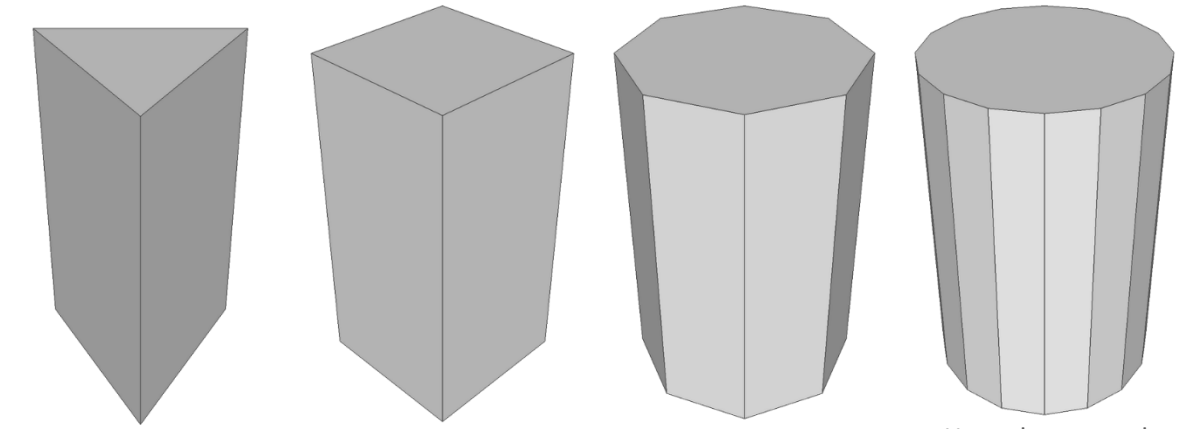
OpenGL Cylinder, Prism & Pipe

This page describes how to generate a cylinder geometry using C++ and how to draw it in OpenGL.

Cylinder & Prism

The definition of a cylinder is *a 3D closed surface that has 2 parallel circular bases at the ends and connected by a curved surface (side)*. Similarly, a prism is a 3D closed surface that has 2 parallel *polygonal* bases connected by *flat* surfaces.

Since we cannot draw a perfect circular base and curved side of the cylinder, we only sample a limited amount of points by dividing the base by sectors (slices). Therefore, it is technically constructing a prism by connecting these sampled points together. As the number of samples increases, the geometry is closer to a cylinder.



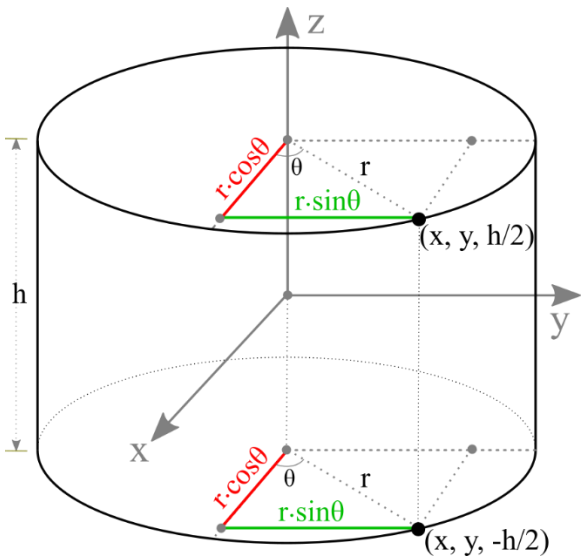
Triangular Prism (3) | Rectangluar Prism (4) | Octagonal Prism (8) | Hexadecagonal Prism (16)

Suppose a cylinder is centered at the origin and its radius is *r* and the height is *h*. An arbitrary point (*x*, *y*, *z*) on the cylinder can be computed from the equation of circle with the corresponding sector angle θ.

$$\begin{aligned} x &= r \cdot \cos\theta \\ y &= r \cdot \sin\theta \\ z &= \frac{h}{2}, \text{ or } -\frac{h}{2} \end{aligned}$$

The range of sector angles is from 0 to 360 degrees. The sector angle for each step can be calculated by the following;

$$\theta = 2\pi \cdot \frac{\text{sectorStep}}{\text{sectorCount}}$$



A vertex coordinate on a cylinder

The following C++ code generates all vertices of the cylinder with the given base radius, height and the number of sectors (slices). It also creates other vertex attributes; surface normals and texture coordinates.

In order to reduce multiple computations of sine and cosine, we compute the vertices of a unit circle on XY plane only once, and then re-use these points multiple times by scaling with the base radius. These are also used for the normal vectors of the side faces of the cylinder.

```
// generate a unit circle on XY-plane
std::vector<float> Cylinder::getUnitCircleVertices()
{
    const float PI = 3.1415926f;
    float sectorStep = 2 * PI / sectorCount;
    float sectorAngle;  // radian

    std::vector<float> unitCircleVertices;
    for(int i = 0; i <= sectorCount; ++i)
    {
        sectorAngle = i * sectorStep;
        unitCircleVertices.push_back(cos(sectorAngle)); // x
        unitCircleVertices.push_back(sin(sectorAngle)); // y
        unitCircleVertices.push_back(0);                // z
    }
    return unitCircleVertices;
}
...

// generate vertices for a cylinder
void Cylinder::buildVerticesSmooth()
{
    // clear memory of prev arrays
    std::vector<float>().swap(vertices);
    std::vector<float>().swap(normals);
    std::vector<float>().swap(texCoords);

    // get unit circle vectors on XY-plane
    std::vector<float> unitVertices = getUnitCircleVertices();
```

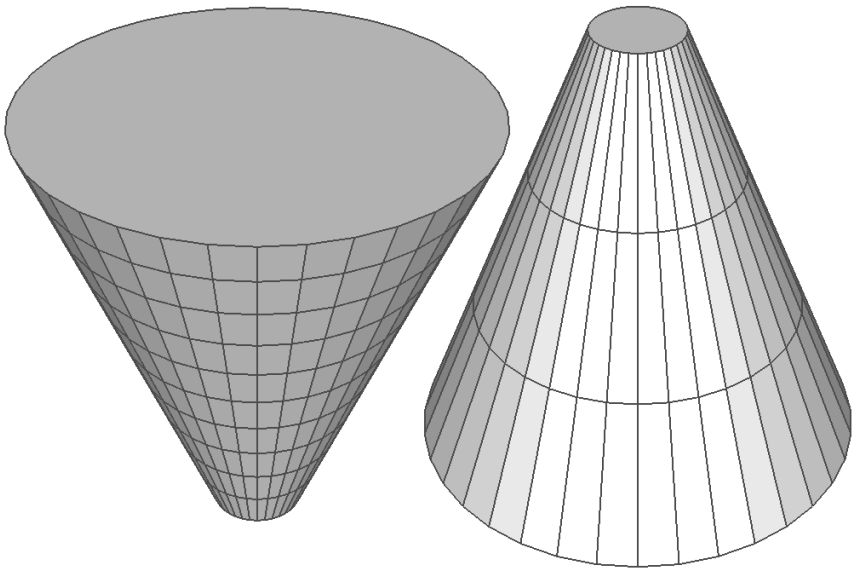
```
    // put side vertices to arrays
    for(int i = 0; i < 2; ++i)
    {
        float h = -height / 2.0f + i * height;   // z value; -h/2 to h/2
        float t = 1.0f - i;                      // vertical tex coord; 1 to 0

        for(int j = 0, k = 0; j <= sectorCount; ++j, k += 3)
        {
            float ux = unitVertices[k];
            float uy = unitVertices[k+1];
            float uz = unitVertices[k+2];
            // position vector
            vertices.push_back(ux * radius);             // vx
            vertices.push_back(uy * radius);             // vy
            vertices.push_back(h);                       // vz
            // normal vector
            normals.push_back(ux);                       // nx
            normals.push_back(uy);                       // ny
            normals.push_back(uz);                       // nz
            // texture coordinate
            texCoords.push_back((float)j / sectorCount); // s
            texCoords.push_back(t);                      // t
        }
    }

    // the starting index for the base/top surface
    //NOTE: it is used for generating indices later
    int baseCenterIndex = (int)vertices.size() / 3;
    int topCenterIndex = baseCenterIndex + sectorCount + 1; // include
center vertex

    // put base and top vertices to arrays
    for(int i = 0; i < 2; ++i)
    {
        float h = -height / 2.0f + i * height;  // z value; -h/2 to h/2
        float nz = -1 + i * 2;                  // z value of normal; -1 to 1

        // center point
        vertices.push_back(0);vertices.push_back(0);vertices.push_back(h);
        normals.push_back(0); normals.push_back(0); normals.push_back(nz);
        texCoords.push_back(0.5f); texCoords.push_back(0.5f);

        for(int j = 0, k = 0; j < sectorCount; ++j, k += 3)
        {
            float ux = unitVertices[k];
            float uy = unitVertices[k+1];
            // position vector
            vertices.push_back(ux * radius);             // vx
            vertices.push_back(uy * radius);             // vy
            vertices.push_back(h);                       // vz
            // normal vector
            normals.push_back(0);                        // nx
            normals.push_back(0);                        // ny
            normals.push_back(nz);                       // nz
            // texture coordinate
            texCoords.push_back(-ux * 0.5f + 0.5f);      // s
            texCoords.push_back(-uy * 0.5f + 0.5f);      // t
        }
    }
}
```

This C++ class provides **buildVerticesSmooth()** and **buildVerticesFlat()** functions depending on surface smoothness. The constructor also takes additional parameters to construct various shapes of a cylinder, similar to OpenGL **gluCylinder()** function.

The parameters of the cylinder class are;

1. the base radius (*float*)
2. the top radius (*float*)
3. the height (*float*)
4. the number of sectors (*int*)
5. the number of stacks (*int*)
6. smoothness (*bool*)

For instance, if the base radius is 0, it becomes a cone shape. For more details, please refer to Cylinder.cpp class.



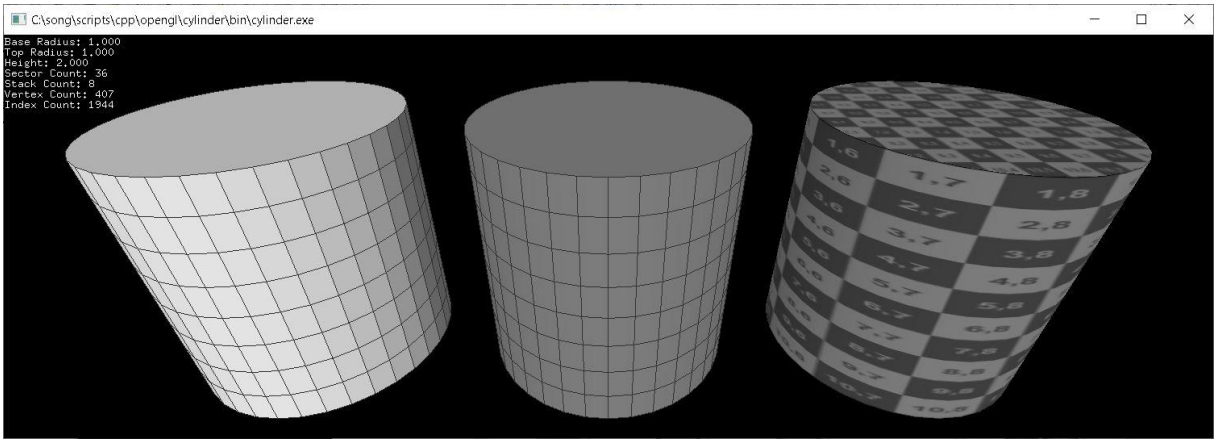
Cylinders with different base/top radii and stack count

In order to draw the surface of a cylinder in OpenGL, you must triangulate adjacent vertices counterclockwise to form polygons. Each sector on the side surface requires 2 triangles. The total number of triangles for the side is $2 \times sectorCount$. And the number of triangles for the base or top surface is the same as the number of sectors. (*You may use **GL_TRIANGLE_FAN** for the base/top instead of **GL_TRIANGLES**.*)

The code snippet to generate all the triangles of a cylinder may look like:

```
// generate CCW index list of cylinder triangles
std::vector<int> indices;
int k1 = 0;                           // 1st vertex index at base
int k2 = sectorCount + 1;             // 1st vertex index at top

// indices for the side surface
for(int i = 0; i < sectorCount; ++i, ++k1, ++k2)
{
    // 2 triangles per sector
    // k1 => k1+1 => k2
    indices.push_back(k1);
    indices.push_back(k1 + 1);
    indices.push_back(k2);

    // k2 => k1+1 => k2+1
    indices.push_back(k2);
    indices.push_back(k1 + 1);
    indices.push_back(k2 + 1);
}

// indices for the base surface
//NOTE: baseCenterIndex and topCenterIndices are pre-computed during vertex
//      generation. Please see the previous code snippet
for(int i = 0, k = baseCenterIndex + 1; i < sectorCount; ++i, ++k)
{
    if(i < sectorCount - 1)
    {
        indices.push_back(baseCenterIndex);
        indices.push_back(k + 1);
        indices.push_back(k);
    }
    else // last triangle
    {
        indices.push_back(baseCenterIndex);
        indices.push_back(baseCenterIndex + 1);
        indices.push_back(k);
    }
}

// indices for the top surface
for(int i = 0, k = topCenterIndex + 1; i < sectorCount; ++i, ++k)
{
    if(i < sectorCount - 1)
    {
        indices.push_back(topCenterIndex);
        indices.push_back(k);
        indices.push_back(k + 1);
    }
    else // last triangle
    {
        indices.push_back(topCenterIndex);
        indices.push_back(k);
        indices.push_back(topCenterIndex + 1);
    }
}
```

Example: Drawing Cylinder



Download: cylinder.zip, cylinderShader.zip *(Updated: 2020-03-14)*

This example constructs cylinders with 36 sectors and 8 stacks, but with different shadings; flat, smooth or textured. With the default constructor (without arguments), it generates a cylinder with base/top radius = 1, height = 2, sectors = 36 and stacks = 1. You could also pass the custom parameters to the constructor, similar to gluCylinder(). Press the space key to change the number of sectors and stacks of the cylinders.

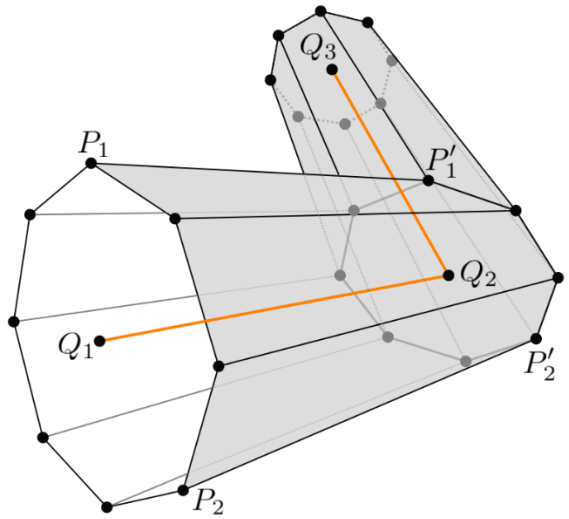
Cylinder.cpp class provides pre-defined drawing functions using OpenGL VertexArray; **draw()**, **drawWithLines()**, **drawLines()**, **drawSide()**, **drawBase()** and **drawTop()**.

```
// create a cylinder with base radius=1, top radius=2, height=3,
// height=4, sectors=5, stacks=6, smooth=true
Cylinder cylinder(1, 2, 3, 4, 5, 6, true);

// can change parameters later
cylinder.setBaseRadius(1.5f);
cylinder.setTopRadius(2.5f);
cylinder.setHeight(3.5f);
cylinder.setSectorCount(36);
cylinder.setStackCount(8);
cylinder.setSmooth(false);
...

// draw cylinder using vertexarray
cylinder.draw();            // draw surface only
cylinder.drawWithLines();   // draw surface and lines
cylinder.drawSide();        // draw side only
cylinder.drawTop();         // draw top only
cylinder.drawBase();        // draw botton only
```

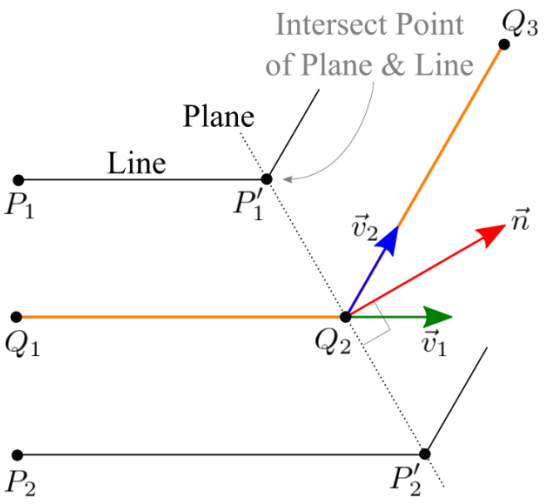
This C++ class also provides **getVertices()**, **getIndices()**, **getInterleavedVertices()**, etc. in order to access the vertex data in GLSL. The following code draws a cylinder with interleaved vertex data using VBO and GLSL. Or, download cylinderShader.zip for more details.

```
// create a cylinder with default params;
// radii=1, height=1, sectors=36, stacks=1, smooth=true
Cylinder cylinder;

// copy interleaved vertex data (V/N/T) to VBO
GLuint vboId;
glGenBuffers(1, &vboId);
glBindBuffer(GL_ARRAY_BUFFER, vboId);             // for vertex data
glBufferData(GL_ARRAY_BUFFER,                     // target
             cylinder.getInterleavedVertexSize(), // data size, # of bytes
             cylinder.getInterleavedVertices(),   // ptr to vertex data
             GL_STATIC_DRAW);                     // usage

// copy index data to VBO
GLuint iboId;
glGenBuffers(1, &iboId);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, iboId);     // for index data
glBufferData(GL_ELEMENT_ARRAY_BUFFER,             // target
             cylinder.getIndexSize(),             // data size, # of bytes
             cylinder.getIndices(),               // ptr to index data
             GL_STATIC_DRAW);                     // usage
...


// bind VBOs
glBindBuffer(GL_ARRAY_BUFFER, vboId);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, iboId);

// activate attrib arrays
glEnableVertexAttribArray(attribVertex);
glEnableVertexAttribArray(attribNormal);
glEnableVertexAttribArray(attribTexCoord);

// set attrib arrays with stride and offset
int stride = cylinder.getInterleavedStride();   // should be 32 bytes
glVertexAttribPointer(attribVertex,   3, GL_FLOAT, false, stride,
(void*)0);
glVertexAttribPointer(attribNormal,   3, GL_FLOAT, false, stride,
(void*)(sizeof(float)*3));
glVertexAttribPointer(attribTexCoord, 2, GL_FLOAT, false, stride,
(void*)(sizeof(float)*6));

// draw a cylinder with VBO
glDrawElements(GL_TRIANGLES,                    // primitive type
               cylinder.getIndexCount(),        // # of indices
               GL_UNSIGNED_INT,                 // data type
               (void*)0);                       // offset to indices

// deactivate attrib arrays
glDisableVertexAttribArray(attribVertex);
glDisableVertexAttribArray(attribNormal);
glDisableVertexAttribArray(attribTexCoord);

// unbind VBOs
glBindBuffer(GL_ARRAY_BUFFER, 0);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, 0);
```

Pipe (Tube)



Extruding a pipe along a path Q_1-Q_2-Q_3



Cross-section view of extruding a pipe
P'_1 is the intersection point on a plane and a line passing P_1

A common application is drawing a pipe, which is extruding a contour along a given path. Suppose the path is Q_1-Q_2-Q_3, and a point of the contour is P_1. To find the next point, P'_1, we need to project P_1 onto the plane at the Q_2 with the normal, $\vec{n}$, where 2 path lines Q_1-Q_2 and Q_2-Q_3 are met.

Projecting P_1 to P'_1 is actually finding the intersection of the point where a line and a plane are met. See the cross-section view (the right-side image above). The line equation is $P_1 + t\vec{v}_1$, which is passing P_1 with the direction vector $\vec{v}_1 = Q_2 - Q_1$.

And, the plane equation can be computed by the normal vector $\vec{n}$and the point on the plane Q_2 (x_2, y_2, z_2);

$$\vec{n} \cdot (x - x_2,\ y - y_2,\ z - z_2) = 0$$

And, the normal vector is computed by adding $\vec{v}_1$and $\vec{v}_2$together;

$$\vec{v}_1 = Q_2 - Q_1$$
$$\vec{v}_2 = Q_3 - Q_2$$
$$\vec{n} = \vec{v}_1 + \vec{v}_2$$

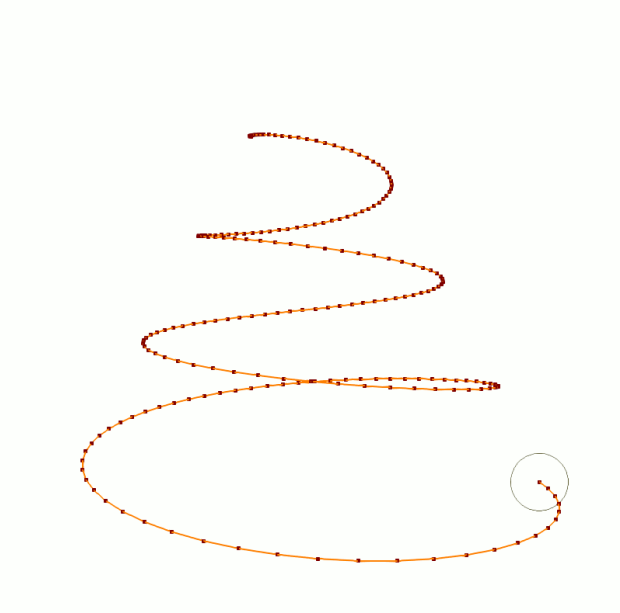
Finding the intersection point P'_1 is solving the linear system of the plane and line:

$$\begin{cases} \text{Plane:} & \vec{n} \cdot (x - x_2,\ y - y_2,\ z - z_2) = 0 \\ \text{Line:} & P_1 + t\vec{v} \end{cases}$$

You can find the solution of the linear system here. Or, see the detail C++ implementation in **Pipe::projectContour()** of Pipe.cpp and Plane.cpp.

```
std::vector<Vector3> Pipe::projectContour(int fromIndex, int toIndex)
{
    Vector3 v1, v2, normal, point;
    Line line;

    // find direction vectors; v1 and v2
    v1 = path[toIndex] - path[fromIndex];
    if(toIndex == (int)path.size()-1)
        v2 = v1;
    else
        v2 = path[toIndex + 1] - path[toIndex];

    // normal vector of plane at toIndex
    normal = v1 + v2;

    // define plane equation at toIndex with normal and point
    Plane plane(normal, path[toIndex]);

    // project each vertex of contour to the plane
    std::vector<Vector3>& fromContour = contours[fromIndex];
    std::vector<Vector3> toContour;
    int count = (int)fromContour.size();
    for(int i = 0; i < count; ++i)
    {
        line.set(v1, fromContour[i]);   // define line with direction and
point
        point = plane.intersect(line);  // find the intersection point
        toContour.push_back(point);
    }

    // return the projected vertices of contour at toIndex
    return toContour;
}
```

Example: Extruding Pipe along Path



This example is drawing a pipe extruding a circular contour following a spiral path. Press D key to switch the rendering modes.
Download: pipe.zip, pipeShader.zip

A pipe can be constructed with a pre-defined path (a sequence of points), or you can add the next point of the path if needed using **Pipe::addPathPoint()**.

The shape of the contour is not necessarily a circle. You can provide an arbitrary shape of a contour.

To draw the surface of the pipe, use **Pipe::getContour()** and **Pipe::getNormal()** to get the vertices and normals at a given path point. Then, draw triangles between 2 contours using **GL_TRIANGLE_STRIP**.

Example: WebGL Cylinder (Interactive Demo)

It is a JavaScript implementation of Cylinder class, Cylinder.js, and rendering it with WebGL. Drag the sliders to change the parameters of the cylinder. The fullscreen version is available here.

The following JavaScript code is to create and to render a cylinder object.

```
// create a cylinder with 6 params:
// baseR, topR, height, sectors, stacks, smooth
let cylinder = new Cylinder(1, 2, 3, 4, 5, false);
...

// change params of cylinder later
cylinder.setBaseRadius(1);
cylinder.setTopRadius(2);
cylinder.setHeight(3);
cylinder.setSectorCount(4);
cylinder.setStackCount(5);
cylinder.setSmooth(true);
...

// draw a cylinder with interleaved mode
gl.bindBuffer(gl.ARRAY_BUFFER, cylinder.vboVertex);
gl.vertexAttribPointer(gl.program.attribPosition, 3, gl.FLOAT, false,
cylinder.stride, 0);
gl.vertexAttribPointer(gl.program.attribNormal, 3, gl.FLOAT, false,
cylinder.stride, 12);
gl.vertexAttribPointer(gl.program.attribTexCoord0, 2, gl.FLOAT, false,
cylinder.stride, 24);

gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, cylinder.vboIndex);
gl.drawElements(gl.TRIANGLES, cylinder.getIndexCount(), gl.UNSIGNED_SHORT,
0);
...
```