Maya®のしくみ

基礎から学ぶDGとパラレル評価、そしてキャッシュプレイバック

株式会社スクウェア・エニックス
テクノロジー推進部
リードテクニカルアーティスト

佐々木　隆典

SQUARE ENIX　ADVANCED TECHNOLOGY DIVISION

アジェンダ

- 背景
- Maya® の基本
- ディペンデンシーグラフ
- プラグインノード開発の基礎
- パラレル評価
- キャッシュプレイバック
- エバリュエータ
- まとめ

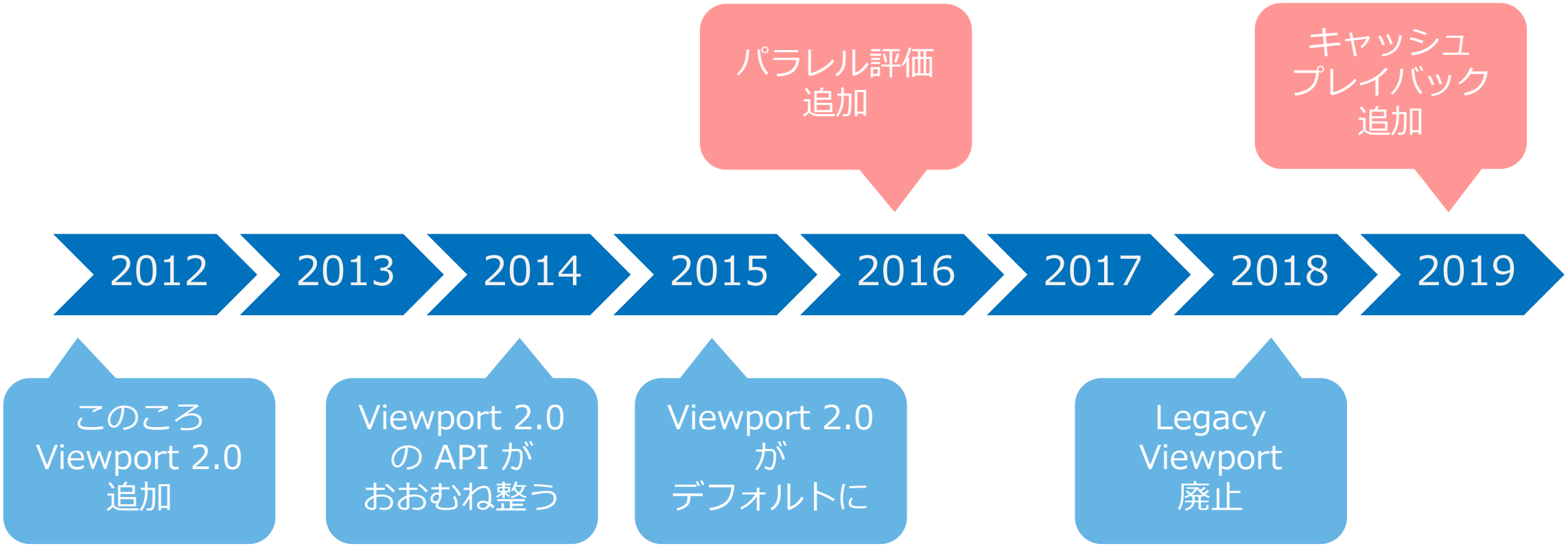
Break?

Break?

背景

Parallel と Viewport 2.0 の現状

Maya® バージョンの変遷



環境変数
MAYA_ENABLE_LEGACY_VIEWPORT=1
をセットすれば、まだ利用可能

困った印象

Viewport 2.0

- 不安定。
- プラグインが動かない。
- Legacy Viewport より遅い。

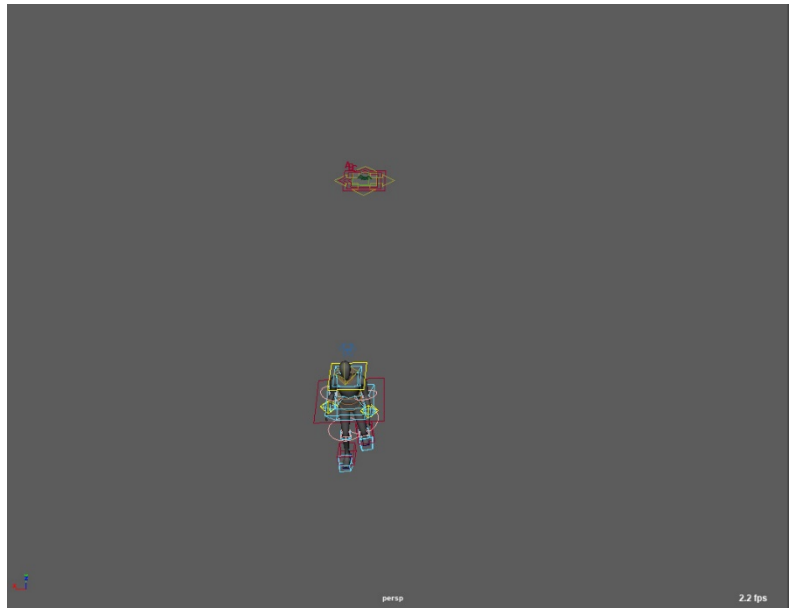
Parallel Evaluation

- 不安定。
- プラグインが動かない。
- DG より遅い。

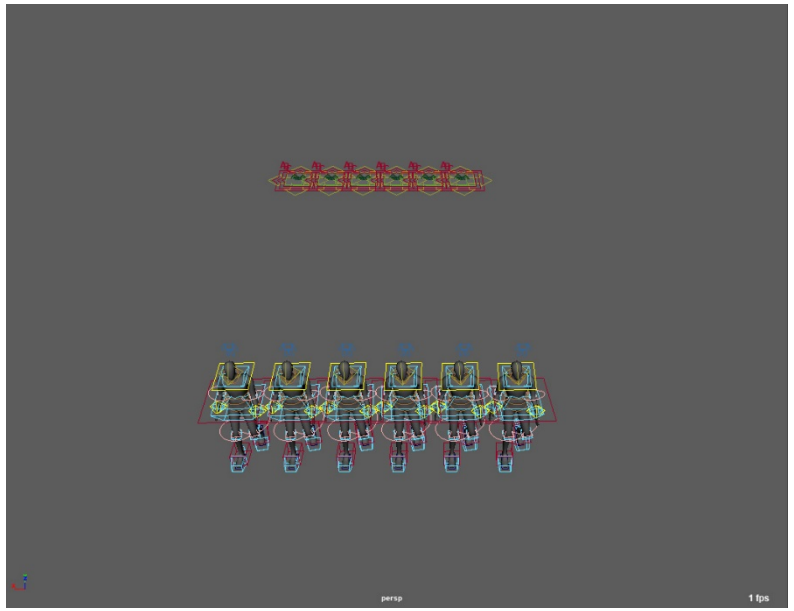
今でも、本当にそうか？

弊社のリグのフレームレート比較

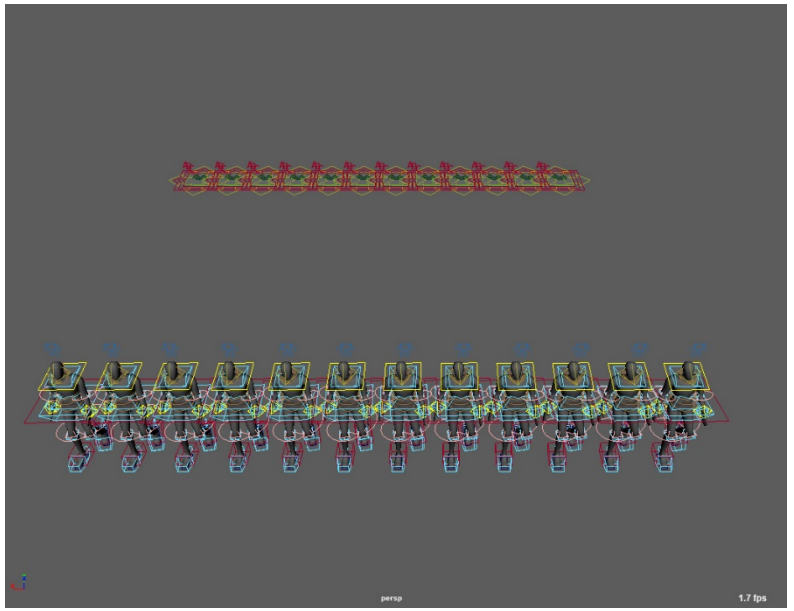
- Maya® のモーションサンプルデータ（3191ポリゴン／体）
- 自作プラグインノードを多数使用
- コントローラのロケータもプラグイン



1 体

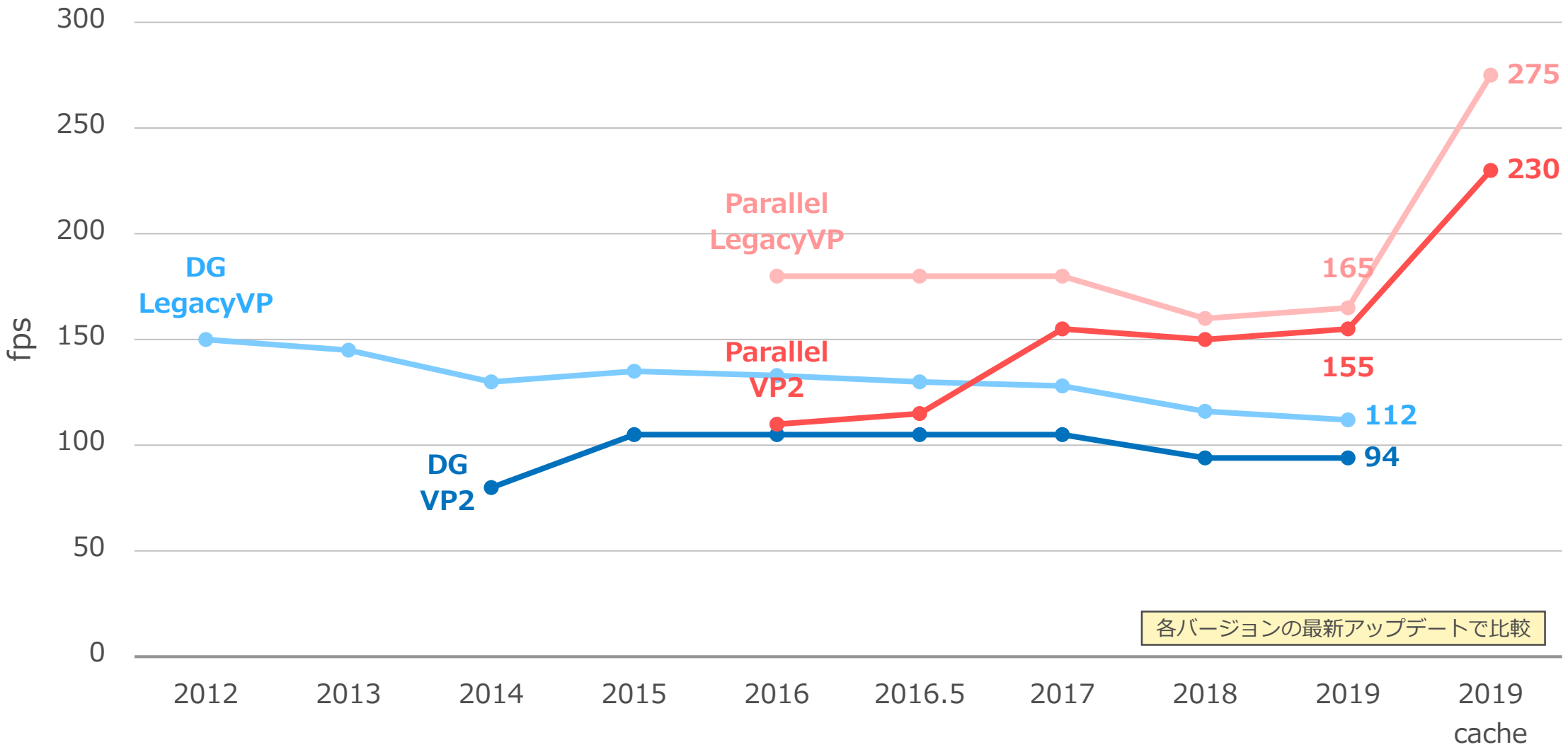


6 体

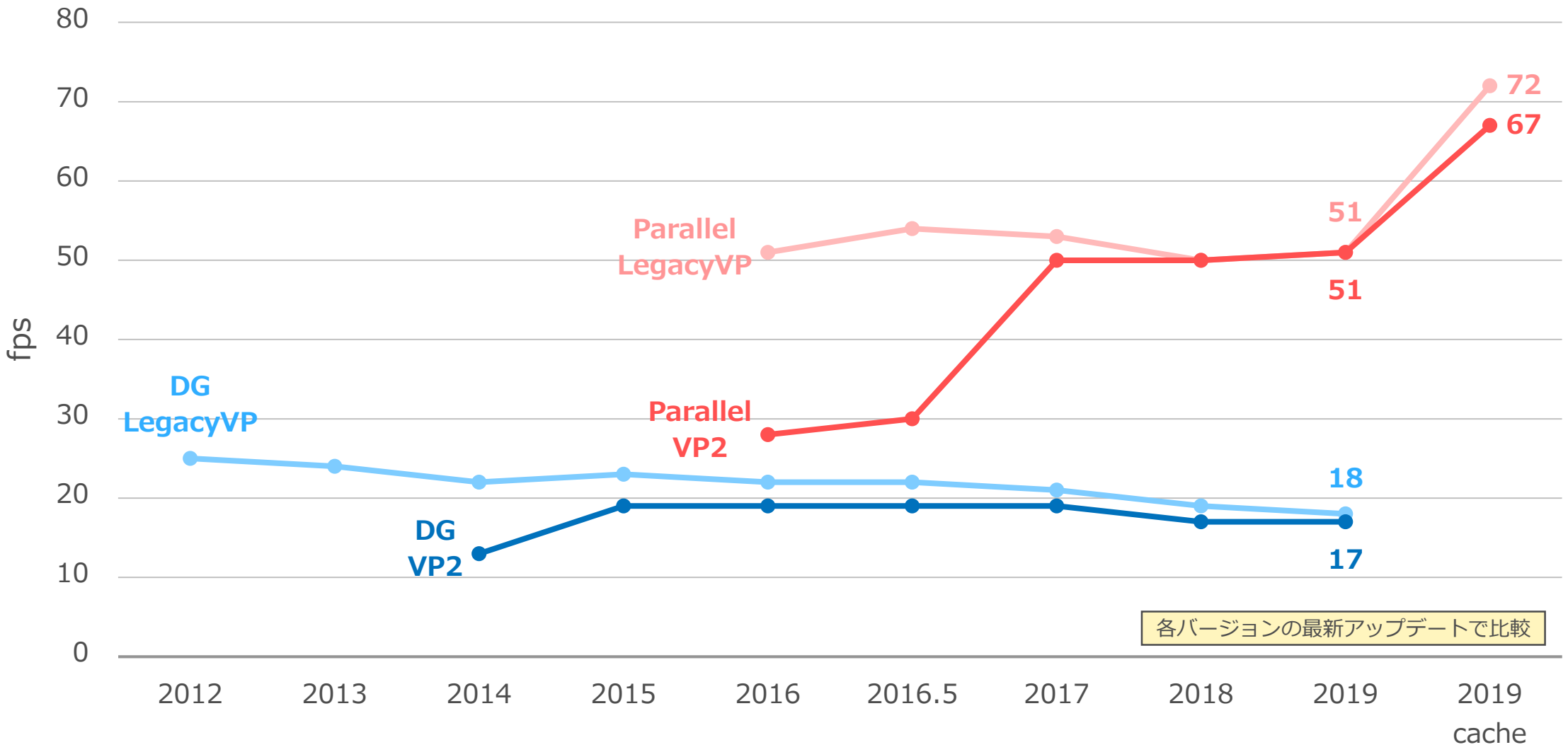


12 体

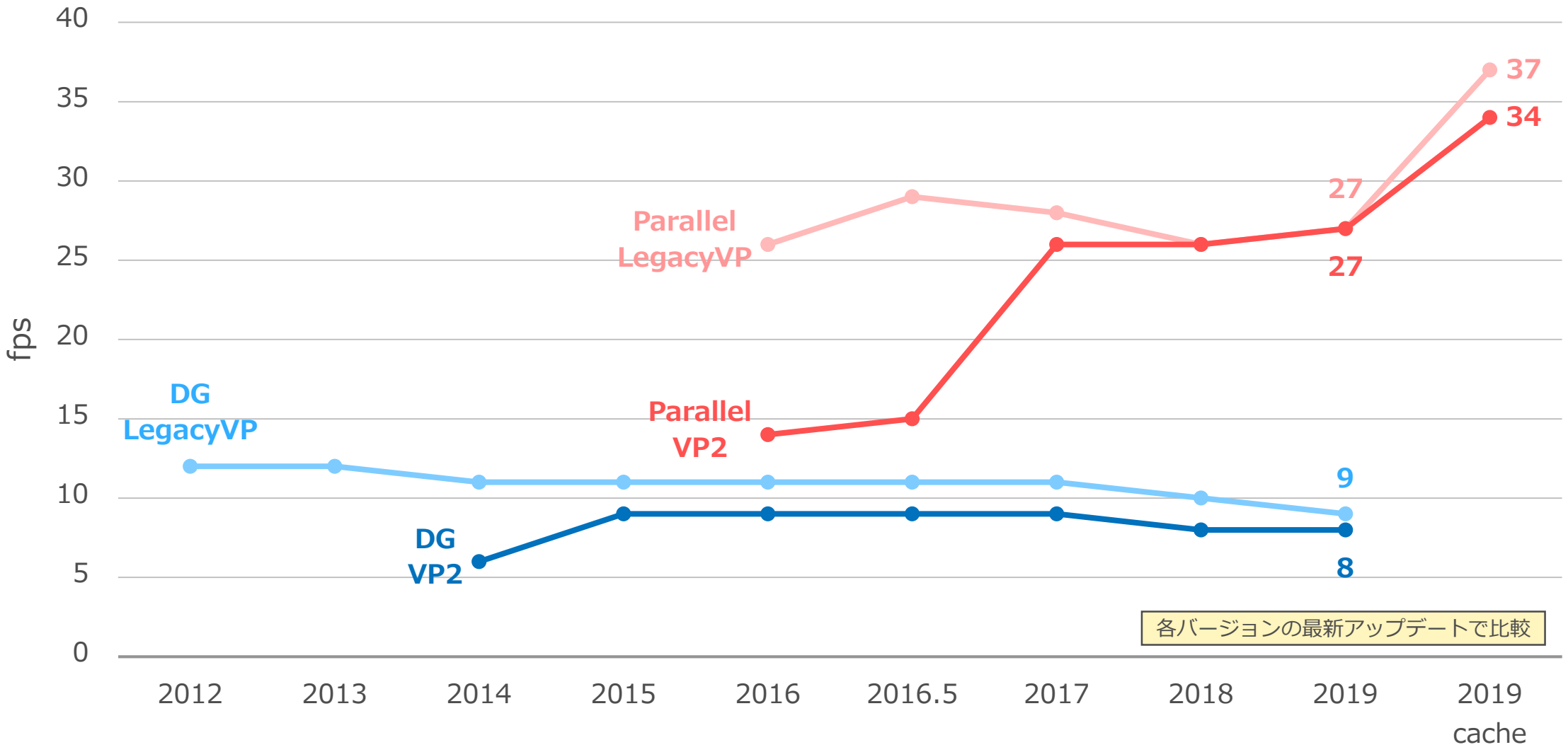
Biped リグ 1 体の場合



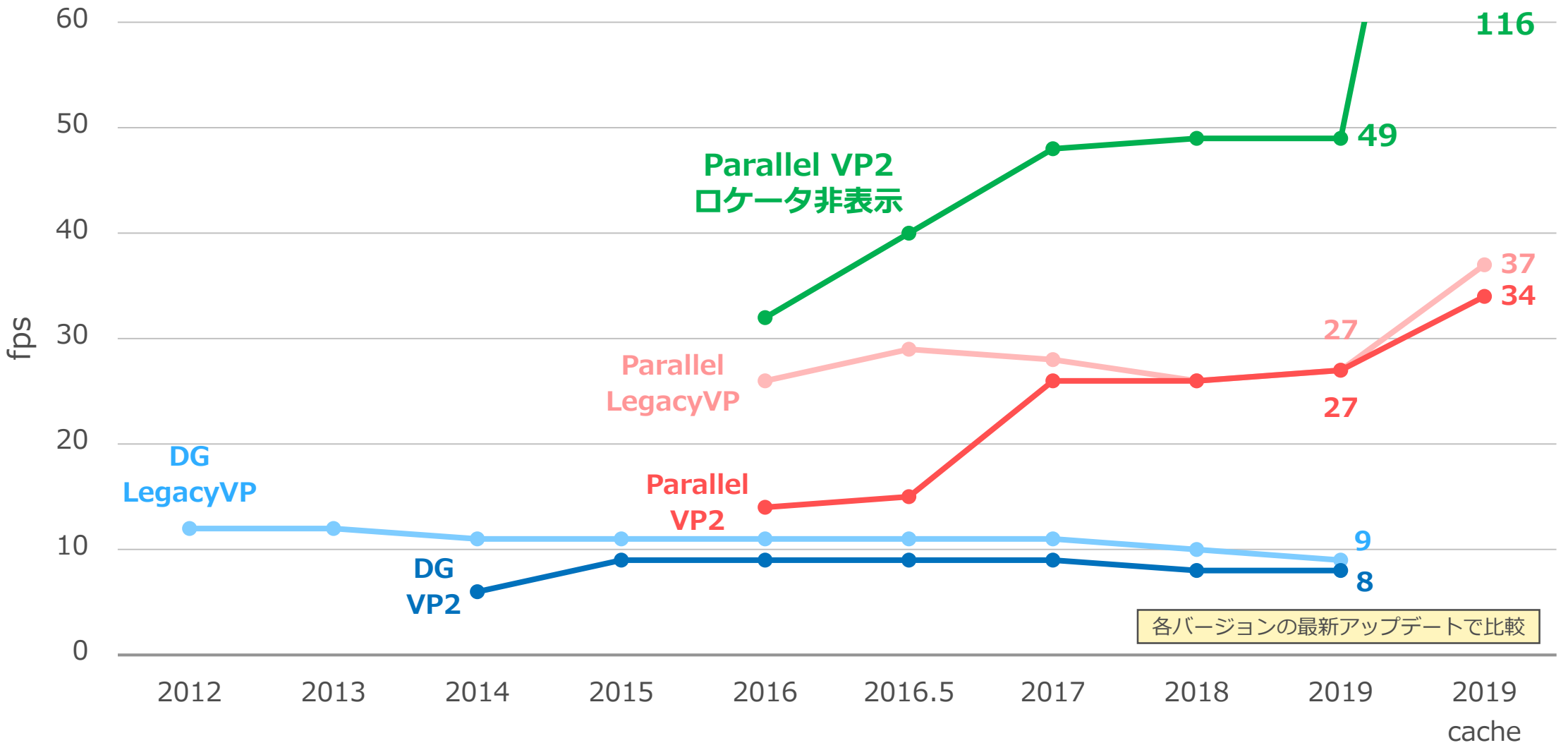
Biped リグ 6 体の場合



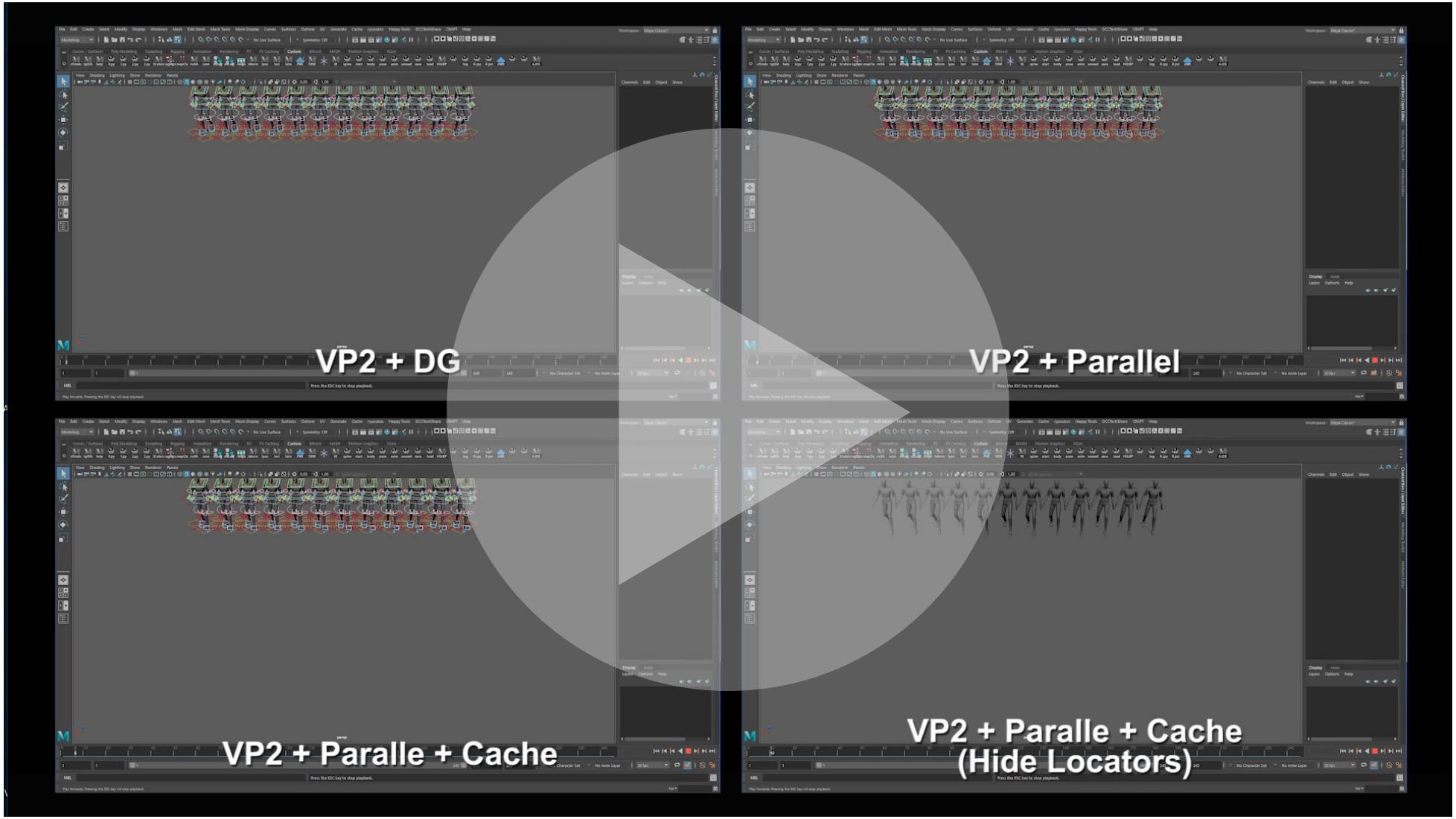
Biped リグ 12 体の場合



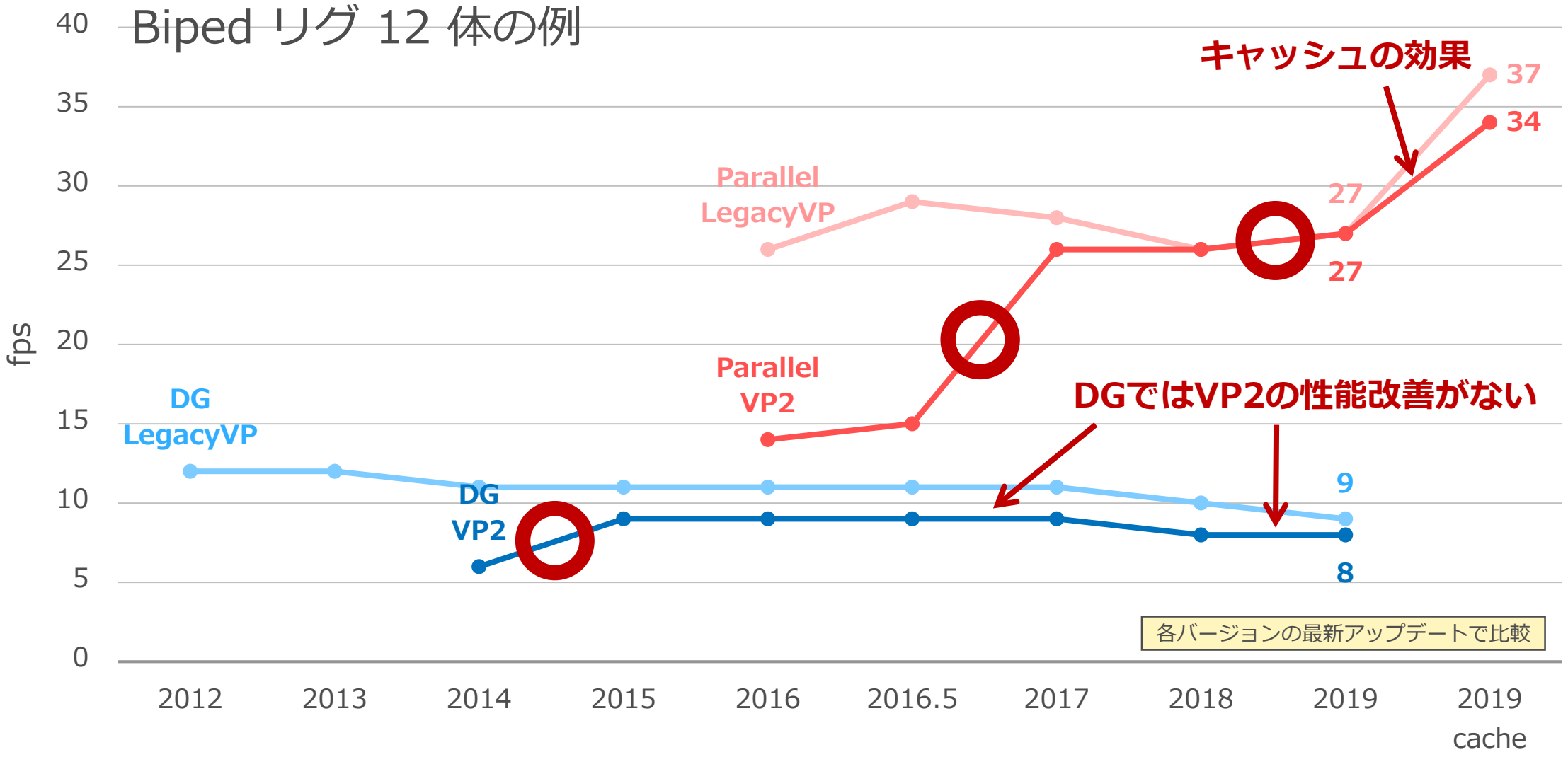
Biped リグ 12 体の場合（ロケータ非表示）



弊社リグのパフォーマンス比較（動画）



VP2 性能改善のターニングポイント

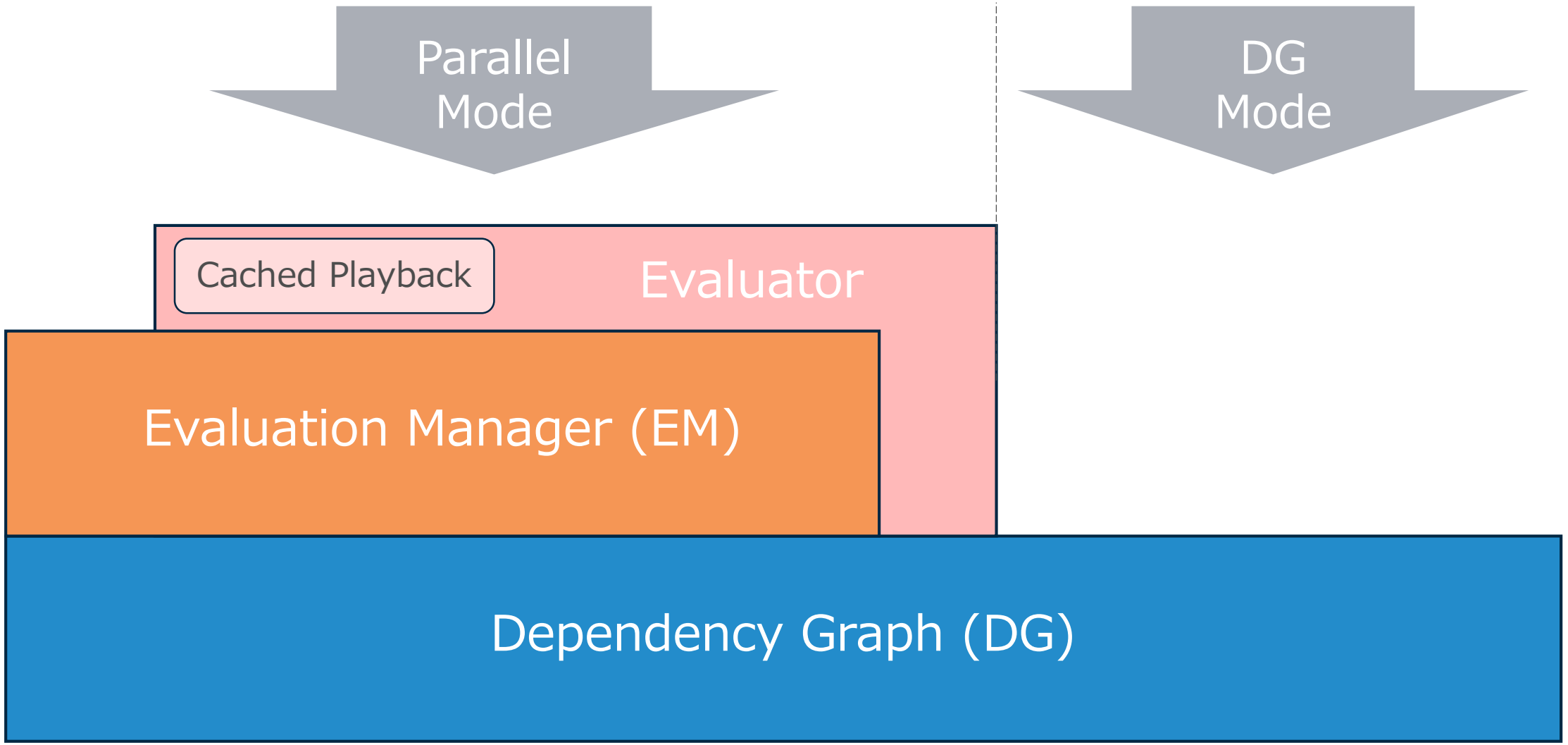


結果から読み取れること

- このケースでは VP2 は Legacy よりやや遅い。
  - バージョンが上がるにつれ改善され、今ではほぼ差がない。
  - VP2 の優位性は条件によって大きく変わる。例えば GPU スキニング。
  - カスタムロケータ描画の負荷は高め。キャッシュの効果も小さい。
- DG の性能は徐々に低下しているが Parallel は徐々に向上。
  - ただし、2017→2018 では、やや落ちた。
  - 2019 では、キャッシュで大きく向上。

VP2 と Parallel を使いこなすことが必須になりつつある

シーン評価システムレイヤー



Evaluation Manager (EM)

- 従来のシーン評価のしくみ（DG）を維持しながら、その上に構築された。
- とても複雑なしくみで、API を駆使して実際にテストのノードプラグインを作るなどして動きを見てみないと、もはや理解できないレベル。
- Viewport 2.0 とは直接は関係ないが、結構深く絡んでくることもある。

必要なこと

シーン評価のしくみの理解

- シーン構造などの基本
- DG（ディペンデンシーグラフ）
- パラレル評価
- キャッシュプレイバック

さらに、プラグイン開発者は

- VP2 についても知る
- プラグインを全てに対応させる

API も使って試す
（しくみが見えやすい）

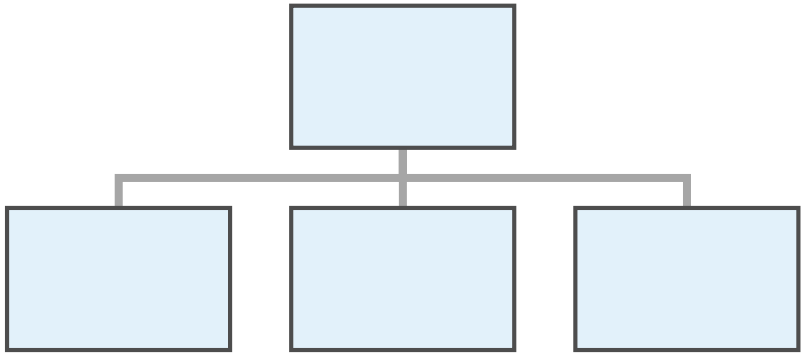
Maya®の基本

シーン構造の基本概念

シーンの構成要素

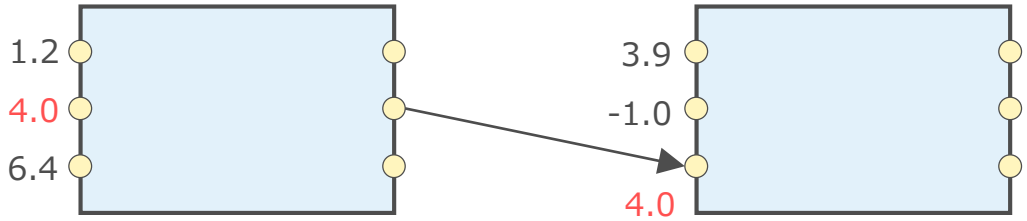
ノード

- 種類によっては「親子関係」を組める。
- 「アトリビュート」を持つ。



アトリビュート

- 「値」を持つ。
- 「コネクション」を作れる。



これが Maya® シーンの全て

ディペンデンシーグラフ（DG）

- Maya® はノードの依存関係（Dependency）でシーンが構築されている。
- 依存関係は、ノードの親子関係とアトリビュートのコネクションによって作られる。
- このグラフを **ディペンデンシーグラフ = DG** と呼ぶ。
- パラレル評価に対して古い評価方法を DG と呼ぶが、実際はパラレルも DG のしくみの上に作られている。

どこまで単純なのか

- あらゆるものが「ノード」と「アトリビュート」。
- 特定の機能のための特別な概念は無い。
- 単一のコンテキスト（グローバル空間上のネットワーク）。

利点もあれば欠点もある

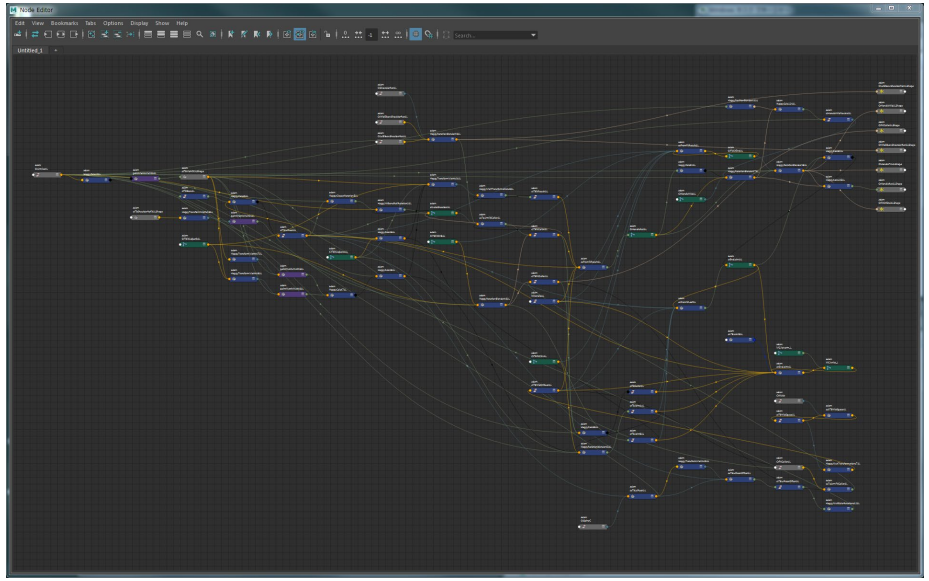
単純さの利点と欠点

利点

- 基本的なしくみの理解が容易。
- 拡張がしやすい。

欠点

- 単純なルールに載せきれないものも。
- 細かいロジックまで露出した複雑なネットワークになりがち。

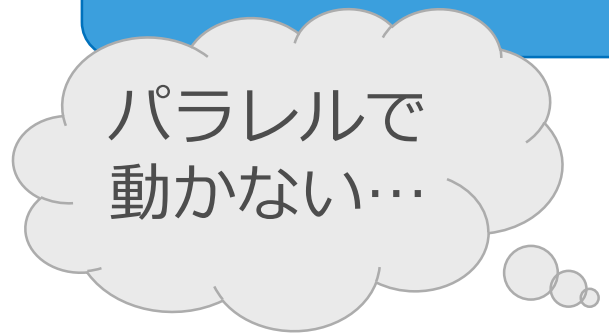


単純さの罠

単純なしくみ上に実現できないもの

特殊（トリッキー）な実装

カオス



開発者は、素直な実装を心がけることが重要

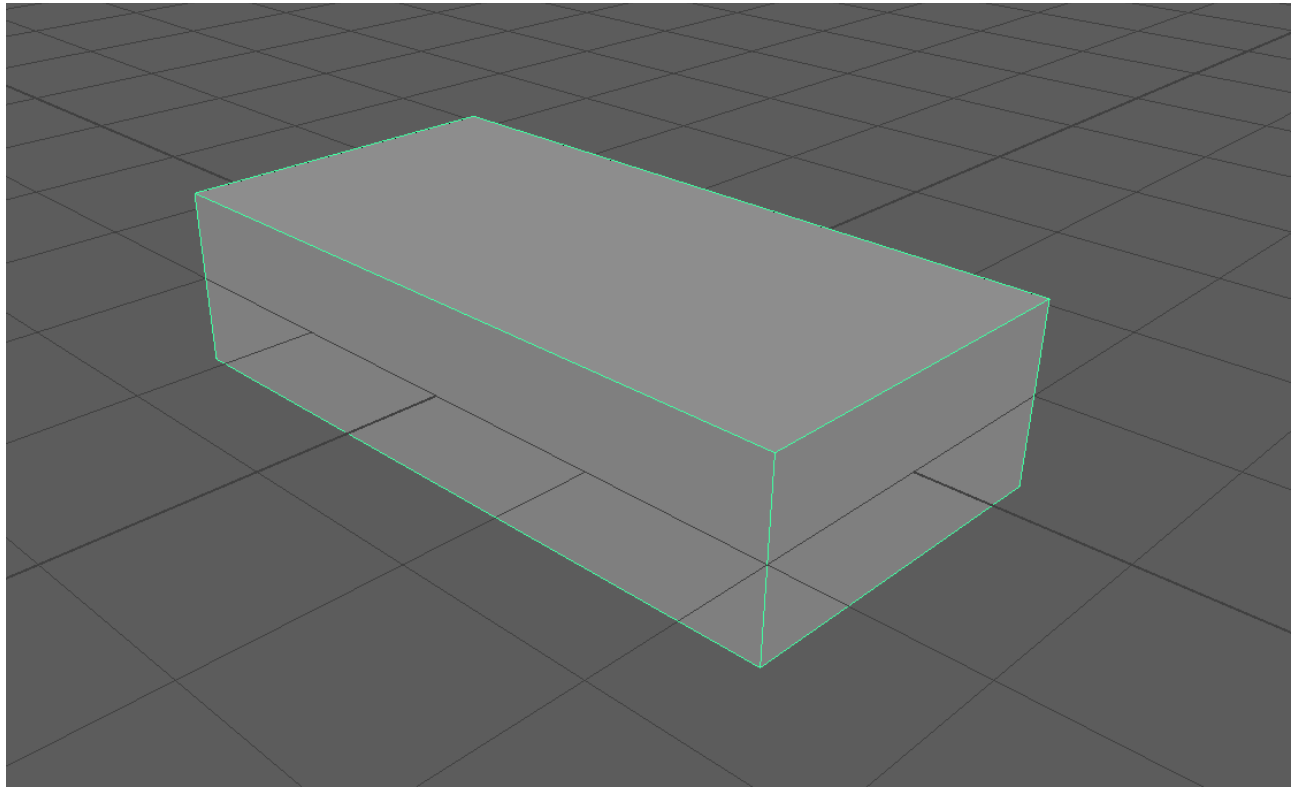
Maya®の基本

シーンファイル

シーンファイルも単純

シーン構造の概念が単純だからこそ、シーンファイルも単純。

たとえば、このような直方体が１つあるシーンでは？



手書きの mayaAscii ファイルの例

```
// transform ノードを作る。
createNode transform -n "pCube1";

// mesh シェイプノードを作る。
createNode mesh -n "pCubeShape1" -p "pCube1";

// キューブ mesh 生成ノードを作り、パラメータを設定する。
createNode polyCube -n "polyCube1";
setAttr ".w" 4;
setAttr ".d" 2;
setAttr ".cuv" 4;

// キューブ生成ノードの出力を mesh シェイプの入力に接続する。
connectAttr "polyCube1.out" "pCubeShape1.i";

// デフォルトマテリアルをアサインする。
connectAttr "pCubeShape1.iog" ":initialShadingGroup.dsm" -na;
```

mayaAscii ファイル ≒ MEL スクリプト

- ほぼ、以下のコマンドの羅列に過ぎない。
  - createNode （ノード生成）
  - addAttr （アトリビュート追加）
  - setAttr （アトリビュートに値を設定）
  - connectAttr （２つのアトリビュートを接続）
  - select （シーンに元から存在するノードの設定用）
- mayaAscii 専用コマンド
  - requires （プラグインの要求）
  - fileInfo （ファイル情報の記述）

MEL
(Maya® Embedded Language)

専用の簡易スクリプト言語。
Python が使えるようになるまでは主流だった。

requires コマンドについて

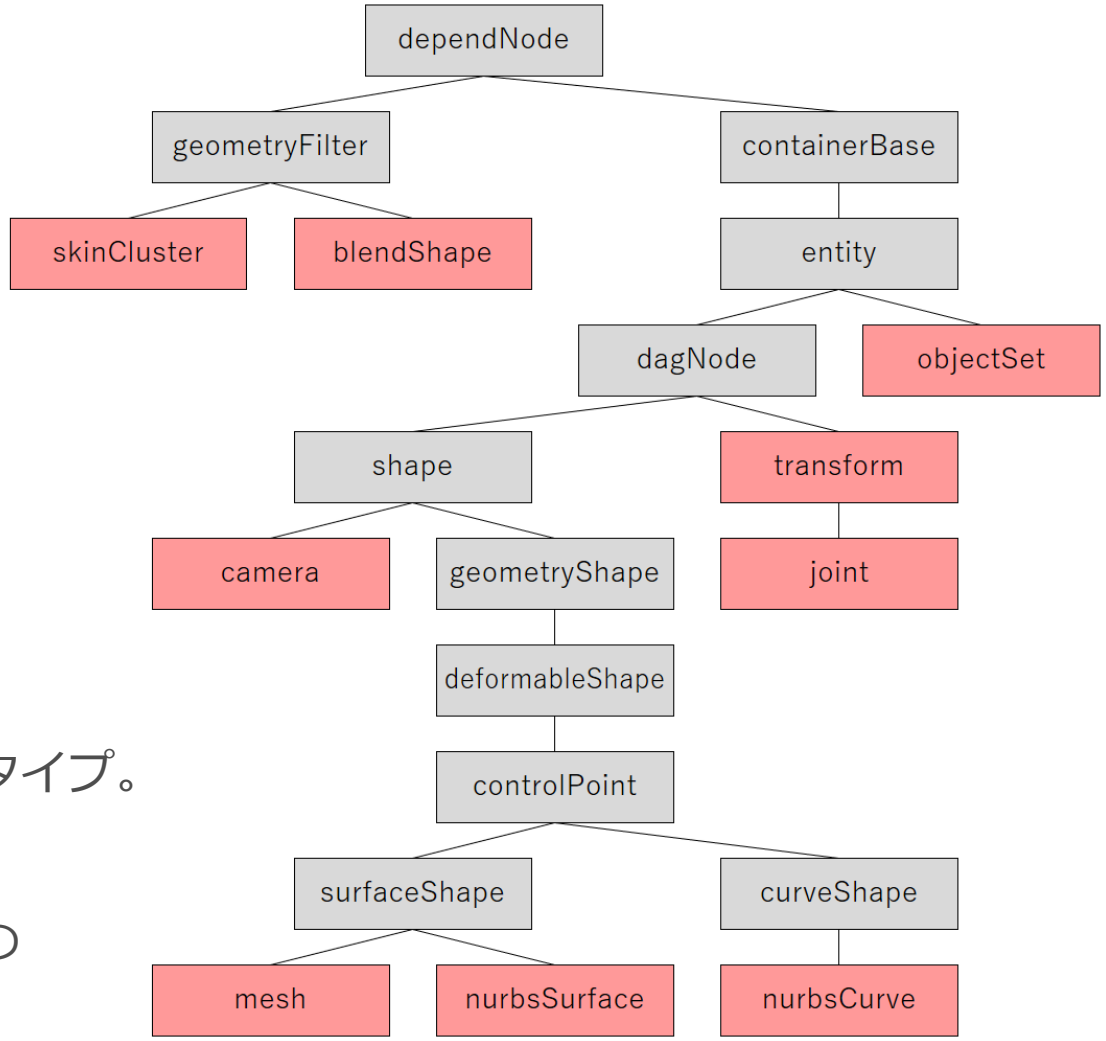
- ファイル内にプラグイン要素が含まれる場合に、ファイル先頭に requires 文が書かれ、そのプラグインを要求する。
- ファイル読み込み時、プラグインが必要に応じてロードされる。
- プラグイン開発者は、オートロードを推奨したり、事前にプラグインをロードさせる必要はない。

Maya®の基本

ノードとアトリビュートによる依存関係

ノードタイプ

- ノードには様々なタイプがある。
- ビルトインタイプの他、
  プラグインでも追加可能。
- タイプは継承が可能で、
  ツリー構造を成す。右図は抜粋。



ノードを実際に生成可能なタイプ。

抽象タイプ。
あらゆるノードは、ルートの
dependNode を継承する。

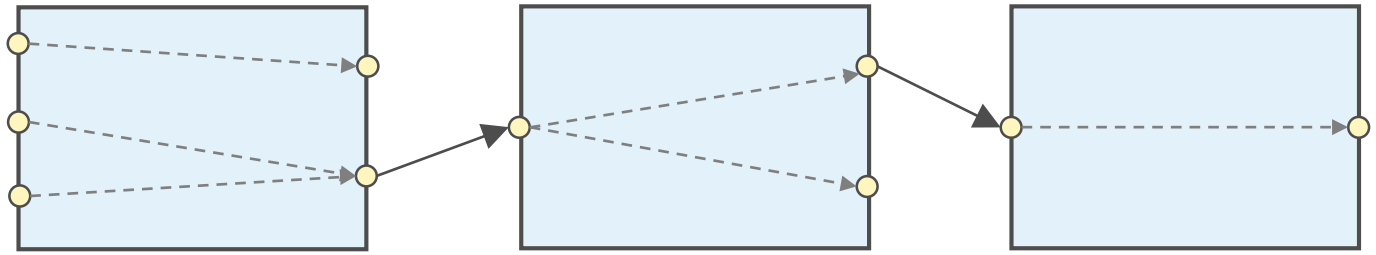
アトリビュートタイプ

ノードタイプによって、アトリビュート構成が決まっている。

- アトリビュートには、格納される値に応じた様々な型がある。
  （整数、実数、文字列、行列、ジオメトリデータなど）
- ノードタイプと違い、継承の概念は無い。
- 型はプラグインでも追加可能。

アトリビュートの依存関係

- DG の依存関係の最小単位はノードではなくアトリビュート。
- ノード外は **コネクション**（下図の実線）
- ノード内は **Attribute Affects**（下図の破線）



Attribute Affects は GUI 上では見えないが、
常識的に想像できることが多い。

通常 Attribute Affects は固定だが、動的なものもある。

Attribute Affects の調べ方

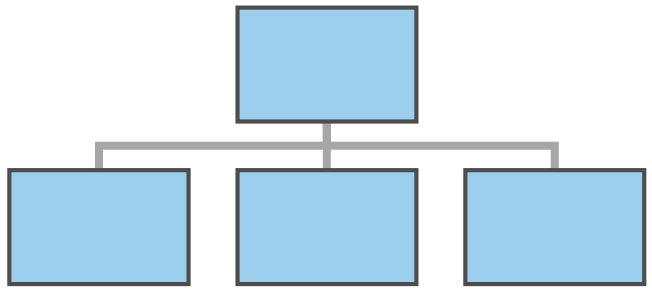
静的なものに限り、
API の MFnDependencyNode::getAffectedAttributes() や
affects コマンドで調べることができる。

```
// 指定アトリビュートが影響を与えているものを得る
affects "attrname" "node" -by;
affects "attrname" -t "nodetype" -by;

// 指定アトリビュートが影響を受けているものを得る
affects "attrname" "node";
affects "attrname" -t "nodetype";
```

DAG ノード

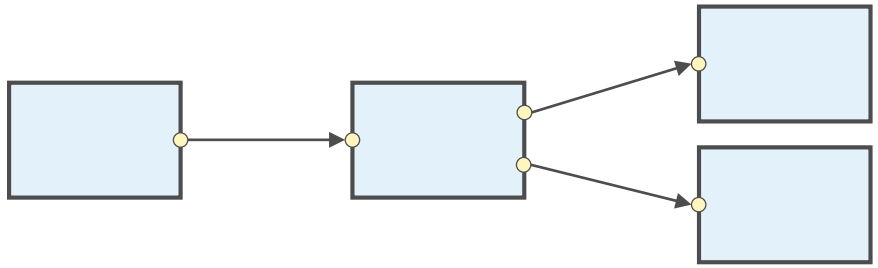
DAG ノード

- 親子構造を作れる。
- dagNode を継承する。
- transform や joint や shape など、一般ユーザーの多くがノードとして認識するもの。



非 DAG ノード

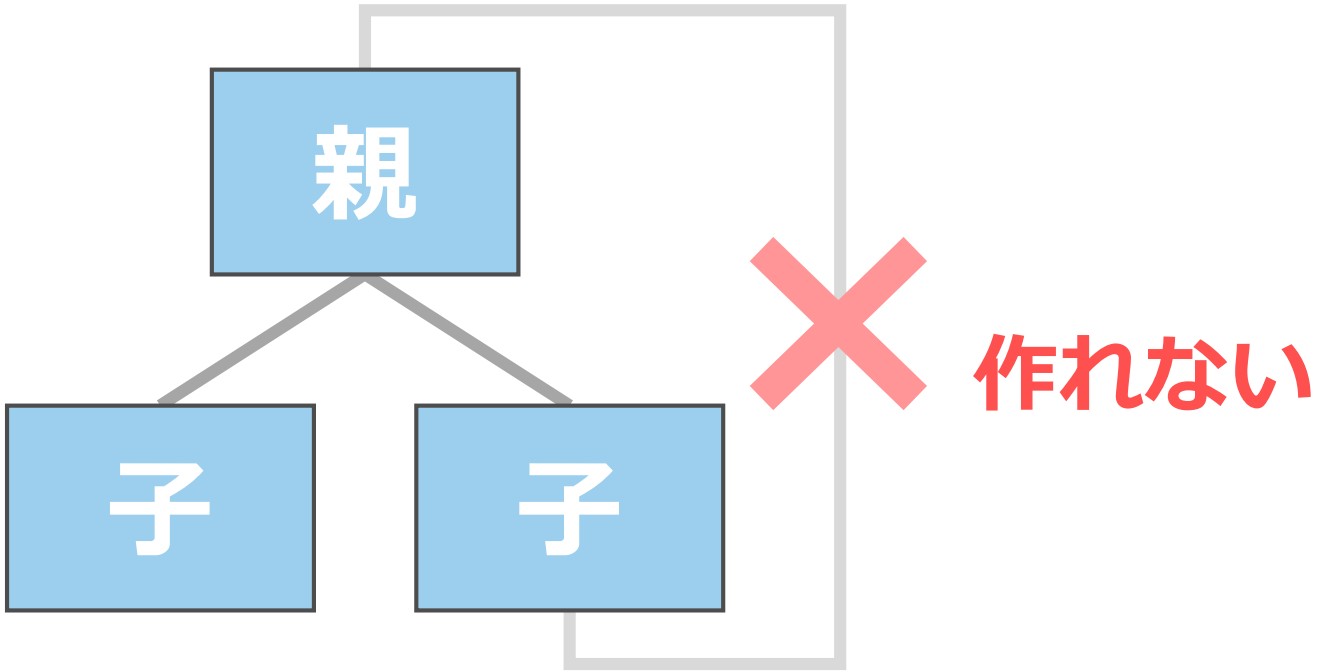
- 親子構造を作れない。
- dagNode を継承しない。
- 一般ユーザーの多くが、モデリングやデフォーメーションのヒストリや、リグのネットワークとして認識するもの。



DAG とは？

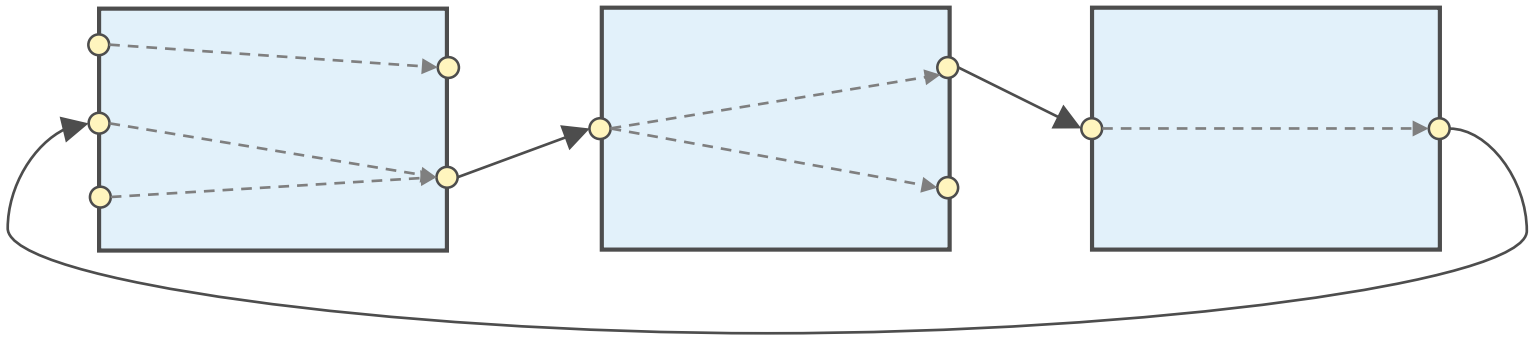
DAG = Directed Acyclic Graph（有向非巡回グラフ）

親子という方向性があり、サイクルを作ることができないグラフ。



サイクル

アトリビュートのコネクションでは、サイクルを作ることができてしまう。

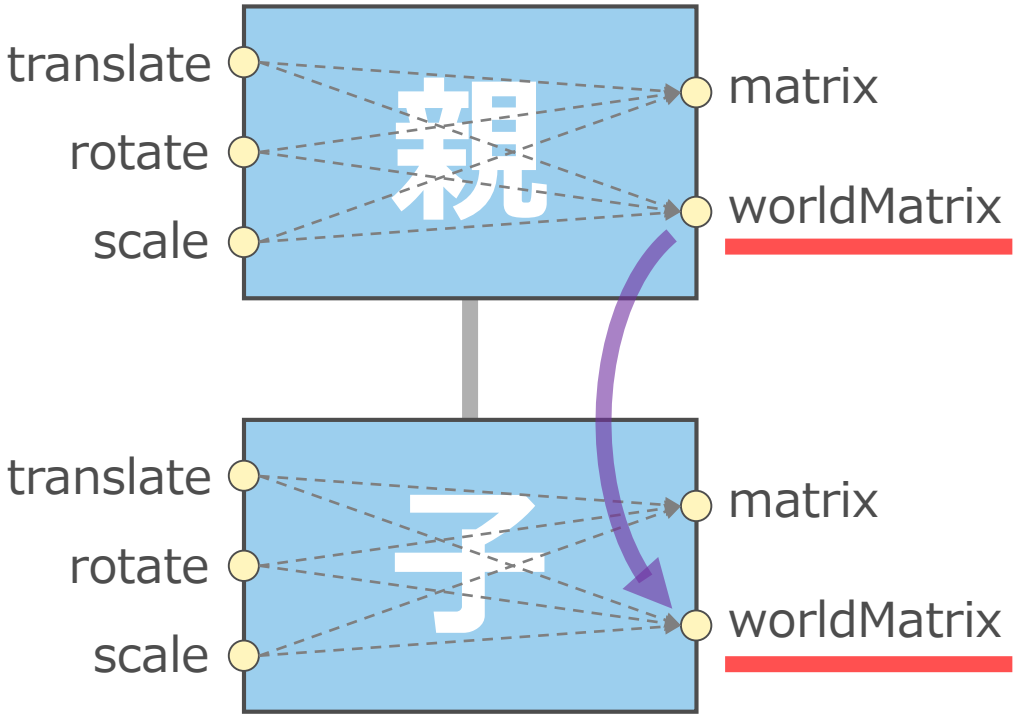


作れてしまうが問題

サイクルがあると評価結果は不安定となるため、避けなければならない。プログラミングに例えるとバグのようなもの。

親子関係による依存関係

親子関係によって worldMatrix の依存関係が作られる。



子の worldMatrix は
親の worldMatrix に
暗黙的に依存している。

依存関係についてのまとめ 重要!!

- 依存関係の最小単位は、ノードではなくアトリビュート。
- 依存関係は以下の３種類の組み合わせで作られる。
  - コネクション
  - Attribute Affects
  - DAGノードの親子関係

グラフ理論にあてはめると、アトリビュートがノード（点）となり、
上記３種類がエッジ（結線）となる有向グラフである。

ノードの役割 重要!!

DG のようなシステムにおいて、ノードは「関数」である

与えられた入力データに基づいた計算結果を出力することが仕事。

ノードは「外の世界」を知るべきではない

- 入出力として明示されたデータ以外にアクセスしない。
- 入出力のその先の関連を検索すべきではない。
- 出力先を直接書き換えるのではなく、計算結果を返すのが仕事。
- 入力より前や出力より先の事情なんて知らなくていい。

Maya®の基本

プログラミング言語

使用できるプログラミング言語

mel

- 積極的に書くことはもうない。
- 多くのシステム mel があるので、読むことは必要。

Python

- 多くの仕事はこれで事が足りる。

C++

- 特に速度が要求される部分で使う。

C#

- Windows 版でのみ、.NET をどうしても使いたいときに使う。

プログラムの種類

スクリプト

- コマンドや API を使って一連の作業を自動化する。
- コマンドや Qt (PySide 等) を使って GUI を作る。
- コンパイル不要。

プラグイン

- API を使ってコア機能を追加する（ノードやコマンドなど）。
- コンパイルが必要。
- Python で書く場合はコンパイル不要。

各プログラミング言語でできること

	コマンド 呼び出し	API 呼び出し	プラグイン 開発	コンパイル 不要	Multi- Platform
mel	○			○	○
Python	○	○	○	○	○
C++	※	○	○		○
C#	※	○	○		Windows のみ

※ API から mel や Python を呼び出すことでコマンドも呼び出せる。

Maya® API とプログラミング言語

C++ API おすすめ!!

- 元祖。

Python API (旧API)

- C++ API を SWIG でほぼ自動的に対応させたもの。
- 使いにくい。

Python API 2.0 おすすめ!!

- SWIG ではなく真面目に作られたもの。
- 使いやすく旧 API よりは高速だが、すべての機能が網羅されてはいない。

C# API

- C++ API を SWIG でほぼ自動的に対応させたもの。

Python vs C++

開発効率と実行速度の点でトレードオフの関係

- 開発効率の点で、多くの仕事に Python を使うべき。
- 速度が必要なコア機能に C++ を使う。

しかし！！

「ノード」は Python で開発してはならない

Python は **GIL (Global Interpreter Lock)** により、
同時に実行できるスレッドは１つだけに制限される。

Maya®の基本

API の基本的なクラスの紹介

MObject

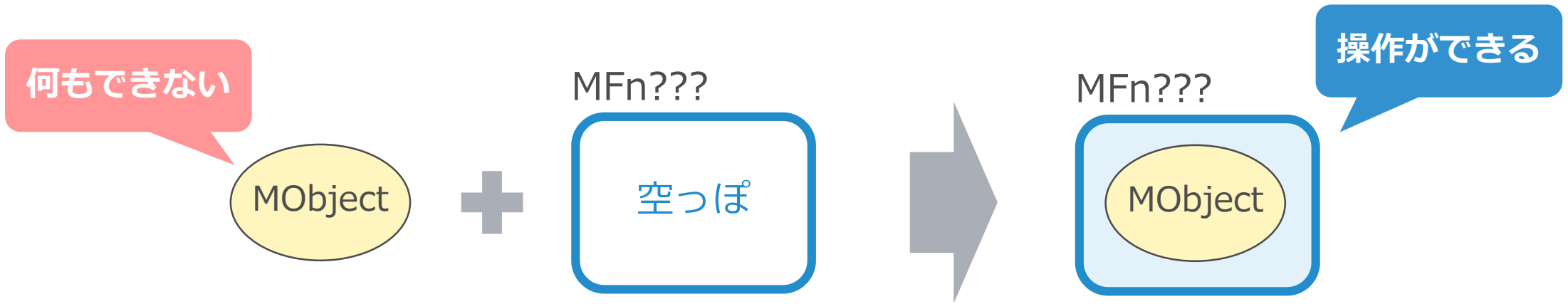
- Maya® の何らかのオブジェクトの参照。
  （ノード、アトリビュート、データ…など）
- ほとんど、あらゆるものが MObject 。
- C や C++ 言語の void ポインターのようなもので、これだけでは何もできない。

MFn…（ファンクションセット）

- Maya® のオブジェクトを操作するためのクラス。

  様々なオブジェクト用にクラスがあり、それぞれに応じたメソッドを持つ。

- MObject に対するラッパーのようなもの
  - create メソッドを呼ぶと MObject を返す。
  - 既存のものを操作するには MObject をセットして使う。



MPx…（プロキシ）

- Maya® に機能を追加するためのクラス。
  - MPxCommand … 新しいコマンドを追加
  - MPxNode … 新しいノードタイプを追加
    - MPxDeformer など、より特化したものもある。
  - MPxData … 新しいアトリビュートタイプを追加
- プラグインとは、これらを実装して Maya® に登録するもの。

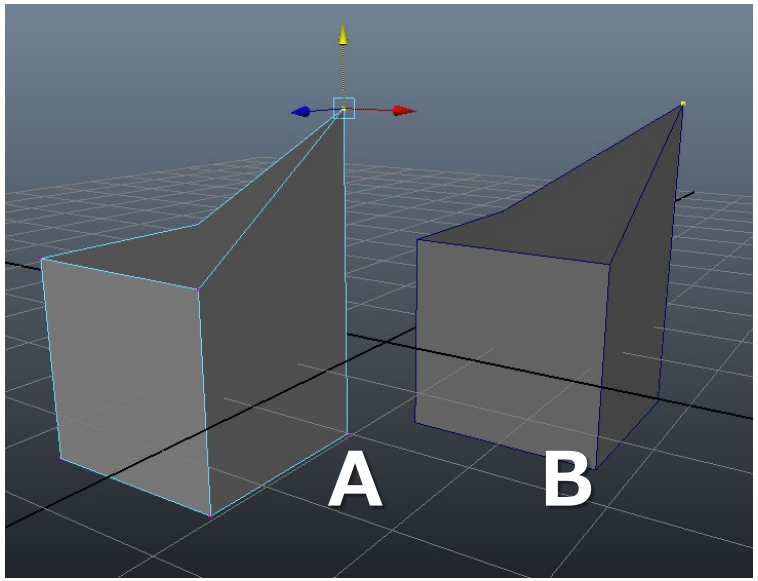
MPlug

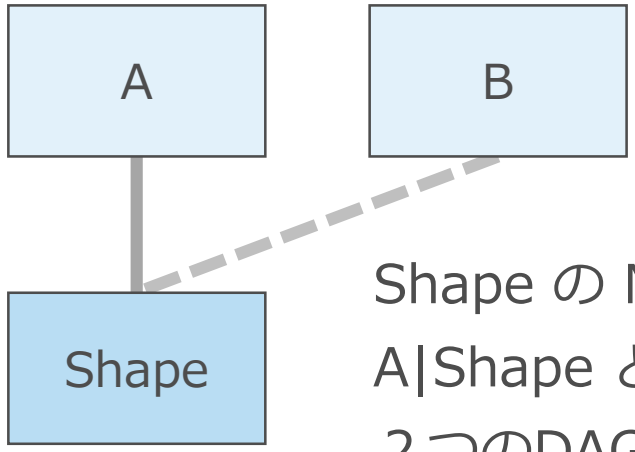
- プラグとは、アトリビュートの実体。
- MFnDependencyNode::findPlug() で得られる。
- ノードとアトリビュートの MObject から得られる。

  ノードの MObject は実体への参照だが、アトリビュートの MObject は実体ではなく定義のようなもの。

MDagPath

- DAGノードの階層パスを持つ MObject ラッパー。
- instance コピーがある場合に１つの MObject に複数の DAG パスがあることになる。





Shape の MObject は１つだが、
A|Shape と B|Shape の
２つのDAGパスがあり、
ワールドマトリックスが異なる。

MDataBlock

ノードの内部実装で、プラグに効率的にアクセスする手段。

- プラグに相当する MDataHandle を得て読み書きする。
  - MPlug を介するより効率が良い。
  - たとえば set が dirty 伝搬（後述）を引き起こさない、など。
- シーン評価のしくみにも深くかかわる。
  - inputValue() で、評価しつつ MDataHandle を得る。
  - outputValue() で、評価せずに MDataHandle を得る。

Maya®の基本

アトリビュートの詳細

数値型とデータ型

アトリビュートには数値型とデータ型がある。

数値型

- addAttr コマンドでは –at で指定する型。
- int や float など、単純な数値が多い。
- 取り扱いも値で直接の set や get になる。

データ型

- addAttr コマンドでは –dt で指定する型。
- matrix 等の他に、mesh 等のように大きめなデータが多い。
- 取り扱いは MObject での参照による set や get になり、参照コピーで済む場合は効率が良い。

数値型アトリビュートの例

型名	概要	備考
message	値を持たず、接続による関連性構築のためにのみ用いられる	
bool	off と on の２値	
long	32bit 整数	
enum	選択式の値	
float	32bit 浮動小数点数	
double	64bit 浮動小数点数	
doubleLinear	double の一種で位置座標や距離の単位を持つ型	
doubleAngle	double の一種で角度の単位を持つ型	
time	時間	
float3	3個の float のコンパウンド	
double3	3個の double（Linear や Angle も可）のコンパウンド	
compound	一般のコンパウンド	
matrix	数値型分類の 4x4 行列	データ型を推奨

データ型アトリビュートの例

型名	概要	備考
string	文字列	
matrix	4x4 行列	
mesh	メッシュ ジオメトリ データ	
nurbsCurve	NURBS カーブ ジオメトリデータ	
nurbsSurface	NURBS サーフェース ジオメトリデータ	
stringArray	string の配列データ	
Int32Array	long の配列データ	
doubleArray	double の配列データ	
vectorArray	３次元ベクトル(MVector)の配列データ	
pointArray	４次元ベクトル(MPoint)の配列データ	
double3	データ型分類の double3	数値型を推奨
function	何らかの機能で用いられるコントロールカーブの参照	仕様は非公開
???	プラグインにより追加されるデータ型	

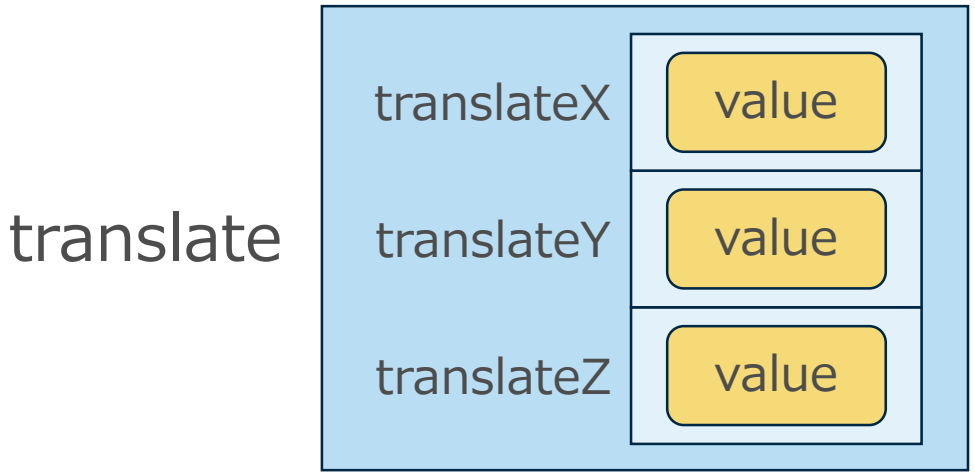
message 型アトリビュート

- 値を持たないアトリビュート。
- つまり、コネクションしかできない。
  - 中をメッセージが流れるわけではない。
  - どう使うかは使い方しだい。
- 一般的に関連を定義するような目的で使う。
  - ノードの外側からの管理で使う分には良いが、ノード自身がこのコネクションを辿って外の世界を把握するのは基本的にはご法度。
  - DG の基本「ノードは関数」を思い出そう。

compound 型（コンパウンドアトリビュート）

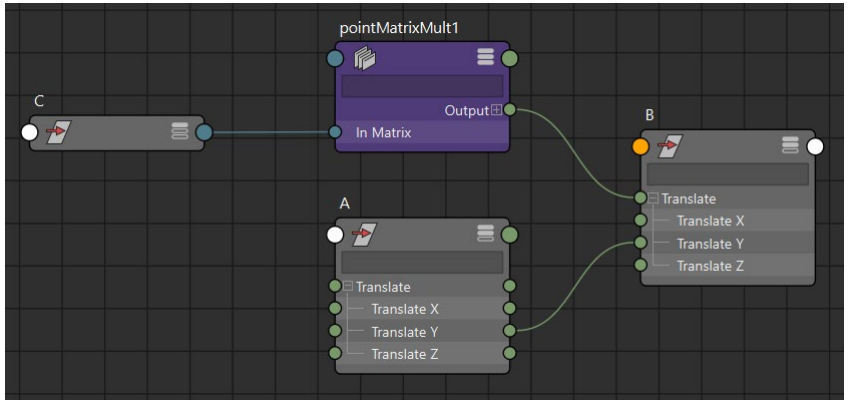
他のアトリビュートを子に持つアトリビュート。

- アトリビュートのツリー構造を作れる。
- 自身は値を持たない。
- double3 型は、 3 値に特化した compound の一種。
  要素ごとでも 3 個まとめてでも、値の get や set や接続ができる。

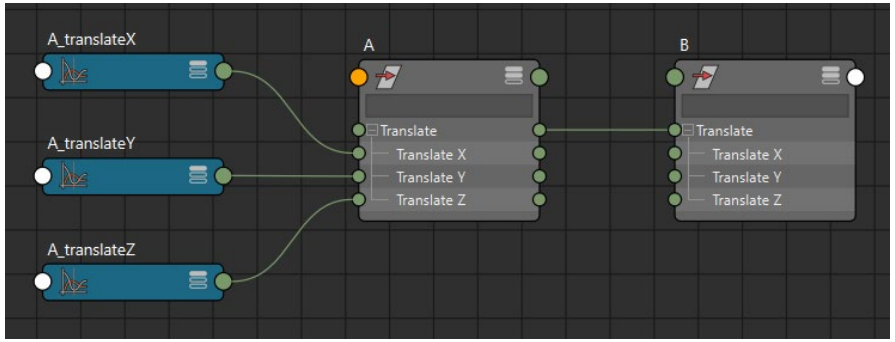


double3 コネクションの注意点

- DG 評価では、要素より double3 で接続した方が高速。
- 入出力の競合や不整合は避ける。
  - double3 と要素の両方への入力。
  - 入力が要素で出力が double3 、またはその逆。



入力の競合



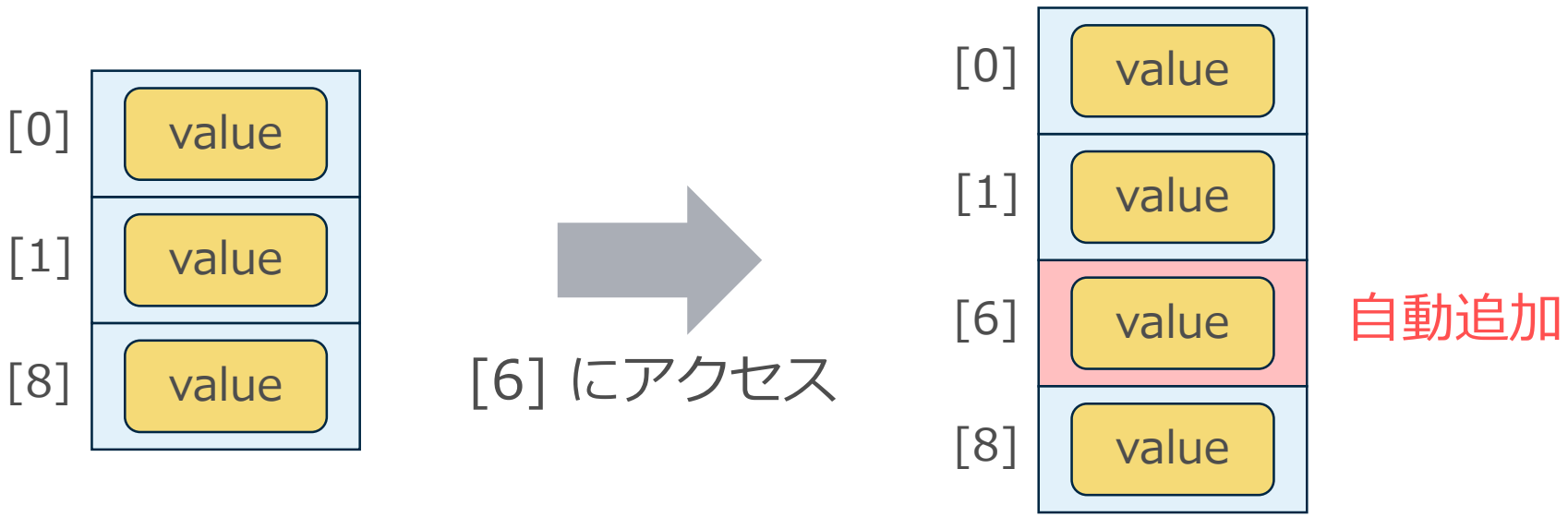
入出力の不整合

transform ノード等の一般ユーザーが触る箇所は基本的に x,y,z 要素で接続、裏方のノードは double3 で接続するのが望ましい。

マルチアトリビュート

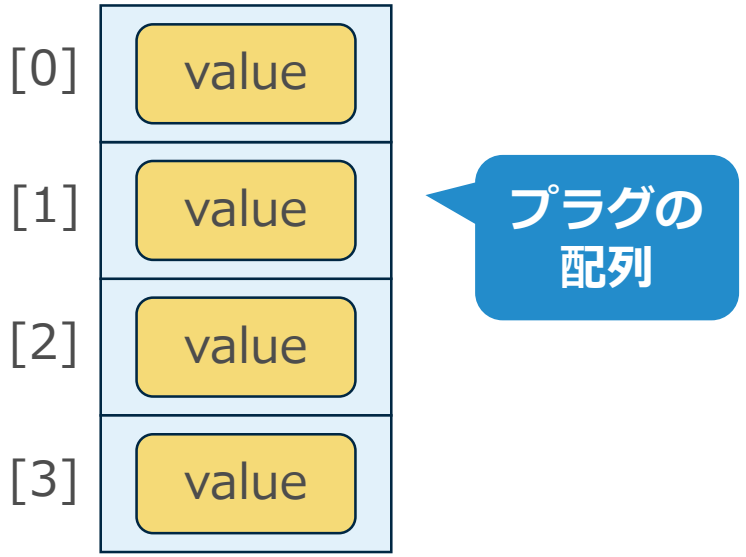
プラグが配列となるアトリビュート。

- アトリビュート追加時のオプションなので、どの型もマルチになれる。
- 要素プラグは最初は 0 個で、アクセスした際に自動生成される。
- インデックスは欠番も許される。

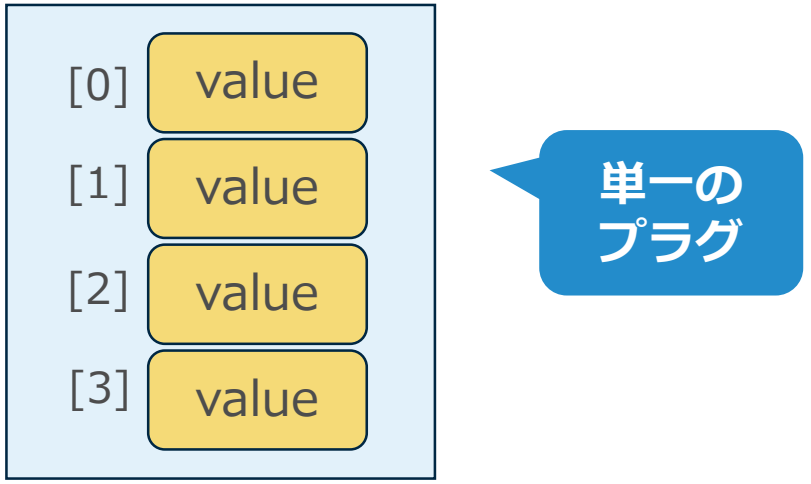


マルチと配列データ型との違い

- ～Array という型は、配列データを持つ単一のアトリビュート。
- たとえば、double 値を複数扱うには「double 型のマルチ」と「doubleArray 型」という異なる手段を選べる。

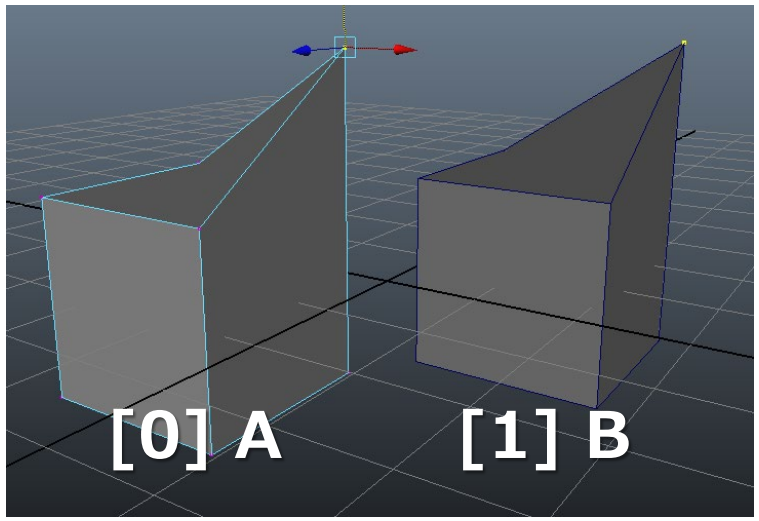


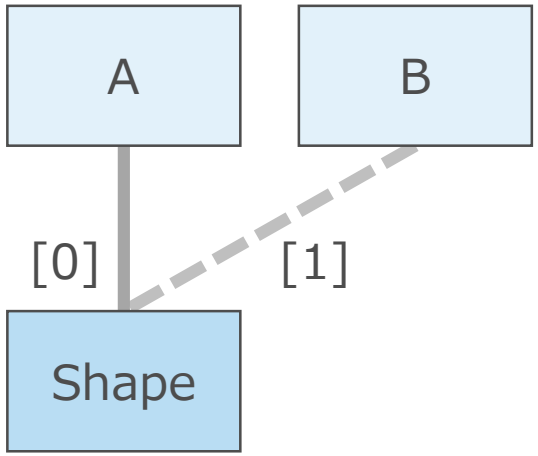
double 型マルチアトリビュート



doubleArray 型アトリビュート

ワールド空間系出力とマルチアトリビュート





- 各種 DAG ノードのワールド空間系出力は、
  複数のインスタンスに対応するため、
  マルチアトリビュートになっている。
  - dagNode の worldMatrix
  - locator の worldPosition
  - mesh の worldMesh

  などなど
- インスタンスインデックス
  - コマンドでは自動的に補完・補正される。
  - API では MDagPath::getInstanceNumber() で得る。

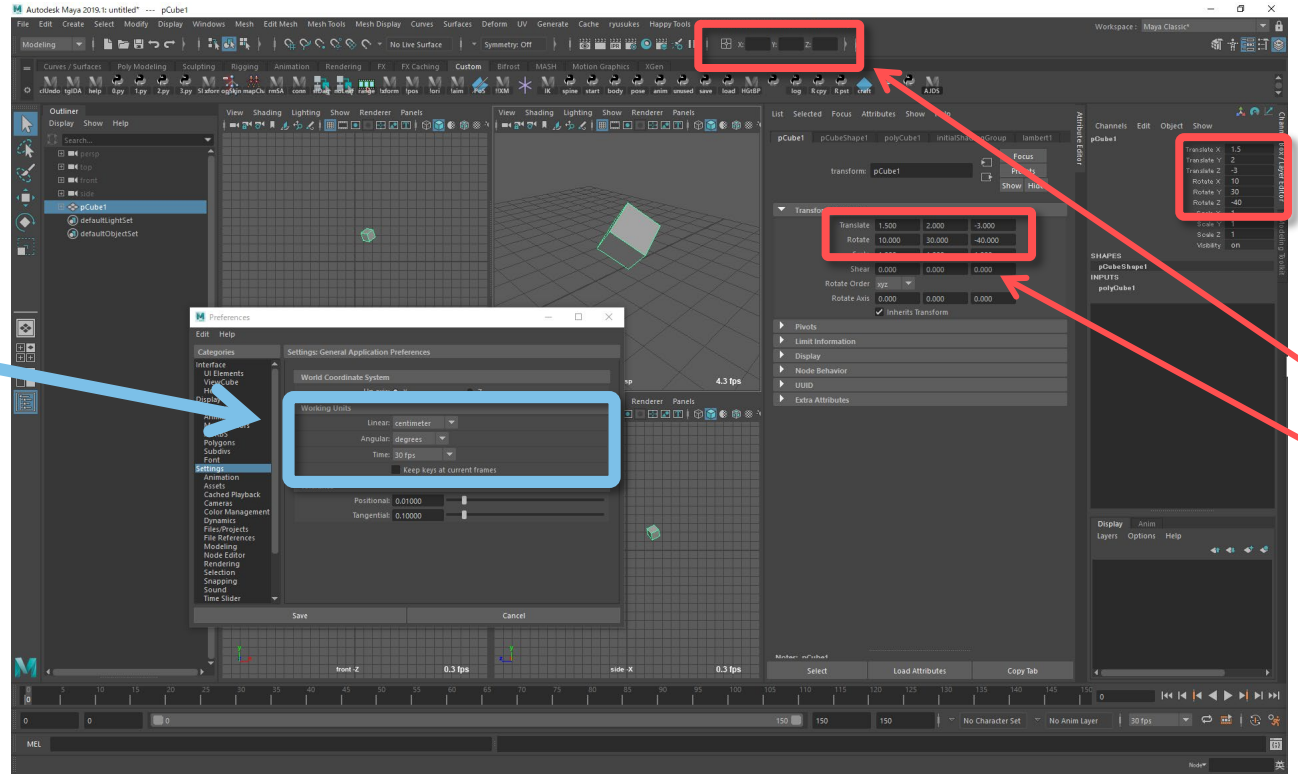
indexMatters 設定について

マルチアトリビュートの設定 indexMatters は、インデックスに意味があり接続時に明示する必要があるかどうかを表す。

- アトリビュート作成時のデフォルトは true 。
- 「複数のものを束ねておく」だけの用途で、インデックスが無意味なものは false とする。
- false の場合、connectAttr –nextAvailable (-na) が利用できる。シーンファイルを読み直すと、欠番が詰められたりする。

単位を持つ型

- distance, angle, time はシーンの設定によって単位が定められる。
- 内部単位は固定であり、UI 上で見える値はシーン設定に応じて単位換算された値となる。



それぞれの内部単位など

型名	用途	内部単位	分解能 (tick)	API Class
doubleLinear	位置座標や距離	centimeter		MDistance
doubleAngle	角度	radians		MAngle
time	時間（フレーム）	second	1/141120000 秒 (2017update2 以前は 1/6000 秒)	MTime

いずれの型も 1 が「内部単位」の 1 とする倍精度浮動小数点数（64bit）。

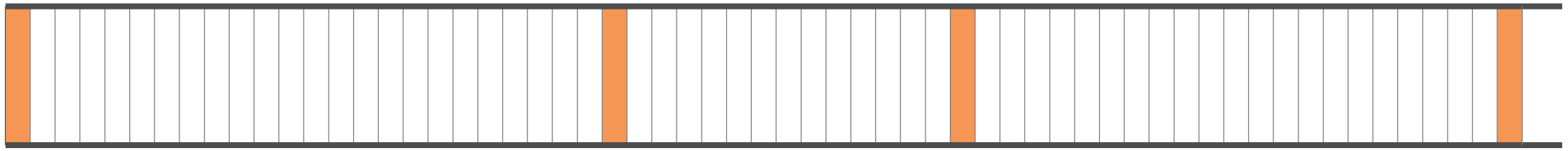
ただし、time は深部では整数表現となっており tick が決まっている。
2017 Update3 より 64bit 化されたが、それより前は 32bit 整数精度だった。

time ノード

- グローバルタイムを表すものとして、
  シーン中にただ１つ time1 ノードが存在する。
- 常にカレントタイムを出力。
- タイムコード表示の管理。
- シーンタイムワープの管理。

タイムラインとキーフレームデータ

24 fps



0 frame
(0.0 sec)

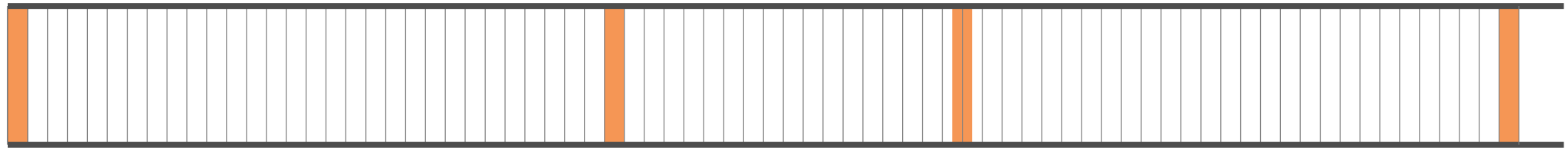
24 frame
(1.0 sec)

34 frame
(1.4166… sec)

60 frame
(2.5 sec)

キーフレームは内部単位で管理されているため、
途中で fps を変更しても実時間は変わらない。
(Keep keys at current frames = off の場合)

30 fps



0 frame
(0.0 sec)

30 frame
(1.0 sec)

42.5 frame
(1.4166… sec)

75 frame
(2.5 sec)

プログラミングにおける単位の扱い

API での扱い

- 内部単位での扱いになる。
- 単位管理を内包するクラス（MDistance, MAngle, MTime）もあるが、生の double 型で直接扱って構わない。

コマンドでの扱い

- シーン設定により単位換算された値での扱いになる。
- setAttr, getAttr, setKeyframe など、様々なコマンド。
- 頂点座標の扱いは単位管理されておらず、内部単位（cm）となる。

スクリプティングの罠

- doubleAngle 型を degrees で get / set しているスクリプトは、角度の単位設定が degrees であることに依存している。
- radians 設定のシーンはほぼ有り得ない奇特な例だが、同様のことが doubleLinear や time にもいえる。
- 単位設定に依存したスクリプトが問題になるのは、特定の単位を前提とする他モジュール（関数）を呼びだす場合や、値をハードコードしている場合だけなので、角度ほどは問題が表面化しにくいだけ。
- 頂点座標やマトリックスに現れる値は内部単位なので注意が必要。

スクリプトでも常に内部単位固定でコーディングするのが良い。
コマンド利用時はシーン設定と内部単位を変換するラッパーが必要。

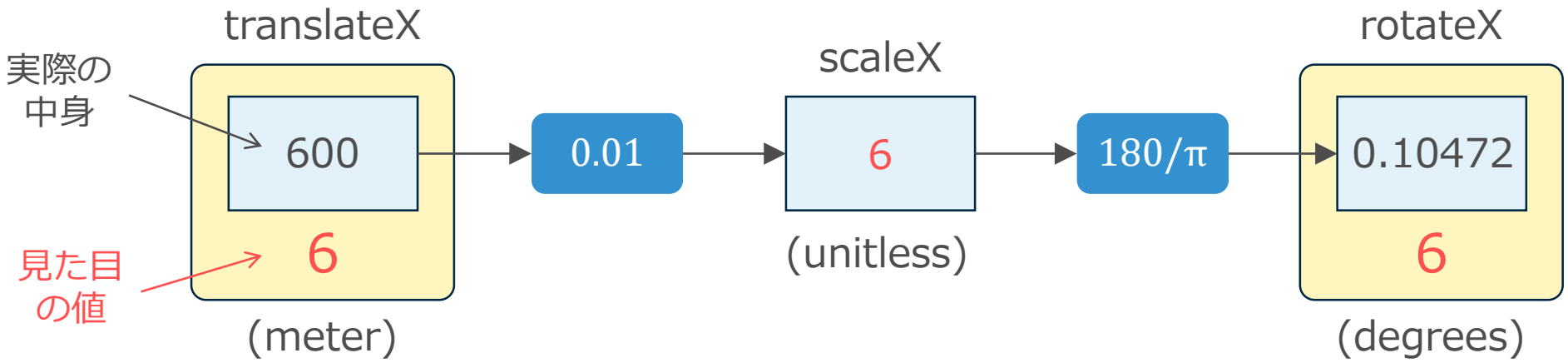
unitConversion ノード

- 単位の異なるアトリビュートを接続して 1.0 以外の「単位変換係数」が必要な場合に自動挿入される。
- 「単位変換係数」を持つ単なる乗算ノード。

一方が time 型の場合だけ、
timeToUnitConversion や unitToTimeConversion というノードになり 1/6000秒 tick との変換係数が設定される。
time には tick があるため特別に扱われていると思われるが、32bit 精度の名残のままとなっている。

unitConversion の考え方

- UI 上に見えているのは「単位設定による見かけ上の値」。
- 接続時に、ユーザーの期待する見た目通りとなる係数を設定。



この係数は、そのときのままシーンに焼き付けられる。
その後、単位設定を変えても係数は変わらない。それが正しい。

generic 型アトリビュート

- 様々な型の入出力接続と値のセットに対応する型。
- どの型を許容するかは API で設定する。

使用例：

- デフォーマーの入出力

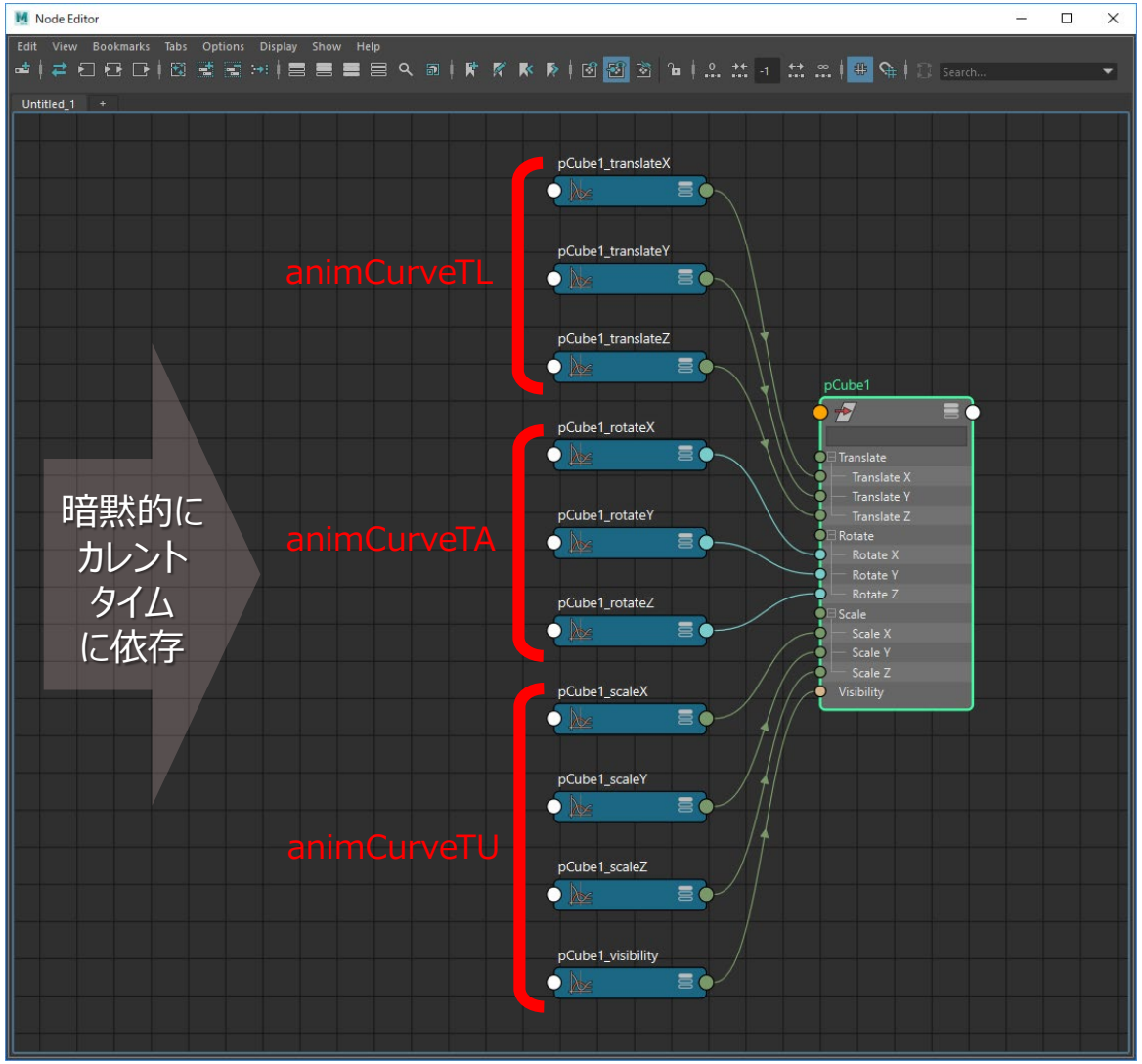
  ジオメトリタイプを限定しない点群のフィルター。
- 様々な型に対応した計算ノード

  unitConversion が挟まるのを防ぎ、オーバーヘッドを減らす。

アニメーションカーブ

- アニメーションカーブもノード。
- 入出力の型に応じたノードタイプが用意されている。
  - 出力は全種類（time, unitless, doubleLinear, doubleAngle）
  - 入力は time と unitless の２種類
- 通常のアニメーションカーブは time 入力。
  - time は暗黙でカレントタイムに依存。
- ドリブンキーでは、入力型に応じて使い分けられる。
  - 入力に unitConversion が挟まることもある。

animCurve ノードの例



ノードタイプ: animCurve**IO**

I … 入力型 (U, T)

O … 出力型 (U, L, A, T)

入力に何も接続されていないが、
time 入力の animCurve は特別で、
time1 ノードの output が入力
していることとほぼ等しい。

数値型とデータ型の両方があるタイプ

double3

- 一般的なのは数値型。コンパウンドなので x, y, z 要素個別に扱える。
- データ型は x, y, z 要素個別には扱えない。
- 常に数値型を使えば良い。

matrix

- 標準ノードでは全てがデータ型。
- サンプルプラグインでは殆どが数値型の例（騙されないで！）
- 常にデータ型を使えば良い。

数値型とデータ型の matrix の違い

数値型

- コマンドでは addAttr -at matrix で作る。
- API では MFnMatrixAttribute で作る。
- API では MDataHandle::asMatrix() で値を得られる。
- トランスフォーメーション形式の情報は扱える。

データ型

- コマンドでは addAttr -dt matrix で作る。
- API では MFnTypedAttribute で作る。
- API では MDataHandle::data() で得た MObject を MFnMatrixData に通すことで値を得られる。
- トランスフォーメーション形式の情報 も扱える。

トランスフォーメーション情報

$$\begin{bmatrix} m_{00} & m_{01} & m_{02} & m_{03} \\ m_{10} & m_{11} & m_{12} & m_{13} \\ m_{20} & m_{21} & m_{22} & m_{23} \\ m_{30} & m_{31} & m_{32} & m_{33} \end{bmatrix}$$

4x4 行列

translate	jointOrient
rotate	rotateAxis
rotateOrder	rotatePivot
scale	rotatePivotTranslate
shear	scalePivot
inverseScale	scalePivotTranslate

トランスフォーメーション情報

- matrix データ型ではどちらの形式もサポート。
- スキンの Go to Bind Pose で、姿勢だけでなくアトリビュート値も完全に復元するのは、dagPose ノードにこの情報が保存されているから。mayaAscii をテキストエディタ等で見れば確認できる。

トランスフォーメーション情報の扱い方

- 保存情報のチェック
  - MFnMatrixData::isTransformation()
- MTransformationMatrix の取得と保存
  - MFnMatrixData::transformation() で取得。
  - MFnMatrixData::set() で保存。
  - joint 特有の情報は扱えないので注意。
- joint 特有の情報も含めた扱い
  - setAttr コマンドが唯一の保存手段。
  - MPlug::getSetAttrCmds() をパースすることが唯一の取得手段。
    ただし、入力接続が有る場合は結果を返さないので注意。
  - getAttr コマンドでの取得はサポートされていない。

スタティックとダイナミック

Static Attribute

- ノードタイプに備わっている（追加する）アトリビュート。
- ノードを生成すれば、それらのアトリビュートが最初から備わる。
- リリーズ済みのプラグインのノードに対して、後から追加することも容易にできる。削除（追加をやめること）は…やめた方が良い。

Dynamic Attribute

- ノード実体ごとに随時追加や削除するアトリビュート。
- 追加: addAttr コマンド
- 削除: deleteAttr コマンド

拡張アトリビュート

既存のノードタイプにスタティックアトリビュート追加する機能。

- Maya®2012 で追加された。
- addExtension コマンドで可能（addAttr と似ている）。

  ただし、addAttr と違って、その行為がシーンファイルに保存されない点に注意。つまり、そのシーンを開くためには、事前に addExtension を正しく実行することが求められる。

- API を使えば、プラグインに紐付けられる。 **おすすめ!!**

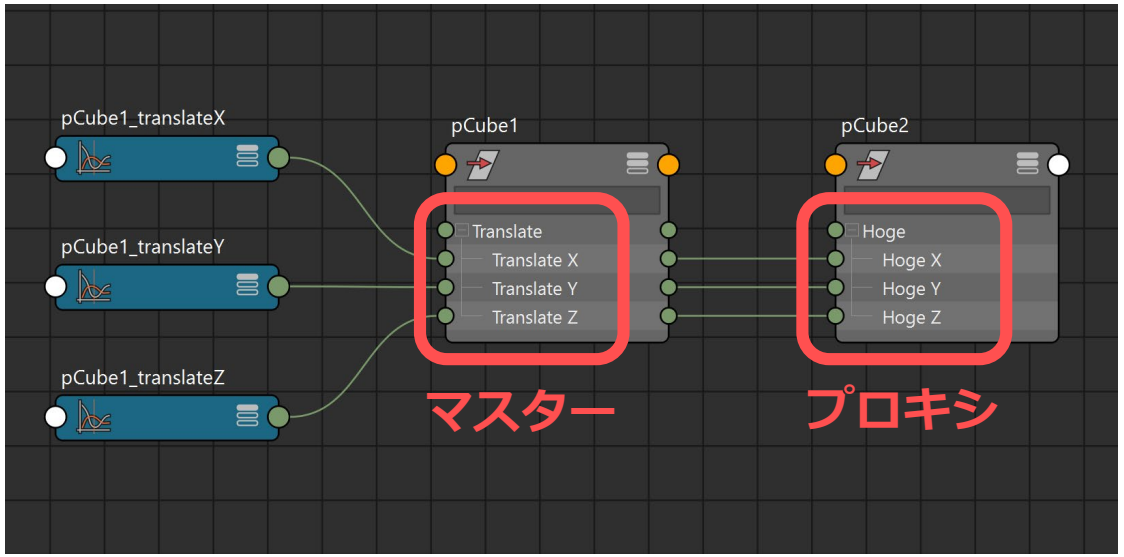
  MDGModifier::linkExtensionAttributeToPlugin()

  参考： https://qiita.com/ryusas/items/aa9edde49a9a6b41e8a2

プロキシアトリビュート

入力している他のアトリビュートと同じもののように振る舞う。

- Maya®2016.5 で追加された。
- 別のノードに代理のアトリビュートを作るような目的で使う。



プロキシはマスターの代理インタフェース。
プロキシに対するセット操作は
マスターに振り替えられるので、
実質どちらでも編集できる。

ディペンデンシーグラフ

DG 評価のしくみ

評価の方向 ～ Push と Pull

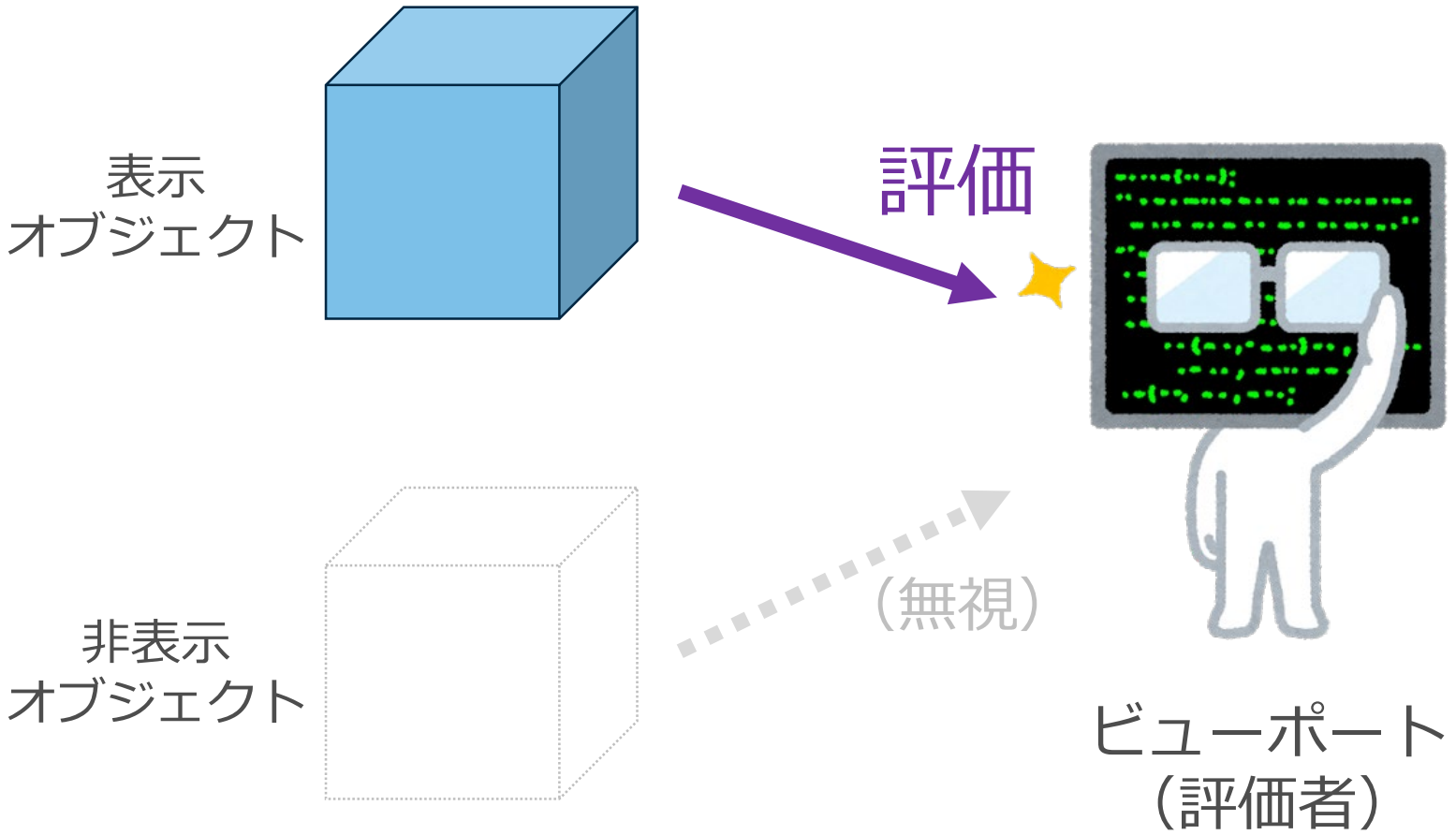
評価者（データを得たい者）からみて、

- むこうの都合で、むこうから、
  データを突っ込まれるのが Push

- 必要なときに、こちらから、
  データを引っ張り出すのが Pull

DG は Pull 評価

ビューの描画における Pull 評価



もしも、イベント駆動アーキテクチャだったら？

・ボタンが押された
・マウスが動いた

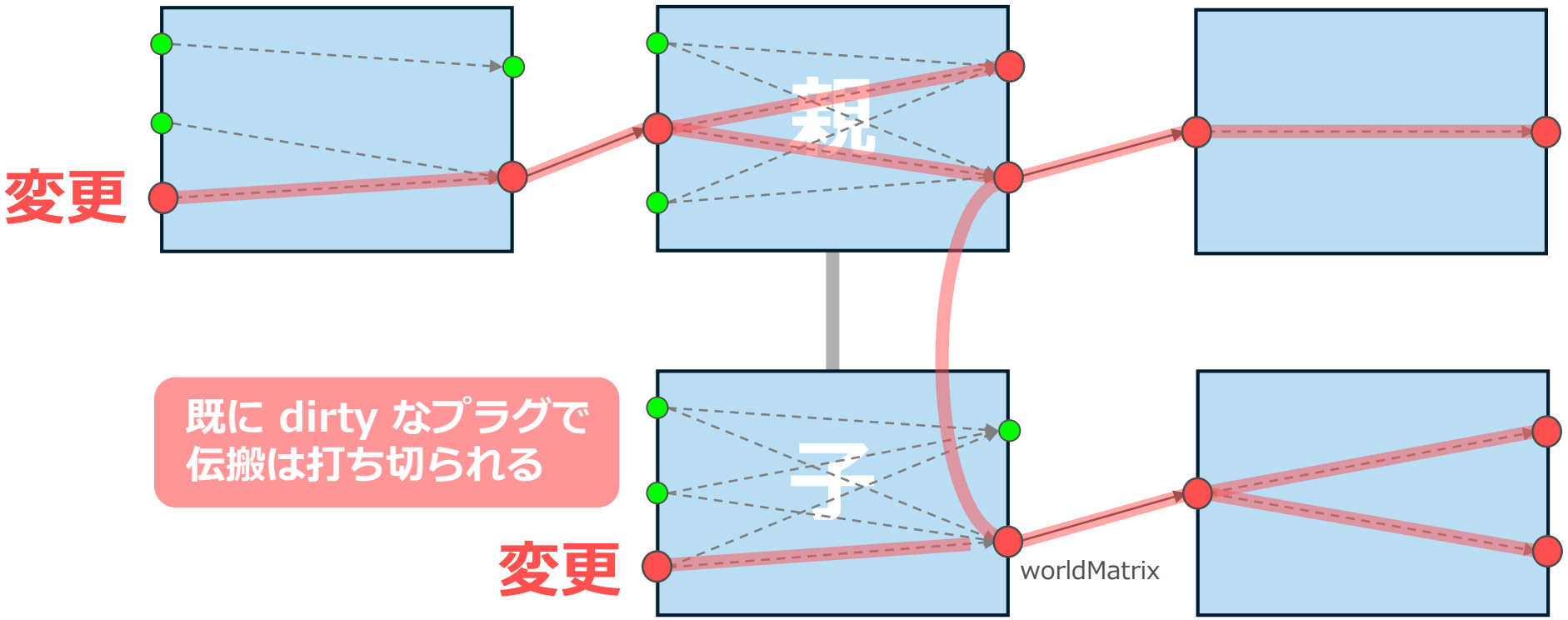
ノード実装に置き換えると

DG のしくみは
こうではない

Pull 評価でのノードの動作

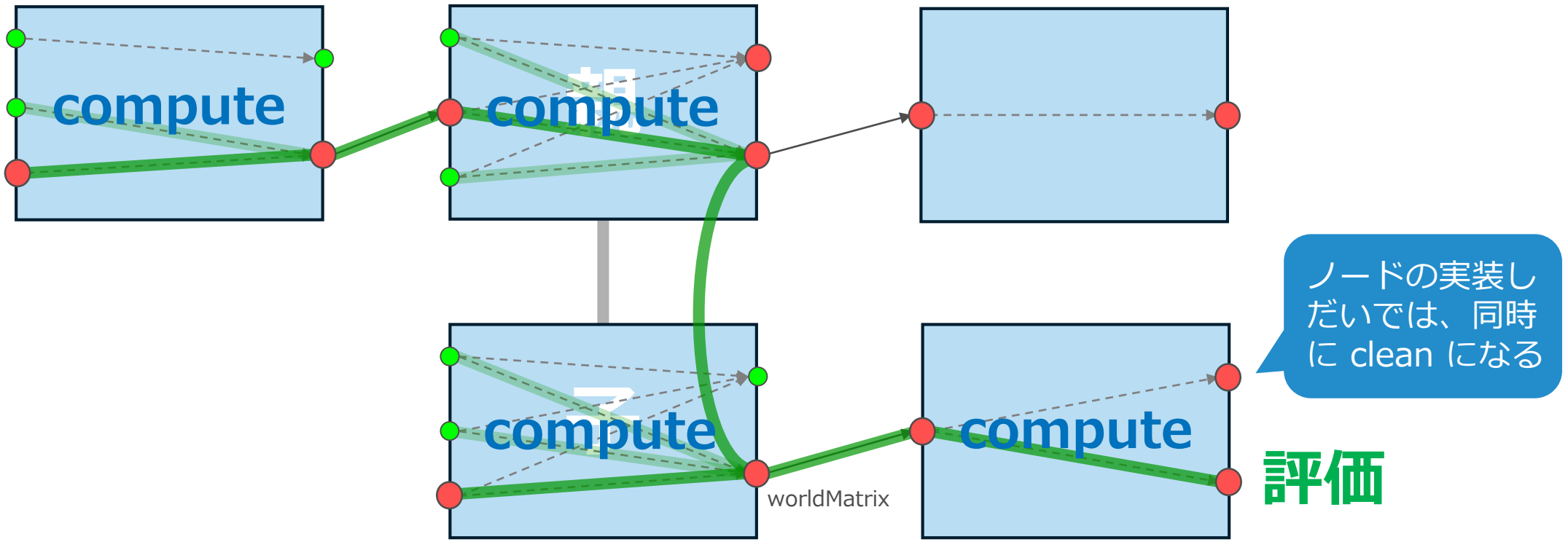
dirty 伝搬

どこかのプラグ値が変更されると、そのプラグは dirty となり、
それは依存関係に沿って下流に伝搬する。



プラグ評価と compute

評価されたプラグが dirty なら、そのノードの compute が呼ばれる。
さらに compute が上流を評価すると clean な箇所まで再帰的に呼ばれる。

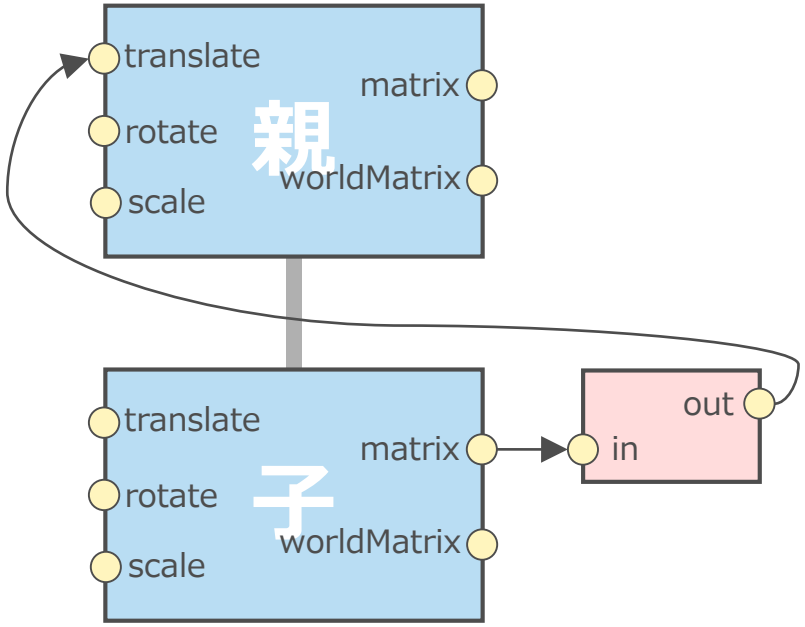


Pull = 遅延評価

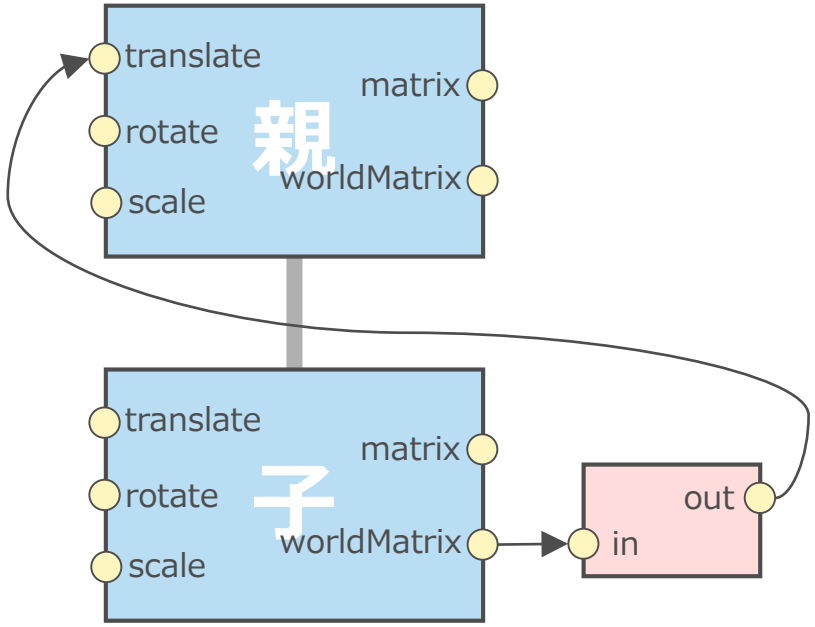
- 更新が必要な状態（dirty）になっても、
  実際に出力が必要になるときまで計算を行わない。
- 遅延評価 ともいう。
- このしくみによって 計算量の最適化 が実現されている。
  - 不要な計算は行わない。
  - 計算済みものは再利用。

cycleCheck コマンドの問題について

cycleCheck –dag で DAGの親子関係も考慮したチェックができるが、本当はサイクルでないものもサイクルとして検出されてしまう。

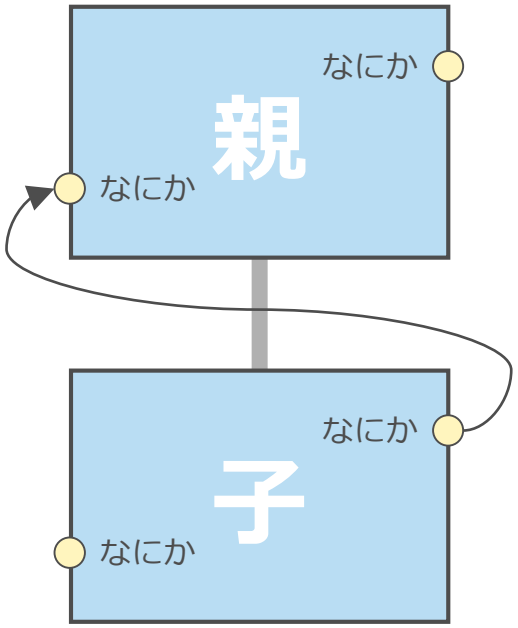


本当はサイクルでない



本当にサイクル

偽りのサイクルと、本当の見分け方



このようなものはすべてサイクルと検出される

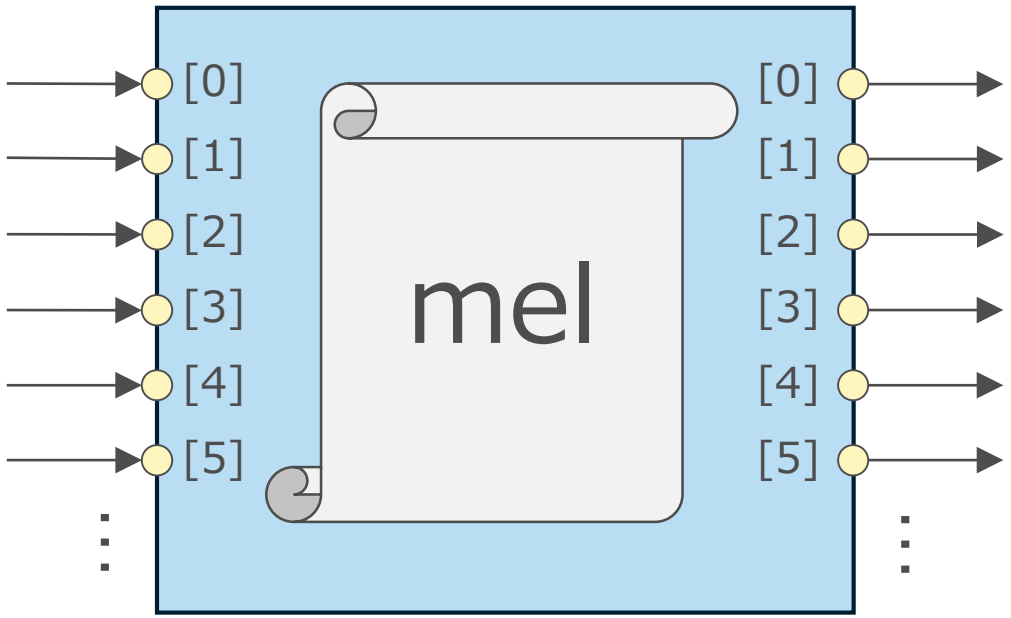
- 階層上の下位から上位へ向けてのコネクションが有ると、すべてサイクルと検出。
- 実際の Affects によるサイクルがあろうがなかろうが関係ない。何の依存もない追加アトリビュートでさえも同様。
  （コンストレイン 1 個でサイクル）
- 実際に評価されたときに出る cycleCheck の warning メッセージは正しい。それが出なければ問題ないということ。

ディペンデンシーグラフ

エクスプレッション

expression のしくみ

「式」記述機能というよりも、簡易的なノード開発機能ともいえる。
ノードの compute を、シーン埋め込みの mel コードで記述できる。



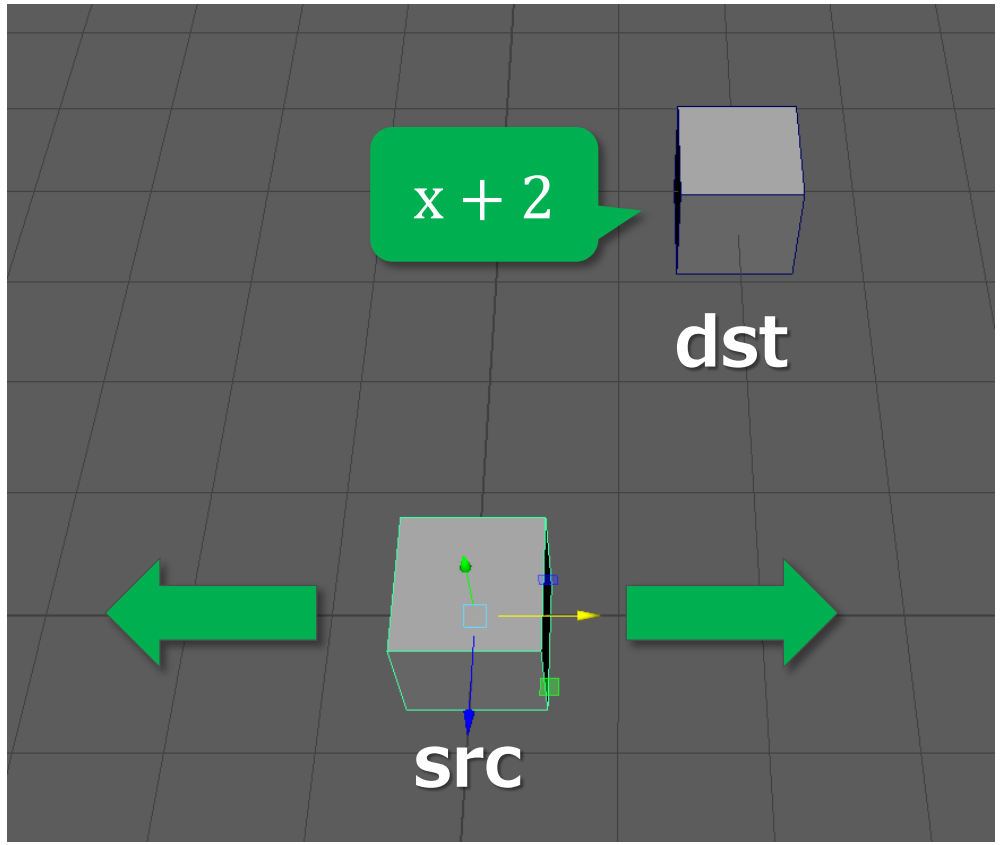
expression ノード

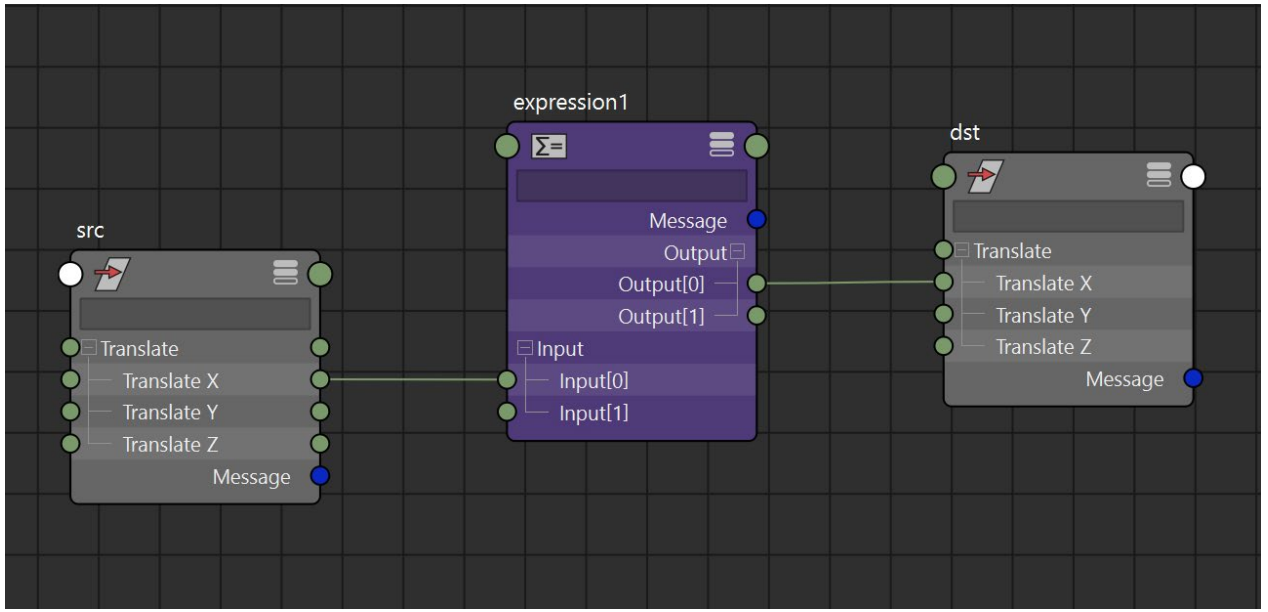
double 型マルチアトリビュートの
汎用入出力を備える。

アトリビュートは、
参照したものが入力に、
代入したものが出力に接続される。

出力すべてが入力すべてに依存する
Attribute Affects になる。

簡単な expression の例





expression コード

正しいコード

```
dst.tx = src.tx + 2;
```

代入(出力)　参照 (入力)

代入と参照はプレースホルダとなり、コネクションが正しく作成される。

誤ったコード

```
setAttr dst.tx (`getAttr src.tx` + 2);
```

コネクションは作成されない、ただの mel コード。

誤った expression による弊害

- Import や Reference 等でノード名が変わると壊れる。
- Evaluation=“Always” で、毎フレームの強制評価をさせることで動く。
- インタラクティブ操作では動かない。
- パラレル評価モードでは動かない。

Evaluation=“Always” とは

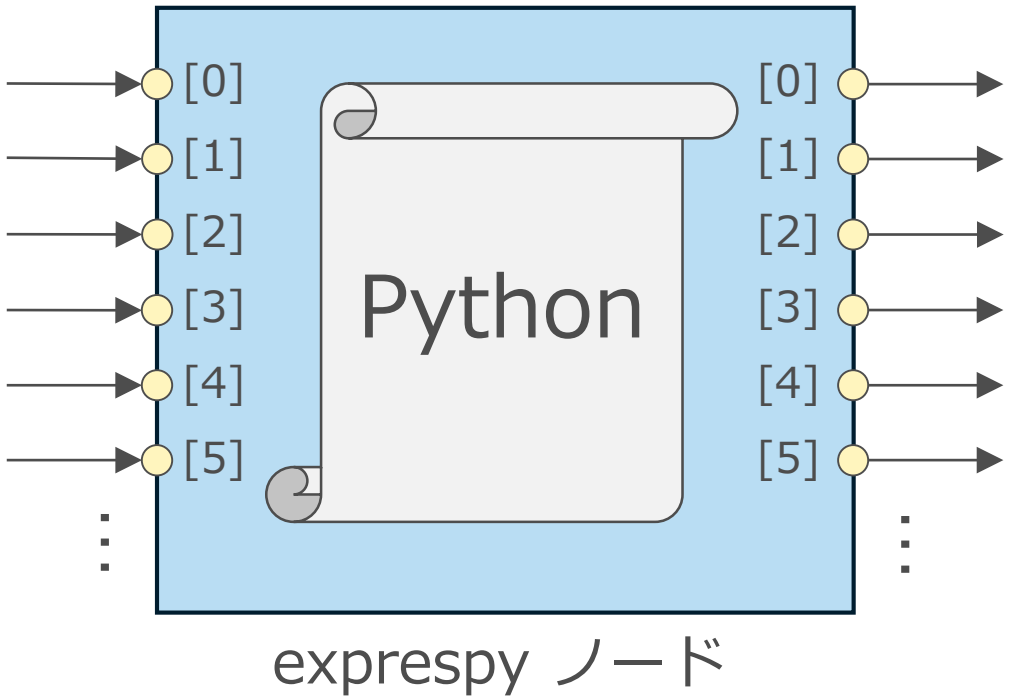
- expression の評価の有無にかかわらず、毎フレーム、必ず呼び出されるようになる。
- time 入力追加で依存は明確だが、出力の評価がなくても呼び出される（この例では出力は何も無かった！）。
- なるべく頼らない方が良いが、困ったことにデフォルトで有効。

問題なければ Evaluation は “On Demand” に設定すべき

exprespy のしくみ

オープンソースの python エクスプレッション・プラグイン。

https://github.com/ryusas/maya_exprespy



exprespy ノード

generic 型マルチアトリビュートの
汎用入出力を備える。

アトリビュートは、
参照したものが入力に、
代入したものが出力に接続される。

出力すべてが入力すべてに依存する
Attribute Affects になる。

exprespy が標準の expression と異なる点

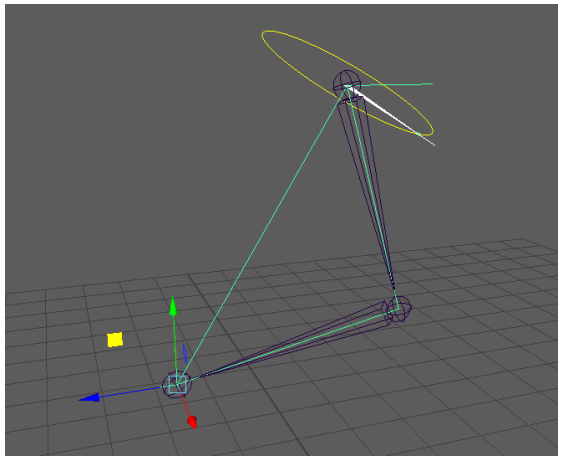
使用感は、標準の expression にとてもよく似ているが、
以下の点が異なる。

- 入出力は double 型ではなく generic 型。
  - ほとんど全ての型の入出力に対応。
  - 単位換算はサポートされず内部単位で入出力する（効率的）。
- Evaluation=“Always” の機能は無い。
  - time1.o を参照すれば、依存させることはできる。
  - 評価されていないのに強制的に呼び出させることはできない。

ディペンデンシーグラフ

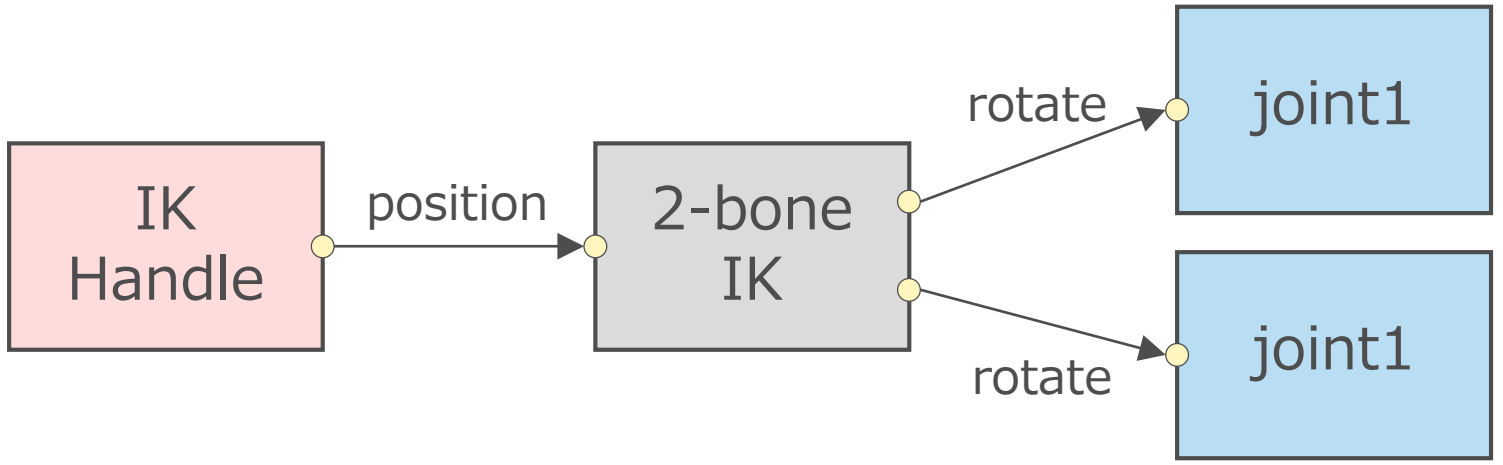
IK の DG 評価

IK の DG コネクションを想像する



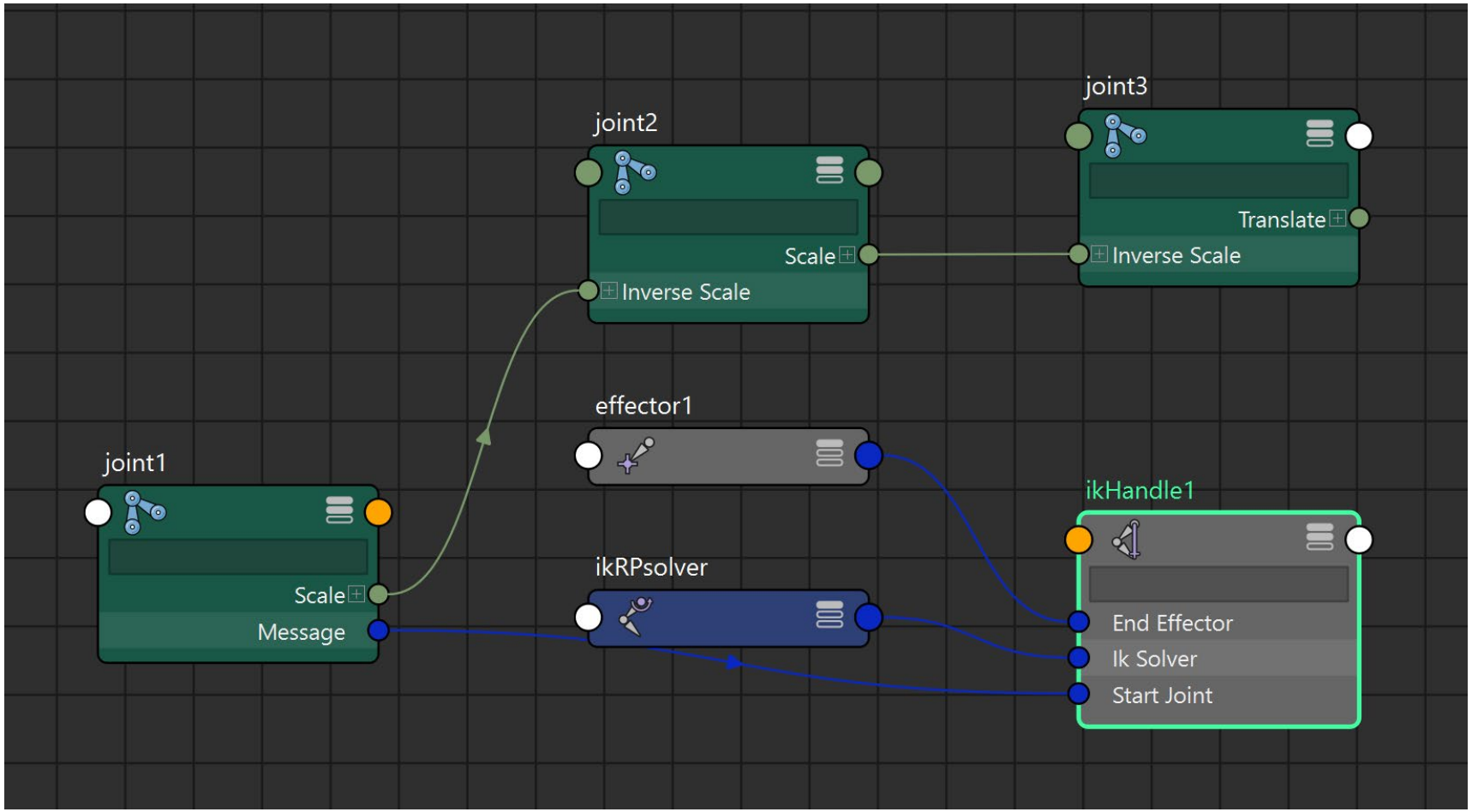
IK は、IK Handle の位置にジョイントチェーンの
先端が到達するように各ジョイントの回転を計算する。

簡略化した DG ネットワークを想像すると、
以下のようになるのではないか？



IK の DG コネクションの実際

…なっていなかった。



なぜ、これで IK が動くのか

- 入出力が無いのに！
  - そもそも、各 joint の rotate に何も入力されていない。
  - そもそも、IK Handle は何も出力していない。
- DG の基本ルールに則らないシステム
  - IK System がシーンを管理、IK Handle の更新に従って IK solver が呼び出される。
  - IK Handle の message コネクションを辿ると、DAGパス上のチェーンの始点と終点がわかる。
  - 入出力に関係なく joint にアクセスし、rotate に値をセット。

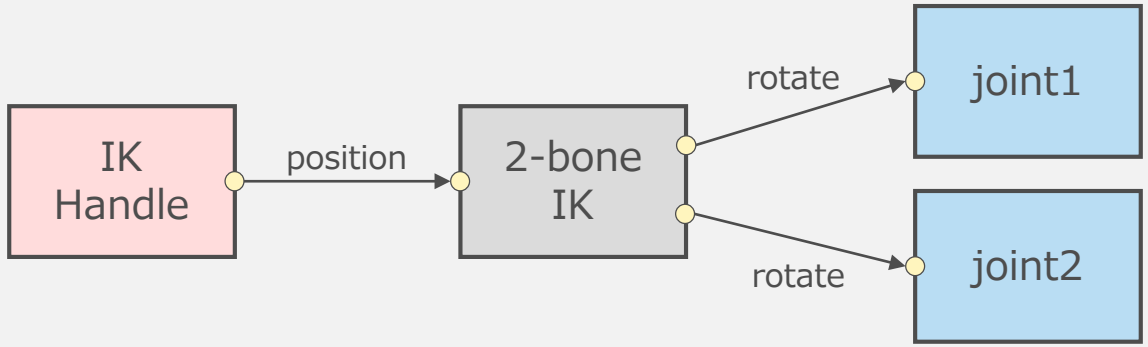
IK の依存関係

DG の基本

アトリビュートの依存関係を作るもの：

- コネクション
- Attribute Affects
- DAGノードの親子関係

IK の依存関係はこのルールに基づかないが、リグを組む際などは、普通に想像できる依存関係があるものと意識すれば概ね問題ない。



これまで示した特別なもの

- animCurve の time 入力は、何も接続しなければ、カレントタイムに暗黙に依存。
- expression には、毎フレーム強制評価するモードがある（しかもデフォルトで有効）。
- 依存関係も評価方法も「DGの基本ルール」に則っていないものがある（わかりやすいものとして IK を例に挙げた）。

DG の基本ルールに則らないものたち

本来の理想

- ノードは独立した関数
- 依存関係が明確

Pull 評価による
安定動作
（単純でわかりやすい）

実際は…

基本ルールに則ってシンプルに構築されていない複雑なものもある。
一概にはいえないが、シミュレーション系など。

特殊なものは、DG で特別な対応（実装）がされているように、
パラレルでも特別な対応がされることになる。

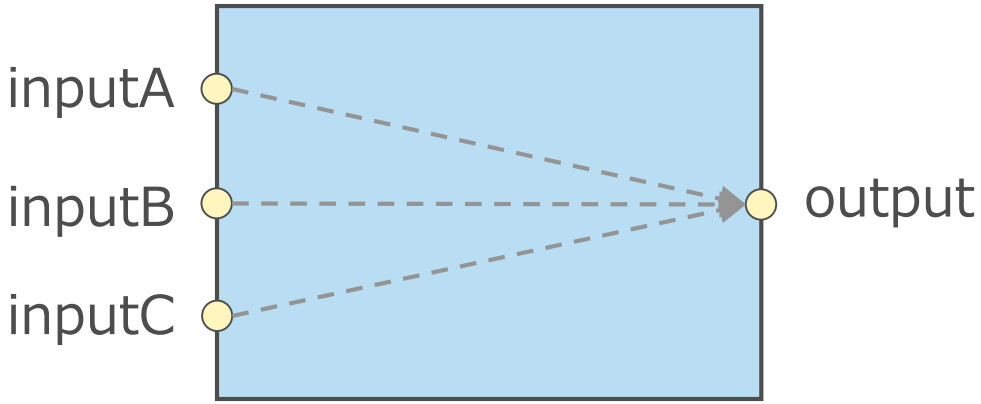
プラグインノード開発

シンプルなノード

simpleCalc ノードの仕様

output = (inputA + inputB) * inputC

output	inputA	inputB	inputC
double3 (doubleLinear)	double3 (doubleLinear)	double3 (doubleLinear)	double



ノードクラスの定義

```
class SimpleCalc : public MPxNode {
public:
    // ノードの計算処理。
    MStatus compute(const MPlug&, MDataBlock&);

    // ノードインスタンスの生成。
    static void* creator() { return new SimpleCalc; }

    // ノードタイプの初期化。
    static MStatus initialize();

    // ノードタイプ名とユニークID。
    static MString name;
    static MTypeId id;

    // アトリビュート。
    static MObject aInputA;
        static MObject aInputAX;
        static MObject aInputAY;
        static MObject aInputAZ;
    static MObject aInputB;
        static MObject aInputBX;
        static MObject aInputBY;
        static MObject aInputBZ;
    static MObject aInputC;
    static MObject aOutput;
        static MObject aOutputX;
        static MObject aOutputY;
        static MObject aOutputZ;
};
```

クラスインスタンスのライフサイクル

- ノードが生成されると、クラスのインスタンスがその数だけ生成される。
- アトリビュートが評価されると、インスタンスの compute メソッドが呼ばれる。
- ノードが削除されても undo の可能性がある限り、インスタンスは破棄されない。
- New Scene されるか undo キューが上限に達した時に、インスタンスが破棄される。

エントリーポイント関数の実装

```
MTypeId SimpleCalc::id   = 0x00066601;  // Local Test Area: 0x00000000 - 0x0007ffff
MString SimpleCalc::name = "simpleCalc";

// プラグインがロードされた時に呼ばれるグローバル関数。
MStatus initializePlugin(MObject obj)
{
    MFnPlugin plugin(obj, "ryusas", "1.0.0", "Any");

    return plugin.registerNode(
        SimpleCalc::name, SimpleCalc::id,
        SimpleCalc::creator, SimpleCalc::initialize,
        MPxNode::kDependNode
    );
}

// プラグインがアンロードされた時に呼ばれるグローバル関数。
MStatus uninitializePlugin(MObject obj)
{
    MFnPlugin plugin(obj);
    plugin.deregisterNode(SimpleCalc::id);
    return MS::kSuccess;
}
```

initialize() の実装

```
MStatus SimpleCalc::initialize()
{
    MFnNumericAttribute fnNumeric;
    MFnUnitAttribute fnUnit;

    // inputA
    aInputAX = fnUnit.create("inputAX", "iax", MFnUnitAttribute::kDistance, 0.);
    aInputAY = fnUnit.create("inputAY", "iay", MFnUnitAttribute::kDistance, 0.);
    aInputAZ = fnUnit.create("inputAZ", "iaz", MFnUnitAttribute::kDistance, 0.);
    aInputA = fnNumeric.create("inputA", "ia", aInputAX, aInputAY, aInputAZ);
    CHECK_MSTATUS_AND_RETURN_IT(addAttribute(aInputA));

    // inputB
    aInputBX = fnUnit.create("inputBX", "ibx", MFnUnitAttribute::kDistance, 0.);
    aInputBY = fnUnit.create("inputBY", "iby", MFnUnitAttribute::kDistance, 0.);
    aInputBZ = fnUnit.create("inputBZ", "ibz", MFnUnitAttribute::kDistance, 0.);
    aInputB = fnNumeric.create("inputB", "ib", aInputBX, aInputBY, aInputBZ);
    CHECK_MSTATUS_AND_RETURN_IT(addAttribute(aInputB));

    // inputC
    aInputC = fnNumeric.create("inputC", "ic", MFnNumericData::kDouble, 1.);
    CHECK_MSTATUS_AND_RETURN_IT(addAttribute(aInputC));

    // output
    aOutputX = fnUnit.create("outputX", "ox", MFnUnitAttribute::kDistance, 0.);
    aOutputY = fnUnit.create("outputY", "oy", MFnUnitAttribute::kDistance, 0.);
    aOutputZ = fnUnit.create("outputZ", "oz", MFnUnitAttribute::kDistance, 0.);
    aOutput = fnNumeric.create("output", "o", aOutputX, aOutputY, aOutputZ);
    fnNumeric.setWritable(false);
    fnNumeric.setStorable(false);
    CHECK_MSTATUS_AND_RETURN_IT(addAttribute(aOutput));

    // アトリビュート間の依存関係を作成。
    attributeAffects(aInputA, aOutput);
    attributeAffects(aInputB, aOutput);
    attributeAffects(aInputC, aOutput);

    return MS::kSuccess;
}
```

compute() の実装

```
MStatus SimpleCalc::compute(const MPlug& plug, MDataBlock& block)
{
    // 評価されているプラグのチェック。
    const MPlug evalPlug = plug.isChild() ? plug.parent() : plug;
    if (evalPlug != aOutput)
        return MS::kUnknownParameter;

    // 入力値の取得。
    const MVector& a = block.inputValue(aInputA).asVector();
    const MVector& b = block.inputValue(aInputB).asVector();
    const double& c = block.inputValue(aInputC).asDouble();

    // 計算。
    MVector res = (a + b) * c;

    // 出力値のセット。setClean も兼ねる。
    block.outputValue(aOutput).set(res);

    return MS::kSuccess;
}
```

output だけでなく outputX 等で評価されることもあるので、どちらにも対応している。

inputValue() でハンドルを得ることで上流の評価も促される。

outputValue() でのハンドル取得は上流の評価を促さないので出力値のセットに適している。

simpleCalc にひと工夫加える

output = (inputA + inputB) * inputC

これを「重たい計算」と仮定

- compute 時、 (A + B) をキャッシュする。
- compute 時、キャッシュ有れば (A + B) を計算しない。
- A や B が dirty になったとき、キャッシュを破棄。
- C が dirty になっただけならキャッシュが再利用可能。

MPxNode::setDependentsDiry() について

Attribute Affects を動的に設定することができるしくみ。

- プラグが dirty になったとき即座に呼ばれる。
- それに依存するプラグのリストを返すことで Affects を定義。

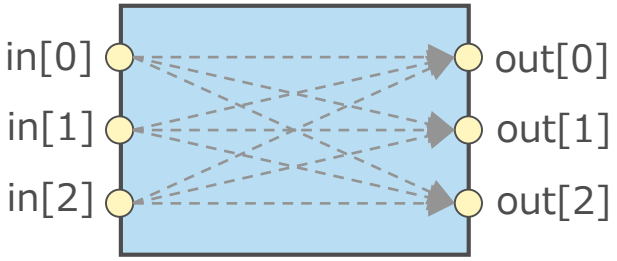
今回は、単なる dirty コールバックとして利用

計算の中間値をキャッシュするには様々な手法や考え方があります。
詳しくは、以下のセッション資料を参照してください。

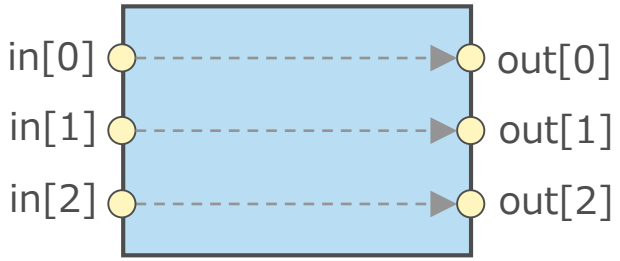
CEDEC2018: ノードがわかるとMaya®がわかる！エンジニアやTAのためのMaya®ノード開発入門
https://cedil.cesa.or.jp/cedil_sessions/view/1849

動的な Attribute Affects とは

- マルチアトリビュートの要素（プラグ）ごとの依存関係



静的な Affects では
全要素に依存してしまう



setDependentsDirty では
どのような依存関係でも作れる

- 後から追加される想定のダイナミックアトリビュートの依存関係
- ~~条件によって変化する依存関係~~

Affects の「追加」は問題ないが「変化」はやめた方がよい。
パラレル評価で動作させることが困難になる。

SimpleCalc クラスの拡張

```
class SimpleCalc : public MPxNode {
    MVector cachedVal; //< キャッシュされた値。
    double isCacheValid; //< キャッシュが有効かどうか。

    void doExpensiveCalc(MDataBlock&); //< (A + B) の計算。

public:
    SimpleCalc() : isCacheValid(false) {}
    MStatus setDependentsDirty(const MPlug&, MPlugArray&);
```

(A + B) の計算：

```
void SimpleCalc::doExpensiveCalc(MDataBlock& block)
{
    const MVector& a = block.inputValue(aInputA).asVector();
    const MVector& b = block.inputValue(aInputB).asVector();
    cachedVal = a + b;
    isCacheValid = true;
}
```

setDependentsDirty() と compute() の実装

```
MStatus SimpleCalc::setDependentsDirty(const MPlug& plug, MPlugArray&)
{
    const MPlug dirtyPlug = plug.isChild() ? plug.parent() : plug;
    if (dirtyPlug == aInputA || dirtyPlug == aInputB) {
        isCacheValid = false;
    }
    return MS::kSuccess;
}
```

A か B が dirty になったとき
キャッシュを無効化。

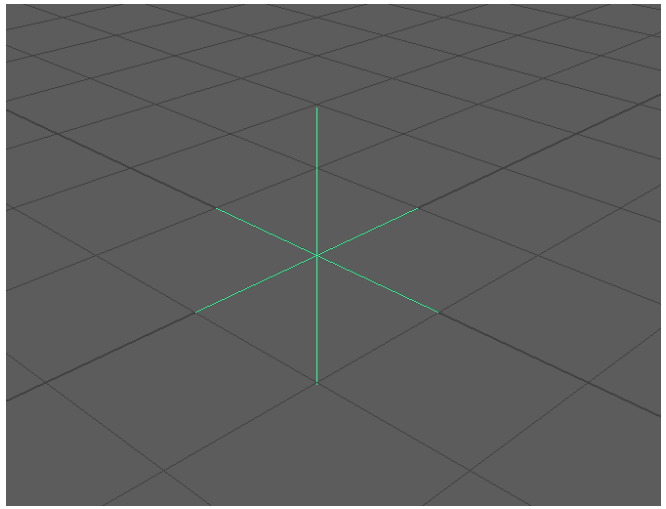
```
MStatus SimpleCalc::compute(const MPlug& plug, MDataBlock& block)
{
    // 評価されているプラグのチェック。
    const MPlug evalPlug = plug.isChild() ? plug.parent() : plug;
    if (evalPlug != aOutput)
        return MS::kUnknownParameter;

    // キャッシュの利用。
    if (! isCacheValid)
        doExpensiveCalc(block);

    // 残りの計算。
    MVector res = cachedVal * block.inputValue(aInputC).asDouble();

    // 出力値のセット。setClean も兼ねる。
    block.outputValue(aOutput).set(res);

    return MS::kSuccess;
}
```

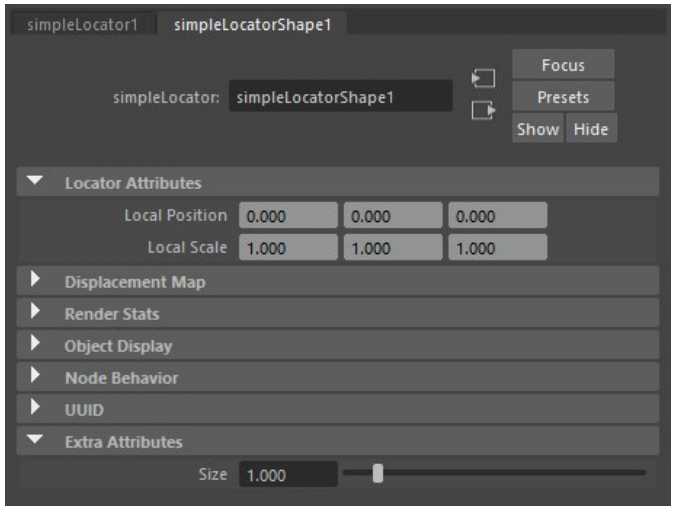
キャッシュが無効なら (A + B) を計算する。

プラグインノード開発

ロケータ

simpleLocator ノードの仕様





- 単なる十字形状。
- locator 標準アトリビュート localPosition と localScale をサポート。
- 追加アトリビュートの scale でサイズ調整。
- まずは Legacy Viewport のみサポート。

実装方針

- compute の実装は不要。
- Legacy Viewport 用 draw を実装。
- 描画のたびにアトリビュートにアクセスすると非効率なので、アトリビュート値をメンバ変数にキャッシュする。
- setDependentsDirty で dirty 監視し、キャッシュを破棄。

ノードクラスの定義

```
class SimpleLocator : public MPxLocatorNode {
    // 頂点データ。
    static const float3 Vertices[];
    static const unsigned NumberOfVertices;

    // 描画に反映させるアトリビュート値のキャッシュ。
    float3 translate;
    float3 scale;
    bool valueDirty;

    // アトリビュート値のキャッシュを更新。
    void updateValues();

public:
    SimpleLocator() : valueDirty(true) {}

    // dirty 時の処理。
    MStatus setDependentsDirty(const MPlug&, MPlugArray&);

    // Legacy Viewport の描画処理。
    void draw(M3dView&, const MDagPath&, M3dView::DisplayStyle, M3dView::DisplayStatus);

    // ノードインスタンスの生成。
    static void* creator() { return new SimpleLocator; }

    // ノードタイプの初期化。
    static MStatus initialize();

    // ノードタイプ名とユニークID。
    static MString name;
    static MTypeId id;

    // アトリビュート。
    static MObject aSize;
};
```

エントリーポイント関数の実装

```
MTypeId SimpleLocator::id    = 0x00066602;  // Local Test Area: 0x00000000 - 0x0007ffff
MString SimpleLocator::name = "simpleLocator";

// プラグインがロードされた時に呼ばれるグローバル関数。
MStatus initializePlugin(MObject obj)
{
    MFnPlugin plugin(obj, "ryusas", "1.0.0", "Any");

    return plugin.registerNode(
        SimpleLocator::name, SimpleLocator::id,
        SimpleLocator::creator, SimpleLocator::initialize,
        MPxNode::kLocatorNode
    );
}

// プラグインがアンロードされた時に呼ばれるグローバル関数。
MStatus uninitializePlugin(MObject obj)
{
    MFnPlugin plugin(obj);
    plugin.deregisterNode(SimpleLocator::id);
    return MS::kSuccess;
}
```

initialize() の実装

```
MStatus SimpleLocator::initialize()
{
    MFnNumericAttribute fnNumeric;

    // size
    aSize = fnNumeric.create("size", "sz", MFnNumericData::kDouble, 1.);
    fnNumeric.setSoftMin(0.);
    fnNumeric.setSoftMax(10.);
    CHECK_MSTATUS_AND_RETURN_IT(addAttribute(aSize));

    return MS::kSuccess;
}

MObject SimpleLocator::aSize;

const float3 SimpleLocator::Vertices[] = {
    { -1.0f,  0.0f,  0.0f },
    {  1.0f,  0.0f,  0.0f },
    {  0.0f, -1.0f,  0.0f },
    {  0.0f,  1.0f,  0.0f },
    {  0.0f,  0.0f, -1.0f },
    {  0.0f,  0.0f,  1.0f },
};
const unsigned SimpleLocator::NumberOfVertices = sizeof(SimpleLocator::Vertices) / sizeof(float3);
```

draw() の実装

```
void SimpleLocator::draw(
    M3dView& view, const MDagPath&, M3dView::DisplayStyle, M3dView::DisplayStatus)
{
    updateValues();

    view.beginGL();

    glBegin(GL_LINES);
    for (unsigned i=0; i<NumberOfVertices; ++i) {
        const float3& v = Vertices[i];
        glVertex3f(
            v[0] * scale[0] + translate[0],
            v[1] * scale[1] + translate[1],
            v[2] * scale[2] + translate[2]
        );
    }
    glEnd();

    view.endGL();
}
```

描画情報キャッシュを必要に応じて更新。

古い OpenGL で描画。

描画用アトリビュートキャッシュの実装

```
MStatus SimpleLocator::setDependentsDirty(const MPlug& plug, MPlugArray&)
{
    const MPlug dirtyPlug = plug.isChild() ? plug.parent() : plug;
    if (dirtyPlug == aSize || dirtyPlug == localPosition || dirtyPlug == localScale) {
        valueDirty = true;
    }
    return MS::kSuccess;
}

void SimpleLocator::updateValues()
{
    if (! valueDirty)
        return;

    MDataBlock block(forceCache());
    const double3& trn = block.inputValue(localPosition).asDouble3();
    const double3& scl = block.inputValue(localScale).asDouble3();
    const double& size = block.inputValue(aSize).asDouble();

    translate[0] = static_cast<float>(trn[0]);
    translate[1] = static_cast<float>(trn[1]);
    translate[2] = static_cast<float>(trn[2]);
    scale[0] = static_cast<float>(scl[0] * size);
    scale[1] = static_cast<float>(scl[1] * size);
    scale[2] = static_cast<float>(scl[2] * size);
    valueDirty = false;
}
```

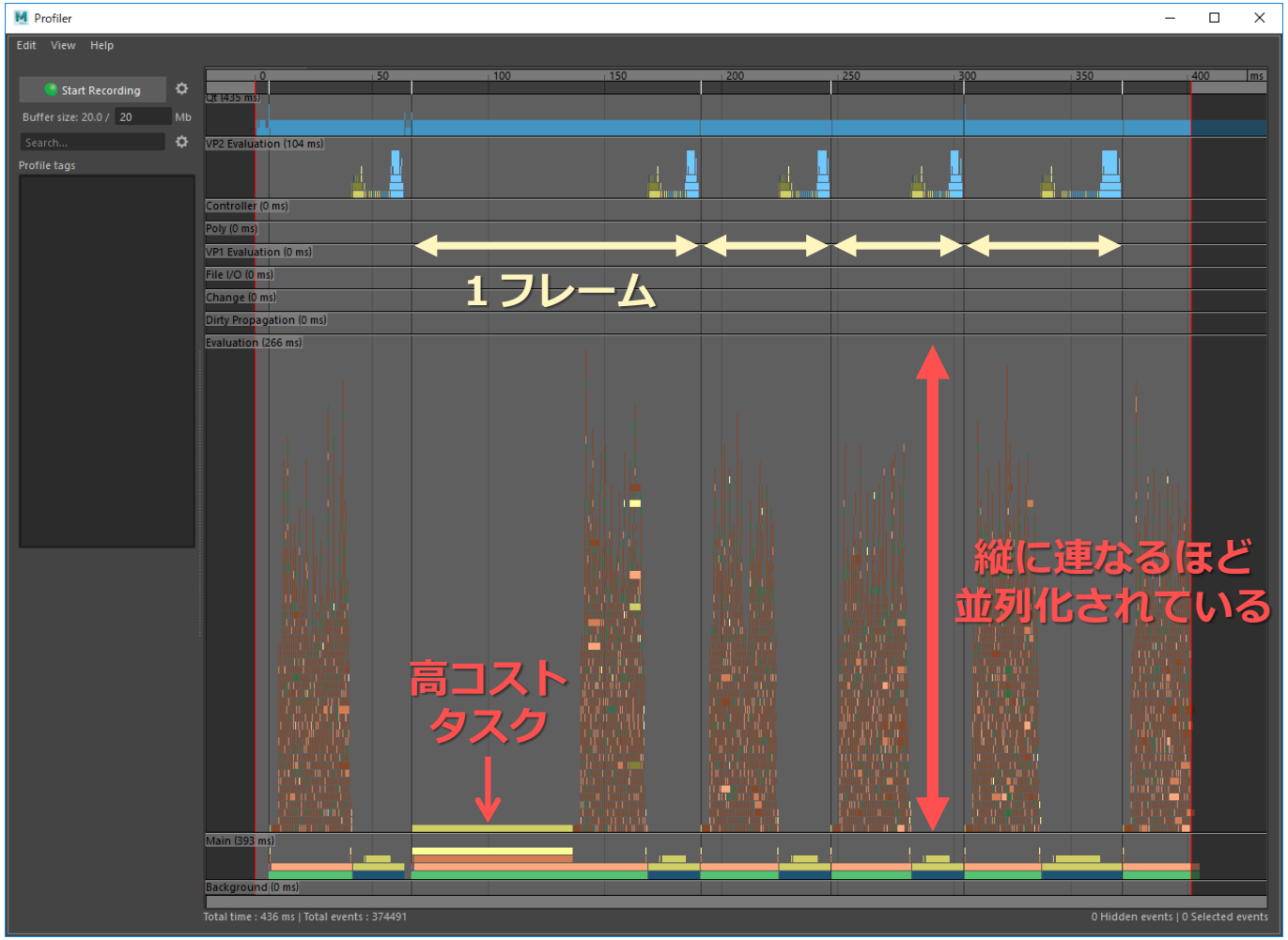
dirty になったとき
描画情報キャッシュを
無効化。

描画情報キャッシュが無効な
場合のみ、アトリビュートか
ら読み取って更新する。

プラグインノード開発

プロファイリング

Profiler



アニメーションプレイバックの
タスクをキャプチャし、
時間軸に沿って表示する。

- タスクカテゴリごとコスト配分をチェック。
- 効率よく並列化されているかをチェック。
- ボトルネックを見つける。

パラレル評価の分析に
特に向いている

dgtimer

DG の詳細なプロファイリング。DG 以外のモードでは動かない。
おおむね、以下のように実行する。

アニメーション１周分のキャプチャ：

```
dgtimer -reset;
dgtimer -on;
play -wait;
dgtimer -off;
dgtimer -o "D:/query.txt" -show compute -show dirty -show fetch –q;
```

１フレームのキャプチャ：

```
proc stopTimer() {
    dgtimer -off;
    dgtimer -o "D:/query.txt" -show compute -show dirty -show fetch -q;
}
dgtimer -reset;
dgtimer -on;
currentTime 1;
evalDeferred -lp "stopTimer";
```

dgtimer: メトリック

以下のメトリック（測定するもの）ごとの集計がレポートされる。

callback

MMessage 派生クラスによるコールバック関数の負荷。

compute

compute() メソッドの負荷。

dirty

dirty 伝搬の負荷。

draw

描画処理の負荷。

fetch

Pull 評価（上流から値を引き出す処理）の負荷。

dgtimer: 出力例

```
======================================
MAYA DEPENDENCY GRAPH TIMING INFORMATION
======================================

SECTION 1: Global timing information:

1.1 Global timing modes:
                Global timing is currently : OFF
                Global tracing is currently: OFF

1.2 Process time since last reset:
                Elapsed real time     : 70.8414 sec
                Elapsed user time     : 0 sec
                Elapsed system time   : 0 sec

1.3 Summary of each metric for all nodes:
                Real time in callbacks                      : 5.32619 sec [6.68553e+06 operations]
                Real time in compute                        : 36.1123 sec [4.32641e+06 operations]
                Real time in dirty propagation              : 35.6731 sec [8.42386e+06 operations]
                Real time in drawing                        : 9.70844 sec [2.48474e+06 operations]
                Real time fetching data from plugs          : 13.3573 sec [1.22812e+07 operations]

1.4 Breakdown of select metrics in greater detail:
                Breakdown of "callback" metric in greater detail:
                                Real time in callbacks registered via API    : 0.454898 sec [865265 operations]
                                Real time in callbacks NOT registered via API: 4.87129 sec [5.82027e+06 operations]
                Breakdown of "compute" metric in greater detail:
                                Real time in compute invoked from callback   : 0 sec [0 operations]
                                Real time in compute not invoked via callback: 36.1123 sec [4.32641e+06 operations]
                Plug caching performance:
                                Ratio of computes to fetches                : 0.3523
                                Ratio of time in compute to (compute + fetch): 0.73

SECTION 2. Per-node timing information:

        COMPUTE                                         DIRTY                       FETCH
Rank ON       Self Percent Cumulative  Inclusive    Count       Self  Inclusive    Count       Self  Inclusive    Count Type   Node
-------------------------------------------------------------------------------------------------------------------------------------
    1 N     0.1302   0.36%     0.1302     0.1724      244     0.0125     0.0174     6100     0.0022     0.1746      244 skinCluster human:skinCluster1
    2 N     0.1280   0.35%     0.2582     0.1697      244     0.0109     0.0134     6100     0.0016     0.1713      244 skinCluster human5:skinCluster1
    3 N     0.1270   0.35%     0.3852     0.1678      244     0.0111     0.0137     6100     0.0017     0.1696      244 skinCluster human6:skinCluster1
    4 N     0.1264   0.35%     0.5116     0.1659      244     0.0107     0.0131     6100     0.0017     0.1676      244 skinCluster human11:skinCluster1
    5 N     0.1256   0.35%     0.6372     0.1658      244     0.0108     0.0135     6100     0.0017     0.1675      244 skinCluster human3:skinCluster1
    6 N     0.1256   0.35%     0.7628     0.1653      244     0.0109     0.0136     6100     0.0017     0.1670      244 skinCluster human7:skinCluster1
    7 N     0.1256   0.35%     0.8883     0.1660      244     0.0111     0.0140     6100     0.0019     0.1679      244 skinCluster human2:skinCluster1
    8 N     0.1254   0.35%     1.0138     0.1657      244     0.0116     0.0148     6100     0.0018     0.1676      244 skinCluster human1:skinCluster1
    9 N     0.1252   0.35%     1.1390     0.1650      244     0.0111     0.0138     6100     0.0018     0.1668      244 skinCluster human4:skinCluster1
```

サマリー

詳細

dgtimer: レポートのセクション

SECTION 1

全体の実行時間などのサマリー。
DG がどのくらい効率的に動いているか分析。

SECTION 2

ノードごとにメトリックの負荷を集計。負荷の高い順にソート。

- 実行カウント
- 処理時間
- パーセンテージ

SECTION 3

コールバックごとの処理時間などの詳細。

dgtimer: プロファイリング結果の考察

Biped リグ 12 体 DG 評価 1 フレームの計測結果：

```
1.3 Summary of each metric for all nodes:
    Real time in callbacks              : 0.00909229 sec [25384 operations]
    Real time in compute                : 0.0520244 sec [6024 operations]
    Real time in dirty propagation      : 0.0549755 sec [12372 operations]
    Real time in drawing                : 0.019266 sec [6752 operations]
    Real time fetching data from plugs  : 0.0384609 sec [17208 operations]
```

compute が純粋な「ノードの計算負荷」で、
dirty と fetch が「DG のシステム負荷」と考えられる。

「DG のシステム負荷」が大きな割合を占める

dgtimer: 結果からみる DG の最適化

「DG のシステム負荷」が高い

リグのコントローラーなどは、だいたいこのような結果になる。
（個々のノードの計算負荷が小さく、ネットワークが複雑になりがち）

効果的な最適化

- コネクション数を減らす（dirty 伝搬の回数を直接的に減らす）。

  x,y,z 個別の接続より double3 での接続が有利など。

- ローカル空間リギング（ワールド空間の fetch を少なくする）。

ただし、このアプローチは、あくまでも DG 評価の最適化。
Parallel 評価では dirty も fetch も無いので違ってくる。
とはいえ、Parallel で不利にもならないはずではある。

パラレル評価

パラレル評価のしくみ

公式ドキュメント

パラレル評価とキャッシュプレイバックについては、本体のマニュアルとは別配布の公式ドキュメントで詳しく解説されている。
https://knowledge.autodesk.com/support/maya/learn-explore/caas/simplecontent/content/using-parallel-maya.html

- Using Parallel Maya®
  - 2019 [PDF] [HTML]
  - 2018 [PDF] [HTML]
  - 2017 [PDF] [HTML]
  - 2016.5 [PDF] [HTML]
  - 2016 [PDF]
- Maya® Cached Playback
  - 2019 [PDF] [HTML]

正確な情報を得るには、当スライドの内容は参考程度に留め、必ず公式ドキュメントを確認するようにしてください。

自作リグでパラレル評価を使いこなすには

できるだけ…

expression を使わない

- mel はスレッドセーフではない。
- どうしても使うなら Evaluation=“On Demand” で。

Python ノードを使わない（exprespy および Python プラグイン）

- Python は GIL による制限で並列実行できない。C++ 移殖を検討すべき。

IK を使わない

- ほぼ問題ないが、やや効率が悪い。

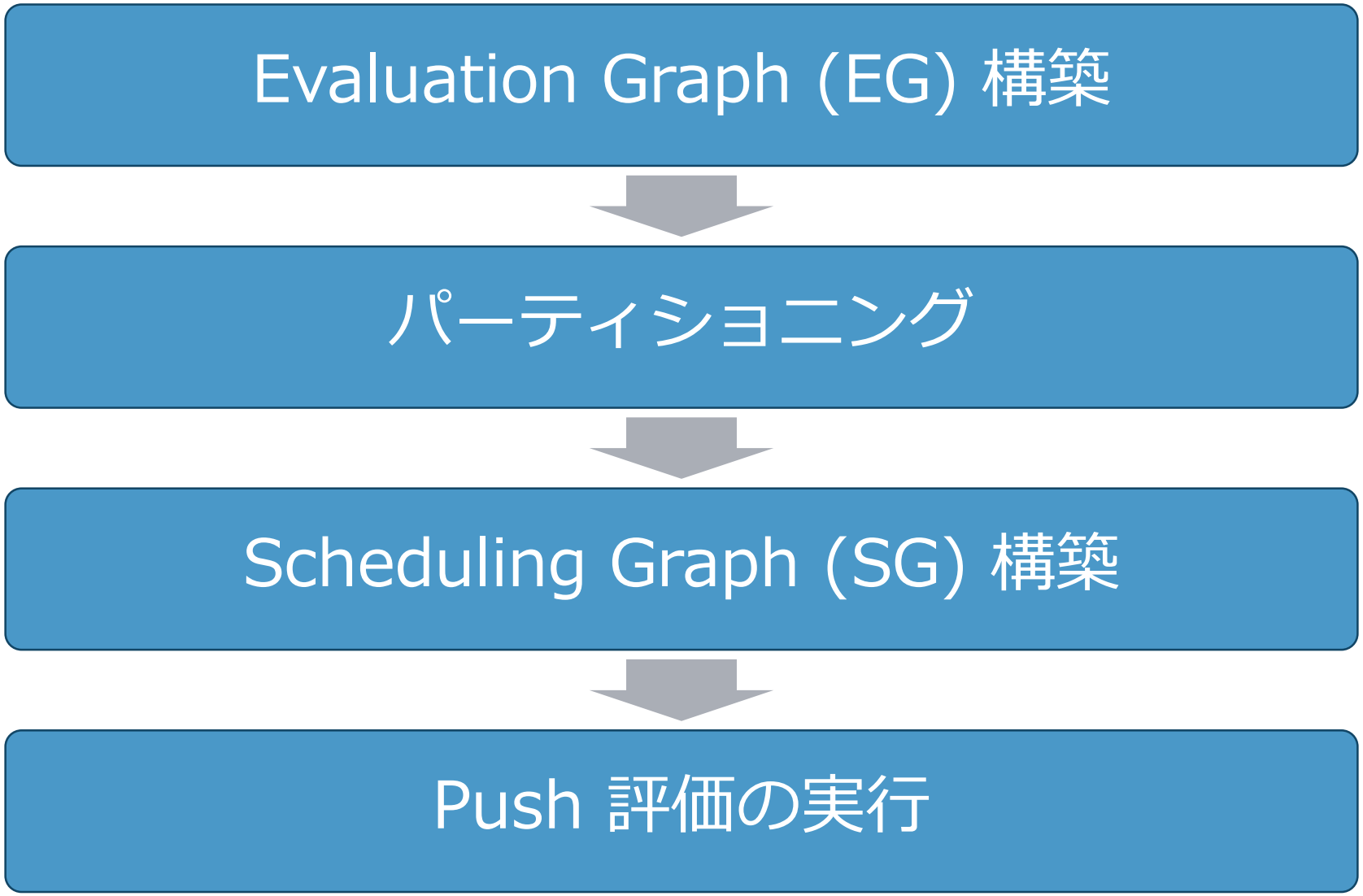
コンストレインを使わない

- やや効率が悪いが、簡単な改善方法はある（後述）。

パラレル評価の概要

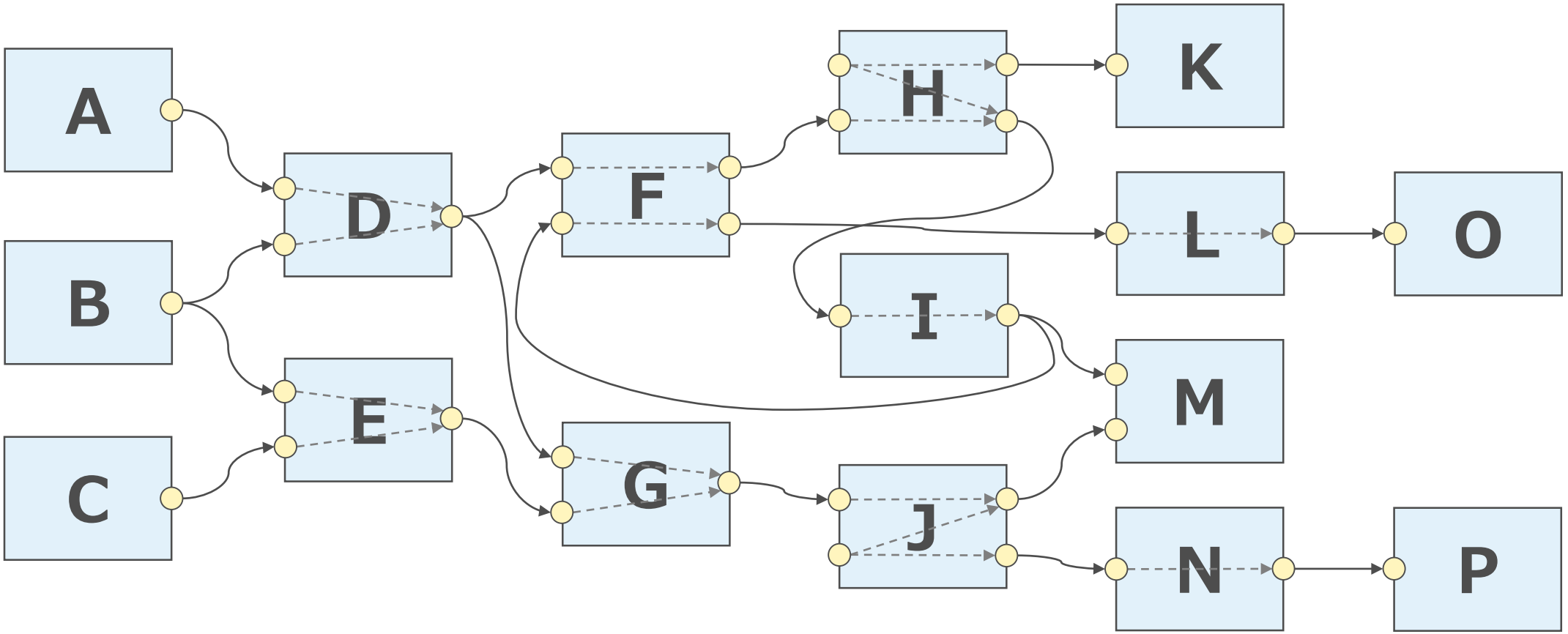
- アニメーションしているブロックのみが対象の「Push 評価」。
- Evaluation Manager (EM) がシーンを監視、DG を基に、パラレルな Push 評価タスクスケジュールが作られる。
- スケジュールに基づいて並列実行される。
- シーン構造が変更されたら、スケジュールも作り直される。
- 効率化や複雑な対応のために、スケジューリングやタスク実行は、様々な Evaluator によってオーバーライドされる。

EM による評価工程



Dependency Graph (DG)

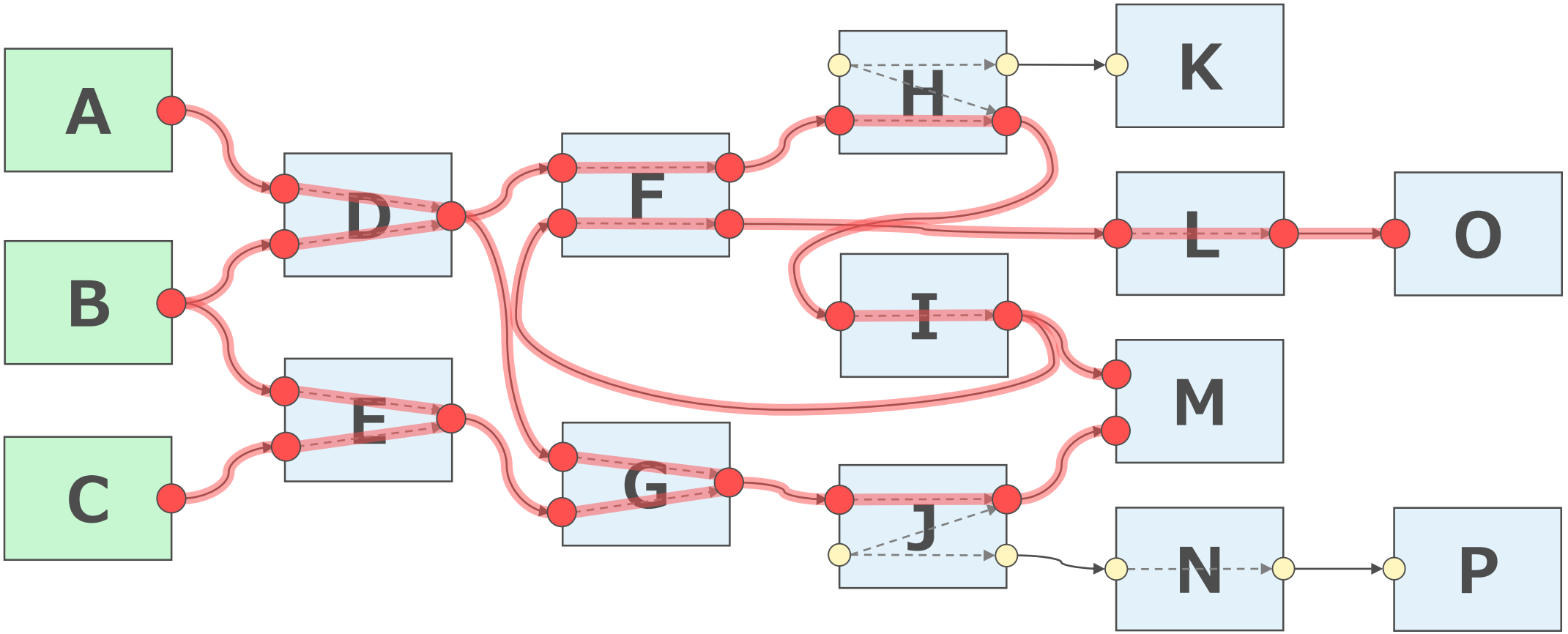
まず、このような DG があるとする。



Evaluation Graph (EG) 構築 – dirty 伝搬

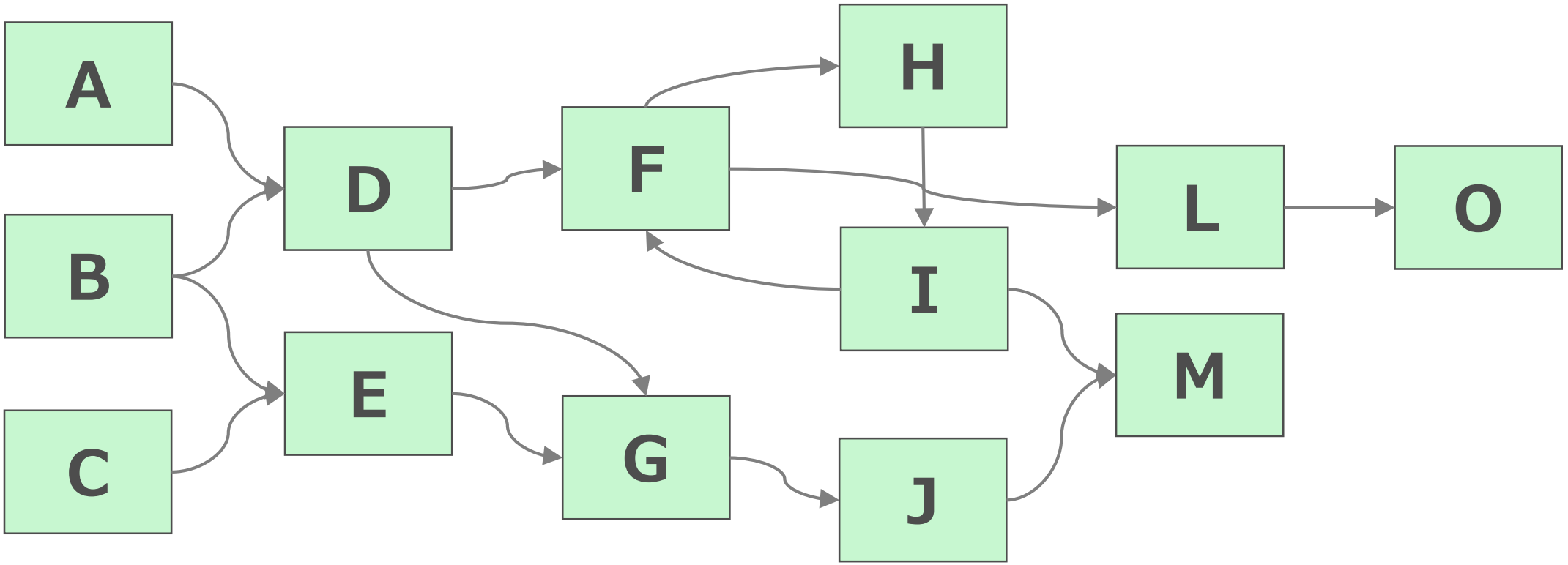
アニメーションの起点となるものから、実際に dirty 伝搬される。

（通常は、異なるキーを２つ以上持つ animCurve ノードか time ノード）



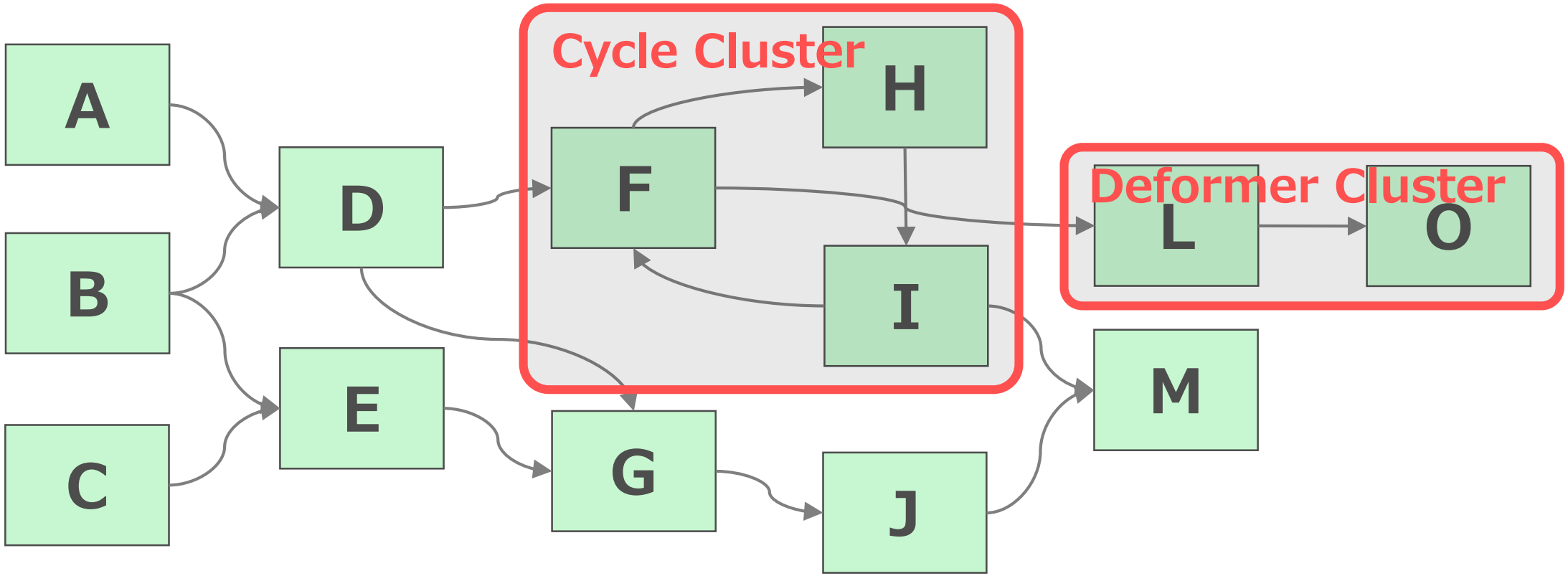
Evaluation Graph (EG) 構築 – グラフ作成

起点から下流にあるノードが収集され、dirty 伝搬で通った依存関係とともに、ノードを最小単位とする単純化したグラフが作成される。



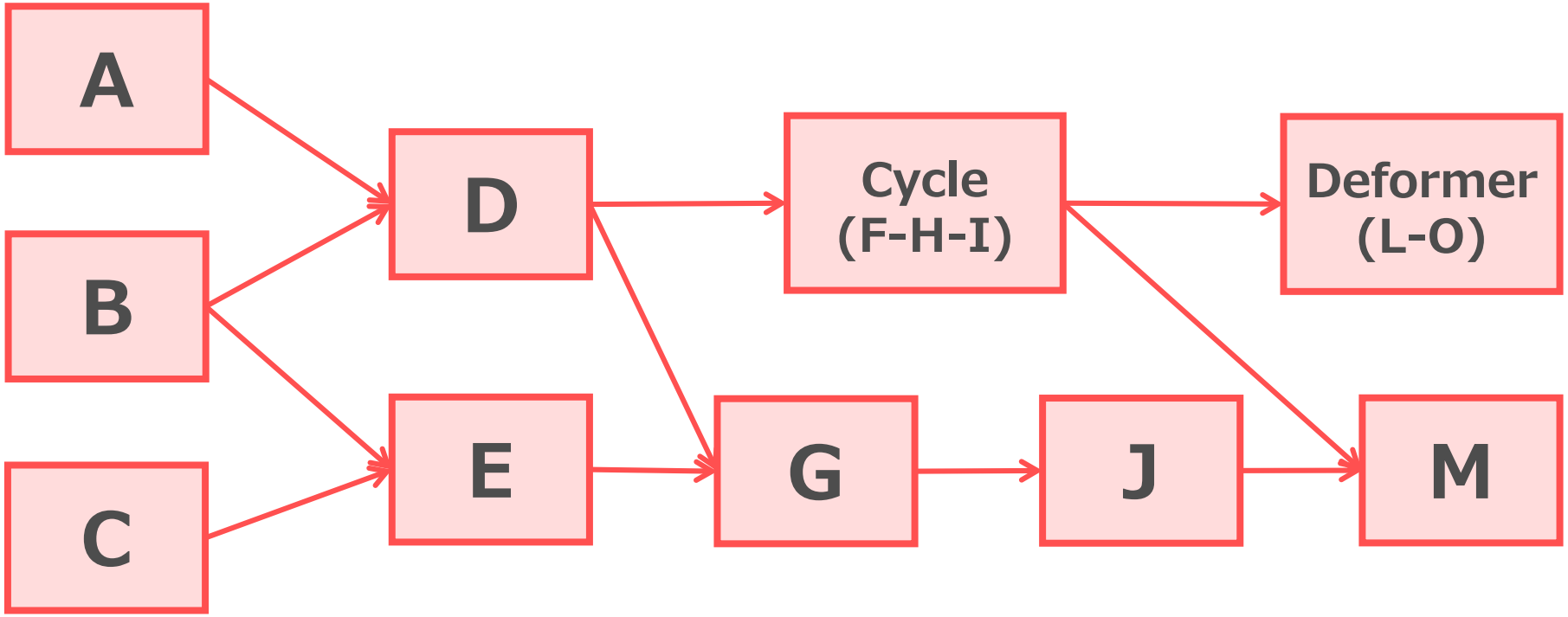
パーティショニング（クラスタリング）

並列実行可能な単位にノードがグループ化（クラスタリング）される。
ノードレベルでのサイクルは「サイクルクラスター」となる。
さらに、様々なエバリュエータによってもクラスターが作られる。



Scheduling Graph (SG) 構築

クラスターやクラスター化されなかったノードが、並列実行スケジュールのための１つの「タスク」となる。個々のタスクは「スケジューリングタイプ」を主張するので、それが考慮されて全体スケジュールが組まれる。



Push 評価の実行

- 毎フレーム、スケジュールに従ってタスクが実行される。
- 通常は、dirty 伝搬は行われない。
- タスクがノードなら、その compute() が呼ばれる。
  その際、評価されるのは Networked Plug の出力。
- クラスター内サブグラフは別の評価メカニズムになる。
  - サイクルクラスターはシリアル評価となる。
    確証はないが、EG の他の並列化部分に進む前に、そこだけ DG による Pull 評価がされると思われる（IK が問題ないので）。
  - その他は Evaluator それぞれの都合による特殊な評価となる。

Networked Plug とは

Maya® によって内部的に管理されているプラグを Networked Plug と呼ぶ。

- MPlug はアトリビュート実体にアクセスするための参照コピーのようなもので、Non-Networked と Networked とがある。
- プログラマがアトリビュートにアクセスする際に使う MPlug は、通常は Non-Networked となり、これはそのプログラム自身が保持し、任意のタイミングで生成・破棄するもの。
- 一方、コネクションが作られると、Maya® は内部で Networked Plug を生成しプラグネットワークの管理に組み入れる。
- MFnDepedencyNode::findPlug() の第 2 引数に true を指定すると、管理された Networked Plug が有るなら、その MPlug を直接取得することができる。
  ただし、その生成・破棄は Maya® が握っているので、プログラムはその MPlug を保持してはならない。
- 一旦 Networked Plug が生成されると、コネクションが切断されてもすぐは破棄されない。そのタイミングは Maya® の任意であると思われる。シーンをセーブして開き直せば破棄される。

Networked Plug と Push 評価

- 通常は、出力コネクションを持つものが Push 評価の計算対象となる。
- 接続操作の undo をしたり切断したりしても、すぐには状況は変わらず、計算対象であり続ける。
- 切断後、いつ評価対象でなくなるのかはわからない。少なくとも、シーンをセーブして開き直すと、評価されなくなる。
- やはり、内部事情の特色が強い概念なので、この挙動をあてにして「出力接続の無いプラグも評価対象にする」手段とするには微妙。
- devkit サンプルには setDependentsDirty() で Affects を作ると出力接続がなくても評価対象になるというハックが紹介されている。たしかに、それによっても Networked Plug にすることができるようだ。

  サンプルプラグイン： footPrintNode_GeometryOverride_AnimatedMaterial

セーフモード

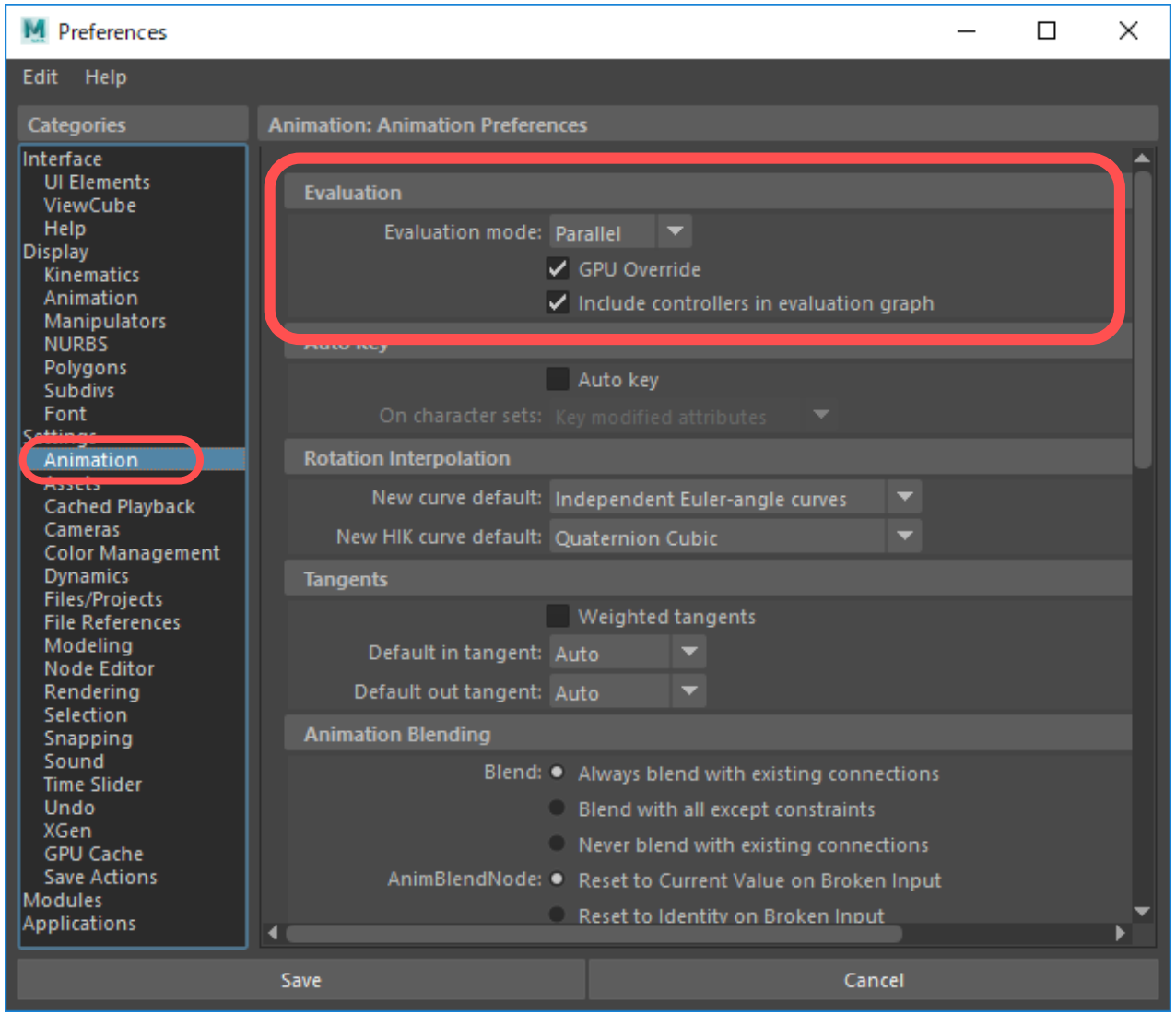
複数のスレッドが、単一のノードに同時にアクセスしようとしていることを Maya® が検出すると、問題を防ぐためシリアル実行モード（セーフモード）に移行する。

EM の基本として、
単一ノードの同時評価は抑制されていると解釈できる。

パラレル評価

Evaluation Manager の設定

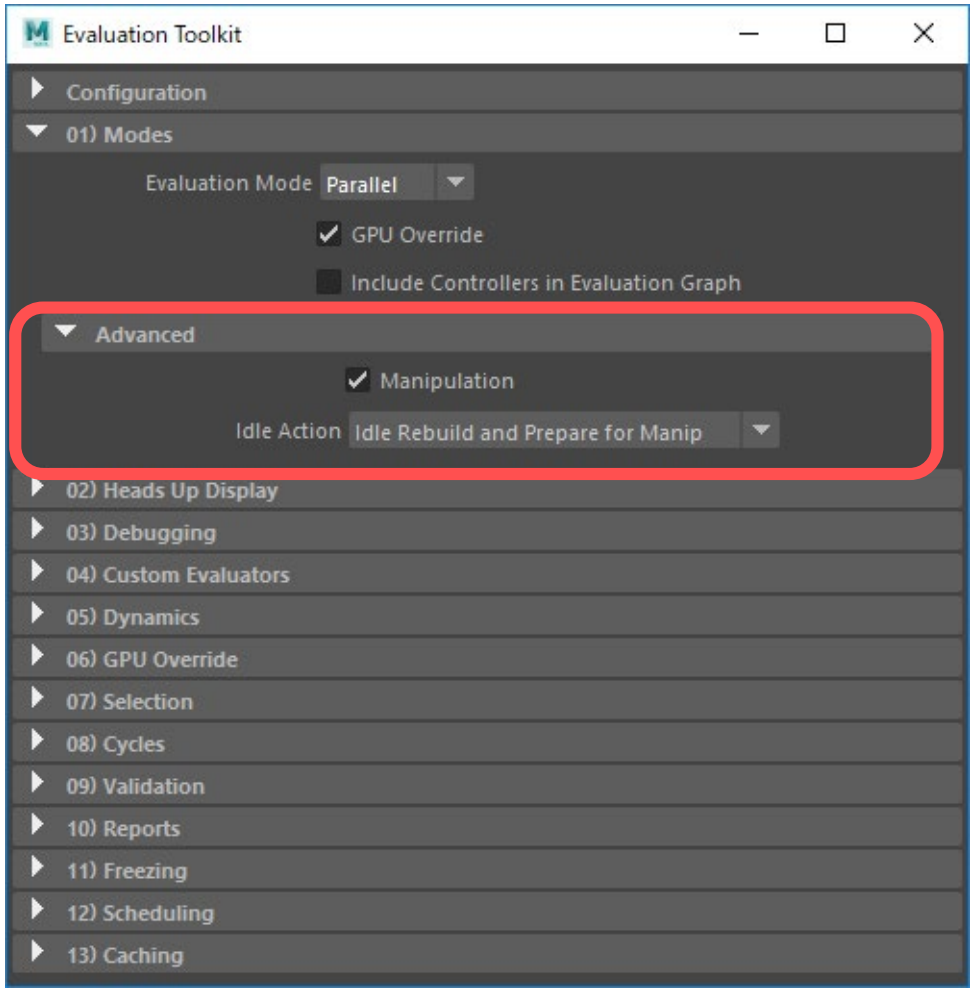
Preferences



プリファレンスでは一部の代表的な設定が保存できるようになっている。

evaluationManager コマンドでは、他にも EM の様々な設定を行えるが保存はされない。

Evaluation Toolkit



- Maya® 2017 ～
- Preferences の設定項目に加えて Advanced 設定がある。
  (Preferences 以外は原則保存されない)
- 他にも様々な設定や分析機能。
- Python で書かれている

  maya.app.evaluationToolkit

  （ソースを読むのがおすすめ！）

Evaluation Mode 設定

Parallel デフォルト

EM によるパラレル Push 評価を行うモード。

Serial

EM によるシリアル Push 評価を行うモード。

Parallel と同じメカニズムを使用するがタスクの並列実行は抑制する。

DG よりも不利なので、通常は Parallel のデバッグ目的でしか使わない。

DG

従来のシリアル Pull 評価モード。

GPU Override 設定

Parallel で VP2 の場合に、デフォーマーが GPU で高速化される。
デフォルトで ON 。

高速化されないケース：

- 必要な頂点数に満たない。(AMD: 500+, NVIDIA: 2000+)
- Backface Culling が有効になっている。
- Smooth Mesh Preview の設定が OpenSubdiv ではない。
- 変形後の結果を参照するコネクションがある。
- アニメーションによって、入力メッシュのトポロジーが変化する。
- デフォーマーの HW 非サポート機能が使われている。
- 描画モードやマテリアルの組み合わせで、サポートされない場合がある。

Include Controllers in Evaluation Graph 設定

リグのコントローラーを最初から EG に組み入れる機能。

- デフォルトでは OFF 。
- EG 構築では、通常は time か animCurve ノードを起点とするが、まだアニメートされていないコントローラーも起点となるようになる。
- コントローラーは「controller タグ」によって識別される。

利点：

- EG の頻繁なリビルドを抑制し、応答速度を高める。
- キーを打つ前から、リグの操作がパラレル評価されるようになる。
  （Manipulation 設定が ON であること）

controller タグ

2016.5 で追加された機能。

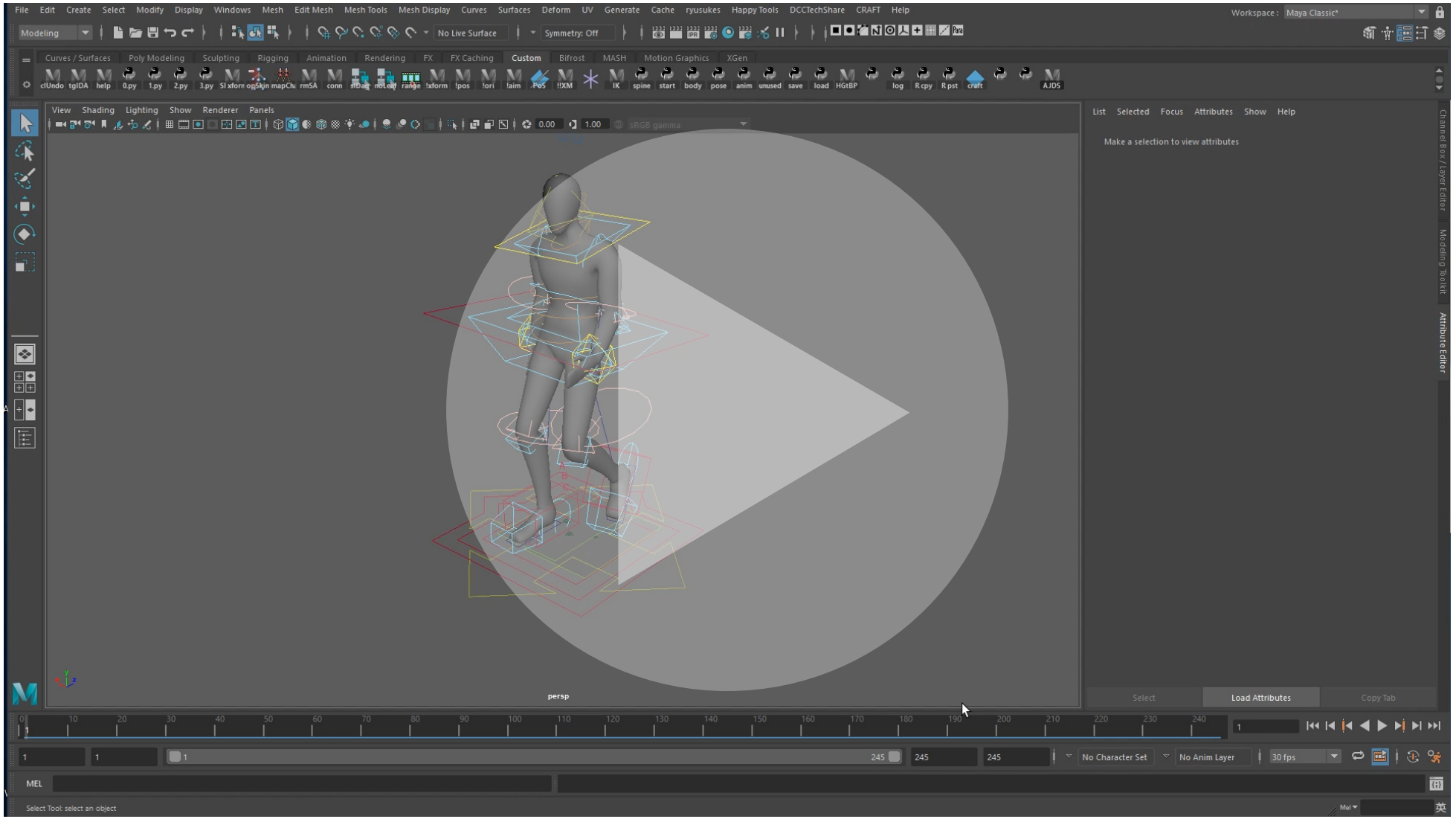
利点：

- アニメーションさせる前からパラレル評価できる。
- タグの親子設定で、DAG階層構造に依存しない pickWalk ができる。
- 2018 以降では、普段は非表示でマウスを近づけたときだけ表示する設定にすることもできる。

欠点：

- プラグインロケータの実装方法によって、表示制御が機能しないものがある。
- タグの親子設定をすると、タグノードがゴミとして残りやすくなる。

controller タグの使用例



controller を起点とする EG 構築の詳細

- controller タグはどのノードにも付けられるが、EG で認識されるのは transform 派生ノード限定されるようだ。
- 出力が認識されるアトリビュートは限られるようだ。

  認識されるアトリビュートの例：

  translate, rotate, scale,
  matrix, inverseMatrix, worldMatrix, worldInverseMatrix

  認識されないアトリビュートの例：

  shear, rotateAxis, jointOrient, rotateOrder など

Manipulation 設定

アニメーション再生でない操作中もパラレル評価するかどうか。
デフォルトで ON 。

ときには操作不能になる問題を引き起こすことがあるので、その場合は OFF にする必要がある。操作したノードから発動する Push 評価によって、そのノード自身も再評価されてしまう競合状態になるようで、特に、依存関係が正しく識別出来ないような DG の場合に発生する傾向がある。

2019 の問題について：

Attribute Editor のスライダーやバーチャルスライダー操作も対象になったようで、EG に組み入れられたノードのスライダーが操作不能になる（2019.1 でも未修正）。困る場合は Manipulation 設定を OFF にする必要がある。

Idle Action 設定

アイドル時にパラレル評価の準備をするかどうかの設定。
以下の２工程を個別に ON/OFF できる。どちらもデフォルトで有効。

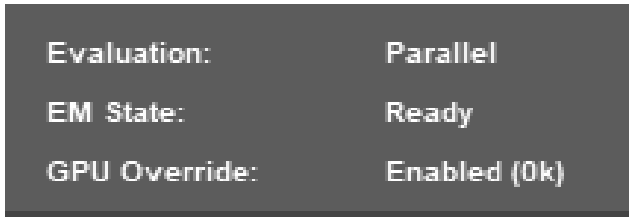
Rebuild

- DG に変更があった場合の EG の再構築。
- 2016.5～2018 では、この ON/OFF のみが可能だった。
- 2016 では設定項目が無いが、常に有効と思われる。

Prepare for Manip

- EG のパーティショニングとスケジューリング（SG 作成）。
- 2019 で追加された。Rebuild と合わせた選択式になっている。
- プレイバックキャッシュのバックグラウンド生成のためにも必要。

Evaluation HUD



- EG 生成など、パラレル評価が可能になったかどうかの状況を確認できる。
- 2019 で Idle Action が追加された影響なのか、
  EM State の Ready の意味は変わったように思われる（推測）
  - 2016.5～2018 では、スケジューリングまで。
  - 2019 では、EG 生成まででスケジューリングは含まない。

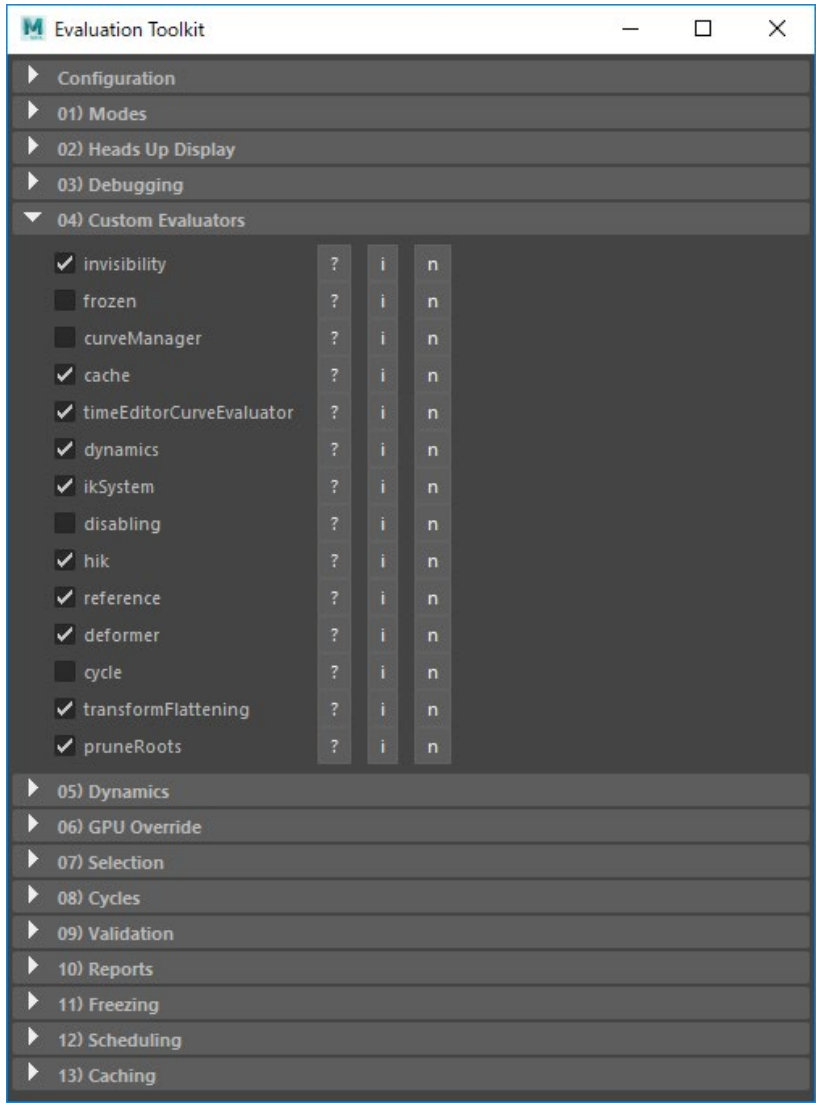
（参考）EM State = Ready の意味の推測の根拠

シーンを開いた直後にどうなるかを検証

- すぐに Ready になるか？
- ならないなら、カレントフレーム変更を何回クリックしたら Ready になるか？
- 2019 では、その後いつキャッシュ生成が開始されるか？

	Rebuild	Prepare for Manip	Ready までに必要なクリック数	Ready 後、キャッシュ開始までに必要なクリック数
2016.5～2018	OFF	n/a	2	n/a
	ON	n/a	1	n/a
2019	OFF	OFF	2	1
	ON	OFF	0	1
	OFF	ON	2	0
	ON	ON	0	0

Custom Evaluators 設定



エバリュエータを個別に ON/OFF できる。

- 必須のものから、評価効率アップのためのもの、まだプロトタイプのものまで様々。
- 「基本に忠実な DG」なら、全て OFF にしても効率の点を除けば問題ない。

パラレル評価

問題の調査

問題の種類

- 操作に対する反応が悪い（遅い、または操作不能）
- 評価結果がおかしい（DG と同じ結果にならない）
- あまり速くならない

主な対応方法：「操作に対する反応が悪い」

シーン構造の編集（リギング）

- キャッシュプレイバックを無効にする（頻繁なキャッシュ生成を回避）。
- “Idle Action” を無効にする（頻繁なリビルドを回避）。
- “Manipulation” を無効にする（操作不能になりやい）。
- プラグインノードを使っている場合は、その問題の可能性も。

アニメーション作業（リグの操作）

“Manipulation” を ON にすることで、操作中もパラレル評価となる。
（しかし、操作不能になる箇所があるなら OFF にしなければならない）

その上で：

- controller タグが使われている？

  Yes -> “Include Controllers in Evaluation Graph” を有効にする。

  No -> コントローラーに値の異なる2箇所のキーを打つ。

- curveManager エバリュエータの利用を検討する。

主な対応方法：「評価結果がおかしい」

- キャッシュプレイバックを無効にしてみる。
- VP2 と Legacy VP の切り替えを試してみる。
- エクスプレッションでコネクションに基づかない操作を行っていないか？
  特に Evaluation=“Always” で使われていないか？
- 問題ありそうなエバリュエータを OFF にしてみる。
  特に invisibility エバリュエータは問題になりやすく、
  OFF にすると通常は効率が落ちるが、それ以外の問題はない。
- 使用しているプラグインが、パラレルに対応しているか？
- そもそも Maya® のバグの可能性も有り得る。
  なるべく最新アップデートを使う。
- 地道に調査する。

主な対応方法：「あまり速くならない」

- GPU Override が効いているか？
  - 無効化されているものを探して、その理由を確認。
  - 条件の詳細は 公式ドキュメント をチェック。
- プラグインノードが効率的に実装されているか？
- 並列化を妨げているボトルネックをチェック
  - Profiler を利用
  - EG や SG を確認

  分かりやすいボトルネックの例：
  - サイクルクラスター → 特に大きなものは要注意！
  - expression → なるべく使わない
  - exprespy → なるべく使わない
  - Python で実装されたノード → C++で実装
  - IK → 仕方がないが、もし可能ならシンプルなプラグイン化

プラグインの問題の分類

プラグインがパラレル評価で正しく動かない主な原因について。

EM と DG の些細な違いによる影響

- 比較的簡単な修正で対応可能。

スレッドセーフでない実装

- 可能なら実装を修正。
- もし修正が困難なら、スケジューリングタイプを設定。

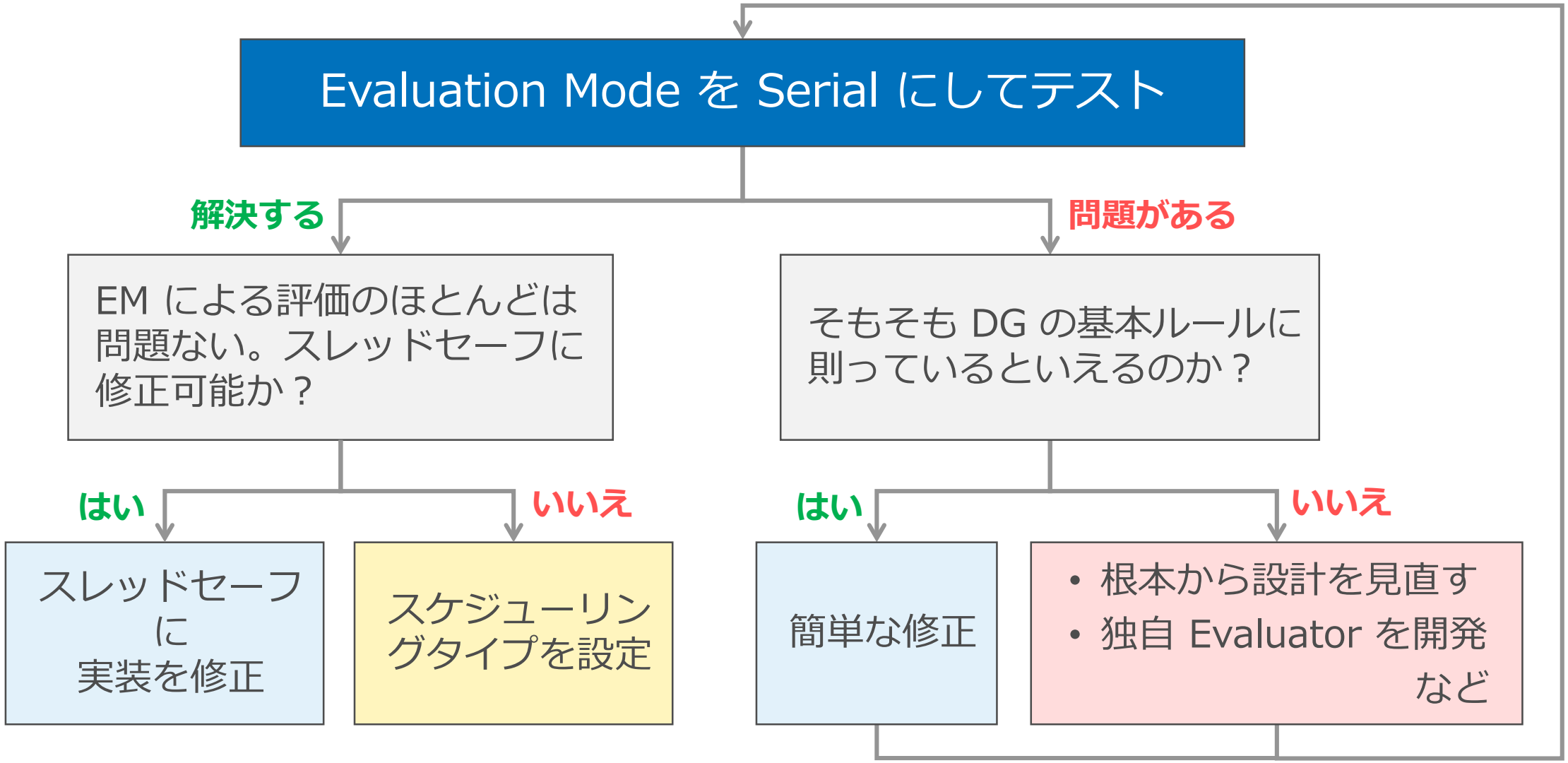
正しい EG が生成されていない

- 要因の多くは、DG の基本ルールに則っていないことだが、問題箇所が分かれば対処しやすい。

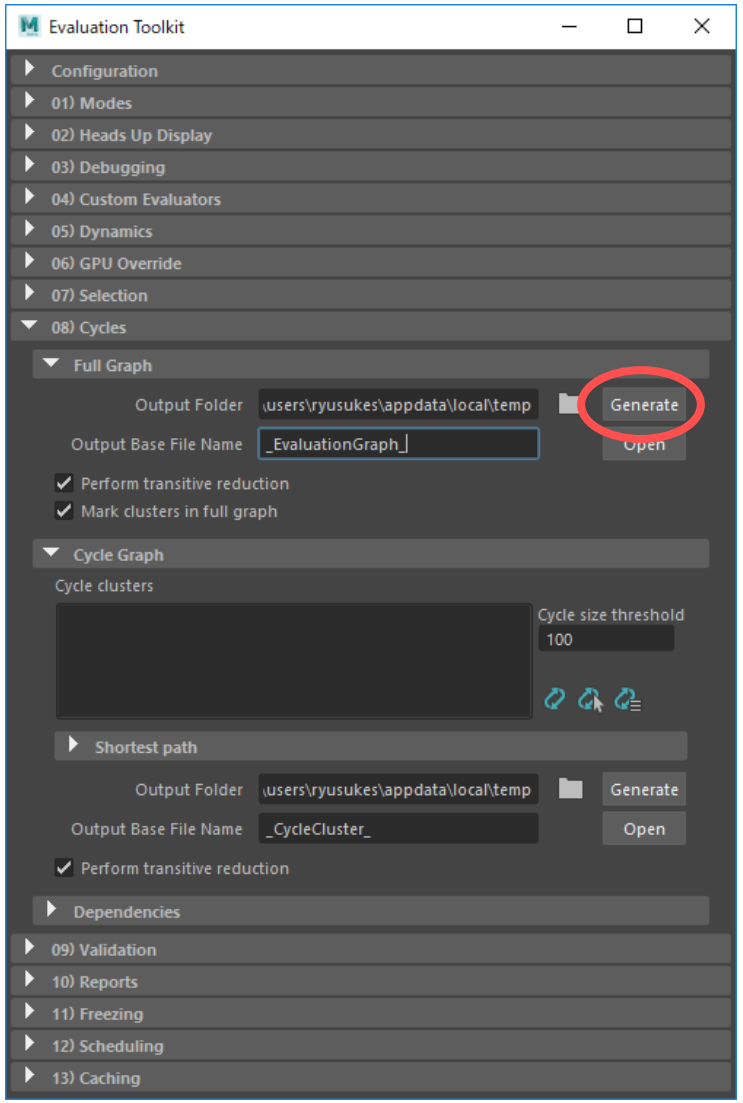
DG の基本ルールに則っていない大掛かりなケース

- 根本から設計を見直さないと修正は困難。
- EG が問題なさそうなら、スケジューリングタイプの設定で解決するかもしれない。
- 独自の Evaluator を実装するという最終手段も。
- 対応をあきらめる場合は disabling Evaluator を利用。

Serial モードでの問題切り分け



EG (Evaluation Graph) の確認



Cycles > Full Graph > Generate

同梱の Graphvis を使って、EG を pdf で図示する。

オプション：

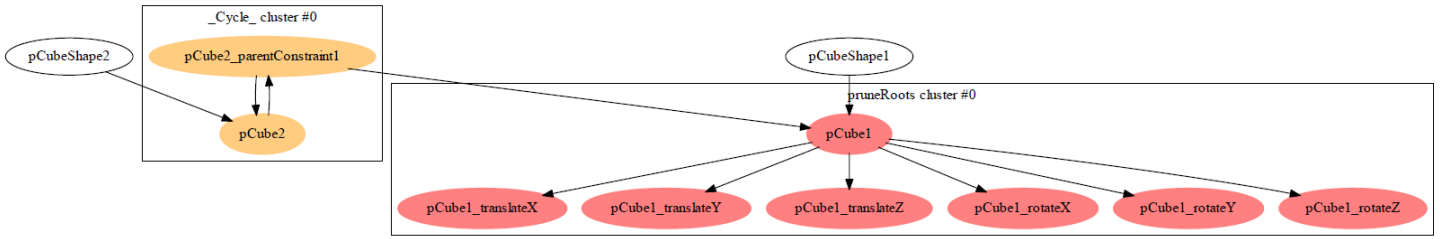
- Perform transitive reduction：

  依存関係が損なわれない範囲で線を間引いて見やすくする。

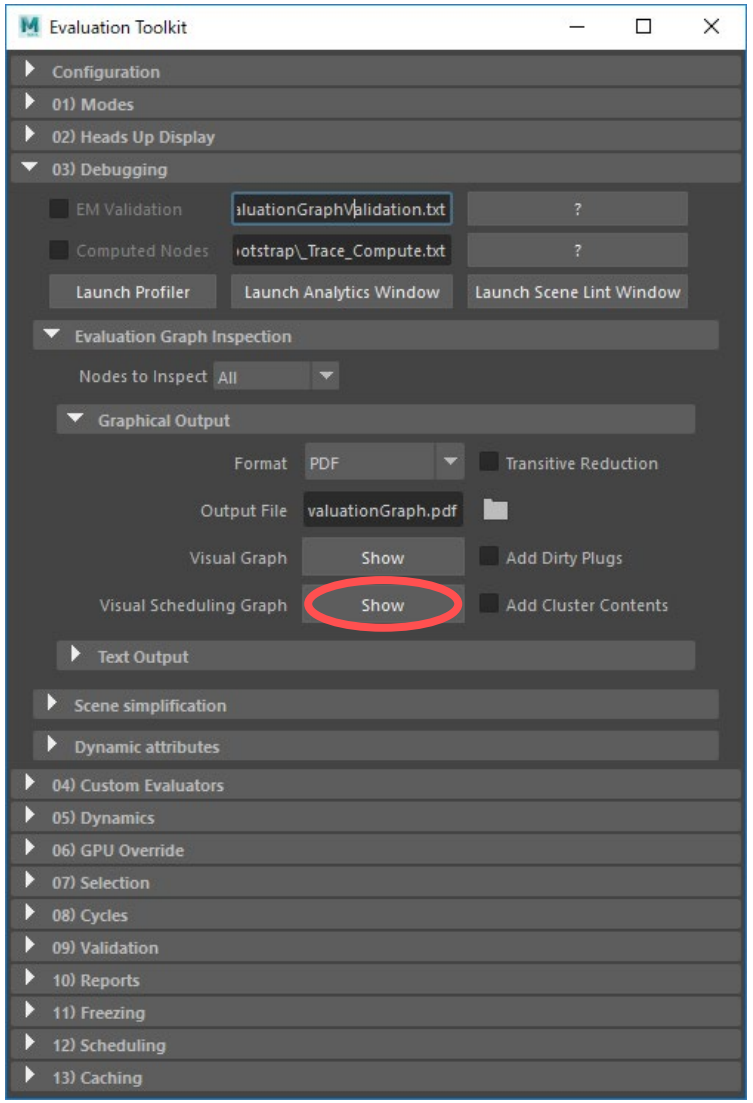
- Mark clusters in full graph：

  クラスターも図示する。
  サイクル以外のクラスターも図示するには、パーティショニング工程まで終わっている必要がある。

出力例：



SG (Scheduling Graph) の確認



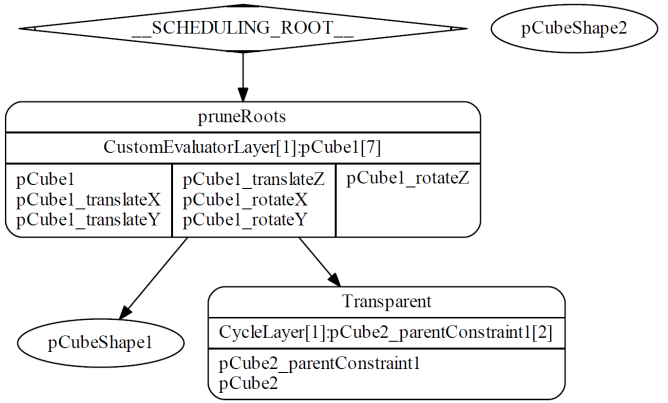
Debugging > Evaluation Graph Inspection > Graphical Output > Visual Scheduling Graph > Show

同梱の Graphvis を使って、SG を pdf で図示する。

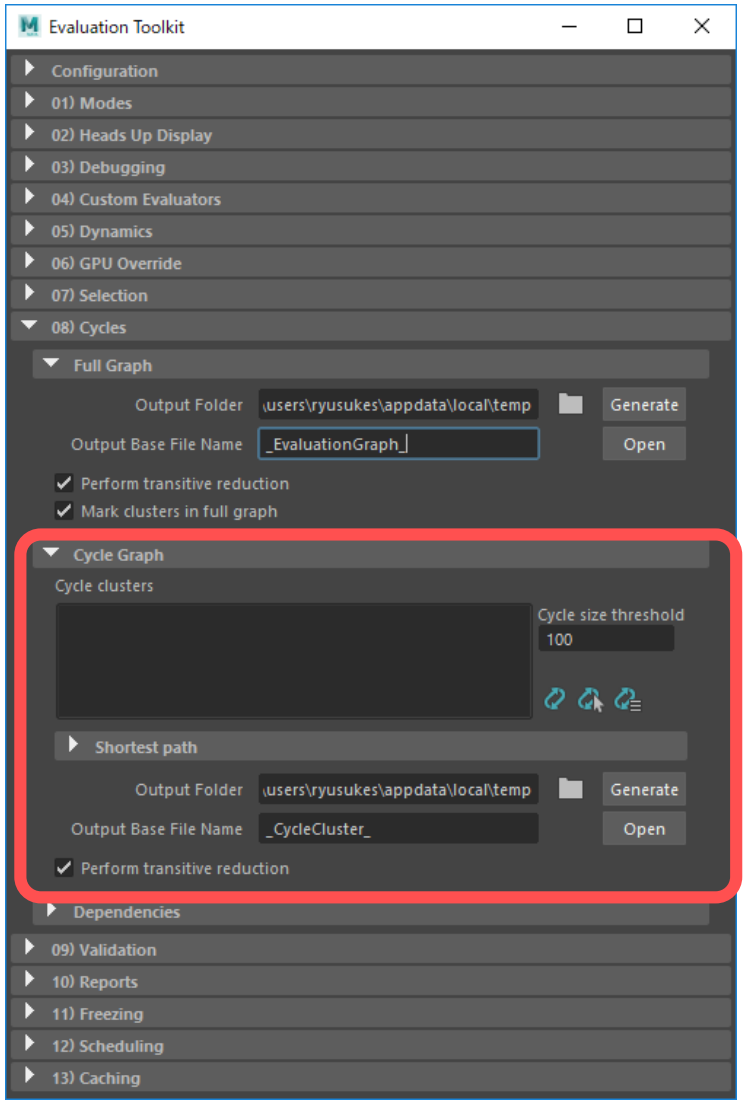
オプション：

- Add Cluster Contents：
  タスクがクラスターの場合に内包する EG ノードも記載する。

出力例：



サイクルクラスターによるボトルネックの確認



Cycles > Cycle Graph

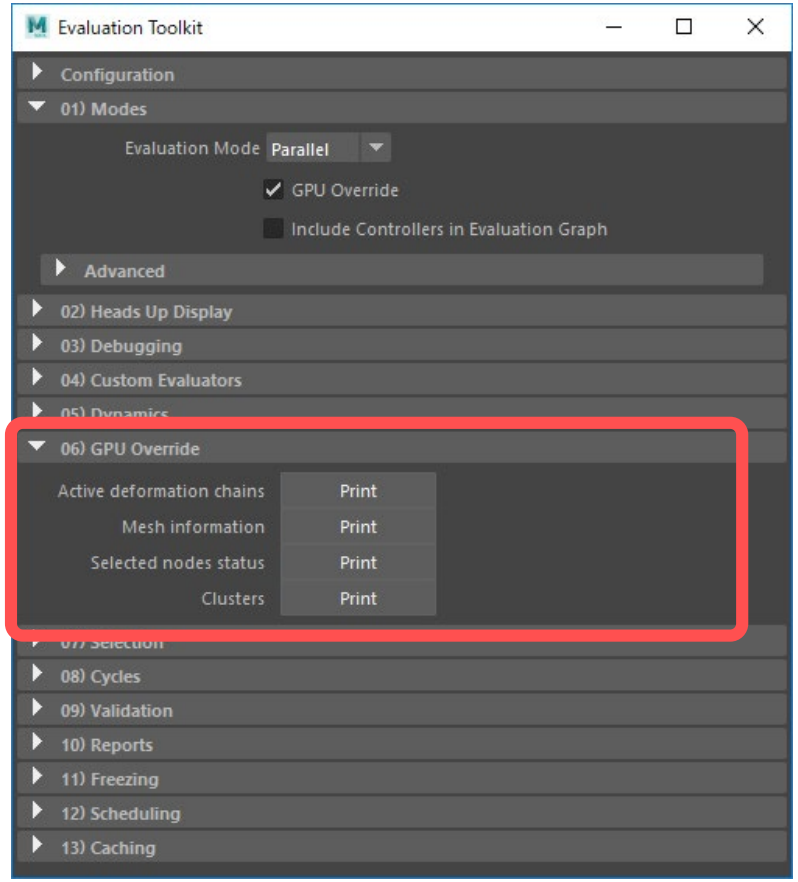
サイクルクラスターは小さければ大きな問題にはならないが、大きいと並列評価のボトルネックになる。

Cycle size threshold に、含まれるノード数の最低数を指定して、それ以上のものをリストアップすることができる。

サイクルクラスターはそのサイズとともにリストアップされる。

リストから選択して **Generate** で、クラスター内サブグラフを pdf で確認もできる。

GPU Override の確認



GPU Override

> Active deformation chains

有効になったデフォーマーとシェイプの情報を表示。

> Mesh information

シーン全体の mesh シェイプの情報を表示。
無効化されたものはその理由を知ることができる。

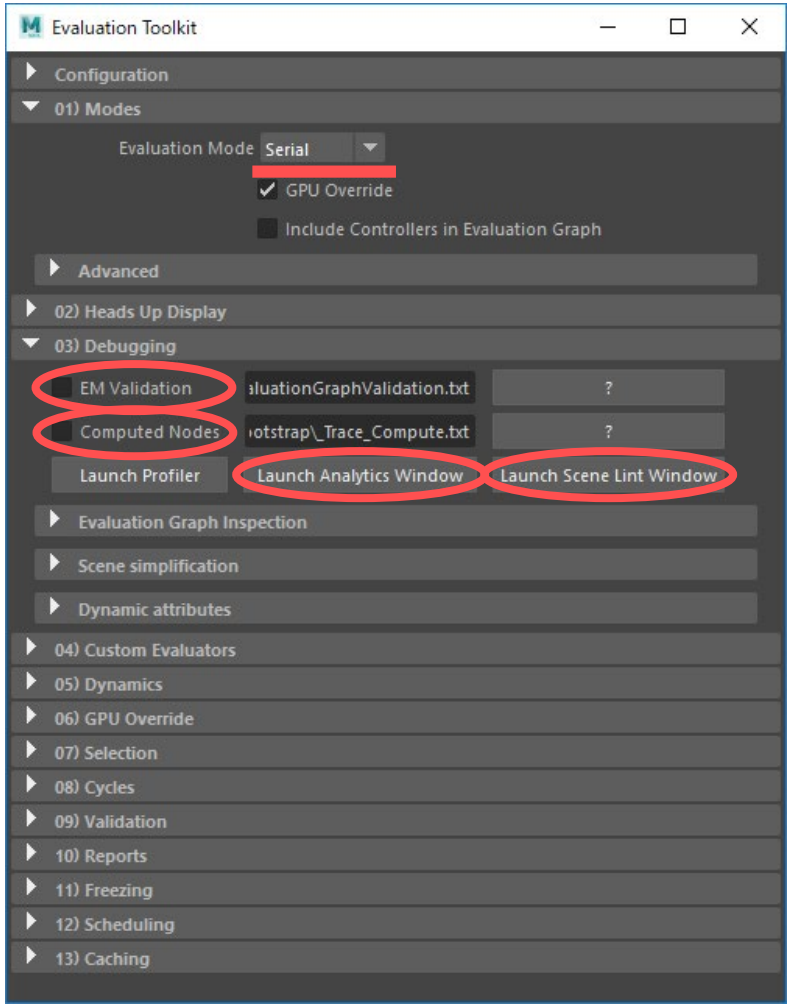
> Selected nodes status

選択しているシェイプの情報を表示。
無効化されたものはその理由を知ることができる。

> Clusters

deformer エバリュエータのクラスターの内容を表示。

デバッグ機能



Debugging

> EM Validation

Serial でのみ動作する診断モード。
正しくない結果になりそうな依存関係などがログされる。

> Computed Nodes

Serial と DG で動作するトレースモード。
各ノードの compute() の呼び出し順がログされる。

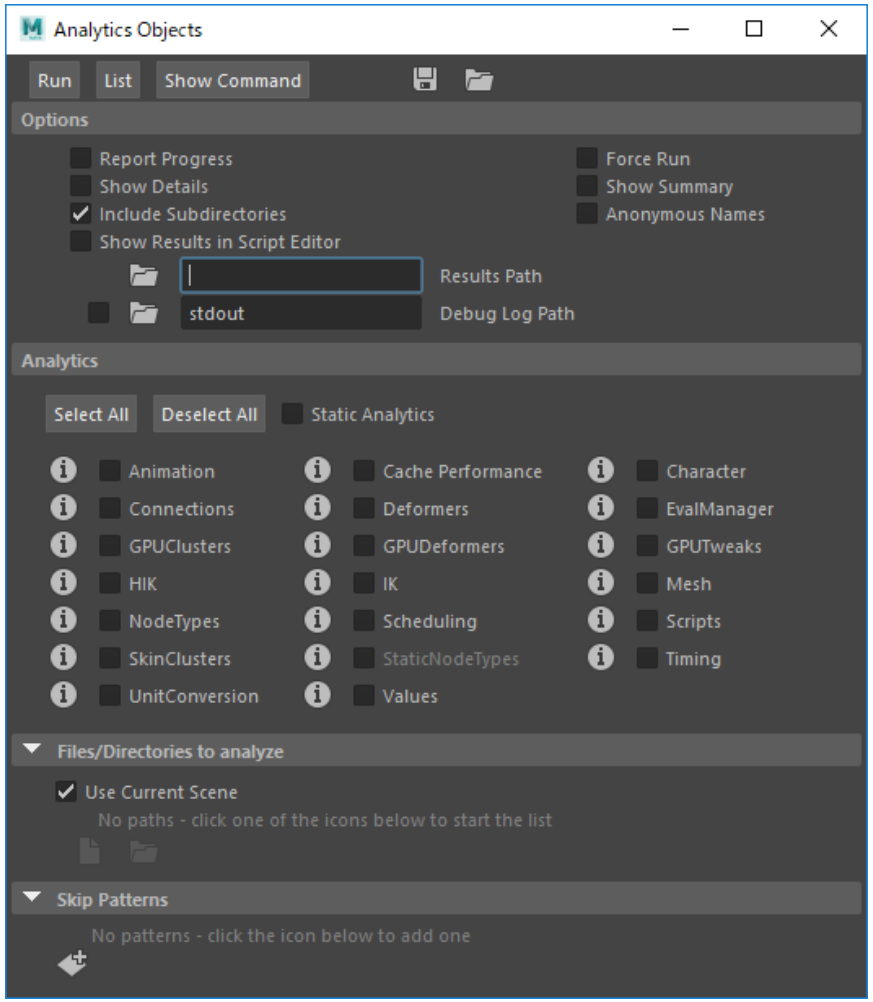
> Launch Analytics Window

解析ツールを開く。

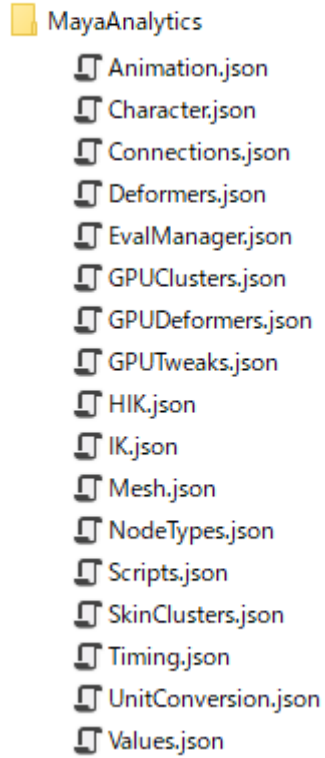
> Launch Scene Lint Window

シーン構造の静的解析ツールを開く。

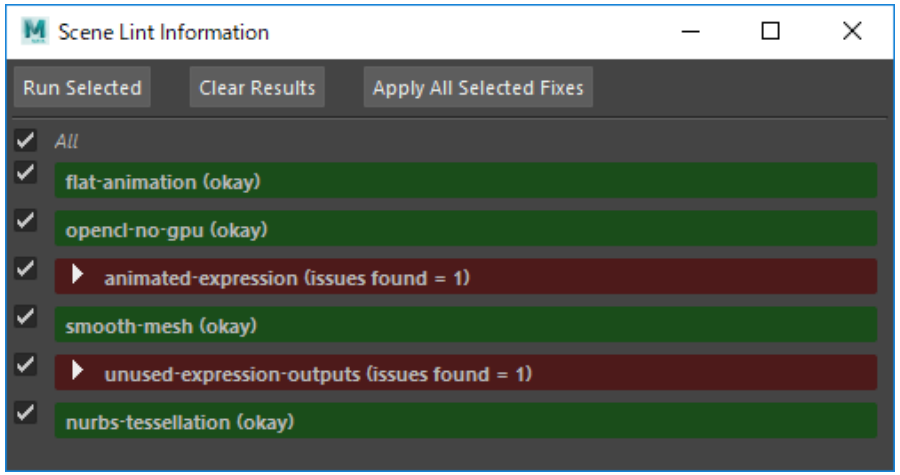
Analytics Window



2018 で追加された機能。
チェックボックスで選択した解析を実行し、
json ファイルに出力する。



Scene Lint Window



シーンの静的解析ツール。2019 で追加された。

多くの場合に問題になったり、非効率であったりすることが、
比較的明らかなものがレポートされる。

EG の破棄

何らかの設定を変更した際、
すぐに反映されなかったり、おかしくなったりすることもある。
そういうときは、手動で EG を破棄すると良い。

```
cmds.evaluationManager(inv=True)
```

- UI には無いのでシェルフなどに登録しておくと良い。
  単に、何か変になったときのリセットボタンとしても使える。
- その後、Idle Action 設定に応じてリビルドされる。
- コマンドマニュアルには「リビルドを Idle 時にするかどうか」の設定であるかのように書かれているが、それはおかしい。

パラレル評価

スケジューリングタイプ

スケジューリングタイプ

- タスクを並列評価できる信頼度を表す。
- 各ノードやクラスターが、自身の実装に基づき表明する。
- 同じ EG でも、各タスクのスケジューリングタイプによって、作られるスケジュール (SG) が変わる。
- evaluationManager コマンドでオーバーライド可能。
  - 一時的なテスト用か？
  - 信頼度は個々の実装に基づき決まるはずなので、通常はシーンで変更する必要はないと思われる。

スケジューリングタイプの種類（良い順）

Parallel

- 並列評価して完全に問題ない。多くのノードはこれ。

Serial

- 並列評価してほぼ問題ないものの、隣接したこのタイプのタスク同士は並列評価されない。
- 不思議な感じだが、ikHandle は Serial であることで推して知るべし。
- プラグインのデフォルトなので、本当に問題ないものは Parallel への格上げを明示すべき。

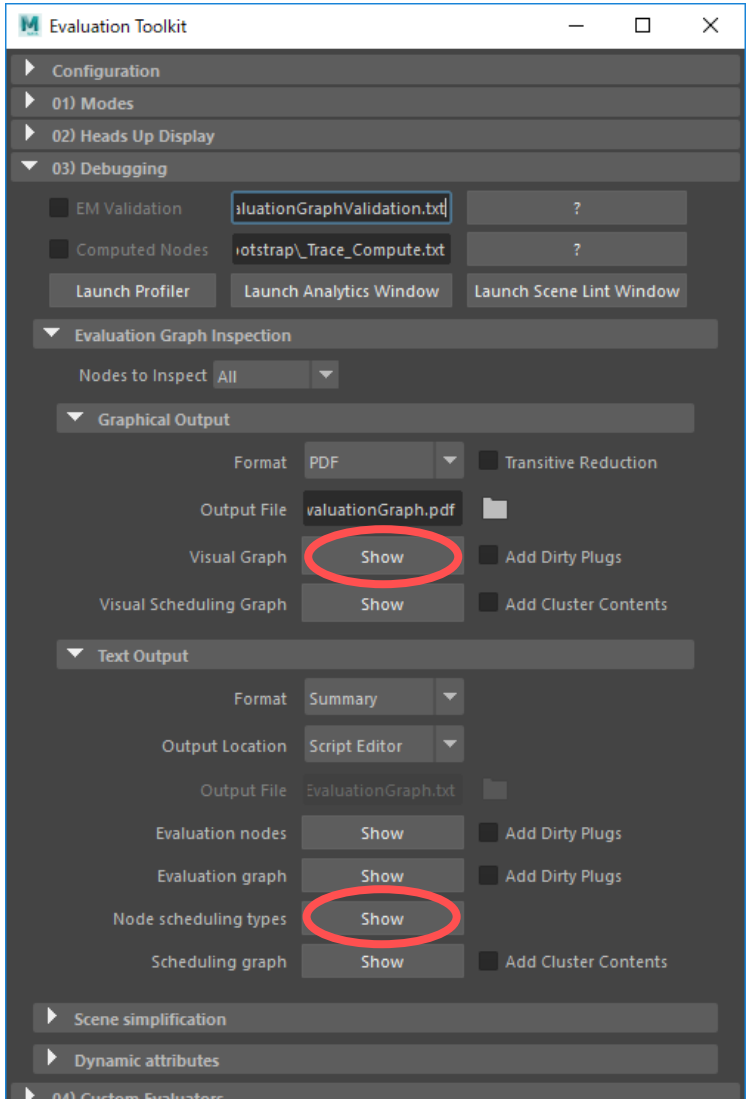
GloballySerial

- 他のノードタイプとは並列評価されるが、このノードタイプ同士は並列評価されない。
- 他のノードタイプとはリソース共有がない場合に。

Untrusted

- このノードが評価される間は、他のノードは一切評価されない。
- ノードタイプを超えたリソース共有がある場合に。

スケジューリングタイプの確認



Debugging > Evaluation Graph Inspection

> Graphical Output > Visual Graph > Show

SG を pdf で図示する。Parallel 以外のスケジューリングタイプに色が付いて確認できる。

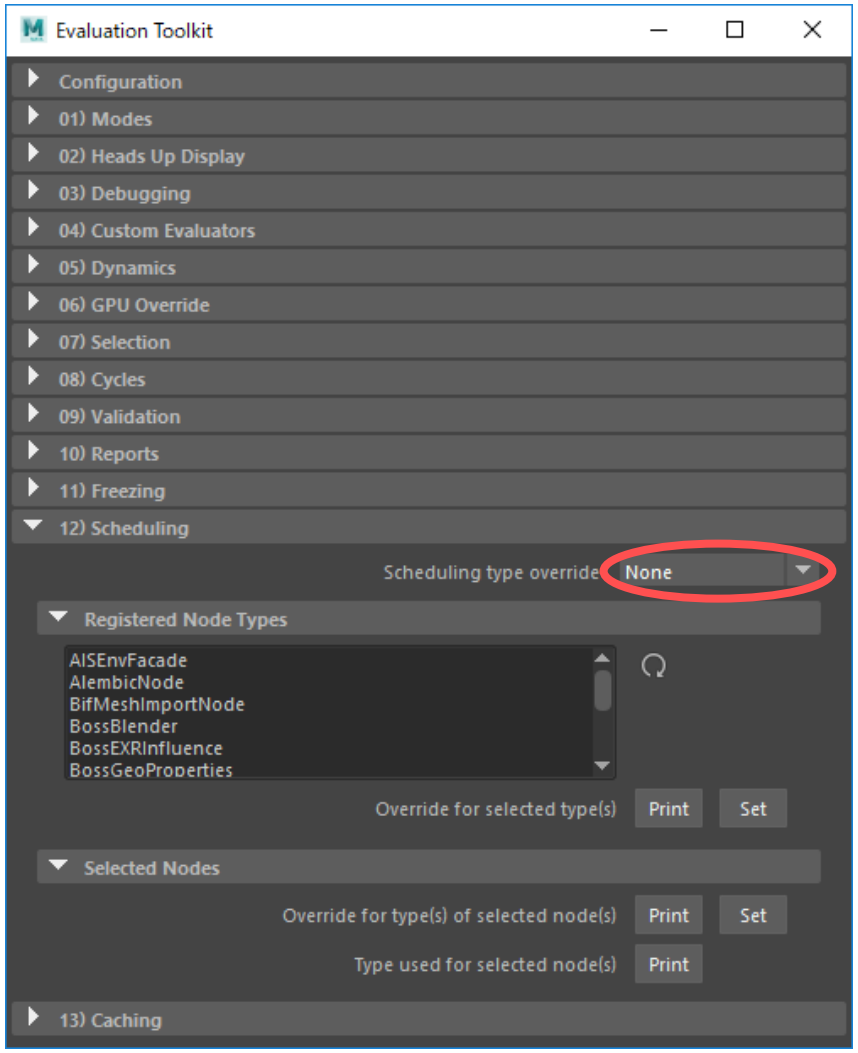
オプション：

- Add Dirty Plugs：
  EG 生成の元になった dirty プラグを表示。

> Text Output > Node scheduling types > Show

各ノードのスケジューリングタイプをテキスト出力する。

スケジューリングタイプのオーバーライド



Scheduling

> Registered Node Types

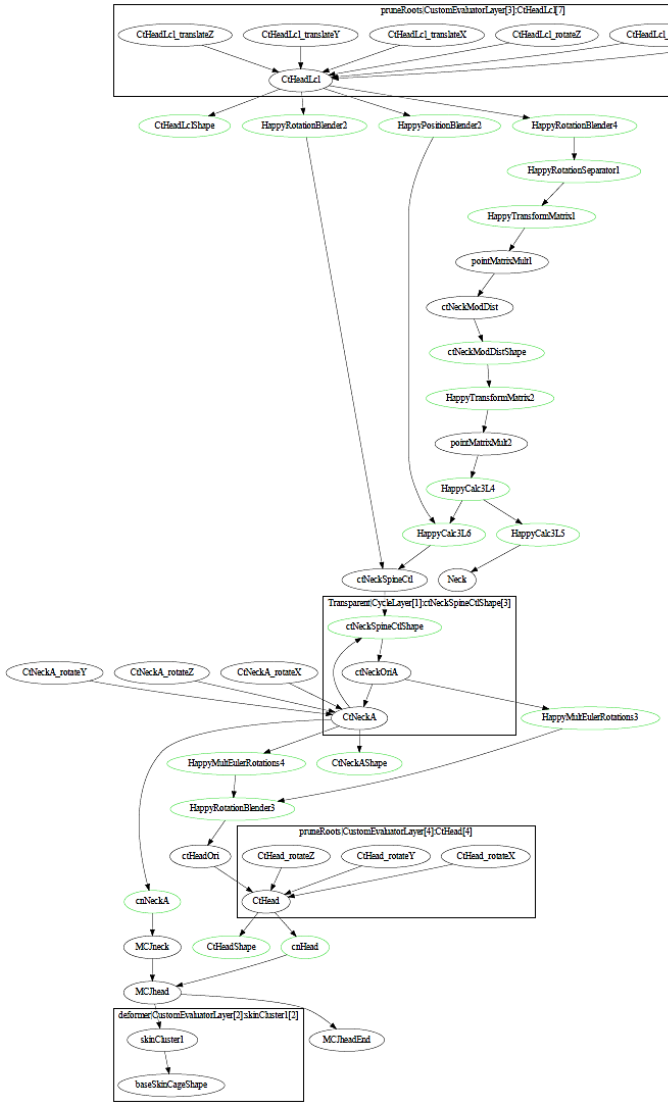
ノードタイプに対してオーバーライドする。

> Selected Nodes

シーン中のノード個別にオーバーライドする。

本来は、各ノードタイプの実装によって
明示されるべきものなので、
一時的なテスト目的と考えたい。

SG がどのように変わるのか？

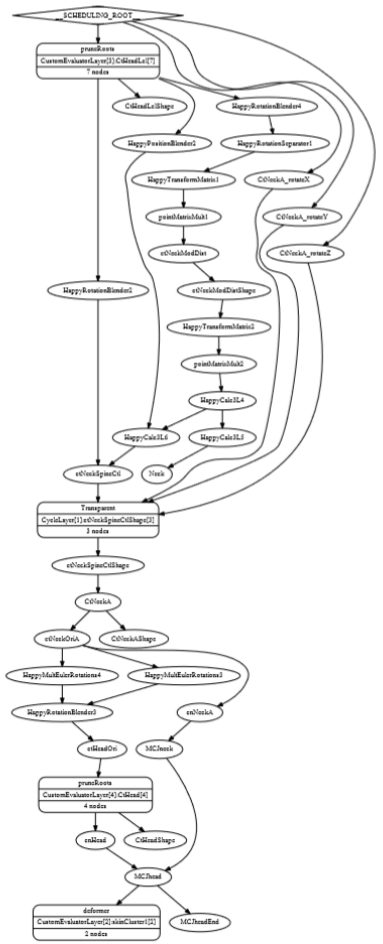


元となる EG 。

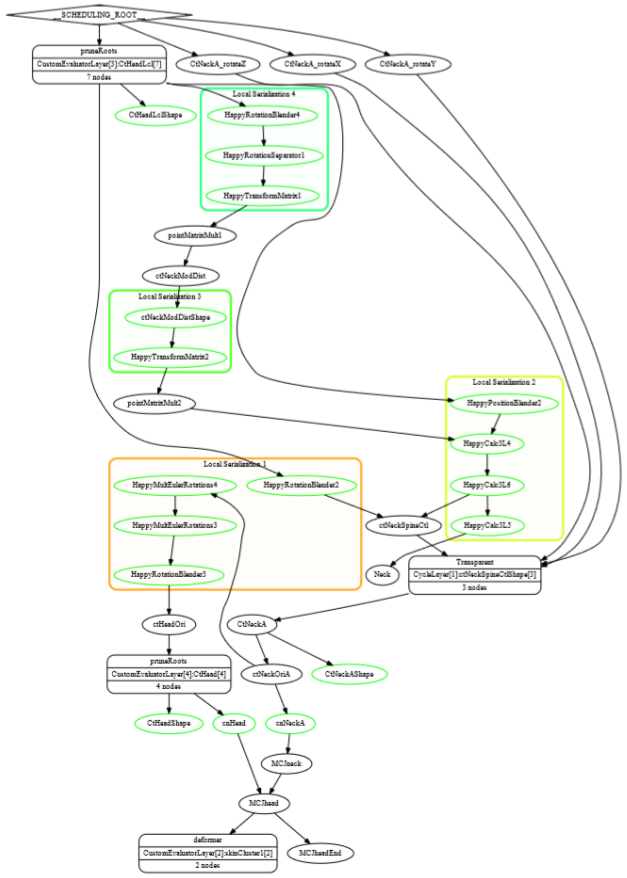
薄いグリーンのノードがプラグインノード。

これらのスケジューリングタイプを変更して、生成される SG の違いを見てみる。

SG の例：Parallel と Serial の場合

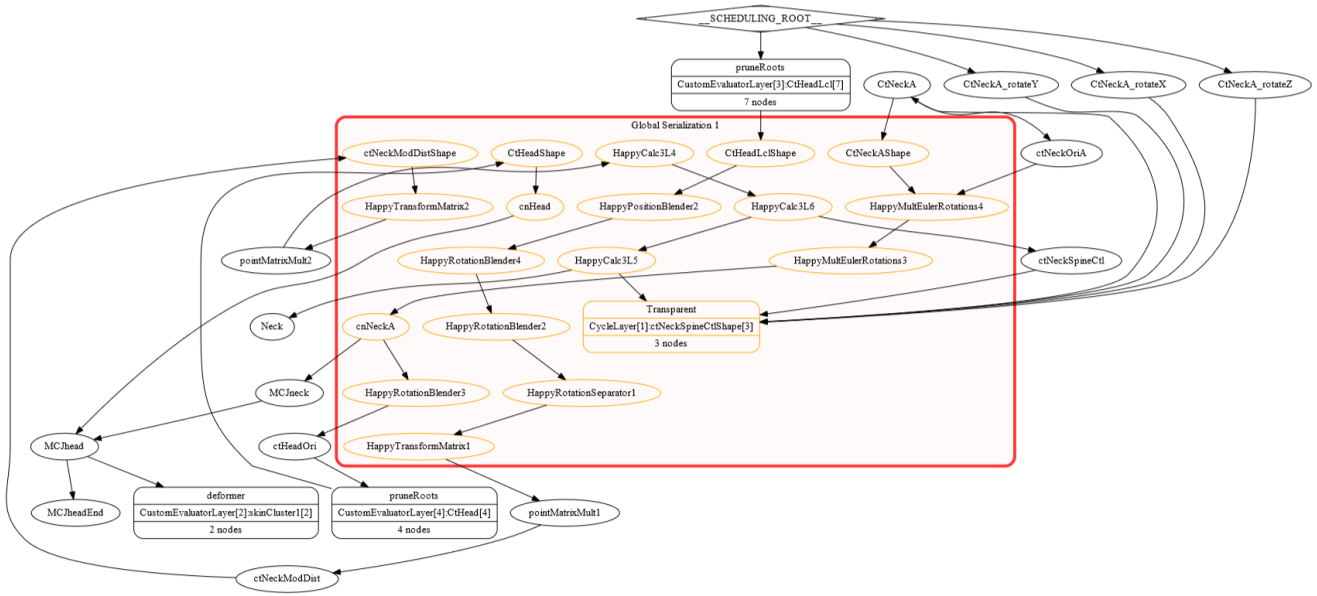


全て Parallel の場合
可能な限り並列化される

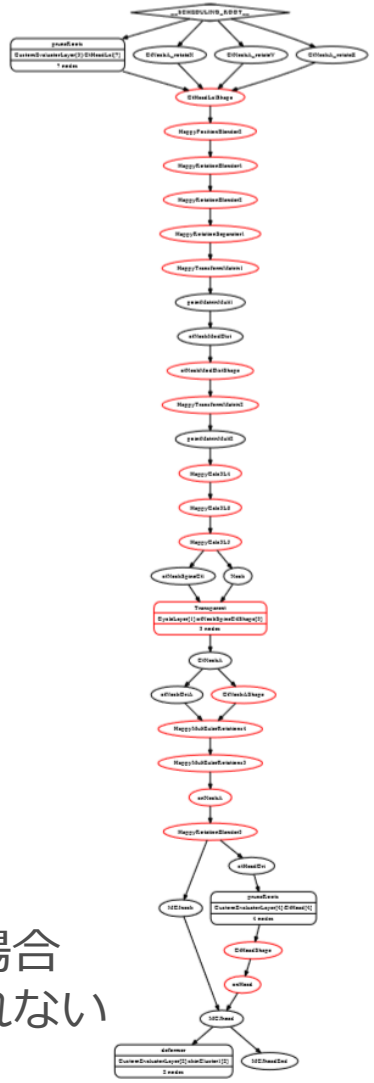


Serial が含まれる場合
隣接する Serial はクラスタ化されるが
全体の並列性はさほど損なわれない

SG の例：GloballySerial と Untrusted の場合



GloballySerial が含まれる場合
その中の同じノードタイプ同士は
並列評価されない



Untrusted が含まれる場合
Untrusted は並列評価されない

スケジューリングタイプの実例

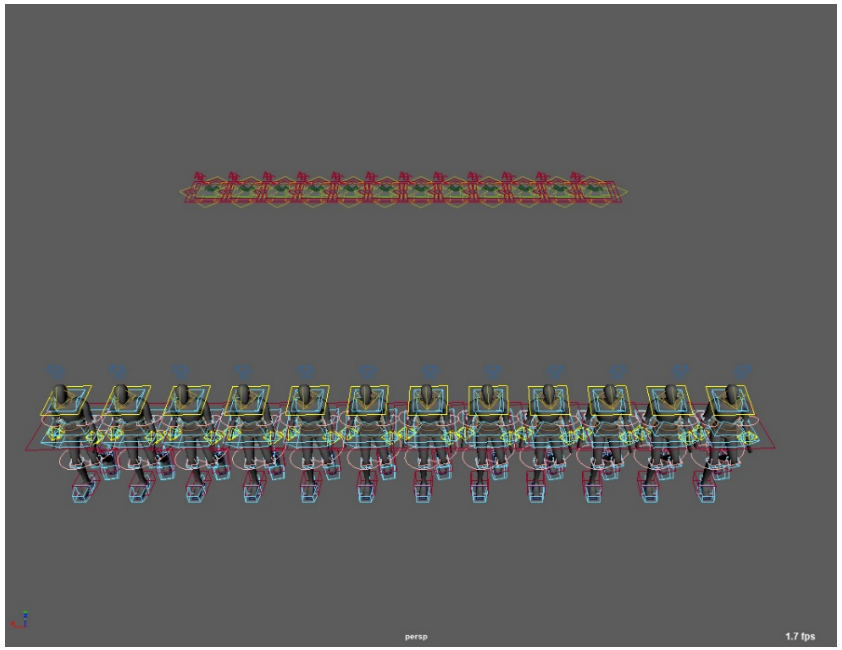
- ikHandle → **Serial**
- expression
  - 入出力に基づかない処理が確認されたら → **Untrusted**
  - 入出力のみに基づいた処理が確認されたら → **GloballySerial**
- Python 関連
  - exprespy → **GloballySerial**
  - Python で実装された全てのノード → **GloballySerial**

  たとえ Parallel としても、どのみち GIL によって Python ノード同士の並列実行はされないが、GloballySerial を表明することで、それが考慮されたスケジューリングを促し、GIL 待ちネックを少しでも回避している。

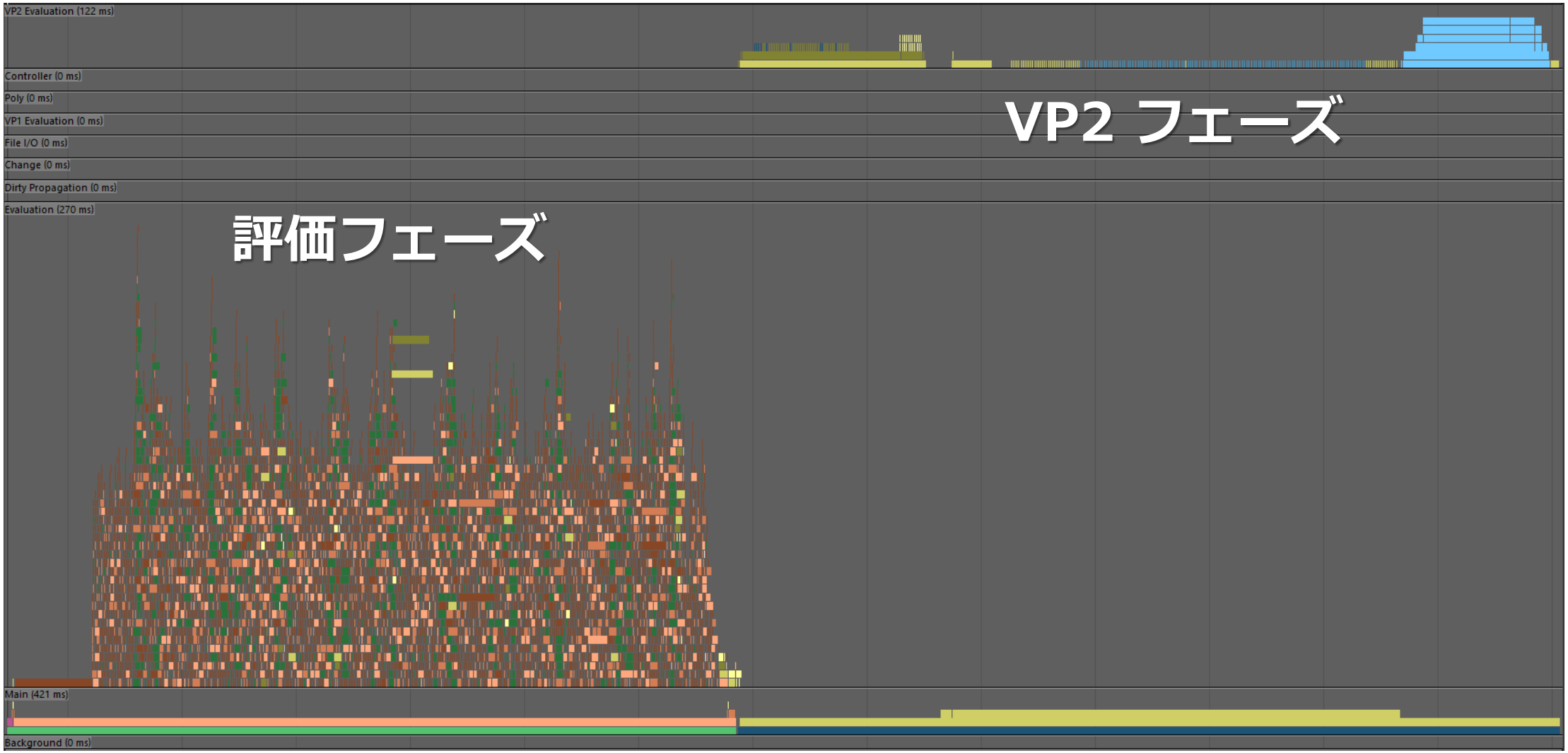
スケジューリングタイプのプロファイリング

Biped リグ 12 体を、VP2 + Parallel 評価。

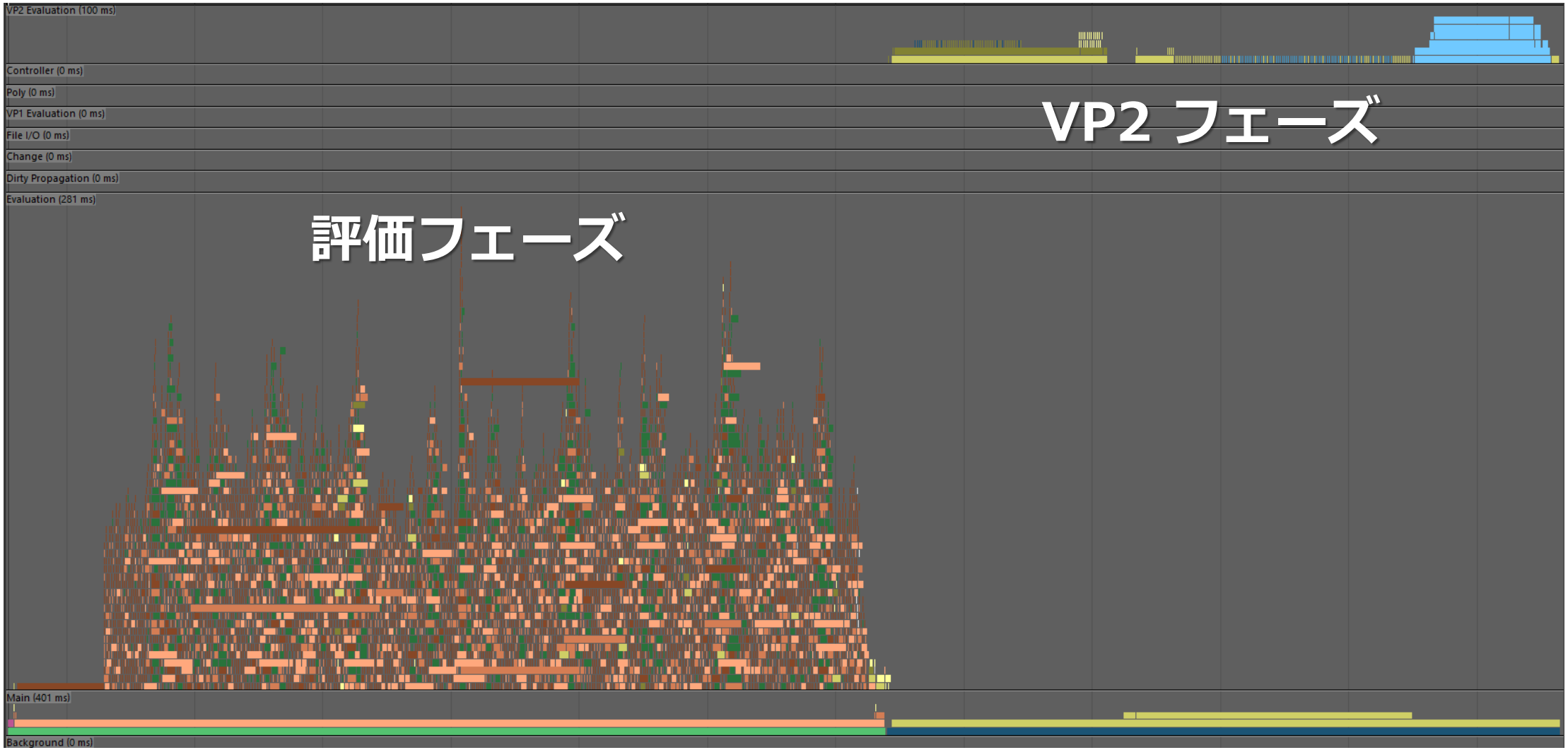
リグで使用しているプラグインノードのスケジューリングタイプを変更し、プロファイリング結果を比較する。



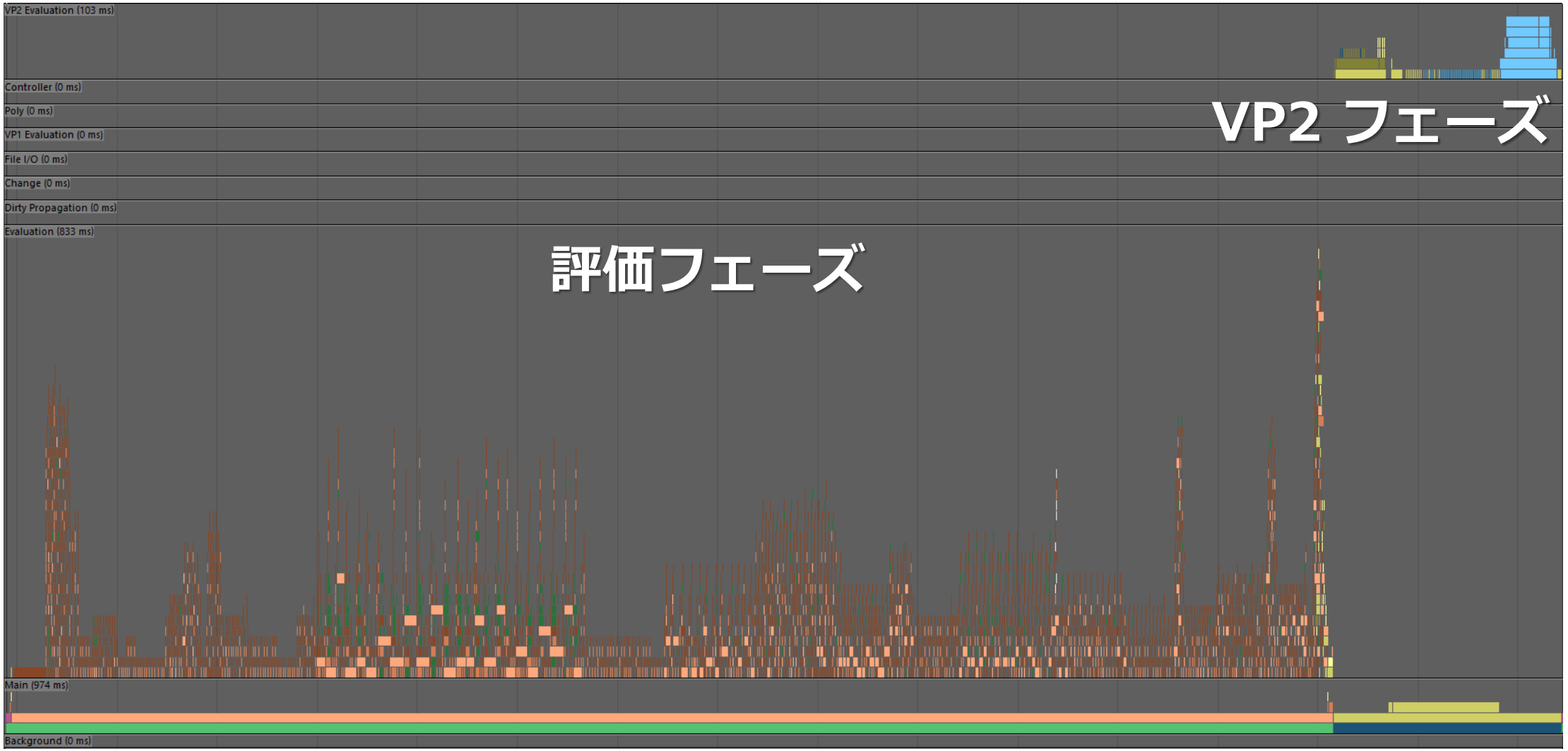
Parallel のプロファイリング



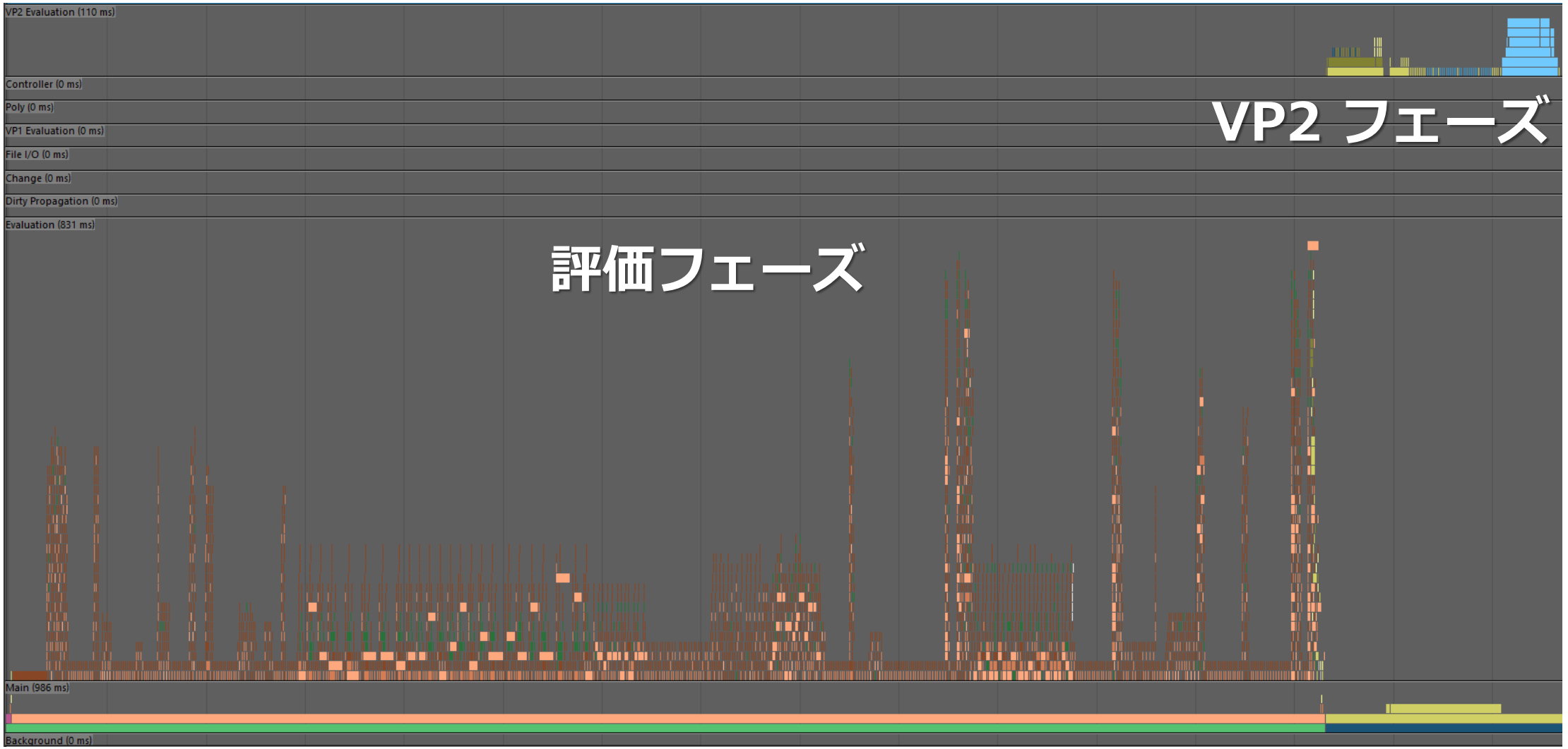
Serial のプロファイリング



GloballySerial のプロファイリング



Untrusted のプロファイリング

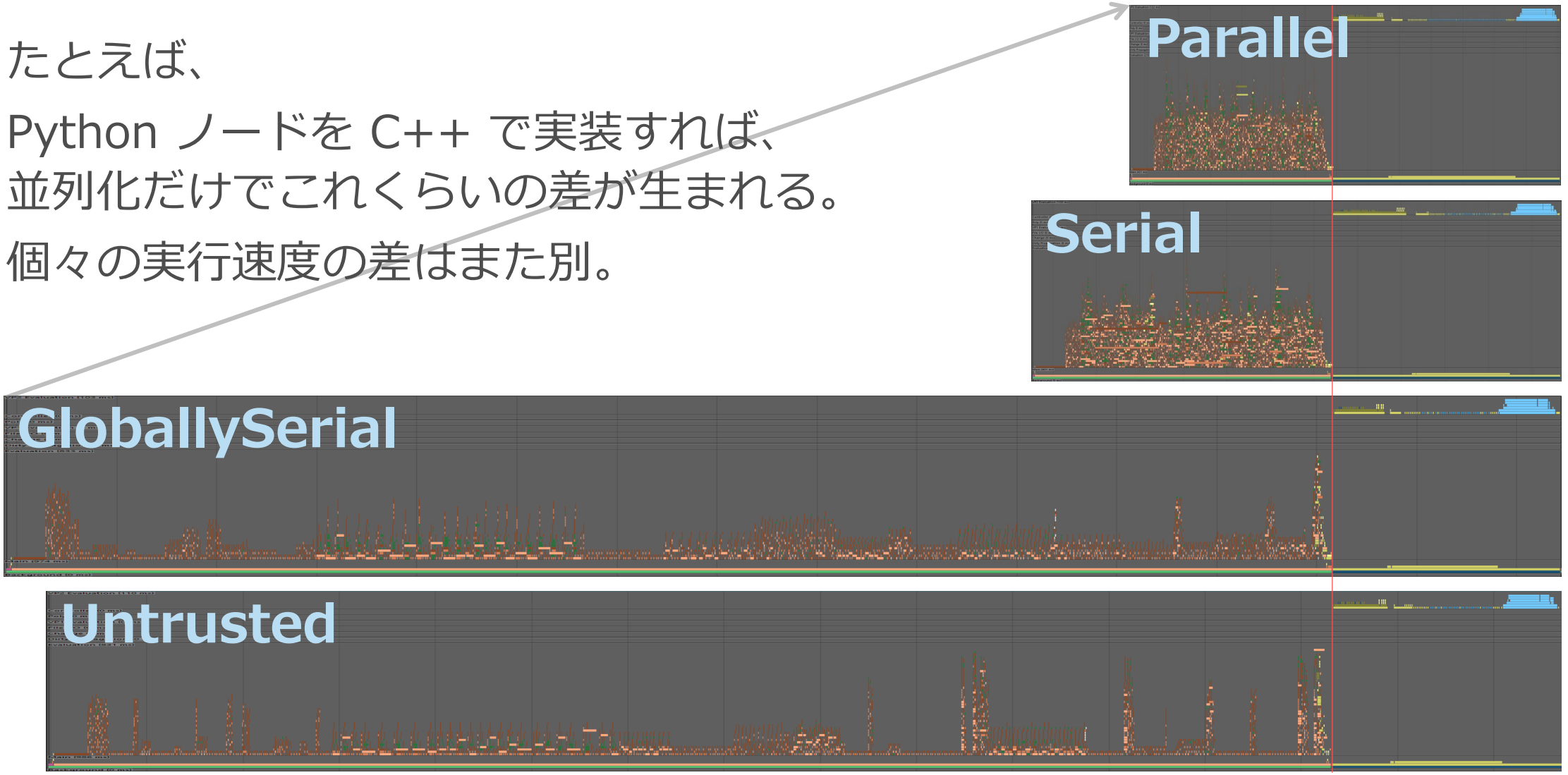


１フレームに要した時間を比較

たとえば、

Python ノードを C++ で実装すれば、
並列化だけでこれくらいの差が生まれる。

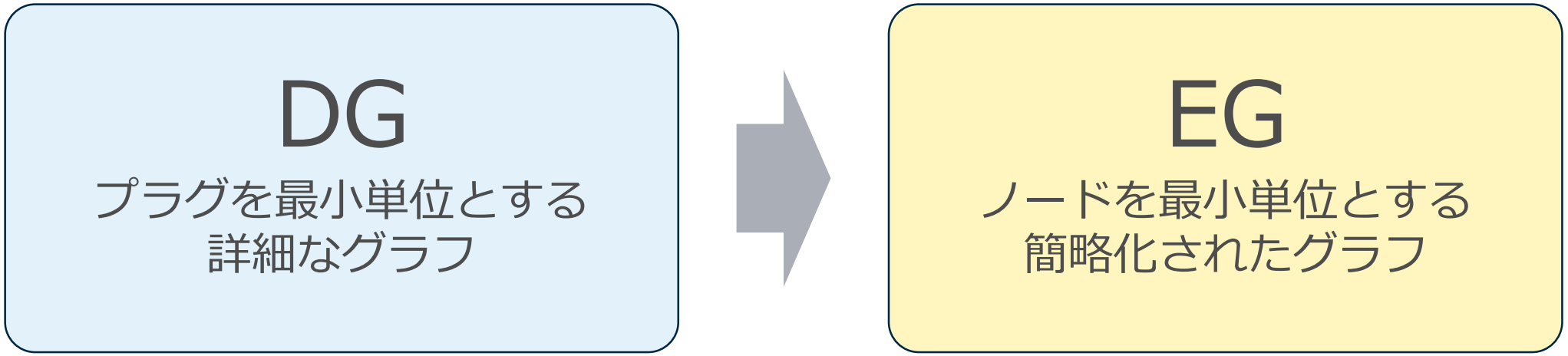
個々の実行速度の差はまた別。



パラレル評価

応用

サイクルクラスターが作られる条件の考察



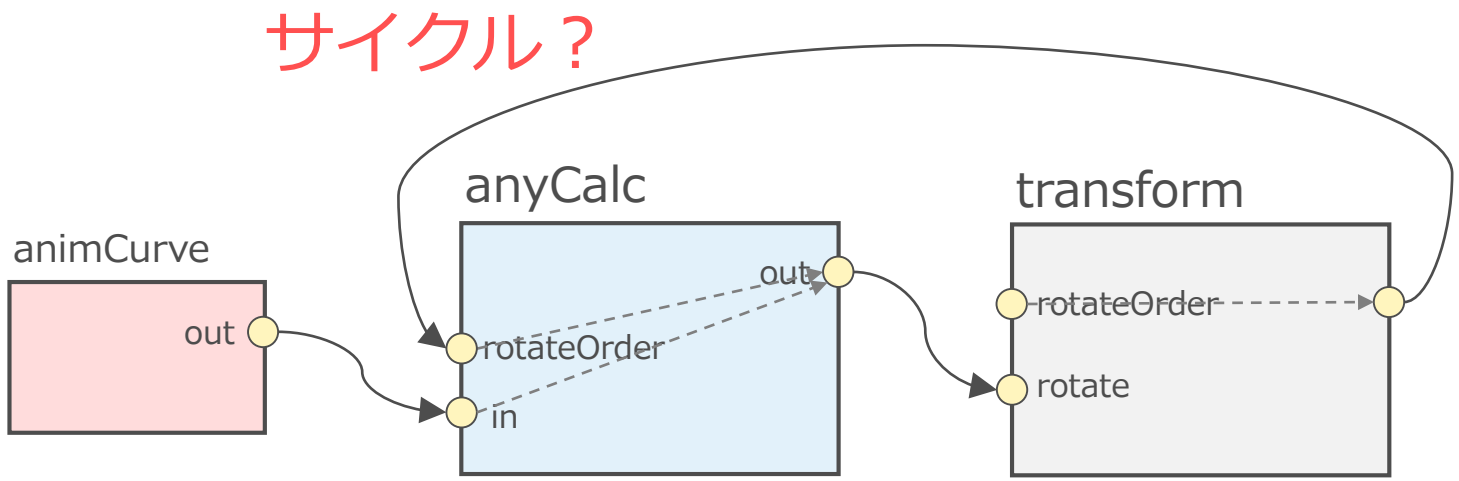
プラグレベルだとサイクルでなくても、
ノードレベルだとサイクルになってしまうことがある。

この説明は、おおむね間違いではないが、正確ではない。
もしその通りなら、EG ではかなりのものがサイクルになってしまう。

よくあるサイクル接続の例

たとえば、rotate を出力する何らかのノードを考えてみる。
出力先に合わせた出力ができるように rotateOrder を受ける。

もちろん DG ではサイクルではないが、ノードレベルではサイクルしている。

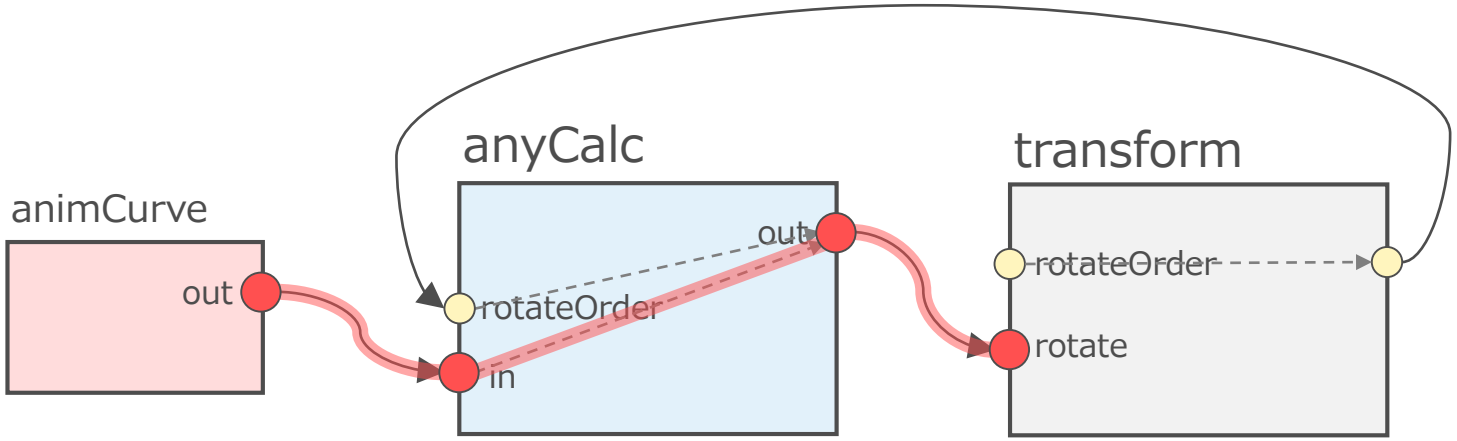


実は、これだけだと EG のサイクルにならない！

なぜ EG でサイクルにならないのか

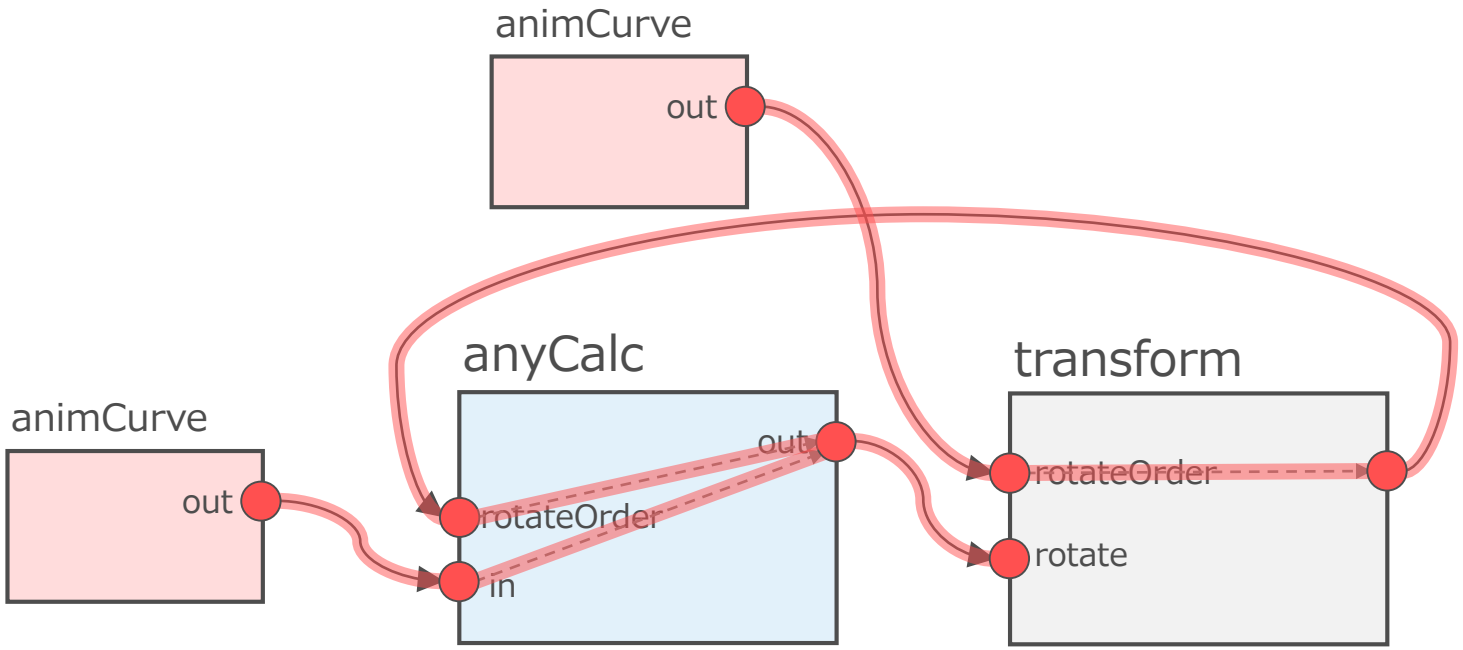
EG は起点（animCurve かタグ付きコントローラ）から実際に dirty 伝搬をさせて、ノードをトラッキングしながら作られる。

rotateOrder の返し接続は上流には依存していないため EG 生成ではピックアップされず、サイクルにはならない。



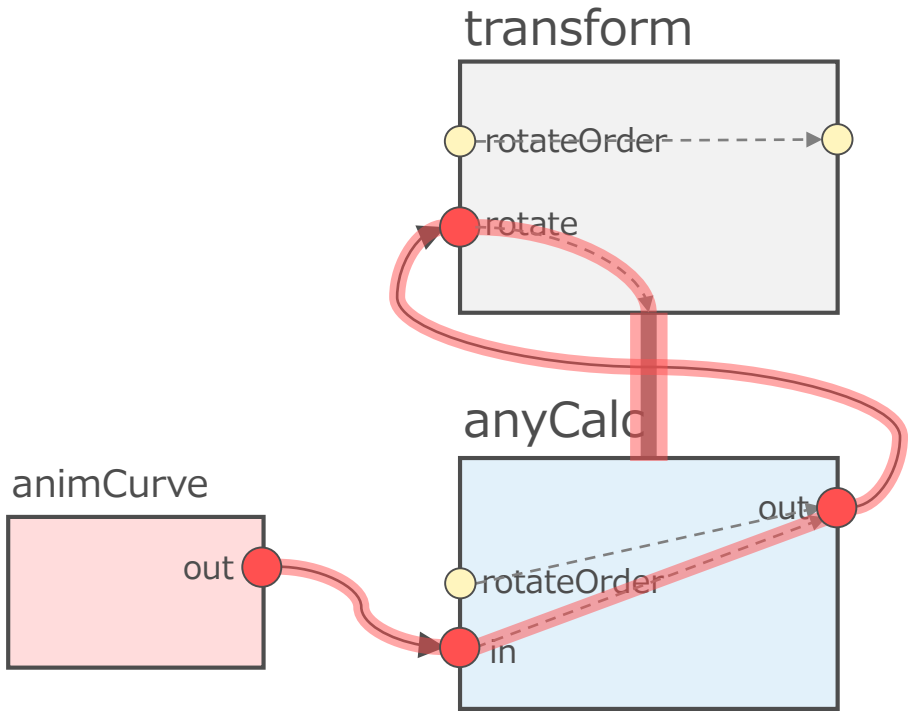
EG でサイクルになるケース – その 1

もし rotateOrder がアニメートされている場合、その dirty 伝搬もトラッキングされるため、EG でノード間の依存関係としてみるとサイクルとなる。



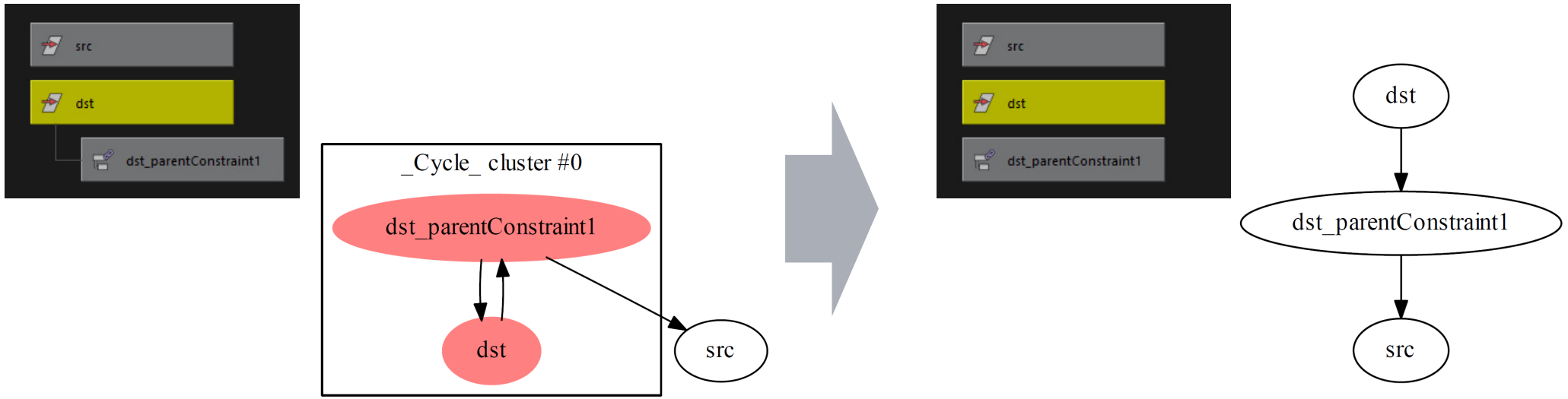
EG でサイクルになるケース – その 2

もし、そのノードが DAG ノードで、出力先ノードの子になっている場合、計算結果が実際に親の影響を受けるのなら本当にサイクルだが、そうでないとしても EG ではサイクルになる。
この場合、rotateOrder の返しの接続の有無などは関係ない。



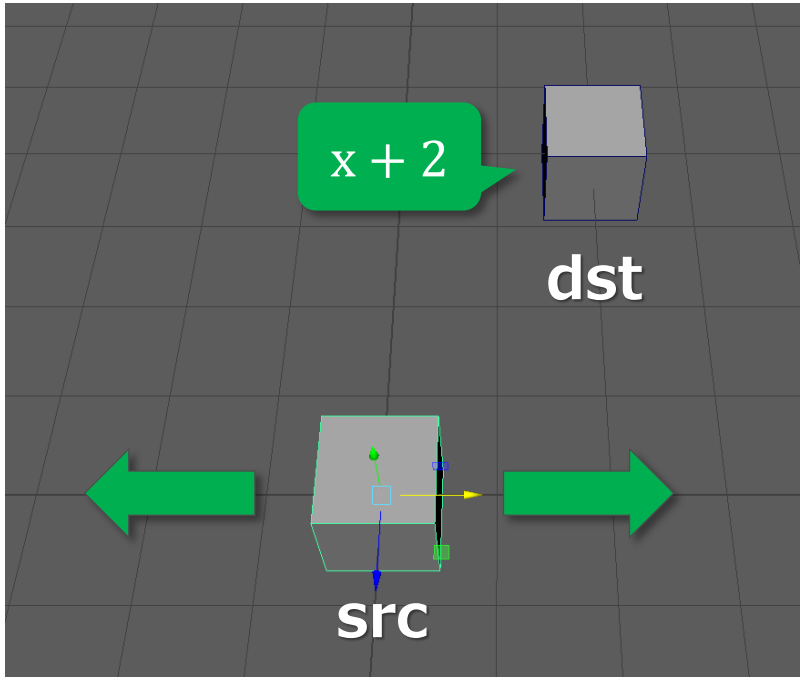
コンストレインの効率化

全種類のコンストレインノードがサイクルクラスターになってしまうのは、コンストレインが DAG ノードで、拘束対象の子になるから。



子であることは必須ではないので、兄弟にしてしまえば、サイクルクラスターにはならない。

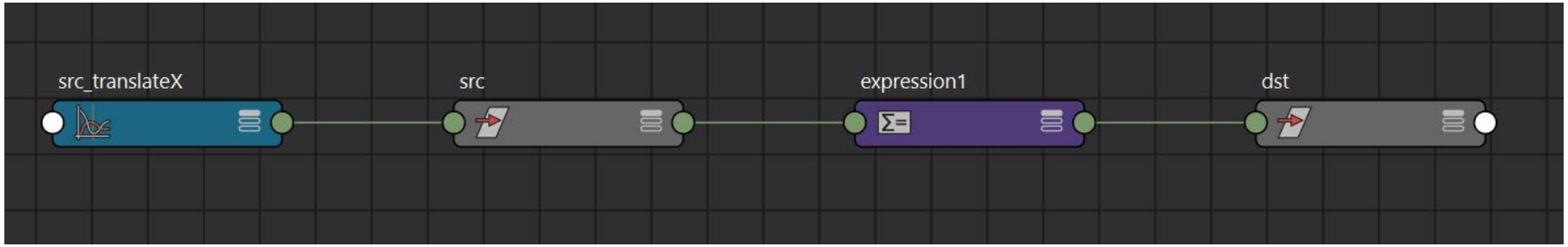
正しい EG が作られるエクスプレッション




```
dst.tx = src.tx + 2;
```

既に説明した通り、エクスプレッションは
このように書けば、コネクションが作られる。

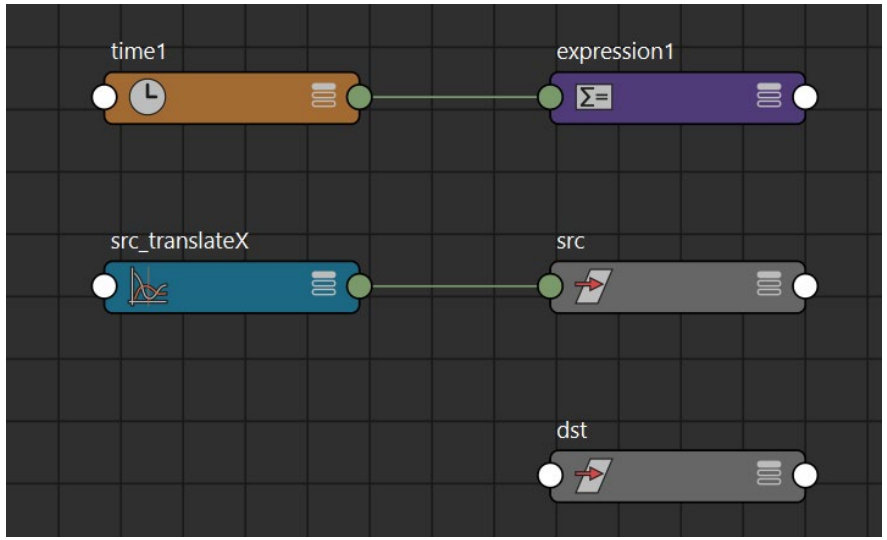
これならば、EG 構築は問題ない。



正しい EG が作られないエクスプレッション

```
setAttr dst.tx (`getAttr src.tx` + 2);
```

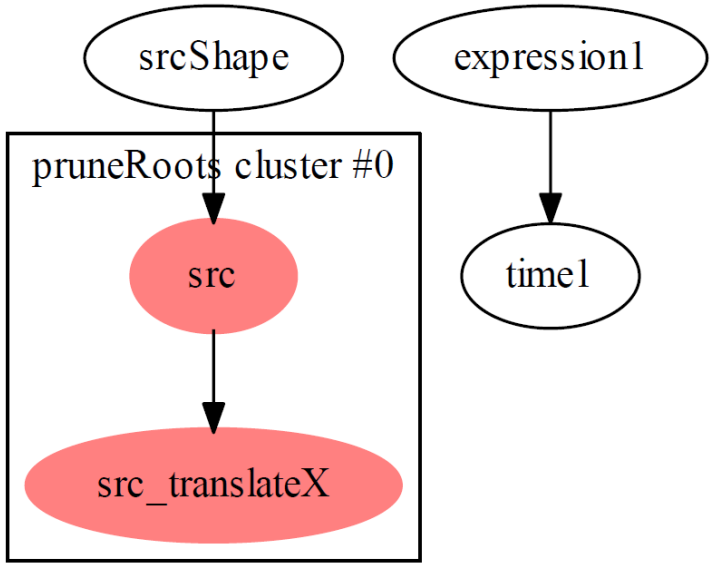
もし、このように書いてしまっているとコネクションが作られないため、
正しい EG が構築されない。



もちろん、本来は正しく修正すべきだが、
「DG に理想的でないシーン構造をパラレルでどう動かすか」の簡単な検証として、
この実装方針のまま無理やり動かすことを考えてみる。

エクスプレッションが動かない原因

修正前の EG はこうなっている。



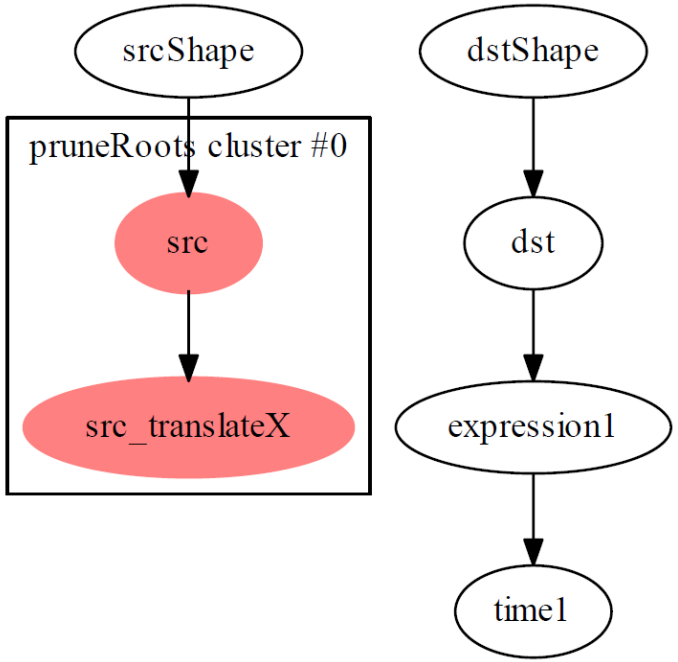
- expression ノードは、本当は良くないが Evaluation=“Always” の働きにより time 入力を持つため、EG に組み入れられていて問題ない。
- src ノードもアニメーションを持っているため、 EG に組み入れられていて問題ない。

dst ノードが EG に組み入れられていない点が問題。

動かないエクスプレッションの修正例

そこで、たとえば以下のように書いて、
何でも良いので expression から dst へのコネクションを作ってしまう。

```
setAttr dst.tx (`getAttr src.tx` + 2);
dst.dummyAttr = 0;
```



dst も EG に組み入れられれば、
エクスプレッションの Push 評価によって
アップデートされるようになる。

パラレル評価

プラグインノードの対応

対応すべきこと

DG の基本ルールに則った実装がされていれば、大抵そのまま動く。

少々の問題があったとしても、やるべきことは限られている。

- schedulingType() をオーバーライド。
- compute() で変わる部分を、必要に応じて対応。
- setDependentsDirty() を実装しているなら、その対応。

schedulingType() のオーバーライド

- デフォルトのままだと Serial なので、やや効率が悪い。
- 普通に問題ないプラグインなら Parallel とすべき。

compute() の対応

渡されるプラグの条件が少し変わる

DG では、実際に評価されたプラグが呼ばれるが、
EM では、コンパウンドアトリビュートは必ず最上位プラグで呼ばれる。

複数のノードの compute() が同時に呼ばれる

- ノードインスタンスの垣根を超えた共有リソースを持たないようにする。
  基本的には…、クラスメンバ変数なら問題ないが static 変数はまずい。
- 問題ないレベルに応じてスケジューリングタイプを決める。
  - 問題ない → **Parallel**
  - ノードタイプ内でリソース共有があり同時に呼ばないで欲しい → **GloballySerial**
  - 他のノードタイプとも同時に呼ばないで欲しい → **Untrusted**

setDependentsDirty() の対応

dirty 伝搬によって EG 構築が行われるが、
その後の評価中は dirty 伝搬されない。

つまり

setDependentsDirty は
EG 構築時に呼ばれるのみで、評価中は呼ばれない。

そのため

- 動的な Attribute Affects 定義の目的としては利用可能。
  EG 再構築を伴わない条件（アトリビュート値など）変化には対応できない。
- 入力値の変更 (dirty) のフックとしては利用できなくなる。

preEvaluation() と postEvaluation()

- setDependentsDirty() による dirty フックの代替となる。
- EM による評価の compute() 呼び出しの前後に呼ばれ、EG 構築のソースとなった dirty プラグを確認できる。
- setDependentsDirty() の実装はそのまま残しておくこと。
  - 動的な Affects 定義にはその手段しかない。
  - Parallel モードであっても、DG 評価は無くならない。

simpleCalc の対応

```
MStatus SimpleCalc::setDependentsDirty(const MPlug& plug, MPlugArray&)
{
    const MPlug dirtyPlug = plug.isChild() ? plug.parent() : plug;
    if (dirtyPlug == aInputA || dirtyPlug == aInputB) {
        isCacheValid = false;
    }
    return MS::kSuccess;
}

#if MAYA_API_VERSION >= 201600
SchedulingType SimpleCalc::schedulingType() const { return MPxNode::kParallel; }

MStatus SimpleCalc::preEvaluation(const MDGContext&, const MEvaluationNode& evalNode)
{
    if (isCacheValid) {
        // dirty なプラグをチェック。トップレベルのものに限られる。
        for (MEvaluationNodeIterator it=evalNode.iterator(); ! it.isDone(); it.next()) {
            const MPlug dirtyPlug = it.plug();
            if (dirtyPlug == aInputA || dirtyPlug == aInputB) {
                isCacheValid = false;
                break;
            }
        }
    }
    return MS::kSuccess;
}
#endif
```

スケジューリングタイプ。

setDependentsDirty()
と同じ処理。

simpleLocator の対応

```
MStatus SimpleLocator::setDependentsDirty(const MPlug& plug, MPlugArray&)
{
    const MPlug dirtyPlug = plug.isChild() ? plug.parent() : plug;
    if (dirtyPlug == aSize || dirtyPlug == localPosition || dirtyPlug == localScale) {
        valueDirty = true;
    }
    return MS::kSuccess;
}

#if MAYA_API_VERSION >= 201600
SchedulingType SimpleLocator::schedulingType() const { return MPxNode::kParallel; }

MStatus SimpleLocator::postEvaluation(const MDGContext&, const MEvaluationNode& evalNode, PostEvaluationType)
{
    if (! valueDirty) {
        // dirty なプラグをチェック。トップレベルのものに限られる。
        for (MEvaluationNodeIterator it=evalNode.iterator(); ! it.isDone(); it.next()) {
            const MPlug dirtyPlug = it.plug();
            if (dirtyPlug == aSize || dirtyPlug == localPosition || dirtyPlug == localScale) {
                valueDirty = true;
                break;
            }
        }
    }
    return MS::kSuccess;
}
#endif
```

スケジューリングタイプ。

setDependentsDirty()
と同じ処理。

ジオメトリのトポロジ変化の有無

Maya® 2019 より

ジオメトリの VP2 描画フェーズも EM とともに並列化

ただし、トポロジ変化がアニメーションされていない場合。

トラッキングトポロジ

Maya® は、可能な限り、シーンからトポロジ変化があるかどうかを読み取り、最適化の可否を決める。

- 標準ノードについては全てを認識している。
- プラグインノードでも、デフォーマーならトポロジ変化しないことが明白。
- 一般のプラグインノードでジオメトリ出力するものは、判断できないため、最悪ケースとして「トポロジを変化させる」ものとして扱われる。

トラッキングトポロジ

一般ノードでジオメトリを扱うものが「トポロジを変化させないこと」を Maya® に知らせる機能が追加された。

- MPxNode::isTrackingTopology() で true を返す。
- ジオメトリ出力の attributeAffects() 第3引数を false にする。

効果は大きいので、該当するものは積極的に対応させるべき。
DG モードでも効果がある。

パラレル評価

プラグインロケータの VP2 対応

さまざまな手法

MPxDrawOverride with MUIDrawManager

- MUIDrawable という図形や文字列などを手軽に描画する方法。
- 比較的簡単に実装できるが低速。

MPxDrawOverride with Low-level Drawing APIs

- OpenGL/DirectX やシェーダーを直接コーディングする方法。
- 高速だが、全ての Rendering Engine に対応するには全てを実装する必要がある。

MPxGeometryOverride おすすめ!!

- レンダーアイテムを作る方法。VertexBuffer や IndexBuffer や Shader を設定していく。
- 高速。さらに、インスタンスを多用する場合は VP2 設定の GPU Instancing が効く。

MPxSubSceneOverride

- サブシーンコンテナに多量のレンダーアイテムを追加していく方法。
- GPU Instancing 設定に関係なく多量のインスタンス描画が高速だが、そうでなければ MPxGeometryOverride より遅い。

devkit サンプルの紹介

MPxDrawOverride with MUIDrawManager

- footPrintNode

MPxDrawOverride with Low-level Drawing APIs

- rawfootPrintNode

MPxGeometryOverride おすすめ!!

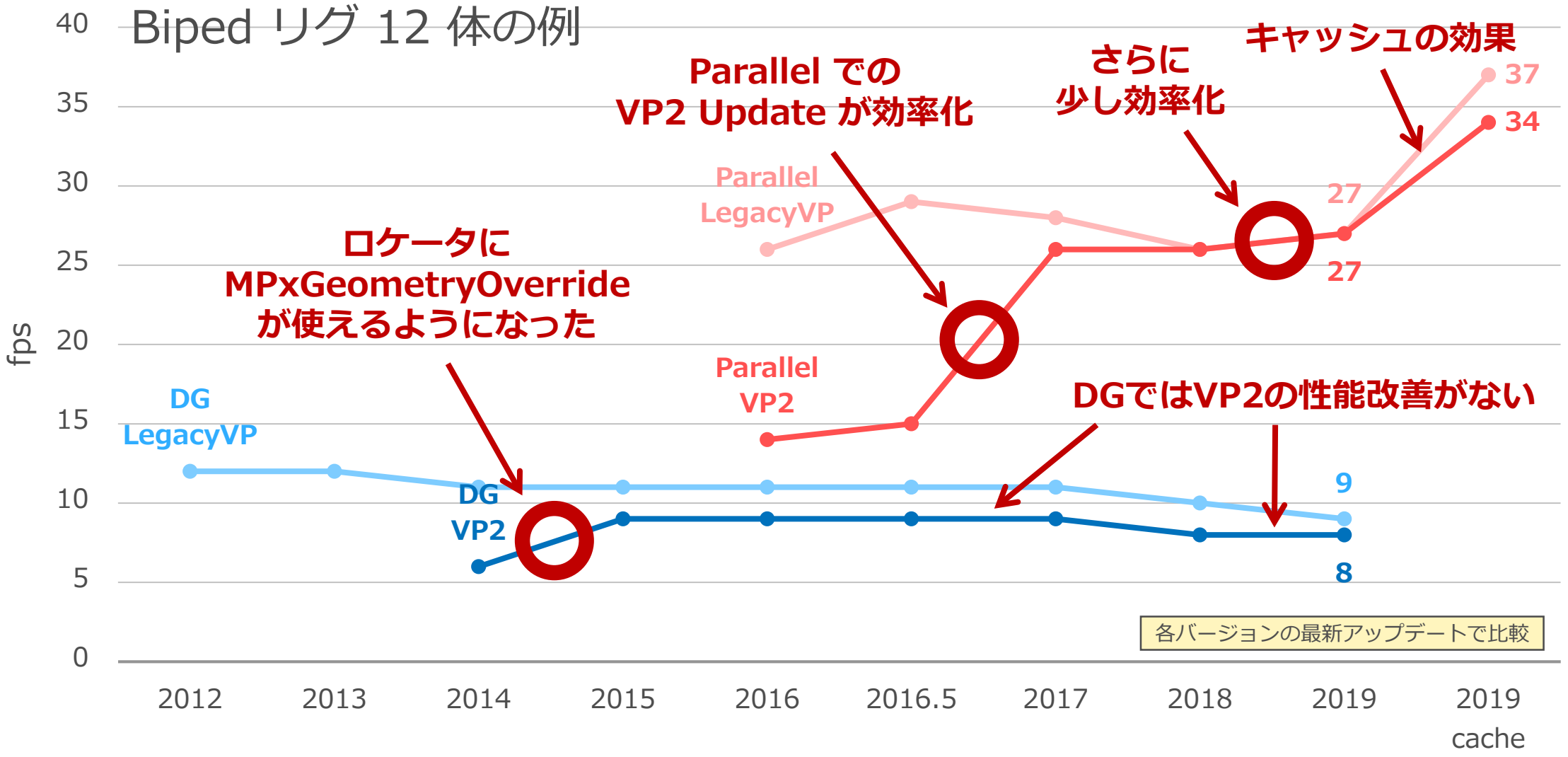
- footPrintNode_GeometryOverride
- footPrintNode_GeometryOverride_AnimatedMaterial

入力情報の更新やシェーダー管理などの考え方は、この２つの違いを理解して応用するのがおすすめ。また、2019.1 現在の仕様を正しく理解するには、特に後者を読む必要がある（2019 とも違う）。

MPxSubSceneOverride

- footPrintNode_SubSceneOverride

プラグインロケータの VP2 性能改善



バージョンによる挙動違いの注意

- 性能改善のターニングポイントでは、実装でも対応が必要。
  - 結構 #if ～ #endif だらけになる。
- マイナーバージョンでも細かく挙動が違う。
  - 特に 2017 と 2019 。
  - 全部対応するのは面倒なので、最新アップデート基準にするなど。
  - devkit サンプルもマイナーバージョンごとに違う。

MPxGeometryOverride の概要

ノード１つにつき、クラスインスタンスが１つ作られる。
レンダーアイテムを作って維持することが仕事なので、描画のたびには呼ばれない。
更新が必要なときに以下の更新フェーズが呼ばれる。

updatedDG()

- ジオメトリに関する情報をアトリビュートから得て保持しておく（ロケータサイズなど）。
- 本スライドの simpleLocator なら updateValues() を呼び出せば良い。

updateRenderItems()

- レンダーアイテムの追加とシェーダーのアサイン。

updateRenderItems()

- もし必要なら、MUIDrawable による簡易な描画を追加できる（ラベルの描画に便利）。

populateGeometry()

- 要求されたジオメトリ情報（IndexBuffer や VertexBuffer）を更新。

cleanup()

- 描画処理フェーズを終了。

ジオメトリ更新通知

2015 ～

Parallel だと遅かった。
おそらく更新フェーズが呼ばれ過ぎるから？

2017 Update3 ～

MPxNode::postEvaluation() で MRenderer::setGeometryDrawDirty() を
明示的に呼ぶことで更新通知する仕様に変わった。

2019

requiresUpdateRenderItems() と requiresGeometryUpdate() が追加。
MRenderer::setGeometryDrawDirty() を呼ぶよりも効率的に。

2019.1 ～

requiresUpdateRenderItems() の仕様が変更。
もう少し効率的に。

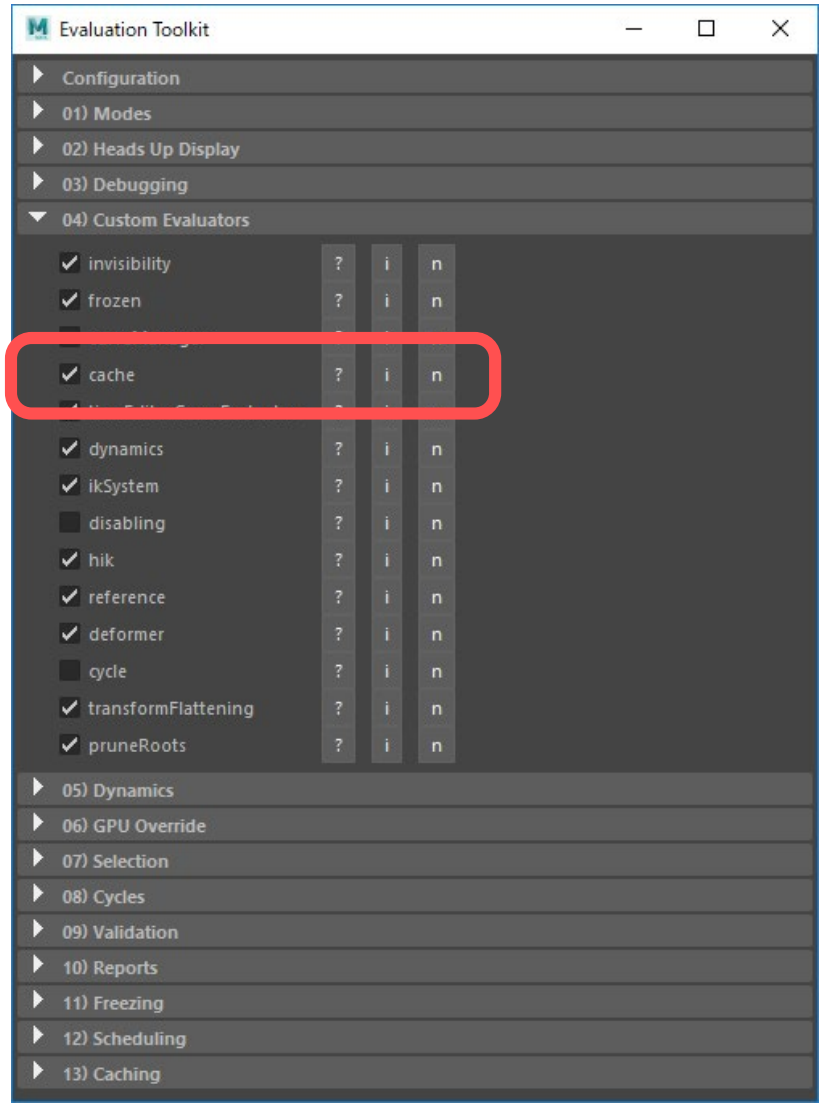
キャッシュプレイバック

概要

キャッシュプレイバックの概要

- アニメーションの開始から終了フレームまでの全フレームの評価結果をキャッシュする。
- キャッシュが完了すると、アニメーション再生は各フレームのキャッシュを取り出すだけとなり、シーン評価はされなくないので高速化される。
- ジオメトリについては VP2 の描画データもキャッシュできる。

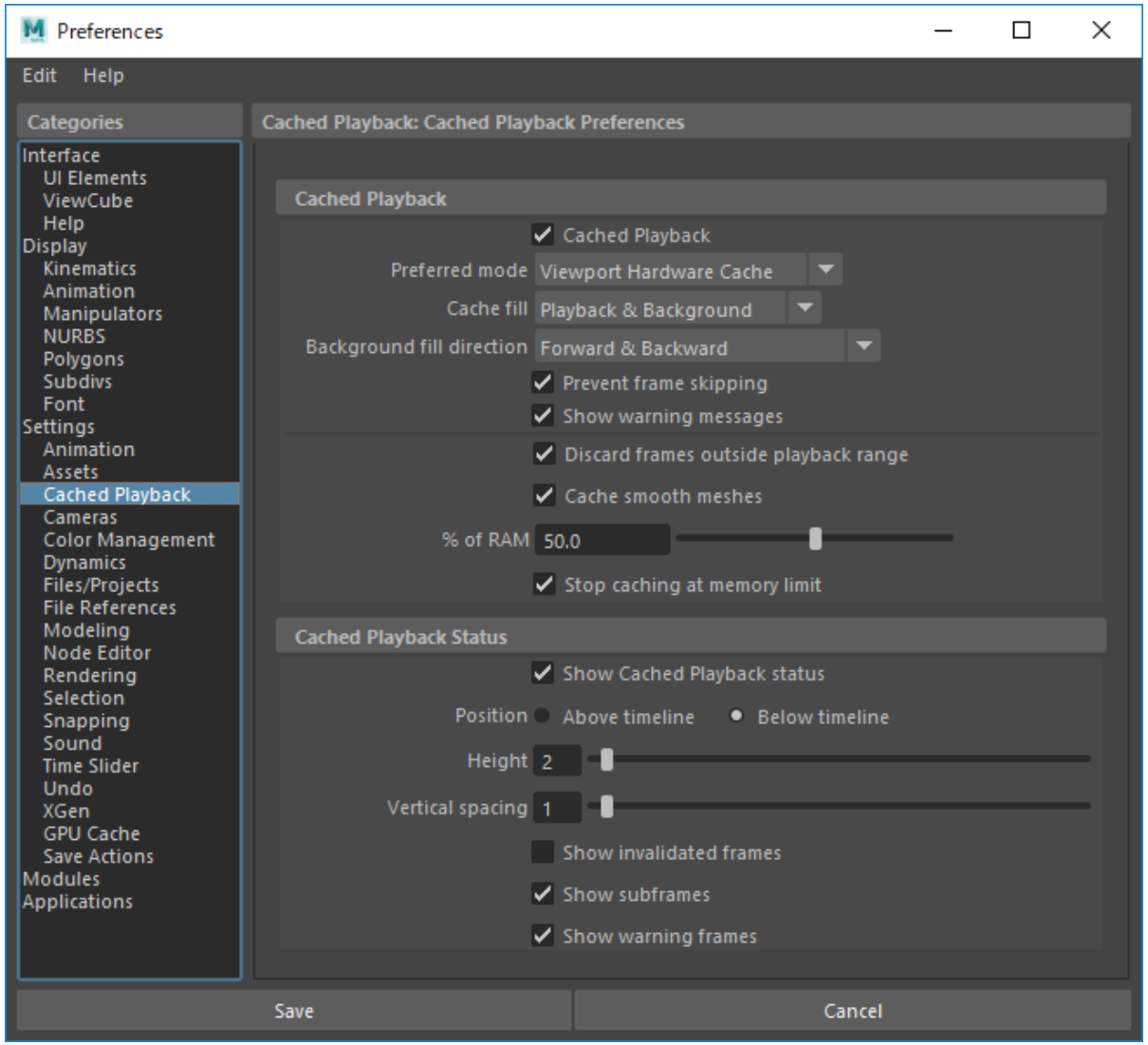
その正体

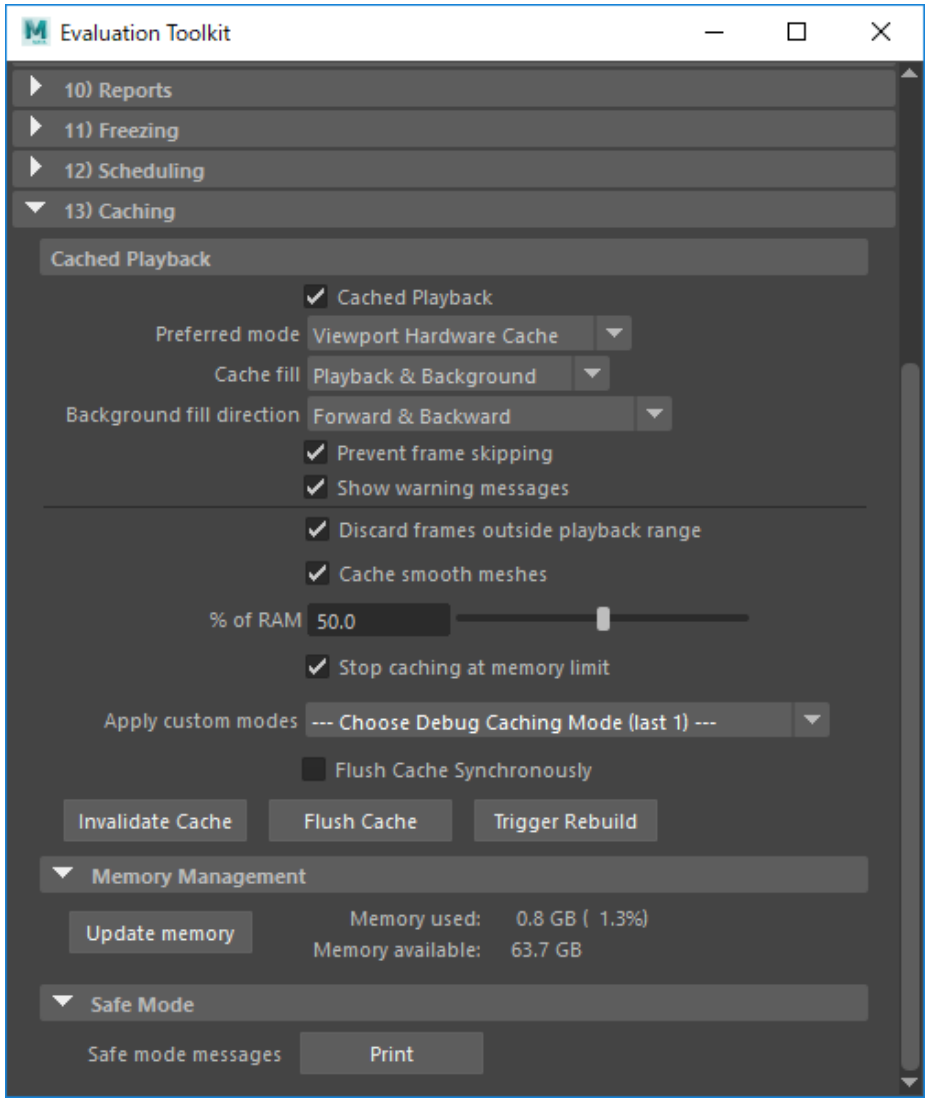


cache エバリュエータ

- cacheEvaluator というプラグインによる機能。
- 設定用の専用コマンド
  cacheEvaluator
- 補助 Python モジュール
  maya.plugin.evaluator

Preferences と Evaluation Toolkit





キャッシングモード（Preferred mode 設定）

Evaluation Cache デフォルト

- ノードの評価結果をキャッシュする。
  ビューポート（VP2）とは無関係で、様々なデータをキャッシュする。
- ジオメトリの描画にあたっては、キャッシュからのデータの復元と、それを VP2 用に変換・転送するコストがかかる。

Viewport Software Cache

ジオメトリの評価キャッシュを、VP2 用に変換した形でメインメモリに格納する。

描画時は、キャッシュからのデータ復元コストは発生しないが、データを GPU に転送するコストがかかる。

Viewport Hardware Cache

ジオメトリの評価キャッシュを、VP2 用に変換した形で GPU メモリに格納する。

描画時は、描画準備コストはほとんど発生しない。

Cache fill 設定

キャッシュ生成方法の設定。

Playback & Background

その名の通り Playback と Background 両方の手段を許可する。

Playback

アニメーション再生などで実際にフレームが評価されたときにキャッシュ生成する。サブフレームのキャッシュは、実際の評価、つまり Playback でのみ生成される。

Background

アイドル中にバックグラウンドで整数フレームを評価しキャッシュ生成する。非対応のプラグインによる問題が起きる場合は Playback にすると良い。

Cache HUD とキャッシュ状況の確認

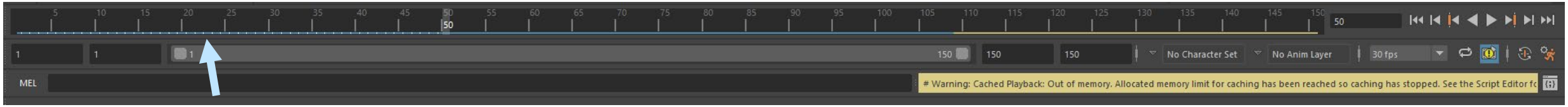
Cache:	On (sync+async)
Memory Limit:	Memory low (48.7% / 50.0%)
GPU Memory Limit:	OK (0.0%)

- キャッシュ機能の状態（Cache fill 設定）
- メモリ設定や使用状況

  デフォルト設定では、使用メモリの上限はシステムメモリの 50% までに抑えられている。

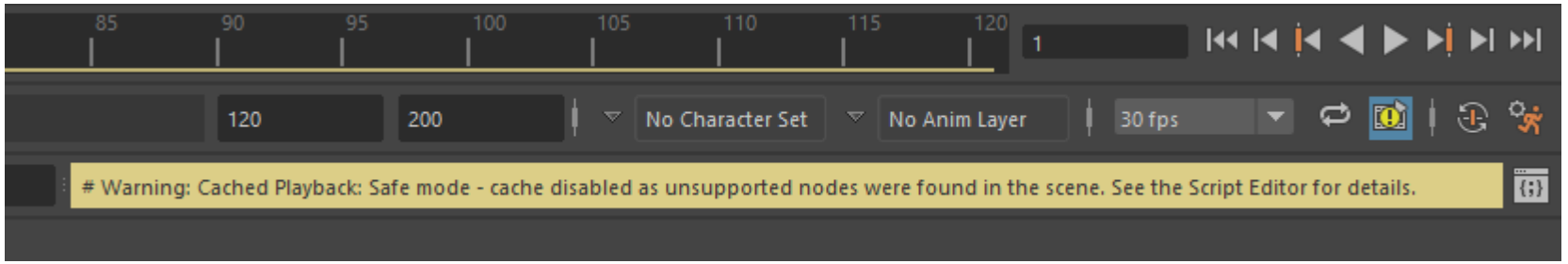
キャッシュされた範囲

メモリ制限で
キャッシュされなかった範囲



白い点は
キャッシュされたサブフレーム

セーフモードと非対応ノード

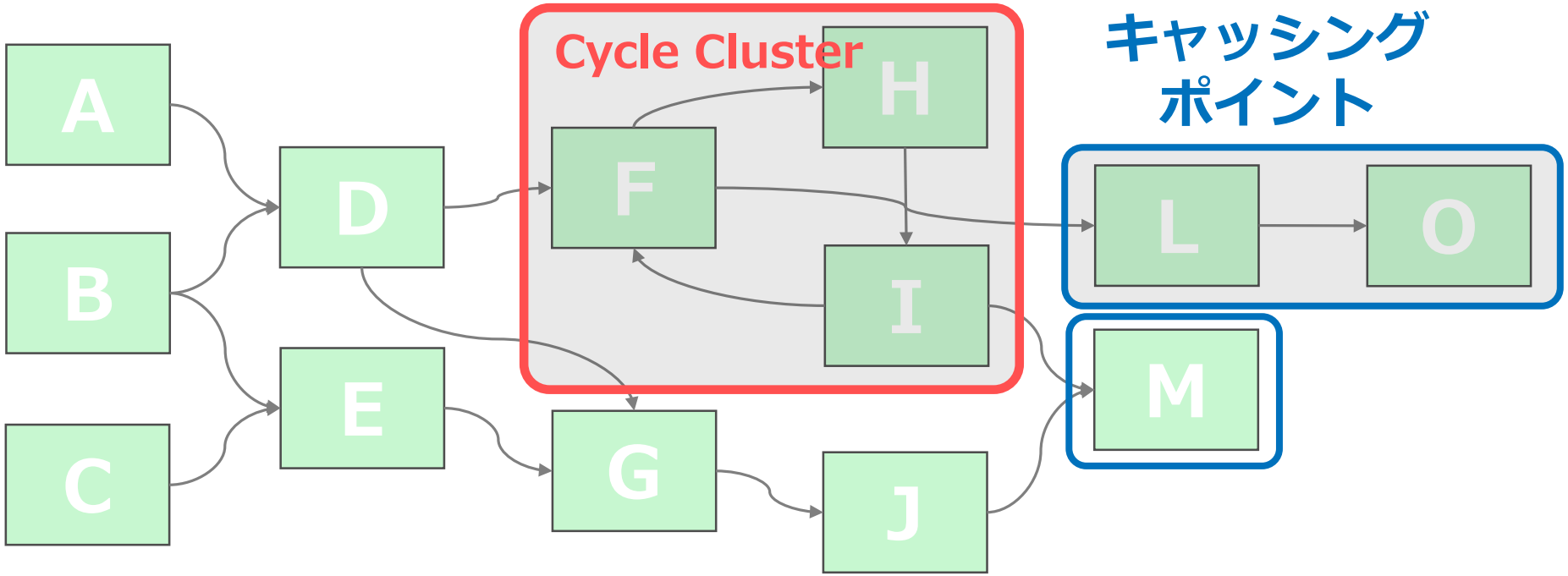


- キャッシュプレイバックに非対応のノードがシーン中に検出されるとセーフモードに移行し、キャッシュが無効になる。
- サポートされていないもの
  - Time Editor や Trax Editor
  - シミュレーション系の多く

    Bifrost, XGen, Muscleの一部, Jiggle, nHair, …
  - MASH の一部
  - その他や詳細は、公式ドキュメントを参照のこと

キャッシングポイント

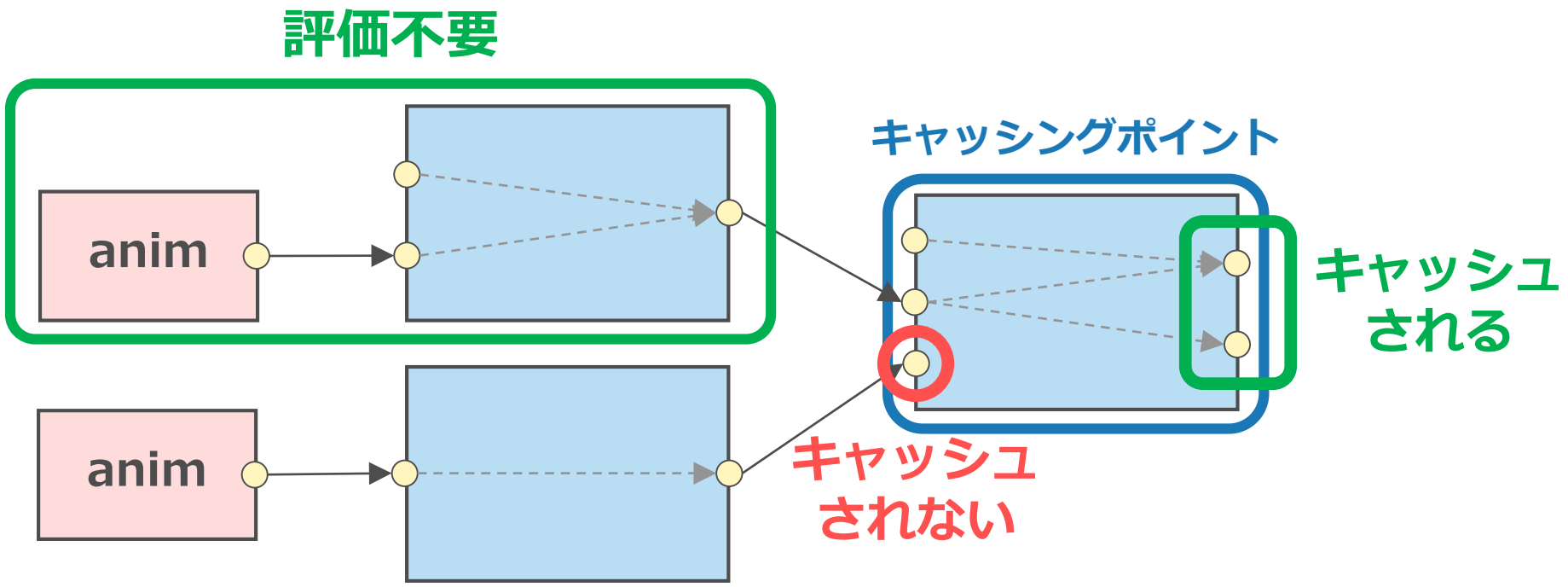
- 全てのノードがキャッシュされるわけでない。少ないほど良い。
  キャッシュされるノードを「キャシングポイント」という。
- transform やジオメトリなど、ユーザーへのフィードバックに必要な箇所が
  キャッシングポイントとなる。ネットワークの最下流であれば理想的。



キャッシュされるアトリビュート

基本的に、ノードの出力がキャッシュされる。
入力のみで、出力への影響がないものはキャッシュされない。

キャッシュ再生時は全ての Push 評価はスキップされる。



キャッシングポイントの確認

キャッシングポイントの確認

```
cmds.cacheEvaluator(q=True, cps=True)
```

ノード名のリストを得られる。

キャッシュの状態の確認

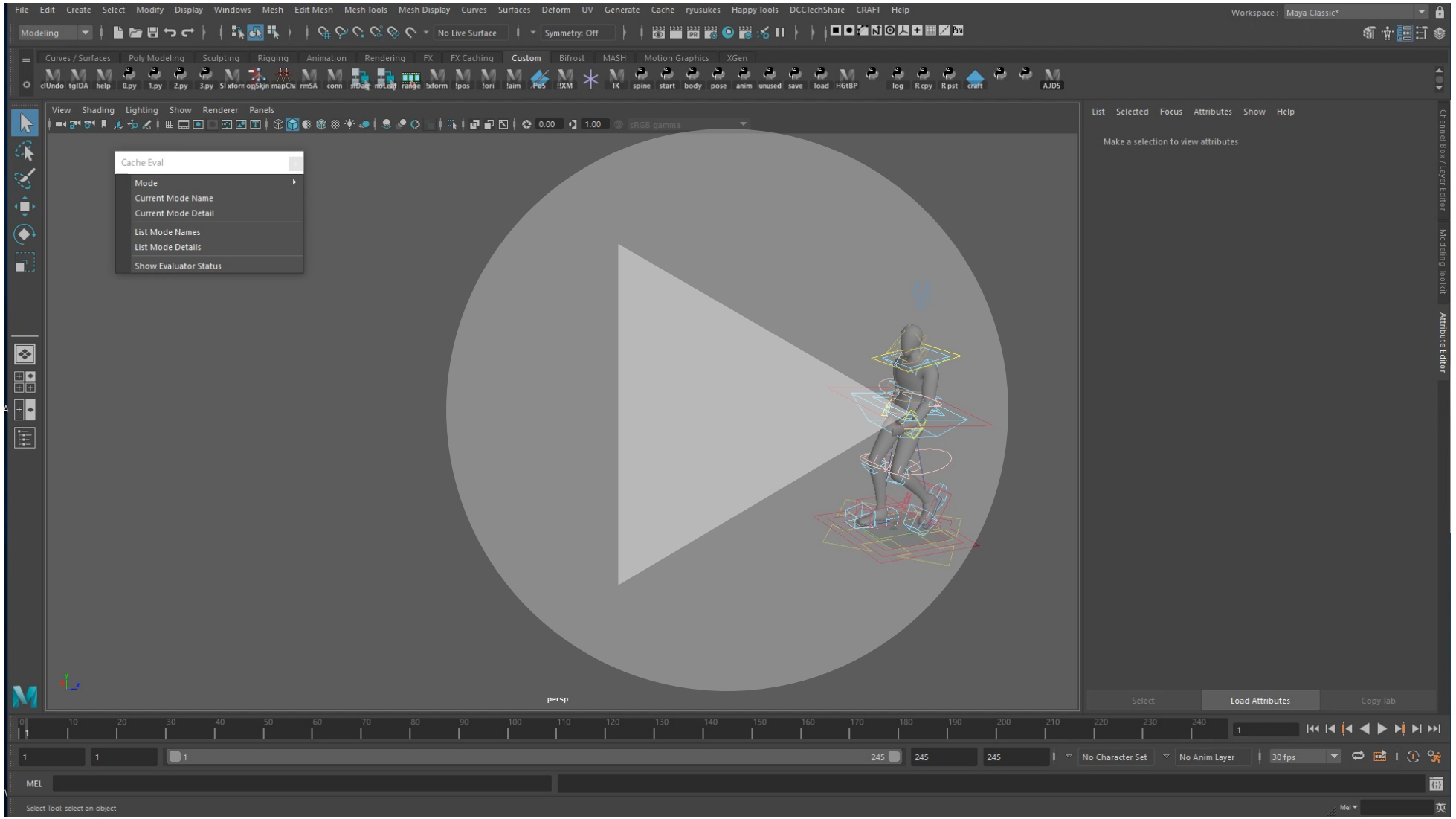
```
print(cmds.evaluator(n='cache', q=True, i=True))
```

ノードタイプごとの数や使用メモリのレポート。

キャッシュプレイバック

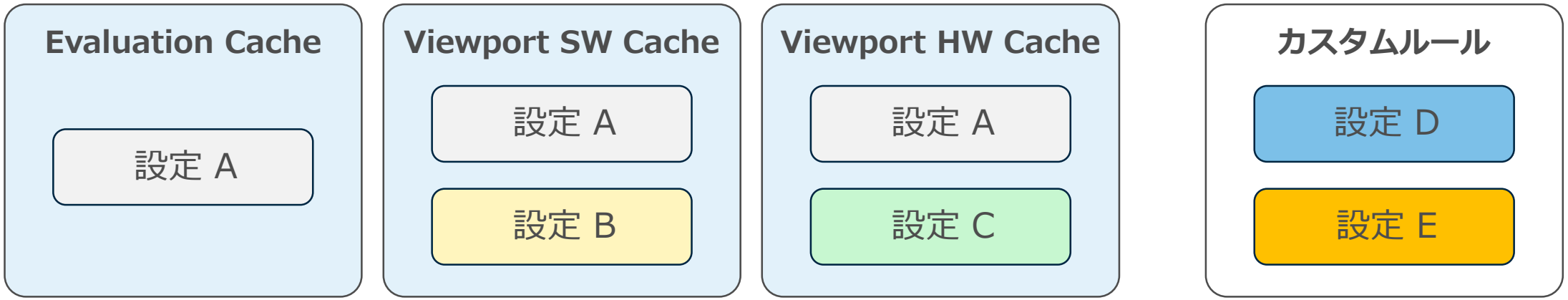
キャッシングルールの設定

正常動作しないプラグインロケータ



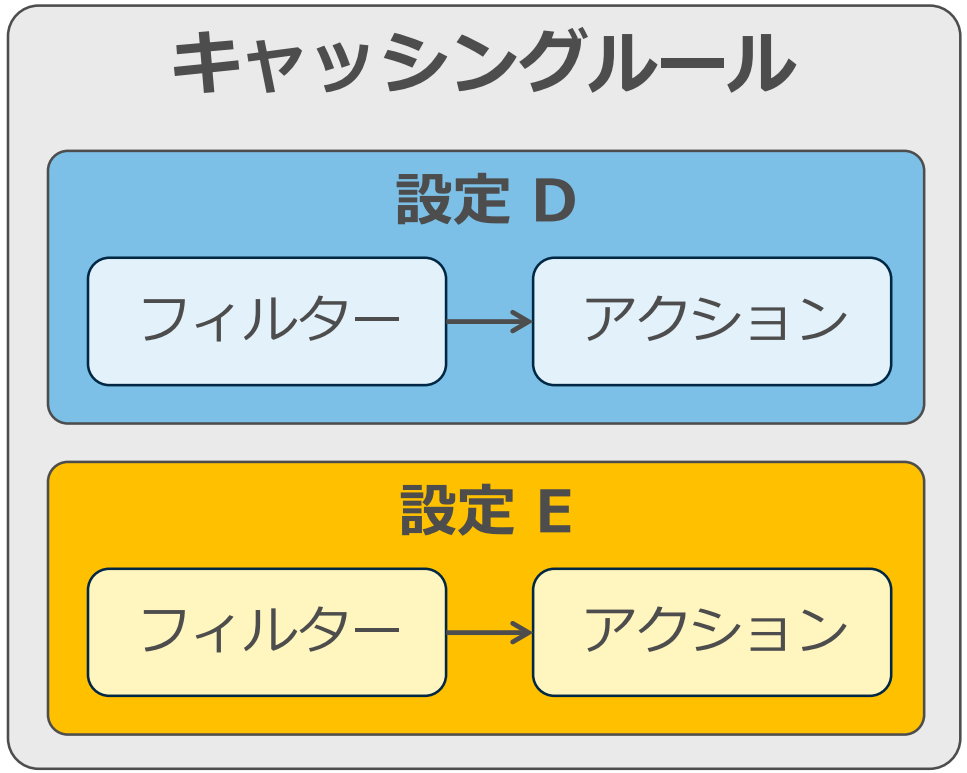
キャッシングルール

- 「何を」「どのように」キャッシュするのかという設定を「キャッシングルール」という。
- 通常、一連の設定をまとめて１つのルールとする。ビルトインの３つのキャッシングモードは、それを選択可能にしたもの。



キャッシングモード（ビルトインルール）

一般的なキャッシングルール設定の概念



最も一般的な設定は
「フィルター」と
「アクション」の組み合わせ。

フィルター

ノードタイプをテストする。

アクション

テストに合格したノードに適用するもの。

一連の設定が 1 つの
「キャッシングルール」となる。

Evaluation Cache ルールの内容

```
cmds.cacheEvaluator(resetRules=True)
```

ルール設定を開始。

```
cmds.cacheEvaluator(
    newFilter='evaluationCacheNodes',
    newAction='enableEvaluationCache')
```

標準のフィルター evaluationCacheNodes とともに「評価キャッシュ」を有効化。

```
cmds.cacheEvaluator(newRule='customEvaluators')
```

ルール設定を終了。

Viewport Software Cache ルールの内容

```
cmds.cacheEvaluator(resetRules=True)

cmds.cacheEvaluator(
    newFilter='evaluationCacheNodes',
    newAction='enableEvaluationCache')

cmds.cacheEvaluator(
    newFilter='vp2CacheNodes',
    newAction='enableVP2Cache',
    newActionParam='useHardware=0')

cmds.cacheEvaluator(newRule='customEvaluators')
```

Evaluation Cache と同じ。

標準のフィルター
vp2CacheNodes とともに
「VP2 キャッシュ」を有効化。
ハードウェアは使わない。

「評価キャッシュ」と「VP2 キャッシュ（ソフトウェア）」の併用

Viewport Hardware Cache ルールの内容

```
cmds.cacheEvaluator(resetRules=True)

cmds.cacheEvaluator(
    newFilter='evaluationCacheNodes',
    newAction='enableEvaluationCache')

cmds.cacheEvaluator(
    newFilter='vp2CacheNodes',
    newAction='enableVP2Cache',
    newActionParam='useHardware=1')

cmds.cacheEvaluator(newRule='customEvaluators')
```

「VP2 キャッシュ」を有効化。
useHardware=1 だけが違う。

「評価キャッシュ」と「VP2 キャッシュ（ハードウェア）」の併用

Python モジュールによるルール設定

```
from maya.plugin.evaluator.CacheEvaluatorManager import CacheEvaluatorManager

# Viewport Hardware Cache の設定
CacheEvaluatorManager().cache_mode = [
    dict(
        newFilter='evaluationCacheNodes',
        newAction='enableEvaluationCache'),
    dict(
        newFilter='vp2CacheNodes',
        newAction='enableVP2Cache',
        newActionParam='useHardware=1'),
    dict(newRule='customEvaluators'),
]
```

resetRules の指定は不要。

「評価キャッシュ」を有効化。

「VP2 キャッシュ」を有効化。

ルール設定を終了。

ルール設定のコマンドをラップした Python モジュールの使用例。
一連のコマンド呼び出しの引数辞書をリストにして代入することでコマンドが呼び出される。これがキャッシングモードの正体。

プラグインロケータが正常動作しない理由

- プラグインによるカスタムシェイプは、キャッシングポイントにしないと、描画データが更新されない。
- 標準フィルター evaluationCacheNodes では、カスタムロケータが除外されるようになっているのが原因。
- カスタムロケータを指定するフィルター設定を追加すれば良い。

nodeTypes フィルター

「除外」か「追加」をするノードタイプを列挙できる。

```
# nodeTypes フィルターの使用例
cmds.cacheEvaluator(
    newFilter='nodeTypes',
    newFilterParam='types=-foo,+bar,+baz‘,
    newAction='enableEvaluationCache')
```

上の例は、ノードタイプ foo を除外し、bar と baz を追加する意味。

ノードタイプは列挙した順に検査され、マッチすると検査を抜ける。
そのため、foo が bar や baz の派生タイプならば、このように先に指定しないと効果がないので注意。

evaluationCacheNodes フィルターのノードタイプ

cacheEvaluator コマンドのマニュアルには、
evaluationCacheNodes に相当する nodeTypes 指定のサンプルが示されている。

```
cmds.cacheEvaluator(
    newFilter='nodeTypes',
    newFilterParam='types=(省略),-THlocatorShape,+locator,(省略)',
    newAction='enableEvaluationCache')
```

長いので、ロケータの部分だけ抜粋すると、 THlocatorShape がカスタムロケータを意味していて、それが除外されている。

実際はバグがあり TH??? 系のタイプ指定は認識されないので、サンプル通りの指定をすると –THlocatorShape は効かず、問題が解決してしまうが。

vp2CacheNodes フィルターのノードタイプ

cacheEvaluator コマンドのマニュアルには、
vp2CacheNodes に相当する nodeTypes 指定のサンプルが示されている。

```
cmds.cacheEvaluator(
    newFilter='nodeTypes',
    newFilterParam='types=+mesh,+nurbsCurve,+bezierCurve,+nurbsSurface',
    newAction='enableVP2Cache',
    newActionParam='useHardware=0')
```

「VP2キャッシュ」は一部のノードタイプでしかサポートされていないため、ビルトインルールのように「評価キャッシュ」を併用するのが一般的。

「VP2キャッシュ」が有効になったノードタイプは「評価キャッシュ」の対象からは取り除かれる。

プラグインロケータを使用可能にする

```
CacheEvaluatorManager().cache_mode = [
    dict(
        newFilter='evaluationCacheNodes',
        newAction='enableEvaluationCache'),
    dict(
        newFilter='nodeTypes',
        newFilterParam='types=+locator',
        newAction='enableEvaluationCache'),
    dict(newRule='customEvaluators'),
]
```

評価キャッシュに
locator 派生タイプを追加。

通常の Evaluation Cache モードの設定に、カスタムロケータのサポートを追加する設定の例。他のモードでもこのアクションを追加すれば良い。

nodeTypes フィルターでは、バグのため +THlocatorShape は認識されないので +locator を指定している。

ビルトインルールに追加する例

```
try:
    import maya.plugin.evaluator.CacheEvaluatorManager as mgr

    def nodeTypesFilter(types):
        return {
            'newAction': 'enableEvaluationCache',
            'newFilter': 'nodeTypes',
            'newFilterParam': 'types=' + types,
        }

    def insertAction(definition, filters):
        # 繰り返し実行しても問題ないように、重複設定を回避。
        definition[-1:1] = [x for x in actions if x not in definition]

    rules = [
        nodeTypesFilter('+locator'),
    ]

    insertAction(mgr.CACHE_STANDARD_MODE_EVAL, rules)
    insertAction(mgr.CACHE_STANDARD_MODE_VP2_HW, rules)
    insertAction(mgr.CACHE_STANDARD_MODE_VP2_SW, rules)

except:
    pass
```

カスタムロケータを使用可能にする
追加ルールのアクション。

３つのビルトインルールに
同じ追加アクションを挿入する。

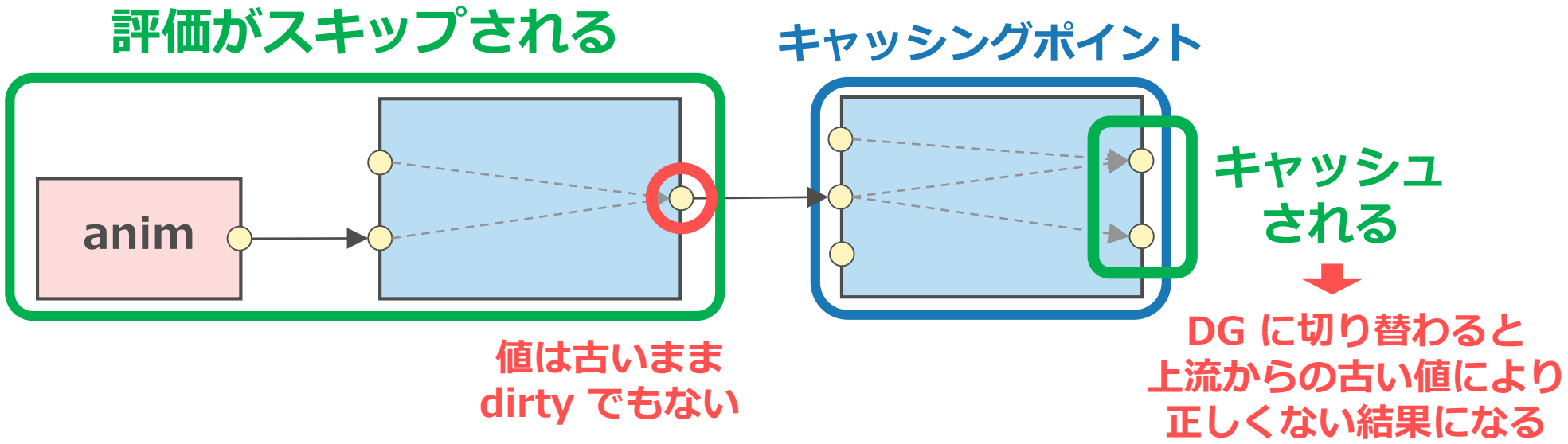
ビルトインルールの変更は自己責任でお願い致します

DG 評価発生時の dirty 不整合の問題事例

Manipulation 設定が OFF の場合、コントローラを操作すると DG 評価となる。
キャッシュされた状態で DG 評価になると、正しくない結果が生成されることがある。

これは、キャッシュされていない上流ノードが dirty 伝搬も評価もスキップされた結果、値は古いままで、且つ dirty にもなっていない状態になるため。

簡単な回避策は、全てのノードをキャッシュすることだが、それは避けたい。



downstreamNodeTypes フィルター

あるノードの１つ上流（コネクションか DAD 親）のノードにマッチさせるフィルター。上流と下流ともノードタイプリストを指定できる。

```
cmds.cacheEvaluator(
    newFilter='downstreamNodeTypes',
    newFilterParam='types=-animCurve,+node downstreamTypes=+transform,+locator',
    newAction='enableEvaluationCache')
```

コントローラとそのシェイプ（transform か locator 派生ノード）の１つ上流の animCurve 以外のノード全てをキャッシュ対象に加えている。

これで問題は解決したが、キャッシングポイントが増えすぎてしまうため、再考の余地はあるかもしれない。　過信はしないでください。

ビルトインルールにさらに追加する例

```
try:
    import maya.plugin.evaluator.CacheEvaluatorManager as mgr

    def nodeTypesFilter(types):
        return {
            'newAction': 'enableEvaluationCache',
            'newFilter': 'nodeTypes',
            'newFilterParam': 'types=' + types,
        }

    def downstreamNodeTypesFilter(types, downstreamTypes):
        return {
            'newAction': 'enableEvaluationCache',
            'newFilter': 'downstreamNodeTypes',
            'newFilterParam': 'types=' + types + ' downstreamTypes=' + downstreamTypes,
        }

    def insertAction(definition, actions):
        # 繰り返し実行しても問題ないように、重複設定を回避。
        definition[-1:1] = [x for x in actions if x not in definition]

    rules = [
        nodeTypesFilter('+locator'),
        downstreamNodeTypesFilter('-animCurve,+node', '+transform,+locator'),
    ]

    insertAction(mgr.CACHE_STANDARD_MODE_EVAL, rules)
    insertAction(mgr.CACHE_STANDARD_MODE_VP2_HW, rules)
    insertAction(mgr.CACHE_STANDARD_MODE_VP2_SW, rules)

except:
    pass
```

ビルトインルールの変更は自己責任でお願い致します。

特に、downstreamNodeTypesFilter による問題解決手法は検証が不十分であるため、過信しないでください。

カスタムフィルター

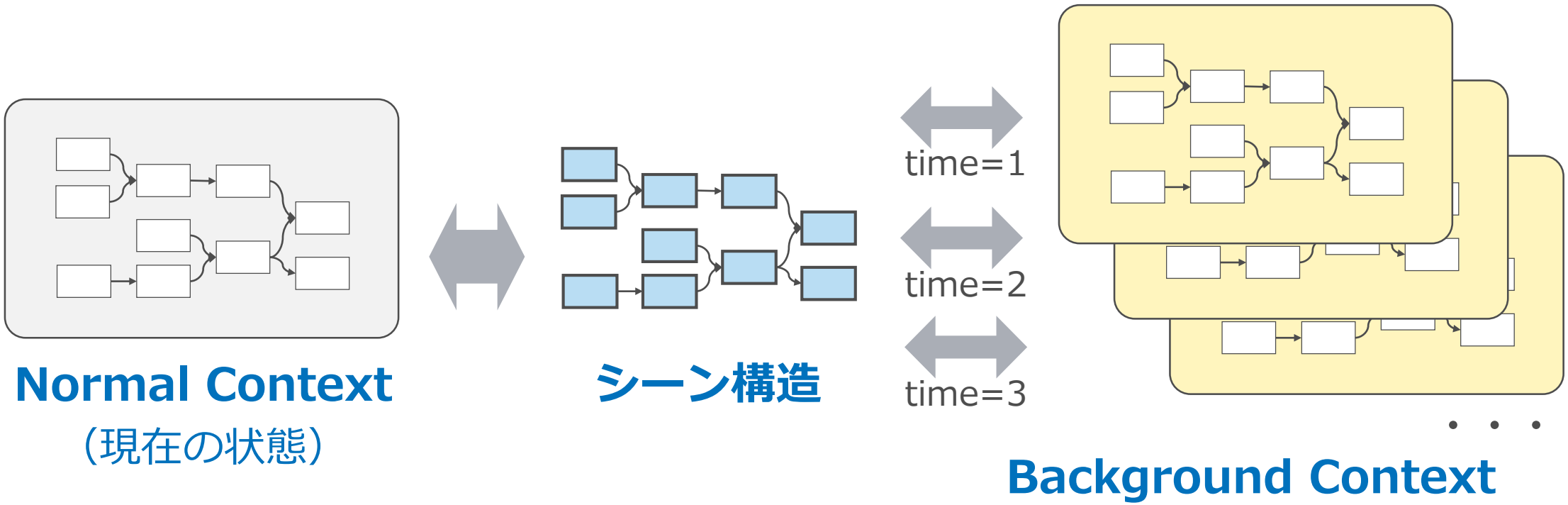
- API で独自の「フィルター」を追加することもできる。
- MPxCacheConfigRuleFilter 派生クラスを書く。
- devkit にごく簡単なサンプル nameFilter がある。
- 柔軟で効率的なキャッシングポイント決定のための手段になる。

キャッシュプレイバック

プラグインの対応

評価コンテキスト

- １つのシーンを、複数の異なる時間で評価するしくみ。
- バックグラウンド評価では、現在の Maya® の状態とは別のコンテキストで、各フレームを評価する。



MDGContext

- 評価コンテキストに相当する API クラス。

  遙か昔から存在するが、表立っては使われてこなかった。キャッシュプレイバックによって、公式に各プラグインの責任が増えた形になる。

- 通常のコンテキストは isNormal()=true となり、MTime は内包しない。
- バックグラウンドコンテキストは isNormal()=false となり、評価フレームの MTime を内包する。
- 2018 で「カレント」の概念が追加されている。バックグラウンド評価中の compute() 呼び出しでは、各フレームのコンテキストがカレントとなる。

プラグインノードのメソッド呼び出しサイクル

キャッシュ生成時

preEvaluation(), compute(), postEvaluation() が順に呼ばれる。

- Playback の場合、通常のコンテキストで呼ばれる。
- Background の場合、評価フレームのコンテキストで呼ばれる。

キャッシュ再生時

- キャッシングポイントに限り、postEvaluation() のみが通常のコンテキストで呼ばれる。
- キャッシングポイント以外は何も呼ばれない。

複数のコンテキストが許容される世界

単一のコンテキストしか存在しない世界：

- ノード内部に現在の状態を保存する実装をしても問題なかった。
  つまり、クラスメンバ変数に状態をキャッシュすることができた。
- パラレルであっても、ノードインスタンス間でのリソース共有がなければ問題なかった。

複数のコンテキストが許容される世界：

- 異なるコンテキストで同一ノードが同時に評価される可能性。
- クラスメンバ変数での状態の保存は、コンテキスト間で競合する。

どう対処すべきか

- 公式ドキュメントによると

  (Maya® Cached Playback Context-specific Data)

  - 途中の状態の保存にもアトリビュートを利用する。
    アトリビュート値はコンテキスト中のデータブロックに保存される。
  - MTime か「Normalを表す値」をキーとするハッシュテーブルに保存する。アトリビュートとして保存しにくいようなデータの場合に。

- devkit サンプル simpleEvaluationNode の場合
  - メンバ変数に「高価な計算」の中間値を保存している。
  - 2019 では「Normal でないコンテキスト」の場合は、中間値を利用せずに計算する実装が加えられている。

simpleCalc と simpleLocator をどうするか

とりあえず何もしない。

simpleCalc

- キャッシュプレイバックでは compute() 呼び出しが競合しないため、実装を変える必要はない。

simpleLocator

- キャッシングポイントの postEvaluation() は呼ばれるため、そこで内部値を無効化する処理は問題ない。
- ただし、そこに入力コネクションが有る場合、キャッシュ再生時はそこだけ Pull 評価となる。

複数コンテキストによる compute()

- 公式ドキュメントによると

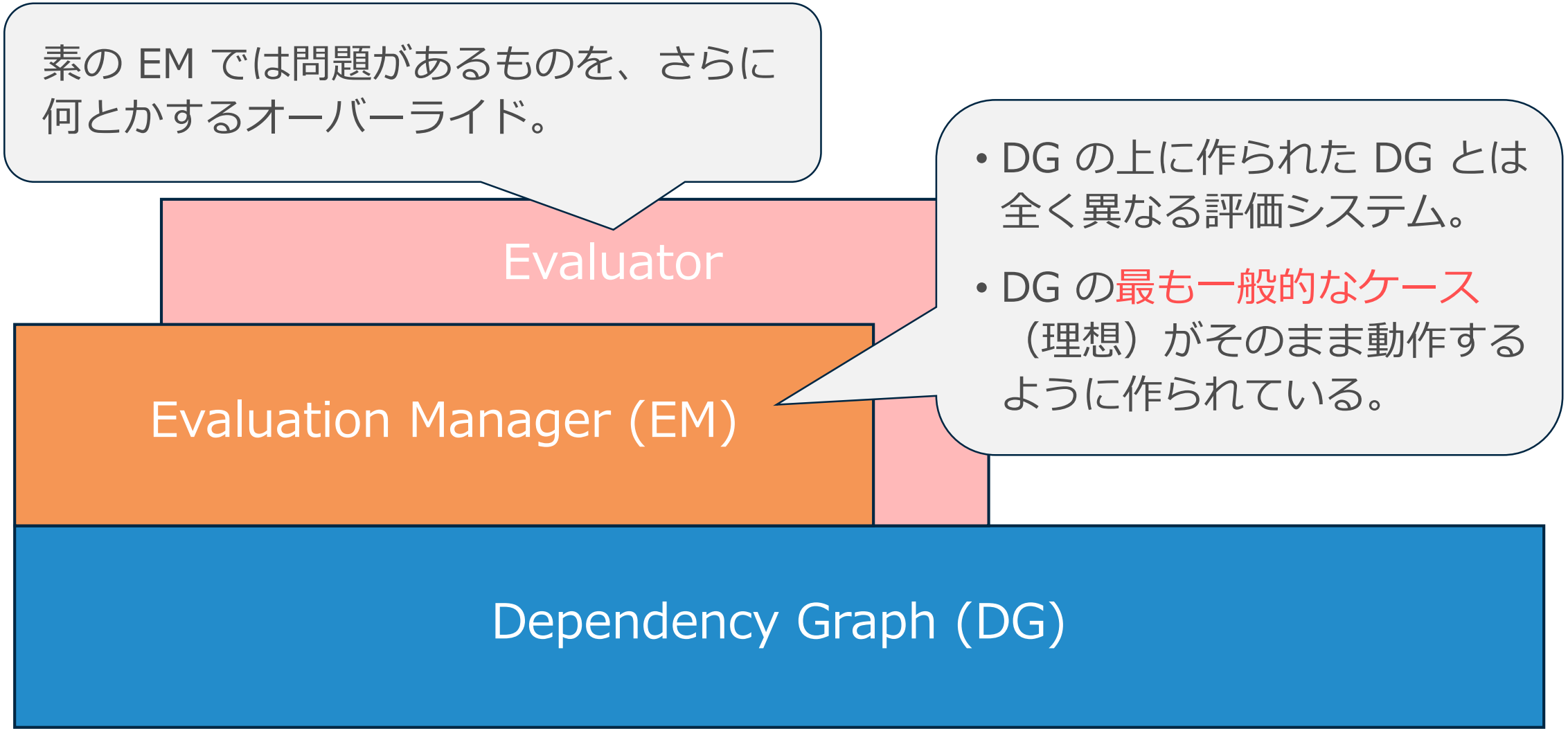
  「ノードは異なるコンテキストを持つ複数のスレッドによってアクセスされる可能性があるが、評価エンジンは１つのノードが複数のコンテキストによって同時に評価されないように制御するため、排他制御は必要ない」

- 結局、面倒な対応は不要なのか？
  - 文面通りに受け取れば、キャッシュプレイバックに限った対応としては不要と思われる。
  - MDGContext の多用途の可能性などを考えるとリスクはある。

エバリュエータ

概要と設定の基本

エバリュエータとは何か



その位置づけ

- 素の EM のままで問題ないなら、それがいちばん。
- 素の EM で起きる問題とは？
  - 正常に動かない（DG 評価と違う結果になる）。
  - 効率的に動かない。
- エバリュエータの役割
  - 独立したノード実装だけではどうしようもないものも動くようにする。
  - EM 全体をもっと効率的にする。

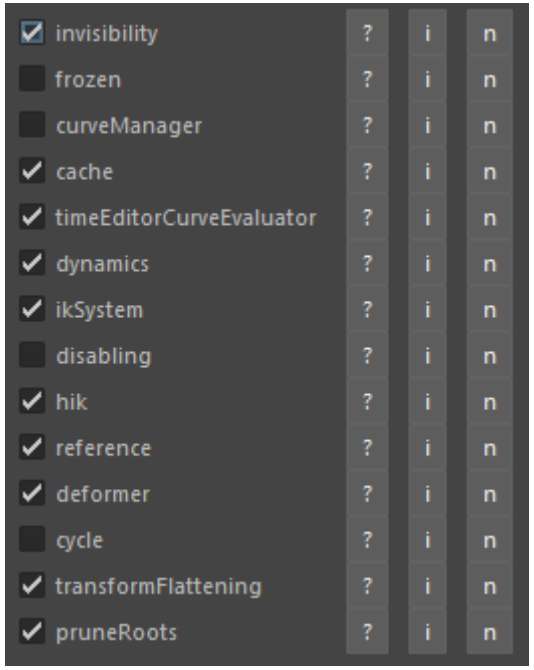
プラグインノードを動くようにする最終手段となる

しくみの概要

- 各エバリュエータは、必要に応じて各ノードの所有を主張する。
- エバリュエータには優先順位があり、各ノードの所有権は優先順位によって決まる。
- エバリュエータは所有したノードの評価プロセスを完全にオーバーライドすることができる。

evaluator コマンド

専用コマンドを持つもの（cache と deformer）も在るが、一般的には共通コマンド evaluator で様々な設定をする。



- **[✓]** 各エバリュエータの ON/OFF

```
cmds.evaluator(n=name, en=bool)
```

- **[i]** エバリュエータの情報を得る（内容はそれぞれ）

```
cmds.evaluator(q=True, n=name, i=True)
```

- **[n]** クラスター（掴んでいるノード）の内容を得る

```
cmds.evaluator(q=True, n=name, cl=True)
```

コンフィグレーション

- **[?]** コンフィグレーションのヘルプを表示

```
print(''.join(cmds.evaluator(n=name, q=True, c=True)))
```

- コンフィグレーションの設定をする。

```
cmds.evaluator(n=name, c='KEY=VALUE')
```

- コンフィグレーションの設定値を得る。

```
cmds.evaluator(n=name, q=True, vn='KEY')
```

ノードタイプの設定

各エバリュエータには紐付けるノードタイプを指定することができる。

実際にこの指定をどのように利用するかは各エバリュエータしだいで、現状、利用を確認できたのは disabling のみ。

```
cmds.evaluator(n=name, nt=nodetype, en=bool, ntc=bool)
```

en (enable)

指定したノードタイプを有効化するか無効化するかを指定する。

ntc (nodeTypeChildren)

指定したノードタイプの派生タイプに対してもこの設定をするかどうか。

設定の保存

- 原則として、エバリュエータの設定は、シーンにもユーザー設定にも保存されない。
- 例外として、cache エバリュエータのように、一部の設定がユーザー設定として保存されるものもある。

エバリュエータ

ビルトインエバリュエータの紹介

ビルトインエバリュエータの一覧 2016

デフォルト	名前	優先度	ON/OFF 状態保存	概要
✓	ikSystem	?	no	IKシステム系の管理
✓	dynamics	?	no	ダイナミクス系の管理
✓	pruneRoots	?	no	EG 起点の削減
	disabling	?	no	非サポートノードによる EM 無効化

ビルトインエバリュエータの一覧 2016.5

デフォルト	名前	優先度	ON/OFF 状態保存	概要
✓	dynamics	103000	no	ダイナミクス系の管理
✓	ikSystem	102000	no	IKシステム系の管理
	disabling	100000	no	非サポートノードによる EM 無効化
✓	deformer	5000	no	デフォーマーの GPU Override の制御
✓	transformFlattening	3000	no	トランスフォーム階層を束ねる
✓	reference	1000	no	リファレンスの評価不要ノードを除外
✓	pruneRoots	999	no	EG 起点の削減

ビルトインエバリュエータの一覧 2017

デフォルト	名前	優先度	ON/OFF 状態保存	概要
✓	invisibility	1003000	no	不可視オブジェクトの評価をスキップ
	frozen	1001000	yes	frozen属性のノードの評価をスキップ
✓	timeEditorCurveEvaluator	104000	no	Time Editor 管理の animCurve を除外
✓	dynamics	103000	no	ダイナミクス系の管理
✓	ikSystem	102000	no	IKシステム系の管理
	disabling	100000	no	非サポートノードによる EM 無効化
✓	reference	6000	no	リファレンスの評価不要ノードを除外
✓	deformer	5000	no	デフォーマーの GPU Override の制御
✓	transformFlattening	3000	no	トランスフォーム階層を束ねる
✓	pruneRoots	1000	no	EG 起点の削減

ビルトインエバリュエータの一覧 2018

デフォルト	名前	優先度	ON/OFF 状態保存	概要
✓	invisibility	1003000	no	不可視オブジェクトの評価をスキップ
	frozen	1002000	yes	frozen属性のノードの評価をスキップ
	curveManager	1001000	no	初期状態で EG に参加するノードを増やす
✓	timeEditorCurveEvaluator	104000	no	Time Editor 管理の animCurve を除外
✓	dynamics	103000	no	ダイナミクス系の管理
✓	ikSystem	102000	no	IKシステム系の管理
	disabling	100000	no	非サポートノードによる EM 無効化
✓	hik	7000	no	HumanIK
✓	reference	6000	no	リファレンスの評価不要ノードを除外
✓	deformer	5000	no	デフォーマーの GPU Override の制御
✓	transformFlattening	3000	no	トランスフォーム階層を束ねる
✓	pruneRoots	1000	no	EG 起点の削減

ビルトインエバリュエータの一覧 2019

デフォルト	名前	優先度	ON/OFF 状態保存	概要
✓	invisibility	1003000	no	不可視オブジェクトの評価をスキップ
	frozen	1002000	yes	frozen属性のノードの評価をスキップ
	curveManager	1001000	no	初期状態で EG に参加するノードを増やす
✓	cache	1000000	yes	キャッシュプレイバック機能
✓	timeEditorCurveEvaluator	104000	no	Time Editor 管理の animCurve を除外
✓	dynamics	103000	no	ダイナミクス系の管理
✓	ikSystem	102000	no	IKシステム系の管理
	disabling	100000	no	非サポートノードによる EM 無効化
✓	hik	7000	no	HumanIK
✓	reference	6000	no	リファレンスの評価不要ノードを除外
✓	deformer	5000	no	デフォーマーの GPU Override の制御
	cycle	4000	no	サイクルクラスターを効率化
✓	transformFlattening	3000	no	トランスフォーム階層を束ねる
✓	pruneRoots	1000	no	EG 起点の削減

invisibility

- 非表示のオブジェクトの評価スキップ機能。
  - 非表示オブジェクトそのもの、及び可視オブジェクトの評価に寄与しないもの全ての評価をスキップする（たとえば、下流に可視 DAG ノードが無いネットワークはスキップされる）。
  - スタティックな状況のみが考慮され、可視性がアニメートされているものは対象外。
  - 表示されるかどうかに影響する様々な要因が考慮される。
- 比較的分かりやすい問題が起きやすい部分でもある。
  - 「なんかおかしければ、まず invisibility エバリュエータを OFF にしてみろ」
  - 細かなバグも多かったが、バージョンを経て改善されてきている。
  - 2018 から、Evaluation=“Always” の expression はスキップ対象外となった。

frozen

- invisibility よりも細かな制御が可能な評価スキップ機能。
- まず、このエバリュエータに関係なく、EM は frozen アトリビュートが ON のノードの評価をスキップする。
- このエバリュエータでは、そこからさらに frozen なノードの上流や下流を評価しないようにするなどの制御ができる。

curveManager

- ノードが最初から EG に組み入れられる条件を拡張する。
  controller タグの目的とするところを一般化した機能といえる。
  - 頻繁な EG 再構築を抑制して応答性を向上。
  - リグの操作の応答性を向上（Manipulation=ON であること）。
- コンフィグレーションによって以下の EG 追加条件を設定
  - animCurve は 1 フレームから対象。
  - 追加アトリビュート forcedAnimated が true に設定されたノード。
- ノードを EG に追加しても、アニメートされていないものはアニメーション再生中の評価はスキップする制御をする。

cache

キャッシュプレイバック機能そのもの。

機能の ON/OFF は、このエバリュエータの ON/OFF に相当。

timeEditorCurveEvaluator

Time Editor が管理している animCurve を EM による一般評価の対象外とする。

dynamics

- Bullet と Bifrost のパラレル評価をサポート。
- 2016.5 以降で Nucleus (nCloth, nHair, nParticle) のパラレル評価を追加サポート。
- 上記以外の非サポートの古いダイナミクス（Particle, Rigid, Fluid など）がシーン中に含まれる場合に EM を無効化。
  非サポートノードの詳細は 公式ドキュメント を参照のこと。

ikSystem

- Multi-Chain Solver を見つけたら EM を無効化。
- それ以外の IK Solver では何もしない？
  - ikSystem エバリュエータを無効化しても問題ないように見える。
  - サイクルクラスター化されるため、それで問題ない？
  - 公式ドキュメントには「並列実行前に遅延更新（Pull評価？）をする」と書かれているが、サイクルクラスターを指している？

disabling

指定したノードタイプがシーン中に含まれるときに EM を無効化できる。
プラグインでパラレルのサポートを根本的に見送る場合の措置として便利。

プラグインのロード時、そのノードタイプ foo と bar を disabling に設定する実装例：

```
// プラグインのロード時の initializePlugin() に追加するコード。
#if MAYA_API_VERSION >= 201600
    MGlobal::executeCommand(
        "evaluator -n disabling -nt foo -en 1;¥n"
        "evaluator -n disabling -nt bar -en 1;¥n"
        "evaluator -n disabling -en 1;"
    );
#endif

// プラグインのアンロード時の uninitializePlugin() に追加するコード。
#if MAYA_API_VERSION >= 201600
    MGlobal::executeCommand(
        "evaluator -n disabling -nt foo -en 0;¥n"
        "evaluator -n disabling -nt bar -en 0;¥n"
        "if(! size(evaluator(¥"-q¥", ¥"-n¥",¥"disabling¥", ¥"-nt¥"))) evaluator -n disabling -en 0;"
    );
#endif
```

hik

HumanIK 関連らしいですが、よくわかりません。

reference

一時的にアンロードされているリファレンスのためにシーン中に残っているノードを評価から除外する。

deformer

デフォーマーの GPU Override を制御する。

GPU Override 機能の ON/OFF は、実質このエバリュエータの ON/OFF に相当するが、2016 では隠されている。

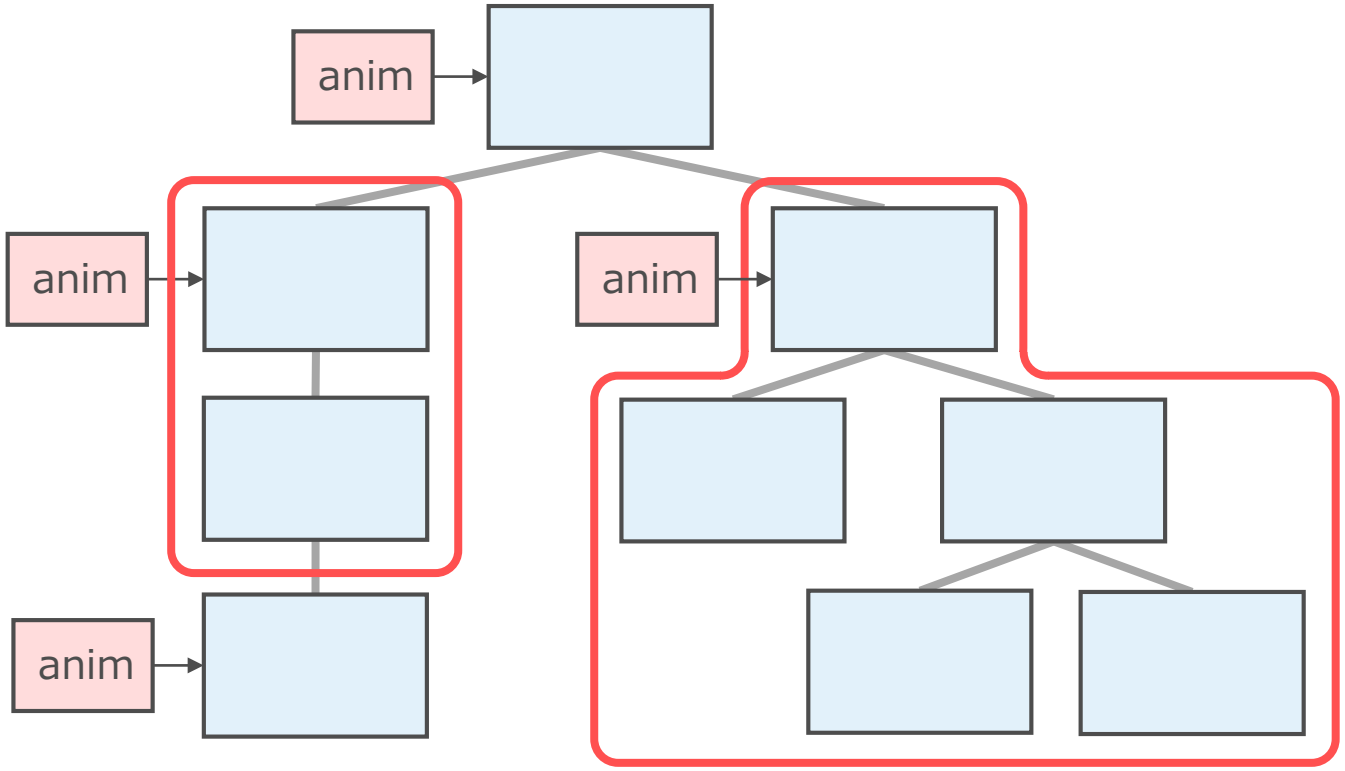
cycle

サイクルクラスターを分解してパラレル評価されるようにしてパフォーマンスを向上させる。

開発中の機能のプロトタイプとのこと。

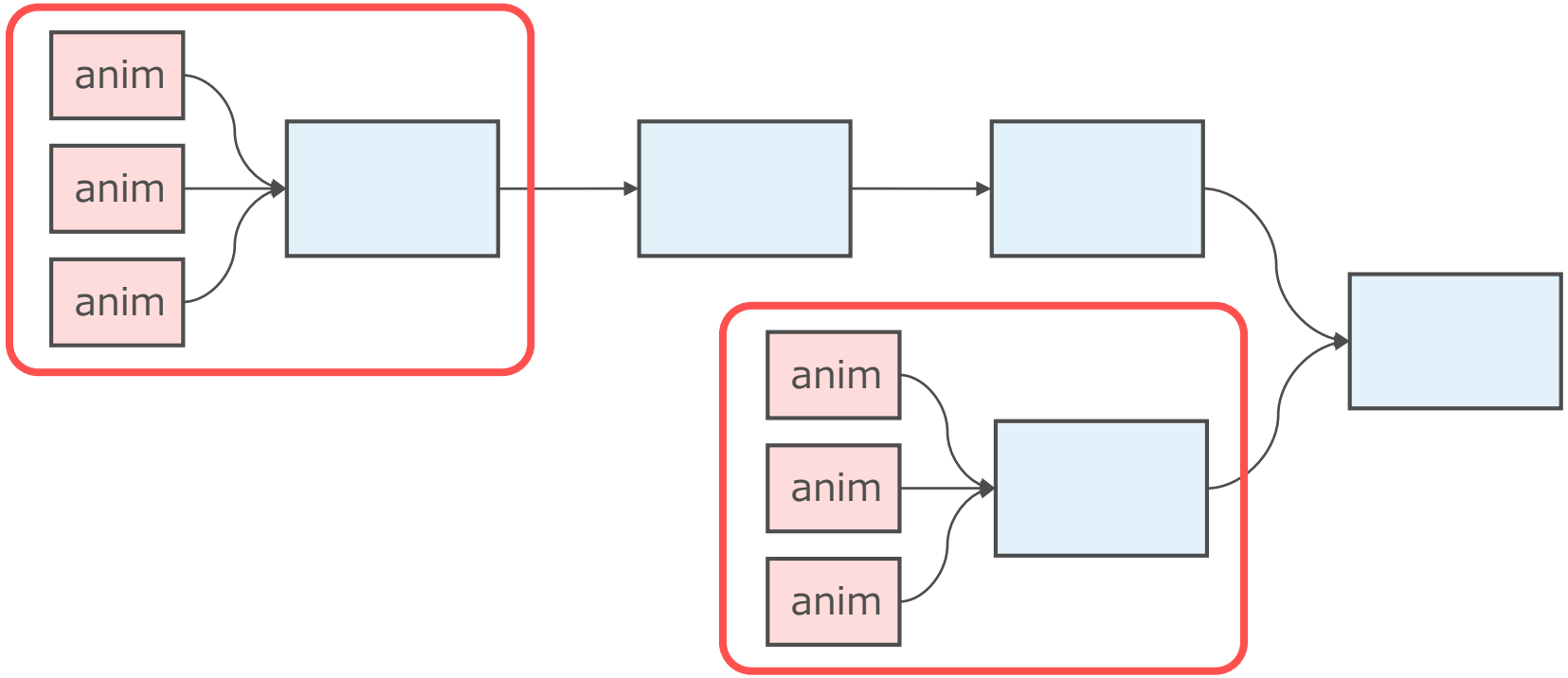
transformFlattening

アニメートされていない DAG ノードをアニメートされた上位のノードとひと固まりで扱う。



pruneRoots

EG の起点となる animCurve を個別に評価ノードとして扱うと膨大な数となり、スケジューリングのボトルネックとなるので、ひと固まりにする。



エバリュエータ

プラグインによるカスタムエバリュエータ

エバリュエータの作り方

２種類のプロキシクラスが存在する。どちらかを継承して作る。

MPxCustomEvaluator

- 通常のエバリュエータ開発に用いる。
- DG ノードを所有し、クラスタリングすることができる。
- 所有するクラスターのスケジューリングタイプを指定できる。
- 所有するクラスターの評価プロセスをオーバーライドできる。

MPxTopologyEvaluator

- 2019 で追加された新しい方法。
- DGノードよりも細かい粒度で EG の評価ノードを構築することができる。

MPxCustomEvaluator のメソッド

markIfSupported()

所有したいノードの検査を実装。

どのようにクラスタリングするか（単体？まとめる？）は、別途 setConsolidation() を呼び出して決めることができる。

schedulingType()

クラスターのスケジューリングタイプを指定する。

clusterInitialize()

クラスターの初期化を実装。所有したノードの数だけ呼ばれる。

preEvaluate(), clusterExecute(), postEvaluate()

評価時に呼ばれるので、評価プロセスを実装する。

まず preEvaluate() が 1 回呼ばれ、clusterExecute() が所有したノードの数だけ呼ばれ、最後に postEvaluate() が 1 回呼ばれる。

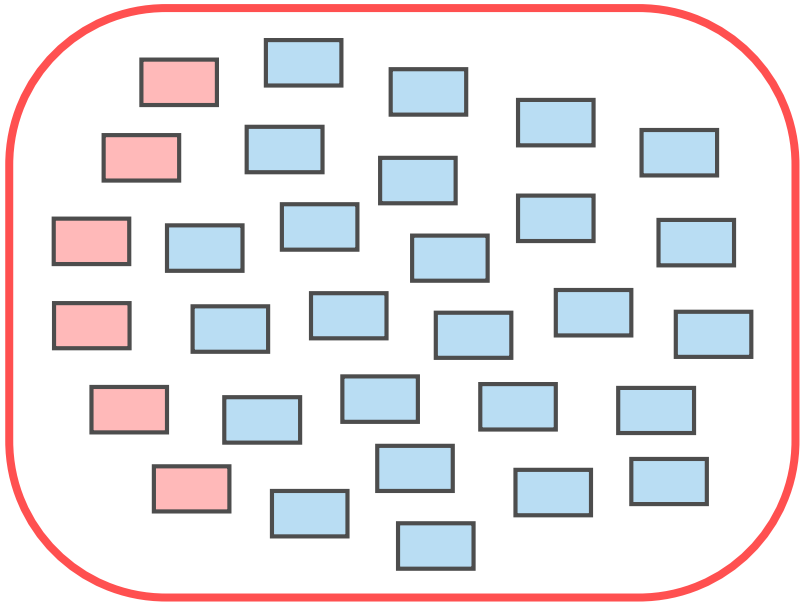
clusterTerminate()

クラスターの後始末を実装。所有したノードの数だけ呼ばれる。

devkit: simpleEvaluator

controller タグに基づき、シーン評価を非常に簡略化する例。

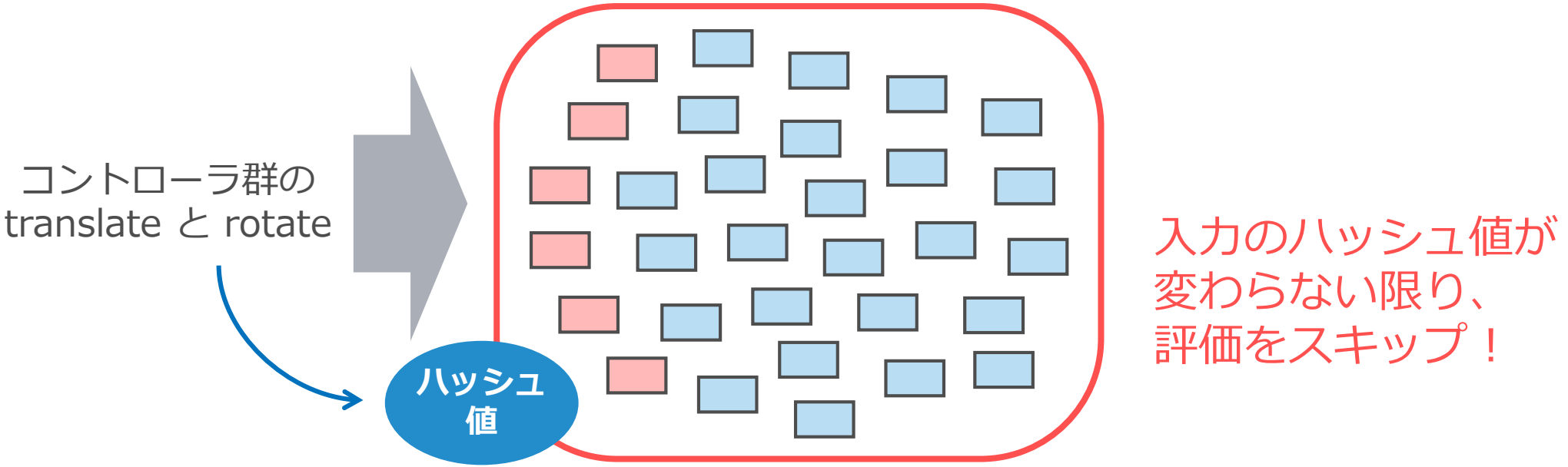
- あらゆるノードの所有を主張。
  ただし優先度は低いので、他で余ったものを所有する形になる。
- 所有したノードの中から「コントローラ」を探し、その translate と rotate プラグを保持する。



devkit: simpleEvaluator

評価プロセス：

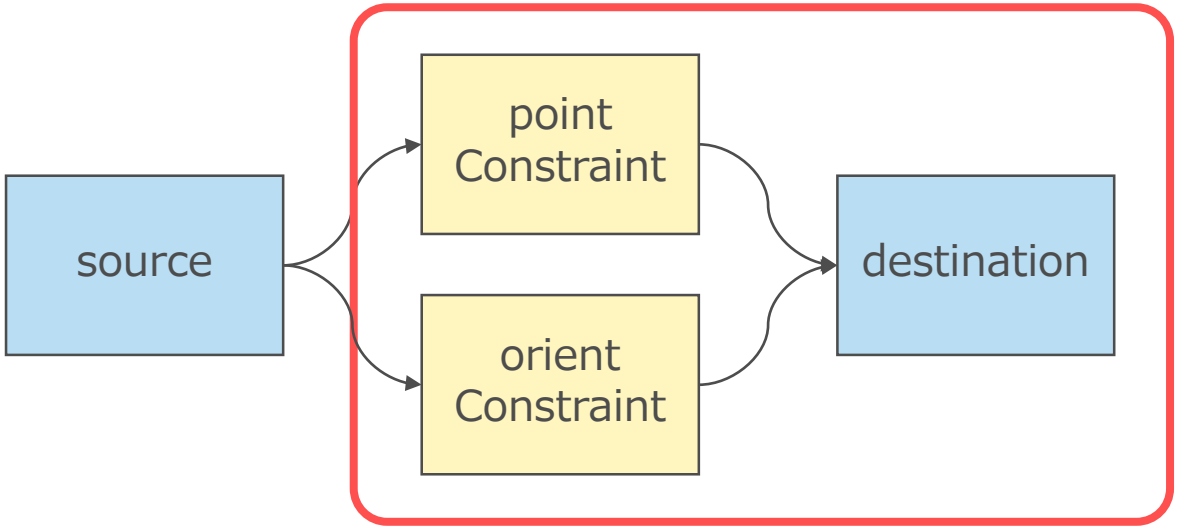
- 全コントローラの translate と rotate 入力から「ハッシュ値」を求める。
- 前回の評価時からハッシュ値が変わった場合に全ての評価を呼び出すが、変わらないなら全ての評価をスキップする。



devkit: constraintEvaluator

コンストレインの計算を完全にオーバーライドしてしまう例。

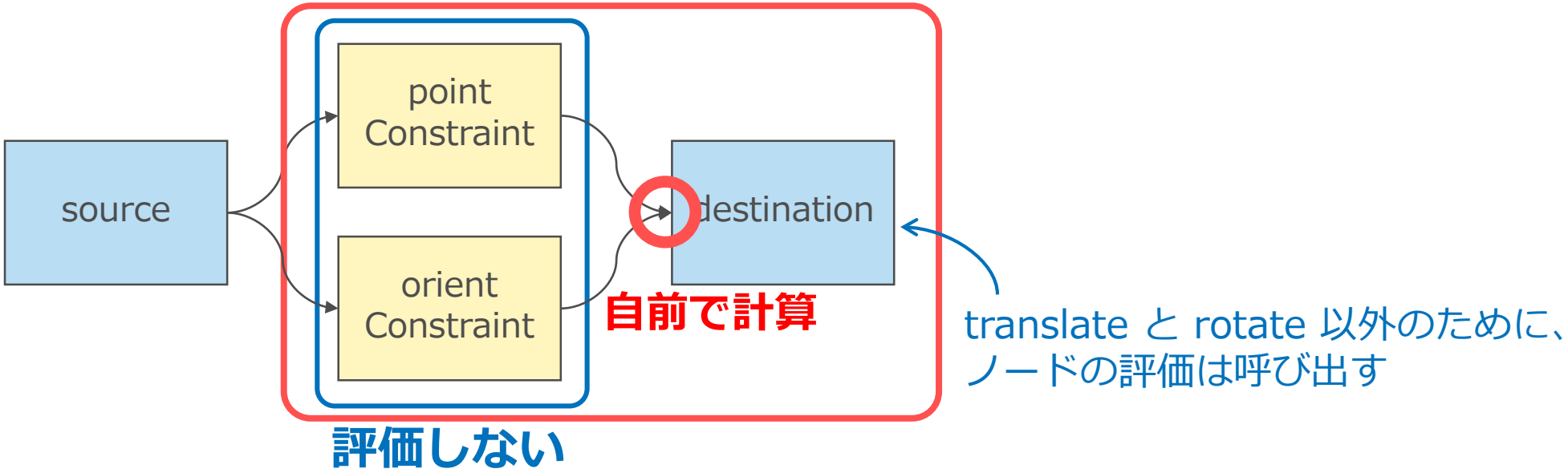
- それらを評価するのではなく、このエバリュエータが自前で計算する。
  コンストレインはサイクルクラスター化されるので、効率化しようというもの。
- 同一ソースの pointConstraint と orientConstraint によって拘束されている
  ノードと、コンストレインノードそのものの所有を主張。



devkit: constraintEvaluator

評価プロセス：

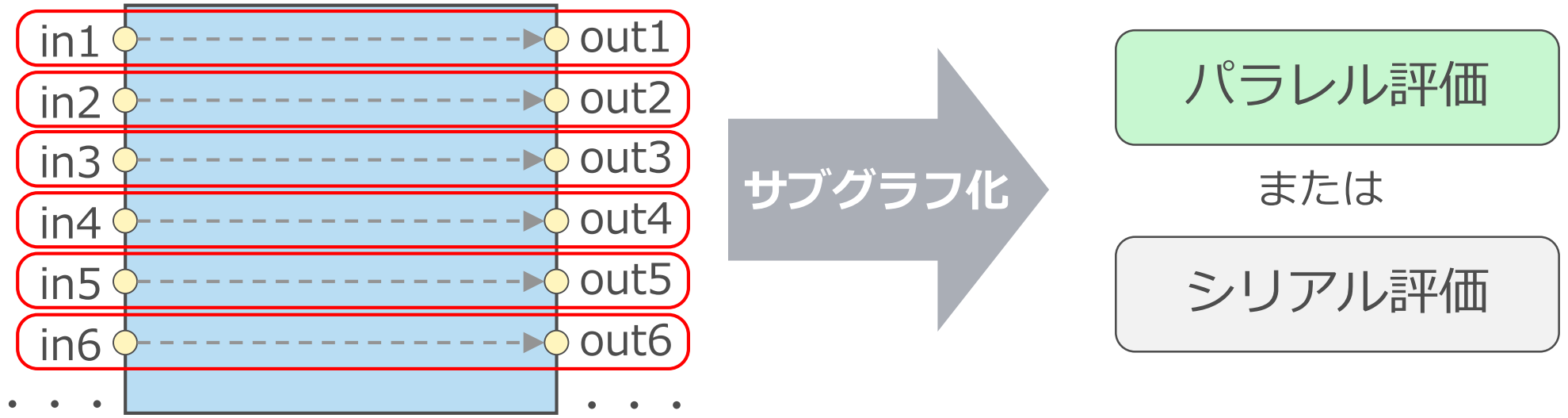
- コンストレインを評価せず、拘束されたノードの translate と rotate を計算してデータブロックにセットする。
- 他の未解決プラグのための評価は呼び出す。



devkit: testMTopologyEvaluator

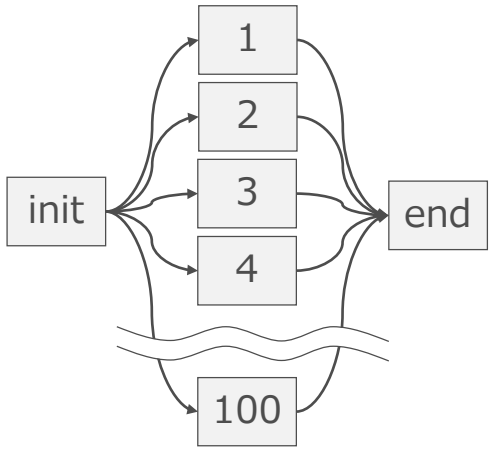
MPxTopologyEvaluator を使用して、DG ノードよりも粒度の細かい評価ノードのグラフを作るエバリュエータの実装例。

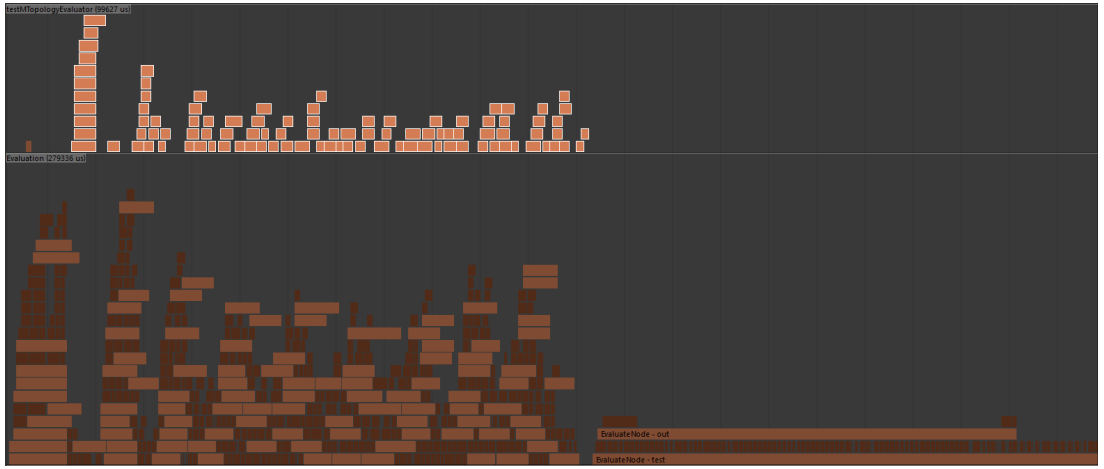
- 100個の独立した in → out の Attribute Affects を持つノードを生成。
- 1 つの in/out ペアを 1 個の「評価ノード」として EG に組み入れる。
- 「パラレル評価」と「シリアル評価」の二種類のサブグラフをテスト。



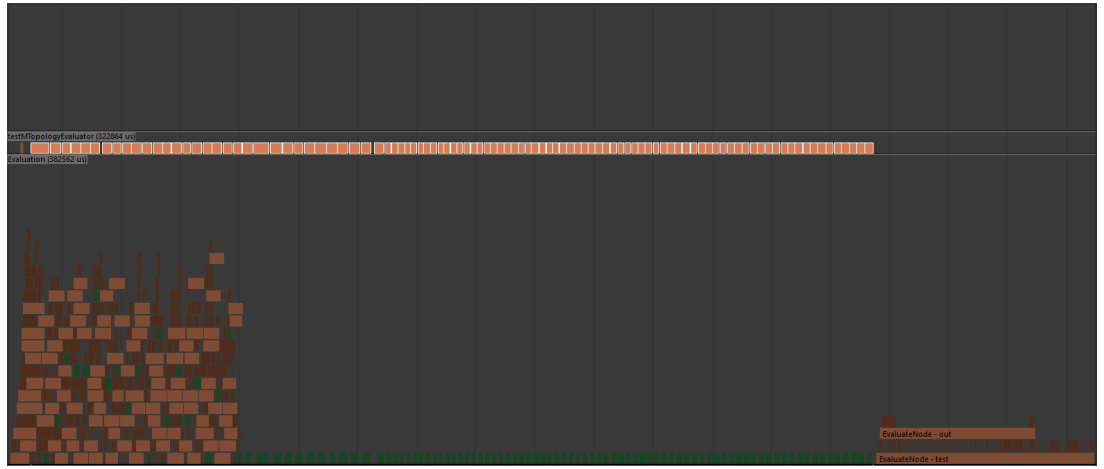
devkit: testMTopologyEvaluator

パラレル評価サブグラフ





シリアル評価サブグラフ



さらなる並列化の意味するところ

パラレル評価といえ、通常は１つのノードが複数スレッドに同時にアクセスされることはない。

これまでのもの（DG の最も一般的なケース）が問題なく動くようにするため？

エバリュエータによって、１つのノード評価をさらに並列化することができる。

- 排他制御のないメンバ変数など、これまでの実装のままでは問題となる。
- キャッシュのバックグラウンド評価では辛うじてしのげたが…。

まとめ

まとめ

- DG の理想的な姿は、各ノードは独立した関数であり、依存関係が明示されていること。
- 理想的なシーン構造やプラグインは、DG でも Parallel でも問題なく動作する。
- Maya® の基本（DG やシーン構造）は非常に単純で美しいが、きれいに載せきれない機能が多く生まれた。
- DG 上の実装がシンプルでないものは Parallel でも複雑な対応となる。
- Parallel で結果が変わってしまったり良い性能がでない場合、問題解決のための Tips はいくつかあるが、万能ではなく、地道な調査が必要になる。
- キャッシュ生成は、別コンテキストでのバックグラウンド評価。しかし、今のところはプラグイン側に大きな変更は要らないようだ。
- エバリュエータの開発は、特殊なノードを動かす最終手段となる。

ご清聴ありがとうございました。

佐々木 隆典

ryusukes@square-enix.com

SQUARE ENIX ADVANCED TECHNOLOGY DIVISION