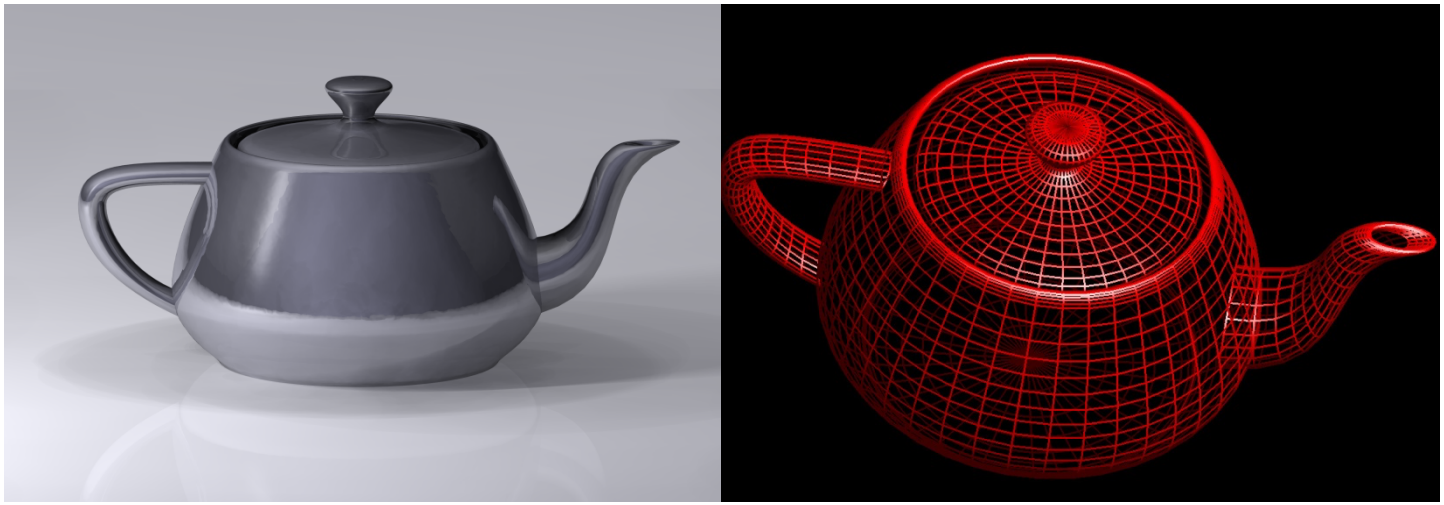


Foundations of Computer Graphics

Saurabh Ray

Reading

Reading for lectures 11, 12: Sections 7.1 - 7.9

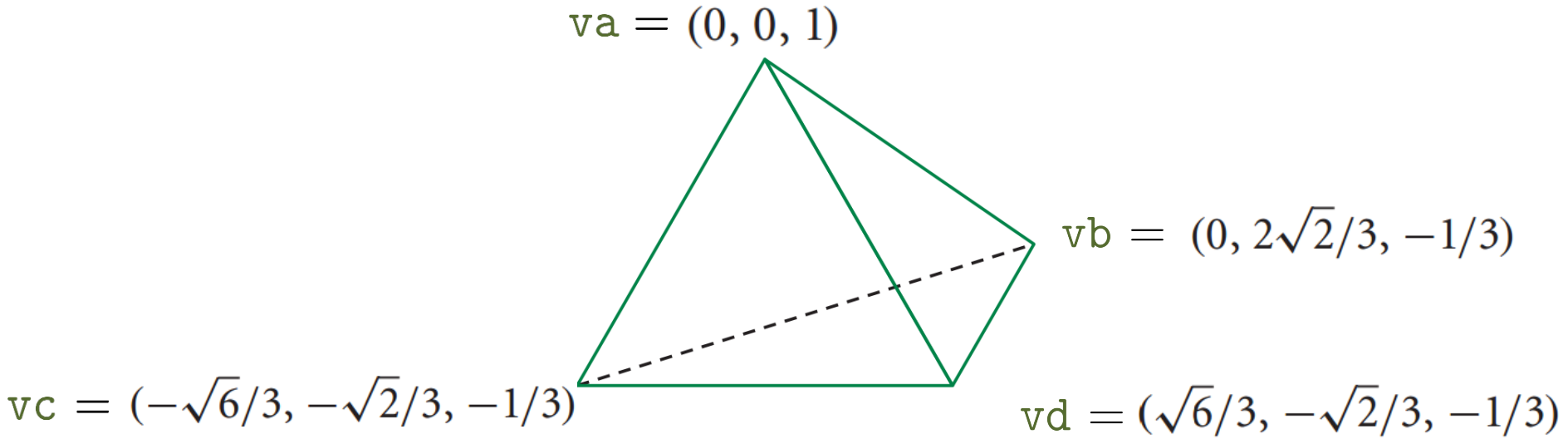
Please also read the code that we post on Github.
(`https://github.abudhabi.nyu.edu/sr194/CG-2020`)

Coding

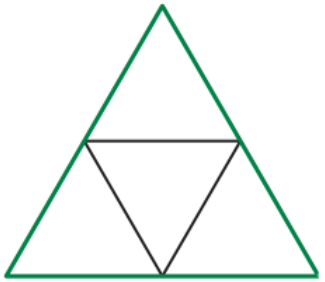
Want: Display a sphere with lighting.

How do we model a sphere?

Idea: Start with a tetrahederon and refine it.



Subdivide each face into four triangles:



Coding

```
function Sphere(n){
    // n is the number of times to
    // subdivide the faces recursively.

    var S = {   positions: [],
                normals: [],
            };



    var s2 = Math.sqrt(2);
    var s6 = Math.sqrt(6);

    var va = vec3(0,0,1);
    var vb = vec3(0, 2*s2/3, -1/3);
    var vc = vec3(-s6/3, -s2/3, -1/3);
    var vd = vec3(s6/3, -s2/3, -1/3);

    tetrahedron(va, vb, vc, vd, n);



    function tetrahedron(a,b,c,d,n){
        divideTriangle(d,c,b,n);
        divideTriangle(a,b,c,n);
        divideTriangle(a,d,b,n);
        divideTriangle(a,c,d,n);
    }
```

```
    function divideTriangle(a,b,c,n){
        if(n>0){
            var ab = normalize(mix(a,b,0.5));
            var ac = normalize(mix(a,c,0.5));
            var bc = normalize(mix(b,c,0.5));

            n--;

            divideTriangle(a,ab,ac,n);
            divideTriangle(ab,b,bc,n);
            divideTriangle(bc,c,ac,n);
            divideTriangle(ab,bc,ac,n);
        }
        else{
            triangle(a,b,c);
        }
    }


    function triangle(a,b,c){
        var norm = normalize(cross(subtract(b,a),
                                   subtract(c,a)));
        S.positions.push(a,b,c);
        S.normals.push(norm, norm, norm);
        //S.normals.push(a,b,c);
    }

    return S;
}
```

Coding

It will be useful to write a function `objInit` which attaches several functions to the sphere object we created.

```
function objInit(Obj){

    /* Initialization Code.
    (declare and initialize variables,
    create buffers and transfer data to buffers,
    get locations of attributes and uniforms, etc) */
    // ...

    /*----- Helper functions -----*/
    // Define any helper functions you need.
    // ...

    /*----- Attach functions to Obj -----*/

    Obj.setModelMatrix = function(m){
        // use m as the modeling matrix
        // ...
    }

    Obj.draw = function(){
        // draw the object
        // ...
    }
}
```

We will create a file called `Object.js` containing this function.

Coding

To make it easy to use the virtual trackball in future projects we create a file `Trackball.js` containing a function `Trackball` that returns a trackball object.

```
function Trackball(canvas){

    var tbMatrix = mat4(); /* initialize virtual trackball matrix */

    var trackball = { /* object to be returned */
        getMatrix: function() { return tbMatrix; }
    };

    /* other variables and helper functions */

    // ...

    function mousedown(event){ /* ... */ }
    function mouseup(event)  { /* ... */ }
    function mousemove(event){ /* ... */ }
    function wheel(event)    { /* ... */ }

    //set event handlers
    canvas.addEventListener("mousedown", mousedown);
    canvas.addEventListener("mouseup", mouseup);
    canvas.addEventListener("mousemove", mousemove);
    canvas.addEventListener("wheel", wheel, {passive: true});

    return trackball;
}
```

Coding

To make it easy to set up the camera, we create a file `Camera.js` containing the function `Camera()` which returns a camera object.

```
function Camera(){

    var Mcam = mat4(); // camera matrix
    var P = mat4();    // projection matrix

    var camFrame = { // camera frame
            e: vec3(0,0,0), // camera location
            u: vec3(1,0,0), // unit vector to "right"
            v: vec3(0,1,0), // unit vector in "up" direction
            w: vec3(0,0,1)  // unit vector opposite "gaze" direction
    };

    addEventHandlers(); // add event handlers for moving the camera


    var Cam = { /* object to be returned */

            lookAt: function (eye, at, up) {
                // set camFrame and Mcam
                // ...
            },

            setPerspective: function(fovy, aspect, near, far){
                // set the projection matrix P
                // ...
            },
```

Coding

```
        setOrthographic: function (r,l,t,b,n,f){
            // set the projection matrix P
            // ...
        },

        getCameraTransformationMatrix: function(){
            return Mcam;
        },

        getProjectionMatrix: function(){
            return P;
        },

        getMatrix: function(){
            // combines camera transformation and projection
            return mult(P,Mcam);
        },

        getFrame: function (){
            // returns camera frame (e, u, v, w)
            return camFrame;
        }
    };


    /* Helper functions */

    function addEventHandlers(){
        // ...
    }

    // ...

    return Cam;
}
```

Coding

The Javascript code for displaying a sphere with lighting looks like this:

```
"use strict";

// global variables
var gl, canvas, program;

var camera;     // camera object
var trackball;  // virtual trackball

var sphere;

var Locations;  // object containing locations of shader variables

window.onload = function init() {
    // Set up WebGL
    canvas = document.getElementById("gl-canvas");
    gl = WebGLUtils.setupWebGL( canvas );
    if(!gl){alert("WebGL setup failed!");}

    // set clear color
    gl.clearColor(0.0, 0.0, 0.0, 1.0);

    //Enable depth test
    gl.enable(gl.DEPTH_TEST);
    gl.depthFunc(gl.LEQUAL);
    gl.clearDepth(1.0);

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

Coding

```
    var Attributes = [];
    var Uniforms = ["VP", "TB", "TBN", "cameraPosition", "Ia", "Id", "Is", "lightPosition"];
    Locations = getLocations(Attributes, Uniforms);

    // set up virtual trackball
    trackball = Trackball(canvas);

    // set up Camera
    camera = Camera();          // defined in Camera.js
    var eye = vec3(0,0, 3);
    var at = vec3(0, 0 ,0);
    var up = vec3(0,1,0);
    camera.lookAt(eye,at,up);
    camera.setPerspective(90,1,0.1,10);

    sphere = Sphere(6);

    objInit(sphere);// defined in Object.js
    sphere.setModelMatrix(scalem(2,1,2));

    // set light source
    var Light = {
        position: vec3(-5,10,20),
        Ia: vec3(0.2, 0.2, 0.2),
        Id: vec3(1,1,1),
        Is: vec3(0.8,0.8,0.8)
    };

    gl.uniform3fv( Locations.lightPosition, flatten(Light.position) );
    gl.uniform3fv( Locations.Ia, flatten(Light.Ia) );
    gl.uniform3fv( Locations.Id, flatten(Light.Id) );
    gl.uniform3fv( Locations.Is, flatten(Light.Is) );

    var shading = 3.0; // flat: 1.0, Gouraud: 2.0, Phong: 3.0
    gl.uniform1f(gl.getUniformLocation(program, "shading"), shading);

    requestAnimationFrame(render);
};
```

Coding

```
function render(now){

    requestAnimationFrame(render);
    gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);

    var TB = trackballWorldMatrix(trackball, camera);
    gl.uniformMatrix4fv(Locations.TB, gl.FALSE, flatten(TB));

    var TBN = normalTransformationMatrix(TB);
    gl.uniformMatrix3fv(Locations.TBN, gl.FALSE, flatten(TBN));

    var VP = camera.getMatrix();
    gl.uniformMatrix4fv(Locations.VP, gl.FALSE, flatten(VP));

    var cameraPosition = camera.getFrame().e;
    gl.uniform3fv(Locations.cameraPosition, flatten(cameraPosition));

    sphere.draw();

}
```

Vertex Shader:

```
precision highp float;

attribute vec4 vPosition;
attribute vec3 vNormal;

uniform mat4 M, TB, VP;
uniform mat3 N, TBN;
uniform vec3 Ka, Kd, Ks, Ia, Id, Is, lightPosition, cameraPosition;
uniform float shininess;
uniform float shading;

varying vec4 fColor;
varying vec3 fNormal, fPosition;

vec4 computeColor(vec3 position, vec3 normal) {
      // we are doing lighting in world coordinate frame

      vec3 lightDir = normalize(lightPosition - position);
      vec3 viewDir = normalize(cameraPosition - position);

      vec3 ambient = Ia*Ka ;
      vec3 diffuse = Id*Kd* max(0.0, dot(normal, lightDir));

      vec3 halfVector = normalize(lightDir + viewDir);
      vec3 specular = Is*Ks* pow( max(dot(halfVector, normal), 0.0), shininess);

      vec3 color = ambient + diffuse + specular;

      return vec4(color, 1.0);
}
```

Coding

```
void main(){
    vec4 wPos = TB*M*vPosition; // world position

    gl_Position = VP*wPos;
    gl_Position.z *= -1.0; // convert to Left Handed Coordinate System

    fPosition = wPos.xyz;

    vec3 normal;

    mat3 NT = TBN*N;

    if(shading == 1.0){ // Flat
     normal = normalize(NT*vNormal);
    }
    else { // Gouraud or Phong
     normal = normalize(NT*vPosition.xyz);
    }
    /* Note: we need to scale the normal vectors to
       unit length since the normal transformation
       matrix does not guarantee that. */

    if(shading==3.0){ // Phong
        fNormal = normal;
    }
    else{ // Flat or Gouraud
        fColor = computeColor(fPosition, normal);
    }
}
```

Fragment Shader:

```
precision highp float;

uniform vec3 Ka, Kd, Ks, Ia, Id, Is, lightPosition, cameraPosition;
uniform float shininess;
uniform float shading;

varying vec4 fColor;
varying vec3 fPosition, fNormal;

vec4 computeColor(vec3 position, vec3 normal) {
    // we are doing lighting in world coordinate frame

    vec3 lightDir = normalize(lightPosition - position);
    vec3 viewDir = normalize(cameraPosition - position);

    vec3 ambient = Ia*Ka ;
    vec3 diffuse = Id*Kd* max(0.0, dot(normal, lightDir));

    vec3 halfVector = normalize(lightDir + viewDir);
    vec3 specular = Is*Ks* pow( max(dot(halfVector, normal), 0.0), shininess);

    vec3 color = ambient + diffuse + specular;

    return vec4(color, 1.0);
}
```

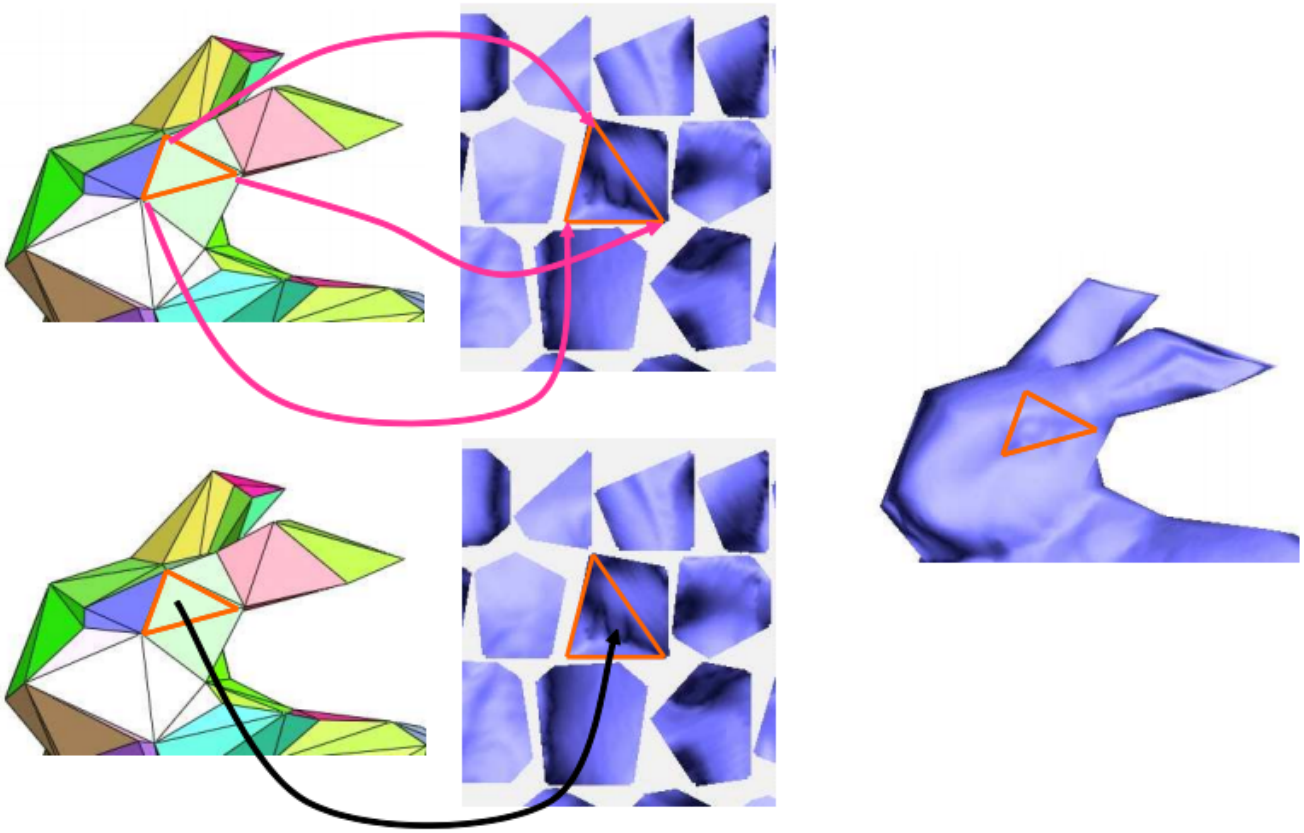
Coding

```
void main(){
    if(shading == 3.0){ // Phong
        gl_FragColor = computeColor(fPosition, normalize(fNormal));
    }
    else{
        gl_FragColor =  fColor;
    }
}
```

Texture Mapping

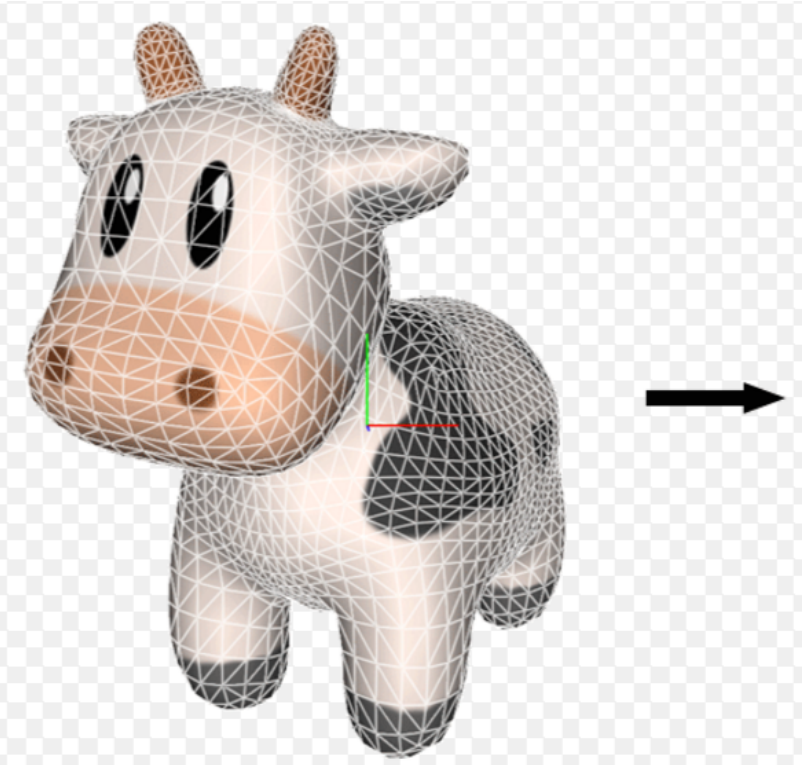
Idea: Glue an image into a surface.

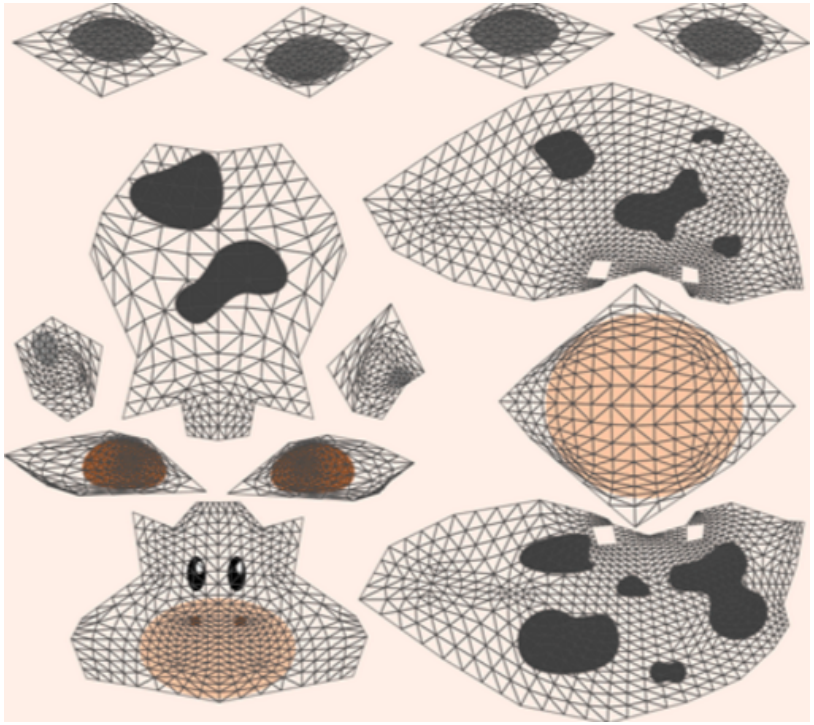
For each triangle in the mesh, we specify textute coordinates at the three vertices.



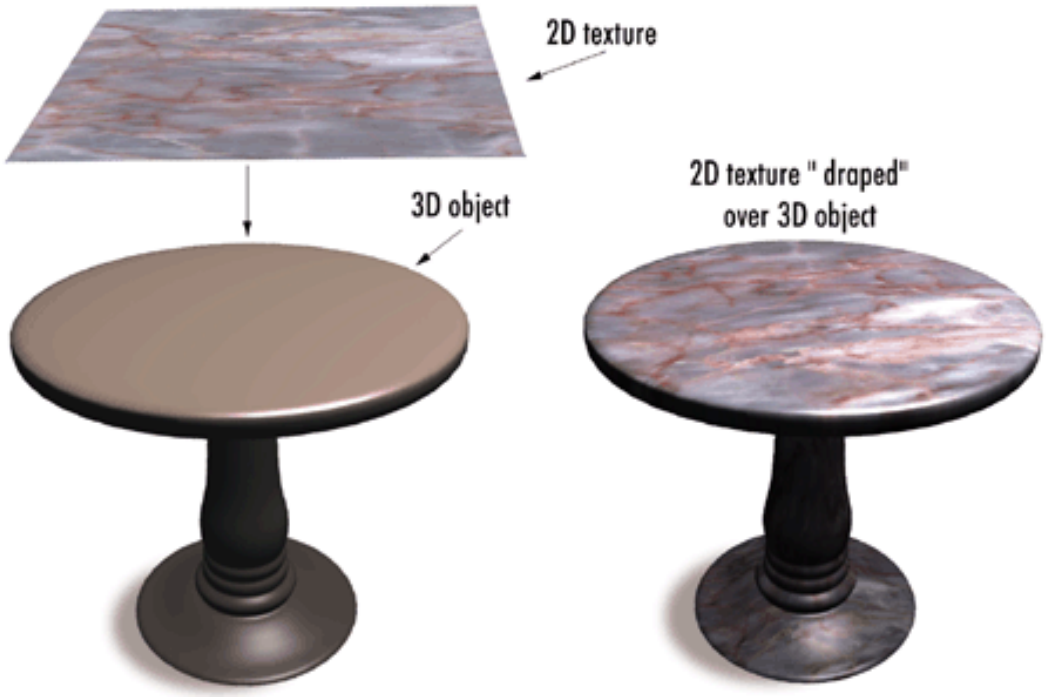
The texture coordinates of any point in the interior of a face is found by interpolation.

Texture Mapping

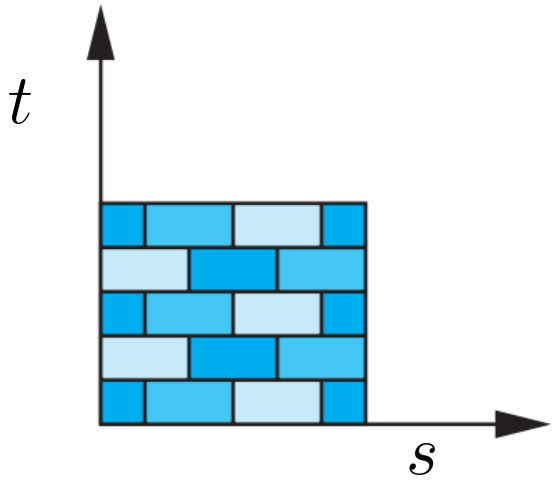




Texture Mapping



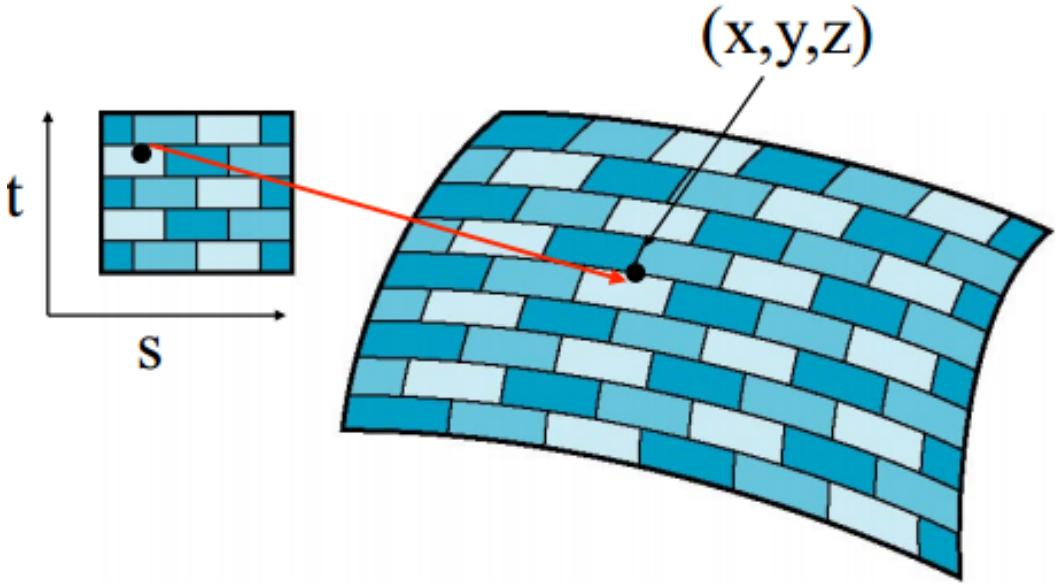
Texture Mapping



A point in the texture has coordinates s and t in the range $[0, 1]$.

Wrapping functions:

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



The inverse of the wrapping function gives us the texture coordinates.

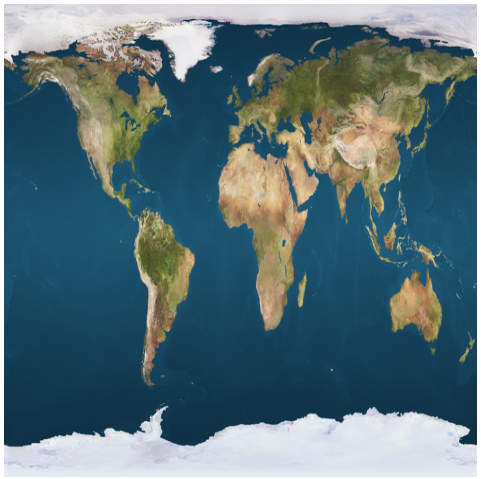
Texture Mapping

To do texture mapping we need the inverse wrapping functions.

We have the coordinates x, y and z of a point in $3D$.
We want to know its texture coordinates.

Example:

Texture Mapping



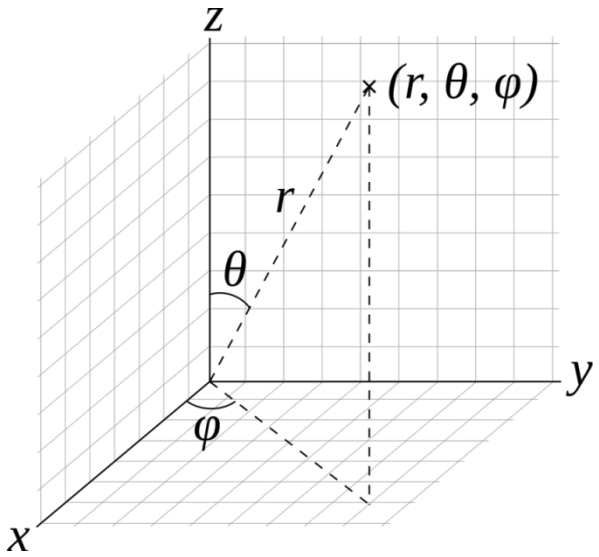
Assume the sphere has $radius = r$, $center = (0, 0, 0)$

Computing x, y and z from r, θ and ϕ:

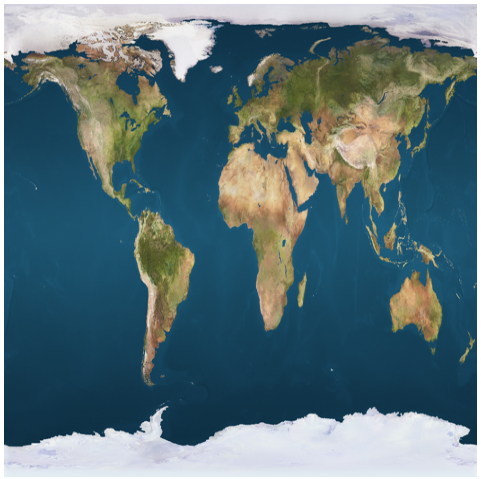
$$x = r\ sin\ \theta\ cos\ \phi$$

$$y = r sin\ \theta\ sin\phi$$

$$z = r\ cos\ \theta$$



Texture Mapping



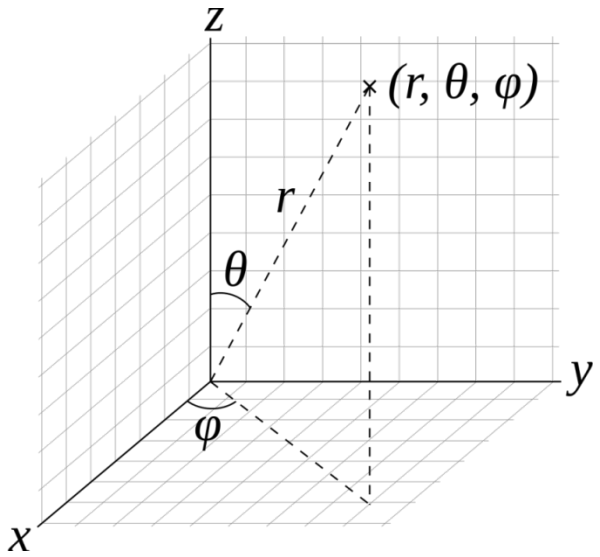
Computing r, θ and ϕ from x, y and z:

$$r = \sqrt{x^2 + y^2 + z^2}$$

$$\theta = atan2(\sqrt{x^2 + y^2}, z) \quad \in [0, \pi]$$

$$\phi' = atan2(y, x) \qquad \in [-\pi, \pi]$$

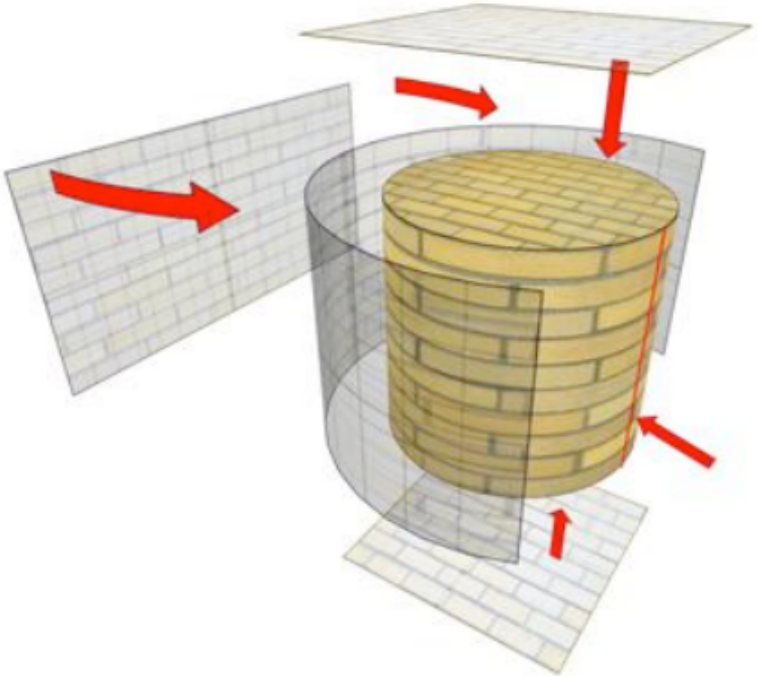
$$\phi = \begin{cases} \phi' \text{ if } \phi' \geq 0 \\ \phi' + 2\pi \text{ if } \phi' < 0 \end{cases}$$

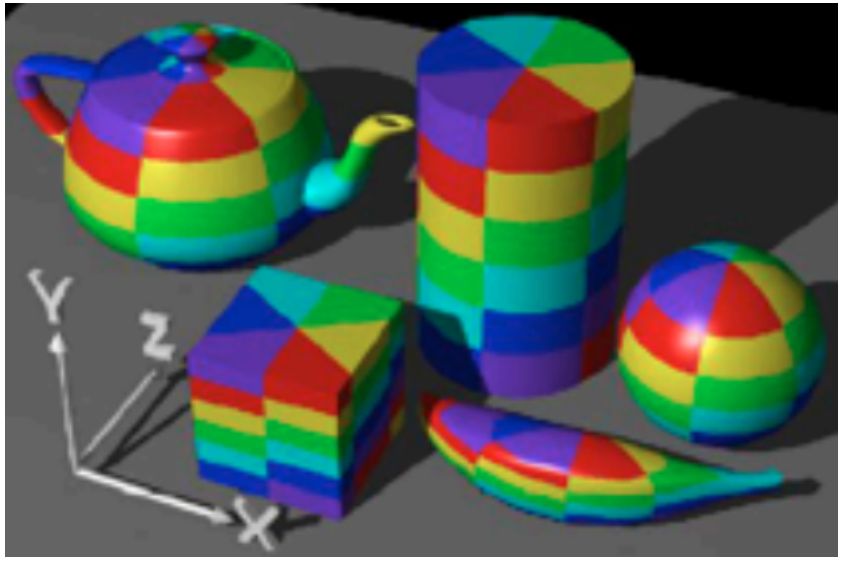


Texture coordinates of (x, y, z): $\quad s = \phi/2\pi \qquad t = 1 - \theta/\pi$

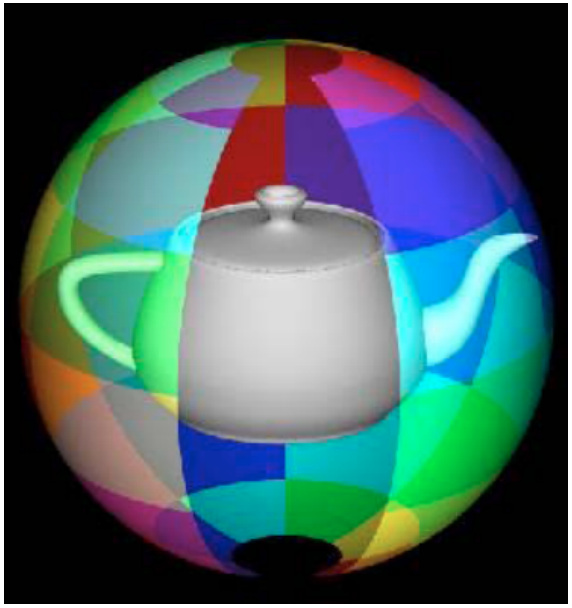
Texture Mapping

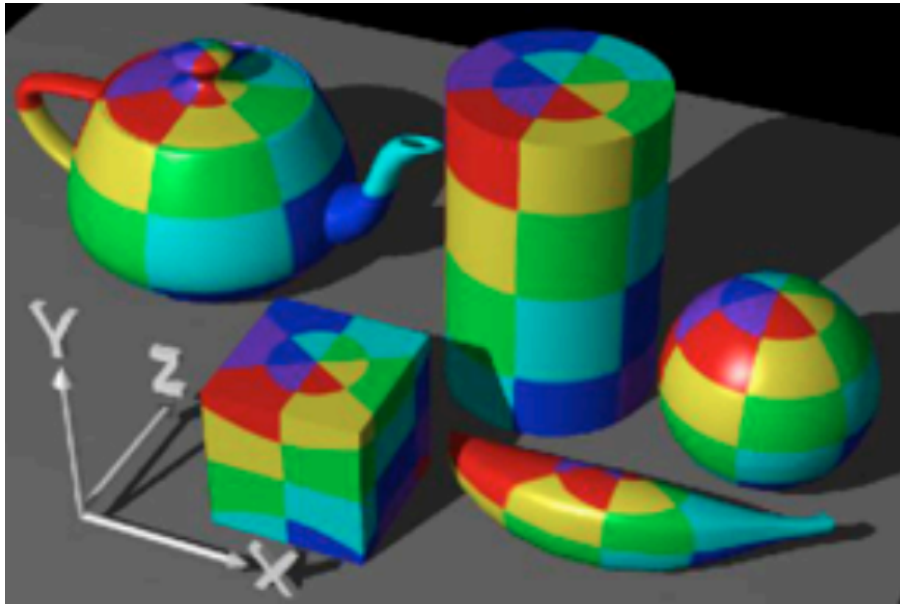
Cylindrical



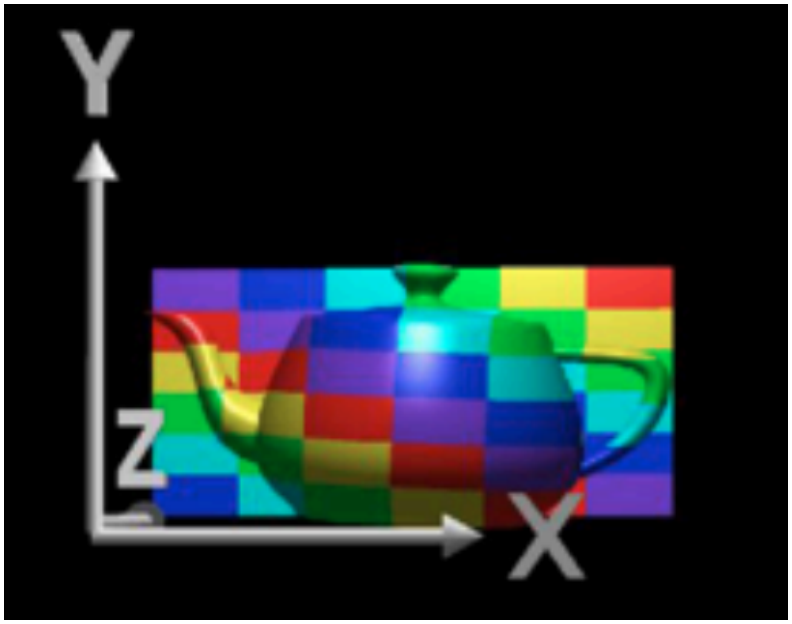


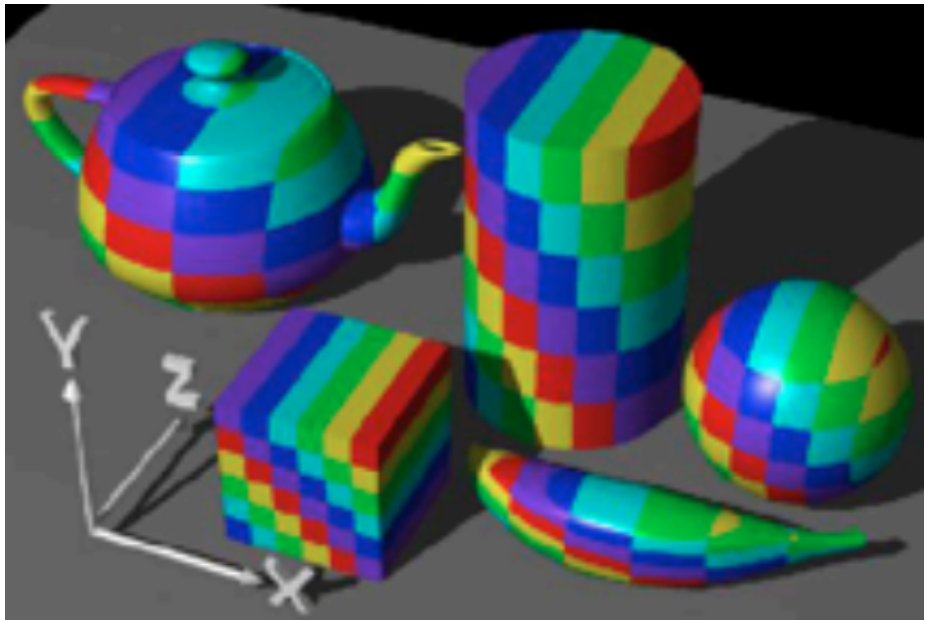
Spherical





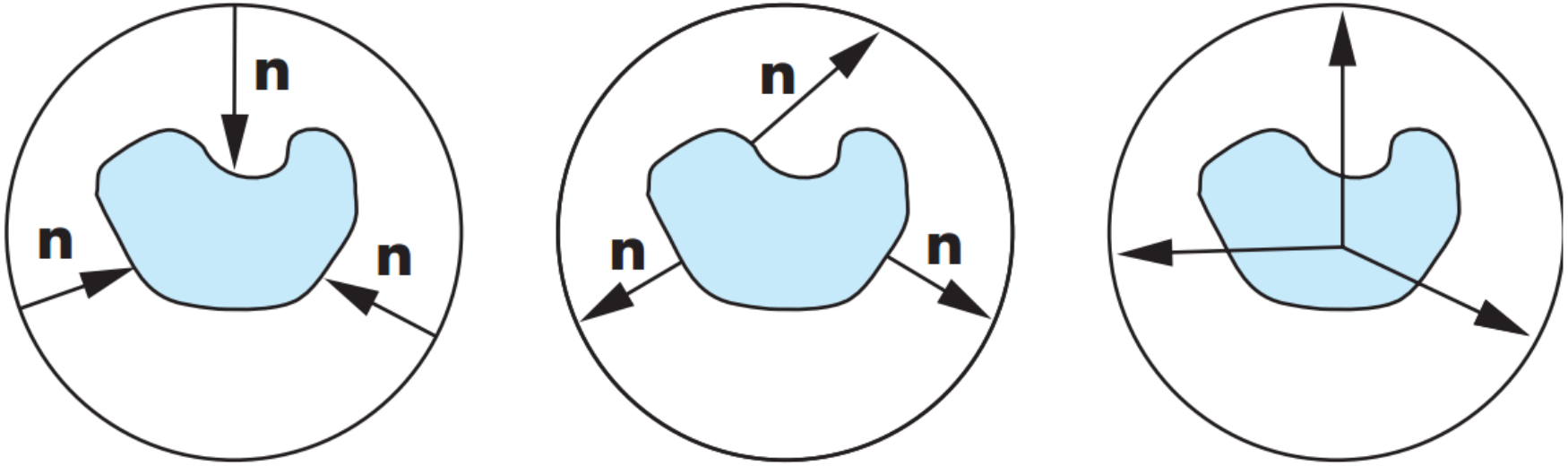
Box





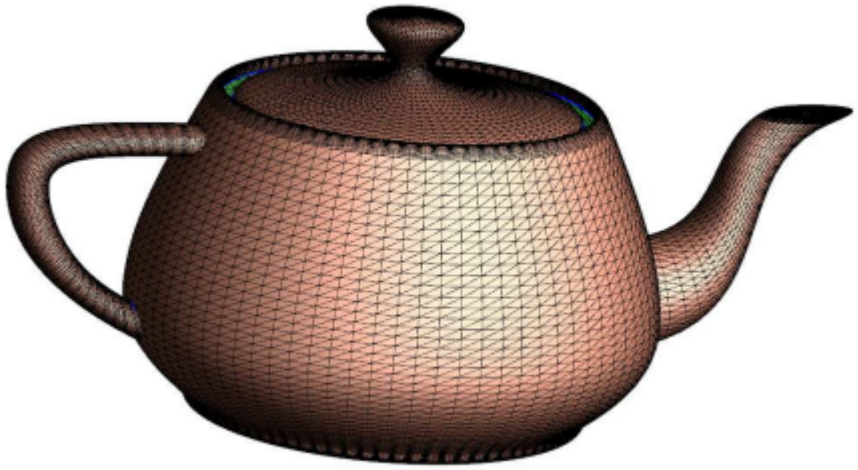
Texture Mapping

There are several ways of mapping a point on the surface of the object to a point on the surrounding surface:



Normal Mapping

The r, g, b values are interpreted as the coordinates of the normal at the point.



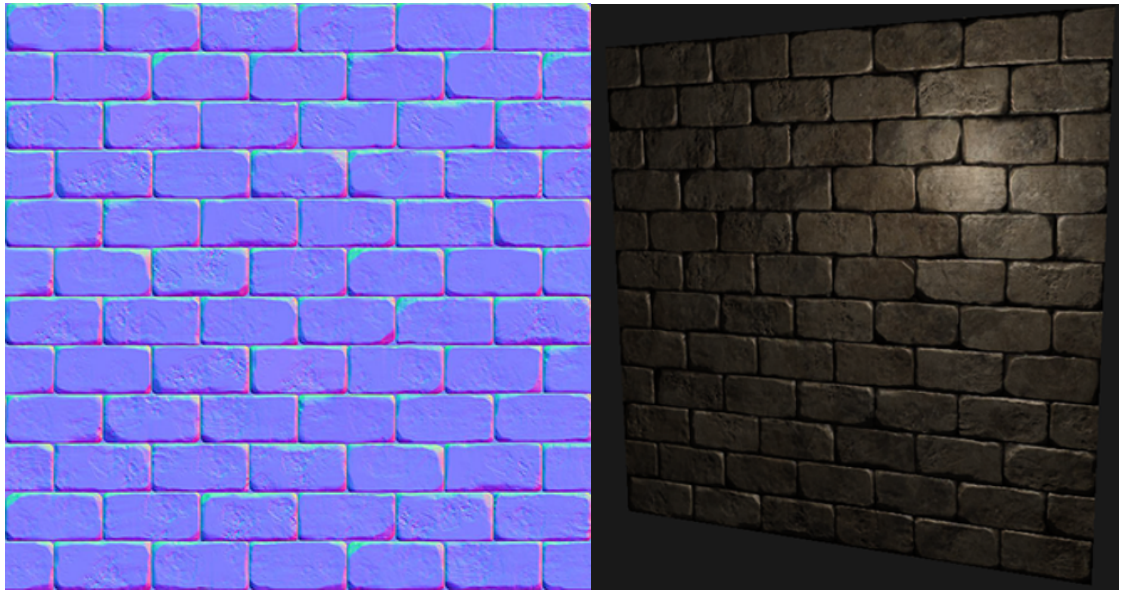
r, g, b values are in the range $[0, 1]$.

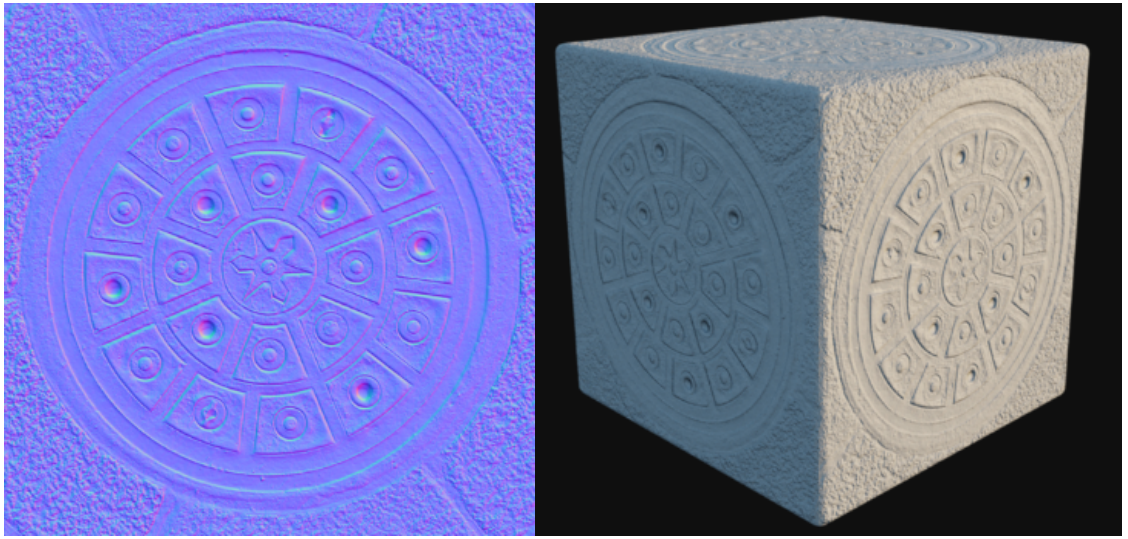
Coordinates of a normal vector are in the range $[-1, 1]$.

Mapping: `normal.x =2*r - 1` (similarly for `y` and `z`).

Normal Mapping

Examples:





Cube Mapping

Model things at a distance around objects being rendered.

Cube Map

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:

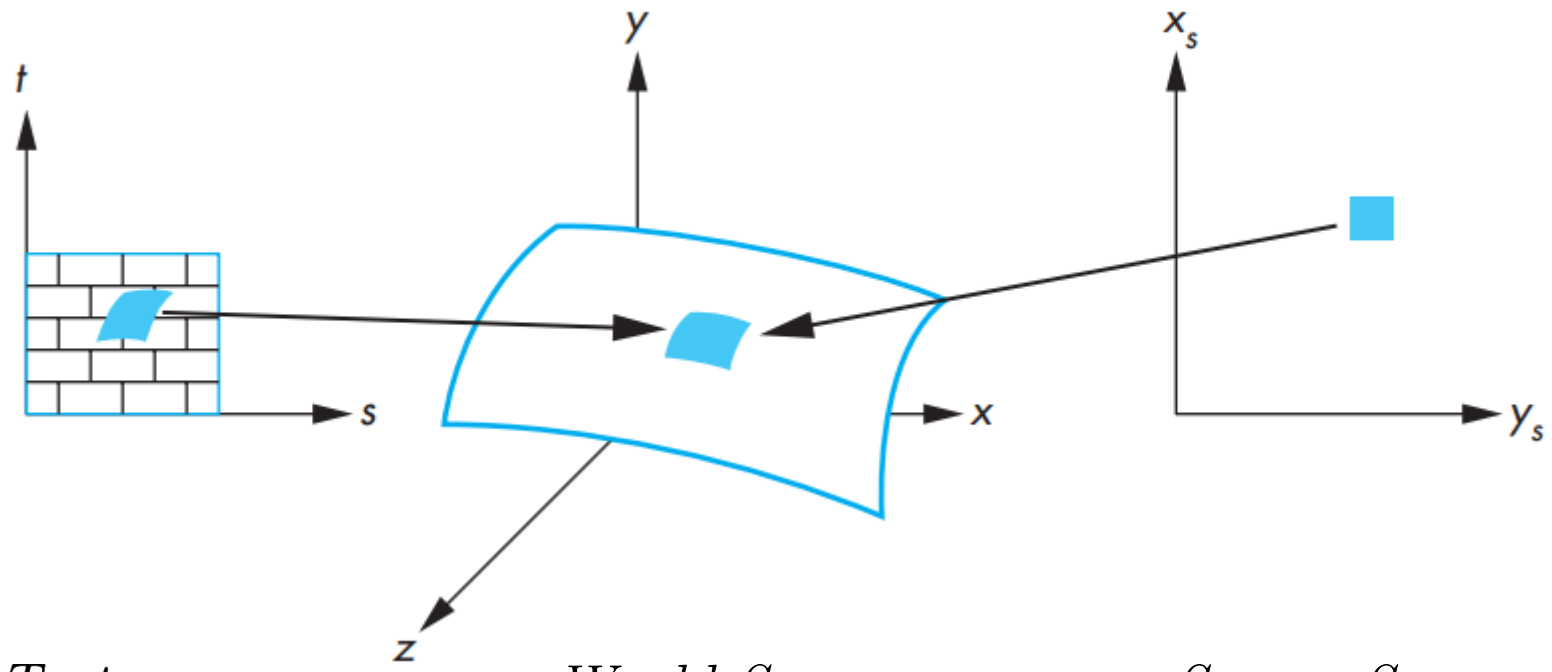
`normals[i]` and `texCoords[i]` correspond to `position[i]`

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

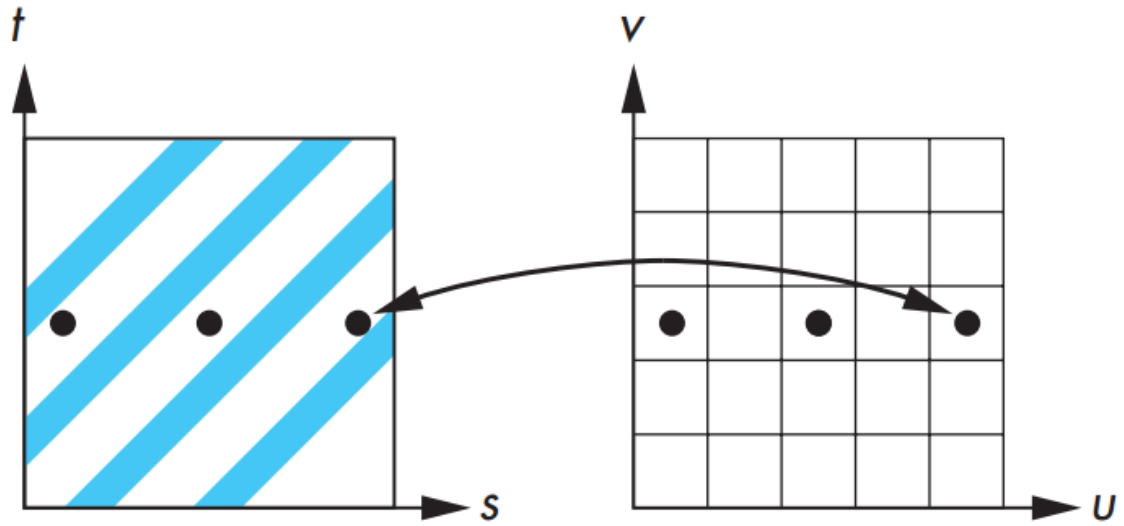
Texture Mapping

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.



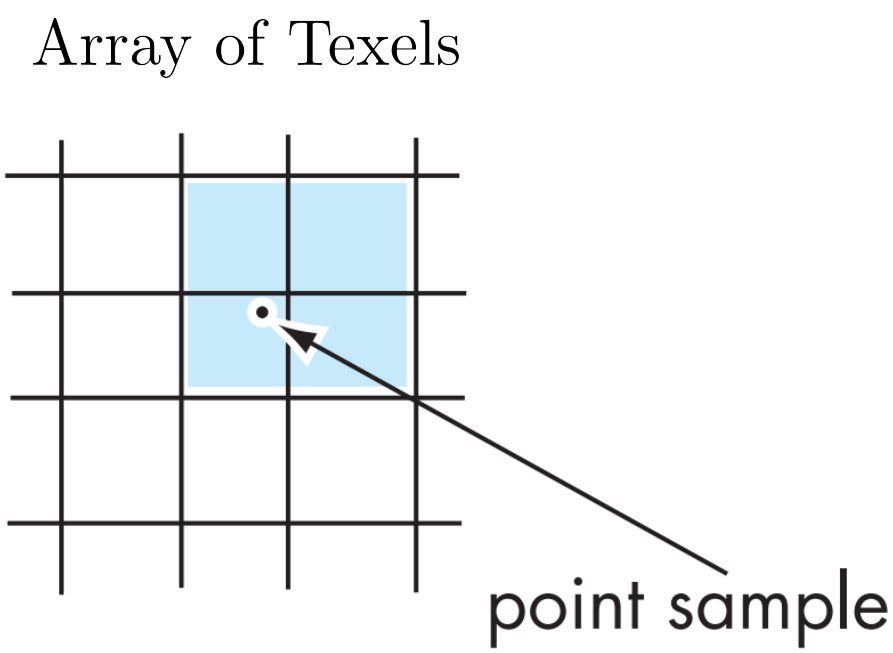
Texture Mapping



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.



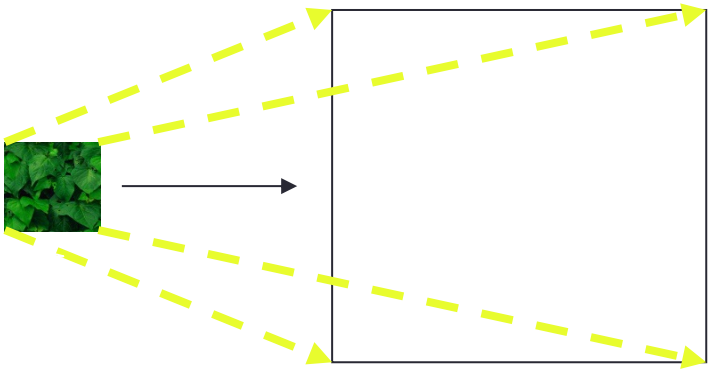
point sampling: Choose the texel whose center is closest to the texture coordinates.

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

Texture Mapping

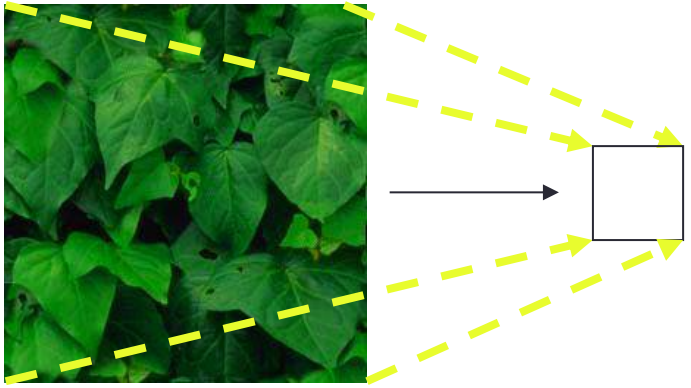
Magnification:

a few texels map to many pixels



Minification:

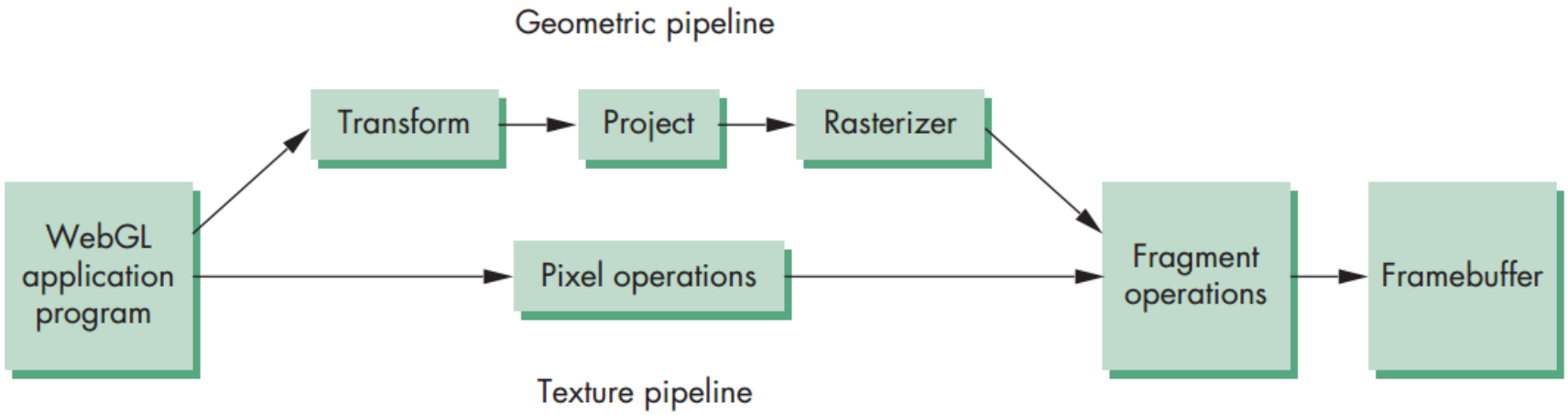
many texels map to a few pixels



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:

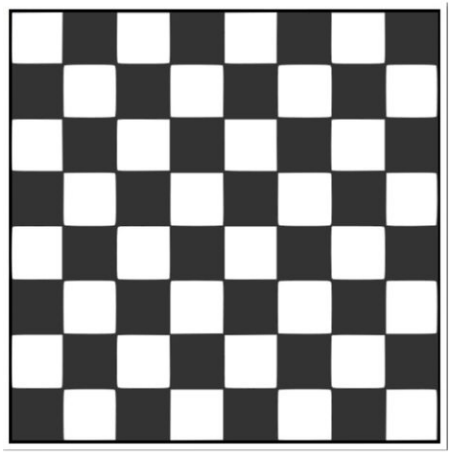
```
var texSize = 64;
var numRows = 8;
var numCols = 8;

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 can 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:

```
var texture = createTexture();
gl.uniform1i(gl.getUniformLocation(program, "texMap"), 0);
```

use default texture unit

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:

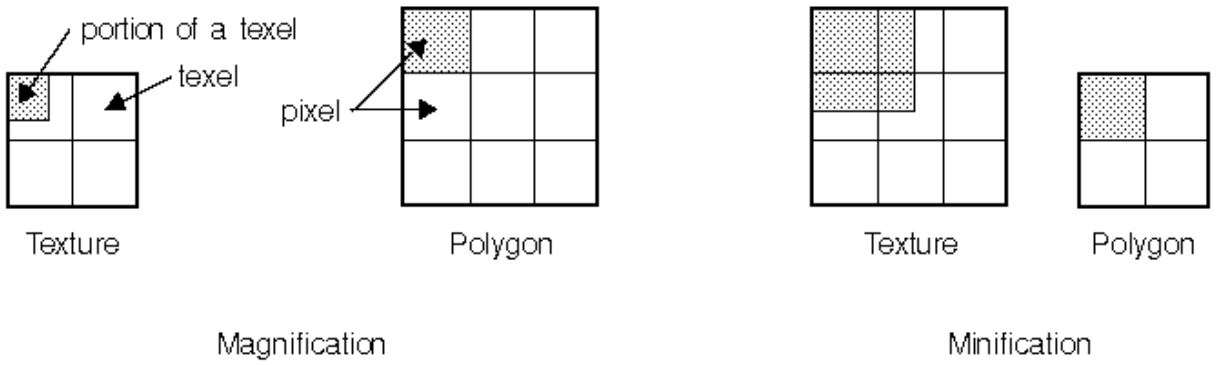
minification and magnification filters

```
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:

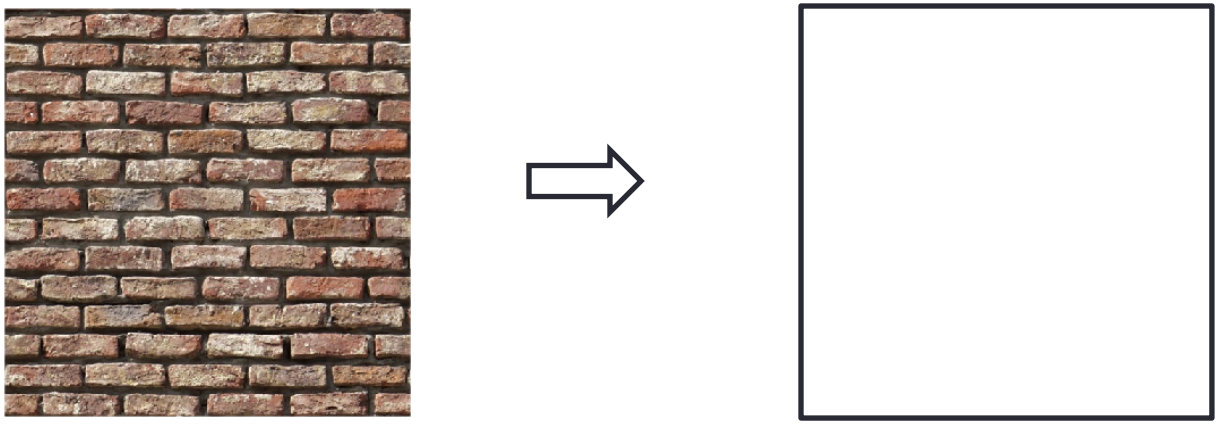
`gl.NEAREST_MIPMAP_NEAREST`: point sampling with best mipmap

`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



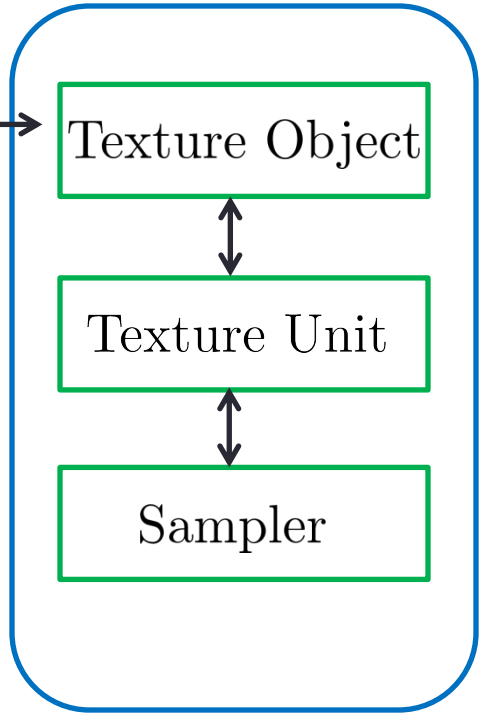
We need:

- positions of the vertices
- texture coordinates of the vertices

Texture Mapping

Image object

create an image object in Javascript.



create a WebGL texture object

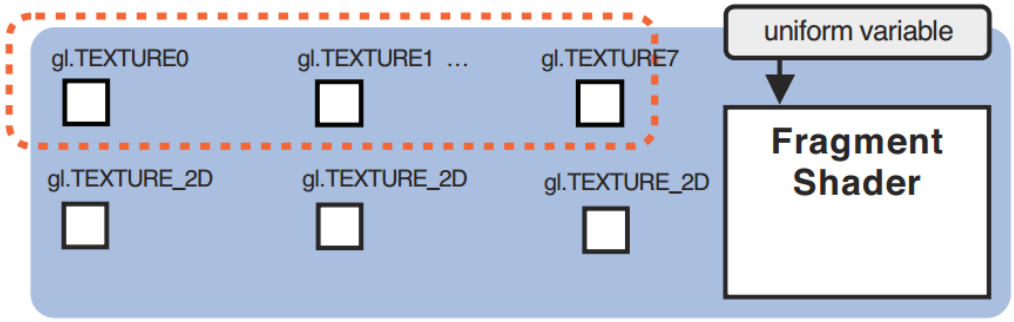
associate the image with it

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);

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);
```

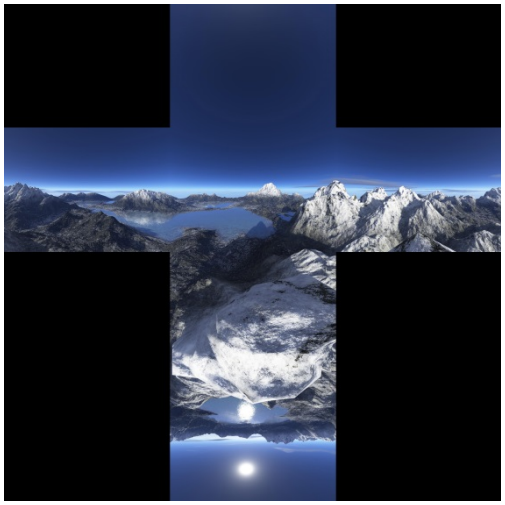
Texture Mapping

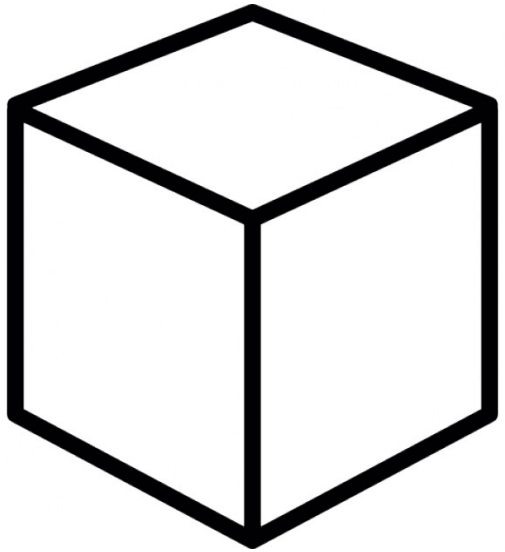
```
    var texture = gl.createTexture();
    var mySampler = gl.getUniformLocation(program, "mySampler");

    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);

    }
};
```

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>

<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>
```

Cube Map





What are the texture coordinates of the corners?

How would you describe the object in Wavefront Obj format?