

ポートフォリオ

はままつ れんと

名前 : 浜松 廉斗
出身 : 福岡情報ITクリエイター専門学校
メール : renia5310@gmail.com
GitURL : [Hmmtrnt \(github.com\)](https://github.com/Hmmtrnt)
YouTube : <https://x.gd/72NIL>
趣味、特技 : ピアノ演奏/ゲーム/外食

ステータス

C++ : 2年
GitHub : 2年
Unity/C# : 1年
HTML : 2ヶ月
PHP : 2ヶ月



最終制作作品

※ドラゴンのスペルミスはタイトルの見栄えのためわざとミスしてます。

作品名 DoragonHunter
ジャンル アクションゲーム
製作期間 2023/10/09～
2024/03/31
制作環境 Unity/C#/
GitHub Desktop



制作人数

プログラマー 1名、モーションデザイナー 1名

UIデザイナー 1名

担当箇所 プログラミング全部

GitHub

<https://github.com/Hmmtrnt/DoragonHunter>

ゲームフロー

目標はドラゴンを狩ること



受注



出発



討伐



帰還

制作作品 駆け引き

モンスターの動きを見て回避を行うか攻撃をするか

モンスターは七種類の攻撃を仕掛けてくる。

モンスターの動きに合わせて隙の少ない攻撃や自身の立ち位置を見極め、回避する方向を判断し単調な戦闘にならないように駆け引きを楽しめるようにしている。

突進攻撃



噛みつき攻撃...etc



制作作品 駆け引き

モーションキャンセルの実装

基本的に回復や攻撃は大きな隙が生じるのため、被弾によるストレスを少なくするため途中でモーションを中断できるようになっている。
ただ攻撃を行った際にある程度隙が生じるようになっておりモンスターの行動を見極めて攻撃を行う駆け引きを楽しめるようにした。

回復を行っている



回復を中断した



制作作品 オブジェクトの状態制御(StateBase)

各状態クラスの遷移

挙動制御を行う抽象クラス、StateBaseを作成した。

StateBaseによって各状態を管理するクラス間の遷移をわかりやすくした。

StateBaseの引数

引数にひとつ前の状態、次に遷移する状態を追加することで特定の状態遷移時にのみ行いたい処理を追加できるようにした

次の状態に遷移する関数

```
/// <summary>
/// 状態遷移.
/// </summary>
/// <param name="nextState">次に遷移する状態</param>
88 個の参照
private void StateTransition(StateBase nextState)
{
    // 状態終了時.
    _currentState.OnExit(this, nextState);
    // 次の状態の呼び出し.
    nextState.OnEnter(this, _currentState);
    // 次に遷移する状態の代入.
    _currentState = nextState;
}
```

StateBase.cs

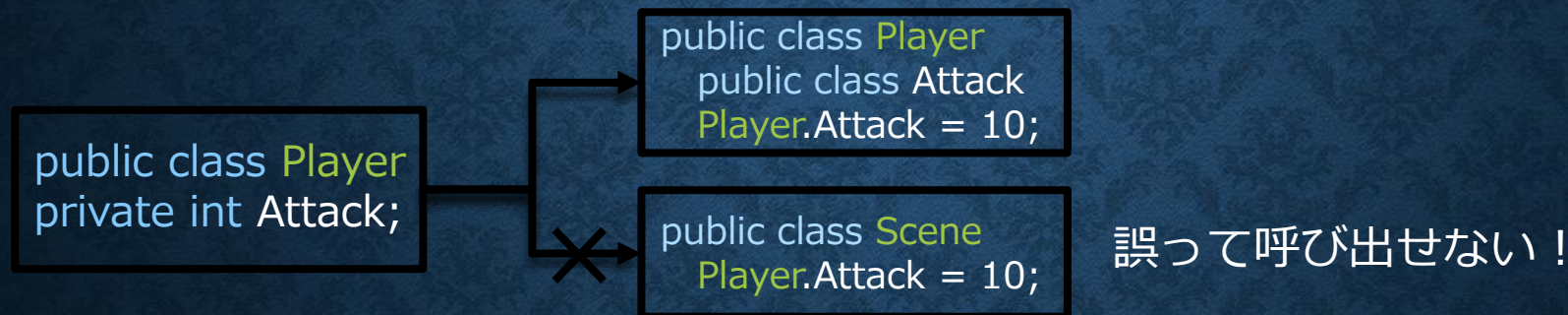
```
1 // 状態管理の抽象クラス
2
3 99+ 個の参照
4 public abstract class StateBase
5 {
6     /*プレイヤー*/
7     /// <summary>
8     /// ステート開始時呼び出し
9     /// </summary>
10    /// <param name="owner">アクセスするための参照</param>
11    /// <param name="prevState">ひとつ前の状態</param>
12    24 個の参照
13    public virtual void OnEnter(Player owner, StateBase prevState) { }
14    /// <summary>
15    /// Update
16    /// </summary>
17    /// <param name="owner">アクセスするための参照</param>
18    23 個の参照
19    public virtual void OnUpdate(Player owner) { }
20    /// <summary>
21    /// FixedUpdate
22    /// </summary>
23    /// <param name="owner">アクセスするための参照</param>
24    23 個の参照
25    public virtual void OnFixedUpdate(Player owner) { }
26    /// <summary>
27    /// ステート終了時呼び出し
28    /// </summary>
29    /// <param name="owner">アクセスするための参照</param>
30    /// <param name="nextState">次に遷移する状態</param>
31    22 個の参照
32    public virtual void OnExit(Player owner, StateBase nextState) { }
33    /// <summary>
34    /// ステート遷移の呼び出し
35    /// </summary>
36    /// <param name="owner">アクセスするための参照</param>
37    22 個の参照
38    public virtual void OnChangeState(Player owner) { }
39 }
```


制作作品 プレイヤー (内部クラス)

各状態クラスを内部クラスとして宣言

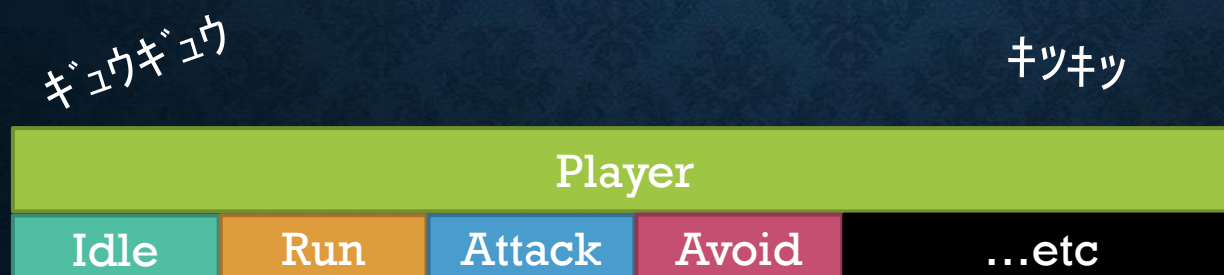
内部クラスにしたメリット

privateフィールドにある変数や関数を各状態クラスのみ呼び出すことができ、その他のクラスに必要な変数や関数を呼び出すことを防ぐことが出来た。そのため状態制御関係のバグの発生率を抑えることが出来た。



内部クラスにしたデメリット

コードの肥大化がおこり、可読性の低下と制作効率を損なうことになった。



制作作品 プレイヤー (partial class)

コードの肥大化を抑える

partial classを使うことで各状態クラスを別々のcsファイルに分けることに成功した。

そのため、前述で説明した内部クラスにしたデメリットを解消できた
可読性の向上や制作の効率にもつながった

Player.cs

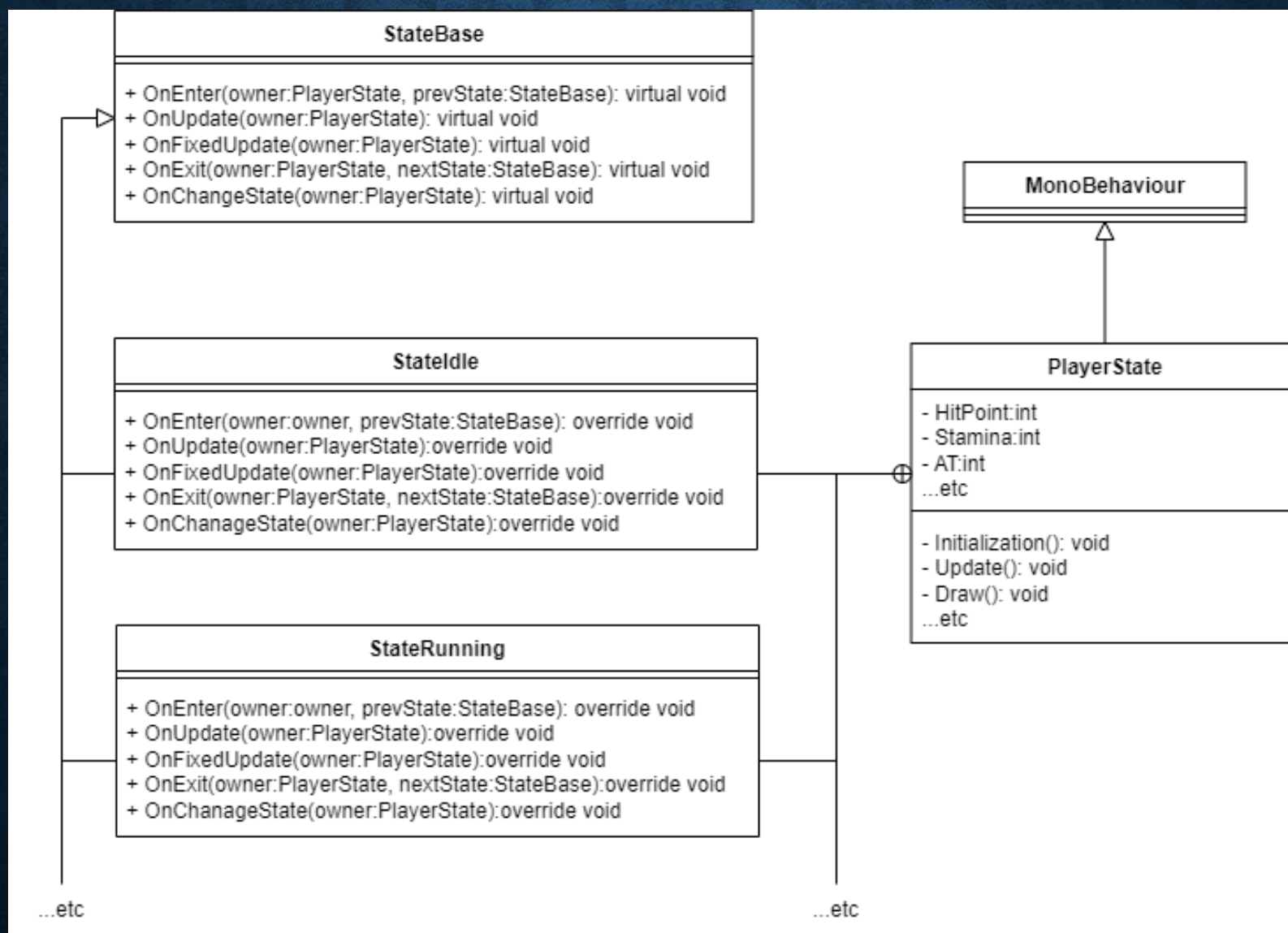
```
/*プレイヤー*/  
  
using UnityEngine;  
  
Unity スクリプト (3 件のアセット参照) 199+ 個の参照  
public partial class Player : MonoBehaviour  
{  
    //---納刀状態---//  
    private static readonly StateIdle _idle = new(); // アイドル。  
    private static readonly StateAvoid _avoid = new(); // 回避。  
    private static readonly StateRunning _running = new(); // 走る。  
    private static readonly StateDash _dash = new(); // ダッシュ。  
    private static readonly StateFatigueDash _fatigueDash = new(); // 疲労時のダッシュ。  
    private static readonly StateRecovery _recovery = new(); // 回復。  
  
    //---抜刀状態---//  
    private static readonly StateDrawnSwordTransition _drawnSwordTransition = new(); // 抜刀する。  
    private static readonly StateIdleDrawnSword _idleDrawnSword = new(); // アイドル。  
    private static readonly StateRunDrawnSword _runDrawnSword = new(); // 走る。  
    private static readonly StateAvoidDrawnSword _avoidDrawnSword = new(); // 抜刀回避。  
    private static readonly StateRightAvoidDrawnSword _rightAvoid = new(); // 攻撃後の右回避。  
    private static readonly StateLeftAvoidDrawnSword _leftAvoid = new(); // 攻撃後の左回避。  
    private static readonly StateSheathingSword _sheathingSword = new(); // 納刀する。  
    private static readonly StateSteppingSlash _steppingSlash = new(); // 踏み込み斬り。  
    private static readonly StatePiercing _piercing = new(); // 突き。  
    private static readonly StateSlashUp _slashUp = new(); // 斬り上げ。  
    private static readonly StateSpiritBlade1 _spiritBlade1 = new(); // 気刃斬り1。  
    private static readonly StateSpiritBlade2 _spiritBlade2 = new(); // 気刃斬り2。  
    private static readonly StateSpiritBlade3 _spiritBlade3 = new(); // 気刃斬り3。  
    private static readonly StateRoundSlash _roundSlash = new(); // 気刃大回転斬り。  
}
```

StateIdle.cs

```
/*待機状態*/  
  
using UnityEngine;  
  
Unity スクリプト 199+ 個の参照  
public partial class Player  
{  
    2 個の参照  
    public class StateIdle : StateBase  
    {  
        3 個の参照  
        public override void OnEnter(Player owner, StateBase prevState)  
        {  
            // アニメーション開始。  
            owner._idleMotion = true;  
  
            if (prevState == _sheathingSword)  
            {  
                owner._motionFrame = 0;  
            }  
        }  
    }  
}
```

Playerクラスに各状態クラスを宣言し呼び出している

制作作品 簡易クラス図(プレイヤー状態制御)



制作作品 モンスター (基本システム)

プレイヤーのステートパターンを流用

前述で紹介したプレイヤーの基本システムの`StateBase`を流用し、状態制御の管理をプレイヤーとほぼ変わらない状態にした。

Monster.cs

```
/*モンスター*/  
  
using UnityEngine;  
using UnityEngine.UI;  
  
Unity スクリプト (3 件のアセット参照) 199+ 個の参照  
public partial class Monster : MonoBehaviour  
{  
    public static readonly MonsterStateRoar _roar = new(); // 咆哮.  
    public static readonly MonsterStateIdle _idle = new(); // アイドル.  
    public static readonly MonsterStateRun _run = new(); // 移動.  
    public static readonly MonsterStateDown _down = new(); // やられる.  
  
    public static readonly MonsterStateAt _at = new(); // 攻撃(デバッグ用).  
    public static readonly MonsterStateRotateAttack _rotate = new(); // 回転攻撃.  
    public static readonly MonsterStateBless _bless = new(); // ブレス攻撃.  
    public static readonly MonsterStateBite _bite = new(); // 噛みつき攻撃.  
    public static readonly MonsterStateRushForward _rush = new(); // 突進攻撃.  
    public static readonly MonsterStateWingBlowRight _wingBlowRight = new(); // 右翼攻撃.  
    public static readonly MonsterStateWingBlowLeft _wingBlowLeft = new(); // 左翼攻撃.  
    public static readonly MonsterStateTailAttack _tail = new(); // 尻尾攻撃.
```

MonsterStateIdle.cs

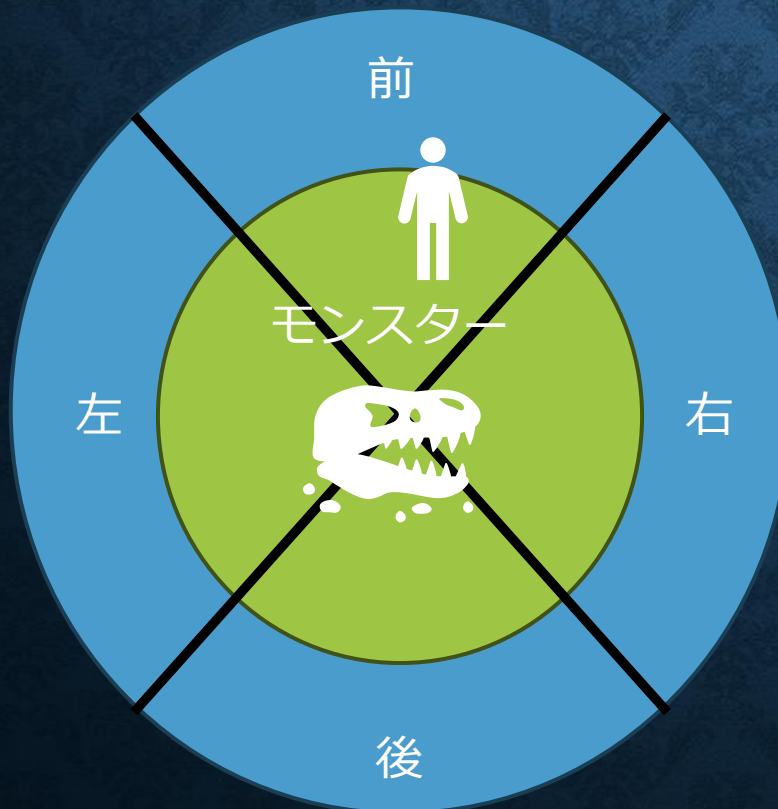
```
/*モンスターのアイドル*/  
  
using UnityEngine;  
  
Unity スクリプト 199+ 個の参照  
public partial class Monster  
{  
    2 個の参照  
    public class MonsterStateIdle : StateBase  
    {  
        3 個の参照  
        public override void OnEnter(Monster owner, StateBase prevState)  
        {  
            owner.StateTransitionInitialization();  
            // アニメーション開始.  
            owner._idleMotion = true;  
        }  
    }  
}
```

Monsterクラスに各状態クラスを宣言し呼び出している

制作作品 モンスター (AIの制御)

モンスターに視野角を作成

モンスターに視野角を追加することで行動パターンを制御している
モンスター側が常にプレイヤーの位置情報を取得し、
正面、右、左、後ろのどの方向にいるのかそして、モンスターとプレイヤーの
距離にもよってモンスターの攻撃の行動パターンを制御している



この場合正面の近距離にいるので
正面に向かって近距離攻撃を行うようにする

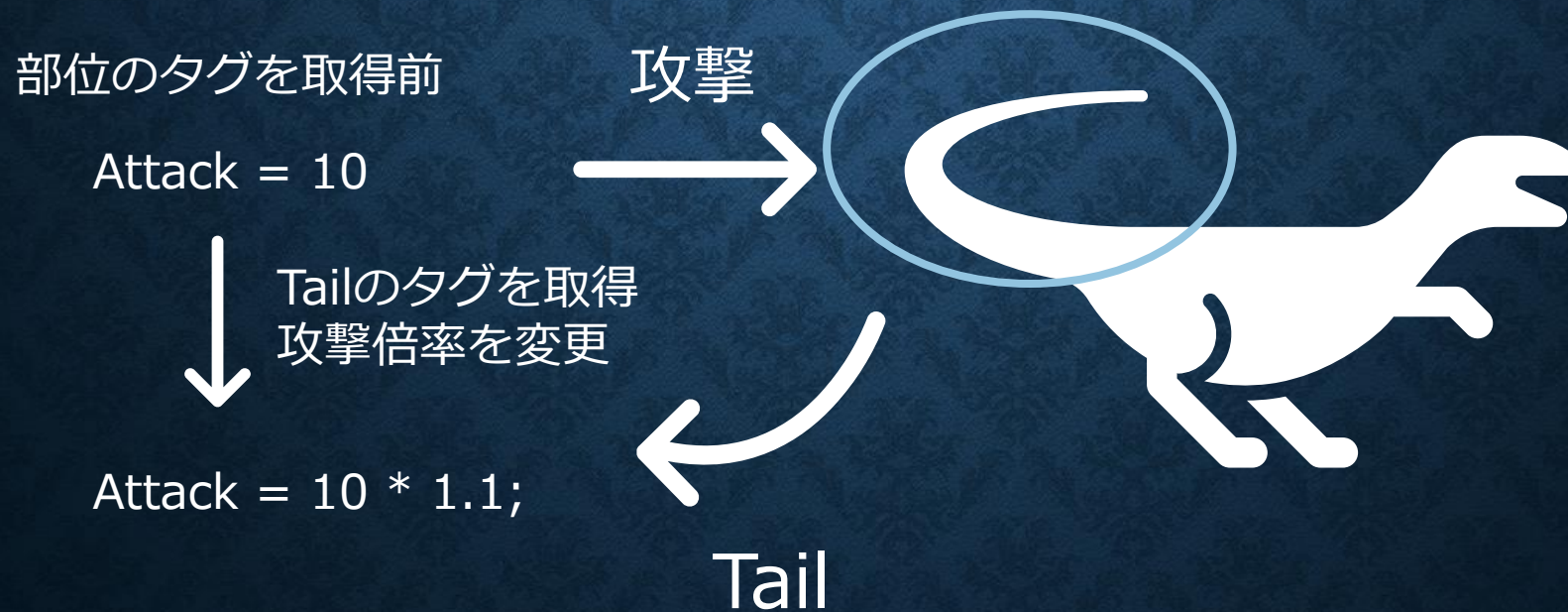
近距離

遠距離

制作作品 モンスター (当たり判定)

モンスターにメッシュコライダーを追加

モンスターの当たり判定はアセットのSAColliderBuilderを使用した。
モンスターに設置された当たり判定にタグを付与して肉質を管理している。
武器側でモンスターの肉質の情報を取得し、攻撃力を変化させている。



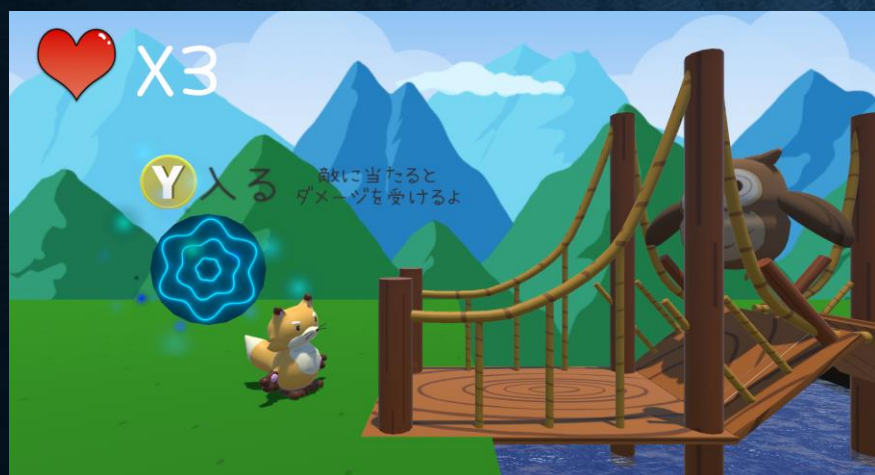
制作実績



作品名 キツネたろう二世の大冒険
ジャンル アクションゲーム
製作期間 2023/09/04～2023/12/08
制作環境 Unity/C#
対応機種 Windows
制作人数 プログラマー 6名
モデラー・モーション 5名
担当 バージョン管理、プレイヤーの制御、
2D視点のステージ作成 シーン管理
目的、学んだこと
Unityを使用したチーム制作の流れ
GFFアワードに提出

GitURL

<https://github.com/fkogcc/FoxProject.git>



制作実績



作品名	Don't Fall
ジャンル	3Dアクションゲーム
製作期間	2023/06/12～2023/08/11
制作環境	DXライブラリ/C++
対応機種	Windows
制作人数	プログラマー 1名
担当	背景と3Dモデル以外
目的、学んだこと	3Dゲーム制作の基礎知識

GitURL

<https://github.com/Hmmtrnt/3DAvoid.git>



制作実績



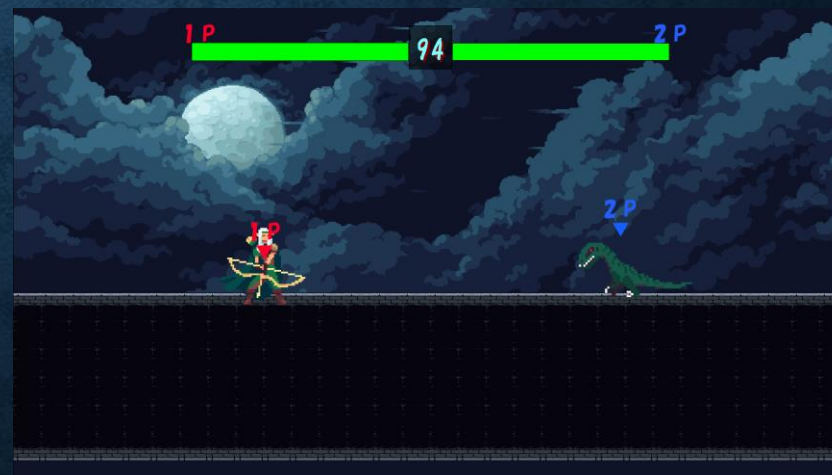
作品名	MFM
ジャンル	2D格闘ゲーム
製作期間	2023/04/17～2023/06/09
制作環境	DXライブラリ/C++
対応機種	Windows
制作人数	プログラマー 5名
担当	プログラムチーフ、バージョン管理、 一体のプレイヤーの作成

目的、学んだこと

チーム制作の基礎知識
GitHubDesktopの知識

GitURL

https://github.com/fkogcc/ProjectK_2.git



制作実績

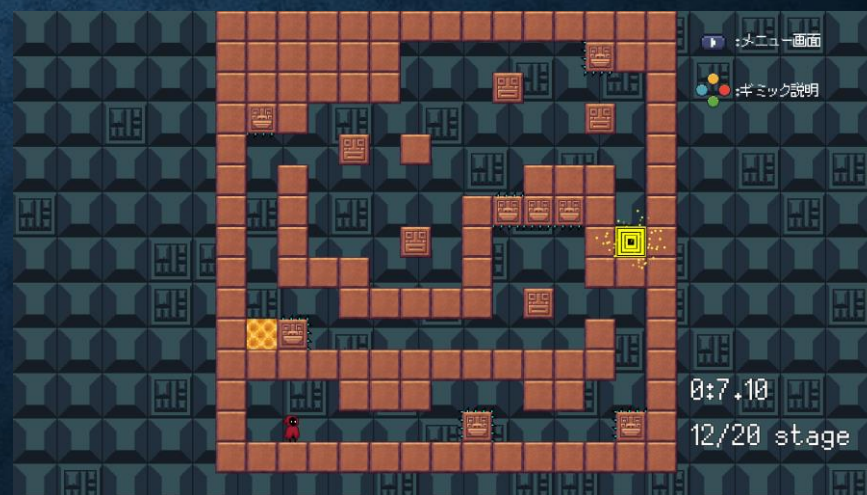
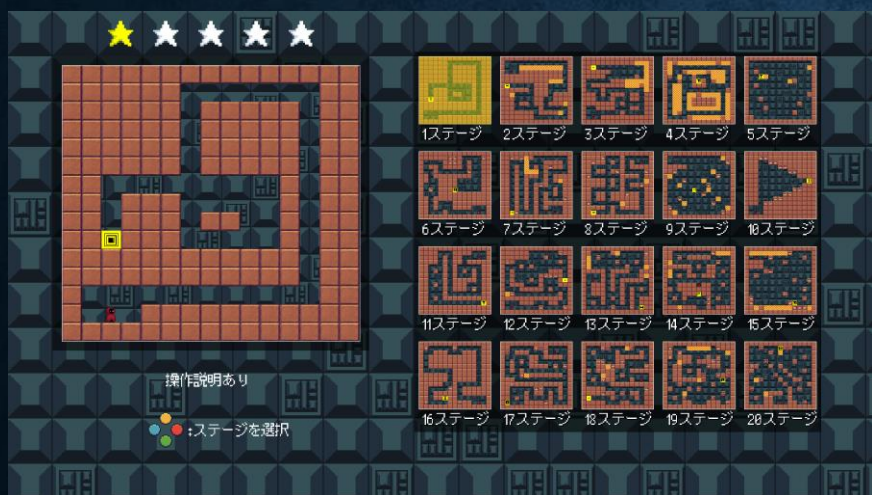


作品名	JumpLabyrinth
ジャンル	2Dパズルゲーム
製作期間	2023/01/09～2023/03/17
制作環境	DXライブラリ/C++
対応機種	Windows
制作人数	プログラマー 1名
担当	グラフィック以外

目的、学んだこと
ゲームの制作の流れ
スケジュール管理

GitURL

<https://github.com/Hmmtrnt/JumpLabyrinth2.git>



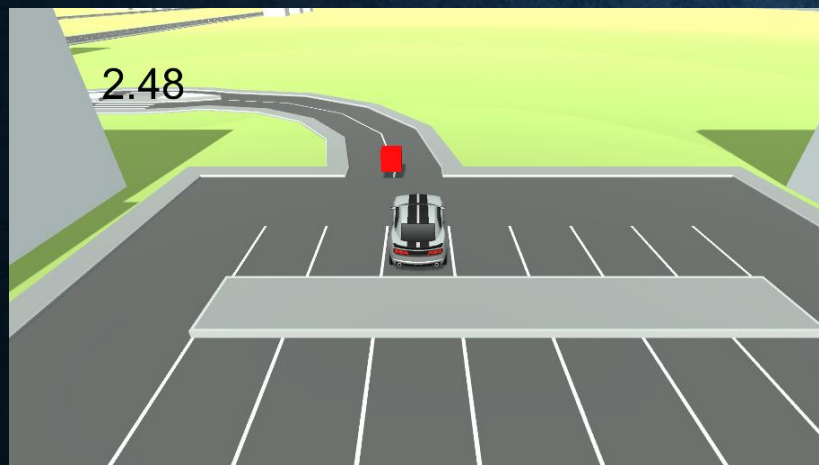
制作実績

作品名
ジャンル
製作期間
制作環境
対応機種
制作人数
担当

ブロックを壊せ
シミュレーションゲーム
2022/08/15～2022/08/31
Unity/C#
Windows
プログラマー 1名
車の制御以外のプログラム全般

目的、学んだこと

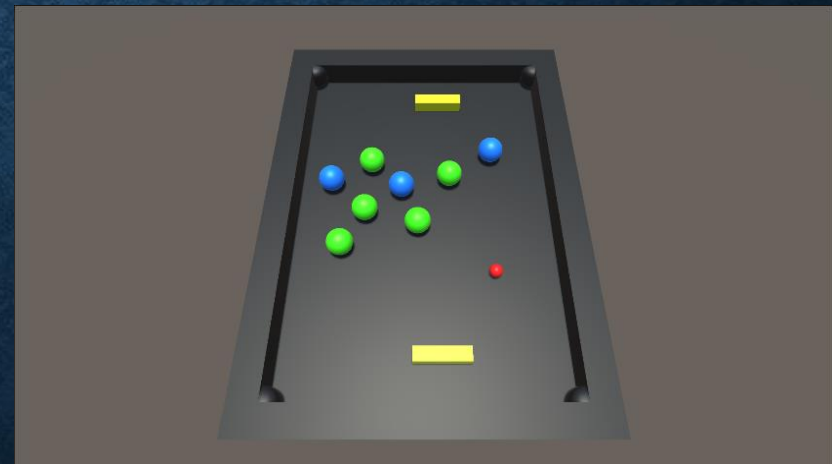
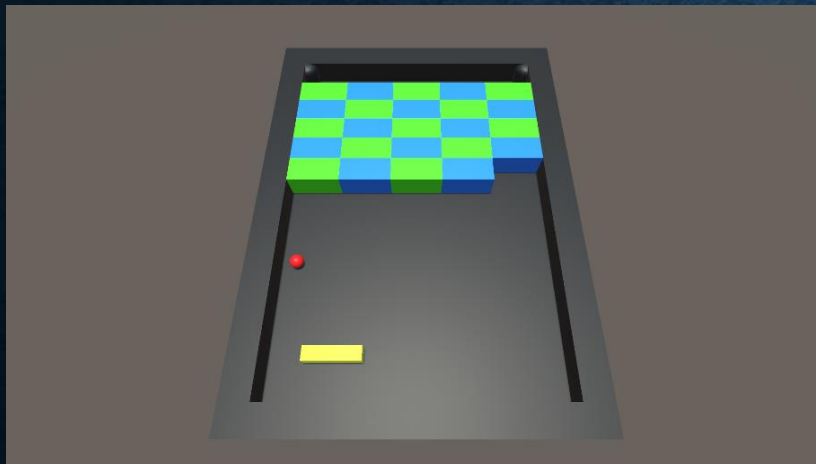
C#の基礎知識
アセットの使用方法



制作実績

作品名	ブロック崩し
ジャンル	ブロック崩し
製作期間	2022/08/01～2022/08/12
制作環境	Unity/C#
対応機種	Windows
制作人数	プログラマー 1名
担当	プログラム全般

目的、学んだこと
Unityの基礎知識



今後の目標

私はプレイアブルキャラクターの挙動制御を担うプログラマーになりたいと考えてる。

そのため、よりリアリティある挙動制御を実現するために物理や数学的知識を学び続ける。