

Unity を使ったゲームの作成

7 番 菊池 愛斗

15 番 菅原 遥翔

19 番 照井 太一

20 番 元良 優作

指導教員 小野陽子

目次

| | |
|-----------------------------------|----|
| 1 はじめに | 4 |
| 2 研究概要 | 5 |
| 2.1 開発環境 | 5 |
| 2.2 開発ツールの説明 | 5 |
| 2.3 ゲームの仕様 | 6 |
| 3 Unity 概要 | 7 |
| 3.1 Unity の歴史 | 7 |
| 3.2 Unity の環境設定 | 8 |
| 3.3 プロジェクトの作成 | 12 |
| 3.4 ゲームの作成 | 13 |
| 3.5 Unity でよく使われるコンポーネントの説明 | 14 |
| 3.6 外部アセットとその使い方 | 17 |
| 4 ゲーム紹介 | 20 |
| 4.1 タンクゲーム | 20 |
| 4.2 カードゲーム | 31 |

| | |
|----------------------------|-----|
| 4.3 シューティングゲーム | 64 |
| 4.4 反射神経ゲーム | 81 |
| 4.5 テトリス | 86 |
| 4.6 横スクロールアクションゲーム 1 | 102 |
| 4.7 横スクロールアクションゲーム 2 | 124 |
| 4.8 メニュー画面 | 134 |
| 5 おわりに | 140 |
| 6 参考文献 | 141 |

1 はじめに

今回卒業研究に取り組む際、私たちのグループでは以下の二つができる研究に取り組みたいと考えた。

1つ目は身近にあるプログラミング技術が使われているものを作成できる研究である。今まで業務システム中心の作成が多かったため、身近にあるアプリやゲームなどといったものを作成したいと考えた。

2つ目は今まで本校で学んだ様々なプログラミング技術の復習と活用ができ、今後必要となるような知識も学べるような研究である。

以上の理由から私たちのグループはゲームの作成に取り組もうと考えた。またゲームの作成では以下の2つができる。

1つ目は本格的にグループで活動できることである。今まで個人での開発が多く、複数人での開発というものをあまりしてこなかった。ゲームを作成するにはゲームの仕様、登場キャラクター、背景、バランス調整など様々な点でコミュニケーションを取り合い、役割分担をし、完成を目指すことができる。このことからグループとしての開発技術も高めることができると考えた。

2つ目は本研究で作成したゲームを産技短展や文化祭にお越しになった方々に遊んでもらうことである。遊んでもらうことで中学生や高校生には産業技術短期大学校に興味を持つ1つのきっかけになり、親御さんや小さなお子さんにはパソコンに親しんでもらったりすることができると考えた。

以上の理由から私たちのグループでは Unity を使ったゲームの作成に取り組むことにした。

2 研究概要

2.1 開発環境

本研究の開発環境は以下の通りである。

表 2.1.1 開発環境

| | |
|------|--------------|
| OS | Windows10 |
| ツール | Unity、Photon |
| 使用言語 | C# |

2.2 開発ツールの説明

2.2.1 Unity

Unity Technologies が有償、無償で提供しているゲームエンジンである。2D ゲーム、3D ゲームの開発に対応しており、ウェブプラグイン、デスクトッププラットホーム、家庭用ゲーム機、携帯ゲーム機、モバイル向けのゲーム開発をするために用いられる。拡張機能がある。Asset Store を用いることで効率の高い開発を行うことができる。詳細は 3 Unity 概要に記述している。

2.2.2 Photon

Photon とは、ネットワークエンジンであり、マルチプレイでの同期を担うソリューションとマルチプレイでのコミュニケーションを補助するソリューションの大きく 2 つを提供する。サーバーとクライアント SDK がセットで提供されており、ルームやロビー機能を含め、マルチプレイを実現するために必要な機能を一通り搭載しているため、ネットワークの実装を簡単に進められる。Unity への導入方法は、4 ゲーム紹介の 4.2 カードゲームのオンライン設計に記述している。

2.2.3 C#

C#とは、マイクロソフトが開発しているプログラミング言語であり、C++やJavaといった言語と同じオブジェクト指向言語である。WindowsやXML Webサービス、サーバーアプリケーション、データベースアプリケーションなどの様々なアプリケーションを作成することが出来る。文法はJavaに似ているところがある。

標準の開発・実行環境は、統合開発環境の「Visual Studio」の一部として「Visual C#」として提供している。開発したプログラムは、NET環境に共通する中間言語CILによるバイナリコードに変換される。UnityのソースコードはC#、JavaScript、Booという言語が使えるが、C#を使うのが最も一般的である。

2.3 ゲームの仕様

7つのゲームとそれを遊ぶためのメニュー画面を作成した。ゲームはルールが簡単で誰もが楽しめるゲームを意識して作成した。

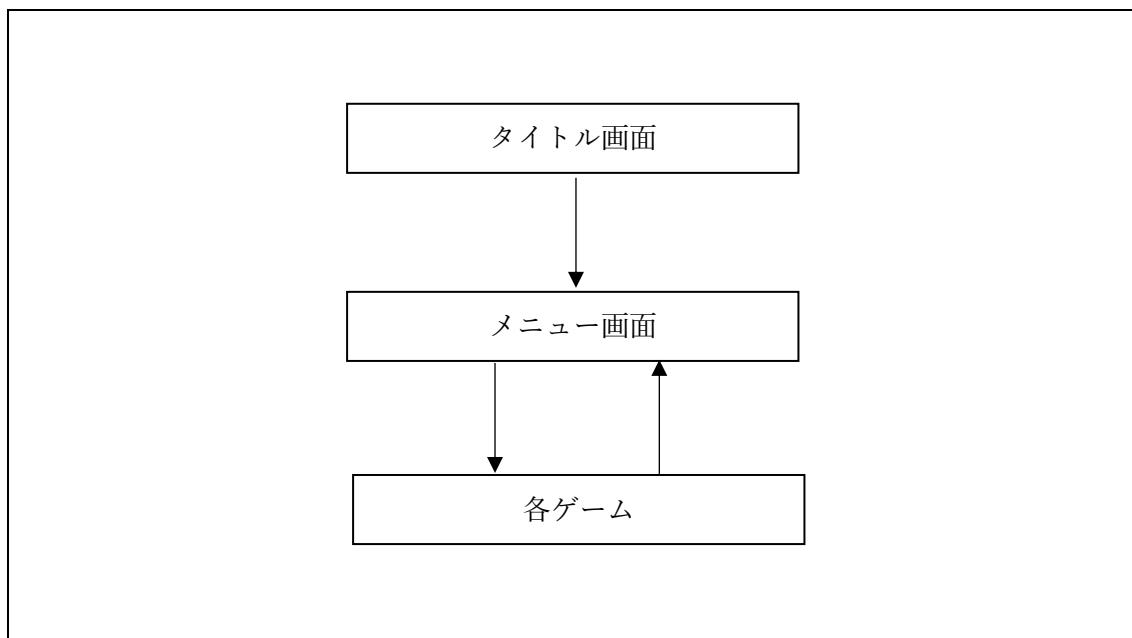


図 2.3.1 ゲームの仕様図

3 Unity 概要

3.1 Unity の歴史

Unity の年表を以下の表に示す。

表 3.1.1 Unity 年表

| | |
|--------|---|
| 2004 年 | デンマーク・コペンハーゲンで Joachim Ante (ヨアキム・アンテ)氏、Nicholas Francis (ニコラス・フランシス)氏、David Helgason (デイビット・ヘルガソン)氏の 3 人によって創業された。 |
| 2005 年 | Over the Edge Entertainment (2007 年に Unity Technologies に改名) が公開したゲーム開発ツール。創業者たちは、当時流行していたゲームエンジンよりも、「オープンでみんなが使用できるゲーム開発環境」を目指した。公開した当初から、ほとんどの機能を無料で利用でき、ゲーム開発のプロフェッショナルのみならず、初心者ユーザー同士が気楽にコミュニケーションできる世界が作られた。 |
| 2006 年 | Unity はマルチプラットフォームに対応しており、Windows だけでなく Mac 上でも動作し、iPhone が普及するとともに驚異的なスピードで普及した。他にも、家庭用ゲーム機である PlayStation や Nintendo Switch など 20 以上のプラットフォームに対応している。 |

現在は、新規モバイルゲームの 50% は Unity で作られており、全世界モバイルゲーム売上位 1000 タイトルのうち 71% が Unity 製であるといわれている。2021 年には、2020 年に比べて Unity クリエイターの数が 31%、Unity で作られたゲームの数は 93% 増加した。

Unity を使ったコンテンツの月間アクティブユーザー数は 28 億人を超える。その大きな理由は、個人で無料*で始められるところである。ゲーム売上額に応じたロイヤリティも取っていない。

*過去 12 か月の収益や調達資金が 10 万米ドル以上の場合は Unity Plus(月額 4,400 円)、または Unity Pro(月額 16,500 円)の契約をする必要がある。

3.2 Unity の環境設定

3.2.1 Unity ID の作成

メールアドレス、パスワード、ユーザー名、フルネームを設定して、設定したメールアドレス宛に確認メールが届くので、記載されている手順に従うと、Unity ID の作成は完了する。



図 3.2.1.1 Unity ID の作成画面 1

A screenshot of the 'Unity ID を作成' (Create Unity ID) form. The form has two main sections: 'メールアドレス' (Email Address) and 'パスワード' (Password). Both sections have input fields. Below these are 'ユーザー名' (Username) and 'フルネーム' (Full Name) input fields. At the bottom of the form, there are several checkboxes for accepting terms and conditions, a CAPTCHA field, and a '私はロボットではありません' (I am not a robot) checkbox. There are two buttons at the bottom: 'Unity ID を作成' (Create Unity ID) and 'Unity ID でログイン' (Log in with Unity ID). Below the buttons is a 'または' (Or) section with social media icons for Google, Facebook, and Apple.

図 3.2.1.2 Unity ID の作成画面 2

3.2.2 Unity Hub のインストール

Unity 公式サイトからダウンロードする。



Unity を使って 3 つのステップで作成



図 3.2.2.1 Unity 公式サイト 引用元：<https://unity.com/ja/download>

「UnityHubSetup.exe」を起動して Unity Hub をインストールする。

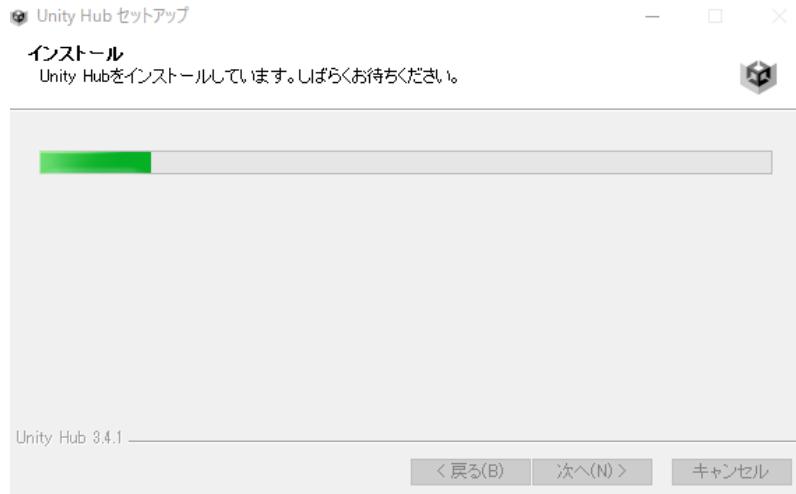


図 3.2.2.2 Unity Hub インストール画面

3.2.3 ライセンスの取得

Unity Hub を起動してライセンスを取得する。

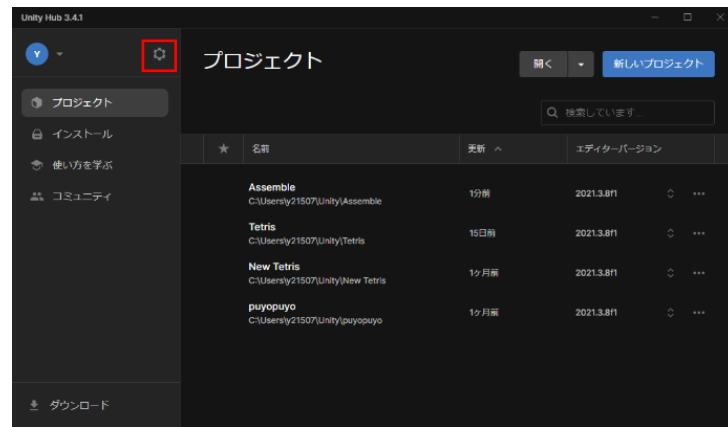


図 3.2.3.1 ライセンスの取得 1

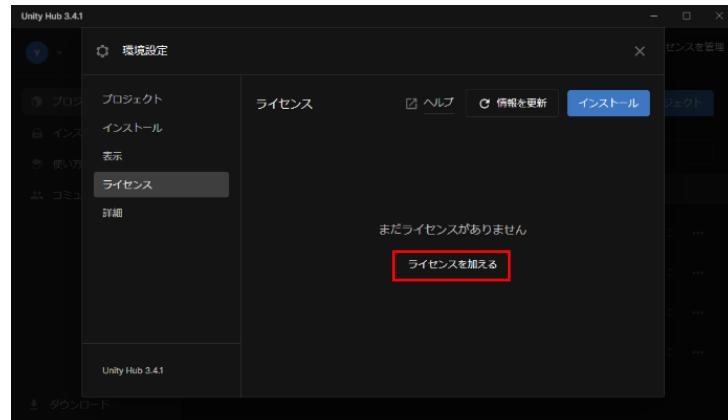


図 3.2.3.2 ライセンスの取得 2

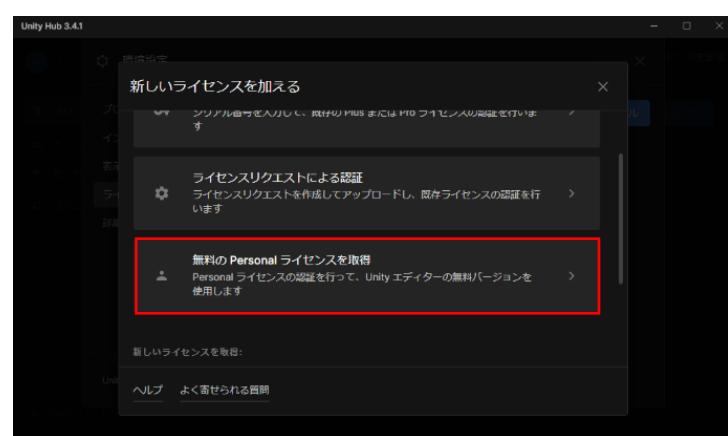


図 3.2.3.3 ライセンスの取得 3

3.2.4 起動用バッチファイルの作成

起動用バッチファイルを作成し、Unity Hub を起動する。

```
@echo off  
set HTTP_PROXY=http://172.16.0.2:8080  
set HTTPS_PROXY=http://172.16.0.2:8080  
start "" "C:\Program Files\Unity Hub\Unity Hub.exe"
```

コード 3.2.4.1 unity-hub-proxy.bat

このバッチファイルでプロキシを回避し、新規インストール、モジュールの追加、サインインなどが行える。また Unity Hub から起動した Unity Editor では、Asset Store からダウンロードが可能になる。

3.2.5 Windows Defender ファイアウォールのブロックの解除

Unity Hub や Unity Editor の初回起動に Windows Defender から警告が出る可能性がある。

プライベートネットワークにチェックを入れてアクセスを許可する。

3.3 プロジェクトの作成

Unityを起動すると、次の画面が表示される。この画面で、「新しいプロジェクト」をクリックする。

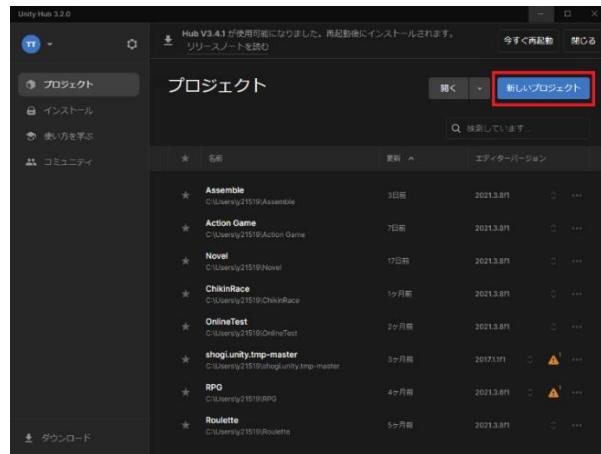


図 3.3.1 新しいオブジェクトの作成 1

作るゲームによって、「2D」または「3D」を選択、プロジェクト名、保存場所を書き、「プロジェクト作成」をクリックする。

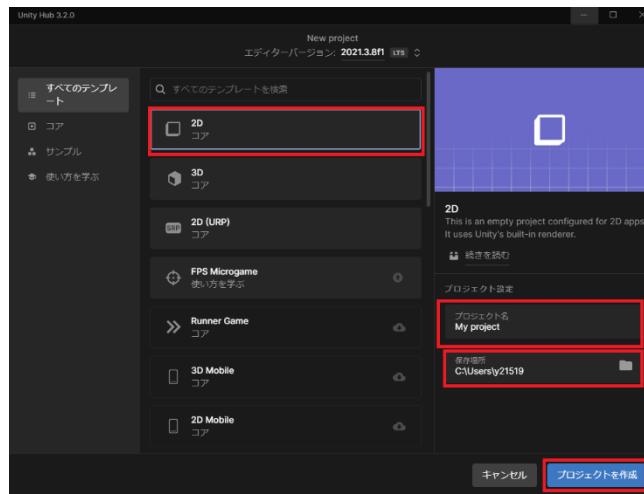


図 3.3.2 新しいオブジェクトの作成 2

3.4 ゲームの作成

「プロジェクトの作成」をクリックすると、Unity の制作画面に移る。主に下記のようなウィンドウでゲーム制作を行う。

① シーンビュー・ゲームビュー

シーンビューはオブジェクトを設置し、ゲームを作っていく画面である。

ゲームビューでは実際にゲームを動かしたときの挙動などを確認できる。

② ヒエラルキー・オブジェクト

ゲーム内オブジェクトの一覧、階層段階を確認できる。

③ インスペクターウィンドウ

ゲーム内オブジェクトのステータス画面で、オブジェクトの動作などを管理している。

④ プロジェクトウィンドウ・コンソール

プロジェクトウィンドウは、ゲーム内で使う絵、音楽、スクリプトなどの素材を置いておく場所であり、コンソールウィンドウはエラー・ログなどを表示する場所である。

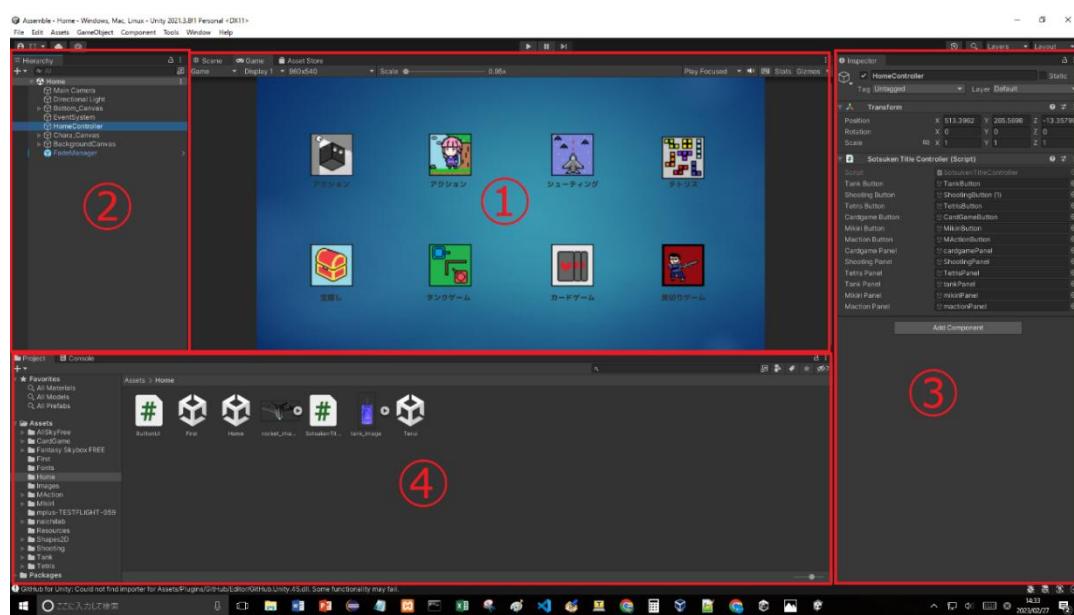


図 3.4.1 ゲーム制作画面

3.5 Unity でよく使われるコンポーネントの説明

Unity では、C#のスクリプトや Box Collider などのコンポーネントをオブジェクトにアタッチすることで、そのオブジェクトの動きや性質をプログラマムすることができる。Unity には、便利なコンポーネントがとても多く、オブジェクトのインスペクタウインドウの「Add Component」ボタンを押すことで、使いたいコンポーネントを簡単にアタッチすることができる。

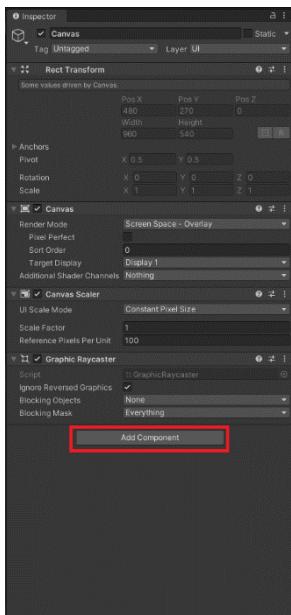


図 3.5.1 Add Component ボタン

3.5.1 C#スクリプト

C#は、Unity で使うコンポーネントの代表格であり、関数を書くことでオブジェクトの動きをプログラミングできる。また、以下の関数が非常によく使われる。

表 3.5.1.1 Unity で使う C#で主に使われる関数

| | |
|---------------|---|
| Start() | ゲームオブジェクトの初期化に使われる関数で、後述の Update 関数が実行される前のタイミングで 1 度だけ実行される。 |
| Update() | 1 フレームに 1 回の頻度で呼び出される関数で、主にキー入力の受付を行うのに使われる。 |
| FixedUpdate() | 基本的には Update 関数と同じ動きであるが、一定秒数ごとに処理を行うときに使われる。 |
| LoadScene() | シーンを移動するときに呼ぶ関数で、ボタンを押してゲームを始める時やゲームオーバーの画面に移るときに使われる。 |

3.5.2 その他のコンポーネント

C#の他にも、Unityには便利なコンポーネントがとても多く存在しており、以下のコンポーネントが使われることが多い。

3.5.2.1 Box Collider

オブジェクトの当たり判定をするうえで必要になるコンポーネントである。C#で OnTriggerEnter 関数や OnCollisionEnter 関数と組み合わせることで、あるオブジェクトが別のオブジェクトと衝突した時にイベントを発生させることができる。

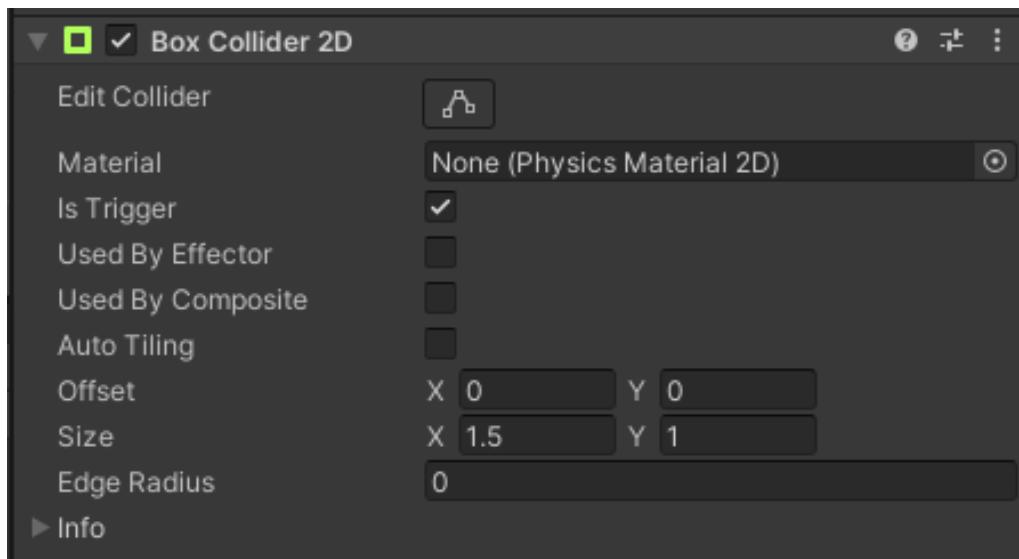


図 3.5.2.1.1 Box Collider の設定画面

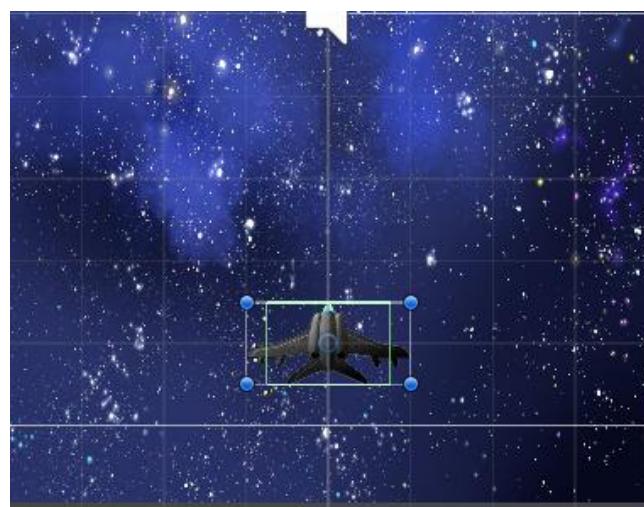


図 3.5.2.1.2 当たり判定を表す緑色の線

3.5.2.2 Rigidbody

オブジェクトを物理法則に則った動きをさせるために必要なコンポーネントである。オブジェクトの重さ、空気抵抗の大きさ、重力の有無、行動制限などを設定することができる。

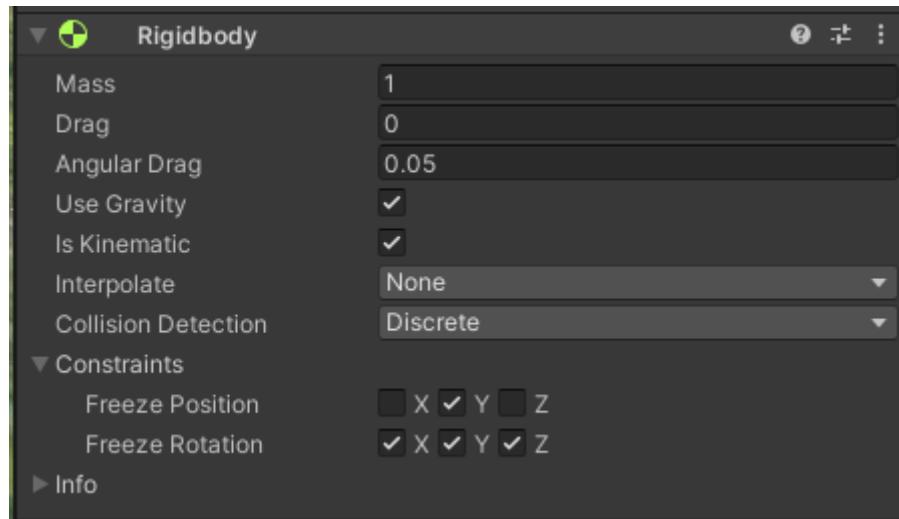


図 3.5.2.2.1 Rigidbody の設定画面

3.5.2.3 Canvas

Canvas コンポーネントは UI が配置、描画される抽象的な領域である。主にテキストを表示したり、ボタンを配置したり、画像を張り付けたりすることができる。また、「Order in Layer」でレイヤーの調整や「Render Mode」でカメラの調節もできる。

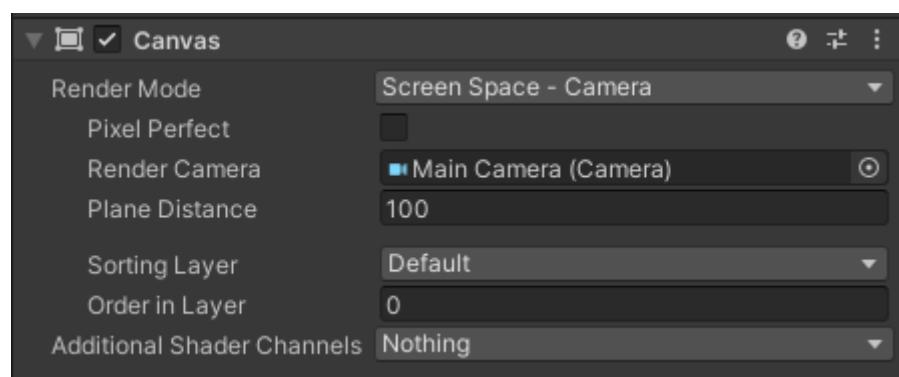


図 3.5.2.3.1 Canvas の設定画面

3.6 外部アセットとその使い方

Unity には、便利なツールが豊富にあるが、Asset Store という Unity のウェブサイトを利用することで、個人及び法人が制作したエフェクト、3D オブジェクト、音声などのアセットをダウンロードしてゲームに組み込むことができる。アセットは有料のものと無料のものがある。

Unity 制作画面の Window タブから「Asset Store」をクリックすると、以下のような画面になる。その後「Search online」をクリックする。

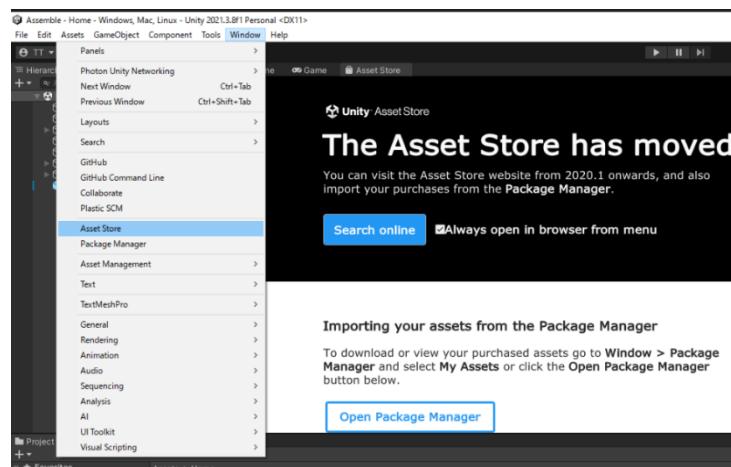


図 3.6.1 アセットストアに遷移するための準備画面

「Search online」をクリックすると、ブラウザが起動し、図 3.6.2 のウェブサイトが表示される。

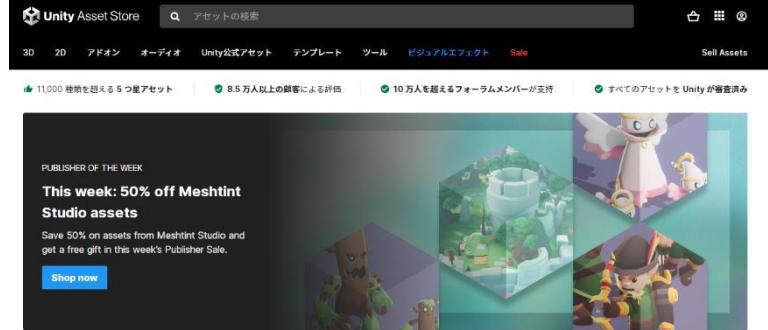


図 3.6.2 Unity Asset Store

URL: <https://assetstore.unity.com/>

使いたいアセットを検索したら、Add to My Assets をクリックする。

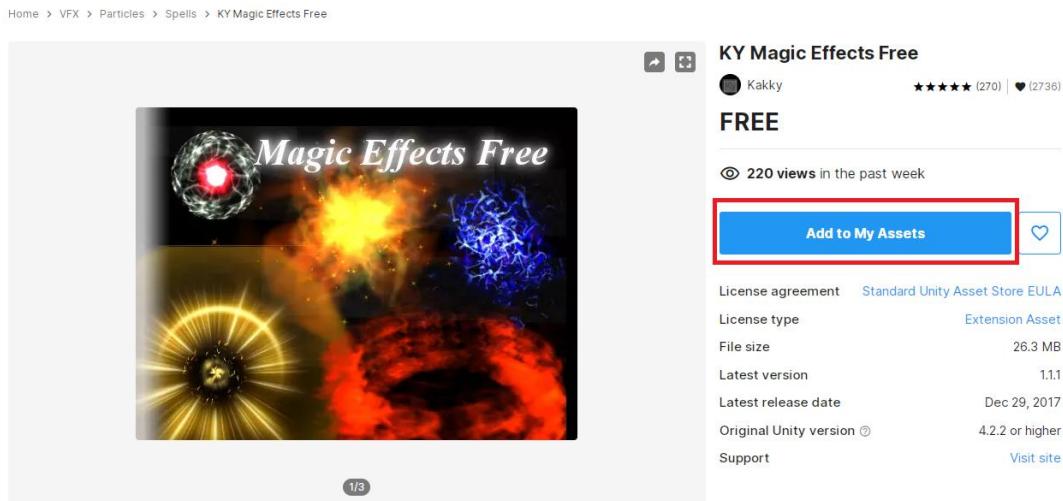


図 3.6.3 アセット購入画面

Open in Unity をクリックすると、Unity の画面に戻る。



図 3.6.4 Open in Unity

Asset Store で購入したアセットが選択されていることを確認し、「Download」をクリックする。



図 3.6.5 アセットのダウンロード画面

ダウンロードが完了したら、「Import」をクリックする。インポートが完了すると、Assets フォルダの下にダウンロードしたアセットフォルダが入っている。

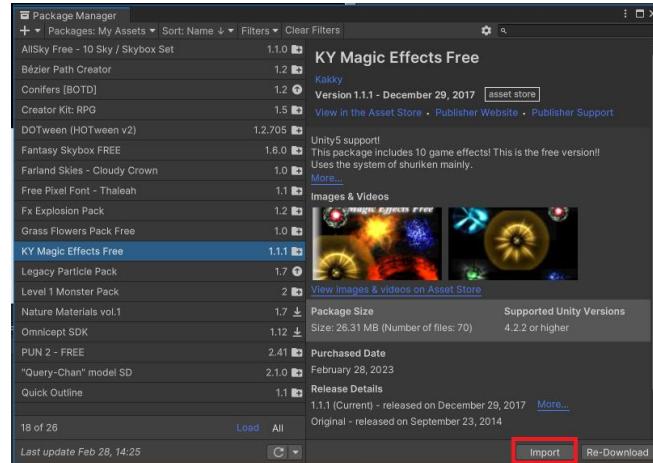


図 3.6.6 アセットのインポート画面 1

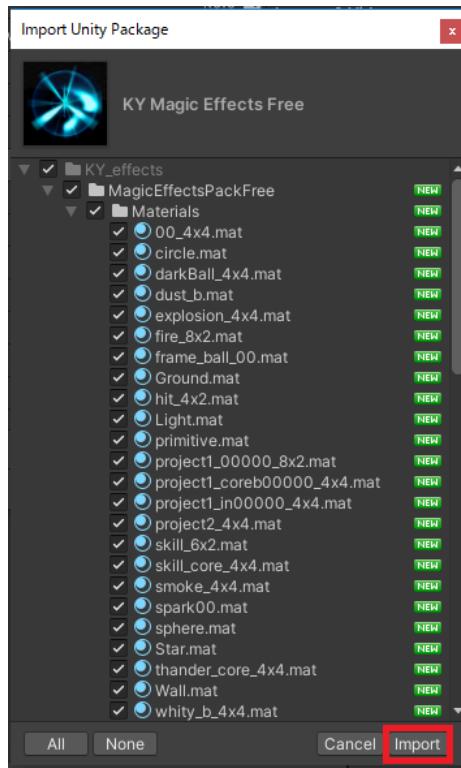


図 3.6.7 アセットのインポート画面 2

4 ゲーム紹介

4.1 タンクゲーム

4.1.1 ゲーム設計

自戦車を動かし、球を発射して、敵戦車を破壊するゲームである。ステージは全部で5つあり、そのすべてのステージのクリア条件を満たせば、ゲームクリアとなる。敵の攻撃により自分のHPが0になるか、制限時間を過ぎると、ゲームオーバーとなる。



図 4.1.1.1 プレイ画面



図 4.1.1.2 ゲームオーバー画面

4.1.2 メニュー設計

ホーム画面にはボタンが3つあり、「START」ボタンを押せばゲームスタート、「HOME」ボタンを押せば、4.8 メニュー画面の図 4.8.1.2 に戻り、「RULE」ボタンを押せば、ルール確認ができる。



図 4.1.2.1 ホーム画面

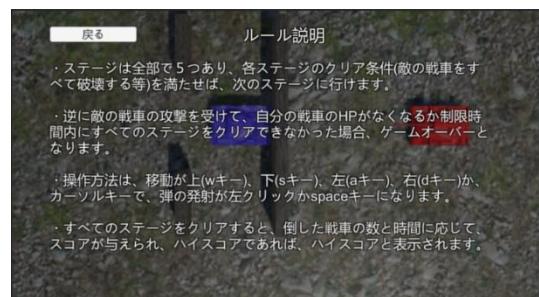


図 4.1.2.2 ルール説明画面

4.1.3 操作設計

プレイ画面では、自戦車を「A キー」で左、「D キー」で右、「W キー」で上、「S キー」で下に動かす。もしくは「カーソル左キー」で左、「カーソル右キー」で右、「カーソル上キー」で上、「カーソル下キー」で下に動かす。

自戦車の移動は Update 関数にコード 4.1.3.1 を記述した。

```
:  
//キーが押されたら、その方向に回転して動く  
if (Input.GetKey("up") || Input.GetKey(KeyCode.W)) {  
    this.transform.rotation = Quaternion.Euler(0, 0, 0);  
    this.transform.position += transform.forward * speed;  
} else if (Input.GetKey("down") || Input.GetKey(KeyCode.S)) {  
    this.transform.rotation = Quaternion.Euler(0, 180, 0);  
    this.transform.position += transform.forward * speed;  
} else if (Input.GetKey("right") || Input.GetKey(KeyCode.D)) {  
    this.transform.rotation = Quaternion.Euler(0, 90, 0);  
    this.transform.position += transform.forward * speed;  
} else if (Input.GetKey("left") || Input.GetKey(KeyCode.A)) {  
    this.transform.rotation = Quaternion.Euler(0, 270, 0);  
    this.transform.position += transform.forward * speed;  
}  
:
```

コード 4.1.3.1 YourTank.cs

マウスの「左クリック」か「スペース」キーを押すことで球が発射される。

```
:  
//スペースキーか、クリックが押されたとき  
if((Input.GetMouseButtonDown(0) == true  
    || Input.GetKeyDown(KeyCode.Space)) && firecount < 1){  
    firecount++;  
    //弾のオブジェクト(プレハブ)を生成  
    GameObject bullet = Instantiate(PreBullet);  
    //弾の発生場所をタンクの少し前方にする  
    bullet.transform.position = this.transform.position + this.transform.forward * 4.0f;  
    //弾の動力を設定  
    Rigidbody rbody = bullet.GetComponent<Rigidbody>();  
    rbody.AddForce(this.transform.forward * 1000);  
    //連射しても、弾が出すぎないように設定  
    if (firecount == 1){  
        //0.5秒かけてfirecountを0にする  
        Invoke("DelayMethod", 0.5f);  
    }  
}  
:  
:
```

コード 4.1.3.2 YourTank.cs

戦車はアセットストアでダウンロードした3Dモデルを使用した。

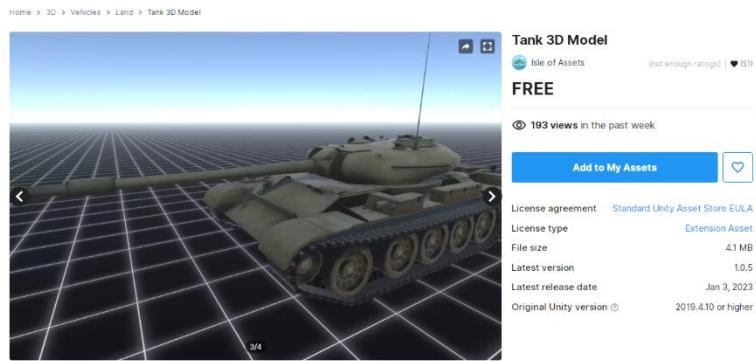


図 4.1.3.1 使用した戦車のアセット

URL: <https://assetstore.unity.com/packages/3d/vehicles/land/tank-3d-model-225955>

4.1.4 画面設計

プレイ画面には、戦車や壁などのゲームオブジェクトの他に画面上部にメニューボタン、HPバー、敵を倒した数、残り秒数が表示され、イベントが起こるたびに表示が変わる。



図 4.1.4.1 プレイ画面

```
//プレイ画面に変わった時に1回だけ実行
void Start() {
    :
    //HPバーを表示
    slider.value = (float)HP / (float)maxHP;
    GameObject gameScoreText = GameObject.Find("ScoreText");
    //倒した数を表示
    scoreText = gameScoreText.GetComponent<TextMeshProUGUI>();
    scoreText.text = score.ToString();
    //残り時間を表示
    GameObject gameLifeTimer = GameObject.Find("LifeTimer");
    LifeTimer = gameLifeTimer.GetComponent<Text>();
    LifeTimer.text = RemainTime.ToString("f0");
}
```

コード 4.1.4.1 GameSceneController.cs

4.1.5 発射設計

自戦車及び他戦車が弾を発射すると、煙エフェクトが発生する。



図 4.1.5.1 弾を発射したタンク

```
:  
void Start()  
{  
    //弾オブジェクト(プレハブ)が生成された時に発射音が鳴る  
    this.aud = GetComponent<AudioSource>();  
    this.aud.PlayOneShot(this.firesound);  
}  
void Update()  
{  
    // パーティクルシステムのインスタンスを生成する。  
    ParticleSystem newParticle = Instantiate(particle);  
    // パーティクルの発生場所をこのスクリプトをアタッチしているGameObjectの  
    // 場所にする。  
    newParticle.transform.position = this.transform.position;  
    // パーティクル(煙エフェクト)を発生させる。  
    newParticle.Play();  
    // インスタンス化したパーティクルシステムのGameObjectを削除する。(任意)  
    // ※第一引数をnewParticleだけにするとコンポーネントしか削除されない。  
    Destroy(newParticle.gameObject, 1.0f);  
}  
:
```

コード 4.1.5.1 Bullet.cs

煙エフェクトを出すために、まず、ヒエラルキーウィンドウから「Effects」を選択し、「Particle System」をクリックする。

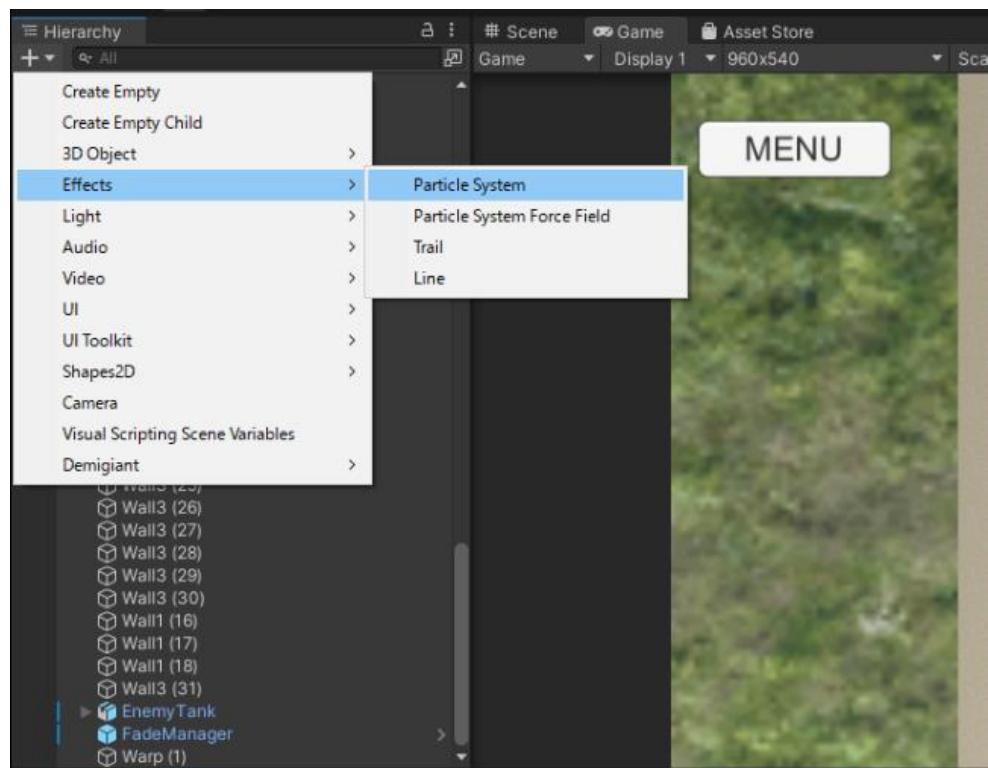


図 4.1.5.2 Particle System の作成

「Particle System」をクリックすると、ヒエラルキーインドウにオブジェクトが作成される。次に作成されたオブジェクトのインスペクタウィンドウから「Start Color」と「Shape」の設定を変更する。

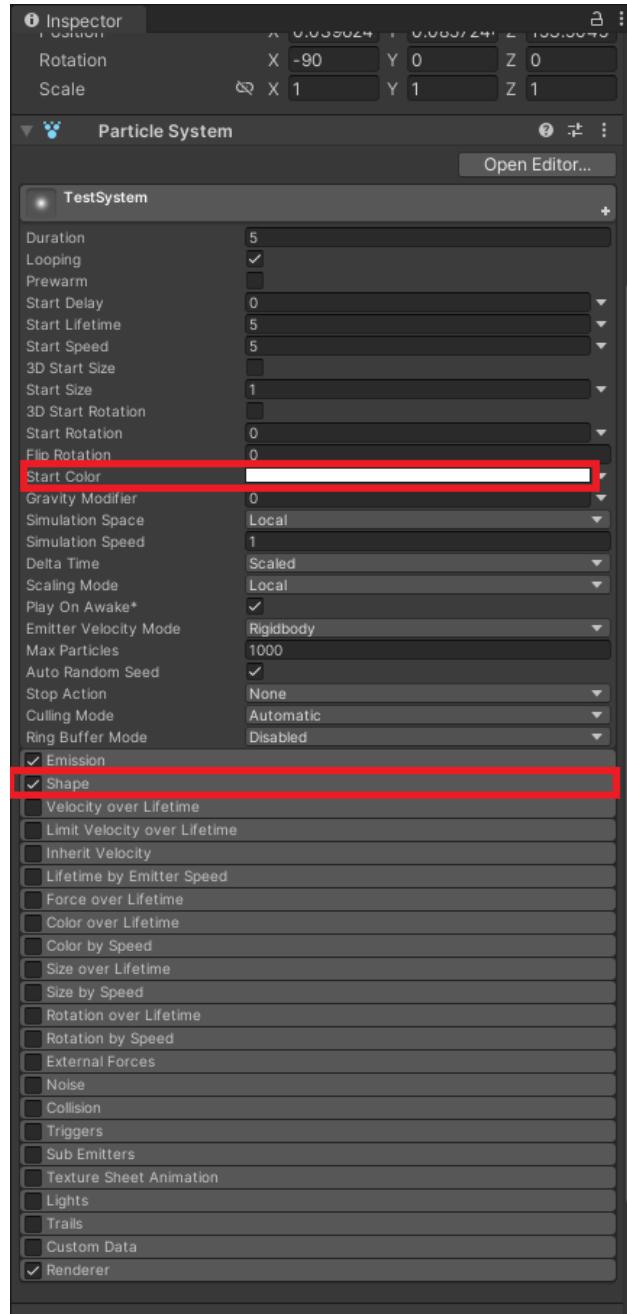


図 4.1.5.3 Particle System のインスペクタウィンドウ

「Start Color」の設定は図 4.1.5.4 に示す通り、A の欄の数値を 50 に変更する。

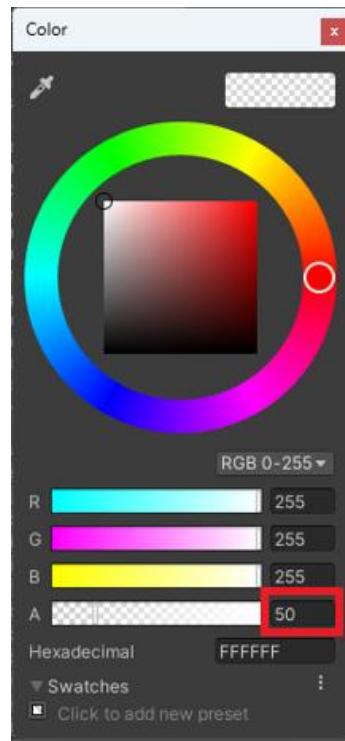


図 4.1.5.4 煙エフェクトのカラー設定

「Shape」の設定は図 4.1.5.5 に示す通り、「Shape」をデフォルトの設定「Cone」から「Sphere」に、「Radius」を 0.0001 に、「Scale」をすべて 0.3 に変更する。

この設定の「ParticleSystem」を「Bullet」オブジェクトの「Bullet.cs」にアタッチすることで弾を発射すると煙エフェクトが発生する。

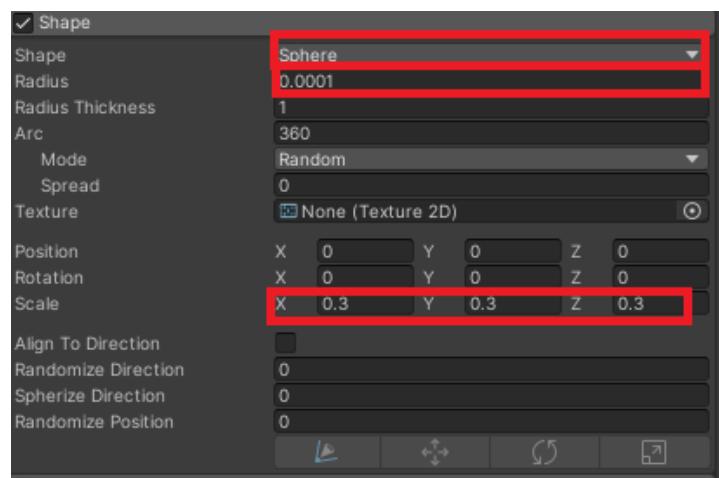


図 4.1.5.5 煙エフェクトの形状の設定

4.1.6 衝突設計

自戦車または、敵戦車によって発射された弾が戦車、別の弾、壁に衝突した場合、爆発エフェクトが発生する。



図 4.1.6.1 敵の弾が自分の戦車に当たった場面

弾は戦車や壁などと衝突した場合、即消滅する。

```
:  
//他のオブジェクトと衝突したときに実行される  
void OnCollisionEnter(Collision collision)  
{  
    //戦車か弾か壁に当たった場合  
    if (collision.gameObject.tag == "Tank"  
        || collision.gameObject.tag == "Bullet"  
        || collision.gameObject.tag == "Wall"  
        || collision.gameObject.tag == "innerWall") {  
        //弾を消滅する  
        Destroy(this.gameObject);  
    }  
}
```

コード 4.1.6.1 Bullet.cs

自戦車に弾が衝突した場合、HP が 1 減り、爆発エフェクトが発生する。HP が 0 になった場合、ゲームオーバーとなる。敵戦車に弾が衝突した場合、爆発エフェクトが発生し、消滅する。

```
:  
void OnCollisionEnter(Collision collision){  
    //弾オブジェクトと衝突した場合  
    if (collision.gameObject.tag == "Bullet") {  
        //GameSceneControllerオブジェクトのスクリプトにアクセス  
        GameObject controller = GameObject.Find("GameSceneController");  
        GameSceneController script =  
            controller.GetComponent<GameSceneController>();  
        //サウンドを鳴らす  
        script.sound();  
        //HPを1減らす  
        script.DamageHP(1);  
        //HPが0になった場合、ゲームオーバーになる  
        if (script.isAlive() == false){  
            script.OnLoadGameOverScene();  
        }  
        //パーティクル(エフェクト)発生  
        ParticleSystem newParticle = Instantiate(particle);  
        newParticle.transform.position = this.transform.position;  
        newParticle.Play();  
        //1秒後にパーティクルを消滅させる  
        Destroy(newParticle.gameObject, 1.0f);  
    }  
}  
:  
:
```

コード 4.1.6.2 YourTank.cs

戦車や壁にパーティクルをアタッチすることで、爆発エフェクトが発生する。爆発エフェクトは、発射設計の煙エフェクトと同様、図 4.1.5.2、図 4.1.5.3 のようにパーティクルを作成し、設定する。

色の設定は、図 4.1.6.2 の通り、G を 170、B を 0 に設定する。

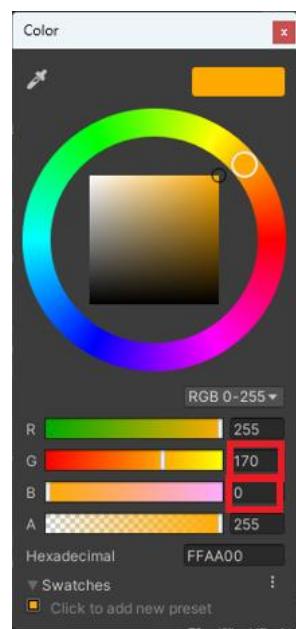


図 4.1.6.2 爆発エフェクトのカラー設定

「Emission」の「Bursts」の「+」をクリックし、「Shape」の「Shape」を「Sphere」にし、Radius を 0.01 に設定し、図 4.1.6.3 の通りにする。

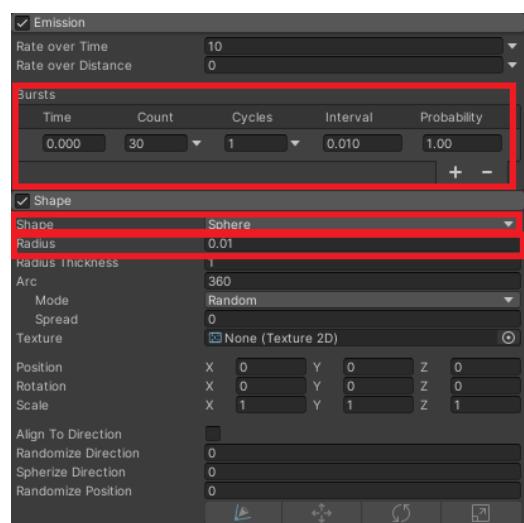


図 4.1.6.3 爆発エフェクトのParticleSystem のインスペクタウインドウ

4.2 カードゲーム

4.2.1 ゲーム設計

このゲームはターン制のカードゲームであり、11種類 15枚のカードの中から1枚クリックし、決定ボタンをクリックする。自分が選んだカードと相手が選んだカードの左上のパワーの数字が大きい方がバトルに勝つ。ターンが終わると、選択したカードは使えなくなる。バトルに勝つと、相手のライフを1000減らすことができる。引き分けの場合は減らない。先に相手のライフを0にした方が勝ちとなる。お互い、ライフが0にならずに手札がなくなった場合、ライフの残量が多い方が勝ちとなる。

4.2.2 メニュー設計

ホーム画面には、ボタンが4つあり、「HOME」ボタンを押せば4.8メニュー画面の図4.8.1.2メニュー画面に戻る。「CPU」ボタンを押せばCPUと、「ONLINE」ボタンを押せば通信対戦でそれぞれカードゲームができる。「RULE」ボタンを押すと、カードゲームのルールやカードの効果を確認することができる。



図 4.2.2.1 ホーム画面

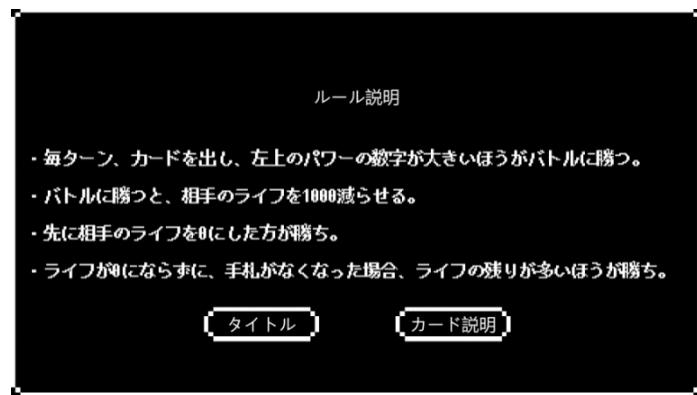


図 4.2.2.2 ルール説明画面

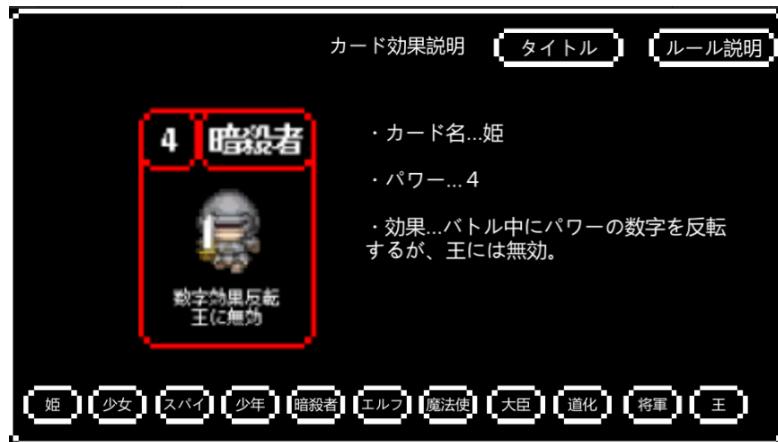


図 4.2.2.3 カード効果説明画面

4.2.3 初期表示設計

ゲームをスタートすると、まず、図 4.2.3.1 のゲーム画面に変わる。画面下部のライフポイントと表向きのカードが自分側で、画面上部は相手側のものになる。カードやボタンを左クリックすることでイベントが発生する。

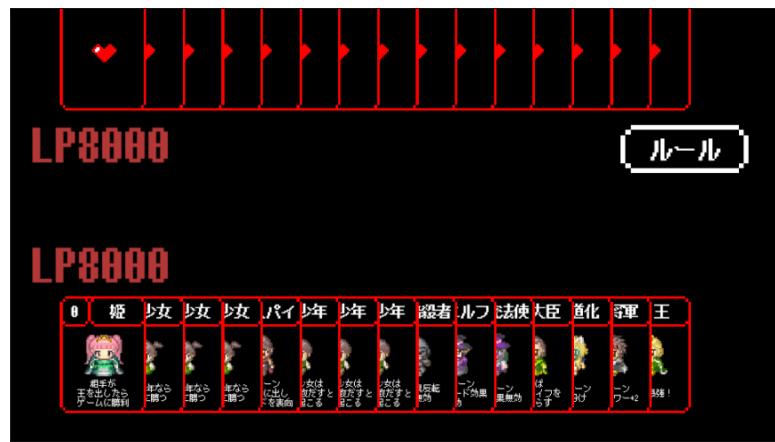


図 4.2.3.1 プレイ開始画面

```

        :
//ゲームの初期化設定
void Setup(){
    playerRetryReady = false;
    enemyRetryReady = false;

    //UIの初期化設定
    gameUI.Init();

    //LP(ライフポイント)の初期化設定と表示
    player.Life = 6000;
    enemy.Life = 6000;
    gameUI.ShowLifes(player.Life, enemy.Life);

    //一部のカードの初期化設定
    player.BoyCount = 0;
    player.GirlCount = 0;
    enemy.BoyCount = 0;
    enemy.GirlCount = 0;
    gameUI.UpdateAddNumber(player.AddNumber, enemy.AddNumber);
    gameUI.UpdateInvalid(false, false);
    gameUI.UpdateFirstSubmit(false, false);

    //カード提出イベント
    player.OnSubmitAction = SubmitttedAction;
    enemy.OnSubmitAction = SubmitttedAction;

    //カードの配置
    SendCardsTo(player, isEnemy: false);
    SendCardsTo(enemy, isEnemy: true);
    //自分が出すカードを決定したときに相手にカード情報を渡す
    if (GameDataManager.Instance.IsOnlineBattle){
        player.OnSubmitAction += SendPlayerCard;
    }
}

:

```

4.2.4 操作設計

場に出したいカードをクリックして、決定ボタンをクリックする。決定ボタンをクリックする前であれば、別のカードをクリックして場に出すカードを変更することができる。図4.2.5は「王」というカードを場に出した場面である。

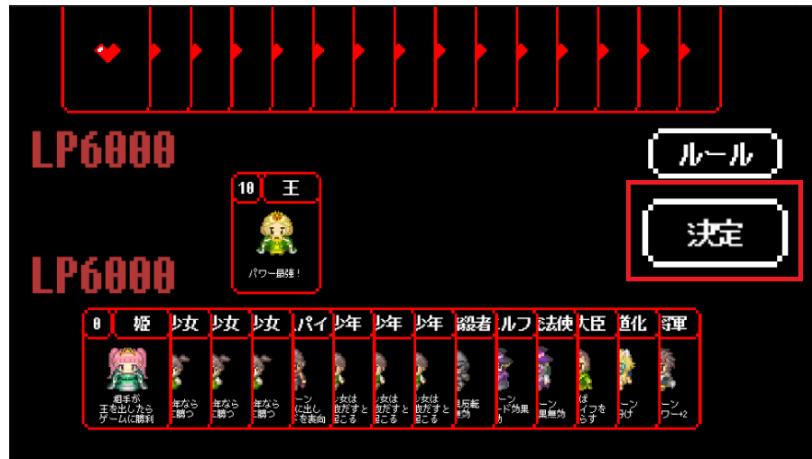


図 4.2.4.1 カードを場に出した場面

決定ボタンをクリックすると、コード4.2.4.1のSubmittedAction()関数が実行される。

```
//カード提出イベント
void SubmittedAction() {
    //自分と相手が両方カードを出した場合
    if(player.IsSubmitted && enemy.IsSubmitted) {
        //カードの勝敗判定
        StartCoroutine(CardsBattle());
    }
    //自分だけカードを出した場合
    else if (player.IsSubmitted) {
        //CPU戦の場合、CPUがランダムでカードを出す
        if(GameDataManager.Instance.IsOnlineBattle == false) {
            enemy.RandomSubmit();
        }
    }
}
```

コード 4.2.4.1 GameMaster.cs

4.2.5 バトル勝敗設計

両者がカードを出すとカードバトルになる。例えば、自分が出したカードがパワー10の「王」で、相手がパワー1の「少女」であれば、自分側が勝利し、相手のライフポイントを1000減らす。通常であればパワーが大きい方がバトルに勝つが、カードの効果により、勝敗が変更されることもある。



図 4.2.5.1 カードの勝敗判定

```
// Cardの勝利判定
IEnumerator CardsBattle() {
    //引数の秒数分処理を待つ
    yield return new WaitForSeconds(1f);
    //相手のカードを表向きにする
    enemy.SubmitCard.Open();
    yield return new WaitForSeconds(0.7f);
    //勝敗判定するための処理を行う
    Result result = ruleBook.GetResult(player, enemy);
    :
```

コード 4.2.5.1 GameMaster.cs

コード 4.2.5.1 の「result」変数を元にして勝敗判定後の負けた方のライフポイントを減らす処理や「WIN」と画面に表示する処理などを switch 文で行う。

```
:  
IEnumerator CardsBattle()  
{  
:  
    //勝敗判定の結果から、その後の処理を行う  
    switch (result)  
    {  
        case Result.TurnWin:  
        case Result.GameWin:  
            gameUI.ShowTurnResult("WIN");  
            enemy.Life-=1000;  
            break;  
        case Result.TurnWin2:  
            gameUI.ShowTurnResult("WIN");  
            enemy.Life -= 2000;  
            break;  
        case Result.TurnLose:  
        case Result.GameLose:  
            gameUI.ShowTurnResult("LOSE");  
            player.Life-=1000;  
            break;  
        case Result.TurnLose2:  
            gameUI.ShowTurnResult("LOSE");  
            player.Life -= 2000;  
            break;  
        case Result.TurnDraw:  
            gameUI.ShowTurnResult("DRAW");  
            break;  
    }  
:  
}
```

コード 4.2.5.2 GameMaster.cs

勝敗判定後のライフを減らす処理などが終わったら、ゲームの勝敗判定をする。
勝敗が決まればリザルト画面を表示する。勝敗がついていない場合はお互い場に出した
カードをゲームから削除し、次のターンになる。



図 4.2.5.2 自分がゲームに勝利した時の画面

```
:  
IEnumerator CardsBattle() {  
    :  
    gameUI.ShowLifes(player.Life, enemy.Life);  
    yield return new WaitForSeconds(1f);  
  
    //試合の勝敗が決まった場合、リザルト画面を表示  
    if ((player.Hand.IsEmpty) || (result == Result.GameWin) || (result ==  
        Result.GameLose) || (player.Life <= 0 || enemy.Life <= 0)) {  
        ShowResult(result);  
    }  
    //まだ続く場合、次のターンの準備をする  
    else {  
        SetupNextTurn();  
    }  
}  
:
```

コード 4.2.5.3 GameMaster.cs

4.2.6 カード設計

4.2.6.1 基本カード設計

カードは上の文字がカード名、左上の数字がパワー、下の文章がカードの特殊効果となる。真ん中の画像がカードイラストとなる。



図 4.2.6.1.1 カード

コード 4.2.6.1.1 の CardBase クラスを作成し、それをアセット化できるように設定した。このようにすることで、複数のカード情報を比較的簡単に管理、変更することが可能になった。

列挙型 CardType の変数「Magician」は「魔法使い」カード、「King」は「王」カード、「Clown」は「道化」カード、「Princess」は「姫」カード、「Spy」は「スパイ」カード、「Assassin」は「暗殺者」カード、「Minister」は「大臣」カード、「Shogun」は「将軍」カード、「Girl」は「少女」カード、「Boy」は「少年」カード、「Elf」は「エルフ」カードを示している。

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

[CreateAssetMenu]
//ScriptableObjectを継承することでクラスをアセット化できる
public class CardBase : ScriptableObject {
    //カードの基礎データの変数
    [SerializeField] new string name;
    [SerializeField] CardType type;
    [SerializeField] int number;
    [SerializeField] Sprite icon;
    [TextArea]
    [SerializeField] string description;
    //カードのデータを外部から取得できるように設定
    public string Name { get => name; }
    public CardType Type { get => type; }
    public int Number { get => number; }
    public Sprite Icon { get => icon; }
    public string Description { get => description; }
}

//カードの種類を列挙型で保存
public enum CardType {
    Magician, King, Clown, Princess, Spy, Assassin,
    Minister, Shogun, Girl, Boy, Elf,
}
```

コード 4.2.6.1.1 CardBase.cs



図 4.2.6.1.2 アセット化された CardBase クラス 11 個

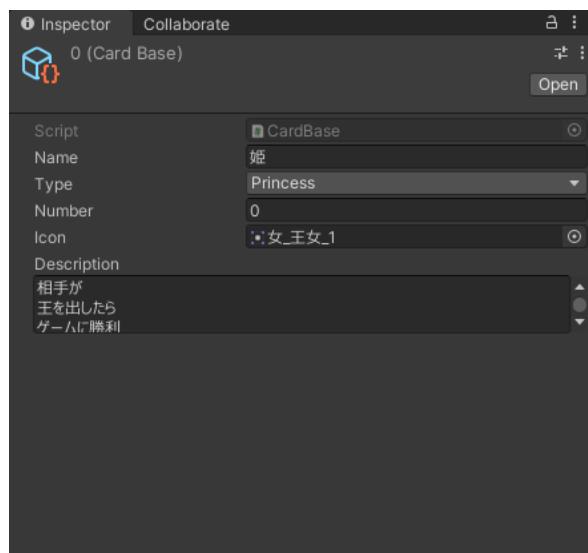


図 4.2.6.1.3 姫カードの設定

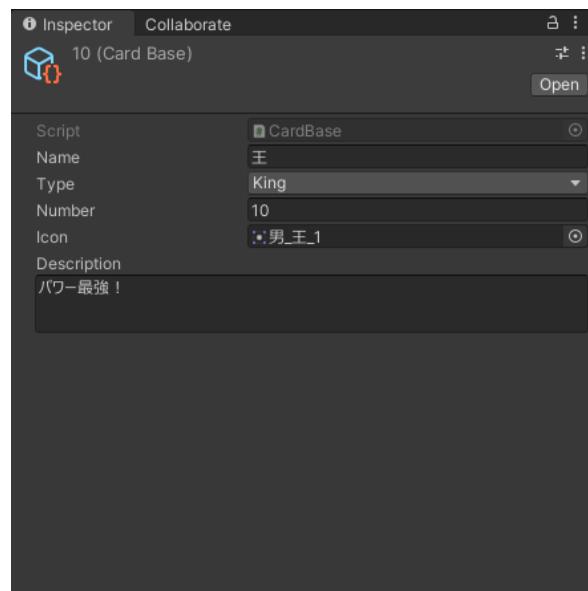


図 4.2.6.1.4 王カードの設定

4.2.6.2 姫カード設計

「姫(Princess)」はゲーム中に1枚使うことができ、パワー0で最弱である。しかし、自分が「姫」を出したターンに相手が「王(King)」を出した場合、即ゲームに勝利するという効果のカードである。



図 4.2.6.2.1 姫カード

```
:  
//カードの効果を処理し、結果を返す  
public Result GetResult(Battler player, Battler enemy)  
{  
    略  
    //自分が姫を使い、相手が王を使った場合、自分が勝つ  
    if (player.SubmitCard.Base.Type == CardType.Princess  
        && enemy.SubmitCard.Base.Type == CardType.King) {  
        return Result.GameWin;  
    }  
    //自分が王を使い、相手が姫を使った場合、相手が勝つ  
    if (enemy.SubmitCard.Base.Type == CardType.Princess  
        && player.SubmitCard.Base.Type == CardType.King) {  
        return Result.GameLose;  
    }  
    :  
}
```

コード 4.2.6.2.1 RuleBook.cs

4.2.6.3 少女カード設計

「少女(Girl)」はパワーが1で、ゲーム中に3枚まで使える。
また、「少女(Girl)」は「自分が『少女(Girl)』を出したターンに相手が『少年(Boy)』を出した場合、自分がバトルに勝つ」という効果を持っている。



図 4.2.6.3.1 少女カード

```
:  
//カードの効果を処理し、結果を返す  
public Result GetResult(Battler player, Battler enemy) {  
    :  
    //自分が少女を出し、相手が少年を出した場合、自分が勝つ  
    if (player.SubmitCard.Base.Type == CardType.Girl  
        && enemy.SubmitCard.Base.Type == CardType.Boy) {  
        return Result.TurnWin;  
    }  
    //相手が少女を出し、自分が少年を出した場合、相手が勝つ  
    if (player.SubmitCard.Base.Type == CardType.Boy  
        && enemy.SubmitCard.Base.Type == CardType.Girl){  
        return Result.TurnLose;  
    }  
    :  
}
```

コード 4.2.6.3.1 RuleBook.cs

また、少女は特定の条件で発動する効果を 2 つ持っている。1 つ目は「3 枚目の『少女(Girl)』を出したターンにのみ発動し、このターン、相手の効果を無効にする」という効果で、2 つ目は、「3 枚目の『少女(Girl)』を出したターンにのみ発動し、このターン、パワーが低い方がバトルに勝つ」という効果である。この 2 つの効果は別のカードの効果によって無効化されない。

```
    :  
    //カードの効果を処理し、結果を返す  
    public Result GetResult(Battler player, Battler enemy) {  
        :  
        //自分が少女を出した場合  
        if (player.SubmitCard.Base.Type == CardType.Girl) {  
            player.GirlCount++;  
            //3枚目の少女を出した場合、パワーが反転して、即バトル  
            if (player.GirlCount == 3) {  
                return NumberBattle(player, enemy,  
                    ministerEffect: false, reverseEffect: true);  
            }  
        }  
        :  
    }
```

コード 4.2.6.3.2 RuleBook.cs

4.2.6.4 スパイカード設計

「スパイ(Spy)」は、パワーが2で、次のターンに相手に先にカードを表向き状態で出させることができる。



図 4.2.6.4.1 スパイカード

```
//カードの効果を処理し、結果を返す
public Result GetResult(Battler player, Battler enemy) {
    :
    //どちらかがスパイを出した場合、次のターン、相手は先にカードを出さなければならぬ。
    if ((player.SubmitCard.Base.Type == CardType.Spy)
        && (enemy.SubmitCard.Base.Type != CardType.Spy)) {
        enemy.IsFirstSubmit = true;
    }
    if ((enemy.SubmitCard.Base.Type == CardType.Spy)
        && (player.SubmitCard.Base.Type != CardType.Spy)) {
        player.IsFirstSubmit = true;
    }
    :
}
```

コード 4.2.6.4.1 RuleBook.cs

```
:  
//次のターンの準備をする  
void SetupNextTurn() {  
    :  
    if(player.IsFirstSubmit || enemy.IsFirstSubmit){  
        //自分が前のターンにスパイを出していた場合  
        if (enemy.IsFirstSubmit) {  
            gameUI.UpdateFirstSubmit(false, true);  
            //CPU戦ならば相手は先にカードを表向きに出す  
            if (GameDataManager.Instance.IsOnlineBattle == false) {  
                enemy.IsFirstSubmit = false;  
                enemy.RandomSubmit();  
                enemy.SubmitCard.Open();  
            }  
            else {  
                player.IsSubmitted = true;  
            }  
        }  
        //前のターンに相手がスパイを出していた場合  
        if (player.IsFirstSubmit) {  
            //playerが先に出すというパネルを表示  
            gameUI.UpdateFirstSubmit(true, false);  
        }  
    }  
    :  
}
```

コード 4.2.6.5.2 GameMaster.cs

4.2.6.5 少年カード設計

「少年(Boy)」はパワーが 3 で、「少女(Girl)」と同様にゲーム中に 3 枚まで使える。「少年(Boy)」は「3 枚目の『少年(Boy)』を場に出した時に発動し、次のターンに出すカードのパワーを 2 上げることができる」という効果を持っている。この効果は他のカードの効果により無効化されない。



図 4.2.6.5.1 少年カード

```
:  
//カードの効果を処理し、結果を返す  
public Result GetResult(Battler player, Battler enemy) {  
    :  
    //少年を出した次のターン、カードのパワーが2上がる。  
    if (player.SubmitCard.Base.Type == CardType.Boy) {  
        player.BoyCount++;  
        if (player.BoyCount == 3) {  
            player.IsAddNumberMode = true;  
        }  
    }  
    if (enemy.SubmitCard.Base.Type == CardType.Boy) {  
        enemy.BoyCount++;  
        if (enemy.BoyCount == 3) {  
            enemy.IsAddNumberMode = true;  
        }  
    }  
    :  
}
```

コード 4.2.6.5.1 RuleBook.cs

```
:  
//次のターンの準備をする(提出カードを削除する)  
public void SetupNextTurn() {  
    :  
    //3枚目の少年か将軍を出した次のターン、カードのパワーが2上がる。  
    if (IsAddNumberMode) {  
        IsAddNumberMode = false;  
        AddNumber = 2;  
    }  
    else {  
        AddNumber = 0;  
    }  
    :  
}
```

コード 4.2.6.5.2 Battler.cs

4.2.6.6 暗殺者カード設計

「暗殺者(Assassin)」はパワー4で、ゲーム中に1枚使える。このカードを使ったターンはパワーが低い方がバトルに勝つが、「王(King)」には負ける。



図 4.2.6.6.1 暗殺者カード

```
//カードの効果を処理し、結果を返す
public Result GetResult(Battler player, Battler enemy) {
    :
    //どちらかが暗殺者を使い、そのもう一方が王子ではない場合、数字の強さが反転する。
    if (player.SubmitCard.Base.Type == CardType.Assassin
        && enemy.SubmitCard.Base.Type != CardType.King
        || (enemy.SubmitCard.Base.Type == CardType.Assassin
            && player.SubmitCard.Base.Type != CardType.King)){
        //パワー反転効果が発動した状態でのバトル結果を返す
        return NumberBattle(player, enemy, ministerEffect: false, reverseEffect: true);
    }
    :
```

コード 4.2.6.6.2 RuleBook.cs

```

        :
//カードバトル判定をし、結果を返す
Result NumberBattle(Battler player, Battler enemy, bool ministerEffect, bool
                    reverseEffect) {
    //等倍の時(大臣を出していない場合)
    if (ministerEffect == false) {
        //自分のカードのパワーが相手より高い場合
        if (player.SubmitCard.Base.Number + player.AddNumber >
            enemy.SubmitCard.Base.Number + enemy.AddNumber) {
            //暗殺者を出していた場合は負ける
            if (reverseEffect) {
                return Result.TurnLose;
            }
            return Result.TurnWin;
        }
        //自分のカードのパワーが相手より低い場合
        else if (player.SubmitCard.Base.Number + player.AddNumber <
                  enemy.SubmitCard.Base.Number + enemy.AddNumber) {
            //暗殺者を出していた場合は勝つ
            if (reverseEffect) {
                return Result.TurnWin;
            }
            return Result.TurnLose;
        }
    }
    :
}

```

コード 4.2.6.6.3 RuleBook.cs

4.2.6.7 エルフカード設計

「エルフ(Elf)」はパワーが5でゲーム中に1枚使える。次のターンに発動する相手のカードの効果を無効にする。しかし、このターンに発動した「スパイ(Spy)」や「エルフ(Elf)」効果を無効化できるわけではない。



図 4.2.6.7.1 エルフカード

```
:  
//カードの効果を処理し、結果を返す  
public Result GetResult(Battler player, Battler enemy) {  
    :  
    //エルフを出した場合、次のターン相手は効果を使えない  
    if (player.SubmitCard.Base.Type == CardType.Elf) {  
        enemy.Invalid = true;  
    }  
    if (enemy.SubmitCard.Base.Type == CardType.Elf) {  
        player.Invalid = true;  
    }  
    :  
}
```

コード 4.2.6.7.2 RuleBook.cs

4.2.6.8 魔法使いカード設計

「魔法使い(Magician)」はパワーが 6 で、このターンに発動する相手のカードの効果を無効にする。しかし、前のターンに発動した「スパイ(Spy)」や「エルフ(Elf)」効果を無効化できるわけではない。



図 4.2.6.8.1 魔法使いカード

「魔法使い(Magician)」の処理は、効果エフェクトを無効にした NumberBattle()関数を戻り値として返すことで、他の効果を発動させないようにしている。

```
//カードの効果を処理し、結果を返す
public Result GetResult(Battler player, Battler enemy) {
    :
    //自分が相手が魔法使いの場合、数字対決で即バトル
    if (player.SubmitCard.Base.Type == CardType.Magician
        || enemy.SubmitCard.Base.Type == CardType.Magician) {
        return NumberBattle(player, enemy, ministerEffect: false, reverseEffect: false);
    }
    :
```

コード 4.2.6.8.1 RuleBook.cs

4.2.6.9 大臣カード設計

「大臣(Minister)」はパワーが7でゲーム中に1枚使える。バトルに勝った時に相手のライフポイントを通常の2倍である2000減らせる。



図 4.2.6.9.1 大臣カード

```

        :
//カードの効果を処理し、結果を返す
public Result GetResult(Battler player, Battler enemy) {
    :
    if (player.SubmitCard.Base.Number + player.AddNumber >
        enemy.SubmitCard.Base.Number + enemy.AddNumber) {
        if (reverseEffect) {
            //相手が大臣の時、自分はライフを2000失う
            if (enemy.SubmitCard.Base.Type == CardType.Minister) {
                return Result.TurnLose2;
            }
            return Result.TurnLose;
        }
        //自分が大臣の時、相手はライフを2000失う
        if (player.SubmitCard.Base.Type == CardType.Minister) {
            return Result.TurnWin2;
        }
        return Result.TurnWin;
    }
    :
}

```

コード 4.2.6.9.1 RuleBook.cs

その後は、コード 4.2.5.2 の switch 文でライフを 2000 減らす処理をする。

4.2.6.10 道化カード設計

「道化(Clown)」はパワーが8で、ゲーム中に1枚使える。パワーに関わらずバトルを引き分けにする。



図 4.2.6.10.1 道化カード

```
:  
//カードの効果を処理し、結果を返す  
public Result GetResult(Battler player, Battler enemy) {  
    :  
    //自分か相手が道化の場合、引き分け  
    if (player.SubmitCard.Base.Type == CardType.Clown  
        || enemy.SubmitCard.Base.Type == CardType.Clown) {  
        return Result.TurnDraw;  
    }  
    :  
}
```

コード 4.2.6.10.1 RuleBook.cs

4.2.6.11 将軍カード設計

「将軍(Shogun)」はパワーが9で、次のターンに出すカードのパワーを2上げる。



図 4.2.6.11.1 将軍カード

```
:  
//カードの効果を処理し、結果を返す  
public Result GetResult(Battler player, Battler enemy) {  
    :  
    //将軍を出した場合、次のターンの数字が2増える(+2)モードにする。  
    if (player.SubmitCard.Base.Type == CardType.Shogun) {  
        player.IsAddNumberMode = true;  
    }  
    if (enemy.SubmitCard.Base.Type == CardType.Shogun) {  
        enemy.IsAddNumberMode = true;  
    }  
    :  
}
```

コード 4.2.6.11.1 RuleBook.cs

その後は、「少年(Boy)」カードと同様に、コード 4.2.6.5.2 の Battler.cs で処理する。

4.2.6.12 王カード設計

「王(King)」はパワーが 10 で、このカードゲームでパワーが一番高いカードであるが、能力はない。



図 4.2.6.12.1 王カード

4.2.7 オンライン設計

カードゲームは1人用のCPUモードの他に2人で戦うオンライン対戦モードを実装している。PhotonというツールをUnityにインポートすることでオンラインゲームを作ることができる。

4.2.7.1 Photon の導入

Unityでオンラインゲームを実装するために、まず、Photon公式サイトにアクセスし、アカウント登録、サインインをした後にアプリケーション登録して、アプリケーションIDを発行する。

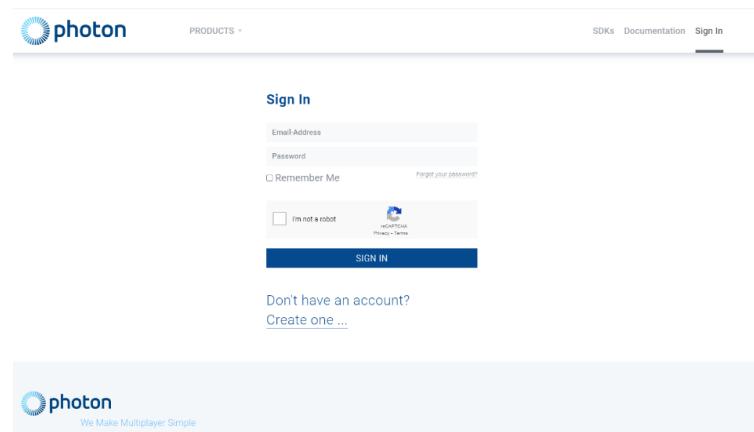


図 4.2.7.1.1 Photon 公式サイトのサインイン画面

図 4.2.24 の画面が表示されたら、「CREATE A NEW APP」をクリックする。

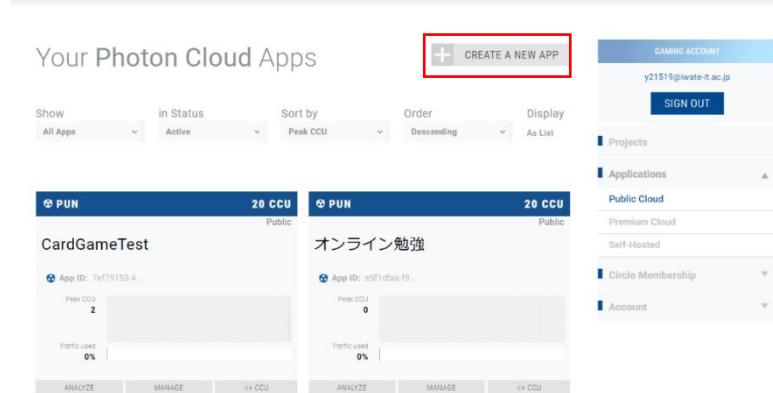


図 4.2.7.1.2 Photon 公式サイトのアプリケーション登録画面

次に「Select Photon SDK」を「Pun」に「Application Name」を任意の名前に設定し、「CREATE」ボタンをクリックすると、アプリケーション登録が完了する。

Select Application Type *

Multiplayer Game
You are a gaming company creating a multiplayer game targeting any device. Your customers are end-consumers.

Non-Gaming App
Applications for the non-gaming industry, and/or not selling directly to end-consumers, e.g. defense, education, medical, simulation, meeting, business, events and metaverse, sports, fitness.

Select Photon SDK *

Pun

Application Name *

Sotsugyo

Description

Short description, 1024 chars max.

Url

http://enter.your-url.here/ e.g. marketing material, landing page, promo site, etc.

CANCEL **CREATE**

図 4.2.7.1.3 アプリケーション登録設定画面

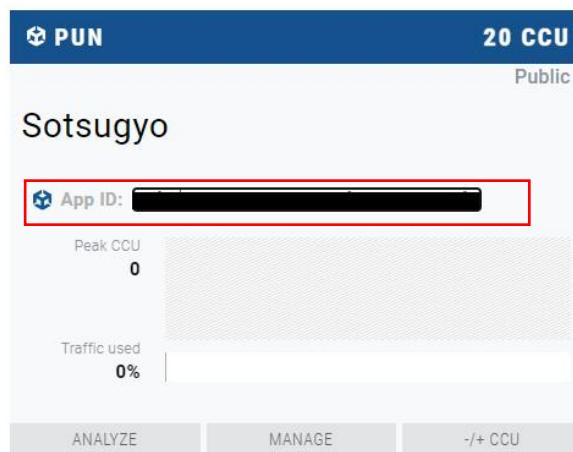


図 4.2.7.1.4 発行されたアプリケーションと APP ID

4.2.7.2 Photon の Unity への実装方法

次に Unity に図 4.2.7.2.1 のアセットを Unity Asset Store から PUN 2 - FREE をダウンロードし、インポートする。



図 4.2.7.2.1 アセットストアの PUN 2 - FREE

その後に Unity で先ほど取得した Photon のサイトでアプリケーション登録をした時に発行された図 4.2.7.1.4 の APP ID を入力し、Photon Server Settings を図 4.2.7.2.2 のように変更した。これで、Photon の基本設定は完了する。

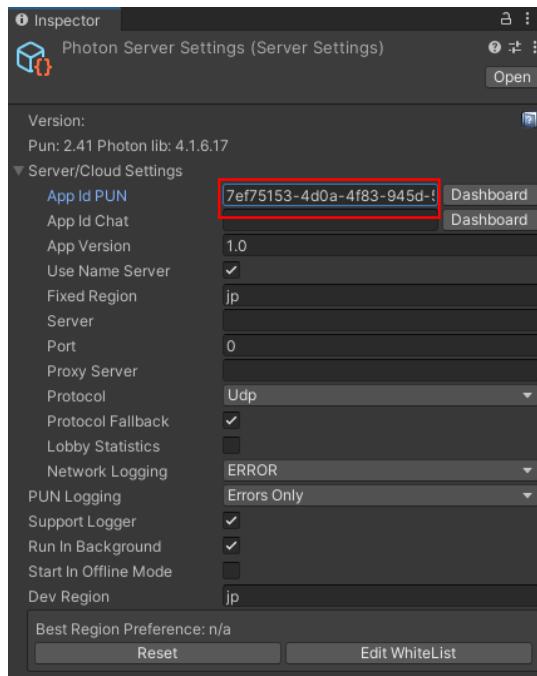


図 4.2.7.2.2 Photon Server Settings のインスペクタ ウィンドウ

4.2.7.3 ゲームへの実装

カードゲームをオンライン対戦するために最初に必要になるのがマッチング作業である。カードゲームの図 4.2.2.1 のホーム画面から「ONLINE」ボタンを押すと、マッチング画面に移り、「Matching」ボタンを押すと、対戦相手を探す。他のプレイヤーとマッチングすると、そのプレイヤーとカードゲームで対戦することができる。



図 4.2.7.3.1 マッチング画面



図 4.2.7.3.2 マッチング中の画面

```

using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using Photon.Pun;
using Photon.Realtime;
using UnityEngine.SceneManagement;

public class OnlineMenuManager : MonoBehaviourPunCallbacks {

    [SerializeField] GameObject loadingAnim;
    [SerializeField] GameObject matchingButton;
    [SerializeField] GameObject titleButton;
    [SerializeField] GameObject matchingMessage;
    bool inRoom;
    bool isMatching;

    //ボタンが押されたら、サーバーへ接続し、マッチング作業を開始
    public void OnMatchingButton() {
        loadingAnim.SetActive(true);
        matchingButton.SetActive(false);
        matchingMessage.SetActive(true);
        // PhotonServerSettingsの設定内容を使ってマスターサーバーへ接続する
        PhotonNetwork.ConnectUsingSettings();
    }

    public void OnTitleBtn() {
        SceneManager.LoadScene("CardTitle");
    }

    //サーバーに接続し、マッチング部屋に入ることに成功した時に実行される関数
    public override void OnJoinedRoom() {
        inRoom = true;
    }

    :
}

```

コード 4.2.7.3.1 OnlineMenuManager.cs

マッチング作業は、まず、マッチング部屋で待っている相手がいるかどうかを探す。いれば、その人と合流し、対戦がスタートし、いなければ、自分がマッチング部屋を作り、他の人が入ってくるのを待つ。

```
:  
public class OnlineMenuManager : MonoBehaviourPunCallbacks {  
    :  
    //部屋が見つからなかった場合  
    public override void OnJoinRandomFailed(short returnCode, string message) {  
        //2人マッチング部屋を作る  
        PhotonNetwork.CreateRoom(null, new RoomOptions()  
            MaxPlayers = 2 }, TypedLobby.Default);  
    }  
    //部屋が二人ならシーンを変える  
    private void Update() {  
        //シーン移動したらマッチング作業をしないためのif  
        if (isMatching) {  
            return;  
        }  
        //マッチング完了した場合、対戦スタート  
        if (inRoom) {  
            if (PhotonNetwork.CurrentRoom.MaxPlayers  
                == PhotonNetwork.CurrentRoom.PlayerCount) {  
                isMatching = true;  
                SceneManager.LoadScene("Game");  
            }  
        }  
    }  
}
```

コード 4.2.7.3.2 OnlineMenuManager.cs

マッチングが完了すると、CPU 戦と同様にカードゲームが始まる。ルールやゲームの進み方は同じだが、ソースコードの処理が一部異なっている部分がある。コード 4.2.23 の関数はオンライン対戦のみで実行される関数である。

```
:  
//自分がカードを出した時  
void SendPlayerCard() {  
    //第1引数が相手に実行させる関数名、第2引数が実行させる相手、第3引数が  
    //送る内容  
    photonView.RPC(nameof(RPCOnReceivedCard), RpcTarget.Others,  
                    player.SubmitCard.Base.Number);  
}  
[PunRPC]  
//相手が出したカード情報を受け取る関数  
void RPCOnReceivedCard(int number) {  
    //相手側の敵カードが出される  
    enemy.SetSubmitCard(number);  
    //前のターンにスパイカードを出していた場合、相手が出したカードを表向  
    //きにする  
    if (enemy.IsFirstSubmit)  
    {  
        enemy.SubmitCard.Open();  
        enemy.IsFirstSubmit = false;  
        player.IsSubmitted = false;  
    }  
}  
:  
:
```

コード 4.2.7.3.3 GameMaster.cs

4.3 シューティングゲーム

4.3.1 ゲーム設計

敵機の攻撃に当たらないように自機を操作しながら弾を発射し、スコアを伸ばすゲームである。



図 4.3.1.1 ホーム画面

敵機やボス敵機の攻撃に当たるとゲームオーバーとなり、リトライ画面へと移動する。

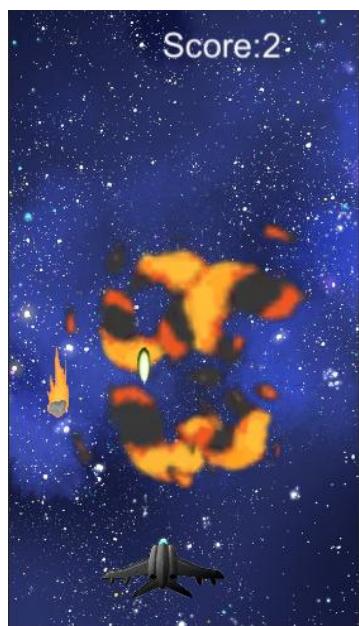


図 4.3.1.2 プレイ画面

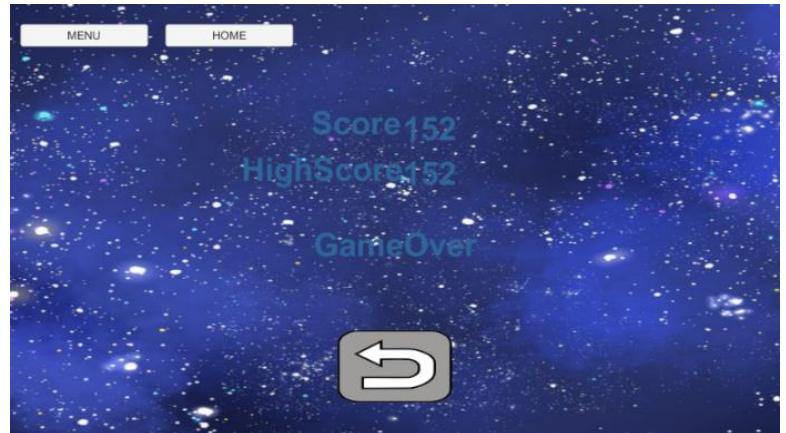


図 4.3.1.3 リトライ画面

4.3.2 メニュー設計

図4.3.1.1のホーム画面からルール説明、敵機情報を確認するパネルを作成した。タイトルボタンを押すことでホーム画面に移動する。

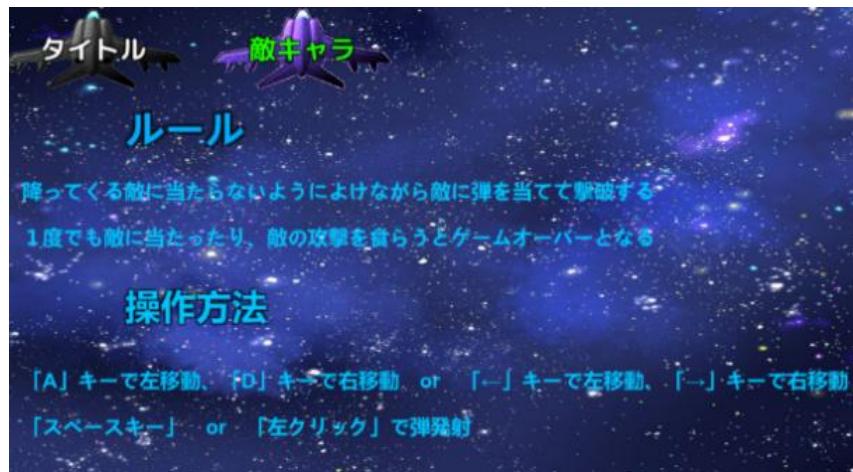


図4.3.2.1 ルール画面



図4.3.2.2 敵機説明画面

4.3.3 操作設計

自機を「A キー」で左、「D キー」で右に動かす。もしくは「カーソル左キー」で左、「カーソル右キー」で右に動かす。

```
:  
if (Input.GetKey(KeyCode.LeftArrow) || Input.GetKey(KeyCode.A)){  
    transform.Translate(-0.0065f, 0, 0);  
}  
if (Input.GetKey(KeyCode.RightArrow) || Input.GetKey(KeyCode.D)){  
    transform.Translate(0.0065f, 0, 0);  
}
```

Transform.Translate はロケットの移動スピードを指定（第一引数で左右、第二引数で上下、第三引数で奥行きのスピードを指定）

:

コード 4.3.3.1 rocket.cs

マウスの「左クリック」か「スペースキー」を押すことで自機から弾が発射される。

```
:  
if (Input.GetKeyDown(KeyCode.Space) || Input.GetMouseButtonDown(0)){  
    Instantiate(bulletPrefab, new Vector3(this.transform.position.x,  
    this.transform.position.y + 0.8f, 0f), Quaternion.identity);  
}
```

:

コード 4.3.3.2 rocket.cs

弾と敵機の衝突時爆発エフェクトが表示される。

```
:
void OnTriggerEnter2D(Collider2D coll) {
    // 爆発エフェクトを生成する
    GameObject effect = Instantiate (explosionPrefab, transform.position,
    Quaternion.identity) as GameObject;
    Destroy (effect, 1.0f);
    Destroy (gameObject);
}
```

コード4.3.3.3 BulletController.cs

使用エフェクトはアセットストアからインポートした。

(Toon Explosion VFX Texture Free)

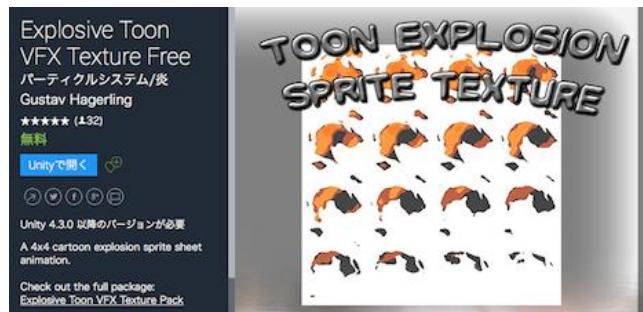


図 4.3.3.1 使用アセット

4.3.4 スコア設計

敵機にはただ降ってくるだけの通常敵機、特定の技を放つボス敵機があり、敵機と弾が衝突するごとにスコアが1上がり、一定値になるごとにボス敵機を出現させる。

```
:  
public void AddScore(){ //弾と敵が衝突すると左の関数が呼び出され、スコアが  
    score += 1;           上がる  
}  
public void GenBoss(){  
    if ((score >= 30) && (exist_luffy == false)){  
        Instantiate(luffyPrefab);  
        exist_luffy = true;  
    }  
}  
:  
:
```

コード 4.3.4.1 UIManager.cs

ハイスコアを更新した場合その値がハイスコアとして反映される。

```
:  
void Update(){  
    if(totalScore > highsore){  
        highsore = totalScore;  
        PlayerPrefs.SetInt(key,highsore);  
        HighScoreText = GameObject.Find("HighScoreText").GetComponent<Text>();  
        HighScoreText.text = highsore.ToString();  
    }  
}  
:  
:
```

コード 4.3.4.2 RestartSceneController.cs

4.3.5 難易度設計

難易度は「Normal」と「Hard」がある。

1つ目の変更点としては、通常敵機の出現頻度を上げたことである。

```
:  
InvokeRepeating ("GenRock", 1, 1); //Normal
```

第2引数が待機時間、第3引数がリピート秒

:

コード 4.3.5.1 RockGenerator.cs

```
:  
InvokeRepeating("GenRock", 1, 0.5f); //Hard
```

:

コード 4.3.5.2 HardRockGenerator.cs

2つ目は Scene が HardShootingScene の時は敵機の HP が増えることである。

```
:  
float HP = 2; //Normal  
:  
if (SceneManager.GetActiveScene().name == "HardShootingScene") {  
    HP = 3; //Hard  
}
```

:

コード 4.3.5.3 RockController.cs

```
:  
float HP = 100; //Normal  
:  
if (SceneManager.GetActiveScene().name == "HardShootingScene") {  
    HP = 120; //Hard  
}
```

:

コード 4.3.5.4 luffyController.cs

4.3.6 第一ボス敵機設計



図 4.3.6.1 第一ボス敵機

仁王像を参考に第一ボス敵機を作成した。イメージに合うよう攻撃用のパンチイラストを作成した。ボス敵機、パンチイラストはペイントで作成した。仕様は以下の 3 つである。

- ① 常に左右に移動する。



図 4.3.6.2 第一ボス敵機移左右移動

```
        :  
void Update () {  
    this.transform.position = newVector3(Mathf.Sin(Time.time) * 2.5f +  
    turnTime.x,2.8f,0f);  
}  
        :
```

コード 4.3.6.1 LuffyController.cs

- ② 一定時間ごとにパンチをくりだす。

```
        :  
void Start(){  
    InvokeRepeating("MakeKong",5,8);  
}  
        :
```

コード 4.3.6.2 LuffyController.cs

- ③ パンチは破壊可能である。

```
        :  
void OnTriggerEnter2D(Collider2D coll) {  
    HP -= 1;  
    if(HP == 0){  
        Destroy (gameObject);  
    }  
}
```

コード 4.3.6.3 KongController.cs

4.3.7 第二ボス敵機設計



図 4.3.7.1 第二ボス敵機

阿修羅を参考に第二ボス敵機を作成した。攻撃用の衝撃波イラストはフリー素材を使用した。ボス敵機イラストはペイントで作成した。移動方法は第一ボス敵機と同じ動きにした。異なる点は一定時間ごとに放つ衝撃波が破壊できないように OnTriggerEnter 関数を使わずに実装した点である。

4.3.8 第三ボス敵機設計



図 4.3.8.1 第三ボス敵機

雷神ソーを参考に第三ボス敵機を作成した。攻撃用のビームイラストはフリー素材を使用した。ボス敵機イラストはペイントで作成した。移動方法は第一ボス敵機と同じ動きにした。

異なる点としては一定時間で放つビームは事前に射程範囲から離れないと回避できない点である。

4.3.9 第四ボス敵機設計



図 4.3.9.1 第四ボス敵機

ピエロを参考に第四ボス敵機を作成した。攻撃用のサークルイラストはフリー素材を使用した。ボス敵機イラストはペイントで作成した。移動方法は第一ボス敵機と同じ動きにした。異なる点は以下の 2 つである。

- ① ボスと青いサークルが同時に出現し、自機の位置が変わる。



図 4.3.9.2 自機の位置変化後

```
:  
private bool change = false;  
void OnTriggerEnter2D(Collider2D coll){  
    if(coll.name == target.name && change == false){  
        target.gameObject.transform.Translate(0,4,0);  
        change = true;  
    }  
}  
:  
:
```

コード 4.3.9.1 LawController.cs

- ② ボスを倒すことでロケットの位置が元に戻る。

```
:  
target = GameObject.Find("rocket");  
void OnTriggerEnter2D(Collider2D coll) {  
    target.gameObject.transform.Translate(0,-4,0);  
}  
:  
:
```

コード 4.3.9.2 LawController.cs

4.3.10 最終敵機(第一形態)設計



図 4.3.10.1 最終敵機(第一形態)

デジコンにて産技短代表のグループが作成したキャラクター「もとまろ」を参考にした。移動方法は第一ボス敵機と同じ動きにした。異なる点は以下の 2 つである。

- ① 攻撃手段は一定時間で「落下→戻る」を繰り返す。

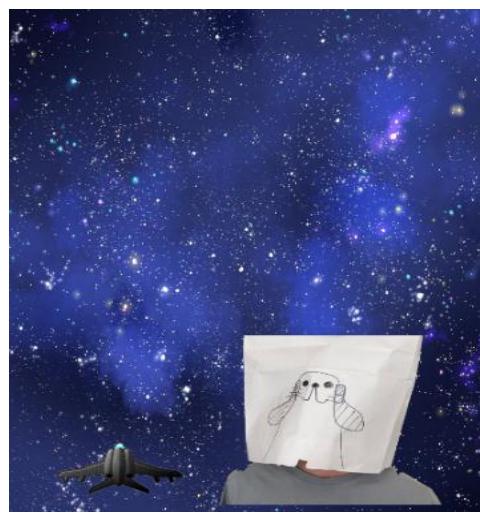


図 4.3.10.2 最終敵機(第一形態)攻撃

```

        :
void Update(){
    timer += Time.deltaTime;
    if(timer <= 4.7f){
        this.transform.position = new Vector3(Mathf.Sin(Time.time) * 2.5f +
        turnTime.x,3.5f,0f);
    }else if(timer > 4.7f && timer <= 5.4f){
        if(this.delta > this.span){
            this.delta = 0;
            transform.Translate( 0, -fallSpeed, 0, Space.World);
        }
    }else if(timer > 5.4f && timer <= 5.82f){
        transform.Translate (0, 0.06f, 0);
    }else{
        timer = 0;
    }
}
        :

```

コード 4.3.10.1 first_motomaroController.cs

- ② HP が 0 になると第二形態に変化する。

```

        :
void OnTriggerEnter2D(Collider2D coll) {
    HP -= 1;
    if(HP == 0){
        second_motomaroBoss();
    }
}
private bool exist_motomaro = false;
public void second_motomaroBoss(){
    if(exist_motomaro == false){
        Instantiate(second_motomaroPrefab);
        exist_motomaro = true;
    }
}

```

コード 4.3.10.2 first_motomaroController.cs

4.3.11 最終敵機(第二形態)設計



図4.3.11.1 最終敵機(第二形態)

デジコンにて産技短代表のグループが作成したキャラクター「もとまろ」を参考にした。移動方法は第一ボス敵機と同じ動きにした。異なる点は以下の3つである。

- ① 第一形態のHPが0になると同時に出現する。
- ② 時間が経過するごとに巨大化する。



図4.3.11.2 最終敵機(第二形態)攻撃

```
        :  
void Update(){  
    transform.localScale = new  
    Vector3(transform.localScale.x+0.0001f,transform.localScale.y+0.0001f,  
transform.localScale.z+0.0001f);  
}  
        :
```

コード4.3.11.1 second_motomaroController.cs

- ③ HPが0になると最終形態に変化する。

```
        :  
void OnTriggerEnter2D(Collider2D coll) {  
    HP -= 1;  
    if(HP == 0){  
        final_motomaroBoss();  
    }  
}
```

コード4.3.11.2 second_motomaroController.cs

4.3.12 最終敵機(最終形態)設計



図4.3.12.1 最終敵機(最終形態)

デジコンにて産技短代表のグループが作成したキャラクター「もとまろ」を参考にした。移動方法は第一ボス敵機と同じ動きにした。異なる点は以下の点である。

第一ボス敵機の技で攻撃してき、HPが低下すると第二ボス敵機の攻撃に変化する。続いて第三、と順番に最終敵機の第二形態の攻撃まで繰り返す。

```
:
void Attack(){
    if((HP > 420) && (HP <= 500) && (kong == false)){
        InvokeRepeating("MakeKong",4,4); kong = true;
    }
    GameObject obj = GameObject.Find ("konguganPrefab");
    if(HP <= 425){
        CancelInvoke("MakeKong");
        Destroy(obj);
    }
}
void zoro(){
    if((HP > 340) && (HP <= 420) && (knife == false)){
        InvokeRepeating("MakePondo",4,4); knife = true;
    }
}
:
```

コード4.3.12.1 final_motomaroController.cs

4.4 反射神経ゲーム

4.4.1 ゲーム設計

「！」が表示されたら素早くクリックし、敵キャラクターを倒すゲームである。敵キャラクターを倒すことで図 4.4.1.2 のステージクリア画面、負けると図 4.4.1.3 のリトライ画面へと移動し、最後の敵キャラクターを倒すと図 4.4.1.4 のクリア画面へ移動する。



図 4.4.1.1 プレイ画面



図 4.4.1.2 ステージクリア画面



図 4.4.1.3 リトライ画面



図 4.4.1.4 クリア画面

4.4.2 ホーム画面設計

図 4.4.2.1 にある「ルール説明ボタン」を押すことでルールを確認できるようパネルを作成した。



図 4.4.2.1 ホーム画面

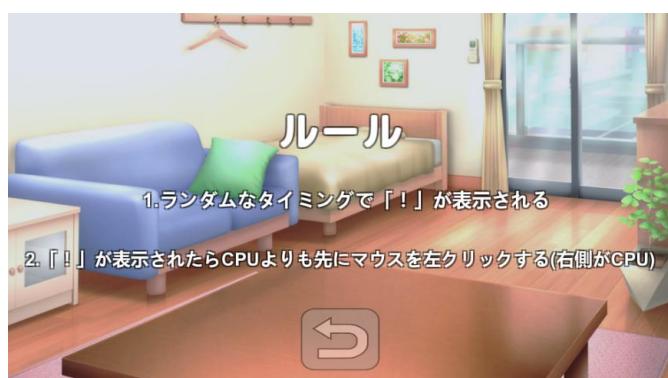


図 4.4.2.2 ルール説明画面

4.4.3 「！」表示設計

ゲームが始まると時間の計測が始まり指定時間内のランダムなタイミングで「！」が表示される。

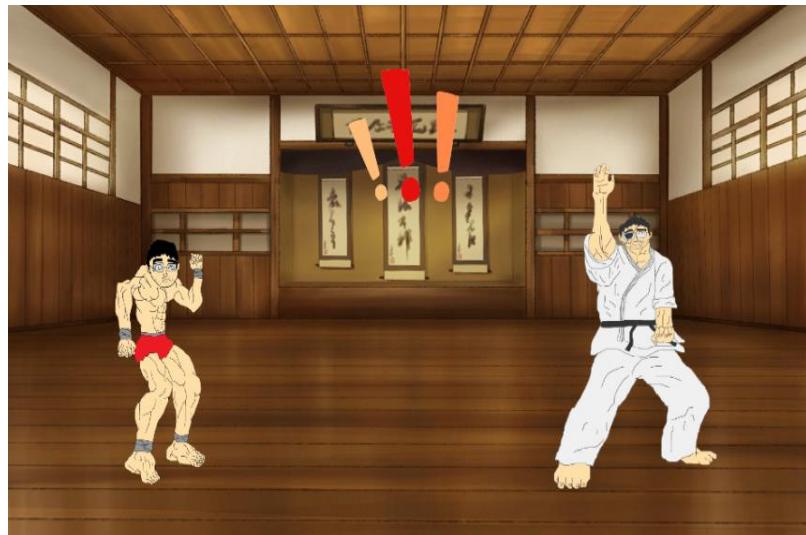


図 4.4.3.1 「！」表示画面

```
:
void Start(){
    StartCoroutine("sign");
}
IEnumerator sign(){
yield return new WaitForSeconds(UnityEngine.Random.Range(4.0f,5.0f));
    Instantiate(start_signPrefab);
    is_sign = true;
    stopwatch.Start(); //計測開始
    StartCoroutine("cpu_sign");
}
:
```

コード 4.4.3.1 UI.cs

4.4.4 勝利判定設計

「！」が表示されたらクリックし、判定時間より早くクリックできれば勝ちとなり、図Xのステージクリア画面へと移動する。ステージが進むごとに判定時間は早くなる。



図 4.4.4.1 勝利画面

```
:
void Update(){
    if(Input.GetMouseButtonDown(0) && is_sign == true){
        test_Character.transform.position = new Vector3(4f,-1.8f,0f);
        stopwatch.Stop(); //計測終了
        float elapsed = (float)stopwatch.Elapsed.TotalSeconds;
        if(elapsed <= 0.4f){
            Destroy(test_cpu.gameObject);
            was_sign = true;
            StartCoroutine("scene_sign");
        }
    }
}
```

コード 4.4.4.1 UI.cs

4.4.5 敗北判定設計

「！」が表示されたらクリックし、判定時間より遅くクリックすると負けとなり、図Xのリトライ画面へと移動する。



図 4.4.5.1 敗北画面

```
:  
IEnumerator cpu_sign(){  
    yield return new WaitForSeconds(0.4f);  
    if(test_cpu != null && was_sign == false){  
        test_cpu.transform.position = new Vector3(-4f,-1f,0f);  
        is_sign = false;  
        Destroy(test_Character.gameObject);  
        was_sign = true;  
        StartCoroutine("restart_sign");  
    }  
}  
:
```

コード 4.4.5.1 UI.cs

4.5 テトリス

4.5.1 ゲーム概要

7種類のブロックがフィールド上部からランダムに1つずつ落下して、フィールドに敷き詰めると揃った横一列を消すことが出来るゲームである。消した段数に応じてスコアが上がっていき、スコアが一定値に達するごとにスピードが上がる。既定のスコアに達することでゲームクリアとなる。

プレイ画面では消したライン数を表示して、現在のレベル、スコアを表示している。次に落ちてくるブロックを表示させ、ブロックが落下してくる位置を表示させるブロックを表示している。

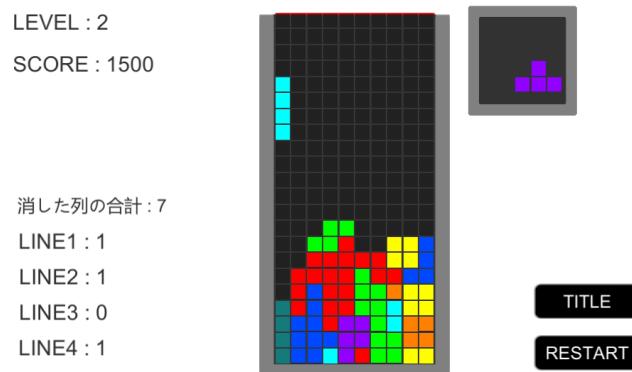


図 4.5.1.1 プレイ画面

ブロックが詰めなくなり、フィールド外に出ることでゲームオーバーになる。



図 4.5.1.2 ゲームオーバー画面

4.5.2 メニュー設計

タイトル画面からホームに戻るボタン、ゲームの遊び方、ゲームスタートを選択することができる。



図 4.5.2.1 タイトル画面

遊び方、プレイ画面からタイトル画面に戻るボタンを設置している。

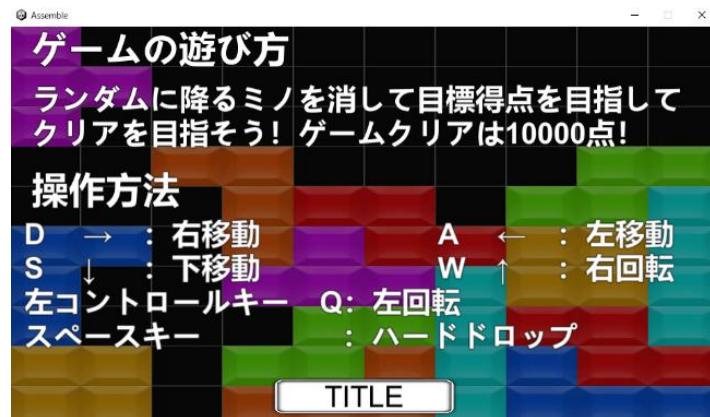


図 4.5.2.2 遊び方画面

4.5.3 操作設計

4.5.3.1 左右移動

「A」キーで左移動、「D」キーで右移動することができる。もしくは「カーソル左」キーで左移動、「カーソル右」キーで右移動することができる。

```
:  
//右移動  
if(Input.GetKeyDown(KeyCode.D) &&  
(Time.time > nextKeyLeftRightTimer)  
|| Input.GetKeyDown(KeyCode.RightArrow))  
{  
    activeBlock.MoveRight(); //右に動かす  
:  
:
```

コード 4.5.3.1.1 GameManager1.cs

```
:  
//左に移動させる  
else if(Input.GetKeyDown(KeyCode.A) && (Time.time > nextKeyLeftRightTimer)  
|| Input.GetKeyDown(KeyCode.LeftArrow))  
{  
    activeBlock.MoveLeft(); //左に動かす  
:  
:
```

コード 4.5.3.1.2 GameManager1.cs

`transform.position` はオブジェクトの位置と回転とサイズを保持しているクラスである。`Vector3` は 3D の位置や方向を表すために使用する構造体である、X、Y、Z で構成されている。例えば、`MoveRight()` では X 軸方向に 1 移動している。

```
//移動用
void Move(Vector3 moveDirection)
{
    transform.position += moveDirection;
}

//移動用関数
public void MoveRight()
{
    Move(new Vector3(1, 0, 0));
}
public void MoveLeft()
{
    Move(new Vector3(-1, 0, 0));
}
public void MoveUp()
{
    Move(new Vector3(0, 1, 0));
}
public void MoveDown()
{
    Move(new Vector3(0, -1, 0));
}
```

コード 4.5.3.1.3 Block.cs

4.5.3.2 回転

「W」キーで右回転をすることができる。もしくは、「カーソル上」キーで右回転することができる。「左コントロール」キー、「Q」キーで左回転をすることができる。

```
:  
//右回転  
if(Input.GetKey(KeyCode.UpArrow) && (Time.time > nextKeyRotateTimer))  
{  
    activeBlock.RotateRight();  
    :  
}
```

コード 4.5.3.2.1 GameManager1.cs

```
:  
//左回転  
else if(Input.GetKey(KeyCode.Q) && (Time.time > nextKeyRotateTimer))  
{  
    activeBlock.RotateLeft();  
    :  
}
```

コード 4.5.3.2.2 GameManager1.cs

`canRotate` で回転していいかどうかを判定している。`RotateAround` は指定した座標を中心におbjectを回転させる。

```
//回転していいブロックかどうか
[SerializeField]
private bool canRotate = true;
public Vector3 rotatepoint;

:

//回転用関数
public void RotateRight()
{
    if(canRotate)
    {
        transform.RotateAround(transform.TransformPoint(rotatepoint),
        new Vector3(0, 0, 1), -90);
    }
}

public void RotateLeft()
{
    if(canRotate)
    {
        transform.RotateAround(transform.TransformPoint(rotatepoint),
        new Vector3(0, 0, 1), 90);
    }
}
```

コード 4.5.3.2.3 Block.cs

4.5.3.3 下移動、自動落下

「S」キーで下移動できる。もしくは、「カーソル下」キーで下移動することができる。
自動落下は時間経過によって落下速度を決めている。

```
:  
//下移動  
else if(Input.GetKey(KeyCode.DownArrow)&&  
(Time.time > nextKeyDownTimer) || (Time.time > nextdropTimer))  
{  
    activeBlock.MoveDown(); //下に動かす  
  
    :  
  
    nextdropTimer = Time.time + dropinterval; //自動落下  
  
    :  
}
```

コード 4.5.3.3.1 GameManager1.cs

nextdropTimer は次にドロップする時間を表している。
Time.time はゲーム開始時から現在までの経過時間(秒)を表している。
dropinterval はドロップする間隔を表している。dropinterval を変更することで自動落下
スピードを調整することができる。

4.5.3.4 即時落下

「スペース」キーでブロックを即時落下させるドロップすることができる。whileでブロックがあるところまで下に移動するようになっている。

```
:  
//即時落下  
else if(Input.GetKeyDown(KeyCode.Space))  
{  
    while(board.CheckPosition(activeBlock))  
    {  
        activeBlock.MoveDown(); //下に動かす  
    }  
:  
:
```

コード 4.5.3.4.1 GameManager1.cs

4.5.4 ブロックの生成設計

テトリスのブロック設計は Hierarchy ビューのプラスから 2D Object→Sprites→Square を選択する。

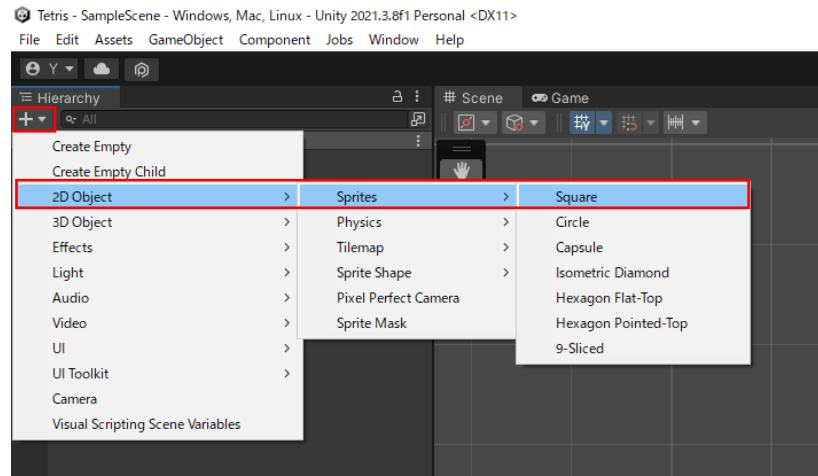


図 4.5.4.1 ブロック作成画面 1

4 つの Square を作成して、使用するブロックの形にする。4 つのブロックを選択して、Scale を 0.95 に設定する。Color を変更することで色を変更することができる。

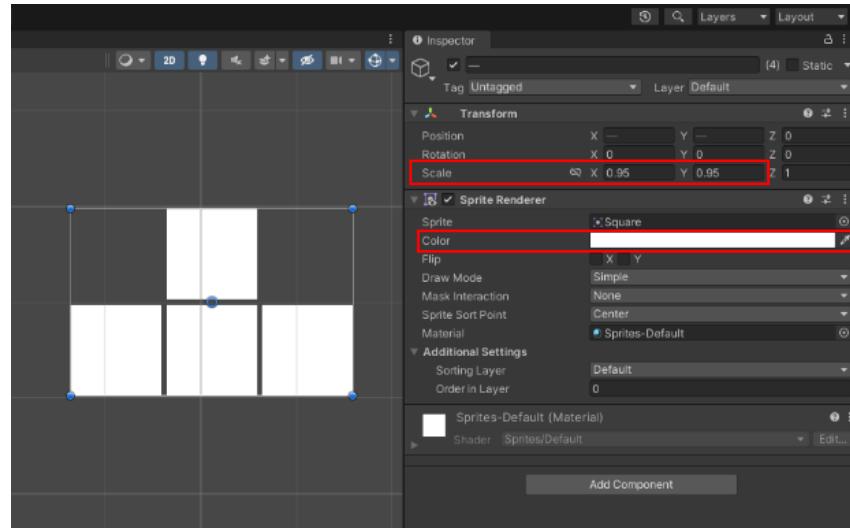


図 4.5.4.2 ブロック作成画面 2

Hierarchy ビューのプラスから Create Empty を選択して、空のオブジェクトを生成する。

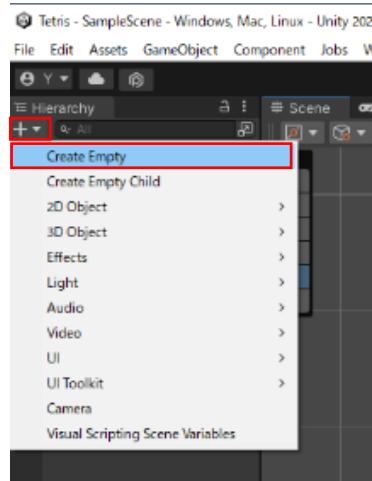


図 4.5.4.3 ブロック作成画面 3

空のオブジェクトに 4 つの Square をアタッチしたらブロックが完成する。

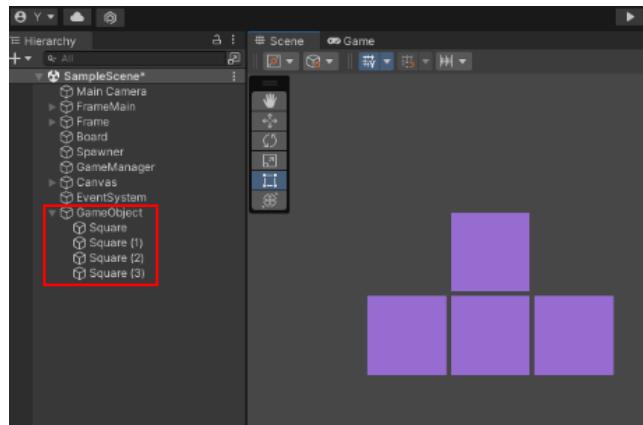


図 4.5.4.4 ブロック作成画面 4

他の形も同様に作成する。

作成した block の中からランダムに 1 つ選んでいる。選ばれたブロックが SpawnBlock() によって生成されている。Instantiate でブロックを生成することができる。Quaternion.identity は回転しないものを生成している。

```
:  
//ランダムなブロックを一つ選ぶ関数  
Block GetRandomBlock()  
{  
    int i = Random.Range(0, block.Length); //ランダムに一つ選ぶ  
    if(block[i])  
    {  
        return block[i];  
    }  
    else  
    {  
        return null;  
    }  
}  
//選ばれたブロックを生成する関数  
public Block SpawnBlock()  
{  
    Block block = Instantiate(GetRandomBlock(),  
    transform.position, Quaternion.identity);  
    if(block)  
    {  
        return block;  
    }  
}
```

:

コード 4.5.4.1 Spawner.cs

4.5.5 ブロックの削除設計

全ての行をチェックしていく、揃っていた場合その行を削除して上の一 段を下げるよう にしている。

```
:
//全ての行をチェックして埋まっているれば削除する関数
public void ClearAllRows()
{
    for(int y = 0; y < height; y++)
    {
        if(IsComplete(y))      //全ての行をチェックする
        {
            ClearRow(y);      //削除する
            ShiftRowsDown(y + 1); //上にあるブロックを一段下げる
            y--;
        }
    }
}
```

コード 4.5.5.1 Board.cs

4.5.6 ゴーストブロックの設計

4.5.6.1 ゴーストブロックとは

ゴーストブロックとは、落下してくるブロックの落下地点を表示させるブロックのことである。落ちてくるブロックが地面につくと、新しく生成されたブロックに対してゴーストブロックが作成される。

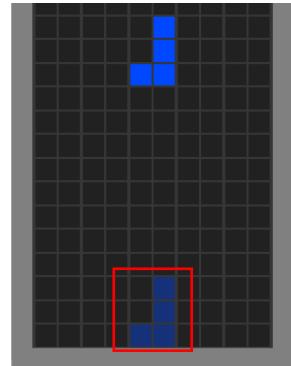


図 4.5.6.1.1 落下地点を表示させるブロック

4.5.6.2 ゴーストブロックの作成

落ちてくるブロックと同じブロックを作成して、ブロックの下に配置している。ブロックの色を同じにして、ブロックの透明度を変更している。大きさを落ちてくるブロックと同じにしている。

```
//ゴーストブロックの生成
public void CreateGhostBlocks()
{
    ghostBlocks = Instantiate(this.activeBlock);
    ghostBlocks.transform.position = this.activeBlock.transform.position;
    Destroy(ghostBlocks.GetComponent<GameManager>());
    foreach(Transform item in ghostBlocks.transform)
    {
        item.GetComponent<SpriteRenderer>().color -= new Color(0, 0, 0, 0.6f);
        item.transform.localScale = new Vector3(1, 1, 1);
    }
}
```

コード 4.5.6.2.1 GameManager1.cs

4.5.6.3 ゴーストブロックの移動

ブロックの移動方法は、落ちてくるブロックと同じ動きをする。

```
void MoveGhostBlocks()
{
    ghostBlocks.transform.position = this.activeBlock.transform.position;
    ghostBlocks.transform.rotation = this.activeBlock.transform.rotation;

    while (board.CheckPosition(ghostBlocks))
    {
        ghostBlocks.gameObject.transform.position += new Vector3(0, -1, 0);
    }
    if(!board.CheckPosition(ghostBlocks))
    {
        ghostBlocks.gameObject.transform.position += new Vector3(0, 1, 0);
    }
}
```

コード 4.5.6.3.1 GameManager1.cs

4.5.7 スコア設計

消した列によってスコアを加算する。num が消した列の数である。

1 列消した場合は 200 点、2 列消した場合は 400 点、3 列消した場合は 600 点、4 列消した場合は 900 点となっている。

```
        :  
public void AddScore(int num)  
{  
    if(num == 1)           //1 列消した場合  
    {  
        score += 200;  
        :  
    }  
    else if(num == 2)      //2 列消した場合  
    {  
        score += 400;  
        :  
    }  
    else if(num == 3)      //3 列消した場合  
    {  
        score += 600;  
        :  
    }  
    else if(num == 4)      //4 列消した場合  
    {  
        score += 900;  
        :  
    }  
    scoreText.text = "SCORE :" + score.ToString(); //スコアの合計を表示  
    :  
}
```

コード 4.5.7.1 Board.cs

4.5.8 ゲームクリア設計

ゲームをクリアするには既定のスコアを達することでゲームがクリアとなる。

既定のスコア以上になった場合、ゲームクリア画面になる。

```
public void AddScore(int num)
{
    int clearScore = 10000; //目標点
    :

    if(score >= clearScore)
    {
        GameClear();
    }
}

//ゲームクリア

public void GameClear()
{
    SceneManager.LoadScene("GameClearScene");
}
```

コード 4.5.8.1 Board.cs



図 4.5.8.1 ゲームクリア画面

4.6 横スクロールアクションゲーム 1

4.6.1 ゲーム概要

自キャラクターを操作してゴールを目指すゲームである。難易度を選択するとプレイ画面に移動する。プレイ画面ではタイトル画面に戻れるボタンを設置している。



図 4.6.1.1 プレイ画面

ゲームをクリアすると次のステージに進むボタンが出現する。



図 4.6.1.2 クリア画面

ゲームオーバーになると次のステージに進むボタンを押せず、リスタートボタンでリスタートすることができる。次のステージに進むにはステージをクリアする必要がある。



図 4.6.1.3 ゲームクリア画面

4.6.2 メニュー説明

タイトル画面からホームに戻るボタン、ゲームの遊び方、難易度を選択することができる。難易度は「EASY」、「NORMAL」、「HARD」を選択することができる。

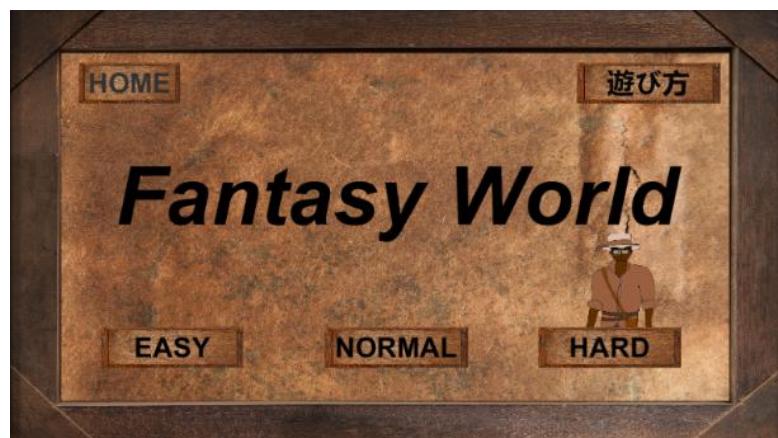


図 4.6.2.1 タイトル画面

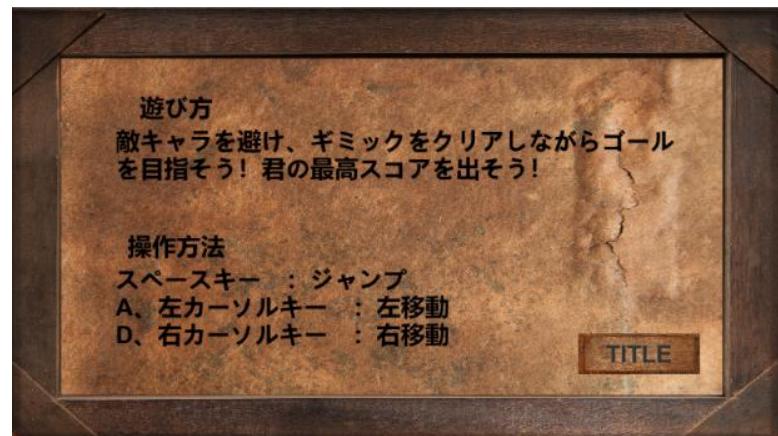


図 4.6.2.2 遊び方画面

4.6.3 自キャラクター設計

4.6.3.1 当たり判定

Capsule Collider 2D を追加することで、自キャラクターの当たり判定を設定している。Capsule Collider 2D は底辺が丸いので物理的な力を加えると転がってしまうので、Freeze Rotation Z にチェックを付けることでそれが予防される。Material に Physics Material 2D を追加する。下記では NPM2D_1 と記述されている。Physics Material 2D は 2D 物理オブジェクト間の衝突時の摩擦や弾性を調整している。Friction はコライダーの摩擦係数、Bounciness は衝突が表面から弾むときの強さを表している。

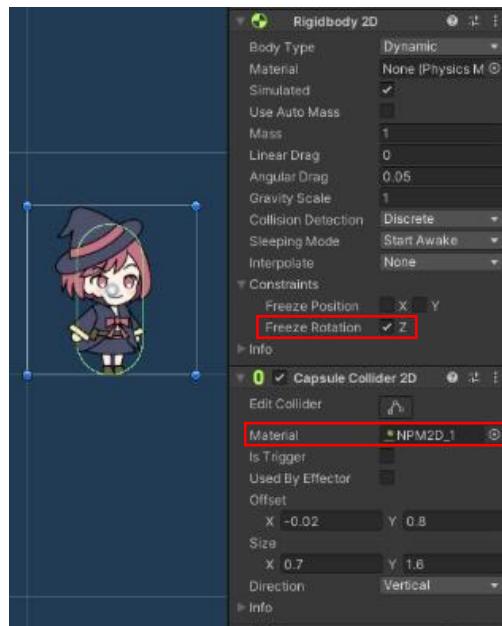


図 4.6.3.1.1 プレイヤー設計画面

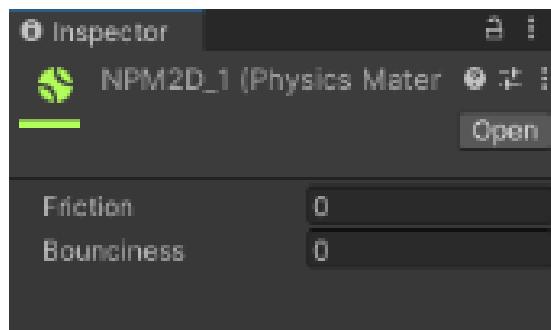


図 4.6.3.1.2 Physics Material 2D

4.6.3.2 操作設計

自キャラクターの向きによって移動させる。引数に Horizontal を使用することでキーボードの「右カーソル」キー、「D」キーで右移動に、「左カーソル」キー、「A」キーを左移動に対応してくれる。

```
        :  
void Update()  
{  
    :  
    //水平方向の入力をチェックする  
    axisH = Input.GetAxisRaw("Horizontal");  
    //向きの調整  
    if(axisH > 0.0f)  
    {  
        //右移動  
        transform.localScale = new Vector2(1, 1);  
    }  
    else if(axisH < 0.0f)  
    {  
        //左移動  
        transform.localScale = new Vector2(-1, 1); //左右反転させる  
    }  
    :  
}
```

コード 4.6.3.2.1 PlayerController.cs

4.6.3.3 アニメーション作成

自キャラクターのアニメーションを作成する。移動の画像を全て選択して Scene ビューにドラッグ&ドロップする。配置するとアニメーションクリップでファイルの名前と保存先を指定するダイアログが開くので、指定してファイルを保存する。

「Window」メニューから「Animation」→「Animation」を選択し、三角形のボタンをクリックすると、右側のビューに登録されているスプライトが時間軸に沿って表示される。表示スケールを変更して、見やすいうように設定する。

「Sample」という項目で 1 秒間に何コマ表示するかを決めることができますので見やすくなるように設定する。

ジャンプ、ゲームクリア、ゲームオーバーのアニメーションも同じように作成する。

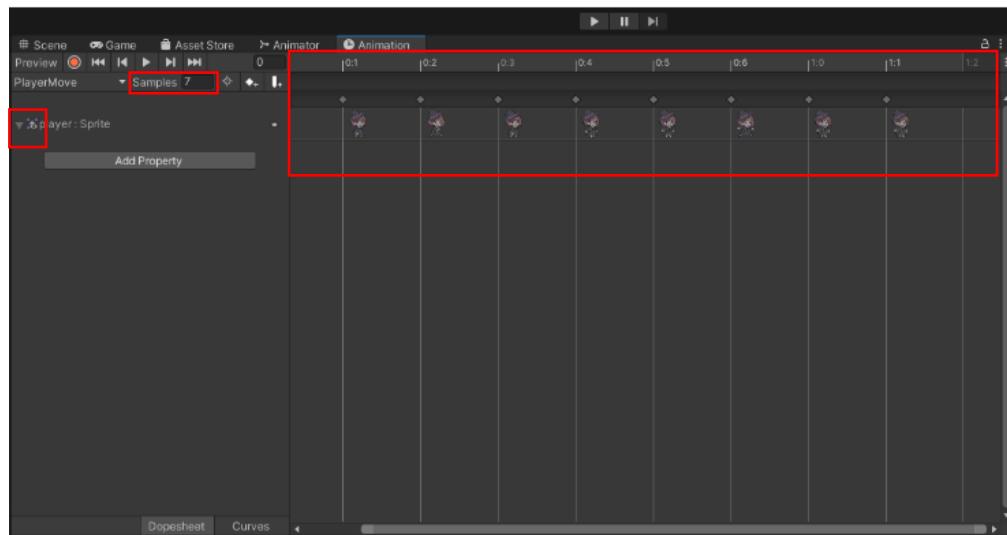


図 4.6.3.3.1 アニメーション作成画面

アニメーションさせるためにスクリプトを変更する。onGround は地面に立っているフラグである。FixedUpdate()でアニメーションを再生してくれる。

```
:  
void FixedUpdate()  
{  
    :  
    if(onGround)  
    {  
        //地面の上  
        if(axisH == 0)  
        {  
            nowAnime = stopAnime; //停止中  
        }  
        else  
        {  
            nowAnime = moveAnime; //移動  
        }  
    }  
    else  
    {  
        //空中  
        nowAnime = jumpAnime;  
    }  
  
    if(nowAnime != oldAnime)  
    {  
        oldAnime = nowAnime;  
        animator.Play(nowAnime); //アニメーション再生  
    }  
}  
:  
:
```

コード 4.6.3.3.1 PlayerController.cs

4.6.4 地面設計

ゲームステージで使われる当たり判定は Box Collider 2D コンポーネントをアタッチする。Inspector ビューの右上にある Layer で、新しいレイヤーを追加する。新しく作成したレイヤーを追加する。

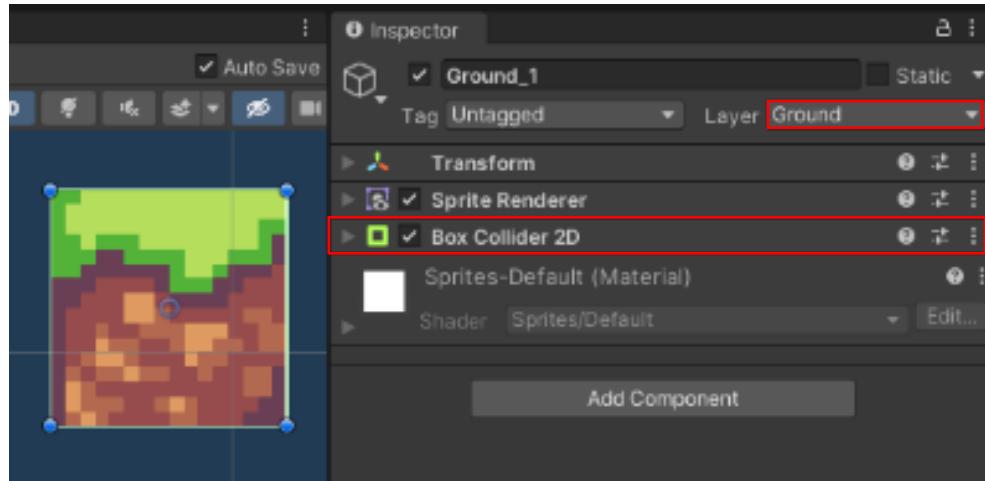


図 4.6.4.1 地面作成画面

4.6.5 ゲームクリア設計

ゴールに接触したことを判定するために Box Collider 2D を追加して Is Trigger にチェックをつける。Inspector ビューの左上にある「Tag」から新しいタグを追加する。作成したタグをゴールに追加する。

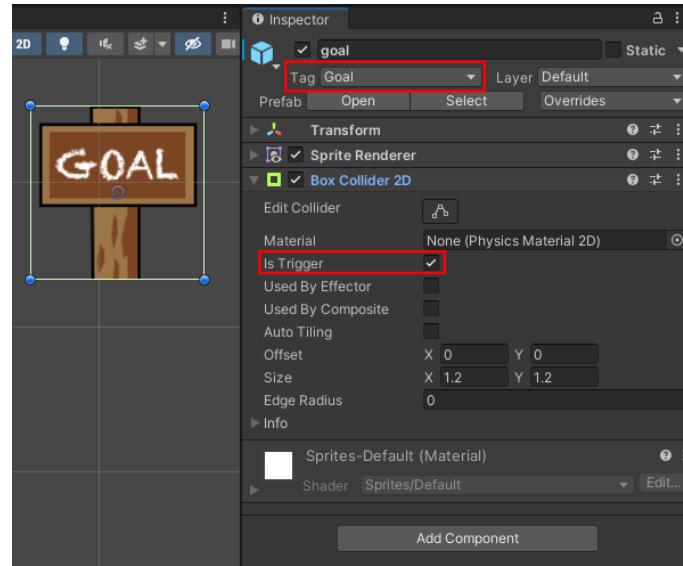


図 4.6.5.1 ゴール設計画面

ゴールのオブジェクトに自キャラクターが触れると gameState がゲームクリア状態になる。ゲームをクリアすると次のステージに進むボタンが押せるようになり、リスタートボタンが押せなくなる。

```
:  
if(PlayerController.gameState == "gameclear")  
{  
    //ゲームクリア  
    mainImage.SetActive(true); // 画像を表示する  
    panel.SetActive(true); // ボタン（パネル）を表示する  
    // RESTART ボタンを無効化する  
    Button bt = restartButton.GetComponent<Button>();  
    bt.interactable = false;  
    // 画像を表示する  
    mainImage.GetComponent<Image>().sprite = gameClearSpr;  
    PlayerController.gameState = "gameend";  
:  
:
```

コード 4.6.5.1 GameManager.cs

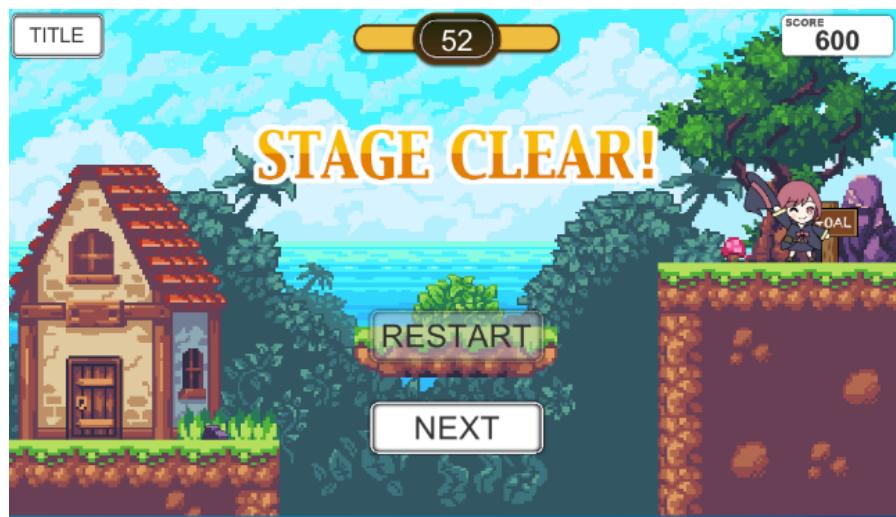


図 4.6.5.2 クリア画面

4.6.6 ゲームオーバー設計

タグで指定したものに接触した場合ゲームオーバーになる。今回は「Dead」を指定している。敵に触れる、穴に落ちる、制限時間が 0になるとゲームオーバーになる。ゲームオーバーになると次のステージに進むボタンが押せず、リスタートボタンしか押せない状態になる。

```
:  
else if(PlayerController.gameState == "gameover")  
{  
    //ゲームオーバー  
    mainImage.SetActive(true); // 画像を表示する  
    panel.SetActive(true); // ボタン（パネル）を表示する  
    // NEXT ボタンを無効化する  
    Button bt = nextButton.GetComponent<Button>();  
    bt.interactable = false;  
    // 画像を表示する  
    mainImage.GetComponent<Image>().sprite = gameOverSpr;  
    PlayerController.gameState = "gameend";  
    :  
}
```

コード 4.6.6.1 GameManager.cs

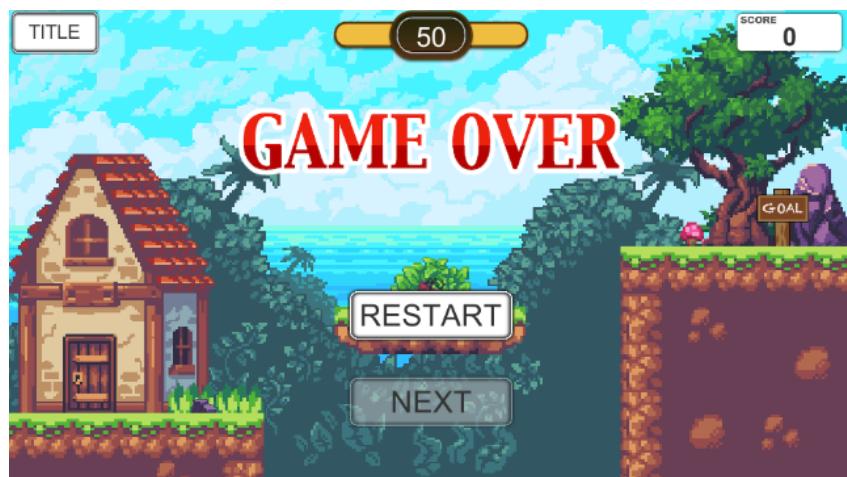


図 4.6.6.1 ゲームクリア画面

4.6.7 カメラ設計

自キャラクターの動きに応じてカメラが動くように設定する。自キャラクターに設定したタグを探して自キャラクターを探している。ステージには強制スクロールするマップを作成したため、時間制御によって強制スクロールする。forceScrollSpeedX は 1 秒間で動かす X 距離を表している。

```
:  
void Update()  
{  
    //プレイヤーを探す  
    GameObject player = GameObject.FindGameObjectWithTag("Player");  
    if(player != null)  
    {  
        // カメラの更新座標  
        float x = player.transform.position.x;  
        float y = player.transform.position.y;  
        float z = transform.position.z;  
        // 横同期させる  
        if(isForceScrollX)  
        {  
            // 強制スクロール  
            x = transform.position.x + (forceScrollSpeedX * Time.deltaTime);  
        }  
        // 両端に移動制限を付ける  
        if(x < leftLimit)  
        {  
            x = leftLimit;  
        }  
        else if(x > rightLimit)  
        {  
            x = rightLimit;  
        }  
    }  
}:
```

コード 4.6.7.1 CameraManager.cs

画面両端から自キャラクターが落ちないように、透明な壁を作成する。空のゲームオブジェクトを作成して、Box Collider 2D を 2 つアタッチする。画面両端に設置して自キャラクターが落ちないようにする。

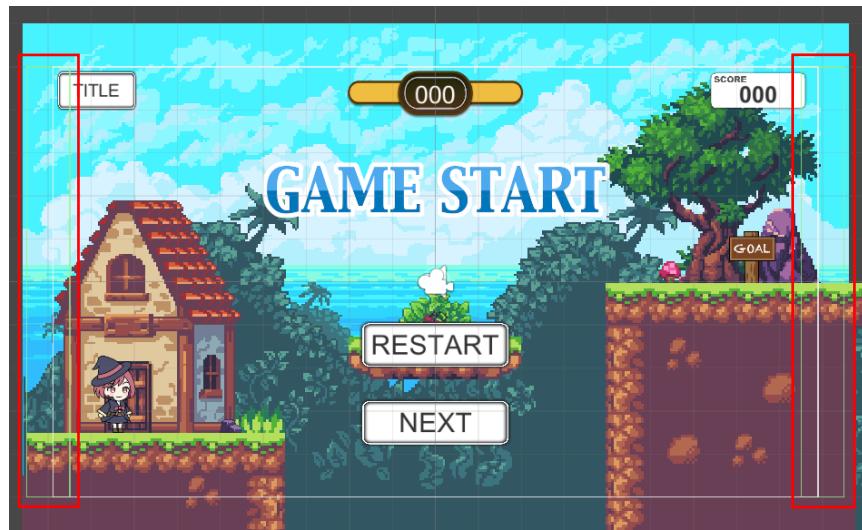


図 4.6.7.1 当たり判定

穴に落ちるとゲームオーバーになるように床下に触れたらゲームオーバーになるように設置する。新しいタグを作成してタグを設定する。Box Collider 2D で触れたらゲームオーバーにするために Is Trigger にチェックを入れる。

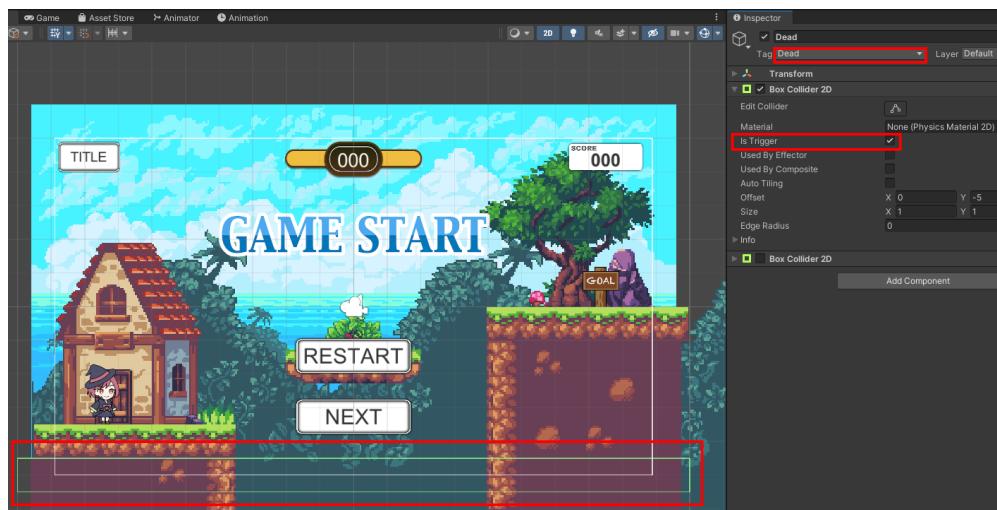


図 4.6.7.2 当たり判定

4.6.8 敵キャラクター設計

4.6.8.1 敵キャラクターの説明

敵キャラクターは、一定の範囲内を行ったり来たりする。スピード、距離を変更することができる。敵キャラクターはアセットストアでダウンロードしたものを使用している。使用したアセットストア、敵キャラクターは以下の通りである。

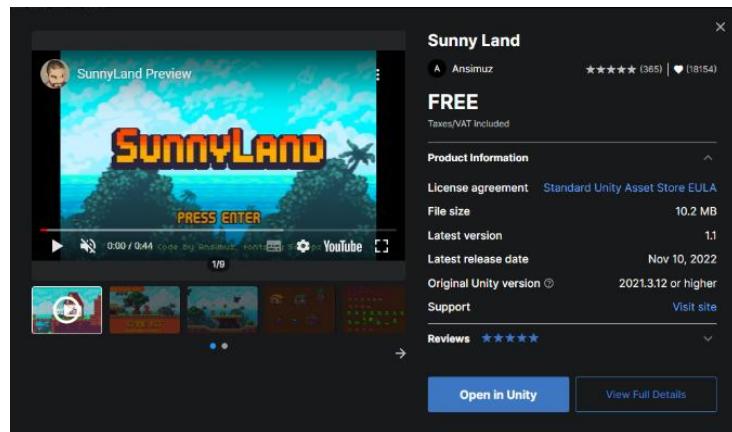


図 4.6.8.1.1 使用したアセット (Sunny Land)

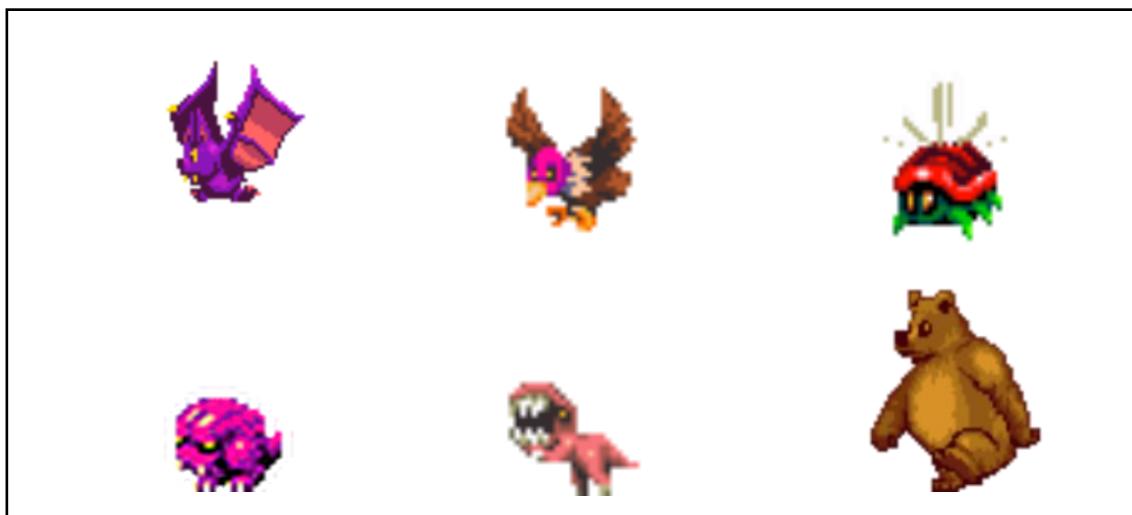


図 4.6.8.1.2 敵キャラクター一覧

4.6.8.2 敵キャラクターの当たり判定

Circle Collider 2D は地面との物理的な当たりを担当して、Box Collider 2D は自キャラクターをゲームオーバーにするイベント当たりを担当している。

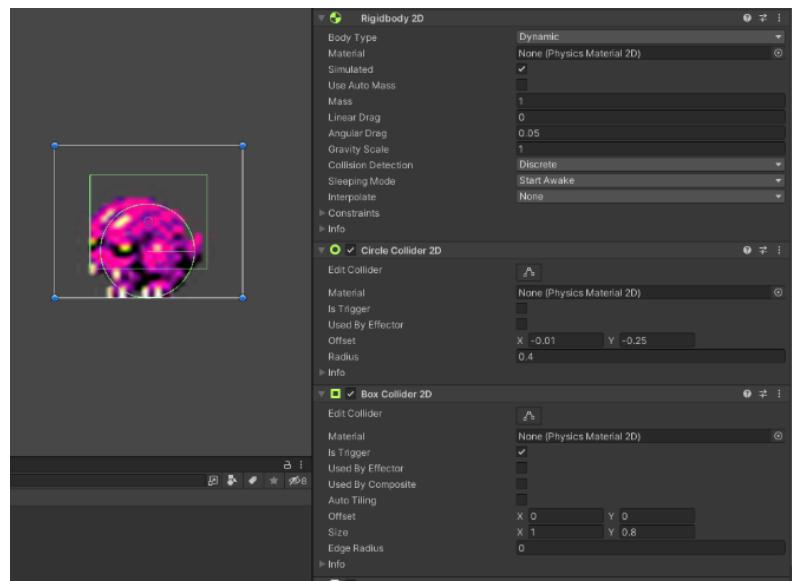


図 4.6.8.2.1 敵キャラクター当たり判定

4.6.8.3 敵キャラクターの移動

Speed は敵キャラクターの速度で、 rbody は敵キャラクターが移動する間隔である。 Rigidbody 2D コンポーネントを取得して、速度を更新し、敵キャラクターを移動させている。

```
:  
void FixedUpdate()  
{  
    // 速度を更新する  
    // Rigidbody2D を取ってくる  
    Rigidbody2D rbody = GetComponent<Rigidbody2D>();  
    if(direccion == "right")  
    {  
        rbody.velocity = new Vector2(speed, rbody.velocity.y);  
    }  
    else  
    {  
        rbody.velocity = new Vector2(-speed, rbody.velocity.y);  
    }  
}
```

コード 4.6.8.3.1 EnemyController1.cs

4.6.8.4 アニメーション作成

アニメーション作成は自キャラクター設計と同様に作成する。

4.6.9 ステージ設計

4.6.9.1 時間のカウントダウン

ステージによって時間を変更できるようにして、時間が減少していくように作成する。

```
public bool isCountDown = true;           // true = 時間をカウントダウン計測する
public float gameTime = 0;                // ゲームの最大時間
public bool isTimeOver = false;            // true = タイマー停止
public float displayTime = 0;              // 表示時間
float times = 0;                         // 現在の時間
:
void Update()
{
    if(isTimeOver == false)
    {
        times += Time.deltaTime;
        if(isCountDown)
        {
            // カウントダウン
            displayTime = gameTime -times;
            if(displayTime <= 0.0f)
            {
                displayTime = 0.0f;
                isTimeOver = true;
            }
        }
    }
}
```

コード 4.6.9.1.1 TimerController.cs

4.6.9.2 ステージアイテム

「Tag」で「ScoreItem」を作成して、設定する。Circle Collider 2D コンポーネントをアタッチして、「Is Trigger」をオンにする。

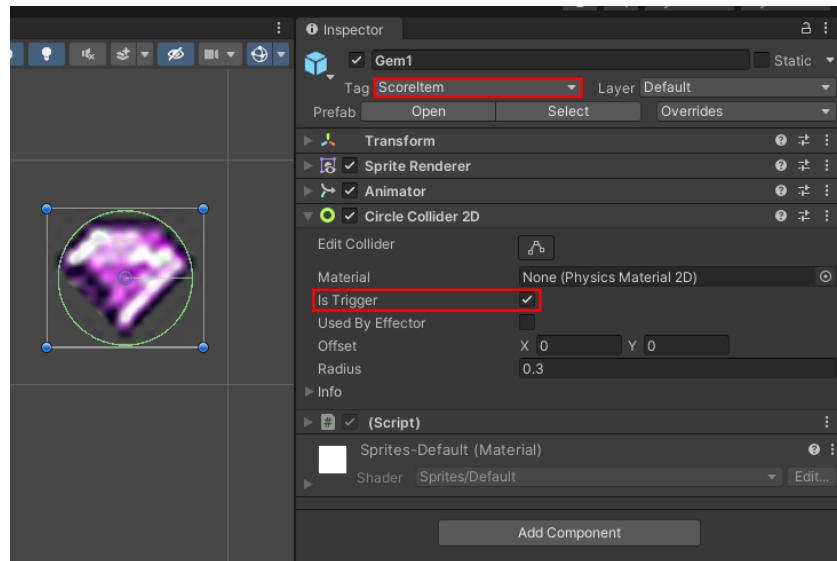


図 4.6.9.2.1 アイテム作成

自キャラクターがアイテムを取った場合、それがどういうアイテムなのかを判断するためのスクリプトを作成する。アイテムを取った時のスコアを設定することができる。アイテムになるゲームオブジェクトにアタッチする。

```
public int value = 0;      // 整数値を設定できる
```

コード 4.6.9.2.1 ItemData.cs

アイテムを取得するスクリプトを作成する。獲得したスコアを記録しておく score 変数を追加する。アイテムを触れたらスコアを取得してアイテムを削除している。

```
:  
else if(collision.gameObject.tag == "ScoreItem")      // スコアアイテムか  
{  
    // スコアアイテム  
    // ItemData を得る  
    ItemData item = collision.gameObject.GetComponent<ItemData>();  
    // スコアを得る  
    score = item.value;  
    // アイテムを削除する  
    Destroy(collision.gameObject);  
}  
:  
:
```

コード 4.6.9.2.2 GameManager.cs

4.6.9.3 ダメージブロック

Box Collider 2D をアタッチして、当たり判定を付ける。Is Trigger をオンにする。タグをゲームオーバーと同じものを設定する。

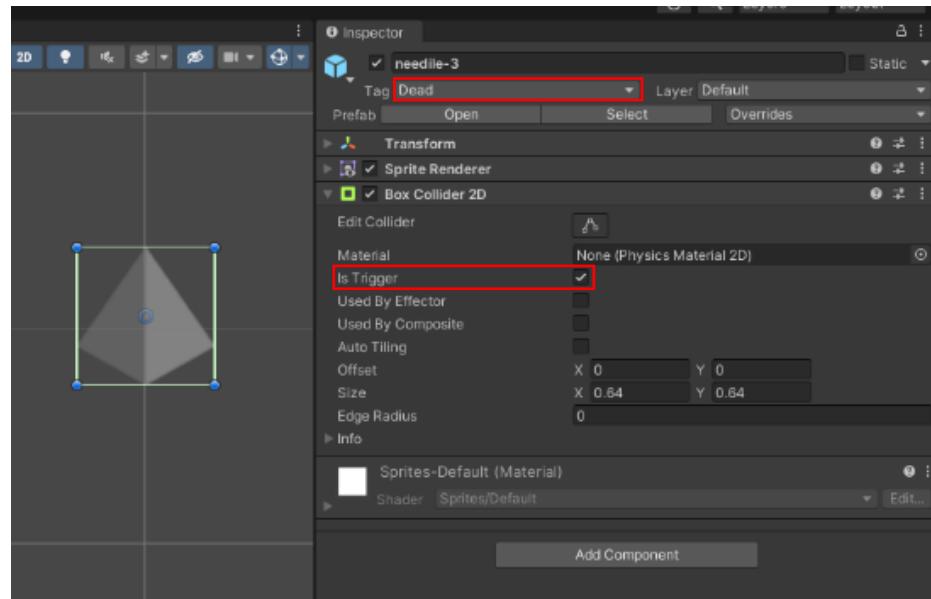


図 4.6.9.3.1 ダメージブロック設計画面

4.6.9.4 落下するブロック

自キャラクターが乗れるようにするために Layer を地面と同じものを設置する。Box Collider 2D、Circle Collider 2D、Rigidbody 2D をアタッチする。Box Collider 2D は自キャラクターが上に乗るための当たり判定、Circle Collider 2D は地面と接触するための当たり判定になる。Rigidbody 2D をアタッチすることで押すことができ、落下させることができる。

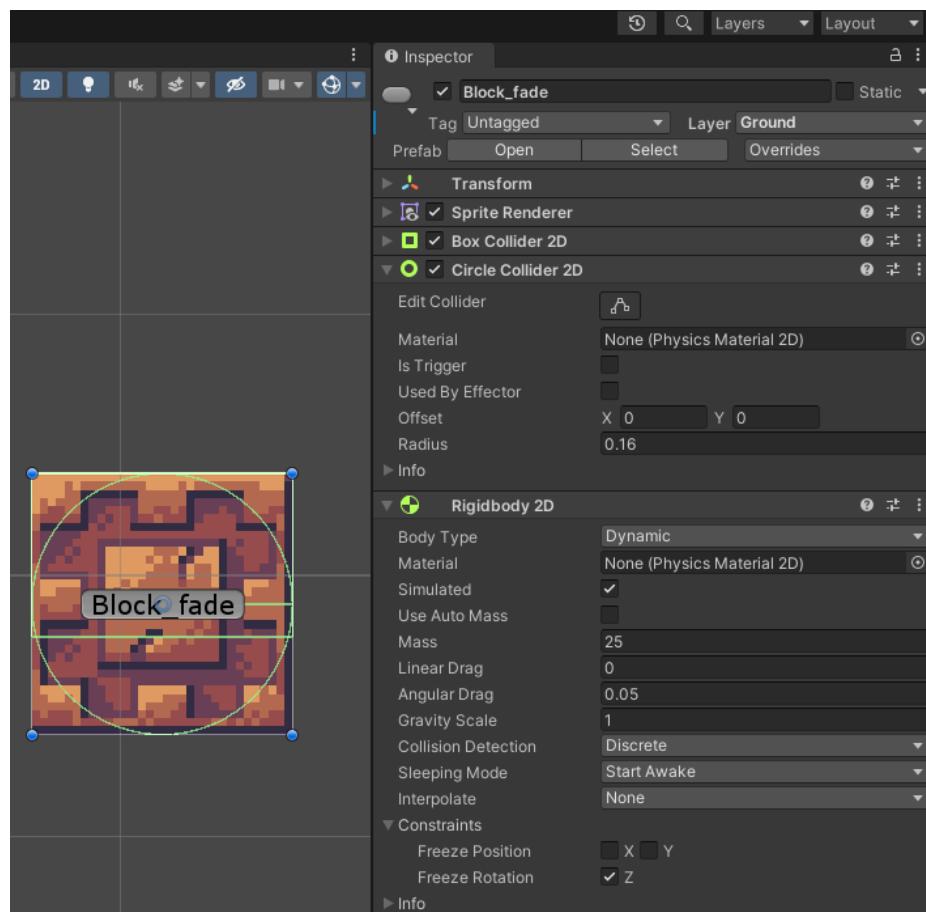


図 4.6.9.4.1 仕掛けブロック作成画面

落下させるためのスクリプトを作成する。落下するブロックは自キャラクターが一定の距離に接近することで落下し、落下後は消えるようになっている。RigidbodyType2D.Dynamicで物理シミュレーションを有効化している。有效地にすることでブロックは重力落下したり、押して動かすことができる。

```
:  
// Rigidbody2D の物理挙動を開始  
rbody.bodyType = RigidbodyType2D.Dynamic;  
:  
// 落下した  
// 透明値を変更してフェードアウトさせる  
fadeTime -= Time.deltaTime; // 前フレームの差分秒マイナス  
// カラーを取り出す  
Color col = GetComponent<SpriteRenderer>().color;  
col.a = fadeTime; // 透明値を変更  
GetComponent<SpriteRenderer>().color = col; // カラーを再設定する  
if(fadeTime <= 0.0f)  
{  
    // 0 以下（透明）になったら消す  
    Destroy(gameObject);  
    :  
}
```

コード 4.6.9.4.1 GimmickBlock.cs

4.6.10 スコア設計

自キャラクターがアイテム拾ったときに取得するスコア、残り時間から得られるスコアの2種類ある。

```
:  
void Update()  
{  
    :  
    // 整数に代入することで少数を切り捨てる  
    int time = (int)timeCnt.displayTime;  
    totalScore += time * 10;    // 残り時間をスコアに加える  
    :  
    totalScore += stageScore;  
    stageScore = 0;  
    UpdateScore();  
    :  
    void UpdateScore()  
    {  
        int score = stageScore + totalScore;  
        scoreText.GetComponent<Text>().text = score.ToString();  
    }  
    :  
}
```

図 4.6.10.1 GameManager.cs

4.7 横スクロールアクションゲーム 2

4.7.1 ゲーム設計

白いキューブを操作し、先に進んでいくゲーム、進んでいくと仕掛けが多くなり難しくなっていく。

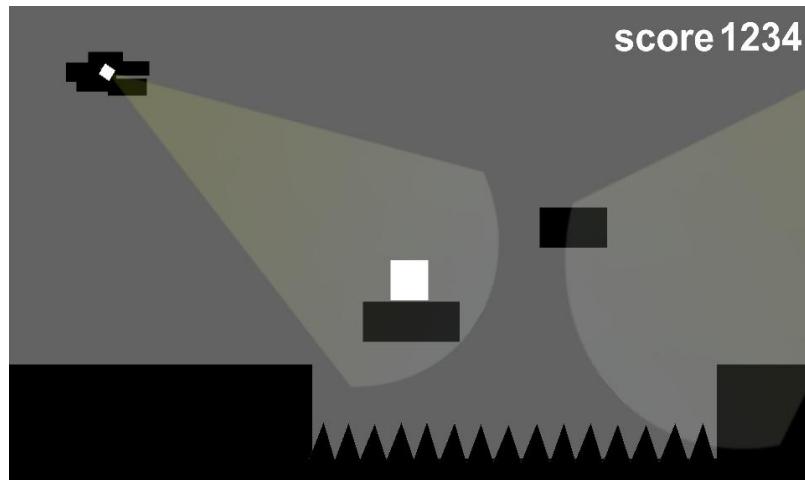


図 4.7.1.1 プレイ画面

敵の攻撃や仕掛けにあたるとゲームオーバーとなる。

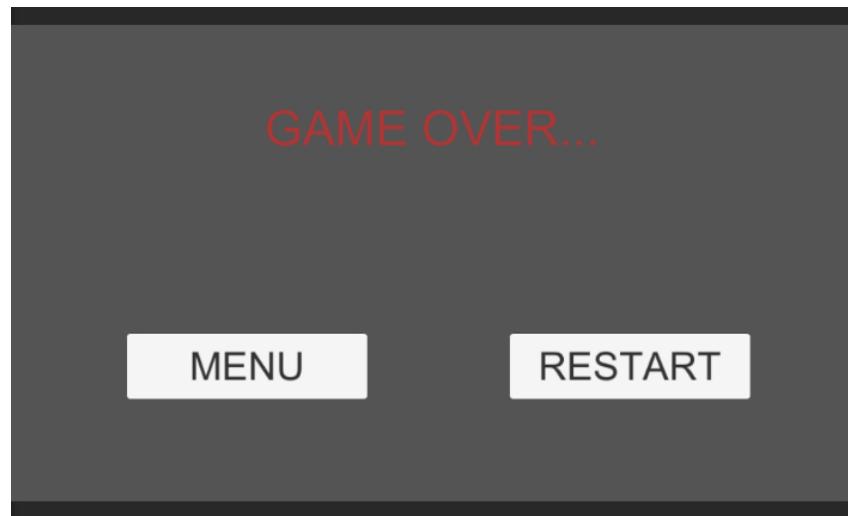


図 4.7.1.2 ゲームオーバー

4.7.2 ゲームの操作説明

「カーソル左」で左移動、「カーソル右」で右移動。
speed、jumpSpeed の値は Unity の Inspector で設定する。
“Horizontal”、“Vertical”にあてるキーは Unity 内で設定する。

```
:  
// X 軸  
float horizontalKey = Input.GetAxis("Horizontal"); //キーの取得  
float xSpeed = 0.0f;  
  
if (horizontalKey > 0) //右移動  
{  
    transform.localScale = new Vector3(1, 1, 1);  
    xSpeed = speed;  
}  
else if (horizontalKey < 0) //左移動  
{  
    transform.localScale = new Vector3(-1, 1, 1);  
    xSpeed = -speed;  
}  
else  
{  
    xSpeed = 0.0f; //操作されないとき、速度 0  
}  
:  
// Y 軸  
float verticalKey = Input.GetAxis("Vertical"); //キーの取得  
:  
//地面についている、かつキー入力でジャンプ。  
if(isGround)  
{  
    if(verticalKey > 0 && isGround)  
    {  
        ySpeed = jumpSpeed;  
        jumpPos = transform.position.y;
```

```

        isJump = true; //ジャンプ判定
        jumpTime = 0.0f;
    }
    else
    {
        isJump = false; //ジャンプ判定
    }
}

//ボタン入力時間によるジャンプ力の調整
else if(isJump)
{
    bool pushUpKey = verticalKey > 0;
    bool canHeight = jumpPos + jumpHeight > transform.position.y;
    bool canTime = jumpLimitTime > jumpTime;
}

//キー入力、高さ上限、ジャンプ時間、上に障害物が無い、の条件をクリア
if(pushUpKey && canHeight && canTime && !isHead)
{
    ySpeed = jumpSpeed;
    jumpTime += Time.deltaTime;
}
else
{
    isJump = false;
    jumpTime = 0.0f;
}
}

:

```

コード 4.7.2.1 Player.cs

4.7.3 仕掛け 1 の設計

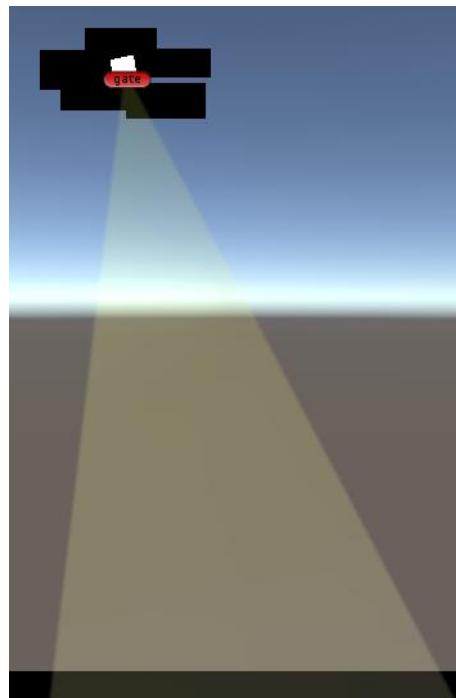


図 4.7.3.1 仕掛け 1

仕掛け 1 はプレイヤーが範囲に入ると、プレイヤーの方向を向いて攻撃してくる。プレイヤーを認識する前は回転している。

プレイヤーを認識する。

```
:  
public bool OnPlayer()  
{  
    // 範囲内に Player がいる  
    if(onPlayerEnter || onPlayerStay)  
    {  
        onPlayer = true;  
    }  
    // 範囲外に Player がいる  
    else if (onPlayerExit)  
    {  
        onPlayer = false;  
    }  
    onPlayerEnter = false;  
    onPlayerStay = false;  
    onPlayerExit = false;  
    return onPlayer;  
}  
:
```

コード 4.7.3.1　GetPlayer.cs

回転し続ける、プレイヤーをとらえるとプレイヤーを見続ける。

```
:  
transform.Rotate(new Vector3(0, 0, 60) * Time.deltaTime, Space.World);  
:  
Vector3 diff = (this.Target.transform.position - this.transform.position).normalized;  
this.transform.rotation = Quaternion.FromToRotation(Vector3.down, diff);  
:
```

コード 4.7.3.2　EnemyRotate.cs

Unity で指定した MovePoint 間を行き来する。

```
:  
// Unity で MovePoint を設定できるようにする  
public EnemyObject[] movePoint;  
:  
if(movePoint != null && movePoint.Length > 0 && rb != null)  
{  
    if(!returnPoint)  
    {  
        int nextPoint = nowPoint + 1;  
  
        if(Vector2.Distance(transform.position,movePoint[nextPoint].transfor  
m.position)> 0.1f)  
        {  
            Vector2 toVector =  
                Vector2.MoveTowards(transform.position,movePoint[nextPoint].  
transform.position,speed * Time.deltaTime);  
            rb.MovePosition(toVector);  
        }  
        else  
        {  
            rb.MovePosition(movePoint[nextPoint].transform.position);  
            ++nowPoint;  
  
            if(nowPoint + 1 >= movePoint.Length)  
            {  
                returnPoint = true;  
            }  
        }  
    }  
    else  
    {  
        int nextPoint = nowPoint - 1;  
  
        if(Vector2.Distance(transform.position,movePoint[nextPoint].transfor  
m.position) > 0.1f)
```

```
{  
    Vector2 toVector =  
        Vector2.MoveTowards(transform.position,movePoint[nextPoint],  
            transform.position,speed * Time.deltaTime);  
    rb.MovePosition(toVector);  
}  
else  
{  
    rb.MovePosition(movePoint[nextPoint].transform.position);  
    --nowPoint;  
  
    if(nowPoint <= 0)  
    {  
        returnPoint = false;  
    }  
}  
}  
}  
;
```

コード 4.7.3.3 EnemyObject.cs

コード 4.7.3.1 より、プレイヤーを認識して攻撃する。

```
:  
if(onPlayer)  
{  
    InvokeRepeating("Shot", gunSta, interval);  
}  
:  
Void Shot()  
{  
    GameObject shell = Instantiate(shellPrefab,  
        transform.position, Quaternion.identity);  
    Rigidbody shellrb = shell.GetComponent<Rigidbody>();  
  
    // 弾速  
    shellrb.AddForce(transform.forward * 500);  
}  
:
```

コード 4.7.3.4 Enemyshot.cs

4.7.4 仕掛け 2 の設計

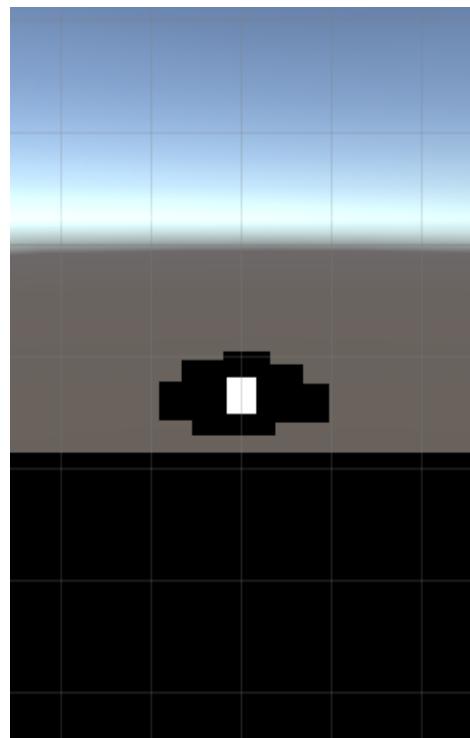


図 4.7.4.1 仕掛け 2

仕掛け 2 はプレイヤーが近づくとプレイヤーに向かって移動し、接触するとゲームオーバーになる。左右にしか動かず、プレイヤーが宙にいれば動かない。

```
// 仕掛け2よりプレイヤーが左にいるとき
;
if(this.Target.transform.position.y - this.transform.position.y < 0)
{
    // 指定のポイントへ
    Vector2 Point = Vector2.MoveTowards(this.transform.position,
    MovePoint1.transform.position, speed);
    rb.MovePosition(Point);
}

// それ以外（プレイヤーが右にいるとき）
else
{
    // 指定のポイントへ
    Vector2 Point = Vector2.MoveTowards(this.transform.position,
    MovePoint2.transform.position, speed);
    rb.MovePosition(Point);
}
```

コード 4.7.4.1 EnemyGo.cs

4.8 メニュー画面

4.8.1 画面設計

ゲームを起動すると、ロゴが表示され、その後にタイトル画面が表示される。タイトル画面の任意の箇所をクリックすると、メニュー画面に移る。



図 4.8.1.1 タイトル画面

メニュー画面のアイコンのボタンを押すと、各ゲームを遊ぶことができる。

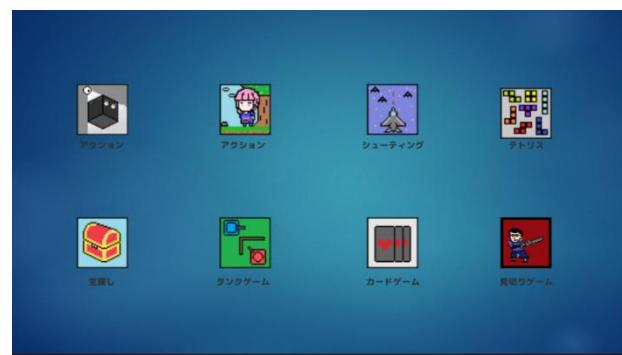


図 4.8.1.2 メニュー画面

各ボタンに以下のような関数をアタッチすることで、それぞれのゲームに画面が移る。
以下のスクリプトはタンクゲームとカードゲーム、それぞれに1秒かけて暗転させながら、遷移するための関数。

```
:  
public void OnTankButton()  
{  
    FadeManager.Instance.LoadScene("StartScene", 1f);  
}  
:
```

コード 4.8.1.1 OnSotsukenTitleController.cs

```
:  
public void OnCardGameButton()  
{  
    FadeManager.Instance.LoadScene("CardTitle", 1f);  
}  
:
```

コード 4.8.1.2 OnSotsukenTitleController.cs

また、上記の関数では、FadeManager という GitHub で naichilab 氏が公開しているソースコードをインポートして使っている。これを使うことで、ゆっくり暗転させながら、シーン移動をすることができる。

Unity-FadeManager

Unity3Dにてフェードイン（アウト）を利用したシーン間遷移を行います。
[README.md](#) | [English](#)

デモ



インストール

1. ダウンロード -> [Download ZIP](#)
2. ダウンロードしたZIPファイルを解凍。
3. *FadeManager.unitypackage* をUnity3Dプロジェクトにインポート

図 4.8.1.3 FadeManager のダウンロード画面 URL: <https://github.com/naichilab/Unity-FadeManager/blob/master/README.ja.md>

4.8.2 メニューと各ゲームの統合方法

各ゲームは別々のパソコンで別のプロジェクト(フォルダ)として制作していたので、それを同一のプロジェクトの「Assemble」にまとめた。

タンクゲームの場合、図 4.8.2.1、図 4.8.2.2 の手順で、プロジェクトウィンドウの「Assets」フォルダに「Tank」というフォルダを作る。

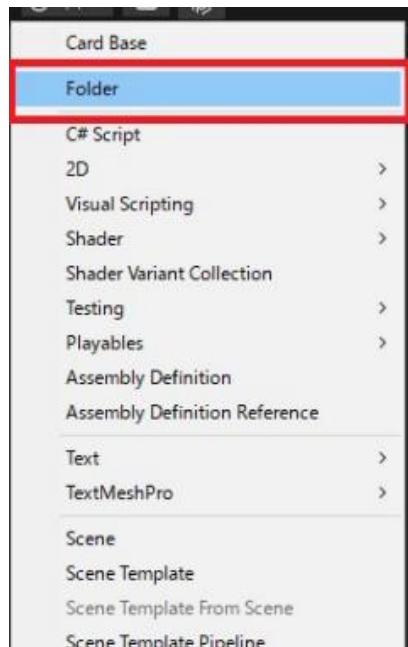


図 4.8.2.1 プロジェクトウィンドウにフォルダを作成している時の画面

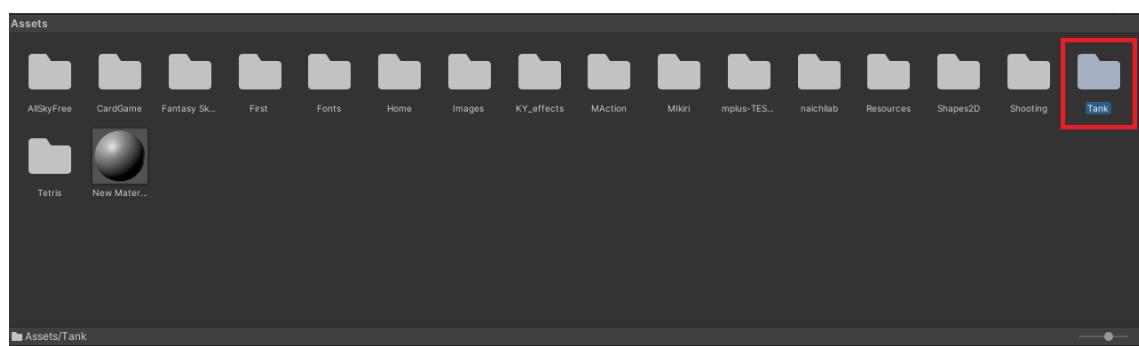


図 4.8.2.2 プロジェクトウィンドウの Assets に作成された Tank フォルダ

別のプロジェクトで作っていた「タンクゲーム」という名前のフォルダの「Assets」フォルダ以下にあるフォルダとファイルを、図 4.8.2.3 のように「Assemble」プロジェクトの「Tank」フォルダにドラッグアンドドロップなどを使ってインポートする。

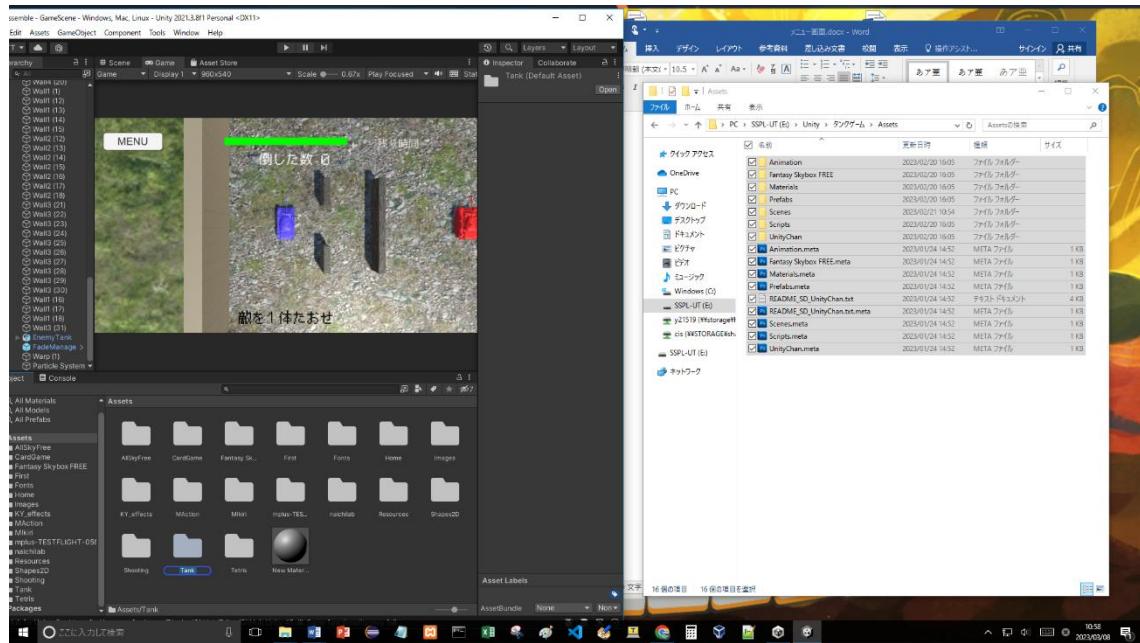


図 4.8.2.3 別のプロジェクトのゲームを Assemble プロジェクトに移動させている画面

次に「File」をクリックし、「Build Settings」をクリックする。

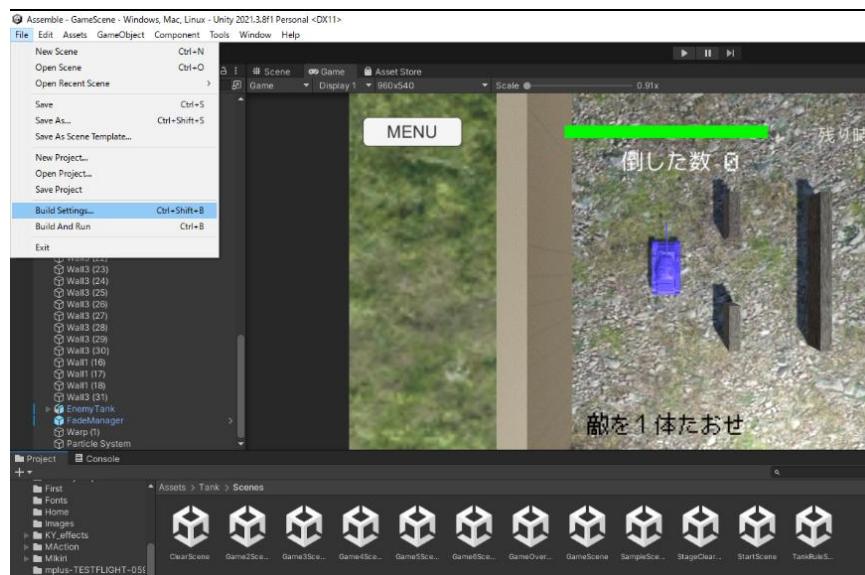


図 4.8.2.4 Build Settings を開く画面

先ほど作成した「Tank」フォルダをクリックして、「Scenes」をクリックする。画面上部の「Scenes in Build」にプロジェクトウィンドウのシーンフォルダをドラッグアンドドロップする。

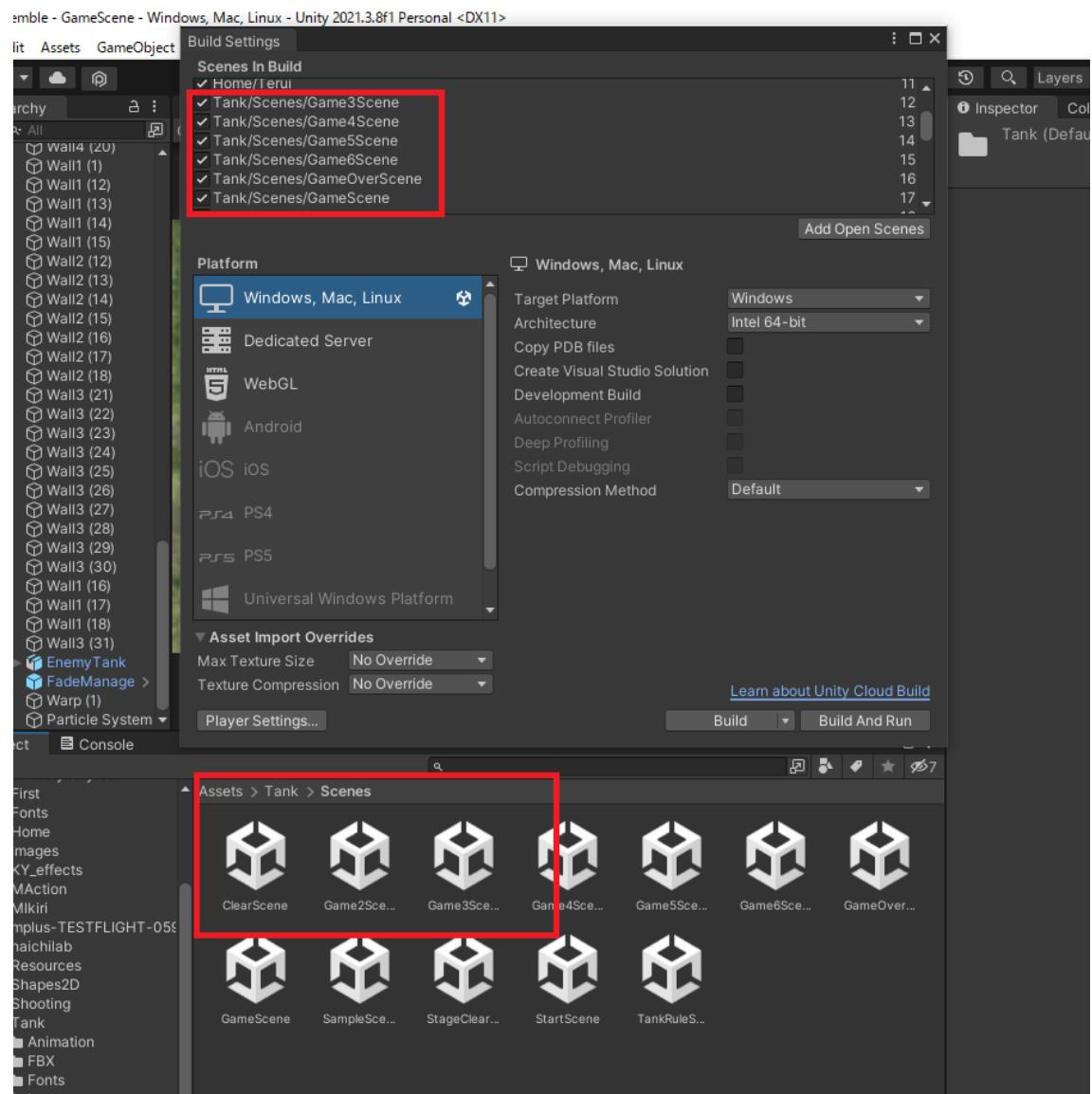


図 4.8.2.5 Build Settings 設定画面

その後は、各ゲーム同様、Unity のボタン機能や「LoadScene」関数などを使うことで、メニュー画面から各ゲームにシーン移動したり、図 4.8.2.6、図 4.8.2.7 のようにゲームから「HOME」ボタンを押して、メニュー画面に戻ったりすることができるようにした。



図 4.8.2.6 タンクゲームのメニュー画面に戻るボタン



図 4.8.2.7 カードゲームのメニュー画面に戻るボタン

5 おわりに

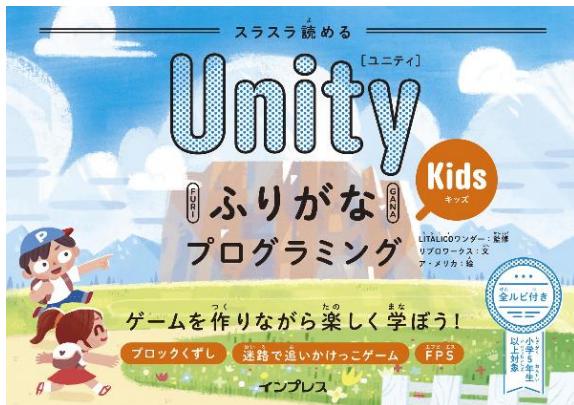
本研究を通して作品を作り上げる難しさを痛感した。思わぬところでエラーが出たり、エラーが出ていないにもかかわらずうまく動作しなかったりすることが多くあった。さらに、パソコンの個体差でもゲームの挙動に微妙な違いが出たり、設定の変更が必要だったりするところが難しいポイントだった。

また、計画を立てる重要性、グループで作り上げる難しさを実感した。チーム内の役割分担や、パソコンのスペックを考慮しての計画ができていなかった。Unity という環境が初めてで、自分が作りたいゲームに応じて必要な機能をひとつずつ加えていくのに予想以上に時間がかかり、計画通りに進まないことが多々あった。その都度行う計画の見直し、計画の変更が大変だった。ゲーム制作という初めてのこと挑戦したわけだが、始めの計画をもつと練っていれば、先を見越した作業ができたのではないかと反省している。しかし、各々が自分でゲームの開発をしつつ、ほかの人が作ったゲームのデバッグを行うなど、難しいこともたくさんあったが、チーム開発での楽しさを感じることもできた。

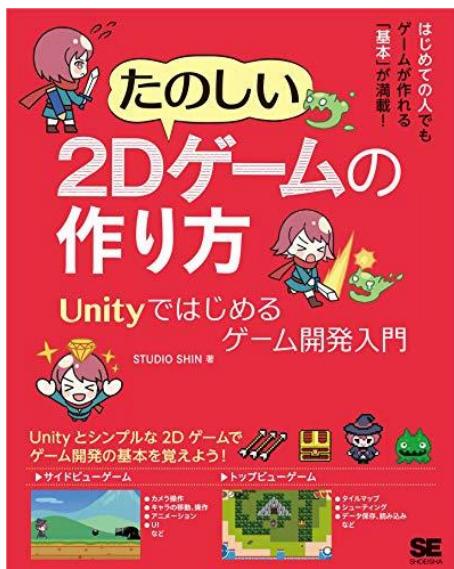
参考にしたものはあるが、オリジナルのゲーム仕様を自分たちだけで一から作り上げる上で、難易度、デザインなどを考えながら作成するのに苦労した。ゲームのルールからデザインまで、オリジナルのものを考えなければならなかつたため大変だった。例えば、カードゲームのカードの効果、各ゲームのキャラクターのデザイン、メニューのアイコンなどは自分たちで考えたものが多い。単にゲームを作るといつても、「みんなができるゲーム」として成り立つように仕上げるのが難しく、仕様や難易度の調節を確定するのに苦労した。ゲームを製作しながら、デバッグ作業として実際に自分たちのゲームをプレイし、だれでも楽しめるゲームになるよう、難易度を調節した。人によって感じる難しさは違うと思うが、誰でもある程度楽しめる難易度にできたので良かった。

グループでの開発ということで、メニュー画面を作り、メニュー画面から各ゲームへ、各ゲームからメニュー画面へ移る機能を実施した。これにより、一つのコンテンツとしてまとめることができ、難易度や遊びやすさなどを含めて、全体に統一感を出すことができた。

6 参考文献



株式会社インプレス 出版
「Unity ふりがなプログラミング」
リブロワークス 著



翔泳社 出版
「たのしい2Dゲームの作り方」
STUDIO SHIN 著

【Unity 入門】60分で作るシューティングゲーム
<https://nn-hokuson.hatenablog.com/entry/2016/07/04/213231>

Unity 入門に最適な、テトリス講座【Unity】
https://www.youtube.com/@Game_craft12

ゲームの作り方【Unity 初心者入門講座】Renewal
<https://www.youtube.com/@dkrevel6735>