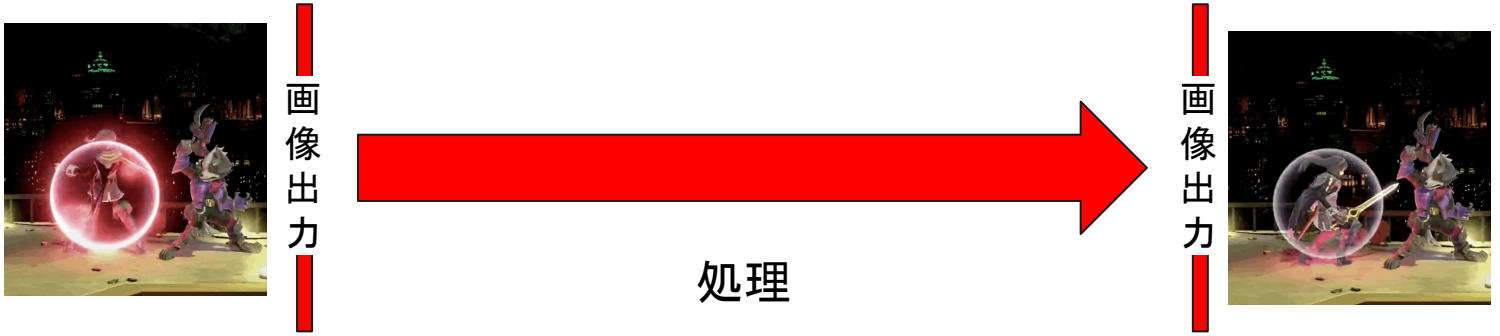
Unityにおけるフレーム

フレーム

実際のゲーム画面

パラパラ漫画のように画面が出力されている

フレーム



今回は、画像を出力してから次の画像を出力するまでを「1フレーム」と呼ぶ

・テレビや映画など：予めもしくは生で映している画像の連続

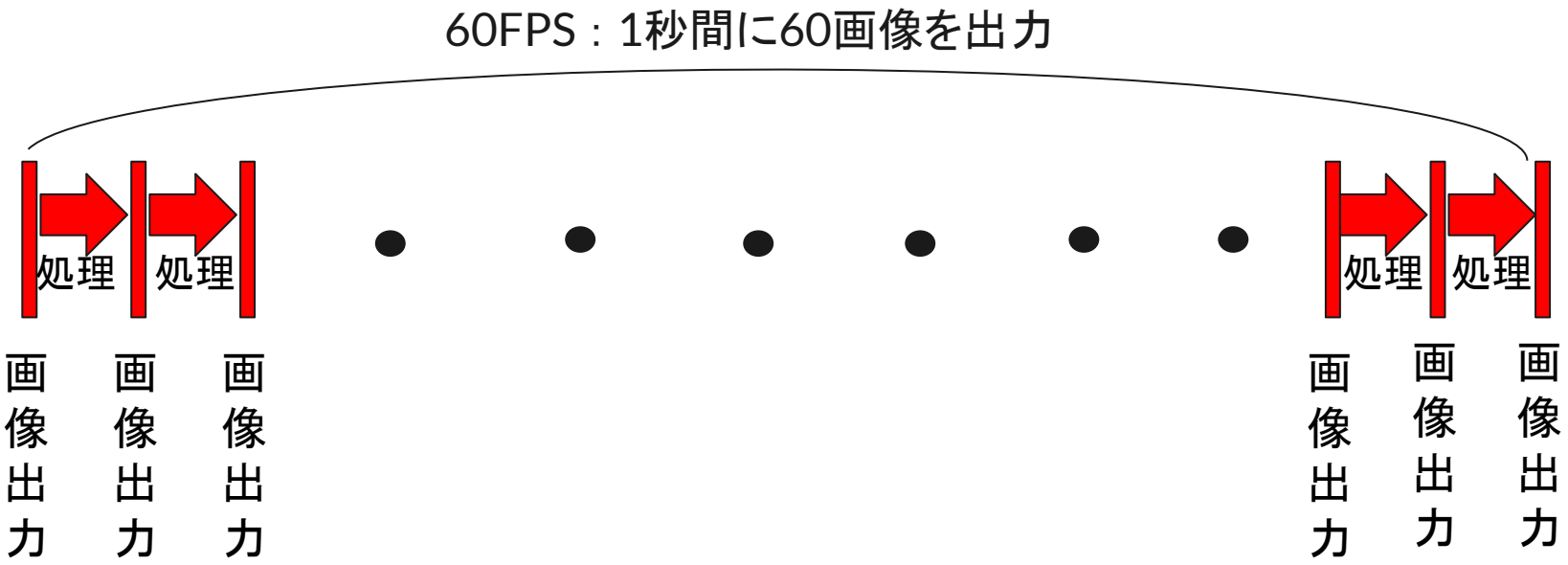
・ゲーム：次にどのような画面が映されるかは、プレイヤーの操作方法により
流動的に変わり、プログラムでの色々な処理作業が行われその結果　　によって
画像が作成されている

（処理時間などは機種の端末によってバラバラ）

フレームレート

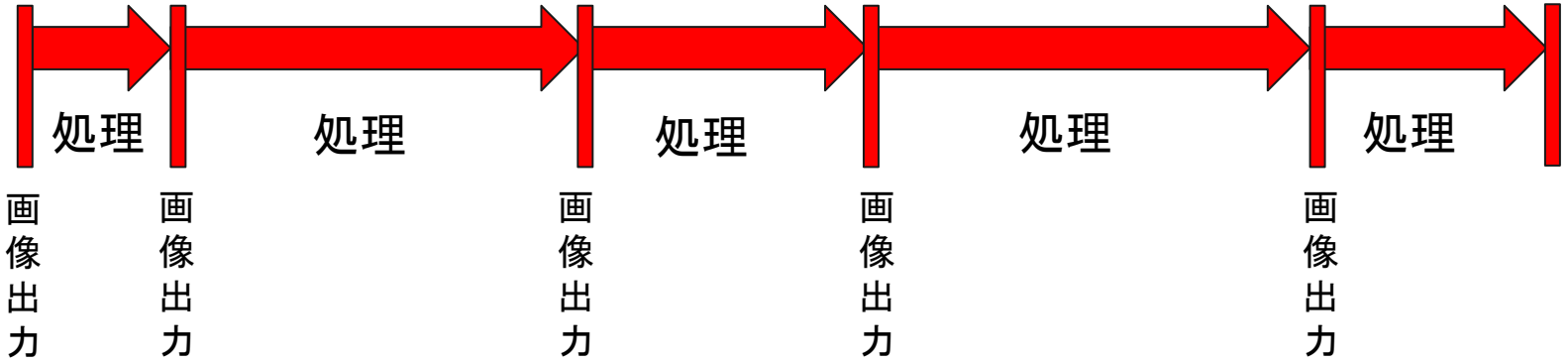
フレームレート＝1秒間に何回画像を出力できるかの目安

FPS（frame per second）とも呼ばれる



フレーム

プログラムの処理時間は状況などによってバラバラ、
いつも画像の出力タイミングがズレると映像に違和感が生まれる

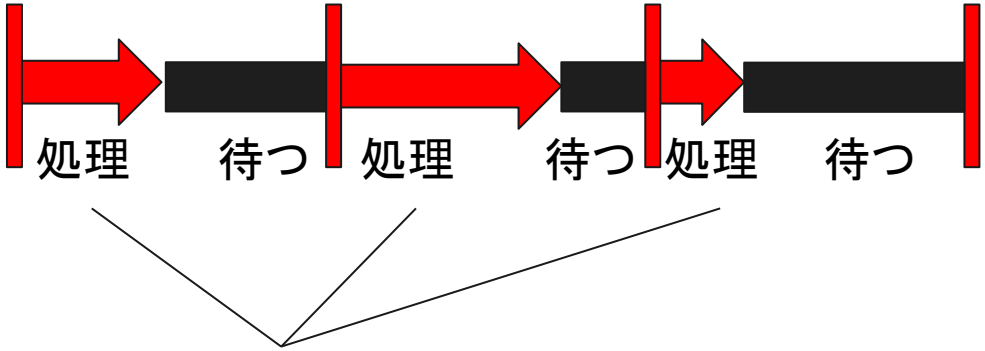


フレーム

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



Update処理はこの中で行われる

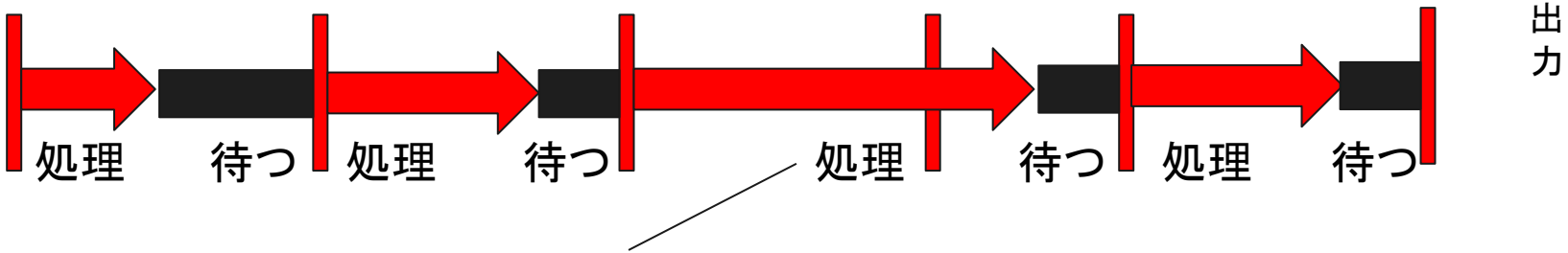
画像出力

フレーム（処理落ち）

基本となる画像出力時間を設定

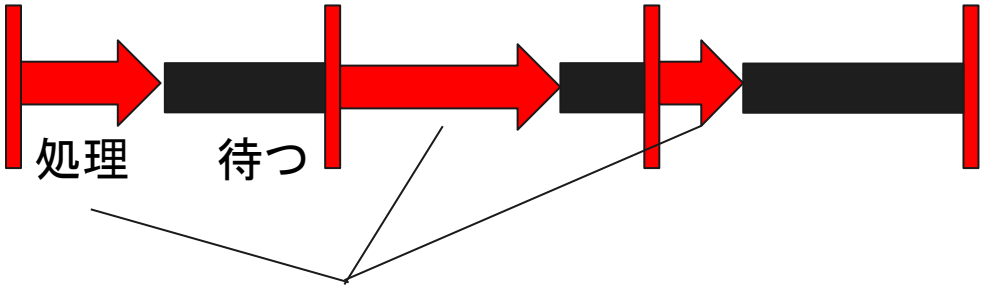
60FPSの場合　1フレーム　＝　1秒÷60fps　＝　0.0166秒

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）

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

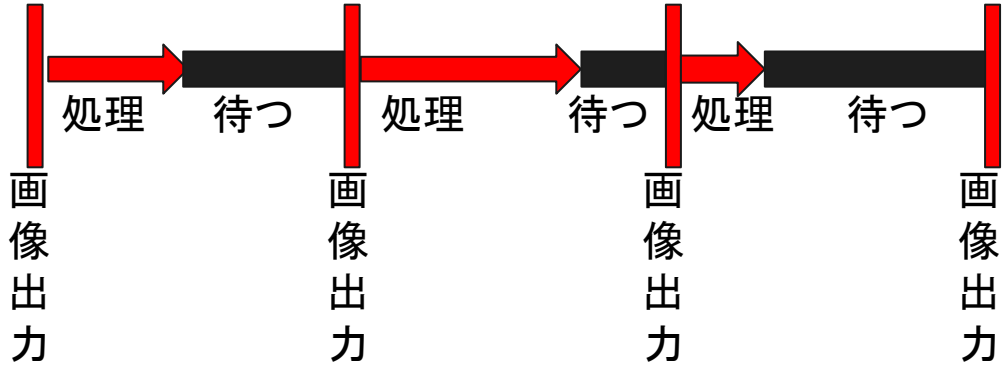
```
public class Test : MonoBehaviour
{
    void Start()
    {
        Debug.Log("スタート");
    }
    void Update()
    {
        Debug.Log("アップデート");
    }
}
```

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

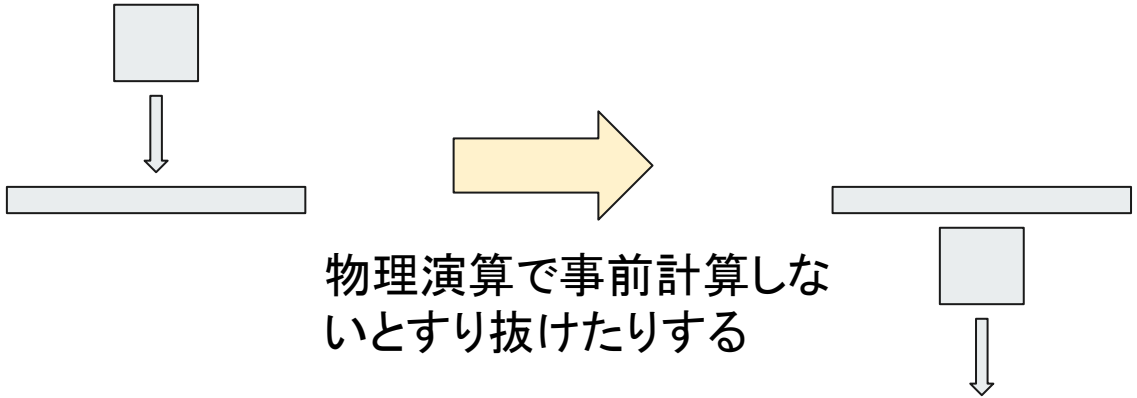
結果の表示のされ方は同じ

メソッドを書く順番は実行順番とは関係ない

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。

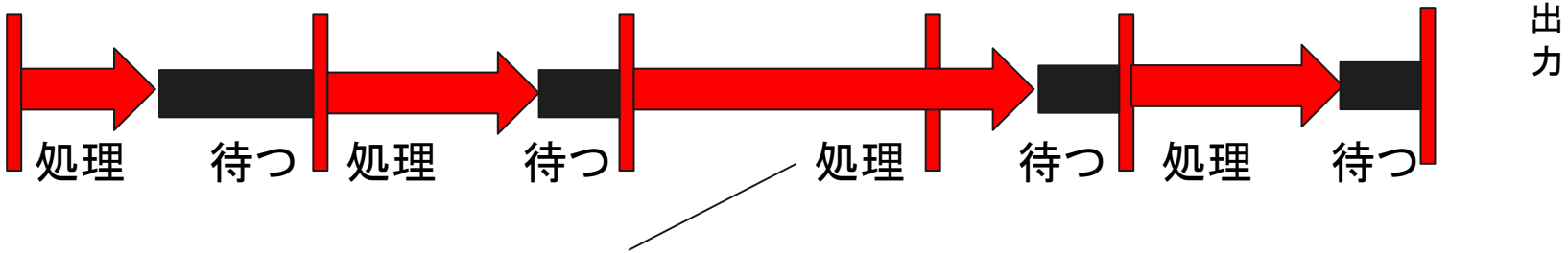
逆に、Rigidbodyに対する処理は、FixedUpdate()で使う方が望ましいです。

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



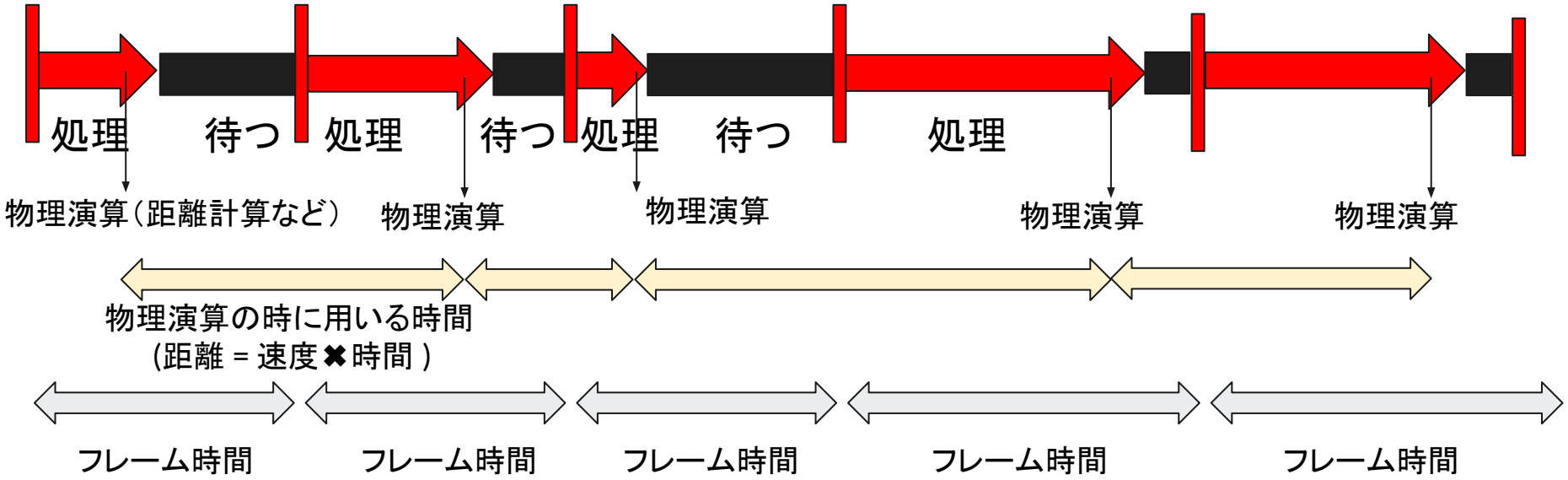
本来の画像出力タイミングでは出力はせず、画面が途切れてりしてフレームレートが一時的に自動的に落ちる。描画の間隔がズレる（フレーム落ち・処理落ち）

フレームの間隔は一定ではない

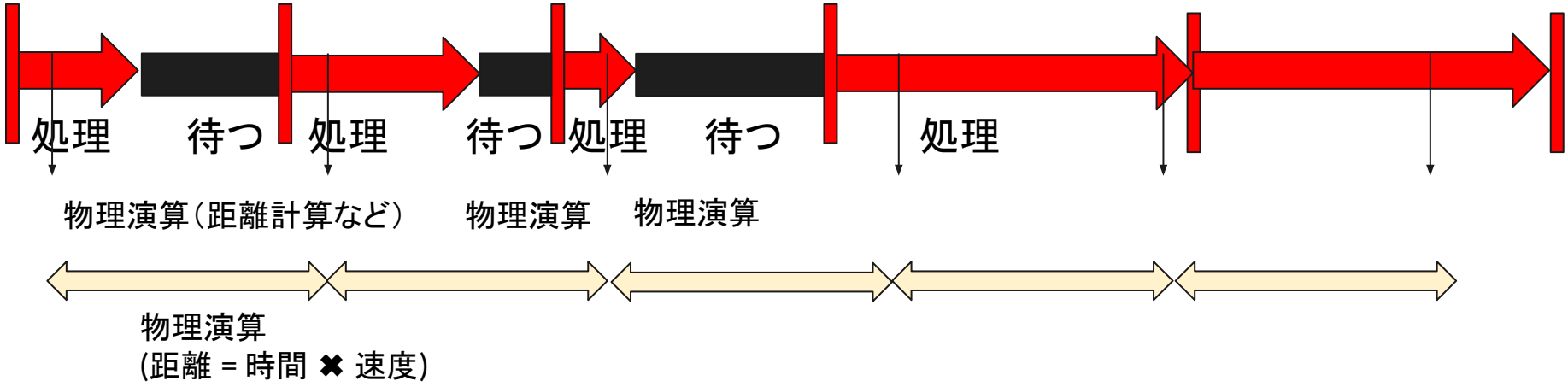
処理落ちなどの関係もあり、フレームの時間間隔というのは常に一定ではない

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

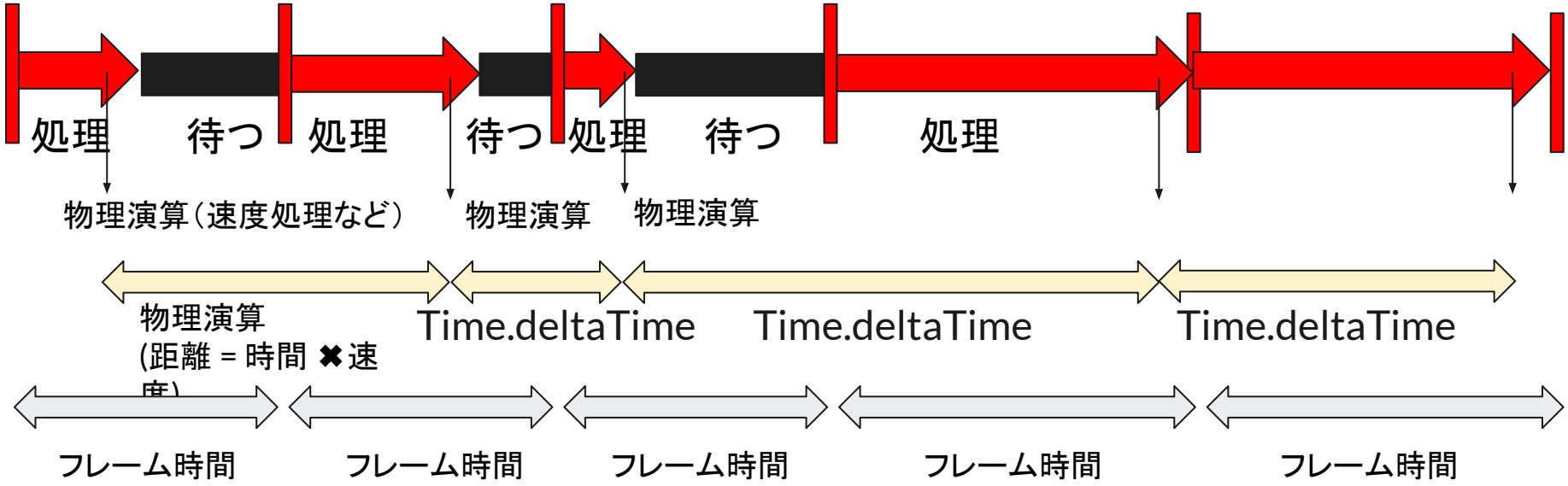


ゲーム上での時間において、常に一定の時間で呼び出されるメソッド

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

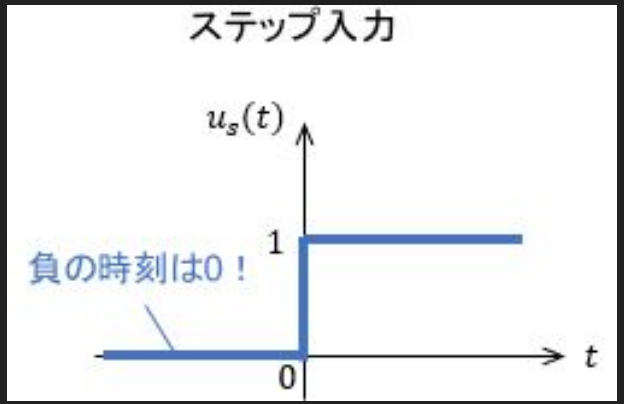


FixedUpdate()は1フレーム2回だったり呼ばれなかったりするのでコントローラー入力などには向かない(入力はフレームごとで検知したい)

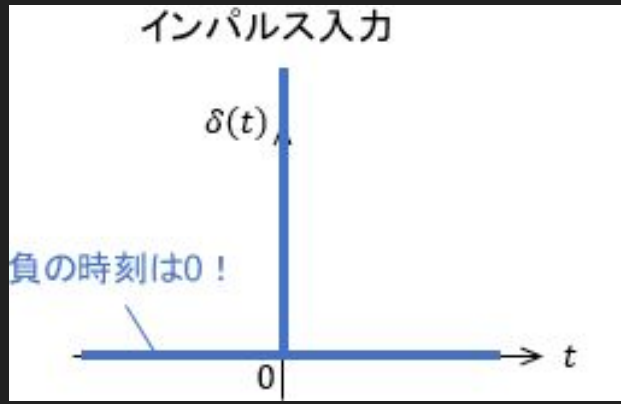
➡ Update()にてTime.deltaTimeを用いる

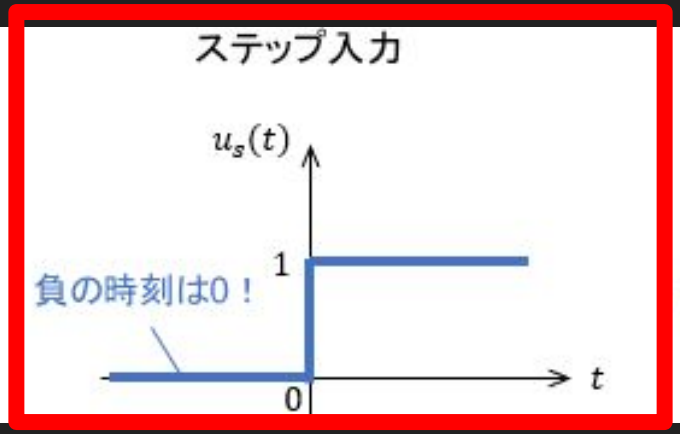
Update
FixedUpdate
物理演算

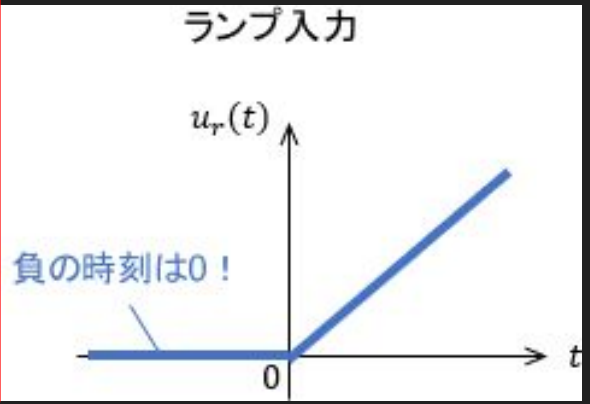

押した
押した
押した
押した
離した
離した
離した
時間

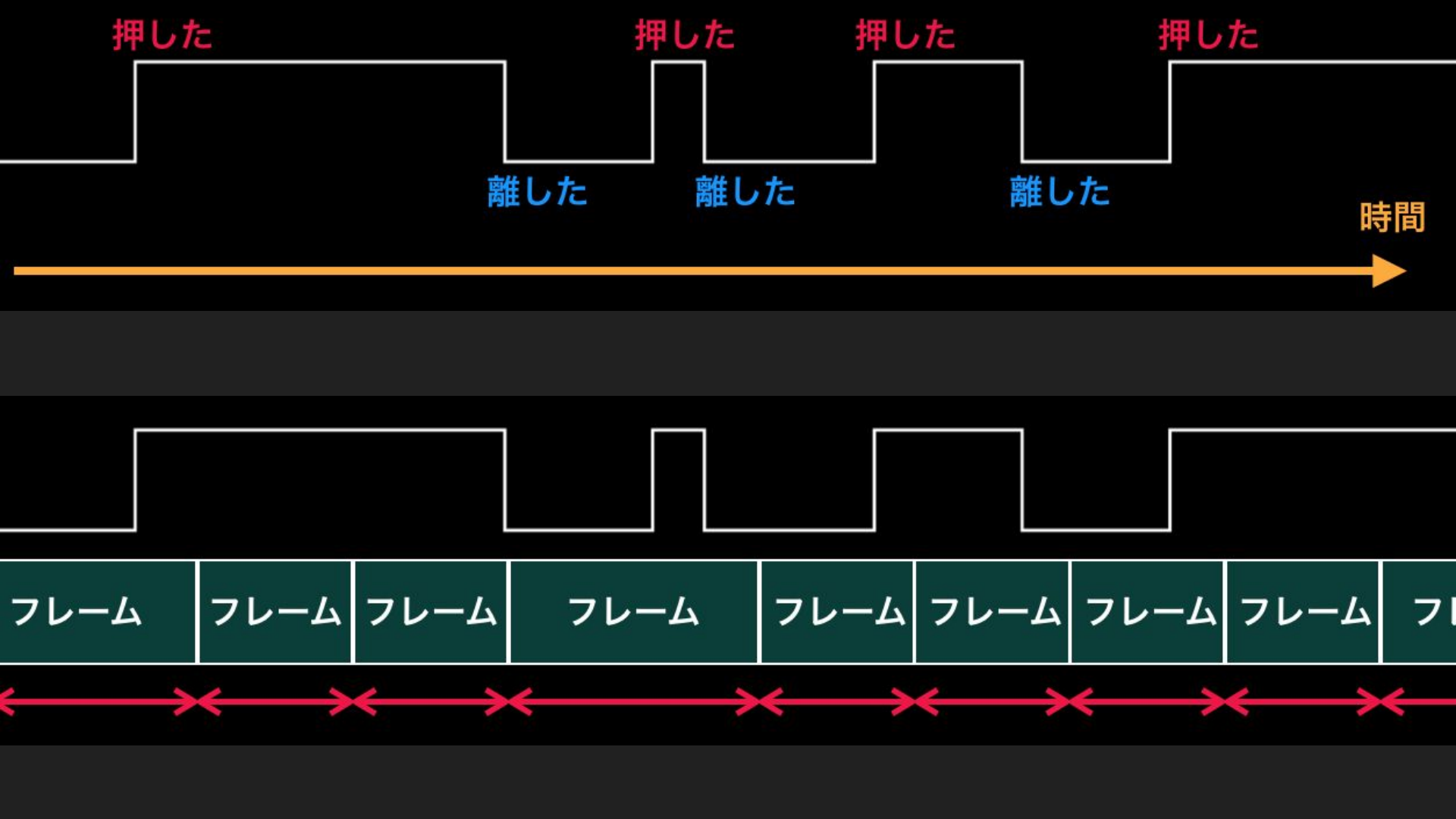

ステップ入力
$u_s(t)$
1
負の時刻は0！
0
t

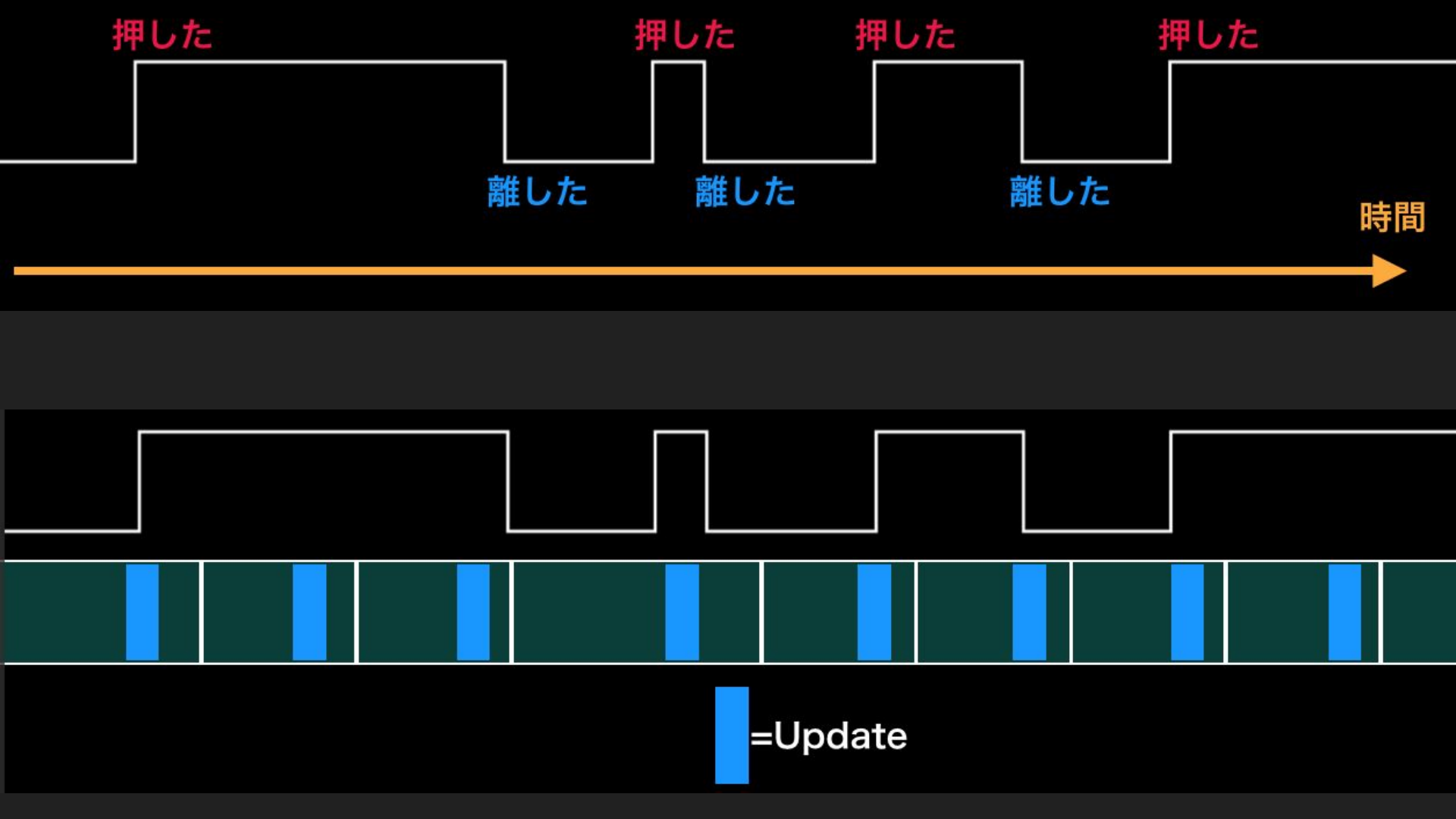

押した
押した
押した
押した
離した
離した
離した
時間

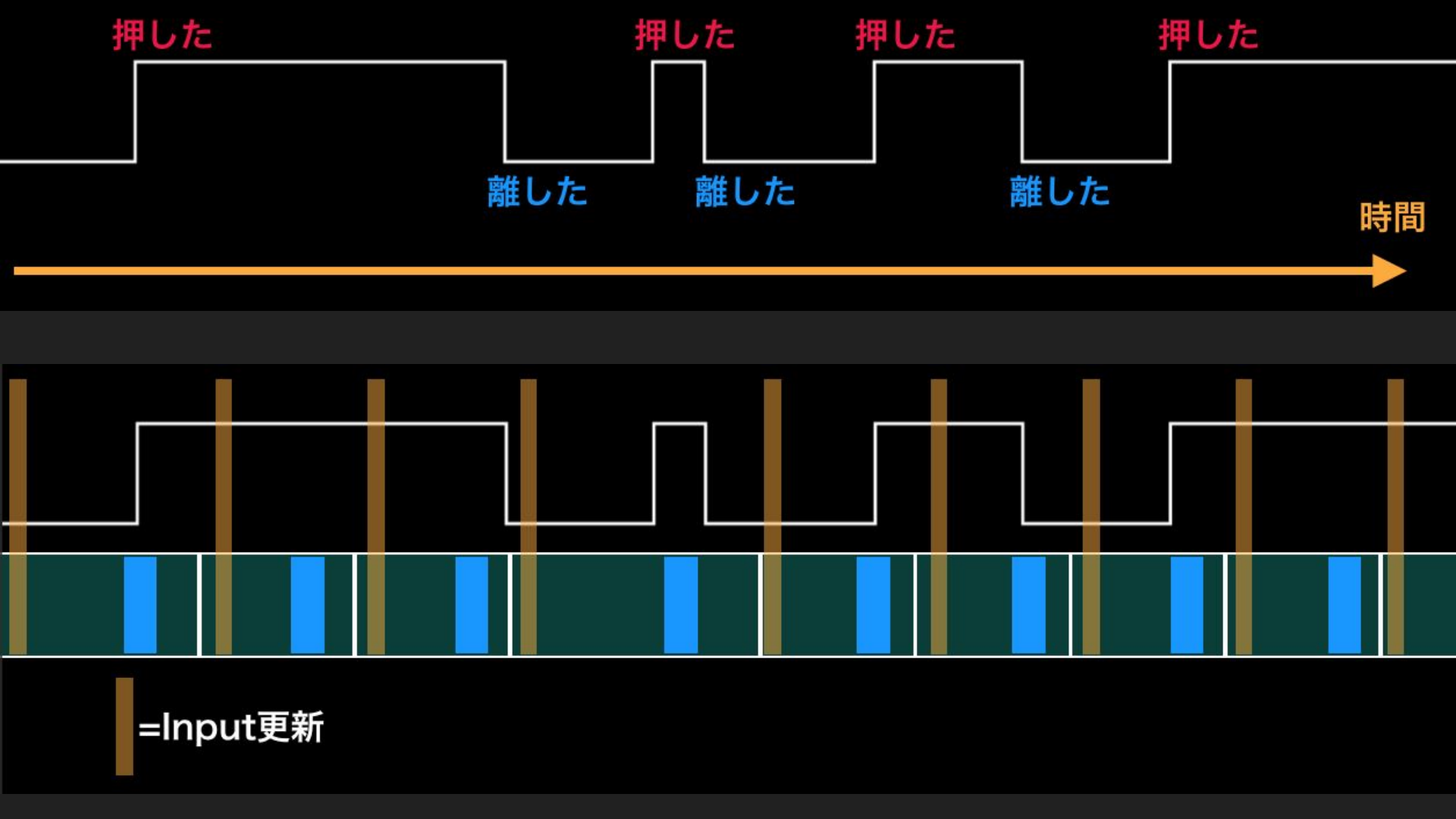

インパルス入力
$\delta(t)$
負の時刻は0！
0
t

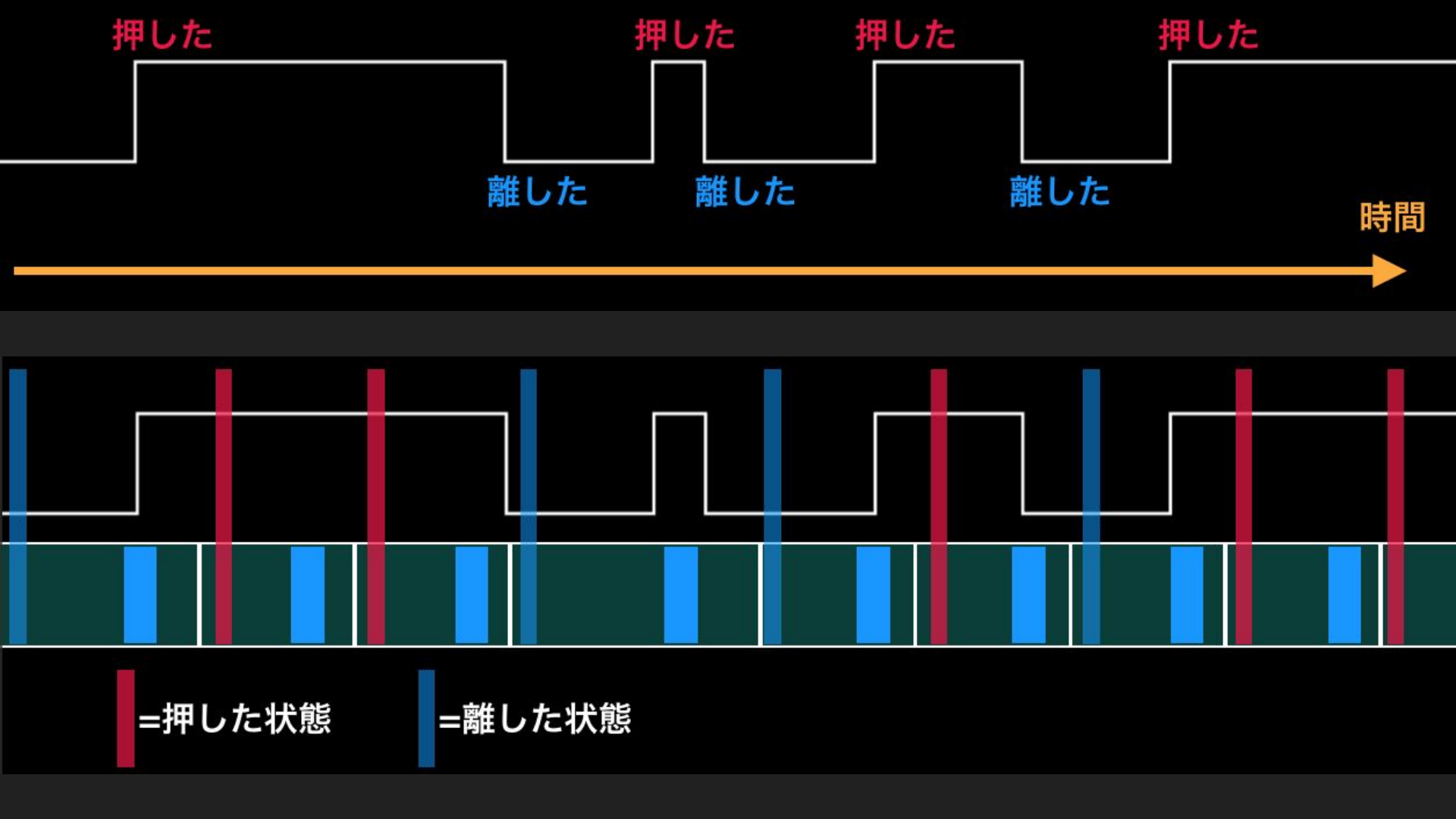

ステップ入力
$u_s(t)$
1
負の時刻は0！
0
t

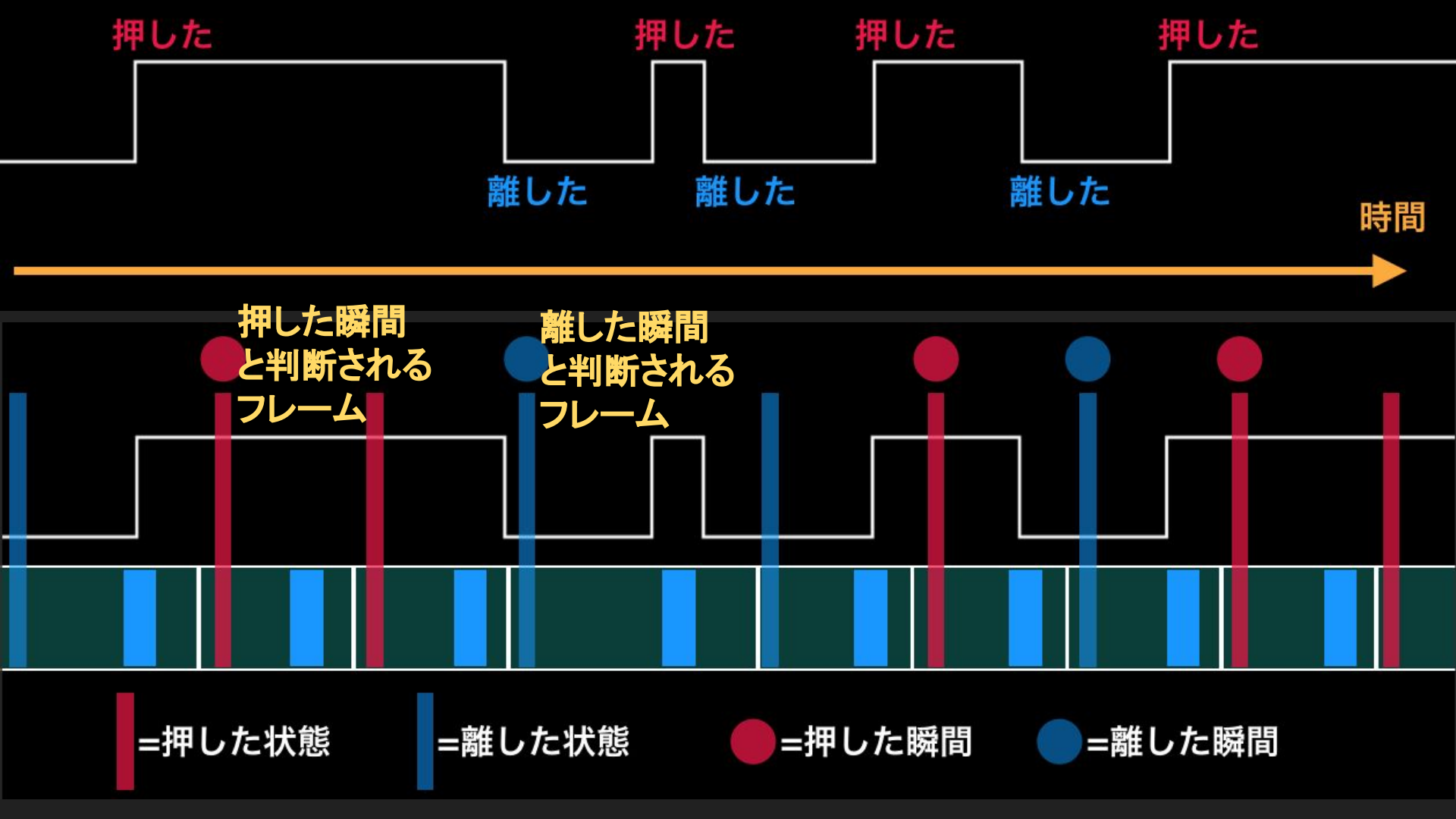

ランプ入力
$u_r(t)$
負の時刻は0！
0
t

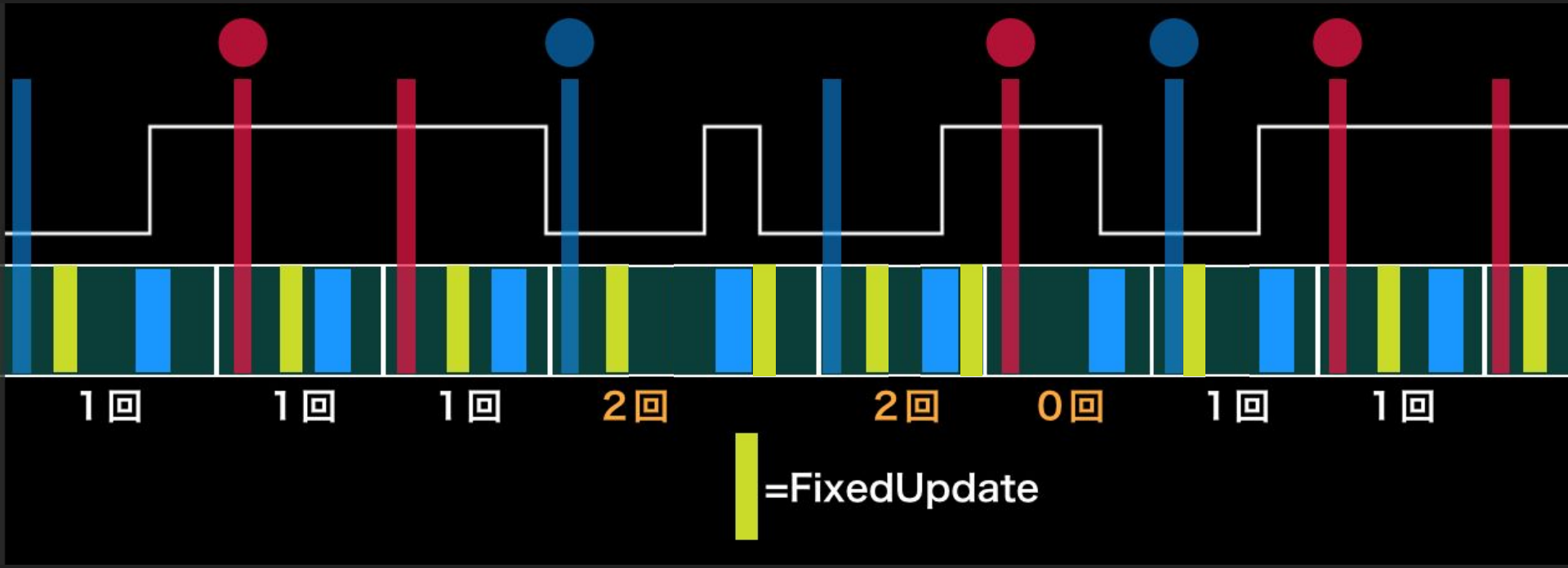

押した
押した
押した
押した
離した
離した
離した
時間
フレーム
フレーム
フレーム
フレーム
フレーム
フレーム
フレーム
フレーム


押した
押した
押した
押した
離した
離した
離した
時間
=Update

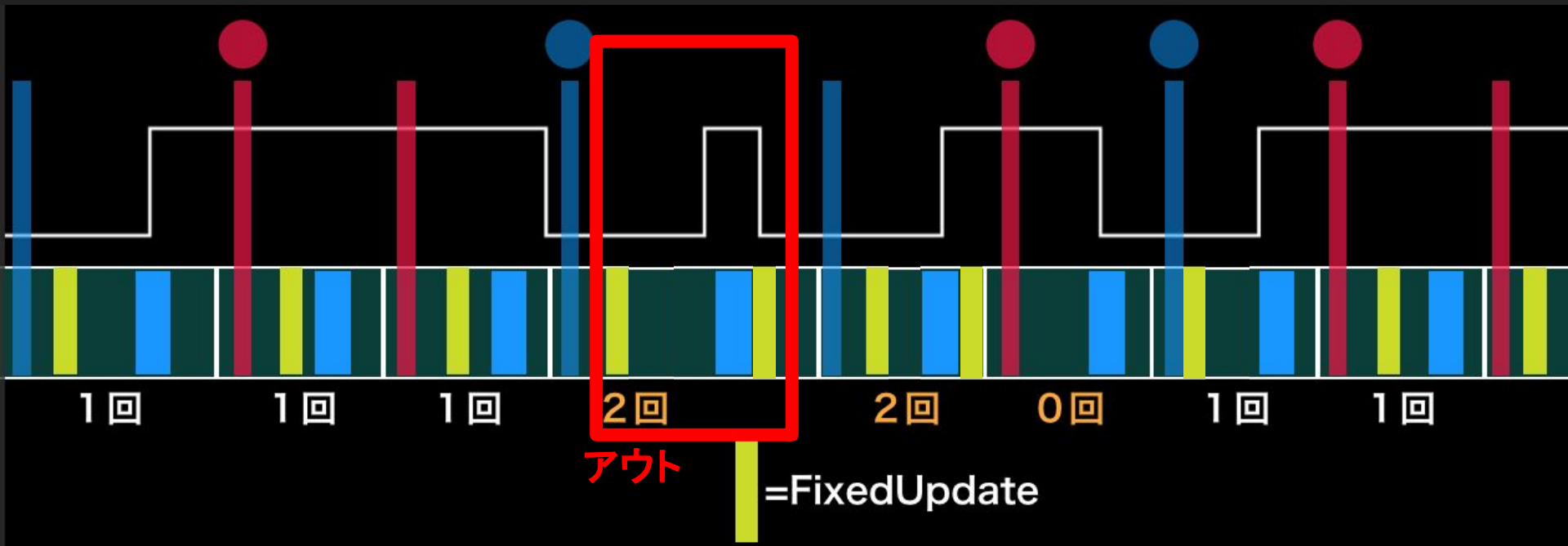

押した
押した
押した
押した
離した
離した
離した
時間
=Input更新


押した
押した
押した
押した
離した
離した
離した
時間
=押した状態
=離した状態

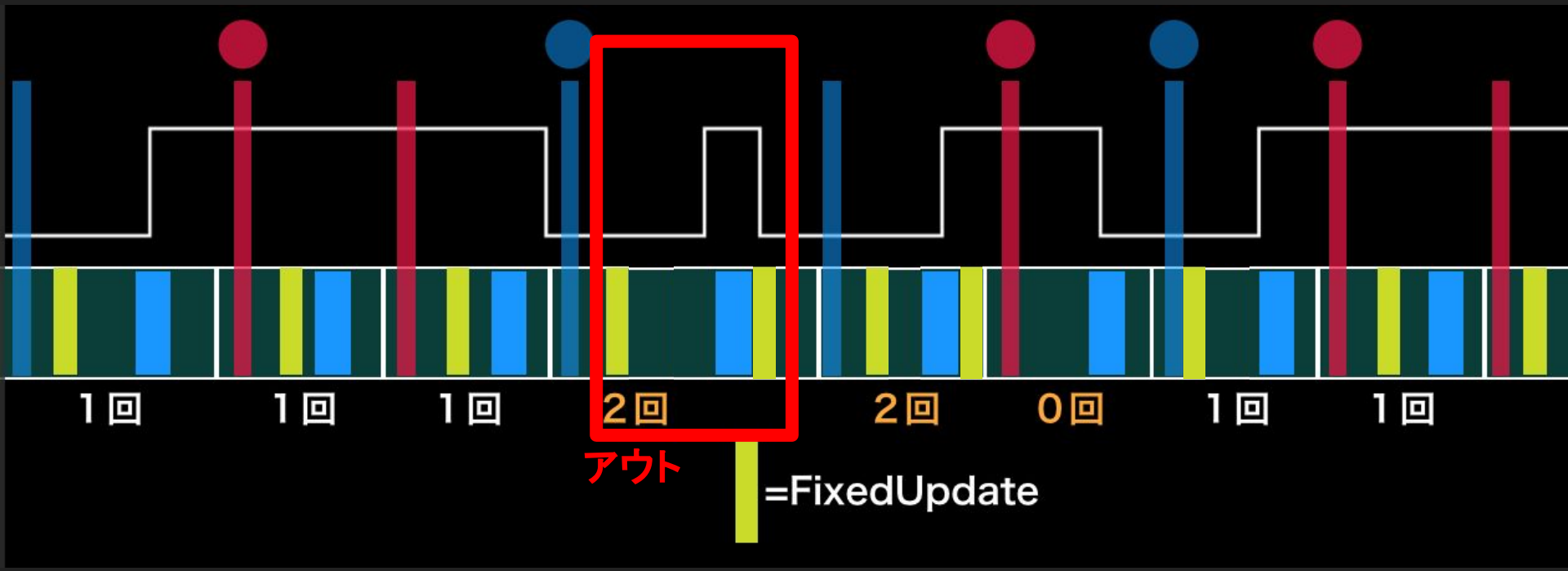

押した
押した
押した
押した
離した
離した
離した
時間
押した瞬間
と判断される
フレーム
離した瞬間
と判断される
フレーム
=押した状態
=離した状態
=押した瞬間
=離した瞬間



Update　　＝　1フレーム(描画と描画の間隔)に1回処理が行われる
FixedUpdate＝ゲーム内時間一定ごとに処理を行う(現実時間とは違う)



FixedUpdateの関数中では、押し下げた瞬間のInput.GetButtonDown()が2回連続でtrueを返すことがありえてしまう。
1回しか押してないのに弾が2発出てしまう。
0回のこともあるので、そのときは「押しても反応しない」という、ゆゆしき問題が発生する。



このような理由で、FixedUpdateではInput.Getナントカ Down を呼ぶな、となる。そしてまったく同じ理屈で、押している状態を検出する **Input.GetButton などは、別にFixedUpdateで呼んでも問題は起きません。**
トラブルが起きるのは「瞬間を返す関数をFixedUpdateで呼ぶこと」に限定される。

AddForce

AddForceの話をします。こいつは ForceMode の引数に応じてぜんぜん別の物理効果を発生させます

```
AddForce(force, ForceMode.Force);
```

この ForceMode.Force は **外力** を意味します。（第2引数のForceMode.Force を省略した場合もこちらになる）

```
AddForce(force, ForceMode.Impulse);
```

この ForceMode.Impulse は **力積** を意味します。

物理シミュレーションについての基本的な理解として、
Δt がすごく小さいことを、物理シミュレーションを実装した人は切実に期待している
ということを覚えておくと良いです。

Δtが小さいとはどういうことかというと、ごくわずかな、計測することすら困難な時間、力を加え、その後、あらゆる効果が数フレームにまたがって実行されて欲しい、ということです。

しかし現実はそんなに甘くなく、Δtはとても大きい。1/60秒とか、やってらんないほどでかい値になっています。

なので仕方なく、瞬間的な力（つまり力積）を表すために特殊な実装が施されたForceMode.Impulseが用意されています（実際にゲームにおいては極めて便利でもありますけど）。

すなわち、以下の２点を守る必要があります。

- AddForce(force, ForceMode.Force) つまり外力を「押した瞬間」で呼ぶべきではない（それが困るからImpulseが用意されている）
- AddForce(force, ForceMode.Impulse) つまり力積を「数フレームにまたがって」呼ぶべきではない（それをしない約束でImpulseが用意されている）

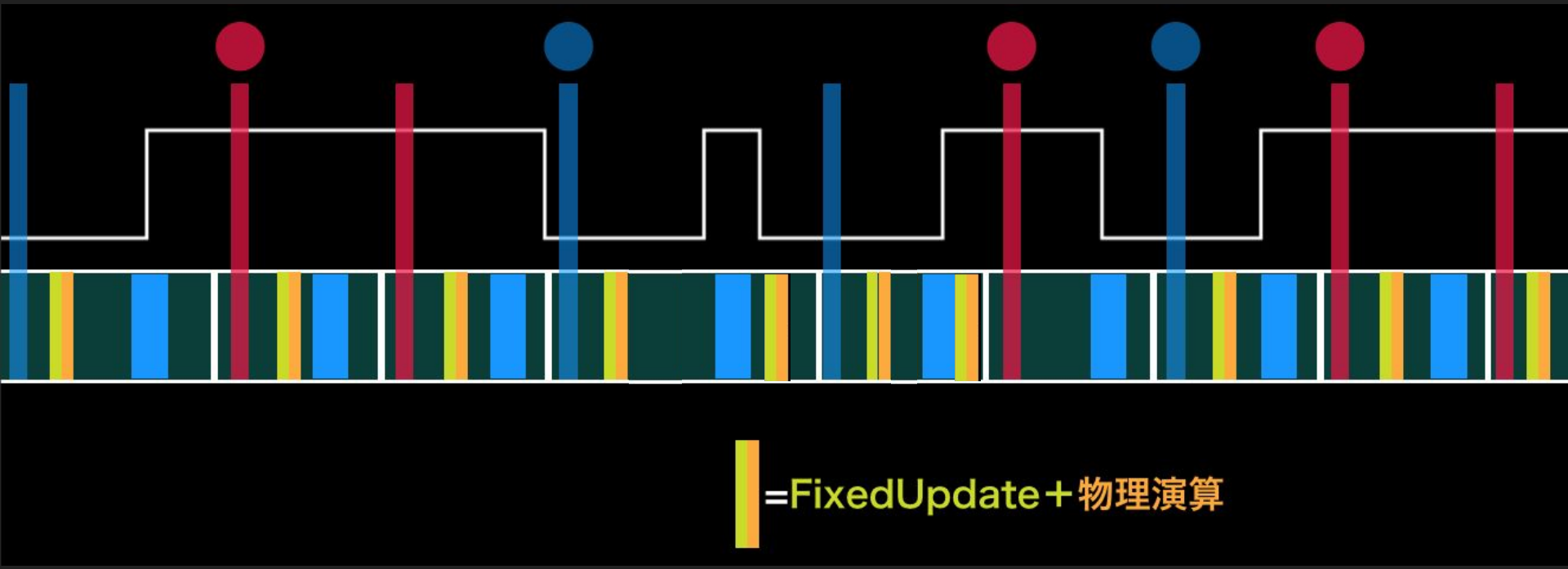
上記の理屈がわかりにくいと思われた方は、こう考えてみてください。運動方程式から考えると明らかですが、

外力Forceは速度への変換にΔtを使用しますが、 $\Delta v = \frac{Force}{m}\Delta t$

力積Impulseは速度への変換にΔtを使用しません。 $\Delta v = \frac{Impulse}{m}$

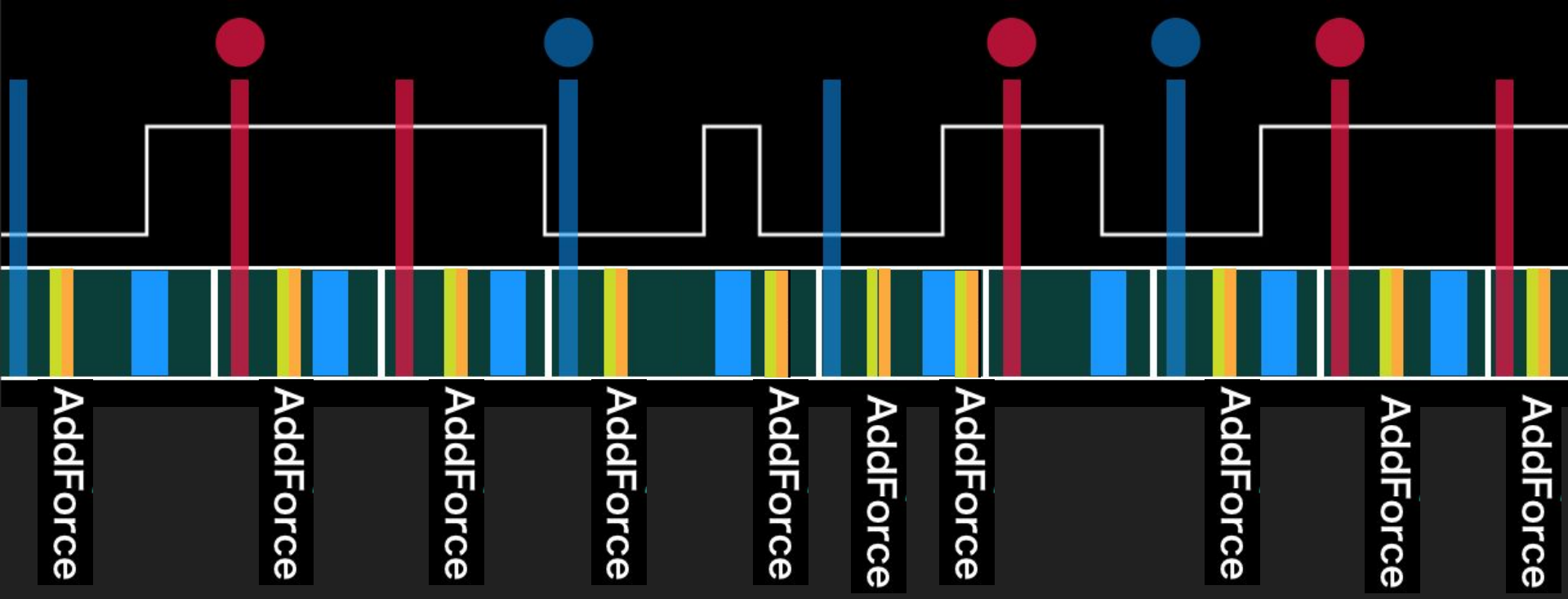
力積はΔtを織り込み済みなので、１フレームだけ発生させること（という物理世界では意味不明な出来事）に矛盾が生じないのです。

FixedUpdateが物理シミュレーションのループと同期している

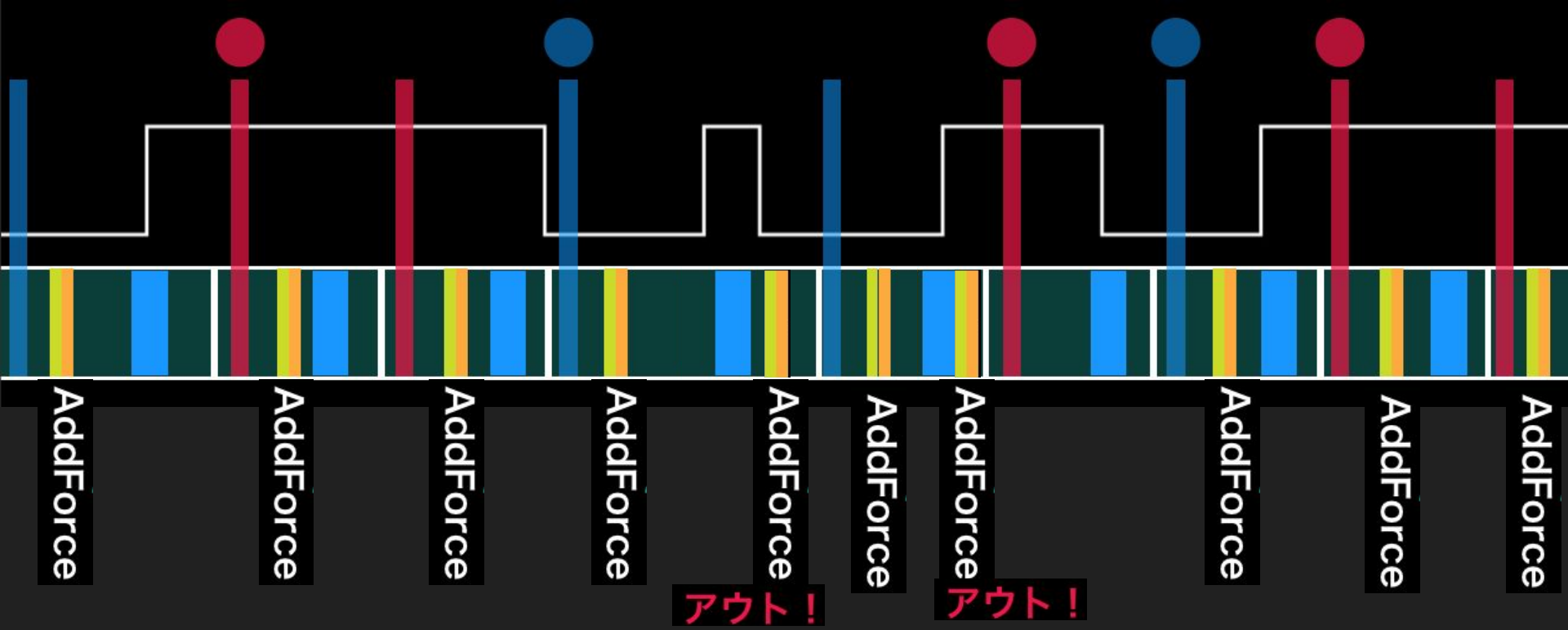


このように、物理演算の1ステップにつき必ず一度、FixedUpdateが呼ばれる、と言い換えても良いでしょう。

つまり、そもそも数フレームにまたがることが期待されている **外力 AddForce(force, ForceMode.Force)は、FixedUpdateで呼ばれなければならない** ということです。Updateで呼んでしまったら物理シミュレーションにとっては断続的になってしまいますから！



等間隔に見えないかもしれないが、簡易的に等間隔として見て



このように Updateで外力のAddForceを呼んでしまうと、物理シミュレーションから見れば、
AddForceが断続的に処理されてしまいますね。(2倍の力で引っ張ってしまう)
図には書いていませんが0回のところもアウトです。

よーわからん、という人は、呼んだAddForceは次回の物理演算で処理されるということをイメージすると(上図の緑の線)、
問題が明らかになると思います。そしてここまで話してきた理屈により、**力積のAddForceをUpdateで呼ぶことにはなんら問題はありません。**

Input.GetButtonDown などの押し下げ瞬間は Update で呼ぶ必要がある。

瞬間なのでAddForceとしては力積が良い

外力の AddForce は FixedUpdate で呼ぶ必要がある。

連続なので入力としては Input.GetButton などの押し下げ状態が良い

の２点が実装の指針となるでしょう。あえて具体的な例を挙げれば、移動はFixedUpdate、ジャンプはUpdate で実装するのがよいです（全体を理解せずここだけ切り取って覚えると失敗するので注意）。

先ほどの例のように弾が2発出てしまったり、**物理シミュレーションのΔtを変更したら動きが変わってしまったり**します。
例えばバレットタイムの実装で泣きをみるかもしれません。

終了