9/5

SHAPE MATCHER

- キー入力をGetAxisRawに変えたい

メモ

- Shape Variants List
    - Shapeは、プレイヤーの形状に対応
    - .Variants[]のVariantで、壁の形を決めている。Shapeのプレヤーがクリアできる角度を、Allowed Anglesで定義している

壁の実装方法の検討

8/29

A game where you jump to avoid thorns

- Unity Cloudへの接続は不要
- prefab > Prefabs
- scripts > Scripts
- FileメニューのProject Profiles > Player Settingsの設定
    - Company Nameを、半角英数で自分の名前などに
    - Product Nameに、実行ファイル名を、半角英数で設定
    - ビルドのバージョンを、Versionに設定。1.0は、正式版。最初に、0.1.0などに設定する

Titleシーン

- Canvasは、Main Cameraの子供から外す
- CanvasのCanvas Scalerの設定で、Screen Match ModeをExpandにする

Gameシーン

- Player
    - 表示の情報は、子供に設定する
- Playerスクリプト
    - jumpcount > jumpCount
    - Update()内のspeedは、インスタンス変数や定数、SerializeFieldなどとして実装した方がよい

        - float speed => 5.0f;
        - [SerializeField] float speed = 5.0f;
        - など
    - Input.GetKeyDownは、デバッグ用。GetButtonDown()を利用したい
    - Rigidbody2Dを使うなら、左右移動も、Rigidbody2Dで動かしたい
    - GetComponentは、やや重い。何回も使う予定がある場合は、インスタンス変数に定義して、Awake()などで、GetComponentしておいた値を保存して使うのがよい
    - Jump()内の、rb != nullのifは不要。rbがnullなら、Rigidbody2Dがないという異常な状態なので、エラーを出して止めたほうがよい
    - ジャンプをAddForceでやると、現在の落下速度に速度を足すことになるので、ジャンプの高さが微妙に変化して、プレイヤーの予想に反した動きになる
        - 微妙な変化は避けて、2〜3段階でジャンプの高さを分けた方が、動きがしっくりくる
        - あるいは、rb.velocity.yに、直にジャンプの初速を設定すれば、安定した高さのジャンプにできる
    - 重力を変えたい場合、Editメニュー > Project Settings > Physics2Dを開いて、Gravityを設定。2Dのジャンプゲームでは、2〜4倍程度にすると、ふんわり感を消せる

その他

- ゴールまでの距離があると、遊びやすい

色々雑な異世界に一瞬だけ飛ばされました

-

block

- Unity Cloudは、接続しなくてよい
- Prehub > Prefabs
- スクリプトは、Scriptsフォルダーに入れておきたい
- 外部のアセットを使っている場合などは、Assetsフォルダー直下に、ゲーム名のフォルダーなどを作って、自分で作成したアセットは、すべてその中に入れておくと、管理しやすい
- player > Player
- player2 > Player2
- ButtonやButton2は、スクリプトの役割がわかる名前にする
- Material > Materials
- Materialフォルダーは、Scenesの外へ
- Ballスクリプトは、Scriptsフォルダーへ
- PhysicsMaterialなどの、あまり数が多くない種類のものは、Settingsフォルダーなどを作って、その中に入れる

Titleシーン

- CanvasのCanvas Scalerを設定

- Buttonscriptの名前は、ChangePlayなどにする。あるいは、スクリプトを、ボタン自体にアタッチする

Buttonスクリプト

- クラス名が、ファイル名と一致していない
- スクリプトファイル名を、SceneChangerにしたい
- Playシーンを呼び出すだけなら、ChangePlayなど、見てわかる名前にしたい
- Button2なら、ChangeTitle

Playシーン

Playerクラス

- Input.GetAxis()は、アナログスティック用。キーを押すと加速、離すと減速して、なめらかな入力をシミュレーションする
- Input.GetAxisRaw()は、デジタル入力用。キーを押したらすぐに最大値になり、離すと0になる

Player.cs

```
myRigidbody.linearVelocity = new Vector3(Input.GetAxisRaw("Horizontal") * speed, 0f, 0f);
```

- linearVelocity = 直線速度ベクトル = 移動速度のベクトル
- angularVelocity = 角速度ベクトル = 回転速度のベクトル
- myRigidbody.linearVelocityは、Vector3型なので、同じ型のデータを受け取る
- やりたいことは
    - myRigidbody.linearVelocity.x = Input.GetAxisRaw("Horizontal") * speed;
- ところが、UnityのVector3は、構造型の参照なので、個別に値の設定ができない
- YとZは0のまま、Xの速度を設定したい
    - new Vector3(Input.GetAxisRaw("Horizontal") * speed, 0f, 0f);
- newは、メモリを確保するので、使いたくない
    - ゲーム業界のみ。通常の業界では、これでいい
    - 解決策1：一度、ローカル変数に取り出す

```
var v = myRigidbody.linearVelocity;
v.x = Input.GetAxisRaw("Horizontal") * speed;
myRigidbody.linearVelocity = v;
```

- 解決策2：ベクトルの掛け算を使う

```
myRigidbody.linearVelocity = Input.GetAxisRaw("Horizontal") * speed * Vector3.right;
```

- Vector3.right = Vector3(1, 0, 0);

Player2.cs

```
float verticalInput = Input.GetAxis("Vertical");
transform.Translate(Vector3.up * verticalInput * speed * Time.deltaTime);
```

- transform.Translate()は、現在の座標から、指定したベクトルへ、直にワープさせる
    - 衝突が起きない。そのため、壁をすり抜けた
- Rigidbodyの衝突を使って、移動制限をしたい場合、transform.positionへの足し算や、transform.Translate()は使えない
    - RigidbodyのlinearVelocityを使う

```
float verticalInput = Input.GetAxis("Vertical");
```

- ローカル変数verticalInputに、上下の入力を代入。-1〜1

```
transform.Translate(Vector3.up * verticalInput * speed * Time.deltaTime);
```

- Vector3.upに、上下の入力値とspeedをかけて、更新秒数であるTime.deltaTimeをかけたものを、かけて、上下方向の移動距離を求めて、移動させている
- Time.deltaTimeは、前回からの経過秒数。例えば、60fpsで動いているなら、0.01666…. 秒が入る
- 例えば、verticalInputが1、speedが1だとすると、Vector3.up * verticalInput * speedは、(0, 1, 0)になる
    - この値を直に座標に足すと、1回の画面の更新で、1m動くことになる
    - 画面の更新が、400fpsだとすると、1秒間に400回Updateは呼び出されることになる
    - そのまま速度を足すと、1秒間で400m動くことになる
    - これを避けるために、経過秒数Time.deltaTimeをかけている
    - 400fpsなら、Time.deltaTime = 1/400なので、これをかければ、1秒間に1m移動に変換できる
- 一方、Rigidbody.linearVelocityは、物理更新のタイミングで、移動速度を考慮して、Unityが自動的に更新する。そのため、更新秒数を気にせず、直に速度を設定できる
- 以上から、Playerのスクリプトを参考に、次のように変更すればよい

```
myRigidbody.linearVelocity = Input.GetAxisRaw("Vertical") * speed * Vector3.up;
```

Player2オブジェクト

- BoxColliderに、Physics Materialが設定されていて、そのBouncenessが1になっている。これにより、完全弾性係数となり、何かにぶつかったときに、同じ速度で跳ね返るようになっている。これが、壁にぶつかった時に震える原因
- ボールとの摩擦をなくしたいなら、Bouncenessを0に設定したPlayer用のPhysicsMaterialを作って、設定する

SHAPE MATCHER

- TextMeshProで、Warningが出る場合、Windowメニュー > TextMesh Pro > Impot TMP から始まる2つのメニューを選んで、読み込むとなおる場合がある
- デバッグ用のログを消しやすくする方法
    - プリプロセッサを使う = C言語でよく使うやつ
        - プリ＝事前
        - プロセッサ＝処理器
        - プログラムをコンパイルする前に、処理をするためのもの
    - #define・・・ラベルを定義する
    - #if・・・これに続くラベルが定義されていたら、ここから#endifまでのコードを有効にする
    - スクリプトの先頭行に、次を入力

```
#define DEBUG_LOG
```

- ログ表示を、次のようにする

```
#if DEBUG_LOG
    Debug.Log("Current Wall Speed: " + CurrentSpeed);
#endif
```

- 以上で、先頭行の#defineをコメントアウトすると、デバッグ表示をスクリプトから消すことができる
- 次のものも便利

```
[System.Diagnostics.Conditional("DEBUG_LOG")]
void DebugLog(string message)
{
    Debug.Log(message);
}
```

壁の形が変わるタイミング

- 形を変えるタイミングを、座標を戻すときに変更する
    - Damager.csのOnTriggerEnter内にある形を変更する処理を、MovingWall.csのFixedUpdate内のnewPosition = initialPosition;をしている場所に、移動すると、座標が戻る瞬間に、形を変えられる
    - いろいろ変更点があるので、整理してから
- あるいは、TheWALLを2つ以上使う
    - この場合、複数の壁を出して、ゲーム性を増やせそう

PlayerMover.cs

- rotating > isRotating

壁の作り方

プログラムのベストプラクティス

- 単一責任の原則
    - ある一つのものは、一つの役割のみを担う
    - 構造をシンプルにしたい！
- 壁の役割
    - 出現＞形と場所が決まる
    - 近づいてくる
    - プレイヤーに接触して、成否が決まる
    - 再利用 or 消滅
        - 再利用の方が、メモリの確保と解放が発生しないので、理想的＞オブジェクトプールで管理
        - 消滅は、構造がシンプル。今回は、それほど壁は出現頻度が高くないので、オブジェクトプールで管理する必要性は低い
    - 今回は、消滅で考えた方が、シンプルでよい
- 実装として、形ごとに、別のプレハブとして壁を用意するのが、単一責任の原則的には適切
    - 分けることで、当たり判定の処理を、簡略化できる可能性がある

正解判定

ねこやじるしよけ

-

よけろ！！

-