CEDEC2015 Computer Entertainment Developers Conference
Reach Next Level

モジュラーリグシステムのアーキテクチャ

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

今回お話しすること

- モジュラーリグシステム開発の概要
- 基本機能の俯瞰
- メタノードについて
- ラベルについて
- リグモジュールの組み合わせについて
- モジュラーリギングの実演
- リグ API とモジュールの実装
- まとめ

モジュラーリグシステム開発の概要

モジュラーリグシステムとは？

- キャラクタリグを、モジュール（部品）の組み合わせで作れる仕組み？
- Maya® で CRAFT というモジュラーリグシステムを開発しました。
- 参考にできる情報が少なく、多くの独自の考え方を導入して開発してきました。
- きっと、世の中の多くのツールがそうだと思うので、この経験が皆様のご参考になれば幸いです。

CRAFT について

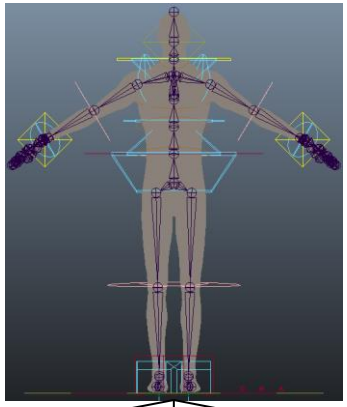
- **C**haracter **R**igging and **A**nimation **F**unctional **T**oolkit
- ゲームのリアルタイムキャラクタアニメーションからプリレンダームービーまで、様々なプロジェクトで利用されている。
- 社内で誰でもすぐに導入出来る汎用ツール。

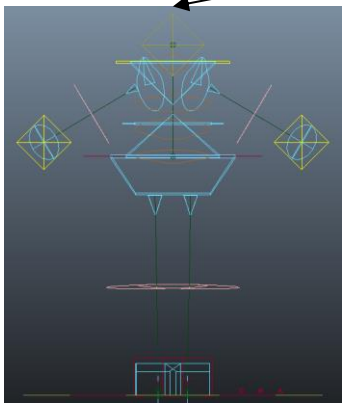
CRAFT の歩み

- 2011年11月
  基礎ライブラリやプラグインの開発着手
- 2012年4月
  CRAFT 試作版（version 0.0） 開発着手
- 2012年6月
  CRAFT 0.0 リリース（Biped相当機能・試作版）
- 2013年12月
  CRAFT 1.0 リリース（モジュラーリグシステム）
- 2015年 現在も機能追加や改善を継続中。

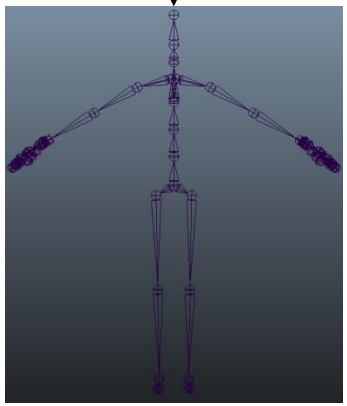
リグの分類

キャラクタリグ

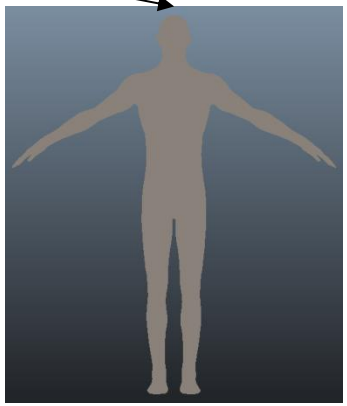




コントロールリグ

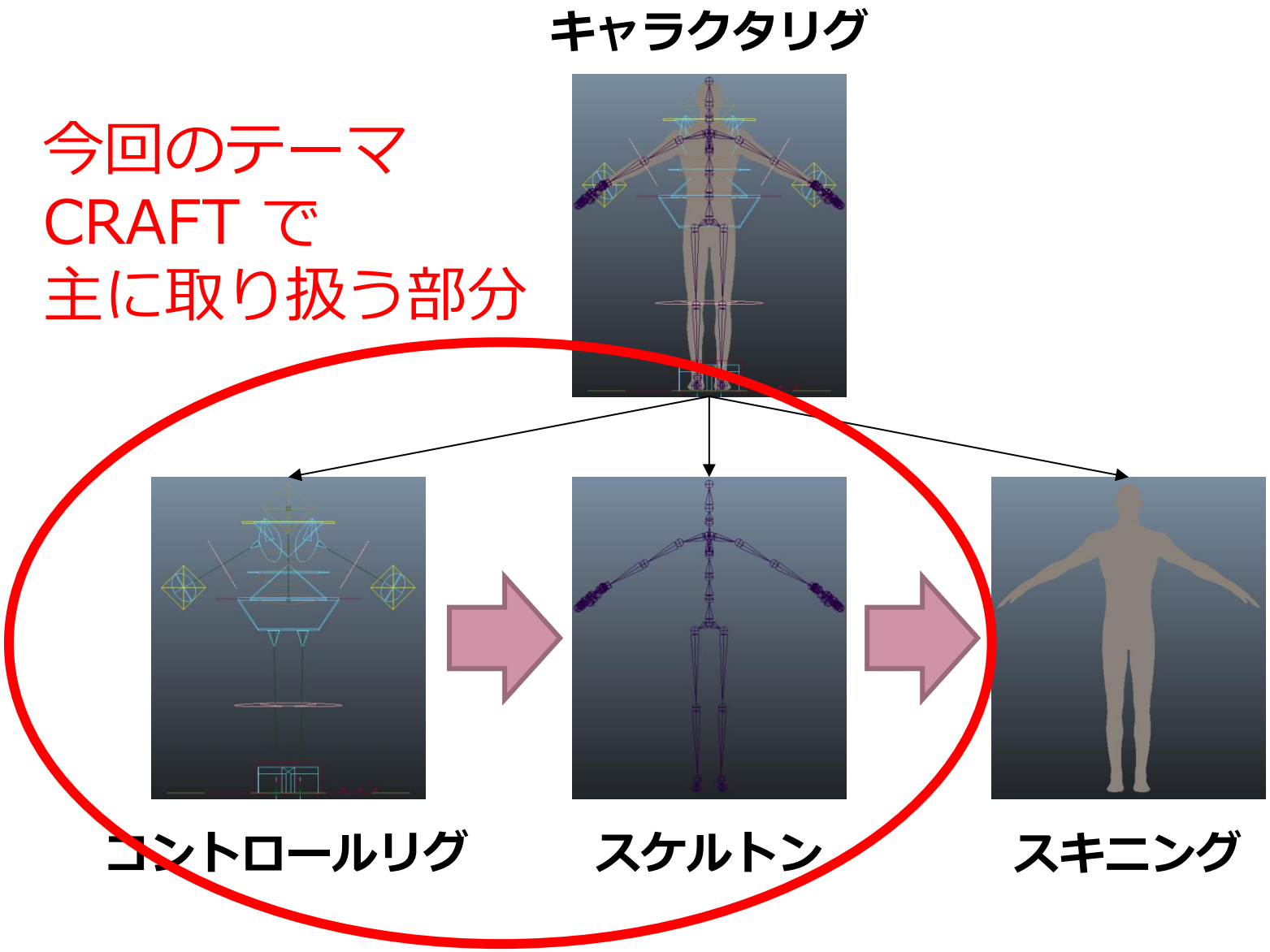


スケルトン



スキニング

リグの分類



スケルトンとコントロールリグ

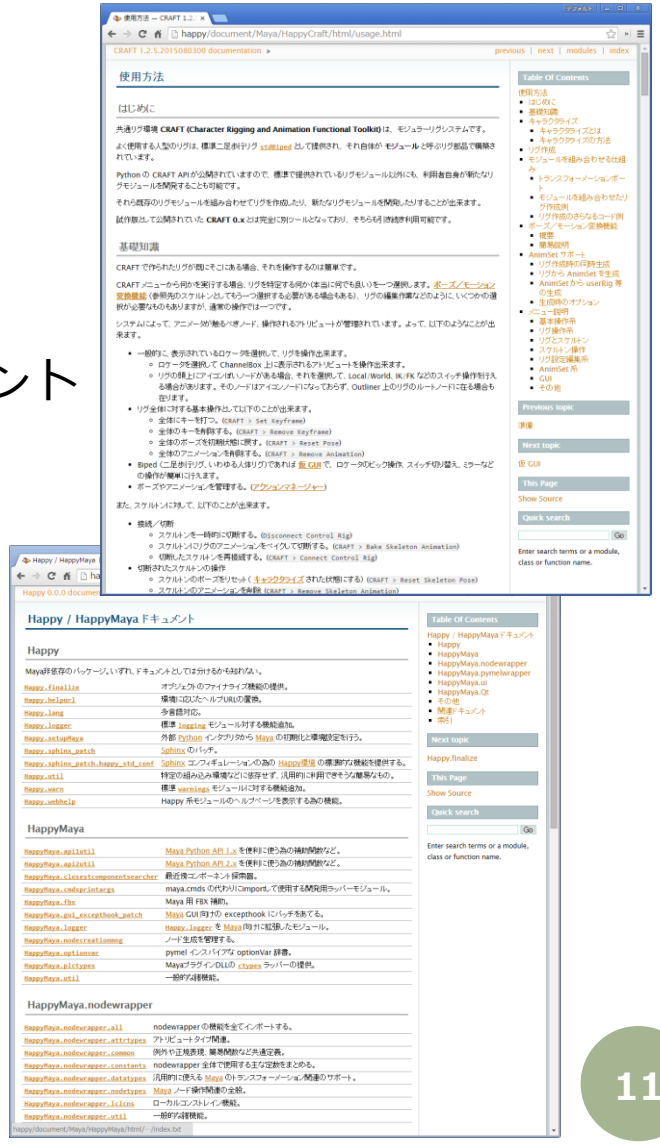
- 世の中のリグシステムには、そのように分離されていないものも多い。
  - モデルデータに骨入れをしながらコントロールリグも作られていくような。階層構造も混在してたりする。
- CRAFT は、スケルトンとコントロールリグを明確に区分する。
  - 様々なプロジェクトの既存のスケルトンにコントロールリグを取り付けたい。
  - ゲームデータとしてエクスポートするのはスケルトンのみ。だから別階層でなければならない。
  - レンダリング時にはコントロールリグは要らない。
  - MotionBuilder や HumanIK と同じ考え方。

開発方針

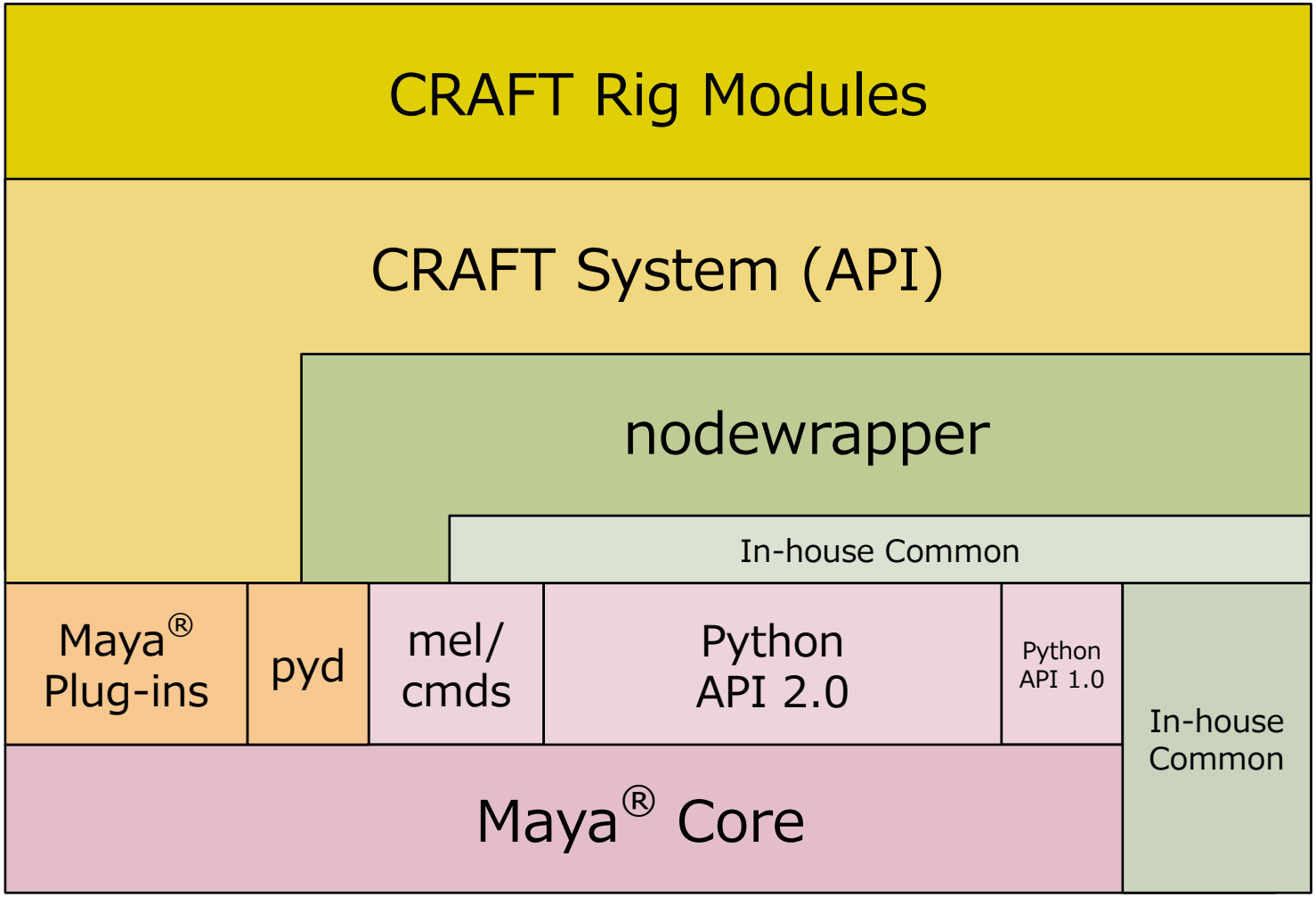
- 独自機能を作るよりも、なるべく Maya® の機能を活かせるようにする。
  - とはいえ、プラグインによる独自ノードは積極的に開発。高速化やコードや構造の単純化に貢献。
  - リギング機能は置き換えつつも Maya®の作法に則り、アニメーション諸機能はそのまま使えるように保つ。
- 安定したハンドリングのための単純なグラフ評価。
  - プル型アーキテクチャの徹底（Maya®の基本）。
  - アニメーションフレーム間独立の徹底。
- モジュールの独立性と全体的な高機能性は相反する要素であり、難しい問題。基本方針としては、独立性を重視（フルボディIKなどは考えない）。

開発言語

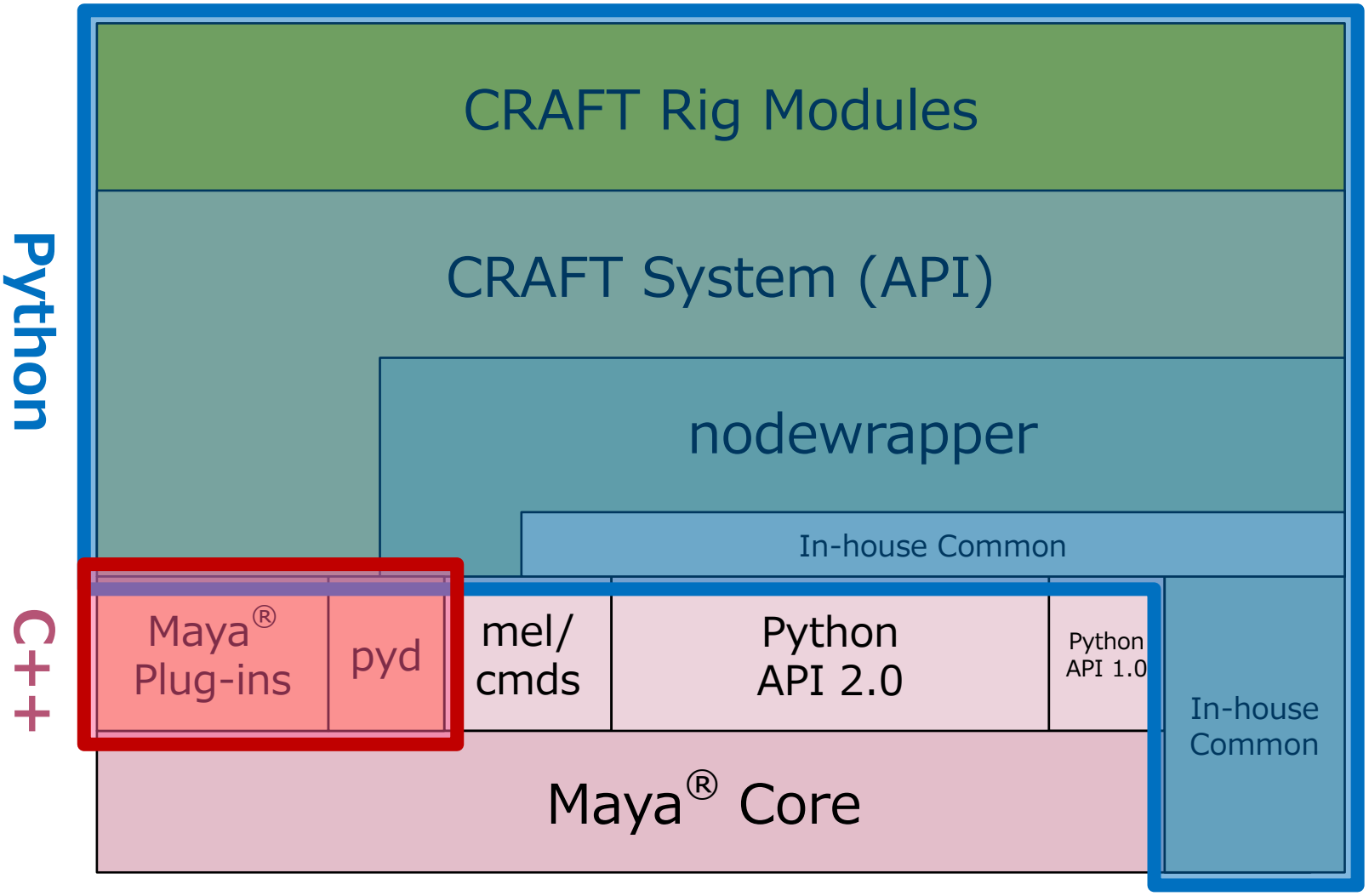
- Python
  - おそらく、全体の95%以上
  - CRAFTシステムも API として開示
    Sphinx を用いた本当に詳細なドキュメント
  - nodewrapper
    pymelインスパイアな独自ライブラリ
- C++
  - プラグイン（ノードやデータ型）
  - 低レベル Python dll モジュール（.pyd）



CRAFT API層



CRAFT API層



なぜ nodewrapper なのか

- pymel を試したが…
  - すごく遅い
  - 操作のundo可否が徹底されていない
  - リギングに便利なトランスフォーム計算関連をもっと追加したかった
- 独自開発で不満を解消
  - 速い（API 2.0 ベース + 速度に拘った実装）
  - Get系はAPI実装、Set系はmel実装（undo可）の徹底
  - リギングで便利な機能満載
- コンテキストによって制御を変える抽象化がしやすくなり、 CRAFTのリグ開発フレームワークの構築にとても役立つことになった。（後述）

なぜ nodewrapper なのか

- pymel を試したが…
  - すごく遅い
  - 操作のundo可否が
  - リギングに便利なト
    加したかった
- 独自開発で不満を解消
  - 速い（API 2.0 ベース + 速度に拘った実装）
  - Get系はAPI実装、Set系はmel実装（undo可）の徹底
  - リギングで便利な機能満載
- コンテキストによって制御を変える抽象化がしやすくなり、 CRAFTのリグ開発フレームワークの構築にとても役立つことになった。（後述）

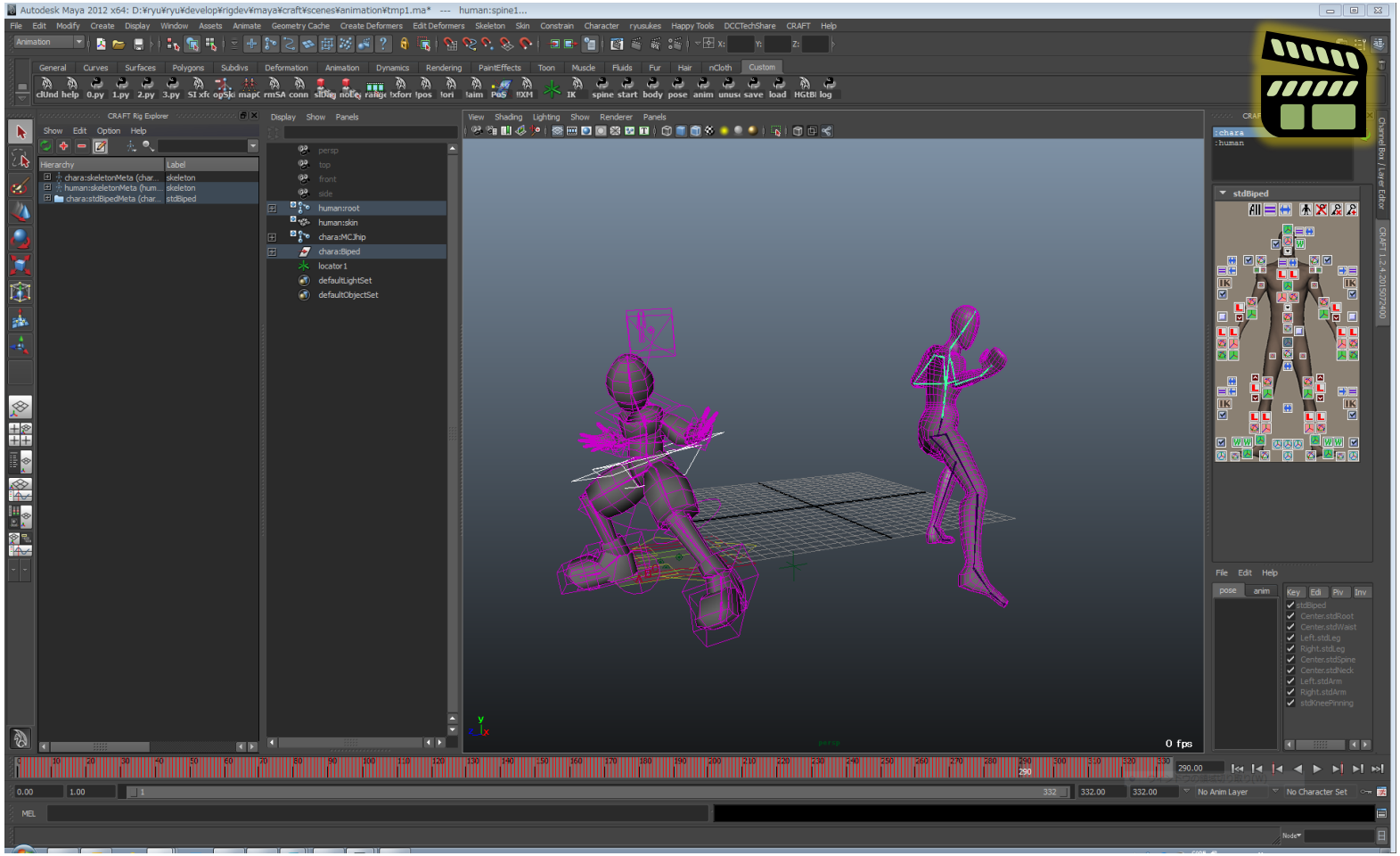
pymel から非常に多くのことを学びました。

素晴らしい功績に感謝致します。

CRAFT API とリグモジュール

- リグモジュールは CRAFT API を用いて開発する。プラグインのようなもの。
- 標準で多くの汎用モジュールを搭載している。
- 各プロジェクト内でカスタムモジュールも開発されている。
  - プロジェクトに特化した機能が必要なもの。
  - 標準（私）の開発が追いつかないもの。

基本機能の俯瞰 ～ Biped



メタノードについて

キャラクタライズ

- スケルトンを CRAFT に認識させること。
- ジョイントに**ラベル**を付ける。
  - どのような構造や命名のスケルトンでも、この段階でシステムに認識させることで共通して取り扱える。
  - 手動（GUIやスクリプト）でも可能だが、予め一通りの社内プロジェクトの命名ルールのテーブルが仕込んであり、通常は全自動で識別される。
- ジョイントの初期ポーズの**マトリックス**を保存。
  - リターゲット機能で重要な情報となる。
- 情報は**メタノード**と呼ばれるノードに保存される。

メタノード

- プラグインで実装されたノード。
- スケルトンやリグ部品ごとに一つ。
- リグのバージョン等の情報を保持。
- メタノードそのもののラベル（リグ名）を保持。
- 登録されたメンバー（リグを構成するノードやアトリビュート）のラベルを保持。
- メンバーノードの初期マトリックスを保持。
- リグ部品間の関連付けや階層構造を定義。
- リグを単に階層ごと削除出来るようにする。

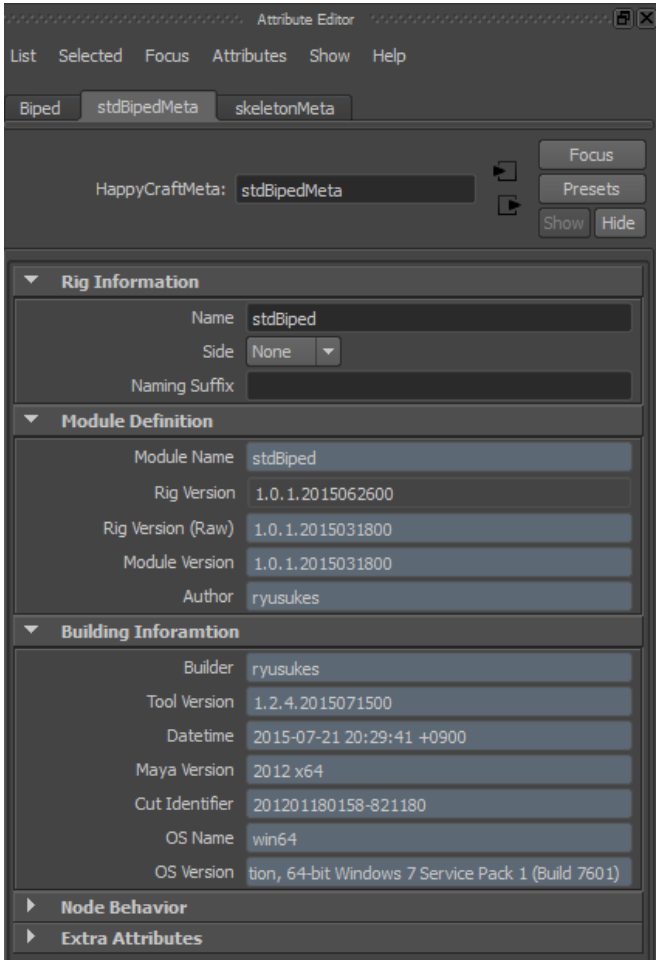
メタノードは何故プラグイン？

情報保存のためのアトリビュートが有れば良いので、何らかのノードに addAttr して代用することも可能だが、プラグインノードとすることで・・・

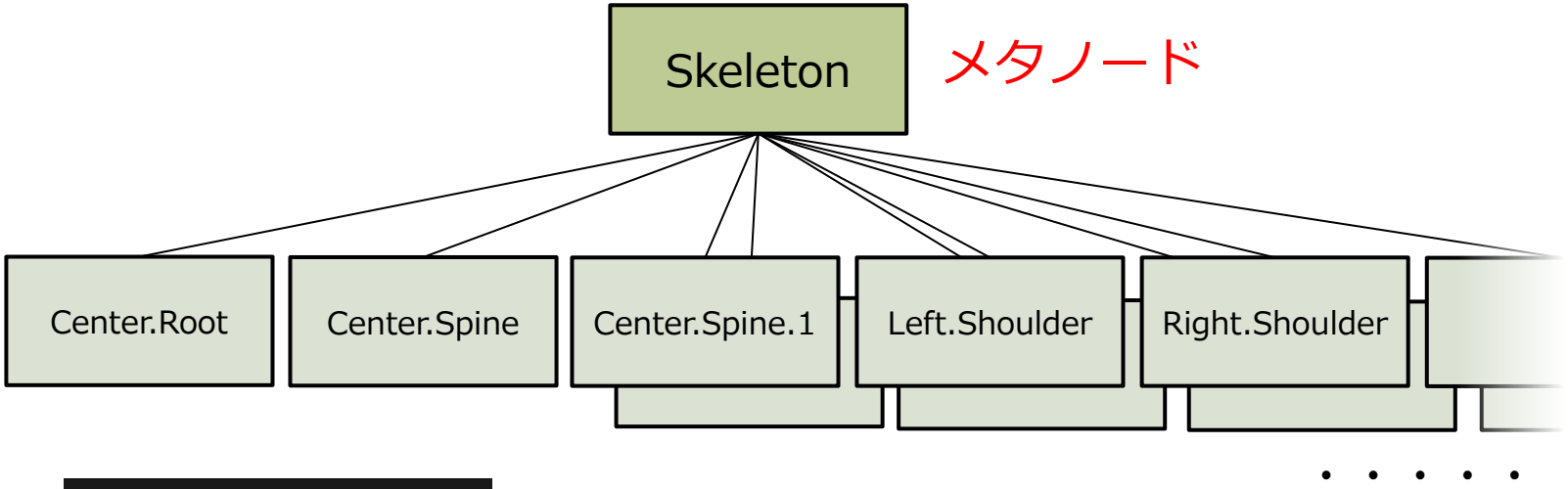
- API では Python dict でのメンバーアクセスを提供。実態は Maya® のコネクションなので、C++ プラグインと Python モジュールの連携で Maya®コネクションをキャッシュ化。
- ゴミを残さずにリグを削除出来るようにする機能を提供。
- Attribute Editor レイアウトを提供。

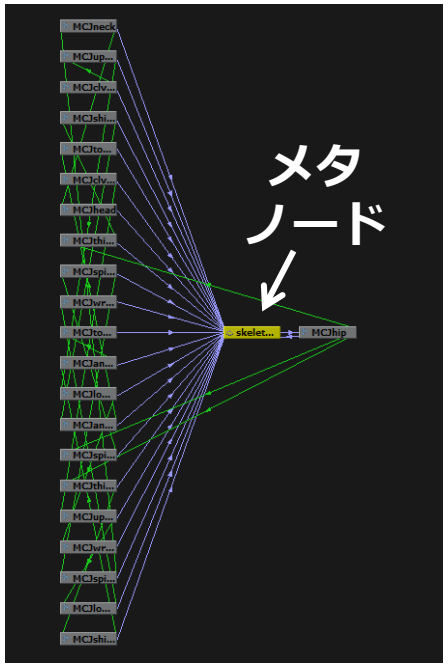
リグ情報

- リグモジュールのバージョン
  - 補助ツールが動作する際に、サポート範囲のバージョンかどうか見極めたり、バージョンに応じた処理に分岐させたり出来る。
  - トラブル時に問題を追跡出来る。
- その他、リグ作成時の情報
  - CRAFT システムのバージョン
  - ユーザー環境についての情報



スケルトンの例



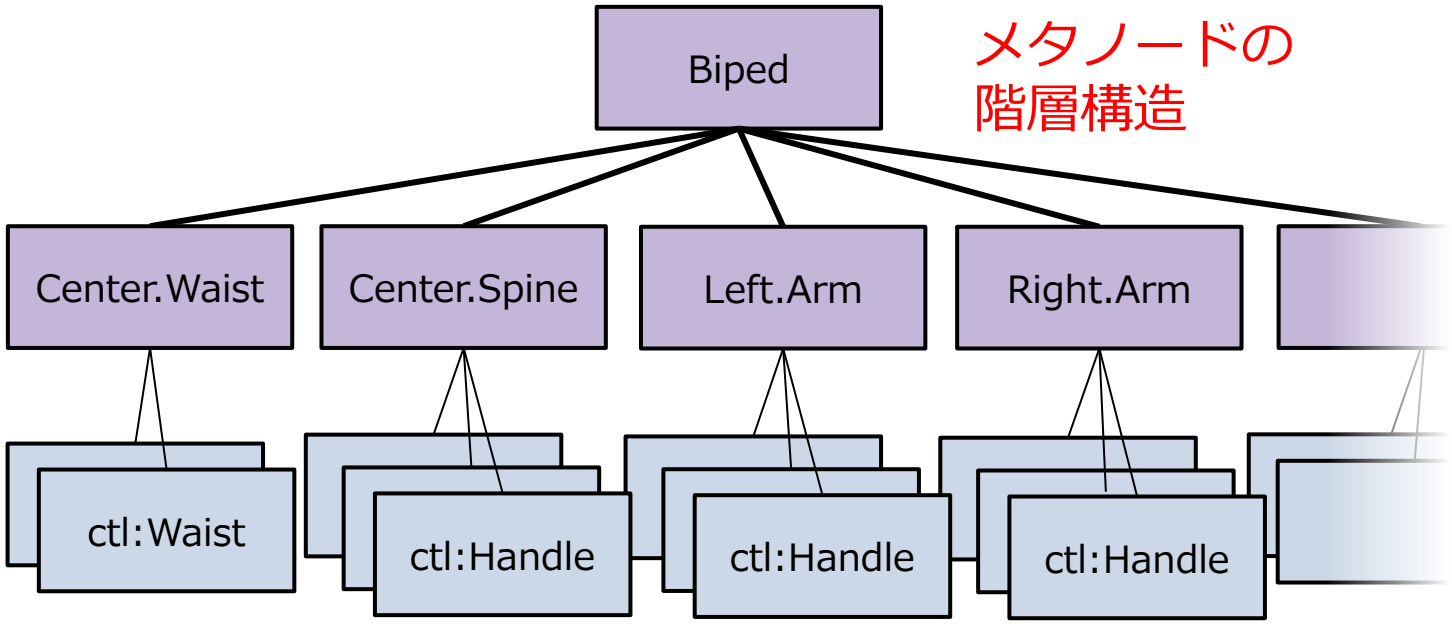


メタノードは登録されたジョイントのラベルと初期マトリックスを保持している。

Maya® 上での
コネクション実態

コントロールリグの例

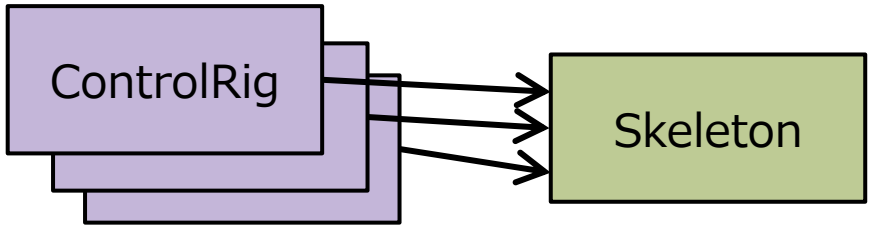
複数のリグモジュールの組み合わせを一体のリグとして扱うため、メタノードの階層を構築する。



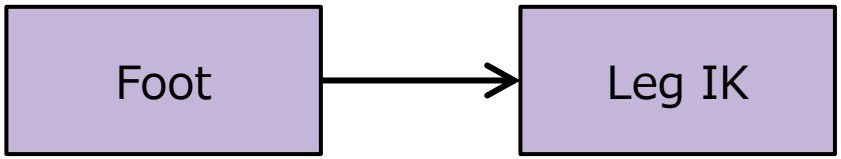
各リグはメンバーノードのラベルと初期マトリックスを保持している。

リグの関連付け

メタノードの階層構造はグループのような概念だが、
リグ間の横方向の関連性も別途定義出来る。



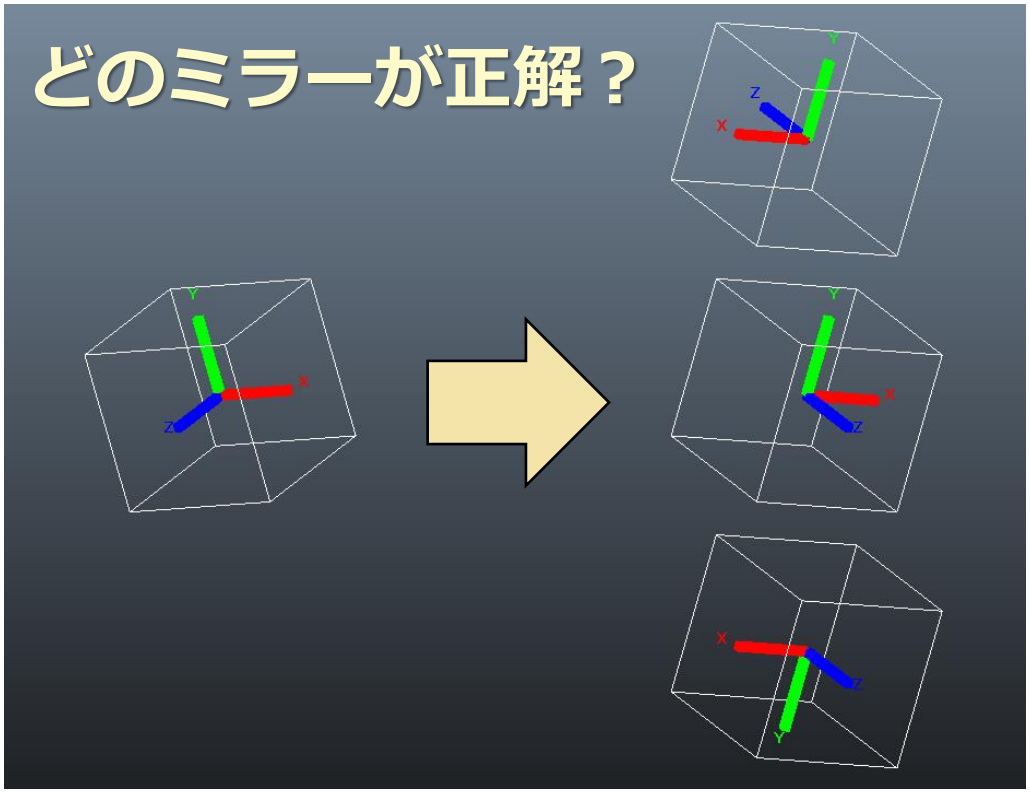
例えば、各 ContorlRig は Skeleton の一部又は
全体を制御する。



例えば、Foot リグは Leg IK リグの IK ハンドル
を乗っ取る。

なぜ初期マトリックスを保存するのか？

- ポーズを全て「初期ポーズからの差分」で扱うため。
  - リターゲッティングの際の軸違い吸収
  - ミラー処理の基準



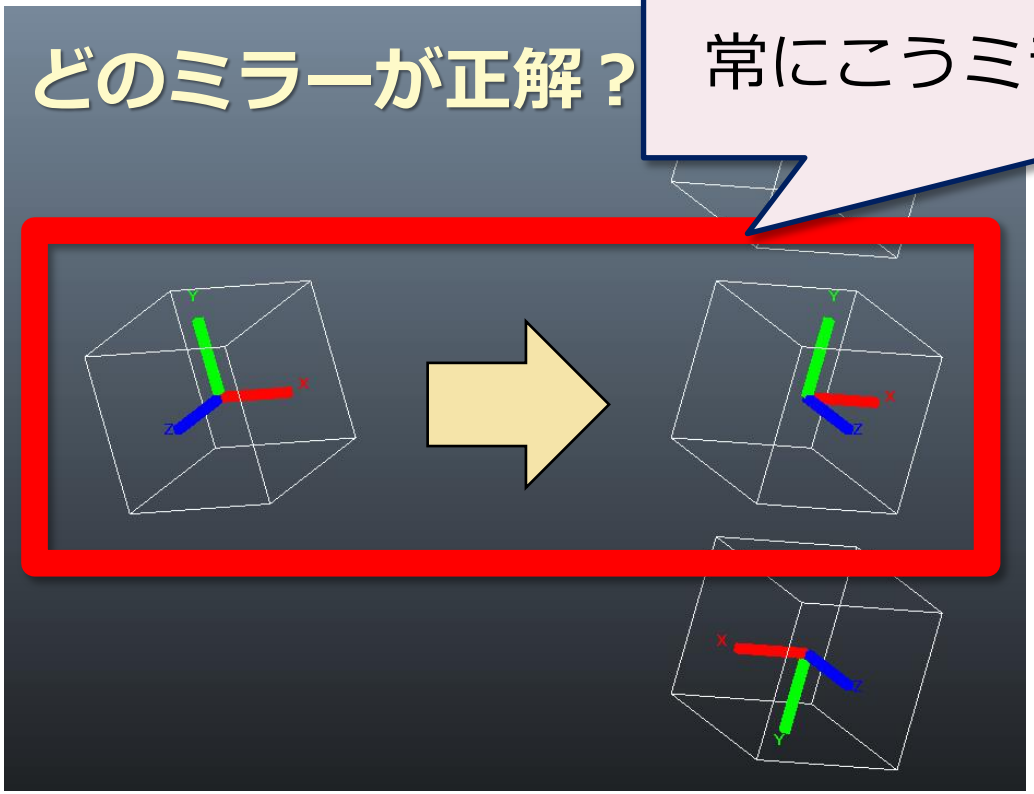
ワールド空間上にキューブが存在する状況を考えると、正解は二番目であろう。

しかし、X軸方向に伸びるボーンだとすると、正解は一番目か三番目であろう。

実は、状況により正解は無数に存在する。

なぜ初期マトリックスを保存するのか？

- ポーズを全て「初期ポーズからの差分」で扱うため。
  - リターゲッティングの際の軸違い吸収
  - ミラー処理の基準



「初期ポーズからの差分」を常にこうミラーする。

ろう。

しかし、X軸方向に伸びるボーンだとすると、正解は一番目か三番目であろう。

実は、状況により正解は無数に存在する。

リグの簡単削除の仕組み ～ その動機

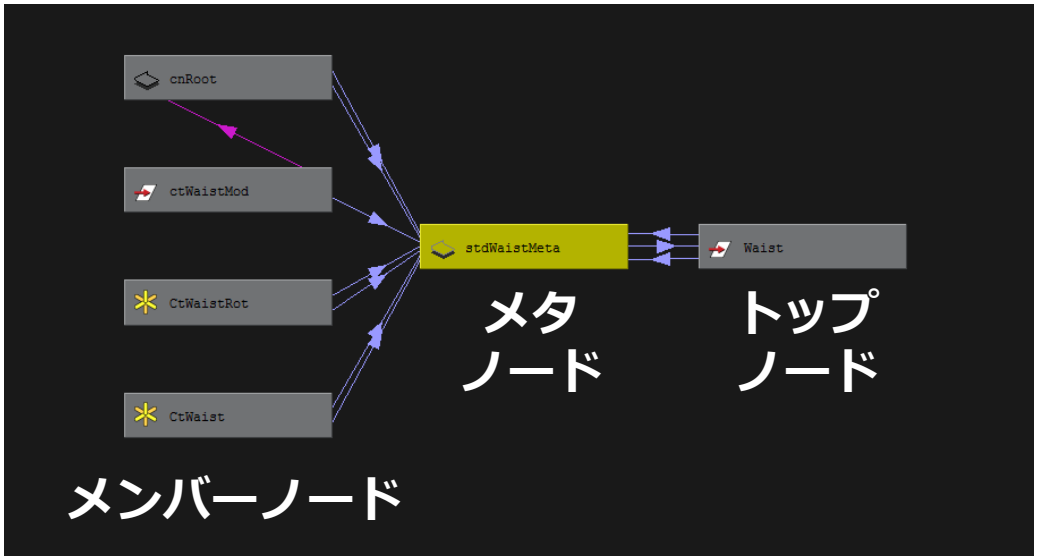
- 単にリグ階層ごと削除
  でゴミが残らないようにしたい。
- Maya® は、
  何らかのノードが削除された際に「不要となったことが明白なノード」が連動して削除される仕組み
  を一応持っている。
- しかし、しばしばゴミノードが残る。
  → 連動削除ルールから漏れるノードがあるから。

Maya® の連動削除の仕組みを知る

- DAGノード（階層構造を組めるノード）は連動削除されない。ヒストリを構成する裏方の計算ノードのみが対象。
- 出力コネクションが無くなった非DAGノード（出力が全て絶たれた計算ノードは不要である）
- 入力コネクションが無くなった非DAGノード（入力が全て絶たれた計算ノードの出力は一定になる筈だから不要である）

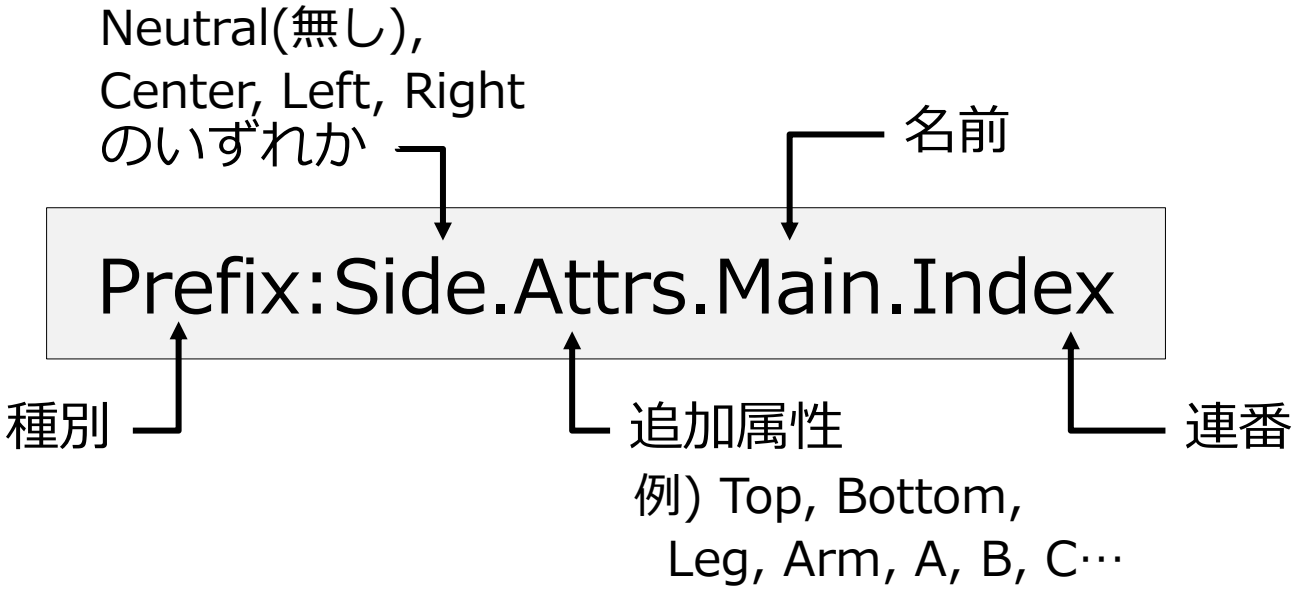
リグの簡単削除の仕組み ～ CRAFTの場合

- 基本は Maya® の連動削除ルールに則る。
  - リグ階層のトップノードに、非DAGノードであるメタノードの唯一の出力 message を接続する。よって、リグ階層が削除されればメタノードも連動削除される。
  - メタノードには、自身が削除される道連れに関連ノードを削除する機能を持たせている。連動削除ルールに適合せずゴミ化しそうなノードはメタノードに束ねておく。



ラベルについて

ラベルのフォーマット



ラベルは単なるタグ付けではなく、より語義的で且つ柔軟な解釈を可能にしている。

- 関節数の可変
- 左・中・右の可変
- 適合部位の可変（同じリグを腕や脚に用いるなど）

リグ作成時のジョイント参照

例えば、尻尾リグを作成する場合、”Tail” というラベルが付いたジョイントが探されるとする。

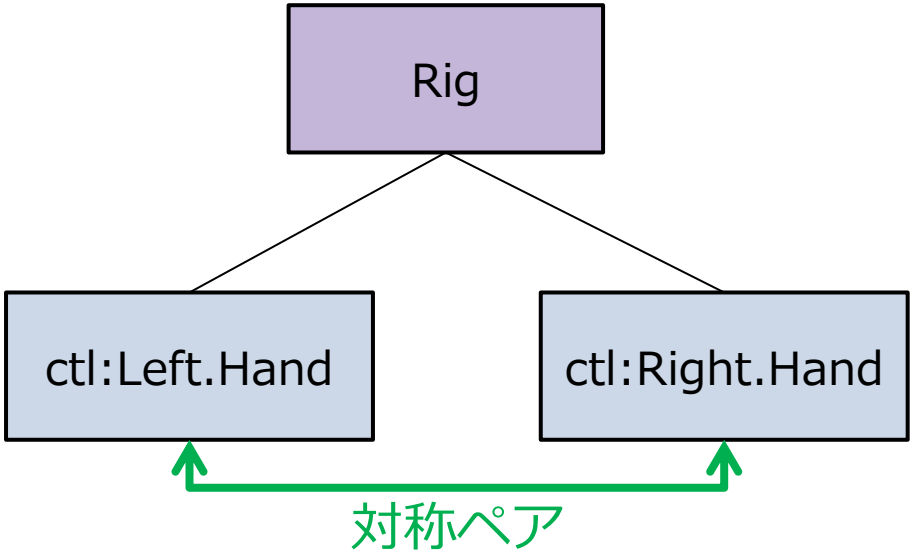
- 関節数が可変？
  - 関節一つの “Tail” にマッチ。
  - “Tail.0”, “Tail.1”, … のような連番の複数関節にマッチ。
- 左右に尻尾？
  - リグ作成時に、例えば Left と指定すれば、”Left.Tail.*” にマッチ。
- もっと尻尾？
  - リグ作成時に、例えば Left と Top と指定すれば、“Left.Top.Tail.*” にマッチ。

ラベルでの左右ペアリング

- アニメーションのミラー処理での識別で利用
  - Center はそれ自身がミラー対象
  - Left と Right をペアリング
  - Neutral はミラー対象外
- さらに、リグの階層によって、左右属性は階層的に解釈される。

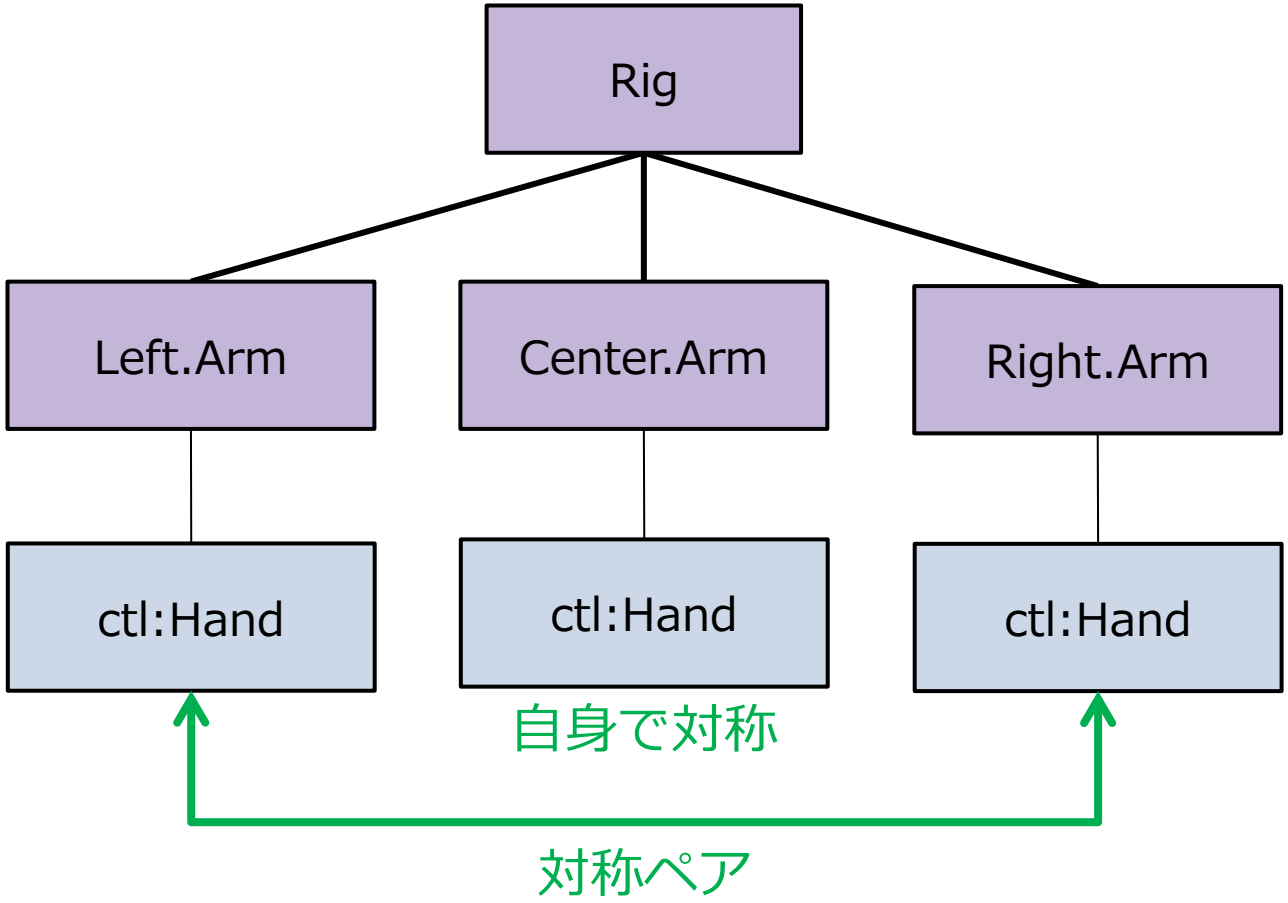
左右ペアリング例（１）

一つのリグが左右のコントロールを持つ例。



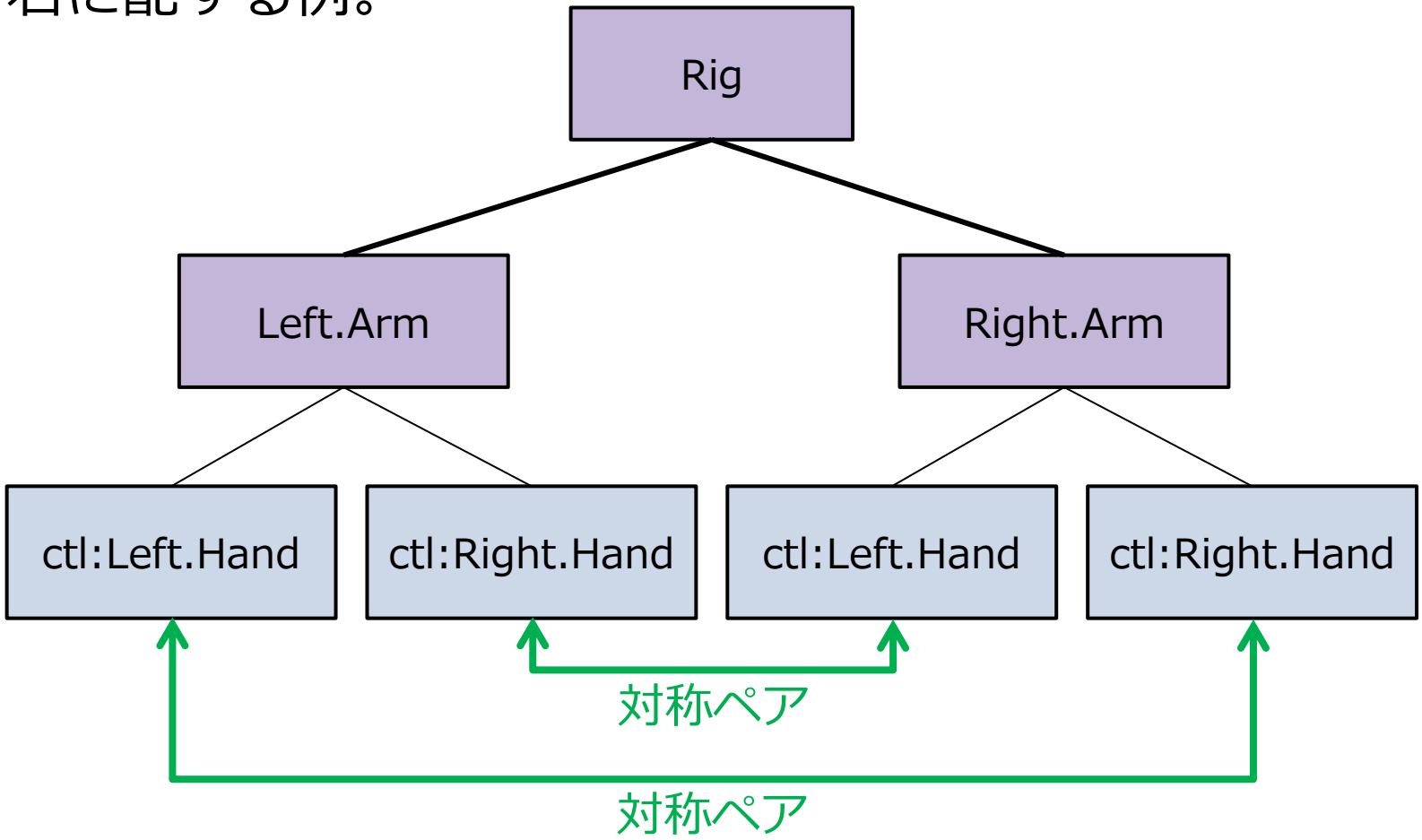
左右ペアリング例（２）

３つの Arm リグを持つリグの例。



左右ペアリング例（３）

左右のコントロールを持つ特殊な Arm リグを体の左右に配する例。



公開アトリビュートの管理

リグのどの部分（アトリビュート）が操作対象か？

- メタノードに ctl 種別で登録されているノードやアトリビュートから一定のルールで公開アトリビュートを抽出する。
- さらに用途別の分類もある。
  - Keyable
  - Editable
  - Pivot
  - Invisible

（参考）公開アトリビュートの抽出

メタノードに ctl 種別で登録されているノードやアトリビュートから以下のルールで抽出する。

- Maya®のチャンネルボックスに表示されているアトリビュート（keyable 又は channelBox フラグが on のもの）。
- ロックされていないピボット関連アトリビュート（Maya® の文化では、ピボットポイントは容易に変更可能なため）。
- ノードでなくアトリビュートが直接登録されている場合は、それらそのもの。

（参考）公開アトリビュートの分類

登録状況により、以下の分類で扱われる。

- Keyable
  通常アニメートすべき箇所。
- Editable
  調整可能だが、通常はアニメートしない箇所。
- Pivot
  ピボットポイント関連。
- Invisible
  通常ユーザーは触らないが、管理された箇所。

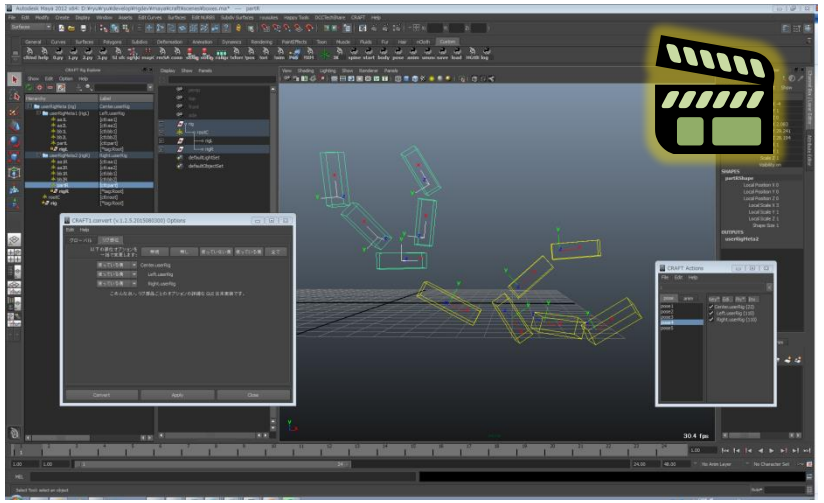
userRig について

- CRAFT には userRig という中身が空っぽのリグモジュールが在る。
- リグモジュールを使わずユーザー独自に作ったリグでも、そのコントロールノード群にラベルを付けて userRig に登録していくことで、CRAFT に認識させることが出来る。
- CRAFT によって、そのリグを構成するノード群や左右ペアリングの状態が認識され、公開アトリビュートも明確になる。
- CRAFT の便利な諸機能が利用できるようになる。

公開アトリビュートに対する諸機能

操作対象のアトリビュートがきちんと管理されていることで様々な便利機能を提供できる。

- クリップ（ポーズ／アニメーション）の保存管理
- リグのアニメーションデータの出荷
- 共通のミラーポーズ／アニメーション機能
- 全体操作
  - キーを打つ
  - キーを削除
  - アニメーションを削除
  - ポーズを初期化



リグモジュールの組み合わせについて

リグモジュールの組み合わせ

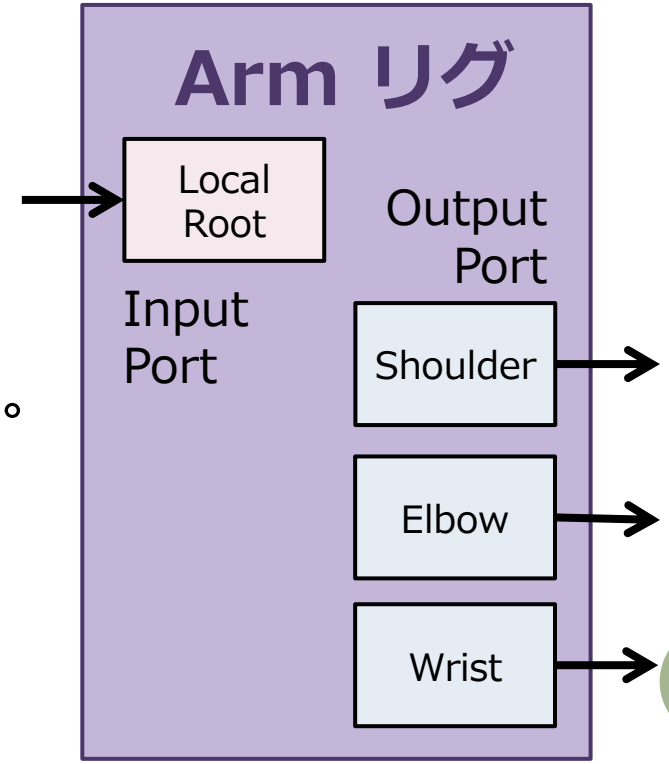
- モジュラーリグシステムは、そもそも個々の部品の独立性が高いことが売りなので、リグを部品ごとに構築していくと、そのままでは個々の部品は連動せずバラバラである。
- CRAFT では、大きく分けて二種類の方法で、モジュール間の連携を実現している。
  - リグのトランスフォーメーション・ポートを接続することで従属関係を設定する（生成後に従属箇所を接続）。
  - 他のリグモジュールに取り付けるタイプのリグモジュールも在る（生成時から連動前提）。

トランスフォーメーション・ポート

各リグモジュールには、外部とトランスフォーメーションの従属関係を構築するための入出力ポートが定義されており、それらを接続する。
平たくいえばコンストレイン。

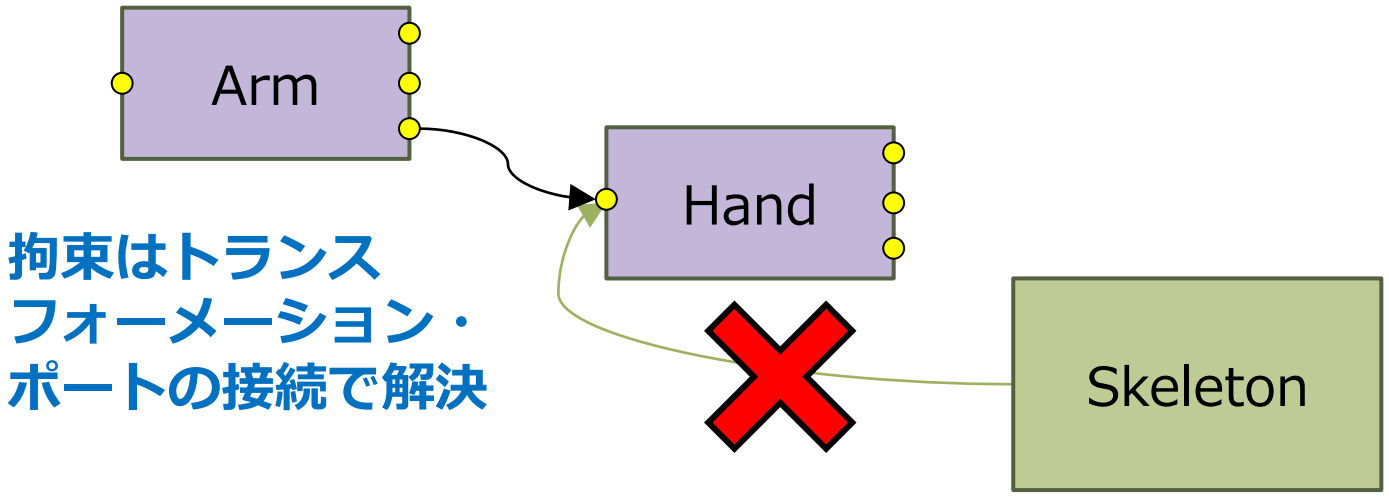
- Input Port
  そのリグのコントロールの置かれる空間定義をポートとして持つ。
  通常はローカルルート空間一つだが、複数の入力を持つリグも在る。

- Output Port
  多くの場合、そのリグのジョイント数だけのポートを持つ。



トランスフォーメーション・ポート

- 従属関係はリグモジュール間のみで解決し、スケルトンからは拘束しない。
  - コントロールリグとスケルトンの独立性を高め、いつでも完全な切断と再接続を可能にしている。
  - 例えば、Handリグはスケルトンの Wrist 関節に拘束するのではなく、Armリグの末端ジョイントの出力ポートを Hand の LocalRoot 入力ポートに接続して拘束する。



他のリグに取り付けるタイプのリグの例

- **Foot (フロアコンタクト)**
  脚のIKリグに取り付け、先端のハンドルの制御を奪い、
  複雑なフロアコンタクト機能を提供する。

- **IKHandle**
  IKリグに取り付け、先端のハンドルの制御を奪い、
  Local/World系の切り替え機能を提供する。

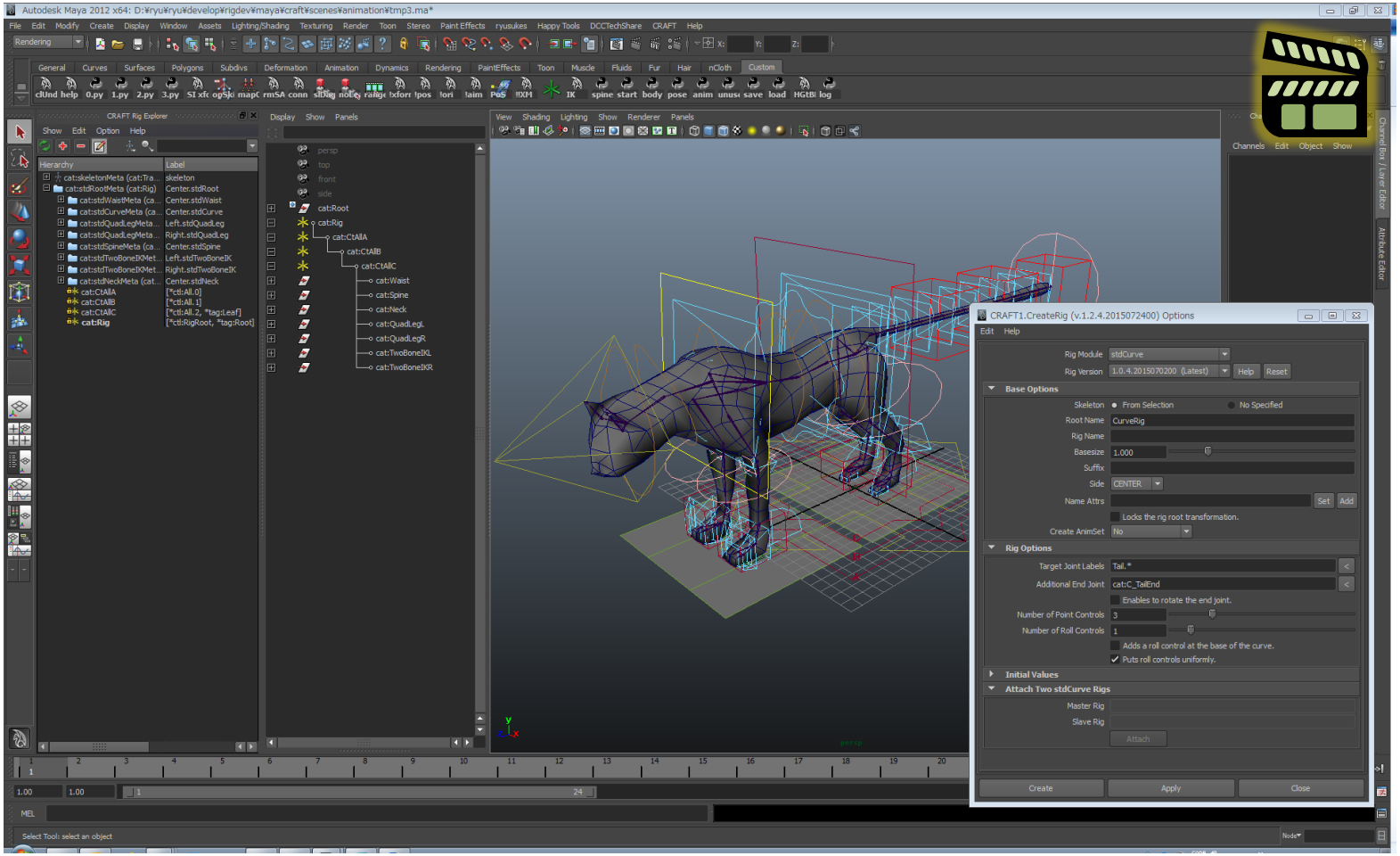
- **Shoulder (自動肩)**
  腕のIKリグに取り付け、腕の肩部分の回転コントロールを
  奪い、仮想的な肘位置から肩の自動的な動きを作り出す。

- **KneePinning (膝固定)**
  二つの Leg リグを入力とし、その膝位置を入力として
  Waist リグのオフセット制御を追加する。

モジュールデザイン

- 先に紹介したどちらの方法にも共通する重要な点は、インタフェースを明確にするということ。
- 機能をモジュール分割し個々の独立性を高めようとすると、一体型で成り立つ高度な機能が作りにくくなる。
- しかし、どんなに密な結びつきが必要な部分でも、必要なインタフェースを明確にしていき何とか分離することで、モジュールの再利用性が向上するのはもちろんのこと、結局はメンテナンス性も向上する。
- ただし、開発コストは余分にかかる。
- 多少のオーバーヘッドには目を瞑る。

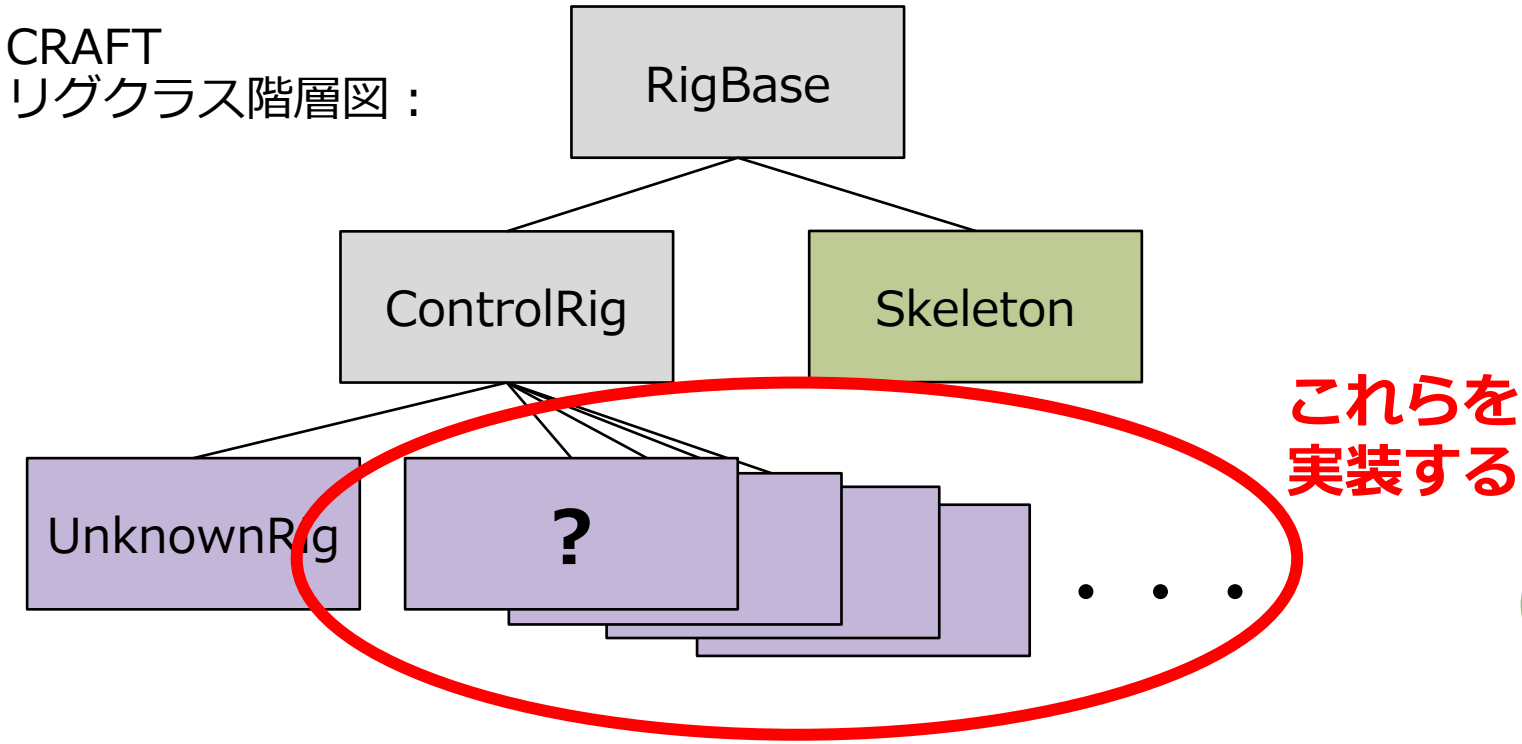
モジュラーリギングの実演



リグAPIとモジュールの実装

リグの Python クラス

- コントロールリグやスケルトンは CRAFT API 上では Python のクラスとして表現される。
- リグモジュールの実装とは、Python でリグクラスを実装して CRAFT に登録すること。



リグクラスの実装

ControlRig を継承したリグクラスを実装。

- さらに機能別のクラスを実装し、それらを保持する。

  機能クラスは、通常は最新版のみ保持すれば良いが、複数バージョンを保持することも可能。

- 必要に応じて細かなメソッドを追加実装。殆どは基底クラスからの継承で事足りるので、通常は追加実装することは何もない。

- このクラスをシステムの rigRegistry に登録すると認識される。プラグインのようなもの。

機能別のクラス実装

機能別に以下のクラスを実装し、リグクラスに保持する。

- RigBuilder を継承したビルダークラス
  - リグ生成処理を実装する。
- RigConverter を継承したコンバータークラス
  - コンバート機能（コントロール間の変換、ミラー、リターゲット等）を実装する。
  - 省略した場合、システムの提供する BasicConveter が使用される（共通のミラー機能を利用可能）。
- UI系のクラス
  - 各機能のオプションボックスや、独自のGUIなどを実装する。（現状、一部の仕組みのみ提供）

ビルダークラスの実装

- 与えられたスケルトンと、ラベル指定によって対象ジョイントが決定され、それに応じてリグ生成する。
- 各モジュールは個別の生成オプションを持ち、それに応じてリグ生成する。
- 処理全体は **BuildContext** 制御下におかれる。
- 公開するコントロールや、外部インターフェースとしてタグ付けしたいノードやアトリビュートをメタノードに登録する。
- ジョイントの拘束には **Connector** という概念を利用するのが推奨。

BuildContext の主な役割（１）

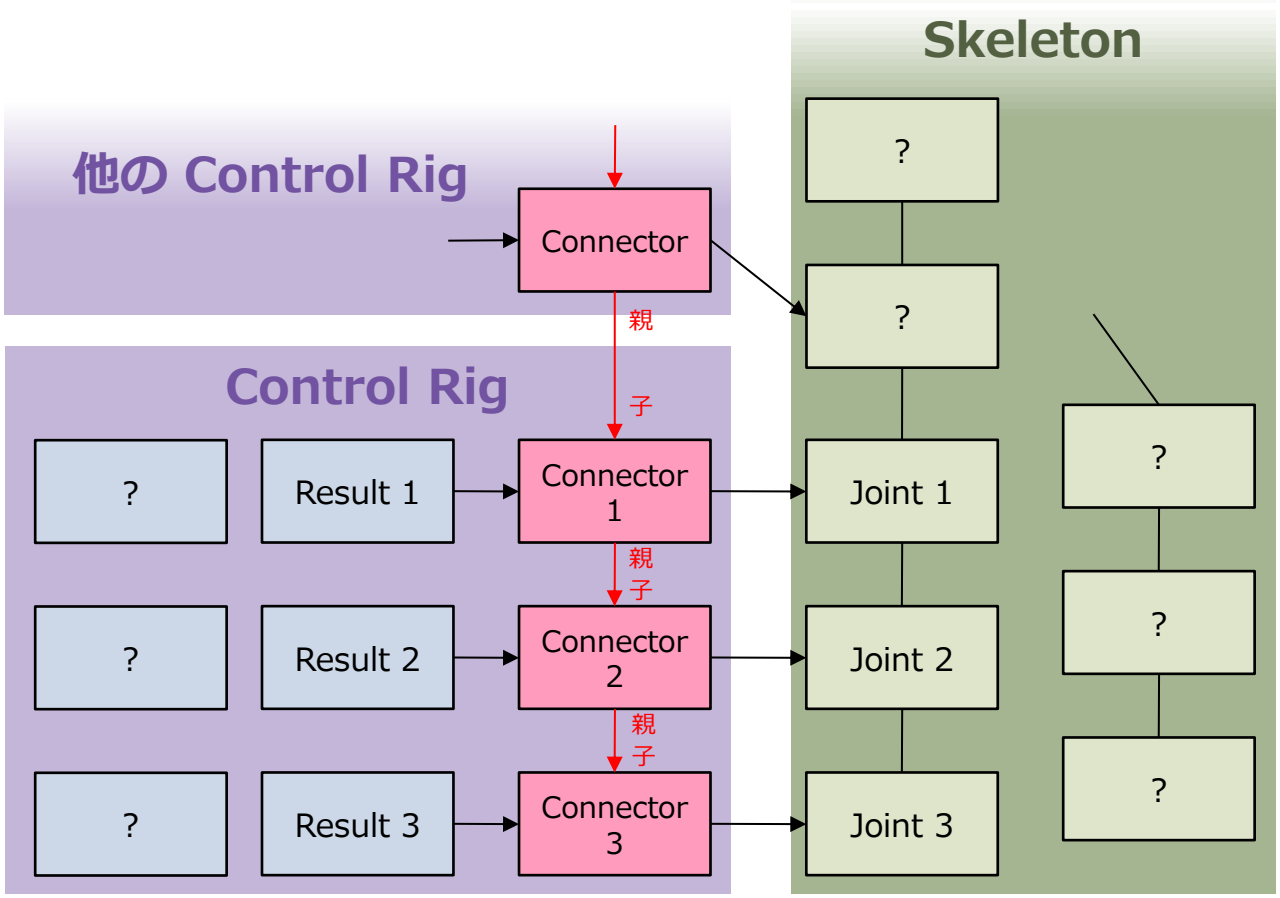
- ネームスペース、又はプレフィックスの付加
  - インポートやリファレンスされたキャラクタに指定されたものを引き継ぐ。
- サフィックスの付加
  - 同じリグを複数箇所に生成するとノード名の重複が起こる。それは問題ないものの、Maya®でパス名で扱われるようになるのは出来れば避けたいため、サフィックスを付加する。
  - 柔軟性の為に、モジュール実装でハードコードさせない。

BuildContext の主な役割（２）

- リグのベースサイズの管理
  - 関節配置からコントロールの視覚的なサイズは自動決定されるべきだが、それだけだと限界はある。
  - 生成オプションで指定された全体スケールを管理する。
- コネクションの自動ロック
  - その後の操作ミス防止のため、リグ生成コードが作ったコネクションは自動的にロックされる。

Connector とは

コントロールリグの最終的な結果を表す「ジョイント代理ノード」と「スケルトンのジョイント」を接続する独自コンストレインノードに情報管理を付加したもの。



Connector の役割

- 拘束するジョイントのラベルと、接続するアトリビュートの情報を保持する。
  - いつでも切断・再接続できる。
- 入力ノードの軸方向と出力ノードの軸方向の差異を管理し吸収する。
  - リグの軸方向はそのモジュールでは共通だが、スケルトンの軸方向はプロジェクトごとに異なる。
  - スケルトンからコントロールリグへの変換や、他スケルトンからのリターゲッティング時の情報源となる。
- 高速化のため、ローカルコンストレインが使われる。

  参考：CEDEC 2014 TA Bootcamp 『ちょっとテクニカルにリギングしてみよう』（CEDiLに資料が在ります）

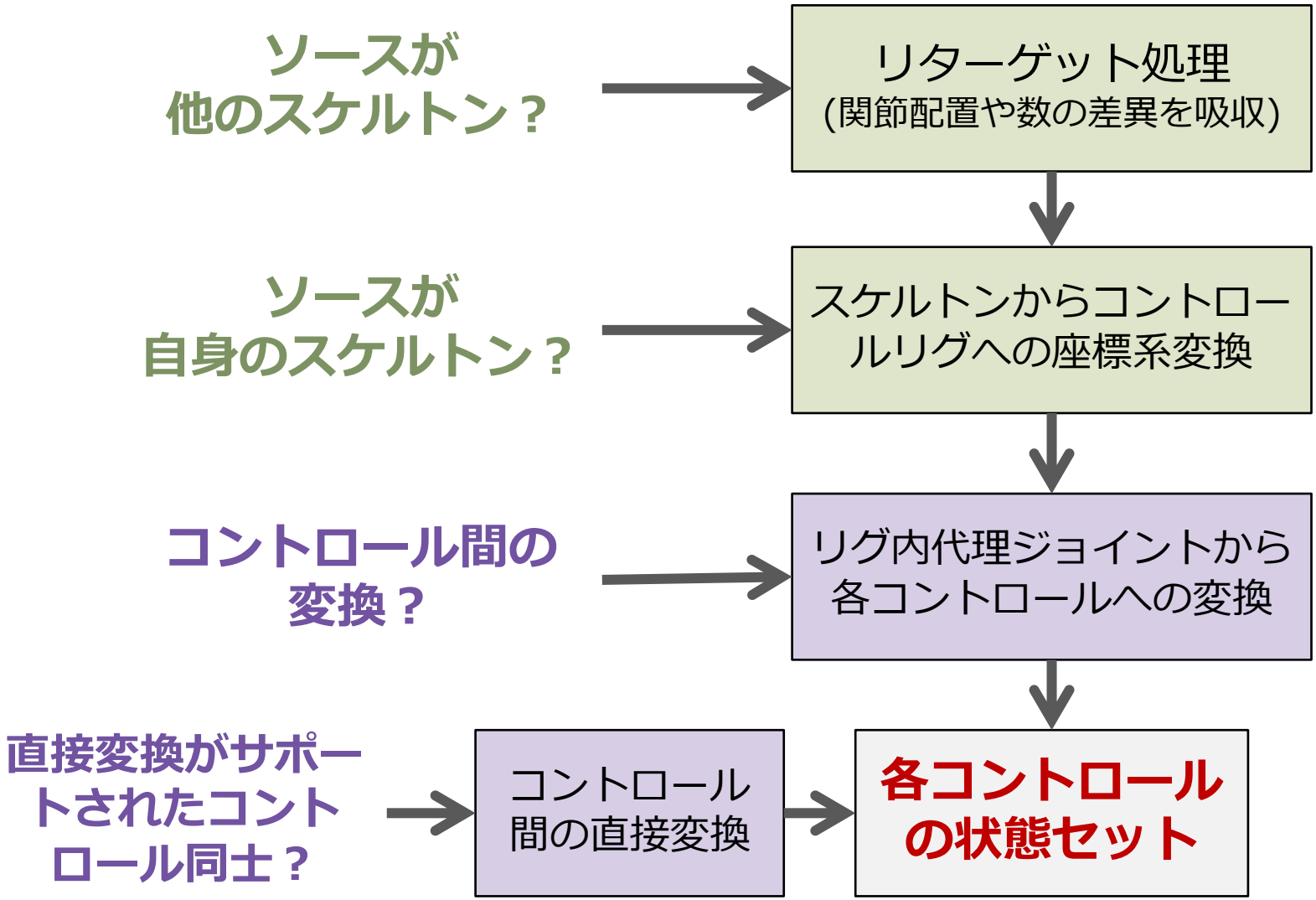
標準リグで拘っていること

- 基本的な心構えとして、何らかの前提を期待するような実装、ハードコーディングは極力避ける。
  - スケルトンのジョイントの位置しか参照しない。
    軸方向は一切参照せずに、関節配置からリグの座標系を決定する。
  - エンドジョイントは無くてもリグ生成可能にする。
    付加的な情報は在るに越したことはないが、無くても推測で成り立つようにする。
- ユーザーが触るべきでない箇所は全てロックする。
- リリース後の変更によるモーション互換性に気を配る。
- 高速化のため、モジュール内ではローカルコンストレインを多用する。
  - ちなみに、モジュール間の拘束（トランスフォーメーション・ポート）ではワールドコンストレインを使う。

コンバートとは

- 以下の機能を「コンバート機能」としてまとめる。
  - コントロール間変換（Local/World, IK/FK など）
  - ミラー
  - スケルトンからコントロールリグへの変換
  - 他キャラからのリターゲッティング
  - ポーズ or アニメーション
- 全部別々には実装したくない！！ → 抽象化
  - いずれも「何らかの情報ソース」からリグのコントロールに「値」をセットする処理である。
  - 「ミラーかどうか」はそれに対するパラメータ。
  - 「ポーズかアニメーションか」もパラメータ。

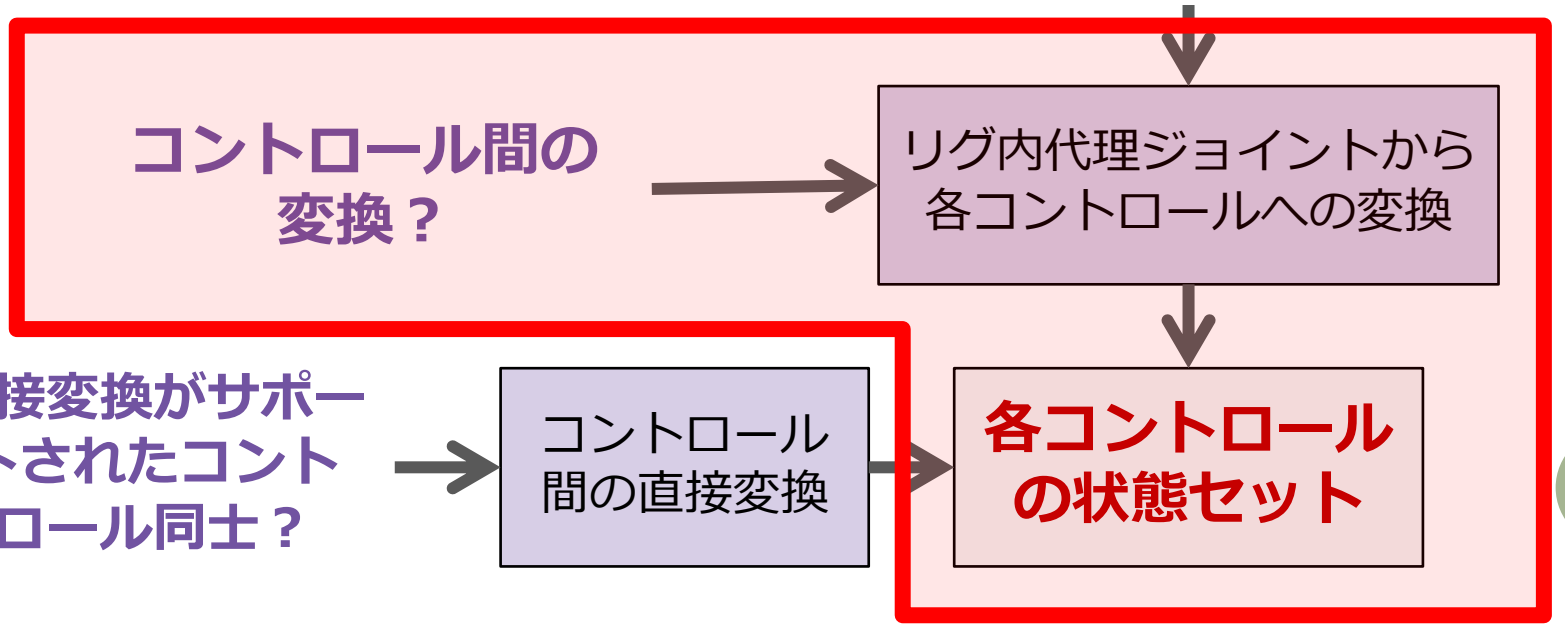
リグコンバーターの処理概要



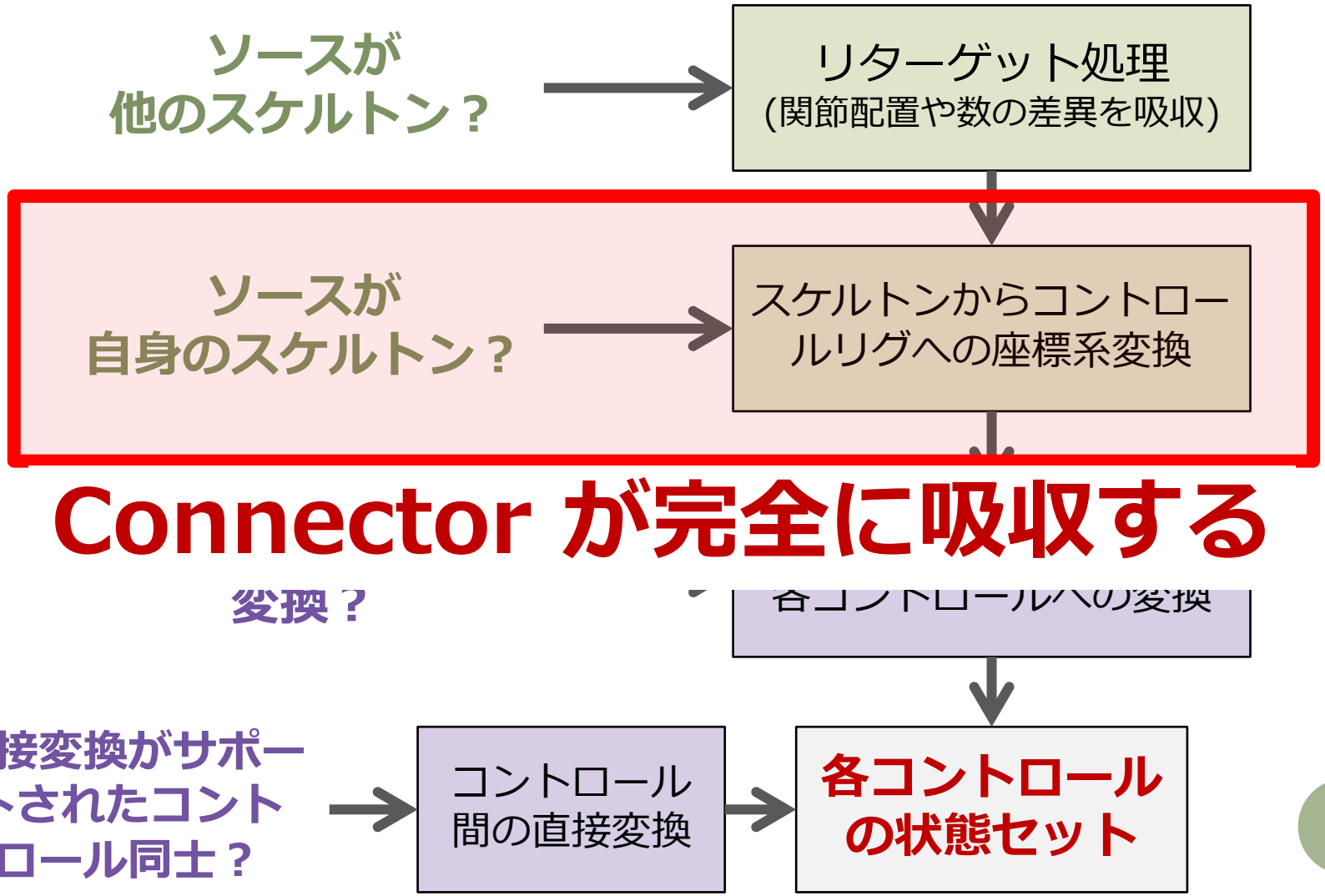
リグコンバーターの処理概要

コンバーターの主となる実装

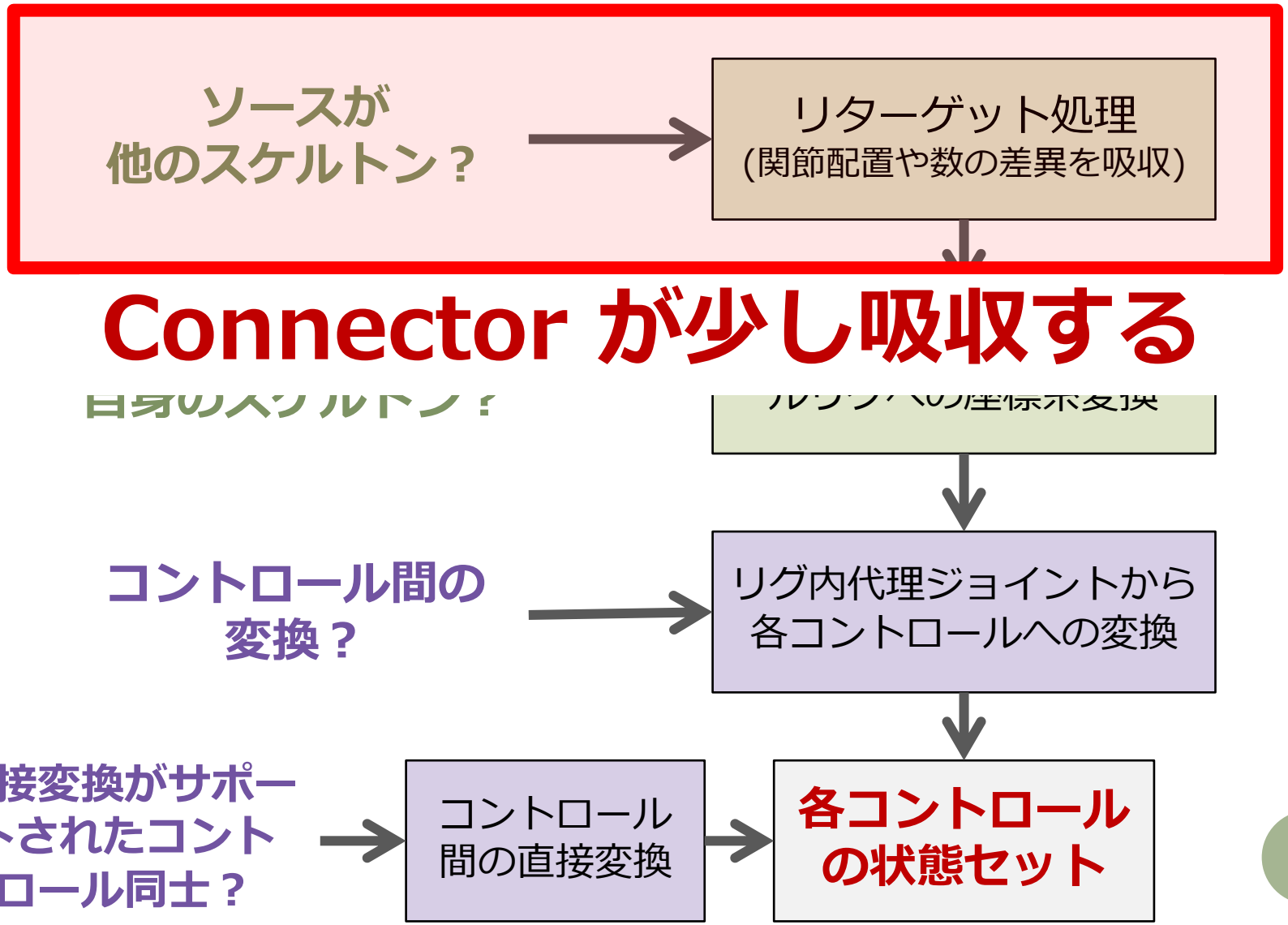
Connector から状態を得て
コントローラへ変換する
（ミラー処理も Connector が吸収）



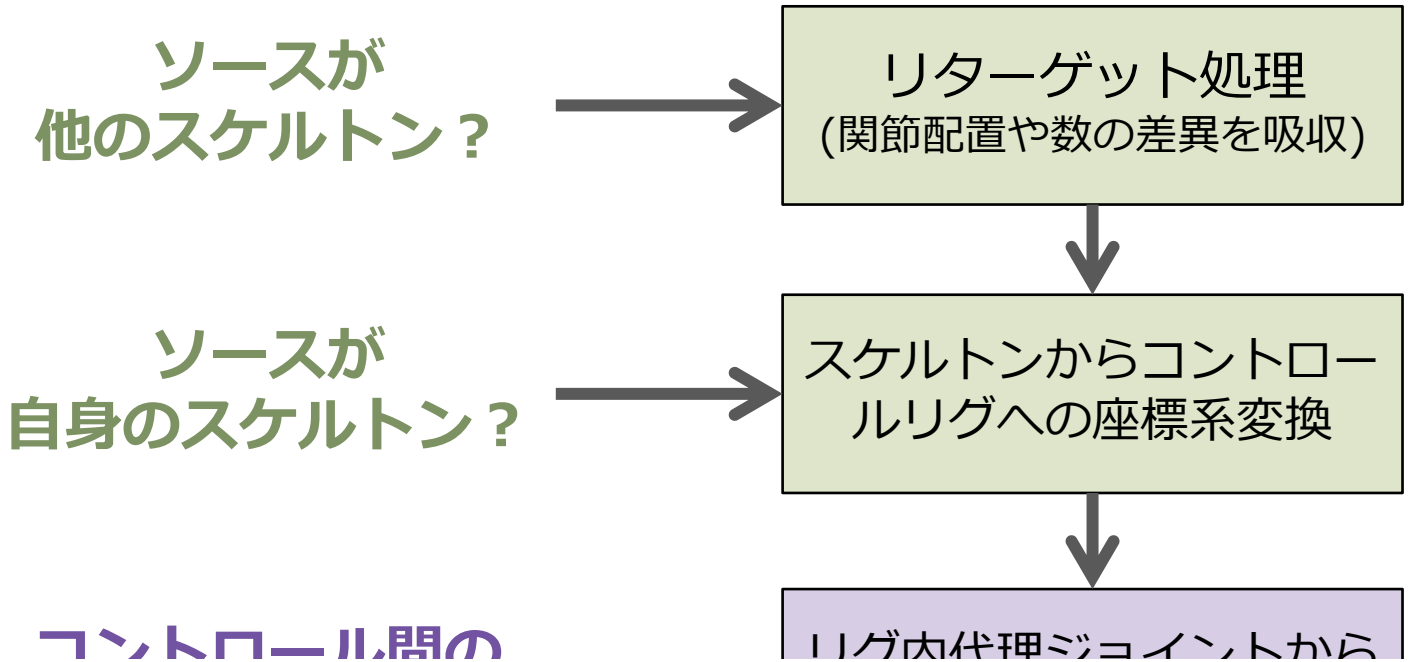
リグコンバーターの処理概要



リグコンバーターの処理概要

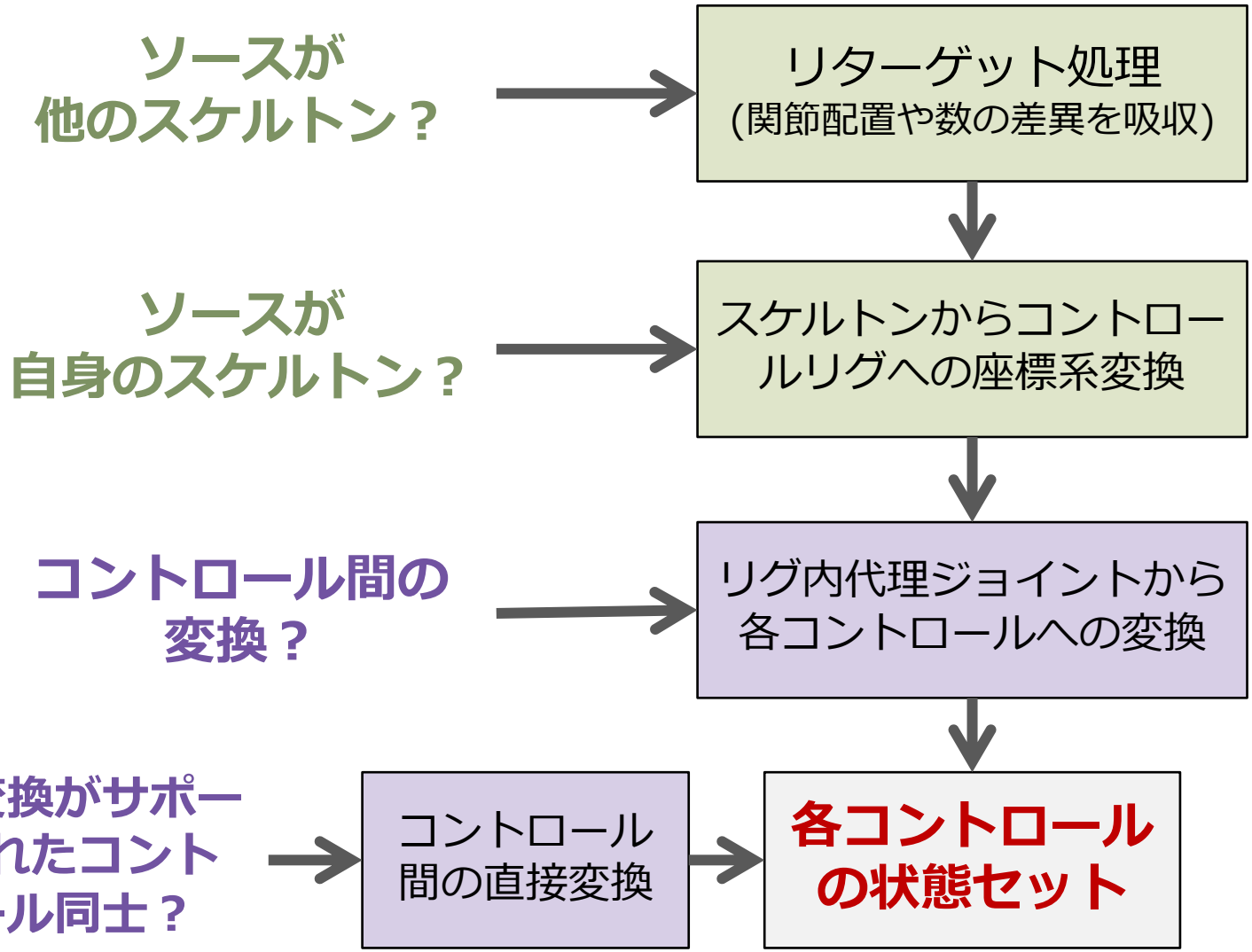


リグコンバーターの処理概要



より便利にしたい場合の特別対応

リグコンバーターの処理概要



コンバート処理の流れと抽象化のまとめ

- 何がソースか？
  - 他のスケルトン
  - 自身のスケルトン
  - コントロールリグ内の代理ジョイント
  - ~~コントロールそのもの（特別対応）~~
- いずれの場合も、コントロールリグ内の代理ジョイント相当のマトリックスを Connector から得る。
  - ミラーが必要？ → マトリックスをミラー
- セットするのは…　ポーズ？　アニメーション？

  ConvertContext が吸収。

ConvertContext の主な役割

- 実行中のリグコンバーターオブジェクトの保持と呼び出し制御。
- コンバーター共通グローバルオプションの保持。
- キャンセル処理や各種後始末処理。
- ポーズとアニメーションの処理の違いの完全な吸収。

ConvertContext による制御抽象化

各コンバータークラスは、値のセット（ポーズセット）のみ実装すれば良い。

アニメーションの場合、システムは…

- フレームを進めながらコンバーターのメソッドを繰り返し呼び出す。

  API の MAnimControl.setCurrentTime() を利用。シーン全体が評価されないので高速でお勧め。

- 「値のセット」の際に「キー追加」も行う。キー追加による影響を除外する制御も行う。

  nodewrapper を使っているからこそ出来る。

コンバータークラスの実装

- サポート範囲のリグバージョンの定義
  - 実際のリグに埋め込まれているバージョンに合わせて使用されるコンバータークラスが選別されるための情報となる。
- __init__()
  - 処理に必要な情報を収集し保持する。
  - 毎フレーム繰り返しやらなくて良いことをここでやっておく。
- getFromSource()
  - 変換ソースから状態を取得して保持する。
  - 子リグを再帰呼び出しする。
- setToControl()
  - 変換先コントロールに値をセットする。
  - 子リグを再帰呼び出しする。

コンバーター呼び出し制御の概略

簡略化した擬似コードを以下に示す。

```
with ConvertContext() as ctx:
    for rig in rigs:
        ctx.addConverter(rig.converter())

    while ctx.nextTime():
        for cnv in ctx.topConverters():
            cnv.getFromSource()
        for cnv in ctx.topConverters():
            cnv.setToControl()
```

各リグコンバーターはリグ階層に沿って順次呼ばれるだけなので、密な関係にあるモジュール同士はそれぞれ自身による細かな呼び出し制御が必要な場合もある。

まとめ

まとめ

- Maya®を徹底的に理解し、プラグインでリギング機能を置き換えつつも作法に忠実であることを守った。複雑なシステムだがトラブルは本当に少ない。
- 高い汎用性を持つ為に、何らかの前提を期待するような実装、ハードコーディングは極力避けた。
- モジュラリティの追求はリグ全体の高機能性の追求と相反するが、モジュール間インタフェースを慎重にデザインすることで何とかなる。
- Python 言語の利用と独自 API の開発は、リグのオブジェクト指向モデル構築と複雑な処理の抽象化に非常に有用だった。

ご清聴ありがとうございました。

何かご質問はありますか？

佐々木隆典 ryusukes@square-enix.com

SQUARE ENIX