5G 時代の可視化技術研究 WG
成果報告書
(WG 活動期間：2020 年 11 月～2022 年 11 月)

2022 年　11 月 30 日
サイエンティフィック・システム研究会
5G 時代の可視化技術研究 WG

- 商標について
  記載されている製品名などの固有名詞は、各社/各機関の商標または登録商標です。
- 著作権について
  著作権は各原稿の著者または所属機関に帰属します。無断転載を禁じます。
- 会社名、商品名等について
  会社名、商品名、商標、URL 等は発行時(2022 年 11 月)の情報です。

5G 時代の可視化技術研究 WG　成果報告書　目次

はじめに

1.1. 本 WG と当報告書について

シミュレーションや計測で得られる数値データを対象とした画像化技術としての可視化は、その研究成果が多くの学術分野に広く浸透した。さらに、情報の描画とともに、データ分析・対話操作・視覚認知・協調的分析・多感覚型情報伝達などと組み合わせ、可視化技術を中心に据えた総合的な視覚的情報分析手法の研究へと発展している。しかし、各データで求められる効果的な可視化法を開発するためには、ヘッドマウントディスプレイなどを介した対話操作を伴う高精細 3D 画像の高速描画・伝送への対応など、骨の折れる困難な作業が必要である。この困難さを解決する方法の一つとしてゲーム開発エンジンに着目し、また、入力データや可視化したデータを高速に通信するため 5G に着目した。

ゲーム開発エンジンやヘッドマウントディスプレイなど、ゲーム業界で先行するコンピュータグラフィック及びレンダリング技術は幅広く科学技術・教育分野に活用できる可能性を秘めている。2018 年 3 月から 2020 年 3 月まで活動した汎用 VR システムの活用研究 WG では、科学・工学や生物学、医療、防災を含む人間社会科学など、様々な科学技術分野や教育現場などで用いられている先進的な入力/出力デバイスやプログラム開発基盤を活用するための調査・研究を行った。その結果、データ可視化の開発技術の現状、ゲーム開発エンジン Unity を用いた標準的なワークフローやデータのインポート・マッピング法、現状のデバイスの紹介、さらに新しい可視化手法や活用方法、コンテンツ開発の際の具体的な課題及びその解決方法の例、WG 参加研究者による可視化事例を報告書としてまとめた。

先の WG で行った Unity の調査・研究の成果を受け、本 WG「5G 時代の可視化技術研究 WG」では、Unity を用いた具体的なアプリケーションの開発・作成・公開を進め、さらには可視化研究に活用するための 5G に関する技術調査および検討を行った。

アプリケーション開発に Unity を用いたのは、先の WG での調査・研究とともに、5G がスマートフォン等で利用されることを踏まえ、Androidに対するアプリケーション開発もUnityは可能であるからだ。その一方、Unity に関する書籍やインターネット上の情報は当然ながらそのほとんどがゲーム開発に関するものであり、それ以外の分野に適用するための情報は不十分である。本アプリケーション開発を通じて得られた、具体的なデータの読み込み方法など、ゲーム分野以外で実際の開発に必要となるノウハウ（特に、解説書では見過ごされがちな Tip や失敗談など）を本報告書にまとめた。

5G のような高速ネットワークは可視化の活用環境を大きく変えると考えられる。これまでにもレンダリングと表示をリモートとローカルに分けて可視化を行うアプリケーションが開発されてきたが、リモートとローカルを接続するネットワークがボトルネックとなり、実用に耐えるものではなかった。しかし、5G では低いレイテンシー、高速で大容量のデータ転送、多地点での接続が可能とされ、これまでのボトルネックが解消されることが期待される。高速ネットワークを可視化技術に応用するため、5G と既存のネットワーク技術との比較や実際に 5G を活用する環境での実アプリケーションの実行など、調査・研究を行った。

シミュレーションなどのデータ解析に応用するためのアプリケーション開発と、5G を基盤として活用した応用事例の調査研究を通じて、高速ネットワークインフラとしての 5G に関する技術情報、および、活用ノウハウと課題など、俯瞰的な情報をまとめることができた。このように、可視化用のデータ処理から Unity を使ったレンダリング、そして、可視化情報をヘッドマウントディスプレイへ送信するという、可視化の一連のプロセスにおける 5G の活用に関する本報告書が、国や分野を超えた可視化技術研究に関する検討の一助となることを期待する。

1.2. 活動概要

1.1.1. 活動期間

2020 年 11 月～2022 年 11 月

1.2.1. WG メンバー

			氏名	所属機関
会員	担当幹事		井口　寧	北陸先端科学技術大学院大学
	推進委員	(まとめ役)	大谷　寛明	核融合科学研究所
			秋永　和計	NTT ドコモ
			河村　拓馬	日本原子力研究開発機構
			坂本　尚久	神戸大学
			宮地　英雄	東京都市大学
			安福　健祐	大阪大学
会員外			川原　慎太郎	海洋研究開発機構
賛助会員 (富士通)	推進委員	(まとめ役)	小笠 温滋	富士通
			荒川　拓也	富士通
			大日向　大地	富士通
			椎崎　耕太郎	富士通
			服部　文子	富士通

1.2.3 活動実績

- 第 1 回会合：2020 年 12 月 11 日
  - WG 活動の方向性/活動スケジュールの検討
- 第 2 回会合：2021 年 2 月 15 日
  - 委員による Unity アプリケーションの紹介
  - 委員による 5G についての紹介
- 第 3 回会合：2021 年 4 月 23 日
  - 委員による 5G についての紹介
  - 委員が抱える通信における課題及び 5G への期待の整理
- 第 4 回会合：2021 年 6 月 24 日
  - ゲストによるローカル 5G についての紹介
  - 委員による 5G 利用シーン紹介と課題の整理
- 第 5 回会合：2021 年 9 月 30 日
  - 委員及びゲストによる 5G ユースケースの紹介
- 第 6 回会合：2021 年 12 月 3 日
  - 成果物の目次の議論
- 第 7 回会合：2022 年 1 月 25 日
  - 委員による 5G 利用環境の紹介及びデモンストレーション
  - 成果物の目次の議論
- 第 8 回会合：2022 年 3 月 10 日
  - ゲストによるデジタルツインの紹介
  - 委員による 5G ユースケースの紹介
  - 成果物の内容の議論
- 第 9 回会合：2022 年 5 月 11 日
  - ゲストによる 5G ユースケースの紹介
  - 成果物の内容の議論
- 第 10 回会合：2022 年 7 月 22 日
  - 成果物の内容の議論
- 第 11 回会合：2022 年 9 月 8 日
  - 成果物の内容の議論
- 第 12 回会合：2022 年 10 月 28 日
  - 成果物の内容の議論

1. Unity を用いた可視化アプリケーションの開発

海洋研究開発機構　川原慎太郎、東京都市大学　宮地英生

Unity はゲームエンジンであり、当然ながらその主目的はゲーム開発である。世に溢れる Unity を使った開発についての情報も、その多くがゲームまたはそれに類する事柄に関するものである。その Unity を「データ可視化に用いる」と簡単に言っても、その具体的な方法やノウハウについては広く公開されておらず、興味はあるもののどこから手を付けてよいのかわからないという方も多いだろう。そこで本章前半では Unity をデータ可視化に用いるための基礎知識に絞って、ファイルから頂点座標データを読み込み、それらから構成される三角形ポリゴンを描画するまでを順を追って紹介する。ここで紹介する情報があれば、データ読み込み、描画の仕組みやその内部処理を経た画面表示までの大まかな流れがわかるだろう。本章後半では、Unity 用可視化フレームワークとして川原と宮地が開発を進めている「VisAssets」[1]について紹介する。前半で紹介する Unity での描画手順を理解できれば、VisAssets の動作原理については理解し易いものと考える。紙面の都合上、要点に絞った説明となるため、Unity の利用に必要なライセンスの取得やインストールの詳細、Unity のスクリプト言語である C#の文法の説明などについては本稿では割愛させて頂く。

2.1　Unity エディタのインストールとバージョン選定

現在、Unity のインストールは「Unity Hub」と呼ばれるアプリケーションを介して行うのが一般的である。Unity Hub を用いることで、異なるバージョンの Unity (Unity エディタ)を複数共存させる形でのインストールや、それぞれに対するモジュール(開発環境とは異なるプラットフォーム向けのビルド環境など)の追加や削除といった操作を一元管理することができる。図 2.1-1 に Unity Hub の実行画面を示す。

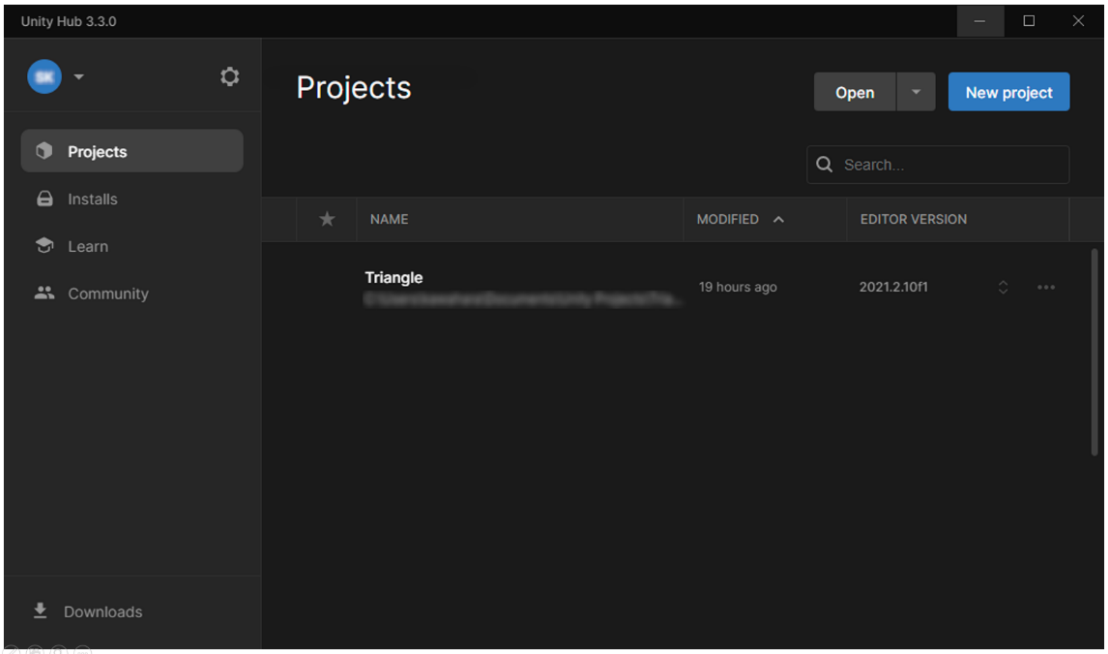


図 2.1-1　Unity Hub

新規プロジェクトの作成や、以降の管理についても Unity Hub 上から行うため、Unity を使用する際には毎回まず始めに Unity Hub を起動する形となる。Unity Hub を初めて起動した後にまず問題となるのは「どのバージョンの Unity エディタをインストールすべきか」である。Unity Hub からインストール可能な Unity エディタの中で、基本的には最新の LTS (Long Term Support、長期サポート)バージョンを選択することが推奨されている。図 2.1-2 は、インストールする Unity エディタのバージョンを選択する際の Unity Hub の画面であるが、本図では「LONG TERM SUPPORTS (LTS)」カテゴリに「2020.3」と「2021.3」の二つが表示されており、それぞれのバージョン番号に続いて「LTS」との表記がされている。また、バージョン番号の新しい「2021.3」には推奨バージョン(Recommended version)である旨も付記されている。この場合、特に理由が無ければ「2021.3」をインストールするのがよいだろう。

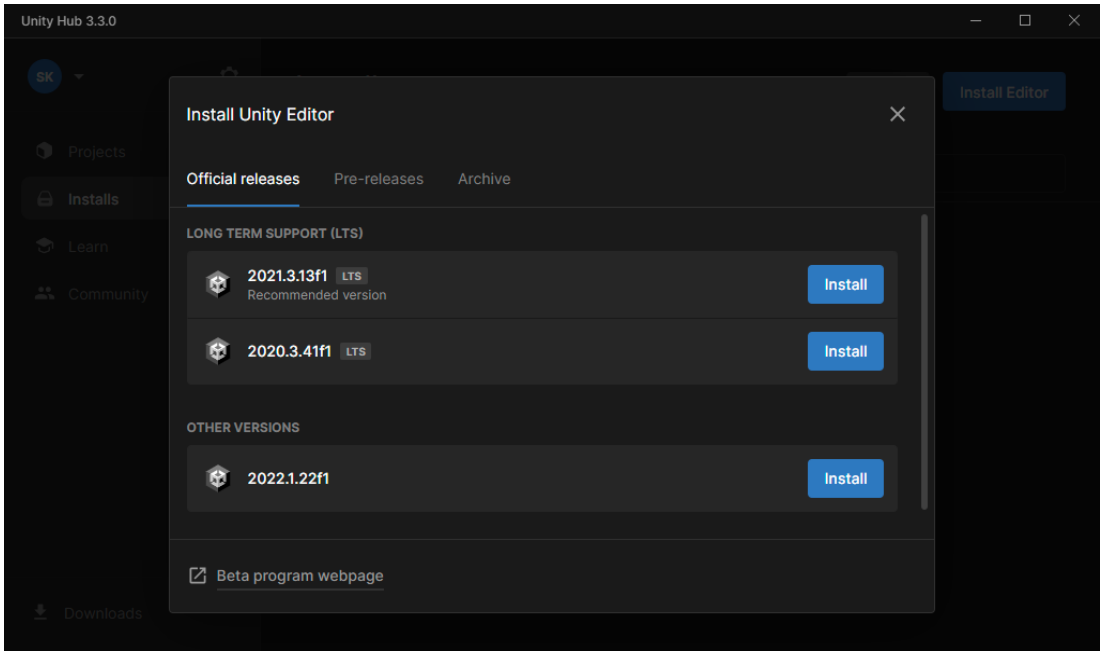


図 2.1-2　Unity Hub でのインストール選択画面

ただし、以下のような場合には他のバージョンがインストールの選択肢として加わる。

・LTS バージョン以降に実装された新しい機能を利用したい
・使用するアセットが特定のバージョンでの動作を対象としている

前者の場合、LTS バージョンよりも新しいバージョン番号の Unity エディタのインストールが必要となる。図 2.1-2 において「OTHER VERSIONS」カテゴリにある「2022.1」や、本図中には表示されていないが Pre-releases タブから選択可能な「2022.2」や「2023.1」が該当する。これらは将来的に LTS バージョンとなる、サポートサイクルの短いテストベッド的なバージョンのため、新機能を検証する目的等で用いることが多い。

後者については使用するアセットに付帯する問題である。例えば、本章で目的とする可視化アプリケーションの開発において、その描画結果を立体映像として提示する目的でヘッドマウントディスプレイ(HMD)を用いる場合を考える。この時、多種多様な XR 機器への対応や空間インタラクション、UI の実装などについては既存のアセットを利用するのが便利かつ有効であるが、クロスプラットフォームの Mixed Reality アプリケーションを開発するための Unity 用ツールキットである MRTK (Microsoft Mixed Reality Toolkit for Unity) [2]を用いる場合、その動作には本稿執筆時の最新バージョン(次期バージョンとなる MRTK3 の最新ブランチ)で「2020.3」または「2021.3」、またはそれ以降のバージョンが必要となる。それ以外のバージョンでの動作は基本的に保証されてはおらず、また非推奨のバージョンでの動作に不具合が発生した場合、ユーザ自身が MRTK3 を改変して対応することは現実的ではないため、この場合は動作対象としてバージョン番号が明示されている 2020.3、2021.3 のいずれかを選択するのが無難だろう。

以下、本章で例示する C#スクリプトでは特定のバージョンの Unity エディタに大きく依存するようなコードは使用していないが、それらの動作確認や、本章後半で紹介する可視化フレームワーク「VisAssets」の開発に筆者が現在使用している 2021.2 での実行を前提として以降の説明を進める。

2.2　Unity エディタ

Unity Hub から新規プロジェクトを作成、または既存のプロジェクトを選択することにより Unity のビジュアルエディタである「Unity エディタ」が起動する。新規プロジェクト作成直後の Unity エディタの実行画面を図 2.2-1 に示す。

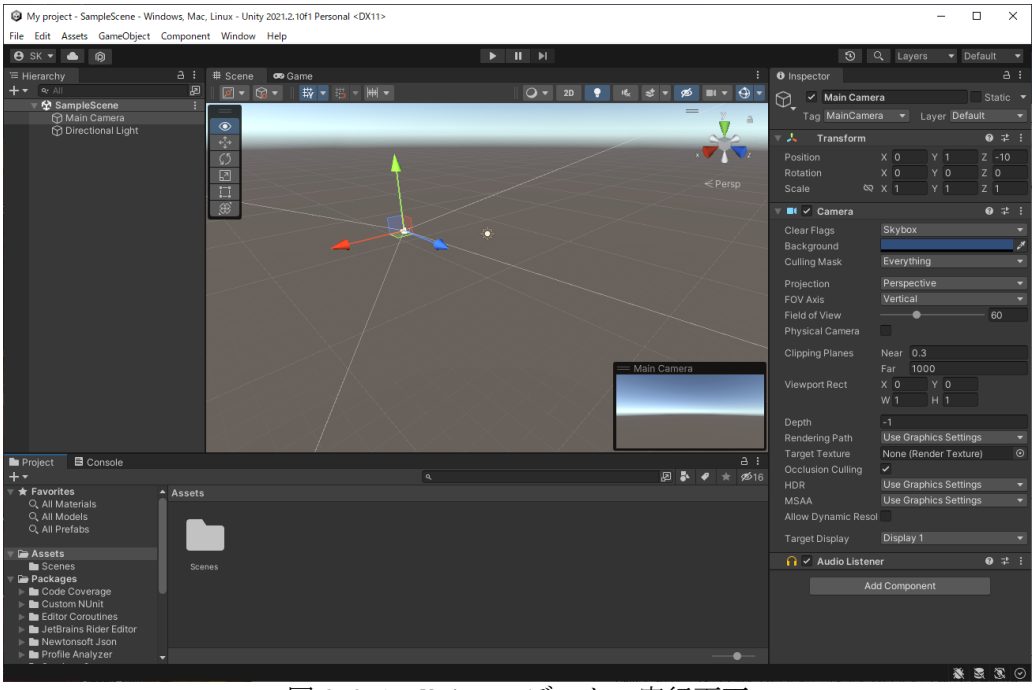


図 2.2-1　Unity エディタの実行画面

図 2.2-2 は、図 2.2-1 に対して Unity エディタの各部を機能別に示したものである。①のヒエラルキーウィンドウには、現在のシーン内に存在する「ゲームオブジェクト」のリストが階層構造で表示される。ゲームオブジェクトについては次節で説明する。②のインスペクタウィンドウには、現在選択中のゲームオブジェクトの詳細情報が表示され、設定値の変更等ができる。③のシーンビューウィンドウには、現在のシーン内に存在するゲームオブジェクトの位置や大きさが表示される。上部のタブ操作により、プレビュー実行時の描画結果を表示するためのゲームビューウィンドウに切り替えることができる。④のプロジェクトウィンドウには、現在のプロジェクトに登録されているアセット(プロジェクト内で利用可能な画像、音声、プログラムなどの素材群)が表示される。上部のタブ操作により、プレビュー実行時のエラーやデバック出力等を表示するためのコンソールウィンドウに切り替えることができる。

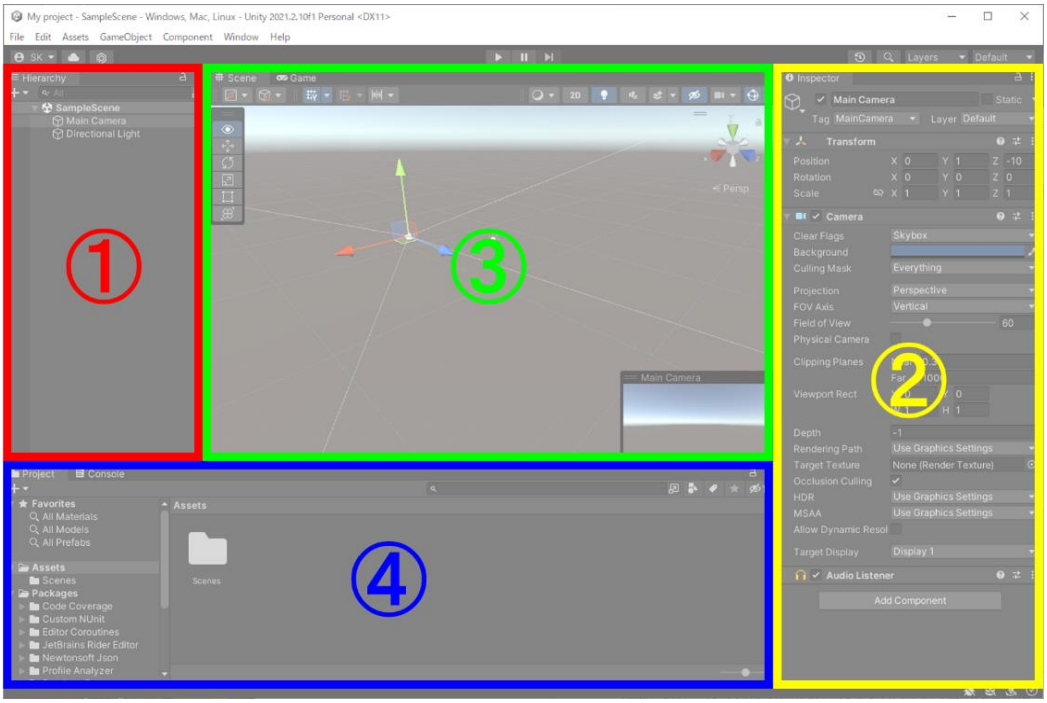
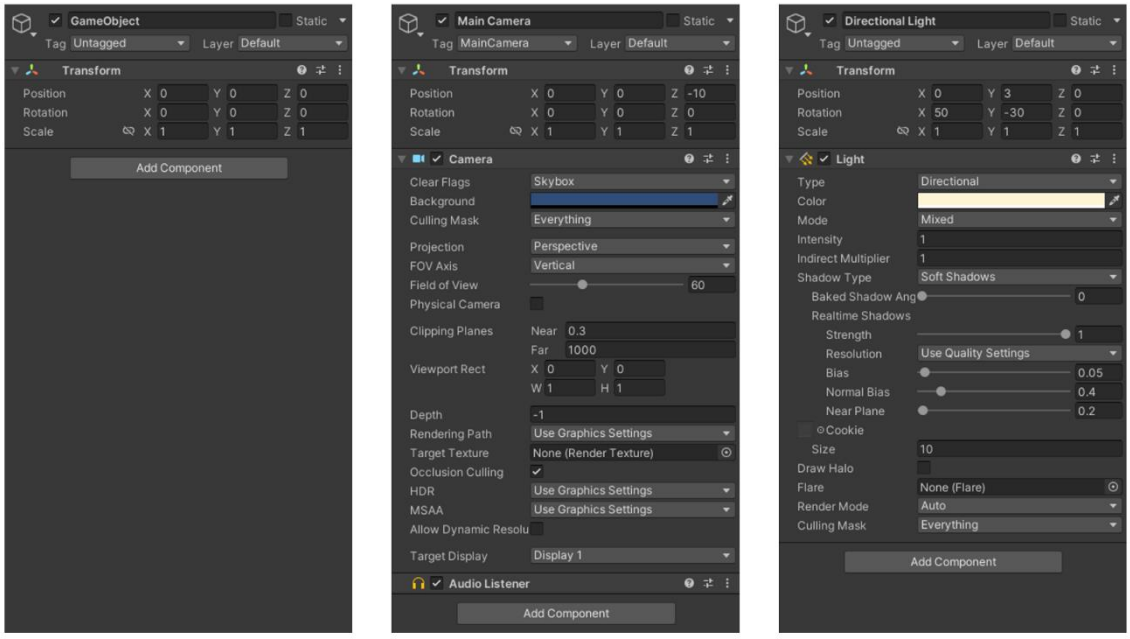


図 2.2-2　Unity エディタの各部

2.3　ゲームオブジェクト

Unity において、シーン内のオブジェクトは「ゲームオブジェクト」と呼ばれる単位で管理される。新規シーンの作成時、シーン内に自動的に生成・配置されるカメラ(Main Camera)や光源(Directional Light)もそれぞれが独立したゲームオブジェクトであり、シーン内での三次元位置情報(位置、回転、スケール)のみを保持する「空のゲームオブジェクト」に対してカメラや光源の機能である“Camera”や“Light”を取り付けたものとなる。つまり、空のゲームオブジェクトは単なる入れ物でしかなく、その中に必要な機能を詰めることで具体的な機能を持ったゲームオブジェクトとなるのである。Unity では、“Camera”や“Light”のようにゲームオブジェクトに機能を付加するものを「コンポーネント」と呼び、ゲームオブジェクトへのコンポーネントや他のゲームオブジェクトの取り付けを「アタッチ」と呼ぶ。

空のゲームオブジェクトは、Unity エディタのヒエラルキーウィンドウで右クリックメニューから[Create Empty]を選択する、またはメインメニューの[GameObject]から[Create Empty]を選択することで新たにシーンに追加することができる。追加時のゲームオブジェクトのデフォルト名は「GameObject」であるが、既に同名のゲームオブジェクトがシーン内にある場合は「GameObject (1)」のようにそれらを区別するための番号が自動的に付与される。図 2.3-1 は空のゲームオブジェクト、カメラ、光源の三つについて、インスペクタウィンドウでの詳細表示をそれぞれ示したものである。三つのゲームオブジェクト全てにおいて、シーン内での位置、回転、スケールを保持するコンポーネントである“Transform”がアタッチされており、カメラと光源にはそれぞれの機能を付加するためのコンポーネントである“Camera”および“Light”がアタッチされていることがわかる。アタッチしたコンポーネントは削除することもできるが、“Transform”については全てのゲームオブジェクトの基本情報であるため削除することができない。空の入れ物であっても、または何らかの機能を持っているが見た目上の画面表示には何の影響も与えないようなゲームオブジェクト(例えば、時間経過によって移動するゲームオブジェクトの現在位置の管理など、パラメータのみを保持するゲームオブジェクト)であっても、シーン内のどこかには置かれている必要がある、ということである。



(a) 空のゲームオブジェクト　　(b) カメラ　　(c) 光源

図 2.3-1　ゲームオブジェクトの構成

2.4　ゲームオブジェクトへのユーザ独自の機能の追加

前節で紹介した“Camera”や“Light”のように、Unity にはゲームオブジェクトにさまざまな機能を付加するためのコンポーネントが用意されているが、任意の機能を実装したプログラムファイルをゲームオブジェクトにアタッチすることで、ユーザ独自の機能を持ったゲームオブジェクトを作成することができる。例えば、「ユーザ独自形式のデータを読み込む」という機能を実装したプログラムを作成す

ることにより、Unity 標準のコンポーネントでは対応していない形式のデータであっても、そのデータを Unity に読み込むことができるゲームオブジェクトを新たに作成することもできる。本稿執筆時のバージョンの Unity でゲームオブジェクトにアタッチすることができるプログラムファイルは、C#言語で記述されたもの(C#スクリプト)のみとなる。

現在のプロジェクトに新たな C#スクリプトを追加するには、プロジェクトウィンドウ上での右クリックメニューの[Create]から[C# Script]を選択、またはメインメニューの[Assets]-[Create]から[C# Script]を選択する。既存の C#スクリプトファイルがある場合、それをプロジェクトウィンドウにドラッグアンドドロップすることで、アセットとしてプロジェクトに追加することもできる。図 2.4-1 は Unity エディタ上での操作により新規作成した直後の C#スクリプトの内容を示したものである。

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class NewBehaviourScript : MonoBehaviour
{
    // Start is called before the first frame update
    void Start()
    {
    }

    // Update is called once per frame
    void Update()
    {
    }
}
```

図 2.4-1　新規作成直後の C#スクリプト (NewBehaviourScript.cs)

本スクリプト内に自動生成されるクラスは、Unity の基本クラスである MonoBehaviour を継承したサブクラスとなる。この時、クラス名 (NewBehaviourScript) については新規作成時の C#スクリプト名 (この例ではデフォルト名である NewBehaviourScript.cs) から拡張子を除いたものが使用される。クラス名は後から変更することも可能であるが、その際には C#スクリプトのファイル名も同じものに変更する必要がある。両者が一致していない場合はエラーとなるので注意が必要である。また、データ保持のみを目的としたクラスなど、必ずしも MonoBehaviour を継承したサブクラスでなくとも Unity では利用可能であるが、コンポーネントとしてゲームオブジェクトにアタッチすることができるのは MonoBehaviour を継承したサブクラスが記載された C#スクリプトのみであることにも注意が必要である。

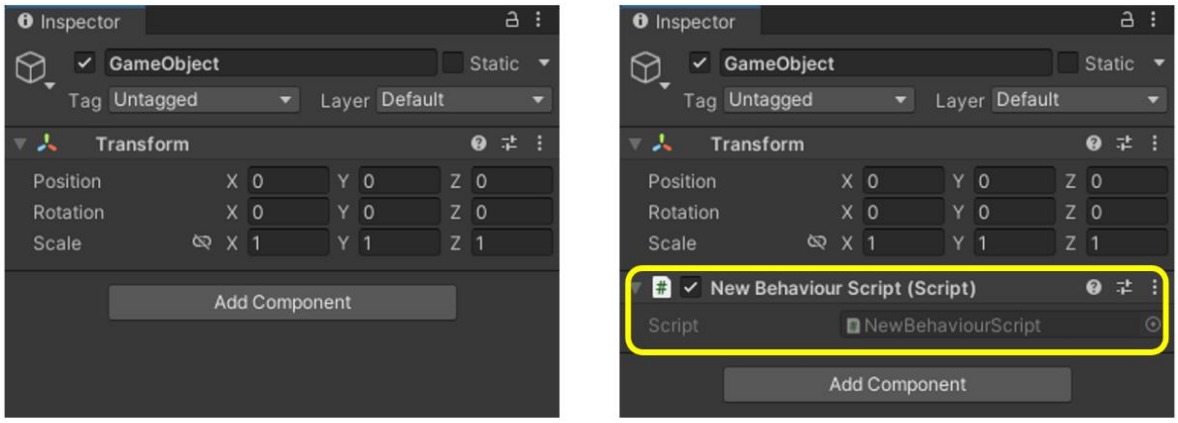
2.5　ゲームオブジェクトへの C#スクリプトのアタッチ

ゲームオブジェクトへの C#スクリプトのアタッチは下記のいずれかの方法で行う。

- プロジェクトウィンドウ内の C#スクリプトのアイコンを、ヒエラルキーウィンドウ上の対象となるゲームオブジェクトにドラッグアンドドロップする
- ヒエラルキーウィンドウ上で対象となるゲームオブジェクトを選択し、プロジェクトウィンドウ内の C#スクリプトのアイコンを、インスペクタウィンドウにドラッグアンドドロップする
- ヒエラルキーウィンドウ上で対象となるゲームオブジェクトを選択し、インスペクタウィンドウ内の「Add Component」ボタンから C#スクリプトを選択する

前節で作成した MonoBehaviour.cs を空のゲームオブジェクトにアタッチした際のインスペクタの表示の変化を図 2.5-1 に示す。C#スクリプトをアタッチする前の図 2.5-1(b)では位置、回転、スケールを保持するコンポーネントである「Transform」のみがアタッチされているが、アタッチ後の図 2.5-1(b)では「New Behaviour Script (Script)」という名称のコンポーネントが追加されていることがわかる。アタッチした C#スクリプト内でのクラス名は「NewBehaviourScript」であるが、インスペクタでは元のクラス名の大文字部を区切りとしてスペースが挿入され、また C#スクリプトであることが末尾に付記された「New Behaviour Script (Script)」のように表示される。本稿では割愛するが、他の C#スクリプトから本 C#スクリプトを参照する場合は、インスペクタでの表記である「New Behaviour Script (Script)」

ではなく、正式なクラス名である「NewBehaviourScript」が用いられる。



(a) アタッチ前　　(b) アタッチ後

図 2.5-1　C#スクリプトのアタッチ前後でのインスペクタの表示の変化

2.6　Unity における C#スクリプトの動作原理

Unity エディタにはプレビュー実行機能があり、特定のプラットフォーム向けの実行バイナリを生成することなく、構築したアプリケーションの動作確認をすることができる。Unity エディタ上部中央にある再生ボタンでプレビュー実行を開始し、もう一度同じボタンを押すことで実行を終了する。前節で作成した C#スクリプトにはまだ具体的な処理内容を記述していないため、この時点ではプレビュー実行しても何もイベントは発生せず、プレビュー画面にも何の変化も起きない。

図 2.4-1 で示した新規作成直後の C#スクリプトにおいて、メンバ関数として自動生成された Start()および Update()は継承元である MonoBehaviour クラスからのオーバーライド関数である。Start()内にはアプリケーション実行後の最初のフレーム描画前に一度だけ実行する処理を、Update()内には毎フレーム実行する処理をそれぞれ記載する。例えば、図 2.4-1 の C#スクリプトに対して図 2.6-1 の網掛部の変更を加えてプレビュー実行すると、コンソールウィンドウには初めに一度だけ文字列「Start()」が表示され、以降は文字列「Update()」が繰り返し表示される（図 2.6-2）。ここで使用した Debug.Log()は、Unity エディタでのプレビュー実行時にコンソールウィンドウに指定した文字列を表示する Unity の関数であり、プリントデバッグ時に有用な関数である。

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class NewBehaviourScript : MonoBehaviour
{
    // Start is called before the first frame update
    void Start()
    {
        Debug.Log("Start()");
    }

    // Update is called once per frame
    void Update()
    {
        Debug.Log("Update()");
    }
}
```

図 2.6-1　オーバーライド関数の動作確認（NewBehaviourScript.cs）

図 2.6-2　コンソールウィンドウでのオーバーライド関数の動作確認

2.7　三角形ポリゴンの描画

ここからは、C#スクリプトを用いて図 2.7-1 に示すような三角形ポリゴンを描画する例を示す。本図中のカメラは、新規シーンにおける Main Camera の初期位置であり、原点を向く形で配置されている。Unity の座標系は左手座標系であるため、右手座標系である OpenGL を使った可視化アプリケーションの開発者は特に注意が必要である。三角形ポリゴンを構成する頂点インデックスの指定順についても、OpenGL の場合は反時計回りの順で指定した面が表側となるが、Unity では時計回りの順で指定した面が表側となる。このため、既存の右手座標系プログラムからルーチンを移植する際には注意が必要である。

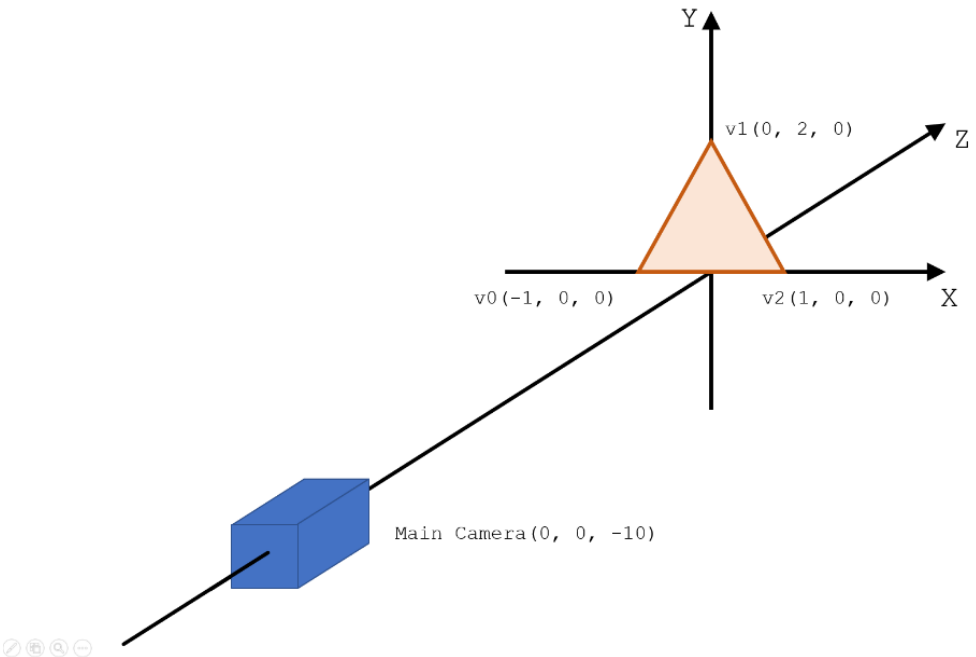


図 2.7-1　Unity における座標系、および描画する三角形とカメラの位置関係

図 2.7-1 の三角形ポリゴンを構成する頂点座標およびインデックス情報について、それらの登録部を追加した C#スクリプトが図 2.7-2 である(前節で追加した、オーバーライド関数の動作確認のためのデバッグ出力については不要なため削除した)。頂点座標およびインデックス情報については固定値としており、アプリケーションの実行直後に一度だけメモリにロードすればよいため、それらの処理を Start()関数内に記載している。本例では頂点数分の Vector3 型の頂点座標および int 型のインデックス情報を格納するのに可変長の List クラスを用いているが、予め使用する頂点数が決まっているのであれば固定長の配列を用いてもよい。等値面のように、閾値によって生成されるポリゴン数が変化する場合は List クラスを使うのがよいだろう。本スクリプトは三角形ポリゴンを描画するものであるため、その機能に合わせてクラス名を Triangle に変更するとともに、C#スクリプトのファイル名を Triangle.cs にそれぞれ変更している。ここまでの変更では三角形ポリゴンの描画に必要な情報を各変数に登録しただけであるため、本スクリプトをゲームオブジェクトにアタッチしてプレビュー実行しても三角形ポリゴンは描画されない。ジオメトリ情報に基づき画面に描画するためには、更に“MeshFilter”と

“MeshRenderer”の二つのコンポーネントが必要となる。まず、画面描画のためのジオメトリ情報を扱う Mesh クラスに対して頂点座標およびインデックス情報を登録する。次に、その Mesh クラスを MeshFilter コンポーネントに登録する。MeshFilter コンポーネントに登録されたジオメトリ情報は MeshRenderer コンポーネントを介して画面に描画される。これらの一連の処理の概念図を図 2.7-3 に示す。

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Triangle : MonoBehaviour
{
    void Start()
    {
        // vertices
        var vertices = new List<Vector3>();
        var v0 = new Vector3(-1, 0, 0);
        var v1 = new Vector3( 0, 2, 0);
        var v2 = new Vector3( 1, 0, 0);
        vertices.Add(v0);
        vertices.Add(v1);
        vertices.Add(v2);

        // indices
        var indices = new List<int>();
        indices.Add(0);
        indices.Add(1);
        indices.Add(2);
    }

    void Update()
    {
    }
}
```

図 2.7-2　三角形ポリゴンを構成するジオメトリ情報の登録（Triangle.cs）

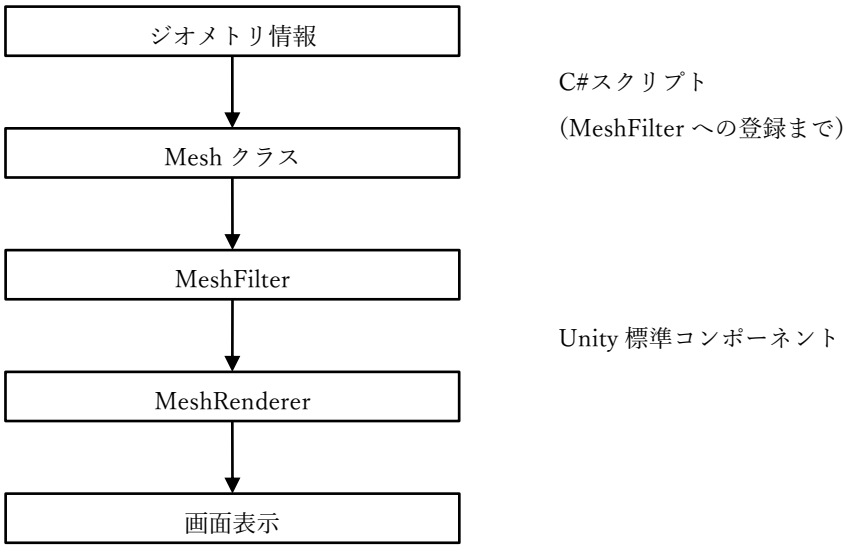


図 2.7-3　ジオメトリ情報に基づく描画処理の概念図

図 2.7-4 は、図 2.7-2 の C#スクリプトに対してジオメトリ情報の Mesh クラスへの登録処理と MeshFilter コンポーネントへの Mesh クラスの登録処理を追加したものである(Triangle2.cs)。両処理ともにアプリケーション実行後の最初のフレーム描画前に一度だけ実行すればよく、以降の更新は無い

ためジオメトリ情報の登録と同じく Start()関数内に記述している。

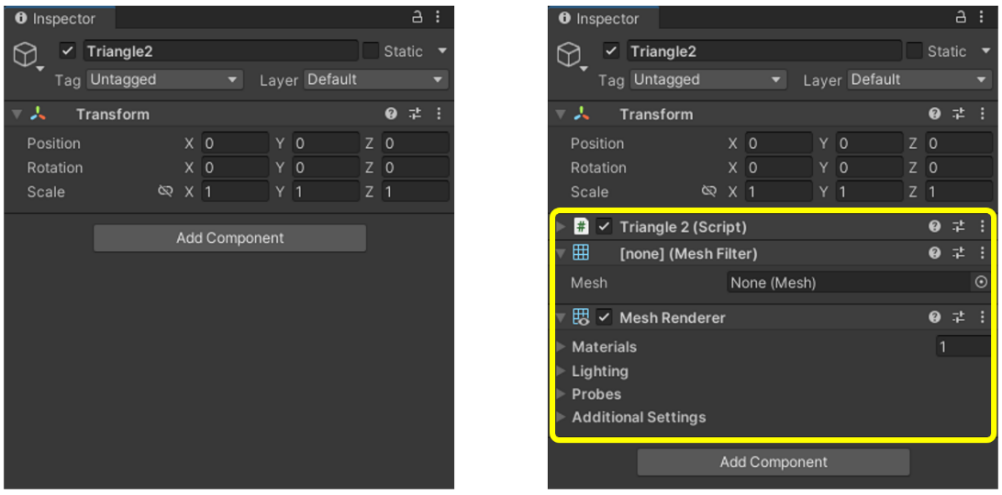
```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Triangle2 : MonoBehaviour
{
    void Start()
    {
        // vertices
        // 変更無し

        // indices
        // 変更無し

        // set vertices and indices to Mesh class
        var mesh = new Mesh();
        mesh.vertices = vertices.ToArray();
        mesh.SetIndices(indices, MeshTopology.Triangles, 0);
        mesh.RecalculateNormals();

        // set Mesh class data to MeshFilter component
        var filter = GetComponent<MeshFilter>();
        if (filter != null) filter.mesh = mesh;
    }

    void Update()
    {
    }
}
```

図 2.7-4　ジオメトリ情報の MeshFilter への登録 (Triangle2.cs)

本変更により、各 List クラスに格納された頂点座標データおよびインデックス情報はまず Mesh クラスに登録される。Mesh.vertices は Mesh クラスに頂点座標データを登録するためのメンバ変数であるが、これは Vector3 型の配列であるため List.ToArray()関数を使ってリスト型から配列に変換したものを渡している。また、Mesh.SetIndices()は Mesh クラスにインデックス情報を登録するためのメンバ関数であるが、MeshTopology.Triangles を指定することで Mesh.vertices に登録された頂点座標データからインデックス順に三角形ポリゴンを構成する。MeshTopology にはこの他に Quads, Lines, LineStrip, Point があり、それぞれ四角形ポリゴン、線分、閉じた線分、点を描画することができる。法線ベクトルについては Mesh.RecalculateNormals()を使って算出したものを使用する。

Mesh クラスに格納したジオメトリ情報を MeshFilter に登録するためには、ゲームオブジェクトにアタッチされた MeshFilter コンポーネントへのアクセスが必要となる。C#スクリプトから他のコンポーネントにアクセスするためには GetComponent()関数を用いる。本稿ではその方法については割愛するが、同じゲームオブジェクトのコンポーネントだけでなく、他のゲームオブジェクトのコンポーネントにアクセスすることも可能である。取得しようとしたコンポーネントが対象のゲームオブジェクトにアタッチされていなかった場合を考慮し、本例のように GetComponent()関数の戻り値を使ったエラー処理を行うのがよいだろう。MeshFilter から MeshRenderer にデータを渡すためのコードは特に記述されていないが、MeshRenderer は同じゲームオブジェクトにアタッチされた MeshFilter に登録されたジオメトリ情報を自動的に描画対象とするためである。

MeshFilter および MeshRenderer のゲームオブジェクトへのアタッチは、インスペクタ下部にある「Add Component」ボタンから、またはメインメニューの[Component]-[Mesh]-[MeshFilter]/[MeshRenderer]から行う。C#スクリプト(Triangle2.cs)と両コンポーネントの追加前後でのインスペクタの表示の変化を図 2.7-5 に示す。図 2.7-6 はここまでのプレビュー実行の結果である。現時点ではピンク一色で塗り潰されているものの、画面中央に三角形ポリゴンが描画されていることがわかる。



(a) アタッチ前　　(b) アタッチ後

図 2.7-5　コンポーネントの追加によるインスペクタの表示の変化

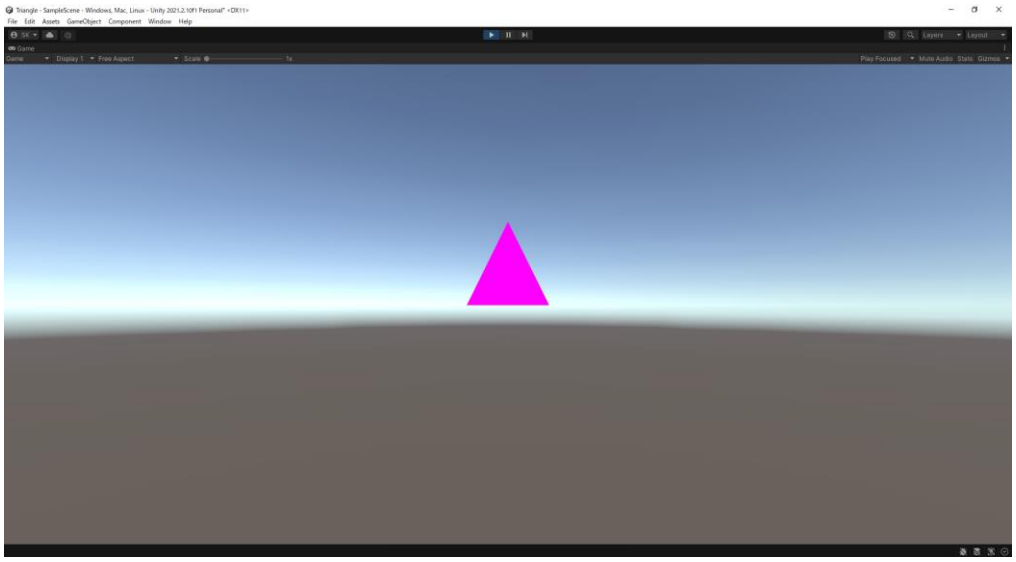


図 2.7-6　ここまでの実行結果

2.8　シェーダによる頂点色の描画

前節の実行結果では、描画された三角形の色はピンクで塗り潰されていた。これは、ポリゴンを構成する各頂点に色情報が設定されていないだけでなく、描画のための適切な「シェーダ」が指定されていないためである。まず、各頂点の色情報を設定するために、C#スクリプトに図 2.8-1 の網掛部分の変更を加える。本変更により、頂点座標やインデックス情報と同様に各頂点に対応する色情報が Mesh クラスに登録されるが、この時点ではまだ適切なシェーダが設定されていないため、プレビュー実行してもその結果は図 2.7-6 と同じものとなる。

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Triangle3 : MonoBehaviour
{
    void Start()
    {
        // vertices
        // 変更無し

        // colors
```

```
        var colors = new List<Color>();
        var c0 = new Color(1, 0, 0);
        var c1 = new Color(0, 1, 0);
        var c2 = new Color(0, 0, 1);
        colors.Add(c0);
        colors.Add(c1);
        colors.Add(c2);

        // indices
        // 変更無し

        var mesh = new Mesh();
        mesh.vertices = vertices.ToArray();
        mesh.colors = colors.ToArray();
        mesh.SetIndices(indices, MeshTopology.Triangles, 0);
        mesh.RecalculateNormals();

        var filter = GetComponent<MeshFilter>();
        if (filter != null) filter.mesh = mesh;
    }

    void Update()
    {
    }
}
```

図 2.8-1　各頂点への色情報の追加（Triangle3.cs）

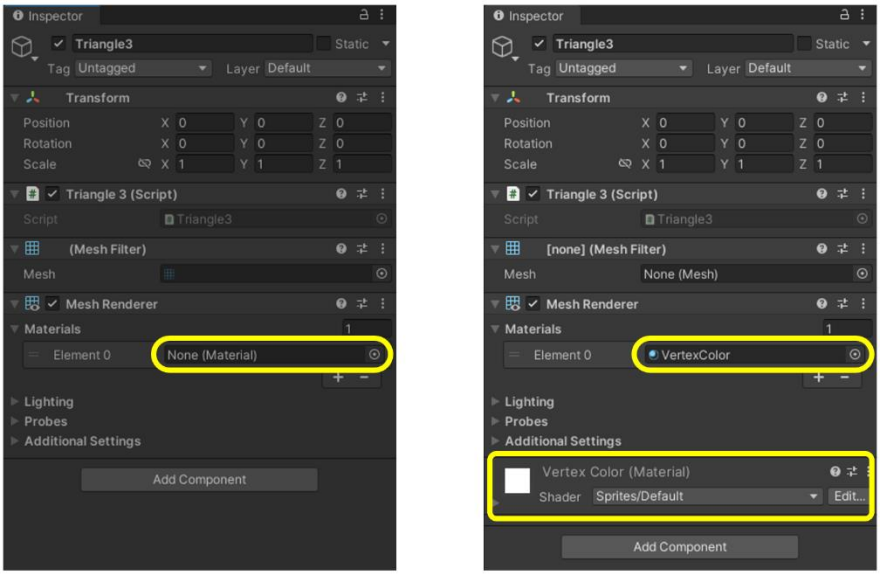
描画時にシェーダを適用するためには、MeshRenderer にシェーダを渡すための「マテリアル」を作成する必要がある。プロジェクトウィンドウ上での右クリックメニューの[Create]から[Material]を選択、またはメインメニューの[Assets]-[Create]から[Material]を選択することでマテリアルファイルが新規作成される。新規作成時のマテリアルファイルのデフォルト名は「New Material」であるが、ここでは「VertexColor」に変更している。

ゲームオブジェクトへのマテリアルのアタッチは、C#スクリプトのアタッチと同様に下記のいずれかの方法で行う。

- プロジェクトウィンドウ内のマテリアルのアイコンを、ヒエラルキーウィンドウ上の対象となるゲームオブジェクトにドラッグアンドドロップする
- ヒエラルキーウィンドウ上で対象となるゲームオブジェクトを選択し、プロジェクトウィンドウ内のマテリアルのアイコンをインスペクタウィンドウにドラッグアンドドロップする
- ヒエラルキーウィンドウ上で対象となるゲームオブジェクトを選択し、インスペクタ下部の「Add Component」ボタンからマテリアルを選択する

この他、インスペクタの MeshRenderer のプロパティ「Materials」からも MeshRenderer が使用するマテリアルを指定することができる。いずれの方法を採る場合でも、対象のゲームオブジェクトに対して MeshRenderer コンポーネントが先にアタッチされていない場合には、マテリアルをアタッチすることはできない。

図 2.8-2 は、マテリアルのアタッチ前後のゲームオブジェクトのインスペクタの表示の変化を示したものである。アタッチ後の図 2.8-2(b)では「VertexColor」がコンポーネントとして追加されている。また、アタッチ前の図 2.8-2(a)では MeshRenderer の Materials プロパティの Element 0 の値が「None」になっているが、アタッチ後には「VertexColor」が設定されている。VertexColor の「Shader」プロパティで「Sprites/Default」を選択してプレビュー実行することで、設定した頂点色が反映された図 2.8-3 の描画結果が得られる。



(a) アタッチ前　　(b) アタッチ後

図 2.8-2　ゲームオブジェクトへのマテリアルのアタッチ

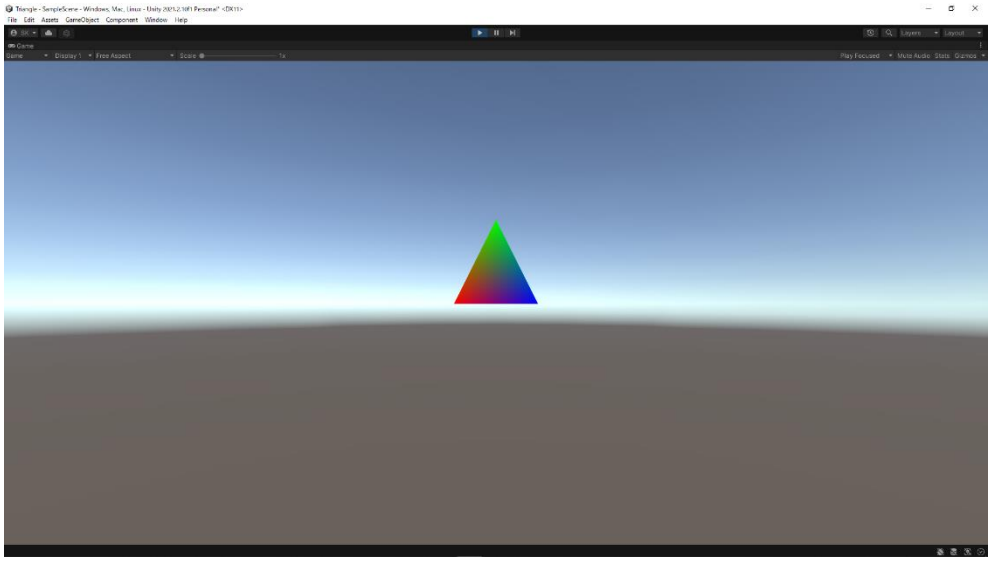


図 2.8-3　ここまでの実行結果

2.9　カスタムシェーダによる頂点色の描画

前節で「VertexColor」マテリアルのシェーダとして適用した「Sprites/Default」は、Unity の組み込みシェーダ(ビルトインシェーダ)である。組み込みシェーダで表現できる描画効果についてはそれらから適切なものを選択すればよいが、そうでない場合はユーザ独自のシェーダを作成し適用する必要がある。ここでは「Sprites/Default」を使わず、これと同様の効果を得るためのカスタムシェーダの作成例を紹介する。

カスタムシェーダを新規作成するには、プロジェクトウィンドウの右クリックメニューの[Create]-[Shader]から、またはメインメニューの[Assets]-[Create]-[Shader]からベースとなるシェーダの種別を選択する。選択可能なシェーダの種別としては、最も基本的な「Standard Surface Shader」、より高度な「Unlit Shader」、ポストエフェクト用の「Image Effect Shader」、GPU を使った処理を実装するための「Compute Shader」などがある。ここでの選択により、新規作成時のテンプレートとして自動生成されるシェーダコードが異なるが、ここでは「Standard Surface Shader」をベースとして用いる。図 2.9-1 は新規作成直後の Standard Surface Shader の内容である。1 行目にある「Custom/NewSurfaceShader」がシェーダ名となる。前節で作成した「VertexColor」マテリアルのシェーダとして、図 2.9-1 の「Custom/NewSurfaceShader」を適用し、プレビュー実行した結果が図 2.9-2 で

ある。

本シェーダは、ベースとなる塗り潰し色とテクスチャ画像などから各ピクセルの描画色を決定する処理内容であるため、この時点ではベース色の初期値(1, 1, 1, 1)で塗り潰された三角形ポリゴンが描画される。三角形ポリゴンの各頂点に対してテクスチャ座標を設定していないため、インスペクタの設定で何らかのテクスチャ画像を指定しても描画結果は同じである。以下、本節ではこのシェーダコードに対して変更を加え、設定した頂点色に基づいた三角形ポリゴンが描画されるようにする。

```
Shader "Custom/NewSurfaceShader"
{
    Properties
    {
        _Color ("Color", Color) = (1,1,1,1)
        _MainTex ("Albedo (RGB)", 2D) = "white" {}
        _Glossiness ("Smoothness", Range(0,1)) = 0.5
        _Metallic ("Metallic", Range(0,1)) = 0.0
    }
    SubShader
    {
        Tags { "RenderType"="Opaque" }
        LOD 200

        CGPROGRAM
// Physically based Standard lighting model,
// and enable shadows on all light types
        #pragma surface surf Standard fullforwardshadows

        // Use shader model 3.0 target, to get nicer looking lighting
        #pragma target 3.0

        sampler2D _MainTex;

        struct Input
        {
            float2 uv_MainTex;
        };

        half _Glossiness;
        half _Metallic;
        fixed4 _Color;

        // Add instancing support for this shader.
// You need to check 'Enable Instancing' on materials that use the shader.
// See https://docs.unity3d.com/Manual/GPUInstancing.html
// for more information about instancing.
// #pragma instancing_options assumeuniformscaling
        UNITY_INSTANCING_BUFFER_START(Props)
            // put more per-instance properties here
        UNITY_INSTANCING_BUFFER_END(Props)

        void surf (Input IN, inout SurfaceOutputStandard o)
        {
            // Albedo comes from a texture tinted by color
            fixed4 c = tex2D (_MainTex, IN.uv_MainTex) * _Color;
            o.Albedo = c.rgb;
            // Metallic and smoothness come from slider variables
            o.Metallic = _Metallic;
            o.Smoothness = _Glossiness;
            o.Alpha = c.a;
        }
        ENDCG
    }
    FallBack "Diffuse"
}
```

図 2.9-1　新規作成したサーフェスシェーダ（NewSurfaceShader.shader）

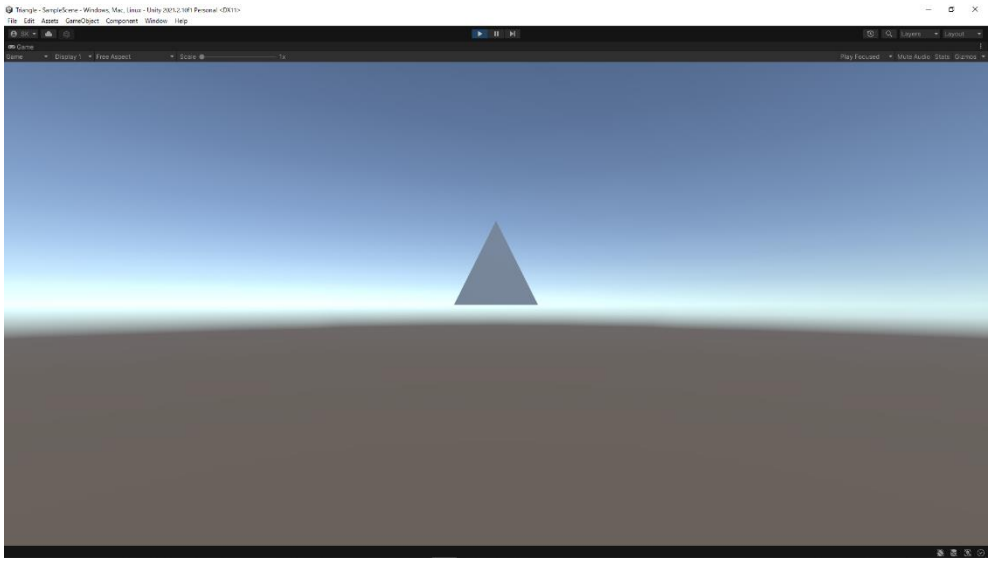


図 2.9-2　ここまでの実行結果

Unity におけるシェーダは Unity 独自の ShaderLab と呼ばれる書式で記述され、その内部で Cg 言語、DirectX 向けの HLSL、OpenGL 向けの GLSL の三種のシェーダ言語を使用することができる。図 2.9-3 は図 2.9-1 のシェーダコードから具体的な処理部分を取り除き、その構成を大まかに示したものである。

```
Shader "Custom/NewSurfaceShader" // シェーダの名称
{
    Properties
    {
        // 外部から参照可能なシェーダプロパティ
    }
    SubShader
    {
        // シェーダ言語での処理内容の記述
        // Cg/HLSL 言語の場合は「CGPROGRAM」から「ENDCG」の間に記述
        // HLSL 言語の場合は「HLSLPROGRAM」から「ENDHLSL」の間に記述
        // GLSL 言語の場合は「GLSLPROGRAM」から「ENDGLSL」の間に記述

        // 図 2.7-1 は 1 パスであるが、SubShader 内に複数のパスを定義する場合には
        // それぞれの処理を Pass{}の中に記述する
        // 例)
        // Pass
        // {
        //       CGPROGRAM
        //         (処理内容)
        //       ENDCG
        // }
    }

    // 互換性のあるシェーダが見つからなかった場合に適用されるシェーダ
    // この場合は"Diffuse"シェーダが適用される
    FallBack "Diffuse"
}
```

図 2.9-3　シェーダの大まかな構成

本図より、“Properties”、“SubShader”、“FallBack”の三つのブロックからシェーダ全体が構成されていることがわかる。Properties ブロックでは C#スクリプトからなど、シェーダ外部から参照可能なプロパティを記述する。ここに記述したプロパティは、図 2.9-4 のようにシェーダを適用したマテリアルのインスペクタにも表示されるようになるため、Unity エディタでの編集時やプレビュー実行中にインスペクタから設定値を変更することができる。例えば、Color の初期値は白(1,　1,　1,　1)になっ

ているが、インスペクタから別の色に変更すると描画される三角形ポリゴンの色も変わる。SubShader ブロックではシェーダの具体的な処理内容を記述する。図 2.9-1 では SubShader 内に「CGPROGRAM」と「ENDCG」で囲まれた箇所があることから、その処理内容が Cg/HLSL 言語で記述されていることがわかる。前述した「ベースとなる塗り潰し色とテクスチャ画像などから各ピクセルの描画色を決定する処理内容」はここに記述されているため、頂点色が描画結果に反映されるよう、この部分に変更を加える。

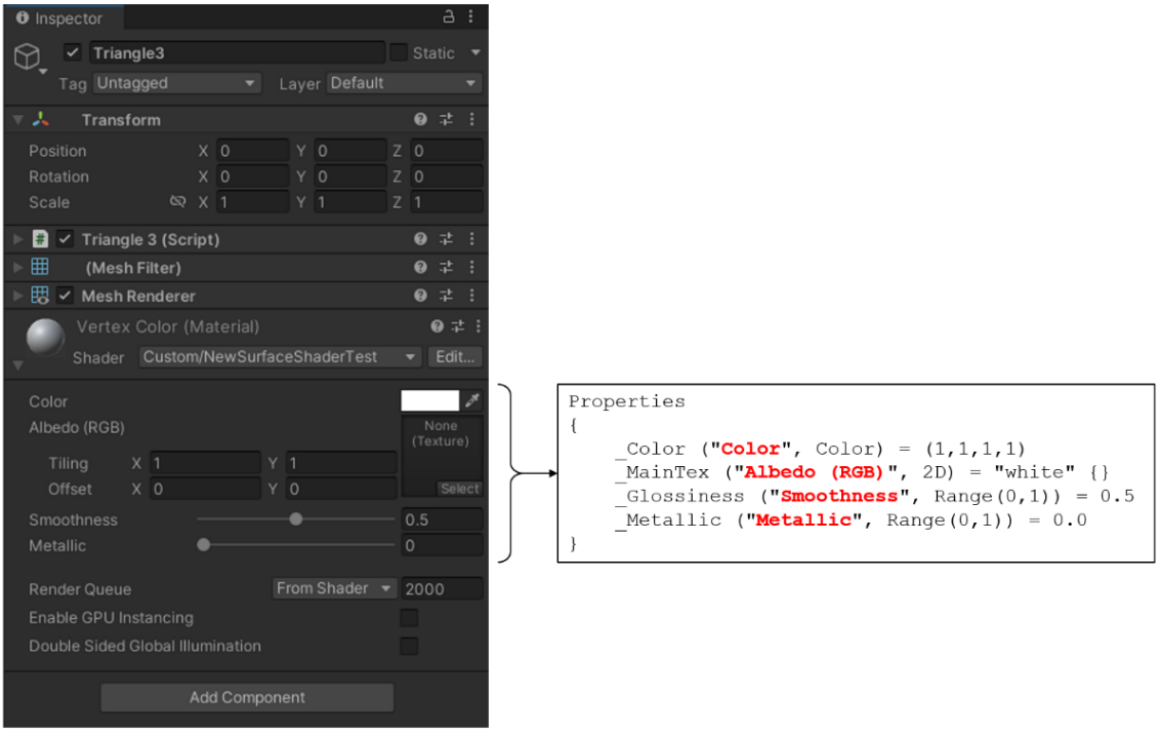


図 2.9-4　インスペクタでのシェーダプロパティの表示

図 2.9-5 は、図 2.9-1 のシェーダに網掛部分の変更を加えたものである。変更部分を最小限とするため、不要なコードの削除等は行っていない。SubShader ブロックに対して、サーフェスシェーダ(surf)への入力となる Input 構造体に頂点色用の変数 vertColor を追加し、サーフェスシェーダでの出力には vertColor の色と不透明度を使用するように変更を加えた。ここで使用した変数名（vertColor）自体は何でもよく、Input 構造体において頂点色を示す変数であることを識別するための“COLOR”セマンティックが指定されていればよい。

```
Shader "Custom/MySurfaceShader"
{
    Properties
    {
        // 変更無し
    }

    SubShader
    {
        Tags { "RenderType"="Opaque" }
        LOD 200
        Cull Off

        CGPROGRAM

        // 変更無し

        struct Input
        {
            float2 uv_MainTex;
            float4 vertColor : COLOR
```

```
        };

        // 変更無し

        void surf (Input IN, inout SurfaceOutputStandard o)
        {
            // Albedo comes from a texture tinted by color
            fixed4 c = tex2D (_MainTex, IN.uv_MainTex) * _Color;
            c = IN.vertColor;
            o.Albedo = c.rgb;
            // Metallic and smoothness come from slider variables
            o.Metallic = _Metallic;
            o.Smoothness = _Glossiness;
            o.Alpha = c.a;
        }
        ENDCG
    }

    FallBack "Diffuse"
}
```

図 2.9-5　新規作成したサーフェスシェーダ（MySurfaceShader.shader）

追加した「Cull Off」はカリングに関する設定である。特に指定が無い場合はカリングによりポリゴンの表面のみが描画されるが、「Cull Off」を指定することにより裏面も描画されるようになる。また、シェーダ名を「Custom/MySurfaceShader」に変更するとともに、ファイル名もシェーダ名と同じものに変更しているが、C#スクリプトとは異なり必ずしも両者を一致させる必要はない。

VertexColor マテリアルのシェーダを「Custom/MySurfaceShader」に変更し、プレビュー実行した結果が図 2.9-6 である。前節で使用した「Sprites/Default」とは処理内容が異なるため、表示される三角形ポリゴンの色味に若干の違いはあるが、各頂点に設定した頂点色が反映されるようになったことがわかる。マテリアルで使用するシェーダの変更は、同じマテリアルを使う別のゲームオブジェクトにも影響を与える(自動的に同じシェーダに変更される)ことに注意が必要である。

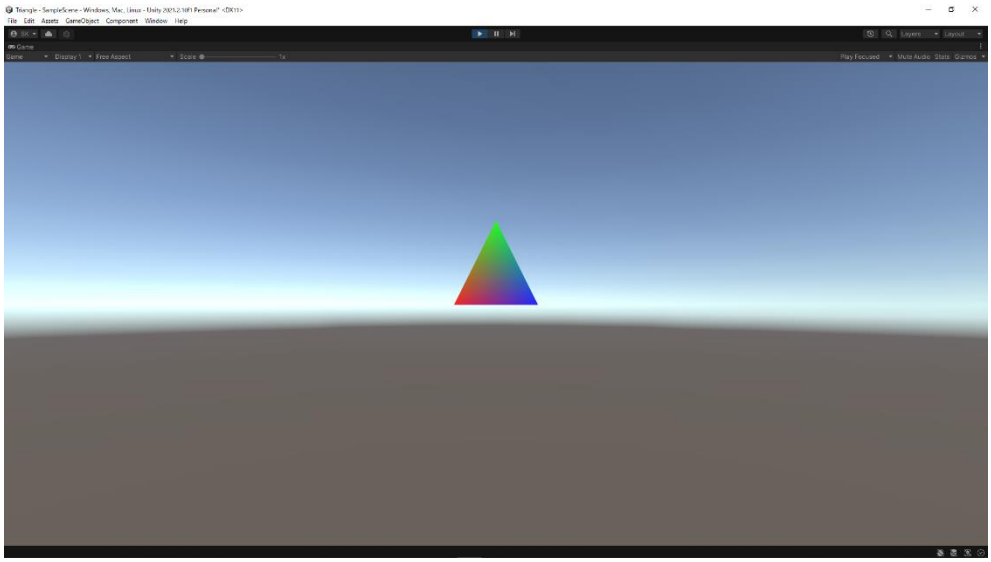
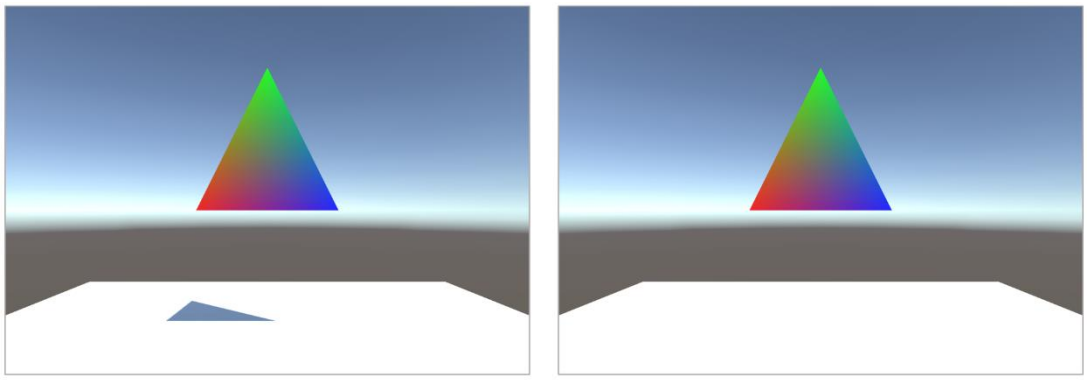


図 2.9-6　ここまでの実行結果

図 2.9-5 の最後で FallBack として指定されているものは、SubShader 内にハードウェアと互換性のあるシェーダが見つからなかった場合に使用するシェーダである。ここで指定したシェーダは、SubShader 内で行われなかった処理に対しても適用される。本節で作成したシェーダにおいて、SubShader ブロック内では描画色に関する処理のみを行っており、影に関する処理内容は記述されていない。このため、描画色については SubShader 内に記述された処理が、影についてはビルトインシェーダの Diffuse による処理が適用される。この場合、図 2.9-7(a)のようにシーン内に板ポリゴンを配置することにより、光源により三角形ポリゴンが落とす影が板ポリゴン上に描画される。シェーダから

「FallBack “Diffuse”」をコメントアウト、または「FallBack Off」としてフォールバックシェーダを無効にすることにより、図 2.9-7(b)のように影のみが描画されなくなる。



(a) FallBack “Diffuse”　　(b) FallBack Off

図 2.9-7　フォールバックシェーダの設定の違いによる比較

カスタムシェーダを使用する際の注意として、Unity エディタでのプレビュー実行時であれば特に問題は無いが、作成したカスタムシェーダをビルド後のアプリケーションでも使うためには設定が必要となる場合がある。これについては 2.17 節のビルドの項で説明する。

本節ではシェーダの働きについて簡単に紹介したが、シェーダについてはそれだけで専門書が一冊書けるぐらいのボリュームがあるので、その詳細について興味のある方は専門書を参照されたい。

2.10　ゲームオブジェクトのプレハブ化

前節までの手順で「頂点色を反映した三角形ポリゴンを描画する」機能を持つゲームオブジェクトを作成することができた。このゲームオブジェクトをヒエラルキーウィンドウ内で複製(Duplicate)することで、位置やスケールを変更した三角形ポリゴンをシーン内に複数配置することができるが、「プレハブ化」という手順を取ることにより作成したゲームオブジェクトを再利用しやすい形のアセットとして扱うことができる。手順は簡単で、ヒエラルキーウィンドウ内のゲームオブジェクトをプロジェクトウィンドウにドラッグドロップするだけである。図 2.10-1 は、前章までに作成した三角形ポリゴンを描画するゲームオブジェクト(ここではアタッチした C#スクリプト名に合わせて Triangle3 という名称とする)をヒエラルキーウィンドウからプロジェクトウィンドウにドラッグドロップした際のプロジェクトウィンドウの表示である。プレハブ化されたゲームオブジェクトはプロジェクトウィンドウ内で本図に示した形状のアイコンで表示される。元のゲームオブジェクトと同じ名称でプレハブは作成されるが、別の名称に変更することもできる。以降は、このアイコンをプロジェクトウィンドウからヒエラルキーウィンドウに必要個数分ドラッグドロップするだけで、三角形ポリゴンを描画するゲームオブジェクトをシーン内に複数配置することができる。本章後半で紹介する VisAssets における可視化モジュールの実体は、可視化フローを構成する小機能を実装したゲームオブジェクトをプレハブ化したものである。

図 2.10-1　プレハブ化されたゲームオブジェクト

2.11　インスペクタからのパラメータ変更(1)

前節までの手順で作成したゲームオブジェクトでは、インスペクタから表示位置やスケールを変更することができるが、三角形ポリゴン自体の形状や描画色を変更することはできない。これは、三角形ポリゴンを構成する各頂点座標が C#スクリプトの Start()関数内に直接記述されており、それらを変更するにはスクリプト内の頂点座標や頂点の色を直接書き換える必要があるためである。図 2.11-1 に示す変

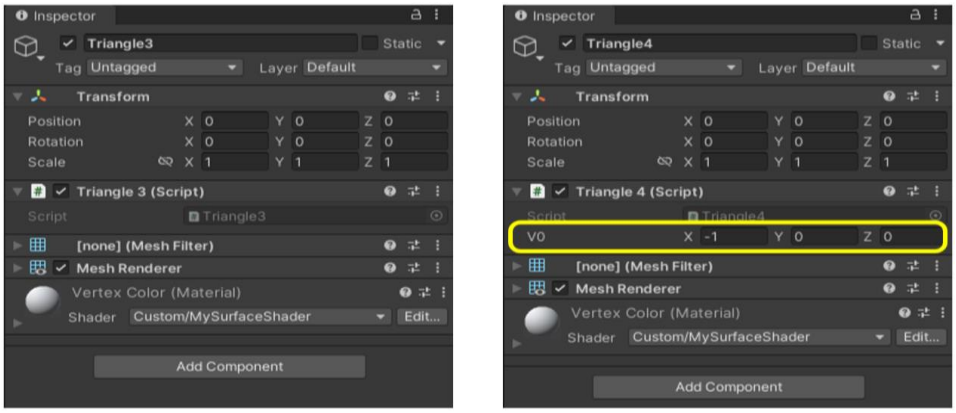
更を加え、これまで Start()関数内で設定していた頂点座標用変数をパブリックなメンバ変数とすることでインスペクタに表示されるようになり、スクリプトを修正することなくインスペクタから各頂点座標の初期位置を設定できるようになる。メンバ変数の宣言部では var による暗黙的な変数型の指定ができないため、Vector3 型であることを明示するよう変更している。

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Triangle4 : MonoBehaviour
{
    public Vector3 v0 = new Vector3(-1, 0, 0);
    [HideInInspector]
    public Vector3 v1 = new Vector3( 0, 2, 0);
    [HideInInspector]
    public Vector3 v2 = new Vector3( 1, 0, 0);

    void Start()
    {
        // vertices
        var vertices = new List<Vector3>();
        var v0 = new Vector3(-1, 0, 0);
        var v1 = new Vector3( 0, 2, 0);
        var v2 = new Vector3( 1, 0, 0);
        vertices.Add(v0);
        vertices.Add(v1);
        vertices.Add(v2);

        /* ～～～ 中略 ～～～ */
    }

    void Update()
    {
    }
}
```

図 2.11-1　頂点座標用変数のパブリックなメンバ変数への変更（Triangle4.cs）

図 2.11-2 にスクリプトに変更を加える前後でのインスペクタの表示の変化を示す。本変更により「Triangle 4 (Script)」の部分に変数 V0 が追加されるが、これらは頂点座標用変数である v0 に対応するものである。元の変数名の先頭文字が小文字のアルファベットである場合、インスペクタ上では自動的に大文字に変換されて表示される。

本例のように、パブリックなメンバ変数とすることでインスペクタに表示することができるが、その量が増えるとインスペクタ表示のための負荷が増大する。例えば、デバッグ等を目的として数百の座標値などのデータを表示しようとすると、Unity エディタがフリーズしたような状態になってしまう。このような変数をパブリックなメンバ変数として宣言したい場合は、対象となる変数に対して個別に[HideInInspector]属性を付けることで、インスペクタへの表示を抑制するなどの対策が必要となる。本例では、頂点座標用変数 v1 と v2 に[HideInInspector]属性を付加することにより、v0 と同様にパブリックなメンバ変数でありながら、インスペクタには表示されないよう制限している。



(a) スクリプト変更前　　(b) スクリプト変更後

図 2.11-2　スクリプト変更前後でのインスペクタの表示の変化

図 2.11-2 (b)において、スクリプトアタッチ直後の変数 v0 の値は C#スクリプト内に記述した初期値となる。プレビュー実行中以外のタイミングで変数 v0 の値を変更すると、変更後の値が新たな初期値としてゲームオブジェクト側に記録される(C#スクリプト自体は変更されない)。プレビュー実行により、Start()関数内ではインスペクタの頂点座標値が Mesh クラスに登録され、それに従って三角形ポリゴンが描画される。

2.12　インスペクタからのパラメータ変更(2)

前節までの手順で、インスペクタからの頂点座標値の変更により三角形ポリゴンの形状を変更できるようになった。しかし、この方法で変更した座標値が反映されるのはプレビュー実行前のみであり、プレビュー実行中にインスペクタの頂点座標を変更しても三角形ポリゴンの形状は何ら変化しない。そこで本節では、プレビュー実行中での頂点座標値の変更が、三角形ポリゴンの形状に即時反映されるよう C#スクリプトに更なる変更を加える。

図 2.12-1 は図 2.11-1 の C#スクリプトに対して網掛部の変更を加えたものである。まず、頂点座標、頂点色、インデックス情報を格納する List クラスと、Mesh クラスを示す変数の 4 つをメンバ変数として宣言するように変更している。これらはスクリプト内部で使用する変数のため、インスペクタに表示する必要はない。このためプライベート変数として宣言している。このようにメンバ変数として宣言することで、これまで Start()関数内のローカル変数としてのみ使用していた各変数を Update()関数内でも参照することができる。今回プレビュー実行中に変更するのは頂点座標のみであるため、頂点色およびインデックス情報については Start()関数内で一度だけ設定し、頂点座標のみを Update()関数内でフレーム毎に更新するようスクリプトに変更を加えている。また、変更した頂点座標による Mesh クラス内のジオメトリ情報の更新についても Update()関数内で行うよう変更を加えている。図 2.12-2 は本変更を加えた後のプレビュー実行の結果である。本図で表示されている三角形ポリゴンは、プレビュー実行中にインスペクタから頂点座標値を変更して変形させたものである。ここでは v0 の X 座標を初期値の「-1」から「-3」に変更している。本変更により、プレビュー実行中に行ったパラメータの変更が描画結果に即時反映されるようになる。また、“RequireComponent”属性で指定されたコンポーネントがアタッチされていない場合、C#スクリプトのアタッチ時にそれらが自動的にアタッチされるよう変更を加えた(本例では MeshFilter および MeshRenderer が自動的にアタッチされる)。

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

[RequireComponent(typeof(MeshFilter))]
[RequireComponent(typeof(MeshRenderer))]
public class Triangle5 : MonoBehaviour
{
    // 頂点座標部には変更無し
    List<Vector3>    vertices;
    List<Color>      colors;
    List<int>        indices;
```

```
    Mesh                mesh;

    void Start()
    {
        vertices = new List<Vector3>();

        colors = new List<Color>();
        var c0 = new Color(1, 0, 0);
        /* ～～～ 中略 ～～～ */
        colors.Add(c2);

        indices = new List<int>();
        indices.Add(0);
        indices.Add(1);
        indices.Add(2);

        mesh = new Mesh();

        var filter = GetComponent<MeshFilter>();
        if (filter != null) filter.mesh = mesh;
    }

    void Update()
    {
        // 以下はStart()からの移動
        vertices.Clear();
        vertices.Add(v0);
        vertices.Add(v1);
        vertices.Add(v2);

        mesh.Clear();
        mesh.vertices = vertices.ToArray();
        mesh.colors = colors.ToArray();
        mesh.SetIndices(indices, MeshTopology.Triangles, 0);
        mesh.RecalculateNormals();
    }
}
```

図 2.12-1　実行中の座標変更を三角形ポリゴンの形状に反映（Triangle5.cs）

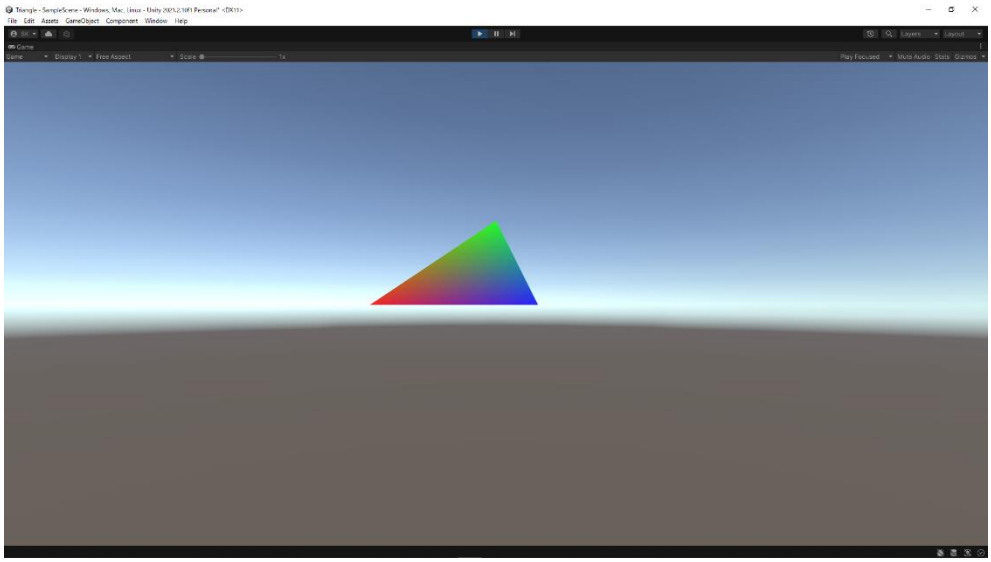


図 2.12-2　プレビュー実行の結果

2.13　インスペクタからのパラメータ変更(3)

前節での変更により、プレビュー実行中のインスペクタからのパラメータ変更が描画結果に即時反映されるようになった。しかし、数値を変更する都度テキストボックスから入力するのは面倒である。そ

こで GUI スライダを使って頂点座標 v0 の X 座標値を設定できるよう変更を加えた C#スクリプトが図 2.13-1 である。

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Triangle6 : MonoBehaviour
{
    [Range (-3, 0)]
    public float v0x = -1;
    // その他のメンバ変数には変更無し

    void Start()
    {
        // 変更無し
    }

    void Update()
    {
        v0.x = v0x;
        vertices.Clear();
        vertices.Add(v0);
        vertices.Add(v1);
        vertices.Add(v2);

        // その他変更無し
    }
}
```

図 2.13-1　GUI スライダによる座標変更（Triangle6.cs）

本変更では、頂点座標 v0 の X 座標値を変更するための変数 v0x をパブリックなメンバ変数として宣言するとともに、[Range]属性を付与している。[Range]属性を付けることにより、変数 v0x は数値を入力するためのテキストボックスではなく GUI スライダとしてインスペクタ上に表示される。また、[Range]属性では入力可能値の範囲指定をしており、本例の場合 -3～0 の範囲に入力値を制限している。図 2.13-2 にインスペクタでの表示を示す。プレビュー実行中のスライダの操作により、頂点座標 v0 の X 座標値も変更されていることがわかる。

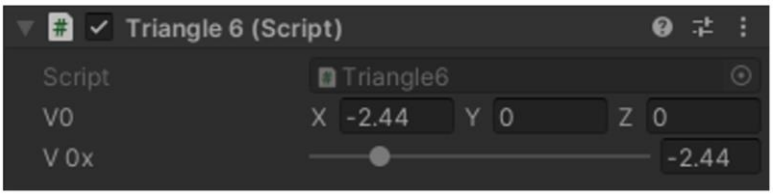


図 2.13-2　[Range]属性を付けた変数のインスペクタでの表示

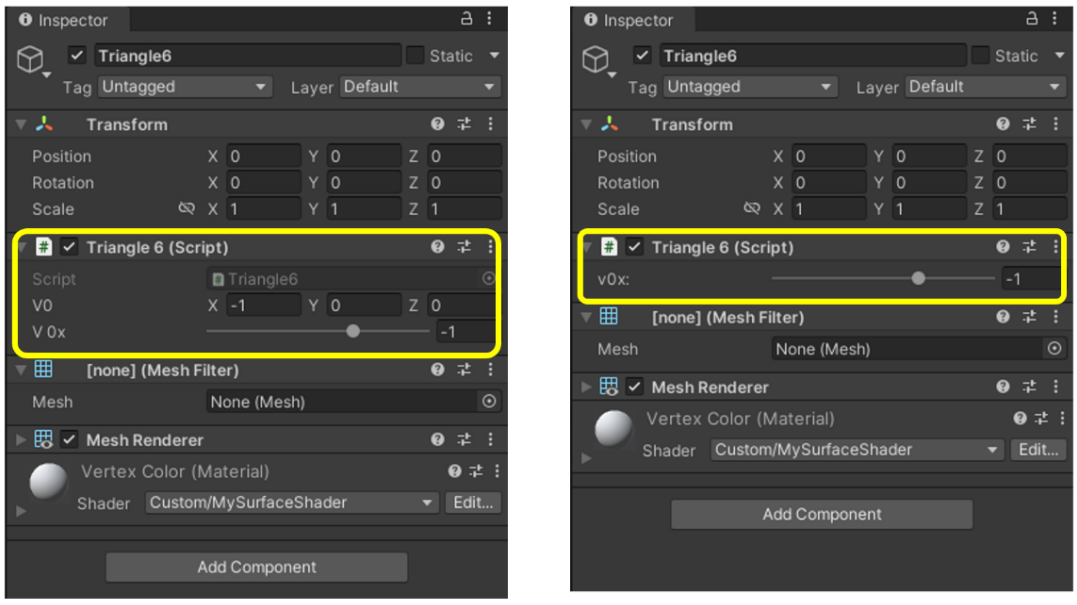
2.14　インスペクタからのパラメータ変更(4)

前節の変更により、GUI スライダを使ってメンバ変数を変更できるようになった。しかし、図 2.13-2 をよく見ると C#スクリプトでは「v0x」というメンバ変数名であったものが、インスペクタ上では「V 0x」として表示されていることがわかる。これはインスペクタへの表示にあたり、先頭文字の小文字から大文字への変更や、区切り文字と判断した位置(本例では数字の「0」)へのスペースの挿入を Unity エディタが自動的に行うためであるが、本来の変数名で表示したい、変数名とは異なるわかりやすい名称に変更して表示したい、C#スクリプト内の出現順とは異なる並びで各変数を表示したい、などの場合には「カスタムエディタ」という機能を用いることでインスペクタ上の表示を整形できる。

```
#if UNITY_EDITOR
using UnityEditor;

[CustomEditor(typeof(Triangle6))]
public class Triangle6Editor : Editor
{
    public override void OnInspectorGUI()
    {
        var triangle = target as Triangle6;
        triangle.v0x = EditorGUILayout.Slider("v0x:", triangle.v0x, -3f, 0);
    }
}
#endif
```

図 2.14-1　カスタムエディタの例（Triangle6Editor.cs）

図 2.14-1 は、図 2.13-1 で示した「Triangle6」クラスに対し、カスタムエディタを作成するための C#スクリプトである。プロジェクトフォルダ内の任意の場所に「Editor」という名称のフォルダを作成し、その中に本スクリプトを配置することでカスタムエディタが適用され、Triangle6.cs をアタッチしたゲームオブジェクトのインスペクタの表示が変化する。Editor フォルダはプロジェクトフォルダ内に複数存在してもよく、カスタムエディタ用酸クリプトはそのいずれかに格納されていれば自動的に認識される。また、本スクリプトの内容を Triangle6.cs の中に直接記述してもよい。カスタムエディタの適用前後でのインスペクタの表示の変化を図 2.14-2 に示す。本例ではカスタムエディタでの操作対象となるのはパブリックなメンバ変数であるが、プライベートな変数として宣言されている場合はカスタムエディタからメンバ変数へのアクセス方法が異なる。本稿ではその方法については割愛する。



(a) カスタムエディタ適用前　　(b) カスタムエディタ適用後

図 2.14-2　カスタムエディタの適用によるインスペクタ表示の変化

ここまで、インスペクタからのパラメータ変更について紹介したが、インスペクタウィンドウが表示されるのは Unity エディタでの作業中のみであるため、ビルド後のアプリケーションで同様の操作を行うためには、パラメータ変更用のユーザインタフェースの実装などが必要となる。

2.15　テキストファイルからのジオメトリ情報の読み込み

これまでの例では、描画対象となる三角形ポリゴンを構成するジオメトリ情報を C#スクリプト内に直接記述していた。しかし、実際に Unity を使って可視化アプリケーションを構築しようとする場合には、まず外部ファイルからデータを読み込み、それに対して描画のためのアルゴリズムを適用するという手順を取ることになろう。そこで本節では外部ファイルからのデータの読み込み例として、これまで C#ス

クリプト内部で直接設定していたジオメトリ情報の代わりに、テキストファイルから取得したジオメトリ情報を使う例を紹介する。Unity におけるファイルからの読み込み方法はいくつかあるが、本節ではビルド後のアプリケーションへの組み込みデータとして用いることを想定し、アセットとしてプロジェクトに登録したテキストファイルからのデータの読み込み例について紹介する。

これまでの例において、C#スクリプト内に記述されていた三角形ポリゴンを構成するジオメトリ情報のみをテキストファイルに書き出したものが図 2.15-1 に示した triangle.txt である。各行には頂点ごとの三次元座標(x, y, z)および色(r, g, b)が順に記載されている。インデックス情報については 0 から始まる各行の行番号を用いる。Unity エディタ上ではテキストファイルを新規作成することはできないため、別途外部のテキストエディタで作成する。次に、このテキストファイルをアセットとしてプロジェクトに登録するために、プロジェクトウィンドウから Assets フォルダ内に「StreamingAssets」という名前のフォルダを新規作成し、その中に triangle.txt をドラッグアンドドロップして追加する(図 2.15-2)。

```
-1, 0, 0, 1, 0, 0
 0, 2, 0, 0, 1, 0
 1, 0, 0, 0, 0, 1
```

図 2.15-1　ジオメトリ情報を記載したテキストファイル（triangle.txt）

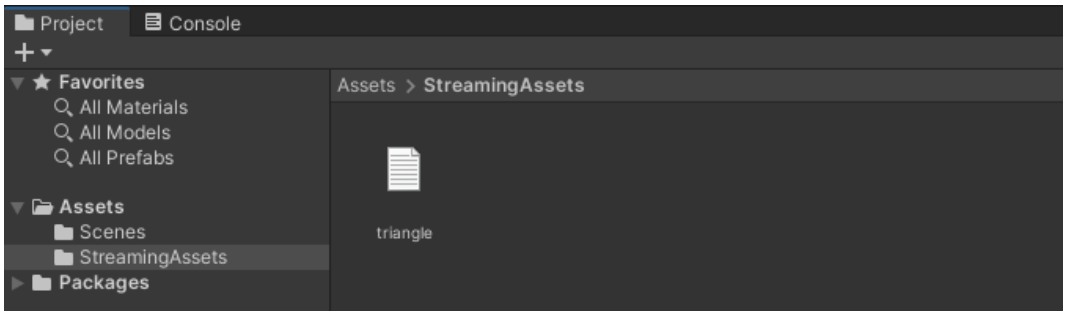


図 2.15-2　Assets/StreamingAssets フォルダに登録したテキストファイル

図 2.15-3 は、図 2.13-1 の C#スクリプトに変更を加え、テキストファイル内のデータからジオメトリ情報を生成し、三角形ポリゴンを描画するようにしたものである。テキストファイルの内容を文字列データとして読み込むための関数 ReadText()については図 2.15-4 に、読み込んだ文字列データから三角形ポリゴンの描画に用いるジオメトリ情報を生成する関数 ParseText()については図 2.15-5 に示す。

本例では、テキストファイルからのデータ読み込みにコルーチンによる非同期処理を用いている。Start()内で使用している StartCoroutine()はコルーチンの実行を開始するための Unity の C#関数であり、引数として指定された関数をコルーチンとして実行する。コルーチンとして実行される ReadText()では、まず読み込み対象となる StreamingAssets 内のテキストファイルのファイルパスを生成し、以降の処理でファイルの内容を文字列データとして取得する。読み込んだテキストファイルに書かれた頂点座標および色の情報は三角形ポリゴンを描画する際の初期値となるが、そのファイル名はパブリックなメンバ変数として宣言されているため、インスペクタの設定で triangle.txt とは別のファイルを指定することで形状の異なる三角形ポリゴンを描画することができる。

```
using System;
using System.IO;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.Networking;

public class Triangle7 : MonoBehaviour
{
    public string           filename = "triangle.txt";

    public List<Vector3>      vertices;
```

```
    public List<Color>        colors;
    List<int>                 indices;
    Mesh mesh;

    void Start()
    {
        vertices  = new List<Vector3>();
        colors    = new List<Color>();
        indices   = new List<int>();
        mesh      = new Mesh();

        var filter = GetComponent<MeshFilter>();
        filter.mesh = mesh;

        StartCoroutine(ReadText(filename, ParseText));
    }

    void Update()
    {
        mesh.Clear();
        mesh.vertices = vertices.ToArray();
        mesh.colors = colors.ToArray();
        mesh.SetIndices(indices, MeshTopology.Triangles, 0);
        mesh.RecalculateNormals();
    }
```

```
    IEnumerator ReadText(string filename, Action<string> callback = null)
    {
        // 図 2.14-4 を参照
    }

    void ParseText(string inputdata)
    {
        // 図 2.14-5 を参照
    }
}
```

図 2.15-3　テキストファイル内のデータからの三角形ポリゴンの描画（Triangle7.cs）

```
IEnumerator ReadText(string filename, Action<string> callback = null)
{
    var path = Path.Combine(Application.streamingAssetsPath, filename);

#if UNITY_ANDROID
    var www = UnityWebRequest.Get(path);
    yield return www.SendWebRequest();

#if UNITY_2020_2_OR_NEWER
    if (www.result == UnityWebRequest.Result.ProtocolError ||
         www.result == UnityWebRequest.Result.ConnectionError)
#else
    if (www.isHttpError || www.isNetworkError)
#endif
    {
        Debug.Log(www.error);
    }
    else
    {
        callback(www.downloadHandler.text);
    }
#else
    using var stream = new StreamReader(path);

    var text = stream.ReadToEnd();
    yield return null;

    callback(text);
#endif
}
```

図 2.15-4　テキストファイルからデータを読み込むための関数 ReadText()

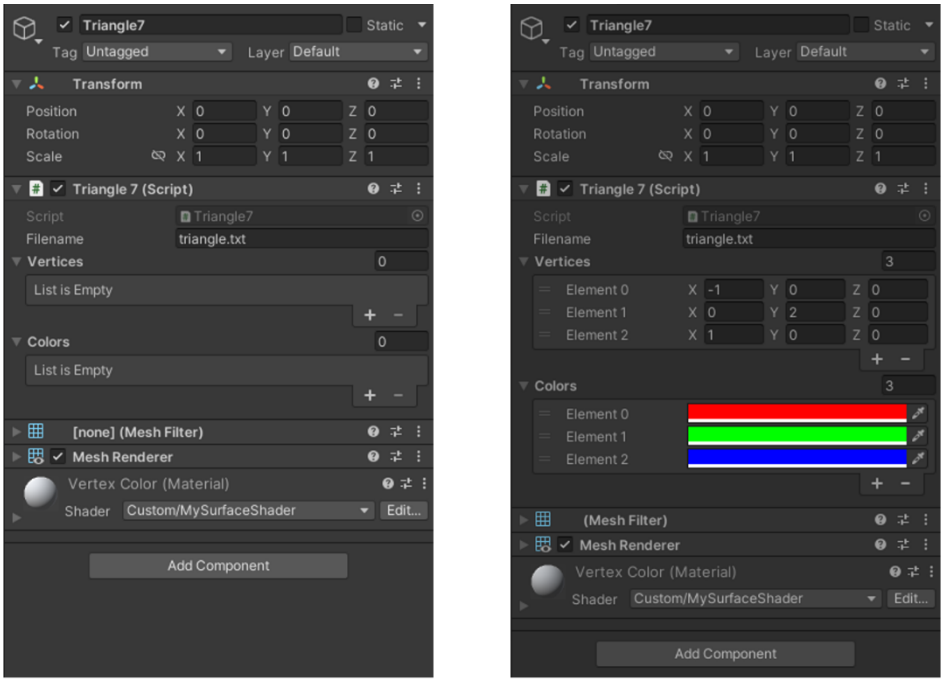
```
void ParseText(string inputdata)
{
    using var reader = new StringReader(inputdata);

    int vertID = 0; // current index of vertices
    var v = new float[6]; // input values
    while (reader.Peek() > -1)
    {
        var line = reader.ReadLine();
        var stringList = line.Split(',');
        for (int i = 0; i < 6; i++)
        {
            v[i] = float.Parse(stringList[i]);
        }
        vertices.Add(new Vector3(v[0], v[1], v[2]));
        colors.Add(new Color(v[3], v[4], v[5]));
        indices.Add(vertID);
        vertID++;
    }
}
```

図 2.15-5　文字列データからジオメトリ情報を生成する関数 ParseText()

ReadText()内では、その一部でディレクティブを用いた実行コードの切り替えを行っている。例えば、ファイルへのアクセスにおいて、対象となるプラットフォームが Windows の場合には C#の標準 I/O 関数を使ったファイル読み込みも可能であるが、Android で StreamingAssets 内のファイルにアクセスするには Unity の HTTP 通信用クラスである“UnityWebRequest”を用いる必要がある。本例では、UNITY_ANDROID ディレクティブを用いてこれらのコード切り替えを行っている。本例の場合、UNITY_ANDROIDディレクティブでのコード切り替えを行わず、UnityWebRequest を使った Android 用のコードを Windows 用にそのまま適用可能であるが、プラットフォームに応じた処理の切り替え例として、

ここではC#の標準I/O関数によるコードについても示している。

また、Unity 2020.2 以降とそれ以前とで UnityWebRequest のエラーチェック方法が異なるため、UNITY_2020_2_OR_NEWER ディレクティブを用いたコード切り替えを行っている。バージョンアップが頻繁に行われる Unity では、これまで利用してきた機能が非推奨となることがあるため、異なるプラットフォームやバージョンに対応した処理の記述が必要となる場合がある。ReadText()のコールバック関数として指定された ParseText()では、文字列変数に格納されたテキストファイルの内容を一行ごとに順に区切り文字で分割処理し、各頂点の三次元座標(vertices)および色(colors)用の変数に格納する。この時の行番号に基づき、三角形ポリゴンを構成するためのインデックス情報(indices)を併せて生成する。生成したジオメトリ情報は、これまでのスクリプトと同様に Update()内で Mesh クラスへの登録が行われ、MeshFilter および MeshRenderer コンポーネントを介して三角形ポリゴンが表示される。図2.15-6 は、図 2.15-3 の C#スクリプトをアタッチしたゲームオブジェクトのプレビュー実行前後でのインスペクタの表示の変化を示したものである。



(a) プレビュー実行前　　(b) プレビュー実行後

図 2.15-6　プレビュー実行前後でのインスペクタの表示の変化

vertices および colors をパブリックなリスト型変数として宣言しているため、プレビュー実行後のインスペクタの表示の変化からテキストファイルの記述順に両者が格納されたことが確認できる。色については Color 型の変数として宣言しているため、インスペクタ上の表示も RGB 値ではなく、RGB 値に基づく描画色で塗り潰された矩形領域で表示される。プレビュー実行直後は、テキストファイルから読み込んだ値がこれらの変数の初期値となる。プレビュー実行中にインスペクタからそれらを変更すると三角形ポリゴンの形状や色を変化させることができる。

2.16　バイナリファイルからのジオメトリ情報の読み込み

前節のテキストファイルに続き、本節ではバイナリファイルからのデータ読み込み例を紹介する。まず、図 2.15-1 で示したテキストファイルの各頂点の三次元座標(x, y, z)および色(r, g, b)を、リトルエンディアンの単精度実数(4 バイト)にて順に格納したバイナリファイル(triangle.bin)を作成する。テキストファイルと同様に、読み込み対象となるバイナリファイルをアセットとしてプロジェクトに登録するために、前節で作成した StreamingAssets フォルダ内に作成した triangle.bin をドラッグアンドドロップして追加する(図 2.16-1)。

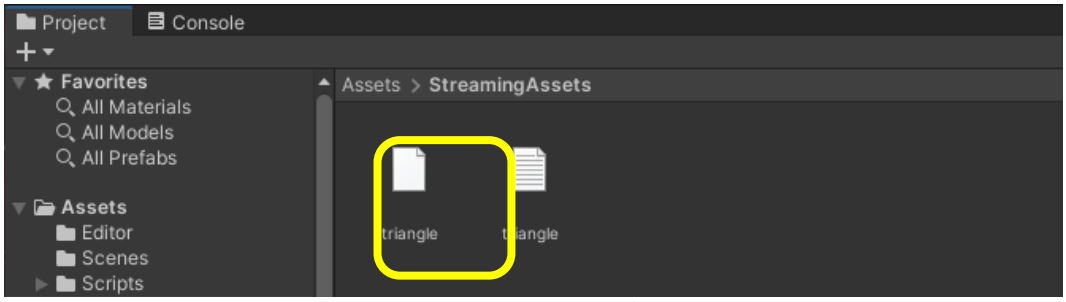


図 2.16-1　Assets/StreamingAssets フォルダに登録したバイナリファイル

図 2.16-2 は、前節の C#スクリプトに一部変更を加え、バイナリファイルを入力とするようにしたものである。バイナリファイルの内容を byte 型配列として Unity に読み込むための関数 ReadBinary()については図 2.16-3 に、読み込んだ byte 型配列データから三角形ポリゴンの描画に用いるジオメトリ情報を生成する関数 ParseBinary()については図 2.16-4 に示す。ReadBinary()および ParseBinary()については、前節の ReadText()および ParseText()からの関数名以外の変更部分を網掛で示した。読み込み対象となるデータ形式がテキストかバイナリかの違いのみであるため、C#スクリプト内での I/O 処理については例示したコードを参照されたい。本スクリプトをアタッチしたゲームオブジェクトのプレビュー実行前後でのインスペクタの表示についても、読み込み対象となるファイル名(triangle.bin)以外は図 2.15-6 と同様のものとなる。

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.IO;
using UnityEngine;
using UnityEngine.Networking;

public class Triangle8 : MonoBehaviour
{
    public string           filename = "triangle.bin";

    // その他の変数宣言部に変更無し

    void Start()
    {
        //変更無し

        StartCoroutine(ReadBinary(filename, ParseBinary));
    }

    void Update()
    {
        //変更無し
    }

    IEnumerator ReadBinary(string filename, Action<byte[]> callback = null)
    {
        // 図 2.16-3 を参照
    }

    void ParseBinary(byte[] inputdata)
    {
        // 図 2.16-4 を参照
    }
}
```

図 2.16-2　バイナリファイル内のデータからの三角形ポリゴンの描画（Triangle8.cs）

```
IEnumerator ReadBinary(string filename, Action<byte[]> callback = null)
{
    // ここまで変更無し
    else
    {
        callback(www.downloadHandler.data);
    }
#else
    using var stream = new FileStream(path, FileMode.Open);
    using var reader = new BinaryReader(stream);

    var length = (int)stream.Length;
    var data = new byte[length];
    reader.Read(data, 0, length);
    yield return null;

    callback(data);
#endif
}
```

図 2.16-3　バイナリファイルからデータを読み込むための関数 ReadBinary()

```
void ParseBinary(byte[] inputdata)
{
    using var stream = new MemoryStream(inputdata);
    using var reader = new BinaryReader(stream);

    int vertID = 0; // current index of vertices
    var v = new float[6]; // input values
    while (reader.BaseStream.Position != reader.BaseStream.Length)
    {
        for (int i = 0; i < 6; i++)
        {
            v[i] = reader.ReadSingle();
        }
        vertices.Add(new Vector3(v[0], v[1], v[2]));
        colors.Add(new Color(v[3], v[4], v[5]));
        indices.Add(vertID);
        vertID++;
    }
}
```

図 2.16-4　バイト列データからジオメトリ情報を生成する関数 ParseBinary()

2.17　プラットフォームを指定したビルド

作成したシーンを、Unity エディタ上でのプレビュー実行ではなく特定のプラットフォーム向けにビルドするにはメインメニューの[File]から[Build Settings]を選択する。図 2.17-1 は Build Settings の画面である。

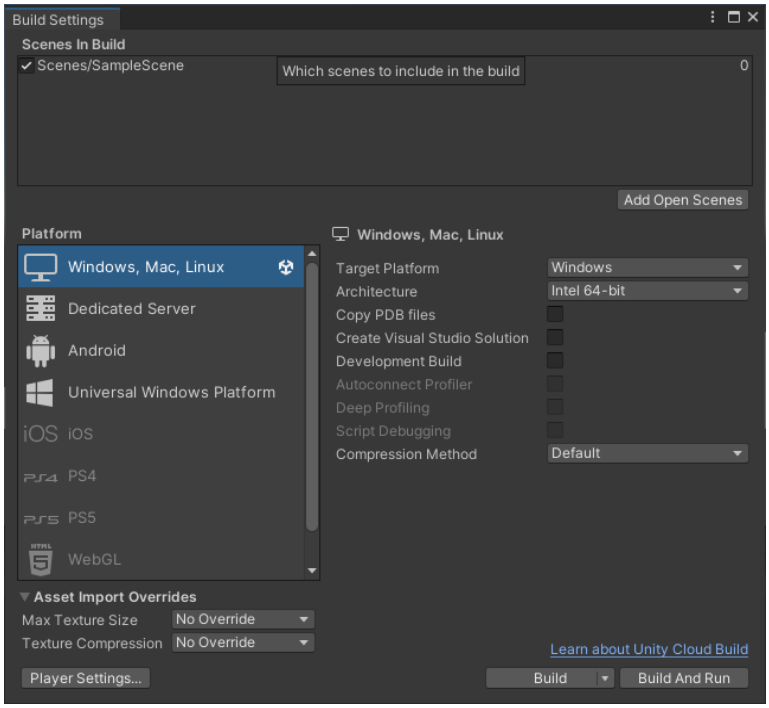


図 2.17-1　「Build Settings」の画面

ここでは、まずビルド対象となるシーンを「Scenes In Build」に追加する。新しいプロジェクトでの初めてのビルドの場合、「Scenes In Build」には何も登録されていない状態のため、ビルド後のファイルを実行しても作成したシーンが表示されないという状況に陥る。「Add Open Scenes」ボタンを押すことで、現在 Unity エディタで編集中のシーンをビルド対象に追加することができる。「Platform」には現在ビルド可能なプラットフォームが表示されている。ここで、現在選択中のものとは別のプラットフォームに変更した場合(例えば「Windows, Mac, Linux」から「Android」に変更)、まず「Switch Platform」ボタンを押してビルド環境の変更を行う。次に「Build」または「Build And Run」ボタンを押すことで、指定したフォルダに実行用ファイルやパッケージファイルを生成する(「Build And Run」の場合は更に実行までを行う。Windows の場合は Alt+F4 キーで実行を終了)。選択したプラットフォームが Android の場合、Android デバイス(スマートフォン、タブレット、Android ベースの HMD 等)を PC に接続した状態で「Build And Run」を選択することで、ビルド後のパッケージファイル(*.apk)のデバイスへの転送およびインストールまでが行われる。対象となるデバイスが HMD の場合、ヘッドトラッキング等のための XR デバイス用プラグインの導入も必要となる。図 2.17-2 は Android スマートフォンでの実行の様子である。2.15 節、2.16 節での C#スクリプトによる三角形ポリゴンを並べて表示したものである。フォールバックシェーダにより、板ポリゴン上には影も描画されている。

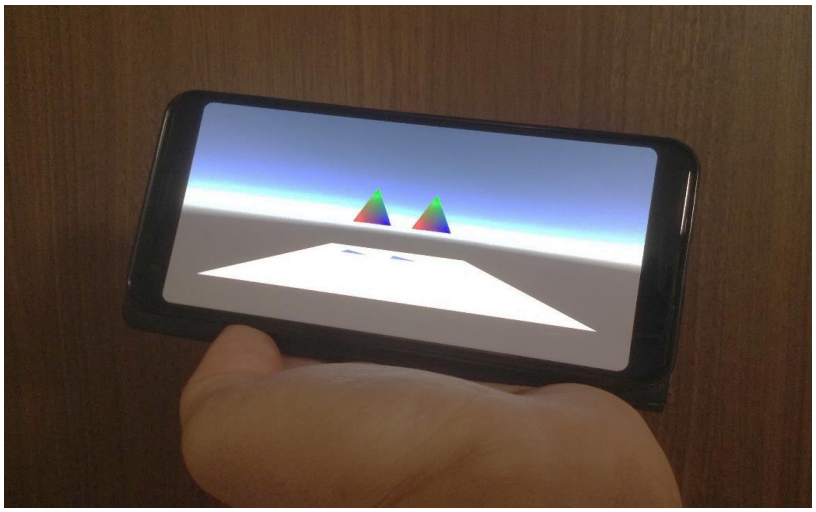


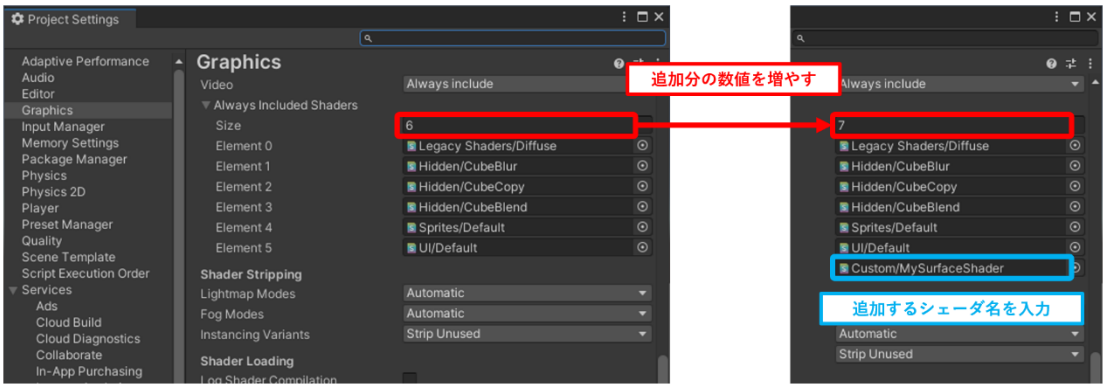
図 2.17-2　ビルドしたアプリケーションの Android スマートフォンでの実行の様子

ビルド後のアプリケーションの実行時、指定したはずのシェーダが適用されず対象のゲームオブジェクトがピンク色で塗り潰された状態で表示される場合がある。指定したシェーダがカスタムシェーダであり、マテリアルに適用するシェーダの指定や実行中での変更に図 2.17-3 に示すようなコードを使っているようであれば、ビルド前の事前設定で解決する可能性が高い。これは、シェーダを検索する Shader.Find()関数が対象とするのはビルトインシェーダのみであるため、それ以外のシェーダを指定した場合は 2.7 節における図 2.7-6 と同じく「シェーダ未設定」の状態で描画が行われるためである。

```
var renderer = GetComponent<MeshRenderer>();
renderer.material.shader = Shader.Find("Custom/MySurfaceShader");
```

図 2.17-3　マテリアルで使用するシェーダの C#スクリプトからの指定

カスタムシェーダを Shader.Find()関数での検索対象に含めたい場合は、ビルド前にそのカスタムシェーダをビルトインシェーダのリストに追加しておく必要がある。ビルトインシェーダのリストへのカスタムシェーダの追加は、[Project Settings]-[Graphics]-[Video]内にある[Always Included Shaders]から行う。図 2.17-4 にビルトインシェーダへのカスタムシェーダ追加のための設定画面と、追加前後での設定の変化を示す。



(a) カスタムシェーダ追加前　　(b) カスタムシェーダ追加後

図 2.17-4　ビルトインシェーダリストへのカスタムシェーダの追加

2.18　ここまでのまとめ

ここまでは、Unity をデータ可視化に用いるための基礎知識として、ファイルから読み込んだジオメトリ情報に基づき三角形ポリゴンを描画するまでの手順を紹介した。実際の可視化アプリケーションでは、まず何らかのデータセットを読み込み、それに対してカラースライスや等値面生成などのアルゴリズムを適用し、描画のためのジオメトリ情報を得る、といった工程が必要となるが、ここまでに紹介した知識を応用することでそれらの実装も可能と考える。続く後半では、これまでに紹介したノウハウも含まれている Unity 用可視化フレームワーク「VisAssets」について紹介するとともに、データ読み込みモジュールの作成方法についても併せて紹介する。

2.19　Unity 用可視化フレームワーク VisAssets

VisAssets は Unity 上で動作する可視化フレームワークである。VisAssets では可視化におけるデータフローを構成する小要素がモジュール化されており、それらを Unity エディタ上で接続するだけで可視化アプリケーションを開発できる。入力データとして三次元構造格子データを対象としており、配布パッケージ[3]に含まれるサンプルモジュールで読み込み可能なデータ形式であれば、Unity 上でのデータ読み込みから可視化結果の表示までをプログラミングレスで行うことができる。また、Unity はマルチプラットフォーム対応のゲームエンジンであり、ビルドターゲットを指定することで同一のプロジェクトから PC だけでなくスマートフォンやヘッドマウントディスプレイ（HMD）用の実行ファイルを生成できるという点も、効率的に可視化アプリケーションを開発する上で利点の一つであると言える。具体的な使い方についてはチュートリアル資料[4]が参考となろう。図 2.19-1 は Unity エディタ上での VisAssets を用いた可視化アプリケーションの構築の様子を示したものである。

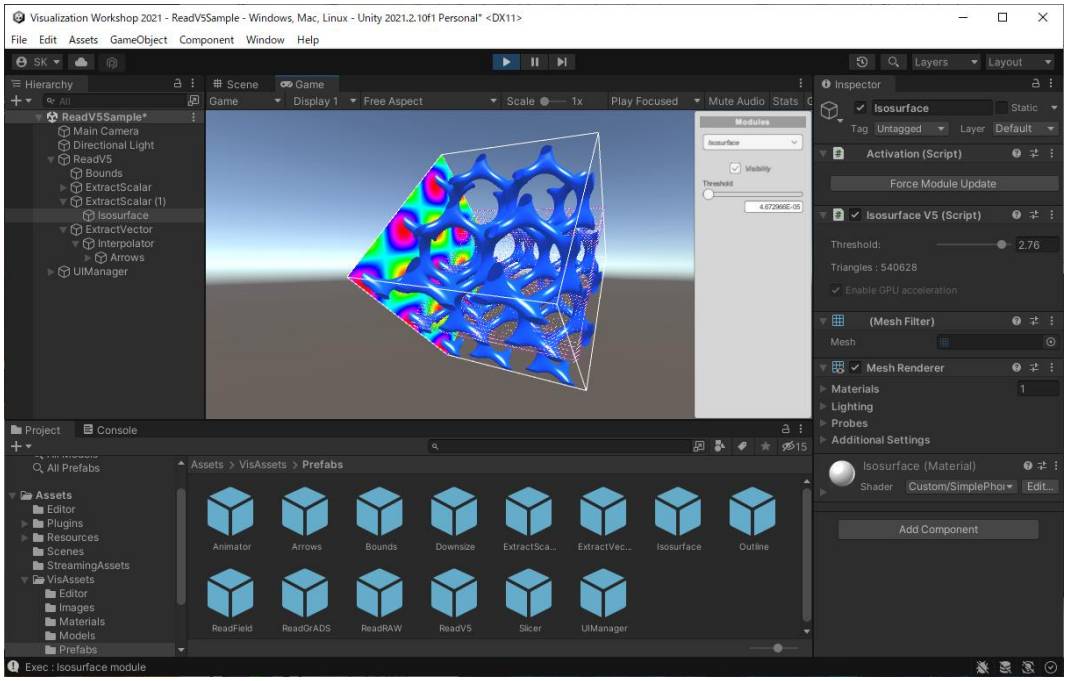


図 2.19-1　VisAssets を用いて構築した可視化アプリケーション

本図において、プロジェクトウィンドウ内にある水色のアイコン群が VisAssets における可視化モジュールである。モジュールの実態は、可視化フローを構成する小機能を実装した C#スクリプトや、その他必要なコンポーネントをアタッチしたゲームオブジェクトをプレハブ化(2.10 節で紹介)したものである。これらをヒエラルキーウィンドウにドラッグアンドドロップし、データフローに沿った適切な順でツリー接続することにより可視化アプリケーションを構築できる。直接接続されたモジュール間では、Unity のイベントループ (2.6 節にて述べた Update()関数)の仕組みを使った状態の監視が行われ、上流のモジュールや自分自身にデータ読み込みやパラメータ変更などの状態の変化が発生した場合には、下流に接続されたモジュールに順次処理が伝搬して駆動する仕組みになっている。

各モジュールを制御するパラメータの変更はプレビュー実行時であればインスペクタから行うことができるが、2.14 節で述べた通りビルド後のアプリケーションにはインスペクタウィンドウは表示されない。そこで、ビルド後のアプリケーションでも同様の操作ができるよう、実行画面内に各モジュールのパラメータ制御用のグラフィカルユーザインタフェース(GUI)を表示するためのモジュール(UIManager)を提供している。図 2.19-1 において、ゲームビューウィンドウの右端に表示されているパネルが UIManager による GUI である。HMD での実行時、本 GUI は 6 自由度の手持ちコントローラによる操作が可能な VRUI として動作する(図 2.19-2)。

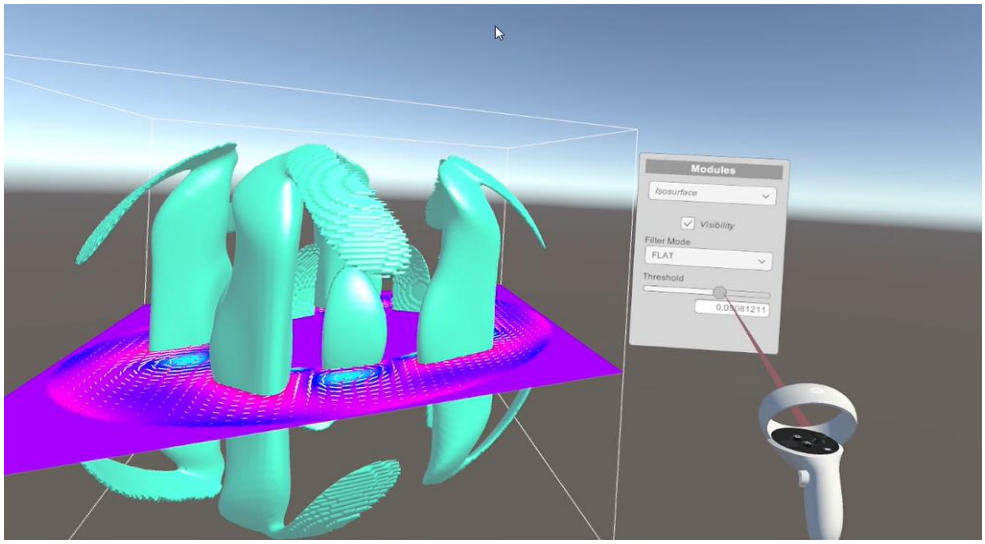


図 2.19-2　HMD(Meta Quest 2)での実行時の GUI モジュールによる XRUI

VisAssets では、いくつかのデータ形式に対する読み込みモジュールをサンプルモジュールとして提供している。表 2.19-1 にその一覧と読み込み可能なデータ形式を示す。各読み込みモジュールは、読み込んだ元データを VisAssets の内部データクラスに格納する。これらのモジュールで読み込み可能なデータセットは、テキストファイル、非圧縮のバイナリファイル、またはそれらの組み合わせから構成される。このため、2.15 節および 2.16 節で紹介したファイル読み込みの例が、サンプルモジュールにアタッチされた C#スクリプトを読む際の参考になるだろう。

表 2.19-1　データ読み込み用サンプルモジュールの一覧

モジュール名	機能
ReadField	テキストファイル(指定書式)の読み込み
ReadRAW	バイナリファイル(ヘッダ無しの生データ)の読み込み
ReadV5	CAVE 装置用可視化ソフトウェア VFIVE データ(実データ部読み込み用テキストファイル＋バイナリファイルによる実データ)の読み込み
ReadGrADS	地球科学関連データ用可視化ソフトウェア GrADS データ(実データ部読み込み用テキストファイル＋バイナリファイルによる実データ)の読み込み

2.20　VisAssets 用データモジュールの開発例

VisAssets では、内部データクラスにデータを格納さえしてしまえば、以降の可視化フローを構成するフィルタリング用やマッピング用のモジュールについては共通に利用できる。このため、それらを適切に接続することによりプログラミングレスで可視化アプリケーションを構築することができる。もちろん、現在 VisAssets では対応していない直交格子データであっても、内部データクラスへの格納用 C#スクリプトを作成し、新たな読み込みモジュールを開発することができる。図 2.20-1 は内部データクラスの構造である。

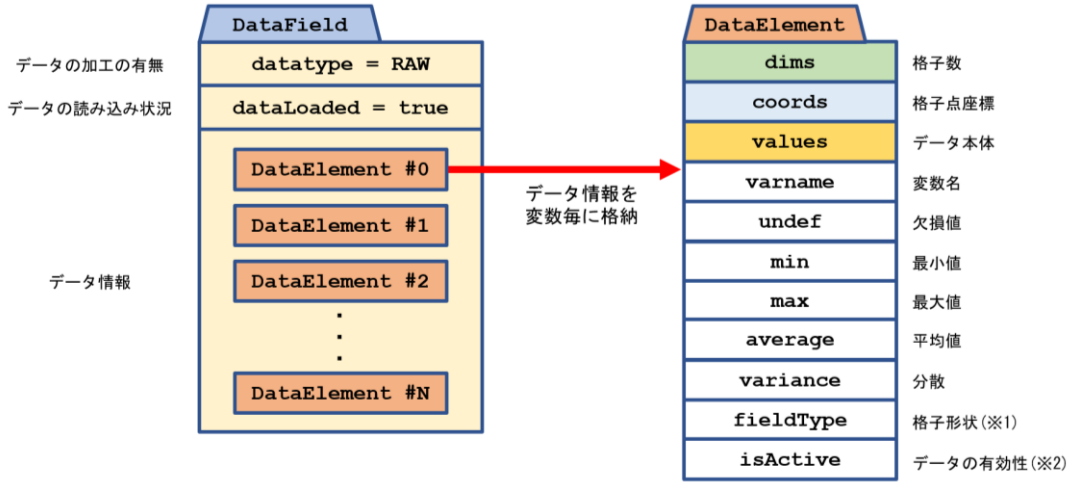


図 2.20-1　内部データクラスの構造

本図において、DataField がデータセット全体を、DataElement がデータセットに含まれる各変数を管理するデータクラスである。複数の変数が含まれるデータセットにおいて、それぞれが異なる格子情報を持つ場合にも対応可能な構成となっている(例えば、気象データにおける高度方向 N 層からなる大気気温データ、高度方向 1 層からなる地表面温度のように層数が異なるデータが同一のデータセット内にあるような場合にも対応可能)。

サンプルモジュールでは未対応のデータ形式を VisAssets で用いるには、前述の通り入力データを本データクラスに格納するための C#スクリプトを記述する必要がある。VisAssets では、データ読み込み、フィルタリング、マッピング、それぞれのモジュールの役割毎に三種のテンプレートクラスを用意して

おり、いずれかのテンプレートクラスを継承したサブクラスを作成することで新たなモジュールを作成することができる。モジュール間の状態遷移の仕組みについては各テンプレートクラスに組み込まれているため、モジュール開発者はモジュールの機能を実現するためのルーチンをサブクラスに実装するだけでよい。

ここでは、バイナリファイル(ヘッダ無しの生データ)の読み込みを行う ReadRAW モジュールを例にその動作を解説する。以下で紹介する C#スクリプトにおけるバイナリファイルの読み込みルーチン自体は、2.16 節で紹介したものとほぼ同じであるため、両方のコードを比較すると VisAssets でのモジュール開発方法を具体的に知ることができるだろう。

まず始めに、データ読み込みモジュール用のサブクラスを作成する。今回対象となるのはデータ読み込みモジュールであるため、テンプレートクラスとして ReadModuleTemplate クラスを継承するサブクラスを作成する。図 2.20-2 にその基礎となるコードを示す。

```
using System;
using System.IO;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.Networking;
using VisAssets;

public class ReadRAW : ReadModuleTemplate
{
    public override void InitModule()
    {
        // テンプレートクラスの Start()関数から呼ばれるオーバーライド関数
        // モジュールの初期化処理を記述する
    }

    public override int BodyFunc()
    {
        // テンプレートクラスの Update()関数から呼ばれるオーバーライド関数
        // モジュールの更新が必要な場合に実行する処理を記述する

        return 1;
    }
}
```

図 2.20-2　ReadModuleTemplate を継承したサブクラス

本図にて赤字で示した通り、ReadRAW は ReadModuleTemplate を継承したサブクラスとなっている。ここで、メンバ関数である InitModule()と BodyFunc()は三種すべてのテンプレートクラスに共通するオーバーライド関数である(その他のオーバーライド関数については割愛する)。InitModule()は、ReadModuleTemplate の Start()関数内で呼び出される仕組みになっており、モジュールの起動時に一度だけ呼び出される。ここにはモジュールの初期化に関する処理を記述する。BodyFunc()についても同様に、Update()関数内で呼び出される仕組みになっている。ReadModuleTemplate において、Update()関数自体はフレーム毎に呼び出されるが、BodyFunc()内の処理は上流のモジュールや自身の変化を検知した場合にのみ実行される仕組みとなっている。両オーバーライド関数に対し、データ読み込みのための具体的な処理を実装したものが図 2.20-3～2.20-6 に分割して示した C#スクリプトである(説明を簡単にするため、実際のコードからその一部を省略している)。本スクリプトでは、入力データの格子数、ファイル名、変数(物理量)名をインスペクタから指定することにより、指定されたファイル内のデータを内部データクラスに格納する。読み込み対象となるファイルは、格子数が dims[0]×dims[1]×dims[2]のデータを、リトルエンディアンの単精度実数(4 バイト)にて順に格納したバイナリファイルである。

```
using System;
using System.IO;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.Networking;

public class ReadRAWBinary : ReadModuleTemplate
{
    public Vector3Int dims; // グリッドサイズ
    public string filename; // ファイル名
    public string varname;  // 変数(物理量)の名称

    List<float> values; // バイナリファイルから読み込んだデータ本体の一時格納用
    List<float> coords; // 格子点座標の一時格納用

    public override void InitModule()
    {
        // テンプレートクラスの Start()関数から呼ばれるオーバーライド関数
        // モジュールの初期化処理を記述する

        // 一時格納用変数の初期化
        values = new List<float>();
        coords = new List<float>[4]; // 0:x, 1:y, 2:z, 3:(x, y, z)
        for (int i = 0; i < 4; i++)
        {
            coords[i] = new List<float>();
        }
    }

    public override int BodyFunc()
    {
        // テンプレートクラスの Update()関数から呼ばれるオーバーライド関数
        // モジュールの更新が必要な場合に実行する処理を記述する

        // データ読み込み用関数をコルーチンとして実行
        StartCoroutine(ReadBinary(filename, ParseBinary));

        return 1;
    }

    private IEnumerator ReadBinary(string filename, Action<byte[]> callback = null)
    {
        // 図 2.20-4 を参照
    }

    private void ParseBinary(byte[] inputdata)
    {
        // 図 2.20-5 を参照
    }

    private void InitCoord() // 格子点座標データの生成用関数
    {
        // 図 2.20-6 を参照
    }
}
```

図 2.20-3　バイナリファイルの VisAsset の内部データクラスへの格納 (ReadRAW.cs)

```
private IEnumerator ReadBinary(string filename, Action<byte[]> callback = null)
{
    var path = Path.Combine(Application.streamingAssetsPath, filename);

    var www = UnityWebRequest.Get(path);
    yield return www.SendWebRequest();

    if (www.result == UnityWebRequest.Result.ProtocolError ||
         www.result == UnityWebRequest.Result.ConnectionError)
    {
        Debug.Log(www.error);
    }
    else
    {
        callback(www.downloadHandler.data);
    }
}
```

図 2.20-4　バイナリファイルからデータを読み込むための関数 ReadBinary()

```
private void ParseBinary(byte[] inputdata)
{
    // 格子点座標データの生成(図 2.19-8 を参照)
    InitCoord();

    using var stream = new MemoryStream(inputdata);
    using var reader = new BinaryReader(stream);

    while (reader.BaseStream.Position != reader.BaseStream.Length)
    {
        values.Add(reader.ReadSingle());
    }

    // 一変数分のデータを格納するため、DataField(df)内に DataElement を一つ生成
    df.CreateElements(1);

    // 生成した DataElement に読み込んだデータ(直交格子データ)を格納
    df.elements[0].SetDims(dims);
    df.elements[0].SetCoords(coords);
    df.elements[0].SetValues(values);
    df.elements[0].SetVarName(varname);
    df.elements[0].SetFieldType(DataElement.FieldType.UNIFORM);
    df.elements[0].SetActive(true);

    // データの読み込みが完了したことを示すフラグを立てる
    df.dataLoaded = true;

    // データが中央に表示されるよう、センタリングおよびリサイズ
    // (テンプレートクラスのメンバ関数を利用)
    if (centering)
    {
        Centering(autoResize);
    }
}
```

図 2.20-5　バイト列データを内部データクラスに格納する関数 ParseBinary()

```
private void InitCoord()
{
    values.Clear();
    for (int i = 0; i < 4; i++)
    {
        coords[i].Clear();
    }

    // set coordinate of each axis as uniform data
    for (int i = 0; i < 3; i++)
    {
        for (int n = 0; n < dims[i]; n++)
        {
            coords[i].Add((float)n);
        }
    }

    // merge coordinates to 4th list
    for (int k = 0; k < dims[2]; k++)
    {
        for (int j = 0; j < dims[1]; j++)
        {
            for (int i = 0; i < dims[0]; i++)
            {
                coords[3].Add(coords[0][i]);
                coords[3].Add(coords[1][j]);
                coords[3].Add(coords[2][k]);
            }
        }
    }
}
```

図 2.20-6　内部データクラス格納用格子点座標データを生成する関数 InitCoord()

各メンバ変数の初期化部分を除き、2.16 節で紹介したコード群と本節のコードとを比較すると、ParseBinary()でのデータ読み込み以降の部分で内部データクラスへの格納処理が加わっていることがわかる。これらは全てのデータ読み込みモジュールに共通する処理であり、DataField クラスのメンバ関数を使ってデータの各情報を登録している。本スクリプトをアタッチしたゲームオブジェクトのインスペクタでの表示を図 2.20-7 に示す。本図において、Activation はモジュールの状態遷移を管理するためのコンポーネントであり、DataField は内部データクラス用のコンポーネントである。両者は共に VisAssets が提供する C#スクリプトであり、データ読み込みモジュールには必須のコンポーネントである。ReadModuleTemplate を継承したサブクラスが書かれた C#スクリプト(本節の例では ReadRAW.cs)をゲームオブジェクトにアタッチする際、両者のどちらがアタッチされていない場合、ReadModuleTemplate クラスの機能により自動的に追加される。CtrlOBJ は、読み込んだデータに基づく可視化結果をマウス操作で回転、移動、拡大縮小するための VisAssets のコンポーネントである。このゲームオブジェクトをプレハブ化することにより可視化モジュールとして完成する。

図 2.20-7　C#スクリプトをアタッチしたゲームオブジェクトのインスペクタ表示

図 2.20-8 は、本モジュールに対してフィルタリング、マッピングのモジュールを接続し、サンプルデータ(100×100×100、sample.bin)を読み込んだ場合のプレビュー実行画面である。

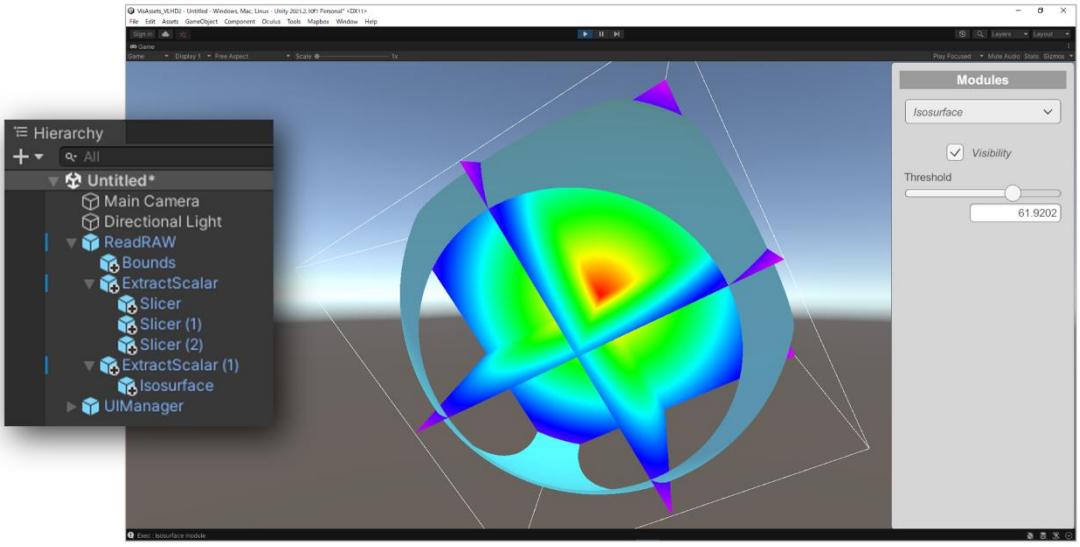


図 2.20-8　作成したデータ読み込みモジュールを使った実行画面

本図の左側に示したヒエラルキーウィンドウの表示では、ReadRAW モジュールに対してデータ領域を枠線で示す Bound、読み込んだデータからスカラー場を抽出する ExtractScalar、カラースライスを表示する Slicer、等値面を表示する Isosurface モジュールが階層的に接続されていることがわかる。また、ビルド後のアプリケーションにモジュール制御用パネルを表示するためのモジュールである UIManager をシーン内に配置することで、画面右端に GUI パネルを表示している。本図では、ドロップダウンメニューで選択されている Isosurface モジュールが現在の制御対象となっており、下にあるチェックボックスで等値面の表示・非表示の切り替えが、スライダで等値面生成の閾値変更を行うことができる。

2.21　おわりに

本節では、Unity を用いて可視化アプリケーションを開発するための基礎知識を紹介するとともに、その応用として VisAssets でのデータ読み込みモジュールの開発までを順を追って紹介した。前半で紹

介した三角形ポリゴンの描画のように、描画のためのジオメトリ情報を直接読み込むだけではなく、後半で紹介した可視化前の直交格子データの読み込みまでの情報が得られれば、そこに更に可視化のためのアルゴリズムを適用し、Unity で描画するまでを各自で行うことができるのではないだろうか。本節前半で使用した C#スクリプトなどの関連ファイルについては、Github のページ[5]からダウンロードできる。VisAssets についても実際にお使い頂くとともに、本節のノウハウを参考に、データ読み込みモジュールだけでなく、フィルタリングやマッピングについても新たなモジュールを是非開発頂き、今後の開発にフィードバックされることを期待したい。

謝辞：
本稿に関する研究の一部は JSPS 科研費 21K11916 および JP22K12054 の助成を受けたものです.

参考文献：
1) MRTK(https://github.com/microsoft/MixedRealityToolkit-Unity)
2) 宮地英生, 川原慎太郎, ″ゲームエンジンを用いた VR 可視化フレームワークの開発″, 日本シミュレーション学会論文誌, Vol.12, No.2, pp59-67 (2020), doi:10.11308/tjsst.12.59
3) VisAssets(https://github.com/kawaharas/VisAssets)
4) 第 5 回ビジュアリゼーションワークショップにおける VisAssets のチュートリアル資料(https://www.vsj.jp/visws2022/data/visws2022-kawahara.pdf)
5) 本節前半で使用したサンプルコード(https://github.com/kawaharas/UnityVis-Tutorial)

3　5G の可視化技術への応用

3.1 携帯電話網の変遷と 5G 時代の到来

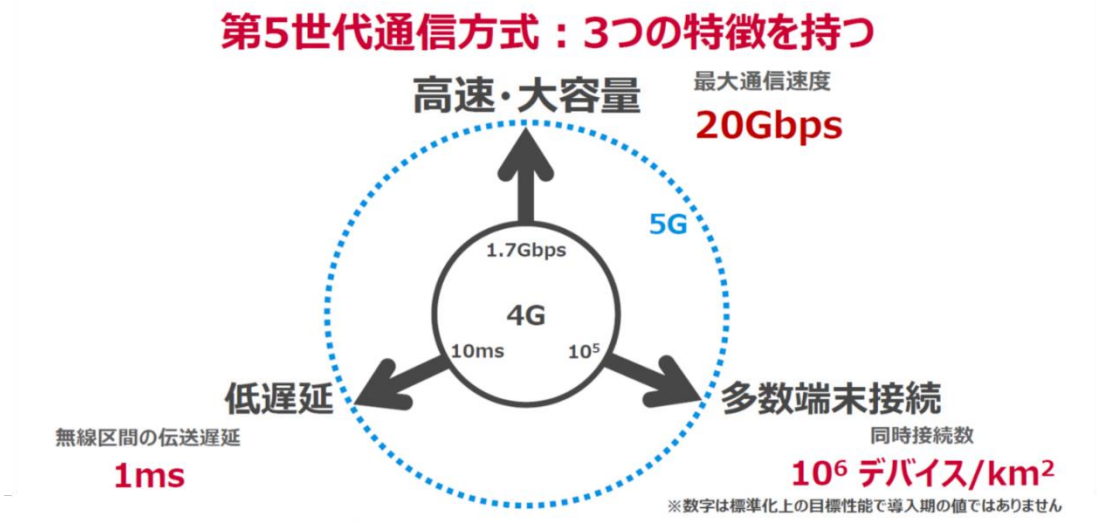
日本の携帯電話の歴史は 80 年代から始まっているが、本格的に普及した 90 年代より人々がいわゆる携帯電話と呼ばれるものが簡単に扱えるようになってから、既に 25 年以上が経過した。当初は携帯電話の呼び名の通り電話中心であったそれであるが、90 年代後半からデータ通信が可能となり、00 年代後半のスマートフォンの登場により、その主役は現在に至るまでデータ通信が主流である。各通信キャリアは繋がりやすさ（カバレッジ）と音声品質を求めていた 00 年代の”携帯電話”競争から、いつしか 3G 以降のデータ通信を主流とする方式に変化を遂げ、現在に至っている。

3G ではそれまでの回線交換と呼ばれるいわゆる通話中心だったシステムから、データと音声通話を ATM 交換機で接続し、両方を同時に扱うというシステムに変化した（２G ではデータ通信は別のシステムとして存在していた）。無線は CDMA 方式を採用し、同一周波数帯の多重度を格段に上げることが可能になった。さらに日本にとってはそれまで独自方式だった携帯電話システムが、グローバルで利用可能な IMT-2000 と呼ばれる方式になり、海外で自らの携帯電話をそのまま利用できるようになり、格段に便利になったのがこの時代である。（それまでは行く国によって携帯電話を各国際空港でレンタルするのが主流であった）この時代の端末機として進化した点があるとすれば、いわゆる写真撮影が可能な機体(写メール機)の登場、音楽再生機能や簡易型のインターネットアクセス機能（i-mode や ezWeb など）を中心にデータ通信の利用機会が増えていった時期である。

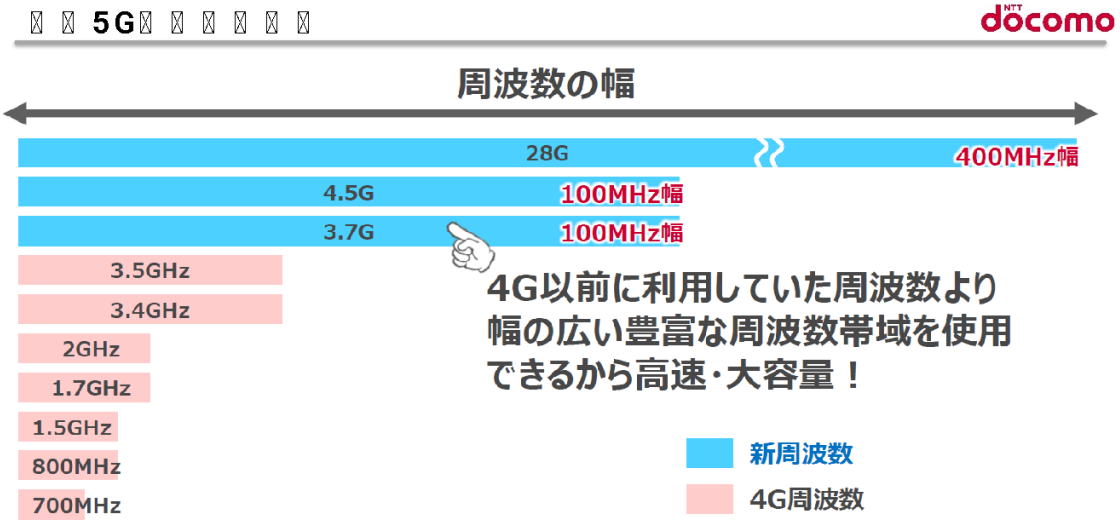
3.5/3.9/4G(もしくは LTE)ではデータ通信を主流とする方式に変更になった。音声通話の需要よりもデータ通信に重きを置いたシステムに変化し、そのシステムもインターネット標準である IP 方式に変化したのがこの時代である。無線通信の多重化も OFDMA 方式となり、多重度も格段に上がった。データ通信の親和性が強くなり、システムも簡易になったことで、データ通信時のシステム負荷が小さくなったため、より大きな帯域を通すことができるようになった。時を同じくしてスマートフォン時代に突入し、携帯電話と呼ばれていた時代の通話機としての利用方法は主流ではなくなり、SNS、動画視聴、ゲームなどが主流となった。同時にデータ通信機としての使い方も主流となり、専用のモバイルルータやテザリングと呼ばれるスマートフォンで WiFi を経由して通信を提供する機能が標準的になり、モバイルインターネットというものが通常のインターネットとは異なり、シームレスに提供されるようになった。これにより産業利用なども促進され、M2M 通信（Machine-to-machine communication）も使われるようになった。

これらの進化を遂げてきた通信網において、5G はどういう存在だろうか。５G は標準化（各国で相互

に接続可能にするための規格の統一を図る団体）においては、４Ｇよりも高速大容量、超低遅延、同時多数接続の３つの指標において、４Ｇ(LTE)よりも大きな進化を目標に置いて、その規格の策定が行われた。

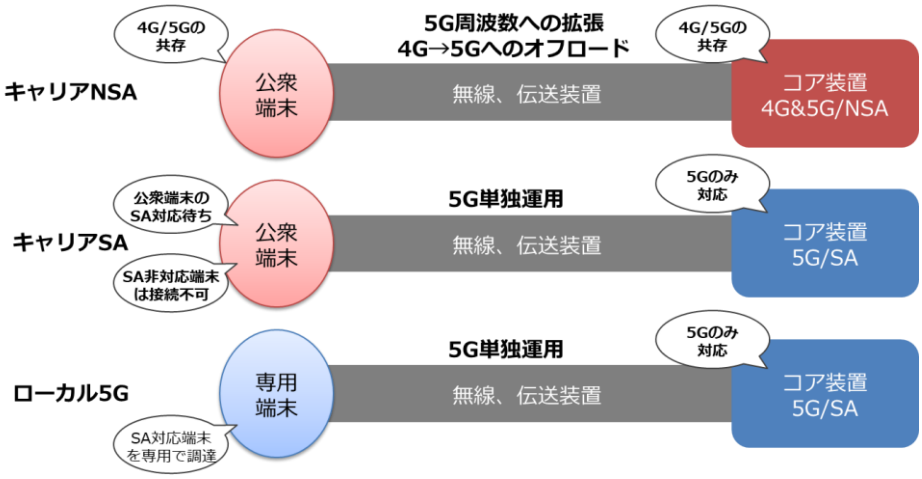


これらの指標を実現するために、大幅な周波数の割当変更が行われ、従来の周波数帯に加えて非常に高い周波数帯 4～6GHz 帯（通称 Sub6 帯）と 22GHz 帯（通称ミリ波帯）を用いて通信が行われることとなった。日本では総務省により、ユーザ数や置局計画に沿ってそれぞれの通信キャリアに公平に分配され、その中で運用されている。なお、5G の規格そのものは新しい周波数だけではなく、従前の 3G/4G が利用されていた周波数帯も使うことができるようになっている。ここまでの特徴を見ると、ほとんどの機能が 4G の特徴の延長上に存在するように見える。しかし、5G で使われている技術としては上記した新周波数の対応や、ネットワークの機能に関してはそれまでの 4G とは実は全く異なる。この進化が実は今後の大きな進化につながる布石となっていて、特にネットワークに関してはその柔軟性が格段に進化している。従来のネットワークとは全く異なる使い方ができるようになっているのが 5G の大きな特徴である。

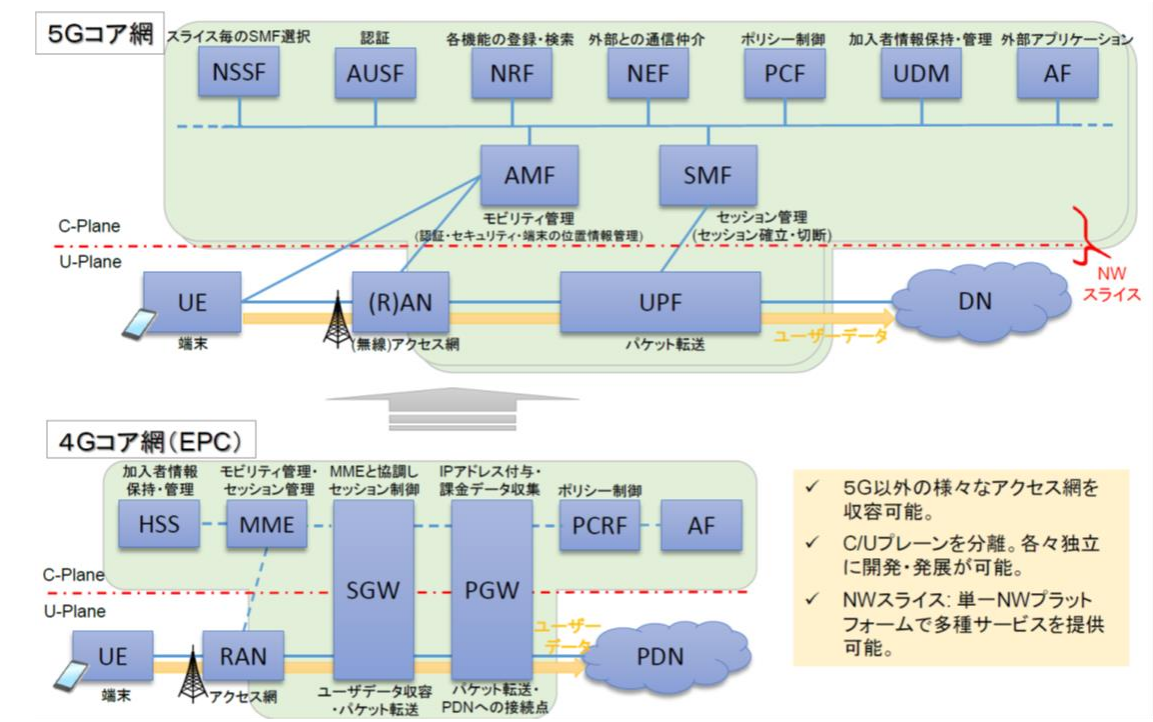


©2021 NTT DOCOMO, INC. All Rights Reserved & Confidential.

1つの例としてはLocal 5Gが存在する。従来であれば、専用の装置を中心に構成されていた携帯電話ネットワークシステムを、5Gでは技術と処理能力の進化により汎用サーバーの上に構築が可能になった。これにより、小さなシステムから大きなシステムまでを自由に構成することが可能になったため、高度なネットワークの制御機能を持つ5Gを自ら持つことができるようになった。これがLocal 5Gである。従来の狭い周波数帯で多数の端末がひしめき合い、時としてその影響で繋がりにくくなっているWiFiとは異なり、キャリアと同じ周波数帯や、制御方式を手に入れ、自らのシステムとして個別に運用できるようになったのがLocal 5Gである。これにより非常に安定した通信を通信キャリアに頼ることなく、自ら構築可能になった。もちろん、広域をカバーしようとすると非常にコストがかかるため、大群化効果を前提にビジネス設計されている通信キャリアのコストには敵わないが、通信キャリアが自らのビジネス性によってその通信網の設計を独自で行っているのに対して、Local 5Gは自らの収支での設計が可能であり、設計の自由度が格段に高い。こういったことができるようになったのも5Gの特徴の一つである。



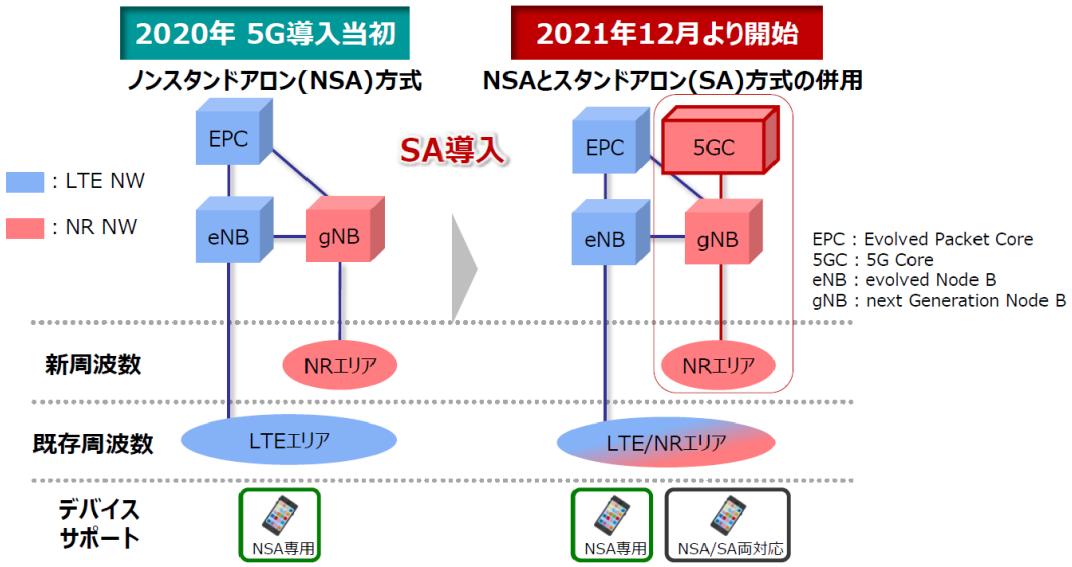
また、CUPSと呼ばれるコントロールプレーン（制御信号系ネットワーク）とユーザプレーン（ユーザのトラフィックが流れるネットワーク）の分離も大きな特徴である。これにより、制御系の独立で、制御システムの汎用化が図られており、前述したような汎用機での技術が利用可能になった。実際には5Gでは制御系のシステムはお互いがREST I/F（インターネットで通常用いられているHTTPをベースとするプロトコル）が用いられ、マイクロサービス化された制御システムへの機能追加や、開発自体の速度向上が期待されている。



3.2 5G の現在の特徴

日本では 5G は 2020 年 3 月に各通信キャリアから順次サービスが開始された。5G に限らず各世代の当初はその性能はその世代の規格上の所定性能を満たしていないことが多い。様々な原因があるが、システムの普及速度や投資の抑制、技術的な制約など（例えば対応するチップが大量生産されていないため、設計がこなれておらず消費電力が高いために性能が抑えられている）があるが、いずれにしても 5G もその世代の中で徐々に進化していくように設計されている。

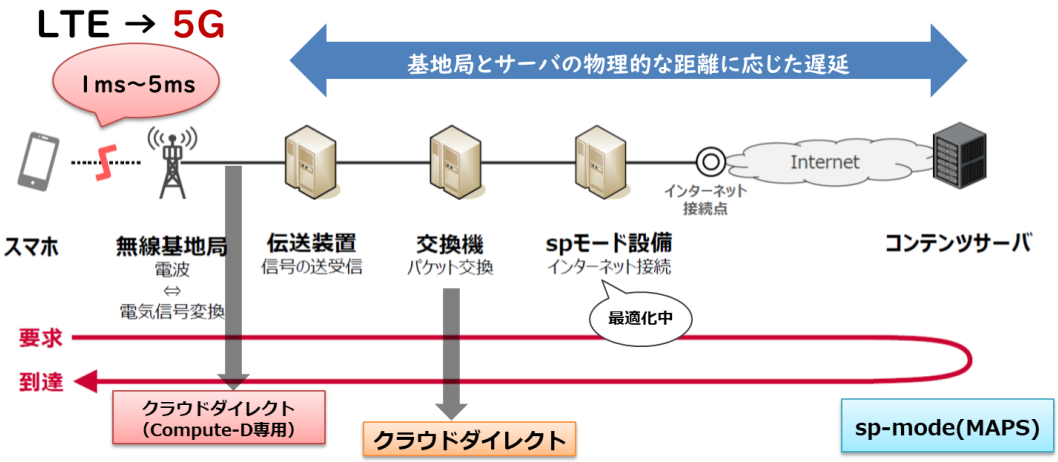
その制限された進化としての特徴を表しているものの一つに NSA がある。これは Non-stand-alone と呼ばれる方式で、5G の周波数を使った 5G 方式を提供しつつ、4G のシステムを一部流用することで、投資を抑制しつつ 5G に移行することができるようになっているシステムである。この方式を用いると、5G の恩恵である通信容量の大幅な拡大のメリットを即座に得られつつ、ネットワークでは従来の 4G ネットワークを使うことができる。これはそもそも日本以外の国での 5G の周波数の獲得をオークション制度で入札して取得しているキャリアの投資体力を温存するための一つの方法として考案されているという一面もある。ただ、この方法では 5G のネットワークの柔軟性に関する特徴は犠牲となり、SA 方式（Stand-alone 方式）と呼ばれる 5G の無線に 5G の交換機を組み合わせた方式ではないと様々な特徴が出せない。



NSA 方式でも特徴が出せるもの、SA 方式で特徴が出せるものがあるが、NSA 方式でも特徴が出せるものの一つに低遅延と広帯域がある。広帯域は CA 方式（キャリアアグリゲーション）と呼ばれる複数の周波数帯域を利用して広帯域を実現するシステムが 4G から実現されており、この機能が 5G でもそのまま利用できる。現状はこの機能を使って 4.2Gbps という広帯域を実現している。

3.3 クラウドダイレクトと Mobile Edge Compuing（MEC の提供）

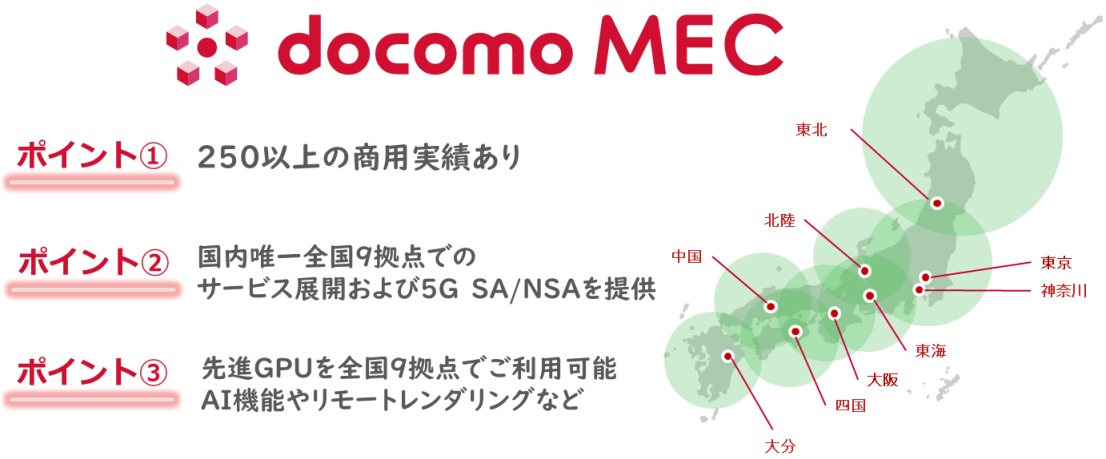
低遅延については、無線方式として従来の 4G に比べて 5G は低遅延になっていることもあり、その低遅延性を利用することができる。ただ、従前と同じネットワーク構成ではその低遅延性を十分に発揮できないため、例えば NTT ドコモでは「クラウドダイレクト」と呼ばれる特殊な接続方式を提供している。



上述した 4G のネットワーク構成による低遅延性を発揮できない理由は、そのネットワークの構造にある。従来通信キャリアが提供するモバイルインターネットは、ネットワークの構造として一つの大きな ISP（Internet Service Provider）として構成されていることが多い。この ISP は大きな IX（Internet Exchange）で他の ISP とつながっており、これにより全世界と通信できる構造となってい

る。この大きな IX は大都市圏に存在しており、日本では東京と大阪に存在している。そのため、この IX を経由しないとモバイルからの接続はインターネットの上に存在するサーバーと通信ができない。例えば、仮に広島に存在するサーバーに対して広島からアクセスしたとしても、その通信は広島⇒大阪⇒広島か、場合によっては広島⇒東京⇒広島という通信経路になり、仮に 5G の無線区間で低遅延を提供したとしても、その通信経路の長さからその恩恵を受けることができない。

そこでクラウドダイレクトと接続では、その拠点での接続を折り返したり、またはそこに仮想マシン（コンピュータ）への接続を行い、低遅延でのサービスを実現している。NTT ドコモではこのシステムを 2020 年の 6 月から提供しており、当初は東京、大阪、川崎、大分の 4 拠点で実施し、2022 年 7 月より広島、高松、名古屋、金沢、山形の 5 拠点での展開をしている。これらではいずれの場所でもクラウドダイレクトとそれにつながる仮想マシンサービスである Compute-O/V/D と呼ばれるサービスが有り、特に低遅延を要求されるサービスに対して利用をされている。クラウドダイレクト方式では、一般的な ISP 接続方式（NTT ドコモでは sp-mode と呼称）とは異なり、SIM レベル（契約レベル）での申し込みが必要となる。一般 ISP 接続では上記の通り遠回りとなるが、クラウドダイレクトではその所定の経路を経由する。ただし、移動は現在のところ考慮されておらず、例えば広島でのクラウドダイレクトの利用は広島での収容（その場所での制御とトラフィックのルーティング）が行われるために、利用には注意が必要になる。他社では今の所そういったサービスは提供されていないが、標準 ISP サービスのルーティングを変更し、できるだけ地方の中でルーティングして物理的距離が近い者同士が短い時間での接続（遅延時間の軽減）を図る取り組みはされており、今後に期待される。



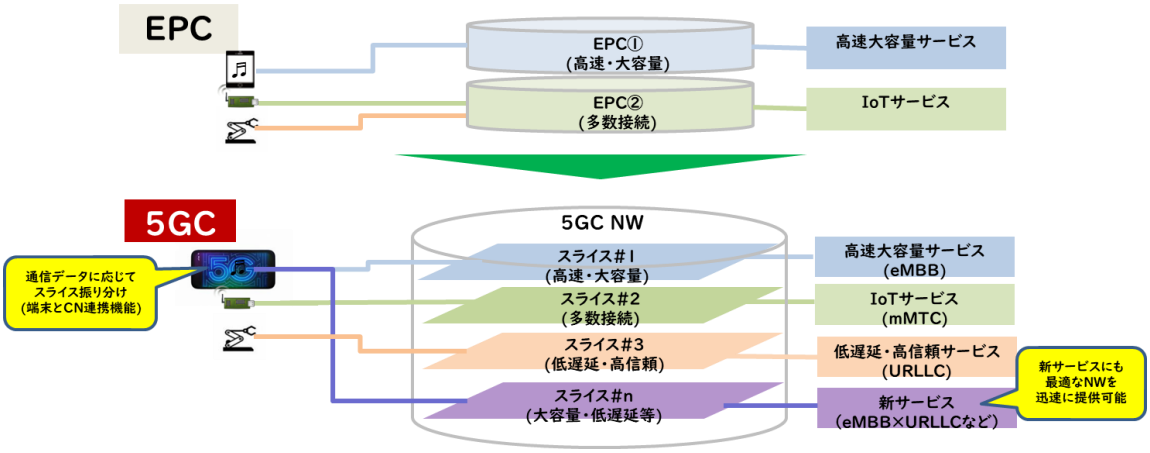
3.4 5G の今後の展開

5G の将来像を語る上では SA 方式への本格移行が欠かせない。NSA では上述したとおり、現状の 4G で実現されている方式を延長した議論しかできていない。今後本格的な 5G 方式の普及とともに SA 方式が主流となってくる。SA 方式では CUPS が導入されるために、様々なネットワークの柔軟性が提供予定になっている。これらの機能は 5G の標準化で議論されており、Rel.17 以降で議論されているため、これについて紹介する。なお、実際の提供については、まだ提供方法や経済合理性などについて議論がされており、実際に提供されるかどうかは未定である点にも注意したい。

3.4.1 ネットワークスライシングの提供

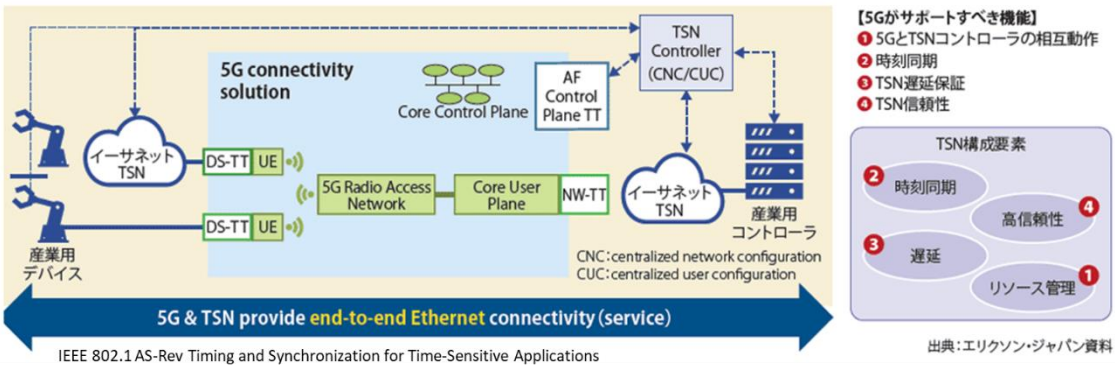
ネットワークスライシングとは、ネットワークの論理的な分割による様々な特徴を持った無線ネットワークサービスの提供のことを指す。これは CUPS によって分離可能となったユーザトラフィックを様々な組み合わせで提供することができる技術的な方式を指している。これが実現すると、例えば従前のモバイルネットワークでは、下りトラフィック帯域（インターネットからユーザ）と上りトラフィック帯域（ユーザからインターネット）は概ね 10 対 1 のような設計になっている（たとえば下りが 1Gbps 出せる場合には上りは 100Mbps）ことが一般的であるが、このバランスを変えたネットワークの提供が

可能になる。また、帯域が制限された非常に細いネットワークを安価に提供するなども理論上は可能になる。QoS レベルなどもコントロールできるために、帯域保証型サービスなども提供可能になると言われている。これらについては、まだ方式も含めて議論中ではあるが、価格なども含めて従来に無いアイデアであるためにどのように提供されるのかについては議論が活発である。



3.4.2 Time Sensitive Network の提供

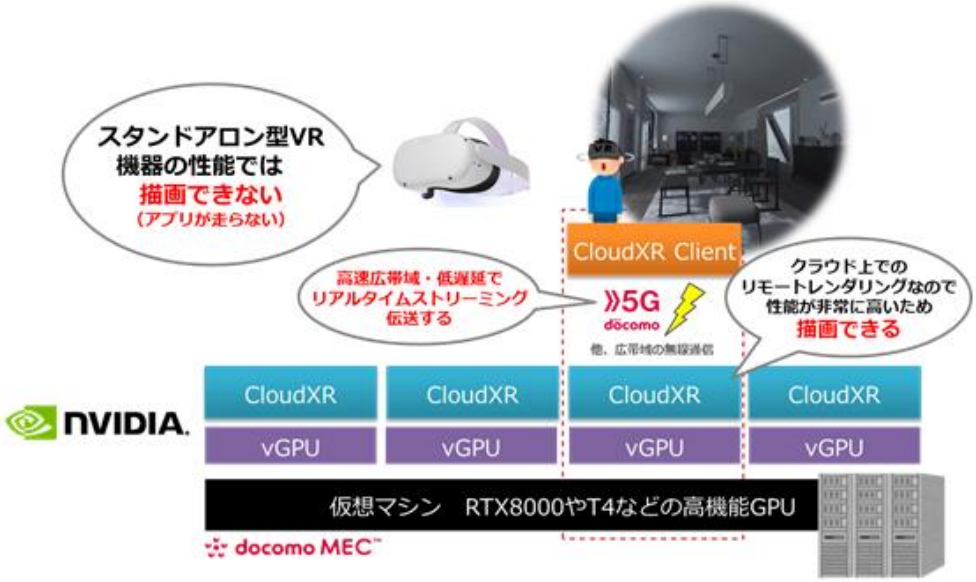
Time Sensitive Network と言われている、遅延保証型のネットワークの提供なども言われている。5G の低遅延性の特徴を活かして、遅延ループの観測をネットワーク内で入れることにより、正確に遅延を測定しつつ、それを安定させることで、ネットワークの遅延をシステムのコントロール可能な項目として設定することができるようになると言われている。遅延の測定をリアルタイムに行うことで、実際に発生した遅延を知ることができ、それをシステムの一部として反映させることができる。



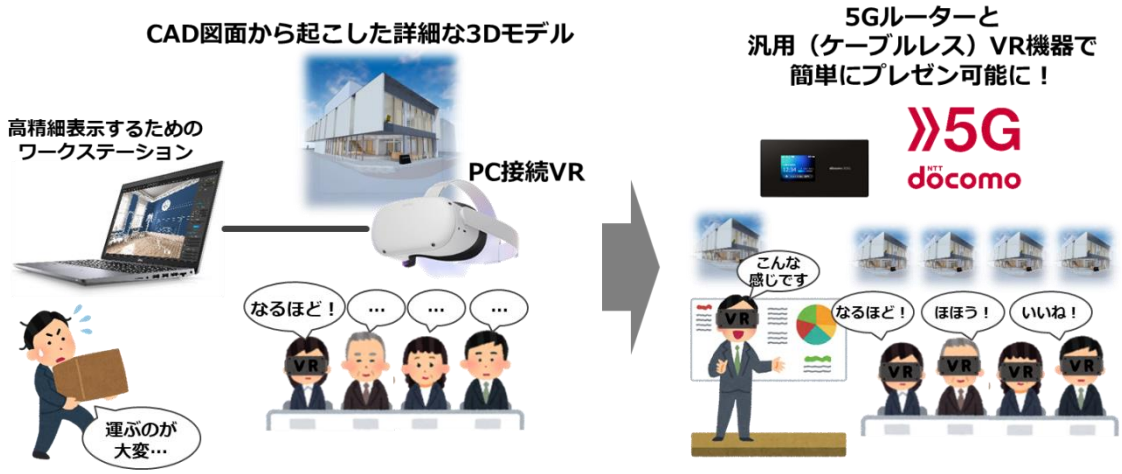
3.5 NTT Docomo における MEC を使った VR/AR リモートレンダリング

NTT ドコモでは上記のような 5G と MEC のシステムを用い「XR リモートレンダリングシステム」を構築した。これはグラフィックス描画能力の低い VR グラスや AR グラスなどに対して、クラウド（もしくは MEC）において、より消費電力の大きな、描画能力の高いチップ（グラフィックボード）を用いて 3D グラフィックスの描画を行い、その画像もしくは映像をストリーミングで端末に送付する。端末では VR グラスなどの 6DoF(前後左右の移動及び回転方向）の動きをサーバー側に送信するだけで、結果がストリーミング映像として返されるので、それを表示するだけで、持っている描画能力を超えて表示することが可能となる。しかし、この方法は通信に対して２つの大きな要求条件が必要となる。一つは広帯域である。描画する代わりにストリーミング映像として受信するために、高精細な映像を連続的に受信するだけの帯域が必要となる。もう一つは低遅延性である。6DoF の動きを送信し、それを元に映像を作成し、

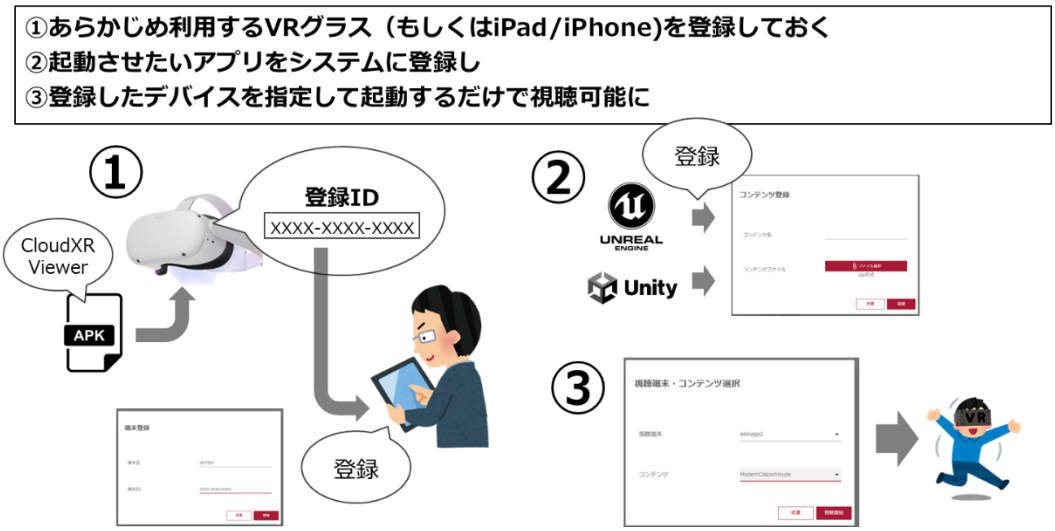
それをストリーミングで受け取る必要があり、その遅延が大きいと自分の動きに映像がついていかず、場合によっては激しく酔を感じてしまう要因になる。遅延を感じさせないよう低遅延性が必須となる。5G と MEC は前述したように、これらの２つの特性を持っており、VR リモートレンダリングの要求条件を満たす。



次に VR リモートレンダリングの利用用途について述べる。本システムを有効に利用する上での１つの例として、建築デザインにおける VR プレゼンがある。昨今では建築デザインは CAD で設計した建築図面を用いて 3D パース図（全体を俯瞰し、図面ではわかりにくい建物の全景を見えるようにした図）を作成するのが通例となっている。施工主に対してより直感的に特長を理解してもらうための工夫であり、実際にはその図を例えば建築中のマンションの販売チラシなどに使われる。この図を昨今より直感的に見せようという取り組みが進んでおり、VR プレゼンという形式で施工主に対して完成後の姿をウォークスルーで閲覧してもらうという取り組みが行われている。ただしこの場合には、建築デザイン会社はプレゼンを行う際に性能の高いノートパソコンを持っていく必要があり、さらに有線の VR グラスを接続して閲覧してもらう必要があり、非常に準備に時間がかかるもしくはコストがかかることから一部でしか実施できていなかった。そこで本システムを使うことにより、安価な VR グラスと 5G ルーターを持っていくだけで、簡単にプレゼンができるようになるといったメリットがある。実際にすでにいくつかの会社でこの取り組みがされており、特に遠方地でのプレゼンを実施する際に威力を発揮している。



本システムの構成と特徴について簡単に触れる。 利用者はあらかじめ準備しておいた VR アプリケーションをシステムにアップロードを行う。平行して自分が利用する VR グラスをシステムに登録を行う。次に閲覧をするために、あらかじめ登録をした VR グラスとアプリケーションを紐付け、アプリケーションの起動を行う。起動指示がされると、クラウド上に専用のレンダリングサーバーが立ち上がり、一定の時間後に閲覧が可能になる。本システムは従量課金となっており閲覧している時間に応じてレンダリングサーバーの利用料を請求する仕組みとなっている。閲覧終了次第レンダリングサーバーを停止することで課金を停止することができるため非常に安価に高精度な VR アプリケーションの走行が可能になる。



本システムの特徴としては、 VR アプリケーションに特になんの追加をすることなく、本システムを使うことができるということである。これは Open VR に対応しているアプリケーションであれば、リモートレンダリングに対応させることができる。つまり、既存のアプリケーションがあれば、すぐにでもこの XR リモートレンダリングシステムを利用することができる。

利用者が唯一気をつける点は、閲覧する場所に最も近いレンダリングサーバーを選択することである。これは前述したように MEC システムは日本全国に展開されており、 適切なサーバーを選ぶことによって最も効率的に利用することができるからである。距離的に一番近いサーバーは低遅延が期待でき、広帯域が保証しやすいというメリットが有る。

3.6　原子炉建屋内での機械学習による放射線源推定と可視化技術

「5G時代の可視化技術研究」WG第5回会合（2021年9月30日）において、国立研究開発法人　日本原子力研究開発機構　システム計算科学センター　副センター長　町田昌彦誌をお招きし、「原子炉建屋内での機械学習技術を用いた放射線源逆推定技術の研究開発と推定線源及び線量率可視化への期待」というタイトルで講演をいただいた。

福島第一原子力発電所（1F）の廃止措置に向けて、原子炉建屋に存在する干渉物撤去に関して、高線量下・高汚染下・不確定要素を含む環境条件での継続的な作業を考慮した技術開発が推進されている。廃止措置における燃料デブリの取り出しに先立って、原子炉建屋には事故による損傷状態が不明な場所が残り、未だに線量率が高い建屋内部でのアクセスルートを構築する必要がある。建屋内のアクセスルート構築準備の作業を行うには、可能な限り作業員の被ばく低減を図り、安全かつ効率的な作業計画を策定することが求められている。建屋内部の放射線量はガンマカメラ等の計測器で測定できるが、一般的に、線量分布状態を調べて放射線源を特定するためには数多くの遠隔操作による測定が必要であり、容易ではない。

日本原子力研究開発機構発機構システム計算科学センター（CCSE）では、少ない線量率の測定データから、線源分布を逆推定し、その線源分布から得られる空間線量率3次元分布を基に、作業計画を立案し、作業者の被曝量低減を図る技術を開発している。この計画では、サーベイメータ等を利用した空間線量率の測定を実施し、そのデータを基に機械学習を活用して3次元線量分布を逆推定する。機械学習で必要とされる学習データの作成には放射線輸送シミュレーションを活用する。逆推定に関する計算は遠隔地のスーパーコンピュータで実施し、計算結果の次元線量分布は作業者に転送され、VRやARを活用して作業者の訓練や作業計画の策定に利用される。図3.1.1-1に本計画の概要を示す。

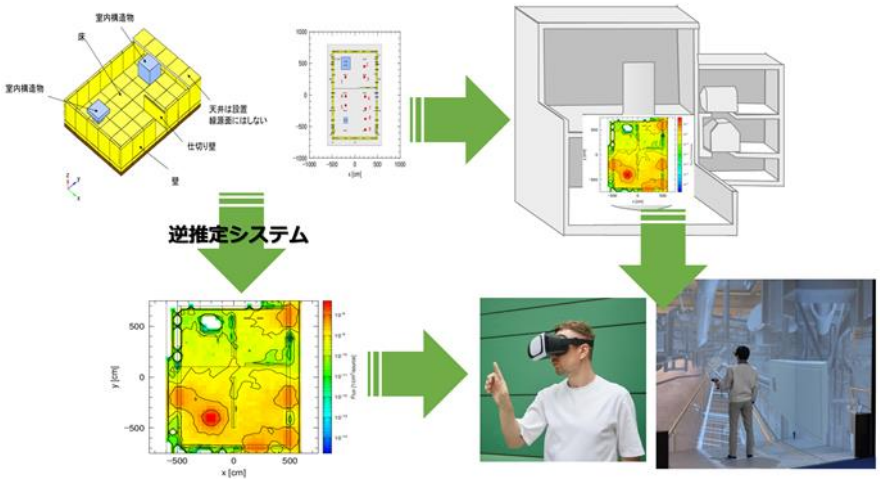


図3.1.1-1 線量分布の逆推定とVRによる可視化

従来の線源分布の推定では、計算対象の空間メッシュと同じ解像度の観測数が必要であり、例えば地表面上の分布を100x100メッシュで計算する場合は10000点の観測データからモンテカルロ輸送計算を行う。しかし1F建屋内部の過酷な環境下では数点～数十点の観測データしか得られない。そこでCCSEでは、スパースモデリングを活用して少ない観測データから線量分布を逆推定する技術を開発した。この技術では、実際の線量分布は高強度線源が支配するという知見を利用して、スパースモデリングの代表的アリゴリズムであるLASSOを導入することにより、少ない観測データからモンテカルロ輸送計算が可能になった。図3.1.1-2にメッシュ数150の室内データに対して18点（12%の情報）の観測データから逆推定した結果を示す。図の左が十分なデータから計算した正解のデータで、右が18点による逆推定の結果である。逆推定が正解のデータをほぼ再現していることが確認できる。

計算結果の3次元線量分布に関して、楢葉遠隔技術センターでVR可視化する技術を開発しており、建屋の3D-CADデータをベースに線量分布を合成して表示し、リアルタイムで作業者に提示するシステムを開発中である。図3.1.1-3に3D-CADデータによる建屋内部のVR可視化の様子を示す。

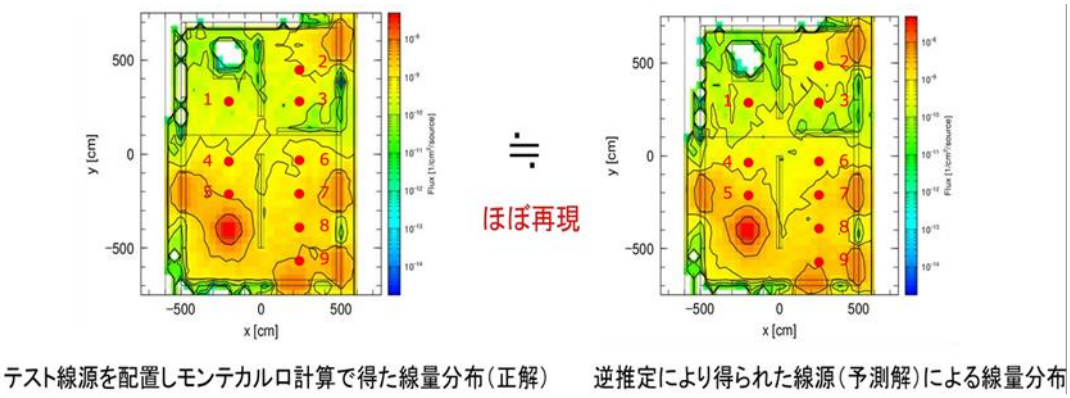


図 3.1.1-2 線量分布の逆推定

計算結果の 3 次元線量分布に関して、楢葉遠隔技術センターで VR 可視化する技術を開発しており、建屋の 3D-CAD データをベースに線量分布を合成して表示し、リアルタイムで作業者に提示するシステムを開発中である。図 3.1.1-3 に 3D-CAD データによる建屋内部の VR 可視化の様子を示す。

図 3.1.1-3 3D-CAD データによる建屋内部の VR 可視化

本計画は経済産業省の令和 3 年度開始廃炉・汚染水対策事業費補助金(原子炉建屋内の環境改善のための技術の開発・被ばく低減のための環境・線源分布のデジタル化　技術の開発　)に係る補助事業」(国プロ)の一部として実施された成果を含む。

3.7 デジタルツインと 5G（Data Science との関連）のまとめ

「5G 時代の可視化技術研究」WG 第 8 回会合（2022 年 3 月 10 日）において、東京理科大学工学部情報工学科 HEXAGON/TUS デジタルツインラボラトリの松尾裕一教授をお招きし、「東京理科大学におけるデジタルツイン関連の研究紹介」というタイトルで講演をいただいた。本講演におけるデジタルツインは、物理資産、プロセス、システム、デバイス等に対してコンピュータ上で構築されたデジタル表現であり、リアルタイム予測、最適化、監視、制御、意思決定等に利用でき、現在幅広い分野での活用が期待されている。従来のコンピュータ・シミュレーションやデジタルモデルとの違い、HEXAGON/TUS デジタルツインラボラトリにおける「ものづくりデジタルツイン」に関する基盤技術の紹介から、点群計測、オンサイト気流解析、機械学習、AR 可視化などの要素技術についても解説をいただいた。詳細については添付資料を参照されたい。

3.8 スマート農業への活用のまとめ

「5G 時代の可視化技術研究」WG 第 9 回会合（2022 年 5 月 11 日）において、山梨大学大学院の茅暁陽教授をお招きし、「AI×スマートグラス×ローカル 5G＝「匠の技」の見える化」というタイトルで講演をいただいた。ここでの匠の技とは、農業における生産者の熟練した技術を指しており、本講演では、ブドウの摘粒作業に AI、スマートグラス、ローカル 5G などの技術を導入した事例を紹介いただいた。提案された手法は、スマートグラスで撮影されているブドウ画像をローカル 5G ネットワークでサーバーに送り、深層学習によってブドウの房を認識して、粒の推定値をスマートグラスにリアルタイムで表示するものである。このシステムによって熟練者ではなくてもスマートグラスをかけることにより摘粒作業を支援することができる。詳細については添付資料を参照されたい。

4. メタバースと可視化技術

メタバース（英: Metaverse）は、コンピュータやコンピュータネットワークの中に構築された、3 次元の仮想空間やそのサービスを指す。定義は文脈によって曖昧で、VR ヘッドマウントディスプレイを被った仮想空間といった狭義のものから、オンラインゲームまで含めた広義のものまである。本章では、ヘッドマウントディスプレイをはじめとする CG 技術を活用した 3 次元の没入視点を用い、複数人でインタラクティブなコミュニケーションが行える仮想空間およびそれを実現するサービス・プラットフォームを指すものとする。

メタバースのアイデアは昔からあり、実際のサービスとしては 1990 年代の「Habitas」「さぱり」「ウルティマオンライン」などから始まり、2000 年代中盤には「SecondLife」が世界的にブームにもなった。ここ数年で急激にサービスとユーザが増加しており、その背景として、VR ヘッドマウントディスプレイなどのデバイスの高性能化・低価格化、その他コンピュータテクノロジー面の進化によるものに加え、VTuber に代表されるアバターコミュニケーションの一般化、COVID-19 パンデミックによるオンラインコミュニケーションへの急進的なシフトなどの社会要因も挙げられる。

ネットワーク技術もまたメタバースの進化を支える要素技術として欠かせないものであり、広帯域低遅延を特徴とする 5G はメタバースへのモバイルアクセス手段として有力なものである。また、コンピュータグラフィックスの技術と切り離せない関係にある。本章ではいくつかのメタバースサービスを比較するとともに、可視化技術の応用先として活用を考える。

4.1　サービス・プラットフォームの比較

Horizon Workspace
Meta 社のメタバースプラットフォーム「Horizon」のうち、会議空間を提供することに特化したものである。
会議室に模した空間内で数人程度のコラボレーションのサービスである。空間内を自由に動き回る事はできず、座席に固定される。（席の移動は可能）
メタバース空間内の黒板や、Oculus Remote Desktop を通じて PC 画面をメタバース空間内に持ち込めるなど、コラボレーションに特化した機能が提供されている。特徴的なのは、ヘッドマウントディスプレイの外部カメラを使用してキーボードと指の位置をメタバース空間内に再現するため、ヘッドマウントディスプレイを被ったまま PC を操作できる。
没入視点は Oculus Quest2 のアプリケーションでのみ利用できる。それ以外の場合は Zoom のような Web アプリケーションから 2D での参加となり、メタバース空間のディスプレイを通じたテレビ会議のようにメタバース空間内とのインタラクションとなる。

図 5-1 Horizon Workspace でのコミュニケーションの様子

VRChat
VRChat.Inc がサービス展開するメタバースプラットフォームである。
自由度が高くエンターテインメント向けのメタバースサービス。現在もっともユーザコミュニティが活発なサービスの 1 つであり、日々さまざまなイベントが行われている。
SDK が提供されており、作成したモデルをインポートして、アバターとして使用する、ワールドを作成することもできる。ただし、自由度が高いぶん利用までのハードルが高い。またユーザのレベル設定があり、初心者は利用できる機能に制限がある。

Spatial
Spatial Systems サービス展開するメタバースプラットフォームである。
外部でデザインした 3D モデルをインポートする、360 度カメラの写真を空間背景に設定するといったことができるので、独自の空間や、現実世界のある場所を再現した空間を作ることができる。
ワークスペースには個別の URL が発行されるため、空間内にハイパーリンクを設置することでスペース間を自由に移動できる。このような機能を利用してショールームを設ける、仮想的な旅行体験を提供する、といったことも行われている。
　イーサリアムの NFT に対応しており、NFT が付与されたデジタルコンテンツの展示や取引もメタバース空間内で行える。
　アバターはシステムで用意したパーツを組み合わせ、その際に顔写真を使ってリアリティある顔をつくることができる。また、Ready Player Me といったサービスで作成したアバターをインポートできる。
Oculus Quest2 などの専用アプリおよび WebXR に対応した Web ブラウザで利用できる。


図 5-2 Spatial でのコミュニケーションの様子

Hubs
Mozilla プロジェクトで開発している OSS のメタバースプラットフォームである。利用にあたっては、Hubs をホスティングするサーバーが必要となる。Hubs を用いたメタバースプラットフォームサービスとしては、Hubs Cloud AWS、CYZY SPACE などがある。一般利用者が自身で Hubs をホスティングするサーバーを構築することも可能である。
WebXR に対応した Web ブラウザで利用できる。専用アプリはなく、スタンドアロンの VR ヘッドマウントディスプレイでも Web ブラウザを用いる。

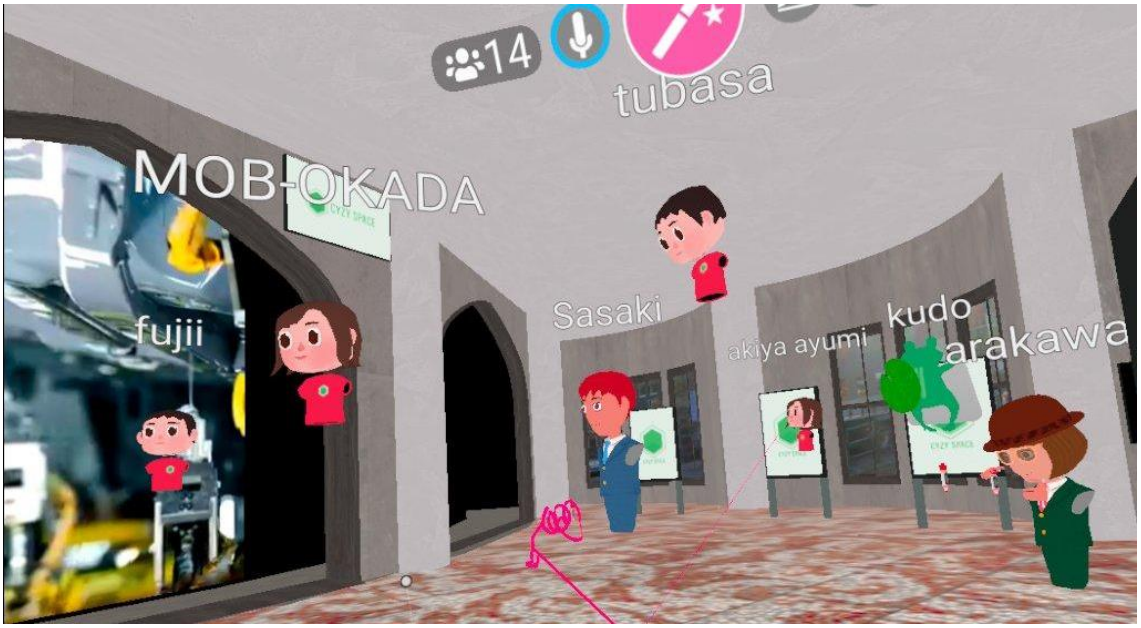


図 5-3 Hubs(CYZY SPACE)でのコミュニケーションの様子

4.2　メタバースへの可視化技術の応用

SS 研「汎用 VR システムの活用研究 WG」では、科学的可視化に VR システムを用いることについて報告している。VisAsset は VR システム上に可視化ソフトウェアの機能を実現している。

これらのアイデアを発展させ、メタバースのテクノロジーを応用することで、単に可視化結果を没入視点で立体視することに留まらず、複数の視点から同一の可視化結果を表示してコミュニケーションを交えながら視ることができそうである。また、メタバース空間内により高度かつリアリティある演出を組み入れてアミューズメント性を発展させることも考えられる。

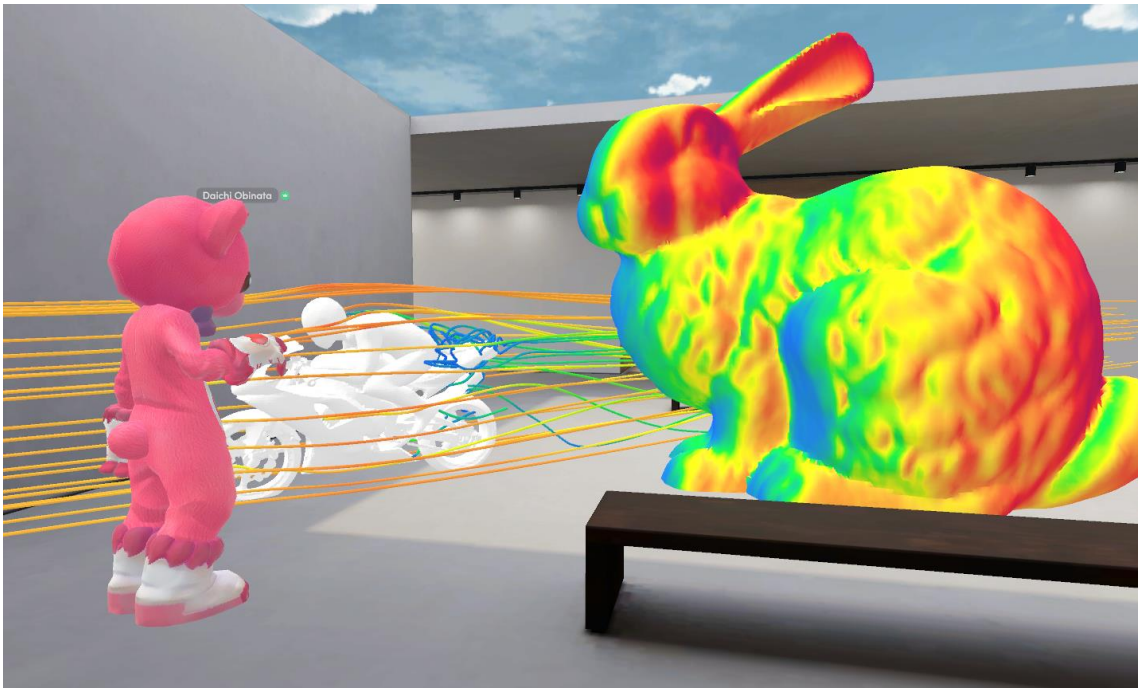


図 5-4 Spatial のメタバース空間内で可視化結果をインポートした例

例えば図 5-4 は、Spatial のメタバース空間に ParaView での可視化結果をインポートした例である。ParaView で可視化して得られた 3 次元形状を glTF 形式で出力し、Spatial の 3D モデルインポート機能でインポート・配置した。glTF にエクスポートさえできれば、特別なアプリケーションソフトウェアを必要とせずメタバース空間内に持ち込めるため、科学可視化の結果を複数人で、没入視点で見ることも容易である。
しかしこの手法には大きく 2 つの解決すべき点がある。

1. メタバース空間内で直接可視化処理ができない。外部の可視化システムで可視化した結果をコンテンツ化してメタバース空間に持ち込んでいるため、可視化のプロットを変える場合の必要操作が多く、ステアリング性が損なわれる。
   事前にシナリオ化されたコンテンツデータを用意できる場合（例えば展示のようなもの）には有効であるが、デザインレビューのように多角的に検討・解析が必要な議論には用いることができない。
2. VR ヘッドセット程度の計算リソース（GPU、メモリ）では、シミュレーションの高解像度データを可視化処理できない。仮にメタバース空間内で可視化操作が行えたとしても、そもそも HPC のような大規模な計算システムで計算するような規模のデータは扱うことができない。

1. については、VisAsset[1]のように、（メタバース空間を表示する）VR システムに可視化機能を実装する方法が考えられる。2. についてはリモートレンダリングが有用であり、3.3 章で示した MEC のようなエッジサーバ上に、メタバースのクライアントプログラムとリモートレンダリングシステムを実装することが考えられる。
これらの方式と HPC システムを結合するには、in-situ 可視化や in-transit 可視化[2]の技術が有用であると考えられる。in-situ 可視化は、HPC システム上での計算（CFD など）時に同一計算機上で可視化

まで行う方式で、データの転送やファイルアクセスを軽減する。in-transit 可視化は複数の計算機リソースの間でデータステージングしながら、可視化パイプラインを構成する方式である。

参考文献

[1] VisAsset プロジェクト http://www.comm.tcu.ac.jp/miyachilab/visassets/index.html
[2] In Situ/In Transit アプローチを用いた大規模数値解析におけるポスト処理効率化 https://www2.nagare.or.jp/cfd/cfd33/CFD33_paper_web/paper/B12-1.pdf

謝辞

メタバースサービスの比較は、富士通メタバースコミュニティに協力いただいた。

5. まとめ

本 WG では、スマートデバイスなどに用いられる次世代移動体通信規格「第 5 世代移動通信システム」（5G）について、現状の技術、サービス内容、活用事例、今後の動向を調査し、5G 技術を可視化環境におけるデータ転送手段として適用する際の可能性と課題を報告書としてまとめた。

科学技術・教育分野における可視化において、ハードウェア面では旧来の PC だけでなくスマートデバイスや HMD などの VR 機器、またソフトウェア面ではそれらの多様な機器に対応したマルチプラットフォームな開発基盤が、目的を達成するのに十分な水準に達しており、これを検証するためのプロトタイピングも前 WG（汎用 VR システムの活用研究 WG）で行った。しかしこのような環境においては、各デバイス間やプロセス間でデータを移動する通信部分がボトルネックになるため、本 WG ではそれを解消するための技術として 5G に期待し、その特徴と課題、活用事例について、多くの講師の先生方に話題提供を頂き、可視化における 5G の活用案に関する議論を行った。また並行して、前 WG でゲーム開発基盤を用いてプロトタイピングした可視化環境を 5G 環境での検証に供するため"VisAssets"として整備し公開した。

現状の 5G の特徴とサービス内容についての調査を通じて、既存の通信インフラを単純に 5G に置き換えればメリットが享受できるものではなく、現状の 5G のサービス内容とコストを考慮した上でその特徴を生かした活用方法を検討する必要があることが明らかになった。例えば、5G 環境の利用基盤として、キャリア 5G とローカル 5G があり、公衆網としてサービス提供されるキャリア 5G は手軽に利用できるが、未だサービスエリアが限られ、5G 固有の機能の利用も未だ制限されている部分がある。ローカル 5G は事業所単位やキャンパス単位など、ある程度広いエリアに対して 5G 固有の機能を提供できるが、基地局開設にあたり免許が必用な上に初期設備投資コストが高く、現状では導入のハードルが高い。しかしこれらの課題は規制緩和や技術の進展により今後大きく変化することが予見されるので、継続して動向を注視して行く必要がある。

5G 技術の活用事例については、工場での生産管理、デジタルツイン、農業での収穫作業支援、cloud レンダリング、リモート会議など、幅広い分野での調査を行った。これらの事例に共通する 5G ならではのメリットとしては「光ファイバーに匹敵する性能のデータ通信網を、光ファイバーの敷設よりも手軽に、ワイヤレスで構築できる」という点に集約されることが明らかになった。実際、既設の工場や農地に光ファイバー網を新たに敷設するのは困難だが、5G であれば実現のハードルは低くなる。また、VizAsset のような可視化環境に対して現状の 5G サービスを有効活用できるケースとして、VR ゴーグルなどの 3D 表示デバイスを用いる場合、従来はこれをローカルの高性能グラフィックスワークステーションにケーブル接続できる範囲内で利用する必要があった。しかし、cloud レンダリングサーバーへ 5G で接続すればケーブルもローカルワークステーションも不要で 3D 表示デバイスを利用でき、VR によるプレゼンテーションを行う際の環境上の制約を取り払えることが期待できる。

このように、可視化での 5G の活用にあたっては、手軽に敷設できる高速通信網のワイヤレス利用という 5G の本質的なメリットを踏まえた上で、今後の技術動向と、利用時点でのサービス内容とコストとのバランスを考慮しながら最適な応用方法を検討することが重要である。そのヒントを見い出すための一助として本報告書を活用頂くことを期待する。

最後に、本 WG 活動で講師としてご発表いただいた先生方、ご協力いただいた皆様に感謝いたします。

（5G 時代の可視化技術研究 WG まとめ役 大谷寛明、小笠温滋）