DCCツールの補助骨システムをUnity®で動かす

- Burst と Job System の活用事例 -

株式会社スクウェア・エニックス
テクノロジー推進部

佐々木　隆典

SQUARE ENIX ADVANCED TECHNOLOGY DIVISION

自己紹介

- テクニカルアーティスト（エンジニア）
- 全社向けに、DCCツールのプラグインなどを作る仕事
- リグやアニメーション関連が多い
- Unity® と C# 歴　１年半弱

今回、ゼロから Unity® を学びながら開発しました。

その経験で得た様々な知見を共有したいと思います。

アジェンダ

本講演資料は、後日、弊社Webページ、及び CEDiL にて公開致します。

- 概要
  - どのような実装が必要か
  - Unity® での開発
- Unity® unsafe C# 基礎
- 実装
  - KDIアセットデータ
  - MT コンポーネント
  - Job コンポーネント
  - Playable コンポーネント
- 負荷計測とまとめ

概要

どのような実装が必要か

補助骨システム KineDriver

「次世代を見据えた新しい補助骨システムの開発」

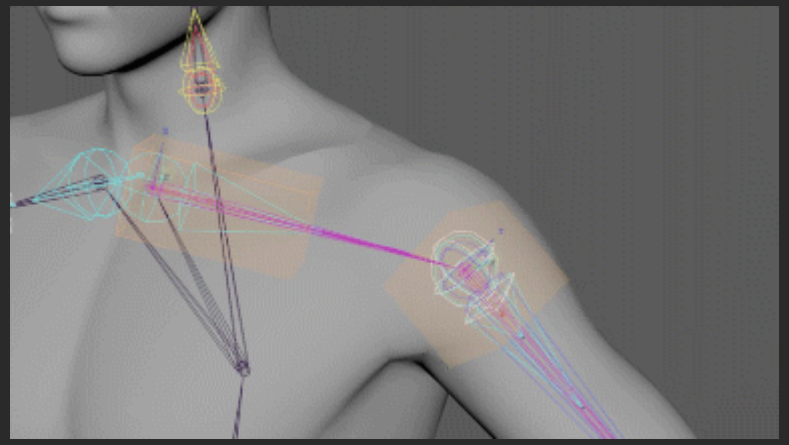
動作環境：

- Maya®
- Motion Builder
- 内製エンジンA, B, C, …
- Unreal® Engine
- Unity® NEW!!

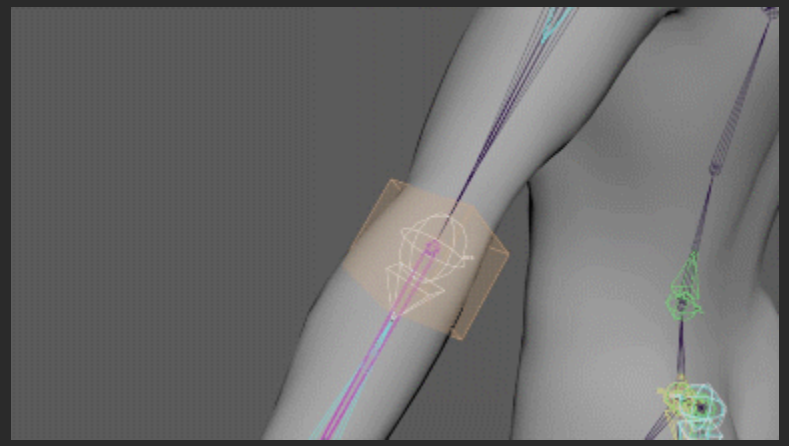
Unity® だけ C# で実装！

というお話です。

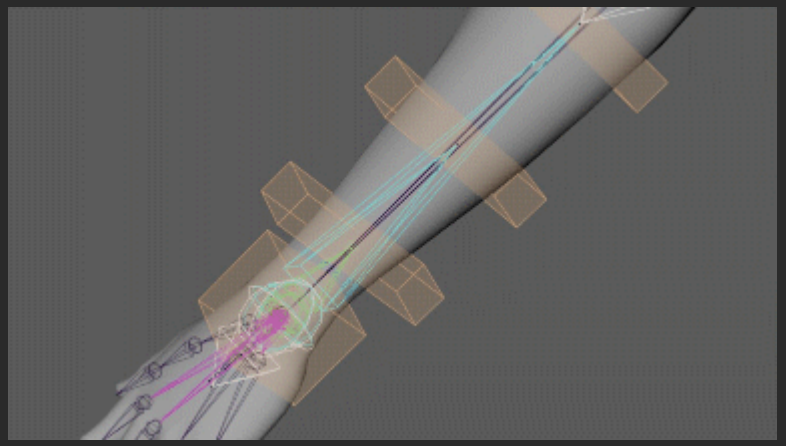
補助骨システムとは



肩周辺の例



肘の例

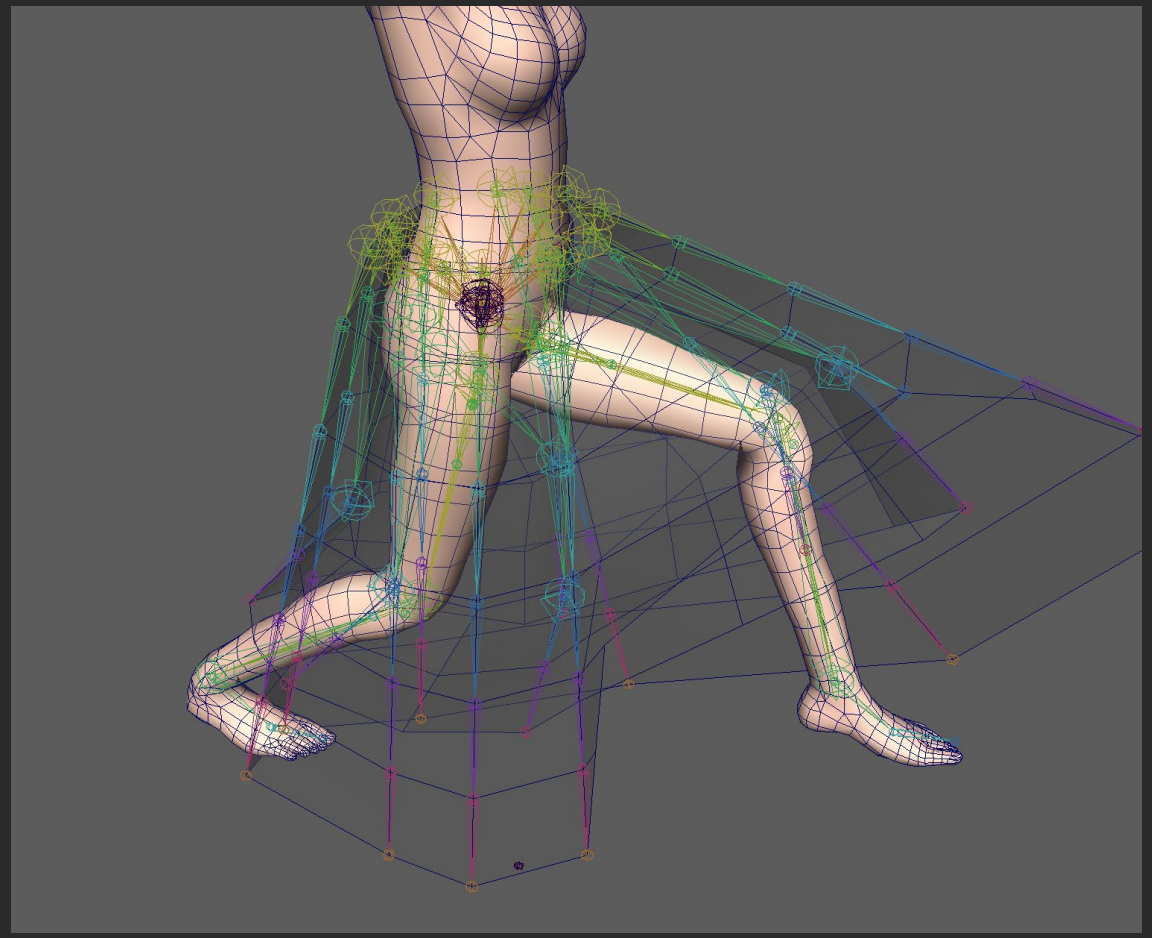


手首の例

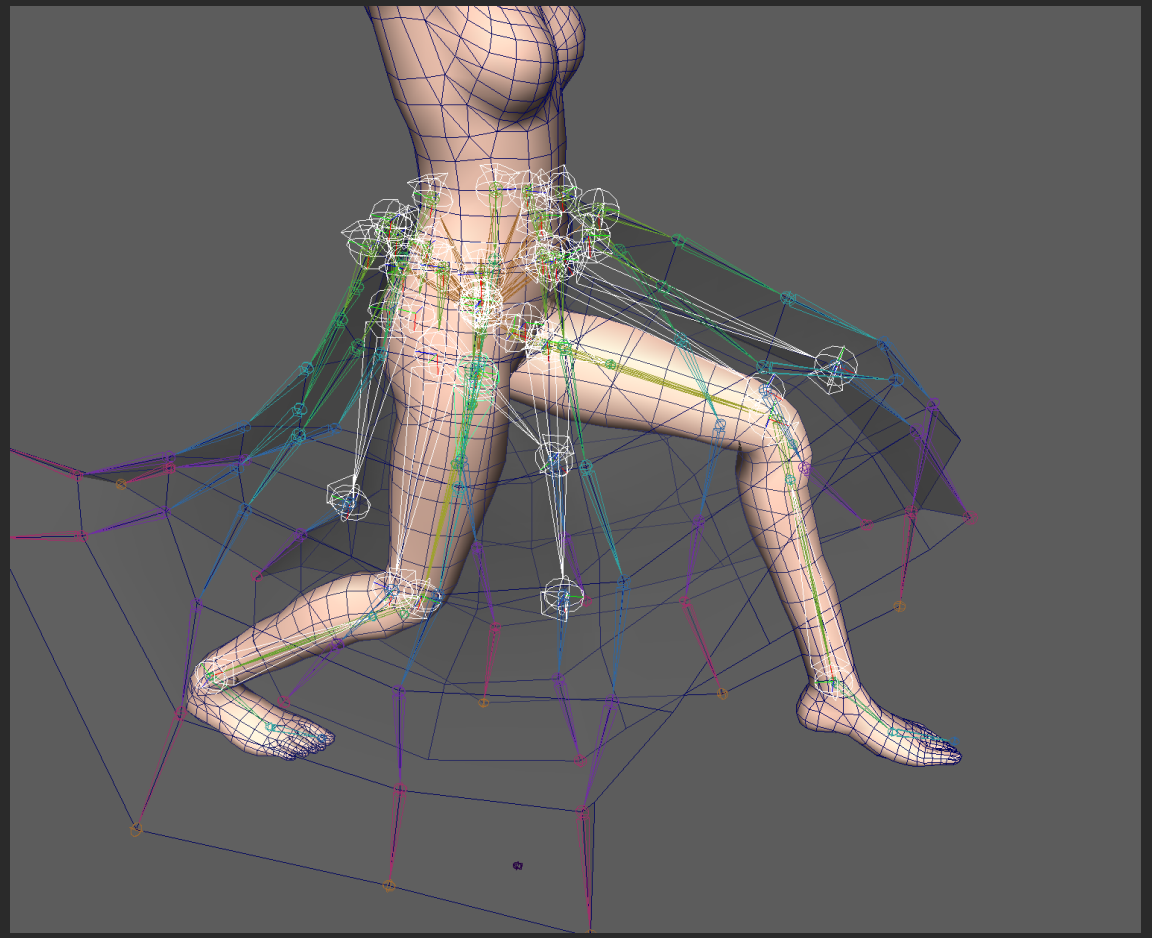
スキンのデフォーメーションを補うなどの目的で挿入される補助的なジョイントを自動的に動かす仕組み。

メインスケルトンの動きに応じて自動的に動かす。

揺れ骨のガイドとしても

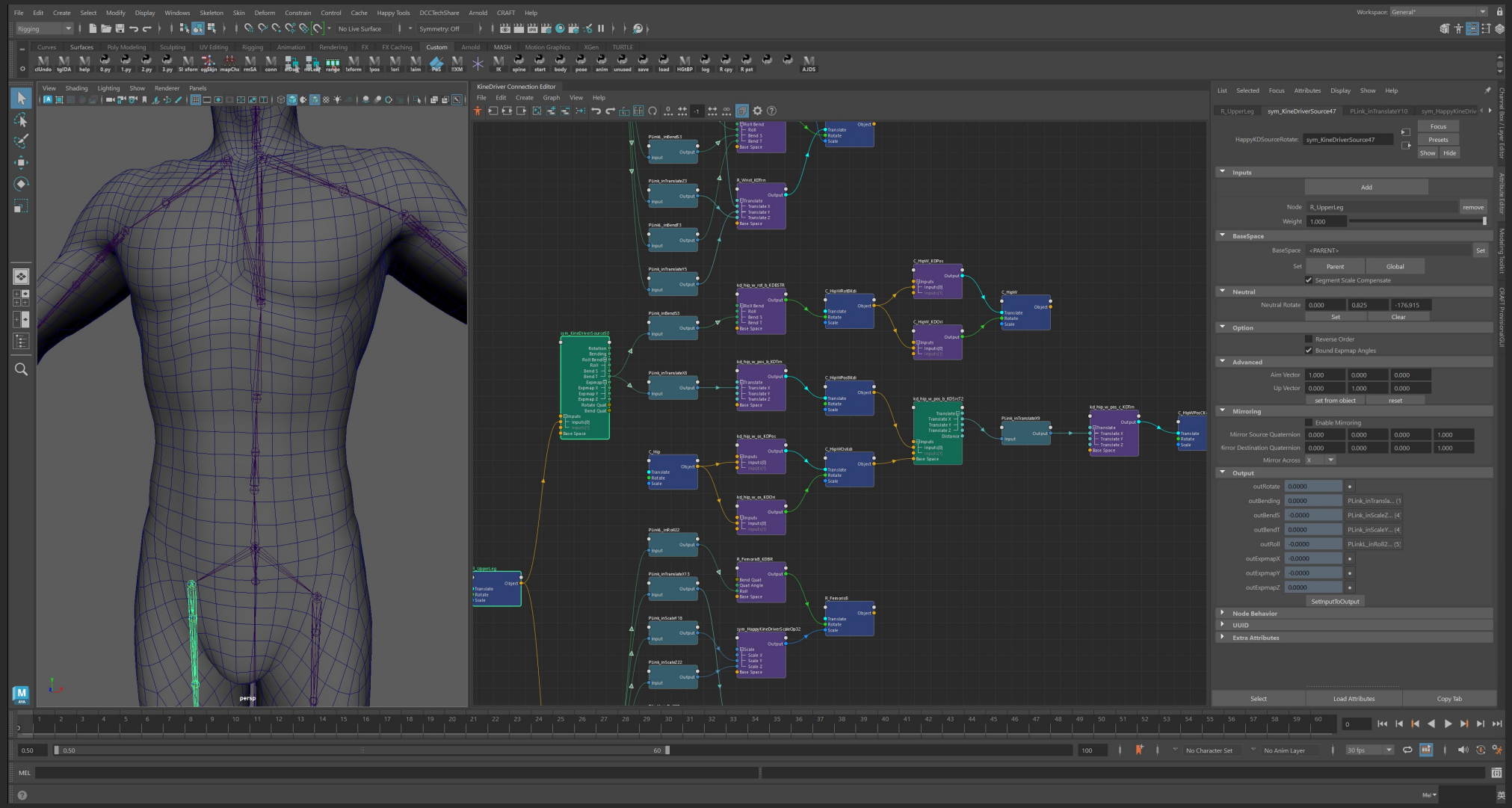


KineDriver のみ

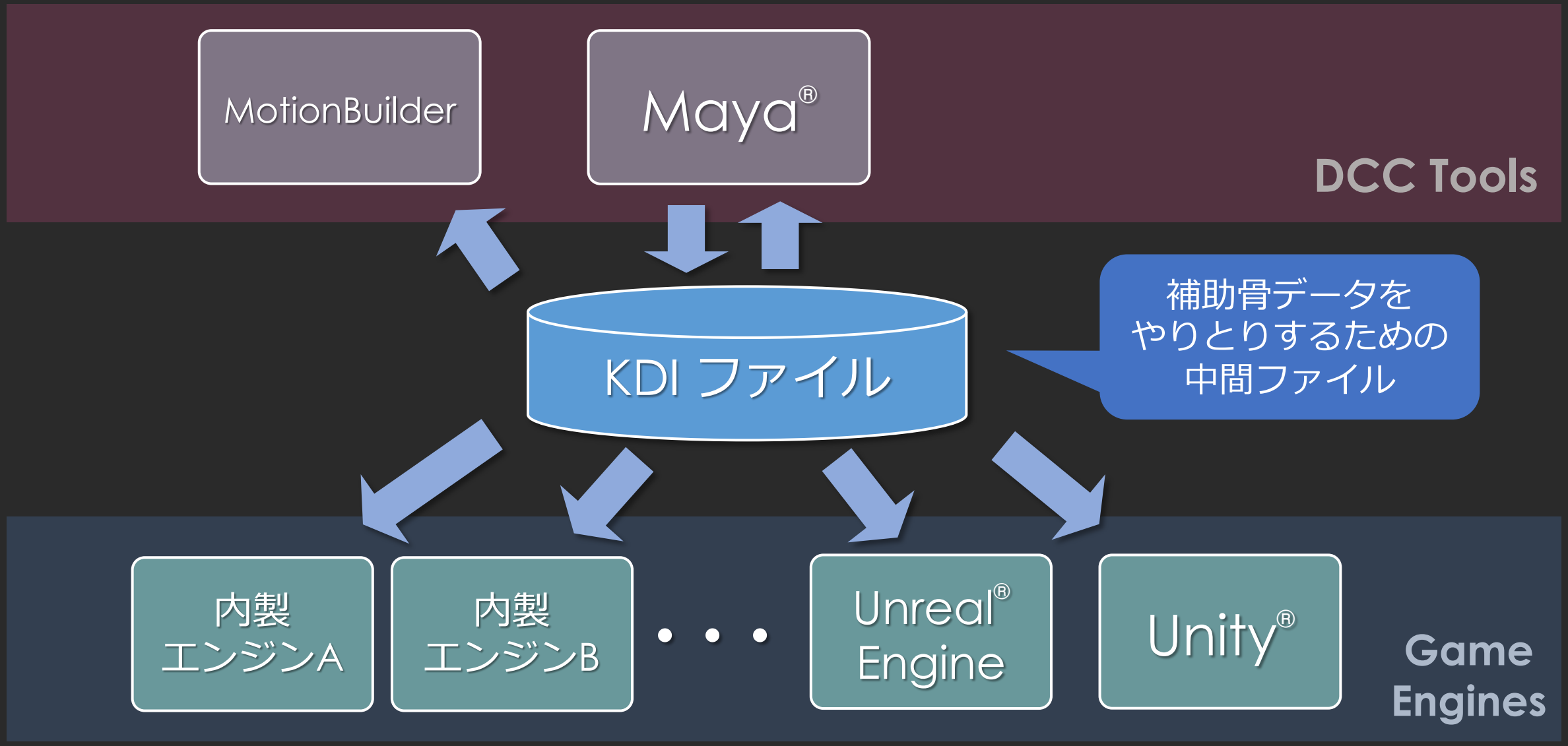


KineDriver + Bonamik
（弊社の揺れ骨システム）

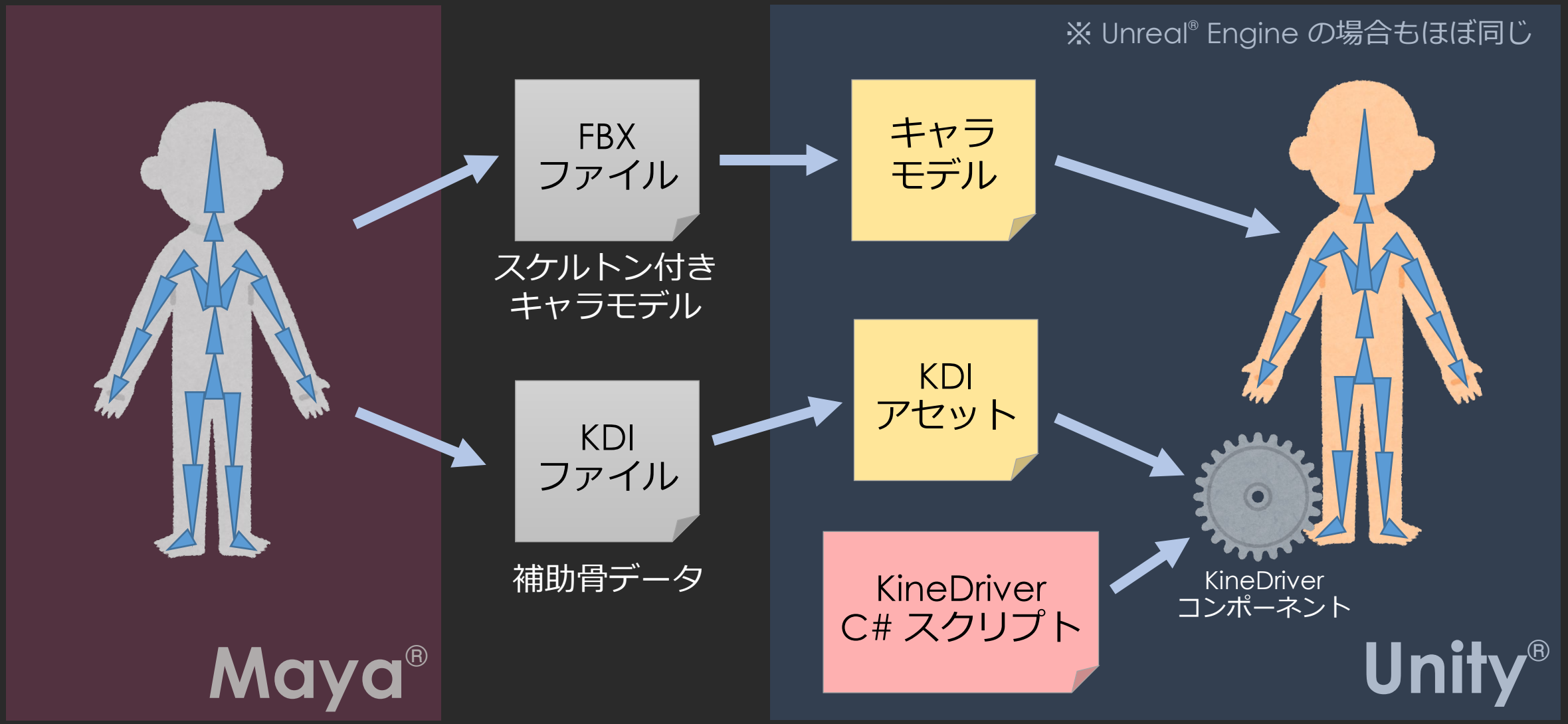
Maya®上でオーサリング



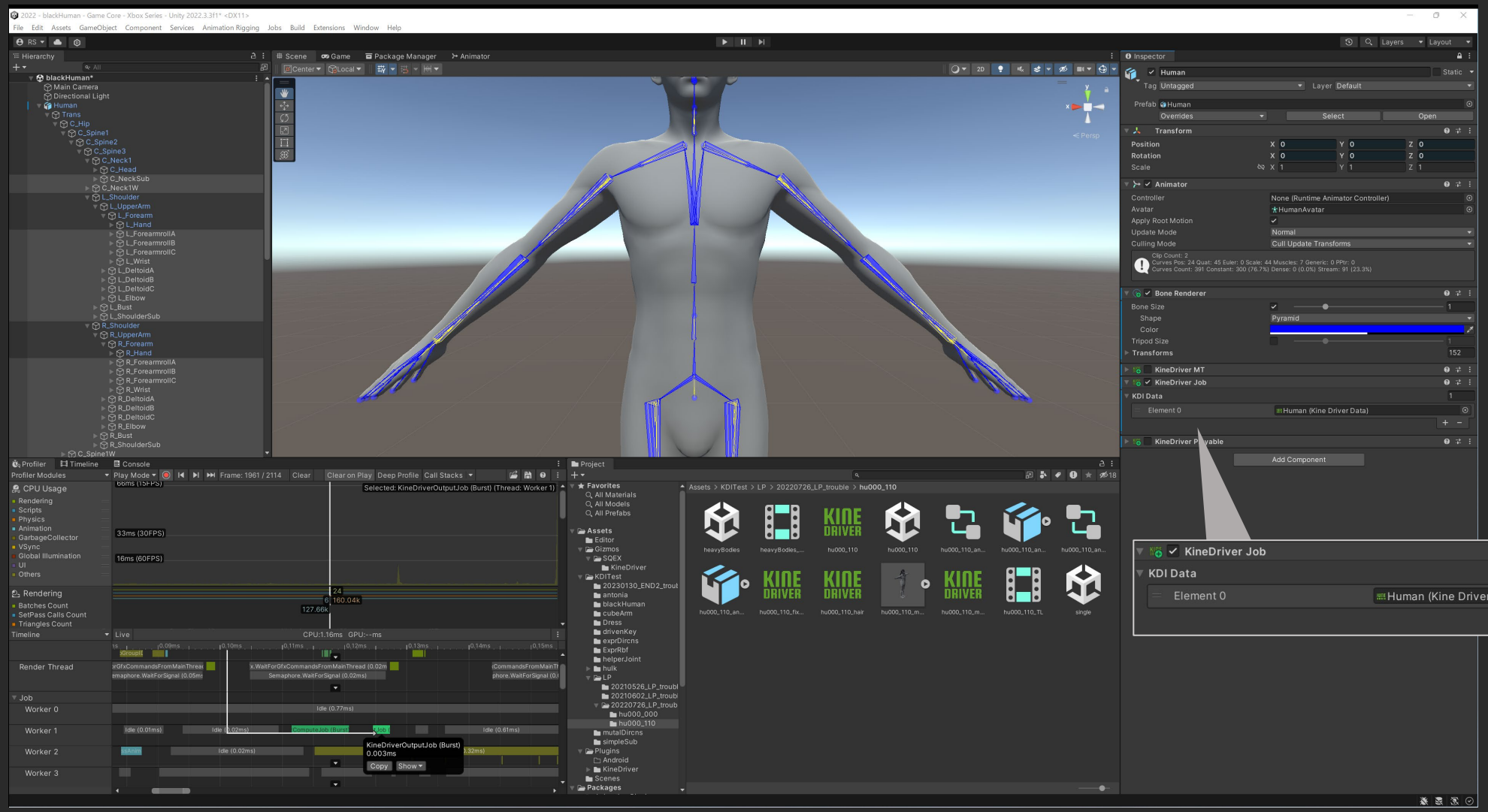
KDIファイル（補助骨データ）



Unity® にインポート

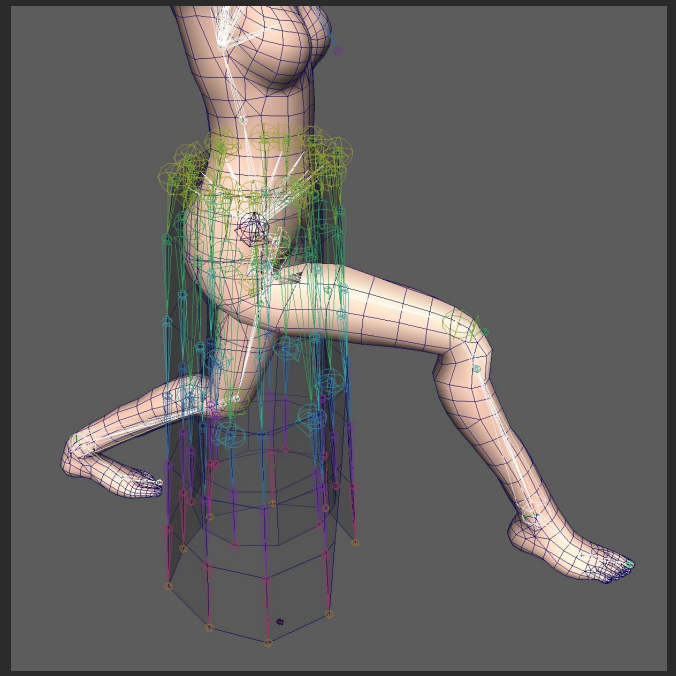


Unity®で再生

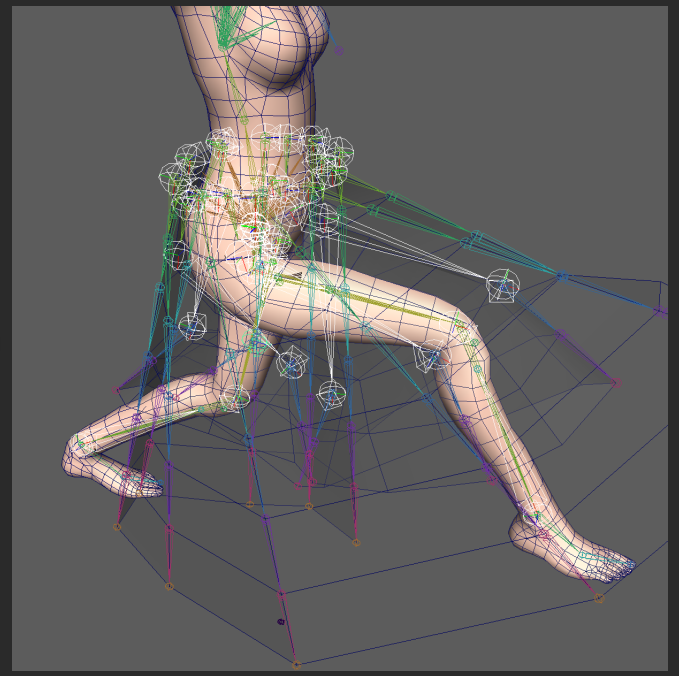


ゲームランタイムの実装

1フレーム中のアニメーション処理の順序

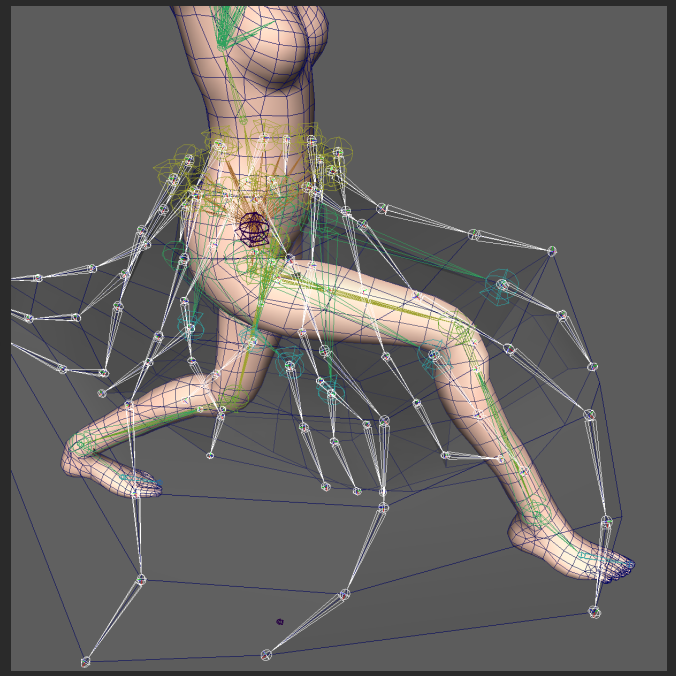


メイン骨
アニメーション



補助骨

メイン骨のポーズから一意に計算

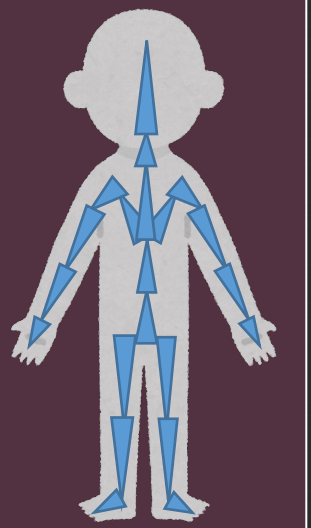


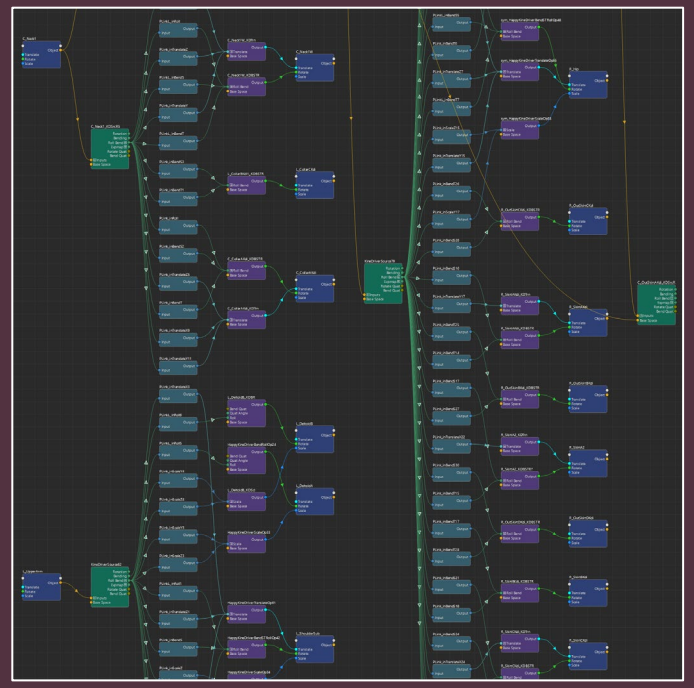
揺れ骨
（物理シミュレーション）

ゲームランタイムの実装

並列処理単位について

並列実行



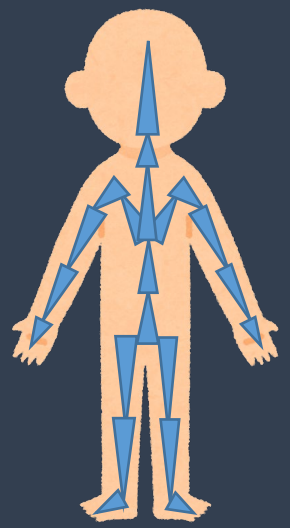


各ノードが並列実行される

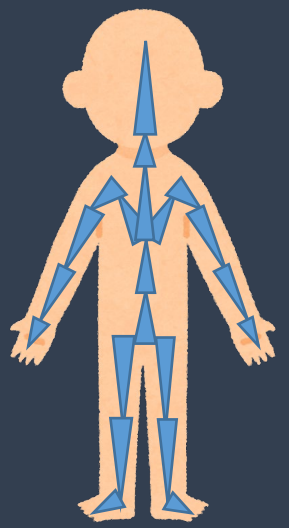
Maya®

VS

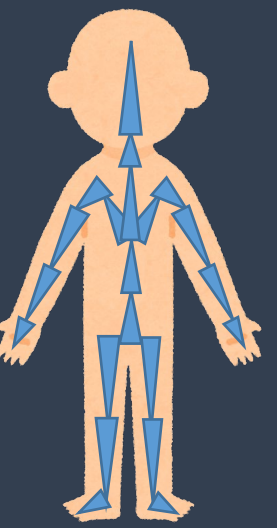
逐次実行



逐次実行



逐次実行

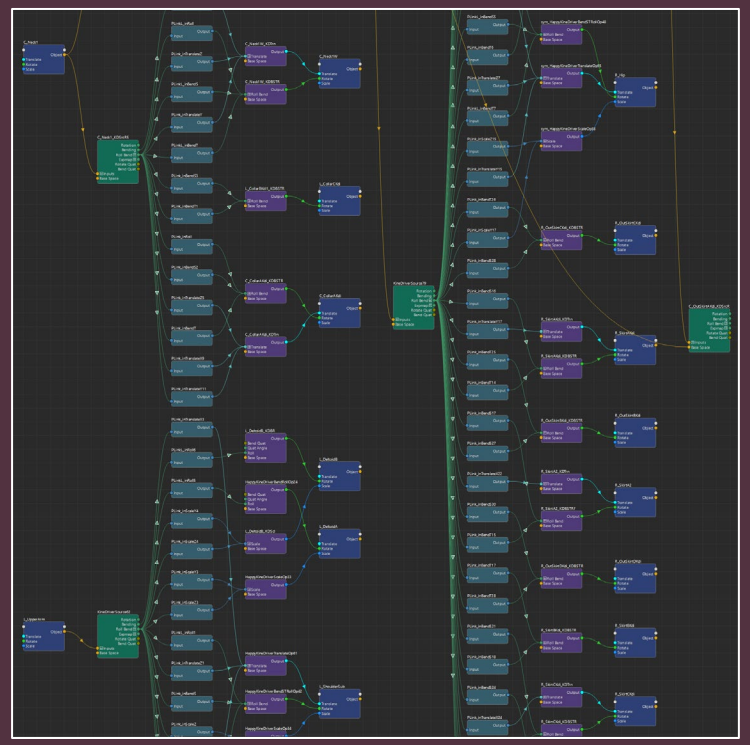


キャラクター単位で並列実行
（そのように実装）

Unity®

ゲームランタイムの実装

ノードの計算順序

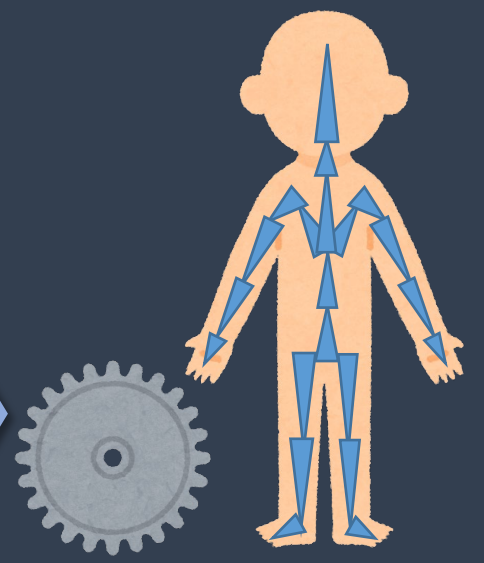


Maya®

トポロジカル
ソート

ノード 1
ノード 2
ノード 3
ノード 4
・
・
・
・
・
・
・
・
ノード ？？

KDIファイル
(線形リスト)



ファイルに書かれた順に
ノードの計算を実行する

Unity®

概要

Unity® での開発

最初の印象

Unity® って、やっぱ C# で開発するもの？

めんどくさいなー

Unity® 初心者な私
（食わず嫌い）

実は C++ でも書ける

C++ で DLL を書いて、 Unity® に読み込ませることができる。
（ネイティブプラグイン）

機能拡張を、 C++ で書くべきか？　C# で書くべきか？

C# vs C++

	メリット	デメリット
C#	• ビルド不要なスクリプト感覚 • プラットフォーム対応が楽 • Unity® の機能にアクセスしやすい • 世の中に情報が多い	• 既存の C++ コードは活用できない • 処理速度が遅そう？
C++	• 既存の C++ コードを活用できる • 処理速度が速そう？	• スクリプトのように手軽じゃない • プラットフォームごとのビルド対応が必要 • Unity® の機能にアクセスしにくい • 世の中に情報が少ない

全てを C# で書いてみることに

- 元々、各DCCツールやゲームエンジンごとに実装していた
  - ライブラリなどの共通コードは少ない
  - C#でフロムスクラッチでもOK
- C# の方がメリットが多そう
  - 開発やメンテナンス、プラットフォーム対応が楽なのが良い
  - しかし、処理速度がちょっと不安？

DOTS (Data-Oriented Technology Stack)

- C# Job System
  - Native Container / UnsafeUtility
- Burst
  - HPC# (High Performance C#)
  - Mathematics
- Animation C# Jobs
  - Playable API
  - PlayableGraph
- ECS (Entity Component System)

DOTS (Data-Oriented Technology Stack)

- C# Job System
  - ワーカースレッドに処理を分散させる仕組み
  - アンマネージドメモリを使う必要がある
- Burst
  - 機能制限された C#（HPC#）で高速なコードを生成するコンパイラー
  - 積極的に最適化、ベクタライズ（Mathematics モジュール便利）
- Animation C# Jobs
  - Playable Graph で動作する Unity® のアニメーションストリームに処理を載せるための Job System 機能
- ECS
  - データ指向のアーキテクチャパターン ECS に基づく仕組み
  - 長いこと Experimental だったが、最新の 2022.3 LTS で正式リリース

なるほど、わからん

C# Job System ？
Animation C# Jobs ？？
ていうか、何が違うのよ

Playable API

PlayableGraph

Native Container

Burst

UnsafeUtility

HPC#

ECS

段階的に開発することにした

普通にメインスレッドで動くもの

↓

Job System で動くもの

↓

Playable Graph で動くもの

KineDriver コンポーネント

KineDriver MT

メインスレッドのみで動作（並列処理されない）

KineDriver Job

Job System で動作

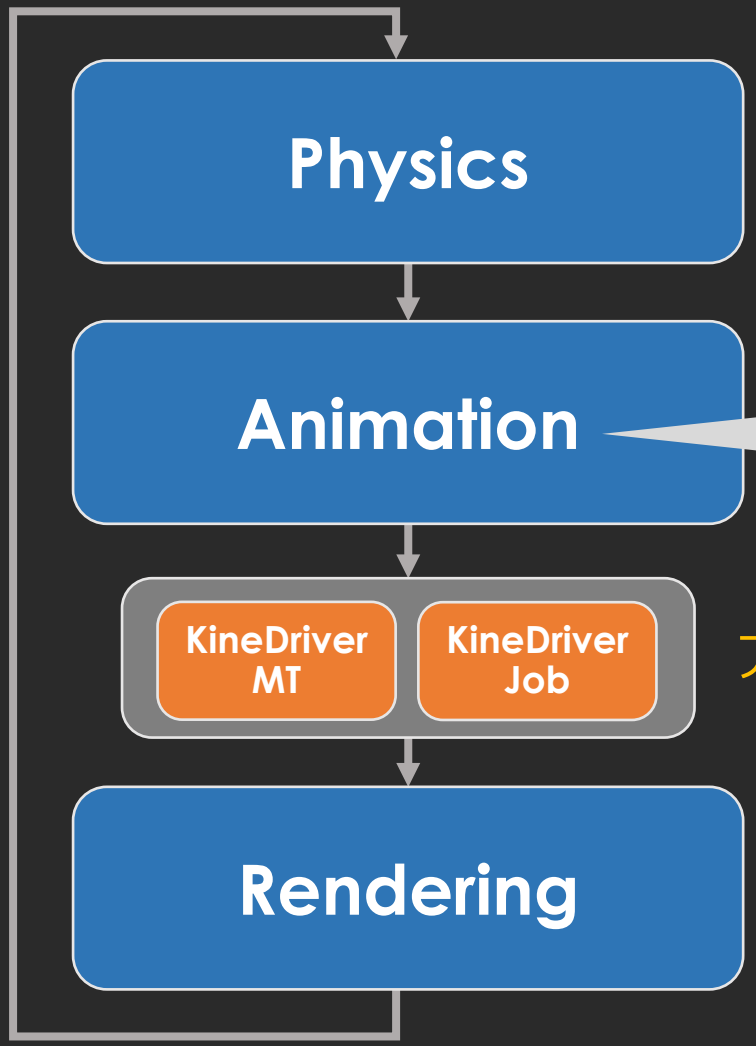
KineDriver Playable

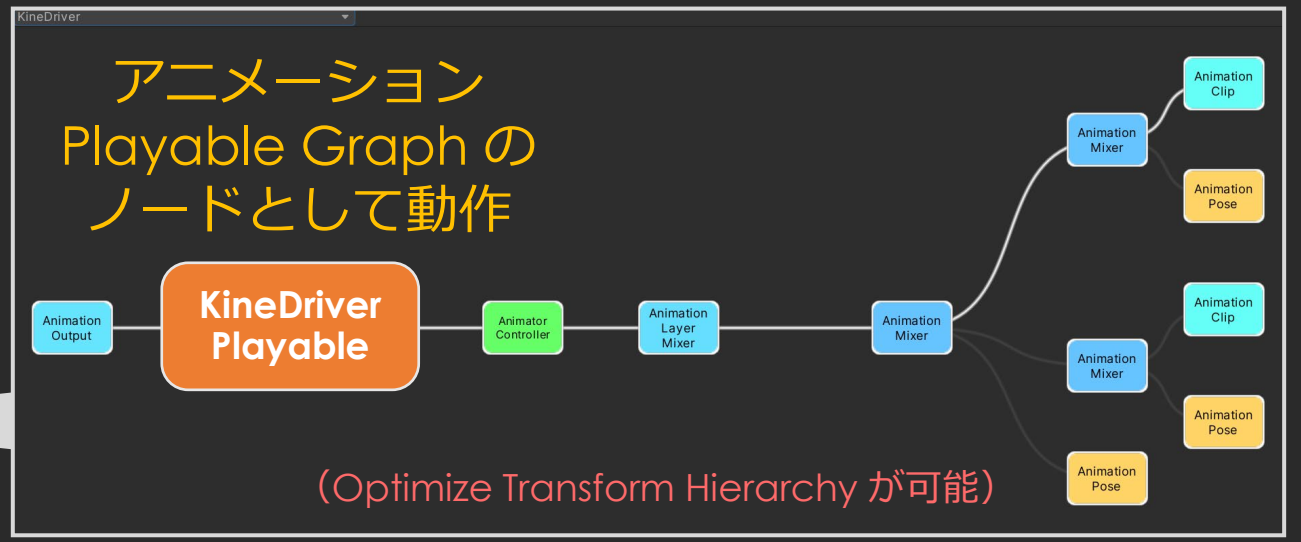
アニメーション Playable Graph のノードとして動作

使い分けられると役立つ局面もあるかも？

- 揺れ骨など、組み合わせる他のモジュールの仕様
- その他、運用上の何らかの制限など

実行タイミングと性質の違い





アニメーションの後処理として動作

ECS ?

開発段階では Experimental だったため、考えないことに

2022.3 LTS で正式リリースされたが、アニメーションが未対応

一応、2022.3 LTS で検証してみた：

- ECS のプログラムを書いてオブジェクト（Entity）を動かせるので、補助骨を動かすことはできる。
- FBX からインポートしたキャラクターは GameObject なので、GameObject から Entity へ変換することになる（それ自体は可能）。
- しかし、Animator コンポーネントや、スキニングが動かない。

Unity® unsafe C# 基礎

struct について

- struct と class は、
  C／C++ ではほぼ同じものだが、
  C# では明確に異なるもの。
- 存在意義は速度向上のためだが、使用には注意が必要。
- 一般的には、非常に小さなサイズの型を除き struct の使用は避けた方が良いとされるが、 DOTS では使用が必須 であり、大きいサイズでも使うことになる。

struct と class の違い

値型

struct

int

float

・
・
・

など

Primitive型

参照型
（管理されたポインタ）

class

string
object
Array
… なども

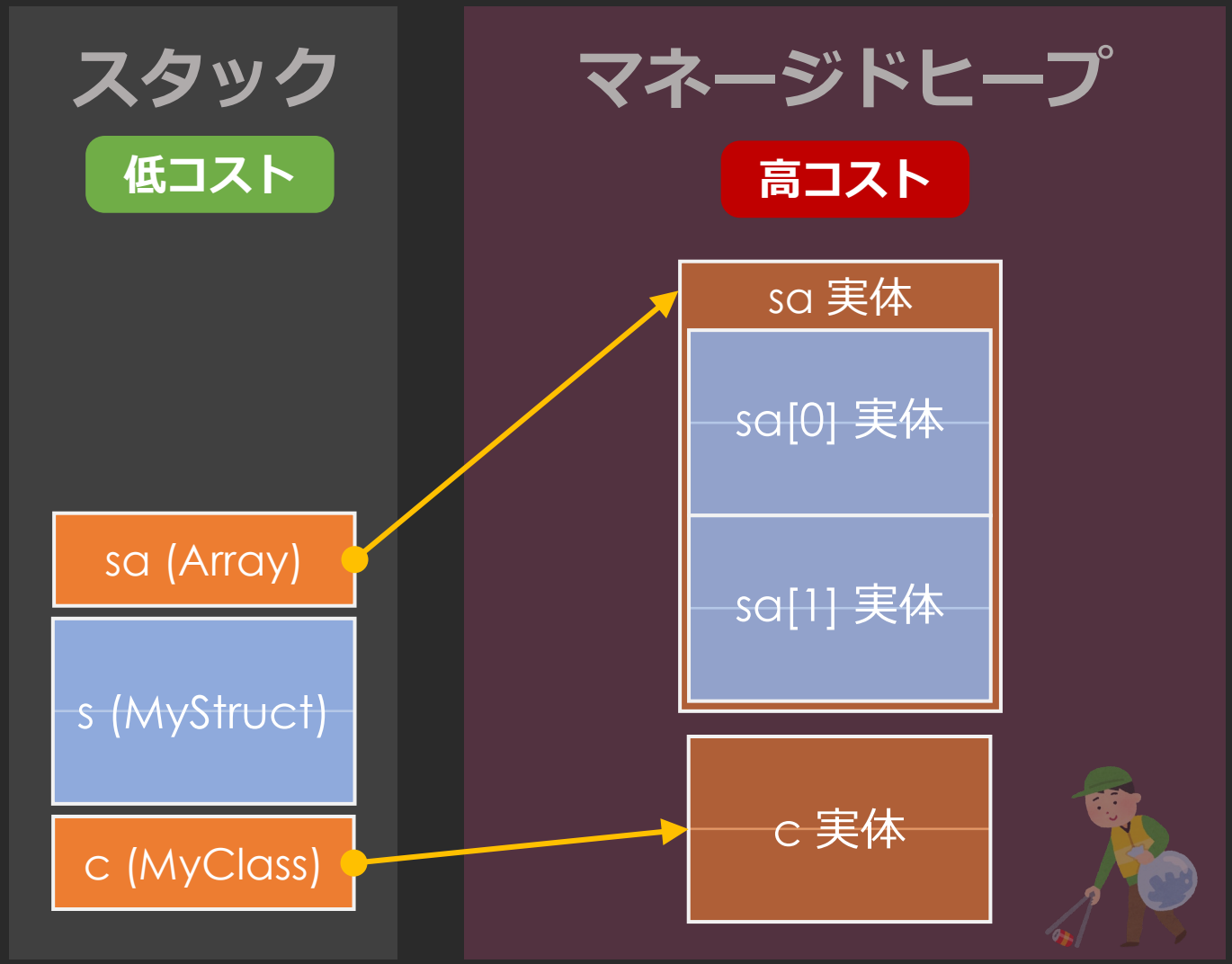
マネージドヒープ
（後始末を自動でやってくれる管理されたメモリ領域）

class
実体

GC（ガベージコレクタ）

スタックかヒープか

```
class MyClass
{
    public int x, y;
}

struct MyStruct
{
    public int x, y;
}

class ExampleClass
{
    static void Main()
    {
        var c = new MyClass();
        var s = new MyStruct();
        var sa = new MyStruct[2];

        ...
    }
}
```



コピー：参照か実体か

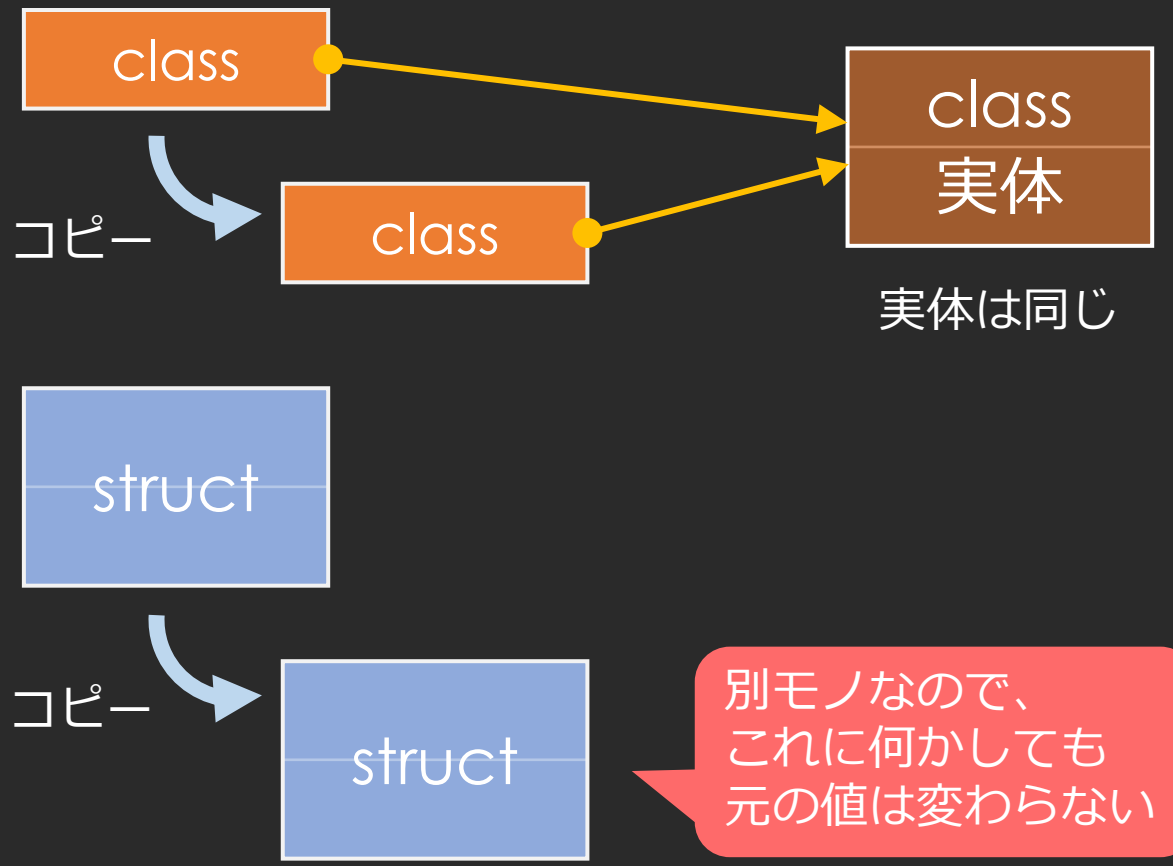
```
class ExampleClass
{
    static void Main()
    {
        var c = new MyClass();
        var s = new MyStruct();

        var cp_c = c;
        var cp_s = s;

        Proc(cp_c, cp_s);
    }

    static void Proc(MyClass c, MyStruct s)
    {
        c.x = 1;
        s.x = 1;
    }
}
```

コピー

値渡し（コピー）

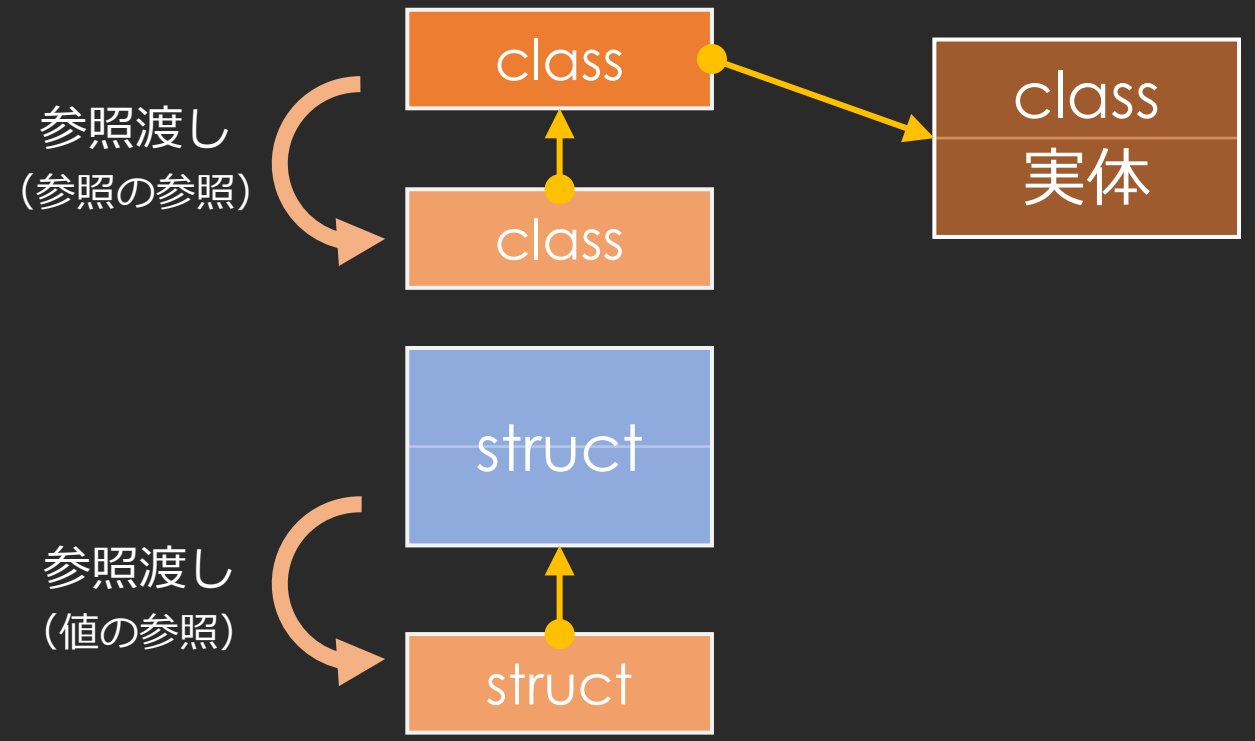
参照先を変更

無意味



参照渡し引数

```
class ExampleClass
{
    static void Main()
    {
        var c = new MyClass();
        var s = new MyStruct();
        ProcC(ref c);
        ProcS(ref s);
    }

    static void ProcC(ref MyClass c)
    {
        c = new MyClass() { x=1, y=2 };
    }

    static void ProcS(ref MyStruct s)
    {
        s.x = 1;
    }
}
```

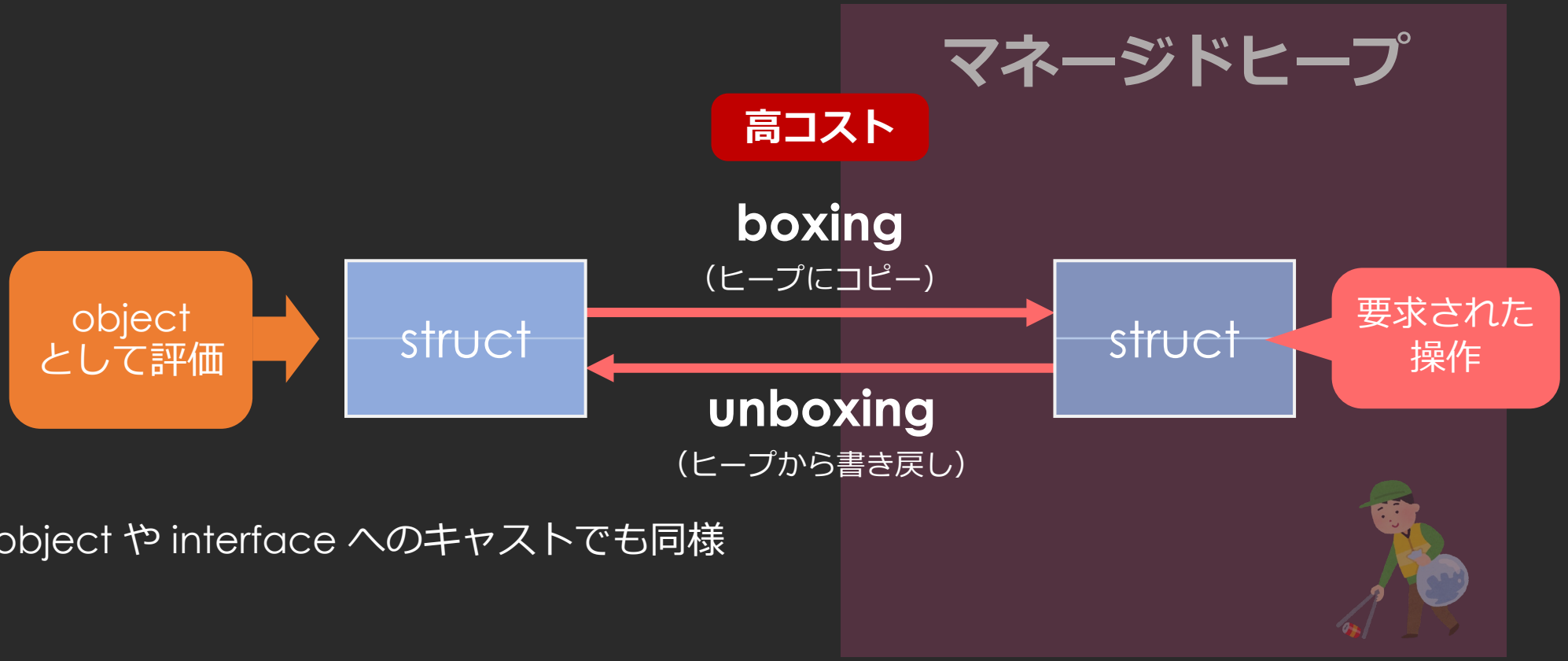


ref	通常の参照引数
in	入力専用の参照引数
out	出力用の参照引数

in は「防衛的コピー」がされる場合があるので注意
（怖い場合は ref にする）

恐怖！ボックス化 (boxing / unboxing)

struct は継承できない（interface の継承は可能）のに、
object（全 class の暗黙のルート class ）として扱えてしまう。



object や interface へのキャストでも同様

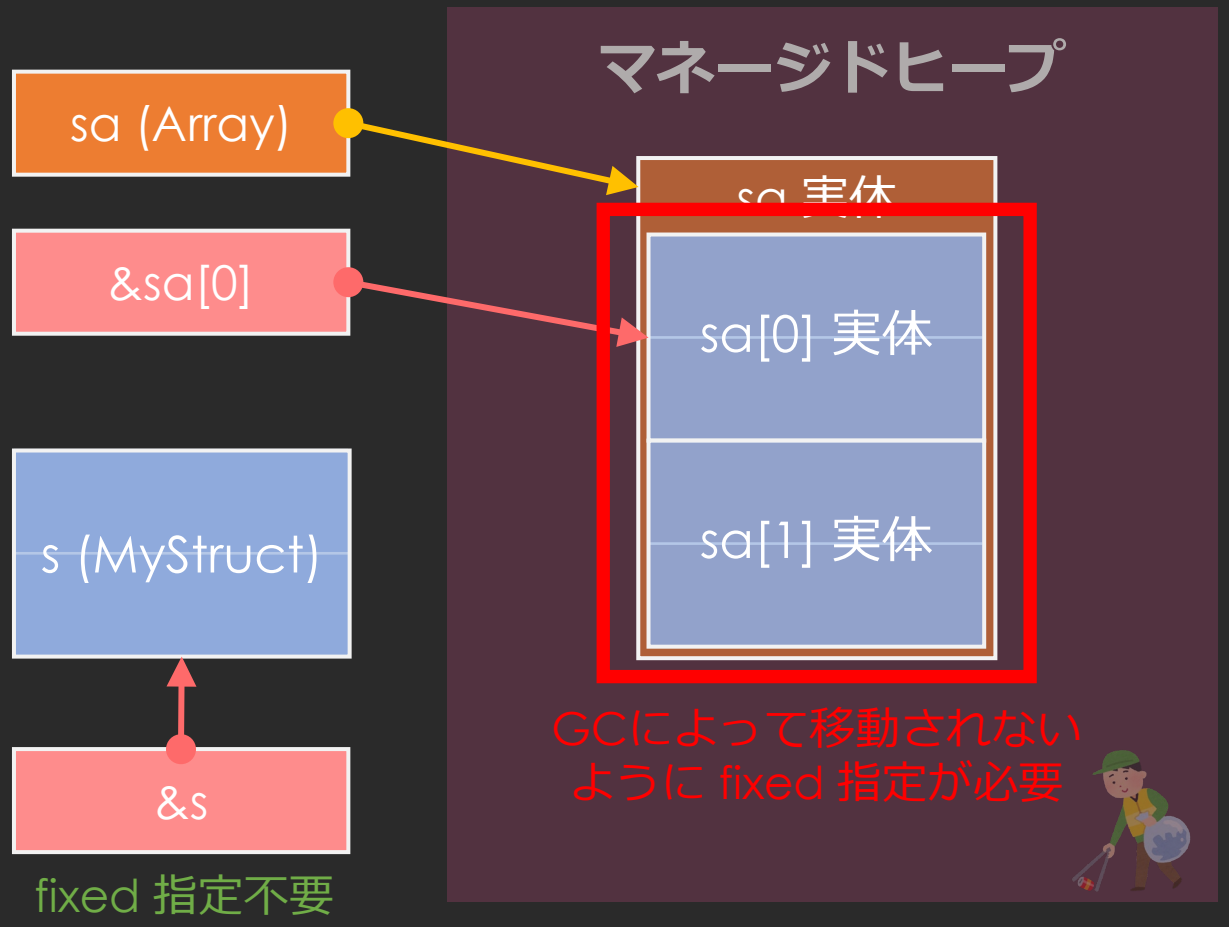
ポインタ

C# でも unsafe コンテキストにすれば普通のポインタが使える。

```
class ExampleClass
{
    static unsafe void Main()
    {
        var sa = new MyStruct[2];
        var s = new MyStruct();

        fixed (MyStruct* p_sa = &sa[0]) {
            p_sa[0] = s;
            p_sa[1].x = 3;
            p_sa[1].y = 4;
        }

        MyStruct* p_s = &s;
        p_s->x = 1;
        p_s->y = 2;
    }
}
```

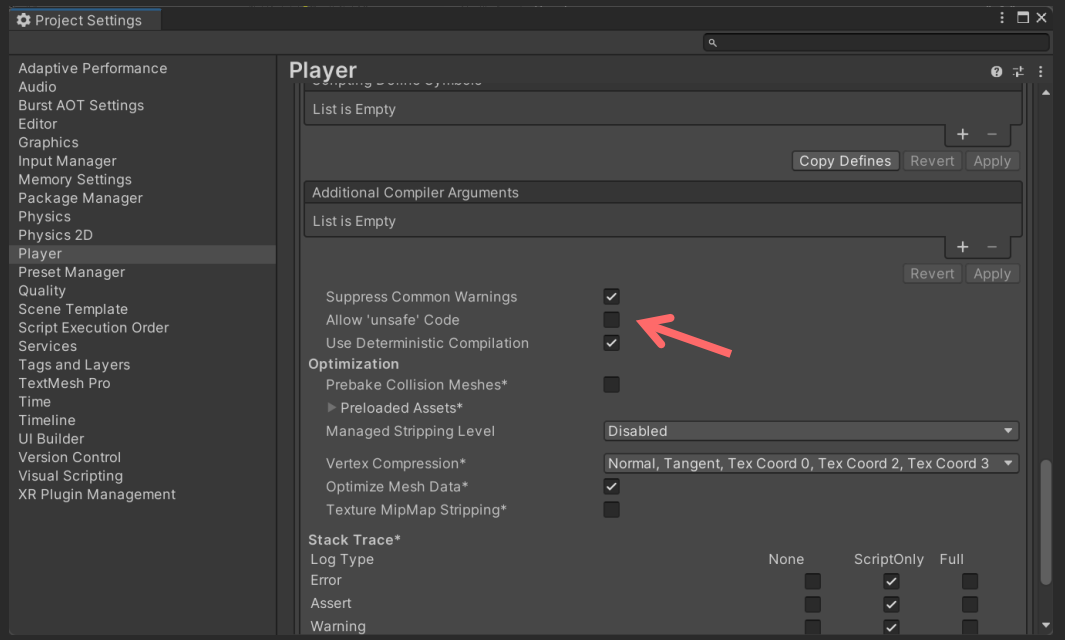
値型

C/C++ と同じ感覚で使える



Allow ‘unsafe’ Code 設定

- unsafe を許可するには、C# コンパイラに指定が必要
- Unity® だと Project Settings に設定がある
  - プロジェクト全体の設定になってしまう
  - プラットフォームごとの個別設定
- アセンブリ定義するのが良い



C# アセンブリ定義

```
{
    "name": "SQEX.KineDriver.Data",
    "rootNamespace": "",
    "references": [
        "Unity.Mathematics",
        "Unity.Burst"
    ],
    "includePlatforms": [],
    "excludePlatforms": [],
    "allowUnsafeCode": true,
    "overrideReferences": false,
    "precompiledReferences": [],
    "autoReferenced": true,
    "defineConstraints": [],
    "versionDefines": [],
    "noEngineReferences": false
}
```

asmdef ファイル

- アセンブリとは .NET のライブラリのこと
- Unity® では asmdef ファイルで定義
  - 置いたフォルダ階層下の C# コードが 1 つのアセンブリ（DLL）になる
  - 未定義の場合、未定義の C# コード全部で 1 つのアセンブリ（DLL）になる
- こういった機能拡張は、アセンブリを分ける
  - 依存関係を書くので再コンパイル頻度が減る
  - unsafe 許可を指定できる

Unity® メモリの種類

- C# スタック
- C# マネージドヒープ
- ネイティブメモリ
  - エンジン内部で使用される C++ メモリ
  - 通常、ユーザーはアクセスできない
- C# アンマネージドメモリ
  - C# からアクセスできるネイティブメモリ
  - 管理されていないので、使用後の解放は自分で

アンマネージドメモリ

- 使えるのは Blittable 型
  - C++ とメモリレイアウトが同じになる型
    - byte, sbyte, short, ushort, int, uint, long, ulong, IntPtr, UIntPtr, float, double
    - struct
    - fixed（固定長バッファ）
  - 値型とは微妙に異なる（bool と char は含まれない）

    ただし Unity® DOTS では bool は Blittable 型として扱える
- NativeArray で利用する
  - 使い終わったら Dispose() で解放が必要
  - 中身は UnsafeUtility の Malloc / Free で実装されている

その他のこと

- Enum は class
  - unsafe の国では、使わない
- [StructLayout(LayoutKind)]
  - struct のフィールドレイアウトを指定できる
    - Sequential – 書いた順番通り（デフォルト）
    - Auto – 自動（class の場合はこれ）
    - Explicit – 細かく指定
  - Explicit では C の union 相当も可能

実装

KDIアセットデータ

独自アセット実装のAPI

- アセット
  ScriptableObject を継承した class を実装。要シリアライズ対応。
- インポーター
  ScriptedImporter を継承した class を実装。要シリアライズ対応。
- アセットの Inspector UI
  Editor を継承した class を実装。
- インポーターの Inspector UI
  ScriptedImporterEditor を継承した class を実装。

Unity®におけるシリアライズ

- 「データ構造やゲームオブジェクトの状態を Unity® が保存して後で再構築できる形式に変換する自動処理」
  （マニュアルより引用 https://docs.unity3d.com/ja/2022.3/Manual/script-Serialization.html ）
- 平たくいうと、ファイルの読み書き
  - シリアライズ = ファイルに書き込み
  - デシリアライズ = ファイルの読み込み
- 基本的に、class や struct に「これはシリアライズして」と指定するだけ。

シリアライズ対応のAPI

- クラス／構造体
  - 対応させる class や struct に [Serializable] 属性を付ける。
- フィールド
  - 条件を満たす public フィールドがシリアライズ対象。
  - [SerializeField] を明示すると、private でもシリアライズ対象になる。
  - [SerializeReference] を付けると、抽象 class や interface の参照でもシリアライズ可能。
- 処理のカスタマイズ
  - Interface ISerializationCallbackReceiver を継承・実装。
  - 本来の用途と異なるが、ホットリロードの対応に用いた。

KDIファイルの内容

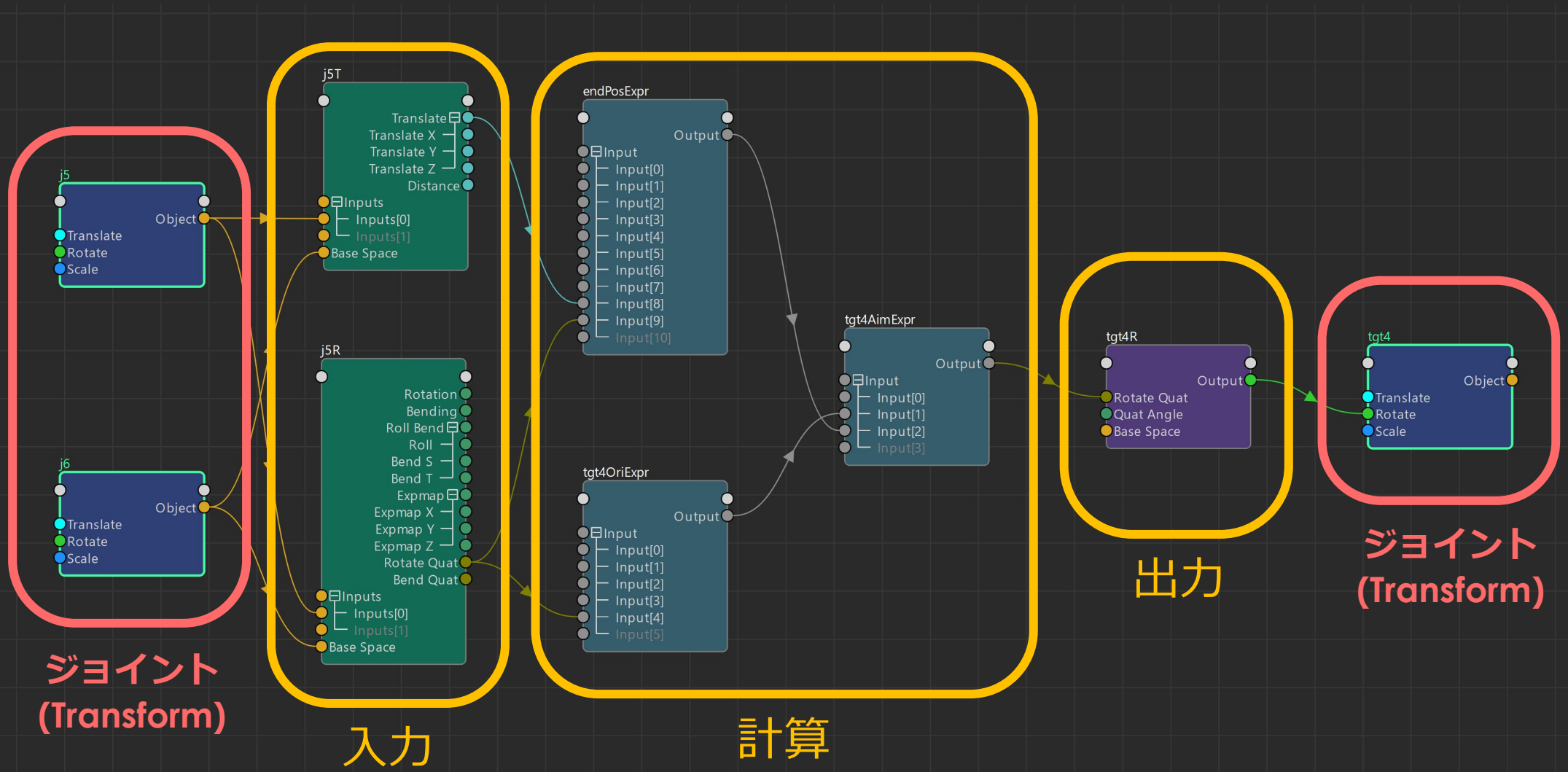
KDIファイル
ノードデータ・リスト ・ ・ ・
コネクション情報・リスト ・ ・
ジョイント名・リスト ・ ・

- ノードタイプ
- パラメータ値リスト（タイプにより異なる）
- 参照ジョイント名（入出力を担うノードの場合）

- 入力ポート（どのノードのどのポート？）
- 出力ポート（どのノードのどのポート？）

- 入力ジョイント（メイン骨、または補助骨）
- 出力ジョイント（補助骨）

ノードの役割



いろいろなノードタイプ

Transform 入出力

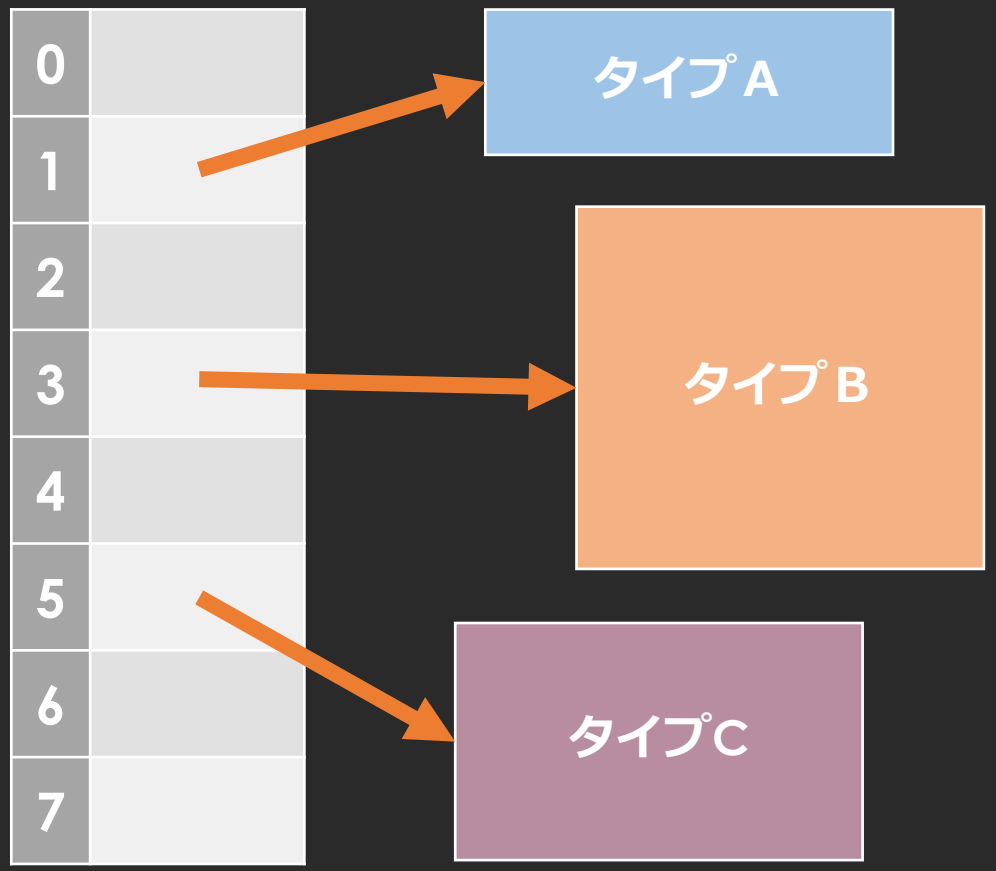
- Translate 入力／出力
- Rotate 入力／出力
- Scale 出力
- 位置コンストレイン
- 回転コンストレイン
- 方向コンストレイン

グラフ内の計算ノード

- リニア補間
- 簡易ベジェ補間
- ベジェ補間（DrivenKey）
- RBF補間（多次元DrivenKey）
- エクスプレッション（式）

など・・・

ノードデータ配列



異なる型データを参照する抽象型配列

struct で実装したいが・・・

- ノードタイプによってサイズが異なる。
- 可変長パラメータを持つ一部のものは、同じタイプでもサイズが異なる。
- シリアライズ対応が必要なため、特殊な実装にはしにくい。

ノードデータは class で実装

（各ノード class 内にデータ実体の struct を置く）

とあるノードデータ実装の概観

```
[Serializable]
public class ExampleNode : KDNodeData
{
    [Serializable]
    public struct Runtime
    {
        実行時にそのまま使うパラメータ情報
    }
    public Runtime RT;

    実行時の初期化のために必要な情報や、可変長パラメータなど
}
```

それでも、なるべく固定長に

- 可変長パラメータのうち、最大数を決めても実用上問題なさそうなものは固定長にした
  - アセットインポート時に上限を超えていないかチェック
- 結果的に可変長パラメータは２つだけになった
  - 「エクスプレッション」ノードのバイトコード配列
  - 「RBF補間」ノードのウェイト配列

固定長にする例

```
[Serializable]
public unsafe struct SrcQSize
{
    [SerializeField]fixed float reserved[4 * 8];
}

[Serializable]
public class ExampleNode : KDNodeData
{
    [Serializable]
    public struct Runtime {
        public SrcQSize SourceOffsets;  // quaternion * N
        public int NumSources;
    }
    public Runtime RT;

    public static unsafe void ExampleProc(Runtime* pData)
    {
        quaternion* offsets = (quaternion*)&pData->SourceOffsets;
        for (int i=0; i<pData->NumSources; ++i) {
            Proc(offsets[i], ...);
        }
    }
}
```

quaternion 最大8個

最大数は sizeof で分かるので、
データファイルインポート時に
超えていないかチェック。

本当の型のポインタに
キャストして参照

インポーター実装

- エディターだけの処理なので、
  C# らしく富豪的プログラミングを味わう！
- たとえば、いろいろなノードタイプの型の処理で、
  ランタイムでは使う気のしないリフレクションなど便利だった。

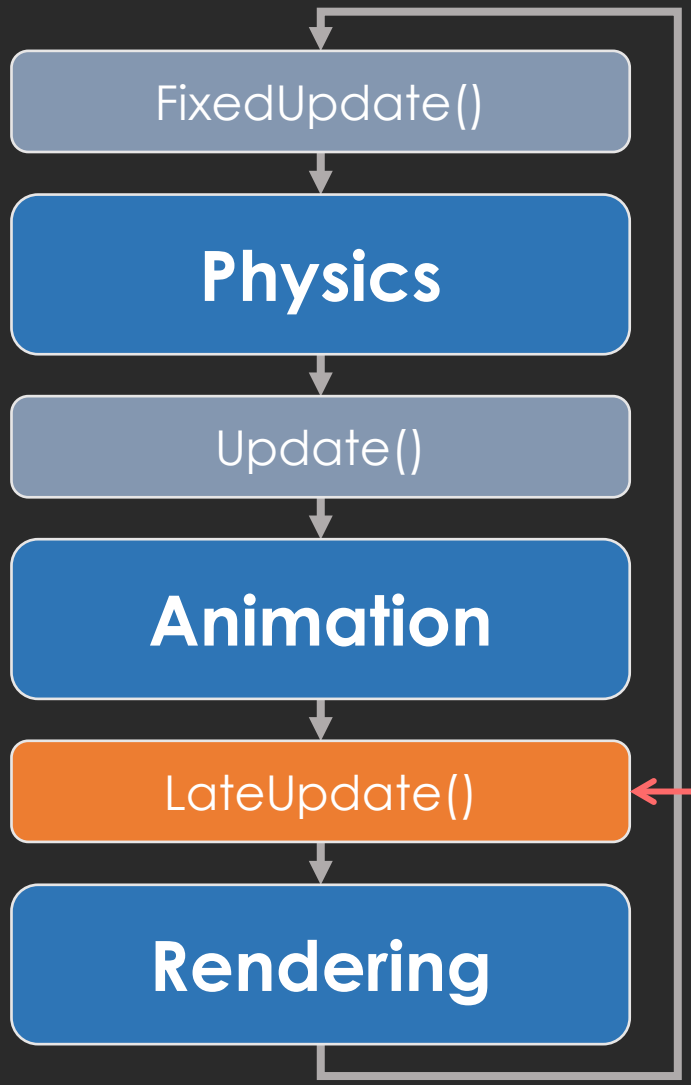
実装

MT コンポーネント

コンポーネントの概要

- MonoBehaviour 継承 class はシンプルに
  - 別 class に実装された処理を呼び出すだけ
  - ニーズに応じてコンポーネントをカスタマイズしやすく
- エディタ上でも常に動作
  - [ExecuteAlways] 属性を付加
- メインアニメーションの後処理
  - LateUpdate() で処理
- 素直にマネージドメモリで実装
  - アンマネージドメモリで実装しても良かったとは思っている
- Burst Direct Call を利用
  - 毎フレーム呼び出す計算処理を高速化

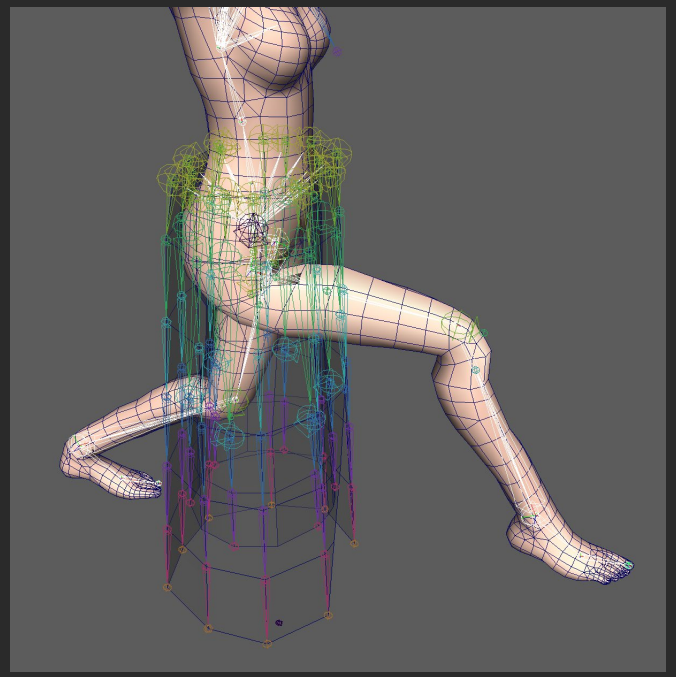
毎フレームの補助骨計算



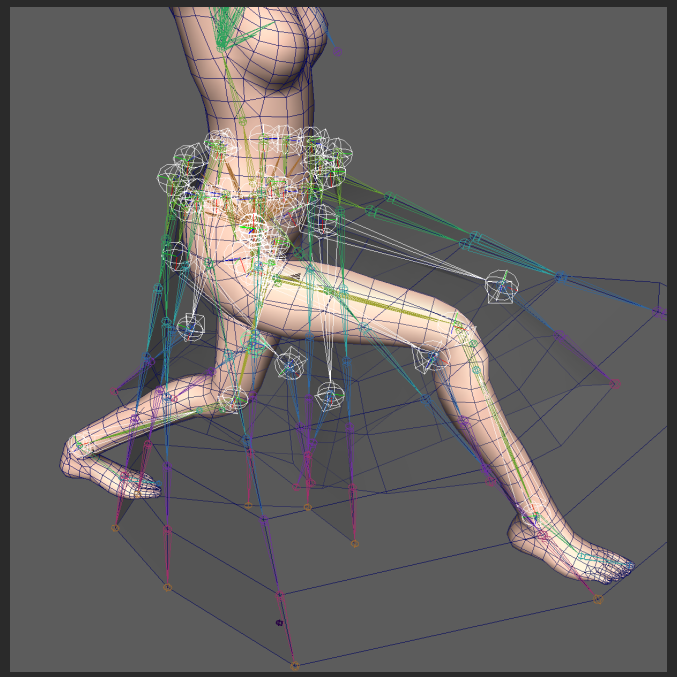
かなりの概略図です。
マニュアルには詳細な説明があります。

メインアニメーション結果を元に
補助骨を動かしたいからここ

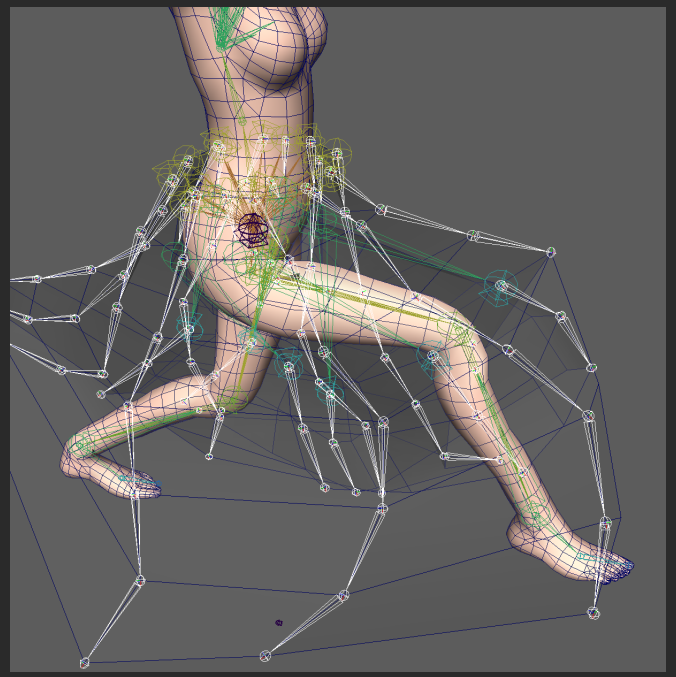
他のコンポーネントとの併用



メイン骨
アニメーション



補助骨

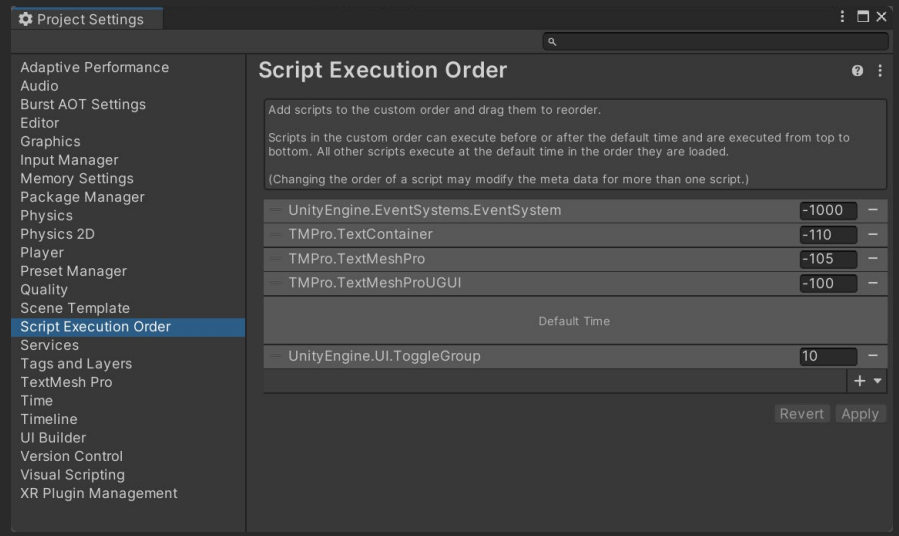


揺れ骨
（物理シミュレーション）

この順番はどう実現するのか？

コンポーネントの実行順

- 基本的に不定
- タイプに優先順位を指定可能
  - Project Settings > Script Execution Order
  - [DefaultExecutionOrder(番号)]
- 同じタイプ同士の順番は不定
  - そのため [DisallowMultipleComponent]
- 可能なら「統合コンポーネント」を書く
  - そのために実装をシンプルにしている




```
void LateUpdate()
{
    KineDriver.Compute();
    TheOtherOne.Compute();
}
```

コンポーネント実装概略

```
[HelpURL("http://example")]
[Icon( "Assets/Gizmos/SQEX/KineDriver/KineDriverData Icon.png")]
[AddComponentMenu("KineDriver/KineDriver MT")]
[DefaultExecutionOrder(0)]
[DisallowMultipleComponent]
[ExecuteAlways]
public class KineDriverMT : MonoBehaviour
{
    public KineDriverData[] KDIData;  // 複数KDIに対応

    KineDriverEvaluator Evaluator;

    public KineDriverMT()
    {
        Evaluator = new KineDriverEvaluator();
    }

    public void Dispose()
    {
        Evaluator = new KineDriverEvaluator();
    }

    void Awake()
    {
        if (KDIData == null)
            KDIData = new KineDriverData[0];
    }
```

← 実装本体

```
#if UNITY_EDITOR
    void OnDisable()
    {
        // エディタでは無効化されたらメモリ解放
        if (! UnityEditor.EditorApplication.isPlaying)
            Dispose();
    }
#else
    void OnEnable()
    {
        // ランタイムでは初回有効化時に初期化
        if (! Evaluator.IsActive)
            Evaluator.Bind(transform, KDIData);
    }
#endif

    void LateUpdate()
    {
#if UNITY_EDITOR
        // エディタではデータの再アサインに対応
        // （KDIの持つハッシュ値でチェック）
        Evaluator.Bind(transform, KDIData);
#endif
        Evaluator.Compute();
    }
}
```

初期化処理

毎フレームの補助骨計算

KineDriverEvaluator 概観

```
public class KineDriverEvaluator
{
    KineDriverNode[] Nodes;

    省略

    public bool IsActive { get => this.Nodes.Length > 0; }

    public void Bind(Transform root, KineDriverData[] kdis)
    {

    }

    public void Compute()
    {
        foreach (var node in this.Nodes)
            node.Compute();
    }
}
```

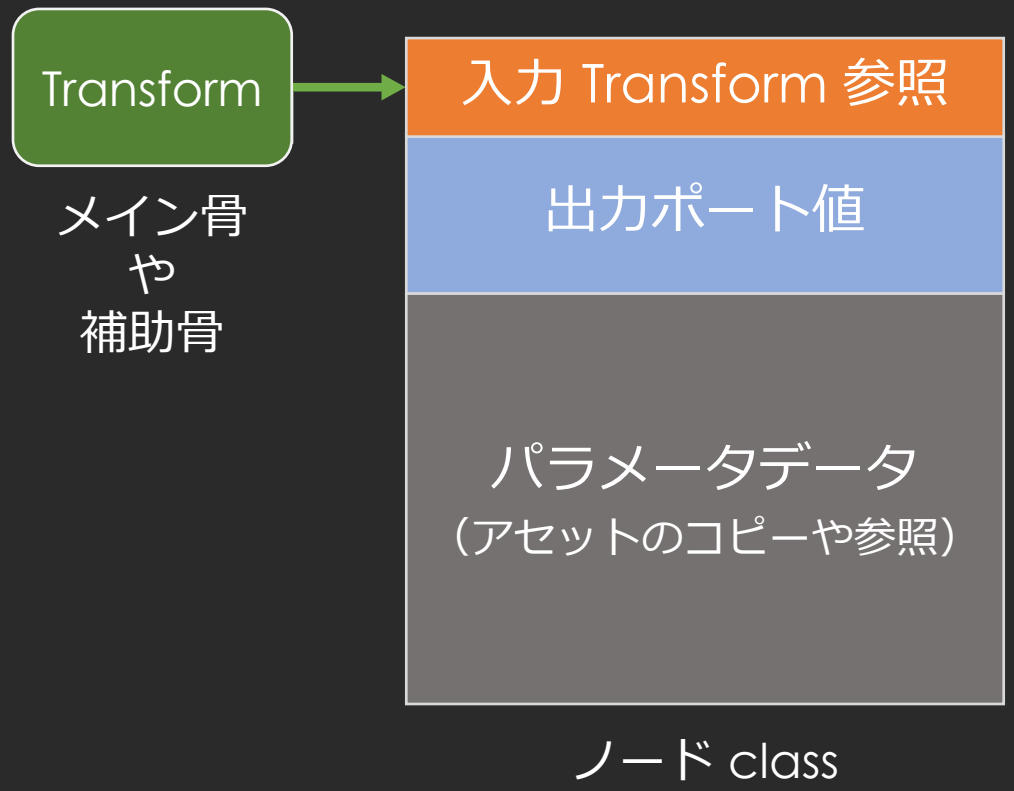
← MTコンポーネントに限り、手抜きで、
実行用メモリもマネージドヒープで実装。

初期化処理
- 実行用メモリ確保とデータロード
- スケルトンと KineDriver データのバインド

← 毎フレーム呼び出される計算。
ノード配列を順番に呼び出すだけ。

ノードの計算処理

Transform 入力ノード

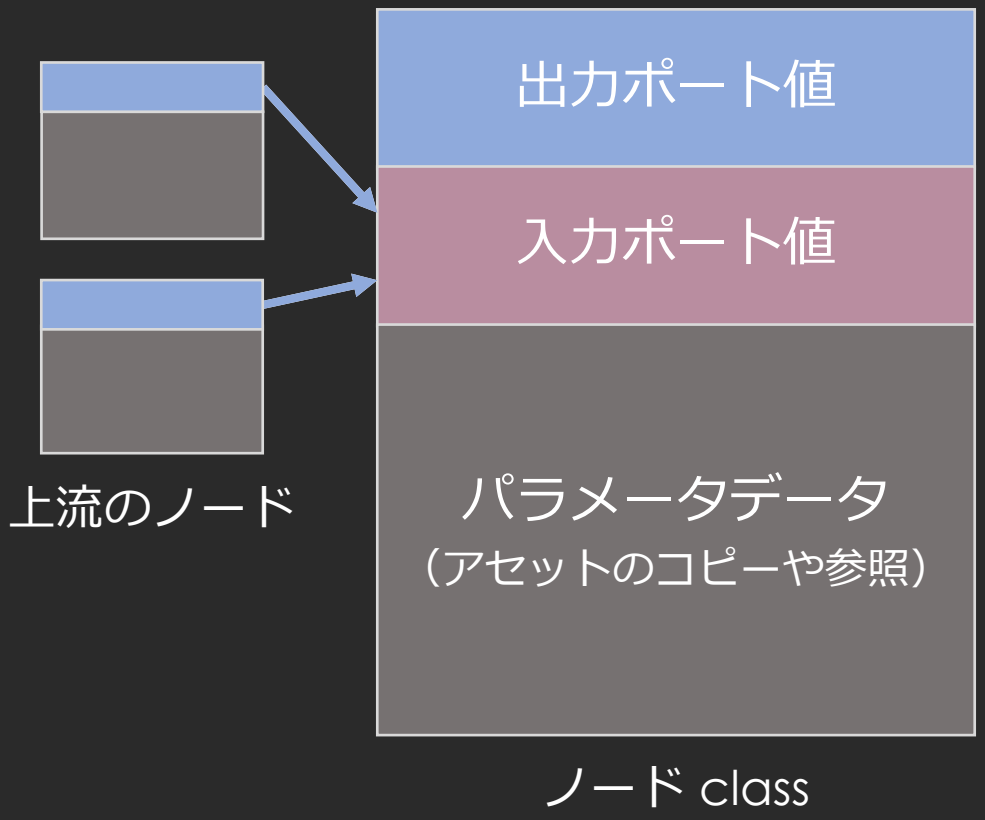


1. 参照するジョイントから Transform 値を得る
2. 計算をする
3. 出力ポートに値を書き込む

※この図はイメージです。実際とは細部が異なります。

ノードの計算処理

グラフ内の計算ノード

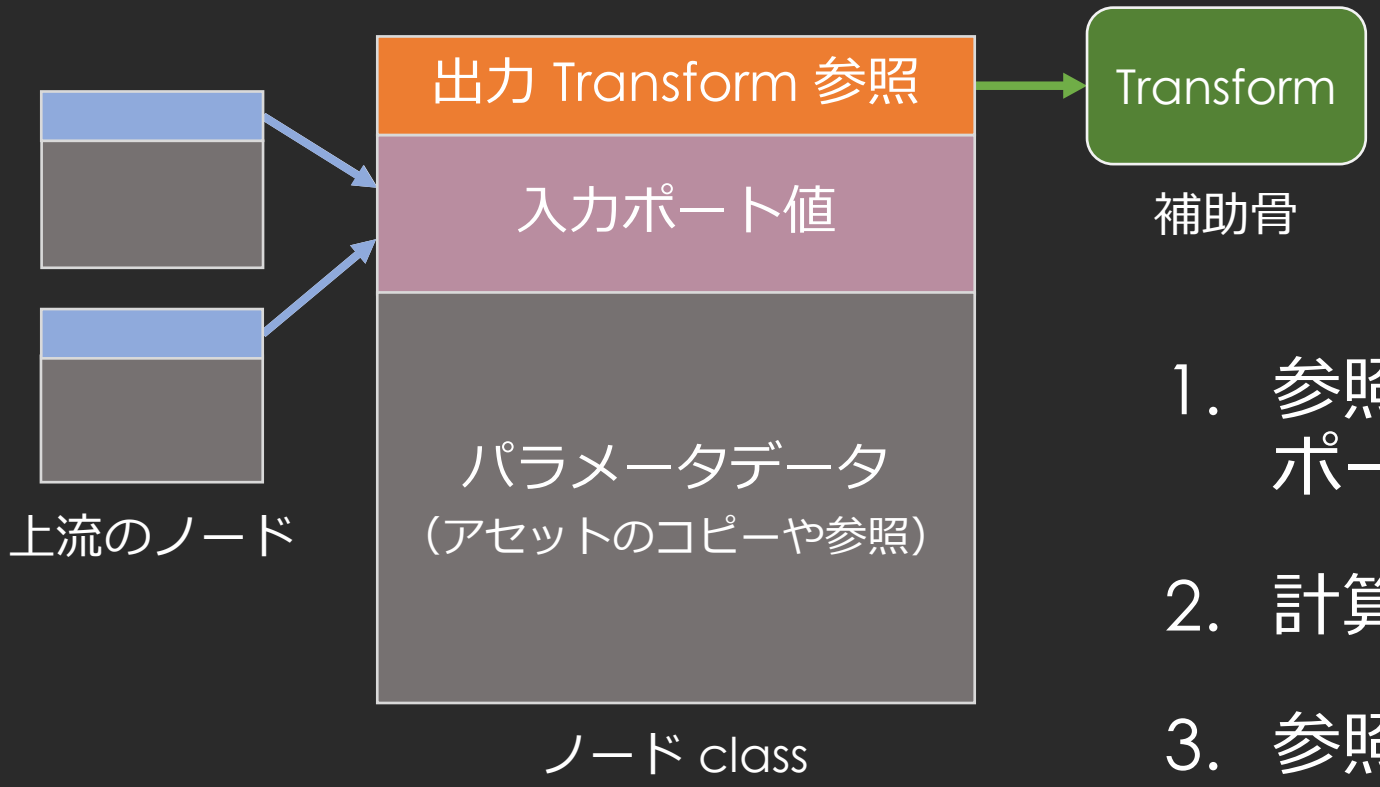


※この図はイメージです。実際とは細部が異なります。

1. 参照する上流のノードの出力ポートから値を一通り得る
2. 計算をする
3. 出力ポートに値を書き込む

ノードの計算処理

Transform 出力ノード



1. 参照する上流のノードの出力ポートから値を一通り得る
2. 計算をする
3. 参照するジョイントの Transform に値を書き込む

※この図はイメージです。実際とは細部が異なります。

Burst Direct Call

Job System でなくても Burst は使える

- 関数単位で使える

  class/struct とスタティックメソッドの両方に
  [BurstCompile] 属性を付けるだけ。便利。

- 引数で受け渡しできるのは Blittable 型

  値そのままか、参照かポインタで渡す。

- マネージドヒープの値も fixed してポインタで渡せる

各ノードの計算は Burst 関数として実装し、全コンポーネントで共用

Burst コード例

```
/// KDIアセット: Expression ノードデータクラス
[Serializable]
[BurstCompile(DisableSafetyChecks=true, FloatMode=FloatMode.Fast)]
public class ExprData : KDNodeData
{
    データフィールドなど

    [BurstCompile(DisableSafetyChecks=true, FloatMode=FloatMode.Fast)]
    [SkipLocalsInit]
    public static unsafe void Compute([NoAlias] out float4 result, [NoAlias] float4* inputs, [NoAlias] Code* bytecode, int nInputs, int nCodes)
    {
        ノードの計算処理（Burst）
    }
}

/// MTコンポーネント: Expression ノードクラス
public class Expr : KineDriverNode
{
    実行時バッファのフィールド

    public override unsafe void Compute()
    {
        fixed (float4* inputs = &_inputs[0]) {
        fixed (ExprData.Code* bytecode = &_data.Bytecode[0]) {
            ポートの入力処理

            ExprData.Compute(out _output, inputs, bytecode, _data.Input.Length, _data.Bytecode.Length);
        }
        }
    }
}
```

両方に指定が必要

ヒープで実装してしまったので fixed

HPC# (High Performance C#)

- Burst を使うために機能が大幅に制限された C#
  - クラス 使えない
  - 例外 使えない
  - デリゲート 使えない
  - 様々なクラスライブラリ 使えない
  - Unity® の機能もほとんど使えない
  - 他にもいろいろ使えない
- ポインタが使えるので、C だと思って書けば問題無し！
- Unity.Mathematics が使える！

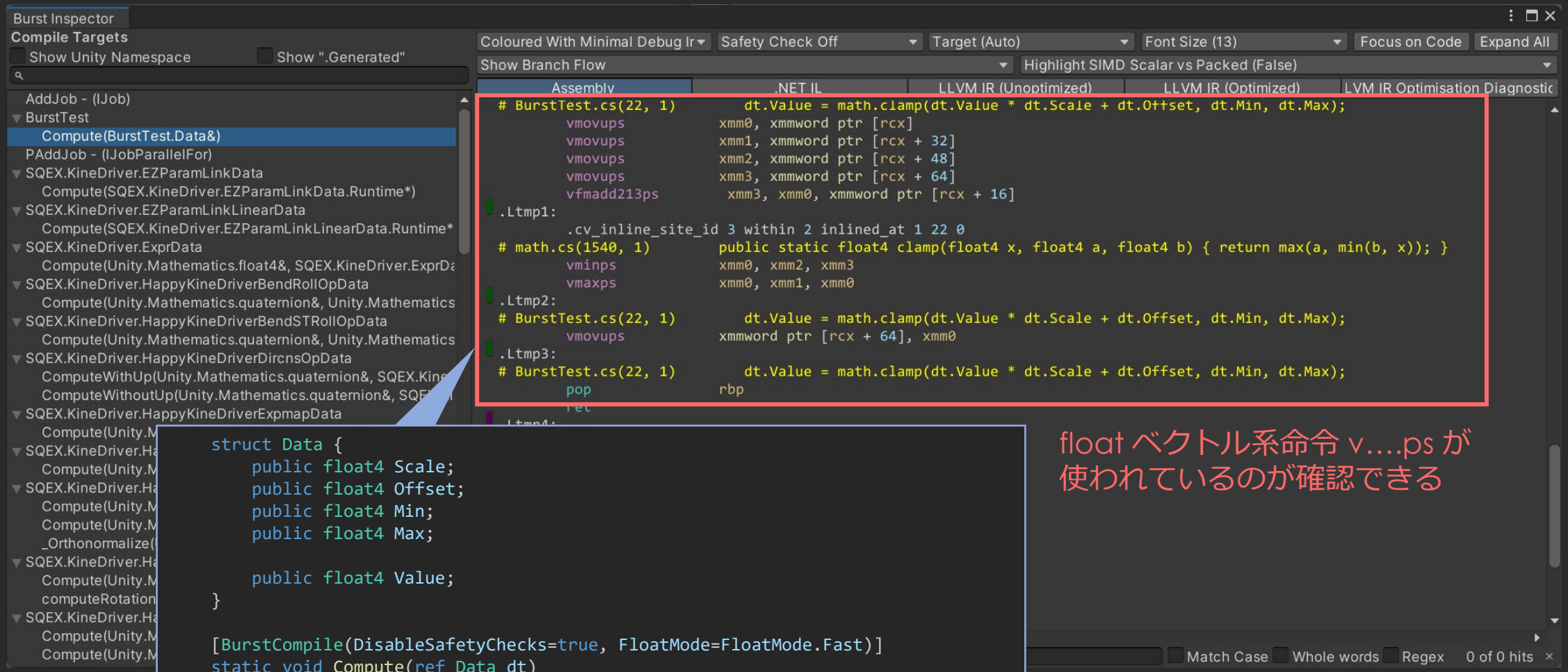
Unity.Mathematics

- Burst のための数学ライブラリ
  - struct だけでできている
  - Burst だと SIMD 最適化される
  - Burst でないと遅そうだけど動く（普通の C# 実装コードも見られる）
- 結構分かりやすい
  - ドキュメントはきちんとしているし、ソースも見られる
  - 型名やメソッド名などは規則的で大変分かりやすい
  - float3, float4, quaternion, float4x4（行列） など豊富にそろっている
  - シェーダー言語のような swizzle 記法もできる　例) float4.xz

Burst オプション

- DisableSafetyChecks=true
  UIで設定しなくても、セーフティチェック無効化。
- FloatMode=Fast
  計算の正確な順序や NaN 値の処理を保証せず、最適化されやすくする。
- [SkipLocalsInit]
  ローカル変数の 0 初期化を無効化。
- [NoAlias]
  引数エイリアス（引数で渡した複数のポインタが同じメモリ位置を指す可能性）が無いことをコンパイラに伝える。
- Hint.Likely, Unlikely, Assume
  ブール条件の可能性をコンパイラに伝える。

Burst Inspector




```
struct Data {
    public float4 Scale;
    public float4 Offset;
    public float4 Min;
    public float4 Max;

    public float4 Value;
}

[BurstCompile(DisableSafetyChecks=true, FloatMode=FloatMode.Fast)]
static void Compute(ref Data dt)
{
    dt.Value = math.clamp(dt.Value * dt.Scale + dt.Offset, dt.Min, dt.Max);
}
```

float ベクトル系命令 v….ps が
使われているのが確認できる

実装

Job コンポーネント

コンポーネントの概要

- MonoBehaviour 継承 class はシンプルに
  - 別 class に実装された処理を呼び出すだけ
  - ニーズに応じてコンポーネントをカスタマイズしやすく
- エディタ上でも常に動作
  - [ExecuteAlways] 属性を付加
- メインアニメーションの後処理
  - LateUpdate() で処理
- C# Job System と Burst で実装
  - アンマネージドメモリを使用

MTと同じ

他コンポーネントとの組み合わせの考え方もほぼ同じ

MTと違う点

複数キャラクターを並列処理可能

C# Job System を使う

- ジョブの struct を実装（interface を継承）
  - IJob
  - IJobParallelFor
  - IJobParallelForTransform
- ジョブ間の依存関係（スケジュール）を組める
- アンマネージドメモリの利用が必須
  - Blittable 型のみ使用可
  - NativeArray を使おう
- Burst コンパイルできる
  - Job System 自体が強い制約を伴うため、普通に使える

IJob

１つのタスク

簡単なジョブの例

```
[BurstCompile]
public struct AddJob : IJob
{
    public NativeArray<float> result;
    [ReadOnly] public float a;
    [ReadOnly] public float b;

    public void Execute()
    {
        result[0] = a + b;
    }
}
```

ジョブの結果を受け取るために
NativeArray が必要

ジョブの呼び出し

```
public class AddJobTest : MonoBehaviour
{
    void Start()
    {
        var job = new AddJob() {
            result = new NativeArray<float>(1, Allocator.TempJob),
            a=1f, b=2f,
        };

        JobHandle hdl = job.Schedule();  // ジョブをスケジュール

        hdl.Complete();  // ジョブの終了まで待つ

        Debug.Log(job.result[0]);  // 3f

        job.result.Dispose();  // メモリの解放
    }
}
```

ここで待ってしまったら
ジョブの意味が無いが、
これはサンプルなので、
その場で結果が得たいから。

IJobParallelFor

並列実行可能な繰り返しタスク

簡単なジョブの例

```
[BurstCompile]
public struct PAddJob : IJobParallelFor
{
    public NativeArray<float> result;
    [ReadOnly] public NativeArray<float> a;
    [ReadOnly] public NativeArray<float> b;

    public void Execute(int i)
    {
        result[i] = a[i] + b[i];
    }
}
```

並列で呼ばれる可能性があるので、
インデックス間に依存があってはならない

ジョブの呼び出し

```
public class PAddJobTest : MonoBehaviour
{
    void Start()
    {
        var job = new PAddJob() {
            result = new NativeArray<float>(2, Allocator.TempJob),
            a = new NativeArray<float>(2, Allocator.TempJob),
            b = new NativeArray<float>(2, Allocator.TempJob),
        };
        job.a[0] = 1f;  job.b[0] = 2f;
        job.a[1] = 3f;  job.b[1] = 4f;

        JobHandle h = job.Schedule(job.result.Length, 1);  // ジョブをスケジュール
        h.Complete();  // ジョブの終了まで待つ

        Debug.Log(job.result[0]);  // 3f
        Debug.Log(job.result[1]);  // 7f

        job.result.Dispose();  // メモリの解放
        job.a.Dispose();
        job.b.Dispose();
    }
}
```

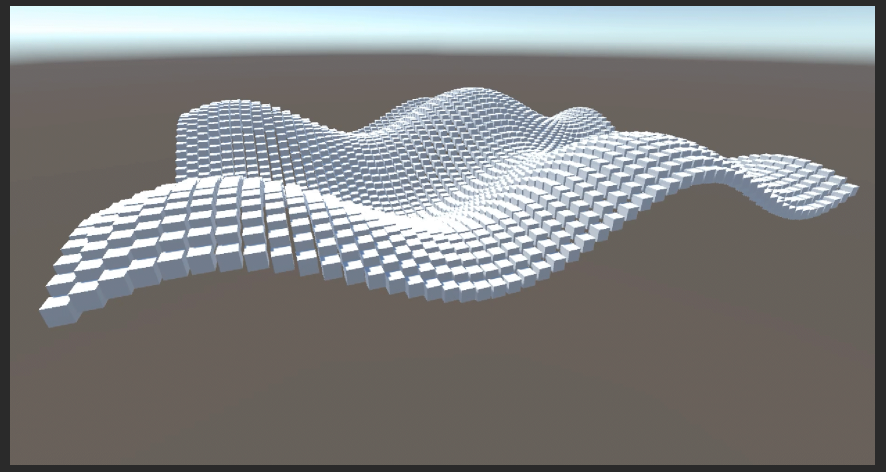
IJobParallelForTransform

並列実行可能かもしれない Transform 入出力タスク

簡単なジョブの例

```
[BurstCompile]
public struct WaveJob : IJobParallelForTransform
{
    [ReadOnly] public float t;
    [ReadOnly] public float f;
    [ReadOnly] public float h;

    public void Execute(int i, TransformAccess xform)
    {
        var v = xform.localPosition;
        v.y = (math.sin(t + v.x * f) + math.sin(t + v.z * f)) * h;
        xform.localPosition = v;
    }
}
```

struct



ジョブの呼び出し

```
public class WaveJobTest : MonoBehaviour
{
    public GameObject Prefab;
    [Range(0f, 10f)] public float Speed = 5f;
    [Range(.01f, 1f)] public float Frequency = .18f;
    [Range(0f, 10f)] public float Amplitude = 3f;
    TransformAccessArray _transformAccessArray;
    WaveJob _job;
    JobHandle _handle;

    void Start()
    {
        var n = (int)math.sqrt(3000);
        var s = (float)(n - 1) * -0.6f;
        _transformAccessArray = new TransformAccessArray(n * n, 1);
        for (int x=0; x<n; ++x) {
            for (int z=0; z<n; ++z) {
                var obj = Instantiate(Prefab);
                obj.transform.localPosition = new Vector3(s + x * 1.2f, 0f, s + z * 1.2f);
                _transformAccessArray.Add(obj.transform);
            }
        }
    }

    void OnDestroy()
    {
        _handle.Complete();
        _transformAccessArray.Dispose();
    }

    void Update()
    {
        _handle.Complete();
        _job.t += Time.deltaTime * Speed;
        _job.f = Frequency;
        _job.h = Amplitude;
        _handle = _job.Schedule(_transformAccessArray);
        JobHandle.ScheduleBatchedJobs();
    }
}
```

扱う Transform の登録

アンマネージドリソース解放

前フレームに投げたジョブの同期

ジョブをスケジュール

ジョブを走らせる

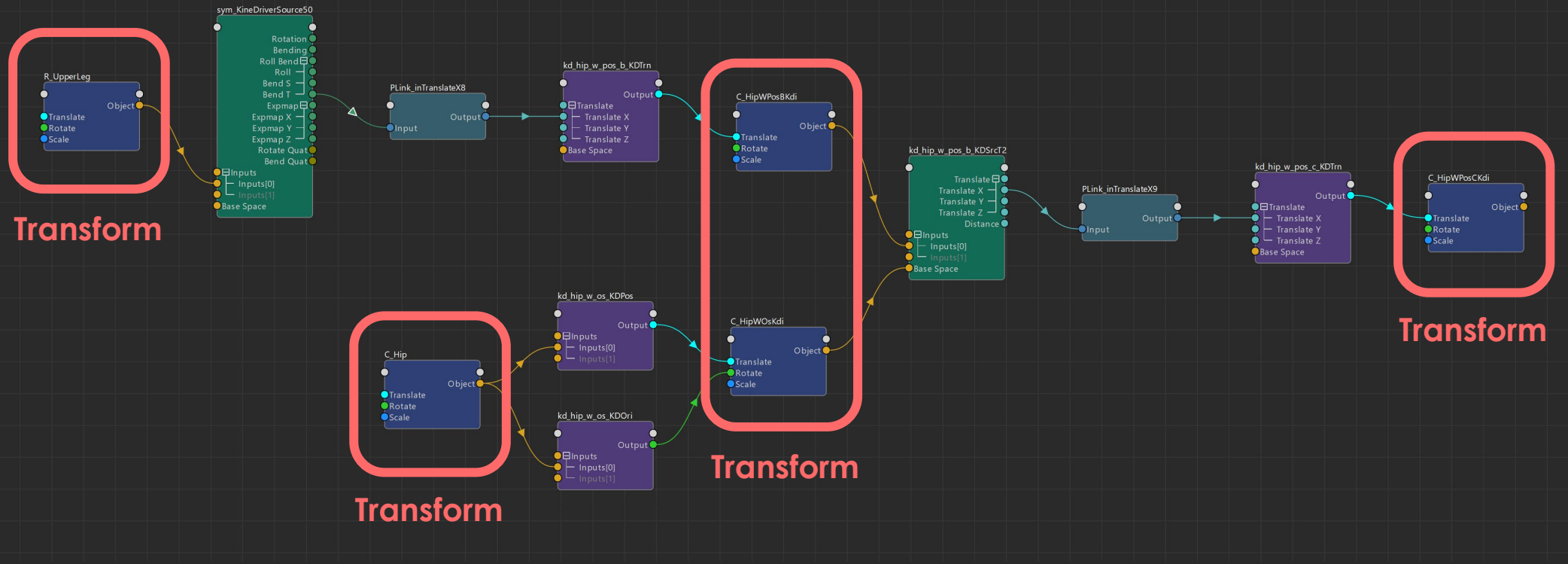
KineDriver の場合

- ノードを順番に計算する（ノード計算は並列化しない）
  - IJobParallelFor ではなく IJob が適する
- Transform の入力と出力を行う
  - IJobParallelForTransform の使用が必要
  - 並列化させたいわけではない
    - IJobPrallelForTransform で並列化されるのは、階層が別のもの
    - スケルトンの中の各ジョイントは並列化されない → 好都合

このようにスケジュールできそう？

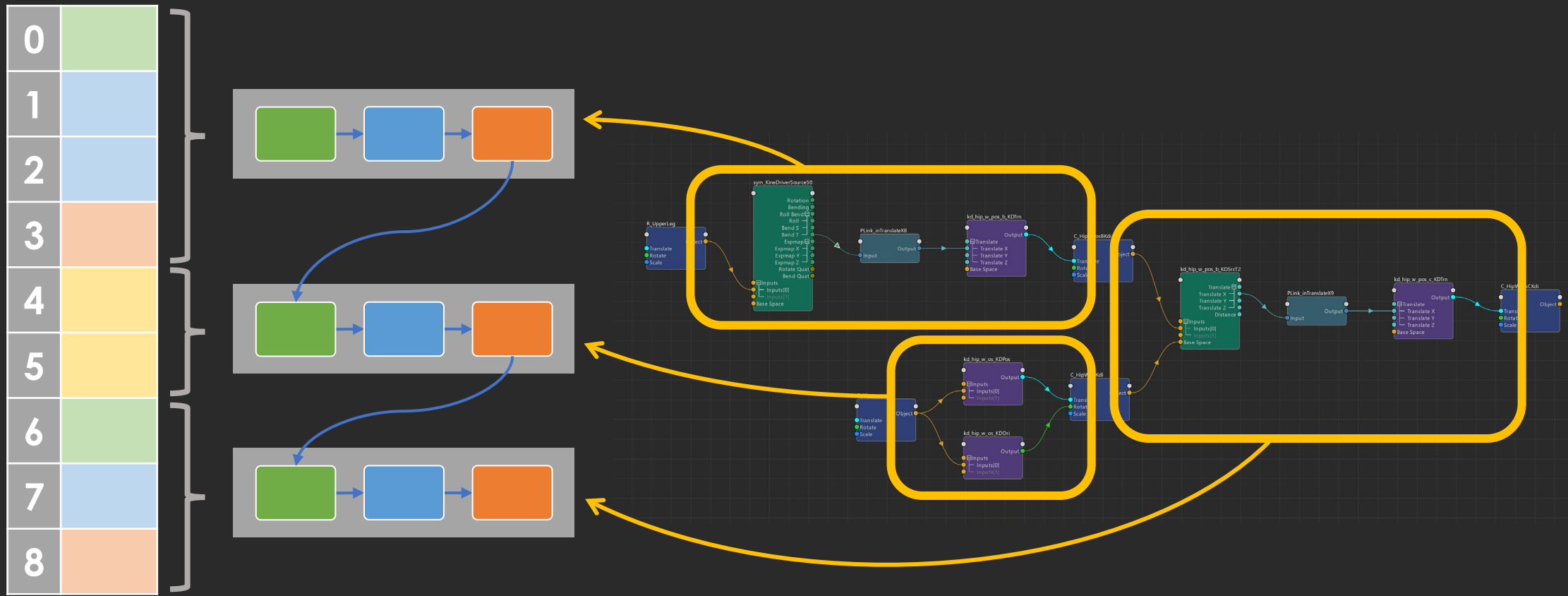
しかし「補助骨の補助骨」が結構ある

- ある補助骨が、さらに別の補助骨を動かす
- ある補助骨の階層下の骨が、さらに別の補助骨を動かす



ジョブセットを分けることに

入力、計算、出力の３ジョブをセットとし、
途中で Transform 再評価が必要なところで分割する。

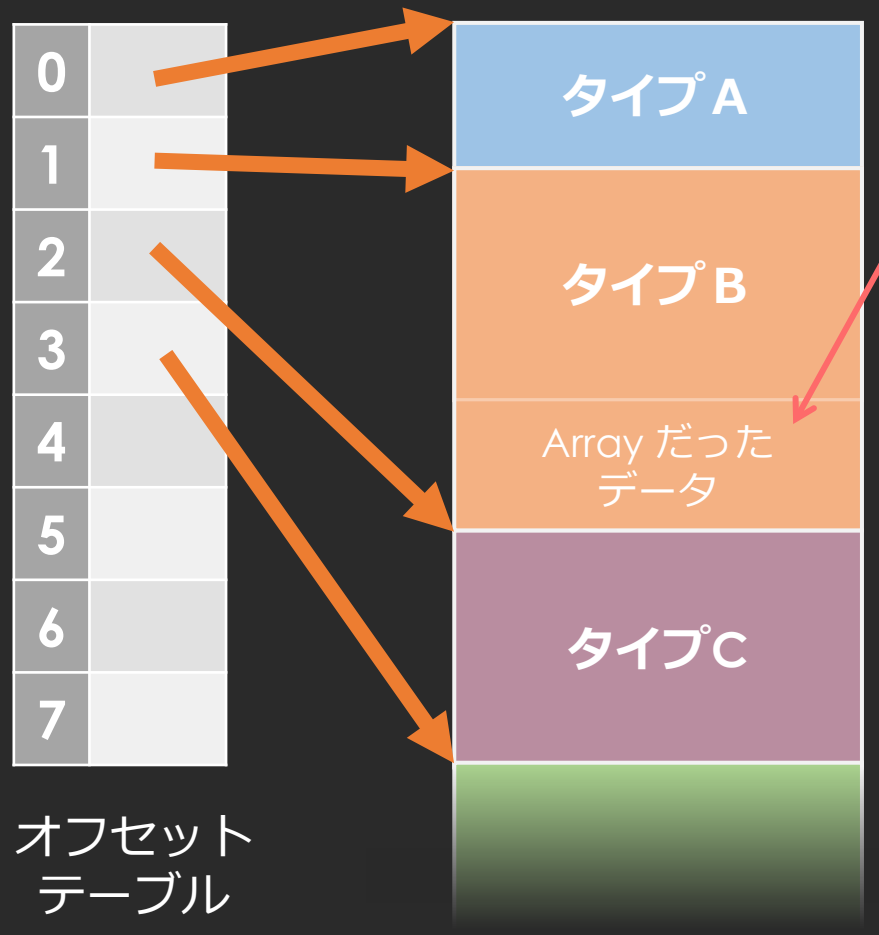


TransformAccess

	Property	Transform	TransformAccess
Local	localPosition	Read Write	Read Write
	localRotation	Read Write	Read Write
	localScale	Read Write	Read Write
Global	position	Read Write	Read Write
	rotation	Read Write	Read Write
	lossyScale	Read	
Matrix	localToWorldMatrix	Read	Read
	worldToLocalMatrix	Read	Read

Global Scale を得られないので、
メインスレッドで Transform から得ておきジョブに渡すことに

アンマネージドメモリを使用



アセットデータでは Array にしたものも
全てサイズが確定している

- 連続した領域にまとめて確保
- 初期化時に NativeArray 1 つだけ確保、
  あとはポインタでアクセス
- ジョブにポインタも持たせられる
  - [NativeDisableUnsafePtrRestriction] を指定
  - [NoAlias] もなるべく指定

※ 後述する Transform 管理配列もこの中にまとめた

コンポーネント実装概略

```
[HelpURL("http://example")]
[Icon( "Assets/Gizmos/SQEX/KineDriver/KineDriverData Icon.png")]
[AddComponentMenu("KineDriver/KineDriver Job")]
[DefaultExecutionOrder(0)]
[DisallowMultipleComponent]
[ExecuteAlways]
public class KineDriverJob : MonoBehaviour
{
    public KineDriverData[] KDIData;  // 複数KDIに対応

    KineDriverScheduler Scheduler;   ← 実装本体

    public KineDriverJob()
    {
        Scheduler = new KineDriverScheduler();
    }

    public void Dispose()
    {
        Scheduler.Dispose();
    }

    void OnDestroy() => Dispose();   ← 後始末が必要

    void Awake()
    {
        if (KDIData == null)
            KDIData = new KineDriverData[0];
    }
```

MTとほぼ同じ構造

```
#if UNITY_EDITOR
    void OnDisable()
    {
        // エディタでは無効化されたらメモリ解放
        if (! UnityEditor.EditorApplication.isPlaying)
            Dispose();
    }

#else
    void OnEnable()
    {
        // ランタイムでは初回有効化時に初期化
        if (! Scheduler.IsActive)
            Scheduler.Bind(transform, KDIData);
    }
#endif

    void LateUpdate()
    {
#if UNITY_EDITOR
        // エディタではデータの再アサインに対応（ハッシュ値チェック）
        Scheduler.Bind(transform, KDIData);
#endif
        Scheduler.Handle.Complete();
        Scheduler.Schedule();
        JobHandle.ScheduleBatchedJobs();
    }
}
```

初期化処理

← ジョブを
スケジュール

KineDriverScheduler 概観

Bind()

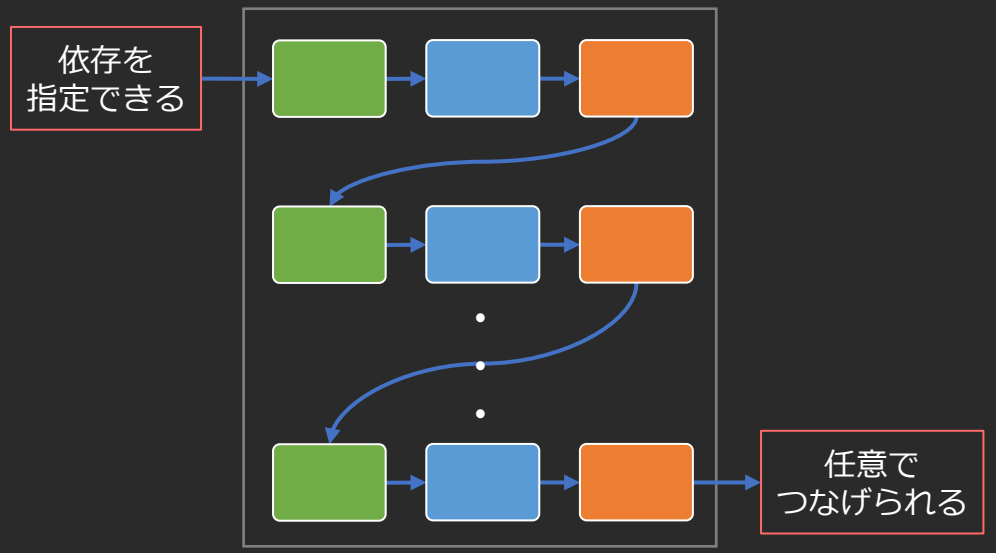
- 実行用メモリ確保とデータロード
- スケルトンと KineDriver データのバインド

Dispose()

- アンマネージドリソースの解放

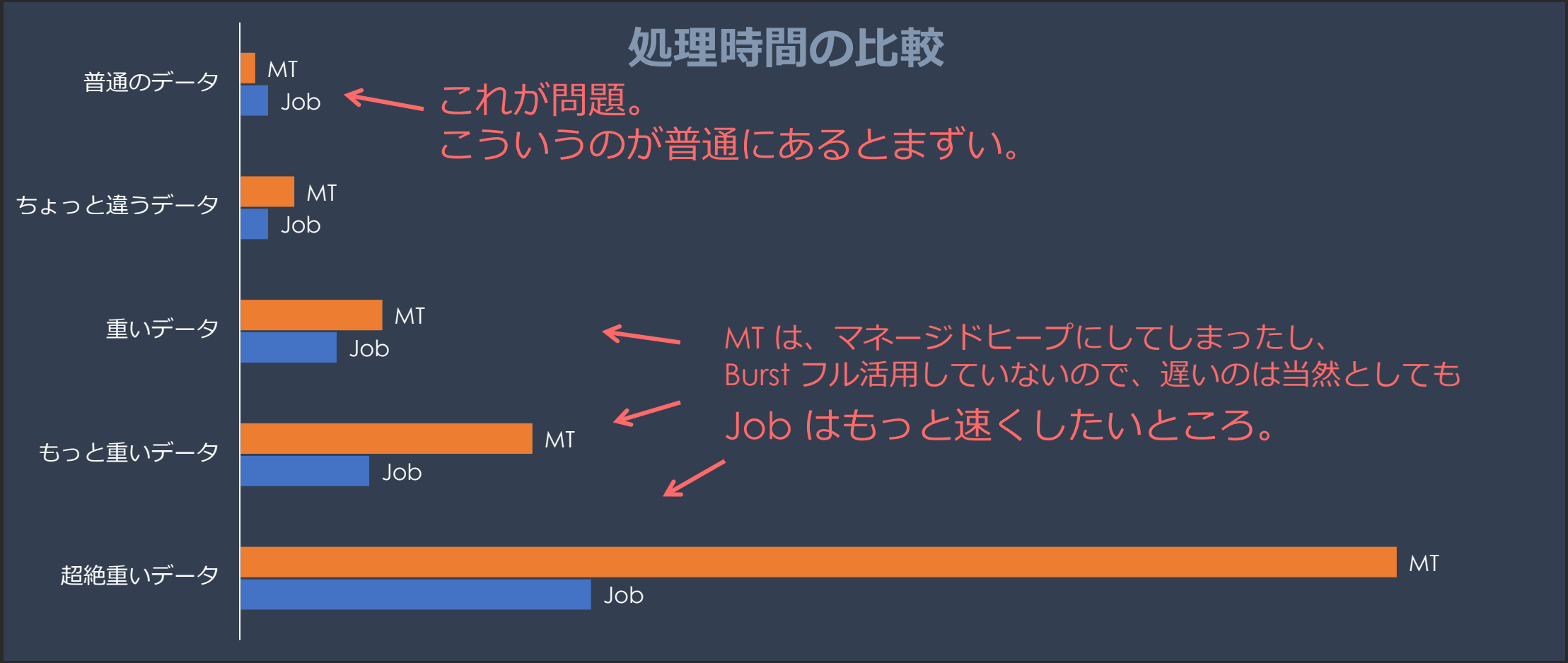
Schedule()

- Transform から Global Scale を取得
- ジョブをスケジュール
  - 依存する JobHandle があれば受け取れる
  - その先のために JobHandle を返す



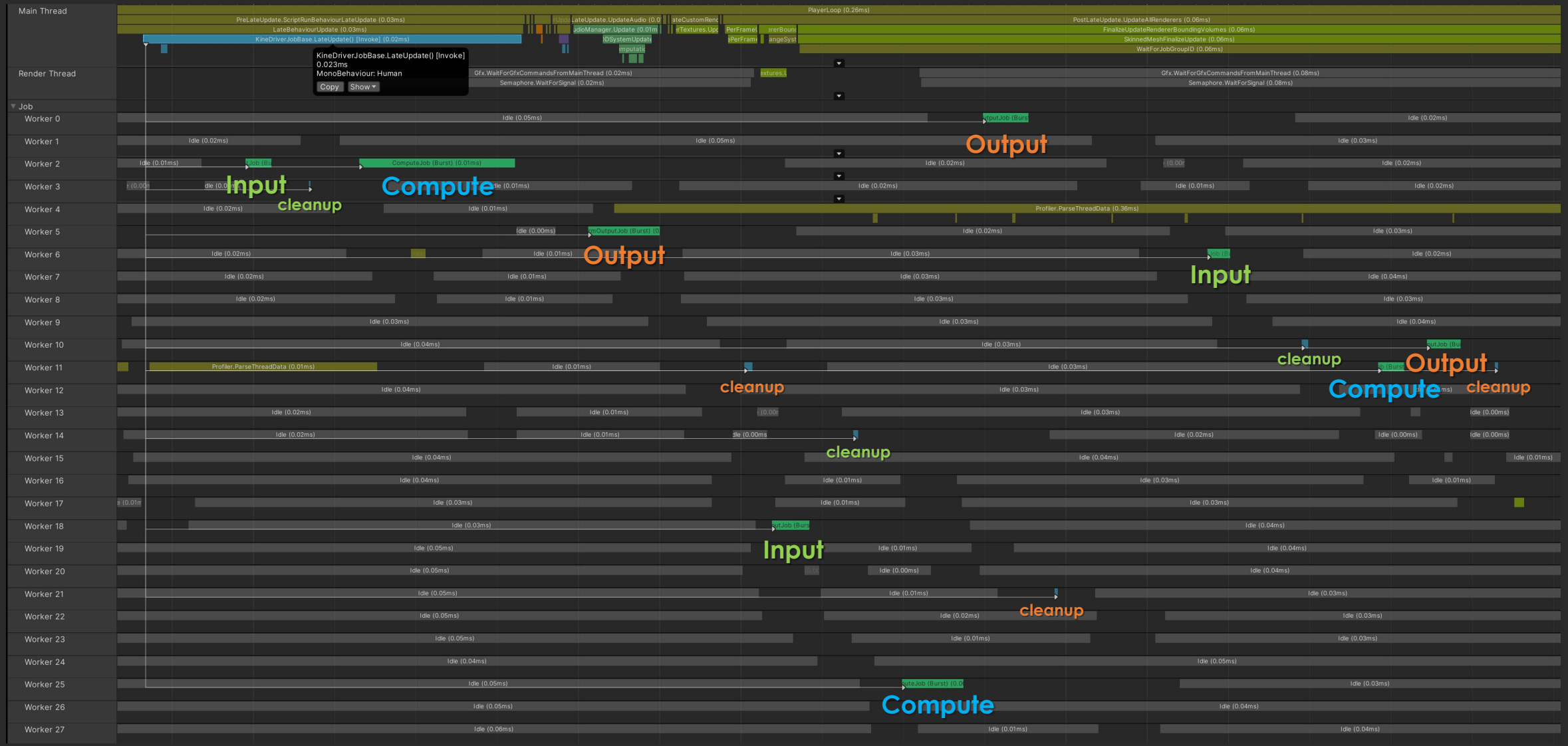
あまり速くない

重いデータは MT より速くなったが、普通のデータは MT より遅い



プロファイル

Intel® Core™ i9-13900KF（Raptor Lake）

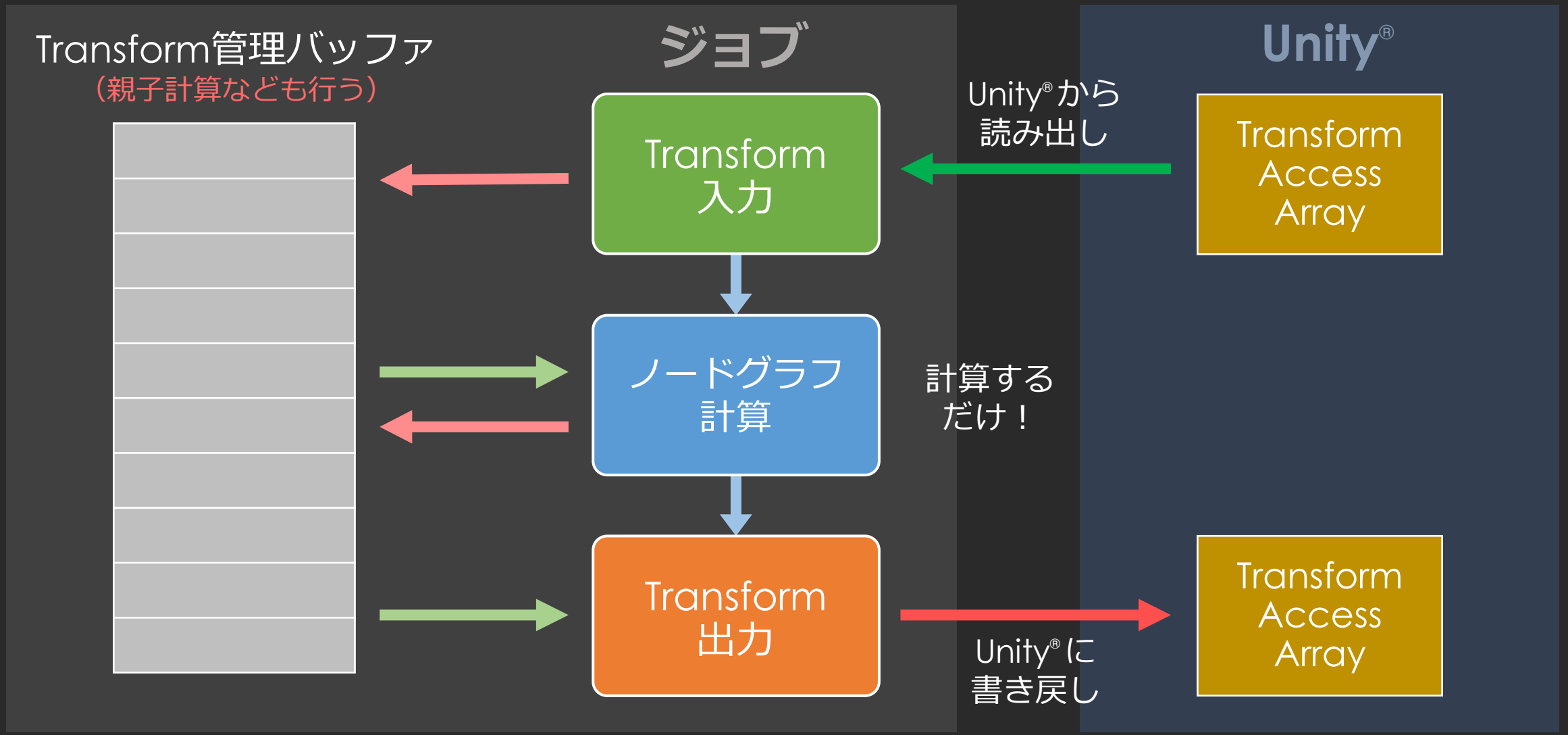


ボトルネックと改善策

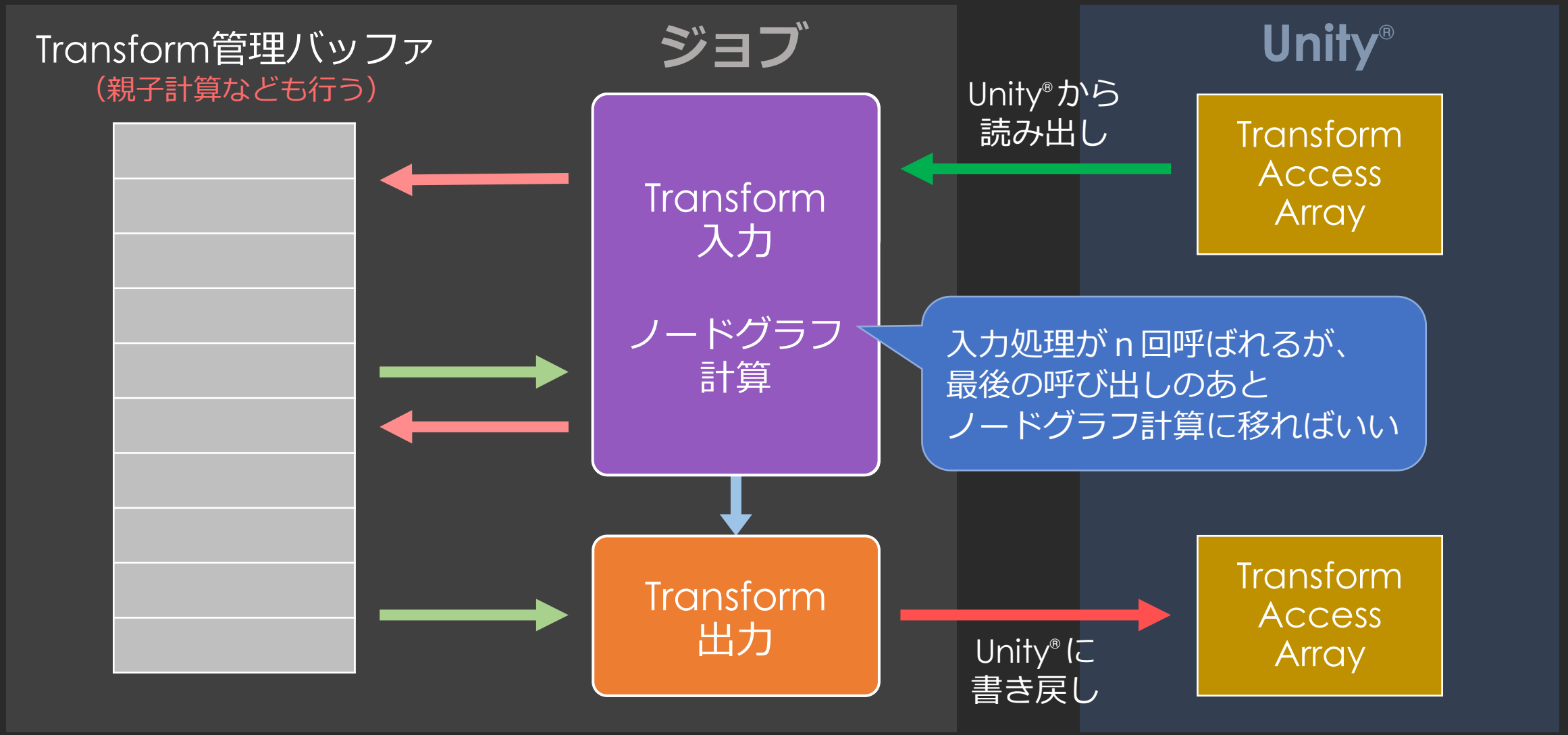
- ジョブのオーバーヘッド
  - 小さなタスクなのにジョブが多すぎ
  - Idle が多く効率悪そう
- Transform アクセスが結構な負荷
  - 特に Output

- ジョブの数を減らせないか？
- Transform のアクセス回数を減らせないか？

ジョブ中の Transform を自前管理

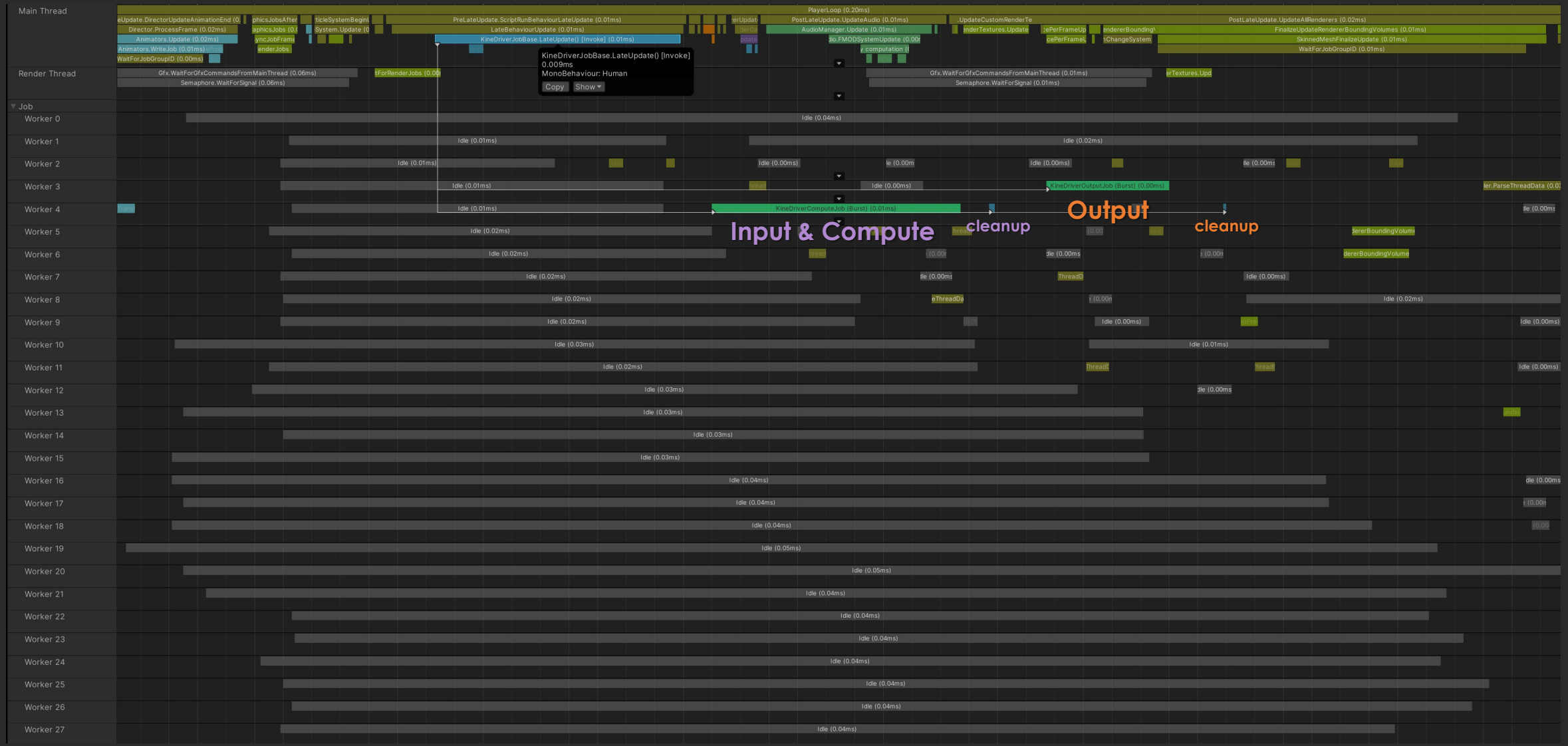


ダメ押しで、ジョブをもう１つ減らす

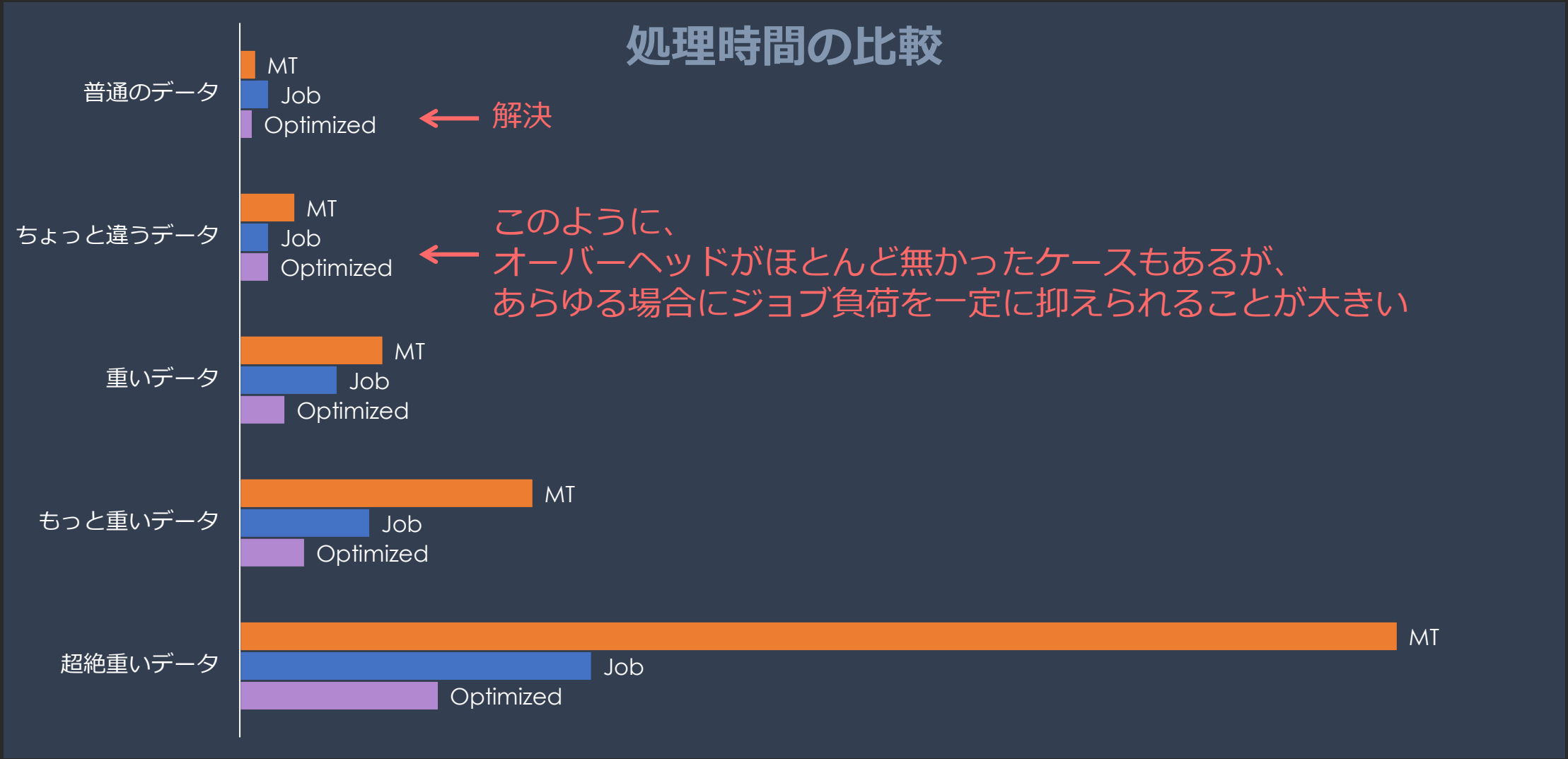


常にジョブが２個になった

Intel® Core™ i9-13900KF（Raptor Lake）



ジョブ最適化の結果



Transform 自前管理は効果的

- 速度アップに効果的だったこと
  - ジョブの削減
  - Transform アクセスの削減
  - Global 値の入出力の削減

アクセスする Transform の上位（親）が管理されていれば、親子計算もできるため Local 値の入出力で済ますことができる

- システム依存を小さく
  - 自身のがんばりでなんとかできる部分を増やす
  - 負荷の多くは、純粋な計算処理よりも、大抵、シーン要素の I/O
  - 特に、計算そのものが大して重くない場合に顕著
  - DCCツールのプラグイン開発でもいえる話

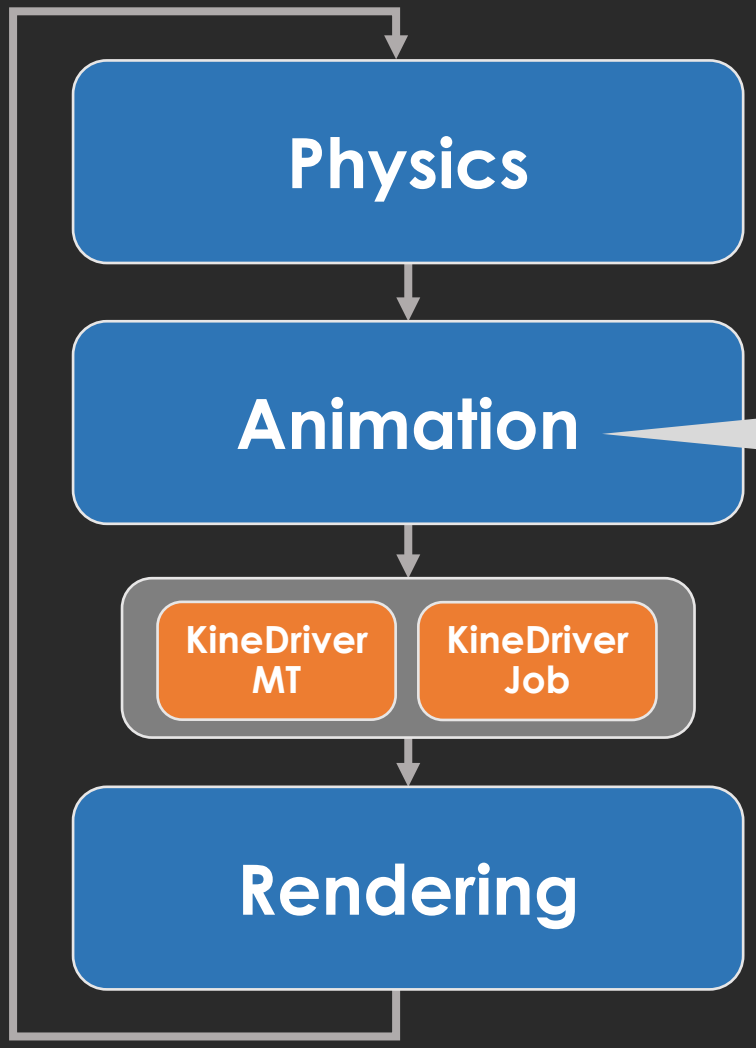
実装

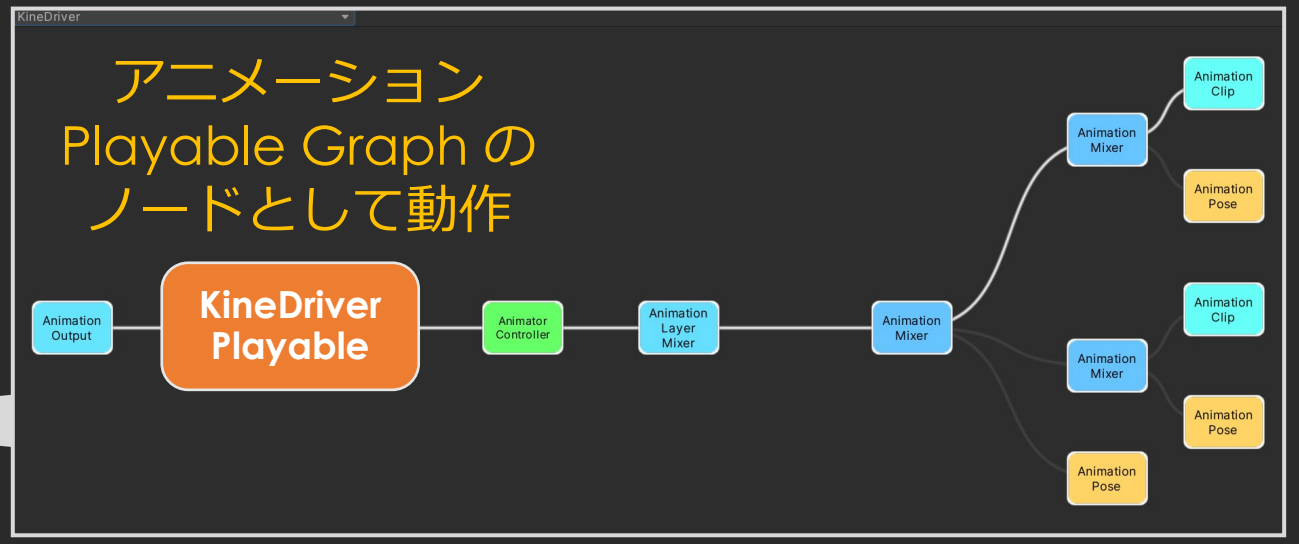
Playable コンポーネント

コンポーネントの概要

- MonoBehaviour 継承 class はシンプルに
  - 別 class に実装された処理を呼び出すだけ
  - ニーズに応じてコンポーネントをカスタマイズしやすく
- Animation C# Jobs と Burst で実装
  - アンマネージドメモリを使用
- Playable API による Playable ノードとして動作
  - アニメーションストリームに組み込む
  - Optimize Transform Hierarchy が可能
  - 他のコンポーネントも Playable なら組み合わせられる
- エディタでは、Animation や Timeline のプレビューに限り動作
  - Interface IAnimationWindowPreview を実装

MT や Job との大きな違い





どのようなグラフを組むかは利用者の自由だが、
コンポーネントでは標準的な実装を提供

Playable API とは

「プレイ可能なもの」のデータフローを作る API

- アニメーション、オーディオ、スクリプト
- 実行時に PlayableGraph を作ることで表現
- Animation System や Timeline もこれで作られている

アニメーション

オーディオ

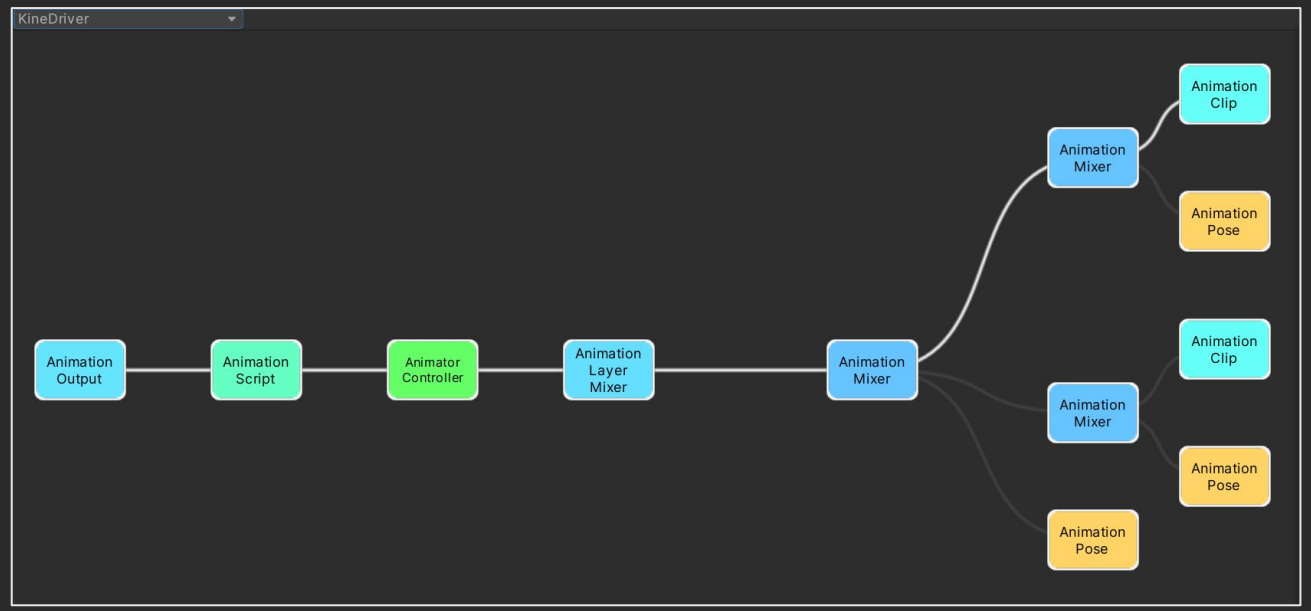
スクリプト

PlayableGraph

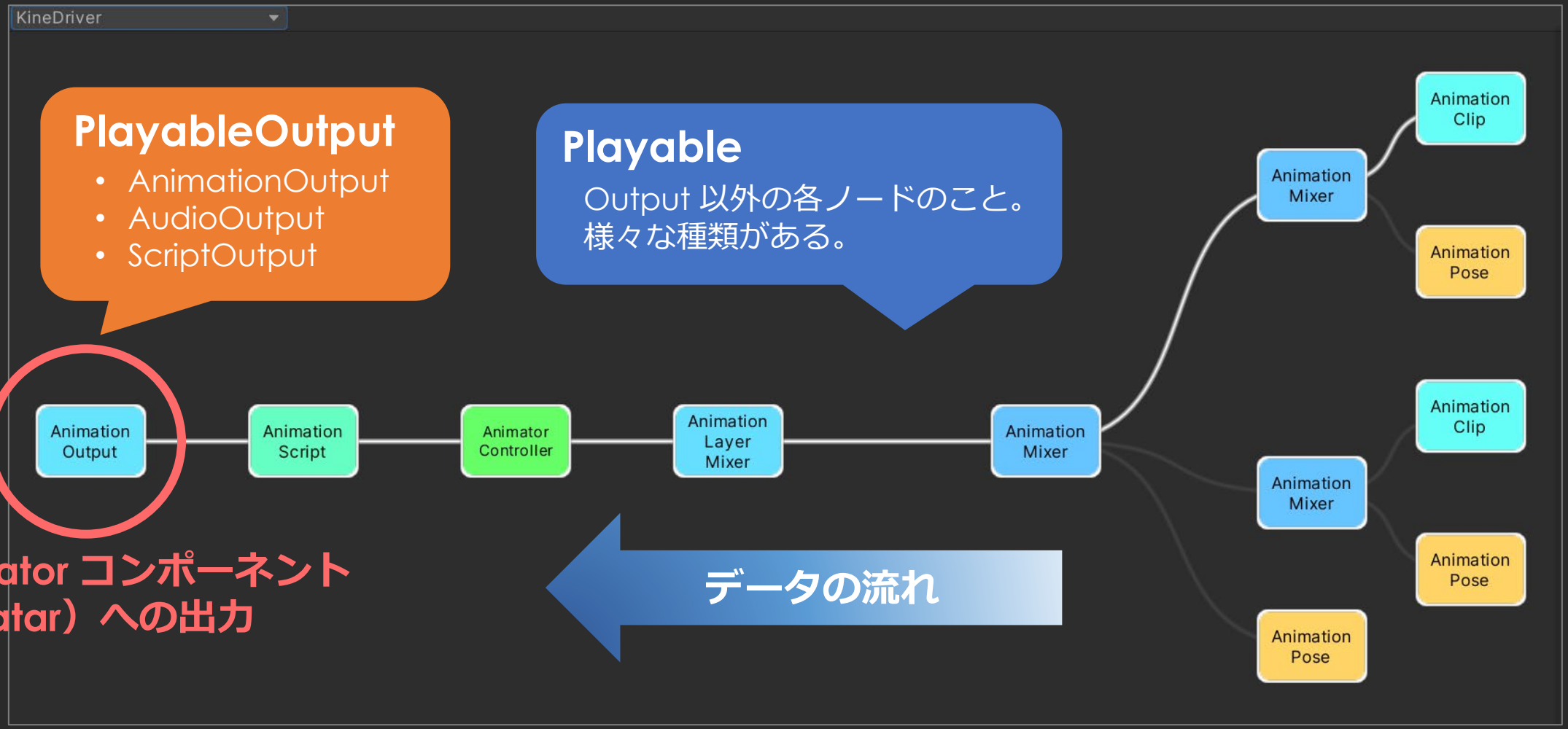
- いわゆる「ノードベースのエディター」ではない
- 実行時にプログラムで作るものなので、目に見えない
- 公式の github から可視化ツールが提供されている

https://github.com/Unity-Technologies/graph-visualizer.git

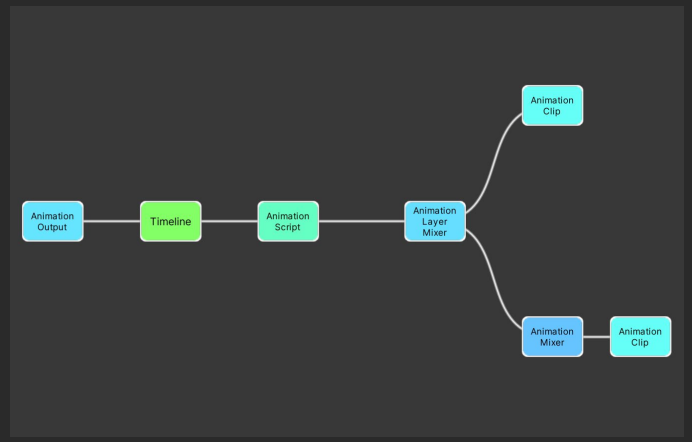
Package Manager に URL 入力



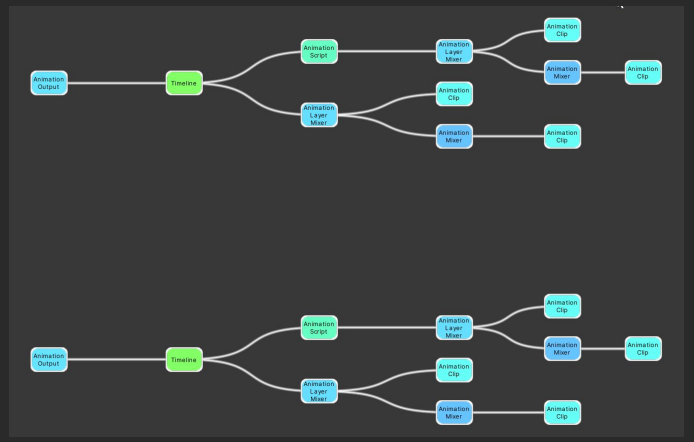
PlayableGraph の構造



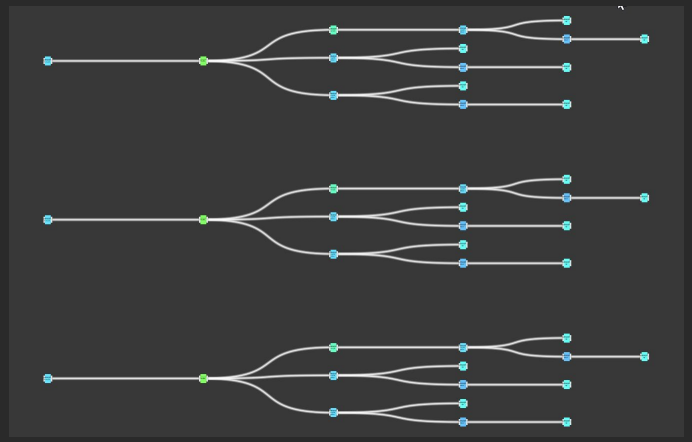
Tips: Timeline のグラフ可視化の問題



トラックが 1 個の例



トラックが 2 個の例

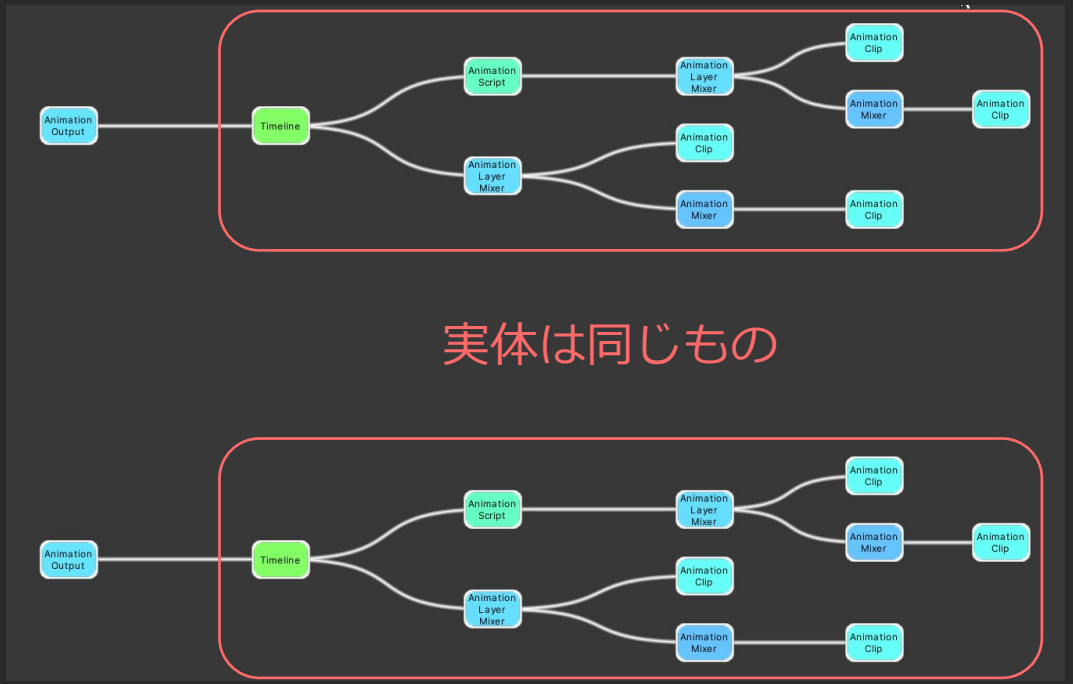


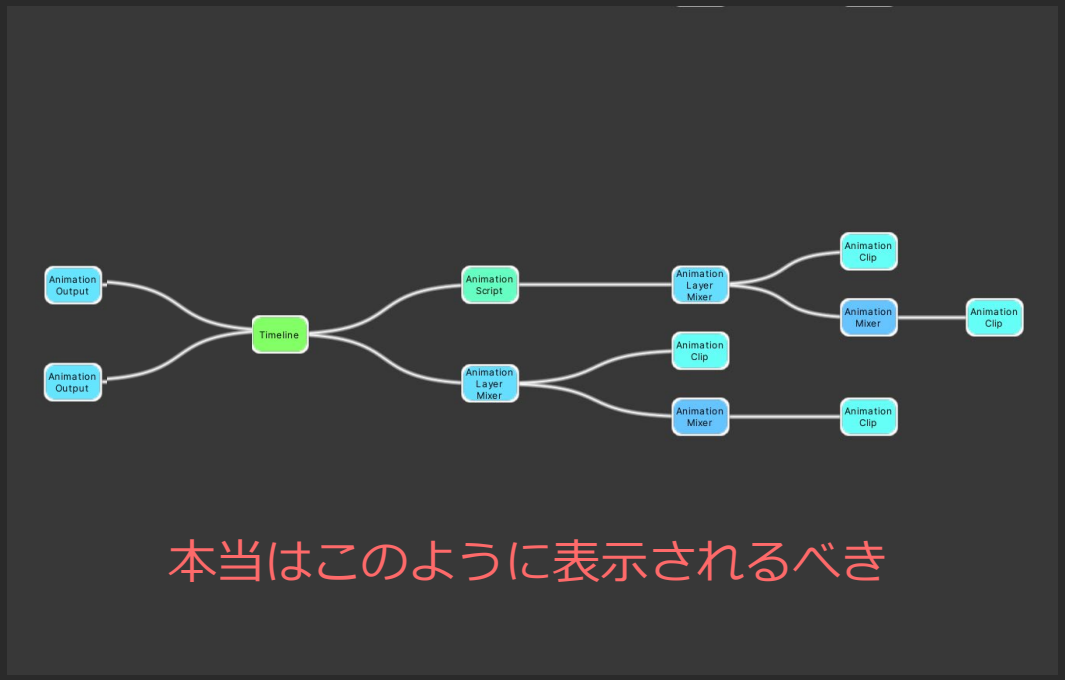
トラックが 3 個の例

AnimationOutput はトラックの数だけ存在するが、
その上流のツリーが重複して表示されてしまう。

Tips: Timeline のグラフ可視化の問題

たとえば、トラックが２個の場合





Animation C# Jobs

- アニメーション処理の Playable を C# Job System で実装できる仕組み
  - Animation Rigging はこの仕組みで実装されている
- Interface IAnimationJob を実装する
  - AnimationScriptPlayable というノードになる
  - １ノード＝１ジョブ
  - Animator (Avatar) の AnimationStream を処理する
- Transform には TransformStreamHandle でアクセス
  （他のタイプのハンドルもあるが割愛）

KineDriverAnimJob 実装概略

```
[BurstCompile(DisableSafetyChecks=true, FloatMode=FloatMode.Fast)]
public unsafe struct KineDriverAnimJob : IAnimationJob
{
    [ReadOnly][NativeDisableUnsafePtrRestriction, NoAlias] public TransformStreamHandle* transStrHandles;
    [ReadOnly][NativeDisableUnsafePtrRestriction, NoAlias] public int* inputTransNodeIdxs;
    [ReadOnly][NativeDisableUnsafePtrRestriction, NoAlias] public int* outputTransNodeIdxs;
    [ReadOnly] public int nInputTransNodes;
    [ReadOnly] public int nOutputTransNodes;

    その他のフィールド

    void IAnimationJob.ProcessRootMotion(AnimationStream stream) {}

    [SkipLocalsInit]
    void IAnimationJob.ProcessAnimation(AnimationStream stream)
    {
        for (var i=0; i<nInputTransNodes; ++i) {
            var handle = transStrHandles + inputTransNodeIdxs[i];
            TransformSreamHandle を使ってストリームから値をゲット
        }

        KineDriver の計算

        for (var i=0; i<nOutputTransNodes; ++i) {
            var handle = transStrHandles + outputTransNodeIdxs[i];
            TransformSreamHandle を使ってストリームに値をセット
        }
    }
}
```

← 入力

← 計算

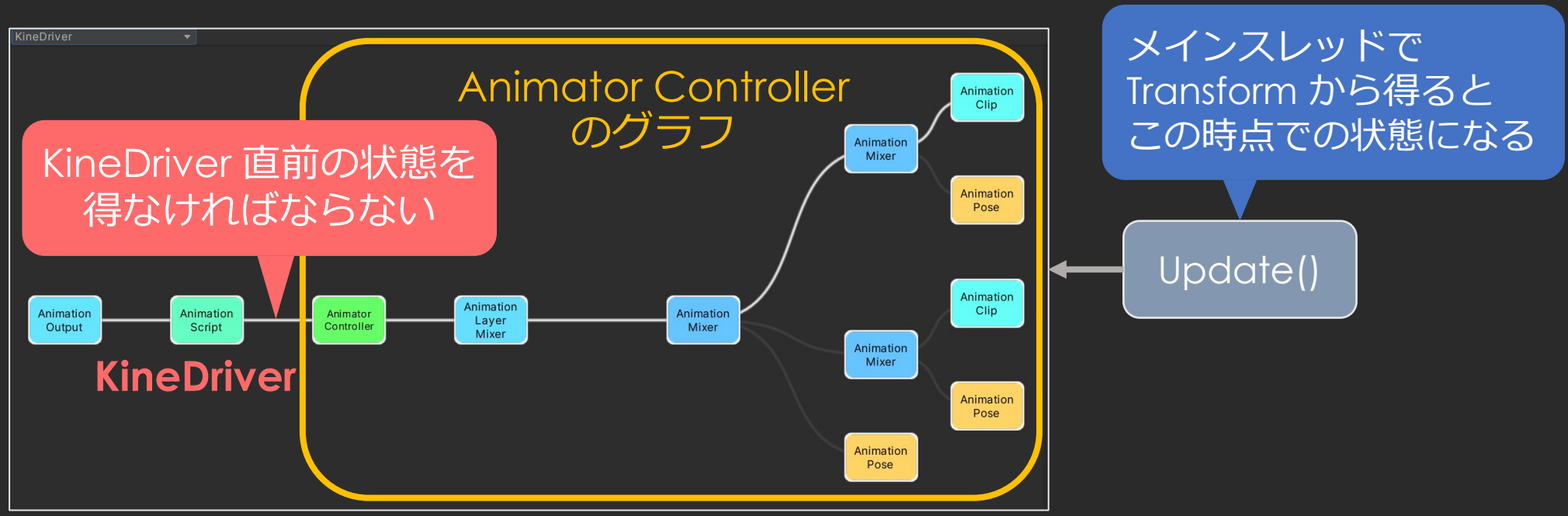
← 出力

TransformStreamHandle

	Property	Transform	TransformAccess	TransformStreamHandle
Local	localPosition	Read Write	Read Write	Read Write
	localRotation	Read Write	Read Write	Read Write
	localScale	Read Write	Read Write	Read Write
Global	position	Read Write	Read Write	Read Write
	rotation	Read Write	Read Write	Read Write
	lossyScale	Read		
Matrix	localToWorldMatrix	Read	Read	Read
	worldToLocalMatrix	Read	Read	
	localToParentMatrix			Read

アクセスはプロパティではなく GetXXX() と SetXXX() のようなメソッドになるが、TransformAccess と同じく Global Scale を得られない。

Global Scale をどうするか



（Animator Controller の後に KineDriver を追加した場合の例）

Jobコンポーネント開発時に、Transform管理の仕組みを作ったので、
その中で Global Scale にも対応できた。

※ Jobコンポーネントでもそうできたが、速度はさほど変わらなかった。

コンポーネント実装概略

※ IAnimationWindowPreview の実装は省略

```
[HelpURL("http://example")]
[Icon( "Assets/Gizmos/SQEX/KineDriver/KineDriverData Icon.png")]
[AddComponentMenu("KineDriver/KineDriver Playable")]
[RequireComponent(typeof(Animator))]
[DisallowMultipleComponent]
public class KineDriverPlayable : MonoBehaviour
{
    public KineDriverData[] KDIData;  // 複数KDIに対応

    KineDriverPlayableBinder _binder;
    PlayableGraph _graph;

    public KineDriverPlayable()
    {
        _binder = new KineDriverPlayableBinder();
    }

    public void Dispose()
    {
        _binder.Dispose();
    }

    void OnDestroy() => Dispose();

    void Awake()
    {
        if (KDIData == null)
            KDIData = new KineDriverData[0];
    }
```

実装本体

後始末が必要

毎フレームの LateUpdate() は無い

```
    void OnEnable()
    {
        if (! _binder.IsActive) {
            _binder.Bind(transform, KDIData);
            if (! _binder.IsActive)
                return;

            AnimatorUtility.OptimizeTransformHierarchy(gameObject, null);
        }

        _graph = PlayableGraph.Create(transform.name + ".KineDriver");
        var output = AnimationPlayableOutput.Create(
            _graph, "ouput", _binder.animator);

        output.SetAnimationStreamSource(
            AnimationStreamSource.PreviousInputs);  // Experimental
        output.SetSortingOrder(2000);  // Experimental

        output.SetSourcePlayable(_binder.CreatePlayable(_graph));
        _graph.Play();
    }

    void OnDisable()
    {
        if (_graph.IsValid())
            _graph.Destroy();
    }
}
```

初期化

Transform最適化

グラフを作って再生

KineDriverPlayableBinder 概観

Bind()

- 実行用メモリ確保とデータロード
- Animator (Avatar)と KineDriver データのバインド

```
TransformStreamHandle animator.BindStreamTransform(transform);
  ...
```

Dispose()

- アンマネージドリソースの解放

CreatePlayable()

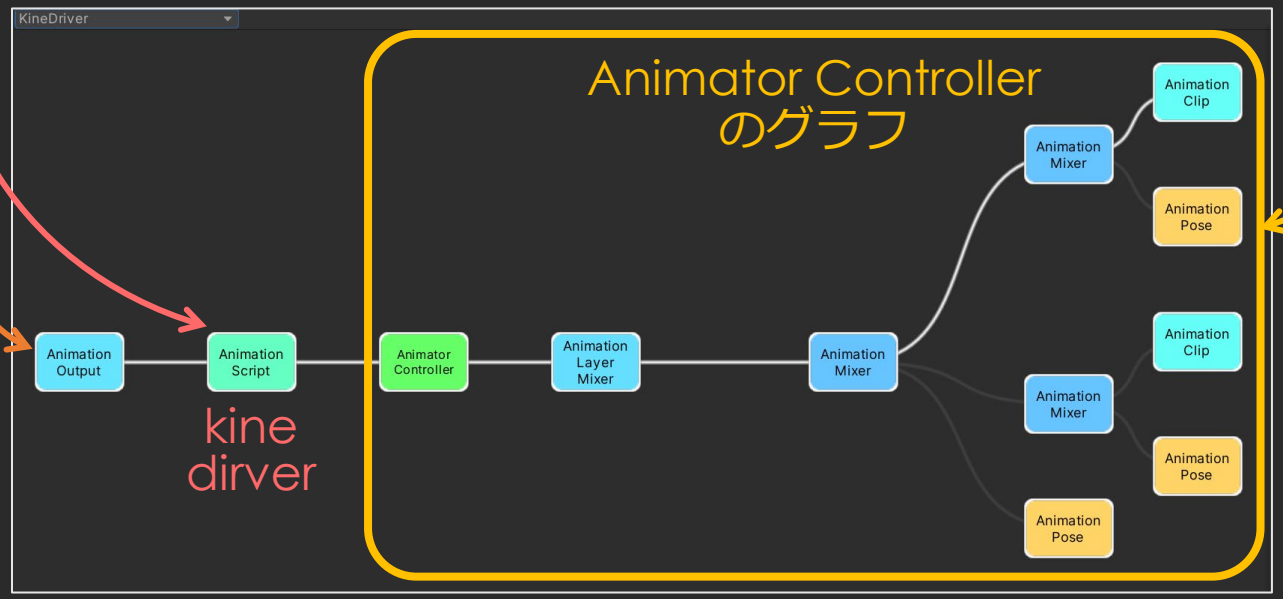
- AnimationScriptPlayable ノードの生成

```
AnimationScriptPlayable.Create(graph, job, inputCount);
```

グラフ作成処理

Animator Controller を後処理する場合

```
_graph = PlayableGraph.Create(transform.name + ".KineDriver");

var anim = AnimatorControllerPlayable.Create(_graph, this.controller);

var kinedriver = _binder.CreatePlayable(_graph, 1);
kinedriver.ConnectInput(0, anim, 0, 1f);

var output = AnimationPlayableOutput.Create(_graph, "ouput", _binder.animator);
output.SetSourcePlayable(kinedriver);
```



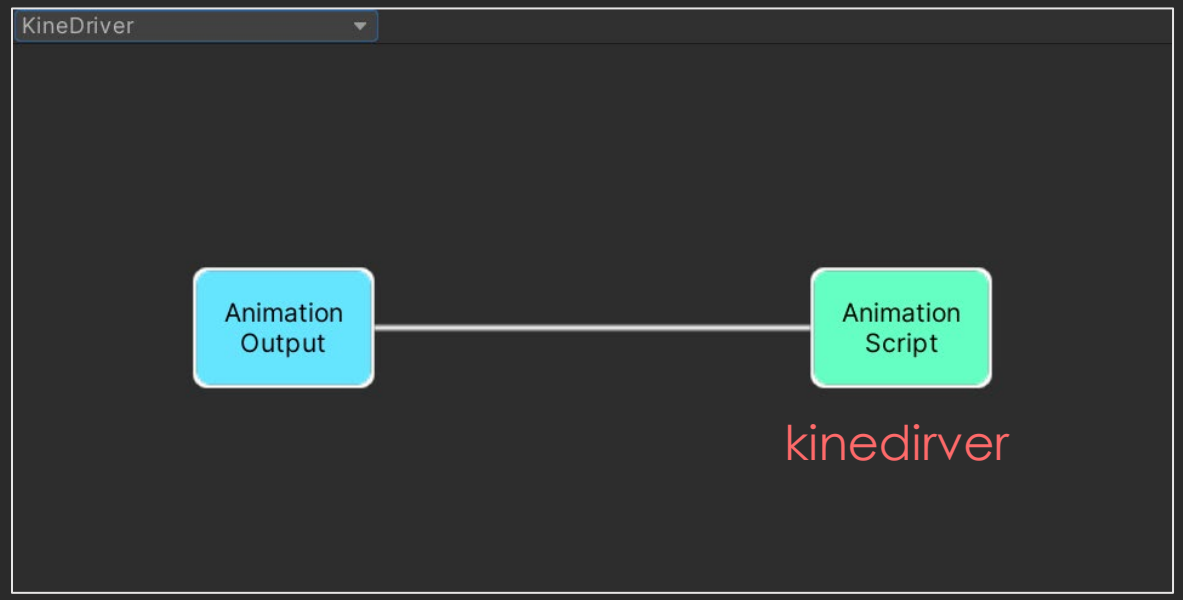
このように、
目的に合わせてグラフを組むのが基本。

そのため、どのような場合にも使える
汎用機能にはならない。

グラフ作成処理

汎用コンポーネント KineDriverPlayable の場合

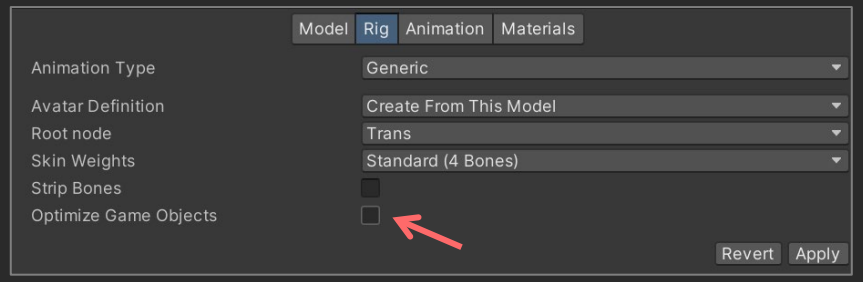
```
_graph = PlayableGraph.Create(transform.name + ".KineDriver");

var kinedriver = _binder.CreatePlayable(_graph);

var output = AnimationPlayableOutput.Create(_graph, "ouput", _binder.animator);
output.SetAnimationStreamSource(AnimationStreamSource.PreviousInputs);  // Experimental
output.SetSortingOrder(2000);  // Experimental
output.SetSourcePlayable(kinedriver);
```



他のグラフを後処理するグラフを追加。

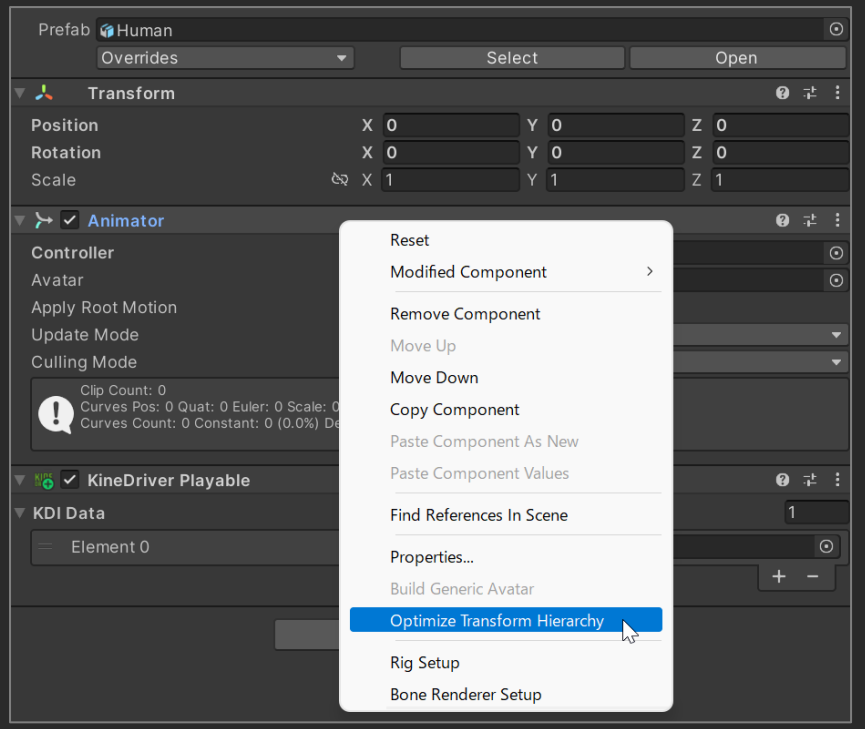
Animation Rigging で採用されている方法だが Experimental なメソッドを使用。
（2019.4 LTS からこれで問題ない）

Optimize Transform Hierarchy



FBX Import オプション

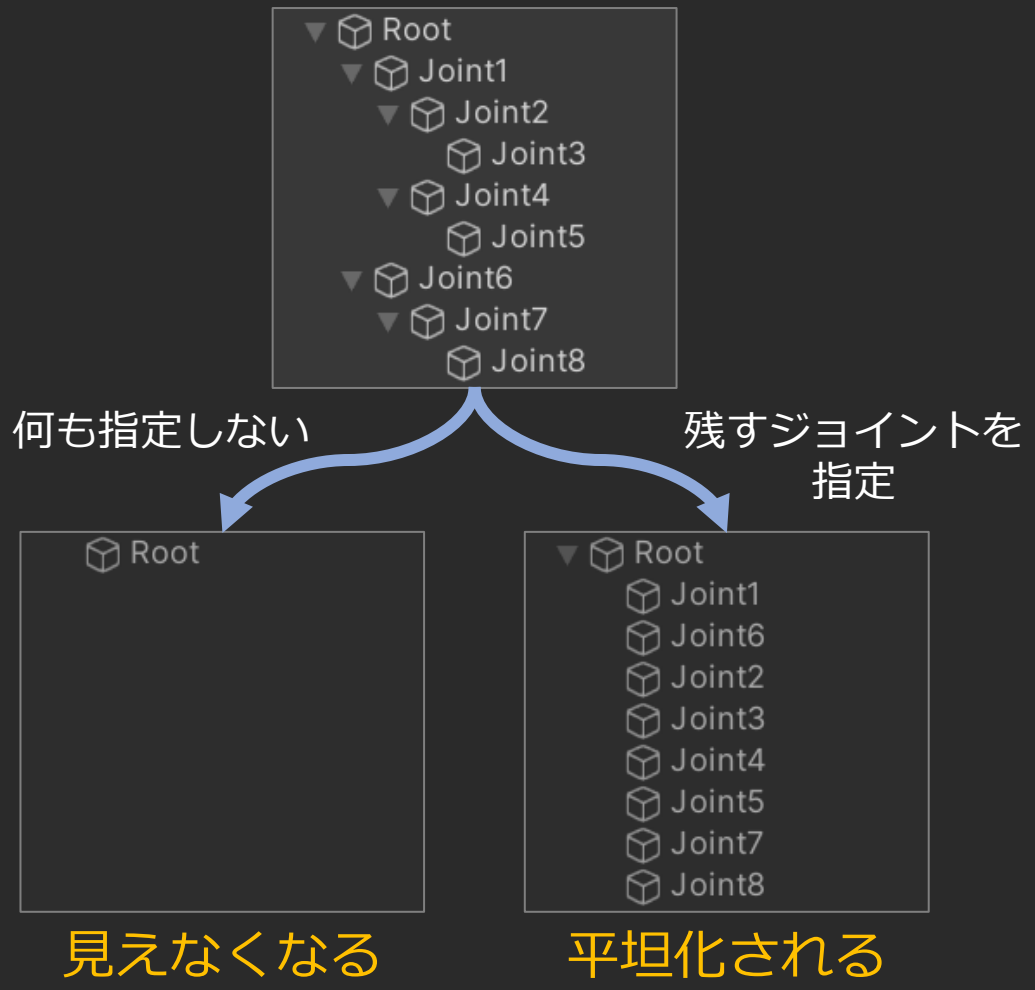
または



Animator コンポーネントの右クリックメニュー
（Prefab はアンパックする必要がある）

アニメーション処理が最適化され、フレームレートが結構上がる

何が起きるか？



（コンポーネントが付いたものは、指定しなくても残る）

- 内部的に最適化されたスケルトンを持っているようだが、Transform としてはアクセスできなくなる
- ジョイント名指定して消えないようにすることはできるが、平坦化されローカル座標系で処理できなくなる

補助骨処理で困る

AnimationStream では問題ない

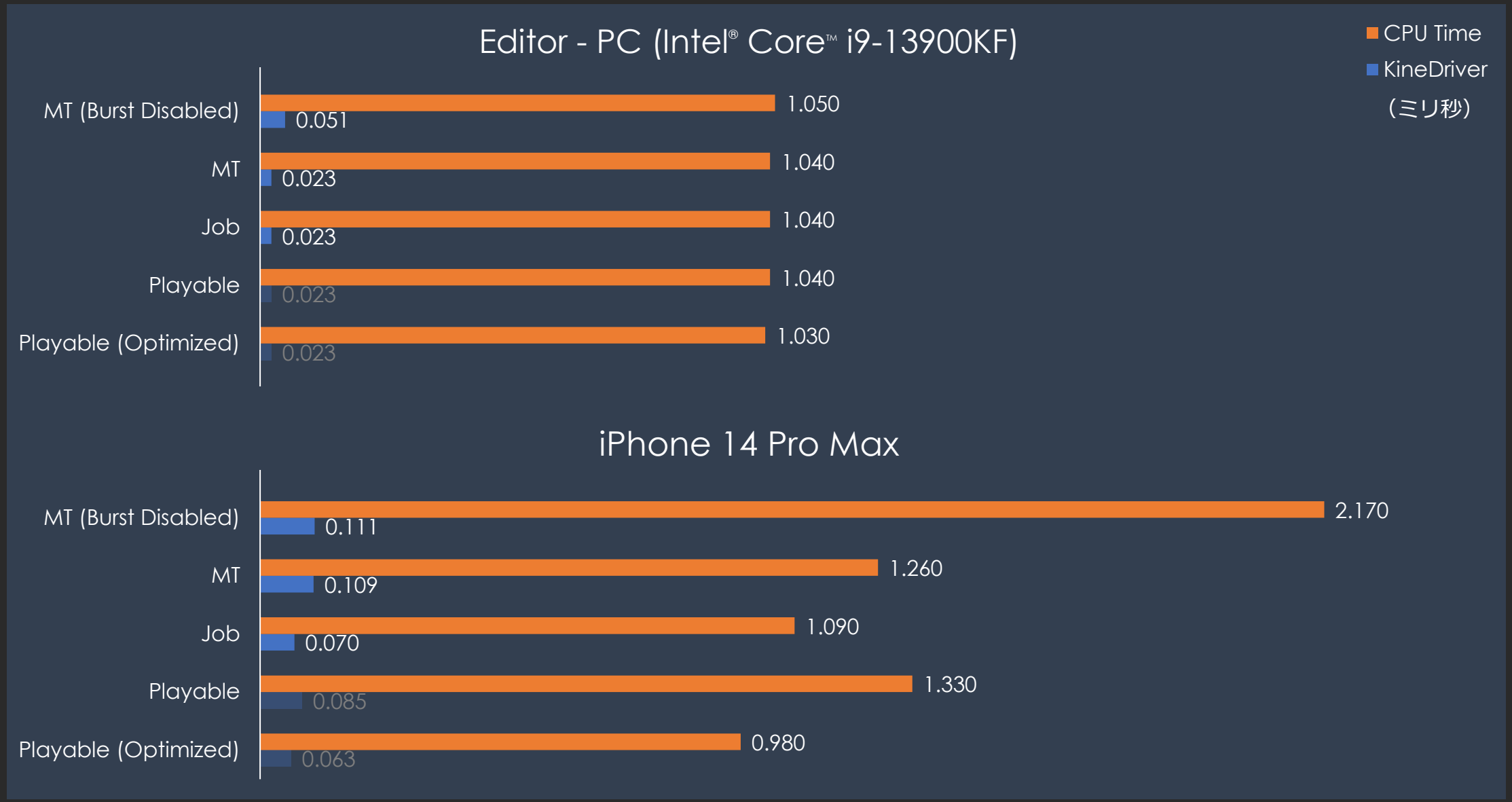
- 最適化されていても Playable では普通に処理できる。
- TransformStreamHandle のバインドのために Transform が必要だが、その後に最適化できる。
  - FBX Import 時ではなく、ランタイムの初期化処理で最適化する。
- プレビューでは最適化せず、プレイ時のみ最適化。
  エディター内プレイでも問題ない。

```
void OnEnable()
{
    if (! _binder.IsActive) {
        _binder.Bind(transform, KDIData);
        if (! _binder.IsActive)
            return;

        AnimatorUtility.OptimizeTransformHierarchy(gameObject, null);
    }

    ...
}
```

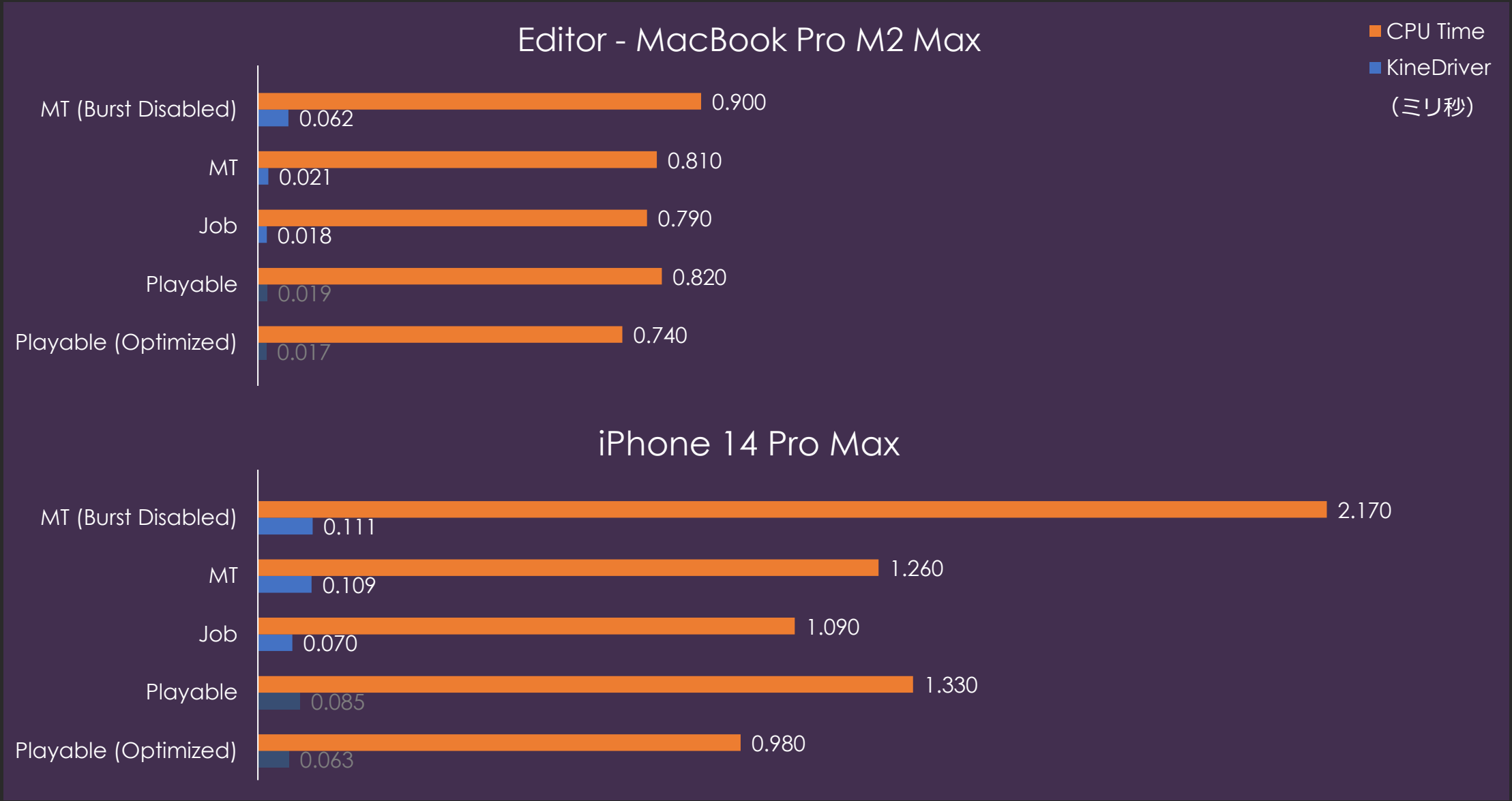
← バインド後は最適化が可能

負荷計測とまとめ

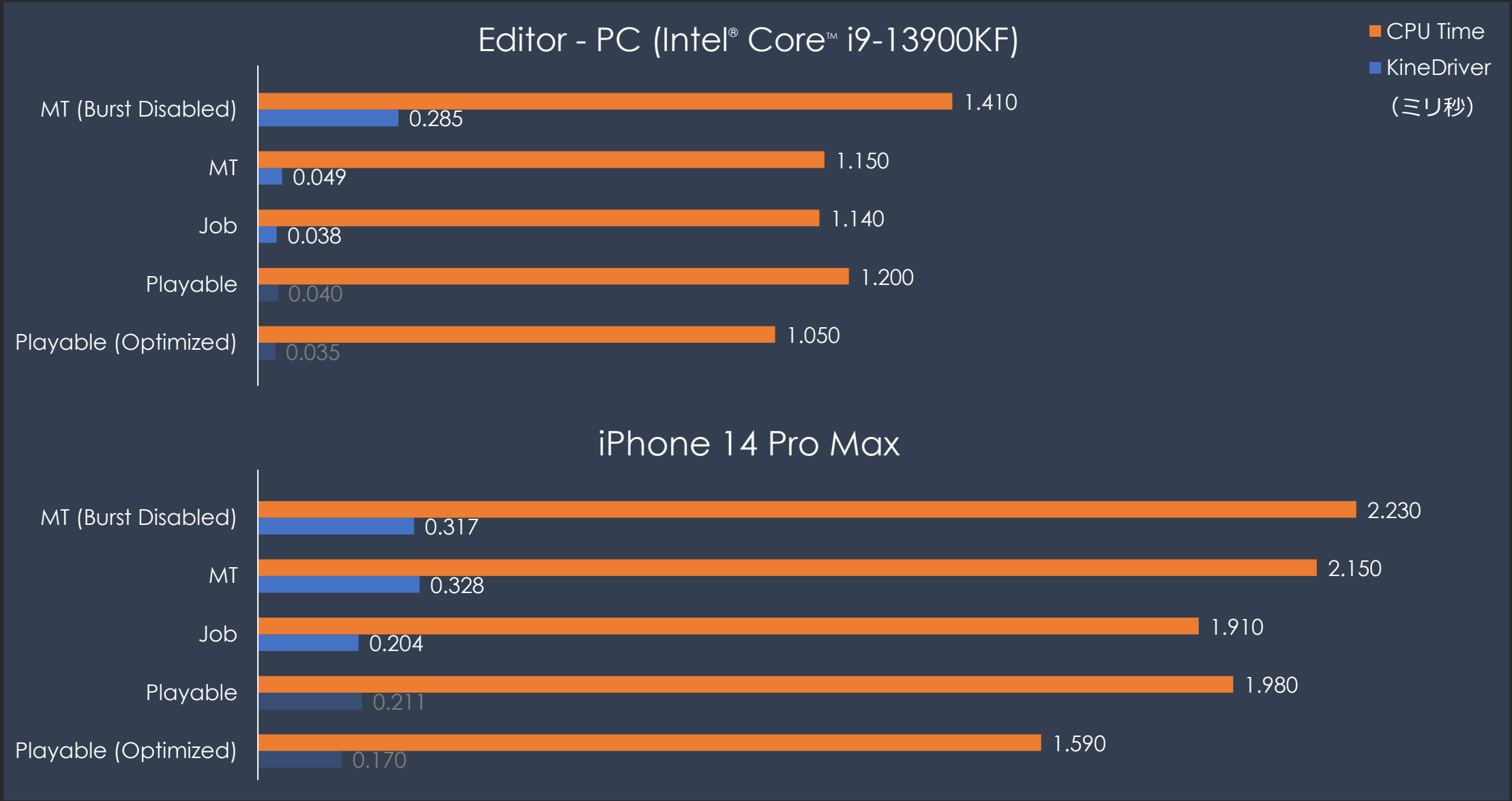
普通のデータ（総骨数 165、補助骨数 45）



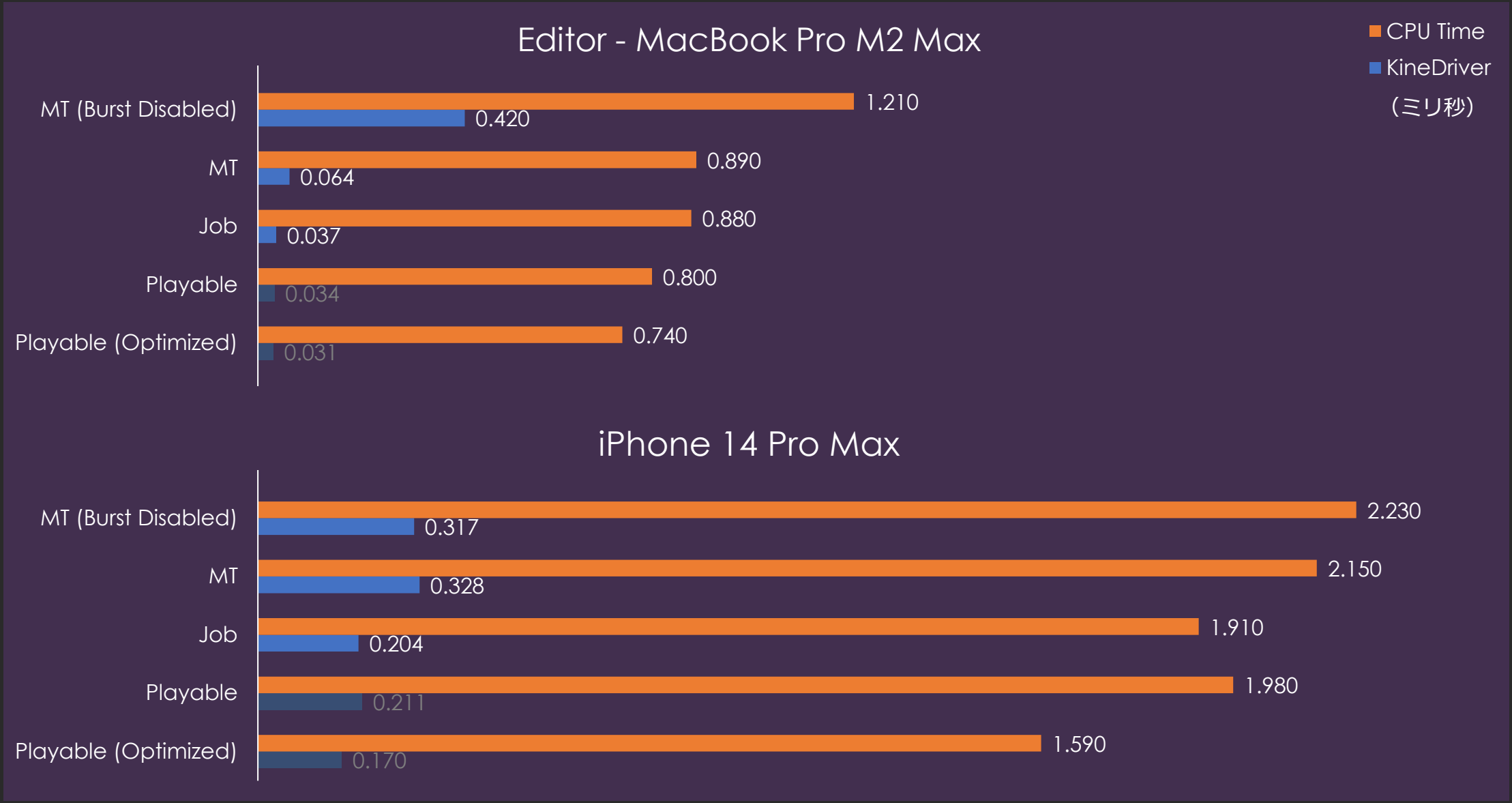
普通のデータ（総骨数 165、補助骨数 45）



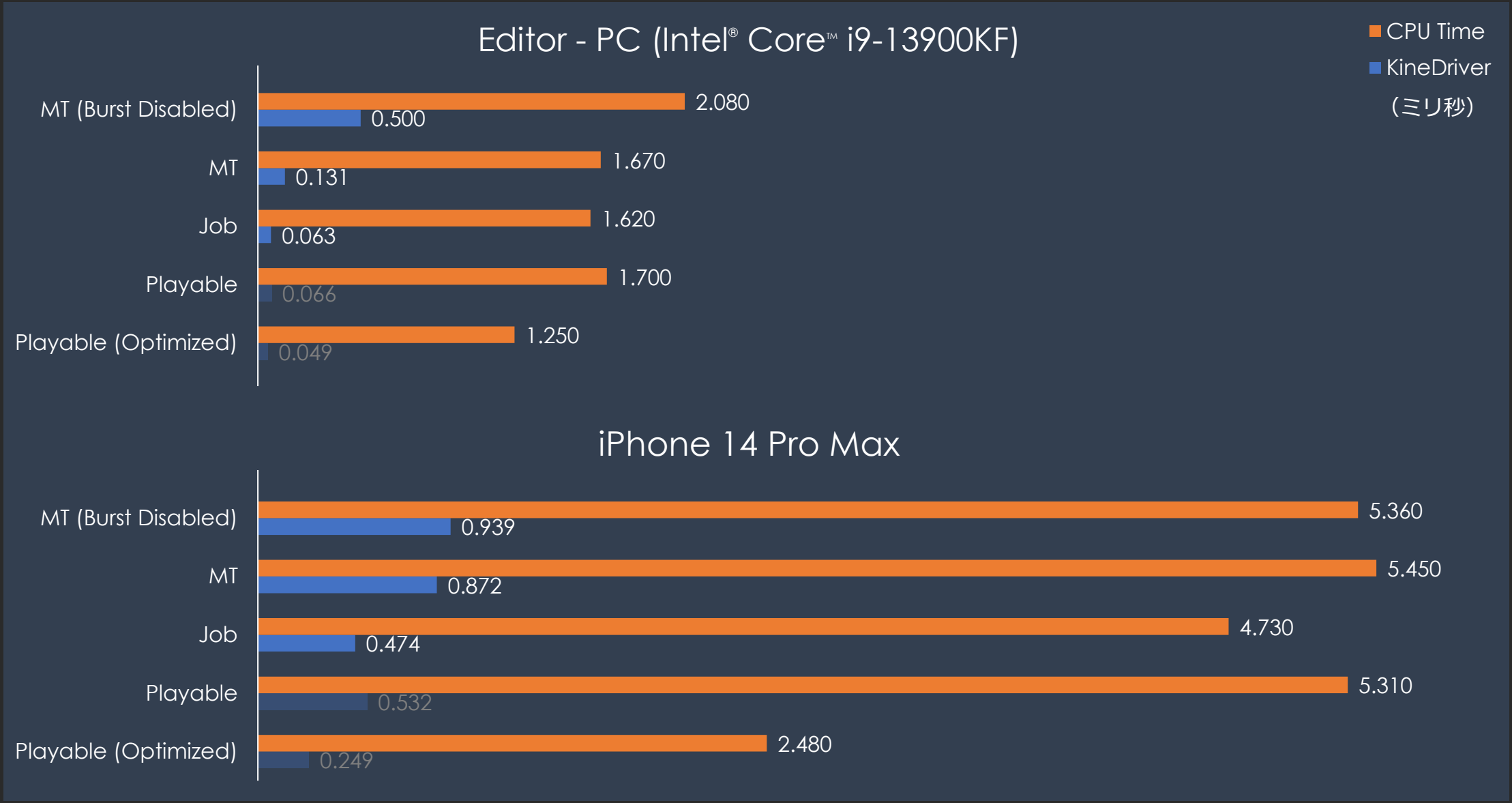
ちょっと違うデータ（RBF補間を多用）（総骨数 415、補助骨数 128）



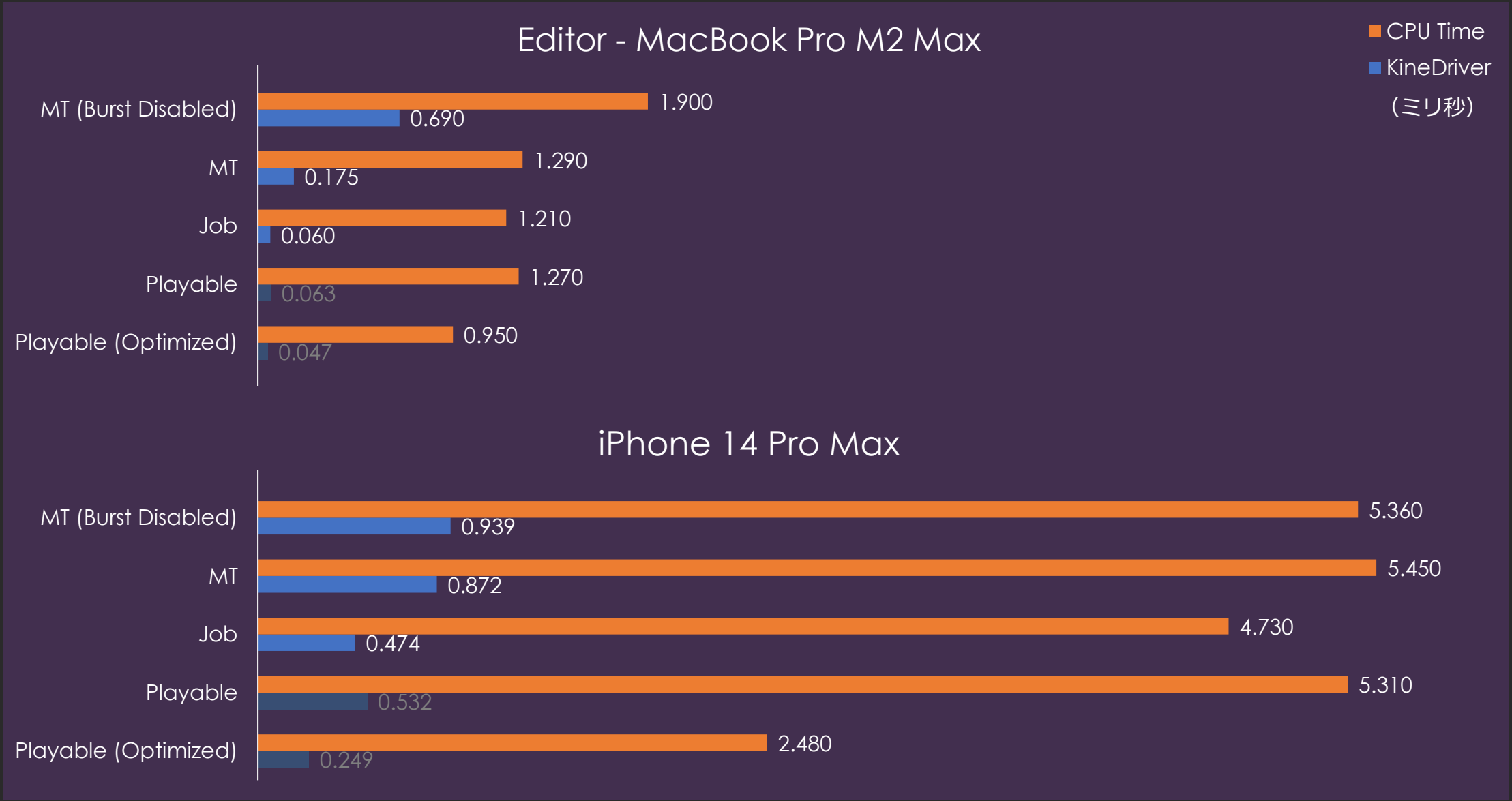
ちょっと違うデータ（RBF補間を多用）（総骨数 415、補助骨数 128）



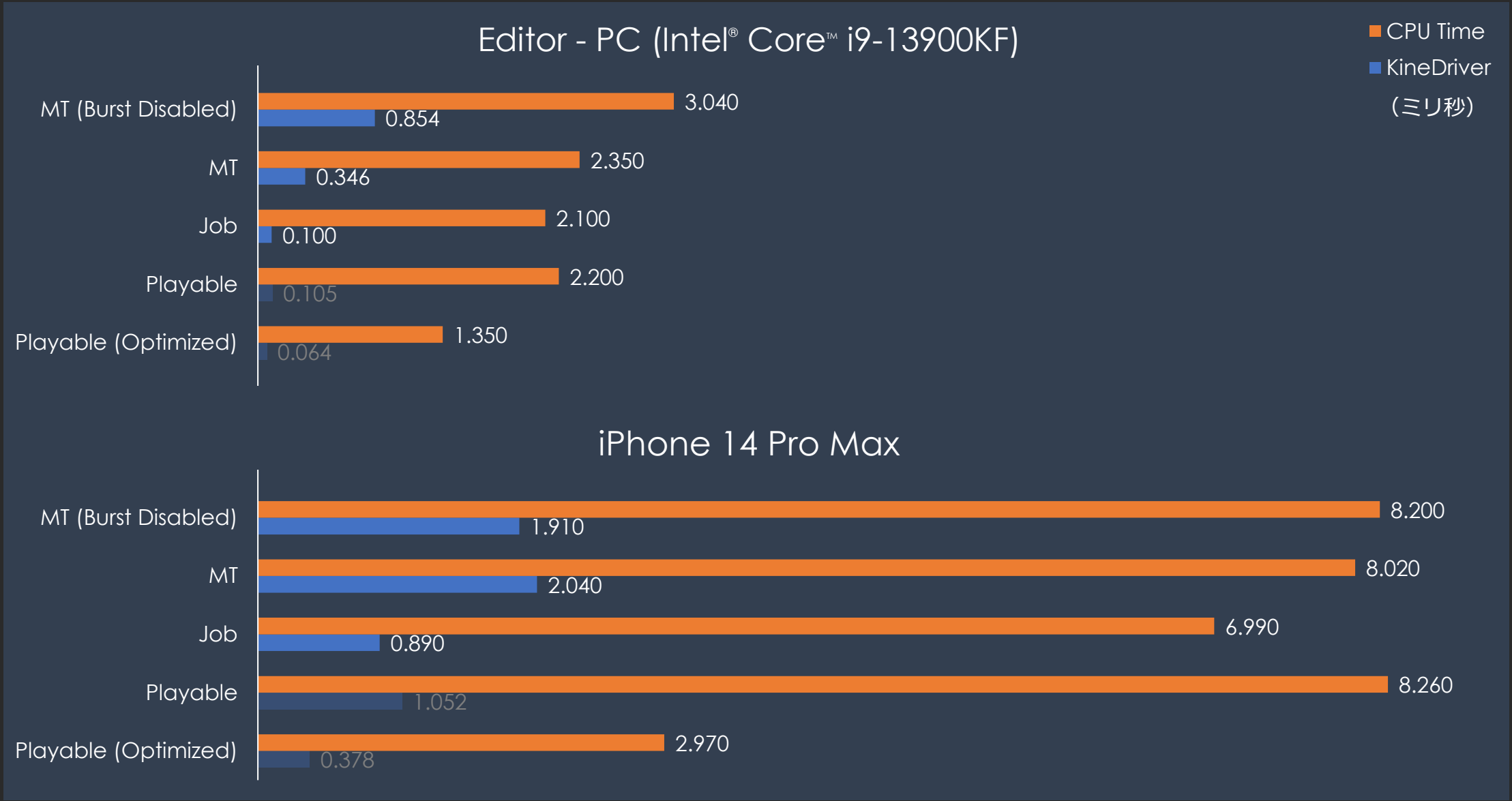
重いデータ（総骨数 2307、補助骨数 290）



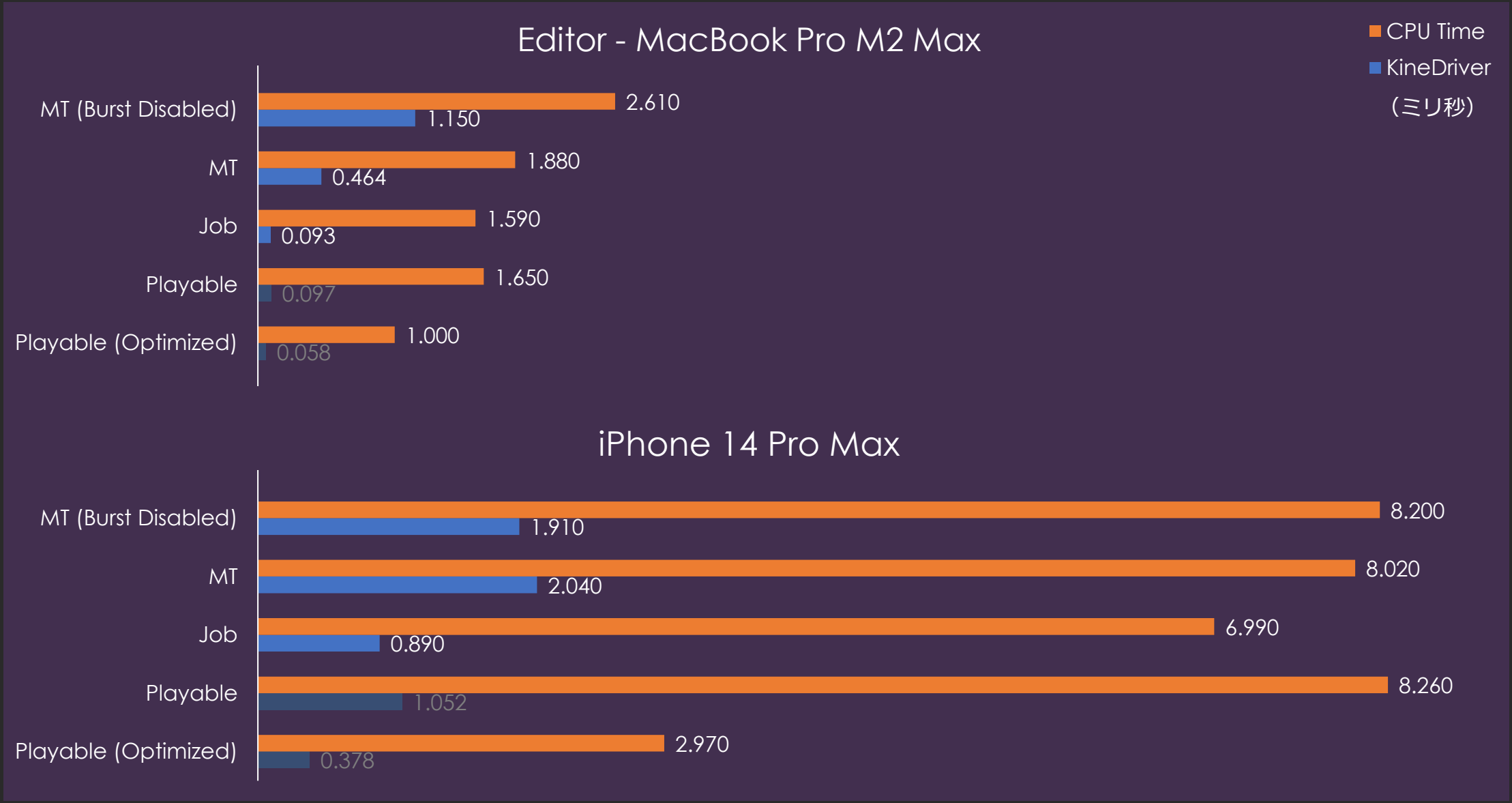
重いデータ（総骨数 2307、補助骨数 290）



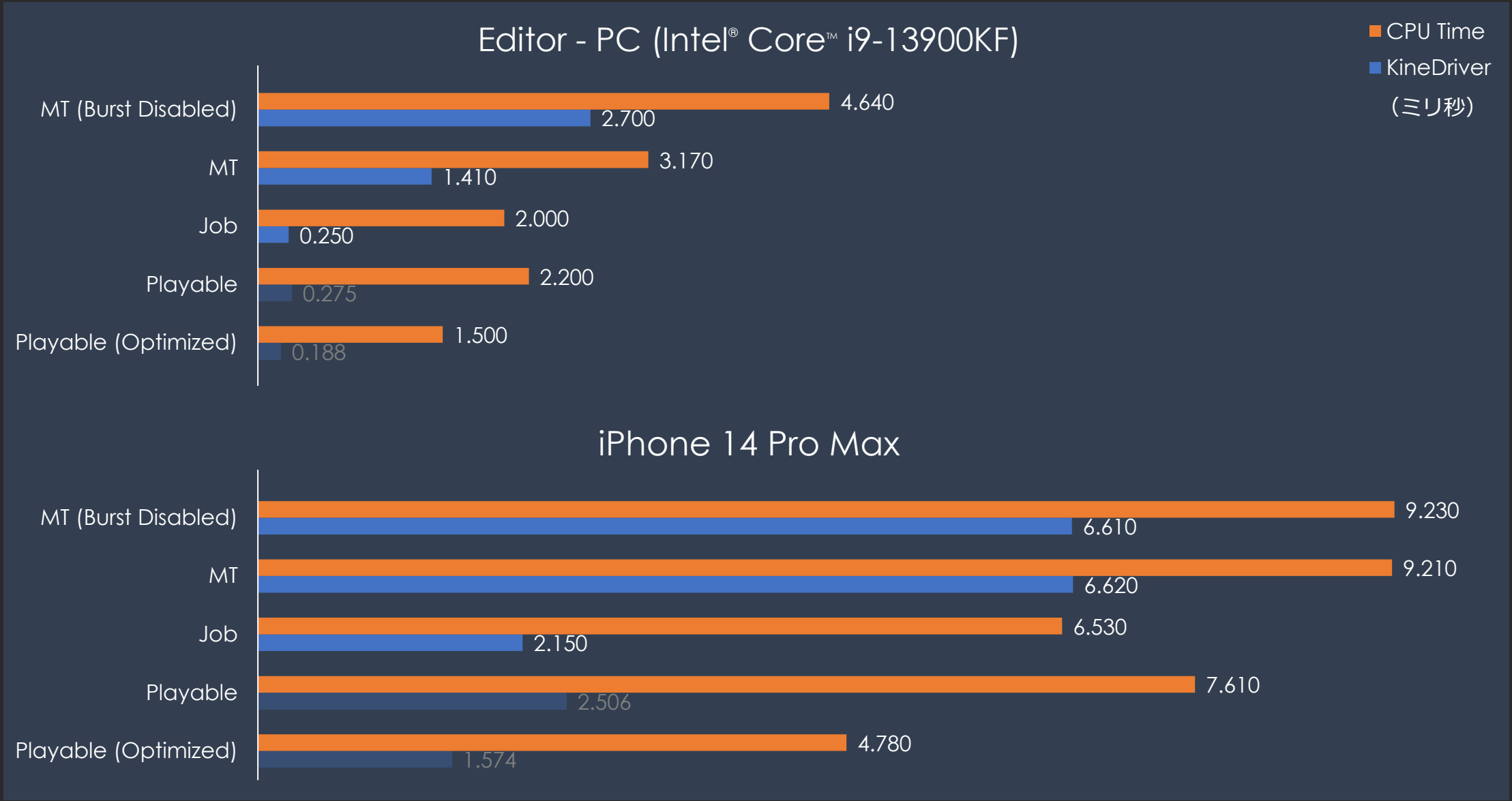
もっと重いデータ（総骨数 1466、補助骨数 372）



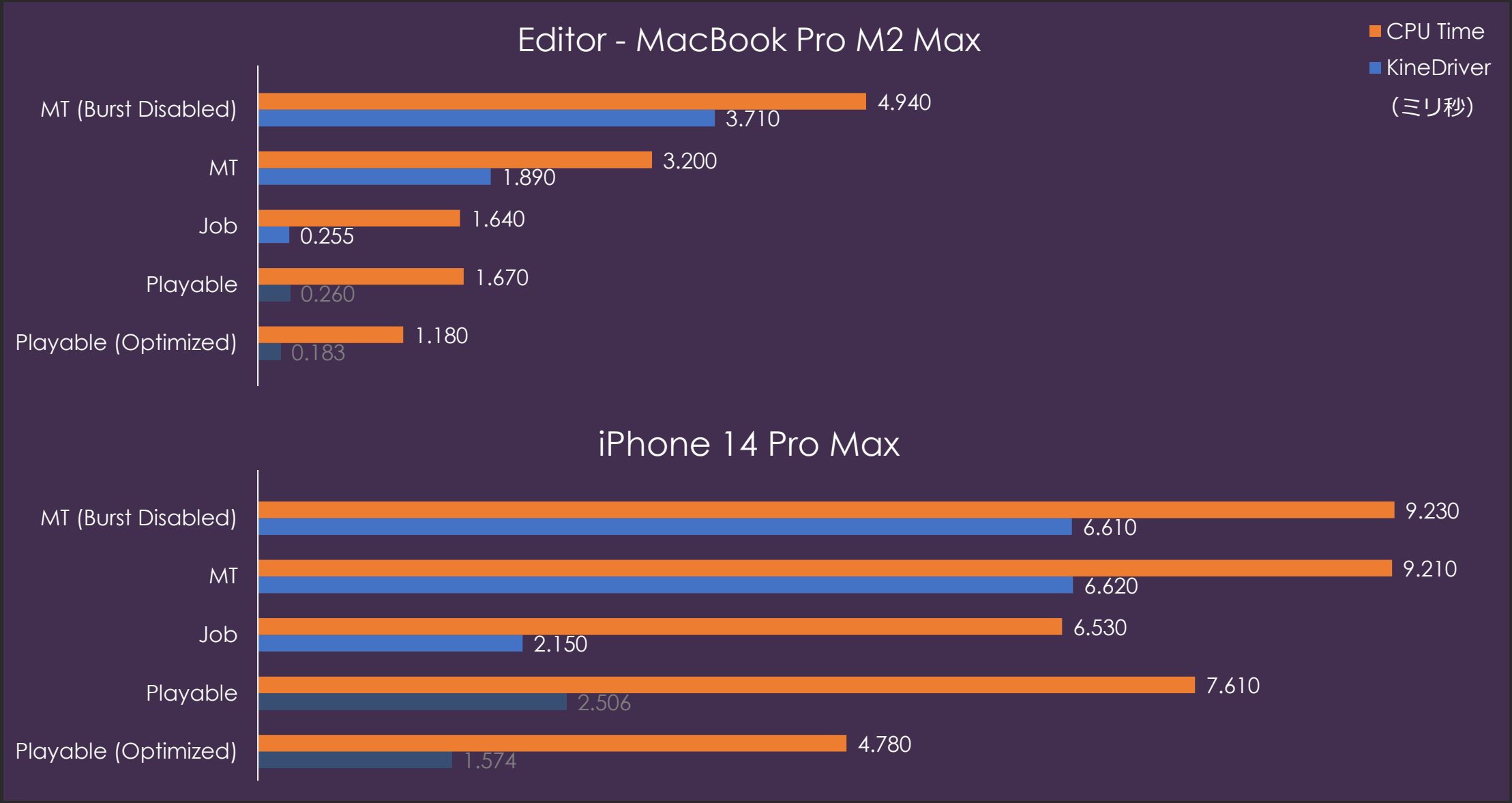
もっと重いデータ（総骨数 1466、補助骨数 372）



超絶重いデータ（総骨数 2319、補助骨数 1352）



超絶重いデータ（総骨数 2319、補助骨数 1352）



まとめ

- Burst は、関数単位ですぐに使えるので手軽で便利。
- Job にすると、さらに Burst が効率的に使える。
- Job のオーバーヘッドは結構あるので、Job数を少なく抑えるのが効果的だった。
- Transform アクセスを少なく抑えるのが効果的だった。
- ただの Job にするか Playable にするかは、利用状況で選択。
- Playable は、コンポーネント単体の負荷は Job と大差ないものの、Optimize Transform Hierarchy できる意味が大きい。
- 将来の公式 ECS-based Animation にも期待。

ご清聴ありがとうございました。

佐々木 隆典

ryusukes@square-enix.com