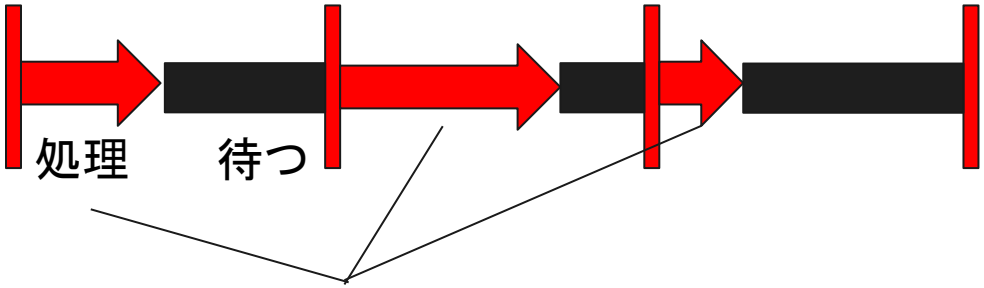
イベント関数の実行順序

イベント関数



処理（プログラム）

・オブジェクトごとにUnityが決めたメソッドを利用し、
　Unityが決めたルールでプログラムが動く

画像出力

例

・Update：1フレームに必ず1回実行される

・Start：GameObjectが生成された最初のフレームに1回だけ行われ、その1回だけ　　の実行は必ずUpdateより先に行われる

実行順番

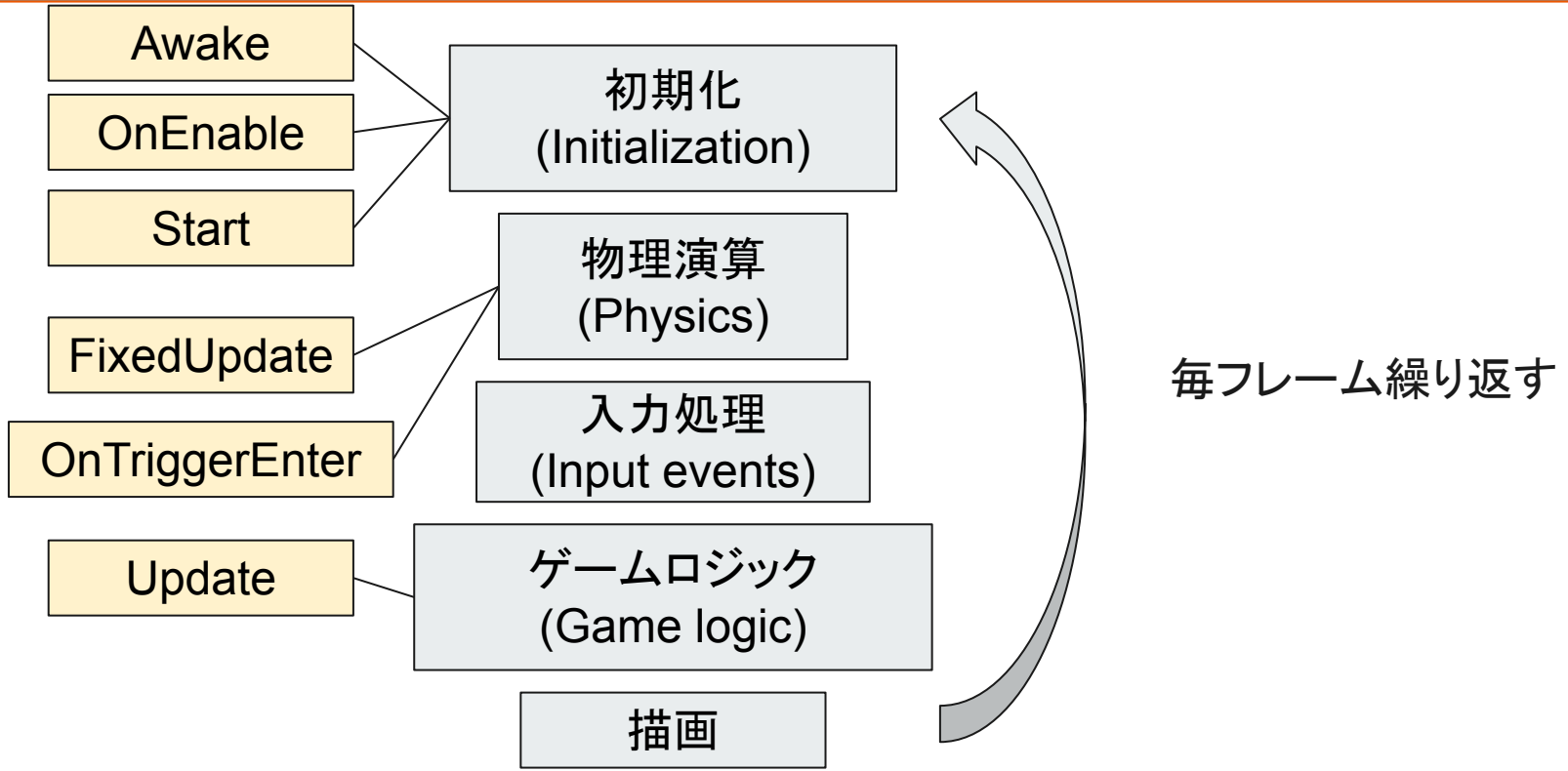
```
public class Test : MonoBehaviour
{
    void Start()
    {
        Debug.Log("スタート");
    }
    void Update()
    {
        Debug.Log("アップデート");
    }
}
```

```
public class Test : MonoBehaviour
{
    void Update()
    {
        Debug.Log("アップデート");
    }
    void Start()
    {
        Debug.Log("スタート");
    }
}
```

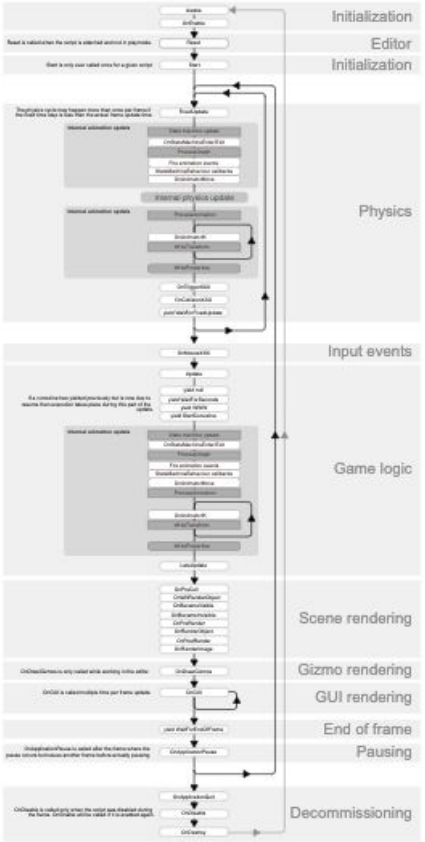
結果の表示のされ方は同じ

メソッドを書く順番は実行順番とは関係ない

イベント関数の実行順


Awake
OnEnable
Start
初期化
(Initialization)
物理演算
(Physics)
FixedUpdate
OnTriggerEnter
入力処理
(Input events)
Update
ゲームロジック
(Game logic)
描画
毎フレーム繰り返す

公式で確認

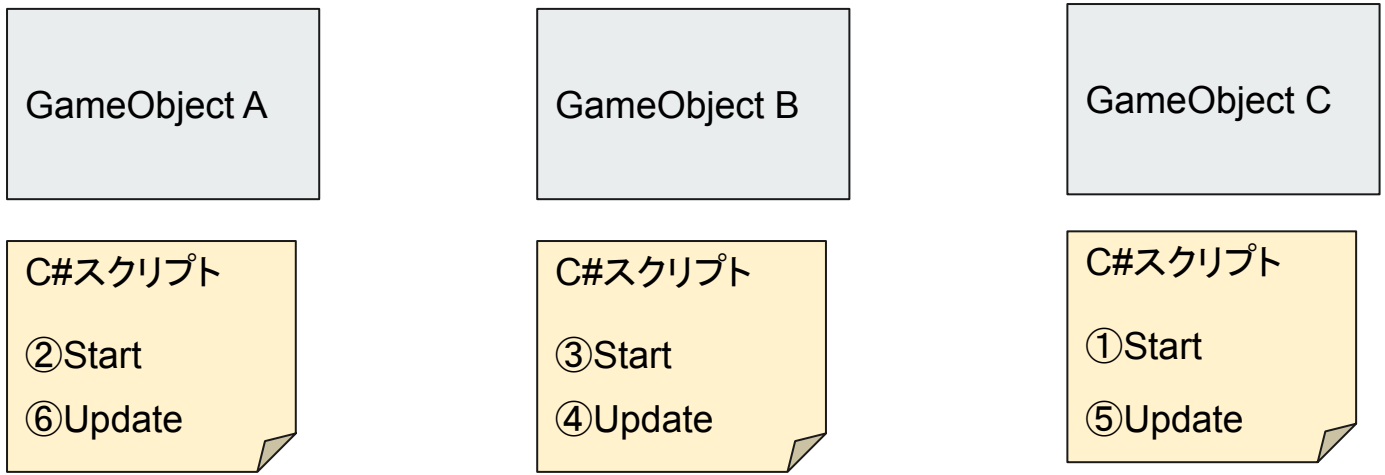


イベント関数の実行順に関して、

詳しくはUnityの公式サイトで確認してください

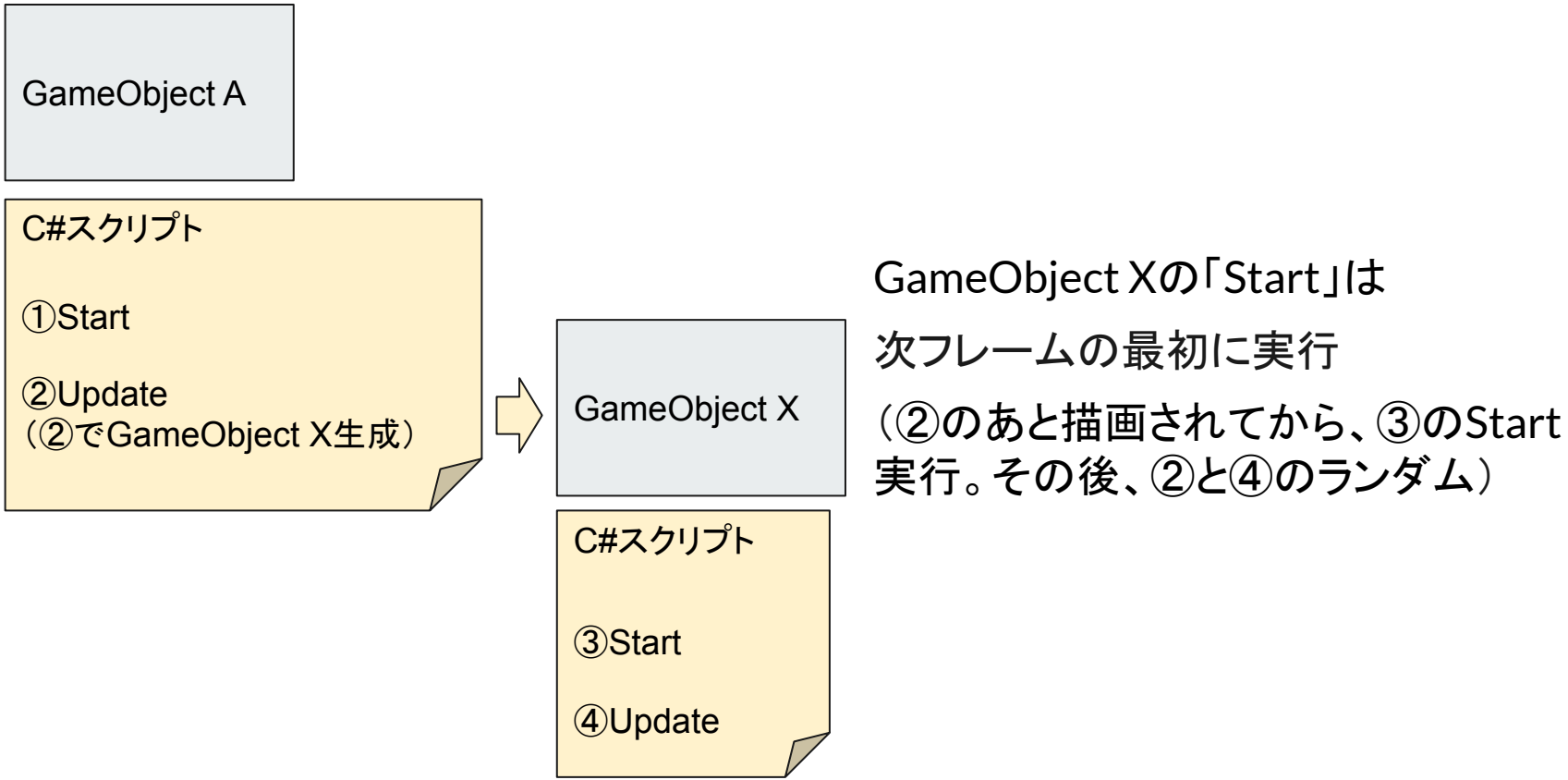
https://docs.unity3d.com/ja/2022.1/Manual/ExecutionOrder.html

実行順例

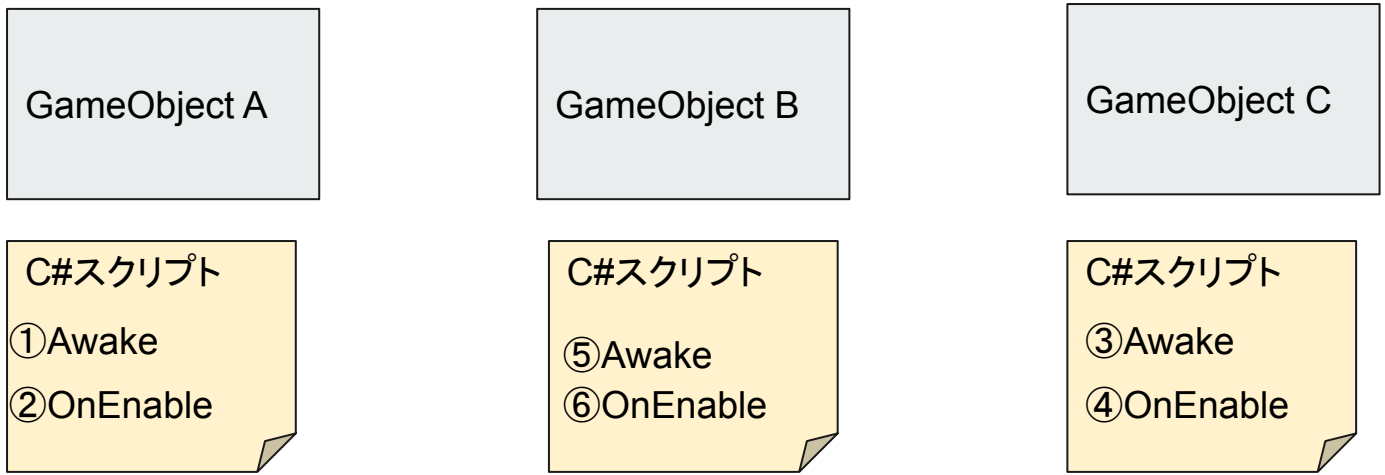


①～③,④～⑥はランダム

実行順

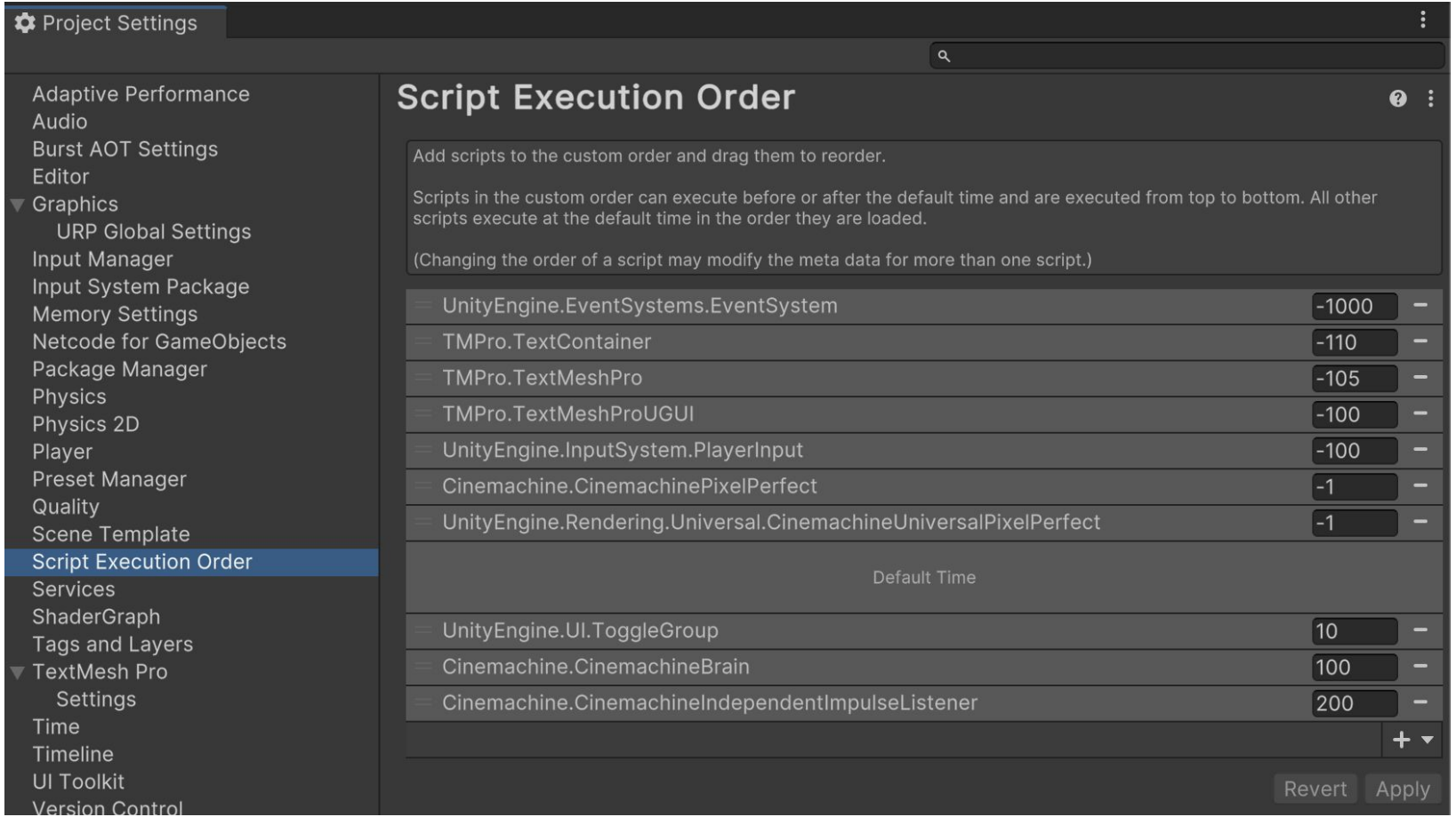

GameObject A
C#スクリプト
①Start
②Update
（②でGameObject X生成）
GameObject X
C#スクリプト
③Start
④Update
GameObject Xの「Start」は
次フレームの最初に実行
（②のあと描画されてから、③のStart
実行。その後、②と④のランダム）

実行順

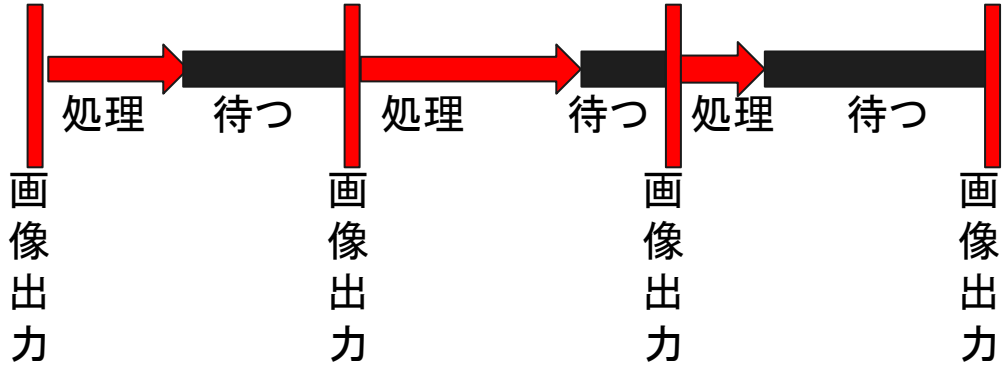


オブジェクト毎にセットで実行される

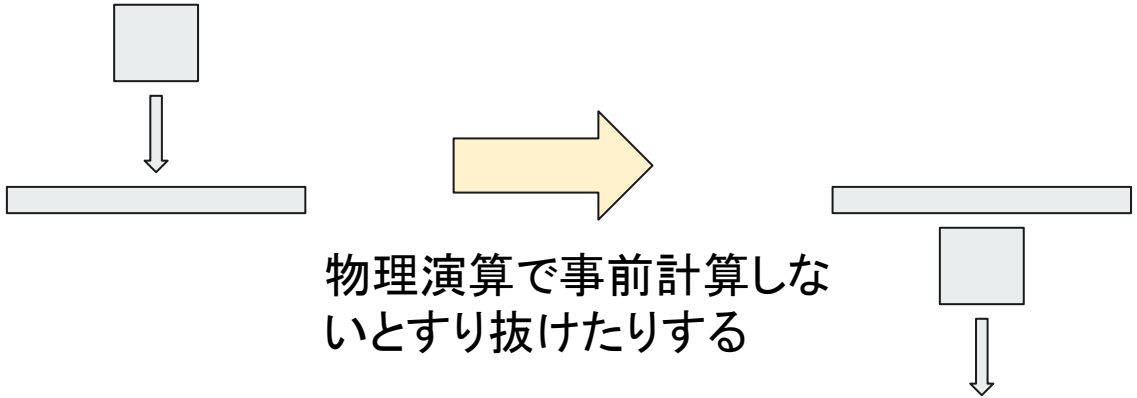
どうしても実行順を指示したい時



物理演算の必要性



パラパラ漫画　＝　「Transform」は、線の移動ではなく、座標の移動（瞬間移動）



物理演算で事前計算しないとすり抜けたりする

物理演算

物理演算とは物体の運動を物理法則（質量・速度・摩擦・風といった、古典力学的な法則）に基づいてシミュレーションすることです。物理シミュレーションとも呼ばれます。

Unityには3Dの物理演算と2Dの物理演算の両方が存在するので、3Dゲームでも2Dゲームでもあらゆる物理演算を表現することができます。現実に限りなく近い物理法則にすることもできますし、現実ではありえないようなゲームならではの物理法則にすることもできます。いかに物理演算を扱うことができるかが、ゲームのクオリティを左右することになります。

Unityでは物理演算エンジンとしてNVIDIAが開発しているPhysXが使われています。PhysXはリアルタイムの物理演算エンジンで、Unreal Engineでも使われています。2018年12月からオープンソースとしてGitHub上で公開されています。オープンソースへのアクセスは、NVIDIAの開発者向けページから可能です。

物理演算

Unityの「物理演算関連で」使用する主なコンポーネントは以下の3つ

・Rigidbody

・Collider

・Joint

Unityでオブジェクトを「操作する時」に使用する主なコンポーネントは以下の3つ

・Transform　　（物理演算考慮しない）

・Rigidbody　　（基本的に物理演算を考慮する）

・CharactorController（基本的なところだけ物理演算を考慮する）

Rigidbodyコンポーネントによる移動方法

物理演算が適用されるオブジェクト(剛体)はRigidbodyコンポーネントを使って動かします。オブジェクトに対して力や回転を加えたりする他、速度を直接操作することも可能です。

前述の通り、剛体はTransformコンポーネントを使った移動との相性があまり良くありません。剛体の座標を直接変更する場合は、Rigidbodyコンポーネントの MovePosition() を使いましょう。

Rigidbodyコンポーネントを取得&使用する

```
var rigidbody = GetComponent<Rigidbody>();
rigidbody.MovePosition(new Vector3(1, 2, 3));   任意の座標に瞬間移動させる
rigidbody.AddForce(transform.forward * 100);   任意の方向に力を加える
rigidbody.velocity = new Vector(10, 0, 0);   移動速度を直接変更する
```

Rigidbody

Rigidbodyコンポーネントを動かすためのメソッドやフィールドは他にもたくさん

https://docs.unity3d.com/ja/2021.3/ScriptReference/Rigidbody.html

CharactorControllerコンポーネントによる移動

ゲームオブジェクト全般で使えるTransformコンポーネントとRigidbodyコンポーネントに対して、CharacterControllerはキャラクターの操作に特化したコンポーネントです。接地判定や当たり判定の処理もセットになっていて、RigidbodyやColliderを使わなくて良いのが特徴です。

CharacterControllerでのキャラクター移動は、Move()かSimpleMove()のいずれかを使用します。

```
CharacterControllerコンポーネントを取得＆使用する
var characterController = GetComponent<CharacterController>();
characterController.Move(new Vector(1, 2, 3));
キャラクターを引数で指定した方向に移動させる（重力がかからない）
characterController.SimpleMove(new Vector(1, 2, 3));
キャラクターを指定方向に移動させる（空中に居るときは引数が無視され、代わりにキャラクターに対して重力がかかる）
```

CharactorController

https://docs.unity3d.com/ja/2019.4/ScriptReference/CharacterController.html

Transformコンポーネントによる移動方法

Transformコンポーネントを使うと、ゲームオブジェクトの位置・回転・スケールを直接操作可能です。物理演算を無視して直接操作しますので、このコンポーネントで物理演算が適用されるオブジェクトを移動すると予期しない動きになる場合があります。

物理演算が適用されるオブジェクトを移動させるときは、Rigidbodyコンポーネント（後述）を使用することをおすすめします。

```
MonoBehaviourを継承したスクリプトであれば、「transform」でTransformコンポーネントにアクセスできる
transform.position = new Vector(1, 2, 3);          ワールド座標を直接変更する
transform.localPosition = new Vector(1, 2, 3);     ローカル座標を直接変更する
transform.Rotate(new Vector3(0, 0, 10));           Z軸に対して10度回転させる
transform.localScale *= 3;                         大きさを現在の3倍にする
```

物理演算

物理演算を行う上で、「障害」があることをスライドを見てもらい、
これから理解してもらいます

その上で、「障害」の解決するために用いるメソッドを2つ紹介します

Update()とFixedUpdate()の使い分け

Tips

Update()とFixedUpdate()

Input・Transform・Rigidbodyなどを操作する場合、Update()とFixedUpdate()の使い分けに注意が必要です。どちらも1秒間に複数回呼ばれるメソッドですが、Update()はフレームごとに呼ばれ、FixedUpdate()は物理演算が行われる周期で呼ばれます。

この影響で、Input.GetButtonDown()など入力があった瞬間を判定するメソッドはFixedUpdate()だと複数回連続で反応してしまう場合がありますので、必ずUpdate()の中で使いましょう。

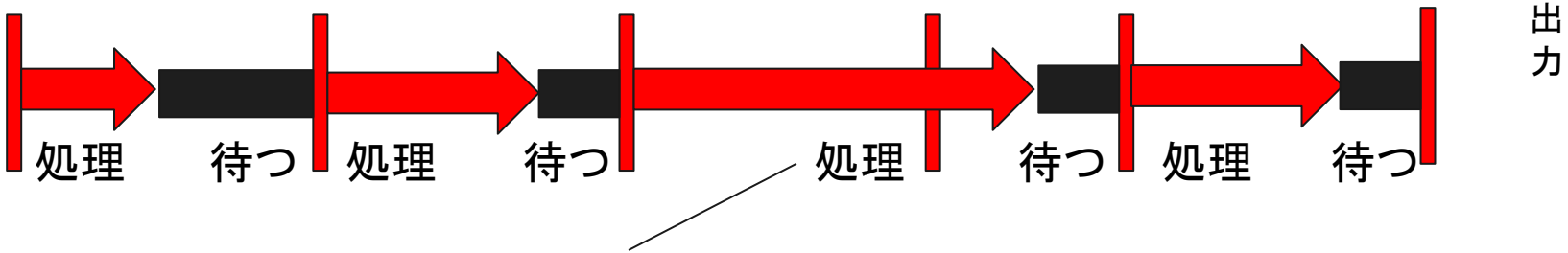
逆に、Rigidbodyに対する処理は、FixedUpdate()で使う方が望ましいです。

フレーム（処理落ち）

基本となる画像出力時間を設定

60FPSの場合　1フレーム　＝　1秒÷60fps　＝　0.0166秒

30FPSの場合　1フレーム　＝　1秒÷30fps　＝　0.0333秒



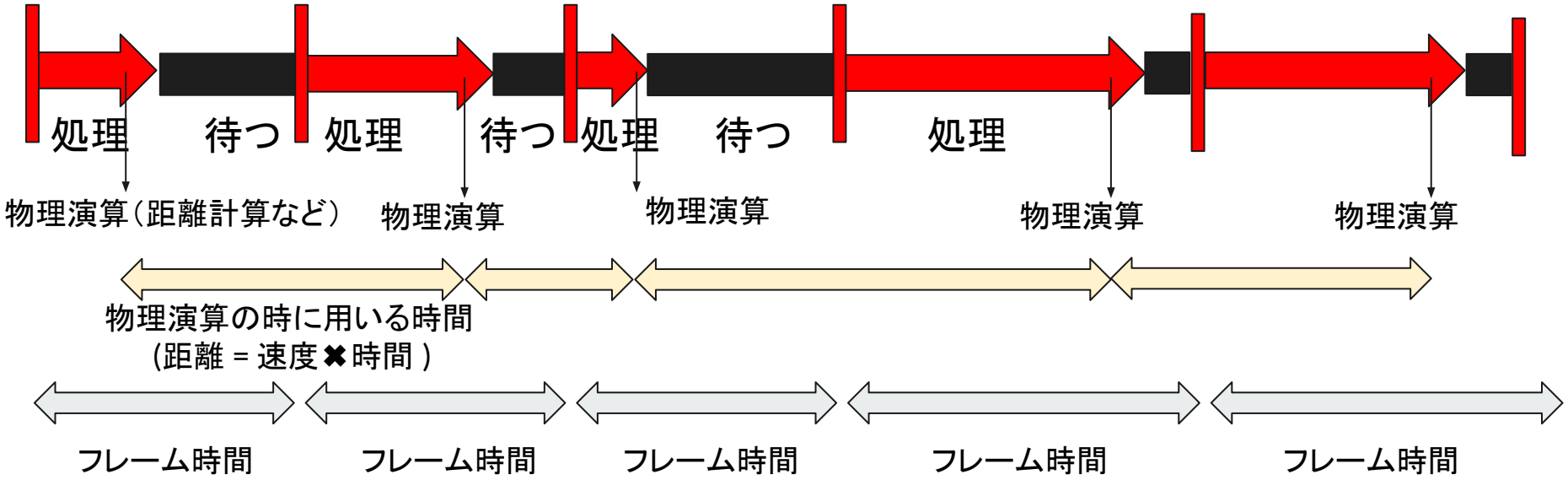
本来の画像出力タイミングでは出力はせず、画面が途切れてりしてフレームレートが一時的に自動的に落ちる。描画の間隔がズレる（フレーム落ち・処理落ち）

フレームの間隔は一定ではない

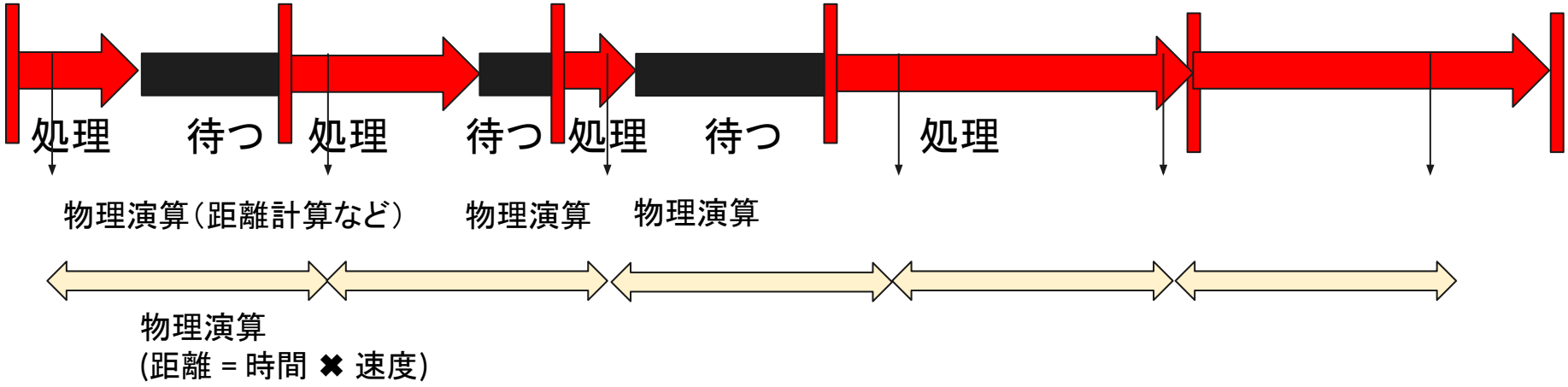
処理落ちなどの関係もあり、フレームの時間間隔というのは常に一定ではない

フレームの描画時間が違う（1フレームが1秒÷60fps=0.0166秒とは限らない）

➡ 物理演算(*物理法則に則った計算*)を1レームごとに計算すると結果が不安定になる



1つの解決策としてのFixedUpdate

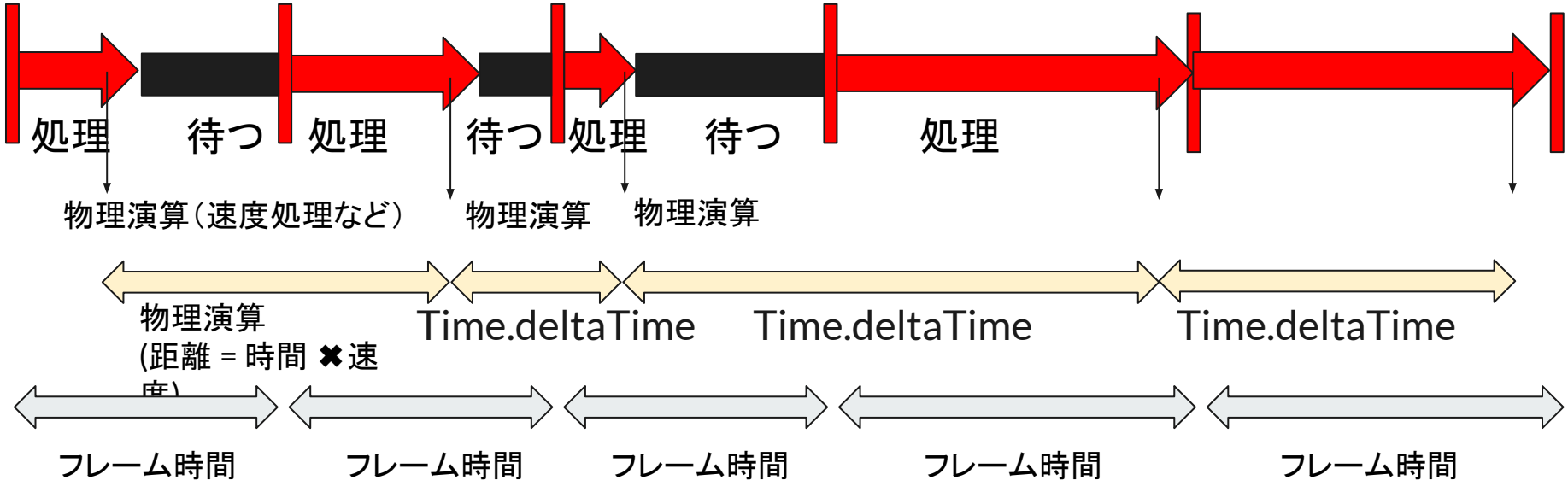


ゲーム上での時間において、常に一定の時間で呼び出されるメソッド

ゲーム上での時間において、一定間隔で処理されるメソッド

➡ Fixed Update()

1つの解決策としてのTime.deltaTime



FixedUpdate()は1フレーム2回だったり呼ばれなかったりするのでコントローラー入力などには向かない(入力はフレームごとで検知したい)

➡ Update()にてTime.deltaTimeを用いる

その他のメソッド

void Awake()

void Awake()は、ゲームオブジェクトが生成される際に、最初に一度だけ呼ばれます。ただし、生成されたゲームオブジェクトが無効(Inspecter左上にあるチェックがOFFになっている)だった場合は、有効になるまで呼ばれません。

void Start()

void Start()は、ゲームオブジェクトが生成されたあと、ゲームオブジェクトのUpdate()コールがはじまる前に一度だけ呼ばれます。

Start()メソッドは戻り値の型をvoidとIEnumeratorの2種類のいずれかを宣言できるという少し特殊なメソッドで、IEnumeratorにするとコルーチンとして実行されます(コルーチンの詳細は3-9参照)。

ゲームオブジェクトの初期化処理を行う場合は、Start()の中で実行することが多いです。

その他のメソッド

void Update()

void Update()は、ゲーム実行中、毎フレーム呼ばれます。

ゲーム中のキャラクターの動作やUIの更新などさまざまな処理に使いますが、時間のかかる処理（ゲームオブジェクトをたくさん生成するなど）を行うと処理落ちが発生してゲームプレイに支障が出ます。

Update()には時間のかかる処理を書かないようにしましょう。

void FixedUpdate()

Update()はフレームごとに呼ばれるのに対し、FixedUpdate()は物理エンジンの演算が行われるタイミングで呼ばれます（デフォルトの状態では、Update()よりもひんぱんに呼ばれます）。

物理演算に関係する処理はFixedUpdateに記載すると良いでしょう。

その他メソッド

void OnDestroy()

void OnDestroy()は、ゲームオブジェクトが破棄される際に呼ばれます。敵キャラクターが消えるとき一緒にライフゲージを消したりなど、後片付けによく使用します。

void OnEnabled()

void OnEnabled()は、ゲームオブジェクトが有効になる際に呼ばれます。

ゲームオブジェクト生成時のAwake()とStart()の間のタイミングで呼ばれますが、Awake()やStart()が1回だけしか呼ばれないのとは異なり、有効→無効→有効とすることで何度でも呼ばれます。なお、無効になる際はOnDisabled()が呼ばれます。

プログラムからGameObjectの有効／無効切り替え

```
using UnityEngine;
public class Sample : MonoBehaviour {
  [Header("オン・オフするゲームオブジェクトをセット↓")]
  public GameObject obj;
  void Start() {
    obj.SetActive( true ) ;  // 有効にする
    obj.SetActive( false );  // 無効にする
  }
}
```

プログラムからComponentの有効/無効を切り替え

```
using UnityEngine;
public class Sample : MonoBehaviour {
  [Header("オン・オフするボックスコライダーをセット↓")]
  public BoxCollider bc;
  void Start() {
    bc.enabled = false; // 無効にする
    bc.enabled = true; // 有効にする
  }
}
```

その他メソッド

void OnBecameInvisible()

void OnBecameInvisible()は、ゲームオブジェクトがカメラの撮影範囲から出た際に呼ばれます。

敵キャラクターが画面外に出たら消す処理などでしばしば使用します。なお、カメラの映す範囲に入った際はOnBecameVisible()が呼ばれます。