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1年

猿渡 昊輝

更新日　2025.6/10

目次

スキルシート

サルワタリ　　コウキ
猿渡　昊輝

性別　男　年齢　19歳
希望職種　プログラマー

保有スキル

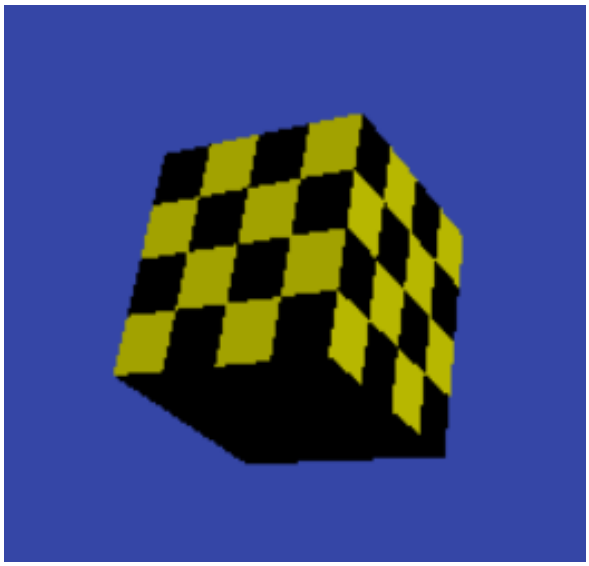
言語	
エンジン	
その他環境	Microsoft DirectX11　(3D)
ツール	

自己PR

私は１年間専門学校に通い、C/C++,C#,HTML,CSS,JavaSctipt,PHPを学んできました。
その知識を生かしながら様々な作品を作り、制作経験を積んできました。
就職先で活躍するというビジョンを実現するために日々勉強に励んでいます。

Direct3D11の初期化からのフルスクラッチ

自作エンジンを目指したデモ作品



現在DirectX3Dを用いて3Dゲームの自作エンジン制作を目指し勉強をしています。左の立方体は初期化からのフルスクラッチでの制作で、非常に難しいですがなんとか表示させることができました。

クラス設計やメモリに対して気を配ったりなど、
ゲーム制作に大切なことを意識しながら作品作りをしていたので、自分の実力を伸ばしつつ勉強できたと思います。

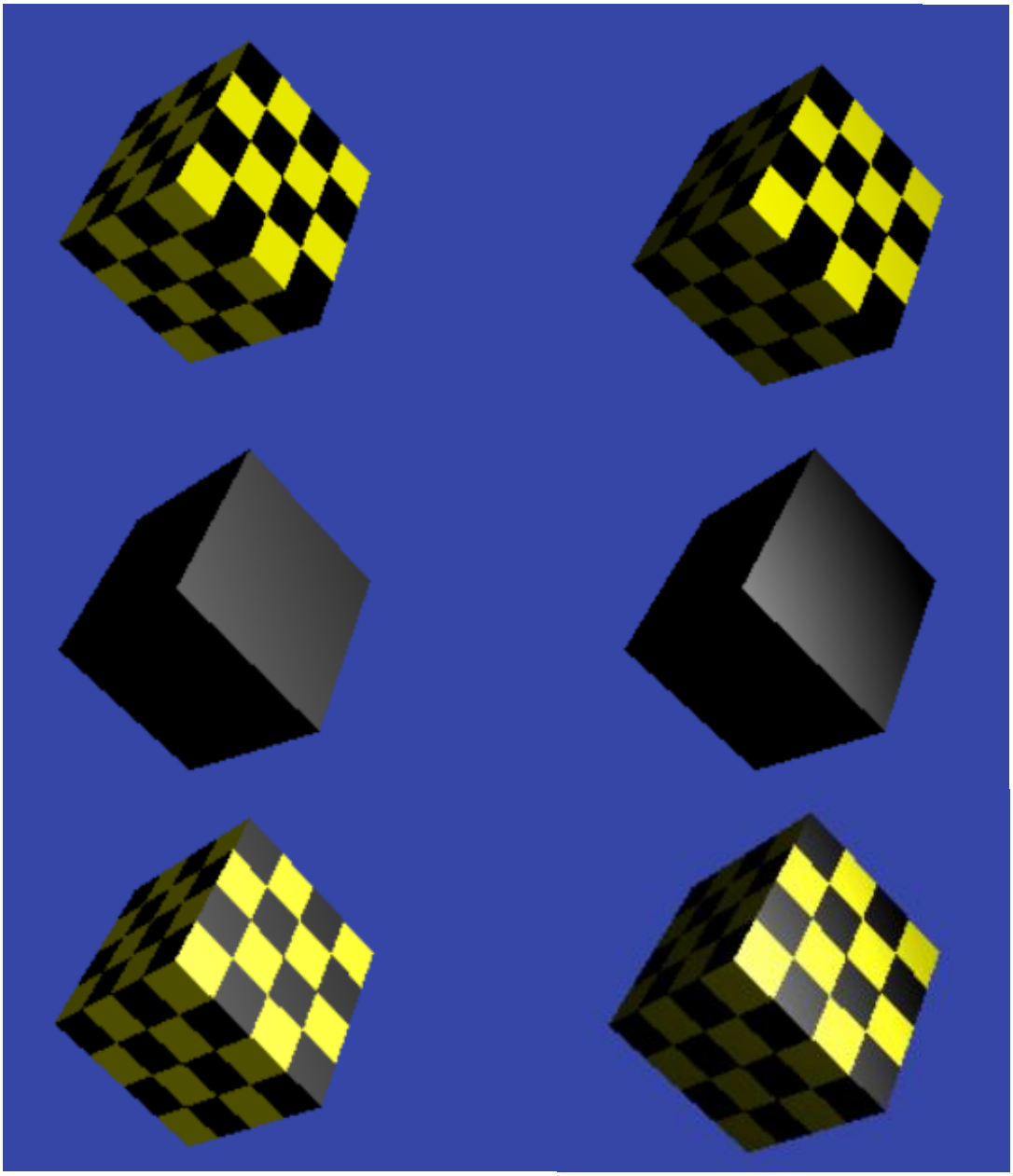
HLSLシェーダープログラミング

現在デモ作品ではHLSLプログラミングを利用しており、
VertexShader,GeometryShader,PixcelShaderの3つを用いて表現しています。その中でもライティング表現については平行光源と
点光源、拡散反射と鏡面反射を実装することができました。

平行光源　　点光源

拡散反射

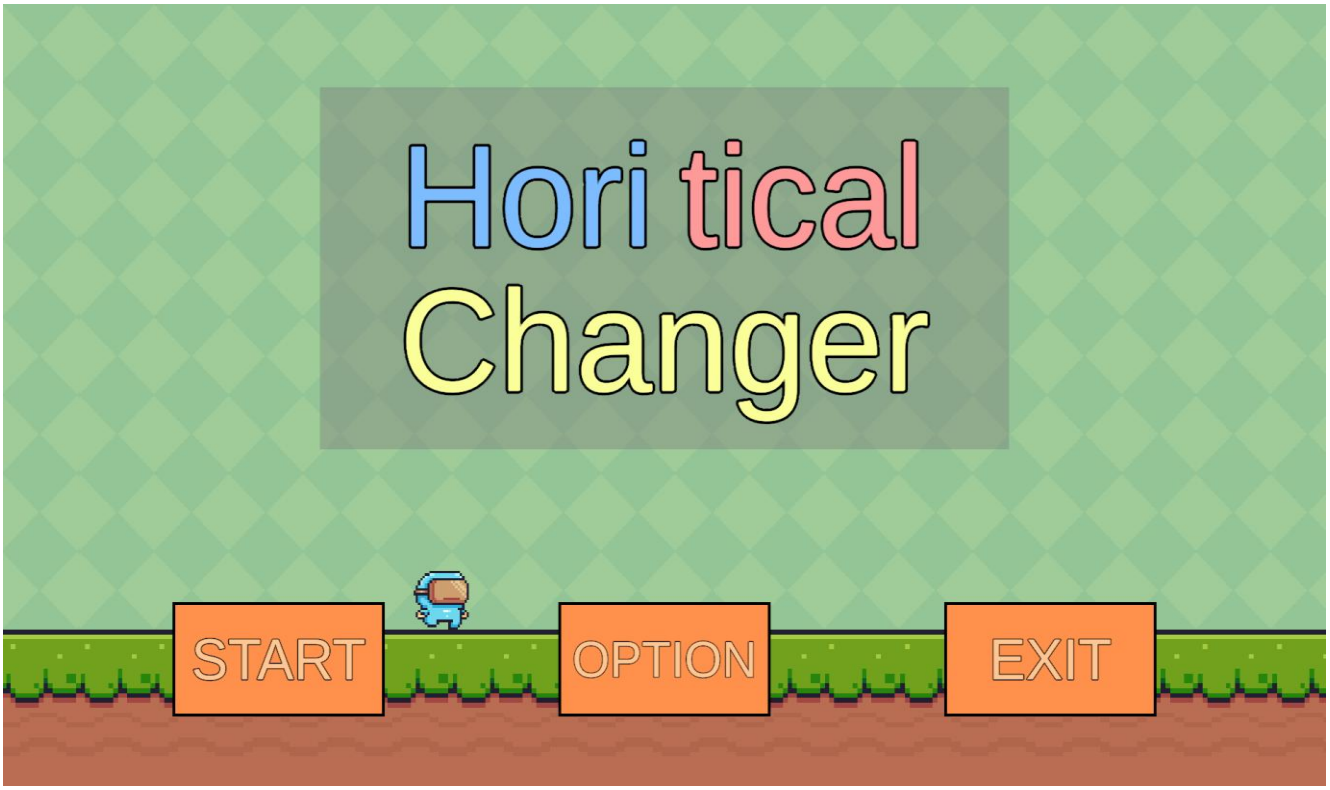
鏡面反射



拡散反射＆鏡面反射

上図が現在のそれぞれライティング表現の組み合わせです。今回は昔ながらのライティング表現を学びましたが、今後はPBR(物理ベースレンダリング)に使われるPBS(物理ベースシェーディング)に挑戦しようと考えています。
また、今回の黄・黒のテクスチャはRawデータとして直接入れているのですが、一般的なjpeg、pngなどを読み込み・書き込み・変換を行うための
WIC(Windows Imaging Conponent)にも挑戦していこうかなと
考えています。

Unity 2Dコレクトアクションゲーム



タイトル	ホリティカル チェンジャー Horitical Changer
ターゲット	暇な人
動作環境	PC
開発環境	Unity 2022.3.23f1
開発言語	C#
開発期間	2週間
開発人数	個人
担当箇所	すべて
ゲーム概要	『2Dコイン集めアクションゲーム』 横と縦、それぞれに動けるキャラを切り替えながら ステージを進んでいき、最終的にステージ上の すべてのコインを集めることでクリアとなります。 キャラ切り替えを工夫し、ギミックをうまく使いながら クリアを目指していくゲームです。

◇列挙型を用いたフラグ管理

```
public enum CharacterState
{
    // X軸方向に動けるキャラ
    XCharacter,
    // Y軸方向に動けるキャラ
    YCharacter,
}
// プレイヤーの運動状態を分ける
29 個の参照
public enum PlayerState
{
    // 待機状態
    Idling,
    // 走行状態
    Running,
    // ジャンプ予備動作中
    JumpStart,
    // ジャンプ中
    Jumping,
    // 落下中
    Falling,
}
```

X
Idling

Y
Idling

X
Running

Y
JumpStart
Jumping

Y
Falling

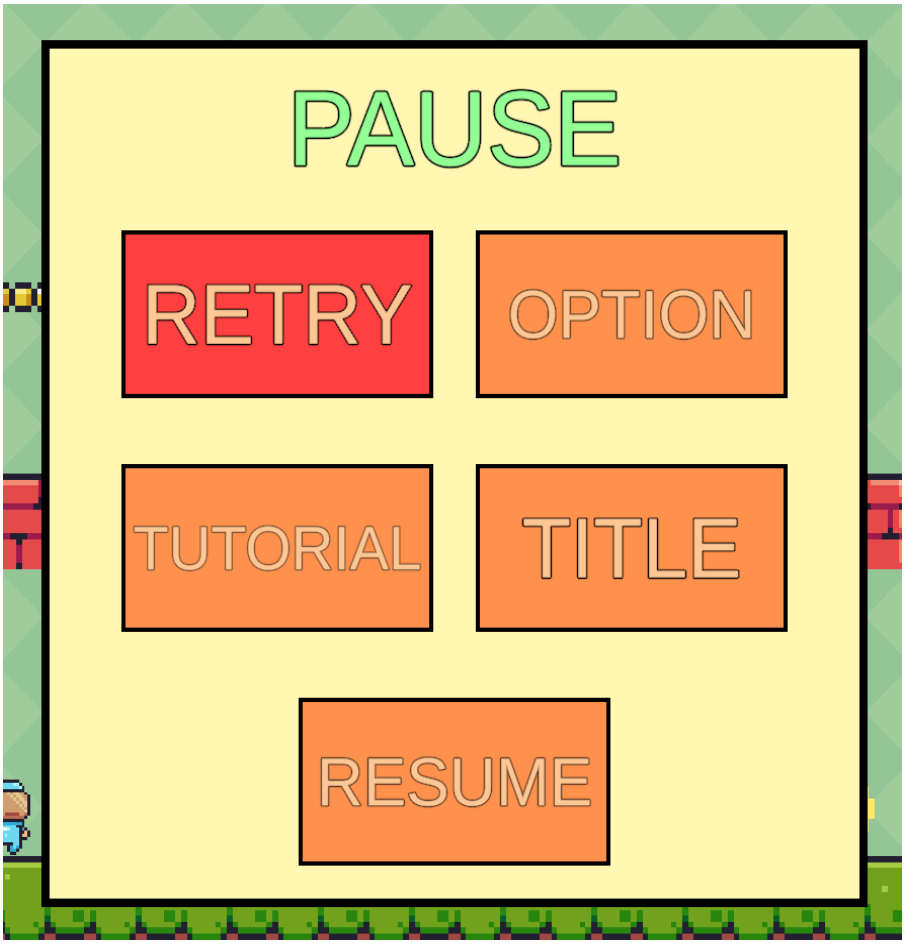
```
// プレイヤーの運動状態で制御
switch (CurrentState)
{
    case PlayerState.Idling:
        // 落下
        if (!isGround)
        {
            CurrentState = PlayerState.Falling;
            animator.SetBool(idleId, false);
            animator.SetBool(fallId, true);
            return;
        }
        // 走る
        if (moveInput != Vector2.zero)
        {
            animator.SetBool(idleId, false);
            Move();
            Run();
            return;
        }
        break;
```

列挙型を用いてプレイヤーのフラグを一括で管理しました。
左図はUpdate内で行っている処理です。プレイヤーの状態を一つずつ分けることで、コードも見やすくなり実装も行いやすくなりました。

ゲーム中の最高審判官として作ったStageSceneクラスもこの列挙型を用いて実装しました。
(Intro, PlayGame, GameOver, StageClear の4つの状態)

◇UniRxを用いた汎用的なボタンの作成

Unity既存のボタンコンポーネントは
使わず、AssetStoreからインポートした
UniRxライブラリを用いて汎用的な
ボタンの作成に挑戦してみました。
既存のボタンコンポーネントは使いずらい
ところがあるので自分で改良してみようと
思ったことが作成の理由です。



作成の具体的な理由

・ボタンの設定をボタン
一つ一つに対して実装
しなければならない。
(何かしらのUIを作成する際
に、ボタンを必要な数だけ
作成して実装することや、
アニメーションを作成する
ときにボタンの種類が多い
ほど作るアニメーションも
多くなること、
ボタンの種類が同じでも
異なるアニメーションに
したい時などに、
無駄な作業が多いところ。)

・既存のものでは長押し
状態を取得できない。
など

```
public class CustomButton : MonoBehaviour
{
    // ボタンのアクティブ状態を保持する変数
    1 個の参照
    public IReadOnlyReactiveProperty<bool> IsActiveRP => isActiveRP;
    private readonly ReactiveProperty<bool> isActiveRP = new(true);

    // 参照するスクリプト(コンポーネント)の変数
    private ObservableEventTrigger observableEventTrigger;

    Unity メッセージ 10 個の参照
    protected virtual void Awake()
    {
        // コンポーネントの参照
        observableEventTrigger = GetComponent<ObservableEventTrigger>();
    }
```

UniRxの
コンポーネントの
参照や変数の宣言

```
// それぞれの場合によるイベントの発行

//// ボタンをクリックした時
//public IObservable<Unit> OnButtonClicked =>
//observableEventTrigger.OnPointerClickAsObservable().AsUnitObservable().Where(_ => isActiveRP.Value);

// ボタンを押した時
17 個の参照
public IObservable<Unit> OnButtonPressed =>
    observableEventTrigger.OnPointerDownAsObservable().AsUnitObservable().Where(_ => isActiveRP.Value);

// ボタンを離した時
17 個の参照
public IObservable<Unit> OnButtonReleased =>
    observableEventTrigger.OnPointerUpAsObservable().AsUnitObservable().Where(_ => isActiveRP.Value);

// ボタンの領域にカーソルが入った時
17 個の参照
public IObservable<Unit> OnButtonEntered =>
    observableEventTrigger.OnPointerEnterAsObservable().AsUnitObservable().Where(_ => isActiveRP.Value);

// ボタンの領域からカーソルが出た時
1 個の参照
public IObservable<Unit> OnButtonExited =>
    observableEventTrigger.OnPointerExitAsObservable().AsUnitObservable().Where(_ => isActiveRP.Value);
```

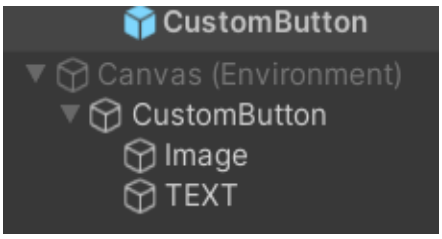
各イベントの発行

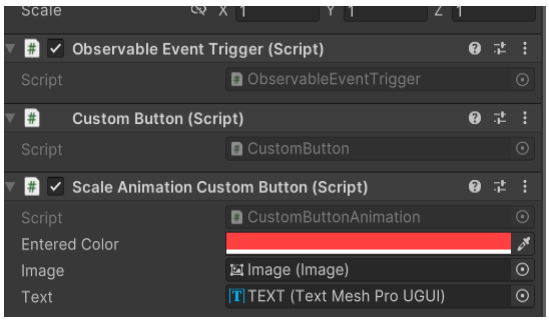
```
// アクティブ状態を取得する
1 個の参照
public bool GetIsActive() => isActiveRP.Value;

// アクティブ状態を変更する
11 個の参照
public void SetButtonActive(bool isActive)
{
    isActiveRP.Value = isActive;
}
```

アクティブ状態
制御の関数

CustomButtonクラスを作り汎用的なボタンの作成に取り組みました。
このクラスではボタンのアクティブ状態の制御や、各イベントの発行
など、ボタンの基礎的なものを書いています。参照するスクリプトである
ObservableEventTriggerは
[RequireComponent(typeof(ObservableEventTrigger))]でこのクラスを
アタッチした際に同時にアタッチされるようにしています。
この実装をしたことで元となるボタンをPrefabにすることができました。





元となるボタンをPrefabにできたことで
新しくボタンが追加されたとしてもテキストを
変えるだけ(画像があれば画像を変えるだけ)で
実装することができ、作業量を大きく抑える
ことができました。

・CustomButtonのアニメーション

```
private void Start()
{
    // コンポーネントの参照
    customButton = GetComponent<CustomButton>();

    // デフォルトのボタンカラーを保存
    defaultColor = image.color;

    // それぞれのイベントの購読
    // このオブジェクトがDestroyされた場合は購読を止める
    customButton.OnButtonPressed.Subscribe(_ => SetScale(pressedScale)).AddTo(gameObject);
    customButton.OnButtonReleased.Subscribe(_ => SetScale(defaultScale)).AddTo(gameObject);
    customButton.OnButtonEntered.Subscribe(_ => SetColor(enteredColor)).AddTo(gameObject);
    customButton.OnButtonExited.Subscribe(_ => SetColor(defaultColor)).AddTo(gameObject);
    customButton.IsActiveRP.Subscribe(SetButtonActive).AddTo(gameObject);
}
```

既存のもののアニメーションを一つ一つ
作らなければならない問題をスクリプトで
アニメーションさせることで問題を解消
しました。
(今回は色とスケールの変更)

```
// 本来は画像の変更だけで行う(CGさんがいる場合)
// スケールの変更
2 個の参照
private void SetScale(float scale)
{
    image.rectTransform.localScale = Vector3.one * scale;
    text.rectTransform.localScale = Vector3.one * scale;
}
// 色の変更
2 個の参照
private void SetColor(Color color)
{
    image.color = Color.white * color;
}

// 非アクティブ状態の色の変更
1 個の参照
private void SetButtonActive(bool isActive)
{
    // アクティブ状態がtureの時は色を変えずfalseの時はアルファ値を変更する
    float alpha = isActive ? ActiveImageAlpha : InactiveImageAlpha;
    image.color = new Color(1, 1, 1, alpha) * defaultColor;
    image.color = new Color(1, 1, 1, alpha) * defaultColor;
    text.color = new Color(1, 1, 1, alpha) * defaultColor;
}
```

StartメソッドでCustomButton
クラスのイベントを購読し、
それぞれのイベントによって
実行物を指定しています。

既存のものではできない
押している間はスケールを
小さくするアニメーションを
作りました。

この実装によって一つ一つ
作る手間や、違うアニメー
ションを作る際も新しいクラ
スを作るだけにできました。

Unity 2D無重力アクションゲーム

タイトル	宇宙（ソラ）のおとしもの
ターゲット	可愛い絵柄が好きな中学生～大人、宇宙が好きな人
動作環境	PC
開発環境	Unity 6000.0.36f1
開発言語	C#
開発期間	3か月
開発人数	プランナー4人, デザイナー3人, プログラマー1人
担当箇所	すべて
ゲーム概要	『横スクロール型２D無重力アクションゲーム』 無重力による操作性の不自由さやギミックによって醸し出される緊張感、飛来物の襲い来る中ぎりぎりで避けた時の安心感と、直後にまた襲い来る緊張感の応酬でプレイ中は常に飽きさせない、一種の忙しさのようなものを与え満足感を感じられる。

落とし物の取得状況管理

このゲームでは落とし物の収集要素があり、ステージ選択画面や、落とし物図鑑、リザルト画面などで落とし物の取得情報が必要になりました。そこで今回はDictionary型を用いて状況管理を行いました。

```
// ステージごとに獲得アイテム(落とし物)を管理するDictionary
private Dictionary<string, List<int>> stageItems;
```

追加

```
// アイテム(落とし物)を追加
1 個の参照
public void AddItem(int itemValue, int itemOrder, GameObject itemObject)
{
    // アイテムを追加
    stageItems[stageName][itemOrder - 1] = itemValue;

    // 配列にアイテムオブジェクト、スクリプトを追加
    item[itemNumber] = itemObject;
    itemScript[itemNumber] = itemObject.GetComponent<Item>();
}
```

削除

```
// 指定したステージのアイテムデータをクリア
2 個の参照
public void ClearItem(string stageName)
{
    for (int i = 0;i < totalItemQuantity;i++)
    {
        stageItems[stageName][i] = 0;
    }
}
```

ステージ選択画面

図鑑画面

リザルト画面

Dictionary型のキーはステージ別の名前とし、値の部分では落とし物それぞれにつけた番号を入れています。(0は未取得の状態とした。)
この実装方法によりステージ別で落とし物の取得状況を管理できるようになり、シンプルでわかりやすい実装にできたと思います。
そして上図のようにステージ選択画面や図鑑画面、リザルト画面での落とし物情報の取得も問題なく簡単に実装することができたので、今回の状況に合ったものがうまく作れたと思います。

JSONファイルを用いたセーブ・ロード

今まで制作してきたゲームでは、データの保存にPlayerPrefsを利用して行っていました。しかし、データ構造の自由度の無さやセーブデータをまとめて管理できない、セーブファイルの位置がアプリの保存領域に残せないなどの不満点がありました。そこで他のデータ保存方法について調べ、今回はJSONファイルを利用することを決めました。

GameData.cs

```cs
using System.Collections.Generic;

[System.Serializable]
14 個の参照
public class GameData
{
    // 音量の値の定義
    public float masterVolume;
    public float bgmVolume;
    public float seVolume;


    // ステージ別で保存する取得アイテムの値の定義
    public Dictionary<string, List<int>> gotItems = new();


    // ステージ別で保存するベストクリアタイムの値の定義
    public Dictionary<string, float> clearTime = new();
    // ステージ別で保存するベストクリアタイムの値の定義
    public Dictionary<string, float> bestClearTime = new();


    // ステージ別で保存するクリア済みか判定する値の定義
    public Dictionary<string, bool> isStageClear = new();
}
```

GameData.csでは保存する変数の定義、SaveSystem.csではセーブとロードなどの関数を行っています。
とてもスタンダードで分かりやすいものにしました。

Dictionary型での保存を行いたかったためNewtonsoft.Jsonを導入しています。

SaveSystem.cs

```cs
using Newtonsoft.Json;
using System.IO;
using UnityEngine;

20 個の参照
public static class SaveSystem
{
    // ファイルパス
    private static string filePath = Application.persistentDataPath + "/gamedata.json";

    // データをセーブする
    8 個の参照
    public static void Save(GameData data)
    {
        string json = JsonConvert.SerializeObject(data, Formatting.Indented);
        File.WriteAllText(filePath, json);

        Debug.Log(json);
    }

    // データをロードする
    11 個の参照
    public static GameData Load()
    {
        if (File.Exists(filePath))
        {
            string json = File.ReadAllText(filePath);
            return JsonConvert.DeserializeObject<GameData>(json);
        }
        else
        {
            return new GameData(); // デフォルト値を返す
        }
    }

    // データを削除する
    0 個の参照
    public static void DeleteSave()
    {
        if (File.Exists(filePath))
        {
            File.Delete(filePath);
        }
    }

    // ファイルが存在するかチェック
    1 個の参照
    public static bool SaveFileExists()
    {
        return File.Exists(filePath);
    }
}
```

セーブデータのロード・セーブを行う場所(例としてクリアタイム用スクリプト)

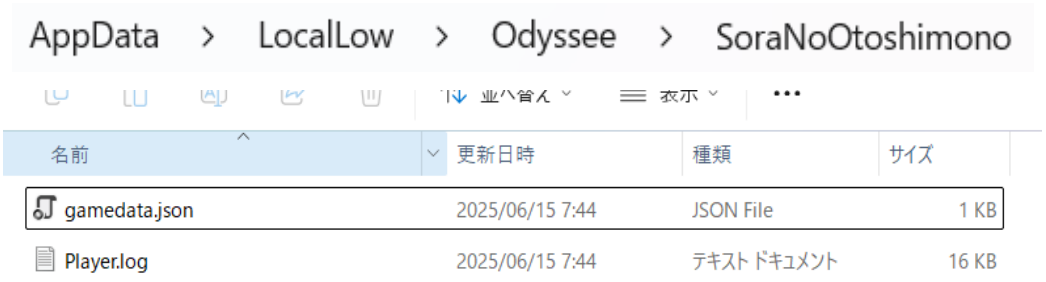
```
// データを更新
saveData = SaveSystem.Load();

// ベストタイムを取得
// なかった場合はCleaTimeをベストタイムとして保存、return(初回クリア時)
if (!saveData.bestClearTime.TryGetValue(clearSceneName, out float savedBestTimeData))
{
    saveData.bestClearTime[clearSceneName] = savedClearTimeData;
    SaveSystem.Save(saveData);

    return;
}

// クリアタイムがベストタイムを超えている場合はベストタイムを更新、セーブ
if (savedClearTimeData < savedBestTimeData)
{
    saveData.bestClearTime[clearSceneName] = savedClearTimeData;
    SaveSystem.Save(saveData);
}
```

この実装のおかげでシンプルなセーブデータ管理を行うことができるようになり、保存できるデータ構造もはるかに多くなり、保守性、拡張性が上がりました。さらに、しっかりとアプリのデータ保存領域に保存できたことで実際の市販のゲームに一歩近づけたと思います。
(以下図がエクスプローラーのスクショ)

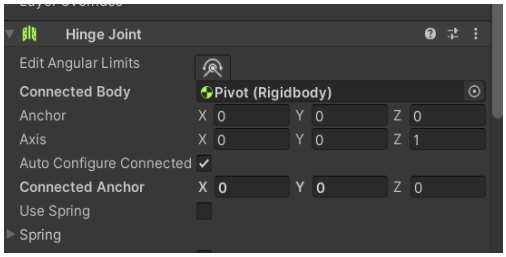


今回はJSONファイルに少し触れるだけであまり深くは掘り下げられませんでしたが、セキュリティ面でいうとこのままではあまり良くないので、次回は暗号化やハッシュをつけて改ざん検知機能などの実装に挑戦していきたいと考えています。

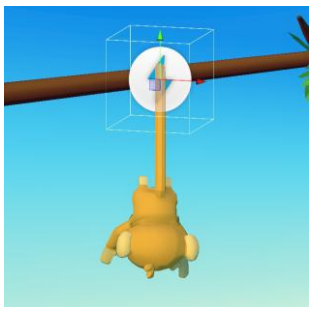
Unity ３Dクライムアクションゲーム

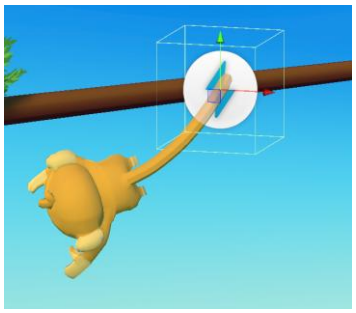
タイトル	サルは木から落ちぬ（キオ）
ターゲット	ポップな世界観が好きな方、タイムアタックが好きな方
動作環境	PC
開発環境	Unity 2022.3.23f1
開発言語	C#
開発期間	3か月
開発人数	プランナー2人, デザイナー3人, プログラマー1人
担当箇所	すべて
ゲーム概要	『縦スクロール型3Dクライムアクションゲーム』 高く生えた木に興味を持った猿が、頂上を目指して振り子のような動作で勢いをつけジャンプをし、その繰り返しで高い木を制覇していくアクションゲーム。 独特なアクションを駆使して登り切った時の楽しさや、タイムアタックの楽しさを感じることができる。

◇Hinge Joint を使ったプレイヤーの動き

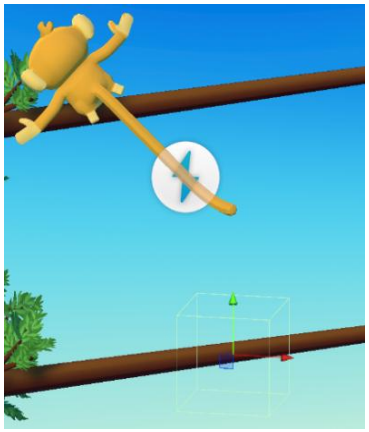


プレイヤーの勢いをつける動作、
振り子(ブランコ)の動きをUnity内の
HingeJointを使い、再現しました。





Pivotの位置を回転の軸に
させたい位置に置き、
プレイヤーの横方向に力を
加え動きを作りました。



Hinge Joint をつけている際はプレイヤーと
Pivotの位置が線でつながっているような状態
なのでジャンプ時には邪魔でした。
そこで変数化したHinge Joint をジャンプを
行うときにDestroyすることでジャンプの動作
を問題なく実行できるようにしました。

```
// ジャンプするときに HingeJoint コンポーネントを外す(ジャンプに影響を与えないため)
Destroy(joint);
```

再度枝にキャッチして振り子の動作に
戻るときはまた新しくHinge Joint をつけて
問題なく動作ができるようにしています。

```
catchRange.CatchEffect();

transform.position = catchRange.CatchPosition(transform.position.x, transform.position.y, transform.position.z);

// 外した HingeJoint コンポーネントを再度つける
joint = gameObject.AddComponent<HingeJoint>();
joint.axis = Vector3.forward;

pivot.transform.position = catchRange.transform.position;

// プレイヤーと pivot をつなげる
joint.connectedBody = pivotRigidbody;
```

◇3次元でのベクトルを使った座標計算

実装前
プレイヤーのしっぽ周りにColliderを置き、
その範囲中に枝がある状態の時にのみ
キャッチを行えるという実装をしていました。
しかし、その実装では範囲の端のほうで
キャッチをするとプレイヤーが浮いているよう
に見えるという問題がありました。

そこで枝の判定の端A,Bをつなぐ線
とプレイヤーから引いた法線の交点
(枝との最短距離の交点)を求め、
プレイヤーの座標を設定する解決策
を思いつき実装に取り組みました。

使用した計算方法

プレイヤー$P(px,py,pz)$　枝の端$A(ax,ay,az)$　枝の端$B(bx,by,bz)$　キャッチさせる点 $H(hx,hy,hz)$

$$k=\frac{(px-ax)(bx-ax)+(py-ay)(by-ay)+(pz-az)(bz-az)}{(bx-ax)^2+(by-ay)^2+(bz-az)^2}$$

$$hx=k*bx+(1-k)*ax \qquad hy=k*by+(1-k)*ay \qquad hz=k*bz+(1-k)*az$$

図１

以下計算中のコード

・枝とキャッチ範囲が重なった時に実行させる計算
枝の端の座標を求め、プレイヤーの座標が必要でない場所の計算を
行っています。　（ｋの分母部分の計算　※図１参照）

```
// CatchRange に入っているか判定する
Unity メッセージ 10 個の参照
private void OnTriggerEnter(Collider collider)
{
    int  catchLayer = 1 << collider.gameObject.layer;

    if((targetLayerName & catchLayer) != 0)
    {
        // 触れた枝のオブジェクトを保存
        currentCollisionBranch = collider.gameObject;

        // コライダーの端のポジションを求める
        collider.transform.GetPositionAndRotation(out var position, out var rotation);
        float height = collider.GetComponent<CapsuleCollider>().height;
        var radius = collider.GetComponent<CapsuleCollider>().radius;
        positionA = rotation * new Vector3(height / 2,-radius, 0) + position;
        positionB = rotation * new Vector3(-(height / 2), -radius, 0) + position;

        // 分母の計算
        denominator = Mathf.Pow((positionB.x - positionA.x), 2) + Mathf.Pow((positionB.y - positionA.y), 2) + Mathf.Pow((positionB.z - positionA.z), 2);

        // キャッチするときに必要な値の計算
        positionABX = positionB.x - positionA.x;
        positionABY = positionB.y - positionA.y;
        positionABZ = positionB.z - positionA.z;
    }
```

・キャッチ動作が行われたときに実行する計算
動作が行われたときの枝との最短距離の交点を求めるため、プレイヤーの座標が必要な計算をしています。プレイヤーの座標は関数の引数として渡しました。

```
public Vector3 CatchPosition(float playerX, float playerY, float playerZ)
{
    // 分子の計算
    var numerator = ((playerX - positionA.x) * positionABX) + ((playerY - positionA.y) * positionABY) + ((playerZ - positionA.z) * positionABZ);

    // 分母分子で割った値を出す
    var k = numerator / denominator;

    // 地点AとBを結んでできる枝の真ん中を通る直線にプレイヤーから引いた法線が交わる座標を求める
    catchPosition = new Vector3((k * positionB.x) + ((1-k) * positionA.x), (k * positionB.y) + ((1- k) *  positionA.y), (k * positionB.z) + ((1- k) * positionA.z));

    // 枝の当たり判定よりも外側のcatchPosition 場合は枝の判定の端のPosition の値を入れる
    if (catchPosition.x < positionA.x)
    {
        catchPosition = positionA;
    }
    else if (catchPosition.x > positionB.x)
    {
        catchPosition = positionB;
    }

    // 計算した値を返す
    return catchPosition;
}
```

最終的に座標を求めた後はVector３型のreturnで値を返しています。

```
transform.position =
    catchRange.CatchPosition(transform.position.x, transform.position.y, transform.position.z);
```

プレイヤーのスクリプト側でreturnされた値を直接座標に入れています。

この実装によって

この実装によっていつキャッチ動作が行われたとしても右写真のように空中にいるように見えることがなくなりました。
この経験から関数の使い方についての理解が深まったと感じています。

Unity 2Dダークミュータントアクションゲーム

タイトル	リバイブ ザ クリーチャー Rivive The Creature
ターゲット	試行錯誤をして攻略に挑むのが好きな方
動作環境	PC
開発環境	Unity 2022.3.23f1
開発言語	C#
開発期間	1か月
開発人数	プランナー2人, デザイナー3人, プログラマー2人
担当箇所	プレイヤーキャラクター周り全般
ゲーム概要	『ステージ上の敵の倒し、敵の力を"捕食"で吸収、パワーアップを行いながら、敵をどんどんと倒していく2Dアクションゲーム』 敵の力を吸収して、強くなった状態で最終的にBossを倒すことを目的とした爽快感あふれるゲームです。

◇プレイヤーのHP管理

このゲームではプレイヤーと敵間でのHPの削り合いを
行うため、その実装をスムーズにする工夫をしました。

```
public interface IDamageable
{
    6 個の参照
    public void Damage(float damage);
}
```

インターフェース

```
// プレイヤーと衝突したとき
Unity メッセージ | 0 個の参照
private void OnCollisionEnter2D(Collision2D collision)
{
    var hit = collision.gameObject.GetComponent<IDamageable>();

    if (hit != null && isDead == false)
    {
        // ダメージを与える
        hit.Damage(20);
    }
}
```

敵の衝突判定

```
void IDamageable.Damage(float damage)
{
    // SEを流す
    DamageSE();

    // 現在HPからダメージ分減らす
    currentHp -= damage;
    hpSlider.value = currentHp;

    if (currentHp <= 0)
    {
        Death();

        player.Dead();
    }
}
```

プレイヤーのHP管理スクリプト

実装方法

インターフェースをい、
HPの削り合いをスムーズに
行えるようにしました。
インターフェース用の
スクリプトを作り、
プレイヤーと敵にそれぞれ
作成したインターフェースを
参照させることで一括での
HP管理を行うことが
できました。

この実装によって

この実装によりプレイヤーと敵のHP管理を楽にすることができました。
たとえ敵の種類や、攻撃力が変わったとしても値を変えるだけで
HP管理を行うことができるので拡張性にも優れています。