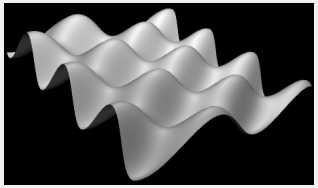
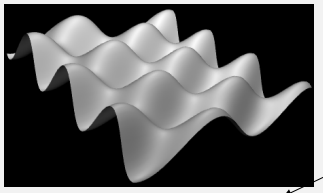
Modeling – Surfaces



- parametric surfaces
  - defined by a function f(u,v)
  - the surface consists of all points f(u,v) for u, v in the range 0-1
- steps
  - define a function with parameters u, v which returns the (x,y,z) point defined by u, v
  - create a `ParametricGeometry` using that function
    - slices and stacks determine the number of subdivisions in each direction
  - create a `Mesh` with that geometry and the desired material
    - creates a mesh approximating the surface from f(u,v) values for a grid of points $0 \le u, v \le 1$

Modeling – Parametric Surfaces



refers to the import map

```
<script type="importmap">
{
  "imports": {
    "three": "../threejs/three.module.min.js",
    "addons/": "../threejs/"
  }
}
</script>
```

```
import {ParametricGeometry} from "addons/geometries/ParametricGeometry.js";
```

```
function surfaceFunction( u, v, vector ) {
    let x,y,z;  // Coordinates for a point on the surface,
                // calculated from u,v, where u and v
                // range from 0.0 to 1.0.
    x = 20 * (u - 0.5);  // x and z range from -10 to 10
    z = 20 * (v - 0.5);
    y = 2*(Math.sin(x/2) * Math.cos(z));
    vector.set( x, y, z );
}
```

define the function

error in book – no `THREE.` as `ParametricGeometry` is not part of the three.js core

```
var surfaceGeometry = new       ParametricGeometry(surfaceFunction, 64, 64);
var surface = new THREE.Mesh( surfaceGeometry, material );
```

create a `ParametricGeometry`

create a `Mesh` from the geometry

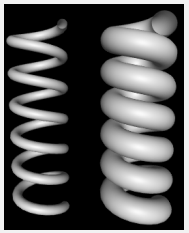
Modeling – Curves

- parametric curves
  - defined by a function f(t)
  - the curve consists of all points f(t) for t in the range 0-1
- steps
  - define a function with a parameter t which returns either a point (x,y) (2D curve) or a point (x,y,z) (3D curve)
  - create a `Curve`
  - set the curve's `getPoint` property to the function

```
var helix = new THREE.Curve();
helix.getPoint = function(t) {
   var s = (t - 0.5) * 12*Math.PI;
         // As t ranges from 0 to 1, s ranges from -6*PI to 6*PI
   return new THREE.Vector3(
        5*Math.cos(s),
        s,
        5*Math.sin(s)
   );
}
```

note that a curve is not itself a geometry, but it can be used to generate a geometry

Modeling – Surfaces From Curves

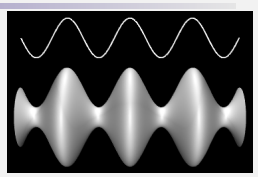


- tube geometries

```
tubeGeometry1 = new THREE.TubeGeometry( helix, 128, 2.5, 32 );
```

  - parameters
    - a `Curve` object (requires a 3D curve)
    - number of subdivisions along the length of the curve
    - radius of the circular cross-section of the tube
    - number of subdivisions around the circumference of the cross-section of the tube
  - then create a `Mesh` with that geometry and the desired material
    - creates a mesh approximating the surface from f(t) values for points $0 \le t \le 1$

Modeling – Surfaces From Curves

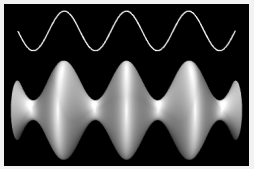


- lathing – rotate curve about a line to generate a surface of rotation
- steps
  - define a `Curve` (2D)
  - generate a set of points along the curve
  - create a `LatheGeometry` using those points

```
new THREE.LatheGeometry( points, slices )
```

    - rotates the curve around the y axis (the curve's points are in the xy plane)
    - parameters
      - `points` – array of `Vector2`
      - `slices` – number of subdivisions around the circle of rotation
  - create a `Mesh` with that geometry and the desired material
    - creates a mesh approximating the surface from f(t) values for points $0 \le t \le 1$

Modeling – Surfaces From Curves



```
var cosine = new THREE.Curve();
cosine.getPoint = function(t) {
   t = (t - 0.5) * 7*Math.PI;
   return new THREE.Vector2( 3 + 2*Math.cos(t), t );
}
```

define 2D curve

```
var points = cosine.getPoints(128);
var latheGeometry = new THREE.LatheGeometry(points, 32);
var lathe = new THREE.Mesh(latheGeometry, material);
scene.add(lathe);
```

get points
create `LatheGeometry`
create a `Mesh` from the geometry
add the object to the scene

Modeling – Surfaces From Curves

- extruding – move a filled 2D shape along a path through space
- steps
  - create a shape defining the (closed) curve

```
var path = new THREE.Shape();
path.moveTo(0,10);
path.bezierCurveTo( 0,5, 20,-10, 0,-10 );
path.bezierCurveTo( -20,-10, 0,5, 0,10 );
```

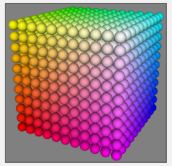
see the 2D Canvas API in section 2.6.2 for how to work with these shapes

  - create an `ExtrudeGeometry` using that shape
  - create a `Mesh` with that geometry and the desired material
    - creates a mesh approximating the surface

```
var extrudeGeom1 = new THREE.ExtrudeGeometry( path, {
   curveSegments: 32,  // How many points on each part of the path
   amount: 6,          // size in the 3rd dimension; how far to extrude.
   bevelSize: 1,
   bevelThickness: 1
});
var extrude1 = new THREE.Mesh(extrudeGeom1, material);
```

or `bevelEnabled: false` to not bevel the edges

Other Modeling



- `InstancedMesh`
  - for a number of objects with the same geometry but different transformations and (possibly) material color
  - see section 5.3.1

Cubemaps

```
let textureURLs = [  // URLs of the six faces of the cube map
        "cubemap-textures/park/posx.jpg",   // Note:  The order in which
        "cubemap-textures/park/negx.jpg",   //   the images are listed is
        "cubemap-textures/park/posy.jpg",   //   important!
        "cubemap-textures/park/negy.jpg",
        "cubemap-textures/park/posz.jpg",
        "cubemap-textures/park/negz.jpg"
    ];

/**
 *  Loads a list of textures.  If a callback function is provided, it is
 *  called once after all textures are loaded.  This is used to load a
 *  set of six textures to use on the six faces of the cube in the first
 *  scene.
 */
function loadTextures(textureURLs, callback) {
    let loaded = 0;
    function loadedOne() {
        loaded++;
        if (callback && loaded == textureURLs.length) { // all textures have been loaded
            callback();
        }
    }
    /* A funtion that will be called if the attempt to load the texture fails. */
    function textureError() {
        document.getElementById("message").innerHTML = "Error: Failed to load texture.";
    }
    let textures = new Array( textureURLs.length );
    for (let i = 0; i < textureURLs.length; i++) {
       let tex = new THREE.TextureLoader().load( textureURLs[i], loadedOne, undefined, textureError );
       textures[i] = tex;
    }
    return textures;
}
```

- to apply a cubemap to the outside of a box, texture each face separately

```
/* Load the six image textures and create the six face materials. */
let textures = loadTextures(textureURLs, render);
let materials = [];
for (let i = 0; i < 6; i++) {
     materials.push( new THREE.MeshPhongMaterial( {
           color: "white",
           map: textures[i]
     } ) );
}

/* Create a cube with the six textures on the six faces of the cube. */
cube = new THREE.Mesh( new THREE.BoxGeometry(20,20,20), materials );
scene.add(cube);
```


Reflection via Environment Mapping

- steps
  - load a cubemap texture

```
var textureURLs = [  // URLs of the six faces of the cube map
        "cubemap-textures/park/posx.jpg",   // Note:  The order in which
        "cubemap-textures/park/negx.jpg",   //   the images are listed is
        "cubemap-textures/park/posy.jpg",   //   important!
        "cubemap-textures/park/negy.jpg",
        "cubemap-textures/park/posz.jpg",
        "cubemap-textures/park/negz.jpg"
    ];
```

```
var texture = new THREE.CubeTextureLoader().load( textureURLs );
```

  - for the skybox, set the scene's background to the cubemap texture

```
scene.background = texture;
```

  - for the reflective object, create a `MeshBasicMaterial`
    - set the material's `envMap` property to the cubemap texture object

```
var material = new THREE.MeshBasicMaterial( {
        color: "white",  // Color will be multiplied by the environment map.
        envMap: texture  // Cubemap texture to be used as an environment map.
    } );
```


Reflection via Environment Mapping

- dynamic cubemaps can be generated using `THREE.CubeCamera`
  - `https://threejs.org/docs/#api/en/cameras/CubeCamera`

Refraction via Environment Mapping

- steps
  - load a cubemap texture
    - specify that the cubemap is to be used for refraction

```
var textureURLs = [  // URLs of the six faces of the cube map
        "cubemap-textures/park/posx.jpg",   // Note:  The order in which
        "cubemap-textures/park/negx.jpg",   //   the images are listed is
        "cubemap-textures/park/posy.jpg",   //   important!
        "cubemap-textures/park/negy.jpg",
        "cubemap-textures/park/posz.jpg",
        "cubemap-textures/park/negz.jpg"
    ];
```

```
texture = new THREE.CubeTextureLoader().load( textureURLs );
texture.mapping = THREE.CubeRefractionMapping;
```


Refraction via Environment Mapping

- steps, continued
  - for the skybox, set the scene's background

```
scene.background = texture;
```

  - for the refractive object, create a `MeshBasicMaterial`
    - set the material's `envMap` property to the cubemap texture object
    - set other material properties
      - `refractionRatio` – ratio of index of refraction of air to material (1 = no bending of light, smaller values = greater bending – default is 0.98)
      - `reflectivity` – proportion of light transmitted (< 1 to make object look cloudy)

```
var material = new THREE.MeshBasicMaterial( {
        color: "white",
        envMap: texture,
        refractionRatio: 0.6
    } );
```


Shadow Mapping in three.js

- steps
  - enable shadow computations in the renderer

```
renderer.shadowMap.enabled = true;
```

    - expensive, so disabled by default
  - enable casting of shadows for each light

```
light.castShadow = true;
```

    - only applicable to `DirectionalLights` and `SpotLights`
  - enable casting and receiving of shadows for each object

```
object.castShadow = true;
object.receiveShadow = true;
```

    - "receiving" means shadows will be visible on that object
  - configure the view volumes of the shadow cameras →
    - directional lights use `OrthographicCamera`
    - spotlights use `PerspectiveCamera`
  - (optionally) increase the size of the shadow map

```
light.shadow.mapSize.width = 1024;
light.shadow.mapSize.height = 1024;
```

    - larger map increases accuracy of the shadows

Shadow Mapping in three.js

- configuring the shadow camera view volume
  - `light.shadow.camera` is the shadow camera for `light`
  - for directional lights (orthographic camera), properties are `left`, `right`, `bottom`, `top`, `near`, `far`
  - for spotlights (perspective camera), properties are `fov`, `near`, `far`
    - `fov` is the field of view angle, in degrees (should match spotlight's cutoff angle)
  - values are in view coordinates
  - view volume should be big enough to include all of the objects that cast shadows, but not too big
    - too big impacts the accuracy of the shadow map

Other three.js Topics

- mouse interaction – covered section 5.3.2
  - `OrbitControls` and `TrackballControls`
  - clicking on objects within the scene
- keyboard input (key presses)
  - not directly covered in the textbook, but present in many of the examples in chapter 5
- custom shaders

Custom Shaders

- steps
  - write GLSL vertex and fragment shaders
    - three.js provides a number of built-in uniforms and attributes
  - create a `ShaderMaterial` which specifies the shader programs and the values for custom (not built in) parameters

Built-in Uniforms and Attributes

- vertex shader
  - uniforms
    - `modelMatrix`
    - `modelViewMatrix`
    - `projectionMatrix`
    - `viewMatrix`
    - `normalMatrix`
    - `cameraPosition` (WC)
  - attributes
    - `position`
    - `normal`
    - `uv`
- fragment shader
  - uniforms
    - `viewMatrix`
    - `cameraPosition`

reference:
https://threejs.org/docs/#api/en/renderers/webgl/WebGLProgram

ShaderMaterial

- key properties
  - `vertexShader`
  - `fragmentShader`
  - uniforms
    - for custom uniforms
    - (custom attributes are set as part of the geometry)

reference:
https://threejs.org/docs/#api/en/materials/ShaderMaterial
https://threejs.org/docs/#api/en/materials/Material

Example

```
<script type="x-shader/x-vertex" id="vshader_silhouette">
    void main() {
        vec4 coords = vec4(position,1.0)+vec4(.03*normal,0);
        vec4 ec = modelViewMatrix * coords;
        gl_Position = projectionMatrix * ec;
    }

</script>

<script type="x-shader/x-fragment" id="fshader_silhouette">
    void main () {
        gl_FragColor = vec4(0.0,0.0,0.0,1.0);
    }
</script>
```

black silhouette

```
silhouetteMaterial = new THREE.ShaderMaterial( {
    vertexShader: getTextContent("vshader_silhouette"),
    fragmentShader: getTextContent("fshader_silhouette"),
    side: THREE.BackSide
} );
```

```
<script type="x-shader/x-vertex" id="vshader_silhouette">
    void main() {
        vec4 coords = vec4(position,1.0)+vec4(.03*normal,0);
        vec4 ec = modelViewMatrix * coords;
        gl_Position = projectionMatrix * ec;
    }

</script>

<script type="x-shader/x-fragment" id="fshader_silhouette">
    uniform vec3 color;

    void main () {
        gl_FragColor = vec4(color,1.0);
    }
</script>
```

silhouette with a specified color

```
silhouetteMaterial = new THREE.ShaderMaterial( {
    uniforms : {
        color: { value: new THREE.Color( 0x000000 ) }
    },
    vertexShader: getTextContent("vshader_silhouette"),
    fragmentShader: getTextContent("fshader_silhouette"),
    side: THREE.BackSide
} );
```

`getTextContent` is a local utility function to get the text from an HTML element

Working With Lights in Shaders

- `UniformsLib` defines a bunch of uniforms that can get passed to your shaders

```
lights: {

        ambientLightColor: { value: [] },

        lightProbe: { value: [] },

        directionalLights: { value: [], properties: {
                direction: {},
                color: {},

                shadow: {},
                shadowBias: {},
                shadowRadius: {},
                shadowMapSize: {}
        } },
```

(a small section)

reference:
https://threejs.org/docs/#api/en/renderers/shaders/UniformsLib

Working With Lights in Shaders

```
<script type="x-shader/x-vertex" id="vshader">
    varying vec3 v_coords;   // EC
    varying vec3 v_normal;   // EC

    void main() {
        vec4 coords = vec4(position,1.0);
        vec4 ec = modelViewMatrix * coords;
        gl_Position = projectionMatrix * ec;

        v_coords = ec.xyz;
        v_normal = normalize(normalMatrix*normal);
    }

</script>
```

need EC point and normal for lighting calculations

Working With Lights

```
struct DirectionalLight {
    vec3 color;
    vec3 direction;
};

uniform vec3 diffuseColor;
uniform vec3 specularColor;
uniform float specularExponent;

uniform vec3 ambientLightColor;
uniform DirectionalLight directionalLights[ NUM_DIR_LIGHTS ];

varying vec3 v_coords;   // EC
varying vec3 v_normal;   // EC
```

circled items match the `UniformLibs` definition

`NUM_DIR_LIGHTS` is also automatically defined

```
lights: {

        ambientLightColor: { value: [] },

        lightProbe: { value: [] },

        directionalLights: { value: [], properties: {
                direction: {},
                color: {},

                shadow: {},
                shadowBias: {},
                shadowRadius: {},
                shadowMapSize: {}
        } },
```

```
<script type="x-shader/x-fragment" id="fshader">
    struct DirectionalLight {
        vec3 color;
        vec3 direction;
    };

    uniform vec3 diffuseColor;
    uniform vec3 specularColor;
    uniform float specularExponent;

    uniform vec3 ambientLightColor;
    uniform DirectionalLight directionalLights[ NUM_DIR_LIGHTS ];

    varying vec3 v_coords;   // EC
    varying vec3 v_normal;   // EC

    vec3 lightingEquation( DirectionalLight light,
                           vec3 eyeCoords, // Eye coordinates for the point.
                           vec3 N, // Normal vector to the surface.
                           vec3 V  // Direction to viewer.
                         ) {

        vec3 reflection = vec3(0);

        vec3 L, R; // Light direction and reflected light direction.
        L = normalize( light.direction );

        if (dot(L,N) <= 0.0) { // light does not illuminate the surface
            return reflection;
        }

        reflection += dot(L,N) * light.color * diffuseColor;

        R = -reflect(L,N);
        if (dot(R,V) > 0.0) { // ray is reflected toward the the viewer
            float factor = pow(dot(R,V),specularExponent);
            reflection += factor * specularColor * light.color;
        }

        return reflection;
    }

    void main () {
        vec3 viewDirection = normalize(-v_coords);

        vec3 color = vec3(0.0);
        color += ambientLightColor*diffuseColor;

        for (int i = 0; i < NUM_DIR_LIGHTS; i++) {
            color += lightingEquation( directionalLights[i],
                                       v_coords, v_normal,
                                       viewDirection );
        }

        gl_FragColor = vec4(color,1.0);
    }
</script>
```


Working With Lights in Shaders

```
shaderMaterial = new THREE.ShaderMaterial( {
    uniforms : THREE.UniformsUtils.merge([
        THREE.UniformsLib['lights'],
        {
            diffuseColor : { value: new THREE.Color(0xffff00) },
            specularColor : { value: new THREE.Color(0xffffff) },
            specularExponent : { value : 50.0 }
        }]),
    vertexShader: getTextContent("vshader"),
    fragmentShader: getTextContent("fshader"),
    side: THREE.FrontSide,
    lights: true
} );
```

merge combines multiple sets of properties

provides the `lights` section from `UniformsLib`

must enable lighting in order for lighting info to be passed to shader

```
// create some lights and add them to the scene.
// dim light shining from above
scene.add( new THREE.DirectionalLight( 0xffffff, 0.4 ) );
// a light to shine in the direction the camera faces
var viewpointLight = new THREE.PointLight( 0xffffff, 0.8 );
viewpointLight.position.set(0,0,10);  // shines down the z-axis
scene.add(viewpointLight);
shaderMaterial.needsUpdate = true;
```

set to `true` if adding or changing lights after shader is compiled