3D ゲームプログラミングⅡ／ゲームプログラミング B
第 3 回 TANKS!

準備

新規にプロジェクトを作成し、Asset Store から Tanks! Tutorial をダウンロード、インポートする。

1. シーンの設定

ステージの作成

① Scene または Hierarchy の Directional Light を削除する(②で使用するプレハブにライトが含まれているので、それを使用する)。

② Prefabs フォルダーの Level Art を Hierarchy にドラッグアンドドロップする。

Global Illumination(GI)の設定

③ Window ▸ Lighting ▸ Settings を選び、Lighting パネルを表示する。

④ Auto Generate を無効にする。

⑤ Mixed Lighting の Baked Global Illumination を無効にする。

⑥ Lightmapping Settings の Indirect Resolution を 0.5 に変更する。

⑦ Environment の Environment Lighting の Source を Color に変更し、Ambient Color(環境光)を RGB で(0.328, 0.276, 0.536)に設定する。

⑧ Generate Lighting をクリックする。

カメラの設定

⑨ Main Camera を選択し、Position を(-43, 42, -25)、Rotation を(40, 60, 0)に設定する。

⑩ Camera の Projection を Orthographic に変更する（遠近感がなくなる)。

⑪ Clear Flags を Solid Color に変更し、BackGround（背景色）を(107, 75, 33, 5)に設定する。

⑫ Size を 10 に設定する。

2. 戦車の設定

戦車の作成

① Models フォルダーの Tank を Hierarchy にドラッグアンドドロップする。

② Tank が選択されている状態で、Inspector の Layer を Players に変更する。Change Layer ダイアログが表示されるので、No, this object only を選ぶ。

Rigidbody の設定

③ Tank に Rigidbody を追加する。

④ Rigidbody の Constraints を展開し、Freeze Position の Y、Freeze Rotation の X と Z に✔を入れる。

コライダーの設定

⑤ Tank に Box Collider を追加する。

⑥ Box Collider の Center を(0, 0.85, 0)、Size を(1.5, 1.7, 1.6)に設定する。

Audio の追加

⑦ Tank に Audio Source を追加する(エンジン音用)。

⑧ AudioSource の AudioClip に EngineIdle を設定し、Loop を有効にする。

⑨ Tank にもうひとつ Audio Source を追加する(発射音用)。

⑩ 追加した Audio Source の Play On Awake を無効にする。

戦車のプレハブ化

⑪ Project の Prefabs フォルダーを開く。

⑫ Tank を Project にドラッグアンドドロップして、プレハブ化する。

砂煙の追加

⑬ Prefabs フォルダーの DustTrail を Hierarchy の Tank にドラッグアンドドロップして、子オブジェクト化する。

⑭ Hierarchy の DustTrail を Duplicate し、名前を LeftDustTrail と RightDustTrail に変更する。

⑮ LeftDustTrail の Position を(-0.5, 0, -0.75)、RightDustTrail の Position を(0.5, 0, -0.75)に設定する。

スクリプトの追加と編集

⑯ Scripts¥Tank フォルダーの TankMovement.cs を Hierarchy の Tank にドラッグアンドドロップする。

⑰ スクリプトを開き、次のように編集する。

```
using UnityEngine;

public class TankMovement : MonoBehaviour
{
    public int m_PlayerNumber = 1;              // プレーヤーの識別番号
    public float m_Speed = 12f;                 // 戦車の移動速度.
    public float m_TurnSpeed = 180f;            // 戦車の旋回の角速度.
    public AudioSource m_MovementAudio;   // エンジン音の AudioSource への参照
    public AudioClip m_EngineIdling;            // アイドリング時のエンジン音
    public AudioClip m_EngineDriving;           // 移動時のエンジン音
    public float m_PitchRange = 0.2f;           // エンジン音のピッチの変化量の範囲
```

```
private string m_MovementAxisName;      // 戦車を前後に移動させる入力軸名
private string m_TurnAxisName;          // 戦車を旋回させる入力軸名
private Rigidbody m_Rigidbody;          // 戦車の Rigidbody への参照
private float m_MovementInputValue;     // 移動の入力軸の値
private float m_TurnInputValue;         // 旋回の入力軸の値
private float m_OriginalPitch;          // エンジン音のピッチの初期値

// 最初に実行される
private void Awake ()
{
    m_Rigidbody = GetComponent<Rigidbody> ();
}

// アクティブならば、Awake()に続いて実行される
private void OnEnable ()
{
    // 戦車が動けるようにする
    m_Rigidbody.isKinematic = false;
    // 入力値のリセット
    m_MovementInputValue = 0f;
    m_TurnInputValue = 0f;
}

// 無効になったら実行される
private void OnDisable ()
{
    // 戦車を動かなくする
    m_Rigidbody.isKinematic = true;
}

private void Start ()
{
    // プレーヤー番号を元に軸名を決定する
    m_MovementAxisName = "Vertical" + m_PlayerNumber;
    m_TurnAxisName = "Horizontal" + m_PlayerNumber;
    // エンジン音のピッチの初期値を保存する
    m_OriginalPitch = m_MovementAudio.pitch;
}

private void Update ()
{
    // 2 つの入力軸の値を保存する
    m_MovementInputValue = Input.GetAxis (m_MovementAxisName);
    m_TurnInputValue = Input.GetAxis (m_TurnAxisName);
    EngineAudio ();
}
```

```
// エンジン音の処理
private void EngineAudio ()
{
    // 入力がない場合
    if (Mathf.Abs (m_MovementInputValue) < 0.1f
                && Mathf.Abs (m_TurnInputValue) < 0.1f)
    {
        // 移動中の音が鳴っている場合
        if (m_MovementAudio.clip == m_EngineDriving)
        {
            // アイドリングの音に変える
            m_MovementAudio.clip = m_EngineIdling;
            m_MovementAudio.pitch = Random.Range(
                m_OriginalPitch - m_PitchRange,
                m_OriginalPitch + m_PitchRange);
            m_MovementAudio.Play ();
        }
    }
    // 入力がある場合
    else
    {
        // アイドリングの音が鳴っている場合
        if (m_MovementAudio.clip == m_EngineIdling)
        {
            // 移動中の音に変える
            m_MovementAudio.clip = m_EngineDriving;
            m_MovementAudio.pitch = Random.Range(
                m_OriginalPitch - m_PitchRange,
                m_OriginalPitch + m_PitchRange);
            m_MovementAudio.Play();
        }
    }
}

// 正確な時間間隔（60 分の 1 秒）で実行される
private void FixedUpdate ()
{
    Move ();
    Turn ();
}

// 戦車の移動処理
private void Move ()
{
    // 移動量を求める.
    Vector3 movement = transform.forward * m_MovementInputValue
```

```
                                    * m_Speed * Time.deltaTime;
        // Rigidbody の移動
        m_Rigidbody.MovePosition(m_Rigidbody.position + movement);
    }

    // 戦車の旋回処理
    private void Turn ()
    {
        // Y 軸回りの回転角を求める
        float turn = m_TurnInputValue * m_TurnSpeed * Time.deltaTime;
        // 回転のクォータニオンを求める
        Quaternion turnRotation = Quaternion.Euler (0f, turn, 0f);
        // Rigidbody の回転
        m_Rigidbody.MoveRotation (m_Rigidbody.rotation * turnRotation);
    }
}
```

⑱ Tank が選択されている状態で、Inspector の TankMovement(Script) の MovementAudio に、最初の Audio Source をドラッグアンドドロップする。

⑲ EngineIdling と EngineDriving に対応する AudioClip を設定する。

プレハブの更新

⑳ Inspector の上の方にある Apply をクリックして、戦車のプレハブを更新する。

シーンを保存し、プレイしてみる。W／S キーで前進／後退、A／W キーで旋回する。Edit ▸ Project Settings ▸ Input で InputManager が開き、キーの割当を確認／編集できる。

3. カメラの移動とズーム

カメラリグの作成

① GameObject ▸ Create Empty で空の GameObject を作成し、名前を CameraRig に変更する。

② CameraRig の Rotation を(40, 60, 0)に設定する。

③ MainCamera を CameraRig にドラッグアンドドロップし、子オブジェクトにする。

④ Main Camera の Position を(0, 0, -65)、Rotation を(0, 0, 0)に変更する。

スクリプトの追加と編集

⑤ Scripts¥Camera フォルダーの CameraControl.cs を Hierarchy の CameraRig にドラッグアンドドロップする。

⑥ CameraControl.cs を開き、[HideInInspector]を削除する。

```
using UnityEngine;

public class CameraControl : MonoBehaviour
{
    public float m_DampTime = 0.2f;                 // カメラが目的地に達するまでの時間
    public float m_ScreenEdgeBuffer = 4f;           // ズームの余裕
    public float m_MinSize = 6.5f;                  // ズームの最小値
    public Transform[] m_Targets;                   // 戦車の Transform の配列

    private Camera m_Camera;
    private float m_ZoomSpeed;                      // ズーム速度
    private Vector3 m_MoveVelocity;                 // カメラの移動速度ベクトル
    private Vector3 m_DesiredPosition;              // カメラの移動先座標

    private void Awake()
    {
        m_Camera = GetComponentInChildren<Camera>();
    }

    private void FixedUpdate()
    {
        Move();
        Zoom();
    }

    // カメラの移
    private void Move()
    {
        // カメラの移動目標座標を得る
        FindAveragePosition();
        // カメラを目的地に向かって指定した時間で達するように移動させる
        transform.position = Vector3.SmoothDamp(
                transform.position, m_DesiredPosition,
                ref m_MoveVelocity, m_DampTime);
    }

    // 戦車の座標の平均値を求める(2 台の場合は中間点)
    private void FindAveragePosition()
    {
        Vector3 averagePos = new Vector3();
        int numTargets = 0;
        for (int i = 0; i < m_Targets.Length; i++)
        {
            // 戦車がアクティブかどうかチェック
            if (!m_Targets[i].gameObject.activeSelf)
                continue;
```

```
            averagePos += m_Targets[i].position;
            numTargets++;
        }
        if (numTargets > 0)
            averagePos /= numTargets;
        // カメラの Y 座標は固定
        averagePos.y = transform.position.y;
        // カメラの移動目標を、戦車の座標の平均値とする
        m_DesiredPosition = averagePos;
    }

    // カメラのズーム
    private void Zoom()
    {
        float requiredSize = FindRequiredSize();
        m_Camera.orthographicSize = Mathf.SmoothDamp(
                m_Camera.orthographicSize, requiredSize,
                ref m_ZoomSpeed, m_DampTime);
    }

    // ズームのサイズを求める
    private float FindRequiredSize()
    {
        // カメラの座標を、ワールド座標からローカル座標に変換する
        Vector3 desiredLocalPos =
                    transform.InverseTransformPoint(m_DesiredPosition);

        float size = 0f;

        for (int i = 0; i < m_Targets.Length; i++)
        {
            // 戦車がアクティブかどうかチェック
            if (!m_Targets[i].gameObject.activeSelf)
                continue;
            // 戦車の座標を、ワールド座標からローカル座標に変換する
            Vector3 targetLocalPos =
                    transform.InverseTransformPoint(m_Targets[i].position);
            // ローカル空間でのカメラと戦車の位置の差
            Vector3 desiredPosToTarget = targetLocalPos - desiredLocalPos;
            // 垂直方向の距離が最大であれば更新
            size = Mathf.Max (size, Mathf.Abs (desiredPosToTarget.y));
            // 水平方向の距離が最大であれば更新
            size = Mathf.Max (size,
                    Mathf.Abs (desiredPosToTarget.x) / m_Camera.aspect);
        }
```

```
            // ズームサイズに余裕を持たせる
            size += m_ScreenEdgeBuffer;
            // ズームサイズの最小値以上になるようにする
            size = Mathf.Max(size, m_MinSize);
            // 結果を返す
            return size;
        }

        // 外部のクラスからカメラの移動とズームを行う
        public void SetStartPositionAndSize()
        {
            FindAveragePosition();
            transform.position = m_DesiredPosition;
            m_Camera.orthographicSize = FindRequiredSize();
        }
    }
```

⑦ CameraRig が選択されている状態で、CameraControl(Script)の Targets の Size を 1 に設定する。

⑧ Elememt0 に、Hierarchy の Tank をドラッグアンドドロップする。

シーンを保存し、プレイしてみる。戦車を移動させると、カメラが移動することを確認する。

⑨ CameraRig が選択されている状態で、CameraControl(Script)の Targets の Size を 2 に変更する。

⑩ Elememt1 に、Hierarchy の Tank(1)をドラッグアンドドロップする。

⑪ 2 台の戦車が重ならないように、Position を調整する。

⑫ Tank(1) が選択されている状態で、Inspector の TankMovement(Script) の PlayerNumber を 2 に変更する。

シーンを保存し、プレイしてみる。2 台目の戦車は、カーソルキーで操作できる。2 台の戦車を移動させると、カメラの移動をズームが変化することを確認する。

4. 体力(HP)

体力を表すスライダーの作成

① Gizmo Display Toggles が Center になっていたら、Pivot に変更する。

② GameObject ▸ UI ▸ Slider を選び、スライダーを作成し、名前を HealthSlider に変更する。

③ Hierarchy の EventSystem を選択し、Inspector の Standalone Input Module の

Horizontal Axis を HorizontalUI、Vertical Axis を VirticalUI に変更する。

④ Hierarchy の Canvas を選択し、Inspector の Canvas Scaler の Reference Pixels per Unit を 1 に変更する。この値は、Canvas 上に配置するスプライトの表示サイズに影響する。

⑤ Inspector の Canvas の Render Mode を World Space に変更する。これによって、Canvas をスクリーン上ではなく、三次元空間内に配置することができる。

⑥ Hierarchy で、Canvas を Tank にドラッグアンドドロップし、子オブジェクトにする。

⑦ Canvas が選択されている状態で、Rect Transform の Position を(0, 0.1, 0)、Width を 3.5、Height も 3.5、Rotation を(90, 0, 0)に設定する。

⑧ Hierarchy で、Canvas の下の HealthSlider の下の Handle Slide Area を削除する。

⑨ HealthSlider、Background、Fill Area、Fill を同時に選択し、Inspector の Anchor Presets を開いて、右下のアイコンを Alt キーを押しながらクリックし、水平・垂直とも Stretch に設定する。

⑩ HealthSlider が選択されている状態で、Inspector の Interactable を無効に、Transition を None に、Max Value を 100 に変更する。

⑪ BackGround が選択されている状態で、Inspector の Image の Image Source に Health Wheel を選択し、Color の A の値を 80 に設定する。

⑫ Fill が選択されている状態で、Inspector の Image の Image Source に Health Wheel を選択し、Color の A の値を 150、Image Type を Filled、Fill Origin を Left、Clockwize を無効に設定する。

⑬ Sprites¥UI フォルダーの UIDirectionControl.cs を Hierarchy の HealthSlider にドラッグアンドドロップする。

⑭ Tank が選択されている状態で、Inspector の上の方にある Apply をクリックして、戦車のプレハブを更新する。

戦車が破壊されたときのエフェクト

⑮ Prefabs フォルダーの TankExplosion を Hierarchy にドラッグアンドドロップする。

⑯ TankExplosion が選択されている状態で、AudioSource コンポーネントを追加し、AudioClip に TankExplosion を選び、Play On Awake を無効にする。

⑰ Apply ボタンをクリックして TankExplosion プレハブを更新し、Hierarchy の TankExplosion を削除する。

スクリプトの追加と編集

⑱ Scripts¥Tank フォルダーの TankHealth.cs を Hierarchy の Tank にドラッグアンドドロップし、次のように編集する。

```
using UnityEngine;
using UnityEngine.UI;
```

```
public class TankHealth : MonoBehaviour
{
    public float m_StartingHealth = 100f;               // 体力の初期値
    public Slider m_Slider;                             // スライダー
    public Image m_FillImage;                           // スライダーの画像
    public Color m_FullHealthColor = Color.green;       // 体力の初期値の色
    public Color m_ZeroHealthColor = Color.red;         // 体力が 0 のときの色
    public GameObject m_ExplosionPrefab;                // 爆発のプレハブ

    private AudioSource m_ExplosionAudio;               // 爆発音の AudioSource
    private ParticleSystem m_ExplosionParticles;        // 爆発のパーティクルシステム
    private float m_CurrentHealth;                      // 現在の体力
    private bool m_Dead;                                // 戦車が破壊されたら true

    private void Awake()
    {
        // 爆発のプレハブをインスタンス化し、パーティクルシステムを取得する
        m_ExplosionParticles =
            Instantiate(m_ExplosionPrefab).GetComponent<ParticleSystem>();
        // AudioSource を取得する
        m_ExplosionAudio = m_ExplosionParticles.GetComponent<AudioSource>();
        // パーティクルシステムを無効化し、すぐに爆発が起こらないようにする
        m_ExplosionParticles.gameObject.SetActive(false);
    }

    private void OnEnable()
    {
        // 体力の初期値をセット
        m_CurrentHealth = m_StartingHealth;
        m_Dead = false;
        // スライダーの更新
        SetHealthUI();
    }

    // ダメージを受けたときの処理
    public void TakeDamage(float amount)
    {
        // 体力を減らす
        m_CurrentHealth -= amount;
        // スライダーの更新
        SetHealthUI();
        // 体力が 0 以下になった時の処理
        if (m_CurrentHealth <= 0f && !m_Dead)
        {
            OnDeath();
        }
```

```
    }

    // スライダーの表示の更新
    private void SetHealthUI()
    {
        // スライダーに体力の値をセット
        m_Slider.value = m_CurrentHealth;
        // スライダーの色を、体力によって 2 色間を補間した値とする
        m_FillImage.color = Color.Lerp(m_ZeroHealthColor, m_FullHealthColor,
                                       m_CurrentHealth / m_StartingHealth);
    }

    // 戦車が破壊されたときの処理
    private void OnDeath()
    {
        m_Dead = true;
        m_ExplosionParticles.transform.position = transform.position;
        m_ExplosionParticles.gameObject.SetActive(true);
        m_ExplosionParticles.Play();
        m_ExplosionAudio.Play();
        gameObject.SetActive(false);
    }
}
```

⑲ Tank が選択されている状態で、Hierarchy の TankHealth(Script)の Slider に HealthSlider、FillImge に Fill、ExplosionPrefab に TankExplosion を指定する。

⑳ Apply ボタンをクリックして Tank プレハブを更新する。

5. 砲弾

砲弾の作成

① Models フォルダーの Shell を Hierarchy にドラッグアンドドロップする。

② Shell に Capsule Collider を追加し、Is Trigger を有効、Direction を Z-Axis、Center を(0, 0, 0.2)、Radius を 0.15、Height を 0.55 に設定する。

③ Shell に RigidBody を追加する。

④ Prefabs フォルダーの ShellExplosion を Hierarchy の Shell にドラッグアンドドロップして、子オブジェクトにする。

⑤ ShellExplosion に AudioSource を追加し、AudioClip に ShellExplosion を指定し、Play On Awake を無効にする。

⑥ Shell が選択されている状態で、Add Component ▸ Rendering ▸ Light を選び、Light コンポーネントを追加する。

スクリプトの追加と編集

⑦ Scripts¥Shell フォルダーの ShellExplosion.cs を Hierarchy の Shell にドラッグアンドドロップし、次のように編集する。

```
using UnityEngine;

public class ShellExplosion : MonoBehaviour
{
    public LayerMask m_TankMask;                    // 爆発に影響を受けるレイヤー
    public ParticleSystem m_ExplosionParticles;     // 爆発のパーティクルシステム
    public AudioSource m_ExplosionAudio;            // 爆発音の AudioSource
    public float m_MaxDamage = 100f;                // ダメージの最大値
    public float m_ExplosionForce = 1000f;          // 爆発の力
    public float m_MaxLifeTime = 2f;                // 砲弾が消滅するまでの時間
    public float m_ExplosionRadius = 5f;            // 爆発の影響を受ける距離

    private void Start()
    {
        // m_MaxLifeTime 秒後に消滅する
        Destroy(gameObject, m_MaxLifeTime);
    }

    // 砲弾が何かに衝突したときの処理
    private void OnTriggerEnter(Collider other)
    {
        // 砲弾を中心とする球に接触する戦車のコライダーを得る
        Collider[] colliders = Physics.OverlapSphere(transform.position,
                                        m_ExplosionRadius, m_TankMask);

        for (int i = 0; i < colliders.Length; i++)
        {
            // コライダーから Rigidbody を取得する
            Rigidbody targetRigidbody = colliders[i].GetComponent<Rigidbody>();
            // Rigidbody の存在チェック
            if (!targetRigidbody)
                continue;
            // RigidBody に爆発の力を加える
            targetRigidbody.AddExplosionForce(m_ExplosionForce,
                            transform.position, m_ExplosionRadius);
            // TankHealth クラスへの参照を得る
            TankHealth targetHealth =
                    targetRigidbody.GetComponent<TankHealth>();
            // TankHealth の存在チェック
            if (!targetHealth)
                continue;
```

```
                // 戦車が受けるダメージ量の計算
                float damage = CalculateDamage(targetRigidbody.position);

                // 戦車にダメージを与える（TankHelth クラスのメソッド）
                targetHealth.TakeDamage(damage);
            }
            // 砲弾とパーティクルシステムの親子関係の解除
            m_ExplosionParticles.transform.parent = null;
            // 爆発のパーティクルシステムを動かす
            m_ExplosionParticles.Play();
            // 爆発音を鳴らす
            m_ExplosionAudio.Play();
            // パーティクルシステムの再生終了時に削除
            ParticleSystem.MainModule mainModule = m_ExplosionParticles.main;
            Destroy(m_ExplosionParticles.gameObject, mainModule.duration);
            // 砲弾の削除
            Destroy(gameObject);
        }

        // 戦車が受けるダメージ量の計算
        private float CalculateDamage(Vector3 targetPosition)
        {
            // 砲弾の位置から戦車の位置けのベクトル
            Vector3 explosionToTarget = targetPosition - transform.position;
            // 砲弾と戦車の距離
            float explosionDistance = explosionToTarget.magnitude;
            // 砲弾と戦車の距離と爆発の影響が及ぶ最大距離との割合
            float relativeDistance = (m_ExplosionRadius - explosionDistance)
                                                / m_ExplosionRadius;
            // 受けるダメージ
            float damage = relativeDistance * m_MaxDamage;
            // ダメージが負の値にならないようにする
            damage = Mathf.Max(0f, damage);
            // ダメージ量を返す
            return damage;
        }
    }
```

⑧ Shell が選択されている状態で、Inspector の ShellExplosion(Script)の TankMask を Players 、ExplosionParticles を ShellExplosion 、ExplosionAudio を ShellExplosion に設定する。

砲弾のプレハブ化

⑨ Hierarchy の Shell を Prefabs フォルダーにドラッグアンドドロップしてプレハブ化し、Hierarchy の Shell を削除する。

6. シューティング

砲弾の出現ポイントの作成

① Tank が選択されている状態で、GameObject ▸ Create Empty Child を選び、作成された子オブジェクトの名前を FireTransform に変更する。

② FireTransform の Position を(0, 1.7, 1.35)、Rotation を(350, 0, 0)に設定する。

砲弾の発射方向と強さを示すスライダーの作成

③ Canvas が選択されている状態で、GameObject ▸ UI ▸ Slider を選んで Slider を作成し、名前を AimSlider に変更する。

④ AimSlider の子オブジェクトの Background と Handle Slide Area を削除する。

⑤ AimSlider が選択されている状態で、Inspector の Slider の Interactable を無効、Transition を None、Direction を Bottom To Top、Min Value を 15、Max Value を 30 に設定する。

⑥ AimSlider と Fill Area を同時に選択し、Inspector の Anchor Presets を開いて、右下のアイコンを Alt キーを押しながらクリックして、水平・垂直とも Stretch に設定する。

⑦ Fill が選択されている状態で、Inspector の Rect Transform の Height を 0 に、Image の Source Image を AimArrow に設定する。

⑧ AimSlider が選択されている状態で、Inspentor の Rect Transform の Left を 1、Top、を-9、PosZ を-1、Right を 1、Bottom を3に設定する。

スクリプトの追加と編集

⑨ Scripts¥Tank フォルダーの TankShooting.cs を Hierarchy の Tank にドラッグアンドドロップし、次のように編集する。

```
using UnityEngine;
using UnityEngine.UI;

public class TankShooting : MonoBehaviour
{
    public int m_PlayerNumber = 1;              // プレーヤー番号
    public Rigidbody m_Shell;                   // 砲弾のプレハブ.
    public Transform m_FireTransform;           // 砲弾の出現場所
    public Slider m_AimSlider;                  // 砲弾を撃つ方向を示すスライダー
    public AudioSource m_ShootingAudio;         // 砲弾の発射音の AudioSource
    public AudioClip m_ChargingClip;            // チャージ音の AudioClip.
    public AudioClip m_FireClip;                // 発射音の AudioClip.
    public float m_MinLaunchForce = 15f;        // チャージ最小のときに砲弾に加える力
    public float m_MaxLaunchForce = 30f;        // チャージ最大のときに砲弾に加える力
    public float m_MaxChargeTime = 0.75f;       // チャージ最大になるのにかかる時間

    private string m_FireButton;                // 砲弾発射に使用される入力軸名
```

```
private float m_CurrentLaunchForce;          // 砲弾に加える力.
private float m_ChargeSpeed;                 // チャージの増加速度.
private bool m_Fired;                        // Fire フラグ:砲弾が発射されたら true.

private void OnEnable()
{
    // 砲弾に加える力の初期値は最小値とする
    m_CurrentLaunchForce = m_MinLaunchForce;
    m_AimSlider.value = m_MinLaunchForce;
}

private void Start()
{
    // Fire の軸名をプレーヤー番号を加えたものに
    m_FireButton = "Fire" + m_PlayerNumber;
    // チャージ速度の計算.
    m_ChargeSpeed = (m_MaxLaunchForce - m_MinLaunchForce)
                                / m_MaxChargeTime;
}

private void Update()
{
    // スライダーの値を初期値にする
    m_AimSlider.value = m_MinLaunchForce;
    // 砲弾に加える力が上限値を超え、且つ砲弾がまだ発射されていないとき
    if (m_CurrentLaunchForce >= m_MaxLaunchForce && !m_Fired)
    {
        // 砲弾に加える力を上限値に抑える
        m_CurrentLaunchForce = m_MaxLaunchForce;
        // 砲弾発射
        Fire();
    }
    // Fire ボタンが押されたとき.
    else if (Input.GetButtonDown(m_FireButton))
    {
        // Fire フラグと砲弾に加える力をリセット.
        m_Fired = false;
        m_CurrentLaunchForce = m_MinLaunchForce;
        // チャージ音を鳴らす.
        m_ShootingAudio.clip = m_ChargingClip;
        m_ShootingAudio.Play();
    }
    // Fire ボタンが押され続け、且つ砲弾がまだ発射されていないとき
    else if (Input.GetButton(m_FireButton) && !m_Fired)
    {
        // 砲弾に加える力を増やす
```

```
            m_CurrentLaunchForce += m_ChargeSpeed * Time.deltaTime;
            // スライダー更新
            m_AimSlider.value = m_CurrentLaunchForce;
        }
        // Fire ボタンが離され、且つ砲弾がまだ発射されていないとき
        else if (Input.GetButtonUp(m_FireButton) && !m_Fired)
        {
            // 砲弾発射
            Fire();
        }
    }

    // 砲弾の発射
    private void Fire()
    {
        // Fire フラグをセットし、複数回実行されないようにする
        m_Fired = true;
        // 砲弾のプレハブのインスタンス化
        Rigidbody shellInstance =
            Instantiate(m_Shell, m_FireTransform.position,
                        m_FireTransform.rotation) as Rigidbody;
        // 砲弾に速度を与える
        shellInstance.velocity = m_CurrentLaunchForce * m_FireTransform.forward;
        // 発射音を鳴らす.
        m_ShootingAudio.clip = m_FireClip;
        m_ShootingAudio.Play();
        // 砲弾に加える力のリセット
        m_CurrentLaunchForce = m_MinLaunchForce;
    }
}
```

⑩ Tank が選択されている状態で、Inspector の TankShooting(Script)の Shell を Shell プレハブに、FireTransform を Hierarchy の FireTransform に、AimSlider を Hierarchy の AimSlider に、ShootingAudio を Tank の 2 番目の AudioSource に、ChargingClip を ShotCharging に、FireClip を ShotFiring に設定する。

⑪ Apply ボタンをクリックして Tank プレハブを更新し、Hierarchy の Tank を削除する。

7. ゲームマネージャー

戦車の出現ポイントの作成

① Empty GameObject を作成して、名前を SpawnPoint1 に変更し、Position を(-3, 0, 30)、Rotation を(0, 180, 0)に設定する。

② Empty GameObject を作成して、名前を SpawnPoint2 に変更し、Position を(13, 0,

-5)、Rotation を(0, 0, 0)に設定する。

メッセージ表示領域の作成

③ GameObject ▸ UI ▸ Text を選び、Canvas と Text を作成し、Canvas の名前を MessageCanvas に変更する。

④ Scene の 2D ボタンをクリックし、MessageCanvas が選択されている状態でマウスカーソルを Scene 上に置き、F キーを押す。

⑤ Text が選択されている状態で、Inspector の Rect Transform の Anchors の Min を X・Y とも 0.1 に、Max を X・Y とも 0.1 に、Left・Top・PosZ・Right・Bottom をすべて 0 に設定する。

⑥ Text の Text を TANKS!に、Font を BowlbyOne-Regular に、Alignment を Center と Middle に、Best Fit を有効に、Max Size を 60 に、Color を白に設定する。

⑦ Add Component のサーチバーに shadow と入力し、Shadow を追加して、Effect Color を(114, 71, 40, 128)に、Effect Distance を(-3, -3)に設定する。

スクリプトの追加と編集

⑧ 2D ボタンをクリックして 2D モードを解除し、CameraRig を選択して、マウスカーソルが Scene 上にある状態で F キーを押す。

⑨ CameraControl(Script)の Targets の Size を 0 にする。

⑩ Empty GameObject を作成して、名前を GameManager に変更し、Scripts¥Managers フォルダーの GameManager.cs をドラッグアンドロップし、次のように編集する。

```
using UnityEngine;
using System.Collections;
using UnityEngine.SceneManagement;
using UnityEngine.UI;

public class GameManager : MonoBehaviour
{
    public int m_NumRoundsToWin = 5;    // ゲームに勝利するために必要なスコア
    public float m_StartDelay = 3f;         // Starting から Playing に移行するまでの時間
    public float m_EndDelay = 3f;           // Playing から Ending に移行するまでの時間
    public CameraControl m_CameraControl;       // CameraControl スクリプトへの参照
    public Text m_MessageText;                  // 表示する文字列
    public GameObject m_TankPrefab;             // 戦車のプレハブ
    public TankManager[] m_Tanks;       // TankManager スクリプトの配列への参照

    private int m_RoundNumber;                  // 現在のラウンド番号
    private WaitForSeconds m_StartWait;         // Starting から Playing へのタイマー
    private WaitForSeconds m_EndWait;           // Playing から Ending へのタイマー
    private TankManager m_RoundWinner;          // 現在のラウンドの勝者への参照
```

```
private TankManager m_GameWinner;           // ゲームの勝者への参照

private void Start()
{
    // タイマー作成
    m_StartWait = new WaitForSeconds(m_StartDelay);
    m_EndWait = new WaitForSeconds(m_EndDelay);
    // 戦車の出現
    SpawnAllTanks();
    // カメラに戦車の位置情報を渡す
    SetCameraTargets();
    // ゲームループの開始
    StartCoroutine(GameLoop());
}

// 戦車の出現
private void SpawnAllTanks()
{
    // 戦車の台数分繰り返す
    for (int i = 0; i < m_Tanks.Length; i++)
    {
        // 戦車のプレハブをインスタンス化
        m_Tanks[i].m_Instance = Instantiate(m_TankPrefab,
            m_Tanks[i].m_SpawnPoint.position,
            m_Tanks[i].m_SpawnPoint.rotation) as GameObject;
        // プレーヤー番号をセット
        m_Tanks[i].m_PlayerNumber = i + 1;
        // TankManager スクリプトの Setup を実行
        m_Tanks[i].Setup();
    }
}

// カメラに戦車の位置情報を渡す
private void SetCameraTargets()
{
    // Transform の配列を戦車の台数分作成
    Transform[] targets = new Transform[m_Tanks.Length];

    for (int i = 0; i < targets.Length; i++)
    {
        // 戦車の Transform 情報を配列に入れる
        targets[i] = m_Tanks[i].m_Instance.transform;
    }
    // CameraControl スクリプトに Transform 情報をセット
    m_CameraControl.m_Targets = targets;
}
```

```
// ゲームループ
private IEnumerator GameLoop()
{
    // RoundStarting の実行が終了するまで戻らない
    yield return StartCoroutine(RoundStarting());
    // RoundPlaying の実行が終了するまで戻らない
    yield return StartCoroutine(RoundPlaying());
    // RoundEnding の実行が終了するまで戻らない
    yield return StartCoroutine(RoundEnding());

    // ゲームの勝者が決定した場合
    if (m_GameWinner != null)
    {
        // シーンの最初に戻る
        SceneManager.LoadScene(0);
    }
    // ゲームの勝者がまだ決定していない場合
    else
    {
        // 現在の GameLoop を終了し、GameLoop の最初に戻る
        StartCoroutine(GameLoop());
    }
}

// ラウンドの開始処理
private IEnumerator RoundStarting()
{
    // すべての戦車を初期化し、動かせなくする
    ResetAllTanks();
    DisableTankControl();
    // CameraControl スクリプトの SetStartPositionAndSize を実行する
    m_CameraControl.SetStartPositionAndSize();
    // ラウンド番号の更新
    m_RoundNumber++;
    // ラウンド番号の表示
    m_MessageText.text = "ROUND " + m_RoundNumber;
    // 3 秒間待つ
    yield return m_StartWait;
}

// プレイ処理
private IEnumerator RoundPlaying()
{
    // 戦車を動かせるようにする
    EnableTankControl();
    // 文字列表示を消す
```

```
        m_MessageText.text = string.Empty;
        // 生き残った戦車が 1 台以下になるまで繰り返す
        while (!OneTankLeft())
        {
            // 次のフレームで戻る
            yield return null;
        }
    }

    // ラウンドの終了処理
    private IEnumerator RoundEnding()
    {
        // 戦車を動かせなくする
        DisableTankControl();
        // 前のラウンドの勝者をクリア
        m_RoundWinner = null;
        // ラウンドの勝者を得る
        m_RoundWinner = GetRoundWinner();
        // 勝者がいたら、スコアを増やす
        if (m_RoundWinner != null)
            m_RoundWinner.m_Wins++;
        // ゲームの勝者が決定したかどうか調べ、決定したら勝者を得る
        m_GameWinner = GetGameWinner();
        // 結果を文字列で表示する
        string message = EndMessage();
        m_MessageText.text = message;
        // 3 秒間待つ
        yield return m_EndWait;
    }

    // 残っている戦車が 1 台以下かどうか調べる
    private bool OneTankLeft()
    {
        // 残っている戦車の台数
        int numTanksLeft = 0;
        // 戦車の台数回繰り返す
        for (int i = 0; i < m_Tanks.Length; i++)
        {
            // アクティブな戦車の台数を数える
            if (m_Tanks[i].m_Instance.activeSelf)
                numTanksLeft++;
        }
        // 残っている戦車が 1 台以下なら true、それ以外なら false を返す
        return numTanksLeft <= 1;
    }
```

```
// ラウンドの勝者を返す（残っている戦車が 1 台以下のとき呼ばれる）
private TankManager GetRoundWinner()
{
    // すべての戦車を調べる
    for (int i = 0; i < m_Tanks.Length; i++)
    {
        // 戦車がアクティブならば、TankManager を返す
        if (m_Tanks[i].m_Instance.activeSelf)
            return m_Tanks[i];
    }
    // アクティブな戦車がない場合（相打ち）、null を返す
    return null;
}

// ゲームの勝者を返す（未決定ならば、null を返す）
private TankManager GetGameWinner()
{
    // すべての戦車を調べる
    for (int i = 0; i < m_Tanks.Length; i++)
    {
        // ラウンドの勝利数が 5 ならば、TankManager を返す
        if (m_Tanks[i].m_Wins == m_NumRoundsToWin)
            return m_Tanks[i];
    }
    // 勝利数が 5 に達したプレーヤーがいない場合、null を返す
    return null;
}

// ラウンド終了時に表示する文字列を返す
private string EndMessage()
{
    // 文字列の初期値は、"DRAW!"（引き分け）
    string message = "DRAW!";
    // 勝者がいる場合の文字列は、"PLAYER 1/2 WINS THE ROUND!"
    if (m_RoundWinner != null)
        message = m_RoundWinner.m_ColoredPlayerText
                  + " WINS THE ROUND!";
    // 改行 5 回を追加
    message += "¥n¥n¥n¥n¥n";
    // 各プレーヤーのラウンド勝利数の表示を追加
    for (int i = 0; i < m_Tanks.Length; i++)
    {
        message += m_Tanks[i].m_ColoredPlayerText + ": "
                   + m_Tanks[i].m_Wins + " WINS¥n";
    }
    // ゲームの勝者が決定した場合、"PLAYER 1/2 WINS THE GAME!"
```

```
        if (m_GameWinner != null)
            message = m_GameWinner.m_ coloredPlayerText + " WINS THE GAME!";
        // 文字列を返す
        return message;
    }

    // すべての戦車を初期化する
    private void ResetAllTanks()
    {
        for (int i = 0; i < m_Tanks.Length; i++)
        {
            m_Tanks[i].Reset();
        }
    }

    // すべての戦車を動かせるようにする
    private void EnableTankControl()
    {
        for (int i = 0; i < m_Tanks.Length; i++)
        {
            m_Tanks[i].EnableControl();
        }
    }

    // すべての戦車を動かせなくする
    private void DisableTankControl()
    {
        for (int i = 0; i < m_Tanks.Length; i++)
        {
            m_Tanks[i].DisableControl();
        }
    }
}
```

⑪ GameManager が選択されている状態で、Inspector の GameManager (script)の CameraControl を CameraRig、MessageText を Text、TankPrefab を Prefabs フォルダーの Tank、Tanks の Size を 2、Elememt0 の Color を (42, 100, 178) 、SpawnPoint を SpawnPoint1、Elememt1 の Color を(229, 46, 40)、SpawnPoint を SpawnPoint2 に設定する。

TankManager クラス

⑫ TankManager クラスは、個別の戦車を管理するものである。GameManager は、TankManager クラスを利用することにより、すべての戦車を管理している。以下に TankManager のスクリプトを示す。

```
using System;
using UnityEngine;

[Serializable]        // Unity Editor で GameManager の Tanks の設定のために必要
public class TankManager
{
    public Color m_PlayerColor;                                 // 戦車と文字の色
    public Transform m_SpawnPoint;                              // 戦車の位置の初期値
    [HideInInspector] public int m_PlayerNumber;                // プレーヤー番号
    [HideInInspector] public string m_ColoredPlayerText;        // 色指定入り文字列
    [HideInInspector] public GameObject m_Instance;             // インスタンス化された戦車
    [HideInInspector] public int m_Wins;                        // 勝利したラウンド数

    private TankMovement m_Movement;            // TankMovement クラスへの参照
    private TankShooting m_Shooting;            // TankShooting クラスへの参照
    private GameObject m_CanvasGameObject;      // Tank の子オブジェクト Canvas

    // 戦車の初期化
    public void Setup()
    {
        m_Movement = m_Instance.GetComponent<TankMovement>();
        m_Shooting = m_Instance.GetComponent<TankShooting>();
        m_CanvasGameObject =
                m_Instance.GetComponentInChildren<Canvas>().gameObject;
        // 外部スクリプトのプレーヤー番号をセット
        m_Movement.m_PlayerNumber = m_PlayerNumber;
        m_Shooting.m_PlayerNumber = m_PlayerNumber;
        // プレーヤー名の文字列を色指定付のリッチテキスト形式で作成
        m_ColoredPlayerText = "<color=#" +
                ColorUtility.ToHtmlStringRGB(m_PlayerColor) +
                ">PLAYER " + m_PlayerNumber + "</color>";
        // Tank の子オブジェクトの MeshRenderer を取得（複数ある）
        MeshRenderer[] renderers =
                m_Instance.GetComponentsInChildren<MeshRenderer>();
        // 戦車の色を変える
        for (int i = 0; i < renderers.Length; i++)
        {
            renderers[i].material.color = m_PlayerColor;
        }
    }

    // 戦車のコントロールの無効化
    public void DisableControl()
    {
        // TankMovement スクリプトを無効化
        m_Movement.enabled = false;
```

```
        // TankShooting スクリプトを無効化
        m_Shooting.enabled = false;
        //  体力スライダーを無効化
        m_CanvasGameObject.SetActive(false);
    }

    //  戦車のコントロールの有効化
    public void EnableControl()
    {
        // TankMovement スクリプトを有効化
        m_Movement.enabled = true;
        // TankShooting スクリプトを有効化
        m_Shooting.enabled = true;
        //  体力スライダーを有効化
        m_CanvasGameObject.SetActive(true);
    }

    //  戦車のリセット
    public void Reset()
    {
        //  戦車の位置と向きを初期化
        m_Instance.transform.position = m_SpawnPoint.position;
        m_Instance.transform.rotation = m_SpawnPoint.rotation;
        //  戦車をリセット
        m_Instance.SetActive(false);
        m_Instance.SetActive(true);
    }
}
```

参考）StartCoroutine の使用例

例 1

```
    void Start ()
    {
        ・・・・・・・・・・
        StartCoroutine (Sample(1f));
        // Sample の終了を待たず、次の行に移る
        ・・・・・・・・・・
    }

    private IEnumerator Sample(float interval)
    {
        ・・・・・・・・・・
        yield return new WaitForSeconds (interval);
```

```
        // 1 秒後に次の行に移る
        ・・・・・・・・・・
    }
```

例 2

```
    void Start ()
    {
        ・・・・・・・・・・
        StartCoroutine (Sample1( ));
        // Sample の終了を待たず、次の行に移る
        ・・・・・・・・・・
    }

    private IEnumerator Sample1( )
    {
        ・・・・・・・・・・
        yield return StartCoroutine (Sample2( ));
        // Sample2 の実行終了後に次の行に移る
        ・・・・・・・・・・
    }

    private IEnumerator Sample2( )
    {
        ・・・・・・・・・・
        yield return WaitForSeconds (1f);
        // 1 秒後に次の行に移る
        ・・・・・・・・・・
    }
```

例 3

```
    void Start ()
    {
        ・・・・・・・・・・
        StartCoroutine (Sample( ));
        // Sample の終了を待たず、次の行に移る
        ・・・・・・・・・・
    }

    private IEnumerator Sample( )
    {
        ・・・・・・・・・・
        //  実行中の Sample を終了し、Sample をリスタートする
        StartCoroutine (Sample( ));
        ・・・・・・・・・・
    }
```

8. オーディオ

BGM の追加

① GameManager に AudioSource を追加し、AudioClip に BackGroundMusic を指定、Loop を有効にする。

Audio Mixer の作成

② Assets の下に AudioMixers フォルダーを作成し、そこで右クリックして、Create ▸ Audio Mixer を選び、作成された Audio Mixer の名前を MainMix に変更する。

③ MainMix が選択されている状態で、Window ▸ Audio Mixer を選び、Audio Mixer Window を開く。

④ Groups の+アイコンをクリックして Group を追加し、名前を Music に変更する。

⑤ Master を選択し、Groups の+アイコンをクリックして Group を追加し、名前を SFX に変更する。

⑥ Master を選択し、Groups の+アイコンをクリックして Group を追加し、名前を Driving に変更する。

⑦ Prefabs フォルダーの Tank を選択し、最初の AudioSource の Output を Driving(MainMix)に、二番目の AudioSource の Output を SFX(MainMix)に設定する。

⑧ Prefabs フォルダーの Shell を展開して ShellExplosion を選択し、AudioSource の Output を SFX(MainMix)に設定する。

⑨ Hierarchy の GameManager を選択し、AudioSource の Output を Music(MainMix)に設定する。

音量バランスの調整

⑩ Audio Mixer Window に戻り、Music を-12、Driving を-25 に設定する。

Duck Volume の追加と調整

⑪ Music Group を選択して、Add.. ▸ Duck Volume を選ぶ。

⑫ SFX Group を選択して、Add.. ▸ Send を選び、Inspector の Send の Receive を Music¥Duck Volume に、Send Level を 0dB(減衰なし)に設定する。

⑬ 再び Music Group を選択して、Inspector の Duck Volume の Threshold を-46、Ratio を 250%、Attack Time を 0ms に設定する。この結果、SFX の音量が-46db を超えた瞬間に Music の音量の減衰係数が 2.5 倍になる。