???????????????

[illegible] Unity[illegible] transform [illegible]

[illegible]

??????

[illegible]

[illegible] [illegible] [illegible]

$$\vec{v'} = \vec{v} + \vec{t} = \begin{bmatrix} v_x \\ v_y \\ v_z \end{bmatrix} + \begin{bmatrix} t_x \\ t_y \\ t_z \end{bmatrix}$$

[illegible] ***v'***[illegible] ***v***[illegible] ***t***[illegible]

??

MyTransform [illegible] MyCube [illegible]

- [illegible] MyCube [illegible] `GetVertices` [illegible] `SetVertices` [illegible]
  - `GetVertices` [illegible] [illegible] `return (Vector3[])vertices.Clone();`
  - [illegible]
- MyTransform [illegible] MyCube[illegible] `GetComponent` [illegible]
  - [illegible] Inspector [illegible]

- `Translate(Vector3[] vertices)`で移動を実装
- `Update`で
  - 頂点を取得
  - `Translate`で移動
  - 変換した頂点を設定

MyCube の GetVertices と SetVertices

```
// メッシュの頂点を返す
public Vector3[] GetVertices()
{
    //元々の頂点データを守りたいので、コピーを返す
    return (Vector3[])vertices.Clone();
}

// メッシュの頂点を設定
public void SetVertices(Vector3[] vertices)
{
    mesh.vertices = vertices;
    mesh.RecalculateNormals(); //方法ベクトル再計算
}
```

MyTransform スクリプト

```
// 自分で作った変換(Transform) コンポーネント
public class MyTransform : MonoBehaviour
{
    // 位置・移動量
    [SerializeField]
    private Vector3 position;

    // 立方体
    private MyCube cube;

    private void Start()
    {
        // MyCubeを取得し、変換する前の頂点を保管しておく
```

```
        cube = GetComponent<MyCube>();
    }

    private void Update()
    {
        // 頂点を取得
        Vector3 [] vertices = cube.GetVertices();

        // 移動を適用
        Translate(vertices);

        // 変換した頂点を適用
        cube.SetVertices(vertices);
    }

    // 頂点を移動する
    private void Translate(Vector3[] vertices)
    {
        // 1個ずつを処理
        for (int i = 0; i < vertices.Length; i++)
        {
            // ベクトルの加算で移動を実現!
            vertices[i] = vertices[i] + position;
        }
    }
}
```

??????????

拡大縮小は「すべての頂点を大きくする」または「小さくする」ということで、それぞれの頂点の「倍率」

の値をかければ良いでしょう（ ベクトルと実数の掛け算 ）

$$\overrightarrow{v'} = \overrightarrow{v} \cdot s = \begin{bmatrix} v_x \cdot s \\ v_y \cdot s \\ v_z \cdot s \end{bmatrix}$$

??

MyTransform を編集して ...

- Inspectorで倍率の係数を変えれるようにする（デフォルト値: 1）
- `Update`で、 `Translate` の前に `Scale` を適用

MyTransform スクリプト（編集分）

```
// 拡大縮小
[SerializeField]
private float scale = 1;

private void Update()
{
//頂点を取得
Vector3 [] vertices = cube.GetVertices();

//拡大縮小
Scale(vertices);

//移動を適用
Translate(vertices);

//変換した頂点を適用
Cube.SetVertices(vertices);
}

private void Scale(Vector3[] vertices)
{
// 1ずつを処理
for (int i = 0; i < vertices.Length; i++)
{
//ベクトルの乗算で拡大縮小を実現!
Vertices[i] = vertices[i] * scale;
}
}
```

???????

よく見れば、 Unityでは軸は別々でスケールできるよね。作ってきた「 Scale 」は3つの軸を同時にスケールしてしまう。ならば、少し式を変えないといけないが、ベクトル同士の掛け算ができないはず …

$$\overrightarrow{v'} = \begin{bmatrix} v_x \cdot s_x \\ v_y \cdot s_y \\ v_z \cdot s_z \end{bmatrix}$$

ベクトル「 v」と「 s 」の各要素の乗算したいけど、ベクトル同士の乗算がない（外積と内積どっちも使えない！）

ちょっと待って …「各要素の乗算」って 行列でできる_____のでは? たしかに、これができる!

$$\overrightarrow{v'} = S \cdot \overrightarrow{v} = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & s_z \end{bmatrix} \cdot \begin{bmatrix} v_x \\ v_y \\ v_z \end{bmatrix} = \begin{bmatrix} v_x \cdot s_x \\ v_y \cdot s_y \\ v_z \cdot s_z \end{bmatrix}$$

ここで「 S」のことは「スケール行列」といい、各頂点にかければ、正しく拡大縮小を実装できる。

??

MyTransform を編集して …

- 倍率の処理を削除
- Inspectorで x, y, zの各軸の拡大縮小ををを変えれるようにする（デフォルト値: 1, 1, 1）
- `Scale` を更新し、スケール行列を作成し、変換してください。

MyTransform スクリプト（編集分）

```
// 拡大縮小
[SerializeField]
private Vector3 scale = new Vector3(1, 1, 1);

private void Scale(Vector3[] vertices)
{
//スケール行列を作成
Matrix s = new Matrix(3, 3);
```

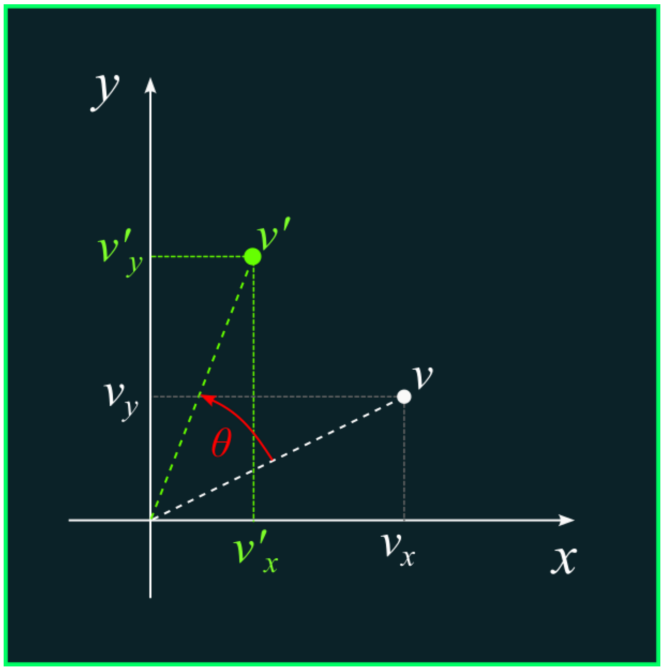
```
□s.Set(0, 0, scale.x);
□s.Set(1, 1, scale.y);
□s.Set(2, 2, scale.z);
□
□// □ずつを処理
□for (int i = 0; i < vertices.Length; i++)
□{
□□/□列の乗算で拡大縮小を実現!
□Vertices[i] = s.Multiply(vertices[i]);
□}
}
```

??

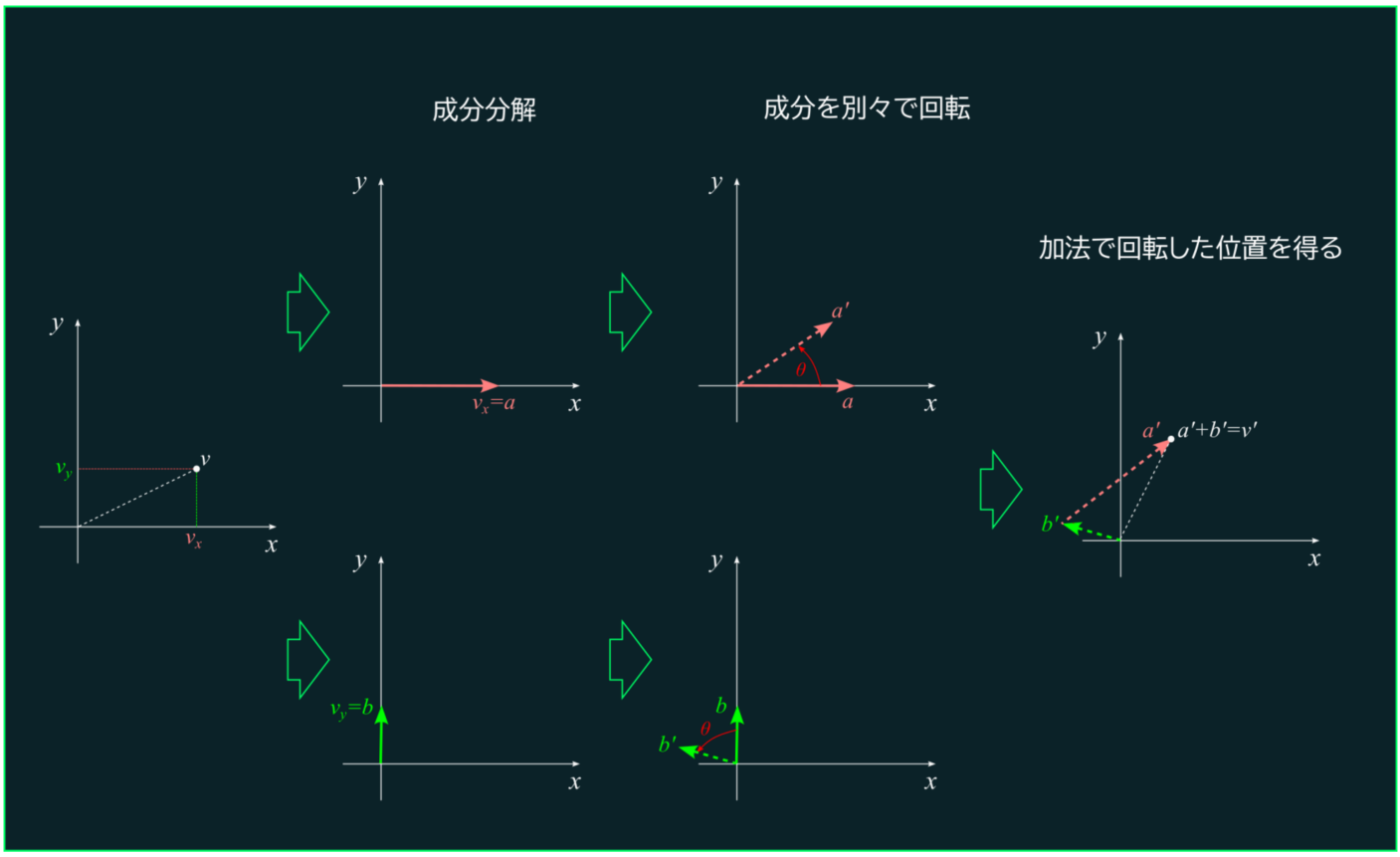
これはややこしい　　...

2?????

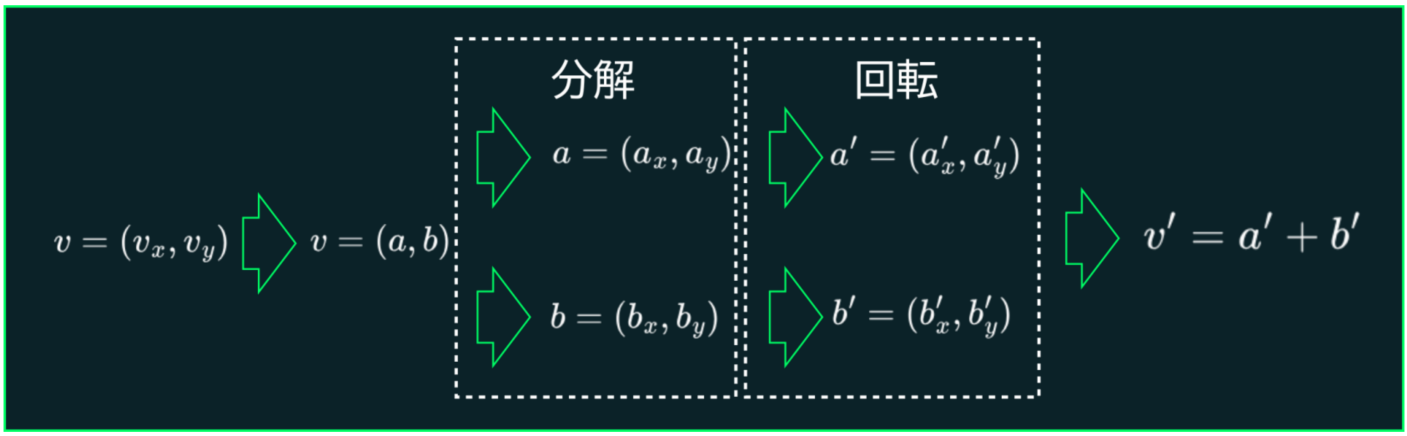
まず、　2Dの回転を考えてみましょう。　　三角関数を使って回転できる＿＿＿＿＿＿＿＿はずなので　　...



ある頂点「　　$v=(v_x, v_y)$」の位置を角度「　　θ」で回転したら、新しい位置「　　$v'=(v'_x, v'_y)$」を求めたい。これで難しいので、問題を分解して解決しましょう。



ベクトル **v=(v_x, v_y)** を成分 **v_x** と **v_y** に分解し、別々で回転し、加法でまた合体すれば、回転した **v'** を求めることができる。名前をわかりやすくするため各成分のベクトルを「 a」と「 b」にしましょう。

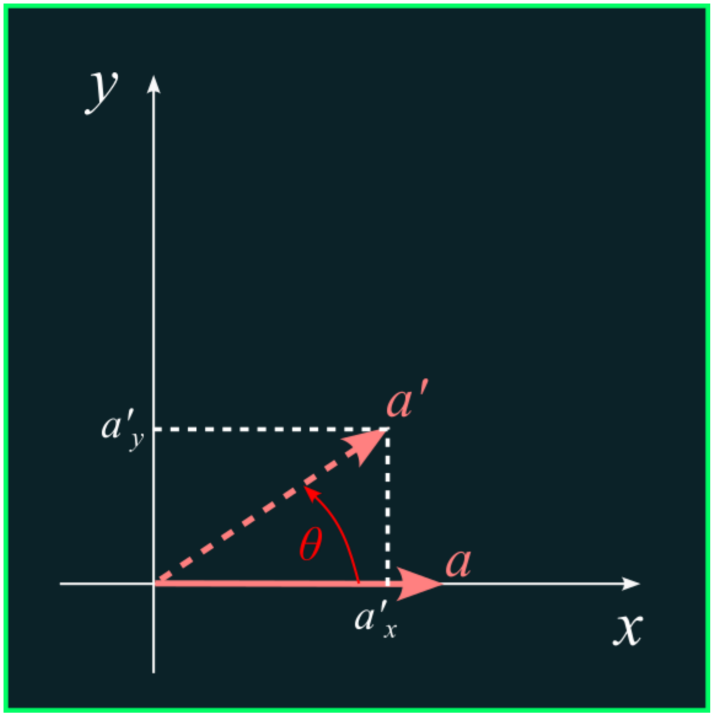


?a????

「 a

」の回転には、単位円、および三角関数で学んだ「サイン」と「コサイン」で回転できることがわかる。確

か:



「 a」を「 θ」度で回転すれば、 a' になる。ここで:

$$a'_x = \left\| \overrightarrow{a} \right\| cos(\theta) \qquad a'_y = \left\| \overrightarrow{a} \right\| sin(\theta)$$

ただし、「 a」は「 v」の「 x」成分なので、以下は事実である:

$$\overrightarrow{a} = (v_x, 0)$$

ので、ベクトル「 a」のノルムは:

$$\left\| \overrightarrow{a} \right\| = v_x$$

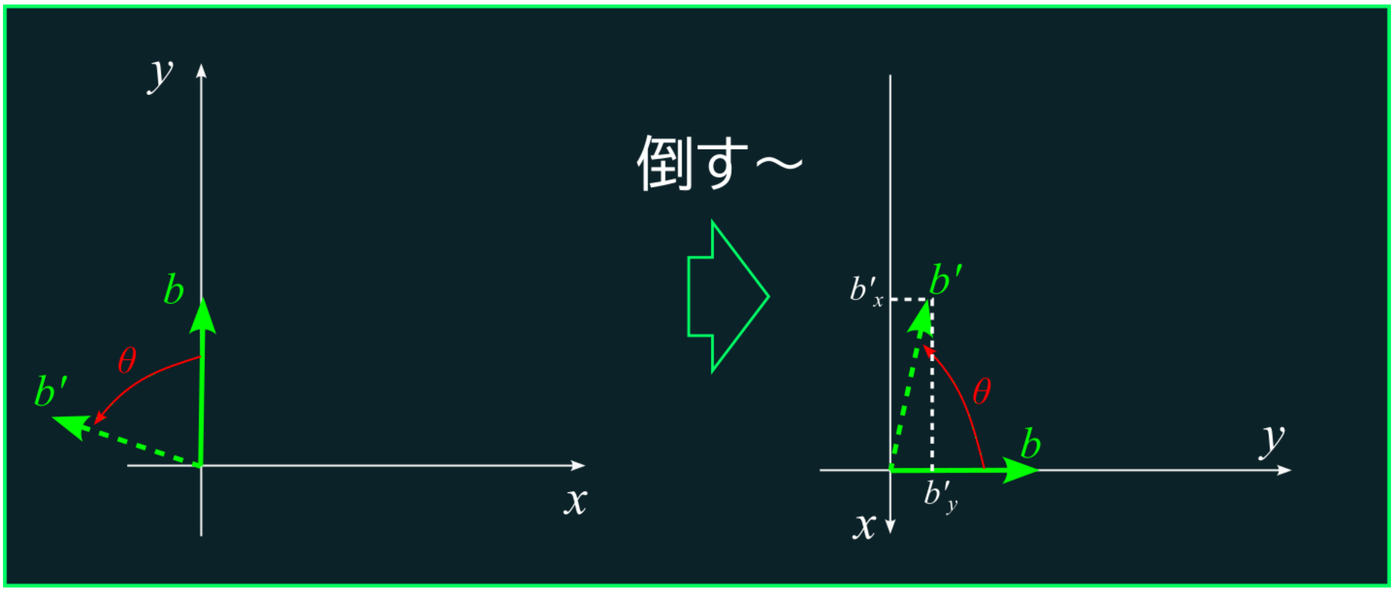
つまり

$$a'_x = v_x \, cos(\theta) \qquad a'_y = v_x \, sin(\theta)$$

?b????

「 b」の回転は同じ考え方で行うのですが、幾何学的にはわかりにくいので、 ***xy***

座標系を倒してみましょう:



□□□ x □ y □□□

$$b'_x = -\left\|\vec{b}\right\| sin(\theta) \qquad b'_y = \left\|\vec{b}\right\| cos(\theta)$$

□□□ b □□ v □□ y □□□

$$\vec{b} = (0, v_y)$$

□□□ b □□□

$$\left\|\vec{b}\right\| = v_y$$

□□□

$$b'_x = -v_y\, sin(\theta) \qquad b'_y = v_y\, cos(\theta)$$

?a'+b'????

□□□ v □□ v' □□□ a' □□ b' □□□

$$v' = a' + b'$$

ベクトルの加法は各成分の足し算なので:

$$(v'_x, v'_y) = (a'_x + b'_x, a'_y + b'_y)$$

上記で求めた式を書き換えるとこうなる:

$$v'_x = v_x \cos(\theta) - v_y \sin(\theta) \qquad v'_y = v_x \sin(\theta) + v_y \cos(\theta)$$

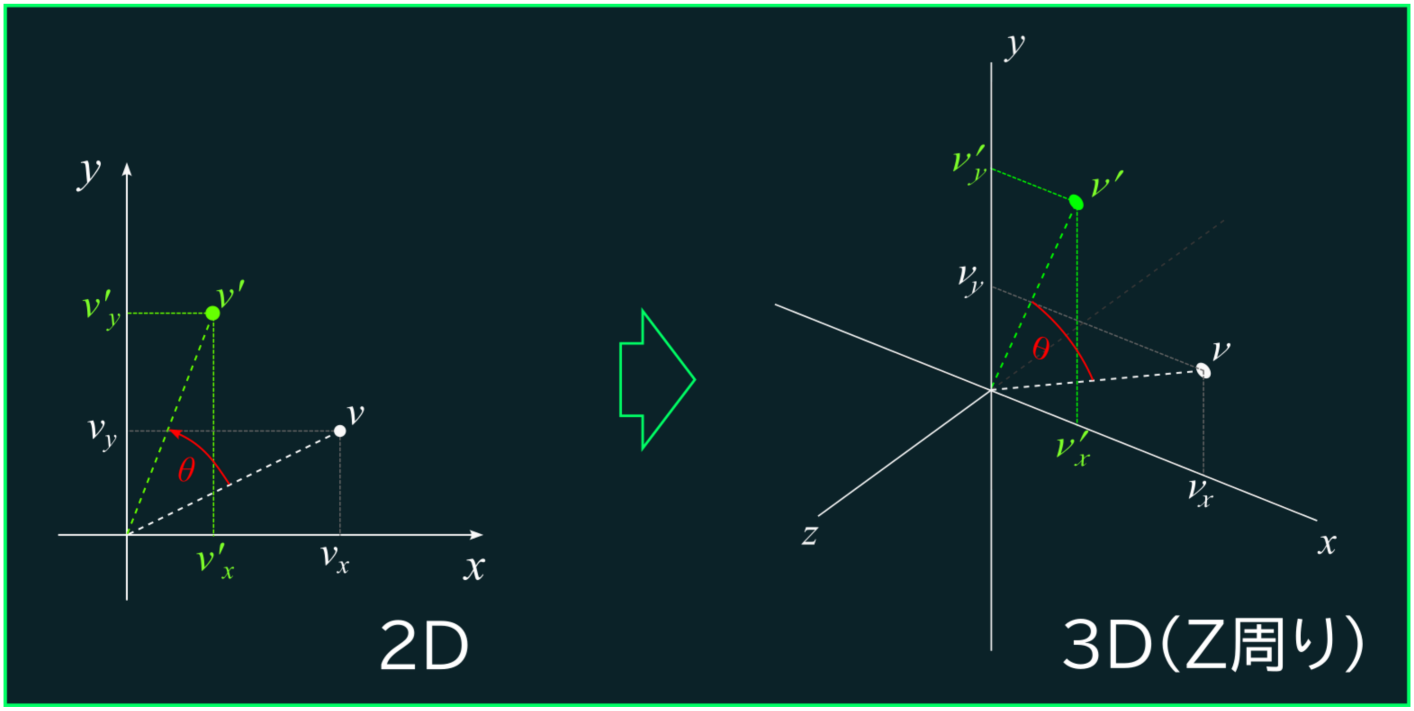
これは回転ベクトル「　　v'」になる。ただし、多少読みにくいので、　　ベクトル　　×行列として表現ができる:

$$R(\theta) \times \overrightarrow{v} = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix} \times \begin{bmatrix} v_x \\ v_y \end{bmatrix}$$

ここで「　　R(θ)」は「　　回転行列　　」という。ある2次元の位置（頂点）「　　v」を角度「　　θ　　」を回転したいなら、この行列の掛け算で回転した後に位置を求めることができる。

3?????

2次元の場合は問題ないが、　　3Dの場合はどう変わる? まず、　　Z　　軸周りの回転を見てみましょう。実は、この回転は　　2Dの回転と全く同じであり、　　Z　　の値はずっとゼロになるだけ:



つまり:

$$R_z(\theta) \times \overrightarrow{v} = \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} v_x \\ v_y \\ v_z \end{bmatrix}$$

これは「 Z軸周りの回転行列 」である。上記と同じ考えを持ち、 X軸と Y軸周りの回転行列を求めることができる

$$R_x(\theta) \times \overrightarrow{v} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\theta) & \sin(\theta) \\ 0 & -\sin(\theta) & \cos(\theta) \end{bmatrix} \times \begin{bmatrix} v_x \\ v_y \\ v_z \end{bmatrix}$$

$$R_y(\theta) \times \overrightarrow{v} = \begin{bmatrix} \cos(\theta) & 0 & -\sin(\theta) \\ 0 & 1 & 0 \\ \sin(\theta) & 0 & \cos(\theta) \end{bmatrix} \times \begin{bmatrix} v_x \\ v_y \\ v_z \end{bmatrix}$$

??

MyTransform を編集して ...

- Inspectorで x, y, zの各軸の回転角度を設定できるようにする。
- 各軸を回転行列を作成するメソッドを作成してください
  - `Matrix MakeRotX();`
  - `Matrix MakeRotY();`
  - `Matrix MakeRotY();`
- `void RotateEuler(Vector3[] vertices)` (オイラー角度の回転) メソッドを実装
  - 3つ回転行列を作成し、 X→Y→Zの順番で処理し、回転する

MyTransform スクリプト (編集分)

```
// X軸周りの回転行列を作成
private Matrix MakeRotX()
{
```

```
    // 度→ラジアン
    float radian = Mathf.Deg2Rad * rotation.x;

    // サインとコサインを計算
    float c = Mathf.Cos(radian);
    float s = Mathf.Sin(radian);

    // 行列を構築
    Matrix m = new Matrix(3, 3);
    m.Set(0, 0, 1);
    m.Set(0, 1, 0);
    m.Set(0, 2, 0);

    m.Set(1, 0, 0);
    m.Set(1, 1, c);
    m.Set(1, 2, s);

    m.Set(2, 0, 0);
    m.Set(2, 1, -s);
    m.Set(2, 2, c);

    return m;
}

// Y軸周りの回転行列を作成
private Matrix MakeRotY()
{
    // 度→ラジアン
    float radian = Mathf.Deg2Rad * rotation.y;

    // サインとコサインを計算
    float c = Mathf.Cos(radian);
    float s = Mathf.Sin(radian);

    // 行列を構築
    Matrix m = new Matrix(3, 3);
    m.Set(0, 0, c);
    m.Set(0, 1, 0);
```

```
    m.Set(0, 2, -s);

    m.Set(1, 0, 0);
    m.Set(1, 1, 1);
    m.Set(1, 2, 0);

    m.Set(2, 0, s);
    m.Set(2, 1, 0);
    m.Set(2, 2, c);

    return m;
}

// Z軸周りの回転行列を作成
private Matrix MakeRotZ()
{
    // 度→ラジアン
    float radian = Mathf.Deg2Rad * rotation.z;

    // サインとコサインを計算
    float c = Mathf.Cos(radian);
    float s = Mathf.Sin(radian);

    // 行列を構築
    Matrix m = new Matrix(3, 3);
    m.Set(0, 0, c);
    m.Set(0, 1, -s);
    m.Set(0, 2, 0);

    m.Set(1, 0, s);
    m.Set(1, 1, c);
    m.Set(1, 2, 0);

    m.Set(2, 0, 0);
    m.Set(2, 1, 0);
    m.Set(2, 2, 1);

    return m;
```

```
}

// オイラー角度の回転
private void RotateEuler(Vector3[] vertices)
{
    Matrix rotX = MakeRotX();
    Matrix rotY = MakeRotY();
    Matrix rotZ = MakeRotZ();

    // 1個ずつを処理
    for (int i = 0; i < vertices.Length; i++)
    {
        // X→Y→Zの順番で回転
        Vector3 v = vertices[i];
        v = rotX.Multiply(v);
        v = rotY.Multiply(v);
        v = rotZ.Multiply(v);
        vertices[i] = v;
    }
}
```

??????????????

スケールと回転とも 3x3の行列で、「行列 × ベクトル」のパターンを使うことができるが、位置（移動）は、ベクトル同士の足し算なので、処理の一体化できない。すべての変換を統一できる行列にできれば、処理も統一できるので、新しい概念を学びましょう。

ちょっと見てみましょう：各成分（ x, y, z）を移動量（ tx, ty, tz ）を足さないといけないので、行列にするとしたら、こういう作戦がある。 3x3 の行列ならば、各成分の掛け算になる＿＿＿＿＿＿＿＿＿＿＿＿が、以下のように工夫すれば …

$$\vec{v} = \begin{bmatrix} v_x \\ v_y \\ v_z \\ 1 \end{bmatrix}$$

4x4の行列との掛け算ができるので、 移動行列 を、以下のように作ることができる:

$$T = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

ベクトルと行列の掛け算をすると:

$$\begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} v_x \\ v_y \\ v_z \\ 1 \end{bmatrix} = \begin{bmatrix} v_x + t_x \\ v_y + t_y \\ v_z + t_z \\ 1 \end{bmatrix}$$

3Dベクトルに「 1」の成分を追加されたベクトルは「 同次座標 ベクトル」という。成分を増やしても、次元数は同じである。

3D?? ? ???????

普通の 3Dベクトルを同次座標に変換するのは、1つの成分を追加し、「 1」にするだけ:

$$\begin{bmatrix} v_x \\ v_y \\ v_z \end{bmatrix} \Rightarrow \begin{bmatrix} v_x \\ v_y \\ v_z \\ 1 \end{bmatrix}$$

???? ? 3D?????

[illegible] 1
[illegible]

$$\begin{bmatrix} v'_x \\ v'_y \\ v'_z \\ w \end{bmatrix} \Rightarrow \begin{bmatrix} v'_x/w \\ v'_y/w \\ v'_z/w \\ 1 \end{bmatrix} \Rightarrow \begin{bmatrix} v_x \\ v_y \\ v_z \end{bmatrix}$$

???????

[illegible] 4x4 [illegible]

<table>
<tr><td colspan="3">[illegible]</td><td colspan="3">[illegible]</td></tr>
<tr><td colspan="3">$$\begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$</td><td colspan="3">$$\begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$</td></tr>
<tr><td colspan="2">[illegible] X[illegible]</td><td colspan="2">[illegible] Y[illegible]</td><td colspan="2">[illegible] Z[illegible]</td></tr>
</table>

$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\theta) & \sin(\theta) & 0 \\ 0 & -\sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$	$\begin{bmatrix} \cos(\theta) & 0 & -\sin(\theta) & 0 \\ 0 & 1 & 0 & 0 \\ \sin(\theta) & 0 & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$	$\begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 & 0 \\ \sin(\theta) & \cos(\theta) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$

??

Matrix を編集し、 Vector4との乗算できるようにする

MyTransform を編集し:

- 頂点の配列を同次座標に変換
- 移動、拡大縮小、回転を同次座標に対応できるようにする
- 変換した後に同次座標の頂点をまた 3Dベクトルに戻す

Matrix の Multiply(Vector4) メソッド

```
// 行列×ベクトルの掛け算
public Vector4 Multiply(Vector4 vector)
{
    // ベクトルを4x1の行列にする
    Matrix b = new Matrix(4, 1);
    b.Set(0, 0, vector.x);
    b.Set(1, 0, vector.y);
    b.Set(2, 0, vector.z);
    b.Set(3, 0, vector.w);

    // 普通の掛け算
    Matrix c = Multiply(b);

    // 結果をまたベクトルにする
    Vector4 result = new Vector4();
    result.x = c.Get(0, 0);
    result.y = c.Get(1, 0);
    result.z = c.Get(2, 0);
    result.w = c.Get(3, 0);

    return result;
```

```
}
```

同次座標を使用する **MyTransform**

```
// 自分で作った変換(Transform) コンポーネント
public class MyTransform : MonoBehaviour
{
    // 位置・移動量
    [SerializeField]
    private Vector3 position;

    // 拡大縮小
    [SerializeField]
    private Vector3 scale = new Vector3(1, 1, 1);

    // 回転
    [SerializeField]
    private Vector3 rotation;

    // 立方体
    private MyCube cube;

    private void Start()
    {
        // MyCubeを取得し、変換する前の頂点を保管しておく
        cube = GetComponent<MyCube>();
    }

    private void Update()
    {
        // 頂点を取得
        Vector3 [] vertices = cube.GetVertices();

        // 同次座標の頂点
        Vector4[] v4 = Vector3ToVector4(vertices);
```

```
        // 回転
        RotateEuler(v4);

        // 拡大縮小
        Scale(v4);

        // 移動を適用
        Translate(v4);

        // 同次座標からまた3次元へ
        vertices = Vector4ToVector3(v4);

        // 変換した頂点を適用
        cube.SetVertices(vertices);
    }

    // 3Dベクトルから4D同次座標へ
    private Vector4[] Vector3ToVector4(Vector3[] vertices)
    {
        Vector4[] convert = new Vector4[vertices.Length];
        for (int i = 0; i < convert.Length; i++)
        {
            Vector3 v3 = vertices[i];
            convert[i] = new Vector4(v3.x, v3.y, v3.z, 1);
        }

        return convert;
    }

    // 4D同次座標から3Dベクトルへ
    private Vector3[] Vector4ToVector3(Vector4[] vertices)
    {
        Vector3[] convert = new Vector3[vertices.Length];
        for (int i = 0; i < convert.Length; i++)
        {
            Vector4 v4 = vertices[i];
            convert[i] = new Vector3(v4.x, v4.y, v4.z) / v4.w;
```

```
        }

        return convert;
    }

    // X軸周りの回転行列を作成
    private Matrix MakeRotX()
    {
        // 度→ラジアン
        float radian = Mathf.Deg2Rad * rotation.x;

        // サインとコサインを計算
        float c = Mathf.Cos(radian);
        float s = Mathf.Sin(radian);

        // 行列を構築
        Matrix m = new Matrix(4, 4);
        m.Set(0, 0, 1);
        m.Set(0, 1, 0);
        m.Set(0, 2, 0);
        m.Set(0, 3, 0);

        m.Set(1, 0, 0);
        m.Set(1, 1, c);
        m.Set(1, 2, s);
        m.Set(1, 3, 0);

        m.Set(2, 0, 0);
        m.Set(2, 1, -s);
        m.Set(2, 2, c);
        m.Set(2, 3, 0);

        m.Set(3, 0, 0);
        m.Set(3, 1, 0);
        m.Set(3, 2, 0);
        m.Set(3, 3, 1);

        return m;
```

```
}

// Y軸周りの回転行列を作成
private Matrix MakeRotY()
{
    // 度→ラジアン
    float radian = Mathf.Deg2Rad * rotation.y;

    // サインとコサインを計算
    float c = Mathf.Cos(radian);
    float s = Mathf.Sin(radian);

    // 行列を構築
    Matrix m = new Matrix(4, 4);
    m.Set(0, 0, c);
    m.Set(0, 1, 0);
    m.Set(0, 2, -s);
    m.Set(0, 3, 0);

    m.Set(1, 0, 0);
    m.Set(1, 1, 1);
    m.Set(1, 2, 0);
    m.Set(1, 3, 0);

    m.Set(2, 0, s);
    m.Set(2, 1, 0);
    m.Set(2, 2, c);
    m.Set(2, 3, 0);

    m.Set(3, 0, 0);
    m.Set(3, 1, 0);
    m.Set(3, 2, 0);
    m.Set(3, 3, 1);

    return m;
}

// Z軸周りの回転行列を作成
```

```
private Matrix MakeRotZ()
{
    // 度→ラジアン
    float radian = Mathf.Deg2Rad * rotation.z;

    // サインとコサインを計算
    float c = Mathf.Cos(radian);
    float s = Mathf.Sin(radian);

    // 行列を構築
    Matrix m = new Matrix(4, 4);
    m.Set(0, 0, c);
    m.Set(0, 1, -s);
    m.Set(0, 2, 0);
    m.Set(0, 3, 0);

    m.Set(1, 0, s);
    m.Set(1, 1, c);
    m.Set(1, 2, 0);
    m.Set(1, 3, 0);

    m.Set(2, 0, 0);
    m.Set(2, 1, 0);
    m.Set(2, 2, 1);
    m.Set(2, 3, 0);

    m.Set(3, 0, 0);
    m.Set(3, 1, 0);
    m.Set(3, 2, 0);
    m.Set(3, 3, 1);

    return m;
}

private void RotateEuler(Vector4[] vertices)
{
    Matrix rotX = MakeRotX();
    Matrix rotY = MakeRotY();
```

```
    Matrix rotZ = MakeRotZ();

    // 1個ずつを処理
    for (int i = 0; i < vertices.Length; i++)
    {
        // X→Y→Zの順番で回転
        Vector4 v = vertices[i];
        v = rotX.Multiply(v);
        v = rotY.Multiply(v);
        v = rotZ.Multiply(v);
        vertices[i] = v;
    }
}

// 拡大縮小
private void Scale(Vector4[] vertices)
{
    // スケール行列を作成
    Matrix s = new Matrix(4, 4);
    s.Set(0, 0, scale.x);
    s.Set(1, 1, scale.y);
    s.Set(2, 2, scale.z);
    s.Set(3, 3, 1);

    // 1個ずつを処理
    for (int i = 0; i < vertices.Length; i++)
    {
        // ベクトルの乗算で拡大縮小を実現!
        vertices[i] = s.Multiply(vertices[i]);
    }
}

// 頂点を移動する
private void Translate(Vector4[] vertices)
{
    // 移動行列を作成
    Matrix t = new Matrix(4, 4);
    t.Set(0, 0, 1);
```

```
        t.Set(1, 1, 1);
        t.Set(2, 2, 1);

        t.Set(0, 3, position.x);
        t.Set(1, 3, position.y);
        t.Set(2, 3, position.z);
        t.Set(3, 3, 1);

        // 1個ずつを処理
        for (int i = 0; i < vertices.Length; i++)
        {
            vertices[i] = t.Multiply(vertices[i]);
        }
    }
}
```

????? ? ????????

移動、スケール、回転それぞれは別々に実現できたが、同時に様々な変換できない状態になっている。あくまで1つの変換だけ。

でも、よく考えてみると、スケールと回転の場合は、行列とベクトルの乗算で変換できているので、連続に乗算を続けていけば、変換の連結ができる! つまり:

$$[\text{最後の位置}] = [\text{変換}1] \times [\text{変換}2] \times [\text{変換}3] \times \cdots [\text{変換}n] \times [\text{元の位置}]$$

これは「Transform」コンポーネントをやっている仕事! 各種の変換（Position, Scale, Rotate）のすべてを一つの行列にまとめて、最後に元の頂点にかけて、各頂点の新しい位置を求める。

??

よく見ると、MyTransform では、変換ごとに for 文で頂点変換されている:

```
// 移動、回転、拡大縮小の変換では、以下の処理がある:
for (int i = 0; i < vertices.Length; i++)
{
    vertices[i] = （なにかの変換行列）.Multiply(vertices[i]);
```

```
}
```

[illegible] MyTransform [illegible]

- [illegible] Matrix [illegible]
- Update[illegible]
  - [illegible] X × [illegible] Y × [illegible] Z × [illegible] × [illegible]
- [illegible] [illegible] [illegible] for[illegible]

[illegible] MyTransform

```
using System;
using UnityEngine;
using UnityEngine.UIElements;

// [illegible](Transform[illegible]
public class MyTransform : MonoBehaviour
{
    // [illegible]
    [SerializeField]
    private Vector3 position;

    // [illegible]
    [SerializeField]
    private Vector3 scale = new Vector3(1, 1, 1);

    // [illegible]
    [SerializeField]
    private Vector3 rotation;

    // [illegible]
    private MyCube cube;

    private void Start()
    {
        // MyCube[illegible]
        cube = GetComponent<MyCube>();
    }

    private void Update()
```

```
    {
        // 頂点を取得
        Vector3 [] vertices = cube.GetVertices();

        // 同次座標の頂点
        Vector4[] v4 = Vector3ToVector4(vertices);

        // 回転
        Matrix rotX = MakeRotX();
        Matrix rotY = MakeRotY();
        Matrix rotZ = MakeRotZ();

        // 拡大縮小
        Matrix scale = MakeScale();

        // 移動
        Matrix translate = MakeTranslate();

        // すべて連結する
        Matrix final =
rotX.Multiply(rotY).Multiply(rotZ).Multiply(scale).Multiply(translate);

        // 変換
        for (int i = 0; i < v4.Length; i++)
            v4[i] = final.Multiply(v4[i]);

        // 同次座標からまた3次元へ
        vertices = Vector4ToVector3(v4);

        // 変換した頂点を適用
        cube.SetVertices(vertices);
    }

    // 3Dベクトルから4D同次座標へ
    private Vector4[] Vector3ToVector4(Vector3[] vertices)
    {
        Vector4[] convert = new Vector4[vertices.Length];
        for (int i = 0; i < convert.Length; i++)
        {
```

```
        Vector3 v3 = vertices[i];
        convert[i] = new Vector4(v3.x, v3.y, v3.z, 1);
    }

    return convert;
}

// 4D同次座標から3Dベクトルへ
private Vector3[] Vector4ToVector3(Vector4[] vertices)
{
    Vector3[] convert = new Vector3[vertices.Length];
    for (int i = 0; i < convert.Length; i++)
    {
        Vector4 v4 = vertices[i];
        convert[i] = new Vector3(v4.x, v4.y, v4.z) / v4.w;
    }

    return convert;
}

// X軸周りの回転行列を作成
private Matrix MakeRotX()
{
    // 度→ラジアン
    float radian = Mathf.Deg2Rad * rotation.x;

    // サインとコサインを計算
    float c = Mathf.Cos(radian);
    float s = Mathf.Sin(radian);

    // 行列を構築
    Matrix m = new Matrix(4, 4);
    m.Set(0, 0, 1);
    m.Set(0, 1, 0);
    m.Set(0, 2, 0);
    m.Set(0, 3, 0);

    m.Set(1, 0, 0);
    m.Set(1, 1, c);
```

```
        m.Set(1, 2, s);
        m.Set(1, 3, 0);

        m.Set(2, 0, 0);
        m.Set(2, 1, -s);
        m.Set(2, 2, c);
        m.Set(2, 3, 0);

        m.Set(3, 0, 0);
        m.Set(3, 1, 0);
        m.Set(3, 2, 0);
        m.Set(3, 3, 1);

        return m;
    }

    // Y軸周りの回転行列を作成
    private Matrix MakeRotY()
    {
        // 度→ラジアン
        float radian = Mathf.Deg2Rad * rotation.y;

        // サインとコサインを計算
        float c = Mathf.Cos(radian);
        float s = Mathf.Sin(radian);

        // 行列を構築
        Matrix m = new Matrix(4, 4);
        m.Set(0, 0, c);
        m.Set(0, 1, 0);
        m.Set(0, 2, -s);
        m.Set(0, 3, 0);

        m.Set(1, 0, 0);
        m.Set(1, 1, 1);
        m.Set(1, 2, 0);
        m.Set(1, 3, 0);

        m.Set(2, 0, s);
```

```
        m.Set(2, 1, 0);
        m.Set(2, 2, c);
        m.Set(2, 3, 0);

        m.Set(3, 0, 0);
        m.Set(3, 1, 0);
        m.Set(3, 2, 0);
        m.Set(3, 3, 1);

        return m;
    }

    // Z軸周りの回転行列を作成
    private Matrix MakeRotZ()
    {
        // 度→ラジアン
        float radian = Mathf.Deg2Rad * rotation.z;

        // サインとコサインを計算
        float c = Mathf.Cos(radian);
        float s = Mathf.Sin(radian);

        // 行列を構築
        Matrix m = new Matrix(4, 4);
        m.Set(0, 0, c);
        m.Set(0, 1, -s);
        m.Set(0, 2, 0);
        m.Set(0, 3, 0);

        m.Set(1, 0, s);
        m.Set(1, 1, c);
        m.Set(1, 2, 0);
        m.Set(1, 3, 0);

        m.Set(2, 0, 0);
        m.Set(2, 1, 0);
        m.Set(2, 2, 1);
        m.Set(2, 3, 0);
```

```
        m.Set(3, 0, 0);
        m.Set(3, 1, 0);
        m.Set(3, 2, 0);
        m.Set(3, 3, 1);

        return m;
    }

    // 拡大縮小
    private Matrix MakeScale()
    {
        // スケール行列を作成
        Matrix s = new Matrix(4, 4);
        s.Set(0, 0, scale.x);
        s.Set(1, 1, scale.y);
        s.Set(2, 2, scale.z);
        s.Set(3, 3, 1);

        return s;
    }

    // 移動変換行列を返す
    private Matrix MakeTranslate()
    {
        // 移動行列を作成
        Matrix t = new Matrix(4, 4);
        t.Set(0, 0, 1);
        t.Set(1, 1, 1);
        t.Set(2, 2, 1);

        t.Set(0, 3, position.x);
        t.Set(1, 3, position.y);
        t.Set(2, 3, position.z);
        t.Set(3, 3, 1);

        return t;
    }
}
```

Revision #9
Created 2026-06-17 01:30:08 UTC by Menendez Francisco
Updated 2026-06-23 05:47:28 UTC by Menendez Francisco