Portfolio

kcg.edu

Kyoto Computer Gakuin

喜多内稜士

自己紹介

名前	喜多内稜士

プログラマー志望

生年月日	2001年3月12日
学校・学科名	京都コンピュータ学院 ゲーム学科
取得資格	・ITパスポート ・情報技術検定1級 ・パソコン利用技術検定2級 ・計算技術検定2級
アルバイト歴	家電量販店で3年(継続中)
趣味	・ゲーム(FPS・サッカーゲーム) ・サッカー観戦(主に海外サッカー) ・アニメ視聴(バトル系・日常系)

賞歴

BITSUMMIT THE 8th BIT
ゲームジャム最優秀作品賞 受賞

使用ツールなど+熟練度(5段階)


5/5
Unity


4/5
UE4


4/5
Visual Studio


4/5
C++


4/5
C#

3/5
DirectX


4/5
Word


4/5
Power Point


3/5
Excel

自己紹介

プログラミングを始めたきっかけ

中学生の時からプログラミングがしたいと考えており、情報系列の高校を選択したのがきっかけです。高校ではC言語を主に学習して、卒業制作ではUnityを用いて2Dのアクションゲームを制作していました。

制作コンセプト

すべてのプログラムをする上で同じ処理をプログラムで書かないDRY原則を守る事や、オブジェクト指向プログラミングをする際は継承をうまく使うためにスーパークラスを用意してできるだけプログラムの記述量を減らすようにしています。
このようにする理由は後に改良や修正を行う時に更新部分を減らすためです。
また、DirectXでのプログラミングではスマートポインタやシングルトンパターンなどの新しめの技術や範囲for文などの特殊な処理を積極的に使っていくことで現場に着いた時に柔軟に対応できるように準備しています。

目次

重点的に学んできた技術

■Unity

イミュニティ

魔女の塔

その他の技術

■UE4

CORPSE DIVISION

■Direct2D

シューティングゲーム

Unity作品1

プレイ動画：https://bit.ly/2ZLb6Bp

イミュニティ

ジャンル：2Dアクションゲーム

制作時期：2021年8月
～
2021年9月

制作期間：1カ月

制作人数：9人

使用ツール

Unity　C#

ゲーム概要

負傷するたびに強くなる「ミュニ―」が
ジャンプすると回転する特性を駆使して
ウイルスに侵略された星を救うゲームです

作品成果

BITSUMMIT THE 8th BIT
ゲームジャム最優秀作品賞 受賞

TOKYO GAME SHOW 2021
学生スカラーシップ学生作品選出

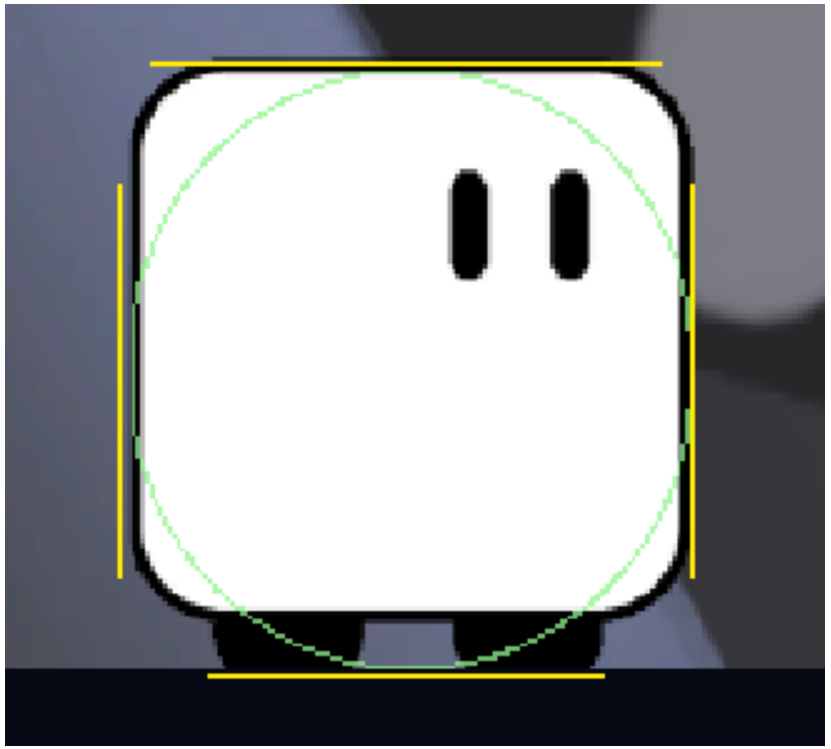
担当箇所

2種類の当たり判定処理

ギミックに対する当たり判定 — RayCast

×

移動に対する当たり判定 — Capsule Collider



このゲームでは通常の移動時の当たり判定と
ギミックに対する4面の当たり判定を切り分けて
キャラクターの当たり判定を実現しました。

担当箇所

Rayでの当たり判定作成

このゲームでは、四角形のキャラクターに
対して各辺別々に当たり判定を取得する
必要がありました。

ですが、Unityでの
コライダーコンポーネントでは細かな
当たり判定が行われた部分を取得できない
ので4辺に対してRayを用いたLineCast
をすることによって処理を実現しました。

LineCastにしたメリットとしては、
当たり判定をするレイヤーを
限定できることです。

今回はダメージ判定のある
ギミックのレイヤーに対してだけ
LineCastをするといった設定にしました。

```
void MakeRay()
{
    int RayCount = 4;

    hits = new RaycastHit2D[RayCount];

    //方向ベクトル
    Vector3[] Dire_Vec = { transform.right, //右
                           transform.up,    //上
                          -transform.right, //左
                          -transform.up};   //下

    StartPos = new Vector3[RayCount];

    EndPos = new Vector3[RayCount];

    StartPos[0] = this.transform.position + (Dire_Vec[1] * 1.6f + Dire_Vec[2] * RayDistanceTop);
    EndPos[0] = this.transform.position + (Dire_Vec[1] * 1.6f + Dire_Vec[0] * RayDistanceTop);

    StartPos[1] = this.transform.position + (Dire_Vec[0] * 1.45f + Dire_Vec[1] * RayDistanceLeftRight);
    EndPos[1] = this.transform.position + (Dire_Vec[0] * 1.45f + Dire_Vec[3] * RayDistanceLeftRight);

    StartPos[2] = this.transform.position + (Dire_Vec[2] * 1.45f + Dire_Vec[1] * RayDistanceLeftRight);
    EndPos[2] = this.transform.position + (Dire_Vec[2] * 1.45f + Dire_Vec[3] * RayDistanceLeftRight);

    StartPos[3] = this.transform.position + (Dire_Vec[3] * 1.5f + Dire_Vec[2] * RayDistanceBottom);
    EndPos[3] = this.transform.position + (Dire_Vec[3] * 1.5f + Dire_Vec[0] * RayDistanceBottom);

    for (int i = 0; i < RayCount; i++)
    {
        hits[i] = Physics2D.Linecast(StartPos[i], EndPos[i], Gimmick_Layer);
        Debug.DrawLine(StartPos[i], EndPos[i], Color.yellow);
    }
}
```

担当箇所

Rayでの当たり判定処理

Rayがギミックに当たった場合に、まずは当たったギミックの種類を判別します。

理由としては、当たった各ギミックに対してスプライトの表示と効果音の再生を分けているからです。

今回は毎if文に処理を書くのではなく、あらかじめ空で初期化してある変数に値を入れて後に出てくる1行で処理をするようにしました。

また、チーム制作ということもありデバッグ時にRayを可視化させるようにしてプランナーの方にも処理が明確にわかるようにをしました。

```
//Rayが当たった場合
if (hits[i])
{
    //デバッグ時にRayを表示
    Debug.DrawLine(StartPos[i], EndPos[i], Color.red);

    //Rayに当たったギミックがマグマの雫の場合
    if (hits[i].collider.gameObject.name == "DropLava(Clone)")
    {
        //マグマの雫を削除する
        Destroy(hits[i].collider.gameObject);
    }

    //Rayに当たったギミックが針の場合
    if (hits[i].collider.gameObject.tag == "Needle")
    {
        //変更スプライトを針に設定
        change_Sprite = NeedleSprite;
        //針ギミックに当たった時の効果音を設定
        DamageSound = "Damage_Needle";
    }
    //Rayに当たったギミックがマグマの場合
    else if (hits[i].collider.gameObject.tag == "Lava")
    {
        //変更スプライトをマグマに設定
        change_Sprite = LavaSprite;
        //マグマギミックに当たった時の効果音を設定
        DamageSound = "Damage_Lava";
    }
    //Rayに当たったギミックが氷柱か氷の地面の場合
    else if (hits[i].collider.gameObject.tag == "Ice" || hits[i].collider.gameObject.tag == "IceGround")
    {
        //変更スプライトを氷に設定
        change_Sprite = IceSprite;
        //氷ギミックに当たった時の効果音を設定
        DamageSound = "Damage_Ice";
    }
    //落下死してしまった場合
    else if (hits[i].collider.gameObject.tag == "Fall")
    {
        //効果音をnullに設定
        DamageSound = null;
        //死んだ時の処理をする関数を呼び出し
        Deth();
        //ギミックにヒットしたことを通知してチェックポイントに戻す
        PG.HitGimmick(hits[i].collider);
    }
```

担当箇所

Rayでの当たり判定処理

次はRayが当たった場所に応じて処理を
分けました。
今回は足元に当たった場合とそれ以外に
当たった場合に分けました。

足元に当たった場合はまず各免疫を消す
処理をするDethメソッドを呼び出します。
このメソッドでは免疫のスプライトと
当たり判定を消去します。

次に、死んだ際に一番近いチェックポイント
へワープさせる処理をします。

最後に、5ページで設定した効果音を
再生し死んだ回数をカウントします。

```
//足元に当たった場合
if (i == 3)
{
    //死んだ時の処理をする関数を呼び出し
    Deth();
    //ギミックにヒットしたことを通知してチェックポイントに戻す
    PG.HitGimmick(hits[i].collider);

    //ダメージサウンドがnullじゃない場合
    if (DamageSound != null)
    {
        //ギミックに対応した効果音を再生
        AudioManager.PlayAudio(DamageSound, false, false);
    }
    //死んだ回数をカウント
    Ending_Manager.AddDead_Count();

    //ループを出る
    break;
}
```

```
//死んだ時の処理をする関数
3 個の参照
void Deth()
{
    //慣性を消す
    UseInertia = 0.0f;

    //全免疫削除処理
    for (int i = 0; i < Cols.Length; i++)
    {
        //各面の免疫を透明化
        Cols[i].GetComponent<Renderer>().material.color = new Color(1, 1, 1, 0);

        //各面のスプライトをnullに設定
        Cols[i].GetComponent<SpriteRenderer>().sprite = null;

        //各面のスプライトをBoxCollider2Dを無効化
        Cols[i].GetComponent<BoxCollider2D>().enabled = false;
    }

    //各角度を0でリセット
    now_Rotate = rotateZ = 0f;
}
```

担当箇所

Rayでの当たり判定処理

次は足元以外に当たった場合の処理についてです。
まず、Rayが当たった部分の当たり判定を有効化します。

次に、免疫を持っていない場合は5ページで設定したスプライトを代入します。

そして、5ページで設定した効果音を再生して死んだ際に一番近いチェックポイントへワープさせる処理をします。

最後に、5ページで設定した効果音を再生し死んだ回数をカウントします。

```
//足元以外に当たった場合
else
{
    //当たった部分の免疫を表示
    Cols[i].GetComponent<Renderer>().material.color = new Color(1, 1, 1, 1);

    //当たった部分のBoxCollider2Dを有効化
    Cols[i].GetComponent<BoxCollider2D>().enabled = true;

    //ジャンプカウントを1に設定
    jumpCount = 1;

    //当たった面がギミック免疫を持っていないか場合
    if (Cols[i].GetComponent<SpriteRenderer>().sprite != change_Sprite)
    {
        //慣性を消す
        UseInertia = 0.0f;

        //ギミックに対応した免疫を付与
        Cols[i].GetComponent<SpriteRenderer>().sprite = change_Sprite;

        //ダメージサウンドがnullじゃない場合
        if (DamageSound != null)
        {
            //ギミックに対応した効果音を再生
            AudioManager.PlayAudio(DamageSound, false, false);
        }
        //ギミックにヒットしたことを通知してチェックポイントに戻す
        PG.HitGimmick(hits[i].collider);

        //死んだ回数をカウント
        Ending_Manager.AddDead_Count();

        //ジャンプカウントを2に設定
        jumpCount = 2;

        //ループを出る
        break;
    }
}
```

担当箇所

キャラクターの移動処理

このゲームではDキーとSキーで移動を可能にしており、押したキーの方向を取得してRigidBody2Dのポジションに加算するという方法を採用しました。
また、移動キーを離した際に急に移動スピードが0になることに違和感があったため疑似的な慣性処理を最後に記述しました。

そして、キャラクターの仕様上回転角度によって免疫のスプライトが埋まってしまうという問題点も改善しました。

```
//左シフトキーを押している場合
if (Input.GetKey(KeyCode.LeftShift))
{
    //初期スピード値にダッシュ倍率を乗算した値に設定する
    Speed = init_speed * dashPower;
}
//左シフトキーを押していない場合
else
{
    //スピードを初期スピードに設定
    Speed = init_speed;
}
//Dキーを押している場合
if (Input.GetKey(KeyCode.D))
{
    //移動方向変数をスピードに設定
    direction = Speed;

    //ジャンプカウントが2の場合
    if (jumpCount == 2)
    {
        //プレイヤーの左右反転
        rotateY = 0;

        //免疫が壁に埋もれないように修正
        for (int i = 0; i < Cols.Length; i++)
        {
            Cols[i].transform.localPosition = new Vector3(Cols[i].transform.localPosition.x,
                                                          Cols[i].transform.localPosition.y,
                                                          -1f);
        }
    }
}
```

```
//移動処理
rb.position += new Vector2(direction, 0.0f);

//慣性処理
direction *= UseInertia;
```

担当箇所

背景のスクロール処理

今回のゲームではメインカメラに
Cinemachineを採用しているので、
プレイヤーが移動した時に追従して移動
するカメラの移動量を取得します。
そして、その移動量に応じてシェーダーを
適応したマテリアルを回転させるといった
処理で実現しました。

今回は背景の種類が5種類あったので、
各背景でスクロール速度や背景の透過を
するかを変更できるようにしています。

```
fixed4 frag(v2f i) : SV_Target
{
//スピード計算
_XShift = _XShift * _XSpeed;

//スクロール
i.uv.x += _XShift / 10000;

//透明度処理
fixed4 col = tex2D(_MainTex, i.uv) * _Color;
col.a = _Alpha;

UNITY_APPLY_FOG(i.fogCoord, col);
return col;
}
```

UVでのシェーダ処理

```
//CinemaSceneカメラが動いてるか判定
if (0.05f < Mathf.Abs(CinemaCam.transform.localPosition.x))
{
    if (0 < CinemaCam.transform.localPosition.x)
        Speed++;
    else if (CinemaCam.transform.localPosition.x < 0)
        Speed--;
}

//背景のシェーダーの移動値を変更する
for (int i = 0; i < BackGroundCount; i++)
    BackGround[i].material.SetFloat("_XSpeed", ScrollSpeed * Speed);

//敵退出時に透明度を元に戻す
if (changeCamera.IfCameraFlag(ChangeCamera.CAMERAFLAG.MAIN) && Alpha < 1)
    Alpha += 0.01f;
//敵出現時に透明度を変更する
else if (!changeCamera.IfCameraFlag(ChangeCamera.CAMERAFLAG.MAIN) && 0 < Alpha)
    Alpha -= 0.1f;

//敵が出現するステージであればシェーダーに透明度を適応する
if (bChange)
    BackGround[1].material.SetFloat("_Alpha",Alpha);
```

C#でのシェーダ処理

Unreal Engine4作品

CORPSE DIVISION

ジャンル：3Dアクションゲーム

制作時期：2021年4月
～
2021年11月

制作期間：7カ月

制作人数：3人

使用ツール

UE4　C++

ゲーム概要

冥界に迷い込んでしまった骸骨人間の
「ジョンソン」が、頭を投げて敵に乗り
移ったりして冥界を脱出するゲームです。

担当箇所

Unreal C++での作成

今回の作品ではプログラミングスキルの向上を目指して、
ブループリントでの処理を極力少なくするようにカメラの処理とUIの処理以外
ほぼ全てC++で処理を実装しました。

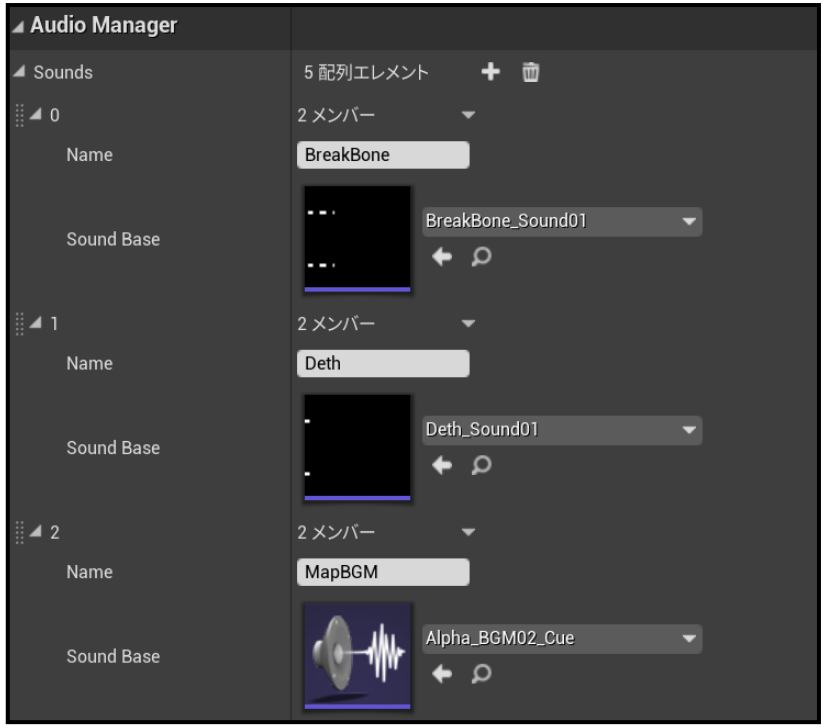
下の図は、キャラクターが別のキャラクターの体に乗っ取る処理です。
Unityとは違い、ポインターの処理やDirectXに近い処理があったため
技術向上に繋がりました。

```
// 投げる処理
void ACreature::ThrowAfter(AHead_Pawn* _head)
{
    //ポーズ画面を開いていない　かつ　頭を投げる構えをしていない場合
    if (head->bPose || !bThrow)
        return;

    //頭の重力処理をtureにする
    head->CollisionComponent->SetSimulatePhysics(true);
    //頭が慣性を受けないように速度を0でリセット
    head->CollisionComponent->SetPhysicsLinearVelocity(FVector::ZeroVector);
    //頭を飛ばす処理(頭に速度を与える)
    head->CollisionComponent->SetPhysicsLinearVelocity(this->GetActorForwardVector() * power + FVector(0.f, 0.f, zPower), true);

    //投げ終わったら、頭を投げる構えのboolをfalseにする
    bThrow = false;

}
```

担当箇所

オーディオマネージャーの作成

右上の画像のようにUE4のエディター上で
オーディオ名とオーディオコンポーネントを設定し
ワールドに配置することで使えるようになります。
そして、プログラムではstaticメソッドとして作成し
ているのでどのクラスからでも呼び出せてオーディオ
を再生できます。
メソッドの仕組みとしては、
引数にオーディオ名を渡すだけでワールド上の
オーディオマネージャーを検索しオーディオ名に
対応するオーディオコンポーネントを再生します。



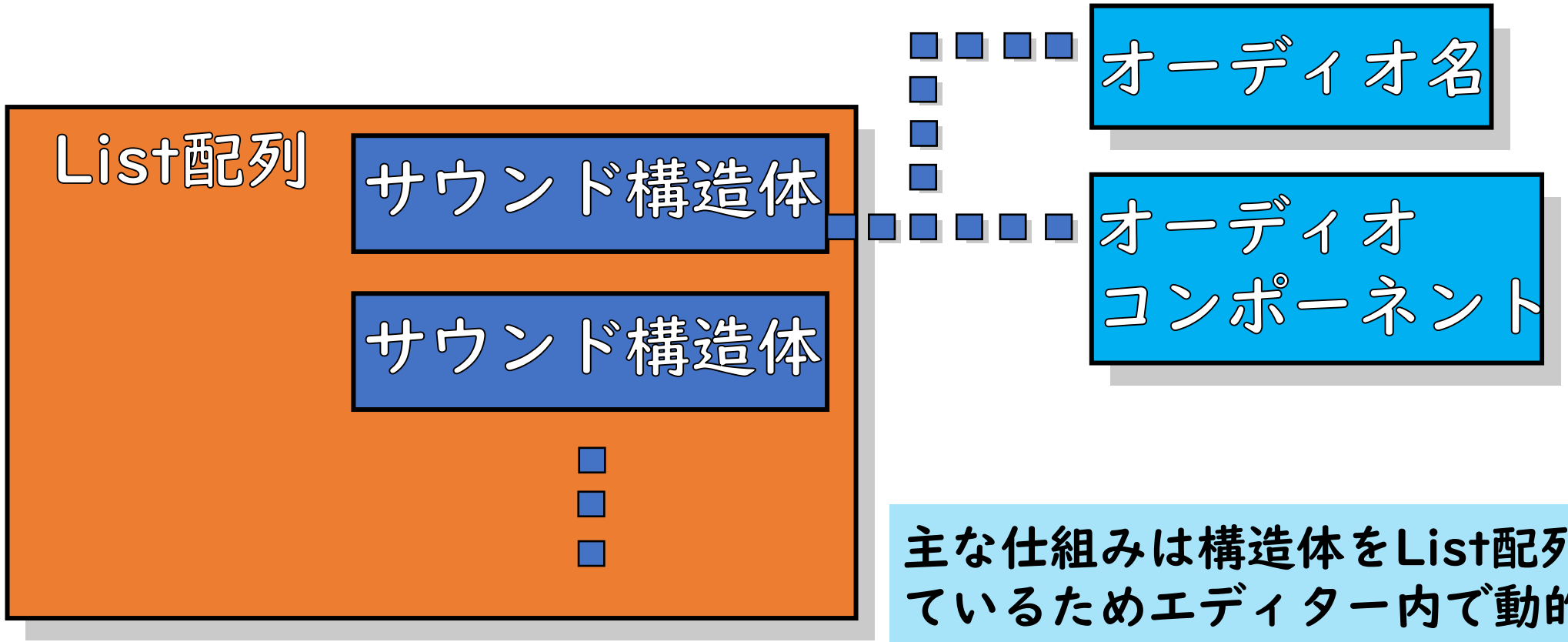
エディター画面

```
Manager::PlaySound2D(GetWorld(), "BreakBone");
```

プログラム画面

担当箇所

オーディオマネージャーのしくみ



主な仕組みは構造体をList配列に格納しているためエディター内で動的に
配列の要素を増やしていけるのが
このオーディオマネージャーの利点です。

担当箇所

オーディオマネージャーの作成(コード)

```
//サウンド再生関数
void K_Manager::PlaySound2D(const UObject* worldContext, FString soundName, float volume, float pitch, float startTime)
{
    //ワールド情報取得
    UWorld* world = GEngine->GetWorldFromContextObject(worldContext, EGetWorldErrorMode::LogAndReturnNull);

    //オーディオマネージャーオブジェクト取得先を定義
    TArray<AActor*> findObj;

    //オーディオマネージャーオブジェクト検索取得
    UGameplayStatics::GetAllActorsOfClass(world, AAudioManager::StaticClass(), findObj);

    //オーディオマネージャーが空なら返す。
    if (findObj[0] == nullptr)
        return;

    //空でなければアクタークラスからオーディオマネージャクラスにキャスト
    auto AM = Cast<AAudioManager>(findObj[0]);
    //オーディオマネージャーの配列をループで回す
    for (auto Sound : AM->Sounds)
    {
        //引数で受け取ったサウンド名と合致したら
        if (Sound.name == soundName)
        {
            //サウンドを生成。
            AM->audioComponents.Add(UGameplayStatics::SpawnSound2D(world, Sound.soundBase, volume, pitch, startTime));

            //サウンドを再生。
            AM->audioComponents[AM->audioComponents.Num() - 1]->Play();
        }
    }

}
```

Unity作品2

プレイ動画：https://bit.ly/3mTBKkl

魔女の塔

ジャンル：
3Dアクションゲーム

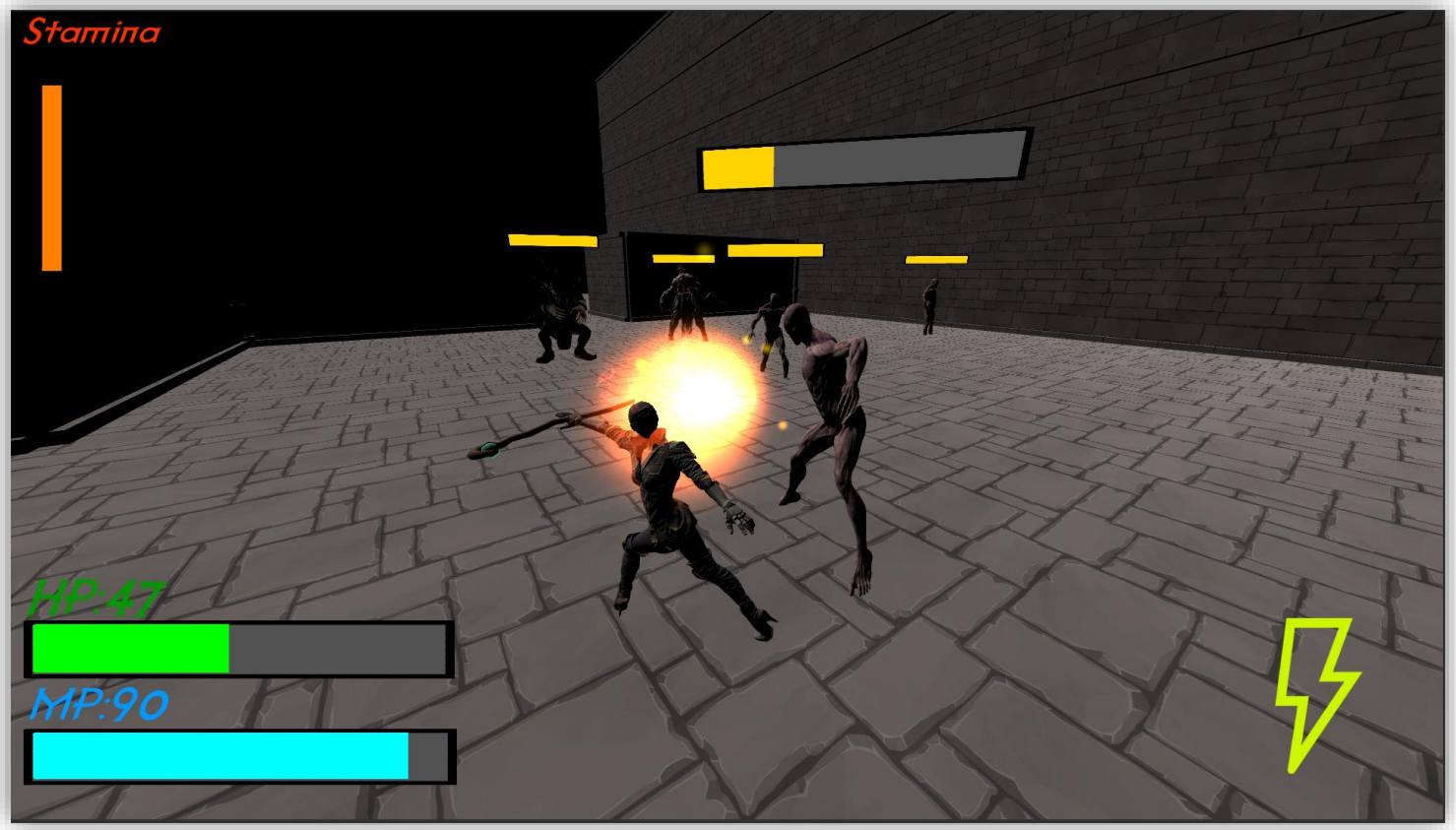
制作時期：2020年10月
～
2021年1月

制作期間：4カ月

制作人数：2人

使用ツール

Unity　C#　Mixamo



ゲーム概要

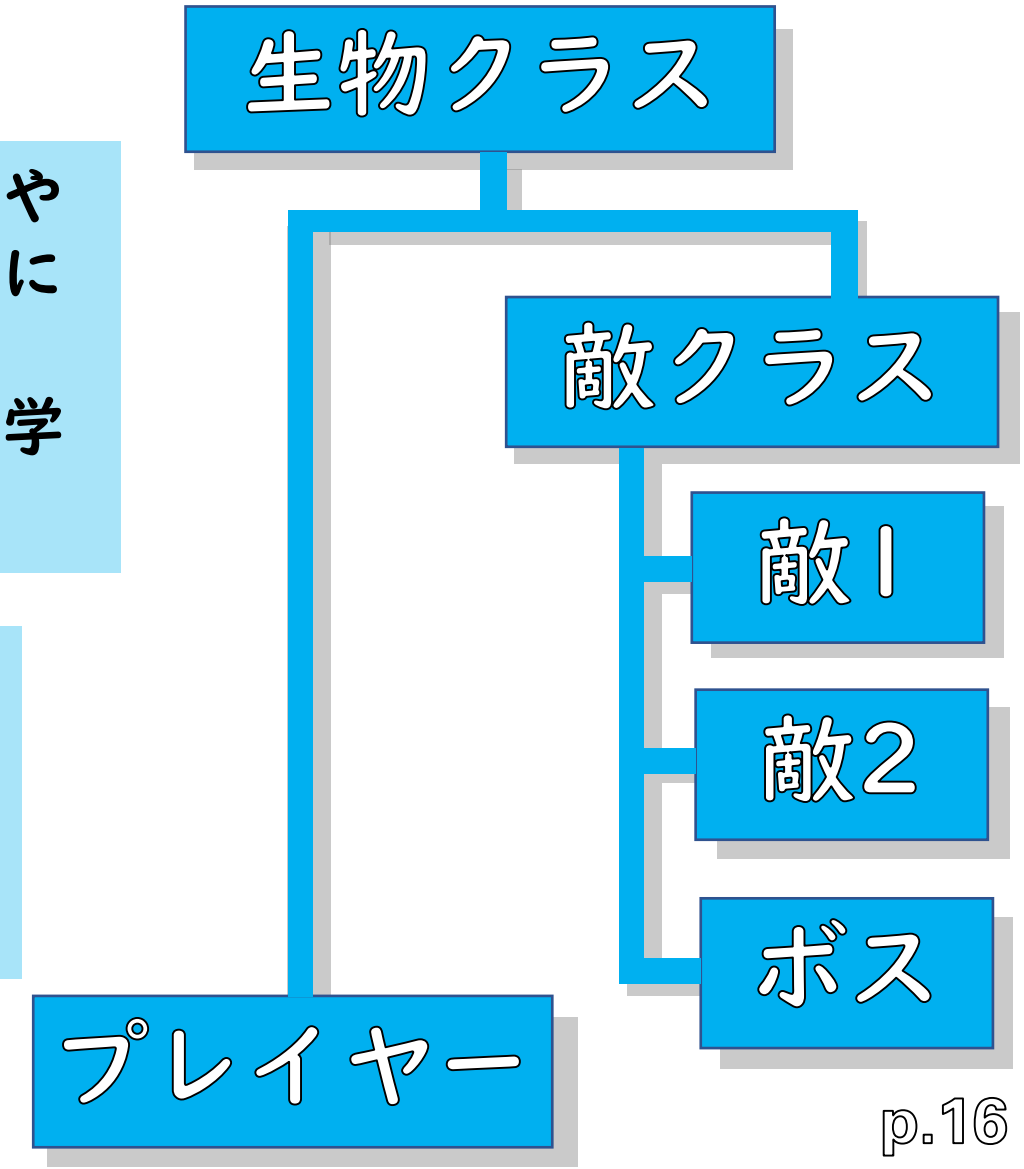
剣と魔法を使って敵を倒していき、
塔の頂上を目指していくゲームです。

担当箇所

クラス継承

この作品ではクラス継承を意識して各クラスの変数や関数の記述量を減らしてプログラミングをすることによって
オブジェクト指向プログラミングの概念を重点的に学ぶことができました。

今回は右のように生物クラスを基準にクラス継承をすることによって
スクリプトの修正をする手間を最小限に抑えることができました。



担当箇所

親クラスの例(生物クラス)

右のプログラムコードのように、
どのキャラクターでも使うようなHPや
アニメーション、生きているかどうかのbool
を変数で、HPをセットする、
オブジェクトを消す、ダメージを受ける
といった処理を関数で持たせました。

そして、
HPの設定や死んでしまった時の処理、
ダメージ処理などを子クラスで
呼び出せるようにprotectedで宣言しました。

```
public class Creatures : Basic
{
    protected float hp = 0;          //プレイヤー・各敵のHP
    protected float Max_hp;          //プレイヤー・各敵のHPの最大値

    protected Animator animator;     //プレイヤー・各敵のアニメーター取得用
    protected bool deth = false;     //生死を判別するbool

    //HPセッター(初期化時に受け取る)
    4 個の参照
    protected void Set_HP(float set_hp)
    {
        hp = Max_hp = set_hp;
    }

    //HPが0になったらオブジェクトを消すメソッド
    3 個の参照
    protected void Destroy_Object()
    {
            Destroy(gameObject);
    }

    //ダメージを受けたらhpを減らすメソッド
    8 個の参照
    public void Damage_Object(float damage)
    {
        if(!deth)
            hp -= damage;
    }
}
```

担当箇所

コルーチン関数

基本的に普通の関数と処理は同じですが
大きく違う点は特定の戻り値を選択することによって処理待ちができることです。

今回の場合は、WaitForSeconds(0.7f)というように、
()内の秒数の間処理を待って次の処理を実行するうという挙動が実現できました。

```
//ダークボール攻撃処理
0 個の参照
IEnumerator Darkball_Attack()
{
    //アニメーションと合わせてオブジェクトを出すために7秒間待機
    yield return new WaitForSeconds(0.7f);
    //ダークボールオブジェクトを生成
    GameObject Darkball_object = Instantiate(Darkball,                    //ダークボールのプレハブ
                                             this.Position,               //生成するキャラクターのポジション
                                             zombie_transform.rotation); //生成するキャラクターの向き
    //3病後に生成したオブジェクトを削除する
    Destroy(Darkball_object, 3.0f);
}
```

担当箇所

使用武器の判別

このコードでは、落ちてる武器に当たったら武器リストに追加するといった処理をしています。

武器リストと持っている武器を判別するためにforeach文で要素を1つずつ取り出して、その名前が同じであるものだけをアクティブにすることで処理を実現できました。

```
//当たり判定処理(トリガー)
Unity メッセージ|0 個の参照
void OnTriggerEnter(Collider collider)
{
    //武器タグが設定された武器オブジェクトに当たった場合
    if (collider.gameObject.tag == "Weapon")
    {
        //当たった武器オブジェクトのポジションを0に設定
        collider.gameObject.transform.position = Vector3.zero;
        //当たった武器オブジェクトをプレイヤーの右手の子オブジェクトに設定
        GameObject WeaponObject = Instantiate(collider.gameObject,                                    //当たった武器オブジェクト
                                              GameObject.Find("RightHand_P").transform.GetChild(5));//持たせる右手の子オブジェクト
        //武器リストに武器オブジェクトを追加
        weapons.Add(WeaponObject);
        //武器リストから今当たった武器オブジェクトを優先的に有効化する
        foreach (var weapon in weapons)
        {
            //当たった武器オブジェクトと名前が一致した場合
            if (weapon.name == WeaponObject.name)
            {
                //オブジェクトを有効化
                weapon.SetActive(true);
            }
            //その他の場合
            else
            {
                //オブジェクトを無効化
                weapon.SetActive(false);
            }
        }
        //配置されていた武器オブジェクトを削除
        Destroy(collider.gameObject);

    }
```

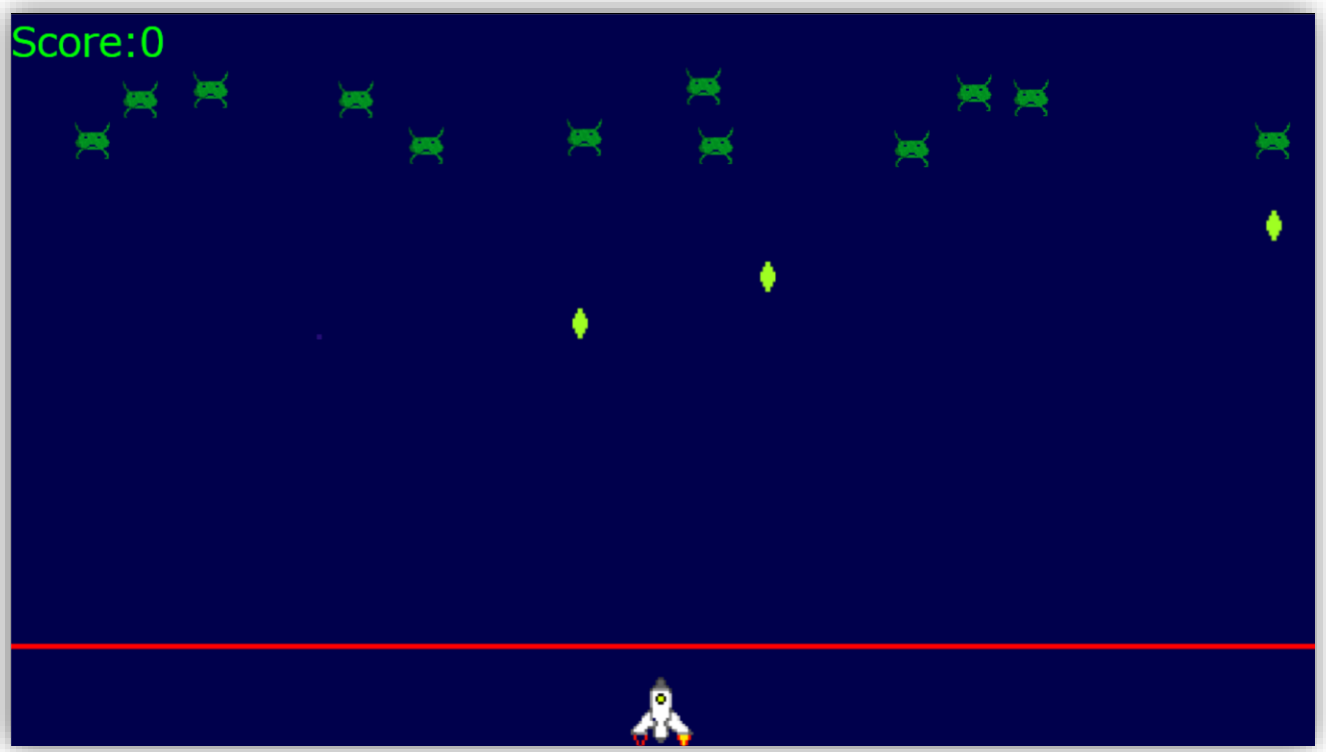
Direct2D作品

シューティングゲーム

ジャンル：
2Dシューティングゲーム

制作時期：2021年6月
～
2021年7月

制作人数：1人



C++ DirectX Piskel

制作概要

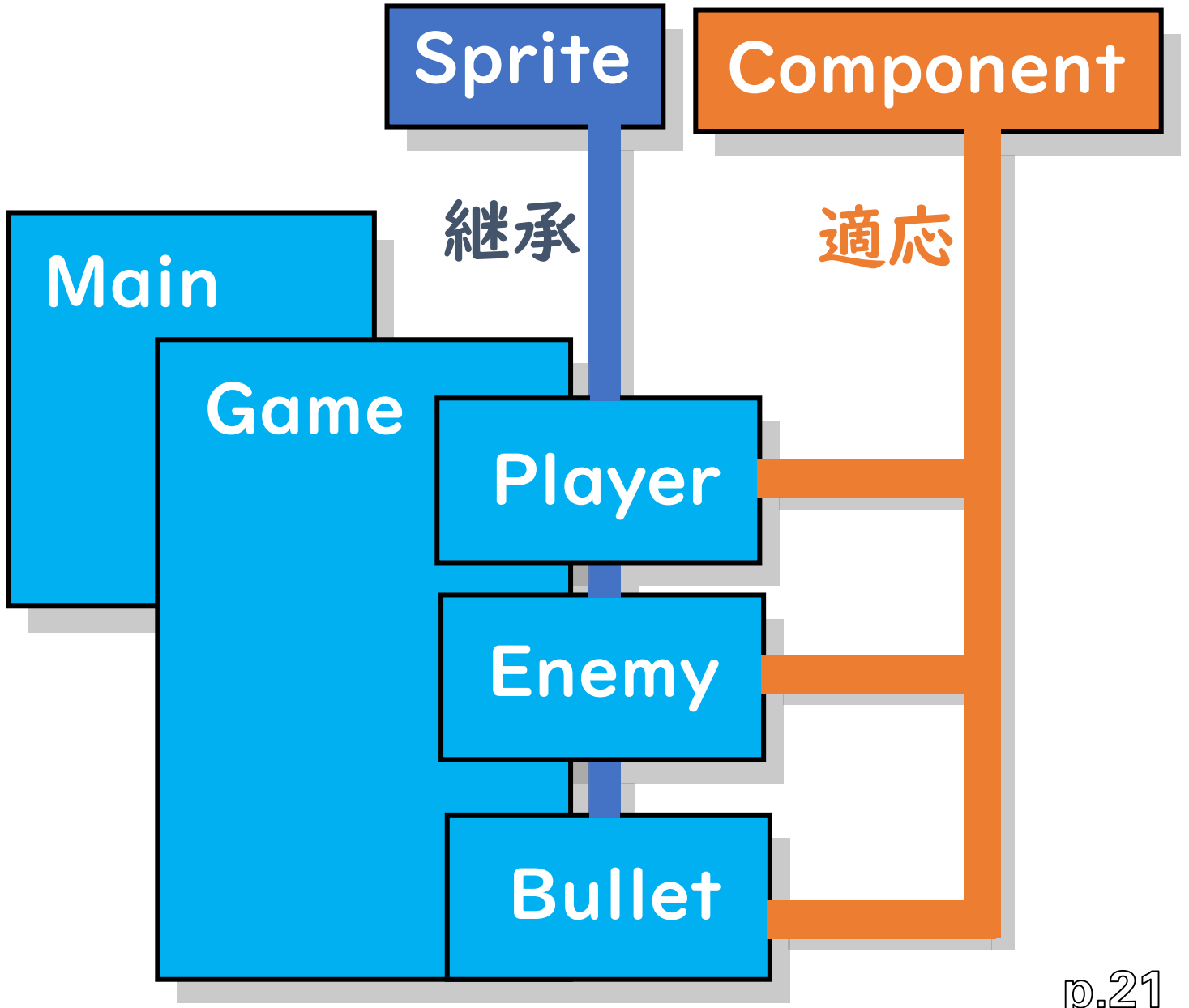
Direct2Dの基礎を1から学ぶために
この作品を制作しました。

担当箇所

クラス設計

SpriteとComponentという
基盤を作った上で各オブジェクト
に適応するといった方法で
実装しました。

基盤として作った理由は、
どのゲームを作る時にも
使用できるような
汎用型プログラミングをすること
を心がけているからです。



担当箇所

スマートポインタの使用

この作品では通常のポインタではなくスマートポインタを使用しました。

スマートポインタを使用することで、確保したメモリの解放を忘れたときの「メモリーリーク」が起きなくてすみます。
また、解放する場所などを気にしなくて良くなるため開発速度にも良い影響を与えてくれるため採用しました。

```
private:
    //システムクラスのポインター
    System* m_sys;
    //プレイヤークラスのポインター
    std::shared_ptr<Player> m_player;
    //弾幕コントロールクラスのポインター
    std::shared_ptr<BulletController> m_bulletController;
    //敵クラスのポインター
    std::shared_ptr<Enemy> m_enemy;
    //敵コントロールクラスのポインター
    std::shared_ptr<EnemyController> m_enemyController;
    //当たり判定コントロールクラスのポインター
    std::shared_ptr<CollisionManager> m_collisionManager;

    //プレイヤー画像のポインター
    Microsoft::WRL::ComPtr<ID2D1Bitmap> m_Playerbitmap;
    //弾幕画像のポインター
    Microsoft::WRL::ComPtr<ID2D1Bitmap> m_Bulletbitmap;
    //敵画像のポインター
    Microsoft::WRL::ComPtr<ID2D1Bitmap> m_EnemyBitmap;

    //スプライトクラス(生成されたオブジェクトクラス)のリスト
    std::list<std::shared_ptr<Sprite>> m_sprites;
```

スマートポインタの変数定義

```
//プレイヤークラスのインスタンスを生成
m_player = std::make_shared<Player>(m_sys, this, renderTargetSizeCenter, L"Player" );
//スプライトリストに追加
m_sprites.push_back(m_player);

//弾幕コントロールクラスのインスタンスを生成
m_bulletController = std::make_shared<BulletController>(m_sys, this);
//スプライトリストに追加
m_sprites.push_back(m_bulletController);

//敵コントロールクラスのインスタンスを生成
m_enemyController = std::make_shared<EnemyController>(m_sys, this);
//スプライトリストに追加
m_sprites.push_back(m_enemyController);
```

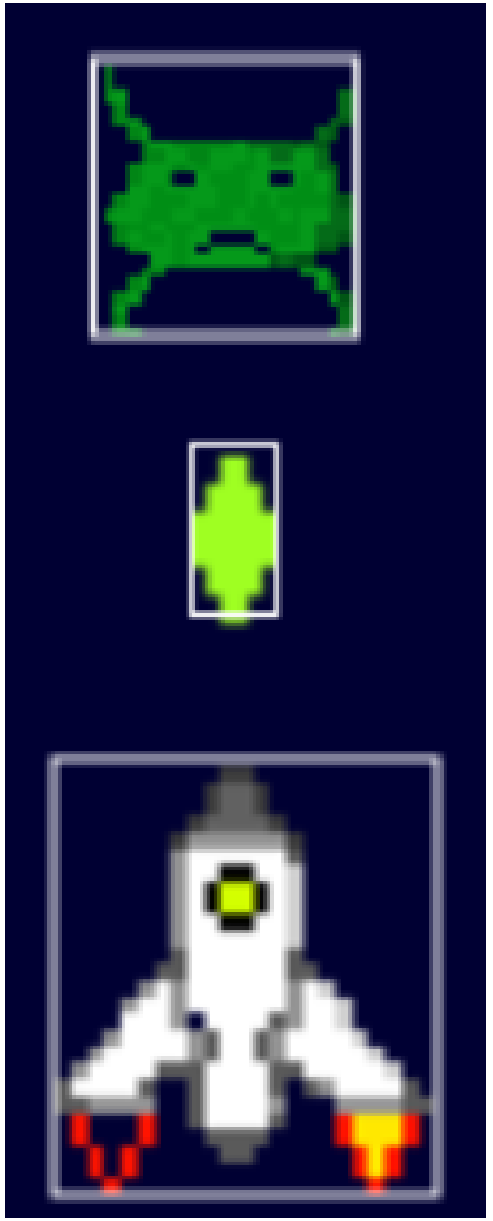
スマートポインタのインスタンス生成

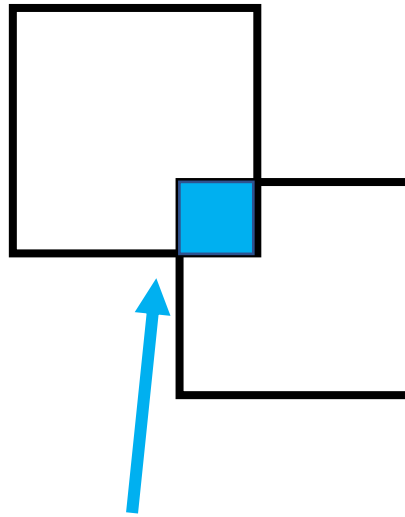
担当箇所

当たり判定処理の作成

各オブジェクトに当たり判定用の四角形の
領域を設定して、
その四角形の領域どうしが重なった時に
trueを返すといった処理をすることで
当たり判定の実装を実現しました。

イメージが湧かないといった人はUnityの
BoxCollider2Dコンポーネントと
同じ処理を自作で実装したと
思ってもらえれば大丈夫です。





担当箇所

当たり判定処理の作成(コード)

```
//当たり判定処理
//引数1:当たり判定対象のオブジェクト
//引数2:当たり判定をするタグ
bool CollisionManager::ChackCollisionBool(Picture* pic, std::wstring tag) {

    //当たり判定処理をするオブジェクトリストのイテレーターをfor文で回す
    for (auto i = m_ColList.begin(); i != m_ColList.end();)
    {
        //イテレーターの情報を取得
        auto col(*i);
        //設定したタグと同じ　かつ　お互いの当たり判定が重なっている場合
        if (tag == col->m_tag &&                            //タグの判定
            col->m_rect.left <= pic->m_rect.right &&        //当たり判定を行うオブジェクト同士の左端・右端の座標値を比較
            col->m_rect.right >= pic->m_rect.left &&        //当たり判定を行うオブジェクト同士の右端・左端の座標値を比較
            col->m_rect.top <= pic->m_rect.bottom &&        //当たり判定を行うオブジェクト同士の上端・下端の座標値を比較
            col->m_rect.bottom >= pic->m_rect.top)          //当たり判定を行うオブジェクト同士の下端・上端の座標値を比較
        {
            //当たったならtrueを返す
            return true;
        }
        i++;
    }

    //通常時はfalseを返す
    return false;
}
```

担当箇所

キーボード入力処理

まずMove関数で常に
キーボード入力状況を取得、
1つ前に入力されたキーを
記憶する処理をしています。

そして、指定したキーコードが
入力されたキーと同じかを
判別してキーボード入力の
処理を実現しました。
更に、キー入力がワンタップか
ホールドを判別する処理も加え
ました。

```
void System::Move() {
    //今押しているキー入力情報をコピーする
    CopyMemory(KeyPrev, Key, sizeof(Key));
    //仮想キーの状態を、バッファへコピーコピーする
    GetKeyboardState(Key);
}
//キーボード入力をチェックする
//引数1:入力を確認したキー
//引数2:長押しを検知するか(trueなら検知、falseなら1回のみ入力を確認)
bool System::KeyCheck(const int KeyCode, bool hold) {

    //キーの長押しを検知する場合
    if (hold) {
        //キー入力を確認した結果を戻す
        return (Key[KeyCode] & 0x80) != 0;
    }
    //キーの1回押しを検知する場合
    else {
        //キー入力を確認した結果を戻す
        return (Key[KeyCode] & 0x80) != 0 && (KeyPrev[KeyCode] & 0x80) == 0;
    }
}
```

最後まで見ていただき
本当にありがとうございました。

喜多内稜士