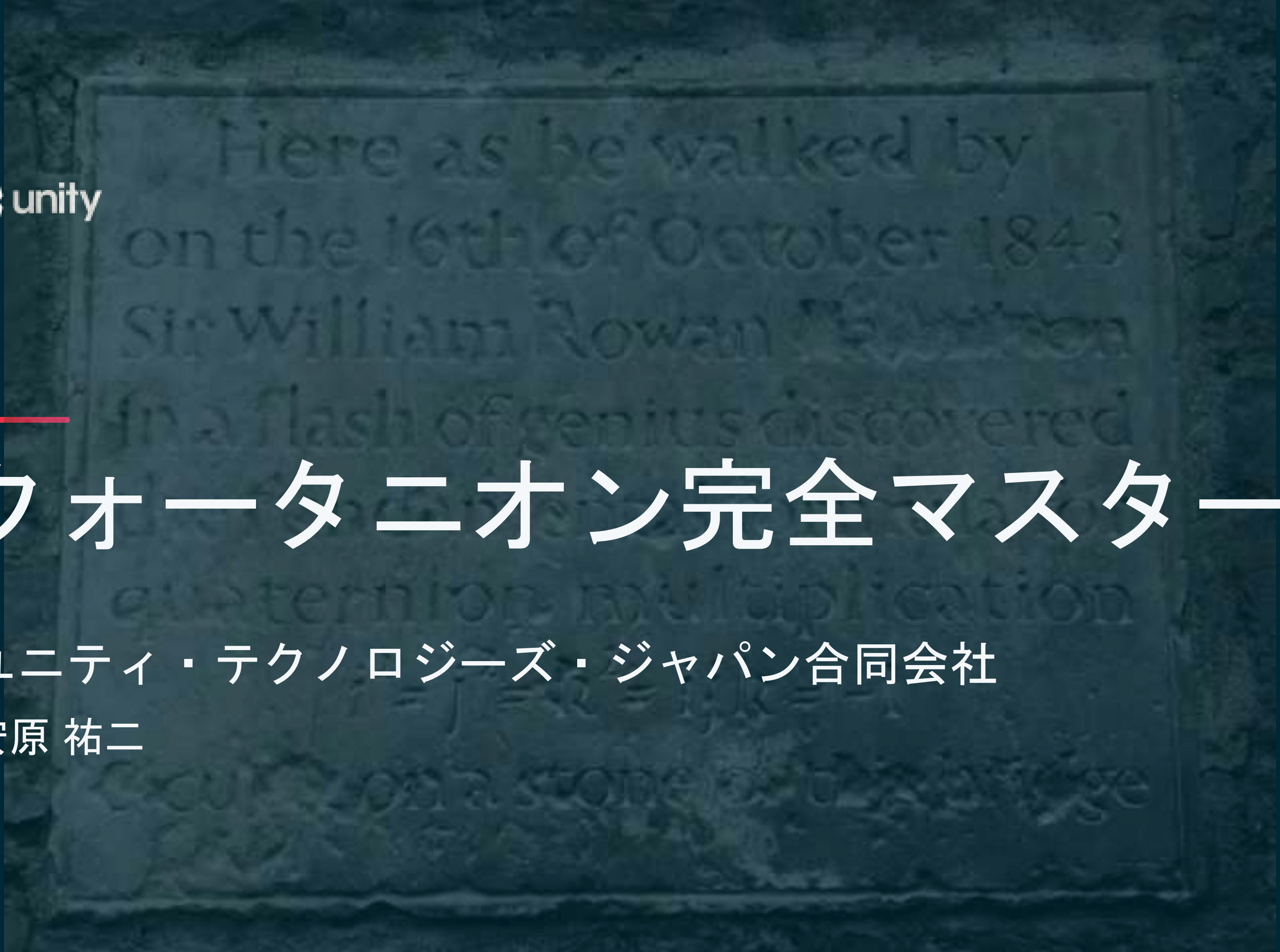


unity

クォータニオン完全マスター

ユニティ・テクノロジーズ・ジャパン合同会社
安原 祐二

Quaternion

struct in UnityEngine

Description

クォータニオンは回転を表すのに使用されます。

コンパクトであり、ジンバルロックの問題がなく、簡単に補間できます。 Unity はすべての回転を表現するのにクォータニオンを内部的に使用します。

しかし複雑な数字にもとづいており、直感的でない側面があります。 このため個別のクォータニオン成分 (x, y, z, w) をアクセスまたは修正することは殆どありません。 主に既存の回転（例. Transform など）にもとづき新規の回転を作成します (例えば、2 つの回転の間をスムーズに補間)。 使用する Quaternion 関数の 99% は、Quaternion.LookRotation、Quaternion.Angle、Quaternion.Euler、Quaternion.Slerp、Quaternion.FromToRotation、Quaternion.identity（他の関数はきわめて稀な用途）

Unity スクリプトリファレンスより抜粋

本セッションの旅程

クォータニオン

複素数

CG

オイラー角

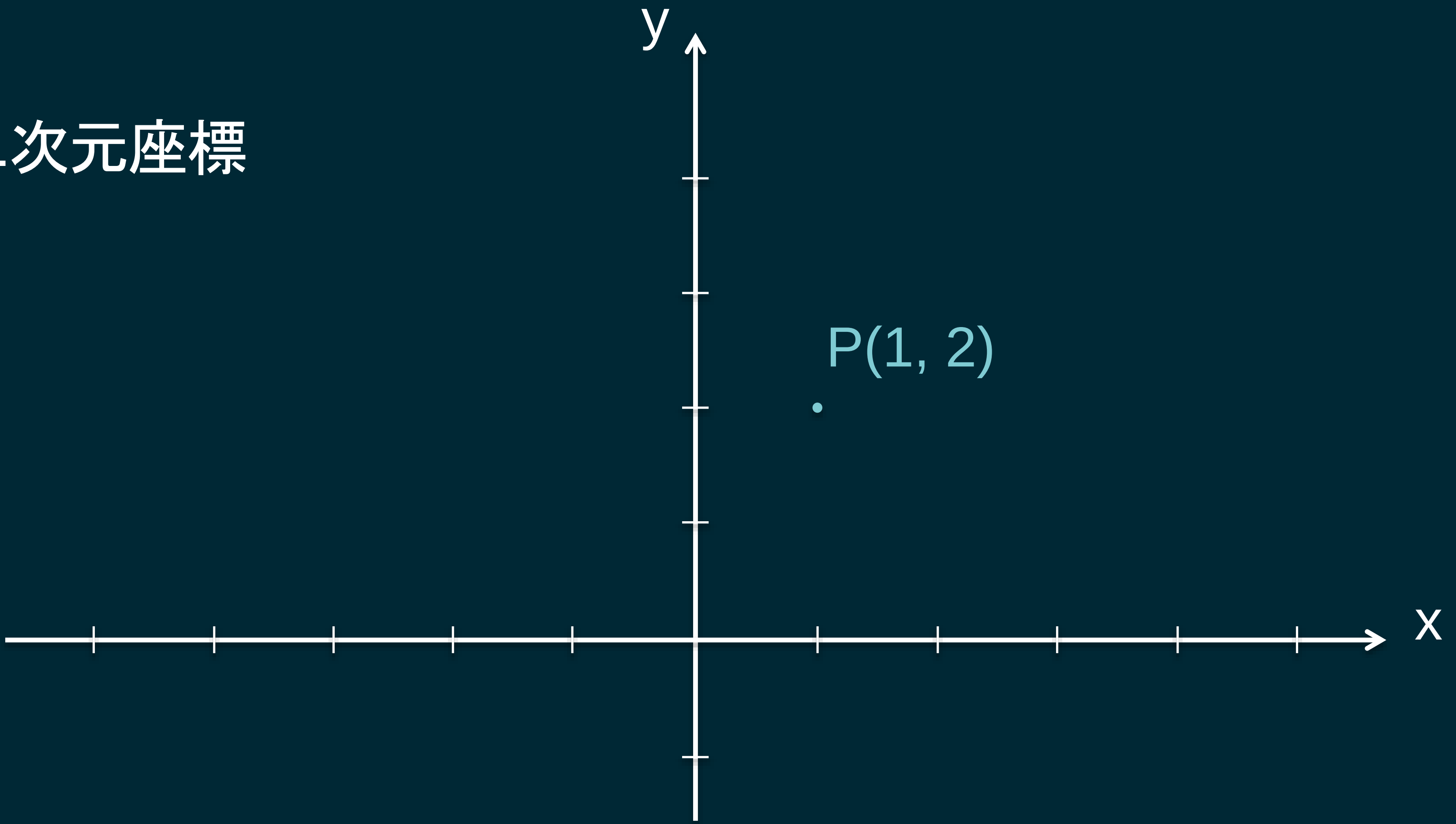
座標変換

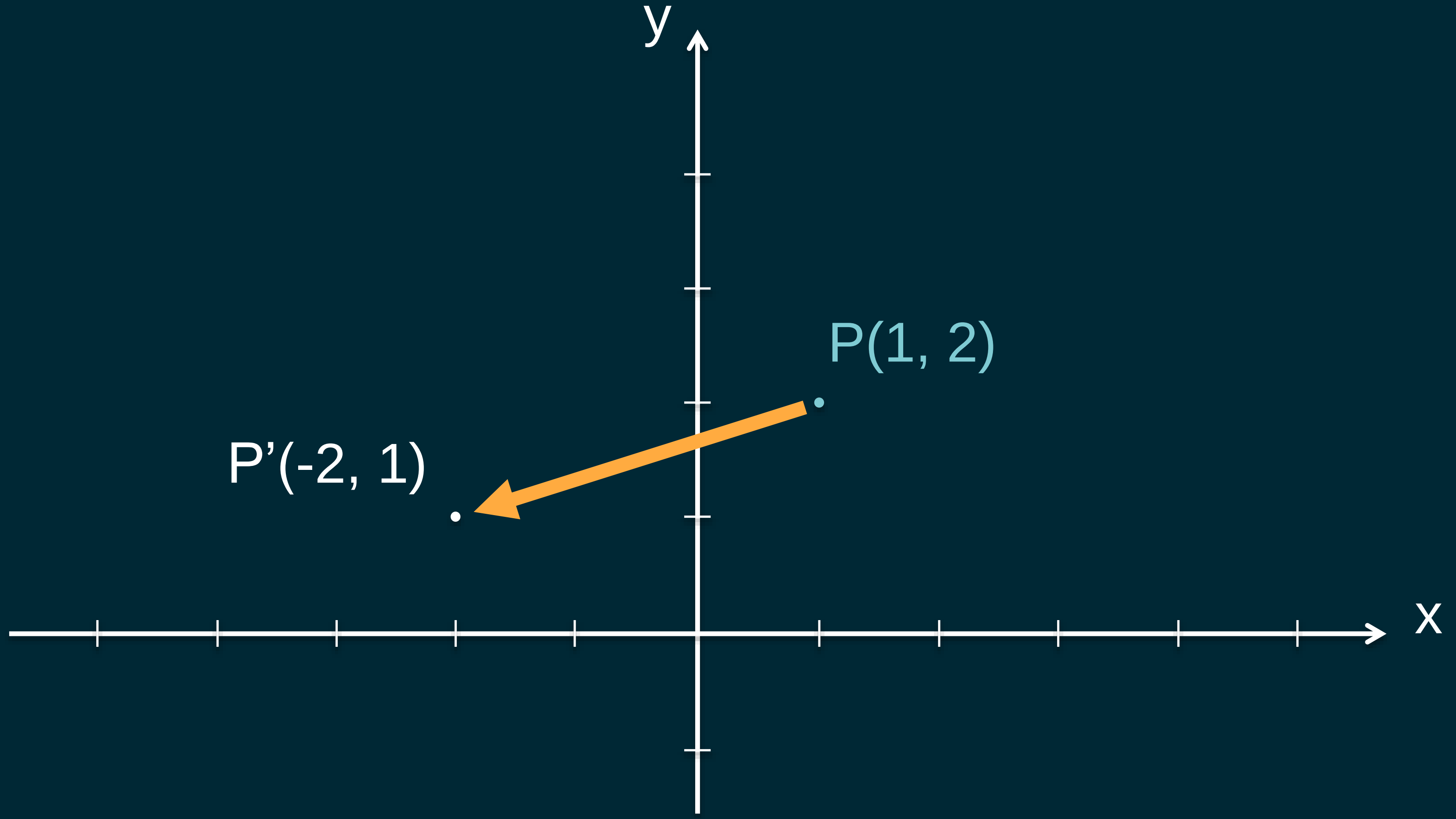
座標変換

CGの
きほん

たくさんの点を動かす！

二次元座標





y
x
P(1, 2)
P’(-2, 1)

ある変換で点を移動

$P(1, 2) \xrightarrow[\text{変換}]{} P'(-2, 1)$

ある変換＝なんらかのルール

同じ変換でたくさんの点を移動

$$P(1, 2) \xrightarrow[\text{変換}]{} P'(-2, 1)$$

$$Q(2, 0) \xrightarrow[\text{変換}]{} Q'(0, 2)$$

$$R(1, 1) \xrightarrow[\text{変換}]{} R'(-1, 1)$$

こういう式にしてみよう

$$P'(x', y') = [変換]P(x, y)$$

↓

$$x' = ax + by$$

$$y' = cx + dy$$

色々できそう！

$$x' = ax + by$$

$$y' = cx + dy$$

a=1　b=0　c=0　d=1 なら

$$x' = 1x + 0y$$

$$y' = 0x + 1y$$

$$x' = x$$

$$y' = y$$

変化しない変換

色々できそう！

$$x' = ax + by$$

$$y' = cx + dy$$

a=1 b=0 c=0 d=1 なら

$$x' = 1x + 0y$$

$$y' = 0x + 1y$$

$$x' = x$$

$$y' = y$$

変化しない変換

a=0 b=1 c=1 d=0 なら

$$x' = 0x + 1y$$

$$y' = 1x + 0y$$

$$x' = y$$

$$y' = x$$

xy入れ替え変換

abcdをまとめてしまおう

$$x' = ax + by$$

$$y' = cx + dy$$

$$\downarrow$$

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix}$$

これを行列と呼ぶ！

変換を行列Mとして

$$P(1, 2) \xrightarrow[\text{変換}]{} P'(-2, 1)$$

$$P'(x', y') = [\text{変換}]P(x, y)$$

$$P' = MP$$

変換を行列Mとして

$$P(1, 2) \xrightarrow[\text{変換}]{} P'(-2, 1)$$

$$P'(x', y') = [\text{変換}]P(x, y)$$

$$P' = MP$$

$$P' = \begin{pmatrix} a & b \\ c & d \end{pmatrix} P$$

行列に点ベクトルを掛ける

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} \longrightarrow \begin{aligned} x' &= ax + by \\ y' &= cx + dy \end{aligned}$$

例えば

$$\begin{pmatrix} -2 \\ 1 \end{pmatrix} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} 1 \\ 2 \end{pmatrix} \longrightarrow \begin{aligned} -2 &= a1 + b2 \\ 1 &= c1 + d2 \end{aligned}$$

行列に点ベクトルを掛ける

$$\begin{pmatrix} -2 \\ 1 \end{pmatrix} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} 1 \\ 2 \end{pmatrix}$$

$$-2 = a1 + b2$$

$$1 = c1 + d2$$

$$\begin{pmatrix}-2\\1\end{pmatrix} = \begin{pmatrix}a & +\,b\\ c & +\,d\end{pmatrix}\begin{pmatrix}1\\2\end{pmatrix}$$

$$-2 = a1 + b2$$

$$1 = c1 + d2$$

三次元では3x3行列になる

$$x' = ax + by + cz$$

$$y' = dx + ey + fz$$

$$z' = gx + hy + iz$$

$$\begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix}$$

概念は同じなので
以降も二次元で続けます

いろいろな行列

$$\begin{pmatrix} 2 & 0 \\ 0 & 2 \end{pmatrix}$$

２倍に拡大

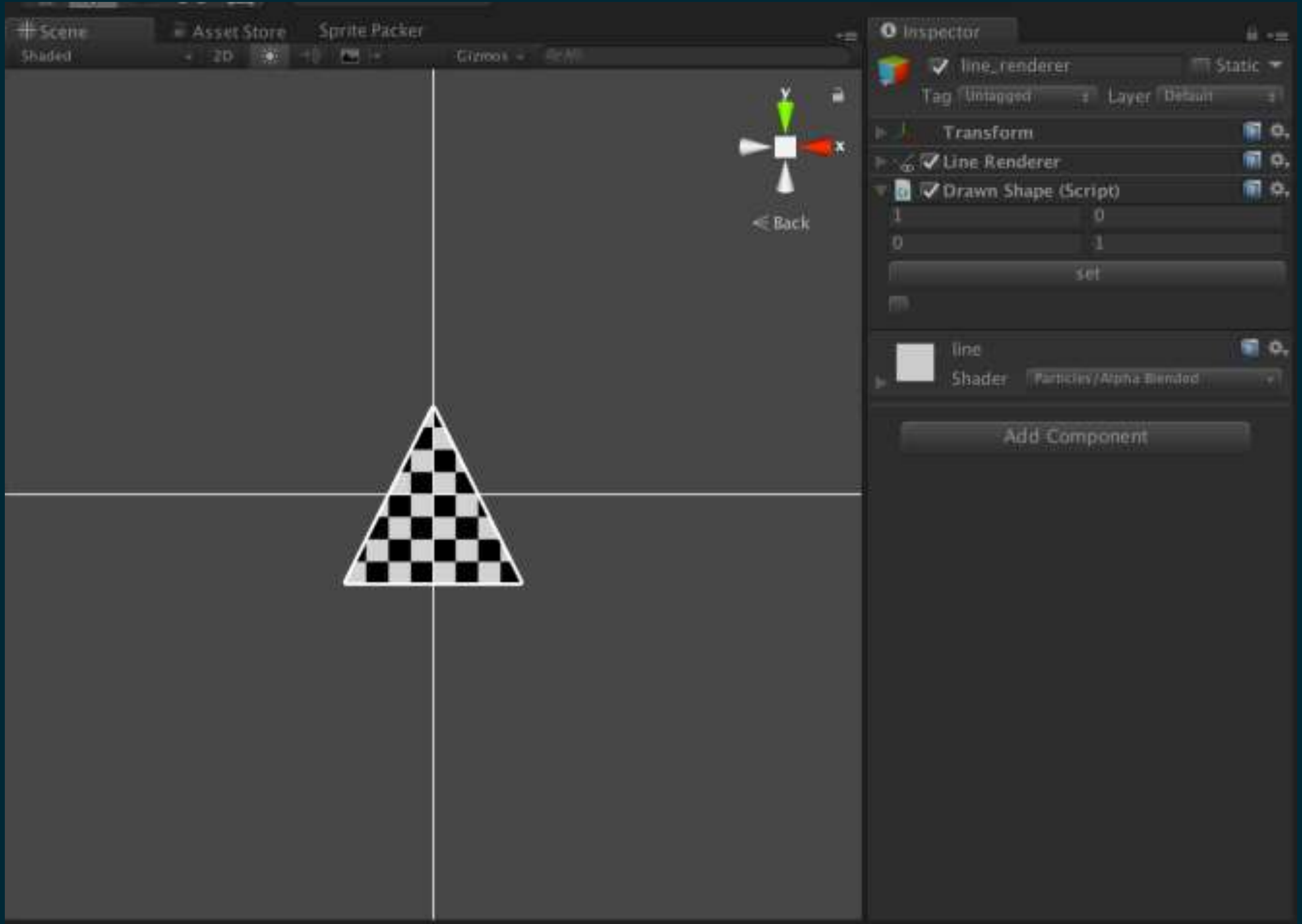
スケーリング行列

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

θ回転する

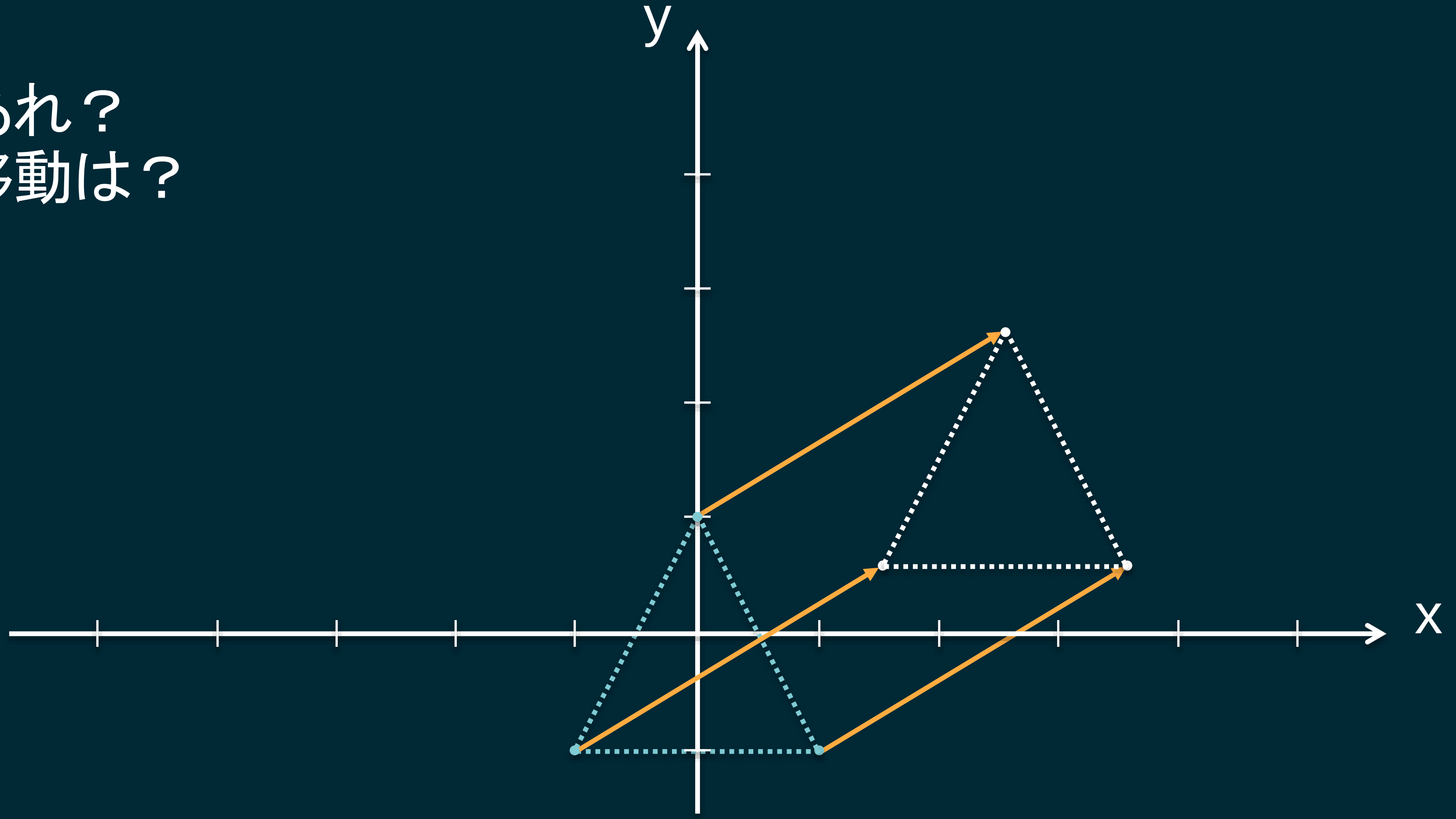
回転行列

デモ



Scene
Asset Store
Sprite Packer
Shaded
2D
Gizmos
Back
Inspector
line_renderer
Static
Tag
Untagged
Layer
Default
Transform
Line Renderer
Drawn Shape (Script)
1
0
0
1
set
line
Shader
Particles/Alpha Blended
Add Component

あれ？
移動は？



行列の値をどう工夫しても移動はできない！

移動するために式を変更

$$x' = ax + by$$

$$y' = cx + dy$$

↓

追加！

$$x' = ax + by \boxed{+ s}$$

$$y' = cx + dy \boxed{+ t}$$

これで(s,t)で移動できる

行と列の数が同じ
正方行列と呼ぶ

移動を実現するために行列を拡張

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix}$$

$$\downarrow$$

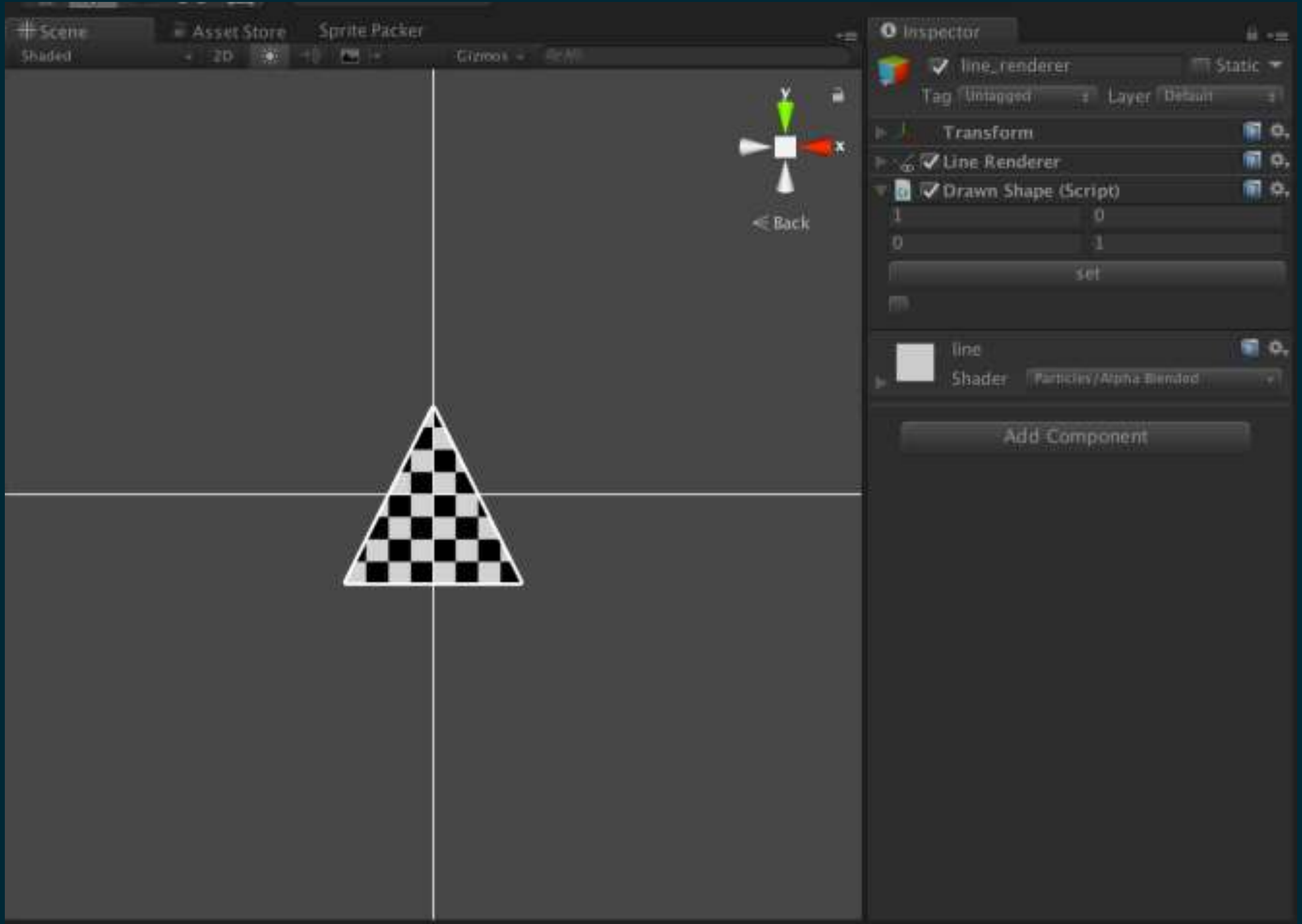
$$x' = ax + by + s$$

$$y' = cx + dy + t$$

$$\rightarrow \begin{pmatrix} a & b & s \\ c & d & t \end{pmatrix} \rightarrow \begin{pmatrix} a & b & s \\ c & d & t \\ 0 & 0 & 1 \end{pmatrix}$$

何かと都合がいいので
3x3 にする

デモ


Scene
Asset Store
Sprite Packer
Shaded
2D
Gizmos
Back
Inspector
line_renderer
Static
Tag
Untagged
Layer
Default
Transform
Line Renderer
Drawn Shape (Script)
1
0
0
1
set
line
Shader
Particles/Alpha Blended
Add Component

三次元でも同様にして移動を実現

$$\begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \begin{pmatrix} a & b & c & s \\ d & e & f & t \\ g & h & i & u \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

つまり三次元では4x4行列を使う

二次元用3x3行列の各部分の役割

回転

$$\begin{pmatrix} a & b & s \\ c & d & t \\ 0 & 0 & 1 \end{pmatrix}$$

移動

三次元用4x4行列の各部分の役割

回転

$$\begin{pmatrix} a & b & c & s \\ d & e & f & t \\ g & h & i & u \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

移動

三次元用4x4行列の各部分の役割

回転
transform.rotation
Quaternion
クォータニオン
で作る

$$\begin{pmatrix} a & b & c & s \\ d & e & f & t \\ g & h & i & u \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

移動
transform.position
Vector3
ベクトル
で作る

異なる変換をいくつも重ねる

$$P' = (\circ(\circ(\circ(\circ P))))$$

これを点ごとに計算するのはたいへん…

いくつも変換する場合は事前に計算しておける

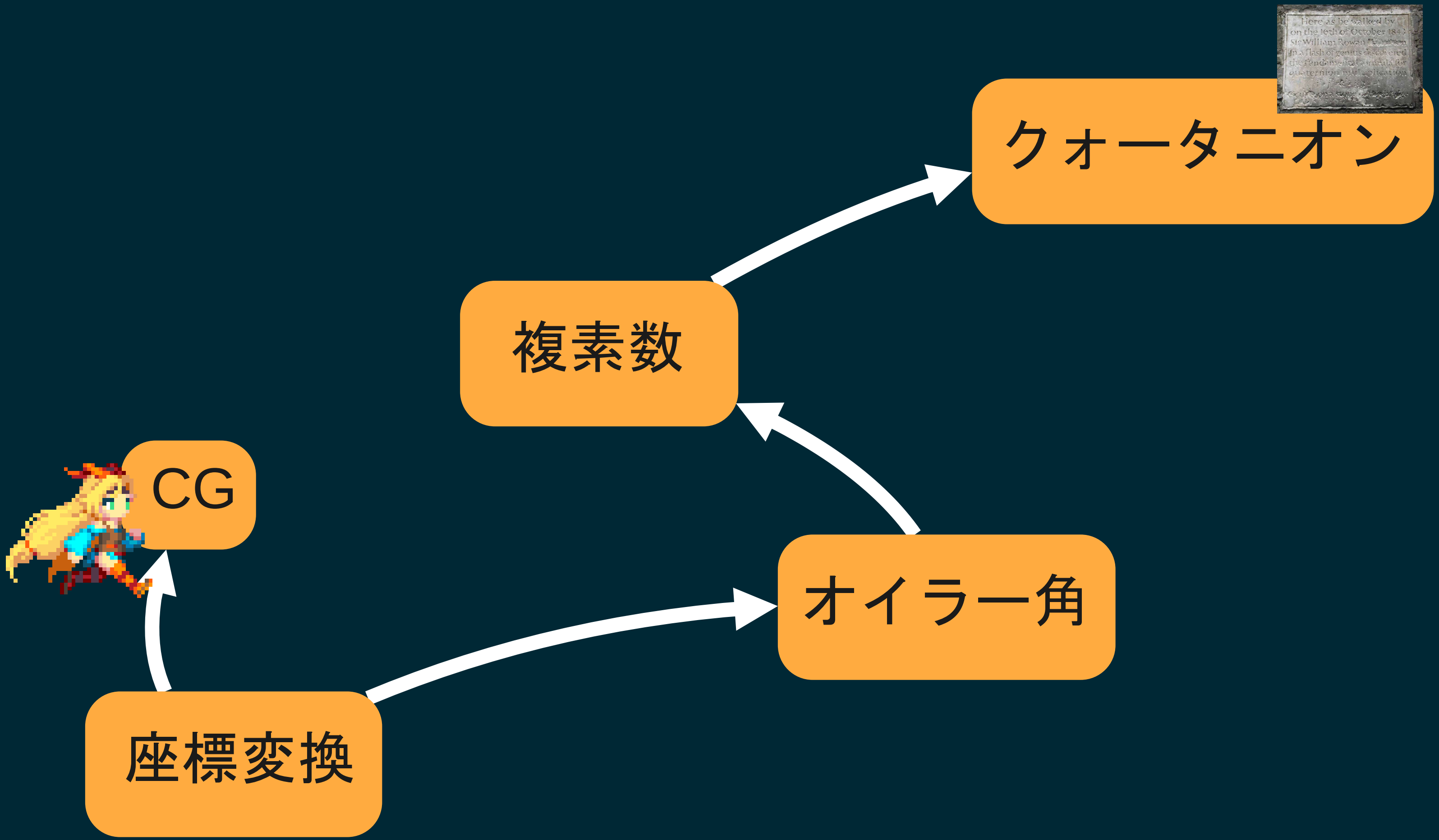
$$P' = (\circ(\circ(\circ(\circ P))))$$

$$P' = \circ\circ\circ\circ P$$

$M = \circ\circ\circ\circ$ を
事前に計算して…

$$P' = MP$$

点ごとにMを掛ければ済む

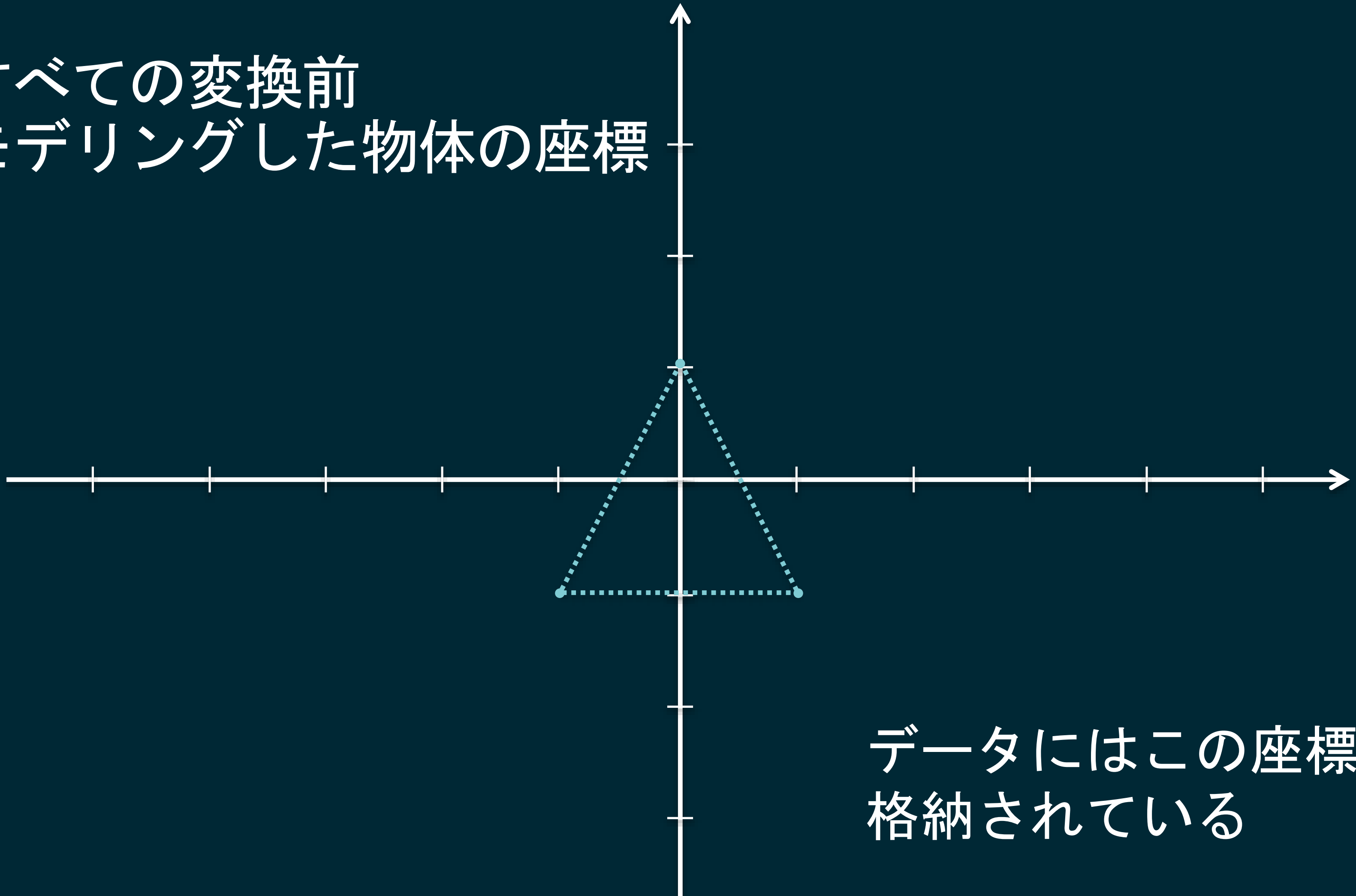

クォータニオン
複素数
CG
オイラー角
座標変換

ＣＧに応用する行列

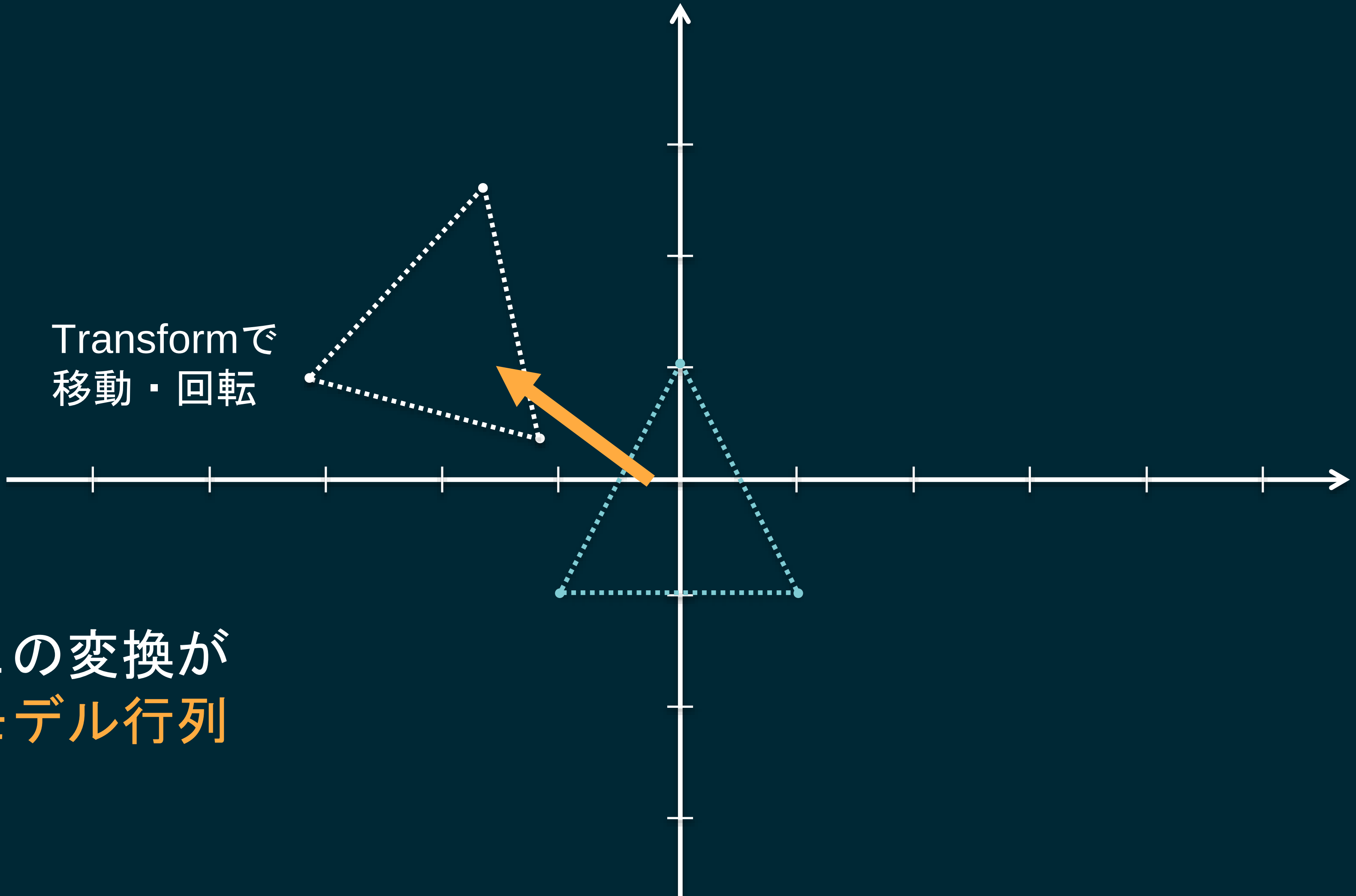
３つの変換行列

- モデル行列
- ビュー行列
- プロジェクション行列

すべての変換前
モデリングした物体の座標

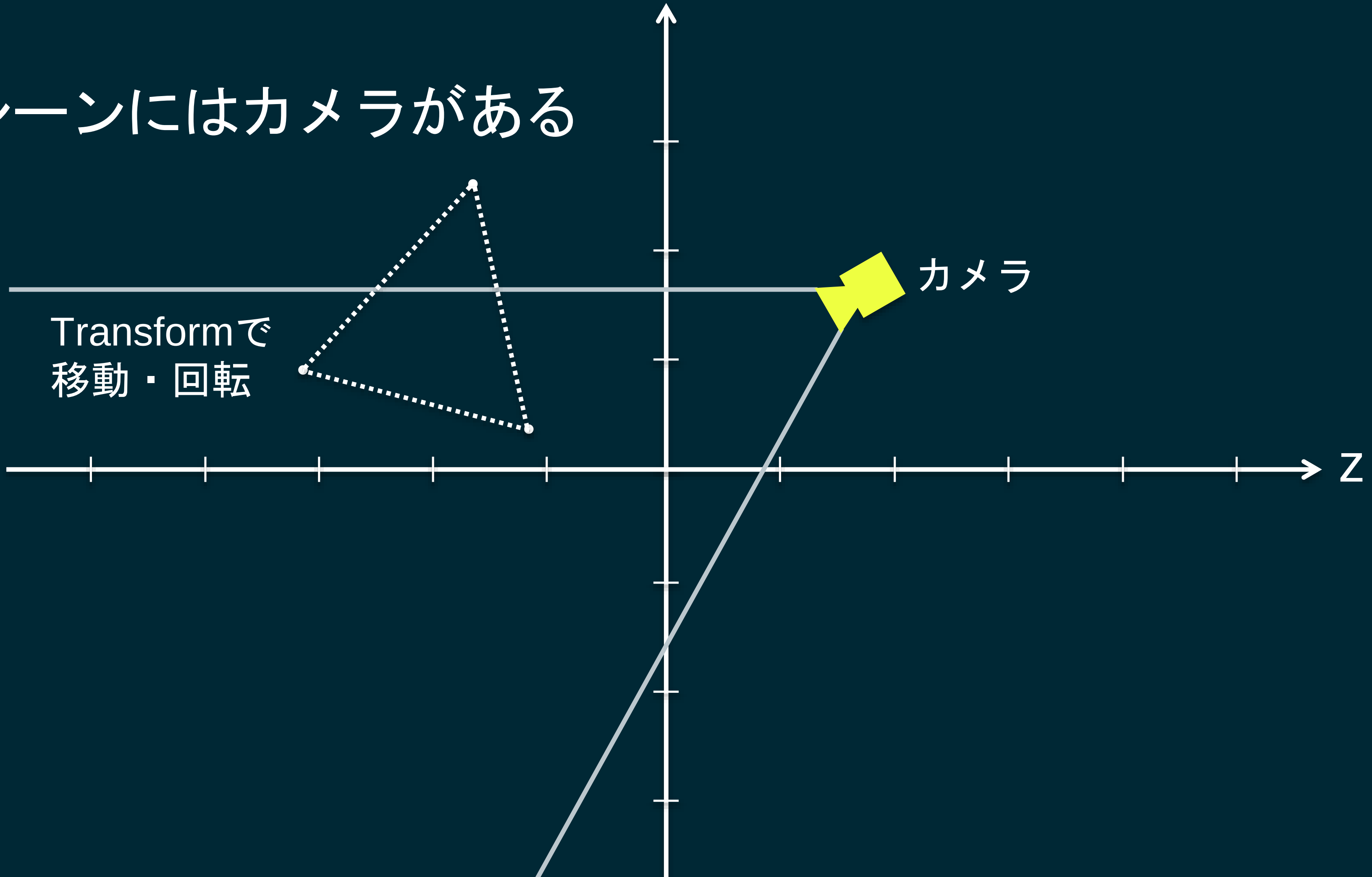


データにはこの座標が
格納されている

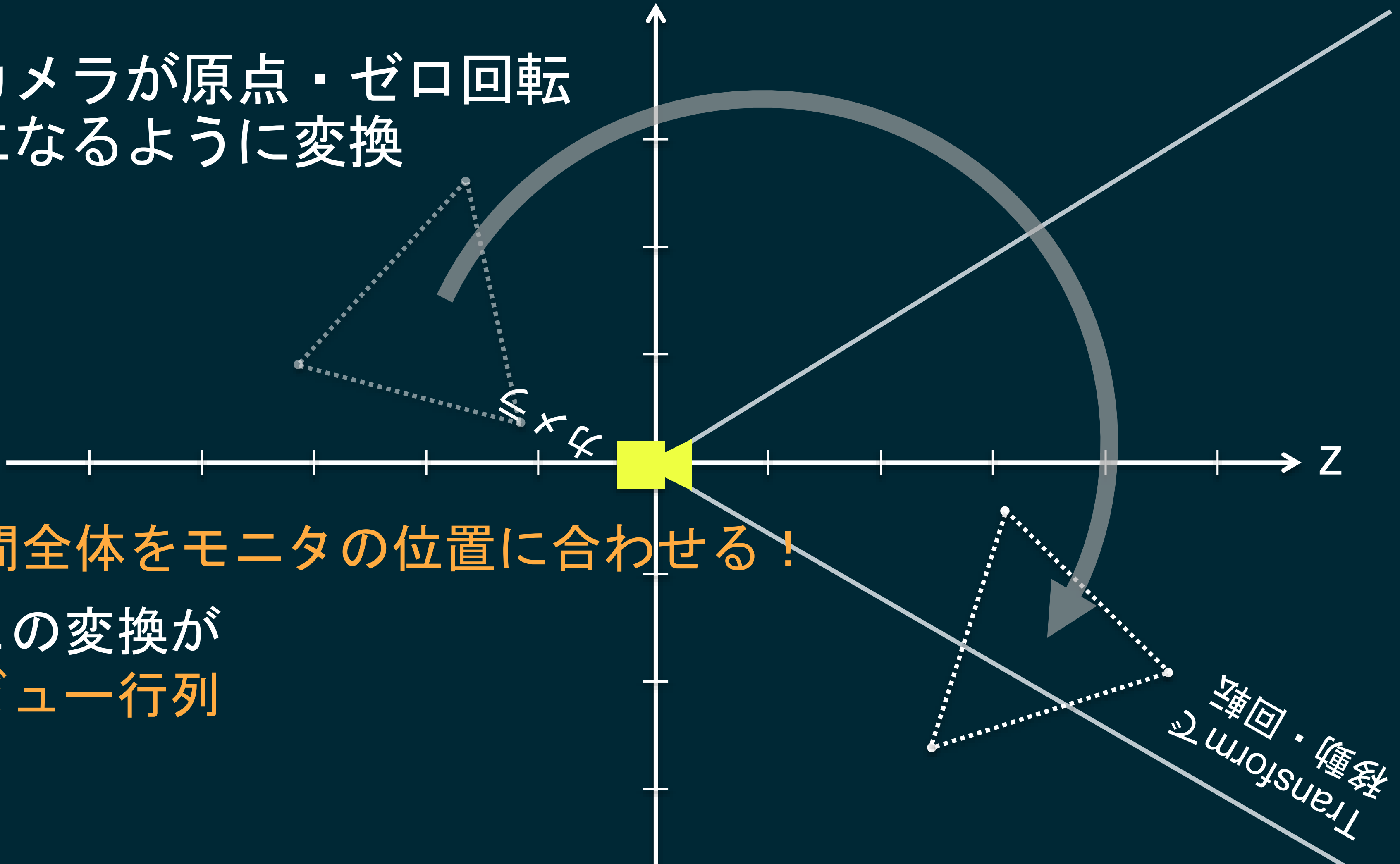


この変換が
モデル行列

シーンにはカメラがある



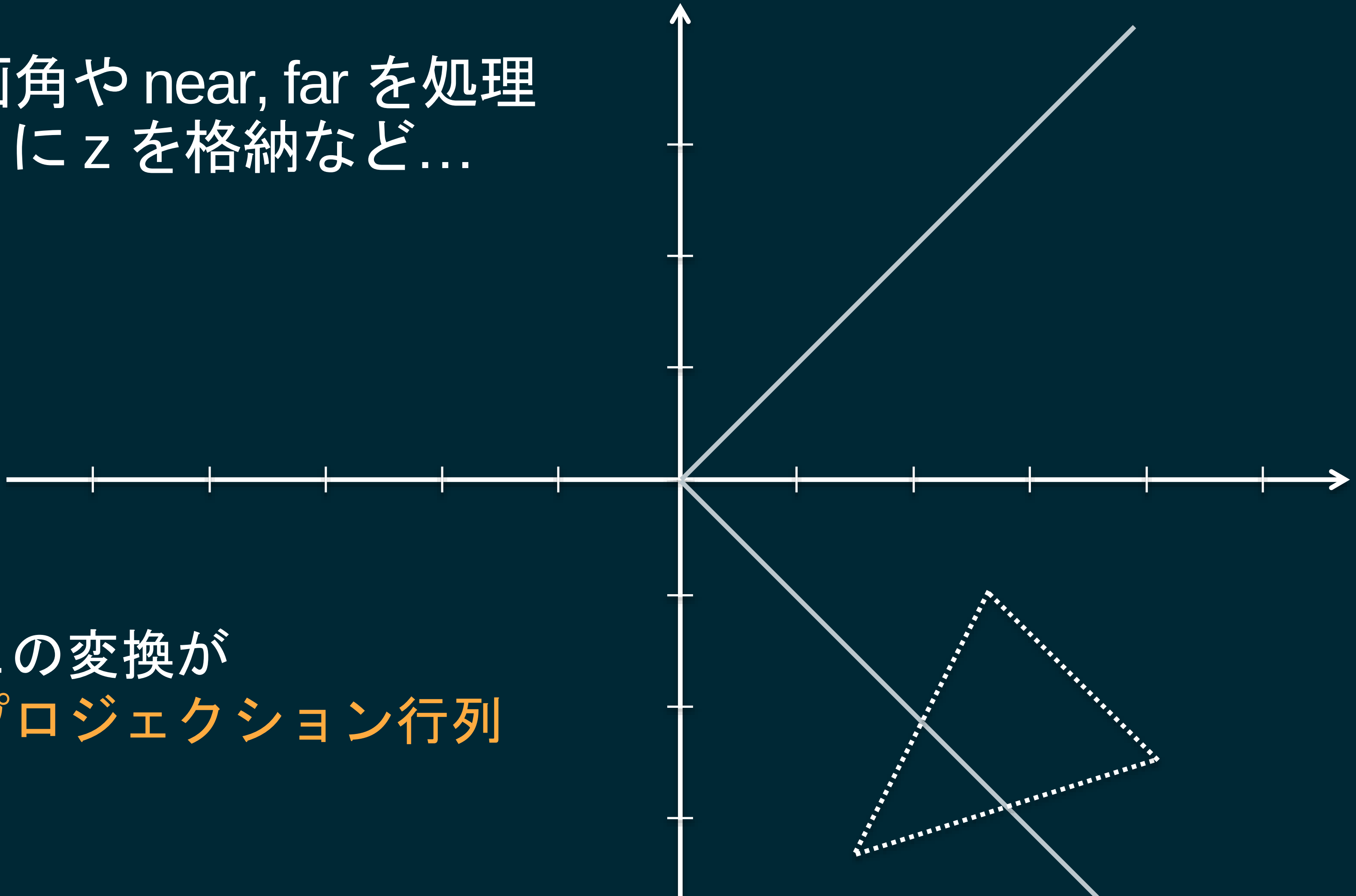
カメラが原点・ゼロ回転
になるように変換



空間全体をモニタの位置に合わせる！

この変換が
ビュー行列

画角や near, far を処理
w に z を格納など…



この変換が
プロジェクション行列

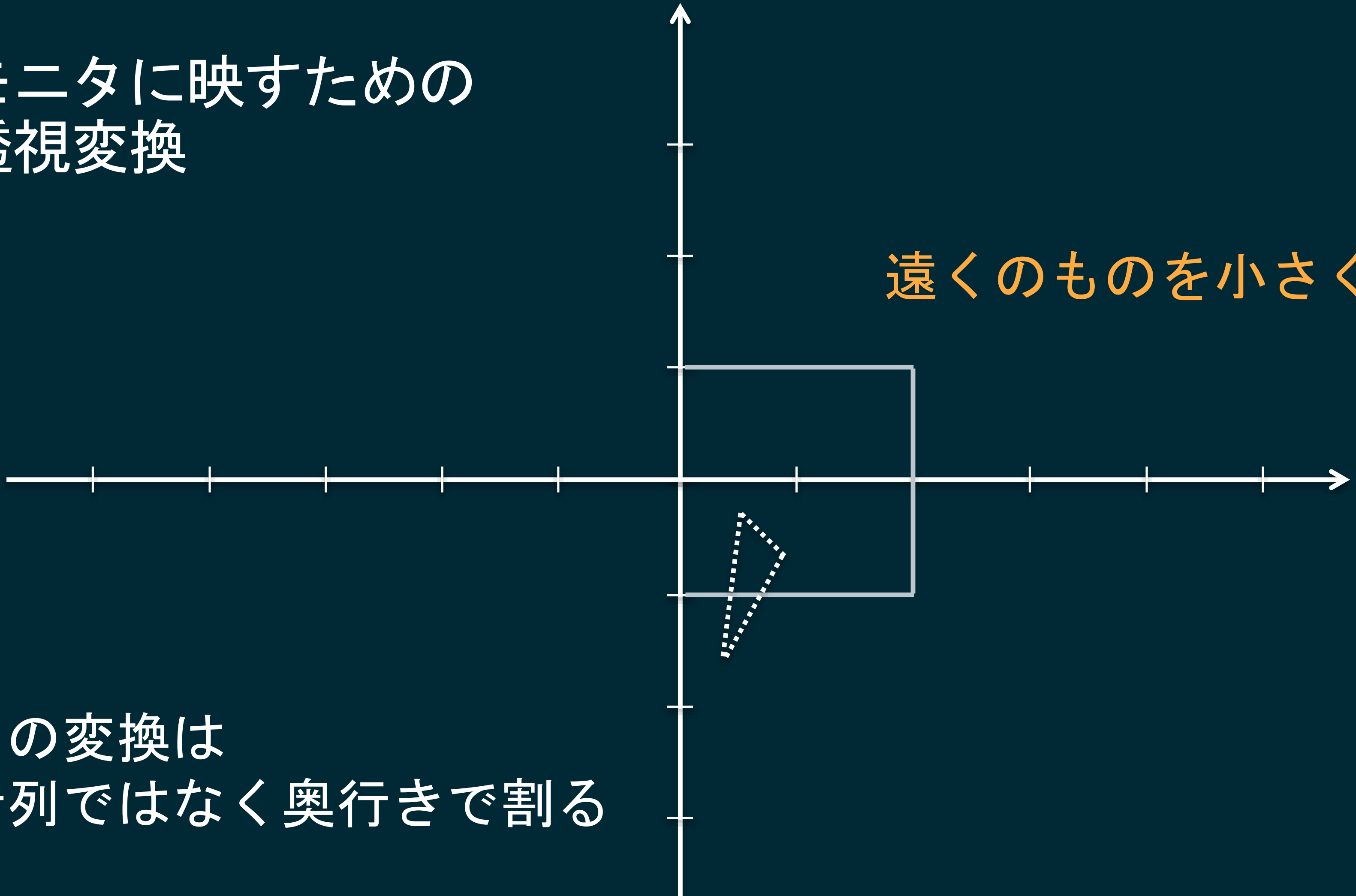
Unity組み込みシェーダ UnityShaderVariables.cginc より抜粋

```
#define UNITY_MATRIX_P glstate_matrix_projection
#define UNITY_MATRIX_V unity_MatrixV
#define UNITY_MATRIX_I_V unity_MatrixInvV
#define UNITY_MATRIX_VP unity_MatrixVP
#define UNITY_MATRIX_M unity_ObjectToWorld

#define UNITY_MATRIX_MVP mul(unity_MatrixVP, unity_ObjectToWorld)
#define UNITY_MATRIX_MV mul(unity_MatrixV, unity_ObjectToWorld)
#define UNITY_MATRIX_T_MV transpose(UNITY_MATRIX_MV)
#define UNITY_MATRIX_IT_MV transpose(mul(unity_WorldToObject, unity_MatrixInvV))
```

M:モデル　V:ビュー　P:プロジェクション

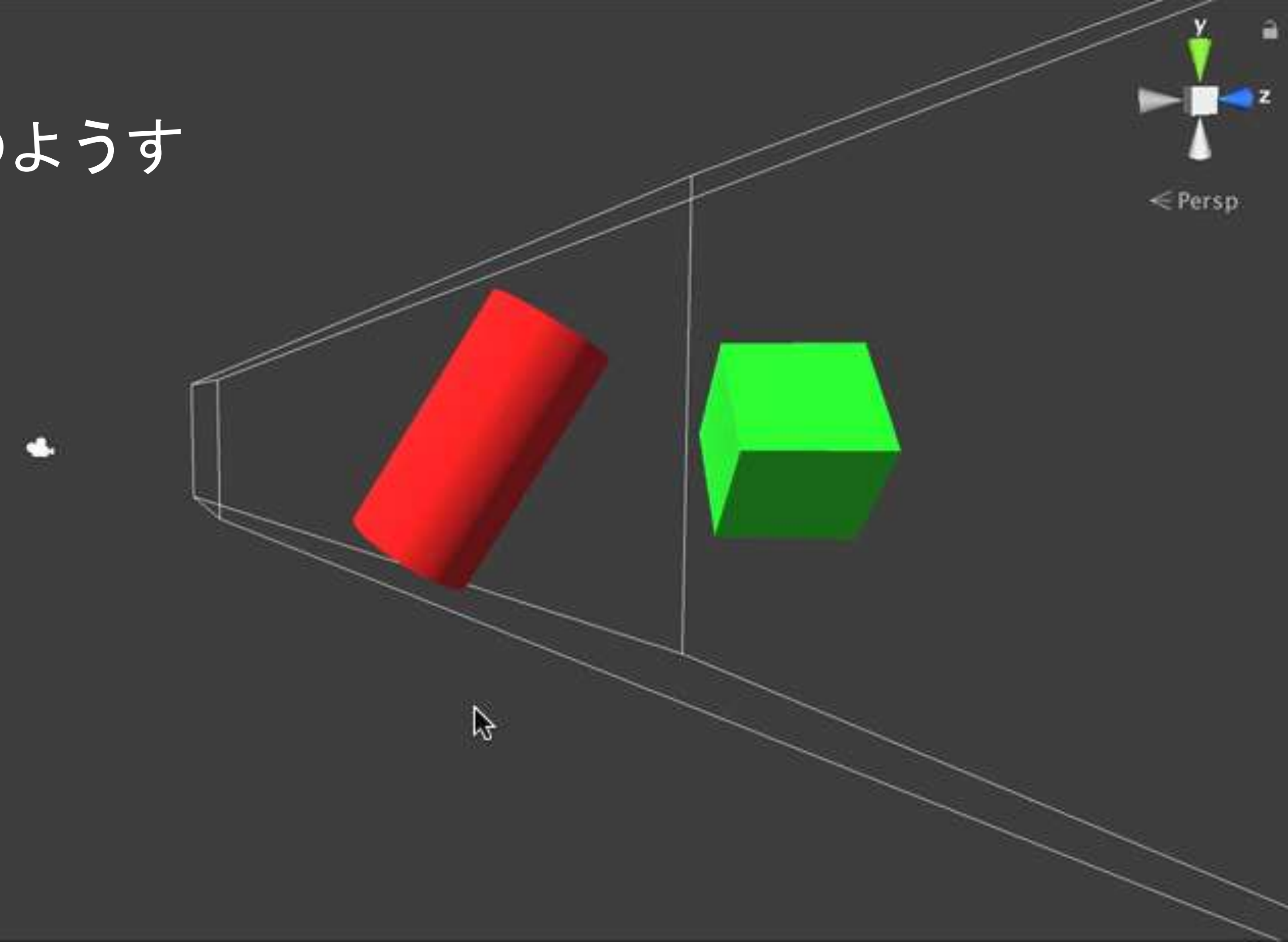
モニタに映すための
透視変換

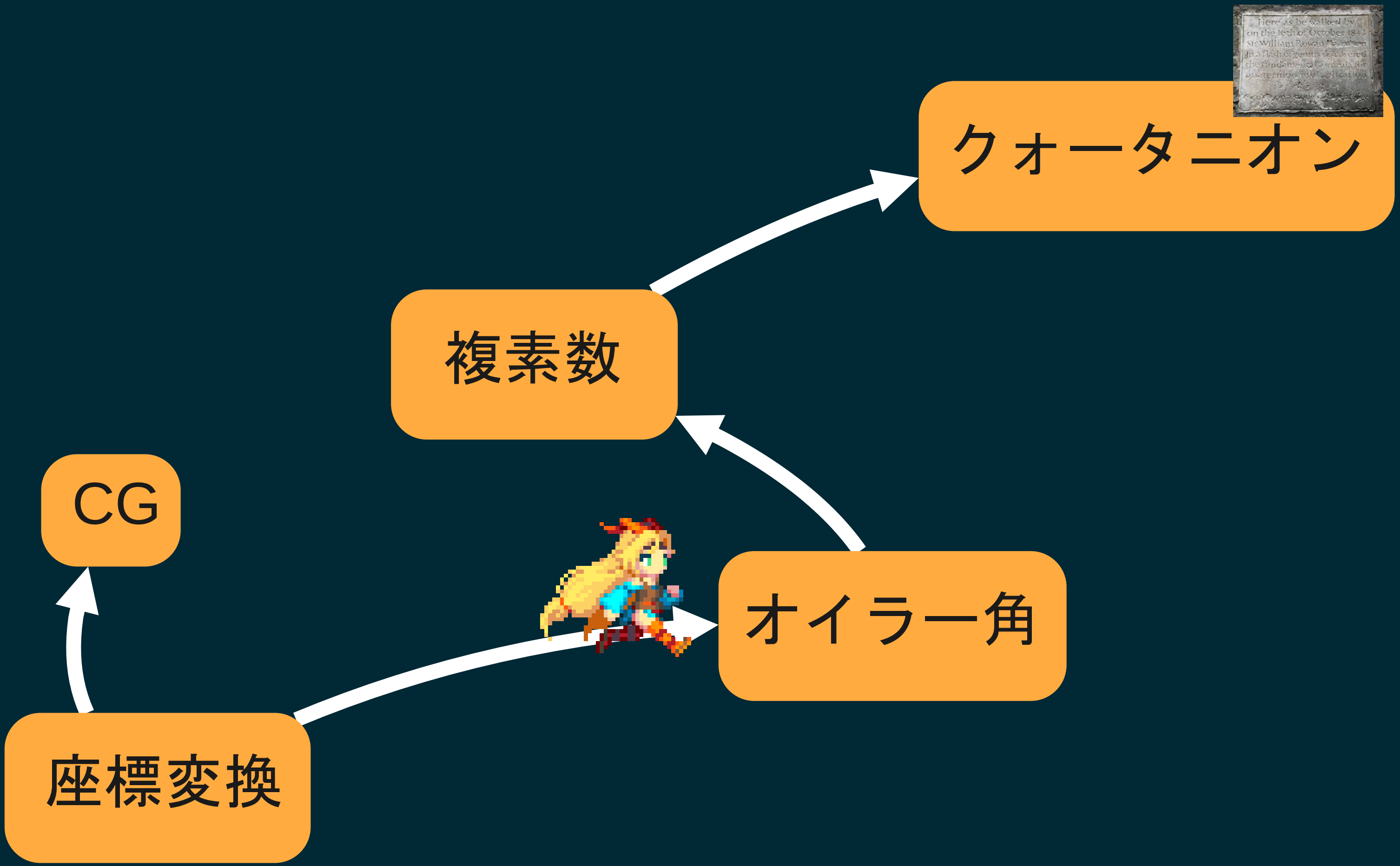


遠くのものを小さく！

この変換は
行列ではなく奥行きで割る

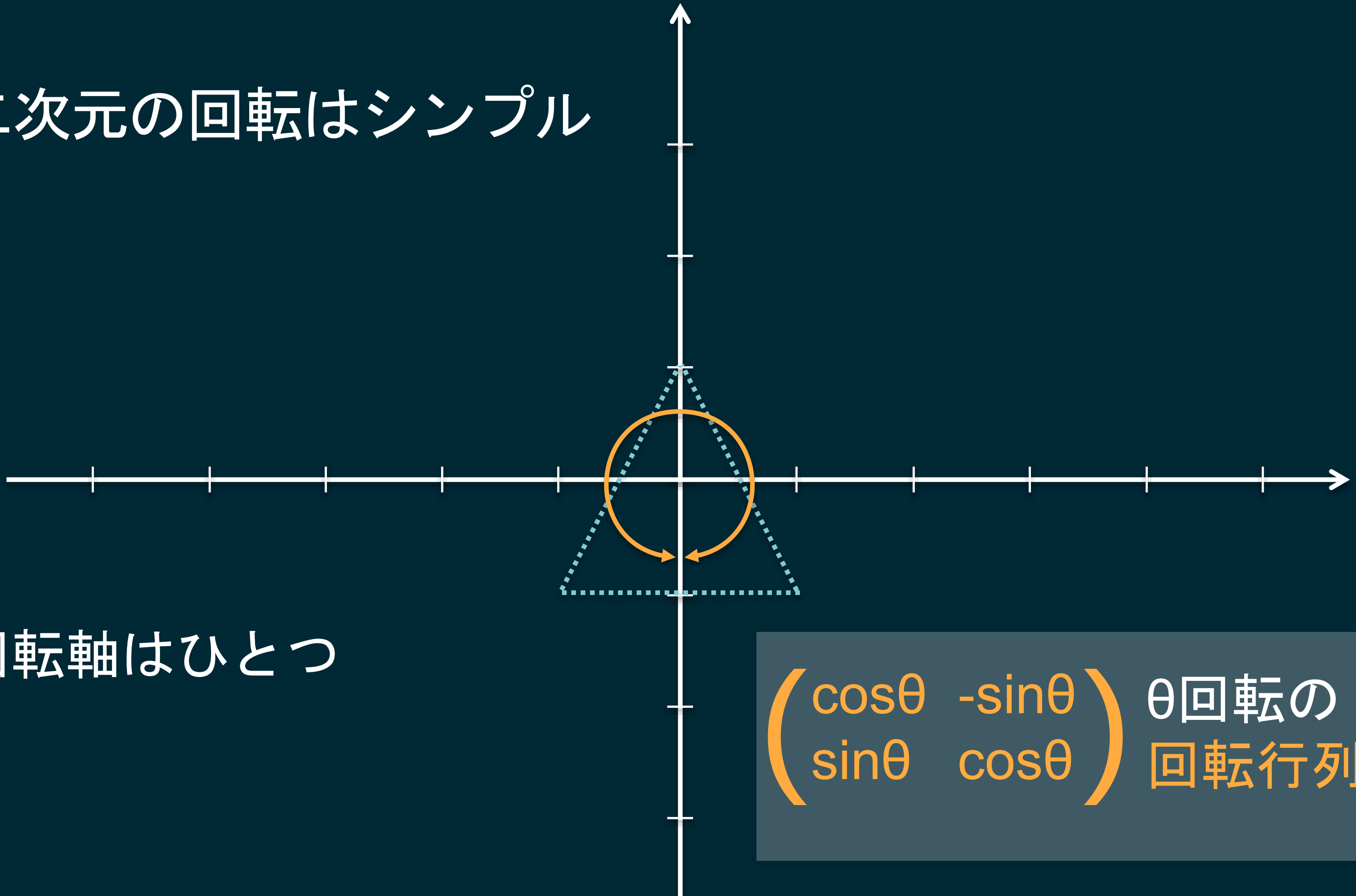
透視変換のようす




クォータニオン
Here as he walked by
on the 16th of October 1843
Sir William Rowan
in a flash of genius discovered
quaternion multiplication
$i^2 = j^2 = k^2 = ijk = -1$
複素数
CG
オイラー角
座標変換

オイラー角方式

二次元の回転はシンプル

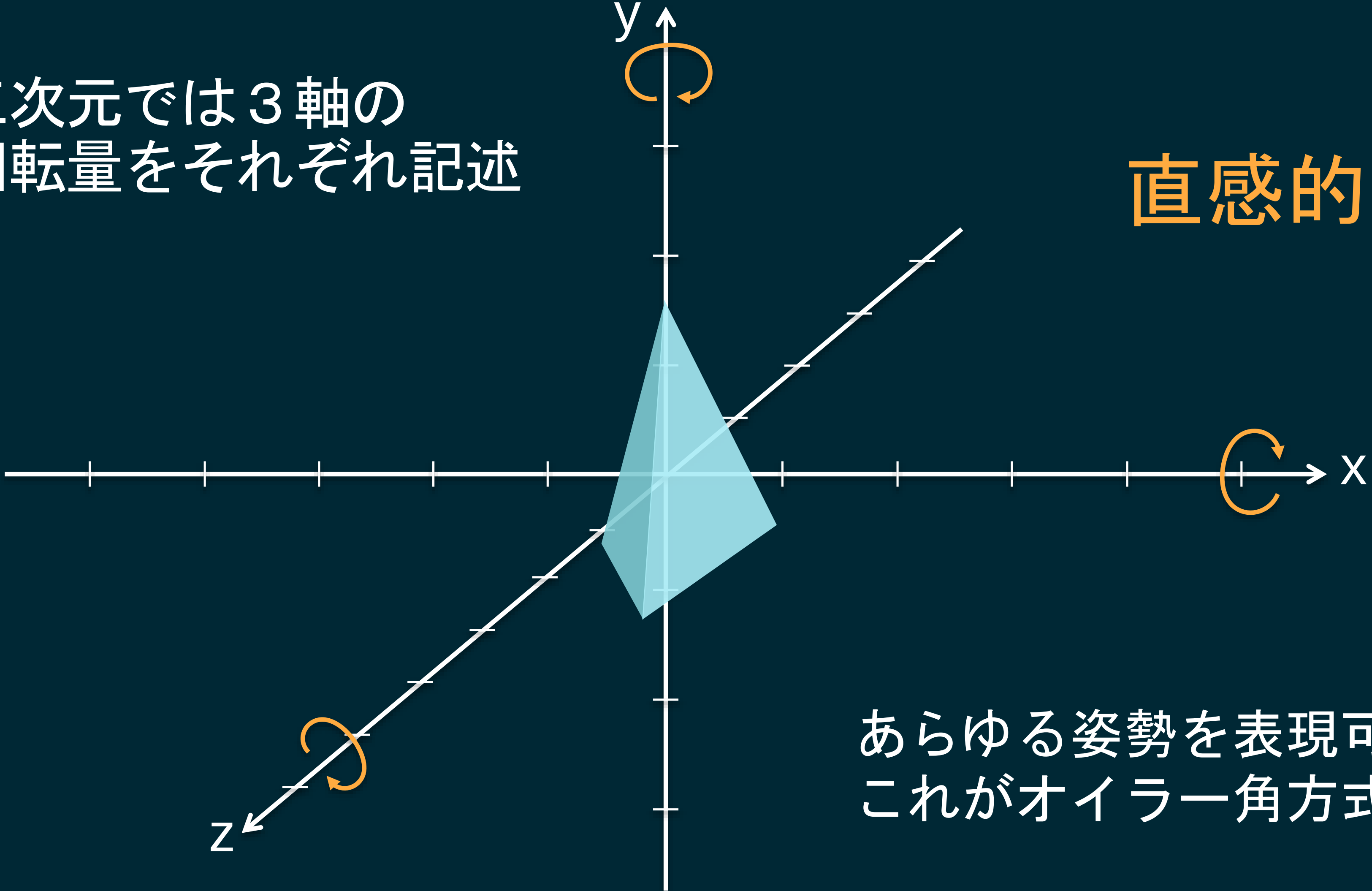


回転軸はひとつ

$$\begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix}$$ θ回転の回転行列

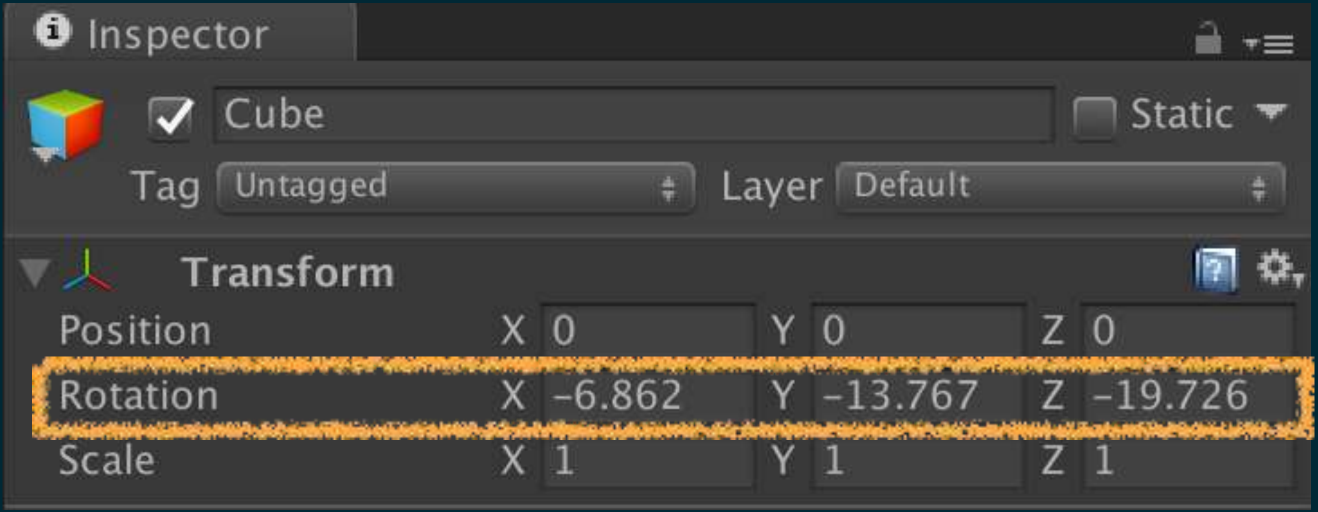
三次元では３軸の
回転量をそれぞれ記述

直感的！



あらゆる姿勢を表現可能
これがオイラー角方式

Unity の Inspector の Rotation は…



表示に関しては
オイラー角方式

オイラー角方式の回転には順番がある

Z　Ｚ軸まわりの回転行列

X　Ｘ軸まわりの回転行列

Y　Ｙ軸まわりの回転行列

P’=YXZP

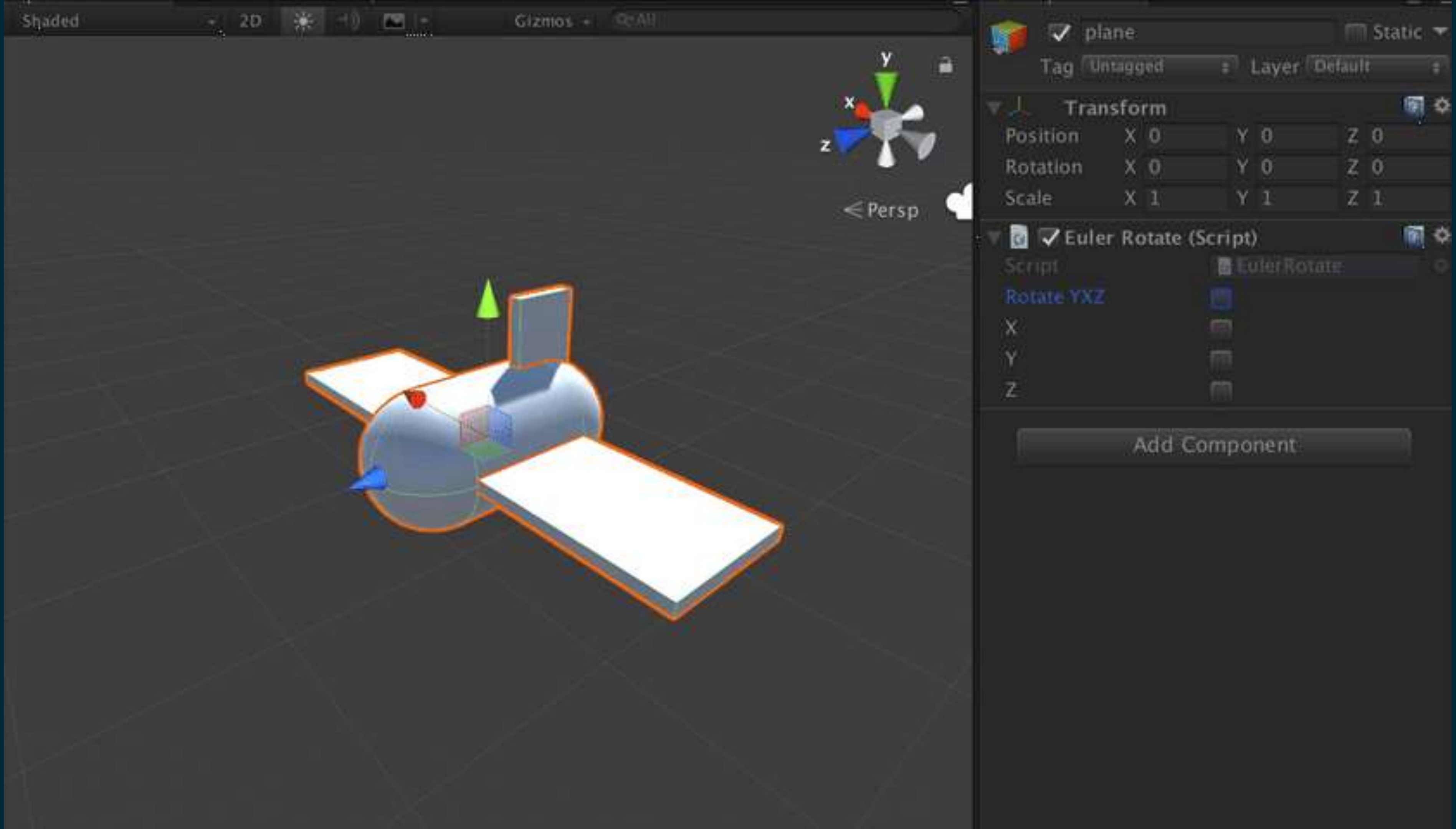
Unityは
ZYXの順番

プログラム例：
オイラー角で作った回転を代入

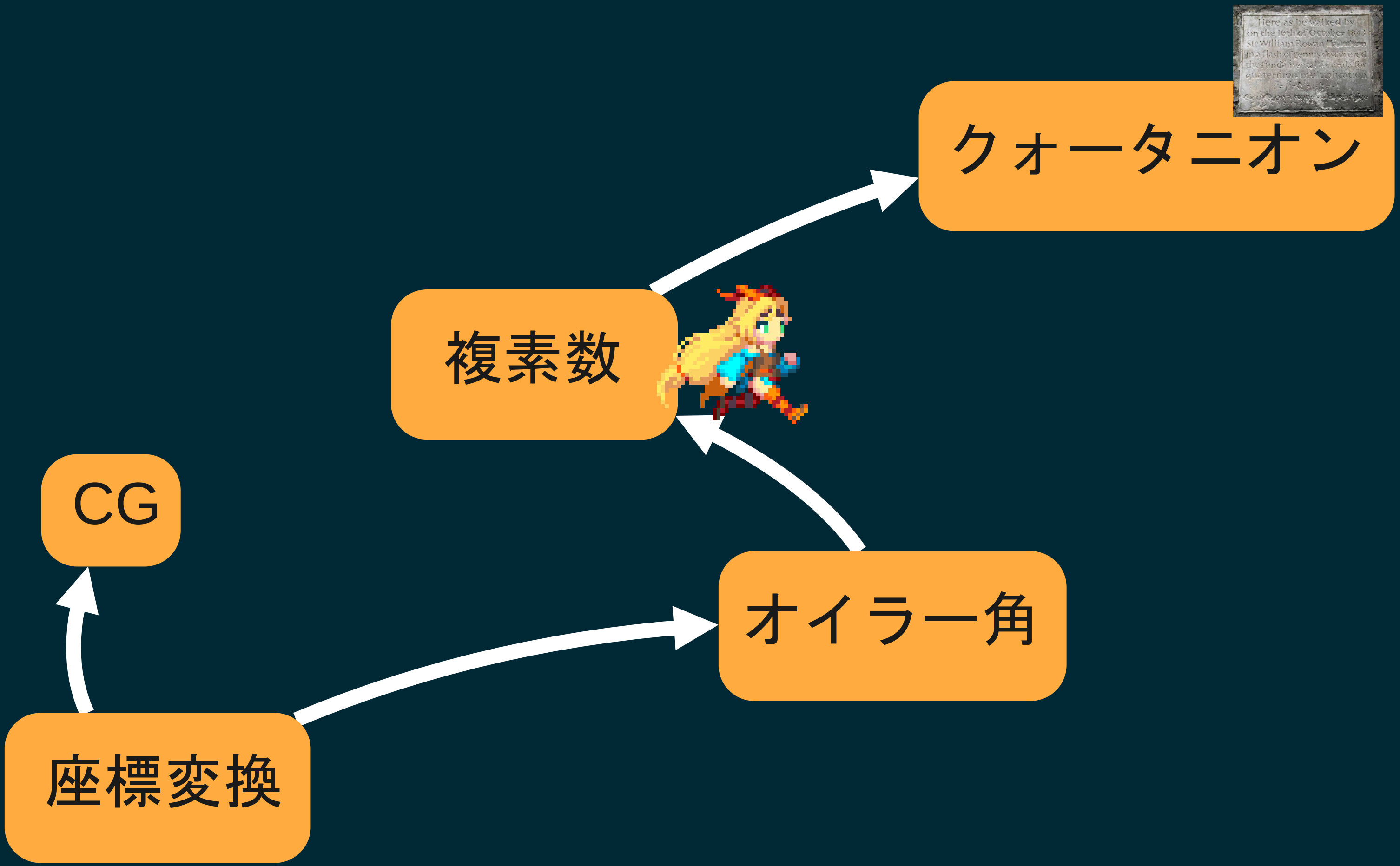
```
transform.rotation = Quaternion.Euler(x, y, z);
```

② ③ ①

変換を３回重ねている



Shaded
2D
Gizmos
All
y
x
z
Persp
plane
Static
Tag Untagged
Layer Default
Transform
Position X 0 Y 0 Z 0
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1
Euler Rotate (Script)
Script EulerRotate
Rotate YXZ
X
Y
Z
Add Component



クォータニオン
Here as he walked by
on the 16th of October 1843
Sir William Rowan
In a flash of genius discovered
quaternion multiplication
複素数
CG
オイラー角
座標変換

複素数と複素平面

クォータニオンは
複素数の拡張

問題

$$x^2 = -1$$

$$x^2 = -1$$

$$x = i, -i$$

i は虚数単位

虚数単位 i

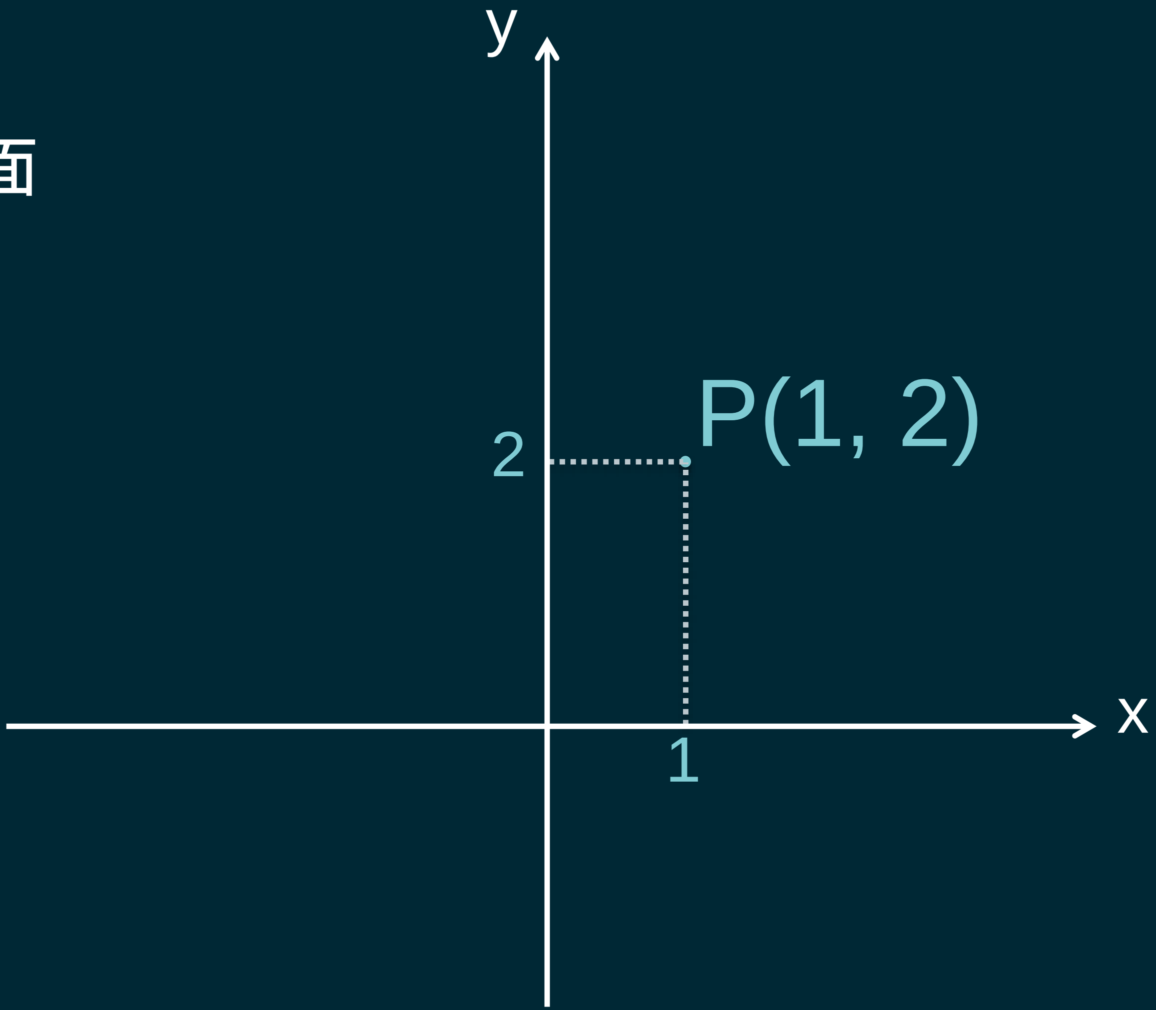
虚数の定義 $i^2 = -1$

$(\sqrt{-1} = i)$

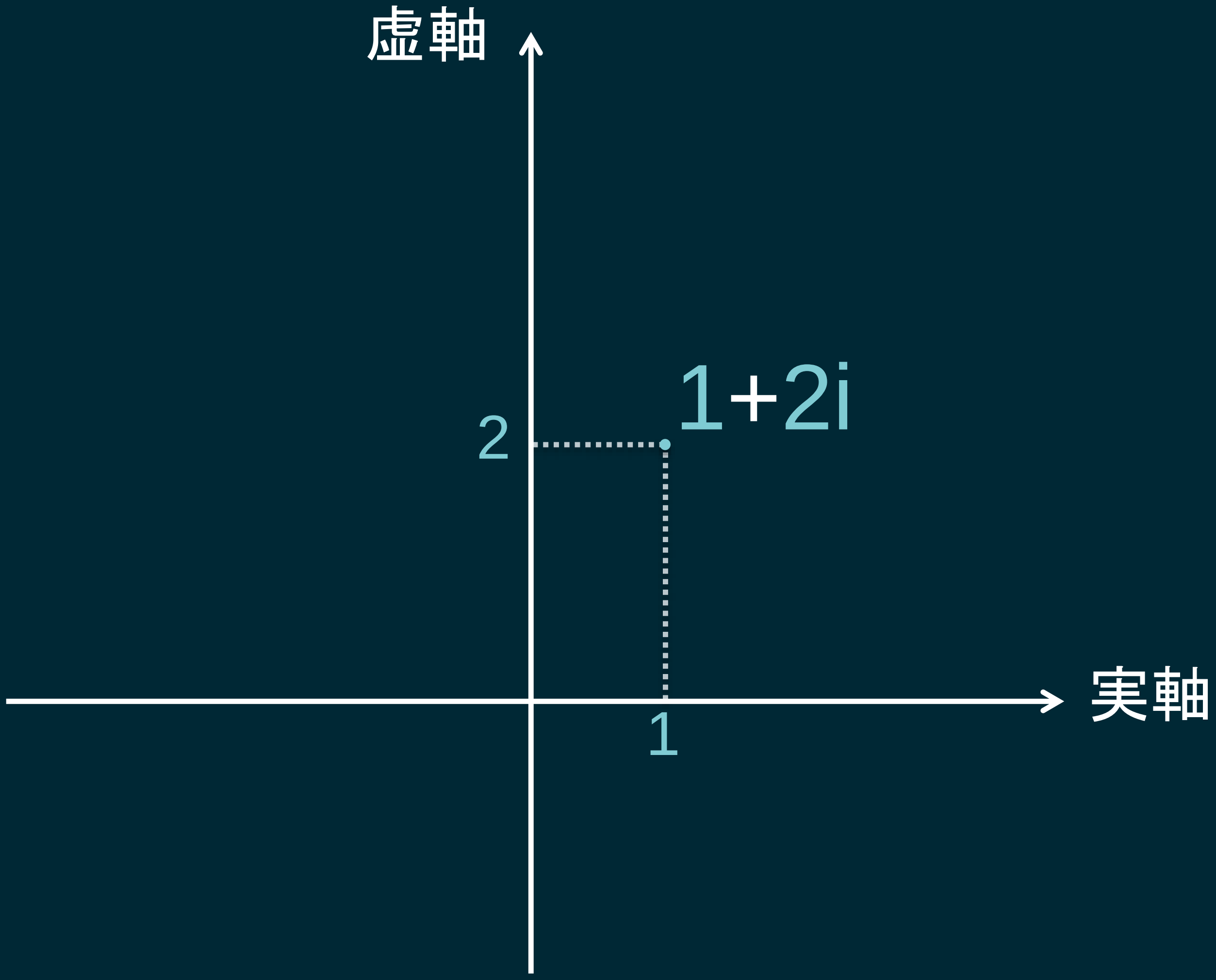
複素数 $a + bi$

a : 実部
b : 虚部

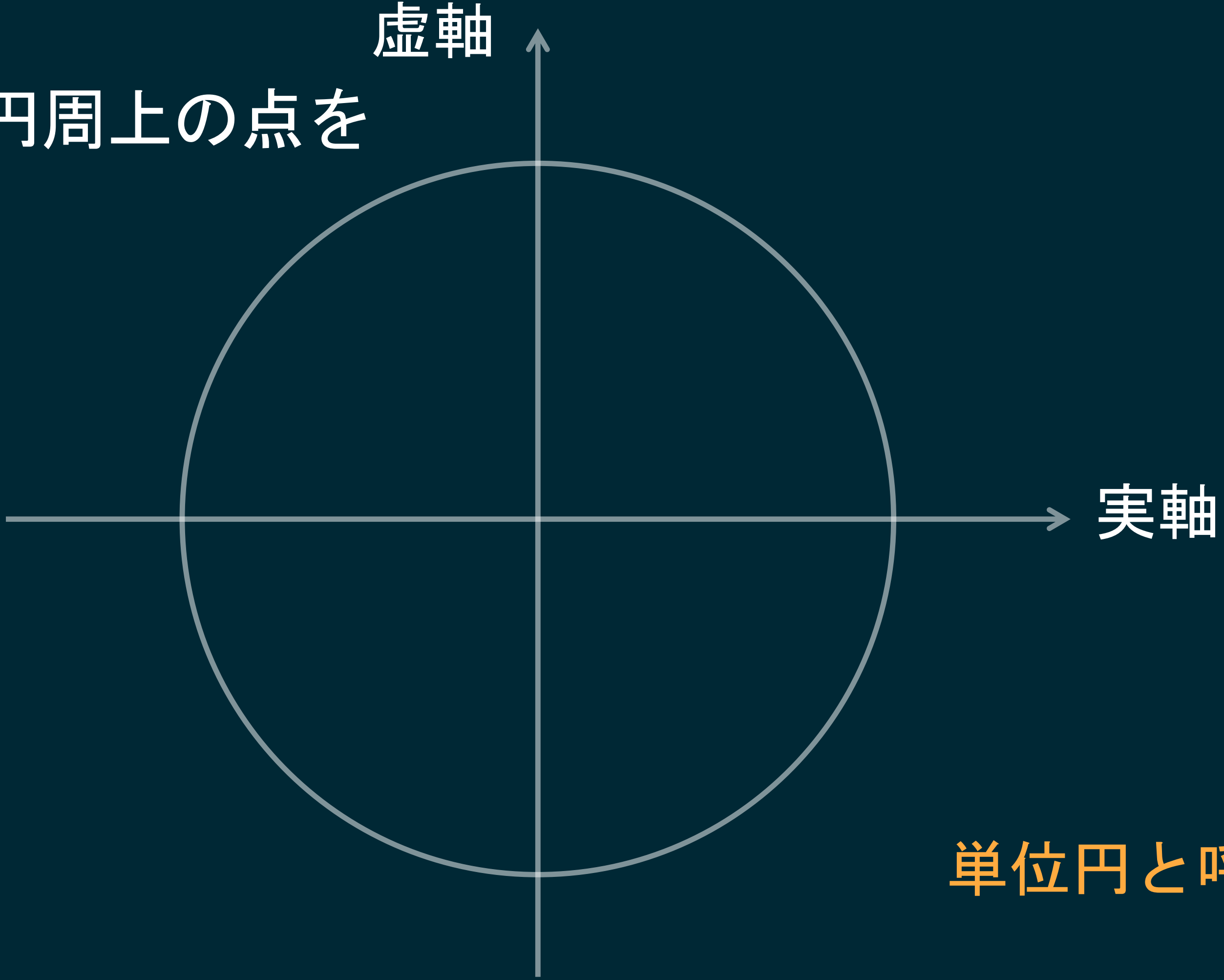
普通の平面



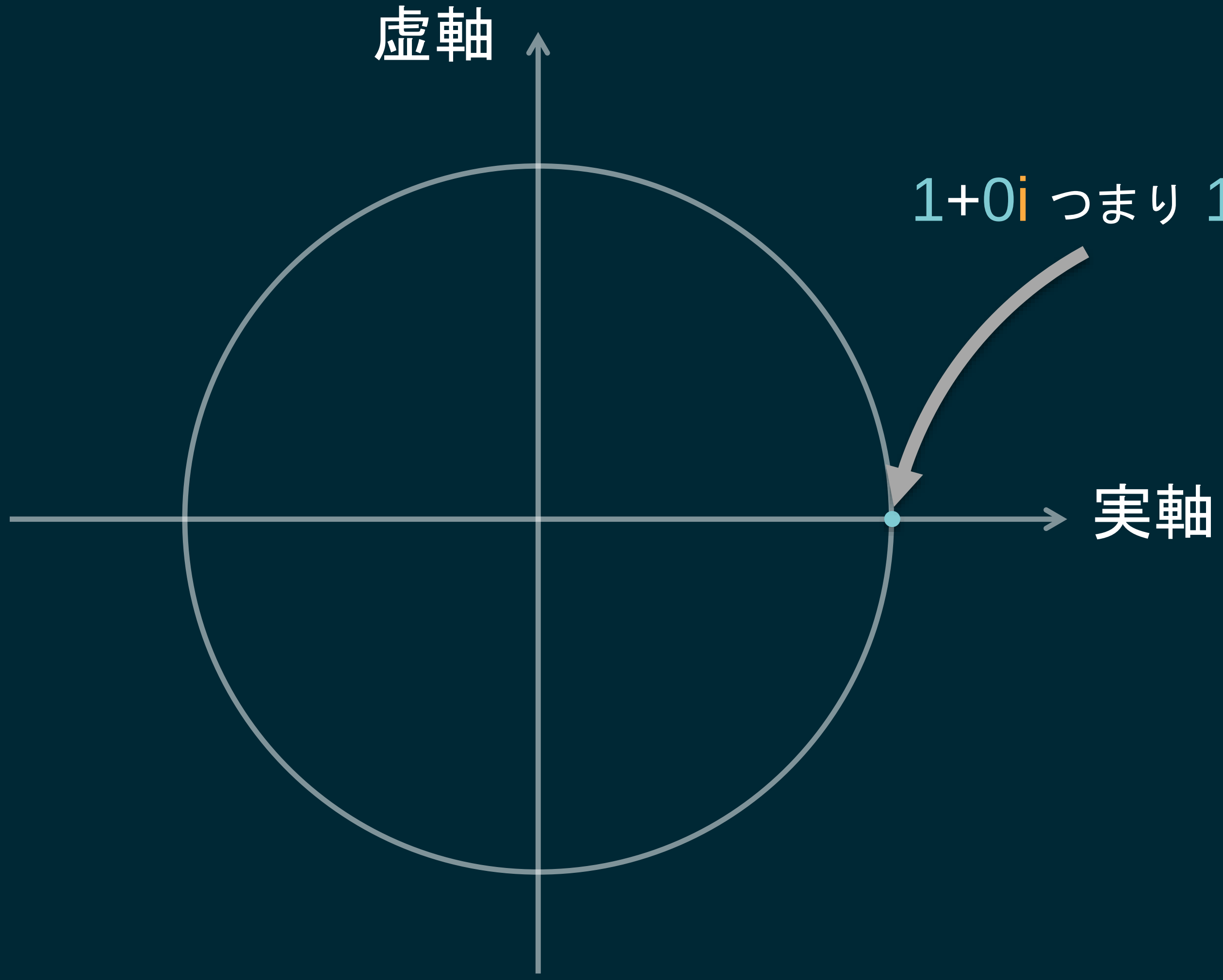
複素平面

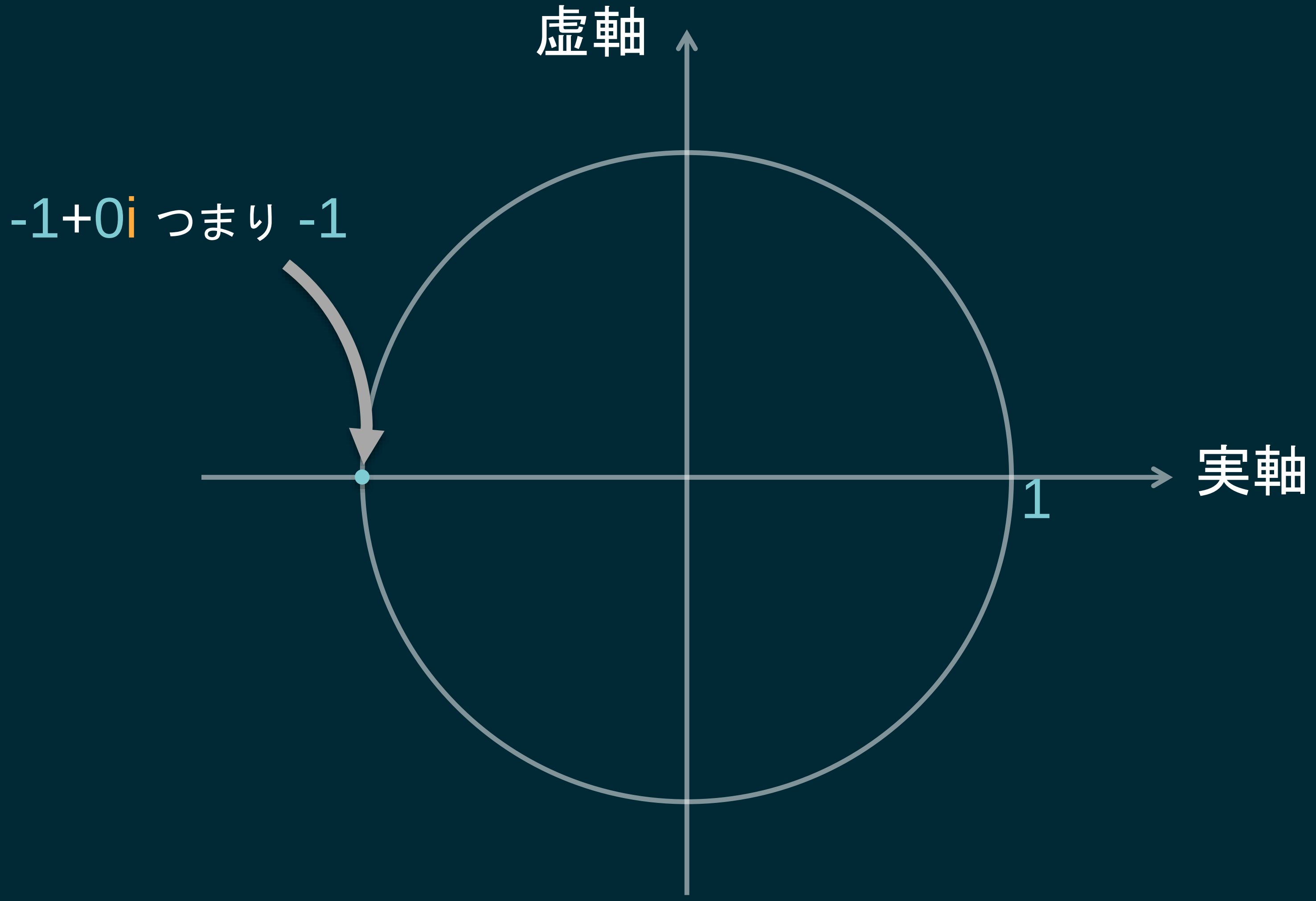


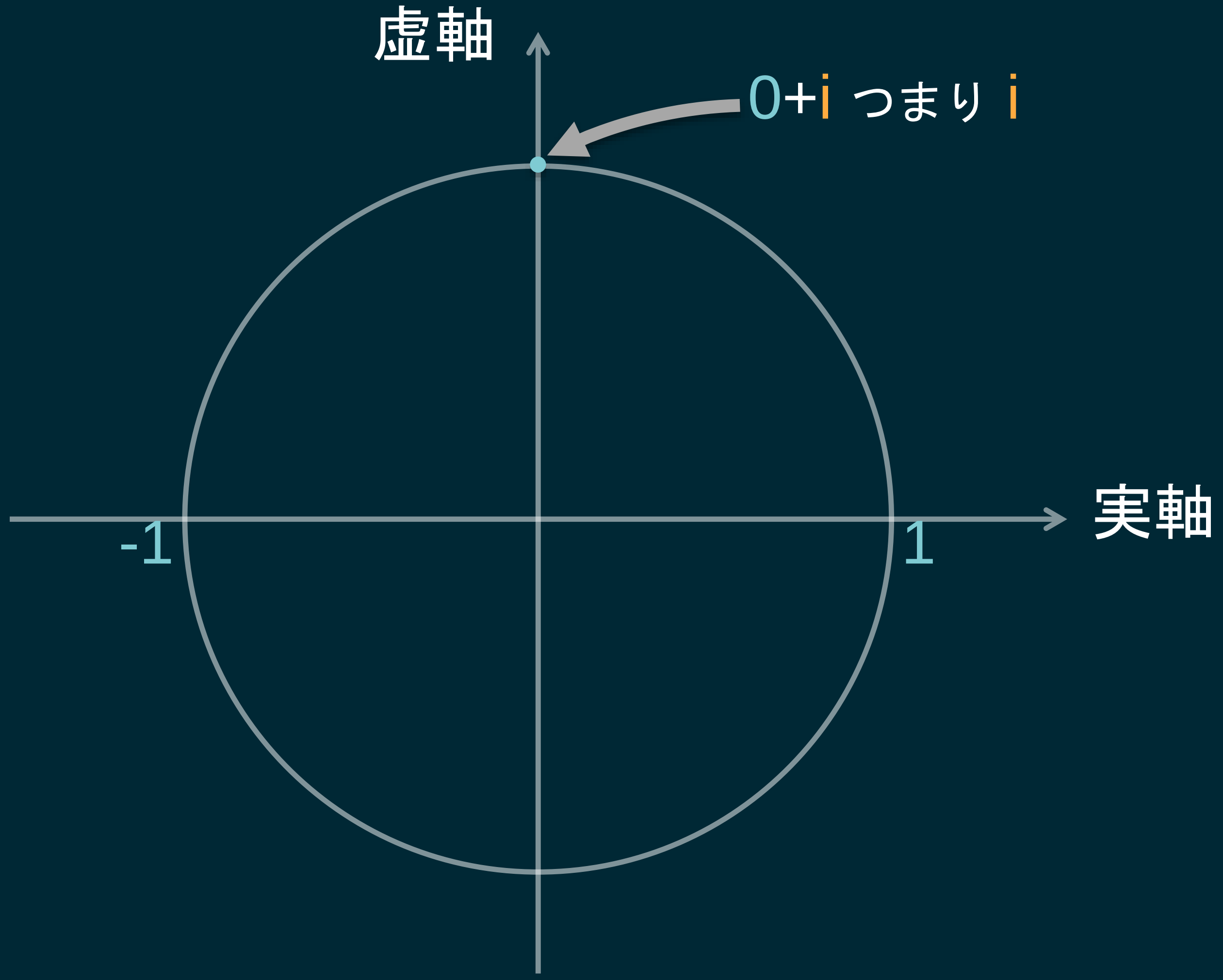
半径１の円周上の点を
考える

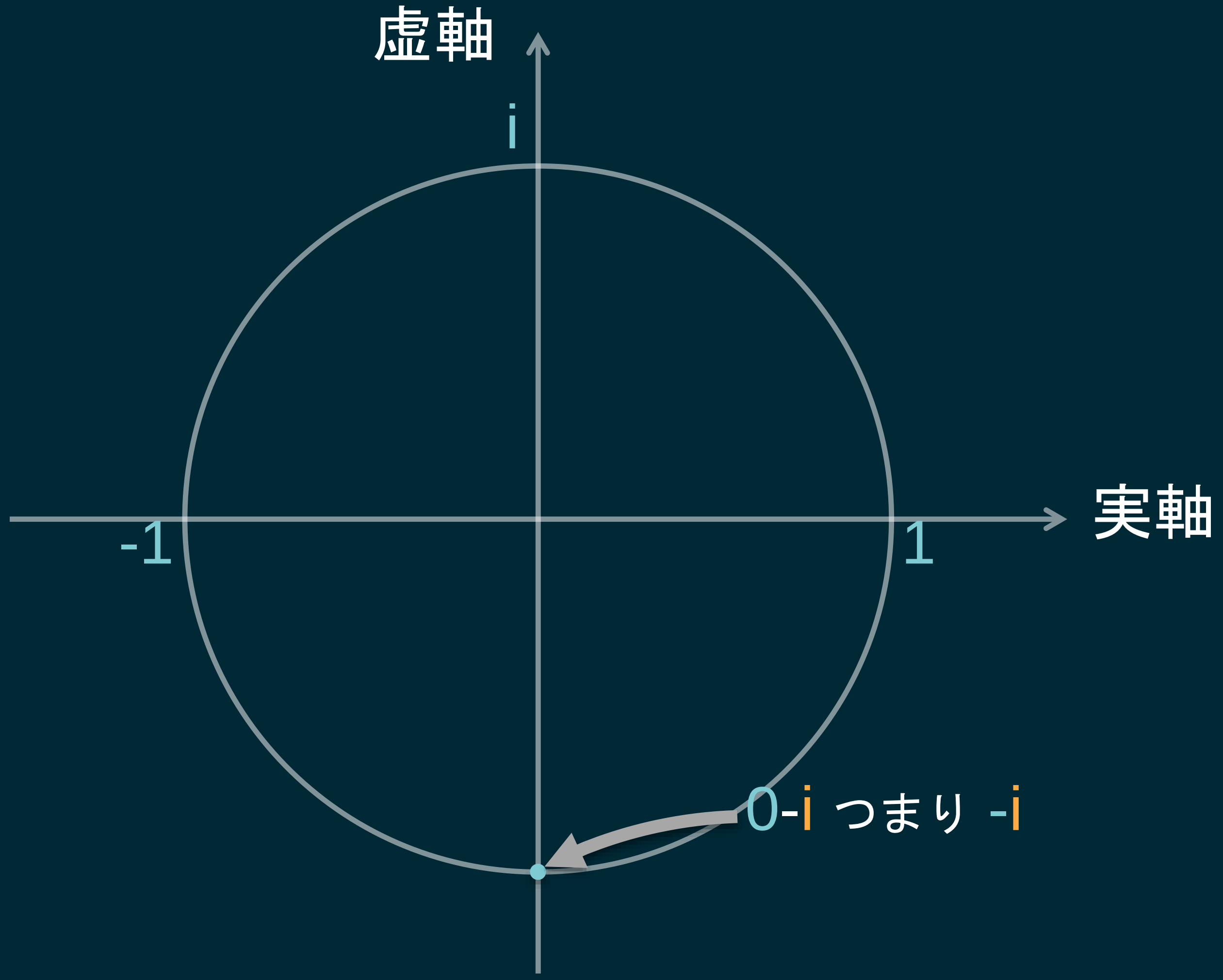


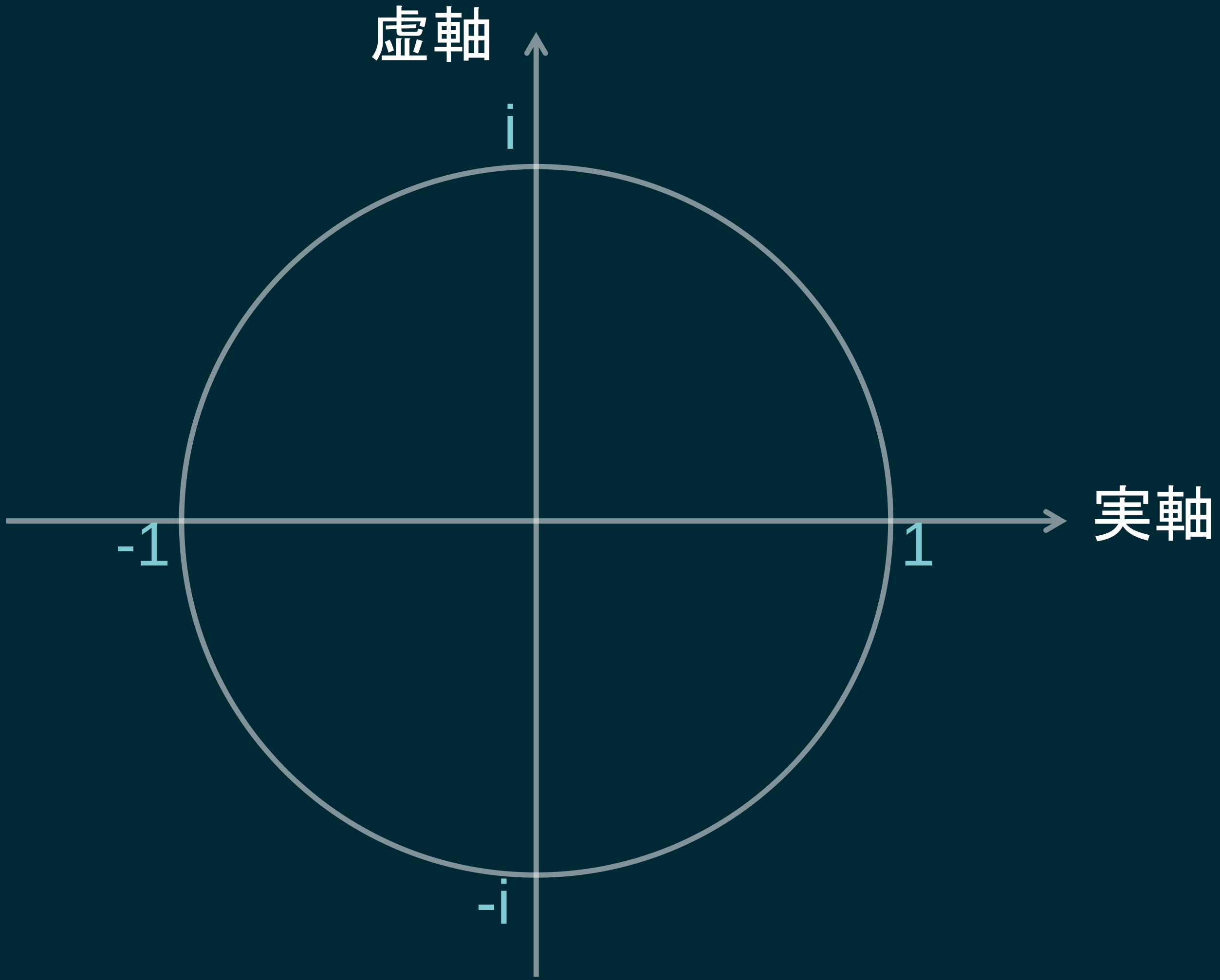
単位円と呼ぶ


虚軸
1+0i つまり 1
実軸

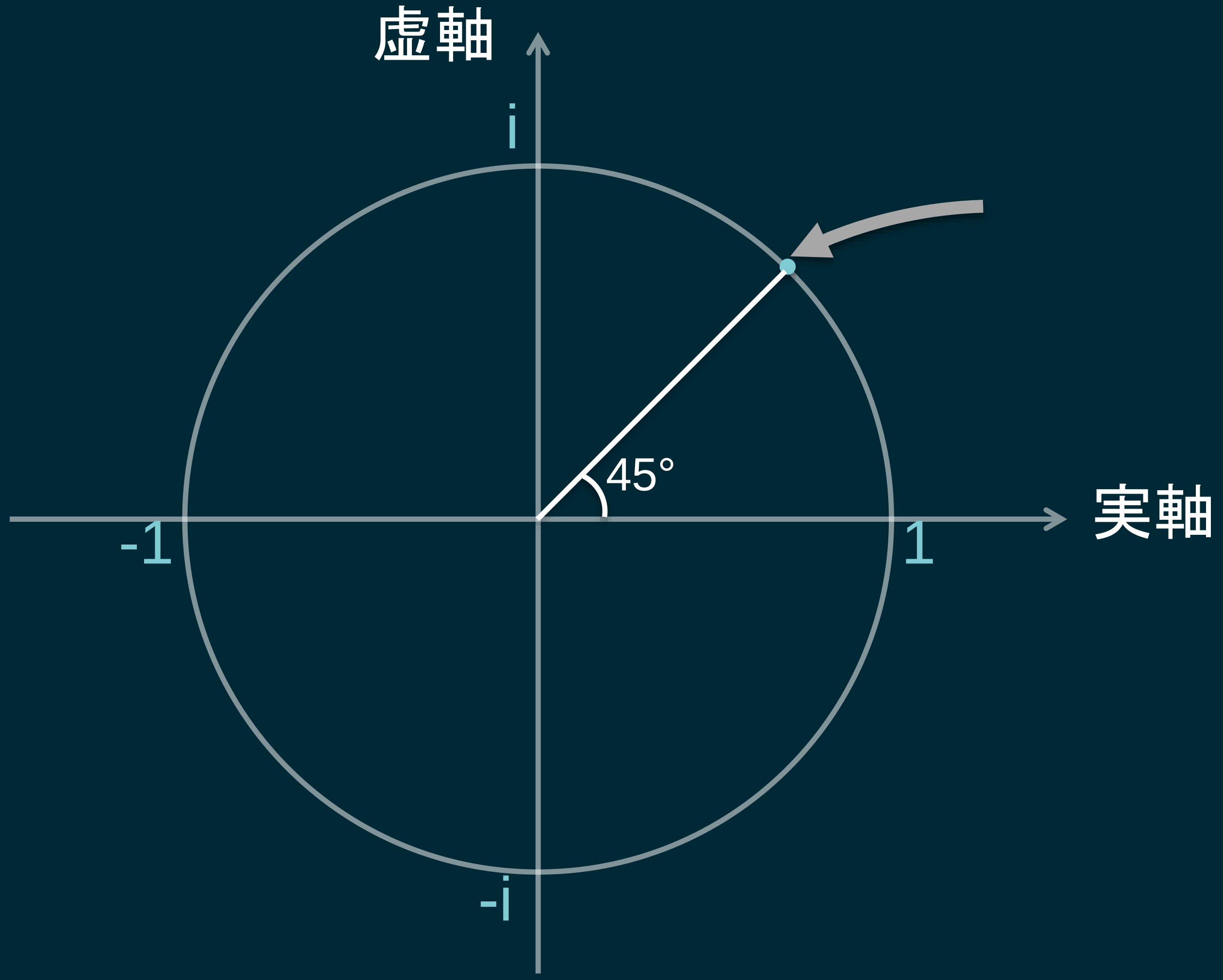

虚軸
-1+0i つまり -1
1
実軸


虚軸
0+i つまり i
実軸
-1
1

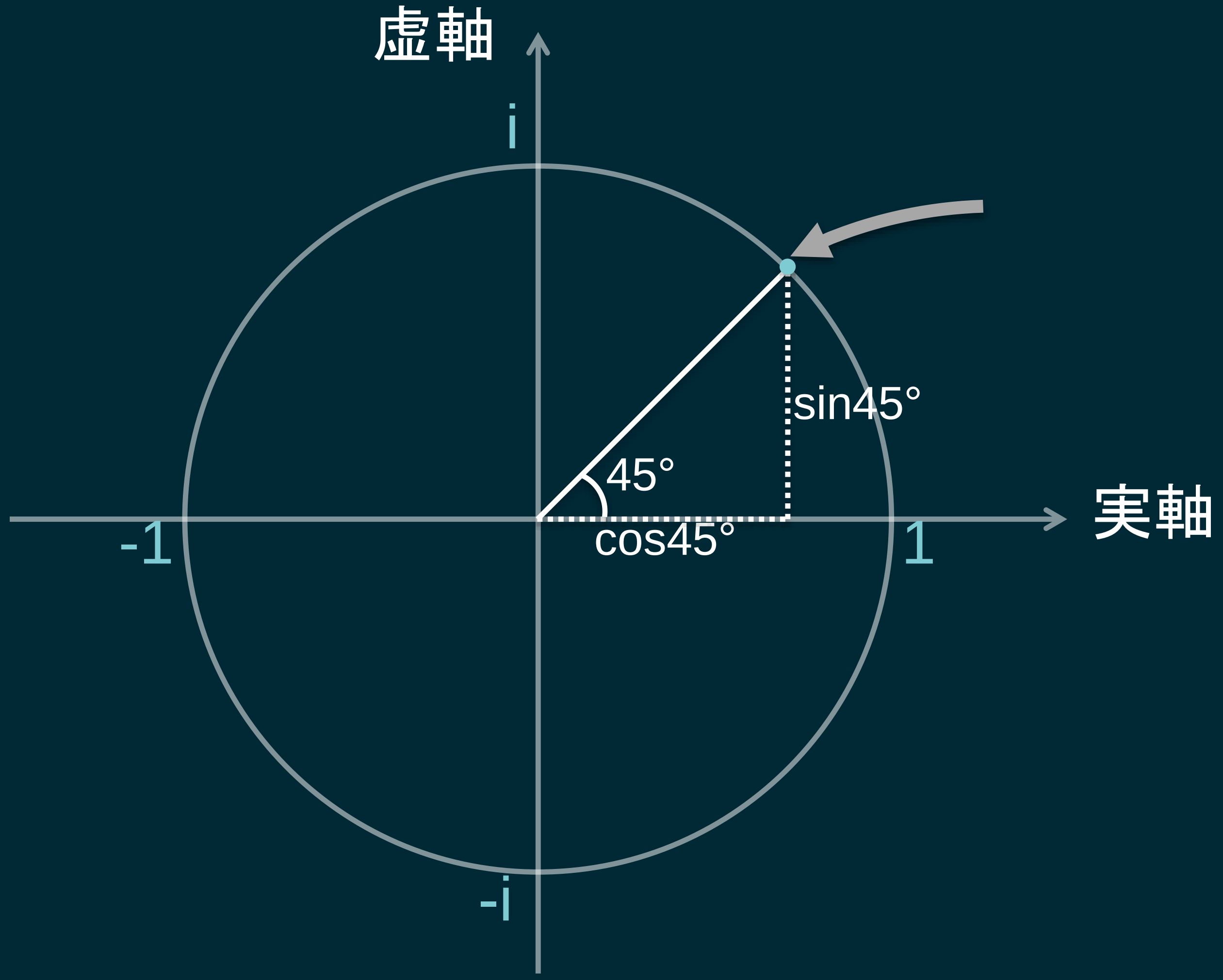

虚軸
i
-1
1
実軸
0-i つまり -i



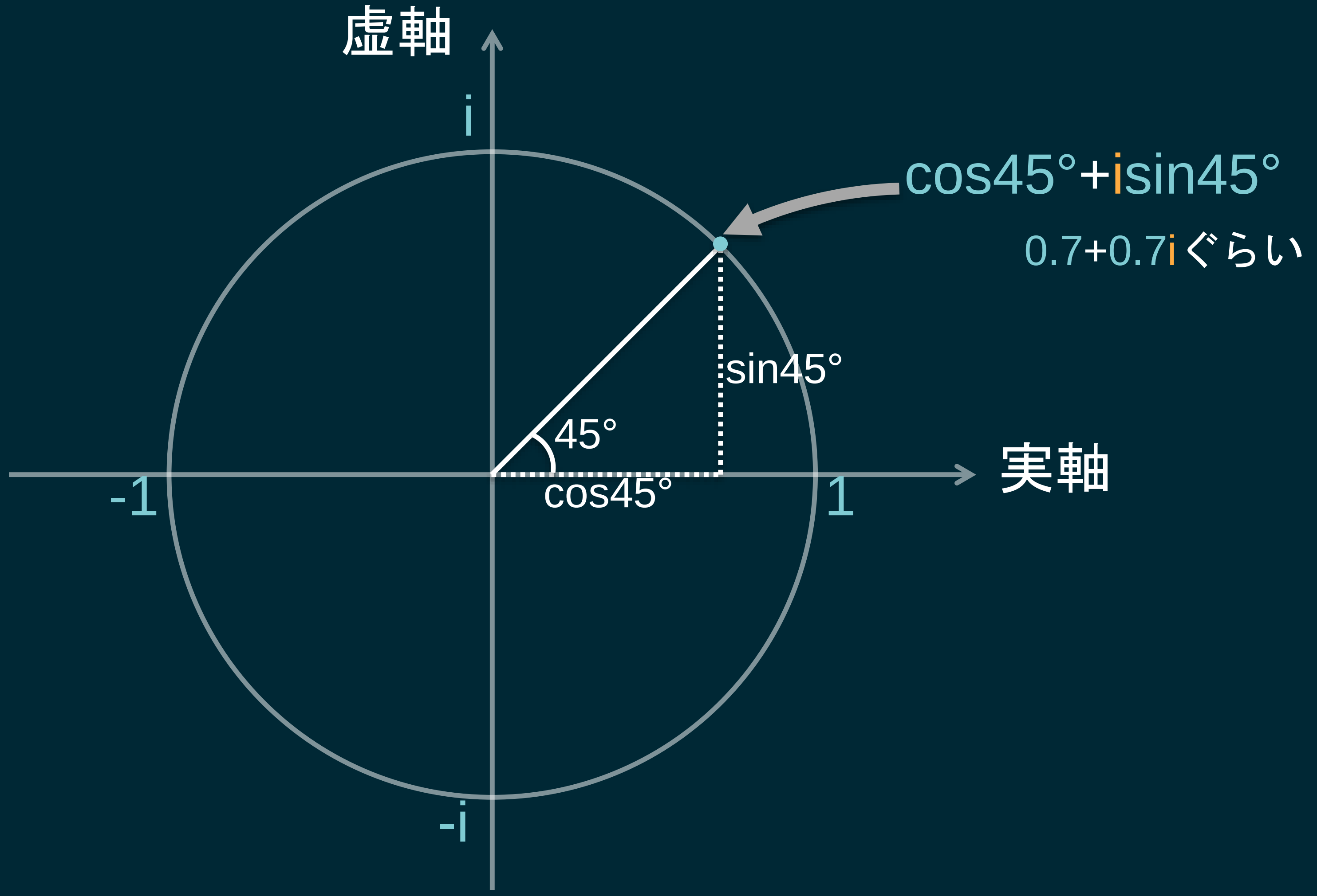
虚軸
i
実軸
-1
1
-i

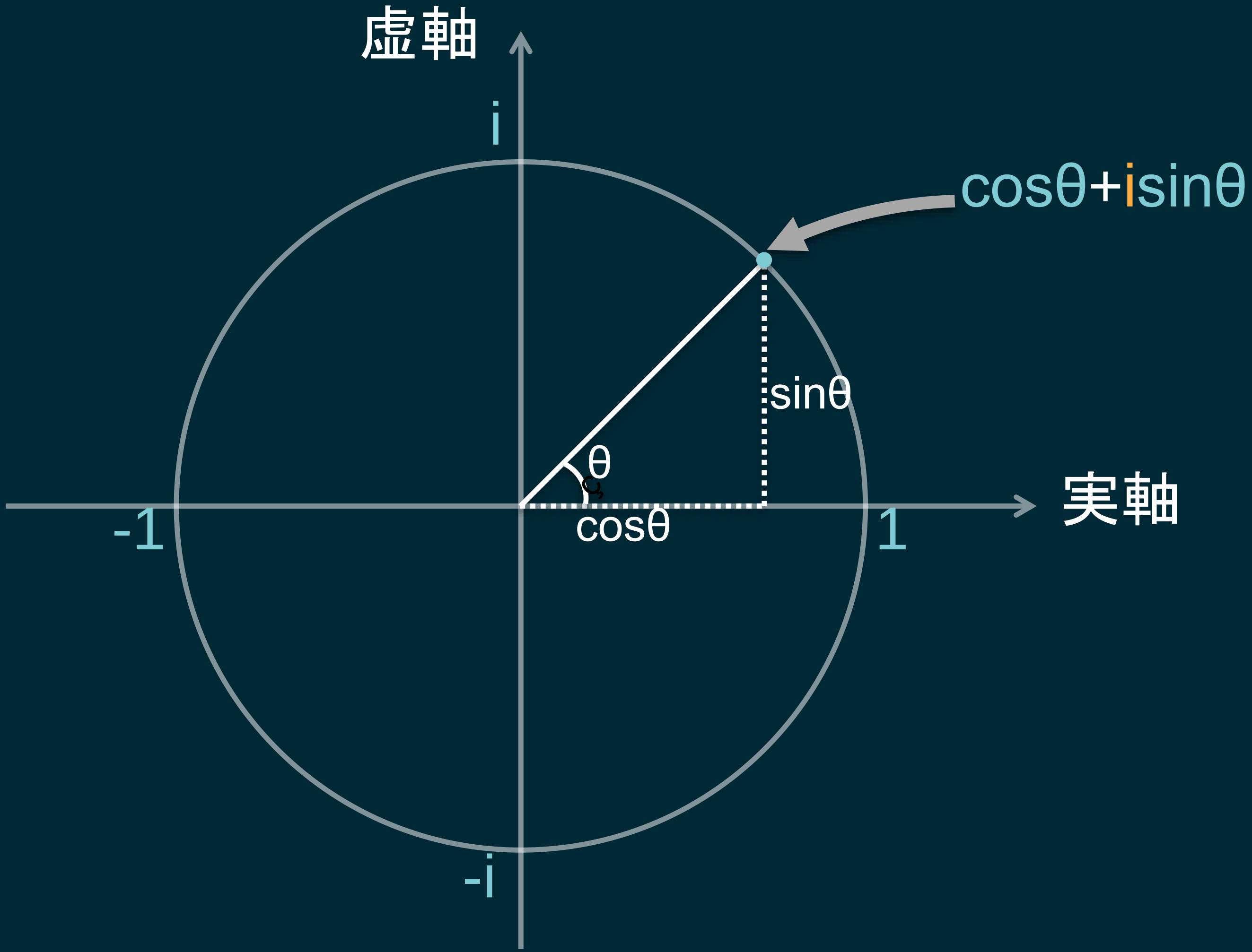


虚軸
i
45°
実軸
-1
1
-i



虚軸
i
sin45°
45°
-1
cos45°
1
実軸
-i

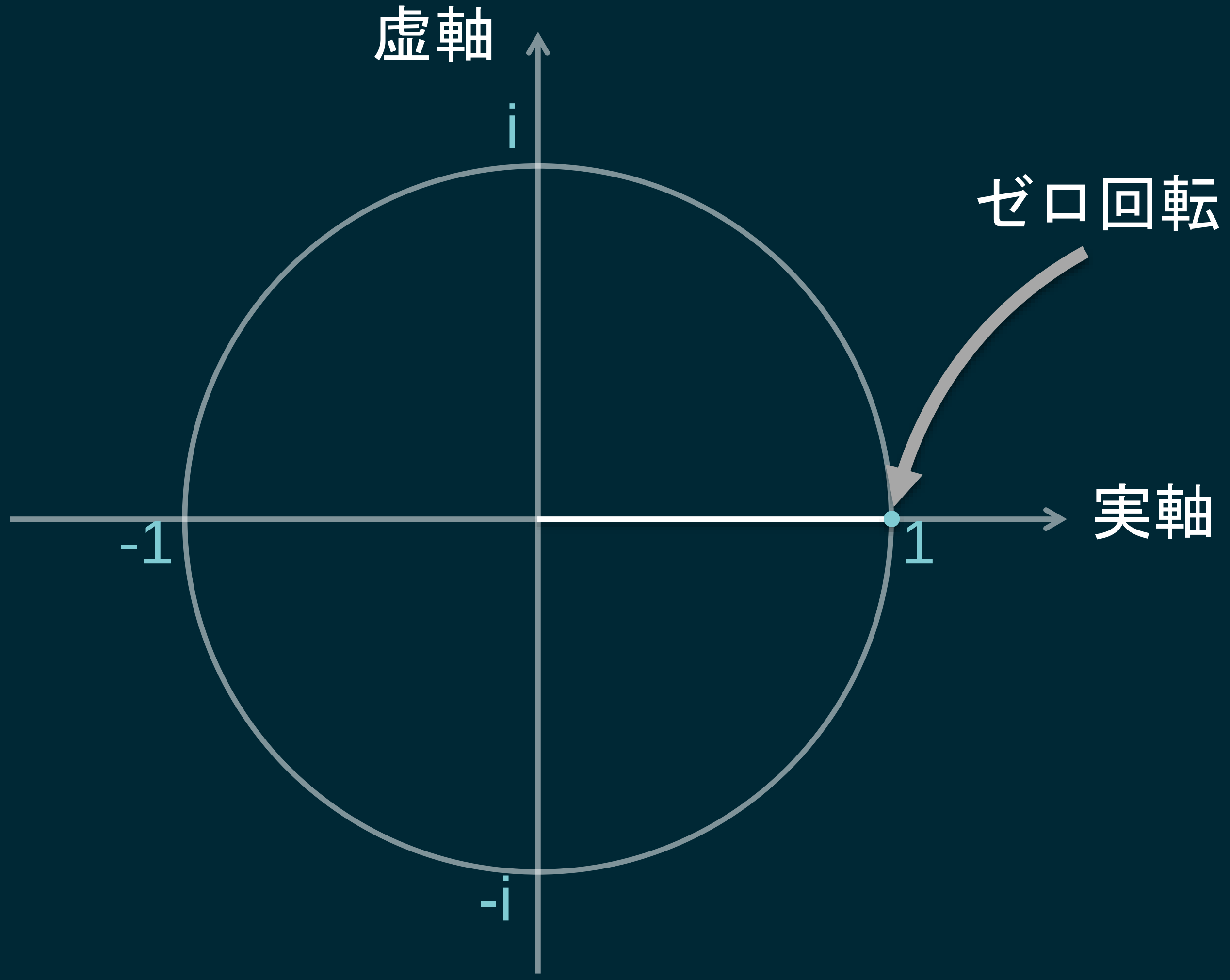

虚軸
i
cos45°+isin45°
0.7+0.7iぐらい
sin45°
45°
実軸
-1
cos45°
1
-i

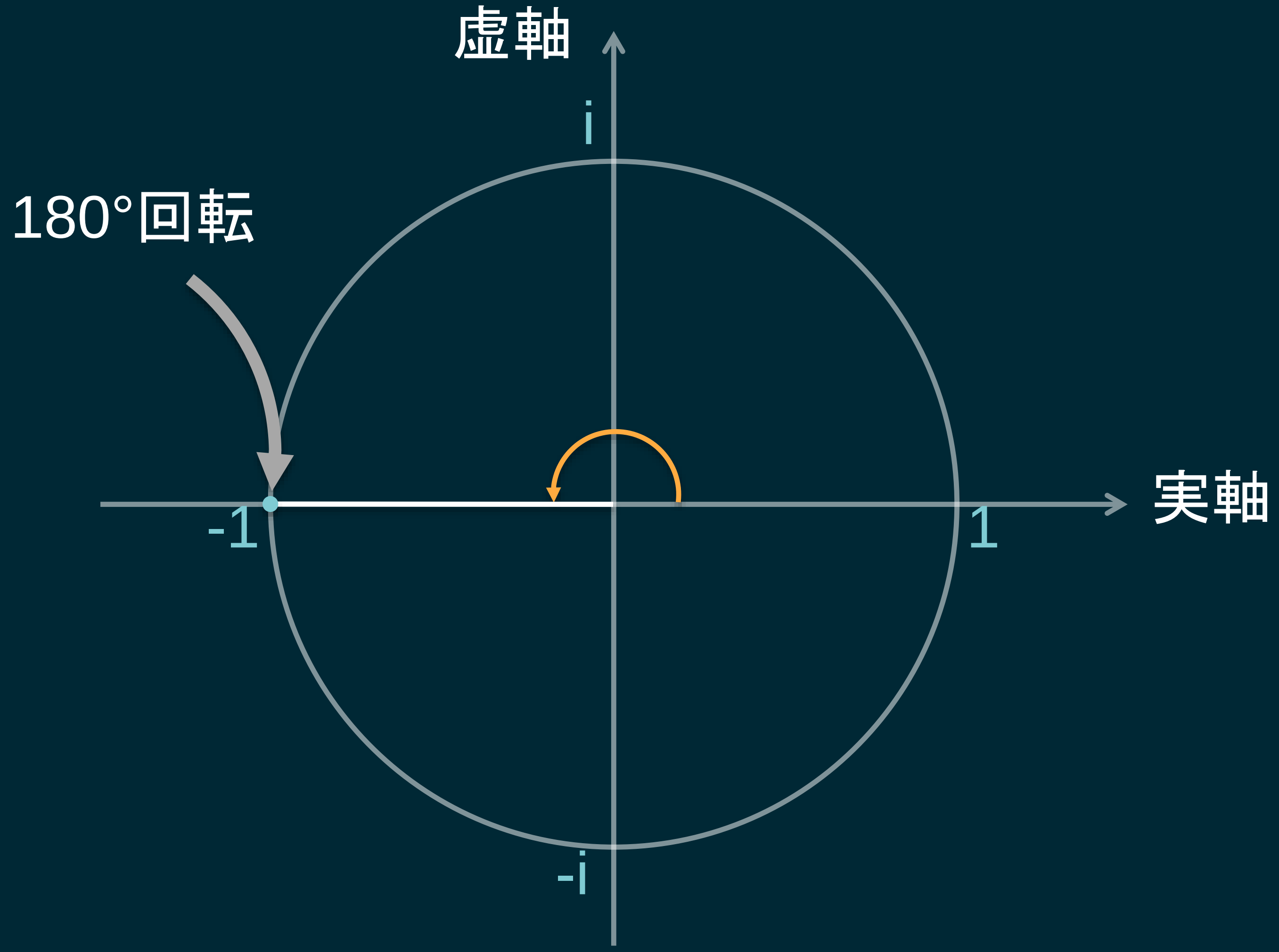


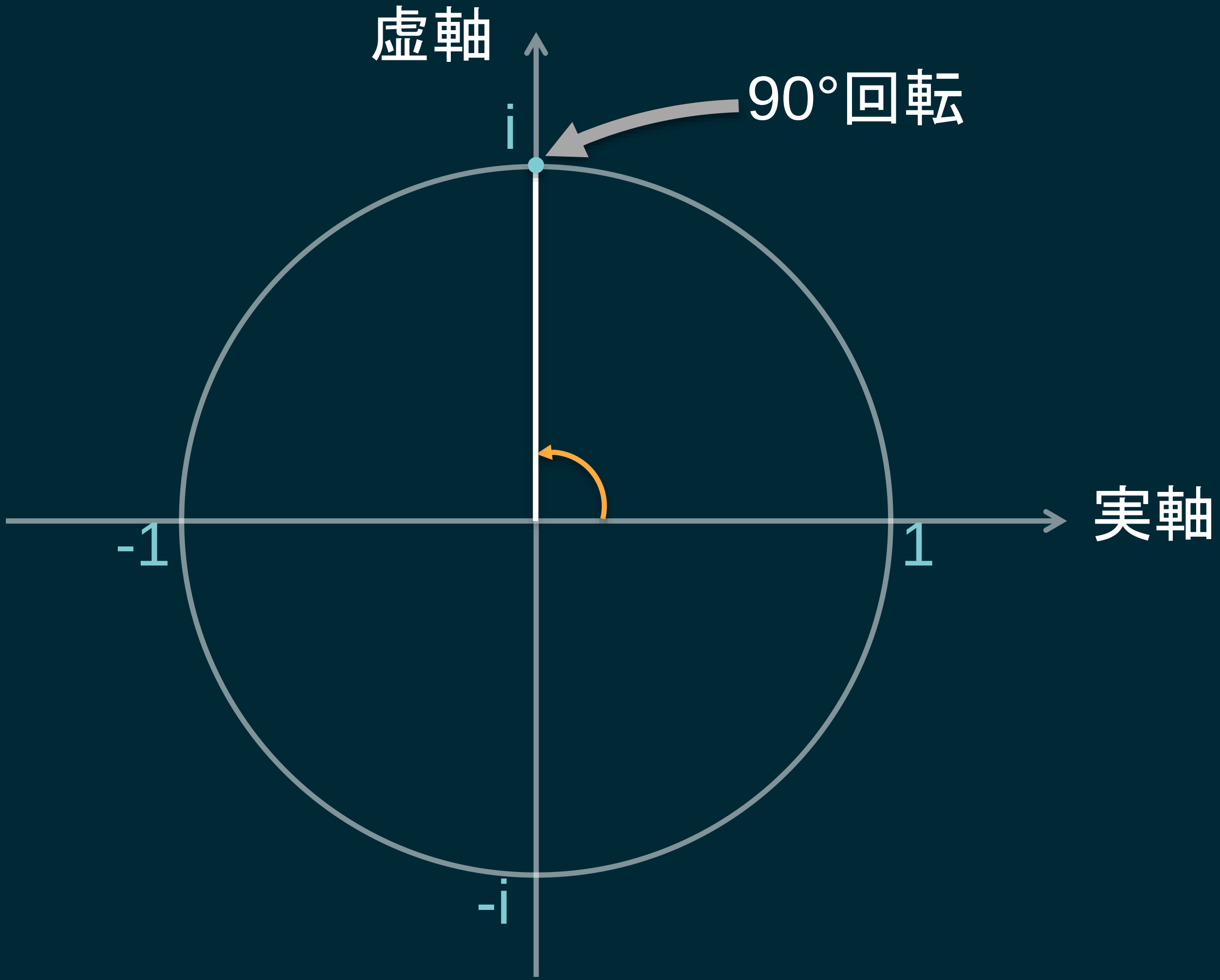
虚軸
i
cosθ+isinθ
sinθ
θ
実軸
-1
cosθ
1
-i

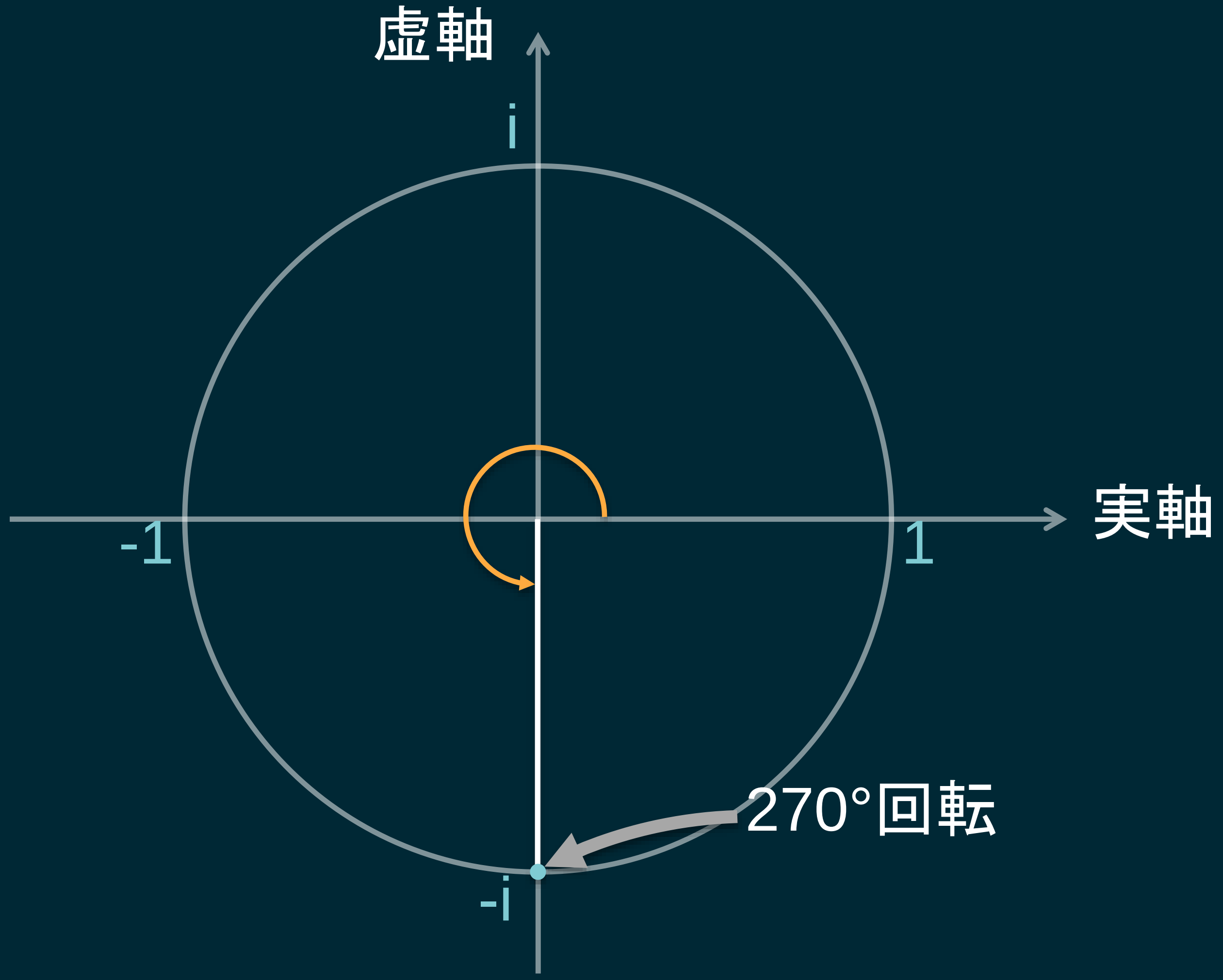
重大な事実

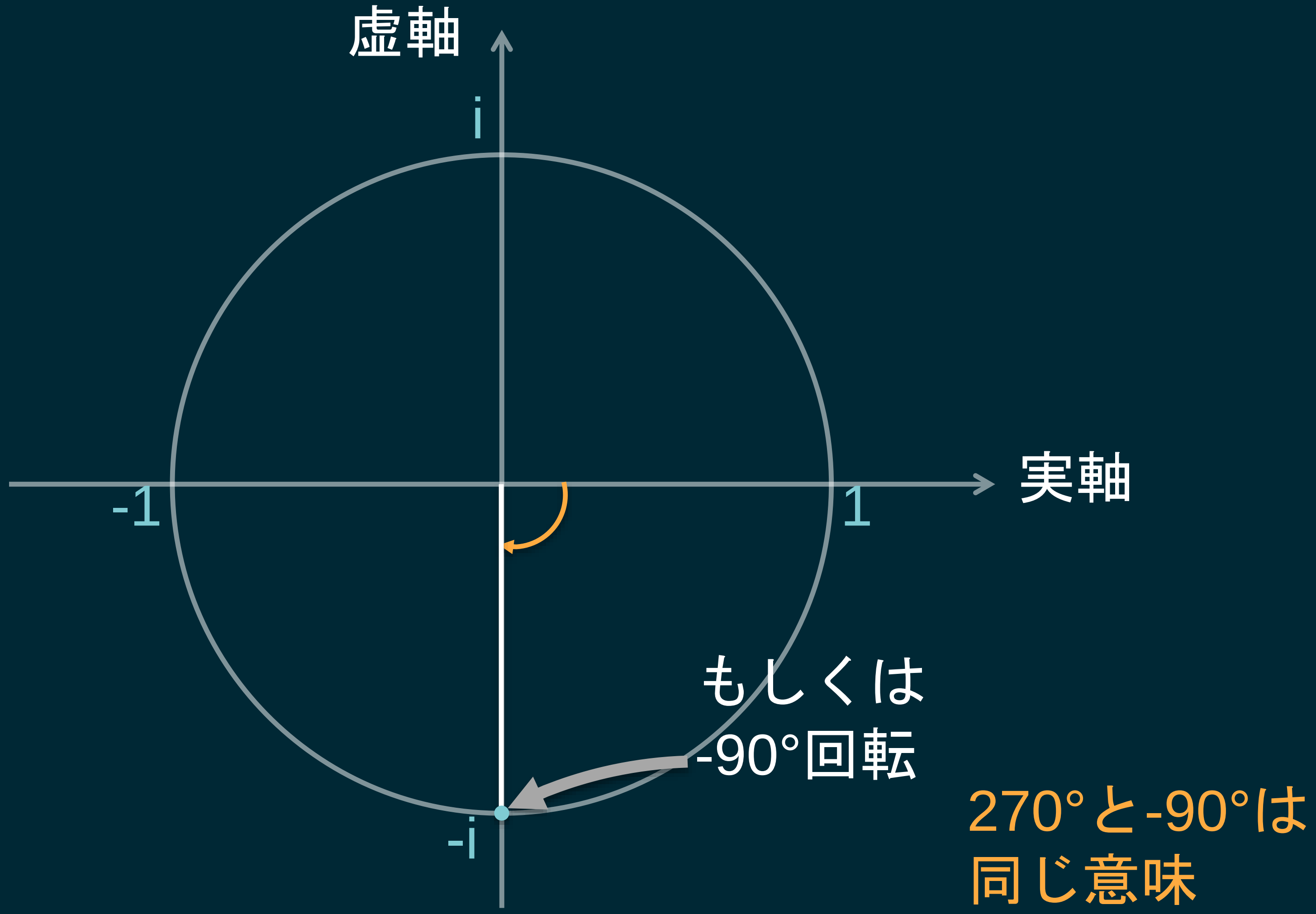
単位円上の点は回転だった


虚軸
i
ゼロ回転
実軸
-1
1
-i


虚軸
i
180°回転
-1
1
実軸
-i


虚軸
90°回転
i
実軸
-1
1
-i


虚軸
i
実軸
-1
1
270°回転
-i

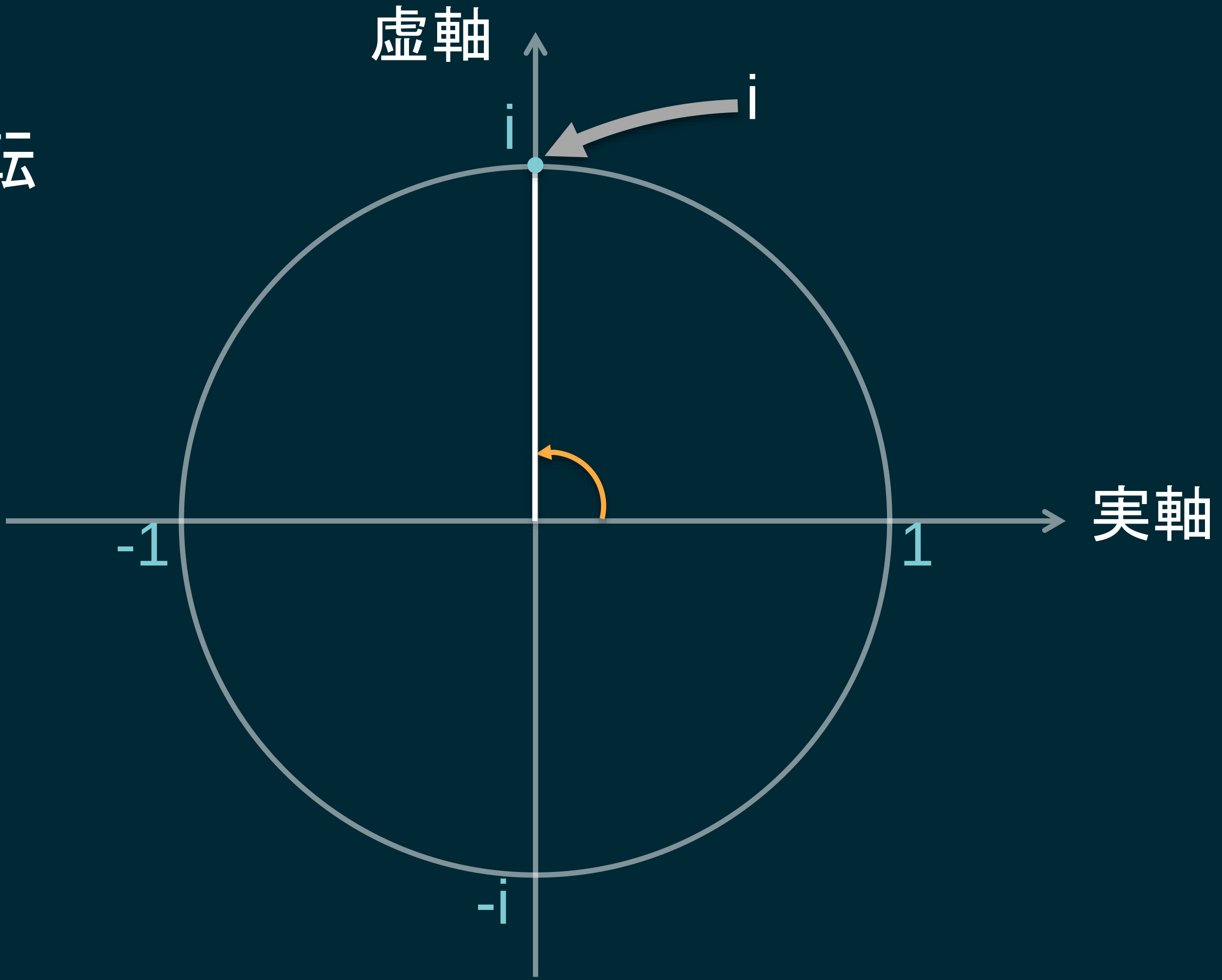

虚軸
i
実軸
-1
1
もしくは
-90°回転
-i
270°と-90°は
同じ意味

もっと重大な事実

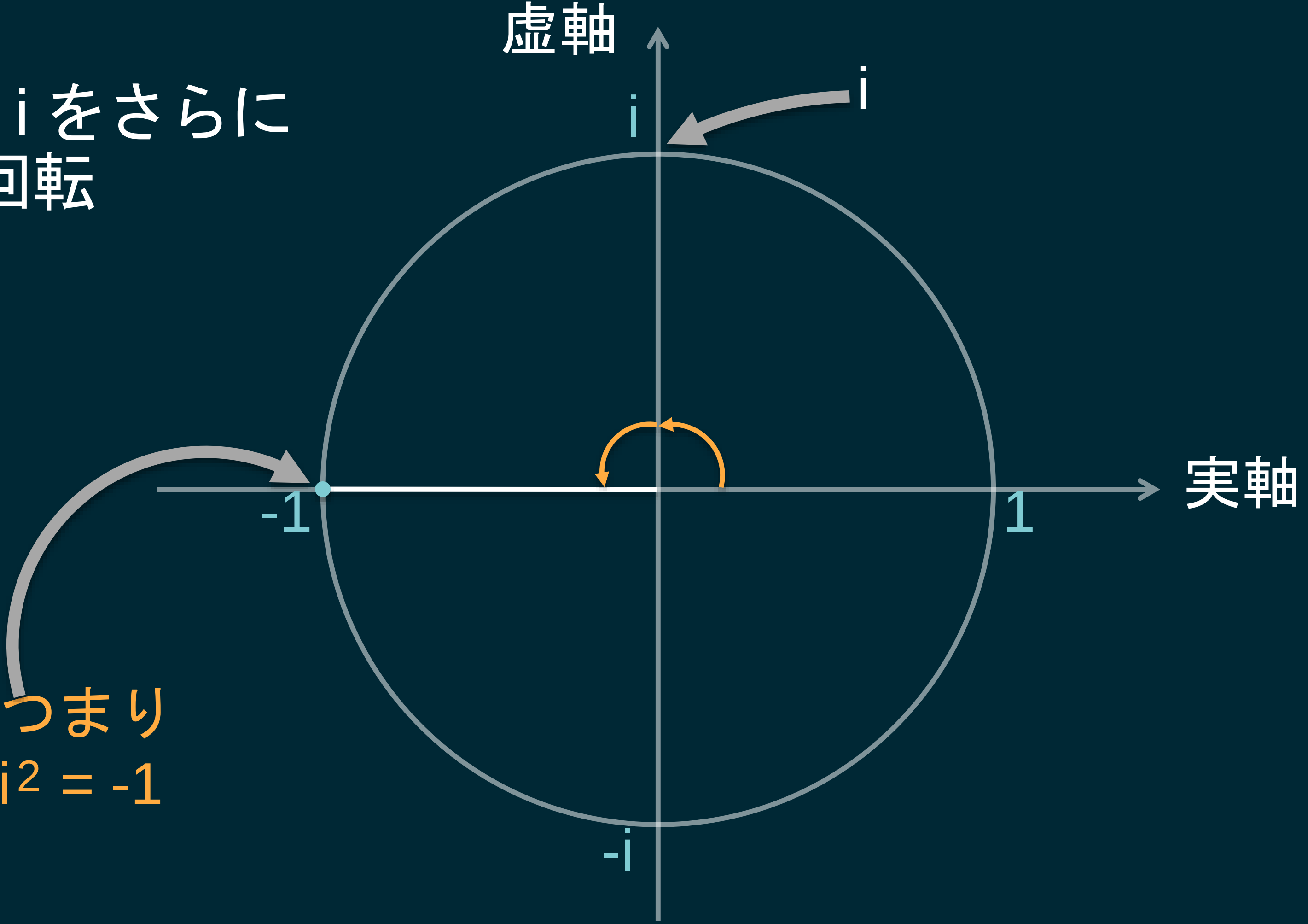
点と点を掛けると回転する

虚数の性質 $i^2 = -1$

i は90°回転

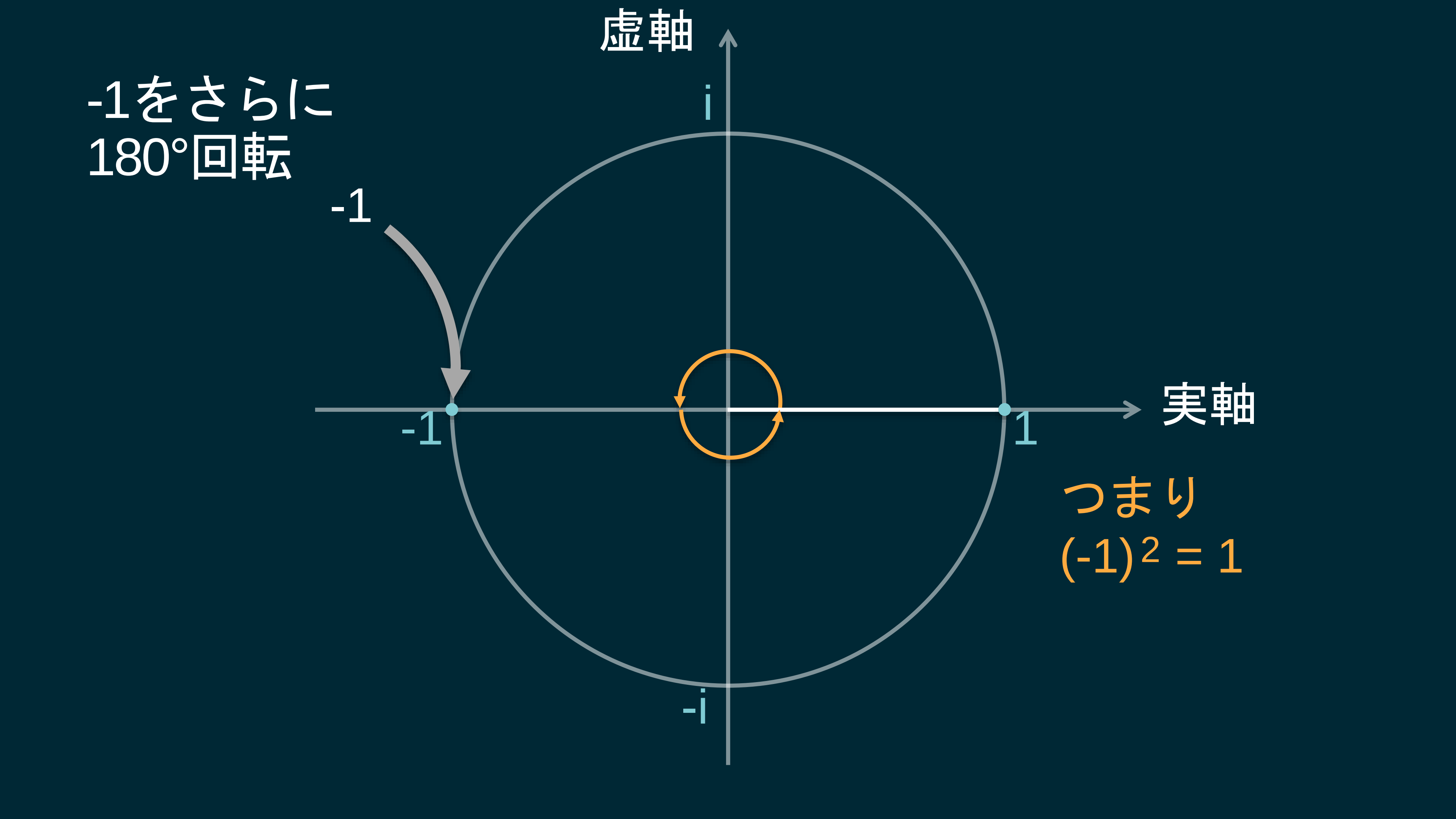


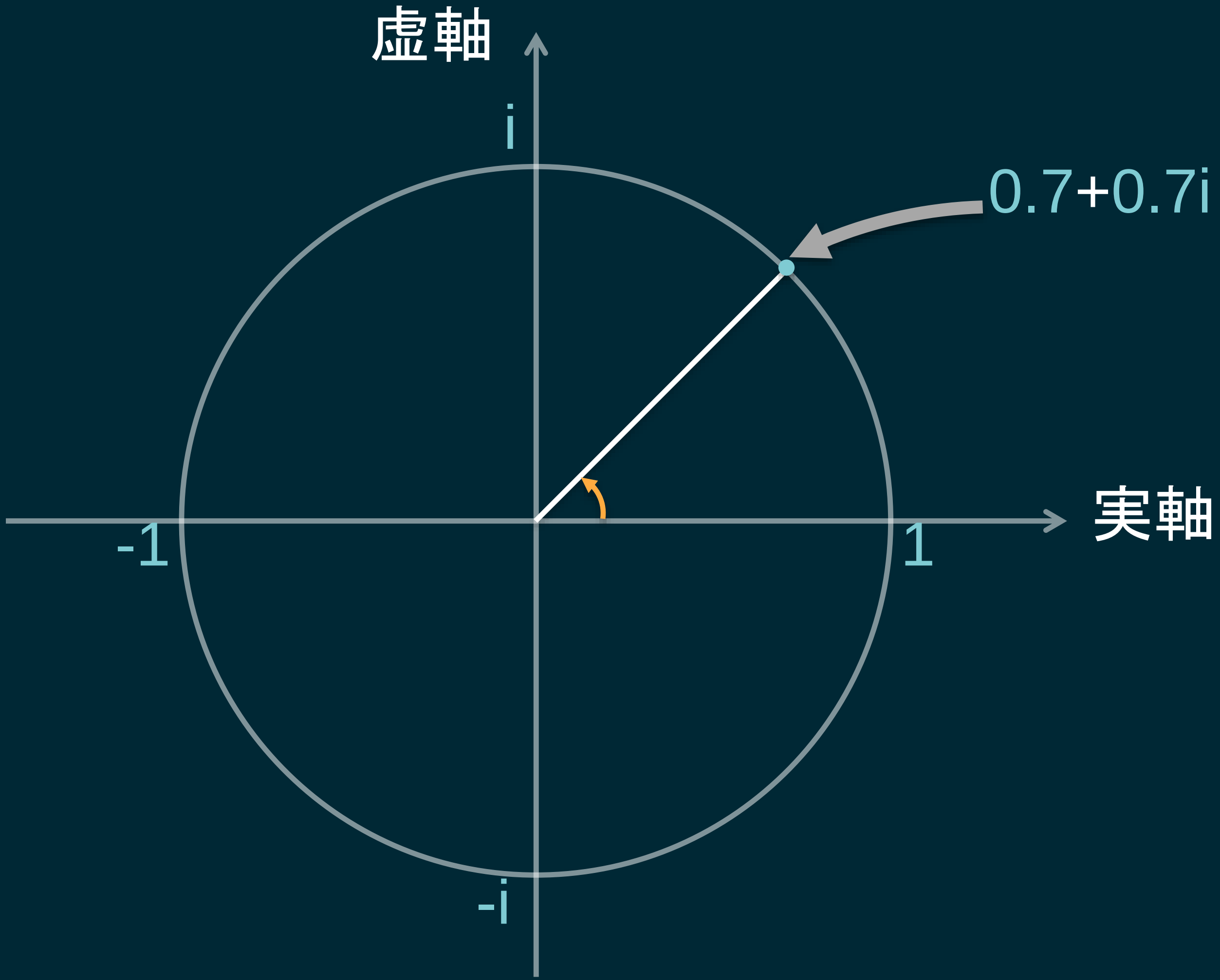
i^2 はiをさらに
90°回転



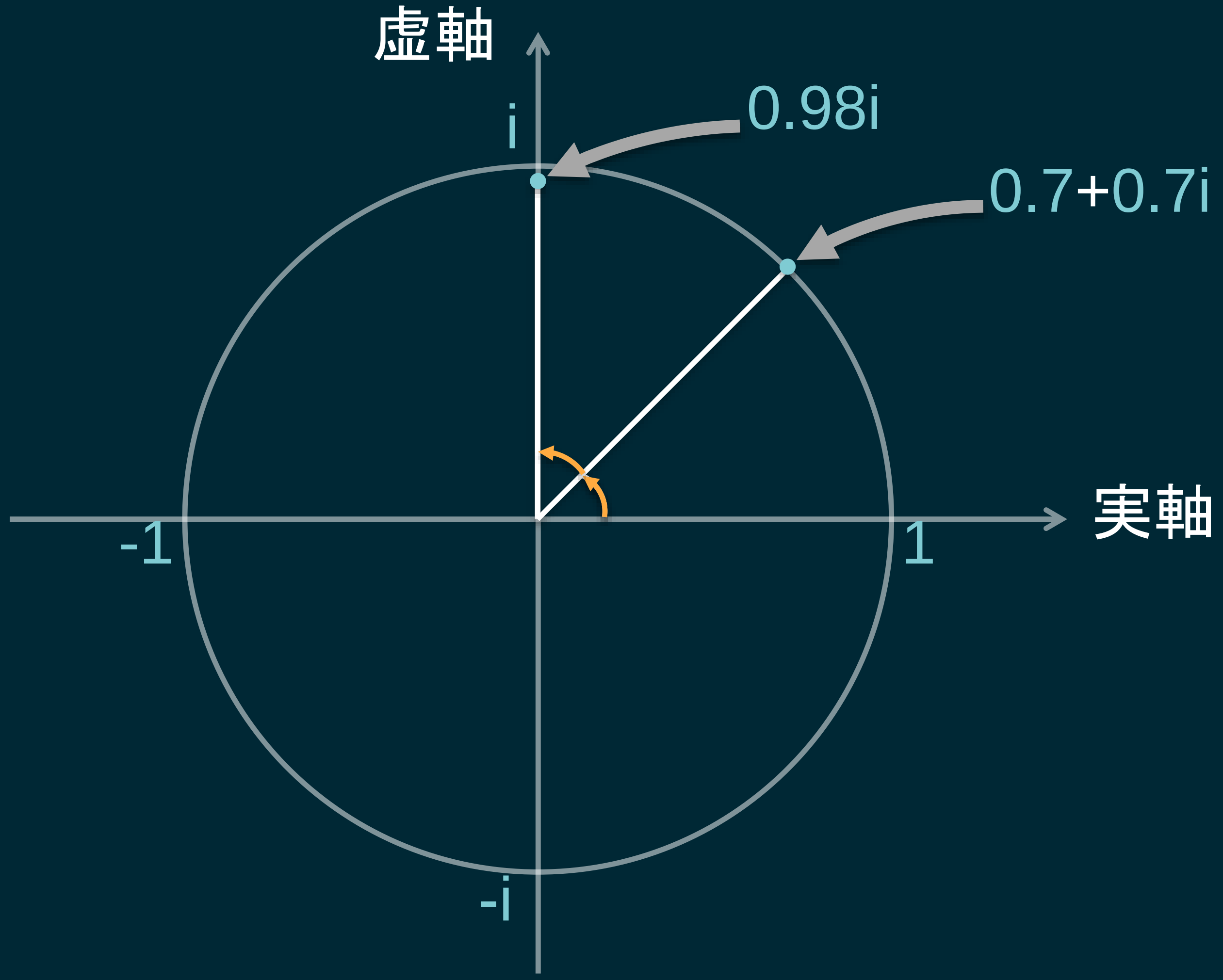
つまり
$i^2 = -1$

$$(-1)^2 = 1$$

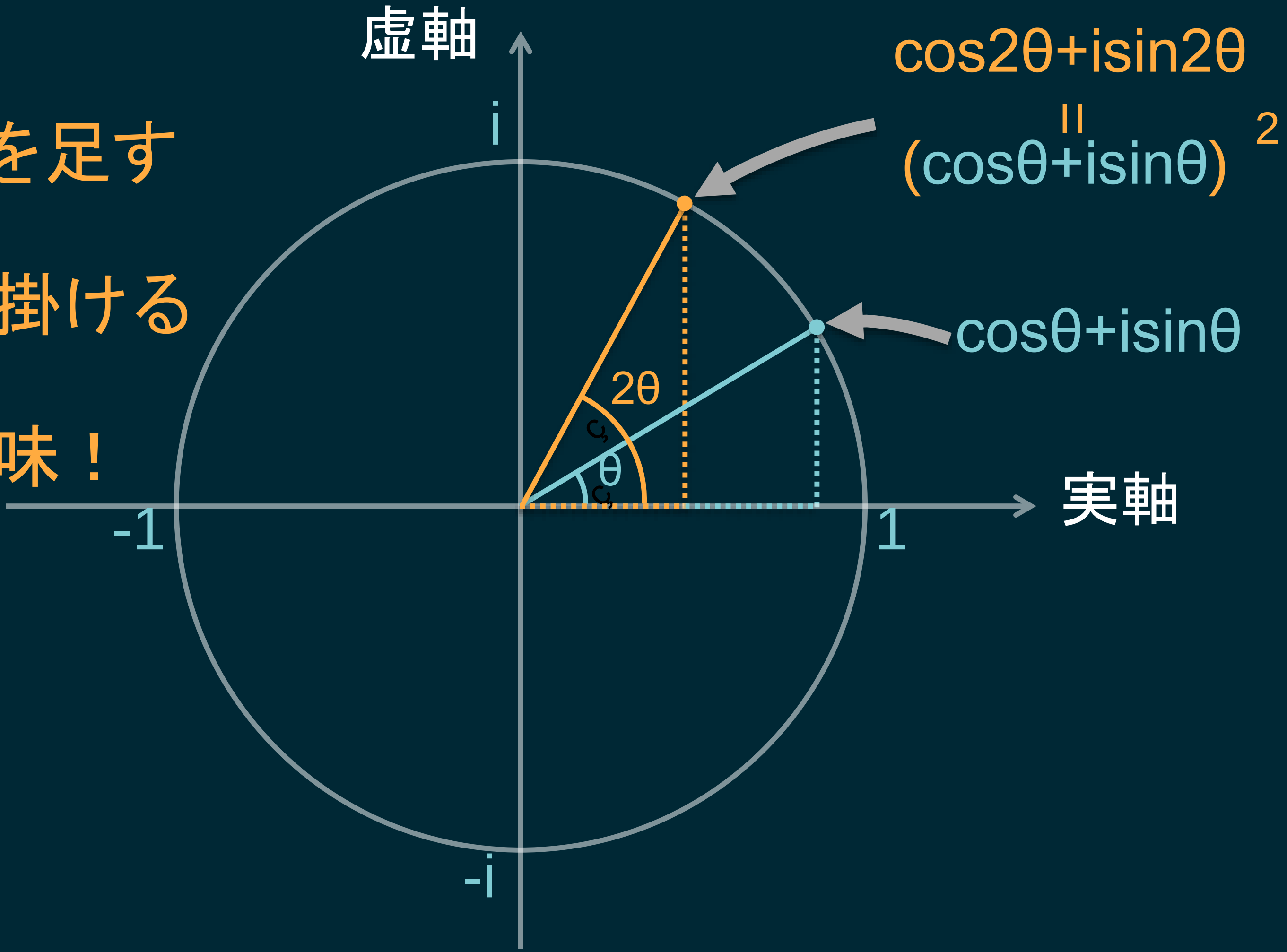

-1をさらに
180°回転
-1
虚軸
i
実軸
-1
1
-i
つまり
(-1)2 = 1


虚軸
i
0.7+0.7i
実軸
-1
1
-i

$$
\begin{aligned}
&(0.7 + 0.7i)^2 \\
&= (0.7 + 0.7i)(0.7 + 0.7i) \\
&= 0.7 \times 0.7 + 0.7i \times 0.7 + 0.7 \times 0.7i + 0.7i \times 0.7i \\
&= 0.49 + 0.49i + 0.49i - 0.49 \\
&= 0.98i
\end{aligned}
$$


虚軸
i
0.98i
0.7+0.7i
-1
1
実軸
-i

回転角を足す
と
複素数を掛ける
は
同じ意味！



ところで…

$$x^3 = 1$$

この方程式を解いてみよう

$$x^3 = 1$$

$$x^3 = 1$$

$$x^3 - 1 = 0$$

$$x^3 = 1$$

$$x^3 - 1 = 0$$

$$(x - 1)(x^2 + x + 1) = 0$$

$$x^3 = 1$$

$$x^3 - 1 = 0$$

$$(x-1)(x^2+x+1) = 0$$

$$(x-1)(x+\frac{1-\sqrt{3}i}{2})(x+\frac{1+\sqrt{3}i}{2}) = 0$$

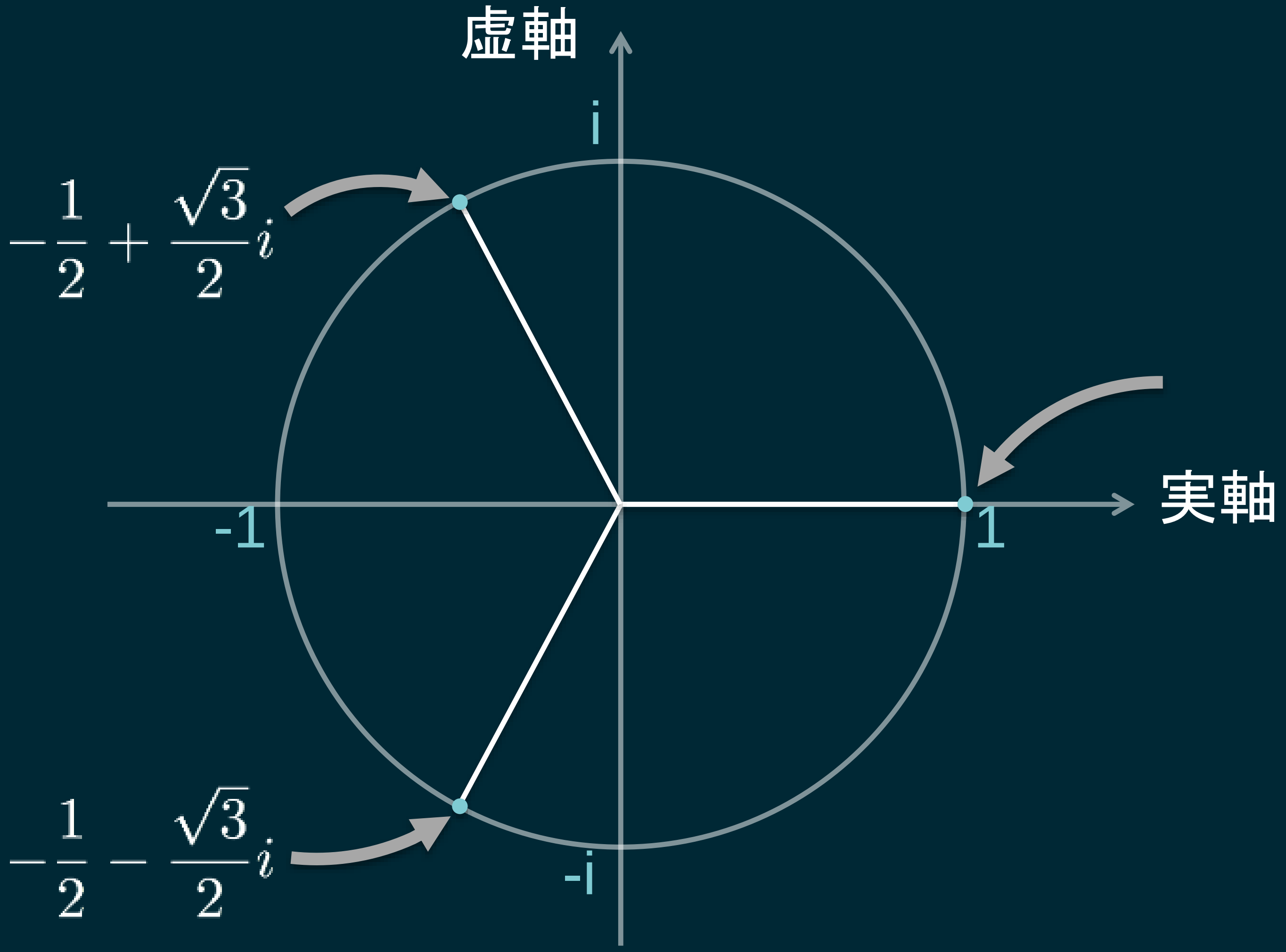
$$x^3 = 1$$

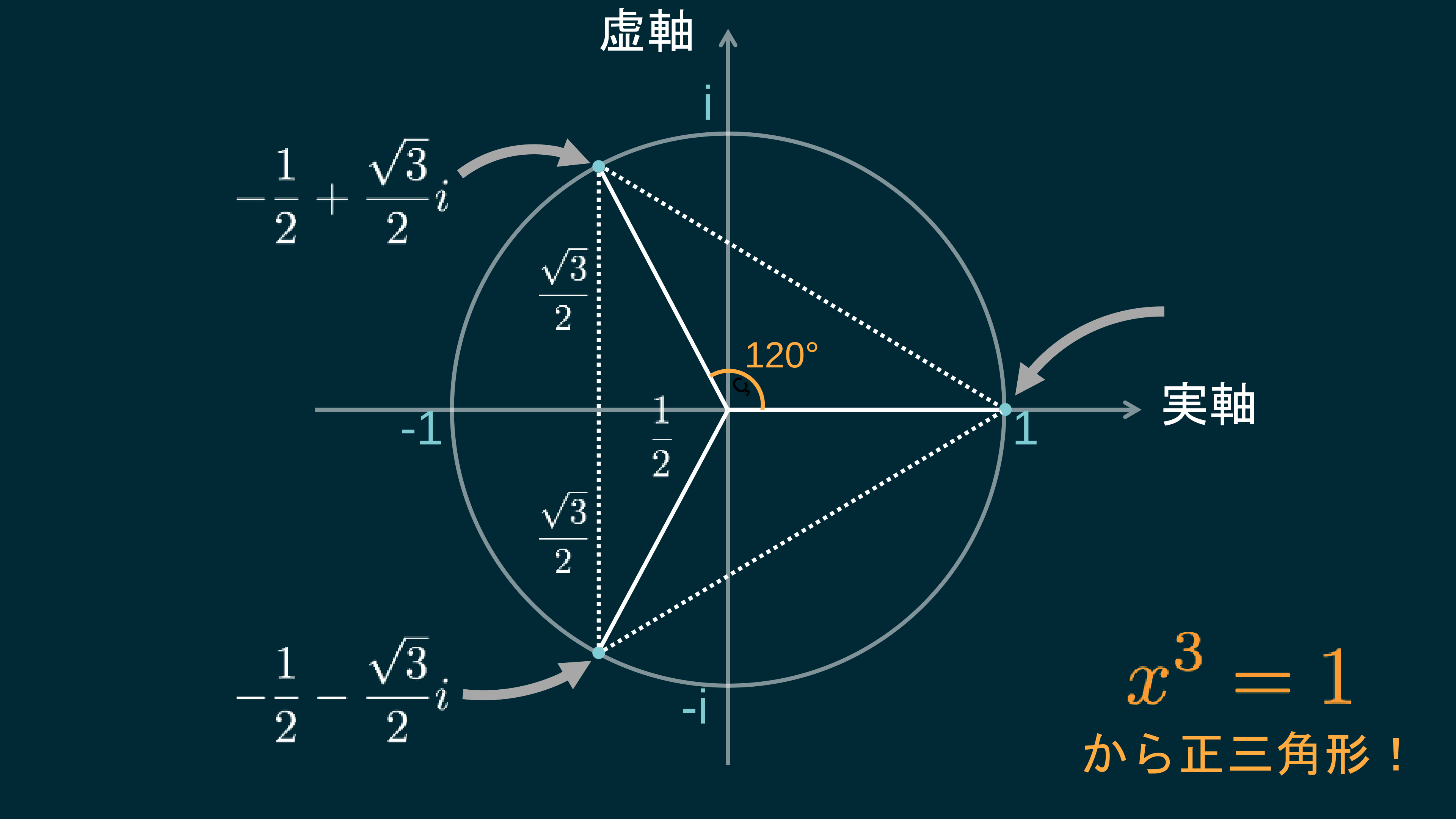
$$x^3 - 1 = 0$$

$$(x-1)(x^2+x+1) = 0$$

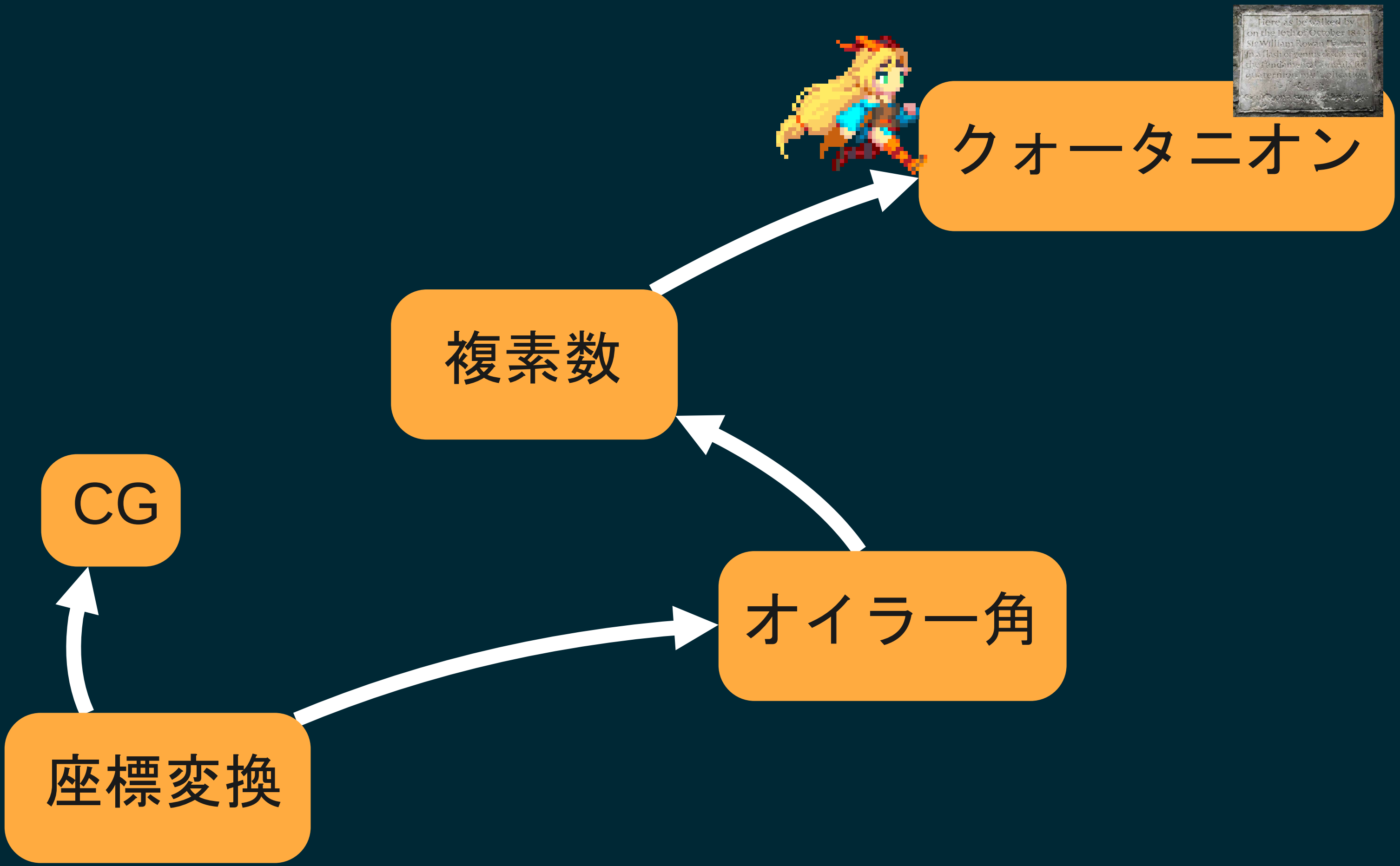
$$(x-1)\left(x+\frac{1-\sqrt{3}i}{2}\right)\left(x+\frac{1+\sqrt{3}i}{2}\right) = 0$$

$$x = \quad 1, \quad -\frac{1}{2}+\frac{\sqrt{3}}{2}i, \quad -\frac{1}{2}-\frac{\sqrt{3}}{2}i$$


虚軸
i
$-\frac{1}{2}+\frac{\sqrt{3}}{2}i$
-1
1
実軸
$-\frac{1}{2}-\frac{\sqrt{3}}{2}i$
-i



虚軸
i
$-\frac{1}{2}+\frac{\sqrt{3}}{2}i$
$\frac{\sqrt{3}}{2}$
120°
実軸
-1
$\frac{1}{2}$
1
$\frac{\sqrt{3}}{2}$
$-\frac{1}{2}-\frac{\sqrt{3}}{2}i$
-i
$x^3=1$
から正三角形！


クォータニオン
Here as he walked by
on the 16th of October 1843
Sir William Rowan
複素数
CG
オイラー角
座標変換

クォータニオン


Here as he walked by
on the 16th of October 1843
Sir William Rowan
in a flash of genius discovered
the fundamental formula for
quaternion multiplication
$i^2 = j^2 = k^2 = ijk = -1$
on a stone

考案者はハミルトン
（William Rowan Hamilton　1805-1865)

「複素平面の三次元版は
作れないものか…」

虚軸を３つにすればうまくいく！

複素数

i が虚数単位

$i^2=-1$

クォータニオン

i, j, kが虚数単位

$i^2=-1$	$j^2=-1$	$k^2=-1$
$ij=k$	$jk=i$	$ki=j$
$ji=-k$	$kj=-i$	$ik=-j$

驚愕のアイデア！

クォータニオンは複素数の三次元版

複素数		クォータニオン
a+ib	→	a+ib+jc+kd

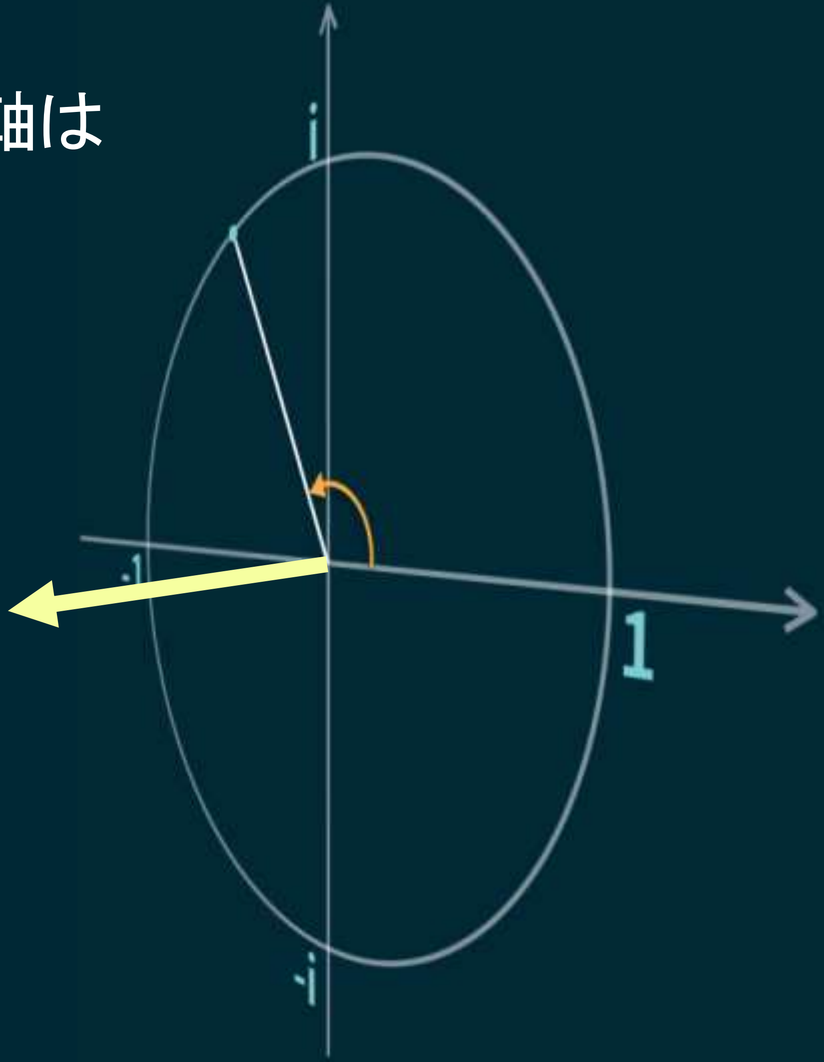
複素数の要素はふたつ
クォータニオンの要素は４つ

クォータニオン要素を
x, y, z, w とする

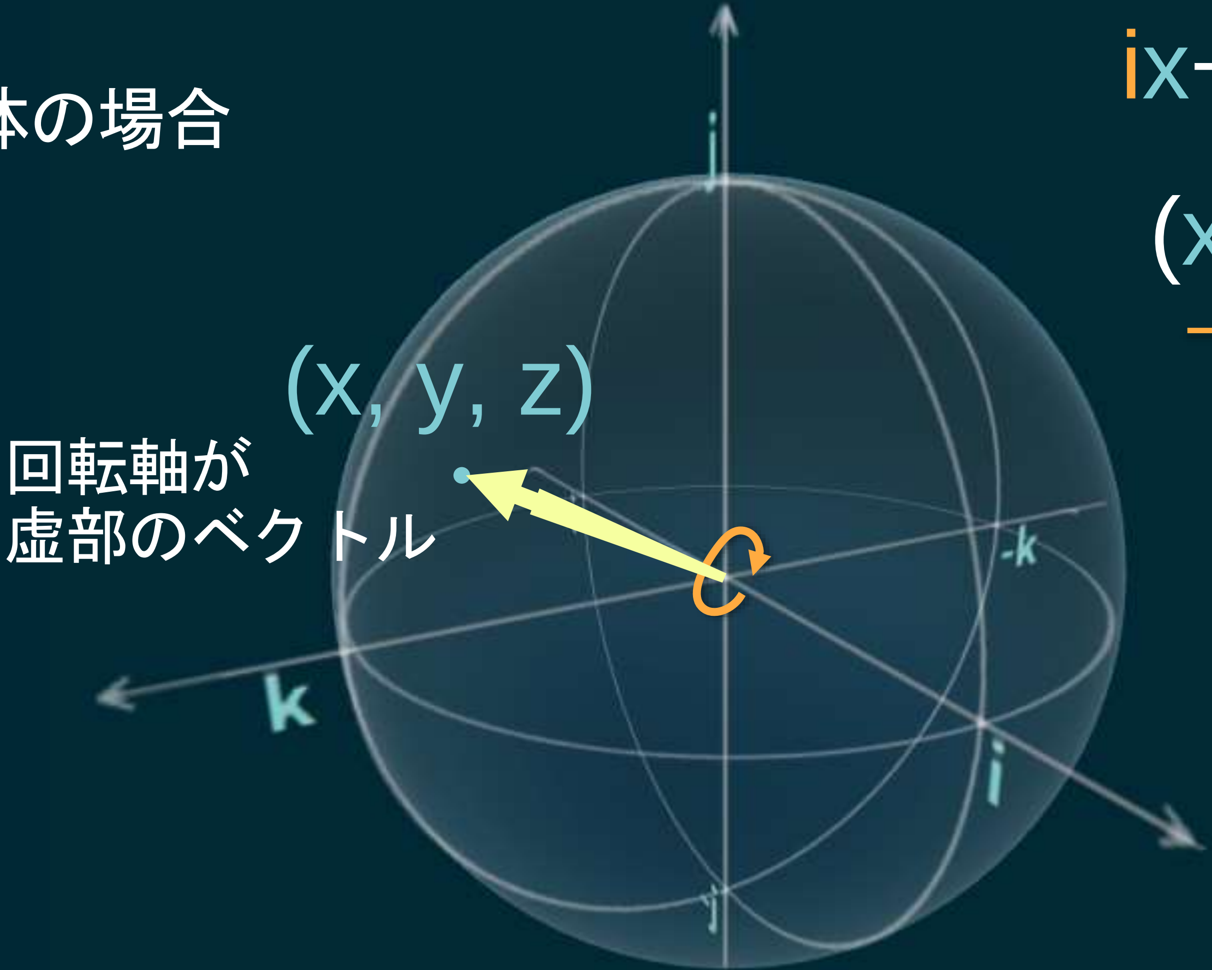
$$ix+jy+kz+w$$

$$(\underbrace{x, y, z}_{\text{虚部}}, \underbrace{w}_{\text{実部}})$$

複素平面の回転軸は
固定だった



立体の場合



ix+jy+kz+w

(x, y, z, w)

虚部　実部

複素数の回転とクォータニオンの回転

複素数

$\cos\theta + i\sin\theta$

クォータニオン

$\cos\frac{\theta}{2} + n\sin\frac{\theta}{2}$

$n = in_x + jn_y + kn_z$

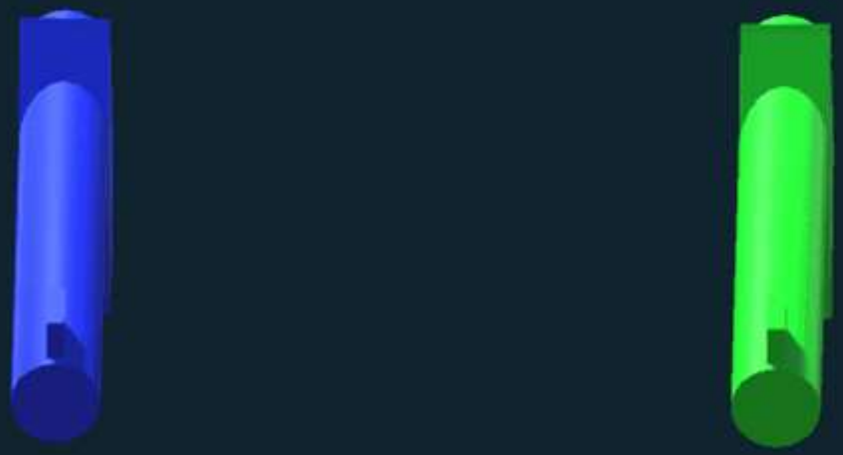
$n(n_x, n_y, n_z)$は回転軸のベクトル

重大な事実

あらゆる回転は
ひとつの軸回転で表現できる

そしてクォータニオンは
軸回転を表現する

あらゆる回転は
ひとつの軸回転で表現できる



クォータニオンなら
一発で決まる！

クォータニオンのテクニック

便利なテクニック
Slerp (Spherical Linear Interpolation):球面線形補間

```
transform.rotation =
Quaternion.Slerp(transform.rotation,
                target_rotation,
                0.1f);
```

現在の値

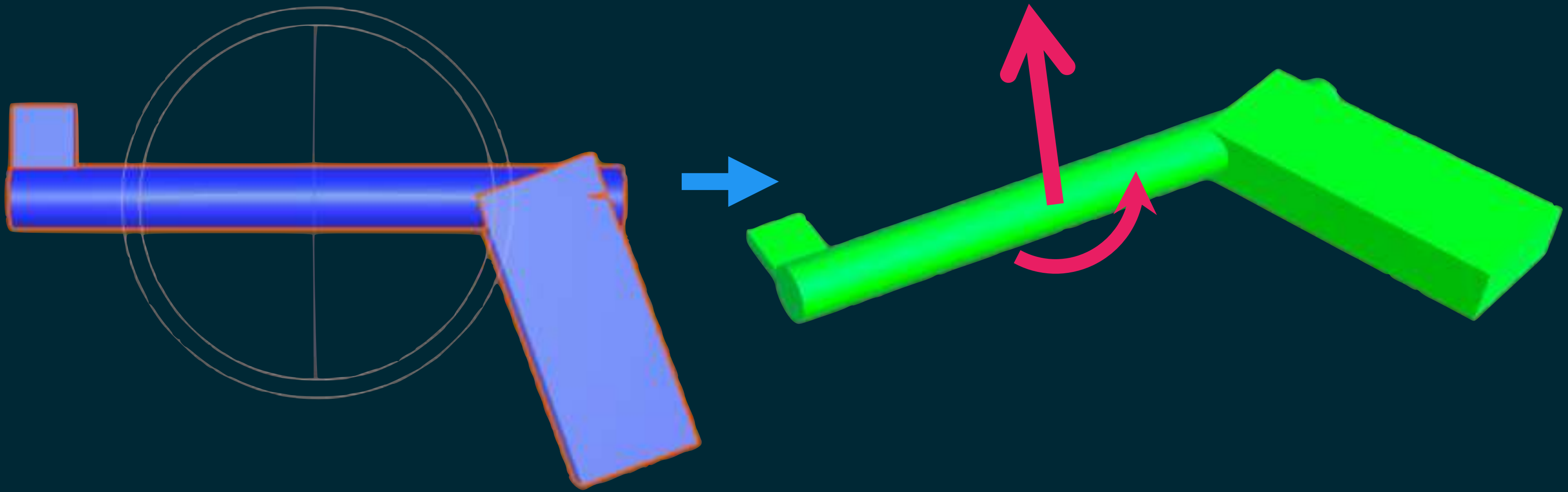
目標の値

適当な率

凝ったテクニック
バネトルク

目標との差分に比例したトルクをかける

基本戦略：
目標への差分を求めて、トルクに変換する



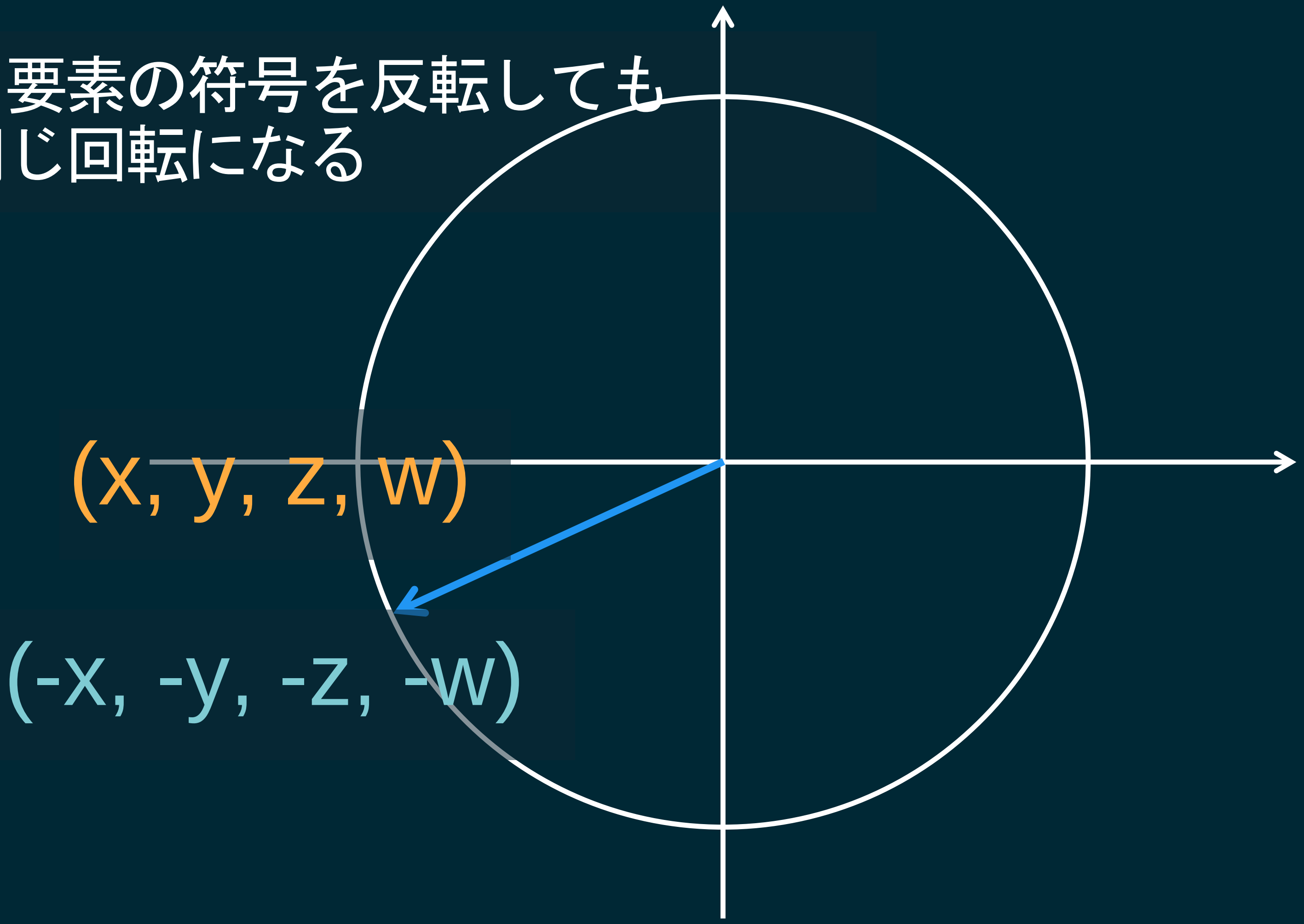
差分クォータニオンの求めかた

差分=目標の値×現在の値$^{-1}$

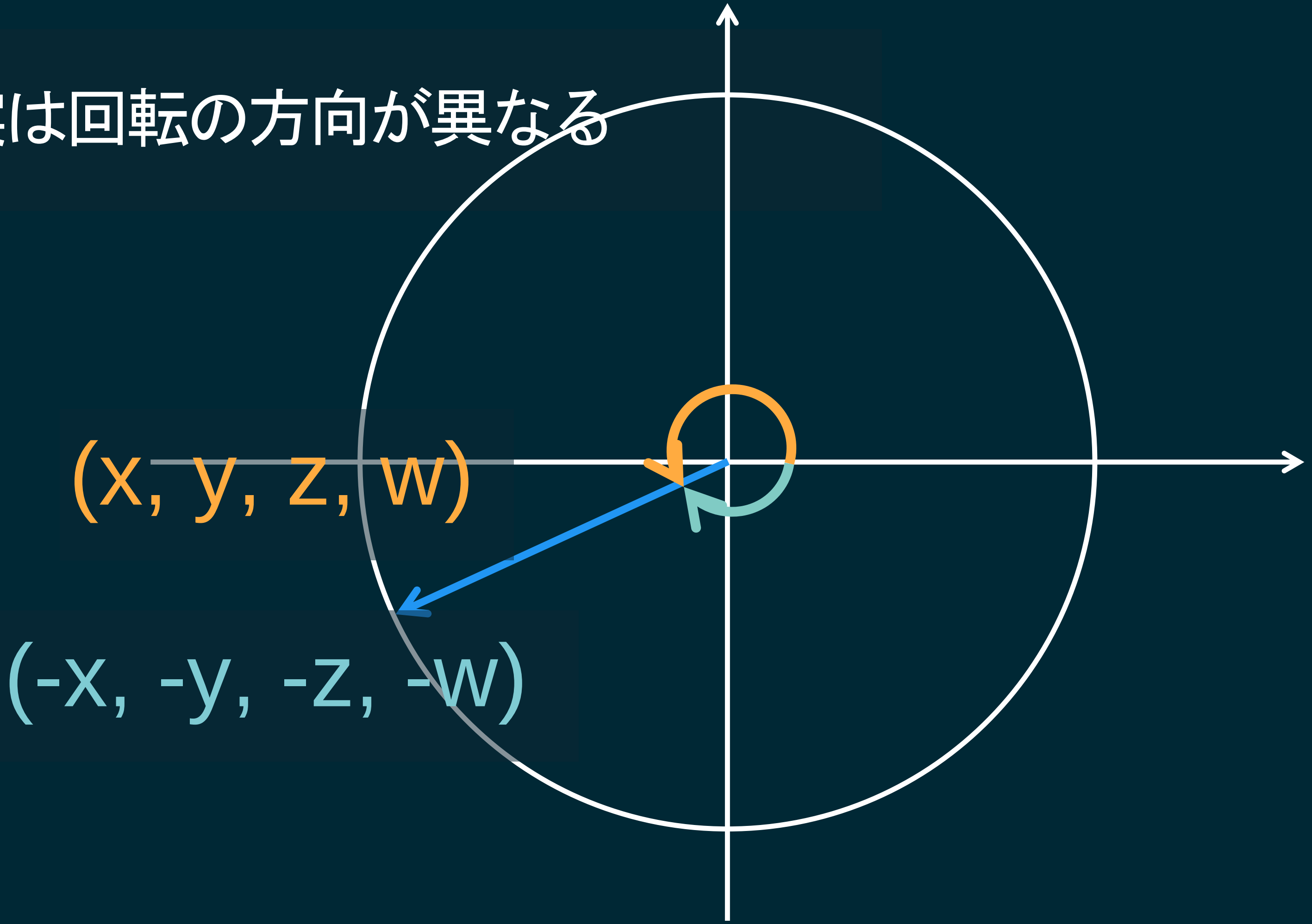
実演
バネトルク実装例

```
void FixedUpdate ()
{
    var rb = GetComponent<Rigidbody>();
    var target_pos = target_go_.transform.position;
    var diff = target_pos - transform.position;
    rb.AddForce(diff*10f);
    var target_rot = Quaternion.LookRotation(diff);
    var rot = target_rot * Quaternion.Inverse(transform.rotation);
    rb.AddTorque(new Vector3(rot.x, rot.y, rot.z)*40f);
}
```

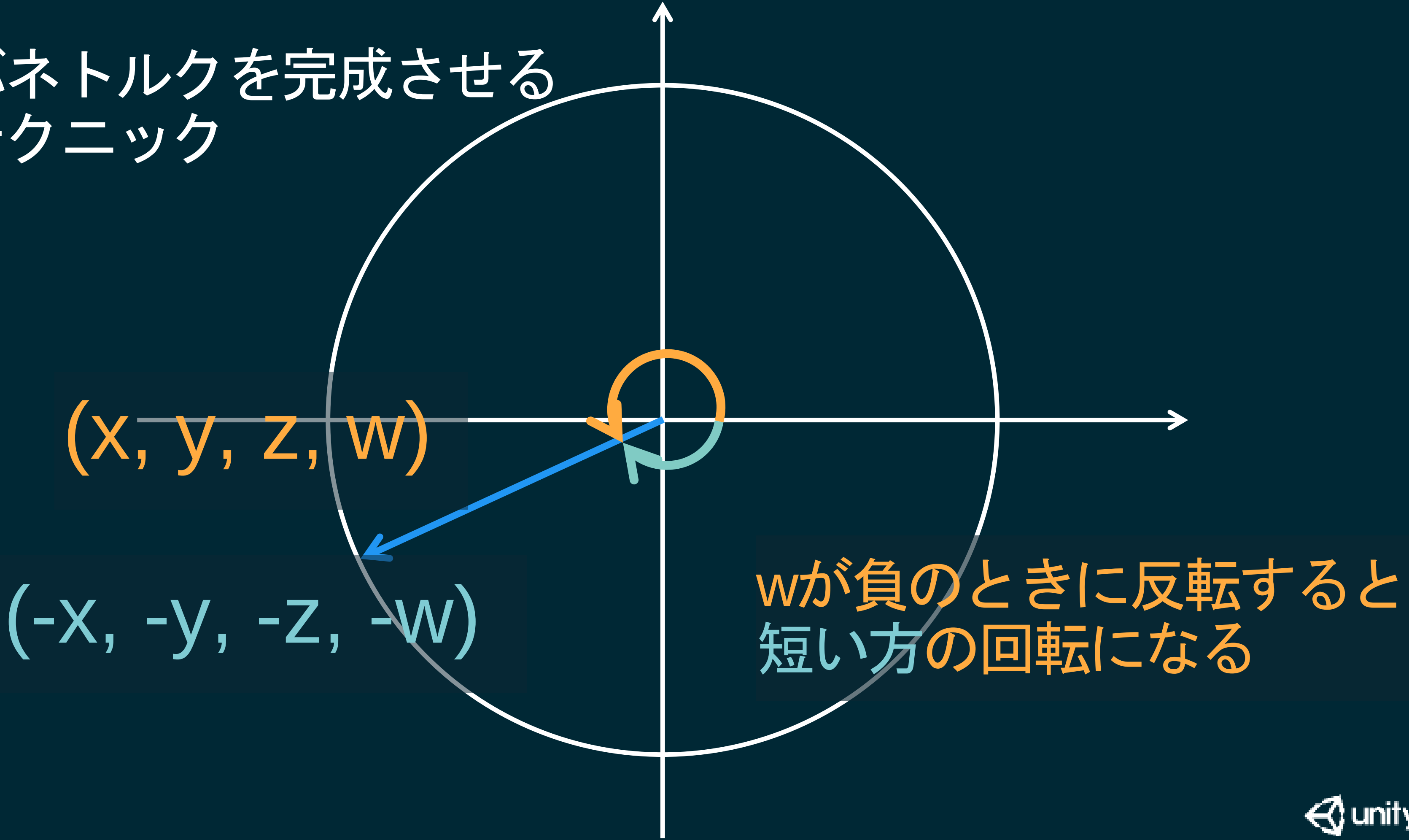
４要素の符号を反転しても同じ回転になる



実は回転の方向が異なる



バネトルクを完成させるテクニック



修正済みコード

```
void FixedUpdate ()
{
    var rb = GetComponent<Rigidbody>();
    var target_pos = target_go_.transform.position;
    var diff = target_pos - transform.position;
    rb.AddForce(diff*10f);
    var target_rot = Quaternion.LookRotation(diff);
    var rot = target_rot * Quaternion.Inverse(transform.rotation);
    if (rot.w < 0f) {
        rot.x = -rot.x;
        rot.y = -rot.y;
        rot.z = -rot.z;
        rot.w = -rot.w;
    }
    rb.AddTorque(new Vector3(rot.x, rot.y, rot.z)*40f);
}
```

追加

水平を維持しないテクニック

```
var rot = Quaternion.LookRotation(diff);
```

第二引数に Vector.up が省略されている

↓

```
var up = transform.TransformVector(Vector3.up);
var rot = Quaternion.LookRotation(diff, up);
```

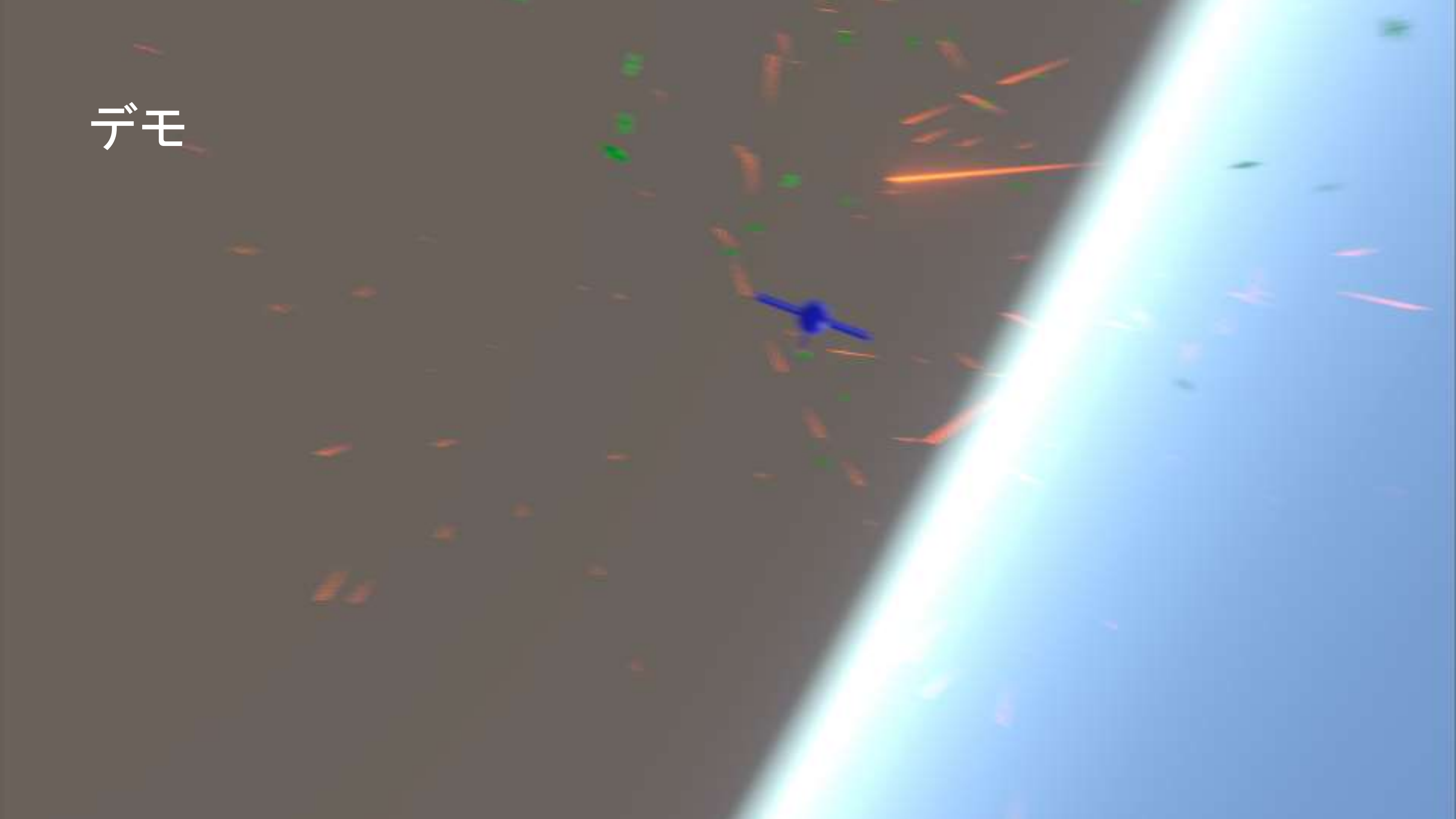
自分の姿勢から up ベクトルを作る

修正済みコードその２

```
void FixedUpdate ()
{
    var rb = GetComponent<Rigidbody>();
    var target_pos = target_go_.transform.position;
    var diff = target_pos - transform.position;
    rb.AddForce(diff*10f);
    var target_rot = Quaternion.LookRotation(diff, transform.TransformVector(Vector3.up));
    var rot = target_rot * Quaternion.Inverse(transform.rotation);
    if (rot.w < 0f) {
        rot.x = -rot.x;
        rot.y = -rot.y;
        rot.z = -rot.z;
        rot.w = -rot.w;
    }
    rb.AddTorque(new Vector3(rot.x, rot.y, rot.z)*40f);
}
```

追加

デモ



このあと学習を続けるなら...

- トルクや物理を詳しく知りたい→Unity道場札幌講演をぜひ！ https://www.youtube.com/watch?v=FqjM9oujyNE&feature=youtu.be
- クォータニオンの応用例を知りたい→Unite2017Tokyoをぜひ！ https://www.youtube.com/watch?v=6EtTI5xC524 27分あたりから
- なんで $\cos\frac{\theta}{2}+n\sin\frac{\theta}{2}$ と、$\frac{\theta}{2}$ になるのか気になる→ ブログ「クォータニオンで回転を表現する定義にθ/2が使用される理由」をぜひ！ http://qiita.com/yuji_yasuhara/items/a5b7c489e1d521adbd72

参考：Inverse自前実装

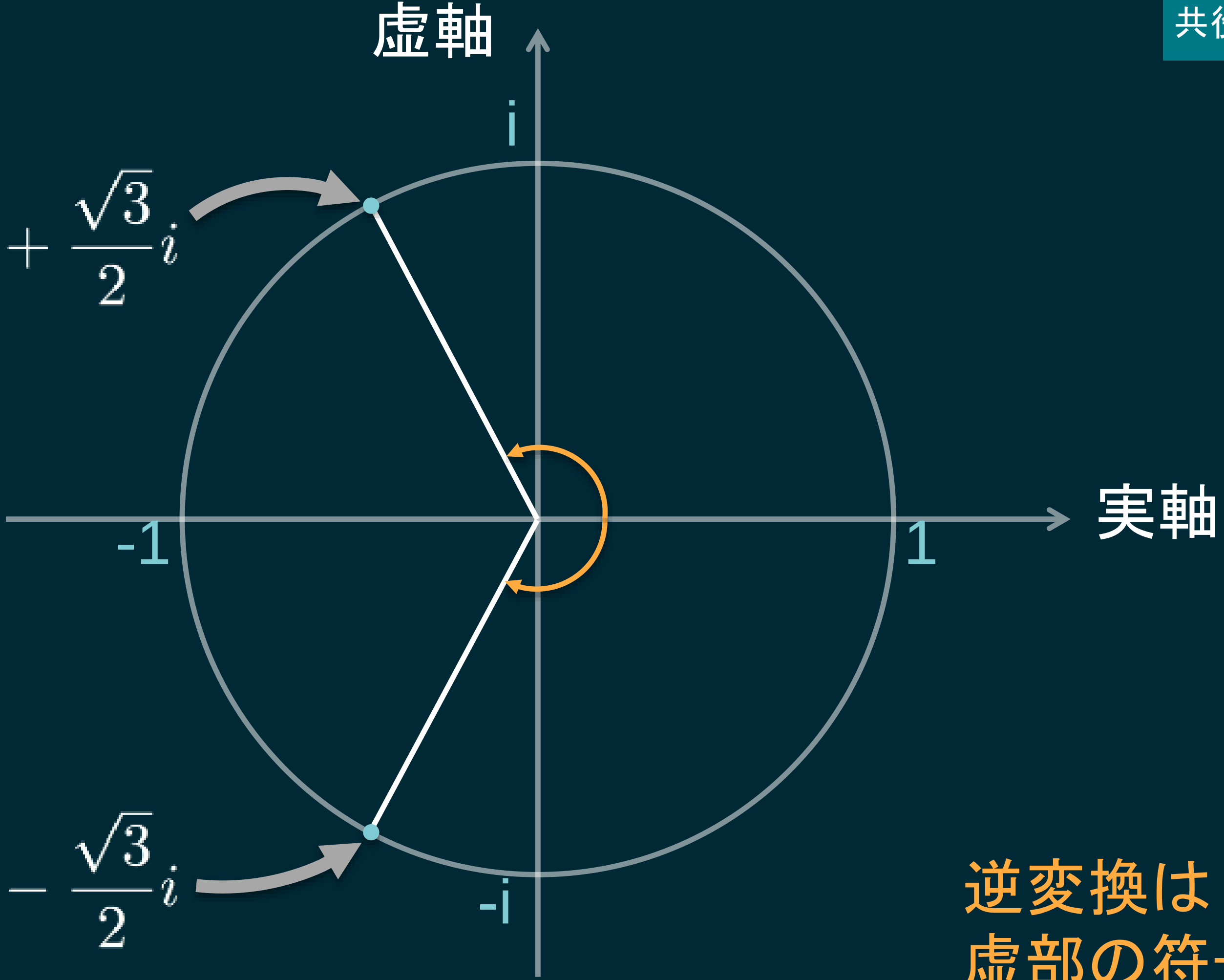
```
void MakeInverse(ref Quaternion rot)
{
    rot.x = -rot.x;
    rot.y = -rot.y;
    rot.z = -rot.z;
}
```

虚部を反転するだけで逆クォータニオン

共役複素数と呼ぶ

マメ知識

$$-\frac{1}{2}+\frac{\sqrt{3}}{2}i$$



$$-\frac{1}{2}-\frac{\sqrt{3}}{2}i$$

逆変換は
虚部の符号を反転

おしまい