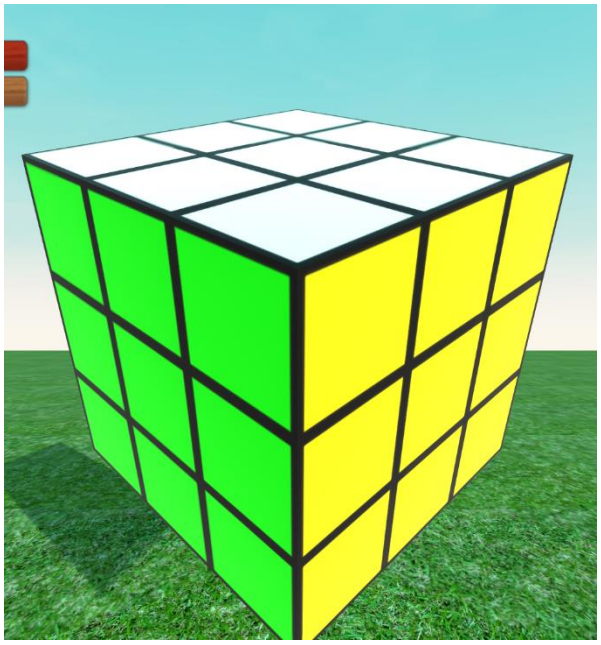
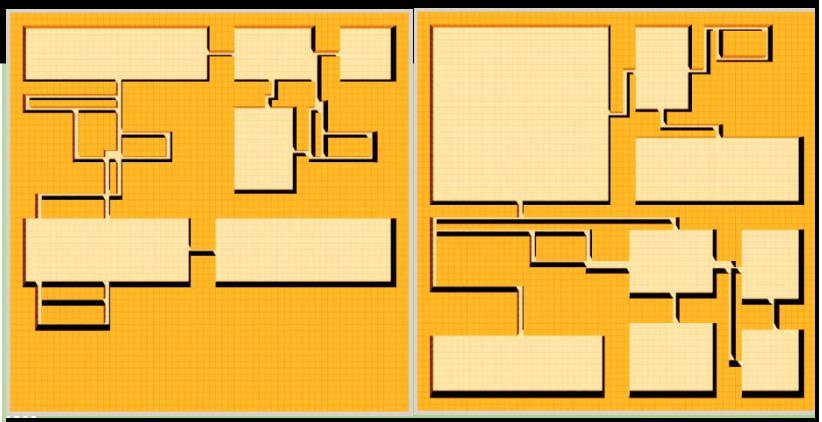
Portfolio





吉中　大貴

ヒューマンアカデミー大宮校

ゲームカレッジプログラマー専攻

プロフィール

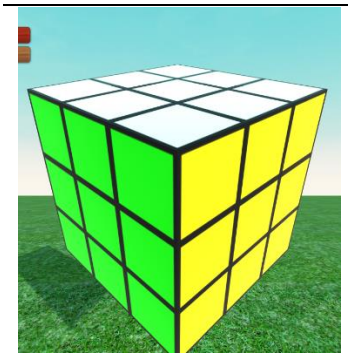
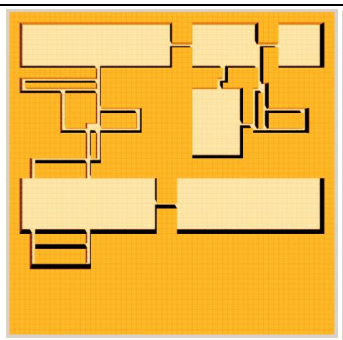
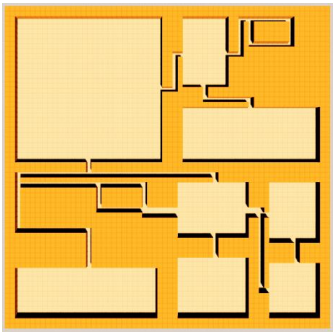
よしなか　ひろき 吉中　大貴	
希望職種	プログラマー
年齢	21 歳

	所有スキル
言語	C/C++,C#,(勉強中:JavaScript,PHP)
開発環境・ 使用ツール	Microsoft Visual Studio 2017, 2019 Unity 2018.4.23.f1, 2019.4.2f1 Git(Sourcetree, TortoiseGit), Github Microsoft Office Excel, Word, PowerPoint Slack, Chatwork, Google スプレッドシート Xampp(Apache)
資格	基本情報技術者試験　(2019 年秋季 日本漢字能力検定 2 級

自己 PR
好きなゲーム ： 風来のシレン、パズル＆ドラゴンズ、モンスターハンターシリーズ、DEATH STRANDING など 好きな漫画トップ３ ： １：HUNTER×HUNTER　２：バキ　３：呪術廻戦 　自分のやりたいことがわからなくなり進学校を中退した後自分を見つめなおし、小学生時代はゲームを作りたかったことを思い出してゲームカレッジに飛び込みました。夢を叶えるための技術の裏付けとして基本情報技術者試験に挑戦し合格しています。ユーザーの「楽しい」を支えるため、地道な努力を積み重ねていく所存です。

目次

個人制作：Unity3D パズルゲーム制作　4p

個人制作：自動生成ダンジョン　9p

DirectX：BMP 画像読み込み　10p

チーム：Unity2D アクションゲーム制作　11p


チーム：日本ゲーム大賞アマチュア部門応募作品　15p

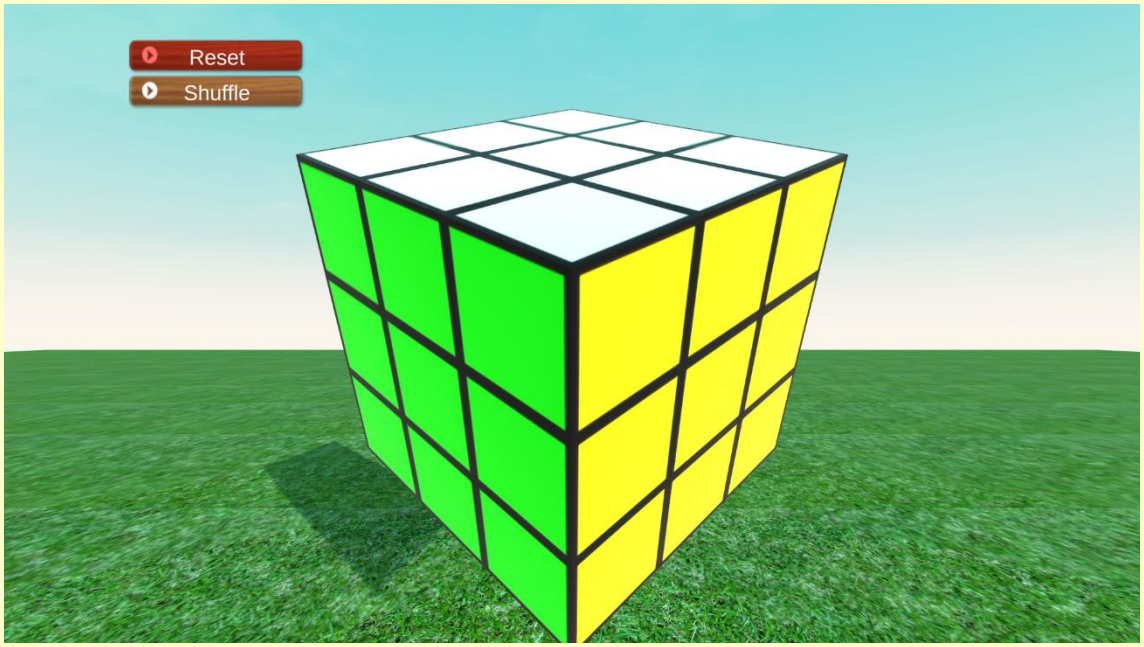
個人制作：Unity3D パズルゲーム制作

開発環境 ： Unity2018.4.13f1(64bit)
Microsoft VisualStudio2017

使用言語 ： C#

動作環境 ： Windows10

開発期間 ： 約 5 日（2020 年 2 月



作品概要 ： ルービックキューブを Unity 上で再現しました。

操作方法 ：

WASD でキューブ全体を回転させ、6 面全てを確認できます。

パズルの回転はマウスでドラッグ＆ドロップです。

既存のスマートフォンアプリなどの操作を研究し、決定しました

製作動機など：

発表展示会までに短期間で個人製作ゲームを一本制作する必要がありました。見かけ上 3D になるデータ構造を管理し、表示されるべき位置に表示されるよう構造への理解を深めました

定数定義、データ配列

パズルデータは[6 面*3*3]の SixColors enum 配列として管理しています。それとは別にワールドに表示する Cube を[3*3*3]の配列として管理しています。また、パズルデータに対応するデータ表示先として Cube の子オブジェクトの一面のみを参照する配列を[6*3*3]の配列で管理しています。

色データを更新する際は renderTargets

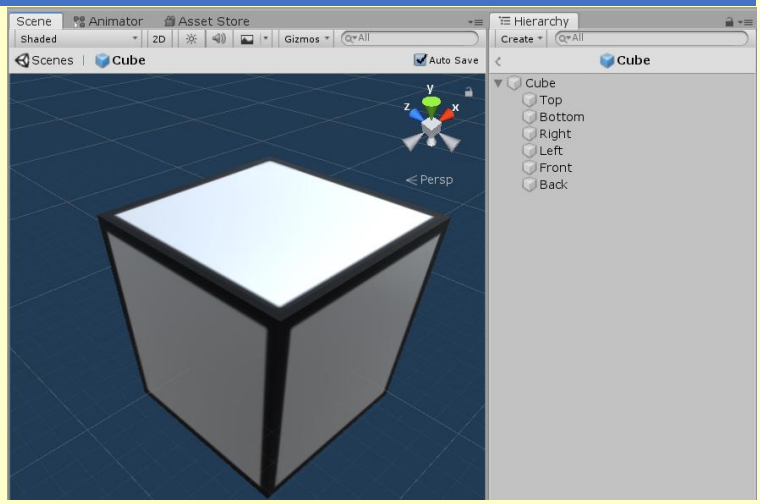
位置データを更新（回転）する際は cubes

役割と形の異なる配列を管理しています。

```
/// <summary>
/// パズルのデータ、ロジックの管理クラス
/// </summary>
public class PuzzleManager : MonoBehaviour
{
    static readonly int cubeLength = 3;
    // 6面
    enum SixSides
    {
        Top = 0, Bottom, Right, Left, Front, Back, MAX
    }
    // 6色、パズルはこの色の配列として考えます
    enum SixColors
    {
        White, Orange, Red, Green, Yellow, Blue, MAX
    }
    // 色定義
    readonly Color[] colorPattern = new Color[6]
    {
        //        R     G     B
        new Color(1.0f, 1.0f, 1.0f),   // White
        new Color(1.0f, 0.3f, 0.1f),   // Orange
        new Color(1.0f, 0.0f, 0.0f),   // Red
        new Color(0.0f, 1.0f, 0.0f),   // Green
        new Color(1.0f, 1.0f, 0.0f),   // Yellow
        new Color(0.0f, 0.0f, 1.0f),   // Blue
    };

    // puzzle全体の親オブジェクト
    GameObject puzzle = null;
    // 描画に使用するCubeオブジェクトの配列(3*3*3)
    GameObject[,,] cubes = new GameObject[cubeLength, cubeLength, cubeLength];
    // 実際のデータ保持配列(6面*3*3)
    SixColors[,,] dataTable = new SixColors[(int)SixSides.MAX, cubeLength, cubeLength];
    // データ描画面のみを取り出しておく
    Renderer[,,] renderTargets = new Renderer[(int)SixSides.MAX, cubeLength, cubeLength];
```

Cube.prefab(各面に異なる色を設定)



6 面それぞれに別の色を設定するために、手軽な方法として Quad を 6 枚組み合わせることで Cube を　表現しました。ルービックキューブの枠線は、GIMP で簡単なグレースケールのテクスチャを用意（所要時間 1 分）し Material にテクスチャを設定して各面に貼り付けています。

データ配列とワールド上 Cube の展開図の関係

			0,0	0,1	0,2						
	l,j		1,0	Top	1,2						
			2,0	2,1	2,2						
0,0	0,1	0,2	0,0	0,1	0,2	0,0	0,1	0,2	0,0	0,1	0,2
1,0	Left	1,2	1,0	Front	1,2	1,0	Right	1,2	1,0	Back	1,2,
2,0	2,1	2,2	2,0	2,1	2,2	2,0	2,1	2,2	2,0	2,1	2,2,
			0,0	0,1	0,2						
			1,0	Botto	1,2						
			2,0	2,1	2,2						

Cube の展開図に対して、データをこのように貼り付けています。

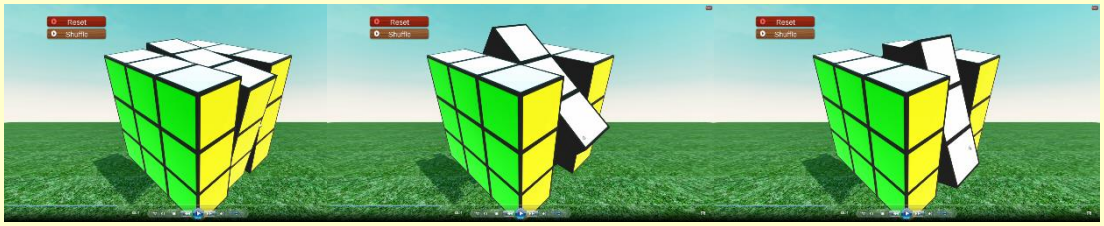
```
// dataTableをカラーパターンに変換してRendererのマテリアルに設定
renderTargets[side, i, j].material.color = colorPattern[(int)dataTable[side, i, j]];
```

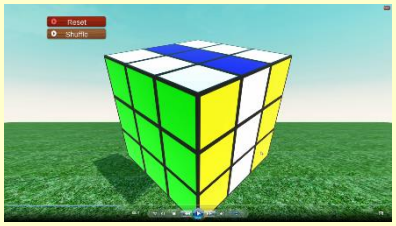
colorPattern 定数配列から dataTable の SixColors enum に応じた色データを取り出し、データ表示先の Renderer の material.color に設定することでグレースケール画像の色変更を行い、少ないリソースでカラフルなキューブを表現しています。

回転アニメーション表現

中心を回転軸として各キューブの回転移動を行い、回転アニメーションを表現しています。ここでのワンポイントは回転移動アニメーションの終了後に位置と回転をリセットし、その後に更新されたデータ配列から色データの更新を行っていることです。

アニメーション中は回転前のデータが表示され・・・





アニメーションが終了したと同時にデータ反映＆Cube の回転・位置をリセット。
こうすることで、回転させる際常に同じ Cube オブジェクトを参照すればよいのでデータ管理が単純になりました

```
// オブジェクトの回転アニメーション
while (true)
{
    const int y = 2;
    // タイム加算
    _time += Time.deltaTime;
    // アニメーションの終了
    if (_time > duration)
    {
        // 回転と位置をリセット
        for (int x = 0; x < cubeLength; x++)
            for (int z = 0; z < cubeLength; z++)
            {
                cubes[x, y, z].transform.localRotation = Quaternion.Euler(0, 0, 0);
                cubes[x, y, z].transform.localPosition = new Vector3(x - 1, y - 1, z - 1);
            }
        break;
    }
    // (0,2,0)~(2,2,2)
    for (int x = 0; x < cubeLength; x++)
        for (int z = 0; z < cubeLength; z++)
            cubes[x, y, z].transform.RotateAround(
                cubes[1, y, 1].transform.position,
                transform.up,// 全体（Managerの上方向
                deg / duration * Time.deltaTime);
    yield return null;
}
```

(switch-case 文の 1case を抜粋)

改善点

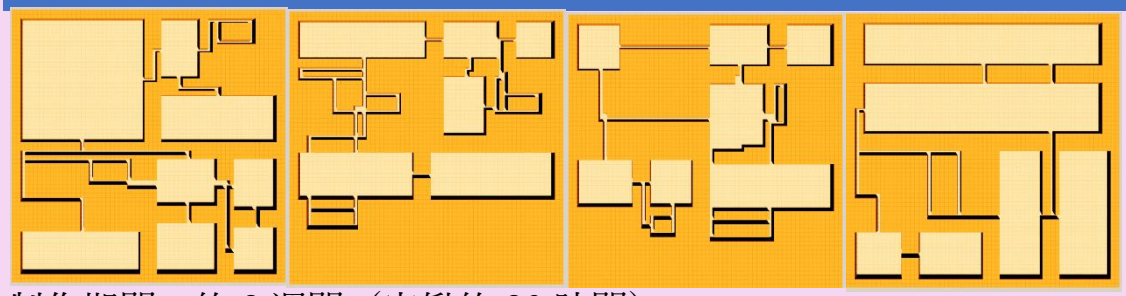
・パズルのデータ管理とプレイヤーの入力は別スクリプトに切り分けるべき
→UI を他人が実装することを考えた際、一つの Manager の中に詰め込まれすぎている。そもそも Manager と命名したことが曖昧だった。
スクリプトを作成した時点で責務の範囲を考えておくべきだった

・switch-case 分岐が長く、重複しているコードが非常に多い。
そのために修正コストが高くなってしまった。
→回転を軸・方向で一般化し関数化することが間に合わなかった。
制作期間が足りず完成を急いだことでコードの余裕がなくなってしまった。
改善案→１．十分な制作期間の確保、残念ながら現実的ではないことも多い
　　２．数学的な素養を鍛えること、長期的な目標としていく。

・データを三次元配列で実装したが、json 形式などでの保存を考えると一次元配列で実装しておきたかった。

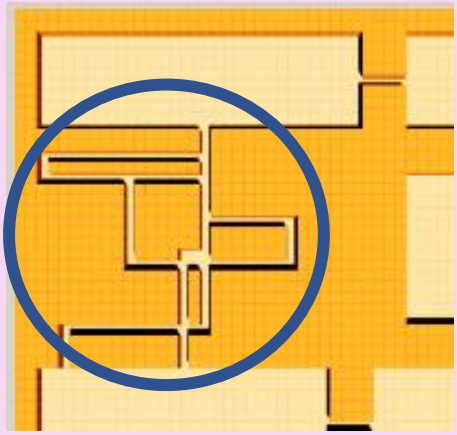
今回の失敗で「その場しのぎのコード」の危険性を痛感しました。

個人制作：自動生成ダンジョン



制作期間：約 2 週間（実働約 20 時間）

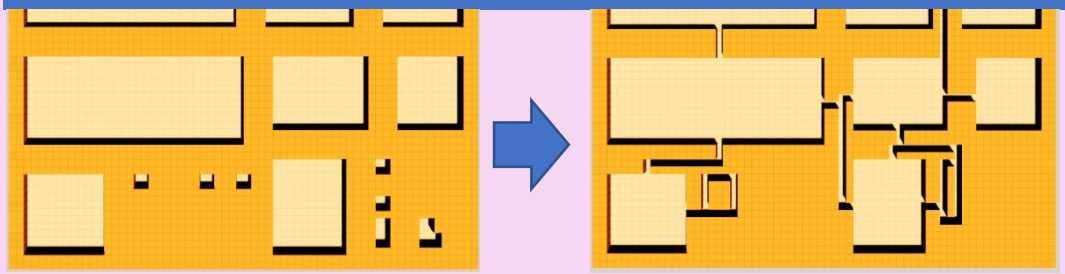
概要：「不思議のダンジョン」の制作途中のものです。



こだわりポイント

・部屋が一本道にならない、
　かつ繋がりすぎないこと。
・左図〇で囲んだように通路の複雑な
　部分が生まれること。（後述）

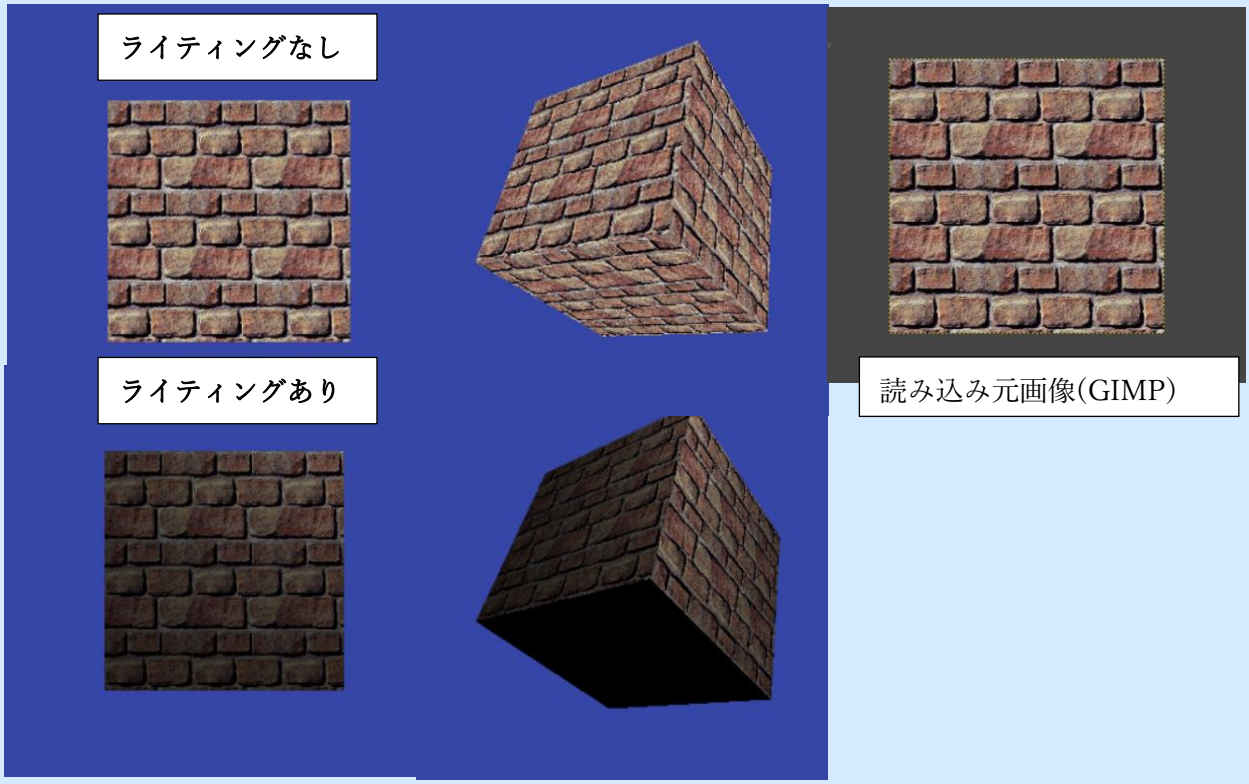
通路生成の一工夫



ランダムに部屋割りを行った後、一部の部屋サイズを 1 にしています。これで部屋を単なる「通路の分岐点」として扱うことで部屋に直接繋がらない複雑な通路が生成されます。

今後敵キャラクター、アイテム、罠の追加を行っていく予定です。

DirectX：BMP 画像読み込み



WindowsBMP(32bitBGRA)画像を読み込み、3D 空間のキューブへ貼り付けて表示しました。

```
//                  : 0123
// BMPは            : BGRA
// 扱いたいのは     : RGBA
// 一ピクセルごとに変換していく
for (int index = 0; index < sourceSize; index += sizeof(uint32_t)) {
    auto temp = *(source + index + 0);
    *(source + index + 0) = *(source + index + 2);
    *(source + index + 2) = temp;
}
```

BMP 形式では色データが BGRA と並んでいますが、
使用したいのは RGBA 形式だったため 1 ピクセルごとに変換を行っています。BMP 形式のファイルヘッダーフォーマットの読み取り、チャンネルの変換への知識を深めることができました。
今後は PNG 形式の読み込みなどに挑戦していきたいと思います。

チーム：Unity2D アクションゲーム制作

タイトル　：　魂殺（たまころ）

開発環境　：　Unity2018.4.13f1(64bit)
　　Microsoft VisualStudio2017

使用言語　：　C#

動作環境　：　Windows10

開発期間　：　4 ヶ月(～2020 年 2 月

制作人数　：　プランナー　：２人
　　グラフィック：２人
　　プログラム　：２人　計６人

担当範囲　：　プログラム部分のほぼ全て
　　もう一人にはデバッグの一部で協力してもらいました

ジャンル　：　２D 横スクロールアクションゲーム

コンセプト

　避けて集めて敵を倒せ！

作品概要

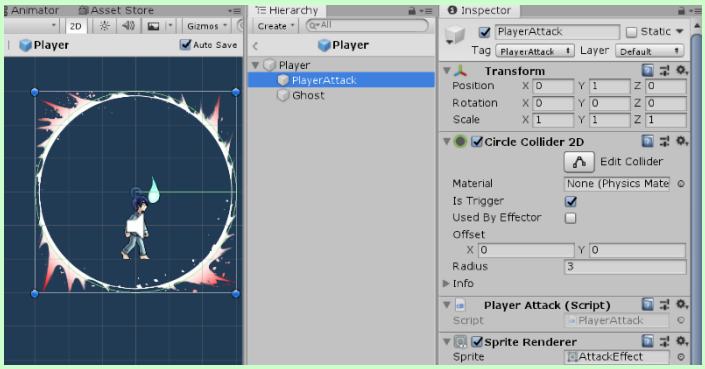
　操作する男の子は逃げることしかできない。
　妖気を集めて侍の幽霊の助けを借りて敵を倒そう。

操作方法↑W（ジャンプ）左←A・D→右　　スペースキーで攻撃

Player の攻撃アニメーション

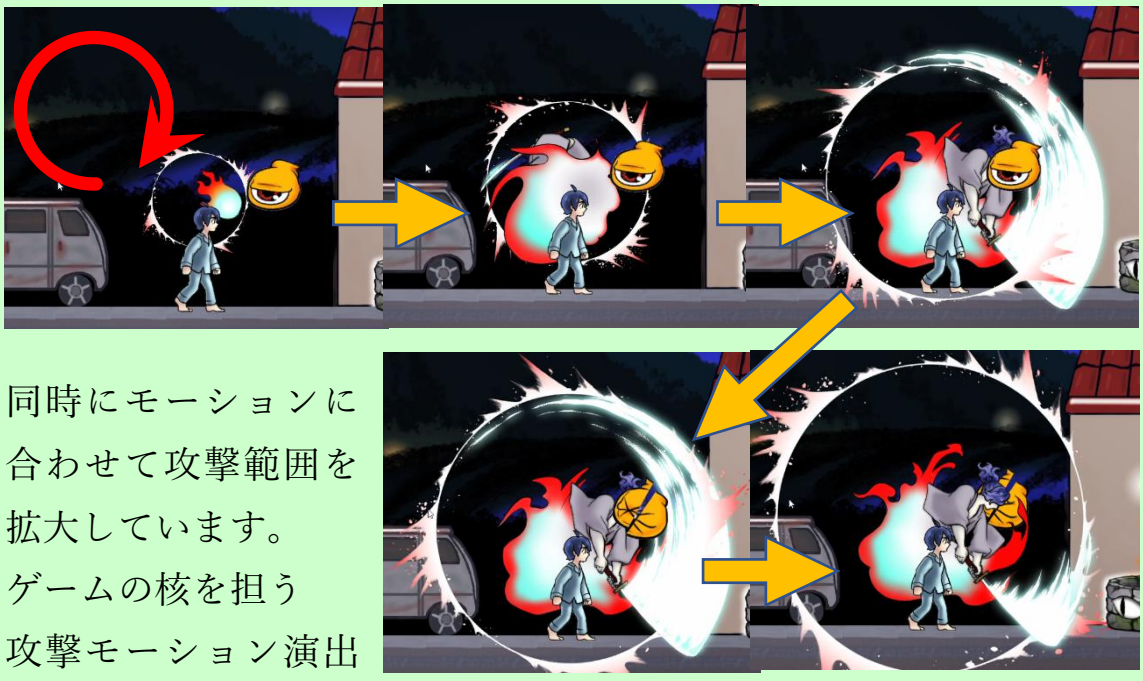
このゲームでは侍が袈裟切りに刀を振り下ろして円状の攻撃判定が発生します。

攻撃判定＆エフェクト画像は Player の子オブジェクトとなっています。



初めはこのオブジェクトを単純に SetActive(true) していたのですが、「刀を振り下ろして敵を斬っている」感は全く出ませんでした。

そこで、回転と拡大を同時に行うことで斬撃の表現が行えないかと試みました。刀を振り下ろす方向に合わせて回転させています。



同時にモーションに合わせて攻撃範囲を拡大しています。

ゲームの核を担う攻撃モーション演出を担当し、アニメーションプログラムの楽しさを実感しました。

攻撃アニメーション回転＆拡大の実装

今回は Sprite 以外のアニメーションをプログラムで実装することにこだわりました。回転、拡大それぞれについて(総変化量)/(時間)で一秒あたりの変化量を算出し、それに Time.deltaTime を乗算することで 1 フレームあたりの変化量を求め、毎フレーム加算することで一定速度での回転・拡大を行います。

この 1 フレームあたりの変化量を求めて加算していく方法をシーン前後のフェードアウトやチュートリアルでのカメラのパンなどにも応用しています。

```
    // 回転させながら拡大
    // 回転
    var startRotation = 0f;
    var endRotation = sign * -360.0f;
    // 拡大
    var startScale = Vector3.zero;
    var endScale = Vector3.one;
    // 変化/秒
    var changeScalePerSecond = (endScale - startScale) / _attackTime;
    var changeEulerZPerSecond = (endRotation - startRotation) / _attackTime;
    // 始点からスタート
    var _scale = startScale;
    var _eulerAngles = transform.localEulerAngles;
    // 時間管理用変数
    float _time = 0;
    // 毎フレーム実行する処理
    while (true)
    {
        _time += Time.deltaTime;
        // 変化/秒にTime.deltaTimeでフレーム変化量
        // 拡大
        _scale += changeScalePerSecond * Time.deltaTime;
        // 1に丸め処理
        if (_scale.x > 1)
            _scale = Vector3.one;
        transform.localScale = _scale;
        // 回転
        transform.Rotate(new Vector3(0, 0, 1), changeEulerZPerSecond * Time.deltaTime);
        if (_time > _attackTime)
            break;
        yield return null;
    }
```

Enemy の死亡演出

Enemy が攻撃を受けて消える際、グラフィッカーの作成したアニメーションに追加して演出効果を狙いました。Scale 縮小と SpriteRenderer のアルファ値減少を組み合わせることで自然に消えていく演出となっています。

```
// 色変化
var startAlpha = 1f;
var endAlpha = 0f;
// スケール変化
var startScale = Vector3.one;
var endScale = Vector3.zero;
// 変化量計算
var changeAlpha = endAlpha - start
var changeScale = endScale - start
float _time = 0;
// 1秒間でアニメーションする
const float duration = 1.0f;
// 毎秒
var alphaPerSecond = changeAlpha / duration;
var scalePerSecond = changeScale / duration;
// startから加算していく
var _alpha = startAlpha;
var _scale = startScale;
while (true)
{
    _time += Time.deltaTime;
    _alpha += alphaPerSecond * Time.deltaTime;
    // 直接アルファ値のみ変更することはできない
    var _color = spriteRenderer.color;
    _color.a = _alpha;
    spriteRenderer.color = _color;
    // スケーリング加算（実際減算だが
    _scale += scalePerSecond * Time.deltaTime;
    transform.localScale = _scale;
    // 終了判定
    if (_time > duration) { break; }
    yield return null;
}
```

チーム：日本ゲーム大賞アマチュア部門応募作品

タイトル　：　MELLOP（メルロップ）
開発環境　：　Unity 2019.3.0f6(64bit)
　　Microsoft Visual Studio2019
使用言語　：　C#
動作環境　：　Windows10
開発期間　：　４ヶ月（～2020 年 6 月
制作人数　：　プランナー　：２人
　　グラフィック：２人
　　プログラム　：４人　計８人
ジャンル　：　アクションパズル
作品概要　：
　「シュワちゃん」の下にあるブロックを溶かして進むパズル
　全体を回転させることで下に来るブロックを変えることができる
操作　　：　キーボード・XBOX コントローラー対応
担当範囲　：　Blocks フォルダ内の.cs ファイル全て
　　PuzzleManager.cs(ゲームルール管理スクリプト)
　　StageFileLoader.cs(ステージ情報読み込みスクリプト)

ステージファイルの読み込み

多数のステージを実装するため、csv ファイルからデータを読み込んでステージを自動生成しています。プランナーが調整した後、csv にしたものをビルドデータの指定フォルダ下に置くだけでテストプレイができ、レベルデザインを効率よく行うことができました。

今回はビルド後にファイルを手作業で追加していましたがダウンロードして自動で更新できるような機能など追加したいと考えています。

	A	B
1	Length	25
2	FriendBloc	1
3	NormalBlock	
4	NormalBlock	
5	FixedBlock	
6	NormalBlock	
7	NormalBlock	
8	NormalBlock	
9	FixedBlock	
10	NormalBlock	
11	NormalBlock	
12	FixedBlock	
13	GoalBlock	
14	AcidBlock