

# Topic XX 3D都市モデルを使った位置情報共有ゲームを作る

---

近年位置情報を基盤としたサービスが多く使われています。Google MapsやUberなどはその典型でしょう。自分の位置情報をスマートフォン端末のGPSで取得し、サービスのサーバーに送信することで、さまざまな便利なサービスを受けることができます。本トピックでは、このような位置情報をオンラインでやり取りしたり、サーバーで処理してサービスを提供するなどの参考になるように、例としてPLATEAUの3D都市モデルを活用した位置情報ゲームを作りながら説明します。

## まずは基本のゲームを作る

本トピックではUnityでスマートフォンAR位置情報ゲームを作ります。ゲームを起動すると、プレイヤーは最初に赤緑青の三色から選び自分が属する陣営を決めます。AR画面になると、端末カメラが映す実際の街並みが表示されます。PLATEAUの3D都市モデルを使い、タップするとその場所の建物に「ペンキ」が塗られ、町中の色々なところを塗って最初の点までつないで閉じると、そこが自陣営の陣地になります。ゲームは獲得した陣地の面積を競います。

サーバーサイドにはPostGIS拡張をインストールしたPostgreSQLデータベースを使います。面積は、ゲームプレイ時はとった陣地の合計をサーバーで計算しますが、ゲーム終了後にバッチ処理で重ね合わせを考慮した結果の面積を計算する仕様にします。これは、時間順に重ね合わせて面積を計算する処理が、リアルタイムに計算にするには重いためです。

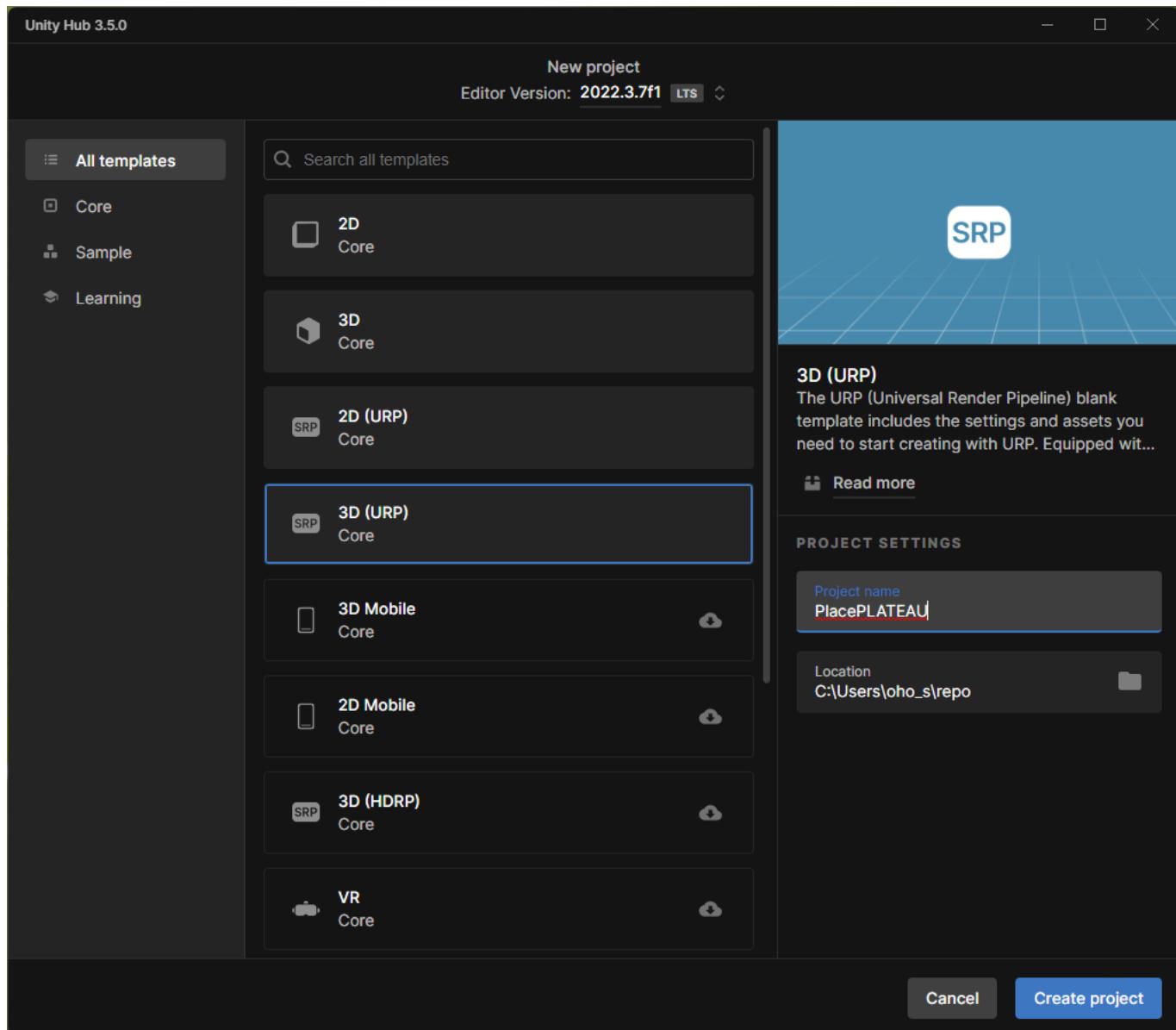
## ARの設定をする

最初にプロジェクトを作成し、基本のAR部分を設定します。本プロジェクトでは、iOS・Androidを対象として、ARFoundationおよびARCoreExtentionでGeospatial APIを使います。

まずはUnityで新規プロジェクトを作成します。今回は、Unity 2022.3.7f1を使います。

ARの設定は、PLATEAU公式サイトのチュートリアルコンテンツの[TOPIC14-3 「Google Geospatial APIで位置情報による3D都市モデルのARを作成する」](#)も参考にしてください。また、UnityでのAR開発の[公式ドキュメント](#)や、ARFoundationの[公式ドキュメント](#)なども必要に応じて参照してください。ここでは、最低限の説明をします。

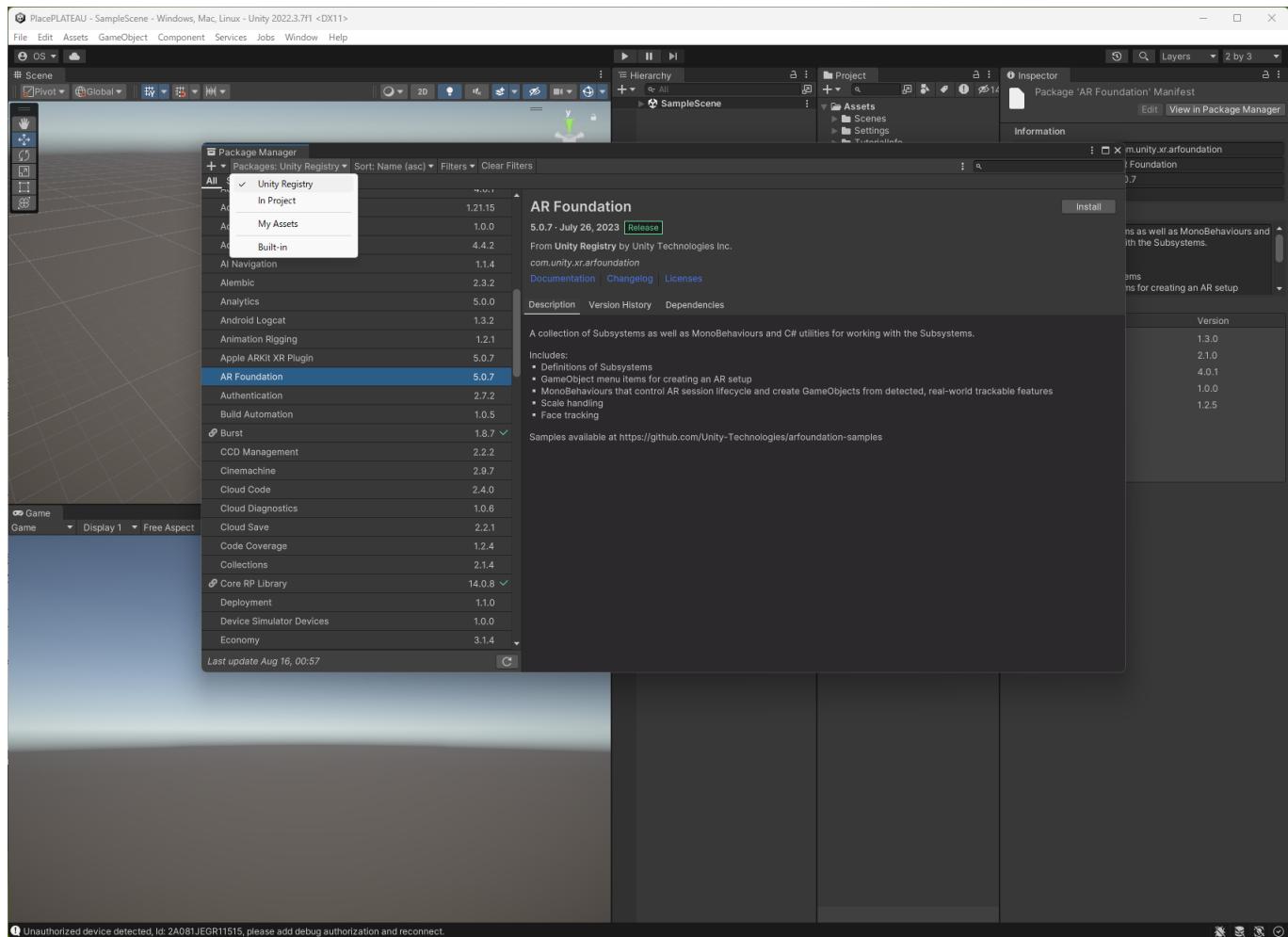
まず、Unity Hubからプロジェクトを新規作成します。今回は、「3D(URP)」を選択します。必要に応じてダウンロードボタンを押してテンプレートをダウンロードしてください。プロジェクト名を「PlacePLATEAU」にします。PLATEAUを使った陣取りゲームからの命名です。



プロジェクトが立ち上がったら、ARFoundationとGepspatialAPIを導入します。

[Window] メニューから [Package Manager] を開きます。

Package Managerが開いたら、「Unity Registry」を選択し、「ARFoundation」をインストールします。何らかのダイアログが開いたら適切に対処します。「New Input System」の有効化が必要であればダイアログの指示にしたがい、再起動などを起こします。

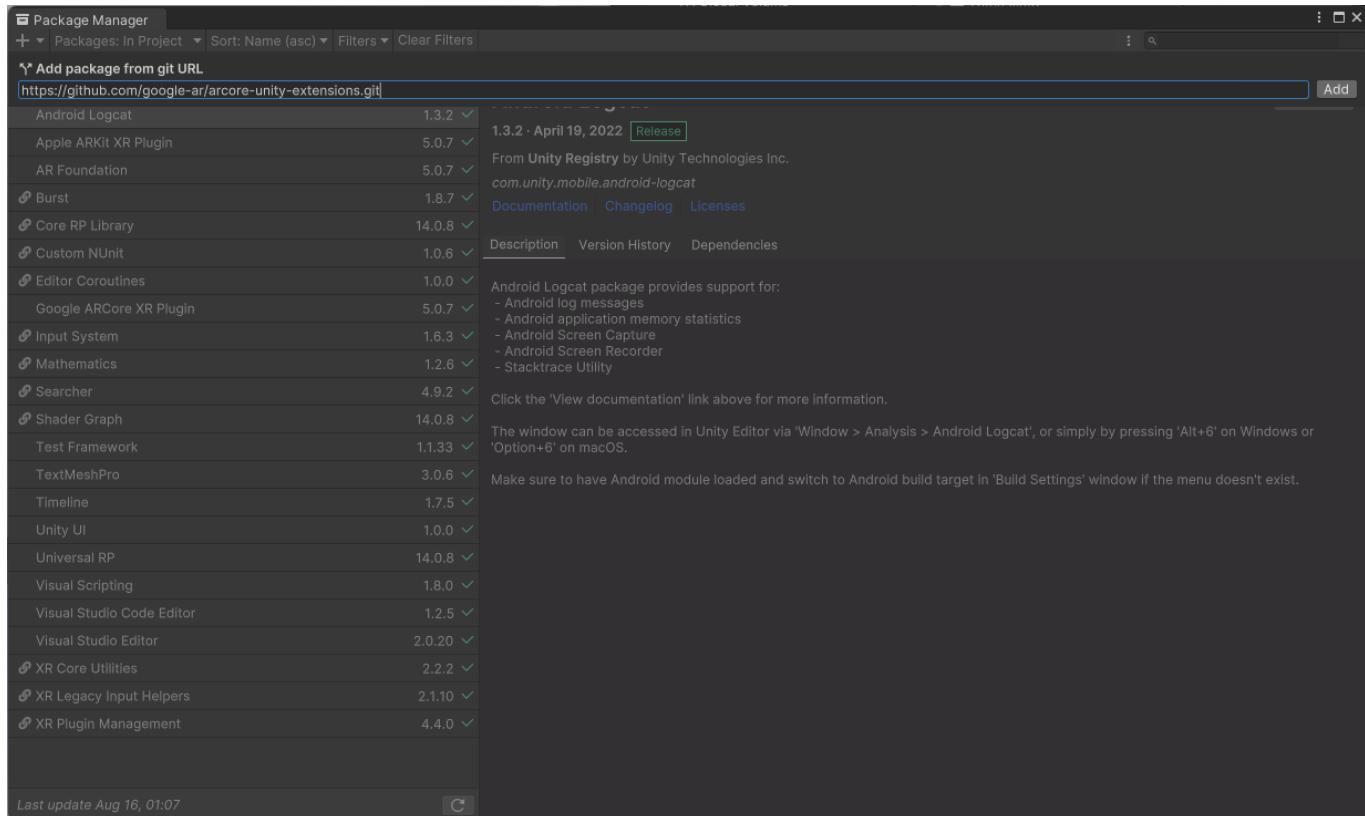


同様にして、「Apple ARKit XR Plugin」、「Google ARCore XR Plugin」をインストールします。

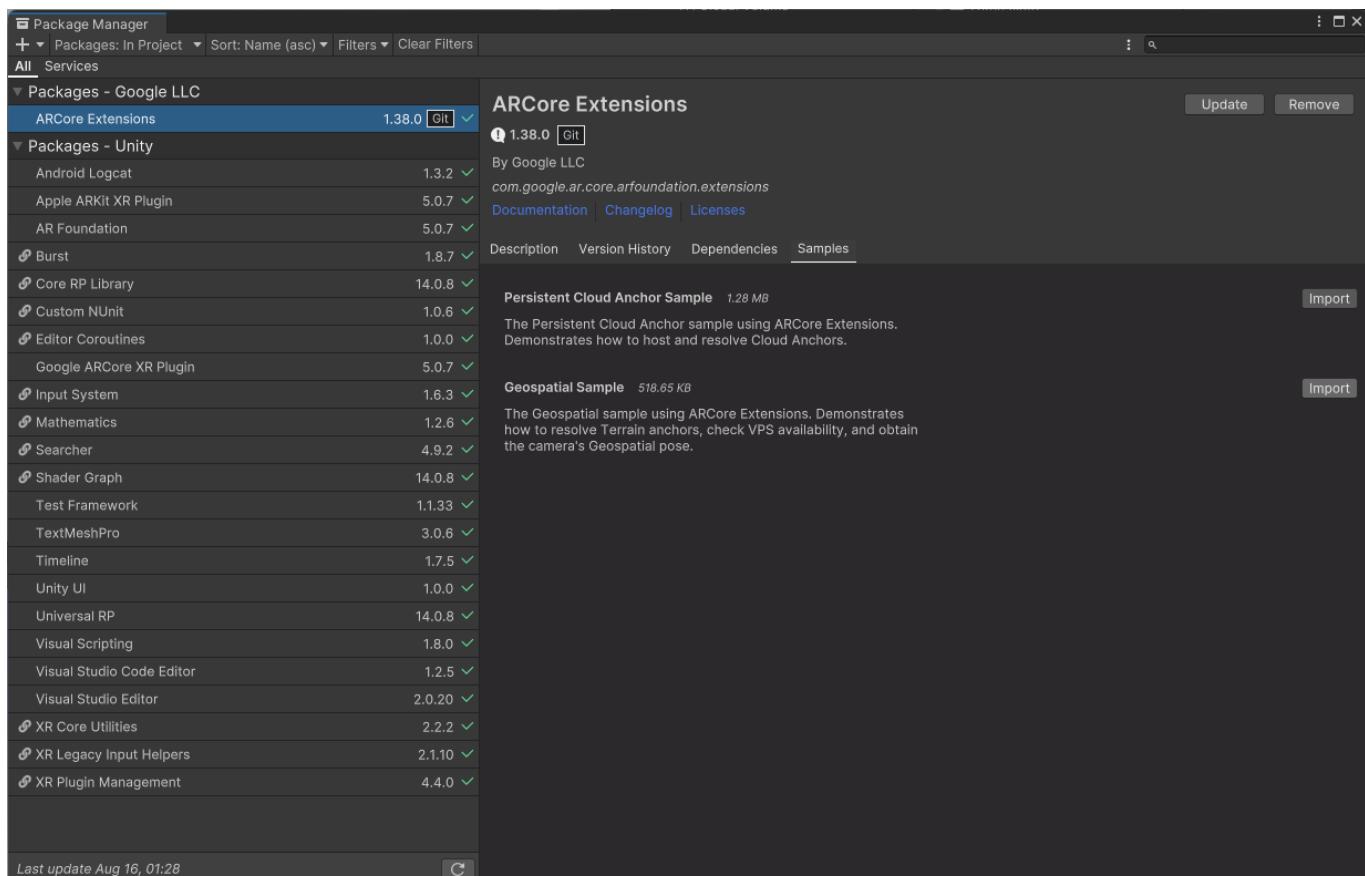
この段階で必要に応じてインストール済みのパッケージのアップデートも行っておくとよいでしょう。

次に、「ARCore Extentions for ARFoundation」をインストールします。このパッケージにGeospatialAPIの機能が含まれます。なお、ARCore Extentionsのドキュメントと、GeospatialAPIのドキュメントも参考にしてください。Package Managerの左上の [+] をクリックし、[Add package from git URL] を選択します。そして次のURLを貼り付け、[Add] をクリックします。

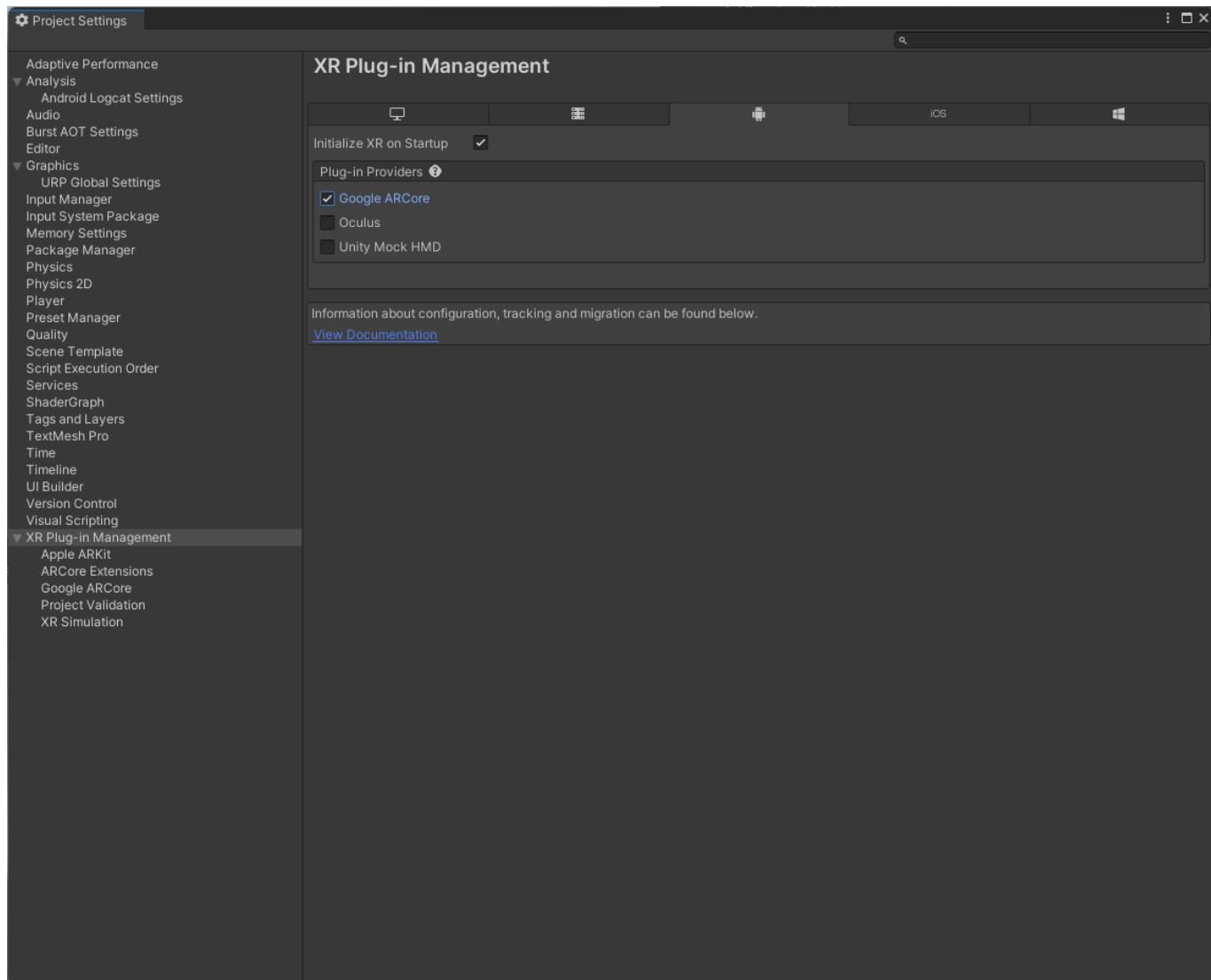
```
https://github.com/google-ar/arcore-unity-extensions.git
```

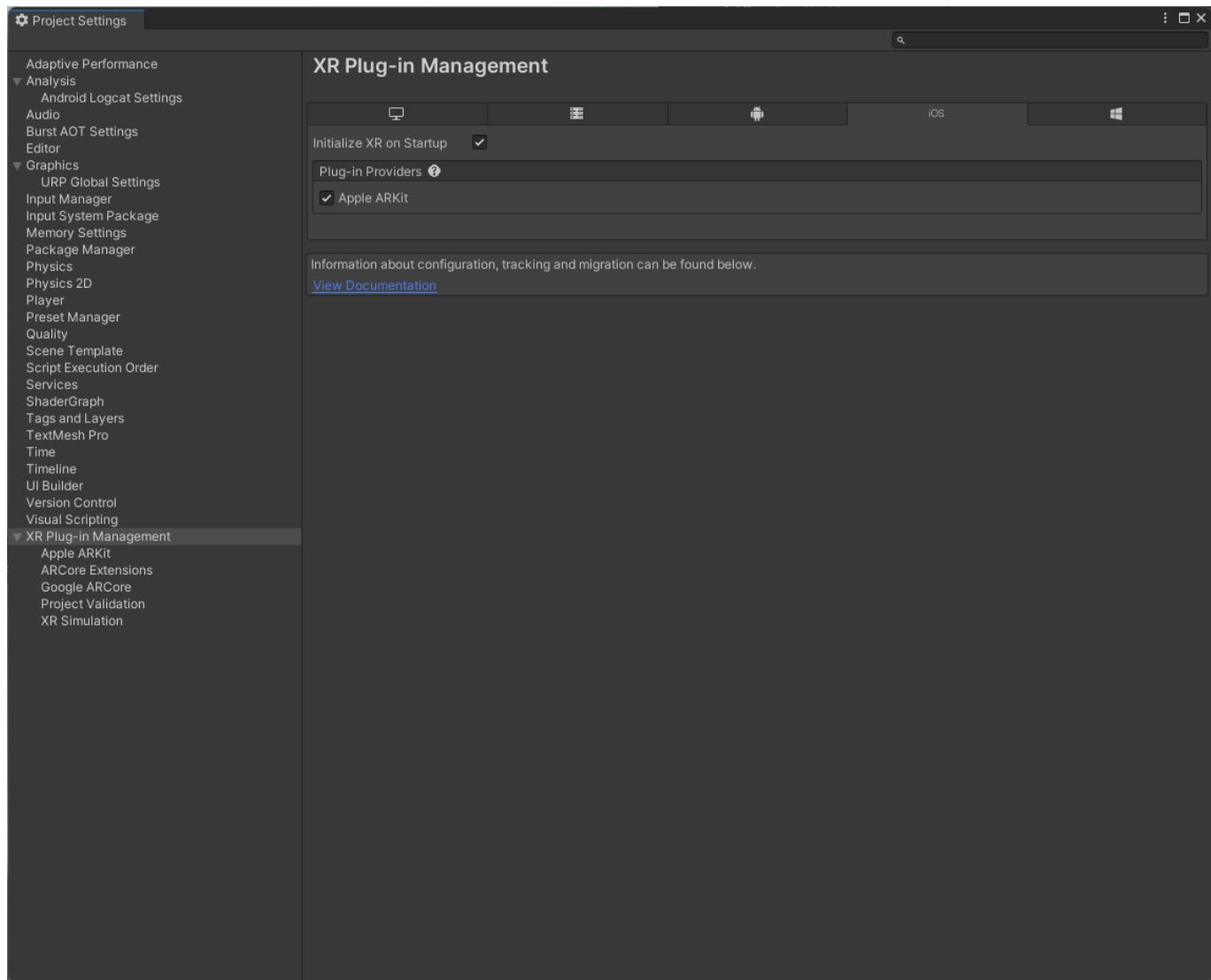


ARCore Extensionsの画面が表示されたら、[Samples] の項目にある [Geospatial Sample] の [Import] をクリックして、このサンプルもインポートしておいてください。

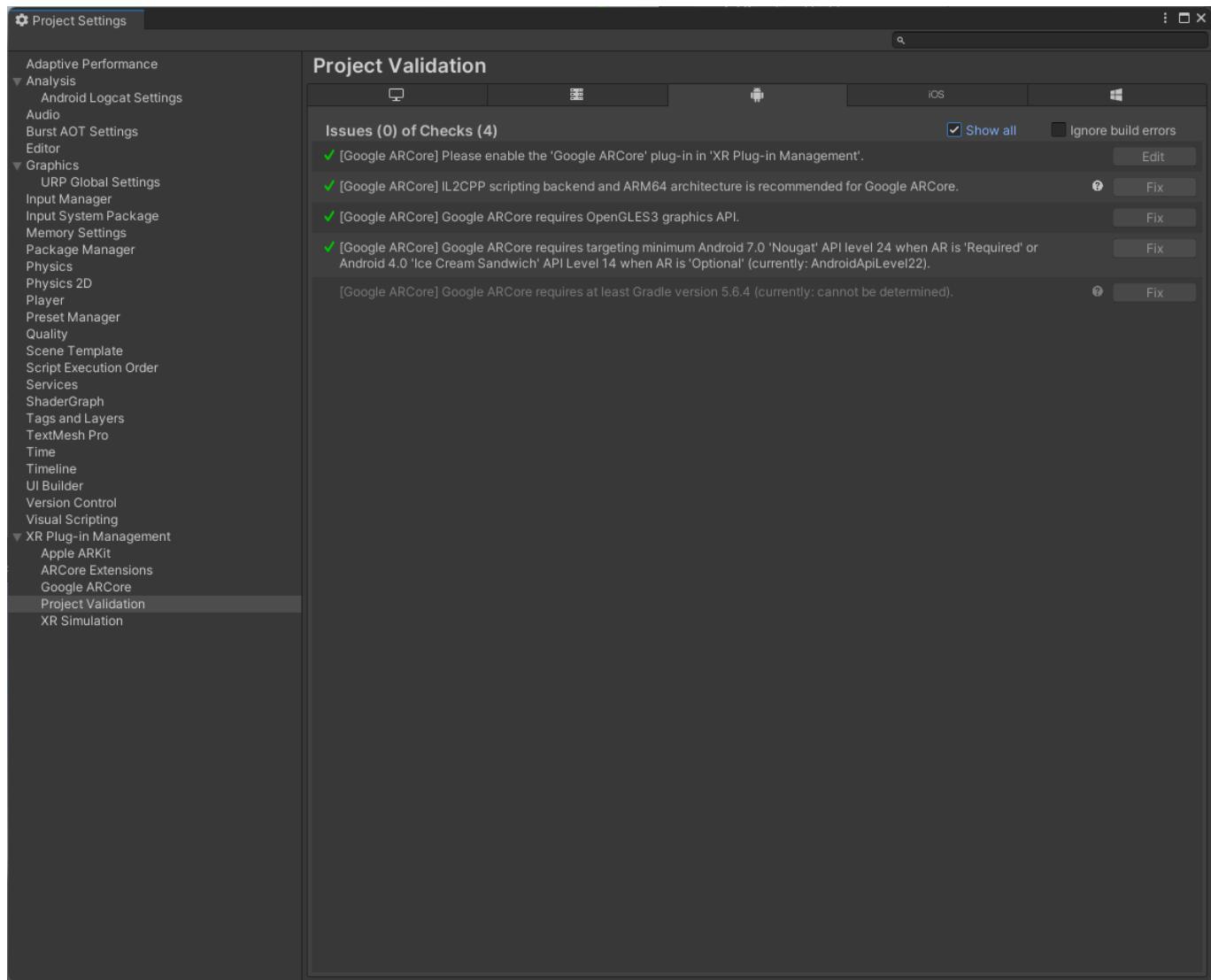


さらに、Unity Editorの[Edit]メニューから[Project Settings]を開き、いくつかの設定を加えます。

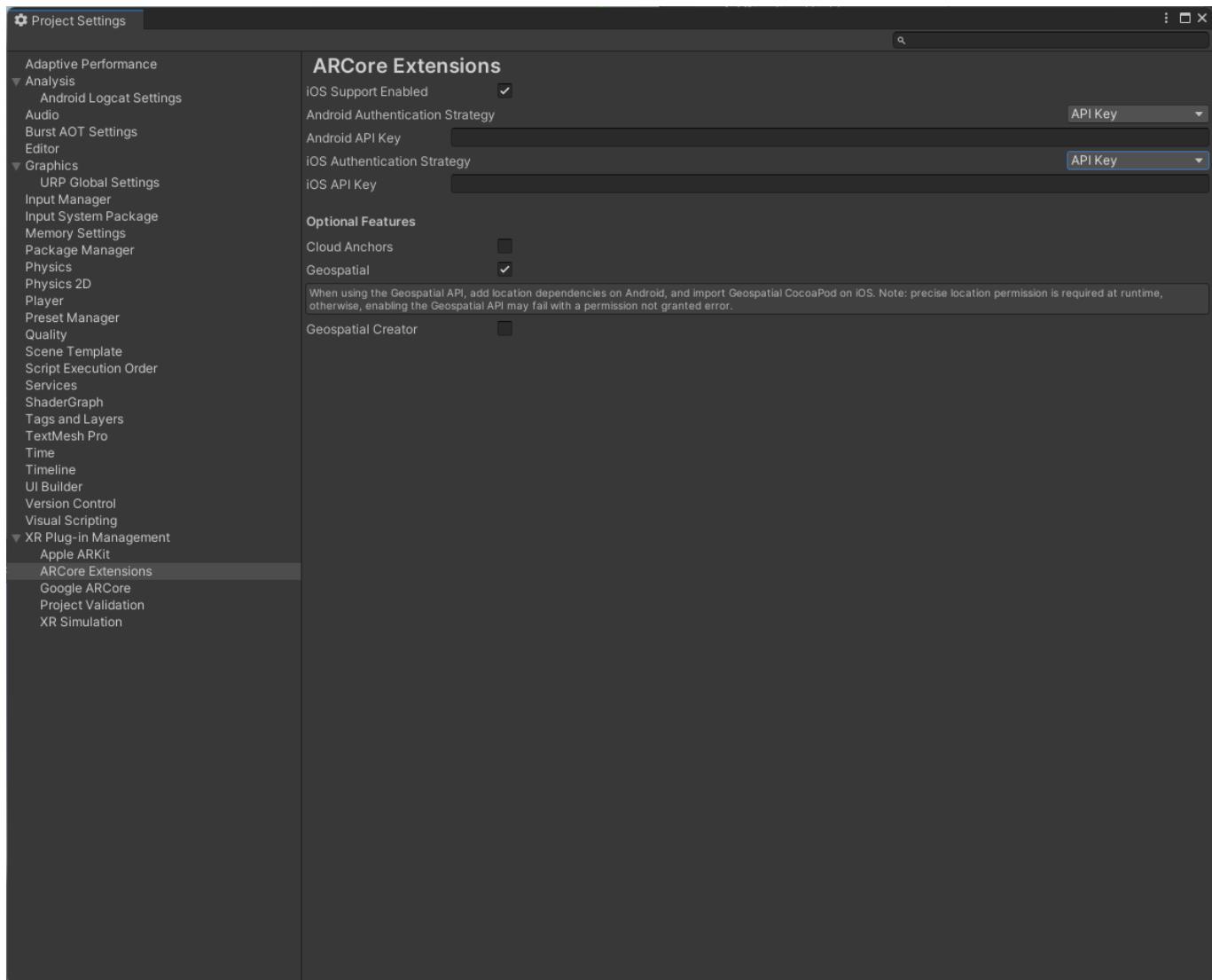




「XR Plug-in Management」の「Project Validation」では、プロジェクトの設定を自動的にARに最適な形に修正してくれます。もし、ここでエラーなどが出ていたら、「Fix All」ボタンを押して修正します。



Geospatial API の API キーは Google Cloud のコンソールで作成し、「XR Plug-in Management」の「ARCore Extentions」で設定してください。詳細な手順は [公式ドキュメント](#) や、PLATEAU 公式サイトのチュートリアルコンテンツの [TOPIC14-3 「Google Geospatial API で位置情報による3D都市モデルのARを作成する」](#) の「■ API キーの作成」の項目を参照してください。また、「iOS Support Enabled」「Geospatial」のチェックもつけておきます。

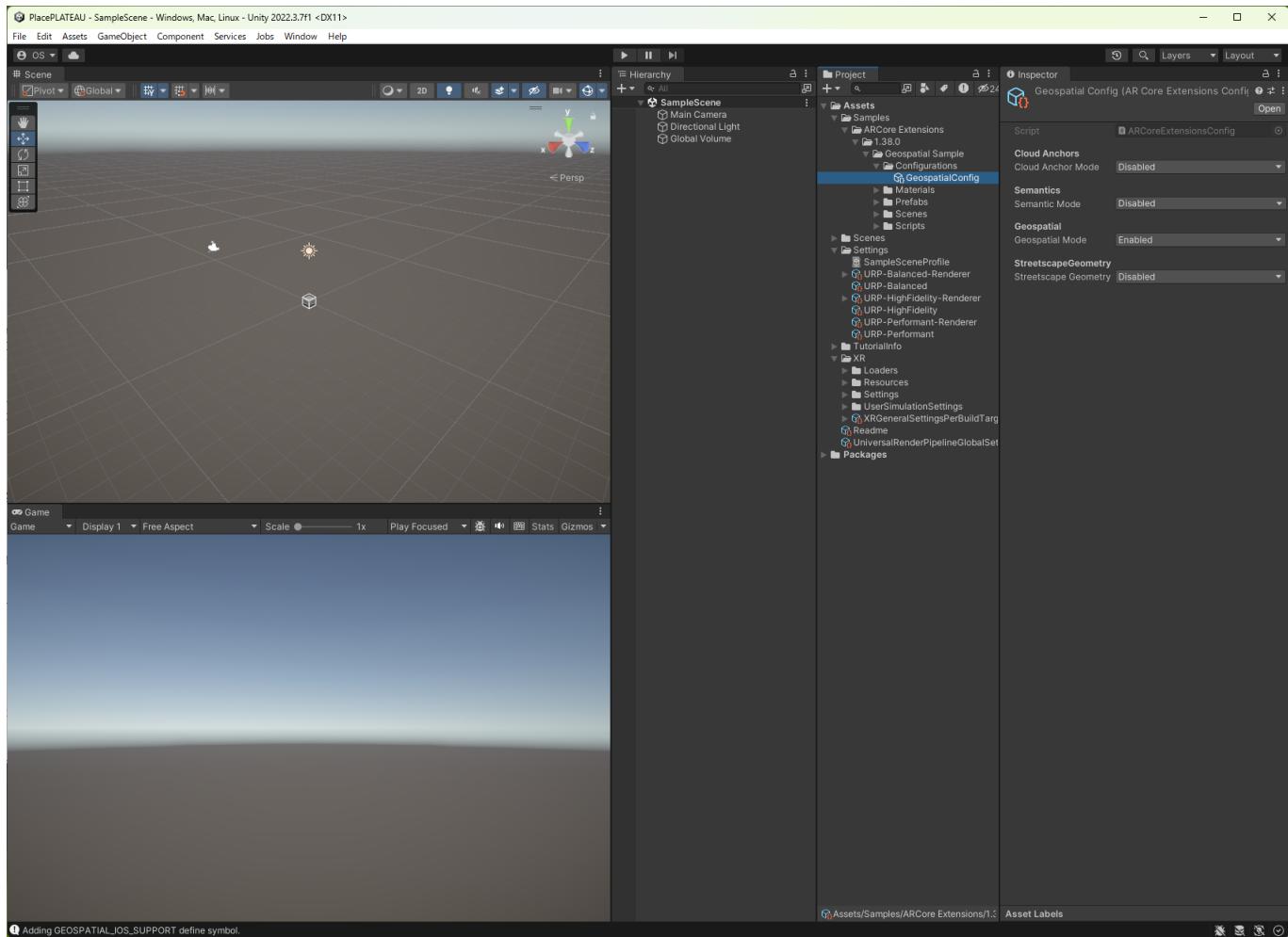


※※※ 注意 ※※※

ここではAPIキーを設定していますが、秘密情報をアプリに組み込むことになるためセキュリティ的には推奨されません。  
後半のアプリをビルドする際に適切な方法の説明をします。  
APIキーのままでアプリを公開しないように気を付けてください。

次に、Unity Editorの「Project」ビューにうつり、「Assets/Samples/ARCore Extentions/1.38.0/Geospatial Samples/Configurations/GeospatialConfig」をクリックします。「Inspector」ビューの「Geospatial」を

「Enabled」にします。



Playerタブで、Android,iOSの各プラットフォームを以下のように設定します。

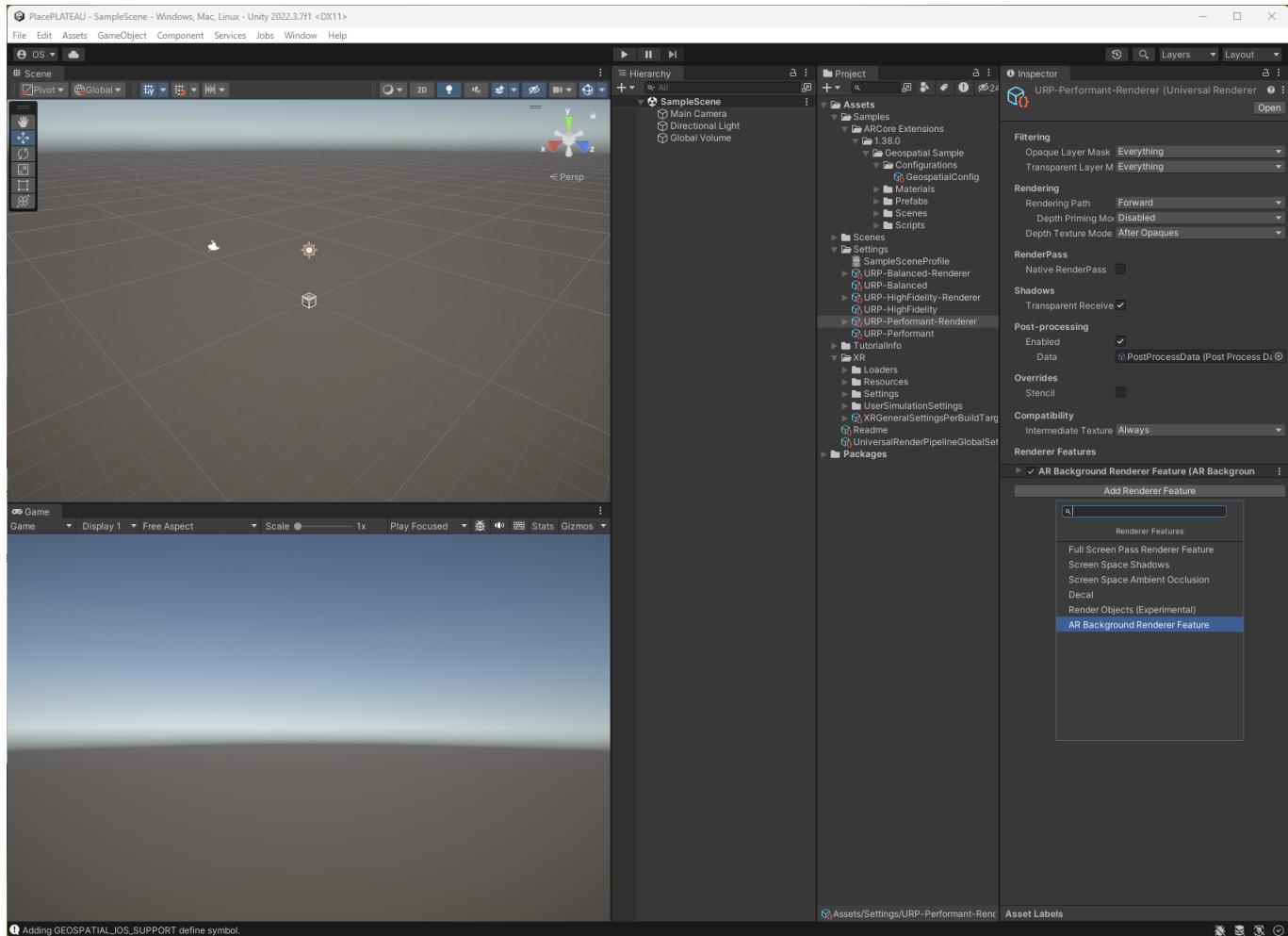
#### 【Androidの場合】

- Minimum API Levelは、28以上が妥当です。
- IL2CPPでビルドします。
- Graphics APIは、OpenGL ES3以上が必要です。（Vulkanは外しておいた方が無難です）
- Target ArchitectureのARMv7のチェックを外す

#### 【iOSの場合】

- サポートするOSは、iOS11以上です。
- ARM64でビルドします。
- 「Require ARKit Support」にチェックを入れます。
- 「Location Usage Description」に、位置情報を使う際にユーザーに通知するメッセージの文字列を入れます。

URPの設定で、Renderer FeaturesにAR Background Renderer Featureが設定されていない場合、追加します。使用するURP設定が分からなければ、すべての設定に追加します。



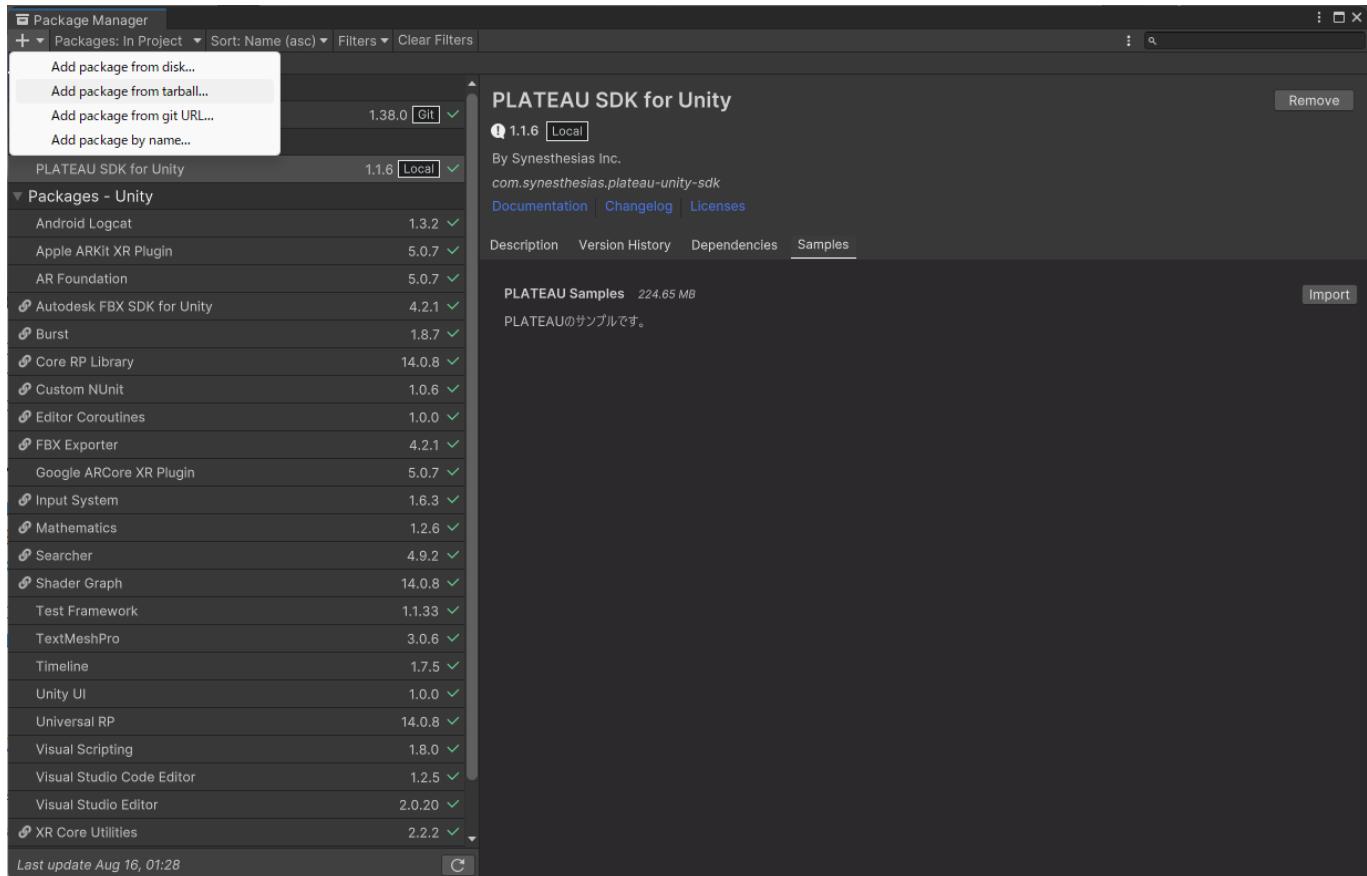
ここまでで、Geospatial APIを使ったARアプリの基本の設定が終わりました。一度各プラットフォーム向けに「Geospatial Samples」の「Geospatial」シーンをビルド対象にして動作確認のためにビルドしておくとよいでしょう。

## PLATEAUを読み込む

「PLATEAU SDK for Unity」をインストールし、ゲーム対象地域の3D都市モデルを読み込みます。詳細は、公式の[ドキュメント](#)や、PLATEAU公式サイトのチュートリアルコンテンツの[TOPIC 17 | PLATEAU SDKでの活用\[1/2\] | PLATEAU SDK for Unityを活用する](#)なども参考にしてください。

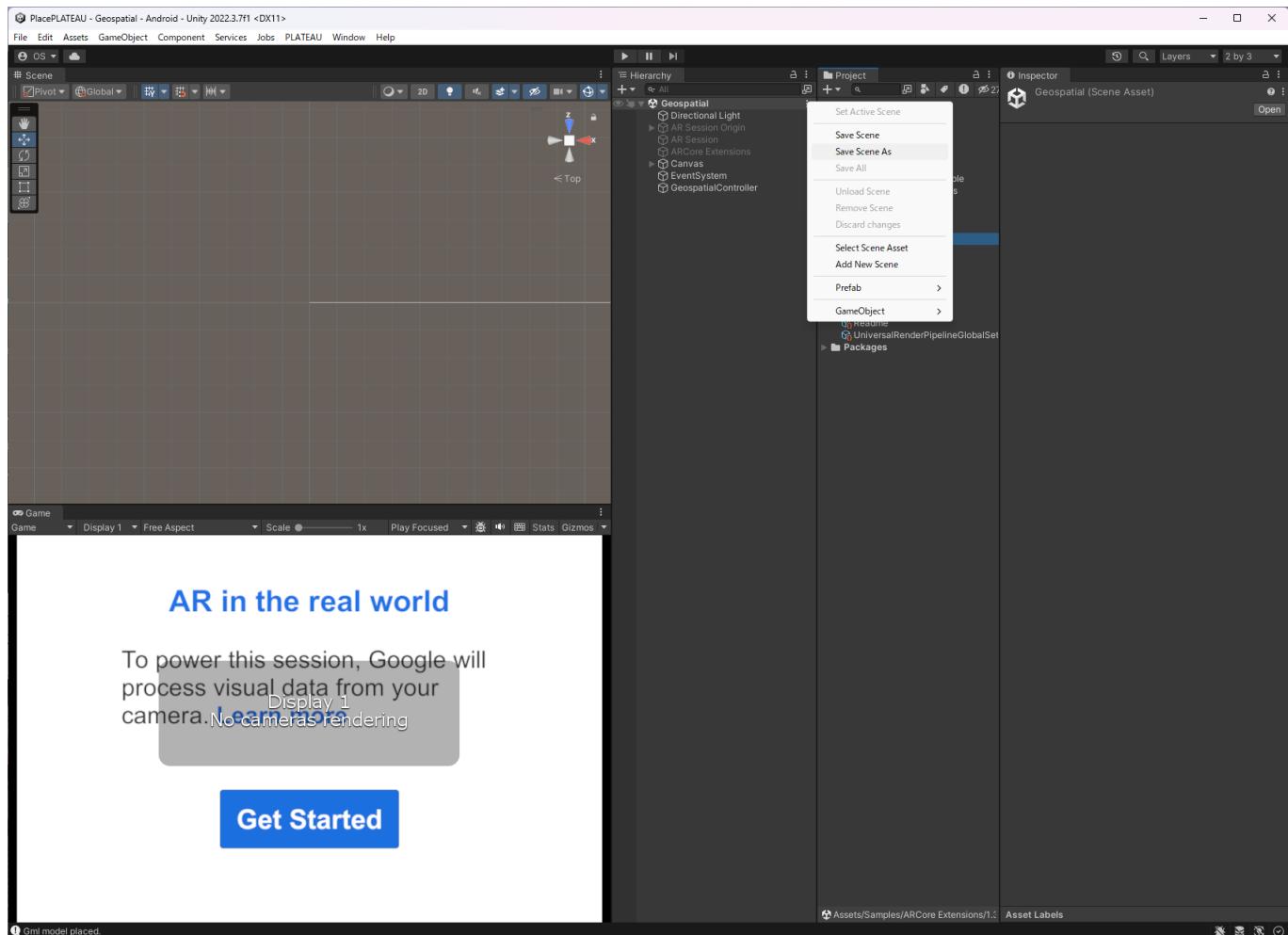
ここでは、GitHubからtarボールでインストールします。「PLATEAU SDK for Unity」のリリースページからtgzファイルをダウンロードします。

Unity Package Managerを開き、左上の+ボタンから「Add Package from tarball...」を選択し、ダウンロードしたtgzファイルを指定してインストールします。

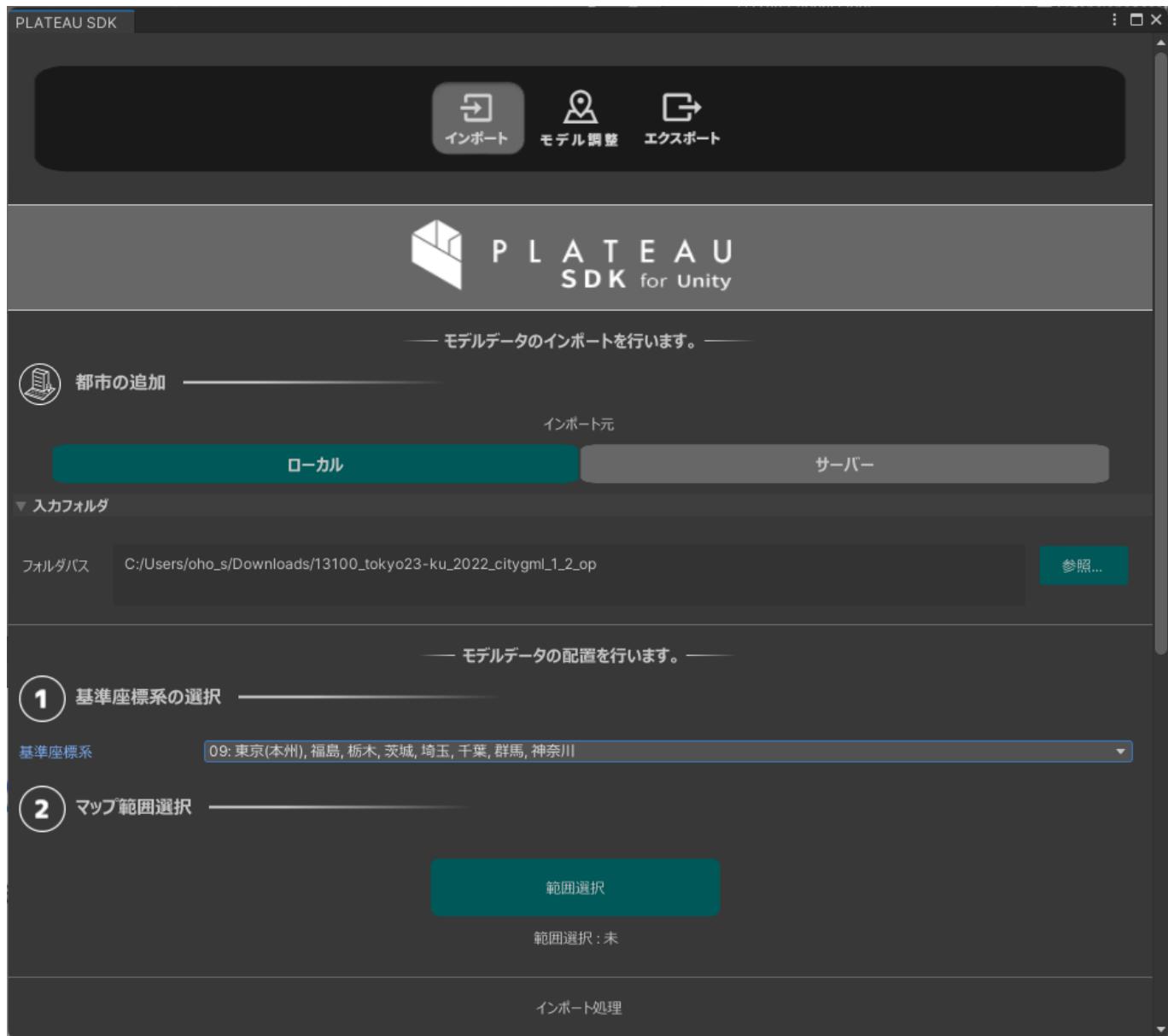


インストールが正常に行われると、Unity Editorのメニューに「PLATEAU」の項目が増えていきます。早速3D都市モデルを読み込んでみます。

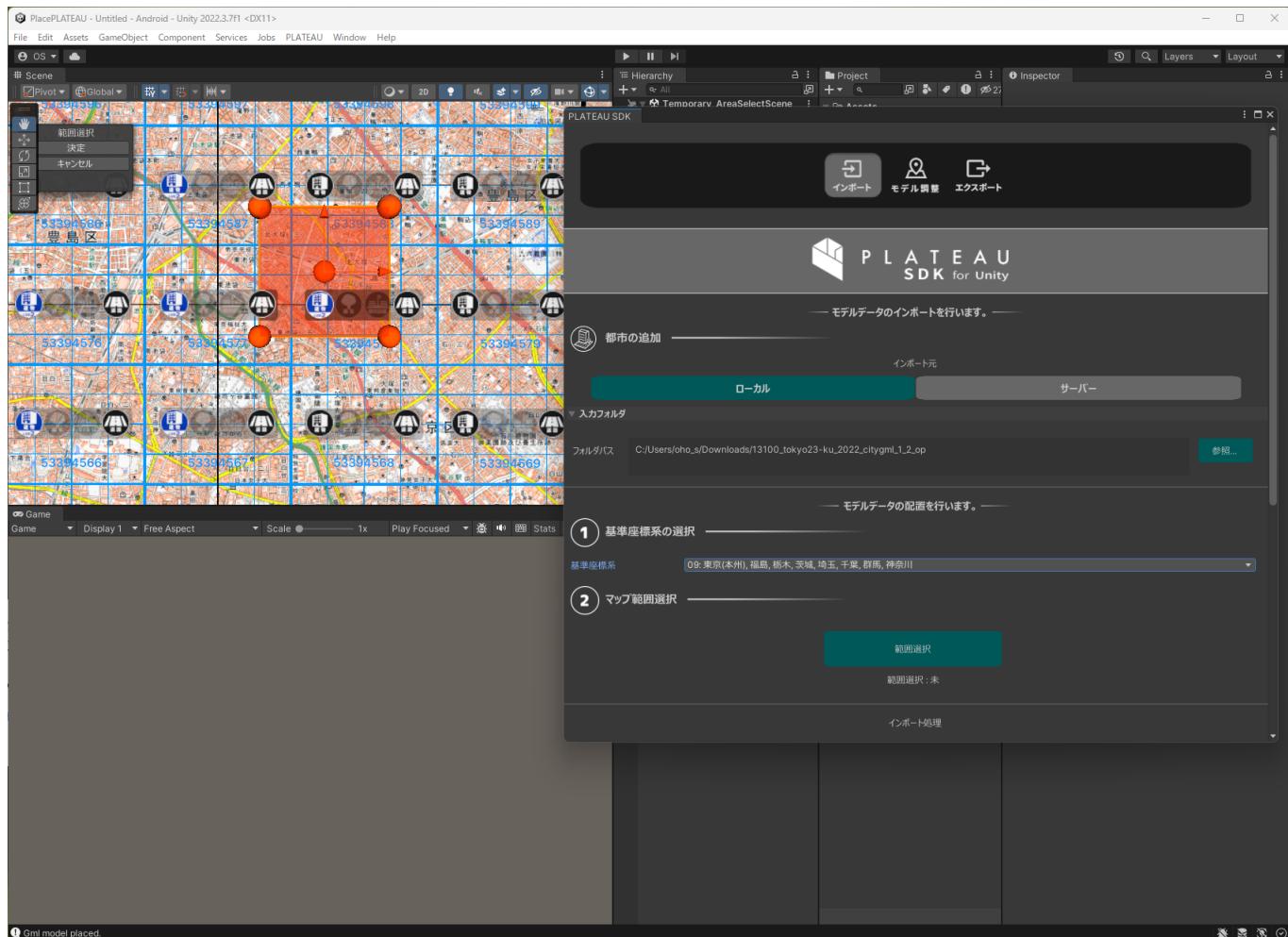
まず、作業用のシーンを用意します。[Assets/Samples/ARCore Extensions/1.38.0/Geospatial Samples/Scenes/Geospatial](#)シーンを開きます。「ヒエラルキー」のシーン名の右からメニューを開くと、シーンを別名で保存する選択肢があるので、[Assets/Scenes](#)などに保存します。（Geospatial Samplesのサンプルシーンをコピーして作業用のシーンとしています。）



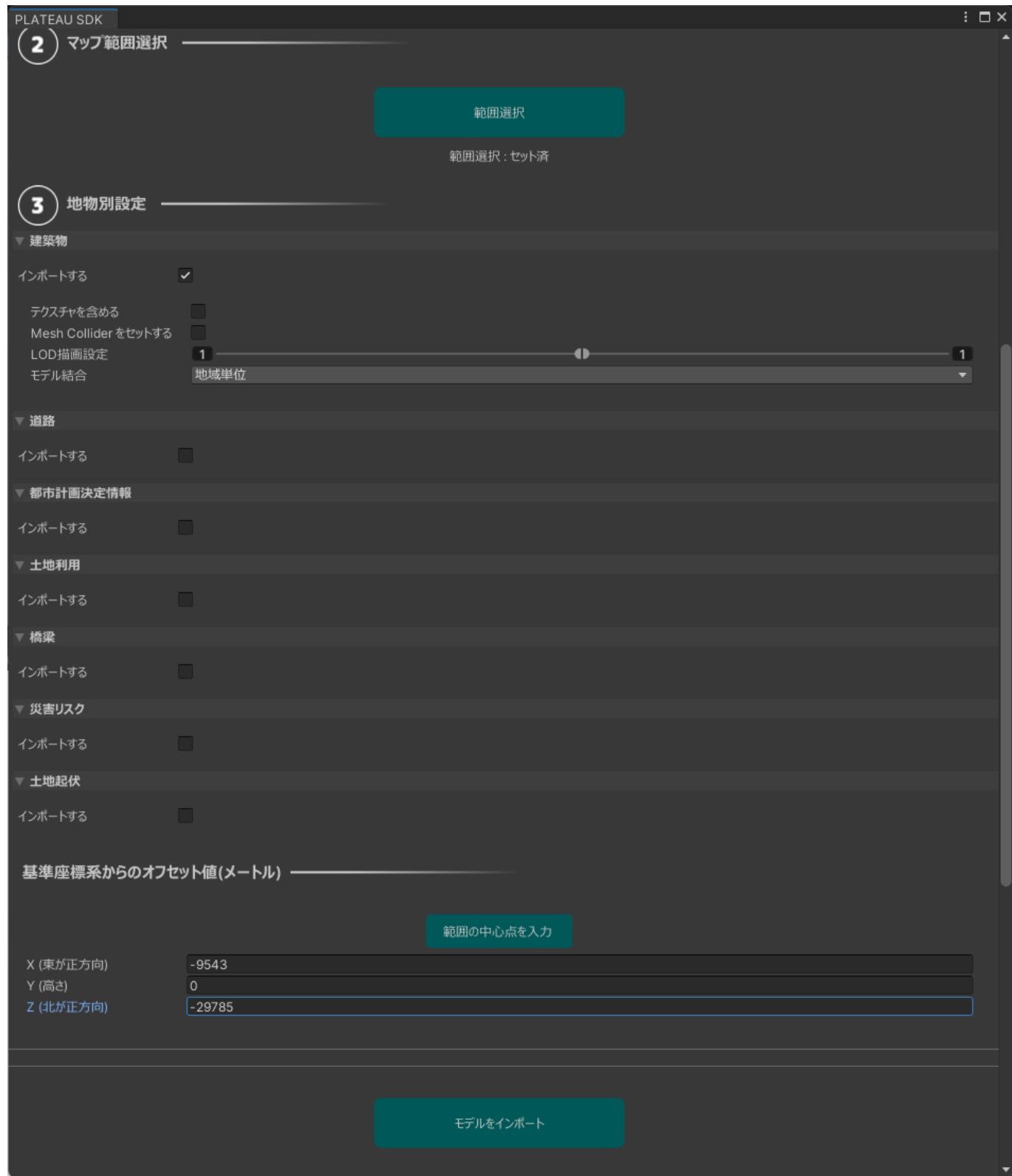
作業用シーンが準備できたら、「PLATEAU」メニューから「PLATEAU SDK」を選択し、SDKのダイアログを表示します。インポートの画面で、ローカルの方であらかじめダウンロードしておいた使いたい地域のPLATEAUの3D都市モデルを展開したフォルダーを選択するか、サーバーの方で使いたい地域を選択します。



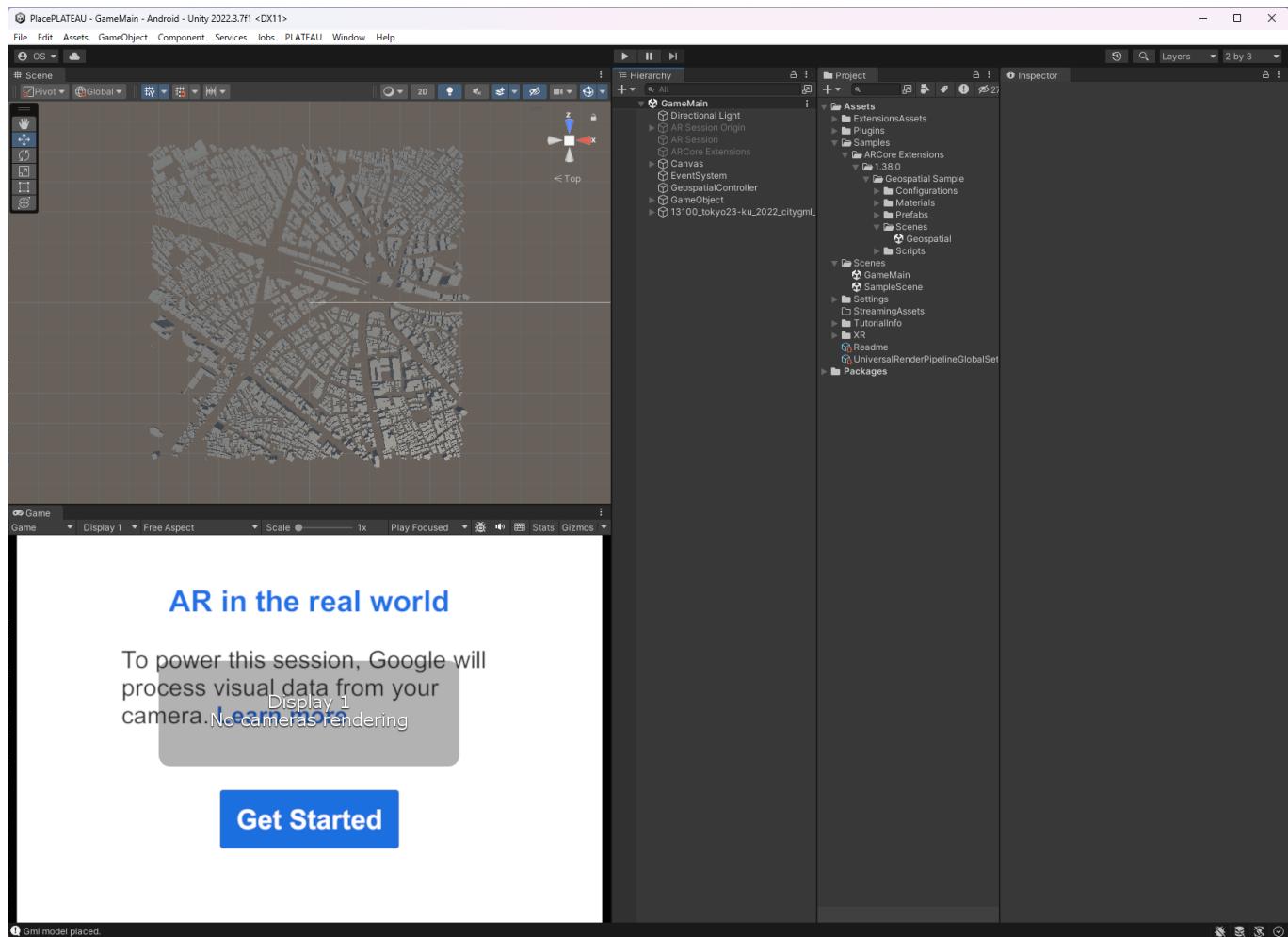
続いて範囲選択を行います。範囲選択ボタンを押すとEditor画面の方で選択UIが起動するので、必要な範囲を指定します。ここでは、あまり広い範囲を指定してしまうと処理負荷がかかるので、気を付けてください。



次に地物別設定をします。今回は、建築物のLOD1のみを「地域単位」で結合してインポートします。建築物以外の「インポートする」のチェックを外します。また、テクスチャは必要ないので「テクスチャを含める」のチェックは外します。これで「モデルをインポート」を押し、処理を開始します。範囲やPCの性能にもよりますが、数分の時間がかかる場合もあります。

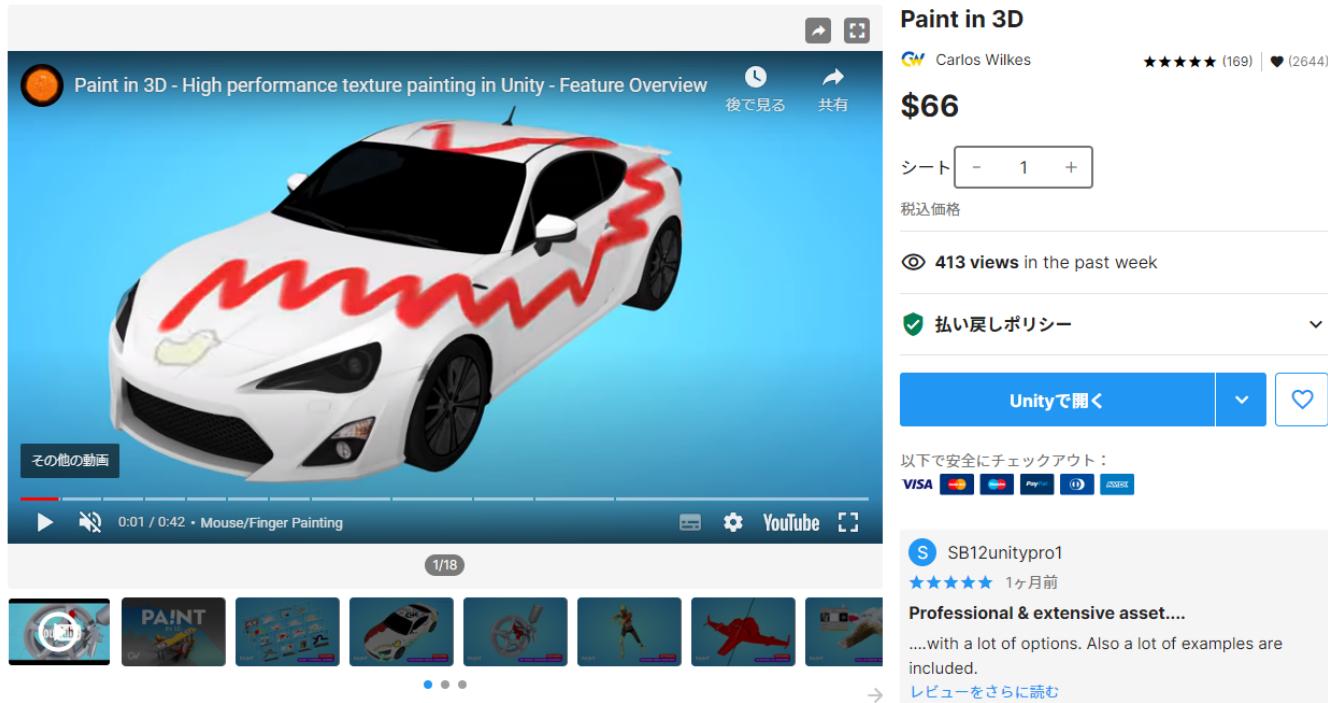


最後に、3D都市モデルがシーンに読み込まれたことを確認します。



## Paint in 3Dを導入し、PLATEAUを塗れるようにする

ここまでがゲーム開発の基本となる準備となります。ここからPLATEAUを使ったゲームロジックを実装します。最初に、ARでPLATEAUの3D都市モデルに弾を発射すると、ビルに印が塗られる仕組みを作ってみます。Unity Asset Storeで「Paint in 3D」というアセットを購入(\$66)し使用します。このアセットは、3Dのオブジェクトにお絵描きができるものです。購入したら通常のアセットの導入と同様にUnity Package Managerから導入してください。

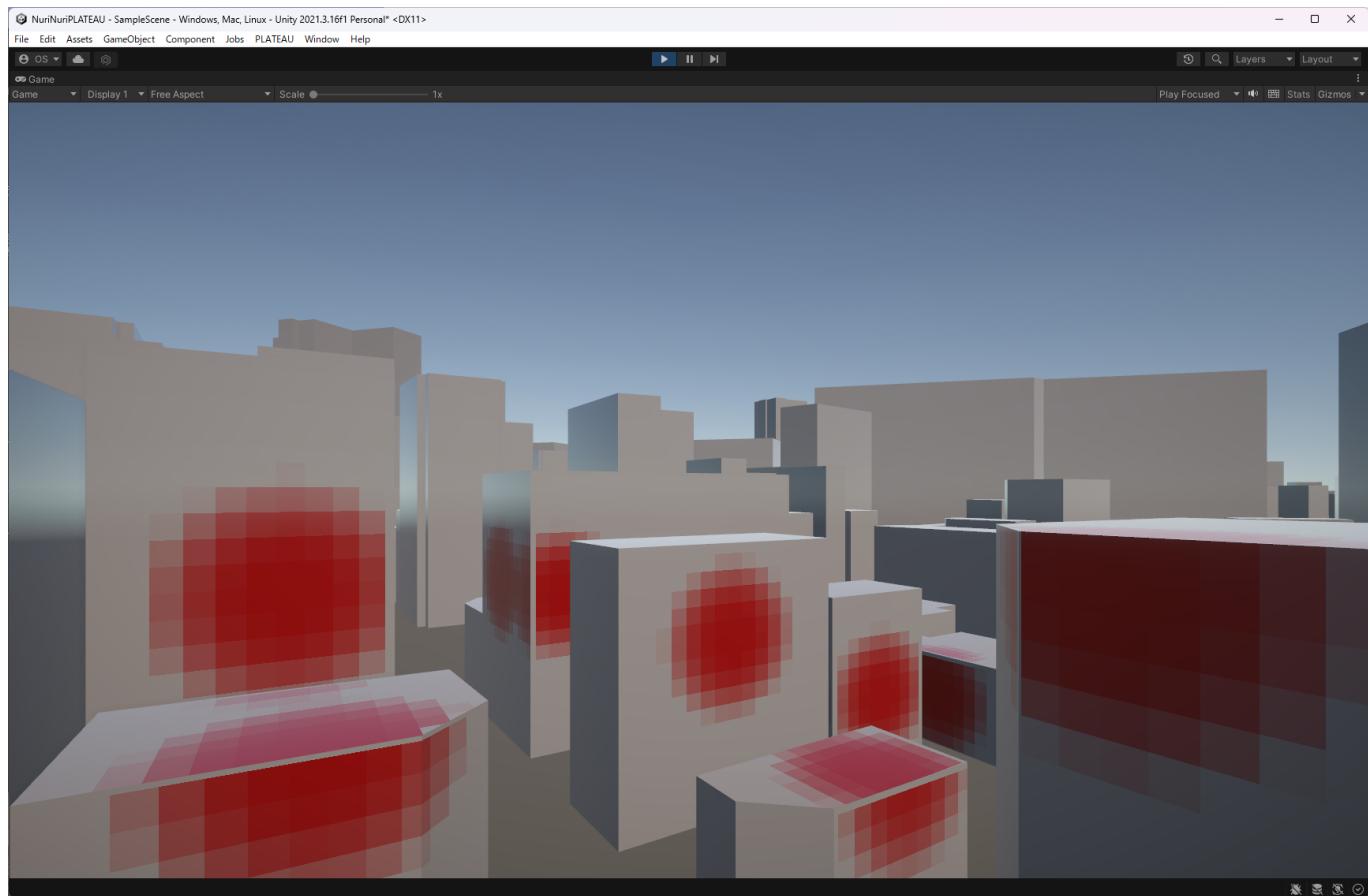


※※※ ヒント ※※※

ここでは、有料アセットを使用しましたが、例えばコライダーとレイキャストの機能を使って、タップ位置からの当たり判定を計算し、そこにオブジェクトを表示するなどでも近いことはできると思います。

興味のある人はチャレンジしてみてください。

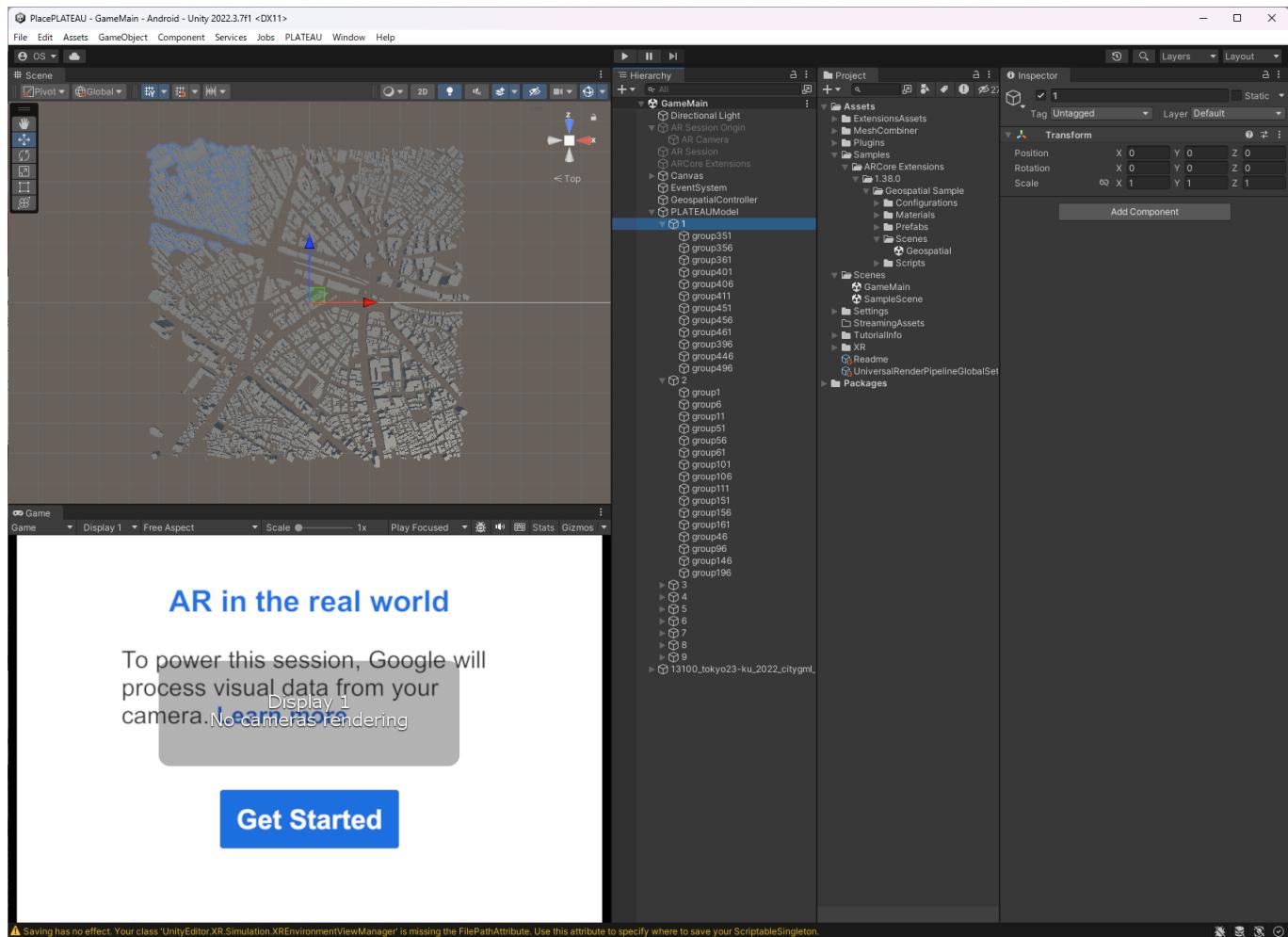
さて、読み込んだままのPLATEAUの3D都市モデルだと、Paint in 3Dで使うためには不便です。Paint in 3Dは、設定したテクスチャ画像に描画することで3Dモデルに描画しているように見せる仕組みです。そのため、広範囲の3Dモデルを結合した巨大なモデルを対象とした場合、テクスチャの解像度が足りずジヤギーが発生したりモザイクのようになってしまいます。一方で、3Dモデルを細かく分割しそれぞれに高解像度のテクスチャを貼ると、描画が非常に高負荷になってしまいます。



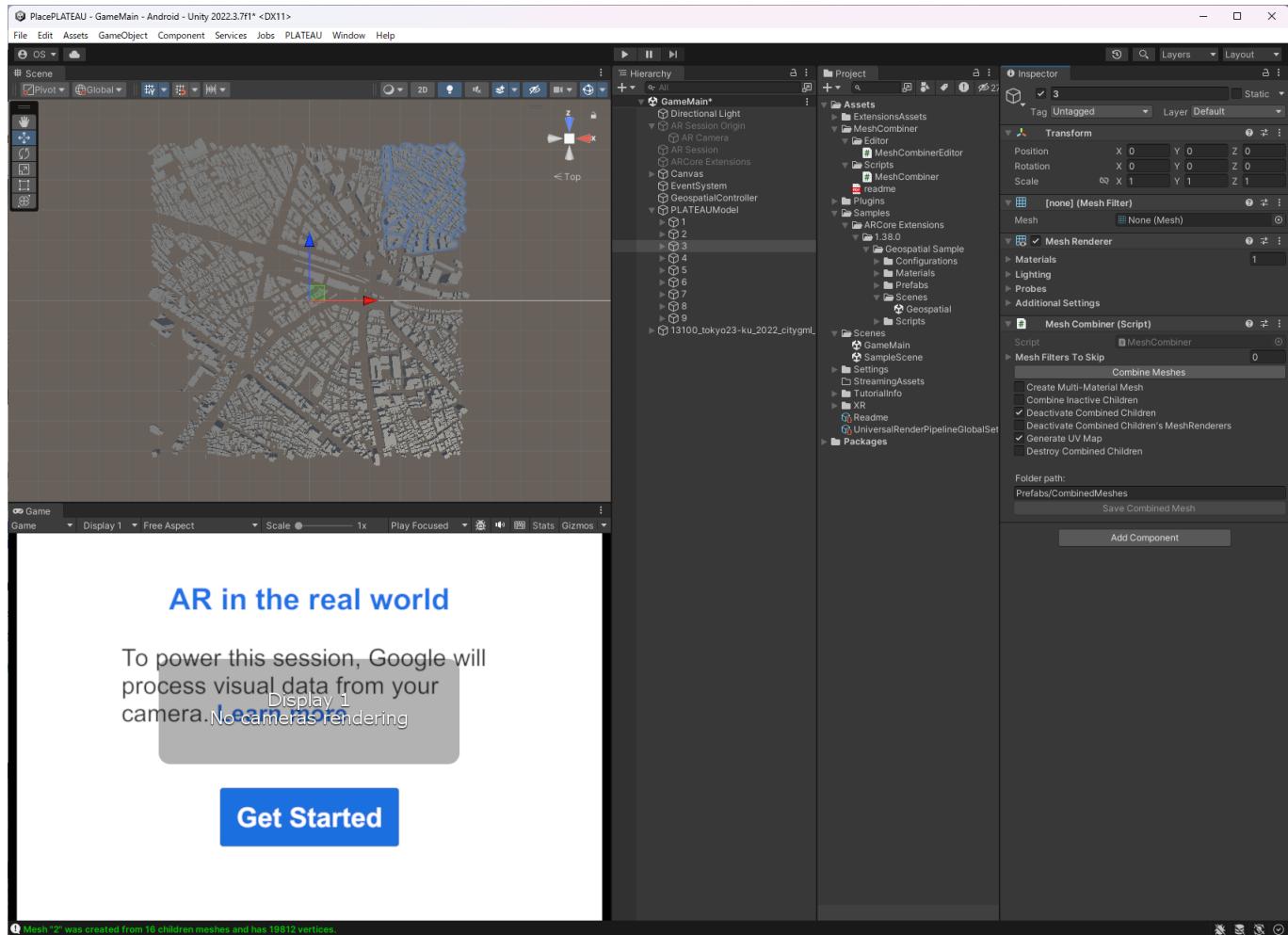
ここでは、読み込んだPLATEAUの3Dモデルを9分割して [「Mesh Combiner」](#) というアセットで結合することで、テクスチャの品質と描画負荷のバランスのよい設定にします。

Mesh CombinerはAsset Storeで入手し、プロジェクトにインストールしておきます。

最初に、Editor上で読み込んだ3D都市モデルをマウスで選択しながらまとめるための空のGameObjectの子にしていき、縦横それぞれ3分割した9つのGameObjectに分けた階層構造にします。

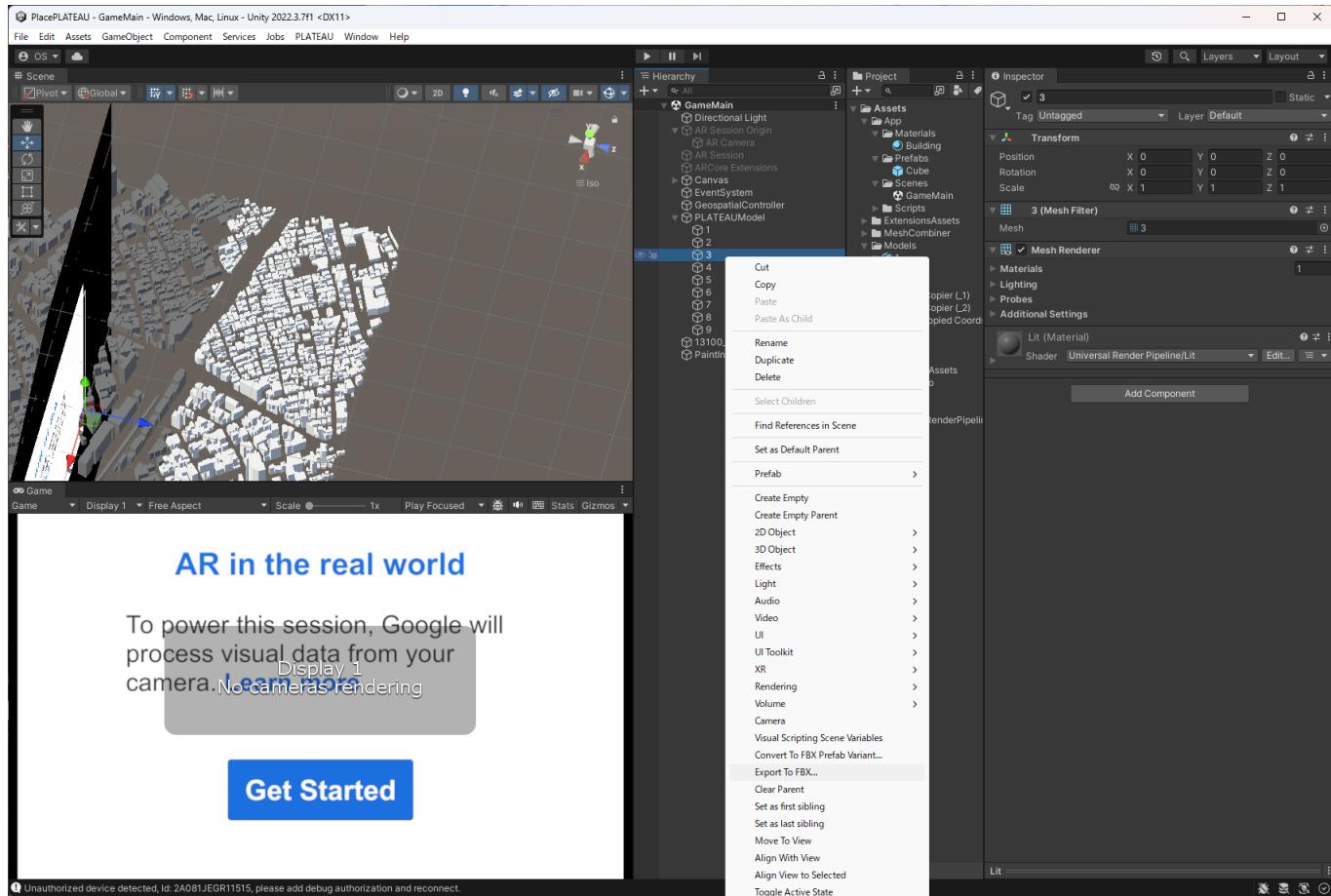


Assets/MeshCombiner/Scripts/MeshCombiner.csを、9分割したそれぞれのGameObjectにD&Dでアタッチします。「Deactivate Combined Children」と「Generate UV Map」にチェックを入れて、「Combine Meshes」を押すと。メッシュが結合されます。正しく結合されていることが分かったら、子オブジェクトは消しても問題ありません。

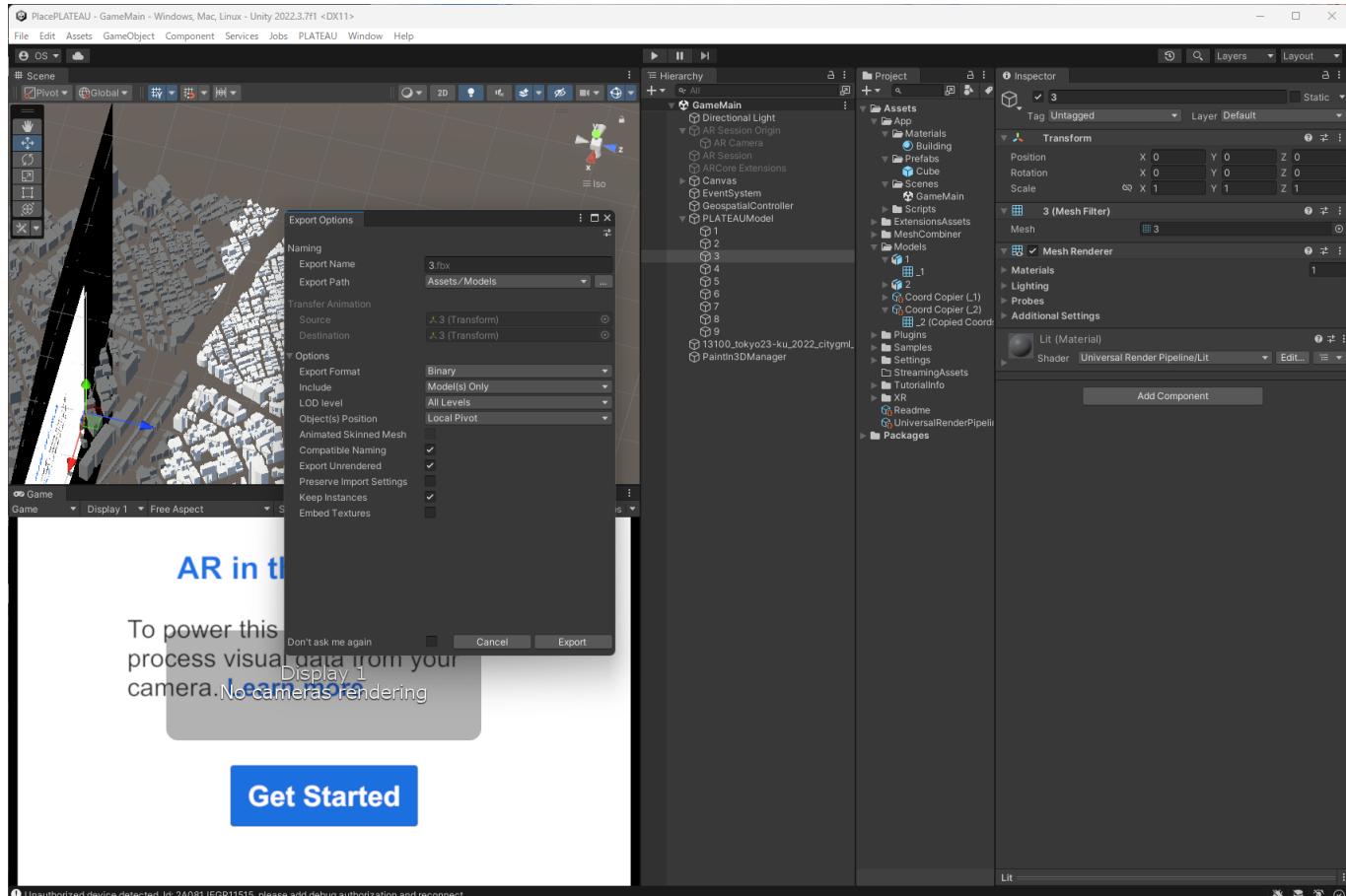


Paint in 3Dは、モデルのUV座標に基づいて、テクスチャのどこにペイントするかを判定します。そのため、モデルに正しくUV座標が設定されていないとおかしな描画になってしまいます。Mesh CombinerでUVは自動的に生成されていますが、現状のMesh Combinerで結合したメッシュだと、UV座標がUV1に入っています。これをPaint in 3Dの機能を使って、UV0にコピーします。

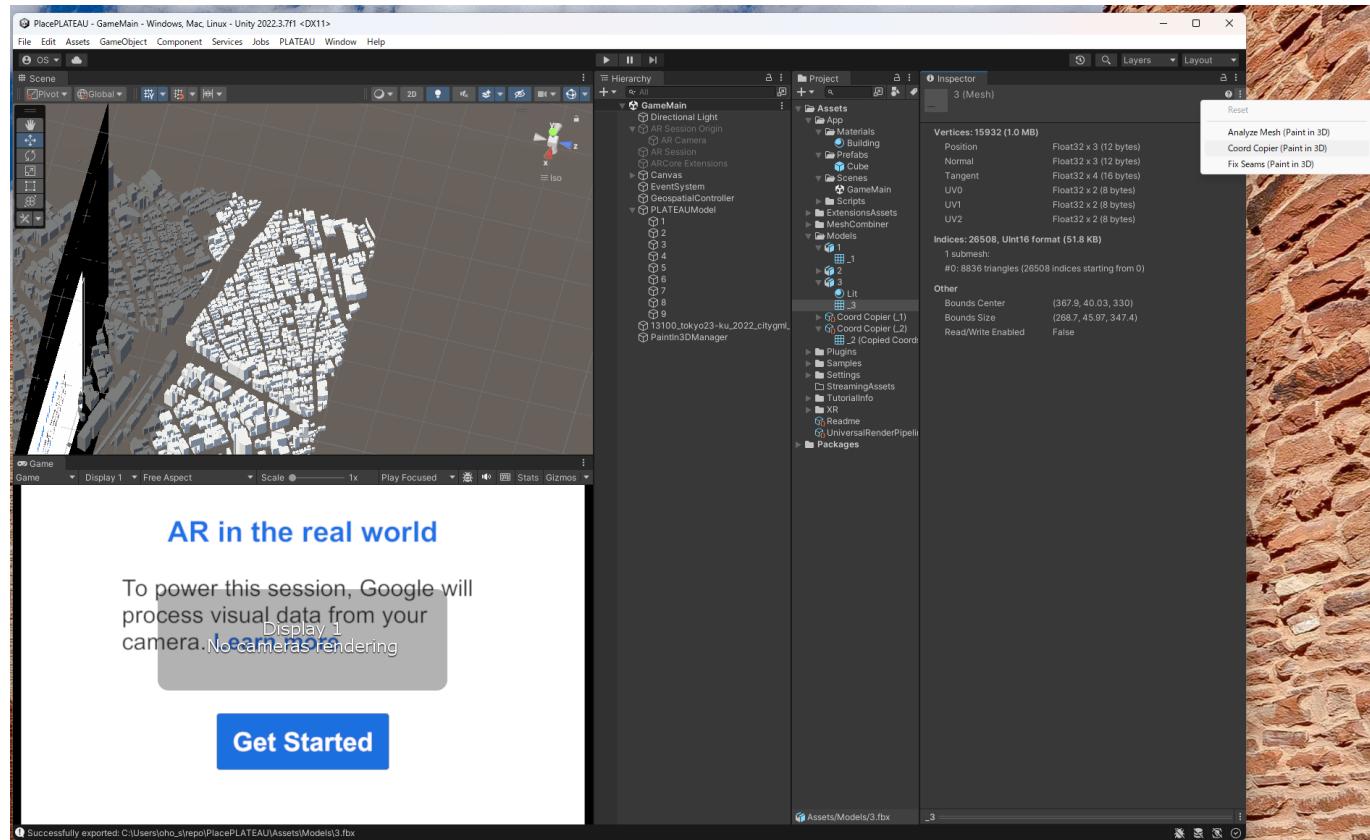
一度FBXに変換するために、「FBX Exporter」をUnity Package Managerからインストールしておきます。Hierarchyのモデルを選択して右クリックし「Export to FBX...」を選択します。



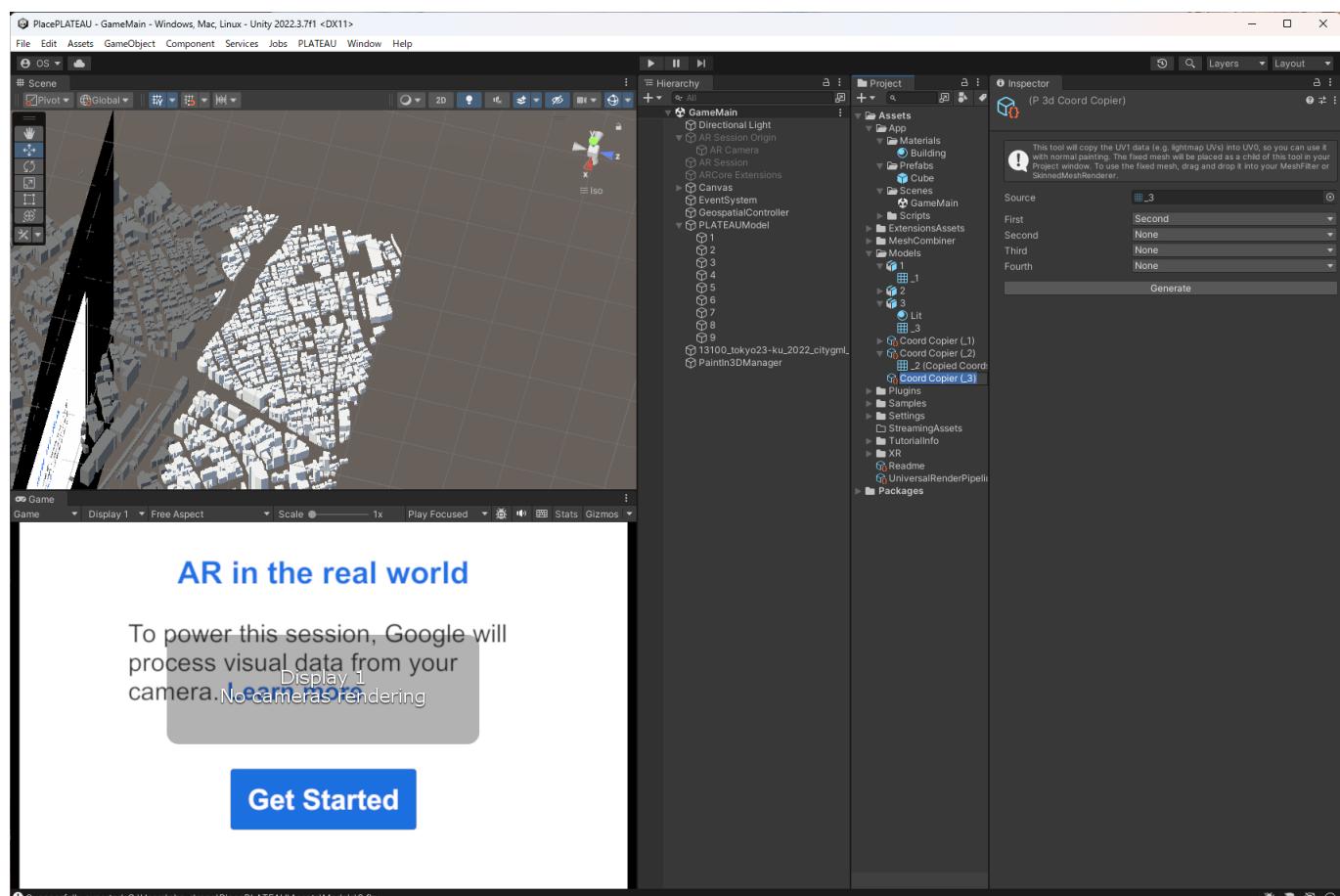
図のように設定して「Export」を押します。



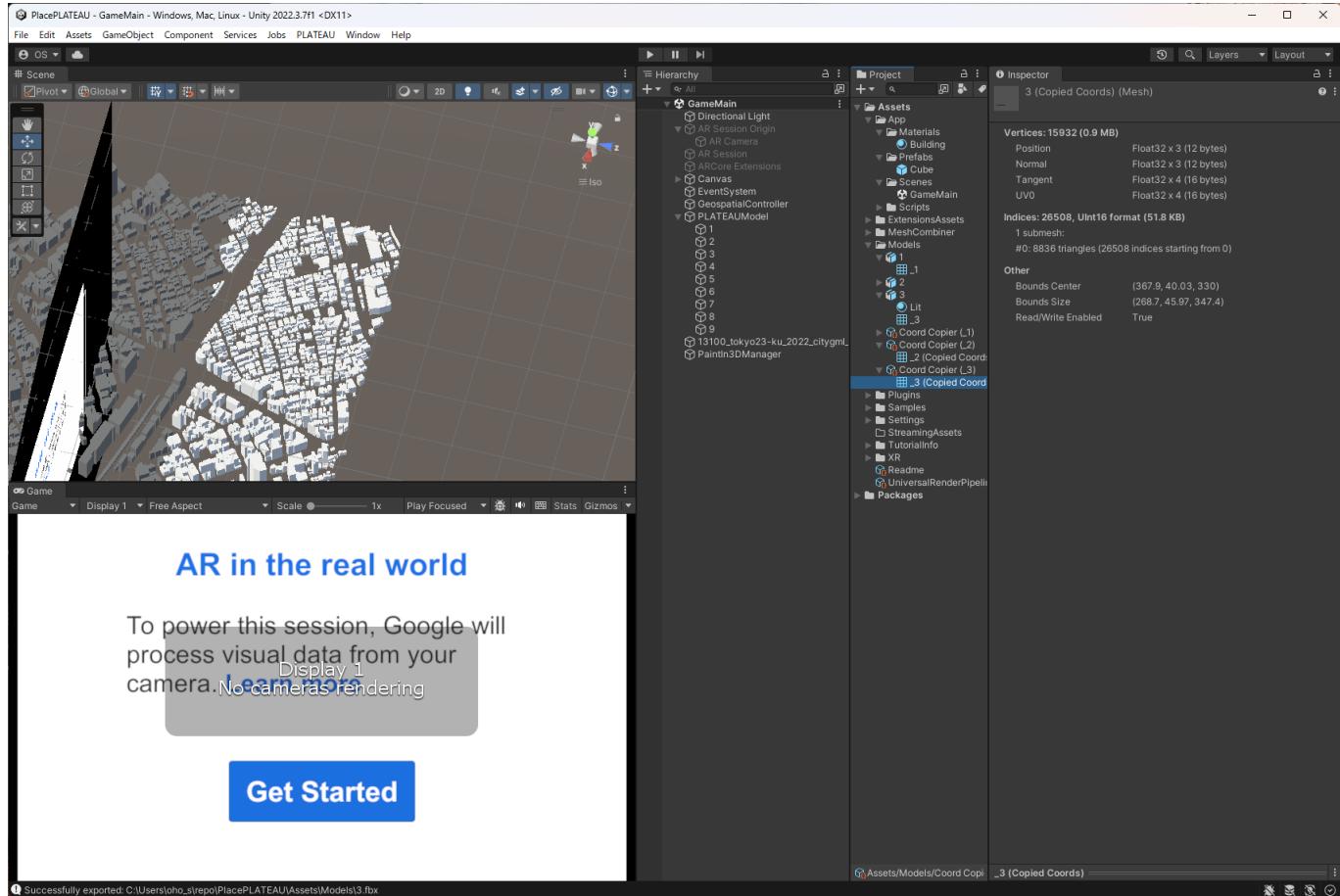
指定したフォルダにFBXとして書き出され、プロジェクトの方に表示されます。ここで、FBXの階層を開いてMeshのInspectorを表示し、右側のメニューを開くと、「Coord Copier(Paint in 3D)」があるのでこれを選択します。



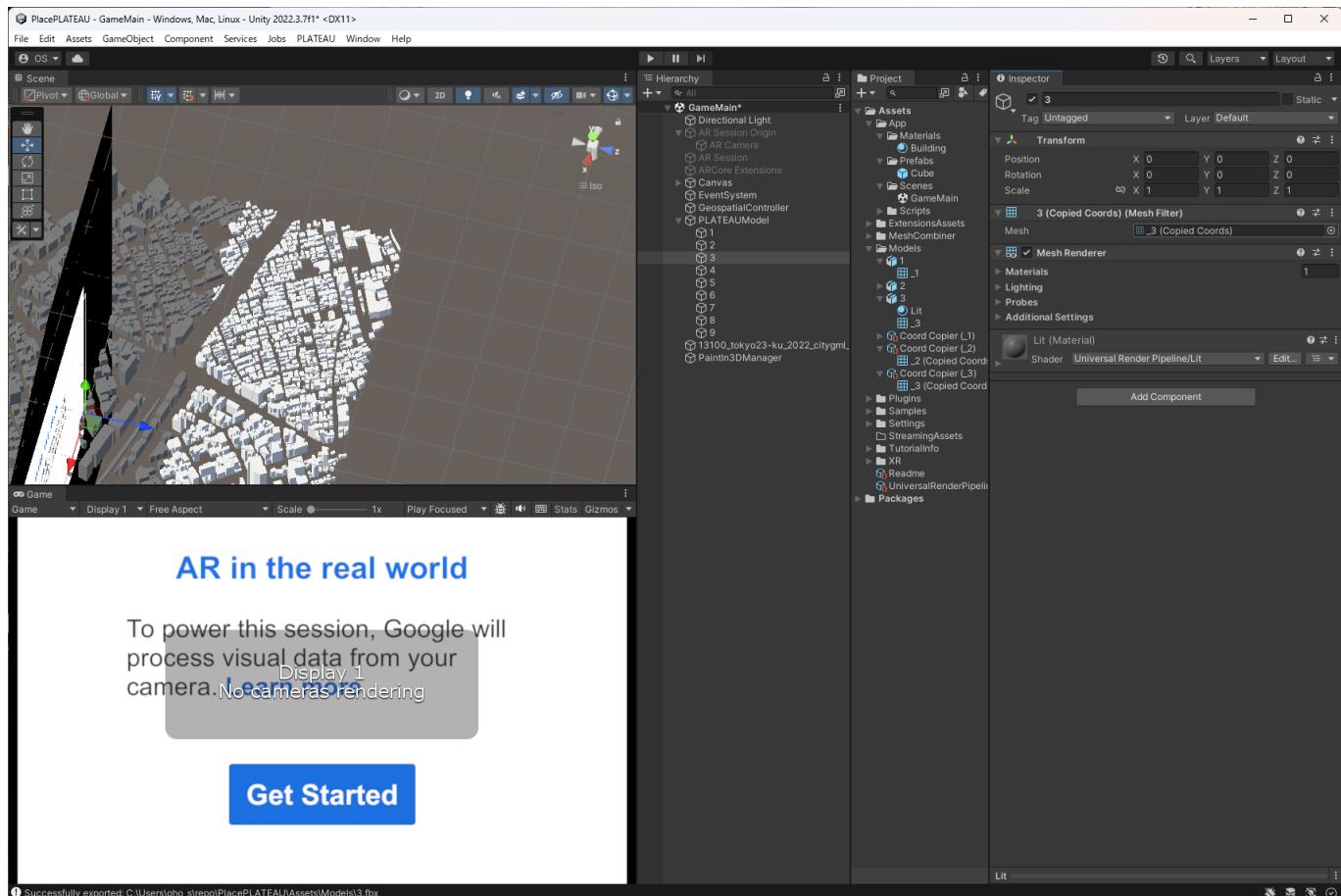
図のような設定で「Generate」ボタンを押します。



すると、UV1の内容をUV0にコピーしたメッシュが新しく作成されます。

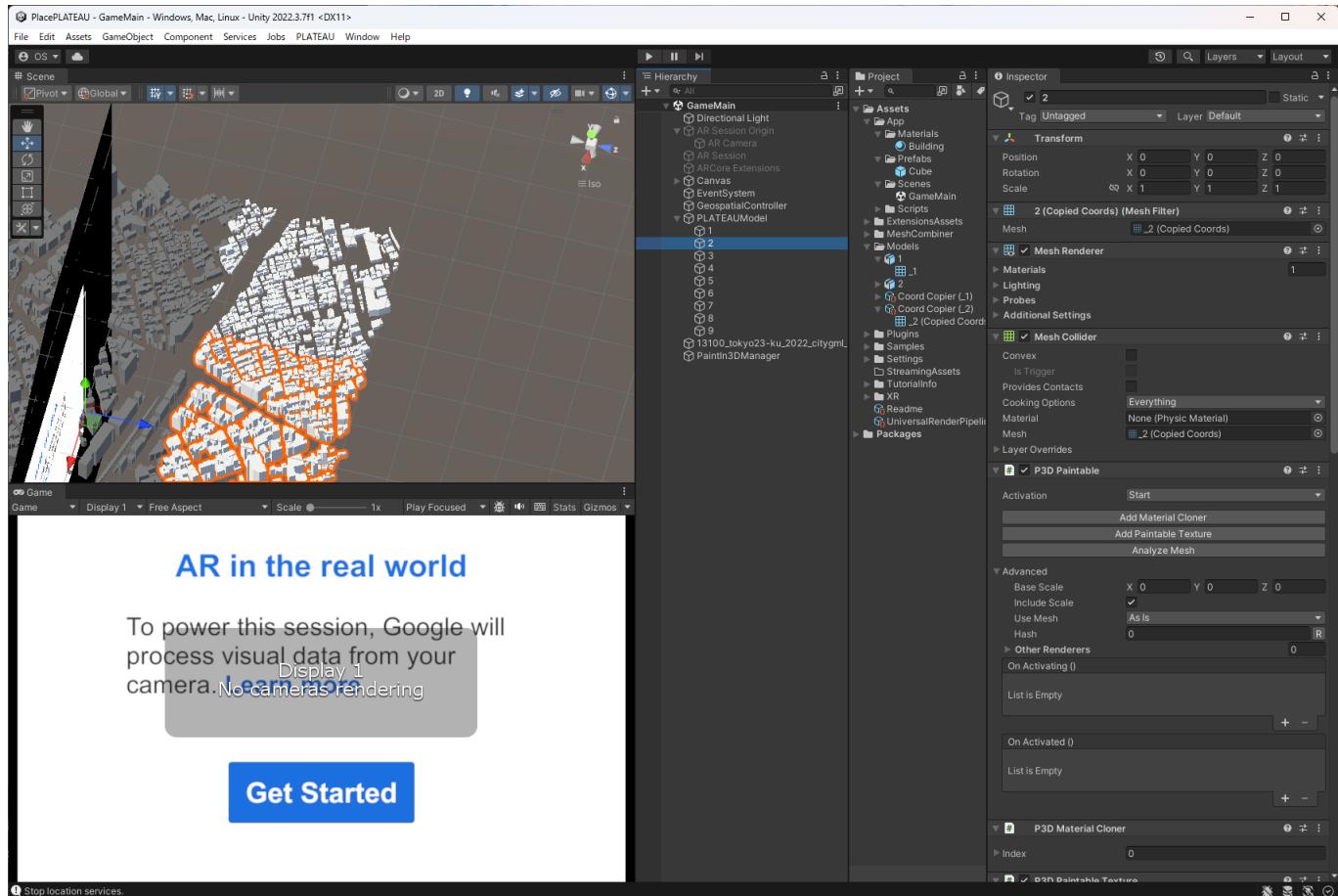


これを、元のメッシュの代わりに3D都市モデルのMesh FilterのMeshに設定します。

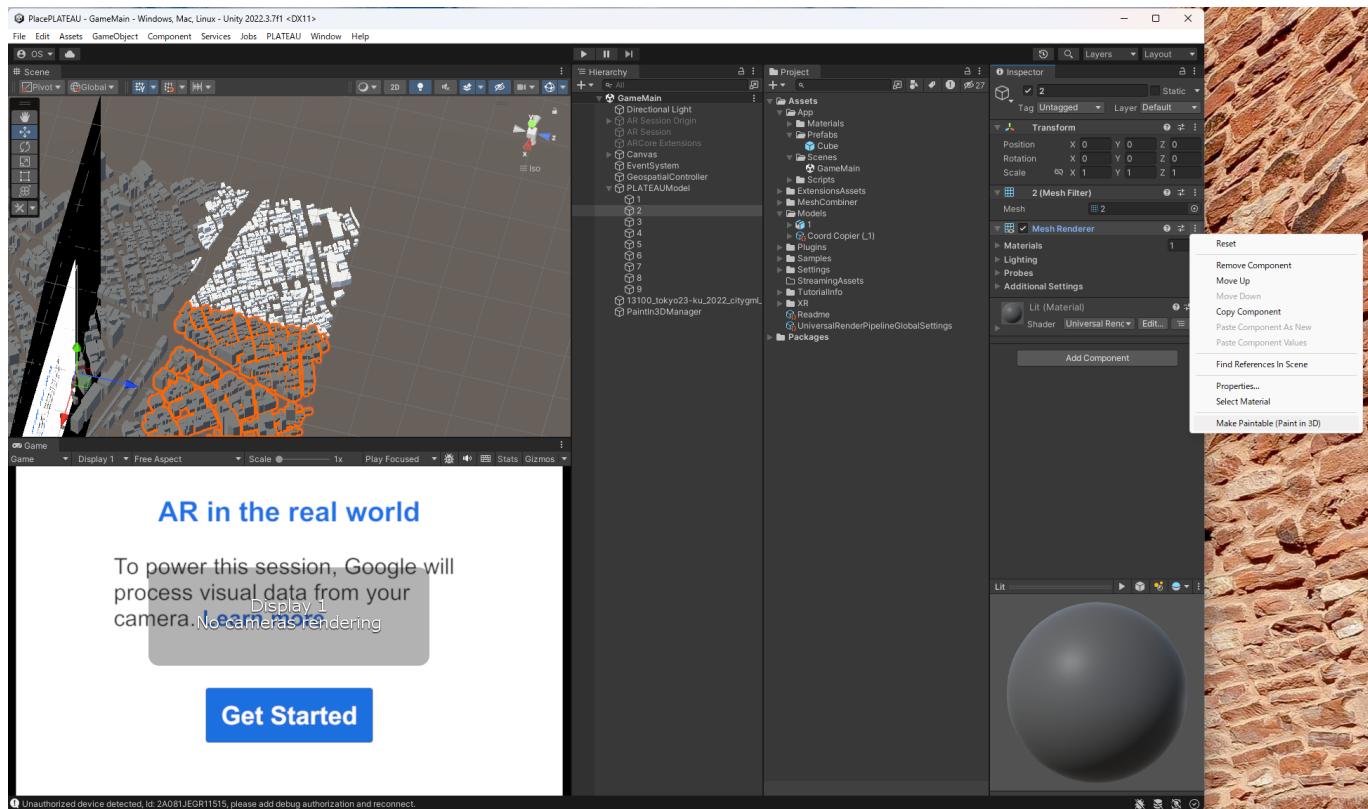


これで、Paint in 3D用に変換されたメッシュの準備ができました。

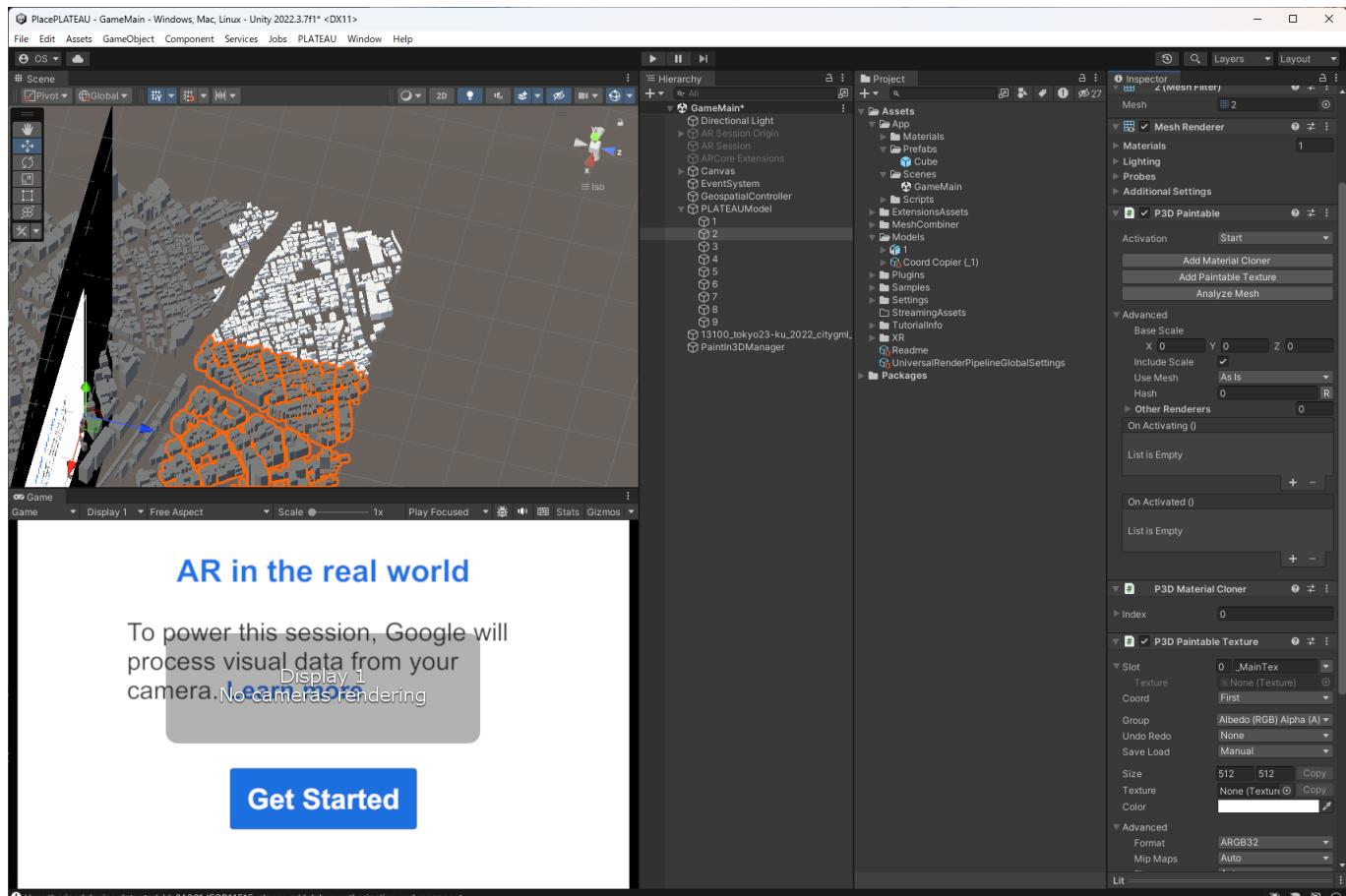
さらに、ペイントの衝突判定用にMesh Colliderを追加します。



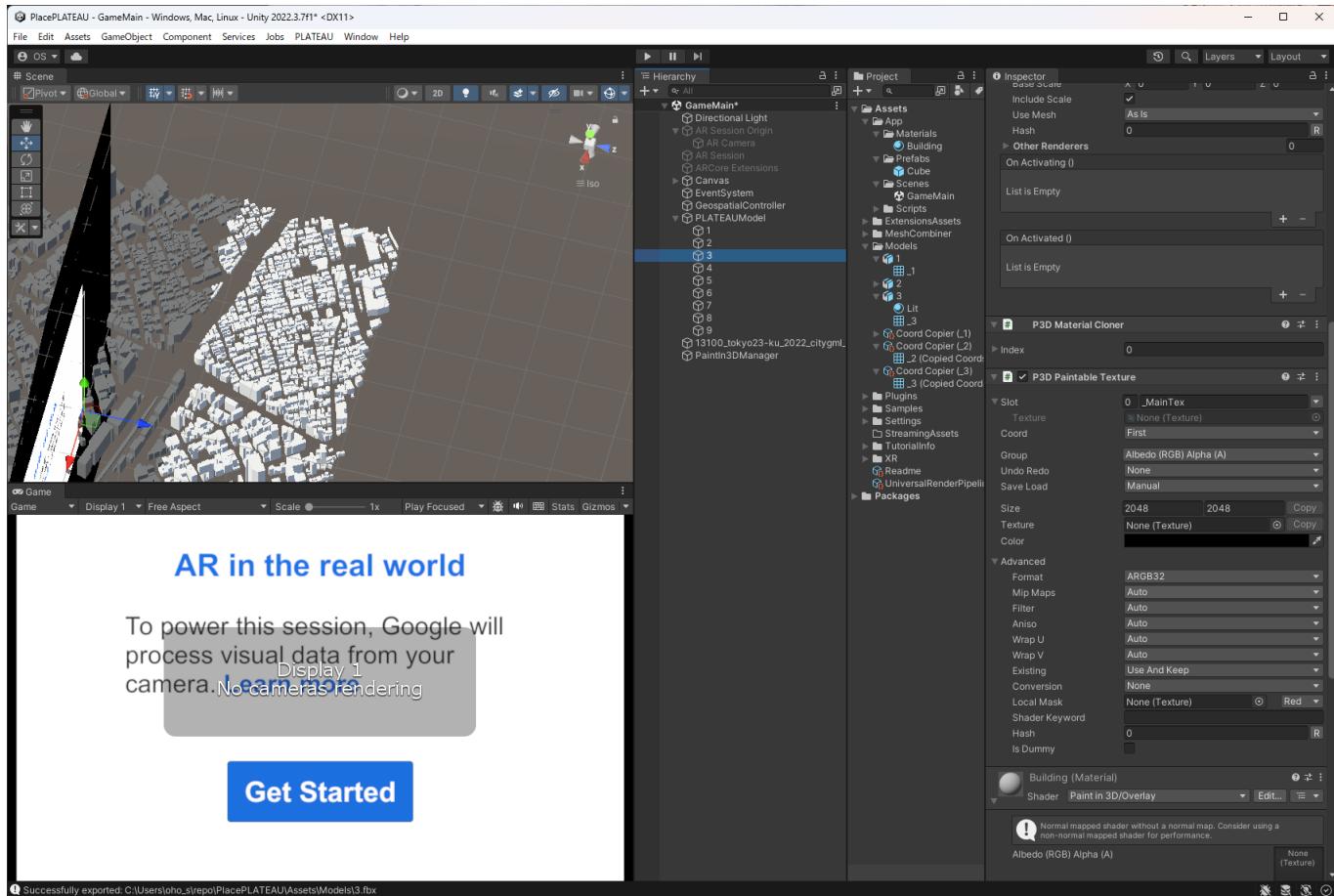
次に、Paint in 3Dで建物をペイントできるように設定します。結合したメッシュを選択し、InspectorのMesh Rendererの右肩のメニュー ボタンからメニューを開き、「Make Paintable(Paint in 3D)」を選択します。これで、Paint in 3Dのペイント対象になります。



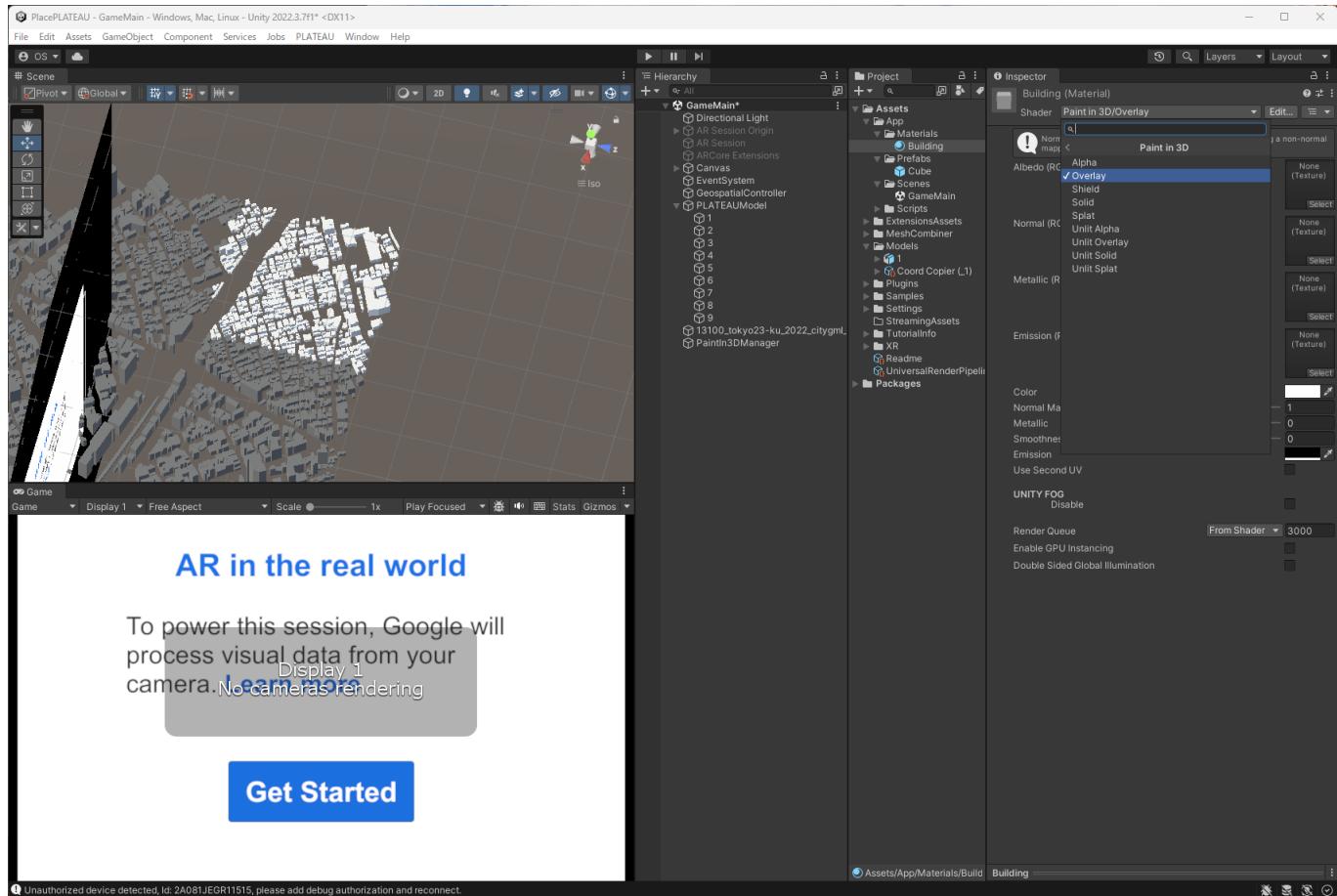
P3D Paintableコンポーネントのボタンで、「Add Material Cloner」と「Add Paintable Texture」を押下し、それぞれのコンポーネントも追加しておきます。



P3D Paintable Textureの設定を図のように変更します。Sizeを2048x2048に、ColorをR:0,G:0,B:0,A:0にしています。これでAR表示したときに、塗られていないところは完全な透過となります。

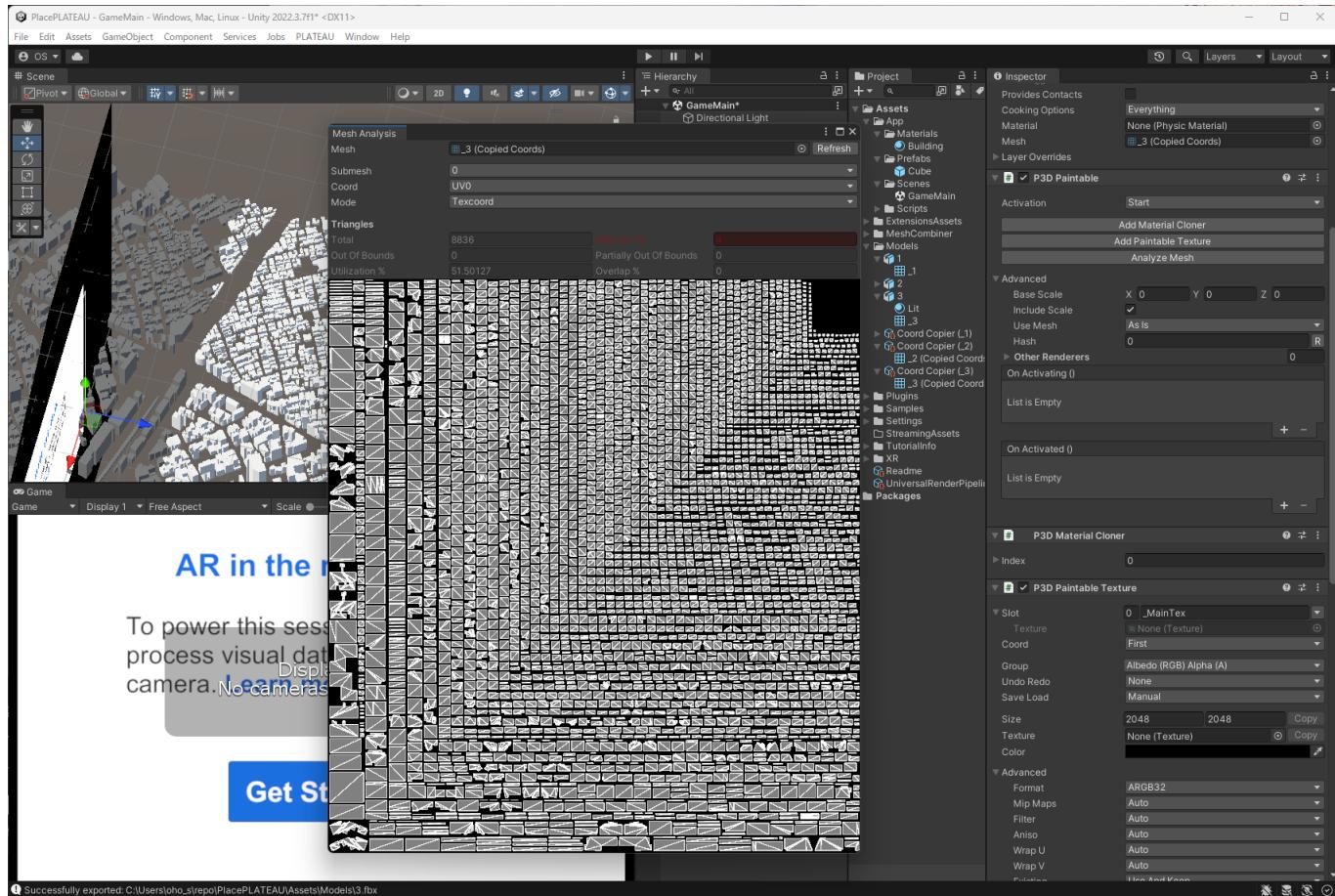


Paint in 3Dは、モデルのテクスチャにペイントした結果を描画しますが、表示のための専用のペイント用のマテリアルを作ります。（Paint in 3Dが自動的に実行時にコピーしてくれるので、このマテリアルは一つ作成してすべての3D都市モデルで共有することができます。）適当なフォルダーにマテリアルを新規に作り、シェーダーを「Paint in 3D/Overlay」にします。



このマテリアルを建物モデルに適用します。

なお、P3D Paintableの「Analyze Mesh」ボタンを押すと、現在のメッシュのUV座標を確認できます。図のように、CoordにUV0を指定した状態で、矩形が多く並んだような状態になっていれば、ここまで手順で正しく変換されていると考えられます。



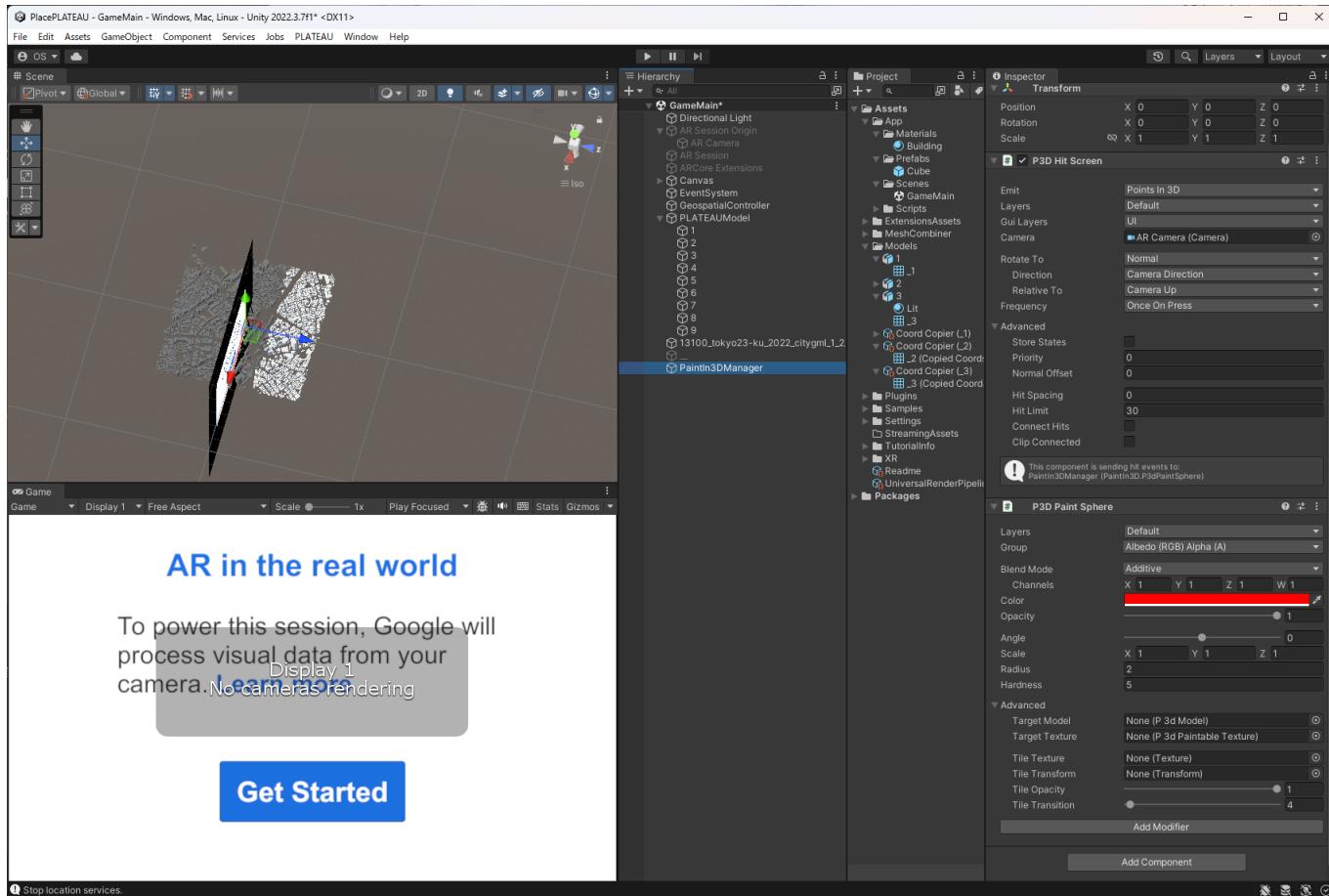
以上のモデルの設定を分割した全てのモデルに対して行います。

このように、ゲームエンジンでPLATEAUの3D都市モデルを表示する以上の活用をしようとすると、ゲームエンジンの様々な機能の活用やその仕組みの理解が必要になります。

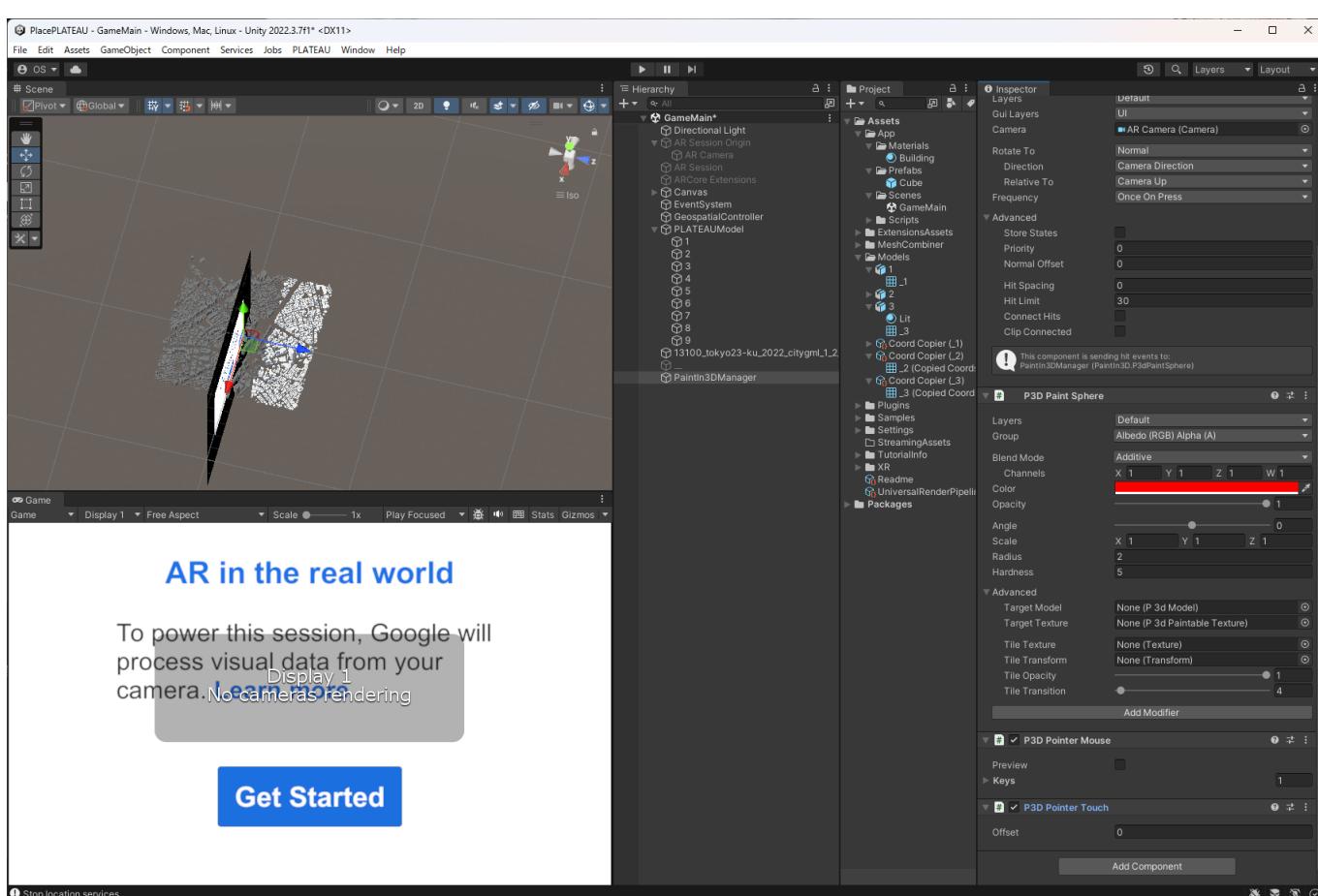
また適切に3D都市モデルのデータを変換や調整していくことも必要になります。

幸いにも、UnityやUnreal Engineでは、公式非公式問わず様々な入門コンテンツやサンプルが充実しています。一步進んだ応用を目指して活用してみましょう。

最後に、Paint in 3Dの描画を制御するコンポーネントを追加します。Hierarchyで、空のゲームオブジェクトを一つ追加し、「P3D Hit Screen」コンポーネントと「P3D Paint Sphere」コンポーネントを追加します。それぞれのコンポーネントは図のようく設定してください。



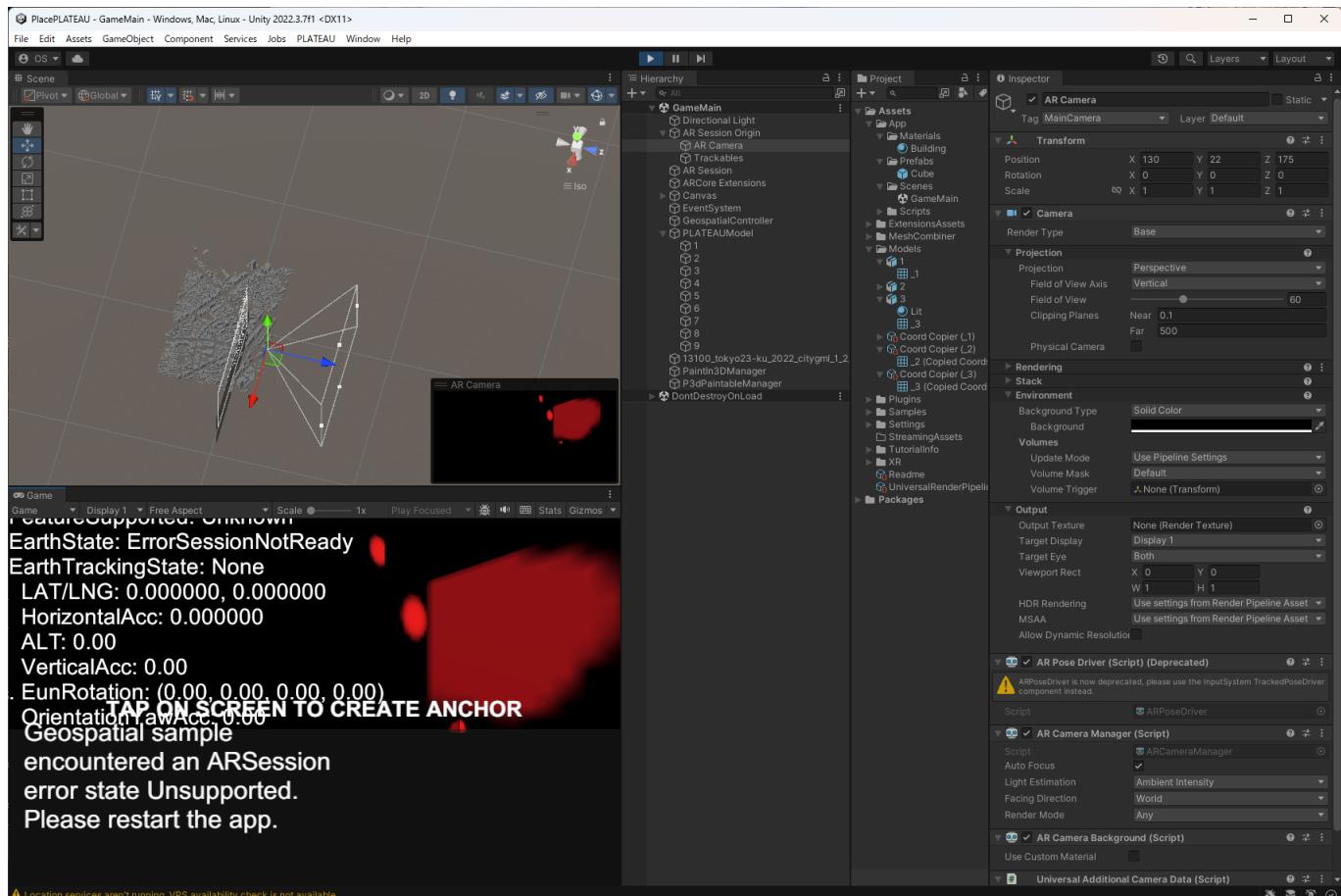
さらに、「P3D Pointer Mouse」と「P3D Pointer Touch」を追加します。これらはデフォルトの設定で構いません。



ここまでで、建物をペイントする仕組みができました。

PLATEAUの3D都市モデルは巨大なので、どうしてもテクスチャの解像度を上げるのが難しい場合があります。そこで、ドット絵風など表現を工夫することも考えられます。その際には、「P3D Paintable Texture」の「Advanced/Filter」の設定を「Point」に変えるとドット絵がくっきり見えるようになります。

Editor上で実行すると、マウスでクリックすると3D都市モデルにペイントできるようになっています。



## Geospatial Anchorで3Dモデルの位置を指定する

詳細は[公式ドキュメント](#)や、PLATEAUのLearningコンテンツのTopic 14-3内、「14.3.3 PLATEAUの3D都市モデルの読み込み」を参照してください。作成した3Dモデルの親要素にGeospatialAPIの地理空間アンカーを利用して位置合わせを行う以下のスクリプトを追加します。

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.XR.ARFoundation;
using UnityEngine.XR.ARSubsystems;
using Google.XR.ARCoreExtensions;

public class Anchor : MonoBehaviour
{
    public AREarthManager EarthManager;
```

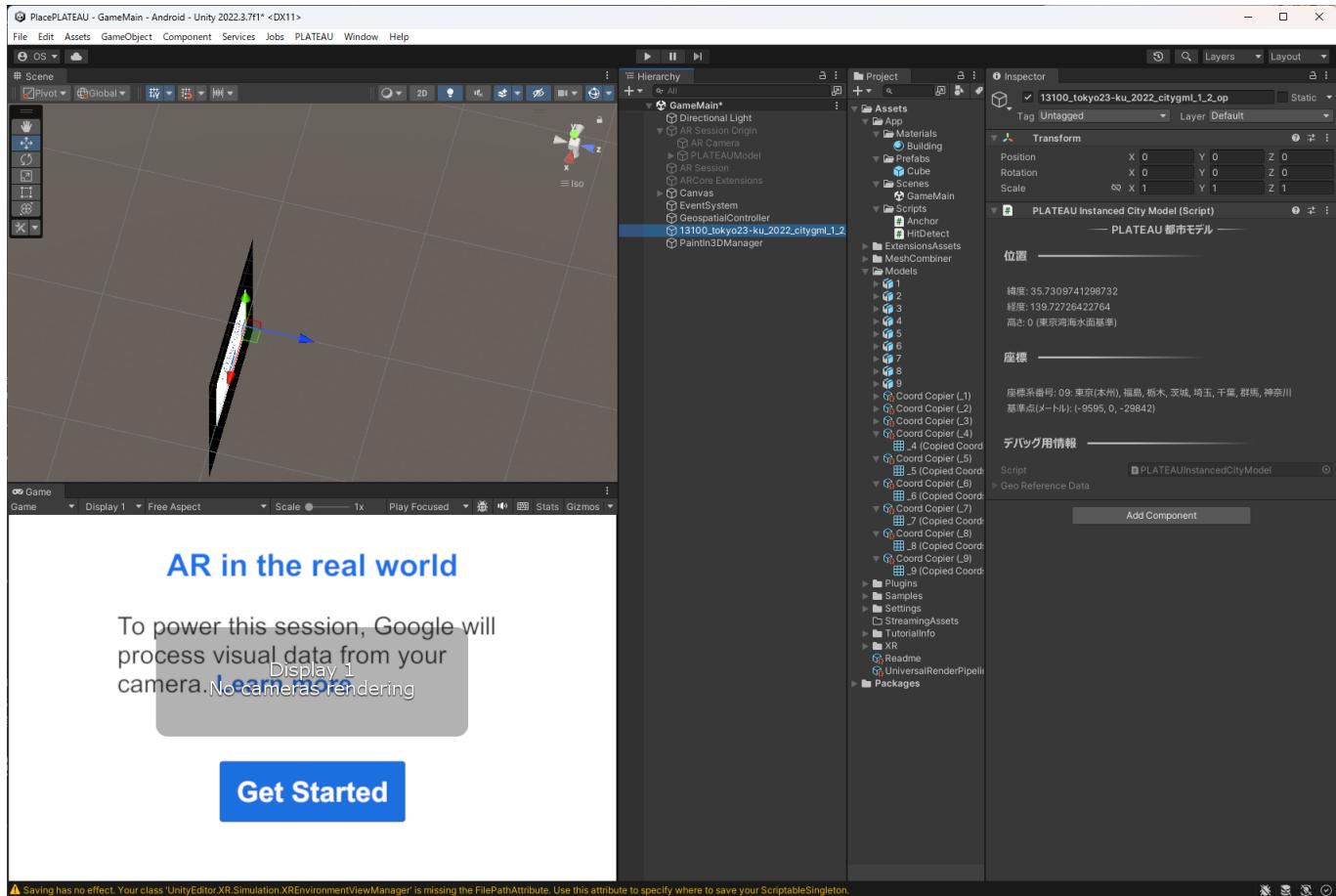
```
public ARAnchorManager AnchorManager;

public double lat, lon, height;

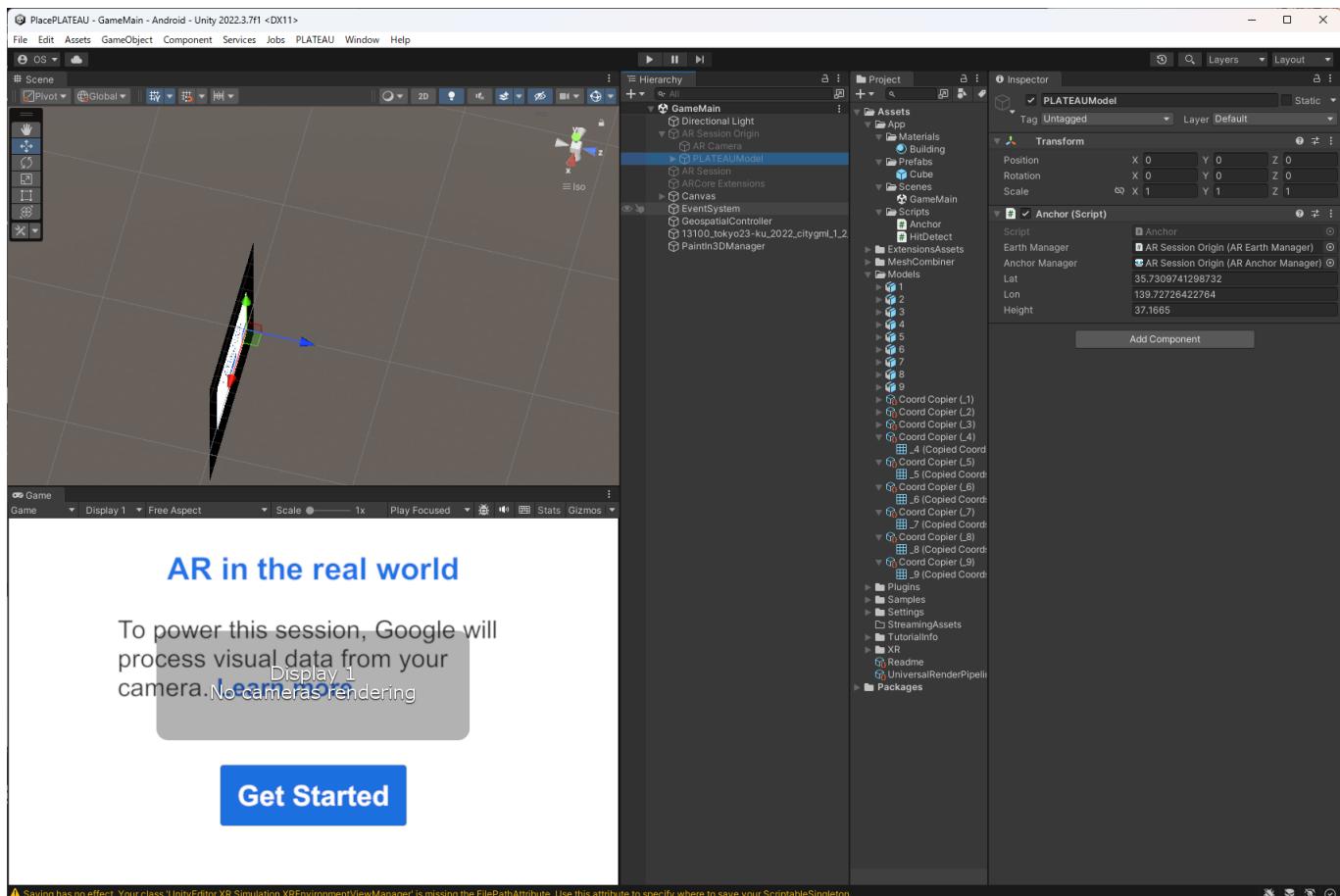
private bool _isInitialized = false;

// Update is called once per frame
void Update()
{
    // isInitializedフラグを見て、初回1回だけ実行
    if (!isInitialized && EarthManager.EarthTrackingState ==
    TrackingState.Tracking)
    {
        // 緯度35.731038475、経度139.72869019
        // 高さ37.1621mのアンカーを作る
        var anchor = AnchorManager.AddAnchor(
            lat,
            lon,
            height,
            Quaternion.identity
        );
        // このアンカーを親として設定する
        // そうすることで配下のオブジェクトは、設定した地理座標の位置に配置される。
        gameObject.transform.parent = anchor.transform;
        _isInitialized = true;
    }
}
```

読み込んだPLATEAUの3DモデルのSDKが表示する緯度経度を参照して、Anchor.csの緯度経度と高さを設定します。地理空間アンカーでは高さは樁円体高なので、[国土地理院のサイト](#)などを参考にしてジオイド高を加味した樁円体高を設定してください。



図のように、緯度経度高さを設定します。また、AR Session Originの参照をEarth ManagerとAnchor Managerに追加します。さらに、AR Session Originの子要素に移動しておきます。



## サーバーに位置情報を送る

位置情報を送る先のサーバーを作ります。機能としては、HTTPでJSONにした位置情報をクライアントとやり取りするシンプルなAPIサーバーです。サーバーでは位置情報を受信すると、それをPostGISの空間情報データとしてデータベースに格納し、空間クエリで面積を計算してクライアントに結果を返します。

サーバーにはさまざまな技術がありますが、ここではフレームワークにはPythonのFlaskを使い、PostgreSQL+PostGISをバックエンドデータベースとして使います。また、動作させるインフラとして、AWSを使うこととします。

これらの構成はあくまでもこのサンプルでの説明用で、同じようなことをやる場合のさまざまな選択肢があります。今回はなるべく汎用的かつ基本的で、読者が自分の環境に読み替えて考えやすい作り方を目指しました。言語もフレームワークもクラウドもデータベースも多くの選択肢があるので、自分の慣れているものに読み替えてください。

また、HTTPでやり取りするAPIは、リアルタイムでの同期には向いていません。FirebaseなどのMBaaSや、Photonなどのリアルタイム通信基盤など、まったく別の考え方の選択肢もあります。ここでは詳細を説明しませんが、作りたいサービスに合わせて選択してください。

### サーバーの準備

AWSでEC2インスタンスを立ち上げます。AWSを使うにはアカウントの登録が必要です。今回は無料枠の範囲内でできるように構成していますが、設定のミスなどで予想外の金額が請求されることもあります。アカウントの2段階認証を設定するなど、セキュリティにも十分気を付けてください。また、本項はある程度AWSを使い慣れている前提で進めます。分からぬことがある場合はAWSのドキュメントなどを参照してください。

また、内容の中にセキュリティの側面などで本番環境で動作させるには適さない部分があります。本項のアプリはあくまでも解説用のサンプルということで、実際のサービスなどを構築していく際には十分考慮してください。

EC2インスタンスはいわゆるクラウドの仮想マシンです。クラウドの使い方としては旧式な物ですが、なるべく基本的なところから説明して分かりやすくという方向で説明します。サーバーレスなど最近のモダンな構成を試したい方は是非チャレンジしてみてください。

OSは、著者が使い慣れていることもあります、Ubuntu 22.04を使います。インスタンスタイプは現時点ではt3a.nanoが単価が安いのでこちらを使っていきます。（※注：無料枠の範囲でやりたい人はt2.microを選択してください）キーペアなどは適切に設定してください。セキュリティグループは、SSHとHTTPSが通るように設定してください。テスト用にHTTPが通ってもよいと思いますが、スマホアプリからのアクセスではHTTPSが必須である場合が大半なので基本はHTTPSを使います。（※注：本番環境ではHTTPは外にさらさない方がいいと思います。）

EC2 > インスタンス > インスタンスを起動

## インスタンスを起動 情報

Amazon EC2 では、AWS クラウドで実行される仮想マシン（インスタンス）を作成できます。以下の簡単なステップに従ってすばやく開始できます。

**名前とタグ** 情報

名前  [さらにタグを追加](#)

**▼ アプリケーションおよび OS イメージ (Amazon マシンイメージ)** 情報

AMI は、インスタンスの起動に必要なソフトウェア設定（オペレーティングシステム、アプリケーションサーバー、アプリケーション）を含むテンプレートです。お探しのものが以下に表示されない場合は、AMI を検索または参照してください。

最新 | 自分の AMI | **クリックスタート**

Amazon Linux | macOS | Ubuntu | Windows | Red Hat | SUSE Linux | [その他 AMI を閲覧する](#)

**Amazon マシンイメージ (AMI)**

Ubuntu Server 22.04 LTS (HVM), SSD Volume Type  
ami-0d52744d6551d851e (64 ビット (x86)) / ami-0342c9aa06bb2a6488 (64 ビット (Arm))  
仮想化: hvm ENA 有効: true ルートデバイスタイプ: ebs

説明  
Canonical, Ubuntu, 22.04 LTS, amd64 jammy image build on 2023-05-16

**概要**

インスタンス数 情報  
1

ソフトウェアイメージ (AMI)  
Canonical, Ubuntu, 22.04 LTS, ...[続きを読む](#)  
ami-0d52744d6551d851e

垂直サーバータイプ (インスタンスタイプ)  
t3a.nano

ファイアウォール (セキュリティグループ)  
SSH&Web

ストレージ (ボリューム)  
1ボリューム - 8 GiB

**① 無料利用枠:** 最初の 1 年には、1 か月あたりの無料利用枠による AMI での t2.micro (または t2.micro が利用できないリージョンでは t3.micro) インスタンスの 750 時間の使用、30 GiB の EBS ストレージ、200 万の IOs、1 GiB のスナップショット、インターネットへの 100 GiB の帯域幅が含まれます。

[キャンセル](#) **インスタンスを起動** [コマンドを確認](#)

SSHで接続し環境構築します。最初に、PostgreSQLとその地理空間拡張のPostGISをインストールします。ついでにシステムのアップデートも一緒にしておきましょう。

```
$ sudo apt update
$ sudo apt upgrade
$ sudo apt install postgresql postgis
```

まずログインできるようにパスワードを設定します。

```
$ sudo -u postgres psql -c "ALTER USER postgres WITH PASSWORD 'password';"
```

このままだと、PostgreSQLのPeer認証になっているので、postgresユーザーしかアクセスできません。この後の作業に不便なので、Peer認証を変更します。以下のコマンドでPostgreSQLの設定ファイルを開きます。（エディターはvimでもEmacsでもお好みのものを使ってください。例ではnanoを使います。）

```
$ sudo nano /etc/postgresql/14/main/pg_hba.conf
```

この行を

```
# Database administrative login by Unix domain socket
local    all            postgres                      peer
```

このように変えて保存します。

```
# Database administrative login by Unix domain socket
local    all            postgres                      scram-sha-256
```

PostgreSQLのサービスを再起動します。

```
$ sudo service postgresql restart
```

次のように`psql`コマンドを`ubuntu`ユーザーで実行し、さきほど設定したパスワードでPostgreSQLのプロンプトに入れたら準備完了です。 (※注: 本来はデータベース操作用の権限を絞ったユーザーを作成しますが、簡単のためこのような設定で進めます。)

```
$ psql -U postgres
Password for user postgres:
psql (14.9 (Ubuntu 14.9-0ubuntu0.22.04.1))
Type "help" for help.

postgres=#
```

## データベースの準備

次にデータベースを作成します。データベースの名前は、`placeplateau`にします。PostgreSQLの詳しい使い方は公式ドキュメントなどを参考にしてください。

```
postgres=# create database placeplateau;
postgres=# \c placeplateau
```

PostGISを有効にします。

```
placeplateau=# CREATE EXTENSION postgis; CREATE EXTENSION postgis_topology;
```

アプリ用のテーブルを作ります。

```
placeplateau=# create table placedata (
  id SERIAL PRIMARY KEY,
```

```
userid VARCHAR(255) NOT NULL,  
side INTEGER NOT NULL,  
created_at TIMESTAMP NOT NULL,  
geom GEOMETRY(POLYGON,4326) NOT NULL);
```

## Webアプリケーションプログラムの作成

Python3とその周辺環境をインストールします。uwsgi経由でnginxをWebサーバーに使います。

```
$ sudo apt install python3 python3-pip python3-venv nginx
```

Pythonのvenvを設定しFlaskと必要なライブラリをインストールします。作業用のディレクトリは、自分のホーム直下のplaceplateau\_webとします。

```
$ sudo apt install libpq-dev  
$ mkdir placeplateau_web  
$ cd placeplateau_web/  
$ pwd  
/home/ubuntu/placeplateau_web  
$ python3 -m venv venv  
$ source venv/bin/activate  
(venv) $ pip install Flask uwsgi geoalchemy2 flask_sqlalchemy psycopg2  
geoalchemy2[shapely] pyjwt cryptography
```

次のPythonプログラムをエディタなどで作成します。

```
from flask import Flask, request  
from flask_sqlalchemy import SQLAlchemy  
from geoalchemy2 import Geometry  
from geoalchemy2.shape import to_shape  
from datetime import datetime, date, timedelta  
from sqlalchemy.sql import func, and_  
import json  
import jwt  
import time  
  
app = Flask(__name__)  
app.config['SQLALCHEMY_DATABASE_URI'] =  
'postgresql://postgres:password@localhost/placeplateau'  
db = SQLAlchemy(app)  
  
# テーブルのスキーマ定義  
class PlaceData(db.Model):  
    __tablename__ = 'placedata'  
    id = db.Column(db.Integer, primary_key=True, autoincrement=True)  
    userid = db.Column(db.String)  
    side = db.Column(db.Integer)
```

```
created_at = db.Column(db.Time)
geom = db.Column(Geometry('POLYGON'))

def getdict(self):
    return {"id":self.id,
            "userid":self.userid,
            "side":self.side,
            "created_at":str(self.created_at),
            "geom":to_shape(self.geom).wkt}

# 動作確認 テスト用
@app.route('/')
def hello():
    return 'PlacePLATEAU API SERVER'

# 当日のデータ取得
@app.route('/getarea')
def getArea():
    today = date.today()
    placedatas = db.session.query(PlaceData).where(PlaceData.created_at >=
func.date(func.now())).order_by(PlaceData.id)
    datas = []
    for placedata in placedatas:
        print(placedata.id)
        datas.append(placedata.getdict())
    return json.dumps(datas)

# 新しい領域の作成と未読データの取得
@app.route('/makearea', methods=['POST'])
def makeArea():
    data = request.json

    lastid=data['lastid']
    userid=data['userid']
    side=data['side']
    newareaWKT=data['newarea']

    newarea = PlaceData(userid=userid, created_at=func.now(), side=side,
geom=newareaWKT)
    db.session.add(newarea)
    db.session.commit()

    placedatas = db.session.query(PlaceData).filter(and_(PlaceData.created_at >=
func.date(func.now()),PlaceData.id >= lastid)).order_by(PlaceData.id)
    datas = []
    for placedata in placedatas:
        print(placedata.id)
        datas.append(placedata.getdict())
    return json.dumps(datas)

if __name__ == '__main__':
    app.run(host='0.0.0.0')
```

PostgreSQLにデータを記録しているところを説明します。WebフレームワークはFlaskを使用しています。FlaskはPythonの軽量なWebフレームワークで、例のように各URLに対する処理をメソッドとして記述することでWebサーバーとして動作します。

ORMとして、[SQLAlchemy](#)に[GeoAlchemy](#)という地理情報拡張を組み合わせて使っています。

スキーマ定義で、Geometry型を使うことで、PostGISの地理情報型を利用できます。

```
# テーブルのスキーマ定義
class PlaceData(db.Model):
    __tablename__ = 'placedata'
    id = db.Column(db.Integer, primary_key=True, autoincrement=True)
    userid = db.Column(db.String)
    side = db.Column(db.Integer)
    created_at = db.Column(db.Time)
    geom = db.Column(Geometry('POLYGON'))
```

新規の領域を作る(SQLのInsertを発行する)際は、WKT(Well Known Text)形式でジオメトリーのデータを渡します。

```
newarea = PlaceData(userid=userid, created_at=func.now(), side=side,
geom=newareaWKT)
db.session.add(newarea)
db.session.commit()
```

ここでは使っていませんが、空間検索なども実行できます。GeoAlchemyの[公式ドキュメント](#)などを参照してください。

最後に動作確認をします。app.pyを保存したディレクトリで、以下のコマンドを実行します。

```
$ python app.py
```

開発サーバーですよという警告が出ますが、IPアドレスとポートが表示され動作します。curlで動作確認をします。以下コマンドで、動作確認用のメッセージが出力されます。

```
$ curl http://127.0.0.1:5000/
```

また、以下のコマンドで、データベースに新しいエリアが作成され、既存のエリアのリストがJSONで返ってきます。

```
$ curl -X POST -H "Content-Type: application/json" -d
'{"lastid":"0","userid":"12345","side":"0","newarea":"POLYGON((1 0,3 0,3 2,1 2,1
0))"}' http://127.0.0.1:5000/makearea
```

## Webサーバーの公開

ここまでAWS上の仮想マシンで外からのアクセスができない状態でのテストでした。次に、nginxを設定して外部に公開する設定をします。

uwsgi用の設定ファイルを作成します。app.pyと同じ場所にapp.iniというファイル名で以下のファイルを作成します。

```
[uwsgi]
module = app
callable = app
master = true
processes = 1
socket = /tmp/uwsgi.sock
chmod-socket = 666
vacuum = true
die-on-term = true
wsgi-file = /home/ubuntu/placeplateau_web/app.py
logto = /home/ubuntu/placeplateau_web/app.log
```

nginxの設定ファイルを修正します。以下のコマンドでroot権限で設定ファイルを開き編集します。

```
$ sudo nano /etc/nginx/nginx.conf
```

httpのセクションに以下の二行があるので、sites-enabledの方をコメントアウトします。

```
include /etc/nginx/conf.d/*.conf;
#include /etc/nginx/sites-enabled/*;
```

nginxのconf.d以下にuwsgi.confを作成します。

```
$ sudo nano /etc/nginx/conf.d/uwsgi.conf
```

uwsgi.confに以下の内容を記載します。

```
server {
    listen 443 ssl;
    ssl_certificate      ※SSL証明書へのパス;
    ssl_certificate_key  ※SSL秘密鍵へのパス;
    server_name  ※サーバーのドメイン名;
    location / {
        include uwsgi_params;
```

```
    uwsgi_pass unix:///tmp/uwsgi.sock;
}
}
```

ドメイン名の取得は有料になることが多いります。SSL証明書は、テスト用と割り切るならLet's Encryptなどを使って取得してもよいでしょう。

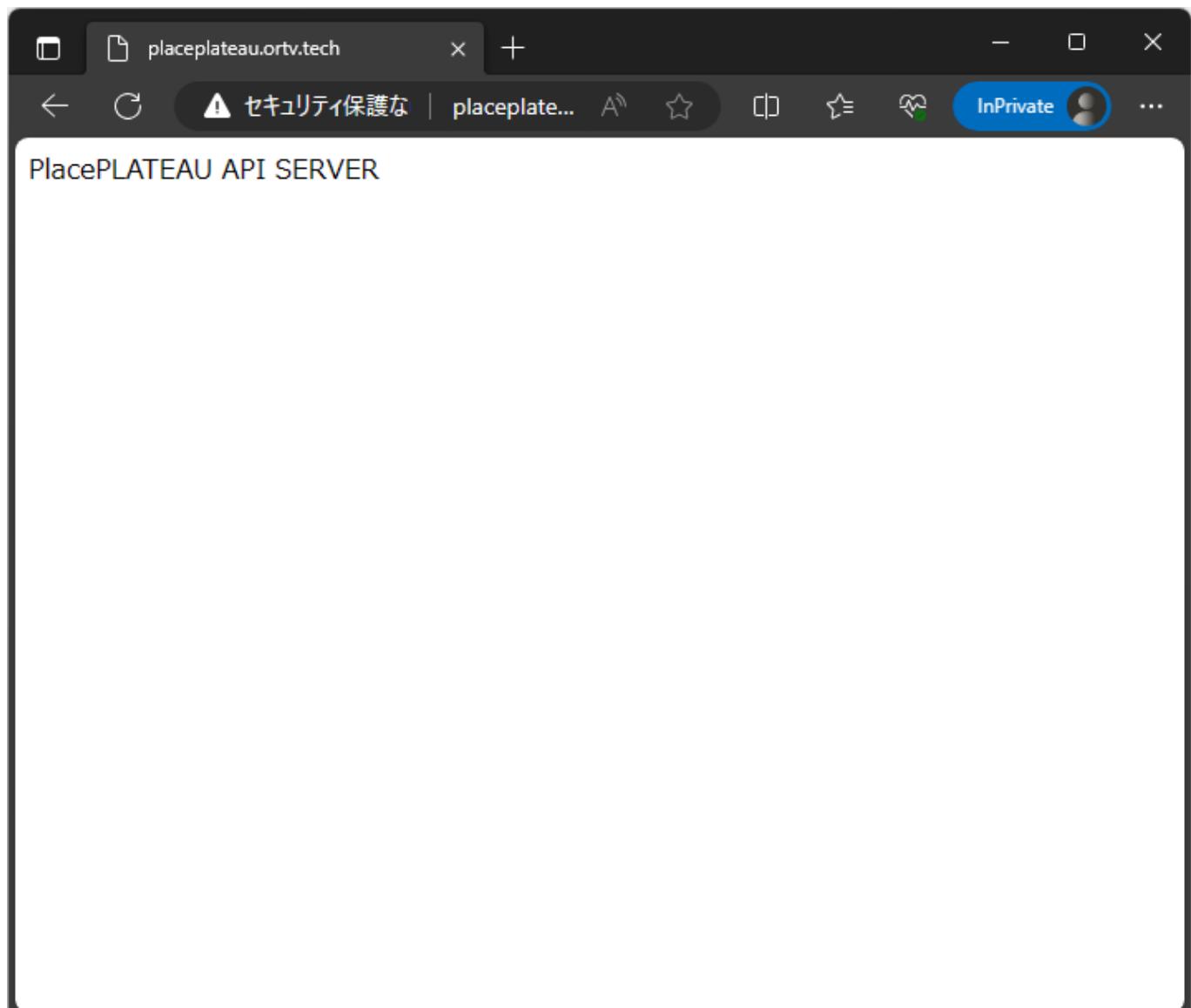
ここまでできたら、nginxを再起動します。

```
$ sudo service nginx restart
```

以下のコマンドでuwsgiを起動して、外部からアクセスして確認します。

```
$ uwsgi --ini app.ini
```

ブラウザにURLを入力することで、アクセスの確認ができます。



## ※ドメイン取得やSSLについて

Unityでは、デフォルトではAndroidやiOSでビルドした場合、httpでのアクセスを制限しています。これは、プラットフォームの制限になります。

ここまで手順では既にドメインや証明書がある前提で検証しています。

実際のサービスを運用するためには、SSLを使ったアクセスや、正式にドメインを取得しての様々な設定などが必要ですが、本項の内容を超えるので、ここでは説明しません。

また、サーバーでのサービスの自動起動にも触れません。

## サーバーへの位置情報の送信

ここまででサーバーのAPIができました。次にUnityのクライアントプログラムと連携させます。まずはUnityクライアント側のプログラムを作成します。

Paint in 3Dでは、`IHitPoint`を継承したクラスを作成し、`HandleHitPoint`を実装することで、塗った場所の座標を取得できます。これを使いサーバーへ位置情報を送信する機能を作成したのが次のプログラムです。これを`P3DHitScreen`と同じゲームオブジェクトに追加します。

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.Networking;
using PaintIn3D;
using Google.XR.ARCoreExtensions;
using System.Text;
using Newtonsoft.Json;

public class GeoPaintManager : MonoBehaviour, IHitPoint
{
    private List<GameObject> polygons = new List<GameObject>();
    private int lastid = 0;
    private int side = 0;
    private string userid = "test";

    private List<GeoPoint> tempPolygon = new List<GeoPoint>();
    private Vector3 firstPoint;
    private bool isFirst = true;

    private bool isInitialized = false;

    public AREarthManager arEarthManager;
    public Transform arOrigin;
    public Transform arcam;

    public GameObject polygonLineRender;
    public GameObject areaParent;

    private const float DISTTHR = 10;
    private const float POLYGON_HEIGHT = 40;

    private const string url = "https://placeplateau.ortv.tech/";
```

```
public void HandleHitPoint(bool preview, int priority, float pressure, int seed, Vector3 position, Quaternion rotation)
{
    if (!preview)
    {
        Debug.Log("HitDetect Pos " + position + "Rot " + rotation);

        if (arEarthManager.EarthTrackingState ==
UnityEngine.XR.ARSubsystems.TrackingState.Tracking)
        {
            Pose p = new Pose(position, rotation);
            GeospatialPose geoPose = arEarthManager.Convert(p);
            Debug.Log(geoPose);

            if (isFirst)
            {
                firstPoint = position;
                tempPolygon.Add(new GeoPoint(geoPose.Latitude,
geoPose.Longitude));
                isFirst = false;
            }
            else if (tempPolygon.Count > 2 && Vector3.Distance(firstPoint,
position) < DISTTHR)
            {
                // Loop Close
                StringBuilder sb = new StringBuilder();
                foreach (GeoPoint gp in tempPolygon)
                {
                    sb.Append($"{gp.lon} {gp.lat},");
                }
                GeoPoint fpgp = tempPolygon[0];
                sb.Append($"{fpgp.lon} {fpgp.lat}");
                string wkt = sb.ToString();

                string reqjson = $"{{{{\"lastid\":\"{lastid}\",\"userid\":\\"
{userid}\\",\"side\":\"{side}\",\"newarea\":\"POLYGON(({wkt}))\"}}}";
                StartCoroutine(SendNewArea(reqjson));

                tempPolygon.Clear();
                isFirst = true;
            }
            else
            {
                tempPolygon.Add(new GeoPoint(geoPose.Latitude,
geoPose.Longitude));
            }
        }
    }
}

IEnumerator SendNewArea(string json)
{
```

```
UnityWebRequest req = new UnityWebRequest(url + "makearea", "POST");
byte[] bodyRaw = System.Text.Encoding.UTF8.GetBytes(json);
req.uploadHandler = (UploadHandler)new UploadHandlerRaw(bodyRaw);
req.downloadHandler = (DownloadHandler)new DownloadHandlerBuffer();
req.SetRequestHeader("Content-Type", "application/json");

yield return req.SendWebRequest();

if (req.result == UnityWebRequest.Result.Success)
{
    Debug.Log(req.downloadHandler.text);
    ParseRecord(req.downloadHandler.text);
}
else
{
    Debug.LogError("Error sending POST request: " + req.error);
}
}

IEnumerator GetArea()
{
    UnityWebRequest req = new UnityWebRequest(url + "getarea", "GET");
    req.downloadHandler = (DownloadHandler)new DownloadHandlerBuffer();

    yield return req.SendWebRequest();

    if (req.result == UnityWebRequest.Result.Success)
    {
        Debug.Log(req.downloadHandler.text);
        ParseRecord(req.downloadHandler.text);
    }
    else
    {
        Debug.LogError("Error sending POST request: " + req.error);
    }
}

void ParseRecord(string json)
{
    List<Record> records = JsonConvert.DeserializeObject<List<Record>>(json);
    foreach (Record record in records)
    {
        var points = new List<Vector3>();
        string geom = record.geom.Replace("POLYGON ((", "").Replace("))", "");

        foreach (string coord in geom.Split(","))
        {
            var tokens = coord.Trim().Split(" ");
            double x = double.Parse(tokens[0].Trim());
            double y = double.Parse(tokens[1].Trim());
            GeospatialPose geoPose = new GeospatialPose();
            geoPose.Latitude = y;
            geoPose.Longitude = x;
            geoPose.Altitude = POLYGON_HEIGHT;
        }
    }
}
```

```
        geoPose.Heading = 0;
        geoPose.EulerRotation = Quaternion.identity;
        Pose pose = arEarthManager.Convert(geoPose);
        points.Add(pose.position);
    }
    GameObject polygon = Instantiate(polygonLineRender,
areaParent.transform);
    var linerenderer = polygon.GetComponent<LineRenderer>();
    linerenderer.SetPositions(points.ToArray());
    linerenderer.positionCount = points.Count;

    if(record.id > lastid)
    {
        lastid = record.id;
    }
}

// Update is called once per frame
void Update()
{
    if(!isInitialized && arEarthManager.EarthTrackingState ==
UnityEngine.XR.ARSubsystems.TrackingState.Tracking)
    {
        isInitialized = true;
        StartCoroutine(GetArea());
    }
}
}

public struct GeoPoint
{
    public double lat, lon;

    public GeoPoint(double lat, double lon)
    {
        this.lat = lat;
        this.lon = lon;
    }
}

public class GeoPolygon
{
    public List<GeoPoint> polygon = new List<GeoPoint>();
    public int side;
}

[JsonObject(MemberSerialization.OptIn)]
public class Record
{
    [JsonProperty]
    public int id { get; set; }
    [JsonProperty]
    public string userid { get; set; }
}
```

```
[JsonProperty]
public int side { get; set; }
[JsonProperty]
public string created_at { get; set; }
[JsonProperty]
public string geom { get; set; }
}
```

AREarthManagerのConvertメソッドで、Unity座標と地理座標の相互の変換ができます。

APIとのやり取りは、UnityWebRequestを使いJSON形式で送受信しています。JSON.Netを使っていますので、別途Unity Package Managerでインストールしておきます。

APIへのデータ送信のタイミングは、始点から閾値距離内の点を塗ったときに閉じたと判定して行っています。

## サーバーでの面積の計算

ここまでで、Unityアプリとサーバーで情報を連携させることができました。アプリから作成した地理情報はPostGISに記録されるので、PostGISの機能を使った処理が可能です。ここでは、簡単なクエリを実行して面積の計算をしてみます。

サーバーにSSHでログインして、psqlコマンドを実行し、PostgreSQLのコンソールに入ります。 \cを使ってデータベースを選択します。

```
$ psql -U postgres
...
postgres=# \c placeplateau
```

次のクエリを実行します。各エリアの面積(m<sup>2</sup>)が計算されます。

```
SELECT *, ST_Area(Geography(ST_Transform(geom,4326))) FROM placedata;
```

SQLの機能をもっと使って集計してみましょう。次のクエリを実行し、陣営単位で面積を集計します。

```
placeplateau=# SELECT side, sum(ST_Area(Geography(ST_Transform(geom,4326)))) FROM placedata GROUP BY side;
 side |      sum
-----+-----
  0  | 1360.2750182356685
  1  | 1773.828494577203
(2 rows)
```

さらに、陣営・ユーザー単位で面積を集計します。

```
placeplateau=# SELECT side,userid,
sum(ST_Area(Geography(ST_Transform(geom,4326)))) FROM placedata GROUP BY
side,userid;
 side | userid |      sum
-----+-----+
 0  |  test  | 1320.7396999783814
 0  | test3 | 39.535318257287145
 1  | test2 | 1773.8014401495457
 1  | test4 | 0.027054427657276392
(4 rows)
```

このように、アプリで作成したデータをSQLで自由自在に分析することができます。データベースを適切に設計すれば、さらに様々な活用ができます。

## QGISで塗った場所の表示

QGISはPostGISに接続する機能があります。サーバーのPostGISに接続して地図上に表示してみます。

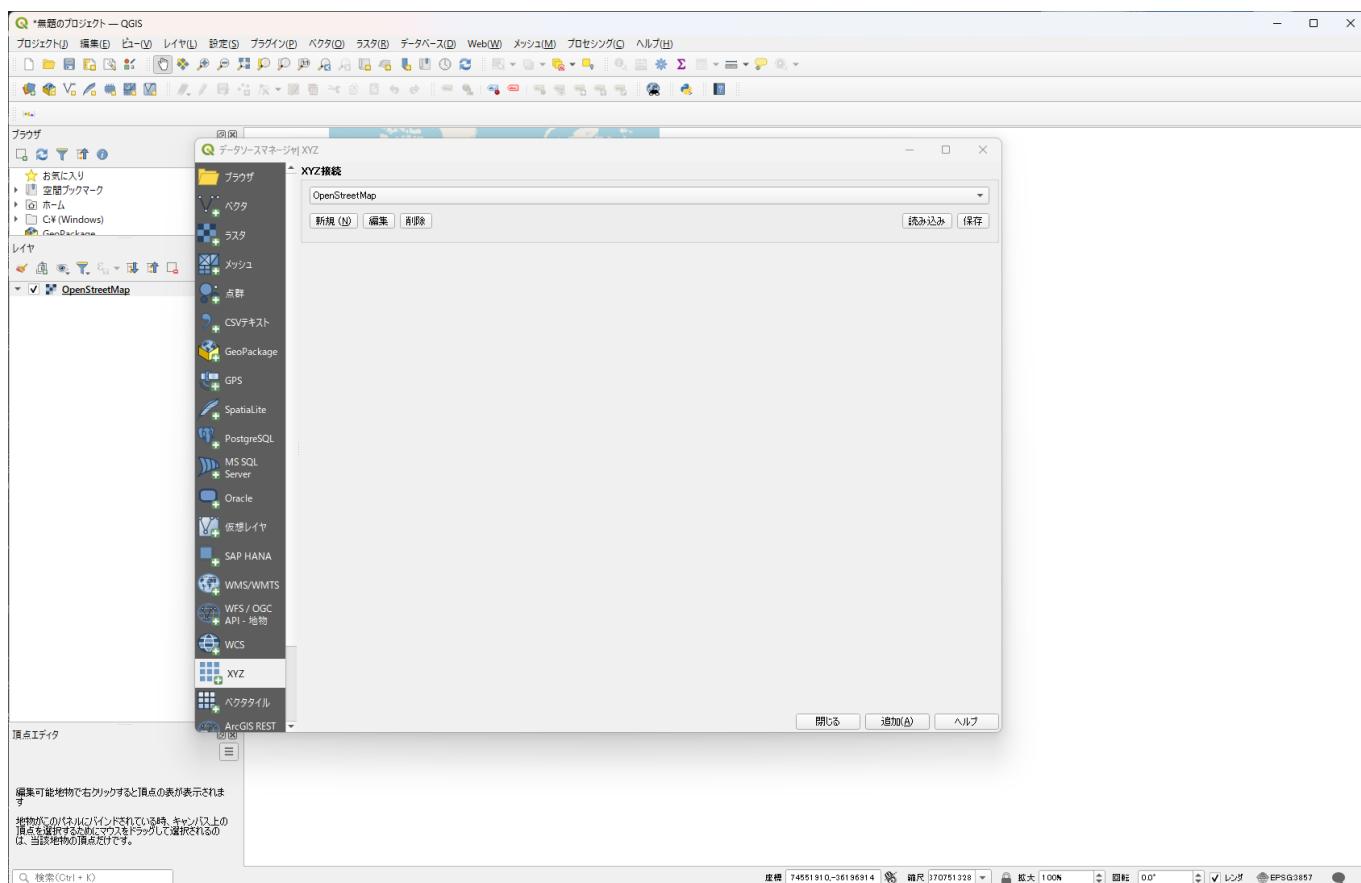
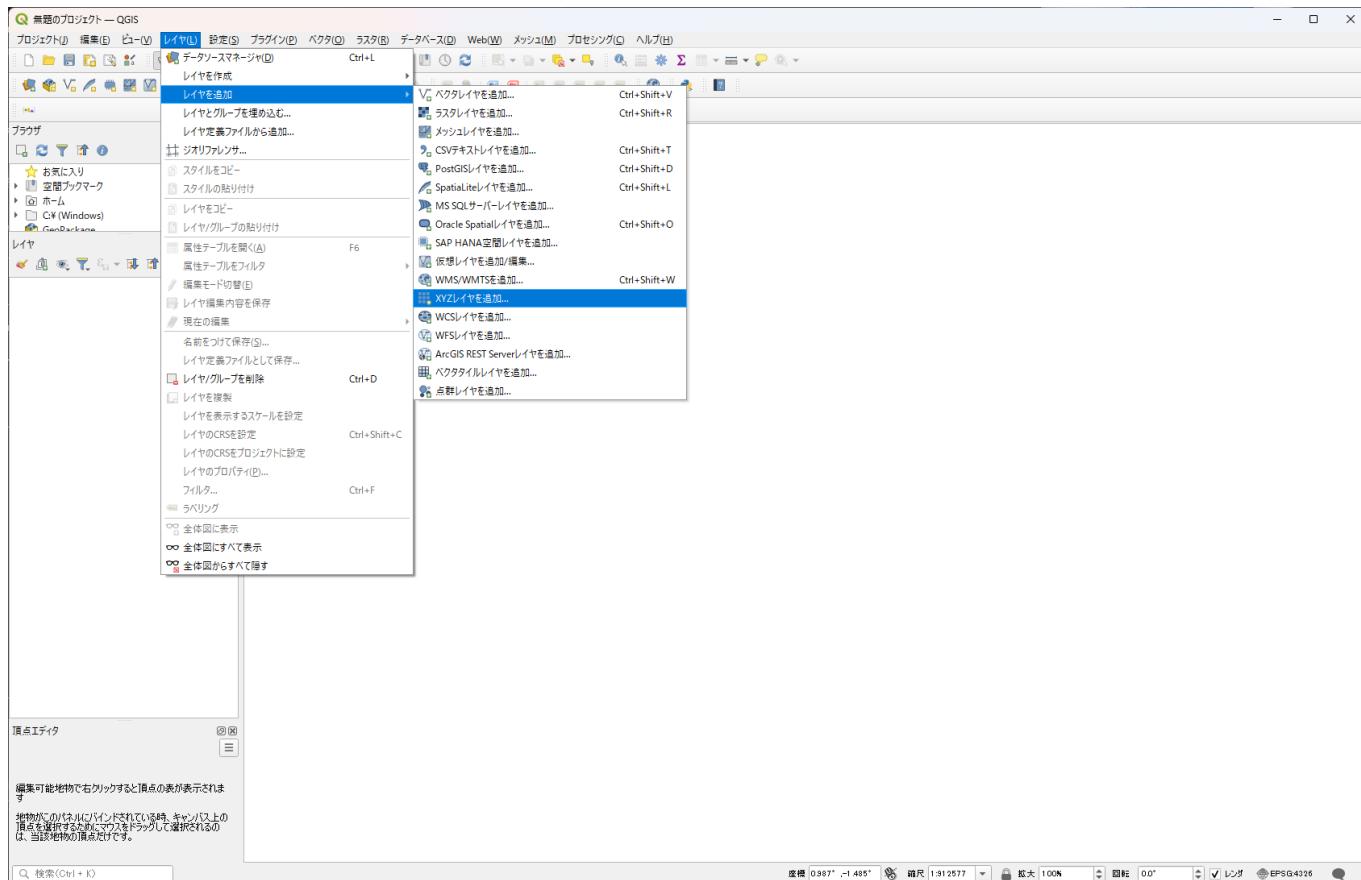
サーバー上で動作しているPostgreSQLには、作業中のPCから直接つなぐことができません。そこで、SSHのトンネリングを利用して接続します。次のコマンドを実行すると、-Lオプションで、手元のPCの63333ポートがサーバーの5432ポートとトンネリングされ、手元PCの63333ポートにアクセスすることでサーバー側のPostgreSQLの5432ポートとやり取りできるようになります。

※例はコマンドで示しますが、任意のSSHクライアントアプリで同様の設定をできます。

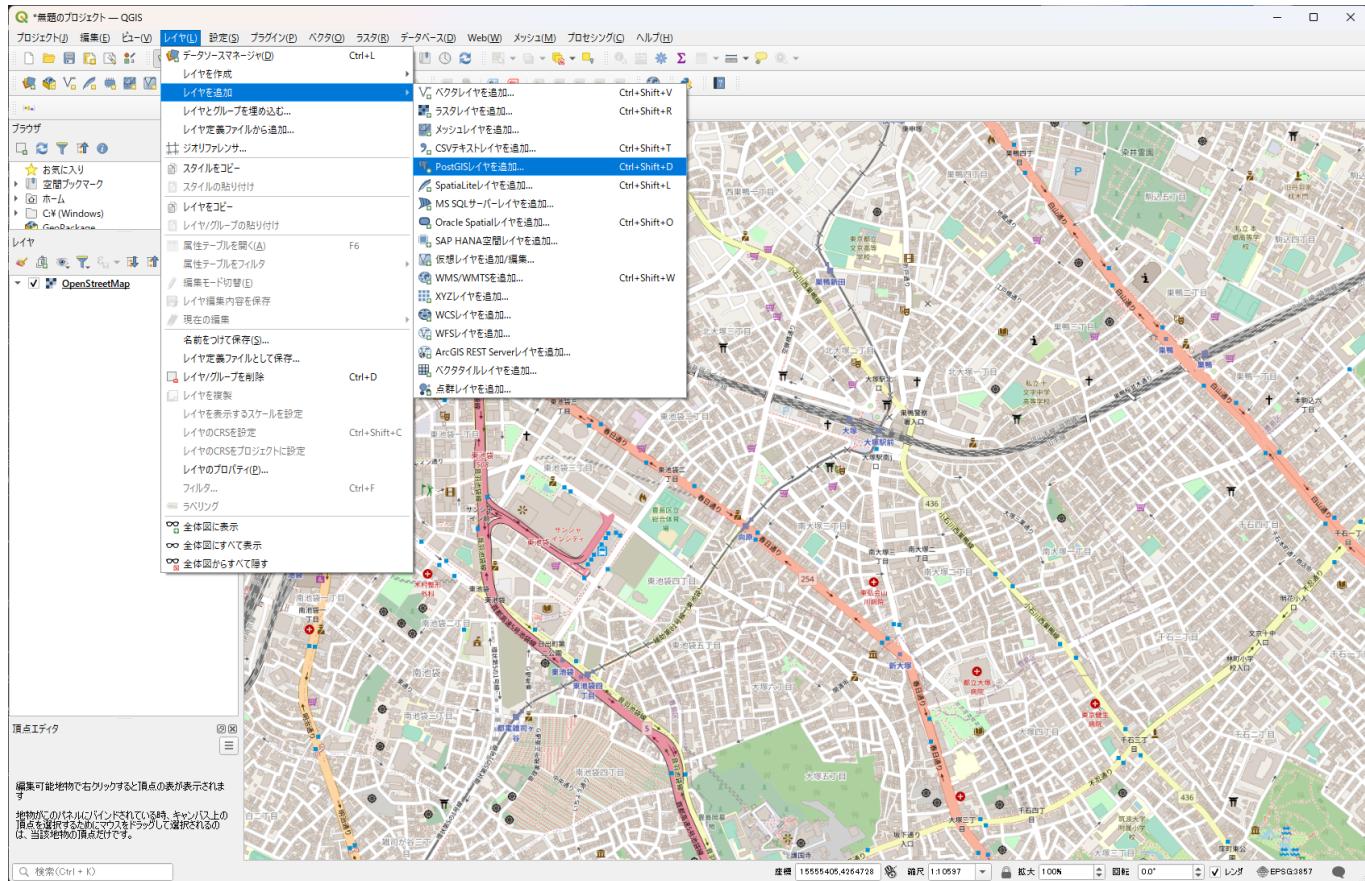
```
$ ssh -i ~/.ssh/SSH鍵ファイル -L 63333:localhost:5432 ubuntu@サーバーアドレス
```

SSHで接続したらQGISを起動し、新規プロジェクトを作成します。

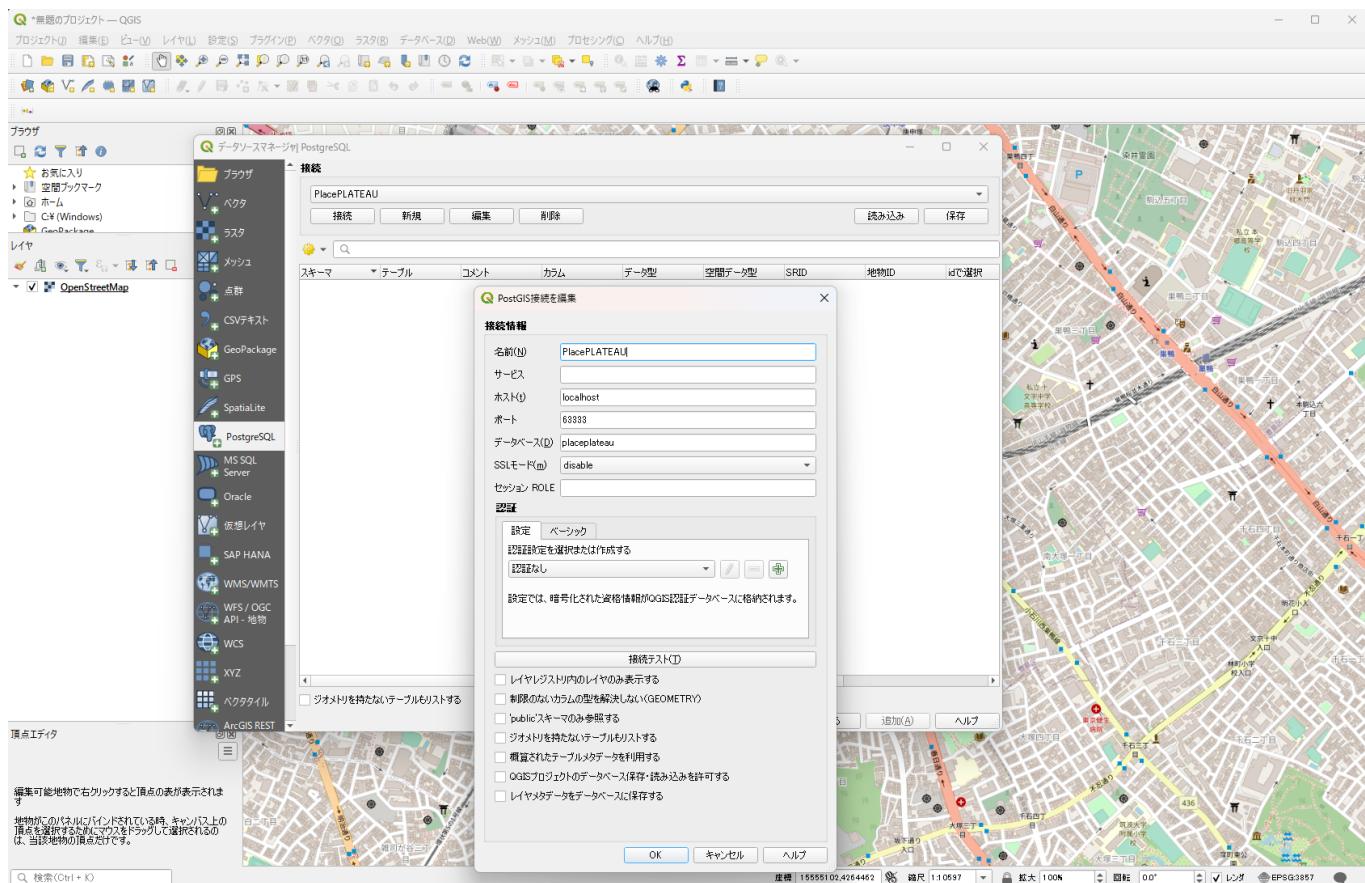
作成したら、XYZレイヤを追加でOpen Street Mapなどのベースマップを追加します。



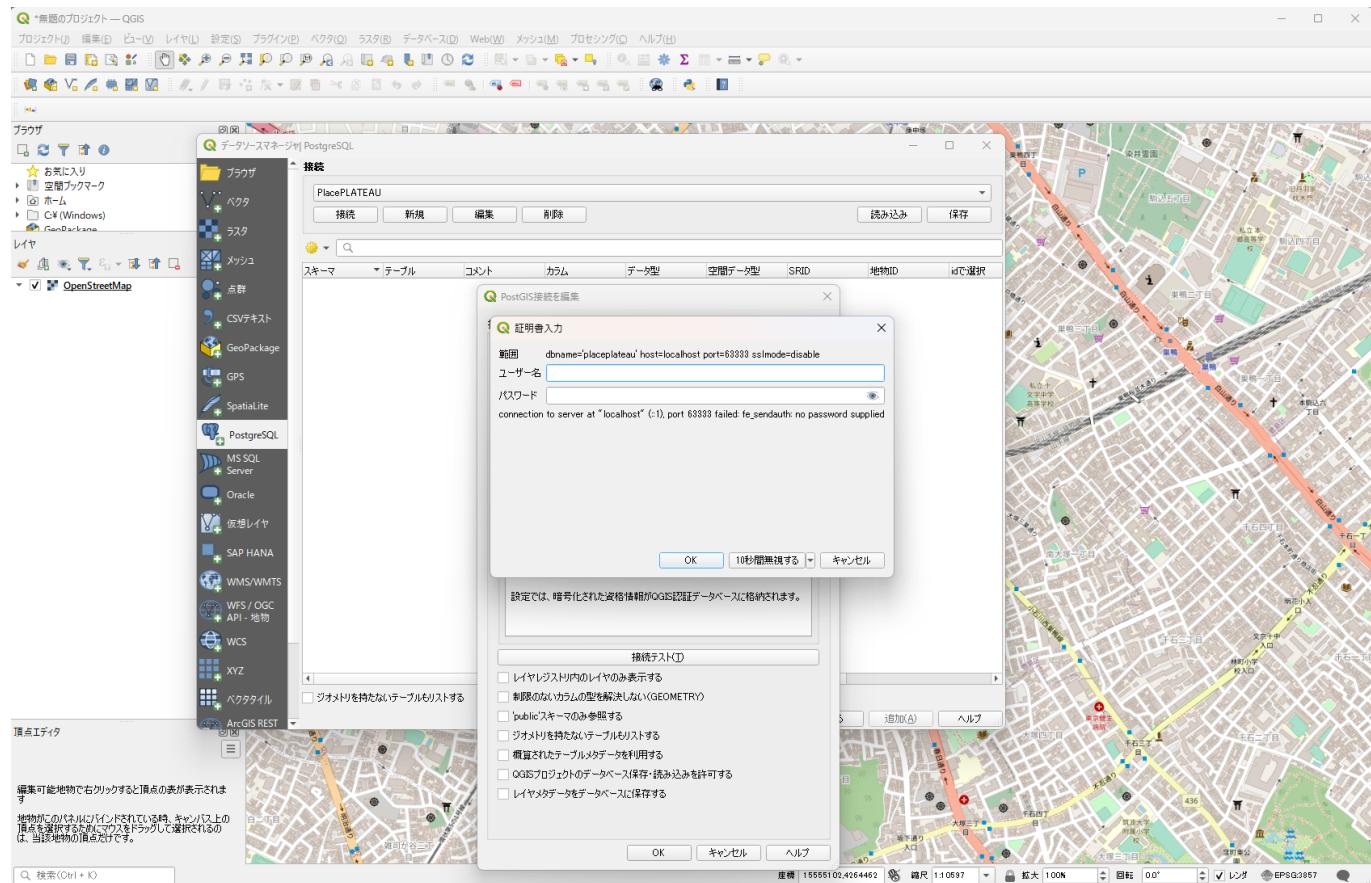
PostGISへの接続を設定します。「PostGISレイヤを追加」を選択します。



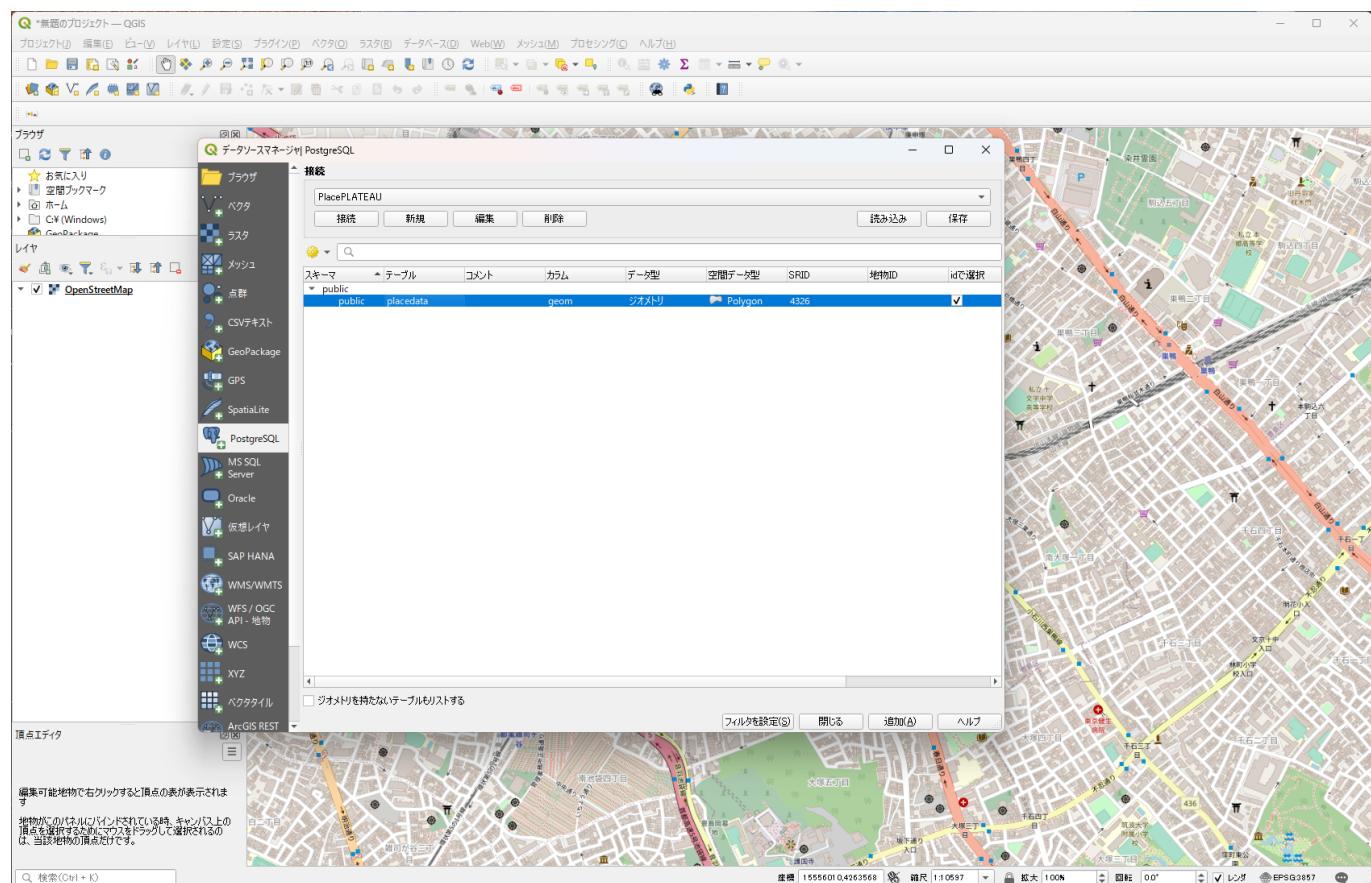
図のように、さきほどSSHで設定したポート番号などを設定します。



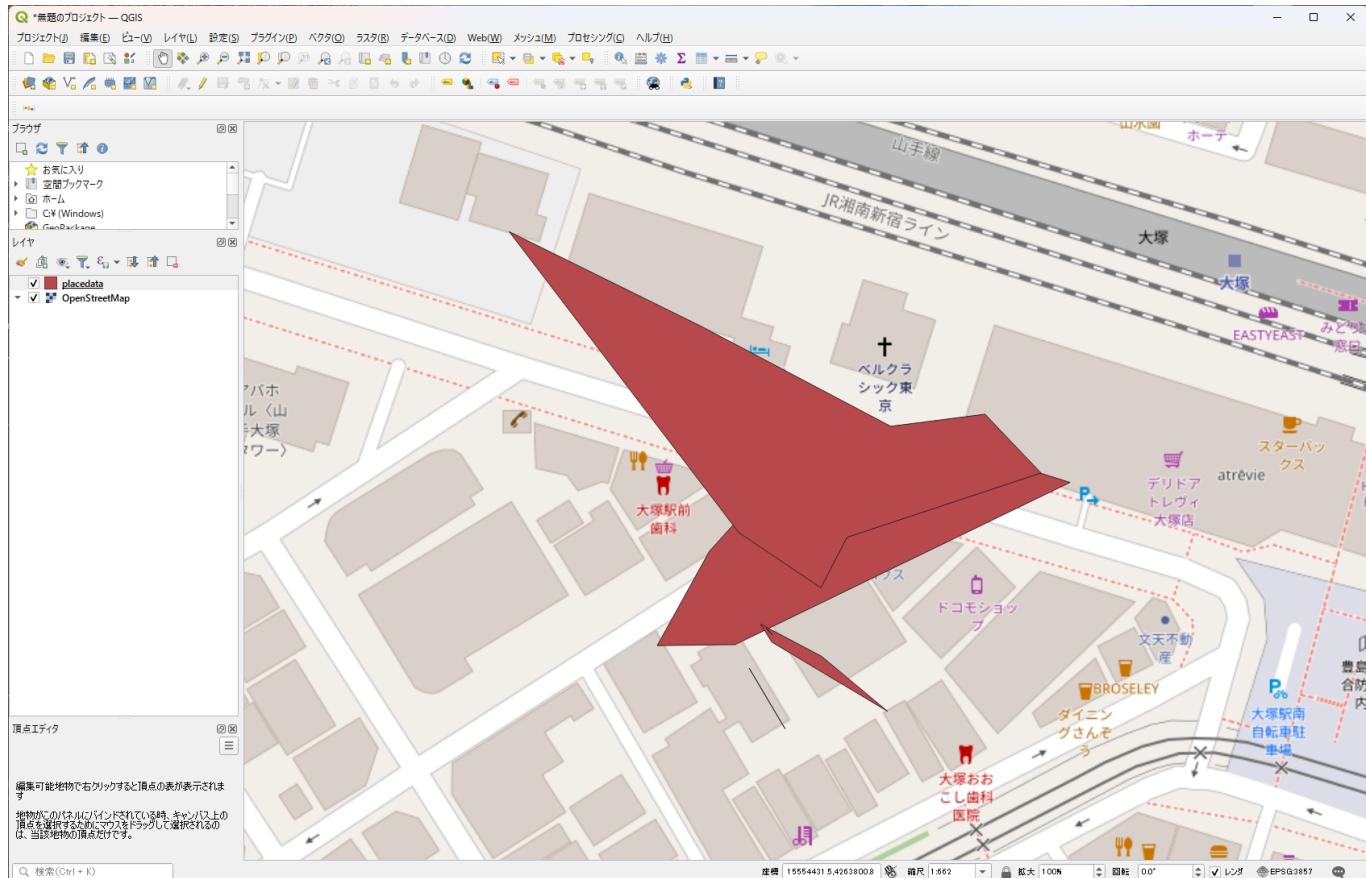
「OK」を押すと、認証情報を入力するダイアログが開くので、ユーザー名に「postgres」パスワードに設定したパスワードをいれて「OK」を押します。



「接続」を押すと、接続できるテーブルが表示されるので、選択して「追加」を押します。



これで地図上に、PostGISの地理情報が表示されました。



このデータはQGIS上で通常のベクターデータと同様に扱い、編集や分析が可能です。

## コラム チート対策

地理情報を扱うアプリ、とくにゲームなどでは、チート対策が重要になります。しかし、スマホのアプリではGPS座標を詐称するようなツールもあり、地理情報のチート対策は簡単ではありません。たとえば、現実にはあり得ないスピードで移動した場合を判定することや、システム側に現実的に侵入不可能な場所の地図を持っておいて、異常な場所にいないか判定するなどの方法がありますが、確実にチートを判定することは困難です。そのため、継続的にデータを取得しておき、統計的に怪しそうな挙動をフィルタリングするなどの後手の対策が主となります。また、取得できる報酬の上限を決めておくなどの対策も必要となります。

## コラム プライバシーについて

チート対策ともかかわりますが、個人に紐づく地理情報は個人情報となります。他人に見られないように管理すること、匿名化して統計処理することなど、取り扱いに注意すること、プライバシーポリシーの提示や取得する情報の利用目的などの明確化など、一般的な個人情報と同様な取り扱いを気を付ける必要があります。

## アプリとしてリリースする

アプリ、とくにiOSやAndroidにインストールするようなモバイル向けアプリとしてリリースするには多くのハードルがあります。もちろん、ちゃんとしたアプリとして使えるようにデザインやUXを作りこむことも大事ですし、サーバーと連携するものでは、インフラの準備なども必要です。

ここでは、一般的な流れの簡単な説明とポイントの説明、そしてGeospatial APIのAPIキーの扱いについて説明します。

各プラットフォームの規約への適合など一般的な詳細については、各プラットフォームの公式ドキュメントの最新版などを参照してください。

## APIキーの扱いについて

外部のAPIを使う場合にAPIキーをどう扱うかが課題となります。アプリ内のプログラムに埋め込んで使うこともできますが、アプリの実行バイナリはダウンロードされユーザーの端末で実行されるものなので、解析されてAPIキーが不正に取得されてしまう可能性があります。実行ファイルの難読化などの手法はありますが、100%守れる保証はありません。

仮にAPIキーが流出した場合、不正にAPIアクセスされ、情報流出や不正な利用による課金などが起こる可能性があります。対策としては、流出したAPIキーを無効にするなどの方法もありますが、アプリに組み込んでいると全て一律のAPIキーを使うことになるので、一般の利用者も使えなくなり、アプリに新しいAPIキーを含めて配信するタイミングや、ユーザーがアップデートするタイミングなどを調整しなければなりません。

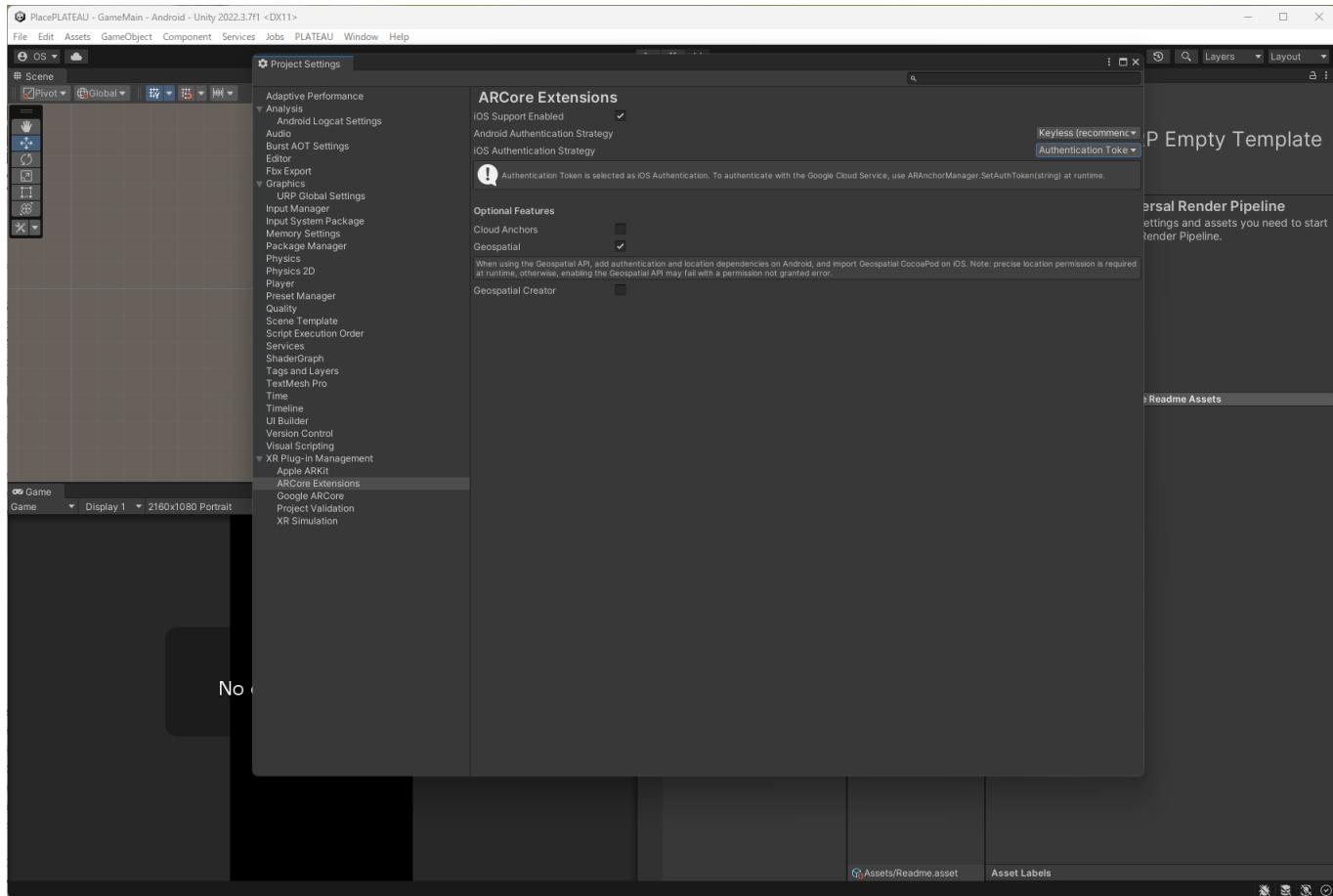
## Google Playでアプリを配信する

Android端末用のGoogle Playへの配信を説明します。

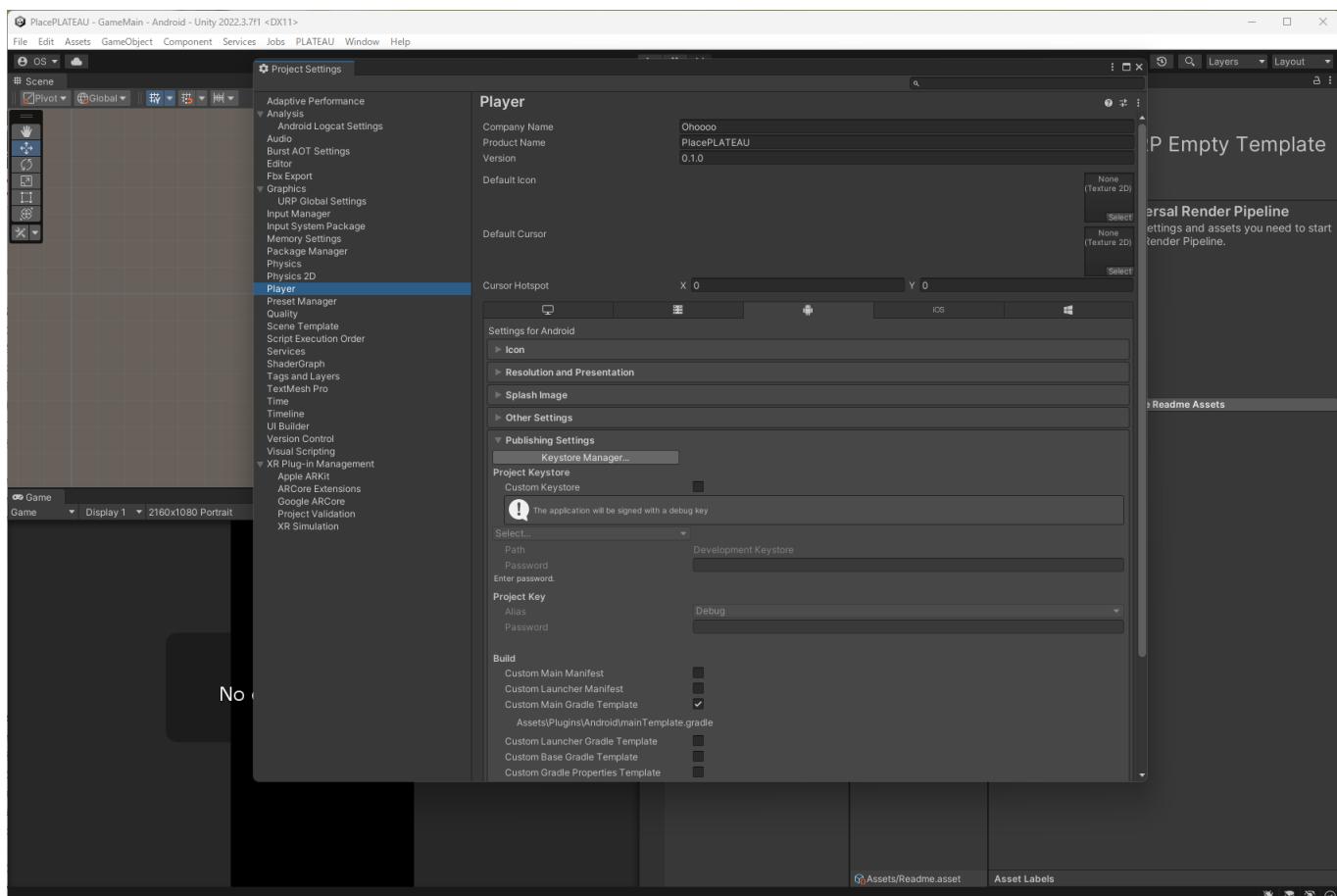
### Keyless認証の設定

Google提供のサービスのAPIキーを管理するのに、Androidでは[Keyless認証](#)を使えます。公式ドキュメントの手順に沿って設定します。

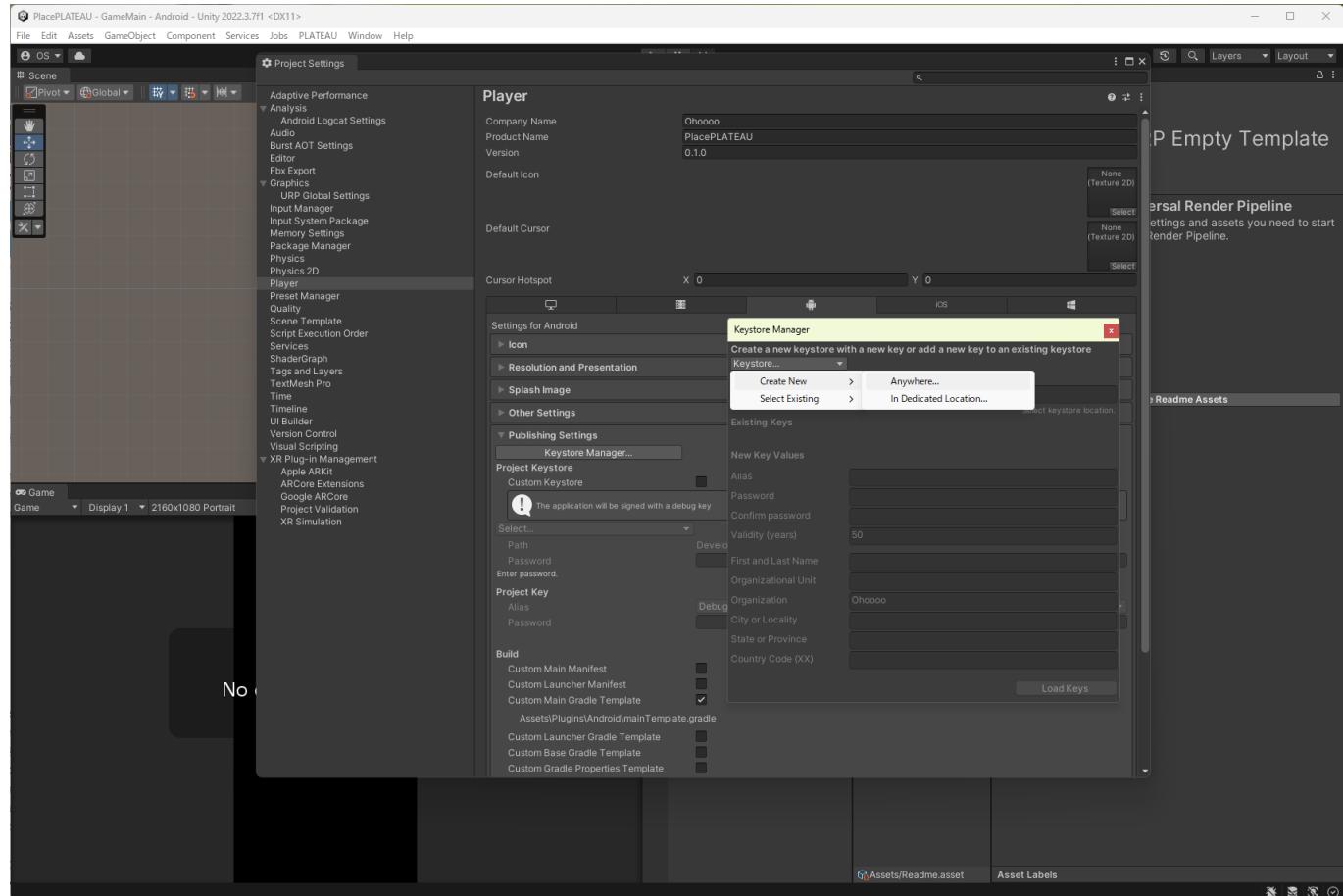
最初に「Project Settings」の「XR Plug-in Management/ARCore Extensions」の設定項目で、「Android Authentication Strategy」で「Keyless(recommended)」を選択します。



次に、「Project Settings」の「Player/Android/Publishing Settings」の「Keystore Manager」ボタンをクリックします。



「Create New/Anywhere」を選択してプロジェクト内にキーストアを作成します。このキーストアには秘密鍵が記録されるので、うっかりリポジトリに入れてGitHubで全世界に公開しないように管理します。



パスワードやエイリアスなどを適切に設定して「Add Key」を押すと自動的にキーが設定されます。

次に、作成したキーのフィンガープリントを表示します。コマンドラインでkeytoolが動作することを確認して次のコマンドを実行してください。この後の手順のためにフィンガープリントの文字列をコピーしておきます。※keytoolはJDKに含まれています。Unityのインストール時にJDKが入っていないければ、別途JDKをインストールすることでも使えるようになります。

```
$ keytool -list -v -keystore キーストアファイルのパス
キーストアのパスワードを入力してください:
キーストアのタイプ: PKCS12
キーストア・プロバイダ: SUN
```

キーストアには1エントリが含まれます

別名: forgeospatialapi  
 作成日: 2023/09/18  
 エントリ・タイプ: PrivateKeyEntry  
 証明書チェーンの長さ: 1  
 証明書[1]:  
 所有者: 0=Oho  
 発行者: 0=Oho  
 シリアル番号: 23bfbe40  
 有効期間の開始日: Mon Sep 18 16:37:06 GMT+09:00 2023 終了日: Tue Sep 05 16:37:06  
 GMT+09:00 2073

証明書のフィンガープリント:

SHA1: 3E:C8:E4:F9:C6:29:EE:65:6E:B5:BE:B6:9C:4C:35:8F:19:78:ED:AD ※←この  
文字列がフィンガープリント

SHA256:

1F:B4:9B:D1:B6:67:B4:00:AB:EE:95:5D:E8:7D:23:5C:9C:3E:D0:76:07:38:87:82:67:A3:AA:A  
3:40:90:6E:25

署名アルゴリズム名: SHA1withRSA (弱)

サブジェクト公開キー・アルゴリズム: 2048ビットRSAキー

バージョン: 3

\*\*\*\*\*

\*\*\*\*\*

Warning:

<forgeospatialapi>はSHA1withRSA署名アルゴリズムを使用しており、これはセキュリティ・リスクとみなされます。このアルゴリズムは将来の更新で無効化されます。

フィンガープリントの文字列をコピーしたら、[Google Cloud Console](#)を開きます。適切なプロジェクトを選択していることを確認して、「APIとサービス/認証情報」のページを開きます。「認証情報の作成」から「OAuth クライアントID」を選びます。

コピーしたフィンガープリントの入力とパッケージ名などを設定し、「作成」ボタンを押します。※OAuth 同意画面の設定などが出る場合があります。必須項目を適切に設定します。



Google Cloud TestGeoSpatialAPI OAuth クライアント ID の作成

API & サービス OAuth クライアント ID の作成

有効な API & サービス OAuth 2.0 の設定

認証情報

Android

名前 PlacePLATEAU

OAuth 2.0 の認証

ページの使用に関する契約

アプリケーションの種類 Android

パッケージ名 com.ohooo.PlacePLATEAU

SHA-1 証明書のフィンガープリント 3EC8E4F9C629EE656EB5BE869C4C358F1978EDAD

SHA-1 証明書のフィンガープリントによって Android アプリのみに使用を制限します。

Use this command to get the fingerprint.

\$ keytool -keystore path-to-debug-or-production-keystore -list

注: 設定が有効になるまで 5 分から数時間かかることがあります

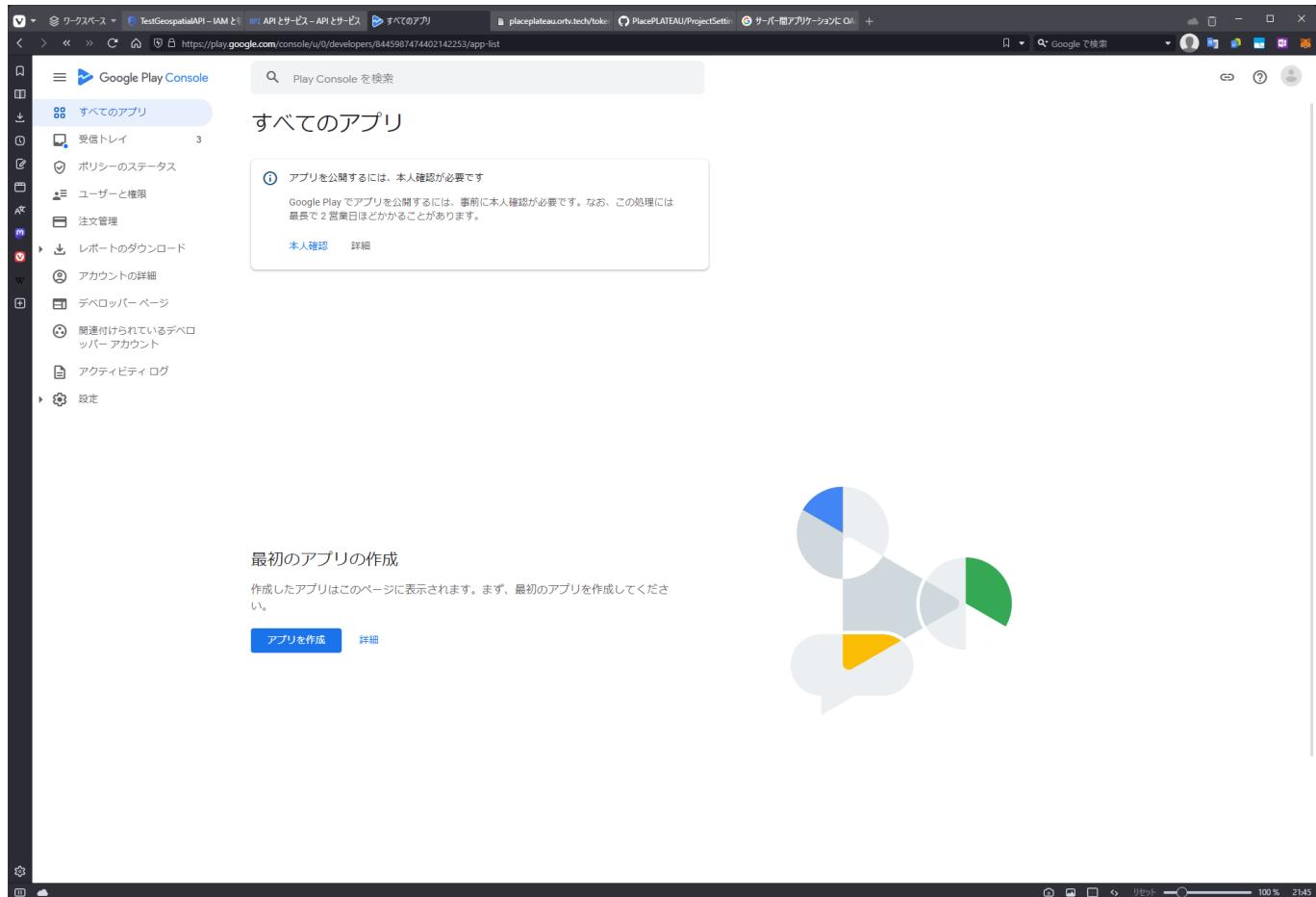
作成 キャンセル

これで、ビルドするとKeyless認証でGeospatial APIの認証が成功するようになります。キーストアのパスワードはUnityのプロジェクトを再度開くときに消えている場合があるので、うまくいかないときは入力されている状態か確認してください。

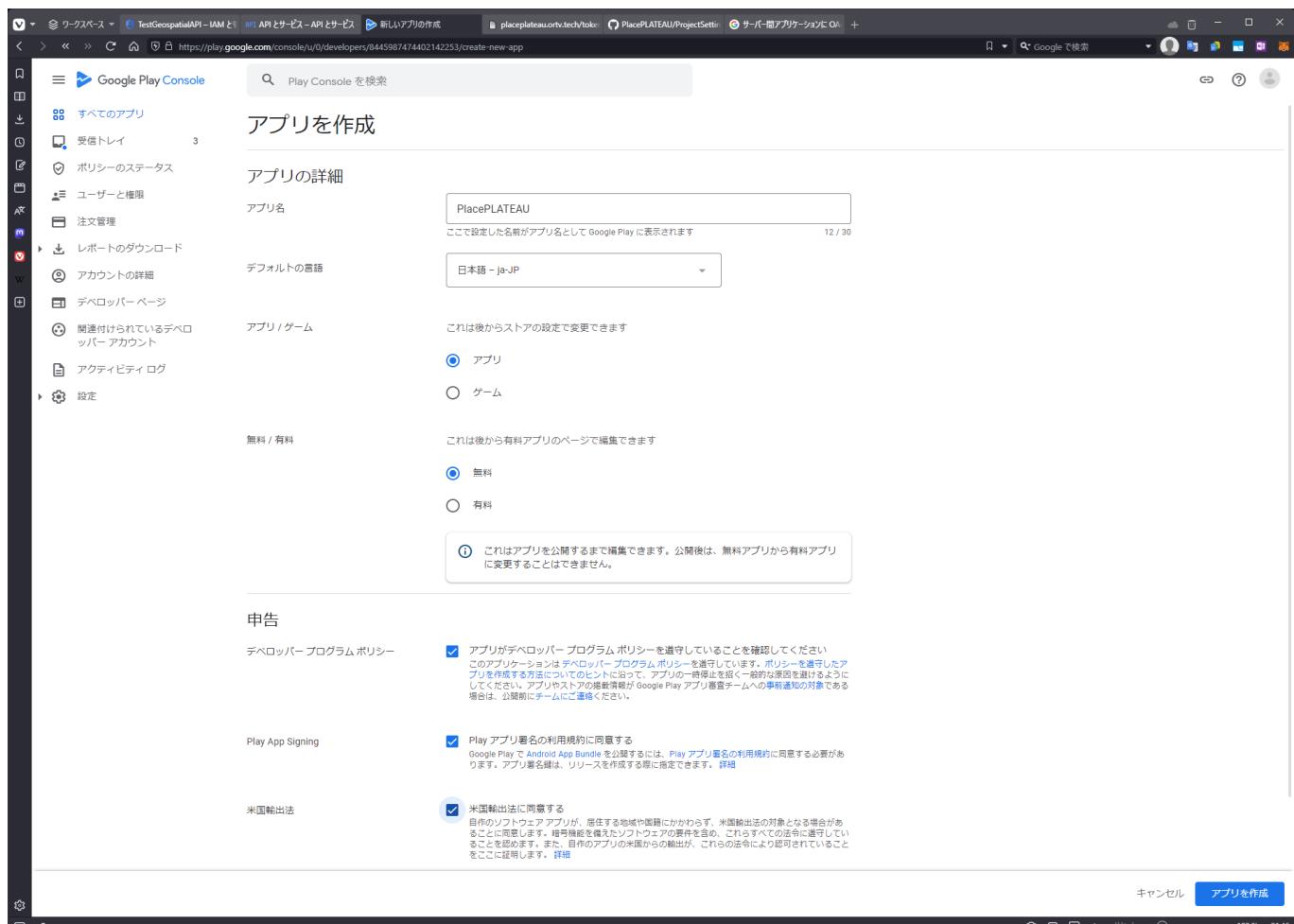
## Google Playでの配信

Google Play Consoleにアクセスし、「Play Consoleに移動」からログインします。この時、はじめての場合はユーザー登録から始まります。個人でアプリを公開する場合は、個人ユーザーの選択肢を選ぶとよいでしょう。連絡先の情報などを入力します。登録の中で\$25の登録料の支払いが必要になるので注意してください。また、実際にストアにアプリを配信する場合、本人確認の手続きが必要です。

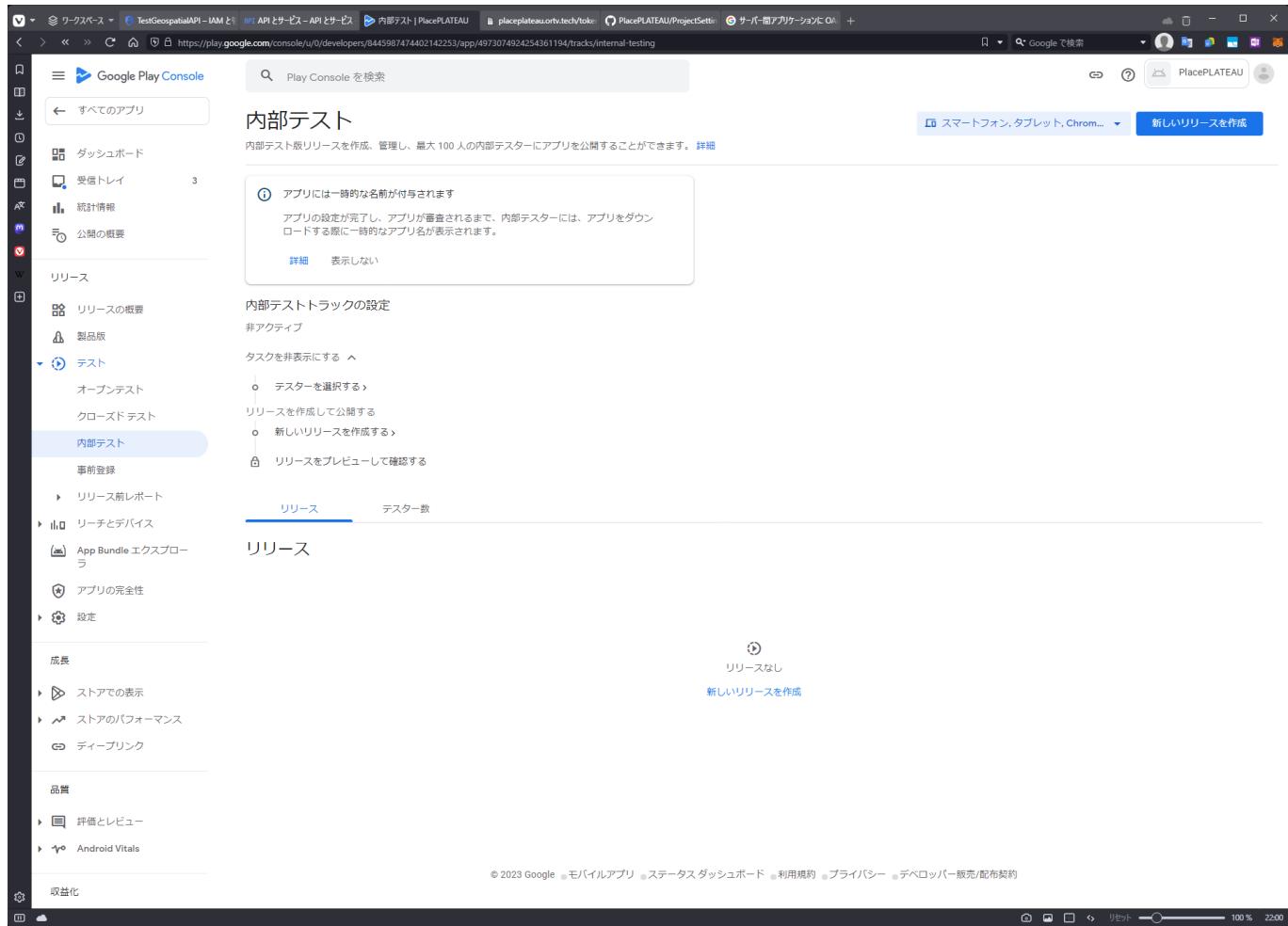
ユーザー登録出来たら、アプリの情報を登録していきます。「すべてのアプリ」ページで、「アプリを作成」ボタンを押します。



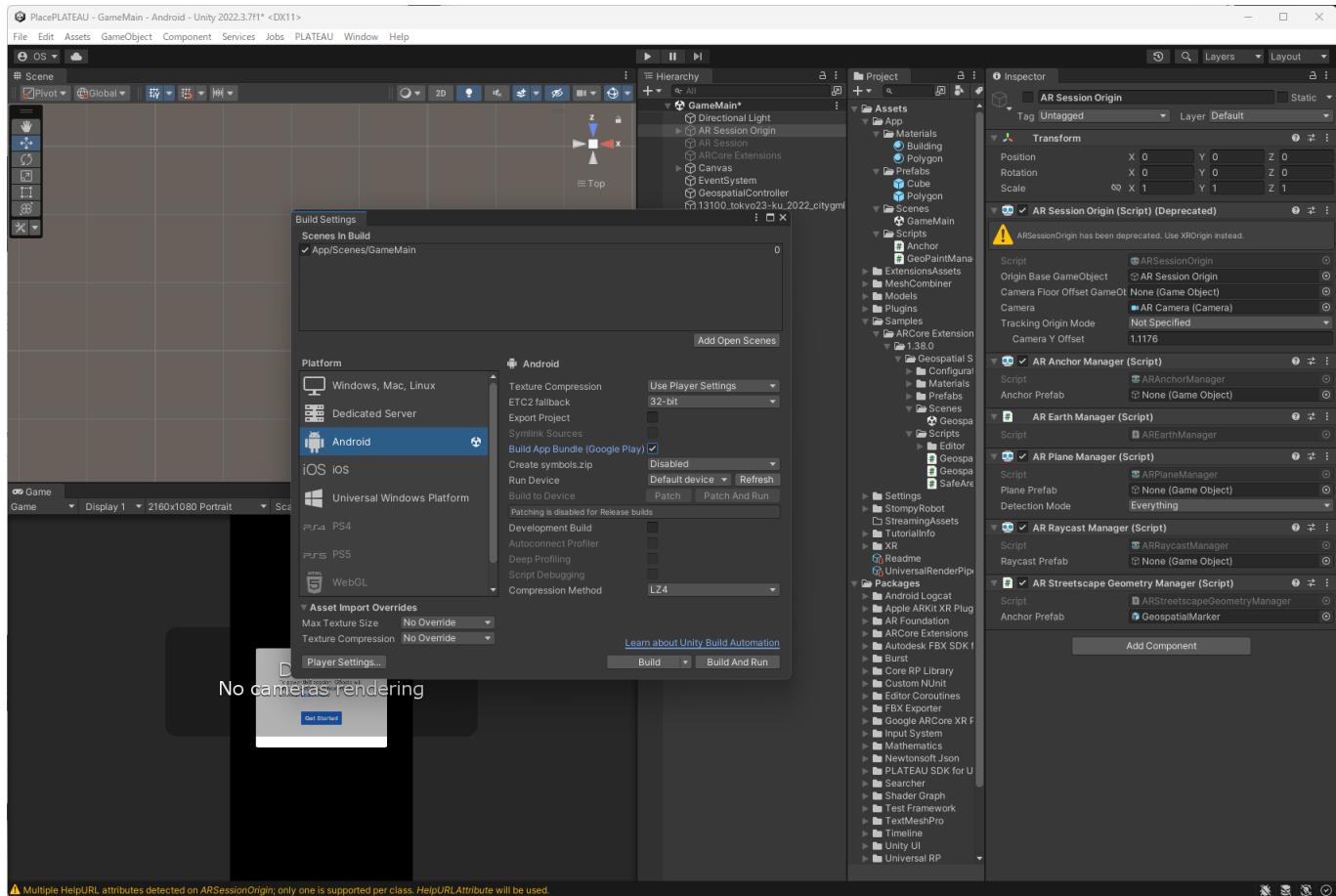
アプリの名前などの情報を登録し、「アプリを作成」を押します。



まずは内部テスト版を作成します。「テスト/内部テスト」から「新しいリリースを作成」を押します。



Unityのビルド設定で「Build App Bundle (Google Play)」にチェックしてリリース設定でビルドします。

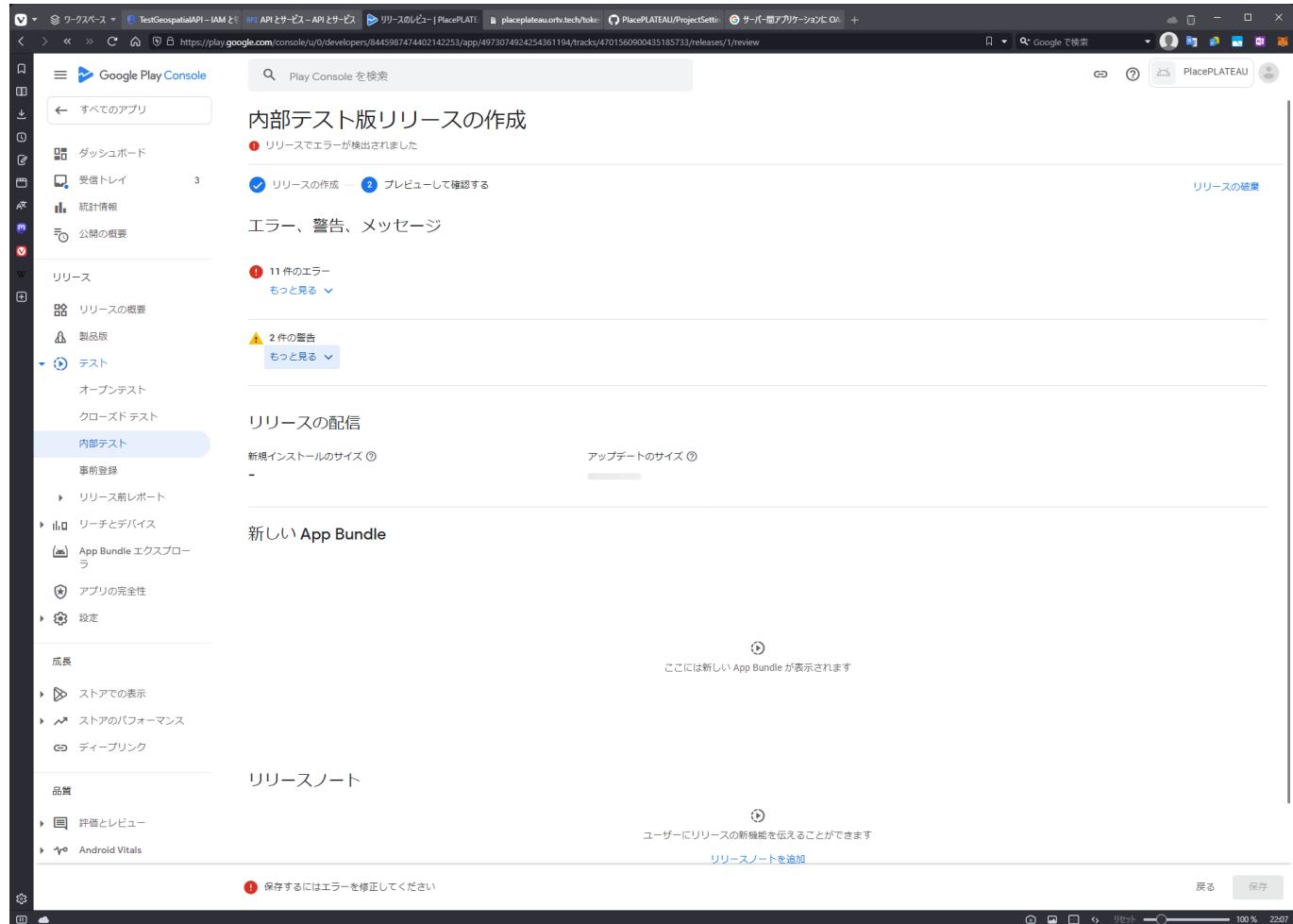


⚠ Multiple HelpURL attributes detected on ARSessionOrigin; only one is supported per class. HelpURLAttribute will be used.

できた.aabファイルをアップロードします。また、リリース名などの情報も入力し、「次へ」を押します。



エラーや警告が表示されています。これらはアップロードしたアプリや開発者ユーザーのアカウントの状況で異なるので、適宜メッセージを確認しながら解決します。



同様の手順を製品版リリースで行い、Googleの審査に合格すると製品版としてGoogle Playストアで公開することができます。

Google Play Consoleではアプリのリリースだけでなく、広告や課金の設定や、インストール数の確認、レビューの確認などの様々な機能があります。使いこなしてアプリをよりよいものに改善してください。

## AppStoreでアプリを配信する

iOS端末用のAppStoreへの配信を説明します。

### JWTでのトークン認証

iOS端末では、トークン認証を使います。これは、ユーザーごとの短期間の認証トークンをサーバー側で生成することで、アプリ内にAPIキーを内包しなくてよい仕組みです。サービス側のサーバーに認証用の秘密鍵を置き、APIなどで、サーバー側の処理で秘密鍵から有効期限が短い認証トークンを発行。それをクライアント側で受け取りGoogleAPIの認証に使います。こうすることで、サーバー側の認証により正しいユーザーのみGoogleのAPIを適正に使うことができます。また、短有効期間のトークンなので、不正に入手されても利用には大きな制限がかかります。一方で、サーバーの用意が必要なことなど、APIキーによる認証より開発・運用に手間がかかります。 [公式ドキュメント](#)を参考にしてください。

最初に、Google Cloud Consoleでサービスアカウントを作成します。「APIとサービス/認証情報」ページから「認証情報を作成」の「サービスアカウント」を選択します。

サービスアカウント名など、必要な情報を入力して進めます。「ロール」には、「サービス アカウント トークン作成者」を設定します。

認証情報ページに戻ると、リストに作成されたサービスアカウントが表示されているので、クリックして表示されたページで、「キー」を選択します。

ここで、「鍵を追加/新しい鍵を作成」で「JSON」を選び、作成します。自動的にJSONファイルがダウンロードされるので、秘密鍵として権限のない人がアクセスできないように保管します。

次に、サーバー側のプログラムを作成します。前節で作成したPythonのプログラムに、トークンを生成する機能を追加します。次のように、importの追加、初期化コードとメソッドの追加を行います。

```
from datetime import datetime, date, timedelta
import jwt

.....
with open ('testgeospatialapi-165ee1b61a8f.json', 'r') as f:
    service_account_info = json.load(f)

private_key = service_account_info['private_key']
client_email = service_account_info['client_email']

.....
# GeospatialAPIの認証トークンの作成 (※実際にサービスに使用する際は認証の仕組みが必要)
@app.route('/token', methods=['GET'])
def token():
    data = request.args.get('key', '')
    if data != 'password':
        return 'error'

    current_time = datetime.now()
    expiration_time = current_time + timedelta(hours=1)

    payload = {
        'iss': client_email,
        'sub': client_email,
        'iat': int(current_time.timestamp()),
        'exp': int(expiration_time.timestamp()),
        'aud': 'https://arcore.googleapis.com/',
    }

    token = jwt.encode(payload, private_key, algorithm='RS256')
    return token
```

ここでは認証は形だけのものですが、実際のサービスでは「必ず」ユーザー単位のちゃんとした認証をしてトークンの発行を行います。

この状態でサーバーにアクセスすると、認証用トークンの文字列が返ってきます。



The screenshot shows a browser window with the URL <https://placeplateau.ortv.tech/token?key=password>. The page content is a single, extremely long string of characters representing a JSON Web Token (JWT). The string starts with 'eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9.eyJpc3MiJoIj0ZXN0Z2Vvc3BhdGhbGFwaUB0ZXN0Z2Vvc3BhdGhbGFwaS5pYW0uZ3NlcnPzY2VhY2NvdW50LmNvbSIsInN1YiI6InRlc3RnZW9zcGF0aWFsYX8pQHRIc3RnZW9zcGF0aWFsYX8pLmhbS5nc4DN1qE5sebSheKeTlGuuy3kSs1Djc-LehtrJ7V2mzb8Z-mkFVezfLlyJWlEqyp4mZlPxRWr1nvPLYT1SlxzBD4faFS82lcrh-052f3H3c0RzC7IM5jNXmDv9M8PW9616YAc5CFygbti0xCZ3ygDyaWyla6QfPmDcyFZHF61UX-Z2qf0h3bc-npfSUbmxVHRL77Fye9OkkHeXhgp-Sh\_di63q27Vkuo4EUQiPl24rQvYnI0-9vsXsgwHTXR3He\_\_e8kjgQ2\_\_uFwtF3p3J4XYIjUcX4RODmeXe4wQsVgSNkFA'.

これをUnity側で取得してAPI認証をします。GeoPaintManager.csに次のメソッドを追加します。トークン認証はiOSの時だけなので、#ifでiOSの時だけ有効になるようにしています。

```
#if UNITY_IOS
    public static IEnumerator GetToken(ARAnchorManager arAnchorManager)
    {
        UnityWebRequest req = new UnityWebRequest(url + "token?key=password",
"GET");
        req.downloadHandler = (DownloadHandler)new DownloadHandlerBuffer();

        yield return req.SendWebRequest();

        if (req.result == UnityWebRequest.Result.Success)
        {
            var token = req.downloadHandler.text.Trim();
            Debug.Log(token);
            ARAnchorManagerExtensions.SetAuthToken(arAnchorManager, token);
        }
        else
        {
            Debug.LogError("Error sending GET request: " + req.error);
        }
    }
#endif
```

また、「Assets/Samples/ARCore Extensions/1.38.0/Geospatial Samples/Scripts/」にある、GeospatialController.csを修正します。AvailabilityCheckメソッド内にトークンを取得し認証するコードを追加します。

```
if (Input.location.status != LocationServiceStatus.Running)
{
    Debug.LogWarning(
        "Location services aren't running. VPS availability check is
not available.");
    yield break;
}

// Update event is executed before coroutines so it checks the latest
error states.
if (_isReturning)
{
    yield break;
}

-----ここから
#if UNITY_IOS
    yield return GeoPaintManager.GetToken(AnchorManager);
#endif
-----ここまで追加

var location = Input.location.lastData;
var vpsAvailabilityPromise =
```

```

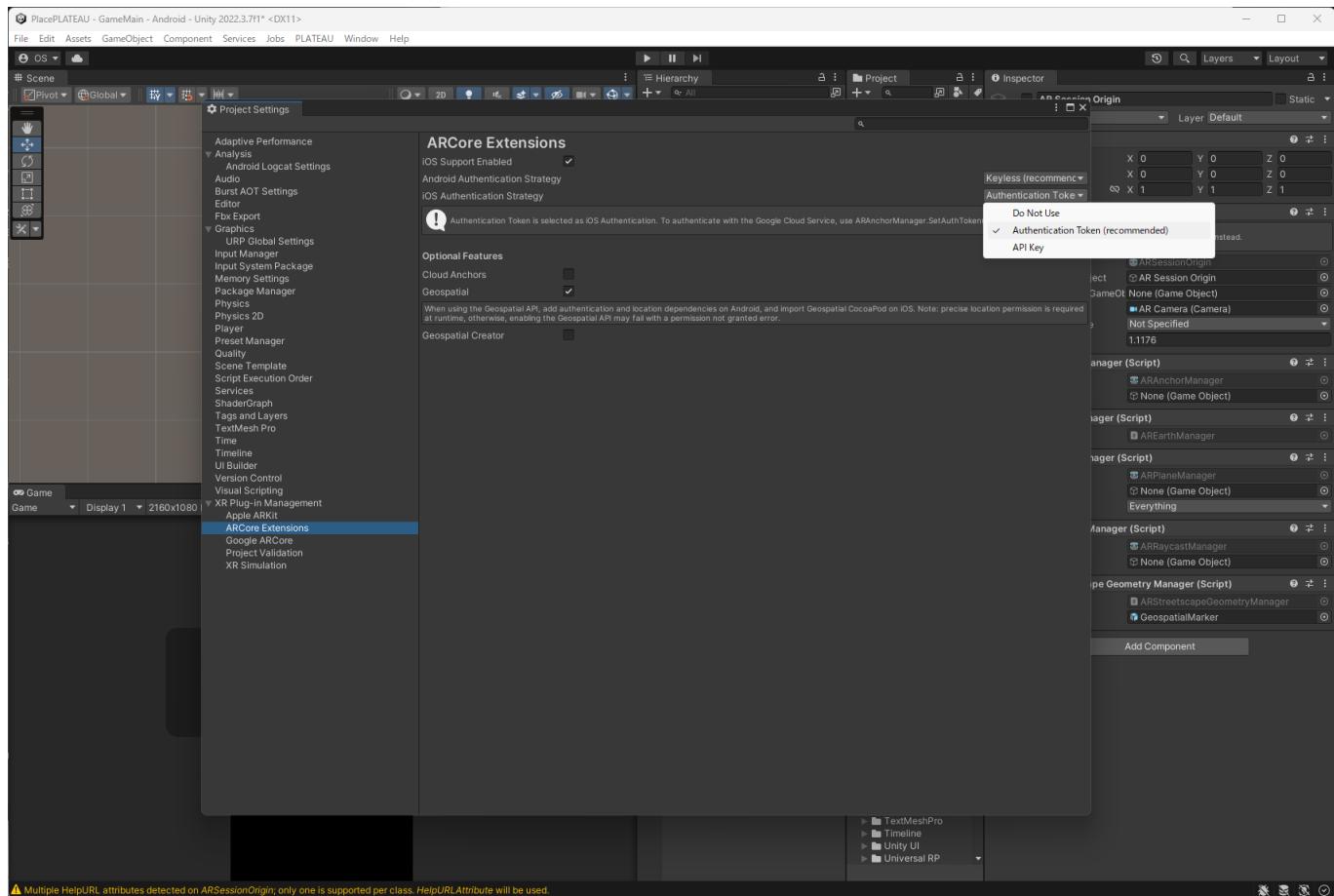
        AREarthManager.CheckVpsAvailabilityAsync(location.latitude,
location.longitude);
        yield return vpsAvailabilityPromise;

        Debug.LogFormat("VPS Availability at ({0}, {1}): {2}",
location.latitude, location.longitude,
vpsAvailabilityPromise.Result);
        VPSCheckCanvas.SetActive(vpsAvailabilityPromise.Result !=
VpsAvailability.Available);

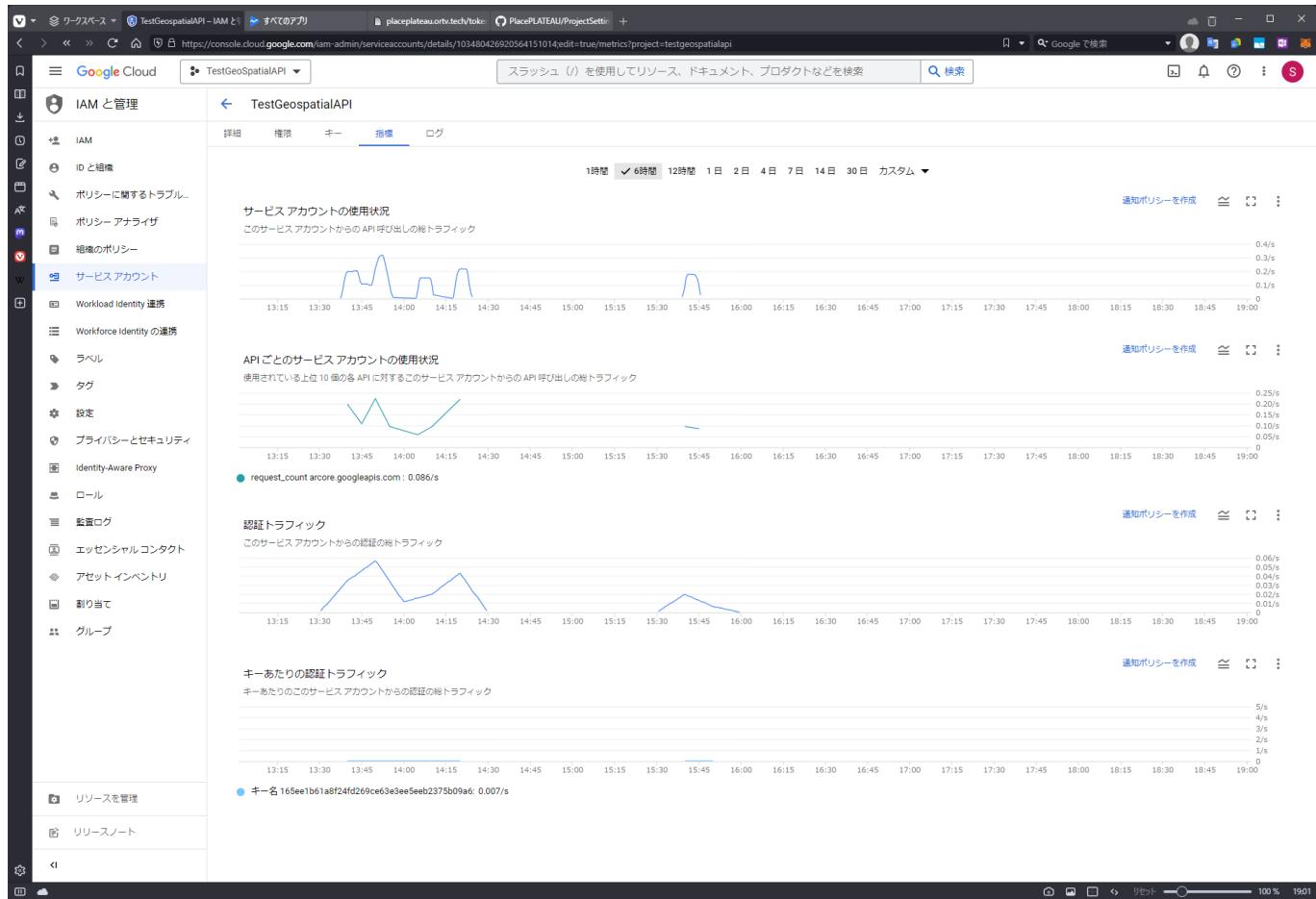
```

ここが連携していないと、GeospatialController.csの方で、認証が通っていないとされてしまうため、このように認証が通った後にGeospatialAPIの有効性チェックが動作するように流れを修正します。

最後に、ARCore Extensionsの設定で、iOSのトークン認証を有効にします。



以上でビルドして動作することを確認しましょう。また、Google Cloud Consoleで「IAMと管理/サービスアカウント」で個々のサービスアカウントを選択し、指標タブを見ると、APIが使用されているかを確認できます。活用してください。



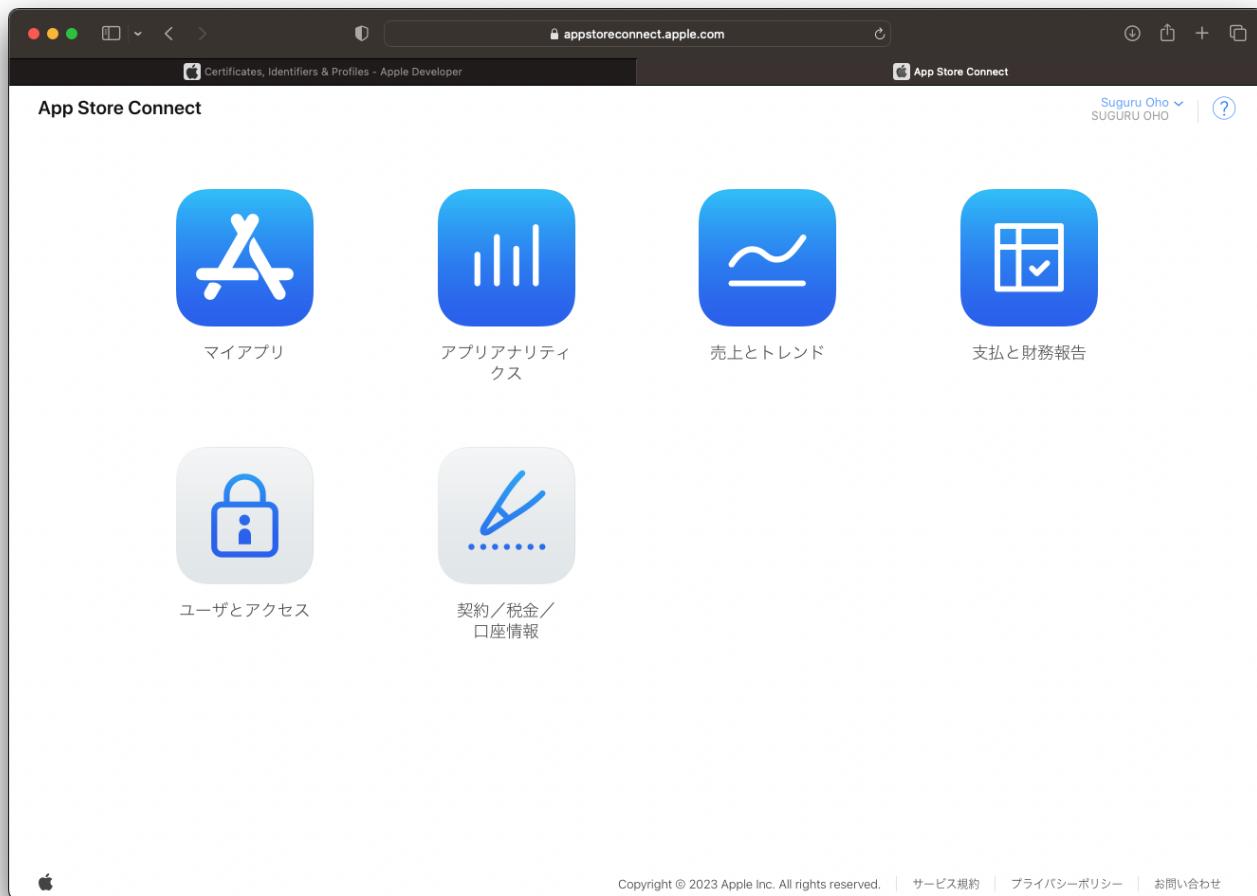
## AppStoreでの配信

AppStoreで配信するためには、Apple Developer Programへの登録が必要です。また、個人としての登録で年間登録料として1万円ほどかかります。ここではApple Developer Programへの登録が完了している前提でアプリの登録作業を見ていきます。

まず、[Appleの開発者サイト](#)で必要なファイルを作成します。

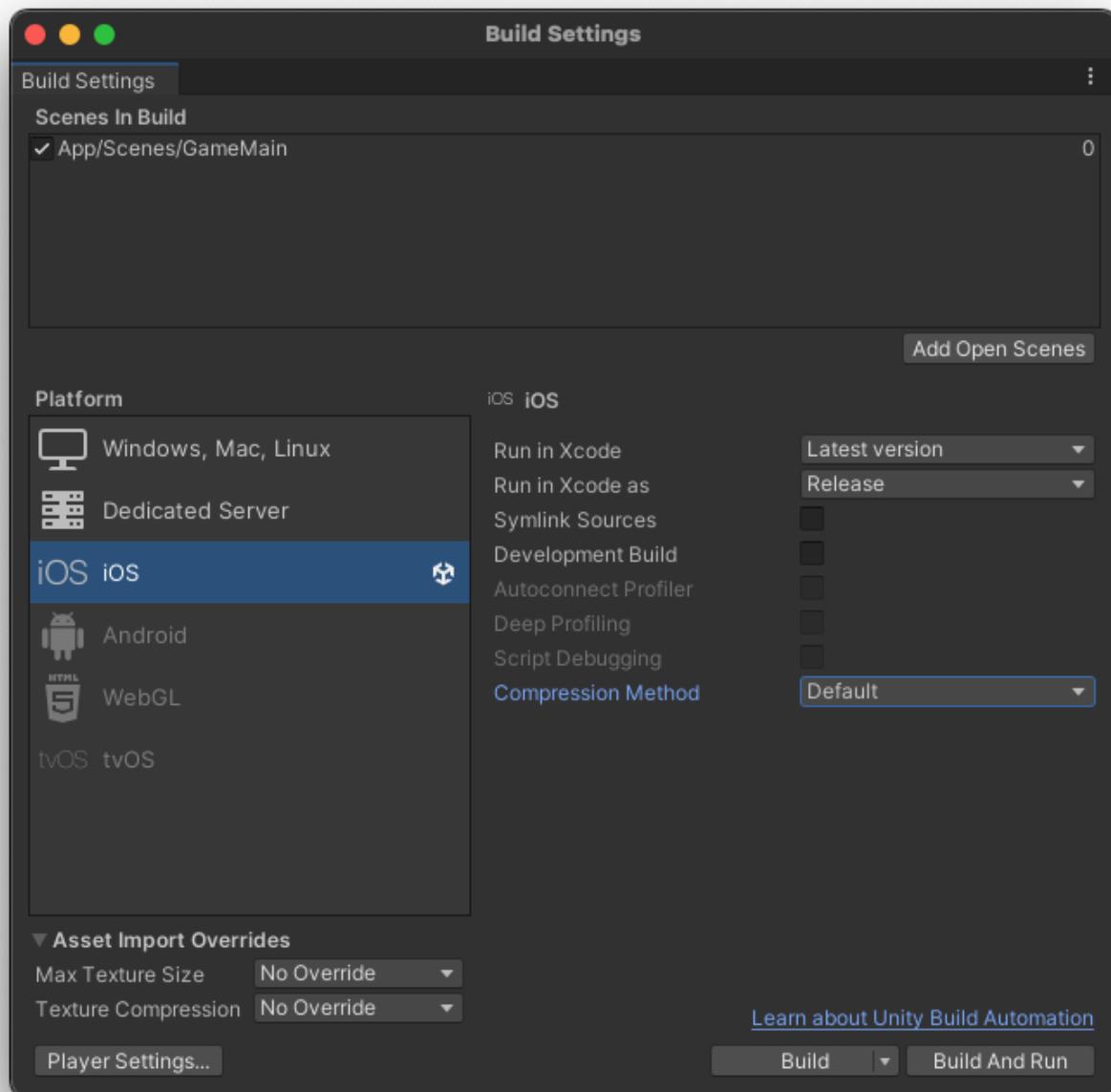
必要なものは、証明書、アプリID、プロファイルの3つです。開発用にはDeveloper、配布用にはDistributeのものが必要になります。 [公式のドキュメント](#)に詳しい方法が載っているので、そちらを参照してください。

同様に、App Store Connectの方でも設定を進めます。「アプリ」ページでアプリを新規作成します。App Store Connectも[公式ドキュメント](#)を参照してください。



AppStoreの方は、Google Playと違いXCodeから作成したアプリのパッケージファイルを直接アップロードします。

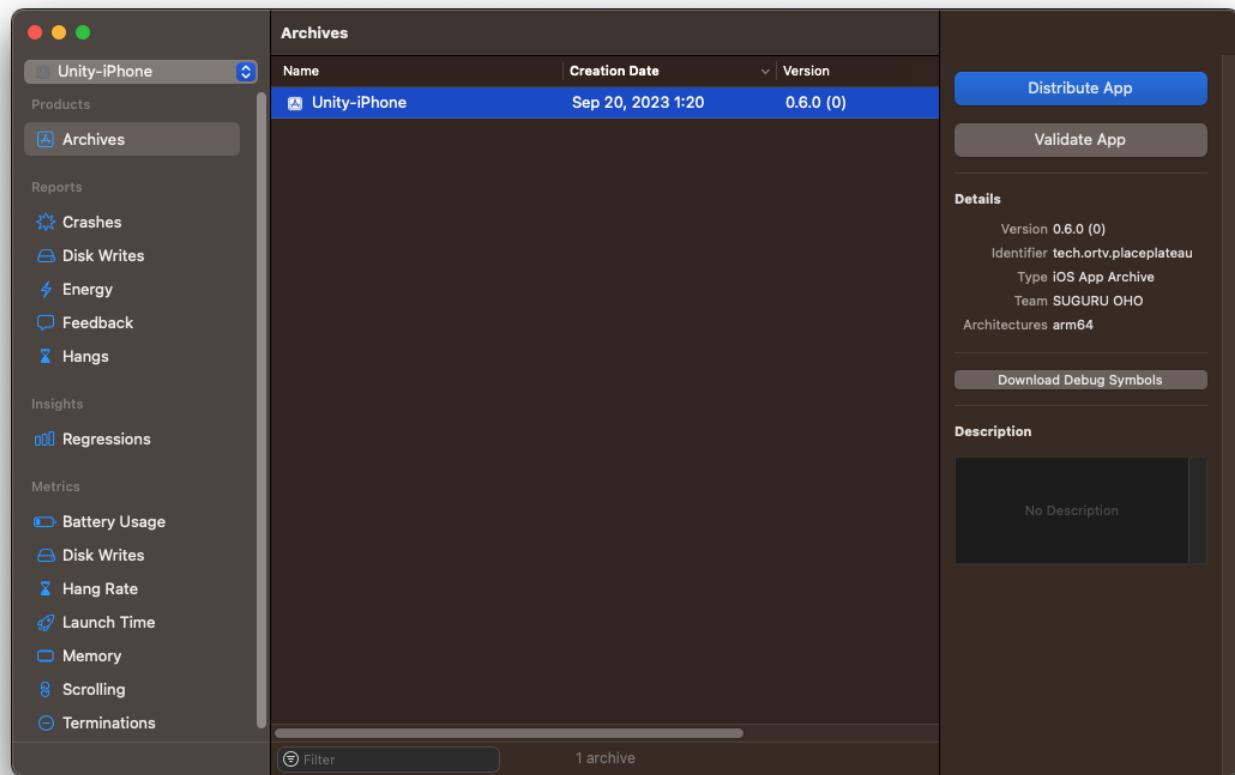
UnityでRelease設定でのiOS向けのビルドをします。



Unityのビルドが成功したらXCodeでプロジェクトを開きます。この時、事前にDeveloperサイトやApp Store Connectで設定したバンドルID、ダウンロードしたProvisioning Profileが設定され、証明書がインストールされて自分のチームが設定されていることを確認します。

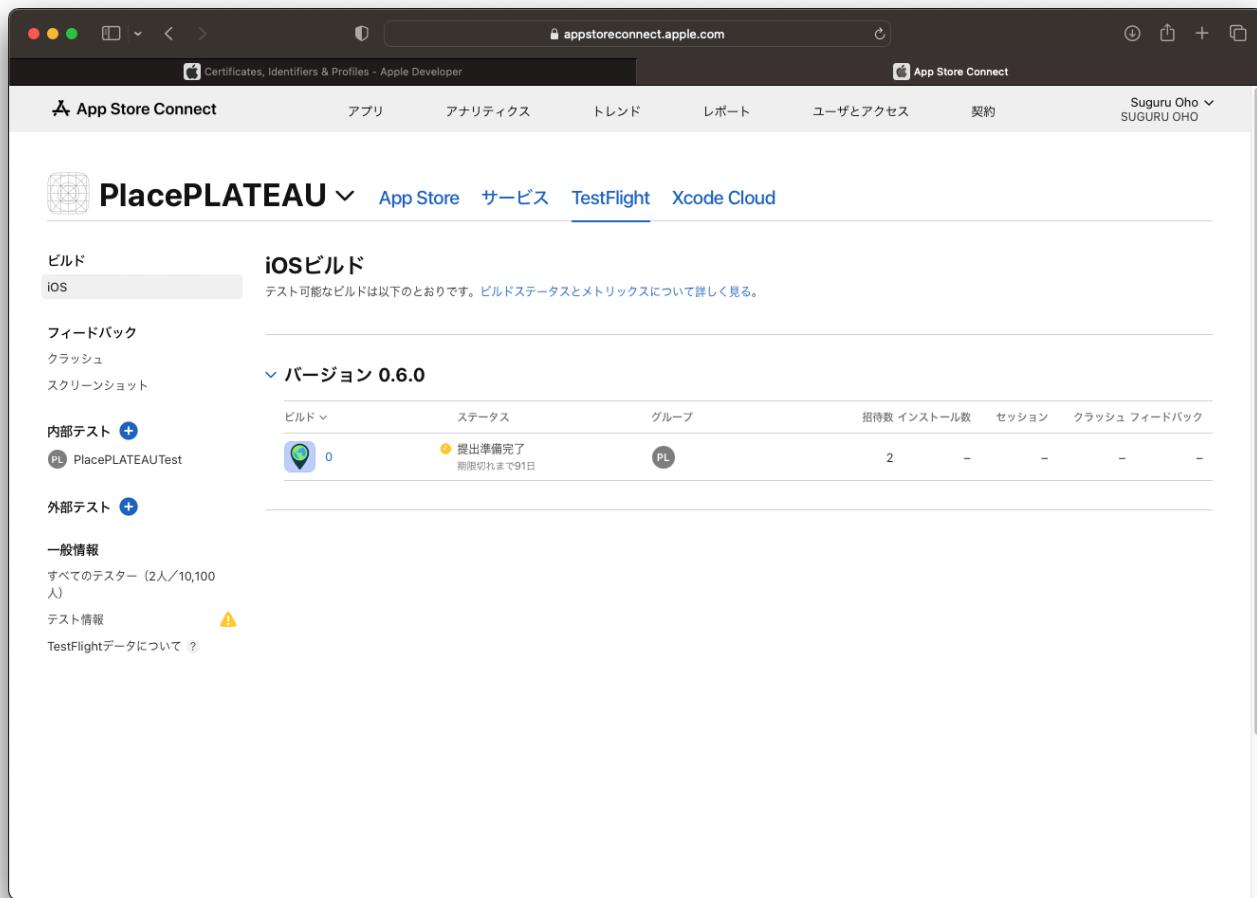
ひと通り設定を確認したら一度正常にビルドできるか確かめてみます。問題ないようなら、XCodeのメニューから「Product/Archive」を選び、App Store Connectにアップロードするためのビルドを作ります。

パッケージ作成が終わると、Archivesのウィンドウが開くので、「Distribute App」ボタンを押して画面の指示に従ってビルドをApp Store Connectにアップロードします。



しばらくすると、App Store Connectの方にアップロードしたビルドが表示されます。

最初はTestFlightにアップロードしたビルドが表示されます。この状態であらかじめ招待した少数のテスターで行う内部テストを実施することができます。また、審査に提出することで、不特定多数に向けた外部テストを行うこともできます。



The screenshot shows the App Store Connect interface for the PlacePLATEAU app. The left sidebar lists build types: ビルド (Build) with iOS selected, フィードバック (Feedback), クラッシュ (Crash), and スクリーンショット (Screenshot). Under Internal Testing (内部テスト), there is a row for PlacePLATEAUTest with a green location pin icon, 0 installs, and a status of "提出準備完了" (Prepared for Submission) with a note "期限切れまで91日" (Valid until 91 days from now). External Testing (外部テスト) and General Information (一般情報) sections are also visible.

ここから、App Storeページで、スクリーンショットやアプリの説明文などのアプリ情報や各種設定を行うことで、審査に提出できるようになります。

最後にAppleの審査が承認されたらApp Storeにリリースします。

機能だけでなく、App Store Reviewガイドラインや、関連するAppleのガイドラインに沿うようにアプリにすることも重要です。公式のドキュメントをよく読み、進めてください。