ポートフォリオ

総合学園ヒューマンアカデミー横浜校

ゲームカレッジプログラマー専攻　2年

能登 巧実

2025/6/29

目次

スキルシート

ノト　タクミ
能登　巧実

性別：男	年齢：24歳
希望職種：プログラマー	

自己PR

前職では1年半ほどITエンジニアとしてツールの改修作業やAWSの運用を行っていました。改修作業を行うことが多かったため、人のコードを読むのが得意です。
大学の頃からプログラムが好きだったため、もっとプログラムに触れることができるゲームプログラマーを志望いたします。

好きなゲーム

主にプレイするジャンルは
アクション系です。
アクション系に疲れたときに
RPG系をやります。

保有スキル

開発環境

開発使用ツール

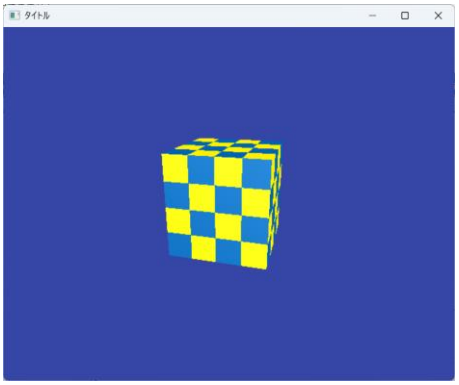
言語

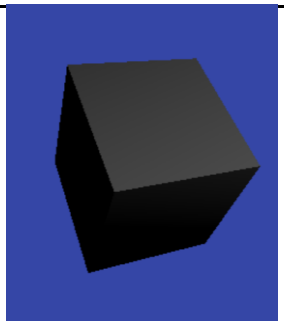
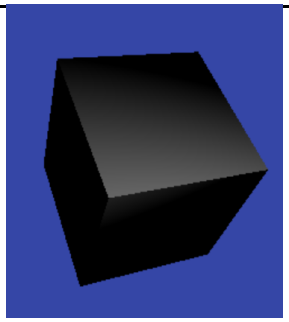
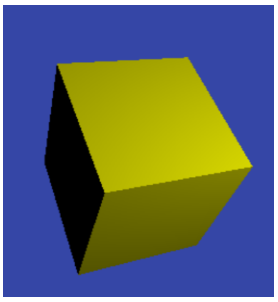
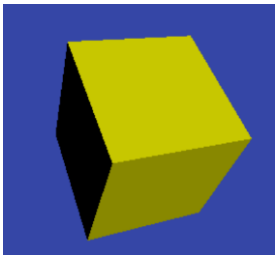
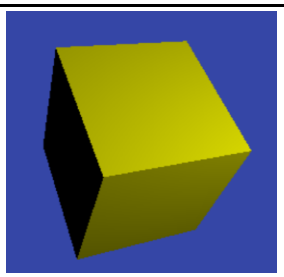
保有資格

Direct3Dゲームエンジン開発

HLSLシェーダープログラミング

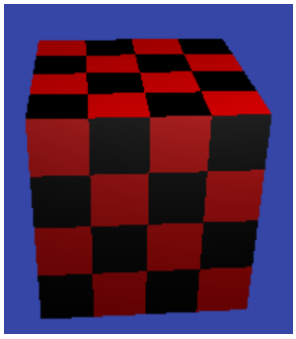
Microsoft社のDirect3Dのパイプラインを参考に
正方形を出力させるようにいたしました。
シェーダプログラミングを勉強するため、
HLSLを用いて表示している正方形に
Phong shadingを実装いたしました。



	平行光源	点光源
Specular		
Diffuse		
Specular＋Diffuse		

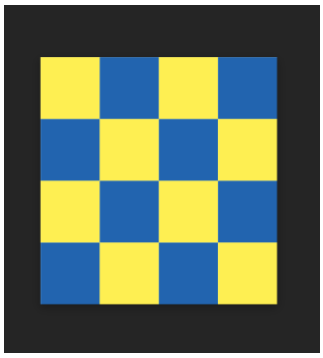
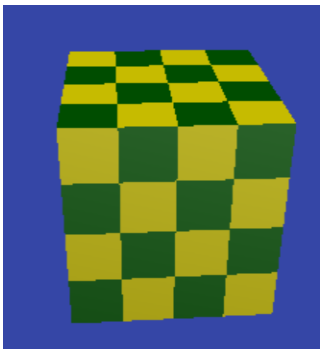
RAWデータの実装

オブジェクトが単色のため直接RAWデータとして色の情報を直接代入したことでオブジェクトに色を実装いたしました。



テクスチャの実装

PNG形式のファイルを読み込んで表示させたところ、テクスチャの色の値にDiffuseの値をかけているためか色のずれが発生しているため
今後はこちらの問題を解決していきたいと思います。

実装したPNG画像	実装結果
 ※4 x 4pxで作成しているため拡大してます	

```
return float4(texel.rgb * diffuseColor + specularColor, MaterialDiffuse.a * texel.a);
```

今後の課題

今後の課題として上記で記載した、PNG形式で実装した際に色がずれてしまう問題とPNG形式のファイルをDDS変換して呼び出す機能を実装していきます。

カスタムロボットゲーム

作品名：リビルディングマン

開発環境	Unity 6(6000.0.36f1)
使用ツール	Microsoft Visual Studio 2022
動作環境	PC(コントローラ/キーボード)
制作期間	3ヶ月(2025年3月～5月末)
制作人数	8人 プランナー　2人 プログラマー　3人 グラフィッカー　3人
担当役職	メインプログラマー

Unity URP

ゲーム概要

『壊して作る』をコンセプトとした、ステージにいるロボットを倒して、
倒した部品からロボットを作成して味方を増やし、
攻略していくゲームとなっています。
作成する部品によって行動パターンや攻撃方法が変わっていくので欲しい

担当箇所

・NPCの行動パターン作成
・ロボットのHP管理
・プレイヤーの攻撃、アイテム取得
・UI
・シーン管理

この作品を通して

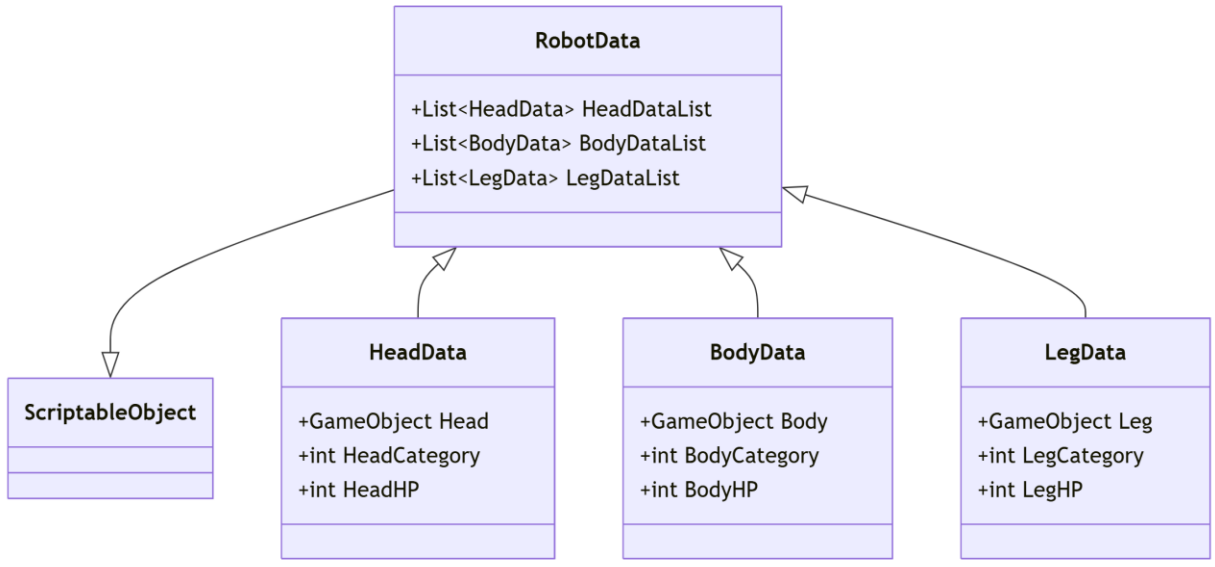
この制作は私がメインプログラマーとして初めての制作でした。
そのため、プログラマーのメンバーのタスクの振り分けやスケジュール管理をする必要があり、思うように行かなかったところもありましたが良い経験をすることができました。
このゲームは頭や胴によって行動パターンや攻撃方法が変わる仕様上、データの管理や参照に気をつける必要がありました。
また、Navmeshを使用しているためか処理が重くなってしまったため
他の箇所で効率化を行うなどして軽量化を実施していきました。
それでも処理が重かったため、今後も効率化を行うことで
軽量化を実施していきます。

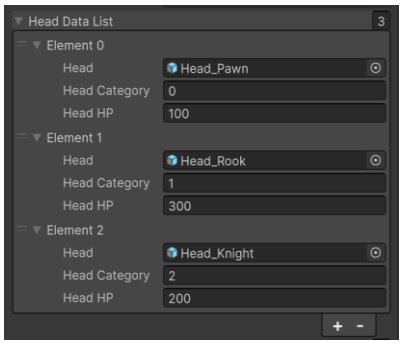
●ロボットのHP管理

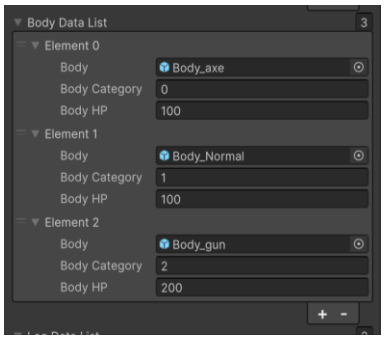
データの管理

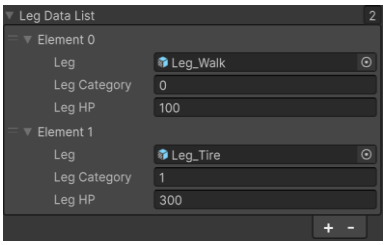
頭、胴、足それぞれにHPがあることとロボットを作成できることから
必要な時にデータを呼び出せるようにする必要があったため、
ScriptiableObjectを使用しデータの管理を行いました。

データ管理構成









RobotData.cs

```
[CreateAssetMenu(menuName = "ScriptableObject/Robot Setting", fileName = "RobotSetting")]
Unity スクリプト | 3 個の参照
public class RobotData : ScriptableObject
{
    public List<HeadData> HeadDataList;
    public List<BodyData> BodyDataList;
    public List<LegData> LegDataList;

}
```

RobotData.cs

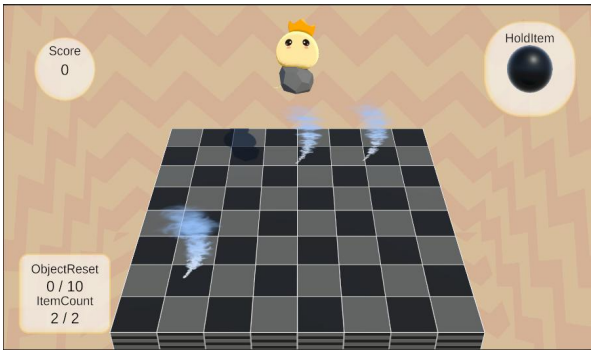
```
[Serializable]
1 個の参照
public class HeadData
{
    public GameObject Head;
    public int HeadCategory;
    public int HeadHP;
}
[Serializable]
1 個の参照
public class BodyData
{
    public GameObject Body;
    public int BodyCategory;
    public int BodyHP;
}
[Serializable]
1 個の参照
public class LegData
{
    public GameObject Leg;
    public int LegCategory;
    public int LegHP;
}
```

反省点と課題

クラス名でHeadDataと記載しているからHeadのことだと分かるのに
すべての値にHeadCategoryなどといった形でHeadと記載しているため
今後、登録するデータ量が増えた際にすべてにこのようなことをしていては
管理が難しくなると考えたため
今後はHeadCategoryの場合はCategoryなどといった形で
管理をしやすくしていきます。

磁石落としゲーム

作品名：Donect

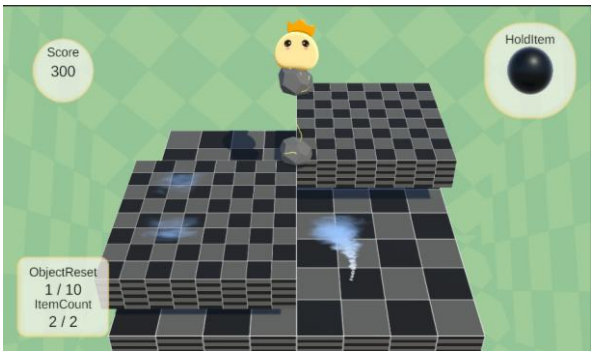


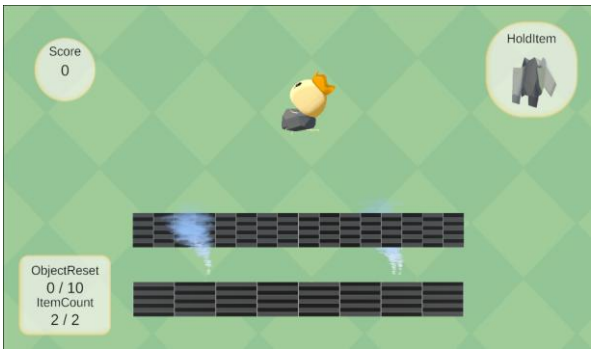
開発環境	Unity 2022.3.49f1
使用ツール	Microsoft Visual Studio 2022
使用ツール	Microsoft Visual Studio 2022
動作環境	PC(コントローラ/キーボード)
制作期間	2週間(2025年1月～1月末)
制作人数	1人

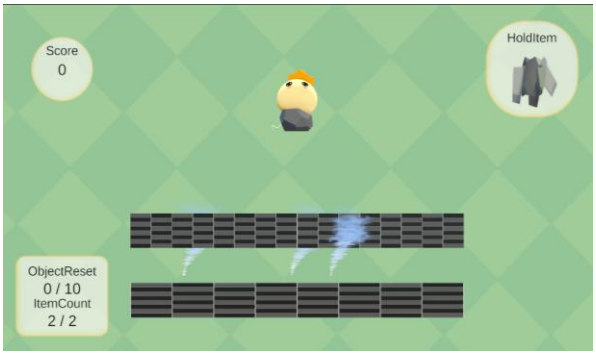
ゲーム概要

『磁石がくっつかないように落としていく3Dアクションゲーム』
オブジェクトを利用または邪魔されながら磁石と磁石がくっつかないようにしながら、
スコアを稼いでいくことを目的としているゲームです。

●カメラ管理







CameraController.c

```
8 個の参照
enum CameraState
{
    // メインカメラ
    Main,
    // 正面カメラ
    Top,
    // 横カメラ
    Left,
}
```

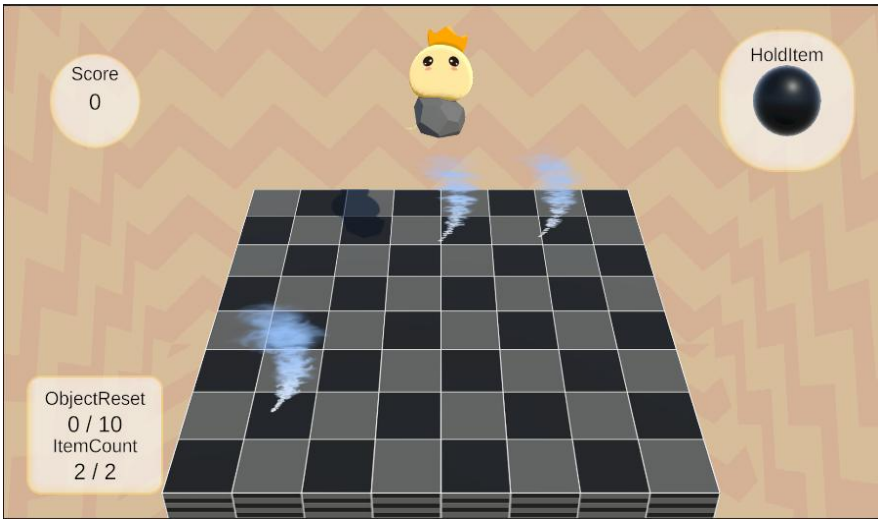
今使用しているカメラをenumで管理

CameraController.cs

```
// ポーズじゃないとき
if (!StageScene.Instance.IsPaused)
{
    switch (cameraState)
    {
        case CameraState.Main:
            //QキーまたはXボタンが押した時、正面カメラをアクティブにする
            if (Input.GetButtonDown("Fire2"))
            {
                //サブカメラをアクティブに設定
                mainCamera.SetActive(false);
                subCamera.SetActive(true);
                leftCamera.SetActive(false);
                cameraState = CameraState.Top;
            }
            break;
        case CameraState.Top:
            //QキーまたはXボタンが押した時、メインカメラをアクティブにする
            if (Input.GetButtonDown("Fire2"))
            {
                //メインカメラをアクティブに設定
                mainCamera.SetActive(true);
                subCamera.SetActive(false);
                leftCamera.SetActive(false);
                cameraState = CameraState.Left;
            }
            break;
        case CameraState.Left:
            //QキーまたはXボタンが押した時、横カメラをアクティブにする
            if (Input.GetButtonDown("Fire2"))
            {
                //メインカメラをアクティブに設定
                mainCamera.SetActive(false);
                subCamera.SetActive(false);
                leftCamera.SetActive(true);
                cameraState = CameraState.Main;
            }
            break;
    }
}
```

特定のキーを押すとカメラが切り替わる

●竜巻生成



TornadoGenerate.cs

```
// 配列の中身をシャッフルする
1 個の参照
void Shuffle(GameObject[] num)
{
    for (int i = 0; i < num.Length; i++)
    {
        GameObject temp = num[i];
        int randomIndex = Random.Range(0, num.Length);
        num[i] = num[randomIndex];
        num[randomIndex] = temp;
    }
}
```

TornadoGenerate.cs

に設置している生成ポイントを配列に格納し中身をシャッフルする
（ランダムで取得した配列からポジションを取得できなかったため）

```
// ランダムな箇所に竜巻を生成
1 個の参照
public void TornadoGenerator()
{
    // 配列の中身をシャッフル
    Shuffle(TornadoPoint);
    //  シャッフルした配列の上から３つの座標に竜巻を生成
    for (int i = 0; i < 3; i++)
    {
        Vector3 vector3 = TornadoPoint[i].transform.position;
        Instantiate(Tornado, vector3, Quaternion.identity);
    }
}
```

シャッフルした配列の上から３つを竜巻の生成ポイントとする

TornadoRotation.cs

```
// オブジェクトを回転移動させる
1 個の参照
private void ObjectSuction()
{
    // 回転の中心座標を取得
    Vector3 center = this.transform.parent.position;

    // 中心点centerの周りを、軸axisで、period周期で円運動
    transform.RotateAround(
        center,
        _axis,
        360 / _period * Time.deltaTime
    );
}

// 竜巻の範囲内に来た時に中心に吸い込む
Unity メッセージ 0 個の参照
private void OnTriggerStay(Collider other)
{
    //竜巻の範囲に磁石が来た時
    if (other.tag == "Magnet")
    {
        // 磁石のRigidbodyを取得
        Rigidbody rigidbody = other.GetComponent<Rigidbody>();

        // 竜巻範囲にいるオブジェクトの向きを取得
        Vector3 vector3 = transform.position - rigidbody.transform.position;

        // 向きをNormalizeして向きの大きさを１にする
        vector3.Normalize();

        // 少しずつ減少させる
        rigidbody.velocity *= 0.8f;

        rigidbody.AddForce(vector3 * rigidbody.mass * 20.0f);
    }
}
```

竜巻を中心を軸に回転するように実装
（竜巻が回転しているように見せるため）

竜巻の範囲内に磁石が来たら吸い込み

●磁石生成

MagnetGenerater.cs

```
// オブジェクトを生成します
3 個の参照
public void ObjectGenerate(GameObject gameObject)
{
    // このコードを付与しているオブジェクトの座標を取得
    Vector3 pos = transform.position;

    // オブジェクトを生成
    GeneOb = Instantiate(gameObject, pos, Quaternion.identity) as GameObject;
    // オブジェクトの名前変更（磁力範囲にいるものすべてを認識するため名前指定で対象を除外）
    GeneOb.transform.GetChild(0).name = "Generate" + ObjectCount;
    // 生成したアイテムを削除できるように配列に格納
    ObjectArray[ObjectCount] = GeneOb;
    // rigidbodyの取得
    generate = ObjectArray[ObjectCount].transform.GetChild(0).GetComponent<Rigidbody>();
    // 生成したオブジェクトのY軸と回転を固定
    generate.constraints = RigidbodyConstraints.FreezePositionY | RigidbodyConstraints.FreezeRotation;
}
```

指定されたオブジェクトを複製します。
Colliderの仕様上、範囲内すべてのものを対象としているため、自分の磁石も対象になる。
この問題を解決するため自分の磁石に関して名前を指定し対象から消すように処理。
複製したオブジェクトを配列に格納します。
複製したオブジェクトが動かないようにY座標と回転を固定します。

MagnetRange.cs

```
Unity メッセージ 0 個の参照
private void OnTriggerStay(Collider other)
{

    //磁力範囲に磁石が来た時
    if (other.tag == "Magnet" && other.name != transform.parent.GetChild(0).name)
    {
        // 引き寄せ
        magnetOperation.MagnetMove(other);
    }
}
```

範囲内にあるオブジェクトの名前と自分の子にある名前が一緒なら処理を無視。
（上記で記載した問題を解消するため）

MagnetGenerater.cs

```
// ランダムにオブジェクトを生成します
4 個の参照
public void RandomGenerate()
{
    // 生成するオブジェクトの確率範囲
    int rand = Random.Range(MinValue, MaxValue);

    // 生成できるアイテムの上限を指定
    if (ObjectCount < MaxObjectCount)
    {
        // 確率によって生成するオブジェクトを指定
        if (rand <= 50)
        {
            GenerateNumber = 0;
            ObjectGenerate(BigMagnet);
        }
        else if (rand <= 70)
        {
            GenerateNumber = 1;
            ObjectGenerate(SphereMagnet);
        }
        else if (rand <= 100)
        {
            GenerateNumber = 2;
            ObjectGenerate(Magnet);
        }
    }
    ObjectCount++;
    gameState = PlayerState.Idle;
}
```

Randomを使用してランダムな磁石を生成します。
GenerateNumberはアイテムをホールドする際の情報として使用します。

MagnetGenerater.cs

```
    // ステージにある個数が一定を超えると削除
    if (ObjectCount > MaxObjectCount)
    {
        ObjectDestroy();
    }
}
```

MagnetGenerater.cs

ステージにあるオブジェクトが一定を超えると配列にあるオブジェクトを削除する。

```
// 指定のオブジェクトを削除
1 個の参照
public void ObjectDestroy()
{
    for (int i = 0; i < ObjectArray.Length; i++)
    {
        Destroy(ObjectArray[i]);
    }
    ObjectArray = new GameObject[MaxObjectCount];
    ObjectCount = 0;
    gameState = PlayerState.Idle;
    RandomGenerate();
}
```

●磁石挙動

MagnetOperation.c

```
//磁力範囲に磁石が来た時の処理
1 個の参照
public void MagnetMove(Collider other)
{
    Rigidbody rigidbody = other.GetComponent<Rigidbody>();
    Rigidbody MagRb = Magnet.transform.GetChild(0).GetComponent<Rigidbody>();

    // 磁力範囲にいるオブジェクトの向きを取得
    Vector3 vector3 = MagRb.transform.position - rigidbody.transform.position;

    // オブジェクトまでの距離の２乗を取得
    var distance = vector3.magnitude;

    distance *= distance;

    // 磁力計算
    var gravity = coefficient * MagRb.mass * rigidbody.mass / distance;

    rigidbody.velocity *= 0.8f;

    // 範囲内にいる磁石は磁力範囲の元（磁石）に近づく
    rigidbody.AddForce(gravity * vector3.normalized, ForceMode.Force);
}
```

磁石範囲内に他の磁石が来た時、RIgidbody を取得し引き寄せる

MagnetOperation.c

```
// 磁石の範囲内に特定のオブジェクトが来た時の処理
1 個の参照
public void MagnetDecision(Collision other)
{
    if (other.gameObject.tag == "Ground")
    {
        // 地面にふれたときにオブジェクトを追加で生成
        if (!IsGround)
        {
            MagnetGenerater.Instance.RandomGenerate();
            MagnetGenerater.Instance.ItemRelease();
            IsGround = true;

            if(MagnetGenerater.Instance.GenerateNumber == 0)
            {
                StageScene.Instance.AddScore(200);
            }
            else if (MagnetGenerater.Instance.GenerateNumber == 1)
            {
                StageScene.Instance.AddScore(300);
            }
            else if (MagnetGenerater.Instance.GenerateNumber == 2)
            {
                StageScene.Instance.AddScore(100);
            }
        }
    }
    else if (other.gameObject.tag == "Magnet")
    {
        StageScene.Instance.GameOver();
    }
}
```

MagnetOperation.c

MagnetOperation.c

ブジェクトがついた時の処理
地面：
スコア加算、アイテム生成、
アイテム効果リセット
磁石：
ゲームオーバー

●プレイヤー操作

MagnetGenerater.cs

プレイヤーの状態をenumにて管理

```
8 個の参照
enum PlayerState
{
    // 待機中
    Idle,
    // 落下中
    Fall,
    // Clear
    Clear,
}
//ステージ開始を初期値に設定
PlayerState gameState = PlayerState.Idle;
```

MagnetGenerater.cs

```
// Update is called once per frame
Unity メッセージ|0 個の参照
void Update()
{
    float x = Input.GetAxisRaw("Horizontal");
    float y = 0;
    float z = Input.GetAxisRaw("Vertical");

    // このコードを付与しているオブジェクトの座標を取得
    Vector3 pos = transform.position;

    switch (gameState)
    {
        case PlayerState.Idle:
            // 移動
            transform.Translate(new Vector3(x, y, z) * Time.deltaTime * PlayerSpeed);
            // 生成したオブジェクトが追従
            generate.transform.position = new Vector3(transform.position.x, transform.position.y, transform.position.z);

            // 移動できる範囲を指定
            Vector3 currentPos = transform.position;
            currentPos.x = Mathf.Clamp(currentPos.x, -5, 5);
            currentPos.z = Mathf.Clamp(currentPos.z, -5, 5);
            transform.position = currentPos;
            break;
        case PlayerState.Fall:
            // 操作不能
            transform.Translate(new Vector3(x, y, z) * Time.deltaTime * PlayerSpeed);
            break;
        case PlayerState.Clear:
            break;
        default:
            break;
    }
```

プレイヤーの状態にて磁石の操作を管理
待機状態：
プレイヤーの移動と所持している磁石とプレイヤーのポジションを固定
落下状態：
磁石の固定を解除
（プレイヤーは待機状態にて移動範囲を指定しているので落下時に範囲外に行っても待機状態時に範囲内に戻るため問題なし）

MagnetGenerater.cs

```
// Update is called once per frame
Unity メッセージ | 0 個の参照
void Update()
{
    switch (gameState)
    {
        case SceneState.Intro:
            break;
        case SceneState.Play:
            // ポーズのトグル操作
            if (Input.GetButtonDown("Cancel"))
            {
                if (!IsPaused)
                {
                    Pause();
                }
                else
                {
                    Resume();
                }
            }
            // 左クリックまたはR1ボタンをクリック時Y軸の固定を解除
            if (Input.GetButtonDown("Fire1"))
            {
                MagnetGenerater.Instance.FallObject();
            }
```

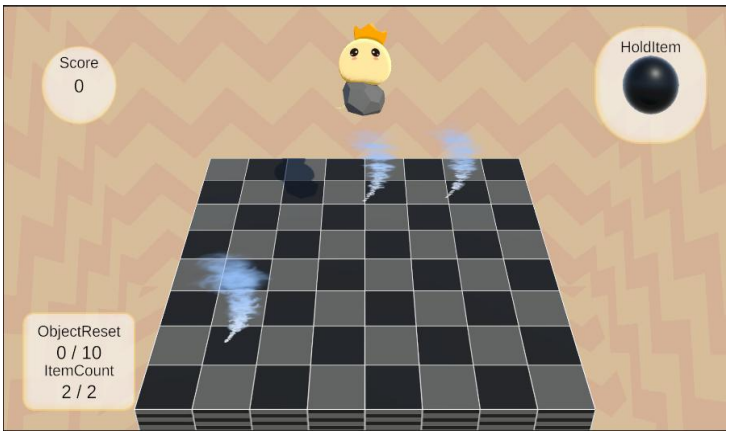
シーンの状態をenumにて管理
Playの時のみ操作可能

MagnetGenerater.cs

```
// 落下
1 個の参照
public void FallObject()
{
    IsHold = false;
    gameState = PlayerState.Fall;
    generate.constraints = RigidbodyConstraints.None;
    audioSource.PlayOneShot(soundOnClickSE);
}
```

RIgidbodyにて固定していたY座標を解除し
SEを再生

●プレイヤースキル



① ホールド

MagnetGenerater.cs

```
// オブジェクトをホールド
1 個の参照
public void SpareGenerate()
{
    if (!IsHold)
    {
        //  ホールドがないとき
        if (!SpareObject)
        {
            IsHold = true;
            SpareObject = ObjectArray[ObjectCount - 1];
            SpareObject.SetActive(false);
            ObjectArray[ObjectCount - 1] = null;
            ObjectCount--;
            HoldItem();
            RandomGenerate();
        }
        else
        {
            IsHold = true;
            ObjectArray[ObjectCount - 1].SetActive(false);
            SpareObject.SetActive(true);
            MidleSpare = SpareObject;
            generate = SpareObject.transform.GetChild(0).GetComponent<Rigidbody>();
            SpareObject = ObjectArray[ObjectCount - 1];
            ObjectArray[ObjectCount - 1] = MidleSpare;
            HoldItem();
        }
    }
}
```

MagnetGenerater.cs

ホールドしているかしていないかで処理を分別
ホールドしていない：
今の磁石を無効化し新たに生成
ホールドしている：
今の磁石とホールドしている磁石を入れ替える

MagnetGenerater.cs

```
// ホールドしているアイテムを表示
2 個の参照
public void HoldItem()
{
    SpareNumber = GenerateNumber;
    for (int i = 0; i < HoldObject.Length; i++)
    {
        if (i == SpareNumber)
        {
            HoldObject[i].SetActive(true);
        }
        else
        {
            HoldObject[i].SetActive(false);
        }
    }
}
```

RenderTextureを使用し画面外にカメラを設置し
ホールドしている磁石を表示しているため
ホールドしている磁石の番号から表示する磁石を指定。

② アイテム

MagnetGenerater.cs

```
// アイテム（光の切り替え）を使用
1 個の参照
public void UseItem()
{
    if(UseCount < MaxUseCount)
    {
        if (!IsItem)
        {
            TopLight.SetActive(true);
            DefaultLight.SetActive(false);
            IsItem = true;
        }
        else
        {
            return;
        }
        UseCount++;
    }
}
```

光を真下にすることで落下地点を予測できるようにしているためアイテムを使用後光の入れ替えをしています。

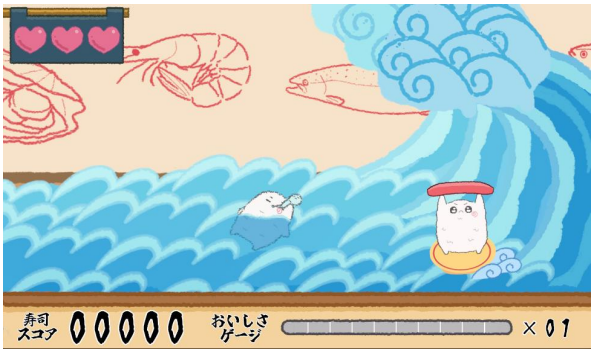
MagnetGenerater.cs

```
// アイテム使用後に解除
1 個の参照
public void ItemRelease()
{
    TopLight.SetActive(false);
    DefaultLight.SetActive(true);
    IsItem = false;
}
```

磁石着地後に光が元に戻るようにしています。

Unity 2Dアクションゲーム

作品名：寿司サーファーズ

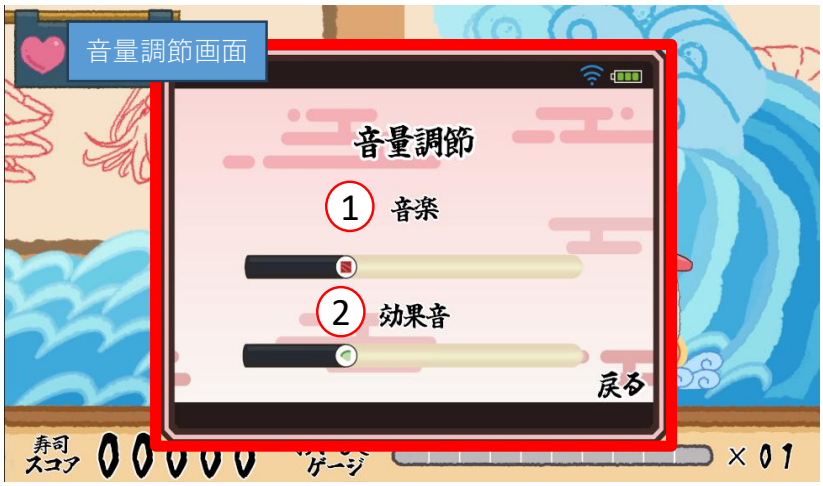


開発環境	Unity 2022.3.42f1
使用ツール	Microsoft Visual Studio 2022
使用ツール	Microsoft Visual Studio 2022
動作環境	PC(コントローラ/キーボード)
制作期間	3ヶ月(2024年10月～12月末)
制作人数	9人 プランナー　3人 プログラマー　3人 グラフィッカー　3人

ゲーム概要

『ステージ上のアイテムを使ってお寿司のスコアを上げていく２Dアクションゲーム』
ステージ上に流れているオブジェクトをアクション(ターン)で避けながら、アイテムを拾いお寿司のスコア(口コミ)を上げていくことを目的としているゲームです。

●音量調節・管理




```
// Exit Button が押されたときに発生する UnityEvent
public UnityEvent onExitButtonClick;
// BGMSliderの値が変更されたときに発生する  UnityEvent
public UnityEvent<float> onBGMSliderValueChanged;
// SESliderの値が変更されたときに発生する  UnityEvent
public UnityEvent<float> onSESliderValueChanged;
```

音楽のスライダーの値が変更された際に呼び出すように設定

①　音楽（BGM）の調整

SoundUI.cs

```
// BGMスライダーの値変更によって呼び出されます。
0 個の参照
public void SetBGM(float value)
{
    //スライダーBGMの現在値を10段階に補正してから[-80, 0]に変換
    var volume = Mathf.Clamp(Mathf.Log10(value / 10) * 20f, -80f, 0f);
    audioMixer.SetFloat("BGM", volume);
}
```

②　効果音（SE）の調整

SoundUI.cs

```
// SEスライダーの値変更によって呼び出されます。
0 個の参照
public void SetSE(float value)
{
    //スライダーSEの現在値を10段階に補正してから[-80, 0]に変換
    var volume = Mathf.Clamp(Mathf.Log10(value / 10) * 20f, -80f, 0f);
    audioMixer.SetFloat("SE", volume);
    // テスト音声を再生
    audioSource.PlayOneShot(soundOnTestSE);
}
```

音楽または効果音のスライダーを変更時、音量を変更するように実装
効果音のみテスト音声が流れるように追加（音楽に関しては常に流れているためなし）

○　音量調整管理

SoundUI.cs

```
Unity メッセージ 0 個の参照
public void OnDestroy()
{
    PlayerPrefs.SetInt("BGMValue", (int)BGMSlider.value);
    PlayerPrefs.SetInt("SEValue", (int)SESlider.value);
    PlayerPrefs.Save();
}
```

タイトル画面とプレイ画面でSceneが分かれているため
Sceneを遷移した際に音楽と効果音の値を保存

SoundUI.cs

```
Unity メッセージ 0 個の参照
void Awake()
{
    audioSource = GetComponent<AudioSource>();

    var bgmValue = (float)PlayerPrefs.GetInt("BGMValue", 3);
    BGMSlider.value = bgmValue;
    var seValue = (float)PlayerPrefs.GetInt("SEValue", 3);
    SESlider.value = seValue;

    // UnityEvent を追加
    BGMSlider.onValueChanged.AddListener((float value) => { onBGMSliderValueChanged.Invoke(value); });
    SESlider.onValueChanged.AddListener((float value) => { onSESliderValueChanged.Invoke(value); });
    exitButton.onClick.AddListener(() => { onExitButtonClick.Invoke(); });
}

Unity メッセージ 0 個の参照
private void Start()
{
    // スライダーUIの現在値を10段階に補正してから[-80, 0]dBに変換
    audioMixer.SetFloat("BGM", Mathf.Clamp(Mathf.Log10(BGMSlider.value / 10) * 20f, -80f, 0f));
    audioMixer.SetFloat("SE", Mathf.Clamp(Mathf.Log10(SESlider.value / 10) * 20f, -80f, 0f));

    Hide();
}
```

OnDestroyにて音楽と効果音の値を保存しているため
シーンが切り替わっても音量の値の差異はなし

●シーン遷移

○　シーン遷移条件

タイトル画面

- プレイ画面に移動
- もう一度プレイ

プレイ画面

- タイトル画面に戻る
- もう一度プレイ
- ゲームクリア

ゲームクリア画面

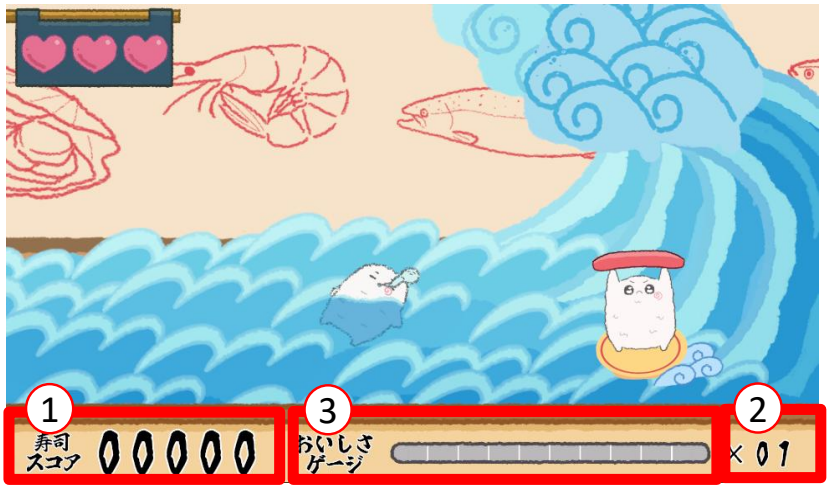
- タイトル画面に戻る
- もう一度プレイ

StageScene.cs, ClearScene.cs,

```
3 個の参照
IEnumerator OnLoadScene(string sceneName)
{
    // アニメーションが終了するまで1秒待機
    yield return new WaitForSecondsRealtime(1);
    // シーンをロードする
    SceneManager.LoadScene(sceneName);
}
```

シーンの遷移条件が多いため
移動先を指定できるようにすることで汎用制を高めた
そのため、上記のコードに加えてシーン遷移に必要な情報
（アニメーション、スコアなど）を記載するだけでよくなった

●スコア・ゲージ管理



① スコアの管理

StageScene.cs

```
//スコアの加算処理
1 個の参照
public void AddWasabi(int value)
{
    ScoreCount += value;
    itemScore += value;
}

1 個の参照
public void AddSoysource(int value)
{
    ScoreCount += value;
    itemScore += value;
}
```

アイテムを取るとスコアと取った回数を取得

StageScene.cs

```
//スコアの減少処理
2 個の参照
public void RemoveScore(int value)
{
    //無敵の時は無視
    if (player.isMuteki)
    {
        return;
    }
    if(scoreCount == 0)
    {
        return;
    }
    ScoreCount -= value;
}
```

スコアが減少する際にはプレイヤーが無敵の時にはスコアが減少されない
スコアが0の時にアイテムに当たるとUIの仕様上エラーが起きてしまうため 0 の時もスコアが減少しない

StageScene.cs

```
2 個の参照
public void AddExcelentTurn(int value)
{
    ScoreCount += value;
    ExcellentTurnCount++;
}

2 個の参照
public void AddGoodTurn(int value)
{
    ScoreCount += value;
    GoodTurnCount++;
}
```

ターン時の評価によってスコアとターンした回数を取得

StageScene.cs

```
2 個の参照
public void updateValue(int count)
{
    var scoreCount = count;
    for (int index = 0; index < values.Length; index++)
    {
        values[index].sprite = numbers[scoreCount % 10];
        scoreCount /= 10;
    }
}
```

スコアの表記方法を0～9までのイラストを使用しているため1桁ごとに数字の指定をしている

② ゲージ倍率の管理

StageScene.cs

```
//ゲージの加算処理
4 個の参照
public void AddGauge(int value)
{
    GaugeScore += value;
    if (GaugeScore >= (gaugeCount) * 10)
    {
        gaugeCount++;
    }
}
```

ゲージの上限値を10としているため
値が10を超えるとゲージ倍率が1ずつ増えていく

③ ゲージの管理

GaugeUI.cs

```
2 個の参照
public void updateGauge(int count, int score)
{
    float gauge = score % 10;
    if(score >= (count-1) * 10)
    {
        for(int index = 0; index < gaugecount.Length; index++)
        {
            gaugecount[index].SetActive(false);
        }
    }
    for (int index = 0; index < gauge; index++)
    {
        gaugecount[index].SetActive(true);
    }

    for (int index = 0; index < gauges.Length; index++)
    {
        gauges[index].sprite = numbers[count % 10];
        count /= 10;
    }
}
```

ゲージが上昇する際にイラストを表示することで

0の時

10の時

クリア時の処理

StageScene.cs

```
// ゲームクリア時にスコアとゲージ、ターンの回数を登録します。
1 個の参照
IEnumerator OnStageClear()
{
    var totalScoreCount = 0;
    var totalGaugeCount = 0;
    var totalItemScore = 0;
    var totalExcellentCount = 0;
    var totalGoodCount = 0;

    totalScoreCount += ScoreCount;
    totalGaugeCount += GaugeCount;
    totalItemScore += itemScore;
    totalExcellentCount += ExcellentTurnCount;
    totalGoodCount += GoodTurnCount;

    PlayerPrefs.SetInt("TotalScoreCountKey", totalScoreCount);
    PlayerPrefs.SetInt("TotalGaugeCountKey", totalGaugeCount);
    PlayerPrefs.SetInt("TotalItemCount", totalItemScore);
    PlayerPrefs.SetInt("ExcellentTurn", totalExcellentCount);
    PlayerPrefs.SetInt("GoodTurn", totalGoodCount);
    PlayerPrefs.Save();
    animetor.SetTrigger(outroId);
    // アニメーションが終了するまで1秒待機
    yield return new WaitForSeconds(1);
    StartCoroutine(OnLoadScene("GameClear"));
}
```

クリア画面にてスコアを参照するため
必要な値を保存しクリア画面に持っていくように追加
保存する回数を減らすためターンに関しては回数のみ