# WebGL Obj 로딩

Wavefront .obj파일은 온라인에서 찾아볼 수 있는 가장 흔한 3D 파일 포맷입니다. 파싱하기도 그리 어렵지 않기 때문에 한번 파싱을 해 봅시다. 아마 일반적인 3D 파일 포맷을 파싱하는 방법을 배우는 좋은 예제가 될 것 같습니다.

**유의사항:** 이 글의 .OBJ 파서는 모든 .OBJ 파일에 대해 완벽하게 동작하지는 않습니다. 그보다는 우리가 하려는 작업을 간단하게 수행해 보는 연습 목적입니다. 따라서 뭔가 문제에 겪었거나 그에 대한 해결책을 찾으셨다면 제일 아래쪽의 코멘트를 사용해 알려주시면 이 코드를 사용하려는 다른 사람들에게 도움이 될 겁니다.

제가 찾은 .OBJ 포맷에 대한 가장 좋은 문서는 이것입니다. 이 페이지는 원본 문서를 포함한 다양한 문서에 대항 링크를 제공합니다.

간단한 예제를 살펴봅시다. 아래는 블렌더의 기본 화면에서 추출한 cube.obj 파일입니다.

```
# Blender v2.80 (sub 75) 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
v -1.000000 -1.000000 -1.000000
v -1.000000 1.000000 1.000000
v -1.000000 -1.000000 1.000000
vt 0.375000 0.000000
vt 0.625000 0.000000
vt 0.625000 0.250000
vt 0.375000 0.250000
vt 0.375000 0.250000
vt 0.625000 0.250000
vt 0.625000 0.500000
vt 0.375000 0.500000
vt 0.625000 0.750000
vt 0.375000 0.750000
vt 0.625000 0.750000
vt 0.625000 1.000000
vt 0.375000 1.000000
vt 0.125000 0.500000
vt 0.375000 0.500000
vt 0.375000 0.750000
vt 0.125000 0.750000
vt 0.625000 0.500000
vt 0.875000 0.500000
vt 0.875000 0.750000
```

```
vn 0.0000 1.0000 0.0000
vn 0.0000 0.0000 1.0000
vn -1.0000 0.0000 0.0000
vn 0.0000 -1.0000 0.0000
vn 1.0000 0.0000 0.0000
vn 0.0000 0.0000 -1.0000
usemtl Material
s off
f 1/1/1 5/2/1 7/3/1 3/4/1
f 4/5/2 3/6/2 7/7/2 8/8/2
f 8/8/3 7/7/3 5/9/3 6/10/3
f 6/10/4 2/11/4 4/12/4 8/13/4
f 2/14/5 1/15/5 3/16/5 4/17/5
f 6/18/6 5/19/6 1/20/6 2/11/6
```

문서를 살펴보지 않아도 `v`로 시작하는 라인은 정점의 위치를, `vt`로 시작하는 라인은 텍스처 좌표를, `vn`으로 시작하는 라인은 법선을 나타낸다는 것을 눈치챌 수 있을겁니다. 남은 것은 나머지 라인들이 무슨 의미이냐 입니다.

.OBJ파일은 텍스트 파일이므로 우선 텍스트 파일을 로딩해야 합니다. 다행히 2020년 현재는 async/await를 사용하면 아주 손쉽게 구현할 수 있습니다.

```
async function main() {
  ...

  const response = await fetch('resources/models/cube/cube.obj');
  const text = await response.text();
```

다음으로 한 라인씩 파싱을 수행하면 되는데, 매 라인은 아래와 같은 형식으로 되어 있습니다.

```
키워드 데이터 데이터 데이터 데이터 ...
```

각 라인의 첫 번째로 키워드가 나타나며 데이터들이 공백을 활용하여 구분되고 있습니다. `#`으로 시작하는 라인은 주석입니다.

이제, 각 라인을 파싱하는 코드를 작성해 봅시다. 빈 라인과 주석 라인은 스킵하고 키워드를 기준으로 함수를 호출합니다.

```
function parseOBJ(text) {

  const keywords = {
  };

  const keywordRE = /(\w*)(?: )*(.*)/;
  const lines = text.split('\n');
  for (let lineNo = 0; lineNo < lines.length; ++lineNo) {
    const line = lines[lineNo].trim();
    if (line === '' || line.startsWith('#')) {
      continue;
    }
    const m = keywordRE.exec(line);
```

```
    if (!m) {
      continue;
    }
    const [, keyword, unparsedArgs] = m;
    const parts = line.split(/\s+/).slice(1);
    const handler = keywords[keyword];
    if (!handler) {
      console.warn('unhandled keyword:', keyword, 'at line', lineNo + 1);
      continue;
    }
    handler(parts, unparsedArgs);
  }
}
```

주의해야 할 사항은 우리가 각 라인의 맨 앞과 맨 뒤의 공백을 제거했다는 것입니다. 이 과정이 꼭 필요한지는 모르겠지만 해서 나쁠 것이 없을 것 같습니다. 각 라인은 `/\s+/` 를 사용해 분할됩니다. 이 역시 꼭 필요한지는 모르겠습니다. 데이터마다 하나 이상의 공백이 있는 경우도 있을까요? 탭이 있을수도 있을까요? 잘 모르지만 텍스트 포맷이기 때문에 다양성을 고려하는 편이 더 안전할 것 같습니다.

한편 첫 번째 부분을 키워드로 추출하여 해당 키워드에 관련된 함수를 찾고, 나머지 데이터를 그 함수에 넘기면서 함수를 호출합니다. 이제는 함수 내부의 코드를 작성하기만 하면 됩니다.

`v`, `vt`, `vn` 이 어떤 데이터인지 짐작해 보았습니다. 문서에 보면 `f` 는 "면" 혹은 폴리곤이라는 뜻이고 각 데이터는 위치, 텍스처 좌표, 법선의 인덱스를 뜻한다고 합니다.

인덱스는 양수라면 1부터 시작하도록 되어있고, 음수라면 지금까지 파싱한 정점 숫자에 상대적인 수입니다. 인덱스의 순서는 위치/텍스처 좌표/법선의 순서로 되어있고 위치를 제외한 나머지는 선택적으로 존재합니다. 즉, 아래와 같습니다.

```
f 1 2 3              # 위치에 대한 인덱스만 존재
f 1/1 2/2 3/3        # 위치와 텍스처 좌표에 대한 인덱스만 존재
f 1/1/1 2/2/2 3/3/3  # 위치, 텍스처 좌표, 법선에 대한 인덱스 존재
f 1//1 2//2 3//3     # 위치와 법선에 대한 인덱스만 존재
```

`f` 는 3개 이상의 정점을 가질 수 있는데 예를 들어 쿼드의 경우 4개입니다. 알다시피 WebGL은 삼각형만 그릴 수 있기 때문에 이 경우 데이터를 삼각형으로 변환해야 합니다. 면이 4개 이상의 정점을 가질 수 있는지, 면이 볼록해야 하는지 오목해야 하는지 등은 정의하고 있지 않습니다. 지금은 그냥 오목하다고 가정합시다.

또한 일반적으로 WebGL에서는 위치, 텍스처 좌표와 법선에 다른 인덱스를 사용하지 않습니다. 대신 "WebGL 정점"이란 정점과 관련된 모든 데이터 세트를 의미합니다. 예를 들어 WebGL에서 정육면체를 그리기 위해서는 36개의 정점이 필요하고, 각 면은 2개의 삼각형으로 이루어져 있으며 각 삼각현은 3개의 정점으로 정의됩니다. 6개의 면 *2개의 삼각형* 삼각형마다 3개의 정점이므로 36입니다. 중복되지 않는 정점 위치는 8개, 법선은 6개고 중복되지 않는 텍스처 좌표는 몇개가 될지 모릅니다. 그러니, 각 면의 정점 인덱스를 읽어와서 "WebGL 정점"을 생성해야 하고, 이는 3개 데이터 세트입니다. *

지금까지 언급한 내용에 따라 다음과 같이 파싱할 수 있을 겁니다.

```
function parseOBJ(text) {
  // 인덱스는 1부터 시작하므로, 0번째 데이터는 그냥 채워 넣습니다.
  const objPositions = [[0, 0, 0]];
  const objTexcoords = [[0, 0]];
  const objNormals = [[0, 0, 0]];

  // `f` 인덱스의 정의 순서와 같습니다.
  const objVertexData = [
    objPositions,
    objTexcoords,
    objNormals,
  ];

  // `f` 인덱스의 정의 순서와 같습니다.
  let webglVertexData = [
    [],   // 위치
    [],   // 텍스처 좌표
    [],   // 법선
  ];

  function addVertex(vert) {
    const ptn = vert.split('/');
    ptn.forEach((objIndexStr, i) => {
      if (!objIndexStr) {
        return;
      }
      const objIndex = parseInt(objIndexStr);
      const index = objIndex + (objIndex >= 0 ? 0 : objVertexData[i].length);
      webglVertexData[i].push(...objVertexData[i][index]);
    });
  }

  const keywords = {
    v(parts) {
      objPositions.push(parts.map(parseFloat));
    },
    vn(parts) {
      objNormals.push(parts.map(parseFloat));
    },
    vt(parts) {
      objTexcoords.push(parts.map(parseFloat));
    },
    f(parts) {
      const numTriangles = parts.length - 2;
      for (let tri = 0; tri < numTriangles; ++tri) {
        addVertex(parts[0]);
        addVertex(parts[tri + 1]);
        addVertex(parts[tri + 2]);
      }
    },
  };
```

위 코드는 OBJ 파일에서 파싱한 위치, 텍스처 좌표, 법선을 저장하는 3개의 배열을 생성합니다. 또한 WebGL을 위해 동일하게 3개의 배열을 생성합니다. `f` 인덱스와 동일한 순서로 배열에 저장되며 이는 `f`를 파싱할때 참조를 용이하게 합니다.

다시말해 `f` 라인은 아래와 같은데,

```
f 1/2/3 4/5/6 7/8/9
```

예를 들어 `4/5/6` 은 이 면의 정점으로 "4번 위치 정보를 사용"하고, "5번 텍스처 좌표를 사용"하며 "6번 법선 정보를 사용"한다는 뜻입니다. 하지만 위치, 텍스처 좌표, 법선 배열을 또 다른 배열인 `objVertexData` 배열에 저장하여 "webglData i는 objData i의 n번째를 사용"하는 식으로 코드를 간략화 할 수 있습니다.

함수의 마지막 부분에서는 생성된 데이터를 반환합니다.

```
  ...

  return {
    position: webglVertexData[0],
    texcoord: webglVertexData[1],
    normal: webglVertexData[2],
  };
}
```

이제 남은 것은 데이터를 그려보는 것입니다. 우선 방향성 조명 효과 글에서 사용한 셰이더를 약간 변형하여 사용합니다.

```
const vs = `
attribute vec4 a_position;
attribute vec3 a_normal;

uniform mat4 u_projection;
uniform mat4 u_view;
uniform mat4 u_world;

varying vec3 v_normal;

void main() {
  gl_Position = u_projection * u_view * u_world * a_position;
  v_normal = mat3(u_world) * a_normal;
}
`;

const fs = `
precision mediump float;

varying vec3 v_normal;

uniform vec4 u_diffuse;
uniform vec3 u_lightDirection;

void main () {
  vec3 normal = normalize(v_normal);
  float fakeLight = dot(u_lightDirection, normal) * .5 + .5;
  gl_FragColor = vec4(u_diffuse.rgb * fakeLight, u_diffuse.a);
}
`;
```

그리고 유틸리티 함수에서 사용한 코드를 활용하여 먼저 데이터를 로딩합니다.

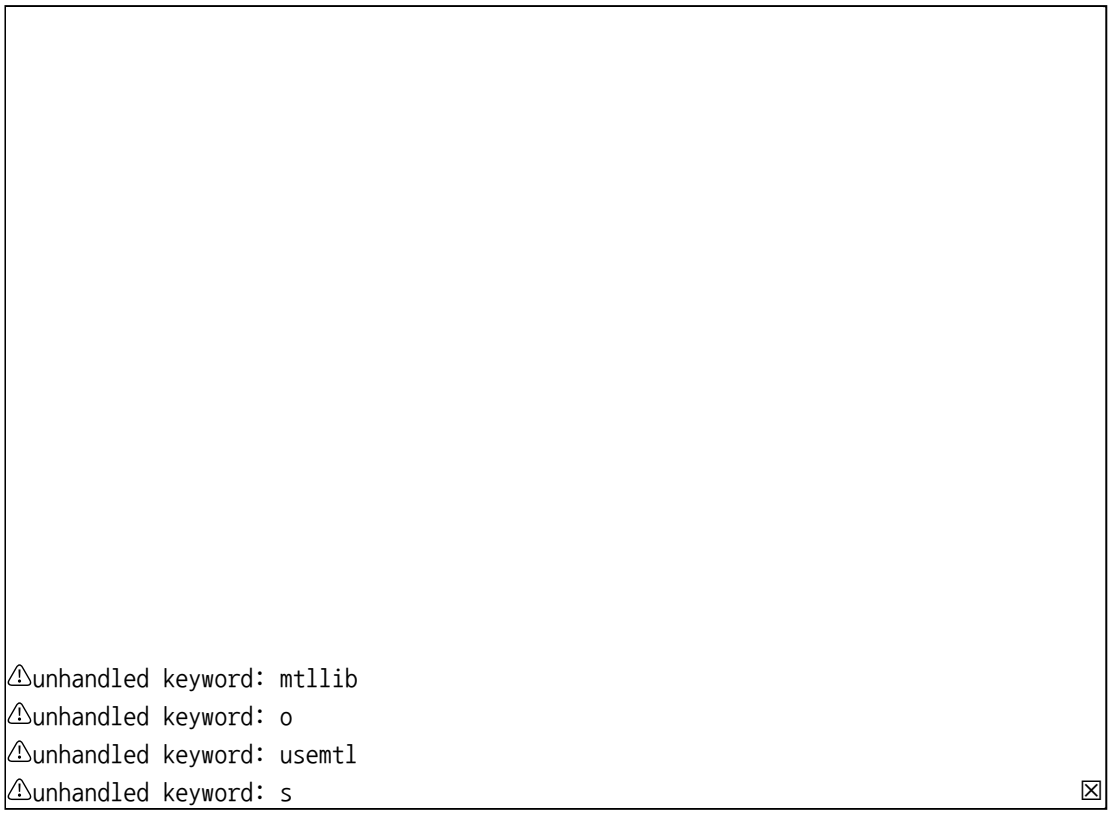
```
async function main() {
  // WebGL Context 얻기
  /** @type {HTMLCanvasElement} */
  const canvas = document.querySelector("#canvas");
  const gl = canvas.getContext("webgl");
  if (!gl) {
    return;
  }

  ... shaders ...

  // 셰이더를 컴파일하고 링크합니다. 또한 속성과 유니폼의 위치를 찾습니다.
  const meshProgramInfo = webglUtils.createProgramInfo(gl, [vs, fs]);

  const data = await loadOBJ('resources/models/cube/cube.obj');

  // 데이터는 아래와 같이 명명된 배열이고,
  //
  // {
  //   position: [...],
  //   texcoord: [...],
  //   normal: [...],
  // }
  //
  // 위 이름들은 정점 셰이더의 속성과 매치되는 이름을 가지고 있으므로,
  // 유틸리티 함수에 대한 글에서 했던 것처럼 바로 "createBufferInfoFromArrays"로 넘겨줄 수 있습니다.

  // gl.createBuffer, gl.bindBuffer, gl.bufferData를 호출하여 각 배열에 대한 버퍼를 생성해 줍니다.
  const bufferInfo = webglUtils.createBufferInfoFromArrays(gl, data);
```

그리고 화면에 그려줍니다.

```
  const cameraTarget = [0, 0, 0];
  const cameraPosition = [0, 0, 4];
  const zNear = 0.1;
  const zFar = 50;

  function degToRad(deg) {
    return deg * Math.PI / 180;
  }

  function render(time) {
    time *= 0.001;  // 초 단위로 변환

    webglUtils.resizeCanvasToDisplaySize(gl.canvas);
    gl.viewport(0, 0, gl.canvas.width, gl.canvas.height);
    gl.enable(gl.DEPTH_TEST);
    gl.enable(gl.CULL_FACE);

    const fieldOfViewRadians = degToRad(60);
    const aspect = gl.canvas.clientWidth / gl.canvas.clientHeight;
    const projection = m4.perspective(fieldOfViewRadians, aspect, zNear, zFar);

    const up = [0, 1, 0];
    // lookAt을 활용해 카메라 행렬 계산
    const camera = m4.lookAt(cameraPosition, cameraTarget, up);

    // 카메라 행렬로 뷰 행렬 생성
    const view = m4.inverse(camera);

    const sharedUniforms = {
      u_lightDirection: m4.normalize([-1, 3, 5]),
      u_view: view,
```

```
      u_projection: projection,
    };

    gl.useProgram(meshProgramInfo.program);

    // gl.uniform 호출
    webglUtils.setUniforms(meshProgramInfo, sharedUniforms);

    // gl.bindBuffer, gl.enableVertexAttribArray, gl.vertexAttribPointer 호출
    webglUtils.setBuffersAndAttributes(gl, meshProgramInfo, bufferInfo);

    // gl.uniform 호출
    webglUtils.setUniforms(meshProgramInfo, {
      u_world: m4.yRotation(time),
      u_diffuse: [1, 0.7, 0.5, 1],
    });

    // gl.drawArrays 또는 gl.drawElements 호출
    webglUtils.drawBufferInfo(gl, bufferInfo);

    requestAnimationFrame(render);
  }
  requestAnimationFrame(render);
}
```

그러면 우리의 정육면체가 로딩되어 그려지는 것을 볼 수 있습니다.




[새 창을 열려면 여기를 클릭]

또한 처리되지 않은 키워드에 대한 메세지를 볼 수 있습니다. 왜 나오는걸까요?

`usemtl` 은 그중 가장 중요한 메세지입니다. 이것은 머티리얼을 사용하는 모든 지오메트리에 등장합니다. 예를 들어 자동차 모델이 있다면 아마도 투명한 창문과 크롬 도금된 범퍼가 표현되는 것이 좋을겁니다. 창문은 투명하고 범퍼는 반사해서 차체의 다른 파트와는 다른 방식으로 그려져야 할 필요가 있습니다. `usemat` 태그는 이렇게 다른 파트의 구분을 나타냅니다.

파트마다 다르게 그려져야 하기 때문에 코드를 수정해서 `usemtl` 이 나타날때마다 새로운 WebGL 데이터 세트를 생성하도록 합시다.

먼저 아무것도 존재하지 않는 경우 새로운 WebGL 데이터를 생성하도록 합니다.

```
function parseOBJ(text) {
  // 인덱스는 1부터 시작하므로, 0번째 데이터는 그냥 채워 넣습니다.
  const objPositions = [[0, 0, 0]];
  const objTexcoords = [[0, 0]];
  const objNormals = [[0, 0, 0]];

  // `f` 인덱스의 정의 순서와 같습니다.
  const objVertexData = [
    objPositions,
    objTexcoords,
    objNormals,
  ];

  // `f` 인덱스의 정의 순서와 같습니다.
  let webglVertexData = [
    [],   // 위치
    [],   // 텍스처 좌표
    [],   // 법선
  ];

  const geometries = [];
  let geometry;
  let material = 'default';

  function newGeometry() {
    // 이미 지오메트리가 있고 데이터가 비어있지 않으면 새로운 지오메트리를 생성합니다.
    if (geometry && geometry.data.position.length) {
      geometry = undefined;
    }
  }

  function setGeometry() {
    if (!geometry) {
      const position = [];
      const texcoord = [];
      const normal = [];
      webglVertexData = [
        position,
        texcoord,
        normal,
      ];
      geometry = {
        material,
        data: {
          position,
          texcoord,
          normal,
        },
      };
      geometries.push(geometry);
    }
  }

...
```

그래도 해당 함수를 키워드를 처리할 때 적절한 곳에서 호출합니다. 또한 `o` 키워드에 대한 함수를 추가합니다.

```
  ...

  const keywords = {
    v(parts) {
      objPositions.push(parts.map(parseFloat));
    },
    vn(parts) {
      objNormals.push(parts.map(parseFloat));
    },
    vt(parts) {
      objTexcoords.push(parts.map(parseFloat));
    },
    f(parts) {
      setGeometry();
      const numTriangles = parts.length - 2;
      for (let tri = 0; tri < numTriangles; ++tri) {
        addVertex(parts[0]);
        addVertex(parts[tri + 1]);
        addVertex(parts[tri + 2]);
      }
    },
    usemtl(parts, unparsedArgs) {
      material = unparsedArgs;
      newGeometry();
    },
  };

  ...
```

usemtl 가 파일에 꼭 필요하지는 않습니다. usemtl 키워드가 파일에 없어도 지오메트리를 생성하도록 동작해야 하기 때문에 f 를 처리하는 과정에서 setGeometry 를 호출하여 usemtl 이 없었더라도 지오메트리가 생성되도록 했습니다.

또한 마지막 부분에 geometries 를 반환하는데 이는 물체의 배열로써 각 물체는 name 과 data 를 갖고 있습니다.

```
  ...

  return {
    position: webglVertexData[0],
    texcoord: webglVertexData[1],
    normal: webglVertexData[2],
  };
  return geometries;
}
```

그리고 텍스처 좌표나 법선 데이터가 없는 경우에는 그것들을 파싱하지 않도록 하는 케이스를 추가합니다.

```
  // 데이터가 없는 배열은 제거합니다.
  for (const geometry of geometries) {
    geometry.data = Object.fromEntries(
        Object.entries(geometry.data).filter(([, array]) => array.length > 0));
  }
```

```
  return {
    materialLibs,
    geometries,
  };
}
```

키워드에 대해 더 이야기해보자면, *공식 명세*에 따르면, `mtllib` 는 머티리얼 정보를 포함하는 별도의 파일을 가리킵니다. 안타깝게도 실제 상황과는 맞지 않는것이 파일 이름에는 공백이 포함될 수 있고 OBJ 포맷은 공백이나 따옴표를 처리할 방법이 없습니다. 이상적으로는 json 이나 xml, yaml과 같은 잘 정의된 포맷을 사용했으면 이런 문제를 해결할 수 있었겠지만 사실 .OBJ는 이러한 포맷들보다도 더 오래된 포맷입니다.

파일 로딩에 관해서는 좀 더 나중에 처리하도록 합시다. 지금은 로딩 과정에 추가해 놓고 나중에 참조할 수 있을 정도로만 해두겠습니다.

```
function parseOBJ(text) {
  ...
  const materialLibs = [];

  ...

  const keywords = {
    ...
    mtllib(parts, unparsedArgs) {
      materialLibs.push(unparsedArgs);
    },
    ...
  };

  return geometries;
  return {
    materialLibs,
    geometries,
  };
}
```

`o` 는 이후에 등장하는 데이터가, 이름이 붙여진 특정 "물체"에 속한다는 것을 나타냅니다. 이 정보를 어떻게 사용해야 할지는 정확하지 않습니다. `o` 는 있는데 `usemtl` 이 없는 파일이 있을 수도 있을까요? 그렇다고 가정합시다.

```
function parseOBJ(text) {
  ...
  let material = 'default';
  let object = 'default';

  ...

  function setGeometry() {
    if (!geometry) {
      const position = [];
      const texcoord = [];
      const normal = [];
      webglVertexData = [
        position,
        texcoord,
```

```
        normal,
      ];
      geometry = {
        object,
        material,
        data: {
          position,
          texcoord,
          normal,
        },
      };
      geometries.push(geometry);
    }
  }

  const keywords = {
    ...
    o(parts, unparsedArgs) {
      object = unparsedArgs;
      newGeometry();
    },
    ...
  };
```

`s` 는 스무딩 그룹을 나타냅니다. 제 생각에 스무딩 그룹은 대부분 무시해도 되는 데이터입니다. 대개 스무딩 그룹은 모델링 프로그램에서 정점 법선을 자동 생성하기 위해 사용됩니다. 정점 법선의 계산을 위해, 우선 *외적*을 통해 손쉽게 계산할 수 있는 각 면의 법선을 계산합니다. 이는 카메라에 관한 글에 설명한 바 있습니다. 그리고 나서 정점을 공유하는 모든 면에 대해 법선의 평균을 정점 법선으로 사용합니다. 하지만 각진 가장자리가 필요할 때는 계산 과정에서 면을 무시하도록 지정할 수 있어야 하는데요. 스무딩 그룹은 정점 법선을 계산할 때 어떤 면이 포함되어야 하는지를 명시할 수 있도록 합니다. 지오메트리의 정점 법선에 대한 계산에 대해서는 선반 가공에 관한 글의 예제를 참고하십시오.

우리의 경우 그냥 무시하는 것으로 합시다. 대개 .OBJ 파일은 법선 정보를 이미 포함하고 있고 그래서 스무딩 그룹이 필요 없습니다. 스무딩 그룹이 명세에 존재하는 이유는 모델링 패키지에서 모델을 수정해 법선을 다시 계산할 경우를 대비해 가지고 있기 위함입니다.

```
  const noop = () => {};

  const keywords = {
    ...
    s: noop,
    ...
  };
```

아직 다루지 않은 하나 남은 키워드는 그룹을 위한 `g` 입니다. 기본적으로 이는 메타데이터입니다. 물체는 하나 이상의 그룹에 속할 수 있습니다. 다음 파일에 이 키워드가 등장하므로 지금은 사용되지 않지만 일단 이에 대한 처리 기능은 추가하도록 합시다.

```
function parseOBJ(text) {
  ...
  let groups = ['default'];
  ...
```

```
  function setGeometry() {
    if (!geometry) {
      const position = [];
      const texcoord = [];
      const normal = [];
      webglVertexData = [
        position,
        texcoord,
        normal,
      ];
      geometry = {
        object,
        groups,
        material,
        data: {
          position,
          texcoord,
          normal,
        },
      };
      geometries.push(geometry);
    }
  }

  ...

  const keywords = {
    ...
    g(parts) {
      groups = parts;
      newGeometry()
    },
    ...
  };
```

이제 여러 지오메트리 세트을 생성하기 때문에 각각에 대해 WebGLBuffers 를 생성하도록 설정 부분의 코드를 수정해야 합니다. 또한 임의의 색상을 사용해 각 파트가 서로 다르게 보이도록 해 봅시다.

```
  const response = await fetch('resources/models/cube/cube.obj');
  const response = await fetch('resources/models/chair/chair.obj');
  const text = await response.text();
  const data = parseOBJ(text);
  const obj = parseOBJ(text);

  const parts = obj.geometries.map(({data}) => {
    // 데이터는 아래와 같이 명명된 배열(named array)이고,
    //
    // {
    //   position: [...],
    //   texcoord: [...],
    //   normal: [...],
    // }
    //
    // 위 이름들은 정점 셰이더의 속성과 매치되는 이름을 가지고 있으므로,
    // 유틸리티 함수에 대한 글에서 했던 것처럼 바로 "createBufferInfoFromArrays"로 넘겨줄 수 있습니다.

    // gl.createBuffer, gl.bindBuffer, gl.bufferData를 호출하여 각 배열에 대한 버퍼를 생성해 줍니다.
    const bufferInfo = webglUtils.createBufferInfoFromArrays(gl, data);
    return {
      material: {
        u_diffuse: [Math.random(), Math.random(), Math.random(), 1],
      },
```

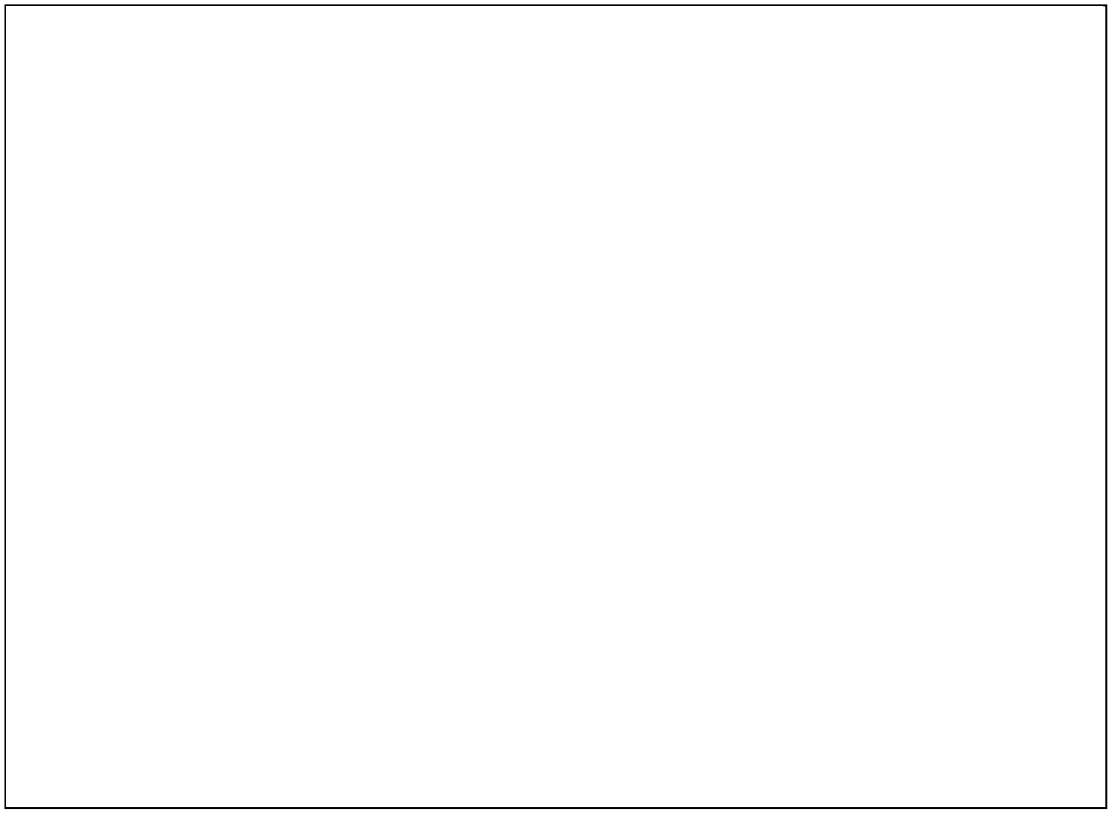
```
      bufferInfo,
    };
  });
```

정육면체 대신 haytonm가 만든 CC-BY 4.0 의자를 로딩하도록 바꿨습니다. 이 모델은 Sketchfab에서 찾았습니다.

렌더링을 위해서는 반복문을 통해 각 파트들을 처리해주면 됩니다.

```
function render(time) {
  ...

  gl.useProgram(meshProgramInfo.program);

  // gl.uniform 호출
  webglUtils.setUniforms(meshProgramInfo, sharedUniforms);

  // 모든 파트가 동일 공간상에 있기 때문에 월드 행렬은 한 번만 계산합니다.
  const u_world = m4.yRotation(time);

  for (const {bufferInfo, material} of parts) {
    // gl.bindBuffer, gl.enableVertexAttribArray, gl.vertexAttribPointer 호출
    webglUtils.setBuffersAndAttributes(gl, meshProgramInfo, bufferInfo);
    // gl.uniform 호출
    webglUtils.setUniforms(meshProgramInfo, {
      u_world: m4.yRotation(time),
      u_diffuse: [1, 0.7, 0.5, 1],
      u_world,
      u_diffuse: material.u_diffuse,
    });
    // gl.drawArrays 또는 gl.drawElements 호출
    webglUtils.drawBufferInfo(gl, bufferInfo);
  }

  ...
```

일단 동작은 하네요.



새 창을 열려면 여기를 클릭

물체를 가운데 놓으면 더 좋겠죠?

이를 위해 정점 위치의 최대 및 최소값인 범위를 계산해야 합니다. 먼저 주어진 위치 데이터의 최소값과 최대값을 찾는 함수를 만듭니다.

```
function getExtents(positions) {
  const min = positions.slice(0, 3);
  const max = positions.slice(0, 3);
  for (let i = 3; i < positions.length; i += 3) {
    for (let j = 0; j < 3; ++j) {
      const v = positions[i + j];
      min[j] = Math.min(v, min[j]);
      max[j] = Math.max(v, max[j]);
    }
  }
  return {min, max};
}
```

그리고 지오메트리에 포함된 모든 파트를 순회하여 전체 파트들의 범위를 계산합니다.

```
function getGeometriesExtents(geometries) {
  return geometries.reduce(({min, max}, {data}) => {
    const minMax = getExtents(data.position);
    return {
      min: min.map((min, ndx) => Math.min(minMax.min[ndx], min)),
      max: max.map((max, ndx) => Math.max(minMax.max[ndx], max)),
    };
  }, {
    min: Array(3).fill(Number.POSITIVE_INFINITY),
    max: Array(3).fill(Number.NEGATIVE_INFINITY),
  });
```

```
}
```

이를 활용해 물체를 얼마나 움직여야 물체의 중심이 원점에 위치하게 되고, 카메라를 얼마나 멀리 배치해야 물체 전체를 볼 수 있을지를 계산합니다.

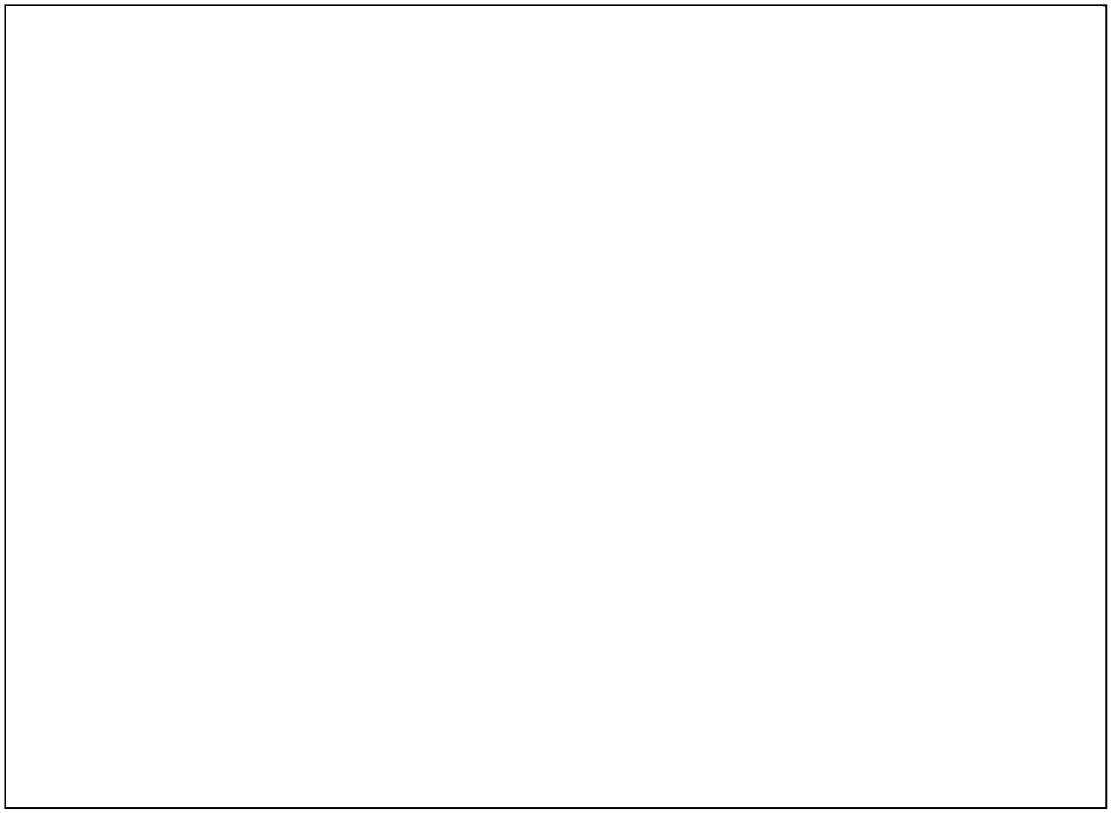
```
~~const cameraTarget = [0, 0, 0];~~
~~const cameraPosition = [0, 0, 4];~~
~~const zNear = 0.1;~~
~~const zFar = 50;~~
  const extents = getGeometriesExtents(obj.geometries);
  const range = m4.subtractVectors(extents.max, extents.min);
  // 물체의 중심이 원점에 오도록 하기 위해 움직여야 하는 거리
  const objOffset = m4.scaleVector(
      m4.addVectors(
        extents.min,
        m4.scaleVector(range, 0.5)),
      -1);
  const cameraTarget = [0, 0, 0];
  // 물체를 볼 수 있으려면 카메라를 얼마나 떨어뜨려야 하는지 계산합니다.
  const radius = m4.length(range) * 1.2;
  const cameraPosition = m4.addVectors(cameraTarget, [
    0,
    0,
    radius,
  ]);
  // 물체의 크기에 적절할 것 같은 "zNear"와 "zFar"를 설정합니다.
  const zNear = radius / 100;
  const zFar = radius * 3;
```

위에서는 물체가 잘 보일 것 같은 `zNear` 와 `zFar` 도 설정하였습니다.

`objOffset` 를 사용해 물체를 원점으로 이동시키기만 하면 됩니다.

```
// 모든 파트가 동일 공간상에 있기 때문에 월드 행렬은 한 번만 계산합니다.
~~const u_world = m4.yRotation(time);~~
let u_world = m4.yRotation(time);
u_world = m4.translate(u_world, ...objOffset);
```

그러면 물체가 중심에 오게 됩니다.



새 창을 열려면 여기를 클릭

인터넷을 돌아다니다 보면 정점의 색상을 포함하는 표준적이지 않은 버전의 .OBJ 파일을 볼 수 있습니다. 이러한 경우 각 정점 위치 뒤에 추가적인 데이터를 더해 놓았는데, 아래와 같은 포맷 대신

```
v <x> <y> <z>
```

아래와 같이 되어 있습니다.

```
v <x> <y> <z> <red> <green> <blue>
```

부가적으로 마지막에 알파값을 사용하는 경우도 있는지 모르겠습니다.

이런 예제를 발견했는데, Oleaf가 만든 CC-BY-NC Book - Vertex chameleon study, 정점 색상을 사용하고 있습니다.

정점 색상 또한 지원할 수 있도록 우리 파서를 수정할 수 있을지 봅시다.

위치, 법선, 텍스처 좌표 부분에 색상을 위한 코드들을 추가해야 합니다.

```
function parseOBJ(text) {
  // 인덱스는 1부터 시작하므로, 0번째 데이터는 그냥 채워 넣습니다.
  const objPositions = [[0, 0, 0]];
  const objTexcoords = [[0, 0]];
  const objNormals = [[0, 0, 0]];
  const objColors = [[0, 0, 0]];

  // `f` 인덱스의 정의 순서와 같습니다.
  const objVertexData = [
    objPositions,
    objTexcoords,
    objNormals,
    objColors,
  ];

  // `f` 인덱스의 정의 순서와 같습니다.
  let webglVertexData = [
    [],   // 위치
    [],   // 텍스처 좌표
    [],   // 법선
    [],   // 색상
  ];

  ...

  function setGeometry() {
    if (!geometry) {
      const position = [];
      const texcoord = [];
      const normal = [];
      const color = [];
      webglVertexData = [
        position,
        texcoord,
        normal,
        color,
      ];
      geometry = {
        object,
        groups,
        material,
        data: {
          position,
          texcoord,
```

```
          normal,
          color,
        },
      };
      geometries.push(geometry);
    }
  }
```

그리고 안타깝지만 실제 파싱 부분에서는 덜 일반적으로 코드가 됩니다.

```
  const keywords = {
    v(parts) {
      objPositions.push(parts.map(parseFloat));
      // 이 부분에서 값이 3개 이상이라면 정점 색상이 포함되어 있습니다.
      if (parts.length > 3) {
        objPositions.push(parts.slice(0, 3).map(parseFloat));
        objColors.push(parts.slice(3).map(parseFloat));
      } else {
        objPositions.push(parts.map(parseFloat));
      }
    },
    ...
  };
```

`f` 면에 대한 라인을 읽어올 때 `addVertex` 를 호출합니다. 정점 색상을 여기에서 읽어와야 합니다.

```
  function addVertex(vert) {
    const ptn = vert.split('/');
    ptn.forEach((objIndexStr, i) => {
      if (!objIndexStr) {
        return;
      }
      const objIndex = parseInt(objIndexStr);
      const index = objIndex + (objIndex >= 0 ? 0 : objVertexData[i].length);
      webglVertexData[i].push(...objVertexData[i][index]);
      // 이게 위치 인덱스(인덱스 0)이고 정점 색상을 파싱한 경우,
      // 정점 색상을 WebGL 정점 색상 데이터로 복사합니다.
      if (i === 0 && objColors.length > 1) {
        geometry.data.color.push(...objColors[index]);
      }
    });
  }
```

이제 셰이더에서 정점 색상을 사용하도록 수정해야 합니다.

```
const vs = `
attribute vec4 a_position;
attribute vec3 a_normal;
attribute vec4 a_color;

uniform mat4 u_projection;
uniform mat4 u_view;
uniform mat4 u_world;
```

```
varying vec3 v_normal;
varying vec4 v_color;

void main() {
  gl_Position = u_projection * u_view * u_world * a_position;
  v_normal = mat3(u_world) * a_normal;
  v_color = a_color;
}
`;

const fs = `
precision mediump float;

varying vec3 v_normal;
varying vec4 v_color;

uniform vec4 u_diffuse;
uniform vec3 u_lightDirection;

void main () {
  vec3 normal = normalize(v_normal);
  float fakeLight = dot(u_lightDirection, normal) * .5 + .5;
  gl_FragColor = vec4(u_diffuse.rgb * fakeLight, u_diffuse.a);
  vec4 diffuse = u_diffuse * v_color;
  gl_FragColor = vec4(diffuse.rgb * fakeLight, diffuse.a);
}
`;
```

위에서 언급했듯이 이 비표준적인 버전의 .OBJ파일이 각 정점 색상에 알파값을 포함할 수도 있는지는 모르겠습니다. 우리의 도우미 라이브러리는 우리가 전달한 데이터를 가지고 버퍼를 자동으로 만들어 줍니다. 라이브러리에서는 데이터의 요소마다 몇개의 컴포넌트가 있는지 추측합니다. `position` 이나 `normal` 같은 이름이 붙은 데이터는 요소마다 3개의 컴포넌트라고 가정합니다. `texcoord` 같은 이름이 붙은 데이터는 요소마다 2개의 컴포넌트라고 가정합니다. 나머지에 대해서는 요소마다 4개라고 가정합니다. 그 말은 색상이 요소마다 r, g, b 세 개의 컴포넌트만을 가진다면, 4개로 가정하지 않도록 알려 주어야 한다는 뜻입니다.

```
const parts = obj.geometries.map(({data}) => {
  // 데이터는 아래와 같이 명명된 배열이고,
  //
  // {
  //   position: [...],
  //   texcoord: [...],
  //   normal: [...],
  // }
  //
  // 위 이름들은 정점 셰이더의 속성과 매치되는 이름을 가지고 있으므로,
  // 유틸리티 함수에 대한 글에서 했던 것처럼 바로 "createBufferInfoFromArrays"로 넘겨줄 수 있습니다.

  // gl.createBuffer, gl.bindBuffer, gl.bufferData를 호출하여 각 배열에 대한 버퍼를 생성해 줍니다.

   if (data.position.length === data.color.length) {
     // 값은 3입니다. 도우미 라이브러리는 4라고 가정하기 때문에 데이터가 3개만 있다고 알려줍니다.
     data.color = { numComponents: 3, data: data.color };
   }

  // gl.createBuffer, gl.bindBuffer, gl.bufferData를 호출하여 각 배열에 대한 버퍼를 생성해 줍니다.
  const bufferInfo = webglUtils.createBufferInfoFromArrays(gl, data);
  return {
    material: {
      u_diffuse: [Math.random(), Math.random(), Math.random(), 1],
    },
```

```
    bufferInfo,
  };
});
```

정점 색상이 없는 보다 흔한 경우에 대해 여전히 처리할 수 있도록 하면 좋겠습니다. 첫 번째 글및 다른 글에서 속성은 대개 버퍼에서부터 데이터를 가져온다고 했습니다. 하지만 사실 속성을 상수로 만들수도 있습니다. 비활성화된 속성은 상수값을 사용합니다. 예를 들어,

```
gl.disableVertexAttribArray(someAttributeLocation);  // 상수값을 사용
const value = [1, 2, 3, 4];
gl.vertexAttrib4fv(someAttributeLocation, value);    // 사용할 상수
```

우리의 도우미 라이브러리는 속성의 데이터를 `{value: [1, 2, 3, 4]}` 로 설정하면 이러한 경우를 대신 처리해 줍니다. 따라서 정점 색상이 없다면 정점 색상 속성을 흰색으로 설정하면 됩니다.

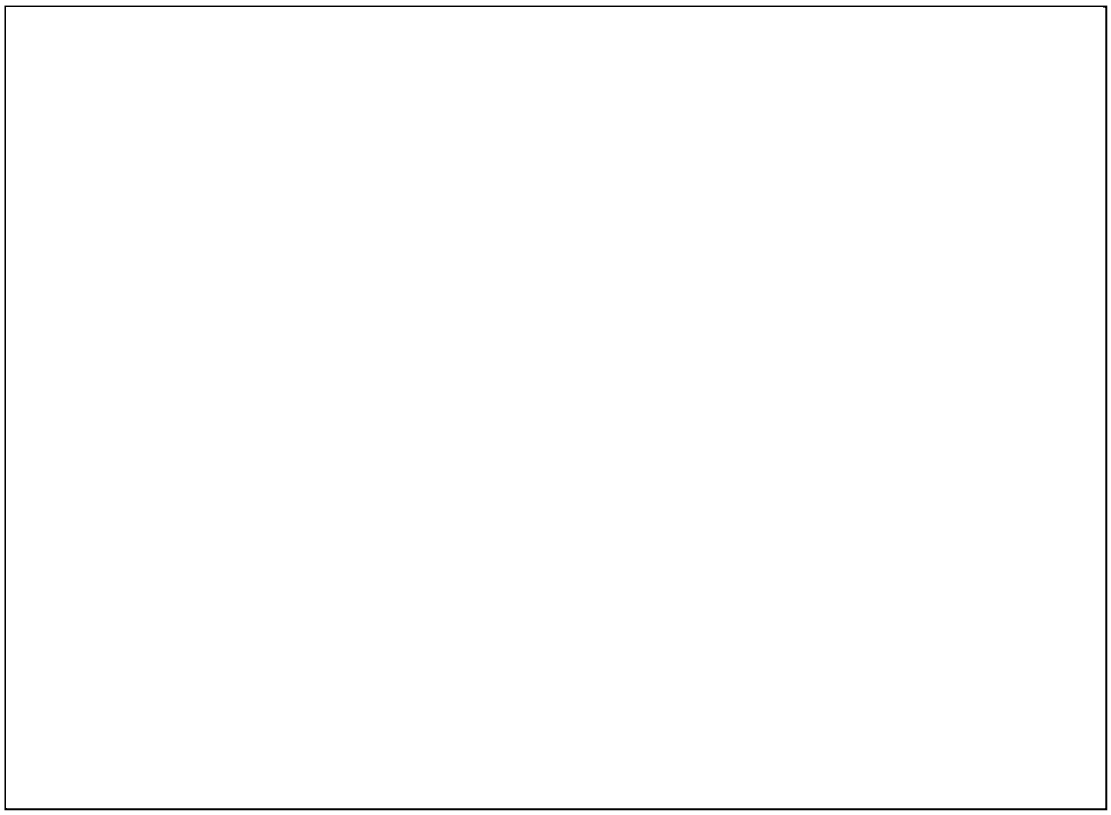
```
const parts = obj.geometries.map(({data}) => {
  // 데이터는 아래와 같이 명명된 배열이고,
  //
  // {
  //   position: [...],
  //   texcoord: [...],
  //   normal: [...],
  // }
  //
  // 위 이름들은 정점 셰이더의 속성과 매치되는 이름을 가지고 있으므로,
  // 유틸리티 함수에 대한 글에서 했던 것처럼 바로 "createBufferInfoFromArrays"로 넘겨줄 수 있습니다.

  if (data.color) {
      if (data.position.length === data.color.length) {
        // 값은 3입니다. 도우미 라이브러리는 4라고 가정하기 때문에 데이터가 3개만 있다고 알려줍니다.
        data.color = { numComponents: 3, data: data.color };
      }
  } else {
    // 정점 색상이 없으므로 흰색으로 설정해줍니다.
    data.color = { value: [1, 1, 1, 1] };
  }

  ...
});
```

이제는 파트마다 임의의 색상을 사용할 수는 없습니다.

```
const parts = obj.geometries.map(({data}) => {
  ...

  // gl.createBuffer, gl.bindBuffer, gl.bufferData를 호출하여 각 배열에 대한 버퍼를 생성해 줍니다.
  const bufferInfo = webglUtils.createBufferInfoFromArrays(gl, data);
  return {
    material: {
      u_diffuse: [Math.random(), Math.random(), Math.random(), 1],
      u_diffuse: [1, 1, 1, 1],
    },
```

```
    bufferInfo,
  };
});
```

이제 정점 색상을 포함하는 .OBJ 파일을 로딩할 수 있습니다.



새 창을 열려면 여기를 클릭

머티리얼을 파싱하고 사용하는 것은 다음 글을 보시면 됩니다.

# 주의사항

**여기 있는 로더는 불완전합니다.**

.obj 포맷에 대해 읽어보십시오. 위 코드가 지원하지 않는 기능은 너무나 많습니다. 또한, 위 코드는 다양한 .obj에 테스트한 것도 아니라서 버그가 숨어있을 수도 있습니다. 제 생각엔 인터넷에 있는 대다수의 .obj파일들은 위에 보여드린 기능만을 사용하고 있으므로, 위 예제만으로 이미 유용할 것입니다.

**로더는 오류를 처리하고 있지 않습니다.**

예를 들어, `vt` 키워드는 2개가 아닌 3개의 값을 가지고 있을 수 있습니다. 3개의 값이 있는 경우는 3D 텍스처를 사용하는 경우를 위함인데, 흔하게 사용되는 것이 아니므로 신경쓰지 않았습니다. 여러분이 3D 텍스처 좌표가 있는 파일을 사용한다면 셰이더에서도 3D 텍스처를 처리하도록 수정해야 하고, `createBufferInfoFromArrays`를 호출하여 `WebGLBuffers`를 생성하는 코드에도 UV 좌표마다 3개의 값이 있다고 알려주어야 합니다.

**데이터가 일관성 있게 표현되었다고 가정하고 있습니다.**

같은 파일 안에서 `f` 키워드에 어떠한 경우 3개의 값을 어떠한 경우 2개의 값을 가지고 있는 경우가 있는지 모르겠습니다. 그러한 경우가 가능하다면, 위 코드는 사용할 수 없습니다.

또한 정점 위치가 x, y, z로 표현되어 있다면 항상 x, y, z라고 가정했습니다. 어떤 파일에서 정점 위치가 어떤 경우에는 x, y, z로, 다른 경우에는 x, y로, 또 다른 경우에는 x, y, z, r, g, b로 되어 있다면 코드를 수정해야 합니다.

**모든 데이터를 하나의 버퍼에 집어넣을 수도 있습니다.**

위 코드에서는 정점 위치, 텍스처 좌표, 법선 데이터를 별개의 버퍼에 집어넣고 있습니다. 하나의 버퍼에 pos,uv,nrm,pos,uv,nrm,...과 같이 묶어서 넣을 수도 있습니다. 하지만 그러한 경우엔 attribute의 설정 과정에서 stride와 offset을 전달하도록 코드를 수정해야 합니다.

더 나아가서 모든 파트에 대한 데이터를 하나의 버퍼에 집어넣을수도 있지만 현재는 파트마다, 그리고 데이터마다 하나씩의 버퍼를 사용하고 있습니다.

그것들은 다루지 않을 예정인데 그렇게 중요하게 생각되지도 않을뿐더러 예제가 복잡해지기 때문입니다.

**정점의 인덱스를 바꿀 수 있습니다.**

위 코드는 정점을 확장해 삼각형의 정점 배열로 바꾸었습니다. 정점들의 인덱스를 바꿀 수 있습니다. 만일 우리가 모든 정점 데이터를 하나의 버퍼에 넣거나 또는 타입마다 하나의 버퍼를 사용하면서 다른 파트에서는 이 값을들 공유하게 하려는 경우 `f` 키워드마다 인덱스를 양수로 바꾸고(음수를 올바른 양수 인덱스로 이동), 해당 숫자를 정점의 *아이디*로 설정하면 됩니다. 그렇게 해서 *아이디 -> 인덱스 맵*을 저장하고 인덱스를 찾아볼 때 사용하면 됩니다.

```
const idToIndexMap = {}
const webglIndices = [];

function addVertex(vert) {
  const ptn = vert.split('/');
  // 먼저 모든 인덱스를 양수로 변경
  const indices = ptn.forEach((objIndexStr, i) => {
    if (!objIndexStr) {
      return;
    }
    const objIndex = parseInt(objIndexStr);
    return objIndex + (objIndex >= 0 ? 0 : objVertexData[i].length);
  });
  // 특정한 위치, 텍스처 좌표, 법선이 존재하는지 확인
  const id = indices.join(',');
  let vertIndex = idToIndexMap[id];
  if (!vertIndex) {
    // 없는 경우 추가
    vertIndex = webglVertexData[0].length / 3;
    idToIndexMap[id] = vertexIndex;
    indices.forEach((index, i) => {
      if (index !== undefined) {
        webglVertexData[i].push(...objVertexData[i][index]);
      }
    }
  }
  webglIndices.push(vertexIndex);
}
```

또는 그 작업이 필요하다고 생각되면 수동으로 인덱스를 새로 부여할 수 있습니다.

**위 코드는 위치 데이터만 존재하는 경우 또는 위치 + 텍스처 좌표만 존재하는 경우는 처리하지 못합니다.**

코드는 법선 정보가 존재한다고 가정하고 작성되었습니다. 이전 예제처럼 법선 데이터가 없는 경우 필요에 따라 스무딩 그룹도 고려하여 법선 데이터를 생성할 수 있습니다. 아니면 법선을 필요로 하지않는, 또는 법선을 계산하는 다른 셰이더를 사용할 수도 있습니다.

**.OBJ 파일을 사용하면 안됩니다.**

솔직히, 제 생각엔 .OBJ 파일은 사용하면 안됩니다. 이 글은 예제 목적으로 쓰여졌습니다. 파일에서부터 정점 데이터를 가져올 수 있다면 어떤 포맷에 대해서도 가져오기 도구를 구현하실 수 있을겁니다.

.OBJ파일이 갖고있는 문제점은

- 조명 또는 카메라를 지원하지 않음

  조명이나 카메라가 필요하지 않고 단지 여러 파트들(예를 들어 배경을 위한 나무, 덤불, 바위 등등)을 로딩하려는 경우에는 괜찮을 수 있습니다. 그래도 아티스트가 만든 전체 장면을 로딩하려 하는 경우에는 이러한 표현을 지원하는 것이 좋겠죠.

- 계층 구조 및 장면 그래프 없음

  자동차 모델을 로드한다면 바퀴는 그 중심축을 기준으로 회전하는 것이 좋을겁니다. .OBJ에서는 이것이 불가능한데 .OBJ는 장면 그래프가 없기 때문입니다. 더 좋은 포맷들은 그러한 데이터를 포함하고 있어서 파트의 자세를 변경하거나 창문을 열거나 캐릭터의 다리를 움직이는 등의 작업을 하고 싶을때 훨씬 유용합니다.

- 애니메이션 또는 스키닝 기능 지원하지 않음

  스키닝에 대해 이야기했었는데 .OBJ는 스키닝을 위한 데이터를 지원하지 않고 애니메이션을 위한 데이터도 지원하지 않습니다. 역시나, 상황에 따라 필요없을수도 있지만 저는 더 많은 표현을 지원하는 포맷을 선호합니다.

- .OBJ는 모던 머티리얼을 지원하지 않음

  머티리얼은 원래 꽤나 엔진에 종속적이었지만 최근에는 물리 기반 렌더링 머티리얼에 대한 어느정도의 공통된 합의가 존재합니다. .OBJ는 제가 아는한 관련된 표현을 지원하지 않습니다.

- .OBJ는 파싱이 필요함

  .OBJ파일을 유저가 업로드하여 확인할 수 있는 뷰어를 만드는 것이 아니라면 사실 파싱을 가장 덜 필요로하는 포맷을 사용하는 것이 좋습니다. .GLTF는 WebGL을 위해 설계

된 포맷입니다. 이 포맷은 JSON 형식이므로 그냥 로드할 수 있습니다. 바이너리의 경우 GPU에 바로 로드할 수 있는 포맷을 사용하고 있어서 대부분의 경우 숫자들을 파싱하여 배열에 집어넣는 작업을 할 필요가 없습니다.

.GLTF를 로딩하는 예제는 스키닝에 대한 글에서 볼 수 있습니다.

사용하고 싶은 .OBJ 파일이 있다면 가장 좋은 방법은 먼저 그 파일을 여러분의 페이지에 맞는 더 좋은 다른 포맷으로 변환하여 사용하는 겁니다.