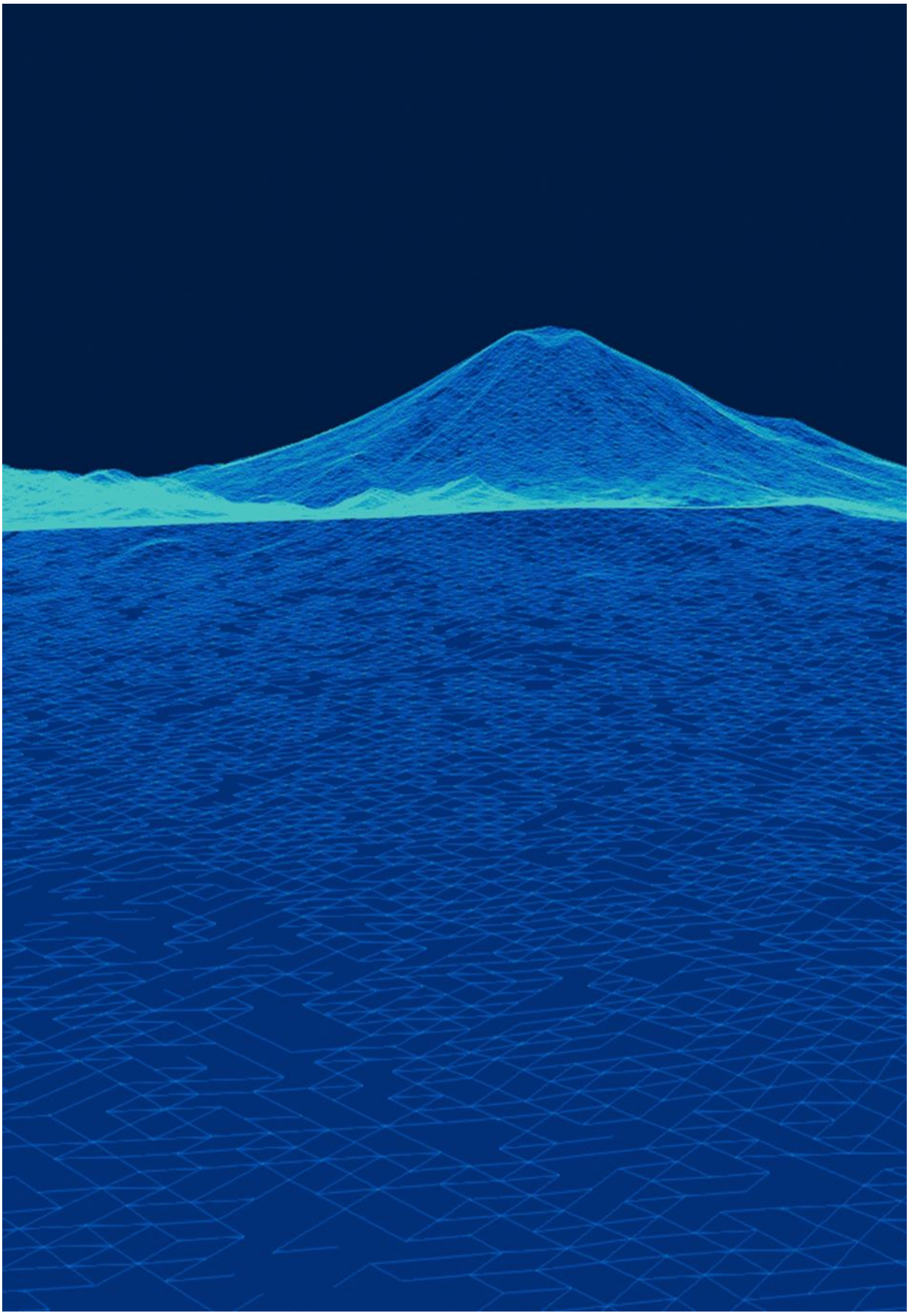
PLATEAU by MLIT

PLATEAU Survey Report

3D 都市モデル活用のための調査資料



3D 都市モデルに最適化された WebGIS 開発に関する調査レポート

series No. 60

Survey Report for the Development of a WebGIS Map Engine Optimized for 3D City Models

目次

1. はじめに

1-1. 調査の背景・課題・目的

本調査は、現在の WebGIS で用いられる地図エンジンのアーキテクチャや、新しい地図エンジンで利用可能な最新技術を調査し、それらの技術を用いることで、以下に述べる課題を解決するような新しい地図エンジンが開発可能かどうかを、プロトタイプを開発しながら検証することを目的とする。

PLATEAU の 3D 都市モデルのデータを Web 等で 3D の地図上に可視化するには、データの表示に使用する地図エンジンが重要な役割を担う。地図エンジンは、GIS データを読み込み、データのみならず地球・大気・地形・太陽などの描画を行い、ユーザーがインタラクティブにカメラ等を操作してデータを閲覧できる画面を提供する。こうした機能を実現するために、3DCG などをはじめとするさまざまな技術が使用されている。

こうしたデータの可視化アプリケーションを開発するにあたり、どのような地図エンジンを選択するか決める必要があるが、選択する地図エンジンの影響はどのようなデータをどのようなスタイルで可視化できるか、だけに限らない。エンジンが先進的で高品質なレンダリングの機能を提供していれば、よりリアルで没入感のあるビジュアルを作り出し、効果的にデータを可視化することができる。また、データの読込や描画の処理速度が速く省メモリな、パフォーマンスに優れたエンジンは、ユーザーによる画面操作の心地よさや体験（UX）を大きく向上させることができたり、端末のバッテリー消費量や電気使用量を抑えたりすることができる。

3D 都市モデルの可視化環境として Project PLATEAU が開発してきた PLATEAU VIEW は、Web 上で PLATEAU のデータの表示が可能なアプリケーションであるが、2020 年度にリリースされた Ver1.0 から 2021 年度の Ver1.1、2022 年度の Ver2.0、そして 2023 年度の Ver3.0 に至るまで、一貫して地図エンジンとして Cesium を使用している。Cesium を使用して PLATEAU VIEW を開発していく中で、以下のような課題が明らかとなっている。

1. より品質の高いビジュアルの作り込みが困難である
   CesiumJS は、WebGL を使用したレンダリング機能を内包している地図エンジンである。そのレンダリング部分の内部は、多くが可視化アプリケーション開発者からは隠蔽されており、レンダリングのカスタマイズの自由度が低い。これは例えば「地球上の地形に落ちる影に光の照り返しのような空気感をもたらす」「水面に映り込むビルの反射を表現する」といった、より高品質なビジュアルを作り込むことを困難にしている。エンジンを改造すれば実現できないことはないかもしれないが、今後のエンジンのアップデートに追従し続けることが困難になるという大きなデメリットがある。

2. 大規模なデータを可視化すると、アプリケーションの動作速度が低下し、ユーザー体験が悪化する
   CesiumJS は Web 上で動作する JavaScript のライブラリである。近年 Web ブラウザにおける

JavaScript ランタイムは目覚ましい高速化を遂げてはいるが、それでも、CesiumJS で大きなデータの読込画面操作を行うと、いわゆる「フレーム落ち」が発生してユーザーの体験を低下させることがある。またスマートフォンなどのモバイル環境では、フレーム落ちのみならず、端末によってはアプリケーションそのものがフリーズしてしまったり、CPU 等が発熱してバッテリーを多く消費したりするといった問題がある。これを回避するには、処理アルゴリズムやデータの持ち方を工夫するだけでなく、マルチスレッドによる並行処理、非同期 IO など、さまざまな技術を用いて、CPU・GPU・メモリといったハードウェアの性能をなるべく無駄なく活かし切ることが必要である。しかし、Web ではマルチスレッドが限定的にしか使用できないなどの制約が存在し、現在のところハードウェアのリソースを十分に使い切ることができているとは言えない。

3. Web 以外のさまざまなプラットフォームで、同様に動作するアプリケーションの開発・移植が困難である

   CesiumJS は Web 上で動作するライブラリであるため、Web ブラウザを使用すれば、PC・モバイルなどの端末や OS を問わず、ある程度同様のビジュアルや機能で閲覧が可能である。ところが前述のように、Web にはパフォーマンスの問題や制約が存在する。それを回避するためにネイティブアプリケーションを開発するという選択肢もある。しかし、Web アプリケーションとは全く別の開発言語・ライブラリ・フレームワーク・グラフィックス API 等を用いて開発する必要があり、Web 版と同様に動作するようなネイティブアプリケーションの開発は容易ではなく、開発コストが多大にかかる。

   また、近年ハードウェア・ソフトウェアともに発展が著しく、例えば VR ゴーグルのような新しいハードウェアが生まれたり、リアルタイムでレイトレーシングを計算可能な回路を積んだ新しい GPU が販売されたりするようになってきている。こうした新しい技術がもたらす新しい表現に地図アプリケーションがいち早く対応するには、それぞれのハードウェアに対応して地図エンジン内部のレンダリング部分の実装をその都度開発する必要がある。ところが多くの地図エンジンはレンダリング部分の実装と密結合になっており、レンダリング部分だけを別の実装に切り替えることが困難である。レンダリング以外の処理を共有しながら Web にもネイティブにも対応できるような地図エンジンは、現在のところ見つけることができていない。以上の問題により、Cesium などの地図エンジンを使用するだけでは、周辺技術の進展に対し素早く柔軟に対応することが困難な状況である。

ところで、CesiumJS の開発が始まったのは 2011 年頃であり、開発開始から 10 年以上経った現在では、ハードウェアやソフトウェアの技術が大きく発展し、これまでになかった技術が使用可能になってきている。これらの技術により、これまでは解決が困難であった課題を解決することができるようになっている可能性がある。

そこで本調査は、現在のさまざまな地図エンジンのアーキテクチャや、新しい地図エンジンで利用可能な最新技術を調査し、それらの技術を用いることで、以上に述べる課題を解決するような、新しい地図エンジンが開発可能かどうかを、プロトタイプを開発しながら検証することを目的とする。

本調査は、前半の「現在の地図エンジンのアーキテクチャと最新技術の調査」と後半の「地図エンジンのプロトタイプの開発」の 2 つに分けて実施する。第 2 章では前半の調査を、第 3 章では後半の開発について述べる。

2. 現在の地図エンジンのアーキテクチャと最新技術の調査

2-1. 調査の目的

本章では、現在の地図エンジンのアーキテクチャと、新しい地図エンジンにおいて利用可能な最新技術の調査を行う。

まずは、現状の技術、例えば一般的な地図エンジンに関する実装や、周辺技術の現状について整理を行う。その後に、地図エンジンに関連する新しい技術の調査を行う。

2-2. 地図エンジンの基本的な仕組み

CesiumJS や MapLibre GL JS などの既存の Web 地図エンジンの内部を機能別に整理すると、4 つのコンポーネントに分けることができる。

1. ライブラリ（アプリケーション開発者が使用し地図エンジンを外部から操作する）
2. レンダリングエンジン（地図エンジンの状態を基に最終的な画面を描画する）
3. メインループ（ユーザー入力などを基に地図エンジン内部の状態を変化させる）
4. GIS エンジン（GIS 特有の処理を行う）

なお、各コンポーネントの名称は一般的に定着しているものではなく、本レポート内でのみ使用する便宜上の名称も含む。

各コンポーネントの関係は以下のとおりである。

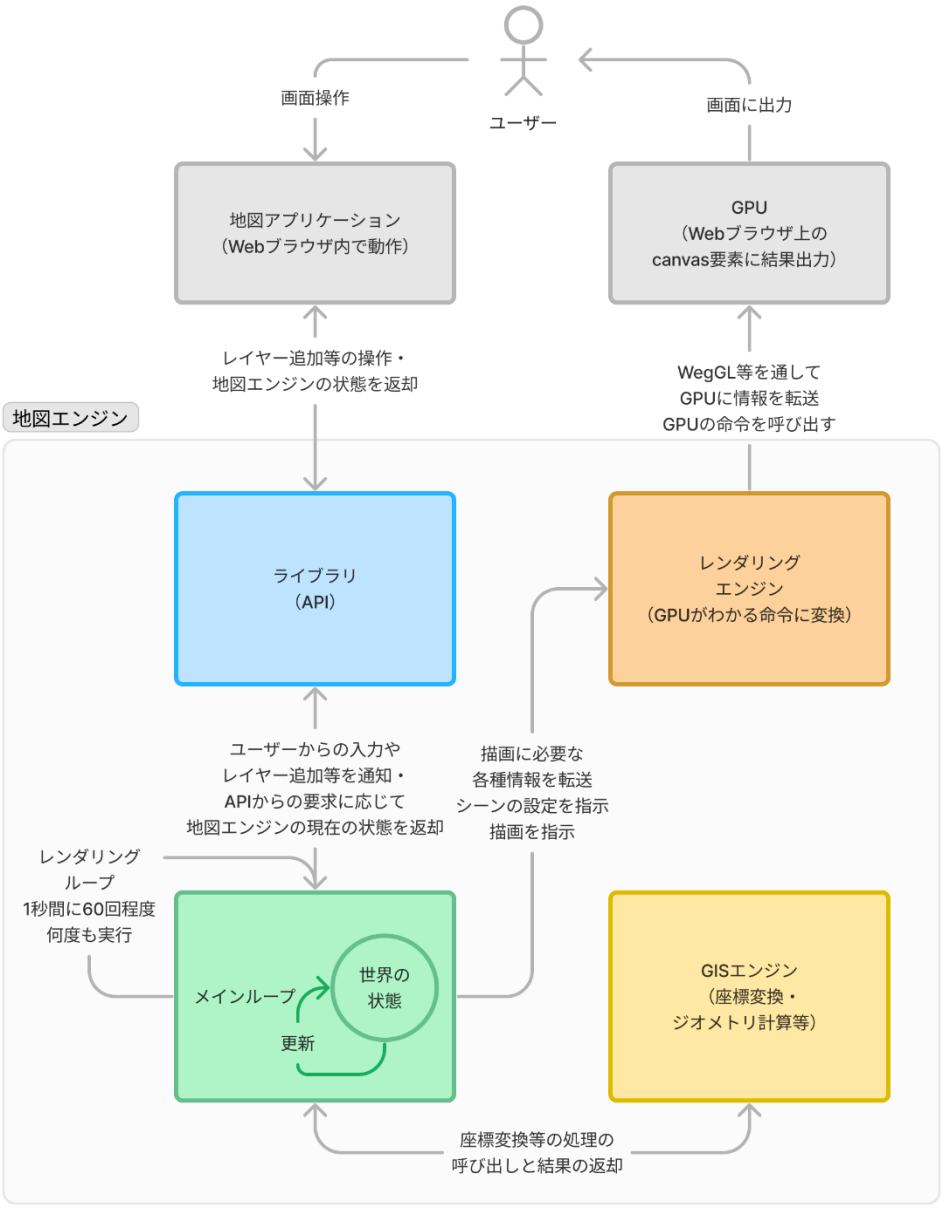


図 1　地図エンジン内部の機能コンポーネント

2-2-1. ライブラリ

地図アプリケーション開発者にとって、地図エンジンは、ライブラリの形で利用できるように提供されていることが多い。例えば、CesiumJS は JavaScript のライブラリであり、ソースコードをダウンロードしてサーバーに配置して利用する他に、CDN や npm 経由でも利用できる。ただし、それだけではアプリケーションは開発できず、ライブラリを呼び出してどんなデータを表示させるか記述するなど追加の実装が必要である。

ライブラリは開発者に対して地図エンジンの各種機能を API として提供する。API が提供する関数を呼び出し必要なデータを与えたり、ユーザーの入力に応じたイベント処理を記述したりすることで、地図アプリケーションが開発できる。開発者にとっては、機能の豊富さやパフォーマンスの良さに加え、API が使いやすいかどうかは開発生産性に直結するため重要である。

例えば、CesiumJS を例に取ると、Viewer、Scene、Globe、Camera、Clock、Entity、Color、DirectionalLight、Cesium3DTileset といったクラスを開発者は利用可能である。座標を表すクラスとして、Cartographic、Cartesian3 などが、地物のスタイルを表すクラスとして、PolygonGraphics、PolylineGraphics などのクラスが存在する。また、よりレンダリングエンジンの処理に近い低レベル API として、Primitive、Geometry、Appearance、Material などのクラスが存在する。
地図エンジンの内部はライブラリの中に隠蔽されており、API が提供する機能以上のことを行うことは、エンジンの改造が必要であり技術的に大きな困難を伴う。

2-2-2. レンダリングエンジン

通常、3DCG に対応した地図エンジンは、地球、大気、地形、太陽、建物、地図タイル、ポリゴンなど、さまざまな視覚的情報を描画し、ユーザーに画面上で表示する機能を持つ。この描画を行うのがレンダリングエンジンである。

レンダリングエンジンは、与えられた情報を基に一枚の静止画を出力する役割を担う。この過程で GPU との通信が発生し、さまざまなデータやコマンドの送受信が行われる。

ユーザーの操作に応じて画面に変化を起こすといった振る舞いは、レンダリングエンジン内部では発生しない（詳しくはメインループの節を参照）。

どのようにして最終的な画面が描画されるかの詳細は、後述の「3DCG のレンダリング・GPU の基本的な仕組み」の節を参照。

2-2-3. メインループ

可視化される世界（地球、大気、地形、太陽、建物、地図タイル、ポリゴンなどを含む状態）に関する情報を保持し、メインループと呼ばれる処理を行い、レンダリングエンジンに指示を出すコンポーネントである。

レンダリングエンジンは必要な情報を渡すと最終的なビジュアルを描画してくれるが、1 回描画しただけではただの静止画である。ところが人間の特性により、少しずつ変化する静止画を連続して見ると、実際には動いていなくてもあたかも動いているように感じられる（仮現運動・アニメーションの原理）。これ

に則り、十分に速い速度で画面のレンダリングを繰り返し行い、画面に映る内容を少しずつ変化させていくことで、ユーザーはアプリケーションを操作しているという感覚を得られる。そのためには、毎レンダリング前に、例えばカメラの位置といった世界の情報を少しずつ変化させていき、その情報を基にレンダリングを行う必要がある。こうした処理を担うのが「メインループ」である。

メインループの内部では、アプリケーションが動作している間、以下の 2 つの処理を高速にひたすら繰り返し実行し続ける。一般的には、60FPS（1 秒間に 60 回）で処理されることが多い。

1. フレームの更新
   経過時間やユーザー操作に応じて、カメラの位置や向き・ライトの種類や向き・建物や地形などのオブジェクトなど、世界に存在するさまざまな情報の内容を更新する。また、必要に応じて外部データ（3D モデル・画像・音など）の読込を指示し、結果を保持するなど、リソースの管理を行うこともある。
2. フレームのレンダリング
   フレームの更新で得た最新の世界の情報を基に、レンダリングエンジンに必要な情報を渡し、1 枚の絵を描画し画面に表示する。フレーム更新が行われなかった場合、世界は変わらないので同じ画面が出力されることになる。

重要なことは、上記 1 と 2 の処理を 1 秒間に 60 回行う必要があることである。つまり、1 と 2 合わせて約 16ms 以内に処理を完了させないと、いわゆる「フレーム落ち」が発生し、ユーザー体験の悪化につながる。特に 3DCG の場合、要求される計算量の割にごくわずかな時間しかレンダリングに使うことができないことから、専門技術であるリアルタイムグラフィックス技術が発展してきた。

2-2-4. GIS エンジン

地図エンジンでは数多くの GIS 特有の概念が登場し、独自のデータ処理が必要なため、メインループの上にさらに、GIS 特有の処理を担うコンポーネントを要する。このコンポーネントは、メインループや、（データ処理をメインループとは非同期に行う）ワーカー、ライブラリから呼び出される。

具体的な概念や処理内容の詳細は、後述の「GIS 特有の概念と処理」の節を参照。

2-3. 3DCG のレンダリング・GPU の基本的な仕組み

地図エンジンのビジュアルの中核を担うのは、レンダリングエンジンと呼ばれるコンポーネントである(図 1 参照)。レンダリングエンジンは、読んで字の如くレンダリング（描画）を担当する。

スマートフォンや PC といったデバイスで高速にレンダリングを行うためには、GPU（Graphics Processing Unit）の活用が欠かせない。
本節では、レンダリングエンジンと GPU の基本的な仕組みについて述べる。

2-3-1. レンダリングエンジン

地図エンジンの重要な役割の一つは、地理空間データを効果的かつ直感的に視覚化することである。例えば Google Earth では、地球上のさまざまな地域や地形が高度なグラフィックスで描画され、ユーザーは地球儀上で没入感のある探索ができる。これを実現するのがレンダリングエンジンのコンポーネントである。ここでは地図エンジンは 3D での描画を行うものとし、本小節では 3DCG のレンダリングエンジンについて述べる。

地図エンジンにおけるレンダリングエンジンの基本的な役割として、まず地表の描画が挙げられる[1]。地球の表面を構成するさまざまな地形や構造物をリアルな 3D モデルとして描画し、山脈、川、湖、都市などが視覚的に表現される。次に、視点と視界の制御が挙げられる。カメラの位置、向き、焦点距離などを制御することで、ユーザーが地球上を動き回る体験を提供する。自然な体験のためには柔軟で洗練された視点制御が求められる。最後に、地図上の情報の表示が挙げられる。建物、道路、地名などの情報を適切に描画し、ユーザーに地理的なコンテキストを提供する。これにより、ユーザーは必要とする情報を得られるだけでなく、地球上のオブジェクトの位置関係が理解しやすくなる。

レンダリングエンジンが高品質に地表を描画し、スムーズにカメラを制御するために、3DCG の技術が用いられる。3DCG におけるレンダリングは、地表のオブジェクトの位置や形状、カメラの位置、光源の位置などの情報を基に、PC やスマートフォンのディスプレイの各ピクセルに描画する色情報を決定するプロセスである。

3DCG のレンダリング手法は、ラスタライズ法とレイトレーシング法に大別できる。レイトレーシング法のほうが現実世界の視界への光の入力を高度にシミュレートしており高品質なグラフィックスを実現できるが、高負荷な処理となるため、地図エンジンやゲームエンジンのようにユーザーとのリアルタイムなインタラクションが求められるケースではラスタライズ法が用いられることが一般的である。本稿でも

[1] 後述するように、レンダリングエンジンはあくまでもポリゴンなどの基本図形を描画するものであり、何を描画するかは応用次第である点には注意。

以降ラスタライズ法についてのみ述べる。

さて、ここまで「オブジェクト」と呼称したものは 3DCG の文脈では「モデル」と言い換えられる。3D モデルはポリゴンの集合であり、最小のポリゴンは 3 つの頂点とそれで囲まれた平面（三角ポリゴン）である。多数のポリゴンを組み合わせることでさまざまな形状のモデルが表現できる。ポリゴンの平面にはテクスチャと呼ばれる色や模様を付加することができる（図 2）。

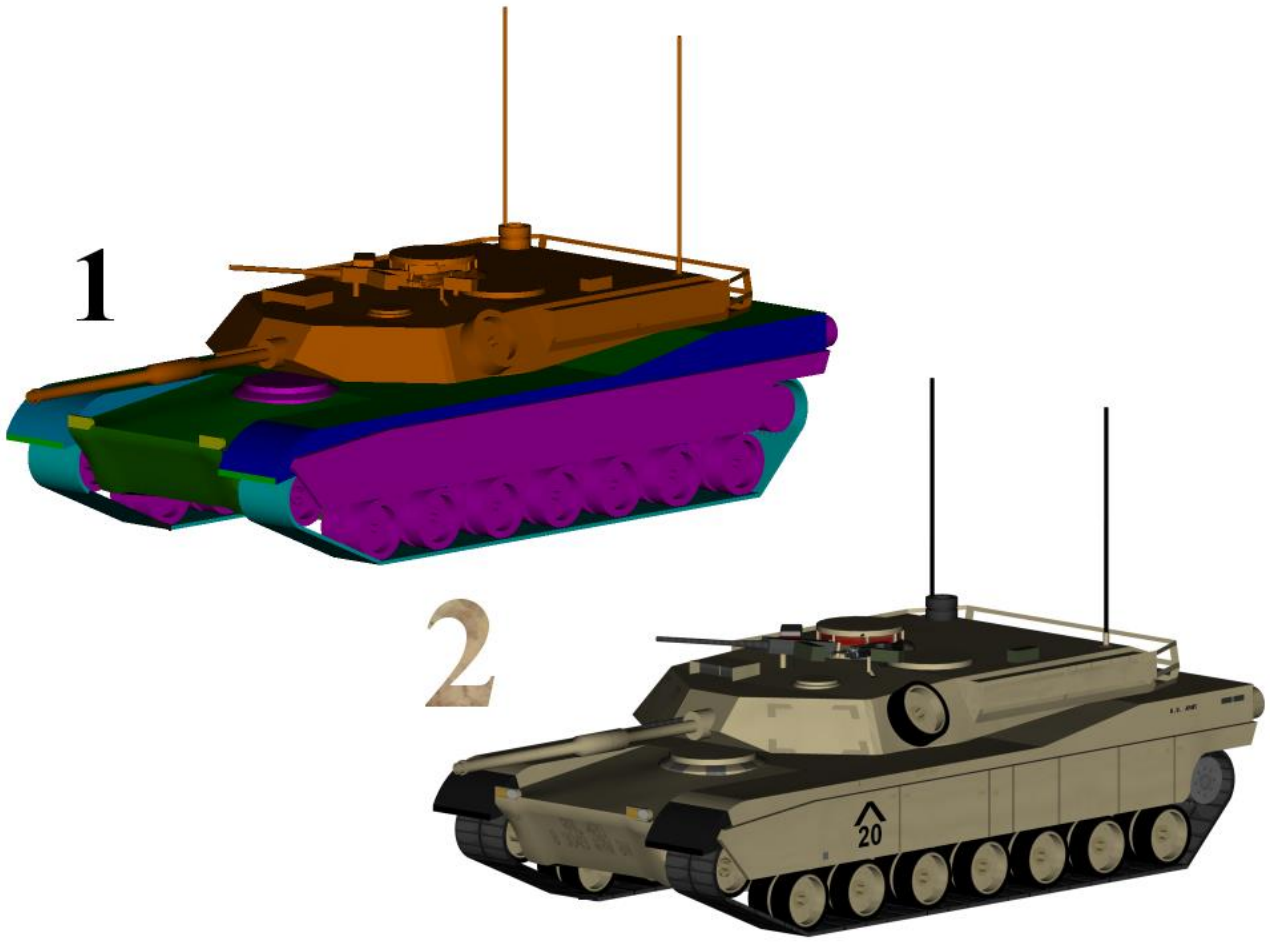


図 2 ポリゴンにテクスチャを貼り付ける（By Anynobody - Own work, CC BY-SA 4.0,https://commons.wikimedia.org/w/index.php?curid=3441137）

2-3-2. レンダリングパイプライン・シェーダー

地表のモデルをディスプレイに描画する際には、カメラの位置を加味してモデルの頂点のディスプレイ上での座標を計算し、光の当たり具合も加味してカメラから見える部分のテクスチャの各ピクセルの色を計算する。この計算を行うプログラムのことをシェーダー、あるいはシェーダープログラムと呼ぶ。

レンダリングエンジンの実装ごとに利用可能なシェーダーは異なるが、その中でも基本的なものとして頂点シェーダーとフラグメントシェーダーが挙げられる。図 3 に示されているのはレンダリングパイプラインと呼ばれ、モデルから画面上に描画されるピクセルに変換されるまでの流れを示すものである。ここではレンダリングパイプラインの全体に触れることは目的とせず、2 つのシェーダーの役割のみを概観し、その計算が GPU で高速化されるという後続の説明につなげることとする。頂点シェーダーは、各モデルの頂点位置とカメラの位置から、全てのモデルの位置・角度を計算する。また、光源情報を加味して頂点単位で陰影の計算を行い、レンダリングパイプラインの後のフラグメントシェーダーでの陰影描画の前処理とする。頂点シェーダーの後にはクリッピング（ディスプレイ外のモデルやその一部を取り除く）・ビューポート変換（全モデルの頂点やポリゴンの 3D 座標からディスプレイの 2D 座標に変換する）・ラスタライズ（ビューポート変換済みの頂点情報から、ディスプレイ上で塗りつぶすピクセルを決定する）が実行される。フラグメントシェーダーはラスタライズ結果・頂点ごとの陰影情報などを基にして、各ピクセルの色情報を最終的に決定する。

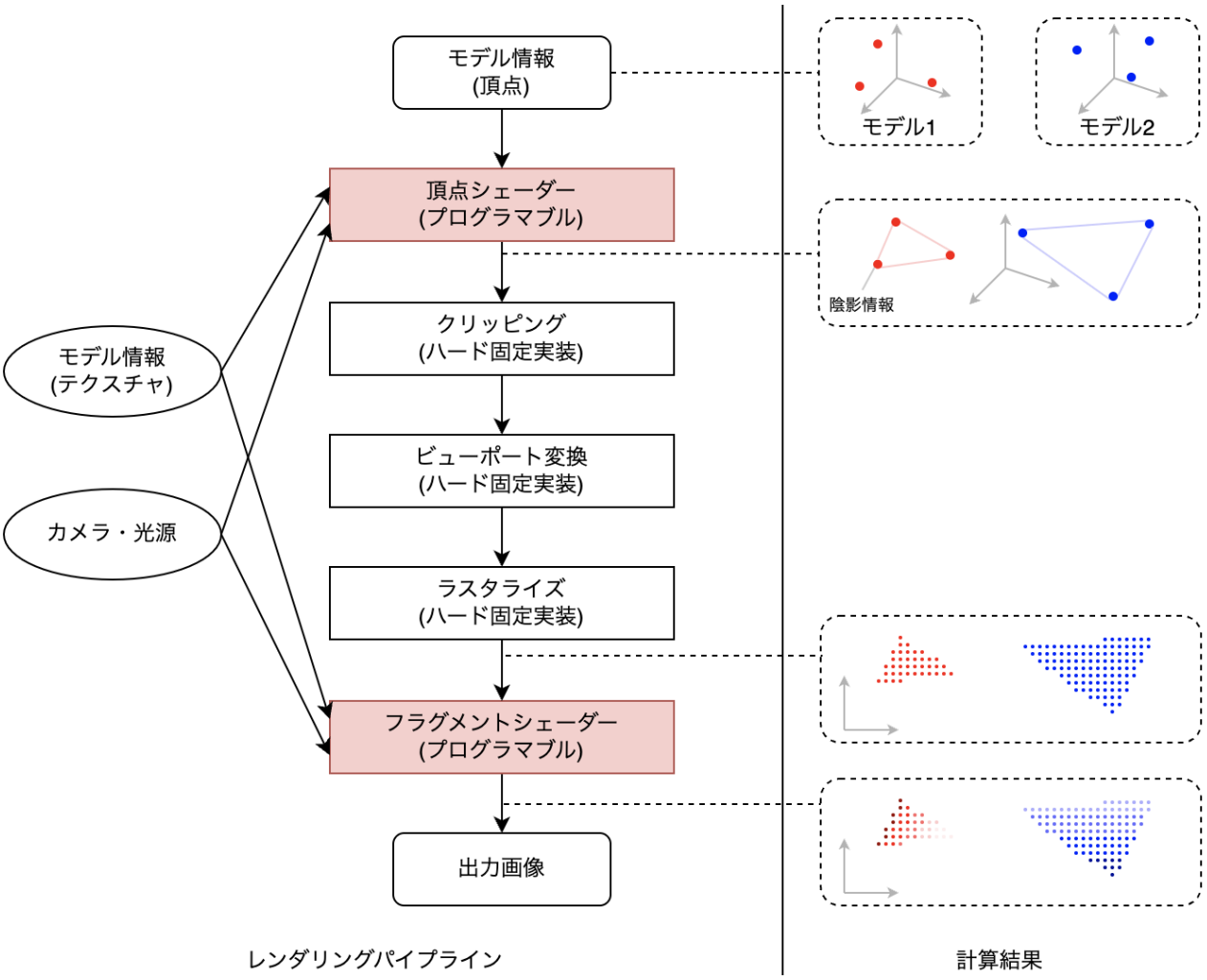


図 3 レンダリングパイプラインとその計算過程

2-3-3. グラフィックス API

実際にはレンダリングは GPU と呼ばれるハードウェアを用いて行われることが多く、その GPU にアクセスするためにグラフィックス API と呼ばれる API を呼び出す必要がある。代表例としては、OpenGL, DirectX, Valkan, Metal などが存在する。しかし、これらのグラフィックス API を直接用いてレンダリングを行う場合、以下の理由により、高度なカスタマイズ性を得られ、パフォーマンスチューニングも行いやすい反面、実装の難易度が高まる。

- グラフィックス API が公開する機能は低レイヤーであり、画面を描画するまでには多数のデータ転送と命令の呼び出しが必要である。例えば、シェーダーのソースコード・頂点バッファ・インデックスバッファ・属性・ユニフォームなどさまざまなデータを GPU に転送し、各種命令を呼び出す必要がある。カメラやライトといった人間にとって分かりやすい概念をグラフィックス API は扱わない。
- 端末に搭載する GPU の種類によって対応する API の種類やバージョンが異なり、レンダリング上の制約も変化する。レンダリングエンジンをさまざまな環境に対応させるには、それぞれのグラフィックス API に対応した実装が必要である。

Web の世界では、OpenGL ES をベースとした WebGL という、ハードウェアの違いを吸収したグラフィックス API が利用可能である。しかし、WebGL を直接呼び出すことなく、カメラやライトといったより分かりやすい概念を用いてシーンを描画するためのライブラリも多く存在する。具体例としては、Three.js が挙げられる。Three.js は WebGL の難しさを覆い隠して、カメラ・ライト・メッシュ・マテリアルなど、より直感的な API を開発者に提供すると同時に、独自のシェーダーやマテリアルの実装も可能でカスタマイズ性が高い。このように、レンダリングエンジンは、直接グラフィックス API を呼び出す実装もあれば、別のライブラリを間に挟んで間接的にグラフィックス API を呼び出すものもある。

多くの地図エンジンはレンダリングエンジンを自作し、WebGL などのグラフィックス API を直接使用している。これはパフォーマンス面では有利に働くが、マルチプラットフォーム性や移植性の面では不利になることが多い。

2-3-4. GPU・GPGPU

頂点シェーダーで行われている計算を概観すると、それは多数の頂点の座標情報（ベクトル）に対し、カメラや光源を加味して、座標変換や陰影計算を行っている。各頂点に対して行われる計算は均質であり、かつ線形代数を基礎とした数値計算が基本となる。

フラグメントシェーダーでは、多数のピクセル（色の RGB 情報はベクトルで扱える）について、頂点ごとの陰影情報などを元に最終的な色を決定している。これも大部分のプロセスは、各ピクセルに対して均質な数値計算で実行できる。

CPU は少数データに対して複雑かつ多彩な計算を実行するのに向いてる一方で、シェーダープログラムのように多数のベクトルに対し均質な数値計算を行う処理に特化しているのは GPU である。GPU には ALU (Arithmetic Logic Unit) と呼ばれる乗算・加算などの数値計算を行う回路が多数搭載されている（代わりに、数値計算以外の命令を実行するような回路面積は小さい）ために、このような特色を持つ（図 4）。

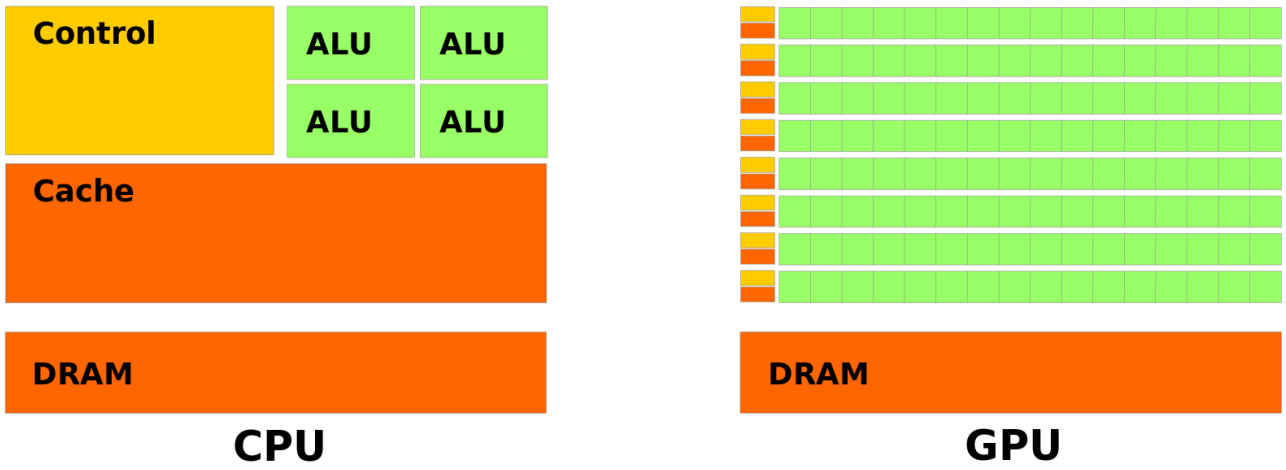


図 4 NVIDIA, CC BY 3.0 https://creativecommons.org/licenses/by/3.0, via Wikimedia Commons

レンダリングパイプラインのシェーダー以外のフェーズも「多数のベクトルに対する均質な数値計算」という特徴は共通であり、GPU の回路によって高速に計算が行える。

レンダリングエンジンが美麗なグラフィックスをリアルタイムに実現するためには、GPU の活用が欠かせないものとなっている。

ここまでは地図エンジンの中のレンダリングエンジンコンポーネントに絞って GPU の効用を述べたが、その他のコンポーネントでも GPGPU（General Purpose GPU）としての GPU 活用により、地図エンジンのインタラクションをより高速化できる余地がある。例えば、地図上の特定区間（タイル）における建物や地形の情報は JSON という文字列表現でデータベース化されていることがある。JSON のような文字

列を、構造化されたメモリ上のデータに変換することをパース処理と呼ぶが、JSON のパース処理を GPGPU で高速化する実装が存在する[2]。このように、レンダリング以外のデータ処理の部分でも、GPU は地図エンジンのリアルタイムなインタラクションに貢献する可能性が考えられる。

[2] https://developer.nvidia.com/blog/gpu-accelerated-json-data-processing-with-rapids/

2-4. Web で使用可能なレンダリングエンジンの調査

Web 上で高速な 3D 描画を行うことが可能な、実用可能な OSS のレンダリングエンジンのライブラリを調査する。

3D 描画を行うために WebGL を直接呼び出すことも理論上は可能ではあるが、API の抽象度が低いことから多くの実装を要し、いわゆる車輪の再発明が発生しやすいため、適切なレンダリングエンジンを使用することで開発を高速化することができる。

なお、CesiumJS、MapLibre GL JS、deck.gl、iTowns といった既存の Web 地図エンジンでは、Three.js を内部的に使用している iTowns を除き、レンダリングエンジンを自作している事例が多い。

2-4-1. Three.js

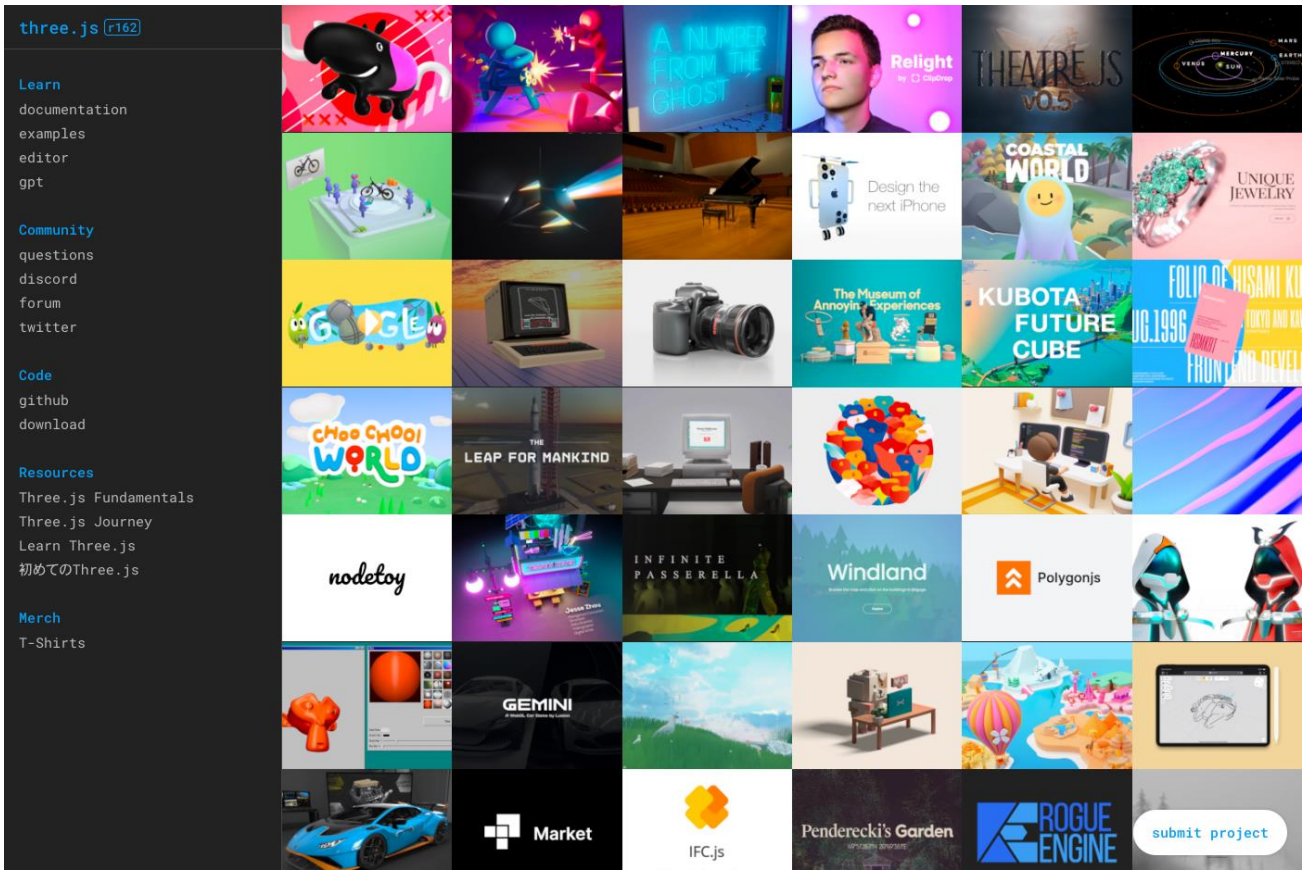


図 5　Three.js 公式 Web サイト https://threejs.org/

WebGL を利用して 3D コンテンツをブラウザ上でレンダリングするための JavaScript ライブラリ。主に WebGL をサポートする。WebGPU のサポートは実験的であり、機能も WebGL に比べると限定的。ライセンスは、MIT ライセンス。

機能としては主にレンダリングを行うことに特化しており、WebGL のグラフィックス API を覆い隠して、人間にとって分かりやすい抽象化した API（シーン・メッシュ・マテリアル・ライトなどの概念が存在）を提供する。シェーダーなどは自由にカスタマイズ可能なため、細かいパフォーマンスチューニングも行いやすい。

多様な用途で使われ、オープンソースコミュニティによって支えられていることから、豊富なドキュメントとサンプルが存在するのも特徴。

地図エンジンが対応するレンダリングエンジンの 1 つとして候補に上がると考えられる。

2-4-2. Balylon.js

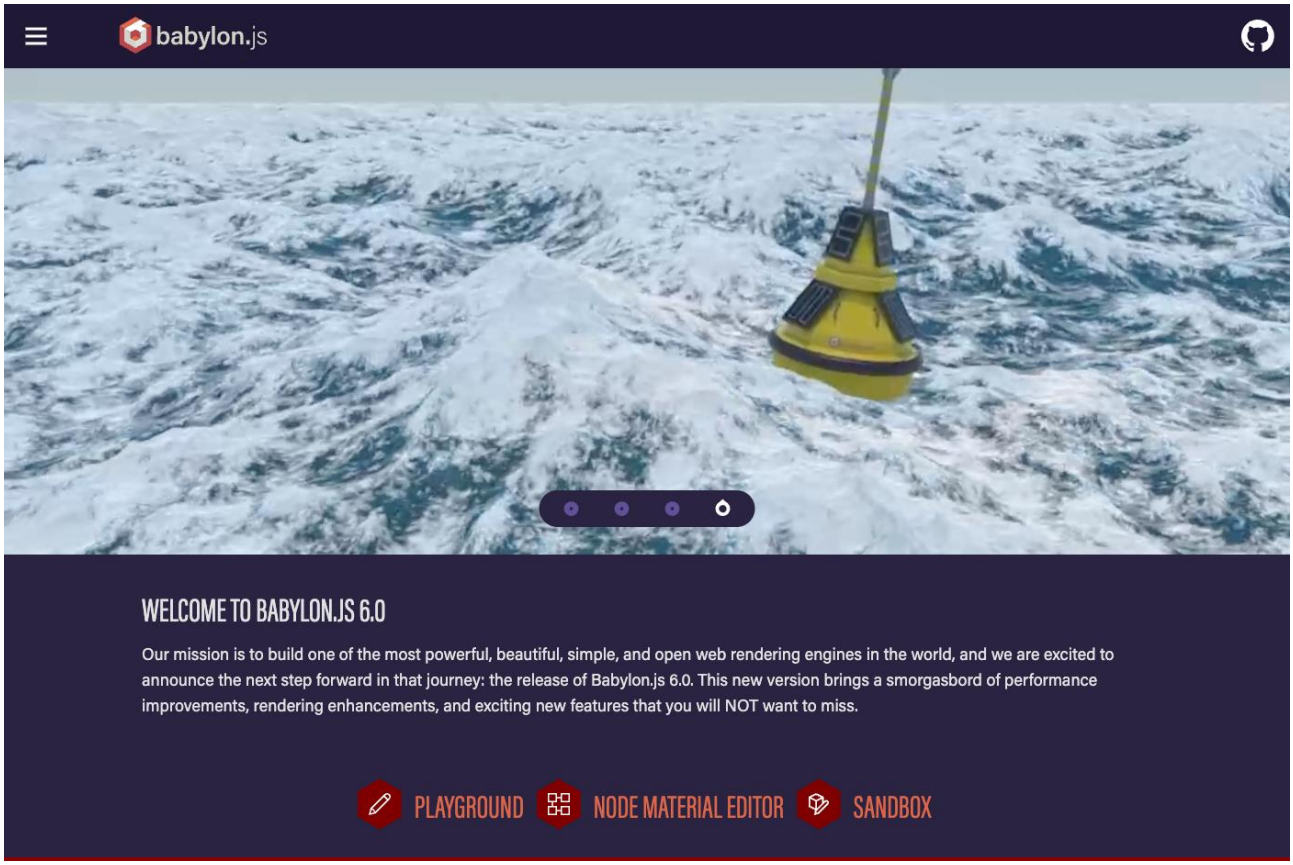


図 6　Babylon.js 公式 Web サイト https://www.babylonjs.com/

Microsoft が開発する、Web 上で高品質な 3D アプリケーション、ゲーム、インタラクティブなシーンを作成可能な JavaScript ライブラリ。WebGL だけでなく WebGPU を正式にサポートしている。ライセンスは、Apache License 2.0。

Three.js と比べると、レンダリングにとどまらずゲームエンジンとしての機能を持ち、物理エンジン、サ

ウンドシステム、GUI、WebXR との統合などが利用可能である。

将来的に地図エンジンが対応するレンダリングエンジンの 1 つとして候補に上がると考えられる。

2-4-3. A-Frame

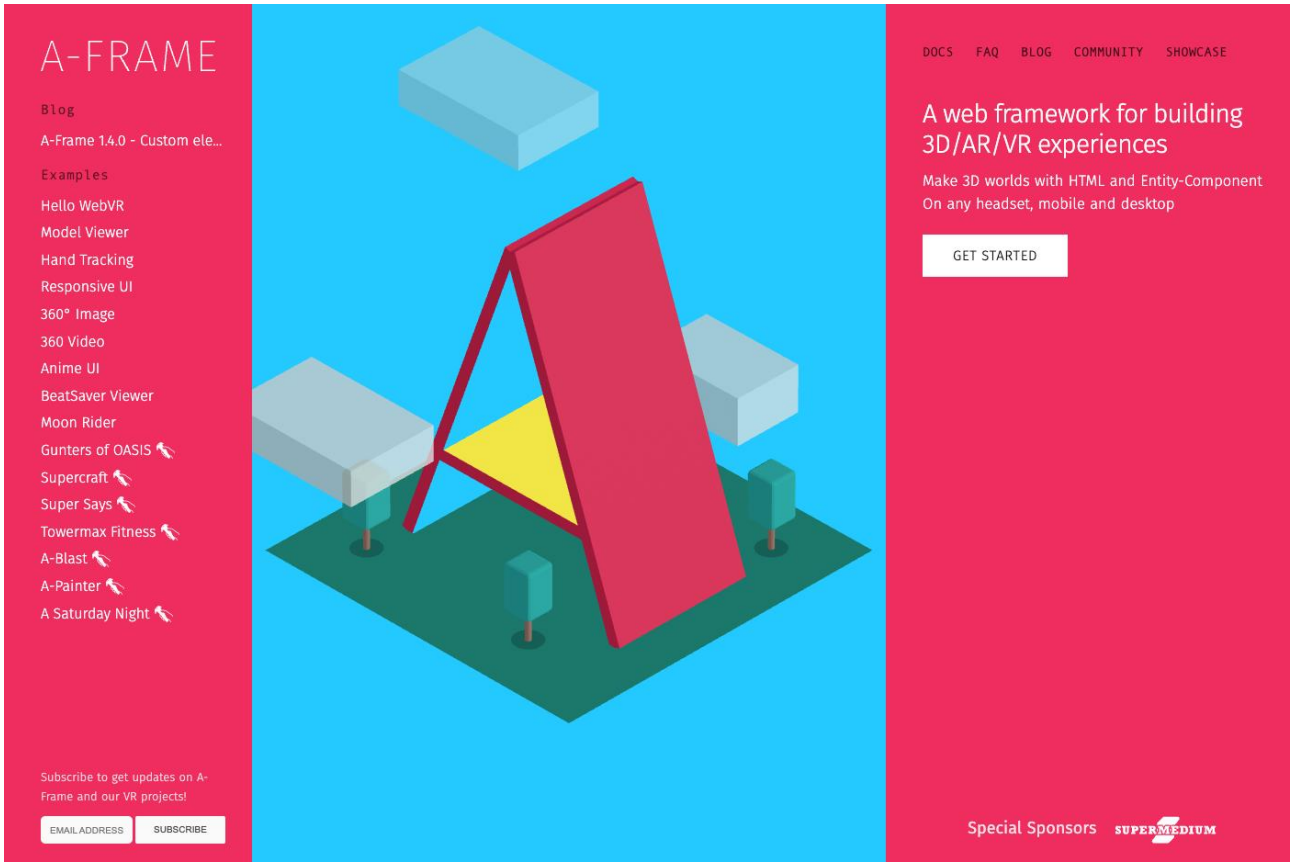


図 7　A_Frame 公式 Web サイト https://aframe.io/

Web 上で 3D 及び仮想現実(VR)体験を構築するための Web フレームワーク。ライセンスは、MIT License。

HTML に似た構文を使用して簡単に 3D シーンを作成できるため、開発者だけでなくデザイナーやアーティストにも親しみやすい。

内部的に Three.js を使用している。また、エンティティ・コンポーネント・システム（ECS）ベースのアーキテクチャを採用しており、再利用可能なコンポーネントを作成可能。

2-4-4. Bevy

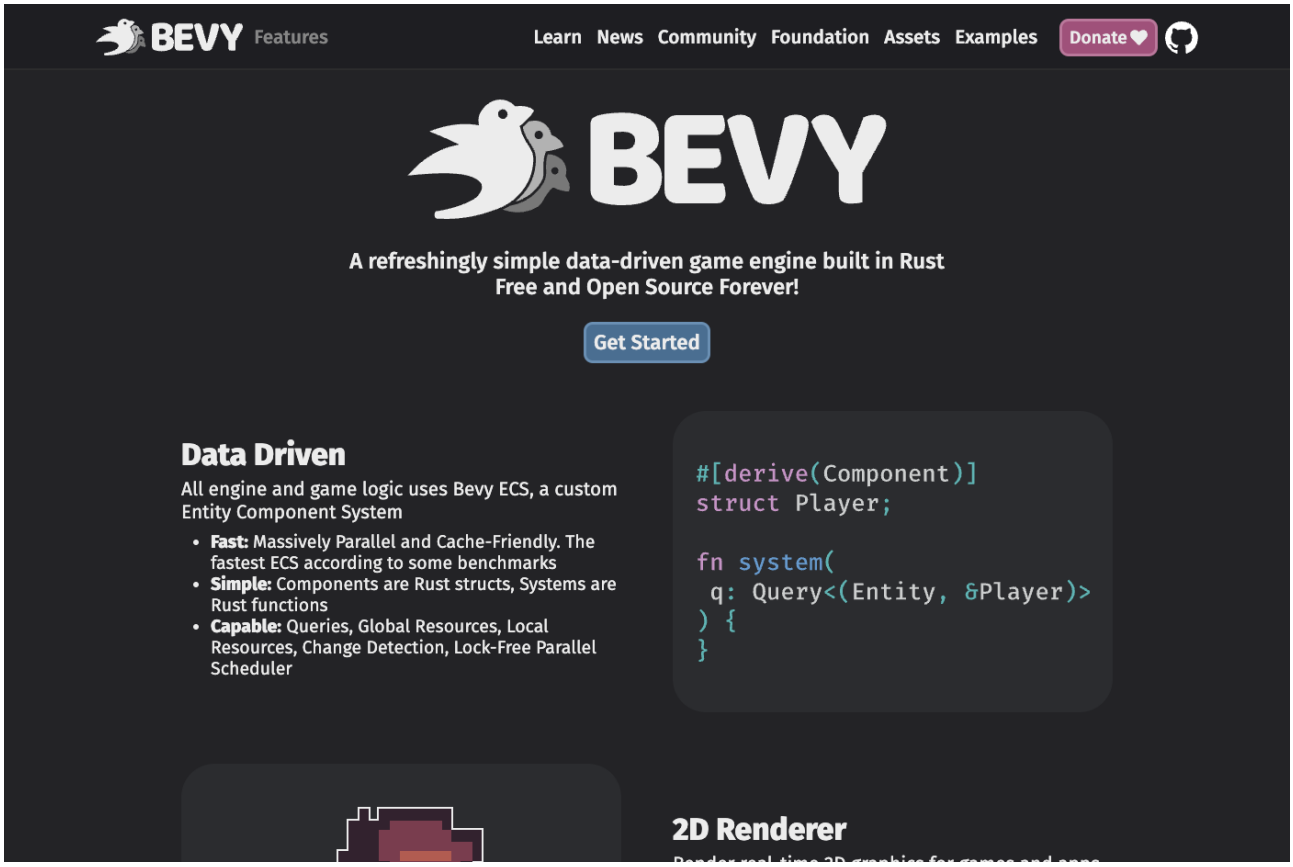


図 8　Bevy 公式 Web サイト https://bevyengine.org/

Rust 言語で書かれた軽量でモダンなゲームエンジン。

データ駆動型の設計思想を採用しており、高速で柔軟性が高く、Web だけでなくデスクトップやモバイルを含むマルチプラットフォームに対応する。Web 向けには WASM にコンパイルされて動作する。ライセンスは、MIT/Apache 2.0 のデュアルライセンス。

プラグインによる拡張をサポートしており、2D・3D ゲーム開発、オーディオ処理、物理シミュレーション、GUI など、幅広い用途に対応する。

エンティティ・コンポーネント・システム（ECS）アーキテクチャを使用しており、ゲームのコードを高度にモジュール化し、性能を最適化することが可能。Rust 言語の安全性と並行処理の特徴を活かし、メモリ安全で高速なゲーム開発が可能。また、各種機能が細かいモジュールに分かれて提供されているため、ECS の機能だけを使用するといった使い方も可能。

2-5. 地図エンジンで必要な GIS 特有の処理

2-5-1. 座標変換・投影

3DCG の世界では XYZ の 3 軸からなる直交座標系で座標が表され、単位にはメートルが採用されることが多い。しかしその座標系に地球を描画し、緯度経度などの地理的な座標を基にオブジェクトを狙った位置に配置するには、以下のような座標系の相互変換が必要となる（さまざまな名称が存在するため、本稿では XYZ 等の略称を用いる）。

- XYZ：地心直交座標系（地心地球固定座標系・Earth-Center, Earth-Fixed・ECEF・ESPG:4978）。地球の重心を原点とし、X 軸を本初子午線と赤道の交点方向、Y 軸を X 軸と直角に東経 90 度の方向、Z 軸を北極方向（慣用国際原点）とする。単位はメートル。自転に同期する。3DCG でのワールド座標系として採用する。準拠楕円体は WGS84 楕円体。CesiumJS など各種地図エンジンでも採用されている。CesiumJS では Fixed frame 及び Cartesian と呼称。
- LLE（Latitude, Longitude, Elevation・ESPG:4326）：経度（度）・緯度（地理的緯度・度）・高さ（楕円体高・メートル）で位置を表す地理的な座標系。準拠楕円体は WGS84 楕円体。GeoJSON などをはじめとする一般的な GIS データフォーマットではこの座標系がよく使われる。CesiumJS では Cartographic と呼称。
- ENU（East, North, Up）：地表上において X 軸を東、Y 軸を北、Z 軸を上方向（地球の中心から遠ざかる方向）とする、メートルを単位とする相対的な座標系。XYZ や LLE と相互変換するには ENU における原点を LLE で指定する必要がある。似た座標系として NED（North, East, Down）等がある。
- Web メルカトル（ESPG:3857）：Web 上でタイルを配信する際によく用いられる投影座標系。Web メルカトルは、WGS84 測地系をベースに、北極及び南極付近の一定緯度以上を省略し、真球を仮定して投影する。世界が正方形に投影されるため分割が行いやすく、タイル配信に都合が良い。

これらの相互変換には、地球楕円体の情報が必要である。地球楕円体とは本来複雑な形状をしている地球をシンプルな回転楕円体で近似したもので、Cesium などの各種地図エンジンでは一般的に WGS84 楕円体が採用されている。

WGS84 楕円体では、地球楕円体の赤道半径が 6378.137m、扁平率が 1/298.257223563 と定義されている。極半径を求めると、6378.137 × (1 - (1/298.257223563)) = 6356.7523142452m となる（実際の計算結果は浮動小数点数の精度により変化する）。これらの半径の情報を用いて楕円体を描くと地球楕円体を 3DCG 空間上に描くことができる。

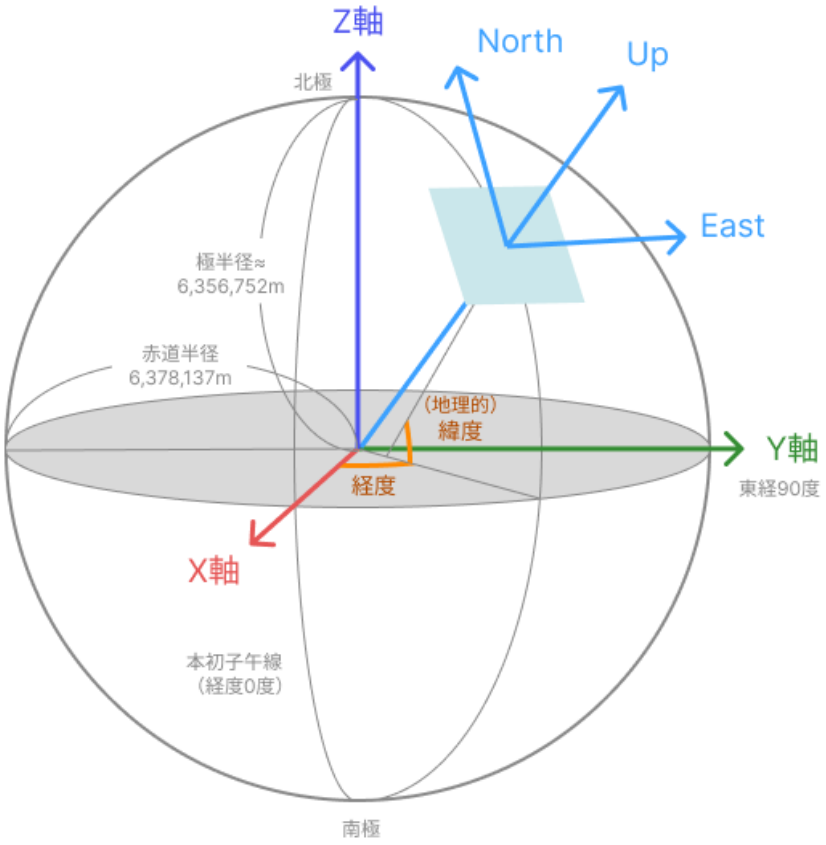


図 9　WGS84 楕円体と各種座標系のイメージ

2-5-2. タイルの計算・分割

GIS では、地球上の広範囲の衛星写真や道路地図などの画像をパフォーマンス良く表示するために、「地図タイル」という手法が用いられる。大きな画像をあらかじめ分割しておき、X・Y・Z（ズームレベル）の数値を指定することで、欲しいエリアだけの画像を取得することができる。地図タイルの座標系には前述の通り Web メルカトルが採用されることが多い。

地球の表面は、びっしり敷き詰められたタイルの集合体としてモデリングすることができる。それらタイルをそれぞれどれだけ分割して細かく表示するかは、カメラとタイルの距離などによって変化する。一般的にカメラがタイルに近づけば近づくほど、タイルはより詳細に分割して表示する必要がある。

そこで、毎フレームの更新時に、カメラの移動に伴いどれほどの詳細度でタイルを表示するべきかを決定するために、タイルごとに画面上の実際の大きさを表す Screen Space Error（SSE）を計算し、SSE と閾値を比較して一定条件を満たすと、そのタイルはより詳細に表示可能であり、より細かいタイルに分割する。

タイルの詳細度が決定された後、各タイルのジオメトリを計算する。（楕円体からタイルが表す一定範囲のみを切り取ったようなジオメトリの頂点座標等を計算する。）

これらのプロセスを経て計算されたタイルに、ラスターレイヤーが読み込まれることでテクスチャがタイルの適切な位置に貼られ、最終的に地表としてレンダリングされる。

図 10　タイルの分割のイメージ

2-5-3. レイヤーの管理・データの読込

地図エンジンにおいてレイヤーとは、何か 1 つのデータソースからデータを読み込んで得られた複数の地物をまとめた概念である。地物には 2D（ベクター・ラスター）と 3D（モデル・ポイントクラウド等）といった種類が存在する。地図エンジンでは、それらレイヤーの追加・変更・削除を管理する必要がある。

レイヤーが追加されると、然るべきタイミングでデータの取得をバックグラウンドで行い、読み込まれたデータのパースを行い、GPU で読込可能な情報（頂点バッファなどの頂点属性・シェーダー・テクスチャなど）に変換する。

その際に、例えば「”種類”の属性が“公園”ならば緑色に塗る」のような、ユーザーから指定された計

算式をもとにスタイルを適用し、地物を色で塗りつぶしたり線の色を変えたりといった見た目の変更を行うこともある。

それらの準備が整うと、メインループを通してレンダリングエンジンに情報が渡され、データが画面に表示される。

データの読込処理は、レンダリングエンジンと並んで地図エンジンのパフォーマンスに大きく影響する部分であり、マルチスレッドを用いて重い処理をメインループから逃す、計算結果をキャッシュする、ドローコールを削減する、データを読み込んでからの地図エンジン上での処理が少なく済むようなデータフォーマット（例：3D Tiles）を使用するなど、さまざまなパフォーマンスチューニングが重要となる。

2-6. 新しいデータフォーマットの調査

地図エンジンで今後用いられ得る、新しいデータフォーマットについて調査する。ここでは特に Web での使用に最適なデータフォーマットを対象とする。

2-6-1. 画像・ラスター

画像には PNG や JPEG といったフォーマットが長らく使われてきたが、最近は品質を保ちながらより圧縮効率の高いフォーマットが登場している。一部は 3D モデルのテクスチャのフォーマットとして使用可能である。

- WebP：Google が開発。従来の画像フォーマットの置き換えを目指しており、PNG よりも 26%、JPG よりも 25%～34%圧縮しながら同等以上の品質を実現可能。アルファチャンネルやアニメーションにも対応。ほとんどのモダンな Web ブラウザで既にサポートされている。glTF のテクスチャとしても使用可能（EXT_texture_webp 拡張）。
- AVIF（AV1 Image File Format）：オープンかつロイヤリティフリーのビデオコーデック AV1 をベースにした画像フォーマット。WebP を超える圧縮効率を実現し、HDR（ハイダイナミックレンジ）コンテンツやさまざまな色空間もサポート。ほとんどのモダンな Web ブラウザで既にサポートされているが、執筆時点ではまだ普及はしておらず、glTF でも正式にサポートされていない。
- HEIF（High Efficiency Image File Format）：特に Apple のデバイスやソフトウェアで広く採用されている。HEVC（High Efficiency Video Codec・別名 H.265）を使用して画像を圧縮し、JPEG に比べて約半分のファイルサイズで同等の画質を実現。使用時には HEVC の特許に注意が必要。
- KTX2：GPU 圧縮テクスチャフォーマット。通常の画像フォーマットを GPU で使用可能にするには、予めピクセルのバイト列にデコードする必要があるため、転送に時間がかかったり、GPU でのメモリ使用量が大きくなったりするという問題がある。KTX2 を使用すると圧縮された状態のまま GPU に転送して使用可能なため、高速に転送が可能で、GPU 上のメモリサイズを削減でき、レンダリングの高速化に寄与する。glTF のテクスチャとして使用可能（KHR_texture_basisu 拡張）。
- Cloud Optimized GeoTIFF：クラウドに最適化された GeoTIFF。HTTP の Range ヘッダを用いた範囲リクエストにより、範囲を指定して欲しいところだけをダウンロードすることができる。

2-6-2. ベクター

GeoJSON、TopoJSON、CZML、MVT（Mapbox Vector Tiles）といったフォーマットが WebGIS ではよく使われる。その他にも、大規模なデータでも高速に読み込むことができるフォーマットが登場している。

- FlatGeobuf：GeoJSON のようにジオメトリと属性を記述できる。FlatBuffers を使用しているため

読み書き速度が高速で圧縮効率に優れ、ストリーム処理も可能。Cloud-Optomized なフォーマットでもあり、ファイル内に空間インデックスを含むことができ、範囲リクエストにより範囲を指定して欲しいところだけをダウンロードすることができる。

- GeoParquet：Apache Parquet を使用したフォーマット。列指向のため圧縮効率や読み書き速度に優れる。

2-6-3. 3D モデル

3D モデルのデータフォーマットとして、glTF が Web では使われるが、glTF そのものは位置情報を持たない（もしくは地心座標系でデータを作成する必要があり手間がかかる）。そこで、位置情報を持ちながらタイルに分割することで処理速度を最適化した 3D モデルのフォーマットとして、3D Tiles や I3S といったフォーマットが存在する。

- 3D Tiles 1.1（3D Tiles Next）：Cesium が策定した従来の 3D Tiles 1.0 は、b3dm など、glTF に独自のヘッダを付加した独自フォーマットを使用していたが、1.1 では純粋な glTF ファイルとしてタイルを記述することが可能となった（メタデータは glTF の拡張として記述する）。これにより既存の glTF エコシステムとの相互運用性が高まる。また、より複雑なメタデータの記述、暗黙的なタイリング、新しい空間インデックスにも対応する。2023 年 1 月に正式にリリースされた。Google Photorealistic 3D Tiles でも使用されている。現状、地球上に 3D モデルを描画する上で最も効率が良いフォーマットと考えられ、本地図エンジンにおいても対応することで良好なパフォーマンスが期待される。

2-6-4. 地形

地形を表現するためのデータフォーマット。

- Mapbox Terrain-DEM（Terrain-RGB）：PNG などの画像フォーマットで表現できる地形のラスターフォーマット。256px 及び 512px 四方の画像の各ピクセルの RGB の値から、高さを 0.1 メートルの分解能で取り出すことができる。なお、国土地理院が公開する標高タイルとは、考え方は近いものの異なるフォーマットである。
- Quantized terrain mesh：Cesium が策定した 3D 地形データを表現するためのベクターフォーマット。地形データを効率的にストリーミングしてレンダリングするために設計され、大規模な地形データセットを扱う際のパフォーマンスとビジュアル品質を両立する。量子化やジグザグエンコード等によるデータ圧縮、タイルによる LOD をサポートし、ウォーターマスク・地形の法線ベクトルなどの追加情報を持つことも可能。GPU が扱えるデータ（頂点バッファ・インデックスバッファ等）に近い形で格納されており、データを読み込んでから GPU への転送までに必要な計算量を抑制できる。地形を処理する上では現状最も効率が良いフォーマットと考えられ、本地図エンジンにおいて

も対応することで良好なパフォーマンスが期待される。

2-6-5. ボクセル

Mapbox GL JS や MapLibre GL JS では、GeoJSON や MVT 等のフォーマットでボクセルを表現可能である。これは、データ内に含まれる 2D ポリゴンに対し、属性の値に応じて「ポリゴンの地面からの高さ」（fill-extrusion-base）と「押し出しの高さ」（fill-extrusion-height）を適用することで表現可能なためである。

なお、ボクセルを表現するためのデータフォーマットとしては、3D プリンティングの分野では「FAV（FAbricatable Voxel）」が存在するが、GIS の用途に適した専用データフォーマットは見つけることができなかった。各ボクセルは大きさが同じという特性等から、既存のデータフォーマットよりもさらにデータを圧縮できる余地があり、GIS に最適なボクセルのデータフォーマットは検討の余地があると思われる。

2-7. 地図エンジンに関連する新しい技術の調査

PLATEAU のデータをより軽量かつ美しく可視化でき、マルチプラットフォームに対応できる新しい地図エンジンをゼロベースで構築することを考え、重要な構成技術として WebAssembly (以下 WASM) , Rust を選定した。

本説ではこれらの構成技術の選定理由と地図エンジン実装に対する寄与について述べる。

2-7-1. WebAssembly (WASM)

地図エンジンを構成するコンポーネントの中でも、レンダリングエンジンは各プラットフォームにおいて成熟した実装がいくつか存在するため、我々の地図エンジンではこれらのレンダリングエンジンをビジュアル・性能要件に応じてプラガブルに切り替えられるように実装する。

レンダリングエンジン以外のコンポーネントは、レンダリングエンジンのプラガブル化を含む先進的な機能を実現するため、大部分を新規実装する。新規実装部分は、WASM として実装することを計画している。

広く一般に公開される PLATEAU の 3D 都市モデルを可視化するための地図エンジンは Web 環境での動作が必須であり、地図エンジン実装の配布形態は JavaScript 又は WASM が有力な候補となる。また、Web 以外のデスクトップ・モバイルネイティブ環境での地図エンジン動作も見据えると、WASM が最有力の候補となる。

地図エンジンはアプリケーションから利用されるものであり、アプリケーションには Web アプリケーションだけでなく、デスクトップアプリやモバイルアプリといったネイティブアプリも含まれる。WebView や Electron などのフレームワークを用いたネイティブアプリであれば JavaScript 製の地図エンジンを自然に活用することができるが、そうではないネイティブアプリでは JavaScript 製では取り回しが悪い。一方 WASM は WebAssembly System Interface (WASI) と併用することによりネイティブ環境で動作することができ、ネイティブアプリと動的リンクすることで協調することが可能と目論む。

以下、WASM の技術的な特徴に触れ、地図エンジン開発に副次的に与えるメリットについても記載する。

Web のクライアントサイド、すなわちブラウザでプログラマブルな処理を実行するためには、長年 JavaScript が使われてきた。JavaScript の実行エンジンの高速化は目覚ましいが、それでも JavaScript が動的型付け言語である以上、実行時のオーバーヘッドは避けられない。JavaScript よりも高速な処理を実現するために 2017 年に誕生したのが WASM である。地図エンジンの中で性能が体験に最も直結するコンポーネントはレンダリングエンジンであると考えられるが、それ以外のコンポーネントでも WASM に

よる高速化の恩恵はあり得る。情報量の多い地域の属性データ処理やレンダリングエンジンへの入力のためのデータ変換は、JavaScript よりも WASM が性能上優位になることが予想される。

また、WASM は WASI を実装した Wasmer や Wasmtime といったランタイムにより、ブラウザ外で動作することができる。これによりネイティブ環境で動作するレンダリングエンジン等との結合が容易になることは先程述べたとおりである。

さらに、WASM を生成することができるプログラミング言語は多岐にわたるため、開発の自由度が上がる点も注目に値する。WASM ターゲットは Rust, C/C++, Go, Java, Swift, Python など多様な言語で開発することができる。地図エンジン開発のメンテナンス性を考えると不用意に複数言語を混ぜ合わせた開発を行うことは避けるべきと考えるが、言語ごとに特定分野のライブラリが充実している、不足しているなどの事情が存在するため、適材適所で言語をまたげる性質には一定の魅力がある。

以上の理由から、新たな地図エンジンの実装には WASM を採用することが適切だと考えられる。

2-7-2. Rust

我々の地図エンジンの WASM 開発では、Rust をメインのプログラミング言語とする。

言語の選定には開発に携わる人員の経験・嗜好などの要素も多分に影響するが、ここでは非属人的な選定理由について述べる。

WASM のバイナリサイズは小さければ小さいほど良い。我々の地図エンジンは Web 環境での動作を主に想定しているため、地図アプリケーションを利用中に WASM がインターネットからダウンロードされる機会が多くある。サイズは小さいほどユーザーの（モバイル含む）回線への負担が小さくなるし、アプリケーションの体験も向上する。
各プログラミング言語が生成する WASM のバイナリサイズを一般的に比較することは難しい。実現するプログラム次第で利用ライブラリが言語ごとに異なるなど、さまざまな事情に左右されるためである。とはいえ、プログラミング言語が高機能なランタイムを持つかどうかは WASM サイズにベースライン的に影響することが考えられる。WASM を実行する環境にはプログラミングごとのランタイムは原則存在しないため、WASM 自体にランタイムを内包する必要があるからである。この観点からすると、Java, Go, Python などのランタイムがリッチな言語と比べると、Rust や C/C++のような言語のほうが WASM サイズは小さくなる傾向となることが期待できる。ただし、ベンチマーク評価によっては Rust よりも C、さらには Java のほうが WASM サイズが小さくなるという報告[3]も存在するため、WASM サイズの観点では Rust は悪い選択ではないが必ずしも最善ではないと言える。

[3] https://stackoverflow.com/a/55154106

それでも、Rust は C/C++よりも後発の言語であり、メモリ安全性や型安全性と、開発生産性や実行性能の高さを両立し人気を博している。Stack Overflow の人気言語調査において、2023 年まで 8 年間連続で開発者に最も愛されている言語に選ばれている。WASM 開発においても Rust が最も人気のある言語である[4]。歴史ある言語と比べると未だ開発者の絶対数は少ないが、今後の成長が見込まれており、地図エンジンの開発コミュニティ拡大にも寄与すると考える。

[4] https://blog.scottlogic.com/2023/10/18/the-state-of-webassembly-2023.html

2-8. その他の関連技術の詳細調査

WASM, Rust の他に、我々の地図エンジンの実装に利用すると恩恵を受けられる可能性がある技術として、WebGPU（並びに wgpu ライブラリ）、Entity Component System（以下 ECS）、Shared Array Buffer（以下 SAB）を検討した。現状の結論としては、WebGPU（並びに wgpu ライブラリ）と ECS を利用し、SAB は利用しないこととした。

以下、それぞれの調査内容と利用是非についての考察を記載する。

2-8-1. WebGPU, wgpu ライブラリ

2.3 節で、レンダリングエンジンからの GPU の利用と、データ処理等における GPGPU の活用について触れた。

Web ブラウザの環境で GPU を活用する際には、2012 年頃から長年 WebGL が使われてきた。これは OpenGL ES 2.0 を JavaScript へ移植したものであるが、ネイティブ環境で Direct3D 12、Metal、Vulkan などのよりモダンなグラフィックス API が登場する中で、OpenGL のアップデートはストップしてしまった。そこで、Web ブラウザでモダンなグラフィックス API を提供するのが WebGPU である。

WebGPU、Direct3D 12、Metal、Vulkan などのモダンなグラフィックス API は、従来の API に比べ、低レベルの API を提供するため GPU の細かい制御が可能である。また、コンテキストの状態を切り替えながら命令を送る必要があった WebGL に比べ、API が改良されており、リソース管理が効率的に行えるため、レンダリング速度などのパフォーマンスが向上する。

また、機械学習の用途拡大などにより、Web ブラウザ環境でも GPGPU の活用機会が増えている。WebGL は HTML のキャンバス、すなわち描画を前提にした作りであるため GPGPU 的な計算を扱うのは容易ではないが、WebGPU はより低レベルな GPU API を備えており、コンピュートシェーダーをサポートするため、GPGPU 活用が容易になる。

WebGPU の主要 Web ブラウザでの正式サポートは、2024 年 3 月の執筆時点ではまだ少ないが、Chrome は 2023 年 5 月リリースの Chrome 113 で正式にサポートした。

我々の地図エンジンでは、Web ブラウザ環境のレンダリングエンジン以外のコンポーネントで GPU を活用する際は、WebGPU を活用することが適切である。Web ブラウザ環境でのレンダリングエンジンは、Three.js などより高レベルなグラフィックス API をプラガブルに切り替えられるように作成し、そのグラフィックス API が WebGL を使うか WebGPU を使うかは問わないものとする。

WebGPU はあくまで Web ブラウザ環境用のグラフィックス API なので、デスクトップ環境やモバイルネイティブ環境では利用できない。環境差異を吸収する抽象化レイヤーとして、wgpu[5] ライブラリ（Rust のクレート）を活用し得る。wgpu は WebGPU をベースとした統一的な API を備え、その裏側で Web ブラウザでは WebGPU 又は WebGL、Linux/Android では Vulkan、macOS/iOS では Metal、Windows では Direct3D 12 といったように、プラットフォームごとに適切なグラフィックス API を呼び分けることができる。これにより、地図エンジン開発において GPGPU を活用する際や、Three.js などの高レベルなグラフィックスエンジン固有の API に依存しないグラフィックス処理を行う際に、プラットフォームごとに異なるコードベースを持つことを避けることができるものと期待する。

2-8-2. Entity Component System（ECS）

地図エンジンと同様にメインループの仕組みを持つゲームアプリケーションでは、メインループにおいて毎フレームごとに、大量のゲームオブジェクト（例えば主人公・敵・アイテム・ステージなど）の状態を変化させながらゲームを駆動させていくことが多い。地図エンジンにおいてもゲームオブジェクトに相当するオブジェクト（地球・地物・タイルなど）が多数存在し、どのようにしてフレーム更新を最適化させるかは、地図エンジンのパフォーマンス向上のために重要である。

一般的なゲーム開発では、オブジェクト指向プログラミング（OOP）アーキテクチャが採用されている。OOP では、ゲームオブジェクトはデータとそれに関連するメソッドをカプセル化したクラスによって表される。それらを継承することでさまざまな種類の振る舞いを行うゲームオブジェクトを実装し、それらのインスタンス同士がメッセージをやり取りし合うことで、ゲームの状態を更新していく。このアプローチは直感的で理解しやすいが、以下の問題がある。

- 継承の複雑さ: 多重継承や深い継承階層によって、コードの理解とメンテナンスが難しくなる。
- 柔軟性の欠如: あるクラスがさまざまなクラスの基底クラスとなっている場合に、派生クラスに影響を与えることなくロジックの変更をすることが困難となる。また、クラスがデータとメソッドを密接に結合しているため、一部の機能を分離して別のオブジェクトで再利用することも困難である。
- パフォーマンスの問題: インスタンスごとにメモリが確保されその領域にはメンバ変数が並ぶため、メンバ変数の型がそれぞれ異なる場合には Position, Rotation, ... のような非均質なメモリ配置になる。それによりオブジェクト間のデータアクセスが分散し、メモリのランダムアクセスが発生したり、CPU キャッシュ効率が低下したりする。また、クラスのメソッド呼び出しには動的ディスパッチを必要とし、大量のクラスを扱う上でオーバーヘッドとなりうる。

一方、近年のゲーム開発において採用されている新しいアーキテクチャとして、Entity Component System（以下 ECS）がある。ECS は、ゲーム開発におけるアーキテクチャパターンの一つで、従来の OOP

[5] https://github.com/gfx-rs/wgpu

と異なる「データ指向」と呼ばれるアプローチを採用している。

ECS には以下のような概念が存在する。

- エンティティ（Entity): ゲーム内のオブジェクトを表す。これ自体には機能はなく、単なる識別子として機能する。
- コンポーネント（Component): エンティティに付与されるデータの集まり。位置、速度、色などの属性を持つ。
- システム（System): ゲームのロジックを担当する。特定のコンポーネントを持つエンティティに対して操作を行う。例えば、「移動システム」は位置と速度のコンポーネントを持つエンティティを対象にして、その位置を更新する。

登録されたシステムがメインループで実行されることで、対象となるコンポーネントが更新される。

ECS の利点は、以下の通りである。

- データとロジックの分離：コンポーネントとシステムの分離により、データとロジックが独立しているためにコードのモジュール化が進み、影響範囲を抑えながら新機能追加や既存機能の変更が行いやすい。また、同じコンポーネントを他のエンティティに再利用しやすく、テスト・デバッグ・メンテナンスが容易。
- 柔軟性の向上：ECS では、エンティティは単なる識別子であり、コンポーネントによって機能が定義される。これにより、新しい機能の追加や既存機能の変更が柔軟に行える。特定の機能はコンポーネントとして独立していて、異なるエンティティに簡単に適用できる。また、システムはこれらのコンポーネントを操作することでゲームのロジックを実現するが、これも個別の機能として扱うことができるため、柔軟なシステム設計が可能。
- パフォーマンスの最適化：ECS では、System は必要なコンポーネントに対してのみ操作を行う。また、同一のコンポーネントを連続領域に配置することで同一型のデータがメモリ上で連続的に配置されることが多くなり、System によってデータ処理が時間局所性高く行われ、CPU のキャッシュラインに「次に処理するデータ」が乗っていることが多くなる。これによって CPU キャッシュの利用効率が向上し、キャッシュミス（必要なデータがキャッシュになく、相対的にアクセス速度の遅いメモリからデータを取得すること）が減る。またゲームのロジックが細かく分離されているため、データ競合やロックが少なく、マルチスレッド化しやすいため、システムリソースの効率的な利用が可能。

ゲームエンジンなどでの ECS の採用事例としては、Bevy、A-Frame、ECS for Unity が存在する。また、地図エンジンにおける ECS の採用事例は、執筆時点では見つけることができなかった。現存の地図エンジンでは、基本的にオブジェクト指向の考え方が用いられている。

2-8-3. Shared Array Buffer (SAB)

Web ブラウザには Web Workers という、JavaScript やそこから駆動される WASM をマルチスレッド処理するための仕組みが存在する。

Linux 等の OS におけるプロセスのメモリモデルでは、スレッド同士はメモリ空間を共有しており、それ故にスレッド間での高速なデータ授受が可能となっている。一方で Web Workers のスレッド間は、原則的にメモリ空間を共有しておらず、ブラウザの postMessage() API[6]などのメッセージパッシングの仕組みでデータ授受を行う必要がある。メッセージパッシング方式では、送信するデータのコピーが発生することなどが要因となり、共有メモリ方式よりもスレッド間データ授受の性能が劣ることが一般的である。

そこで SAB を補足的に利用することを検討した。SAB を使えば Web Workers のスレッド間でメモリ空間を共有することができ、複数スレッドでの高性能なデータ処理に寄与する可能性があるためである。

しかしながら、現在の Web ブラウザで SAB を使うためには強い制約がある。具体的には、SAB はクロスオリジン分離環境でなければ使えない。

現在の Web ブラウザでは、セキュリティを強化するためにオリジンの概念が導入されている。オリジンとは簡単に言えば「サイト」の単位であり、「同一のオリジンから取得したリソース同士であれば信頼し合える事が多い（同一の開発主体が提供しているものなので）」「あのオリジンは提携先なのでリソースを信頼するが、それ以外のオリジンのリソースは信頼しない」というような区分を実現するために導入されたものである。

現在の Web ブラウザは、異なるオリジンのリソース同士が干渉し合うこと（例えば、あるリソースのスクリプトを起因とした別リソースの情報窃取）を避けるために、異なるオリジンの Web アプリケーションを異なる OS プロセスで駆動し、オリジン同士のメモリ空間を分離することを志向している。これが実現された状況のことをクロスオリジン分離環境と呼ぶ。クロスオリジン分離環境でない場合、Spectre という著名な脆弱性等により、異なるオリジンの情報窃取が成立し得ることが知られている。

一方で歴史的な経緯やブラウザごとの実装差異などを背景として、意識せずに実装された Web アプリケーションではクロスオリジン分離環境にならない。アプリケーションのサーバー側で特定の HTTP ヘッダを設定することにより、初めてクロスオリジン分離環境となる。

クロスオリジン分離環境で動作するアプリケーションは決して多くはない中、我々の地図エンジンが SAB を前提とするということは、クロスオリジン分離環境以外の Web アプリケーションをサポートしな

[6] https://developer.mozilla.org/ja/docs/Web/API/Window/postMessage

いということにつながる。クロスオリジン分離環境の Web アプリケーションにだけは SAB を使った実装を、そうでないアプリケーションには SAB を使わない実装を、動的に切り替えて提供するということも可能ではあるが、相応の実装・メンテナンスコストが要求される。

クロスオリジン分離環境となる条件を緩和し、より多くのアプリケーションでクロスオリジン分離環境が得られることを目指すブラウザ標準のディスカッションも見受けられる[7]が、大きな進展はないと評価した。

以上の事情をもって、少なくとも開発当初は、本業務で開発する地図エンジンでは SAB を活用しないこととした。

[7] https://github.com/whatwg/html/issues/6364

3. 地図エンジンのプロトタイプの開発

3-1. プロトタイプの要件定義

本章では、前述の調査内容及び設計を基にプロトタイプを実際に実装することで、新しい地図エンジンの実現可能性を確認する。

まず、第 1 章で述べた地図エンジンの課題を再度整理する。

- ビジュアル：地図エンジンがレンダリングも内蔵していることから、ビジュアルに関するカスタマイズが困難、新技術や新ハードウェアへの柔軟な対応が困難。
- パフォーマンス：大規模なデータを読み込むとフレーム落ちが発生し、ユーザー体験が悪化することがある。
- マルチプラットフォーム：Web 以外のさまざまなプラットフォームで動作するアプリケーションの開発が難しく、新技術や新ハードウェアに柔軟に対応できない。

以上の課題を解決するために、新しい地図エンジンでは以下の仮説を立て、アーキテクチャを設計する。

- ビジュアル:地図エンジンのアーキテクチャ設計を見直す。レンダリングエンジンとそれ以外の CPU バウンドな処理（メインループや GIS に関する演算）を分離し、レンダリングエンジンを切り替えられるようにする。これにより各プラットフォームやハードウェアの特性に対して対応力が向上し、地図エンジンを利用しながらも、よりリッチなビジュアルを作り込むことがこれまでより容易になったり、新技術に対応しやすくなったりする。また、用途によってはレンダリング自体を省略することもでき、必ずしも絵を描画しなくとも仮想的に地図エンジンを動作させることが可能である。例えばレンダリングは自前で行うゲームアプリケーションやサーバー上で仮想的な計算のみを行い、自動運転のシミュレーションなどさまざまな環境や用途に組み込みやすくなる（ヘッドレスな地図エンジン）。
- パフォーマンス:データ処理などの重い処理を担う部分は、マルチプラットフォーム性も加味して、多くの環境で同一の実装が高速に動作する WASM と Rust を選定する。WASM は JavaScript と違いバイトコードであることや、Rust で開発することで、Garbage Collection（GC）を省きメモリアクセスを削減できるため、高速化が見込める。また可能な限り、マルチスレッドなどの技術を用いて、メインループの処理を軽量化し、データ処理を高速化する。
- マルチプラットフォーム：レンダリング以外の CPU バウンドな計算が必要であるメインループと GIS エンジンは、パフォーマンスとマルチプラットフォーム性を重視し、配布形態として WASM を、開発言語として Rust を選定する。WASM はさまざまなプラットフォームで高速に動作する実行可能バイナリの形式であり、WASM を Web ブラウザ外でも実行可能にする WASI という技術と組み合わせることで、OS などを問わず地図エンジンをさまざまな環境に組み込みやすくなると見込まれる。

これらの構想をもとに、新たな地図エンジンでは以下のようなアーキテクチャを考案する（図 7）。

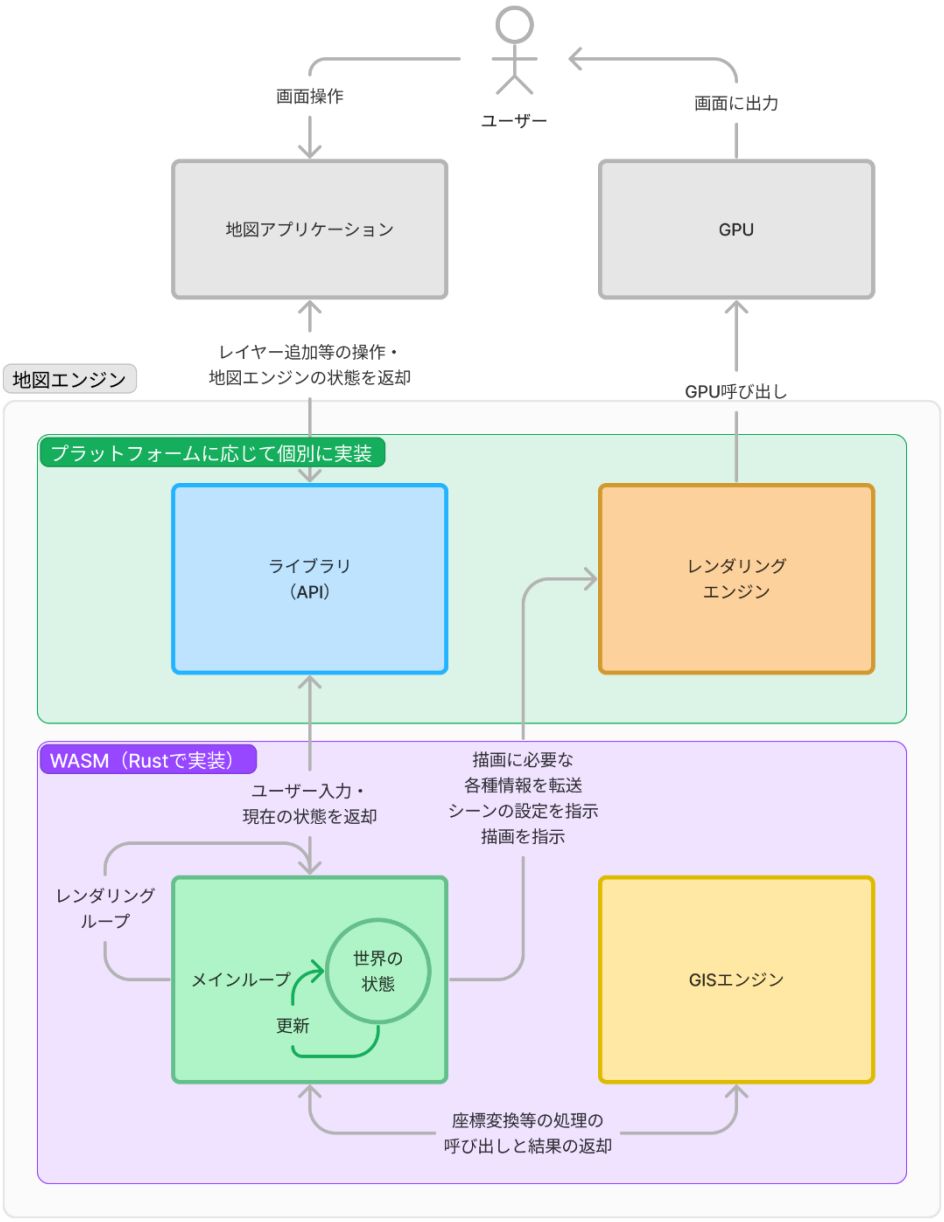


図 11　地図エンジンのアーキテクチャ

さらに、以上の仮説から、以下のような追加のメリットも得られる。

- 開発生産性：第 2 章で述べたように、開発言語として Rust を採用することで、メモリ安全性とパフォーマンスを両立できる。堅牢な型システムも備え、これまでさまざまなミドルウェア開発者の頭を悩ませてきた、メモリ管理に起因する数多くのバグを事前に防ぐことができる。また、Cargo と呼

ばれるパッケージマネージャーも付属し、依存ライブラリの管理がしやすい。

- ポータビリティ：WASM （WASI）を採用することで、実行環境を問わず高速実行可能なバイナリを配布可能である。将来的にはゲームアプリケーションや CDN（エッジ）、その他 WASM によるプラグインの仕組みを備えたアプリケーションなど、より多様な環境でも地図エンジンが動作するようになることが期待される。
- 拡張性：将来的には WASM によって地図エンジンを拡張するようなプラグインの仕組みを導入するという可能性も考えられる。WASM はさまざまなプログラミング言語からコンパイル可能であり、WASM はサンドボックス内で実行され比較的安全に第三者のコードを実行することができるため、プラグインの仕組みを実現するのに都合が良い。

その上で、今回のプロトタイプ開発では、特に技術的に新規性が高い部分である「CPU バウンドな処理とレンダリングエンジンを分離し、レンダリングエンジンがプラガブルな形で動作するような地図エンジン」が実際に開発可能か確かめることを重視する。

プロトタイプのレンダリングエンジンはゼロから開発するのではなく、Three.js を用いてレンダリングを行うことにする。Three.js は第 2 章で述べたように、JavaScript において有名な 3DCG レンダリングライブラリであり、WebGL の難しさを覆い隠して人間にとって分かりやすい API を提供しながら、シェーダーなどの細かいカスタマイズもしやすいのが特徴である。コミュニティの資産も豊富で、地図エンジンの Web におけるレンダリングエンジン実装の 1 つとして今後も有力株である。

その上で、本プロトタイプでは、現在の PLATEAU VIEW において実現している機能のうち、最も基本的な以下の 5 つの機能を実現することを目指す。

1. 地球の表示
2. マウス操作によるカメラの移動
3. 地図タイルの表示（一部の範囲・一定のズームレベルで表示）
4. 地形の表示（一部の範囲・一定のズームレベルで表示）
5. 3D Tiles・MVT の表示（建築物モデル・植生モデル・道路モデルの表示）
6. シェーダーの実装による新たな表現に基づくデータ可視化

3-2. プロトタイプのアーキテクチャ設計

3-2-1. プロトタイプのモジュール設計

本プロトタイプでは、前節で述べたアーキテクチャの実現可能性を確認するため、さまざまな処理を以下のモジュール（Rust ではクレートと呼ぶ）に分離して実装を行う。

- map-engine-core: Rust で実装される、地図エンジンに必要な処理を担うライブラリ。map-engine-ecs から呼び出される。座標の変換や、タイルのジオメトリ（頂点・インデックス・UV）の計算を担う。
- map-engine-ecs: Rust で実装される、地図エンジンの中核となる部分。2.2 節で述べた GIS エンジンとメインループに相当。ECS を採用し、ECS 上でメインループを実行する。
  ECS のライブラリとして、Rust 製のゲームエンジンである Bevy を部分的に採用した。Bevy は多数のクレートの集合から成るライブラリであり、一部の機能だけを使用することができる。ここでは Bevy ECS のみを使用するため、クレートとして主に bevy-app・bevy-ecs を使用した。
- map-engine-wasm: map-engine-ecs を呼び出す、Web ブラウザのための薄いクレート。map-engine-ecs と map-engine-threejs の間をつなぐ役目を果たす。
  Rust で実装され、wasm-pack によってビルドされ、WASM バイナリだけでなく JavaScript のグルーコードや TypeScript の型定義を出力する。これにより JavaScript から呼び出し可能なライブラリがビルドできる。
  なお、今後 WASI に対応するには、本クレートとは別で map-engine-wasi のような別のクレートを作成し実装が必要である。
- map-engine-threejs: Three.js によるレンダリングエンジンを TypeScript で実装した、JavaScript のライブラリ。Three.js の初期化や WASM の呼び出しを隠蔽する。地図エンジンの利用者（地図アプリケーション開発者）はこのライブラリを使用して地図エンジンの機能にアクセスするという想定である。加えて本プロトタイプ、map-engine-threejs を使用するデモアプリケーションを TypeScript で実装し、そのコードも含む。

ここで、WASM 上で動作するクレートは、map-engine-core、map-engine-ecs、map-engine-wasm の 3 つである。本稿では便宜上これらをまとめて「WASM 側」と呼称する。一方で、JavaScript の世界で動作するモジュールは map-engine-threejs である。これを便宜上「JS 側」又は「レンダリングエンジン」と呼称する。

各モジュールの責務と依存関係を図に表すと以下の通りである。

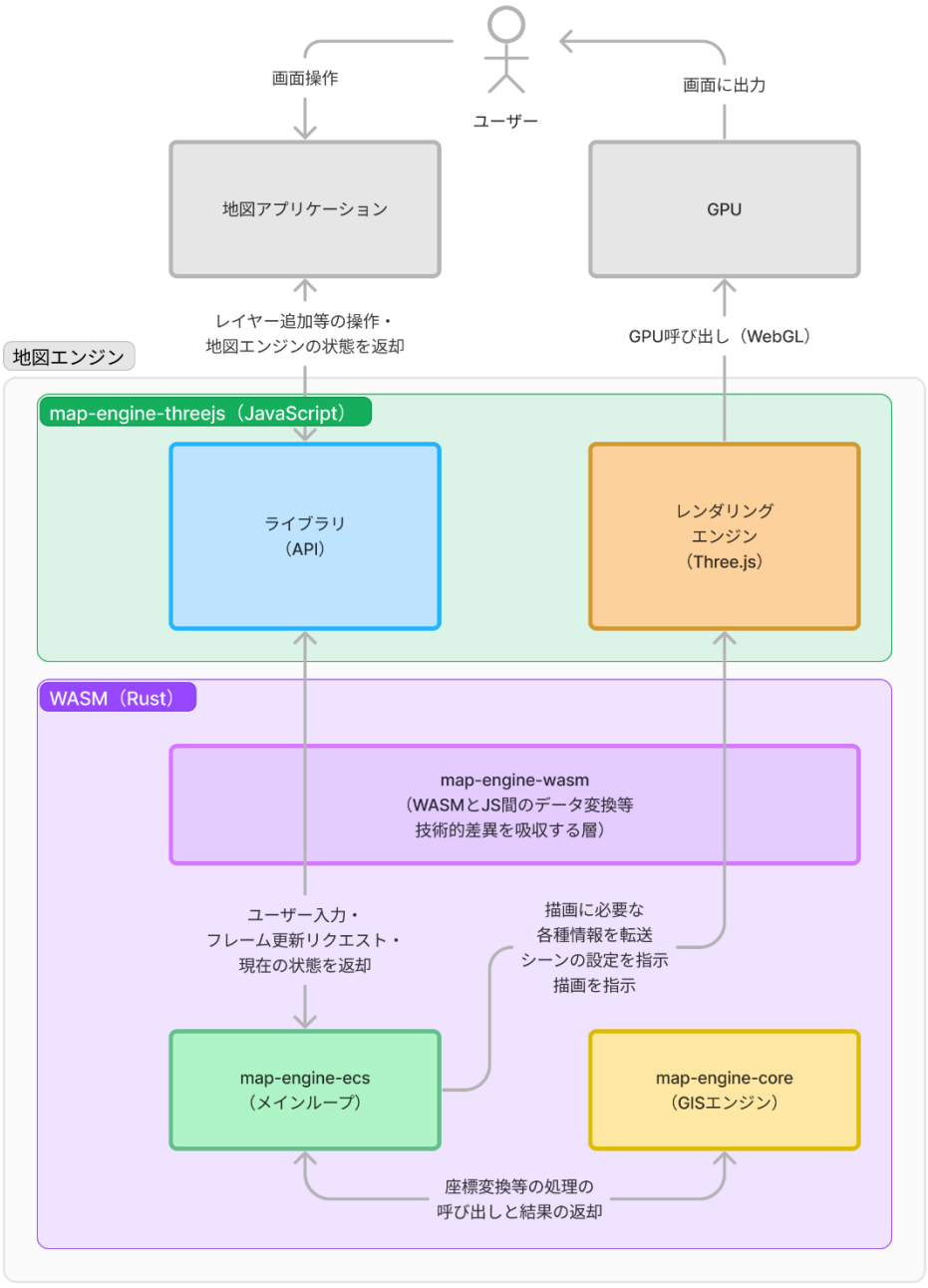


図 12　プロトタイプにおけるモジュールの責務と関係

3-2-2. ECS の組み込み

本プロトタイプでは、ECS を用いてメインループの処理を実装する。ECS のライブラリとして、Bevy ECS を採用した。Bevy は Rust 製のゲームエンジンであるが、多数のクレートの集合体であり、一部機能のみを使うこともできる。

本プロトタイプでは、WASM とレンダリングエンジンを分離する。すなわち、レンダリングは WASM 内部のメインループでは実行されず、WASM の外側で実行される必要がある。

Bevy ECS では、Entity が持つデータとして Component を、各種 Component を更新する処理内容として System と呼ばれる関数を実装することができ、実装した System を App と呼ばれる概念に登録する。App は System の集合と、各種 Entity や Component を含む世界の状態を表す World を保持し、フレーム更新を発生させる update メソッドを持つ。メインループが駆動すると App の update メソッドが呼ばれ、App に登録された System が自動的に実行され、さまざまな Component を持つ Entity が必要に応じて作成・更新・削除され、App が持つ World の内容が変更される。

通常 Bevy では、World を更新するための System が実行された後、それらの情報を用いて bevy-render というクレートが持つ多数の System がレンダリングエンジンとして動作し、ECS 上でレンダリングまでを行う。ところが、本プロトタイプでは WASM とレンダリングエンジンを分離することから、その方法が採用できない。

加えて、メインループを回し続ける処理を WASM 内部で行うようにしてしまうと、メインループを一定間隔で呼び出し続けるために、時刻などの情報にアクセスする、時計に関する機能を WASM 側が持つ必要が出てきてしまう。時計は OS から提供されるハードウェア依存の機能であり、WASM 環境下では直接使用できない。WASM 側は OS やハードウェアに非依存である必要があり、メインループの実行を WASM で行うことはマルチプラットフォーム性の観点から適切ではない。

そこで以下のようなフローで、WASM とレンダリングエンジンがお互いに情報をやり取りするように設計した。WASM はメインループそのものは実行せず、レンダリングエンジン側でメインループを回す。ただし World に関する情報は WASM が保有しており、メインループが WASM に対して更新をリクエストすると、WASM 内部で ECS が実行され World が更新される。

以下はメインループにおける 1 フレーム分の更新開始からレンダリング完了までの処理の流れを示したシーケンス図である。

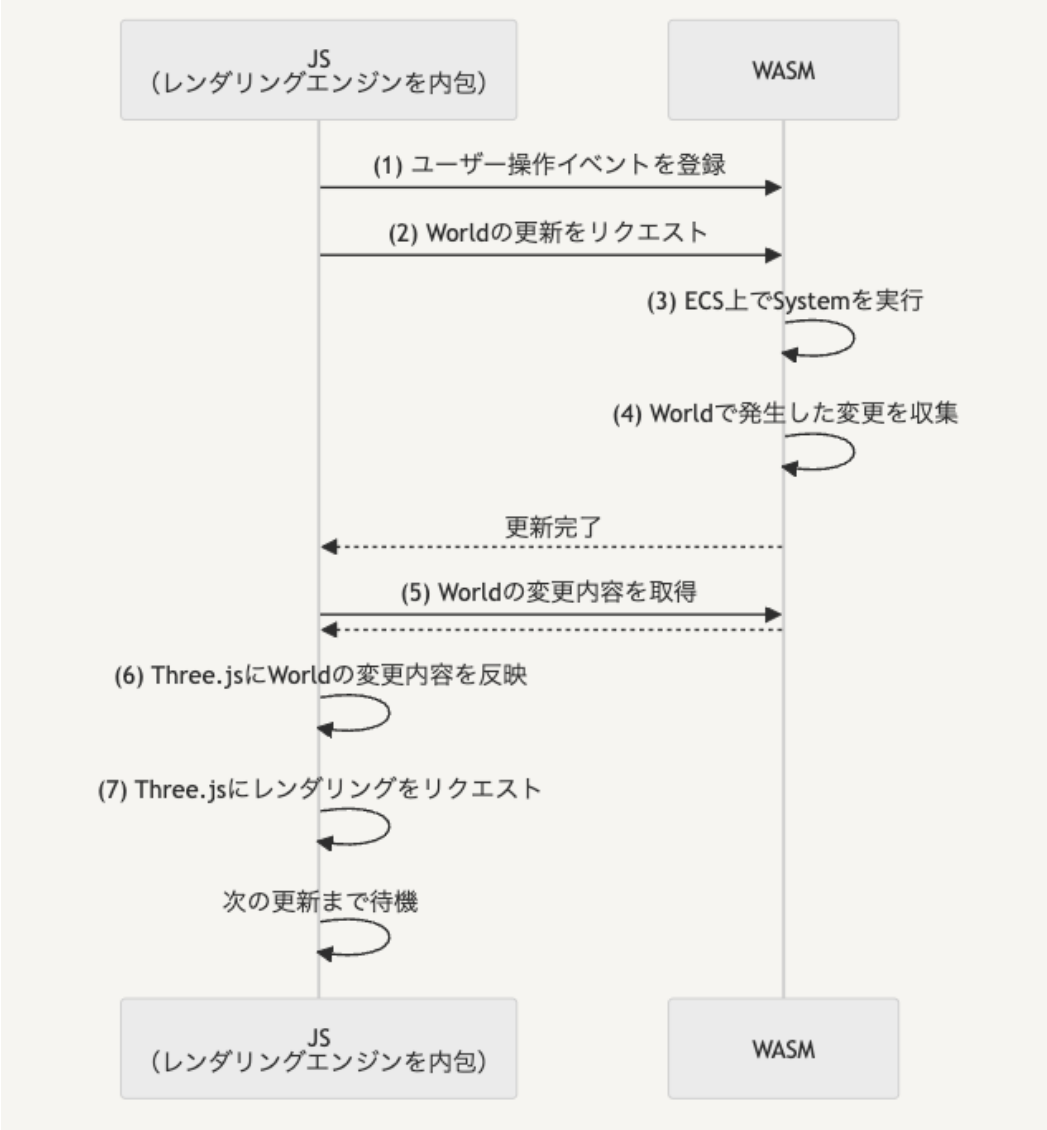


図 13　シーケンス図

シーケンス図の内容について補足する。

1. フレーム更新前に、ユーザーからマウスイベントなどの入力を受け取ったら、その都度 WASM 側に通知し、WASM 内部のイベントキューにイベントを蓄積する。
   これはメインループのフレーム更新処理とは無関係に随時実行される。例えば Web ブラウザであれば「マウスがクリックされた」「マウスカーソルが動いた」などのイベントが発生したときに、そのイベントに関する情報（どのボタンが押されたか、カーソルが何 px 動いたかなど）をそのまま WASM に通知し、WASM はその情報を自身が持つイベントキューに溜めておく。やがて実行される次のフレーム更新時に、ECS 内部で登録されたイベントのキューを取り出していって、ユーザー入力に反応した処理を実行する。これによりユーザー操作に対して地図エンジンが反応することができる。
2. JS 側が WASM 側に対しフレーム更新をリクエストする。
3. WASM は ECS を実行し、登録された System を実行していきながら、WASM が持つ World の状態

を更新する。このとき 1 で蓄積されたイベントも順次処理される。
4. World に対して発生した変更を全て取得し、その情報を WASM が保持する。例えば、「カメラの位置や向きの変更」「新しいオブジェクト（メッシュとマテリアル）の追加」「オブジェクトの位置や向きの変更」「オブジェクトの削除」などのイベントを検出する。
5. フレーム更新完了後、JS 側は再度 WASM に対して問い合わせを行い、前回のフレーム更新時に発生した変更内容を全て取得する。WASM は 4 で収集された内容を返す。
6. 取得した変更内容を基に、レンダリングエンジンにその変更を反映させる。今回のプロトタイプはレンダリングエンジンに Three.js を採用していることから、Three.js のカメラオブジェクトのプロパティを更新したり、オブジェクトを新規作成してシーンに追加したりするといった処理を行う。
7. レンダリングエンジンに対してレンダリングをリクエストすることで、画面上に最新の画面が反映される。

以上の処理により、WASM 内部の World とレンダリングエンジンの状態の同期が取れ、最新状態がレンダリング結果に反映されるようになる。

3-2-3. WASM とレンダリングエンジンの責務の分離

アーキテクチャ設計上重要なのは、どういった抽象度で WASM 内部の World の変更差分をレンダリングエンジンに伝えるかである。

例えば、レンダリングエンジンに要求する命令が「立方体を描画する」といった抽象度なのか、「頂点バッファを描画する」といった抽象度なのかといった違いがある。一方、もし GPU での描画に必要なシェーダープログラムの生成までを WASM が担い、その結果をレンダリングエンジンに対して送信するような設計にしてしまうと、シェーダーは環境依存であるため動作しない GPU や環境が出てきしまうことが考えられることから、結果 WASM がハードウェアに依存した実装となってしまう。また、WASM がビジュアルの品質にまで責務を負ってしまうと、レンダリングエンジンの責務は非常に小さくなってしまう。

最終的に、以下のように決定した。

- ジオメトリ：WASM 内部で、頂点バッファやインデックスバッファなど、ジオメトリを構成するのに必要な計算を行い、レンダリングエンジンに渡す。レンダリングエンジンはその結果を GPU に転送することが主な責務であり、追加の計算はほとんど行わない。バッファの計算はハードウェア非依存の CPU バウンドな処理が多く、マテリアルとは独立して計算することができるため、WASM の責務に適する。
- マテリアル：WASM 内では、カラー・ラフネス・メタルネスといった抽象的なパラメータを扱い、レンダリングエンジンでそのパラメータに基づき実際の描画を行う。最終的にどのようなシェーダーを用いるかや、ポストプロセッシングなどの処理は、レンダリングエンジンに委ねられる。これにより、レンダリングエンジン側にさまざまなハードウェアをサポートする柔軟性も持たせることも

できる。

この抽象度を採用することで、今後地図エンジンを Three.js 以外のレンダリングエンジンにも対応させることが可能と考える。

3-3. プロトタイプの実装結果と考察

これまで述べた設計を踏まえ、実際にプロトタイプを実装して動作確認を行い、以下の結果が得られた。また、そこから得られた課題も述べる。

3-3-1. 地球の表示

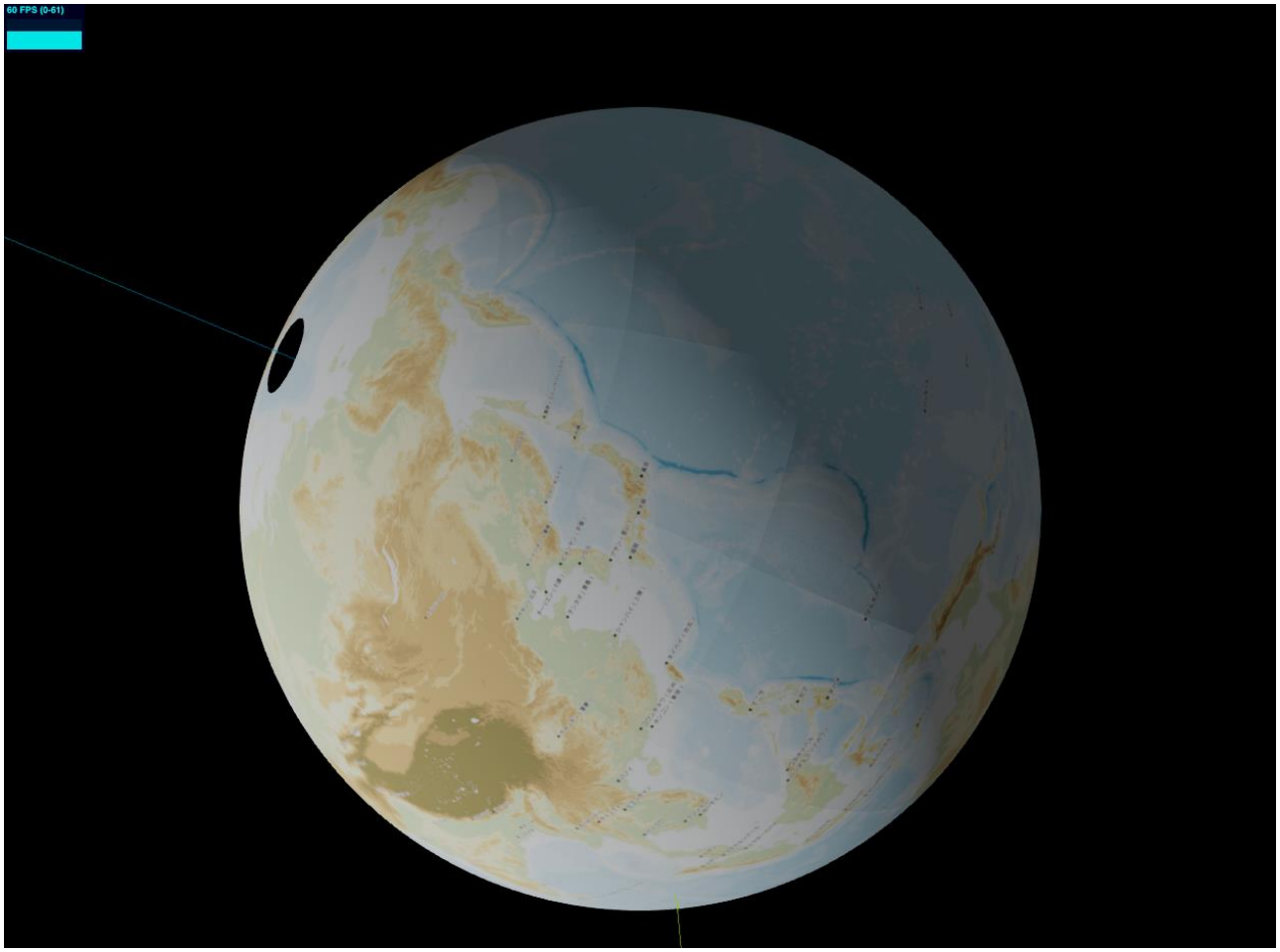


図 14　タイルを描画することで地球が見えている様子

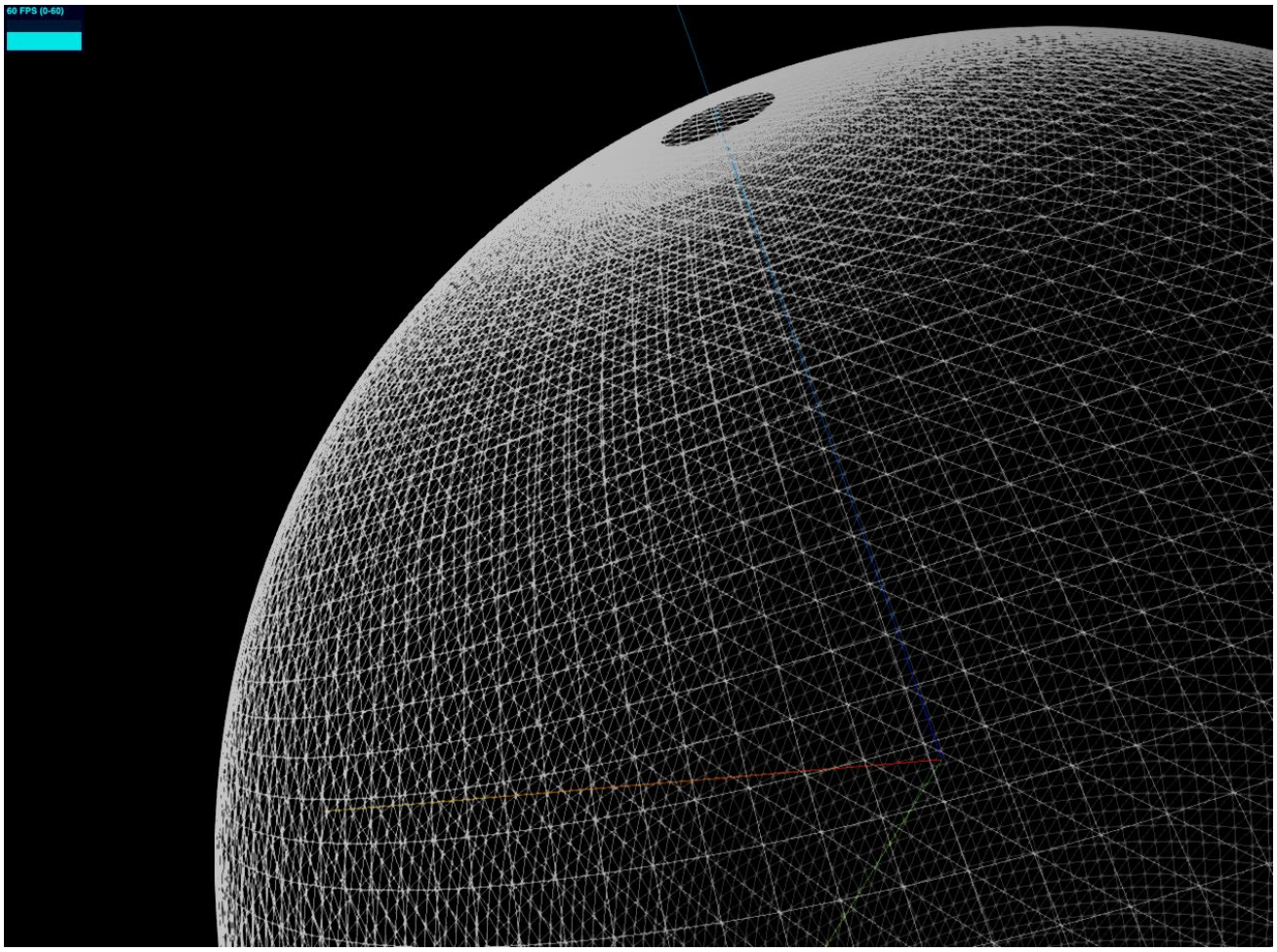


図 15　各タイルのワイヤーフレームを表示した様子

ここでは、地球の楕円体を表す単独のジオメトリを描画するのではなく、ズームレベル 5、10 セグメント（1 タイルあたりの分割数）のタイルを敷き詰めることで、結果的に地球全体を描画することができた。ここでは地図タイルに国土地理院が配信するタイルを使用した。なお、北極付近に穴が空いているが、これは極付近は Web メルカトルの座標系では座標を表すことができず、タイルが存在しないためである。

なお、ズームレベル 5 では、2^5^2=32*32=1024 枚のタイルが存在する。

各タイルの頂点バッファ・UV バッファ・インデックスバッファの計算は WASM 内部で行なっている。すなわち、JavaScript 側でタイルがレイヤーとして追加されると WASM 側に通知され、次回フレーム更新時に、WASM 側でジオメトリの計算を行う。まずズームレベルを基にタイルが経度方向及び緯度方向にそれぞれいくつ存在するのかを計算し、ループを回して、XYZ の 3 パラメータで位置が決定されるタイルごとにジオメトリ（頂点バッファ・UV バッファ・インデックスバッファ）を計算する。具体的にはセグメント分のループを回し、Web メルカトル上の座標を少しずつずらしながら、その場で XYZ（地心直交座標系）への変換を行い、その結果を頂点バッファに書き込んでいく。以下はその実装のコードである。

```
pub fn tile_triangles<F: Fn(usize, usize) -> f32>(
    ellipsoid: Ellipsoid<f32>,
    segments: usize,
    // { west: Rad<f32>, east: Rad<f32>, north: Rad<f32>, south: Rad<f32> }
    extent: Extent<f32, Radians>,
    height: F,
) -> Geometry {
    let segments = if segments == 0 { 1 } else { segments };

    let verties_count = (segments + 1) * (segments + 1);
    let mut vertices = Vec::with_capacity(verties_count);
    let mut uvs = Vec::with_capacity(verties_count);
    let mut indices = Vec::with_capacity(segments * segments * 6);

    let dlng = (extent.east - extent.west) / segments as f32;
    let dlat = (extent.north - extent.south) / segments as f32;

    for i in 0..=segments {
        for j in 0..=segments {
            let lle = LLE {
                lng: extent.west + dlng * i as f32,
                lat: extent.south + dlat * j as f32,
                height: Meters::new(height(i, j)),
            };
            let xyz = lle.to_xyz(ellipsoid);

            vertices.push([xyz.x.val(), xyz.y.val(), xyz.z.val()]);
            uvs.push([i as f32 / segments as f32, j as f32 / segments as f32]);
        }
    }

    for i in 0..segments {
        for j in 0..segments {
            let a = i * (segments + 1) + j;
            let b = (i + 1) * (segments + 1) + j;
            let c = b + 1;
            let d = a + 1;
```

```
                indices.push(a as u32); // a -> b -> d
                indices.push(b as u32);
                indices.push(d as u32);
                indices.push(b as u32); // b -> c -> d
                indices.push(c as u32);
                indices.push(d as u32);
            }
        }

        Geometry {
            vertices,
            uvs,
            indices,
        }
}
```

なお、本プロトタイプでは一律のズームレベルで全てのタイルを描画している。そのため、執筆者の環境（MacBook Pro, M1, 2020）では、ズームレベルを 6 以上にすると頂点数が爆発的に増加し、フレームレートが著しく低下した。これは頂点数やドローコールが多すぎるためで、これを回避するには、既存の地図エンジンで実装されているように、カメラの位置や向きに応じて見えないタイルをレンダリング対象から省く（カリング）とともに、カメラに近いタイルはズームレベルを大きくしてより詳細なタイルを表示し、遠いタイルは低いレベルで大きなタイルを表示するように、最適化することが必要である。ただし、どのタイルがどれくらいのズームレベルで表示するべきかを適切に決定するには、各タイルの SSE（Screen Space Error）を計算し閾値と比較するような、より複雑な処理を実装する必要がある。

また、隣り合うタイル同士で陰影の差が現れ、タイルの境界が見えることがある。これは各タイルのエッジにある頂点の法線が隣のタイルの頂点の法線と一致しないためである。これを防ぐため、ジオメトリを計算し法線を計算した後に、エッジ上の頂点の法線を補正することが必要である。

3-3-2. マウス操作によるカメラの移動

上記のスクリーンショットでは、マウスをドラッグすることでパン（地球の周りをぐるぐる回る）、ホイールを操作することでドリー（地球に向かって近づく又は遠ざかる）の操作が行える。またマウスの右ボタンを押しながらドラッグすることで、カメラをその場で回転することができる。

実際には前述のような処理の流れで、ユーザーの入力を受け付け、カメラの位置が計算され、最終的な画面に反映される。すなわち、ユーザーのマウスによる操作を行うとマウスの位置などの情報が WASM 側に通知される。次にフレーム更新が行われるときに、WASM 内部で、ユーザーから受け付けたマウス操

作をもとに、新しいカメラのトランスフォーム（位置を表す Vector3・回転を表すクォータニオン・スケールを表す Vector3 の組み合わせ、又は 4 次元の回転行列）が計算され、その結果が JS 側に通知される。フレーム更新が終わると JavaScript 側で、Three.js のカメラのトランスフォームを変更する。

このように、ユーザー入力を受け付けて WASM で計算を行い、その結果を Three.js に反映させるという一連の流れが実際に実装できることを確認した。カメラの操作中でもフレームレートにも影響はなく、十分に高速に動作していることが確認できた。

3-3-3. 地図タイルの表示

図 16　横浜市周辺のタイルを描画した様子

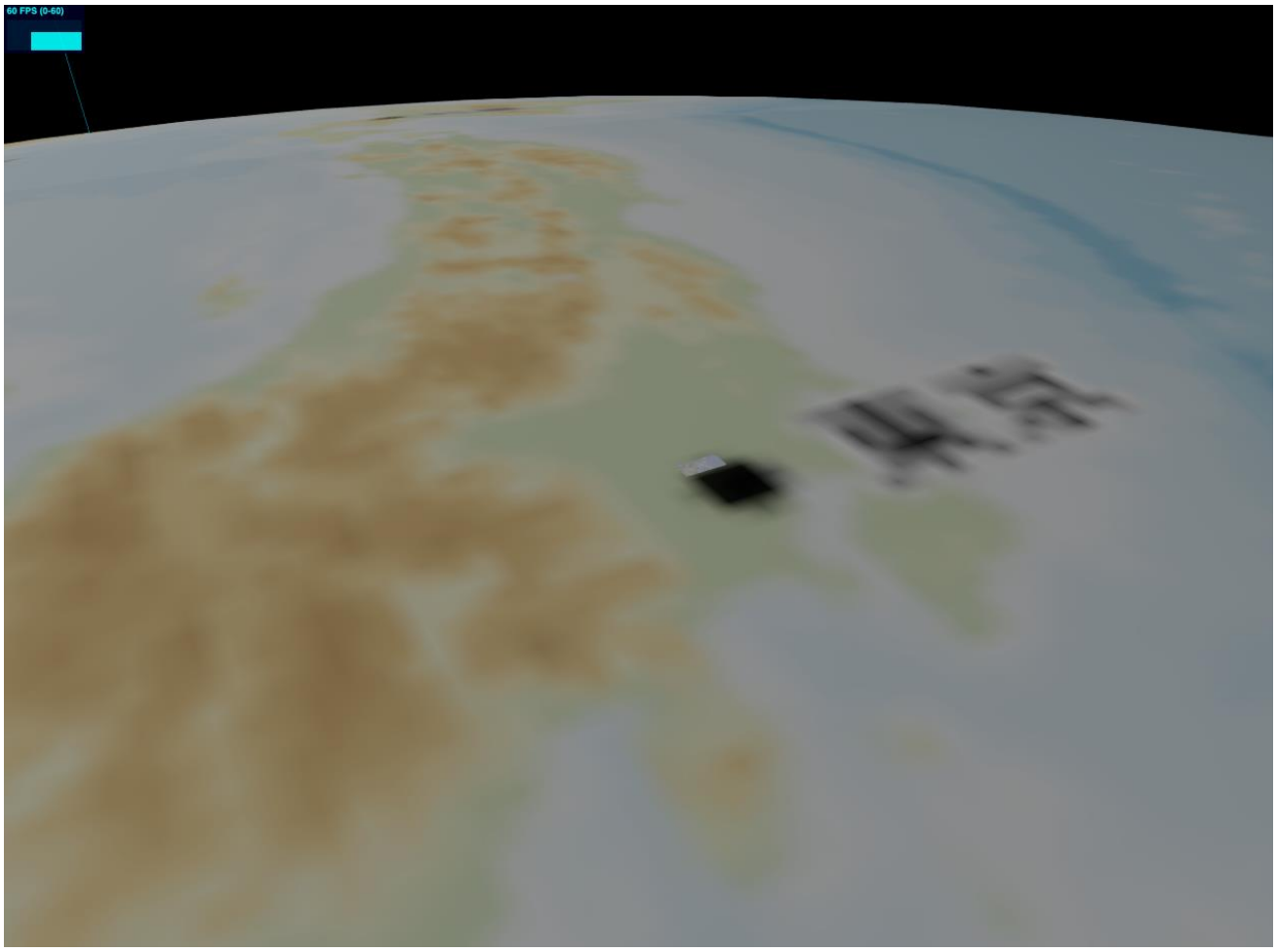


図 17　（ぼやけていて分かりづらいが）画面中央にある横浜市のタイルが地球上の正しい位置に描画されていることが確認できる様子

前述した地球の表示では、地球全体のタイルの描画を行なったが、さらにより詳細なズームレベルでタイルの描画を試みた。ただし本プロトタイプでは、一定のズームレベルで一律にタイルを描画するため、タイルを描画する地域を限定しないとフレームレートが悪化する。ここでは横浜市内の地域のタイルをズームレベル 12 で描画した。

このタイルは前述と同様の計算で Web メルカトルから地心直交座標系に変換しているため、遠ざかって閲覧し既に描画している地球全体のタイルと比較すると、このタイルは地球上の正しい位置にタイルが表示されていることが確認できる。

3-3-4. 3D Tiles・MVT の表示

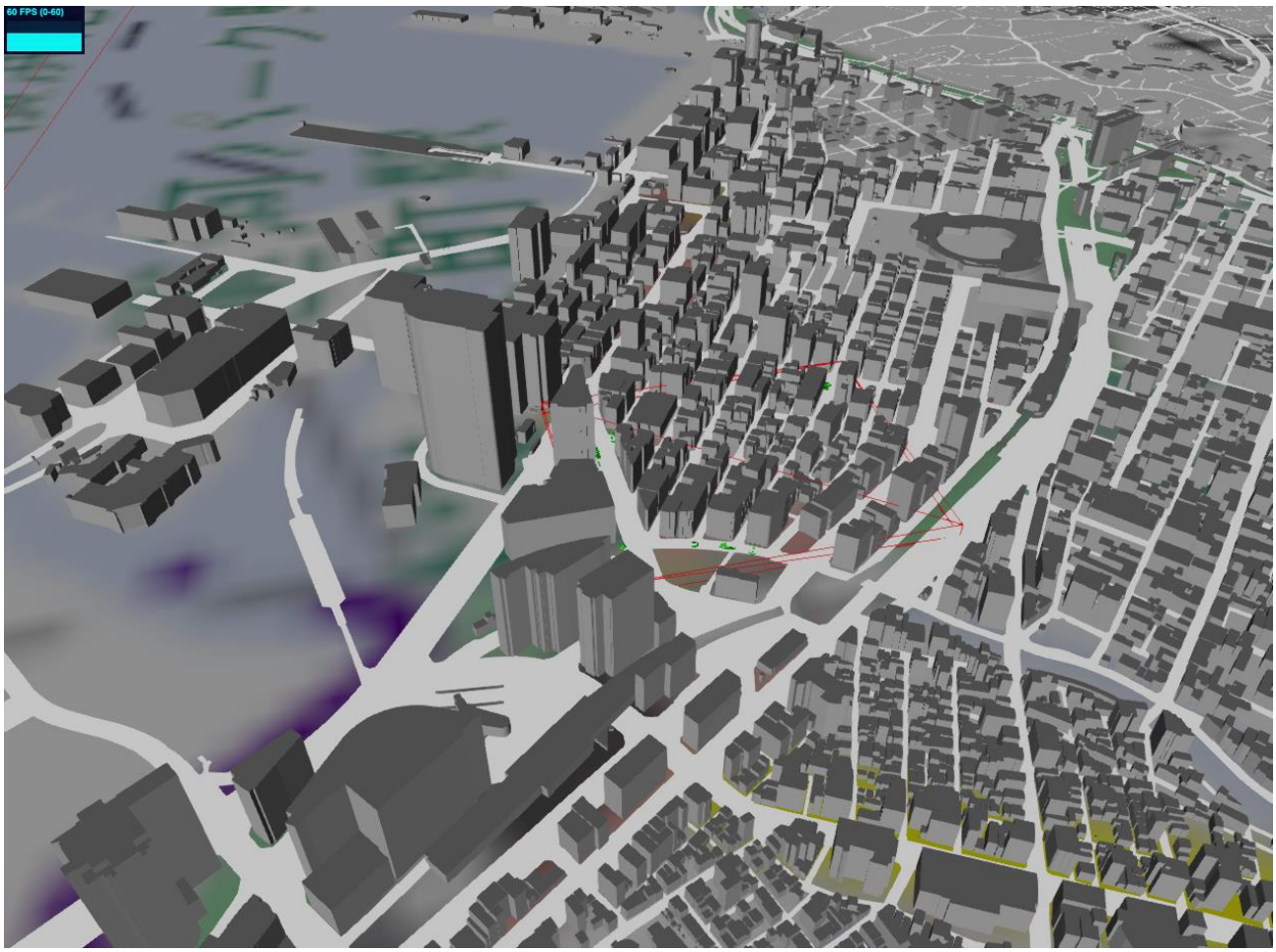


図 18　横浜市の建築物モデルと道路モデル（白いポリゴン）が描画されている様子

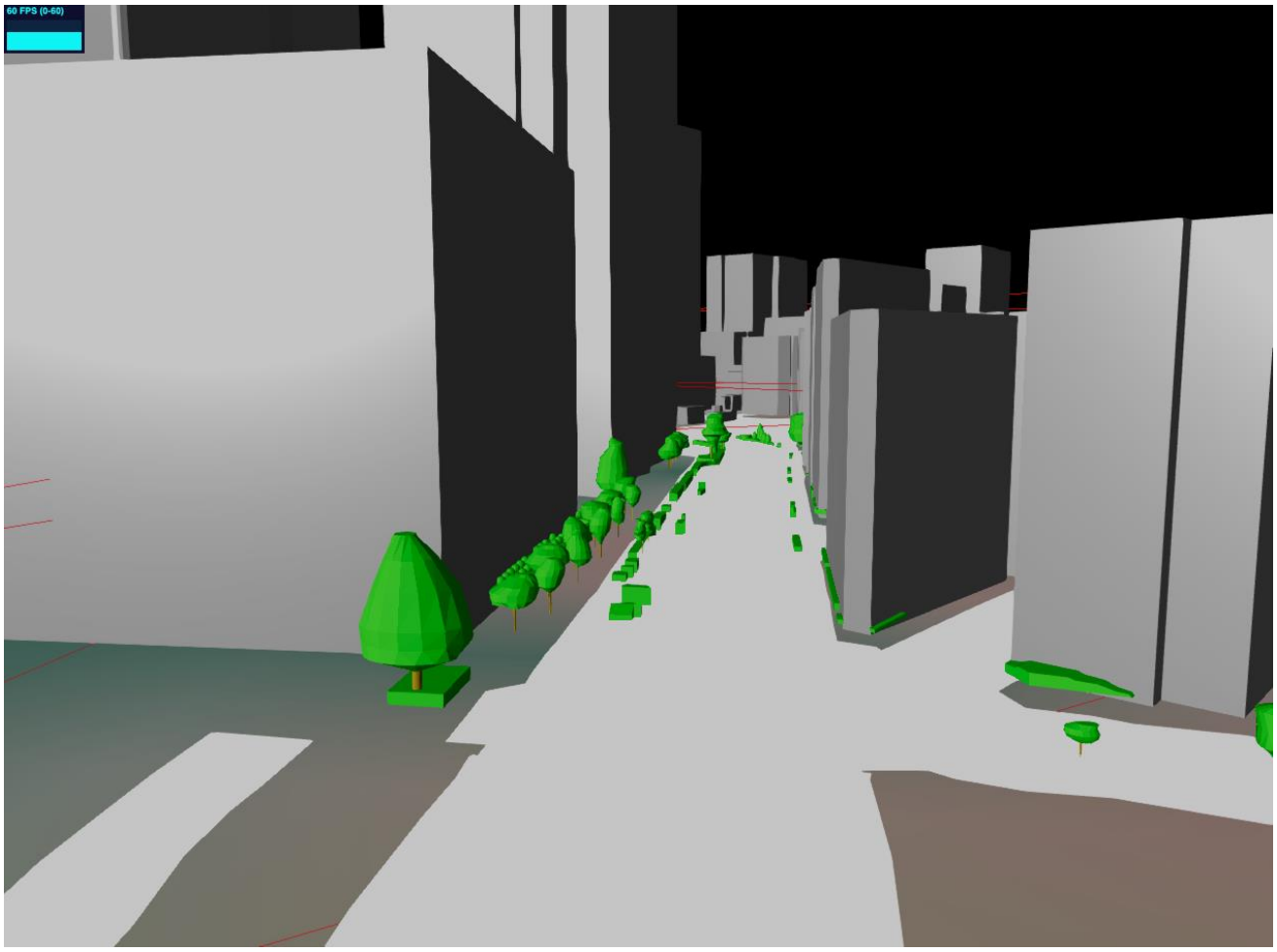


図 19　横浜市の建築物モデル・道路モデル・植生モデルが描画されている様子

本プロトタイプでは、3D Tiles として横浜市の建築物モデル（LOD1）及び植生モデル(LOD3)を、MVT として横浜市の道路モデル（LOD1）の描画を試みた。

原理的には WASM 側で 3D Tiles の各タイルのジオメトリや表示タイルの選択などの計算を行うことができるが、実際の 3D Tiles の描画ロジックはカメラとタイルの位置関係から適切なタイルを決定する必要があるなど複雑である。ここでは NASA AMMOS が開発する Three.js のライブラリ 3d-tiles-renderer を使用した。将来的に、3D Tiles のファイルのパースやタイルの選択などの処理を WASM 側に移植していくことで、Three.js 以外のレンダリングエンジンでも 3D Tiles を描画可能になることが期待される。

MVT の描画では、MVT ファイルをパースし、最終的なジオメトリを計算している。ただし MVT をはじめとする GIS のデータでは、ポリゴンは主に外周を表す線（点の配列）で記述されているが、GPU に転送する際には三角形の集合にしなければ（つまり、頂点バッファのみならず、3 つずつ頂点を結ぶインデックスバッファを計算しなければ）3D でポリゴンの描画はできない。そこで Mapbox が開発する earcut というライブラリを使用した。これはポリゴンの外周や穴を表す線のデータを与えると、高速に三角形（頂点バッファとインデックスバッファ）を計算するライブラリで、CesiumJS 内部でも使用されている。

これを用いることで、MVT のような GIS のデータを三角化して GPU で描画することが可能である。また、MVT の描画においては頂点数が非常に多くなるため、全てのタイルのジオメトリをマージして 1 つのジオメトリとした。

実装してみたところ、MVT の計算には時間がかかり、プロトタイプの Web ページ表示直後に数秒間画面が応答しなくなった。ただし一度ジオメトリの計算が終わってしまえば高速に描画が可能で、執筆者の環境でもカメラを移動しながらでもフレームレートも 60FPS を維持することができた。このことから今後は、こうしたジオメトリの計算を WebWorker で行うようにすることで、UI スレッドのブロックを防ぎ、UX の改善が可能であると期待される。またジオメトリをマージすることでフレームレートが向上し描画が高速化されたため、高速化手法として有効である。

3-3-5. 地形の表示

図 20　富士山と周辺の地形を描画した様子

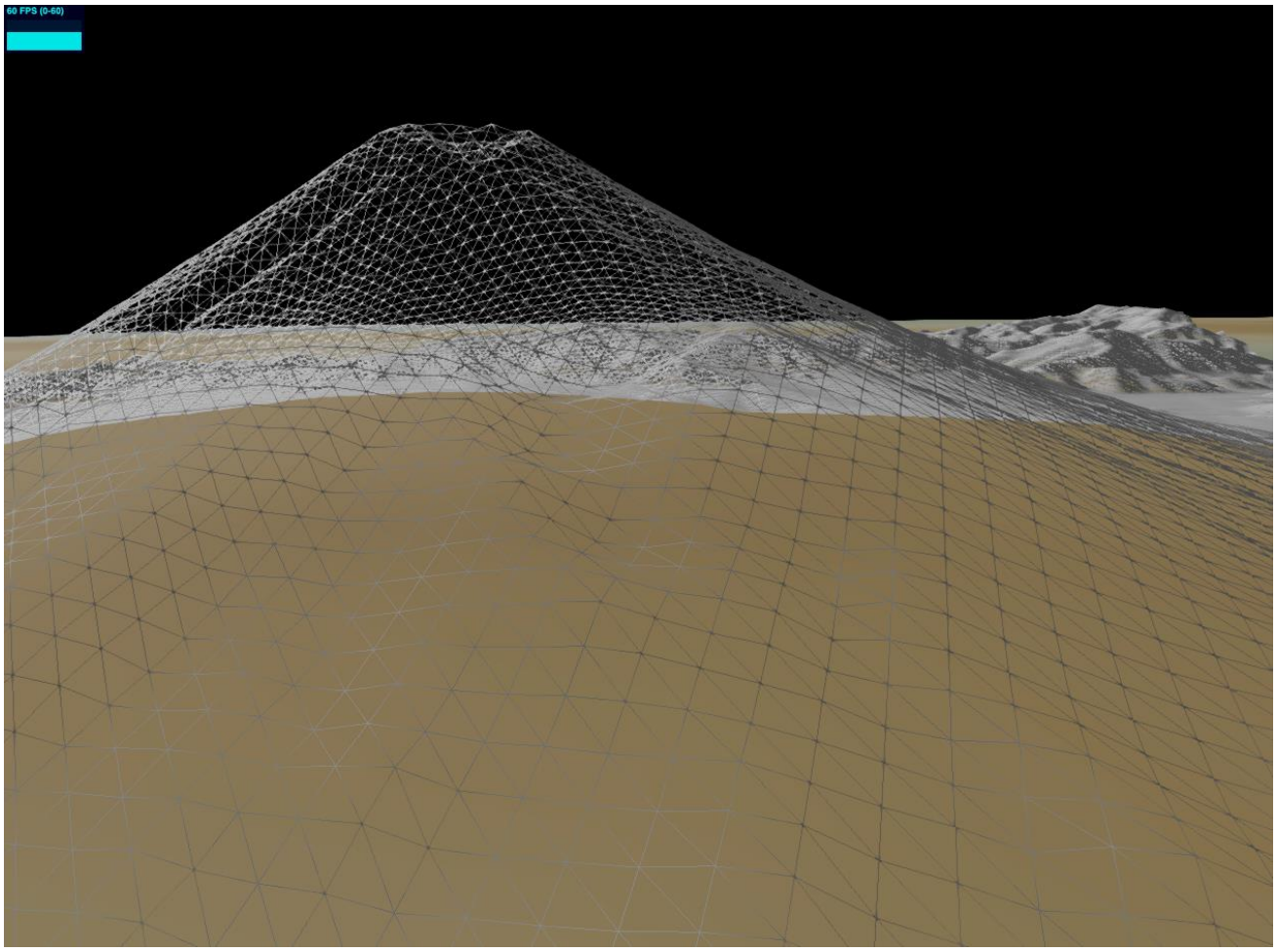


図 21　地形のワイヤーフレームを表示し、富士山に接近した様子

国土地理院が配信する標高タイルを使用して、地形の描画を行なった。ここでは地形の特徴が目視で確認しやすい富士山とその周辺の地域をズームレベル 10 で描画を行なった。地形のジオメトリの計算は WASM 側で行なっている。

タイルの描画は前述と全く同じであるが、各タイルのセグメント（分割数）を 256 に設定した。これは標高タイルの画像の幅と高さが 256px だからである。分割することで生まれた多数の頂点ごとに、対応する標高タイルのピクセルの色を取得し、RGB 値を元に実際の標高を計算した。経度・緯度・高度が分かれば、XYZ（地心直交座標系）に変換することができるため、最終的な頂点の座標が求まる。

「地理院地図｜標高タイルの詳細仕様」[8]によると、ピクセルの色の RGB それぞれの値（0~255）から、標高（メートル）を 0.01m の分解能で算出することができる。以下はその計算を実装した Rust のコードの抜粋である。

[8] https://maps.gsi.go.jp/development/demtile.html

```
let image_x = x * (terrain_w - 1) / segments;
let image_y = (terrain_h - 1) - y * (terrain_h - 1) / segments;

let i = image_y * terrain_w + image_x;
let r = terrain[i * 4] as i64;
let g = terrain[i * 4 + 1] as i64;
let b = terrain[i * 4 + 2] as i64;

let h = if r != 128 || g != 0 || b != 0 {
    if r >= 128 {
        r * 65536 + g * 256 + b - 16777216
    } else {
        r * 65536 + g * 256 + b
    }
} else {
    0
};

h as f32 * 0.01 // meters
```

実際に描画してみて、地形のジオメトリ計算は、タイルのジオメトリ計算よりもさらに負荷が高い処理のため、執筆者の環境では Web ページが 10 秒前後フリーズすることが確認された。これを軽量化するには、前述のように描画するタイルのズームレベルやカリングを最適化することに加え、WebWorker に処理を移動することや、Mapbox が開発する地形の形状を保ちながら頂点数を最適化する MARTINIi のようなライブラリを用いたり、より計算やメモリ使用量が少なく済むデータフォーマットである quantaized mesh terrain を使用したりするといった工夫が考えられる。

また、本プロトタイプではジオイド高の考慮は行わなかった。本来 XYZ（地心直交座標系）では、地球楕円体からの高度は標高ではなく楕円体高となる。そのため正確な高度を得るために、ジオイド高を実際のデータから計算することも検討した。しかし、国土地理院が配布するジオイド高のデータは再利用性が高い状態とはいえず、地図エンジンから利用しやすいように事前にデータの編集作業が必要である。またジオイド高は大きなデータのため、Web 上でリアルタイムに読み込むデータとしては現状適していない。そのため、地形データを事前に作成する際に、ジオイド高データを参照し、楕円体高の地形データを作成することが望ましい。

3-3-6. シェーダーの実装による新たな表現に基づくデータ可視化

本プロトタイプでは、レンダリングエンジンに Three.js を使用していることから、シェーダーを独自にカスタマイズして、見た目を大きく変更することが可能である。これは、今回開発する地図エンジンが持つ「レンダリングエンジンの分離」によって得られた利点である。

レンダリングエンジンの分離によって生まれるこの利点を確認するため、ここでは例として「SF のようなサイバー空間」をイメージした画面作りを試みた。その結果を以下に示す。これらの結果から、レンダリングエンジンの分離によって、従来の地図エンジンでは難しかった画面作りが可能になっていることが示唆される。

なお、この画面作りにおいては、Three.js を使用して以下の実装を行なった。

- エッジを光らせるため、別途、各メッシュのジオメトリデータをもとに、エッジを表すジオメトリを EdgesGeometry を使用して計算し、シーンに追加している。
- メッシュに適用するマテリアルは、LineBasicMaterial を拡張して実装を行なった。実際の画面では、白い光の波が 3D モデル上を地面側から上空側へと流れていくアニメーションを見ることができる。このアニメーションを正しく実装するためには、シェーダーが各頂点の楕円体高（=地球楕円体表面からの高さ）を取得できる必要があるが、地心座標系のためそのままでは取得することができない。各頂点の楕円体高を事前計算して頂点属性に格納する方法もあるが、その分 GPU への転送量が増え VRAM を消費してしまう。そこで頂点シェーダー内で各頂点の楕円体高を地心座標系から変換して求め、その結果を基にフラグメントシェーダーで白く光らせるかどうかを計算している。
- 各メッシュのエッジのみにブルームのポストプロセッシングを適用し、エッジが光っているように見せている。

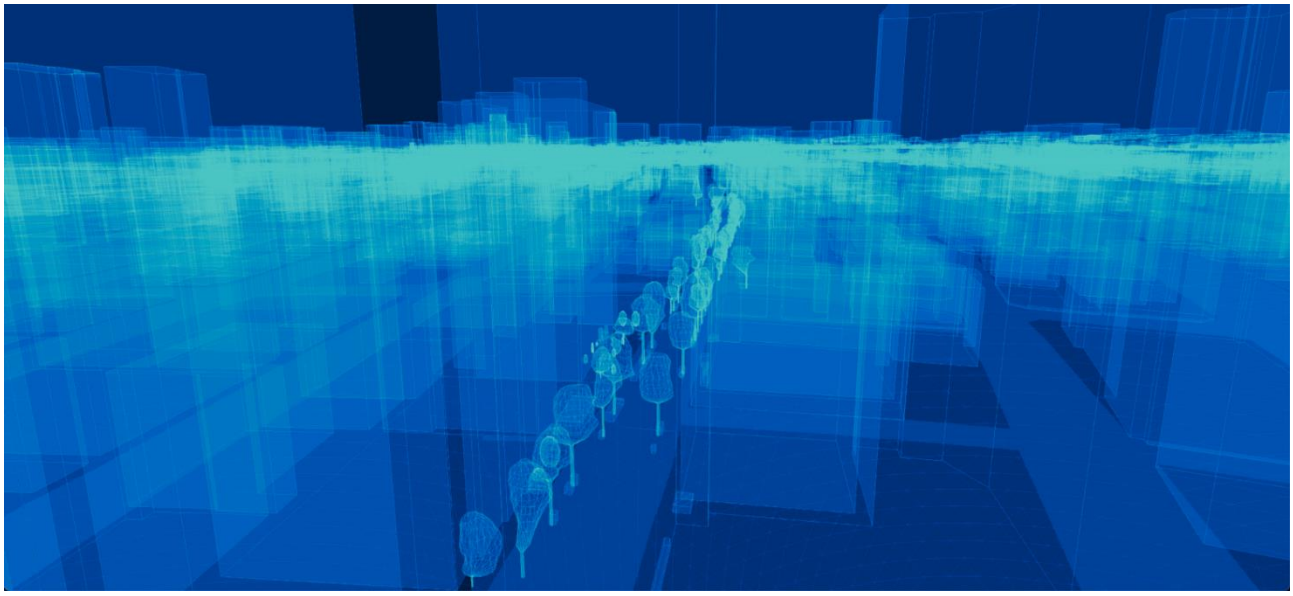


図 22　サイバー空間のような表現で街並みを描画した様子

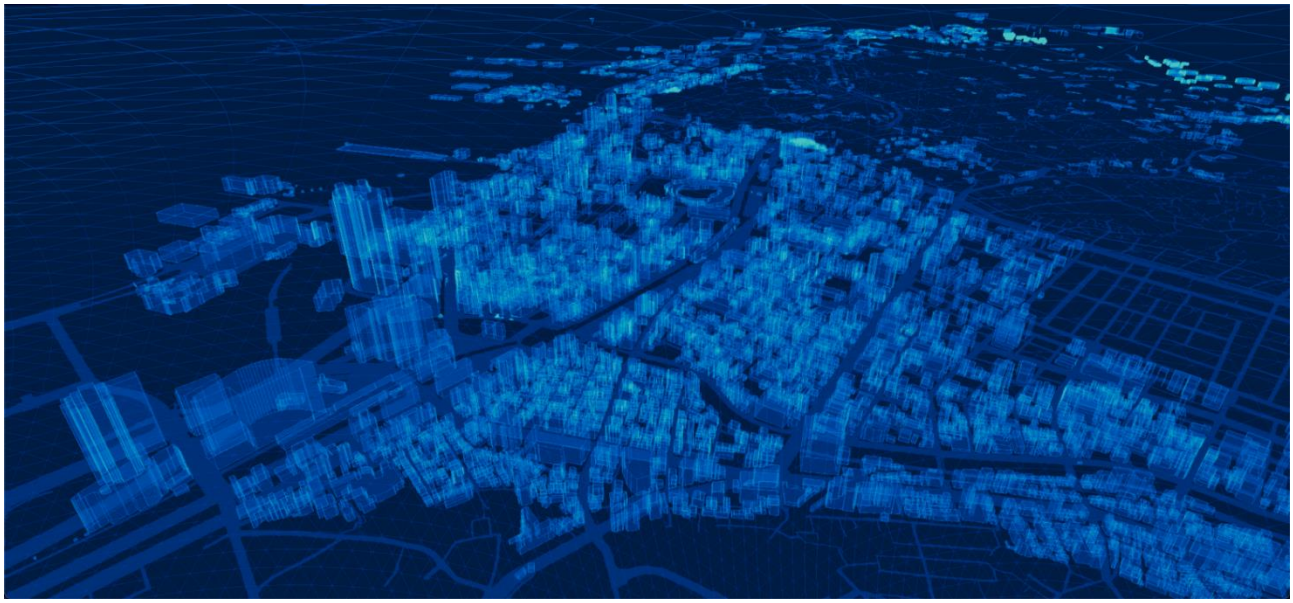


図 23　サイバー空間のような表現で描画された街並みを一望した様子

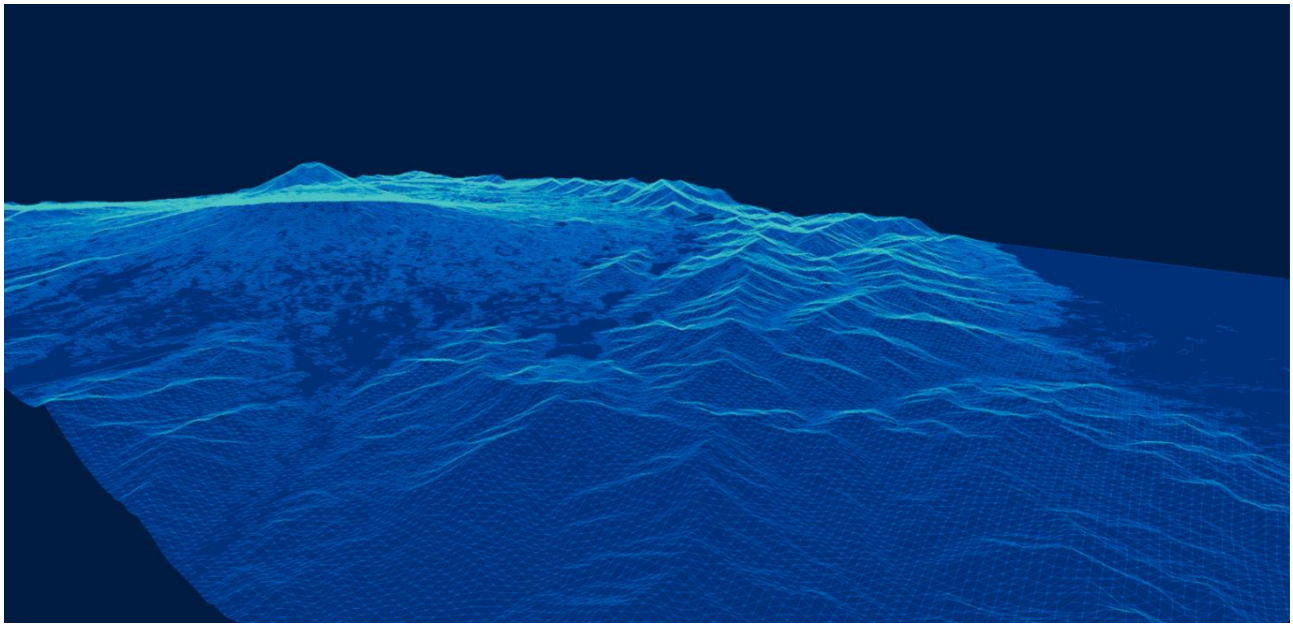


図 24　サイバー空間のような表現で富士山周辺を描画した様子

3-4. 今後の展望

ここまでで、現在の地図エンジンと最新技術に関する調査、及び図エンジンのプロトタイプの開発を行ってきた。ここでは、今後も地図エンジンの開発を継続し、プロダクションレベルまで到達するために、将来的に解決が必要と思われる課題を整理する。

1. レンダリングエンジンと WASM 間のデータの送受信の最適化が必要
   WASM は処理を高速化する可能性があるが、WASM 外部（例えば JavaScript）とのデータのやり取りの頻度が多くなってしまうと、かえって WASM を採用しない実装よりもパフォーマンスが低下してしまう可能性がある。よりパフォーマンスを高めるため、なるべく WASM とレンダリングエンジン間のデータのやり取りの回数や、データのコピーを最小限にすることが望ましい。
   また、地図エンジンを WASI アプリケーションとして実装する場合、標準入力を入力して標準出力及び標準エラー出力に結果を出力するような、コマンドラインアプリケーション（CLI）として振る舞う必要がある。どのような内容を入力し、出力するべきなのか、出力データの抽象度をどれくらいに置くべきなのかは、ベンチマークを行い、処理効率も踏まえて今後より深く検討していく必要がある。
2. Web におけるパフォーマンスの制約は現在も依然として大きい
   第 2 章の調査で、現在のところ SharedArrayBuffer（SAB）を使用することが困難であるという結論が得られた。この制約により、Web ではマルチスレッドを用いた並行処理の実装が大きく制限される。また、WASM Threads などの関連仕様は 2023 年 1 月現在、未だ策定途中である（策定や実装が進んだとしても SAB の制約は依然として残る）。
   こうしたことから、Web でマルチスレッドを実現するとしても現在では、ある程度大きな粒度の処理を WebWorker を用いて処理するにとどまる。その中でも少しでもパフォーマンスを高めるため、Web 向けの実装においては、WebWorker によるワーカープールや優先度付きキューを構築し、優先度に応じてタスクの処理順が変化するような仕組みを持つことが望ましい。
   実際、本プロトタイプでは全てシングルスレッドで動作し、マルチスレッドでの処理が行われていない。これにより、タイルの計算処理などによって、ページを読み込んでの最初の数秒間は Web ページがフリーズしてしまう。パフォーマンスを最適化する上では WebWorker の使用は必須と言える。
3. Draco のデコードが地図エンジン内部の WASM 上で処理できない
   3D Tiles や glTF の圧縮技術である Draco は C++で実装されたライブラリで、WASM 版も公開されているが、JavaScript から呼び出すことが前提のライブラリである。また、WASM には外部の WASM をライブラリとして利用できる仕組みがまだ確立されていない（現在、WASI Preview 2 でコンポーネントモデルが検討されている）。WASM では外部の C/C++のライブラリを FFI を通して呼び出すといった方法は困難であり、Rust で WASM アプリケーションを開発しているならば、そうしたライブラリもまた Rust で実装されなければいけない。より高度な高速化のためには、Draco デコード処理を行うピュアな Rust 実装、又は WASI のコンポーネントモデルに準拠した

Draco の WASI ライブラリが必要である。公開されているライブラリが世の中になければ、独自実装が必要となる。
この問題は、Draco 以外にも、C/C++の実装しか存在しないようなライブラリを使用する際に発生する。

以上の課題を解決しつつ、対応データフォーマットの増加、マルチスレッドを活用してデータ読込効率化、シェーダーの最適化や新規実装を含むビジュアルの強化などの工夫により、より軽量でリッチなビジュアルを持つマルチプラットフォームな地図エンジンの開発が可能と見込まれる。

4. まとめ

本調査は、現在のさまざまな地図エンジンのアーキテクチャや、新しい地図エンジンで利用可能な最新技術を調査し、それらの技術を用いることで、以下に述べる課題を解決するような、新しい地図エンジンの開発が可能かどうかを、プロトタイプを開発しながら検証した。

最初に現状の地図エンジンのアーキテクチャや、地図エンジンで利用可能な最新技術に関する調査を行なった。その結果、新しい地図エンジンにおいては、メインループおよび GIS エンジンに WASM を採用し、WASM を採用することからプログラミング言語は Rust が最適であり、レンダリングエンジンはそれぞれのプラットフォーム向けに実装するような設計にすることが良いと結論づけた。また、ECS のような最新のゲーム開発における技術を採用することで、従来より動作が高速で開発効率が高い地図エンジンが開発できる可能性を見出した。しかし、SharedArrayBuffer が現実的に使用できないなど、Web においては依然としてパフォーマンスの制約が存在することも明らかになった。

次に地図エンジンのプロトタイプの開発を行った。プロトタイプの開発においては、ECS のライブラリとして Bevy を採用し、レンダリングエンジンには Three.js を採用し、メインループ等は WASM で動作しレンダリングは Three.js を通して WebGL で行う、Web のデモアプリケーションの開発を行った。特にメインループとレンダリングエンジンのやり取りの設計を慎重に行うことで、地球の描画、カメラの移動、地図タイルの表示、地形の表示、3D Tiles による建築物モデルの表示を行うデモアプリケーションを開発することができた。これにより、地図エンジンの CPU バウンドな処理とレンダリングエンジンを分離した状態で動作する地図エンジンが開発可能であることが確かめられた。一方で、十分にハードウェアの性能を引き出す地図エンジンを開発するには、Web におけるマルチスレッディングの制約をはじめとする課題が依然として存在している。

今後の開発の展望としては、地図エンジン開発において将来的に解決と思われる課題について、それぞれ以下のように解決を目指す。

1. レンダリングエンジンと WASM 間のデータの送受信の最適化が必要
   現状、地図エンジンの WASM とレンダリングエンジン間の通信（データ授受）は、フレーム更新後に結果をレンダリングエンジンに反映させるタイミングだけでなく、マウス操作などのユーザー入力が発生するたびに行われている。しかし実際にはこれらのデータを使用するのは、フレーム更新が実行された時のみである。そこでフレーム更新が発生するまでは JavaScript 側で発生したイベントのデータを溜め込んでおき、フレーム更新時に一括で WASM へ転送するといったアプローチを試みることができる。これにより WASM とレンダリングエンジン間のデータ授受の頻度を減らすことができ、パフォーマンスの向上につながる可能性がある。
2. Web におけるマルチスレッディングの制約
   WebWorker を積極的に使用することを検討する。具体的には、レンダリングエンジンを

WebWorker 化する（OffscreenCanvas を用いてレンダリングをワーカーで行う）、WASM の処理全体を WebWorker 化する、WASM の計算量が多い一部処理のみ（ジオメトリの計算など）を WebWorker 化する、といったパターンが考えられる。更に、WebWorker を複数同時に起動させ、優先度つきワーカープールのような仕組みを構築することで、例えばカメラに近いタイルの処理を優先するといった処理を可能にしつつ、小さいタスクを効率よく実行する仕組みが実現する可能性がある。これにより、地図エンジンの UI の応答停止を最小化することができ、ユーザー体験が向上する可能性がある。

3. Draco のデコード等が WASM 上で処理できない
   直近では、3D Tiles のデータ読み込み時に JavaScript 側で Draco のデコードまでを行い、その結果を WASM 側に転送することを検討する。長期的には、Rust のみで実装された Draco のライブラリを開発することも考えられる。

本稿による調査が、我が国における Web 技術、GIS 技術、3DCG 技術、並びに、3D 都市モデルに最適化された WebGIS 地図エンジン開発の発展に寄与すれば幸いである。

3D 都市モデルに最適化された WebGIS 開発に関する調査レポート

2024 年 3 月　発行
委託者：国土交通省　都市局
受託者：株式会社ユーカリヤ