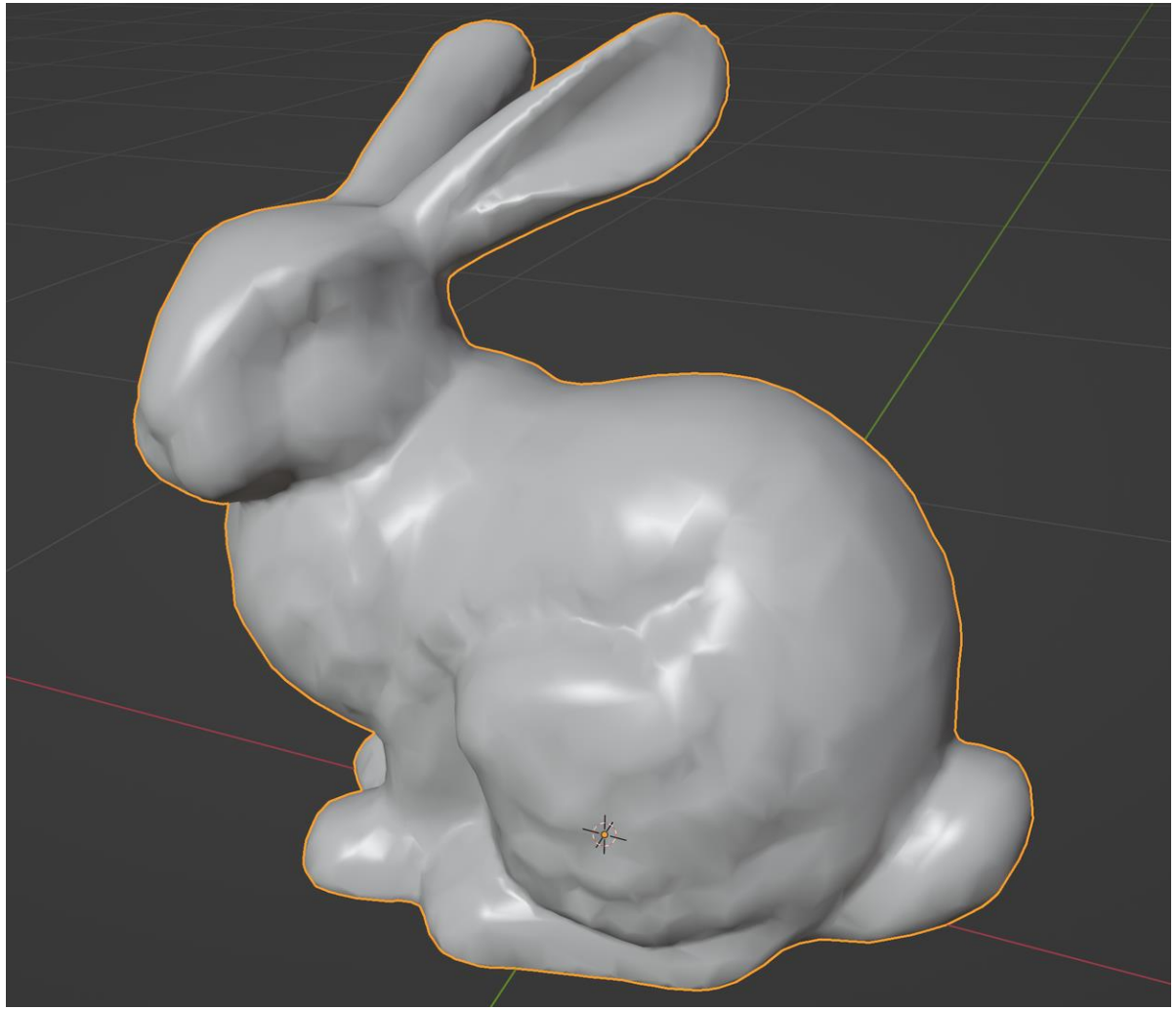
Unity のメッシュ

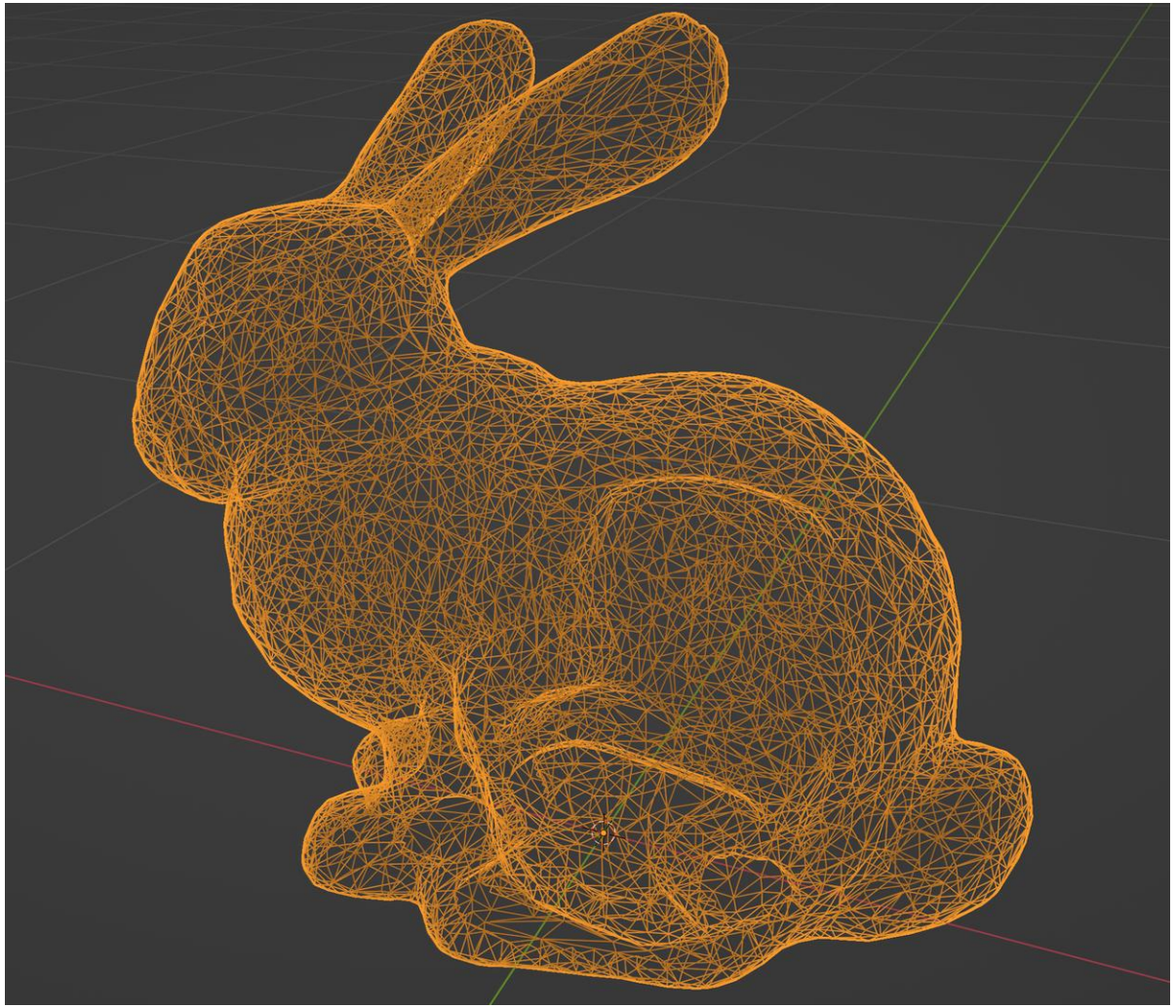
メディアデザインセミナー2B

メッシュというデータ

すべての形状はメッシュで表される

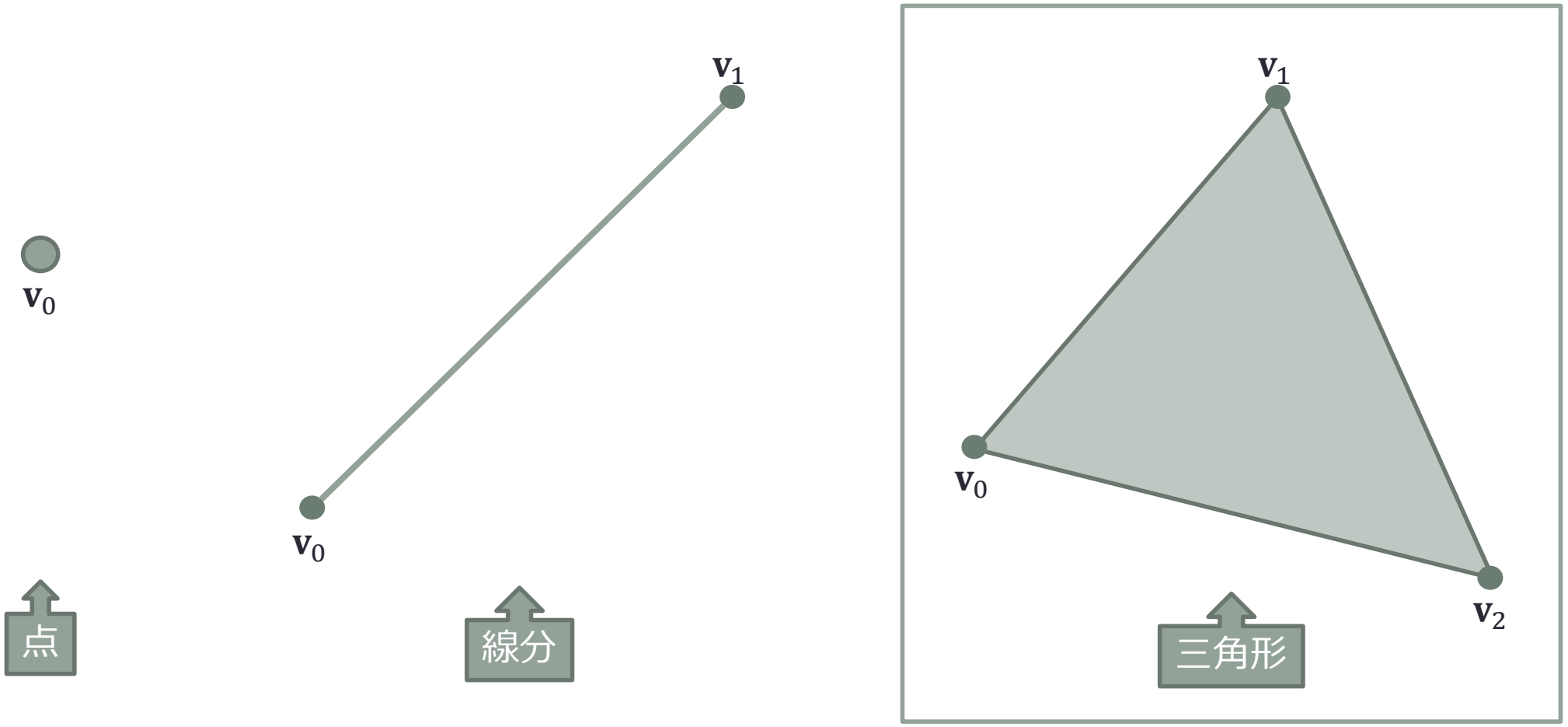
すべてのオブジェクトはメッシュで作られている



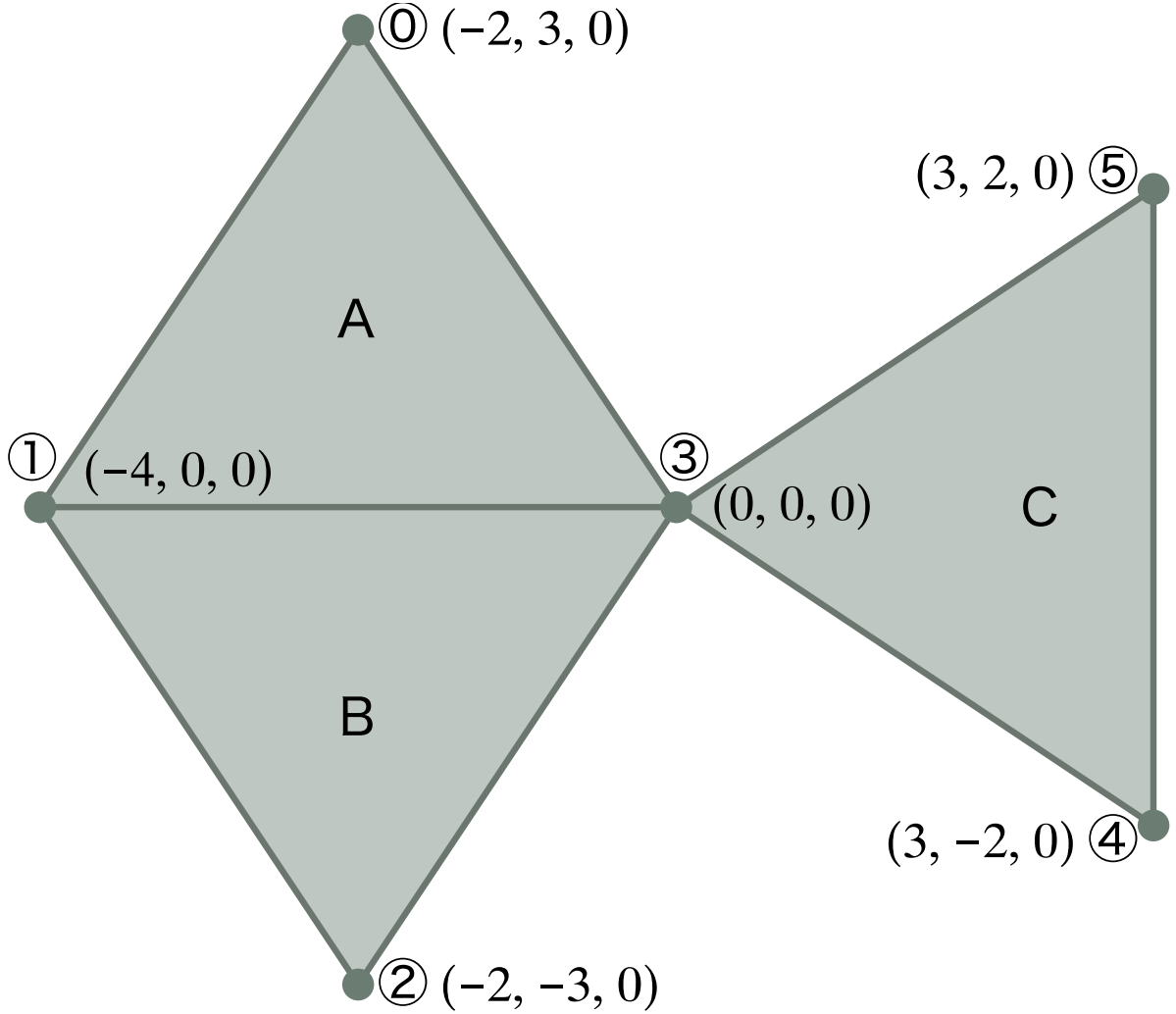


Unity のメッシュは三角形の集合体である

- 多くのグラフィックスハードウェアは点・線分・三角形しか描けない



図形は頂点アトリビュートと頂点インデックスで表される



- 頂点アトリビュート
  - 位置や法線など個々の頂点が持つデータ
  - 頂点属性

頂点番号	位置		
	x	y	z
⓪	−2	3	0
①	−4	0	0
②	−2	−3	0
③	0	0	0
④	3	−2	0
⑤	3	2	0

- 頂点インデックス
  - 三角形を構成する頂点の番号の組み合わせ

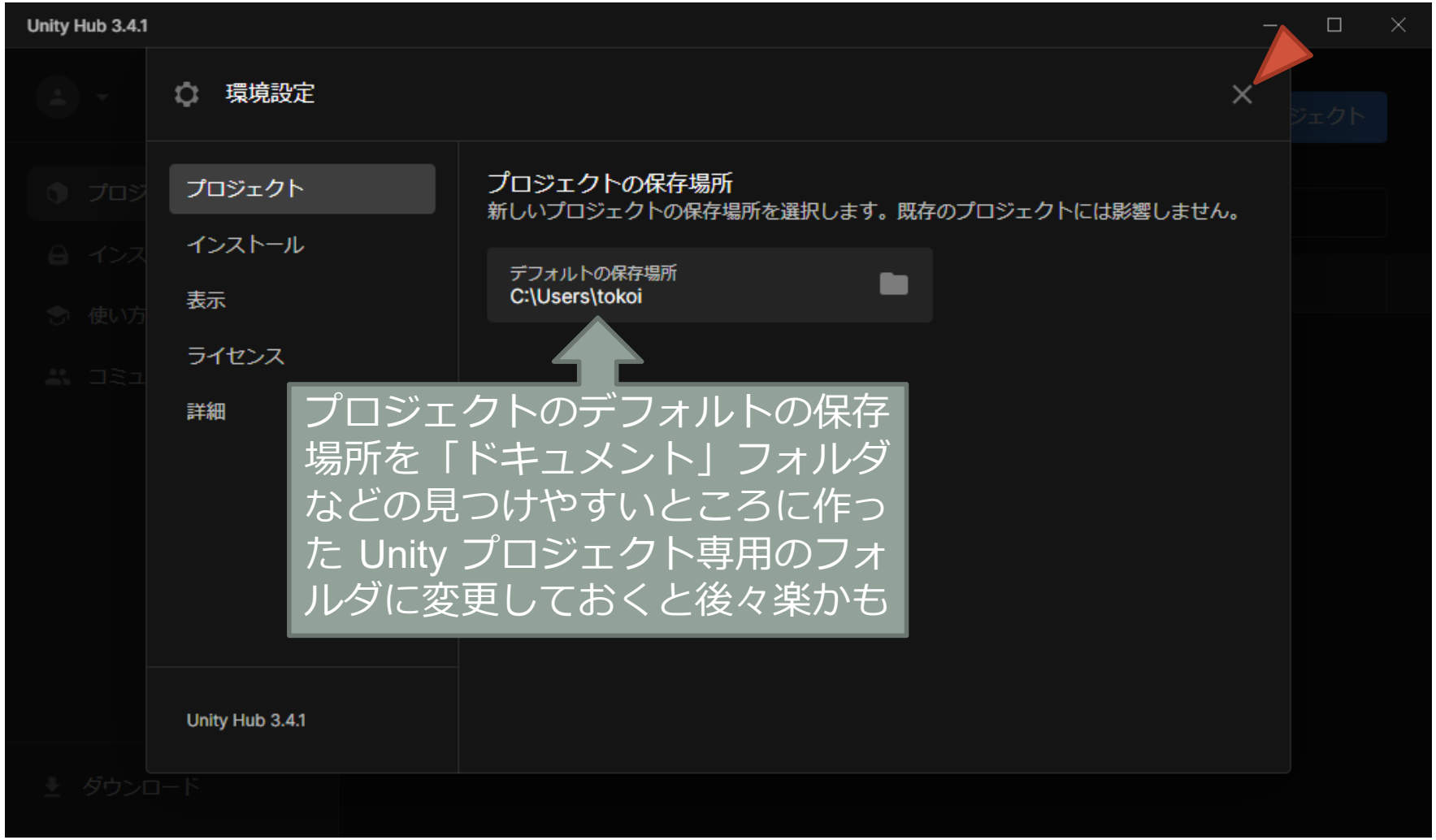
三角形	A			B			C		
頂点番号	⓪	③	①	①	③	②	③	⑤	④

Unity プロジェクトを作成する

Empty 1 個のシーン

Unity Hub を起動して環境設定をクリック

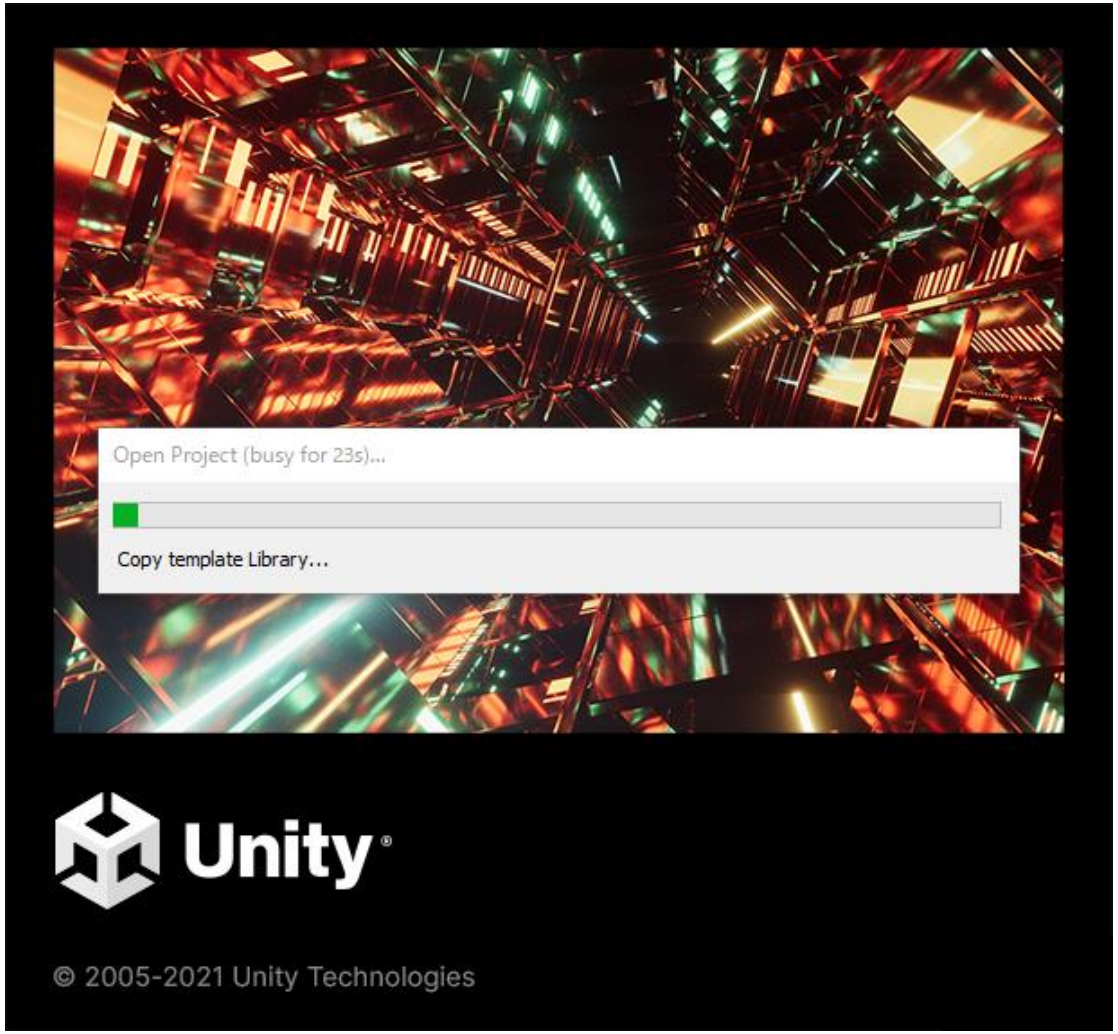
プロジェクトの保存場所を決める



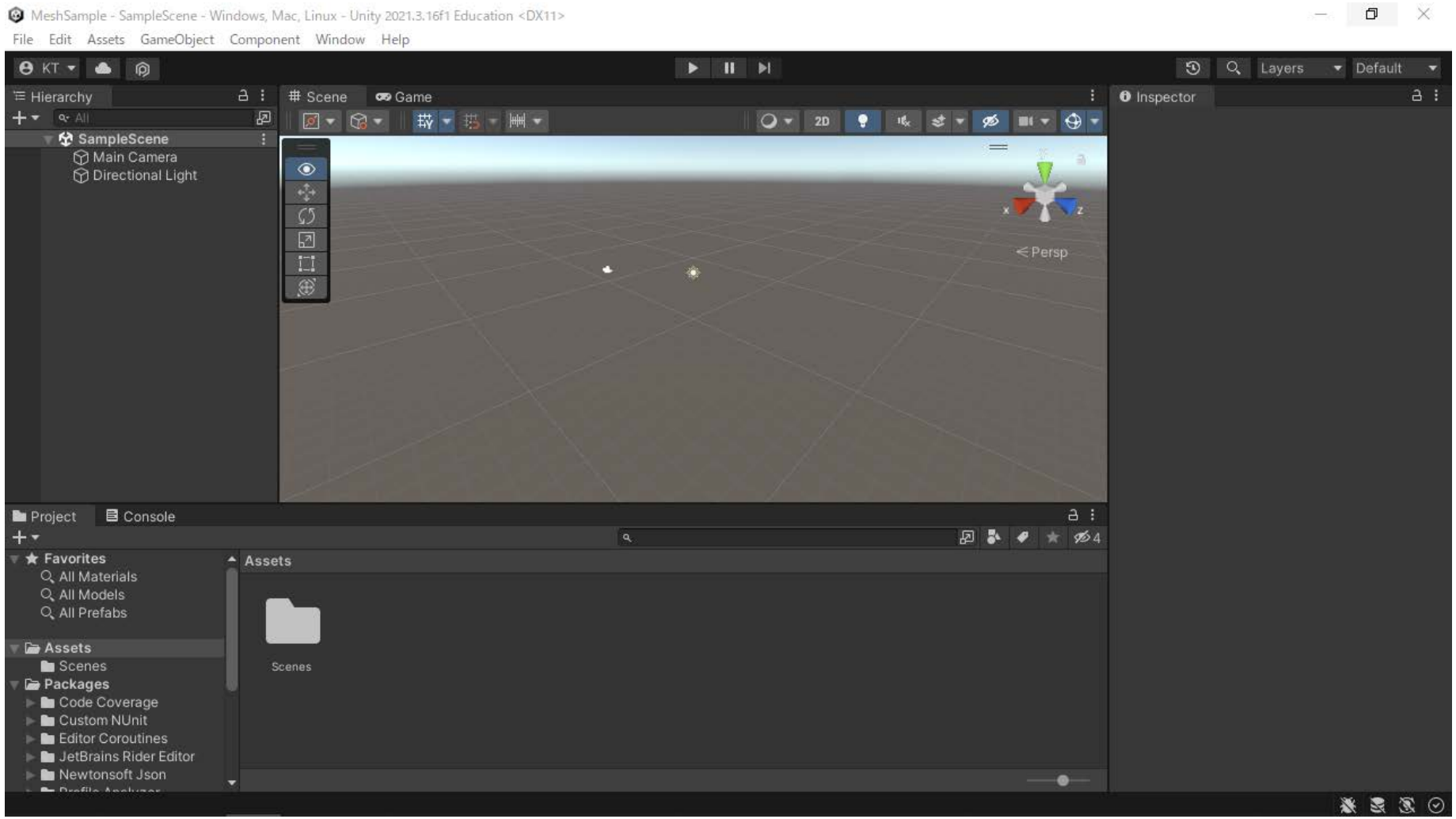
新しいプロジェクトを作成する

3D のテンプレートを選んでプロジェクトを作成する

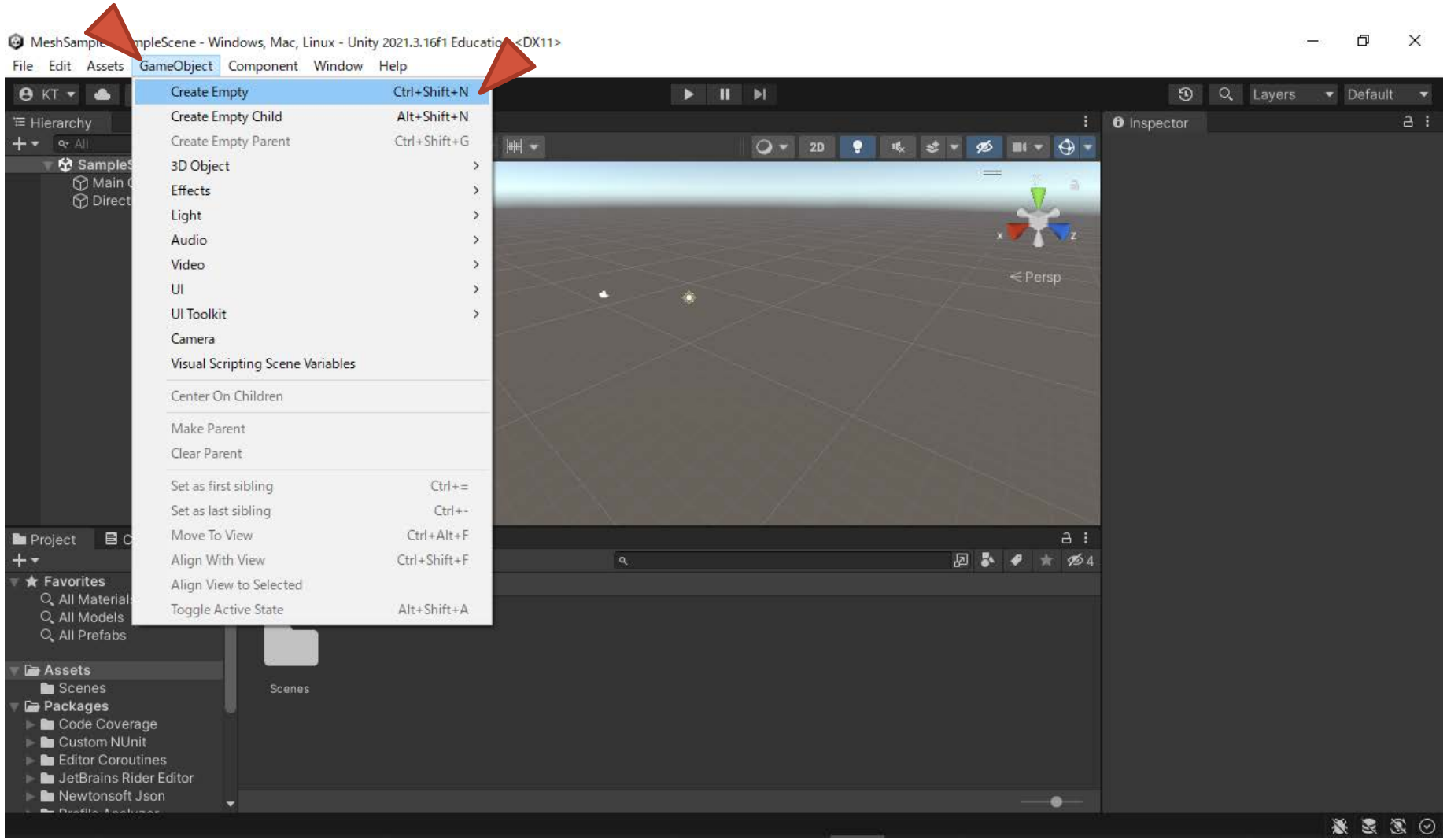
Unity エディタの起動中



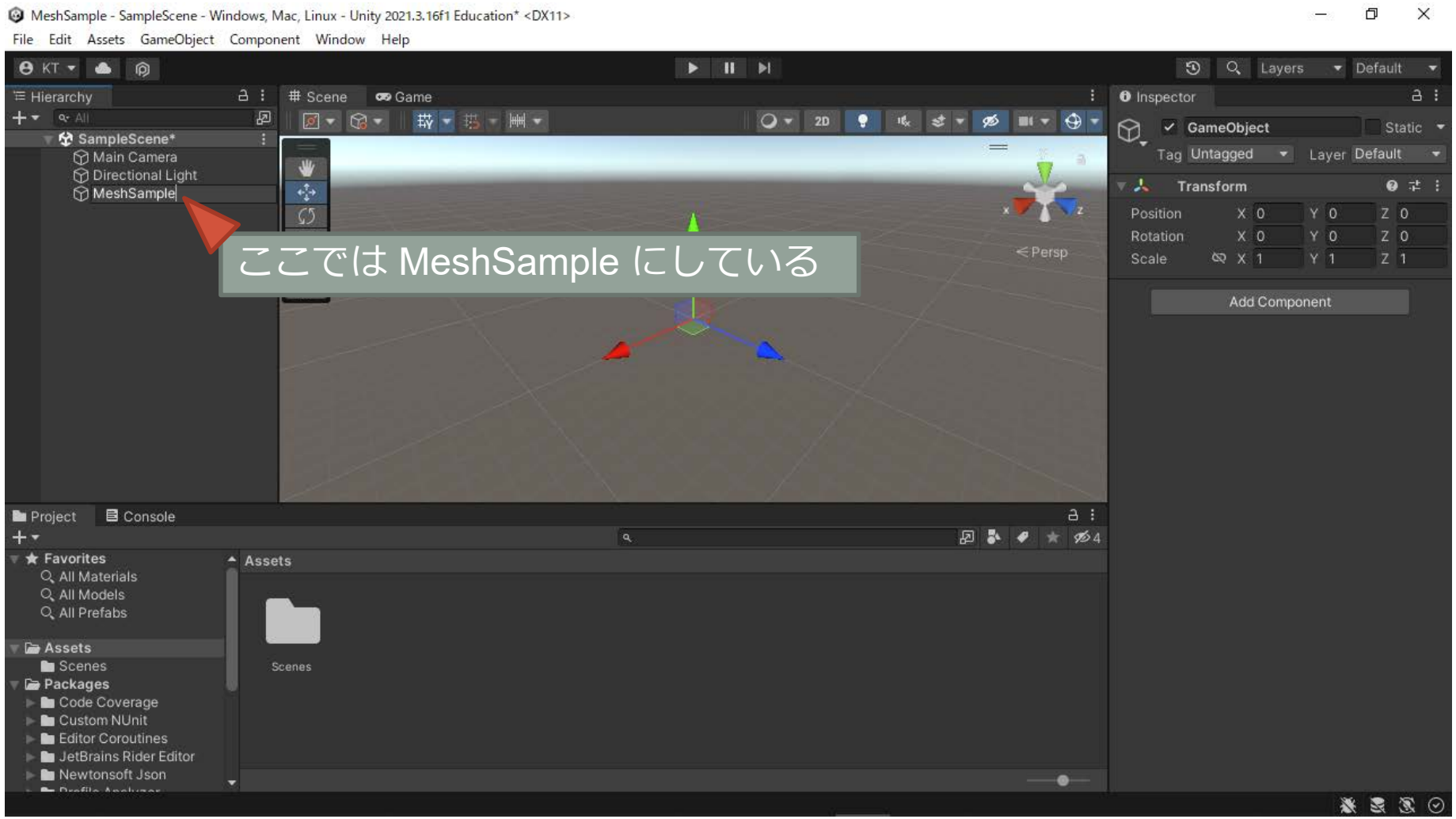
起動直後の Unity エディタのウィンドウ



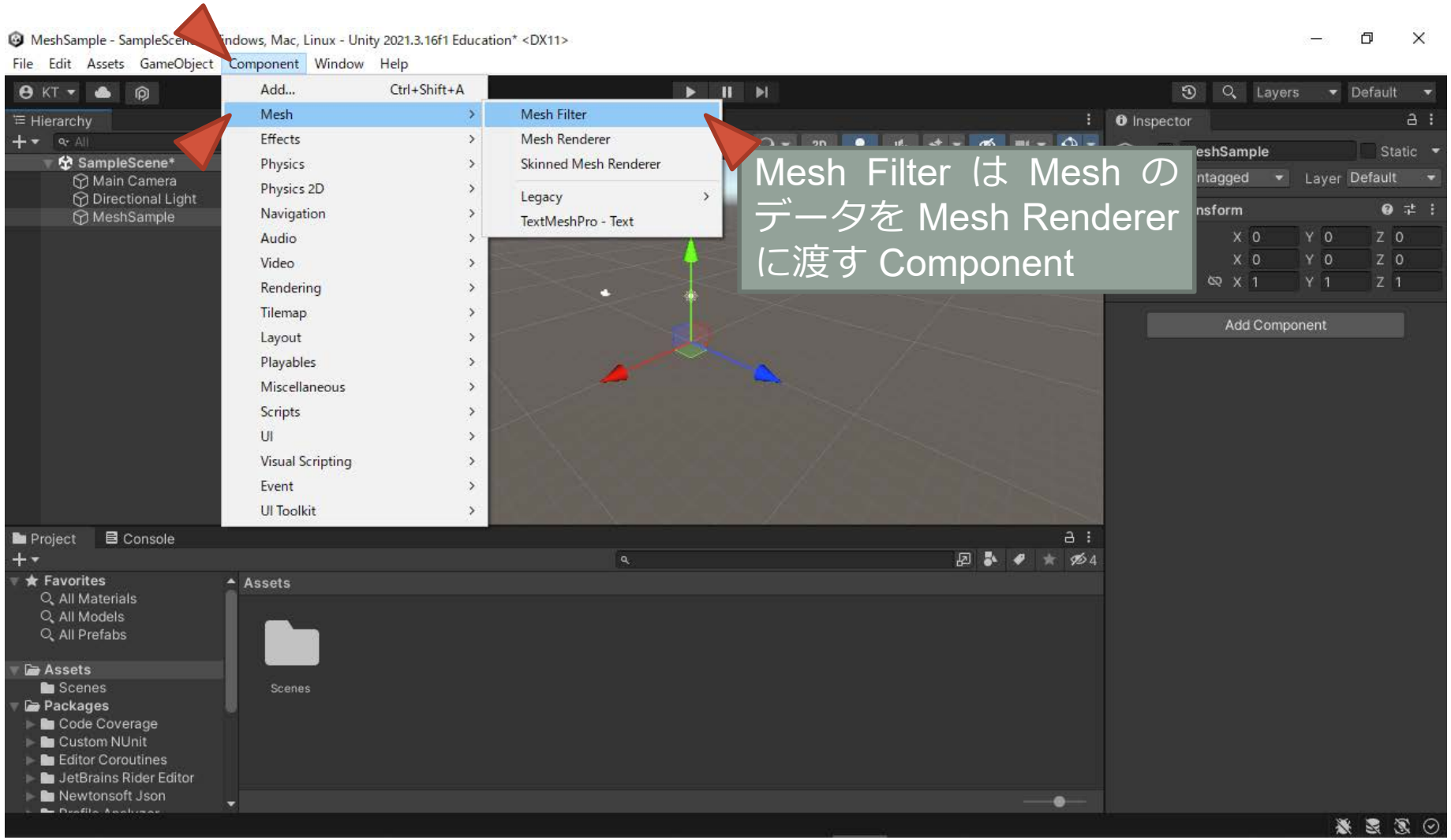
Empty の GameObject を作る



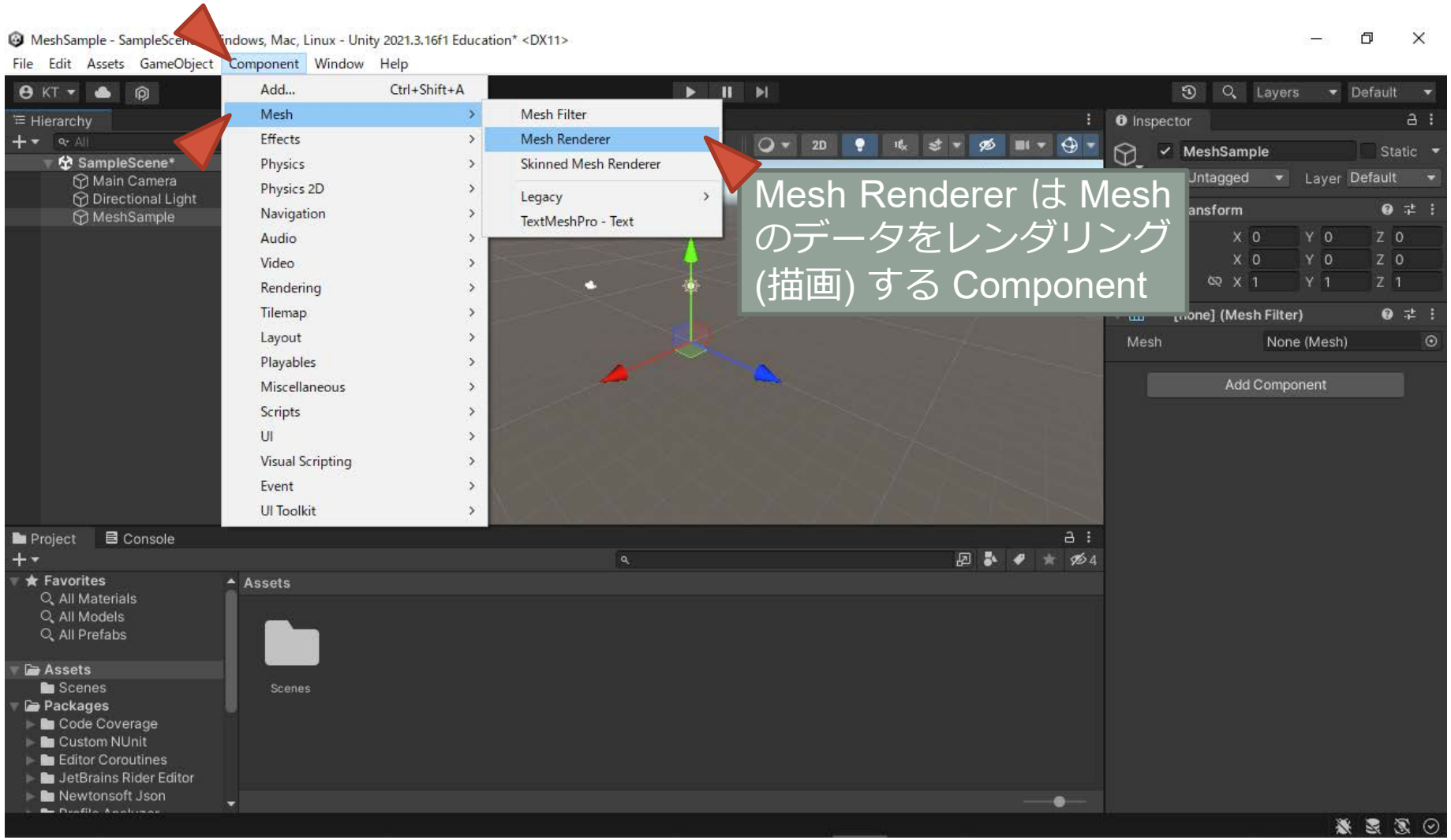
作成した GameObject の名前を変更する



GameObject に Mesh Filter の Component を追加する



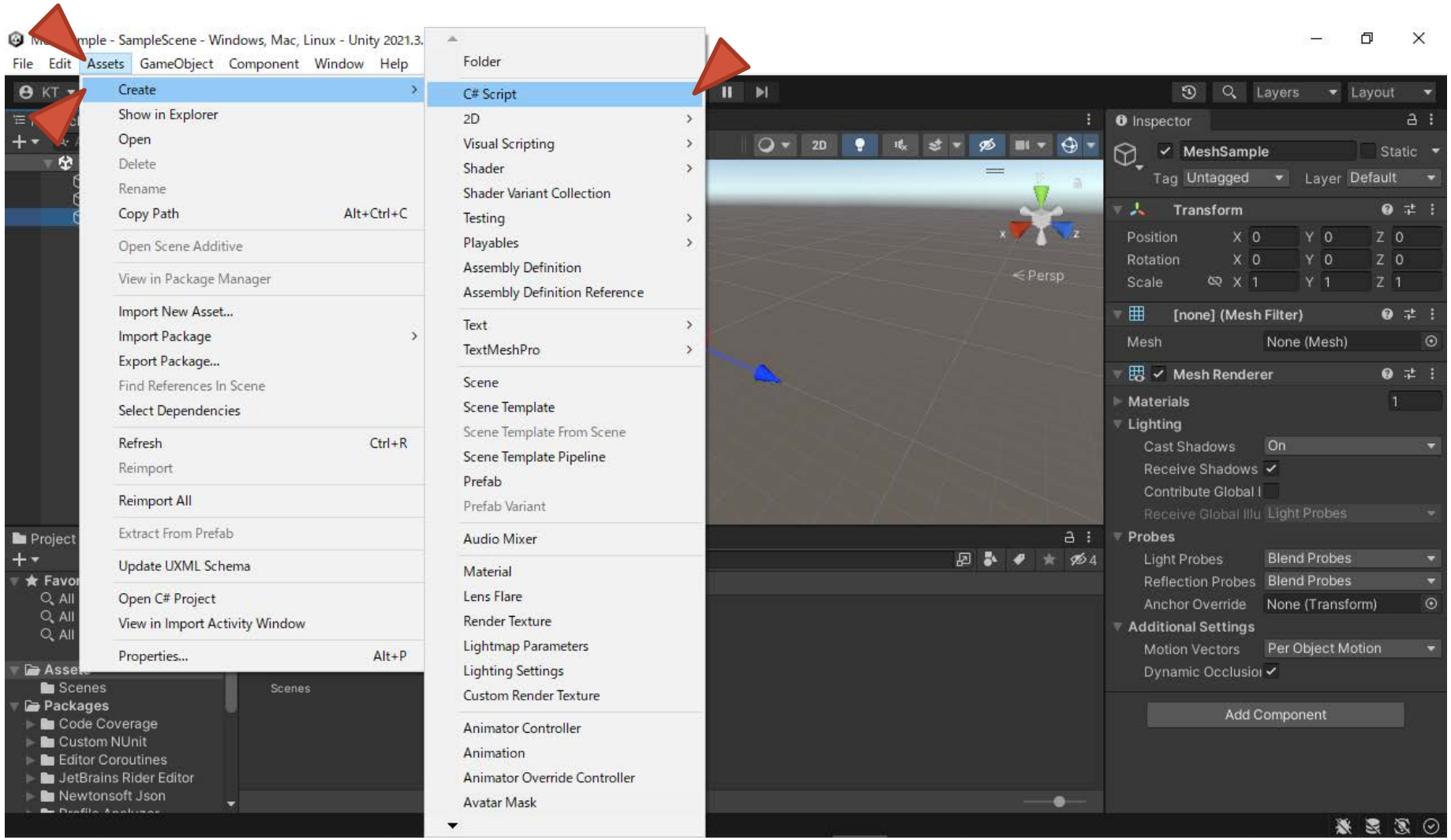
さらに Mesh Renderer の Component を追加する



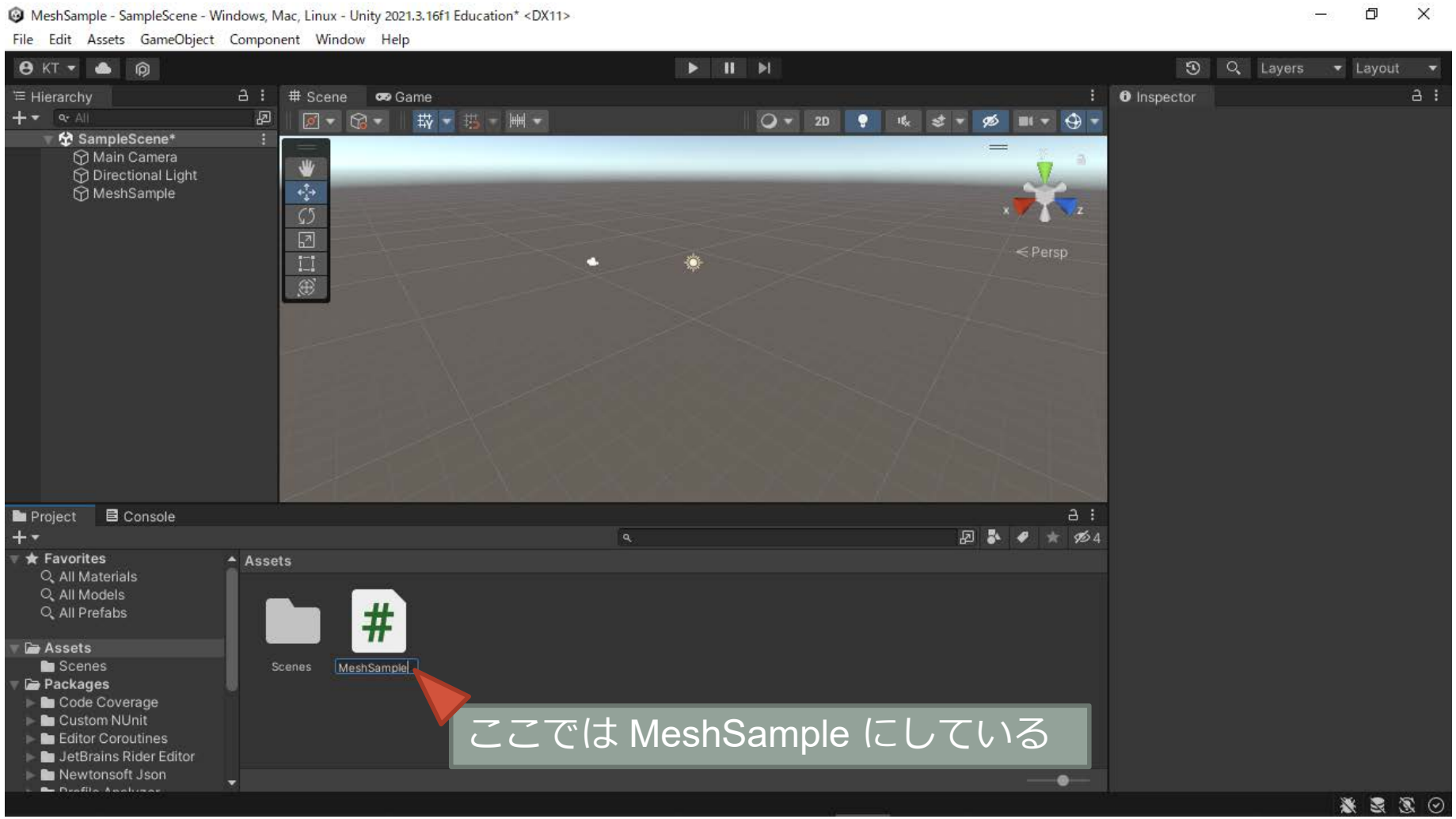
スクリプトで Mesh を作る

頂点アトリビュートと頂点インデックスの作成

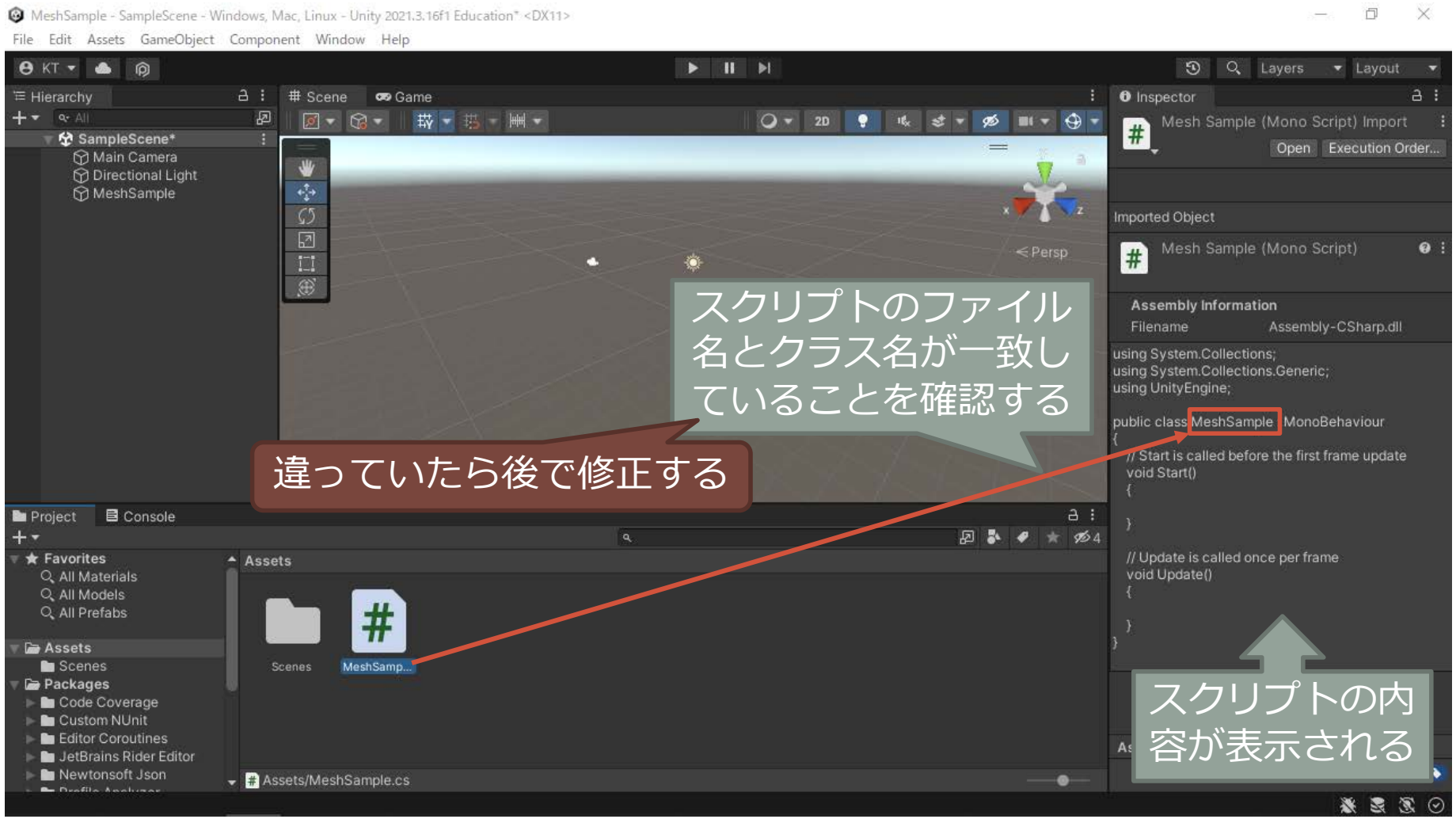
C# スクリプトを作成する



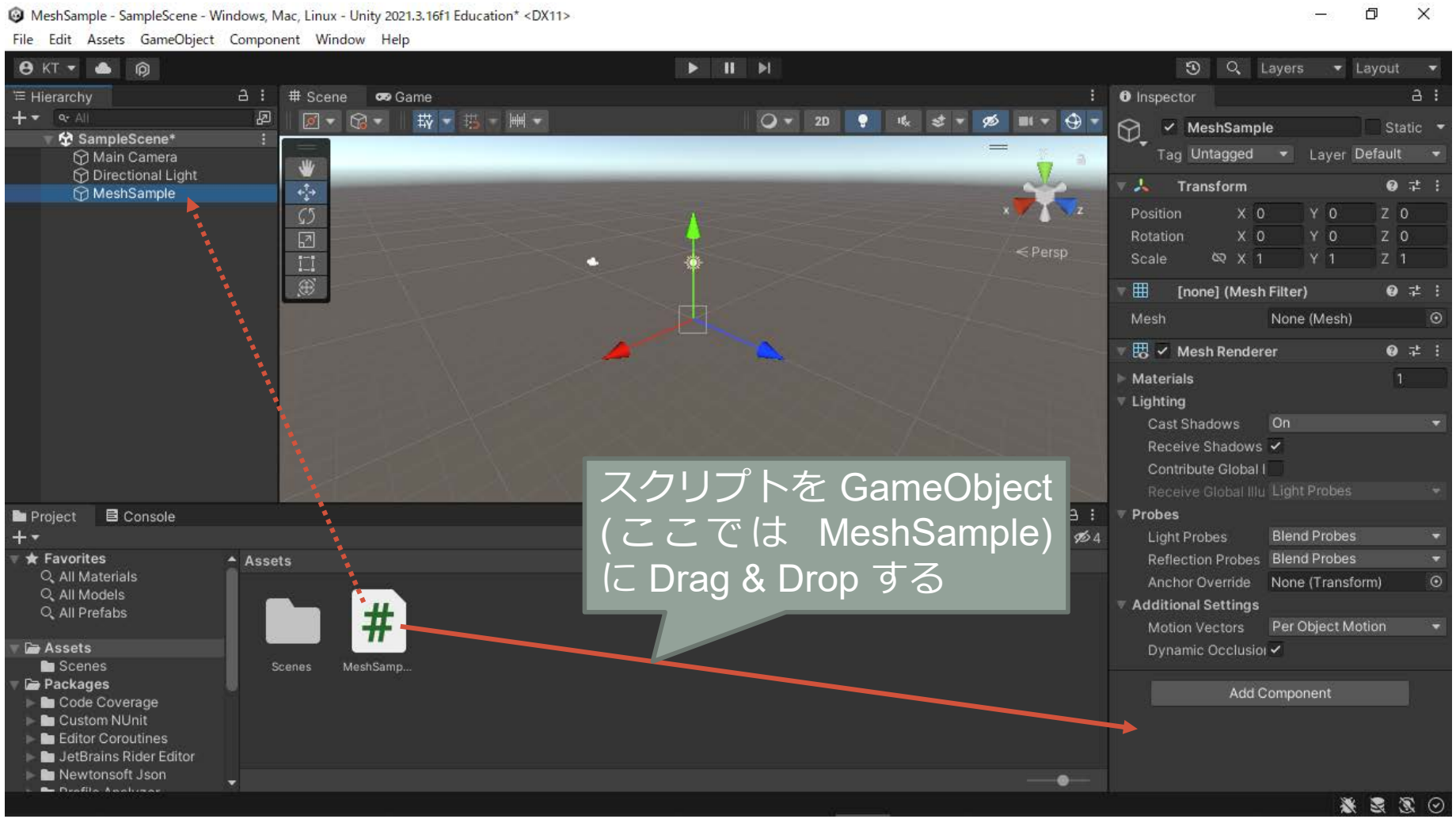
作成した C# スクリプトの名前を変更する



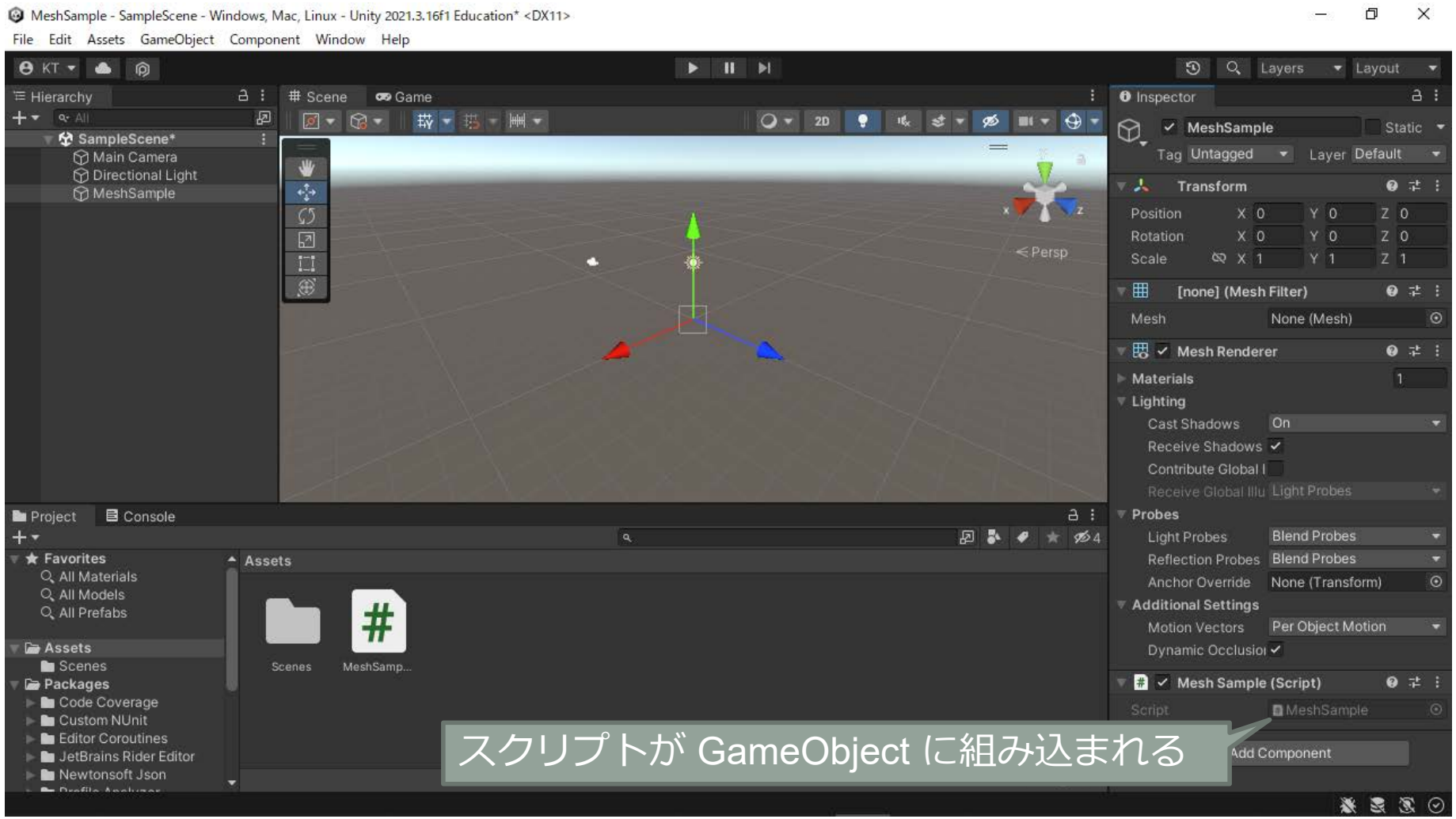
C# スクリプトのファイル名がクラス名になる



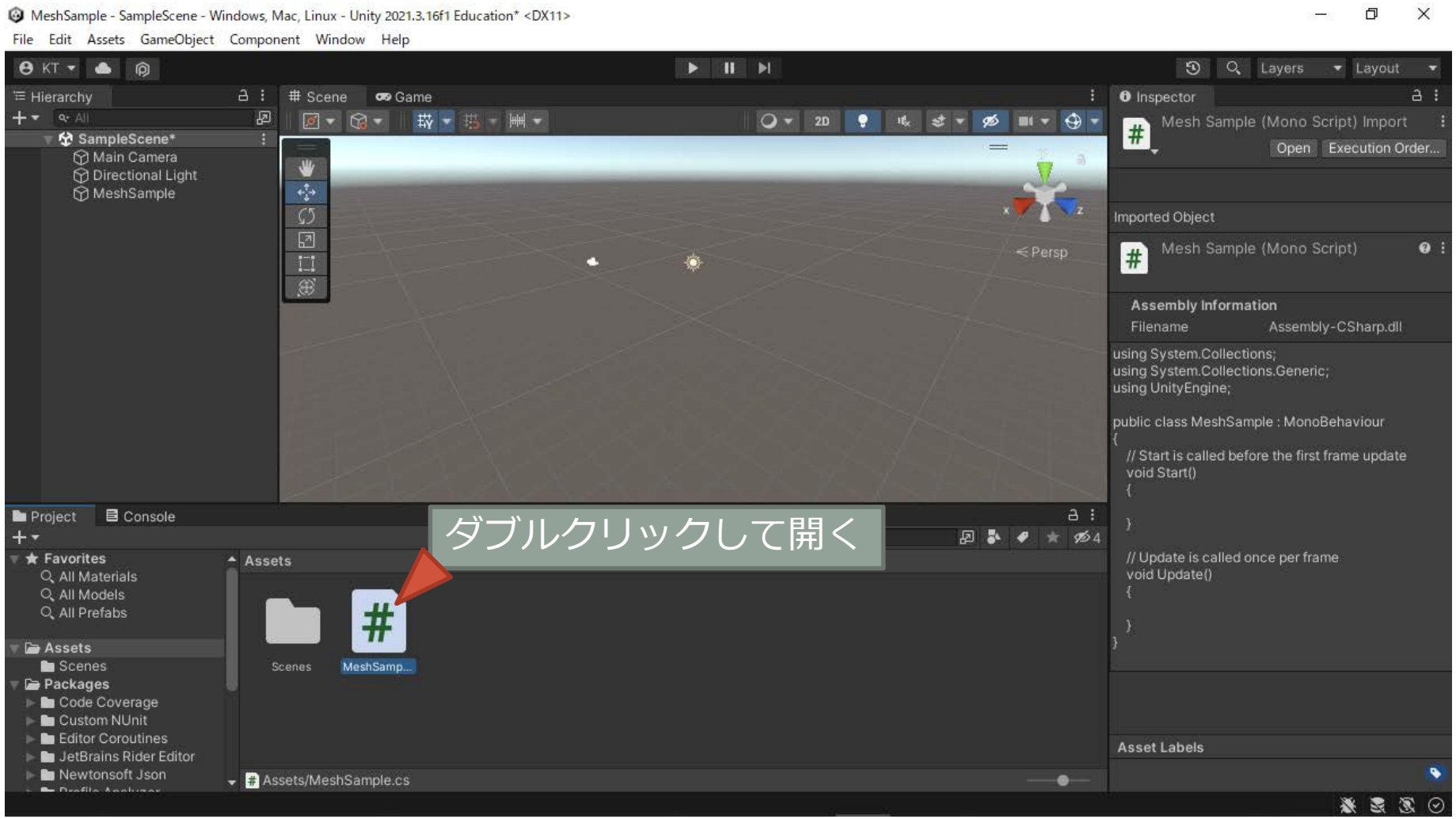
C# スクリプトを GameObject に組み込む



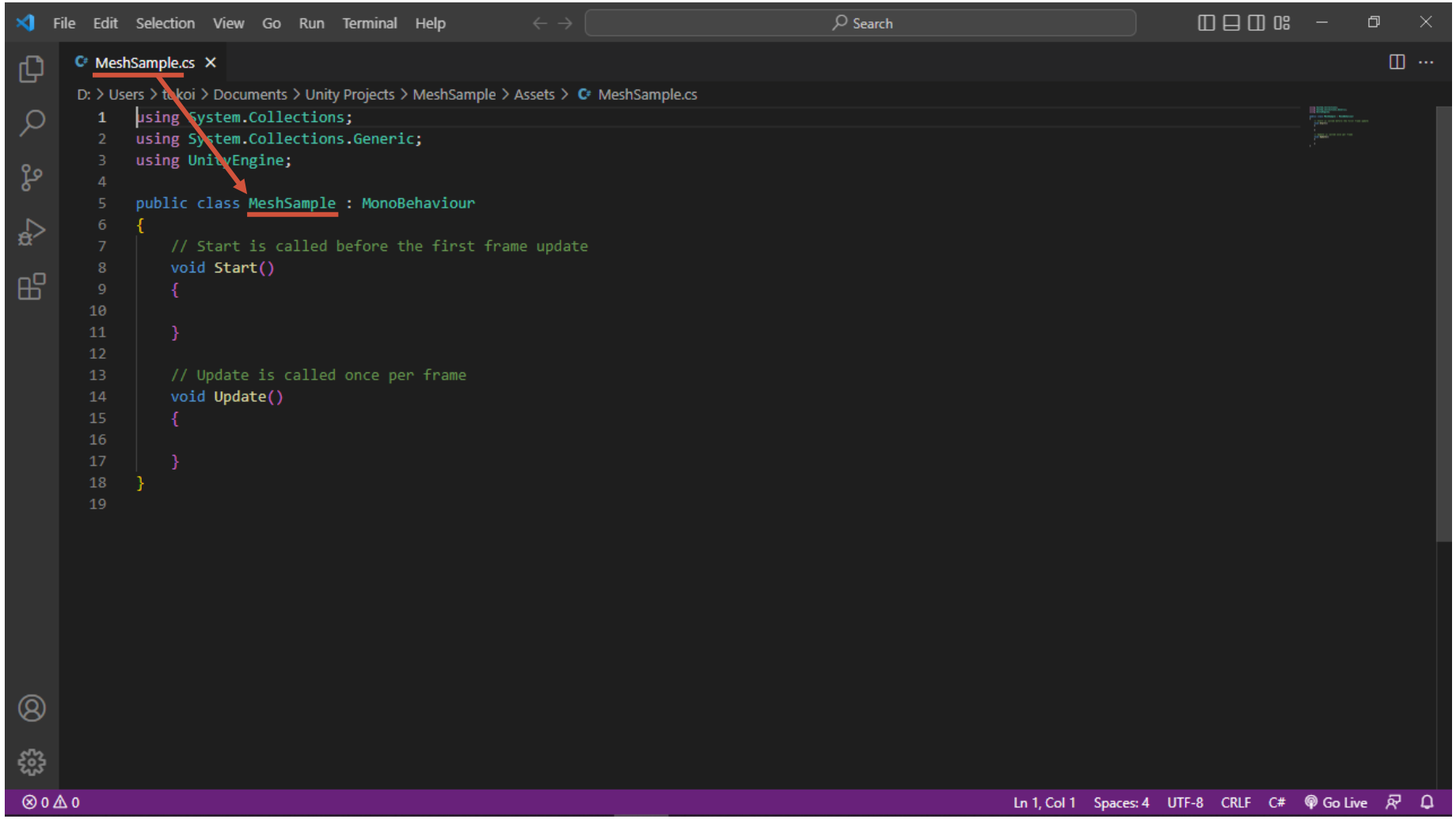
C# スクリプトが GameObject に組み込まれる



C# スクリプトをテキストエディタで編集する

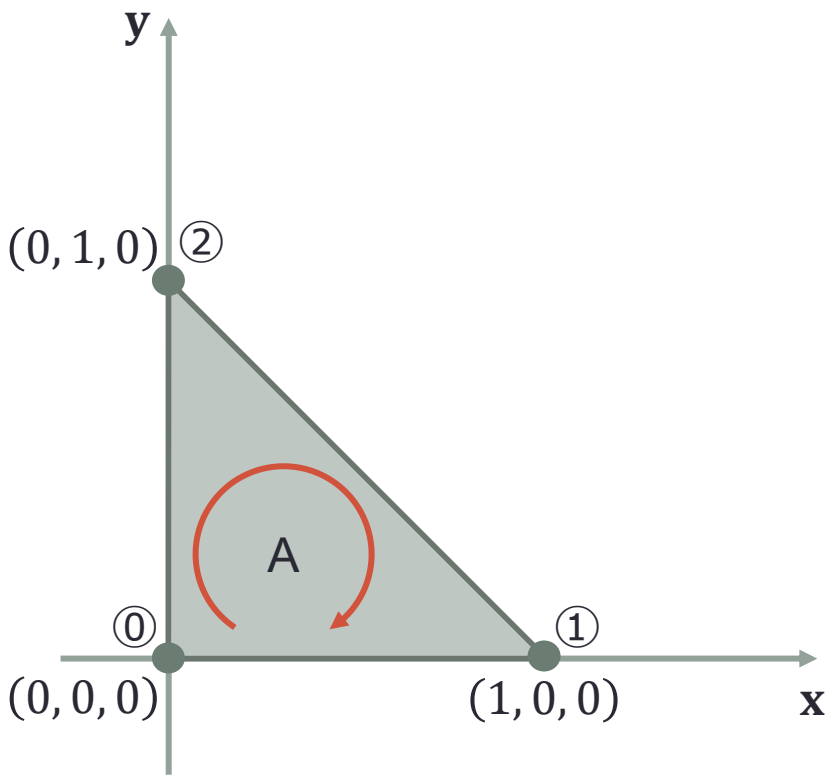


クラス名が C# スクリプト名と一致していなければ修正する



三角形をデータ化する

- １つの三角形は3つの頂点の位置などの**データ**（属性）と頂点の**番号**で表される



頂点アトリビュート

頂点番号	位置		
	x	y	z
⓪	0	0	0
①	1	0	0
②	0	1	0

頂点インデックス

三角形	頂点番号		
A	⓪	②	①

右回り

頂点インデックスの順序は
三角形を表から見たときに
右回りになるようにとる

頂点アトリビュートに設定する位置のデータを準備する

- 頂点のデータ
  - Vector3
    - 3つの実数値 (float) を要素にもつベクトル
  - Vector3[] myVertices = new Vector3[3];
    - 3つの Vector3 型の値を要素に持つ配列を生成 (new) して myVertices から参照する
  - myVertices[0].Set(0, 0, 0);
    - myVertices の 0 番目から参照しているVector3 型の値に (0, 0, 0) を設定する

```
myVertices[0] ⟶ Vector3(0, 0, 0)
myVertices[1] ⟶ Vector3(1, 0, 0)
myVertices[2] ⟶ Vector3(0, 1, 0)
```

頂点番号	位置		
	x	y	z
⓪	0	0	0
①	1	0	0
②	0	1	0

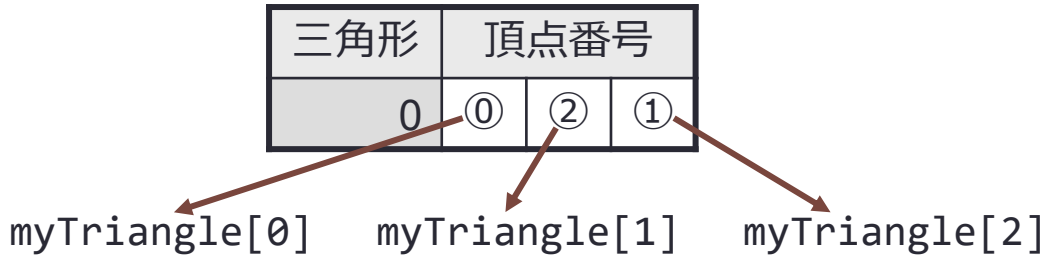
```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class MeshSample : MonoBehapiour
{
    // Start is called before the first frame update
    void Start()
    {
        // 三角形が１つなので頂点の位置のデータの数は 3
        Vector3[] myVertices = new Vector3[3];

        // 三角形の頂点の位置
        myVertices[0].Set(0, 0, 0);
        myVertices[1].Set(1, 0, 0);
        myVertices[2].Set(0, 1, 0);
```

頂点インデックス

頂点アトリビュート (位置)

三角形の頂点インデックスを設定する

- 三角形１つを表す頂点のインデックス
  - int[] myTriangles = new int[3];
    - 3つの整数値 (int) を要素にもつ配列を生成して myTriangles から参照する
  - myTriangles[0] = 0;
    - 三角形の 0 番目の頂点に頂点インデックス 0（頂点番号⓪）を代入する




```
// 三角形が１つなので必要な頂点の番号の数は 3
int[] myTriangles = new int[3];

// 三角形の頂点番号は 0, 2, 1
myTriangles[0] = 0;
myTriangles[1] = 2;
myTriangles[2] = 1;
```

頂点インデックス

メッシュを作成して頂点の位置と三角形のデータを設定する

- メッシュ myMesh を生成する
- 頂点の位置のデータ myVertices を myMesh に設定する

頂点番号	位置		
	x	y	z
⓪	0	0	0
①	1	0	0
②	0	1	0

- 三角形のデータ（頂点番号）myTriangles を myMesh に設定する

三角形	頂点番号		
0	⓪	②	①

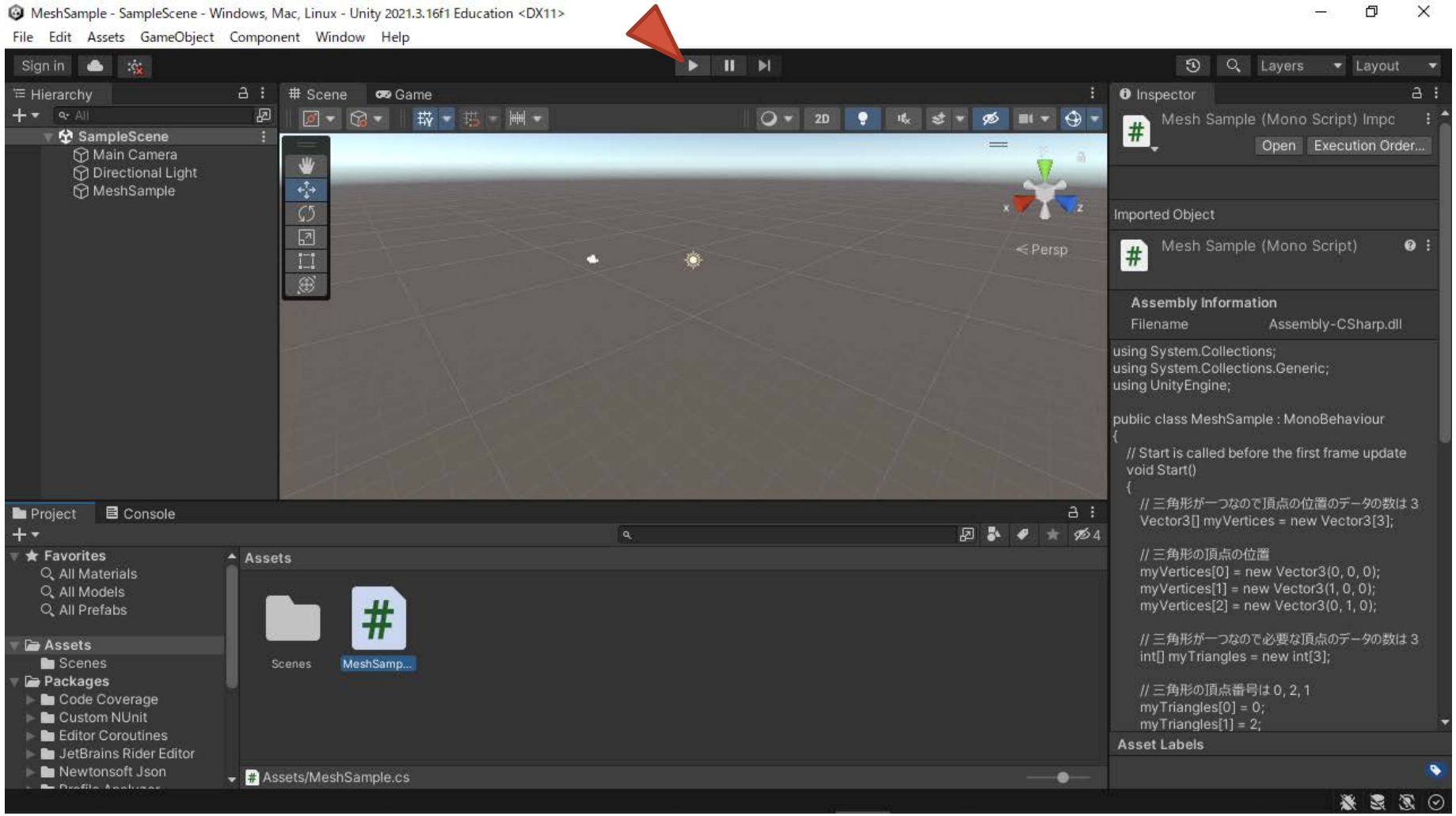
```
// Mesh のオブジェクトを作る
Mesh myMesh = new Mesh();

// Mesh のオブジェクトに頂点の位置を設定する
myMesh.SetVertices(myVertices);

// Mesh のオブジェクトに頂点インデックスを設定する
myMesh.SetTriangles(myTriangles, 0);
```

メッシュフィルタにメッシュを設定する

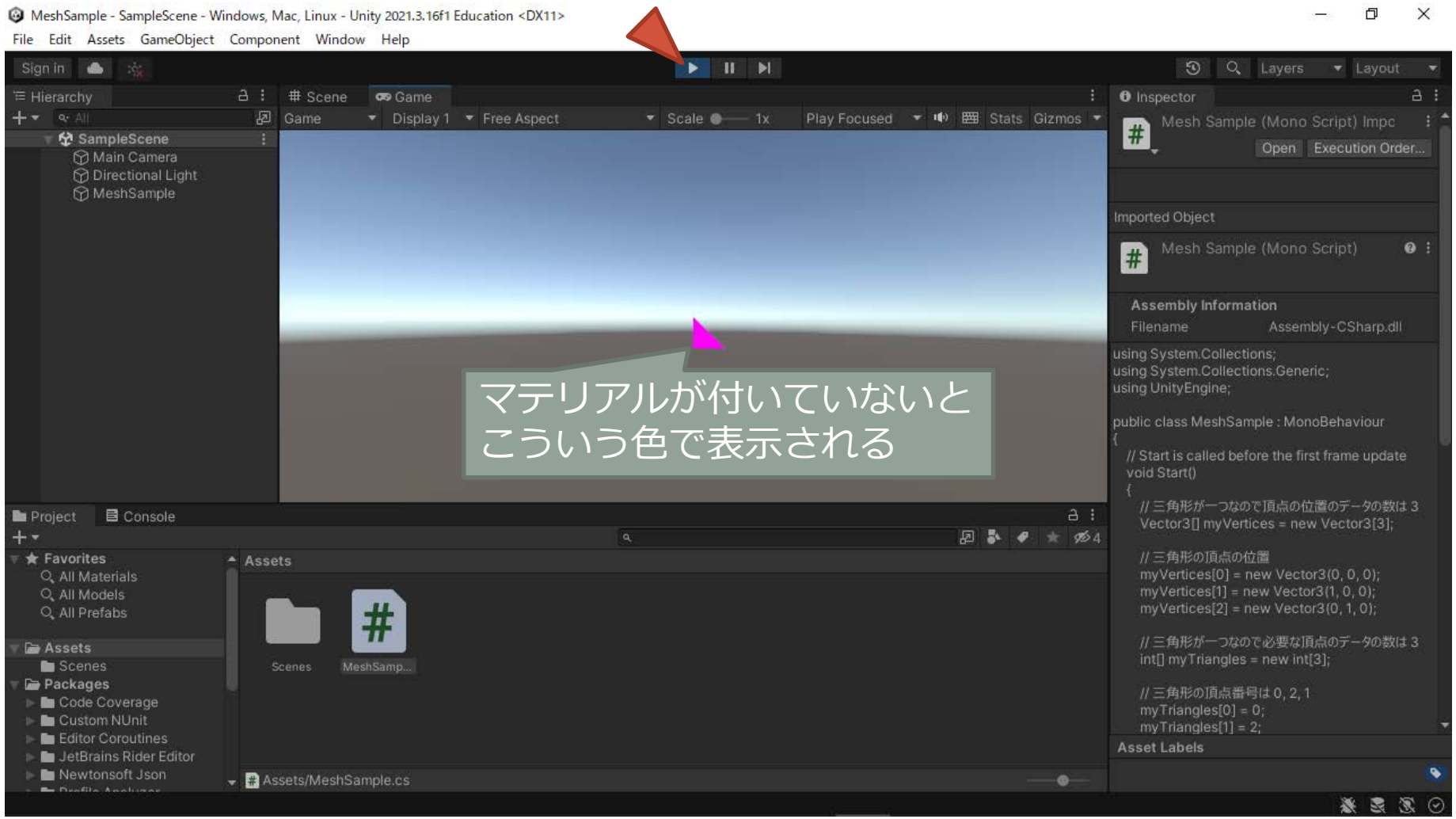
- GameObject に組み込んだ MeshFilter のコンポーネントを meshFilter に取り出す
  - GetComponent<MeshFilter>()
- meshFilter のメンバの mesh に myMesh を設定する
  - meshFilter.mesh = myMesh;

```
        // GameObject から MeshFilter の Component を取り出す
        MeshFilter meshFilter = GetComponent<MeshFilter>();

        // MeshFilter に Mesh のオブジェクトを設定する
        meshFilter.mesh = myMesh;
    }

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

    }
}
```

プロジェクトを実行する



停止する

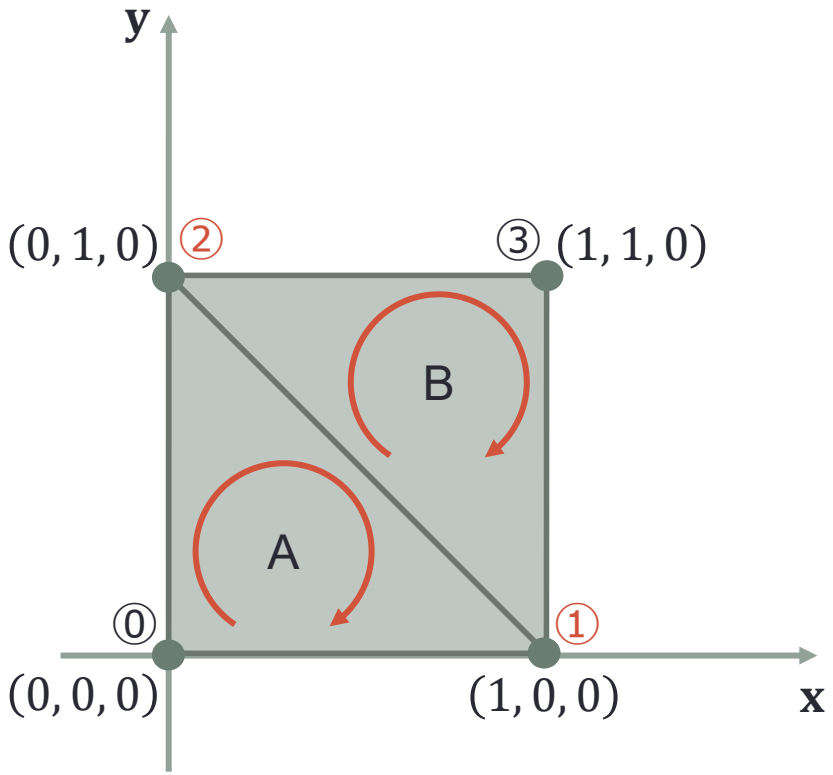

MeshSample - SampleScene - Windows, Mac, Linux - Unity 2021.3.16f1 Education <DX11>
File Edit Assets GameObject Component Window Help
Sign in
Layers
Layout
Hierarchy
SampleScene
Main Camera
Directional Light
MeshSample
Scene
Game
Game
Display 1
Free Aspect
Scale
1x
Play Focused
Stats
Gizmos
マテリアルが付いていないと
こういう色で表示される
Inspector
Mesh Sample (Mono Script) Impc
Open
Execution Order...
Imported Object
Mesh Sample (Mono Script)
Assembly Information
Filename
Assembly-CSharp.dll
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
public class MeshSample : MonoBehaviour
{
// Start is called before the first frame update
void Start()
{
// 三角形が一つなので頂点の位置のデータの数は 3
Vector3[] myVertices = new Vector3[3];
// 三角形の頂点の位置
myVertices[0] = new Vector3(0, 0, 0);
myVertices[1] = new Vector3(1, 0, 0);
myVertices[2] = new Vector3(0, 1, 0);
// 三角形が一つなので必要な頂点のデータの数は 3
int[] myTriangles = new int[3];
// 三角形の頂点番号は 0, 2, 1
myTriangles[0] = 0;
myTriangles[1] = 2;
Asset Labels
Project
Console
Favorites
All Materials
All Models
All Prefabs
Assets
Scenes
Packages
Code Coverage
Custom NUnit
Editor Coroutines
JetBrains Rider Editor
Newtonsoft Json
Assets
Scenes
MeshSamp...
Assets/MeshSample.cs

四角形を描く

四角形以上の多角形は三角形を組み合わせて表現する

頂点インデックスで複数の三角形を表す

- １つの四角形は1辺（2頂点）を共有する２つの三角形で表す



頂点アトリビュート

頂点番号	位置		
	x	y	z
⓪	0	0	0
①	1	0	0
②	0	1	0
③	1	1	0

←追加分

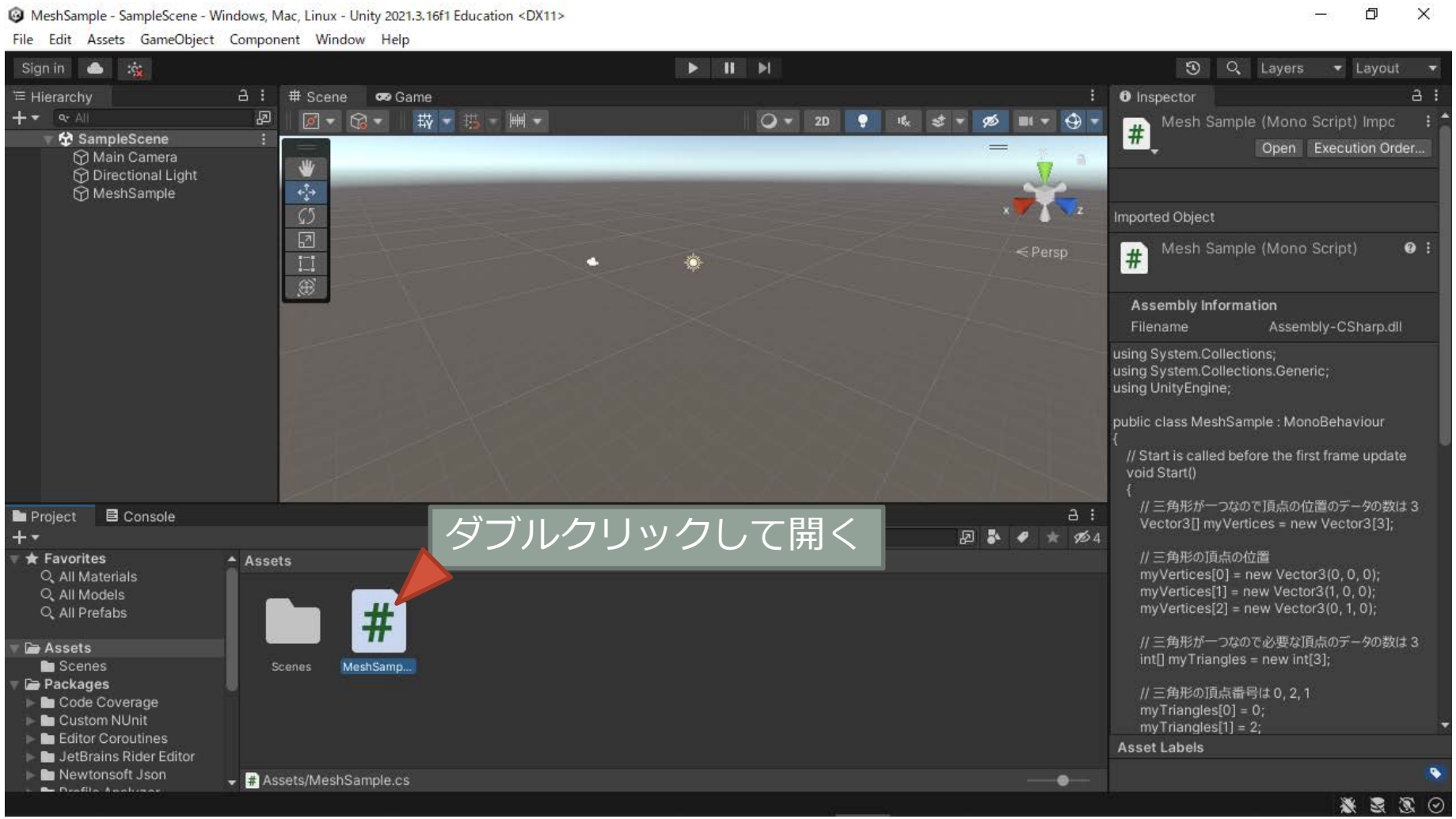
頂点インデックス

三角形	頂点番号		
A	⓪	②	①
B	①	②	③

←追加分

①と②の頂点をＡ，Ｂの２つの三角形で共有しているところがポイント

C# スクリプトを編集する



四角形の頂点アトリビュートを設定する

- 頂点の数は 4

頂点番号	位置		
	x	y	z
⓪	0	0	0
①	1	0	0
②	0	1	0
③	1	1	0

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

public class MeshSample : MonoBehapiour
{
    // Start is called before the first frame update
    void Start()
    {
        // 四角形が１つなので頂点の位置のデータの数は 4
        Vector3[] myVertices = new Vector3[4];←修正

        // 四角形の頂点の位置
        myVertices[0].Set(0, 0, 0);
        myVertices[1].Set(1, 0, 0);
        myVertices[2].Set(0, 1, 0);
        myVertices[3].Set(1, 1, 0); ←追加

        // (以下略)
```

四角形の頂点インデックスを設定する

- 頂点インデックスの数は 6
  - 四角形を三角形２個で表す

三角形	頂点番号		
A	⓪	②	①
B	①	②	③

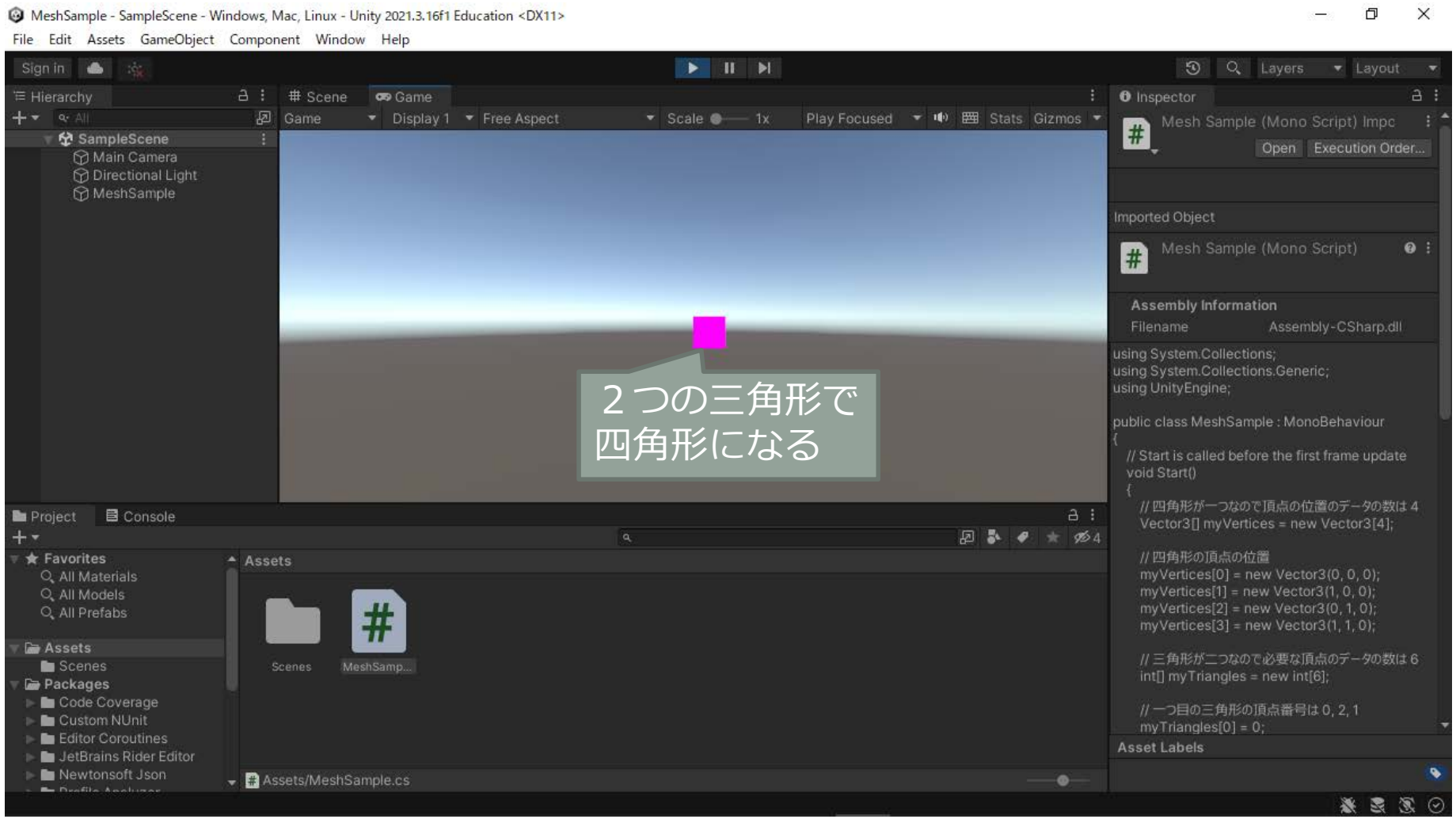
このように格納される

myTriangles	[0]	[1]	[2]	[3]	[4]	[5]
頂点番号	⓪	②	①	①	②	③

```
        // 三角形が２つなので必要な頂点のデータの数は 6
        int[] myTriangles = new int[6];←修正

        // １つ目の三角形の頂点番号は 0, 2, 1
        myTriangles[0] = 0;
        myTriangles[1] = 2;
        myTriangles[2] = 1;

        // ２つ目の三角形の頂点番号は 0, 3, 2
        myTriangles[3] = 1;
        myTriangles[4] = 2;
        myTriangles[5] = 3;

        // メッシュのオブジェクトを作る
        Mesh myMesh = new Mesh();

        // Mesh のオブジェクトに頂点の位置を設定する
        myMesh.SetVertices(myVertices);

        // Mesh のオブジェクトに頂点インデックスを設定する
        myMesh.SetTriangles(myTriangles, 0);

        // (中略)
    }

    // (中略)
}
```

追加

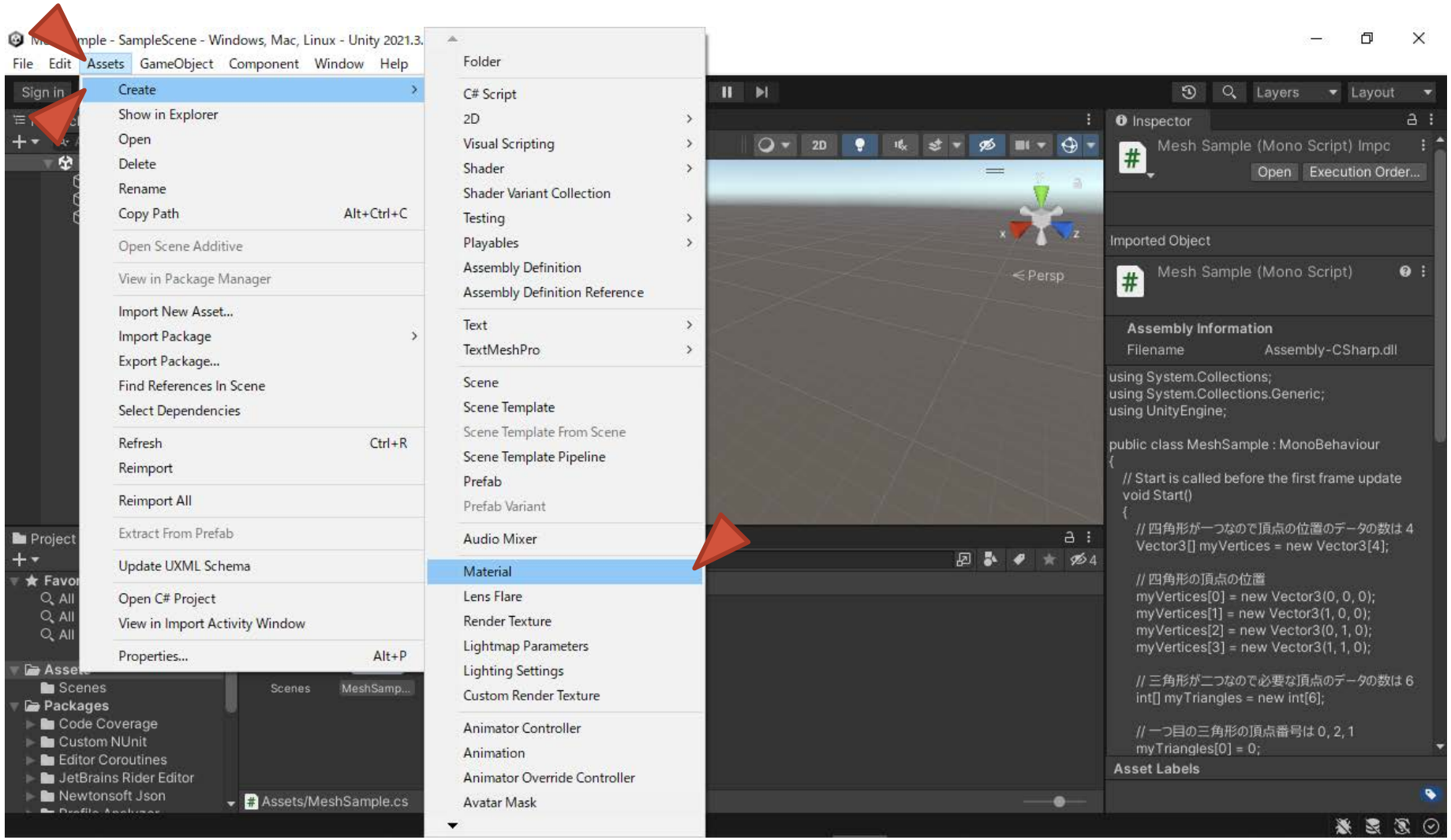
プロジェクトを実行する



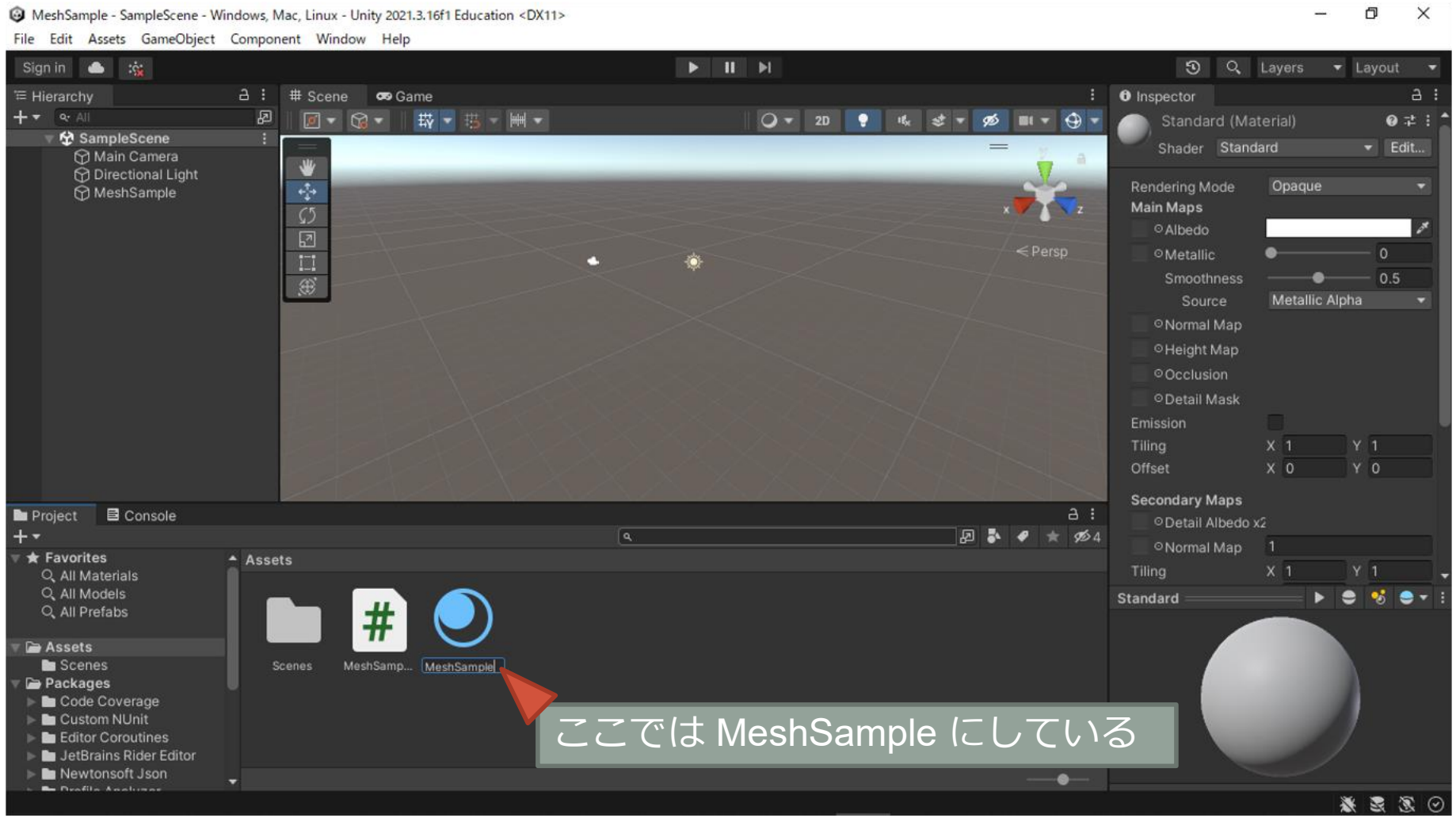
陰影をつける

マテリアルと法線を設定する

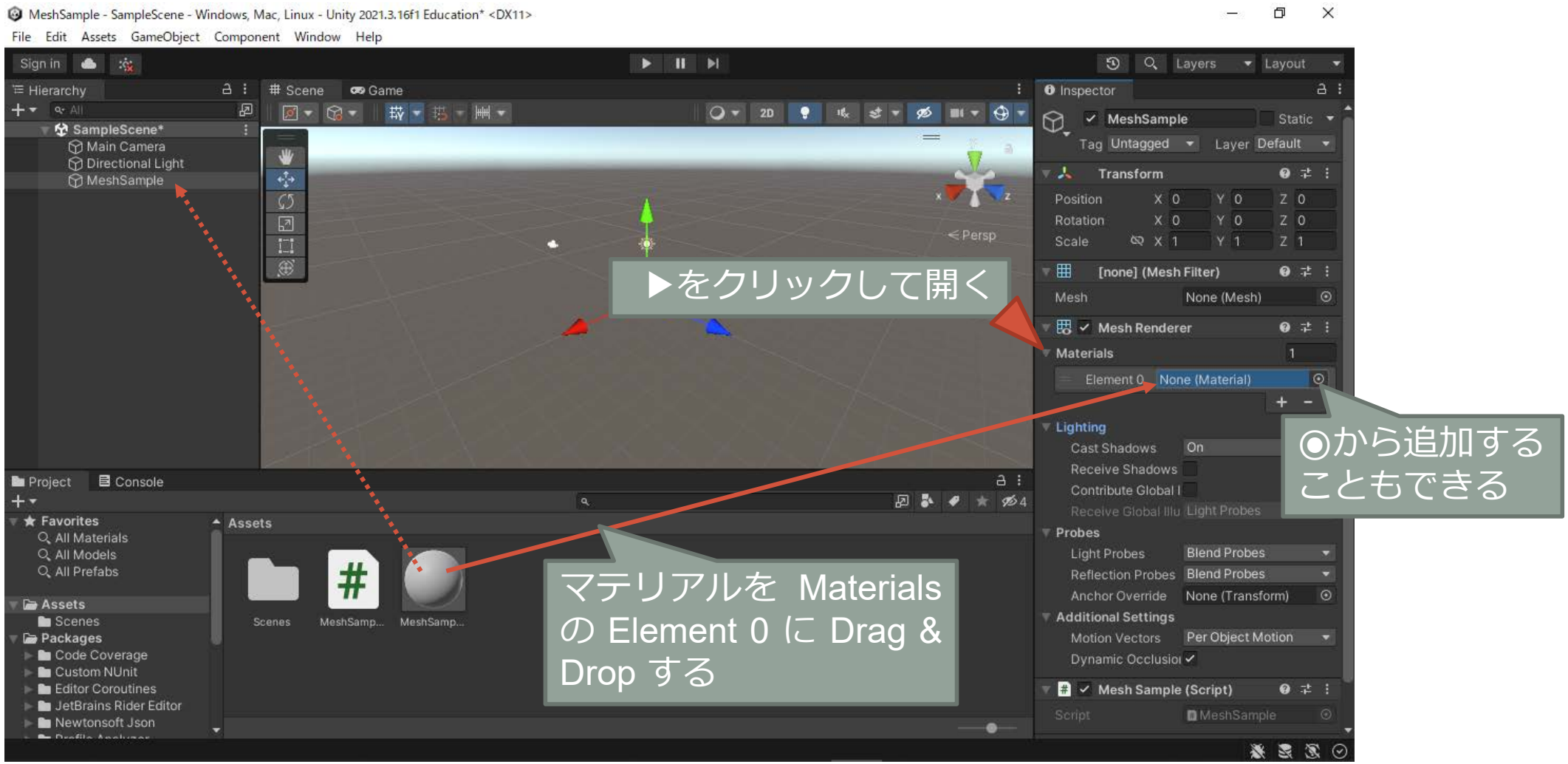
マテリアル（材質）を作成する



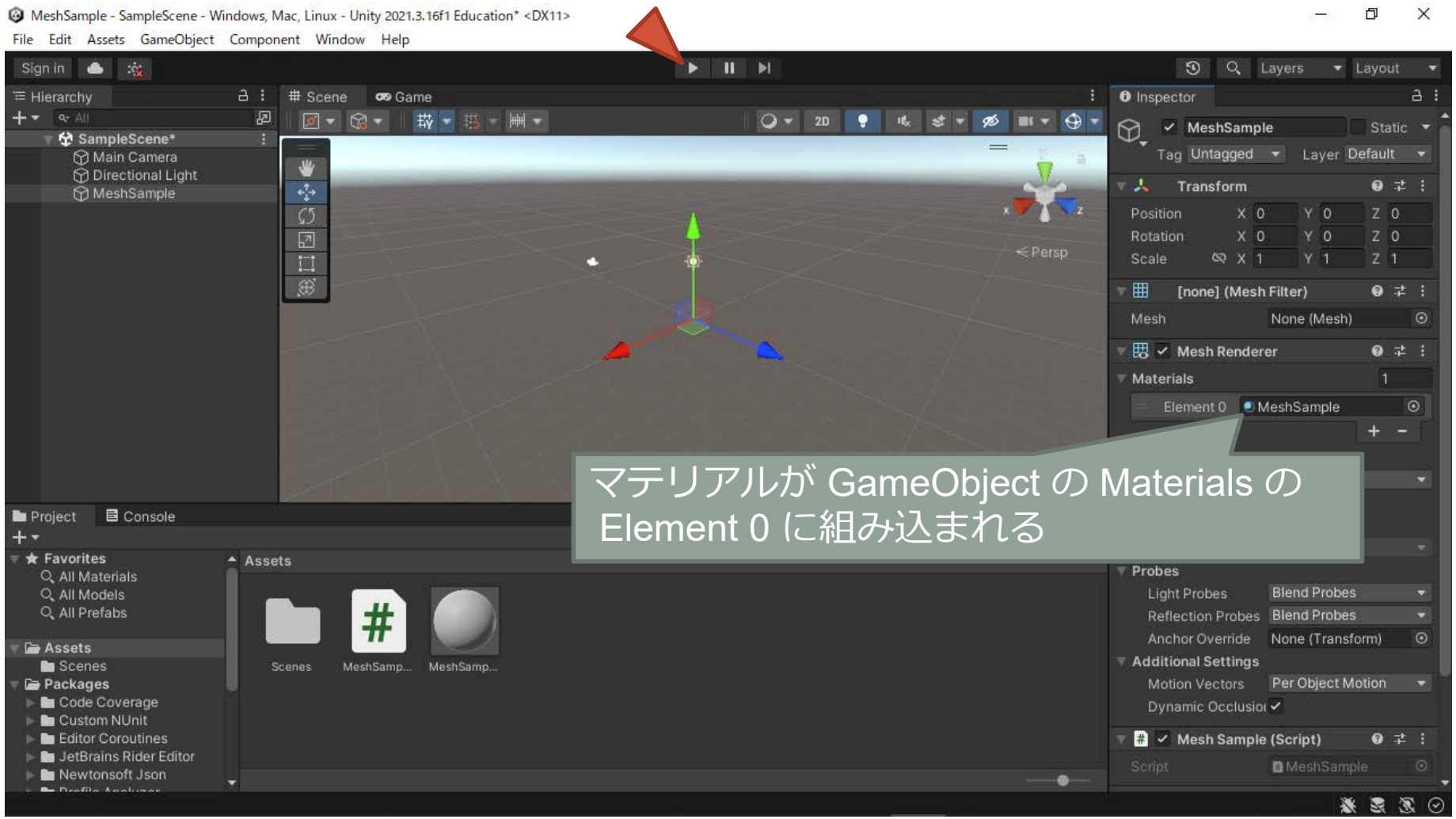
作成したマテリアルの名前を変更する



マテリアルを GameObject に組み込む



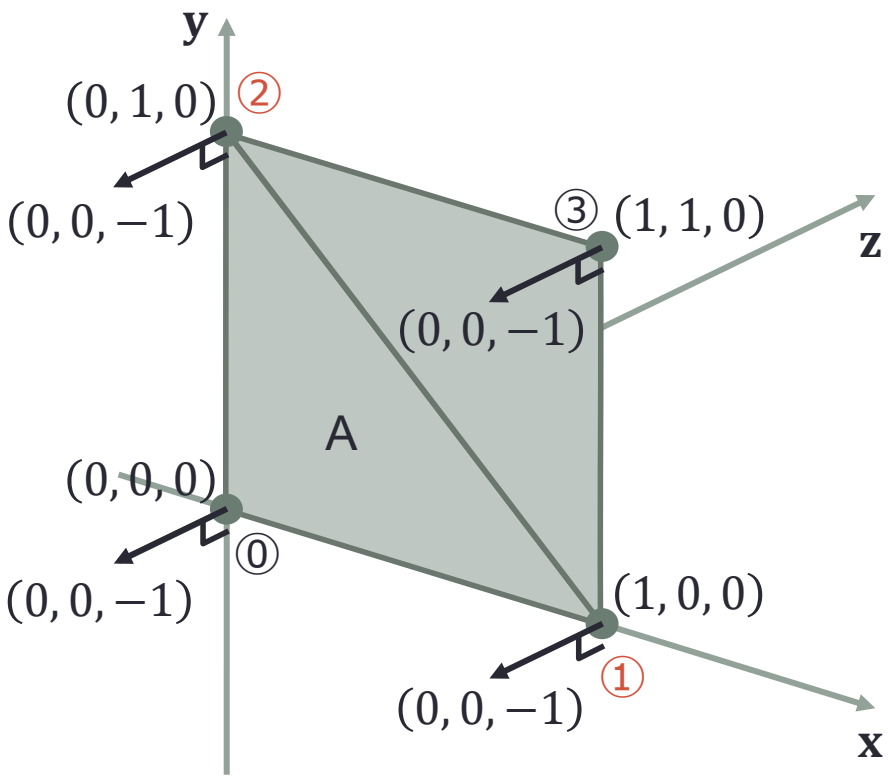
マテリアルが GameObject の Materials に組み込まれる



プロジェクトを実行する

頂点の法線

- 法線は頂点ごとに設定する
  - 共有されている頂点では法線も共有される



頂点アトリビュート

頂点番号	位置		
	x	y	z
⓪	0	0	0
①	1	0	0
②	0	1	0
③	1	1	0

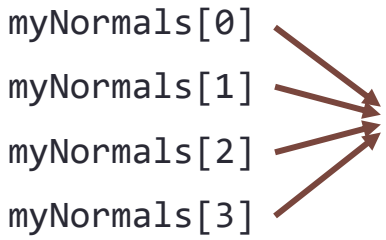
頂点番号	法線		
	x	y	z
⓪	0	0	−1
①	0	0	−1
②	0	0	−1
③	0	0	−1

頂点インデックス

三角形	頂点番号		
A	⓪	②	①
B	①	②	③

頂点アトリビュートに設定する法線のデータを準備する

- 法線のデータ
  - Vector3[] myNormals = new Vector3[4];
    - 4つの Vector3 型の値を要素に持つ配列を生成して myVertices から参照する
  - myNormals[0] = Vector3.back;
    - myNormals の 0 番目から Vector3 型の値 (0, 0, -1) をもつ静的変数 back を参照する
  - myNormals[1] = myNormals[0];
    - myNormals の 1 番目は myNormals の 0 番目と同じものを参照する



頂点番号	法線		
	x	y	z
⓪	0	0	−1
①	0	0	−1
②	0	0	−1
③	0	0	−1

```
// Start is called before the first frame update
void Start()
{
    // 四角形が１つなので頂点の位置のデータの数は 4
    Vector3[] myVertices = new Vector3[4];

    // 四角形の頂点の位置
    myVertices[0].Set(0, 0, 0);
    myVertices[1].Set(1, 0, 0);
    myVertices[2].Set(0, 1, 0);
    myVertices[3].Set(1, 1, 0);

    // メッシュの頂点の法線の配列
    Vector3[] myNormals = new Vector3[4];

    // 四角形の頂点の法線                    new Vector3(0, 0, -1)
    myNormals[0] = Vector3.back;
    myNormals[1] = myNormals[0];
    myNormals[2] = myNormals[0];
    myNormals[3] = myNormals[0];

    // 三角形が２つなので必要な頂点のデータの数は 6
    int[] myTriangles = new int[6];

    // (以下略)
```

メッシュに頂点の法線のデータを設定する

- 頂点の法線のデータ myNormals を myMesh に設定する
  - myMesh.SetNormals(myNormals);

頂点番号	法線		
	x	y	z
⓪	0	0	−1
①	0	0	−1
②	0	0	−1
③	0	0	−1

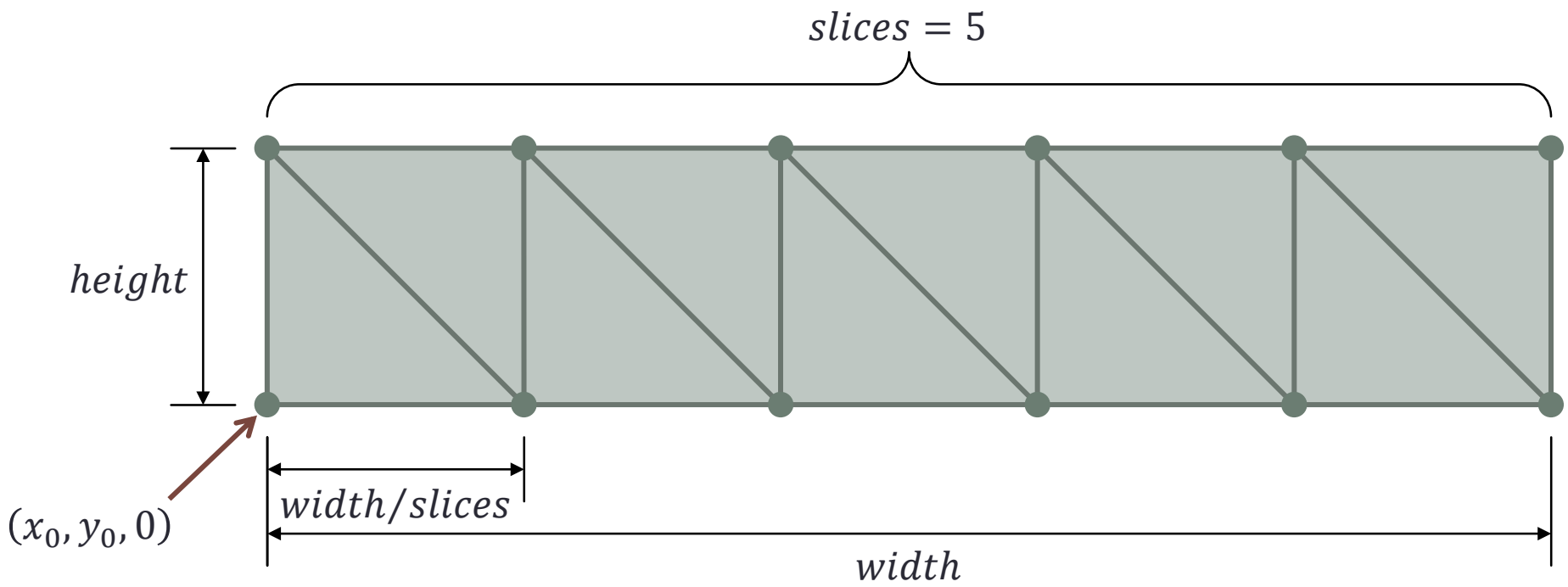
```
    // メッシュのオブジェクトを作る
    Mesh myMesh = new Mesh();

    // Mesh のオブジェクトに頂点の位置を設定する
    myMesh.SetVertices(myVertices);

    // Mesh のオブジェクトに頂点の法線を設定する
    myMesh.SetNormals(myNormals);

    // Mesh のオブジェクトに頂点インデックスを設定する
    myMesh.SetTriangles(myTriangles, 0);

    // GameObject から MeshFilter の Component を取り出す
    MeshFilter meshFilter = GetComponent<MeshFilter>();

    // MeshFilter に Mesh のオブジェクトを設定する
    meshFilter.mesh = myMesh;
  }

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

  }
}
```

プロジェクトを実行する

帯を描く

四角形を横に並べる

四角形を横に並べる



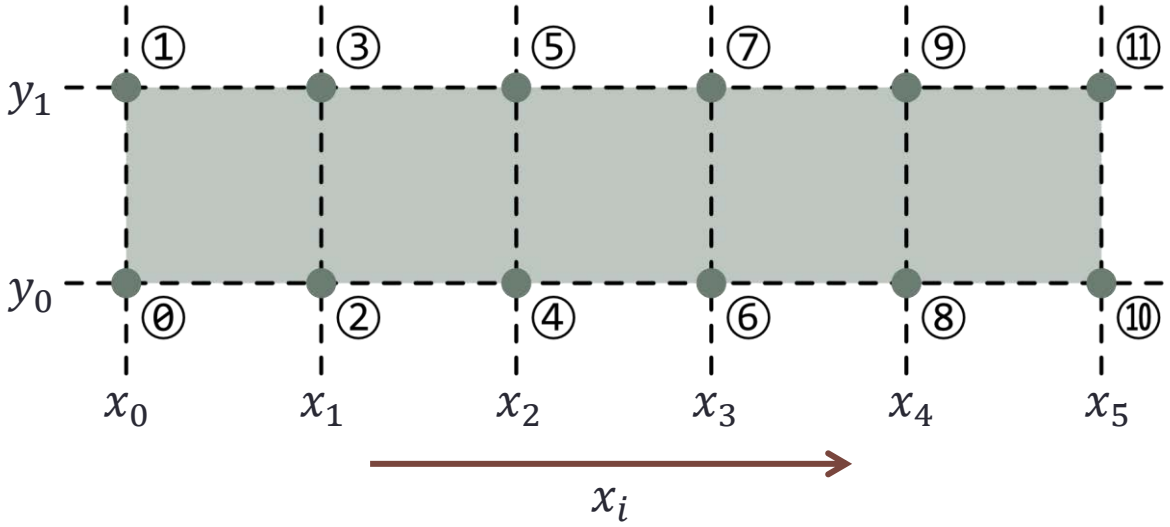
定数をあらかじめ求めておく

- 四角形が $slice$ 枚あるとき頂点の数 $nVertices$ は
  - $nVertices = (slice + 1) \times 2$
- 全体の高さが $height$ 下側の頂点の高さが y_0 のとき上側の頂点の高さ y_1 は
  - $y_1 = y_0 + height$

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

public class MeshSample : MonoBehapiour
{
    const int slices = 5;
    const float x0 = -5f;
    const float y0 = -1f;
    const float width = 10f;
    const float height = 2f;

    // Start is called before the first frame update
    void Start()
    {
        // 頂点の数
        int nVertices = (slices + 1) * 2;

        //メッシュの頂点の位置の配列
        Vector3[] myVertices = new Vector3[nVertices];

        // メッシュの頂点の法線の配列
        Vector3[] myNormals = new Vector3[nVertices];

        // 上側の点の高さ
        float y1 = y0 + height;
```

頂点の位置と法線を求める

- 左端の頂点 $i = 0$ の x 座標が x_0 のとき左から i 番目の頂点の x 座標 x_i は
  - $x_i = x_0 + width \times i / slices$
- i が 1 進むごとに頂点番号は②進む
  - 頂点番号は下側が偶数で上側が奇数

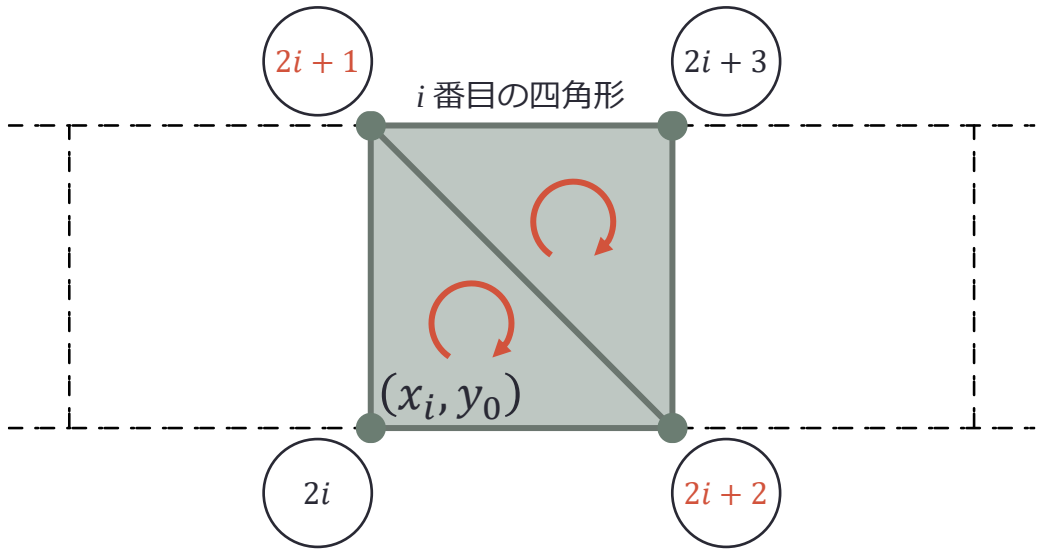



```
// 各頂点の位置と法線を求める
for (int i = 0; i <= slices; ++i)
{
    // 横方向のパラメータ (0→1)
    float u = (float)i / slices;

    // 左から i 番目の四角形の左下の頂点の x 座標値
    float xi = x0 + width * u;

    // 左から i 番目の四角形の左下の頂点の頂点番号
    int pi = i * 2;

    // 左から i 番目の四角形の左下の頂点の位置と法線
    myVertices[pi].Set(xi, y0, 0);
    myNormals[pi] = Vector3.back;

    // 左から i 番目の四角形の左上の頂点の位置と法線
    myVertices[pi + 1].Set(xi, y1, 0);
    myNormals[pi + 1] = Vector3.back;
}
```

三角形の頂点番号を求める

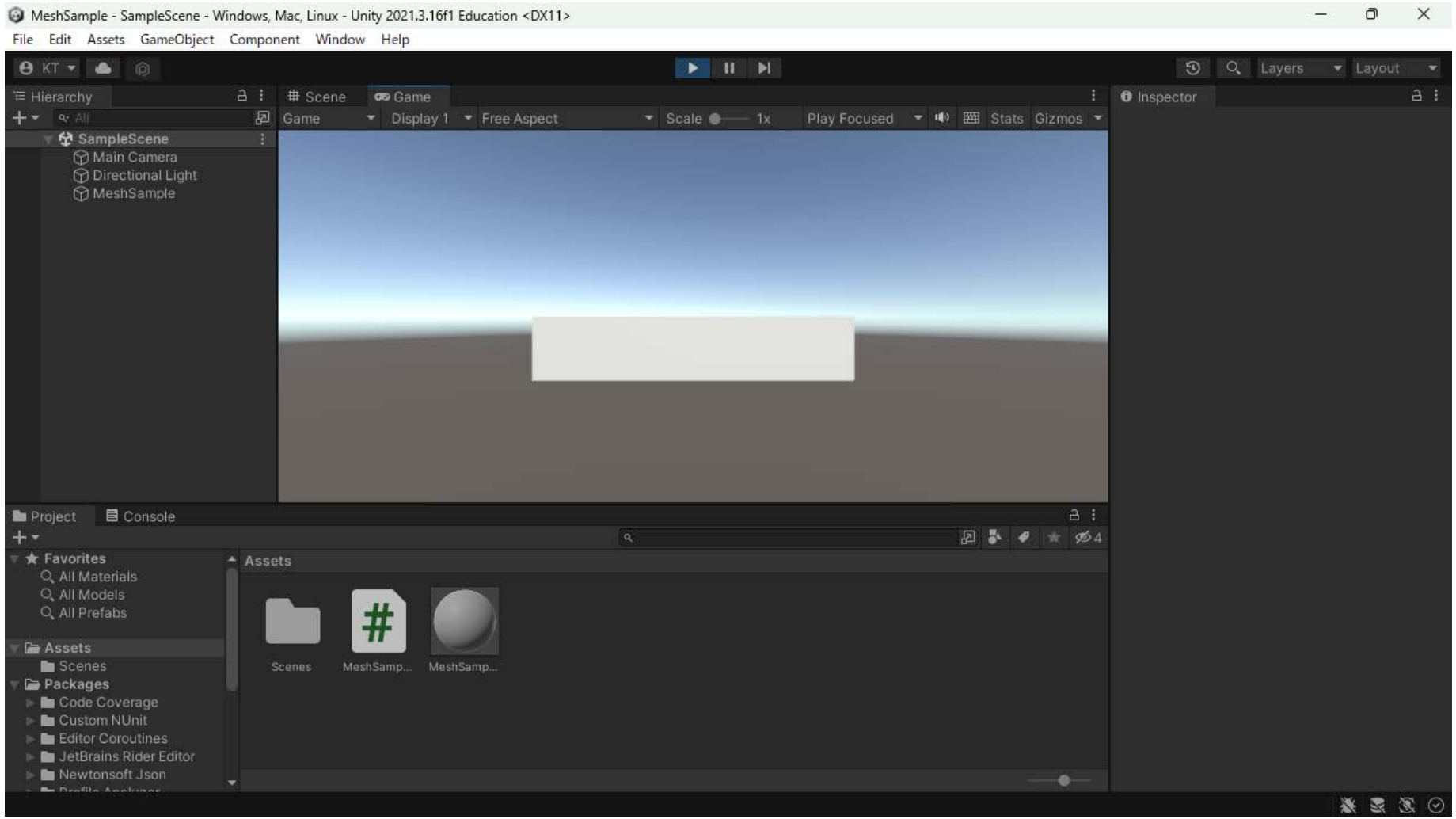
- 1 つの四角形を 2 つの三角形で表す
  - したがって頂点は 6 個ある
  - i 番目の四角形の最初の頂点番号の格納先は myTriangles[$6i$]

myTriangles	…	$[6i]$	$[6i+1]$	$[6i+2]$	$[6i+3]$	$[6i+4]$	$[6i+5]$	…
頂点番号	…	$2i$	$2i+1$	$2i+2$	$2i+3$	$2i+2$	$2i+1$	…

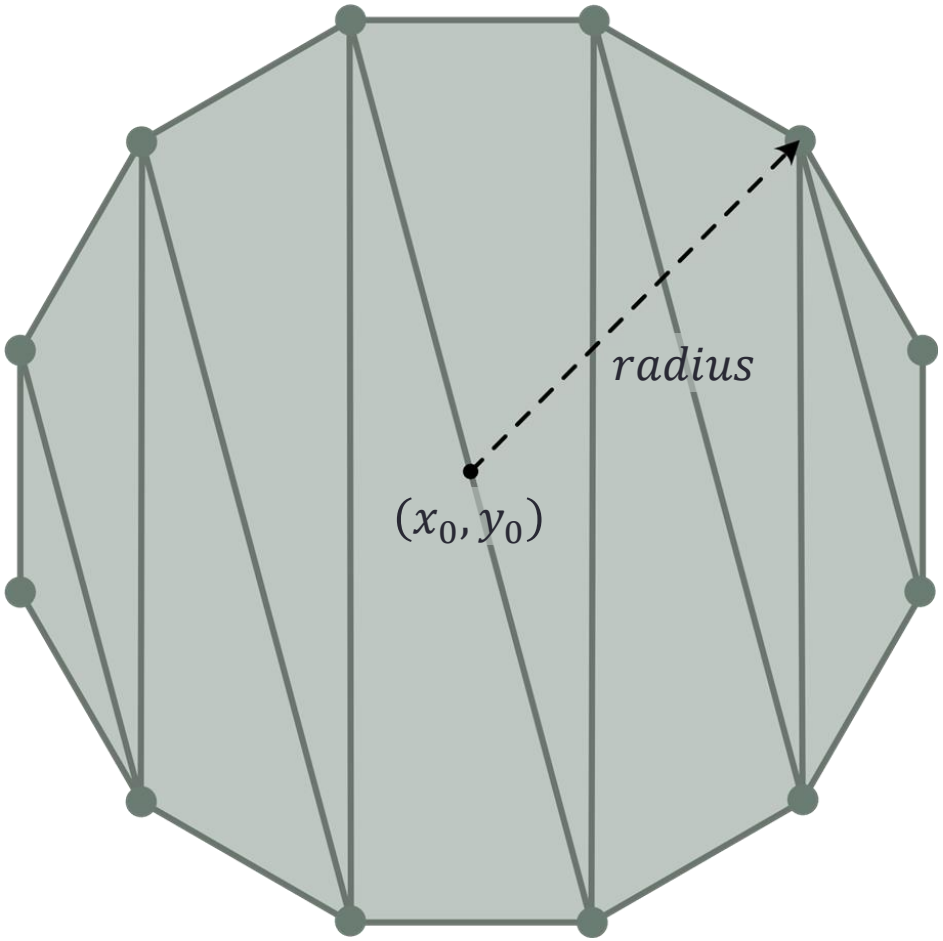



```
        // 三角形の頂点の数
        int nTriangles = slices * 6;

        // 三角形のデータ
        int[] myTriangles = new int[nTriangles];

        // 三角形の頂点番号を求める
        for (int i = 0; i < slices; ++i)
        {
            // 左から i 番目の四角形の左下の頂点番号の格納先
            int fi = i * 6;

            // 左から i 番目の四角形の左下の頂点番号
            int pi = i * 2;

            //  1 つ目の三角形の頂点番号
            myTriangles[fi + 0] = pi;
            myTriangles[fi + 1] = pi + 1;
            myTriangles[fi + 2] = pi + 2;

            //  2 つ目の三角形の頂点番号
            myTriangles[fi + 3] = pi + 3;
            myTriangles[fi + 4] = pi + 2;
            myTriangles[fi + 5] = pi + 1;
        }

        // (以下略)
```

プロジェクトを実行する

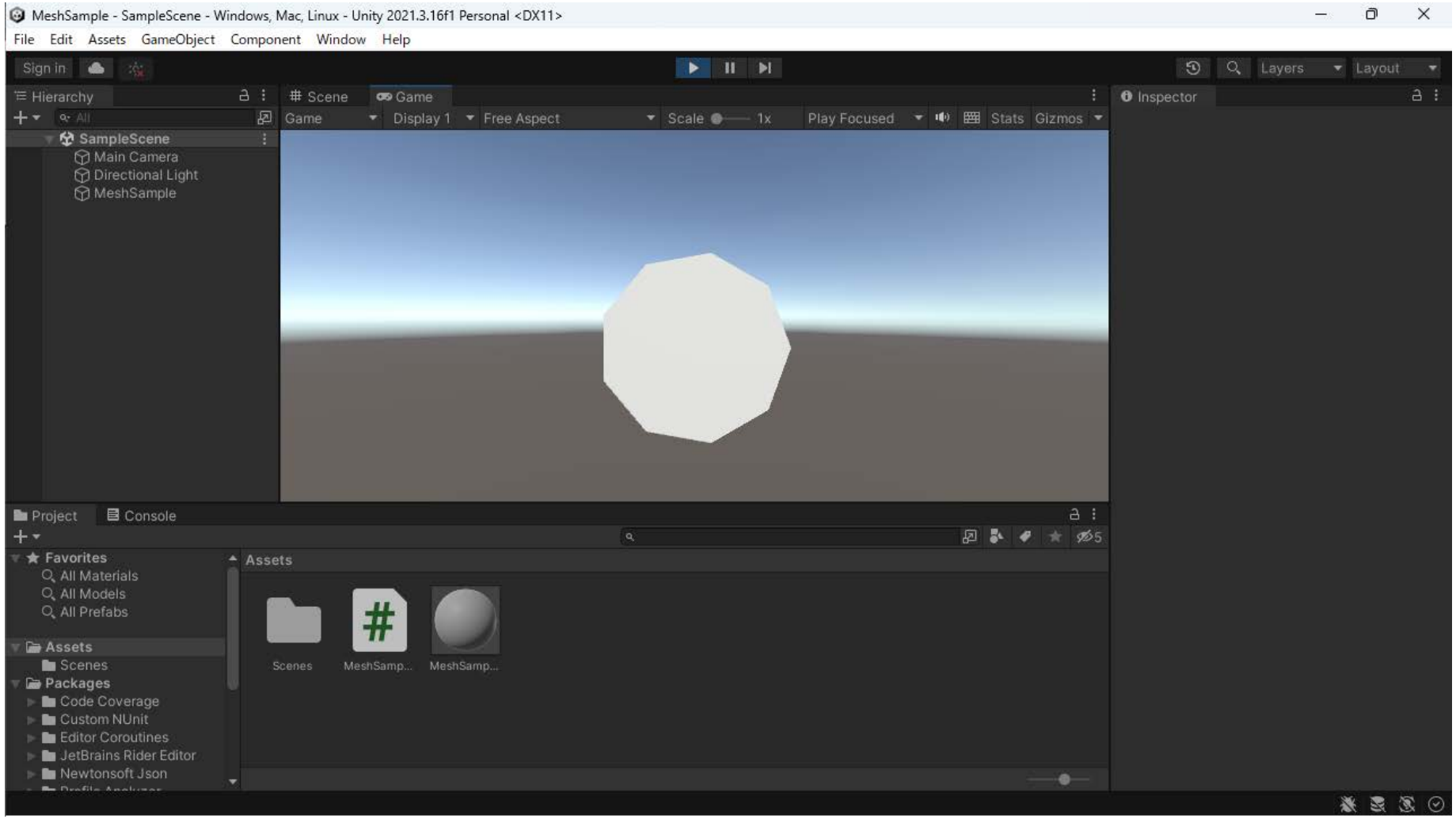


演習１：正多角形を描いてください



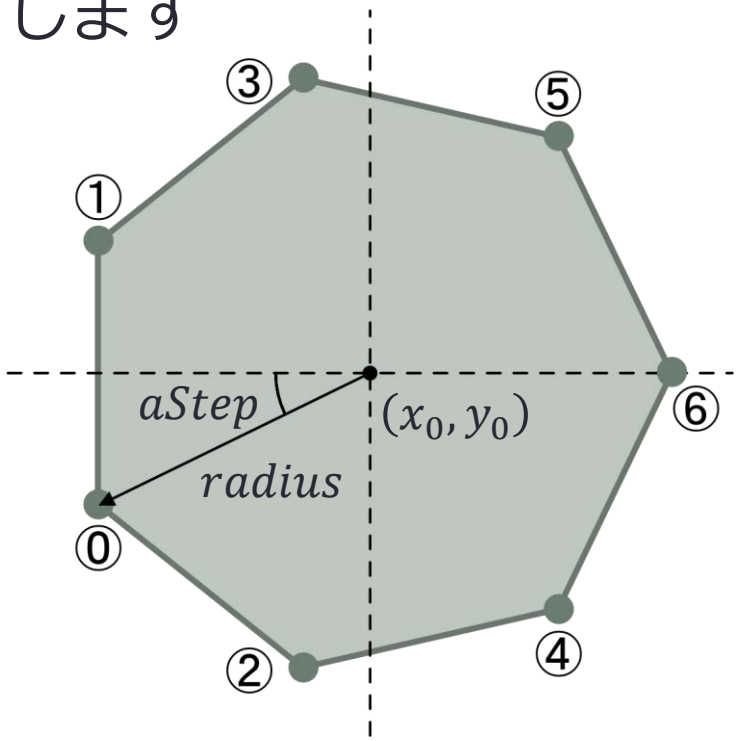
- 中心が (x_0, y_0) にあり半径が $radius$ の正 n 角形を描いてください
  - 三角関数 $\cos\theta$ は Mathf.Cos(θ)、$\sin\theta$ は Mathf.Sin(θ)、円周率 π は Mathf.PI で求めることができます

実行例


MeshSample - SampleScene - Windows, Mac, Linux - Unity 2021.3.16f1 Personal <DX11>
File Edit Assets GameObject Component Window Help
Sign in
Layers
Layout
Hierarchy
SampleScene
Main Camera
Directional Light
MeshSample
Scene
Game
Game
Display 1
Free Aspect
Scale
1x
Play Focused
Stats
Gizmos
Inspector
Project
Console
Favorites
All Materials
All Models
All Prefabs
Assets
Scenes
Packages
Code Coverage
Custom NUnit
Editor Coroutines
JetBrains Rider Editor
Newtonsoft Json
Assets
Scenes
MeshSamp...
MeshSamp...

解答例 (1) 一辺当たりの中心角の二分の一を求める

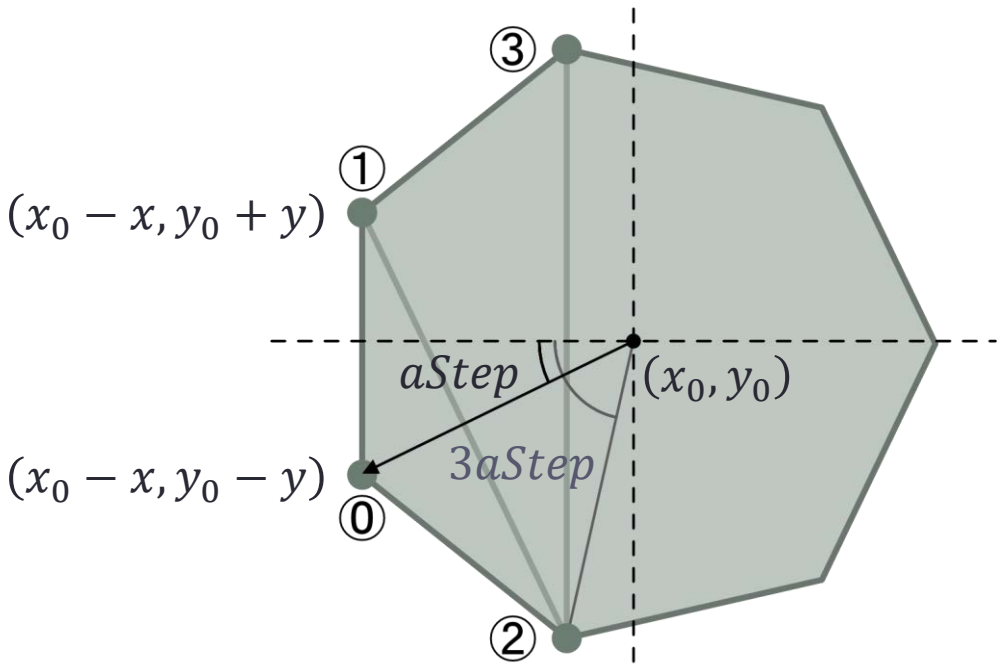
- 正 n 角形では n は頂点の数すなわち $nVertices$ です
- 一辺あたりの中心角の２分の１を $aStep$ とします




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

public class MeshSample : MonoBehapiour
{
    const int nVertices = 9;
    const float x0 = 0f;
    const float y0 = 0f;
    const float radius = 3f;

    // Start is called before the first frame update
    void Start()
    {
        // メッシュの頂点の位置の配列
        Vector3[] myVertices = new Vector3[nVertices];

        // メッシュの頂点の法線の配列
        Vector3[] myNormals = new Vector3[nVertices];

        // 一辺当たりの中心角の 1 / 2
        float aStep = Mathf.PI / nVertices;
```

解答例 (2) 二つの頂点の位置と法線を求める

- 各頂点の位置と法線を求めます
  - 途中の ++i により $i = 0, 2, 4, 6, \cdots$ と増加するので $i + 1 = 1, 3, 5, 7, \cdots$ となります




```
// 各頂点の位置と法線を求める
for (int i = 0; i < nVertices; ++i)
{
    // 中心から i 番目の頂点に向かう角度
    float angle = (i + 1) * aStep;

    // 頂点の x 座標値
    float x = radius * Mathf.Cos(angle);

    // 頂点の y 座標値
    float y = radius * Mathf.Sin(angle);

    // 下側の頂点の位置と法線
    myVertices[i].Set(x0 - x, y0 - y, 0);
    myNormals[i] = Vector3.back;

    // 最後の頂点を生成したら終了
    if (++i >= nVertices) break;

    // 上側の頂点の位置と法線
    myVertices[i].Set(x0 - x, y0 + y, 0);
    myNormals[i] = Vector3.back;
}
```

解答例 (3) 頂点番号を３つずつ求める

- すべての三角形について
  - 頂点３個ごとにスライドしながら頂点番号を格納します
  - 頂点の数を偶数個に限定すれば「帯」と同じ算出方法が使えます

$i = 0$ のとき：

$i \quad \rightarrow ⓪$

$i + 1 \quad \rightarrow ①$

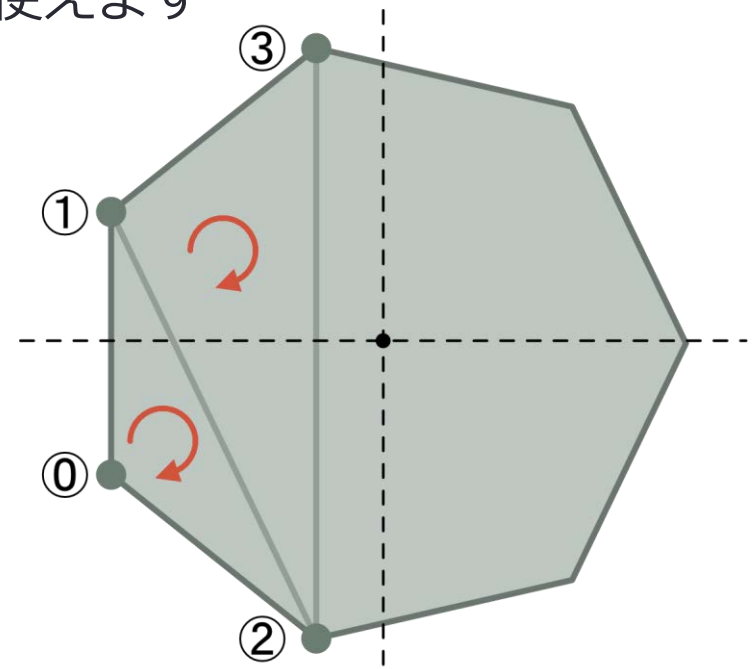
$i + 2 \quad \rightarrow ②$

$i = 1$ のとき：

$i + 2 \quad \rightarrow ③$

$i + 1 \quad \rightarrow ②$

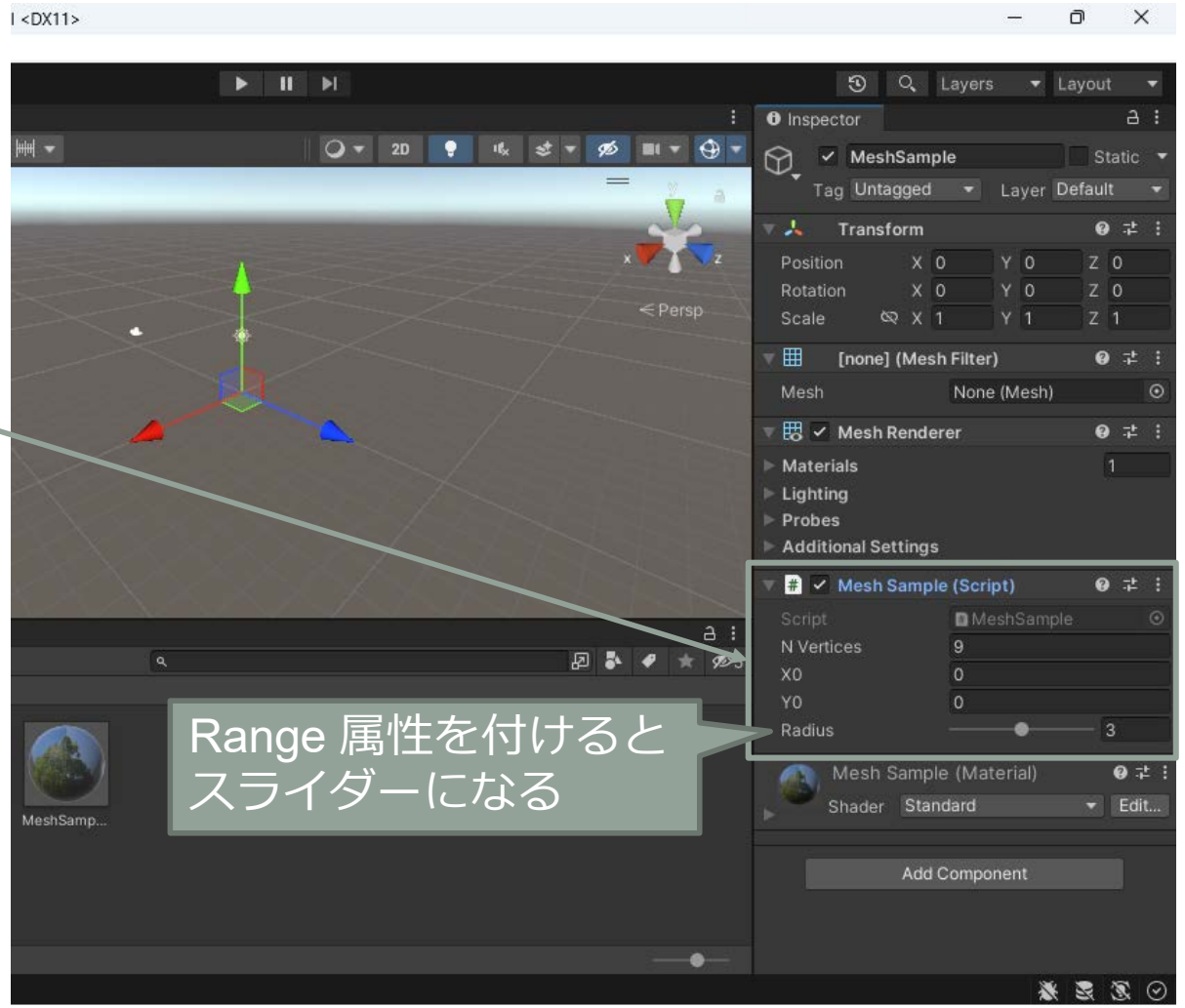
$i \quad \rightarrow ①$




```
// 三角形の数と三角形の頂点の数
int nPolygons = nVertices - 2;
int nTriangles = nPolygons * 3;

// 三角形のデータ
int[] myTriangles = new int[nTriangles];

// 三角形の頂点番号を求める
for (int i = 0; i < nPolygons; ++i)
{
    // １つ目の三角形の最初の頂点の頂点番号の格納先
    int fi = i * 3;

    // １つ目の三角形の頂点番号
    myTriangles[fi + 0] = i;
    myTriangles[fi + 1] = i + 1;
    myTriangles[fi + 2] = i + 2;

    // 最後の頂点番号を格納したら終了
    if (++i >= nPolygons) break;

    // ２つ目の三角形の頂点番号
    myTriangles[fi + 3] = i + 2;
    myTriangles[fi + 4] = i + 1;
    myTriangles[fi + 5] = i;
}
```

補足：メンバ変数を public にするとインスペクタで設定できる

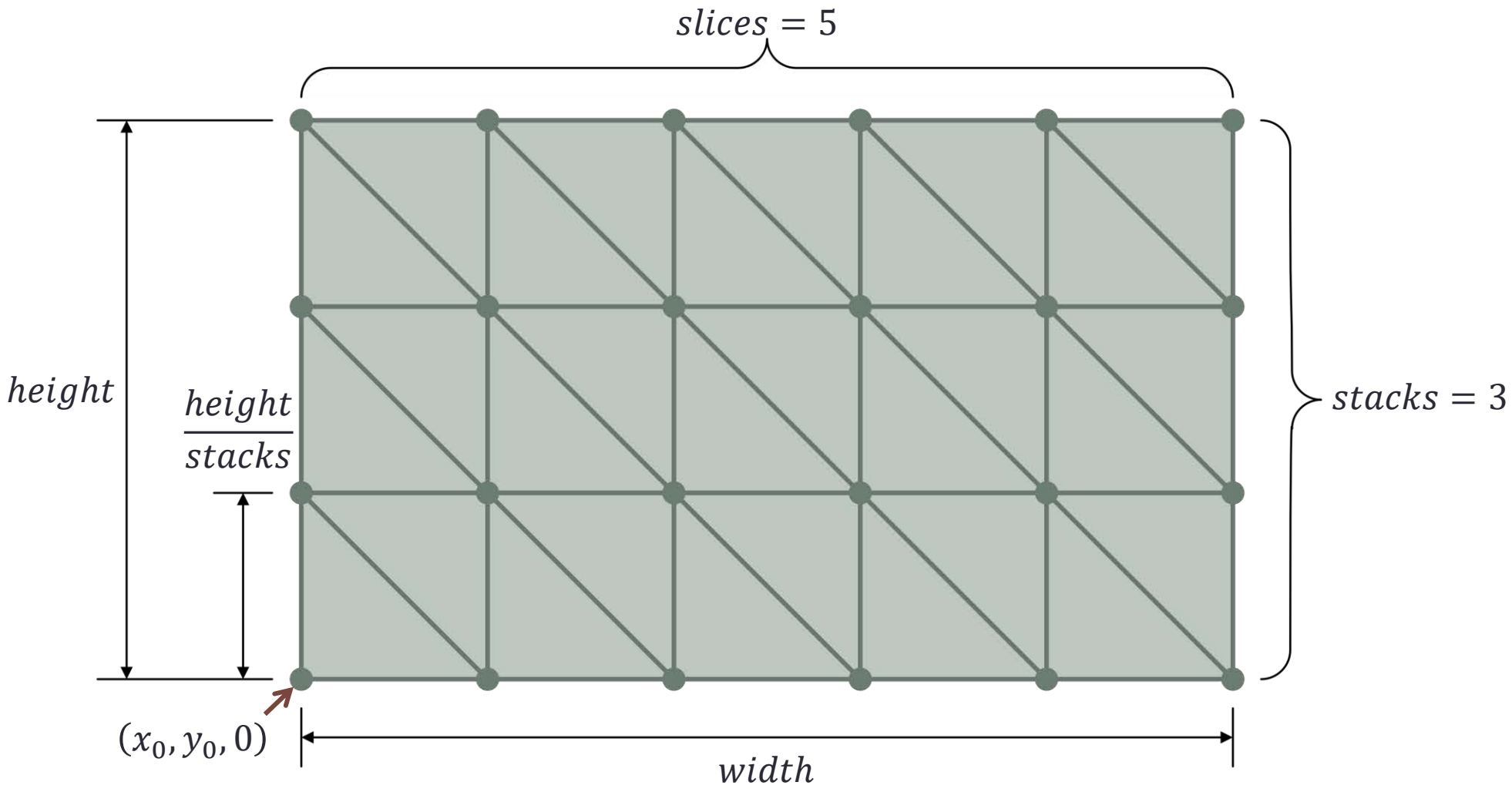
```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class MeshSample : MonoBehaviour
{
    public int nVertices = 9;
    public float x0 = 0f;
    public float y0 = 0f;
    [Range(1, 5)] public float radius = 3f;

    // Start is called before the first frame update
    void Start()
    {
        // メッシュの頂点の位置の配列
        Vector3[] myVertices = new Vector3[nVertices];

        // メッシュの頂点の法線の配列
        Vector3[] myNormals = new Vector3[nVertices];

        // 一辺当たりの中心角の 1 / 2
        float aStep = Mathf.PI / nVertices;
```



格子を描く

四角形を縦横に並べる

四角形を縦横に並べる



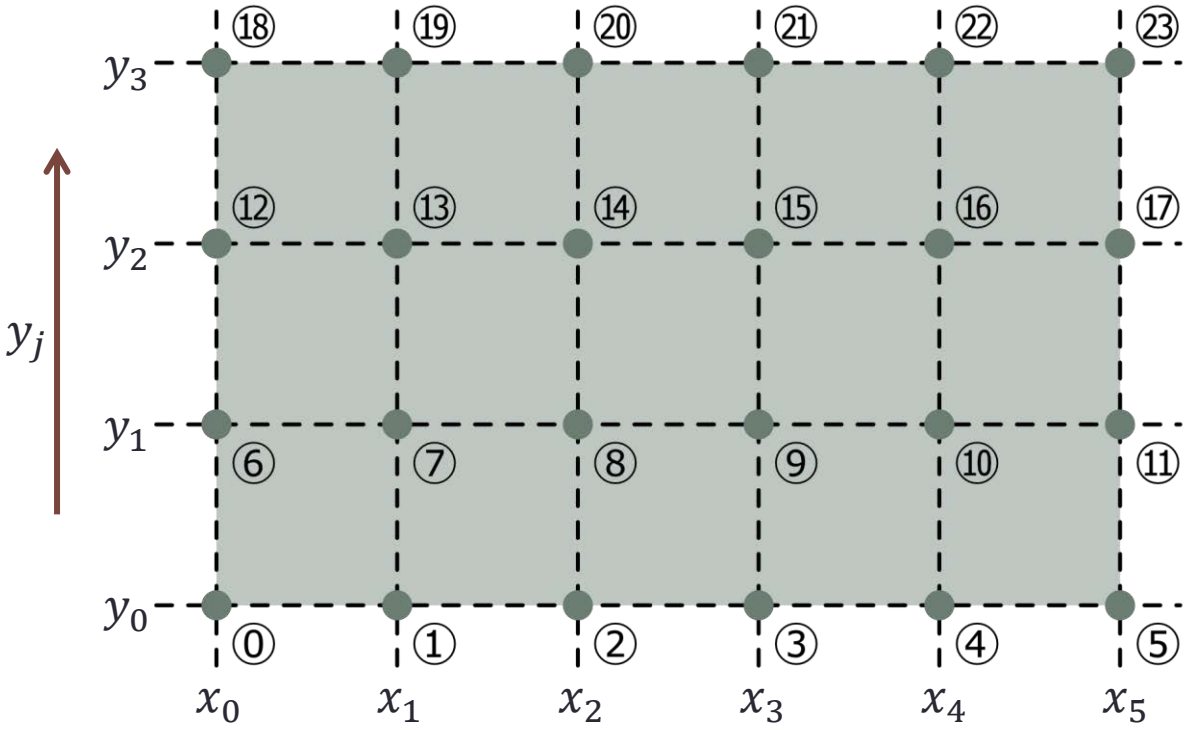
定数をあらかじめ求めておく

- 四角形が $slice \times stacks$ 枚あるとき頂点の数 $nVertices$ は
  - $nVertices = (slice + 1) \times (stacks + 1)$

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

public class MeshSample : MonoBehapiour
{
    const int slices = 5;
    const int stacks = 3;
    const float x0 = -5f;
    const float y0 = -1f;
    const float width = 10f;
    const float height = 6f;

    // Start is called before the first frame update
    void Start()
    {
        // 頂点の数
        int nVertices = (stacks + 1) * (slices + 1);

        // メッシュの頂点の位置の配列
        Vector3[] myVertices = new Vector3[nVertices];

        // メッシュの頂点の法線の配列
        Vector3[] myNormals = new Vector3[nVertices];
```

頂点の位置と法線を求める

- 下端の頂点 $j = 0$ の y 座標が y_0 のとき下から j 番目の頂点の y 座標 y_j は
  - $y_j = y_0 + j \times hStep$




```
        // 各頂点の位置と法線を求める
        for (int j = 0; j <= stacks; ++j)
        {
            // 縦方向のパラメータ（0→1）
            float v = (float)j / stacks;

            // 各段の下側の頂点の高さ
            float yj = y0 + height * v;

            // 各段の左端の下側の頂点の頂点番号
            int p0 = j * (slices + 1);

            for (int i = 0; i <= slices; ++i)
            {
                // 横方向のパラメータ（0→1）
                float u = (float)i / slices;

                // 左から i 番目の四角形の左下の頂点の x 座標値
                float xi = x0 + width * u;

                // 左から i 番目の四角形の左下の頂点番号
                int pi = p0 + i;

                // 左から i 番目の四角形の左下の頂点の位置と法線
                myVertices[pi].Set(xi, yj, 0);
                myNormals[pi] = Vector3.back;
            }
        }
```

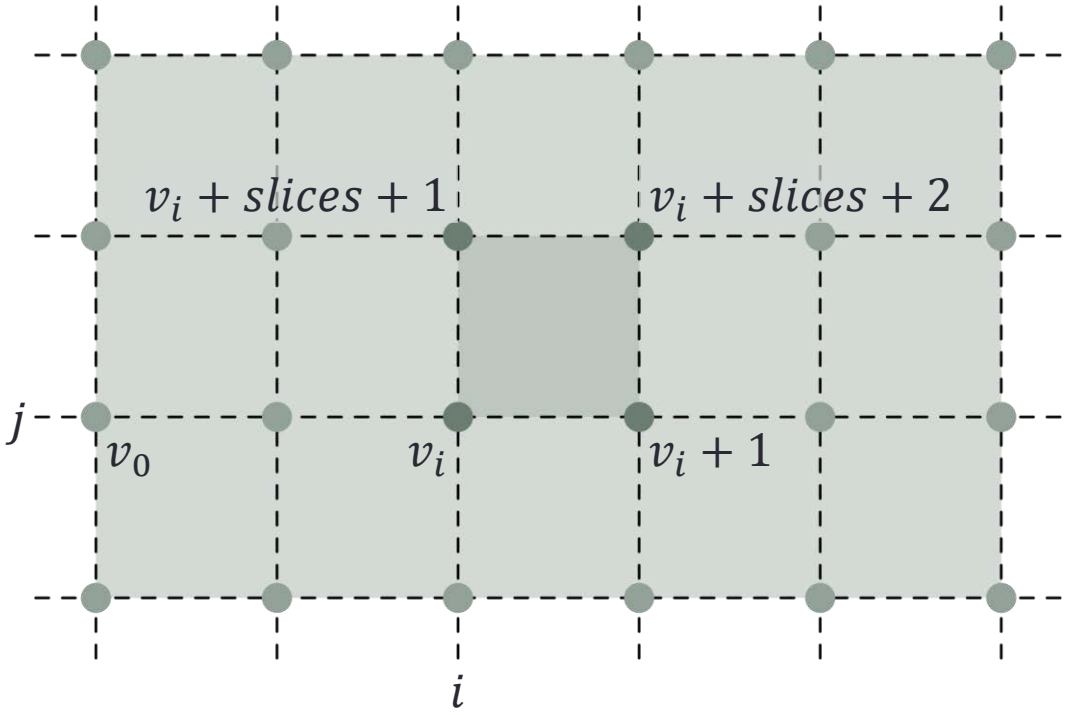
四角形ごとの頂点の格納先と頂点番号を求める

- 四角形の数は $slices \times stacks$ なので三角形の数は $slices \times stacks \times 2$ だから頂点の数 $nTriangles$ は
  - $nTriangles = stacks \times slices \times 6$
- 格納先は 1 段ごとに $slices \times 6$ ずれる
  - $f_0 = j \times slices \times 6$
- 頂点番号は 1 段ごとに $slices + 1$ 増す
  - $v_0 = j \times (slices + 1)$

```
        // 三角形の頂点の数
        int nTriangles = stacks * slices * 6;

        // 三角形のデータ
        int[] myTriangles = new int[nTriangles];

        // 三角形の頂点番号を求める
        for (int j = 0; j < stacks; ++j)
        {
            // 各段の左端の下側の頂点番号の格納先
            int f0 = j * slices * 6;

            // 各段の左端の下側の頂点の頂点番号
            int p0 = j * (slices + 1);
```

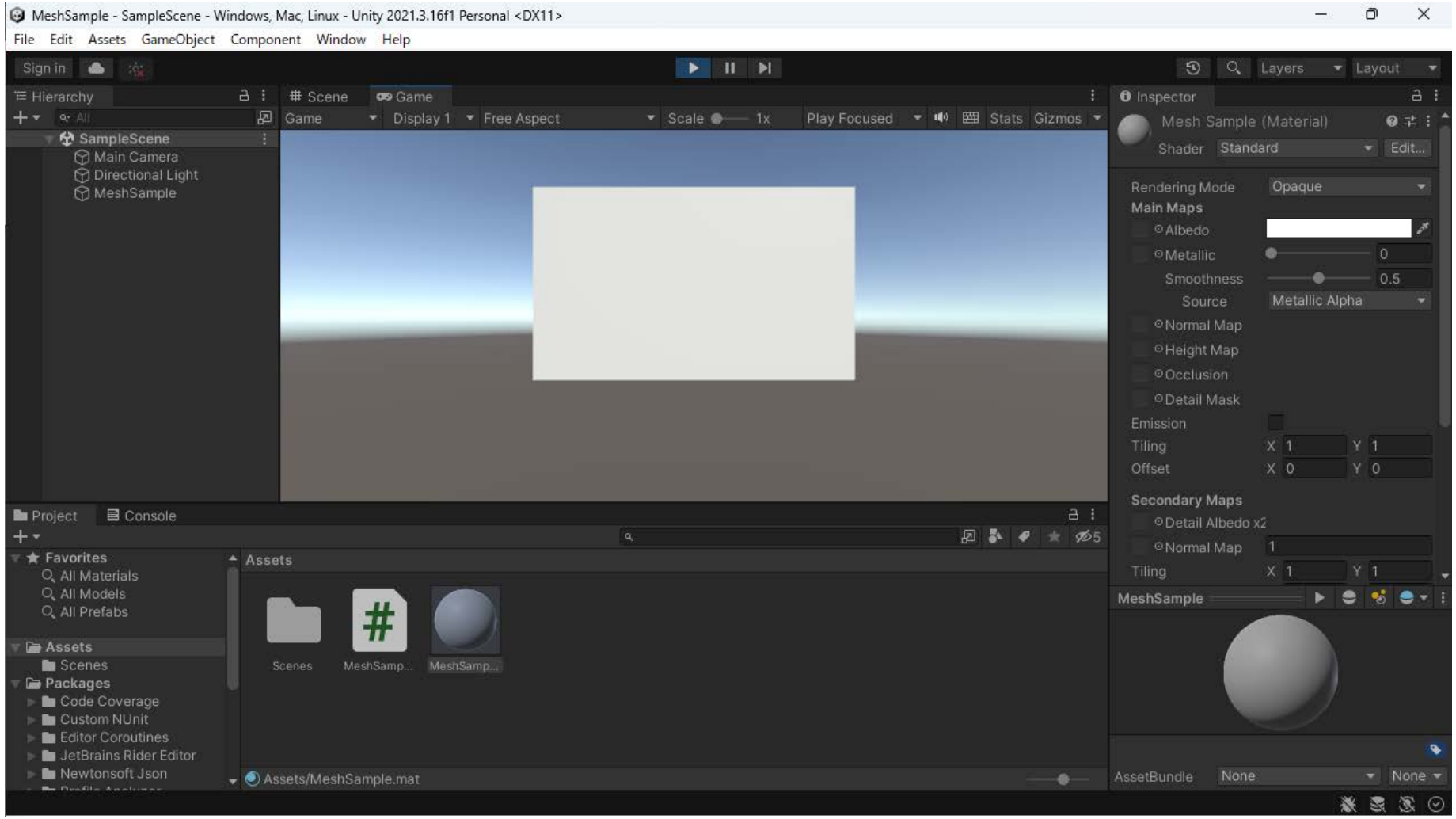
三角形の頂点番号を求める

- 頂点は 1 段当たり $slices + 1$ 個あるから頂点番号 v の頂点の真上の頂点の頂点番号は $v + slices + 1$

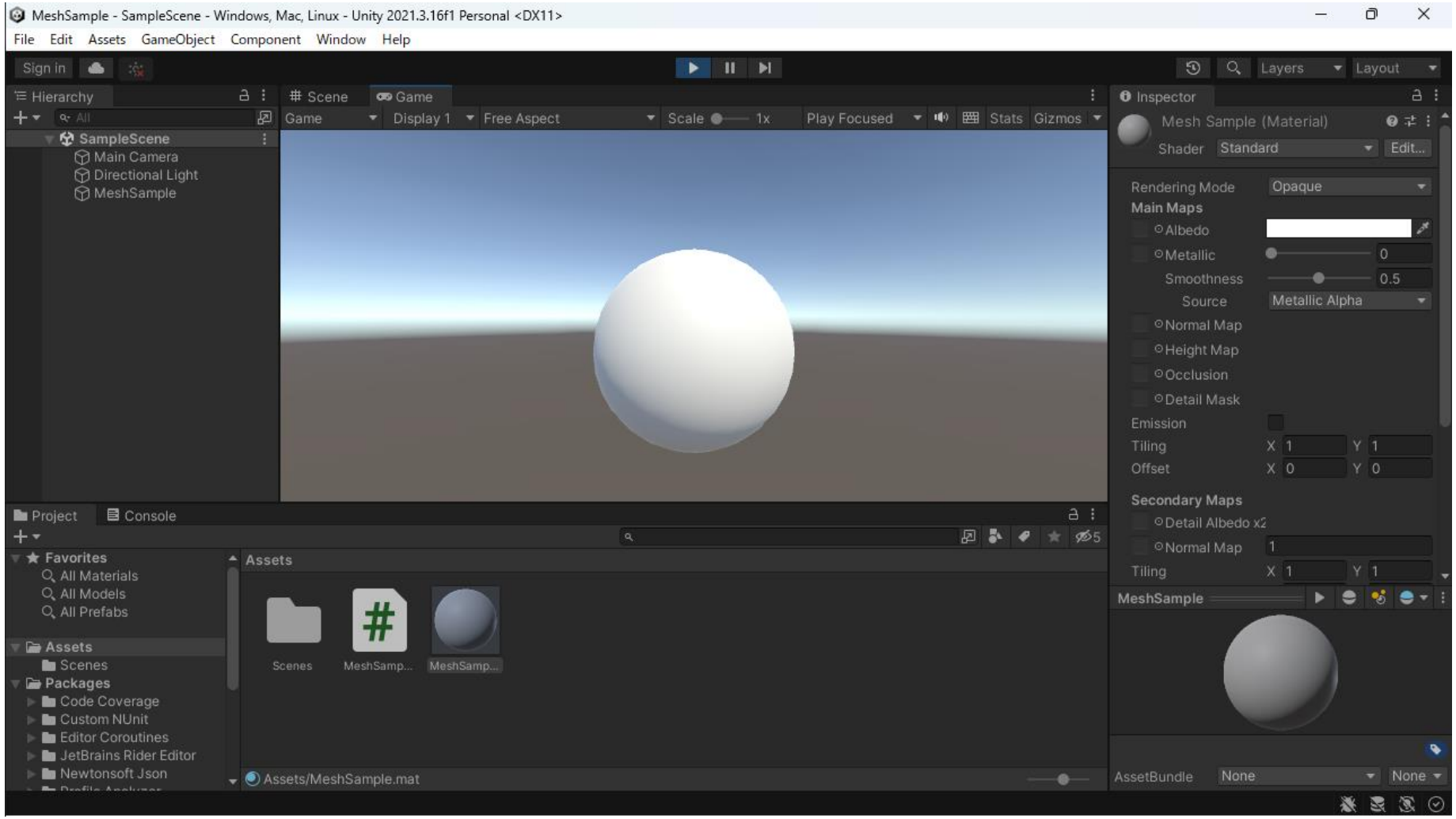



```
        for (int i = 0; i < slices; ++i)
        {
            // 左から i 番目の四角形の左下の頂点番号の格納先
            int fi = f0 + i * 6;

            // 左から i 番目の四角形の左下の頂点番号
            int pi = p0 + i;

            // １つ目の三角形の頂点番号
            myTriangles[fi + 0] = pi;
            myTriangles[fi + 1] = pi + slices + 1;
            myTriangles[fi + 2] = pi + 1;

            // ２つ目の三角形の頂点番号
            myTriangles[fi + 3] = pi + slices + 2;
            myTriangles[fi + 4] = pi + 1;
            myTriangles[fi + 5] = pi + slices + 1;
        }
    }

    // (以下略)
```

プロジェクトを実行する

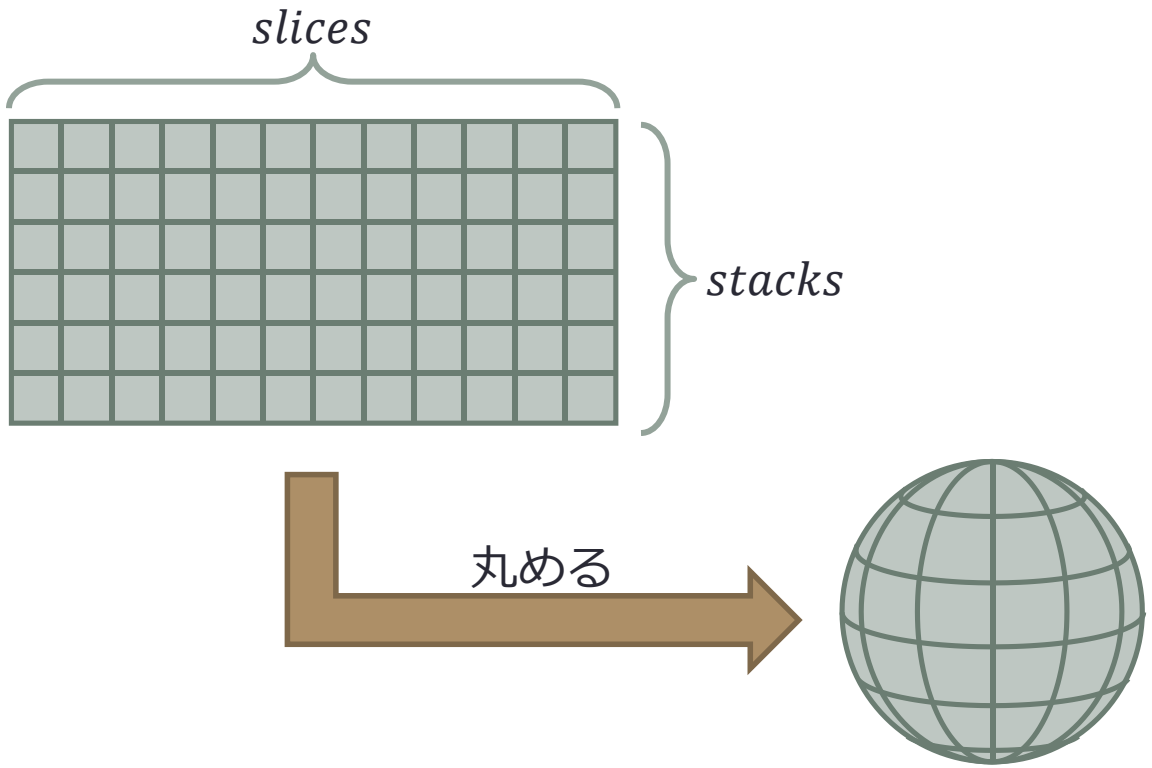


演習２：球を描いてください



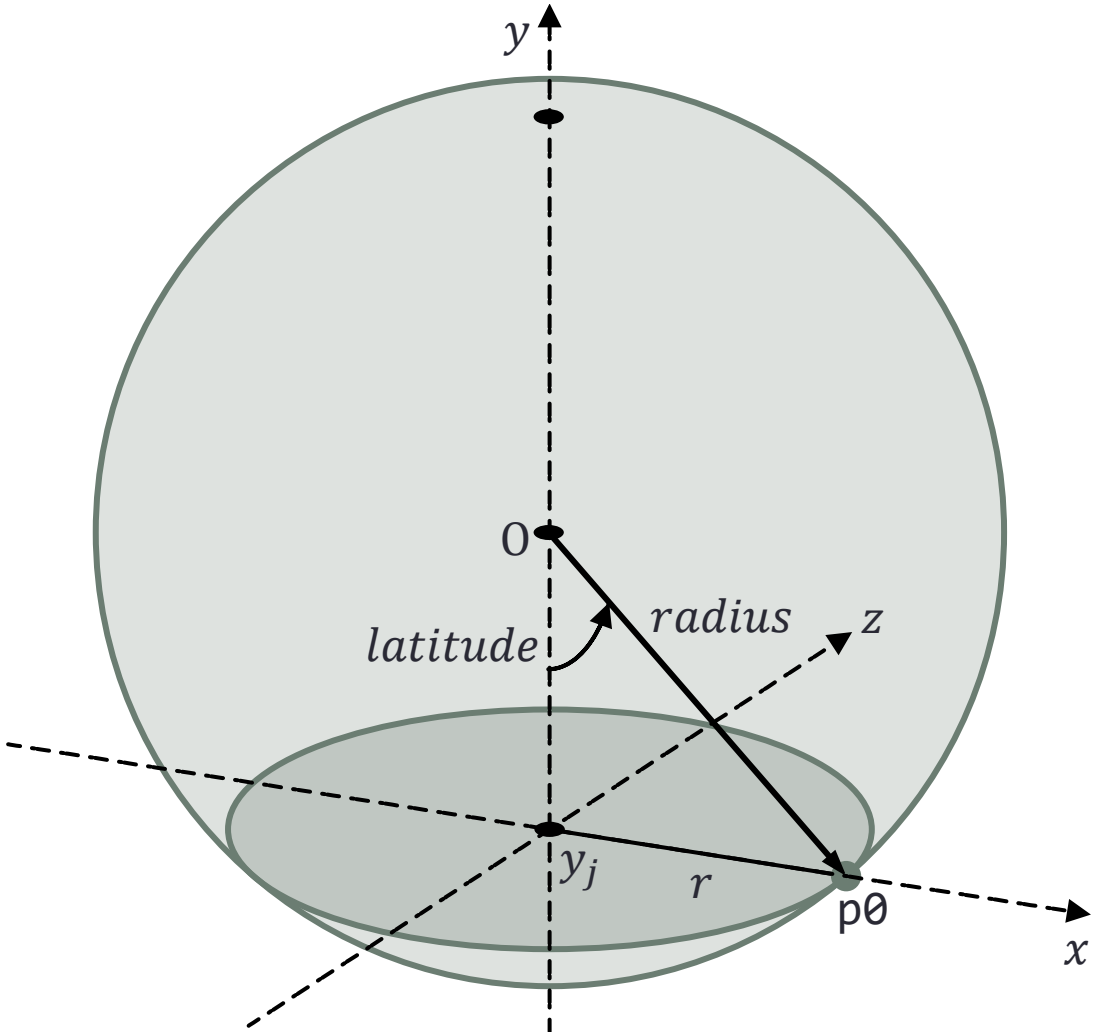
解答例 (1) 頂点数は四角形を縦横に並べた場合と同じ

- 四角形を縦横に並べたものを丸めるなら頂点の数はそれと同じです
  - $nVertices = (slice + 1) \times (stacks + 1)$



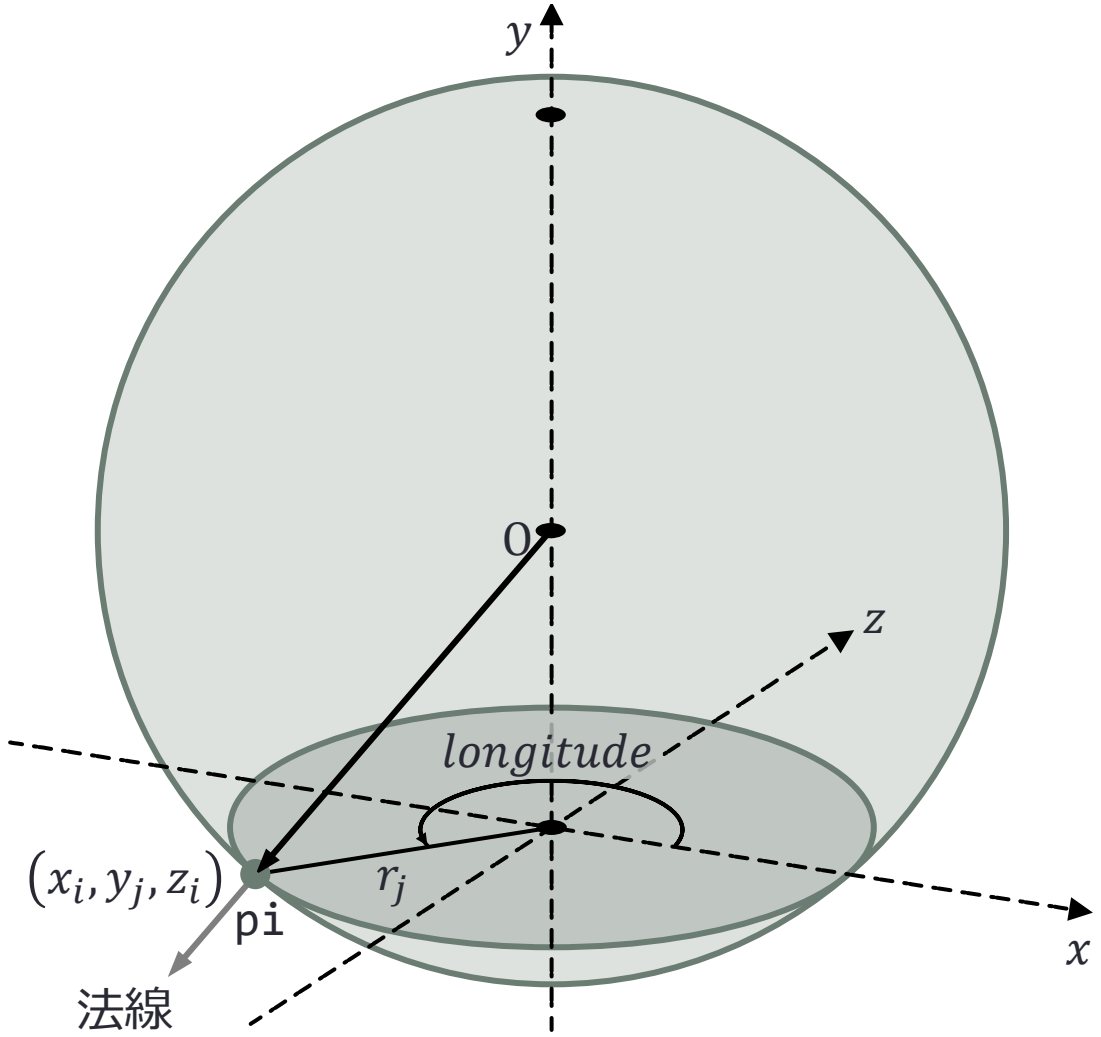

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class MeshSample : MonoBehapiour
{
    const int slices = 32;
    const int stacks = 16;
    const float radius = 3f;

    // Start is called before the first frame update
    void Start()
    {
        // 頂点の数
        int nVertices = (stacks + 1) * (slices + 1);

        // メッシュの頂点の位置の配列
        Vector3[] myVertices = new Vector3[nVertices];

        // メッシュの頂点の法線の配列
        Vector3[] myNormals = new Vector3[nVertices];
```

解答例 (2) 頂点の緯度から高さ y_j とその断面の半径 r を求める




```
// 各頂点の位置と法線を求める
for (int j = 0; j <= stacks; ++j)
{
    // 縦方向のパラメータ（0→1）
    float v = (float)j / stacks;

    // 各段の緯度（南極が 0）
    float latitude = Mathf.PI * v;

    // 各段の下側の頂点の高さ
    float yj = -radius * Mathf.Cos(latitude);

    // 各段の下段の半径
    float rj = radius * Mathf.Sin(latitude);

    // 各段の左端の下側の頂点の頂点番号
    int p0 = j * (slices + 1);
```

解答例 (3) 頂点の経度から y_j の断面上の位置 x_i , z_i を求める



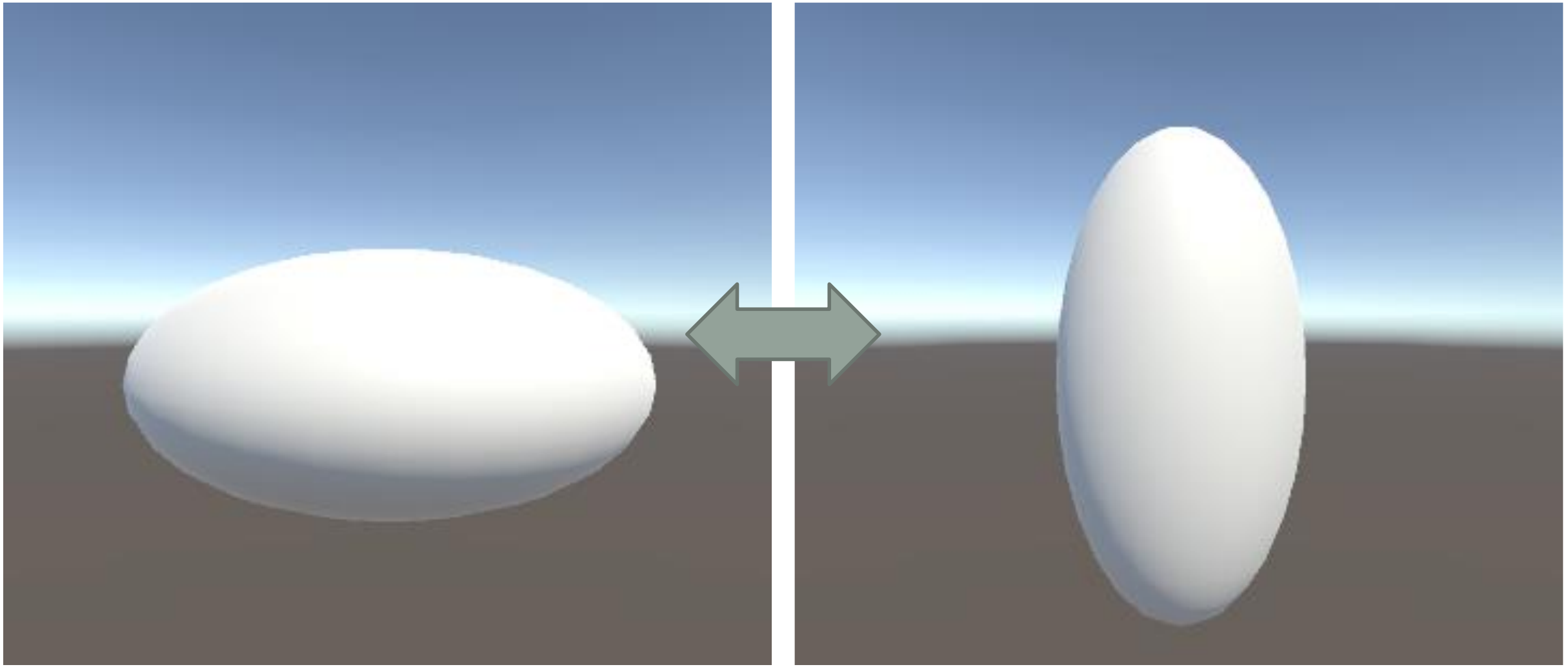

```
        for (int i = 0; i <= slices; ++i)
        {
            // 横方向のパラメータ (0→1)
            float u = (float)i / slices;

            // 経度
            float longitude = 2f * Mathf.PI * u;

            // 左から i 番目の四角形の左下の頂点の x 座標値
            float xi = rj * Mathf.Cos(longitude);

            // 左から i 番目の四角形の左下の頂点の z 座標値
            float zi = rj * Mathf.Sin(longitude);

            // 左から i 番目の四角形の左下の頂点の頂点番号
            int pi = p0 + i;

            // 左から i 番目の四角形の左下の頂点の位置と法線
            myVertices[pi].Set(xi, yj, zi);
            myNormals[pi] = myVertices[pi].normalized;
        }
    }

    // (以下略)
```

球面上の 1 点における法線は
球の中心からその点に向かう
ベクトルを正規化したもの

変形する

頂点の位置を移動する

時刻に応じて縦横に伸ばす



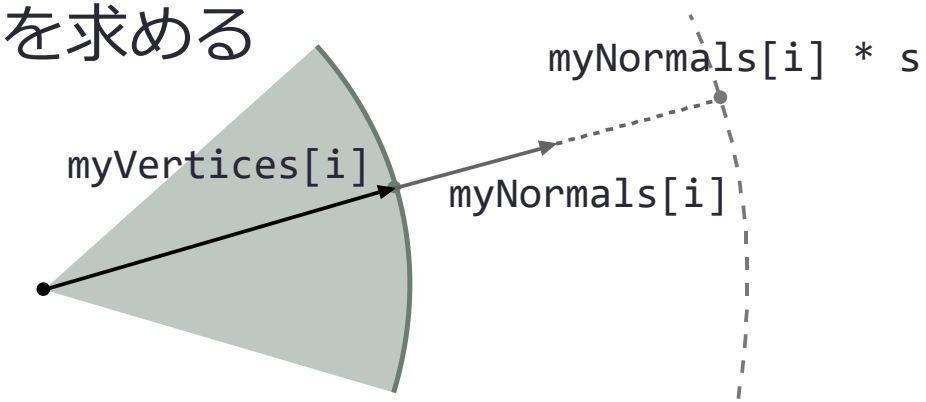
頂点と法線のデータをクラスのメンバにする

- myVertices と myNormals の変数宣言を Start() の外で行う
  - myVertices と myNormals はメンバ変数になるのでメソッド間で共有される
  - Start() で行っていた変数宣言は削除する

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

public class MeshSample : MonoBehaviour
{
    const int slices = 32;
    const int stacks = 16;
    const float radius = 3f;

    Vector3[] myVertices;
    Vector3[] myNormals;

    // Start is called before the first frame update
    void Start()
    {
        // 頂点の数
        int nVertices = (stacks + 1) * (slices + 1);

        // メッシュの頂点の位置の配列
        myVertices = new Vector3[nVertices];

        // メッシュの頂点の法線の配列
        myNormals = new Vector3[nVertices];
```

元の図形の各頂点の位置を移動する

- 頂点の位置 myVertices と法線 myNormals はメンバ変数
  - Start() で値を設定している
- 変更後の頂点位置を格納する配列を用意する
  - 要素数は元の頂点の数 myVertices.Length
- 元の頂点の位置を法線方向に s 動かした位置を求める



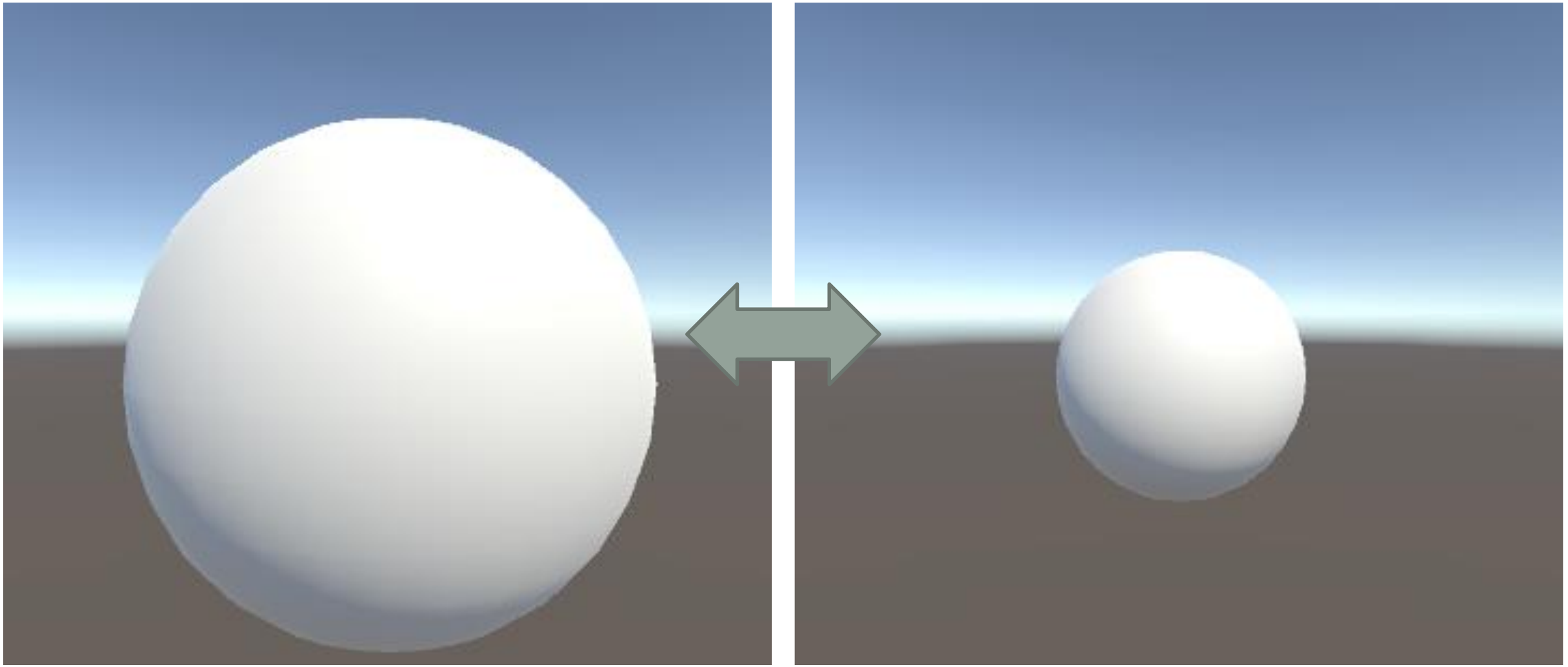

```
// Update is called once per frame
void Update()
{
    // 経過時間を取り出す
    float t = Time.time;

    // 経過時間にもとづいて変形率を決定する
    float s = Mathf.Sin(t * 20f);

    // 変更後の頂点位置の格納先
    Vector3[] vertices = new Vector3[myVertices.Length];

    // すべての頂点について
    for (int i = 0; i < myVertices.Length; ++i)
    {
        // 元の図形の各頂点の位置を法線方向に移動する
        vertices[i] = myVertices[i]
                    + myNormals[i] * s;
    }
```

GameObject のメッシュを更新する

- GameObject からメッシュを取り出す
  - GetComponent<MeshFilter>().mesh
- 取り出したメッシュの頂点の位置を更新した位置で置き換える
  - myMesh.SetVertices(vertices);
- 更新した位置における法線ベクトルを再計算する
  - Mesh::RecalculateNormals() メソッド

```
        // MeshFilter から Mesh のオブジェクトを取り出す
        Mesh myMesh = GetComponent<MeshFilter>().mesh;

        // Mesh の頂点のデータを挿げ替える
        myMesh.SetVertices(vertices);

        // 法線ベクトルを再計算する
        myMesh.RecalculateNormals();
    }
}
```

時刻に応じて全体的に拡大縮小するようになる

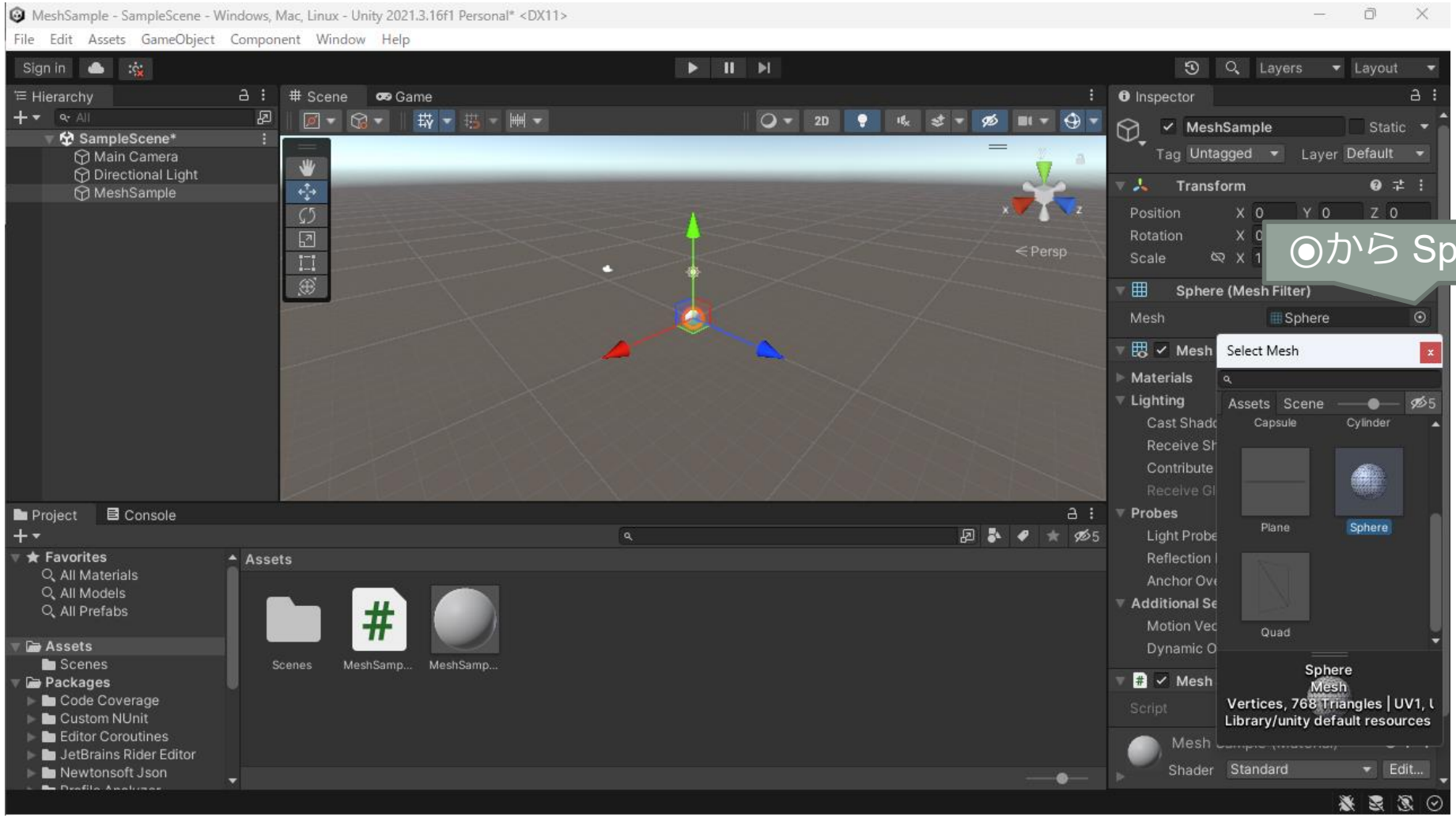


上下方向の変形率を反転する

- 変形率を Vector3 型にする
  - Vector3 型の変形率を r とする
  - r の x, z 成分は s とし y のみ –s とする
- 法線ベクトルを r 倍して元の頂点位置に加える
  - 二つの Vector3 型の要素同士の乗算には Vector3.Scale(Vector3 a, Vector3 b) を用いる

```
// Update is called once per frame
void Update()
{
    // 経過時間を取り出す
    float t = Time.time;

    // 経過時間にもとづいて変形率を決定する
    float s = Mathf.Sin(t * 20f);

    // 上下方向の変形率を反転する
    Vector3 r = new Vector3(s, -s, s);

    // 変更後の頂点位置
    Vector3[] vertices = new Vector3[myVertices.Length];

    // すべての頂点について
    for (int i = 0; i < myVertices.Length; ++i)
    {
        // 元の図形の各頂点の位置を法線方向に移動する
        vertices[i] = myVertices[i]
                    + Vector3.Scale(myNormals[i], r);
    }
```

プリミティブを変形する

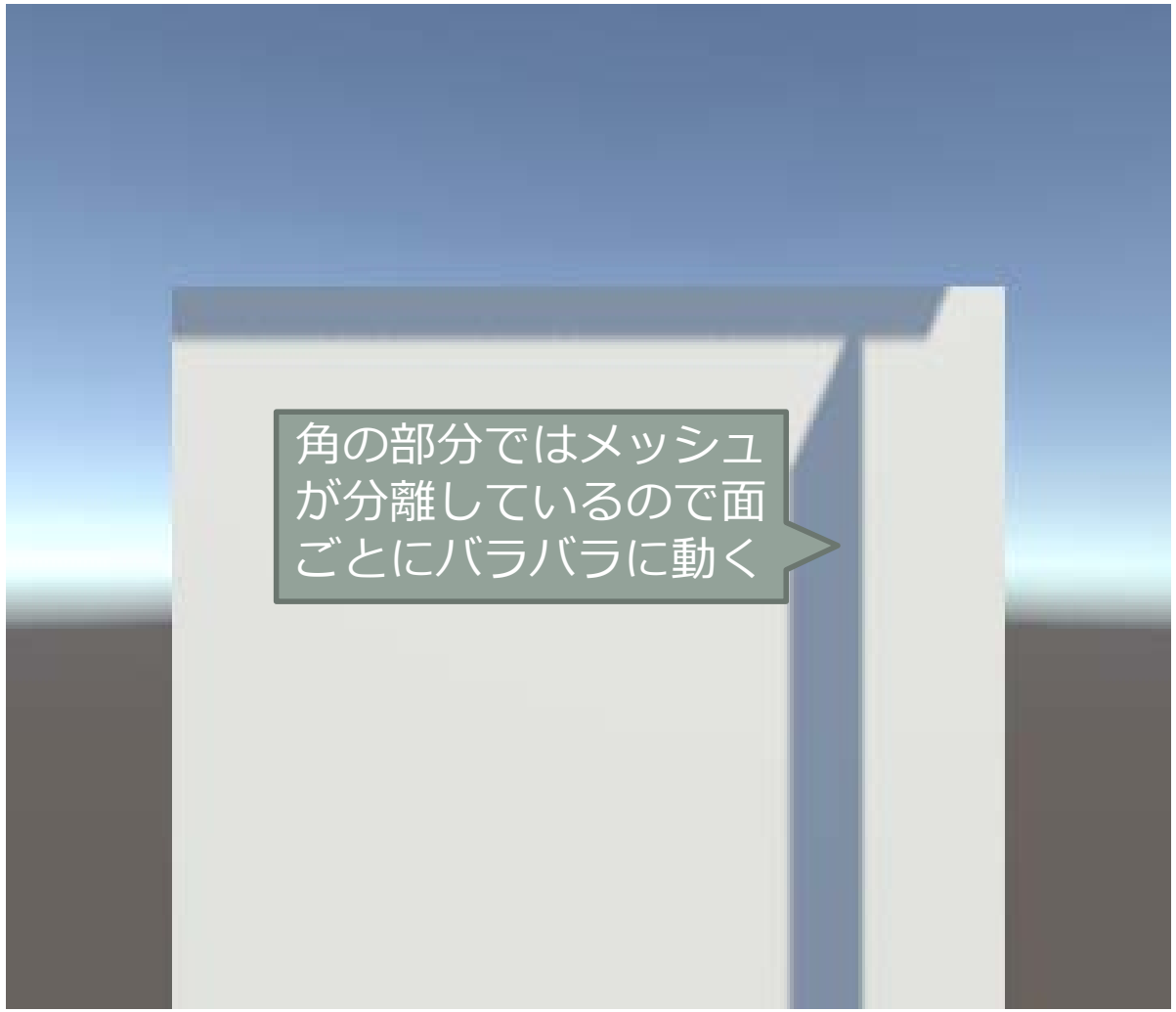


球のデータを生成するかわりにプリミティブのメッシュを使う

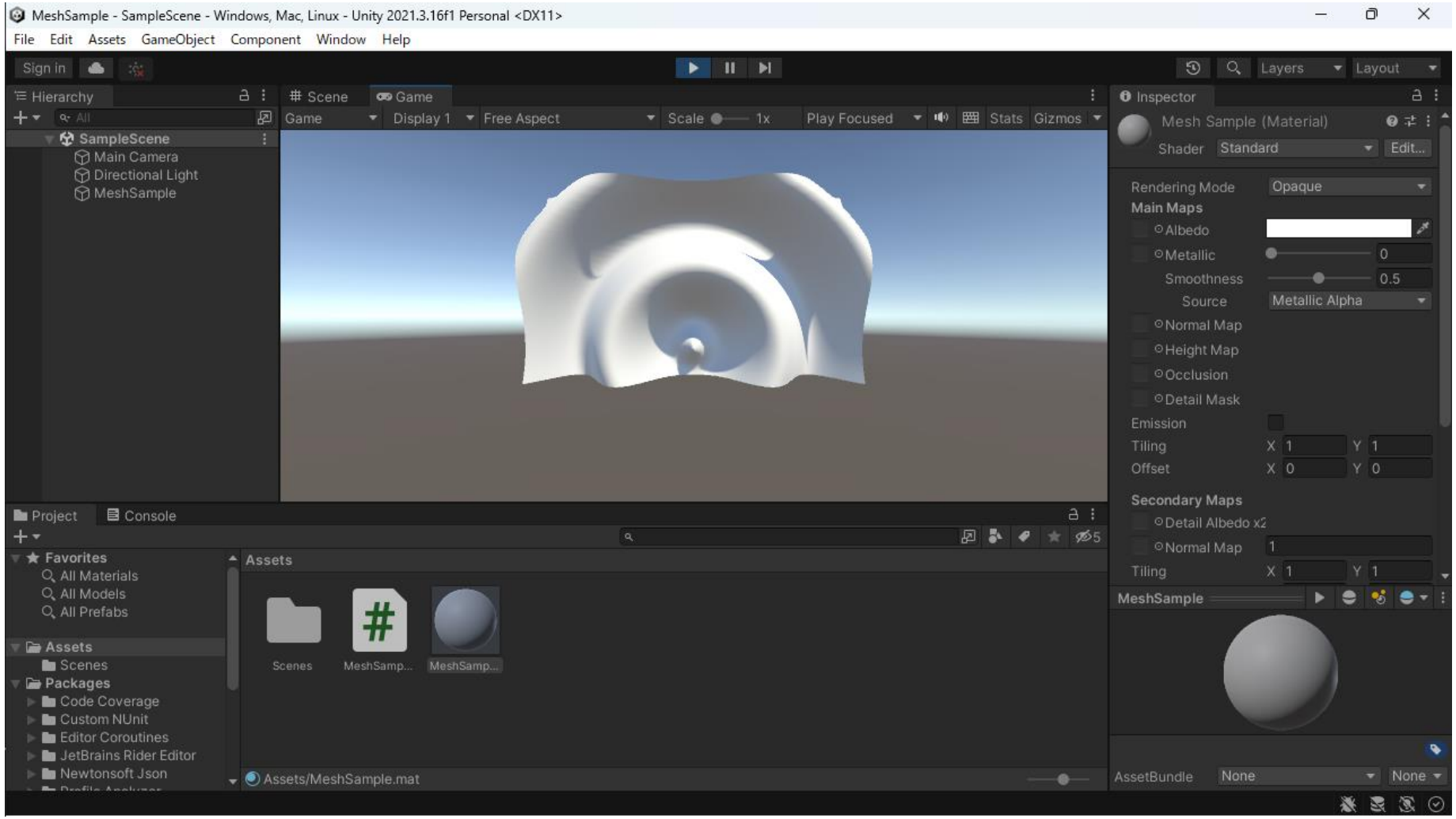
- 球のデータを作成する部分を削除する
- 代わりにゲームオブジェクト自体の MeshFilter からメッシュを取り出す
- そのメッシュの頂点の位置と法線を保存しておく
- 頂点の位置は自前で作成したときと同じくらいにスケーリングしておく
  - 6 倍くらい
- メッシュが接続されていない部分では陰影が連続しない（筋が出る）

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

public class MeshSample : MonoBehaviour
{
    Vector3[] myVertices;
    Vector3[] myNormals;

    // Start is called before the first frame update
    void Start()
    {
        // GameObject から MeshFilter の Component を取り出す
        MeshFilter meshFilter = GetComponent<MeshFilter>();

        // MeshFilter から Mesh を取り出す
        Mesh myMesh = meshFilter.mesh;

        // Mesh から頂点を取り出す
        myVertices = myMesh.vertices;

        // Mesh から法線を取り出す
        myNormals = myMesh.normals;

        // 頂点の位置をスケーリングする
        for (int i = 0; i < myVertices.Length; ++i) myVertices[i] *= 6f;
    }
```

演習 3 ：Sphere 以外のプリミティブを変形してください



演習４：四角形を縦横に並べた格子を波立たせてください



解答例 (1) myVertices と myNormals をメンバ変数にする

- 細かな変形になるのでメッシュの分割数 slices, stacks を多くしておきます
  - const int slices = 100;
  - const int stacks = 30;
- myVertices と myNormals をメンバ変数にします
  - Vector3[] myVertices;
  - Vector3[] myNormals;
- 変数の初期化を代入に変更します
  - myVertices = new Vector3[nVertices];
  - myNormals = new Vector3[nVertices];

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

public class MeshSample : MonoBehaviour
{
    const int slices = 100;
    const int stacks = 30;
    const float x0 = -5f;
    const float y0 = -1f;
    const float width = 10f;
    const float height = 6f;

    Vector3[] myVertices;
    Vector3[] myNormals;

    // Start is called before the first frame update
    void Start()
    {
        // 頂点の数
        int nVertices = (stacks + 1) * (slices + 1);

        // メッシュの頂点の位置の配列
        myVertices = new Vector3[nVertices];

        // メッシュの頂点の法線の配列
        myNormals = new Vector3[nVertices];
```

解答例 (2) 法線方向に頂点の位置を移動する

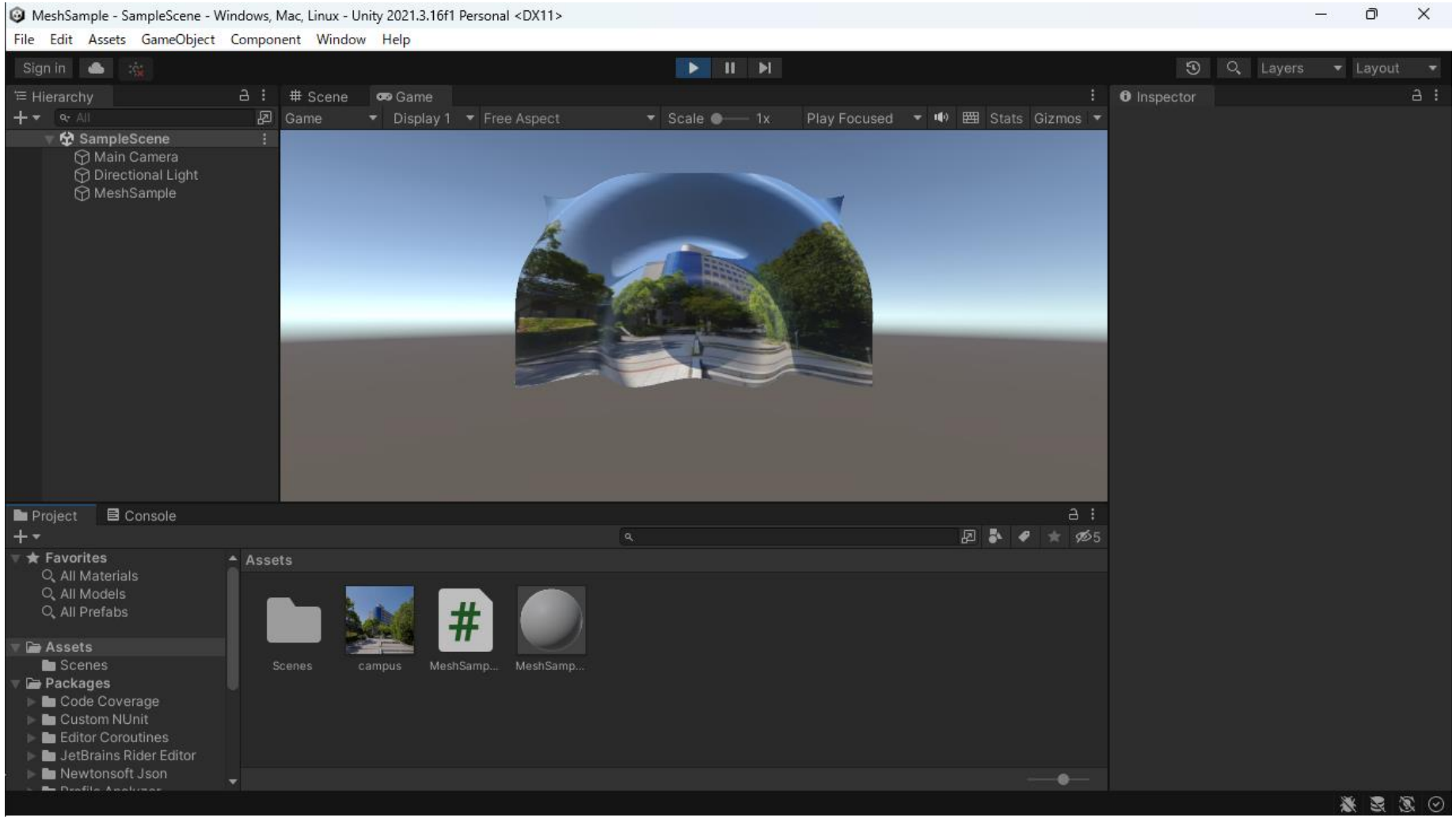
- 法線方向に頂点の位置を移動します
  - vertices[i] = myVertices[i] + myNormals[i] * r;
- この r を時間で変化させます
  - float r = Mathf.Sin(s);
  - こうすると平面全体が平行移動してしまう
- 場所によって移動する位相を変えます
  - float r =
    Mathf.Sin(s + myVertices[i].magnitude * 2f);


```
    // Update is called once per frame
    void Update()
    {
        // 経過時間を取り出す
        float t = Time.time;

        // 経過時間にもとづいて変形速度を決定する
        float s = t * 20f;

        // 変更後の頂点位置
        Vector3[] vertices = new Vector3[myVertices.Length];

        // すべての頂点について
        for (int i = 0; i < myVertices.Length; ++i)
        {
            // 経過時間にもとづいて変形率を決定する
            //float r = Mathf.Sin(s);
            //float r = Mathf.Sin(s + myVertices[i].x);
            //float r = Mathf.Sin(s + myVertices[i].magnitude);
            float r =
                Mathf.Sin(s + myVertices[i].magnitude * 2f);

            // 元の図形の各頂点の位置を法線方向に移動する
            vertices[i] = myVertices[i] + myNormals[i] * r;
        }

        // (以下略)
```

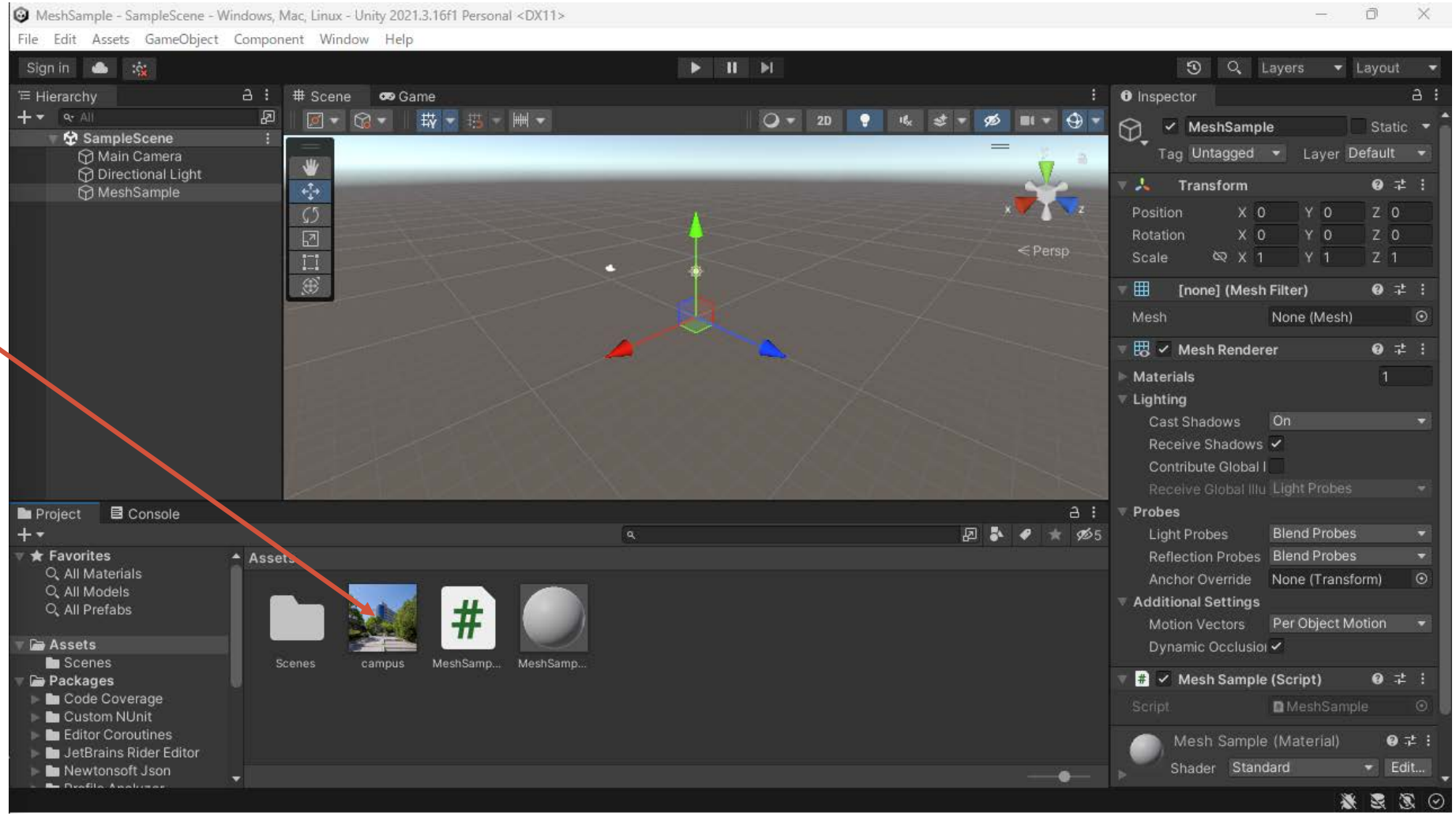
テクスチャを貼る

テクスチャ座標を設定する

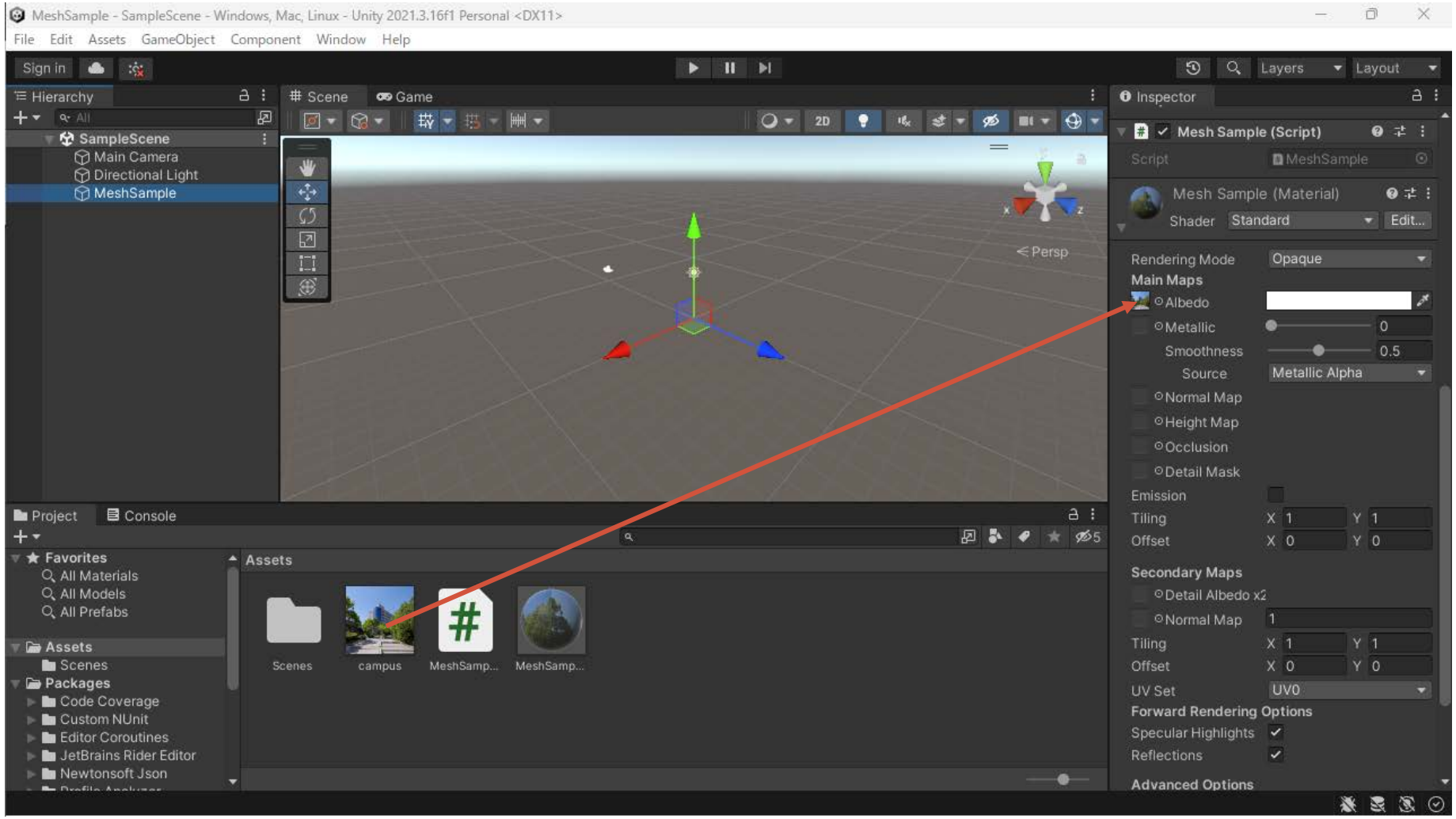
演習４の結果にテクスチャを貼る



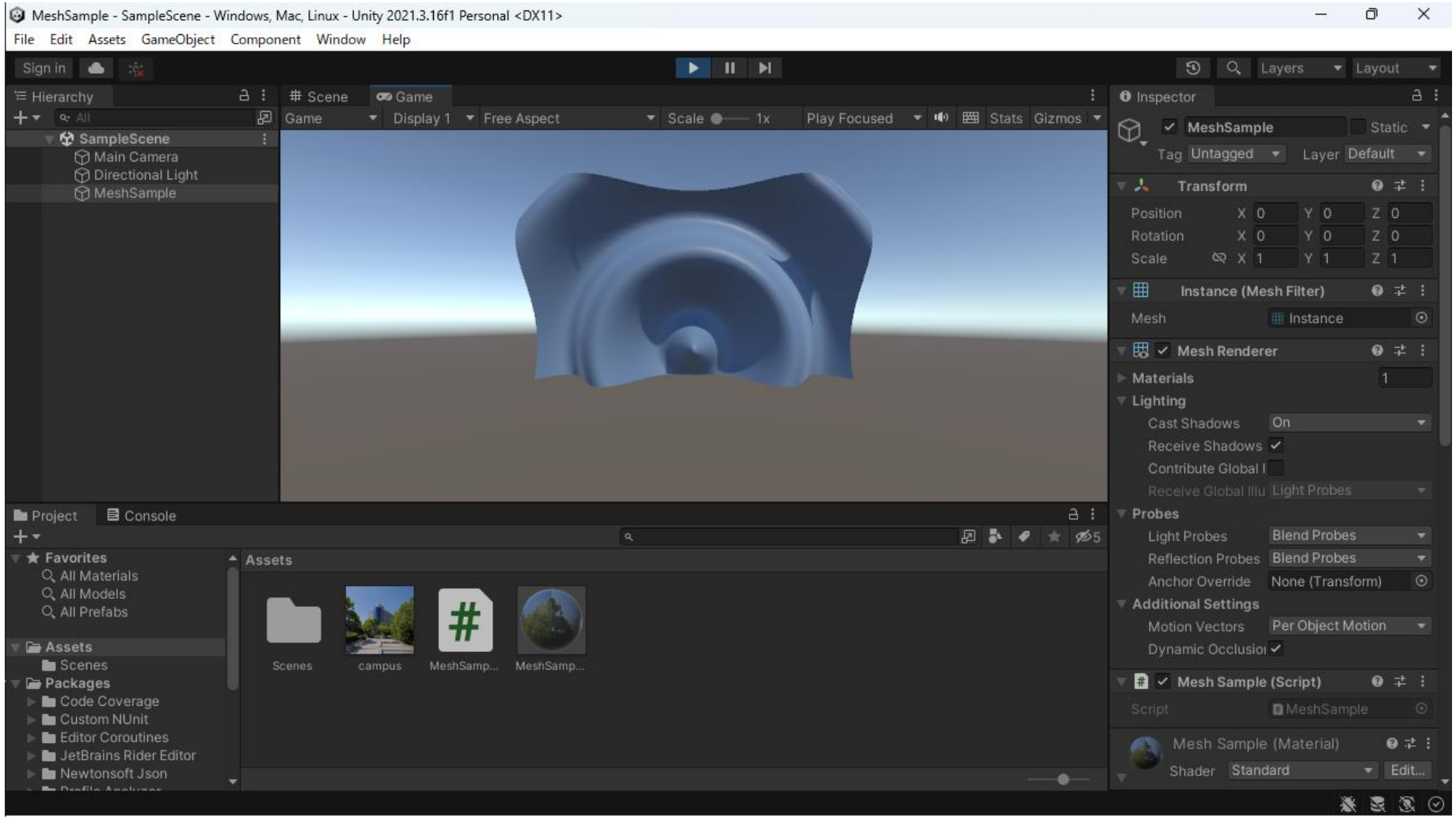
画像ファイルを Asset に追加する



画像を Material の Albedo に設定する

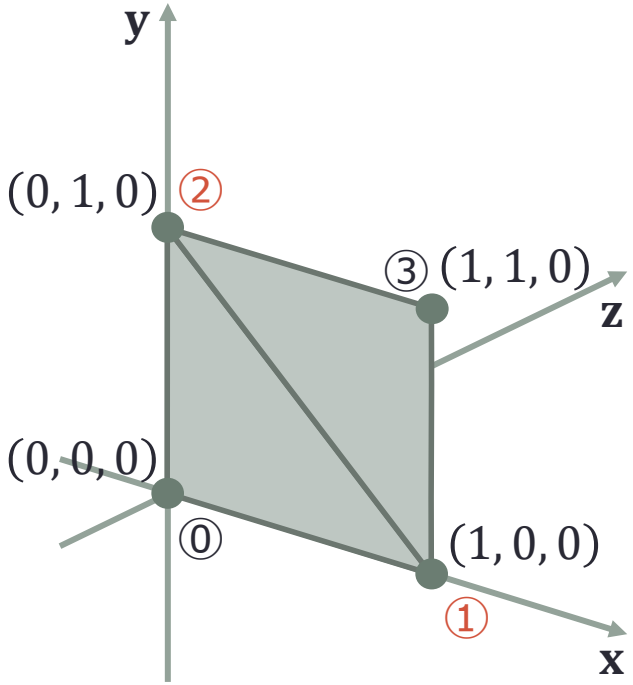


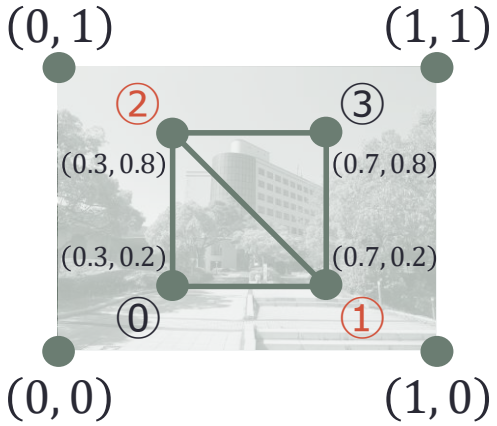
実行すると色は付くけどテクスチャ全体は貼られない



頂点にテクスチャ座標を設定する

- テクスチャ座標は頂点ごとに設定する
  - 共有されている頂点ではテクスチャ座標も共有される
  - テクスチャ座標の範囲は $0 \leq u, v \leq 1$（はみ出たら同じテクスチャが繰り返し貼られる）





テクスチャ空間

頂点アトリビュート

頂点番号	位置		
	x	y	z
⓪	0	0	0
①	1	0	0
②	0	1	0
③	1	1	0

頂点番号	法線		
	x	y	z
⓪	0	0	−1
①	0	0	−1
②	0	0	−1
③	0	0	−1

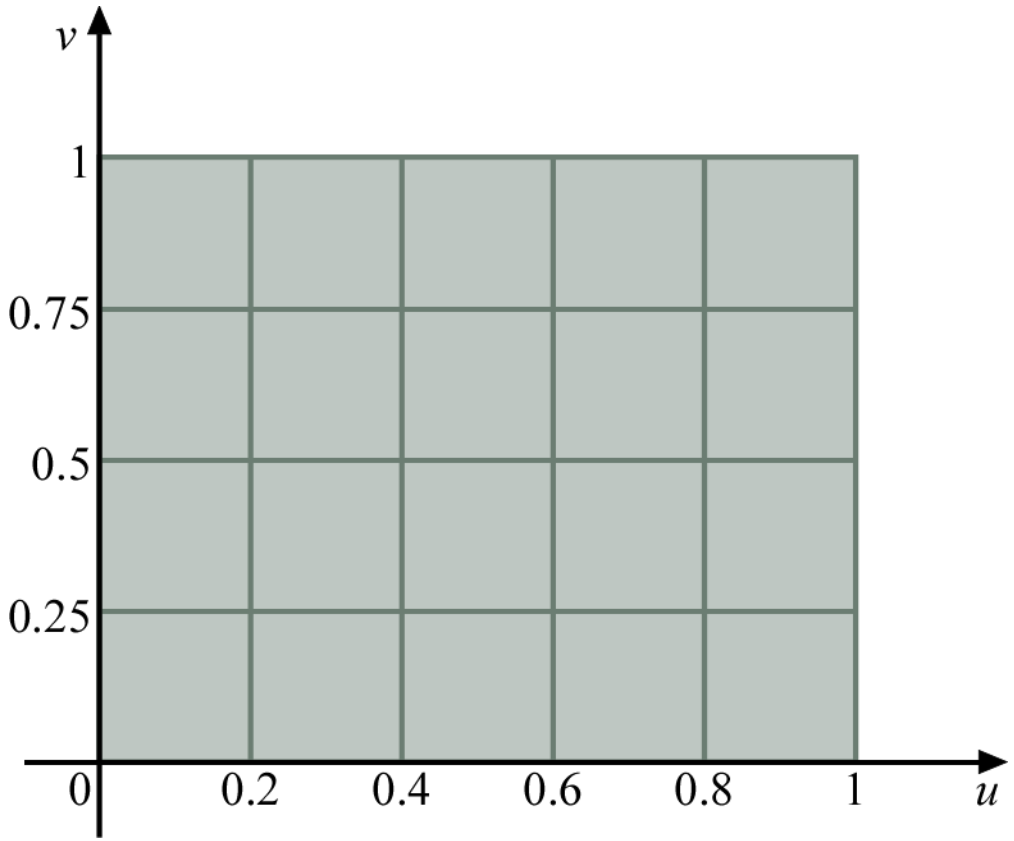
頂点番号	テクスチャ座標	
	u	v
⓪	0.3	0.2
①	0.7	0.2
②	0.3	0.8
③	0.7	0.8

テクスチャ座標の格納先

- テクスチャ座標を格納する配列を用意する
  - Vector2[] myUVs = new Vector2[nVertices];
  - この場合のテクスチャ座標は２次元

```
// Start is called before the first frame update
void Start()
{
    // 頂点の数
    int nVertices = (stacks + 1) * (slices + 1);

    // メッシュの頂点の位置の配列
    myVertices = new Vector3[nVertices];

    // メッシュの頂点の法線の配列
    myNormals = new Vector3[nVertices];

    // メッシュのテクスチャ座標
    Vector2[] myUVs = new Vector2[nVertices];

    // 各頂点の位置と法線を求める
    for (int j = 0; j <= stacks; ++j)
    {
        // 縦方向のパラメータ
        float v = (float)j / stacks;

        // 各段の下側の頂点の高さ
        float yj = y0 + height * v;

        // 各段の左端の下側の頂点の頂点番号
        int p0 = j * (slices + 1);
```

テクスチャ座標は矩形を縦横に等分分割したときの比を用いる




```
            for (int i = 0; i <= slices; ++i)
            {
                // 横方向のパラメータ
                float u = (float)i / slices;

                // 左から i 番目の四角形の左下の頂点の x 座標値
                float xi = x0 + width * u;

                // 左から i 番目の四角形の左下の頂点の頂点番号
                int pi = p0 + i;

                // 左から i 番目の四角形の左下の頂点の位置と法線
                myVertices[pi].Set(xi, yj, 0);
                myNormals[pi] = Vector3.back;

                // 左から i 番目の四角形の左下の頂点のテクスチャ座標
                myUVs[pi].Set(u, v);
            }
        }
```

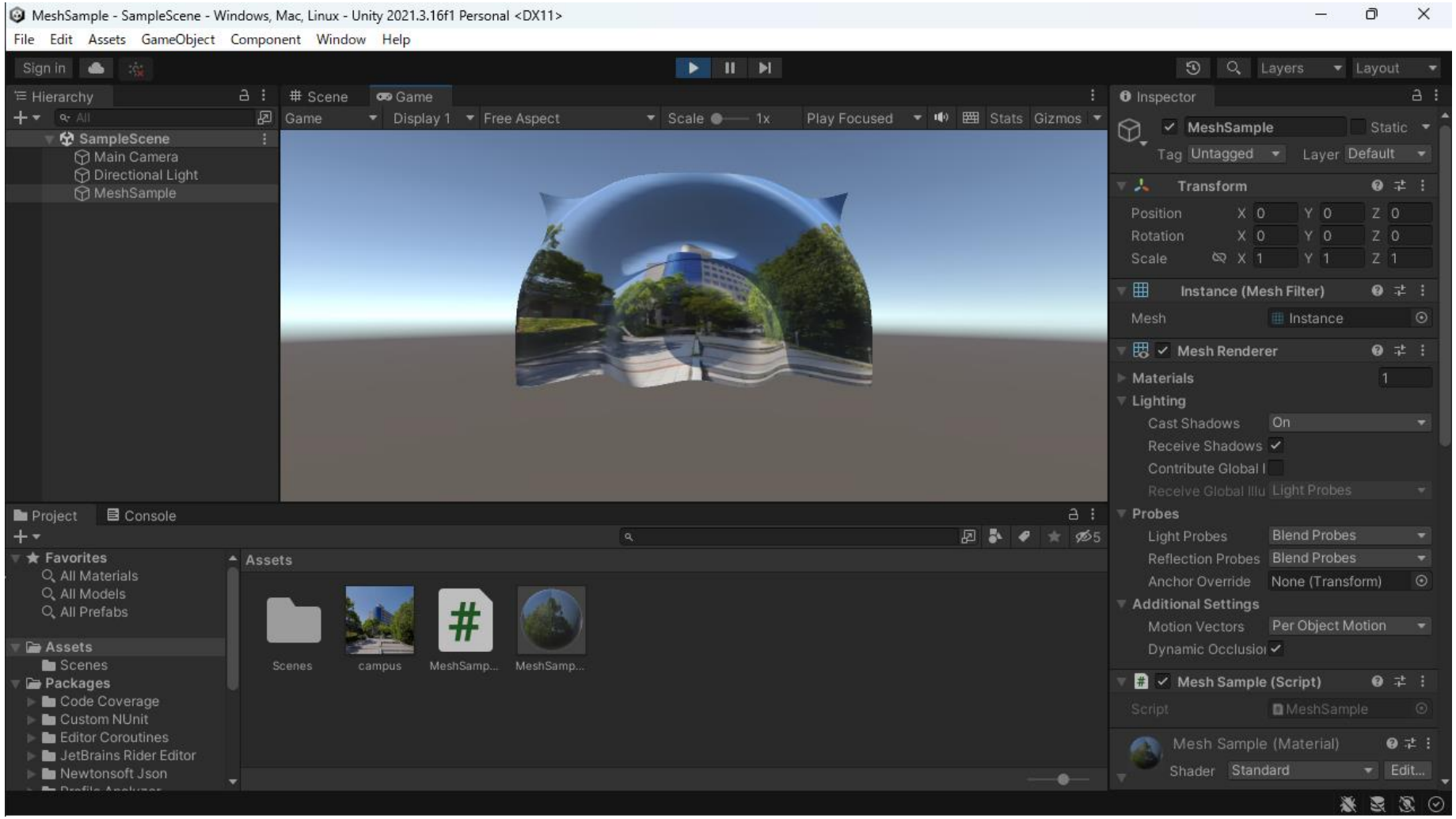
メッシュにテクスチャ座標を設定する

- 頂点アトリビュート
  - 位置 (myVertices)
  - 法線 (myNormals)
  - テクスチャ座標 (myUVs)
- 求めたテクスチャ座標 myUVs を myMesh に設定する
  - myMesh.SetUVs(0, myUVs);

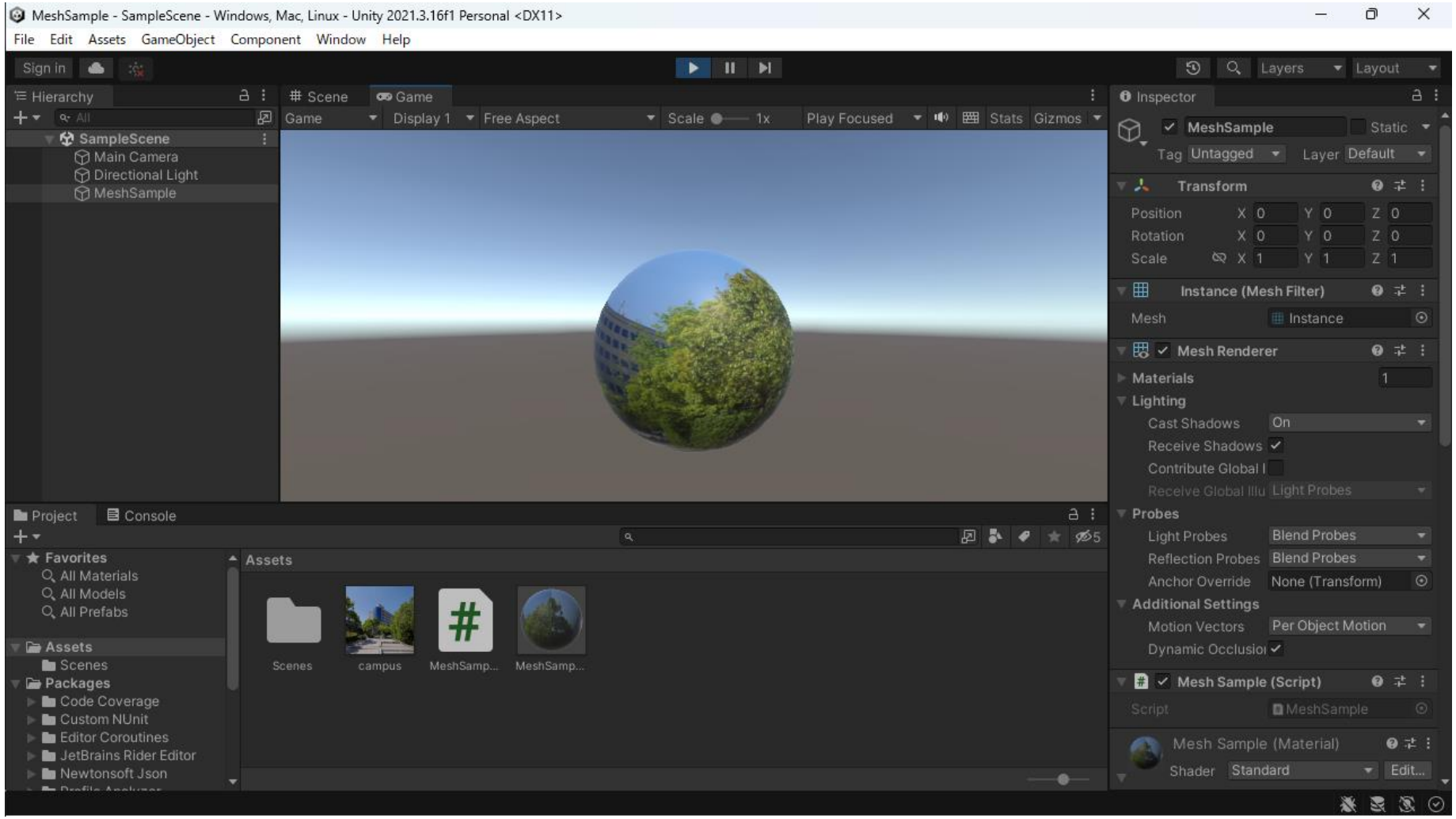
channel
（テクスチャを参照するときの識別子）

```
        // メッシュのオブジェクトを作る
        Mesh myMesh = new Mesh();

        // Mesh のオブジェクトに頂点の位置を設定する
        myMesh.SetVertices(myVertices);

        // Mesh のオブジェクトに頂点の法線を設定する
        myMesh.SetNormals(myNormals);

        // Mesh のオブジェクトに頂点のテクスチャ座標を設定する
        myMesh.SetUVs(0, myUVs);

        // Mesh のオブジェクトに頂点インデックスを設定する
        myMesh.SetTriangles(myTriangles, 0);

        // GameObject から MeshFilter の Component を取り出す
        MeshFilter meshFilter = GetComponent<MeshFilter>();

        // MeshFilter に Mesh のオブジェクトを設定する
        meshFilter.mesh = myMesh;
    }
```

プロジェクトを実行する



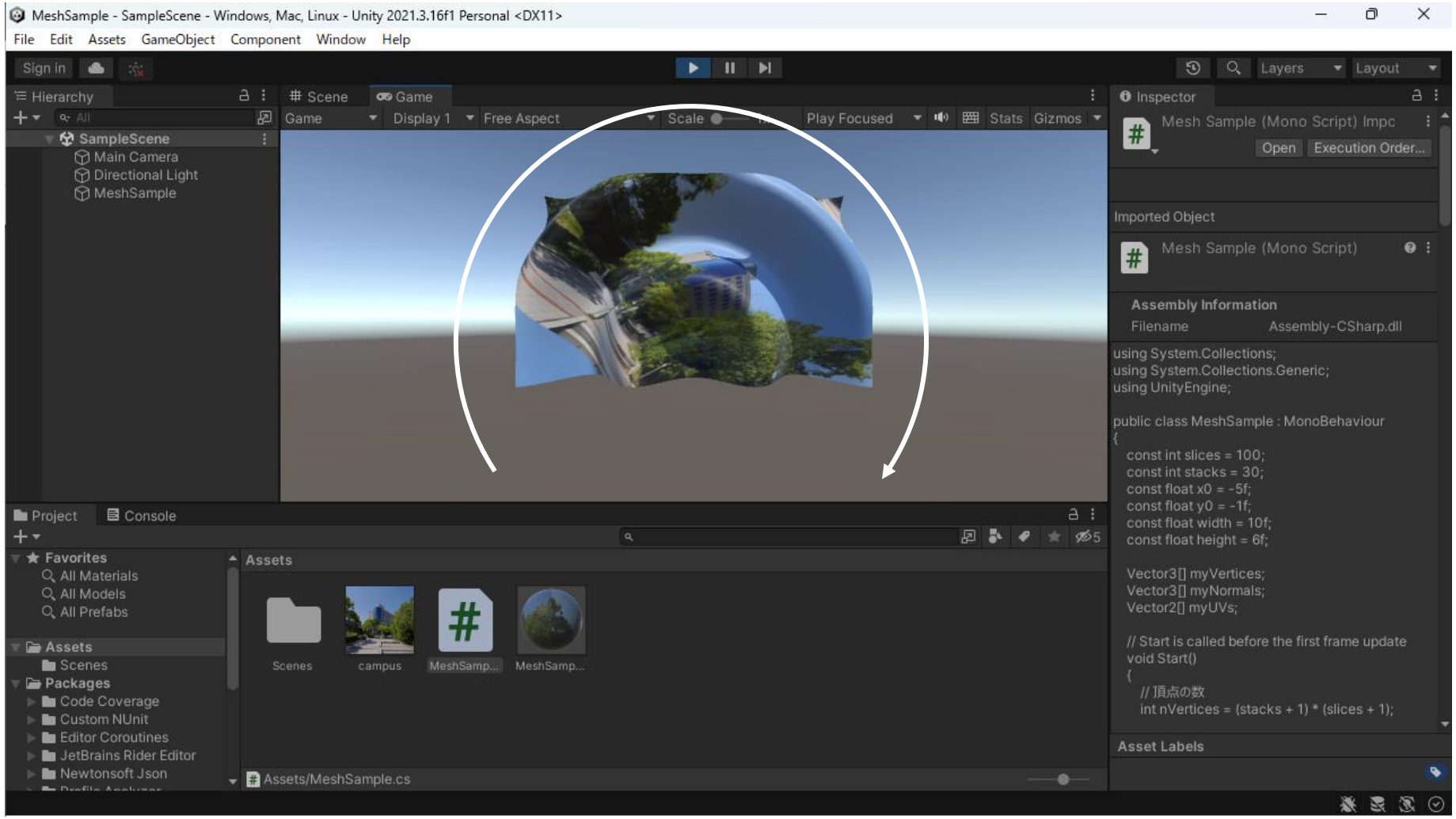
演習５：球にテクスチャを貼ってください



テクスチャを動かす

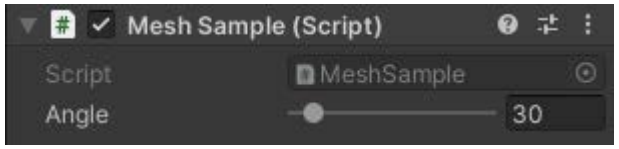
テクスチャ座標を更新する

テクスチャを回転して貼り付ける



回転角を設定する

- 回転角を設定する変数を public メンバとして追加する
  - [Range(0, 360)] public float angle = 30f;
  - インスペクタから値を設定できる
  - Range 属性を付けるとスライダになる




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

public class MeshSample : MonoBehaviour
{
    const int slices = 100;
    const int stacks = 30;
    const float x0 = -5f;
    const float y0 = -1f;
    const float width = 10f;
    const float height = 6f;

    [Range(0, 360)] public float angle = 30f;

    Vector3[] myVertices;
    Vector3[] myNormals;

    // Start is called before the first frame update
    void Start()
    {
        // 頂点の数
        int nVertices = (stacks + 1) * (slices + 1);
```

回転の変換を求める

- 角度を度で設定している場合はラジアンに直す
  - float theta = angle * Mathf.Deg2Rad;
- 回転の変換を求める
  - float cosTheta = Mathf.Cos(theta);
  - float sinTheta = Mathf.Sin(theta);

```
        // メッシュの頂点の位置の配列
        myVertices = new Vector3[nVertices];

        // メッシュの頂点の法線の配列
        myNormals = new Vector3[nVertices];

        // メッシュのテクスチャ座標
        Vector2[] myUVs = new Vector2[nVertices];

        // 角度をラジアンに変換する
        float theta = angle * Mathf.Deg2Rad;

        // 回転の変換
        float cosTheta = Mathf.Cos(theta);
        float sinTheta = Mathf.Sin(theta);

        // 各頂点の位置と法線を求める
        for (int j = 0; j <= stacks; ++j)
        {
            // 縦方向のパラメータ (0→1)
            float v = (float)j / stacks;

            // 各段の下側の頂点の高さ
            float yj = y0 + height * v;

            // 各段の左端の下側の頂点の頂点番号
            int p0 = j * (slices + 1);
```

テクスチャ座標を回転する

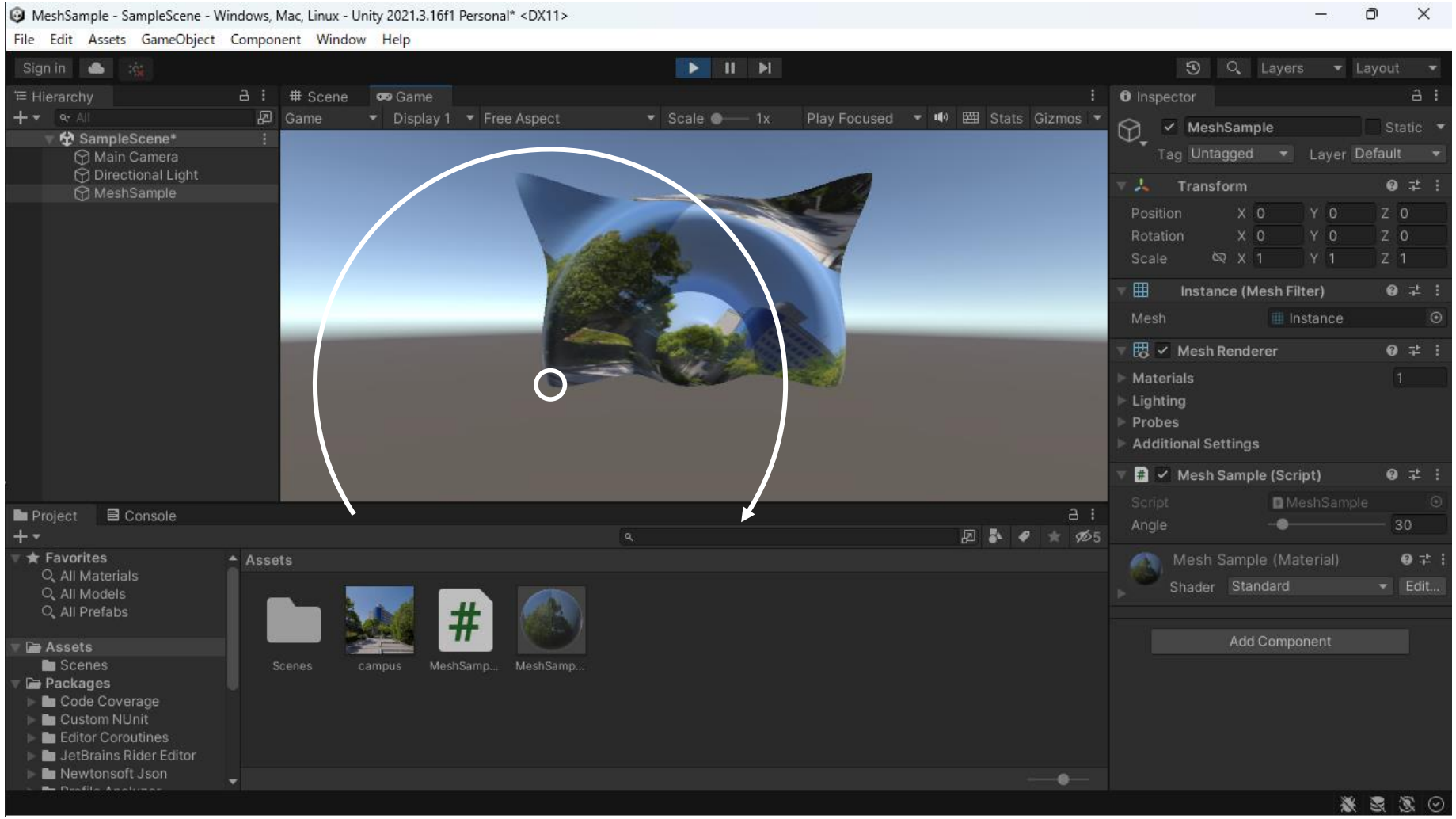
- テクスチャ座標に回転の変換を乗じる

$$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix} \begin{pmatrix} u \\ v \end{pmatrix}$$

  - float x = cosTheta * u – sinTheta * v;
  - float y = sinTheta * u + cosTheta * v;
- 回転したテクスチャ座標を設定する
  - myUVs[pi].Set(x, y);

```
            for (int i = 0; i <= slices; ++i)
            {
                // 横方向のパラメータ (0→1)
                float u = (float)i / slices;

                // 左から i 番目の四角形の左下の頂点の x 座標値
                float xi = x0 + width * u;

                // 左から i 番目の四角形の左下の頂点の頂点番号
                int pi = p0 + i;

                // 左から i 番目の四角形の左下の頂点の位置と法線
                myVertices[pi].Set(xi, yj, 0);
                myNormals[pi] = Vector3.back;

                // テクスチャ座標を回転する
                float x = cosTheta * u - sinTheta * v;
                float y = sinTheta * u + cosTheta * v;

                // 左から i 番目の四角形の左下の頂点のテクスチャ座標
                myUVs[pi].Set(x, y);
            }
        }
```

テクスチャ座標の原点の左下を中心に回転してしまう



回転の中心が原点なるよう平行移動してから回転する

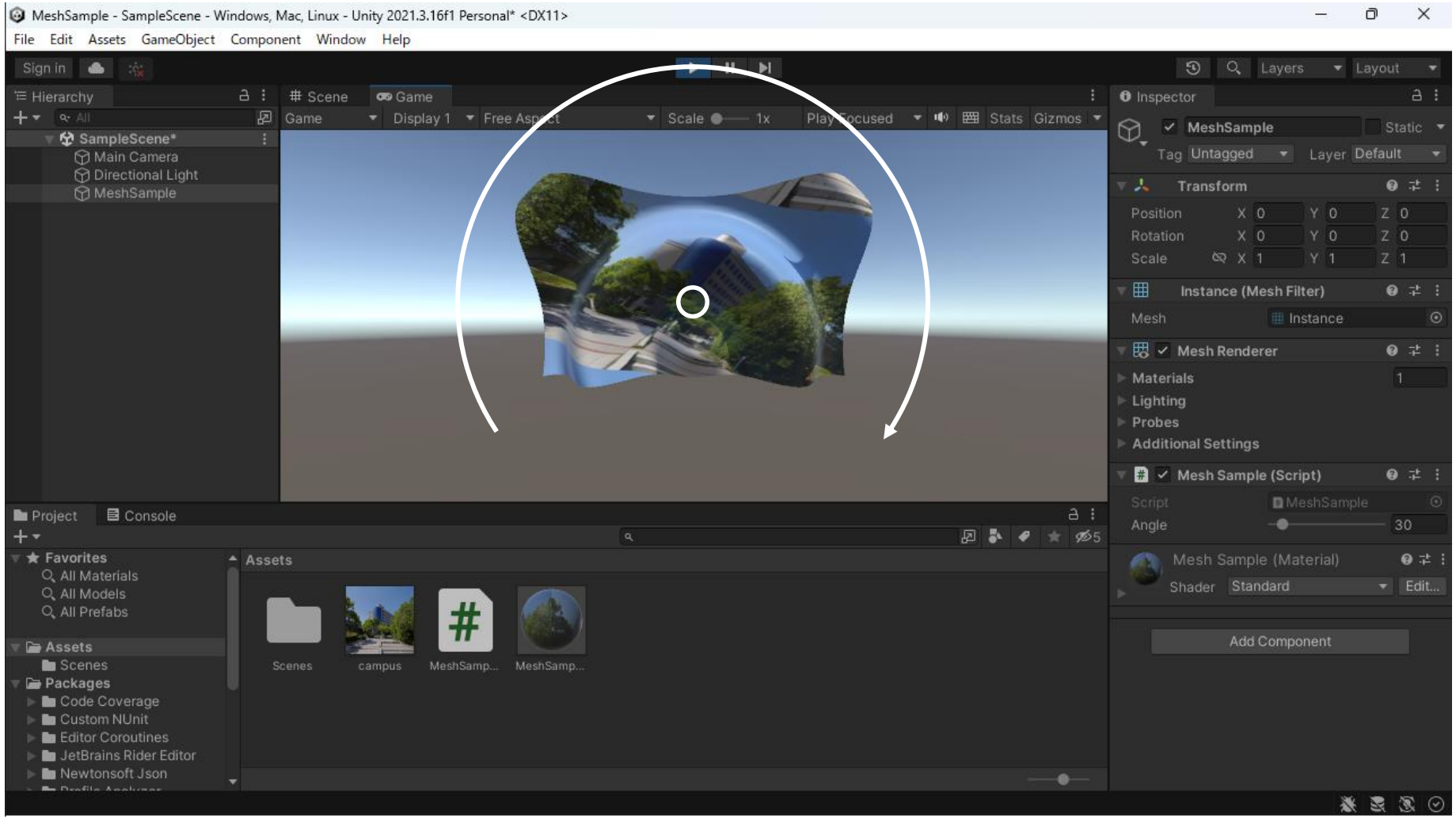
- テクスチャの中心が原点になるよう平行移動してから回転の変換を乗じる

$$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix} \left\{ \begin{pmatrix} u \\ v \end{pmatrix} - \begin{pmatrix} 0.5 \\ 0.5 \end{pmatrix} \right\} + \begin{pmatrix} 0.5 \\ 0.5 \end{pmatrix}$$

  - float s = u – 0.5f;
  - float t = v – 0.5f;
  - float x = cosTheta * s – sinTheta * t;
  - float y = sinTheta * s + cosTheta * t;
- 回転したテクスチャ座標を元の位置に戻して設定する
  - myUVs[pi].Set(x + 0.5f, y + 0.5f);

```
            for (int i = 0; i <= slices; ++i)
            {
                // 横方向のパラメータ (0→1)
                float u = (float)i / slices;

                // 左から i 番目の四角形の左下の頂点の x 座標値
                float xi = x0 + width * u;

                // 左から i 番目の四角形の左下の頂点の頂点番号
                int pi = p0 + i;

                // 左から i 番目の四角形の左下の頂点の位置と法線
                myVertices[pi].Set(xi, yj, 0);
                myNormals[pi] = Vector3.back;

                // 回転の中心が原点になるよう平行移動する
                float s = u - 0.5f;
                float t = v - 0.5f;

                // テクスチャ座標を回転する
                float x = cosTheta * s - sinTheta * t;
                float y = sinTheta * s + cosTheta * t;

                // 左から i 番目の四角形の左下の頂点のテクスチャ座標
                myUVs[pi].Set(x + 0.5f, y + 0.5f);
            }
        }
```

テクスチャの中心を中心にして回転する



四元数を使って回転する

- 回転の中心の位置を Vector2 型の変数に格納しておく
  - Vector2 c = new Vector2(0.5f, 0.5f);

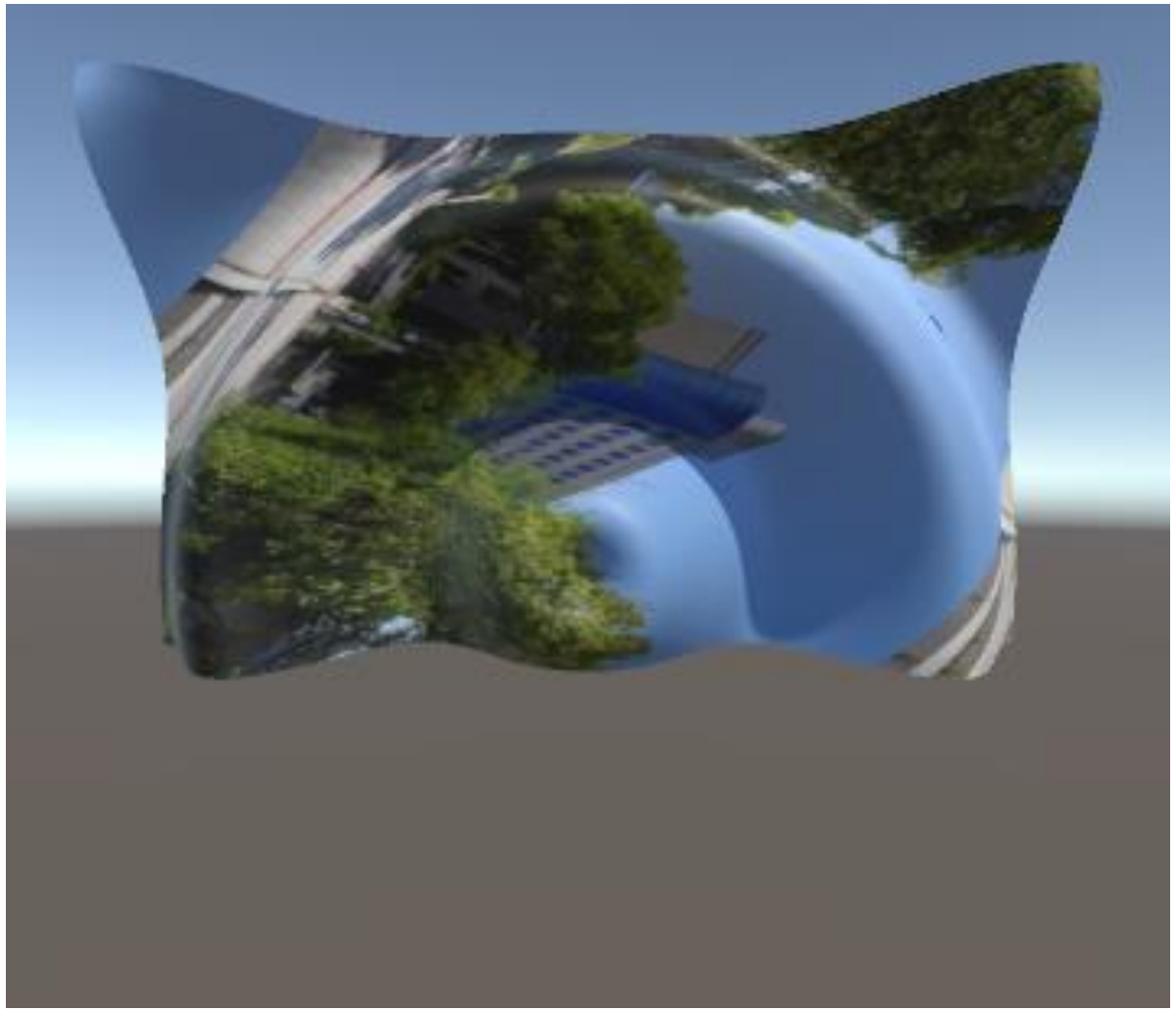
```
    [Range(0, 360)] public float angle = 30f;

    Vector3[] myVertices;
    Vector3[] myNormals;

    // Start is called before the first frame update
    void Start()
    {
        // 頂点の数
        int nVertices = (stacks + 1) * (slices + 1);

        // メッシュの頂点の位置の配列
        myVertices = new Vector3[nVertices];

        // メッシュの頂点の法線の配列
        myNormals = new Vector3[nVertices];

        // メッシュのテクスチャ座標
        Vector2[] myUVs = new Vector2[nVertices];

        // 回転の中心
        Vector2 c = new Vector2(0.5f, 0.5f);

        // 各頂点の位置と法線を求める
        for (int j = 0; j <= stacks; ++j)
        {
            // (中略)
```

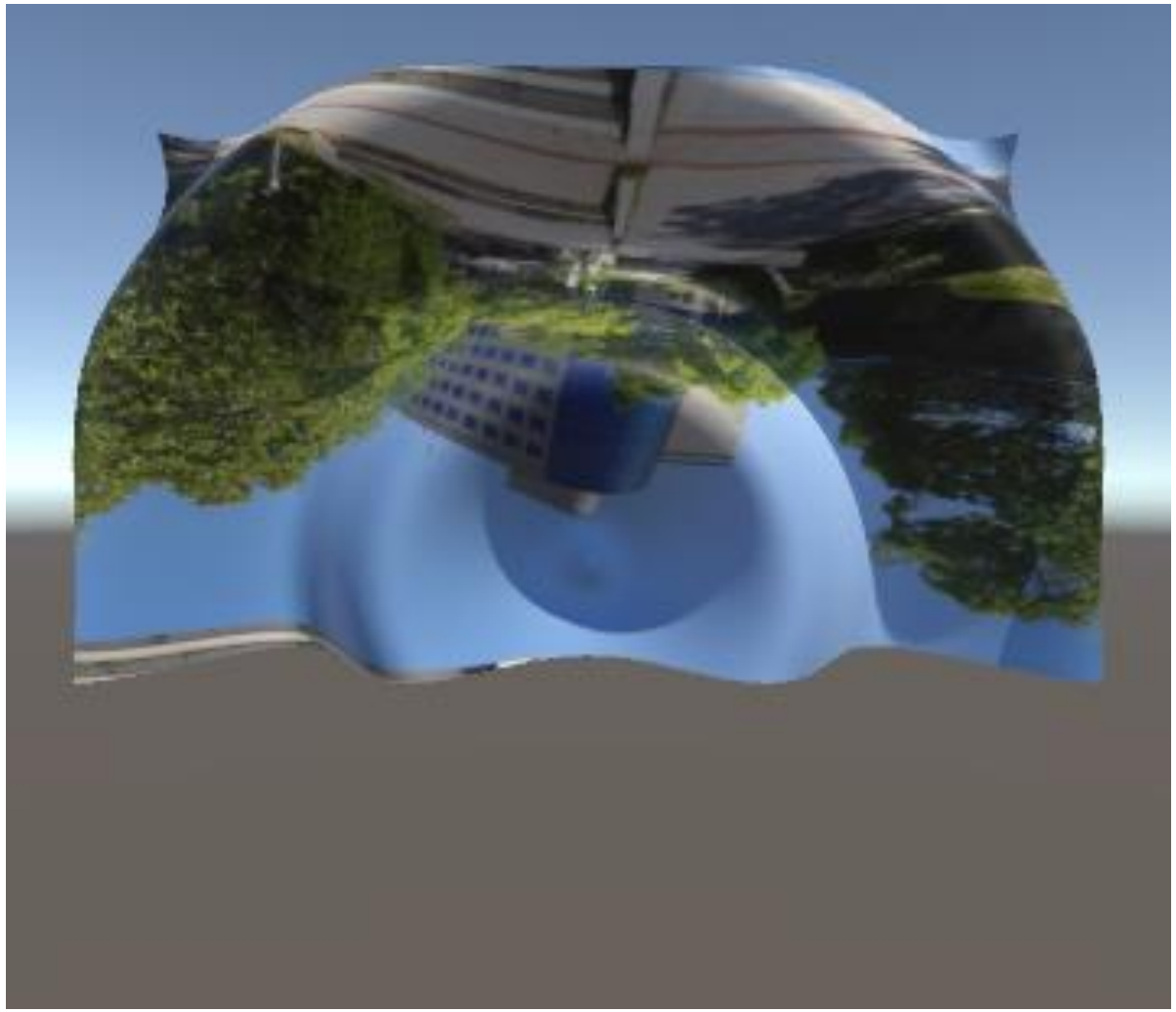
回転の中心を原点にして回転する

- テクスチャ座標 (u, v) を Vector2 型にして回転の中心 c が原点になるよう平行移動する
  - Vector2 uv = new Vector2(u, v) – c;
- 四元数を使ってテクスチャ座標を回転する
  - Vector2 xy
    = Quaternion.Euler(0, 0, angle) * uv;
  - Quaternion.Euler(*x*, *y*, *z*)
    - *x*, *y*, *z*: 各軸中心の回転角（単位は度）
- 結果 xz を元の位置に平行移動する
  - myUVs[pi] = xy + c;

```
        for (int i = 0; i <= slices; ++i)
        {
            // 横方向のパラメータ (0→1)
            float u = (float)i / slices;

            // 左から i 番目の四角形の左下の頂点の x 座標値
            float xi = x0 + width * u;

            // 左から i 番目の四角形の左下の頂点の頂点番号
            int pi = p0 + i;

            // 左から i 番目の四角形の左下の頂点の位置と法線
            myVertices[pi].Set(xi, yj, 0);
            myNormals[pi] = Vector3.back;

            // 回転の中心が原点になるよう平行移動する
            Vector2 uv = new Vector2(u, v) - c;

            // テクスチャ座標を回転する
            Vector2 xy = Quaternion.Euler(0, 0, angle) * uv;

            // 左から i 番目の四角形の左下の頂点のテクスチャ座標
            myUVs[pi] = xy + c;
        }
    }
```

演習６：時間に伴ってテクスチャを連続で回転させてください





解答例 (1) c と myUVs をメンバ変数にする

- 回転の中心 c とテクスチャ座標 myUVs の宣言を Start() の外に出してメンバ変数にします

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

public class MeshSample : MonoBehaviour
{
    const int slices = 100;
    const int stacks = 30;
    const float x0 = -5f;
    const float y0 = -1f;
    const float width = 10f;
    const float height = 6f;

    [Range(0, 360)] public float angle = 30f;

    // 回転の中心
    Vector2 c = new Vector2(0.5f, 0.5f);

    Vector3[] myVertices;
    Vector3[] myNormals;
    Vector2[] myUVs;

    // Start is called before the first frame update
    void Start()
    {
        // 頂点の数
        int nVertices = (stacks + 1) * (slices + 1);
```

解答例 (2) テクスチャ座標の配列を生成する

- テクスチャ座標の配列を生成してメンバ変数 myUVs から参照します

```
        // メッシュの頂点の位置の配列
        myVertices = new Vector3[nVertices];

        // メッシュの頂点の法線の配列
        myNormals = new Vector3[nVertices];

        // メッシュのテクスチャ座標
        myUVs = new Vector2[nVertices];

        // 各頂点の位置と法線を求める
        for (int j = 0; j <= stacks; ++j)
        {
            // 縦方向のパラメータ (0→1)
            float v = (float)j / stacks;

            // 各段の下側の頂点の高さ
            float yj = y0 + height * v;

            // 各段の左端の下側の頂点の頂点番号
            int p0 = j * (slices + 1);

            for (int i = 0; i <= slices; ++i)
            {
                // 横方向のパラメータ (0→1)
                float u = (float)i / slices;

                // (中略)
```

解答例 (3) 変更後のテクスチャ座標の格納先を用意する

- 元のテクスチャ座標 myUVs を回転したものを格納する配列を用意します

```
// Update is called once per frame
void Update()
{
    // 経過時間を取り出す
    float t = Time.time;

    // 経過時間にもとづいて変形速度を決定する
    float s = t * 20f;

    // 変更後の頂点位置の格納先
    Vector3[] vertices = new Vector3[myVertices.Length];

    // 変更後のテクスチャ座標の格納先
    Vector2[] uvs = new Vector2[myVertices.Length];

    // すべての頂点について
    for (int i = 0; i < myVertices.Length; ++i)
    {
        // 経過時間にもとづいて変形率を決定する
        //float r = Mathf.Sin(s);
        //float r = Mathf.Sin(s + myVertices[i].x);
        //float r = Mathf.Sin(s + myVertices[i].magnitude);
        float r =
            Mathf.Sin(s + myVertices[i].magnitude * 2f);

        // 元の図形の各頂点の位置を法線方向に移動する
        vertices[i] = myVertices[i] + myNormals[i] * r;
```

解答例 (4) テクスチャ座標を回転してメッシュを更新する

- テクスチャ座標 myUVs を回転の中心 c を原点とするように平行移動します
  - Vector2 uv = myUVs[i] – c;
- これを時間とともに変化する値（例えば s）を用いて回転します
  - Vector2 xy = Quaternion.Euler(0, 0, s) * uv;
- 結果を元の位置に平行移動します
  - uvs[i] = xy + c;
- 更新したテクスチャ座標をメッシュに設定します
  - myMesh.SetUVs(0, uvs);

```
            // 回転の中心が原点になるよう平行移動する
            Vector2 uv = myUVs[i] - c;

            // テクスチャ座標を回転する
            Vector2 xy = Quaternion.Euler(0, 0, s) * uv;

            // 左から i 番目の四角形の左下の頂点のテクスチャ座標
            uvs[i] = xy + c;
        }

        // MeshFilter から Mesh のオブジェクトを取り出す
        Mesh myMesh = GetComponent<MeshFilter>().mesh;

        // Mesh の頂点のデータを挿げ替える
        myMesh.SetVertices(vertices);

        // Mesh のテクスチャ座標を挿げ替える
        myMesh.SetUVs(0, uvs);

        // 法線ベクトルを再計算する
        myMesh.RecalculateNormals();
    }
}
```

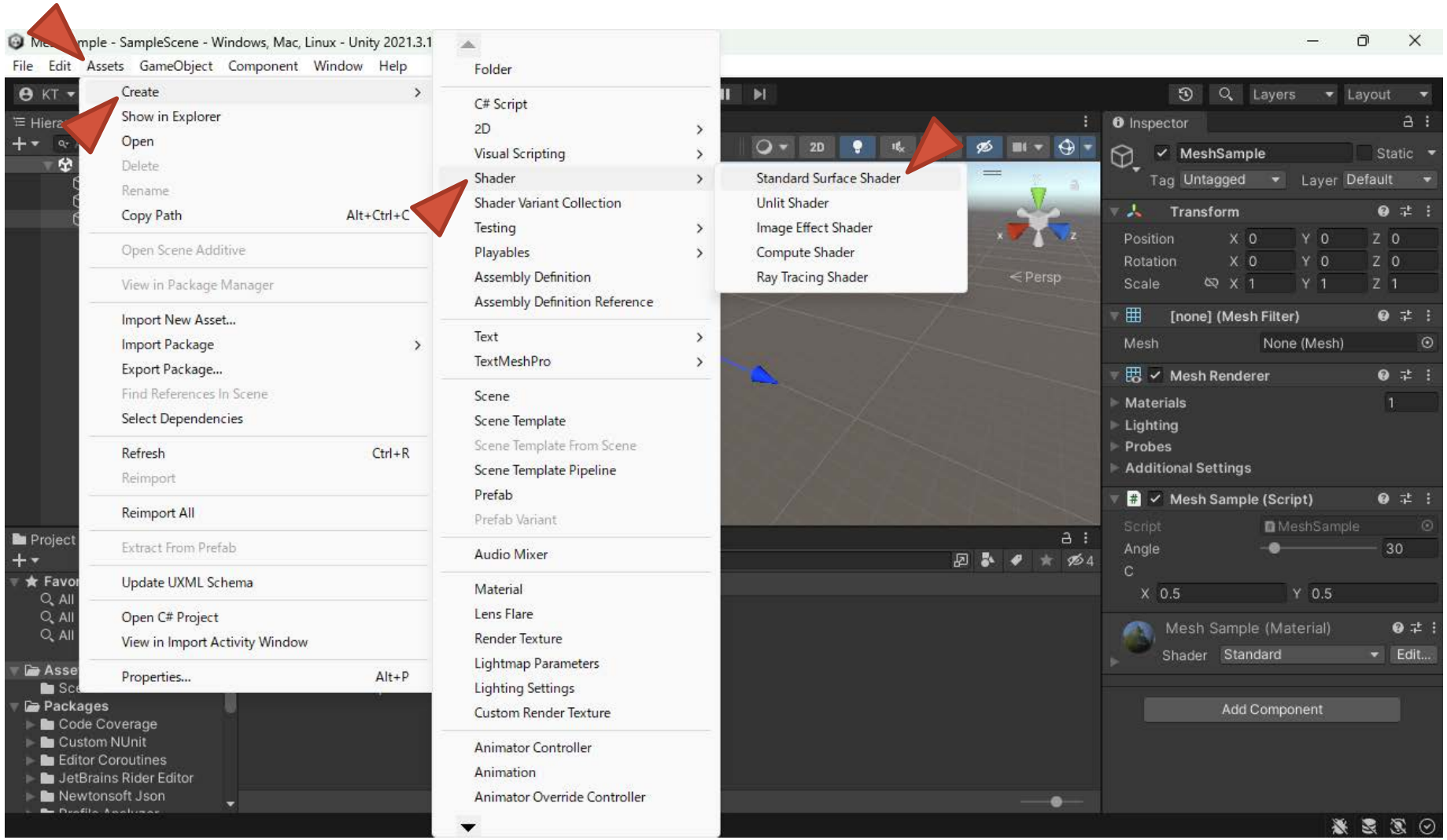
シェーダでテクスチャ座標を操作する

Standard Surface Shader

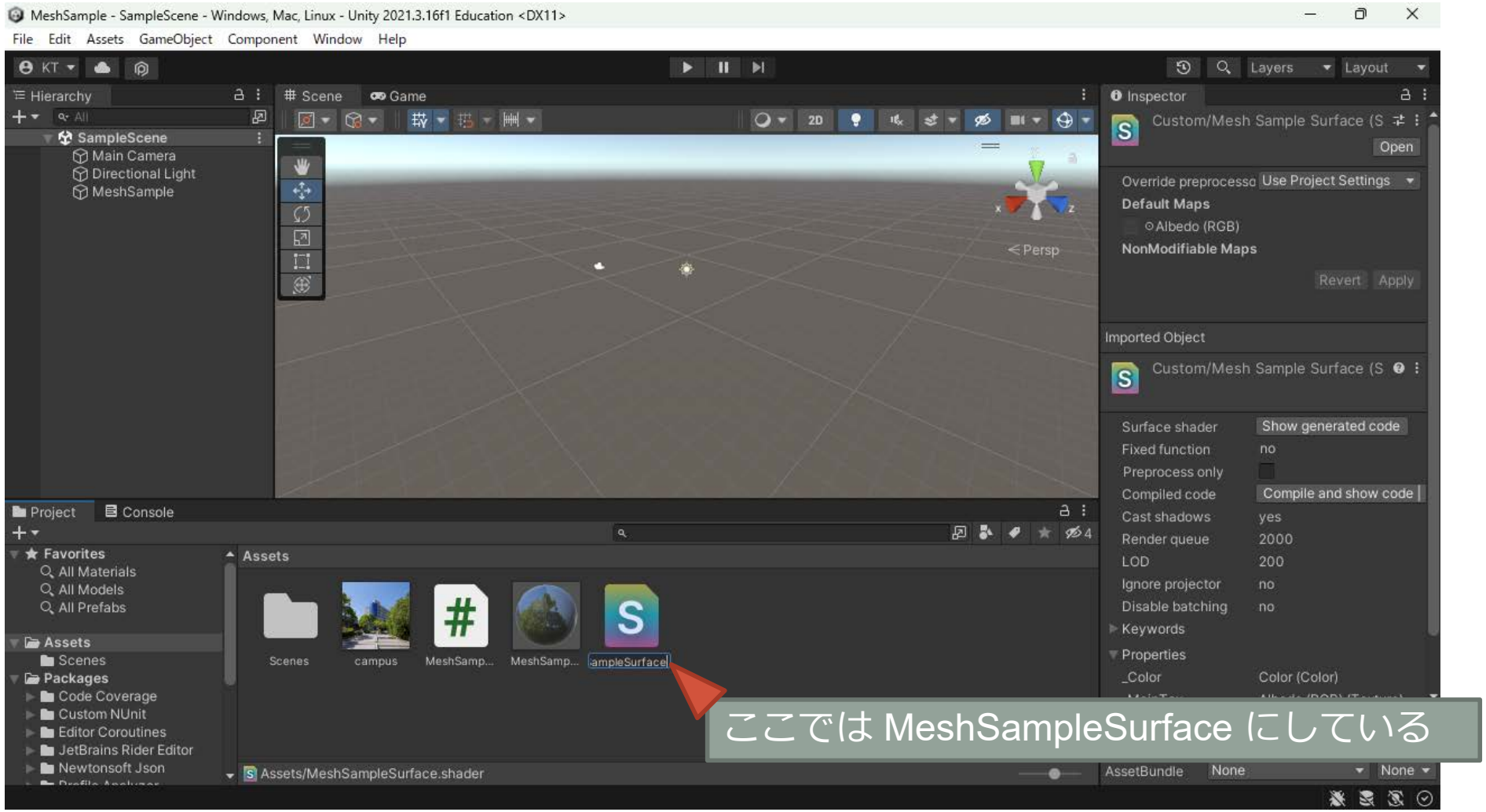
演習６のプログラムからテクスチャの回転処理を削除する

```
    // Update is called once per frame
    void Update()
    {
        // 経過時間を取り出す
        float t = Time.time;

        // 経過時間にもとづいて変形速度を決定する
        float s = t * 20f;

        // 変更後の頂点位置の格納先
        Vector3[] vertices = new Vector3[myVertices.Length];

        // 変更後のテクスチャ座標の格納先
        Vector2[] uvs = new Vector2[myVertices.Length];

        // すべての頂点について
        for (int i = 0; i < myVertices.Length; ++i)
        {
            // 経過時間にもとづいて変形率を決定する
            //float r = Mathf.Sin(s);
            //float r = Mathf.Sin(s + myVertices[i].x);
            //float r = Mathf.Sin(s + myVertices[i].magnitude);
            float r =
                Mathf.Sin(s + myVertices[i].magnitude * 2f);

            // 元の図形の各頂点の位置を法線方向に移動する
            vertices[i] = myVertices[i] + myNormals[i] * r;

            // 回転の中心が原点になるよう平行移動する
            Vector2 uv = myUVs[i] - c;

            // テクスチャ座標を回転する
            Vector2 xy = Quaternion.Euler(0, 0, s) * uv;

            // 左から i 番目の四角形の左下の頂点のテクスチャ座標
            uvs[i] = xy + c;
        }

        // MeshFilter から Mesh のオブジェクトを取り出す
        Mesh myMesh = GetComponent<MeshFilter>().mesh;

        // Mesh の頂点のデータを挿げ替える
        myMesh.SetVertices(vertices);

        // Mesh のテクスチャ座標を挿げ替える
        myMesh.SetUVs(0, uvs);

        // 法線ベクトルを再計算する
        myMesh.RecalculateNormals();
    }
}
```

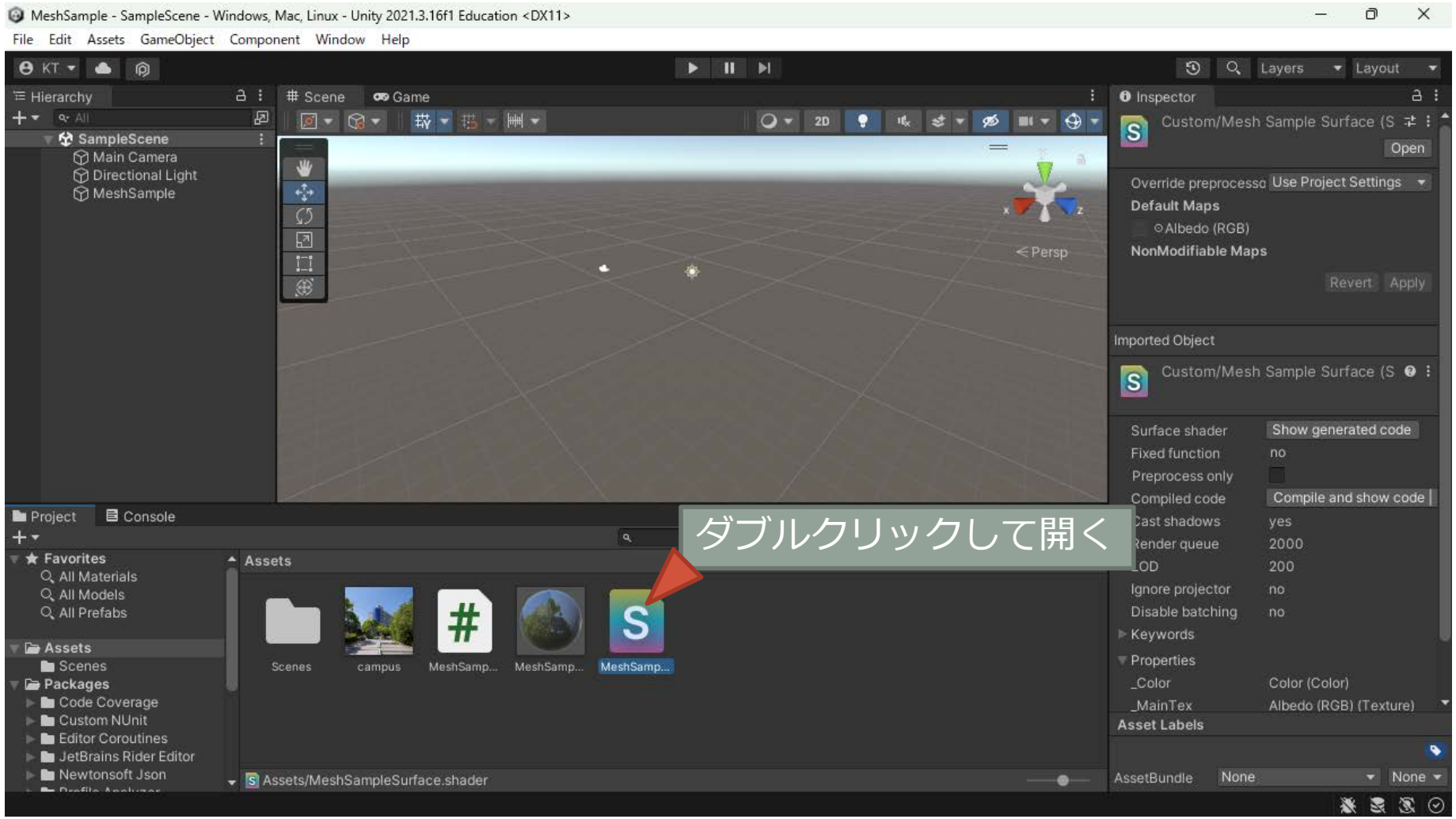
Standard Surface Shader を作る



作成したシェーダの名前を変更する



シェーダのソースファイルをテキストエディタで編集する



シェーダに角度を受け取る uniform 変数 _Theta を追加する

- SubShader で _Theta という uniform 変数を宣言する
  - float _Theta;
  - uniform 変数は C# のプログラム (CPU 側) からシェーダに値を渡すのに用いる
  - １回の描画 (Draw Call) の間は変化しない
- Properties で uniform 変数にプロパティを設定すればインスペクタから値を設定できるようになる

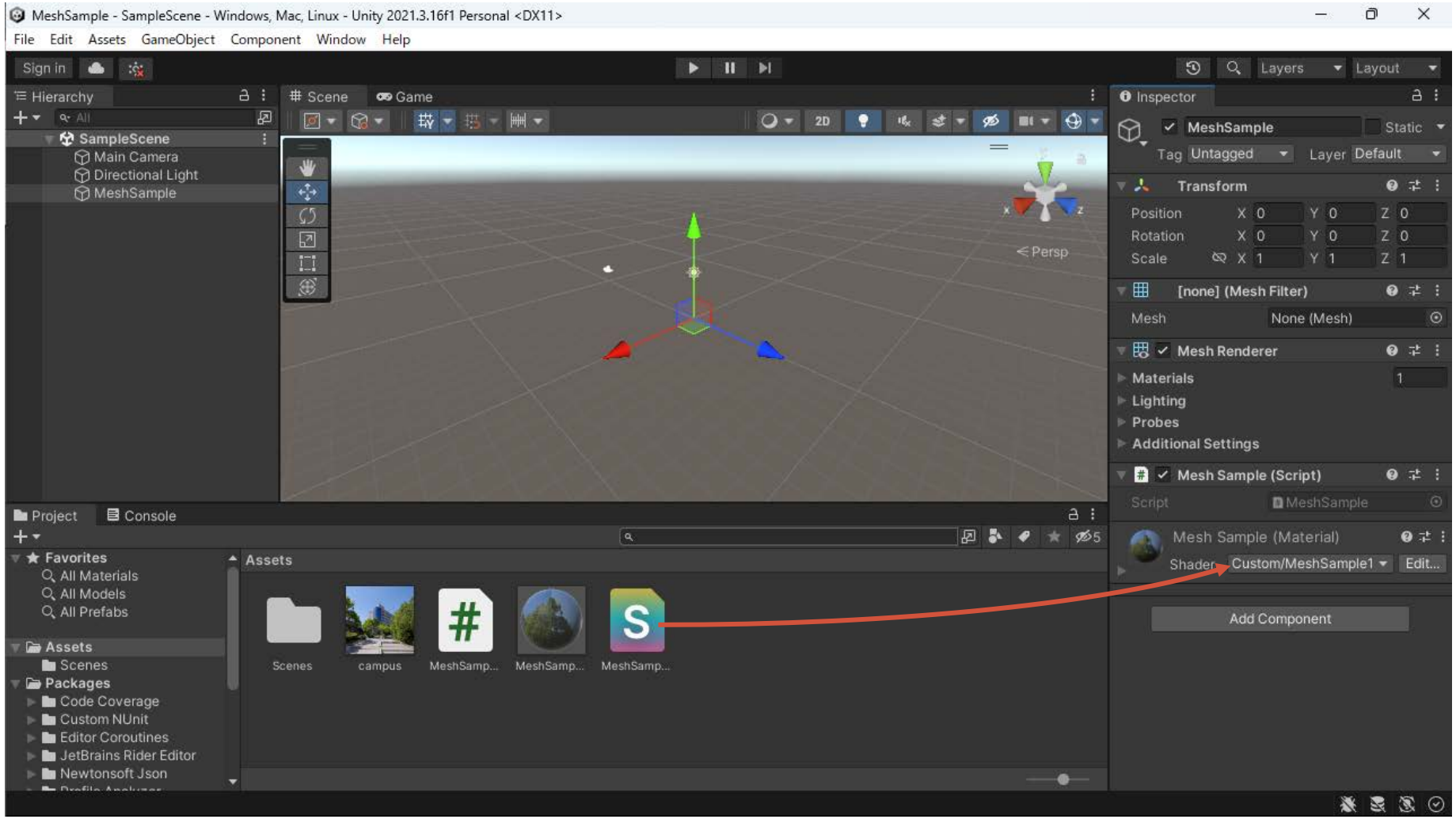
```
    Properties
    {
        _Color ("Color", Color) = (1,1,1,1)
        _MainTex ("Albedo (RGB)", 2D) = "white" {}
        _Glossiness ("Smoothness", Range(0,1)) = 0.5
        _Metallic ("Metallic", Range(0,1)) = 0.0
        _Theta ("Theta", Range(0,6.2831853)) = 0.0
    }
    SubShader
    {
        // (中略)

        sampler2D _MainTex;

        struct Input
        {
            float2 uv_MainTex;
        };

        half _Glossiness;
        half _Metallic;
        fixed4 _Color;
        float _Theta;

        // (中略)
```

テクスチャ座標を回転してテクスチャをサンプリングする

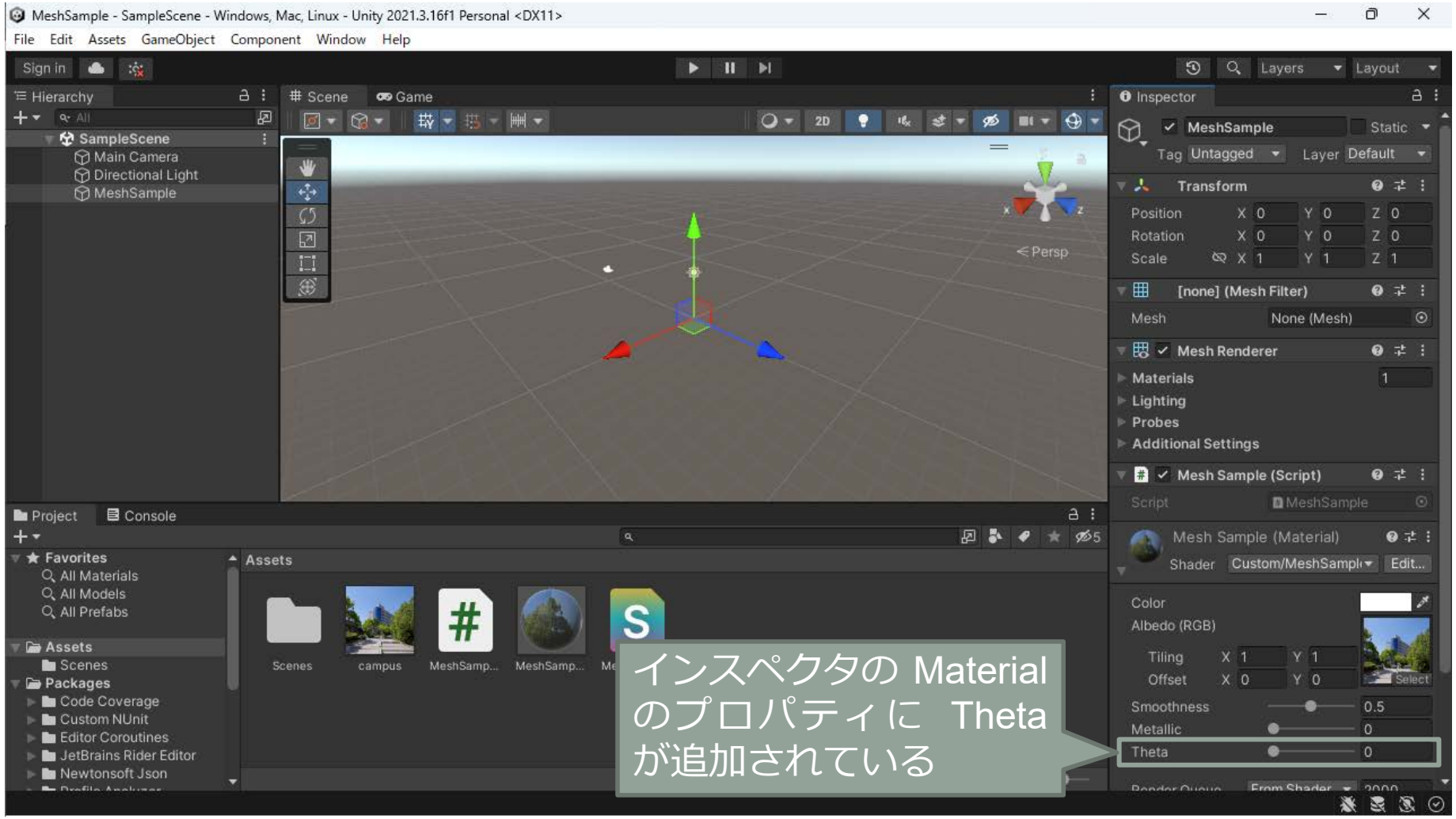
- _Theta の正弦と余弦を求めておく
  - float cosTheta = cos(_Theta);
  - float sinTheta = sin(_Theta);
- テクスチャ座標の u,v 成分を取り出す
  - テクスチャ座標は IN.uv_MainTex に入っている
  - float x = IN.uv_MainTex.x;
  - float y = IN.uv_MainTex.y;
- 取り出したテクスチャ座標を回転する
  - float u = cosTheta * x – sinTheta * y;
  - float v = sinTheta * x + cosTheta * y;

```
        // (中略)

        void surf (Input IN, inout SurfaceOutputStandard o)
        {
            float cosTheta = cos(_Theta);
            float sinTheta = sin(_Theta);
            float u = IN.uv_MainTex.x;
            float v = IN.uv_MainTex.y;
            float x = cosTheta * u - sinTheta * v;
            float y = sinTheta * u + cosTheta * v;
            // Albedo comes from a texture tinted by color
            fixed4 c = tex2D (_MainTex, float2(x, y)) * _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"
}
```

Material の Shader に設定する



インスペクタの Material にプロパティが追加されている



Theta を変更してからプロジェクトを実行する

行列を使って回転する

- テクスチャ座標の回転に行列を用いる

$$\mathbf{M} = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix}$$

  - float2x2 m = float2x2(cosTheta, –sinTheta,
    sinTheta, cosTheta);
- テクスチャ座標に回転の行列をかける

$$\mathbf{t}_{uv} = \mathbf{M}\mathbf{t}_{uv\_MainTex}$$

  - float2 uv = mul(m, IN.uv_MainTex);
- 回転したテクスチャ座標でテクスチャをサンプリングする
  - fixed4 c = tex2D (_MainTex, uv) * _Color;

```
        // (中略)

        void surf (Input IN, inout SurfaceOutputStandard o)
        {
            float cosTheta = cos(_Theta);
            float sinTheta = sin(_Theta);
            float2x2 m = float2x2(cosTheta, -sinTheta,
                                  sinTheta, cosTheta);
            float2 uv = mul(m, IN.uv_MainTex);
            // Albedo comes from a texture tinted by color
            fixed4 c = tex2D (_MainTex, uv) * _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"
}
```

テクスチャの中心を中心にして回転する

- テクスチャの中心位置 $\mathbf{c}$ が原点になるようテクスチャ座標を平行移動して回転の行列をかけてから元の位置に戻す

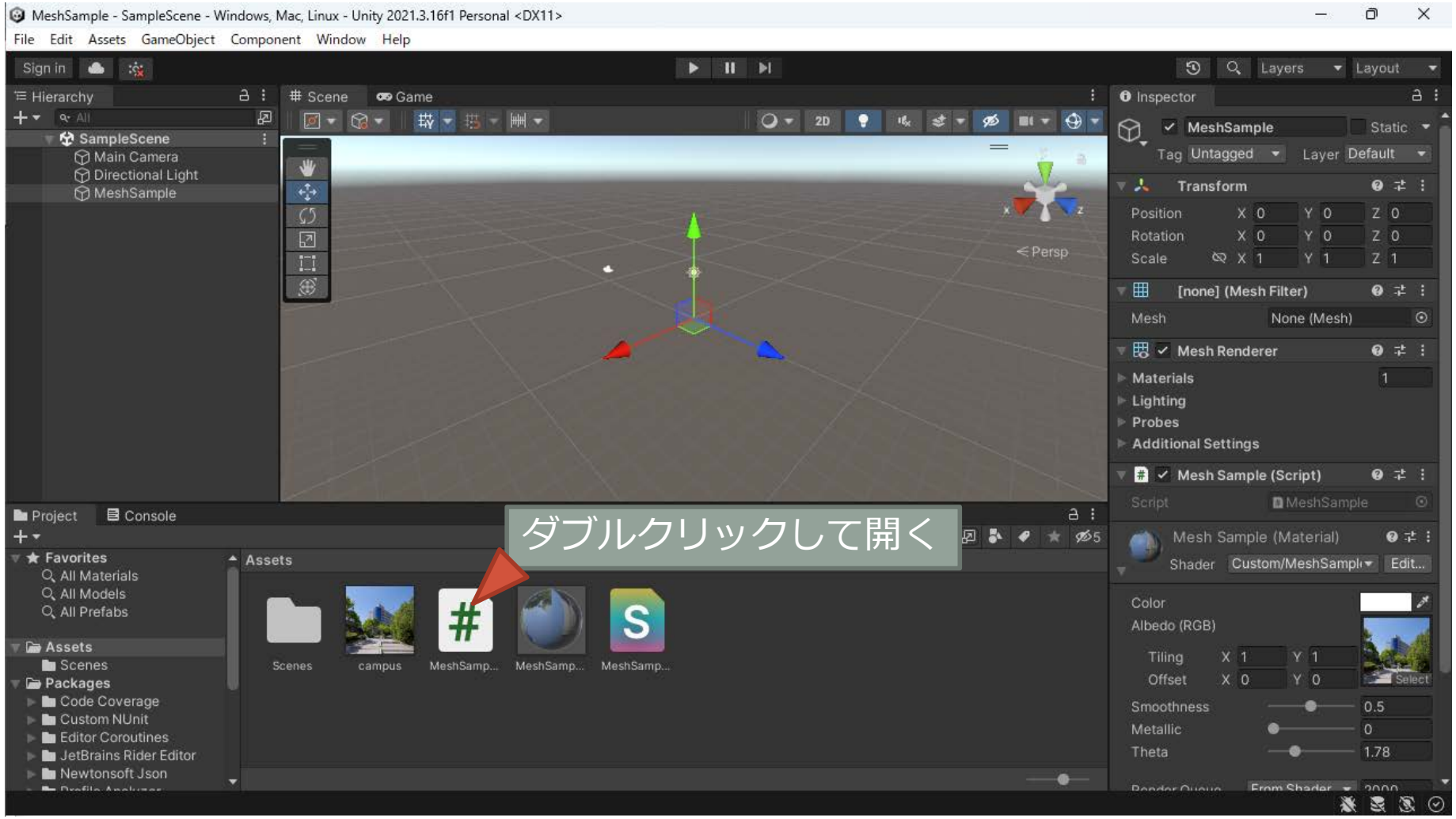
$$\mathbf{c} = \begin{pmatrix} 0.5 \\ 0.5 \end{pmatrix}$$

$$\mathbf{t}_{uv} = \mathbf{M}\left(\mathbf{t}_{uv\_MainTex} - \mathbf{c}\right) + \mathbf{c}$$

  - float2 uv
    = mul(m, IN.uv_MainTex – 0.5) + 0.5;
- シェーダ (HLSL) ではベクトルとスカラーの演算はスカラーをベクトルに型変換して実行される

```
        // (中略)

        void surf (Input IN, inout SurfaceOutputStandard o)
        {
            float cosTheta = cos(_Theta);
            float sinTheta = sin(_Theta);
            float2x2 m = float2x2(cosTheta, -sinTheta,
                                  sinTheta, cosTheta);
            float2 uv = mul(m, IN.uv_MainTex - 0.5) + 0.5;
            // Albedo comes from a texture tinted by color
            fixed4 c = tex2D (_MainTex, uv) * _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"
}
```

時間に伴ってテクスチャを連続で回転させる



uniform 変数 _Theta に値を設定する

- ゲームオブジェクトから Material を取り出す
  - Material myMaterial
    = GetComponent<Renderer>().material;
- Material に組み込んだシェーダの uniform 変数 _Theta に値を設定する
  - myMaterial.SetFloat("_Theta", t);

```
    // Update is called once per frame
    void Update()
    {
        // 変更後の頂点位置
        Vector3[] vertices = new Vector3[myVertices.Length];

        // 経過時間を取り出す
        float t = Time.time;

        // (中略)

        // MeshFilter から Mesh のオブジェクトを取り出す」
        Mesh myMesh = GetComponent<MeshFilter>().mesh;

        // Mesh の頂点のデータを挿げ替える
        myMesh.SetVertices(vertices);

        // 法線ベクトルを再計算する
        myMesh.RecalculateNormals();

        // GameObject から Material の Component を取り出す
        Material myMaterial = GetComponent<Renderer>().material;

        // シェーダの uniform 変数 _Theta に t を設定する
        myMaterial.SetFloat("_Theta", t);
    }
}
```

バーテックスシェーダでテクスチャ座標を回転する

- Surface シェーダ (surf) は画面上の画素ごとに実行される
- C# スクリプトと同様に頂点ごとにテクスチャ座標の座標を回転するにはバーテックスシェーダ (vert) を用いる
  - バーテックスシェーダを使用するには #pragma surface に vertex:vert を追加する

```
#pragma surface surf Standard fullforwardshadows vertex:vert
```

```
        // (中略)

        void vert (inout appdata_full v)
        {
            float cosTheta = cos(_Theta);
            float sinTheta = sin(_Theta);
            float2x2 m = float2x2(cosTheta, -sinTheta,
                                  sinTheta, cosTheta);
            v.texcoord.xy = mul(m, v.texcoord.xy - 0.5) + 0.5;
        }

        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"
}
```

バーテックスシェーダで頂点の位置を変更する

- バーテックスシェーダでは頂点のテクスチャ座標のほかにも位置などの頂点アトリビュートを修正できる
  - float s = t * 20f;
  - float r = Mathf.Sin(s + myVertices[i].magnitude * 2f);
  - vertices[i] = myVertices[i] + myNormals[i] * r;
- ただし頂点の位置の変更に伴う法線を再計算するには隣接する頂点の位置が必要になるがバーテックスシェーダでは他の頂点の頂点アトリビュートは参照できない

```
        // (中略)

        void vert (inout appdata_base v)
        {
            float magnitude = length(v.vertex.xyz);
            float t = _Theta * 20.0 + magnitude * 2.0;
            v.vertex.xyz += v.normal * sin(t);
            float cosTheta = cos(_Theta);
            float sinTheta = sin(_Theta);
            float2x2 m = float2x2(cosTheta, -sinTheta,
                                  sinTheta, cosTheta);
            v.texcoord.xy = mul(m, v.texcoord.xy - 0.5) + 0.5;
        }

        // (後略)
```

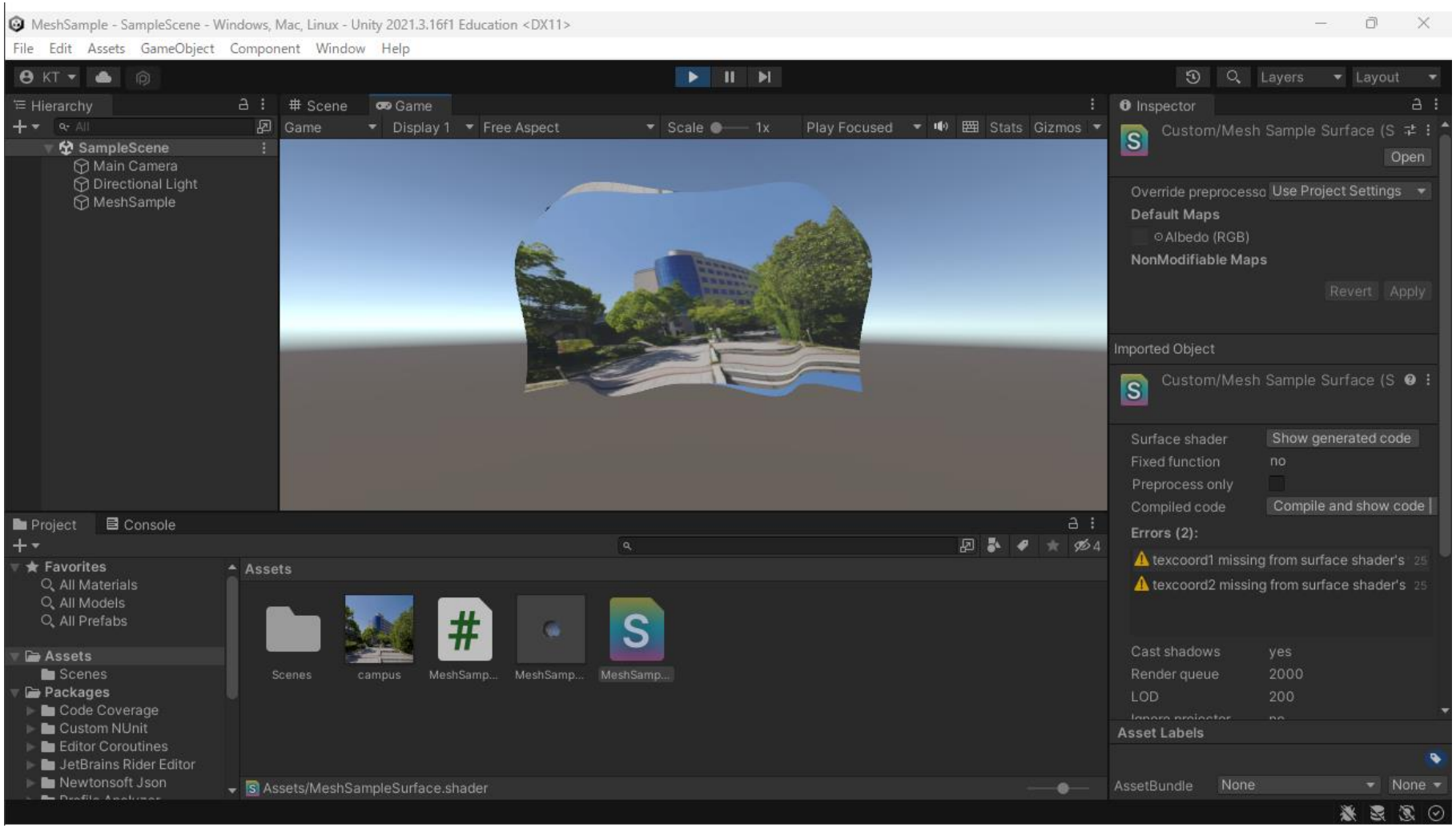
頂点の位置の変更処理を削除する

```
    // Update is called once per frame
    void Update()
    {
        // 経過時間を取り出す
        float t = Time.time;

        // 経過時間にもとづいて変形速度を決定する
        float s = t * 20f;

        // 変更後の頂点位置の格納先
        Vector3[] vertices = new Vector3[myVertices.Length];

        // すべての頂点について
        for (int i = 0; i < myVertices.Length; ++i)
        {
            // 経過時間にもとづいて変形率を決定する
            //float r = Mathf.Sin(s);
            //float r = Mathf.Sin(s + myVertices[i].x);
            //float r = Mathf.Sin(s + myVertices[i].magnitude);
            float r = Mathf.Sin(
                s + myVertices[i].magnitude * 2f);

            // 元の図形の各頂点の位置を法線方向に移動する
            vertices[i] = myVertices[i] + myNormals[i] * r;
        }
```

```
        // MeshFilter から Mesh のオブジェクトを取り出す
        Mesh myMesh = GetComponent<MeshFilter>().mesh;

        // Mesh の頂点のデータを挿げ替える
        myMesh.SetVertices(vertices);

        // 法線ベクトルを再計算する
        myMesh.RecalculateNormals();

        // GameObject から Material の Component を取り出す
        Material myMaterial = GetComponent<Renderer>().material;

        // シェーダの uniform 変数 _Theta に t を設定する
        myMaterial.SetFloat("_Theta", t);
    }
}
```

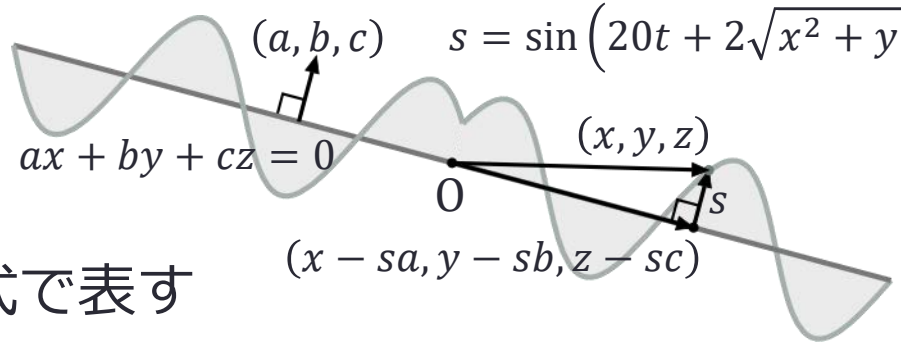
頂点の位置を変更しないので法線を再計算する意味がない

法線ベクトルが変化していないので陰影が付かない



法線ベクトルを解析的に求める

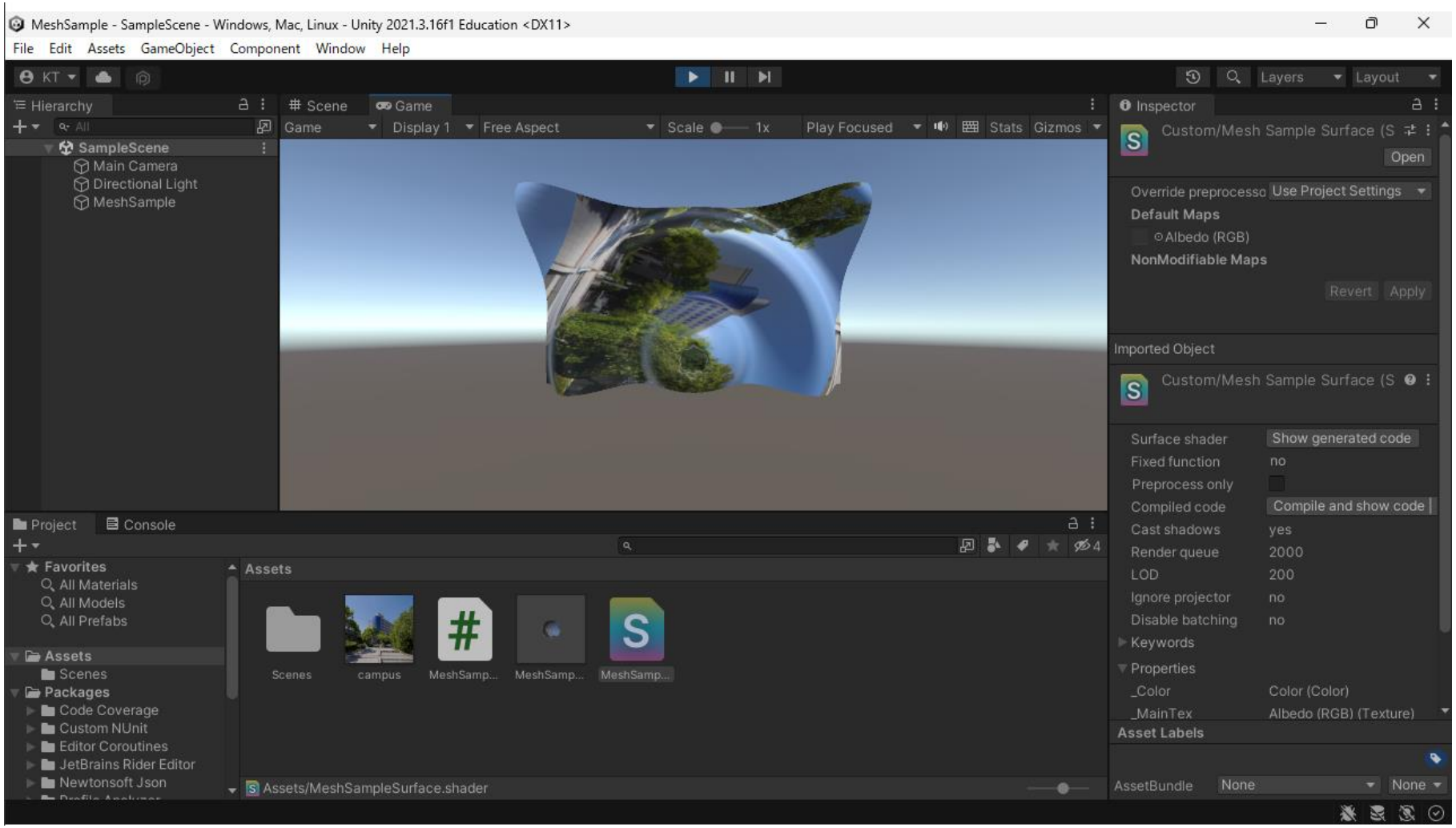
- 波面上の位置



- 数式で表す
  - $f(x,y,z) = \sin\left(20t + 2\sqrt{x^2+y^2+z^2}\right) - ax - by - cz = 0$
- 勾配を求めて符号を反転する
  - $\mathbf{n} = -\nabla f(x,y,z) = \begin{pmatrix} a - \frac{2x\cos\left(20t+2\sqrt{x^2+y^2+z^2}\right)}{\sqrt{x^2+y^2+z^2}} \\ b - \frac{2y\cos\left(20t+2\sqrt{x^2+y^2+z^2}\right)}{\sqrt{x^2+y^2+z^2}} \\ c - \frac{2z\cos\left(20t+2\sqrt{x^2+y^2+z^2}\right)}{\sqrt{x^2+y^2+z^2}} \end{pmatrix}$

```
        // (中略)

        void vert (inout appdata_base v)
        {
            float magnitude = length(v.vertex.xyz);
            float t = _Theta * 20.0 + magnitude * 2.0;
            v.vertex.xyz += v.normal * sin(t);
            v.normal = normalize(magnitude * v.normal
              - 2.0 * v.vertex.xyz * cos(t));
            float cosTheta = cos(_Theta);
            float sinTheta = sin(_Theta);
            float2x2 m = float2x2(cosTheta, -sinTheta,
                                  sinTheta, cosTheta);
            v.texcoord.xy = mul(m, v.texcoord.xy - 0.5) + 0.5;
        }

        // (後略)
```

それらしい陰影が付く（けど影はつかない）



参考
「コンピュータグラフィックス」の資料

現在この内容は実施していません

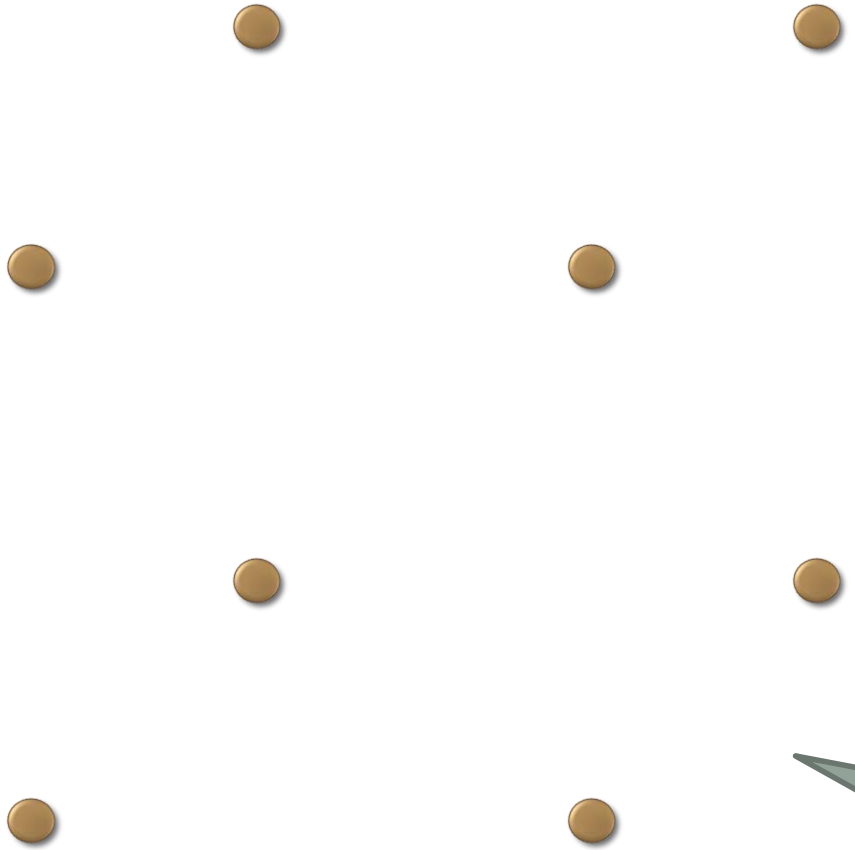
立体形状の表現方法

- 境界の情報による表現
- 立体の組み合わせによる表現
- 密度や濃度による表現
- 生成過程の記述による表現
- 手続きによる表現

境界による表現

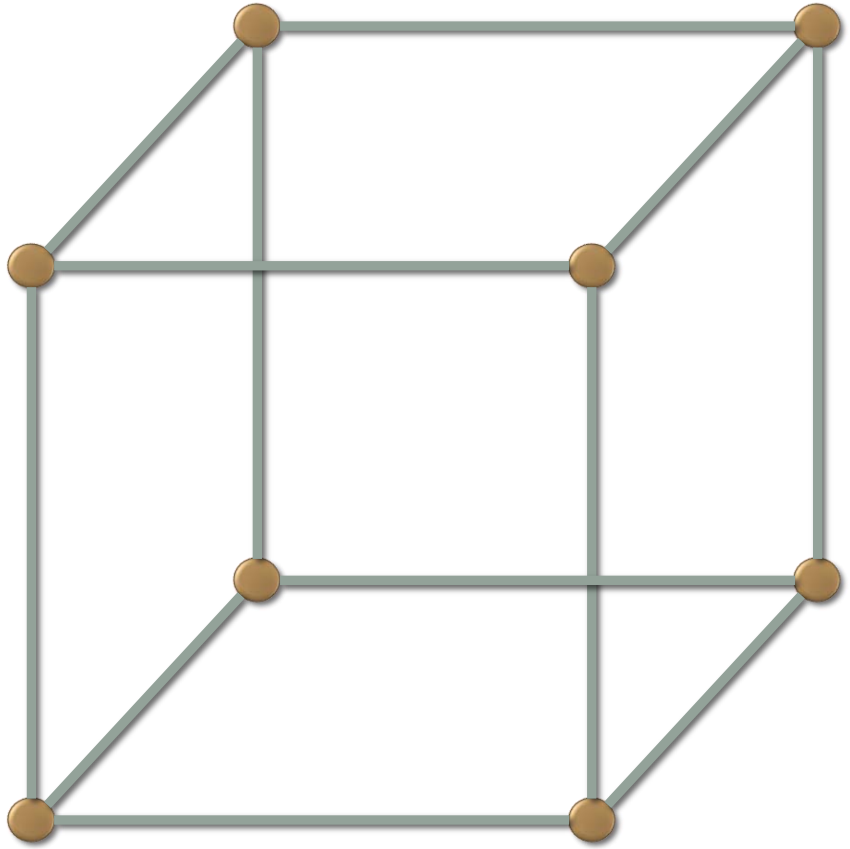
- 頂点・稜線・面などの情報で形を表現する
  - 立体の内部と外部を隔てる境界
- ワイヤフレームモデル
  - 簡易で高速⇔情報量は不十分
- サーフェースモデル
  - ハーフトーン画像の生成が可能
- ソリッドモデル
  - 立体同士の演算等が可能

点で形を表してみる



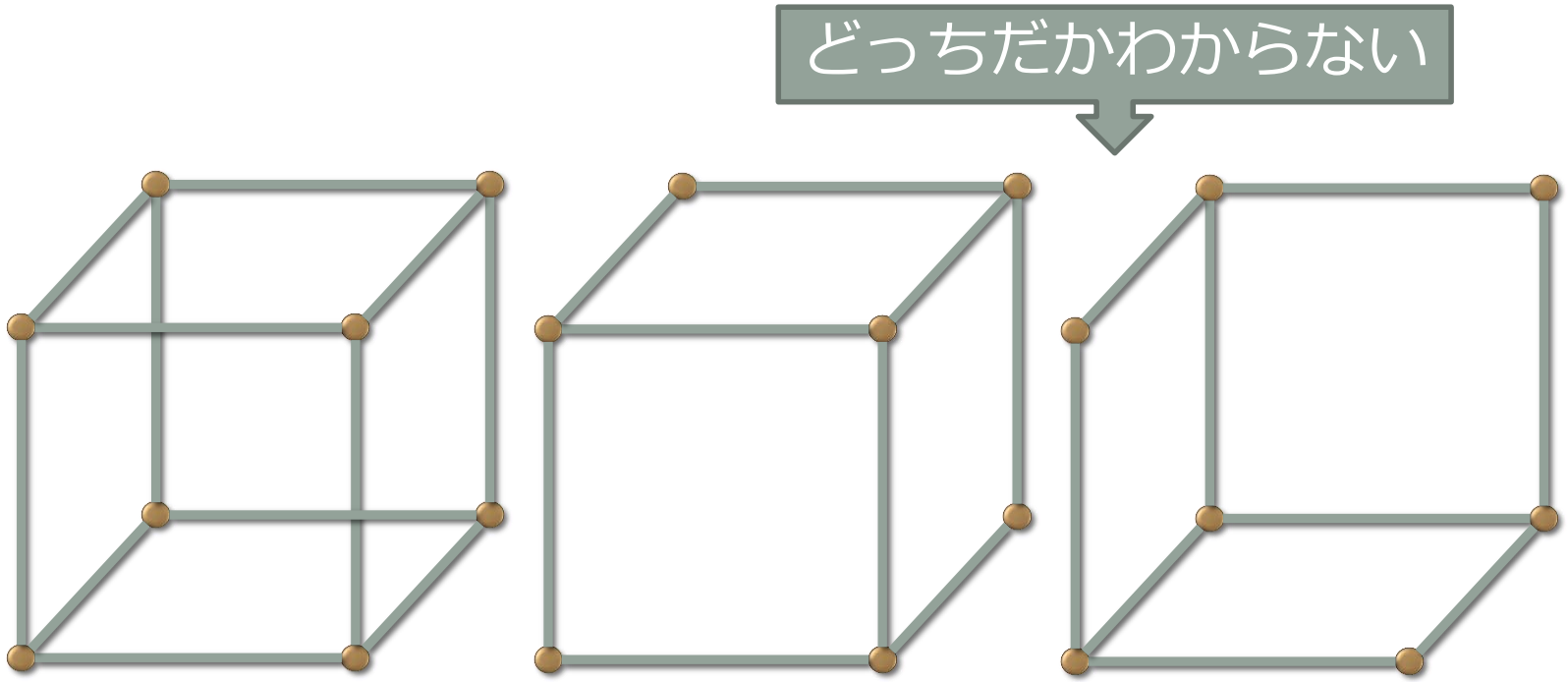
点が少ないと
形がよくわからない

点を線分で結んでみる

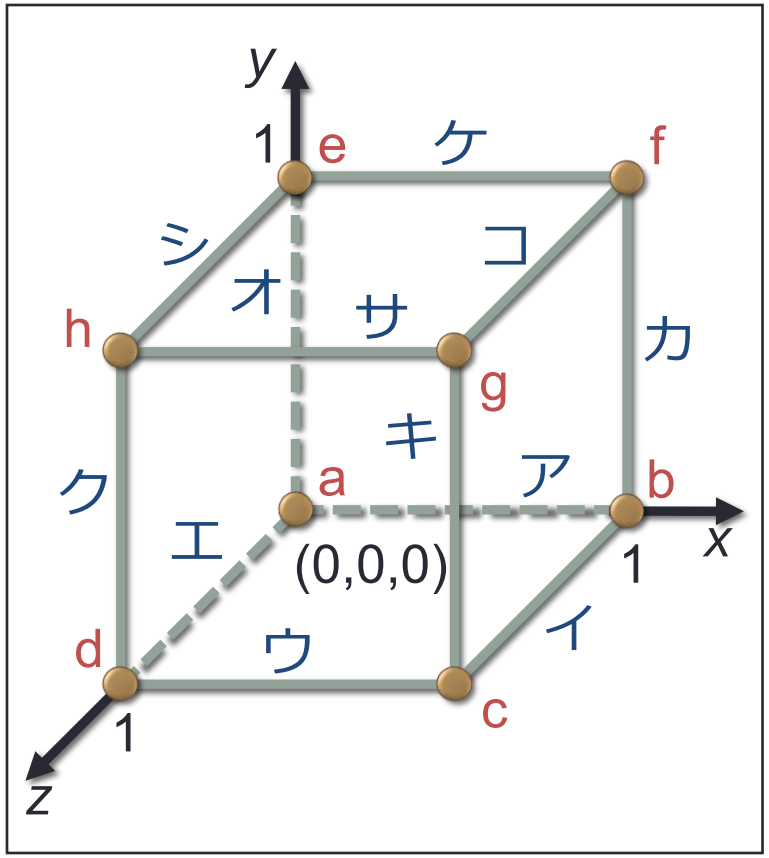


ワイヤーフレームモデル

- 頂点と，それらを結ぶ線分で形を表現する
  - 形状を把握することができる
  - 曖昧さは残る



頂点データと稜線データ



頂点表

頂点	座標値		
	x	*y*	*z*
a	0	0	0
b	1	0	0
c	1	0	1
d	0	0	1
e	0	1	0
f	1	1	0
g	1	1	1
h	0	1	1

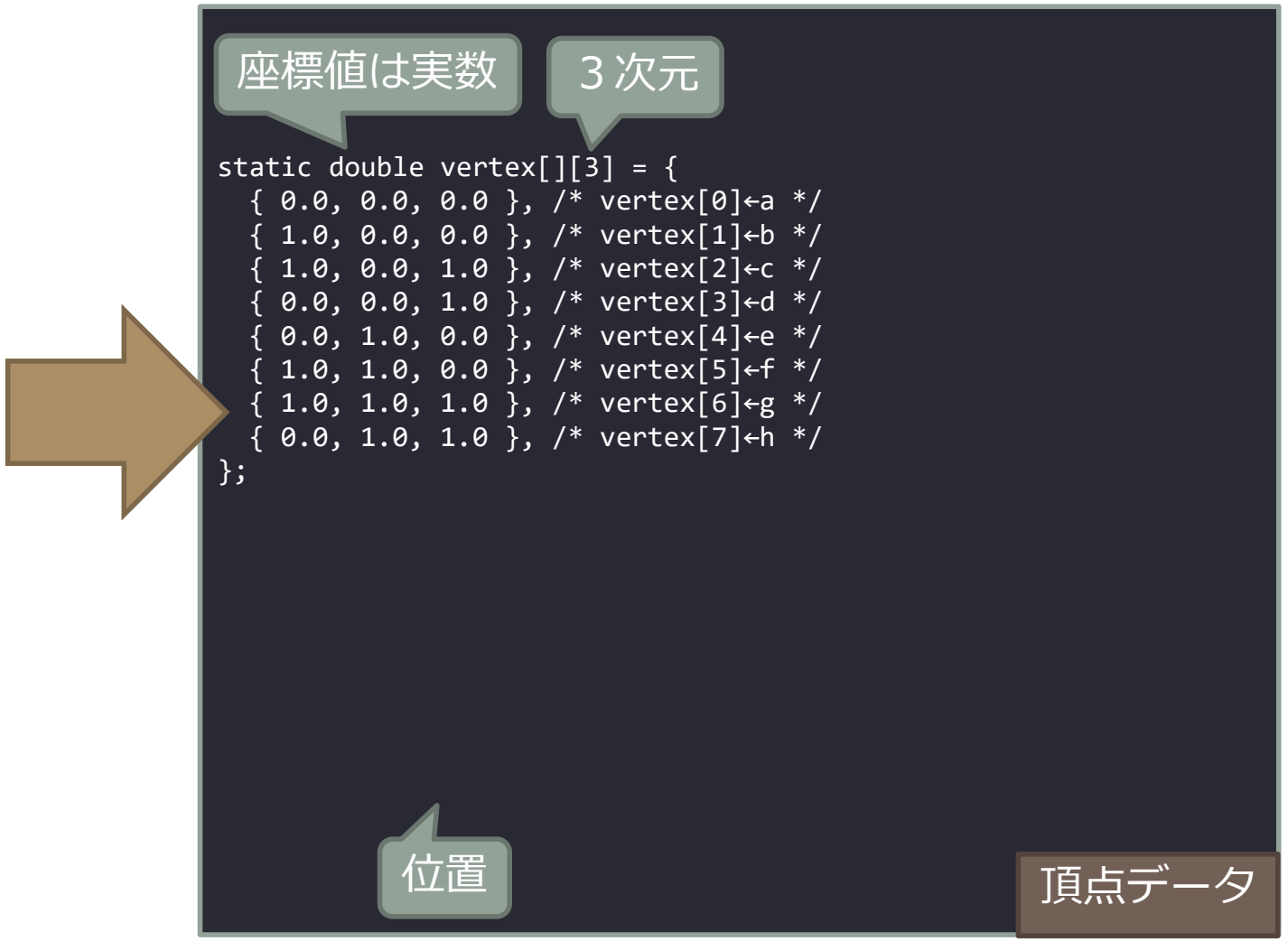
稜線表

稜線	頂点	
	始点	終点
ア	a	b
イ	b	c
ウ	c	d
エ	d	a
オ	a	e
カ	b	f
キ	c	g
ク	d	h
ケ	e	f
コ	f	g
サ	g	h
シ	h	e

頂点表のデータ化の例

頂点表

頂点	位置		
	x	*y*	*z*
a	0	0	0
b	1	0	0
c	1	0	1
d	0	0	1
e	0	1	0
f	1	1	0
g	1	1	1
h	0	1	1



稜線表のデータ化の例

稜線表

稜線	頂点	
	始点	終点
ア	a	b
イ	b	c
ウ	c	d
エ	d	a
オ	a	e
カ	b	f
キ	c	g
ク	d	h
ケ	e	f
コ	f	g
サ	g	h
シ	h	e

番号なので整数

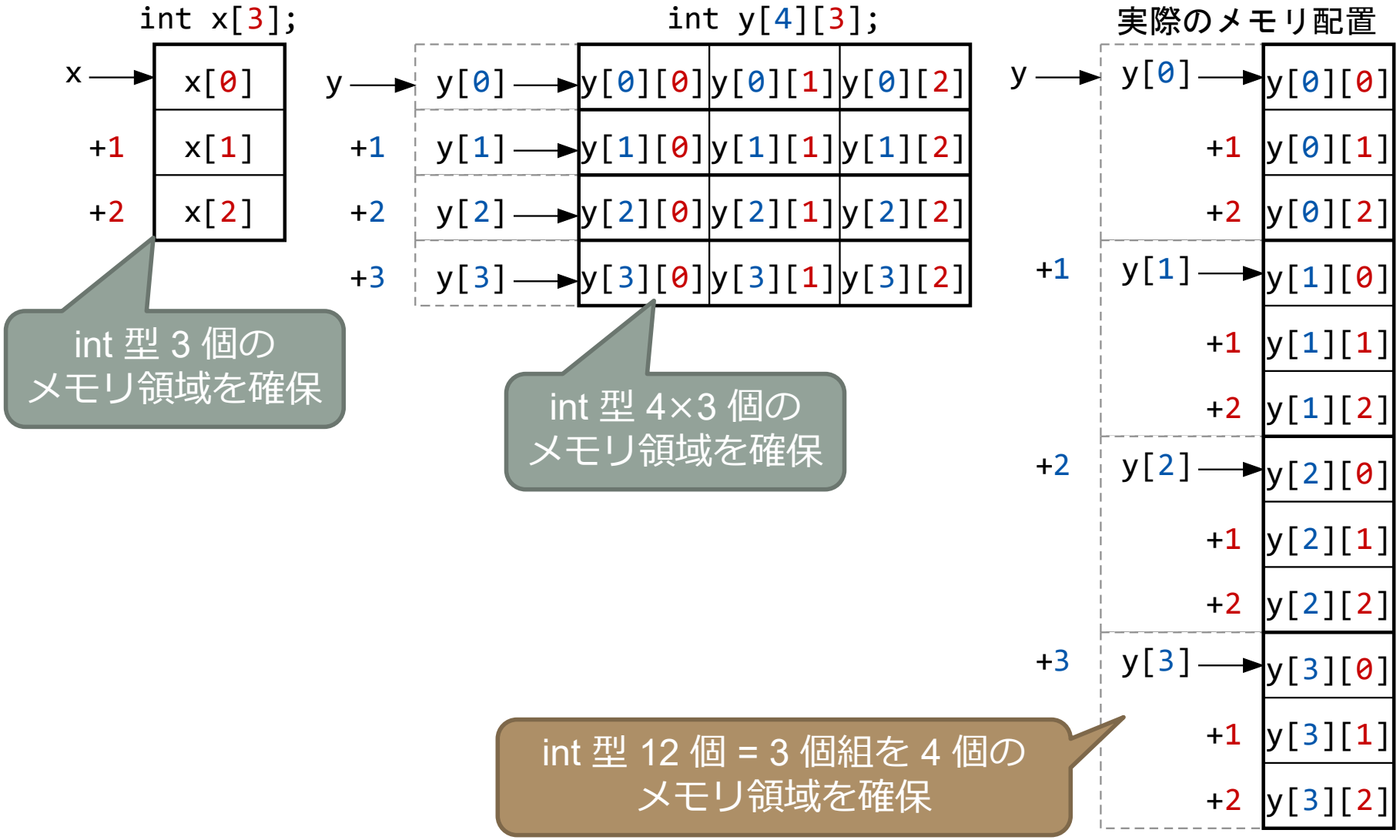
始点と終点の２点

番号に
置き換え

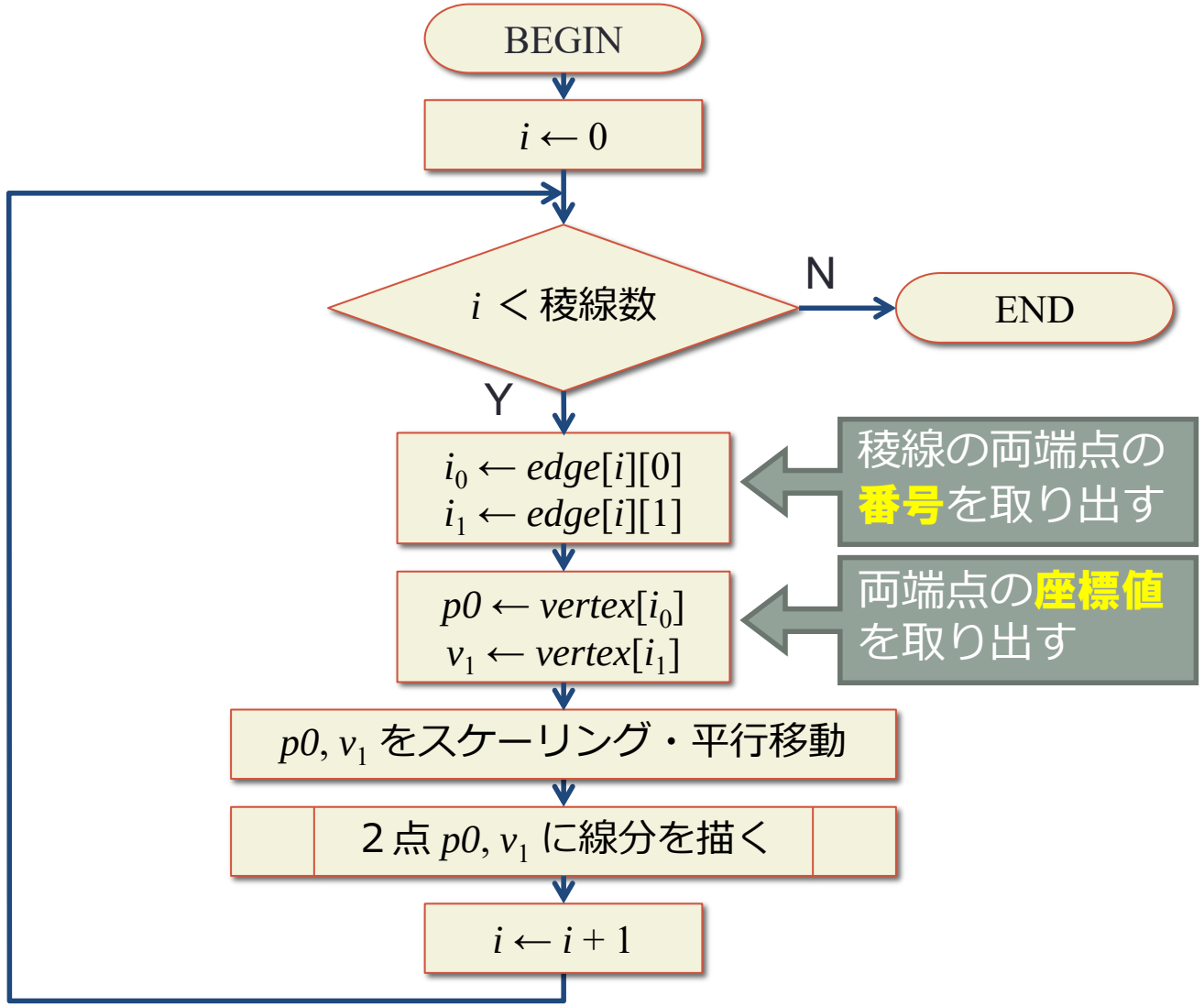
```
static int edge[][2] = {
  { 0, 1 }, /* ア (a-b) */
  { 1, 2 }, /* イ (b-c) */
  { 2, 3 }, /* ウ (c-d) */
  { 3, 0 }, /* エ (d-a) */
  { 0, 4 }, /* オ (a-e) */
  { 1, 5 }, /* カ (b-f) */
  { 2, 6 }, /* キ (c-g) */
  { 3, 7 }, /* ク (d-h) */
  { 4, 5 }, /* ケ (e-f) */
  { 5, 6 }, /* コ (f-g) */
  { 6, 7 }, /* サ (g-h) */
  { 7, 4 }, /* シ (h-e) */
};
```

稜線データ

二次元配列のメモリ配置

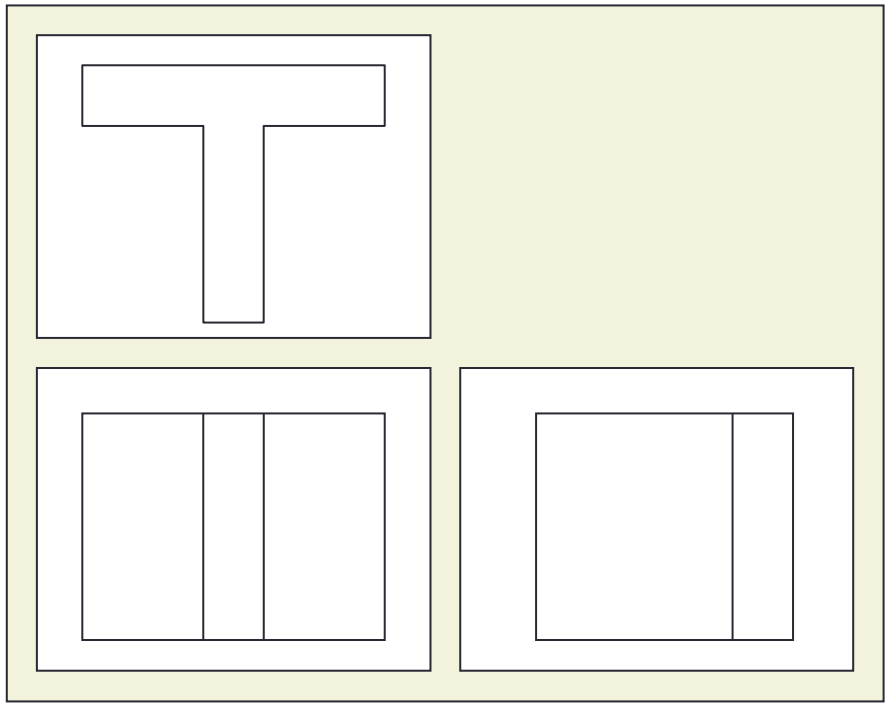


ワイヤーフレームの描画



三面図

(x_s, y_s) 画面上の座標値



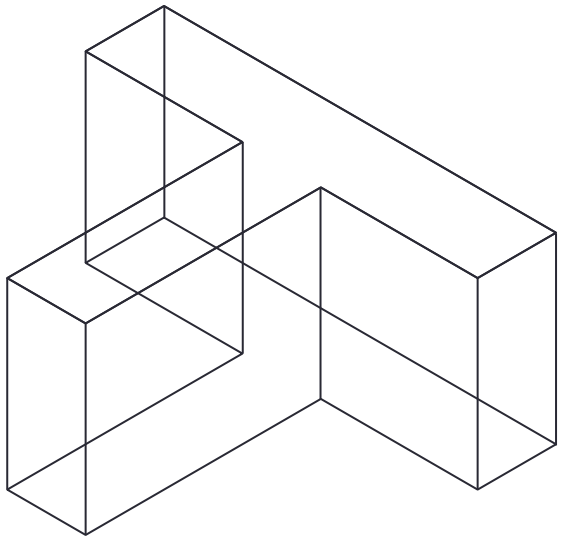
上面図（xz平面）
$x_s \leftarrow x$座標値
$y_s \leftarrow z$座標値

正面図（xy平面）
$x_s \leftarrow x$座標値
$y_s \leftarrow y$座標値

側面図（yz平面）
$x_s \leftarrow z$座標値
$y_s \leftarrow y$座標値

アイソメ図

(x_s, y_s) 画面上の座標値



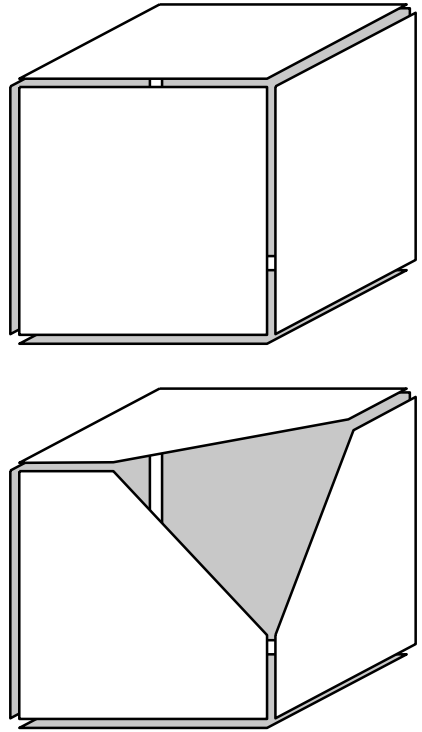
アイソメ図（等測投影図）

$$x_s \neg \sqrt{3}(x+z)/2$$

$$y_s \neg y - (x - z)/2$$

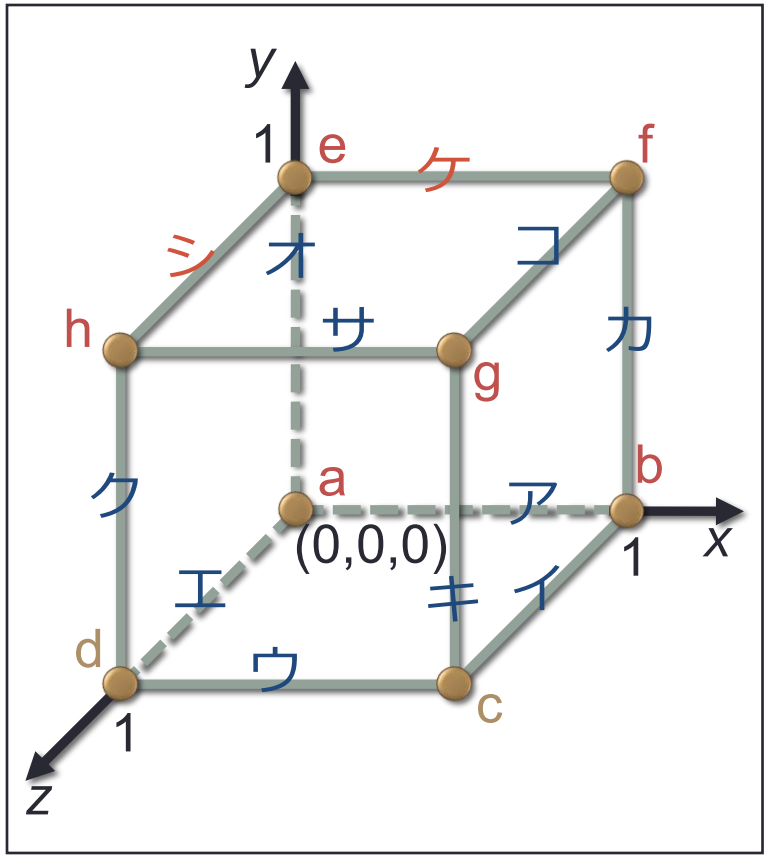
サーフェースモデル

- 面を構成する稜線(あるいは頂点)をひとまとめにして，面のデータとして保持する
  - 内部と外部の区別が無い
  - 立体は閉じていなくても良い
  - 隠面消去可能



切ると中身が空洞になっている

面データ



稜線リスト形式

面	稜線			
下	ア	イ	ウ	エ
右	カ	コ	キ	イ
前	キ	サ	ク	ウ
左	ク	シ	オ	エ
後	オ	ケ	カ	ア
上	シ	サ	コ	ケ

頂点リスト形式

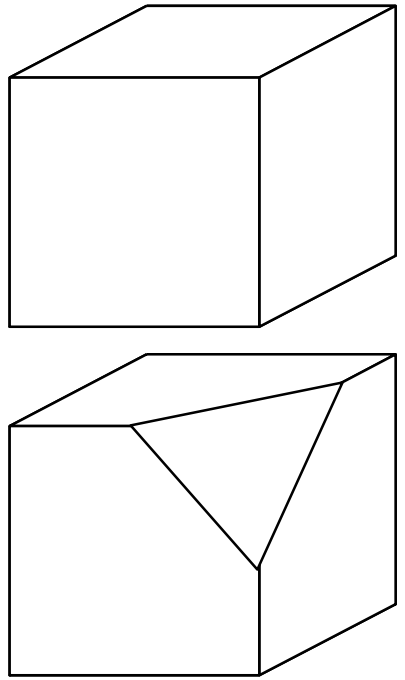
面	頂点			
下	a	b	c	d
右	b	f	g	c
前	c	g	h	d
左	d	h	e	a
後	a	e	f	b
上	h	g	f	e

実際には各欄の要素数は一定ではない
(四角形以外の面もある)

全ての面を三角形で表現することも多い

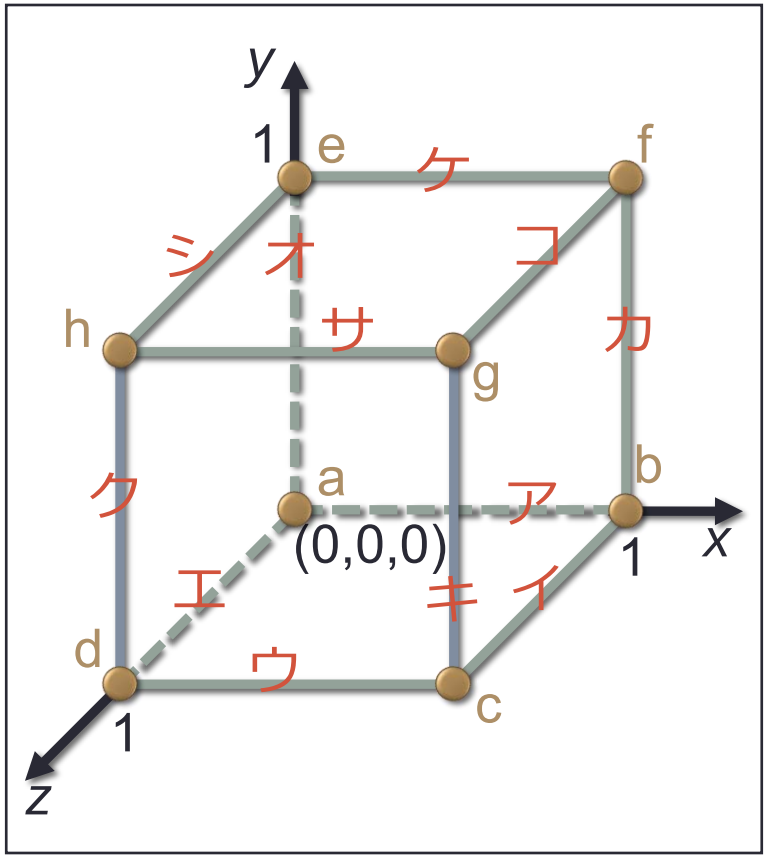
ソリッドモデル

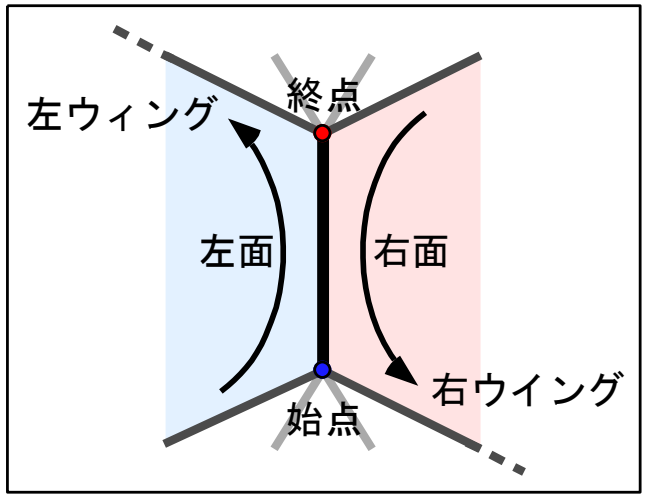
- 立体を構成する面をひとまとめにして立体のデータとして保持する
- 面に向き(表裏)を付ける
  - 立体の内部と外部が区別できる
  - 中身が詰まっているものとして処理できる
  - 集合演算が可能
  - マスプロパティの算出が可能



切ると中身が詰まっている

WED (Winged Edge Data)





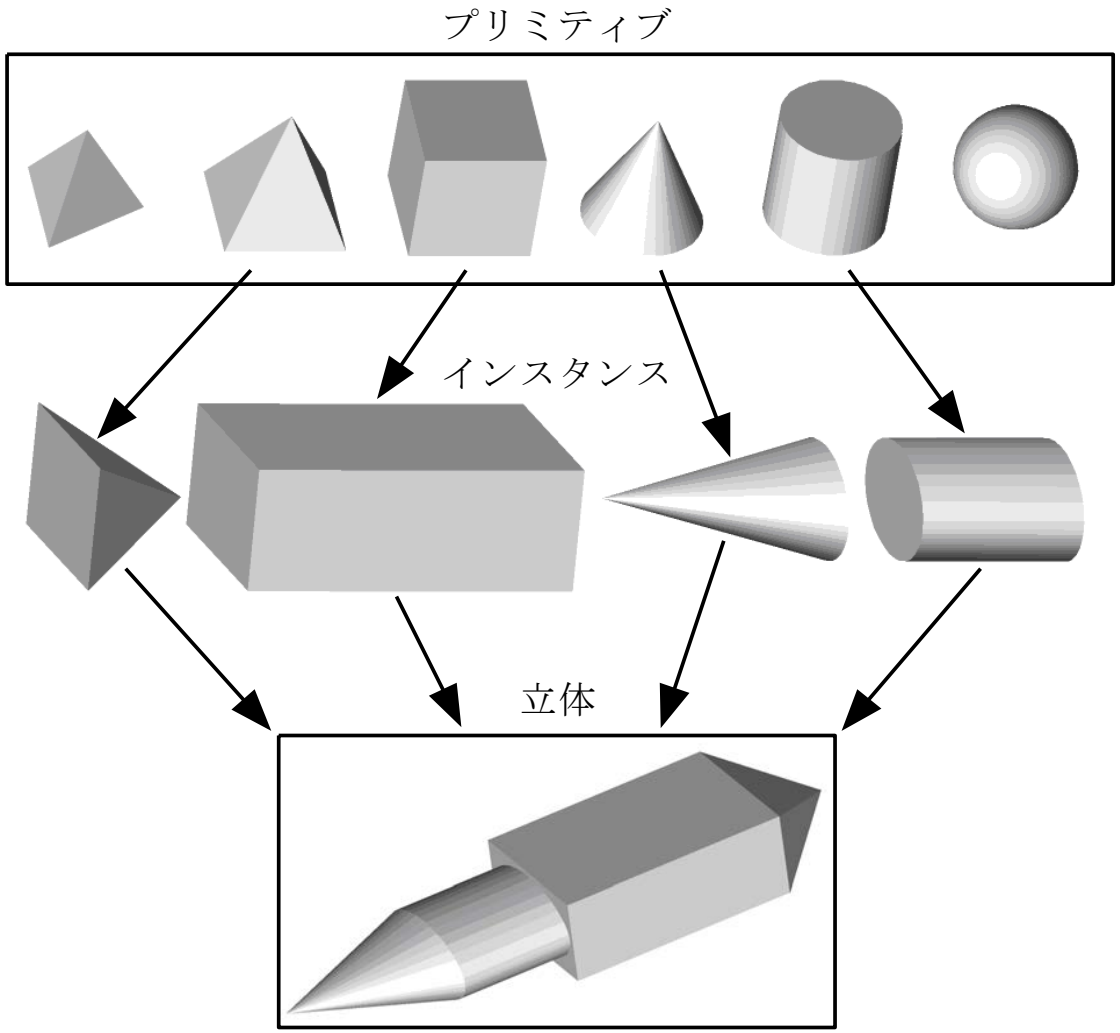
稜線	始点	終点	左ウィング	右ウィング	左面	右面
キ	c	g	サ	イ	前	右
ク	d	h	シ	ウ	左	前

面	法線ベクトル	最初の稜線
前	(0, 0, 1)	キ
右	(-1, 0, 0)	ク

立体の組み合わせによる表現

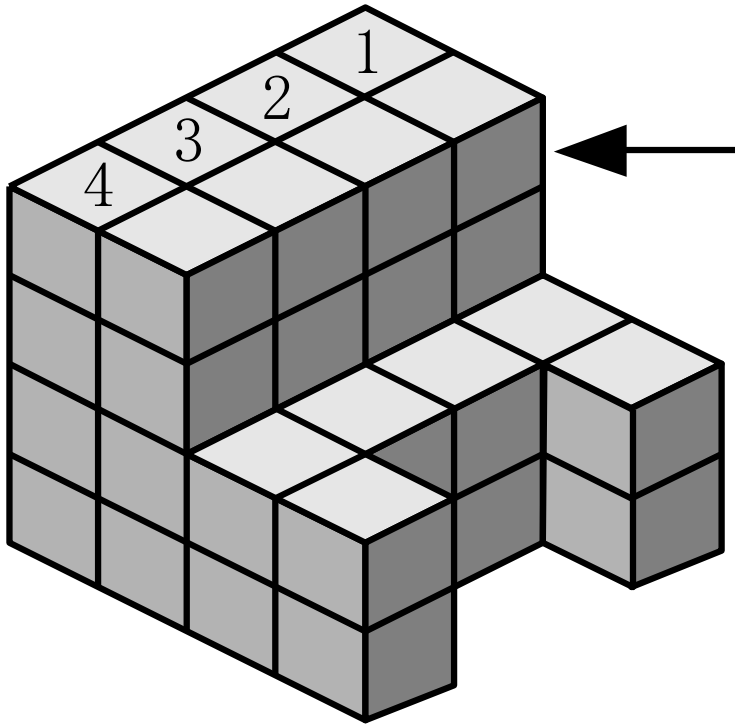
- プリミティブインスタンス法
  - 積み木(パーツは拡大・縮小可能)
- Voxel表現
  - 同じ大きさのブロックの集合
- Octree表現
  - Voxel 8 個をひとつの単位にまとめて階層的に表現

プリミティブインスタンス法



- プリミティブ
  - 基本形状（図形のタイプ）
- インスタンス
  - 拡大縮小などの変形を加えた形状
- 配置して接合
  - 回転と平行移動（剛体変換）

Voxel 表現



○ その空間は空

● その空間は詰まっている

1 2 3 4

●●●● ●●●● ○○○○ ○○○○

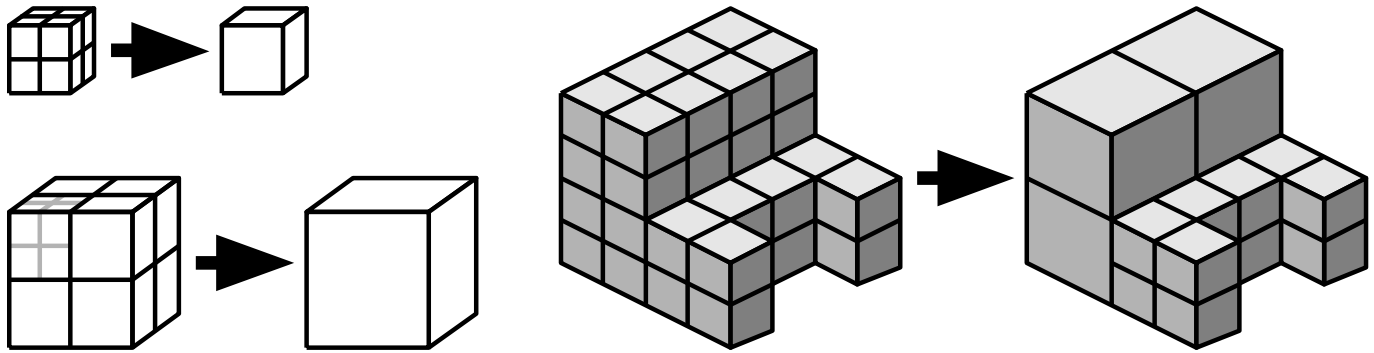
●●●● ●●●● ○○○○ ○○○○

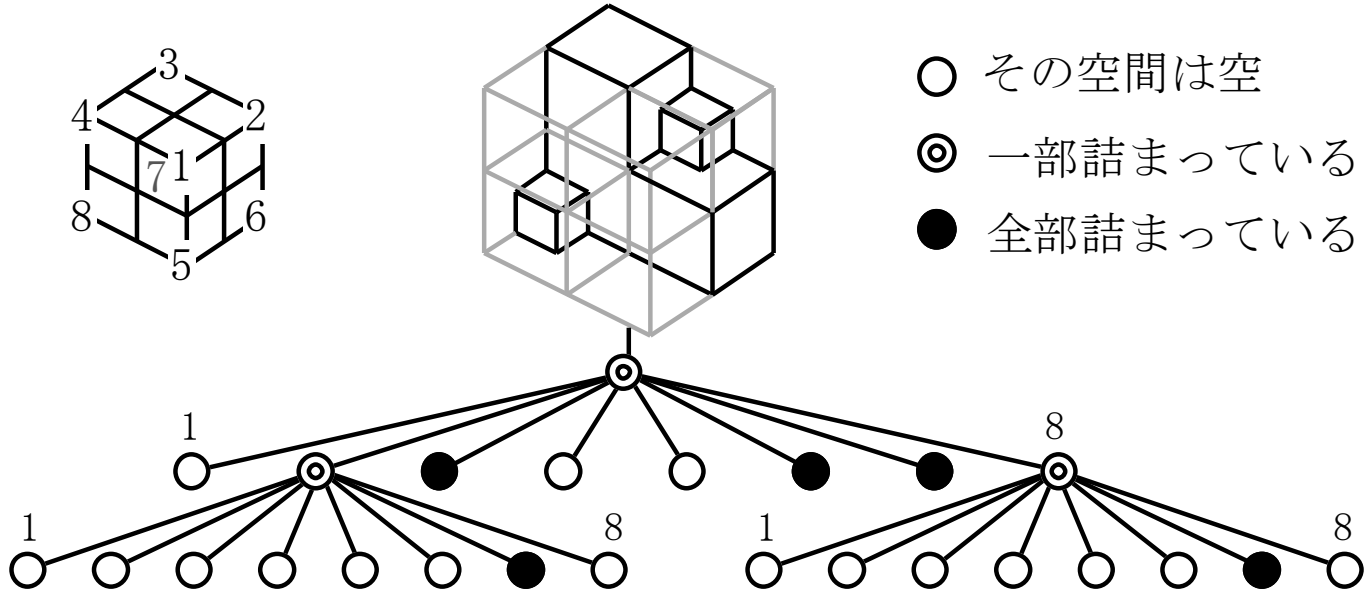
●●●● ●●●● ●●●● ●○○●

●●●● ●●●● ●●●● ●○○●

形状を二進数で表現することができる

Octree（八分木）表現

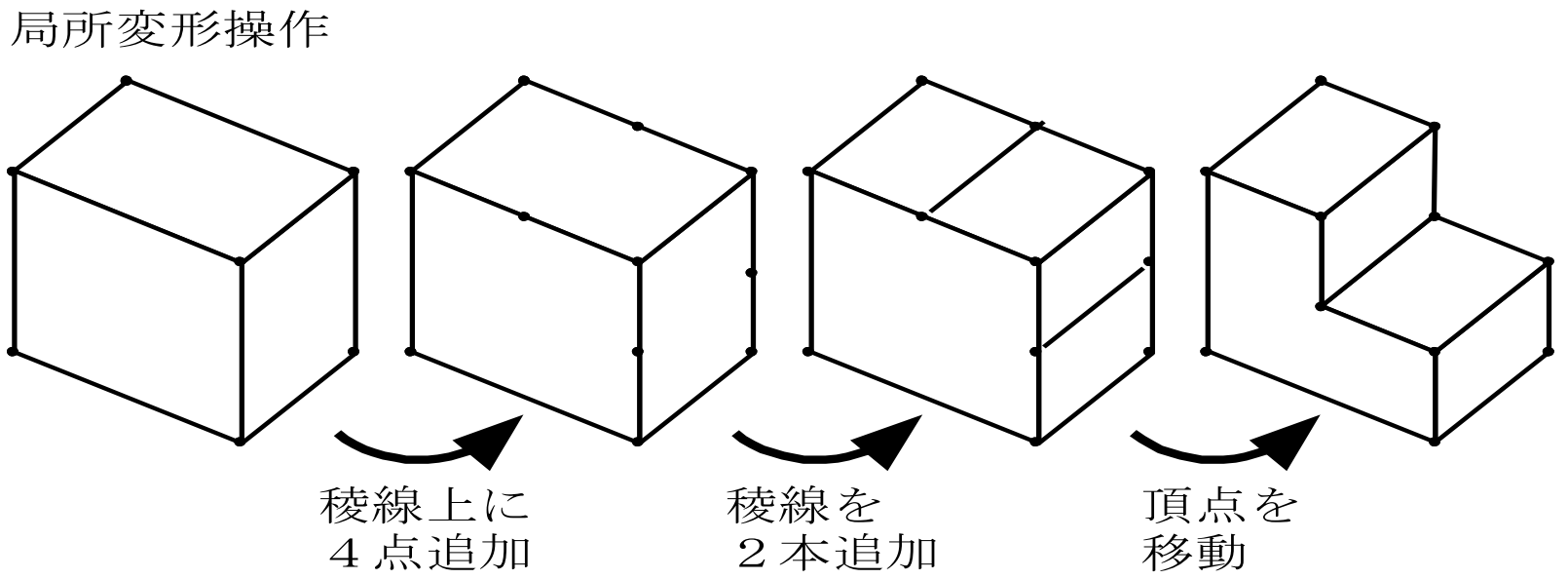


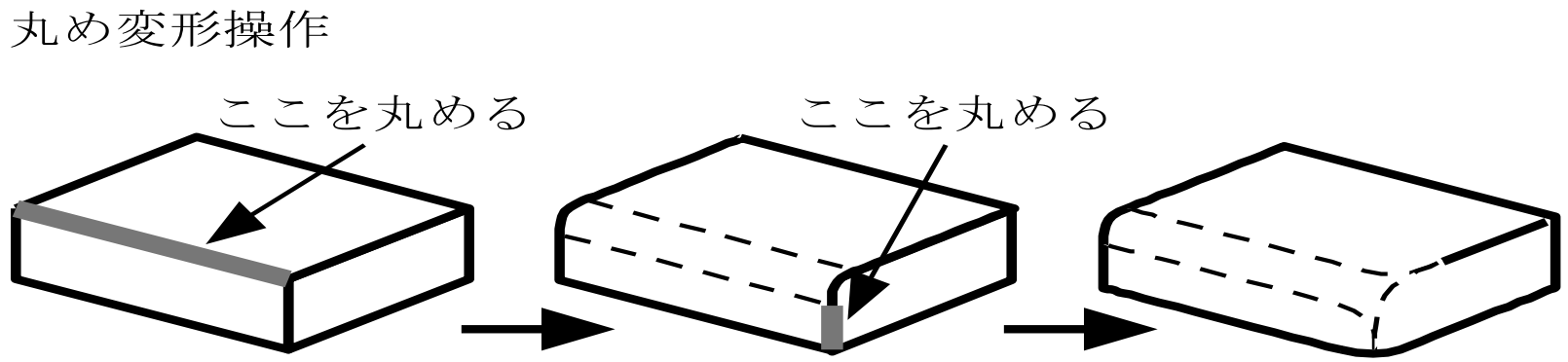


生成過程による表現

- 局所操作の組み合わせによる表現
  - 形状データの変更手順そのものを形状データに使う
- CSG (Constructive Solid Geometry)
  - プリミティブインスタンス法においてプリミティブ間の論理演算を行うもの
- スイープ
  - 掃引体(平面図形を経路に沿って移動したときにできる空間)

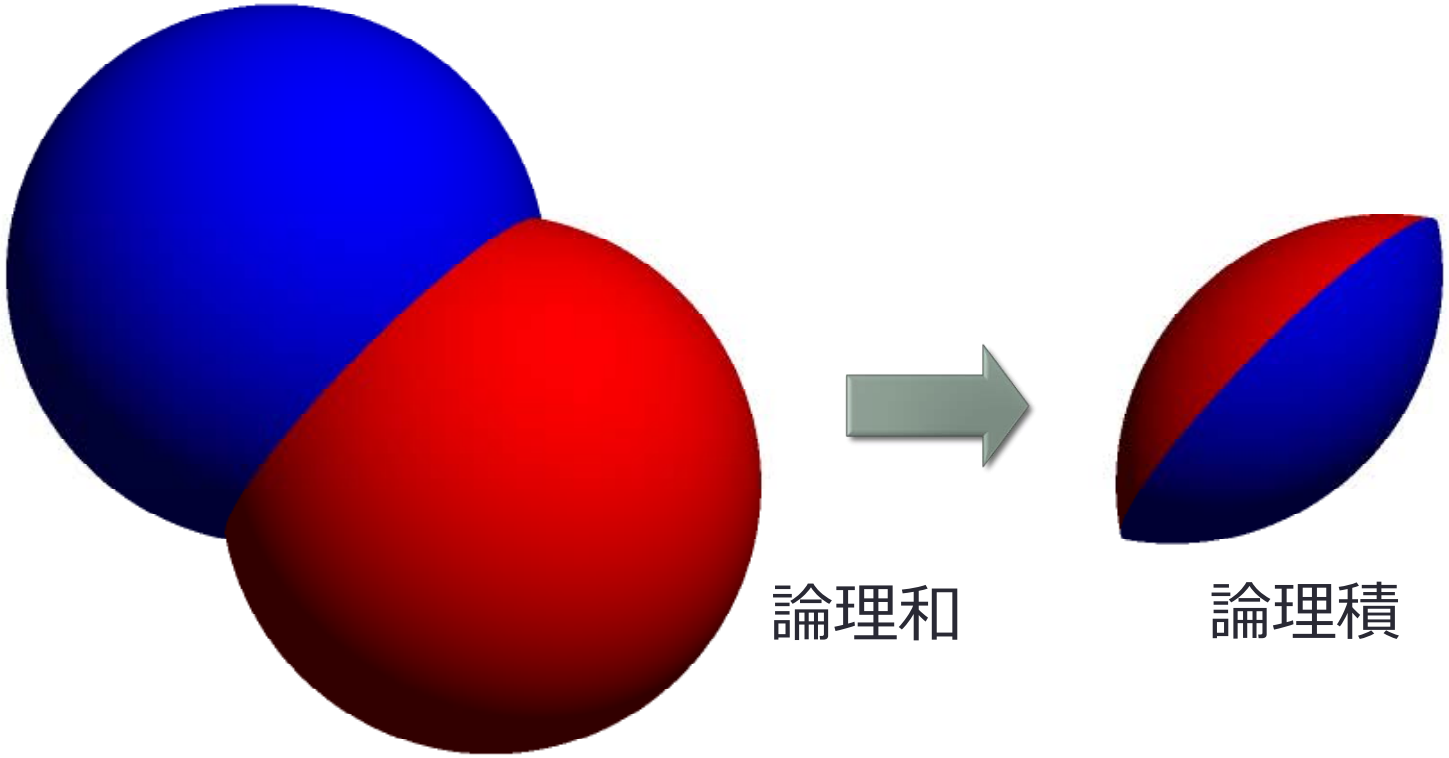
局所変形操作



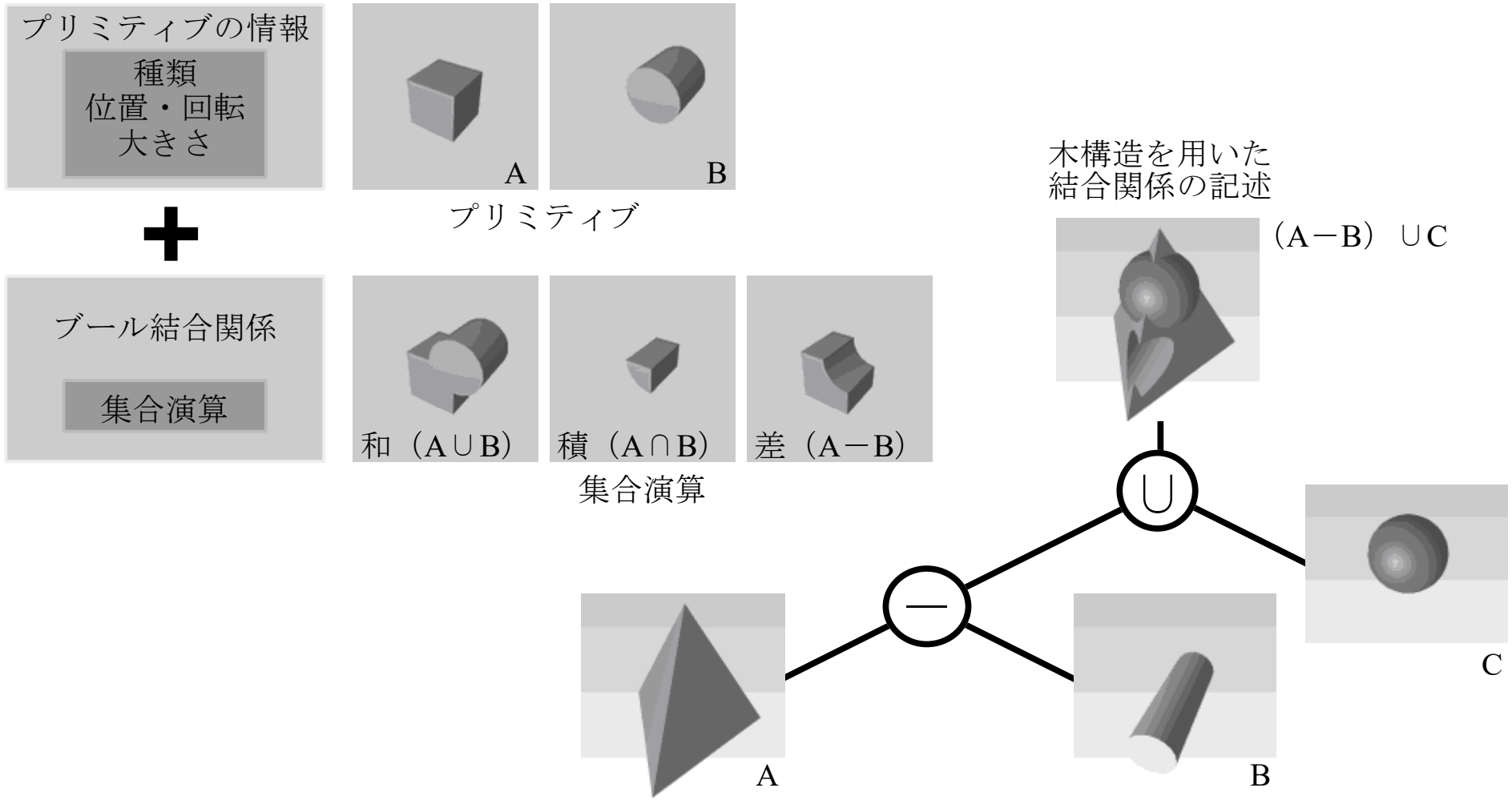


集合演算処理 (Boolean)

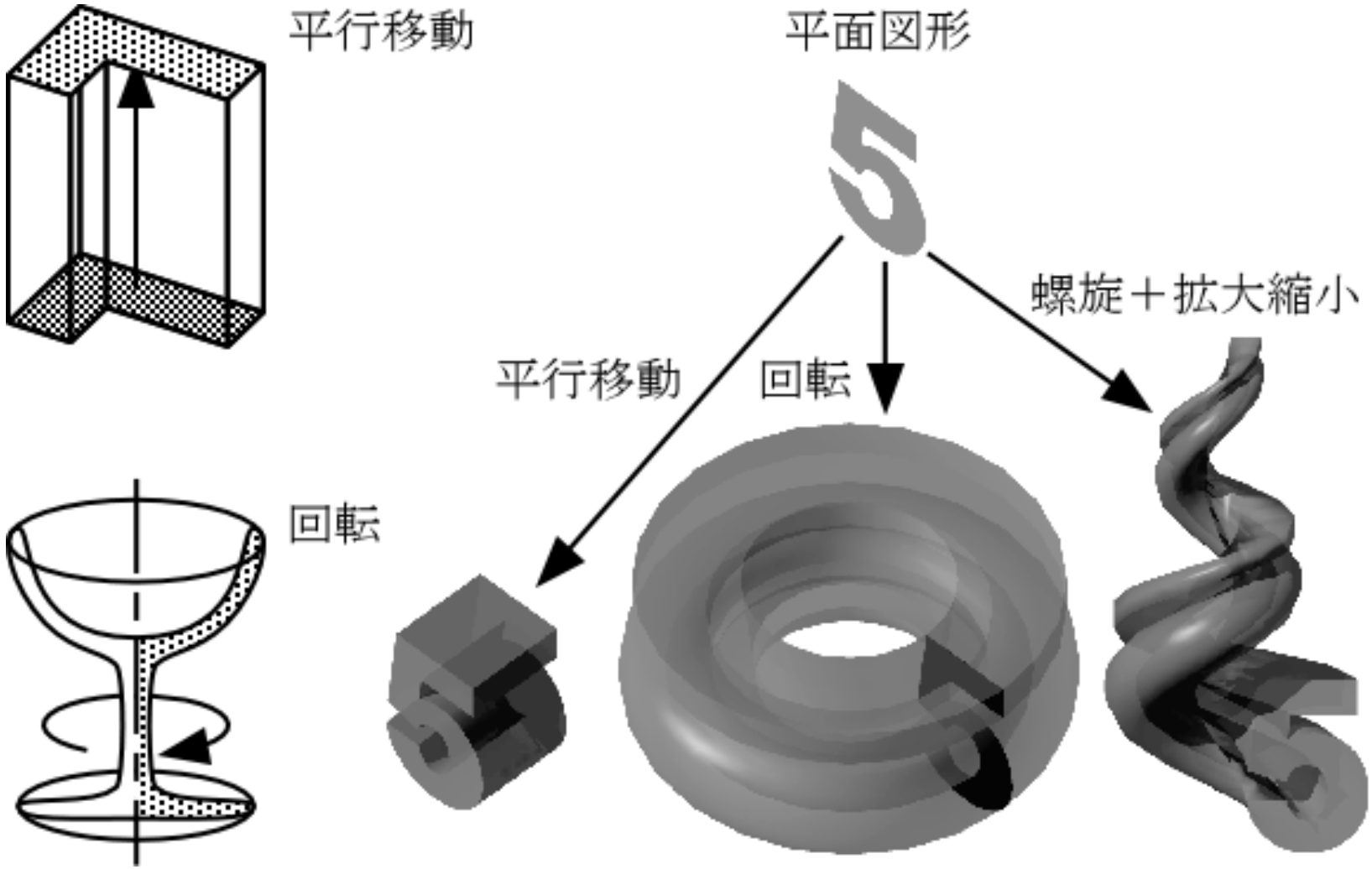
- 2 つの球の共通部分だけを表示するには



CSG (Constructive Solid Geometry)

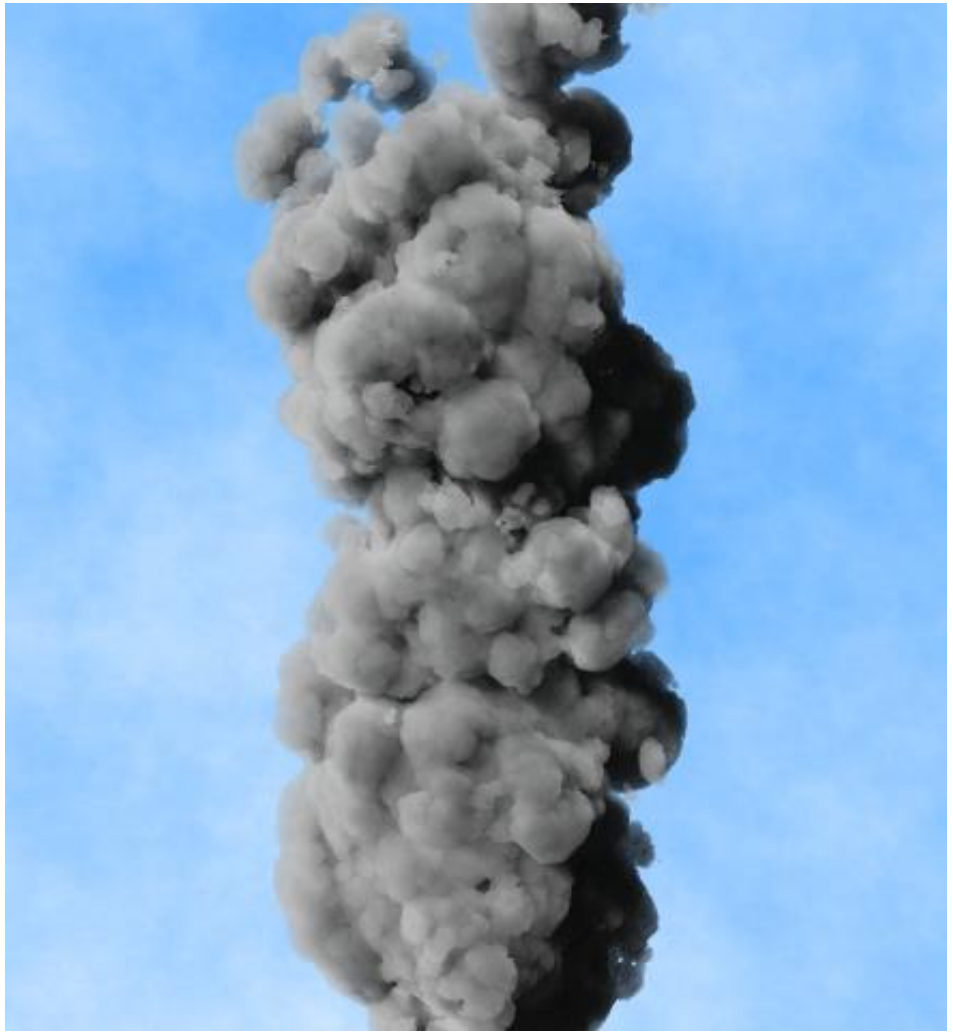


スイープ



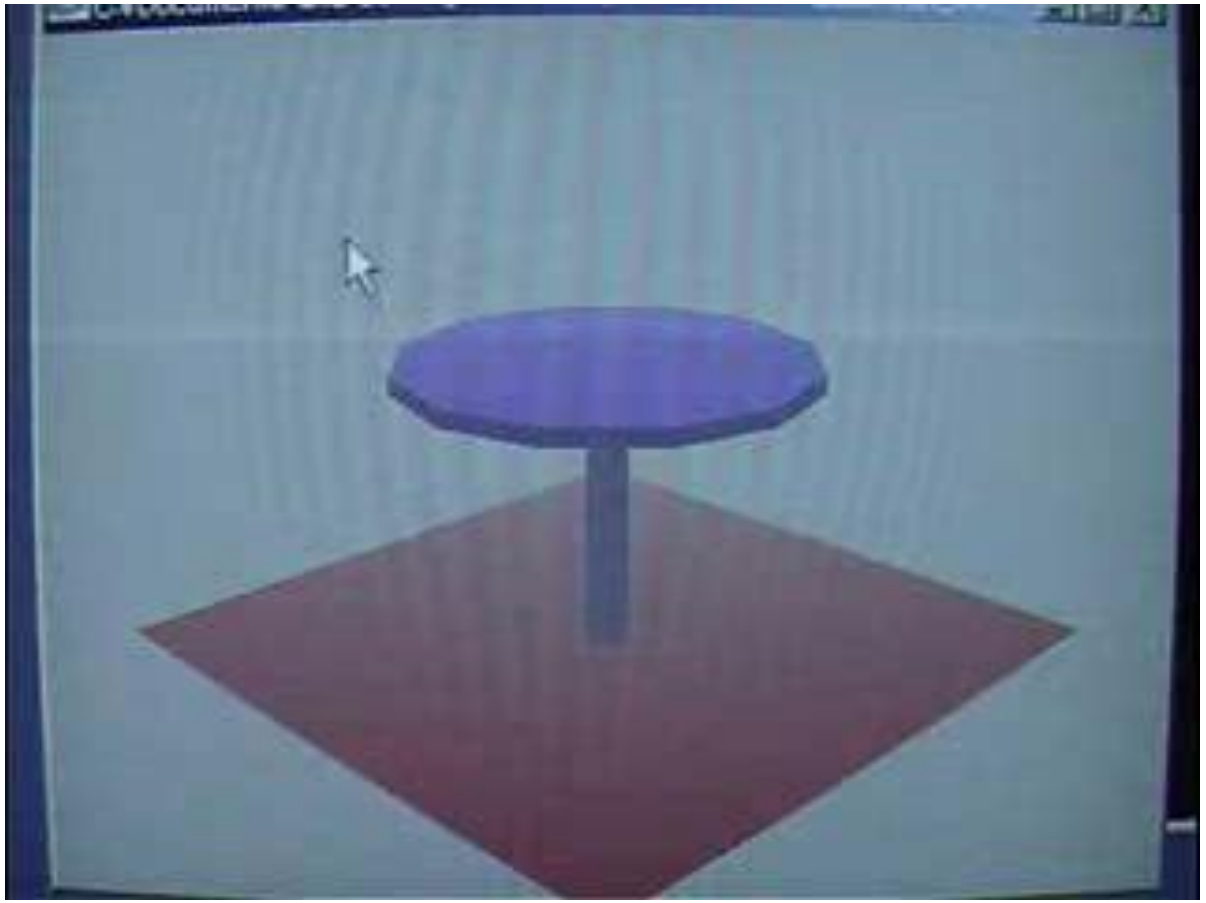
粒子の集まりによる表現

- パーティクル
  - 微粒子の集合体
    - 煙，雲，雪など明確な形を持たないもの
    - 木の葉のように１つ１つは形を持っているが，数が多く集合体として形を表しているもの

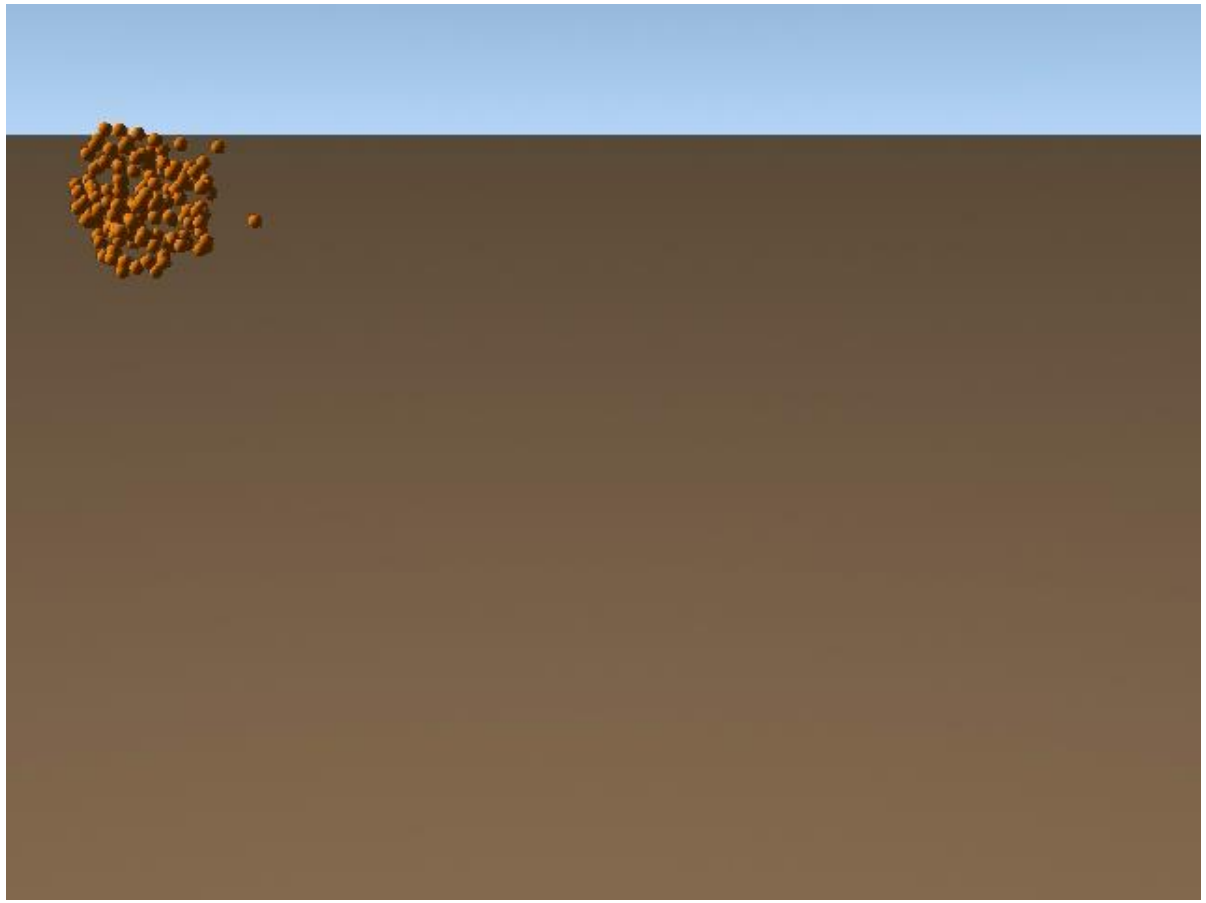


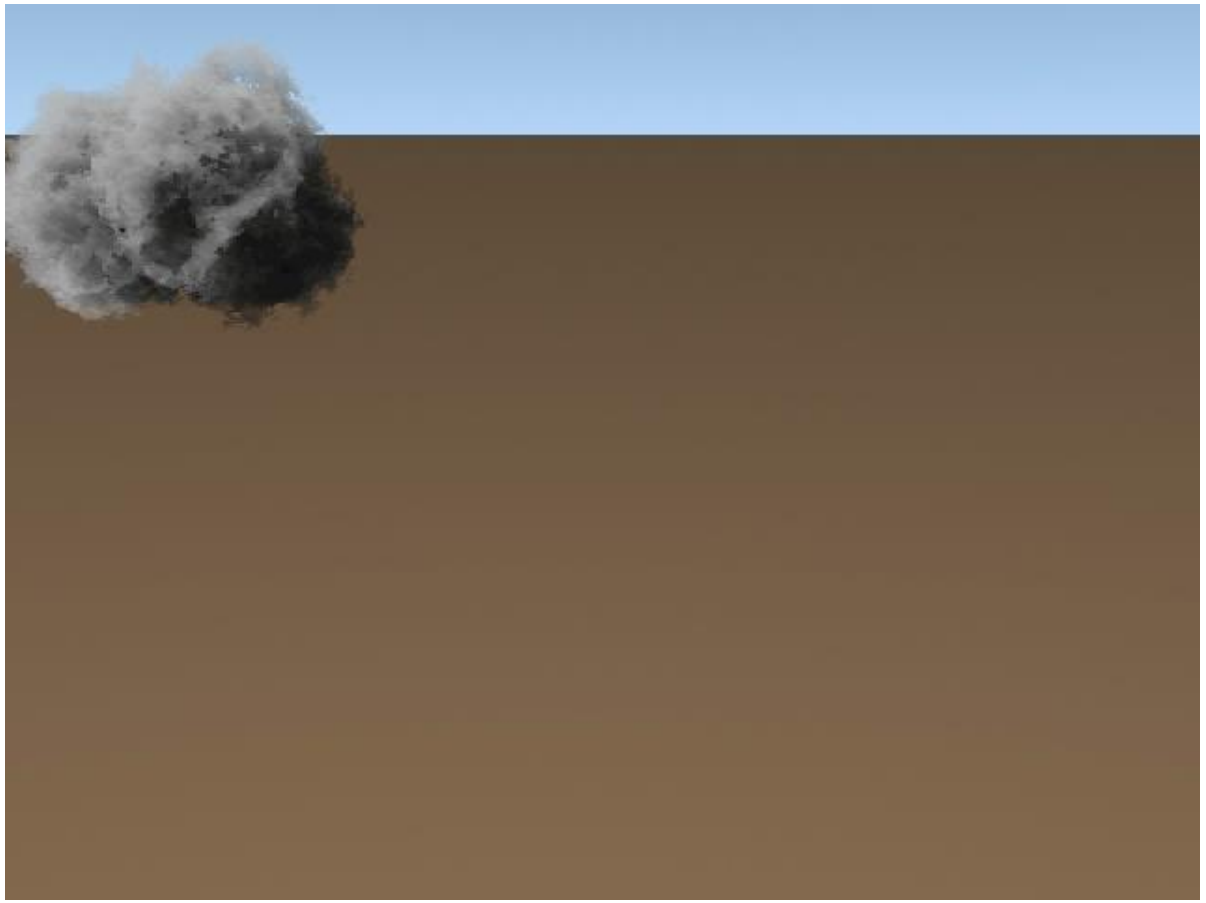
積雪形状のリアルタイムモデリング

- 複雑なシーン上に積もる雪の形状の時間変化の表現
  - 積雪景観の表現



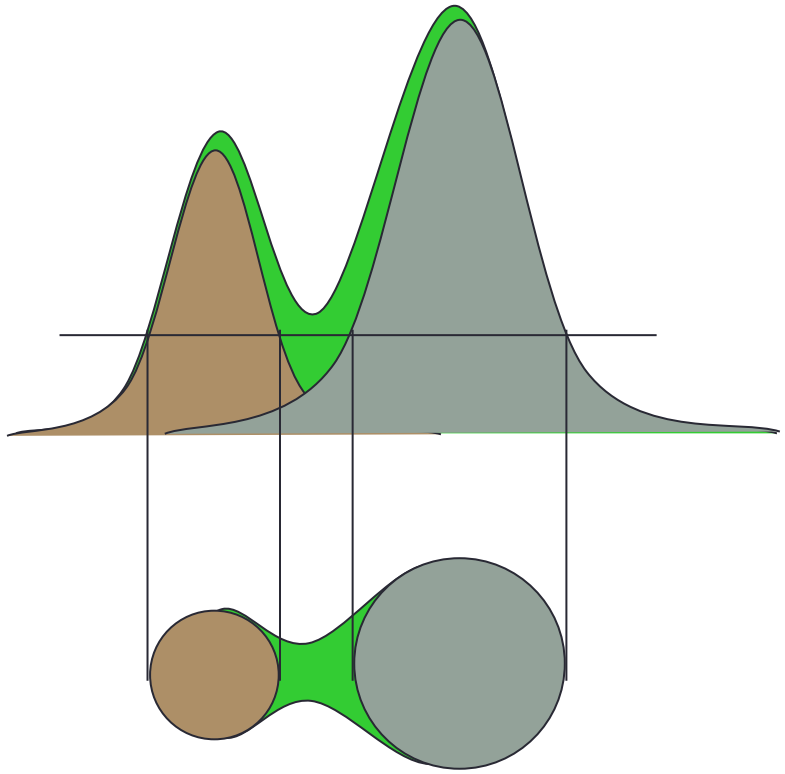
パーティクルとノイズ

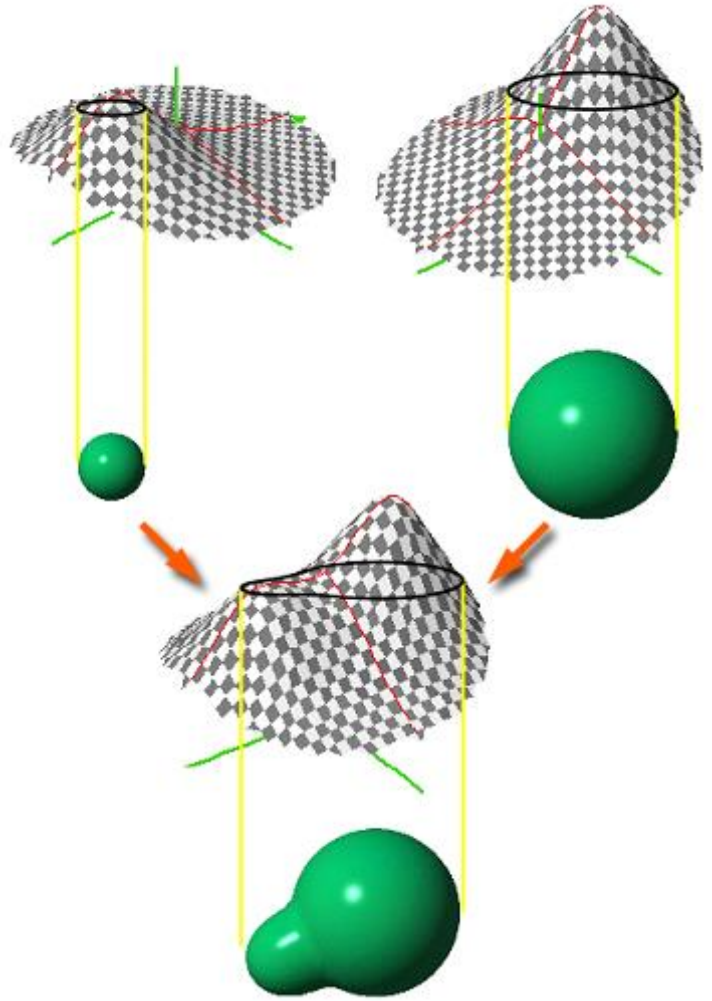




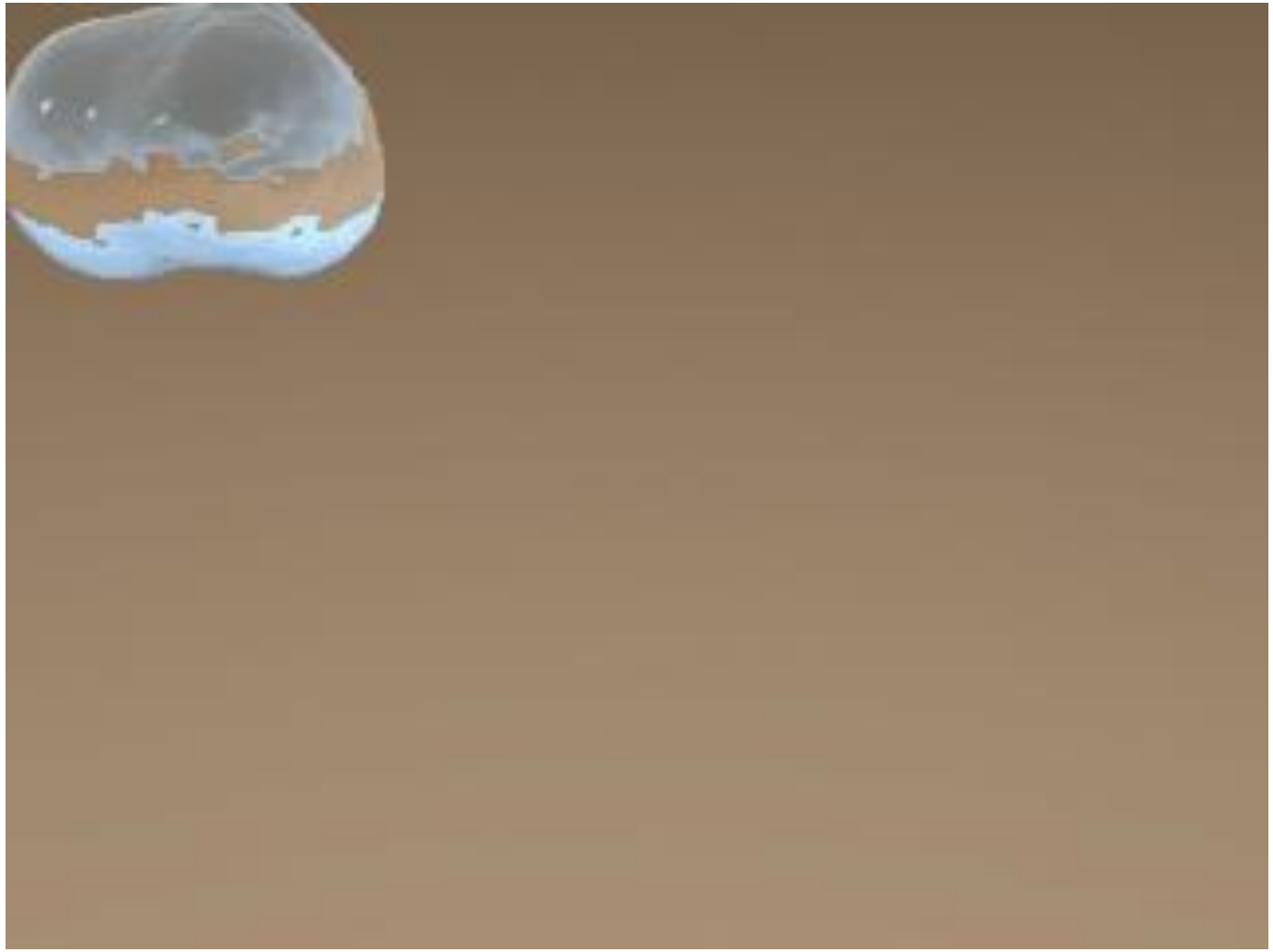
濃度による表現

- メタボール
  - 等値面（等濃度面）を用いて形状を表す





パーティクルとメタボール



「水芸」（って言われた）