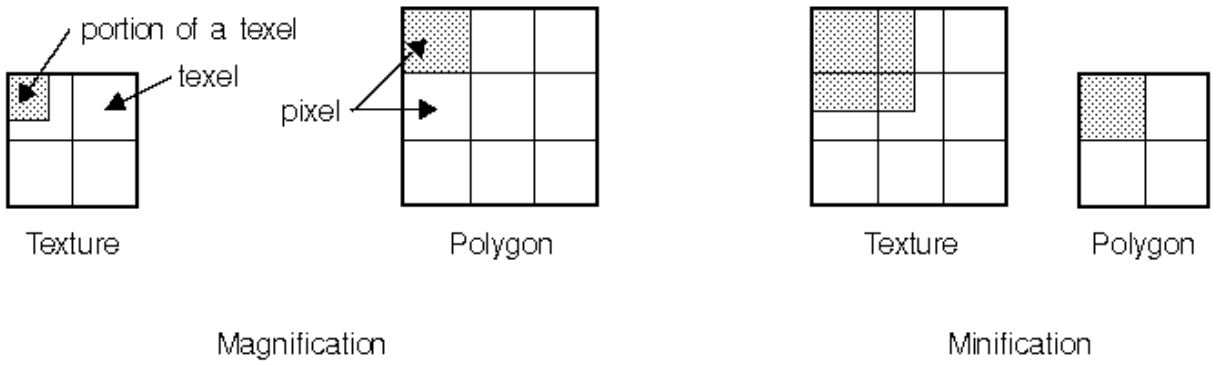
minification and magnification filters

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

Alternative: use `gl.LINEAR` - more accurate but slower.

Mipmapping: solution to the minification problem

many texels map to a few pixels



Idea: Use series of texture arrays with decreasing sizes.

For a 64×64 original array, we can set up arrays of sizes 32×32, 16×16, 8×8, 4×4, 2×2 and 1×1 as follows:

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

Note that original dimensions have to be the same and a power of 2.

Texture Mapping in WebGL

We could also set up the maps using the `level` parameter in `gl.TexImage2D`.

That allows us to specify different images for different levels.

We then specify the minification filter:

```
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER,
                                        gl.NEAREST_MIPMAP_NEAREST);
```

Options:

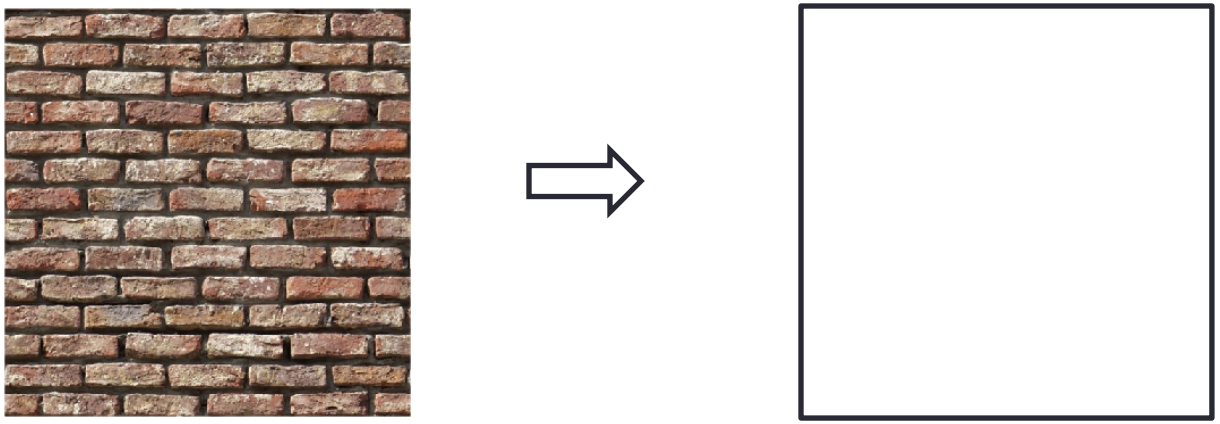
`gl.NEAREST_MIPMAP_NEAREST`: point sampling with best mipmap

`gl.NEAREST_MIPMAP_LINEAR`: linear filtering with best mipmap

`gl.LINEAR_MIPMAP_LINEAR`: linear filtering with linear filtering between mipmaps

Texture Mapping

Minimal example: pasting a picture on a square



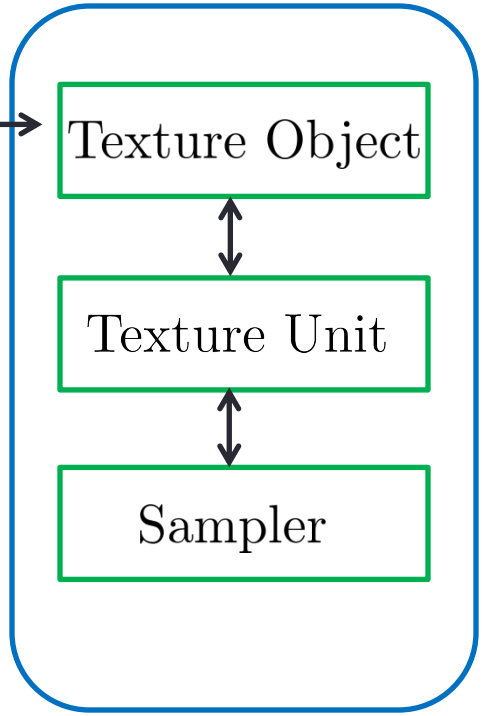
We need:

- positions of the vertices
- texture coordinates of the vertices

Texture Mapping

Image object

create an image object in Javascript.



create a WebGL texture object

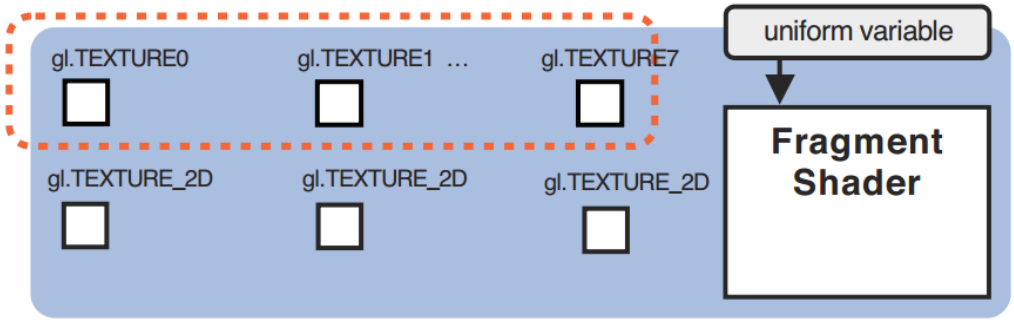
associate the image with it

associate the texture object with a texture unit

create a sampler in the shader and associate it with the texture unit

There are multiple texture units, usually at least 8.

We can create multiple texture objects and assign them to different texture units.



Texture Mapping

```
"use strict";
var gl; // global variable
var image;

window.onload = function init(){
    //Set  up WebGL
    var canvas = document.getElementById( "gl-canvas" );
    gl = WebGLUtils.setupWebGL( canvas );
    if ( !gl ) {alert( "WebGL isn't available" );}

    // Set viewport and clear canvas
    gl.viewport( 0, 0, canvas.width, canvas.height );
    gl.clearColor( 0.0, 0.0, 0.0, 1.0 );
    gl.clear(gl.COLOR_BUFFER_BIT);

    // Load shaders and initialize attribute buffers
    var program = initShaders( gl, "vertex-shader", "fragment-shader" );
    gl.useProgram( program );
```

Texture Mapping

```
// Set up buffers and attributes
var s = 0.7;
var vertices =  [-s,-s,  s,-s,  s,s,  -s,s];
var texCoords = [ 0,0,   1,0,  1,1,    0,1];
var indices =   [0,1,2,  0,2,3];

var vbuffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, vbuffer);
gl.bufferData(gl.ARRAY_BUFFER, flatten(vertices), gl.STATIC_DRAW);
var vPosition = gl.getAttribLocation(program, "vPosition");
gl.vertexAttribPointer(vPosition, 2, gl.FLOAT, false, 0, 0);
gl.enableVertexAttribArray(vPosition);
```

Associating texture coordinates with vertices.


```
var tbuffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, tbuffer);
gl.bufferData(gl.ARRAY_BUFFER, flatten(texCoords), gl.STATIC_DRAW);
var vTexCoord = gl.getAttribLocation(program, "vTexCoord");
gl.vertexAttribPointer(vTexCoord, 2, gl.FLOAT, false, 0, 0);
gl.enableVertexAttribArray(vTexCoord);
```

```
var ibuffer = gl.createBuffer();
gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, ibuffer);
gl.bufferData(gl.ELEMENT_ARRAY_BUFFER, new Uint8Array(flatten(indices)), gl.STATIC_DRAW);
```

Texture Mapping

```
    var texture = gl.createTexture();
    var mySampler = gl.getUniformLocation(program, "mySampler");

    image = new Image();
    image.onload = function(){handler(texture);};
    image.src = "brick.gif";

    function handler(texture){
        gl.activeTexture(gl.TEXTURE0);            // enable texture unit 0
        gl.bindTexture(gl.TEXTURE_2D, texture);  // bind texture object to target
        gl.uniform1i(mySampler, 0);              // connect sampler to texture unit 0

        gl.pixelStorei(gl.UNPACK_FLIP_Y_WEBGL, true); // flip image's y axis
        gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGBA, gl.RGBA, gl.UNSIGNED_BYTE, image);
        gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER, gl.NEAREST);
        gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.NEAREST);

        gl.drawElements(gl.TRIANGLES,6,gl.UNSIGNED_BYTE,0);

    }
};
```

We do three things:

1. Create a texture object and associate it with an image.
2. Associate a sampler with the texture object.
3. Associate the sampler with a texture unit.

Texture Mapping

```
<script id="vertex-shader" type="x-shader/x-vertex">
precision mediump float;

attribute vec4 vPosition;
attribute vec2 vTexCoord;

varying vec2 fTexCoord;

void main(){
    gl_Position = vPosition;
    fTexCoord = vTexCoord;
}
</script>
```

Using a varying variable to get texture coordinates at each fragment.

```
<script id="fragment-shader" type="x-shader/x-fragment">
precision mediump float;

varying vec2 fTexCoord;

uniform sampler2D mySampler;

void main(){
    gl_FragColor = texture2D(mySampler,fTexCoord);
}
</script>
```

Using the sampler.

Multiple Textures

Example:

create two texture objects

```
texture1 = gl.createTexture();
gl.bindTexture(gl.TEXTURE_2D, texture1);
gl.pixelStorei(gl.UNPACK_FLIP_Y_WEBGL, true);
gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGBA, texSize, texSize, 0,
              gl.RGBA, gl.UNSIGNED_BYTE, image1);
gl.generateMipmap(gl.TEXTURE_2D);
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER,
gl.NEAREST_MIPMAP_LINEAR);
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER, gl.NEAREST);

texture2 = gl.createTexture();
gl.bindTexture(gl.TEXTURE_2D, texture2);
gl.pixelStorei(gl.UNPACK_FLIP_Y_WEBGL, true);
gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGBA, texSize, texSize, 0,
              gl.RGBA, gl.UNSIGNED_BYTE, image2);
gl.generateMipmap(gl.TEXTURE_2D);
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER,
gl.NEAREST_MIPMAP_LINEAR);
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER, gl.NEAREST);
```

Multiple Textures

In the fragment shader, we need two samplers to access them.

```
uniform sampler2D Tex0;
uniform sampler2D Tex1;
```

create two samplers

Connect the texture objects with samplers.

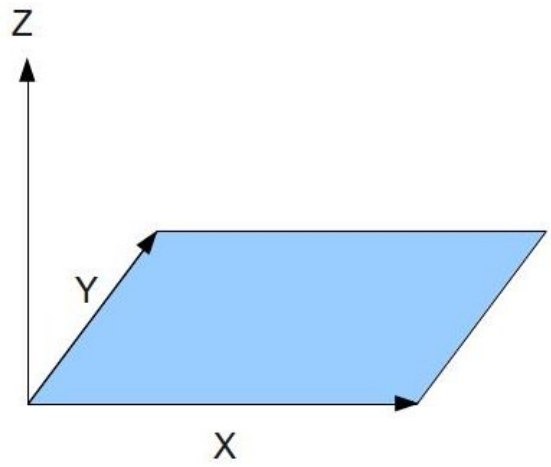
```
gl.activeTexture(gl.TEXTURE0);      subsequent calls affect texture unit 0
gl.bindTexture(gl.TEXTURE_2D, texture1);
gl.uniform1i(gl.getUniformLocation(program, "Tex0"), 0);

gl.activeTexture(gl.TEXTURE1);      subsequent calls affect texture unit 1
gl.bindTexture(gl.TEXTURE_2D, texture2);
gl.uniform1i(gl.getUniformLocation(program, "Tex1"), 1);
```

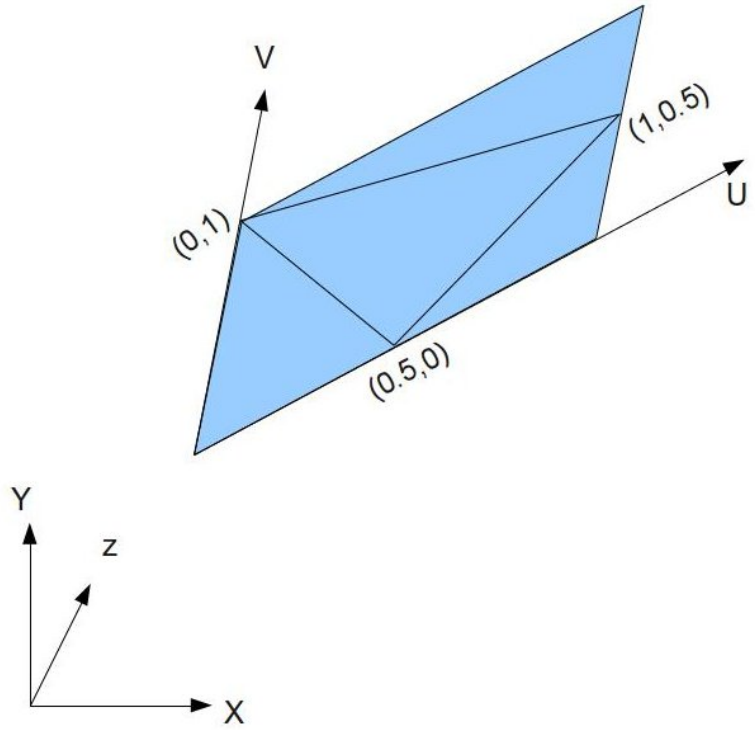
Now the fragment shader can do something like this:

```
gl_FragColor = color * texture2D(Tex0, texCoord)
                     * texture2D(Tex1, texCoord);
```

Normal Mapping



Coordinate system in which normals are defined.

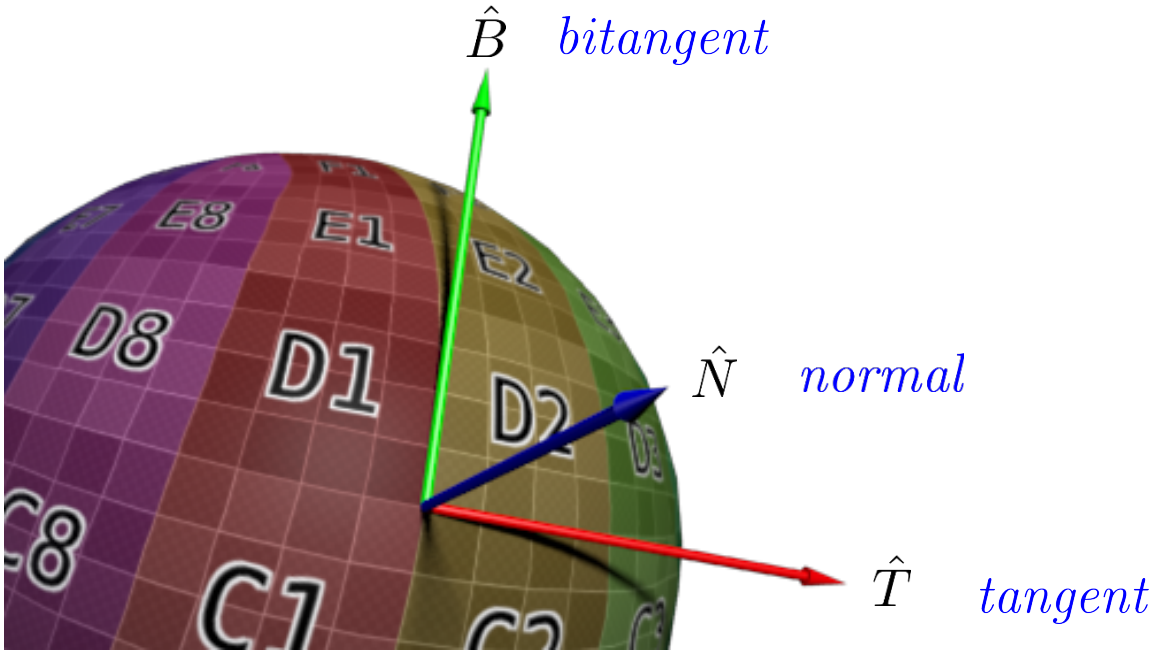


Placing the normal map on a triangle whose texture coordinates are $(0.5, 0)$, $(1, 0.5)$ and $(0, 1)$.

The normal we get from the texture needs to be modified according to the orientation of surface.

Normal Mapping

Set up a local coordinate frame:



The normal we get from the texture is described in this frame.

We convert it to the coordinate frame where we do lighting.

Normal Mapping

The tangent frame can be easily computed for surfaces described by an equation using partial derivatives.

How do we compute the tangent frame for triangular meshes?

- For each triangle, compute the tangent, bitangent and normal

- For each vertex, average the values at adjacent triangles

- For fragments, we get the vectors by linear interpolation

Note: This leads to slight inaccuracy since after the averaging and interpolation, the three vectors are not perpendicular to each other.

We ignore this since if the triangles are small, the error is small.

Normal Mapping

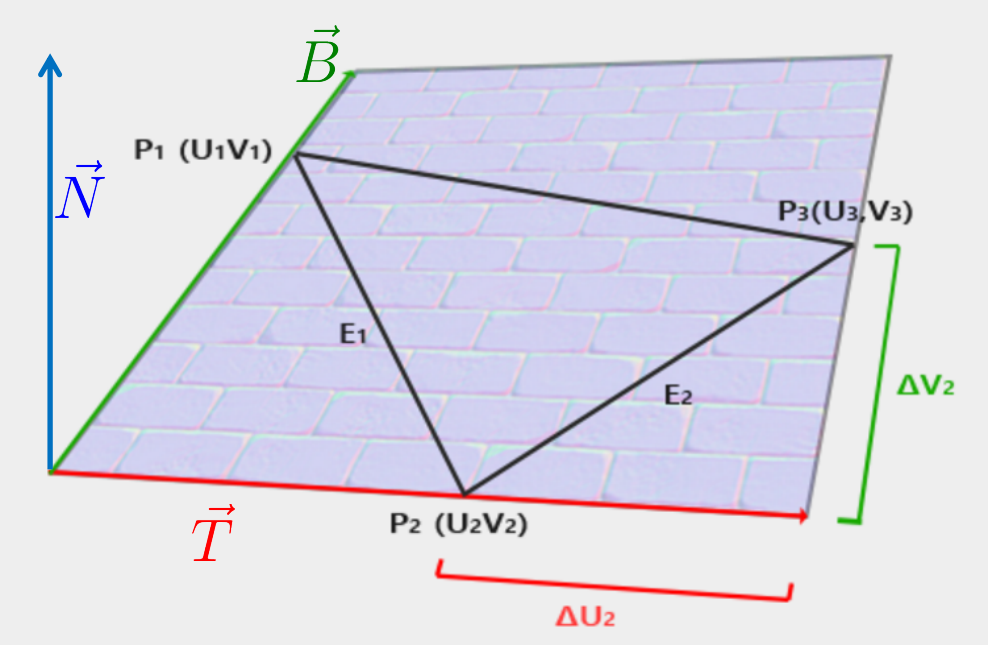
How do we compute tangents and bitangent on a triangle?

We have texture coordinates at each vertex.

These are along $\vec{T}$ and $\vec{B}$.

We also know the positions of the vertices.

From this we can figure out $\vec{T}$ and $\vec{B}$.



This is how it would look if we put the entire texture in the plane of the triangle so that the texture coordinates at the vertices match.

Note that $\vec{T}$ and $\vec{B}$ need not be of unit length.

They are as long as the length and height of the texture respectively.

Normal Mapping

Let $\vec{E}_1 = P_2 - P_1$

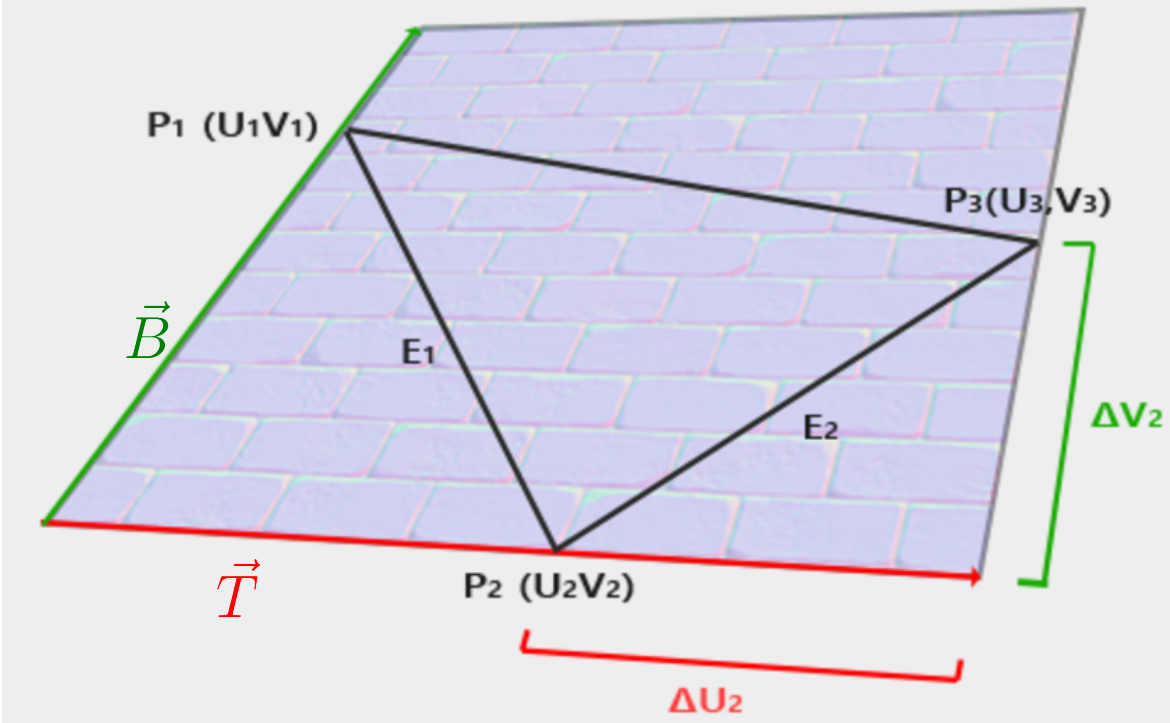
$\vec{E}_2 = P_3 - P_2$

$\Delta U_1 = U_2 - U_1$

$\Delta V_1 = V_2 - V_1$

$\Delta U_2 = U_3 - U_2$

$\Delta V_2 = V_3 - V_2$



$\vec{E}_1 = \Delta U_1 \vec{T} + \Delta V_1 \vec{B}$

$\vec{E}_2 = \Delta U_2 \vec{T} + \Delta V_2 \vec{B}$

two equations in two variables

Normal Mapping

$$\vec{E}_1 = \Delta U_1 \vec{T} + \Delta V_1 \vec{B}$$

$$\vec{E}_2 = \Delta U_2 \vec{T} + \Delta V_2 \vec{B}$$

Expanding to three coordinates:

$$(E_{1x}, E_{1y}, E_{1z}) = \Delta U_1 (T_x, T_y, T_z) + \Delta V_1 (B_x, B_y, B_z)$$

$$(E_{2x}, E_{2y}, E_{2z}) = \Delta U_2 (T_x, T_y, T_z) + \Delta V_2 (B_x, B_y, B_z)$$

In matrix notation:

$$\begin{bmatrix} E_{1x} & E_{1y} & E_{1z} \\ E_{2x} & E_{2y} & E_{2z} \end{bmatrix} = \begin{bmatrix} \Delta U_1 & \Delta V_1 \\ \Delta U_2 & \Delta V_2 \end{bmatrix} \begin{bmatrix} T_x & T_y & T_z \\ B_x & B_y & B_z \end{bmatrix}$$

Normal Mapping

$$\begin{bmatrix} E_{1x} & E_{1y} & E_{1z} \\ E_{2x} & E_{2y} & E_{2z} \end{bmatrix} = \begin{bmatrix} \Delta U_1 & \Delta V_1 \\ \Delta U_2 & \Delta V_2 \end{bmatrix} \begin{bmatrix} T_x & T_y & T_z \\ B_x & B_y & B_z \end{bmatrix}$$

$$\Downarrow$$

$$\begin{bmatrix} \Delta U_1 & \Delta V_1 \\ \Delta U_2 & \Delta V_2 \end{bmatrix}^{-1} \begin{bmatrix} E_{1x} & E_{1y} & E_{1z} \\ E_{2x} & E_{2y} & E_{2z} \end{bmatrix} = \begin{bmatrix} T_x & T_y & T_z \\ B_x & B_y & B_z \end{bmatrix}$$

$$\Downarrow$$

$$\begin{bmatrix} T_x & T_y & T_z \\ B_x & B_y & B_z \end{bmatrix} = \frac{1}{\Delta U_1 \Delta V_2 - \Delta U_2 \Delta V_1} \begin{bmatrix} \Delta V_2 & -\Delta V_1 \\ -\Delta U_2 & \Delta U_1 \end{bmatrix} \begin{bmatrix} E_{1x} & E_{1y} & E_{1z} \\ E_{2x} & E_{2y} & E_{2z} \end{bmatrix}$$

Normal Mapping

We now need two more vertex attributes: **tangent** and **bitangent**.

Just like normals, these are interpolated for each fragment so that we get the normal, tangent and bitangent at each fragment.

Now, we use the texture coordinates to read the color (r, g, b) from the normal map texture.

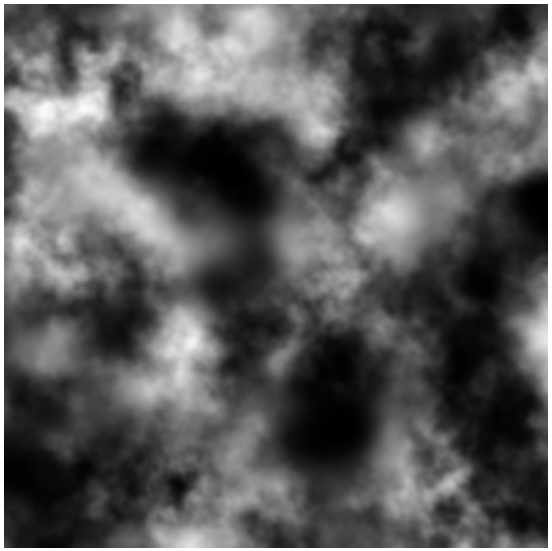
Convert color to normal in the tangent frame: $(2r-1, 2g-1, 2b-1)$

This is converted to world frame for lighting:

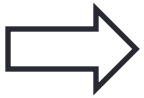
$$\vec{N}' = (2r-1)\hat{T} + (2g-1)\hat{B} + (2b-1)\hat{N}$$

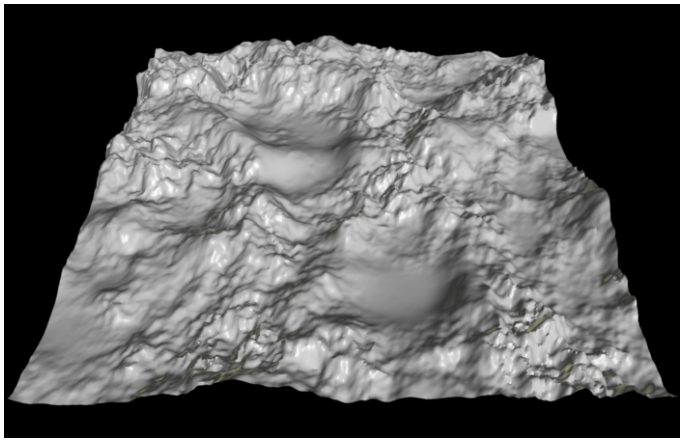
where $\hat{T} = \frac{\vec{T}}{\|\vec{T}\|}$, $\hat{B} = \frac{\vec{B}}{\|\vec{B}\|}$ and $\hat{N} = \frac{\vec{N}}{\|\vec{N}\|}$.

Terrains using Height Maps



Height Map





Generated Terrain

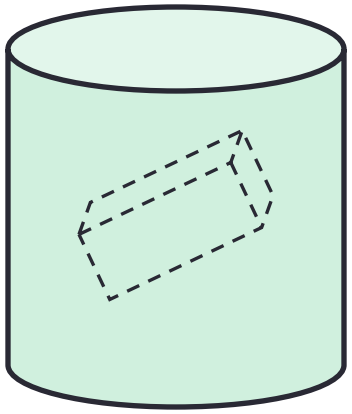
We can use a terrain texture like this:

Solid (3d) Textures

Looks better than a 2d texture pasted on the surface of the model.

Storing the texture in a file limits resolution and requires more space, but is faster.

Procedural generation is slower but requires less storage space and has potentially unlimited resolution.

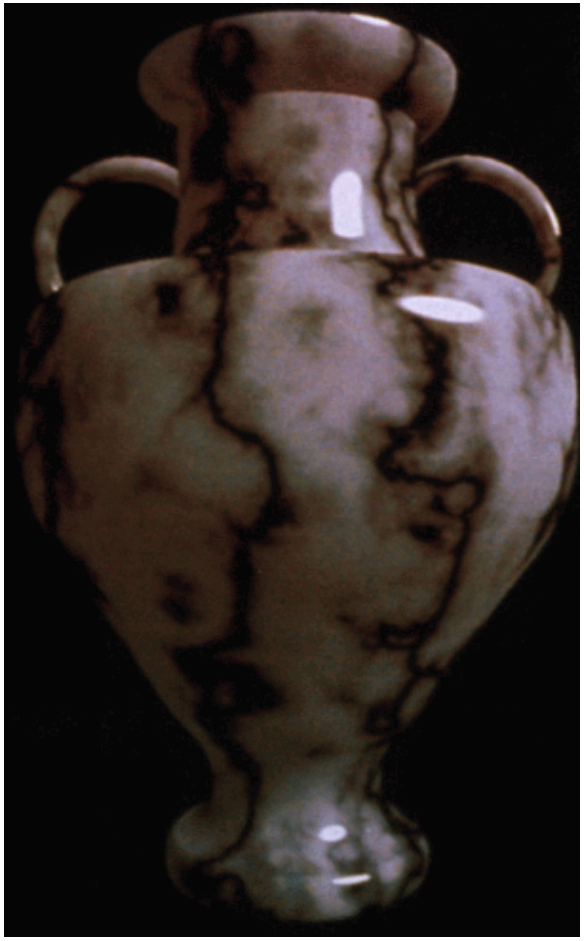


We want a function that given any point in the cylinder outputs the wood color at that point.

Creating a function so that each horizontal cross section has perfectly circular rings with two colors is easy.

Can be made more realistic using perturbations.

Procedural Generation



Many natural phenomena can be simulated procedurally.

Key ingredient: Randomness.

Procedural Noise

Natural phenomena contain a mixture of structure and randomness.

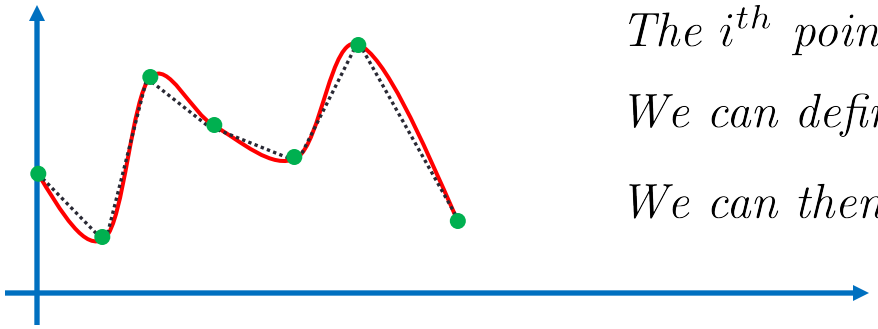
For example, altitude in a terrain isn't totally random.

There is random but gradual change as we move around in the terrain.

And this happens at all scales.

How do we obtain a smooth random function using a pseudo random number generator (PRNG)?

Idea. Interpolation of randomly generated points.



The i^{th} point has coordinates $(i, rand(i))$.

We can define the function on $[0, n]$ s.t. $f(0) = f(n)$.

We can then make it periodic: $g(t) = f(t \bmod n)$.

Procedural Noise

How do we achieve randomness at every scale?

$$N(t) = g(t) + \frac{1}{2}\, g(2t) + \frac{1}{4}\, g(4t) + \frac{1}{8}\, g(8t) + \cdots$$

We add terms with exponetially increasing frequency and exponentially decreasing amplitude.

This also gives us *self similarity*.

We could also add an *offset* to each scaled copy.