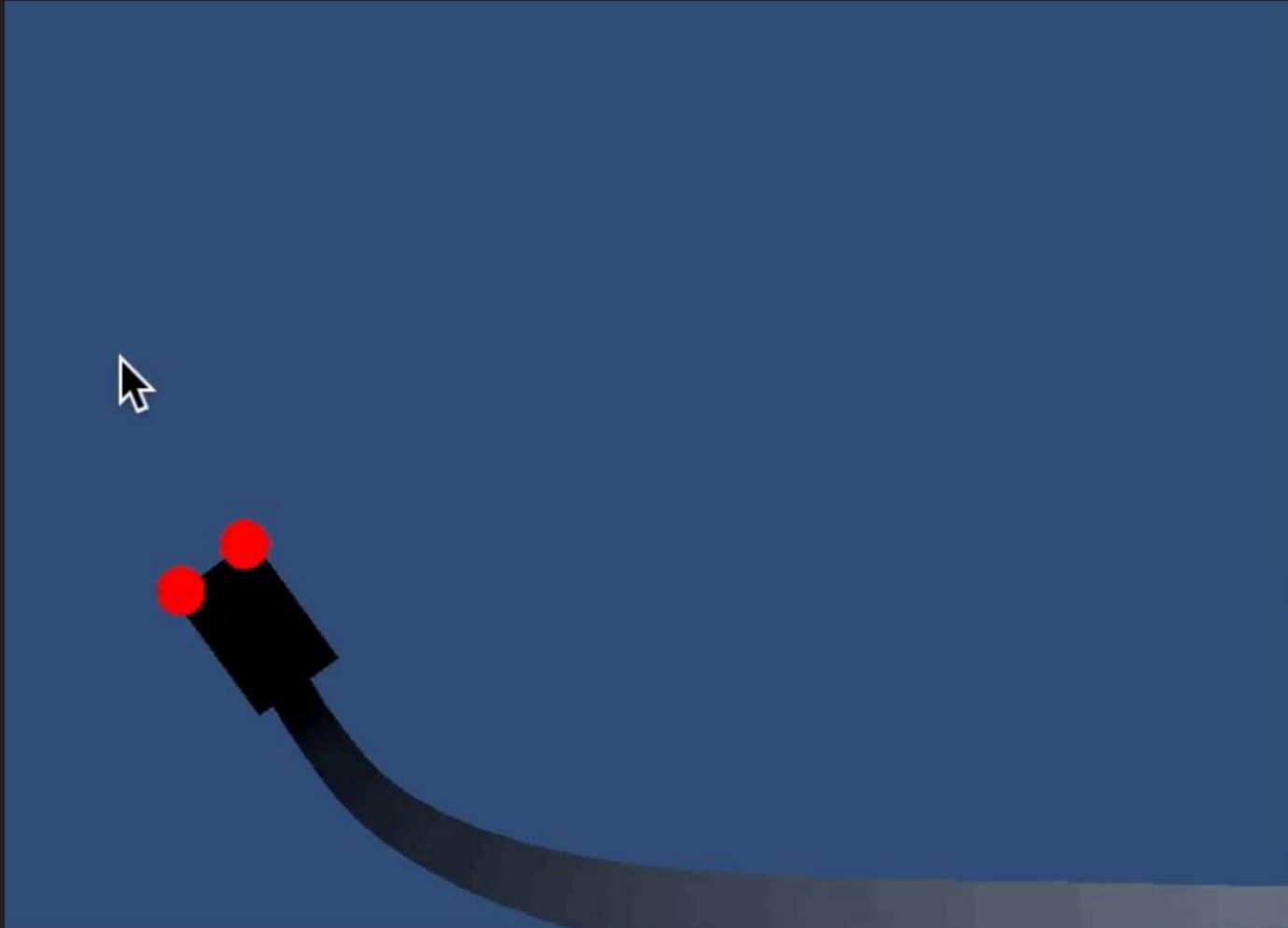


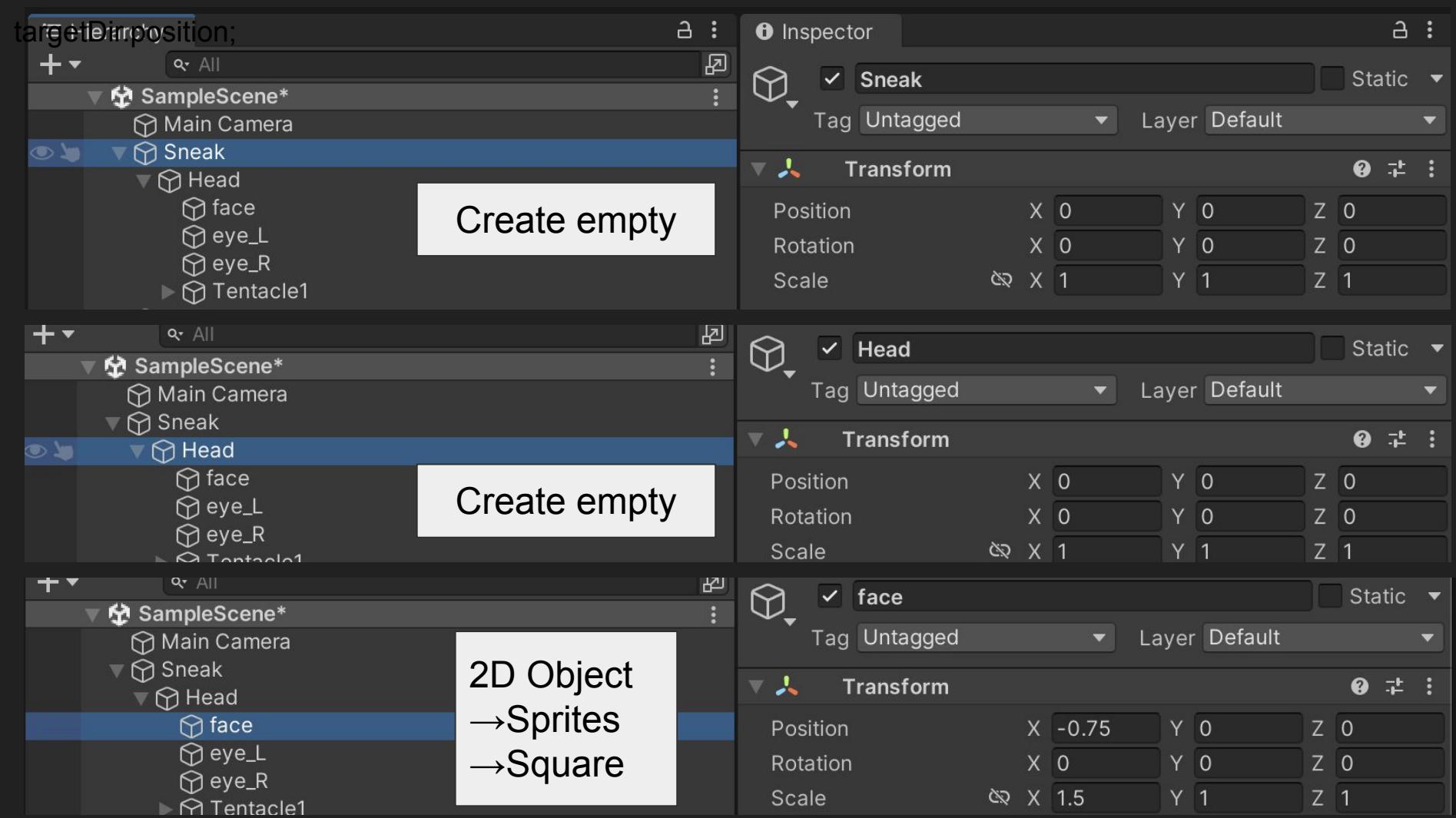
LineRenderer

2D Procedural Animation



2D Sprite で簡易的な顔の用意v



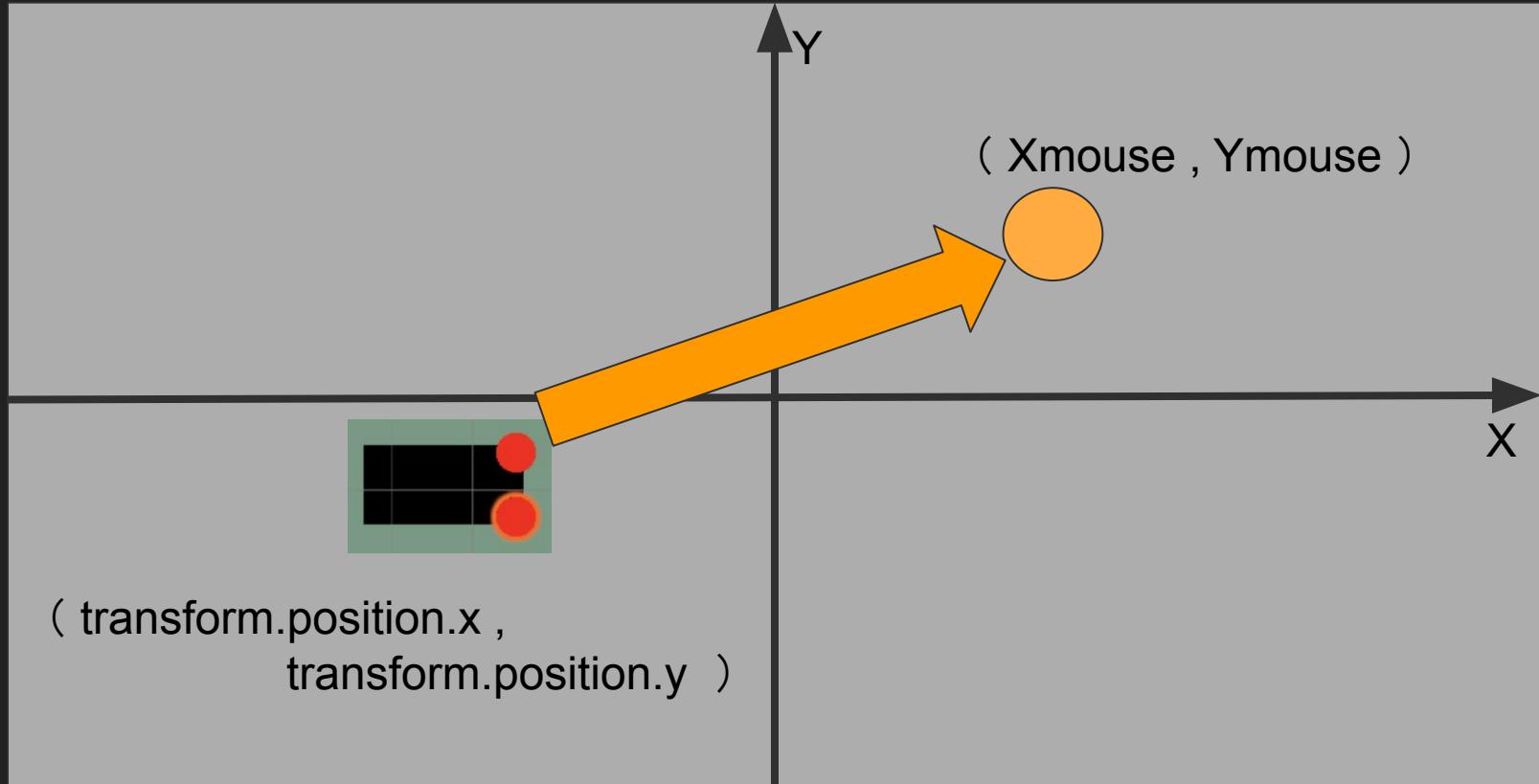


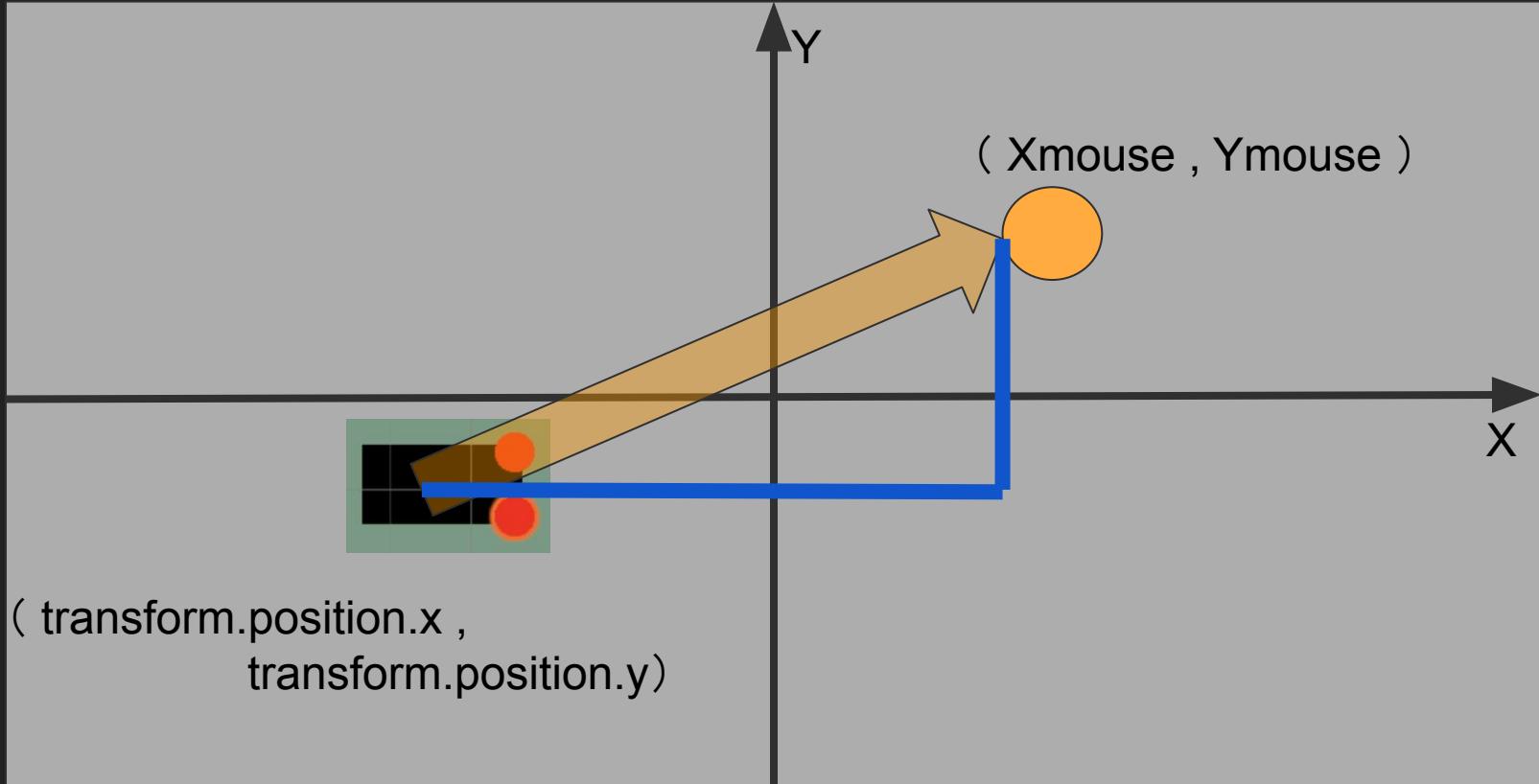
これがマウスカーソルの方を向くようにするスクリプト作成する

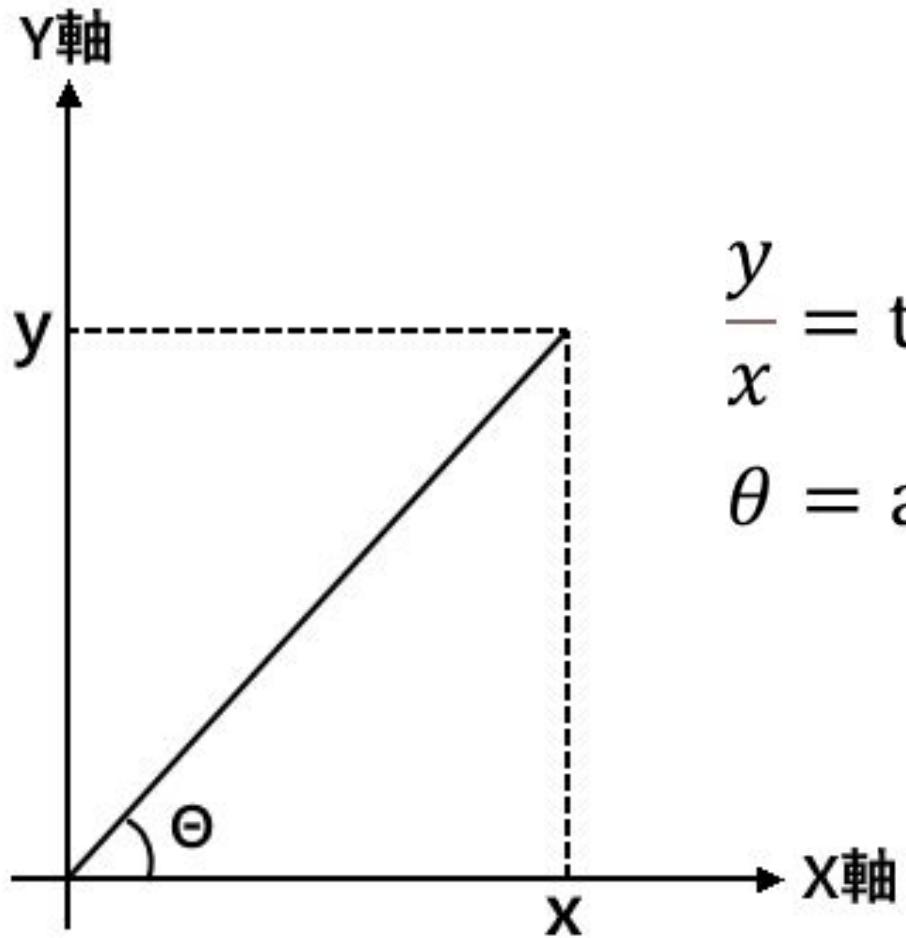
RotateToTarget.cs

```
public class RotateToTarget : MonoBehaviour
{
    public float rotationSpeed;
    private Vector2 direction;

    void Update()
    {
        //マウスの位置 - 現在地点
        direction = Camera.main.ScreenToWorldPoint(Input.mousePosition) - transform.position;
        //角度算出 ラジアンを度数法に変換
        float angle = Mathf.Atan2(direction.y, direction.x) * Mathf.Rad2Deg;
        //Quaternion型で 前方向(1,0,0) を angleの方向に向かせる
        Quaternion rotation = Quaternion.AngleAxis(angle, Vector3.forward);
        //1秒あたりrotationSpeedの回転をするように 緩やかに回転してもらう
        transform.rotation = Quaternion.Slerp(transform.rotation, rotation, rotationSpeed*Time.deltaTime);
    }
}
```



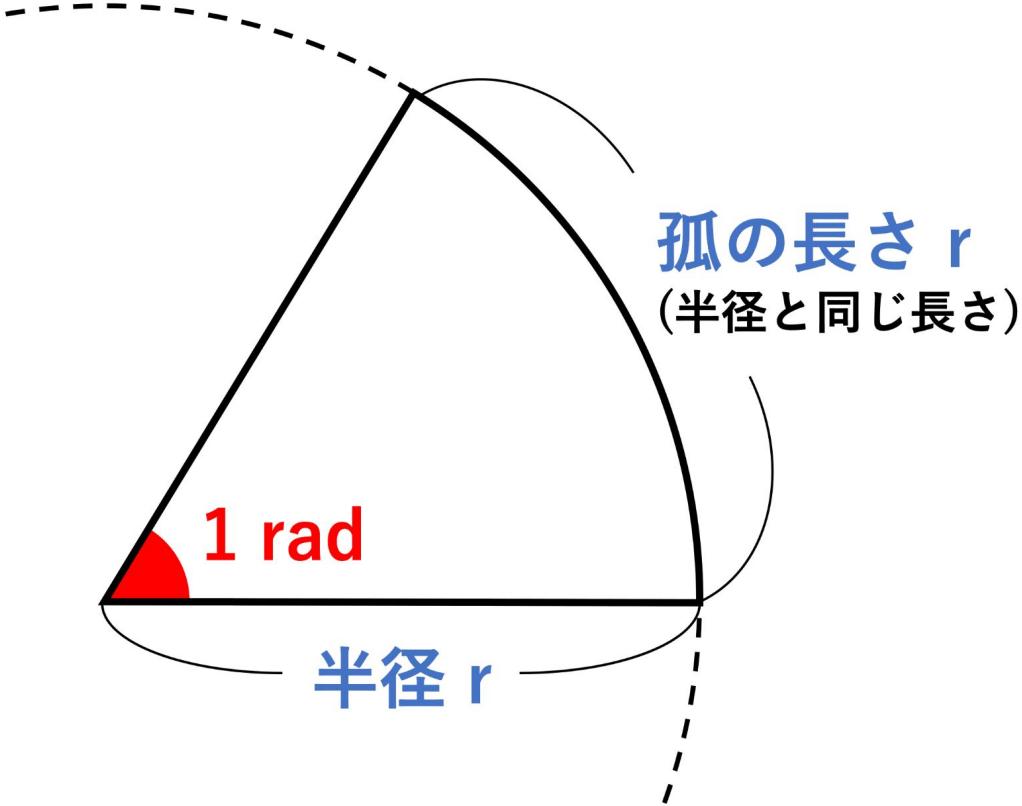




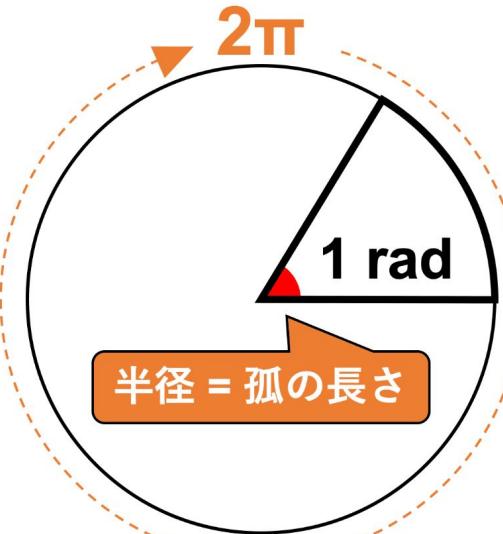
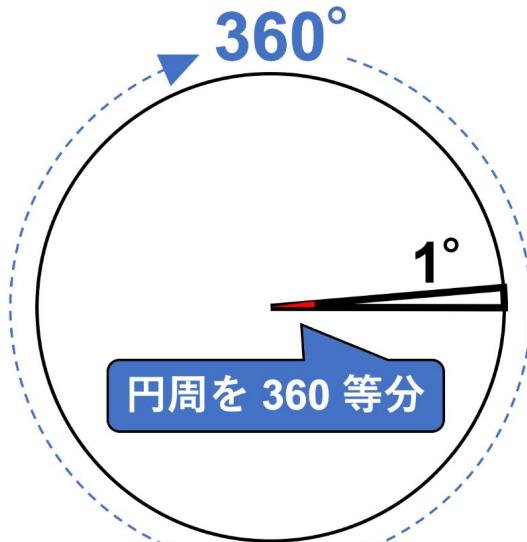
$$\frac{y}{x} = \tan \theta$$

$$\theta = \text{atan} \frac{y}{x} = \text{atan} 2(x, y)$$

ラジアン(弧度)とは



度数法と弧度法【定義】



度数法

360°

弧度法

$2\pi \text{ rad}$



Hierarchy Inspector

SampleScene*

Main Camera

Head

face

eye_L

eye_R

Static

Tag Untagged Layer Default

Transform

Position X -0.35100 Y 0.064263 Z 0.049816

Rotation X 0 Y 0 Z 0

Scale X 1 Y 1 Z 1

Rotate To Target (Script)

Script # RotateToTarget

Rotation Speed 25

Add Component

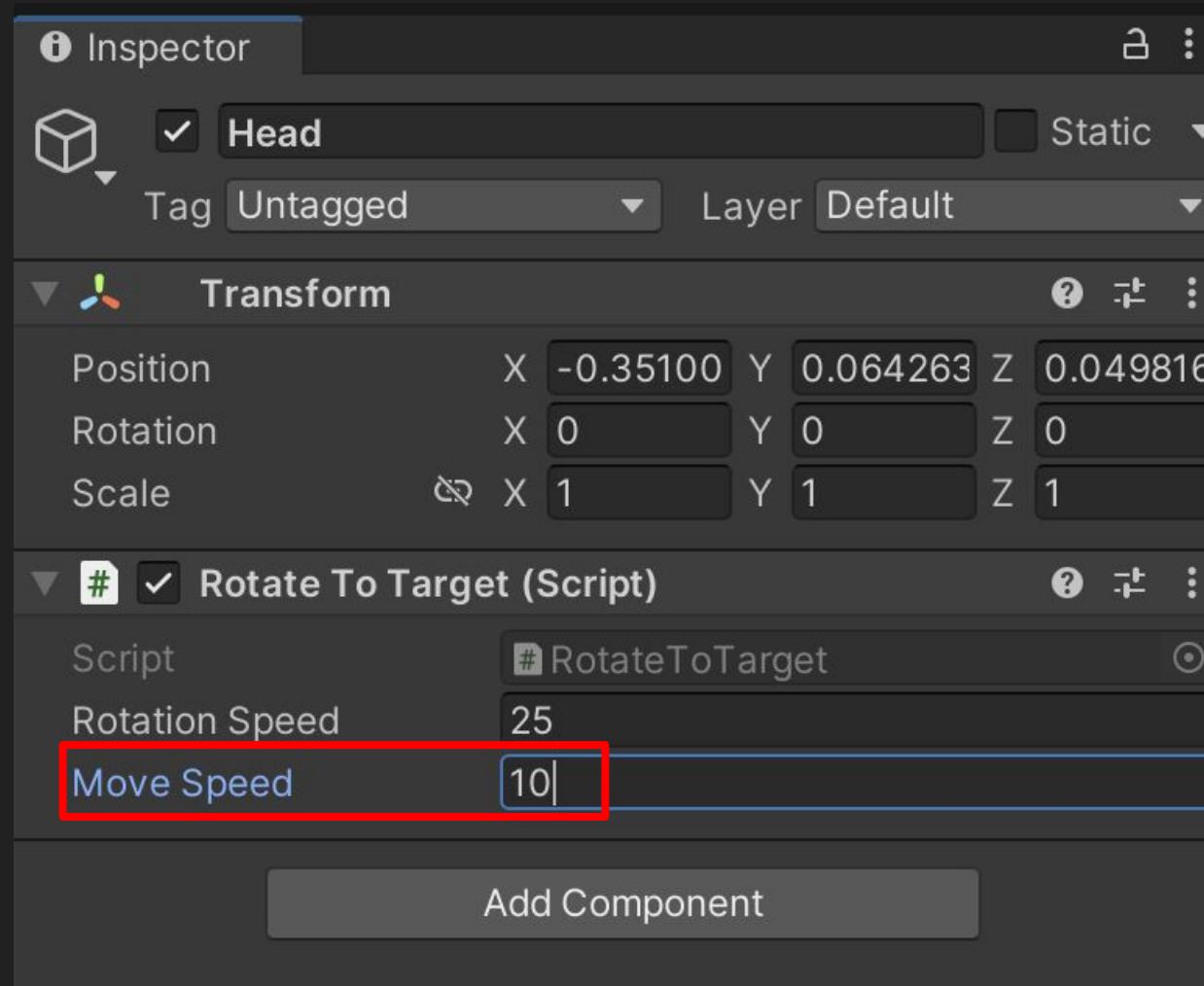
RotateToTarget.cs

```
public class RotateToTarget : MonoBehaviour
{
    public float rotationSpeed;
    private Vector2 direction;

    public float moveSpeed;
    void Update()
    {
        //マウスの位置 - 現在地点
        direction = Camera.main.ScreenToWorldPoint(Input.mousePosition) - transform.position;
        //角度算出 ラジアンを度数法に変換
        float angle = Mathf.Atan2(direction.y, direction.x) * Mathf.Rad2Deg;
        //Quaternion型で 前方向(1,0,0) を angleの方向に向かせる
        Quaternion rotation = Quaternion.AngleAxis(angle, Vector3.forward);
        //1秒あたりrotationSpeedの回転をするように 緩やかに回転してもらう
        transform.rotation = Quaternion.Slerp(transform.rotation, rotation, rotationSpeed*Time.deltaTime);

        //カーソルの位置
        Vector2 cursorPos = Camera.main.ScreenToWorldPoint(Input.mousePosition);
        //現在地からマウス位置に向けてsだけ移動させた分のベクトルを取得出来ます
        transform.position = Vector2.MoveTowards(transform.position, cursorPos, moveSpeed*Time.deltaTime);
    }
}
```

RotateToTarget.cs



マウスカーソルと重なった瞬間、右を向く

マウスカーソルと重なる時に、それまでの方向を維持したままにして下さい

```
public class RotateToTarget : MonoBehaviour
{
    public float rotationSpeed;
    private Vector2 direction;

    public float moveSpeed;
    void Update()
    {
        //マウスの位置 - 現在地点
        direction = Camera.main.ScreenToWorldPoint(Input.mousePosition) - transform.position;

        if (direction.magnitude < 0.1f) return; //回答例

        //角度算出 ラジアンを度数法に変換
        float angle = Mathf.Atan2(direction.y, direction.x) * Mathf.Rad2Deg;
        //Quaternion型で 前方向 を angleの方向に向かせる
        Quaternion rotation = Quaternion.AngleAxis(angle, Vector3.forward);
        //1秒あたりrotationSpeedの回転をするように 緩やかに回転してもらう
        transform.rotation = Quaternion.Slerp(transform.rotation, rotation, rotationSpeed*Time.deltaTime);

        //カーソルの位置
        Vector2 cursorPos = Camera.main.ScreenToWorldPoint(Input.mousePosition);
        //現在地からマウス位置に向けてsだけ移動させた分のベクトルを取得出来ます
        transform.position = Vector2.MoveTowards(transform.position, cursorPos, moveSpeed*Time.deltaTime);
    }
}
```

Hierarchy

All

SampleScene*

Main Camera

Head

head_struct

Tentacle1

Inspector

Tentacle1

Static

Untagged

Layer Default

Transform

Position X -0.948 Y -0.064 Z -0.049

Rotation X 0 Y 0 Z 0

Scale X 1 Y 1 Z 1

Add Component

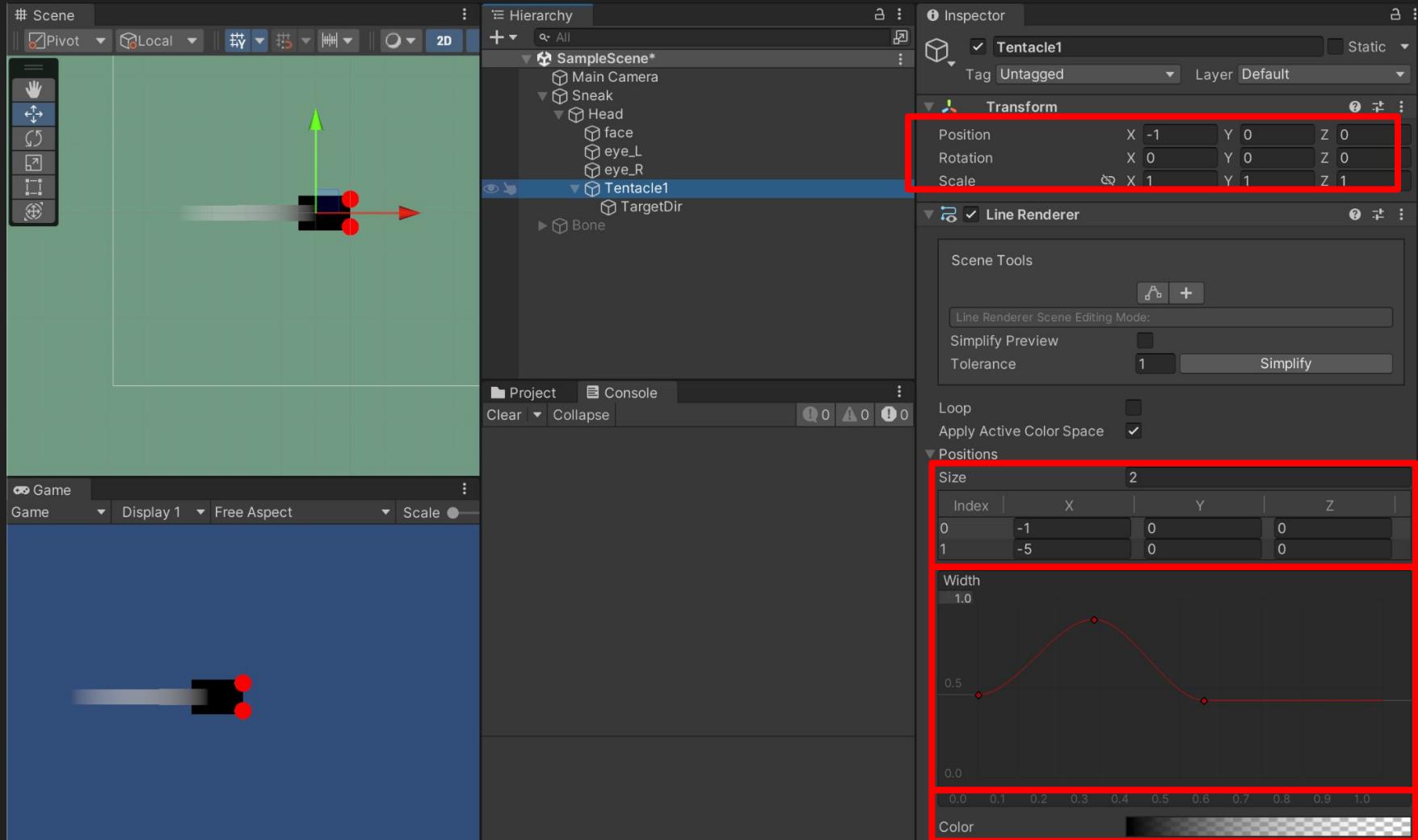
Line

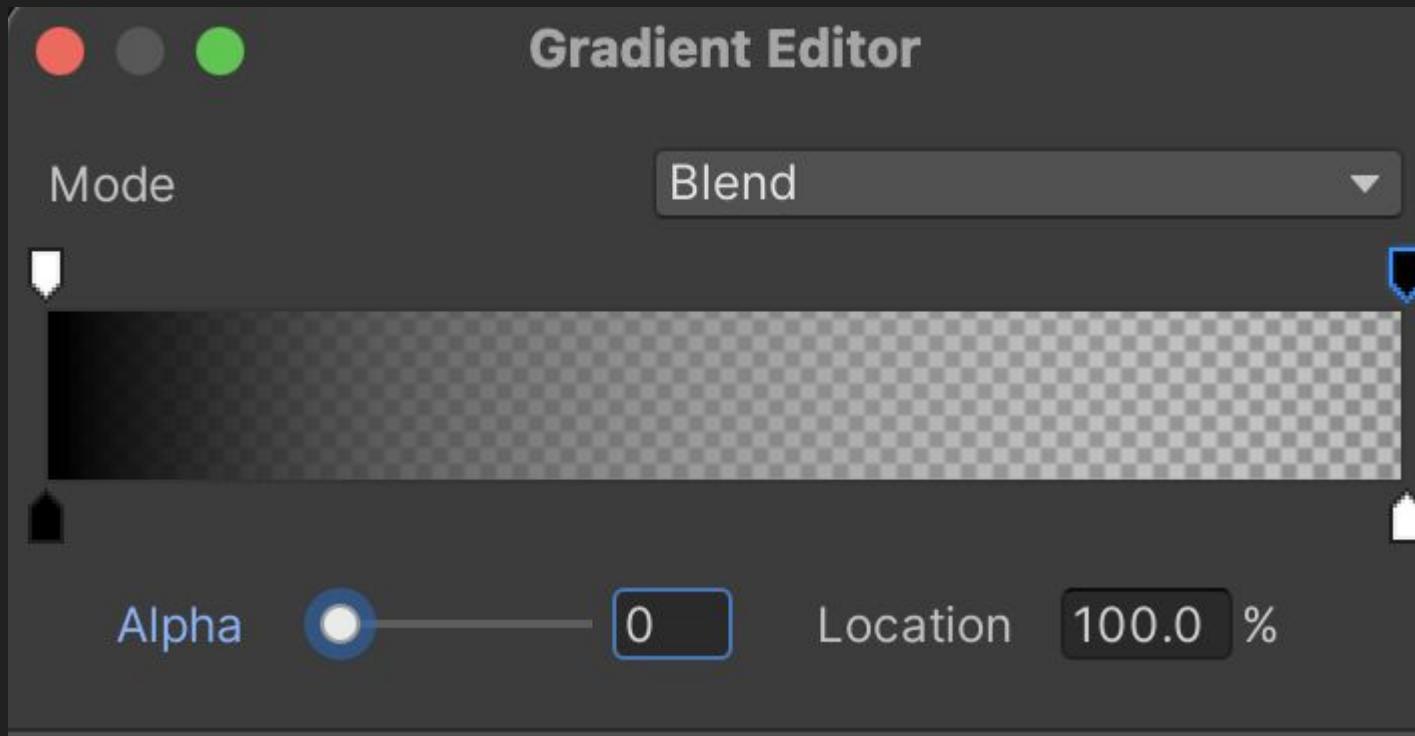
Search

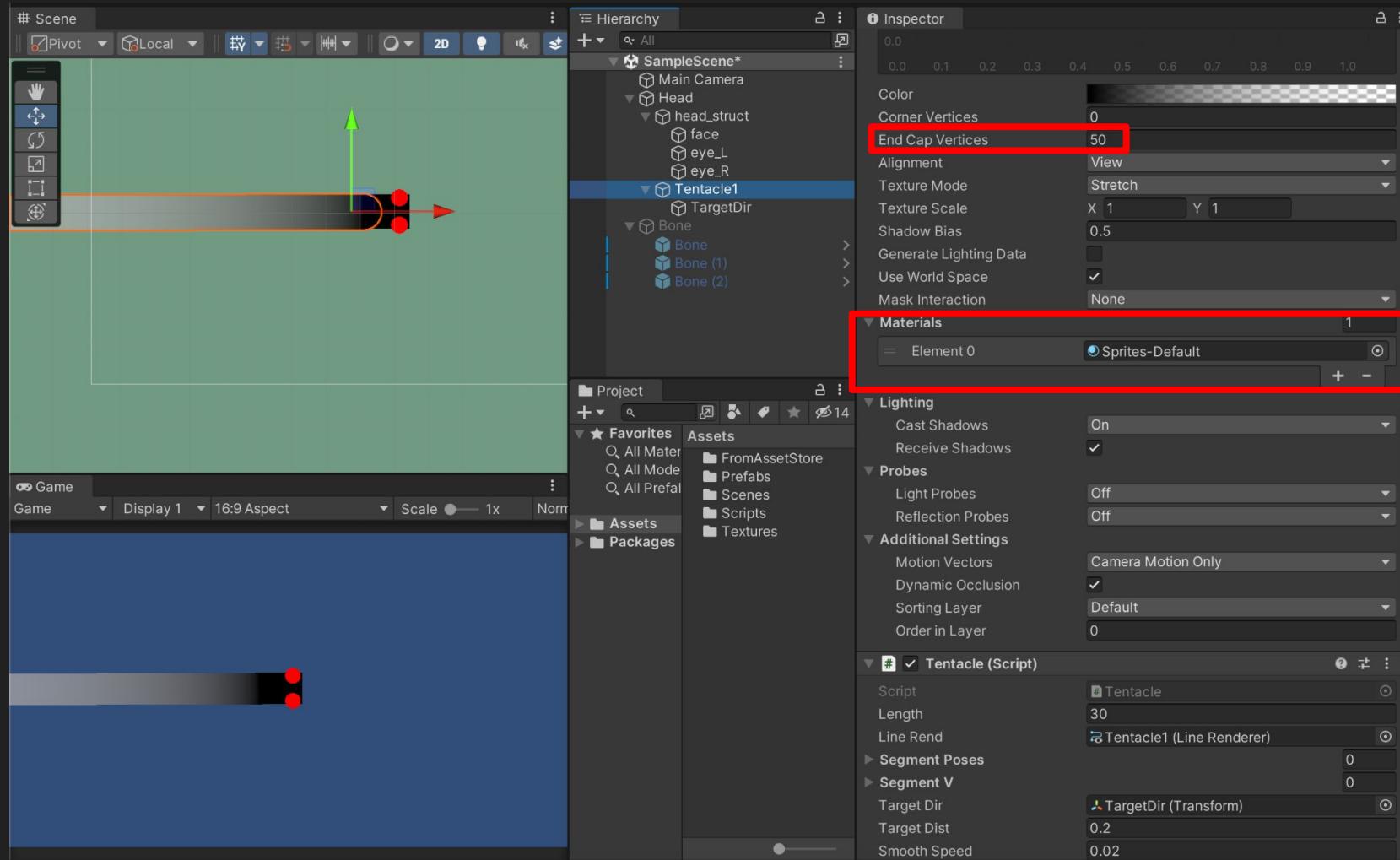
Line Renderer

Outline

The image shows the Unity Editor interface with two main panels: the Hierarchy panel on the left and the Inspector panel on the right. In the Hierarchy panel, there is a scene named 'SampleScene*' containing a 'Main Camera', a 'Head' object which contains a 'head_struct' component, and a 'Tentacle1' object. The 'Tentacle1' object is currently selected. In the Inspector panel, the 'Tentacle1' component is shown with its basic settings: it is not static, has no tag, and is in the default layer. Under the 'Transform' section, its position is at (-0.948, -0.064, -0.049), rotation is (0, 0, 0), and scale is (1, 1, 1). Below the Inspector, there is a component search interface with a search bar containing 'Line', a 'Search' button, and a list of available components: 'Line Renderer' and 'Outline'. A tooltip 'Add Component' is visible above the search bar.



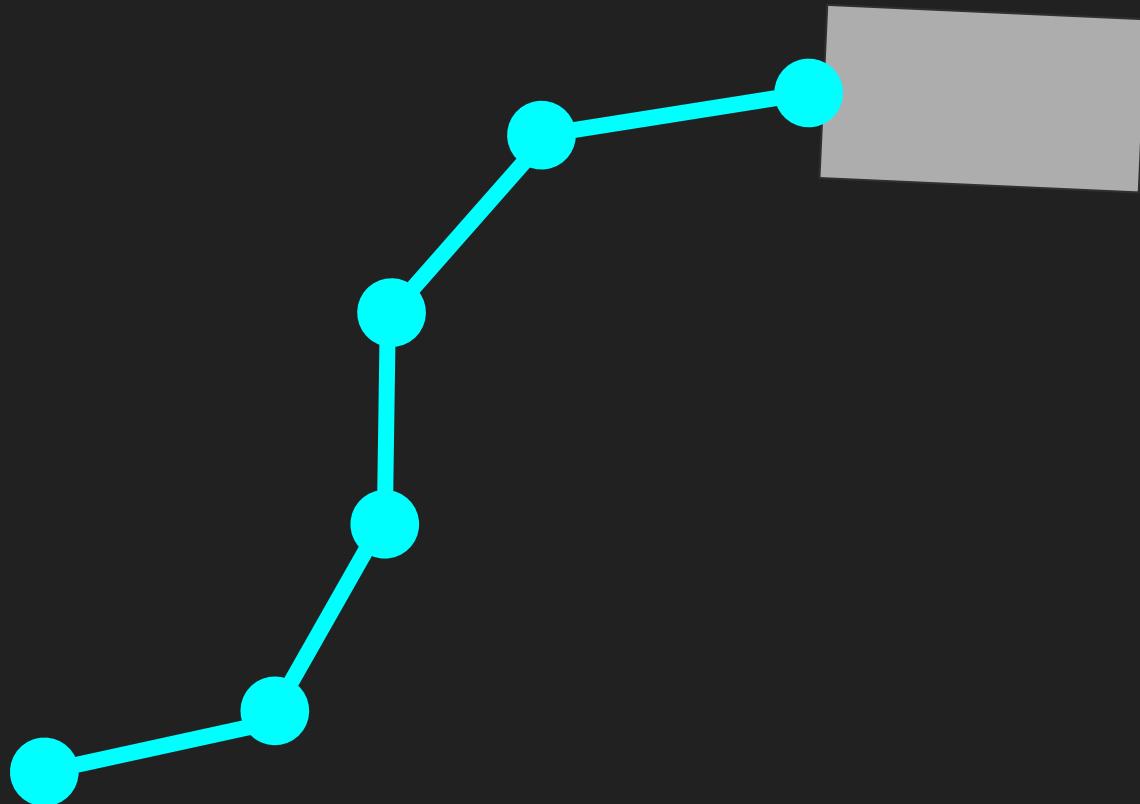




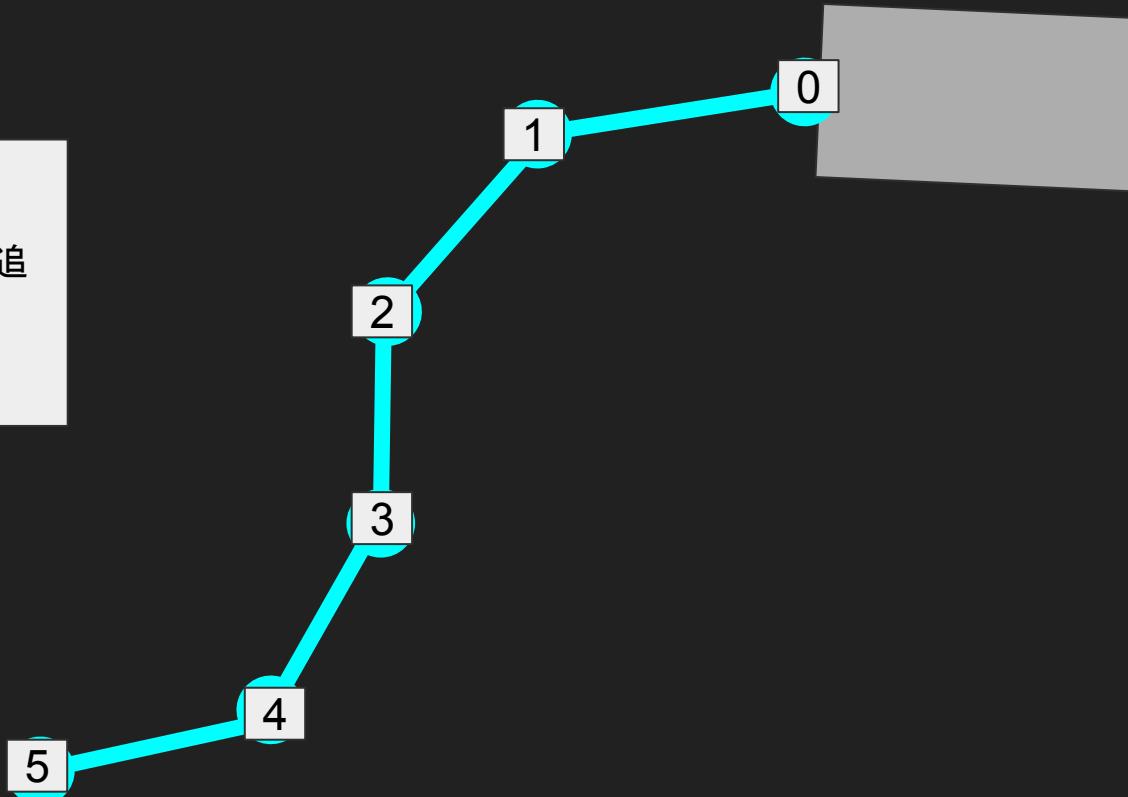
Pos1

Pos2

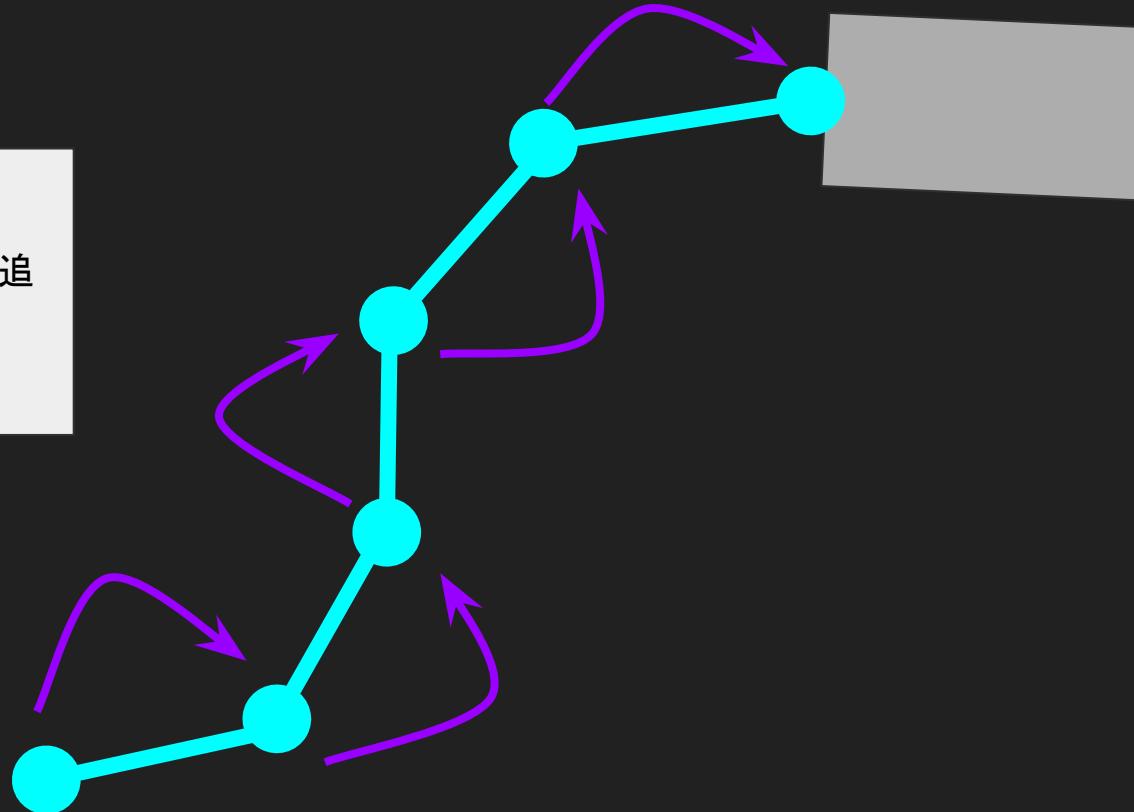




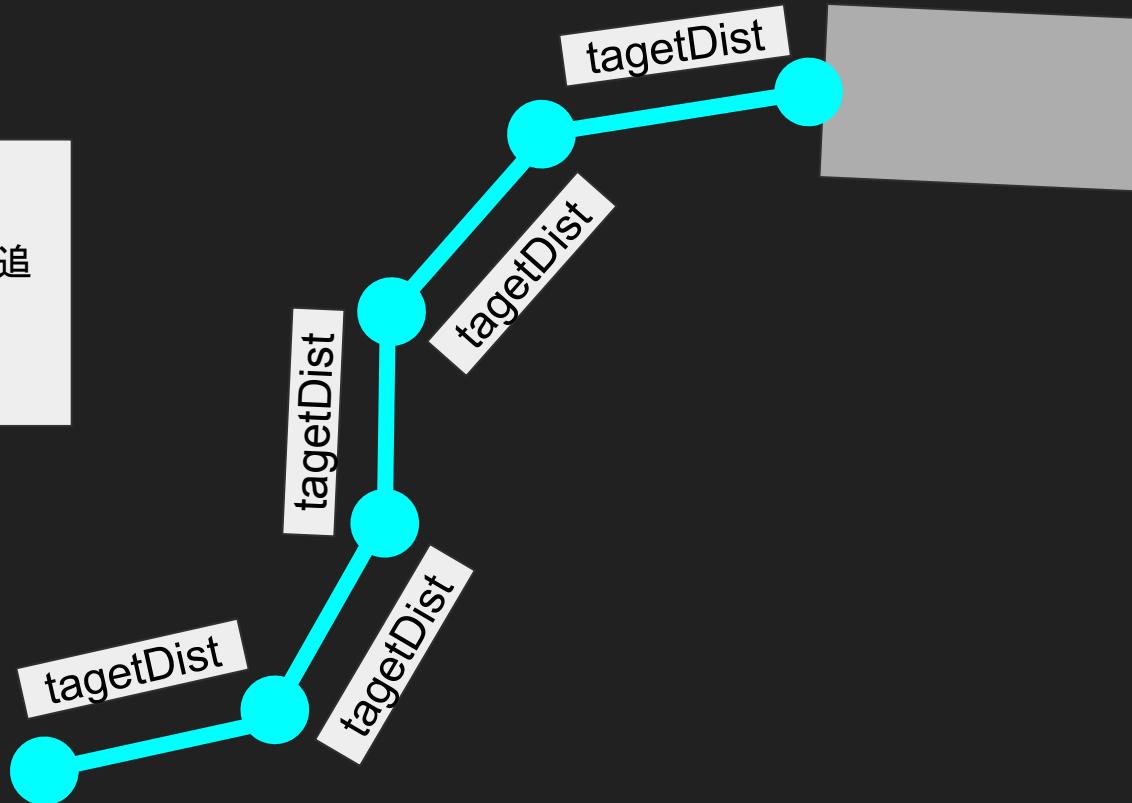
前のポイントに追従するように



前のポイントに追従するように



前のポイントに追従するように

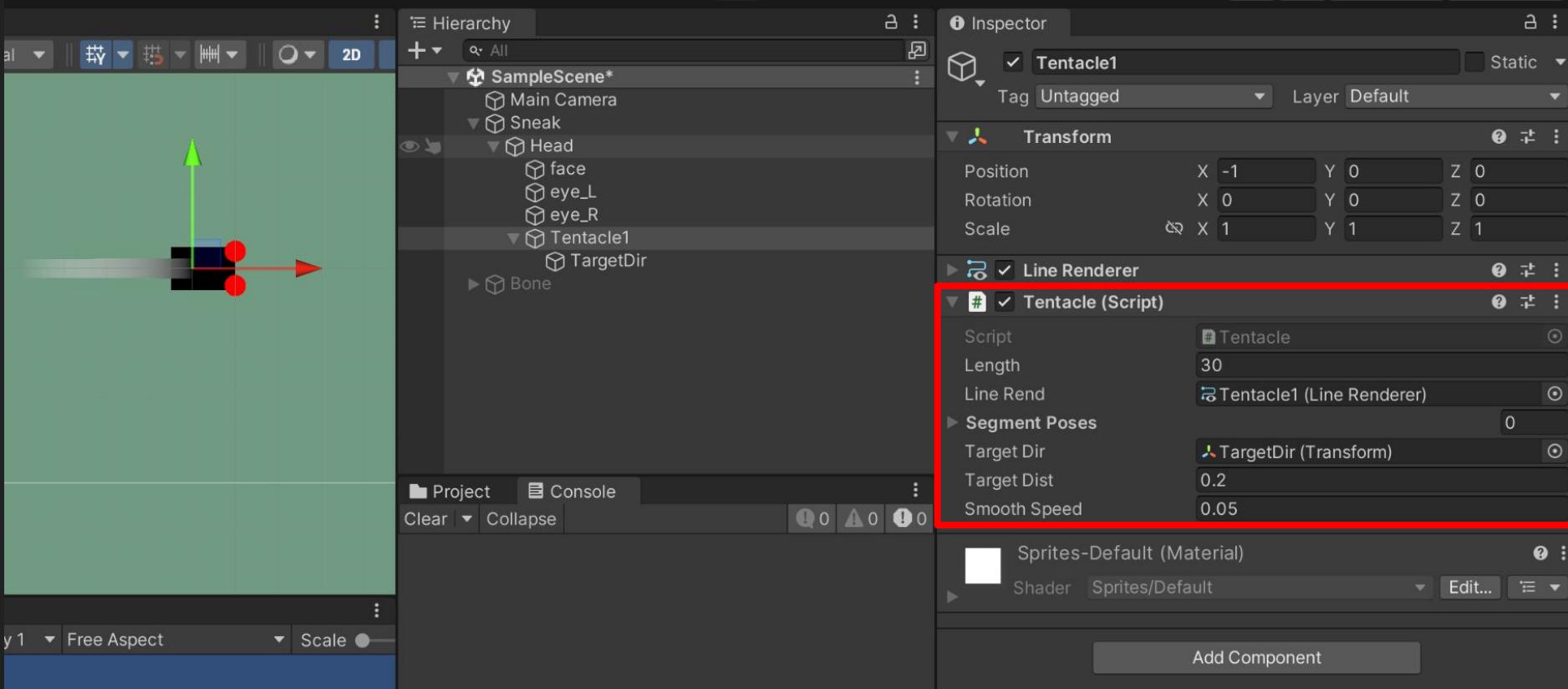


```
public class Tentacle : MonoBehaviour
{
    public int length;           //長さ
    public LineRenderer lineRend; //コンポーネント操作
    public Vector3[] segmentPoses; //節の場所
    private Vector3[] segmentV;   //節ごとのスピード

    public Transform targetDir; //ターゲットの方向
    public float targetDist;   //ターゲットとの距離
    public float smoothSpeed;  //ターゲットまでの到達時間

    void Start()
    {
        lineRend.positionCount = length; //LineRendererの設定
        segmentPoses = new Vector3[length]; //節の場所の数分の配列
        segmentV      = new Vector3[length];
    }
}
```

```
void Update()
{
    //0番目は頭の位置(targetDir)に
    segmentPoses[0] = targetDir.position;
    //全てのポジション情報をsegmentPosesに入れる
    //配列の1番目は0番目の場所に 配列の2番目は1番目の場所に
    for (int i = 1; i < segmentPoses.Length; i++)
    {
        //目標に向かって、滑らかな追従をするように
        segmentPoses[i] = Vector3.SmoothDamp(
            segmentPoses[i],
            segmentPoses[i - 1] - targetDir.right * targetDist,
            ref segmentV[i],
            smoothSpeed);
        //SmoothDamp関数の引数
        //第1引数 → 移動させるオブジェクトの座標 (Vector3)
        //第2引数 → 目的地の座標 (Vector3)
        //第3引数 → 事前に初期化した速度 (Vector3) の参照
        //第4引数 → 目的までの到達時間 (float)
        //第5引数 (省略可) → 最高速度 (float)
        //第6引数 (省略可) → この関数が前回実行されてからの経過時間 (デフォルトはTime.deltaTime)
    }
    //Vector3 の配列を渡すことで、Line の座標を一括でセット
    lineRend.SetPositions(segmentPoses);
}
```



触手の挙動を工夫する

末端の挙動を気持ち遅くする

```
public class Tentacle : MonoBehaviour
{
    public int length;           //長さ
    public LineRenderer lineRend; //コンポーネント操作
    public Vector3[] segmentPoses; //節の場所
    private Vector3[] segmentV;   //節ごとのスピード

    public Transform targetDir; //ターゲットの方向
    public float targetDist;   //ターゲットとの距離
    public float smoothSpeed;  //ターゲットまでの到達時間

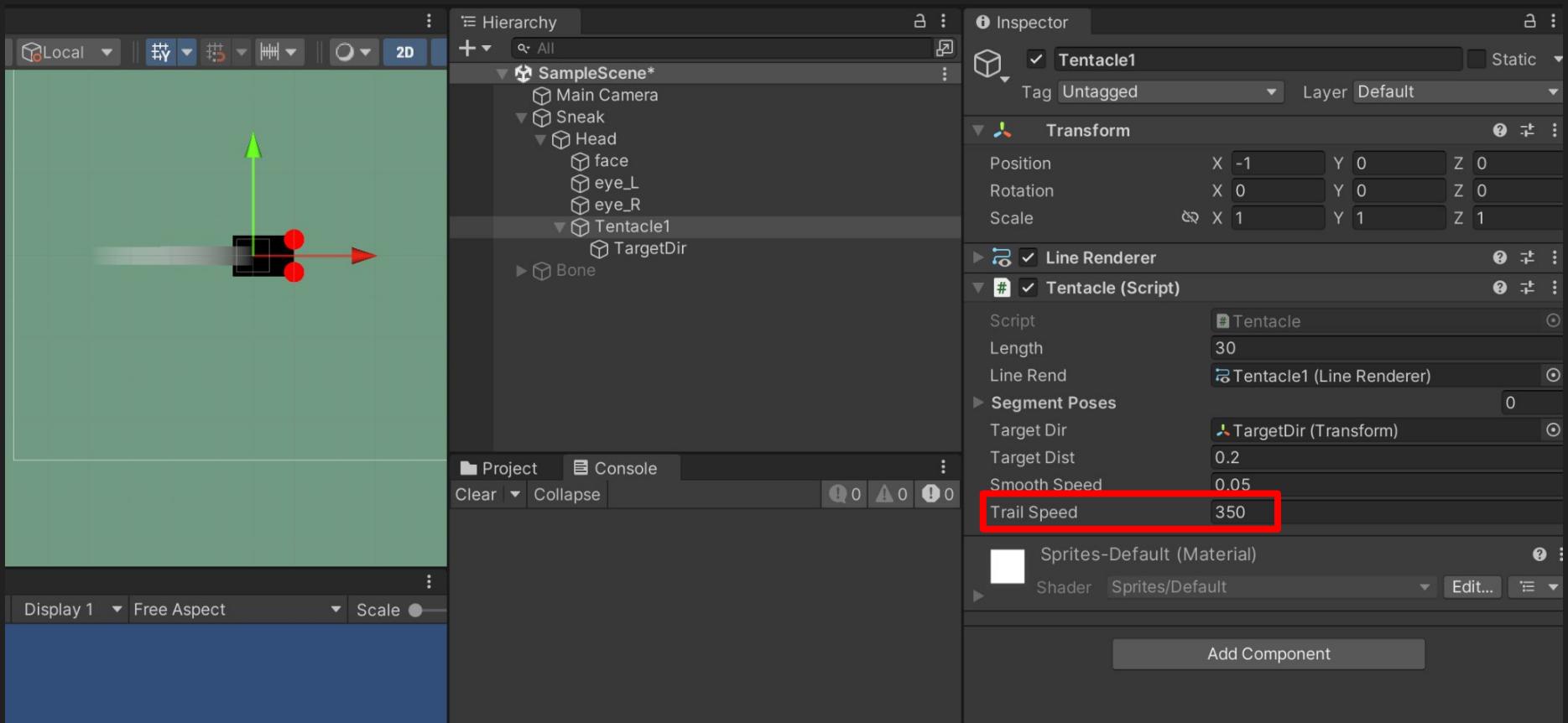
    public float trailSpeed; //尻尾の速度

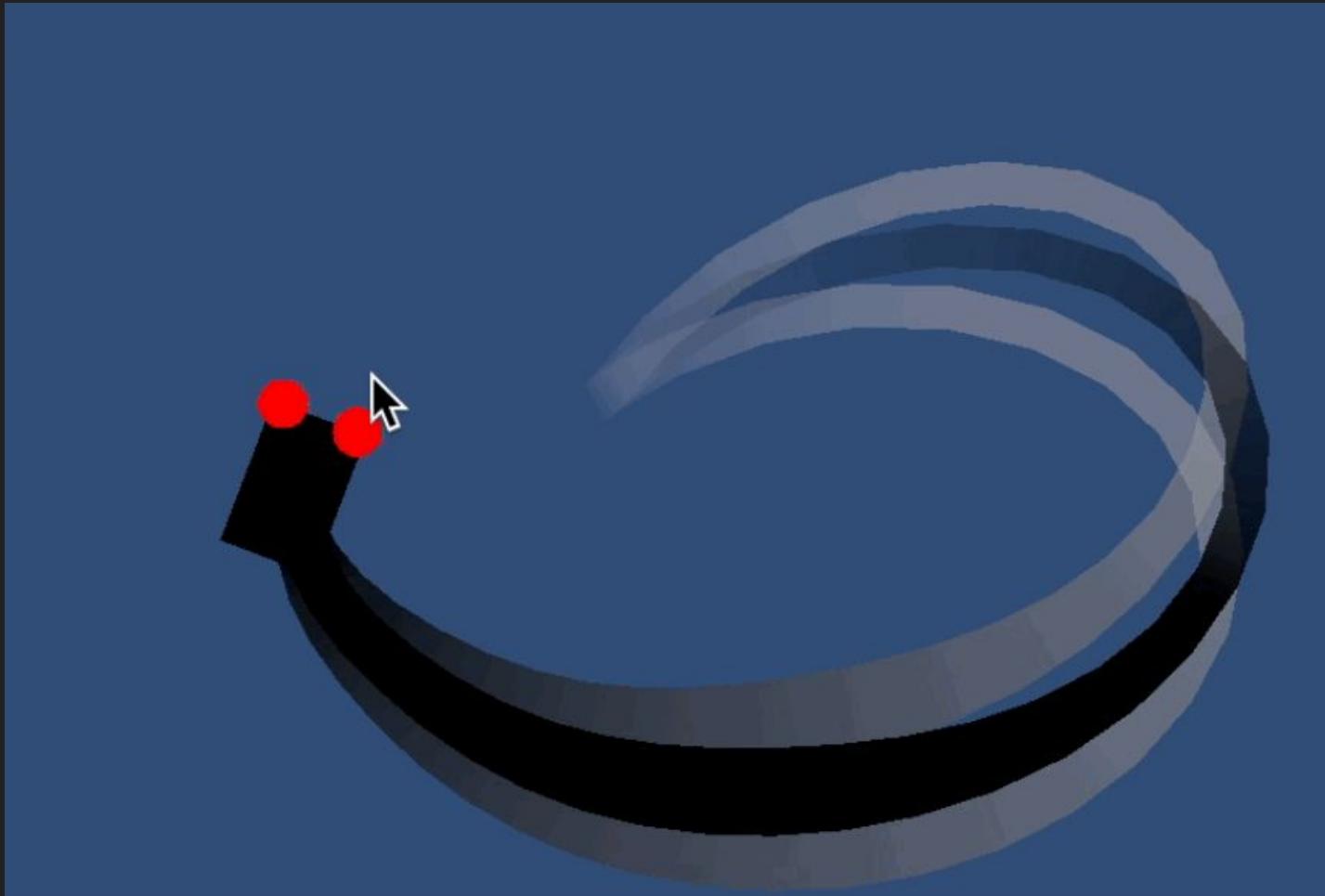
    void Start()
    {
        lineRend.positionCount = length; //LineRendererの設定
        segmentPoses = new Vector3[length]; //節の場所の数分の配列
        segmentV      = new Vector3[length];
    }

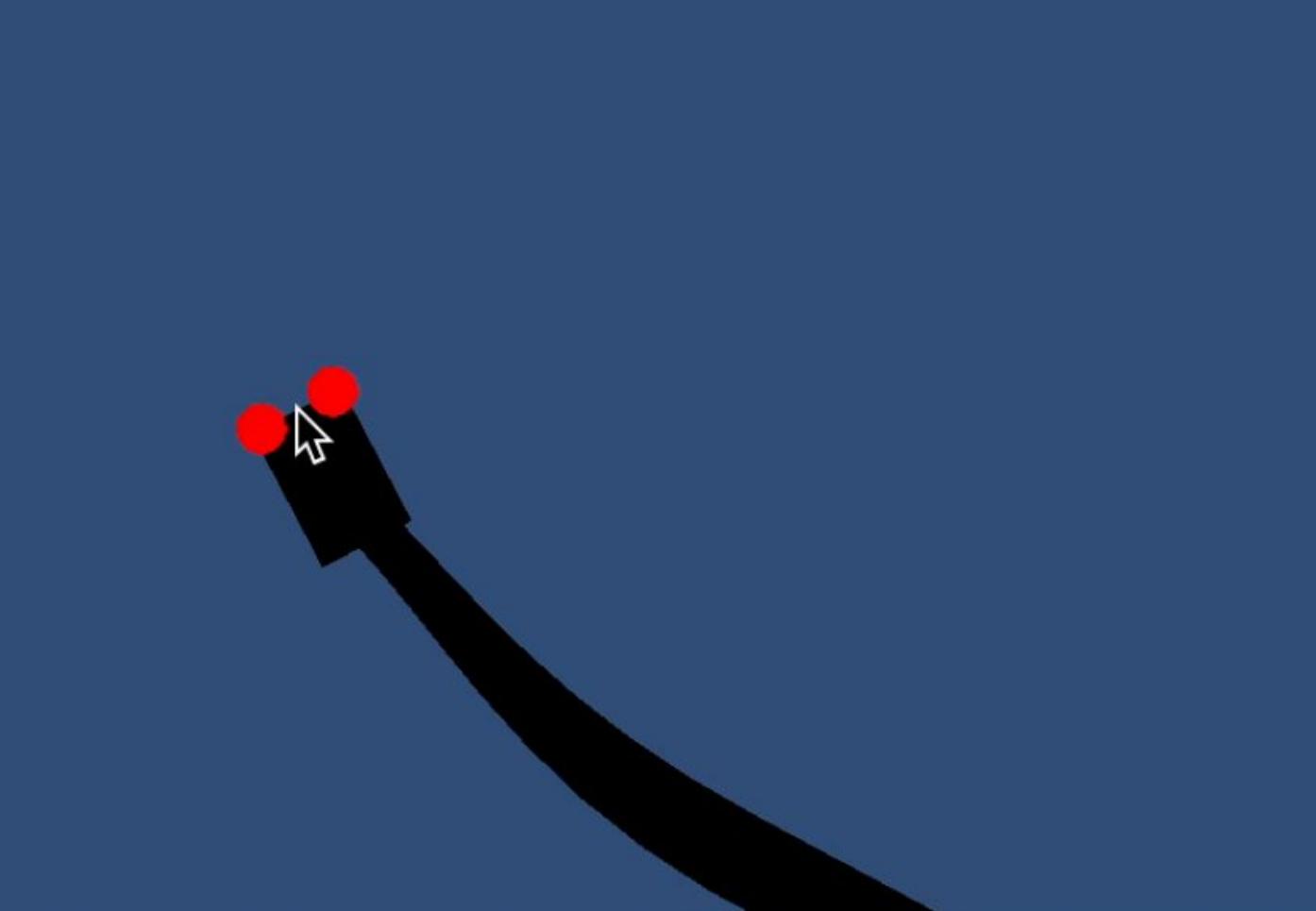
    void Update()
    {
```

Tentacle.cs

```
void Update()
{
    //0番目は頭の位置(targetDir)に
    segmentPoses[0] = targetDir.position;
    //全てのポジション情報をsegmentPosesに入れる
    //配列の1番目は0番目の場所に 配列の2番目は1番目の場所に
    for (int i = 1; i < segmentPoses.Length; i++)
    {
        //目標に向かって、滑らかな追従をするように
        segmentPoses[i] = Vector3.SmoothDamp(
            segmentPoses[i],
            segmentPoses[i - 1] - targetDir.right * targetDist,
            ref segmentVel,
            smoothSpeed + i / trailSpeed);
        //SmoothDamp関数の引数
        //第1引数 → 移動させるオブジェクトの座標 (Vector3)
        //第2引数 → 目的地の座標 (Vector3)
        //第3引数 → 事前に初期化した速度 (Vector3) の参照
        //第4引数 → 目的までの到達時間 (float)
        //第5引数 (省略可) → 最高速度 (float)
        //第6引数 (省略可) → この関数が前回実行されてからの経過時間 (デフォルトはTime.deltaTime)
    }
    //Vector3 の配列を渡すことで、Line の座標を一括でセット
    lineRend.SetPositions(segmentPoses);
}
```







class Tentacle : MonoBehaviour

```
{  
    public int length;           //長さ  
    public LineRenderer lineRend; //コンポーネント操作  
    public Vector3[] segmentPoses; //節の場所  
    private Vector3[] segmentV;   //節ごとのスピード  
  
    public Transform targetDir; //ターゲットの方向  
    public float targetDist;   //ターゲットとの距離  
    public float smoothSpeed;  //ターゲットまでの到達時間  
  
    public float trailSpeed; //尻尾の速度  
  
    public float wiggleSpeed; //くねくねするスピード  
    public float wiggleMagnitude; //くねくねする大きさ  
    public Transform wiggleDir; //くねくねする方向
```

Tentacle.cs

```
void Update()
{
    wiggleDir.localRotation = Quaternion.Euler(0,0,Mathf.Sin(Time.time*wiggleSpeed)*wiggleMagnitude);
```

Hierarchy



All

SampleScene*

Main Camera

Sneak

Head

face

eye_L

eye_R

Tentacle1

WiggleDir

TargetDir

Tentacle2

Tentacle3

Bone



Tentacle (Script)



Script

Tentacle



Length

30

Line Rend

Tentacle1 (Line Renderer)



▶ Segment Poses

0

Target Dir

TargetDir (Transform)



Target Dist

0.2

Smooth Speed

0.05

Trail Speed

350

Wiggle Speed

10

Wiggle Magnitude

20

Wiggle Dir

WiggleDir (Transform)



触手の挙動を工夫する

一定の距離間を保つようにする

```
void Update()
{
    wiggleDir.localRotation = Quaternion.Euler(0,0,Mathf.Sin(Time.time*wiggleSpeed)*wiggleMagnitude);

    //0番目は頭の位置(targetDir)に
    segmentPoses[0] = targetDir.position;
    //全てのポジション情報をsegmentPosesに入れる
    //配列の1番目は0番目の場所に 配列の2番目は1番目の場所に
    for (int i = 1; i < segmentPoses.Length; i++)
    {
        //前のポジションと次のポジションの位置の距離を一定に保つ
        Vector3 targetPos = segmentPoses[i - 1] + (segmentPoses[i] - segmentPoses[i - 1]).normalized * targetDist;

        //目標に向かって、滑らかな追従をするように
        segmentPoses[i] = Vector3.SmoothDamp(
            segmentPoses[i],
            targetPos,
            ref segmentV[i],
            smoothSpeed + i / trailSpeed);
    }
    //SmoothDamp関数の引数
    //第1引数 → 移動させるオブジェクトの座標 (Vector3)
    //第2引数 → 目的地の座標 (Vector3)
    //第3引数 → 事前に初期化した速度 (Vector3) の参照
    //第4引数 → 目的までの到達時間 (float)
    //第5引数 (省略可) → 最高速度 (float)
    //第6引数 (省略可) → この関数が前回実行されてからの経過時間 (デフォルトはTime.deltaTime)
}

//Vector3 の配列を渡すことで、Line の座標を一括でセット
lineRend.SetPositions(segmentPoses);
}
```

▼ # ✓ Tentacle Two (Script)



Script	# TentacleTwo	(○)
Length	30	
Line Rend	☛ Tentacle1 (Line Renderer)	(○)
▶ Segment Poses	0	
Target Dir	TargetException (Transform)	(○)
Target Dist	0.2	
Smooth Speed	0.2	
Trail Speed	350	
Wiggle Speed	0	
Wiggle Magnitude	0	
Wiggle Dir	TargetException (Transform)	(○)

尻尾の部分を強調

```
public class TentacleTwo : MonoBehaviour
{
    public int length; //長さ
    public LineRenderer lineRend; //コンポーネント操作
    public Vector3[] segmentPoses; //節の場所
    private Vector3[] segmentV; //節ごとのスピード

    public Transform targetDir; //ターゲットの方向
    public float targetDist; //ターゲットとの距離
    public float smoothSpeed; //ターゲットまでの到達時間

    public float trailSpeed; //尻尾の速度

    public float wiggleSpeed; //くねくねするスピード
    public float wiggleMagnitude; //くねくねする大きさ
    public Transform wiggleDir; //くねくねする方向

    public Transform trailEnd; //尻尾
```

Tentacle.cs

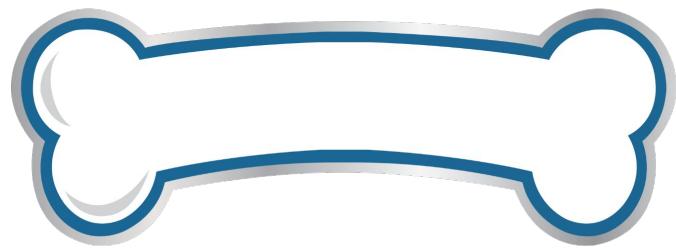
```
void Update()
{
    wiggleDir.localRotation = Quaternion.Euler(0,0,Mathf.Sin(Time.time*wiggleSpeed)*wiggleMagnitude);

    //0番目は頭の位置(targetDir)に
    segmentPoses[0] = targetDir.position;
    //全てのポジション情報をsegmentPosesに入れる
    //配列の1番目は0番目の場所に 配列の2番目は1番目の場所に
    for (int i = 1; i < segmentPoses.Length; i++)
    {
        //前のポジションと次のポジションの位置の距離を一定に保つ
        Vector3 targetPos = segmentPoses[i - 1] + (segmentPoses[i] - segmentPoses[i - 1]).normalized * targetDist;

        //目標に向かって、滑らかな追従をするように
        segmentPoses[i] = Vector3.SmoothDamp(
            segmentPoses[i],
            targetPos,
            ref segmentV[i],
            smoothSpeed + i / trailSpeed);
        //SmoothDamp関数の引数
        //第1引数 → 移動させるオブジェクトの座標 (Vector3)
        //第2引数 → 目的地の座標 (Vector3)
        //第3引数 → 事前に初期化した速度 (Vector3) の参照
        //第4引数 → 目的までの到達時間 (float)
        //第5引数 (省略可) → 最高速度 (float)
        //第6引数 (省略可) → この関数が前回実行されてからの経過時間 (デフォルトはTime.deltaTime)
    }
    //Vector3 の配列を渡すことで、Line の座標を一括でセット
    lineRend.SetPositions(segmentPoses);
    trailEnd.position = segmentPoses[segmentPoses.Length - 1];
}
```

体

骨 節ごとに体が



```
public class TentacleTwo : MonoBehaviour
{
    public int length;           //長さ
    public LineRenderer lineRend; //コンポーネント操作
    public Vector3[] segmentPoses; //節の場所
    private Vector3[] segmentV;   //節ごとのスピード

    public Transform targetDir; //ターゲットの方向
    public float targetDist;   //ターゲットとの距離
    public float smoothSpeed;  //ターゲットまでの到達時間

    public float trailSpeed; //尻尾の速度

    public float wiggleSpeed; //くねくねするスピード
    public float wiggleMagnitude; //くねくねする大きさ
    public Transform wiggleDir; //くねくねする方向

    public Transform trailEnd; //尻尾
    public Transform[] bodyParts; //体の部位
```

Tentacle.cs

```
void Update()
{
    wiggleDir.localRotation = Quaternion.Euler(0,0,Mathf.Sin(Time.time*wiggleSpeed)*wiggleMagnitude);

    //0番目は頭の位置(targetDir)に
    segmentPoses[0] = targetDir.position;
    //全てのポジション情報をsegmentPosesに入れる
    //配列の1番目は0番目の場所に 配列の2番目は1番目の場所に
    for (int i = 1; i < segmentPoses.Length; i++)
    {
        //前のポジションと次のポジションの位置の距離を一定に保つ
        Vector3 targetPos = segmentPoses[i - 1] + (segmentPoses[i] - segmentPoses[i - 1]).normalized * targetDist;

        //目標に向かって、滑らかな追従をするように
        segmentPoses[i] = Vector3.SmoothDamp(
            segmentPoses[i],
            targetPos,
            ref segmentV[i],
            smoothSpeed + i / trailSpeed);
        bodyParts[i - 1].transform.position = segmentPoses[i];
    }
    //Vector3 の配列を渡すことで、Line の座標を一括でセット
    lineRend.SetPositions(segmentPoses);
    //trailEnd.position = segmentPoses[segmentPoses.Length - 1];
}
```

Tentacle Two (Script) ? + ⋮

Script	# TentacleTwo	○
Length	6	○
Line Rend	Tentacle1 (Line Renderer)	○
▶ Segment Poses	0	○
Target Dir	TargetDir (Transform)	○
Target Dist	0.2	○
Smooth Speed	0.2	○
Trail Speed	350	○
Wiggle Speed	0	○
Wiggle Magnitude	0	○
Wiggle Dir	WiggleDir (Transform)	○
Trail End	tail (Transform)	○
▼ Body Parts	5	○
= Element 0	Bone (Transform)	○
= Element 1	Bone (1) (Transform)	○
= Element 2	Bone (2) (Transform)	○
= Element 3	Bone (3) (Transform)	○
= Element 4	tail (Transform)	○

+ -

節ごとに回転して欲しい



BodyRotation.cs

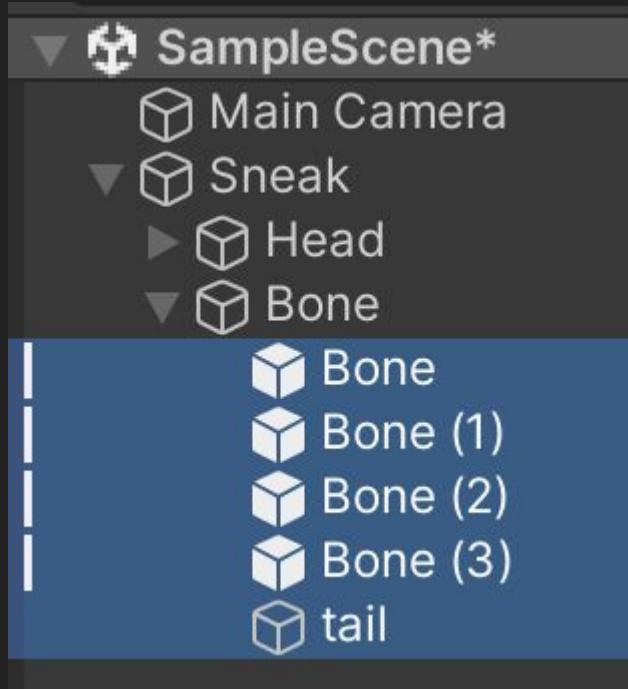
```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class BodyRotation : MonoBehaviour
{
    public float speed;

    private Vector2 direction;
    public Transform target;

    void Start()
    {
    }

    void Update()
    {
        direction = target.position - transform.position;
        float angle = Mathf.Atan2(direction.y,direction.x)*Mathf.Rad2Deg;
        Quaternion rotaion = Quaternion.AngleAxis(angle-90f,Vector3.forward);
        transform.rotation = Quaternion.Slerp(transform.rotation,rotaion,speed*Time.deltaTime);
    }
}
```



Hierarchy

+ All SampleScene*

- Main Camera
- Sneak
 - Head
 - Bone
 - Bone
 - Bone (1)
 - Bone (2)
 - Bone (3)
 - tail

Inspector

Bone

- Static
- Untagged Layer Default
- Prefab Open Select Overrides

Transform

Position	X -1.5	Y 0	Z 0
Rotation	X 0	Y 0	Z -90
Scale	X 0.100	Y 0.100	Z 1

Sprite Renderer

Sprite	Bone
Color	[Color Swatch]
Flip	X Y
Draw Mode	Simple
Mask Interaction	None
Sprite Sort Point	Center
Material	Sprites-Default

Additional Settings

Sorting Layer	Default
Order in Layer	0

Body Rotation (Script)

Script	# BodyRotation
Speed	10
Target	None (Transform)

Materials

Sprites-Default (Material)

Shader Sprites/D...

Edit...



ゲーム起動時の位置を修正

BodyRotation.cs

```
    --> 1; i < segmentPoses.Length; i++)
{
    if ((segmentPoses[i] - segmentPoses[i - 1]).magnitude < targetDist){
        continue;
    }
    else {
        //前のポジションと次のポジションの位置の距離を一定に保つ
        Vector3 targetPos = segmentPoses[i - 1] + (segmentPoses[i] - segmentPoses[i - 1]);
        segmentPoses[i] = Vector3.SmoothDamp(segmentPoses[i], targetPos, //目標に向かって、
                                              ref segmentV[i], smoothSpeed + i / trailSpeed);
        bodyParts[i - 1].transform.position = segmentPoses[i];
    }
}
lineRend.SetPositions(segmentPoses); //Vector3 の配列を渡すことで、Line の座標を一括でセット
}

private void ResetPos(){
    segmentPoses[0] = targetDir.position;
    for (int i = 1; i < length; i++){
        segmentPoses[i] = segmentPoses[i - 1] - targetDir.right * targetDist;
    }
    lineRend.SetPositions(segmentPoses);
}
```

```
void Start()
{
    LineRend.positionCount = length;    //LineRendererの設定
    segmentPoses = new Vector3[length]; //節の場所の数分の配列
    segmentV     = new Vector3[length];

    ResetPos();
}
```