Chapter 7
Transformations in 3D

Dr. Terence van Zyl

University of the Witwatersrand

11th May 2016

Transformations in 3D

Introduction

Transformations in 3D are also similar to 2D, but for transformations the increase in complexity that comes with the third dimension is substantial.

About Shader Scripts

Introduction

Before we begin working more seriously with WebGL, it will be nice to have a better way to include shader source code on a web page

Example

```
<script type="x-shader/x-vertex" id="vshader">
    attribute vec3 a_coords;
    uniform mat4 modelviewProjection;
    void main() {
        vec4 coords = vec4(a_coords,1.0);
        gl_Position = modelviewProjection * coords;
    }
</script>
```

Example

```
1  function getTextContent( elementID ) {
2      var element = document.getElementById(elementID);
3      var node = element.firstChild;
4      var str = "";
5      while (node) {
6          if (node.nodeType == 3) // this is a text node
7              str += node.textContent;
8          node = node.nextSibling;
9      }
10     return str;
11 }
```

Introducing glMatrix

Introduction

Transformations are essential to computer graphics. The WebGL API does not provide any functions for working with transformations.

Definition

glMatrix An open-source JavaScript library for vector and matrix maths in two and three dimensions.

Fact

The glMatrix library defines what it calls "classes" named vec2, vec3, and vec4 for working with vectors of 2, 3, and 4 numbers.

Fact

It defines mat3 for working with 3-by-3 matrices and mat4 for 4-by-4 matrices.

Fact

if the glMatrix documentation says that a parameter should be of type vec3, it is OK to pass either a Float32Array or a regular JavaScript array of three numbers as the value of that parameter.

Fact

Note that it is also the case that either kind of array can be used in WebGL functions such as glUniform3fv() and glUniformMatrix4fv(). glMatrix is designed to work with those functions. For example, a mat4 in glMatrix is an array of length 16 that holds the elements of

Example (javascript: creating a glmatrix or vec3)

```
1  transform = mat4.create();
2  vector = vec3.create();
3
4  var modelview = mat4.create();
5  saveTransform = mat4.clone(modelview);
```

Fact

Most other functions do not create new arrays. Instead, they modify the contents of their first parameter.

Example (javascript: glmatrix multiply operation)

```
1 mat4.multiply(A,B,C);
2 mat4.multiply(A,A,B);
```

Example (Functions for multiplying a matrix by standard transformations such as scaling and rotation)

```
1  //equivalent to calling glTranslatef(dx,dy,dz) in OpenGL
2  mat4.translate( modelview, modelview, [dx,dy,dz] );
3
4  mat4.scale( modelview, modelview, [sx,sy,sz] );
5
6  mat4.rotateX( modelview, modelview, radians );
7  mat4.rotateY( modelview, modelview, radians );
8  mat4.rotateZ( modelview, modelview, radians );
9  mat4.rotate( modelview, modelview, radians, [dx,dy,dz] );
```

Fact

Next we need a stack for the equivalent of glPushMatrix() and glPopMatrix().

Example (So, we can create the stack as an empty array)

```
1 var matrixStack = [];
```

Example (We can then push a copy of the current modelview matrix onto the stack by saying)

```
1 matrixStack.push( mat4.clone(modelview) );
```

Example (and we can remove a matrix from the stack and set it to be the current modelview matrix with)

```
1 modelview = matrixStack.pop();
```

Fact

The starting point for the modelview transform is usually a viewing transform. Something like glLoadIdentity(); gluLookAt(eyex,eyey,eyez,refx,refy,refz,upx,upy,upz);

Example (javascript: the glmatrix lookat)

```
1  mat4.lookAt( modelview, [eyex,eyey,eyez], [refx,refy,refz], [upx,upy,upz] );
```

Example (javascript: using glmatrix to set up other viewing transforms.)

```
1 mat4.ortho( projection, left, right, bottom, top, near, far );
2 mat4.frustum( projection, left, right, bottom, top, near, far );
3 mat4.perspective( projection, fovyInRadians, aspect, near, far );
```

Transforming Coordinates

Introduction

The point of making a projection and a modelview transformation is to use them to transform coordinates while drawing primitives.

Example (GLSL: combined model view projection transform shader)

```
1  attribute vec3 a_coords;            // (x,y,z) object coordinates of vertex.
2  uniform mat4 modelviewProjection;  // Combined transformation matrix.
3  void main() {
4      vec4 coords = vec4(a_coords,1.0);   // Add 1.0 for the w-coordinate.
5      gl_Position = modelviewProjection * coords;  // Transform the coordinates.
6  }
```

Demo

webgl/glmatrix-cube-unlit.html [source]

Example

```
1  var projection = mat4.create();  // projection matrix
2  var modelview = mat4.create();   // modelview matrix
3  var modelviewProjection = mat4.create();  // combined matrix
```

Example

```
1  u_modelviewProjection = gl.getUniformLocation(prog, "modelviewProjection");
```

Example

```
1  /* Set the value of projection to represent the projection transformation */
2
3  if (document.getElementById("persproj").checked) {
4       mat4.perspective(projection, Math.PI/5, 1, 4, 8);
5  }
6  else {
7       mat4.ortho(projection, -2, 2, -2, 2, 4, 8);
8  }
9
10 /* Set the value of modelview to represent the viewing transform. */
11 mat4.lookAt(modelview, [2,2,6], [0,0,0], [0,1,0]);
12
13 /* Apply the modeling tranformation to modelview. */
14 mat4.rotateX(modelview, modelview, rotateX);
15 mat4.rotateY(modelview, modelview, rotateY);
16 mat4.rotateZ(modelview, modelview, rotateZ);
17
18 /* Multiply the projection matrix times the modelview matrix to give the
19    combined transformation matrix, and send that to the shader program. */
20
21 mat4.multiply( modelviewProjection, projection, modelview );
22 gl.uniformMatrix4fv(u_modelviewProjection, false, modelviewProjection );
```

Example (GLSL: minimal vertex shader with separate modelview and projection transform)

```
1  attribute vec3 a_coords;  // (x,y,z) object coordinates of vertex.
2  uniform mat4 modelview;   // Modelview transformation.
3  uniform mat4 projection;  // Projection transformation
4  void main() {
5      vec4 coords = vec4(a_coords,1.0);      // Add 1.0 for w-coordinate.
6      vec4 eyeCoords = modelview * coords;   // Apply modelview transform.
7      gl_Position = projection * eyeCoords;  // Apply projection transform.
8  }
```

Transforming Normals

Introduction

Normal vectors are essential for lighting calculations (Subsection 4.1.3). When a surface is transformed in some way, it seems that the normal vectors to that surface will also change. However, that is not true if the transformation is a translation.

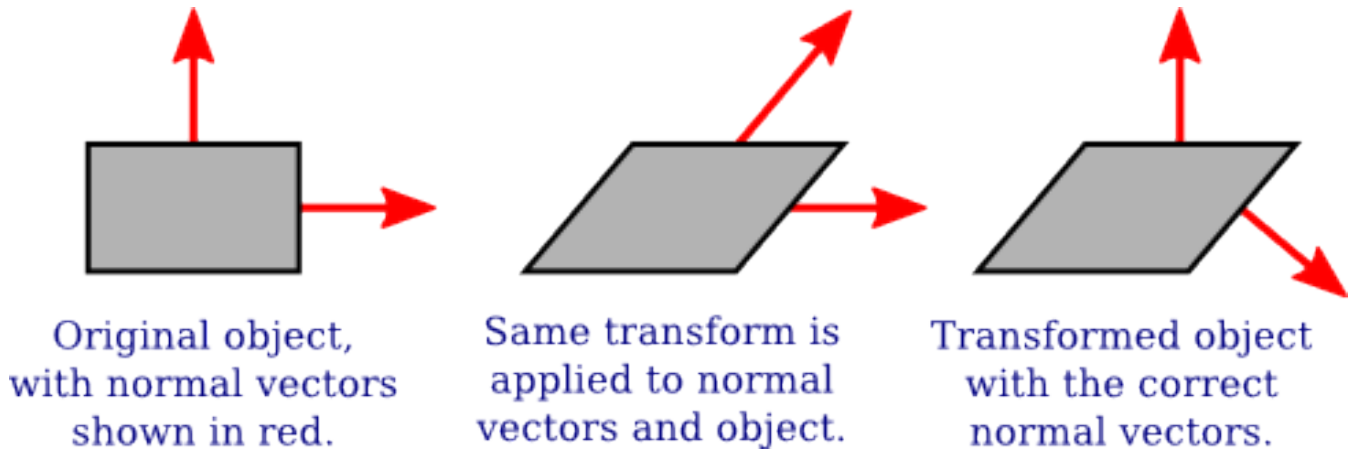


Figure: Incorrect transformation of normals for shear.

Fact

it is possible to get the correct transformation matrix for normal vectors from the coordinate transformation matrix. It turns out that you need to drop the fourth row and the fourth column and then take something called the "inverse transpose" of the resulting 3-by-3 matrix.

Example (javascript: built-in glmatrix operation for normal transforms.)

```
1 //normalMatrix is a mat3 that already exists.
2 mat3.normalFromMat4( normalMatrix, coordinateMatrix );
```

Example (GLSL: vertex shader for lights)

```
1  attribute vec3 a_coords;   // Untransformed object coordinates.
2  attribute vec3 normal;     // Normal vector.
3  uniform mat4 projection;   // Projection transformation matrix.
4  uniform mat4 modelview;    // Modelview transformation matrix.
5  uniform mat3 normalMatrix; // Transform matrix for normal vectors.
6    .
7    .  // Variables to define light and material properties.
8    .
9  void main() {
10     vec4 coords = vec4(a_coords,1.0);  // Add a 1.0 for the w-coordinate.
11     vec4 eyeCoords = modelview * coords;  // Transform to eye coordinates.
12     gl_Position = projection * eyeCoords;  // Transform to clip coordinates.
13     vec3 transformedNormal = normalMatrix*normal;  // Transform normal vector.
14     vec3 unitNormal = normalize(transformedNormal);  // Normalize.
15        .
16        .  // Use eyeCoords, unitNormal, and light and material
17        .  // properties to compute a colour for the vertex.
18        .
19 }
```

Rotation by Mouse

Introduction

Computer graphics is a lot more interesting when there is user interaction. The 3D experience is enhanced considerably just by letting the user rotate the scene, to view it from various directions.

Fact

A SimpleRotator keeps track of a viewing transformation that changes as the user rotates the scene. The most important function is rotator.getViewMatrix(). This function returns an array of 16 numbers representing the matrix for the viewing transformation in column-major order. The matrix can be sent directly to the shader program using gl.uniformMatrix4fv, or it can be used with functions from the glMatrix library as the initial value of the modelview matrix.

Demo

webgl/cube-with-simple-rotator.html [source]

Fact

1. *Move the view center to the origin: Translate by (-tx,-ty,-tz).*
2. *Rotate the scene by ry radians about the y-axis.*
3. *Rotate the scene by rx radians about the x-axis.*
4. *Move the origin back to view center: Translate by (tx,ty,tz).*
5. *Move the scene away from the viewer: Translate by (0,0,-d).*

Example

```
1 viewmatrix = mat4.create();
2 mat4.translate(viewmatrix, viewmatrix, [0,0,-d]);
3 mat4.translate(viewmatrix, viewmatrix, [tx,ty,tz]);
4 mat4.rotateX(viewmatrix, viewmatrix, rx);
5 mat4.rotateY(viewmatrix, viewmatrix, ry);
6 mat4.translate(viewmatrix, viewmatrix, [-tx,-ty,-tz]);
```

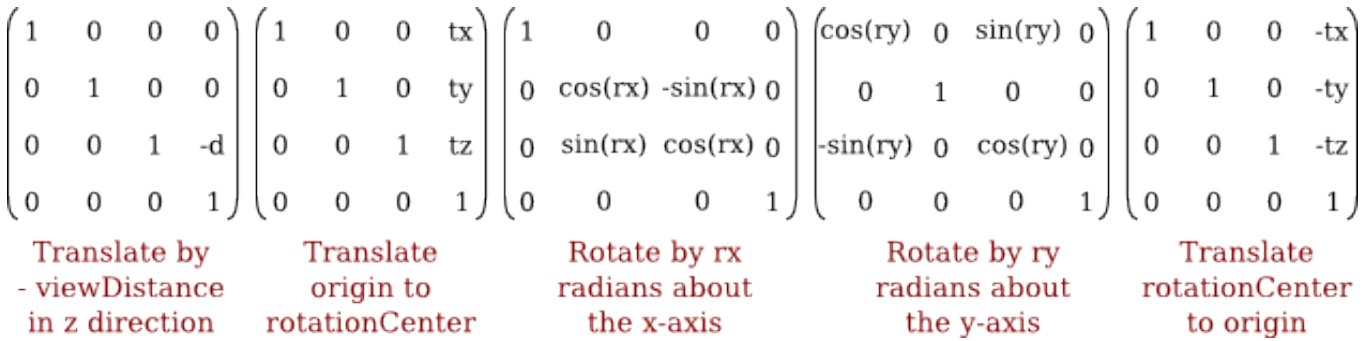


Figure: The matrices

Lighting and Material

Introduction

The goal of the lighting equation is to compute a colour for a point on a surface. The inputs to the equation include the material properties of the surface and the properties of light sources that illuminate the surface. The angle at which the light hits the surface plays an important role. The angle can be computed from the direction to the light source and the normal vector to the surface.

Definition

material The properties of an object that determine how that object interacts with light in the environment. Material properties in OpenGL include, for example, diffuse colour, specular colour, and shininess.

normal vector A normal vector to a surface at a point on that surface is a vector that is perpendicular to the surface at that point.

specular reflection Mirror-like reflection of light rays from a surface.Reflected as a ray in the direction that makes the angle of reflection equal to the angle of incidence. Can only be seen by a viewer whose position lies on the path of the reflected ray.

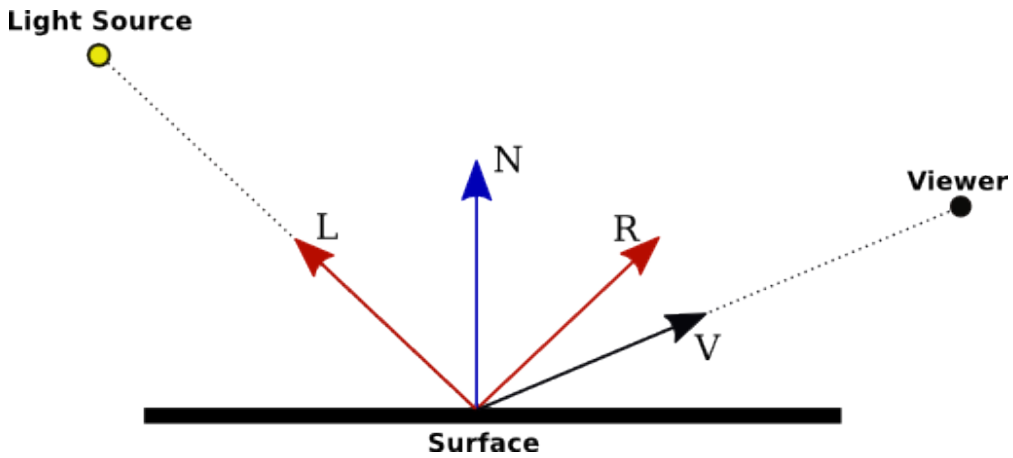


Figure: The vectors L, N, R, and V should be unit vectors. Recall that unit vectors have the property that the cosine of the angle between two unit vectors is given by the dot product of the two vectors.

Minimal Lighting

Introduction

Even very simple lighting can make 3D graphics look more realistic. For minimal lighting, a "viewpoint light," is a white light that shines from the direction of the viewer into the scene. In eye coordinates, L is $(0, 0, 1)$.

Definition

diffuse reflection Reflection of incident light in all directions from a surface, so that diffuse illumination of a surface is visible to all viewers, independent of the viewer's position.

Fact

$L = (0, 0, 1)$
N.L is the lighting in the N direction
Assumes N is in eye coordinates
$\therefore N \cdot L = N.z$

Example (GLSL: diffuse reflection from a directional light at (0,0,1))

```
1  attribute vec3 a_coords;              // Object coordinates for the vertex.
2  uniform mat4 modelviewProjection;     // Combined transformation matrix.
3  uniform bool lit;                   // Should lighting be applied?
4  uniform vec3 normal;                // Normal vector (in object coordinates).
5  uniform mat3 normalMatrix;          // Transformation matrix for normal vectors.
6  uniform vec4 colour;                 // Basic (diffuse) colour.
7  varying vec4 v_colour;               // Color to be sent to fragment shader.
8  void main() {
9      vec4 coords = vec4(a_coords,1.0);
10     gl_Position = modelviewProjection * coords;
11     if (lit) {
12         vec3 N = normalize(normalMatrix*normal); // Transformed unit normal
13         float dotProduct = abs(N.z);       //deal with back front
14         v_colour = vec4( dotProduct*colour.rgb, colour.a );
15     }
16     else {
17         v_colour = colour;
18     }
19 }
```

Fact

It would be easy to add ambient light to this model, using a uniform variable to specify the ambient light level. Emission colour is also easy.

Example (GLSL: more complex perspective projection light)

```
1  attribute vec3 a_coords;       // Object coordinates for the vertex.
2  uniform mat4 modelview;        // Modelview transformation matrix
3  uniform mat4 projection;       // Projection transformation matrix.
4  uniform bool lit;              // Should lighting be applied?
5  uniform vec3 normal;           // Normal vector (in object coordinates).
6  uniform mat3 normalMatrix;     // Transformation matrix for normal vectors.
7  uniform vec4 colour;            // Basic (diffuse) colour.
8  varying vec4 v_colour;          // Color to be sent to fragment shader.
9  void main() {
10     vec4 coords = vec4(a_coords,1.0);
11     vec4 eyeCoords = modelview * coords;
12     gl_Position = projection * eyeCoords;
13     if (lit) {
14         vec3 L = normalize( -eyeCoords.xyz );     // Points to light.
15         vec3 N = normalize(normalMatrix*normal); // Transformed unit normal.
16         float dotProduct = abs( dot(N,L) );
17         v_colour = vec4( dotProduct*colour.rgb, colour.a );
18     }
19     else {
20         v_colour = colour;
21     }
22 }
```

Specular Reflection and Phong Shading

Introduction

To add specular lighting to our basic lighting model, we need to work with the vectors R and V in the lighting diagram.

Fact

The formula for the contribution of specular reflection to the visible colour is $(R \cdot V)^s * \text{specularMaterialColor} * \text{lightIntensity}$ *where s is the specular exponent: (the material property called "shininess"). Now* $R = 2(N \cdot L)N - L)$ *implemented in GLSL reflect(I,N) however, the value of reflect(L,N) is* $-R$ *rather than* R

Example (GLSL: specular light in perspective projection)

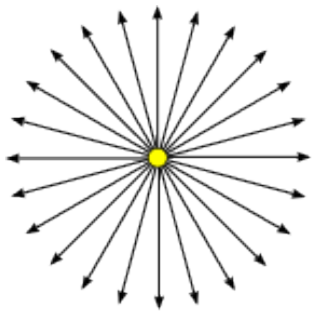
```
R = -reflect(L,N);
V = normalize( -eyeCoords.xyz );  // (Assumes a perspective projection.)
vec3 colour = dot(L,N) * diffuseMaterialColor.rgb * diffuseLightColor;
if (dot(R,V) > 0.0) {
    colour = colour + ( pow(dot(R,V),specularExponent) *
                        specularMaterialColor * specularLightColor );
}
```

Demo

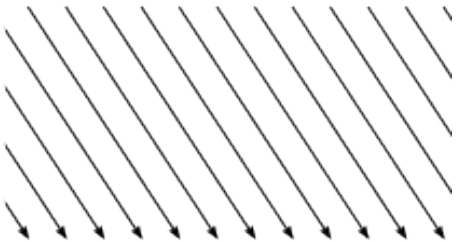
webgl/basic-specular-lighting.html

Fact

Following the OpenGL convention, lightPosition is a vec4. For a directional light, the w-coordinate is 0, and the eye coordinates of the light are lightPosition.xyz. If the w-coordinate is non-zero, the light is a point light, and its eye coordinates are lightPosition.xyz/lightPosition.w.



POINT LIGHT
emits light in
all directions.



DIRECTIONAL LIGHT
has parallel light rays, all
from the same direction.

Example (GLSL: diffuse intensity 0.8 and specular intensity 0.4 lighting)

```
1  attribute vec3 a_coords;
2  attribute vec3 a_normal;
3  uniform mat4 modelview;
4  uniform mat4 projection;
5  uniform mat3 normalMatrix;
6  uniform vec4 lightPosition;
7  uniform vec4 diffuseColor;
8  uniform vec3 specularColor;
9  uniform float specularExponent;
10 varying vec4 v_colour;
11 void main() {
12     vec4 coords = vec4(a_coords,1.0);
13     vec4 eyeCoords = modelview * coords;
14     gl_Position = projection * eyeCoords;
15     vec3 N, L, R, V;  // Vectors for lighting equation.
16     N = normalize( normalMatrix*a_normal );
```

Example (GLSL: diffuse intensity 0.8 and specular intensity 0.4 lighting)

```
1     if ( lightPosition.w == 0.0 ) { // Directional light.
2         L = normalize( lightPosition.xyz );
3     }
4     else { // Point light.
5         L = normalize( lightPosition.xyz/lightPosition.w - eyeCoords.xyz );
6     }
7     R = -reflect(L,N);
8     V = normalize( -eyeCoords.xyz);  // (Assumes a perspective projection.)
9     if ( dot(L,N) <= 0.0 ) {
10        v_colour = vec4(0,0,0,1);  // The vertex is not illuminated.
11    }
12    else {
13        vec3 colour = 0.8 * dot(L,N) * diffuseColor.rgb;
14        if (dot(R,V) > 0.0) {
15            colour += 0.4 * pow(dot(R,V),specularExponent) * specularColor;
16        }
17        v_colour = vec4(colour, diffuseColor.a);
18    }
19 }
```

Definition

per-vertex lighting Doing lighting calculations only at the vertices of a primitive, and interpolating the results to get the colours of interior pixels. Per-vertex lighting is the standard in traditional OpenGL. Per-vertex lighting without specular reflection is Lambert shading.

Lambert shading A technique for computing pixel colours on a primitive using a lighting equation that takes into account ambient and diffuse reflection. In Lambert shading, the lighting equation is applied only at the vertices of the primitive. Colour values for pixels in the primitive are calculated by interpolating the values that were computed for the vertices.

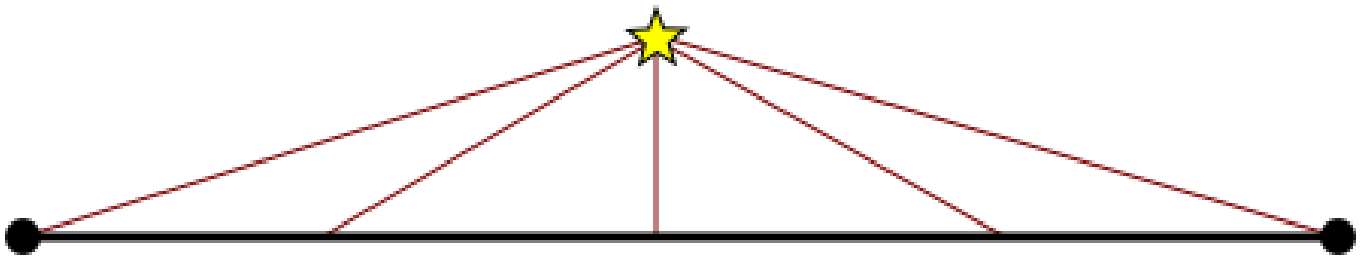


Figure: per-vertex and Lambert shading fail when the light source is close to the surface.

Definition

per-pixel lighting Doing lighting calculations at each pixel of a primitive, which gives better results in most cases than per-vertex lighting. Phong shading uses per-pixel lighting, with normal vectors interpolated from the vertices.

Phong shading A technique for computing pixel colours on a primitive using a lighting equation that takes into account ambient, diffuse, and specular reflection. In Phong shading, the lighting equation is applied at each pixel. Normal vectors are specified only at the vertices of the primitive. The normal vector that is used in the lighting equation at a pixel is obtained by interpolating the normal vectors for the vertices.

Fact

To do per-pixel lighting, certain data that is available in the vertex shader must be passed to the fragment shader in varying variables.

Demo

webgl/basic-specular-lighting-Phong.html

Demo

Per-vertex vs. Per-pixel Lighting

Adding Complexity

Introduction

Our shader programs are getting more complex. As we add support for multiple lights, additional light properties, two-sided lighting, textures, and other features, it will be useful to use data structures and functions to help manage the complexity.

Example (GLSL: It makes sense to define a struct to hold the properties of a light.)

```
1  struct LightProperties {
2     bool enabled;
3     vec4 position;
4     vec3 colour;
5  };
```

Example (GLSL: Material properties can also be represented as a struct.)

```
1 struct MaterialProperties {
2     vec3 diffuseColor;
3     vec3 specularColor;
4     float specularExponent;
5 };
```

Example (GLSL: function to help with the lighting calculation)

```
vec3 lightingEquation( LightProperties light,
                       MaterialProperties material,
                       vec3 eyeCoords, // Eye coordinates for the point.
                       vec3 N, // Normal vector to the surface.
                       vec3 V  // Direction to viewer.) {
    vec3 L, R; // Light direction and reflected light direction.
    if ( light.position.w == 0.0 ) { // directional light
        L = normalize( light.position.xyz );
    }
    else { // point light
        L = normalize( light.position.xyz/light.position.w - eyeCoords );
    }
    if (dot(L,N) <= 0.0) { // light does not illuminate the surface
        return vec3(0.0);
    }
    vec3 reflection = dot(L,N) * light.colour * material.diffuseColor;
    R = -reflect(L,N);
    if (dot(R,V) > 0.0) { // ray is reflected toward the the viewer
        float factor = pow(dot(R,V),material.specularExponent);
        reflection += factor * material.specularColor * light.colour;
    }
    return reflection;
}
```

Example (GLSL: full calculation of the lighting equation)

```
1  vec3 colour = vec3(0.0);  // Start with black (all colour components zero).
2  for (int i = 0; i < 4; i++) {  // Add in the contribution from light i.
3      if (lights[i].enabled) { // Light can only contribute colour if enabled.
4          colour += lightingEquation(
5                      lights[i], material,eyeCoords, normal, viewDirection );
6      }
7  }
```

Two-sided Lighting

Demo

webgl/parametric-function-grapher.html

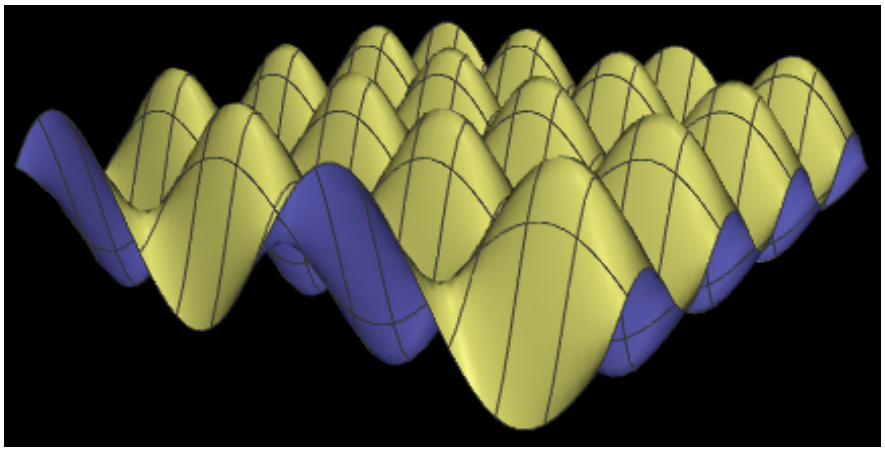


Figure: two sided parametric surface to light.

Fact

when the shader program does lighting calculations, how can does it know whether it's drawing a front face or a back face? That information comes from outside the shader program: The fragment shader has a built-in boolean variable named gl_FrontFacing whose value is set to true before the fragment shader is called, if the shader is working on the front face of a polygon

Example (GLSL: back and front lighting using glFrontFacing)

```
uniform MaterialProperties frontMaterial;
uniform MaterialProperties backMaterial;
uniform LightProperties lights[3];
uniform mat3 normalMatrix;
varying vec3 v_normal;
varying vec3 v_eyeCoords;
  .
vec3 normal = normalize( normalMatrix * v_normal );
vec3 viewDirection = normalize( -v_eyeCoords);
vec3 colour = vec3(0.0);
for (int i = 0; i < 3; i++) {
    if (lights[i].enabled) {
        if (gl_FrontFacing) {  // Computing colour for a front face.
            colour += lightingEquation( lights[i], frontMaterial, v_eyeCoords,
                                        normal, viewDirection);
        }
        else {  // Computing colour for a back face.
            colour += lightingEquation( lights[i], backMaterial, v_eyeCoords,
                                        -normal, viewDirection);
        }
    }
}
gl_FragColor = vec4(colour,1.0);
```

Fact

In some cases, you can be sure that no back faces are visible. This will happen when the objects are closed surfaces seen from the outside, and all the polygons face towards the outside. In such cases, it is wasted effort to draw back faces, since you can be sure that they will be hidden by front faces. The JavaScript command gl.enable(gl.CULL_FACE) tells WebGL to discard polygons without drawing them, based on whether they are front-facing or back-facing. The commands gl.cullFace(gl.BACK) and gl.cullFace(gl.FRONT) determine whether it is back-facing or front-facing polygons that are discarded when CULL_FACE is enabled; the default is back-facing.

Moving Lights

Introduction

In our examples so far, lights have been fixed with respect to the viewer. But some lights, such as the headlights on a car, should move along with an object. And some, such as a street light, should stay in the same position in the world, but changing position in the rendered scene as the point of view changes.

Fact

Lighting calculations are done in eye coordinates. When the position of a light is given in object coordinates or in world coordinates, the position must be transformed to eye coordinates, by applying the appropriate modelview transformation. The transformation can't be done in the shader program, because the modleview matrix in the shader program represents the transformation for the object that is being rendered, and that is almost never the same as the transformation for the light. The solution is to store the light position in eye coordinates. That is, the shader's uniform variable that represents the position of the light must be set to the position in eye coordinates.

Fact

For a light that is at a fixed position in world coordinates, the appropriate modelview transformation is the viewing transformation. The viewing transformation must be applied to the world-coordinate light position to transform it to eye coordinates.

Example (GLSL: using transformMat4 from glMatrix to transform the light position)

```
1  /* Set the position of a light, in eye coordinates.
2   * @param u_position_loc The uniform variable location for
3   *                       the position property of the light.
4   * @param modelview The modelview matrix that transforms object
5   *                  coordinates to eye coordinates.
6   * @param lightPosition The location of the light, in object
7   *                      coordinates (a vec4).
8   */
9  function setLightPosition( u_position_loc, modelview, lightPosition ) {
10     var transformedPosition = new Float32Array(4);
11     vec4.transformMat4( transformedPosition, lightPosition, modelview );
12     gl.uniform4fv( u_position_loc, transformedPosition );
13 }
```

Spotlights

Introduction

A spotlight emits only a cone of light. A spotlight is a kind of point light. The vertex of the cone is located at the position of the light. The cone points in some direction, called the spot direction. The spot direction is specified as a vector. The size of the cone is specified by a cutoff angle; light is only emitted from the light position in directions whose angle with the spot direction is less than the cutoff angle. Furthermore, for angles less than the cutoff angle, the intensity of the light ray can decrease as the angle between the ray and spot direction increases. The rate at which the intensity decreases is determined by a non-negative number called the spot exponent.

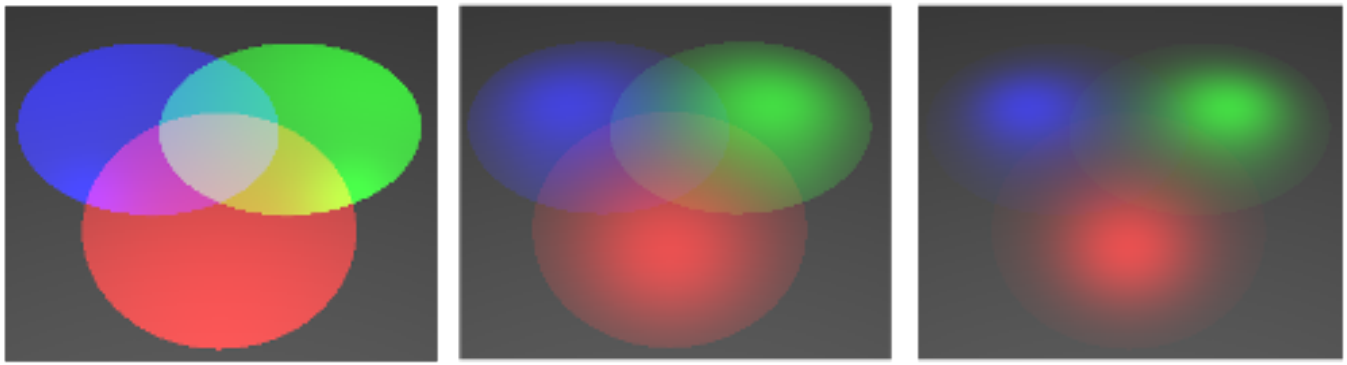


Figure: example spotlights

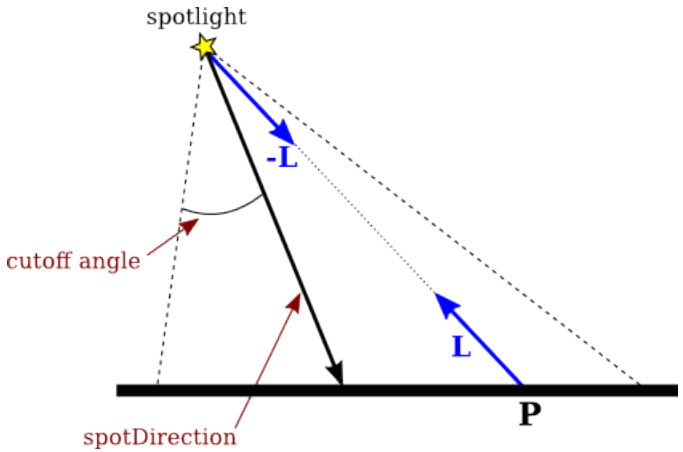


Figure:

Example (GLSL: a shader program)

```
1  struct LightProperties {
2      bool enabled;
3      vec4 position;
4      vec3 colour;
5      vec3 spotDirection;
6      float spotCosineCutoff;
7      float spotExponent;
8  };
9    .
10   .
11 L = normalize( light.position.xyz/light.position.w - v_eyeCoords );
12 if (light.spotCosineCutoff > 0.0) { // the light is a spotlight
13     vec3 D = normalize(light.spotDirection);  // unit vector!
14     float spotCosine = dot(D,-L);
15     if (spotCosine >= light.spotCosineCutoff) {
16         spotFactor = pow(spotCosine,light.spotExponent);
17     }
18     else { // The point is outside the cone of light from the spotlight.
19         return vec3(0.0); // The light will add no colour to the point.
20     }
21 }
22 // Light intensity will be multiplied by spotFactor
```

Light Attenuation

Introduction

There is one more general property of light to consider: attenuation. This refers to the fact that the amount of illumination from a light source should decrease with increasing distance from the light.

Definition

attenuation Refers to the way that illumination from a point light or spot light decreases with distance from the light. Physically, illumination should decrease with the square of the distance, but computer graphics often uses a linear attenuation with distance, or no attenuation at at all.

Fact

OpenGL 1.1 supports attenuation. The light intensity can be multiplied by $1.0/(a + b * d + c * d2)$, *where d is the distance to the light source, and a, b, and c are properties of the light. The numbers a, b, and c are called the "constant attenuation," "linear attenuation," and "quadratic attenuation" of the light source. By default, a is one, and b and c are zero, which means that there is no attenuation.*

Diskworld 2

Introduction

Demo

webgl/diskworld-2.html

Definition

multi-pass algorithm A rendering algorithm that draws a scene several times and combines the results somehow to compute the final image. A simple example is anaglyph stereo, in which a left-eye and right-eye image of the scene are rendered separately and combined.

Texture Transforms with glMatrix

Introduction

We need to compute the texture transform matrix on the JavaScript side. The transform matrix is then sent to a uniform matrix variable in the the shader program, where it can be applied to the texture coordinates.

Example (WebGL, glMatrix: texture transform)

```
1 var textureTransform = mat3.create();
2 mat3.scale( textureTransform, textureTransform, [2,2] );
3 gl.uniformMatrix3fv( u_textureTransform, false, textureTransform );
```

Example (GLSL: vertex shader texture coordinate transform)

```
1 vec3 texcoords = textureTransform * vec3(a_texCoords,1.0);
2 v_texCoords = texcoords.xy;
```

Demo

webgl/texture-transform.html (source)

Generated Texture Coordinates

Introduction

Texture coordinates are typically provided to the shader program as an attribute variable. However, when texture coordinates are not available, it is possible to generate them in the shader program. While the results will not usually look as good as using texture coordinates that are customized for the object that is being rendered, they can be acceptable in some cases.

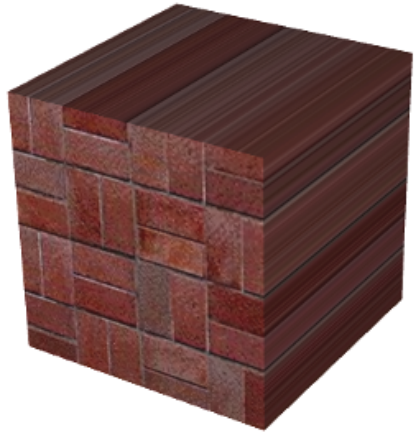


Figure: Generate Texture Coordinates as XY

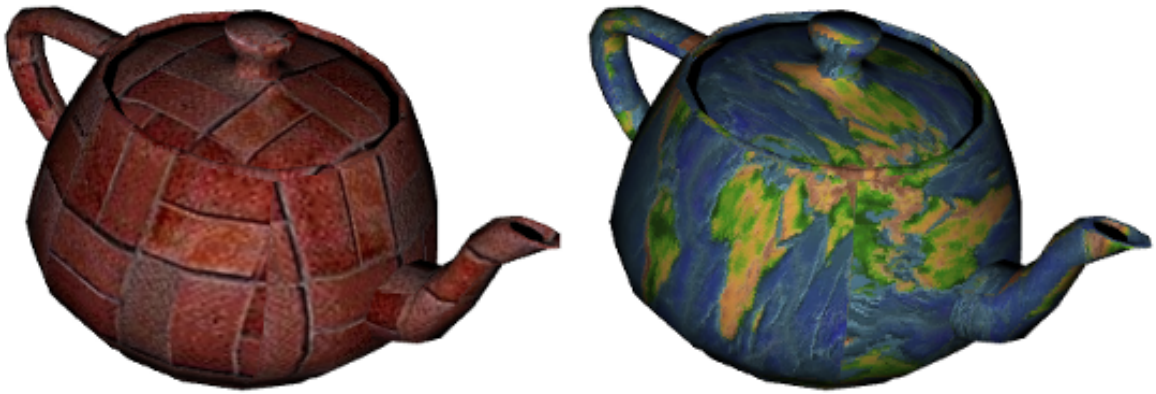


Figure: Generate Texture Coordinates as cubical using largest direction of normals

Demo

Generate Texture Coordinates

Procedural Textures

Introduction

There are other ways to define such functions besides looking up values in an image. A procedural texture is defined by a function whose value is computed rather than looked up. That is, the texture coordinates are used as input to a code segment whose output is a colour value.

Definition

procedural texture A texture for which the value at a given set of texture coordinates is computed as a mathematical function of the coordinates, as opposed to an image texture where the value is obtained by sampling an image.

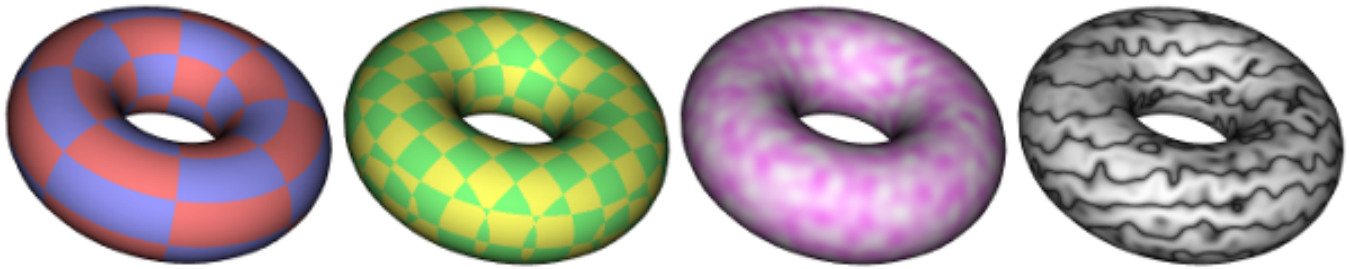


Figure: Some procedural textures generated in the fragment shader

Example (GLSL: fragment shader snippit for a checker board)

```
1  vec4 colour;
2  float a = floor(v_texCoords.x * scale);
3  float b = floor(v_texCoords.y * scale);
4  if (mod(a+b, 2.0) > 0.5) {  // a+b is odd
5      colour = vec3(1.0, 0.5, 0.5, 1.0); // pink
6  }
7  else {  // a+b is even
8      colour = vec3(0.6, 0.6, 1.0, 1.0); // light blue
9  }
```

Demo

2D and 3D Procedural Textures in WebGL

Bumpmaps

Introduction

So far, the only textures that we have encountered have affected colour. Whether they were image textures, environment maps, or procedural textures, their effect has been to vary the colour on the surfaces to which they were applied. But, more generally, texture can refer to variation in any property.

Definition

bumpmapping Using a texture to modify the normal vectors on a surface, to give the appearance of variations in height without actually modifying the geometry of the surface.

Fact

The typical way to do bumpmapping is with a height map. A height map, is a grayscale image in which the variation in colour is used to specify the amount by which points on the surface are (or appear to be) displaced.

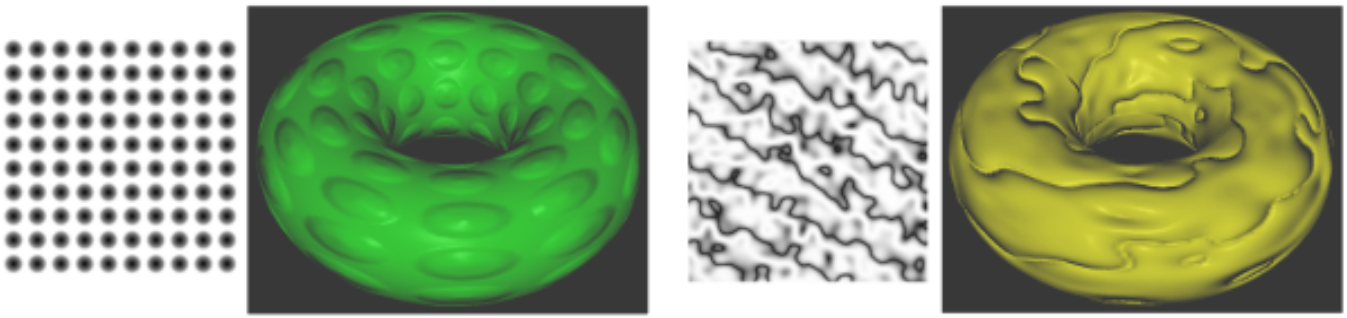


Figure: Examples of bumpmapping

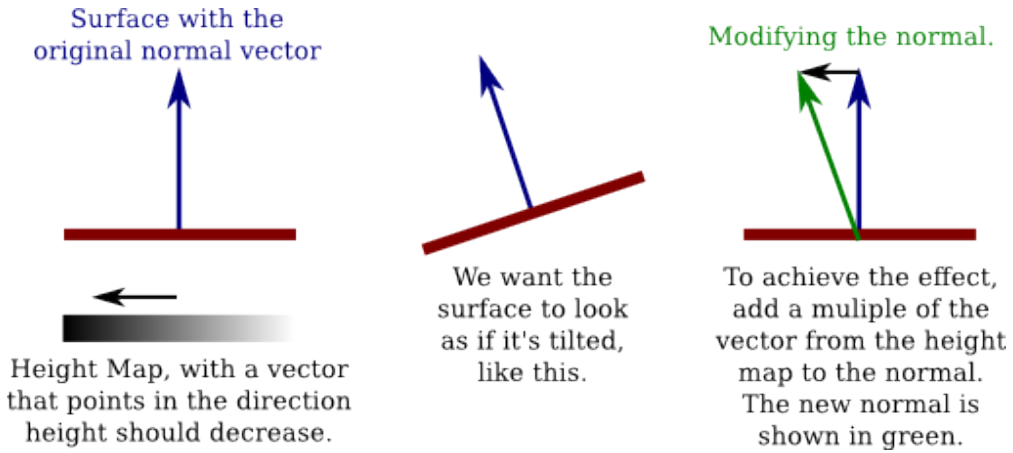


Figure: How to do bumpmapping in 1D

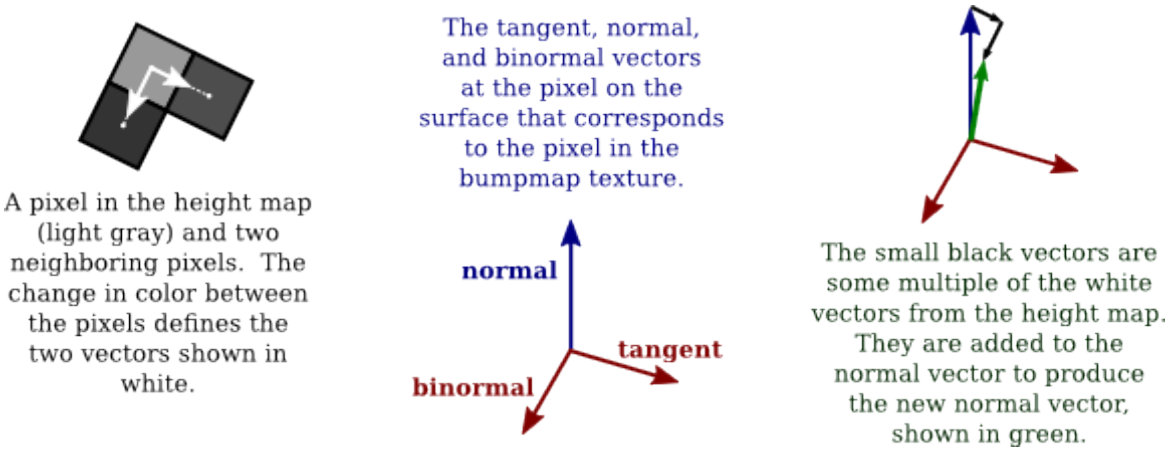


Figure: How to do bumpmapping in 3D

Demo

webgl/bumpmap.html

Example (GLSL: shader to do bumpmapping)

```
1  vec3 normal = normalize( v_normal );
2  vec3 tangent = normalize( v_tangent );
3  vec3 binormal = cross(normal,tangent);
4
5  float bm0, bmUp, bmRight;  // Samples from the bumpmap at three texels.
6  bm0 = texture2D( bumpmap, v_texCoords ).r;
7  bmUp = texture2D( bumpmap, v_texCoords + vec2(0.0, 1.0/bumpmapSize.y) ).r;
8  bmRight = texture2D( bumpmap, v_texCoords + vec2(1.0/bumpmapSize.x, 0.0) ).r;
9
10 vec3 bumpVector = (bmRight - bm0)*tangent + (bmUp - bm0)*binormal;
11 normal += bumpmapStrength*bumpVector;
12 normal = normalize( normalMatrix*normal );
```

Environment Mapping

Introduction

We can make it look as if the object is reflecting its environment by adding a skybox—a large cube surrounding the scene, with the cubemap mapped onto its interior. However, the object will only seem to be reflecting the skybox. And if there are other objects in the environment, they won't be part of the reflection.

Example (GLSL: vertex shader for a skybox)

```
1  uniform mat4 projection;
2  uniform mat4 modelview;
3  attribute vec3 coords;
4  varying vec3 v_objCoords;
5  void main() {
6      vec4 eyeCoords = modelview * vec4(coords,1.0);
7      gl_Position = projection * eyeCoords;
8      v_objCoords = coords;
9  }
```

Example (GLSL: fragment shader for a skybox)

```
precision mediump float;
varying vec3 v_objCoords;
uniform samplerCube skybox;
void main() {
      gl_FragColor = textureCube(skybox, v_objCoords);
}
```

Demo

webgl/skybox-and-env-map.html

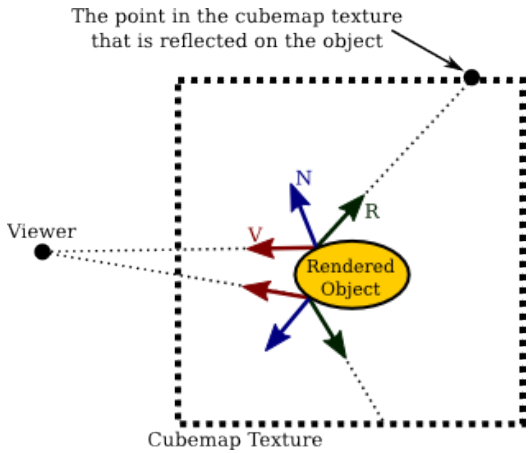


Figure: Cubemap vectors for a reflection

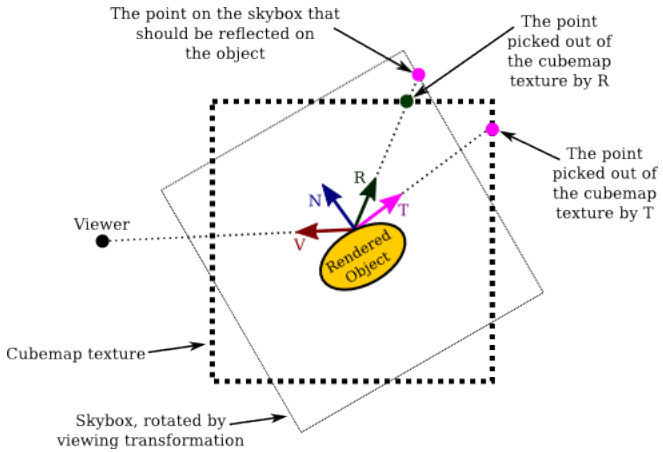


Figure: Cubemap vectors for a reflection after rotation

Example (GLSL: vertex shader for a reflection)

```
1  uniform mat4 projection;
2  uniform mat4 modelview;
3  attribute vec3 coords;
4  attribute vec3 normal;
5  varying vec3 v_eyeCoords;
6  varying vec3 v_normal;
7  void main() {
8      vec4 eyeCoords = modelview * vec4(coords,1.0);
9      gl_Position = projection * eyeCoords;
10     v_eyeCoords = eyeCoords.xyz;
11     v_normal = normalize(normal);
12 }
```

Example (GLSL: fragment shader for a reflection)

```
precision mediump float;
varying vec3 vCoords;
varying vec3 v_normal;
varying vec3 v_eyeCoords;
uniform samplerCube skybox;
uniform mat3 normalMatrix;
uniform mat3 inverseViewTransform;
void main() {
      vec3 N = normalize(normalMatrix * v_normal);
      vec3 V = -v_eyeCoords;
      vec3 R = -reflect(V,N);
      vec3 T = inverseViewTransform * R;
      gl_FragColor = textureCube(skybox, T);
}
```

Framebuffers

Introduction

The term "frame buffer" traditionally refers to the region of memory that holds the colour data for the image displayed on a computer screen. In WebGL, a framebuffer is a data structure that organizes the memory resources that are needed to render an image.

Framebuffer Operations

Introduction

We look at some WebGL settings that affect rendering into whichever framebuffer is current. Examples that we have already seen include the clear colour, which is used to fill the colour buffer when gl.clear() is called, and the enabled state of the depth test.

Definition

framebuffer In WebGL, a data structure that organizes the buffers for rendering an image, possibly including a colour buffer, a depth buffer, and a stencil buffer. A WebGL graphics context has a default framebuffer for on-screen rendering, and additional framebuffers can be created for off-screen rendering.

Example (current framebuffer can be changed by calling)

```
1  gl.bindFramebuffer( gl.FRAMEBUFFER, frameBufferObject );
```

Definition

depth mask In WebGL, a setting that controls whether depth values are written to the depth buffer during rendering. When the depth mask is set to false, the depth value is discarded and the depth buffer is unchanged.

Example (WebGL: depthmask can be turned on and off by calling)

```
1 gl.depthMask( false );
2 gl.depthMask( true );
```

Fact

One example of using the depth mask is for rendering translucent geometry. When some of the objects in a scene are translucent, then all of the opaque objects should be rendered first, followed by the translucent objects. (Suppose that you rendered a translucent object, and then rendered an opaque object that lies behind the translucent object. The depth test would cause the opaque object to be hidden by the translucent object. But "translucent" means that the opaque object should be visible through the translucent object. So it's important to render all the opaque objects first.) Note that the depth test must still be enabled while the translucent objects are being rendered, since a translucent object can be hidden by an opaque object. Also, alpha blending must be on while rendering the translucent objects.

Definition

colour mask In WebGL, a setting that determines which "channels" in the colour buffer are written during rendering. The channels are the RGBA colour components red, green, blue, and alpha. A colour mask consists of four boolean values, one for each channel. A false value prevents any change from being made to the corresponding colour component in the colour buffer.

Example (That would be done with the command)

```
1 gl.colorMask( true, false, false, true );
```

Fact

One use of the colour mask is for anaglyph stereo rendering

Fact

The default, assuming that the fragment passes the depth test, is to replace the current colour with the fragment colour. When blending is enabled, the current colour can be replaced with some combination of the current colour and the fragment colour.

Example (Enabling blending)

```
1 gl.enable( gl.BLEND );
2 gl.blendFunc( gl.SRC_ALPHA, gl.ONE_MINUS_SRC_ALPHA );
```

Fact

Note that blending applies to the alpha component as well as the RGB components of the colour, which is probably not what you want. When drawing with a translucent colour, it means that the colour that is written to the colour buffer will have an alpha component less than 1. When rendering to a canvas on a web page, this will make the canvas itself translucent, allowing the background of the canvas to show through.

Example (Enable blending without affecting the alpha component)

```
1  gl.blendFuncSeparate( gl.SRC_ALPHA, gl.ONE_MINUS_SRC_ALPHA, gl.ZERO, gl.ONE );
```

Definition

multi-pass algorithm A rendering algorithm that draws a scene several times and combines the results somehow to compute the final image. A simple example is anaglyph stereo, in which a left-eye and right-eye image of the scene are rendered separately and combined.

Render To Texture

Introduction

The previous subsection applies to any framebuffer. But we haven't yet used a non-default framebuffer. We turn to that topic now.

Definition

render-to-texture A technique in which the output of a rendering operation is written directly to a texture. In WebGL, render-to-texture can be implemented by attaching the texture as one of the buffers in a framebuffer.

Example (create a framebuffer object and make that object the current framebuffer)

```
1  framebuffer = gl.createFramebuffer(); gl.bindFramebuffer(gl.FRAMEBUFFER, framebuffer);
```

Demo

webgl/render-to-texture.html

Example

```
1  function draw() {
2      /* Draw the 2D image into a texture attached to a frambuffer. */
3      gl.bindFramebuffer(gl.FRAMEBUFFER,framebuffer);
4      gl.useProgram(prog_texture);  // shader program for the texture
5
6      gl.clearColor(1,1,1,1);
7      gl.clear(gl.COLOR_BUFFER_BIT);
8
9      gl.disable(gl.DEPTH_TEST); // framebuffer doesn't even have a depth buffer!
10     gl.viewport(0,0,512,512);  // Viewport is not set automatically!
11       .
12       .  // draw the texture image, which changes in each frame
13       .
14     gl.disable(gl.BLEND);
15
16     /*  Now draw the main scene, which is 3D, using the texture. */
17     gl.bindFramebuffer(gl.FRAMEBUFFER,null); // Draw to default framebuffer.
18     gl.useProgram(prog);  // shader program for the on-screen image
19     gl.clearColor(0,0,0,1);
20     gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);
21     gl.enable(gl.DEPTH_TEST);
22     gl.viewport(0,0,canvas.width,canvas.height);  // Reset the viewport!
23     .
24     .   // draw the scene
25     .
```

Renderbuffers

Introduction

It is often convenient to use memory from a texture object as the colour buffer for a framebuffer. However, sometimes its more appropriate to create separate memory for the buffer, not associated with any texture.

Definition

renderbuffer In WebGL, a buffer (that is, a region of memory) that can be attached to a framebuffer for use as a color buffer, depth buffer, or stencil buffer.

Example (creates a renderbuffer for use as a depth buffer)

```
1  var depthBuffer = gl.createRenderbuffer();
2  gl.bindRenderbuffer(gl.RENDERBUFFER, depthBuffer);
3  gl.renderbufferStorage(gl.RENDERBUFFER, gl.DEPTH_COMPONENT16, 512, 512);
4    .
5    .
6    .
7  //Attach a renderbufffer to be used as one of the buffers in a framebuffer
8  gl.framebufferRenderbuffer(gl.FRAMEBUFFER, gl.DEPTH_ATTACHMENT,
9    gl.RENDERBUFFER, renderbuffer);
```

Dynamic Cubemap Textures

Introduction

To render a 3D scene to a framebuffer, we need both a colour buffer and a depth buffer. An example can be found in the sample program webgl/cube-camera.html. This example uses render-to-texture for a cubemap texture. The cubemap texture is then used as an environment map on a reflective surface.

Demo

webgl/cube-camera.html


David J. Eck; Introduction to Computer Graphics; 2016; http://math.hws.edu/graphicsbook/