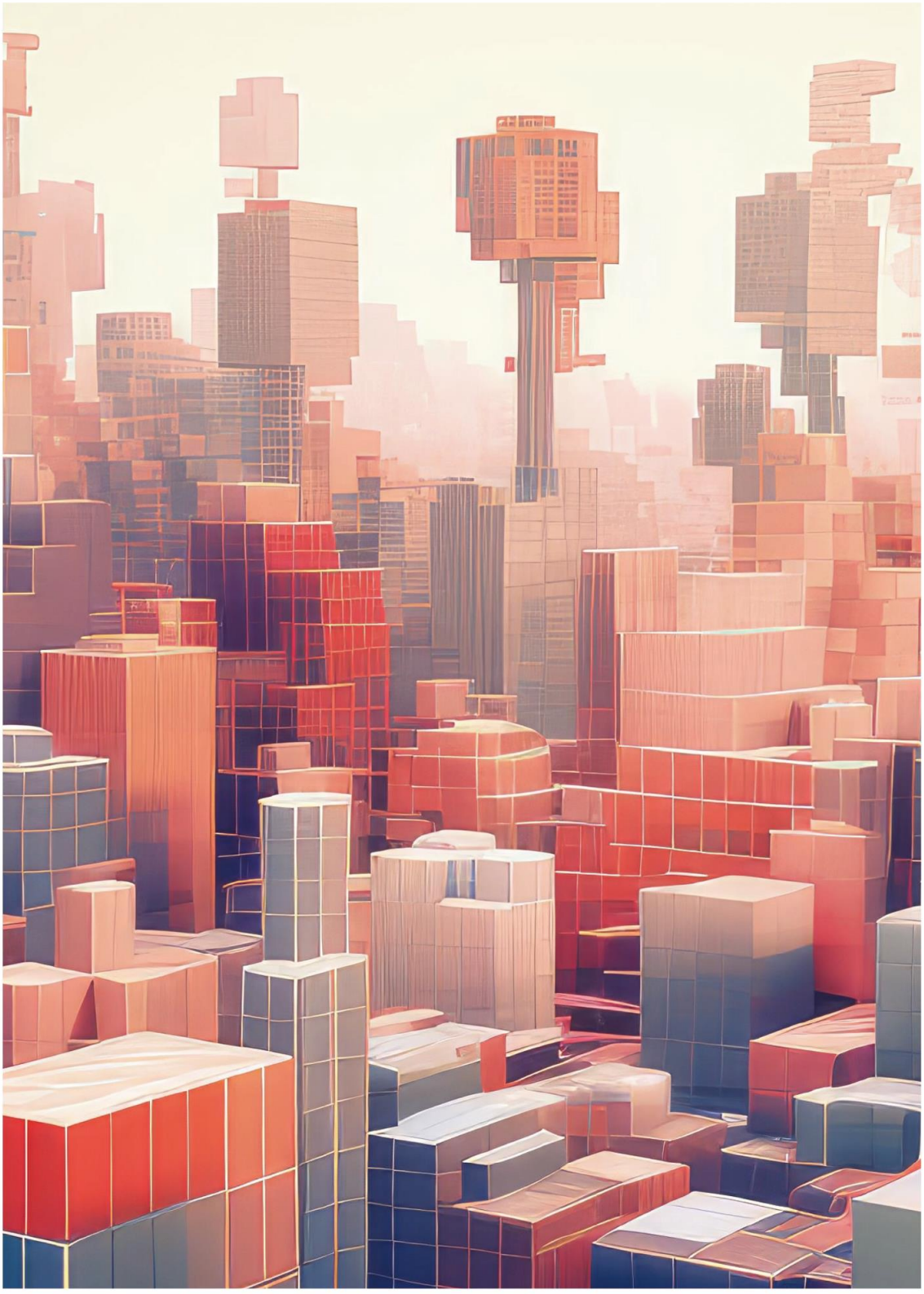
PLATEAU
by MLIT

Handbook of 3D City Models

3D都市モデル導入のためのガイドブック



PLATEAU SDK 技術解説書

series No. 12

Technical Reference for PLATEAU SDK

はじめに

- 令和2年度からスタートした Project PLATEAUでは、国際標準規格CityGML2.0 に基づき、我が国初の都市デジタルツインの標準データモデルを「3D 都市モデル」の標準製品仕様という形で定め、全国でデータ整備とその活用を拡大してきた。
- 3D都市モデルを活用した様々な領域におけるアプリケーション開発をさらに促進していくためには、データ利用の利便性をより高めるための開発者向けツールキット （SDK）の充実が重要である。このため、本業務では、令和4年度に開発した PLATEAU SDK for Unity/Unreal を改修・拡充し、ゲームエンジン領域における3D 都市モデルを利用した開発環境のさらなる改善を目指している。
- 令和5年度には、PLATEAU SDK for Unity/Unrealの機能拡充を行った。属性情報へのアクセシビリティ改善、テクスチャの結合機能、3D都市モデルのマテリアル改善機能、オブジェクトの結合・分割機能を追加しユーザビリティを改善した。
- また、Unity上での利用環境の向上のため、「PLATEAU SDK Toolkits for Unity」として、GISの利用最適化（Maps Toolkit）、モバイル・PCでのアプリケーション開発支援ツール群の構築（Sandbox Toolkit、 Rendering Toolkit）、拡張現実（AR）アプリケーション構築の支援パッケージ（AR Extensions）、アプリケーション構築のサンプルファイルの作成（Sample）を構築した。
- 本技術解説書は、PLATEAU SDK及びPLATEAU SDK Toolkits for Unityの機能や仕様、利用方法を解説することで、PLTEAUを利用する地方公共団体や民間企業、技術者コミュニティ等に所属する様々な方が3D都市モデルをゲームエンジン上で活用するための知見を提供することを目的としたものである。
- 3D都市モデルのユーザーに本技術解説書を参照していただくことで、ゲームエンジンを活用した3D都市モデルのユースケース開発が加速していくことを期待する。

アップデートノート

2024.3.31　「PLATEAU SDK　技術解説書」初版公開

■目次

第１編　PLATEAU SDK 2.0

第1章　ソフトウェアの構成 ・・・・・・・・・・・・・・・・・・・・・・・・ **5**
1.1　PLATEAU SDK for Unity/Unrealの概要 ・・・・・・・・・・・・・・・・・・・・ 6
1.2　アーキテクチャ ・・・・・・・・・・・・・・・・・・・・・・・・・・・・ 8
1.3　用語集 ・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・ 12

第2章　ソフトウェアの機能詳細 ・・・・・・・・・・・・・・・・・・・・・・・・ **13**
2.1　PLATEAU SDK for Unity/Unrealの機能一覧 ・・・・・・・・・・・・・・・・・・・・ 14
2.2　3D都市モデルインポート機能 ・・・・・・・・・・・・・・・・・・・・・・・・ 15
2.3　3D都市モデルエクスポート機能 ・・・・・・・・・・・・・・・・・・・・・・・・ 40
2.4　属性情報取得機能 ・・・・・・・・・・・・・・・・・・・・・・・・・・・・ 42
2.5　ゲームオブジェクトON/OFF機能 ・・・・・・・・・・・・・・・・・・・・・・・・ 44
2.6　分割結合機能 ・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・ 46
2.7　マテリアル分け機能 ・・・・・・・・・・・・・・・・・・・・・・・・・・・・ 49

第２編　PLATEAU SDK Toolkits for Unity

第1章　Toolkitsの構成 ・・・・・・・・・・・・・・・・・・・・・・・・・・ **53**
1.1　PLATEAU SDK Toolkits for Unityの概要 ・・・・・・・・・・・・・・・・・・・・・ 54
1.2　アーキテクチャ ・・・・・・・・・・・・・・・・・・・・・・・・・・・・・ 56
1.3　用語集 ・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・ 63

第2章　Toolkitsの機能詳細 ・・・・・・・・・・・・・・・・・・・・・・・・・・ **64**
2.1　PLATEAU SDK Toolkits for Unityの機能一覧 ・・・・・・・・・・・・・・・・・・・・ 65
2.2　各機能の処理詳細 ・・・・・・・・・・・・・・・・・・・・・・・・・・・・ 67
2.3　ユーザーインターフェース ・・・・・・・・・・・・・・・・・・・・・・・・・・ 82
2.4　アルゴリズム ・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・ 100

第１編　PLATEAU SDK 2.0

第１章　ソフトウェアの構成

1.1 PLATEAU SDK for Unity/Unrealの概要

　PLATEAU SDK for Unity及びPLATEAU SDK for Unrealについて解説する。SDKとは、Software Development Kit （ソフトウェア開発キット）の略であり、両SDKはUnity及びUnreal Engine上でPLATEAUの3D都市モデルデータを利用するためのツールである。Unity及びUnreal Engine とはゲームエンジンの一種であり、ゲーム制作、映像制作、シミュレーションなどの目的で広く利用されているツールである。

　このSDKを使用することで、実世界を舞台にしたアプリケーションの開発やPLATEAUの豊富なデータを活用した都市シミュレーションを開発できる。

- GitHub（Unity）：https://github.com/Synesthesias/PLATEAU-SDK-for-Unity
- Unity対応バージョン：Unity 2021.3.35f1
- GitHub（Unreal Engine）：https://github.com/Synesthesias/PLATEAU-SDK-for-Unreal
- UnrealEngine対応バージョン：5.3.2

上記、対応バージョンは本書作成時点のものである。

図1-1-1　PLATEAU SDK for Unity/Unrealで読み込んだ3D都市モデル

1.2 PLATEAU SDK for Unity/Unrealのシステム全体構成

PLATEAU SDK 2.0を構成するソフトウェアの概要について解説する。PLATEAU SDK 2.0のシステム全体構成図は以下のとおりである。

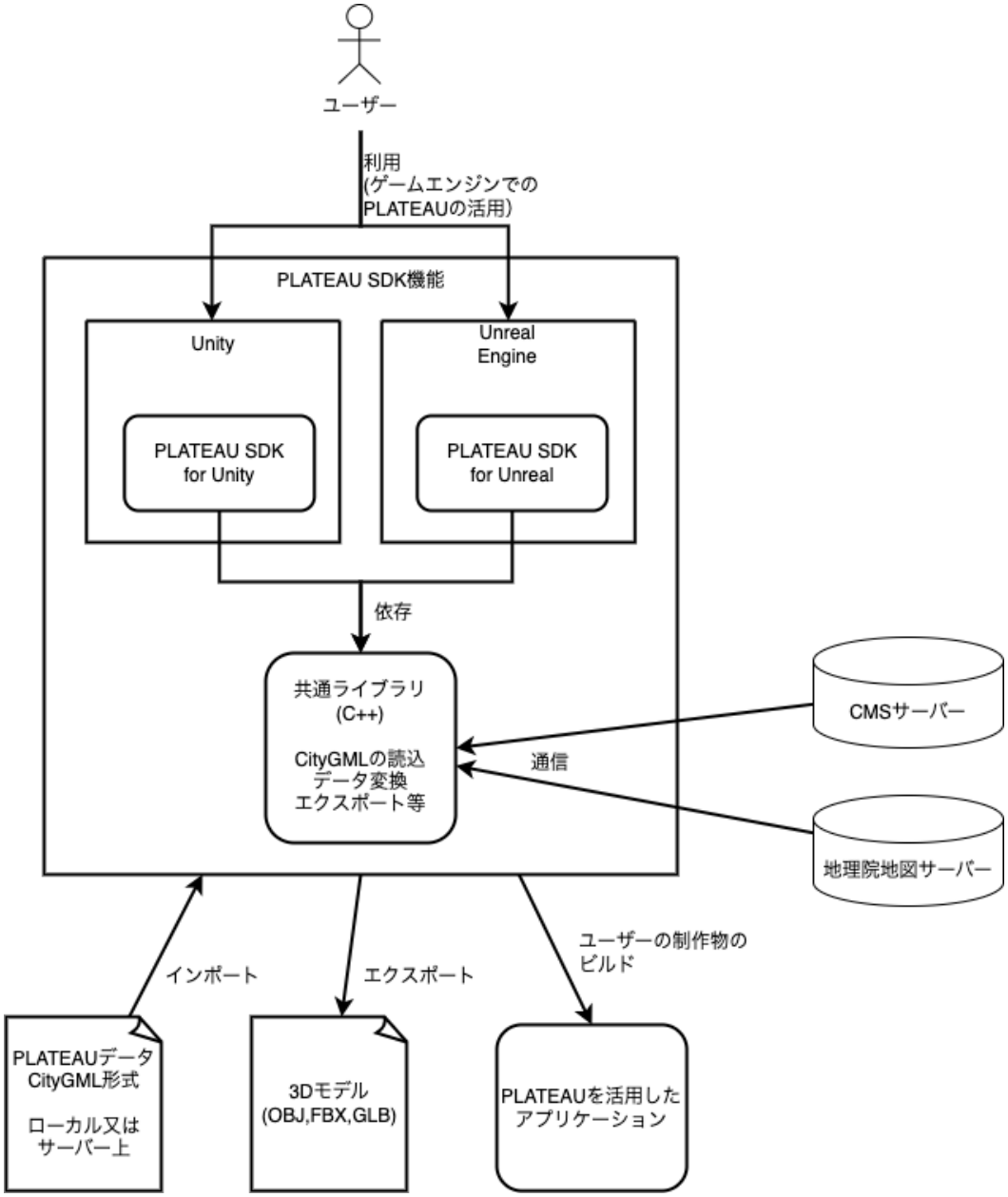


図1-1-2　PLATEAU SDK 2.0 システム全体構成

1.3 PLATEAU SDK for Unity/Unrealのアーキテクチャ

PLATEAU SDK for Unity/Unrealは以下のコンポーネントによって構成されている。

- ゲームエンジン（Unity/Unreal）のエディタ拡張
  - 本ソフトウェアはゲームエンジンの拡張機能として動作するため、ゲームエンジン標準のユーザーインターフェースを拡張する機能を提供している
- ゲームエンジン（Unity/Unreal）のランタイム実装
  - 本ソフトウェアをゲームエンジンから利用できるようにするための機能・APIを提供している
  - 内部的には主にゲームエンジンとのインテグレーション処理が実装されており、コアなロジックは共通ライブラリで実装されている
- 共通ライブラリ（libplateau）
  - Unity、Unreal向けSDKが共通で利用する、本ソフトウェアに固有のロジックを提供するライブラリ
  - 内部実装はC++で記述されているが、Unity向けにC#のラッパー実装も提供している

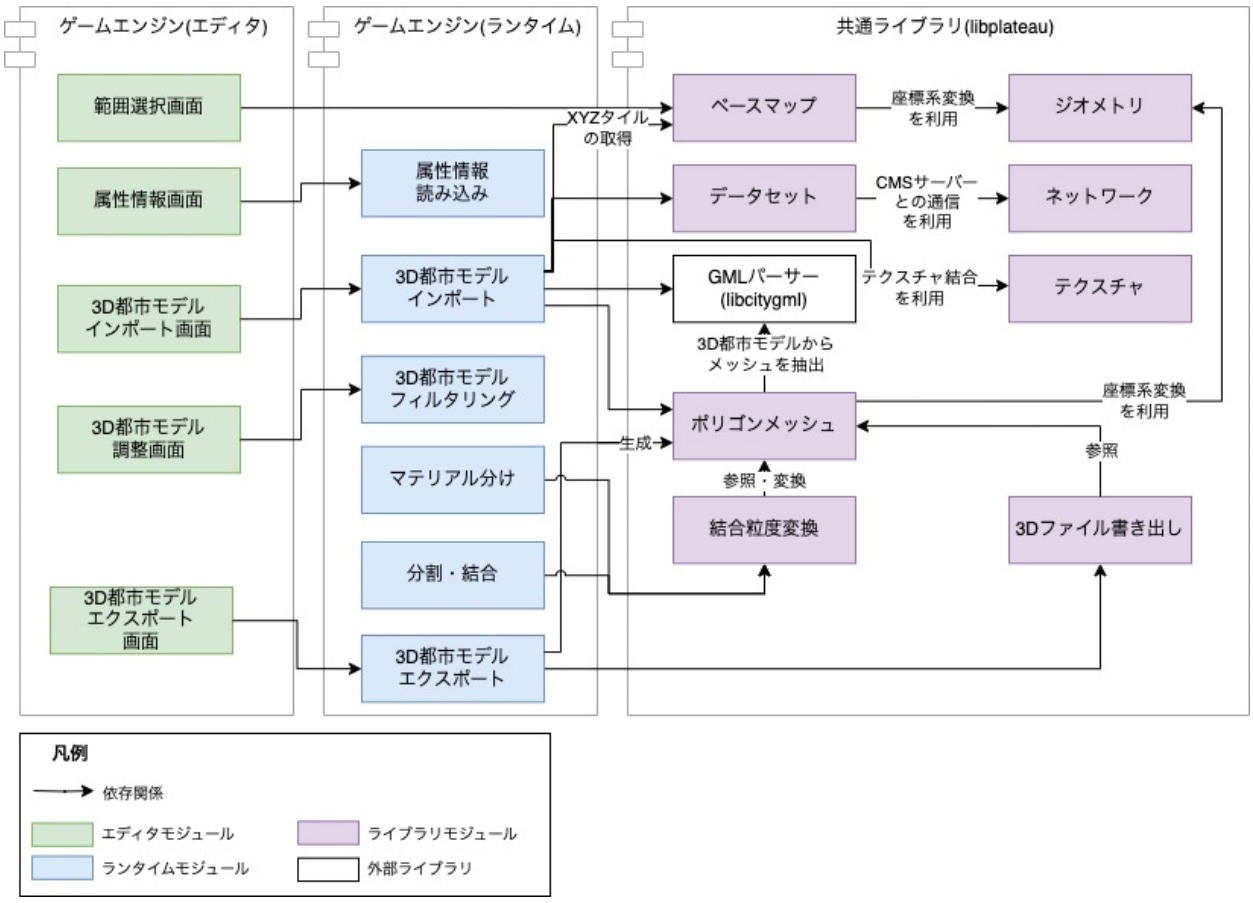


図1-1-3　PLATEAU SDK for Unity/Unrealのアーキテクチャ

以下に、ユーザーインターフェースを除いたソフトウェアのコアなモジュールの一覧を示す。

ゲームエンジン（Unity/Unreal）のランタイム実装

ランタイム実装は以下のモジュールによって構成されている。

表1-1-1　ランタイム実装のモジュール

モジュール名(名前空間名)	説明
3D都市モデルインポート	PLATEAUの3D都市モデル標準製品仕様書（第3版）に準拠している全てのCityGMLを入力として、ゲームエンジンに形状・マテリアル・属性情報をインポートできる。
属性情報読込	CityGMLの属性情報をゲームエンジン内で保持し、スクリプトからアクセスするためのAPIを提供する。
ゲームオブジェクトON/OFF	地物型とLODによる条件指定に基づいて、一括でゲームオブジェクトの表示/非表示を切り替える機能を提供する。
マテリアル分け	インポートされた3D都市モデル内の各オブジェクトのマテリアルを地物型に応じて分割する機能を提供する。
分割・結合	インポートされた3D都市モデル内の各オブジェクトの結合粒度を変更する機能を提供する。
3D都市モデルエクスポート	インポートされた3D都市モデルを3Dファイル形式でエクスポートする機能を提供する。

共通ライブラリ

共通ライブラリは以下のモジュールによって構成されている。

表1-1-2　共通ライブラリのモジュール

モジュール名（名前空間名）	説明
ポリゴンメッシュ（polygon_mesh）	パースされたCityGMLに含まれるジオメトリ情報のポリゴンメッシュ化・生成されたポリゴンメッシュへのアクセス・ポリゴンメッシュ化する際の結合単位や抽出範囲などの設定を行うAPIを提供する。
ジオメトリ（geometry）	空間座標の変換に関する機能を提供する。
ベースマップ（basemap）	XYZタイル形式の地図の読み込みを行うAPIを提供する。
ネットワーク（network）	REST APIを利用してPLATEAU CMSサーバーにアクセスするAPIを提供する。
データセット（dataset）	PLATEAUの3D都市モデル標準製品仕様書（第3版）に準拠するデータセットへのアクセス（PLATEAU CMSサーバーに存在するデータセットの一覧の取得、提供されるLOD・地物の種類の取得、CityGMLのダウンロード等）を行うAPIを提供する。
3Dファイル書き出し（mesh_writer）	ポリゴンメッシュを入力として、FBX、glTF、OBJ形式へのエクスポート処理・エクスポートする際の座標変換やテクスチャの有無等の設定を行うAPIを提供する。
結合粒度変換（granularity_convert）	ポリゴンメッシュの結合単位を変換する機能を提供する。
テクスチャ（texture）	jpg, png, tif形式の画像の読み書き、テクスチャ結合時の画像処理を行う。また、地形に航空写真等を貼る際の地図画像の結合時の画像処理を行う。

共通ライブラリが利用するサードパーティ製ライブラリ

共通ライブラリでは以下のサードパーティ製ライブラリを利用している。

表1-1-3　共通ライブラリが利用するサードパーティ製ライブラリ

ライブラリ名	説明
libcitygml	CityGMLのパース、可視化のためのオープンソースのライブラリ。CityGML 2.0に準拠している。共通ライブラリではPLATEAUでのCityGMLの拡張仕様に対応するためにソースコードを改変して利用している。 https://github.com/jklimke/libcitygml
cpp-httplib	HTTP通信のためのオープンソースのライブラリ。XYZタイルへのアクセスとPLATEAU CMSサーバーへのアクセスに利用している。 https://github.com/yhirose/cpp-httplib
JSON	JSONのシリアライズ、デシリアライズのためのオープンソースのライブラリ。PLATEAU CMSサーバーから得られたHTTPレスポンスのパースに利用している。 https://github.com/nlohmann/json
glTF SDK	Microsoft社が提供するglTFのシリアライズ、デシリアライズのためのオープンソースのライブラリ。glTF形式でのエクスポートに利用している。 https://github.com/microsoft/glTF-SDK
FBX SDK	Autodesk社が提供するFBXのシリアライズ、デシリアライズのためのライブラリ。FBX形式でのエクスポートに利用している。再配布が禁止されているため、共通ライブラリ本体に静的リンクする形で組み込まれている。 https://www.autodesk.com/developer-network/platform-technologies/fbx-sdk-2020-3-1
xerces-c	Libcitygmlの依存ライブラリ https://github.com/Synesthesias/xerces-c.git
libxml2	Libcitygmlの依存ライブラリ https://github.com/GNOME/libxml2
openssl-cmake	cpp-httplibの依存ライブラリ https://github.com/janbar/openssl-cmake
zlib	FBX SDKの依存ライブラリ https://github.com/madler/zlib
libjpeg-turbo	jpeg画像の読み書きのためのオープンソースのライブラリ。テクスチャ結合機能で利用している。 https://libjpeg-turbo.org
libpng	png画像の読み書きのためのオープンソースのライブラリ。テクスチャ結合機能で利用している。 http://www.libpng.org/pub/png/libpng.html
libtiff	tiff画像の読み書きのためのオープンソースのライブラリ。テクスチャ結合機能で利用している。 http://www.libtiff.org

3.3 用語集

PLATEAU SDKにおいて独自に使用される用語をまとめる。

表1-1-4　SDK 用語集

用語	説明
最小地物	壁面（WallSurface）、屋根面（RoofSurface）等、1つの地物の構成部品として使用される地物が相当する。地物形状を結合・分割する際の粒度として用いられる。
主要地物	最小地物以外の地物が相当する。地物形状を結合・分割する際の粒度として用いられる。
ゲームオブジェクト	UnityでのゲームオブジェクトとUnreal Engineでのコンポーネントを総じてゲームオブジェクトと呼ぶ。
シーン	Unity/Unreal Engineにおけるコンテンツの配置空間（シーン、レベル）を総じてシーンと呼ぶ。
PLATEAU CMSサーバー	PLATEAUの3D都市モデルを管理・配信するサーバー
LOD（Level of Details）	本ドキュメントでは「LOD」はゲームエンジンでの描画距離による詳細度ではなく、CityGMLにおける3D都市モデルの詳細度を指す。

第１編　PLATEAU SDK 2.0

第２章　ソフトウェアの機能詳細

2.1 PLATEAU SDK for Unity/Unrealの機能一覧

PLATEAU SDK for Unity/Unrealが提供する機能について解説する。PLATEAU SDK for Unity/Unrealは以下の機能を提供している。

- 3D都市モデルインポート機能
- 3D都市モデルエクスポート機能
- 属性情報取得機能
- ゲームオブジェクトON/OFF機能
- 分割結合機能
- マテリアル分け機能

図1-2-1　PLATEAU SDK for Unity/Unrealの機能一覧

2.2 3D都市モデルインポート機能

3D都市モデルインポート機能では、PLATEAUの3D都市モデル標準製品仕様書（第3版）に準拠している全ての3D都市モデルデータを入力としてゲームエンジンにインポートできる。

図1-2-2 インポート画面

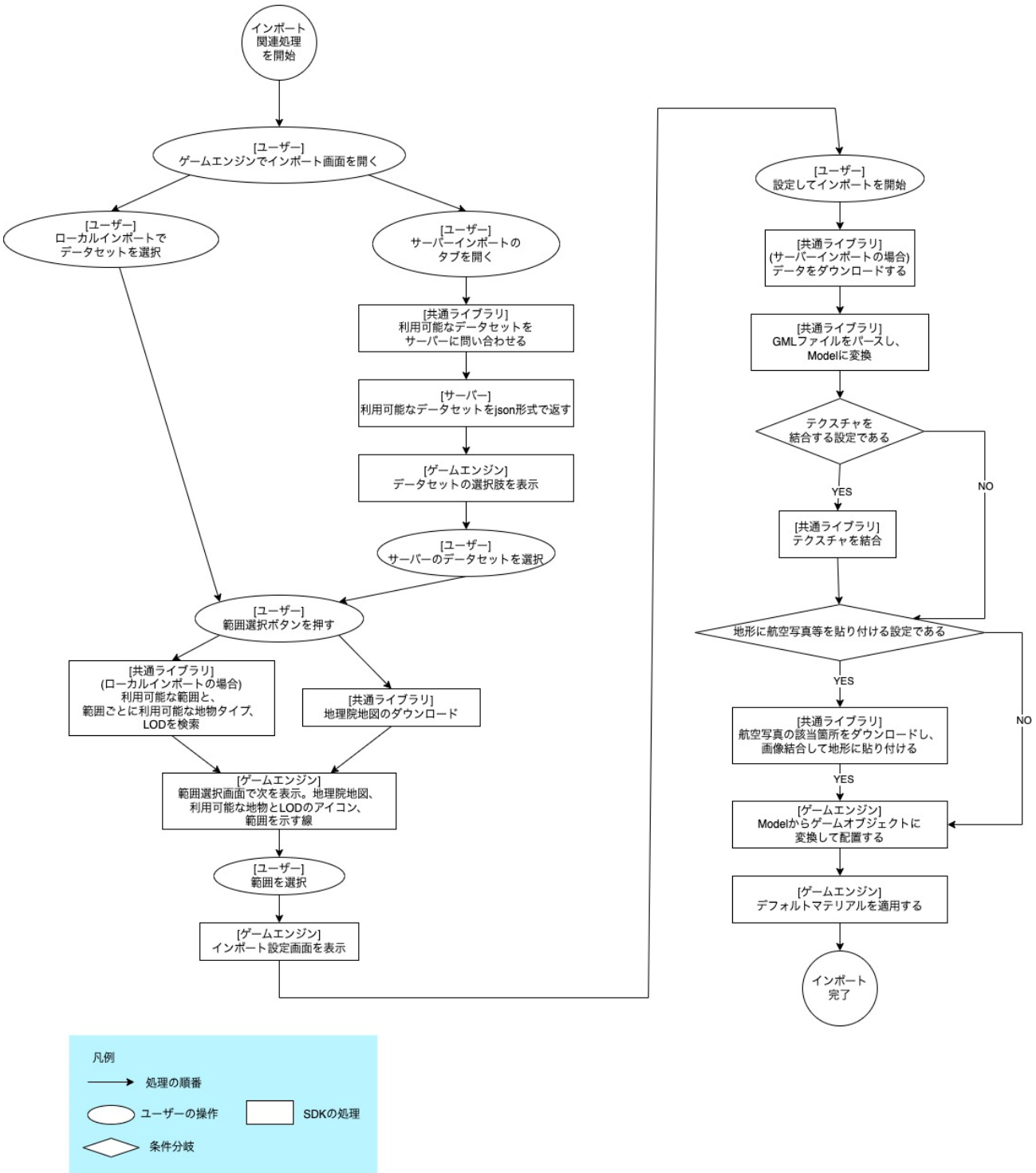


図1-2-3　インポート処理の概要

　前ページ図の各処理の仕組みについて、以下に説明する。
なお、[共通ライブラリ] という表記は、SDKのうちC++で実装されたUnityとUnreal両方で用いる共通機能部分を指し、[ゲームエンジン] という表記は、SDKのうちUnity及びUnreal Engineで実装された機能部分を指す。

2.2.1：［共通ライブラリ］利用可能なデータセットをサーバーに問い合わせる
　ユーザーがサーバーインポートを選択したとき、HTTPS通信によって、PLATEAUデータを配信するPLATEAU CMSサーバーと通信する。通信にはオープンソースライブラリであるcpp-httplibを用いる。

2.2.2：[サーバー] 利用可能なデータセットをJSON形式で返す
　PLATEAU CMSサーバーは、自身がどのようなPLATEAUデータを保持するかをJSONで返す。
JSONには次の情報が含まれる。

- 大分類（東京都、北海道など）
- 各大分類に含まれるデータセットのタイトル（東京都23区、札幌市など）、PLATEAU CMSサーバーでデータ作成者が記載した解説や注意書きなどの説明、内部的に使用している値であるデータセットID
- JSONの仕様のバージョン（spec）


```json
{
  "data": [
    {
      "title": "東京都",
      "data": [
        {
          "id": "tokyo-23ku",
          "title": "23区",
          "spec": "3.0",
          "description": "xxxx",
          "featureTypes": ["bldg", "dem"]
        },
        {
          "id": "hachioji-shi",
          "title": "八王子市",
          "spec": "3.0",
          "description": "xxxx",
          "featureTypes": ["bldg", "dem"]
        }
      ]
    },
    {
      "title": "神奈川県",
      "data": [
        {
          "id": "yokohama-shi",
          "title": "横浜市",
          "spec": "3.0",
          "description": "xxxx",
          "featureTypes": ["bldg"]
        }
      ]
    }
  ]
}
```


図　1-2-4　データセットの応答JSONの例

さらにサーバーはデータセットIDをもとに問い合わせを受けると、そのデータに関して次の情報をJSONで返す。

・利用可能な地物タイプ
・各地物タイプに含まれるCityGMLファイル、そのURL、及びその最大LOD

```json
{
  "bldg": [
    {
      "code": "12345678",
      "url": "https://xxxx/xxxxx/12345678_hogehoge.gml",
      "maxLod": 2
    },
    {
      "code": "12345679",
      "url": "https://xxxx/xxxxx/12345679_hogehoge.gml",
      "maxLod": 2
    }
  ],
  "veg": [
    {
      "code": "123456",
      "url": "https://xxxx/xxxxx/123456_hogehoge.gml",
      "maxLod": 2
    }
  ],
  "...": []
}
```

図　1-2-5　データセット内容の応答JSONの例

共通ライブラリは、受け取ったJSONをパースしてゲームエンジンに渡す。
JSONのパースにはオープンソースライブラリである nlohmann/jsonを用いる。

2.2.3 : [ゲームエンジン] データセットの選択肢を表示

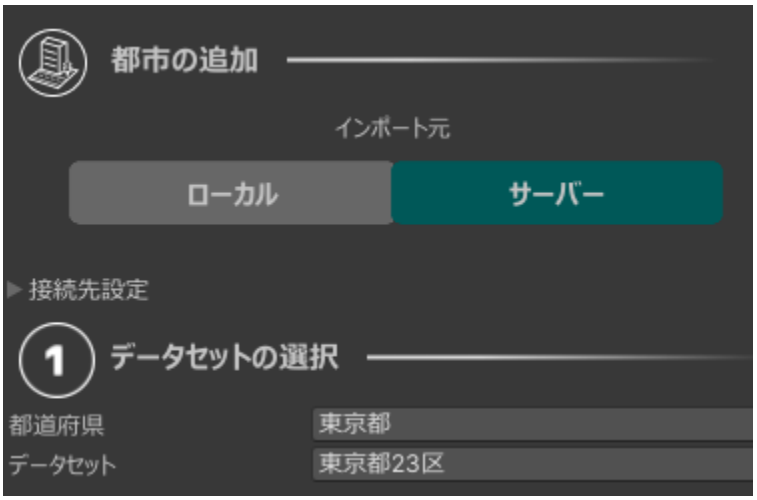


図1-2-6　データセット選択画面

2.2.4 : [共通ライブラリ] 利用可能な範囲と地物タイプ、LODを検索

　ローカルインポートの場合、ユーザーが範囲選択画面を開くと、共通ライブラリは3次メッシュごとにどの地物タイプがどのLODまで存在するのかを検索する。ゲームエンジンはその結果をアイコンで表示する。

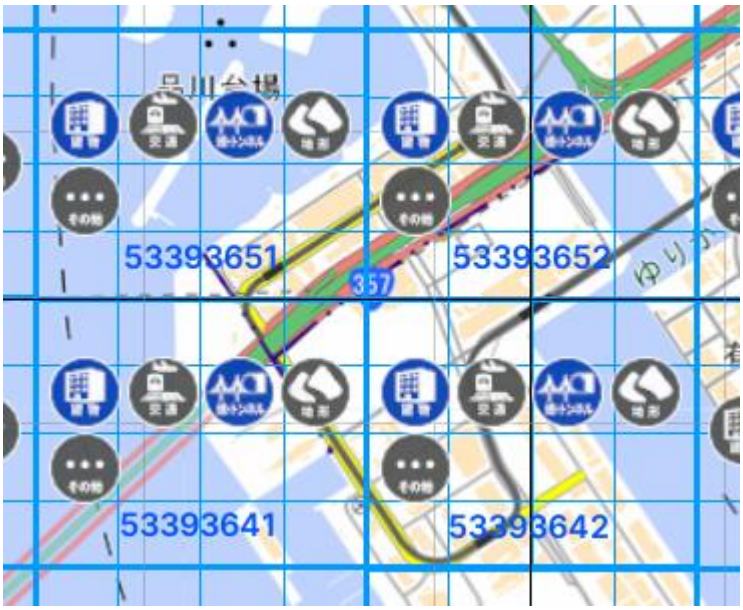


図1-2-7　LOD表示

最大LODの検索は次のように行う。

1. 各CityGMLファイルの中身を文字列検索し、”:lod（数字）” または “:lod>（数字）“ の数字にあたるものを探す。後者はdem（地形）に見られるLOD表記で、前者はそれ以外で見られるLOD表記である。
2. 見つかった数字の最大値をそのGMLで利用可能な最大LODとする。ただし、GMLファイルの全文検索は時間がかかるため、次の高速化手段を盛り込む。
3. 既定の容量を読み込んだ時点で探索を停止する。その容量とは最初のLOD表記が見つかった箇所を起点にして10MBである。2023年の東京、沼津、新潟の全データにおいて、これによる検索結果が全文検索と相違ないことを確認している。
4. 仕様上あり得る最大LODが見つかったら探索を中止する。

サーバーインポートの場合、上述のPLATEAU CMSサーバーから受け取ったJSONに利用可能な最大LODが記載されているため、ファイル内検索は行わずJSONの情報を利用する。

2.2.5：[共通ライブラリ] 地理院地図のダウンロード

範囲選択画面では、LODアイコンの表示と並行して国土地理院が提供する地理院地図がダウンロードされて表示される。
共通ライブラリは地理院地図の該当箇所となる地図タイル（画像）をダウンロードし、ゲームエンジンは地図画像を並べて範囲選択画面に表示する。

2.2.6：[ゲームエンジン] 範囲選択画面を表示

図1-2-8　範囲選択画面

ゲームエンジンは上記で取得した情報と地図画像を範囲選択画面に表示し、ユーザーに範囲の選択を促す。

ユーザーは5次メッシュ単位（約250m×約250m）で範囲を選択できる。1つのCityGMLファイルの範囲は3次メッシュ1つ相当（場合によっては2次メッシュ1つ相当）のため、5次メッシュ単位での選択は、3次メッシュGMLファイル1つを4×4分割した範囲の粒度での選択となる。

範囲選択画面での背景地図は、XYZタイルを使用している。XYZタイルではxyz座標に対応したタイル画像が配信されている。zの値はズームレベルであり、z=0は1枚の正方形の世界地図のタイルである。タイルを縦横に2×2分割することをn回繰り返したときのタイルの大きさはズームレベル z=n で表される。z回分割された世界地図タイルのうちどのタイルを使うかが xy座標 （左上がxy=（0,0））である。

範囲選択画面のカメラの位置から、適切なxyz形式に変換して地図タイルのxyz座標を求めて利用する。具体的には、範囲選択画面のカメラに映っている範囲に応じてズームレベルを動的に切り替え、またカメラの描画範囲から緯度経度範囲を計算し、それを内包するタイルの一覧を算出する。

2.2.7：[ゲームエンジン] インポート設定画面を表示

　ユーザーが範囲を決定すると、その範囲内で利用可能な地物型に関して設定画面が表示される。ここでの設定内容はインポート時に共通ライブラリに渡される。

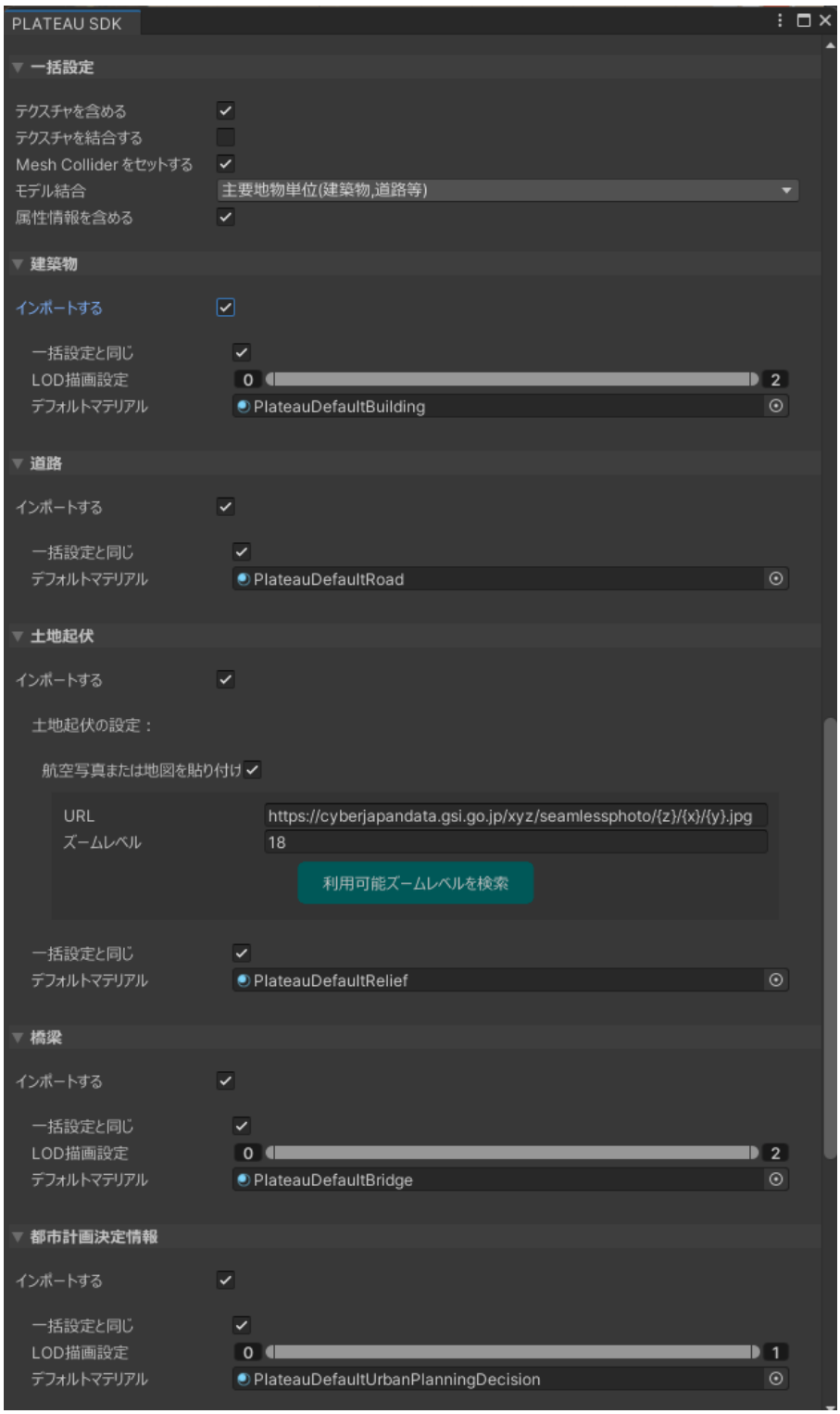


図1-2-9　インポート設定画面

2.2.8： [共通ライブラリ] データをダウンロードする

サーバーインポートの場合、共通ライブラリはHTTPS通信でPLATEAU CMSサーバーからPLATEAUデータをダウンロードする。

CityGMLファイルについては、2.2.2で記載しているJSONに含まれるURLからダウンロードする。テクスチャファイルについては、各CityGMLファイルの中身を検索し、そこに含まれるテクスチャパスをURLに変換してダウンロードする。

ローカルインポートの場合は、データはすでにコンピュータ内にあるため、この手順をスキップする。

2.2.9：[共通ライブラリ] GMLファイルをパースし、Modelに変換

インポートの対象となっている各GMLファイルに対してインポートを実行する。
共通ライブラリでのインポート処理とは、CityGMLファイルをパースし、Model（後述）に変換することである。変換は以下の手順で行われる。

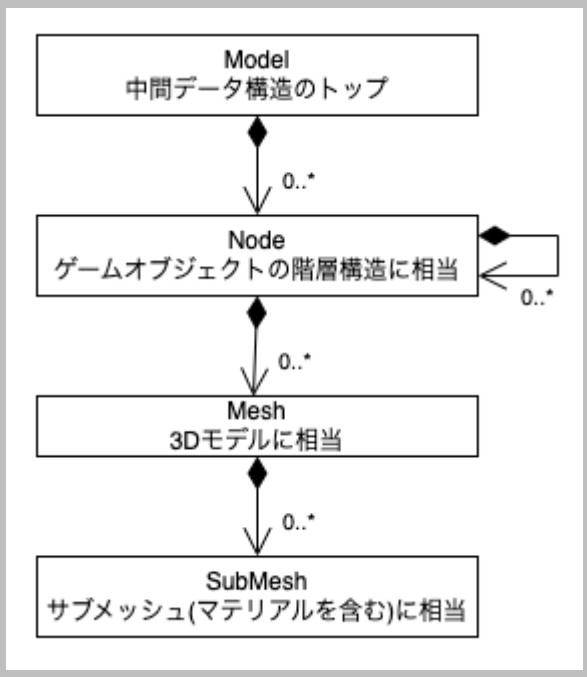
1. CityGMLファイルをパースする。パースにはオープンソースライブラリである libcitygml（https://github.com/jklimke/libcitygml）を用いる。ただし、PLATEAUの仕様に対応するためにLOD0・codelist・ADEによる拡張仕様への対応等独自改変をしている。改変したリポジトリのURLはhttps://github.com/Synesthesias/libcitygmlである。
2. GMLファイルの座標は緯度、経度、高さで表される一方で、ゲームエンジンで扱われる座標は直交座標系のため、直交座標への変換を行う。座標変換の式は国土地理院が公開している「Gauss-Krüger投影による経緯度座標及び平面直角座標相互間の座標換算についてのより簡明な計算方法（※）」を用いる。
3. インポートの設定で指定された結合粒度でポリゴンメッシュへの変換を行う。最小地物単位では最も細かい単位（例えば屋根面、壁面ごと）となる。主要地物単位はそれより大きい単位であり、（例えば建物ごとなど）CityGML上でgml:CityModelの直下に配置された地物の単位となる。地域単位はさらに大きい単位で、複数の建物を1つにまとめた単位である。インポートの設定が主要地物または地域単位である場合は、メッシュを指定の単位に結合する。この際、インポート後に属性情報を取得できるようにするため、追加の情報としてメッシュの各頂点と地物IDの対応関係を格納する。この詳細については属性情報取得機能の項で説明する。
4. ポリゴンメッシュをModelに格納する。

※出典／国土交通省　国土交通省地理院　国土地理院時報(2011,121集)
https://www.gsi.go.jp/common/000061216.pdf

NOTE：Modelとは
　Model は、GMLファイルパーサーから読み取った3Dメッシュやマテリアルを、UnityとUnreal Engineに渡すための中間データ構造として設計されたクラスである。
そのデータにはメッシュ、テクスチャパス、ゲームオブジェクトの階層構造が含まれており、Unity や Unreal Engine がメッシュやゲームオブジェクトを生成するために必要な情報が入るよう意図されている。Model はそのデータ構造の階層のトップに位置する。

　Model が所有する Node の階層関係は、ゲームエンジン側でのゲームオブジェクトの階層関係に対応する。
　Node が所有する Mesh は、そのゲームオブジェクトが保持する3Dメッシュに対応する。
Mesh が所有する SubMesh は、そのメッシュのサブメッシュ（マテリアル、テクスチャパスを含む）に対応する。

Modelデータ構造の階層：



2.2.10：[共通ライブラリ] テクスチャを結合

　下図はPLATEAUの複数のテクスチャを1つの画像に結合した例である。
インポート設定でテクスチャを結合するよう指定されている場合、共通ライブラリは複数枚のテクスチャをより少ない枚数のテクスチャにまとめる。この処理はテクスチャアトラス化と呼ばれ、ゲームエンジン側でマテリアル数の削減によってパフォーマンスを向上するために一般的に用いられている。

パフォーマンス向上の程度

　例えば、新宿区の2km×2kmのデータについてテクスチャ結合して4096×4096のテクスチャサイズにまとめたとき、テクスチャ結合しない場合と比較して、ドローコール数は4160から483に向上し、あるPC（PCスペック　CPU：AMD Ryzen 7 5800HS／GPU：GeForce RTX 3060 Laptop GPU ／メモリ：40GB）ではFPSが70から90に向上した。

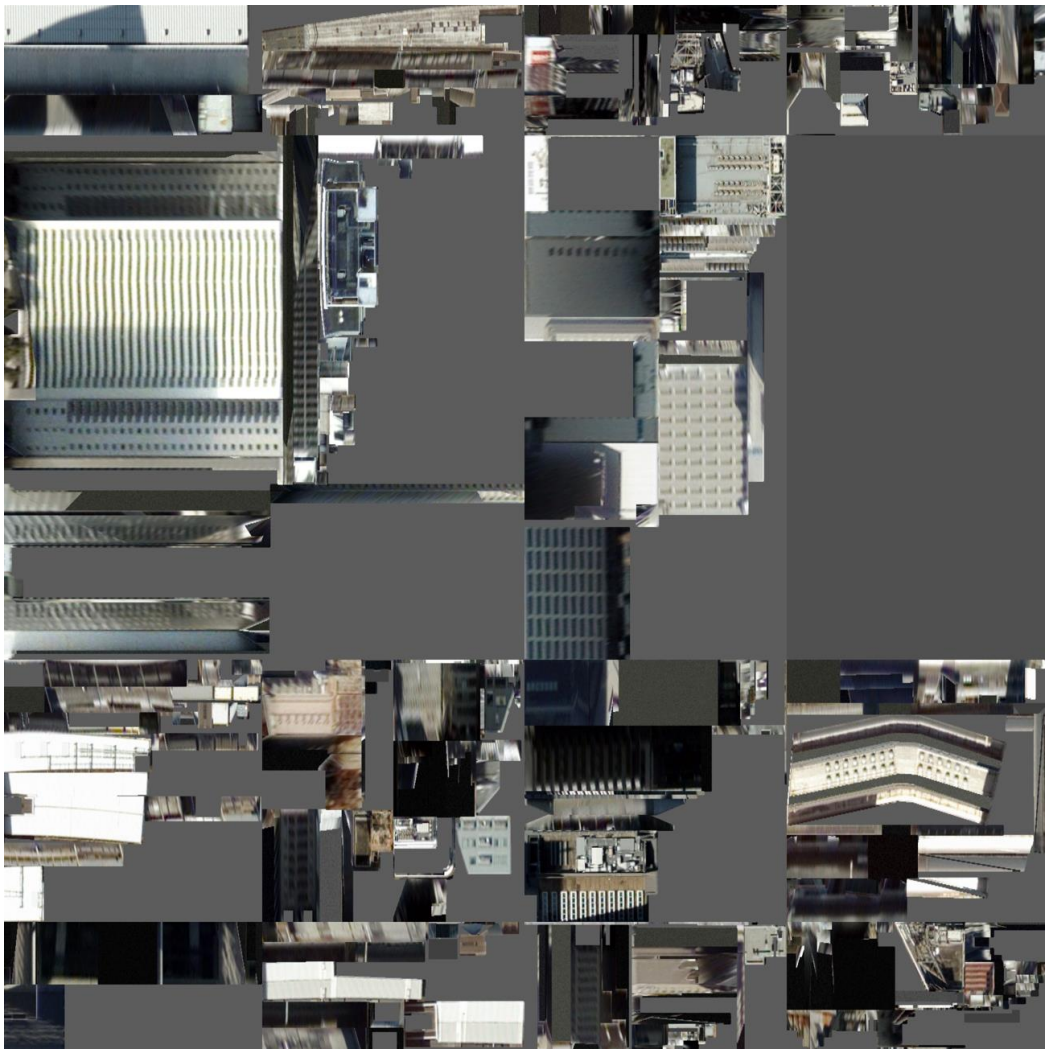


図1-2-10　テクスチャ結合

結合処理では最初に空の結合テクスチャのリストを保持し、Model内の各テクスチャについて以下を行うことで結合テクスチャに格納している。

1. テクスチャのサイズ取得：テクスチャの幅と高さを取得する。テクスチャの幅または高さが結合後テクスチャのサイズ以上であれば結合不可能なためスキップする。

2. 適切な結合先の検索：結合テクスチャリスト内の全てのテクスチャに対して、テクスチャが格納可能かどうかをチェックする。テクスチャを隙間なく結合するため、以下のアルゴリズムによって結合は行われる。
    1. 前提：各結合テクスチャの格納可能な領域として複数の「コンテナ」と「空き領域」に分けて扱っている。コンテナは結合テクスチャを縦に分割したものであり、コンテナ内に格納されるテクスチャの縦サイズは全て同じである。コンテナ領域以外の領域を空き領域と呼ぶ。
    2. テクスチャと縦サイズが同一かつ十分な領域が空いているコンテナを検索する。見つかった場合はそのコンテナにテクスチャを格納する。
    3. 見つからなかった場合は空き領域の縦サイズを取得し、テクスチャの縦サイズよりも大きい場合は新たにコンテナを作成して格納する。
    4. 上記処理で格納できなかった場合、この結合テクスチャには格納不可能であるとみなす。

3. 格納可能な結合テクスチャが見つからない場合の処理：どの結合テクスチャにも格納できない場合、新たに結合テクスチャを作成して格納する。

4. Meshの更新：各テクスチャが貼られているMeshのUV座標とSubMeshのテクスチャパスを、結合後のテクスチャに合わせて更新する。

全てのテクスチャの格納先が決定された後、全ての結合テクスチャを画像として出力する。

画像の読み込みと保存にはオープンソースライブラリであるlibpng（http://www.libpng.org/pub/png/libpng.html），libtiff （http://www.libtiff.org/），libjpeg-turbo（https://github.com/libjpeg-turbo/libjpeg-turbo）を利用する。

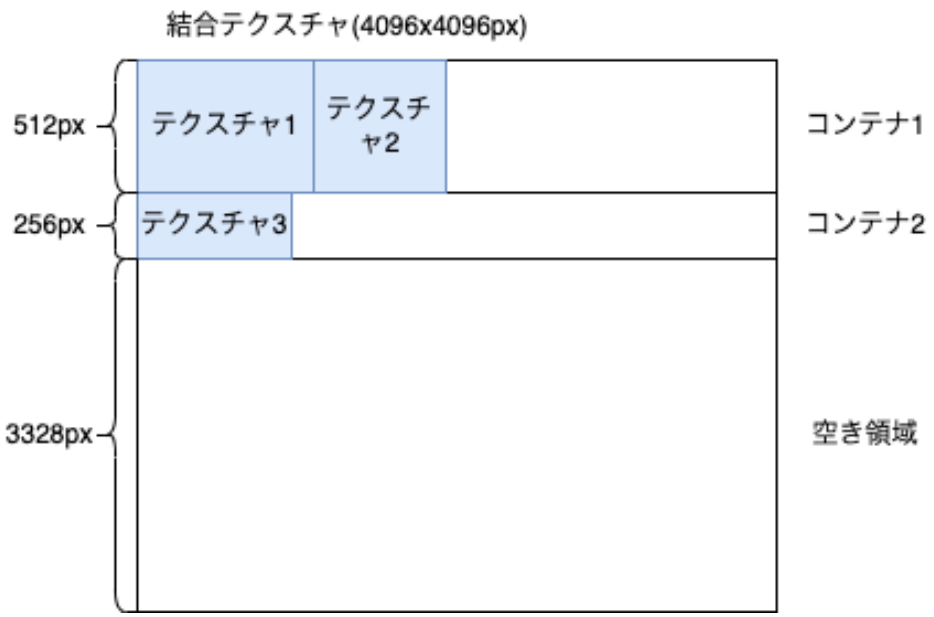


図1-2-11　テクスチャ結合アルゴリズム

2.2.11：[共通ライブラリ] 航空写真を地形に貼り付ける

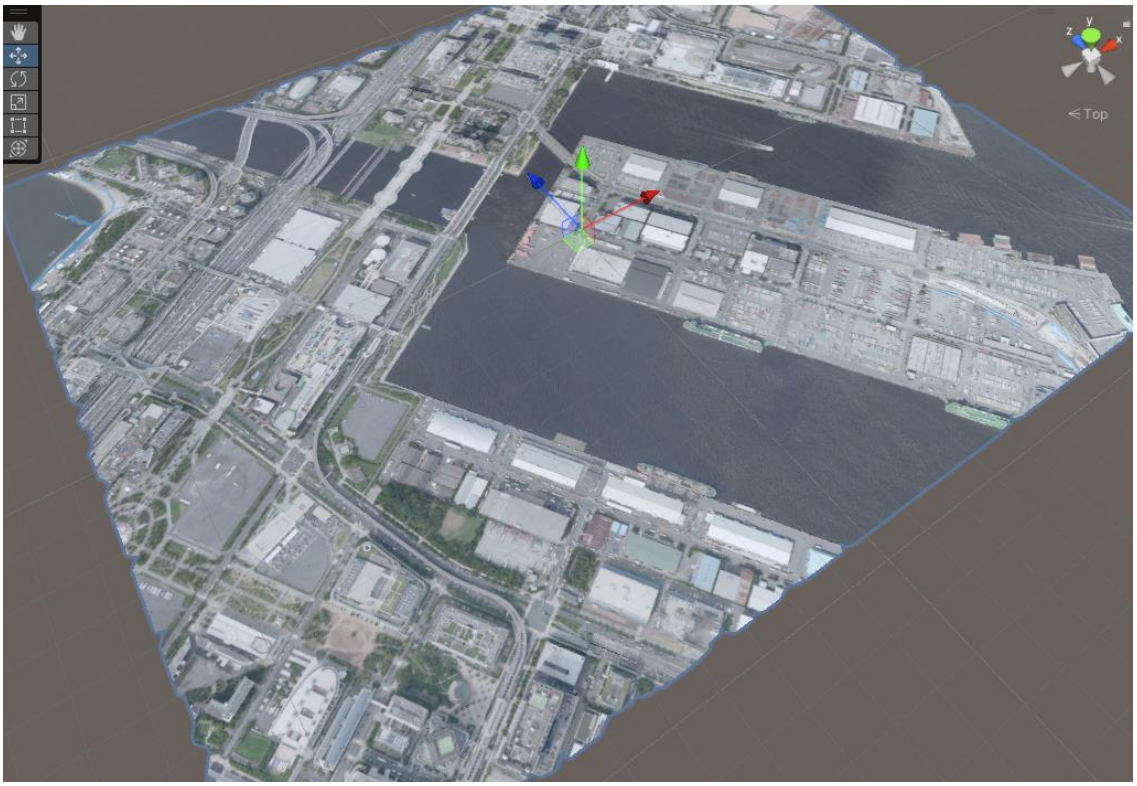


図1-2-12　航空写真の適用例

地形のインポート設定では、下図のように航空写真等の貼り付けに関する設定項目がある。

図1-2-13　航空写真貼り付け画面

設定項目に関する詳細を以下に記す。

- 「衛星写真または地図を貼り付ける」のチェックボックス
    - チェックが入っているとき、インポート時に地図タイルをダウンロードしてテクスチャとして貼り付ける。
- 地図タイルURL
    - {z}, {x}, {y} を含む地図タイルURLの入力欄
    - SDKは、地物の位置をもとに地図タイル座標を算出し、{z}, {x}, {y} を整数に置き換えてダウンロードする
    - 入力されたURLに {z}, {x}, {y} を含まない場合など、書式として正しくない場合は警告文を表示し、機能を無効化する
    - デフォルトでは地理院地図の航空写真 https://cyberjapandata.gsi.go.jp/xyz/seamlessphoto/{z}/{x}/{y}.jpg のURLになっているが、他の地図（標準地図、標高図など）も利用可能である
    - URLの入力欄に変更を加えると、入力欄の下に「決定」ボタンが表示される。それをクリックすると、ズームレベルの選択肢が更新される
- ズームレベル
    - 上記地図タイルURLの {z} の値を指定する
    - 例：14
    - ズームレベルは、利用可能な値のなかからドロップダウンから選べるようにする。デフォルトのズームレベル選択肢は、地理院地図の衛星写真で対応しているズームレベル14～18である。しかし、地図URLが変更されて決定ボタンが押されたとき、次の方法で利用可能なズームレベルを求め、選択肢を更新する
    - 利用可能なズームレベルを求めるための方法
        - 利用可能なズームレベルの範囲は地図の種類ごとに異なるが、ありうる最小は0、最大は19として走査する。
        - 求め方は、{x}, {y} に選択範囲の中心に近い値を入れ、{z}に0から18までの範囲を全探索で入れてアクセスを試みる。HTTPステータスが404なら対応範囲外、200なら対応範囲内とみなす。

次に、航空写真貼り付けの処理内容を以下に記す。

1. 共通ライブラリ（クラス名： MapAttacher）は、地図タイルURLテンプレートを受け取る。
2. インポートする地形について、座標変換をして地図タイルURLの {x}, {y} の具体的な数値を求める。{z} はユーザーが指定した値である。
    1. 変換前の例
       https://cyberjapandata.gsi.go.jp/xyz/seamlessphoto/{z}/{x}/{y}.jpg
    2. 変換後の例 （範囲によっては該当する {x}, {y} の組が複数ある。
       https://cyberjapandata.gsi.go.jp/xyz/seamlessphoto/14/10/10.jpg
       https://cyberjapandata.gsi.go.jp/xyz/seamlessphoto/14/11/10.jpg
3. 各URLで画像をダウンロードする。ダウンロード処理は範囲選択画面の地図ダウンロード機能を流用する。
4. 各画像を結合して1枚の画像にしたあと、地形の範囲外の箇所をトリミングする。地形モデルの南西端が画像の左下端と対応し、地形モデルの北東端が画像の右上端と対応するようにする。
5. 地形モデルと地図タイルの画像を合わせるため、地形モデルにUVを設定する。モデルのUV座標は、南西端が （0,0） , 北東端が （1,1） とし、その間は各頂点の東西南北の位置で線形補間する。
6. マテリアルのテクスチャパスを結合後の画像で置き換える。
7. 結合前の画像は利用しないので削除する。
8. 共通ライブラリは、UVとテクスチャパスが再設定された地形のModelを出力してゲームエンジンに渡す。

ゲームエンジンは、地形向けに再設定されたModelを受け取りシーンに配置する。

2.2.12 : [ゲームエンジン] Modelからゲームオブジェクトに変換して配置する

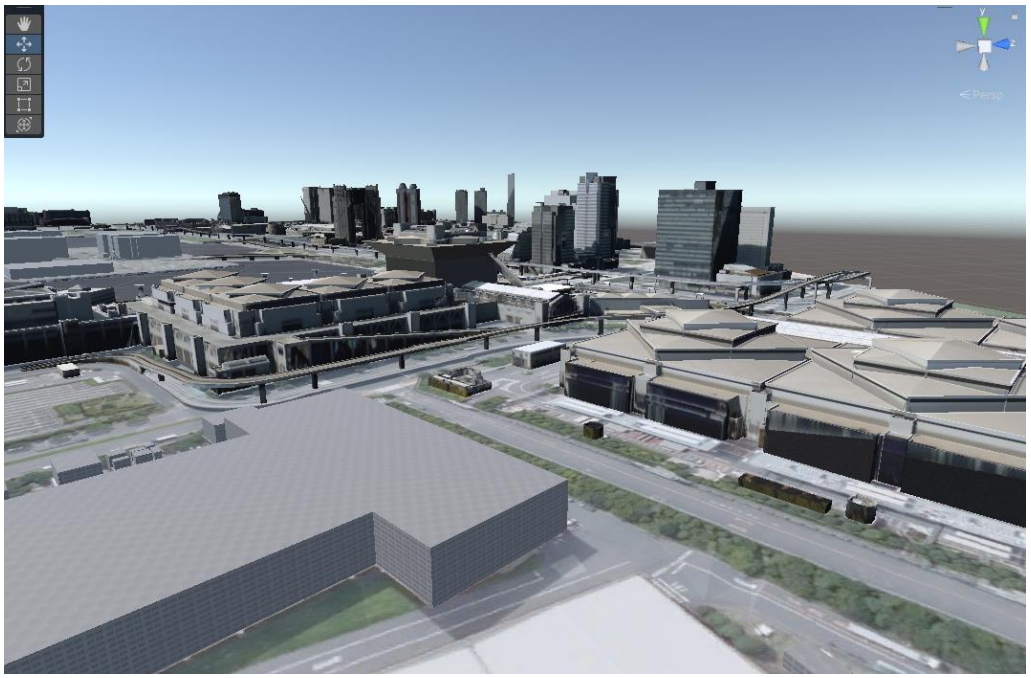


図1-2-14　インポートされた3D都市モデル

ゲームエンジンは共通ライブラリからModelを受け取り、それをシーン、レベルに配置する。
その手順は次のとおりである。

1. Nodeの階層構造を走査し、Nodeと対応するようにゲームオブジェクトを生成する。
2. Meshの頂点、インデックス情報、UV1、UV4をコピーすることで、ゲームオブジェクト上のメッシュを生成する。
3. SubMeshについては、テクスチャパスが指定されている場合は、そのテクスチャを当てはめて適用する。テクスチャの代わりに数値でのマテリアル指定がなされている場合は、「数値によるマテリアルのインポート」で後述する。
4. Model内の各頂点の座標は平面直角座標系となっているが、ゲームエンジンでそのままの座標で描画しようとするとfloatの丸め誤差によって描画が乱れてしまう。そのため、インポート範囲の中心が原点となるように各頂点を平行移動する。なお原点位置は設定により変更可能である。
5. 属性情報は、上述のGMLパース時に得たものをもとにコンポーネントに保存する。詳細は「2.4 属性情報取得機能」を参照されたい。

数値によるマテリアルのインポート

PLATEAUの3D都市モデルの見た目は、CityGML内でテクスチャによって表現されている場合の他に、emissiveColor等の数値によってパラメーターが指定されている場合がある。SDKでは次のパラメーターをゲームエンジンでのマテリアルパラメーターに変換している。

- app:ambientIntensity
    - 周囲光からの影響の強さ
- app:diffuseColor
    - 拡散反射
- app:emissiveColor
    - 発光
- app:specularColor
    - 鏡面反射
- app:shininess
    - 光沢
- app:transparency
    - 透過度

前述のCityGMLのマテリアル情報とUnity、Unreal Engineの対応関係は次の表のとおりである。この表に基づきゲームエンジンでマテリアルを生成する。

Unity

PLATEAU X3DMaterial	Unity Standard(Specular Setup)シェーダーに類似した独自シェーダー
app:ambientIntensity	Occlusion ただし、Standardシェーダーでは画像しか指定できないので float での指定を受け付けるシェーダーを作成する。
app:diffuseColor	Albedo
app:emissiveColor	Emission/Color
app:specularColor	Specular
app:shininess	Smoothness
app:transparency (0で不透明、1で透明)	Albedo / アルファ値 (1で不透明、0で透明) 不透明(アルファが1)の場合は Opaque モード、透明(アルファが1以外)の場合はTransparentモードになる。

Unreal Engine

PLATEAU X3DMaterial	Unrealマテリアル
app:ambientIntensity	Ambient Occlusion
app:diffuseColor	Base Color
app:emissiveColor	Emissive Color
app:specularColor	下記「Unrealのスペキュラについて」を参照
app:shininess (0でザラザラ感、1でツヤツヤ感)	Roughness (0でツヤツヤ感、1でザラザラ感)
app:transparency (0で不透明、1で透明)	Opacity (1で不透明、0で透明)

Unrealのスペキュラについて
　PLATEAUのスペキュラはRGBの3値であるのに対し、Unrealのスペキュラはfloat1つであり型が異なる。
また、UnrealではUnityと違ってスペキュラを1にしても輝いている質感がそれほど出ない。
そこで、Unrealでは次の式でスペキュラとメタリックを利用する。

ベースカラー（拡散反射）とスペキュラの R:G:B の比率がほぼ同じ場合は
(Unrealのメタリック) = (PLATEAUのスペキュラのR) / (PLATEAUのベースカラーのR)

図1-2-15　メタリック=1, スペキュラ=デフォルト(0.5), ラフネス=0.1 の場合

CityGMLのスペキュラのR,G,Bの値がほぼ同じ場合はメタリックは0でスペキュラの値をそのまま適用する。

図1-2-16　メタリック=0, スペキュラ=1, ラフネス=0.1 の場合

一般的にはベースカラーと違う色で反射させることはないためこのような処理で変換を行っている。すなわち、ベースカラーのR:G:Bの比率とスペキュラのR:G:Bの比率がおおむね同じ、またはスペキュラが無彩色（黒、白、グレー）になっていると想定する。

2.2.13 : [ゲームエンジン] デフォルトマテリアルを適用する

GMLファイルのパースの結果、3Dモデルのうちマテリアル指定がない箇所については、ゲームエンジンは地物タイプに応じたデフォルトマテリアルで置き換える。

下図はSDKに同梱されたデフォルトマテリアルのうち、建築物向けのものを適用した例である。

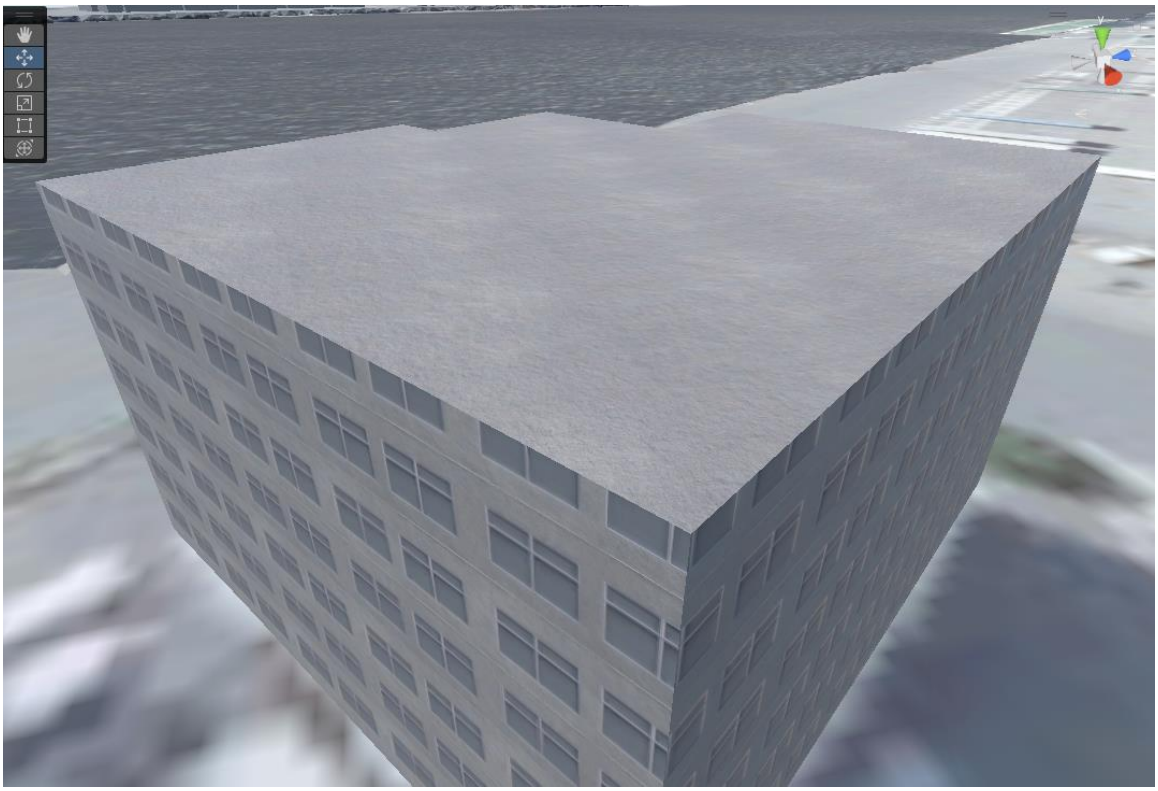


図1-2-17　デフォルトマテリアルの適用例

3D都市モデルにおいて、マテリアルやテクスチャの指定がない箇所は通常UVを持たない。そのため、デフォルトマテリアルはUVがなくても動作するよう、Triplanarシェーダーを利用する。Triplanarシェーダーとは、上、奥及び横方向からテクスチャパターンを投影するように動作し、UVを必要としないシェーダーである。SDKは、Triplanarシェーダーを利用したマテリアルをデフォルトマテリアルとして地物タイプ別に同梱している。

テクスチャがなく、したがってUVのない地物には、次のものがある。

- LOD1以下の建築物
- LOD2以下の交通（道路）
- 土地利用
- 災害リスク
- 都市計画決定情報
- 水部
- 地形
- 区域
- 汎用都市オブジェクト

下図は、Triplanarシェーダーの仕組みを図示したものである。

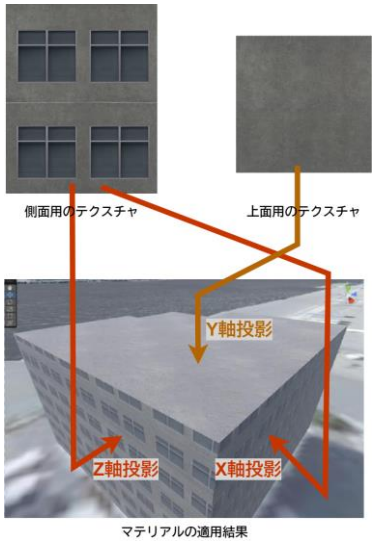


図1-2-18 Triplanarシェーダーの仕組み

Triplanarシェーダーでは、UVによってテクスチャを貼り付ける代わりに、ワールド座標系における X軸、Y軸、Z軸の方向にテクスチャを投影することでUVを不要としている。
UnityではShaderGraphを使って実装されおり、次の3つのシェーダーをSDKに同梱している。

- PlateauTriplanarShader(Opaque)
- PlateauTriplanarShader(Transparent)
- PlateauTriplanarShader(DualTextures)

PlateauTriplanarShader(Opaque)はX軸、Y軸、Z軸すべてに同じテクスチャを投影するシェーダーであり、次の地物のデフォルトマテリアルで利用されている。

- 橋梁
- 都市設備
- 鉄道
- 土地起伏
- 道路
- 広場
- 徒歩道
- トンネル
- 地下街
- 地下埋設物
- 水路
- 「その他」分類

PlateauTriplanarShader(Transparent)はPlateauTriplanarShader(Opaque)に透明度を付与する機能を付与したものである。なお、透明と不透明でシェーダーを分けることは、描画負荷の改善のためにゲームエンジン利用者の間でよく用いられる手法である。このシェーダーは次の地物のデフォルトマテリアルで利用されている。

- 災害リスク
- 土地利用
- 都市計画決定情報
- 植生
- 航路

PlateauTriplanarShader(DualTextures)は、X軸・Z軸（側面方向）と、Y軸（上下方向）で異なるテクスチャを投影する不透明シェーダーである。このシェーダーは建築物のデフォルトマテリアルにおいて、上面は屋根のテクスチャ、側面は窓を含んだ壁のテクスチャを割り当てる形で利用されている。

Triplanarシェーダーの実装の詳細を以下で解説する。
下図はPlateauTriplanarShader(Opaque)の実装である。

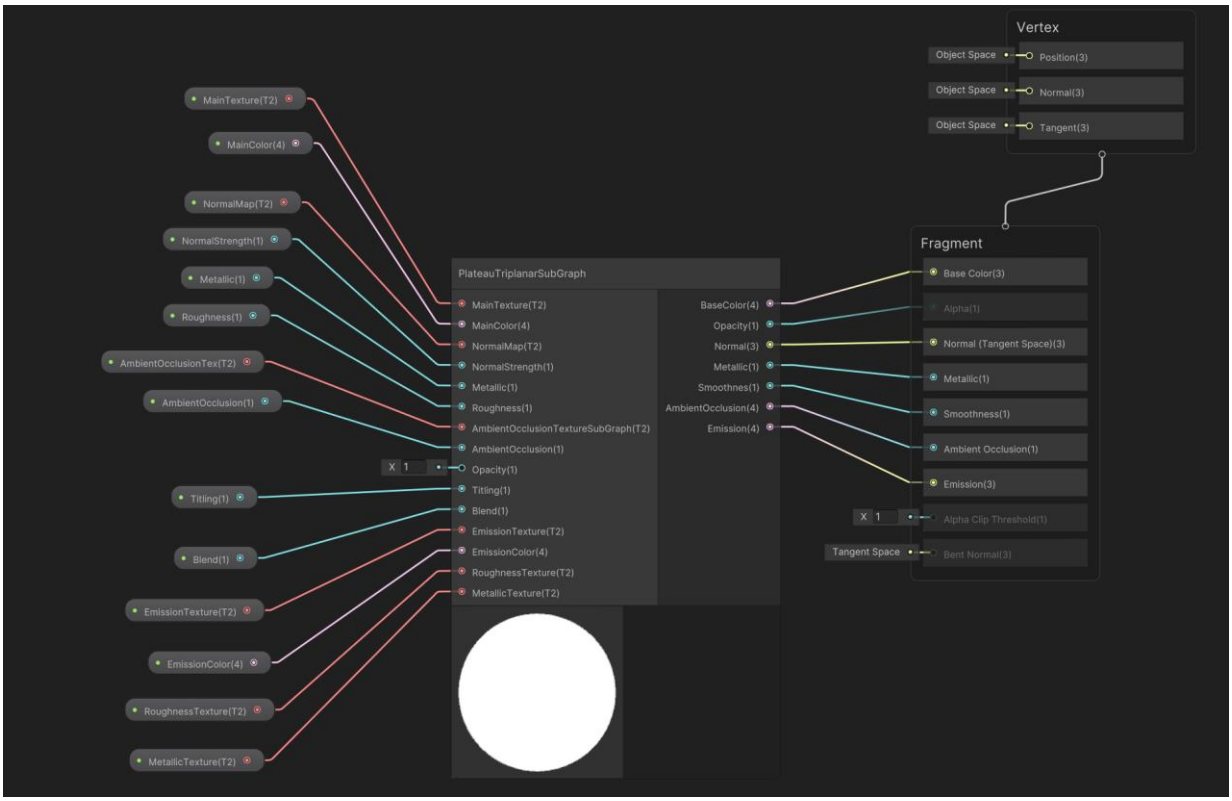


図1-2-19 PlateauTriplanarShader(Opaque)の実装

シェーダーのパラメーターは次のとおりである。

- MainTexture
  ベースカラーとなるテクスチャ
- MainColor
  MainTextureと乗算する色
- NormalMap
  ノーマルマップのテクスチャ
- NormalStrength
  ノーマルマップを適用する強さ
- MetallicTexture
  メタリックの強さを表現するテクスチャ
- Metallic
  MetallicTextureを適用する強さ
- RoughnessTexture
  ラフネスの強さを表現するテクスチャ
- Roughness
  RoughnessTextureを適用する強さ
- AmbientOcclusionTex
  アンビエントオクルージョンを表現するテクスチャ
- AmbientOcclusion
  AmbientOcclusionTexを適用する強さ
- Tiling
  投影して並べるテクスチャの大きさ調整
- EmissionTexture
  エミッションを表現するテクスチャ
- EmissionColor
  EmissionTextureに乗算する色
- Blend
  XYZそれぞれの投影をブレンドする強さ

上記のパラメーターをPlateauTriplanarSubGraphの入力に渡し、同サブグラフの出力をシェーダーグラフの出力に繋ぐ。ここでPlateauTriplanarSubGraphの実装は次のとおりである。

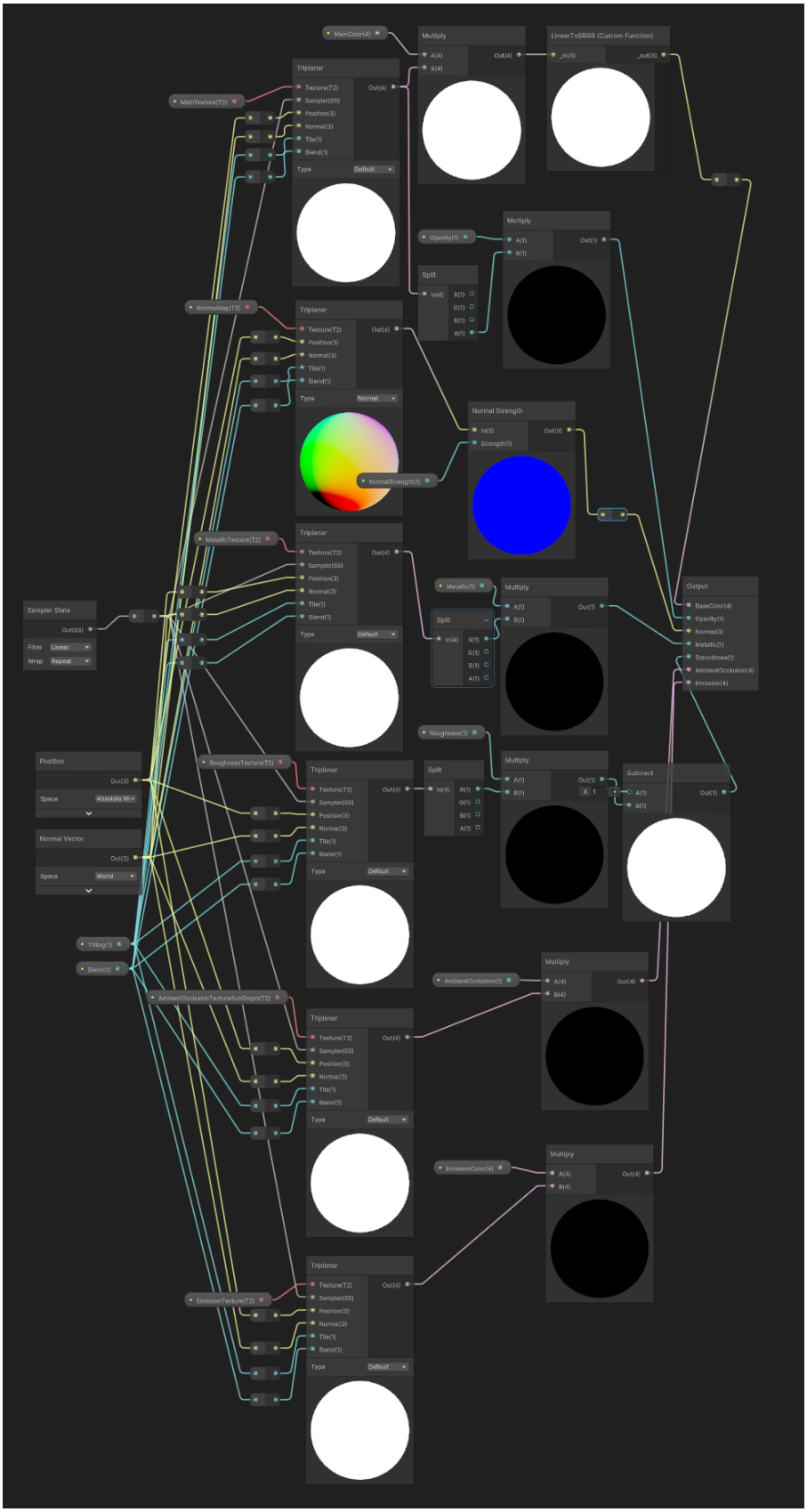


図1-2-20 PlateauTriplanarSubGraphの実装

上図のとおり、ベースカラー、ノーマル、メタリック、アンビエントオクルージョン、エミッションのテクスチャをそれぞれTriplanarノードに与えている。

Triplanarノードについて：

TriplanarノードはUnityのShaderGraphで標準のノードであり、ワールド空間への投影によってx,y,z軸で1回ずつテクスチャサンプリングする。もっとも法線の向きに合った平面の影響を大きくして出力する。
このノードの機能は以下のコードと等価である。

```
// Normal以外の場合
float3 Node_UV = Position * Tile;
float3 Node_Blend = pow(abs(Normal), Blend);
Node_Blend /= dot(Node_Blend, 1.0);
float4 Node_X = SAMPLE_TEXTURE2D(Texture, Sampler, Node_UV.zy);
float4 Node_Y = SAMPLE_TEXTURE2D(Texture, Sampler, Node_UV.xz);
float4 Node_Z = SAMPLE_TEXTURE2D(Texture, Sampler, Node_UV.xy);
float4 Out = Node_X * Node_Blend.x + Node_Y * Node_Blend.y + Node_Z * Node_Blend.z;

// Normalの場合
float3 Node_UV = Position * Tile;
float3 Node_Blend = max(pow(abs(Normal), Blend), 0);
Node_Blend /= (Node_Blend.x + Node_Blend.y + Node_Blend.z ).xxx;
float3 Node_X = UnpackNormal(SAMPLE_TEXTURE2D(Texture, Sampler, Node_UV.zy));
float3 Node_Y = UnpackNormal(SAMPLE_TEXTURE2D(Texture, Sampler, Node_UV.xz));
float3 Node_Z = UnpackNormal(SAMPLE_TEXTURE2D(Texture, Sampler, Node_UV.xy));
Node_X = float3(Node_X.xy + Normal.zy, abs(Node_X.z) * Normal.x);
Node_Y = float3(Node_Y.xy + Normal.xz, abs(Node_Y.z) * Normal.y);
Node_Z = float3(Node_Z.xy + Normal.xy, abs(Node_Z.z) * Normal.z);
float4 Out = float4(normalize(Node_X.zyx * Node_Blend.x + Node_Y.xzy * Node_Blend.y +
Node_Z.xyz * Node_Blend.z), 1);
float3x3 Node_Transform = float3x3(IN.WorldSpaceTangent, IN.WorldSpaceBiTangent,
IN.WorldSpaceNormal);
Out.rgb = TransformWorldToTangent(Out.rgb, Node_Transform);
```

Triplanarノードの出力にNormal Strength、Metallicなどパラメーターで指定された強さを乗算し、サブグラフの出力としている。

PlateauTriplanarShader(Transparent)の実装については、PlateauTriplanarShader(Opaque)の実装に不透明度パラメーターを追加したものである。

PlateauTriplanarShader(DualTextures)の実装は下図のとおりである。

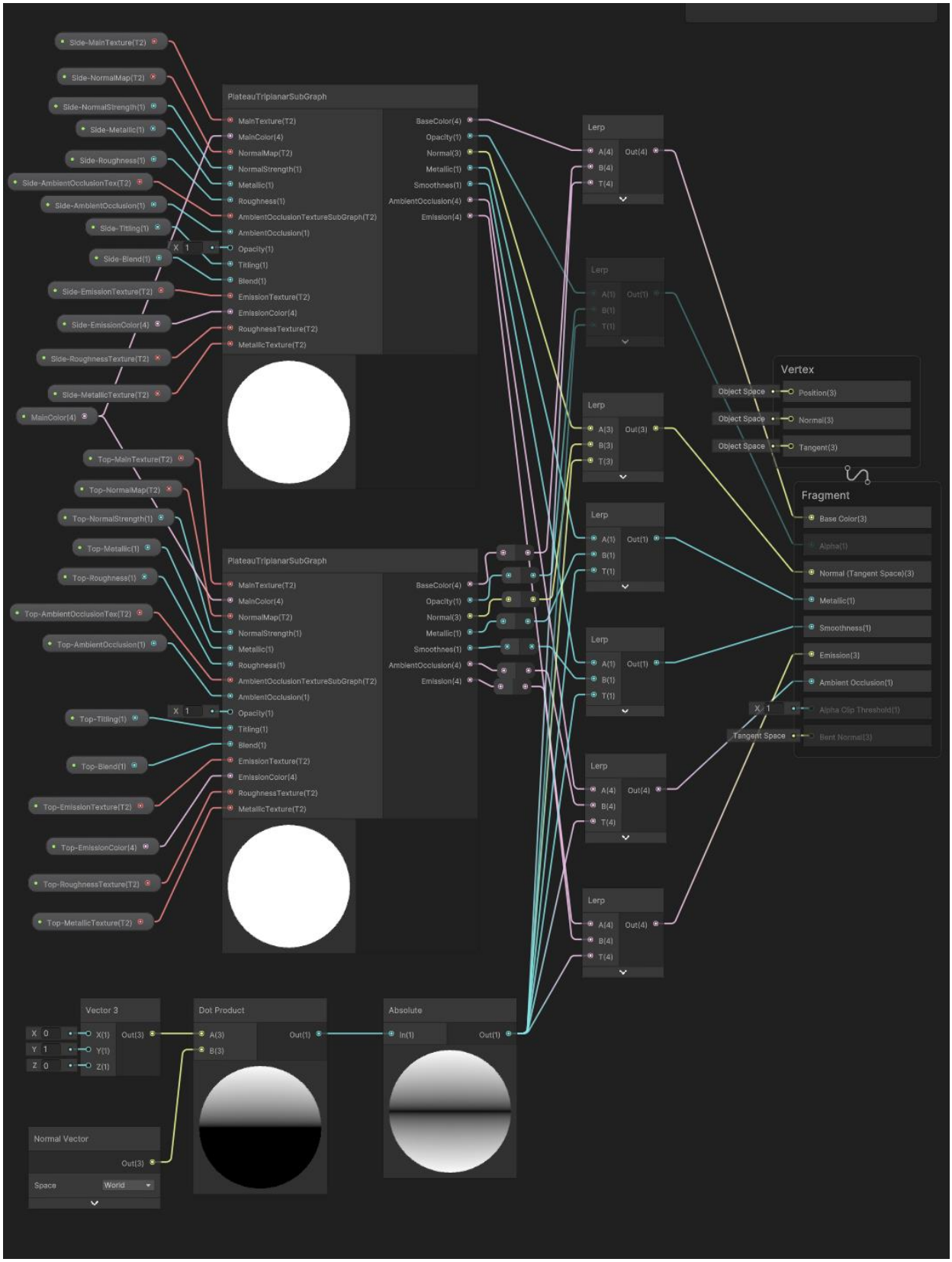


図1-2-21 PlateauTriplanarShader(DualTextures)の実装

上図のとおり、側面用と上面用で2つのPlateauTriplanarSubGraphを利用している。
入力パラメーターはそれぞれ側面用のSideと上面用のTopの2つに分かれ、したがってパラメーターの数は2倍になっている。
2つのPlateauTriplanarSubGraphのブレンドに関しては、法線と上ベクトルの内積をブレンド係数とすることで、上面と側面の表示を分けている。

Unrealでも同様の仕組みをマテリアルによって実装している。

2.3 3D都市モデルエクスポート機能

エクスポート機能では、インポートされた3D都市モデルを3Dファイル形式でエクスポートすることができる。

処理の流れは以下のとおりである。

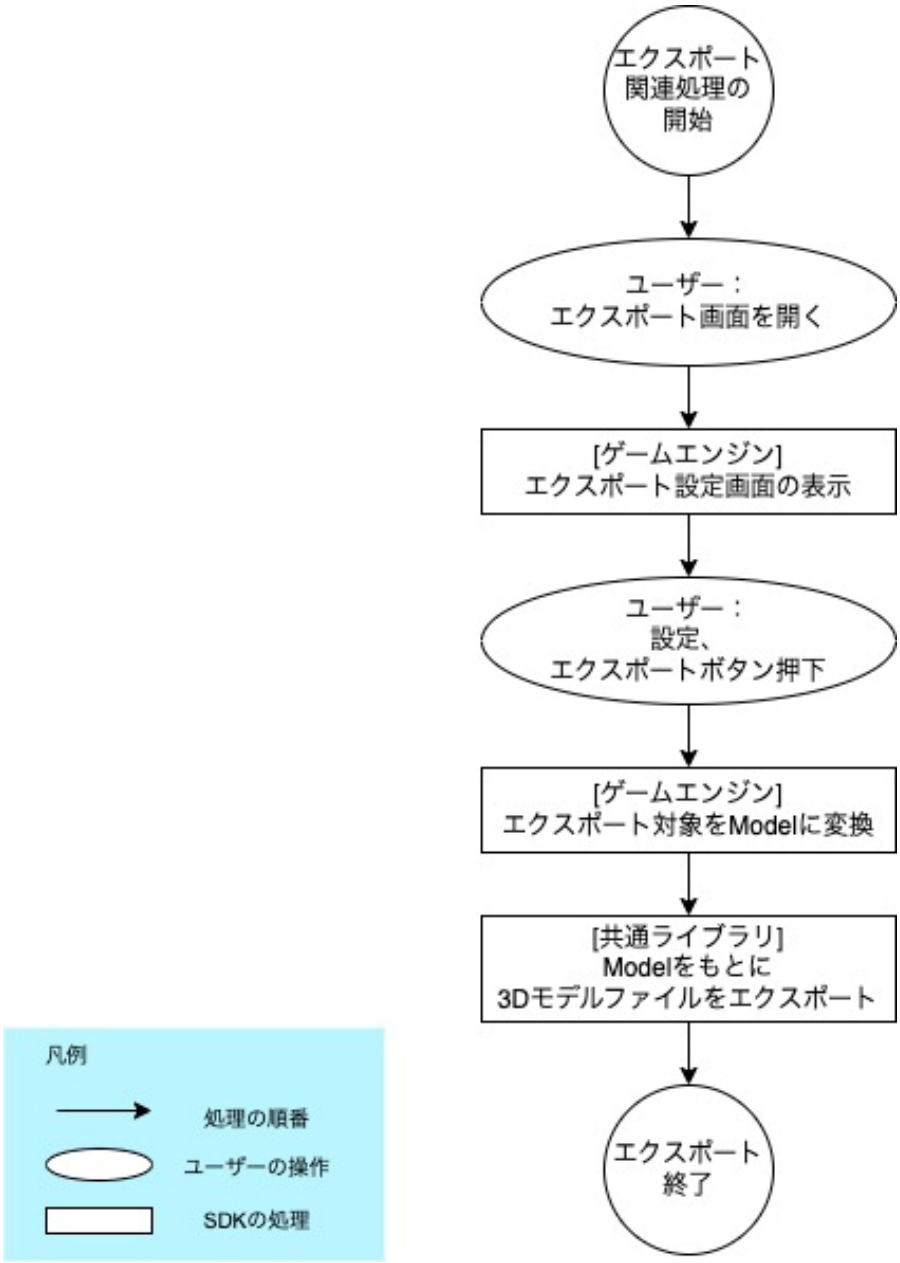


図1-2-22 エクスポート処理の流れ

2.3.1：[ゲームエンジン] エクスポート設定画面の表示

図1-2-23　エクスポート画面

エクスポートで設定可能な項目は上図のとおりである。

2.3.2：[ゲームエンジン] エクスポート対象をModelに変換

インポートの項で述べたように、Modelは共通ライブラリとゲームエンジンの間で3Dモデルとゲームオブジェクト階層をやりとりするための中間データ形式である。インポート時にはModelをもとにゲームオブジェクトが生成されるが、エクスポート時はその逆変換が行われる。変換処理においてユーザーが指定したエクスポート対象とその子ゲームオブジェクトの階層、3Dモデルのメッシュ、テクスチャパスはModelに格納され、共通ライブラリに渡される。また、オプションとしてテクスチャの有無、非アクティブオブジェクトの有無、座標変換、座標軸の変換もゲームエンジンの処理として行われる。

2.3.3：[共通ライブラリ] Modelをもとに3Dモデルファイルをエクスポート

共通ライブラリは、受け取ったModelを3Dモデルファイルとして書き出す。また、3Dモデルファイルから参照されるテクスチャファイルがコピーされる。
glTF形式のファイルを扱うため、オープンソースライブラリのglTF-SDK（https://github.com/microsoft/glTF-SDK）を利用する。
またFBX形式のファイルを扱うため、サードパーティライブラリのFBX SDK（https://aps.autodesk.com/developer/overview/fbx-sdk）を利用する。OBJ形式は構造が単純なため書き出し処理の実装は容易である。また、汎用的なライブラリが存在しないため、OBJのエクスポート処理は自前で実装している。

2.4 属性情報取得機能

属性情報取得機能では、クリックされた地物の属性情報を表示する機能と、実行時に属性情報を取得するためのAPIを提供している。

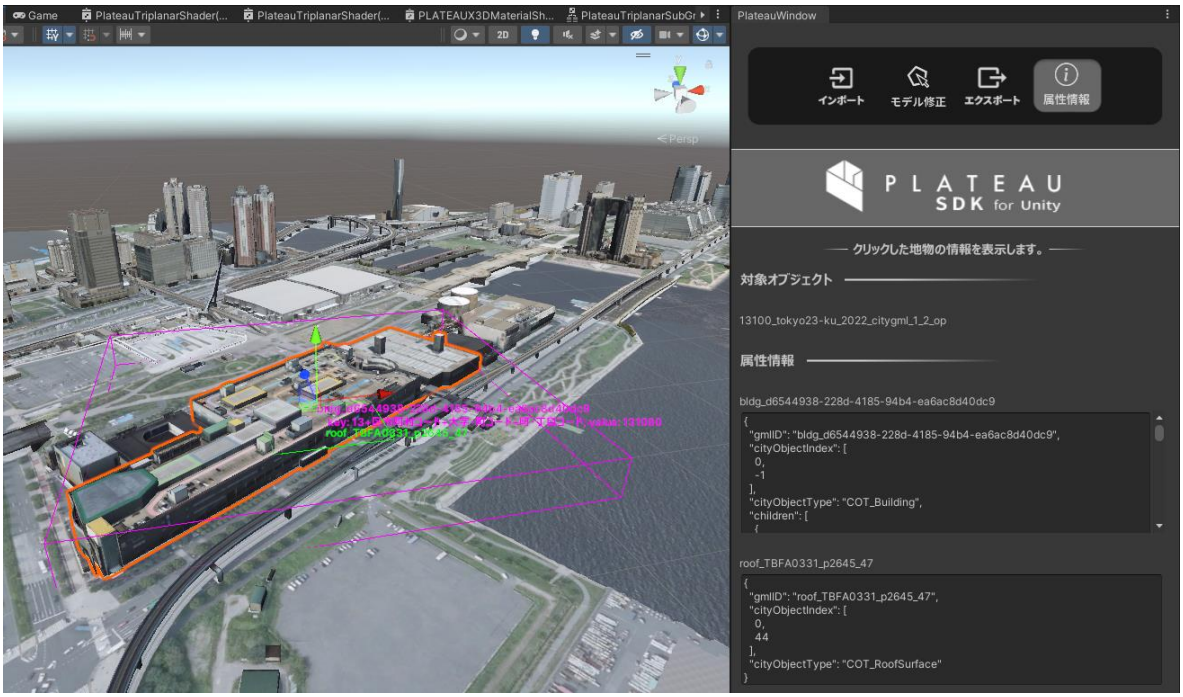


図1-2-24　属性情報の表示例

インポートの項で述べたように、インポート時、CityGMLファイルのパースによって取得された属性情報はゲームエンジン内に保存される。

PLATEAU SDKでは、ゲームエンジンでCityGMLを扱うために構造を単純化しており、地物を主要地物（建築物、道路等）と最小地物（LOD2以上の構成部品。屋根面、壁面等）に分類している。ゲームエンジンにインポートされた3D都市モデルは、以下の概要図のように主要地物の下に最小地物が含まれる構成になっている。

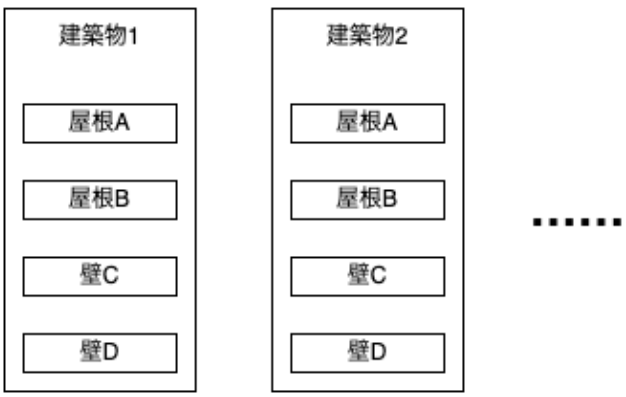


図1-2-25 SDKでの地物構成

図の中の「建築物1」「建築物2」が主要地物であり、その下の階層にある「屋根A」「壁C」等が最小地物である。

- インポート時にゲームオブジェクトの粒度で「最小地物単位」を選択した場合、最小地物ごと（上の例では屋根、壁ごと）にゲームオブジェクトが生成される。
- 粒度設定で「主要地物単位」を選択した場合、最小地物のメッシュは結合され、主要地物ごと（上の例では建築物ごと）にゲームオブジェクトが生成される。
- 粒度設定で「地域単位」を選択した場合、各建物も結合され、全体で地域ごとにまとまったゲームオブジェクトが生成される。

以上の理由により、ゲームオブジェクトと地物は必ずしも1対1に対応するとは限らない。主要地物単位及び地域単位では、1つのゲームオブジェクトの中に複数の地物、複数の属性情報が含まれることとなる。

この場合、1つのゲームオブジェクトのメッシュの中のどの部分が建物1で、どの部分が壁Cなのかを判別する手段が必要となる。その手段は次のとおりである。

インポート時、属性情報を取得できるようにするため、ポリゴンメッシュのどの面が主要地物、最小地物に対応するのかを識別するためのインデックス（主要地物インデックス、最小地物インデックス）をそれぞれポリゴンメッシュのUV4 （4番目のUV）のx、yに格納し、各インデックスと地物IDの対応関係を別途保持する。具体的には、地域単位で結合されており1つのポリゴンメッシュに複数の主要地物が含まれる場合は、1つのポリゴンメッシュのUV4のx座標として複数の値が格納される。さらに、1つのNodeに複数の最小地物が含まれる場合は1つのポリゴンメッシュのUV4のy座標として複数の値が格納される。最小地物単位の場合は主要地物と最小地物は1つしかないためUV4の座標値は常に（0,0）となる。ここで格納された各地物インデックスとgml:idの対応関係は別途保持される。

以上の工程により、UV4を確認することで1つのポリゴンメッシュのうちどの面がどの地物の形状を表すのか識別することができる。

ゲームエンジンで属性情報を保存するにあたっては、地物インデックスとそれに対応する属性情報を下図のようなJSON形式で保存している。


```
{
 "rootCityObjects": [
  {
   "gmlID": "bldg_d6544938-228d-4185-94b4-ea6ac8d40dc9",
   "cityObjectIndex": [
    0,
    -1
   ],
   "cityObjectType": "COT_Building",
   "children": [
    {
     "gmlID": "gnd_TBFA0331_b_0",
     "cityObjectIndex": [
      0,
      0
     ],
     "cityObjectType": "COT_GroundSurface"
    },
    {
     "gmlID": "gnd_TBFA0331_b_1",
     "cityObjectIndex": [
      0,
      1
     ],
     "cityObjectType": "COT_GroundSurface"
    },
```


図1-2-26　属性情報のJSON例

2.5 ゲームオブジェクトON/OFF機能

ゲームオブジェクトON/OFF機能は、地物タイプとLODによる条件指定に基づいて、一括でゲームオブジェクトの表示/非表示を切り替える機能である。

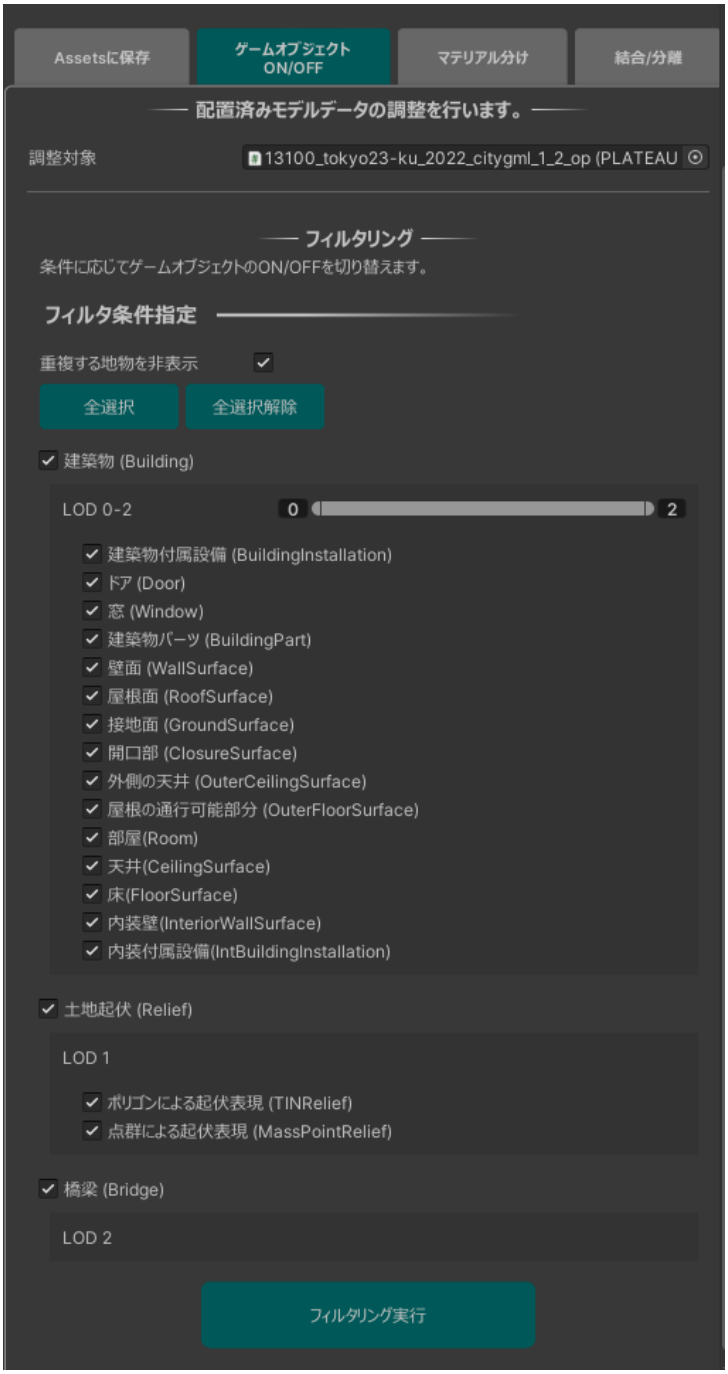


図1-2-27　ゲームオブジェクトON/OFF画面

この機能の処理は共通ライブラリ側は関与せず、ゲームエンジン側のみで行われる。
「フィルタリング実行」ボタンが押されると、調整対象として指定されたオブジェクトとその子オブジェクト全てについてLOD、地物型の条件に合うものを表示し、合わないものを非表示にする。表示・非表示化はUnityではゲームオブジェクトのアクティブ状態の切り替え、Unrealではコンポーネントの可視化状態（IsVisible）の切り替えによって実装されている。また、「重複する地物を非表示」のオプションがONの場合、同じ地物に相当する各オブジェクトのうちLODが最も高いもの以外は非表示化される。このオプションがOFFの場合は各地物について複数のLODが重複して表示される。

2.6 分割結合機能

分割結合機能は、インポートされた3D都市モデル内の各オブジェクトの結合粒度を変更する機能である。
その処理の流れは次のとおりである。

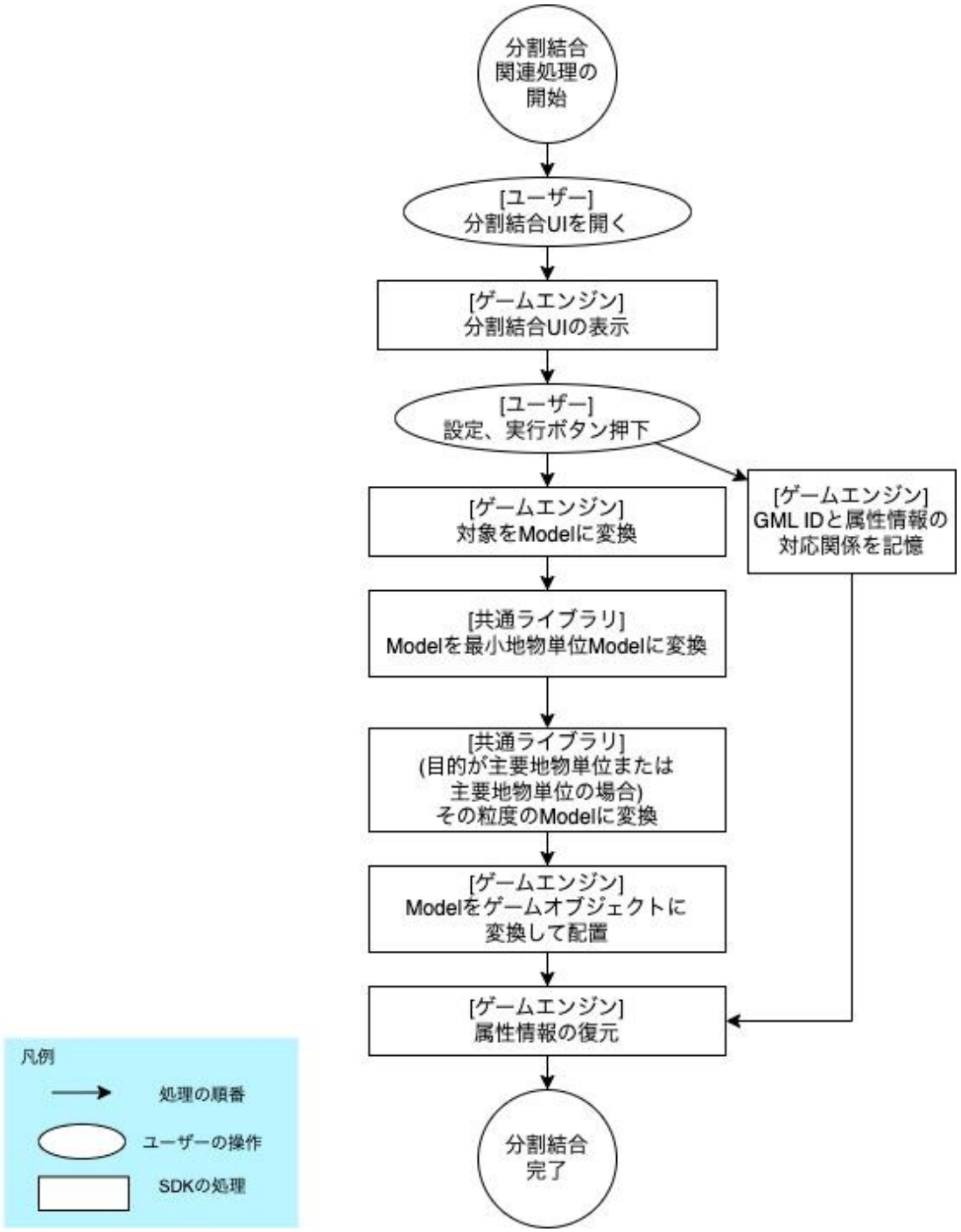


図1-2-28 分割結合処理の流れ

2.6.1 : [ゲームエンジン] 分割結合UIの表示
　対象オブジェトの配列と、分割・結合単位（地域単位、主要地物単位、最小地物単位）をユーザーに選択させる。

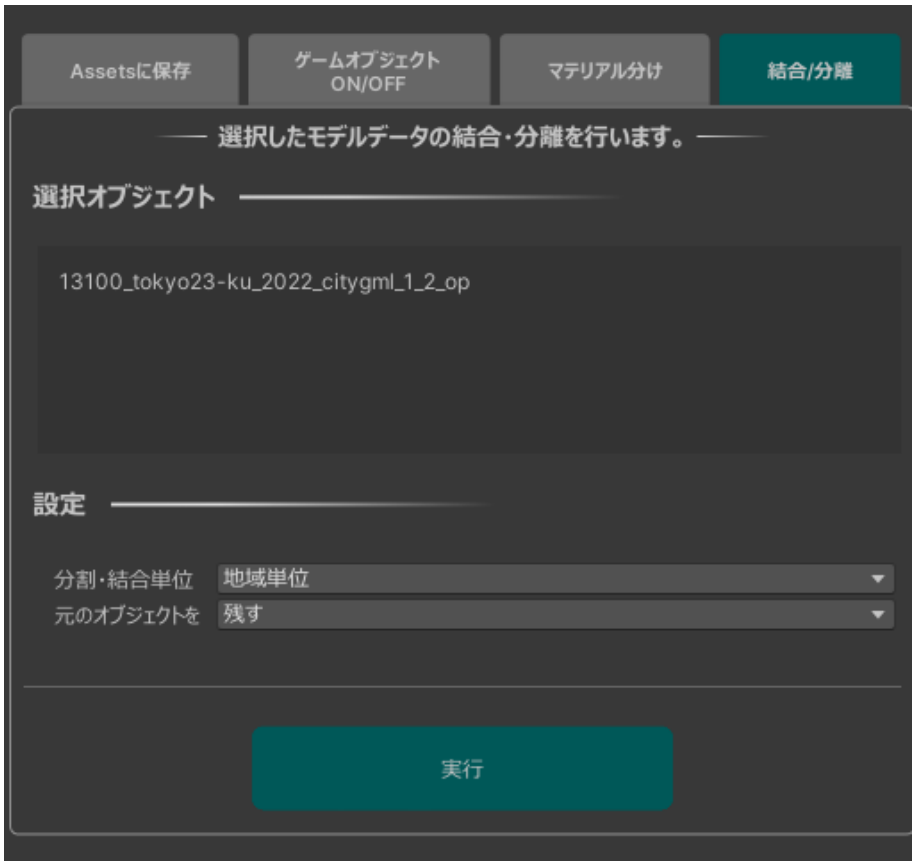


図1-2-29 分割結合画面

2.6.2 : [ゲームエンジン]対象をModelに変換
　共通ライブラリで変換を実行する準備として、エクスポートの項と同様に選択ゲームオブジェクトとその子をもとにModelを構築する。

2.6.3 : [ゲームエンジン] GML IDと属性情報の対応関係を記憶
　粒度の変更前後で地物の属性情報が維持されるようにするため、対象ゲームオブジェクトとその子を走査し、全てのgml:idとひも付いた属性情報を記憶する。

2.6.4： [共通ライブラリ] Modelを最小地物単位Modelに変換

対応する粒度は3種類なので、入力の粒度と出力の粒度を合わせた組み合わせは3×3で9通り存在する。変換のロジックを単純化するため、まず入力を最小地物に変換し次に最小地物を目的粒度に変換するという2段階に分けることで組み合わせ数を減らす。
入力Modelの最小地物への変換は以下の処理によって行われる。

1. 幅優先探索でNodeを走査する。
2. 各Nodeに含まれるMeshの地物インデックス（=UV4。詳しくは「2.4 属性情報取得機能」の項を参照されたい）を列挙する。
3. Node内のMeshをUV4が同じ値である面ごとに分割することで最小地物単位に変換する。

2.6.5： [共通ライブラリ] 目的粒度のModelに変換

この処理では最小地物単位でのModelを目的粒度に変換する。目的粒度が最小地物単位の場合は何も行わず、目的粒度が主要地物単位である場合は主要地物のNodeごとに結合、目的粒度が地域単位である場合は、全てのNodeが結合される。
結合処理によって地物インデックス（ポリゴンメッシュのUV4の値）が重複する可能性があるため、再度固有な値に割り当てられる。

2.6.6：[ゲームエンジン] Modelをゲームオブジェクトに変換して配置

これまでの処理で変換されたModelはゲームエンジン側に受け渡され、インポート処理と同様にゲームエンジン内のオブジェクトに変換して配置される。

2.6.7： [ゲームエンジン] 属性情報の復元

上記で記憶したgml:idと属性情報の対応関係をもとに、変換後オブジェクトに対して属性情報を復元する。

2.7 マテリアル分け機能

　マテリアル分け機能は、シーンに配置されたオブジェクトに対して、地物型に応じてマテリアルを変更する機能である。
　下図は屋上を緑色のマテリアルに変更した例である。

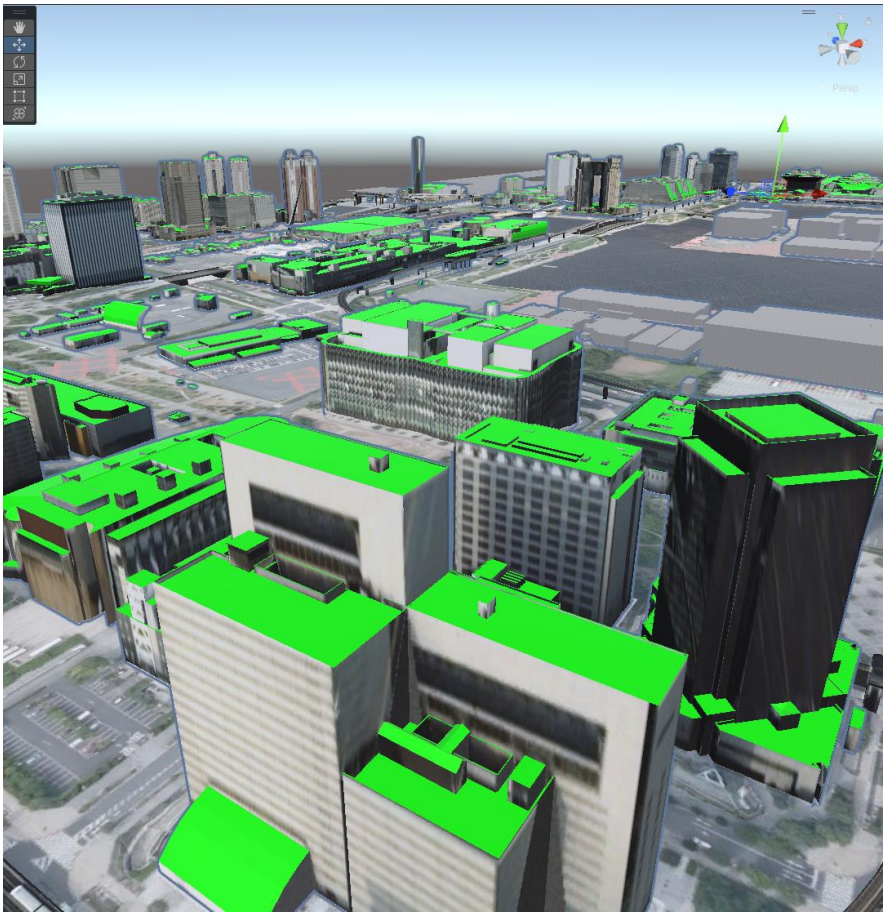


図1-2-30　マテリアル分けの例

マテリアル分け機能の処理の流れは以下のとおりである。

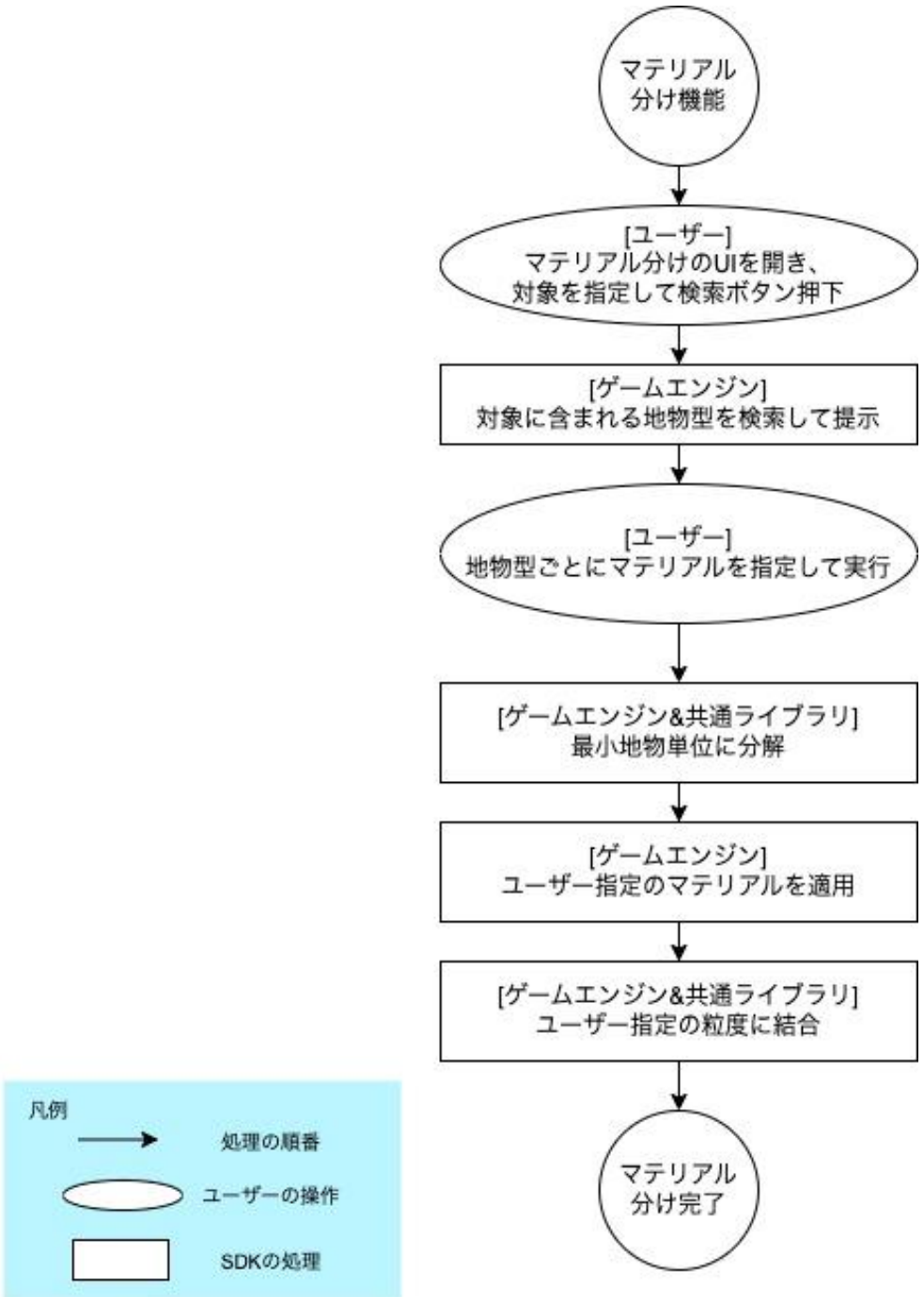


図1-2-31 マテリアル分け処理の流れ

2.7.1： [ゲームエンジン] 対象に含まれる地物型を検索して提示

ユーザーが処理の対象ゲームオブジェクトを指定して検索ボタンを押下すると、ゲームエンジンは対象とその子に含まれる地物型を検索して提示する。
提示された地物型ごとに、ユーザーはマテリアルを指定、またはマテリアルを変更しないことを選択できる。
ユーザーが実行ボタン押下すると次の処理に進む。

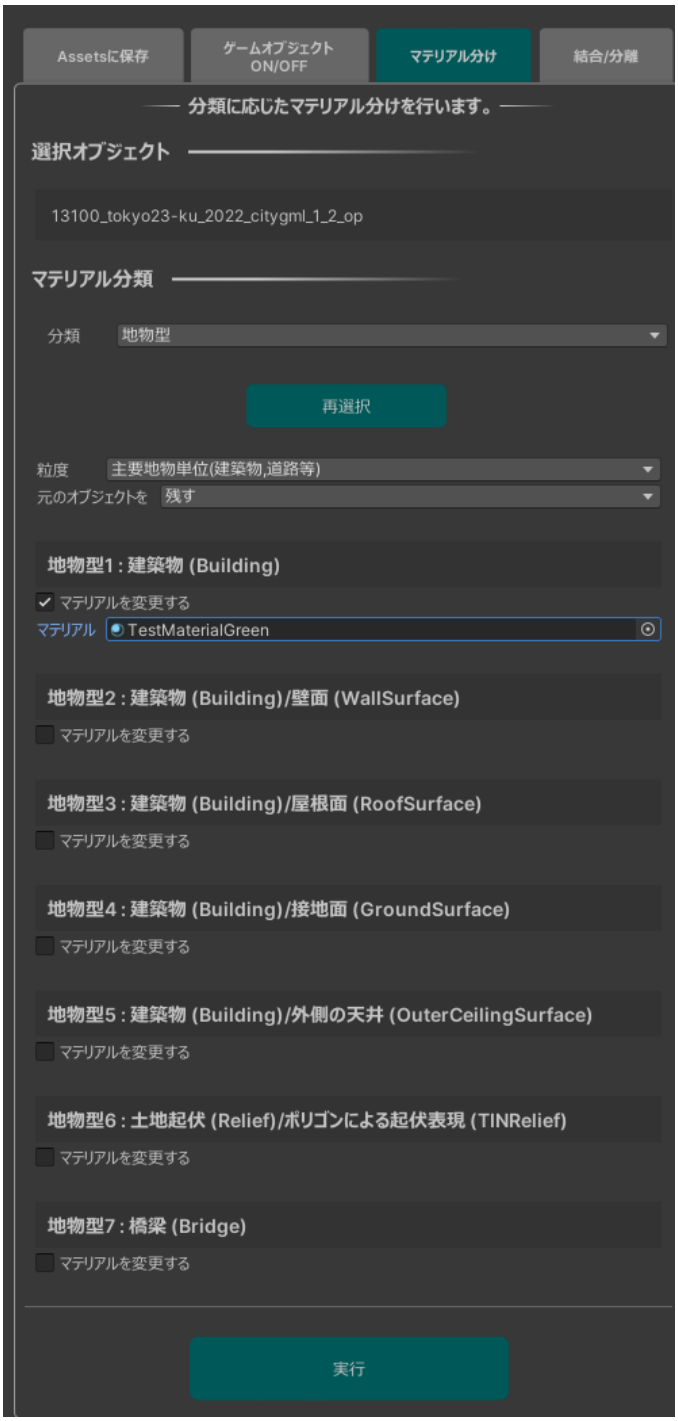


図1-2-32 マテリアル分け画面

2.7.2：最小地物単位に分解

処理の第一段階は、対象を最小地物に分解することである。この処理の内容は「2.6 分割・結合機能」の項で述べた。

こうすることで屋根面、壁面など、最小地物のみに表れる地物型を個々のゲームオブジェクトに分離し、マテリアルを適用できるようにする。

2.7.3：[ゲームエンジン] ユーザー指定のマテリアルを適用

ゲームエンジンは、最小地物として配置されたゲームオブジェクトに対し、地物別のユーザー指定のマテリアルを適用する。

2.7.4：ユーザー指定の粒度に結合

最小地物単位のゲームオブジェクト階層をユーザー指定の粒度に変更する。これも分割・結合の処理であるが、次の違いがある。

ゲームエンジンのマテリアルを維持するため、SubMeshにマテリアルIDを付与する。ゲームエンジンではマテリアルIDに対応するマテリアルの辞書を構築し、共通ライブラリではSubMeshのマテリアルIDを維持したまま分割結合処理を行う。変換後、ゲームエンジンは辞書をもとにマテリアルIDからマテリアルを復元する。

第2編　PLATEAU SDK Toolkits for Unity

第1章　Toolkitsの構成

1.1 PLATEAU SDK Toolkits for Unityの概要

　PLATEAU SDK Toolkits for Unityは、PLATEAUが提供する3D都市モデルのデータを利用したUnity上でのアプリケーション開発を支援するためのツールキット群である。

このツールキットを使用することで、Unity環境上で3D都市モデルを活用したシミュレーション環境の構築や、GIS/BIMデータの連携、拡張現実（AR）のアプリケーション構築などを行うことができる。利用には対応バージョンのUnity Editor、PLATEAU SDKの導入が必要である。本書作成時点での対応バージョンは以下の通り。

- GitHub：https://github.com/Project-PLATEAU/PLATEAU-SDK-Toolkits-for-Unity
- Unity対応バージョン：Unity 2021.3.35f1
- PLATEAU SDK対応バージョン：2.3.2

PLATEAU SDK Toolkits for Unity は「Rendering Toolkit」「AR Extensions」「Maps Toolkit」「Sandbox Toolkit」「PLATEAU Utilities」の5つのコンポーネントから構成される。 また、開発時の参考として、実際にToolkitsを活用して構築した4種類のサンプルシーンが提供されている。

PLATEAU SDK Toolkits for UnityはPLATEAU SDK for Unityを前提としており、利用の際にはPLATEAU SDK for Unityの導入が必要である。PLATEAU SDK for Unityについては第一編を参照されたい。

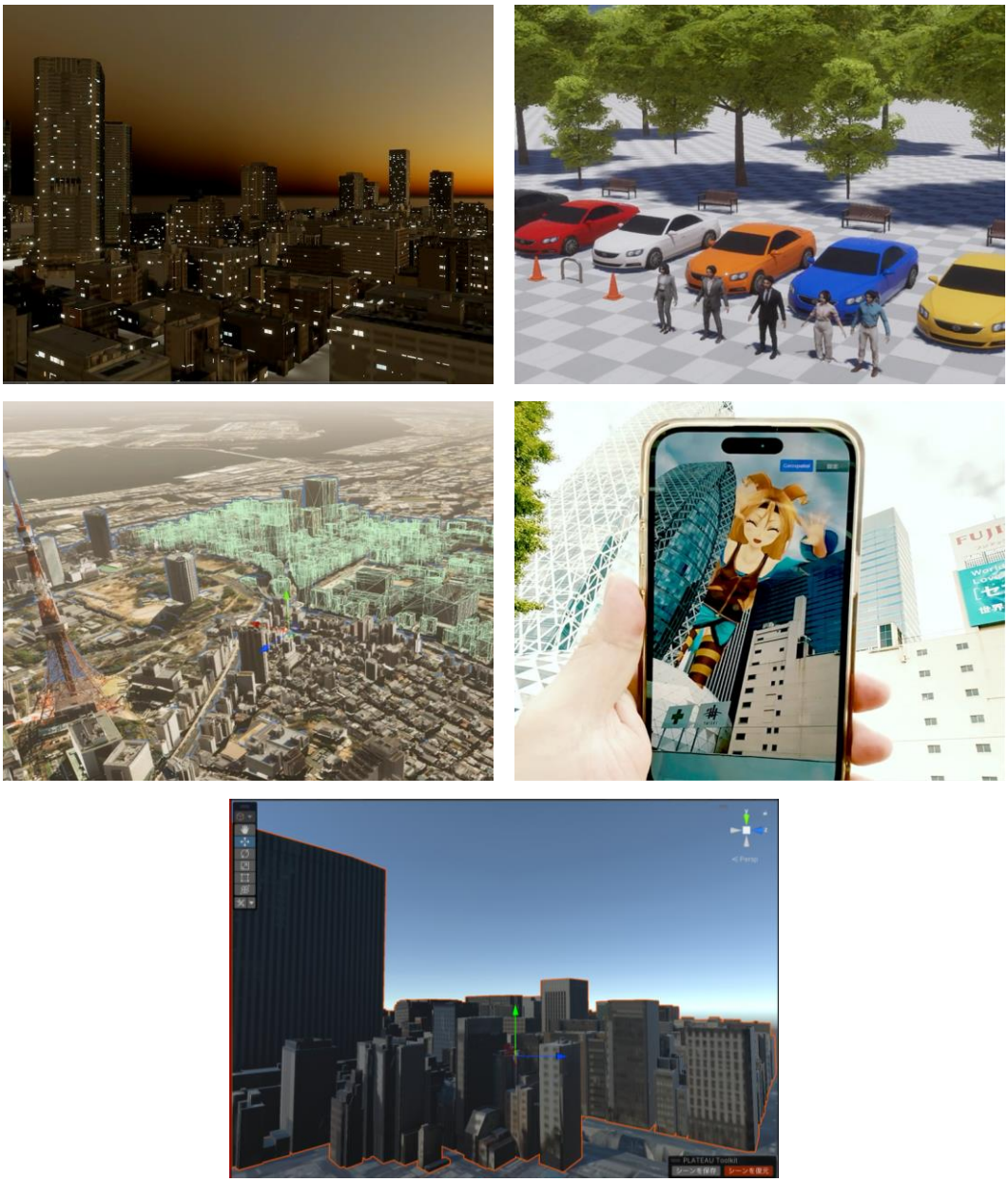


図2-1-1　PLATEAU SDK Toolkits for Unityを構成する5つのコンポーネント
(左上からRendering, Sandbox, Maps, AR Extensions, PLATEAU Utilities)

1.2 アーキテクチャ

PLATEAU SDK Toolkits for Unityを構成するシステム及び想定ユーザーを解説する。PLATEAU SDK Toolkits for Unityは複数のシステムで構成されており、以下に全体図を示す。

Rendering Toolkitは3D都市モデルのグラフィックスを向上させる処理を提供するToolkitである。Unityの機能を組み合わせて構築されているため、外部ライブラリ等を参照していない。

Sandbox Toolkitは3D都市モデルを用いたアプリケーション開発などの際に併用可能な3Dオブジェクトの配置や配置したオブジェクトの動きを制御する機能を提供している。こちらもRendering Toolkit同様、Unityの機能を活用しているため外部ライブラリ等を参照していない。

Maps Toolkitは3D都市モデルを利用したGIS開発向けツールキットである。3D Tiles形式で公開されているGISデータを活用するため、Cesium for Unityを使用している。また、BIMモデルの読み込みをサポートするため、IfcOpenShellを利用している。

AR Extensionsは3D都市モデルを利用したリアルロケーションAR体験の構築を行うための拡張機能である。AR機能の実装のため、AR Foundationを参照している。AR Foundationはマルチプラットフォームで AR機能を開発するためにUnity上で提供されているライブラリであり、Google社が提供するAR Core、Apple社が提供するAR Kitを参照している。

PLATEAU UtilitiesはUnityエディター上で3D都市モデルを編集操作用の機能を提供する。Unityの機能を活用しているため外部ライブラリ等を参照していない。

これらのToolkitsのコンポーネントはGithub上でユーザーに提供されている。UnityのPackage Managerでインポート可能なTarball形式に変換してアップロードされており、ユーザーは直接ファイルをダウンロードした後、Unityエディターからインポートを行い利用する。

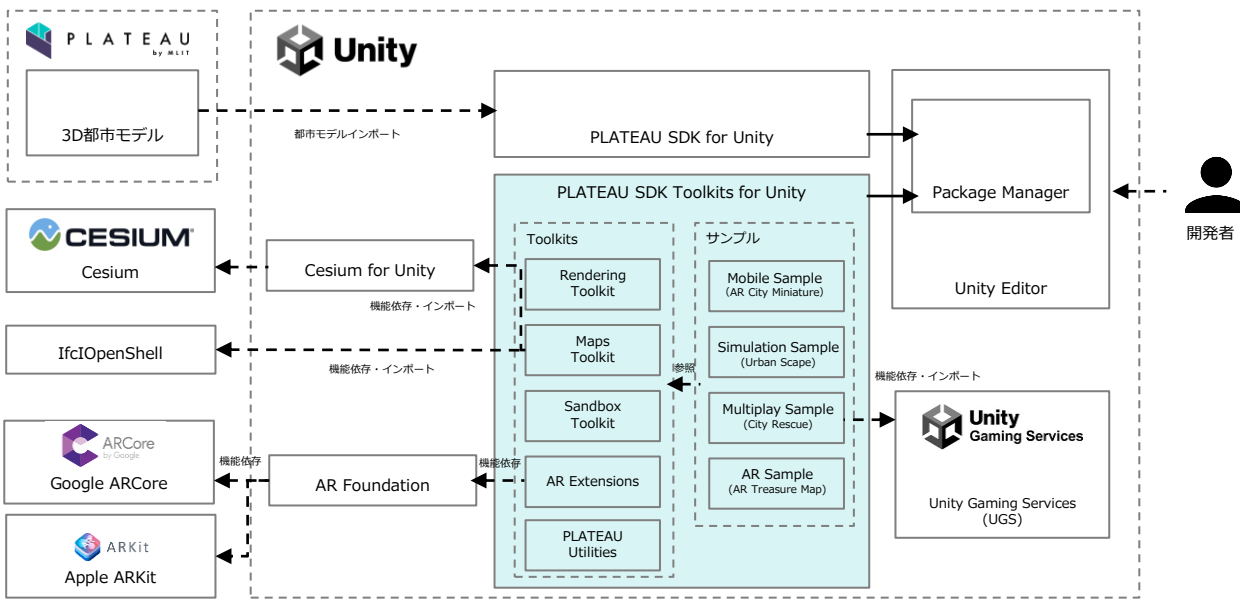


図2-1-2　PLATEAU SDK Toolkits for Unityのアーキテクチャ

Rendering Toolkitのアーキテクチャは以下の通り。Unity Editor上でのレンダリング設定を操作するツールキットであるため、各コンポーネントを通じてUnityの既存コンポーネントを操作している。

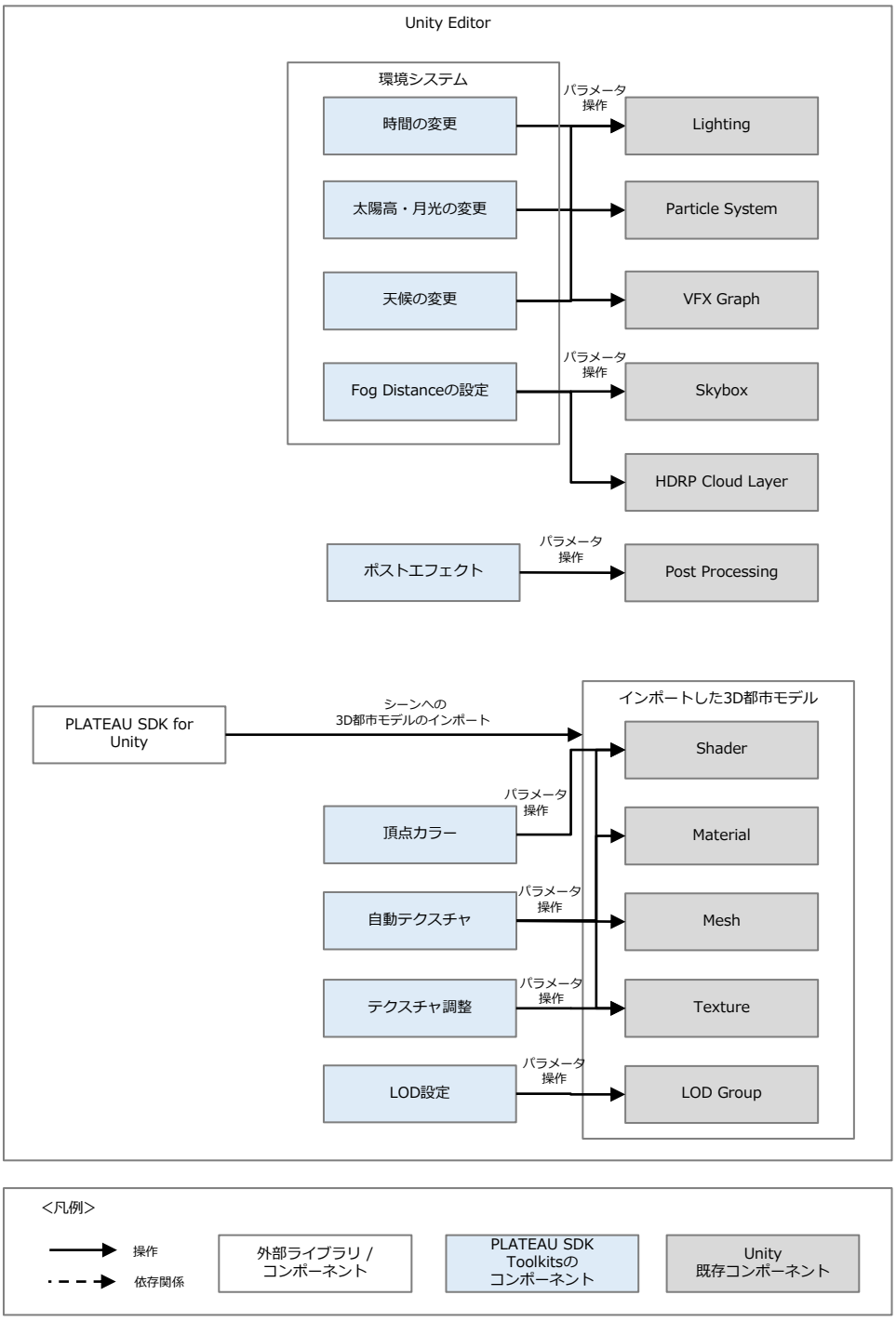


図2-1-3　Rendering Toolkitのアーキテクチャ

Sandbox Toolkitのアーキテクチャは以下の通り。オブジェクト配置を行うためのトラック作成やトラック移動コンポーネントはUnity標準のSplineパッケージに依存している。オブジェクト配置を行う際には必要なSandbox Assetをプロジェクトにダウンロードして配置する。

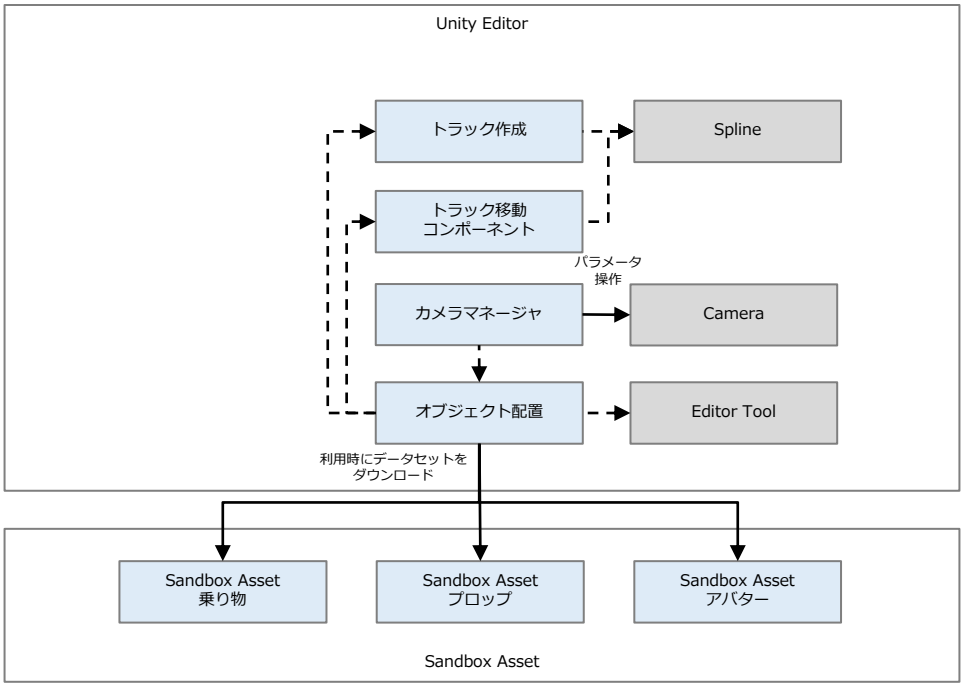


図2-1-4　Sandbox Toolkitのアーキテクチャ

AR Extensionsのアーキテクチャは以下の通り。

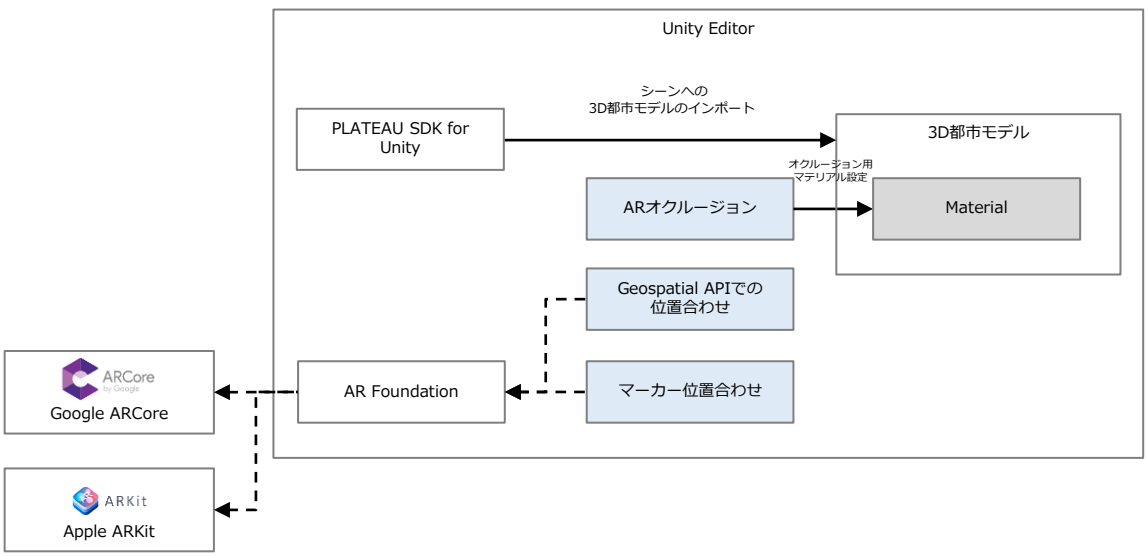


図2-1-5　AR Extensionsのアーキテクチャ

PLATEAU Utilitiesのアーキテクチャは以下の通り。

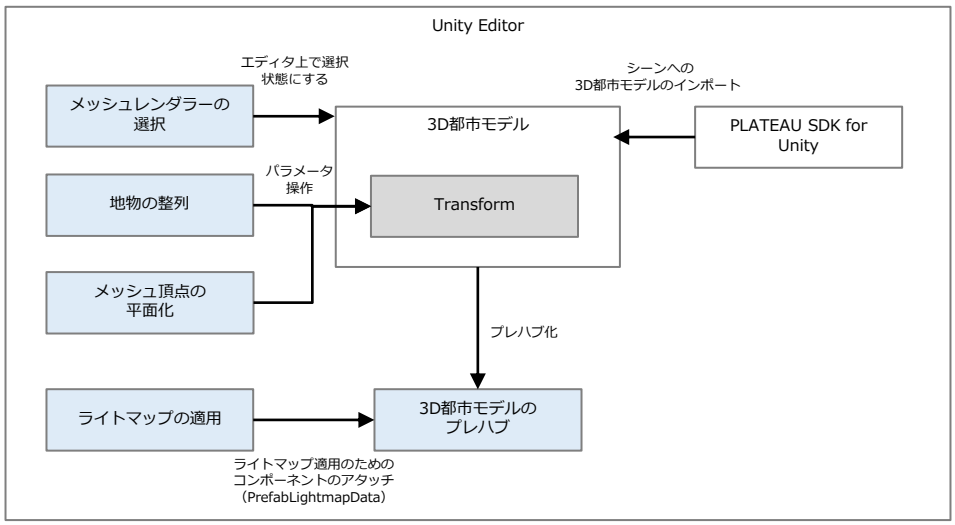


図2-1-6　PLATEAU Utilitiesのアーキテクチャ

4 種類のサンプルプロジェクトはそれぞれ以下のToolkitを参照している。

Mobile（AR City Miniature）サンプルはPLATEAUの3D都市モデルをジオラマのように卓上でAR表示する、モバイル端末向けのサンプルアプリケーションである。AR表示を行うため、AR Extensionsを参照している。

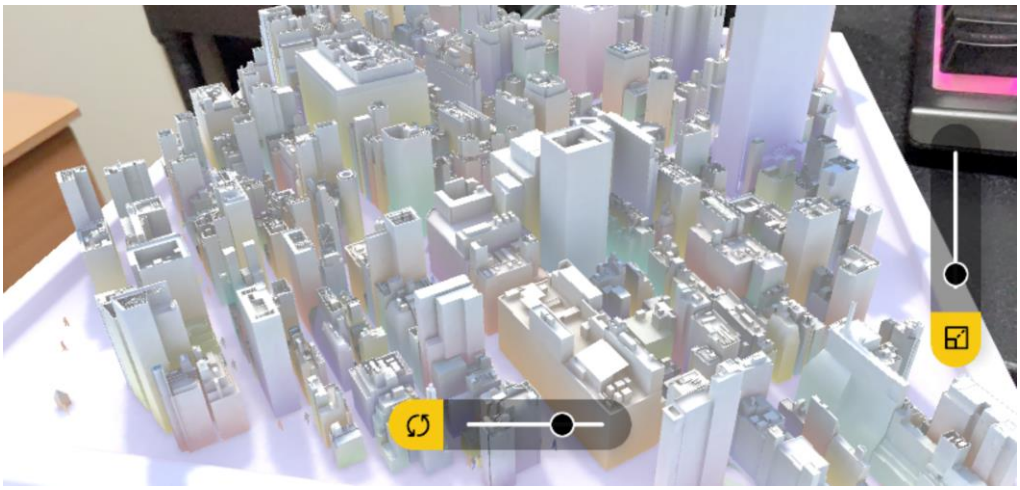


図2-1-7　AR City Miniatureの動作イメージ

Simulation（Urban Scape）サンプルは映像撮影、高画質でのシミュレーションなどを行うためのサンプルアプリケーションである。映像の高画質化のためRendering Toolkitを使用している。また、都市空間のリアリティを高めるため、Sandbox Toolkitを利用して車などのモデルを配置している。

図2-1-8　Urban Scapeの動作イメージ

Multi Play（City Rescue）サンプルは複数人で同時にプレイ可能なアプリケーションを構築するサンプルである。3D都市モデルと洪水情報のデータを組み合わせることで、災害シナリオを複数人で閲覧することができる。Rendering ToolkitとMaps Toolkitが活用されている他、マルチプレイ機能の実装例を示すため、実装例としてUnityが提供しているUnity Game Services （UGS） を使用している。

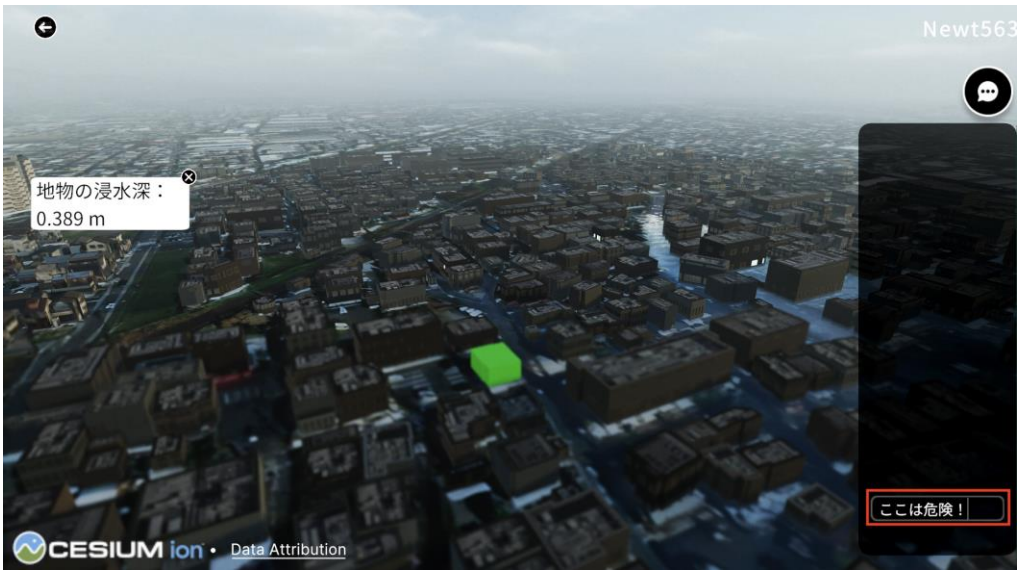


図2-1-9　City Rescueの動作イメージ

AR （Treasure Map）サンプルは現実の都市空間内で宝探し体験のような周遊型のARアプリケーションを構築するためのサンプルである。AR Extensionsを使用してアプリケーションを構築している。

図2-1-10　AR Treasure Mapの動作イメージ

表2-1-1　サンプルプロジェクトが参照しているToolkitsのコンポーネント

サンプル名	説明				
	Rendering	Maps	Sandbox	AR	Utilities
Mobile Sample （AR City Miniature）				○	○
AR Sample （AR Treasure Map）				○	
Simulation Sample （Urban Scape）	○		○		
Multi Play Sample （City Rescue）	○	○			

1.3 用語集

PLATEAU SDK Toolkits for Unityにおいて独自に使用される用語をまとめる。

表2-1-2　PLATEAU SDK Toolkits for Unity 用語集

用語	説明
環境システム	天候や時間帯を設定するとライティングやVFX Graph, Particle Systemと連携し、Unityのシーン上で環境シミュレーションを行うことができるRendering Toolkitの機能
Sandboxアセット	Sandbox Toolkitで提供している、シーン上に配置するためのアセット群。アバター、乗り物、プロップの三種類が存在する。
アバター	歩行者の3Dモデル。Sandboxアセットの一種で、Sandbox Toolkitの配置機能を用いてシーン上に配置できる。
乗り物	車両の3Dモデル。Sandboxアセットの一種で、Sandbox Toolkitの配置機能を用いてシーン上に配置できる。
プロップ	植樹や標識、看板など、都市空間内に一般的にある物の3Dモデル。Sandboxアセットの一種で、Sandbox Toolkitの配置機能を用いてシーン上に配置できる。
トラック	Sandbox Toolkitを用いて配置可能な、オブジェクトの移動経路。アバターや乗り物などのオブジェクトをトラック上に配置すると自動で走行させることができる。
位置合わせ（GIS）	PLATEAU SDK for Unityを用いてインポートした3D都市モデルデータと、Maps Toolkitsを用いてインポートした地形モデル、ラスタデータの位置を合わせる処理。
位置合わせ（AR）	端末のカメラ映像等を用いて自己位置認識を行い、周囲に表示するARオブジェクトの位置を合わせる処理。
PLATEAU Terrain	Project PLATEAUの一環で整備され、配信されている日本全国の3次元地形データ。以下のGithubリポジトリで公開されている。https://github.com/Project-PLATEAU/plateau-streaming-tutorial/blob/main/terrain/plateau-terrain-streaming.md

第2編　PLATEAU SDK Toolkit for Unity

第２章　Toolkitsの機能詳細

2.1 PLATEAU SDK Toolkits for Unityの機能一覧

PLATEAU SDK Toolkitsの各コンポーネントの実装機能の一覧は以下のとおりである。

表2-2-1　PLATEAU SDK Toolkits for Unity の機能一覧

Toolkit	小分類	ID	機能名	機能説明
Rendering	環境システムの設定	FN001	時間の変更	・シーンの時間を変更し、太陽・月の位置に反映させる
Rendering	環境システムの設定	FN002	天候の変更	・シーンの天候を晴れ・曇り・雨・雪に設定する
Rendering	環境システムの設定	FN003	太陽光・月光の調整	・シーンの太陽光・月光の色を任意の色に設定する
Rendering	環境システムの設定	FN004	Fog Distanceの設定	・Fog（霧）の深さを設定する
Rendering	環境システムの設定	FN005	Material Fadeの設定	・自動テクスチャで作成したテクスチャのマテリアルを単色化する
Rendering	自動テクスチャの生成	FN006	自動テクスチャの生成	・建築物モデルに対してランダムで外観のテクスチャを生成する
Rendering	LODグループの生成	FN007	LODグループの生成	・カメラ位置に応じて表示モデルを最適化できるよう、3D都市モデルに対してUnityのLOD設定を適用する
Rendering	シーンの保存	FN008	シーンの保存/復元	・シーンに対する編集を保存・復元する
Rendering	ポストエフェクト	FN009	トイカメラ	・画面全体にトイカメラ調の（中央のみ鮮明で画面の上下端に強いぼかしがかかる）視覚効果を追加する
Rendering	ポストエフェクト	FN010	ハーフトーン	・画面全体にハーフトーンの（ドット絵のような）視覚効果を追加する
Rendering	ポストエフェクト	FN011	ナイトビジョン	・画面全体に暗視スコープで覗いたような視覚効果を追加する
Rendering	頂点カラーの設定	FN012	頂点カラーG設定	・自動テクスチャ生成時のマスク範囲を設定する
Rendering	頂点カラーの設定	FN013	頂点アルファ設定	・頂点アルファ値をランダムに設定するためのシード値を入力して指定する
Rendering	テクスチャ調整機能	FN014	画素調整機能	・選択した地物に対して、ハイパスフィルター、コントラスト、ブライトネス、シャープネスの調整を行う
Rendering	テクスチャ調整機能	FN015	テクスチャ解像度変更機能	・データ圧縮のためテクスチャデータの解像度を下げる
Sandbox	トラックの作成	FN016	新しいトラックの作成	・オブジェクト配置用のトラックを作成する
Sandbox	トラックの作成	FN017	トラック上の移動制御	・トラック上でのオブジェクトの動作を制御する
Sandbox	トラックの作成	FN018	トラックに沿ってオブジェクトを生成	・トラックに沿ってオブジェクトを生成させる
Sandbox	オブジェクト配置	FN019	共通配置ツール	・アバター、乗り物、プロップの配置操作を行う
Sandbox	オブジェクト配置	FN020	アバターの配置	・アバターモデルを選択して配置する
Sandbox	オブジェクト配置	FN021	乗り物の配置	・乗り物モデルを選択して配置する
Sandbox	オブジェクト配置	FN022	プロップの配置	・プロップモデルを選択して配置する
Sandbox	カメラインタラクション機能	FN023	カメラインタラクション機能	・Sandboxツールキットを用いて配置したオブジェクトに合わせてカメラを配置する

（次ページに続く）

Toolkit	小分類	ID	機能名	機能説明
Maps	Cesium for Unityとの連携	FN024	3D都市モデルの位置合わせ機能	・PLATEAU SDKを用いてインポートした3D都市モデルと3D Tilesデータの位置合わせを行う
Maps	Cesium for Unityとの連携	FN025	3D都市モデルのストリーミング設定	・PLATEAU配信サービスを用いて3D都市モデルをストリーミング表示する
Maps	IFCモデルの読込	FN026	IFCファイルの読込	・IFCファイルをインポートしシーン上に配置する
Maps	GISデータ読込	FN027	GeoJSONファイルの読込	・GeoJSONファイルをインポートする
Maps	GISデータ読込	FN028	シェープファイルの読込	・シェープファイルをインポートする
AR	ARオクルージョン	FN029	ARオクルージョンの設定	・ARオクルージョン用に3D都市モデルに専用のシェーダーをアタッチして表示する
AR	AR空間内の位置合わせ機能	FN030	Geospatial APIを用いた位置合わせ機能	・Googleが提供するGeospatial APIを活用し、ユーザーの自己位置認識処理を行う
AR	AR空間内の位置合わせ機能	FN031	ARマーカーを用いた高さ補正機能	・AR機能利用時に、ARマーカーを利用することで位置合わせの高さを補正する
AR	AR空間内の位置合わせ機能	FN032	ARマーカーを用いた位置合わせ機能	・自己位置認識にARマーカーを使用し位置合わせを行う
AR	AR空間内の位置合わせ機能	FN033	手動位置合わせ機能	・AR表示位置を手動で調整する
Utilities	メッシュレンダラーの選択	FN034	メッシュレンダラーの選択	・種別を示す接頭辞を指定し、ヒエラルキー上の3D都市モデルを一括選択する
Utilities	選択地物の整列	FN035	選択地物の整列	・Unityのヒエラルキー上で選択した地物を同一の高さで整列する
Utilities	メッシュ頂点の平面化	FN036	メッシュ頂点の平面化	・Unityのヒエラルキー上でメッシュを任意の高さで平面化する
Utilities	プレハブへのライトマップの適用	FN037	プレハブへのライトマップの適用	・プレハブ化した3D都市モデルにライトマップを適用する

2.2 各機能の処理詳細

各ツールキットの機能詳細を以下に示す。なお、各機能に関連するUIUX設計は「2.3 ユーザーインターフェース」に記載している。また、各機能におけるデータ処理などの処理詳細や具体的な実装内容は「2.4 アルゴリズム」のセクションに記載しているので、併せて参照されたい。

2.2.1 Rendering Toolkitの機能詳細

【FN001-FN005】環境システム

- **機能概要：**シーン上の時間帯や日照条件、天候のパラメーターを操作することで、環境のシミュレーションを行う。
- **データ仕様：**
  - 入力：UI上での値の指定
  - 出力：なし
- **環境システム内の各機能の詳細：**
  - **【FN001】時間の変更**
    - **機能詳細：**ユーザーがTimeOfDayスライダを操作してシミュレーション内の時間を調整すると、設定値に基づいて太陽と月の位置を変更し、シーン内のライティングの設定に反映する。
    - **アルゴリズム：**【AL001】時間の変更
  - **【FN002】天候の変更**
    - **機能詳細：**ユーザーがRain / Snow / Cloudyスライダーを操作すると、設定値に基づいて天候が曇りや雨、雪に変化する。
    - **利用するライブラリ：**VFX Graph, Particle System
    - **アルゴリズム：**【AL002】雨、雪の表現、【AL003】雲の濃度設定
  - **【FN003】太陽光・月光の調整**
    - **機能詳細：**
      - ユーザーがSun Color / Moon Colorに指定した色を太陽または月に反映させる。
      - Sun Intensity / Moon Intensityの値に応じて太陽光・月光の強さを調節する。
      - FN001で計算した太陽・月の方角と組み合わせて太陽/月をレンダリングする。
    - **アルゴリズム：**【AL001】時間の変更
  - **【FN004】Fog Distanceの設定**
    - **機能詳細：**Fog Distance とFog Color パラメーターを介して、シーン全体を覆うフォグの距離と色味を調整し、霧の効果をシミュレートする。シーン全体に視覚的な奥行きの効果を与え、大気のリアリズムを表現する。
    - **アルゴリズム：**【AL004】フォグ設定
  - **【FN005】Material Fadeの設定**
    - **機能詳細：**Material Fade とBuilding Color パラメーターを用いて、3D都市モデルの地物のテクスチャ色を任意の色とブレンディングする。
    - **アルゴリズム：**【AL005】マテリアルフェード

【FN006】自動テクスチャ生成

- **機能概要：**
  - テクスチャが貼られていない3D都市モデルに対して、その形状や配置パターンから、自動的にテクスチャを生成して貼り付けを行う。
  - 生成されたテクスチャは【FN001】時間の変更機能と連動しており、夜間には窓や底面が発光する他、高さ60m以上の高層ビルは航空障害灯が点灯する。
- **データ仕様：**
  - 入力：なし
  - 出力：なし
- **機能詳細：**
  - ヒエラルキー上で複数の建物がまとめられているメッシュを建物単位に分割する。
  - 【AL006】テクスチャ生成のアルゴリズムを利用して建物の上面に事前に準備したテクスチャを適用する。また、建物の側面にはプロシージャルに窓を描画する。描画された窓は環境システムで時間帯を夜に設定すると発光する。加えて、建物底面に発光マテリアルを適用する。
  - 高さが60m以上の建物に対しては航空障害灯を設定する。
- **アルゴリズム：** 【AL006】テクスチャの生成

【FN007】LODグループの生成

- **機能概要：**
  - UnityのLOD設定をインポートした3D都市モデルに適用することで、カメラからの距離に応じて表示する建物の詳細度を変化させ、処理を最適化できるようにする。
- **データ仕様：**
  - 入力：なし
  - 出力：なし
- **機能詳細：**
  - ヒエラルキー上での建物モデルの親子構造を編集用に再構成する。
  - SDKのインポート時にデフォルトで最大LODのもののみが有効になっているため、全ての建物を有効化する。
  - 【AL007】LODグループの生成アルゴリズムで、UnityのLOD設定にコンポーネントを適用する。
- **アルゴリズム：** 【AL007】LODグループの生成

【FN008】シーンの保存/復元

- **機能概要：**
  - シーンに対する編集を保存・復元する。
- **データ仕様：**
  - 入力：なし
  - 出力：なし
- **機能詳細：**
  - UnityエディターのSave、Open Scene機能のラッパーのため、処理内容は標準のものと同様。
- **アルゴリズム：** なし

【FN009 – 011】ポストプロセスの適用

- **機能概要：**
  - カメラ画像に追加的な視覚効果を施すポストプロセスを適用する機能。
  - 具体的な視覚効果の内容は後述する。
- **データ仕様：**
  - 入力：なし
  - 出力：なし
- **機能詳細：**
  - スクリーンからレンダリング画像を取得：ゲームやアプリケーションの現在レンダリングされているフレームのスクリーン画像を取得する。
  - ポストプロセス効果の適用：画像処理のフィルターが実装されたシェーダーを介して取得した画像に任意のポストプロセスの効果を適用する。
  - ポストプロセスが適用された画像を一時的なレンダーターゲットに書き込み：ポストプロセス効果が適用された画像を一時的なレンダーターゲット（オフスクリーンバッファー）に保存する。これにより、元の画像データは変更されずに保持される。
  - オフスクリーンバッファーの画像をメインのレンダーターゲットにコピー：一時的なレンダーターゲットの画像を元のレンダーターゲット（スクリーン）にコピーし、最終的な画像をスクリーンに表示する。
- **各ポストプロセスで適用する視覚効果の詳細：**
  - **【FN009】トイカメラ**
    - **機能詳細：**トイカメラで撮影した写真のようなぼかし効果を適用する。一般的には「ティルト・シフト」とも呼ばれる。
    - **アルゴリズム：**【AL008】トイカメラ
  - **【FN010】ハーフトーン**
    - **機能詳細：** ハーフトーン効果を画像に適用する。ハーフトーン効果とは、画像や映像にドットパターンを適用し、輝度や色の違いをドットの大きさや密度で表現する視覚効果のことである。プリントメディアやレトロなビジュアルスタイルを模倣する際に使用される。
    - **アルゴリズム：**【AL009】ハーフトーン
  - **【FN011】ナイトビジョン**
    - **機能詳細：**画像にナイトビジョン（暗視）効果を適用する。ナイトビジョン効果は、低光量の環境下でも視認可能な画像を生成するために使用される。この効果は、特定の色調（通常は緑色）を使用して、低照度下での視認性を向上させる。
    - **アルゴリズム：**【AL010】ナイトビジョン

【FN012 – FN013】頂点カラー設定

- **機能概要：**
  - 頂点カラーの調整機能では、ユーザーがエディターのGUIを通じて地物のメッシュの頂点カラーの調整を行う機能を提供する。この機能は、特に建物の窓用頂点カラーマスク（Gチャンネル）の調整を行い、各地物にランダムな頂点アルファ値を割り当てる。
  - ユーザーが「頂点カラーの調整」ボタンを押下すると、選択された地物のメッシュに対して指定されたパラメーターに基づく頂点カラーの調整が行われる。
- **データ仕様：**
  - 入力：UI上での値の指定
  - 出力：なし
- **機能詳細：**
  - UIから設定値を取得：頂点カラーG設定用のマスク割合、頂点カラーA設定用のランダムシード値を取得する。
  - 【AL011】頂点カラー設定アルゴリズムを用いて頂点カラー調整処理を実施。
- **アルゴリズム：**【AL011】頂点カラー設定
- **頂点カラーの設定項目の詳細：**
  - **【FN012】頂点カラーG設定**
    - **設定項目：**この値は、地物の上部からどの範囲まで頂点カラーGチャンネルのマスクを適用するかを0から100%の範囲で指定する。このマスクは、建物の上部、天井等に窓が表示されないようにするために使用される。

  - **【FN013】頂点アルファ設定**
    - **設定項目：**この値は、頂点カラーAチャンネルの値のランダム化プロセスにおける基点となるランダムシード値を設定する。ランダムシード値により、地物ごとに一貫性はあるが異なるランダムアルファ値が設定される。

【FN014 – FN015】テクスチャ調整機能

- **機能概要：**
  - 選択した地物のテクスチャ画像の外観の調整を行う。
  - 画素調整機能では4種類の画像処理パラメーターを調整することで画像の外観を調整する。
  - 解像度変更機能ではテクスチャ解像度を下げることでテクスチャ画像サイズを圧縮し、処理負荷を軽減する。
- **データ仕様：**
  - 入力：UI上での値の指定
  - 出力：なし
- **テクスチャ調整機能の詳細：**
  - **【FN014】画素調整機能**
    - **機能詳細：**
      - プレビュー用メッシュの選択：シーンビュー/ヒエラルキー上でプレビュー用のメッシュを選択して「編集開始」を押下する。
      - 適用値の設定：編集適用後の外観をプレビューしながら、以下の4つのパラメーターを調整する。
        - ハイパスフィルター：ハイパスフィルターの設定値を調節する。ハイパスフィルターを使用することで、元のテクスチャから不要な影や光の影響を簡易的に取り除くことができる（ハイパスフィルターは、画像の周波数成分のうち高い周波数の部分を残し、低い周波数の部分を除去する）。
        - コントラスト調整：コントラストの設定値を調節する。コントラストを高く設定すると、明るい部分と暗い部分の差が大きくなる。
        - 輝度調整：画像全体の明るさを調節する。ブライトネスを高く設定すると画像全体が明るくなる。
        - シャープネス調整：画像の輪郭の鮮明さを調節する。シャープネスを高く設定すると輪郭がはっきりと表示される。
      - 適用値の保存：「編集を保存する」を押下すると4つのフィルターへの適用値が保存される。
      - メッシュへの適用：調整を適用したいメッシュをヒエラルキーから複数選択し、「選択したオブジェクトのテクスチャに保存済みの画素パラメーターをコピー」を押下するとテクスチャ調整が反映される。
    - **アルゴリズム：**【AL012】画素調整機能

  - **【FN015】テクスチャ解像度変更機能**
    - **機能詳細：**
      - プレビュー用メッシュの選択：シーンビュー/ヒエラルキー上でプレビュー用のメッシュを選択して「編集開始」を押下する。
      - 適用値の設定：編集適用後の外観をプレビューしながら、解像度のスケールを調整する。解像度変更のスケールは1, 1/2,1/4,1/8,1/16倍から選択できる。
      - 適用値の保存：「編集を保存する」を押下すると4つのフィルターへの適用値が保存される。
      - メッシュへの適用：調整を適用したいメッシュをヒエラルキーから複数選択し、「選択したオブジェクトのテクスチャに保存済みの画素パラメーターをコピー」を押下するとテクスチャ調整が反映される。
    - **アルゴリズム：**【AL013】解像度変更

2.2.2 Sandbox Toolkitの機能詳細

【FN016 - FN018】トラック機能

- **機能概要：**
  - 乗り物、アバターを配置可能な移動用の経路（トラック）を作成する。
  - トラックを作成するだけでなく、トラック上でのオブジェクトの移動や、 トラックに沿ったオブジェクトの生成をこの機能で制御している。
  - Unity標準のスプラインツールを拡張する形で実装している。
- **データ仕様**
  - 入力：UI上での値の指定
  - 出力：なし
- **利用するライブラリ：**スプライン （com.unity.splines）
- **トラック機能の個別機能の詳細：**
  - **【FN016】新しいトラックの作成**
    - **機能詳細：**
      - トラック編集ツールの起動：「新しいトラックを作成」を押下するとシーンビュー上でトラック編集ツールが起動する。
      - ポイントの配置：シーンビュー上でマウスカーソルを移動すると、トラックを形成するためのポイントのプレビューが表示される。クリックするとポイントを配置できる。ポイントを複数配置していくと、ポイントに沿って曲線のトラックが自動で生成される。
      - 配置の完了：始点と同じポイントをクリックするとトラックの作成が完了する。
    - **アルゴリズム：**なし（スプラインをラッピングして利用）

  - **【FN017】トラック上の移動制御**
    - **機能詳細：**
      - オブジェクトの配置：【FN019】共通配置ツールを用いて乗り物やアバターをトラックに沿って配置する。具体的な手順は後述する。
      - 移動制御コンポーネントのアタッチ：共通配置ツールでトラック上に配置されたオブジェクトに移動制御用のコンポーネントが自動でアタッチされる。
      - パラメーターの設定：インスペクタ上で以下のパラメーターを設定して反映する。
        - 制限速度：トラック上での制限速度をトラック全体またはオブジェクトごとに設定できる。
        - 衝突判定：周囲の配置オブジェクトとの衝突判定を行い、自動で速度調節を行う。
      - プレイモード: プレイモードを起動すると設定値に従ってオブジェクトがトラック上を移動する。分岐のあるトラックを作成した場合にはランダムに分岐して移動する。
    - **アルゴリズム：**【AL015】トラック移動コンポーネント、【AL016】ランダムウォーク

  - **【FN018】トラックに沿ってオブジェクトを生成**
    - **機能詳細：**
      - 【FN016】新しいトラックの作成機能で作成したトラックに沿って、任意のオブジェクトを自動生成する。
      - インスペクターのリストから生成するオブジェクトを設定し、生成する割合とランダムシード値を指定する。
      - 「生成」を押下するとランダムシード値に基づき生成位置がランダムに決定され、設定したオブジェクトが自動配置される。「ランダム生成」を押下するとランダムシード値自体がランダムに設定されてランダムにオブジェクトが生成される。
    - **アルゴリズム：**なし（スプライン標準機能のSplineInstantiateでオブジェクトを生成）

【FN019 – FN022】オブジェクト配置

- **機能概要：**
  - Sandbox Toolkitのアセット（アバター、乗り物、プロップ）をシーンに配置するツール。
  - 【FN019】の共通配置ツールはアセット種別に関わらず共通して利用できる配置機能。
  - Unityエディターの標準機能を拡張して構築されており、アセット配置時の向きの変更やトラック作成機能で作成したトラック上への配置、複数オブジェクトの一括配置などを行う。
  - シーンビューに表示されるオーバーレイUIから配置位置、配置方法、オブジェクトの向きのオプションを選択できる。
  - 【FN020】アバターの配置機能は【SC112】アバター配置タブで表示されているアセットを選択し、【FN019】の共通配置ツールを起動して配置可能な機能である。【FN021】乗り物の配置機能、【FN022】プロップの配置機能についても同様。
- **データ仕様**
  - 入力：UI上でのオブジェクトの選択、配置位置の指定
  - 出力：なし
- **利用するライブラリ：**EditorTool API
- **機能詳細：**
  - 配置モードの起動：配置メニューから「配置ツールを起動」を押下するとオブジェクト配置ツールが起動する。
  - オブジェクトの配置：シーンビュー上でマウス操作を行い、オブジェクトを配置する。配置の向きや配置方法を変更するときはオーバーレイUIのドロップダウンメニューを選択する。以下のように、配置方法により処理が異なる。
    - クリック配置：
      - シーンビュー上でマウスカーソルを移動すると選択したオブジェクトを配置したときのプレビューが表示される。
      - シーンビュー上でクリックするとオブジェクト配置位置が確定し、配置が完了する。
    - ブラシ配置
      - ブラシ配置の設定パラメーターに応じて、マウスカーソル位置に合わせてオブジェクト設置パターンがハイライト表示される。各パラメーターの設定内容は以下のとおり。

表2-2-3　ブラシ配置機能のパラメーター一覧

設定項目	概要
オブジェクトの回転	配置されるオブジェクトの向き(0～360度)を設定する。この角度は「オブジェクトの向き」を軸とした回転で定義される。
配置数	一回のブラシ配置で配置されるオブジェクトの数を設定する。
ブラシサイズ	ドラッグで連続配置を行う際の配置間隔を設定する。
シード値固定	配置ごとにシード値を振りなおすかどうかを設定する。固定されていない場合は、配置ごとに新しいシード値が設定されるため、ブラシの形状が自動的に変化する。
ブラシ乱数シード値	ブラシの形状に現在使用されている乱数が設定される。任意のシード値を設定することも可能である。

      - シーンビュー上でマウスをクリックまたはドラッグするとオブジェクトが配置される。なお、クリック配置と異なりブラシ配置ではクリック・ドラッグ時に設置面の判定を行っているため、Toolkitで設定されている設置範囲内に接地面がない場合はオブジェクトは配置されない。
- **アルゴリズム：**【AL017】オブジェクト配置

【FN023】カメラインタラクション機能

- **機能概要：**
  - Sandboxツールキットを用いて配置したオブジェクトに合わせてカメラを配置し、視点モードを切り替える。
- **データ仕様**
  - 入力：UI上での値の指定
  - 出力：なし
- **利用するライブラリ：**なし
- **機能詳細：**
  - カメラマネージャーの作成：Sandboxツールキットウィンドウから「カメラマネージャーを作成」を押下すると、ヒエラルキーにPlateauSandboxCameraManagerが作成される。
  - カメラマネージャーの起動：プレイモードを実行し、配置したアバターや乗り物をクリックするとカメラマネージャーが起動し視点が移動する。
  - 視点切り替え：キーボード操作を行うと、一人称、三人称、三人称中心視点の3つの視点モードが切り替わる。
- **利用するアルゴリズム：**【AL018】カメラマネージャー

2.2.3 Maps Toolkitの機能詳細

【FN024】3D都市モデルの位置合わせ機能

- **機能概要：**
  - Cesiumの地形データ（3D Tiles）とPLATEAUの3D都市モデルを簡単に重畳表示できるような位置合わせ機能。
  - 3D TilesのインポートにCesium for Unityを使用するため、当機能の利用にはCesium for Unityの導入が必須である。
- **データ仕様**
  - 入力：UI上での値の指定、Cesium for Unityで取得する地形モデル、ラスタデータ
  - 出力：なし
- **利用するライブラリ：**Cesium for Unity
- **機能詳細：**
  - 地形モデル作成：Cesium for Unityを用いて空の3D Tiles Tilesetを作成する。
  - 地形モデルにPLATEAU Terrainを設定：地形モデルにPLATEAU Terrainを利用できるよう、作成した3D Tiles Tileset（Cesium3DTileset）に対して、インスペクターからTilesetSourceとion Asset ID, Tokenを変更。正しく設定するとCesium for Unityが地形モデルをインポートしシーン上にPLATEAU Terrainが描画される。
  - 地形モデルのテクスチャを表示：Cesium for UnityのRaster Overlay機能を用いて地形モデルのテクスチャをインポートして表示する。
  - 3D都市モデルの配置：PLATEAU SDKを用いて3D都市モデルをインポートする。Cesium for Unity上でのグローバル座標を付与するため、3D都市モデルに対してCesium Globe Anchorコンポーネントをアタッチする。
  - 位置合わせの実行：「PLATEAUモデルの位置を合わせる」を押下すると位置合わせ処理が行われ、PLATEAUモデルと地形モデルの位置が合わせられる。
- **アルゴリズム：**【AL019】3D都市モデルの位置合わせ

【FN025】3D都市モデルのストリーミング機能

- **機能概要：**
  - Cesiumの地形データ（3D Tiles）とPLATEAUの3D都市モデルを簡単に重畳表示できるような位置合わせ機能。
  - 3D TilesのインポートにCesium for Unityを使用するため、当機能の利用にはCesium for Unityの導入が必須である。
- **データ仕様**
  - 入力：UI上での値の指定、PLATEAU配信サービスからの3D Tilesデータ
  - 出力：なし
- **利用するライブラリ：**Cesium for Unity、 PLATEAU配信サービス
- **機能詳細：**
  - 3D Tilesetオブジェクトの作成：Cesium for Unityを用いてCesium 3D Tilesetを作成。
  - データソースにURLを指定：Cesium 3D TilesetのTile Sourceをfrom URL（URL経由）に設定し、PLATEAU配信サービスのURLを指定する。PLATEAU配信サービスのURLは都市単位で指定されているので利用したいエリアのURLを確認してURL入力欄に入力する。
  - ストリーミングの実行：設定値に基づき、Cesium for Unityにより3D都市モデルがストリーミングされシーンビューに描画される。
- **アルゴリズム：**なし（Cesium for Unityの機能をラッピングして提供しているため）

【FN026】IFCファイルの読込

- **機能概要：**
  - IFC形式のBIMデータを読み込むことで、3D都市モデルや地形データと重畳して建築データをUnity内で表示させる機能。
  - IFCファイルの他、XML形式の属性ファイルを指定すると、インポートするIFCファイルに属性情報を付与可能。
  - IFCファイルのインポートにIfc Open Shellを利用しているため、この機能の利用にはIfc Open Shellの導入が必須。
- **データ仕様**
  - 入力：IFCファイル、XMLファイル、UI上での値の指定
  - 出力：なし
- **利用するライブラリ：**Ifc Open Shell（Ifc Convertモジュール）、国土地理院API
- **機能詳細：**
  - Ifc Open Shellの導入：Maps ToolkitのUIからIfc Convertライセンスへのリンク及びインストール確認ポップアップを表示。「承諾」を押下するとIfc Convertモジュールが導入される。
  - インポートするIFCファイルと属性ファイルの指定：ローカルディスク上のIFCファイルを指定する。属性ファイルをインポートしたい場合はあわせて指定する。
  - Ifc Open Shellを用いてIFCファイルを変換：指定したIFCファイルがGLB形式に変換される。
  - シーン上に配置：変換したIFCファイルは以下の2通りでシーン上に配置される。どちらの配置方法でも、国土地理院APIを使用してジオイド高を取得し、ジオイド高を加算している。
    - UI入力値から配置：UI上で緯度、経度、標高、回転角度、縮尺を入力する。入力値に基づき、Cesium for Unityでインポートした3D Tilesに合わせて建物モデルが配置される。
    - 属性情報から自動配置：属性情報から緯度、経度、標高、回転角度、縮尺を取得し、3D Tilesに合わせて建物モデルを配置する。
- **アルゴリズム：**なし（Ifc Open Shellを用いてIFCファイルをUnityで利用可能なGLB形式に変換している）

【FN027】GeoJSONファイルの読込

- **機能概要：**
  - GeoJSON形式のGISデータをインポートする。
  - この機能を使用することで、PLATEAUの3D都市モデルや地形データと、様々な空間データを重畳して表示できる。
- **データ仕様**
  - 入力：GeoJSONファイル、UI上での値の指定
  - 出力：なし
- **利用するライブラリ：**なし
- **機能詳細：**
  - インポートするファイルの指定：ローカルディスク上のGeoJSONファイルを指定する。
  - パラメーターの指定：配置用のパラメーターを指定する。
  - GeoJSONファイルをゲームオブジェクト化: 後述するゲームオブジェクト化のアルゴリズムでGeoJSONファイルを読み込み、ゲームオブジェクト化する。
- **アルゴリズム：**【AL020】GISデータのゲームオブジェクト化

【FN028】シェープファイルの読込

- **機能概要：**
  - SHP形式のGISデータをインポートする。
  - GeoJSONファイルの読込と同様に、この機能を使用することで、PLATEAUの3D都市モデルや地形データと、さまざまな空間データを重畳して表示できる。
- **データ仕様**
  - 入力：SHPファイル、UI上での値の指定
  - 出力：なし
- **利用するライブラリ：**なし
- **機能詳細：**
  - インポートするファイルの指定：ローカルディスク上のSHPファイルを指定する。
  - パラメーターの指定：配置用のパラメーターを指定する。レンダー方法（メッシュとして描画するか、線として描画するか）を併せて指定する。
  - SHPファイルをゲームオブジェクト化：後述するゲームオブジェクト化のアルゴリズムでGeoJSONファイルを読み込み、ゲームオブジェクト化する。
- **アルゴリズム：**【AL020】GISデータのゲームオブジェクト化

2.2.4 AR Extensionsの機能詳細

【FN029】ARオクルージョン機能

- **機能概要：**
  - ARアプリケーションにおける遮蔽表現（オクルージョン）のため、手前側にある透明なオブジェクトが背後のオブジェクトを遮蔽する。
- **データ仕様：**
  - 入力：UI上での値の指定
  - 出力：なし
- **利用するライブラリ:** なし
- **機能詳細：**
  - レイヤー設定：遮蔽用のオブジェクトのレイヤー（AR Occluder）、遮蔽される側のレイヤー（AR Occludee）を手動で設定する。
  - オクルージョン表現：専用のレンダラーフィーチャー（PlateauAROcclusionRendererFeature）を適用することで、AR OccludeeをAR Occluderで隠すというレンダリング動作を実現する。
- **アルゴリズム：**【AL021】ARオクルージョン

【FN030 – 033】AR空間内の位置合わせ機能

- **機能概要：**
  - ARマーカーやカメラ映像、GPS情報を用いてARやコンテンツの位置合わせを行う。
  - 処理の詳細は位置合わせの手法により異なる。以下それぞれの位置合わせ手法別に記載する。
  - **位置合わせの個別機能の詳細**

  【FN030】Geospatial APIを用いた位置合わせ機能
  - **データ仕様：**
    - 入力：Geospatial APIからの自己位置情報、カメラ画像
    - 出力：なし
  - **利用するライブラリ：**Google Geospatial API、AR Foundation
  - **機能詳細：**
    - Geospatial APIを利用するため、Geospatialコントローラーを初期化。
    - 後述するGeospatial API を用いた位置合わせアルゴリズムを用いて、緯度経度とジオイド高を指定してアンカーを作成。
  - **アルゴリズム：**【AL022】Geospatial APIを用いた位置合わせ

  【FN031】ARマーカーを用いた高さ補正機能
  - **データ仕様：**
    - 入力：カメラ画像
    - 出力：なし
  - **利用するライブラリ：**AR Foundation
  - **機能詳細：**
    - 高さ補正を行いたい場所でARマーカーを読み取る。
    - 後述するARマーカーを用いた高さ合わせアルゴリズムを用いて、高さの差分を取得。
    - 高さの差分からAR表示位置をオフセットし、表示を補正。
  - **アルゴリズム：**【AL023】ARマーカーを用いた高さ合わせ

【FN032】ARマーカーを用いた位置合わせ機能

- **データ仕様：**
  - 入力：カメラ画像
  - 出力：なし
- **利用するライブラリ：**Google Geospatial API、AR Foundation
- **機能詳細：**
  - 位置合わせを行いたい場所でARマーカーを読み取る。
  - 後述するARマーカーを用いた高さ合わせアルゴリズムを用いて、高さの差分を取得。
  - 高さの差分からAR表示位置をオフセットし、表示を補正。
- **アルゴリズム：**【AL024】ARマーカーを用いた位置合わせ

【FN033】手動位置合わせ機能

- **データ仕様：**
  - 入力：UIからの入力値
  - 出力：なし
- **利用するライブラリ：**AR Foundation
- **機能詳細：**
  - ビルドしたARアプリケーション上で手動位置合わせ用のUIを開く。
  - UIで位置調整用のオフセット値を入力。
  - 手動位置合わせアルゴリズムを用いて、表示を補正。
- **アルゴリズム：**【AL025】手動位置合わせ

2.2.5 PLATEAU Utilitiesの機能詳細

【FN034】メッシュレンダラーの選択

- **機能概要：**
  - PLATEAU SDKからインポートした3D都市モデルの中で、指定した種別の地物を一括選択する。
  - オートテクスチャリング機能を一括で適用する場合など、ビル群をまとめて選択したい場合に有効な機能。
- **データ仕様：**
  - 入力：UI上での値の指定
  - 出力：なし
- **利用するライブラリ：**なし
- **機能詳細：**
  - トップノードを選択：選択したい地物を含む3D都市モデルのトップノードをヒエラルキーから選択する。
  - 地物の接頭辞を指定：メッシュレンダラーの選択機能のUIで、選択したい地物を示す接頭辞を入力。
  - 指定したメッシュレンダラーを選択：「メッシュレンダラーの選択」を押下すると、トップノード配下の地物オブジェクトの中で指定した接頭辞を含むものが検索され、選択状態になる。
- **アルゴリズム：**なし（Unityエディター標準のヒエラルキー検索を拡張）

【FN035】選択地物の整列

- **機能概要：**
  - 地物の底面の高さを原点に整列させるための機能。
- **データ仕様：**
  - 入力：UI上での値の指定
  - 出力：なし
- **利用するライブラリ：**なし
- **機能詳細：**
  - 整列する地物を選択：整列を行いたい地物をシーンビューまたはヒエラルキーから複数選択。
  - 整列を行う高さを指定：選択地物の整列UIで、整列を行いたい高さの数値を入力。
  - 地物を整列：「選択地物の整列」を押下すると、選択した地物が指定の高さで整列される。
- **アルゴリズム：**なし

【FN036】メッシュ頂点の平面化

- **機能概要：**
  - 3D都市モデルの地面メッシュ（DEM）上のメッシュ頂点を平面化する機能。
- **データ仕様：**
  - 入力：UI上での値の指定
  - 出力：なし
- **利用するライブラリ：**なし
- **機能詳細：**
  - 平面化するメッシュを選択：平面化を行いたいメッシュをシーンビューまたはヒエラルキーから複数選択。
  - 平面化を行う高さを指定：メッシュ頂点の平面化UIで、整列を行いたい高さの数値を入力。
  - メッシュ頂点の平面化：「メッシュ頂点の平面化」を押下すると、選択したメッシュが平面化される。
- **アルゴリズム：**なし

【FN037】プレハブへのライトマップの適用

- **機能概要：**
  - プレハブに対してシーンのライトマップを保存する。
  - プレハブを呼び出すことが一般的なARアプリケーションなど向けの機能。特にモバイルアプリではリアルタイムライティングではなくライトマップの事前作成がパフォーマンス最適化のために有効だが、Unity標準ではライトマップをプレハブに適用できないため、この機能をUtilitiesに追加した。
  - 機能利用前に通常のワークフローでシーンのライトマップを作成する必要がある。
- **データ仕様：**
  - 入力：なし
  - 出力：なし
- **利用するライブラリ：**なし
- **機能詳細：**
  - プレハブの選択：シーンからライトマップを適用したいプレハブを選択する。
  - PrefabLightmapDataの適用：選択したプレハブにPrefabLightmapDataコンポーネントをアタッチする。
  - シーンのライトマップを適用：ライトマップ情報がプレハブに適用される。
- **アルゴリズム：**【AL026】プレハブへのライトマップの適用

2.3 ユーザーインタフェース

2.3.1 画面一覧

表2-2-4　PLATEAU SDK Toolkits for Unity のユーザーインタフェース一覧

ID	連携（ID）	画面名	画面説明
SC001	SC002, SC003, SC004, SC005, SC006	メインウィンドウ	各Toolkitのパネルを開くウィンドウ
SC002	SC001, SC007,SC008,SC009, SC010	Rendering Toolkitメインパネル	Rendering Toolkitの機能にアクセスするためのパネル
SC003	SC001, SC011, SC012, SC013, SC014	Sandbox Toolkitメインパネル	Sandbox Toolkitの機能にアクセスするためのパネル
SC004	SC001, SC016, SC017, SC018	Maps Toolkitメインパネル	Maps Toolkitの機能にアクセスするためのパネル
SC005	SC001	AR Extensionsメインパネル	AR Extensionsの機能にアクセスするためのパネル
SC006	SC001	PLATEAU Utilitiesメインパネル	PLATEAU Utilitiesの機能にアクセスするためのパネル
SC007	SC002	環境システムタブ	Rendering Toolkitの環境システムを利用するための画面
SC007-2	SC002	環境システム編集画面	環境システム起動後に表示される、パラメーターの編集用の画面
SC008	SC002	自動テクスチャ生成タブ	自動テクスチャ生成、頂点カラーのパラメーター設定用画面
SC009	SC002	LODグループタブ	LODグループを生成するための画面
SC010	SC002	テクスチャ調整タブ	画素調整機能、解像度変更機能を利用するための画面
SC011	SC003	トラック機能タブ	Sandbox Toolkitのトラック作成機能を利用するための画面
SC012	SC003, SC015	アバター配置タブ	アバターモデルを選択して配置するための画面
SC013	SC003, SC015	乗り物配置タブ	乗り物モデルを選択して配置するための画面
SC014	SC003, SC015	プロップ配置タブ	プロップモデルを選択して配置するための画面
SC015	SC012, SC013, SC014	配置ツールウィンドウ	オブジェクト配置時のオプションを制御する画面
SC016	SC004	PLATEAUモデル位置合わせタブ	3D都市モデルとCesium SDKで読み込んだ3DTilesを位置合わせするための画面
SC017	SC004	IFCモデル読込タブ	IFCモデル読込機能を利用するための画面
SC018	SC004	GISデータ読込タブ	GISデータ読込機能を利用するための画面
SC019		シーン保存UI	Toolkitを用いて行った編集を保存したり、過去に保存した内容を復元したりするためのUI

2.3.2 画面遷移

Toolkits各機能の画面遷移は以下のとおり。

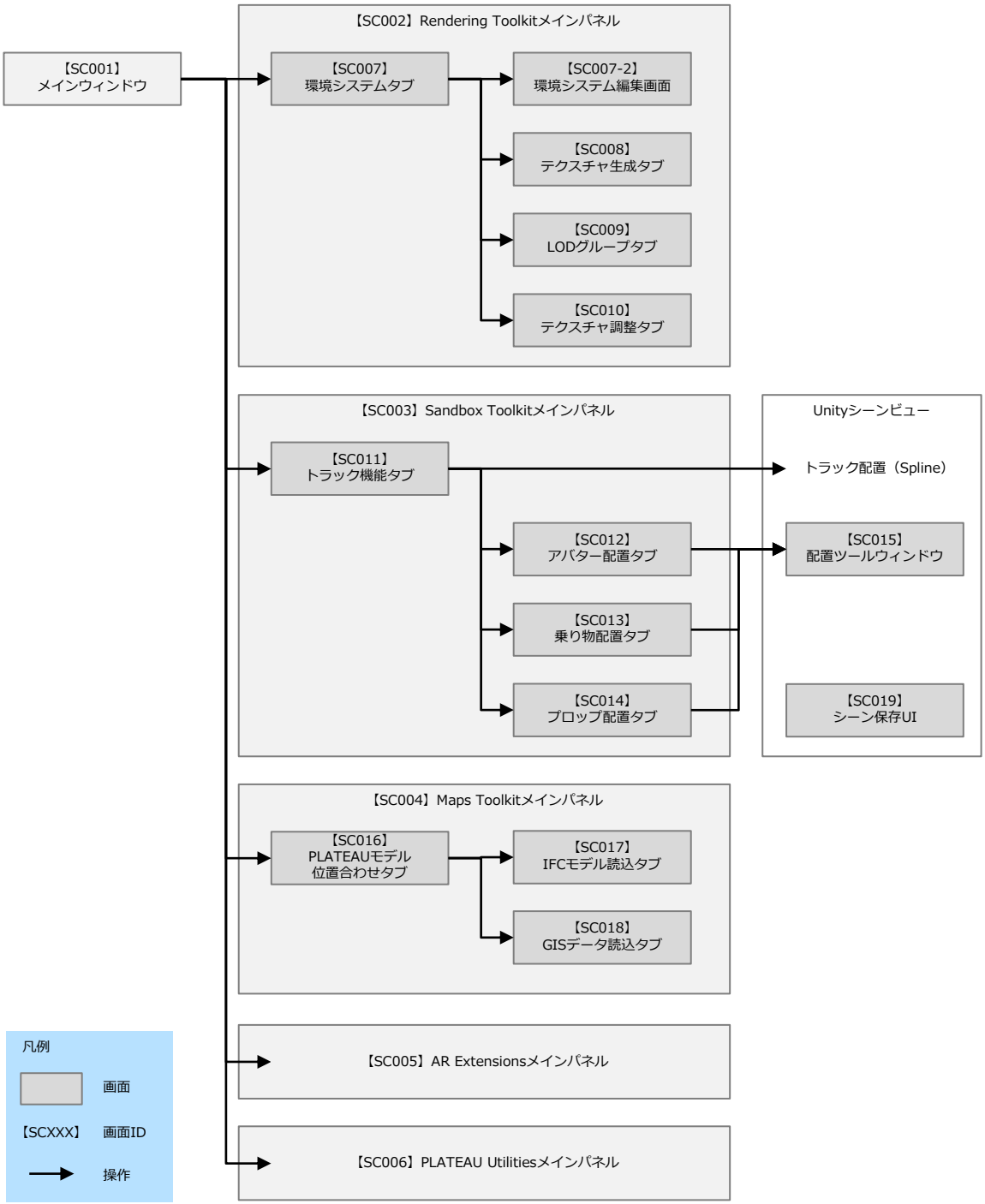


図2-2-1　PLATEAU SDK Toolkits for Unity のユーザーインタフェース一覧

2.3.3 各画面の詳細

各画面の具体的な仕様は以下のとおり。

【SC001】メインウィンドウ

- **画面の目的・概要：**
  - Toolkitsの各機能にアクセスするためのウィンドウ。
  - メニューバーの「PLATEAU」からアクセスすることができ、インストール済みのToolkitが「PLATEAU Toolkit」配下に表示される。
  - 開きたいToolkitを選択すると、そのToolkitのメインパネルが表示される。
- **画面イメージ：**

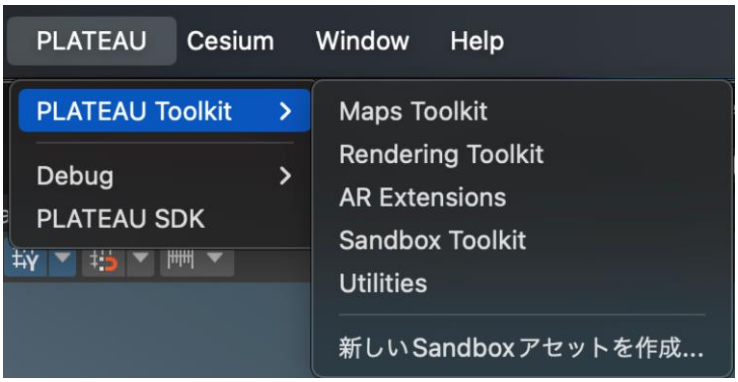


図2-2-2　メインウィンドウ

【SC002】Rendering Toolkitメインパネル

- **画面の目的・概要：**
  - Rendering Toolkitの各機能にアクセスするためのパネル。
  - PLATEAUロゴ直下の4つのアイコンがボタンになっており、左から【SC007】環境システムタブ、【SC008】テクスチャ生成タブ、【SC009】LODグループタブ、【SC010】テクスチャ調整タブにアクセスすることができる。
- **画面イメージ：**

図2-2-3　Rendering Toolkitメインパネル

【SC003】Sandbox Toolkitメインパネル

- **画面の目的・概要：**
  - Sandbox Toolkitの各機能にアクセスするためのパネル。
  - PLATEAUロゴ直下の4つのアイコンがボタンになっており、左から【SC011】トラック機能タブ、【SC012】アバター配置タブ、【SC013】乗り物配置タブ、【SC014】プロップ配置タブにアクセスすることができる。
- **画面イメージ：**

図2-2-4　Sandbox Toolkitメインパネル

【SC004】Maps Toolkitメインパネル

- **画面の目的・概要：**
  - Maps Toolkitの各機能にアクセスするためのパネル。
  - PLATEAUロゴ直下の3つのアイコンがボタンになっており、左から【SC016】PLATEAUモデル位置合わせタブ、【SC017】IFCモデル読込タブ、【SC018】GISデータ読込タブにアクセスすることができる。
- **画面イメージ：**

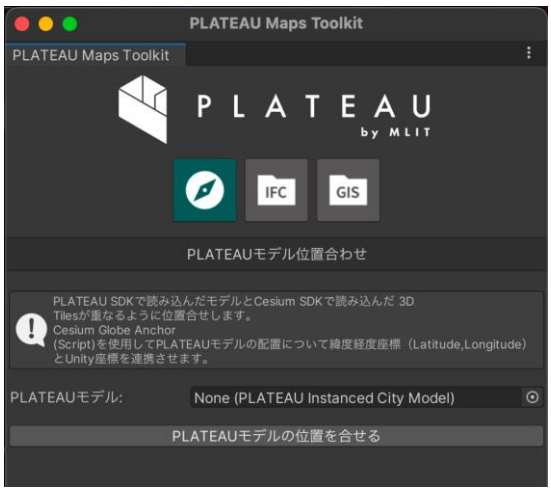


図2-2-5　Maps Toolkitメインパネル

【SC005】AR Extensionsメインパネル

- **画面の目的・概要：**
  - AR Extensionsの機能にアクセスするためパネル。
  - パネルからはARオクルージョン用のマテリアル変更機能にアクセス可能。
- **画面イメージ：**

図2-2-6　AR Extensionsメインパネル

【SC006】PLATEAU Utilitiesメインパネル

- **画面の目的・概要：**
  - PLATEAU Utilitiesの機能にアクセスするためパネル。
  - メッシュレンダラーの選択機能【FN034】、選択地物の整列機能【FN035】、メッシュ頂点の平面化機能【FN036】、プレハブへのライトマップの適用機能【FN037】 を利用可能。
- **画面イメージ：**

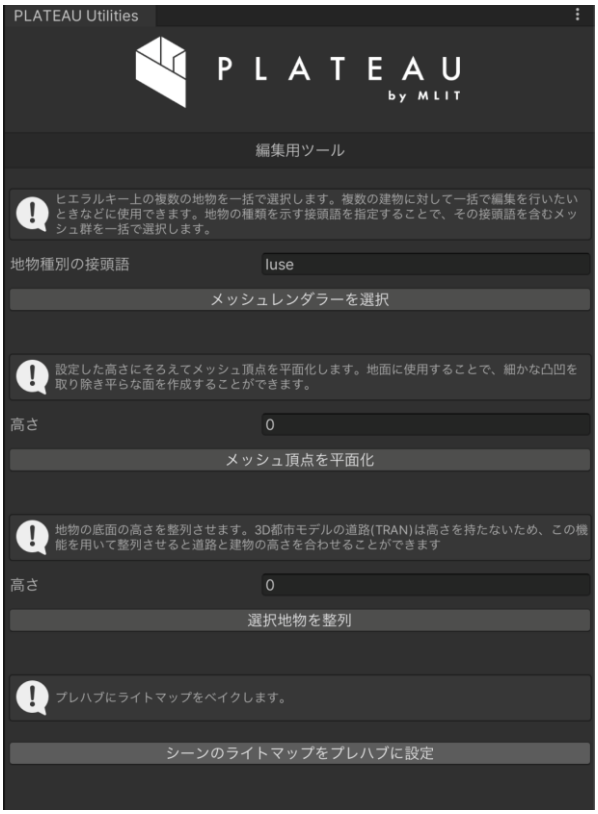


図2-2-7　PLATEAU Utilitiesメインパネル

【SC007】環境システムタブ

- **画面の目的・概要：**
  - Rendering Toolkitの環境システムを操作するためのタブ。
  - 「環境要素」を押下すると【SC007-2】環境システム編集画面に移行する。
- **画面イメージ：**

図2-2-8　環境システムタブ

【SC007-2】環境システム編集画面

- **画面の目的・概要：**
  - 環境システムの各パラメーターを編集するための画面。
  - 環境システムの詳細は【FN001】 ～【FN005】の環境システム機能を参照のこと。
- **画面イメージ：**

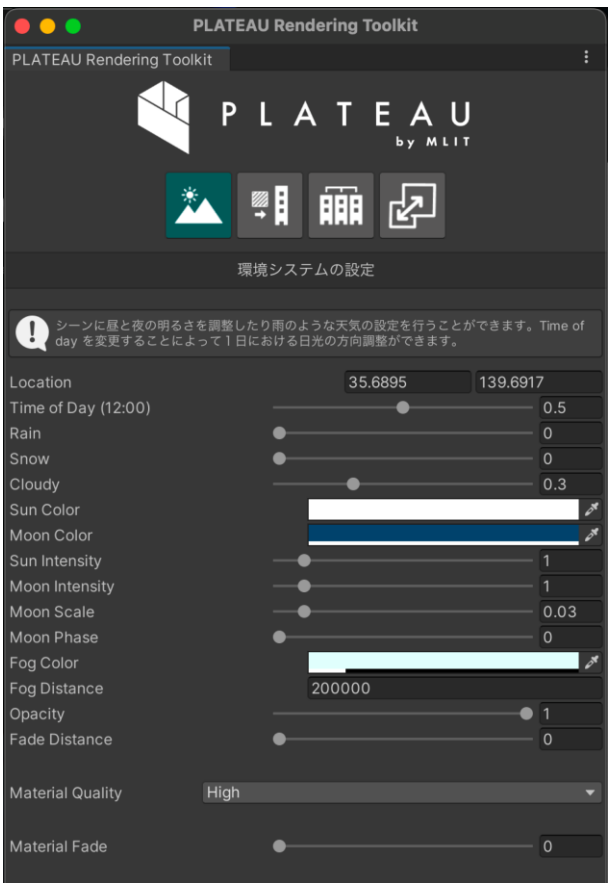


図2-2-9　環境システム編集画面

【SC008】自動テクスチャ生成タブ

- **画面の目的・概要：**
  - Rendering Toolkitの自動テクスチャの生成機能【FN006】、頂点カラーの設定機能【FN012、FN013】を利用するためのタブ。
  - 詳細は各機能の説明箇所を参照のこと。
- **画面イメージ：**

図2-2-10　自動テクスチャ生成タブ

【SC009】LODグループ生成タブ

- **画面の目的・概要：**
  - Rendering ToolkitのLODグループの生成機能【FN007】にアクセスするためのタブ。
  - 詳細は各機能の説明箇所を参照のこと。
- **画面イメージ：**

図2-2-11　LODグループ生成タブ

【SC010】テクスチャ調整タブ

- **画面の目的・概要：**
  - 画素調整機能【FN014】、テクスチャ解像度変更機能【FN015】を利用するためのタブ。
  - 詳細は各機能の説明箇所を参照のこと。
- **画面イメージ：**

図2-2-12　テクスチャ調整タブ

【SC011】トラック機能タブ

- **画面の目的・概要：**
  - Sandbox Toolkitのトラック機能を利用するためのタブ。
  - 「新しいトラックを作成」を押下すると新しいトラックの作成機能【FN016】が起動する。
  - 画面下部の「カメラマネージャーを作成」ボタンからは【FN023】のカメラインタラクション機能を起動可能。
  - 詳細は各機能の説明箇所を参照のこと。
- **画面イメージ：**

図2-2-13　トラック機能タブ

【SC012】アバター配置タブ

- **画面の目的・概要：**
  - 【FN020】アバターの配置機能を使用するためのタブ。
  - 「配置ツールを起動」を押下すると【FN019】共通配置ツールが起動する。
  - 画面下部の「カメラマネージャーを作成」ボタンからは【FN023】のカメラインタラクション機能を起動可能。
- **画面イメージ：**

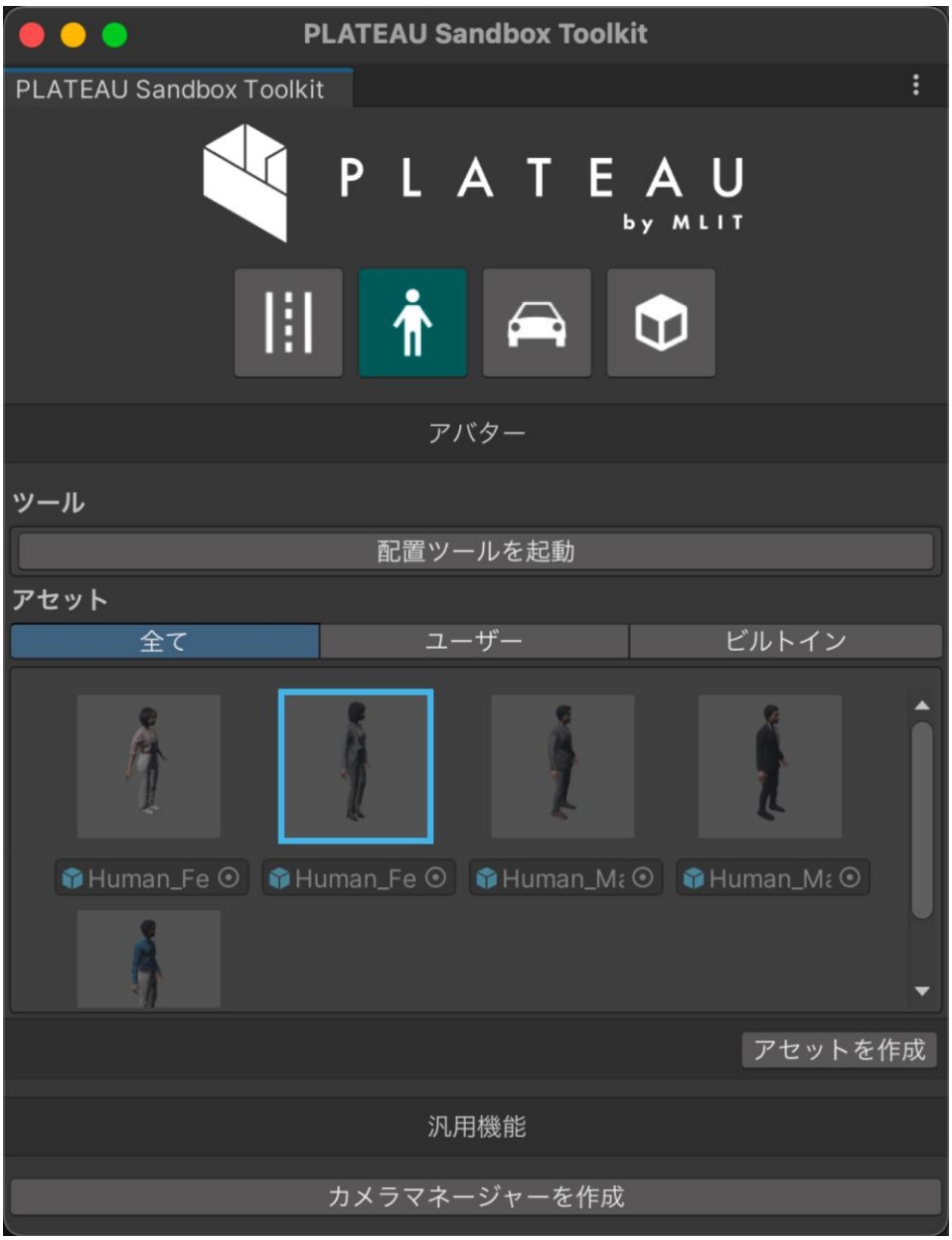


図2-2-14　アバター配置タブ

【SC013】乗り物配置タブ

- **画面の目的・概要：**
  - 【FN021】 乗り物の配置機能を使用するためのタブ。
  - 「配置ツールを起動」を押下すると【FN019】共通配置ツールが起動する。
  - 画面下部の「カメラマネージャーを作成」ボタンからは【FN023】のカメラインタラクション機能を起動可能。
- **画面イメージ：**

図2-2-15　乗り物配置タブ

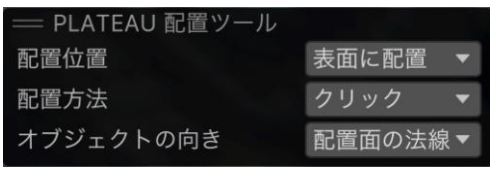
【SC014】プロップ配置タブ

- **画面の目的・概要：**
  - 【FN022】プロップの配置機能を使用するためのタブ。
  - 「配置ツールを起動」を押下すると【FN019】共通配置ツールが起動する。
  - 画面下部の「カメラマネージャーを作成」ボタンからは【FN023】のカメラインタラクション機能を起動可能。
- **画面イメージ：**

図2-2-16　プロップ配置タブ

【SC015】配置ツールウィンドウ

- **画面の目的・概要：**
  - 【SC012-SC014】のアセット配置画面で、【FN019】共通配置ツールを起動するとシーンビュー右下に表示されるウィンドウ。
  - 「配置方法」の選択値により表示が変化する。配置方法を「クリック」に設定しているときは左図の画面が表示される。配置方法を「ブラシ」に設定すると右側の画面が表示され、ブラシ配置専用の設定パラメーターが表示される。
  - 詳細は各機能の説明箇所を参照のこと。
- **画面イメージ：**



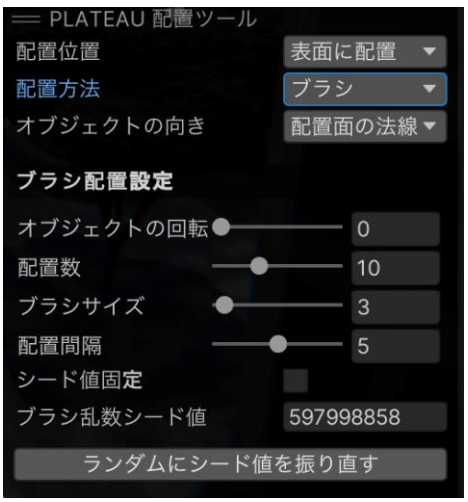


図2-2-17　配置ツールウィンドウ（クリック配置/ブラシ配置）

【SC016】PLATEAUモデル位置合わせタブ

- **画面の目的・概要：**
  - Maps ToolkitのPLATEAUモデル位置合わせ機能【FN024】にアクセスするためのタブ。
  - 詳細は各機能の説明箇所を参照のこと。
- **画面イメージ：**

図2-2-18　PLATEAUモデル位置合わせタブ

【SC017】IFCモデル読込タブ

- **画面の目的・概要：**
  - Maps ToolkitのIFCファイルの読込機能【FN026】を利用するための画面。
  - 詳細は各機能の説明箇所を参照のこと。
- **画面イメージ：**

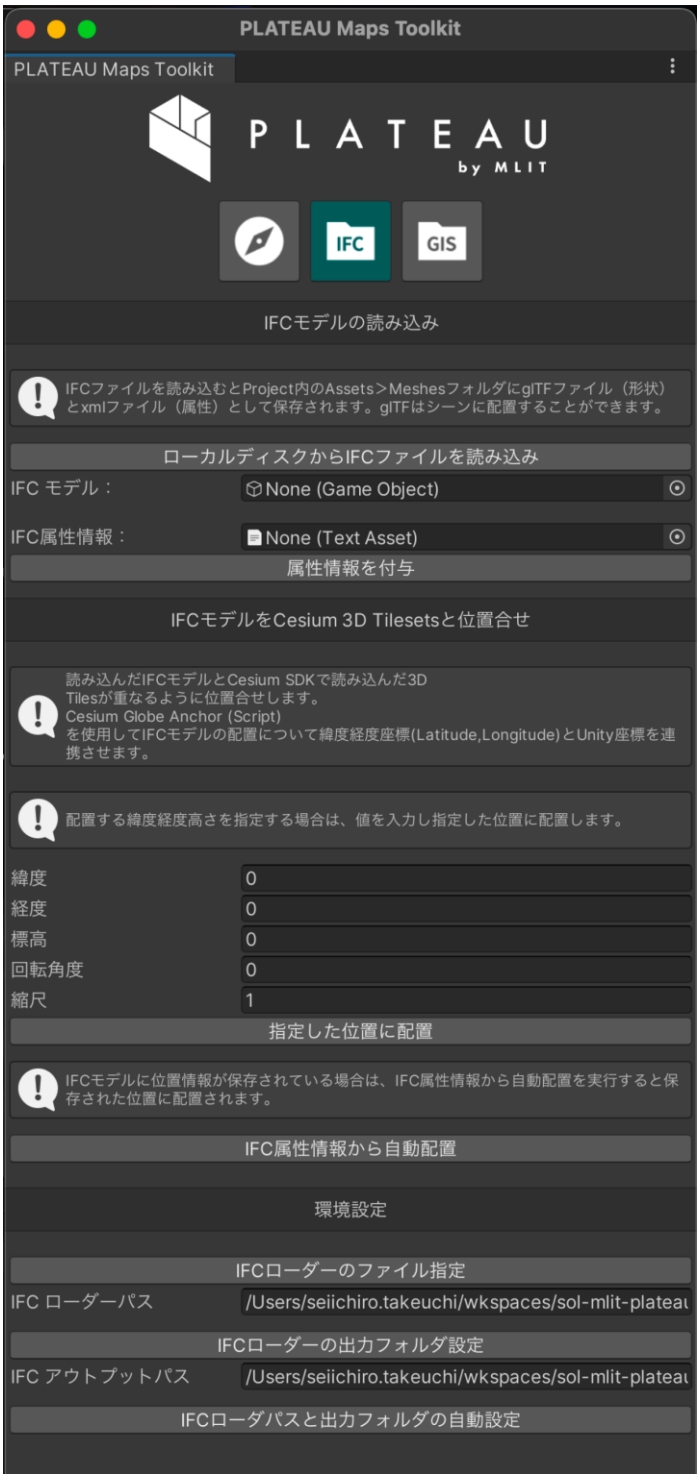


図2-2-19　IFCモデル読込タブ

【SC018】GISデータ読込タブ

- **画面の目的・概要：**
  - GISデータ読込機能（GeoJSONファイルの読込【FN027】、シェープファイルの読込【FN028】）にアクセスするためのタブ。
  - 詳細は各機能の説明箇所を参照のこと。
- **画面イメージ：**

図2-2-20　GISデータ読込タブ

【SC019】シーン保存UI

- **画面の目的・概要：**
  - シーンへの編集内容を保存したり、保存済みの編集内容を復元したりするためのUI。
  - Toolkitを導入すると自動的にシーンビュー右下に表示される。
- **画面イメージ：**

図2-2-21　シーン保存UI

2.4 アルゴリズム

2.4.1 Toolkitsで使用したアルゴリズム一覧

表2-2-5　PLATEAU SDK Toolkits for Unity のアルゴリズム一覧

ID	名称
AL001	時間の変更
AL002	雨、雪の表現
AL003	雲の濃度設定
AL004	フォグ設定
AL005	マテリアルフェード
AL006	テクスチャの生成
AL007	LODグループの生成
AL008	トイカメラ
AL009	ハーフトーン
AL010	ナイトビジョン
AL011	頂点カラー設定
AL012	画素調整機能
AL013	解像度変更
AL014	トラック作成
AL015	トラック移動コンポーネント
AL016	ランダムウォーク
AL017	オブジェクト配置
AL018	カメラマネージャー
AL019	3D都市モデルの位置合わせ
AL020	GISデータのゲームオブジェクト化
AL021	ARオクルージョン
AL022	Geospatial APIを用いた位置合わせ
AL023	ARマーカーを用いた高さ合わせ
AL024	ARマーカーを用いた位置合わせ
AL025	手動位置合わせ
AL026	プレハブへのライトマップの適用

2.4.1 Rendering Toolkitの使用アルゴリズム

【AL001】時間の変更

- 環境システムにおける時間変更機能は、ユーザがTimeOfDayスライダを操作してシミュレーション内の時間を調整し、それに基づいて太陽と月の動きを制御し、照明を適宜変更することを可能にする。このスライダは0から1の値を取り、24時間サイクルを表す。ユーザの操作に応じて、システムは太陽と月の位置を計算し、太陽と月のディレクショナルライトをシーン内で動的に調整する。
- **昼夜サイクルのシミュレーション：**
  TimeOfDayの値に基づき、CalculateSunDirection及びCalculateMoonDirectionメソッドを用いて環境システムの太陽と月のディレクショナルライトの角度を算出する。このプロセスは地理的位置情報を利用し、現実の時間に沿った太陽と月の動きを決定する。UpdateLightsメソッドにより、太陽が地平線以下にあるとき（夜）、あるいは月が地平線以下にあるとき（昼）は、それぞれのディレクショナルライトの強度が0に設定され、シーンへの照明が行われない。
- **太陽と月の照明表現：**
  - **太陽照明 (Sun)：**CalculateSunDirectionメソッドで太陽の方向を計算する。m_SunIntensityとm_SunColorパラメーターを用いてライトの強度と色を調節する。このディレクショナルライトが太陽として、昼間のシーンの物体を照らす。
  - **月照明 (Moon)：**CalculateMoonDirectionメソッドは月の方向を計算する。m_MoonIntensityとm_MoonColorパラメーターを用いてライトの強度と色を調節する。このディレクショナルライトが月明かりとして、夜間のシーンの物体を照らす。
- **ディレクショナルライトの実装と動作：**
  - ディレクショナルライト：UnityのLightコンポーネントを使用して太陽と月のディレクショナルライトを実装する。これらのライトは無限遠の光源として機能し、シーン内の全物体に対して平行光を投射し、鮮明な影を生成する。
  - 動作：UpdateLightsメソッドは、算出された計算結果に基づきライトの状態を更新する。時間と地理的位置に応じて、ライトの回転、強度、色がリアルタイムで調整され、日の出から日中、日没、夜に至る各シーンが自然に遷移する。光源の仰角により、ライトをON/OFFするロジックを含む。光源の仰角が地平線よりも上にある場合にのみONになり、その仰角が地平線下になるとOFFになる。

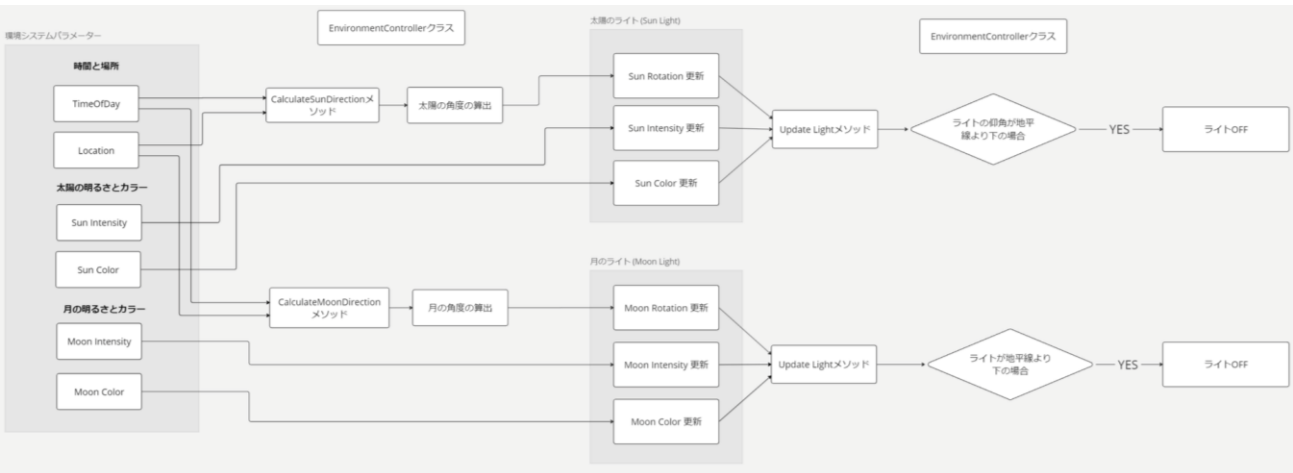


図2-2-22　時間の変更アルゴリズム

- **太陽の方向の計算: CalculateSunDirection**
  - CalculateSunDirection関数は、入力された日時と地理的位置（緯度及び経度）に基づいて、太陽の方向（高度と方位）を計算する。計算は天文学的アルゴリズムに従っており、以下の手順で実行される：
    1. **ユリウス日の計算：**特定の日時を天文学で広く受け入れられているユリウス日の形式に変換する。

```
double julianDate = dateTime.ToOADate() + 2415018.5;
```

    2. **恒星時の計算：**地球の自転による恒星の相対的な動きを追跡するために使用される恒星時を求める。

```
double julianCenturies = julianDate / 36525.0; // ユリウス世紀の計算
double siderealTimeHours = 6.6974 + 2400.0513 * julianCenturies;
double siderealTimeUT = siderealTimeHours +
    (366.2422 / 365.2422) * dateTime.TimeOfDay.TotalHours;
double siderealTime = siderealTimeUT * 15 + longitude;
```

    3. **太陽の平均黄経と平均近点角の計算：**太陽の軌道上の位置を定義するために必要な平均黄経と平均近点角を算出する。

```
const double k_Deg2Rad = Math.PI / 180;
const double k_Rad2Deg = 180 / Math.PI;

double meanLongitude = NormalizeAngle(k_Deg2Rad * (280.466 + 36000.77 *
julianCenturies));
double meanAnomaly = NormalizeAngle(k_Deg2Rad * (357.529 + 35999.05 *
julianCenturies));
```

    4. **黄道傾斜角の計算：**地球の軌道面に対する太陽の軌道の傾きを示す黄道傾斜角を計算する。

```
double obliquity = (23.439 - 0.013 * julianCenturies) * Math.PI / 180.0;
```

    5. **太陽の赤緯と赤経の計算：**天球上での太陽の位置を定義するための赤緯と赤経を計算する。

```
double rightAscension = Math.Atan2(
    Math.Cos(obliquity) * Math.Sin(elipticalLongitude),
    Math.Cos(elipticalLongitude));
double declination = Math.Asin(
    Math.Sin(rightAscension) * Math.Sin(obliquity));
```

6. **時角の計算：**観測者の経度と恒星時に基づき、特定の時点での太陽の位置を示す時角を求める。

```
double siderealTimeUT = siderealTimeHours +
            (366.2422 / 365.2422) * dateTime.TimeOfDay.TotalHours;
double siderealTime = siderealTimeUT * 15 + longitude;
double hourAngle = NormalizeAngle(siderealTime * Math.PI / 180.0) - rightAscension;
```

7. **高度と方位の計算：**計算された角度を使用し、観測者の地点から見たときの太陽の高度と方位を計算する。

```
double altitude = Math.Asin(Math.Sin(latitude * Math.PI / 180.0) *
   Math.Sin(declination) + Math.Cos(latitude * Math.PI / 180.0) *
   Math.Cos(declination) * Math.Cos(hourAngle));
double aziNom = -Math.Sin(hourAngle);
double aziDenom =
   Math.Tan(declination) * Math.Cos(latitude * Math.PI / 180.0) -
   Math.Sin(latitude * Math.PI / 180.0) * Math.Cos(hourAngle);
double azimuth = Math.Atan(aziNom / aziDenom);
```

- **月の方向の計算：CalculateMoonDirection**
  - CalculateMoonDirection関数は、月の方向（高度と方位）を決定するために同様の天文学的計算を行うが、月の独自の運動を考慮に入れる。：
    1. **月の軌道要素の計算：**月の黄道経度、平均近点角及び平均距離を含む月の軌道要素を算出する。

```
double eclipLongitude = (218.316 + 13.176396 * julianDate) * k_Deg2Rad;
double lunarMeanAnomaly = (134.963 + 13.064993 * julianDate) * k_Deg2Rad;
double lunarMeanDistance = (93.272 + 13.229350 * julianDate) * k_Deg2Rad;
```

    2. **高度と方位の計算：**上記の軌道要素に基づいて、観測者の地点から見た月の高度と方位を計算する。

```
double lng = eclipLongitude + k_Deg2Rad * 6.289 * Math.Sin(lunarMeanAnomaly);
double lat = k_Deg2Rad * 5.128 * Math.Sin(lunarMeanDistance);
double obliquity = k_Deg2Rad * 23.4397; // 黄道傾斜角

// 仮想傾斜を適用した赤緯の計算
double dec = Math.Asin(Math.Sin(lat) * Math.Cos(obliquity) +
   Math.Cos(lat) * Math.Sin(obliquity) * Math.Sin(lng));
double distance = 385000 - 20905 * Math.Cos(lunarMeanAnomaly);
double rightAscension = Math.Atan2(Math.Sin(lng) * Math.Cos(obliquity) -
   Math.Tan(lat) * Math.Sin(obliquity), Math.Cos(lng));
double h = NormalizeAngle((k_Deg2Rad * (280.16 + 360.9856235 * julianDate) - lw)
- rightAscension);
```

```
double altitude = Math.Asin(Math.Sin(phi) * Math.Sin(dec) +
    Math.Cos(phi) * Math.Cos(dec) * Math.Cos(h));
double azimuth = Math.Atan2(Math.Sin(h), Math.Cos(h) * Math.Sin(phi) -
    Math.Tan(dec) * Math.Cos(phi));
```

- **高度と方位のディレクショナルライトへの適用：**
  太陽や月の高度と方位は、Unity内でのディレクショナルライトの方向を決定するために使用される。
  - **高度（Altitude）：**太陽や月が地平線からどれだけ上にあるかを示す角度。この値は、ディレクショナルライトの垂直方向の傾斜に対応する。
  - **方位（Azimuth）：**北を基準とした太陽や月の水平方向の位置を示す。この値は、ディレクショナルライトの水平方向を決定する。
- **ライトの方向の計算：**
  Unityでは、これらの角度をオイラー角に変換し、ディレクショナルライトのtransform.rotationプロパティに適用する。オイラー角は、3次元空間におけるオブジェクトの向きを表す一般的な方法である。
  このコード例は、太陽のディレクショナルライトに方向を設定する方法を示している。

```
Quaternion sunRotation = Quaternion.Euler(sunAltitude, sunAzimuth, 0);
sunLight.transform.rotation = sunRotation;
```

  Quaternion.Eulerメソッドは高度と方位からQuaternionを生成し、ライトの回転に適用する。これにより、CalculateSunDirection及びCalculateMoonDirection関数から得られた高度と方位をもとに、太陽と月のディレクショナルライトの方向を設定し、昼夜サイクルを表現することが可能となる。
- **URPとHDRPでの空の実装（天球の描画）：**
  - **HDRP** では、EnvironmentVolumeにひも付けられたVolumeコンポーネント内のPhysically Based Sky機能と連携し、物理ベースの大気散乱をリアルタイムでシミュレートし、太陽のディレクショナルライトの角度に基づいて空の色が動的に変化する。
  - **URP** では、HDRPの Physically Based Sky と類似した表現をURPで実現するために、カスタムSkyboxシェーダー PhysicallyBasedSky.shaderを実装した。このカスタムシェーダーも物理ベースの大気の散乱効果をシミュレートし、太陽と月の動きに応じて空の色が動的に変化する。
  - **HDRP環境における空の色の変化と太陽・月のレンダリング**
    - **物理ベースの空のレンダリング （Physically Based Sky Rendering）：**
      HDRP では Physically Based Sky を採用し、大気の物理的特性に基づく空のレンダリングを行う。このモデルは、ディレクショナルライトの角度や強度に応じて自動的に空の色調や明るさを計算し、現実世界に近い空の現象をシミュレートする。例えば、太陽が地平線に近づく際、ミー散乱（大気中の大きな粒子による散乱）とレイリー散乱（分子や小さな粒子による散乱）の効果が増加し、空はオレンジや赤に染まる。日の出や日没時に鮮やかな色のグラデーションが自然に表現される。
    - **太陽のレンダリング：**
      Physically Based Sky モデルは太陽の光源としてディレクショナルライトを用いる。ディレクショナルライトの角度と強度によって、太陽は天球に自動的に描画される。

- **月のレンダリング：**
  HDRPではライトのパラメーターに、Celestial Body > Surface Textureという項目が存在する。ここに月のテクスチャを設定することで、ライトの角度に応じて指定したテクスチャが天球に投影される。月のレンダリングに関してはこの機能を介して月のディレクショナルライトにテクスチャを設定し、月を描画している。

- **URP環境における空の色の変化と太陽・月のレンダリング：**
  - **物理ベースの空のレンダリング：**
    URP では、カスタムSkyboxシェーダーPhysicallyBasedSky.shaderを通じて物理ベースの大気散乱を計算し、太陽と月のディレクショナルライトの角度をもとに、昼間の青空、日の出や日没時の赤みがかった空を描画する。レイリー散乱とミー散乱を含む計算を行い、時間の経過に伴い空の色がシュミレートされる。
    - **大気散乱の計算: Scatter 関数：**
      Scatter関数は、大気を通過する光の散乱の計算をするために使用される。レイリー散乱は、大気中の分子によって引き起こされる散乱であり、空が青く見える主な理由である。ミー散乱は、大気中のより大きな粒子（エアロゾルなど）による散乱で、日の出や日没時に空が赤く見える理由となる。オゾン層の効果も考慮に入れ、これらの散乱メカニズムを合わせて、大気を通過する光の色と強度の変化を計算する。

```
float3 Scatter(float3 vDepth)
{
  return exp(-(vDepth.x * RAYLEIGH + vDepth.y * MIE * 1.1 + vDepth.z * OZONE));
}
```

    - **レイマーチによる光の散乱計算: RayMarch 関数：**
      RayMarch関数は、レイマーチング技術を利用して、視線に沿った光の散乱を積分する。この関数は、GetMainLight()で取得された太陽光（主要な光源）の方向と強度を考慮し、空の色の変化をシミュレートする。サンプル数Samplesに基づき、光のパスを細かくサンプリングし、それぞれのポイントでの大気散乱を計算することで、大気を通過する光の全体的な影響がシミュレートされる。
  - **太陽のレンダリング：**
    PhysicallyBasedSkyシェーダー内で太陽はディレクショナルライトとして表され、その動きは太陽の位置（角度）に基づきシミュレートされる。時間の経過に伴い、太陽の位置、強度及び色の調整が行われ、空の色の変化を引き起こす。太陽の光源はGetMainLight()関数を通じて取得され、この情報はRayMarch関数内で使用され、空のレンダリングにおける光の散乱を計算する。
  - **月のレンダリング：**
    月のレンダリングには環境システムの月のディレクショナルライトの角度に基づき、シェーダーのテクスチャプロパティを介して月のテクスチャが天球に投影されることで描画される。

```
// 月のテクスチャから色情報を取得
float4 moonSample = tex2D(_MoonTexture, moonTransformedUV);

// 月の最終的な表示色を計算
scattering += moonFinalVisibility * moonSample * _MoonEmissionColor;
```

【AL002】雨、雪の表現

- 環境システムにおける天候変更機能は、Unityのビジュアル エフェクト機能（VFX Graph や Particle System）を活用して降水現象をシミュレーションする。雨や雪のパラメーターのスライダーの値を調整することでシステムは適宜パーティクルの密度を調整し、雨粒や雪片の視覚効果が生成される。
- **雨と雪の表現:**
  - **雨:** 雨のパーティクルは密度、雨粒のサイズ、重力、速度と方向などを細かく設定し、現実の雨の動きを表現。環境システムのRainスライダーで降雨強度を調整可能で、値が高いほど降雨量が増加する。
  - **雪:** 雪のパーティクルは密度、雪片のサイズ、重力、乱気流などを設定し、現実の雪の動きを表現。環境システムのSnowスライダーで降雪量を調整可能で、値が高いほど降雪量が増加する。
- **エフェクトの最適化:**
  雨や雪はシーン全体に降らせると大量のパーティクルが必要になり、処理負荷が非常に高くなるため、プレイヤー（カメラ）の前に集中させる形で降らせて、パフォーマンスの最適化を行う。カメラの移動に合わせて雨や雪のパーティクルの位置が動的に更新される。
- **エフェクトの選択:**
  HDRP環境では下図のとおり、自動的にVFX Graphが選択されるが、URP環境では環境システムの作成の際にVFX GraphとParticleSystemで作成されたエフェクトのどちらを利用するかを選択可能。VFX Graphは、GPUベースのエフェクト作成ツールでノードベースのインターフェースで大量のパーティクルを扱うことや複雑なVFXの作成が可能な一方で、ComputeShaderに対応したデバイスが推奨される。そのため、古いモバイルデバイスでは動作に対応していない。一方ParticleSystemはCPUベースのエフェクト制作ツールで、デバイスによる制限がない為、シンプルな効果や、パフォーマンスを重視するモバイル向けアプリケーションに適している。

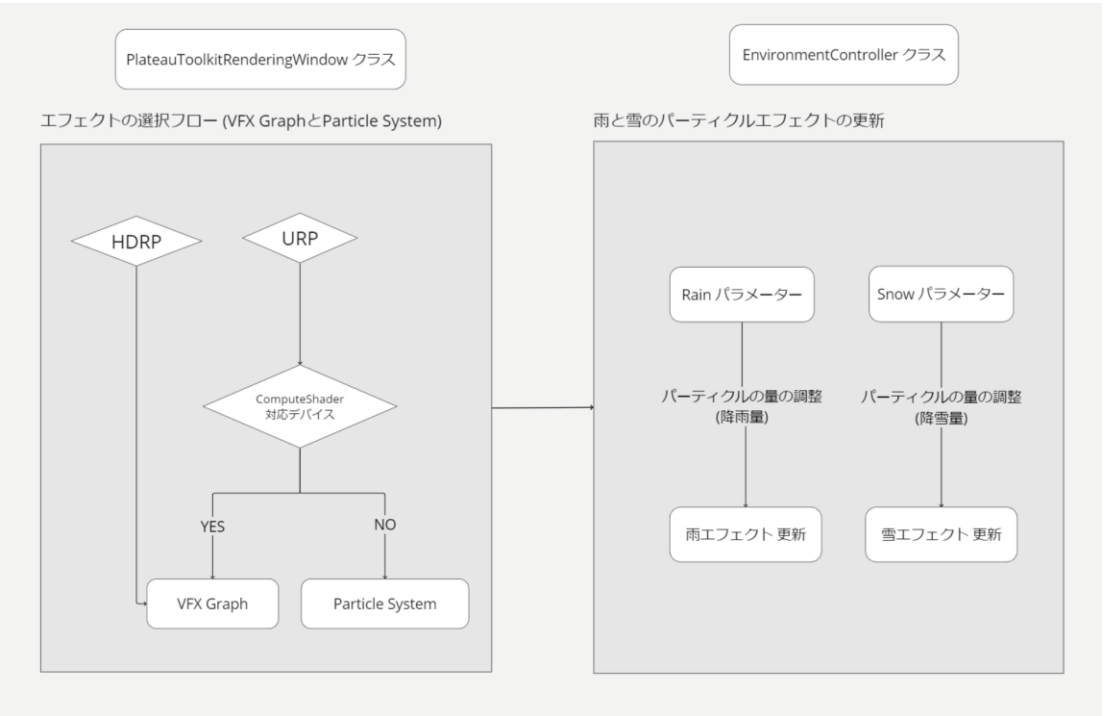


図2-2-23　雨・雪の表現のアルゴリズム

- **雨の実装:**
  以下のコードは雨の実装を示す。m_Rain パラメータを使用して降雨量を動的に調整する。VFX Graphおよび Particle Systemの両方で、パーティクルの生成率が m_Rain の値に基づいて更新される。またエフェクトの位置はカメラの移動に追従し動的に更新される。

```
// 雨エフェクトの設定 (VFX Graph)
if (m_RainVFX)
{
    // 雨のパーティクル発生率を設定
    if (m_RainVFX.HasInt("Particle Rate"))
    {
        m_RainVFX.SetInt("Particle Rate", (int)(m_Rain * 20000));
    }

    // カメラの位置に雨エフェクトを移動
    if (Camera.main != null)
    {
        m_RainVFX.transform.position = Camera.main.transform.position;
    }
}

// 雨エフェクトの設定 (Particle System)
if (m_RainMobile)
{
    ParticleSystem.EmissionModule emission = m_RainMobile.emission;
    emission.rateOverTime = m_Rain * 500; // 発生率を調整
    // カメラの位置にエフェクトを移動
    if (Camera.main != null)
    {
        m_RainMobile.transform.position = Camera.main.transform.position;
    }
}
```

- **雪の実装:**
  雪のエフェクトについても同様に m_Snow パラメータを使用して降雪量を動的に調整する。VFX Graphおよび Particle Systemの両方で、パーティクルの生成率が m_Snow の値に基づいて更新される。

```
// 雪エフェクトの設定 (VFX Graph)
if (m_SnowVFX)
{
    // 雪のパーティクル発生率を設定
    if (m_SnowVFX.HasInt("Particle Rate"))
    {
        m_SnowVFX.SetInt("Particle Rate", (int)(m_Snow * 20000));
    }

    // カメラの位置に雪エフェクトを移動
    if (Camera.main != null)
    {
        m_SnowVFX.transform.position = Camera.main.transform.position;
    }
}

// モバイル版の雪エフェクト設定 (Particle System)
if (m_SnowMobile)
{
    ParticleSystem.EmissionModule emission = m_SnowMobile.emission;
    emission.rateOverTime = m_Snow * 500; // 発生率を調整
    // カメラの位置にエフェクトを移動
    if (Camera.main != null)
    {
        m_SnowMobile.transform.position = Camera.main.transform.position;
    }
}
```

【AL003】雲の濃度設定

- 環境システムにおける雲の機能は、Cloudyパラメーターに応じて空に雲を描画する。HDRPとURPでは雲の実装方法が異なる。HDRPではCloudLayerを利用し、詳細な雲の表現が可能。URPではカスタムSkyboxシェーダーを用いて、テクスチャで設定した雲が描画される。
- **雲の濃度設定：**
  環境システムの Cloudy パラメーターに応じて雲の濃度が変わる。パラメータの値が高くなるほど、空に描画される雲の濃度が濃くなる。
- **HDRPでの雲の実装：**
  HDRP ではEnvironmentVolumeにひも付けられたVolumeコンポーネント内の CloudLayer 機能を用いてリアリスティックな雲が描画される。環境システムの Cloudy パラメーターで設定された値に基づき、UpdateCloudLayerVolume メソッドを通じて CloudLayer の不透明度を更新し、雲の濃度を動的に調整する。
- **URPでの雲の実装：**
  URP では、カスタムSkyboxシェーダーPhysicallyBasedSky.shaderを通じて雲が描画される。このシェーダーで設定された雲のテクスチャを用いて天球上に雲が描画される。環境システムのCloudyパラメーターを用いてシェーダー内で雲のテクスチャの不透明度を更新し、雲の濃度を動的に調整する。

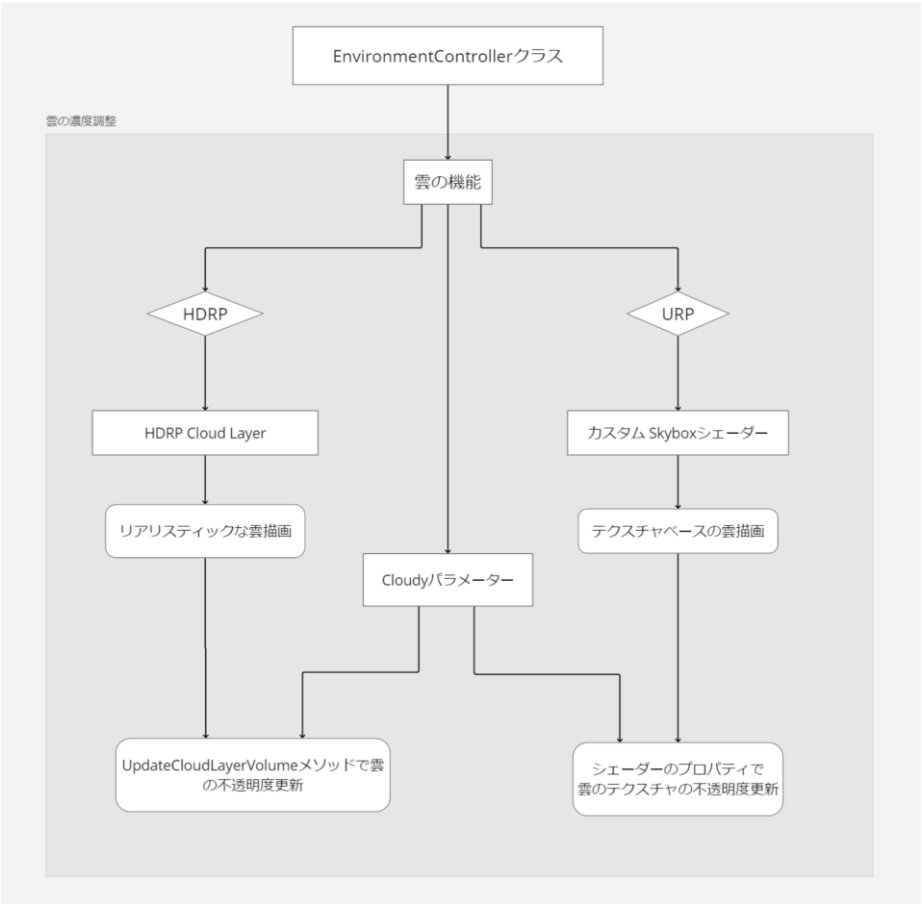


図2-2-24　雲の濃度設定のアルゴリズム

【AL004】フォグ設定

- 環境システムでは、Fog DistanceとFog Colorパラメーターを介して、シーン全体を覆うフォグの距離と色味を調整し、霧の効果をシミュレートする。シーン全体に視覚的な奥行きの効果を与え、大気のリアリズムを表現する。
- **HDRPでのフォグの実装：**
  HDRP では、環境システムのFogDistance及びFogColorパラメーターに基づき、EnvironmentVolumeにひも付けられたVolumeコンポーネント内のFogの設定を動的に更新する。meanFreePathプロパティにFogDistance、tintプロパティにFogColorを設定し、フォグの距離と色を更新する。
- **URPでのフォグの実装：**
  URP では、環境システムのFogDistance及びFogColorパラメーターに基づき、UpdateLightsメソッド内で、RenderSettingsの設定を動的に更新する。RenderSettings.fogEndDistanceにFogDistance、RenderSettings.fogColorにFogColorを設定し、フォグの距離と色を更新する。

【AL005】マテリアルフェード

- 環境システムでは、Material FadeとBuilding Colorパラメーターを用いて、3D都市モデルの地物のテクスチャ色を任意の色とブレンディングすることが可能。テクスチャ色を保持した状態から完全な単色までの遷移が可能で、この機能を用いることで、モデルの視認性向上や視覚的な統一感を出すことができる。
- **単色化機能の仕組み：**
  - この機能はRendering Toolkitの地物用のカスタムシェーダーに実装されている。地物を選択してRendering Toolkitが提供する自動テクスチャリング機能を実行することで、このカスタムシェーダーが自動的に地物に適用される。Buildingカスタムシェーダーは、主にUnityのシェーダーグラフを使用し、ノードベースのビジュアルプログラミング環境で実装されている。
  - 環境システムのBuilding Colorで指定した色とMaterial Fadeで設定した値をカスタムシェーダーに渡し、シェーダー内で定義された地物のテクスチャとBuilding ColorをMaterial Fade の値を用いて線形補間することによって地物の色がブレンディングされる。
  - 以下に示すサンプルコードは、シェーダーグラフでの実装をテキストベースで表現した疑似コードとなる。シェーダーの単色化機能の仕組みを理解しやすくすることを目的とする。

```
BlendingColor = lerp(Original, BuildingColor, MaterialFade);
```

  - Original - 遷移の開始点となる値。この場合、地物の元のテクスチャ色
  - BuildingColor - 遷移の終点となる値。この場合、環境システムで指定された単色
  - Fade - 2つの値の間での遷移を制御するパラメーター。0から1の値を取る

  Lerp関数を使用してOriginalの色とBuildingColorの色をMaterialFadeパラメーターに基づいてブレンディングする。Fadeの値に応じて、出力されるBlendingColorはOriginalとBuildingColorの間を滑らかに遷移する

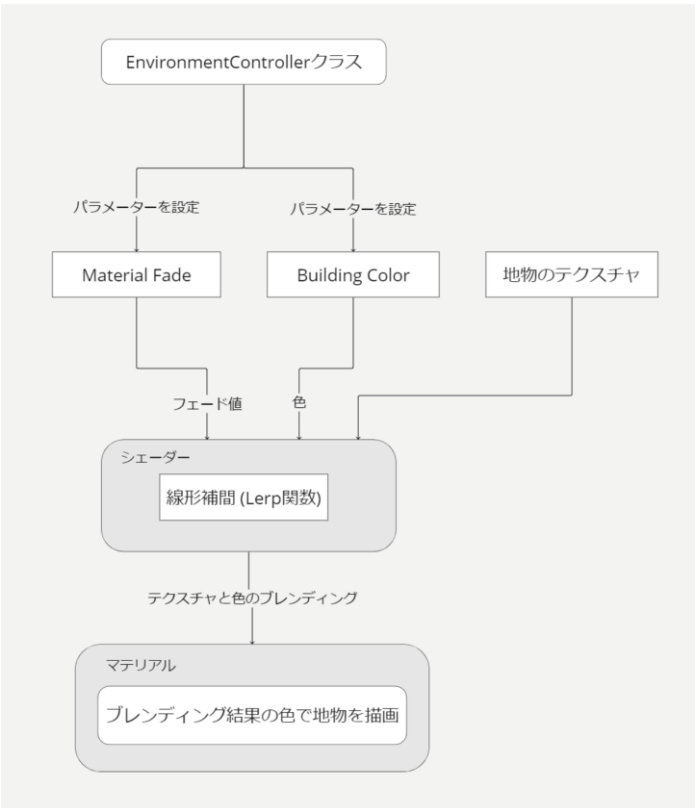


図2-2-25　マテリアルフェードのアルゴリズム

【AL006】テクスチャの生成

- PLATEAU SDKでインポートした3D都市モデルに対してランダムなパターンのテクスチャを生成して適用する。【FN006】自動テクスチャの生成機能で使用しているアルゴリズム。
- **ヒエラルキーの階層変更：**
  インポートした3D都市モデルのLODにより、テクスチャ生成処理のプロセスが異なる。生成処理適用の前段階として、PLATEAU SDKを用いてインポートした3D都市モデルオブジェクトのヒエラルキー上の階層を建物単位に変更する。以下に具体例を示す。
  - PLATEAU SDKでダウンロードした直後は専用親ゲームオブジェクト（下記の場合は 13100_tokyo23-ku_2022_citygml_1_2_op 2）に3D都市オブジェクトが格納されている。まとまったmeshグループの中で、各LODに応じた中間階層が用意されており、その中に各建物に対応したmeshが付与されたゲームオブジェクトが配置されている状態となっている。

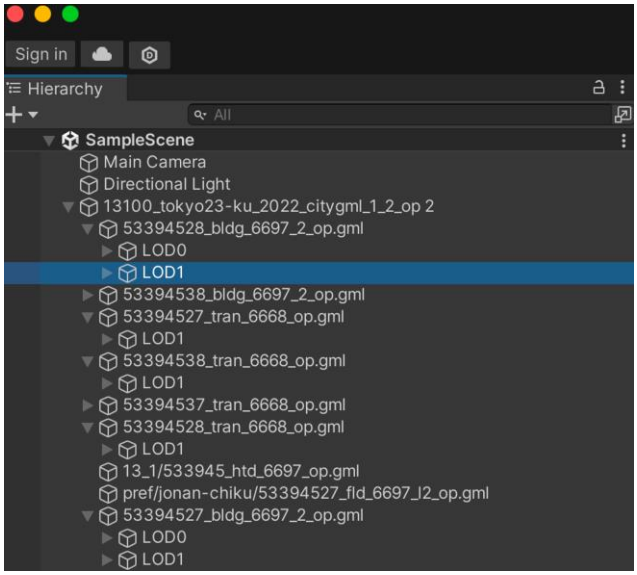


図2-2-26　SDKからインポート直後のヒエラルキー

  - 「テクスチャ自動生成」押下後は複数の建物がまとまってしまっているmeshが存在する。そのため、テクスチャを適用するために、建物単位のmeshのグループを作成する。
    「ParentForGroupedObjects」と呼ばれるゲームオブジェクトに全てのモデルデータが移動する。

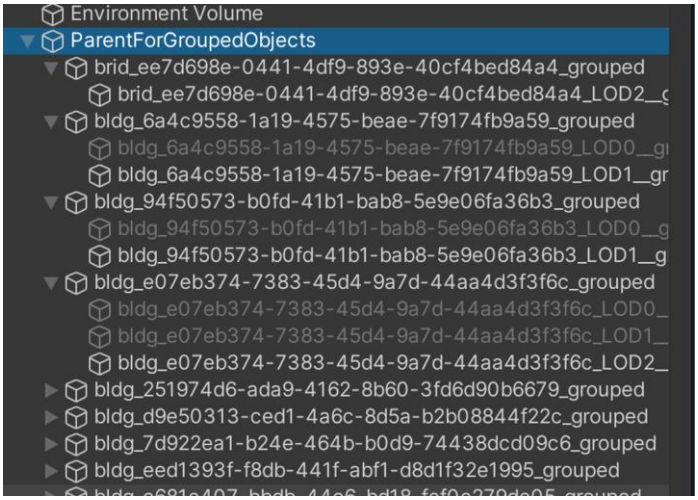


図2-2-27　テクスチャ自動生成後のヒエラルキー

- **ビル専用シェーダー：**
  夜間に発光する窓を設置するため、作成されたメッシュデータに対して一律で専用のカスタムシェーダーを適用する。メッシュの法線方向から側面/屋上を判別し、側面に対しては窓を生成する。

- **テクスチャ画像の生成：**
  モデルのLODにより処理が分岐する。
  - **LOD1モデルの処理：**
    - **LOD2モデルが存在しない場合：**メッシュの法線が上向きである場合は屋上として認識する。屋上の淵部分を作成する為に、屋上の外周エッジを内側に押し出すメッシュ処理を行う。その後、UV展開を行った上で、予め用意された側面と屋上のテクスチャファイルを利用して複数パターンのマテリアルをランダムに適応する。

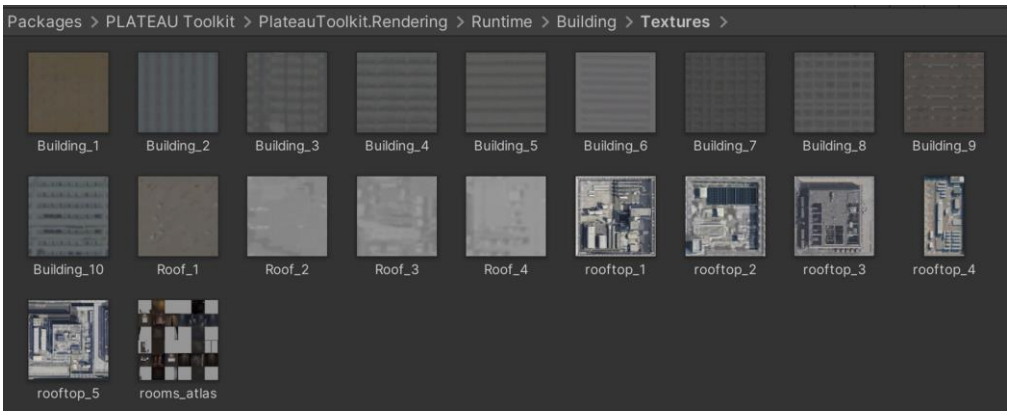


図2-2-28　自動テクスチャ用のテクスチャファイル

    - **LOD2モデルが存在する場合：**LOD2モデルを複製し、屋上と底面の頂点をフラット化した上で頂点の結合処理を行い、LOD2のメッシュを簡略化する。フラット化する際の天井の高さはLOD１から取得した高さを参照する。テクスチャはLOD2のものを利用する。
  - **LOD2モデルの処理：**ビル専用シェーダーのみを適用する。殆どのLOD2はデフォルトのテクスチャが存在するため、テクスチャが重複して見えないように昼間は薄く窓の色を重ね、夜間は通常通り発光するよう調整する。

- **底面特定と発光表現：**
  ビルの天井部分の最小バウンディングボックス平面を計算し、求めた座標から平面メッシュを作成して、ビルの底面に配置する。その上で、建物の底が光って見えるように発光マテリアルを作成した平面メッシュに適応する。

- **高層建築向けの処理（航空障害灯の適用）：**
  ビルのバウンディングボックスを参照し、高さが60m以上の場合は高層建築と区分し以下の処理を行う。
  1. ビルのバウンディングボックスの上面の四隅に近接するメッシュの頂点座標を取得
  2. 予め用意された航空障害灯用の赤いライトのビルボードのプレハブを取得した頂点座標に配置

- **ヒエラルキーの階層変更の実装詳細：**
  テクスチャ生成を行う前のヒエラルキー階層の変更処理のスクリプトは以下の通り。
  PLATEAU SDKでインポートした3D都市モデルのGameObjectにはPlateauInstancedCityModel コンポーネントが付与されている。このコンポーネントを検索し、操作の対象とする。

```
 internal void GroupObjects()
     {
         Dictionary<string, GameObject> rootObjects = new Dictionary<string, GameObject>();
         Dictionary<string, Dictionary<string, GameObject>> lodObjects = new Dictionary<string,
Dictionary<string, GameObject>>();

         PLATEAUInstancedCityModel[] plateauModelRootObject =
GameObject.FindObjectsByType<PLATEAUInstancedCityModel>(FindObjectsSortMode.None);
         List<Transform> copyOfGroupedBuildingsList = new List<Transform>();
         List<GameObject> objectsToDelete = new List<GameObject>();

         for (int i = 0; i < plateauModelRootObject.Length; i++)
         {
             if (plateauModelRootObject[i] == null)
             {
                 continue;
             }

             for (int j = 0; j < plateauModelRootObject[i].transform.childCount; j++)
             {
                 Transform gmlRoot = plateauModelRootObject[i].transform.GetChild(j);
                 float stepProgress = i / (float)plateauModelRootObject.Length;
                 if (EditorUtility.DisplayCancelableProgressBar("地物のグルーピング", "グルーピング中",
stepProgress))
                 {
                     UnityEngine.Debug.Log("LOD生成用のグルーピングがキャンセルされました。");
                     break;
                 }
```

（次ページに続く）

```
                objectsToDelete.Clear();
                for (int k = 0; k < gmlRoot.childCount; k++)
                {
                    Transform lodGroupedBuildings = gmlRoot.transform.GetChild(k);
                    if (lodGroupedBuildings.name.Contains("LOD0") || lodGroupedBuildings.name.Contains("LOD1") ||
lodGroupedBuildings.name.Contains("LOD2"))
                    {
                        copyOfGroupedBuildingsList.Clear();
                        for (int l = 0; l < lodGroupedBuildings.transform.childCount; l++)
                        {
                            Transform building = lodGroupedBuildings.transform.GetChild(l);
                            copyOfGroupedBuildingsList.Add(building);
                        }

                        for (int m = 0; m < copyOfGroupedBuildingsList.Count; m++)
                        {
                            Transform buildingCopy = copyOfGroupedBuildingsList[m];
                            GroupByLodName(gmlRoot, rootObjects, lodObjects, buildingCopy.gameObject,
lodGroupedBuildings.name);
                        }
                    }

                    if (lodGroupedBuildings.childCount == 0
&& !PrefabUtility.IsPartOfAnyPrefab(lodGroupedBuildings.gameObject))
                    {
                        objectsToDelete.Add(lodGroupedBuildings.gameObject);
                    }
                }

                foreach (GameObject obj in objectsToDelete)
                {
                    try
                    {
                        GameObject.DestroyImmediate(obj);
                    }
                    catch (Exception e)
                    {
                        Debug.LogException(e);
                        Debug.Log("object name: " + obj.name);
                    }
                }
            }
        }
        EditorUtility.ClearProgressBar();
        OnProcessingFinished?.Invoke();
    }
```

- **LOD1モデル（LOD2モデルがないもの）のテクスチャ生成処理：**
  前述の通り、モデルのLODによりテクスチャ生成処理の内容が分岐する。ProcessLOD1メソッドは、LOD2モデルがないLOD1モデル向けのもので、以下の処理を行う。
  1. 航空障害灯や屋上の囲いを表現するテクスチャーを適応するため、屋上部分に該当する点の位置を計算する（listofRoofTriangles）
  2. 夜間にビルの底面が光るエフェクトをつけるため、床の検出を行う（listofBottomTriangles）
  3. 地物の高さによって取りうるテクスチャの種類が異なるため、地物の高さを計測してその高さに合わせてテクスチャの選定を行う。（SetMaterialFromHeight(meshRenderer, height); ）
  4. 上記で計算した屋上の点を使ってライトを配置（PlaceObstacleLightsOnBuildingCorners）
  5. 夜間にビルの底面が光るエフェクトの適用（CreatePlaneUnderBuilding）

```
void ProcessLOD1(GameObject go, MeshRenderer meshRenderer, MeshFilter meshFilter)
    {
        Undo.RecordObject(meshRenderer, "Optimize Mesh");
        Undo.RecordObject(meshFilter, "Optimize Mesh");

        // Add the Processed component to the GameObject
        Undo.AddComponent<Processed>(go);

        // make new instance of mesh
        meshFilter.sharedMesh =
PlateauRenderingMeshUtilities.CreateMeshInstance(meshFilter.sharedMesh, false);

        // unwrap uvs as best fit around edge of building
        PlateauRenderingMeshUtilities.UnwrapUVs(meshFilter.sharedMesh, 10);

        // select roof triangles
        List<int> listofRoofTriangles =
PlateauRenderingMeshUtilities.SelectFacesFacingUp(meshFilter.sharedMesh, 15);

        // get roof boundary edges
        List<System.Tuple<int, int>> roofBoundaryEdges =
PlateauRenderingMeshUtilities.GetBoundaryEdges(listofRoofTriangles);

        // extrude roof by 1m
        List<List<Tuple<int, int>>> sortedEdgeGroups =
PlateauRenderingMeshUtilities.SortEdgesByConnection(roofBoundaryEdges, meshFilter.sharedMesh);
        List<Tuple<int, int>> newRoofEdges = null;
        if (sortedEdgeGroups.Count > 0)
        {
            PlateauRenderingMeshUtilities.OffsetRoofEdgesResult result =
PlateauRenderingMeshUtilities.OffsetRoofEdges(meshFilter.sharedMesh, sortedEdgeGroups[0], -0.8f,
0.001f);
```

（次ページに続く）

```
        newRoofEdges = result.NewEdges;
        }

        // select bottom triangles
        List<int> listofBottomTriangles =
PlateauRenderingMeshUtilities.SelectFacesFacingDown(meshFilter.sharedMesh, 15);

        // delete bottom triables
        List<System.Tuple<int, int>> remainingEdges =
PlateauRenderingMeshUtilities.DeleteSelectedTriangles(meshFilter.sharedMesh,
PlateauRenderingMeshUtilities.SelectFacesFacingDown(meshFilter.sharedMesh, 15));

        // set base color to green for Windows
        PlateauRenderingMeshUtilities.SetTriangleVertexColors(meshFilter.sharedMesh,
PlateauRenderingMeshUtilities.GetAllTriangles(meshFilter.sharedMesh), Color.green);

        // get all roof triangles
        listofRoofTriangles =
PlateauRenderingMeshUtilities.SelectFacesFacingUp(meshFilter.sharedMesh, 15);

        // set new edge color to green for for first UV bake
        PlateauRenderingMeshUtilities.SetEdgeColors(meshFilter.sharedMesh, roofBoundaryEdges,
Color.green);

        Bounds boundingBox = meshRenderer.bounds;
        PlateauRenderingBuildingUtilities.SetBuildingVertexColorForWindow(meshFilter.sharedMesh,
boundingBox, go);

        PlateauRenderingMeshUtilities.SetEdgeColors(meshFilter.sharedMesh, newRoofEdges,
Color.blue);

        // set new edge color to red for for roof UV unwrap
        PlateauRenderingMeshUtilities.SetEdgeColors(meshFilter.sharedMesh, roofBoundaryEdges,
Color.red);

        List<int> roofTopFaces =
PlateauRenderingMeshUtilities.GetTrianglesOfSelectedColor(meshFilter.sharedMesh, Color.red);
        List<int> roofEdgeFaces =
PlateauRenderingMeshUtilities.GetTrianglesOfSelectedColor(meshFilter.sharedMesh, Color.blue);

        List<int> roofFaces = new List<int>();
        roofFaces.AddRange(roofTopFaces);
        roofFaces.AddRange(roofEdgeFaces);
```

（次ページに続く）

```
            // unwrap uvs for roof with planar projection
            PlateauRenderingMeshUtilities.FlattenUVsOnBoundingBox(go, meshFilter.sharedMesh,
roofTopFaces);

            //meshFilter.sharedMesh.RecalculateNormals();
            meshFilter.sharedMesh.Optimize();

            // get height of building
            float height = PlateauRenderingMeshUtilities.GetMeshHeight(meshFilter.sharedMesh);

            // set materials based on height
            SetMaterialFromHeight(meshRenderer, height);

            PlateauRenderingMeshUtilities.SetRandomAlpha(meshFilter.sharedMesh);

            PlateauRenderingBuildingUtilities.PlaceObstacleLightsOnBuildingCorners(go);
            PlateauRenderingBuildingUtilities.CreatePlaneUnderBuilding(go);
        }
```

- **LOD1モデル（LOD2モデルがあるもの）のテクスチャ生成処理：**
  以下のProcessNewLOD1というメソッドでテクスチャ生成を行う。こちらのケースではLOD2と同じ見た目のテクスチャLOD1モデルに適用することでテクスチャ生成を行う。ただし、LOD2とLOD1ではオブジェクトの形状が大きく異なるケースがあるため、LOD2をもとに簡略化したオブジェクトを新たに生成する。この新しい簡略化されたLOD2をスクリプト内ではNewLOD1としている。

```
void ProcessNewLOD1(GameObject go, MeshRenderer meshRenderer, MeshFilter meshFilter,
GameObject targetObject, GameObject referenceObject)
    {
        Undo.RecordObject(meshFilter, "Optimize Mesh");
        Undo.RecordObject(meshRenderer, "Optimize Mesh");

        // Add the Processed component to the GameObject
        Undo.AddComponent<Processed>(go);

        float bottomRatio = 0.3f;
        float topRatio = 0.8f;

        Mesh newLOD1Mesh = PlateauRenderingLOD2MeshSimplifier.LOD2Simplify(targetObject,
referenceObject, bottomRatio, topRatio);

        Bounds boundingBox = newLOD1Mesh.bounds;
        PlateauRenderingBuildingUtilities.SetBuildingVertexColorForWindow(newLOD1Mesh,
boundingBox, go);

        PlateauRenderingBuildingUtilities.ChangeLOD2BuildingShader(go);
        PlateauRenderingBuildingUtilities.PlaceObstacleLightsOnBuildingCorners(go);
        PlateauRenderingBuildingUtilities.CreatePlaneUnderBuilding(go);
    }
```

また、末尾の4行で夜間に窓や航空障害灯、底面を発光させるための処理を前述のLOD2モデルを持たないLOD1モデルの処理と同様に行っている。

LOD2の地物を簡略化してNewLOD1を生成する処理は、PlateauRenderingLOD2MeshSimplifier クラスのLOD2Simplifyメソッドを用いて行っている。実装内容は以下の通り。

```
public static Mesh LOD2Simplify(GameObject target, GameObject reference, float heightRatioBottom, float heightRatioTop)
    {

        // Flatten bottom and top faces
        FlattenFaces(target, reference, heightRatioTop, heightRatioBottom);

        // Get the mesh filter and the mesh
        MeshFilter meshFilter = reference.GetComponent<MeshFilter>();

        Mesh mesh = meshFilter.sharedMesh;

        // Get the mesh renderer
        MeshRenderer meshRenderer = reference.GetComponent<MeshRenderer>();
        if (meshRenderer == null)
        {
            return null;
        }

        // Record changes for the Undo operation
        Undo.RecordObject(mesh, "Weld Vertices");
        Undo.RecordObject(meshRenderer, "Change Material");

        // Weld the vertices
        Mesh newMesh = PlateauRenderingMeshUtilities.WeldVertices(mesh, meshRenderer, preserveNormals: false);

        meshFilter.sharedMesh = newMesh;

        // Return the new mesh
        return newMesh;
    }
```

- **LOD2モデルのテクスチャ生成処理：**
  以下のProcessLOD2というメソッドで処理を行う。LOD2モデルの場合は殆どのモデルにテクスチャがすでに存在するため、LOD1向けの処理と同様の夜間用の窓や航空障害灯、底面の発光エフェクトのみを付与する。

```
void ProcessLod2(GameObject go, MeshRenderer meshRenderer, MeshFilter meshFilter)
    {
        Undo.RecordObject(meshFilter, "Optimize Mesh");
        Undo.RecordObject(meshRenderer, "Optimize Mesh");

        // Add the Processed component to the GameObject
        Undo.AddComponent<Processed>(go);

        Bounds boundingBox = meshRenderer.bounds;
        PlateauRenderingBuildingUtilities.SetBuildingVertexColorForWindow(meshFilter.sharedMesh,
boundingBox, go);

        PlateauRenderingBuildingUtilities.ChangeLOD2BuildingShader(go);
        PlateauRenderingBuildingUtilities.PlaceObstacleLightsOnBuildingCorners(go);
        PlateauRenderingBuildingUtilities.CreatePlaneUnderBuilding(go);
    }
```

【AL007】LODグループの生成

- LODグループの生成アルゴリズムでは以下の処理を行う。それぞれについて解説する。
  1. ヒエラルキーの変更と建築物のグルーピング
  2. Activeでない地物のActive化
  3. LODコンポーネントの付与
- **ヒエラルキーの変更と建築物のグルーピング：**
  PLATEAU SDKを用いて3D都市モデルをインポートすると、デフォルトでは以下の階層でヒエラルキーに配置され、トップノード配下でLODごとにモデルがグルーピングされる。

図2-2-29　LODのグルーピング

LODグループの生成を行うため以下のような階層構造に再構成する。地物単位でグルーピングを行い、その配下にLOD別のモデルが配置されるようにする。

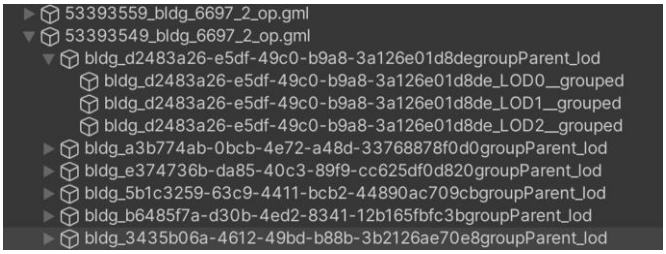


図2-2-30　グルーピング後のヒエラルキー上の階層構造

- **Activeでない地物のActive化：**
  SDKでのインポート後は、PLATEAUの最大LODのモデルのみActiveとなり、その他のLODのモデルは重複表示を避けるため非Active化されている。LODグループの生成操作のため、非Active化されているモデルを一括でActive化する。

- **LODコンポーネントの付与：**
  3D都市モデルのLOD別のモデルをUnity標準のLOD Group機能にアサインする。この際、UnityのLOD Groupでは3D都市モデルと異なり数値が小さいほうが詳細度が高いモデルとなるため、3D都市モデルのLODとは逆順でLOD Groupにモデルをアサインする。
  例えば、LOD 0～2の3D都市モデルがある場合は、Unity LOD GroupのLOD 0に3D都市モデルのLOD 2を、Unity LOD GroupのLOD 1に3D都市モデルのLOD 1を、 Unity LOD GroupのLOD 2に3D都市モデルのLOD 0を割り当てる。

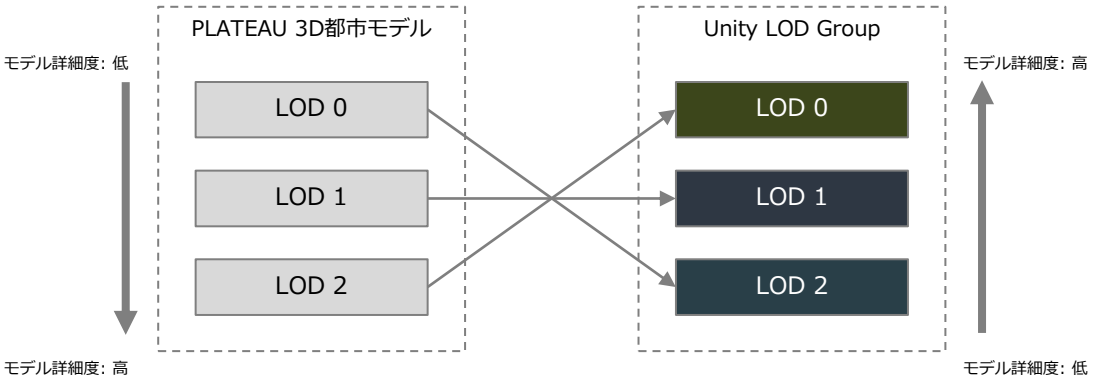


図2-2-31　LODコンポーネント付与時の対応関係

- **ヒエラルキーの変更の実装詳細：**
  インポートした3D都市モデルオブジェクトのヒエラルキー上の階層を建物単位に変更する処理のスクリプトは以下の通り。このグルーピングは【AL006】テクスチャーの生成 のものと同様のため、テクスチャー生成を行っている場合このステップはスキップされる。

```
internal void GroupObjects()
        {
            Dictionary<string, GameObject> rootObjects = new Dictionary<string, GameObject>();
            Dictionary<string, Dictionary<string, GameObject>> lodObjects = new Dictionary<string,
Dictionary<string, GameObject>>();

            PLATEAUInstancedCityModel[] plateauModelRootObject =
GameObject.FindObjectsByType<PLATEAUInstancedCityModel>(FindObjectsSortMode.None);
            List<Transform> copyOfGroupedBuildingsList = new List<Transform>();
            List<GameObject> objectsToDelete = new List<GameObject>();

            for (int i = 0; i < plateauModelRootObject.Length; i++)
            {
                if (plateauModelRootObject[i] == null)
                {
                    continue;
                }

                for (int j = 0; j < plateauModelRootObject[i].transform.childCount; j++)
                {
                    Transform gmlRoot = plateauModelRootObject[i].transform.GetChild(j);
                    float stepProgress = i / (float)plateauModelRootObject.Length;
                    if (EditorUtility.DisplayCancelableProgressBar("地物のグルーピング", "グルーピング中",
stepProgress))
                    {
                        UnityEngine.Debug.Log("LOD生成用のグルーピングがキャンセルされました。");
                        break;
                    }

                    objectsToDelete.Clear();
                    for (int k = 0; k < gmlRoot.childCount; k++)
                    {
                        Transform lodGroupedBuildings = gmlRoot.transform.GetChild(k);
                        if (lodGroupedBuildings.name.Contains("LOD0") ||
lodGroupedBuildings.name.Contains("LOD1") || lodGroupedBuildings.name.Contains("LOD2"))
                        {
                            copyOfGroupedBuildingsList.Clear();
```

（次ページに続く）

```
                    for (int l = 0; l < lodGroupedBuildings.transform.childCount; l++)
                    {
                        Transform building = lodGroupedBuildings.transform.GetChild(l);
                        copyOfGroupedBuildingsList.Add(building);
                    }

                    for (int m = 0; m < copyOfGroupedBuildingsList.Count; m++)
                    {
                        Transform buildingCopy = copyOfGroupedBuildingsList[m];
                        GroupByLodName(gmlRoot, rootObjects, lodObjects, buildingCopy.gameObject,
lodGroupedBuildings.name);
                    }
                }

                if (lodGroupedBuildings.childCount == 0
&& !PrefabUtility.IsPartOfAnyPrefab(lodGroupedBuildings.gameObject))
                {
                    objectsToDelete.Add(lodGroupedBuildings.gameObject);
                }
            }

            foreach (GameObject obj in objectsToDelete)
            {
                try
                {
                    GameObject.DestroyImmediate(obj);
                }
                catch (Exception e)
                {
                    Debug.LogException(e);
                    Debug.Log("object name: " + obj.name);
                }
            }
        }
        EditorUtility.ClearProgressBar();
        OnProcessingFinished?.Invoke();
    }
```

- **LODコンポーネントの付与の実装詳細：**
  ヒエラルキー変更後の3D都市モデルにUnityのLOD Groupコンポーネントを付与するスクリプトは以下の通り。コードの冒頭は選択されているGameObjectが建築物であるかどうかなど操作ミスを防ぐ処理で、後半でLODコンポーネント付与を行っている。

```
LODGroup lodGroup = buildingGroup.AddComponent<LODGroup>();
lodGroup.fadeMode = LODFadeMode.CrossFade;

List<Renderer> lodRenderers = new List<Renderer>();
List<LOD> lods = new List<LOD>();

// Because the loop is reversed, the highest LOD as defined by Plateau will be stored as Unity's lowest LOD
// This is assuming Plateau model will be stacked LOD0, Lod1, Lod2... etc
for (int i = buildingGroup.transform.childCount - 1; i >= 0; i--)
{
    lodRenderers.Clear();
    Transform lodBldgGroup = buildingGroup.transform.GetChild(i);
    Renderer[] renderersInChildren = lodBldgGroup.GetComponentsInChildren<Renderer>();
    lodBldgGroup.gameObject.SetActive(true);
    lodRenderers.AddRange(renderersInChildren);
    lods.Add(new LOD(PlateauRenderingConstants.k_LodDistances[i], lodRenderers.ToArray()));
}

lodGroup.SetLODs(lods.ToArray());
lodGroup.RecalculateBounds();
```

【AL008】トイカメラ

- このポストプロセスは画像にトイカメラで撮影した写真のようなぼかし効果を適用する。一般的には「ティルト・シフト」とも呼ばれる。専用のカスタムシェーダー（TiltShiftシェーダー）を用いて実装しているため、カスタムシェーダーの内容を解説する。
- **TiltShiftシェーダーのパラメーター：**
  - **BlurSize:** ブラーの範囲または強度を決定する。この値が大きいほど、ブラーはより広い範囲にわたり、より強くなる。
  - **Samples:** ブラーを適用する際にサンプリングするピクセルの数を指定する。サンプル数が多いほど、ブラーは滑らかになるが、計算コストが高くなる。
  - **BlurStartRange:** ブラーを開始する深度の範囲を指定する。この値を使用して、シーンの特定の深度範囲にのみブラーを適用することができる。
  - **BlurIterations:** ブラー処理を適用する反復回数を指定する。この値が高いほど、より強いブラー効果が得られますが、パフォーマンスへの影響も大きくなる。ブラーは、指定された回数だけ反復処理され、各反復でエフェクトが累積されていく。
- **TiltShiftシェーダーの処理内容：**
  1. **ピクセルのUV座標の取得:：**
     現在処理中のピクセルのUV座標を取得する。これは、テクスチャ上の特定のピクセル位置を特定するために必要な処理となる。
  2. **サンプリングオフセットの計算:：**
     ポアソンディスク配列からサンプリングオフセットを取得し、BlurSizeパラメーターを用いてスケーリングする。これにより、ブラーの範囲を調整できる。値が大きいほど、ブラー効果が広がる。
  3. **ピクセル値のサンプリングと加算：**
     オフセットされた位置でのピクセル値をサンプリングし、これらの値を加算していく。
     Samplesパラメーターはこのステップで重要で、サンプリングするピクセルの数を定義し、多くのサンプルを取るほどブラーが滑らかになるが、計算コストが増加する。
  4. **平均値の計算：**
     加算されたピクセル値の合計をSamplesの数で割り、平均値を求める。この平均化プロセスでブラー効果が生み出される。
  5. **結果の出力：**
     計算された平均値を現在のピクセルの色として出力する。BlurStartRangeパラメーターは、ブラーが適用される深度範囲を決定する。これにより、画像の特定の深度範囲にブラーを適用し、その他の範囲は影響を受けないようにすることができる。

  これらのプロセスにより、画像のピクセル間で色の情報が平滑化され、ソフトフォーカス効果が得られる。ポアソンディスクサンプリングは、ランダムながらも均一な分布を生成するため、自然なブラー効果が得られる。

- **バッファーの実装 （HDRP）：**
  HDRPでの実装は、カスタムのポストプロセスエフェクトを実装するために、CustomPassクラスを継承したクラスを実装する。この方法により、開発者はHDRPのレンダリングプロセスに独自のポストプロセスを組み込むことが可能となる。
  - **CustomPassを使用したバッファーの実装：**
    - **初期設定 （Setup メソッド内）：**
      - CustomPassの設定では、主にカメラのカラーバッファーに対するエフェクト適用が前提となる。ここで、ブラー効果などのポストプロセスエフェクトを適用するために必要な一時的なレンダーターゲットを確保する。
      - 2つの一時的なレンダーターゲット（例: tempRT1 と tempRT2）が確保され、これらはブラー処理の中間結果を格納するために使用される。

- **ブラー処理の適用 （Execute メソッド内）：**
  - 現在のカメラのカラーバッファーから一時的なレンダーターゲット（tempRT1）へ初期画像がコピーする。これがブラー処理の出発点となる。
  - 指定された反復回数（blurIterations）にわたってブラー処理が適用され、各反復でtempRT1とtempRT2間で画像が反復処理を行う。これにより、ブラー効果が段階的に蓄積される。
  - ブラー処理が完了した後、最終的なブラーが適用された画像（tempRT1に格納されている）が元のカメラのカラーバッファーにコピーされ、スクリーンに表示される。
- **クリーンアップ（Cleanup メソッド内）：**
  - 使用した一時的なレンダーターゲットを適切に解放し、リソースをクリーンアップすることで、メモリリークを防止する。このステップは、ポストプロセス適用後に不要になった一時バッファーの解放に必須の処理となる。

- **バッファーの実装 （URP）：**
  URPでは、ScriptableRendererFeatureを拡張し、ScriptableRenderPassを実装することでトイカメラのポストプロセス効果を適用する。
  - **CustomPassを使用したバッファーの実装：**
    - **初期設定 （OnCameraSetup メソッド内）：**
      - レンダリングデータからカメラのターゲットディスクリプタを取得し、レンダーターゲットのサイズを確定させる。
      - blurTempBufferID と blurBufferID に対応する一時レンダーターゲット（テンポラリRT）を確保する。これらはブラー処理に使用される一時バッファーとなる。
      - これらの一時バッファーは、ブラー効果の中間結果として使用され、最終的なブラー画像を生成するために相互にデータを転送を行う。
    - **実行フェーズ (Execute メソッド内)：**
      - ブラー処理に必要なパラメーター（BlurSize, Samples, BlurStartRange）をマテリアルに設定する。
      - ソース画像（source）から最初の一時バッファー（blurBuffer）へ画像を転送し、ブラー処理の基点とする。
      - 設定されたBlurIterations回数だけブラーパスを繰り返す。各繰り返しで、ブラー処理を施した画像を一時バッファー間で転送し、次のイテレーションの入力とする。
      - 最終的に、ブラー処理が完了した画像を元のソースバッファー（source）に戻す。
    - **クリーンアップ （OnCameraCleanup メソッド内）：**
      - 使用した一時的なレンダーターゲットを適切に解放し、リソースをクリーンアップすることで、メモリリークを防止する。このステップは、ポストプロセス適用後に不要になった一時バッファーの解放に必須な処理となる。

【AL009】ハーフトーン

- このポストプロセスは、ハーフトーン効果を画像に適用する。ハーフトーン効果とは、画像や映像にドットパターンを適用し、輝度や色の違いをドットの大きさや密度で表現する視覚効果のことである。プリントメディアやレトロなビジュアルスタイルを模倣する際に使用される。専用のカスタムシェーダー（Halftoneシェーダー）を用いて実装しているため、カスタムシェーダーの内容を解説する。
- **Halftoneシェーダーのパラメーター：**
  - **HalfToneSize：**ハーフトーンドットの基本サイズ。この値が大きくなるほど、生成されるドットの直径が大きくなる。
  - **HalfToneRange：**ドットサイズの変動範囲を調整する。この値を変えることで、ドットの大きさが変わる感度の調節が可能。画像の輝度に基づいてドットの大きさがどの程度変化するかを制御する。輝度が高い領域では、値に応じて大きなドットが生成され、輝度が低い領域では小さなドットが生成される。
  - **UseColor：**カラー情報の使用を選択するフラグ。0の場合はグレースケールで処理し、1の場合は元のカラーを使用。
- **Halftoneシェーダーの処理内容：**
  1. **ピクセルのUV座標の取得：**
     各ピクセルに対して、そのUV座標を取得する。これは、テクスチャ上の位置情報を得るための基本的なステップとなる。
  2. **UV座標のスケーリング：**
     UV座標をHalftoneSizeパラメーターを用いてスケーリングし、ハーフトーンドットの配置とサイズを決定する。このスケーリングにより、ドットの間隔が調整される。
  3. **色の取得と輝度の計算：**
     HalftoneBufferから色をサンプリングし、その輝度を計算する。輝度は、ドットの大きさに影響を与える要素として使用する。
  4. **ドットパターンの生成：**
     UV座標を用いて、ドットパターンを生成する。ここでは、ドットの中心からの距離に基づいてドットを描画する。HalfToneRangeパラメーターの値に基づき、輝度が高い部分では大きなドットが、低い部分では小さなドットが形成される。
  5. **カラーの適用：**
     UseColorパラメーターに基づき、ドットに元の色を適用するか、グレースケールの輝度値を使用するかを決定する。

- **ドットの形状生成について：**
  ドットの形状は、ピクセルのUV座標から計算された距離に基づく。具体的には、ドットの中心からの相対距離（d = length(uvFrac - 0.5) * 2.0）を計算し、この距離がドットの輝度によって決定された閾値内にある場合に、そのピクセルに色を塗る。距離が閾値より小さい（つまり、ドットの中心に近い）ピクセルはドットの一部として描画され、大きい場合は背景となる。これにより、円形のドットが形成される。UseColorが1の場合、このドットには元の画像の色が使用され、0の場合は輝度に基づいてグレースケールで描画される。この方法により、画像全体にわたってハーフトーン効果が作成される。

- **バッファーの実装 （HDRP）：**
  HDRPにおけるハーフトーンのポストプロセス実装では、CustomPassクラスを継承してカスタムポストプロセスを実装する。
  - **初期設定 （Setupメソッド内）：**
    - エフェクト適用に必要なマテリアルと一時的なレンダーターゲット（halftoneBuffer）を準備する。このバッファーは、ハーフトーン処理の結果を一時的に格納するために使用される。
    - halftoneBufferは、RTHandlesを用いて確保される。これにより、画像のハーフトーン処理が可能になる。

- **ハーフトーン処理の適用 （Executeメソッド内）：**
  - カメラのカラーバッファーからhalftoneBufferへ画像をコピーする。
  - 次に、ハーフトーンのポストプロセス効果を適用するため、halftoneBufferに対するハーフトーン処理を実行する。この処理により、画像上にハーフトーンパターンが生成される。
  - 処理されたhalftoneBufferの内容を再度カメラのカラーバッファーにコピーし、ハーフトーン効果が適用された最終結果の画像がスクリーンに表示される。
- **クリーンアップ（Cleanup メソッド内）：**
  - 使用したレンダーターゲットを適切に解放し、リソースを適切にクリーンアップすることでメモリリークを防止する。

- **バッファーの実装 （URP）：**
  URPでのハーフトーンエフェクト実装は、ScriptableRendererFeatureを拡張し、カスタムレンダーパス（HalftonePass）を作成する。この方法を通じて、URPのレンダリングパイプラインにカスタムポストプロセスを実装を行う。
  - **CustomPassを使用したバッファーの実装：**
    - **初期設定 （Createメソッド内）：**
      - ハーフトーンエフェクトを適用するためのカスタムレンダーパスを初期化する。このカスタムレンダーパスを介して画像処理を行う。
    - **ハーフトーン処理の適用 （ExecuteメソッドHalftonePassクラス）：**
      - ハーフトーン処理を行う一時バッファー（halftoneBuffer）を用意し、カメラのカラーバッファーをコピーする。
      - halftoneBufferに対してハーフトーン効果を適用し、その結果を元の画像バッファー（スクリーン）にコピーする。この処理により、スクリーンにハーフトーン効果が適用された最終結果の画像が表示される。
    - **クリーンアップ （OnCameraCleanup メソッド内）：**
      - ポストプロセス適用後に使用した一時バッファーを解放し、リソースを適切にクリーンアップすることでメモリリークを防止する。

【AL010】ナイトビジョン

- このポストプロセスは、画像にナイトビジョン（暗視）効果を適用するものである。ナイトビジョン効果は、低光量の環境下でも視認可能な画像を生成するために使用される。この効果は、特定の色調（通常は緑色）を使用して、低照度下での視認性を向上させる。専用のカスタムシェーダー（NightVisionシェーダー）を用いて実装しているため、カスタムシェーダーの内容を解説する。
- **NightVisionシェーダーのパラメーター：**
  - **Sensitivity:** 暗視効果の感度。この値が高いほど、画像の輝度が強調される。
  - **Color:** ナイトビジョン効果に使用される色。通常は緑が使用されるが、任意の色で設定可能。
- **NightVisionシェーダーの処理内容：**
  1. **ピクセルのUV座標の取得：**
     各ピクセルのUV座標を取得し、テクスチャ上の対応する位置を特定する。
  2. **輝度の計算：**
     テクスチャから色をサンプリングし、その輝度を計算する。輝度は、ピクセルの明るさを決定するために使用される。
  3. **色の適用：**
     設定されたColorに基づき、計算された輝度に色を適用する。輝度が高いピクセルほど色が強く表示される。
  4. **感度の調整：**
     Sensitivityパラメーターに基づき、輝度の強調を調整する。感度が高いほど、より多くのピクセルが強調表示される。

- **バッファーの実装 （HDRP）：**
  HDRPでは、CustomPassクラスを継承してナイトビジョンエフェクトを実装する。エフェクトの適用には、一時的なレンダーターゲットnightVisionBufferを使用する。
  - **初期設定：**
    - エフェクトの適用に必要なマテリアルとnightVisionBufferを準備する。このバッファーはエフェクト処理の結果を格納するために使用される。
  - **エフェクトの適用：**
    - カメラのカラーバッファーからnightVisionBufferへ画像をコピーし、ナイトビジョンエフェクトを適用する。その後、処理された画像をカメラのカラーバッファーに戻す。
  - **クリーンアップ：**
    - ポストプロセス適用後に使用した一時バッファーを解放し、リソースを適切にクリーンアップすることでメモリリークを防止する。

- **バッファーの実装 （URP）：**
  URPでは、ScriptableRendererFeatureを拡張し、ScriptableRenderPassを実装することでナイトビジョンエフェクトを適用する。
  - **初期設定：**
    - ナイトビジョンエフェクトの適用に必要なカスタムレンダーパスを初期化し、一時バッファーnightVisionBufferを準備する。
  - **エフェクトの適用：**
    - カメラのカラーバッファーをnightVisionBufferにコピーし、ナイトビジョンエフェクトを適用する。その後、処理された画像をスクリーンに戻す。
  - **クリーンアップ：**
    - ポストプロセス適用後に使用した一時バッファーを解放し、リソースを適切にクリーンアップすることでメモリリークを防止する。

【AL011】頂点カラー設定

- 頂点カラーの調整機能では、ユーザーがエディタのGUIを通じて地物のメッシュの頂点カラーの調整を行う機能を提供する。この機能は、特に建物の窓用頂点カラーマスク（Gチャンネル）の調整を行い、各地物にランダムな頂点アルファ値を割り当てる。ユーザーが「頂点カラーの調整」ボタンを押下すると、選択された地物のメッシュに対して指定されたパラメーターに基づく頂点カラーの調整が行われる。
- **UI入力値から取得するパラメーター：**
  - **地物上部からX%マスク （頂点カラーG） MaskPercentage：**この値は、地物の上部からどの範囲まで頂点カラーGチャンネルのマスクを適用するかを0から100%の範囲で指定する。このマスクは、建物の上部、天井等に窓が表示されないようにするために使用される。
  - **ランダムシード値 （頂点カラーA） RandomAlphaSeed：**この値は、頂点カラーAチャンネルの値のランダム化プロセスにおける基点となるランダムシード値を設定する。ランダムシード値により、地物ごとに一貫性のあるが異なるランダムアルファ値が設定される。

- **アルゴリズムの処理内容：**
  - **頂点カラーGチャンネルのマスク適用処理：**
    Gチャンネルのマスク適用処理は、建物のメッシュに対して緑色の頂点カラーマスクを部分的に適用し、特定の高さ以上に窓が表示されないようにすることを目的としている。この処理は以下のステップで実行される。
    1. **頂点の高さ判定：**メッシュの各頂点に対して、そのY座標（高さ）を計算し、メッシュ全体の最小Y座標と最大Y座標を決定する。このY座標は、頂点がメッシュ内でどの位置にあるかを示す。
    2. **マスク適用範囲の計算：** 入力パラメーター MaskPercentageをもとに、メッシュのどの高さ範囲にGチャンネルのマスクを適用するかを決定する。具体的には、メッシュの最大Y座標から計算された範囲内にある頂点にのみマスクが適用される。
    3. **Gチャンネル調整：**マスク適用範囲内の頂点に対して、Gチャンネルの値を調整する。この調整により、指定された範囲以上の高さにある頂点にはマスクが適用されず、下部にある頂点にはマスクが適用される。
  - **頂点カラーAチャンネルのランダム化処理：**
    Aチャンネルのランダム化処理は、地物単位のメッシュでランダムな値のバリエーションを持たせることを目的としている。この処理は以下のステップで実行される。
    1. **ランダムシードの設定：**入力パラメーターRandomAlphaSeedをもとに、ランダム数生成器のシード値を初期化する。このシード値により、後続のランダム数生成プロセスの一貫性が保証される。
    2. **ランダムアルファ値の生成：**シード値をもとに、0から1の範囲でランダムなアルファ値を生成する。このランダムアルファ値は、メッシュの全頂点に一貫して適用され、メッシュ全体に均一な頂点カラーの値を付与する。
    3. 処理が複数の地物のメッシュに対して行われる際、内部のcurrentSeedの値をインクリメントすることで、次のオブジェクトに対しては異なるランダムアルファ値が生成され、地物ごとに異なる値が適用される。

- **シェーダーとの連携:**
  Rendering Toolkitが提供する地物用のBuildingカスタムシェーダーは、メッシュの頂点カラーのチャンネル情報を参照していくつかの要素を描画する機能を持つ。これらの処理は、主にUnityのシェーダーグラフを使用し、ノードベースのビジュアルプログラミング環境で実装されている。
  以下に示すサンプルコードは、シェーダーグラフでの実装をテキストベースで表現した疑似コードであり、この疑似コードを通じて、シェーダーの挙動と頂点カラー情報の利用方法を理解しやすくすることを目的とする。

- **頂点カラーGチャンネルの使用例:**
  地物用カスタムシェーダーでは、Gチャンネルの値を閾値として、地物の窓を描画するために利用される。Gチャンネルの値が閾値を超える部分は窓としてレンダリングされ、そうでない部分は窓以外の領域として扱われる。

  窓の描画を頂点カラーGでマスクするサンプルコード：

```
WindowMask = step(MaskThreshold, VertexColorG);
WindowColor *= WindowMask;
```

  - **MaskThreshold：**Gチャンネルのマスクを適用する閾値。この値よりもGチャンネルの値が大きい場合、窓が存在する領域とみなされる。
  - **VertexColorG：**頂点カラーのGチャンネル値。
  - **WindowMask：**step関数によって計算されるマスク値。MaskThresholdよりVertexColorGの値が大きい場合は1、そうでない場合は0となる。
  - **WindowColor：**最終的にマスクに基づいて調整される窓の色。WindowMaskが0の場合、窓が描画されない。WindowMaskが1の場合、窓が描画される。

- **頂点カラーAチャンネルの使用例：**
  頂点カラーのAチャンネルは、地物ごとに異なるビジュアル効果を生成するために利用される。夜間のシーンにおける建物に対し、頂点のAチャンネル値を用いて色温度の異なる色を地物に適用することで、建物は一つ一つ異なった色温度の光を発しているように見える。

  頂点カラーAを用いた地物ごとの色の適用サンプルコード：

```
NightBuildingColor += lerp(CoolColor, WarmColor, VertexColorA) * IntensityFactor;
```

  - **CoolColorとWarmColor：**冷たい色（例：青色）と温かい色（例：オレンジ色）が設定され、これらは色温度の差を表現するために使用される。
  - **VertexColor：**頂点カラーのAチャンネル値により、CoolColorからWarmColorへのグラデーション内でランダムに色を選択する基準となる。頂点カラーAは0から1でグラデーションにマッピングされ、この範囲内で色の選択が行われる。
  - **IntensityFactor:：**適用強度を調整する係数であり、ビジュアル効果の強度を制御する。

【AL012】画素調整機能

- 画素調整機能では、【FN014】画素調整機能に記載したとおり、ハイパスフィルター、輝度フィルター、コントラストフィルター、シャープネスフィルターの4種類のフィルタを選択した地物に適用する。各フィルターの具体的な処理内容を以下に記載する。

- **ハイパスフィルター：**
  ハイパスフィルターは高周波成分を強調して細かいディテールやエッジを際立たせる効果がある。具体的には、画像の中間色調を中和しつつ、細部やエッジの強調を行う。この処理を通じて、元のテクスチャに含まれる不要な影や光の影響を簡易的に除去することができる。
  - **パラメーター：Radius**
    ガウスぼかしの範囲（半径）を指定する。大きい値ほどぼかし効果が広がり低周波成分が除去される。Radiusが大きいほど、画像はより滑らかになり、細かいディテールが除去される。このプロセスにより、エッジと細部が強調される。
    - UI範囲: 0から100
    - 実際の処理値範囲: 直接使用
  - **フィルター処理の実装**
    ガウスぼかしを適用後、元の画像からぼかし画像を引くことで高周波成分を抽出。

    ```
    Output = Original − GaussianBlur(Original, Radius) + 0.5;
    ```

- **輝度フィルター：**
  輝度フィルターは画像全体の明るさを一律に調整する。このフィルターにより、画像が明るくまたは暗くなり、ディテールがはっきりと見えるようになる場合がある。
  - **パラメーター：BrightnessFactor**
    画像の各ピクセルのRGB値に一定の値を加える（または減らす）ことで明るさを調整する。
    - UI範囲: -100から100
    - 実際の処理値範囲: -0.999から0.9にリマップ
  - **フィルター処理の実装**
    画像の各ピクセルのRGB値に一定の値を加える（または減らす）ことで明るさを調整する。
    画像を明るくする場合: （BrightnessFactorが正の値）

    ```
    Output = Original + BrightnessFactor;
    ```

    画像を暗くする場合: （BrightnessFactorが正の値）

    ```
    Output = Original * (1.0 + BrightnessFactor) ;
    ```

- **コントラストフィルター：**
  コントラストフィルターは画像の明暗差を調整し、深みや立体感を強調する。高いコントラストは明るい部分と暗い部分の差をはっきりさせ、画像を鮮明に見せる効果がある。逆にコントラストを下げると、画像の明暗差が緩和され、柔らかい印象になる。
  - **パラメーター: ContrastFactor**
    コントラスト係数は画像のコントラストレベルを調整。1より大きい値でコントラストが増し、1より小さい値で減る。ContrastFactorが高いほどコントラストが増し、画像の明暗差がはっきりする。
    - UI範囲: -100から100
    - 実際の処理値範囲: 0.1から2.5にリマップ
  - **フィルター処理の実装**
    中間色からの差分にコントラスト係数を乗じ、再び中間色に加算することでコントラストを調整。

```
Output = 0.5 + (Original − 0.5) × ContrastFactor;
```

- **シャープネスフィルター：**
  シャープネスフィルターは画像のエッジを強調し、全体の鮮明さを向上させる。エッジの明瞭化により、画像のディテールが強調される。
  - **パラメータ: SharpnessFactor**
    シャープネス係数はエッジ強調の程度を決定。値が大きいほどエッジが強調され、ディテールが鮮明になる。
    - UI範囲: 0から100
    - 実際の処理値範囲: 0.0から1.0にリマップ
  - **フィルター処理の実装**
    ラプラシアンフィルターを使ってエッジ検出を行い、その結果にSharpnessFactorを乗じて元の画像に加算することでシャープネスを向上させる。SharpnessFactorが高いほど、エッジの強調が強くなり、画像のディテールが鮮明になる。

```
Output = Original + SharpnessFactor × Laplacian;
```

- **4種のフィルターの処理：**
  Rendering Toolkitの画素調整アルゴリズムでは、上記のフィルタを①ハイパスフィルター→②コントラストフィルター→③輝度フィルター→④シャープネスフィルターの順で適用する。フィルター値が0に設定されている場合はそのフィルターをバイパスする。

【AL013】解像度変更

- Rendering Toolkitでテクスチャの解像度を変更する際には、TextureScalerクラスのResizeメソッドで処理を行う。具体的な実装は以下の通り。
  変更後のテクスチャの解像度をtargetX、targetYで指定する。現在のテクスチャを新しいテクスチャにコピーする処理は Graphics.Blit(texture2D, current); で行っている。

```
public static Texture2D Resize(Texture2D texture2D, int targetX, int targetY)
{
  RenderTexture previous = RenderTexture.active;
  RenderTexture current = new RenderTexture(targetX, targetY, 24);
  RenderTexture.active = current;
  Graphics.Blit(texture2D, current);
  Texture2D result = new Texture2D(targetX, targetY, TextureFormat.ARGB32, true);
  result.ReadPixels(new Rect(0, 0, targetX, targetY), 0, 0);
  result.Compress(true);
  result.Apply(false);
  RenderTexture.active = previous;
  current.Release();

  return result;
}
```

- Graphics.BlitはUnity Scripting APIで提供されているGraphicsクラス内のメソッドである。詳細は以下の公式ドキュメントを参照されたい。
  https://docs.unity3D.com/2021.3/Documentation/ScriptReference/Graphics.Blit.html

- テクスチャをリサイズする処理もGraphics.Blitメソッドで行っている。内部的にはバイリアフィルタリングのアルゴリズムを使用している。次頁にバイリニアフィルタリングの概要を示す。

- **バイリニアフィルタリング：**
  バイリニアフィルタリングは、テクスチャを拡大または縮小するときにテクスチャを滑らかにするための手法である。バイリニアフィルタリングでは、水平と垂直の二方向で線形補間を行う。

- **バイリニアフィルタリングの処理内容：**
  画面上のピクセルがテクスチャ内の4つのテクセルの間の位置に対応する場合、バイリニアフィルタリングはこれら4つのテクセルの加重平均を取り、最終的な色を決定する。
  1. 水平補間
     - AとBの間: 色1 = A * (1-x) + B * x
     - CとDの間: 色2 = C * (1-x) + D * x
  2. 垂直補間
     - 色1と色2の間: P = 色1 * (1-y) + 色2 * y

  点(x,y)の色を計算するための一般的な計算式に整理すると以下のようになる。

$$f(x,y) \approx \frac{1}{(x_2-x_1)(y_2-y_1)} \begin{bmatrix} f(Q_{11}) & f(Q_{12}) & f(Q_{21}) & f(Q_{22}) \end{bmatrix} \begin{bmatrix} x_2y_2 & -y_2 & -x_2 & 1 \\ -x_2y_1 & y_1 & x_2 & -1 \\ -x_1y_2 & y_2 & x_1 & -1 \\ x_1y_1 & -y_1 & -x_1 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ x \\ y \\ xy \end{bmatrix}.$$

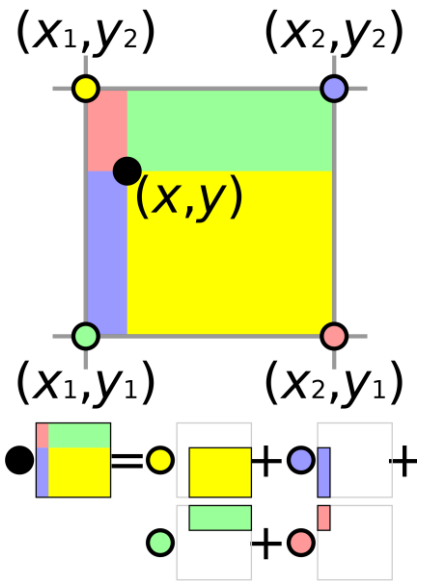


図2-2-32　バイリニアフィルタリングの計算イメージ

2.4.2 Sandbox Toolkitの使用アルゴリズム

【AL014】トラック作成

- Sandbox Toolkitのトラックは Unity 公式パッケージのスプライン （com.unity.splines） をもとに実装されており、スプライン作成ツールと同じ方法を利用してトラックが作成される。独自実装のアルゴリズムとして、Sandbox Toolkit向けに分岐のあるスプライン上での位置管理ロジックを実装した。
- **分岐のあるスプラインの補間値による位置管理**
  スプラインでは補間値 （interpolation value）を用いてスプライン上の位置を制御する。スプラインの長さに関わらず、始点の補間値を0、終点の補間値を1として位置を指定することができる。分岐がある場合、以下の図のようにスプラインは複数のスプラインによって構成されているため、スプラインのインデックスと補間値を指定することで分岐のあるスプラインの任意の位置を指定することができる。
  例えば、図の点Pがインデックス0のスプラインの全体の長さの80%の位置にある場合、インデックスの0と補間値の0.8という情報から位置を特定することができる。

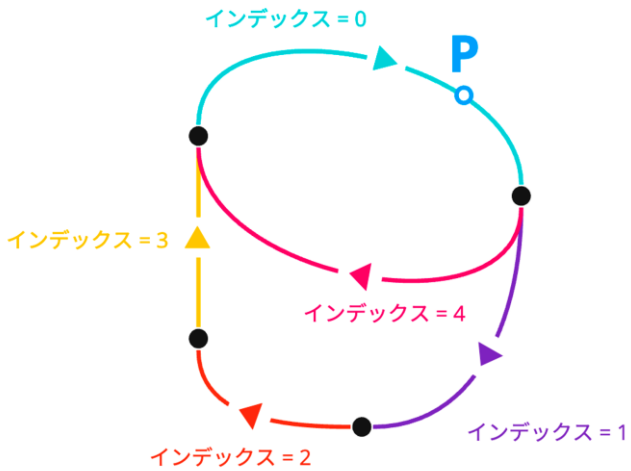


図2-2-33　分岐トラックの位置管理

【AL015】トラック移動コンポーネント

- トラック上で乗り物やアバターを動かすためのコンポーネントとしてトラック移動コンポーネント（PlateauSandboxTrackMovement）を実装した。利用の際には移動させたいオブジェクトにトラック移動コンポーネントをアタッチする。

- **移動処理:**
  トラック移動コンポーネントはアタッチされたオブジェクトの IPlateauSandboxMovingObject インタフェースを経由して移動処理を実現する。乗り物とアバターがこのインタフェースを実装しており、以下のメソッドを持つ。
  - GetVelocityRatio(float lookAheadAngle)
    - lookAheadAngle：移動先読み角度。現在の位置から衝突検知点への角度を意味する。
    - このメソッドは移動先読み角度を考慮した現在の速度の係数を定義する。ある点でのオブジェクトの移動方向と移動先読み角度が大きい場合、速度を落とすことで自然な移動アニメーションを実現する。
  - OnMoveBegin()
    - 移動を開始する際に呼び出されるメソッド。移動開始時に必要な初期処理を行う。
  - OnMove(in MovementInfo movementInfo, PlateauSandboxTrack track)
    - movementInfo：現在フレームでの移動に関する以下の情報を保持する。
      - 現在速度
      - フレーム内移動差分
      - トラック内の場所を示す補間値
      - 移動先読み角度
      - 第二軸のワールド座標:
        第二軸は乗り物など、複数の軸をトラックに沿わせる必要のある移動対象で利用する。例えば車の場合、前輪と後輪の中心をトラックに沿わせるように動作させることで自然な動作を実現することができるため、後輪の中心を移動させるオブジェクトの位置、前輪の中心を第二軸として計算している。
      - 第二軸のトラック上角度
    - track：移動しているトラックの参照。
    - 移動中のイベントを定義するメソッド。トラック移動コンポーネントはこのインタフェースを呼び出すだけなので、実際のオブジェクトの位置はそれぞれインタフェースを実装するクラスごとに定義している。与えられたトラックの参照や移動情報をもとにこのフレームでの位置を更新する。
  - OnMoveEnd()
    - 移動を終了する際に呼び出されるメソッド。移動終了時に必要な終了処理を実装している。

【AL016】ランダムウォーク

- 移動オブジェクトが衝突を避けながら滑らかに移動速度を変化させる処理や、分岐があるトラック上でランダムな分岐先の選択処理を実現するためのアルゴリズム。処理の概略は以下のとおり。
  - 後述するトラックのランダムウォークパスイテレーターを利用し、移動対象のオブジェクトの IPlateauSandboxMovingObject の各イベントを呼び出す。
  - OnMove の毎フレーム呼び出しでは移動情報 （MovementInfo）や衝突判定（後述）の計算などを行う。
  - IEnumerator RandomWalkEnumerator をコルーチンとして起動することでオブジェクトの移動を更新し、コルーチンはトラック移動コンポーネント内で管理される。移動を強制終了する場合などはこのコルーチンを停止することで実現する。

- **ランダムウォークパスイテレーター：**
  TrackPathIterator クラスで実現。スプラインが分岐を持たない場合、補間値を0から1へ線形補間することで簡単に移動アニメーションを実現できるが、分岐がある場合の処理は複雑になる。そのため、ランダムに分岐先を選択するランダムウォークを実現するための機能をランダムウォークパスイテレーターとして提供している。
  - public bool MovePoint(float delta, out float t)
    移動距離を指定することで、ランダムウォークパス内の次の補間値を取得する。終点のあるトラックで終点に到達した場合は返り値として false が返され、それ以外の場合は true が返される。各移動オブジェクトの実装では、取得した補間値をもとにトラックの機能でワールド座標に変換することでゲームオブジェクトの位置を設定することができる。
  - public TrackPathIterator Clone()
    ランダムウォークパスイテレーターのコピーを作成する。ランダムウォークでは分岐先をランダムに決定するため、シード値やパスの進捗状況などを共有するために必要となる機能である。主に、衝突点と第二軸の場所を決定するために使用される。

- **移動速度の管理：**
  移動速度はトラック移動コンポーネントで管理されている。移動速度は現在の速度（初期値は0）からプロパティに設定された最大速度になるまで時間経過で現在速度から線形補間される。そのため、移動オブジェクトは即時に最大速度にはならず、徐々に速度が変化する。移動速度はコンポーネント内のフィールドに保持される。
  ランダムウォークパスイテレーターの MovePoint() には現在速度と Time.deltaTime をもとに移動フレーム内での移動距離に変換したものを入力する。

- **衝突判定：**
  トラック移動コンポーネントに設定された衝突判定のプロパティをもとに移動オブジェクトが他のコライダーを持つオブジェクトに衝突する可能性を計算する。
  衝突判定は判定する位置に球体レイキャストを行うことで実現される。球体レイキャストではレイキャストの始点から終点へ球体を飛ばし、他のコライダーに衝突するポイントを検知する。球体レイキャストが判定された場合、移動オブジェクトが別のオブジェクトに衝突する可能性があるため、衝突判定が続く間は最大速度を0に設定し、線形補間の速度を2倍にする。

【AL017】オブジェクト配置

- オブジェクト配置ツールは EditorTool API を用いて実装されている。

- **配置に関する設定情報：**
  Sandbox Toolkit 内のステートは Sandbox コンテキスト （PlateauSandboxContext）を介して共有される。使用されている Sandbox コンテキストは PlateauSandboxContext.GetCurrent から参照を取得することができる。
  - シーンに存在する全てのトラックのリスト
  - 配置するオブジェクトの参照（プレハブの参照）
  - 配置設定
    - 配置する場所
      - コライダーの表面に配置
      - トラック上に配置
    - 配置するときの上向きのベクトル
      - コライダーもしくはトラックの法線に従う
      - 常にワールド座標の上に向かせる
    - 配置モード
      - クリック
      - ブラシ
    - ブラシ配置設定
      - 一回で配置するオブジェクトの数
      - 配置する半径
      - 配置するオブジェクトの平面方向の回転

- **シーンビューに表示されるオーバーレイUI：**
  EditorTool では特定の編集モードに対してシーンビューにオーバーレイUIを実装することができるため、配置方法の調整などはこのオーバーレイの機能を利用して設定ウィンドウを表示している。
  オーバーレイUIは UI Toolkit を用いて実装した。UI Toolkit ではUIのレイアウトを UXML と呼ばれるマークマップ言語で定義し、C#スクリプトからボタンなどの管理を行う。
  オブジェクト配置ツールのオーバーレイUIのUXMLファイルは
  “com.unity.plateautoolkit/PlateauToolkit.Sandbox/Editor/UI/PlacementToolOverlay.uxml”
  に配置されている。

  オーバーレイを管理するための処理は UnityEditor.Overlays.Overlay を継承した、以下のクラスに実装されている。このクラスでオブジェクト配置ツールが起動している際のオーバーレイUIを管理している。

```
[Icon("UnityEditor.InspectorWindow")]
[Overlay(typeof(SceneView), "plateau-sandbox-placement-tool", "PLATEAU 配置ツール",
"PlateauPlacementInspector")]
sealed class PlateauSandboxPlacementToolOverlay : Overlay, ITransientOverlay
{
}
```

オブジェクト配置ツールが起動すると CreatePanelContent が呼び出されるため、このメソッドの中で UXML ファイルを読み込み、ビューのインスタンスが生成される。生成したインスタンスからボタンやプルダウンなどオーバーレイに実装したUIコンポーネントを取得し、ボタン、プルダウンやチェックボックスのイベント管理を行う。
オブジェクト配置ツールのオーバーレイでは、UIから設定された値を Sandbox Toolkit 内で共有される PlateauSandboxContext を介してオーバーレイ以外の部分で同期している。ここでは、配置に関する情報を取りまとめる PlateauSandboxContext.PlacementSettings に設定内容を格納している。反対に、オーバーレイUIの初期化時は PlateauSandboxContext.PlacementSettings の値を初期値として設定している。

- **配置の実装:**
  オブジェクトを配置するための処理は PlateauSandboxPlacementTool が起点となって実行される。オブジェクト配置ツールが起動している間、public override void OnToolGUI（EditorWindow window）が更新処理として Unityエディターから呼び出されるため、ここにオブジェクトをシーン上に配置する処理を実装している。ただし、配置には「クリック配置」と「ブラシ配置」の2種類があり、それらは IPlacement インタフェースとして抽象化されているため、より具体的な実装はそれを実装するクラスに定義されている。PlateauSandboxPlacementTool が IPlacement を m_Placement フィールドに保持しており、以下のような実装で配置の具体的な実装を呼び出している（このスクリプトは説明用に簡略化したサンプルスクリプトである）。

```
public override void OnToolGUI(EditorWindow window)
{
    switch (Event.current.GetTypeForControl(controlId))
    {
        case EventType.MouseLeaveWindow:
            m_Placement.Dispose();
            break;
        case EventType.MouseMove:
            m_Placement.MouseMove(window);
            break;
        case EventType.Repaint:
            m_Placement.Repaint(window);
            break;
        case EventType.MouseDown:
            m_Placement.MouseDown(window);
            break;
        case EventType.MouseDrag:
            m_Placement.MouseDrag(window);
            break;
        case EventType.MouseUp:
            m_Placement.MouseUp(window);
            break;
    }
}
```

クリック配置とブラシ配置の具体的な実装クラスは以下のように内部クラスとして定義されている。

```
partial class PlateauSandboxPlacementTool
{
    class ClickPlacement : IPlacement
    {
    }
}
```

```
partial class PlateauSandboxPlacementTool
{
    class BrushPlacement : IPlacement
    {
    }
}
```

- **配置のインタフェース （ IPlacement ）:**
  前述のとおり、具体的な配置処理は IPlacement インタフェースによって定義される。

```
interface IPlacement : IDisposable
{
    public bool IsPlaceable { get; }

    void Repaint(EditorWindow window);

    void MouseMove(EditorWindow window);

    void MouseDown(EditorWindow window);

    void MouseUp(EditorWindow window);

    void MouseDrag(EditorWindow window);
}
```

このインタフェースは主に PlateauSandboxPlacementTool の OnToolGUI から呼び出される。Unity エディター内の操作イベントは Event.current から取得できるため、 Event.current.type から操作の種類を特定し、対応したイベントを呼び出す。

Repaint	描画の更新処理
MouseMove	マウスが動いたときの処理
MouseDown	マウスをクリックしたときの処理
MouseUp	マウスのクリックが離されたときの処理
MouseDrag	マウスをクリックした状態でマウスを動かしたときの処理

IsPlaceable は使用している配置モードが配置可能かどうかを判定するために使用する。主に、コライダー上に配置する場合にコライダーがない場合や、トラック上に配置する際にマウスカーソルの近くにトラックがないケースをハンドリングするために使用する。

- **クリック配置の実装 （ ClickPlacement ）**
  - フィールド
    - 配置する位置はPlacePoint クラスを用いて m_PlacePoint フィールドに保持される
      - 配置する位置
      - 配置する位置の法線ベクトル
      - トラックの参照（トラック上に配置するときのみ）
      - 別のSandboxオブジェクトにレイキャストが当たっているかどうか（コライダー上に配置するときのみ）
    - オブジェクトを配置する向きは m_HandleDirectionVector フィールドに保持される
      - 向きはドラッグで決定するため、 MouseDrag メソッドで制御される
  - MouseMove
    - コライダー上に配置する場合
      - マウスカーソルの位置にレイキャストを行い、コライダーに当たった場所を m_PlacePoint に保持する。このとき、レイキャストが別の Sandbox のオブジェクトに当たったかどうかの判定結果も保持する
    - トラック上に配置する場合
      - マウスカーソルの近くにあるトラックの中点において、最も近い場所を m_PlacePoint として保持する
  - Repaint
    - m_PlacePoint に値を設定。配置するオブジェクトが選択されている場合に描画処理を行う
    - 配置するオブジェクトのプレビューが生成されていない場合はプレビュー用オブジェクトを生成する
      - プレビュー用オブジェクトは m_PreviewInstantiation フィールドに保持される。
      - プレビュー用オブジェクトが生成されているとき、そのオブジェクトが配置用に選択されているオブジェクトと異なる場合は選択されているオブジェクトに変更される（破棄して再生成）
      - プレビュー用オブジェクトは配置時の見た目の分かりやすさのためだけでなく、別のオブジェクトとの重なりを判定をするためにも使用される。この判定には PlateauSandboxPlacementCollision コンポーネントを使用する。このコンポーネントは Unity 標準の OnCollision イベント結果を外部から取得するためのものである。これに加えて、衝突判定をするための Rigidbody などの設定もプレビュー生成時に行う
    - 衝突判定の物理演算用の更新処理
      - Unity エディターがプレイモードではないとき、物理演算の計算は自動では行われない。オブジェクト配置ツールでは重なりの判定などを Rigidbody 及びコライダーを用いて行っているため、物理演算の計算を実行するために Physics.Simulate を呼び出している
    - 配置するオブジェクトの向きなどを示すカーソルを表示
      - カーソルは Handles を用いて描画している
      - m_PlacePoint と m_HandleDirectionVector の情報を利用する
    - MouseDown
      - マウスがクリック開始された場所を m_MouseDownPosition に保持
    - MouseDrag
      - m_MouseDownPosition からドラッグした位置への差異を利用し、m_HandleDirectionVector を計算
      - 計算の処理が終わったら、カーソルの表示を更新用に window.Repaint を呼び出す

    - MouseUp
        - オブジェクトの配置を実行する
        - m_PlacePoint に設定されている場所にオブジェクトを生成する
        - トラック上に配置する、かつオブジェクトが移動可能なオブジェクトの場合はトラック移動コンポーネントを自動でアタッチする
        - 配置をやり直し（Ctrl + Z / ⌘ + Z）できるように、Undo にオブジェクトの生成を記録する
        - m_PlacePoint と m_MouseDownPosition に保持していた値をリセットする（ null を代入する ）

- **ブラシ配置の実装 （ BrushPlacement ）**
    - フィールド
        - 配置する位置はクリック配置と同様に PlacePoint クラスを用いて m_PlacePoint フィールドに保持される
    - ブラシ配置における配置するオブジェクトの位置の決定方法
        - BrushSettings.GetRandomOffsets を用いると、設定されている配置オブジェクト数分だけ配置するオブジェクトのオフセットのリストを取得することができる
        - オフセットは0から設定されている配置半径の距離まででランダムに決定される
        - このメソッドは RandomizeShapeSeed でランダムシード値を更新するまで同じオフセットのリストが返却される。一方で、ランダムシード値を更新すると異なるオフセットのリストが返却される
    - Placeメソッド
        - BrushSettings.GetRandomOffsets でオフセットのリストを取得し、 m_PlacePoint を中心に繰り返し処理でそれぞれのオフセットを用いてオブジェクトを生成する。各生成に関してはクリック配置と同じ処理を行う
    - Repaint
        - BrushSettings.GetRandomOffsets でオフセットのリストを取得し、 m_PlacePoint を中心に繰り返し処理でそれぞれのオフセットを用いてカーソルを表示する
          ※ブラシ配置ではカーソルの向きは BrushSettings.ForwardYAxis に従う
    - MouseMove
        - クリック配置と同様
    - MouseDown
        - ブラシ配置では次々と連続で複数のオブジェクトを配置できるため、Sandbox Toolkit では「クリック、ドラッグ、クリックを離す」までを一回の配置として Undo の履歴に追加している。そのため、 MouseDown では Undo のグループを作成し、そのグループの識別インデックスをフィールドに保持する
        - ブラシ配置ではクリック配置と違ってクリックした瞬間に1回目の配置が開始され、そこからドラッグすることで次々にオブジェクトが配置されていく処理のほうが自然なため、このイベントではまず一回目の配置（ Place メソッドの呼び出し ）を行う
    - MouseUp
        - 配置処理はクリック開始時からドラッグ中に完了しているため、MouseUpでは配置に関する処理は何も行わない
        - このタイミングでブラシ配置における一回の配置が完了するため、保持している Undo の識別インデックスを一連の操作として Undo に登録する

【AL018】カメラマネージャー

- Sandbox Toolkit のカメラマネージャー（ PlateauSandboxCameraManager ）では、視点として選択しているオブジェクトに合わせて m_SubCamera にセットされたカメラの位置（ m_CameraPivot ）を変更することで視点切り替えを実現している。従って、複数の視点として利用可能なオブジェクトを用意したとしても常にカメラは一つのみとなっている。これにより、効率的に視点を管理することができる。

- 視点となりうるオブジェクトは IPlateauSandboxCameraTarget インタフェースを実装したクラスをアタッチする必要があり、乗り物とアバターのコンポーネントがこのインタフェースを実装している。PlateauSandboxCameraSettings は以下の設定項目を定義している。

m_HorizontalSpeed	一人称視点の横方向のマウス感度
m_VerticalSpeed	一人称視点の縦方向のマウス感度
m_HorizontalDragSpeed	三人称視点の横方向のドラッグマウス感度
m_VerticalDragSpeed	三人称視点の縦方向のドラッグマウス感度
m_ZoomSpeed	三人称中心視点のマウスホイールによるズーム感度
m_MinOffset	三人称中心視点のマウスホイールによる最小ズーム距離
m_MaxOffset	三人称中心視点のマウスホイールによる最大ズーム距離

- **視点の対象 (IPlateauSandboxCameraTarget):**
  カメラの対象の実装には以下の情報をインタフェースを介して提供する必要がある。
  - カメラが利用可能かどうか
    - 視点の対象となるオブジェクトの状態によって視点の利用可否を定義することができる
    - 特定の条件下で視点として利用したくない場合はその条件下のときにこのプロパティの戻り値が false になるように実装する
  - オブジェクトの回転
    - 一部の視点モードでカメラの対象が向いている方向が必要になる場合があるため、回転の情報を提供する必要がある
    - 基本的にはそのゲームオブジェクトの transform.rotation を返却すれば問題なく、Sandbox Toolkit で実装している乗り物とアバターのコンポーネントではそのように実装されている
  - 視点設定
    - 視点に関する詳細な設定を定義する必要がある
    - PlateauSandboxCameraTargetSettings クラスによって定義する
    - PlateauSandboxCameraTargetSettings は Serializable 属性を付与しているため、[SerializeField] として実装コンポーネント内に定義することでインスペクターから調整可能な設定として定義できる

m_FirstPersonViewPosition	（一人称視点）視点にするオブジェクト原点からの相対位置
m_ThirdPersonViewDefaultCameraPosition	（三人称視点）視点の初期位置にするオブジェクト原点からの相対位置
m_ThirdPersonOrbitInitialRotation	（三人称中心視点）カメラの初期回転。オブジェクトのZ軸方向と逆ベクトル上にある場合を0とし、Y軸を中心とした回転の角度を0から360で設定します。
m_ThirdPersonOrbitOffset	（三人称中心視点）注視点にするオブジェクト原点からの相対位置
m_ThirdPersonOrbitDefaultDistance	（三人称中心視点）注視点からのデフォルトの距離（距離はマウスホイールで操作可能）

- **カメラの構成:**
  カメラマネージャーでは、カメラの位置を直接変更するのではなく、カメラの親オブジェクトにピボット用のオブジェクトを用意し、そのピボットの位置を調整することでカメラの位置を制御している。ユーザーによる将来的な実装の柔軟性をもたせるため、カメラの位置を直接変更していない。

- **視点の切り替え:**
  クリックによる視点切り替えをカメラマネージャーに実装している。ゲーム画面がクリックされた場合に、レイキャストを行い、IPlateauSandboxCameraTargetインタフェースを実装するオブジェクトを検索する。参照が見つかり、 IsCameraViewAvailableが true を返す場合はこのターゲットの視点に切り替える。
  視点の制御は PlateauSandboxCameraControllerで実装されている。
  IPlateauSandboxCameraTarget の参照、カメラのピボット、視点モード、カメラ設定を引数にインスタンスを生成し、 HandleInput を更新処理として毎フレーム呼び出すことで視点の回転や位置変更の操作を行っている。このメソッド内でマウス入力を読み取り、視点モードに応じたカメラ制御を行う。
  読み取るマウス入力は視点モードに関わらず同じで、以下の入力を取得している。
  - フレーム内のマウスの移動ベクトル
  - フレーム内のスクロール量
  - 左クリックが押下されたかどうか
  - ホイールクリックが押下されたかどうか

- **視点モードの切り替え:**
  カメラマネージャでは①一人称視点、②三人称視点、③三人称中心視点の3つの視点モードを実装している。それぞれの視点モードの具体的な実現方法は以下のとおり。
  - 一人称視点（FirstPersonView）
    - 入力処理と回転の計算
      1. マウス入力の取得: マウスの動き（delta.xとdelta.y）を取得する
      2. 回転角度の更新: マウスのY軸移動（左右の移動）でm_RotationYを減算し（カメラを左右に回転）、X軸移動（上下の移動）でm_RotationXを加算する（カメラを上下に回転）
      3. 回転の制限: m_RotationX（上下の回転角度）を-80度から25度の範囲に制限して、カメラが完全に反転するのを防ぐ
      4. カメラピボットの回転の適用: 計算された回転角度をカメラピボットのローカル回転に適用します。これにより、カメラがプレイヤーの視点を表現する
  - 三人称視点（ThirdPersonView）
    - ズームと視点移動の計算
      1. マウスの中央ボタンによる視点のドラッグ: マウスの動きに基づき、視点の水平及び垂直オフセットを更新する
      2. マウスのスクロールによるズーム: スクロール入力に基づき、カメラとターゲットとの距離（m_ViewOffset.z）を調整する
      3. 回カメラ位置の更新: ターゲットの位置と計算されたオフセットをもとにカメラピボットの位置を更新する
    - 視点回転の計算
      1. マウスの左ボタンによる回転: マウスの動きに基づき、ヨーとピッチを更新する
      2. ピッチの制限: ピッチを-70度から70度の範囲に制限する
      3. カメラピボットの向きの更新: 更新されたヨーとピッチを使用して、カメラピボットの向きを更新する

- 三人称中心視点（ThirdPersonOrbit）
  - ズームの計算
    1. マウスのスクロールによるズーム: スクロール入力に基づき、m_ViewOffset.z（カメラとターゲットとの距離）を調整する
    2. ズームの制限: ズームレベルを設定された最小値と最大値の範囲内に制限する
  - オービット回転の計算
    1. マウスの左ボタンによる回転: マウスの動きに応じて、ヨーを加算し（水平回転）、ピッチを減算する（垂直回転）
    2. 回転の適用: 計算されたヨーとピッチをターゲットの回転に適用して、カメラピボットの絶対回転を決定する
    3. カメラ位置の更新: カメラピボットの位置を、ターゲットに対するオフセット位置に更新する

- **メインカメラに視点を戻す:**
  カメラマネージャーを用いた視点に切り替える際、もともと使用されていたメインカメラの参照を保持し、元に戻す機能を実装している。スクリプトからカメラを任意のタイミングで戻したい場合 PlateauSandboxCameraManager.SwitchCamera に PlateauSandboxCameraMode.None を指定することでメインカメラに戻すことができる。

2.4.3 Maps Toolkitの使用アルゴリズム

【AL019】3D都市モデルの位置合わせ

- 3D都市モデルの位置合わせは、以下の流れで行われる。
  1. 地形モデル・テクスチャの取得: Cesium SDK for Unityを用いてPLATEAU Terrain、地形モデルのラスターデータを取得する
  2. 3D都市モデルの配置:PLATEAU SDKを用いて3D都市モデルをインポートする
  3. 位置合わせの実行: 「PLATEAUモデルの位置を合わせる」を押下すると位置合わせ処理が行われ、PLATEAUモデルと地形モデルの位置が合わせられる

  2.はPLATEAU SDKの機能のため説明は割愛し、1, 3についてMaps Toolkitでどのような処理を行っているかを記載する。

- **1. 地形モデル・テクスチャの取得:**
  - 地形モデル（PLATEAU Terrain）、テクスチャ（PLATEAU Raster）の取得の流れは以下のとおり。Cesium SDK for Unityを利用しているため、具体的な実装はCesiumのドキュメンテーションを参照されたい。（https://cesium.com/learn/apis/）

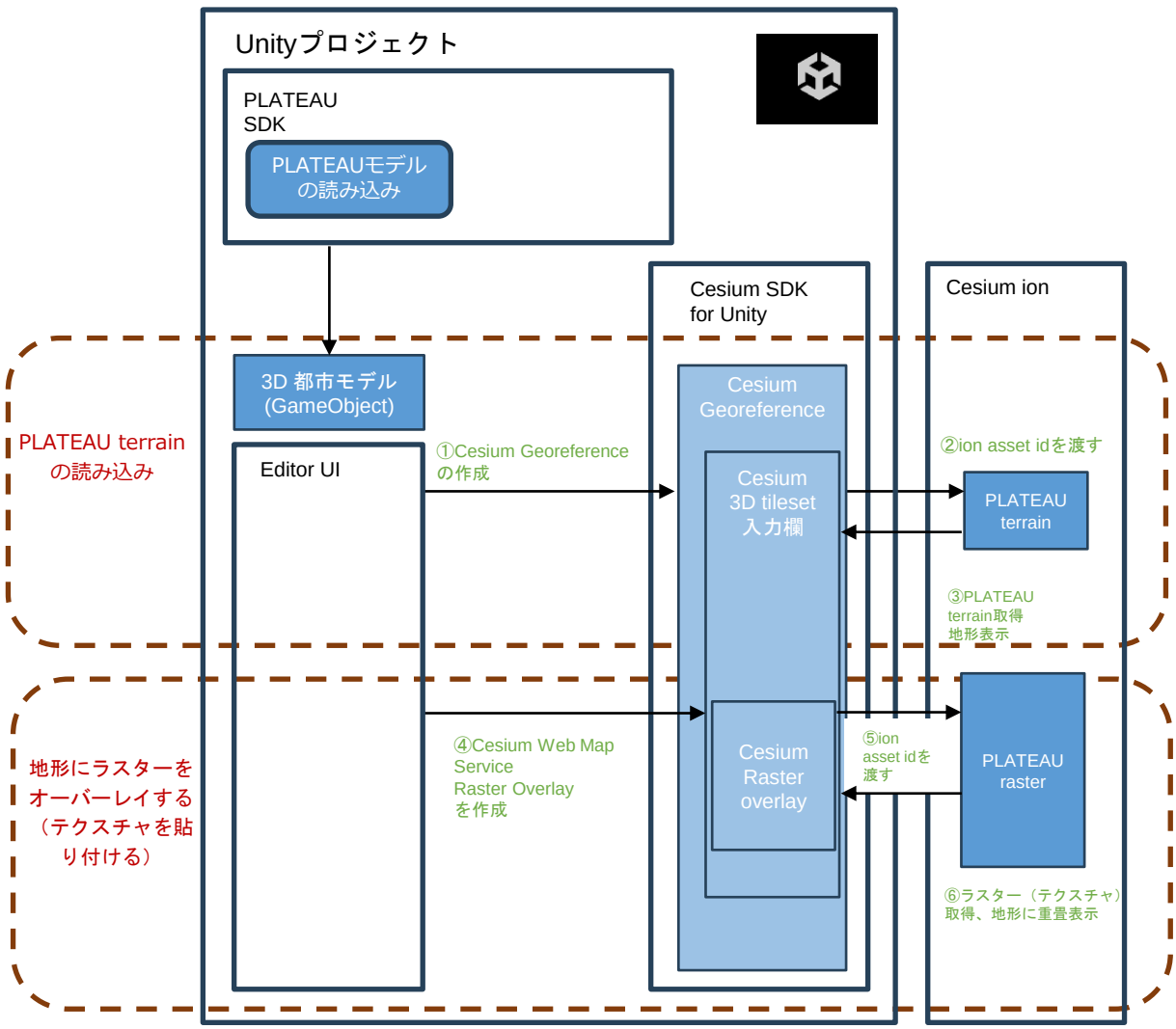


図2-2-34　地形モデル・テクスチャ取得の流れ

- **3. 位置合わせの実行:**
  - 位置合わせ処理の流れは以下のとおり

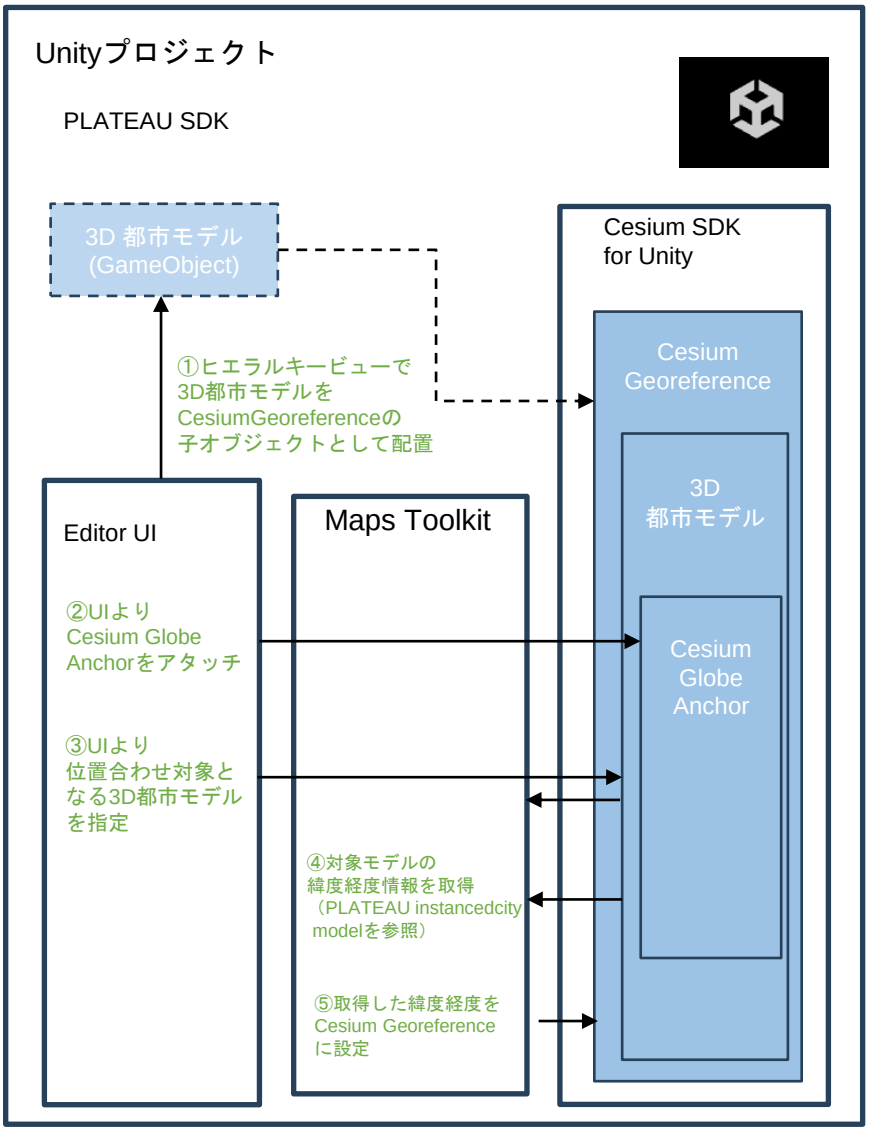


図2-2-35　3D都市モデルの位置合わせ処理の流れ

- 位置合わせ処理の具体的な実装は以下のとおり。

1. PLATEAUモデルのルートオブジェクトが選択されているかを確認する
2. PlateauMapsUtilities.GetOriginLongLatOfCityModelメソッドでSDK経由でインポートした3D都市モデルの緯度経度情報を取得する
3. PlateauMapsUtilities.RequestGeoidHeightメソッドで、取得した緯度経度情報を渡しその位置のジオイド高を取得する
4. MoveModelというコールバックを実行してモデルの位置を調整する

```
if (GUILayout.Button("PLATEAUモデルの位置を合せる"))
{
    if (m_CityModel != null
        && m_CityModel.transform.parent != null
        && m_CityModel.transform.GetComponent<CesiumGlobeAnchor>() != null
        && m_CityModel.transform.parent.GetComponent<CesiumGeoreference>() != null)
    {
        m_CityModelPosition =
PlateauMapsUtilities.GetOriginLongLatOfCityModel(m_CityModel);
        string geoidRequestUri = PlateauToolkitMapsConstants.k_GeoidApiUrl + "&latitude="
+ m_CityModelPosition.x + "&longitude=" + m_CityModelPosition.y;
        PlateauMapsUtilities.RequestGeoidHeight(geoidRequestUri, MoveModel);
    }
    else
    {
        EditorUtility.DisplayDialog(
            "中心合わせが失敗しました",
            "シーンにPlateauモデルが存在し、CesiumGlobeAnchorコンポネントが負荷されること
かつCesiumGeoreferenceにParentされていることを確認してください。",
            "OK"
            );
    }
}
```

MoveModelのメソッドの実装内容は以下の通り。

```
#if CESIUM_FOR_UNITY
void MoveModel(float resultOfGeoidHeightQuery)
{
    if (m_PositionMode == PlateauMapsPositioningMode.IfcModel && m_IfcGlobeAnchor ==
null)
    {
        return;
    }
```

（次ページに続く）

```
    if (m_PositionMode == PlateauMapsPositioningMode.PlateauModel && (m_CityModel == null ||
m_CityModel.GetComponent<CesiumGlobeAnchor>() == null))
    {
        return;
    }

    // Plateau model
    if (m_PositionMode == PlateauMapsPositioningMode.PlateauModel)
    {
        double3 longLatHeight = new double3 { x = m_CityModelPosition.y, y = m_CityModelPosition.x,
z = resultOfGeoidHeightQuery };

m_CityModel.transform.parent.GetComponent<CesiumGeoreference>().SetOriginLongitudeLatitudeHei
ght(longLatHeight[0], longLatHeight[1], longLatHeight[2]);
        m_CityModel.transform.GetComponent<CesiumGlobeAnchor>().longitudeLatitudeHeight =
longLatHeight;
        m_CityModel.transform.rotation = Quaternion.identity;
    }
    else if (m_PositionMode == PlateauMapsPositioningMode.IfcModel)// IFC model
    {

m_IfcGlobeAnchor.transform.parent.GetComponent<CesiumGeoreference>().SetOriginLongitudeLatitu
deHeight(m_IfcWgsPosition[0], m_IfcWgsPosition[1], resultOfGeoidHeightQuery);
        m_IfcWgsPosition[2] /= 1000f;
        m_IfcWgsPosition[2] += resultOfGeoidHeightQuery ;
        m_IfcGlobeAnchor.longitudeLatitudeHeight = m_IfcWgsPosition;
        Quaternion rotation = Quaternion.Euler(0, 180f -
Mathf.Atan2((float)m_IfcCoordinateInfo.XOrdinate, (float)m_IfcCoordinateInfo.XAbscissa), 0);
        m_IfcGlobeAnchor.transform.rotation = rotation;
        Vector3 scale = new Vector3((float)m_IfcCoordinateInfo.Scale, (float)m_IfcCoordinateInfo.Scale,
(float)m_IfcCoordinateInfo.Scale);
        m_IfcGlobeAnchor.transform.localScale = scale;
    }

    if (resultOfGeoidHeightQuery == 0f)
    {
        EditorUtility.DisplayDialog(
                    "高さ合わせが失敗しました",
                    "インターネットに接続されていると確認した上で再度お試しください。高さはゼロに設定さ
れました。",
                    "OK"
                    );
    }
}
#endif
```

【AL020】GISデータのゲームオブジェクト化

- GISデータ（GeoJSON、SHPファイル）をインポートする際には、Unityのシーン上でレンダリングを行えるようレンダラーを設定する必要がある。Maps Toolkitでは以下の2種類のレンダラーをファイルの形式に応じて使用している。
  - LineRenderer:
    Unityエンジン内で線や曲線を描画するためのコンポーネント。このコンポーネントを使用することで、開発者はプログラムにより直線や複雑な曲線を生成し、それらを3D空間内で視覚化することができる。
  - MeshRenderer:
    Mesh Rendererは、3Dモデルや形状のメッシュをレンダリングするためのコンポーネント。メッシュにマテリアルを適用し、それをゲームのカメラに対して可視化する役割を持っている。

- **GeoJSONのゲームオブジェクト化と表示処理**
  1. **指定されたファイルの読み出し及びパース処理：**GeoJsonLoaderが指定されたファイルを参照し、JSONデータのパースを行う
  2. **取得したデータからゲームオブジェクトを作成：**取得されたデータより、Unityプロジェクト内に表示データを配置可能にするためのゲームオブジェクトを作成する。ゲームオブジェクトはUnityシーン内に残り、シーンを保存すれば次回起動時には設定した状態のまま作業を継続できる。
  3. **作成したゲームオブジェクトへの表示関連データの格納：**ゲームオブジェクトに対し、表示に必要となるLineRendererコンポーネントをアタッチする。また、頂点データなどからグラフィックスデータを格納する。

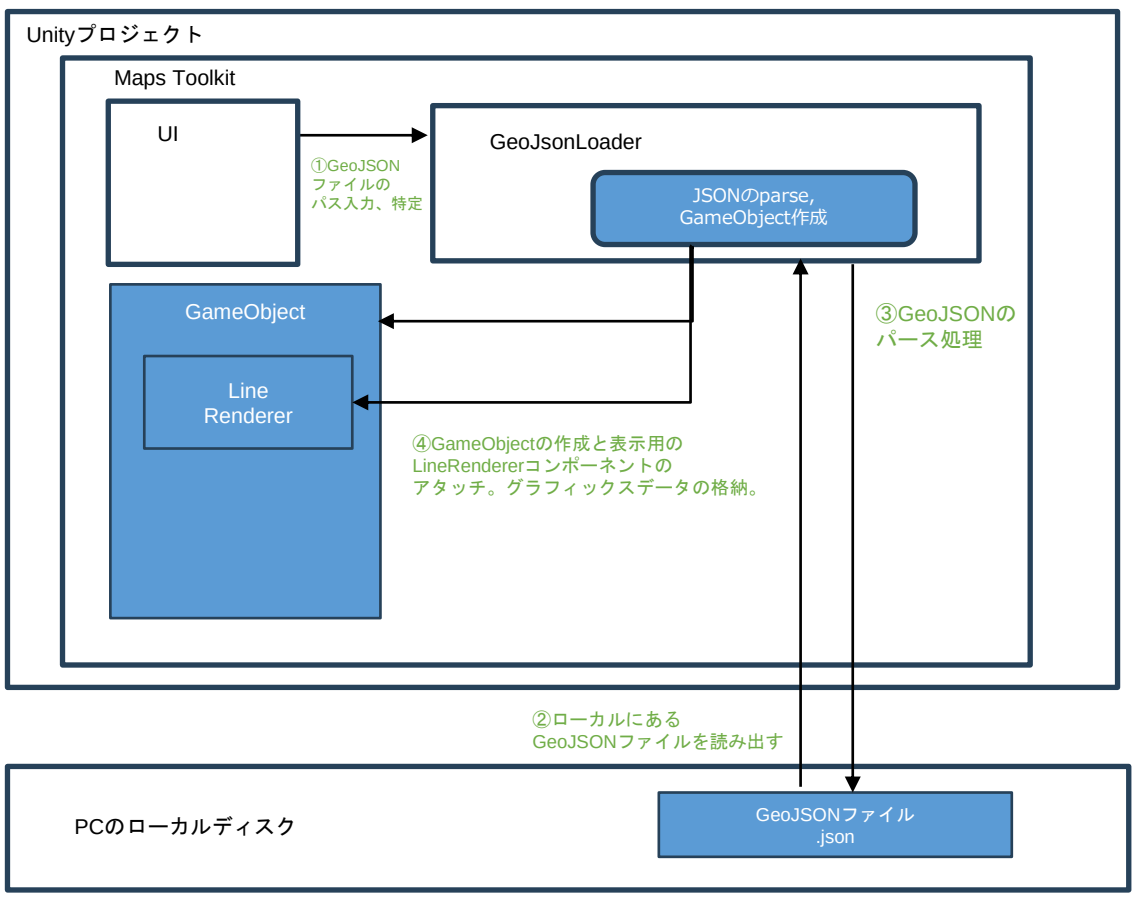


図2-2-36　GISデータ変換の流れ

- **SHPファイルのゲームオブジェクト化と表示処理**
  1. **指定されたファイルの読み出し及びパース処理：**ShapefileReaderが指定されたファイルを参照し、Byteデータの読み込みをSHPファイルのデータ仕様に従って順次読み出す
  2. **取得したデータからゲームオブジェクトを作成:**
  3. ShapefileReaderが読み出したデータをもとにUnityプロジェクト内に表示データを配置可能にするためのゲームオブジェクトを作成する。ゲームオブジェクトはUnityシーン内に残り、シーンを保存すれば次回起動時には設定した状態のまま作業を継続できる。
  4. **作成したゲームオブジェクトへの表示関連データの格納:** 作成したゲームオブジェクトに対し、UI上で指定されたレンダー方法に合わせて表示に必要となるLineRendererもしくはMesh Rendererコンポーネントをアタッチする。複数の頂点データなどからLineやMeshを作成、表示する。

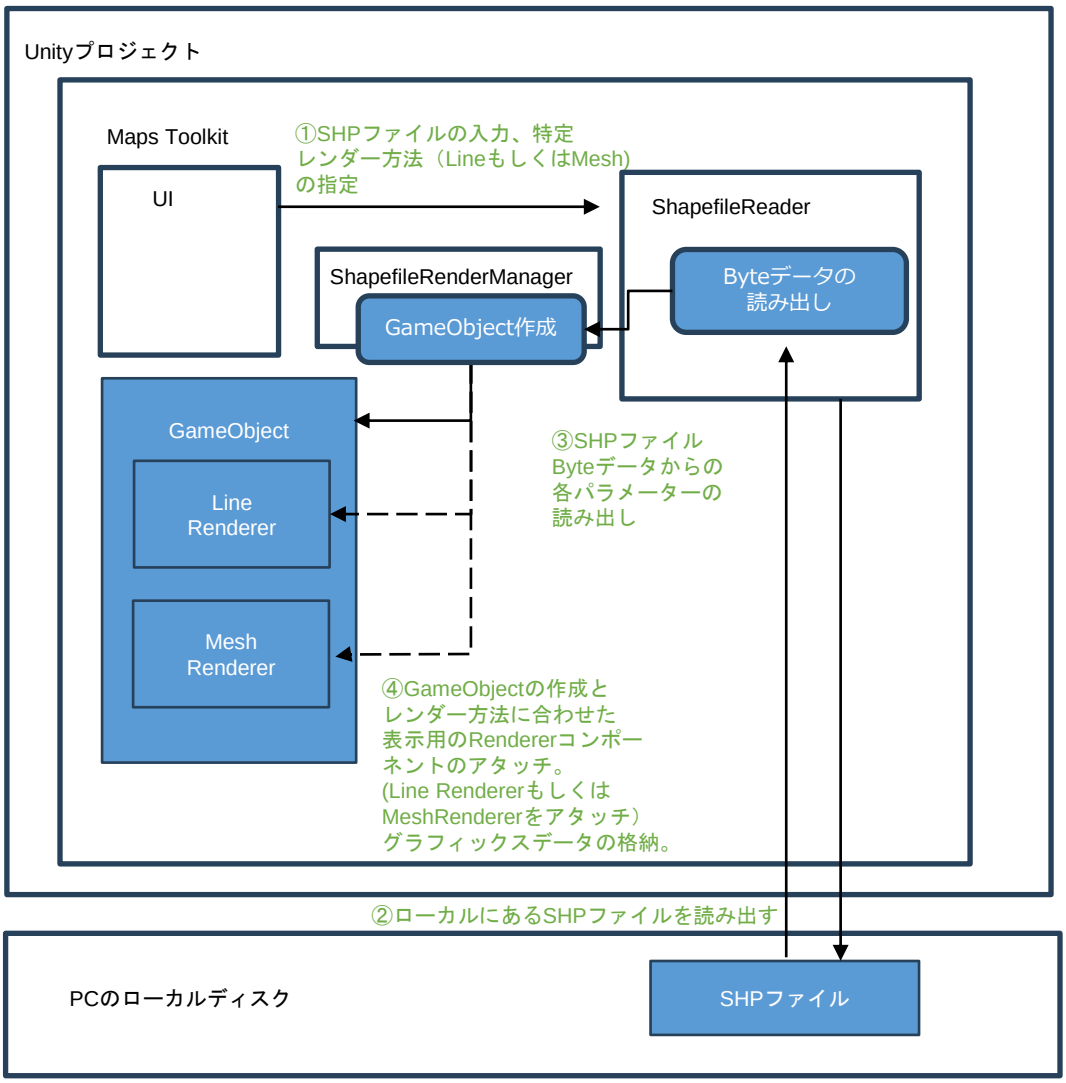


図2-2-36　SHPファイル変換の流れ

- **GISデータからのLine/Mesh生成イメージ**
  GeoJSONデータ、SHPファイルには頂点情報が含まれている。
  前述のゲームオブジェクト化の際には設定するRendererの種別に応じてLineやMeshを生成してコンポーネントとして格納する。

頂点配列を順に繋いでいく形で線分を作成しLineを作成し、
LineRendererにコンポーネントとして格納する。

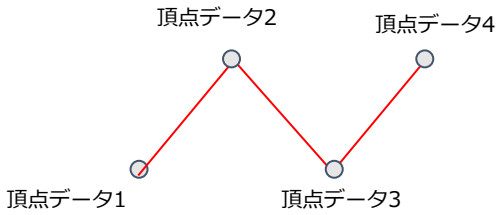


図2-2-37　頂点データからのLineの生成イメージ

頂点配列を３つ分の組み合わせ順に繋いでいく形で
三角形の面を作成し繋げることでMeshを作成し、
MeshRendererにコンポーネントを格納する。

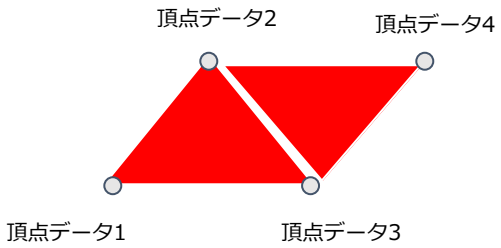


図2-2-38　頂点データからのMeshの生成イメージ

- **GISデータ種別の切り替え実装**
  PlateauToolkitMapsWindowクラス内の以下のスクリプトでSHPファイルかGeoJSONファイルを読み込むかを指定する。

```
m_GisModeIndex = EditorGUILayout.Popup("GIS type", m_GisModeIndex, k_GisMode);
```

- **SHPファイル読み込みの実装**
  ShapefileReaderクラスで実装を行っている。
  以下のReadShapesメソッドで、Shapefileの仕様で定められているバイトアドレスを読み取りSHPファイルの種類を判別する。

```
public List<IShape> ReadShapes()
{
    using (FileStream fileStream = new FileStream(m_ShpPath, FileMode.Open))
    {
        m_ShpReader = new BinaryReader(fileStream);
        ReadHeader();
        List<IShape> shapes = new List<IShape>();

        while (m_ShpReader.BaseStream.Position < m_ShpReader.BaseStream.Length)
        {
            m_ShpReader.ReadInt32BE(); // Record number
            m_ShpReader.ReadInt32BE(); // Content length

            m_ShapeType = m_ShpReader.ReadInt32();
            if (m_ShapeType == (int)ShapeType.Polygon) // Cast to int since m_ShapeType is likely an int
            {
                shapes.Add(ReadPolygonShape());
            }
            else if (m_ShapeType == (int)ShapeType.Polyline)
            {
                shapes.Add(ReadPolylineShape());
            }
            else if (m_ShapeType == (int)ShapeType.Point)
            {
                shapes.Add(ReadPointShape());
            }
            else
            {
                Debug.LogError($"Unsupported shape type: {m_ShapeType}");
                break;
            }

        }
        return shapes;
    }
}
```

- **SHPファイルの付属情報（DBF）の読み込み実装**
  SHPファイルにはDBFファイルという付属情報が格納されたファイルが同梱されている場合がある。
  このDBFファイルを読み込む処理はDBFReaderクラスを用いて行う。
  まず、ReadHeaderメソッドを用いてファイルの中身、カラムの構成やレコードの長さを取得する。

```
public void ReadHeader()
{
    if (File.Exists(m_DbfFilePath))
    {
        m_FileStream = new FileStream(m_DbfFilePath, FileMode.Open);
        m_Reader = new BinaryReader(m_FileStream);
        m_Reader.BaseStream.Seek(4, SeekOrigin.Begin);
        m_RecordCount = m_Reader.ReadInt32();
        m_Reader.BaseStream.Seek(8, SeekOrigin.Begin); // skip initial bytes
        m_HeaderLength = m_Reader.ReadInt16();
        m_RecordLength = m_Reader.ReadInt16();
        m_FieldCount = (m_HeaderLength - 32) / 32;
        // Read the field names and lengths
        m_FieldNames = new string[m_FieldCount];
        m_FieldLengths = new int[m_FieldCount];
        for (int i = 0; i < m_FieldCount; i++)
        {
            m_Reader.BaseStream.Seek(32 + i * 32, SeekOrigin.Begin);
            byte[] nameBytes = m_Reader.ReadBytes(11);
            m_FieldNames[i] = Encoding.ASCII.GetString(nameBytes).TrimEnd('¥0'); //
remove trailing null characters
            m_Reader.BaseStream.Seek(32 + i * 32 + 16, SeekOrigin.Begin);
            m_FieldLengths[i] = m_Reader.ReadByte();
        }

        m_Reader.BaseStream.Seek(m_HeaderLength, SeekOrigin.Begin);
    }
}
```

ReadHeaderメソッドでファイルの構成を確認したら、以下のParseDbfメソッドで一つ一つのレコード（付属情報の項目）を読み込む。

```
public void ParseDbf()
{
    if (m_HeaderLength <= 0)
    {
        return;
    }

    // Print the field names
    for (int i = 0; i < m_FieldNames.Length; i++)
    {
        UnityEngine.Debug.Log(m_FieldNames[i] + "¥t");
    }

    int records = 0;
    while ((m_Record = ReadNextRecord()) != null)
    {
        if (!m_Record.IsDeleted)
        {
            records++;
            foreach (string field in m_Record.Fields)
            {
                UnityEngine.Debug.Log(field);
            }
        }
    }
    UnityEngine.Debug.Log(records);
}
```

- **SHPファイルの描画実装**
  前述の通り、メッシュかラインを指定してSHPファイルを描画することができる。
  ShapefileRenderManagerクラスで描画処理を行う。
  まず、ポイント（点）データを以下のDrawPointメソッドを用いて描画する。

```
void DrawPoint(IShape shape, GameObject parentObject, bool dbfRead, DbfReader
dbfReader, DbfRecord record)
{
#if UNITY_URP
   GameObject markerObjectDefault =
AssetDatabase.LoadAssetAtPath<GameObject>(PlateauToolkitMapsConstants.k_PointMark
erPrefab);
#endif
#if UNITY_HDRP
   GameObject markerObjectDefault =
AssetDatabase.LoadAssetAtPath<GameObject>(PlateauToolkitMapsConstants.k_PointMark
erHdrpPrefab);
#endif


   foreach (Vector3 point in shape.Points)
   {
      double3 coordinates = new(point.x, point.z, m_RenderHeight);

m_PositionMarkerSphere.GetComponent<CesiumGlobeAnchor>().longitudeLatitudeHeight
= coordinates;
      Vector3 pointPos = m_PositionMarkerSphere.transform.position;
      GameObject marker = m_PointDataPrefab == null ?
(GameObject)PrefabUtility.InstantiatePrefab(markerObjectDefault) :
GameObject.Instantiate(m_PointDataPrefab);
      marker.name = "point_data";
      marker.transform.parent = parentObject.transform;
      marker.AddComponent<CesiumGlobeAnchor>().longitudeLatitudeHeight =
coordinates;
      if (!string.IsNullOrEmpty(m_DbfFilePath) && dbfRead &&
dbfReader.GetRecordLength() == m_ListOfShapes.Count)
      {
         AttachMetadata(marker, record);
      }
   }
}
```

ライン形式で描画を行う場合、以下のDrawPolygonOrPolylineメソッドで描画を行う。内部的には、UnityのLineRender機能を用いて描画しており、描画の際付属データ（Dbf）があれば付与する。

```
void DrawPolygonOrPolyline(IShape shape, int index, float lineWidth, GameObject shapeParent,
GameObject parentObject, bool dbfRead, DbfReader dbfReader, DbfRecord record, GameObject mesh)
{
    for (int i = 0; i < shape.Parts.Count - 1; i++)
    {
        int start = shape.Parts[i];
        int end = shape.Parts[i + 1];

        // get the points
        List<Vector3> partPoints = shape.Points.GetRange(start, end - start);
        List<Vector3> partPointsWorld = new List<Vector3>();

        foreach (Vector3 point in partPoints)
        {
            double3 coordinates = new(point.x, point.z, m_RenderHeight);
            m_PositionMarkerSphere.GetComponent<CesiumGlobeAnchor>().longitudeLatitudeHeight =
coordinates;
            Vector3 pointPos = m_PositionMarkerSphere.transform.position;
            partPointsWorld.Add(pointPos);
        }

        if (m_RenderMode == 1)
        {
            GameObject shpParentInstance = (GameObject)PrefabUtility.InstantiatePrefab(shapeParent);

            shpParentInstance.transform.position = Vector3.zero;
            shpParentInstance.name = "shpParent_" + index;

            shpParentInstance.transform.parent = parentObject.transform;
            LineRenderer lineRenderer = shpParentInstance.GetComponent<LineRenderer>();
            lineRenderer.positionCount = partPointsWorld.Count;
            lineRenderer.useWorldSpace = false;
            lineRenderer.startWidth = lineWidth;  // Set the start width
            lineRenderer.endWidth = lineWidth;
            lineRenderer.SetPositions(partPointsWorld.ToArray());
            if (m_LoopLineRenderer)
            {
                lineRenderer.loop = true;
            }
```

（次ページに続く）

```
            else
            {
                lineRenderer.loop = false;
            }
            if (!string.IsNullOrEmpty(m_DbfFilePath) && dbfRead && dbfReader.GetRecordLength() ==
m_ListOfShapes.Count)
            {
                AttachMetadata(shpParentInstance, record);
            }
        }
        else if (m_RenderMode == 0)
        {
            GameObject meshObject = (GameObject)PrefabUtility.InstantiatePrefab(mesh);

            meshObject.transform.position = Vector3.zero;
            meshObject.transform.parent = parentObject.transform;
            if (!string.IsNullOrEmpty(m_DbfFilePath) && dbfRead && dbfReader.GetRecordLength() ==
m_ListOfShapes.Count)
            {
                AttachMetadata(meshObject, record);
            }
            CreateMesh(false, partPointsWorld, meshObject.GetComponent<MeshFilter>(),
meshObject.GetComponent<MeshRenderer>());
        }
        else
        {
            Debug.LogError("Failed to instantiate shpLineRendererObject");
        }
    }
}
```

メッシュ形式で描画を行う場合、以下のCreateMeshメソッドで描画を行う。

```
public void CreateMesh(bool isHole, List<Vector3> points, MeshFilter meshFilter, MeshRenderer
meshRenderer)
{
    var vertices = new List<Vertex>();
    // add your Vector3 points as Vertex to the points list here.

    for (int k = 0; k < points.Count; k++)
    {
        vertices.Add(new TriangleNet.Geometry.Vertex(points[k].x, points[k].z));
    }

    Polygon polygon = new Polygon();
    polygon.Add(new Contour(vertices), isHole);



    TriangleNet.Meshing.IMesh mesh = polygon.Triangulate();

    List<int> triangles = new List<int>();
    List<Vector3> unityVertices = new List<Vector3>();

    foreach (TriangleNet.Topology.Triangle triangle in mesh.Triangles)
    {
        unityVertices.Add(new Vector3((float)triangle.GetVertex(0).X, 0, (float)triangle.GetVertex(0).Y));
// Assume Y is up
        unityVertices.Add(new Vector3((float)triangle.GetVertex(1).X, 0, (float)triangle.GetVertex(1).Y));
        unityVertices.Add(new Vector3((float)triangle.GetVertex(2).X, 0, (float)triangle.GetVertex(2).Y));

        // Add the indices of the triangle vertices to the triangles list
        triangles.Add(unityVertices.Count - 1);
        triangles.Add(unityVertices.Count - 2);
        triangles.Add(unityVertices.Count - 3);
    }

    // Create a new Unity Mesh
    Mesh unityMesh = new Mesh();
    unityMesh.vertices = unityVertices.ToArray();
    unityMesh.triangles = triangles.ToArray();

    unityMesh.RecalculateNormals();
    meshFilter.sharedMesh = unityMesh;

    meshRenderer.sharedMaterial = isHole ? m_Counter : m_Clockwise;
}
```

- **GeoJSONファイルの読み込み実装**
  GeoJsonLoaderクラスでGeoJSONファイルの処理を行う。以下のReadGeoJsonDataメソッドを用いてGeoJSONファイルを開き、データの種類を判別する。

```
void ReadGeoJsonData(string filePath, float height, float lineWidth, string
currentRenderingObjectName)
{
    string jsonString = File.ReadAllText(filePath);
    GeoJson geoJson = JsonConvert.DeserializeObject<GeoJson>(jsonString);

    GameObject parentObject = new GameObject(currentRenderingObjectName +
"_GeoJSON");
    parentObject.transform.parent = m_GeoRef.transform;
    parentObject.AddComponent<CesiumGlobeAnchor>();

#if UNITY_URP
    GameObject shapeParent =
AssetDatabase.LoadAssetAtPath<GameObject>(PlateauToolkitMapsConstants.k_ShapePar
entPrefab);
    m_PointMarkerDefaultPrefab =
AssetDatabase.LoadAssetAtPath<GameObject>(PlateauToolkitMapsConstants.k_PointMark
erPrefab);
#endif
#if UNITY_HDRP
    GameObject shapeParent =
AssetDatabase.LoadAssetAtPath<GameObject>(PlateauToolkitMapsConstants.k_ShapePar
entHdrpPrefab);
    m_PointMarkerDefaultPrefab =
AssetDatabase.LoadAssetAtPath<GameObject>(PlateauToolkitMapsConstants.k_PointMark
erHdrpPrefab);
#endif
    GameObject mesh =
AssetDatabase.LoadAssetAtPath<GameObject>(PlateauToolkitMapsConstants.k_MeshObje
ctPrefab);

    ProcessFeatures(geoJson.features, parentObject, shapeParent, height, lineWidth);
}
```

- **GeoJSONファイルの描画実装**
  以下のProcessFeatureメソッドを用いてGeoJSONファイルを描画する。
  Shapefile同様、ポイント、線などのデータタイプを判別した後、描画を行う（switch(type)）。
  付属情報があれば付与したうえで描画を行う。

```
void ProcessFeatures(IEnumerable<Feature> features, GameObject parentObject,
GameObject shapeParent, float height, float lineWidth)
{
    foreach (Feature feature in features)
    {
        string type = feature.geometry.type;
        Dictionary<string, object> properties = feature.properties;
        var coordinates = (JArray)feature.geometry.coordinates;

        switch (type)
        {
            case "Point":
                ProcessPoint(coordinates, parentObject, height, properties);
                break;
            case "MultiPolygon":
                ProcessMultiPolygon(coordinates, parentObject, shapeParent, height, lineWidth,
properties);
                break;
            case "Polygon":
                ProcessPolygon(coordinates, parentObject, shapeParent, height, lineWidth,
properties);
                break;
            case "LineString":
                ProcessLineString(coordinates, parentObject, shapeParent, height, lineWidth,
properties);
                break;
            default:
                Debug.LogError($"Unsupported geometry type: {type}");
                break;
        }
    }
}
```

2.4.4 AR Extensionsの使用アルゴリズム

【AL021】ARオクルージョン

- オクルージョンとは前面にあるオブジェクトがその後ろにあるオブジェクトを遮蔽する機能全般を意味するが、ここでは特に透明なオブジェクトが背後のオブジェクトを遮蔽する機能を指す。本来、透明なオブジェクトの背後に別のオブジェクトがある場合、手前のオブジェクトを透過して背後のオブジェクトが描画される。ARアプリケーションでは透明な3D都市モデルを実際の建物の位置にオーバーレイすることで、ARアプリケーションで表示する3Dオブジェクトが実際の建物に遮蔽されるように表示させるユースケースがあるため、この機能を開発した。
- Built-in レンダリングパイプラインの場合はシェーダーを用いて簡単にオクルージョンを実装することができるが、URPを利用している場合はシェーダーのみで実現することが難しいため、URPの Renderer Feature 機能を用いてオクルージョンを実現する。
- AR Extensions で提供するオクルージョンを実現するための Renderer Feature、PlateauAROcclusionRendererFeature はURPに標準で用意されているRender Objects Renderer Featureの実装を再利用している。
- **実装詳細**
  - 生成される3つの RenderObjectsPass はそれぞれ次のような役割を持つ。
  - m_RenderObjectsPassOpaque
    - 役割：このパスは、不透明なオブジェクト（オクルーディ）のレンダリングを担当する。レンダーキューの種類をOpaqueとして設定しており、シーン内の不透明なオブジェクトを描画する際に使用される。
    - オクルージョンとの連携: 不透明オブジェクトは最初にレンダリングされ、その後、オクルーダーによって隠される部分を決定する。これは、オクルーダーがこれらのオブジェクトの前に描画された時に、オクルーディ（不透明オブジェクト）が見えなくなる基盤を作成する。
  - m_RenderObjectsPassTransparent
    - 役割：透明オブジェクト（オクルーディ）のレンダリングを管理する。レンダーキューの種類をTransparentとして設定し、シーン内の透明オブジェクトを描画するために使用される。透明オブジェクトは、不透明オブジェクトとは異なるレンダリングの扱いが必要であり、その透明度に応じて背後のオブジェクトの可視性が制御される。
    - オクルージョンとの連携: 透明オブジェクトのレンダリングは、オクルーダーの描画後に行われることが多い。このパスは、これらのオブジェクトがどのように背景と合成されるか、そしてオクルーダーによってどの部分が隠されるかを決定する。
  - m_RenderObjectMaterialOverride
    - 役割: このパスは、オクルーダーに特定のマテリアルを適用することで、オクルージョン処理をカスタマイズする。レンダーキューの種類をTransparentに設定し、オクルーダーマスクを使用してオクルーダーを特定する。このパスにより、オクルーダーに適用されるマテリアルをオーバーライドし、オクルージョンの視覚的効果を制御する。
    - オクルージョンとの連携: このパスは、オクルーダーがオクルーディの前に描画される際に、特定のマテリアルを適用することで、オクルーディがどのように隠されるかを細かく制御する。例えば、オクルーダーに深度情報のみをレンダリングするマテリアルを適用することで、オクルーディがオクルーダーの背後にある場合にのみレンダリングされる。
  - これら3つのパスは連携して、AR環境におけるリアルなオクルージョン効果を提供している。不透明及び透明オブジェクトのレンダリングを分けることで、オクルーダーによってオブジェクトがどのように隠されるかを適切に管理し、オクルーダーマテリアルのオーバーライドを通じてオクルージョンのビジュアルをカスタマイズしている。

【AL022】Geospatial APIを用いた位置合わせ（PlateauARPositioning）

- PlateauARPositioningは Google Geospatial API （ARCore API）を用いた3D都市モデルのAR空間内の位置合わせ機能に使用する。
- このコンポーネントでは Cesium for Unity を用いて取得するPLATEAUの3D Tilesと PLATEAU SDK でインポートした3D都市モデルの位置合わせを行うことができる。なお、それぞれの3D都市モデルが持つ緯度経度高度の情報の取り方が異なるため、一部処理が分岐する。

- **ジオイド高取得コンポーネント：**
  ジオイド高は PlateauARGeoidHeightProvider を用いて取得する。このコンポーネントはインタフェースを定義するためのコンポーネントになっており、具体的な処理を持たない。そのため、ユーザーが利用できるジオイド高取得APIを用いた実装を準備する必要がある。サンプルでは国土地理院のジオイド高APIを利用した実装 （ GsiGeoidHeightProvider ）を提供している。このように抽象化することでPLATEAU AR Extensions自体が外部のAPIの変更に影響されることを避けている。

- **Geospatial APIコントローラー：**
  Geospatial API の管理はジオイド高取得コンポーネントと同様に PlateauARGeospatialController というコンポーネントで抽象化している。こちらも、具体的な実装はサンプルで GeospatialController として提供している。このインタフェースは Geospatial API を利用した緯度経度の取得やグローブアンカーオブジェクト（AR空間内の特定の位置に3Dオブジェクトを配置する機能）の作成など、AR Extensions で利用する機能を定義している。

```
public virtual bool TryGetPose(out GeospatialPose pose) {}
public virtual void CreateAnchoredObject(
                    double latitude, double longitude, double altitude, Transform obj) {}
```

- **位置合わせ計算のフロー：**
  1. PlateauARPositioning は PLATEAU SDK でインポートした PLATEAUInstancedCityModel か CesiumGeoreference がアタッチされているオブジェクトにアタッチする
  2. Gespatial API コントローラーを初期化して、Geospatial API を利用できるようにします
  3. 3D都市モデルの中心点の緯度経度を取得します
     - PLATEAU SDKから3D都市モデルをインポートしている場合
       i. 3D都市モデルインスタンスのAPIを用いて、インポートした中心点である緯度経度を取得する

```
GeoCoordinate coords = m_PlateauCityModel.GeoReference.Unproject(new
PlateauVector3D(0, 0, 0));
```

     - Cesiumから3D Tilesをストリーミングしている場合
       i. TryGetPose を利用して、プレイヤーの位置（カメラの位置）の緯度経度を取得する
       ii. Cesium を利用する場合はプレイヤーの位置を中心に3D都市をストリーミングで配置するため、プレイヤーの位置を位置合わせの基準点に使用する
  4. （Cesiumのみ）Geospatial API で取得した緯度経度を CesiumGeoreference に設定する。CesiumGeoreference は設定した緯度経度を座標の中心から周囲に取得して配置する

5. ジオイド高取得コンポーネントを利用して緯度経度からジオイド高を取得する
6. Geospatial API コントローラーの CreateAnchoredObject を利用して、緯度経度とジオイド高を指定してアンカーを作成する
    - アンカー作成時に PlateauARPositioning の transform （配置される建物の3Dオブジェクトの親オブジェクト）を指定することで、3D都市モデルの中心が Geospatial APIで計算されたAR空間内の正しい位置に固定される
    - ジオイド高を指定するのは、PLATEAU では個々の建物が親オブジェクトのローカル座標で標高の高さに設定されており、親オブジェクトの高さをジオイド高に設定すればワールド座標で見たときに個々の建物の高さが楕円体高になるためである。Geosptial APIのAR空間では楕円体高が高さの単位として使用されているため、上記の設定により建物が意図した高さに配置される

【AL023】ARマーカーを用いた高さ合わせ（PlateauARmarkerGroundController）

- ARマーカー機能を利用した高さ合わせ機能では、ARマーカーを地面に置くことで表示される建物の高さのズレを修正する。ARマーカーの設定はAR Foundationの標準の機能を利用する。
- PlateauARMarkerGroundController がこれらの機能を実装しており、このコンポーネントをアタッチすることで、ARマーカーがカメラで検出された際に3D都市モデルが配置されている高さとARマーカーの高さの差を計算する。建物のAR空間内の位置は PlateauARPositioning で管理されているため、計算された高さの差分を適用するためには PlateauARPositioning.SetOffset に配置する位置のオフセットを指定する必要がある。

- **高さのズレの計算方法：**
  - ARマーカーが読み込まれたときのコールバックを ARTrackedImageManager.trackedImagesChanged に登録し、ARマーカーが読み込まれたときに高さのズレの検出を開始する
  - レイキャストを用いて、ARマーカーの近くの建物の底辺の座標を検出する
    - 3D都市モデルに含まれる個々の建物の標高（建物メッシュの底辺の座標）は個別に保持されないため、レイキャストを使わない場合は個々の建物のメッシュの頂点情報から計算を行う必要があり、非常に複雑な実装を行う必要がある。一方でレイキャストを使えば、計算コスト自体は同様にかかる一方で、実装を簡略化することができるため、本Toolkitではレイキャストを仕様した実装を採用している。
    - レイキャストを用いた底辺の検出の流れは以下の通り。
      1. ARマーカーが検出された座標から、初期値0.5m×0.5mの大きさで上方向にボックスキャストを行う
      2. レイキャストの衝突が検出されない場合は、さらにボックスキャストの大きさを0.5mずつ大きくして再度レイキャストを行う。その後、建物が検出されるまで最大100m×100mの大きさまでボックスキャストによる底辺の検出を継続する
      3. レイキャストの衝突が検出されたとき、その座標がARマーカーに一番近い建物の底辺として検出を終了する
  - ARマーカーと検出された底辺の高さの座標の差分を計算する
    - 高さの差分＝底辺の高さーARマーカーの高さ

【AL024】ARマーカーを用いた位置合わせ（PlateauARmarkerCityModel）

- PlateauARMarkerCityModel で、ARマーカーを用いた3D都市モデルのAR空間への配置を実装している。予め、ARマーカーに対応した3D都市モデルの相対位置を設定しておくことにより、現実世界の特定の位置・向きにARマーカーを配置して端末で読み取ると、その周囲に3D都市モデルを描画させることができる。
- 位置の計算は以下の通り。回転も同様である。
  3D都市モデルの相対位置 = 検出されたマーカーの位置 - 3D都市モデルからのマーカーの相対位置
- 位置を反映する際は、即時に計算されたARマーカーからの3D都市モデルの相対位置を3D都市モデルオブジェクトに設定せず、少しずつ計算された位置に近づけている。これは、ARマーカーの位置がARの性質上、完全に安定した値にならないためである。

```
cityModelTransform.position =
    Vector3.Lerp(cityModelTransform.position, m_ToPosition, Time.deltaTime * 10);
cityModelTransform.rotation =
    Quaternion.Lerp(cityModelTransform.rotation, m_ToRotation, Time.deltaTime * 10);
```

  - cityModelTransform：3D都市モデルの座標
  - m_ToPosition , m_ToRotation：計算された相対位置と回転

【AL025】手動位置合わせ

- PlateauARPositioning をアタッチするオブジェクトは、位置合わせ時にARマーカーの子オブジェクトとして登録するため、PlateauARPositioning の transform.localPosition （初期値は 0, 0, 0）に位置を指定することでオフセットを設定することができる。

2.4.5 PLATEAU Utilitiesの使用アルゴリズム

【AL026】プレハブへのライトマップの適用

- 通常Unityのライトマップはシーン単位での運用を行う仕様となっている。「ライトマップのプレハブへの適用」機能は、任意のプレハブにPrefabLightmapDataというコンポーネントを追加することで、シーンにひも付いたライトマップに関連するデータをプレハブ側に保存することができ、プレハブ単位でのライトマップの運用が可能になる。
- 具体的には、任意のプレハブ内で使用されているライトマップと、メッシュ及びメッシュが参照するライトマップのID、ライトマップUVのオフセットスケール情報を保存しそれらの情報をプレハブ生成時に再マッピングする。

- **PrefabLightmapDataクラスの詳細**
  - 格納するデータ変数:
    - RendererInfo: 各メッシュのライトマップインデックスとオフセットスケールを格納
      - Renderer: ライトマップ適用済みのメッシュ
      - LightmapIndex: メッシュが使用するライトマップテクスチャのID
      - LightmapOffsetScale: メッシュのライトマップUVのオフセットスケール

```
[Serializable]
struct RendererInfo
{
    public Renderer m_Renderer;
    public int m_LightmapIndex;
    public Vector4 m_LightmapOffsetScale;
}
```

    - Lightmaps: ライトマップのカラーテクスチャを格納

```
[SerializeField]
private Texture2D[] m_Lightmaps;
```

    - LightmapsDir: ライトマップの方向テクスチャを格納

```
[SerializeField]
private Texture2D[] m_LightmapsDir;
```

    - ShadowMasks: ライトマップのシャドウマスクテクスチャを格納

```
[SerializeField]
private Texture2D[] m_ShadowMasks;
```

  - 主要メソッド:
    - GenerateLightmapInfo: プレハブにライトマップ情報を保存する。
    - Init: シーンにプレハブがロードされたとき、またはエディタでプレハブが操作されたときに実行される初期化処理
    - ApplyRendererInfo: プレハブのメッシュに対して、保存されたライトトマップ情報に基づき、ライトマップ情報を再適用する

- **ライトマップ各種データをプレハブに保存する**
  - ライトマップの各種データのひも付け （GenerateLightmapInfoメソッド）：
    - シーン内のライトマップ設定を確認し、ベイクが必要な場合はライトマッピングプロセスを実行
    - MeshRenderer コンポーネントのライトマップ情報を含む RendererInfo 配列を作成
    - 各 MeshRenderer の lightmapIndex と lightmapOffsetScale を取得し、RendererInfo オブジェクトに割り当て
    - 新しく生成された RendererInfo オブジェクトをプレハブの PrefabLightmapData コンポーネントに保存
    - シーン内で使用されているライトマップテクスチャを m_Lightmaps、方向テクスチャを m_LightmapsDir、シャドウマスクを m_ShadowMasks 配列に収集
    - メッシュのライトマップ情報及び、ライトマップの各種テクスチャデータがコンポーネントに保存される

```
// GenerateLightmapInfoメソッド
// ライトマップの各種データをPrefabLightmapDataコンポーネントに保存
static void GenerateLightmapInfo(GameObject root, List<RendererInfo> rendererInfos,
List<Texture2D> lightmaps, List<Texture2D> lightmapsDir, List<Texture2D> shadowMasks,
List<LightInfo> lightsInfo)
{
    // ルートGameObjectとその子孫からMeshRendererコンポーネントを全て取得
    MeshRenderer[] renderers = root.GetComponentsInChildren<MeshRenderer>();

    // 取得した各MeshRendererについて処理
    foreach (MeshRenderer renderer in renderers)
    {
        // ライトマップが適用されている（lightmapIndexが-1ではない）レンダラーのみ処理
        if (renderer.lightmapIndex != -1)
        {
            RendererInfo info = new RendererInfo
            {
                m_Renderer = renderer, // RendererInfo構造体を新たに作成し、基本情報を格納
                m_LightmapOffsetScale = renderer.lightmapScaleOffset, // ライトマップスケールとオフセッ
トを設定
            };

            // 現在のレンダラーに適用されているライトマップテクスチャを取得
            Texture2D lightmap = LightmapSettings.lightmaps[renderer.lightmapIndex].lightmapColor;
            Texture2D lightmapDir = LightmapSettings.lightmaps[renderer.lightmapIndex].lightmapDir;
            Texture2D shadowMask = LightmapSettings.lightmaps[renderer.lightmapIndex].shadowMask;

            // 既に同じライトマップテクスチャがリストに存在するかチェックし、新しいもののみ追加
            info.m_LightmapIndex = lightmaps.IndexOf(lightmap);
```

（次ページに続く）

```
            if (info.m_LightmapIndex == -1) // 未追加のテクスチャの場合
            {
                info.m_LightmapIndex = lightmaps.Count; // 新しいインデックスを割り当て
                lightmaps.Add(lightmap); // ライトマップテクスチャをリストに追加
                lightmapsDir.Add(lightmapDir); // 方向性ライトマップテクスチャをリストに追加
                shadowMasks.Add(shadowMask); // シャドウマスクテクスチャをリストに追加
            }

            // 処理したRendererInfoをリストに追加
            rendererInfos.Add(info);
        }
    }
}
```

- **ライトマップ各種データ再利用の流れ**
  - 初期化 （Initメソッド）：
    - シーンにプレハブが追加されたとき、初期化処理が実行される
    - シーンのLightmapSettingsに保存されているライトマップと新しいライトマップを比較し、重複がないように統合する
    - 統合されたライトマップ情報をシーンのLightmapSettingsに適用する
    - プレハブに保存されたライトマップが現在のシーンに再登録される

```
void Init()
{
    // 現在のシーンのライトマップ情報を取得
    LightmapData[] lightmaps = LightmapSettings.lightmaps;

    // 新しいライトマップと既存のライトマップのインデックスを格納する配列
    int[] offsetsIndexes = new int[m_Lightmaps.Length];

    // 現在のシーンのライトマップの数
    int totalCount = lightmaps.Length;

    // 統合後のライトマップリストを作成
    List<LightmapData> combinedLightmaps = new List<LightmapData>();

    // プレハブに保存されたライトマップを現在のシーンのライトマップと比較し、統合する
    for (int i = 0; i < m_Lightmaps.Length; i++)
    {
        bool exists = false;
        for (int j = 0; j < lightmaps.Length; j++)
        {
            // 既存のライトマップと一致するかチェック
            if (m_Lightmaps[i] == lightmaps[j].lightmapColor)
            {
                exists = true;
                offsetsIndexes[i] = j; // 一致するインデックスを保存
                break;
            }
        }
```

（次ページに続く）

```
        // 一致するライトマップがない場合、新たに統合リストに追加
        if (!exists)
        {
            offsetsIndexes[i] = totalCount; // 新しいインデックスを割り当て
            LightmapData newLightmapData = new LightmapData
            {
                lightmapColor = m_Lightmaps[i],
                lightmapDir = m_LightmapsDir.Length == m_Lightmaps.Length ? m_LightmapsDir[i] :
null,
                shadowMask = m_ShadowMasks.Length == m_Lightmaps.Length ? m_ShadowMasks[i] :
null,
            };
            combinedLightmaps.Add(newLightmapData);
            totalCount++;
        }
    }

    // 統合されたライトマップ情報を現在のシーンに適用
    LightmapSettings.lightmaps = combinedLightmaps.ToArray();

    // レンダラー情報に基づいてライトマップ情報を適用
    ApplyRendererInfo(m_RendererInfo, offsetsIndexes, m_LightInfo);
}
```

- ライトマップの再適用 （ApplyRendererInfoメソッド）：
  - RendererInfo配列をループし、各レンダラーにlightmapIndexとlightmapOffsetScaleを設定する
  - 各メッシュのライトマップUVが復元され、ライトマップが最適用される

```
static void ApplyRendererInfo(RendererInfo[] infos, int[] lightmapOffsetIndex, LightInfo[] lightsInfo)
{
    // レンダラー情報をループして、各レンダラーにライトマップ情報を適用
    foreach (RendererInfo info in infos)
    {
        // ライトマップのインデックスを更新
        info.m_Renderer.lightmapIndex = lightmapOffsetIndex[info.m_LightmapIndex];
        // ライトマップのUVスケールとオフセットを更新
        info.m_Renderer.lightmapScaleOffset = info.m_LightmapOffsetScale;
    }
}
```

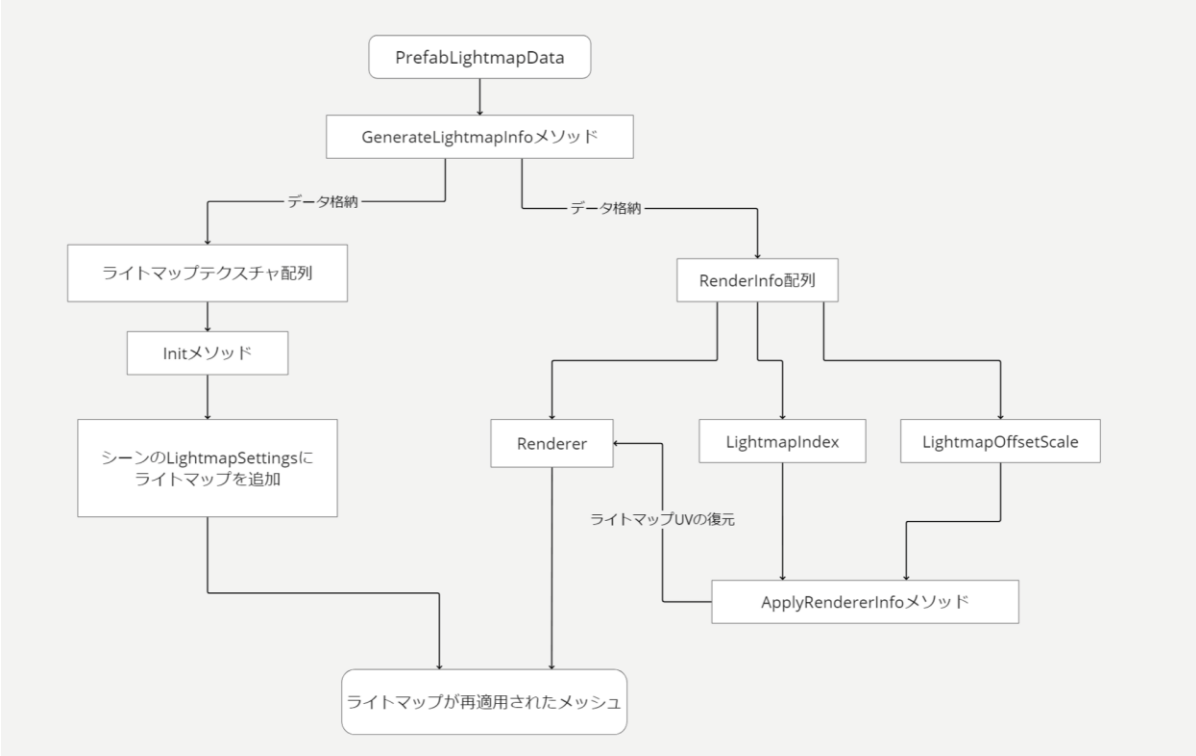


図2-2-39　ライトマップの適用イメージ

PLATEAU SDK 技術解説書 第1.0版
Technical Reference for PLATEAU SDK

令和6年3月　発行
国土交通省 都市局

PLATEAU
by MLIT