

Pixar USD 入門

新たなコンテンツパイプラインを構築する

手島孝人 Polyphony Digital Inc.

スライドはセッション後すぐに CEDiL, SlideShare で公開します 撮影不要

自己紹介



- | | |
|-------------|----------------------------------------------------------------|
| 1992 - 1993 | KMC / DoGA
CG ツールエンジニア |
| 1996 - 1999 | ナムコ アーケード開発
CG ツールエンジニア |
| 1999 - 2011 | ポリリフォニー・デジタル GT シリーズ
ツール、パイプライン構築 |
| 2011 - 2017 | ピクサー・アニメーションスタジオ R&D
Presto, OpenSubdiv, <u>USD</u> , Hydra |
| 2017 - 現在 | ポリリフォニー・デジタル (US ベニススタジオ) |

はじめに

このセッションは

テクニカルディレクター・テクニカルアーティスト

ツールエンジニア・パイプラインエンジニア

向けに、

コンテンツパイプラインの諸問題をソフトウェアエンジニアリングで解決

する新手法として、**Pixar USD** を紹介します。

※バージョン管理をしましょうとか、DCC ツールの便利な使い方とか、物理ベースシェーディング、レビュー方法、などの話などはしません

目次

- USD 一般入門
 - コンカレントパイプライン
 - USD とは何か、USD をめぐる状況
 - ピクサーのパイプライン実例
- USD テクニカル入門
 - オブジェクトモデル
 - コンポジションアーク
 - スキーマ
 - API 構成
 - 課題、展望

わかりやすい話

ツールエンジニア
テクニカルアーティスト
向けの専門的な話

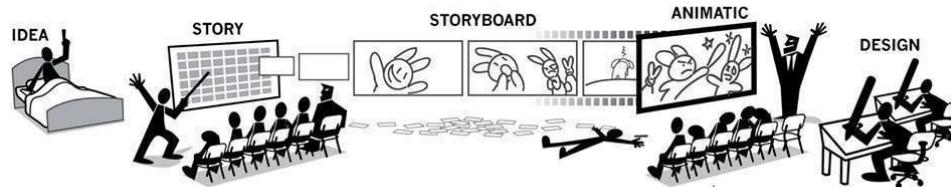
パイプラインとは

- ・ゲーム・映像制作のデータを、作業工程の間でやり取りする仕組み

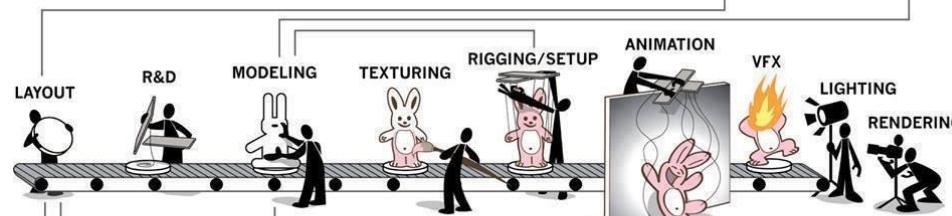


3D Production Pipeline

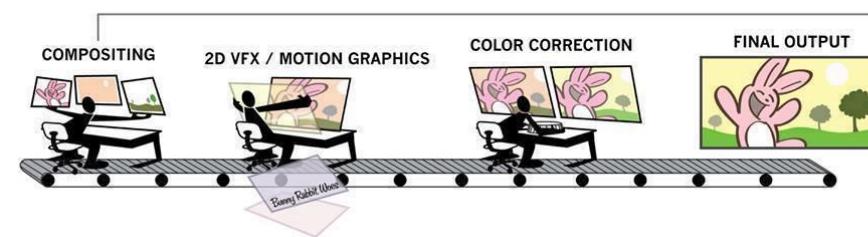
PRE-PRODUCTION



PRODUCTION

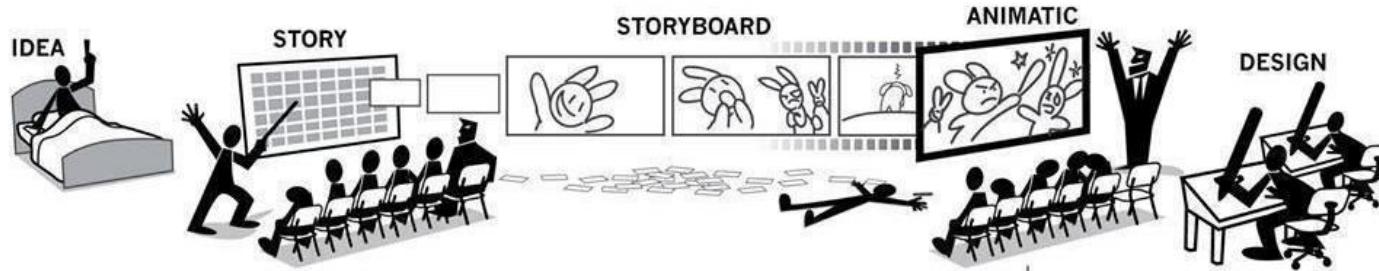


POST-PRODUCTION

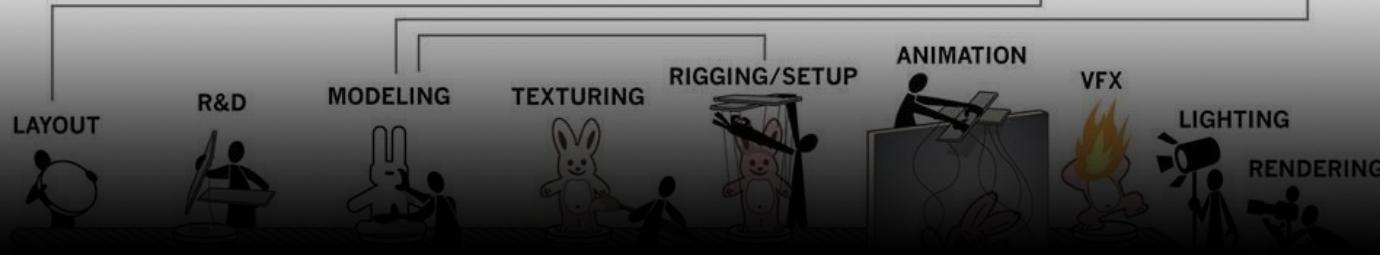


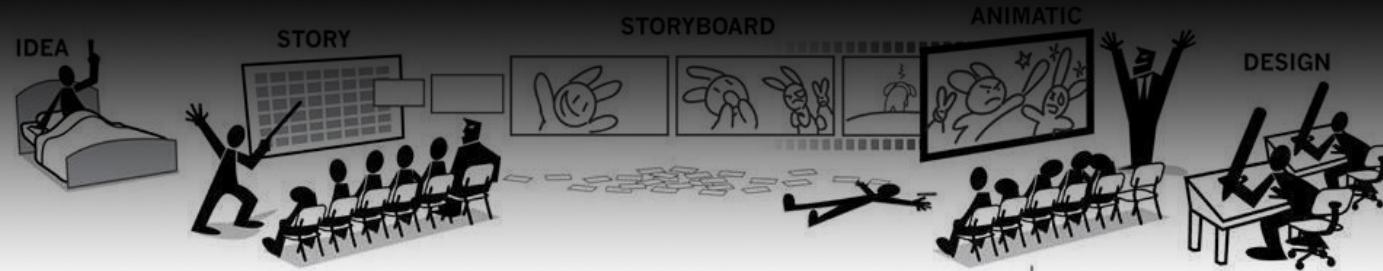
3D Production Pipeline

PRE-PRODUCTION

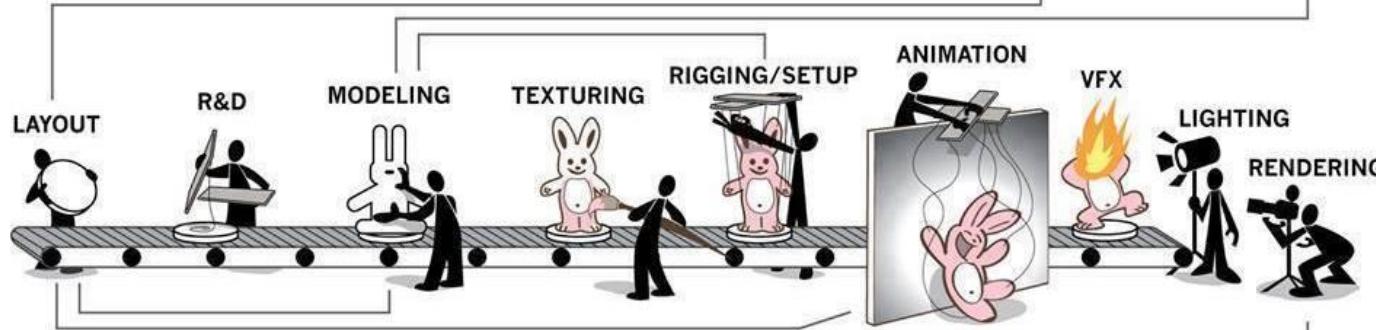


PRODUCTION

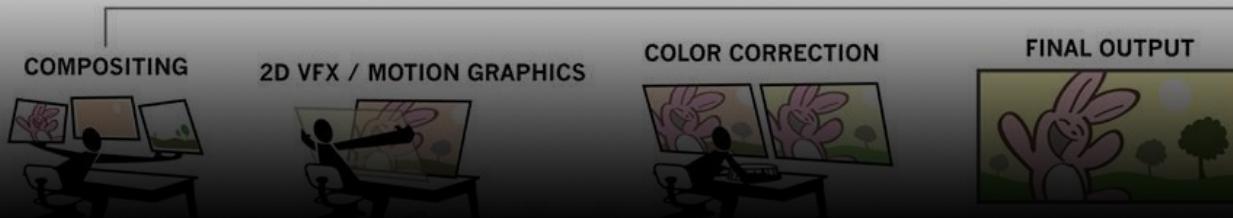


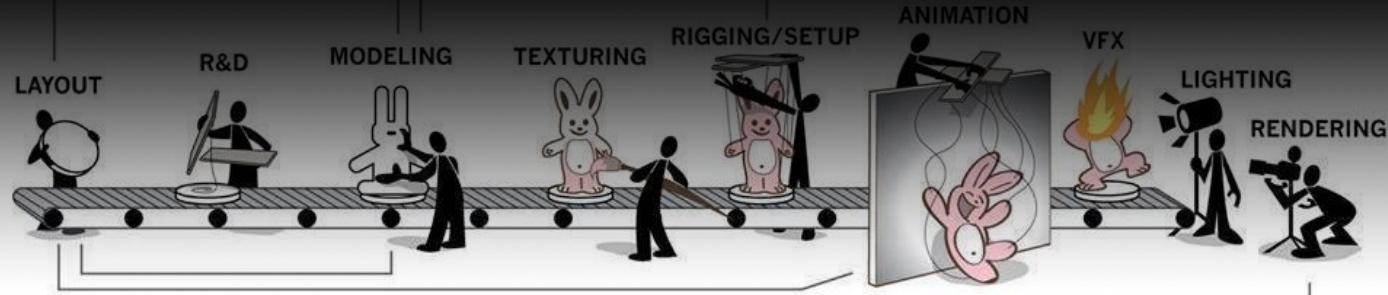


PRODUCTION

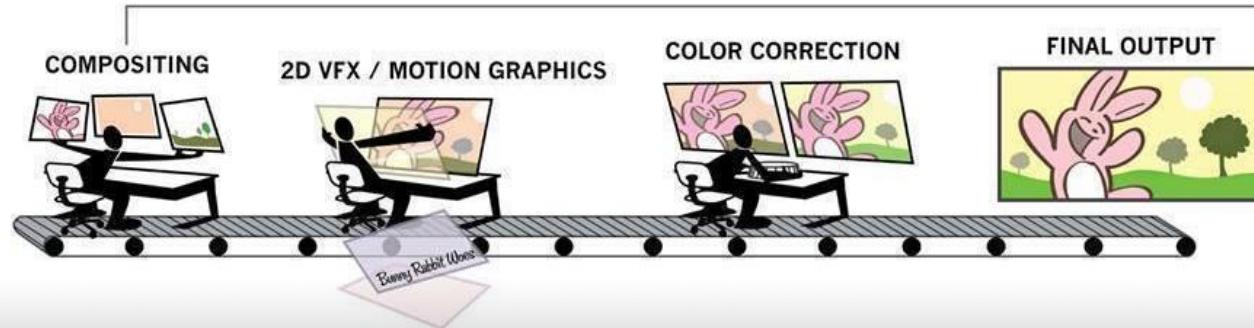


POST-PRODUCTION

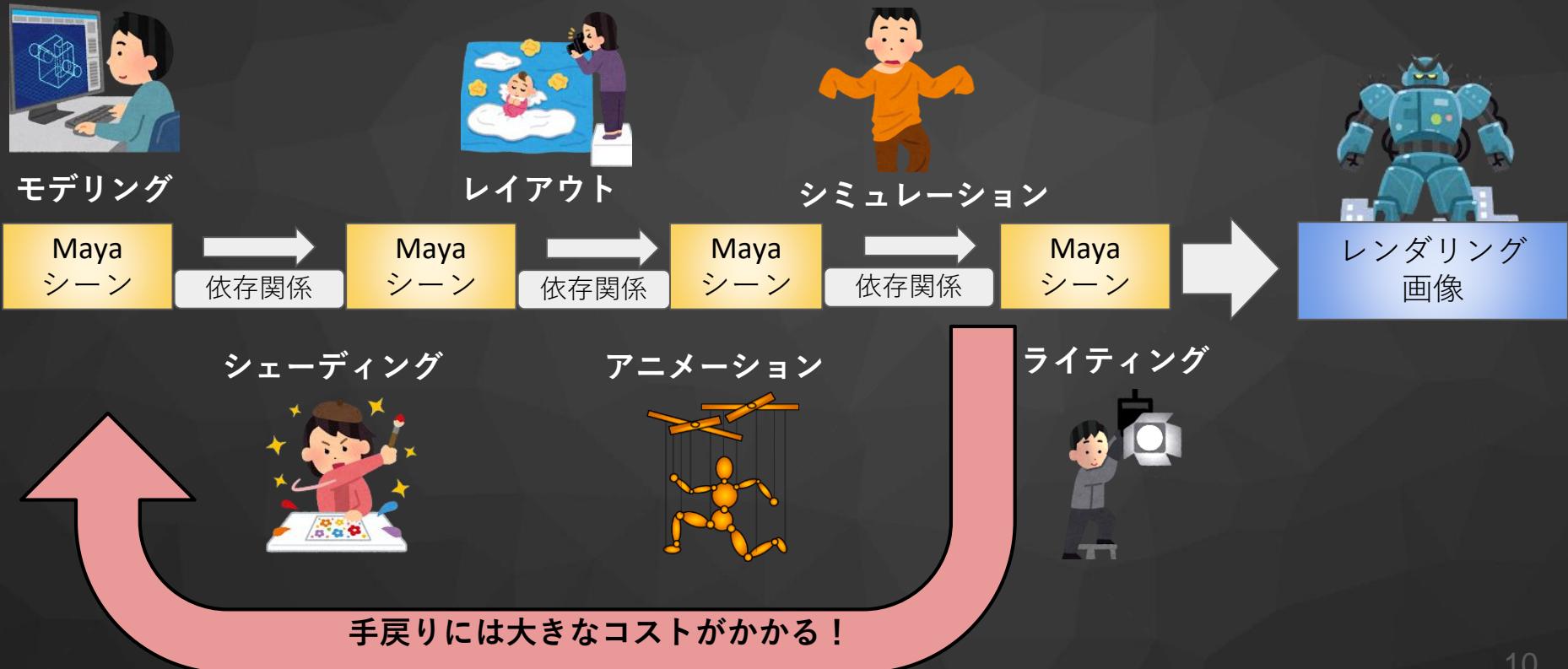




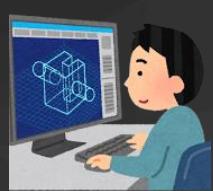
POST-PRODUCTION



古典的なウォーターフォール型パイプライン



コンカレントパイプライン



モデリング



シェーディング



レイアウト



アニメーション



シミュレーション



ライティング

モデル
データ

シェーダ

配置情報
・カメラ

アニメー
ション

アニメー
ション

ライ
ト



自動合成



各セクションが自由にイテレーション可能



プロトタイピング、クオリティアップが容易

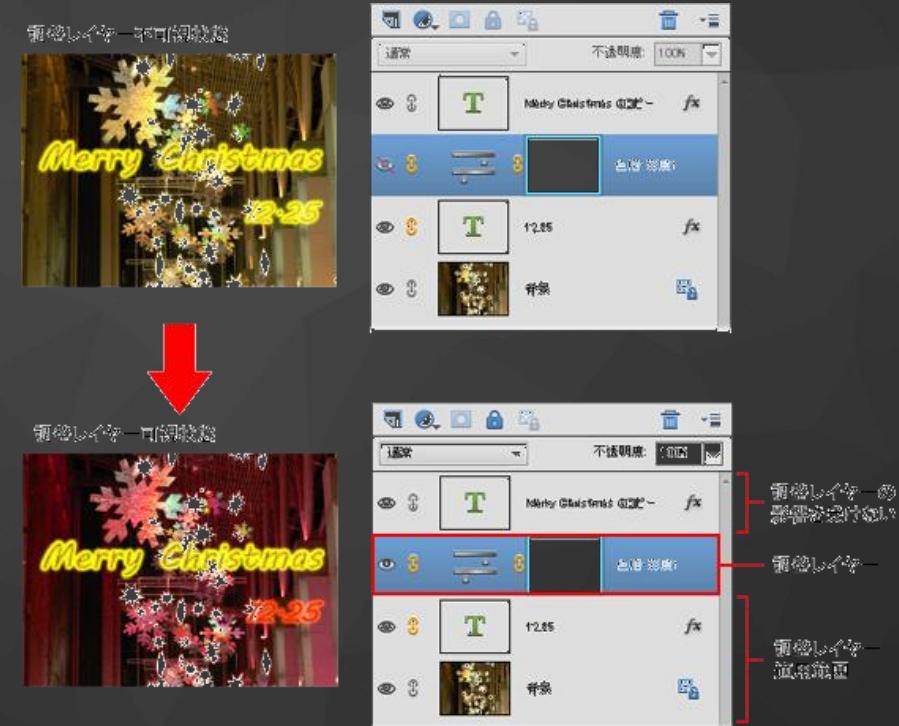
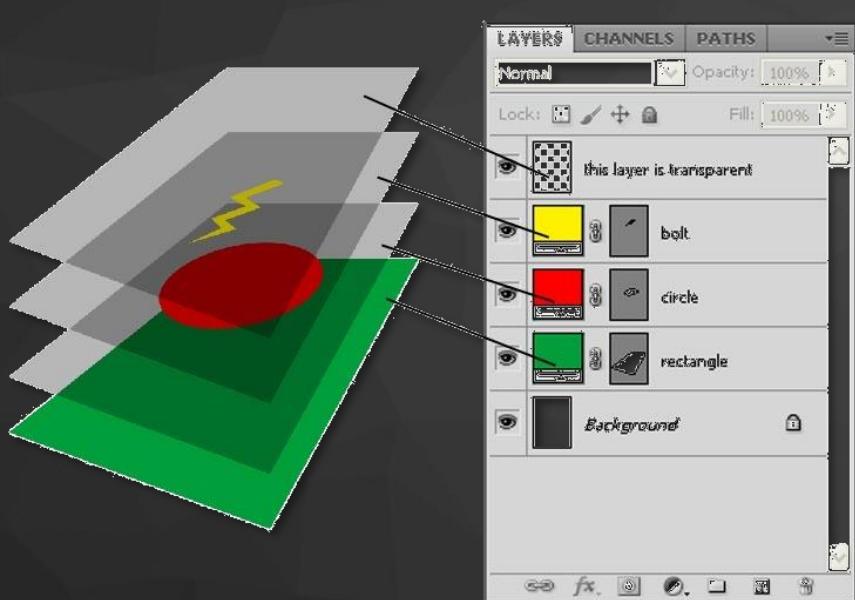
最終データ



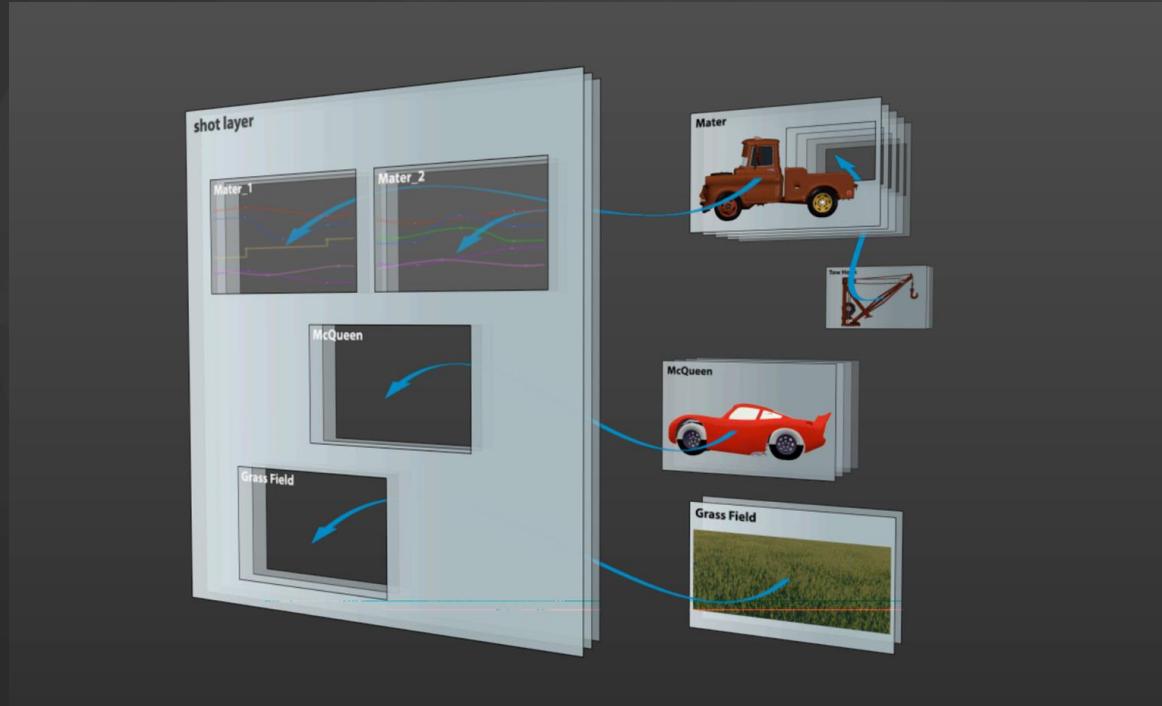
レンダリング
画像

※ゲーム制作では部分的に実現されている

Photoshop のレイヤー（非破壊編集）



3DCG でも同じ考え方ができるのでは？



しかし既存ソフトのリファレンス機能は使いにくい・・・ 13

ピクサーのチャレンジ



1995



1998

すべてのショットごとに、別々のプログラムでシーディングラフを記述



2012

基本的なリファレンスの仕組みを開発

シーディングラフコンポジションを体系化して整理
新システム Presto を構築



2016

Presto シーディングラフを USD として取り出し最適化
全パイプラインで共通利用。
オープンソース公開

マリオネット (MENV)

Presto シーディングラフ

パフォーマンス上の問題で
背景モデルなどはキャッシュを併用していた

TidScene (ジオメトリキャッシュ)

USD
UNIVERSAL SCENE DESCRIPTION



DCC ツール間でシーンをやりとりするファイルフォーマット

- シーングラフAPI
- プロシージャルシーングラフ処理系
- パイプラインそのものになる、「共同作業のための言語」

荒っぽく言うと…

FBX + Alembic + コンポジション = USD

今までのシーディングラフとの違い

スキーマシステムによる汎用性

強力な非破壊コンポジション機能

非常に高いスケーラビリティ

徹底したマルチコア最適化

大規模プロダクションで実戦投入ずみ

ピクサー内で20年間改良、4世代目

3Dコンテンツパイプラインのすべてで使える

フレーム間補間機能など



ALEMBIC



FBX

Open Inventor®

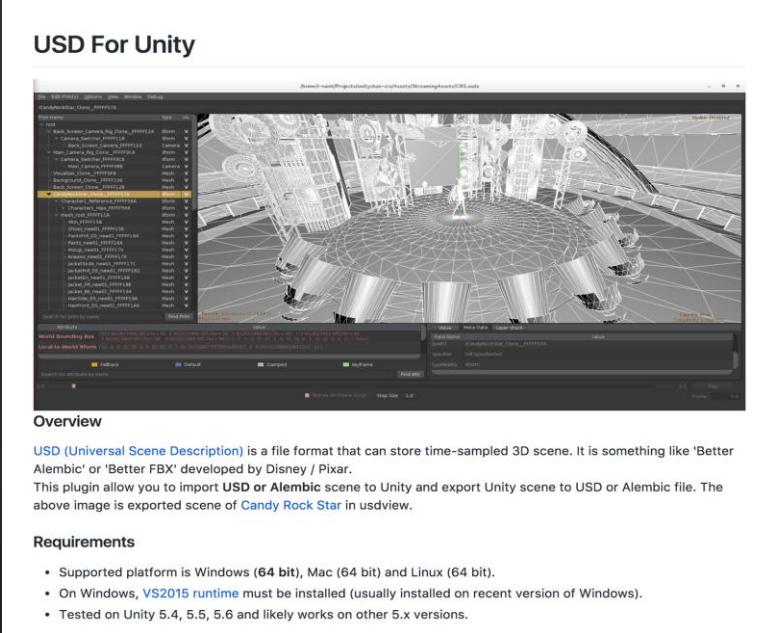
USD への映像業界のコミット

- Maya, Houdini, Katana など主要ツールは対応している
- ILM : Alembic を USD でレイアウトしている
- Animal Logic : パイプラインを XSI から USD に移行するため
AL_usdMaya プラグインを開発、オープンソース化
- MPC : 内製のファイルフォーマットを USD で置き換える
- LUMA : USD のアーノルドバックエンドを開発
- Polygon Pictures : Multiverse で USD 対応
- ピクサーが映画を作り続ける限りサポートが期待できる
 - ピクサーのパイプラインは USD
 - オープンソース版と同じコードベース
 - github のコミットログを見てください

Unity

Unity：プラグインで USD 再生が可能

<https://github.com/unity3d-jp/USDForUnity>

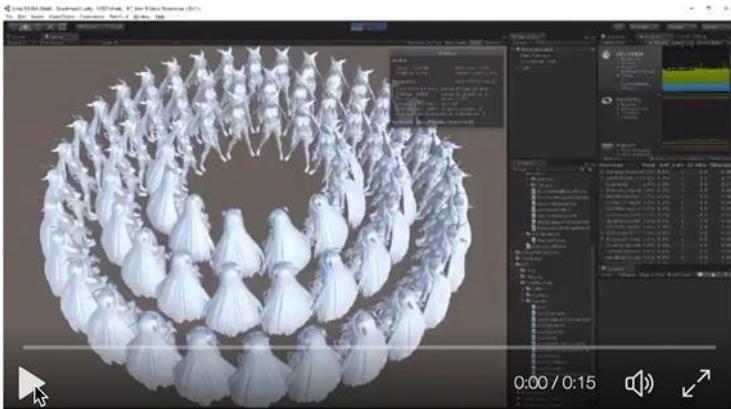


のツイート

フォロー中

i-saint
@i_saint

USD importer、いろんな工夫を重ねて 1M vertices @ 60 FPS のストリーミングを達成できた。これだけで次の Unite のネタにできそうな勢い。



9:15 - 2016年10月5日

98件のリツイート 201件のいいね

3 98 201

Unreal Engine

- Unreal Engine
VR エディタで USD のインポート可能
 - UE4/Plugins/Editor/USDImporter



GDC

Check out all of Polygon's coverage from Game Developers Conference 2017

During its Game Developers Conference presentation, Epic showcased the capabilities of its VR Editor using a scene from Pixar's *Finding Dory*.

Epic announced today that its editor would fully support Universal Scene Description. The tool is an open-source project used by a number of companies and editors, including Disney and Pixar, when working in animation. According to a description on Pixar's website, "Universal Scene Description (USD) is the first publicly available software that addresses the need to robustly and scalably interchange and augment arbitrary 3D scenes

- <https://www.polygon.com/2017/3/1/14778602/epic-gdc-finding-dory>

USD への Apple の取り組み

- Apple Scene Kit はすでに対応済み
 - OSX 10.12 なら標準ファインダで usd ファイルをプレビュー可能
- Apple GPU チームは積極的に USD, OpenSubdiv をサポート中
 - USD のスキーニングスキーマ、OpenSubdiv の Metal バックエンドは
 - Apple の GPU チームがコントリビュート

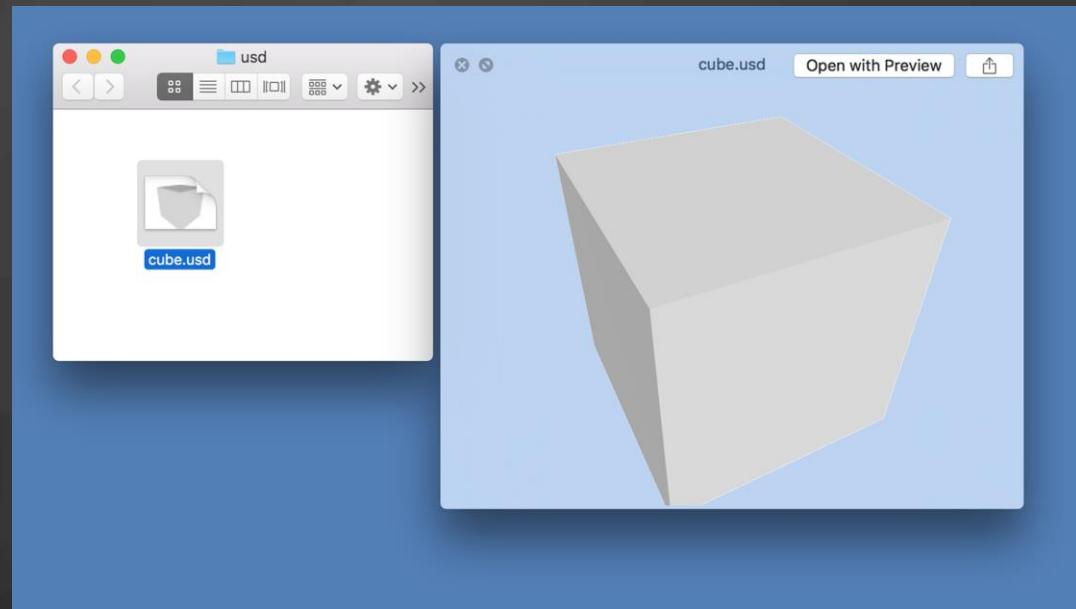
OSX の USD インテグレーション (SceneKit)

たとえば テキストエディタで次の 2 行を書き、
`cube.usd` というファイル名で保存する

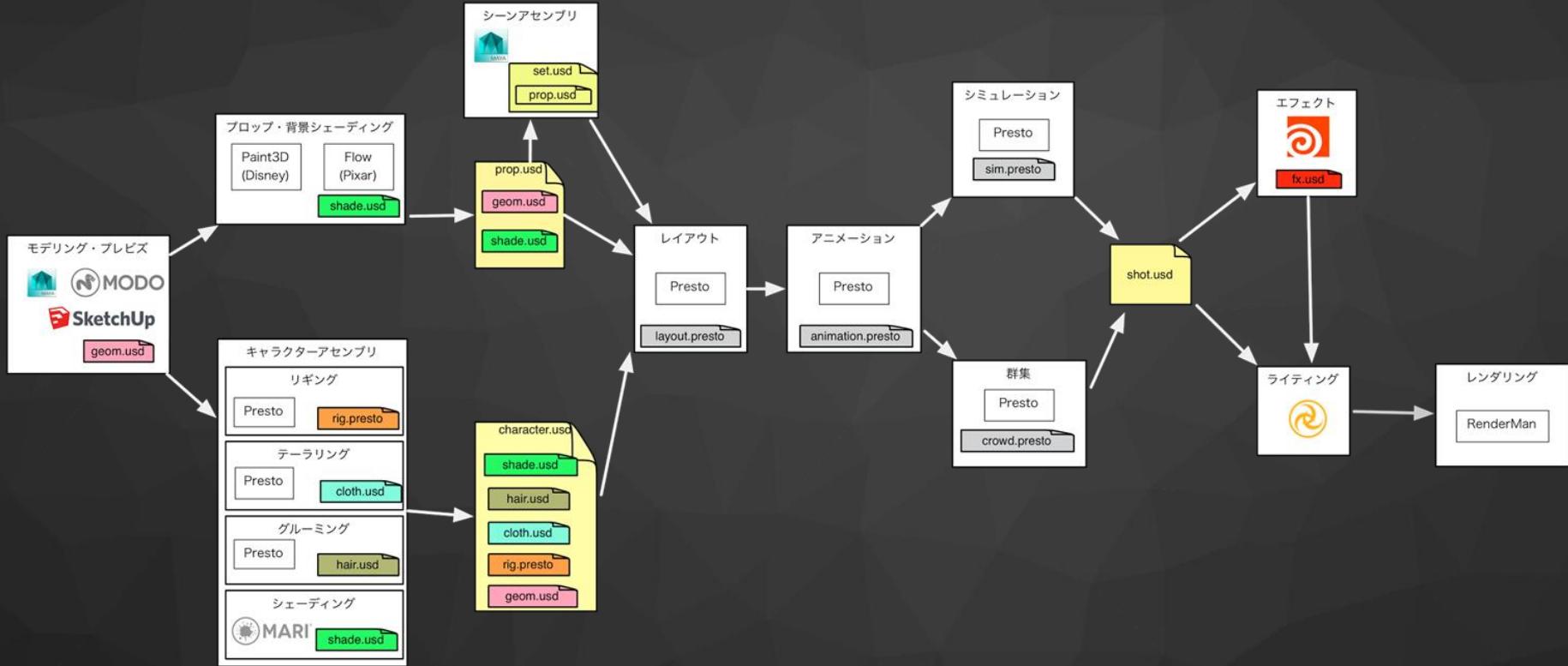
```
#usda 1.0
def Cube "Cube" { }
```

スペースキーでクイックルック
あるいは標準のプレビューで開ける

XCode にドロップすれば
シーンデータになる



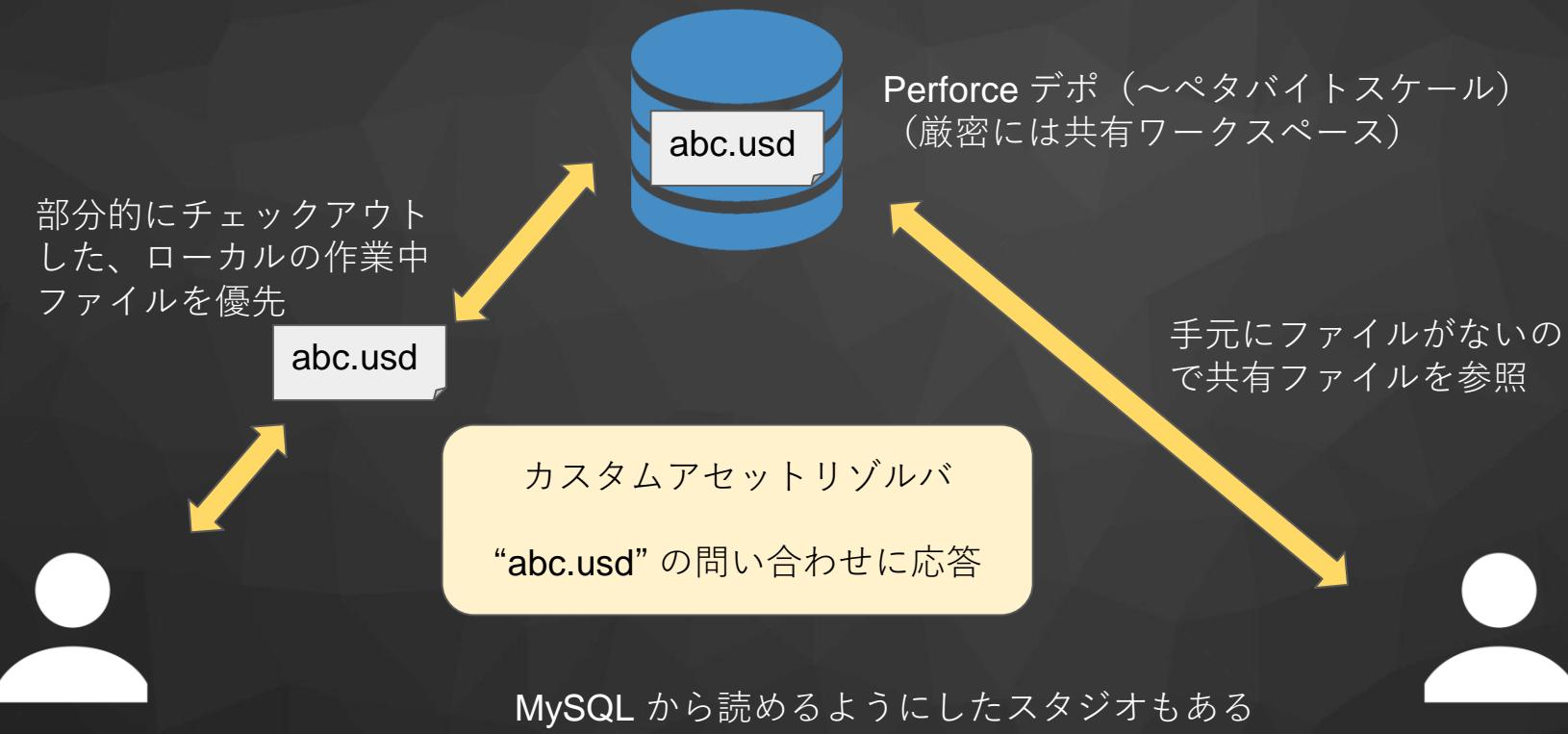
ピクサーサーパイプライン



USD は汎用で、拡張可能

- モデルプラグインで他フォーマットをそのまま取り込める
 - Alembic
 - OBJ
 - その他カスタムで
- スキーマ定義（後述）さえすれば、3DCG 以外にも使える
- アセットトリゾルバー プラグインで、あらゆるスタジオ環境にフィット可能
 - アセットトリゾルバーとは
あるファイル名 “foo/bar/geom.usd” が与えられたときに、
どうやってそのファイルを探すかを処理するコード

アセットトリゾルバーの利用例



USD は高速・スケーラブル・ロバスト

- ”ペイロード” の概念により、ロードしたいものだけをロードできる
- バイナリ形式 (crate ファイルフォーマット) はマルチコア環境での読み込みにも最適化
- Pcp によるロバストな超高速コンポジション (後述)
- ネットワークストレージでの運用実績
- 数百万オブジェクト、数千ファイルを扱える

USD のビルドは簡単になった

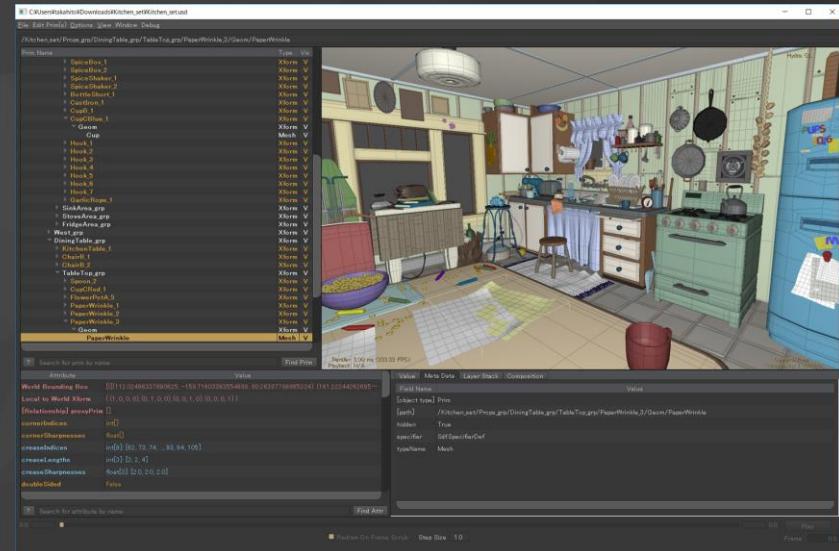
- 去年はビルドの説明だけで CEDEC 60分セッションできそうだった
- 今はビルドスクリプトがある
 - `python build_scripts/build_usd.py` だけ
 - TBB, Boost, OpenImageIO, OpenEXR, Ptex, OpenSubdiv, Alembic などは勝手にダウンロードしてコンパイルしてくれる
 - `python(x64)`, `cmake`, `nasm`, コンパイラだけ入れておく
- OSX, Windows 版も動く（まだ 100% ではないので注意）
- 単一 DLLやスタティックライブラリ版も構成可能、組み込みもしやすく

usdview は非常に便利なツール

- 全セクションで毎日使う
 - データのデバッグに最適
 - 高速なレンダリングエンジン
 - 各スタジオから絶賛

”史上最速の Alembic ビューワーだ”
“it works faster than any maya viewport.”

- Playblast (ビューポートレコーディング) の代わりに使っている人も多い
<https://github.com/LumaPictures/USD/blob/tg/luma/luma/pxr/usdImaging/bin/usdrender/usdrender.py>

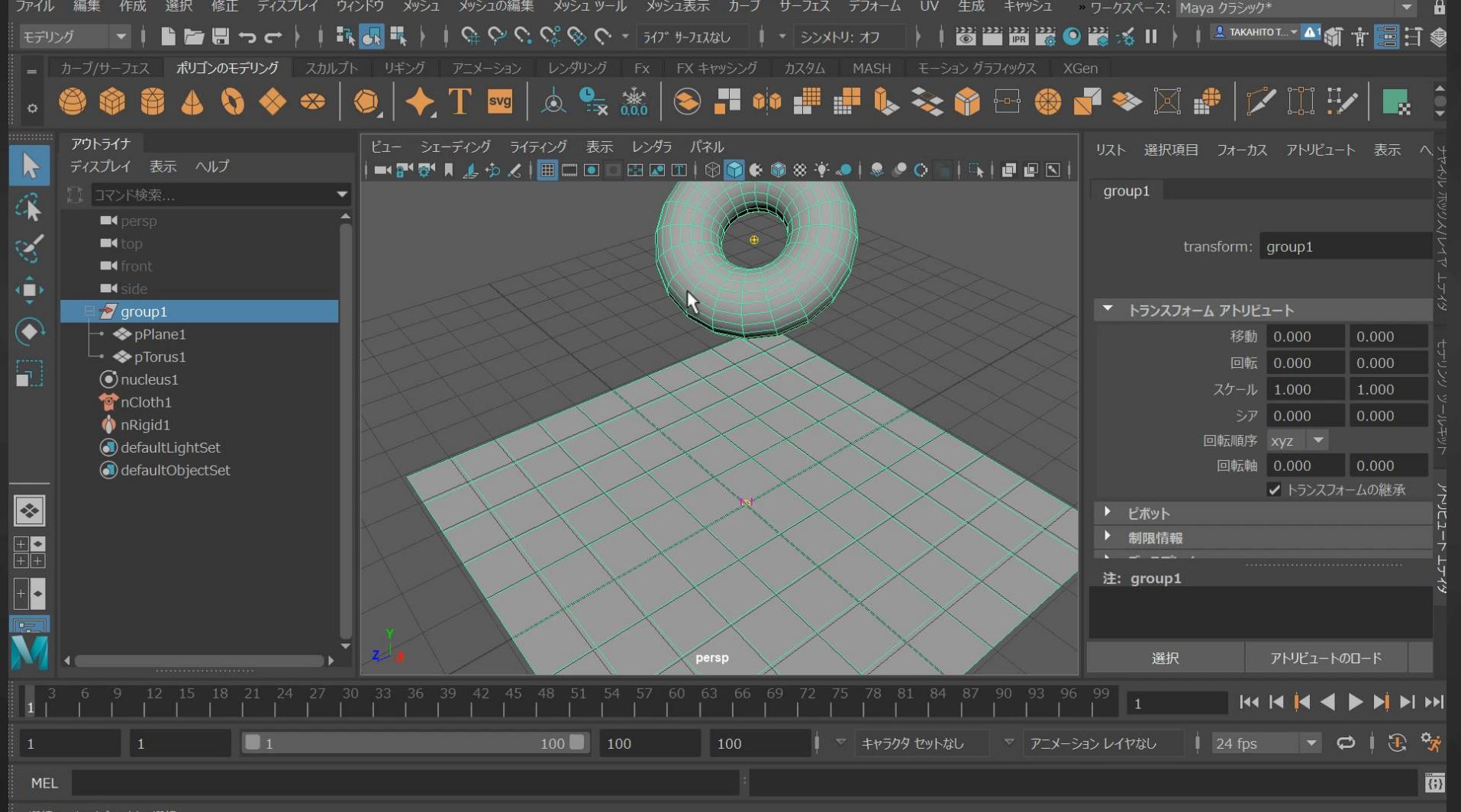


USD ファイルの作り方 (2017.8 現在)

- Maya プラグインなどでエキスポート
- python API でシーン構築、出力
- C++ API でシーン構築、出力
- ASCII ファイルをテキストエディタで書く

USD ファイルの使い方 (2017.8 現在)

- 今できること
 - Katana に読み込み、RenderMan や Arnold でレンダリング
 - Maya などに読み込んで、別の作業の素材にする
 - USDファイルをシーンアセンブリとしてカプセル化して読める
 - usdview に読み込んで、ストリーミング再生、データデバッグ
 - ゲームエンジンに読み込んで、ストリーミング再生
- 今後やりそなこと
 - ゲームアセットの中間フォーマットとして利用
 - 将来的にも再利用可能なアセットライブラリを作る
 - サブディビジョンサーフェスにほぼ完全対応しているメリットを生かす



USD テクニカル入門

ここから難しくなります！

USD テクニカル入門

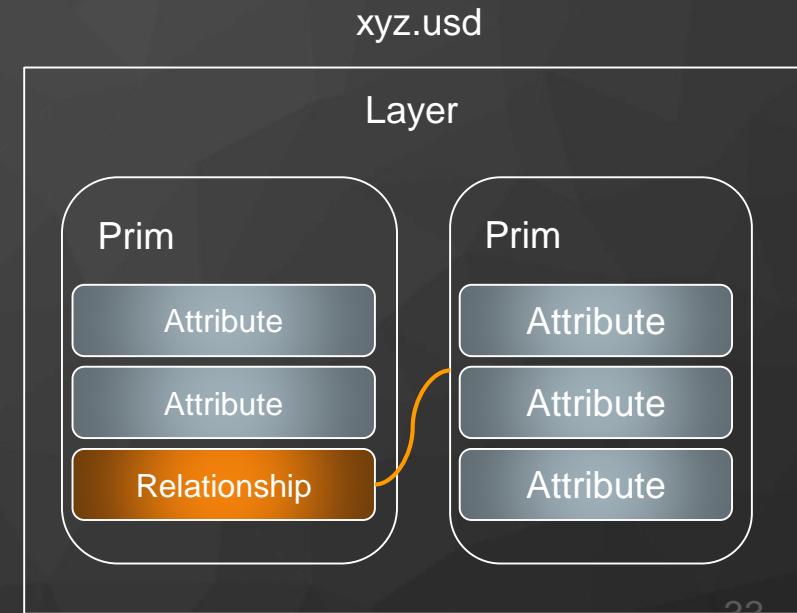
目次

- オブジェクトモデル（用語解説）
- コンポジションアーク
- スキーマ
- API 構成
- 今後の予定など

USD のオブジェクトモデル

USDのコアは 3DCG に限定されていない

- ごくシンプルなオブジェクトシステム
- **Prim** (プリム : 名前空間を構成する要素)
- **Property** (プロパティ)
 - **Attribute** (時間変化可能なデータ)
 - **Relationship** (ポインタ)
- 基本データ型
- メタデータ
- これらをまとめたものが **Layer**
(= 1 ファイル)



基本データ型

- int, float, float3, matrix4d など
- string = std::string
- TfToken = 辞書化された文字列

bool	bool	
uchar	uint8_t	8 bit unsigned integer
int	int32_t	32 bit signed integer
uint	uint32_t	32 bit unsigned integer
int64	int64_t	64 bit signed integer
uint64	uint64_t	64 bit unsigned integer
half	half	16 bit floating point (imath half)
float	float	32 bit floating point
double	double	64 bit floating point
string	std::string	stl string
token	TfToken	interned string with fast comparison and hashing
asset	SdfAssetPath	represents a resolvable path to another asset
matrix2d	GfMatrix2d	2x2 matrix of doubles
matrix3d	GfMatrix3d	3x3 matrix of doubles
matrix4d	GfMatrix4d	4x4 matrix of doubles
quatd	GfQuatd	double-precision quaternion
quatf	GfQuatf	single-precision quaternion
quath	GfQuath	half-precision quaternion
double2	GfVec2d	vector of 2 doubles
float2	GfVec2f	vector of 2 floats
half2	GfVec2h	vector of 2 half's
int2	GfVec2i	vector of 2 ints

用語：レイヤー (SdfLayer)

- 最も基本的なシーン定義
 - PrimSpec を 0 個以上含む
- 1 レイヤー = 1 ファイル
 - .usda (ASCII)
 - .usd (Binary : crate format)
- PrimSpec とは
 - **Prim** 自体はコンポジションの結果生まれる
 - レイヤーに記述されているのはあくまでこのレイヤーの ”意見”
 - PrimSpec = 「コンポジション前 Prim」とも言える
 - (注) 複数の PrimSpec から Prim が作られることもある

layer1.usd

PrimSpec “Cube”

PrimSpec “Sphere”

用語：名前空間 (namespace)

- レイヤーの中で、Prim, Property は階層化された名前空間に存在する
- レイヤーの最上位は Root Prim
- SdfPath を使って一意に見つける



用語 : SdfPath

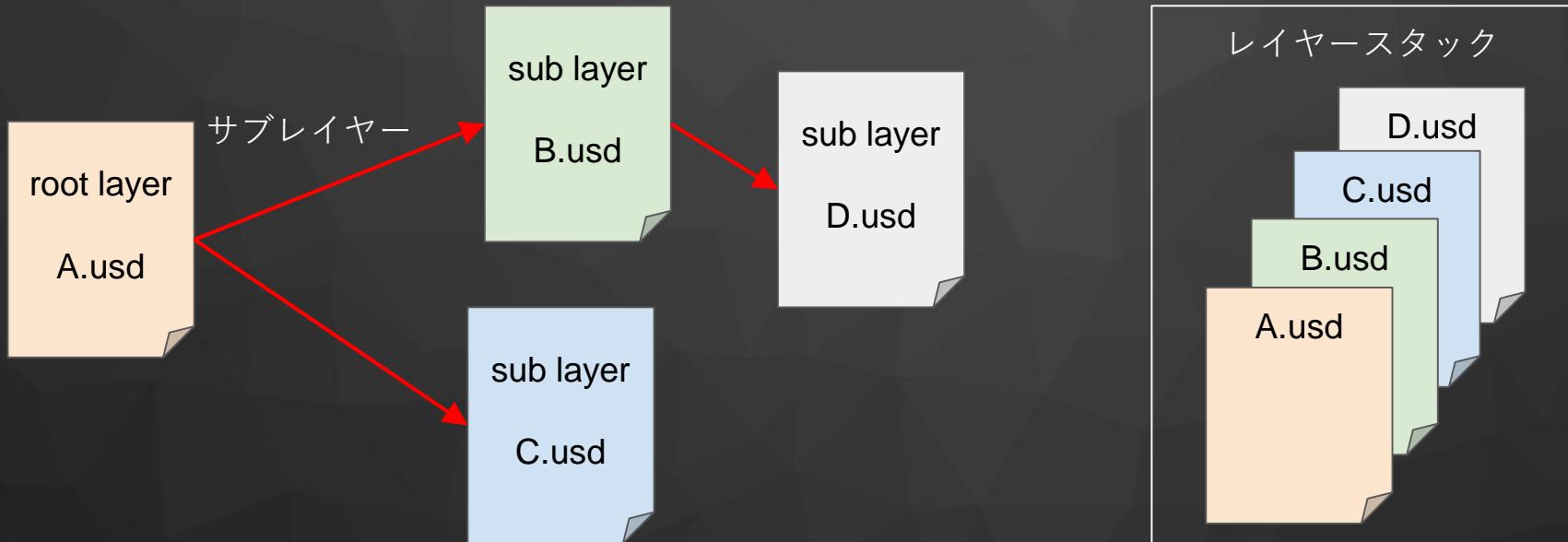
- シーディングラフ の名前空間ID
- XML の XPath のようなもの
- Prim だったり、Prim のプロパティだったり
- 例

/Foo	ルートの “Foo” を指す絶対パス
/Foo/Bar	Bar を指す絶対パス
/Foo/Bar.baz	Bar のプロパティ baz を指す絶対パス
Foo	現在の Prim の相対にある Foo
.bar	現在の Prim のプロパティ bar
/Foo.bar[/Foo].attrib	Foo.bar は /Foo を指すリレーションシップで、そのリレーションシップのアトリビュート attrib

```
def "Foo" {
    def "Bar" {
        Vec3f baz = (0, 1, 2)
    }
    rel bar = </Foo>
}
```

用語：レイヤースタック

- レイヤーは他のレイヤーをサブレイヤーとして再帰的に結合可能
- すべてのサブレイヤーを集めたものを「レイヤースタック」と呼ぶ



用語：リファレンス

- サブレイヤーとは別に、名前空間の特定の場所で他のレイヤー（usdファイル）を階層的に取り込むこと
- この結果もレイヤースタックになる



用語：ステージ (UsdStage)

- サブレイヤーやリファレンスによって、複数のレイヤー (=usd ファイル) のスタックを合成（コンポジション）して得られたビューが **UsdStage**
- UsdStage の PseudoRoot 以下の名前空間に結合される



合成の方法

- 単なる結合、リファレンスだけでは共同作業はできない
- コンカレントパイプラインを実現するには、ワークフローの分析から
 - 担当者 A はレイヤー A を所有し、自分の作業はそのレイヤー A 上で行う
 - 担当者 B はレイヤー A をなんらかの方法で自分の UsdStage に取り込み合成、自分の作業を別のレイヤー B に更新していく
 - 担当者 C はレイヤー A、レイヤー B を自分の UsdStage に取り込み合成、...
- 合成の方法は、パイプラインの設計次第
- USD ではこの合成方法を「**コンポジションアーク**」と呼ぶ

コンポジションアーク（演算子）

- サブレイヤー
- 繙承
- バリアントセット
- リファレンス
- ペイロード
- 特殊化

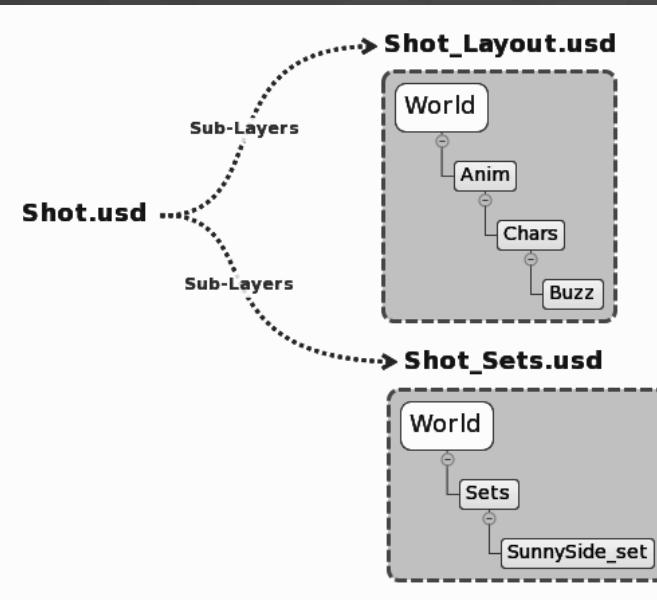
これらは
「共同作業のための演算子」

算術演算子同様、優先順位が厳密に定義されている
→ ロバストな合成結果を生む

1.サブレイヤー

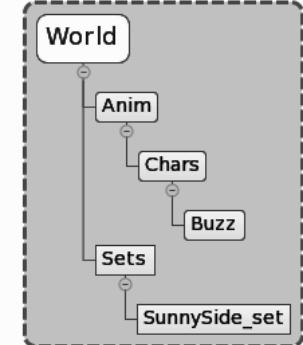
- レイヤーをまるごと合成
- C 言語で言えばリンク

```
Shot.usd  
  
#usda 1.0  
(  
    subLayers = [  
        @Shot_Layout.usd@,  
        @Shot_Sets.usd@  
    ]  
)
```



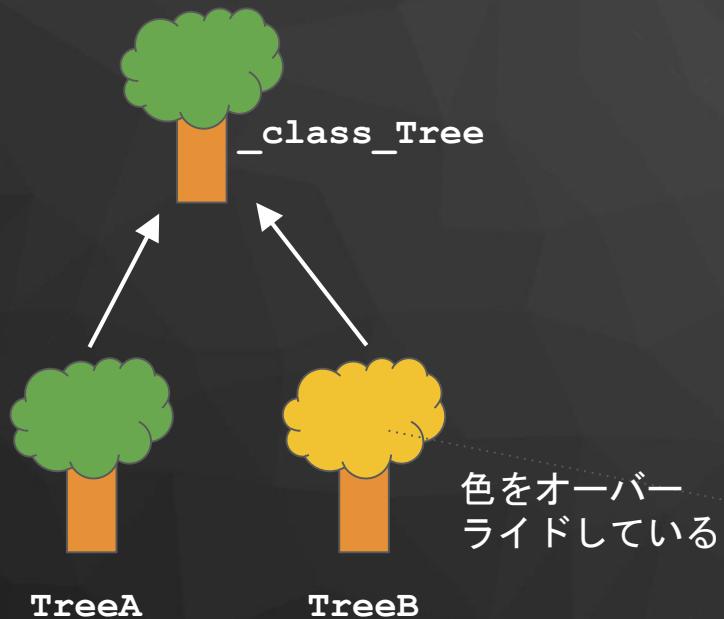
※ Layout と Sets で
/World が重複定義されている。
名前に一貫性があれば適切にマージされる

Composed Scene



2. 繙承

- C++ などの Class に類似



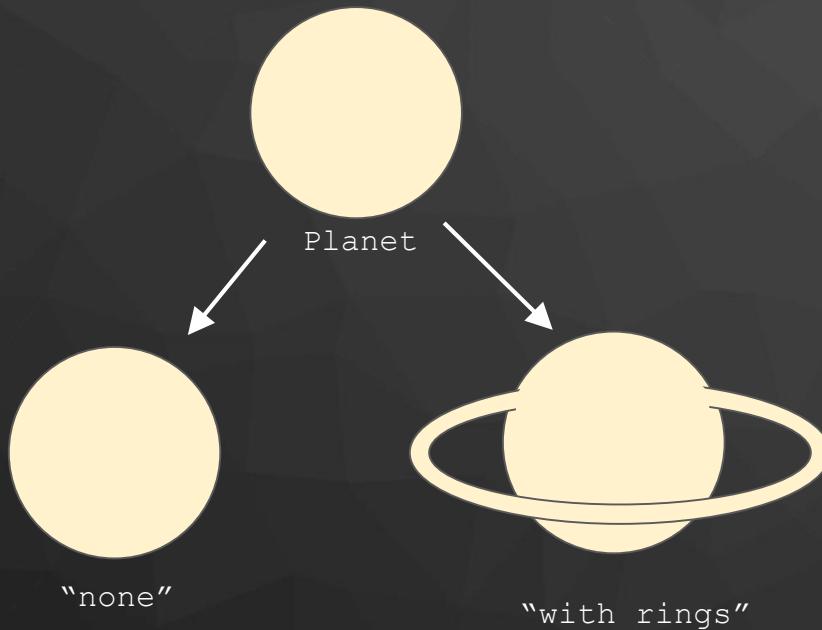
```
class Xform "_class_Tree"
{
    def Mesh "Trunk"
    {
        color3f[] primvars:displayColor = [ (.8, .8, .2) ]
    }
    def Mesh "Leaves"
    {
        color3f[] primvars:displayColor = [ (0, 1, 0) ]
    }
}

def "TreeA" ( inherits = [ </_class_Tree> ] )
{
}

def "TreeB" ( inherits = [ </_class_Tree> ] )
{
    over "Leaves"
    {
        color3f[] primvars:displayColor = [ (0.8, 1, 0) ]
    }
}
```

3. バリアントセット

- 種々のバリエーションを選択する switch-case



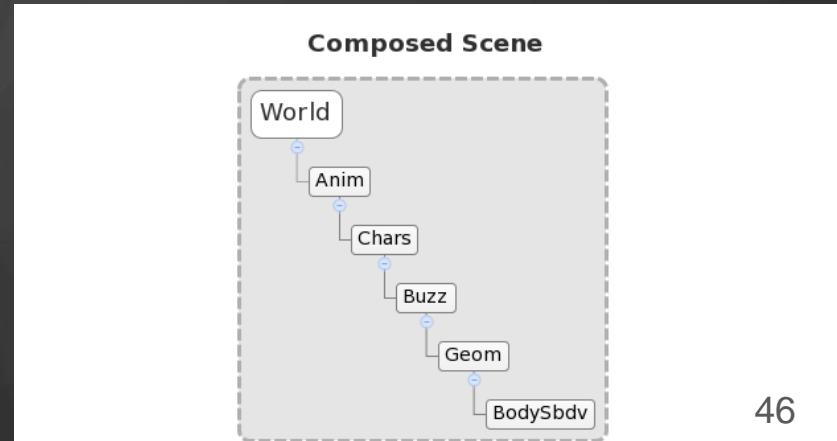
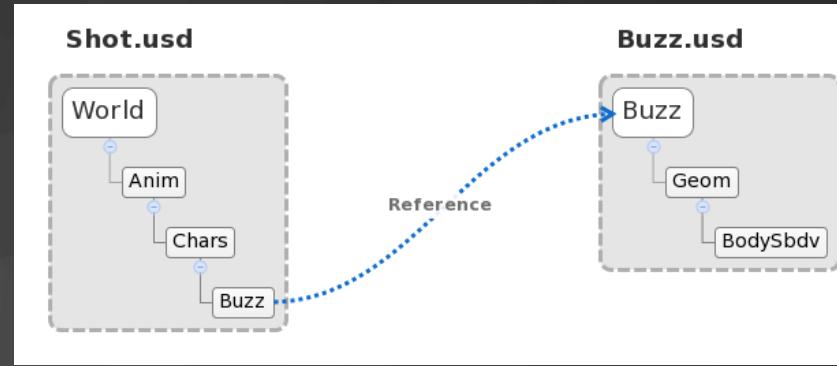
```
def Sphere "Planet" (
    variantSets = "rings"
    variants = { string rings = "none" }
)
{
    variantSet "rings" = {
        "none" {}
        "with_rings" {
            def Mesh "Ring" {
            }
        }
    }
}
```

4. リファレンス

- #include のようなもの
- ネストできる

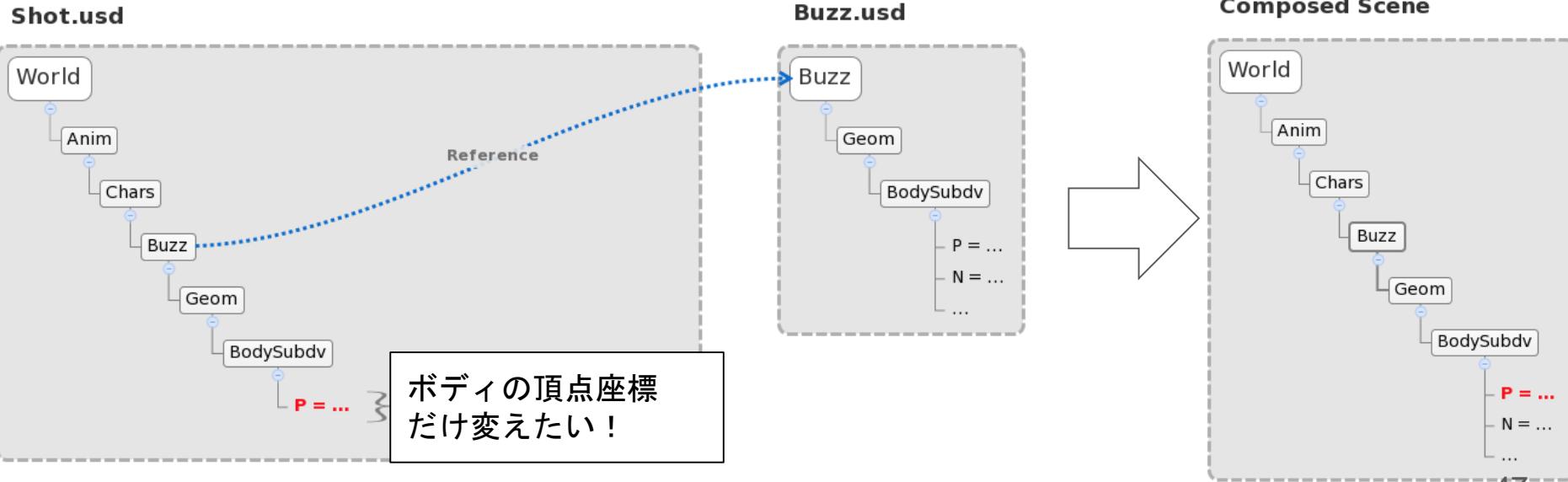
```
def "World" {  
    def "Anim" {  
        def "Chars" {  
            def "Buzz" ( references = @Buzz.usd@ )  
            {}  
        }  
    }  
}
```

```
def "Buzz" {  
    def "Geom" {  
        def Mesh "BodySubdiv"  
    }  
}
```



4. リファレンス（オーバーライド）

リファレンス時に、特定の要素の値を上書きできる
= **非破壊編集**



5. ペイロード

- ・ 巨大なシーンを分割可能なロード単位にわける
- ・ サブツリーを後から構築、load/unload が可能
 - ・ 例：巨大なシーンファイルを全部ロードする時間をかけずに、ライティングやレイアウトの変更だけをしたいときなど

```
def "World" {  
    def "Asia" (payload = @asia.usda@) ->  
    {<empty>}  
    def "Europe" (payload = @europe.usda@)  
    ..  
}
```

```
def Scope "Asia" {  
    def Xform "Japan" {  
        def Xform "Tokyo" {  
            ..  
        }  
    }  
}
```

6. 特殊化

- 繙承と似ているが、リファレンス側でオーバライドできない
- マテリアルライブラリなどに有効

```
def Sphere "BallA" {
    def "Material" (
        references = [@mat.usda@</Normal>]
    ) {
        over color3f color = (1, 0, 0) # より強い意見、赤になる
    }
}
def Sphere "BallB" {
    def "Material" (
        references = [@mat.usda@</Legendary>]
    ) {
        over color3f color = (1, 0, 0) # 弱い意見、紫が勝つ
    }
}
```

```
class "Material"
{
    color3f color = (1, 1, 1)
}
def "Normal" ( inherits = [ </Material> ] )
{
    color3f color = (0, 0, 1) # 青にはなるが・・・
}
def "Legendary" ( specializes = [ </Material> ] )
{
    color3f color = (1, 0, 1) # 紫
}
```

レイヤー同士の”意見”の調停

- コンポジションは「**名前空間の合成**」と「**意見の合成**」の2要素
- レイヤーに記述されている PrimSpec は、そのレイヤーが
「この Prim はこうあって欲しい」という意見を記述していた
- オーバラップやコンフリクトする



”球 A は緑”

”球 A は赤い”



何色？

コンポジション優先順位：LIVRPS 原則

SdfPath で指定された、あるデータの値を取得したい場合：

- 定義がなければ  **Local**（ローカル）
レイヤースタックをたどり、その SdfPath の値があれば（「意見があれば」）それを返す。
- 定義がなければ  **Inherits**（継承）
その SdfPath になりうる継承関係を解決し、再帰的に **LIVRP** を適用する（Sはない）。
- 定義がなければ  **VariantSets**（バリエントセット）
指定されたバリエントを選択し、再帰的に **LIVRP** を適用する。
- 定義がなければ  **References**（リファレンス）
その SdfPath になりうるリファレンスを解決し、再帰的に **LIVRP** を適用する。
- 定義がなければ  **Payloads**（ペイロード）
その SdfPath がペイロードでかつロード済みであれば、リファレンスと同様の処理を行う。
- 定義がなければ  **Specializes**（特殊化）
その SdfPath になりうる特殊化関係を解決し、再帰的に **LIVRPS** を適用する（Sを含む）。
その値は記述されていない。

例

モデリング

```
head_geom.usd  
"Head" をモデリング  
  
def Sphere "Head" {  
    float radius = 1.0  
}
```

シェーディング

```
head_shade.usd  
"Head" をシェーディング  
バリエントセットで色違いを作る  
  
over "Head" {  
    variantSet "head_colors" = {  
        "red" { color3f displayColor = [(1,0,0)] }  
        "green" { color3f displayColor = [(0,1,0)] }  
    }  
}
```

セットアップ

```
head.usd  
geom と shade を合成  
赤のバリエントを選択  
  
( subLayers = [  
    @head_geom.usd@,  
    @head_shade.usd@  
])  
over "Head" (  
    variants = { string color = "red" }  
) {}
```

リギング

```
hero.usd  
キャラクターを組み立てる
```

```
def "Hero" {  
    ...  
    def "Head" (  
        references = @head.usd@</Head>  
    ) {}  
    def "Body" ....  
}
```

レイアウト

```
shot.usd  
ショットに取り込むが、すぐロードしない  
  
def "World" {  
    def "Character" {  
        def "Hero" (payload = @hero_anim.usd@)  
        ...  
    }  
}
```

アニメーション

```
hero_anim.usd  
アニメーションを付ける
```

```
def "Hero" ( references = @hero.usd@) {  
    ...  
    over "Head" {  
        double3 xfromOp:translate = ....  
    }  
}
```

もちろんこれに限らず、多種多様なパイプラインを構築可能

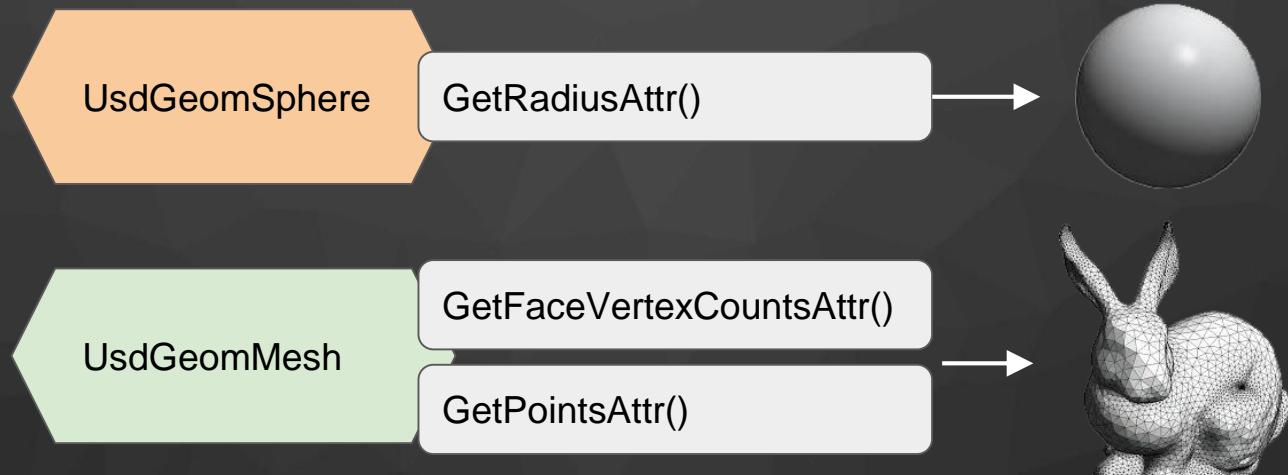
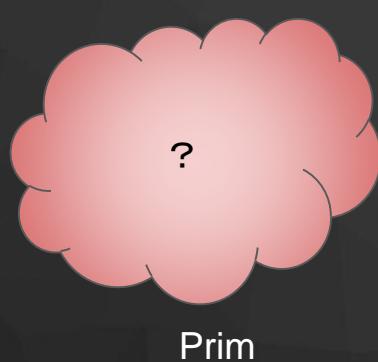
スキーマ

汎用オブジェクトシステムである USD を、CG の世界で扱うための約束

Concrete IsA スキーマ	実体化可能 <code>Define()</code> がある 具体的な Prim インタフェース	<code>UsdGeomMesh</code> , <code>UsdGeomSphere</code> など
Abstract IsA スキーマ	実体化不可能 <code>Define()</code> がない 共通インターフェース	<code>UsdGeomXformable</code> , <code>UsdGeomBoundable</code> など
API スキーマ	USD の型階層に入らない、ユーティリティ的なもの	<code>CollectionAPI</code> など

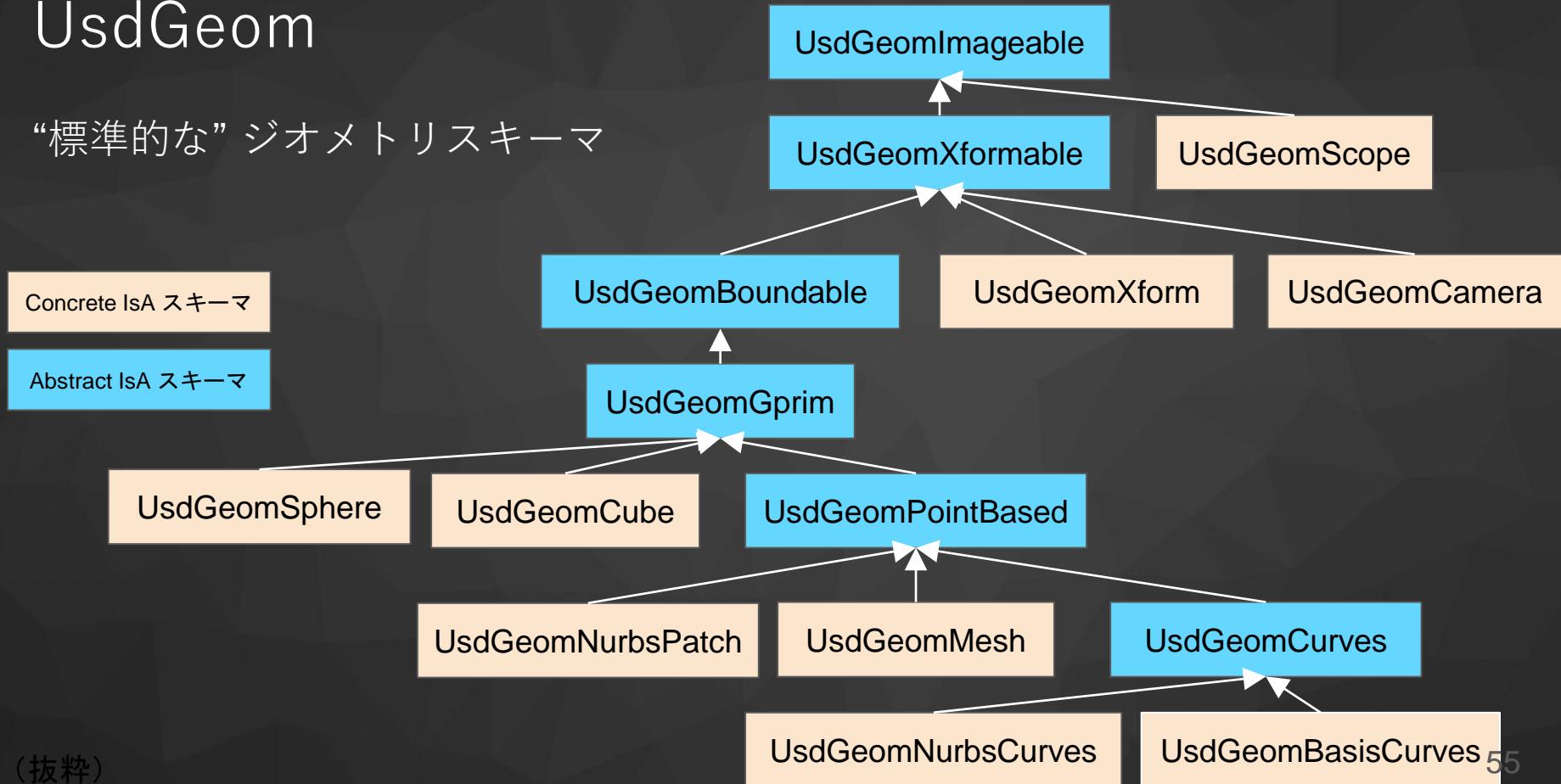
スキーマを使う

- スキーマクラスは API を提供する
- Maya の MFn… クラスに類似



UsdGeom

“標準的な” ジオメトリスキーマ



メッシュのスキーマ階層

UsdGeomMesh

UsdGeomImageable	レンダリングに参加するジオメトリ	visibility, purpose, proxyPrim
UsdGeomXformable	アフィン変換	xformOpOrder
UsdGeomBoundable	バウンディング	extent
UsdGeomGprim	グラフィックスプリミティブ	displayColor, displayOpacity, doubleSided, orientation
UsdGeomPointBased	“頂点”を持つジオメトリ	points, normals velocities (モーションブラー用)
UsdGeomMesh	サブディビジョンサーフェス (ポリゴンメッシュ)	faceVertexIndices, faceVertexCounts subdiv tags

トランスフォーム

UsdGeomXformable

```
class "Xformable" (inherits = </Imageable> )
{
    uniform token[] xformOpOrder
}
class "Xform" (inherits = </Xformable> )
{
    float3[] extent
}
```

- SRTなどの演算順を Prim ごとに指定でき、あらゆる変換ポリシー、表現方法に対応可能
- 逆行列演算子あり（ピボット用）

例

```
def Xform "MyCar"
{
    uniform token[] xformOpOrder = ["xformOp:translate", "xformOp:rotateXYZ"]
    ...
}
```

xformOps:

translate	-	3D
scale	-	3D
rotateX	-	3D
rotateY	-	3D
rotateZ	-	3D
rotateABC	-	3D (ex. rotateXYZ)
orient	-	4D (quaternion)
transform	-	4x4D matrix

Primvar (Primitive Variables)

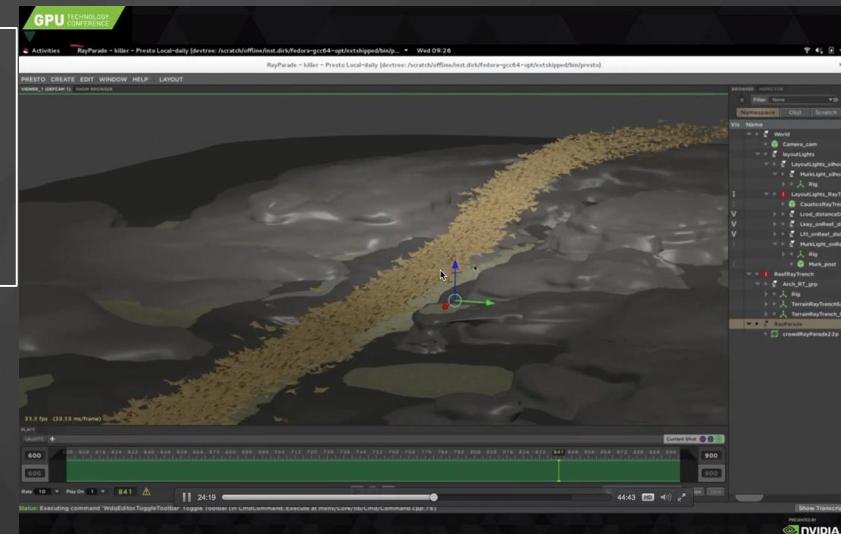
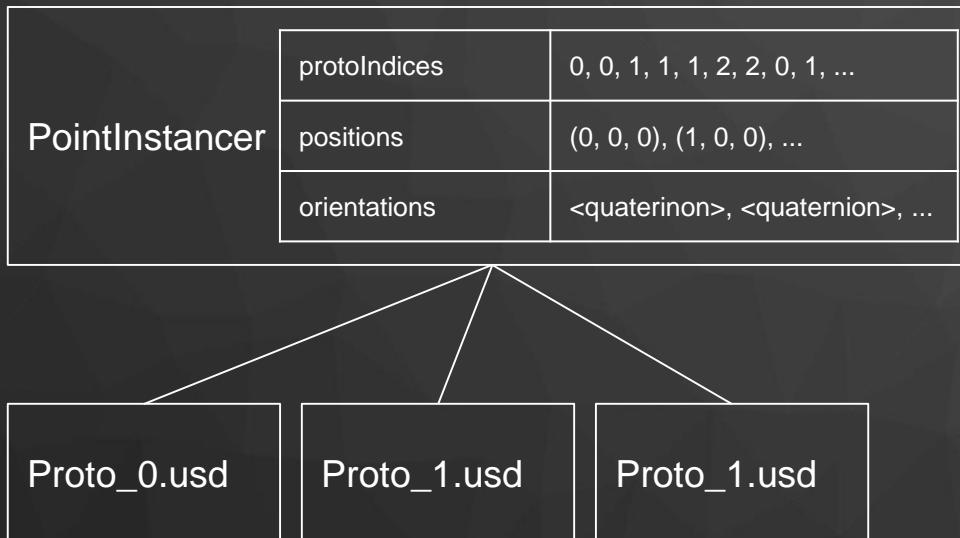
- `UsdGeomPrimvar` でアクセスする
- UV や頂点カラーなどはすべて Primvar として扱われる
- Primvar は、様々な補間モードを持つ汎用的なデータ
 - サブディビジョンサーフェスでも Well-Defined

<code>constant</code>	オブジェクト内で单一の値
<code>uniform</code>	フェイスやパッチごとに单一の値(GLSL などと違うので注意)
<code>varying</code>	線形補間される頂点ごとの値
<code>vertex</code>	サーフェス基底関数で補間される頂点ごとの値
<code>faceVarying</code>	フェイス頂点ごとの値(UV など)

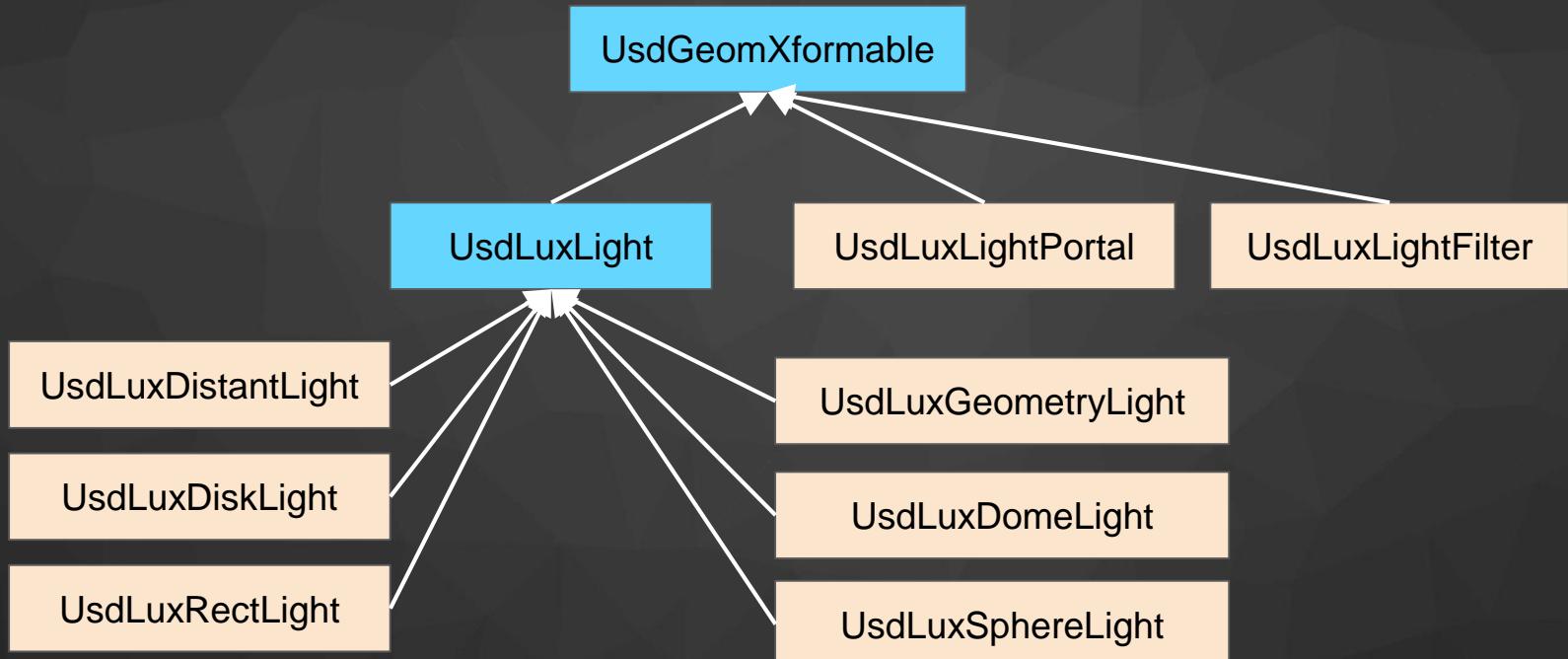
- RenderMan, OpenSubdiv の定義と一致

UsdGeomPointInstancer

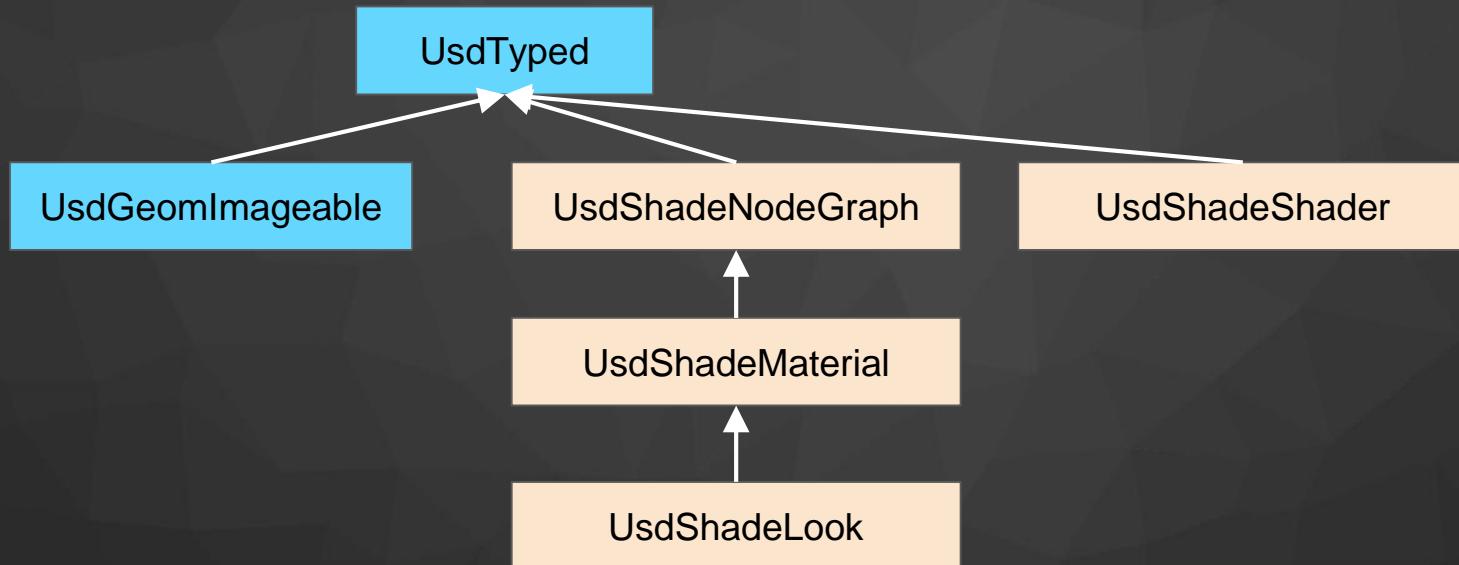
100万単位の群集モデルを記述するためのスキーマ



ライト : UsdLux



シェーディング : UsdShade



スキニング : UsdSkel



レンダリング：UsdRi, UsdHydra

- レンダラーが解釈するデータを記述したスキーマ
- 詳細は usdRi, usdHydra 内の schema.usda を参照
- いずれ Alembic Preview Material
<https://github.com/alembic/alembic/wiki/Alembic-Preview-Material-Specification>
などがスキーマで対応されると、ゲームでの使い勝手はぐっと向上しそう

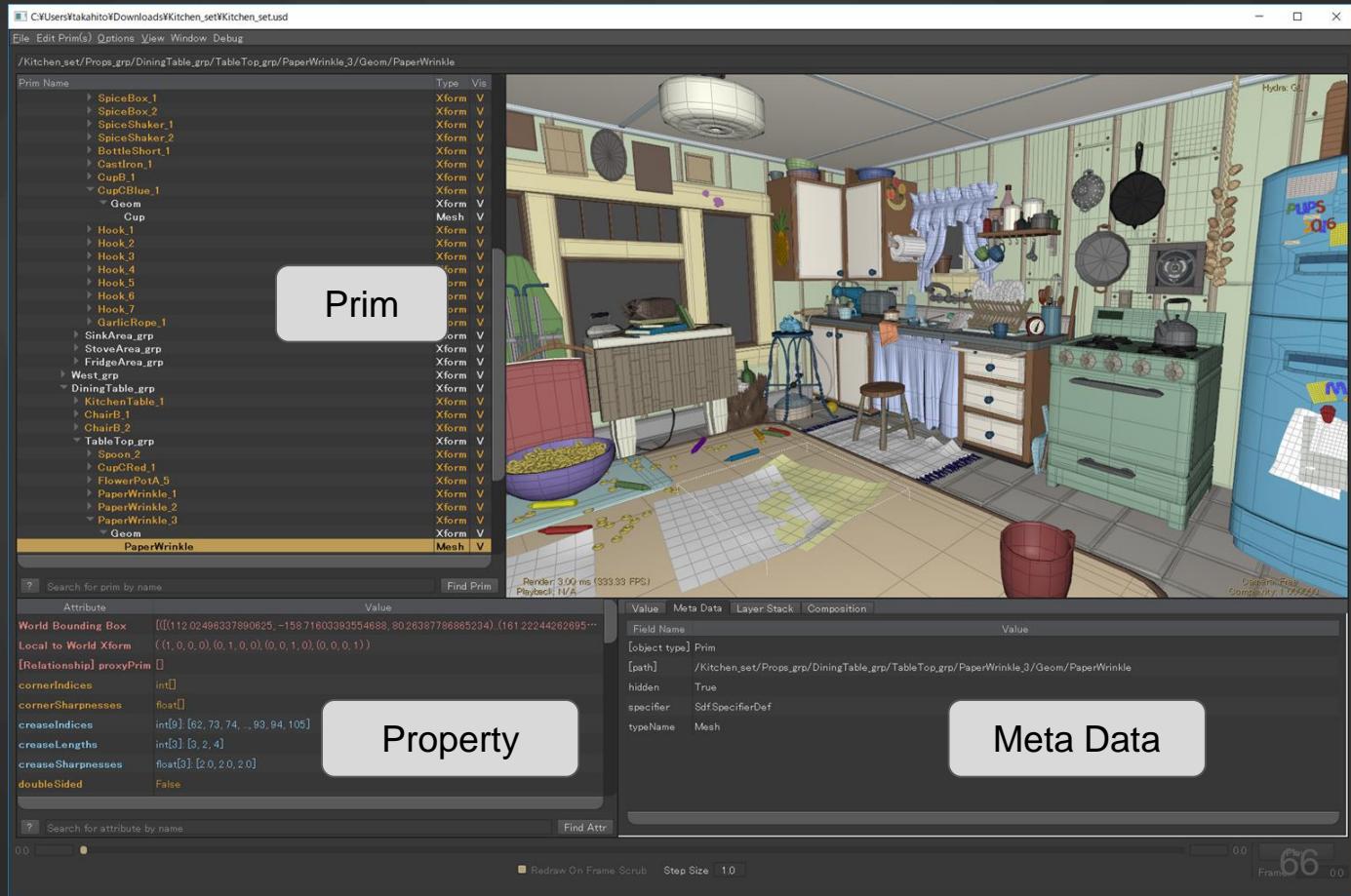
主要な API 構成

pxr/usd	UsdGeom, UsdRi, ...	標準スキーマ
	Usd	USD API (クライアントアプリケーション用)
	Pcp	コンポジションセマンティクス
	Sdf	シーン記述レイヤ
pxr/base	Tf, Gf, Arch, Vt, Work, ...	基本ライブラリ (メモリ管理、グラフィックス、機種依存等)
pxr/usdImaging	UsdImaging	USD 描画アダプタ
pxr/imaging	Hd, HdStream, HdEmbree	標準レンダリングエンジン Hydra

Pcp の役割

- Pcp はコンポジションセマンティクスを実装 (LIVRPS を解決)
- 必要なだけの途中結果をキャッシュする
- list-editing
 - 値の非破壊編集だけでなく、名前空間内の定義順の編集も行う
- 変更トラッキング
- 依存トラッキング
- USD の中核だが、直接使う必要はほぼない

usdview



USD でできないこと

- リギング、デフォーメーションの実行
 - A と B を計算して C にする、というような処理はアプリケーションが行う
 - ただしコンポジションは例外
- リグ用のスキーマを自分で足すことはできる
 - API クラスは自由な計算処理を実行できる
 - UsdGeomXformable の例
- ジョイントスキーニングは追加された。UsdSkel
 - FBX のスーパーセットになった

今後解決しなければいけないこと

- Python, Boost に依存している
 - Python 依存はいずれオプション化される予定
- メモリ確保の頻度が多く、 malloc の性能にも大きく依存する (特に Windows 環境)
 - 代替の malloc を推奨 <http://jemalloc.net/>
 - @i-saint さんが usd-windows の最適化にとても詳しいので聞いてみよう
- Linux 以外の動作確認があまりされていない
- ドキュメント、ユーザ事例がまだ乏しい

USD の開発フォーカス (~ 2018)

- MaterialX スキーマ UsdMatX が近々リリース
- 各種スキーマを充実、洗練させる
- インスタンスツールセットの強化
- Sub-root リファレンスを検討中
- パフォーマンスのさらなる向上



<http://www.materialx.org/>

まとめ

- USD は急速に映像業界の支持を集めている
 - 各種DCCベンダ、VFXスタジオ、ゲームエンジン、レンダラーが続々対応中
- USD はすでに十分な実績がある
 - 長編映画が作れるスケーラビリティ・パフォーマンス
- プロシージャルなアセット構造を作るための、宣言型言語
 - 非破壊編集は非常に強力
- 単なるシンググラフというより、
CGデータの**共同作業のための言語と処理系**
あるいは
パイプラインをプログラミングする環境
だと思ってほしい

パイプラインの未来

- ・ 「パイプラインにもデザインパターンがある」 (Bill Polson)
- ・ 欧米のゲームスタジオは USD に興味を示しているが、具体的な動きはまだ。
- ・ 今がチャンス、**一緒に次世代パイプラインをデザインしましょう！**



ポリフォニー・デジタルのセッション

8/30 (水) 17:50 - 18:50 R315

HDR 理論と実践

内村 創

8/31 (木) 10:00 - 11:00 R502

Position Based Dynamics Combo 物理シミュレーション関連の最新論文実装

中川展男

8/31 (木) 10:00 - 11:00 R302

Maya + Python でインハウスツールを作ろう！

富田岳伸

8/31 (木) 16:30 - 17:30 R302

Houdini16にHDAを2つ追加したらビルが量産できた(プロシージャルモデリング事始め)

齋藤 彰

參考資料（1／2）

Pixar Graphics Technologies

<http://graphics.pixar.com>

USD GitHub repository

<https://github.com/PixarAnimationStudios/USD>

GTC 2016 Talk : Real-Time Graphics for Feature Film Production

<http://on-demand.gputechconf.com/gtc/2016/video/S6454.html>

GTC 2017 Talk : Advances in Real-Time Graphics at Pixar

http://on-demand.gputechconf.com/gtc/2017/video/s7482-david-yu-advances-in-real-time-graphics-at-pixar_POLJEREMIASVILA.mp4

AL_usdMaya

https://github.com/AnimalLogic/AL_USDMaya

usdQt

<https://github.com/LumaPictures/usd-qt>

参考資料（2／2）

usd-arnold

<https://github.com/LumaPictures/usd-arnold>

USDforUnity

<https://github.com/unity3d-jp/USDForUnity>

サンプルアセット

<http://graphics.pixar.com/usd/downloads.html>