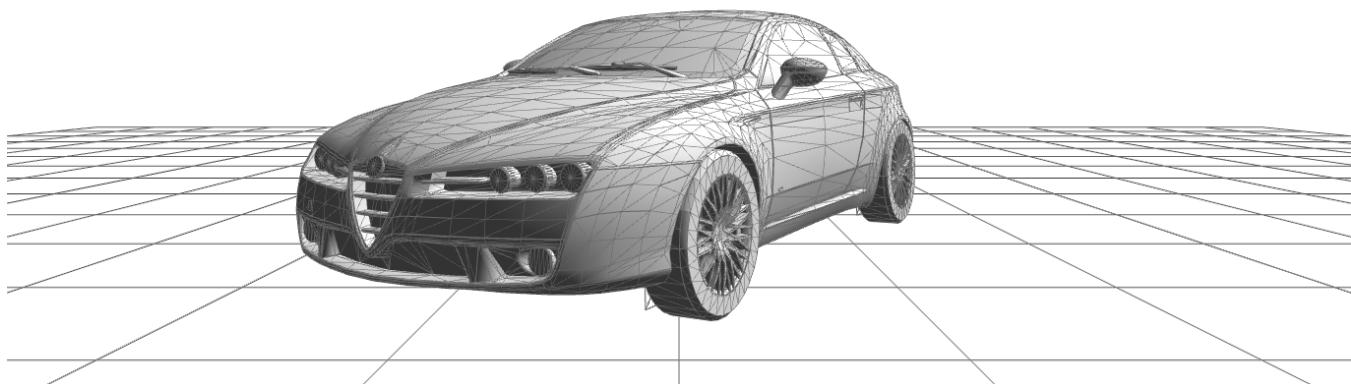


WebGL School 2018 プラスワン講義

# three.js と 3D モデルの連携

小山田 晃浩

---



WebGL コンテンツを作るには、3D モデルを扱う技術も必要です。例えば、ゲーム、表品ビューアー、お部屋シミュレータなど、Web のコンテンツとしてはさまざま考えられます。

three.js には数多くのプリミティブ图形が用意されていますが、それだけでは足りません。本講義では glTF と呼ばれる 3D モデル形式についてを知り、glTF を three.js で扱う方法を解説します。

# 配信用 3D モデル形式: glTF

画像の一般的な形式には PNG や JPEG があります。音声には MP3 がありますし、動画には MP4 があります。それでは、3D モデルではどんな形式がもっとも一般的でしょうか？

3D モデル形式には、.obj や.fbx、.dae (COLLADA) といった汎用形式や、.3ds や.blend などの 3D モデリングツール専用ファイル形式がすでに存在します。しかし、これらのファイル形式はモデリングツールに強い依存があったり、また、たとえ汎用形式であってもオフラインでの利用が前提のため、インターネットでの利用を考えると非効率です。

glTF (GL Transmission Format) はこうした問題を解決するために登場した 3D ランタイム向けのファイル形式で、glTF 2.0 が 2017 年に仕様として定められています。

glTF は完全にオープンで誰でも利用でき、プラットフォームにも依存しないファイル形式です。3D モデルにおける PNG や MP4 のような立ち位置を目指して登場しました。glTF は、インターネットでの配信を前提に作られた WebGL や OpenGL を始めとする 3D ランタイム向けのファイル形式なのです。

glTF の仕様策定は、クロノス・グループという、HTML でいう W3C のような団体が GitHub 上で進めており、現在も glTF の拡張仕様や問題点などのディスカッションが活発に行われています。

The screenshot shows the GitHub repository page for 'KhrongosGroup / glTF'. At the top, there's a dark header with the GitHub logo, a search bar, and navigation links for 'Pull requests', 'Issues', 'Marketplace', and 'Explore'. Below the header, the repository name 'KhrongosGroup / glTF' is displayed along with a dropdown menu. To the right are buttons for 'Watch' (270), 'Star' (2,196), 'Fork' (429), and a bell icon. A horizontal navigation bar below the repository name includes links for 'Code', 'Issues 130', 'Pull requests 21', 'ZenHub', 'Projects 0', 'Wiki', and 'Insights'. The main content area is titled 'glTF – Runtime 3D Asset Delivery'. It features a summary bar with metrics: 2,945 commits, 6 branches, 4 releases, and 77 contributors. Below this are buttons for 'Branch: master', 'New pull request', 'Create new file', 'Upload files', 'Find file', and 'Clone or download'. A list of recent activity is shown, including a merge pull request from 'peted70/master' by 'pjcozzi' (18 hours ago), a commit to 'extensions' (a month ago), a merge pull request to 'specification' (2 months ago), a commit to '.gitignore' (11 months ago), and a commit to 'README.md' (2 days ago). The GitHub interface is light gray with dark blue and green accents.

glTF の MIME type は `model/gltf+json` として正式に登録されており、一つの仕様として公に認められていると言えます。

<https://www.iana.org/assignments/media-types/media-types.xhtml#model>

vnd.si.simp - OBSOLETE by request	<a href="#">message/vnd.si.simp</a>	[Nicholas Parks Young]
vnd.wfa.wsc	<a href="#">message/vnd.wfa.wsc</a>	[Mick Conley]
<b>model</b>		
<b>Available Formats</b>		
 <a href="#">CSV</a>		
Name	Template	Reference
3mf	<a href="#">model/3mf</a>	[ <a href="http://www.3mf.io/specification">http://www.3mf.io/specification</a> ][3MF][Michael Sweet]
example	<a href="#">model/example</a>	[RFC4735]
gltf-binary	<a href="#">model/gltf-binary</a>	[Khronos][Saurabh Bhatia]
gltf+json	<a href="#">model/gltf+json</a>	[Khronos][Uli Klumpp]
iges	<a href="#">model/iges</a>	[Curtis Parks]
mesh		[RFC2077]
stl	<a href="#">model/stl</a>	[DICOM Standards Committee][Lisa Spellman]
vnd.collada+xml	<a href="#">model/vnd.collada+xml</a>	[James Riordon]
vnd.dxf	<a href="#">model/vnd.dxf</a>	[Jesse Porta]

さて、MIME type を見て気づいた方がいるかもしれません、glTF は JSON 形式です。ですので、テキストエディタで開くこともできます。ただ、実際のところ、JSON ではありますが人が手で書くファイルではありません。glTF ファイルは 3D モデリングツールにより直接出力されることになります。一方で、JSON 形式であるため、解析がしやすいという利点があり JavaScript で読み込んで WebGL で利用することができます。

glTF は 2015 年に初めて 1.0 がリリースされました。2017 年には、より抽象的で汎用性のある glTF 2.0 がリリースされました。glTF 2.0 では、glTF 1.0 に存在した shader の直接参照が削除される一方、PBR (物理ベースレンダリング) \*向けパラメーターの対応などが追加されています。

#### 用語：物理ベースレンダリング：

物理ベースレンダリング (Physically Based Shading) は、光の反射や散乱など現実の物理現象を再現するレンダリング手法です。

glTFの一般的なファイル構造の例を、glTF 1.0とglTF 2.0で見てみましょう。glTF 1.0のファイルの内容例は、次のようなコードになります（安心してください、人間が手で書く前提のファイルではありません）。

```
"programs": {
    "program0": {
        "attributes": [
            "a_position"
        ],
        "fragmentShader": "fragmentShader0",
        "vertexShader": "vertexShader0"
    }
},
"shaders": {
    "vertexShader0": {
        "type": 35633,
        "uri": "./my-vertex-shader-source.txt"
    },
    "fragmentShader0": {
        "type": 35632,
        "uri": "./my-fragment-shader-source.txt"
    }
},
```

glTF 2.0でのPBRであれば、ベースカラー（アルベド）、ラフネス（ざらつき）、メタリック（金属質）の最低限の値があれば表現できます。

<https://github.com/KhronosGroup/glTF/blob/master/specification/2.0/README.md#pbrmetallicroughness>

glTF 2.0のファイルの内容例は次のようなコードになります（安心してください、こちらも人間が手で書く前提のファイルではありません）。

```
{
    "materials": [
        {
            "name": "gold",
            "pbrMetallicRoughness": {
                "baseColorFactor": [ 1.000, 0.766, 0.336, 1.0 ],
                "metallicFactor": 1.0,
                "roughnessFactor": 0.0
            }
        }
    ],
}
```

こうした抽象性により、OpenGL や DirectX などラインタイムや Shader 言語に依存することなく、glTF は幅広く利用できます。

glTF はメッシュやマテリアルはもちろん、ボーンアニメーション、モーフアニメーションを定義できるフィールドも用意されており、たとえばゲームのキャラクターなどにも利用できます。また、拡張領域も用意されており、拡張の一つとして Draco 圧縮サポートがあります\*。ただし、実際に拡張が利用できるかは glTF のエンコーダーとデコーダーの機能に依存します。

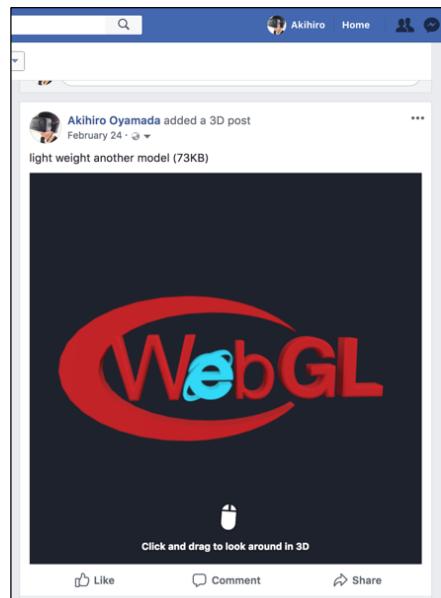
#### 用語：Draco

Draco は Google が開発した、オープンライセンスの 3D モデル専用の超強力な圧縮方式です。3D モデルのデータ容量を 1/10 程度にまで圧縮することが可能です。

すでに glTF は多くの 3D モデリングツール、ゲームエンジンや WebGL ライブラリーでサポートされています。

身近なところでは、Facebook では glTF ファイルをアップロードすることもできます。ARKit や ARCore 対応のスマートフォンのカメラで 3D スキャンした情景を、Facebook に載せる、なんていう時代もすぐ近くかもしれませんね。

glTF の仕様は複雑ですが、利用する上ではこれくらい理解しておけばいいでしょう。私たちは普段、Web での開発で PNG 画像を多用しますが、PNG 自体の仕様を深く知っている人は非常に稀ですよね。それと同じように、glTF についても、「**ファイル形式そのものの知識**」よりも「**利用できる知識**」のほうが重要だと筆者は考えています。



# glTF ファイルを出力する

ここからは、3D モデルを glTF ファイルとして出力する方法を解説します。glTF ファイルを出力するには、大きく分けて次の 2 通りの方法があります。

- 既存の 3D モデルファイル (.obj や.dae など) を、glTF 形式に変換する
- 3D モデリングツールを使い、glTF 形式で出力する

前者は、OBJ や COLLADA といった従来のファイル形式の 3D モデルデータを glTF に変換するだけの処理です。Web サービスやコマンドラインから変換機能を利用できます。変換の過程で 3D モデル自体の編集はできませんが、すでにある資産を使って glTF ファイル入手することが目的でしたら事足りるでしょう。

変換ツールの例としては、モデルファイルをドラッグ&ドロップするだけで利用できる Web サービスの「glTF Model Converter」などがあります。

<https://cesiumjs.org/convertmodel.html>



一方で、3D モデリングツールから 3D モデルを直接 glTF として出力して、glTF ファイルを取得する方法もあります。現在はモデリングツールのサードパーティー製エクステンションとして提供されていることが多いですが、glTF がさらに普及すればモデリングツールのビルトイン機能として提供されるかもしれません。

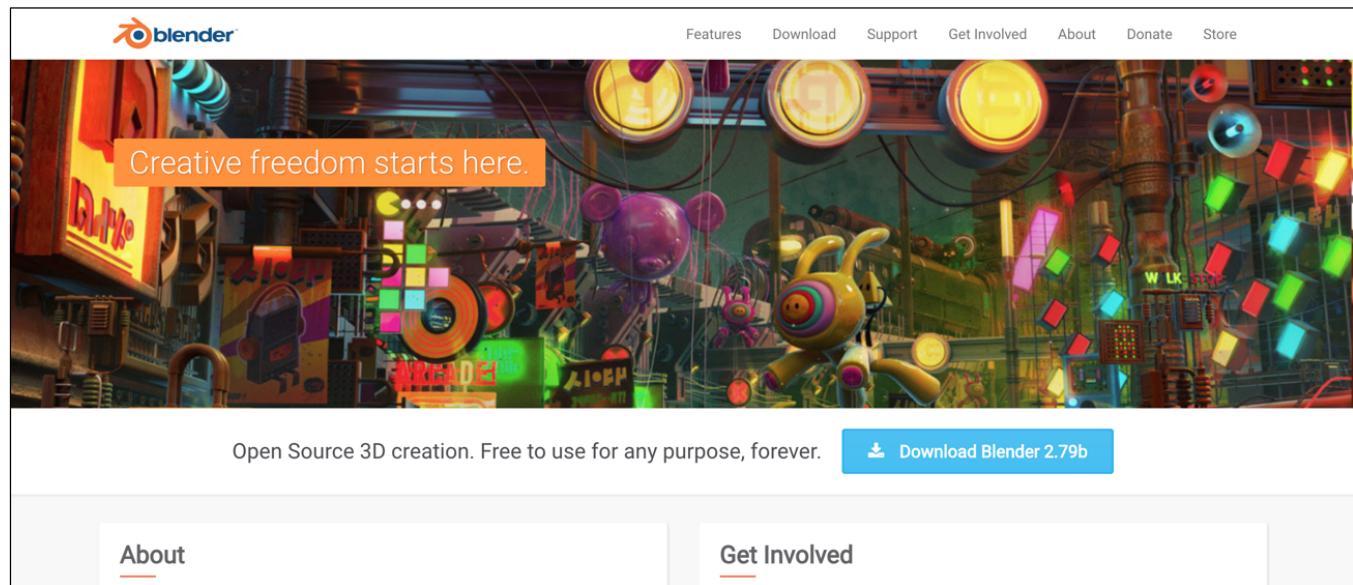
エクスポートの例としては、クロノス・グループが提供する Blender 用の「Khronos Group Blender Exporter」などがあります。

ここでは、「Khronos Group Blender Exporter」を利用した glTF の出力方法を見てみましょう。3D モデリングツールは高価なソフトウェアが多いですが、その中でも Blender はオープンソースで無償で利用でき、他の 3D モデリングツールにも引けを取らない機能を有します。

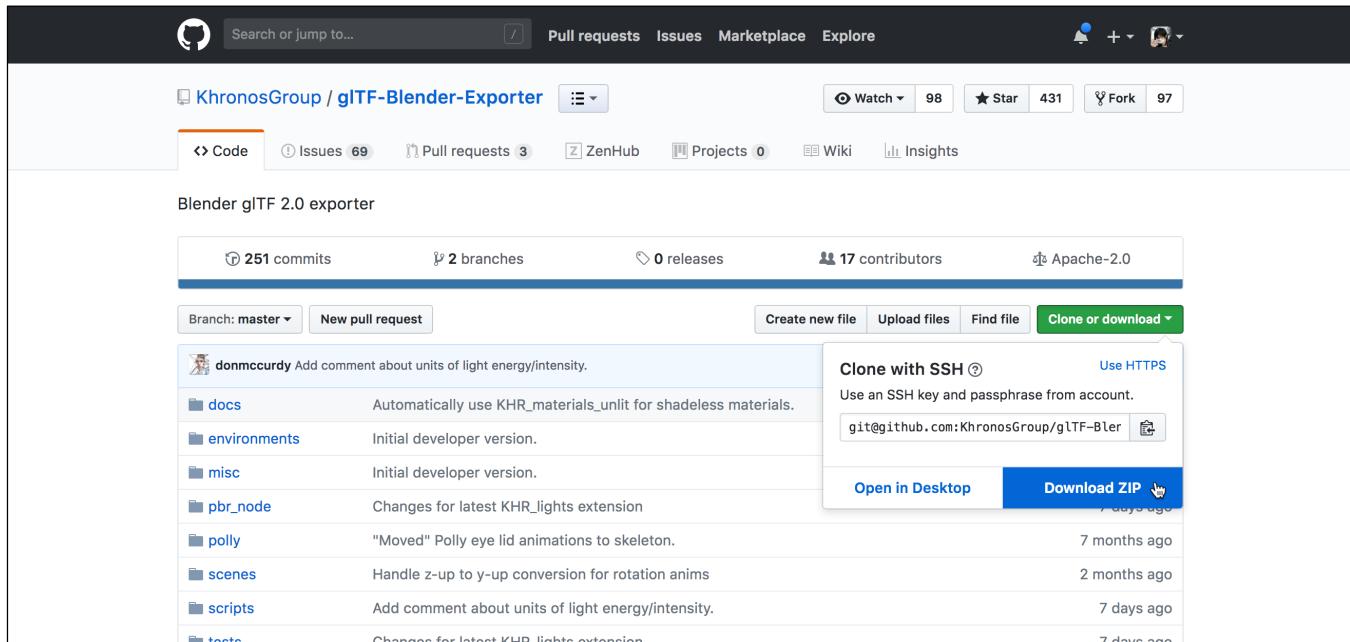
## Blender で 3D モデルを glTF に出力する

それでは、Blender を使って glTF に出力する方法を具体的に見てていきましょう。まず、Blender をインストールしていなければ、Blender を手に入れましょう。Blender はオープンソースの 3D モデリングツールでコミュニティーも活発です。Windows、macOS、Linux のそれぞれの OS 向けのアプリケーションを無償でダウンロードすることができます。

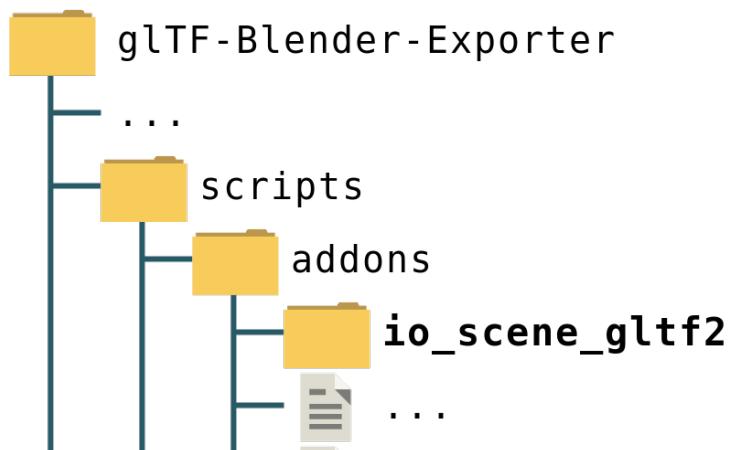
Blender のホームページ（<https://www.blender.org/>）から、「Download Blender」を選び、自分が使っている OS に合ったアプリケーションをダウンロードしインストールします。



加えて、Blender 用のアドオンとして提供されている「Khronos Group Blender Exporter」もダウンロードしておきましょう。「Khronos Group Blender Exporter」は GitHub で管理されています。プロジェクトのリポジトリ（<https://github.com/KhronosGroup/glTF-Blender-Exporter>）の「Clone or download」から、「Download ZIP」を選びアドオンを入手します。



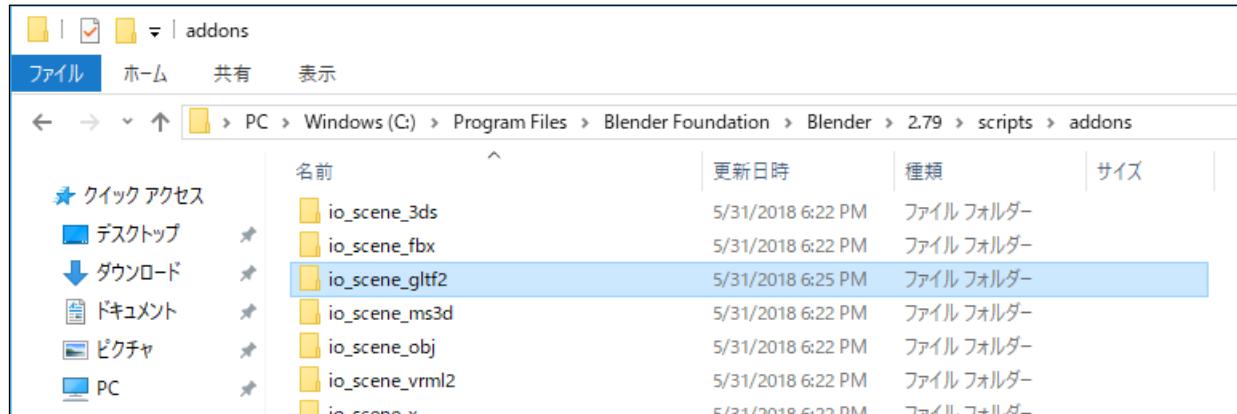
Blender 本体と「Khronos Group Blender Exporter」の両方が手元に揃ったら、「Khronos Group Blender Exporter」を Blender にインストールします。まずは、「io\_scene\_gltf2」フォルダーを指定のディレクトリーに配置します。配置場所は OS ごとに異なります。「io\_scene\_gltf2」フォルダーはダウンロードした ZIP ファイル内の「scripts / addons」の中に格納されています。



取り出した「io\_scene\_gltf2」フォルダーを、OS 環境に応じて次の位置に配置します。

## Windows 用 Blender

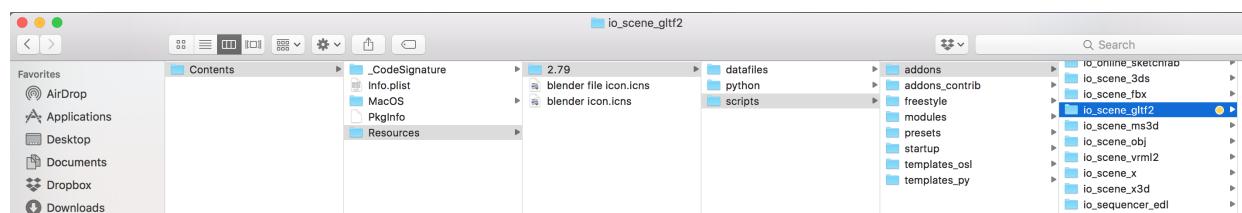
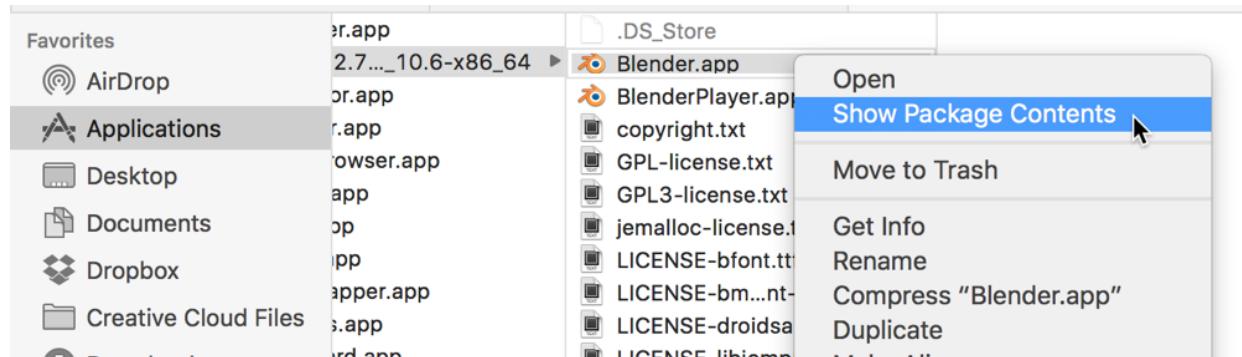
C:\Program Files\Blender Foundation\Blender\2.7X\scripts\addons\



## macOS 用 Blender

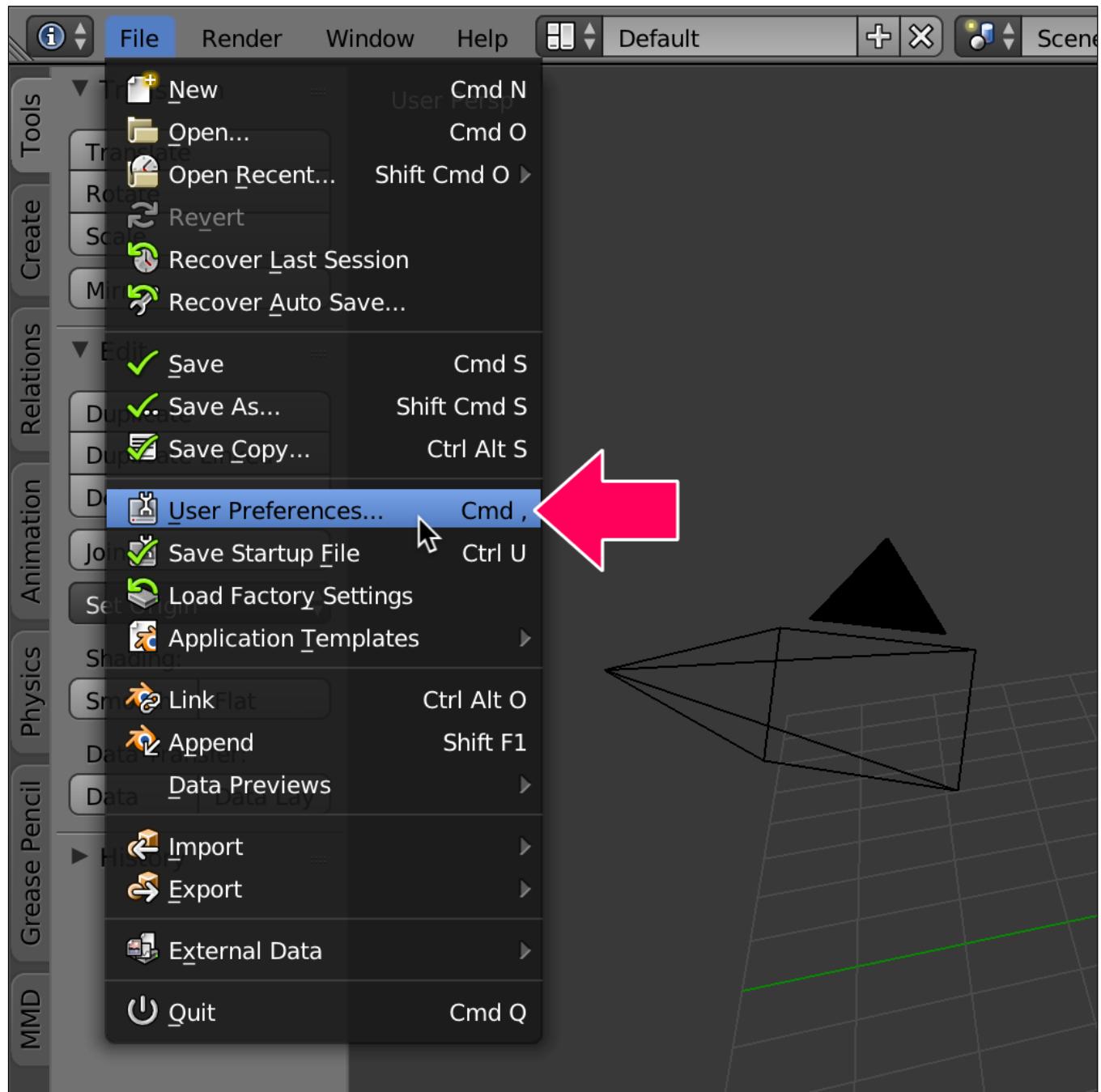
/Applications/blender.app/Contents/Resources/2.7X/scripts/addons/

blender.app はフォルダーではないため、通常は開けません。「blender.app」を右クリック（2本指クリック）し、「Show Package Contents」を選択するとその中にすすむことができます。



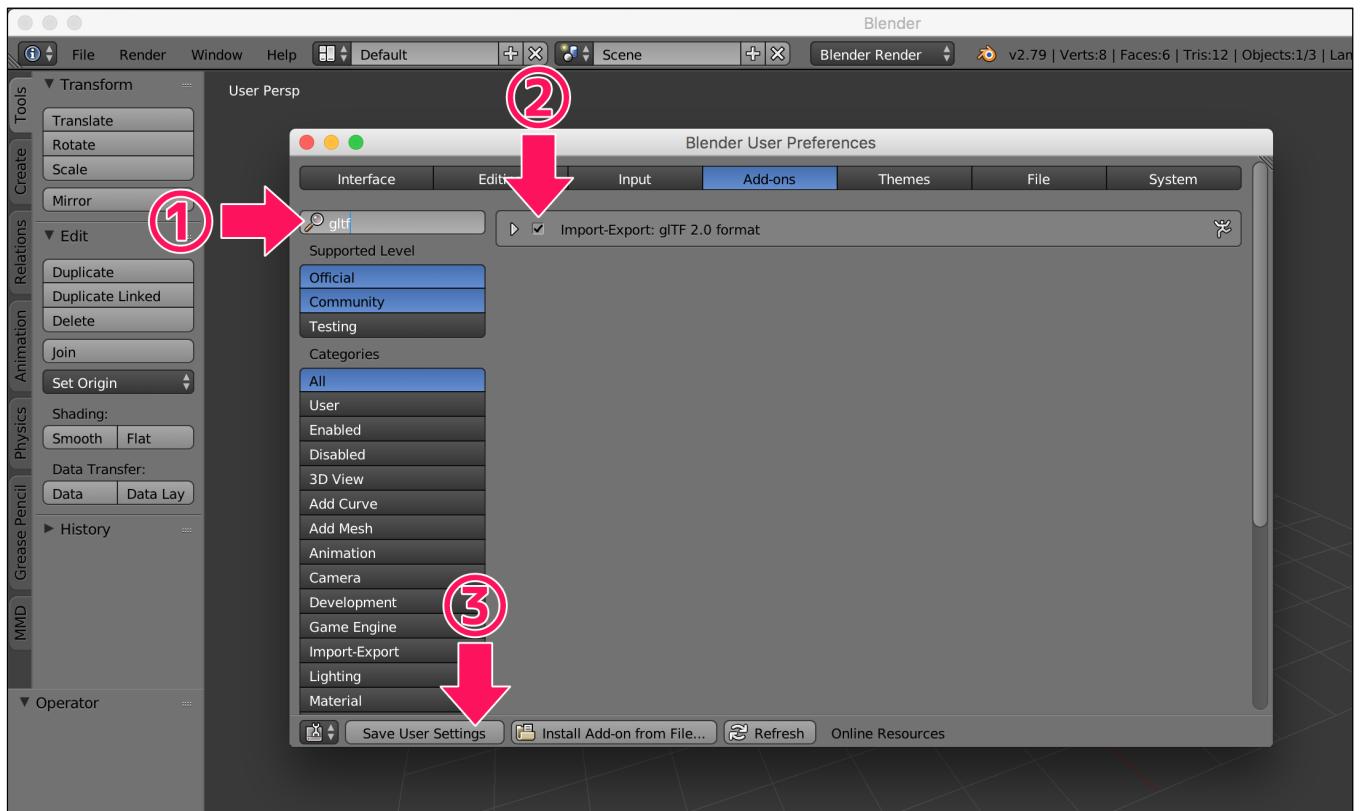
これから先は Windows も macOS も共通の操作となります。

「Khronos Group Blender Exporter」の配置が終わったら、Blender でアドオンを有効化します。Blender を起動し（Blender を開いている場合には一度終了する）、メニューバーの[File]から[User Preferences...]を選択して設定を開きます。



設定画面が開くので、次の手順でアドオンを有効化します。

1. [Add-ons]タブを選択し、「gltf」で検索する
2. 先程配置したアドオンが「Import-Export: glTF 2.0 format」という名前で見つかるのでこれをチェックする
3. 下部の[Save User Setting]ボタンを押して適用する

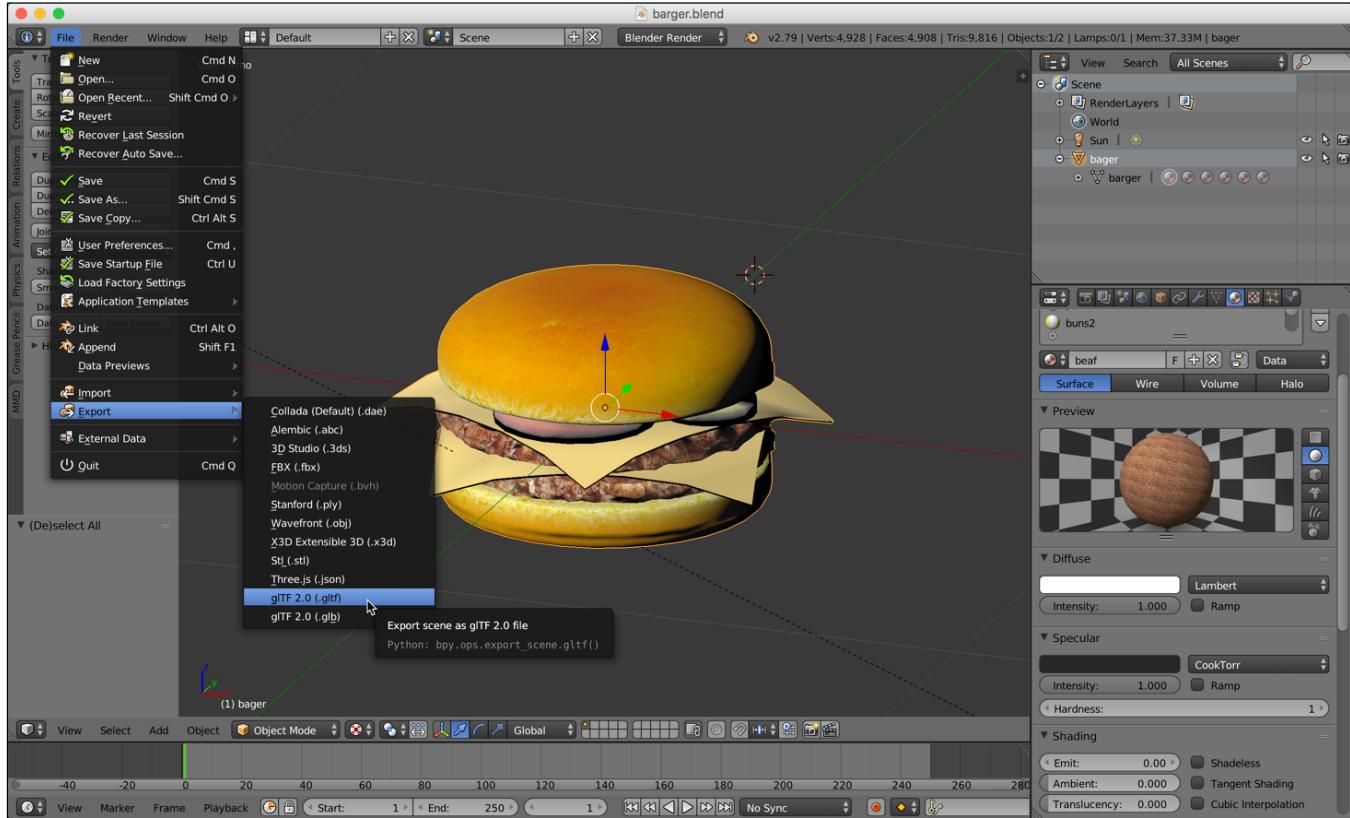


これでセットアップは完了です。Blender から glTF を出力してみましょう。操作の練習用の Blender 用のモデルファイルが以下から入手できます。実際に手を動かしながら試してみるといいでしょう。巻末の「Blender の基本操作」も参考にしてみてください。

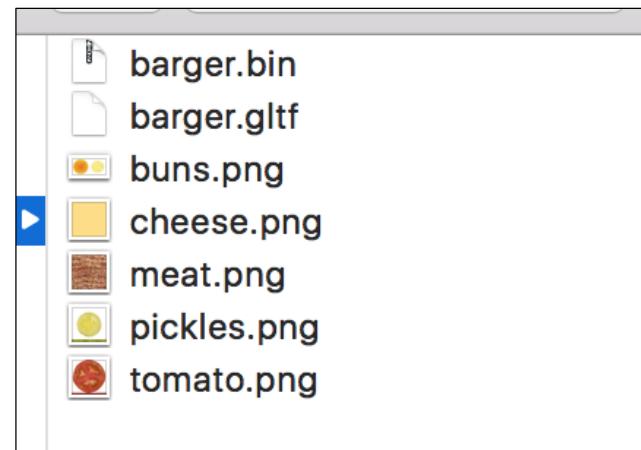
操作の練習用の Blender 用のモデルファイル

examples/blend/barge/barge.blend

Blender でモデルファイルを開きます。開いたら、メニューバーから[File]>[Export]>[glTF2.0 (glTF)]を選択します。もし、Exportの一覧に「glTF2.0」がない場合はアドオンのインストール手順を見直してください。



Export が完了すると、複数のファイルを得ることができます。本体となる、「.gltf」ファイル、メッシュなどの頂点がバイナリー形式で格納された「.bin」ファイル、そして、テクスチャ画像群です。ここで出力された複数のファイルはすべてが必要です。



余談ですが、本体の glTF ファイルをテキストエディタで開いてみると次のように「.gltf」ファイルからそれぞれのファイルを参照しています。

```
"buffers" : [
    {
        "byteLength" : 325416,
        "uri" : "barger.bin"
    }
],
"images" : [
    {
        "uri" : "meat.png"
    },
    {
        "uri" : "buns.png"
    },
    {
        "uri" : "cheese.png"
    },
    {
        "uri" : "pickles.png"
    },
    {
        "uri" : "tomato.png"
    }
],
```

これで WebGL 用の 3D モデルファイルを得ることができました！

## テクスチャー画像の形式と大きさに注意

Web で利用する場合、テクスチャー画像の形式は PNG や JPEG などの Web ブラウザーで表示できる形式になっている必要があります。

一方で、3D モデルでは、テクスチャー形式に TGA や TIFF、DDS など Web ブラウザーが対応していない形式が利用されていることがあります。その場合、glTF からリンクされている画像ファイルが Web ブラウザー未対応の形式のまま出力されてしまうことがあります。

テクスチャー画像が Web フレンドリーでない場合、3D モデラー上でテクスチャー画像の形式をあらかじめ変えておく、または、glTF 出力後に画像ファイルを JPEG などに変換することで、glTF を Web ブラウザーで読み込むようになります。画像形式には注意するよにしましょう。

また、テクスチャー画像がPNGやJPEGで用意されていたとしても、画像が巨大な場合があります。Webで使う場合、大きなテクスチャー画像であっても2048pxあればいいでしょう。大きすぎるテクスチャー画像は、ダウンロードに時間がかかります。必要に応じて縮小しておきましょう。ただし、テクスチャー画像は2のべき乗サイズを保つように注意しましょう。2のべき乗サイズでない場合、浮動小数点（float型）として処理がしづらく、パフォーマンスが落ちたり画像の縮小がうまくできなくなってしまいます。

## 関連ファイルをまとめられる glb 形式

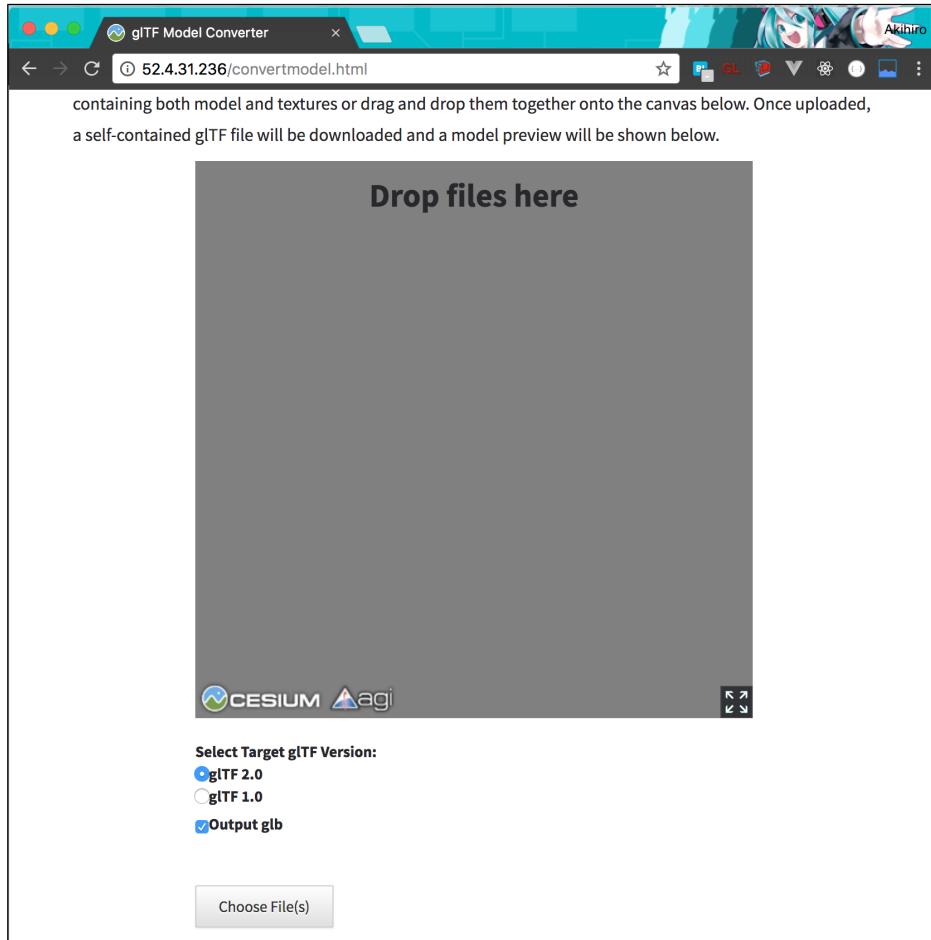
基本的にglTFによる3Dモデルは、JSON形式のglTFファイルと、そこから参照されるメッシュデータファイルや画像ファイル群の複数の関連ファイルで構成されています<sup>\*1</sup>。しかし、複数ファイルに分かれてしまっていると管理は煩雑になってしまいます。

<sup>\*1</sup> glTFファイルからの関連リソースは、基本的に相対URLで参照されています。実は、URLによる参照にはdataURIを利用することもできます。その場合glTFのJSONファイル内にすべてのリソースを埋め込んでしまうこともできます。ただし、dataURI化されたリソースは本来のファイルよりも30%ほど容量が増えてしまいます。

そのため、これらを一つのファイルにまとめることができます。この单一ファイルはglb（glTF-Binary）と呼ばれています。

glTFのファイル群とglbとは、内容はまったく同じで、glTF群からglbに、またはglbからglTF群にと相互に変換することができます。

「glTF Model Converter（<https://cesiumjs.org/convertmodel.html>）」を使うと、glTFとglbの相互変換を行うことができます。画面下部の「Select Target glTF Version」で、「glTF 2.0」と「Output glb」にチェックが付いた状態でglTFファイル一式を画面内にドラッグ＆ドロップすると单一のglbファイルに変換することができます。



さて、一見、glb のほうが管理しやすく、glTF より優れているように見えます。しかし、glb は単一ファイルであるがゆえに中身がわかりません。

- glb: 単一ファイルだが、中身はわからない
- glTF: 複数ファイルで、構成要素がわかる

glTF、glb 両方の特徴を理解し、私は次のフローを取ることをおすすめします。

1. 開発時は glTF で出力し、ファイル構成を確認する
2. もしテクスチャー画像ファイルが大きすぎたり、画像形式が Web ブラウザーで扱えない場合には編集したあと、glb に変換してプロダクション用ファイルとする

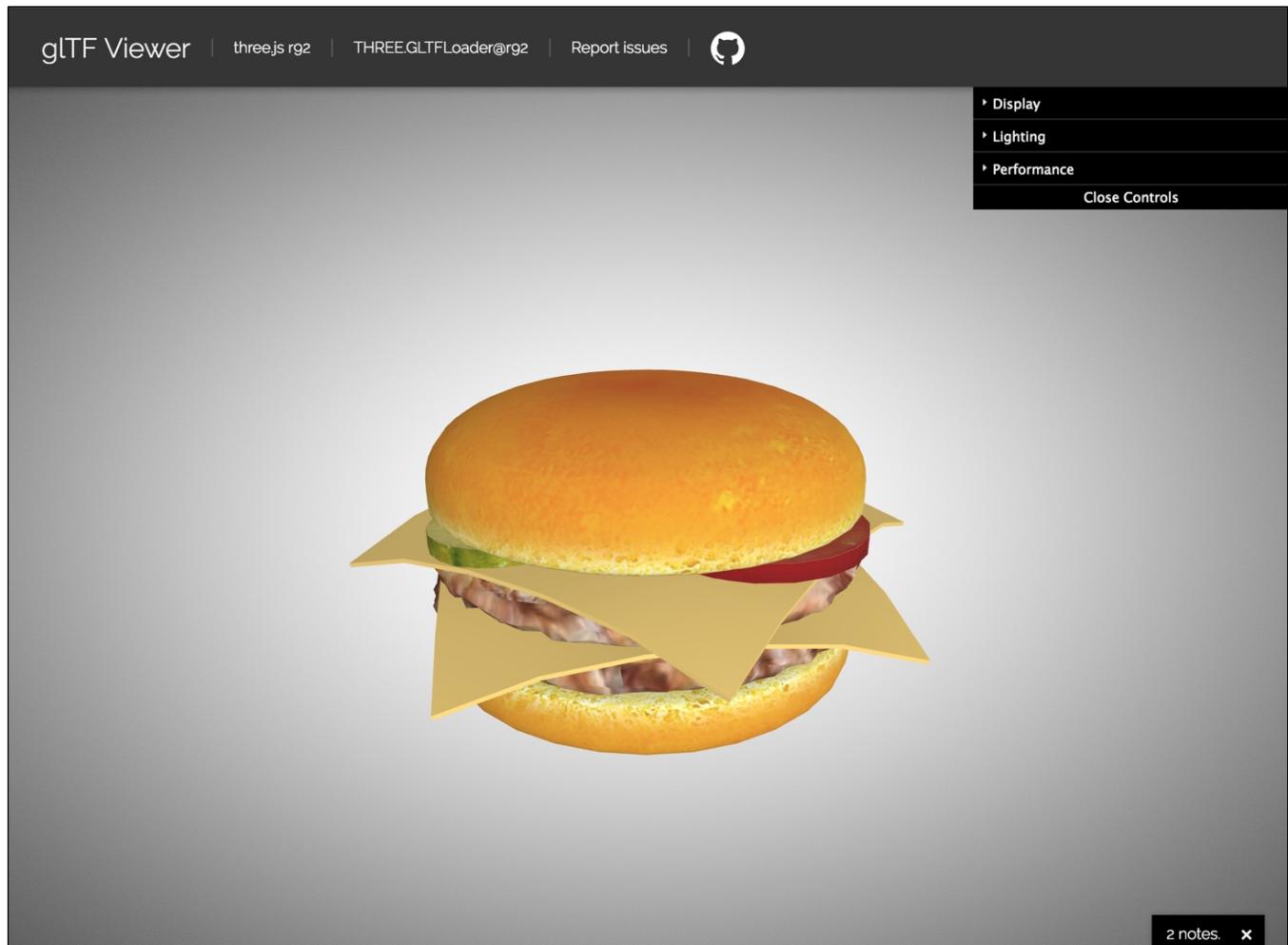
というようにする。

なお、Facebook への 3D モデル投稿は glb の単一ファイルである必要があります。

## glTF を Web ブラウザーで表示する

得られた glTF ファイルを Web ブラウザーで表示してみましょう。「glTF ビューアー（<https://gltf-viewer.donmccurdy.com/>）」を使うとドラッグ&ドロップだけで glTF ファイルを表示できます。

本体の「.gltf」ファイル、「.bin」ファイル、テクスチャーバイナリをすべてまとめてビューアー内にドロップしてみましょう。



どうです？ Web ブラウザー上で表示できましたか？

でもせっかくなら、あなたの Web アプリケーションの中で、glTF を自由に表示してみたいですね。three.js や babylon.js は glTF の取り込みに対応しています。

(ところで、先述の glTF ビューアーも、実は **three.js** で作られているんですよ)

# Three.js で glTF をロードしてみよう

three.js では、さまざまな形式の 3D モデルファイルのローディング機能がサポートされています。three.js 本体とはさまざまな追加ローダーが用意されており、3D モデルのファイル形式に合わせてローダーを選択して利用します。もちろん、glTF ファイルのローダーも用意されています。

<https://threejs.org/examples/?q=loader>



Blender の操作により出力できた glTF ファイルを、three.js を使って表示してみましょう。前述の通り three.js 本体には glTF を読み込む機能はありませんが、three.js のプラグインとして GLTFLoader が用意されています。また、three.js の GLTFLoader は glTF の Draco 圧縮拡張にも対応しています。

まずは、three.js と GLTFLoader.js を HTML に読み込みましょう。

three.js をダウンロードし展開すると、

- three.js 本体は、 build/three.js
- GLTFLoader.js は、 examples/js/loaders/GLTFLoader.js

からそれぞれ入手することができます。この 2 つを読み込みます。

```
<script src="three.js"></script>
<script src="GLTFLoader.js"></script>
```

#### FYI: webpack などのバンドラーを利用している場合

three.js 本体はモジュール化されており、 import で取り込むことができます。一方で、 GLTFLoader.js などの本体から外れるプラグインは、2018 年 6 月現在ではモジュール化されていません。そのため、 THREE をグローバルから参照できるようにしたり、また、 webpack であれば import-loader、 export-loader を利用するなどの工夫が必要です。

なお、 three.js の examples 内の JS ファイルの ES モジュール化は長い間検討され続けています。

<https://github.com/pixgrid/codagrid-draft/pull/1288/files>

```
const THREE = window.THREE = require( 'three' );
require( 'three/examples/js/loaders/GLTFLoader.js' );

console.log( THREE.GLTFLoader );
```

```
import * as THREE from 'THREE';
THREE.GLTFLoader = require( 'imports-loader?THREE!exports-
loader?THREE.GLTFLoader!../node_modules/three/examples/js/loaders/GLTFLoader.js' );

console.log( THREE.GLTFLoader );
```

ライブラリーが用意できたら `THREE.GLTFLoader` を使って glTF ファイルを読みましょう。

まずは、three.js のレンダラー、カメラ、シーンを用意しておきます。この時点では何も表示されません。

```
const width = window.innerWidth;
const height = window.innerHeight;

const scene = new THREE.Scene();
const camera = new THREE.PerspectiveCamera( 45, width / height, 0.001, 100 );
camera.position.set( 0, 0, 0.5 );
const renderer = new THREE.WebGLRenderer( { antialias: true } );
renderer.setSize( width, height );
renderer.gammaInput = true;
renderer.gammaOutput = true;
document.body.appendChild( renderer.domElement );

scene.add( new THREE.HemisphereLight( 0xfffffff , 0x332222 ) );

// ここに GLTFLoader を追加していく
// (予定地)

(function anim () {

    requestAnimationFrame( anim );
    renderer.render( scene, camera );
})();
```

つぎに、`GLTFLoader` を利用してみましょう。`new GLTFLoader()` でローダーインスタンスを生成します。glTF ファイルをロードするには、ローダーインスタンスの `load` メソッドを利用します。第一引数に glTF ファイルのパスを、第二引数にはロード後のコールバックを渡します。

第一引数に glTF ファイルのパスは、glTF ファイルと glb ファイルのどちらの形式にも対応しています。

`load` メソッドの内部では XHR (Ajax) が利用されています。ですのでコードはサーバー上で実行する必要があります。ローカルでは Ajax は制限されるので試す際には注意してください。ローカルサーバーの利用については、巻末の「ローカルサーバーを利用する」を参考にしてください。

また、内部で ES6 の Promise が利用されています。ですので、IE11 向けには Promise の Polyfill を利用する必要があります。

<https://github.com/stefanpenner/es6-promise>

```

// ここに GLTFLoader を追加していく
const loader = new THREE.GLTFLoader();

loader.load(
    // 読み込む glTF ファイルへのパス
    './model/barger/barger.gltf',
    // 読み込み (Ajax) 完了後のコールバック
    function ( gltf ) {

        scene.add( gltf.scene );

    }
);

```

demo: examples/1.html

ここまでコードを組み立てると全体は以下となります。

```

const width  = window.innerWidth;
const height = window.innerHeight;

const scene  = new THREE.Scene();
const camera = new THREE.PerspectiveCamera( 45, width / height, 0.001, 100 );
camera.position.set( 0, 0, 0.5 );
const renderer = new THREE.WebGLRenderer( { antialias: true } );
renderer.setSize( width, height );
renderer.gammaInput = true;
renderer.gammaOutput = true;
document.body.appendChild( renderer.domElement );

scene.add( new THREE.HemisphereLight( 0xfffffff , 0x332222 ) );

// ここに GLTFLoader を追加していく
const loader = new THREE.GLTFLoader();

loader.load(
    // 読み込む glTF ファイルへのパス
    './model/barger/barger.gltf',
    // 読み込み (Ajax) 完了後のコールバック
    function ( gltf ) {

        scene.add( gltf.scene );

    }
);
(function anim () {

    requestAnimationFrame( anim );
    renderer.render( scene, camera );
})();

```

## GLTFLoader のその他の機能

loader インスタンスには、第三引数にロード進行中、第四引数にロード失敗時のコールバックも用意されています。また、第二引数、つまりロード完了後のコールバックに渡される gltf オブジェクトは、以下の 5 つの要素で構成されています。

プロパティ	意味
gltf.animations	<THREE.AnimationClip>の配列
gltf.scene	THREE.Scene インスタンス（グルーピングされたオブジェクト群）
gltf.scenes	THREE.Scene インスタンスの配列（シーンが複数格納されている場合）
gltf.cameras	THREE.Camera インスタンスの配列（カメラが複数格納されている場合）
gltf.asset	ファイルのメタ情報（glTF のバージョンなど）

```
const loader = new THREE.GLTFLoader();

loader.load(
    './model/barge.gltf',
    function ( gltf ) {

        scene.add( gltf.asset );

        gltf.animations; // Array<THREE.AnimationClip>
        gltf.scene; // THREE.Scene
        gltf.scenes; // Array<THREE.Scene>
        gltf.cameras; // Array<THREE.Camera>
        gltf.asset; // Object

    },
    // ロードの進行度を取得することができます
    function ( xhr ) {

        console.log( ( xhr.loaded / xhr.total * 100 ) + '% のロード完了' );

    },
    // エラーが発生した場合のコールバック
    function ( error ) {

        console.log( '読み込みに失敗しました' );

    }
);
```

## three.js の旧 Blender Exporter

従来 three.js 用の 3D モデル形式では、three.js オフィシャルの Blender Exporter で出力した JSON ファイルがよく使われていました。しかし、three.js r93 のリリース（2018 年 5 月 31 日）で three.js のリポジトリから Blender Exporter が削除されました。その理由は、Blender Exporter 開発者が three.js コミュニティーから去ってしまい、数年間メンテナンスされておらず、また将来的にも積み重なった問題の解決や新機能対応の見込みがなかったためです。

この決定は「Remove Blender exporter, add 'Loading 3D models' guide.」での会話できました。

一方で、今後 three.js での 3D モデルの読み込みには glTF を利用することが勧められています。

<https://github.com/mrdoob/three.js/pull/14117>



## アニメーションつきの glTF: スケルタルアニメーション

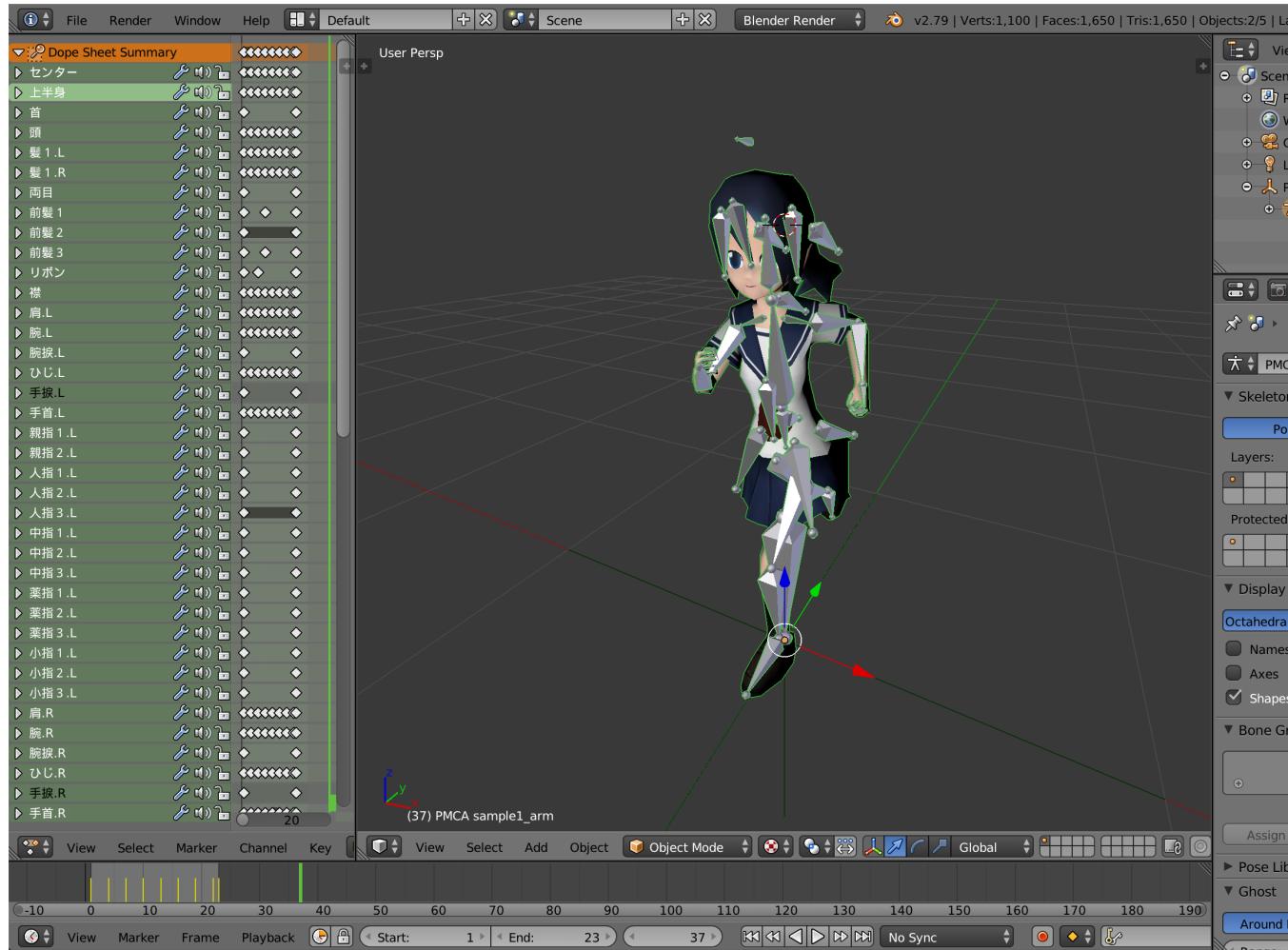
glTF はアニメーションもサポートしています。スケルタルアニメーションが付与された 3D モデルを glTF で出力し、three.js で表示してみましょう。

練習用ファイルとして mobko-bake-run.blend を用意しています。手元にスケルタルアニメーション付きの Blender ファイルがあればそれを使ってもいいでしょう。

操作の練習用の Blender 用のモデルファイル

examples/blend/mobko-bake-run/mobko-bake-run.blend

なお、執筆時点では glTF-Blender-Exporter は、複数のスケルタルアニメーションに対応しておらず、1つしか出力されません。複数のアニメーションは今後のアップデートが期待されます。



スケルタルアニメーションが付与された Blender モデル

スケルタルアニメーション付きの3Dモデルでも、glTF出力時に初期設定のままでアニメーション付きのglTFファイルを出力することができます。出力設定の項目は多いですが、なにもせず「Export glTF 2.0」ボタンを押して出力しましょう。

three.jsで読み込んだ際に、アニメーションの再生を行えば、Blenderで用意したアニメーションがWebGL上で再現できます。

アニメーション再生のためのコードを見てみましょう。



three.js の基本コードを用意します。今回は身長 1.5 メートルの人物を表示するので、カメラは 3 メートル手前に引き、1 メートルの高さに配置しています。

```
const width  = window.innerWidth;
const height = window.innerHeight;

const scene  = new THREE.Scene();
const camera = new THREE.PerspectiveCamera( 45, width / height, 0.001, 100 );
camera.position.set( 0, 1, 3 );
const renderer = new THREE.WebGLRenderer( { antialias: true } );
renderer.setSize( width, height );
renderer.gammaInput = true;
renderer.gammaOutput = true;
document.body.appendChild( renderer.domElement );

scene.add( new THREE.HemisphereLight( 0xffffff , 0x332222 ) );

// ここに GLTFLoader を追加していく
// 予定地

( function anim () {

    requestAnimationFrame( anim );
    renderer.render( scene, camera );

} )();
```

上記コードの「ここに GLTFLoader を追加していく」の部分に必要なコードを追加します。アニメーション再生のために、経過時間の管理用に `THREE.Clock` インスタンスを用意します。また、アニメーションプレイヤー用の変数を `mixer` という名前で用意しておきます。

```
// ここに GLTFLoader を追加していく
const clock  = new THREE.Clock();
const loader = new THREE.GLTFLoader();
let mixer;
```

glTF ファイルのロードが完了すると、`gltf` オブジェクトには `animations` という配列が格納されています。`animations` には、Blender 上でのキーフレームアニメーションが格納されています。歩行アニメーション、ジャンプアニメーションなどといった、複数種類のアニメーションを扱うことができる前提ですが、残念ながら前述の通り、現状の「glTF-Blender-Exporter」は複数アニメーションの出力には対応できません。

```
// ここに GLTFLoader を追加していく
const clock = new THREE.Clock();
const loader = new THREE.GLTFLoader();
let mixer;

loader.load(
    // 読み込む glTF ファイルへのパス
    './model/mobko/mobko-bake-run.gltf',
    // 読み込み (Ajax) 完了後のコールバック
    function ( gltf ) {

        scene.add( gltf.scene );

        // gltf オブジェクト用のアニメーションプレイヤーを作る
        mixer = new THREE.AnimationMixer( gltf.scene );

        // アニメーションプレイヤーにアニメーションを登録する。
        // Blender でのアニメーションが gltf.animations に配列で格納されている
        // ただし、現時点では 1 つ (0 番目) しか出力されない。
        const runAnimation = mixer.clipAction( gltf.animations[ 0 ] );

        // アニメーションプレイヤー (mixer) に登録されたアニメーションを再生する
        runAnimation.play();

    }
);
```

アニメーションループの関数内に、アニメーションプレイヤーの更新を追加します。変数 `delta` に格納されたデルタ時間は、前回のレンダリング時からの差分時間が格納されます。多くの場合、約 0.16 秒、つまり 1 フレーム分となります。このデルタ時間をアニメーションプレイヤーの更新に利用します。

アニメーションプレイヤーは glTF 読み込みの非同期で作成されていますので、`if ( mixer )` で、作成済みの場合のみ実行しています。

```
( function anim () {  
    requestAnimationFrame( anim );  
  
    const delta = clock.getDelta();  
  
    // アニメーションプレイヤーの時間を送る  
    if ( mixer ) mixer.update( delta );  
  
    renderer.render( scene, camera );  
} )();
```

demo: examples/3.html

ここまでコードを組み立てると全体は次のようにになります。

```

const width  = window.innerWidth;
const height = window.innerHeight;

const scene  = new THREE.Scene();
const camera = new THREE.PerspectiveCamera( 45, width / height, 0.001, 100 );
camera.position.set( 0, 1, 3 );
const renderer = new THREE.WebGLRenderer( { antialias: true } );
renderer.setSize( width, height );
renderer.gammaInput = true;
renderer.gammaOutput = true;
document.body.appendChild( renderer.domElement );

scene.add( new THREE.HemisphereLight( 0xfffffff , 0x332222 ) );

// ここに GLTFLoader を追加していく
const clock  = new THREE.Clock();
const loader = new THREE.GLTFLoader();
let mixer;

loader.load(
    // 読み込む glTF ファイルへのパス
    './model/mobko/mobko-bake-run.gltf',
    // 読み込み (Ajax) 完了後のコールバック
    function ( gltf ) {

        scene.add( gltf.scene );

        // gltf オブジェクト用のアニメーションプレイヤーを作る
        mixer = new THREE.AnimationMixer( gltf.scene );

        // アニメーションプレイヤーにアニメーションを登録する。
        // Blender でのアニメーションが gltf.animations に配列で格納されている
        // ただし、現時点では 1 つ (0 番目) しか出力されない。
        const runAnimation = mixer.clipAction( gltf.animations[ 0 ] );

        // アニメーションプレイヤー (mixer) に登録されたアニメーションを再生する
        runAnimation.play();

    }
);

( function anim () {

    requestAnimationFrame( anim );

    const delta = clock.getDelta();

    // アニメーションプレイヤーの時間を送る
    if ( mixer ) mixer.update( delta );

    renderer.render( scene, camera );

} )();

```

## アニメーションつきの glTF: モーフアニメーション

glTF はモーフアニメーションにも対応しています。モーフアニメーションはシェイプアニメーションとも呼ばれます。変形前と変形後の 2 つのメッシュを用意し、「どちらにどれだけ近づけるか」でアニメーションを実現します。ボーンでは表せないような複雑な場面で使われることが多い、キャラクターの表情や、体格（太らせたり痩せさせたり）などに利用されます。

glTF-Blender-Exporter はモーフアニメーション出力にも対応しており、Blender 上での Shape Key がモーフアニメーションとして glTF に埋め込まれます。モーフアニメーションが付与された 3D モデルを three.js で表示してみましょう。

練習用ファイルとして monster.blend を用意しています。手元にシェイプキー付きの Blender ファイルがあればそれを使ってもいいでしょう。

操作の練習用の Blender 用のモデルファイル

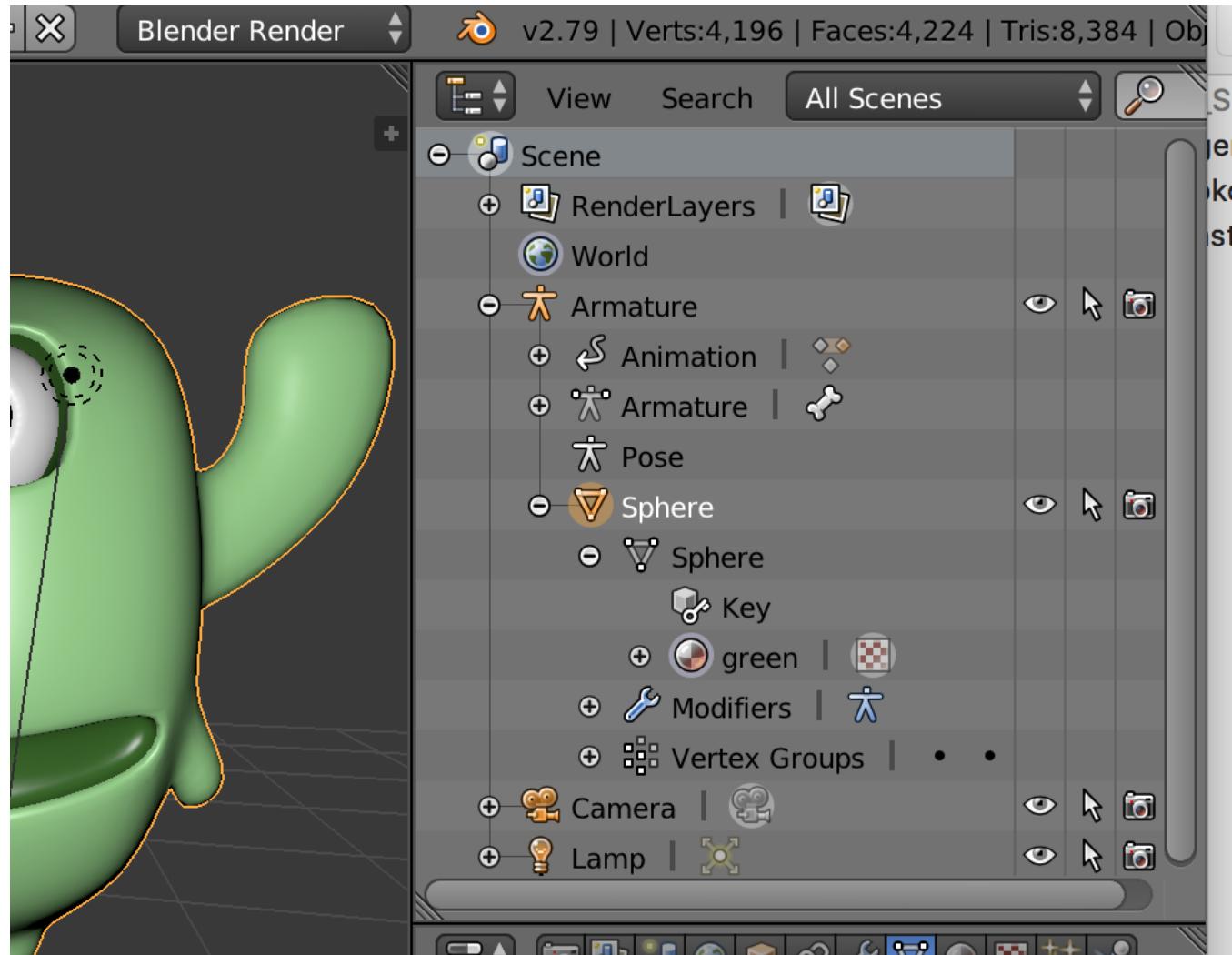
examples/blend/monster/monster.blend

monster.blend には「まばたき」「口の開閉」「口の角度」の 3 つのシェイプキーが登録されています。



シェイプキーが付与された Blender モデル

Blender 上での構造は、シーン内にアーマチュア（骨格）がありその中にモデル本体が格納されています。この構造はプログラミング時に必要になります。



Blender の Outliner パネル

では早速コードを書いていきましょう。まずは three.js の基本コードを用意します。

```

const width = window.innerWidth;
const height = window.innerHeight;

const scene = new THREE.Scene();
const camera = new THREE.PerspectiveCamera( 45, width / height, 0.001, 100 );
camera.position.set( 0, 1, 3 );
const renderer = new THREE.WebGLRenderer( { antialias: true } );
renderer.setSize( width, height );
renderer.gammaInput = true;
renderer.gammaOutput = true;
document.body.appendChild( renderer.domElement );

scene.add( new THREE.HemisphereLight( 0xffffff , 0x332222 ) );

// ここに GLTFLoader を追加していく

( function anim () {

    requestAnimationFrame( anim );
    renderer.render( scene, camera );

} )();

```

`GLTFLoader` のローダーインスタンスを作成し、glTF ファイルを読み込みます。その際に、シェイプキーを有しているメッシュ本体を参照できるようにしておきます。メッシュ本体は `gltf.scene` に含まれており、Blender 上での構造と同じです。ここでは `monsterMesh` という名前で参照できるようにしています。

```

// ここに GLTFLoader を追加していく
const loader = new THREE.GLTFLoader();
let monsterMesh;

loader.load(
    // 読み込む glTF ファイルへのパス
    './model/monster/monster.gltf',
    // 読み込み (Ajax) 完了後のコールバック
    function ( gltf ) {

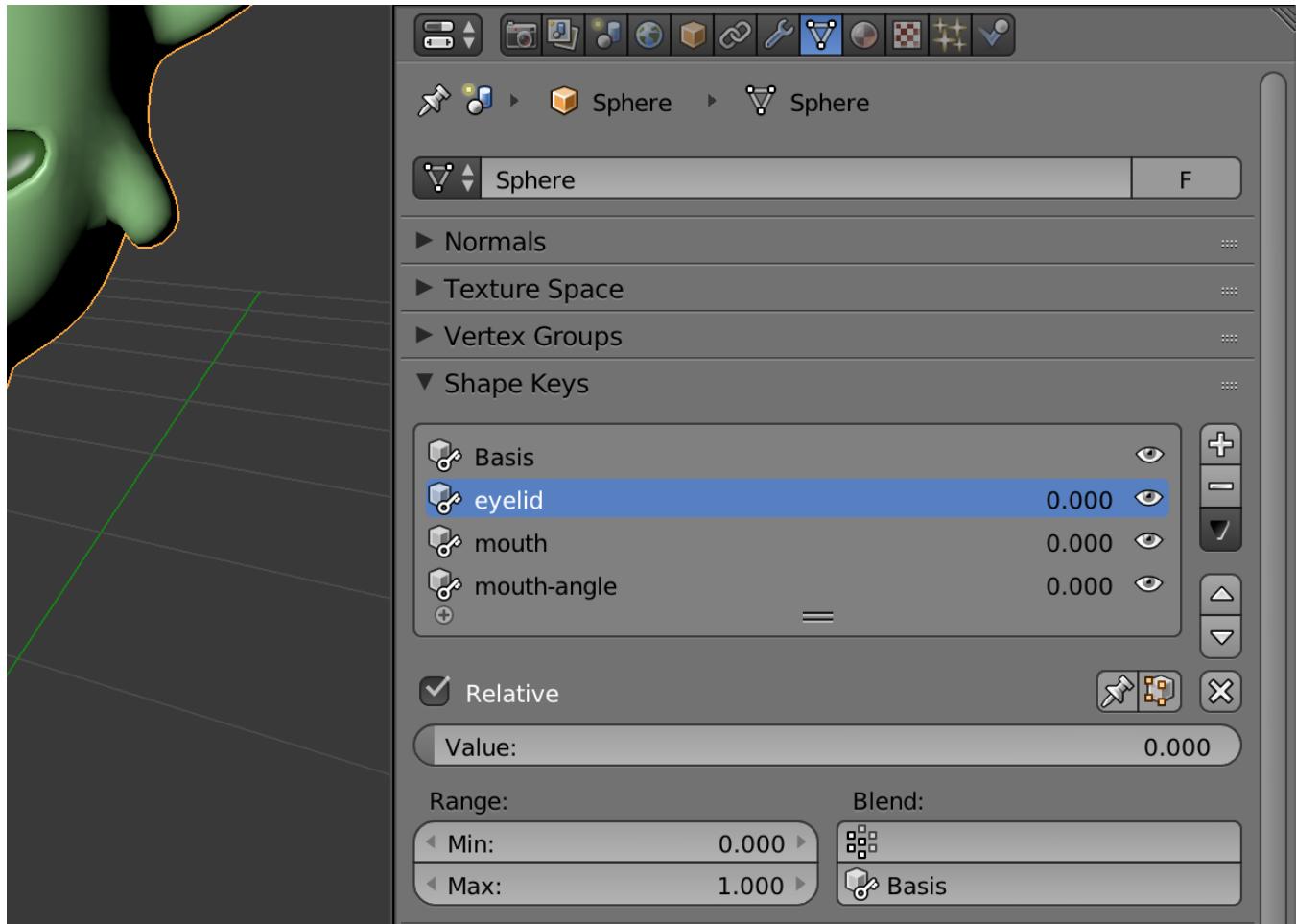
        // シェイプキーが付与されたメッシュを「monsterMesh」として保持する
        // gltf 内のシーンの構造は Blender 上での構造と同様になっており、
        // gltf.scene.children は Blender でもともと光源だったオブジェクトなども含まれている。
        const armature = gltf.scene.children[ 0 ];
        monsterMesh = armature.children[ 0 ];

        scene.add( gltf.scene );

    }
);

```

シェイプキーを有しているメッシュには配列で `morphTargetInfluences` というプロパティが存在します。この配列の要素数と順序は Blender 上でのシェイプキーパネルと同一です。ここで例では配列の 0 番目は「eyelid（まばたき）」、1 番目は「mouth（口の開閉）」、2 番目は「mouth-angle（口の角度）」です。この値を変更すると表情を変形することができます。値は 0 から 1 の間の小数点数で指定します。



Blender 上でのシェイプキーパネル

```

function morph0( value ) {

    monsterMesh.morphTargetInfluences[ 0 ] = value; // 0 から 1 の間

}

function morph1( value ) {

    monsterMesh.morphTargetInfluences[ 1 ] = value;

}

function morph2( value ) {

    monsterMesh.morphTargetInfluences[ 2 ] = value;

}

```

demo: examples/4.html

ここまでコードを組み立てると全体は以下となります。

```

const width  = window.innerWidth;
const height = window.innerHeight;

const scene  = new THREE.Scene();
const camera = new THREE.PerspectiveCamera( 45, width / height, 0.001, 100 );
camera.position.set( 0, 1, 3 );
const renderer = new THREE.WebGLRenderer( { antialias: true } );
renderer.setSize( width, height );
renderer.gammaInput = true;
renderer.gammaOutput = true;
document.body.appendChild( renderer.domElement );

scene.add( new THREE.HemisphereLight( 0xffffff , 0x332222 ) );

// ここに GLTFLoader を追加していく
const loader = new THREE.GLTFLoader();
let monsterMesh;

loader.load(
    // 読み込む glTF ファイルへのパス
    './model/monster/monster.gltf',
    // 読み込み (Ajax) 完了後のコールバック
    function ( gltf ) {

        // シエイプキーが付与されたメッシュを「monsterMesh」として保持する
        // gltf 内のシーンの構造は Blender 上での構造と同様になっており、
        // gltf.scene.children は Blender でもともと光源だったオブジェクトなども含まれている。
        const armature = gltf.scene.children[ 0 ];
        monsterMesh = armature.children[ 0 ];

        scene.add( gltf.scene );
    }
);

```

```
        }
    );
    ( function anim () {
        requestAnimationFrame( anim );
        renderer.render( scene, camera );
    } )();
    function morph0( value ) {
        monsterMesh.morphTargetInfluences[ 0 ] = value;
    }
    function morph1( value ) {
        monsterMesh.morphTargetInfluences[ 1 ] = value;
    }
    function morph2( value ) {
        monsterMesh.morphTargetInfluences[ 2 ] = value;
    }
}
```

スケルタルアニメーションとモーフアニメーションは共存することもできます。以下はスケルタルアニメーションとモーフアニメーションを同時にコントールする例です。

```
const width = window.innerWidth;
const height = window.innerHeight;

const scene = new THREE.Scene();
const camera = new THREE.PerspectiveCamera( 45, width / height, 0.001, 100 );
camera.position.set( 0, 1, 3 );
const renderer = new THREE.WebGLRenderer( { antialias: true } );
renderer.setSize( width, height );
renderer.gammaInput = true;
renderer.gammaOutput = true;
document.body.appendChild( renderer.domElement );

scene.add( new THREE.HemisphereLight( 0xffffff , 0x332222 ) );

// ここに GLTFLoader を追加していく
const clock = new THREE.Clock();
const loader = new THREE.GLTFLoader();
let mixer;
let monsterMesh;

loader.load(
    // 読み込む glTF ファイルへのパス
    './model/monster/monster.gltf',
    // 読み込み (Ajax) 完了後のコールバック
    function ( gltf ) {

        // シエイプキーが付与されたメッシュを「monsterMesh」として保持する
        // gltf 内のシーンの構造は Blender 上での構造と同様になっており、
        // gltf.scene.children には Blender でもともと光源だったオブジェクトなども含まれている。
        const armature = gltf.scene.children[ 0 ];
        monsterMesh = armature.children[ 0 ];

        scene.add( gltf.scene );

        mixer = new THREE.AnimationMixer( gltf.scene );
        const runAnimation = mixer.clipAction( gltf.animations[ 0 ] );
        runAnimation.play();

    }
);

(function anim () {

    requestAnimationFrame( anim );

    const delta = clock.getDelta();

    // アニメーションプレイヤーの時間を送る
    if ( mixer ) mixer.update( delta );
});
```

```
    renderer.render( scene, camera );  
} );  
  
function morph0( value ) {  
    monsterMesh.morphTargetInfluences[ 0 ] = value;  
}  
  
function morph1( value ) {  
    monsterMesh.morphTargetInfluences[ 1 ] = value;  
}  
  
function morph2( value ) {  
    monsterMesh.morphTargetInfluences[ 2 ] = value;  
}
```

左右に揺れているアニメーションはボーンによりコントロールされたスケルタルアニメーションです。これに加えて、表情をモーフアニメーションにてコントロールすることができます。

demo: examples/5.html

# Draco 壓縮対応の glTF

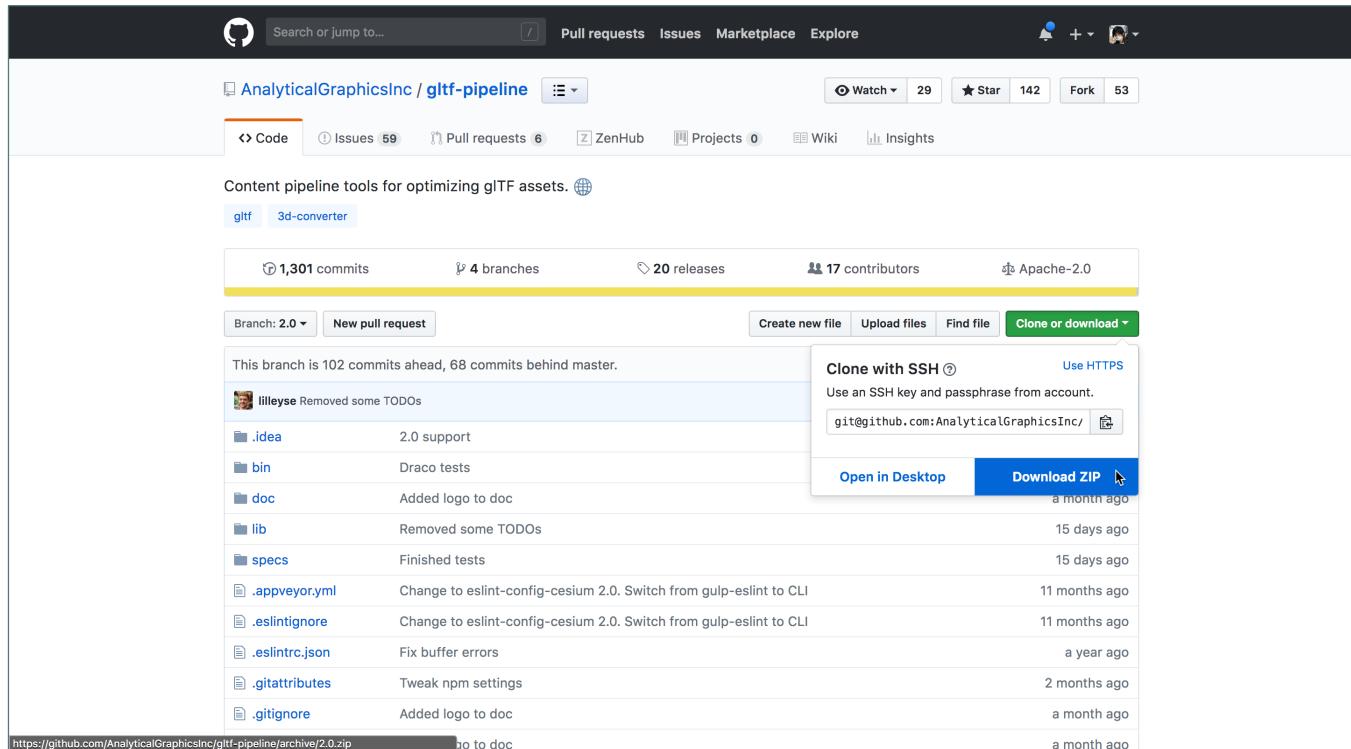
glTF には本来の標準仕様に加えて、独自機能を拡張する領域が設けられています。Draco 壓縮は glTF の標準仕様には含まれていませんが、独自拡張「KHR\_draco\_mesh\_compression」として利用することができます。なお、KHR 接頭辞はクロノス（Khronos）により用意されたことを意味します。

Draco 壓縮をすると、glTF ファイルの bin 部分のファイルサイズを大幅に圧縮することができます。

gltf-pipeline というコマンドラインツールを使うと、glTF ファイルを Draco 壓縮することができます。また、three.js では KHR\_draco\_mesh\_compression の展開にも対応しています。

gltf-pipeline と three.js を使い、glTF の Draco 壓縮をし、Draco 壓縮された glTF ファイルを Web ブラウザーで表示してみましょう。

まずは gltf-pipeline をダウンロード（<https://github.com/AnalyticalGraphicsInc/gltf-pipeline/tree/2.0>）します。本稿執筆時点の 2018 年 6 月現在では、「2.0」ブランチを利用します。



ダウンロードが完了したら、コマンドラインで gltf-pipeline フォルダーへ移動します。

```
cd path/to/gltf-pipeline-2.0
```

npm を使い、gltf-pipeline フォルダーで、追加パッケージをインストールします。

```
$ npm install
```

これで準備は完了です。node.js で「./bin/gltf-pipeline.js」を利用し、入力ファイルと出力ファイル名を指定すれば、glTF ファイルの bin 部分を Draco 圧縮することができます。

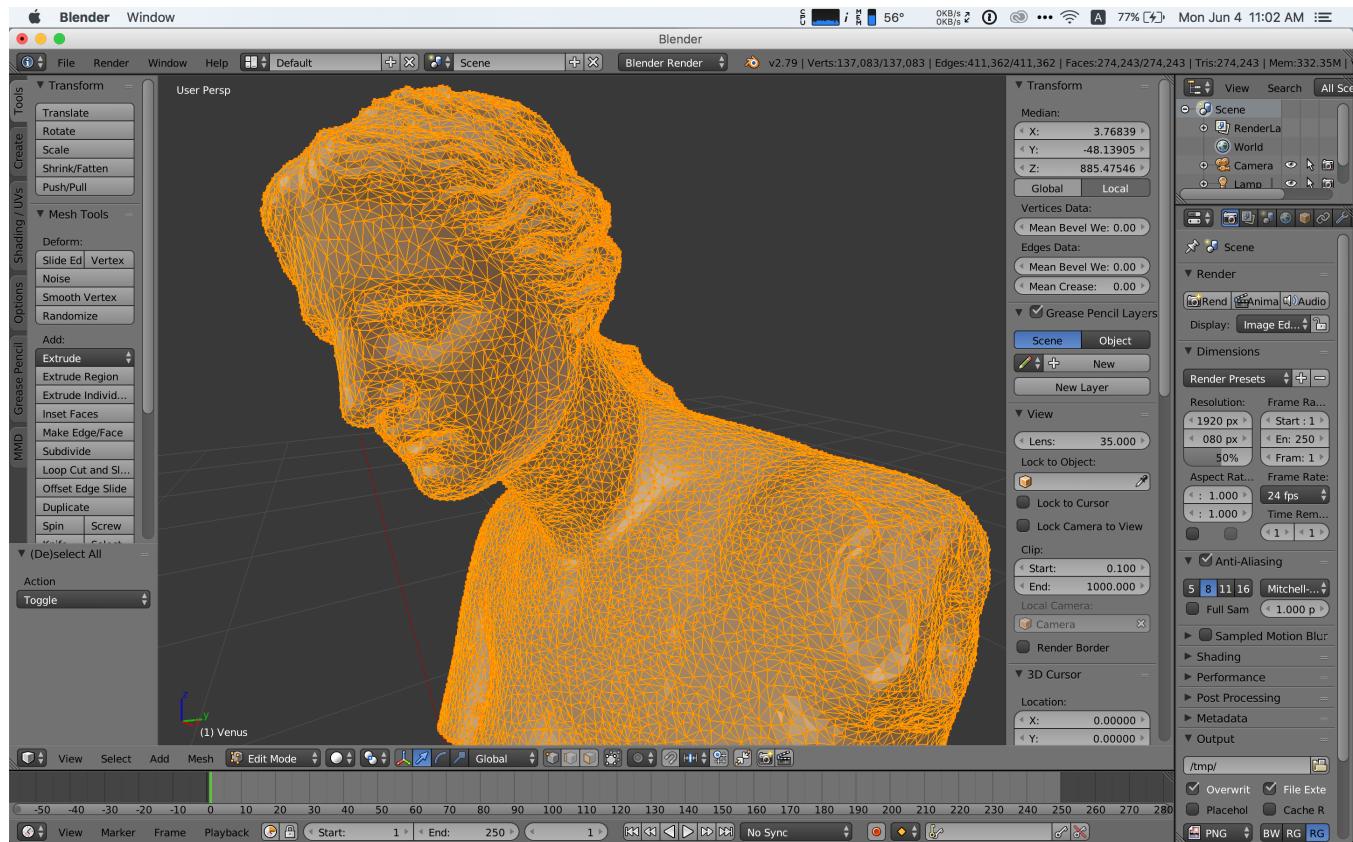
```
node ./bin/gltf-pipeline.js -i ../model.gltf -d -s -o ../modelDraco.gltf
```

各オプションの意味は以下のとおりです。

- `-i` はインプットするファイルのパス
- `-d` は Draco 圧縮の有効化
- `-s` はセパレート（関連アセットの外部化）
- `-o` はアウトプットファイルのパスと名前

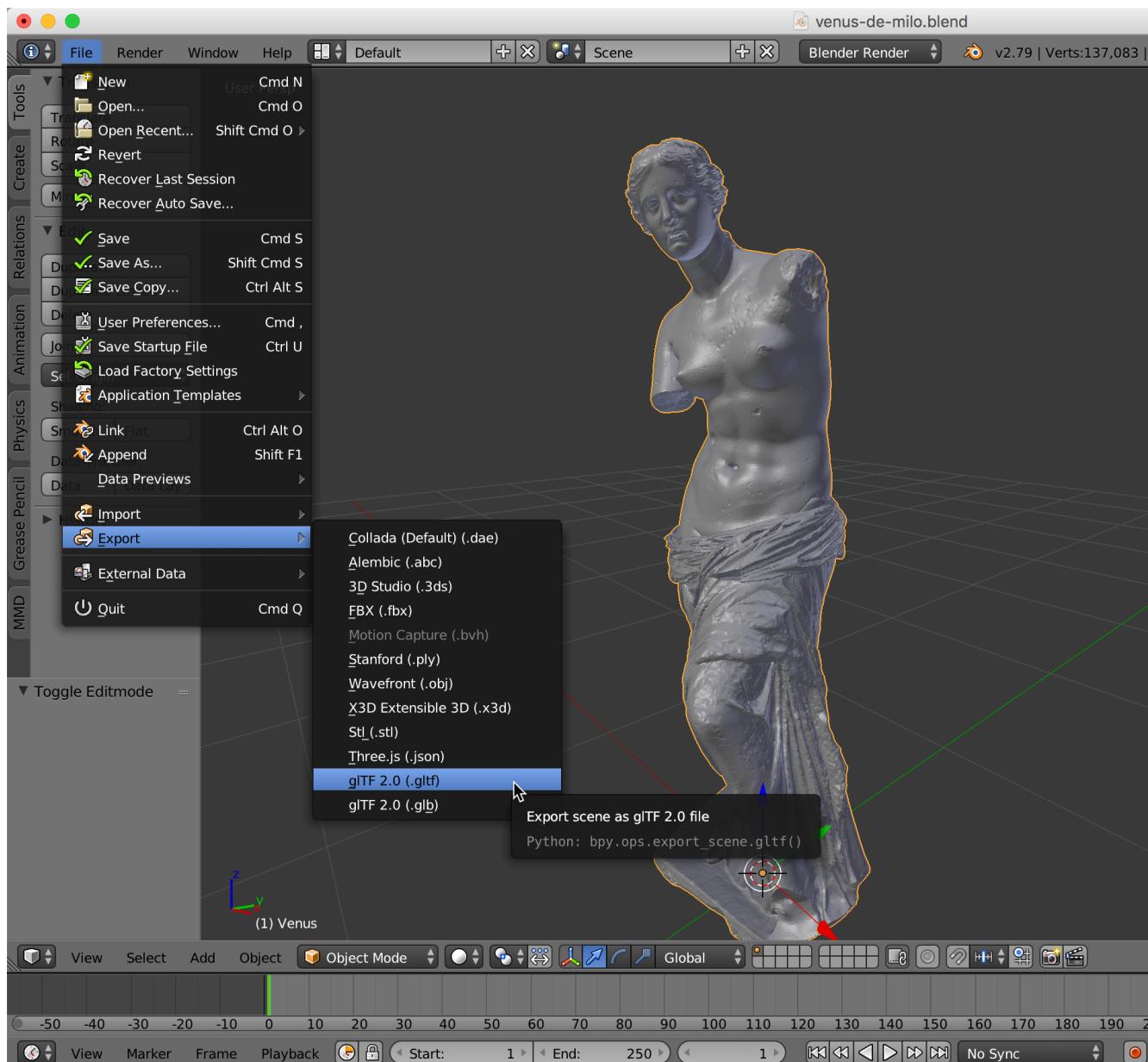
それでは、実際に 3D モデルファイルを Draco 圧縮してみましょう。

今回は3Dスキャンされたミロのビーナス像を3Dモデルの例に利用します。3Dスキャンされたモデルは多くの場合、頂点が最適化されておらず、頂点数が非常に多いためファイル容量が大きくなりがちです。



ブレンダーに読み込んだ3Dスキャンされたミロのビーナス像

3Dスキャンされたミロのビーナス像は約13万頂点で構成されています。これを「venus-de-milo.gltf」という名前でglTFファイルに出力してみます。



エクスポート

glTF の頂点情報のファイルである bin 部分が 23MB になってしましました！

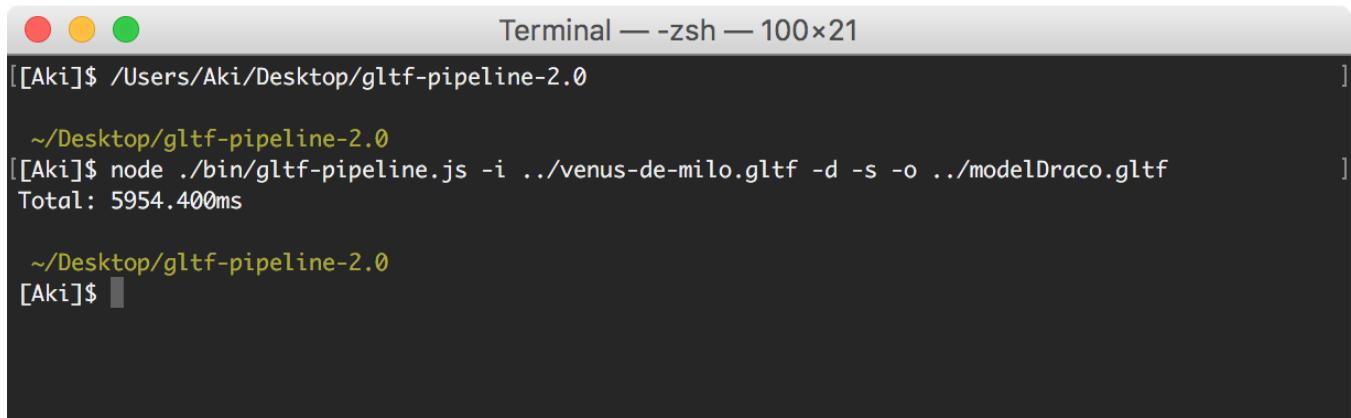


venus-de-milo.gltf のファイル容量

「venus-de-milo.gltf」を Draco 壓縮してみましょう。「venus-de-milo.gltf」を入力ファイルとして指定し、出力ファイル名を「venus-de-milo-draco.gltf」としています。

```
cd path/to/gltf-pipeline-2.0
```

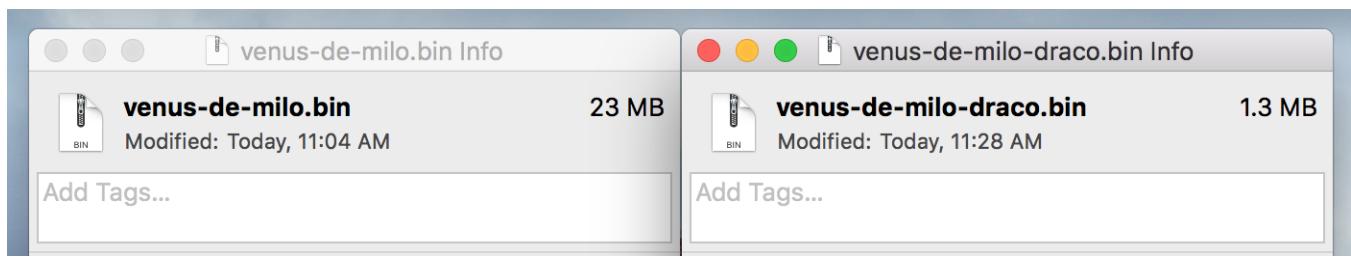
```
node ./bin/gltf-pipeline.js -i ../venus-de-milo.gltf -d -s -o ../venus-de-milo-draco.gltf
```



```
Terminal — -zsh — 100x21
[[Aki]$ /Users/Aki/Desktop/gltf-pipeline-2.0
~/Desktop/gltf-pipeline-2.0
[[Aki]$ node ./bin/gltf-pipeline.js -i ../venus-de-milo.gltf -d -s -o ../modelDraco.gltf
Total: 5954.400ms
~/Desktop/gltf-pipeline-2.0
[Aki]$ ]]
```

コマンドラインでの圧縮

圧縮には少し時間がかかります。圧縮が完了すると新たに「venus-de-milo-draco.gltf」と「venus-de-milo-draco.bin」が生成されます。結果は...なんということでしょう！bin 部分は、圧縮前が 23MB だったのに対し、圧縮後は 1.3MB になりました！



ファイル容量の比較

Draco は可逆圧縮で、復元が可能です。three.js で Draco 圧縮された glTF ファイルを表示してみましょう。

Draco 圧縮された glTF ファイルの読み込みには、追加で以下のライブラリーが必要です。ダウンロードした three.js の中から以下を取り出します。

- three.js-master/examples/js/loaders/DRACOLoader.js
- three.js-master/examples/js/libs/draco/gltf/ フォルダー一式

そして、DRACOLoader.js を読み込みます。「draco/gltf」フォルダー一式は、WebWorker 用のライブラリーで、JavaScript コード内から読み込みます。

```
<script src="./three.min.js"></script>
<script src="./GLTFLoader.js"></script>
<script src="./DRACOLoader.js"></script>
```

まずは three.js の基本コードを用意します。

```
const width  = window.innerWidth;
const height = window.innerHeight;

const scene  = new THREE.Scene();
const camera = new THREE.PerspectiveCamera( 45, width / height, 0.001, 100 );
camera.position.set( 0, 1, 3 );
const renderer = new THREE.WebGLRenderer( { antialias: true } );
renderer.setSize( width, height );
renderer.gammaInput = true;
renderer.gammaOutput = true;
document.body.appendChild( renderer.domElement );

scene.add( new THREE.HemisphereLight( 0xfffffff, 0x332222 ) );

// ここに GLTFLoader を追加していく
// 予定地

( function anim () {

    requestAnimationFrame( anim );
    renderer.render( scene, camera );

} )();
```

これまでと同様、`GLTFLoader` インスタンスを作ります。それに加えて、`DRACOLoader` インスタンスも作ります。`DRACOLoader` は WebWorker 経由でデコーダーライブラリーを利用します。そのため、デコーダーライブラリーのパスを設定する必要があります。

three.js-master/examples/js/libs/draco/gltf/ から取り出してコピーしたディレクトリーを指定します。パスは必ず「/」で終わるようにしてください。

そして、作成した `DRACOLoader` インスタンスを `GLTFLoader` インスタンスに埋め込みます。

```
// ここに GLTFLoader を追加していく
const loader = new THREE.GLTFLoader();

THREE.DRACOLoader.setDecoderPath( './js/libs/draco/gltf/' );
const dracoLoader = new THREE.DRACOLoader();
loader.setDRACOLoader( dracoLoader );
```

あとは、これまで通り、Draco圧縮された glTF ファイルを読み込むだけです。

DRACOLoader のデコーダーは、C 言語から生成された JavaScript で、ガベージコレクトがされません。メモリーの開放は手動で行う必要があります。そのためには  
THREE.DRACOLoader.releaseDecoderModule() を実行します。

```
// ここに GLTFLoader を追加していく
const loader = new THREE.GLTFLoader();

THREE.DRACOLoader.setDecoderPath( './js/libs/draco/gltf/' );
const dracoLoader = new THREE.DRACOLoader();
loader.setDRACOLoader( dracoLoader );

loader.load(
    // 読み込む glTF ファイルへのパス
    './model/venus-de-milo/venus-de-milo-draco.gltf',
    // 読み込み (Ajax) 完了後のコールバック
    function ( gltf ) {

        scene.add( gltf.scene );

        // https://github.com/google/draco/issues/349
        THREE.DRACOLoader.releaseDecoderModule();

    }
);
```

これで Draco 圧縮された glTF モデルを表示することができました！

example/6.html (カメラを回転する処理を加えています)

ここまでコードを組み立てると全体は以下となります。

```
const width = window.innerWidth;
const height = window.innerHeight;

const scene = new THREE.Scene();
const camera = new THREE.PerspectiveCamera( 45, width / height, 0.001, 100 );
camera.position.set( 0, 1, 3 );
const renderer = new THREE.WebGLRenderer( { antialias: true } );
renderer.setSize( width, height );
renderer.gammaInput = true;
renderer.gammaOutput = true;
document.body.appendChild( renderer.domElement );

scene.add( new THREE.HemisphereLight( 0xffffff, 0x332222 ) );

// ここに GLTFLoader を追加していく
THREE.DRACOLoader.setDecoderPath( './js/libs/draco/gltf/' );
const dracoLoader = new THREE.DRACOLoader();
loader.setDRACOLoader( dracoLoader );

loader.load(
    // 読み込む glTF ファイルへのパス
    './model/venus-de-milo/venus-de-milo-draco.gltf',
    // 読み込み (Ajax) 完了後のコールバック
    function ( gltf ) {

        scene.add( gltf.scene );

        // https://github.com/google/draco/issues/349
        THREE.DRACOLoader.releaseDecoderModule();

    }
);

( function anim () {

    requestAnimationFrame( anim );
    renderer.render( scene, camera );

} )();
```

注意: Draco のデコーダーは 1.5MB (JS バージョンの場合) ほどあります。Draco 圧縮をしても削減できる容量が 1.5MB 以上を見込めないのであれば、かえってロード時間が増えててしまいます。Draco の機能は特に容量の大きなファイルにのみ、利用するといいでしょう。

## さいごに

glTF が登場するまで、公式にインターネットでの配信前提の WebGL 用のモデルファイル形式はありませんでした。

glTF 登場してから歴史は長くありませんが（といっても、前身の COLLADA の歴史はあります）、すでに出力ツールが整っており、また three.js や babylon.js といった多くのライブラリーで読み込むことができます。スケルタルアニメーションが1つのみといった、ほんの少しの問題もありますが、プロダクトコードでも利用していいくらいに安定していると言えます。

Web で 3D モデルを扱う機会があれば、glTF を活用してみるといいでしょう。

## Appendix 1: Blender の基本操作

今回は three.js の講義ですので、詳細な使い方については割愛し、ここでは基本操作のみの説明を行います。

いきなり何かモデルをつくる、というのはなかなか大変なものです。今回はさわりとして、すでに用意されたモデルを操作してみましょう。

配布したデータの models/barge 内に barge.blend を用意しています。この 3D モデルを開き、3D ビューで様々な方向から眺めたり、拡大したりして blender の操作感を体験してみましょう。

もっと他のモデルも試したい場合は、Blend Swap (<https://www.blendswap.com/>) からさまざまな 3D モデル入手することもできます。（3D モデルの著作権に気をつけて利用しましょう）

なお、Blender 上での 1 単位は three.js でも 1 単位として扱われます。適切なカメラの位置の位置はこの単位をもとに計算するといいでしよう。

一般的に、ゲームや物理エンジンなどでは 1 単位を 1 メートルとして扱います。3D モデルを用意する際もこれに習い、1 単位を 1 メートルとしておくと、他の物体等との比較がしやすくなります。

## ビューの基本操作

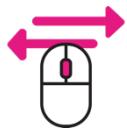
操作

Windows

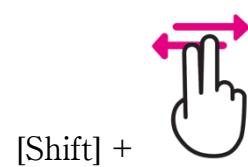
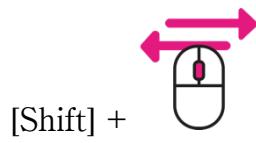
macOS

---

回転



横移動



拡大/縮小



選択



## オブジェクト選択後の基本操作

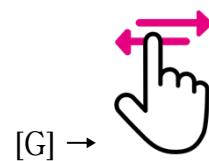
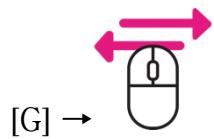
操作

Windows

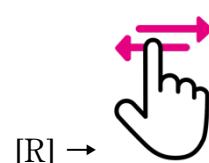
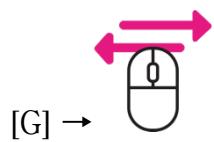
macOS

---

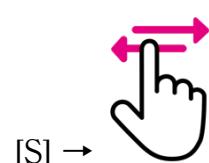
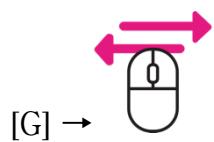
回転



横移動



拡大/縮小



選択

[Z]

[Z]

---

## Appendix2: ローカルサーバーを利用する

JSON ファイルのロードには XHR を利用するため、 CORS のセキュリティにより file://から始まるパスでは表示ができません。ローカルサーバーを起動し、表示確認をしましょう。

手軽にローカルサーバーを利用する方法として、 Web Server for Chrome という Google Chrome 用のアドオンがあります。chrome web store から検索して利用するといいでしよう。

また、macOS の場合には、ターミナルから

```
python -m SimpleHTTPServer
```

を実行すると現在のディレクトリーが <http://localhost:8000/> で開けるようになります。