



# Unityはじめるよ

～初心者向け講座！簡易レースゲーム制作でUnityを覚えよう～

統合開発環境を内蔵したゲームエンジン  
<http://japan.unity3d.com/>

※いろんな職業の方が見る資料なので説明を簡単にしている部分があります。正確には本来の意味と違いますが上記理由のためです。ご了承ください。  
この資料内の一部の画像、一部の文章はUnity公式サイトから引用しています。

# 資料の内容

- ・ レースゲームを作るにあたって考えること
- ・ 設計
- ・ 実装

# まえおき

この資料で伝えたいことは、  
Unityの使い方や、ゲームを作る上での  
考え方をまとめたものです。

企画やゲームルールは深く追求した内容  
ではありません。

また、初心者向けではありますが、  
フォルダやゲームオブジェクトを作成できる  
くらいの基本操作はできることが前提の  
内容となります。

レースゲームを作るにあたって考えること

# ルールを明確にし、仕様を決める

ルールは以下のとおり。

- ・コースを3周したらゴール
- ・敵(AI)と順位を競う

仕様は次のページから。

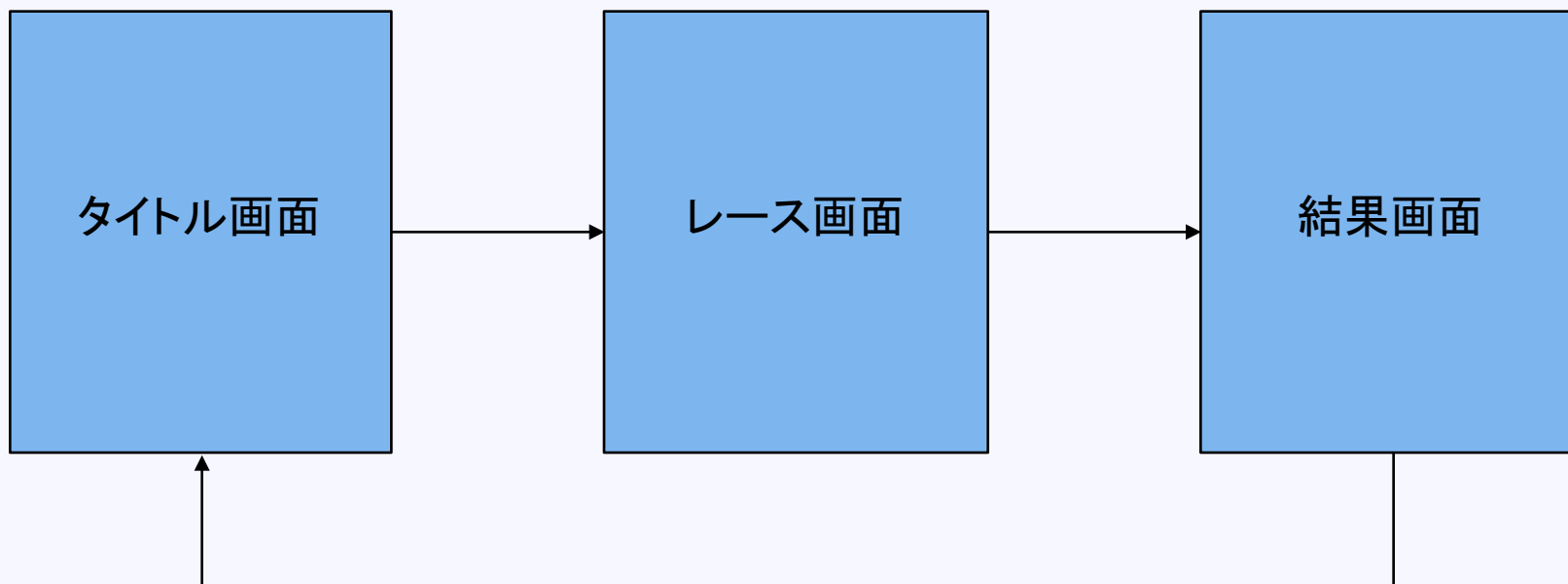
## ポイント！

ルール・仕様を明確にすることで、  
開発中の迷いやブレをなくし、効率よく開発することができる。  
※企画を考える時にターゲットを明確にするのと同じ

また、必要な素材や実装手順を、実装前に把握できるので、  
起こりうる問題にも事前に対処可能。

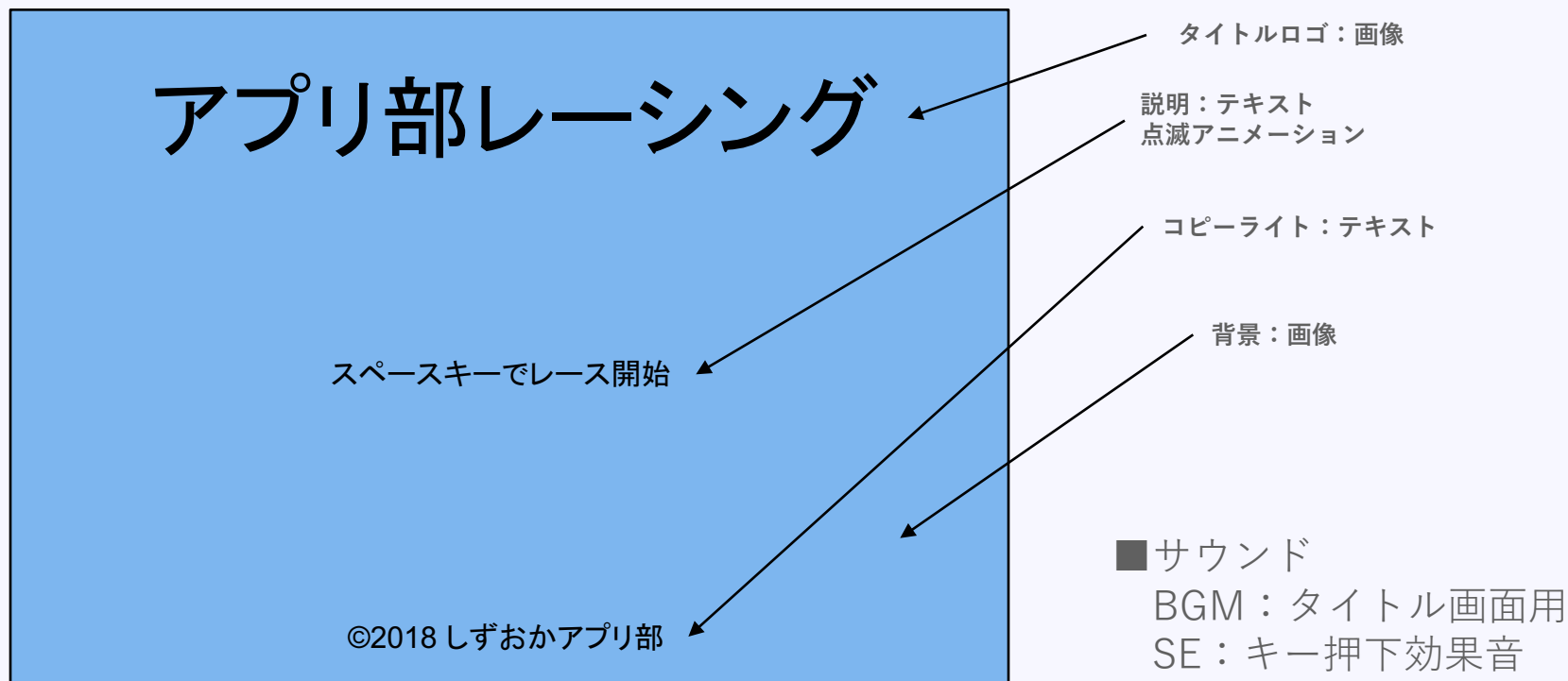
# 仕様を考える（簡易版）

まずは画面構成と遷移図



次のページから、簡単に各画面の仕様をまとめていく。

## 各画面の詳細：タイトル画面



### ■動き

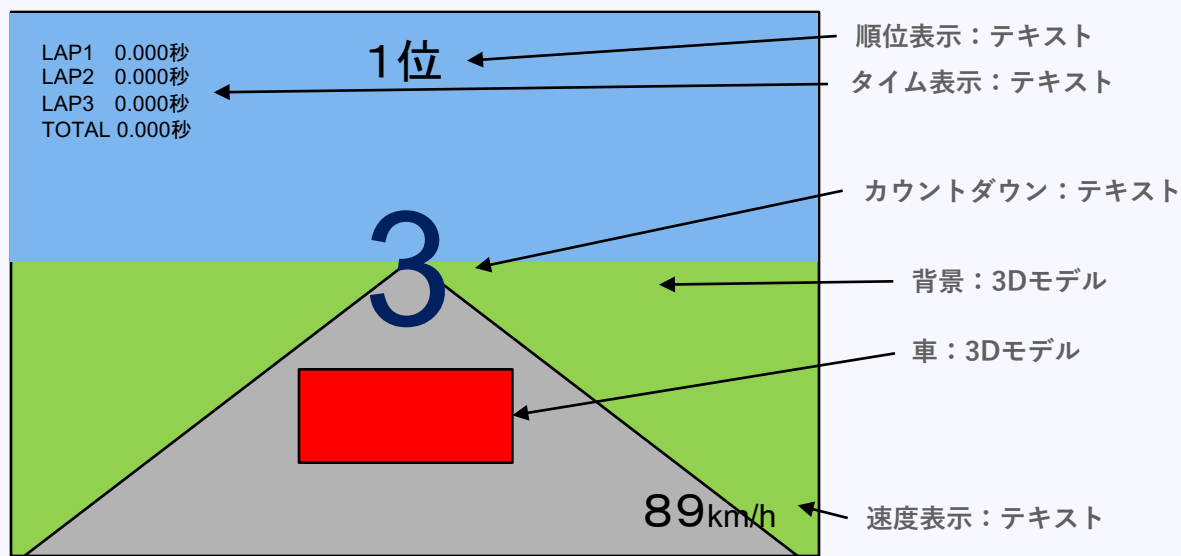
スペースキー押下で、効果音を鳴らし、1秒後にBGMを止め、レース画面へ遷移。

### ■画面切り替えトランジション

イン・アウトともなし。パッと切り替える。

# 各画面の詳細：レース画面

## カウントダウン時



## ■ サウンド

BGM: レース用BGM  
SE: カウントダウン用に2種  
プップップピー

## ■ 動き

画面遷移1秒後から、3、2、1、GOのカウントダウン。カウントダウンは1秒間隔。  
カウントに合わせて効果音を鳴らす。キー入力は無効。  
GOになった瞬間、BGM再生開始、キー入力有効に。

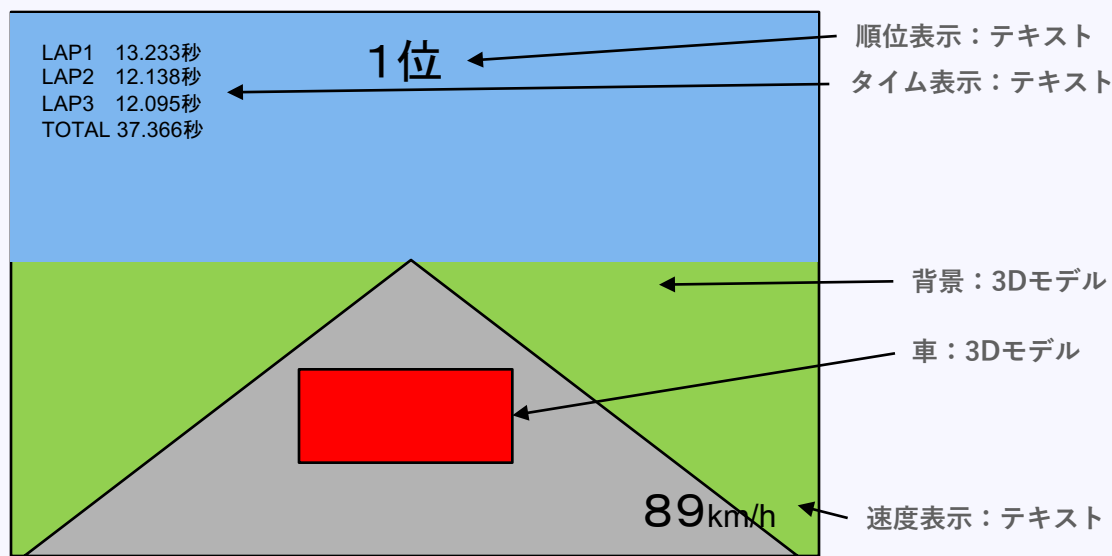
## ■ 画面切り替えトランジション

インなし。パッと切り替える。



# 各画面の詳細：レース画面

## レース時



## ■ サウンド

BGM：レース用BGM

SE：エンジン音

ぶつかった時の音

チェックポイント音

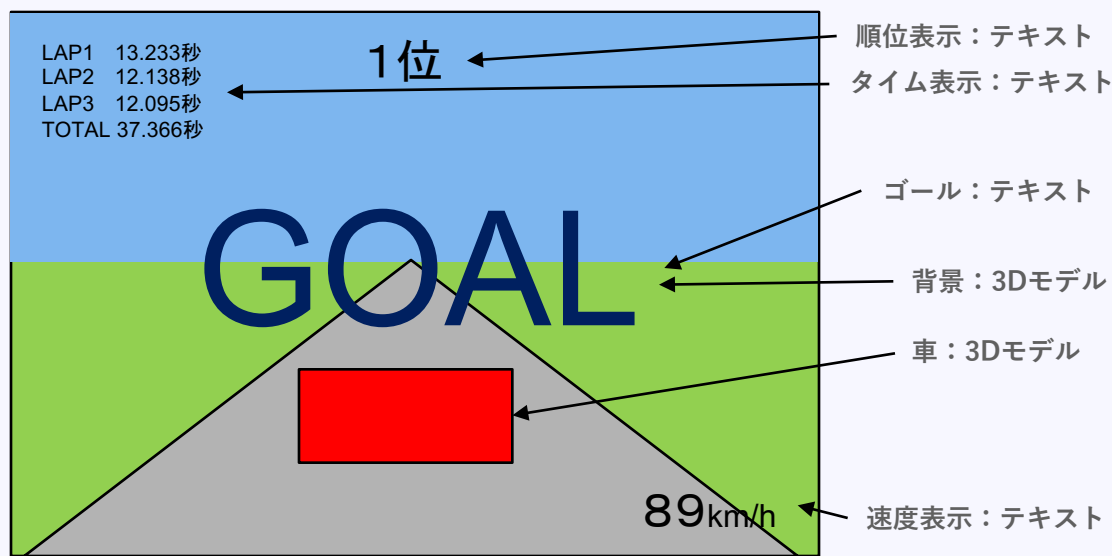
## ■ 動き

操作方法に従って車を動かす。カメラは車の広報斜め上から見下ろす。  
タイム表示、順位表示、速度表示は随時更新。効果音も適宜鳴音。

最高速・旋回性能・敵の強さは、実装しながら調整する。

# 各画面の詳細：レース画面

## ゴール時



- サウンド  
BGM：ゴール用ジングル

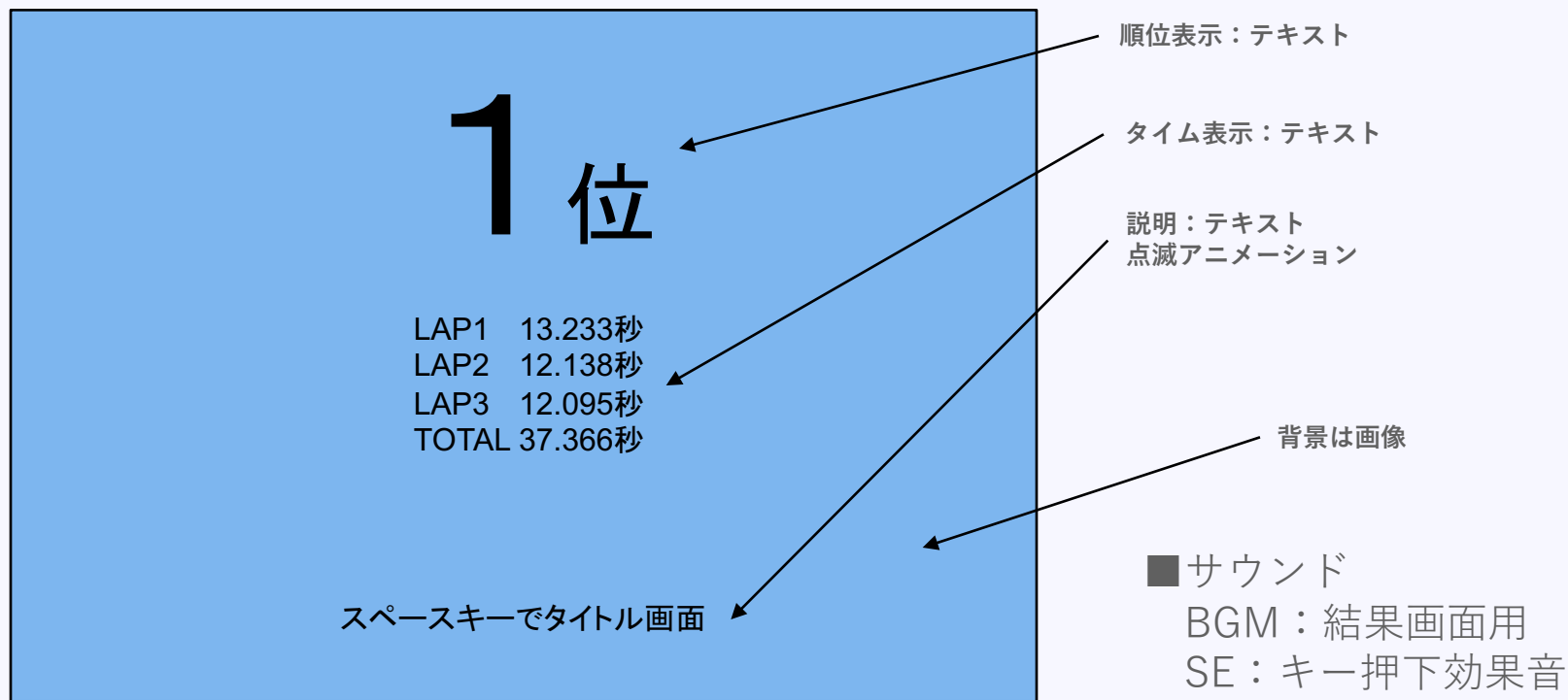
## ■ 動き

レース用のBGMを止め、ゴール用ジングルを鳴らす。  
合わせて画面にGOALテキストを表示し、キー入力を無効。  
自車の操作を自動操作に切り替え。  
5秒後に結果画面に遷移。

## ■ 画面切り替えトランジション

アウトなし。パッと切り替える。

## 各画面の詳細：結果画面



### ■動き

スペースキー押下で、効果音を鳴らし、1秒後にBGMを止め、タイトル画面へ遷移。

### ■画面切り替えトランジション

イン・アウトともなし。パッと切り替える。

# 必要な素材

画面の仕様を考えると必要な素材がわかる。  
簡単にまとめてみる。

| 画面   | 種類    | 内容       |
|------|-------|----------|
| タイトル | 画像    | ロゴ       |
|      |       | 背景       |
|      | サウンド  | タイトルBGM  |
| レース  | 3Dモデル | コース      |
|      |       | 車        |
|      | サウンド  | レースBGM   |
|      |       | ゴール用ジングル |
|      |       | エンジン音    |
|      |       | ぶつかった時の音 |
| 結果   | サウンド  | チェックポイント |
|      |       | 結果画面BGM  |
| 共通   | サウンド  | キー押下効果音  |

# レース画面のその他の仕様

画面の仕様は決まったが、車の挙動や周回判定AIなどを  
どうやってまとめようか…。

という時は、とりあえず、  
必要事項を箇条書きで書き出してみよう。  
図で補足できるところは、図でわかりやすくしてみよう。

# レース画面のその他の仕様

## ■車の挙動

- ・徐々に加速する。
- ・アクセルを離せば徐々に減速。
- ・バック動作＝ブレーキとする。
- ・壁にぶつかったら減速。

- ・ドリフトできたらいいな。

## ■操作方法

- ↑キー：前進
- ↓キー：後退（ブレーキにも）
- ←キー：左旋回
- 右キー：右旋回

※調整しやすいつくりとする（インスペクタに表示させる）

# レース画面のその他の仕様

## ■周回チェック、順位チェック方法

- ・スタート地点から、コースの形状に合わせて複数のチェックポイントを並べておき、順番に通過することで周回チェックを行う。  
また、どこのチェックポイントまで通過しているかの比較で順位を決める。  
同じチェックポイントにいる場合は、次のチェックポイントに近い方が、順位が上と判定する。

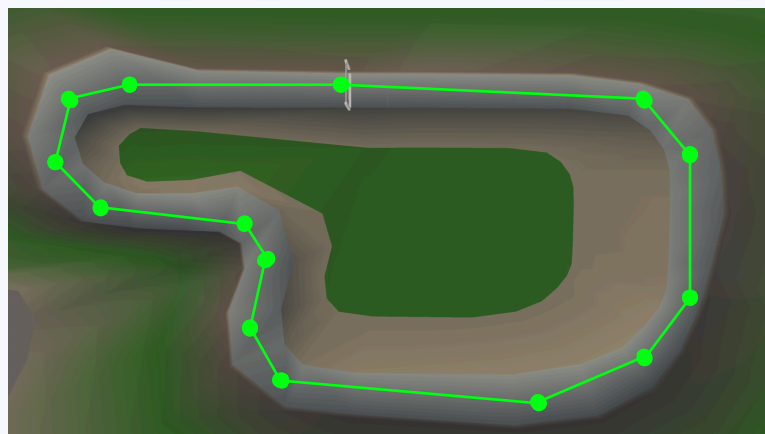


チェックポイント

# レース画面のその他の仕様

## ■AIについて

- ・プレイヤーと同じように、アクセル、ブレーキ、ハンドル操作を行わせたい。
- ・周回チェックと同じように、コース上に目的地点を配置して、そこに向かうように操作させる。目的地点にたどり着いたら次の目的地点に変更、を繰り返して、コース上を走るような作りとする。
- ・上記のチェックポイントを、インコースルート、アウトコースルート、真ん中ルートの3パターンを用意し、リアルタイムに適宜切り替えることで難易度調整を行う。



### ●目的地点

上図では1ルートだが、3ルート用意して難易度調整を行えるようにする。



細かいことを言えば仕様書としては全然足りていないが、指標となるものができた。

次はこの指標を元に設計を行う。

# 設計

# 設計の仕方

設計というと難しく感じてしまうので、  
最初のうちは、

- ・ **どんな処理が必要なのか**  
**その処理をどのスクリプトにさせるか**

を考えてみれば良い。


# 画面ごと考えてみる

## ■ タイトル画面

スクリプト：**TitleManager.cs**

役割：キー判定、シーン切り替え

必要な処理



## ■ 結果画面

スクリプト：**ResultManager.cs**

役割：順位表示、タイム表示、  
キー判定、シーン切り替え

## ■ レース画面

スクリプト：**RaceManager.cs**

役割：カウントダウン、ゴール後処理、  
順位表示、タイム表示

スクリプト：**CarController.cs**

役割：車の移動、エンジン音の制御

スクリプト：**Player.cs**

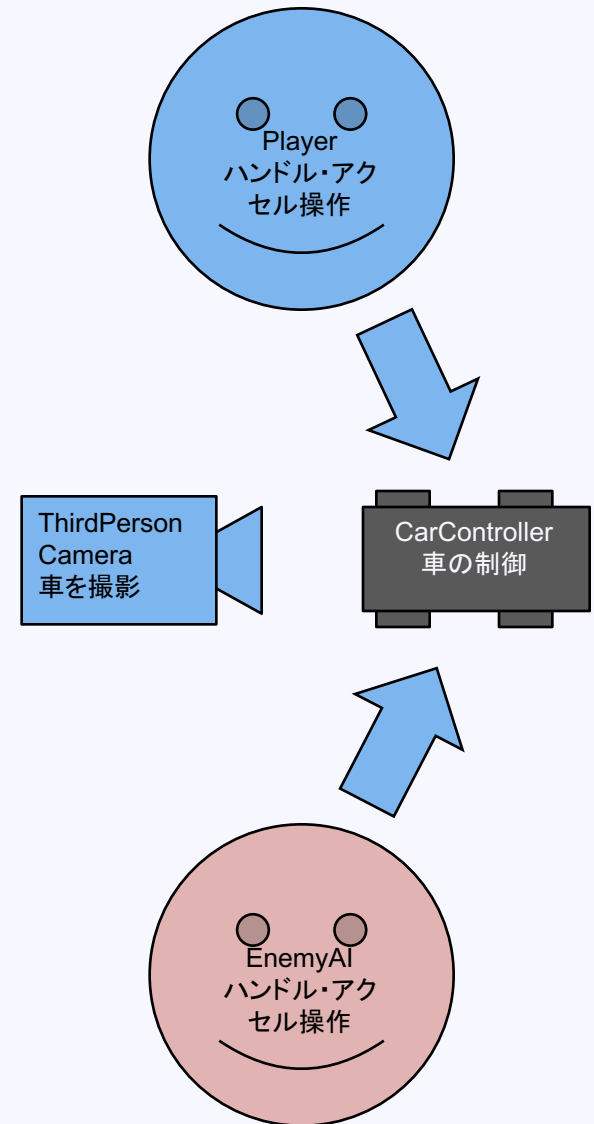
役割：キー入力を受けて車を操作する

スクリプト：**EnemyAI.cs**

役割：CPU車の操作

スクリプト：**ThirdPersonCamera**

役割：車を追うカメラ制御



# 実装

# 下準備

作る指標が定まったので、早速実装に。

まずはプロジェクトを滞りなくする進めるために、  
整理整頓とルール（命名規則）を明確にする。

## 整理整頓

フォルダを切る。

あらかじめ必要となるであろうフォルダを作っておく。

- Animations
- Materials
- Models
- Prefabs
- Scenes
- Scripts
- Shaders
- Sounds
- Textures

今回は上記の9つを用意した。どこに何があるか一目でわかる。

規模が大きいゲームは、部品ごとに上記のフォルダ構成を作ると良い。

例えば、Playerフォルダを作り、その中に上記フォルダ群を入れる。

部品単位で完結させるイメージ。そうすることで、他のプロジェクトでも再利用しやすくなるし、データの受け渡しがしやすくなる。



# 命名規則

ファイル名の付け方や、  
プログラム内での変数名や関数名などの名前の付け方を  
統一しておくことで、可読性が上がるメリットがある。

Unityだと、下記の通り。

ファイル名：アッパーキャメルケース

クラス名：アッパーキャメルケース

変数名：ローワーキャメルケース

関数名：アッパーキャメルケース

列挙体：アッパーキャメルケース

アッパーキャメルケース（パスカルケース）

単語の先頭が大文字。例：DoSomething

ローワーキャメルケース（キャメルケース）

最初の単語は小文字で以降は大文字始まり。例：doSomething

名前は長くなっても良いので、**見たら意味がわかる名前をつけることが大切**

# 用語の整理

| 用語         | 内容   |
|------------|--|
| GameObject | ゲームシーンに出てくるオブジェクトは全てGameObjectという。<br>(3Dモデル、カメラ、ライトなど)  |
| コンポーネント    | GameObjectにくっつける（アタッチ）する部品。<br>GameObjectに見た目や振る舞いを持たせる部品。<br>(レンダラ、コライダー、リジッドボディなど)<br>スクリプトもコンポーネントである。<br><br>どんなGameObjectにも、位置、向き、大きさを表す<br><b>Transformコンポーネント</b> がアタッチされている。 |
| アセット       | 素材ファイル。コンポーネントにセットすることで、コンポーネントの見た目や動作を変化させるものもある。<br>(テクスチャ、マテリアル、オーディオクリップなど)  |

# 車の移動について考えてみる

車を移動させるにはどんな方法が良いだろうか？  
パッと思いつく方法が、

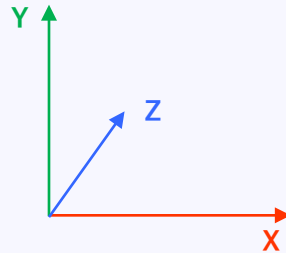
- ・ 座標を制御する
- ・ 物理演算で制御する

の2つ。

リアルな動きを追求するなら物理演算。

# 座標を制御する方法

Unityの座標の基本は、  
X軸が横、Y軸が高さ、Z軸が奥行きとなっている。



なので、X座標とZ座標を制御すれば移動ができる。  
ただ、X座標とZ座標を直接いじって車の動きを表現するのはちょっと大変。

※X座表 =  $\sin(\text{向き}) \times \text{移動距離}$ 、Z座表 =  $\cos(\text{向き}) \times \text{移動距離}$ 、でできるが  
これだけだと坂道を登るような状況では移動量が正確でない

なので、Unityの便利な仕組みを使わせてもらう。  
日本語で書くと、

**移動後の座表 = 現在の座表 + (前方向 × 移動距離)**

となる。  
普通に計算すると**前方向**を求めるのが大変だけど、  
Unityでは、**transform.forward** が前方向を表す。  
プログラムで書くと、

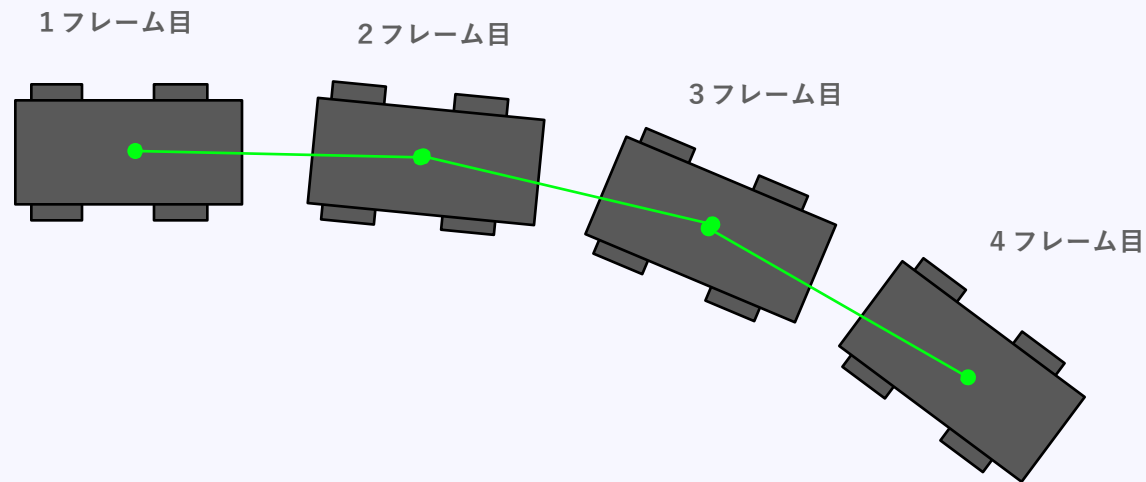
**transform.position += transform.forward \* moveDistance**

という感じ。



Transformコンポーネントの位置情報という意味

考え方としては、  
移動後の座標を毎フレーム求め続ける  
ということ。



# 物理演算で制御する方法

ちゃんと物理演算やるなら、車にタイヤをつけて、そのタイヤに回転の力を与えて・・・と、それはそれで大変なので、今回は、車に直接前進する力をかける簡易実装で考える。

物理演算を使う場合は、「移動後の座表」を求めるのではなく、現フレームでどれだけ力を加えるかを考える。

プログラムで書くと

```
rigidBody.AddForce(transform.forward * 力);
```

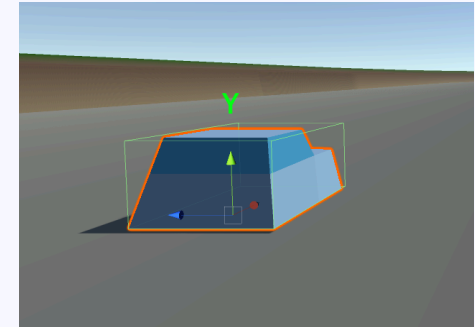
となる。

# 車の向き

座標を制御する方法でも物理演算を使う方法でも車の向きの制御が必要。

向きは、単純にY軸の回転で制御する。

プログラムで書くと、



```
Vector3 rot = transform.rotation.eulerAngles;
```

```
rot.y += 変化させたい回転量;
```

```
transform.rotation = Quaternion.Euler(rot);
```

Transformコンポーネントから回転情報を受け取る

受け取った回転情報を元に回転値に変化をつける

変更した回転値をTransformコンポーネントに反映

※回転させる関数もあるが、仕組みを知ってもらうために、上記のように記述。



# 1 フレームあたりの移動量

ゲームは1秒間に何十コマもの絵を切り替えて、なめらかにアニメーションしているように見せている。

コマのことを**フレーム**という。

1秒間のコマ数のことを**フレームレート**という。

フレーム数が多い方がなめらかに見えるが、高画質なグラフィックにすればするほど処理に時間がかかり、フレームレートが落ちてしまう。また、パソコンやスマホの性能にも影響される。

ここでひとつ考えなくてはいけないことがある。

1 フレームに 2 m進むというような実装をしてしまうと、フレームレートが違う環境では、1秒後に進む距離が変わってしまう問題が発生する。

なので、考え方を「1 秒に何m進むか」とする必要がある。この考え方で1 フレームの移動量を求める計算式は、

**1 フレームの移動量 = 移動距離 × 1 フレームの処理時間**

となる。

Unityでは、1 フレームの処理時間は、

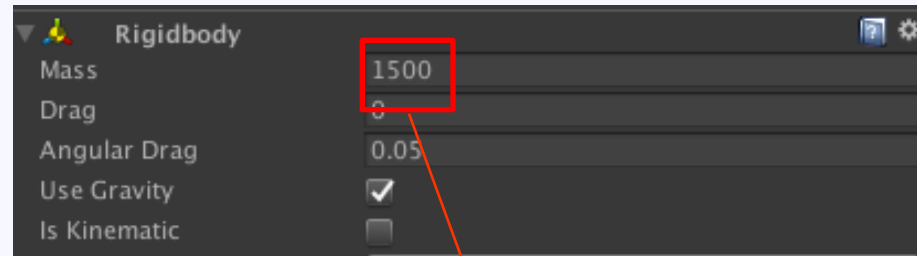
**Time.deltaTime**

で表される。

# 物理演算の制御で実装してみる

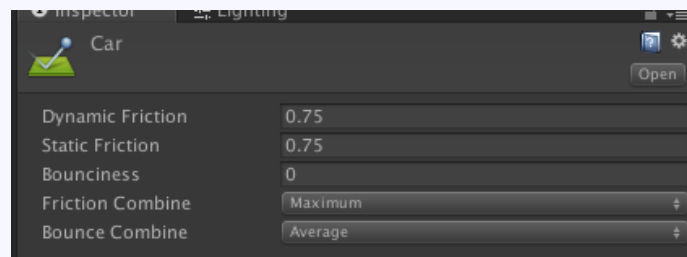
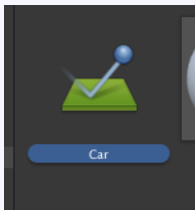
# 車両の設定

車重を本物の車と同じくらいに。



1500kgにした

PhysicMaterialを使い、抵抗を設定



〇〇Frictionと書いてあるのが、抵抗値。  
0が抵抗なし、1がMax抵抗。

# 車の処理(CarControllerクラス)を実装してみる

まずは前進処理。

「加速力」と「前に進ませる力の上限」、  
「現在の前に進ませる力」を管理する変数を用意。

```
[SerializeField]  
float m_accelSpec = 5f;           // 加速力  
  
[SerializeField]  
float m_maxForce = 100f;          // 前に進ませる力の上限  
  
float m_force = 0f;               // 前に進ませる力
```

[SerializeField]をつけるとインスペクタに表示される。  
publicをつけてもインスペクタに表示されるが、どこからでもアクセスできる変数になってしまうので、管理上好ましくない場合が多い。

プレイヤーも敵も同じ操作方法にするため、  
「アクセル」「ブレーキ」「ハンドル」操作を行う  
インターフェース(操作を行う口)を用意する。

```
bool m_forward = false; // アクセルを踏んでいるか
bool m_back = false; // ブレーキを踏んでいるか
bool m_left = false; // 左にステアリングを切っているか
bool m_right = false; // 右にステアリングを切っているか
```

```
/// 前進操作。
public bool Forward {
    set {
        m_forward = value;
    }
}
```

```
/// 後退操作。
public bool Back {
    set {
        m_back = value;
    }
}
```

```
/// 左に曲がる操作。
public bool Left {
    set {
        m_left = value;
    }
}
```

```
/// 右に曲がる操作。
public bool Right {
    set {
        m_right = value;
    }
}
```

関数と変数の利点を持ち合わせた「プロパティ」という仕組み。  
変数のようにアクセスできて、関数のように受け取った値を加工できる。

プレイヤーも敵も、  
CarControllerクラスの  
Forward, Back, Left, Right  
を呼ぶことで車を制御する。

# アクセルとブレーキ処理は「Pedal」という関数にまとめた。

```
/// ペダル操作。
void Pedal()
{
    // アクセル
    if (m_forward)
    {
        m_force += m_accelSpec * Time.deltaTime;
    }
    // ブレーキ(バック)
    else if (m_back)
    {
        m_force -= m_accelSpec * Time.deltaTime;
    }
    // 減速
    else
    {
        m_force = Mathf.Lerp(m_force, 0, 0.2f * Time.deltaTime);
    }

    // 最高速度キャップ
    m_force = Mathf.Clamp(m_force, -m_maxSpeed, m_maxSpeed);

    // 前方向に力を加える
    m_rigidBody.AddForce(transform.forward * m_force * Time.deltaTime, ForceMode.Acceleration);
}
```

アクセル操作がされているなら、  
スピードに加速力を足す処理を行う。  
Time.deltaTimeをかけることで、  
可変フレームレート対応

ブレーキ操作

なにも操作していないなら減速処理。  
次のページで紹介。

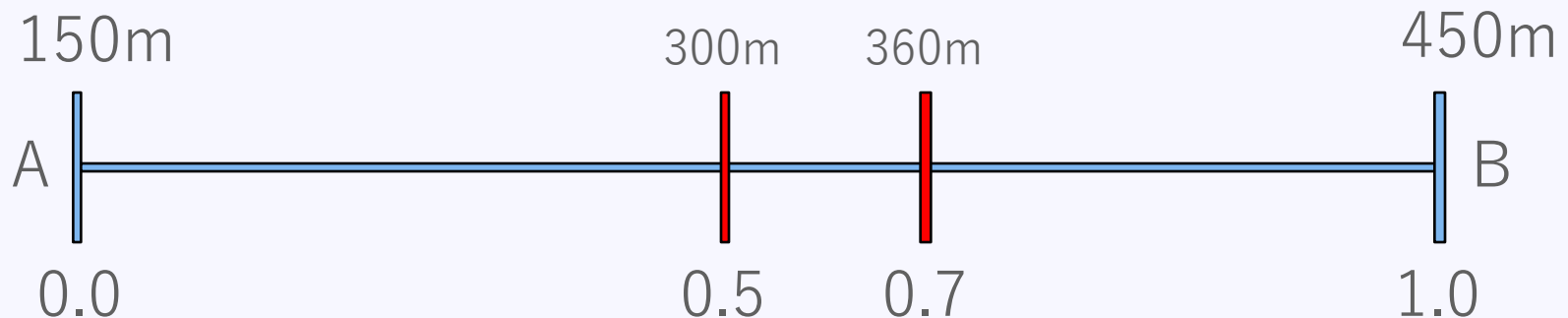
前に進ませる力が上限をこえないようにする処理。  
上限・下限を超える場合はそこでカットする。  
Clamp関数を使うと、上限のキャップが簡単にできる。

物理エンジンを使って力を加える

減速処理で使った、便利な関数「**Lerp**」。  
**線形補間**という処理を行う関数。

2 値の間の値を求めることができる。  
途中の値を補間して作り出すイメージ。

例えば、A地点が150m、B地点が450mだとして、  
A地点からB地点までの0.5(50%)の位置は300mです。  
0.7(70%)なら360mです。  
と。それを簡単に求めることができる。



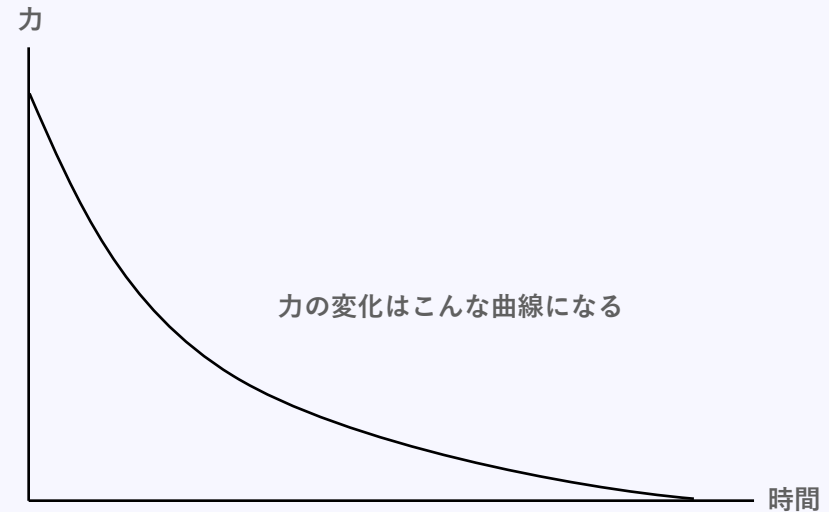


Lerpを使った減速処理はどういうアルゴリズムなのかというと、前に進ませる力が0になるまで、●%ずつ減らしていくということを繰り返している。

### 減らす割合が1%の場合

1フレーム目 100  
2フレーム目 99  
3フレーム目 98.01  
4フレーム目 97.03  
5フレーム目 96.06

となる。



目標値(0)から離れていれば大きく減速し、0に近づくにつれて減速値が少なくなっていく。

※一般的な減速処理というわけではありません。今回は減速処理にLerpを使っただけです。

次はコーナリング処理。

基本的にはアクセルと同じ仕組み。

「1秒に切れるハンドル角」と「最高ハンドル切れ角」  
「現在のハンドルの切れ角」を管理する変数を用意。

```
[SerializeField]  
float m_corneringSpec = 5f; // 1秒に切れるハンドル角  
  
[SerializeField]  
float m_maxCornering = 50f; // 最高ハンドル切れ角  
  
float m_steering = 0; // 現在のハンドル切れ角
```

# Steeringという関数にまとめた。

///  
ステアリング操作。

```
void Steering()
{
    if (m_left)
    {
        m_steering -= m_corneringSpec * Time.deltaTime;
    }
    else if (m_right)
    {
        m_steering += m_corneringSpec * Time.deltaTime;
    }
    else
    {
        m_steering = Mathf.Lerp(m_steering, 0, 0.1f);
    }

    m_steering = Mathf.Clamp(m_steering, -m_maxCornering, m_maxCornering);

    Vector3 rot = transform.rotation.eulerAngles;
    rot.y += m_steering;
    transform.rotation = Quaternion.Euler(rot);
}
```

左右に曲がる処理

ハンドルをセンターに戻す処理

ハンドルの切れ角の上限

ハンドルの切れ角を車体の向きとして反映

# ペダル操作とハンドル操作を 毎フレーム呼ばれるUpdate()関数内で呼ぶ。

```
void Update () { ← 毎フレーム呼ばれる関数
```

```
    // ステアリング  
    Steering();
```

← 前項で作ったステアリング処理

```
    // ペダル操作  
    Pedal();
```

← 前項で作ったアクセルブレーキ処理

```
    // 操作を初期化  
    m_forward = false;  
    m_back = false;  
    m_left = false;  
    m_right = false;
```

← プレイヤーや敵AIが操作したフラグ変数を初期化。  
減速するために操作をしない、コーナーを縫えた  
のでハンドル操作をしないということが考えられ  
るので、毎フレームの処理後に初期化する。

```
}
```

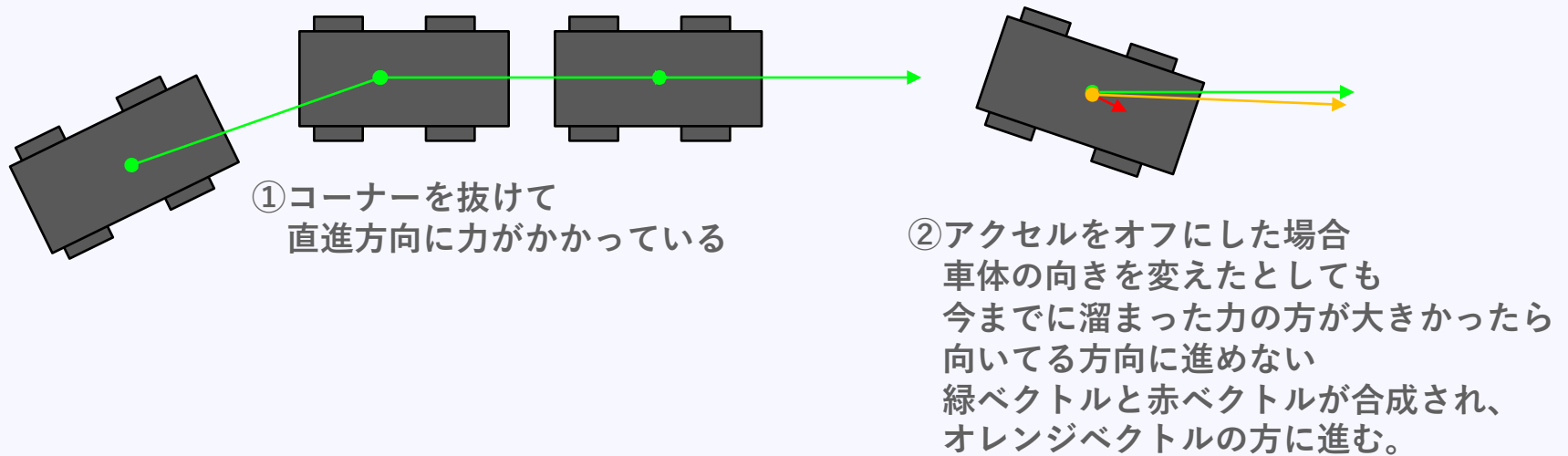
一応これでも動くのだが、コーナリング後の動きが不自然。

なぜか。

- ・アクセル処理では、向いている方向に力を加えている
- ・ハンドル処理では、車体の向きを変えている

一見うまくいきそうなのだが、  
**車体を動かすためには車重と抵抗を超える力がないと動かない。**  
アクセルを踏んでいれば十分な力を得られるが、**アクセル操作をやめ、力が一定以下になると、向いている方向に十分な力を与えられなくなってしまう。**  
つまり**アクセルを踏んでいない時は、車体の向いている方向に力をかけることができず、最後に力がかかっていた方向に進んでしまう。**

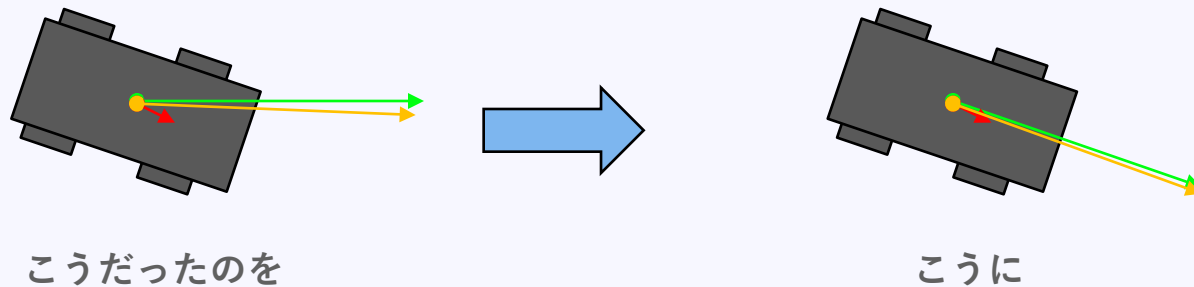
図で表すと、



ドリフトっぽい挙動になるが不完全。

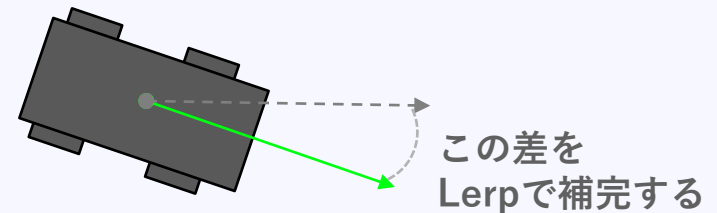
なので、

**車体の向いている方向に力の向きを徐々に変えてあげる**  
処理を追加。



こうに

アルゴリズムとしては、  
減速処理に使ったLerpを使って、  
車体の向いている方向に力のベクトルを近づける。



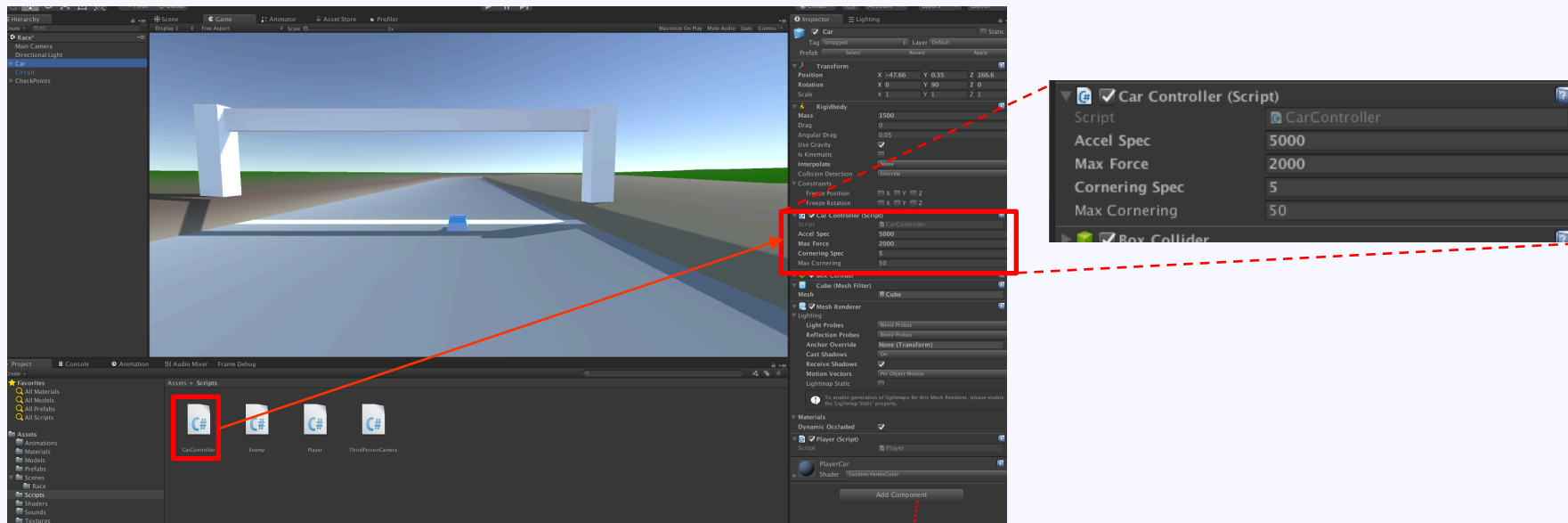
プログラムは以下の通り。  
Update()関数内に書く。

```
// 車体が向いている方向に力のベクトルを向けていく
① Vector3 targetVector = transform.forward;           // 車体の向き
② float magunitude = m_rigidBody.velocity.magnitude;  // 車体にかかっている力の強さ
③ targetVector.y = m_rigidBody.velocity.y / magunitude; // 車体にかかっている下向きの力
// 現在かかっている力を、車体の向いている方向に近づける
④ m_rigidBody.velocity = Vector3.Lerp(m_rigidBody.velocity,
                                       targetVector * magunitude,
                                       0.5f * Time.deltaTime);
```

- ①現在の車体の向きを目標ベクトルとして取得
- ②現在車体にかかっている力ベクトルから、力だけを取り出す
- ③現在車体にかかっている力ベクトルから、重力成分を抜き出し  
目標ベクトルに反映（クラッシュした時に車体がくるくる  
回ったとしても、正しい方向に力をかけられるように）
- ④目標ベクトルに向かって、徐々に向きを変える



ひとまず車を制御するスクリプトができた。  
スクリプトもコンポーネントなので、  
必ずゲームオブジェクトにアタッチする。



アタッチする方法は、

- ①ヒエラルキーの該当オブジェクトにドラッグ&ドロップ
- ②該当オブジェクトを選択後インスペクタにドラッグ&ドロップ
- ③該当オブジェクトを選択後AddComponentボタンから追加

次に車を操作するための、  
Playerクラスの実装を行う。

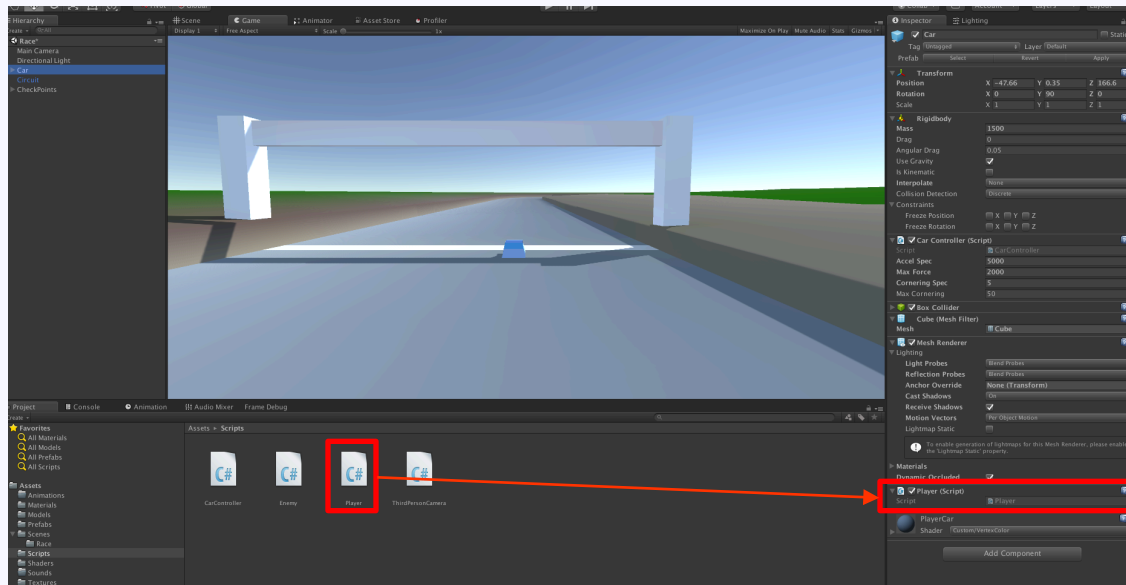
```
CarController m_carController; // 車制御用コントローラ

// Use this for initialization
void Start () {
    // 車制御用コントローラを取得
    m_carController = GetComponent<CarController>();
}

// Update is called once per frame
void Update () {

    // キーボードによる車の操作
    if (Input.GetKey(KeyCode.UpArrow))
    {
        m_carController.Forward = true;           // アクセル操作
    }
    else if (Input.GetKey(KeyCode.DownArrow))
    {
        m_carController.Back = true;               // ブレーキ(バック)操作
    }
    if (Input.GetKey(KeyCode.LeftArrow))
    {
        m_carController.Left = true;               // ステアリングを左に切る
    }
    else if (Input.GetKey(KeyCode.RightArrow))
    {
        m_carController.Right = true;              // ステアリングを右に切る
    }
}
```

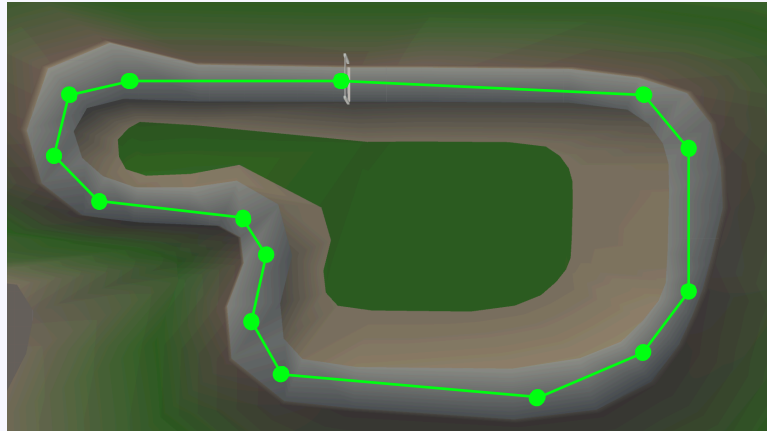
Playerクラスは操作するだけなのでシンプル。  
アタッチを忘れないように。



動作確認してみる。  
ドリフトもそれっぽくて、いい感じの挙動。

# 敵のAIの実装を考える

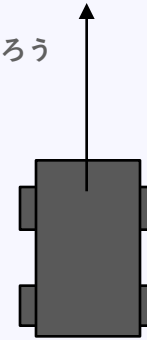
チェックポイントを次々と目指すことで周回させる。



● 目的地点

● 目的地点

左側にあるから  
ハンドルを左に切ろう



車体を目標チェックポイントの方に向かせてあげることがキモ。  
これをハンドル操作で行う。

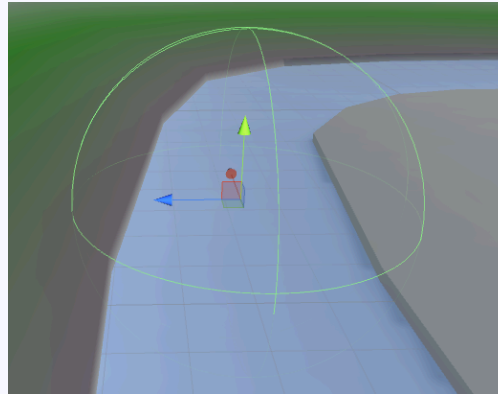
車体が向いている方向より、

左にチェックポイントがあれば左にハンドルを切る

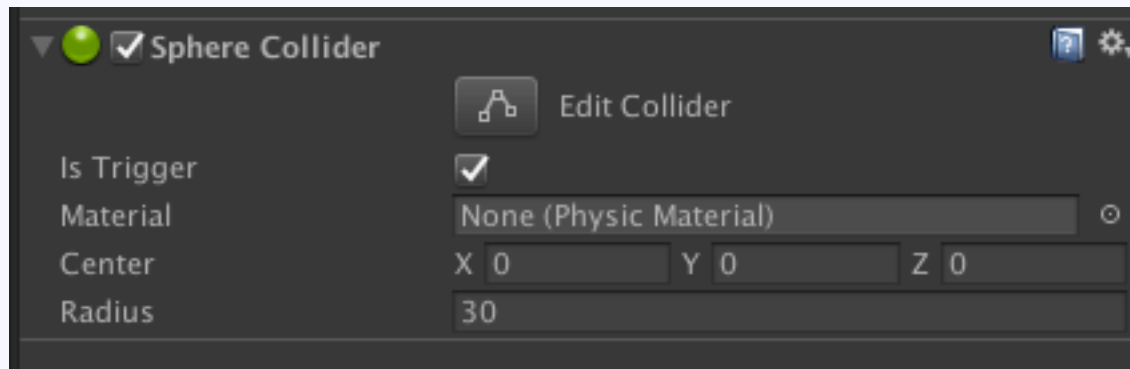
右にチェックポイントがあれば右にハンドルを切る

簡単そうだ。

チェックポイントはある程度の大きさのエリアで判定する



SphereColliderを使った



IsTriggerにチェックを入れておけば、このコライダーとぶつかっても物理挙動は行わなくなる。スクリプトでぶつかったかの判定ができるので、エリア判定などによく使われる。

車体が向いている方向より、右にいるか左にいるかは、**外積**を使えば求めることができる。

外積の計算はUnityが用意している関数を使えばOK。

具体的には、

目標地点をTargetとする

// 自分の位置と目標位置の差ベクトルを求める

**Vector3** targetDir = Target.transform.position - transform.position;

差ベクトル = 目標位置 - 自分の位置

// 自分の前方向と、上記の差ベクトルの外積をとる(右にいるのか左にいるのかを調べるため)

**Vector3** axis = **Vector3**.Cross(transform.forward, targetDir);

これだけ。

求めた結果の「**axis.y**」が

**マイナスなら左**

**プラスなら右**

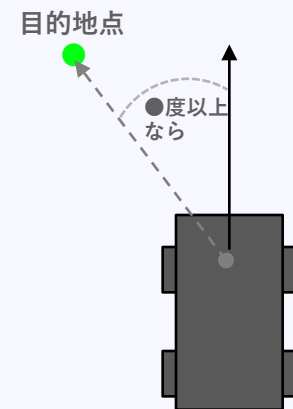
となる。

しかし、ただ右側にあるからハンドルを右に、左なら左にとやるだけでは、かなりハンドル操作が荒い運転になってしまう。

どれだけ左にあるか合わせて、ハンドルを切る量を調整するのがベスト。

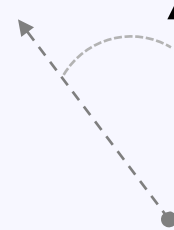
しかし、今回の設計では、ハンドルを切る量を調整するようなインターフェースを用意していない・・・。

なので、簡易実装として、目標物との角度が●度以上ならハンドル操作をするという実装にする。





Vector3.Angleという関数を使えば、  
2ベクトルの角度を求めることができる。



// 自分の位置と目標位置の差ベクトルを求める

```
Vector3 targetDir = m_targetPositions[m_targetNo].position - transform.position;
```

// 自分の前方向と、上記の差ベクトルから、角度の差を求める

```
float angle = Vector3.Angle(transform.forward, targetDir);
```

車体前向きのベクトル

車体の位置から目標位置までのベクトル

```
if (angle > 3) {
```

3度以上なら  
ハンドル操作

```
    if (axis.y < 0)
```

```
    {
```

```
        m_carController.Left = true; // ステアリングを左に切る
```

```
    }
```

```
    else if (axis.y > 0)
```

```
    {
```

```
        m_carController.Right = true; // ステアリングを右に切る
```

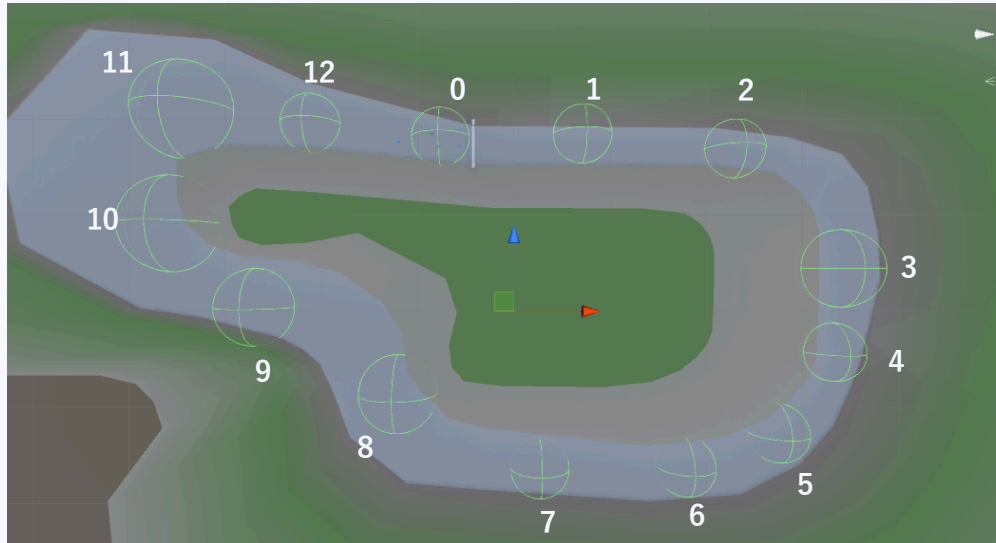
```
    }
```

```
}
```

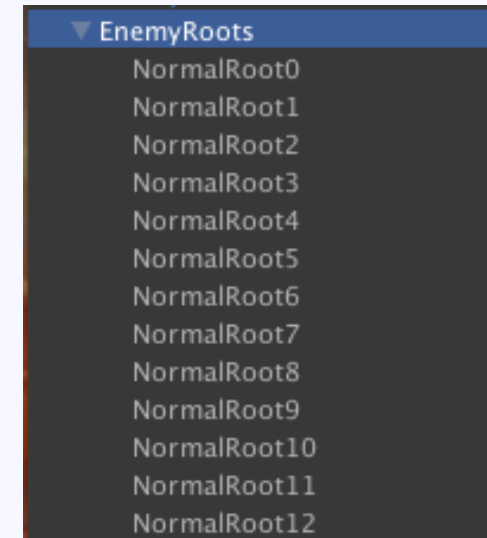
# 敵AI部のプログラム全貌

```
void Update () {  
    // 自分の位置と目標位置の差ベクトルを求める  
    Vector3 targetDir = m_targetPositions[m_targetNo].position - transform.position;  
  
    // 自分の前方向と、上記の差ベクトルの外積をとる(右にいるのか左にいるのかを調べるため)  
    Vector3 axis = Vector3.Cross(transform.forward, targetDir);  
  
    // 自分の前方向と、上記の差ベクトルから、角度の差を求める  
    float angle = Vector3.Angle(transform.forward, targetDir);  
  
    // とりあえずアクセル全開  
    m_carController.Forward = true;  
  
    if (angle > 3)  
    {  
        if (axis.y < 0)  
        {  
            m_carController.Left = true;           // ステアリングを左に切る  
        }  
        else if (axis.y > 0)  
        {  
            m_carController.Right = true;          // ステアリングを右に切る  
        }  
    }  
}
```

# チェックポイント通過判定



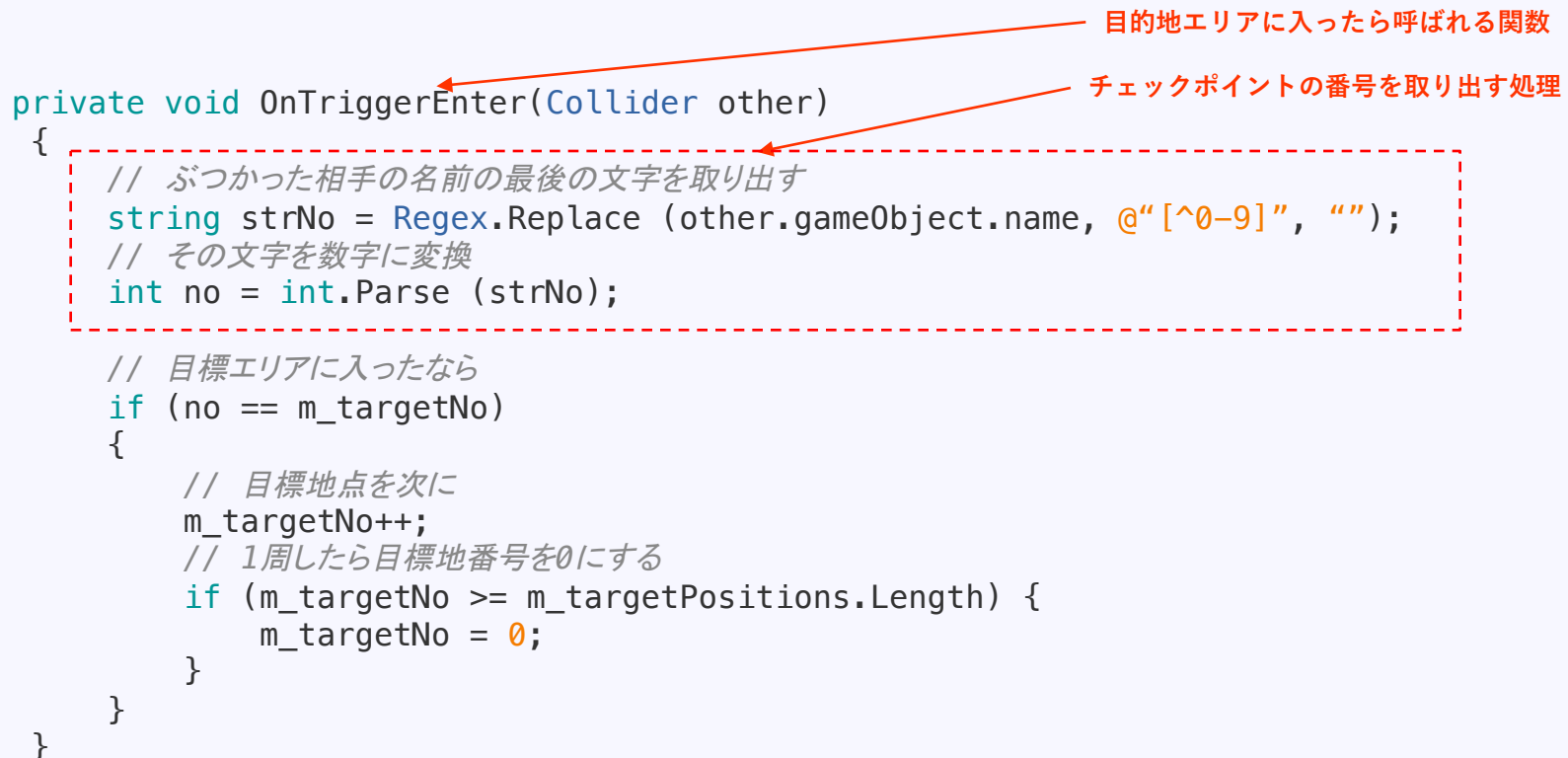
コース上に多数のチェックポイントを配置



名前の末尾に数字をつけた

- ・チェックポイントを配列として持つ
- ・目標チェックポイント番号を保持する変数を用意  
目標エリアに入るたびに、変数の値を増やし、  
次のチェックポイントに目標を切り替える。

OnTriggerEnterを使って、目標エリアに入ったかを判定。  
目標チェックポイント番号保持用変数の値と、に入ったエリアの  
番号が一致したら、次のチェックポイントに切り替える。



```
private void OnTriggerEnter(Collider other)
{
    // ぶつかった相手の名前の最後の文字を取り出す
    string strNo = Regex.Replace (other.gameObject.name, @"[^0-9]", "");
    // その文字を数字に変換
    int no = int.Parse (strNo);

    // 目標エリアに入ったなら
    if (no == m_targetNo)
    {
        // 目標地点を次に
        m_targetNo++;
        // 1周したら目標地番号を0にする
        if (m_targetNo >= m_targetPositions.Length) {
            m_targetNo = 0;
        }
    }
}
```

目的地エリアに入ったら呼ばれる関数

チェックポイントの番号を取り出す処理

ここまでの制作結果を実行してみよう！

## **Assets/Scenes/Race.unity**

をダブルクリックしてシーンを切り替え  
再生ボタンで実行してみてください。

キーボードの十字キーで操作します。

本日はここまで。

まだ不具合もあるし、順位や周回数もなし。  
次回以降で対応していきます。

ご清聴ありがとうございました