KLab Tech Book

Vol.03

- プロシージャルモデリングを支えるHoudiniの機能紹介
- 2.5万円で買える3Dプリンタのススメ
- Airtestを用いたUnityアプリの自動実機テスト
- Rider+UnityでRoslyn Analyzersを使う
- バーコードリーダーになろう
- Unity Timeline Tips集
- 物理ベースレンダラーをRust実装して、ちょっと高速化した話
- ヘッドレスChromeでリボ払いを回避している話

KLab Tech Book Vol. 3

2018-10-08 版　　**KLab** 技術書サークル 発行

はじめに

このたびは、本書をお手に取っていただきありがとうございます。本書は、KLab 株式会社の有志にて作成された KLab Tech Book の第 3 弾です。

KLab 株式会社では、主にスマートフォン向けのゲームを開発していますが、本書ではこれまでどおり、社内のエンジニアの多岐にわたる興味を反映したさまざまな記事を収録しました。業務に関係のあることもあり、完全に趣味の内容もあります。

あまりにもバラバラな内容なので、統一感がまったくありませんが、ひとつ共通していえるのは、著者自身がその内容に興味を持ち、楽しさを感じて書いていることです。

その楽しさが読者のみなさまにも伝わればさいわいです。

於保 俊

お問い合わせ先

本書に関するお問い合わせは tech-book@support.klab.com まで。

免責事項

本書に記載された内容は、情報の提供のみを目的としています。したがって、本書を用いた開発、製作、運用は、必ずご自身の責任と判断によって行ってください。これらの情報による開発、製作、運用の結果について、著者はいかなる責任も負いません。

目次


KlabTech
vol.03

第 1 章

プロシージャルモデリングを支える Houdini の機能紹介

Kinuko MIZUSAWA

KLab でクライアントエンジニアをやっている人間です。普段はビルドのマシンやジョブの管理、OS ネイティブ周りの機能を担当しているのですが、最近 3D 関連の業務に関わる機会が出来、**Houdini**[*1]の機能について調べたり実際にソフトを触ったりしています。

▲図 1.1　Houdini の公式サイト上でのビジュアル。プロシージャルなコンテンツ製作ツールとして訴求されています

Houdini とは、SideFX 社が提供している 3DCG ソフトです。モデリング、アニメー

[*1] https://www.sidefx.com/ja/

ション、エフェクトの制作から、それらを組み合わせて合成するコンポジットまでと、3DCG 制作の全行程をカバーしているソフトになります[*2]。

Houdini は元々、プロシージャルにモデル製作ができ、効率的にハイクオリティなアセットを作ることができるツールと期待して学習を始めたのですが、Houdini を知れば知るほど、その強力かつ柔軟性のある機能に魅力を感じました。

このため、今回は Houdini と、Houdini のどういった機能、特徴がプロシージャルモデリングを可能としているかということを紹介したいと思います。

本章が、Houdini を利用したアセット制作を検討する際にお役に立てば幸いです。

1.1 本章で定義するプロシージャルモデリングとは

プロシージャルモデリングとは、昨今のゲームコンテンツのリッチ化、制作コストの高騰を受けて特に注目されている技術です。

プロシージャルモデリングでは数式やプログラムによって数値などの入力を元にモデルを作成する仕組みを作り、その仕組みによってモデルを作成することができます。

非プロシージャルなモデリングでは、量産にあたりモデリングを行うアーティストの手作業が発生し、たとえばキャラクターのシルエット、体型の変更を行った際にディテールの追加修正作業も同時に発生します。

一方で、プロシージャルなモデリングでは、たとえキャラクターのシルエット、体型の変更を行った場合でも、一度モデリング作業そのものを自動化する仕組みを作っていることから、体型に合わせてディテールの調整も行ってくれ、作業を不要とすることができます。

このように、プロシージャルモデリングによって、3D アセットを作る人は、ツールのオペレーションのような単純作業を減らし、デザインの検討のようなクリエイティブな作業に時間を割くことができるようになります。また、3D アセットの作成スキルを持っていない人でもパラメータの変更やネットワークのみでアセットの大きさや形を変えることができるようになります。

1.2 Houdini でプロシージャルモデリング

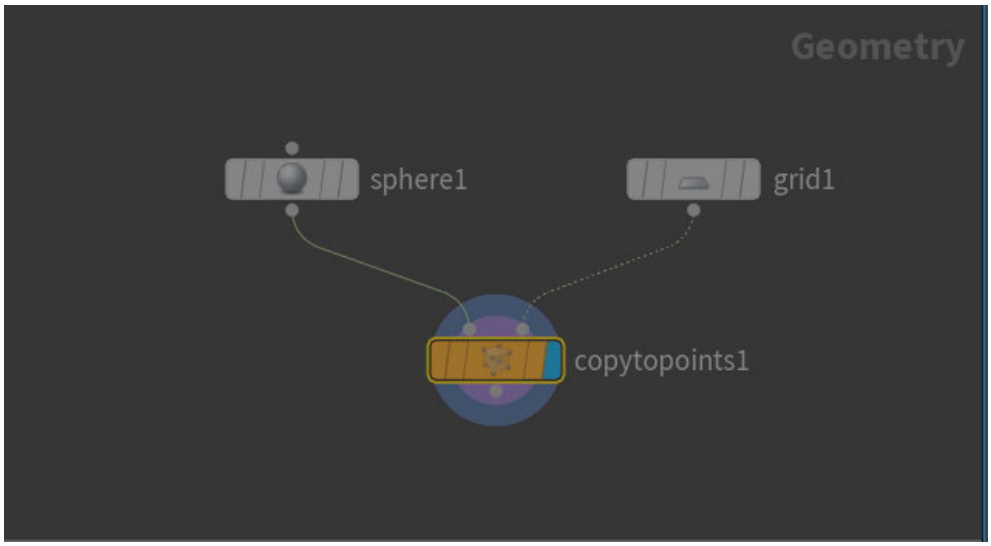
Houdini でプロシージャルモデリングすることを語る上で、語るべきは大きく次の三点です。これらについて触れていきたいと思います。

- ノードベースプログラミング
- Houdini Digital Assets
- Houdini Engine

[*2] 特にエフェクト、シミュレーションは機能が豊富かつ強力で、映画の VFX なら Houdini を使っている現場が増えているようです

ノードベースプログラミングによる手作業の自動化

そもそもノードベースプログラミングとはどういうものでしょうか？ これは、最近多くのゲームエンジンやツールで採用されている視覚的にプログラムを組めるものになります。



▲図 1.2　Houdini で、ノードを追加したりつないだりするネットワークエディタの画面

視覚的にプログラムを組めるものの例としては、Houdini 以外のゲーム開発のツールだと Unreal Engine の Blueprints[*3]や、Unity の Shader Graph[*4]などがあります。

これらは視覚的にプログラムを組めて、処理の流れが目でみてわかりやすいのでアーティストにとっても扱いやすくまた普段コードでプログラムしているプログラマにとっても管理がしやすかったり、ノードを繋いだりする作業自体が直感的で楽しいものとなっていることから人気です。

Houdini ではこのノードベースプログラミングで記述できることにより、アセット作成を自動化することができ、また、視覚的にも処理がわかりやすくなるメリットがあります。また、副次的な作用として、たとえば Photoshop や Illustlator にあるレイヤー機能のように、処理のひとつひとつに対してオンオフの切り替えができるようになっていることで手戻りが起きた時の手作業をなくしたり、減らしたりすることができます。

[*3] http://api.unrealengine.com/JPN/Engine/Blueprints/
[*4] https://unity3d.com/jp/shader-graph

Houdini で扱うことのできるノードとは

Houdini のノードにはその機能ごとにいくつかの種類があります。一度どういう種類があるのか、大まかに把握しておくと実際に作業で使うべきノードを探すときに助かりますので、ここではざっくりとしたノードの種類を紹介しつつ、特にモデリングで多用することになるノードタイプ、SOP の具体例について順に紹介していきたいと思います。

Houdini のノードの種類

Houdini のノードはすべて、次のいずれかのノードに該当します。

CHOP ノード

- 波形の作成、編集を行うノードです。アニメーションやオーディオのデータを編集したりします。

COP ノード

- 2D 画像の作成、編集、合成を行うコンポジットのノードです。レンダリング画像を合成したり、色の調整をしたりします。

DOP ノード

- ダイナミクス、シミュレーション処理を扱うノードです。煙や炎、水や布の揺れ、破壊の挙動を計算したりします。15 以前の Houdini のバージョンでは POP ノードとして扱われていたパーティクルもこの中に含まれます。

OBJ ノード

- シーンの最上位階層を表すノードです。ジオメトリ、ライト、カメラなどのタイプがあります。

ROP ノード

- 出力を行うコンポジットのノードです。レンダリングやキャッシュ作成などのデータ出力をしたりします。

VOP ノード

- Houdini 独自のプログラミング言語「VEX」を用いて書かれた処理を表すノードです。VOP ではジオメトリの操作、シェーダー作成などいろいろなことができます。15 以前の Houdini のバージョンでは SHOP ノードとして扱われていたマテリアルもこの中に含まれます。

SOP ノード

- 本章で紹介しているジオメトリの作成、編集を行うノードです。

モデリングで特に利用する SOP の具体例

ここでは、特にモデリングで多用することになる SOP のノードを具体的に紹介することで、Houdini でプロシージャルモデリングすると、どういうネットワークを組むことができるのかイメージできるようにしたいと思います。

SOP のノードには、たとえば次のようなものがあります。

■基本形状の生成

ノード名	説明
Curve	曲線を作成するノード
Line	直線を作成するノード
Sweep	スイープ曲面を作成するノード
Sphere	球体を作成するノード
Grid	平面を作成するノード

■ジオメトリの変更

ノード名	説明
Transform	ジオメトリのサイズを変更するノード
Bend	ジオメトリを曲げるノード
Merge	ジオメトリを統合するノード

■ジオメトリの追加

ノード名	説明
Copy Stamp	ジオメトリのコピーを作成し配置するノード
Copy To Point	ジオメトリをポリゴンのポイントに合わせてコピーして配置するノード
Scatter	ジオメトリを散乱させて配置するノード

これらの基本となるノードやよく使われるであろう便利なノードは Houdini の標準機能として用意されているので、Houdini ユーザーはそれらを組み合わせることでモデリングすることができます。

Houdini の標準機能として用意されているノードは、次のサイトにて公開されています。実際に Houdini でモデリングしていく際にはそちらの情報をまず見てみるのが良さそうです。

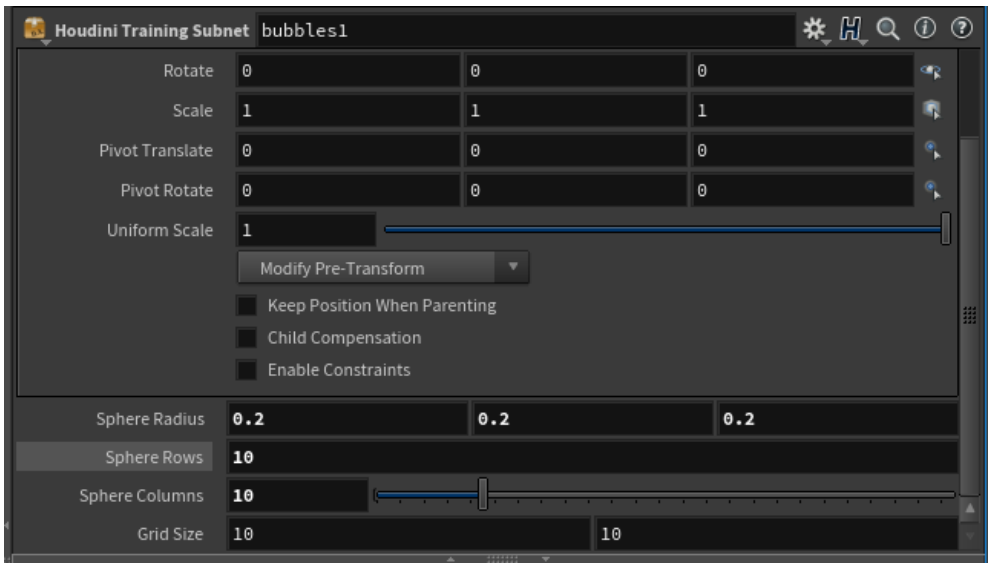
公式サイトリンク

上記のノードは、次の公式ページでも紹介されています。

https://www.sidefx.com/ja/docs/houdini/examples/nodes/index.html

デジタルアセットによるプログラム化されたモデリング作業の移植容易化

さて、さらに Houdini では、ノードベースプログラミングによって組んだネットワーク、ノードの仕組みはそのかたまりごとに Houdini Digital Assets という形で外部ファイル化し、ノードのパラメータを変更可能なものとすることができます。



▲図 1.3　Houdini で、ノードのパラメーターを編集するパラメータエディタの画面

デジタルアセットがあることで、アセットを作る作業そのものを部品化し、組み合わせることができるようになります。

Maya や Unity, Unreal Engine との連携を可能にする Houdini Engine

ここまで紹介して来たノードベースプログラミングで作ったネットワークは、デジタルアセット化してネットワークそのものを外部ファイルとして吐き出すことで、GUI 上で利用できるプラグインとして **Maya** のような DCC ツールや **Unity**, **Unreal Engine** といったゲームエンジンでも利用できるようにすることができます。

この利用により、Houdini の機能を利用するために Houdini と DCC ツール、ゲームエンジンといったツールの行き来をなくし、DCC ツールやゲームエンジン上で Houdini で作ったネットワークによってリアルタイムに最終的な見た目を確認しながらアセット制作などの作業を行うことができるようになります。

Houdini が連携できる外部ソフトは次のとおりです。

- ゲームエンジン
  - Unity
  - UE4
- DCC ツール
  - Maya
  - 3DS Max
  - Cinema 4D

この、Houdini と外部ツールを連携させるためのツールが Houdini Engine です。現在はノードロック版の Indie ライセンスであれば、3 ライセンスまで無料で使えるようになっています。また、フローティング版のライセンスも、サブスクリプションで 1 年/90 日/60 日/30 日/14 日/7 日といった期間で使えるようになっています。

サブスクリプションの期間	価格	備考
1 年	$ 795 USD	1 ライセンスから購入可能
90 日	$ 525 USD	2 ライセンスから購入可能
60 日	$ 350 USD	2 ライセンスから購入可能
30 日	$ 175 USD	5 ライセンスから購入可能
14 日	$ 150 USD	5 ライセンスから購入可能
7 日	$ 75 USD	10 ライセンスから購入可能

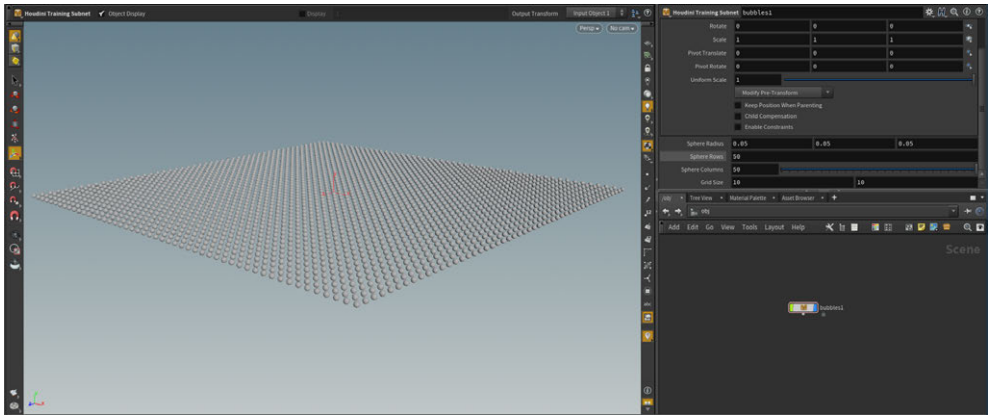
また、Houdini Engine は GUI 上で利用できるプラグインとしてだけではなくプログラムから呼び出せる API も提供されています。この API を利用して、深夜にアセットをビルドする、統合する、といったことも可能になります。

1.3 HDA を触ってみる：パラメータ変更のみでアセットの数やサイズを変更

それでは、ここで「グリッドのポイントに沿ってきれいに配置された球の群れ」「bubble」と名付けた HDA を触ってみたいと思います。

- シーンへの読み込み

ネットワークエディタ上で、右クリックあるいはタブボタンを押してノードの追加メニューから「Digital Assets」を選択して読み込むアセットを選択します。
選択すると、シーンエディタ上に球の群れが表示されます。

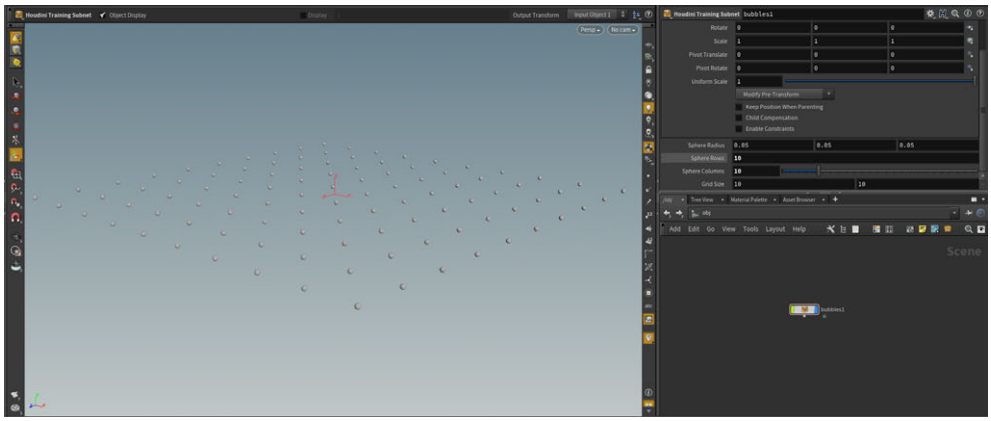


▲図 1.4　あらかじめ作成していた「bubbles」を Houdini 上で読み込み、シーン上に配置させた時の画面

- 球の数を変えてみる

読み込んだアセットのノードを選択し、パラメータのエディタで「Sphere Rows」「Sphere Columns」を 50 から 10 に変更してみます。

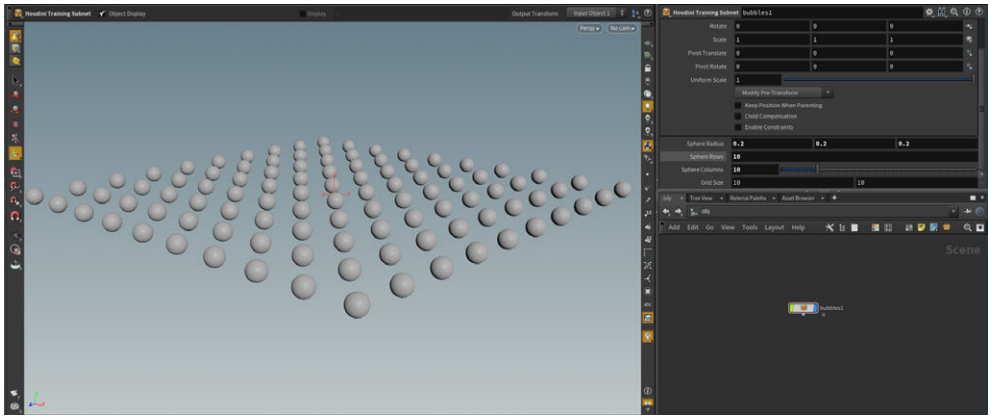
そうすると、シーンエディタ上で表示される球の数が減っていることが確認できます。



▲図 1.5　「bubbles」のパラメータ「Sphere Rows」「Sphere Columns」の設定値を下げ、シーン上に表示される球の数を減らしてみた時の画面

- 球のサイズを変えてみる

また、続いてパラメータのエディタで「Sphere Radius」というパラメータを 0.05 から 0.2 に変更してみます。



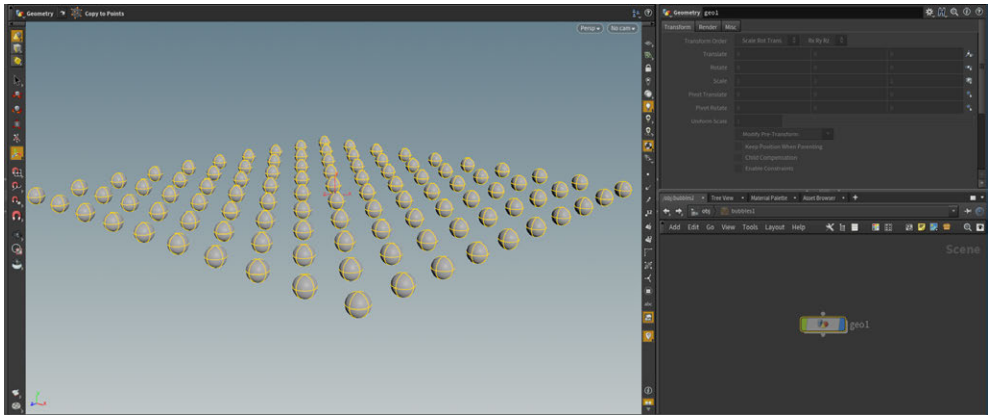
▲図 1.6　「bubbles」のパラメータ「Sphere Radius」の設定値を上げ、シーン上に表示される球のサイズを大きくしてみた時の画面

そうすると、シーンエディタ上で表示される球のサイズが大きくなっていることが確認できます。

1.4　HDA を触ってみる：ノードやパラメータを確認

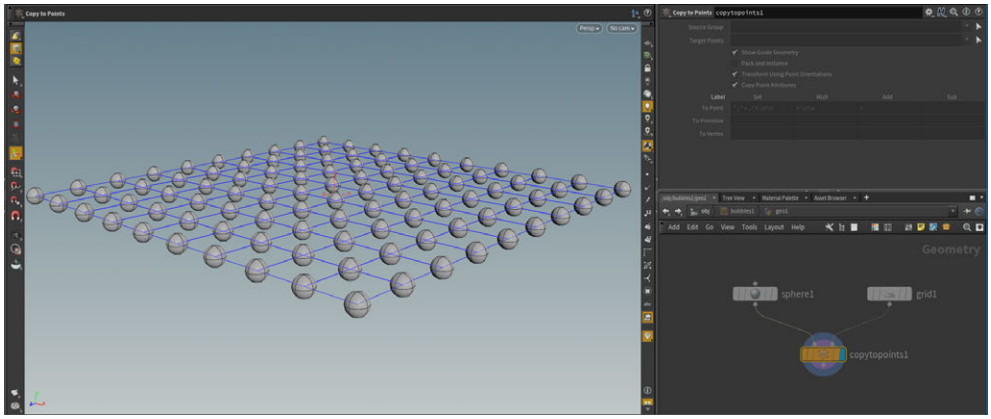
次に、ちょっとだけ HDA の設定を見てみようと思います。

- ネットワークエディタを開きます
- bubble1 という名前のノードがあることがわかります
- ダブルクリックすると geo1 という名前のノードがあることがわかります



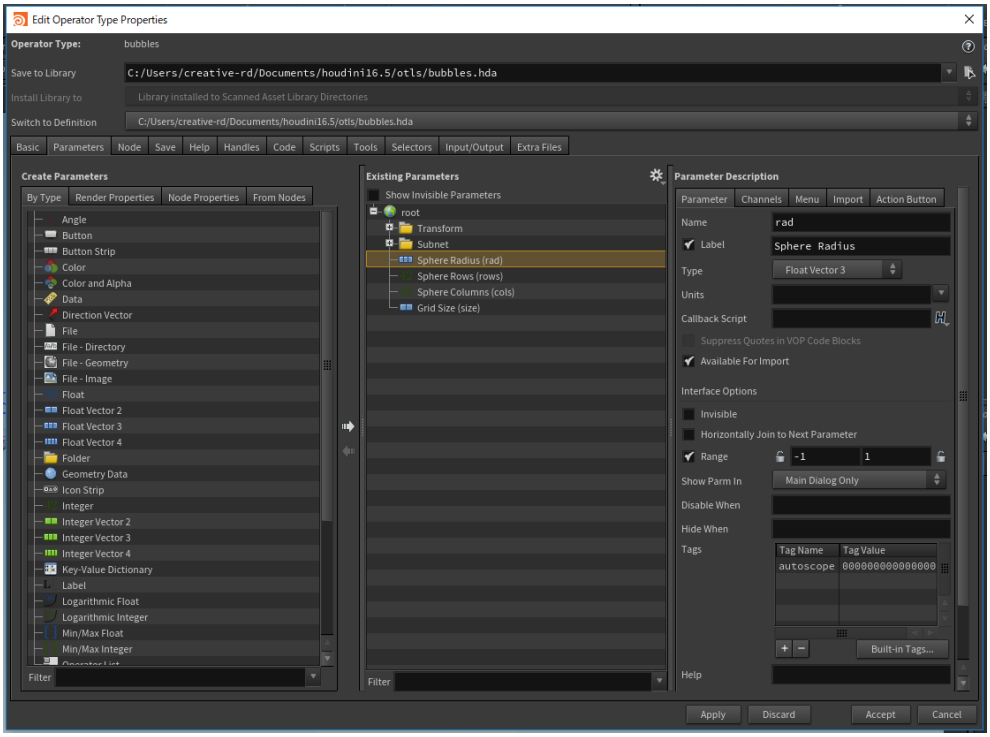
▲図 1.7　Houdini のネットワークエディタにて、ノードの階層を下りていった時の画面

- さらにダブルクリックすると sphere1, grid1 というノードと、copytopoints1 というノードが繋がっていることがわかります
  - sphere というノードは球を配置するノードです
  - grid というノードは平面を配置するノードです
  - copytopoints というノードはコピーするオブジェクトのノードと、コピー先のポイントのノードを入力として、オブジェクトをポイント上に配置するノードです。
    - 今回は、コピーオブジェクトを sphere, コピー先のポイントを grid にしています



▲図 1.8　さらにノードの階層を下り、「bubbles」の特徴をなしている「球の群れ」を表現しているネットワークを確認している画面

- Edit Digital Assets の画面で、Parameter の画面を開くと、bubble の Sphere Radius, Sphere Rows, Sphere Columns, Grid Size といったパラメータが Existing Parameters リスト上に配置されていることがわかります



▲図 1.9　HDA の編集画面を開き、外部から編集可能なパラメータの中に「Sphere Radius」「Sphere Columns」「Sphere Rows」「Grid Size」という名前が載っていることを確認している画面

なお、ここで紹介している HDA は、Houdini 公式サイトにある次のチュートリアル動画を参考にして作ったものです。

https://www.sidefx.com/ja/tutorials/asset-menu/?collection=27

1.5　おわりに

以上、この章では Houdini を利用したプロシージャルモデリングについてお伝えしてきました。

今年は CEDEC でも多くの Houdini のセッションがありましたが、CEDEC のセッションは非常に参考になるものでしたので、URL を記載しておきます。興味のある方はぜひご覧ください。

1.6 参考文献、資料

書籍

- 理論と実践で学ぶ Houdini -SOP&VEX 編-／ボーンデジタル
- Houdini ビジュアルエフェクトの教科書／エムディエヌコーポレーション

CEDEC 2014

- Houdini Engine と Houdini Indie によるプロシージャルコンテンツ作成 (http://cedil.cesa.or.jp/cedil_sessions/view/1153)

CEDEC 2017

- 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)(http://cedil.cesa.or.jp/cedil_sessions/view/1710)

CEDEC 2018

- Maya と Houdini を連携させて効率的な環境を構築してみよう！ (http://cedil.cesa.or.jp/cedil_sessions/view/1840)

第 2 章

2.5 万円で買える 3D プリンタのススメ

黒井 春人/ @halt

2.1 はじめに

この章ではプログラミングの話ではなく、3D プリンタについて紹介します。

KLab 内での私の職種はサーバーサイドエンジニアで PHP がメインなんですが、エンジニアの中途採用や新卒採用に関わる書類選考、面接、技術教育などの採用系を担当しています。[*1]

会社の業務で 3D プリンタを触っているわけではなくて、個人的な趣味で電子工作をやっていて、その延長で 3D プリンタを買ってみたらものすごく楽しかったので 3D プリンタ布教のためにこの章を書きました。

3 年前くらいに 15 万円の 3D プリンタを購入して楽しんでいたのですが、さすがに 15 万円は気軽に払える値段ではないため、周りに布教するにはいたっていませんでした。しかしここ数年で 3D プリンタはどんどん価格を下げ、とうとう 2.5 万円でそこそこの性能のものが購入できるようになったため、満を持して布教活動させていただくことにしたわけです。

2.2 3D プリンタはいいぞ!

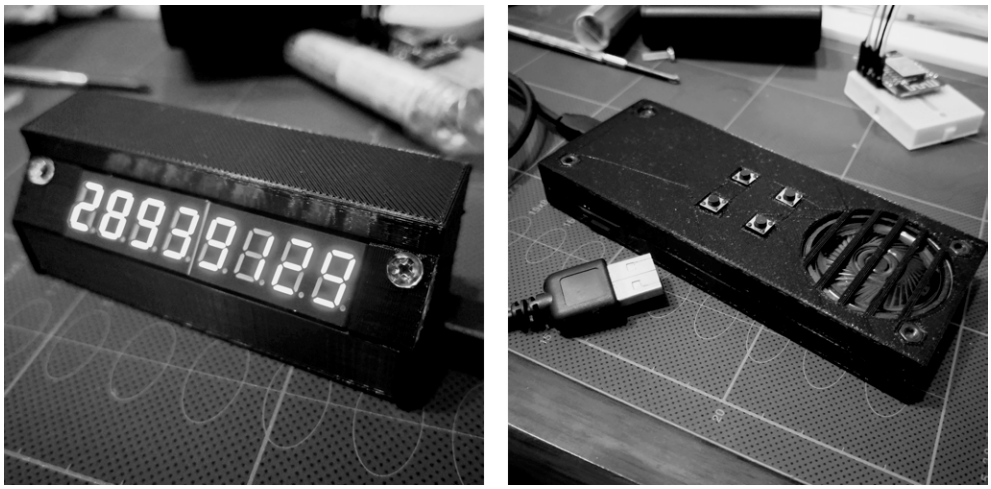
3D プリンタの魅力はなんといっても「**画面の中にあるものを現実世界に取り出すことができる**」につきます。画面の中にいる美少女キャラをフィギュアとしてプリントするに

[*1] マニアックでカオスなこの本を楽しめる人は KLab 適性高いと思うので是非面接に来てください (切実)。

はまだまだ精度的に厳しいですが、シンプルな 3D モデル[*2]であればそれほど違和感なく出力ができます。

▲図 2.1　CG の世界で有名なあのティーポットも現実世界に呼び出せます

また、お店では単独で売ってない部品（なくしてしまったテレビリモコンのフタとか）や、自分で工作したものをおさめるための専用のケースなども簡単に作ることができます。



▲図 2.2　専用ケースも自分で設計して自分でプリントできます。

[*2] Utah Teapot https://www.thingiverse.com/thing:2067607

2.3　今回紹介する 3D プリンタの種類について

一言で 3D プリンタといってもさまざまな種類の 3D プリンタがあります。形や機構の違いも大きいですが、造形方式によっていくつかの種類に分類できます。個人向けにおいて一番メジャーなのが**熱熔解積層方式**（FDM）です。熱を使って細い樹脂を溶かし、1 層ずつ積層していく方式です。個人向けの 3D プリンタは FDM 一択でしたが最近になって 6 万円から 15 万円くらいで購入できる格安の光造形プリンタが登場し、光造形式についても手が届く値段になっています。光造形は、紫外線で硬化する樹脂に光をあてて積層していきます。他にも粉末焼結方式という粉にレーザーをあてて硬化させる仕組みがあるのですがこれは最低 80 万-200 万円程度なのでまだ個人で扱うには難しい状況です。

今回は一番低価格でメジャーな FDM 方式を前提とします。

光造形のプリンタは"まだ"オススメしない

個人で趣味として使う光造形プリンタには FDM と比較していくつかの課題があるように感じるので個人的にはまだオススメできません。細かい話でいうと、液体を扱うのでミスってこぼすと大惨事とか、プリントごとに清掃するのが面倒とかありますが私がオススメできない大きな理由としては次の 2 点の問題があると考えています。

- 印刷に必要なレジンの価格の高さ（FDM 形式のプリンタの約 5 倍）
- レジンの硬化に使う LCD 寿命の短さ（使用期間にもよるが数ヶ月で交換する消耗品扱い）

この 2 点の問題はようするに「コスパが悪い」という話なのでカネで解決できる問題でもあります。光造形プリンタは FDM よりも遥かに小さなものや細かいものをプリントできるので、フィギュアの原型師の方がプロトタイプ作成に利用しているという話も聞きます。造形物をもっときれいにプリントしたいんだという人は挑戦してみてはいかがでしょうか。

2.4 3D プリントに必要なものを揃えよう

3D プリント生活をはじめるにあたって購入しなければならないものは実はたった 2 つしかありません。3D プリンタ本体とフィラメント[*3]です。フィラメントとは 2D プリンタでいうインクのことで、プリントする物体の素材になるものです。厳密には他にもいろいろなモノが必要になるのですが、たいていの場合、必要な機材は工具や SD カード含めて 3D プリンタに同梱されているので、あとはフィラメントを一緒に買えばプリントまでもっていけます。[*4]

しかも 3D プリンタとフィラメントは両方とも Amazon に売っているので数回クリックすると届きます。簡単ですね。

早速 3D プリンタを購入しよう

3D プリンタは安いものから高いものまで本当にたくさんあるのでどれを買えばいいのか最初はわからないと思います。いろいろ調べて何が違うのかを知ることは大切ですが、最初の 1 台であれば重要視するのは気軽に試せる安さと、不具合なく動かせる安心感かと思います。

そんな中で私がこれから 3D プリントはじめてみようかなと思っている初心者に勧めたいのは **Creality3D Ender-3** です。組み立てキットが本家である Creality3D から 2018 年 9 月 27 日時点での販売価格で 23300 円で購入できます。[*5]

Ender-3 はオープンソースのハードウェアなので、製造に必要な情報やファームウェアがすべて Web に公開されており、簡単にコピー品を作ることができるため、検索すると怪しい中華業者を筆頭にたくさんの販売業者が出てきます。国内の Amazon を使って Creality 3D から購入するのが現時点で一番安心ですが、aliexpress などの中国のショッピングサイトを使うと、国内の Amazon で買うよりもかなり安く購入できます。ただし、日本語サポートや何かあったときの返品手続きの難易度は大幅にあがるので、そのリスクに見合う値段の差かどうかは十分に検討したほうがよいでしょう。

フィラメントの購入

フィラメントとは、先ほども説明したように 2D プリンタでいうインクのようなもので、細長いプラスチックをリールにグルグル巻きにしたような形状で売っています。フィラメントはさまざまな種類の素材で作られており、その素材によってフィラメントの特性

[*3] 実はたいていの 3D プリンタ本体にはテスト用のフィラメントが少量同梱されているので、フィラメントを買わなくても最初のテストプリントまではできるようになっています。

[*4] 当然ですが、プリントデータを用意したり、プリント設定を行うために PC が必要です。

[*5] Creality3D Ender-3 販売ページ：`https://www.amazon.co.jp/dp/B07H2QN1H4/`

が変わります。最近はかなり種類が増えていますが[*6]、一番メジャーなフィラメント素材は PLA と ABS になります。

プリントするときは、フィラメントを 190-220 度程度の高温で溶かしつつ押し出すエクストルーダーと、プリント中の物体が収縮したりしないように物体の下にある板（ベッド）を 0-60 度前後に加熱するのですが、この加熱の条件がフィラメントによって変わってきます。PLA はベッドをそれほどあたためる必要がなく印刷開始までの時間が短いというメリットがあります。ABS は PLA と比較して強度は高いものの、エクストルーダーの温度が高めでベッドの十分な加熱が必要、プリント中に素材からでるニオイが少し強いという特徴をもっています。

3D プリンタによって扱うフィラメントの幅が異なり、Ender-3 の場合、フィラメントの直径が 1.75mm のものを使うので購入するときは 1.75mm のものを購入してください。

私は加熱時の負担を考えて PLA を中心に使っています。ちなみにだいたい 1kg で 2000 円です。[*7]先ほど紹介したプリンタ本体の価格と合計すると**約 2.5 万円で必要なものがすべて購入できることになります。**

2.5 プリントしてみよう

3D プリンタとフィラメントが届いて、説明書を見ながら組み立てたらいよいよプリントです。3D プリンタ本体に同梱された SD カードにはテストプリント用のデータが入っているので、それを使えばプリント自体はできるのですが、自分がプリントしたいと思うようなものを用意してプリントするまでにはいくつかの手順があるので、その手順を紹介します。

Thingiverse からプリントしたい 3D モデルをダウンロードする

3D プリンタがどんなに素晴らしいものであっても、プリントしたいと思うものがなければ無意味です。毎年お正月の年賀状のときだけ稼働する 2D プリンタと同じです。

自分でプリントしたいものをデザインできる能力や作るモチベーションがある人はよいのですが、何かを作る前にまずは便利なものやかっこいいものをプリントしてみたいですよね。

そんな時は Thingiverse というサイトが使えます。このサイトは一言でいえば 3D プリント版 Github のようなもので、みんなが自分で作った 3D データを公開しており、そこからデータをダウンロードしてプリントしたり、データを修正して再公開する Remix を行ったりできます。Thingiverse の Popular ページ[*8]にアクセスすると人気の作品がで

[*6] プリントすると木目のような質感になる wood フィラメントや、スマホケースに使われている弾性のある TPU フィラメント、透明フィラメントなどもあります。

[*7] HICTOP 3D プリンター用 高強度 PLA 樹脂 材料 フィラメント https://www.amazon.co.jp/dp/B07539P1JL/

[*8] Thingiverse Popular ページ：`https://www.thingiverse.com/explore/popular/`

てくるので、これを見ながら自分がプリントしてみたいものを探して見るとよいでしょう。[*9]

私がオススメするプリントデータはエンジニアなら誰でも 1 つは持っている Raspberry Pi のケースです。ラズパイは本体が安いので、ケースを買おうとすると値段が相対的に高く見えてしまい購入していない人も多いのではないでしょうか。3D プリンタなら電気代を加味しても原価 100 円以下でプリント可能です。

プリント手順を理解するために今回はこのケースをプリントする場合の手順を紹介します。

Thingiverse にはさまざまな種類のラズパイ用ケースがありますが、今回は Compact Raspberry pi B+ case derivative[*10]をプリントしてみましょう。URL にアクセスしたあと、「Thing Files」というリンクをクリックすると、3D モデルが記録されている STL ファイルのリストが表示されるのでここから STL ファイルをダウンロードします。

Cura のインストール

Thingiverse で 3D プリントするデータを手に入れることができたわけですが、このデータには 3D モデルに関するデータしかないためこれだけではプリントできません。

3D データをどの向きで、どれくらいの温度で、どれくらいの密度でプリントするかというプリントに関する情報が必要で、3D モデルをもとにその情報を作成し、gcode と呼ばれる最終的なプリント情報に変換するのが Cura というアプリケーションです。[*11]

Cura は Ultimaker という 3D プリンタを作っている会社が公開しています。[*12] Cura の Web ページから Download for Free と書かれたボタンを押し、Personal Projects を選択、適当に欄を埋めて Download を押しましょう。

Cura の設定

インストールが完了したら早速 Cura を起動して、メニューバーにある「ファイル」の「開く」から Thingiverse でダウンロードしたケースの STL データを読み込みます。

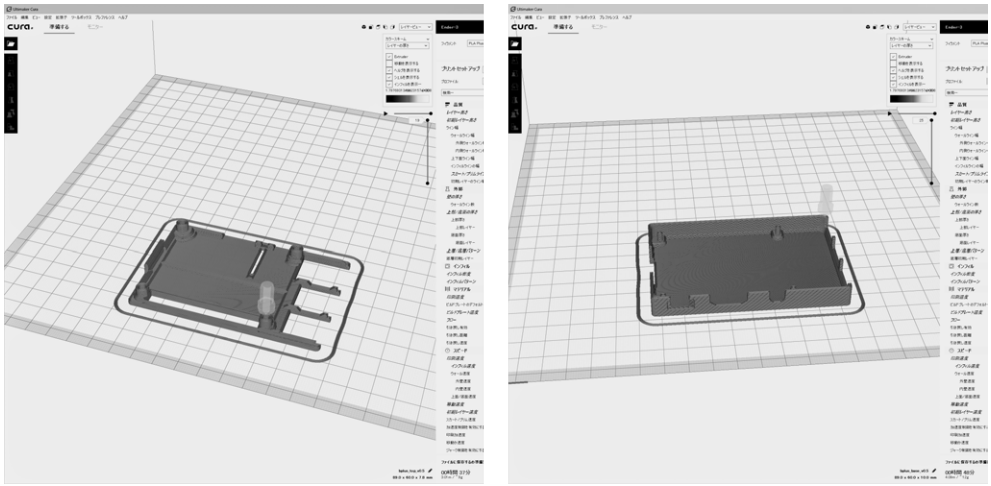
この画面では、読み込んだ 3D モデルを、プリンタのどの位置でプリントするのかなどを変更できます。プリンタのベッドと 3D モデルの接地面が多い方がプリントが安定するので、プリントしやすい位置に移動と回転させると次の画像のようになります。

[*9] もちろん作品を利用する際はライセンスを確認しておきましょう。作品ページに表示されているので簡単に確認することができます。

[*10] Compact Raspberry pi B+ case derivative https://www.thingiverse.com/thing:400624

[*11] 3D データやプリンタ情報をもとに gcode を出力するアプリケーションをスライサーといい、Cura 以外にも Slic3r や Simplify3D などがあります。

[*12] Cura Web ページ：`https://ultimaker.com/en/products/ultimaker-cura-software`



▲図 2.3　ケースは表面と裏面の 2 つで構成されているので 2 回にわけてプリントします

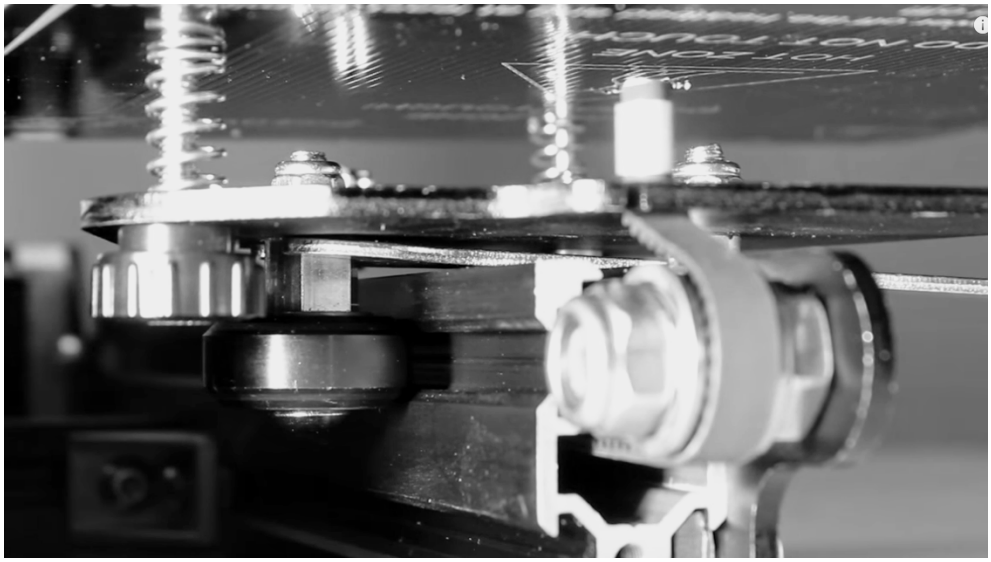
プリント位置が定まったら、プリント設定の確認です。フィラメントの加熱温度やベッドの温度、印刷速度は素材やプリントデータの複雑さ、プリンタの種類によって最適値が異なります。[*13]一度安定してプリントできる設定が見つかればそれ以降はあまり変更しなくてもよくなるのですが、はじめてプリントする場合、どの値が最適解なのかわかりづらいので、数をこなして少しづつ調整していくのがよいでしょう。

設定値が決まったら MicroSD カードに gcode を書き込みます。右下の保存ボタンから書き込みを行い、3D プリンタに MicroSD カードを刺せば印刷準備完了です。

プリント前ベッドレベルチェック

さぁ印刷！ と言いたいところですが、やっておかないといけないことがまだあります。3D プリンタのベッドのレベリングです。簡単にいうと、フィラメントを出力するノズルの先端からまでのベッドの距離を紙一枚通る距離で水平にする手続きなんですが、これは印刷品質に直結するので特に時間がかかる大きなものを印刷する時は失敗するリスクを減らすために入念にチェックしておくことをオススメします。Ender-3 の場合、ベッド底面に 4 箇所あるハンドルを回すとスプリングの圧力が変わってベッドの傾きが変わるようになっているので、この 4 箇所のハンドルを少しづつ回しながら紙をノズルとベッドの間に差し込んでベッドのどの位置でも同じ圧力になるようにしていきます。

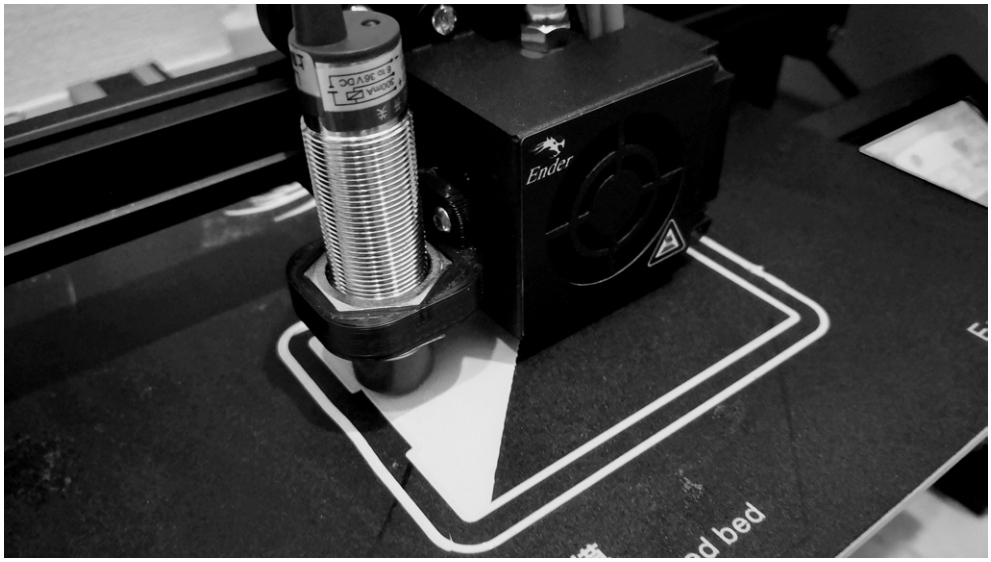
[*13] 同じ種類のプリンタでもタイミングベルトの貼り方や、XYZ 軸の状態によって変わってくるので真にベストな設定は各人が見つける必要があります



▲図 2.4　スプリングの圧力を微妙に変えながらベッドを水平にしていきます。慣れるとすぐできます。

プリント開始

プリントボタンを押したら我々にできることはほとんどなくて成功を祈って見守るか、失敗を確認して中止ボタンを押すかの 2 択だけです。プリント開始ボタンを押すと同時に完成予定時間まで外出するのもよいですが、プリントの成功可否は最初の一層目で決まることが多いので、一層目が正しくでているかは目視で確認しておくとよいです。



▲図 2.5　きれいにできることを祈りながら 1 層 1 層積まれていくのを見守りましょう

プリント終了

フィラメントの素材や、造形物の形状、ベッドの素材によって状況は変わってくるのですが、PLA はプリントが終わってすぐの状態だと剥がれづらいので、簡単にとれそうにない場合はいったん冷えるまで待つと剥がしやすくなるでしょう。今回のプリントデータには印刷しやすくするためのサポートがついているのでそれを剥がせば印刷完了です。

▲図 2.6　出力できたラズパイケース

2.6 おわりに

3D プリンタがあれば、デジタルの世界にあるものをリアルに持ってくることができます。もちろん 2.5 万円で買えるプリンタではまだまだ限界を感じることも多いですが、ケースや小さな部品などは実用的なレベルで印刷できます。数年後にはきっと今のものよりさらに使いやすいプリンタが開発されて、より大きなものや複雑なものを誰もが簡単にプリントできる時代がやってきそうですが、そうなる前にプリンタを購入して、他の人よりも早く 3D プリント体験をしてみるのはいかがでしょうか。

第 3 章

Airtest を用いた Unity アプリの自動実機テスト

Daisuke TAKAI

3.1 はじめに

アプリのテストを自動で行いたいと思ったことはありませんか？

そこで、この章では Unity で開発されたスマートフォン向けアプリの実機テストについて書きたいと思います。Airtest というフレームワークを利用しますが、Unity 製スマートフォン向けアプリの実機テストができるのは Android のみとなっています。（執筆時点）

Airtest とは

Airtest[*1]とは、ゲームやアプリのための UI 自動テストフレームワークです。クロスプラットフォームの自動テストフレームワークで Windows や Android、iOS でアプリの自動テストができます。Unity などで開発したアプリもテストできるフレームワーク Poco[*2]が提供されています。今回はこの Poco も一緒に使っていきます。

Airtest の仕組み

Airtest は Android の場合は ADB で接続しています。タップなどの操作は ADB を介して行われます。また、ADB とは別に RPC で内部メソッドを呼び出しており、Unity シーンの Hierarchy 情報などをダンプして Airtest IDE に送っています。この情報を使い、テストを簡単に記述できるような仕組みがあります。

[*1] https://github.com/AirtestProject/Airtest
[*2] https://github.com/AirtestProject/Poco

さらに OpenCV も利用することができ、画像が正しく表示されているかの判定などに用いることが可能です。

3.2 導入方法

導入方法について説明します。Unity プロジェクトへの導入方法は、公式ドキュメント[*3]にもありますので、あわせてお読みください。Airtest を使うためには PocoManager を GameObject にアタッチして Unity シーン上に配置しておく必要があります。

まず、Poco[*4]を clone します。clone したプロジェクト内の Unity3D フォルダを Unity プロジェクトにコピーします。NGUI を使っているのであれば Unity3D/ngui のみを残し、uGUI を使っているのであれば Unity3D/ugui を残してください。PocoManager.cs は mainCamera などにアタッチするとよいと書いてありますが、結局は全シーンでテストを行うため、常にシーン上に配置されるように永続化するためのクラスを作成しました。

▼リスト 3.1　PocoManager 永続化のためのクラス

```
 1: using UnityEngine;
 2:
 3: /// <summary>
 4: /// Poco を永続化するための GameObject
 5: /// </summary>
 6: public class PocoObject : MonoBehaviour
 7: {
 8:     static PocoObject instance;
 9:
10:     public static PocoObject Instance
11:     {
12:         get
13:         {
14:             if (instance == null)
15:             {
16:                 instance = new GameObject(typeof(PocoObject).Name)
17:                                 .AddComponent<PocoObject>();
18:                 instance.gameObject.AddComponent<PocoManager>();
19:                 DontDestroyOnLoad(instance.gameObject);
20:             }
21:
22:             return instance;
23:         }
24:     }
25:
26:     public void Initialize()
27:     {
28:     }
29: }
```

アプリの最初の方でこのクラスの Initialize() を呼び出せば、シーン上に永続化された形で配置されます。

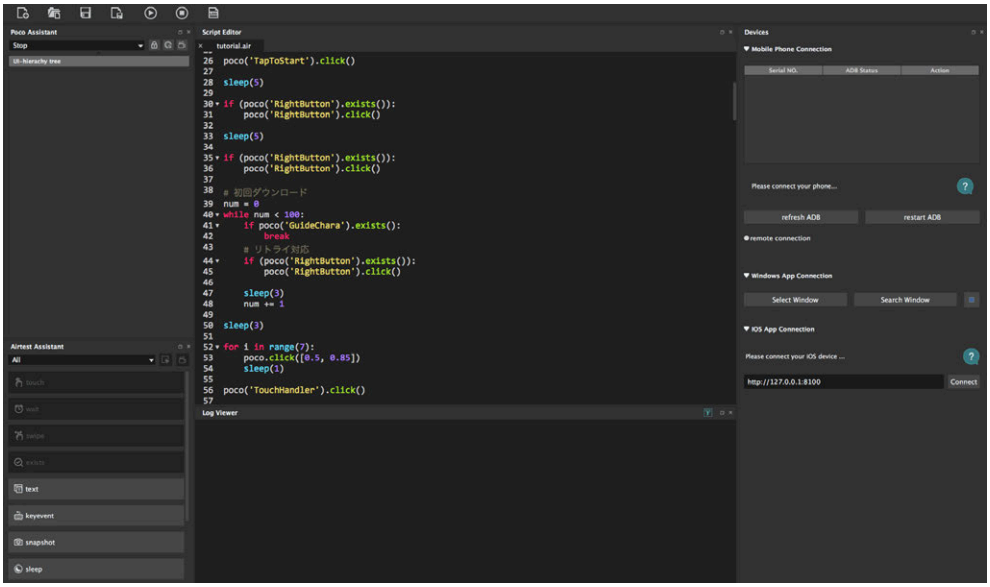
ハマるポイントとして Transform の親の下に RectTransform の子供がいるシーン構造

[*3] https://poco.readthedocs.io/en/latest/source/doc/integration.html
[*4] https://github.com/AirtestProject/Poco-SDK

だと Airtest 実行時にエラーになります。開発の進んだ Unity プロジェクトに Airtest を導入する場合は気をつけてください。

3.3 テストの書き方

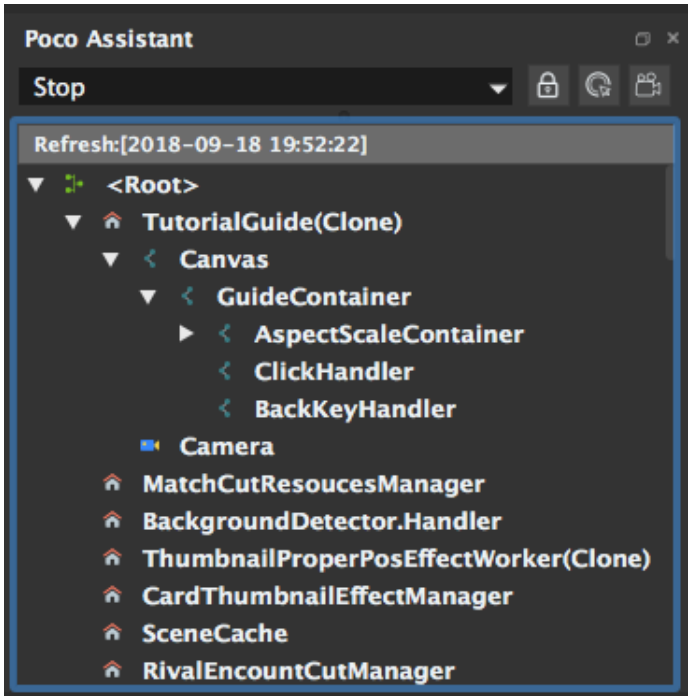
テストは Python で書くことができます。図 3.1 はテストを書くための IDE[*5]です。これを利用すれば簡単にテストを作成することができます。



▲図 3.1　Airtest IDE の画面

Android を接続し、実際にアプリを起動しながらテストを書くことで Unity の Hierarchy 構造を取得し、対象の GameObject に対しての操作をテストとして書けます。Hierarchy の構造は図 3.2 のように表示されます。

[*5] http://airtest.netease.com/index.html



▲図 3.2　Airtest IDE 上の Hierarchy 情報表示の例

さらに Hierarchy の構造だけでなく、実機画面も IDE 上にプレビューされます。「タップする」などのアクションを図 3.1 の左下にあるような一覧から選び、実機プレビュー上で「ボタン」の GameObject を選択すれば「ボタンをタップする」というテストコードを自動生成することもできます。

テストコードの一例を載せておきます。if 文などのコード以外は前述したように自動的に生成することができます。タップ操作などは自動生成し、条件に合わせて if 文や for 文を追加する方法が楽だと思います。

▼リスト 3.2　テストスクリプトの一例

```
# タップするだけ
poco('Button').click()

# 親子関係がある場合
poco('Window').child('Button').click()

# 表示されているかチェック
if (poco('Button').exists()):
    poco('Button').click()
```

3.4 Jenkins との連携

やはりテストの実行も自動化したいので、Jenkins から叩けるようにしました。上流のプロジェクトでアプリをビルドし、そのバイナリを Airtest を走らせるジョブに成果物としてコピーしています。テストスクリプトは専用のリポジトリを作り、ジョブ実行時に clone して最新の状態にしています。今回はゲームのチュートリアルのテストを想定して、テストのファイル名を tutorial.air にしています。ADB を使うことで、インストールから実行まで Jenkins 上から操作できます。

▼リスト 3.3　Jenkins で走らせるスクリプト

```
 1: #!/bin/bash
 2:
 3: # ADB からインストール
 4: adb install ${WORKSPACE}/*.apk
 5:
 6: # コピーした成果物、前回のログなどを消している
 7: git clean -fd
 8:
 9: # テストを実行するアプリのアクティビティ名を指定して、アプリを起動
10: adb shell am start -n "${ANDROID_BUNDLE_IDENTIFIER}/アクティビティ名"
11:
12: # airtest の実行
13: airtest run "${WORKSPACE}/tests/tutorial.air" --device Android:/// --log
14: RET=$?
15: # レポートの生成
16: airtest report "${WORKSPACE}/tests/tutorial.air"
17:
18: # アプリ終了
19: adb shell am force-stop "${ANDROID_BUNDLE_IDENTIFIER}"
20: # アプリのアンインストール
21: adb uninstall ${ANDROID_BUNDLE_IDENTIFIER}
22:
23: exit $RET
```

3.5 おわりに

Airtest というフレームワークを使い、Android 実機上で自動テストをすることができました。テストシナリオを比較的簡単に作成できる Airtest IDE を紹介しました。さらに、ADB を利用して Jenkins からテストを実行することもできました。

iOS もネイティブアプリはテスト実行できるようになっているようです。今後、Unity 製アプリの iOS 実機テストもできるようになるとよいなと思っています。

第 4 章

Rider + Unity で Roslyn Analyzers を使う

Yoshihiro KASHIMA

4.1 はじめに

3 年遅れ （CTP から計算すると 4 年遅れ）で Roslyn Analyzers をやっています。

筆者の開発する環境では、開発者全員が Windows ではなく、Mac が 8 割で Windows が 2 割という環境です。Mac については、Visual Studio for Mac を使っている人がほとんどおらず、JetBrains Rider のユーザーの方が多い状態です。そのため Unity で Roslyn Analyzers を使おう、と思っても IDE 環境の違いからコーディング規約、実装スタイルが人によって異なるという形にしかできませんでした。

しかし 2018 年になり、JetBrains Rider の Roslyn Analyzers 対応[*1]、Visual Studio Code への実装[*2]が進んでいるなど、Visual Studio 以外の環境でも Roslyn Analyzers を使える環境が整いつつあります。さらに、JetBrains Rider では Visual Studio の Roslyn Analyzers に関する設定ファイルを読み込むことも可能になっています。これらの変更によって、OS や IDE の差異の影響を受けない、一元的な規則に基づいて Roslyn Analyzers を導入することができるようになっています。

本稿では Visual Studio と Rider のように複数の IDE の共存を図りつつ、 Unity 環境で Roslyn Analyzers を導入する方法について紹介していきます。

[*1] https://blog.jetbrains.com/dotnet/2018/03/22/roslyn-analyzer-support-rider-2018-1-eap/
[*2] https://github.com/OmniSharp/omnisharp-roslyn/pull/1076

4.2 Roslyn Analyzers について

Roslyn Analyzers は、.NET コンパイラプラットフォームである Roslyn のコード解析部分を API として提供しているものです。主に構文木を解析する Analyzer と、エラーに対して修正処理を行う CodeFix Provider の 2 つが機能として提供されています。

2 つを組み合わせることで、Microsoft が規定していない独自のルールセットを定義し、さらに違反したコードの修正機能を提供することができます。

たとえば、コーディングルールとして string.Empty、もしくは""の強制といったことができます。

さらに、CodeFix Provider の活用メインで利用するケースもあります。たとえば、DataContract 属性のついたクラスについて、すべてのメンバーに DataMember 属性をつけてほしいという場合があるかと思います。そのような時に、DataContract 属性があったら DataMember 属性をつけるというアナライザーと CodeFix を作成しておき、次のようなコードを書きます。

▼リスト 4.1　DataMember 付与前のコード

```
[DataContract]
public class TestData
{
    public string Name { get; set; }
    public int Age { get; set; }
}
```

DataMember を付与していないため、TestData の宣言部分でアナライザー違反が出るように作っておきます。この違反に対する CodeFix を行うことで、リスト 4.2 のコードに変換できます。

▼リスト 4.2　DataMember 付与後のコード

```
[DataContract]
public class TestData
{
    [DataMember]
    public string Name { get; set; }
    [DataMember]
    public int Age { get; set; }
}
```

若干わかりづらい例になってしまいましたが、クラスに属性を付与して CodeFix を実行して目的のコードを得る、という使い方も可能となっています。そのため Roslyn Analyzers は独自のコード解析と CodeFix、またそれらを用いた開発効率の向上を図るための機能となっています。

4.3 導入の前提について

どんな Mac でも OK！ というわけではないので、いくつか前提条件を書いておきます。

- Mac OSX
  - Mono 5.8.0.0 以上 （可能であれば 5.14 以降を推奨）
- Unity
- Rider 2018.2 以上

Rider 以外の下限については正確には把握できていません。Mono についてはもう少し低いバージョンでも可能かもしれません。（とはいえ Mono は 5.6 がスキップされていて、5.4.1.6 では動作しなかったことから 5.8.0.0 がいったんベースになりそうです。）Unity は 2018 年 9 月現在でサポートされているバージョンなら可能です。5.6.x 以降であれば問題なく動きます。

Rider については、2018.1 で Roslyn Analyzers に対応しているのですが、他 PC 環境との設定の共有が非常に難しく、Windows との互換性は皆無なため、2018.2 以上を推奨します。個人環境であれば 2018.1 でも実行可能です。

Windows については Visual Studio 2015 が動く環境であれば問題ありません。

4.4 Unity の自動生成プロジェクトファイルを編集する

Roslyn Analyzers を任意のプロジェクトで実行するには、そのプロジェクトの csproj ファイルに Analyzer の DLL を追加する必要があります。Unity プロジェクトでは、Assembly-CSharp.csproj を修正することになるのですが、若干問題となるのが.csproj ファイルが自動で生成されるという点です。

自動生成されるということは、一回手で書き換えても再度生成すると変更した内容が失われるということを示しています。

この点について、Visual Studio では、Visual Studio Tools for Unity（以降 VSTU）にある Project File Generation[*3]というイベントを提供しています。`ProjectFilesGenerator.ProjectFileGeneration`イベントに紐づけたメソッドについて、Unity が.csproj ファイルを生成するときに呼び出すようにする、という仕組みです。

しかしこの仕組みは全員が Visual Studio を使っている環境でないといけません。今回のように Rider を用いるユーザーがいるケースでは VSTU だけの解決法は利用できません。

IDE 間の差異をなくすため、Rider 固有の解決方法ではなく、Unity プロジェクトを扱える IDE であればどれを使っても解決できる方法を利用していきます。

[*3] `https://docs.microsoft.com/en-us/visualstudio/cross-platform/customize-project-files-created-by-vstu?view=vs-2017`

Unity では、アセットのインポートパイプラインに干渉する手段として AssetPostprocessorを提供しています。この中に、csproj ファイルの生成タイミングで任意のメソッドを呼び出す処理が実装されています。

今回使う CallOnGeneratedCSProjectFiles[*4] を書き出してみると次のとおりです。

▼リスト 4.3　Unity 側から csproj 生成時にユーザーコードを呼び出すメソッド

```
static internal void CallOnGeneratedCSProjectFiles()
{
    object[] args = {};
    string methodName = "OnGeneratedCSProjectFiles";
    foreach (var method in AllPostProcessorMethodsNamed(methodName))
    {
        method.Invoke(null, args);
    }
}
```

AllPostProcessorMethodsNamedは、AssetPostprocessorを継承するすべてのクラスから、引数で指定した名前で定義されたメソッドの一覧を取得しています。取得したメソッド一覧を foreach ですべて呼び出すというだけの簡単な仕組みです。ただしドキュメント化されていないため[*5]、実装するにあたり MethodInfo.Invokeの引数を頼りに実装する必要があります。

OnGeneratedCSProjectFiles を最低限実装した結果が次のとおりです。

▼リスト 4.4　csproj 生成イベントをハンドルするサンプル

```
public class AnalyzersPostprocessor : AssetPostprocessor
{
    public static void OnGeneratedCSProjectFiles()
    {
        Debug.Log("AnalyzersPostprocessor Called!")
    }
}
```

この AnalyzersPostprocessor をエディタ拡張スクリプトのフォルダに追加します。Asset 以下にある任意の Editor フォルダに追加すればよいです。

ここまでの内容で csproj に干渉する準備が整ったので、実際に Roslyn Analyzers を追加していきます。

Roslyn Analyzers を追加する

まず、前提として Roslyn Analyzers の DLL ファイルを手元に用意しておく必要があります。DLL についてはアナライザーの GitHub リポジトリから、または.nuget フォル

[*4] https://github.com/Unity-Technologies/UnityCsReference/blob/2018.2/Editor/Mono/AssetPostprocessor.cs#L96-L105

[*5] Visual Studio の仕様変更で生成した csproj が読み込めないときに、ユーザーが緊急回避できるようにするための実装なので基本は不要

ダを探索することで入手できます。NuGet から.nupkg を入手しても直接解凍できないので注意してください。

追加したいアナライザーの DLL ファイルですが、分かりやすい位置にまとめて入れておきます。今回は csproj と同じ階層に"Analyzers"　というフォルダを作り、その中に入れておくことにします。ディレクトリ構造について一部抜粋すると次のような形です。

```
*
├── Analyzers
├── Assets
├── ProjectSettings
│
└── Assembly-CSharp.csproj
```

ファイルの準備ができたら実際に csproj ファイルを編集していきます。今後のコードについては先ほどの `AnalyzersPostprocessor`に記述します。

csproj については中身は XML のため、編集しやすいように `System.Xml.Linq.XDocument`に読み込んで編集します。

▼リスト 4.5　AssetPostprocessor で csproj ファイルの一覧を取得するように変更したもの

```
public class AnalyzersPostprocessor : AssetPostprocessor
{
    public static void OnGeneratedCSProjectFiles()
    {
        var currentDir = Directory.GetCurrentDirectory();
        var projectFiles = Directory.GetFiles(currentDir, "*.csproj");

        foreach (var file in projectFiles)
        {
            var document = XDocument.Load(file);
            // ToDo : 編集するメソッドを呼び出す
            // AddAnalyzerReference(
            //     document.Root, document.Root.Name.NamespaceName);
        }
    }
}
```

続いて、Roslyn Analyzers を csproj に追加するメソッドを作成します。ItemGroup の要素に Analyzer という要素を追加することで追加することができます。Analyzer の要素は Include 属性で DLL のパスを指定する必要があります。

▼リスト 4.6　Roslyn Analyzer のライブラリを追加するメソッド

```
private static readonly string AnalyzerDirectoryPath = "Analyzers";

private static void AddAnalyzerReference(XElement projectRoot, XNamespace xmlns)
{
    var itemGroup = new XElement(xmlns + "ItemGroup");

    var currentDir = Directory.GetCurrentDirectory();
    var analyzerPath = Path.Combine(currentDir, AnalyzerDirectoryPath);
    var files = Directory.GetFiles(
        analyzerPath, "*.dll", SearchOption.AllDirectories);

    foreach (var analyzer in files)
```

```
    {
        var analyzerElement = new XElement(xmlns + "Analyzer");
        analyzerElement.Add(new XAttribute("Include", analyzer));
        itemGroup.Add(analyzerElement);
    }

    projectRoot.Add(itemGroup);
}
```

先ほど作成した Analyzer フォルダの子孫フォルダ内に存在するすべての.dll ファイルをアナライザーとして追加しています。フォルダ構成に応じてコードは適宜変更してください。

ここまで記述できたら Unity 側の Asset メニューにある"Open C# Project"を実行します。csproj ファイルを再生成することで Analyzer が追加されるようになるはずです。追加できたらアナライザーの設定ファイルも読み込むようにしていきます。

規則セットと AdditionalFiles の読み込み

Roslyn Analyzers でも、通常の C#言語規則同様に、.ruleset ファイルで適用するルールを設定することができます。.ruleset の内容については詳しくは触れませんが、Visual Studio であれば追加のルールセットの作成で作成ができるため、まず Visual Studio で作成を行い、その後調整を行っていく形が手軽に作成できるかと思います。中身については XML のため、一度作れば編集は容易です。

追加にあたっては、PropertyGroup に CodeAnalysisRuleSet として追加する必要があります。追加する PropertyGroup はプロジェクト全体の設定ではなく、ソリューション構成、プラットフォームごとの PropertyGroup に追加する必要があります。ソリューション構成の PropertyGroup すべてに追加する必要があるので若干手間ですが、Condition 属性があるかどうかで判定が可能なため追加処理は難しくありません。なお csproj の完全な構成については把握していないため、次のコードについては Visual Studio の出力に基づいて追加するようにしています。そのため不要な属性も追加している可能性があります。

▼リスト 4.7　ルールセットファイルを追加する処理

```
private static void AddRuleSetFiles(XElement rootElement, XNamespace xmlns)
{
    var itemGroup = new XElement(xmlns + "ItemGroup");
    var currentDir = Directory.GetCurrentDirectory();
    var analyzerPath = Path.Combine(currentDir, AnalyzerDirectoryPath);
    var files = Directory.GetFiles(
        analyzerPath, "*.ruleset", SearchOption.AllDirectories);

    var attrVal = "'$(Configuration)|$(Platform)'";
    var propGroups = rootElement.Elements().Where(e =>
        e.Attribute("Condition") != null &&
        e.Attribute("Condition").Value.ToString().Contains(attrVal)).ToArray();

    if (files.Length == 0)
    {
```

```
            return;
        }

        foreach (var ruleset in files)
        {
            var relativePath = ruleset.Replace(currentDir, "").Substring(1);
            var rulesetReference = new XElement(xmlns + "None");
            rulesetReference.Add(new XAttribute("Include", relativePath));
            var ruleElement = new XElement(xmlns + "CodeAnalysisRuleSet");
            ruleElement.SetValue(relativePath);
            foreach (var group in propGroups)
            {
                group.Add(ruleElement);
            }

            itemGroup.Add(rulesetReference);
        }

        projectContentElement.Add(itemGroup);
    }
```

ビルドアクションの None で ruleset ファイルを追加した上で、CodeAnalysisRuleSet も追加しています。このとき、CodeAnalysisRuleSet については正確な相対パスを入力する必要があります。絶対パスの入力では追加できないため注意が必要です。

続いて Roslyn Analyzers の各ライブラリが利用する設定ファイルを読み込みます。ビルドアクションのひとつである"Additional Files"を Roslyn Analyzers は受け取ることができるので、Additional Files で設定を記述することがスタンダードになっています。csproj 上での Additional Files の扱いについては Roslyn Analyzers とほとんど一緒です。追加する手法についてもほぼ同じにすることができます。設定ファイルについてはフォーマットが決まっていませんが、いったん StyleCop.Analyzers に則って json ファイルであると仮定してコードを書くと次のとおりとなります。

▼リスト 4.8　オプションとなる AdditionalFiles を追加する処理

```
private static void AddAnalyzerOptions(XElement projectRoot, XNamespace xmlns)
{
    var itemGroup = new XElement(xmlns + "ItemGroup");

    var currentDir = Directory.GetCurrentDirectory();
    var analyzerPath = Path.Combine(currentDir, AnalyzerDirectoryPath);
    var files = Directory.GetFiles(
        analyzerPath, "*.json", SearchOption.AllDirectories);

    foreach (var additionalFile in files)
    {
        var optionElement = new XElement(xmlns + "AdditionalFiles");
        optionElement.Add(new XAttribute("Include", additionalFile));
        itemGroup.Add(optionElement);
    }

    projectRoot.Add(itemGroup);
}
```

これでほぼすべてのアナライザーで必要なファイルを読み込むようになります。

4.5 Roslyn Analyzers の制限

ここまでの話で Roslyn Analyzers を Unity Project に追加することができました。めでたしめでたしというところなのですが、Unity ではアセットストアからパッケージを落としてきたくなります。ちょっとユニティちゃんモデルを使おうと思って入れることはよくある話かと思います。

しかし、このようにしてアセットストアからパッケージを入れると自分のプロジェクトに別の人のコードが入ってしまいます。ユニティちゃんモデルでも、Scripts 以下にモデルをキー入力で操作するコントローラや、カメラを制御する C#ファイルが含まれています。これらは DLL ではなく.cs ファイルです。当然 csproj の中の 1 ファイルとして認識され、ビルド時には一緒にビルドされます。

何が問題かというと、この別の人の書いたコードが Roslyn Analyzers の解析対象とされてしまうことです。別の人の書いた C#なのですから、当然自分たちのコーディング規約など知るはずもありません。Analyzer は大量に他の人のコードの規約違反を指摘するでしょう。もし CI テストとして Roslyn Analyzers を導入したい、と考えても他人のファイルがエラーとして大量に記録されてしまっては導入することができません。

Roslyn Analyzers では、コードがあるということはユーザーが管理すべきソースコードとして認識します。管理してるコードについては規約に沿っているはずであるという方針と、C#では NuGet を用いて DLL を追加する方法が主流となってきた時期であったため、特定のファイルを制限しようという動きがそもそもなかったためです。

上記については Roslyn の Issue[*6] でも話題となっており、改善しようという流れ自体はあるのですが、具体的な仕様までは決まってはいません。公式なサポートを待つのもひとつの手なのですが、すぐに解決する話とも思えません。

自分のコードは Roslyn Analyzers で解析していきたいけど、アセットストアから持ってきたコードは解析対象外としたい！ ということで、Roslyn Analyzers の解析範囲を何とか制限する方法について書いていきます。

4.6 自動生成ファイルとしてマークして解析範囲外にする。

最初期の Roslyn は、本当にすべてのファイルを解析対象としていました。ですがこれだと Visual Studio が生成しているファイルも解析対象になってしまいます。生成されるファイルはビルドタイミングで再生成をするため直したところで再度違反するようになってしまいます。

そのため、Roslyn Analyzers では、一定の規則に基づいて自動生成されたファイルを認識しており、自動生成として認識したものは Roslyn 側で解析しないような仕組みが導入

[*6] https://github.com/dotnet/roslyn/issues/3705

されています。この解析の実装は Roslyn.Utilities.GeneratedCodeUtilities[*7]にまとまっています。

上記のファイルに記述されている条件を書き出すと、次のとおりとなります。

- ファイル名の条件
  - ファイル名が次の文字列から始まる（大文字小文字問わず）
    - TemporaryGeneratedFile_
  - 拡張子を含まないファイルの末尾が次のいずれかである。
    - .designer
    - .generated
    - .g
    - .g.i
- ファイル内容の条件
  - SyntaxTree に含まれる最初のコメントの SyntaxTrivia が次のどちらかの文言を含んでいる
    - <autogenerated
    - <auto-generated

これらの条件のいずれかに当てはまるコードについては自動生成したファイルとして認識され、Roslyn 側で除外されます。

とはいえ、ファイル名を変えるのはどこに依存しているか分かったものではないです。特に Unity ではファイル名を変更することで Unity 側の参照が切れる、といった可能性も考えられます。そのように考えた場合、ファイル名の変更を行うよりもファイル内容を変更する方がリスクを低減できます。

一個一個ファイルの先頭にコメントをつけるのは現実的ではないので C#で簡単なロジックを記述して対処します。（ここは AnalyzersPostprocessor とは関係ないです。）

▼リスト 4.9　auto-generated をファイルに付与する処理

```
private string[] targetDir = { "" };
private void WriteAutoGenerated(string path, bool isRemove)
{
    var autoGenTag = "// <auto-generated />";
    var culture = StringComparison.CurrentCultureIgnoreCase;
    var files = Directory.GetFiles(path, "*.cs", SearchOption.AllDirectories)
        .Where(x => targetDir.Any(y => x.Contains(y)));
    foreach (var file in files)
    {
        var isExists = false;
        using (var sr = new StreamReader(file))
        {
            var firstLine = sr.ReadLine();
            isExists = firstLine.Contains("<auto-generate", culture)
                || firstLine.Contains("<autogenerate", culture);
```

[*7] https://github.com/dotnet/roslyn/blob/master/src/Compilers/Core/Portable/InternalUtilities/GeneratedCodeUtilities.cs

```
        }

        if (isExists && isRemove)
        {
            var lines = File.ReadAllLines(file);
            File.WriteAllLines(file, lines.Skip(1).ToArray());
        }
        else if (!isExists && !isRemove)
        {
            var lines = File.ReadAllText(file);
            lines = string.Concat(autoGenTag, Environment.NewLine, lines);
            File.WriteAllText(file, lines);
        }
    }
}
```

.NET Core で作れば OS 間の差異も無視できるので問題ありません。対象ファイルの判定については適宜書き直す必要がありますが、これで auto-generated の付与、および削除をコマンド 1 回で行えます。1 個ずつ足した場合は負債になりかねませんが、このレベルであれば低いメンテナンスコストを維持できると考えます。

4.7 アナライザー側でファイル判定を行う処理を追加する。

auto-generated をコードの最初に記述することで解析対象にならないことについて紹介しましたが、どうしても auto-generated を付与するのを避けたいという場合があると思います。そのようなケースではこの節で紹介するアナライザーそのものを書き換え、ファイル判定するロジックを追加する手法を取らざるを得ません。

処理を追加する場合、コードを書き換える必要があるためアナライザーがオープンソースである必要があります。またメンテナンスコストについても auto-generated を付与するよりはかかるため、アナライザーを複数導入したいという場合は注意が必要です。

では実際にファイルを判定する処理について記述していきます。ポイントとなるのは次の 3 点です。

- `DiagnosticAnalyzer`への介入
- 解析中のファイルパスの取得
- パスの判定処理

上から見ていくと、`DiagnosticAnalyser`への介入については選択肢がほぼないため、`DiagnosticAnalyzer.Initialize`を上書きすることで対処します。というのも、DiagnosticAnalyser でオーバーライドできるのは `SupportedDiagnostics`と `Initialize`しかありません。SupportedDiagnostics はアナライザーの ID、解析の内容などの情報が入っているだけなので、実際の解析処理を行う Initialize に手を加えます。

修正の内容としては、引数の `AnalysisContext`と同一のメンバーをもつクラス A を定義し、元の Initialize の引数に A を渡すことで介入の余地を作ります。例として、修正を行う前の DiagnosticAnalyser のサンプルを次に示します。

▼リスト 4.10　DiagnosticAnalyzer の実装サンプル

```
[DiagnosticAnalyzer(LanguageNames.CSharp)]
internal class SampleAnalyzer : DiagnosticAnalyzer
{
    public override ImmutableArray<DiagnosticDescriptor> SupportedDiagnostics
        => ImmutableArray.Create(descriptor);

    public override void Initialize(AnalysisContext context)
    {
        context.ConfigureGeneratedCodeAnalysis(
            GeneratedCodeAnalysisFlags.None);
        context.EnableConcurrentExecution();

        context.RegisterSyntaxNodeAction(
            XmlElementAction, SyntaxKind.XmlElement);
    }
}
```

上記のコードをクラス A で置き換えた結果が次のコードとなります。

▼リスト 4.11　ファイルパス判定処理を追加できるようにした DiagnosticAnalyzer

```
[DiagnosticAnalyzer(LanguageNames.CSharp)]
internal class SampleAnalyzer : DiagnosticAnalyzer
{
    public override ImmutableArray<DiagnosticDescriptor> SupportedDiagnostics
        => ImmutableArray.Create(descriptor);

    public override void Initialize(AnalysisContext context)
    {
        this.Initialize(new CapsuledAnalysisContext(context));
    }

    private void Initialize(CapsuledAnalysisContext context)
    {
        context.ConfigureGeneratedCodeAnalysis(
            GeneratedCodeAnalysisFlags.None);
        context.EnableConcurrentExecution();

        context.RegisterSyntaxNodeAction(
            XmlElementAction, SyntaxKind.XmlElement);
    }
}
```

`CapsuledAnalysisContext` がクラス A の定義となります。この方法で実装することで、既存の `DiagnosticAnalyzer`への変更が、複数行の置換で対応できます。

▼リスト 4.12　置き換え処理の置き換え元

```
    public override void Initialize(AnalysisContext context)
```

▼リスト 4.13　置き換え処理の置き換え先

```
    public override void Initialize(AnalysisContext context)
    {
        this.Initialize(new CapsuledAnalysisContext(context));
    }

    private void Initialize(CapsuledAnalysisContext context)
```

プロジェクト内に含まれるリスト 4.12 をリスト 4.13 で置換することで作業が終わるので、再度置換の必要が出ても少しのコストで対応ができます。ただし二重置換については注意が必要なため、2 回目以降は置換する範囲を絞る必要があります。

`CapsuledAnalysisContext` の実装については、AnalysisContext と同じメソッドを持っていれば問題ありません。方法としては、AnalysisContext と同じメソッド・引数をもつ新しいクラスを作成するか、AnalysisContext を継承したクラスで可能な限りオーバーライドして実装を行う方法の 2 つが存在します。基本どちらでも問題はないですが、AnalysisContext に定義されているメソッドのすべてがオーバーライド可能ではないため、必要なメソッドが abstract・virtual で定義されていない場合は継承の方法は取れません。その場合は同名の定義を持ったラッパークラスを作成する必要があります。

パスの取得については別途対応することとして、ひとまず実装を行います。`CapsuledAnalysisContext`の一部を次に示します。

▼リスト 4.14　AnalysisContext のラッパークラスの実装

```
internal class CapsuledAnalysisContext
{
    public CapsuledAnalysisContext(AnalysisContext c) => this.Context = c;

    private AnalysisContext Context { get; }

    public void ConfigureGeneratedCodeAnalysis(
        GeneratedCodeAnalysisFlags analysisMode)
    {
        this.Context.ConfigureGeneratedCodeAnalysis(analysisMode);
    }

    public void RegisterSyntaxTreeAction(
        Action<SyntaxTreeAnalysisContext> action)
    {
        this.Context.RegisterSyntaxTreeAction(
            _ =>
            {
                if (AnalyzerPathDetector.IsExcludePath(_.Tree))
                {
                    return;
                }

                action.Invoke(_);
            });
    }

    public void RegisterCompilationStartAction(
        Action<CompilationStartAnalysisContext> action)
    {
        this.Context.RegisterCompilationStartAction(_ =>
        {
            var interrupted = new CapsuledCompilationStartAnalysisContext(_);
            action.Invoke(interrupted);
        });
    }
}
```

基本的にはあるメソッドが呼ばれたら AnalysisContext の同じ名前のメソッドを呼び出し、引数に Action が渡されていたらファイルパス判定を先に行う処理を追加していま

す。このパターンでほとんど実装できますが、RegisterCompilationStartAction については引数を AnalysisContext と同じように扱うメソッドとなるので、別途 CapsuledCompilationStartAnalysisContextを定義して渡してあげる必要があります。CapsuledCompilationStartAnalysisContext の実装方針は CapsuledAnalysisContext と同じです。

メソッドのオーバーライドが終わったらファイルパスから解析対象を判定する処理を書いていきます。ファイルパスの判定にあたり、HogeHogeAnalysisContextから現在のコードのファイルパスを取得する必要があります。

取得については、SyntaxTree・ISymbolのどちらかが取れればパスの取得が可能です。

▼リスト 4.15　ファイルパスの判定処理

```
internal class AnalyzerPathDetector
{
    // Exclude にマッチしているが解析対象としたいファイルのパスの一部を指定
    private static string[] analyzerIncludePath = { "" };
    // 解析対象にしないパスの一部を指定する
    private static string[] analyzerExcludePath = { "" };

    internal static bool IsExcludePath(SyntaxTree tree)
    {
        return IsExcludePath(tree.FilePath);
    }

    internal static bool IsExcludePath(ISymbol symbol)
    {
        if (symbol.Locations.Length == 0)
        {
            return false;
        }

        var path = symbol.Locations.First().SourceTree.FilePath;
        return IsExcludePath(path);
    }

    private static bool IsExcludePath(string path, StyleCopSettings settings)
    {
        if (analyzerIncludePath.Any(pattern => IsPathMatch(path, pattern)))
        {
            return false;
        }

        return analyzerExcludePath.Any(patten => IsPathMatch(path, pattern))
    }

    private static bool IsPathMatch(string path, string searchText)
    {
        var localText = System.IO.Path.Combine(sratchText.Split('\\', '/'));
        return path.Contains(localText);
    }
}
```

設定関連については、配列ベタ書きではなく別途ファイルから読み込むようにしてもよいかと思います。基本的には解析対象外にするファイルパスの一部を指定していく形です。ここにはディレクトリ名などを追加していきます。ただパスの一部だと誤検知するパターンが存在するため、除外されていても解析対象とする設定についても用意しています。

IsPathMatchでは、OS ごとのパス区切り文字の差異を吸収するため、一度パスを分解

して再度結合することで現在の OS にあったパスを取得し直しています。OS のパス区切り文字は.NET Standard では取得できないため、Path.Combine を使って設定しています。

ここまでの実装をうまく組み合わせることで、解析対象外のファイルに存在するルール違反については表示されないようになります。

4.8 まとめ

かなり駆け足ですが、ここまで複数の IDE 環境の Unity プロジェクトで Roslyn Analyzers を導入する手法について書いてきました。全員が Visual Studio じゃなくて導入できなかった！ という方がもしいたら試していただけると幸いです。IDE 関係なしにこれまであまり触ってこなかった、という方もこれを機に触ってみるのはいかがでしょうか。

今後についてですが、文中で一瞬触れましたが Roslyn Analyzers の CI テストへの転用について考えています。IDE と CI で用いる規則が異なるというのは好ましくない状況ではあるので、.NET Core を用いてテストを行うだけの仕組みを作れないか検討中です。こちらについてはまた別の機会に記述出来ればよいなと考えています。長くなりましたがここまで読んでいただき本当にありがとうございました。読んでいただいた方の何らかの助けになっていれば幸いです。

第 5 章

バーコードリーダーになろう

Daisuke MAKIUCHI / @makki_d

近年、バーコードや QR コード[*1]でさまざまな情報を受け渡しする場面が増えてきました。専用の機械がなくてもスマートフォンで簡単に読み取ることができ、皆さんもよく利用しているのではないでしょうか。

そんな便利なスマートフォンも電池切れや家に忘れてしまいがち。そんなとき、機械に頼らずとも肉眼で読み取ることができればとても便利ですよね。

ただ、QR コードのような二次元コードは複雑で、慣れていないと読むのがかなり大変です。そこでまずは比較的単純なバーコードから読めるようになりましょう。

5.1　バーコードの規格について

ひと口にバーコードと言っても世の中にはたくさんの規格があります。この章では、おそらくもっともよく見かけるであろう、EAN（European Article Number）規格のバーコードを扱います。

EAN の E は European の略ですが、ISO/ECI 15420 として標準化もされている国際規格で、一般的な商品を表すバーコードとして世界中で使われています[*2]。日本国内でも JIS-X-0507 として標準化されていて[*3]、特に国コードが日本（49 または 45）のもは JAN（Japanese Article Number）と呼ぶこともあります。

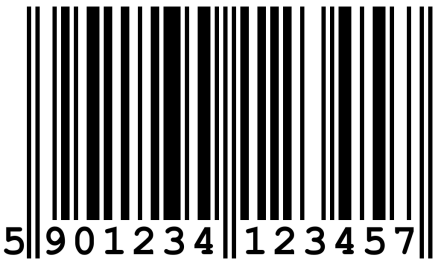
EAN には 13 桁のものと 8 桁のものがありますが、今回は 13 桁の **EAN-13** を読めるようになりましょう。

[*1] QR コードは株式会社デンソーウェーブの登録商標です

[*2] International Article Number とも呼ばれます

[*3] JIS 規格は「日本工業標準調査会」のウェブサイトで閲覧できます。
`http://www.jisc.go.jp/app/jis/general/GnrJISSearch.html`

5.2　バーコードの構造



▲図 5.1　EAN-13 バーコード

題材として図 5.1 にバーコードを用意しました[*4]。まずはこの構造を詳しく見ていきます。機械で読み取れなかった場合に備えて、バーコードの下に内容がそのまま書かれていることが多いですが、書かれていなかったり破損している場合のためにバーコード本体を読めなくてはなりません。今回は練習として、答え合わせに利用しながら読み進めてみてください。

EAN-13 の共通の固定パターンとして、両側に**標準ガードパターン**、そして中央に**中央ガードパターン**があります。両端と中央の長く書かれている 2 本線がそれぞれのガードパターンです。正確には、標準ガードパターンは「黒・白・黒」中央ガードパターンは「白・黒・白・黒・白」のパターンになっています。このガードパターンによって、バー 1 本あたりの幅が分かるようになっています。

中央ガードパターンを挟んで、左半分と右半分にそれぞれ 6 個の**シンボルキャラクタ**が描かれています。1 個のシンボルキャラクタはバー 7 本分の幅で、そのまま数値 1 桁を表しています。

ところで EAN-13 は 13 桁なので、これでは 1 桁足りません。実は先頭の 1 桁は、左側 6 個のシンボルキャラクタの中に隠されているのです。

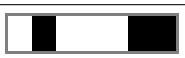
それではさっそく、シンボルキャラクタから数値を読み取ってみましょう。

5.3　シンボルキャラクタの読み方

表 5.1 がシンボルキャラクタと数値の対応表です。まずはこの表を覚えなくてはなりません。数値 1 つあたり 3 種類のパターンがあって大変に見えるかもしれませんが、よく見るとセット A を白黒反転するとセット C に、さらにそれを逆順にするとセット B になります。

[*4] 出典：https://en.wikipedia.org/wiki/International_Article_Number

▼表 5.1　数字セット

数値	セット A	セット B	セット C
0	 3:2:1:1	 1:1:2:3	 3:2:1:1
1	 2:2:2:1	 1:2:2:2	 2:2:2:1
2	 2:1:2:2	 2:2:1:2	 2:1:2:2
3	 1:4:1:1	 1:1:4:1	 1:4:1:1
4	 1:1:3:2	 2:3:1:1	 1:1:3:2
5	 1:2:3:1	 1:3:2:1	 1:2:3:1
6	 1:1:1:4	 4:1:1:1	 1:1:1:4
7	 1:3:1:2	 2:1:3:1	 1:3:1:2
8	 1:2:1:3	 3:1:2:1	 1:2:1:3
9	 3:1:1:2	 2:1:1:3	 3:1:1:2

左半分の 6 個はセット A またはセット B、右半分の 6 個はセット C が使われます。ガードパターンからだけでは逆さになっているかわからないのですが、逆さにして読んでしまうとこの表に無いパターンが途中で必ず出てくるので判断できます。

さっそく図 5.1 を読んでみましょう。左のガードパターンの次から、白 3 黒 1 白 1 黒 2 と並んでいるので、セット A の 9 になります。その次は白 1 黒 1 白 2 黒 3 なのでセット B の 0 ですね。その調子で進めていくと、左側は「A9-B0-B1-A2-A3-B4」になりました。中央ガードパターンを超えて右側に入ります。こからはセット C しか出てこないのでもっと簡単です。「1-2-3-4-5-7」になっています。読めましたか？

先頭の 1 桁は左側 6 個のシンボルキャラクタに隠されているのでした。まず左側の 6 個で、セット A・セット B のどちらが使われていたか書き出してみましょう。「A-B-B-A-A-B」ですね。このパターンを表 5.2 と照らし合わせると「5」になっています。これが先頭の 1 桁です。

こうして無事、「5-901234-123457」の 13 桁を読み取ることができました。

▼表 5.2　先頭桁の導出

数値	左半分の数字セット
0	A A A A A A
1	A A B A B B
2	A A B B A B
3	A A B B B A
4	A B A A B B
5	A B B A A B
6	A B B B A A
7	A B A B A B
8	A B A B B A
9	A B B A B A

5.4　チェックデジットの検証

EAN-13 では、13 桁のうち最後の 1 桁がチェックデジットになっていて、読み取りミスを検知できるようになっています。

計算方法は次のようになっています:

1. 先頭の桁はそのまま、2 桁目は 3 倍、3 桁目はそのまま、4 桁目は 3 倍、と交互に係数を掛けながら 12 桁を合計する
2. 1 で合計した値を 10 で割った余り（$mod10$）を計算する
3. 余りが 0 ならチェックデジットは 0、それ以外なら 10 から余りを引いた値がチェックデジット

この値と最後の 1 桁の値が一致していなければ、どこかで読み取りミスがあったことがわかります。

それでは「5901234123457」のチェックデジットを検証してみましょう。まず係数を掛けながら 12 桁を合計します。

$$5+9\times3+0+1\times3+2+3\times3+4+1\times3+2+3\times3+4+5\times3=83$$

$$83mod10=3$$

$$10-3=7$$

チェックデジットは 7 で、最後の 1 桁と一致しました。読み取りミスはなかったようです。

5.5 おわりに

EAN-13 バーコードから 13 桁の数値を読み取ることができました。身の回りのバーコードも読み取ってみてください。最後に、その数値がどんな意味を持っているかは皆さんへの宿題とします。

これで今日からあなたもバーコードリーダー！

バーコードの逆向き判定

表 5.1 の特定のセットに着目すると、同じセットの中では逆向きにしたときに重複しないようなパターンが巧妙に選ばれています。たとえば、セット A の 0「3:2:1:1」を逆にした「1:1:2:3」はセット A には含まれていません。

セット B とセット C はお互い逆向きにのパターンになっているため、逆向きに読んだ場合も対応するパターンがテーブルから必ず見つかってしまいます。一方でセット A を逆向きにしたセットはテーブルに存在せず、それが見つかったらバーコード自体が逆向きだと判断できます。表 5.2 を見ると A が含まれないパターンはひとつも無いため、バーコードが逆向きだった場合、必ず A の逆向きのパターンが見つかります。

ちなみに、EAN-13 の元になった規格、UPC-A は元々 12 桁で、セット A とセット C しか使われていませんでした。EAN-13 の策定過程で桁数を増やすとき、セット C を逆にしたセット B が新たに加えられましたが、逆向きかの判定ができなくならないように、注意深く導出テーブルが決められているわけです。

もし入れられる情報をさらに増やそうとして、セット A を逆順にしたセットを追加したり、先頭桁の導出テーブルに A が出てこないパターンを加えてしまうと、バーコードが逆向きかどうかの判断できなくなってしまいます。

第 6 章

Unity Timeline Tips 集

Junichi KIKUCHI @jukey17

6.1 はじめに

Unity2017 から正式版がスタートした Timeline 機能は、登場から 1 年あまり経過して日本語の解説記事も少しずつ増えてきました。しかし、まだ機能の説明やサンプルコードの紹介などにとどまる内容が多く、そこからさらに踏み込んだ内容の記事はあまり出揃っていません。そこで、この章では私の所属しているプロジェクトが実際にいざ Unity Timeline を導入してみようと検討した際に Unity エンジニア的な観点で開拓していった内容をいくつか紹介します。主に Unity 公式が用意しているサンプルコードプロジェクトの DefaultPlayables を触ったことがある人向けの内容になっています。Unity Timeline の導入を考えている方の参考に少しでもなれば幸いです。

※この章で紹介している機能やコードは Unity2018.3.0b1 で動作確認をしています。それ以外のバージョンでは動作しないケースがあるかもしれませんのでご了承ください。

6.2 Tips.1 [PlayableAsset] TimelineAsset にキャストせずにトラック情報を取り出す

Timeline の機能を使おうとするとまずややこしいのが、Timeline の機能と Playables の機能です。`TimelineAsset`の再生に使う `PlayableDirector`は Timeline ではなく Playables の機能になるため `PlayableDirector`から直接 `TimelineAsset`や `TrackAsset`の情報にアクセスすることができません。

一応、Timeline の機能として使う際は `PlayableDirector.playableAsset`に `TimelineAsset`を設定しているはずなので、`PlayableDirector.playableAsset`を `Timelin`

eAssetにダウンキャストすることで Timeline の情報を取り出すことができますが、それをしなくても PlayableAssetのプロパティからトラックの情報にアクセスする方法があります。

PlayableAsset.outputsにトラックの情報が格納されているのです

▼リスト 6.1　PlayableAsset.outputs と TimelineAsset.GetOutputTracks はほぼ同等

```
var timelineAsset = playableDirector.playableAsset as TimelineAsset;
foreach (var track in timelineAsset.GetOutputTracks())
{
    Debug.Log(track);
}

foreach (var binding in playableDirector.playableAsset.outputs)
{
    // TimelineAsset.GetOutputTracks で列挙した TrackAsset と同じものになる
    UnityEngine.Debug.Log(binding.sourceObject);
    // 上記の TrackAsset の名前
    UnityEngine.Debug.Log(binding.streamName);
    // 上記の TrackAsset に Bind されるオブジェクトの型情報
    // ※ TrackBindingTypeAttribute で指定されている型
    UnityEngine.Debug.Log(binding.outputTargetType);
}
```

TrackAssetから TrackBindingTypeAttributeを直接取得してくるコードはないので、PlayableBindingから取り出せる情報のほうが単純に TrackAssetを見るよりも詳しかったりします。

Unity の機能を紹介してくれるブログで有名なテラシュールブログさんの Timeline の紹介記事でも簡単に紹介されていますが、これは Track に対して動的にオブジェクトを Bind する際に便利です。

▼リスト 6.2　Timeline の名前空間を使わずに動的 Bind

```
public class DynamicBinder : MonoBehaviour
{
    [SerializeField] PlayableDirector director;

    // 指定の名前のトラックに任意のオブジェクトを Bind する
    public void Bind(string trackName, Object target)
    {
        var binding = director.playableAsset.output
            .FirstOrDefault(o => o.steamName == trackName);
        if (binding != null)
        {
            director.SetGenericBinding(binding.sourceObject, target);
        }
    }
}
```

余計な変換を噛まさずに処理できるものは処理していきたいので、こういったものはどんどん活用していきましょう。

6.3 Tips.2 [PlayableBehaviour] PlayableDirector の参照を取得する

Timeline の機能で独自に拡張したトラックに対応した処理（振る舞い）を作るためには、PlayableBehaviourを継承した Mixer や Behaviour といった独自クラスも合わせて作らなければなりません。ここは各クリップの情報を受け取ってアレコレ処理をする部分になるので、シーン上にある情報にもアクセスできるようにしたいです。

対象の機能をシングルトンパターンで実装して Mixer/Behaviour から直接アクセスしに行くのもアリですが、PlayableDirectorを通して Playable の機能を使って再生制御されているはずなので、Playable の機能を使って上手くシーン上の参照が取れるようにしたいです。

PlayableBehaviourには自身 (ScriptPlayable) の入出力を制御してくれる PlayableGraphを取得するためのメソッドが用意されています。Timeline の機能の場合この PlayableGraphから取得できる Resolver が実は PlayableDirectorとなるので、そこから参照を取ってくるようにしてみます。

具体的には次のコードで取得が可能です。

▼リスト 6.3 PlayableBehaviour.OnPlayableCreate のタイミングで PlayableDirector の参照を保持する

```
// サンプル用独自トラック
public class SampleTrack : TrackAsset
{
    public override Playable CreateTrackMixer(
        PlayableGraph graph, GameObject go, int inputCount)
    {
        // Track 側で独自の Mixer 生成をしておく
        return ScriptPlayable<SampleMixer>.Create(graph, inputCount);
    }
}

//  SampleTrack の Mixer
public class SampleMixer : PlayableBehaviour
{
    public override void OnPlayableCreate(Playable playable)
    {
        // Resolver が PlayableDirector になっている
        var director = playable.GetGraph().GetResolver() as PlayableDirector;
    }
}
```

このようにして生成時に自身の親玉となる PlayableDirectorの参照を保持しておくことで、ProcessFrameなどの何度も呼び出されるメソッドの中で再生中の全体の情報にアクセスするといったことが容易にできるようになります。

たとえば PlayableDirectorをアタッチしている GameObjectに独自のコンポーネントをセットでアタッチしておく、というルールを作っておけば PlayableDirectorからさらにその独自管理コンポーネントを取得して Mixer に参照を持たせるという芸当も可

能になります。

▼リスト 6.4　生成時に独自コンポーネントの参照を保持する

```
//  SampleTrack の Mixer
public class SampleMixer : PlayableBehaviour
{
    TexturePool texturePool;

    public override void OnPlayableCreate(Playable playable)
    {
        var director = playable.GetGraph().GetResolver() as PlayableDirector;
        // テクスチャ管理機構を保持しておく
        texturePool = director.gameObject.GetComponent<TexturePool>();
    }

    public override void ProcessFrame(
        Playable playable, FrameData info, object playerData)
    {
        var sample = playerData as SampleComponent;
        var inputCount = playable.GetInputCount();
        for(var i = 0; i i < inputCount; i++)
        {
            var weight = playable.GetInputWeight(i);
            if (Mathf.Approximately(weight, 0))
            {
                continue;
            }

            // クリップ (Behaviour) に指定されているテクスチャを Bind 対象にセット
            var input = (ScriptPlayable<SampleBehaviour>)playable.GetInput(i);
            var behaviour = input.GetBehaviour();
            var texture = texturePool.FindTexture(behaviour.TextureName);
            sample.SetTexture(texture);
            break;
        }
    }
}
```

上記の例では、クリップの情報を元に独自のテクスチャ管理機構からテクスチャをもらってきてそれを Track に Bind されているコンポーネントに対して設定する。といった処理を書いてみました。テクスチャが個別に AssetBundle 化されていたりすると、クリップに直接テクスチャを埋め込むわけにもいかなくなるのでこのようにして外部の管理機構からテクスチャをもらってきて設定するといった流れを取ることになります。

Timeline のトラックやクリップ、ミキサーなどはランタイム動作の際は Timeline の機能の中に埋まってしまっていて Timeline の機能外へのアクセスが中々に面倒なことになっているので、このようにしてアクセスできる口を作っておくと拡張がしやすくなります。

6.4　Tips.3 [PlayableBehaviour] 各クリップの時間に関する情報を取得する

Mixer を独自に拡張していると Mixer の `ProcessFrame`メソッド内で各クリップの長さや、そのクリップ内でどこまで再生されたかの再生時間などが欲しくなるケースが出てきます。各クリップの長さは `TimelineClip.duration`で取ってこれますが、`Timeline`

Clipは Timeline の機能であるのに対して PlayableBehaviourは Playables の機能であるため ProcessFrameなどの用意されているメソッド内では直接 TimelineClipの参照は取ってくることができません。

DefaultPlayables のサンプルコードでも、クリップごとの Behaviour を定義してそこに独自パラメータのプロパティを生やしておき Mixer 内でその Behaviour の参照を掘り起こして処理しているだけなので、TimelineClip の情報は勿論取ってきていません。

▼リスト 6.5　DefaultPlayables の例

```
[Serializable]
public class SampleBehaviour : PlayableBehaviour
{
    // クリップごとにもつ独自パラメータ
    public int integerField = 1;
}

public class SampleClip : PlayableAsset, ITimelineClipAsset
{
    // DefaultPlayables では Editor 拡張で Behaviour を Inspector から編集できるようにしている
    // 確かに Playable の機能を使って Mixer 内から直接取り出せるのでそのほうが便利だが…
    // そうすると Timeline 機能としてのクリップの情報が取ってこれない（悲）
    public SampleBehaviour template = new SampleBehaviour();

    public override Playable CreatePlayable(PlayableGraph graph, GameObject owner)
    {
        return ScriptPlayable<SampleBehaviour>.Create(graph, template);
    }
}

public class SampleMixer : PlayableBehaviour
{
    public override void ProcessFrame(
        Playable playable, FrameData info, object playerData)
    {
        var inputCount = playable.GetInputCount();
        var value = 0.0f;
        for(var i = 0; i i < inputCount; i++)
        {
            // Input から取り出した Behaviour のパラメータを使ってクリップ毎の情報を処理している
            var input = (ScriptPlayable<SampleBehaviour>)playable.GetInput(i);
            var weight = playable.GetInputWeight(i);
            var behaviour = input.GetBehaviour();
            value += (behaviour.integerField * weight);
        }

        var sample = playerData as SampleComponent;
        sample.SetValue(value);
    }
}
```

ここで上記サンプルコードの playable.GetInput()の処理をしていることに注目しましょう。playable.GetInput()して取得した ScriptPlayable<T>がクリップ単位のもの (TimelineClipと 1 対 1 の関係) であることがわかります。

実は ScriptPlayable<T>.GetDuration()メソッドを使うことで TimelineClip.durationと同等の内容を取得することができるのです。さらに ScriptPlayable<T>.GetTime()メソッドではそのクリップ内でどこまで再生されたかの再生時間も取ることができます。これはとても便利です。

▼リスト 6.6　クリップ単位の Playable から時間の情報を取り出す

```
public class SampleMixer : PlayableBehaviour
{
    public override void ProcessFrame(Playable playable, FrameData info, object playerData)
    {
        var inputCount = playable.GetInputCount();
        for(var i = 0; i i < inputCount; i++)
        {
            var input = (ScriptPlayable<SampleBehaviour>)playable.GetInput(i);
            // クリップの中でどこまで再生しているのかの再生時間
            var currentTime = input.GetTime();
            // クリップ自体の長さ
            var duration = input.GetDuration();
            // クリップ内でどこまで再生が進んだのかの割合（進捗率）を求めることも可能
            var progress = currentTime / duration;
        }
    }
}
```

上記のような形で、クリップの長さやそのクリップ内の現在の再生時間を取得することでそのクリップの中でどれだけ再生が進んだかの割合値も求めることができます。前述の Tips で親玉の `PlayableDirector`の参照も取ってこれるので、組み合わせで色々と応用ができるようになるはずです。是非とも活用していきましょう。

6.5　Tips.4 [AssetBundle] TimelineAsset を AssetBundle 化する

オンラインアップデートが当たり前な昨今のゲームでは、Unity で開発する上で扱うアセットの AssetBundle 化は必須と言っても過言ではありません。勿論、Timeline の機能として扱う `TimelineAsset`も例外ではありません。

とは言っても `TimelineAsset`をそのまま AssetBundle 化するだけであれば何も難しいことはありません。その他のアセットの AssetBundle 化と内容は変わりません。

▼リスト 6.7　TimelineAsset の AssetBundle 化

```
public void Build()
{
    // 指定のディレクトリにある TimelineAsset をすべて Build 対象にする
    var targetPath = "Assets/Resources/TimelineAssets";
    var builds = AssetDatabase.FindAssets("t:TimelineAsset", new[] { targetPath })
        .Select(AssetDatabase.GUIDToAssetPath)
        .Select(CreateAssetBundleBuild)
        .ToArray();
    // Options や BuildTarget はよしなに書き換えてください
    var options = BuildAssetBundleOptions.None;
    var platforms = new[] { BuildTarget.iOS, BuildTarget.Android };
    var outputPath = "Assets/StreamingAssets/AssetBundles/TimelineAssets";

    foreach (var platform in platforms)
    {
        BuildPipeline.BuildAssetBundles(outputPath, builds, options, platform);
    }
}

AssetBundleBuild CreateAssetBundleBuild(string path)
```

```
{
    // サンプルなので最低限必要になるものだけ埋めてます
    // 必要に応じて Variant や AddressableName を使ってください
    return new AssetBundleBuild
    {
        assetBundleName = Path.GetFileName(path),
        assetNames = new []{ path },
        assetBundleVariant = "",
        addressableNames = new string[0],
    };
}
```

ただ単に AssetBundle 化するだけであれば上記のように一般的な AssetBundle のビルド処理を作れば終わりになります。

しかし、この方法だと中のクリップで指定する `AnimationClip`や `AudioClip`などが増えてくると、この AssetBundle に内包されていきドンドン容量が増えていきます。また、AssetBundle の仕様上の問題で内部的に使われているアセットに重複があってもそれぞれの AssetBundle にそのアセットを含めてしまうため無駄ができてしまいます。関係するアセットを `TimelineAsset`と一緒に明示的に全部 AssetBundle 化することで Dependecies に含まれている依存関係をよい感じにしてくれるのですが、ある程度の規模の開発になってくると、各種単体のアセット制作と Timeline の演出制作とで制作パイプラインも変わってくるので単純にまとめてビルドというのも難しくなります。※ディレクトリ構成が違ったりアセットの制作タイミングがずれたりするため

ここは最終納品物として `TimelineAsset`を指定したらそれに使われているアセットもまとめてビルドしてくれるような仕組みにしたいです。使われるアセットは `AnimationTrack`や `AudioTrack`から各種クリップごとの `PlayableAsset`を引っ張り出すことができれば合わせて自動的に AssetBundle 化できそうです。

▼リスト 6.8　TimelineAsset に内包されているアセットを取り出す

```
// TimelineAsset の中身に使っているアセットのパスをすべて取得
string[] GetIncludeAssetPaths(TimelineAsset timelineAsset)
{
    var outputPaths = new List<string>();
    foreach (var track in timelineAsset.GetRootTracks())
    {
        GetIncludeAssetPathsRecursive(track, outputPaths);
    }

    return outputPaths.ToArray();
}

// 再帰的にトラックの中身のアセットを探す
void GetIncludeAssetPathsRecursive(TrackAsset track, List<string> outputPaths)
{
    foreach (var clip in track.GetClips())
    {
        // デフォルトの機能では AnimationClip/AudioClip/Prefab の 3 種類
        UnityEngine.Object asset = null;
        if (track is AnimationTrack)
        {
            var playableAsset = clip.asset as AnimationPlayableAsset;
            asset = playableAsset.clip;
        }
        else if (track is AudioTrack)
```

```
        {
            var playableAsset = clip.asset as AudioPlayableAsset;
            asset = playableAsset.clip;
        }
        else if (track is ControlTrack)
        {
            var playableAsset = clip.asset as ControlPlayableAsset;
            asset = playableAsset.prefabGameObject;
        }
        // 独自トラックでアセットを扱っている場合はここに if 文を足していく
        else if (track is SampleTrack)
        {
            var playableAsset = clips.asset as SamplePlayableAsset;
            asset = playableAsset.SampleData;
        }

        if (asset != null)
        {
            var path = AssetDatabase.GetAssetPath(asset);
            outputPaths.Add(path);
        }
    }

    // 子トラックへ再帰的に処理していく
    foreach (var childTrack in track.GetChildTracks())
    {
        GetIncludeAssetPathsRecursive(childTrack, outputPaths);
    }
}
```

このようにして TimelineAssetの中身を調べることで内包しているアセットのパスを取得することができます。あとはこの処理をビルド対象となるすべての TimelineAsset に通して重複を消し、TimelineAssetと一緒にビルドすれば別々に AssetBundle 化することができます。

▼リスト 6.9　依存アセットも含めた AssetBundle 化

```
var targetPath = "Assets/Resources/TimelineAssets";
var timelinePaths = AssetDatabase.FindAssets("t:TimelineAsset", new[] { targetPath })
    .Select(AssetDatabase.GUIDToAssetPath)
    .ToArray();
// 重複を除いた上で内包するアセットのパスも取得してビルド対象に含める
var includeAssetPaths = timelinePaths
    .Select(AssetDatabase.LoadAssetAtPath<TimelineAsset>)
    .SelectMany(GetIncludeAssetPaths)
    .Distinct()
    .ToArray();
var builds = includeAssetPaths.Concat(timelinePaths)
    .Select(CreateAssetBundleBuild)
    .ToArray();
```

6.6　Tips.5 [SelectionManager] 選択中のトラック/クリップを取得・設定する

Unity Timeline の API は内部的には使用されているものの外部に公開されていないものも多く、特に TimelineWindowに関するものはまだほとんどと言っていいほど internalで定義されているのが実情です。

実は、この項目のタイトルにもなっている選択中のトラックやクリップの情報も内部的には API が存在します。この Tips ではこれらの `internal`な API を C#の Reflection を使ってアクセスできるようにしてしまおうという内容です。

Timeline に関する選択中のオブジェクトを管理する機構として `SelectionManager`というクラスが `namespace UnityEditor.Timeline`内に定義されています。

`UnityEditor.Selection`クラスからも情報は取れますが Timeline 機能以外の選択オブジェクトを考慮したり、Timeline 機能内での連携も考慮しなければならないため `SelectionManager`経由で操作しましょう。

まずはじめに、`SelectionManager`の型情報を取得します

※ Reflection の機能を使っているので `using System.Reflection;`を忘れずに。

▼リスト 6.10　SelectionManager の型情報を取得する

```
var assembly = Assembly.Load("UnityEditor.Timeline");
var type = assembly.GetType("UnityEditor.Timeline.SelectionManager");
```

型情報が取れたらそこからさらにメソッド情報を取得していきます。

▼リスト 6.11　SelectionManager のメソッド情報を取得する

```
// 選択中の TimelineClip/TrackAsset をすべて取得する
var selectedClipsMethod = type.GetMethod("SelectedClips",
    BindingFlags.Public | BindingFlags.Static);
var selectedTracksMethod = type.GetMethod("SelectedTracks",
    BindingFlags.Public | BindingFlags.Static);

// 任意の TimelineClip/TrackAsset を選択状態する
var addClipMethod = type.GetMethod("Add",
    BindingFlags.Public | BindingFlags.Static, null,
    new [] { typeof(TimelineClip) }, null);
var addTrackMethod = type.GetMethod("Add",
    BindingFlags.Public | BindingFlags.Static, null,
    new [] { typeof(TrackAsset) }, null);

// 任意の選択されている TimelineClip/TrackAsset を選択解除状態にする
var removeClipMethod = type.GetMethod("Remove",
    BindingFlags.Public | BindingFlags.Static, null,
    new [] { typeof(TimelineClip) }, null);
var removeTrackMethod = type.GetMethod("Remove",
    BindingFlags.Public | BindingFlags.Static, null,
    new [] { typeof(TrackAsset) }, null);
```

用途が分かりやすいものをいくつかピックアップしてみました。Timeline の編集で扱うデータは `TimelineClip`と `TrackAsset`なので、それらに関するメソッドが並んでいます。Add/Remove メソッドに関しては引数違いの同名メソッドになっているので、引数の型を明示的に指定してあげています。

※ Reflection についてはここでは詳しく解説しません。各自調べてください。

最後に上記の掘り起こしたメソッドを使った独自の拡張も一緒に紹介します。

▼リスト 6.12　SelectionManager クラスを活用した拡張

```
// 選択されている TimelineClip の長さを一律で設定する
public void SetDurationToSelectedClips(float duration)
{
    var selectedClips = (IEnumerable<TimelineClip>) selectedClipsMethod
        .Invoke(null, null);
    foreach (var clip in selectedClips)
    {
        clip.duration = duration;
    }
}

// TTrack の型の TrackAsset だけを選択状態にする
public void FilteringSelectedTracks<TTrack>()
    where TTrack : TrackAsset
{
    var selectedTracks = (IEnumerable<TrackAsset>) selectedTracksMethod
        .Invoke(null, null);
    foreach (var remove in selectedTracks.Where(track => !(track is TTrack))
    {
        removeTrackMethod.Invoke(null, new object[] { remove });
    }
}
```

このようにして内部の機能を掘り起こすことで、通常では難しい/出来ないような拡張もできるようになります。次にもうひとつ、内部 API を掘り起こした拡張を紹介しましょう。

6.7 Tips.6 [TimelineWindow] スクロール位置を制御する

Tips.5 では `internal`な静的クラス・メソッドを取得してくるだけの単純な内容でしたが、この Tips ではもう少し踏み込んで複数の `internal`なクラスを組み合わせる内容です。

Timeline の編集をする際に一番操作すると言っても過言ではない `TimelineWindow`のスクロール。既存の動作だけでは痒いところに手が届かないと感じる方も多いと思うので、ここはエンジニアの力でサポートしてあげたいところです。

`namespace UnityEditor.Timeline`内に `WindowState`という `internal`なクラスがあり、`TimelineWindow`の状態の管理はこのクラスの中で行われています。このクラスからスクロールの情報を引っ張り出して制御していきます。

次に再生バーの位置までスクロールさせるサンプルコードを載せます。一気に載せるので少し長いです。

▼リスト 6.13　ScrollToCurrentTime

```
using System.Reflection;
using UnityEditor.Timeline;

// TimelineWindow のスクロール位置を再生バーの位置まで移動させるメソッド
void ScrollToCurrentTime()
{
    // UnityEditor.Timeline.dll のアセンブリ情報を取得する
    var assembly = Assembly.Load("UnityEditor.Timeline");

    // TimelineWindow と WindowState の型情報を取得する
```

```
    var timelineWindowType = assembly.GetType("UnityEditor.Timeline.TimelineWindow");
    var windowStateType = assembly.GetType("UnityEditor.Timeline.WindowState");

    // TimelineWindow のシングルトンインスタンスの取得を試みる
    // TimelineWindow が開かれていたら取得できる
    var timelineWindowInstanceProp = timelineWindowType
        .GetProperty("instance", BindingFlags.Public | BindingFlags.Static);
    var timelineWindowInstance = timelineWindowInstanceProp.GetValue(null, null);
    if (timelineWindowInstance == null)
    {
        Debug.LogWarning("TimelineWindow が開かれていません");
        return;
    }

    // TimelineWindow のインスタンスから WindowState インスタンスを取得する
    var stateProp = timelineWindowType
        .GetProperty("state", BindingFlags.Public | BindingFlags.Instance);
    var windowState = stateProp.GetValue(timelineWindowInstance, null);

    // TimelineWindow の横軸の中央値を取得する
    var timeAreaRectProp = windowStateType
        .GetProperty("timeAreaRect", BindingFlags.Public | BindingFlags.Instance);
    var timeAreaRect = (Rect) timeAreaRectProp.GetValue(windowState, null);
    var timeAreaCenterX = timeAreaRect.width * 0.5f;

    // 現在の再生時間から再生バーの X 座標を算出する
    var timeToScreenSpacePixelMethod = windowStateType.GetMethod(
        "TimeToScreenSpacePixel", BindingFlags.Public | BindingFlags.Instance,
        null, new[] { typeof(double) }, null);
    var currentTime = TimelineEditor.masterDirector.time;
    var cursorPosX = (float) timeToScreenSpacePixelMethod
        .Invoke(windowState, new object[] { currentTime });

    // TimelineWindow のスクロール位置を再生バーが中央に来るように補正する
    var timeAreaTranslationProp = windowStateType.GetProperty(
        "timeAreaTranslation", BindingFlags.Public | BindingFlags.Instance);
    var timeAreaTranslation = (Vector2) timeAreaTranslationProp
        .GetValue(windowState, null);
    timeAreaTranslation.x -= cursorPosX;
    timeAreaTranslation.x += timeAreaCenterX;

    // 補正したスクロール位置を TimelineWindow に設定する ※ Scale は変更しないのでそのまま値を渡し直
す
    var timeAreaScaleProp = windowStateType.GetProperty(
        "timeAreaScale", BindingFlags.Public | BindingFlags.Instance);
    var timeAreaScale = (Vector2) timeAreaScaleProp.GetValue(windowState, null);
    var setTimeAreaTransformMethod = windowStateType.GetMethod(
        "SetTimeAreaTransform", BindingFlags.Public | BindingFlags.Instance, null,
        new[] { typeof(Vector2), typeof(Vector2) }, null);
    setTimeAreaTransformMethod.Invoke(windowState,
        new object[] { timeAreaTranslation, timeAreaScale });

    // TimelineWindow の再描画をする
    TimelineEditor.Refresh(RefreshReason.WindowNeedsRedraw);
}
```

上記のコードを実行すると、再生バーがちょうど TimelimeWindowの真ん中に来るようにスクロールされます。

ある程度長い TimelineAssetを扱っていると、再生バーとスクロール位置が大きくずれて面倒なことになるケースが多発します。上記のようにボタンひとつで再生バーの位置までスクロールできるようになるだけでかなり便利になると思います。

またこれを応用すれば、縦のスクロールを操ったり、再生中にスクロールを追従させるというようなこともできるようになります。

6.8 最後に

いかがでしたでしょうか。今回紹介したものは必要にかられて Timeline の機能を手探りで開拓していったものの一部で、まだ開拓できていない内容も沢山あります。私が Timeline の機能を触り始めた Unity2017 から Timeline の機能は少しずつですが確実に進化していっています。もしかしたらここで紹介しているコードも次のバージョンからは Legacy になっていたり、そもそも呼び出せなくなってしまったりしているかもしれません。※特に Reflection で掘り起こしているところは容赦なくリファクタリングされていくので要注意です。

ただ、このようにしてできたばかりの機能を開拓していくのはエンジニアとしては結構やりごたえがあって楽しいです。皆さんも、是非とも一緒に Unity の新機能を開拓して情報を出していきましょう！ ありがとうございました。

第 7 章

物理ベースレンダラーを Rust 実装して、ちょっと高速化した話

Sho HOSODA / @gam0022

本章では、筆者自作の物理ベースレンダラを題材にして、**パストレーシング**の高速化の手法を紹介します。また技術書典 3 の『KLab Tech Book Vol. 1』[*1]の第一章『物理ベースレンダラーを Rust 実装して、表紙絵をレンダリングした話』の続編です。

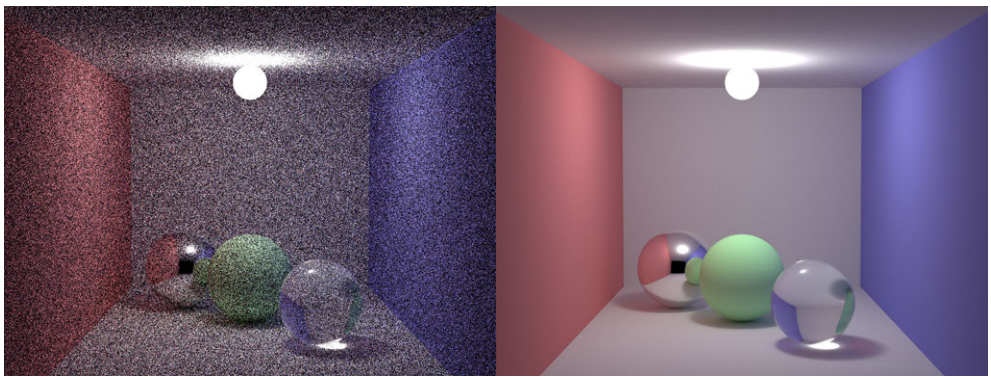
7.1 パストレーシングと計算時間

パストレーシングを簡単にいうと、現実世界の光の振る舞いを簡略化してシミュレートすることで、写実的なレンダリングができる 3D の描画手法です。まず 3DCG の描画方法は**レイトレーシング法**と**ラスタライズ法**の大きく 2 つに分類できます。レイトレーシング法（以下、レイトレ）ではカメラからレイを飛ばし、レイがシーン中の物体に衝突したらシーンを描画し、衝突しなかったら背景を描画します。そして物体表面がどのくらい照らされているかを計算し、色付けをしていきます。この色付け処理を**シェーディング**と呼びます。パストレーシングは、このシェーディングの処理として、光の伝達・挙動を記述したレンダリング方程式を用います。レンダリング方程式を正確に計算することで大域照明を考慮した写実的な描画ができますが、実際のシーンのレンダリングにおいて解析的に正確な計算をすることは非現実的です。そこで、モンテカルロ積分（乱数を使ったシミュレーションを繰り返して数値的に積分を計算する手法）を用いてレンダリング方程式を解きます。以上をまとめると、**パストレーシングはレイトレに分類される手法で、モンテカルロ積分を用いてレンダリング方程式を計算することでシェーディングを行います**。

モンテカルロ積分では**サンプリング数**（シミュレーションの回数）が計算時間と比例します。図 7.1 は、サンプリング数による描画結果と計算時間の違いを示したものです。十

[*1] http://klabgames.tech.blog.jp.klab.com/techbook/KLab-Tech-Book-Vol-1-1.2-ebook.pdf

分な計算時間があればノイズの無いきれいな描画結果（右）となりますが、計算時間が限られた状況では、サンプリング数が不足してノイズだらけの描画結果（左）になってしまいます。



▲図 7.1　パストレーシングの描画結果。（左）は 1 サブピクセルあたり 10 サンプリング、Intel Core i7-6700K による計算時間は 20.7 秒。（右）は 10000 サンプリング、5 時間 45 分 18.6 秒

高品質なシーンを限られた時間内で描画しなければならないという場面は往々にしてあるかと思います。品質を保ちつつ、計算時間を短縮するにはどうすればよいでしょうか。以降ではパストレーシングの高速化の手法として、次に挙げる 5 つの手法を紹介していきます。

- BVH による衝突判定の高速化
- 式を整理して不要な計算を省くことによる高速化
- Next Event Estimation（NEE）によるサンプリングの効率化
- デノイズによるノイズの除去
- コンパイラの最適化を利用

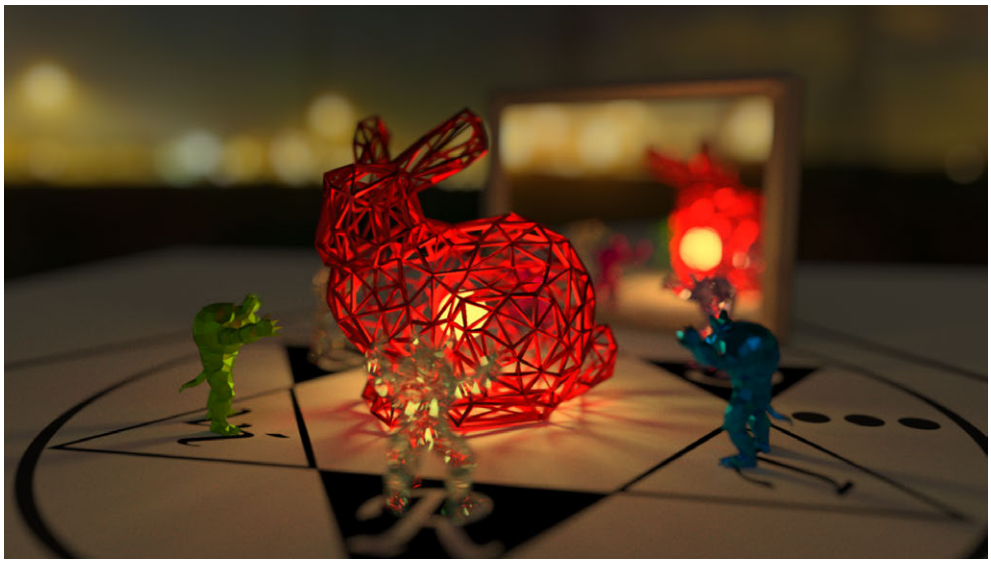
7.2　自作レンダラーの紹介

高速化の話を始める前に、本章で題材とする筆者自作の **Hanamaru レンダラー**[*2]を簡単に紹介します。Hanamaru レンダラーはパストレーシングによる物理ベースレンダラーであり、図 7.2 はその描画結果です。Rust で実装されていて、次のとおりレンダラーとしての基本機能が揃っています。

- パストレーシング（BSDF による重点的サンプリングあり）

[*2] `https://github.com/gam0022/hanamaru-renderer`

- シーン上のオブジェクトとして Sphere, Polygon Mesh, AABB に対応
- 並列処理による高速化
- 対応マテリアル
  - 完全拡散反射
  - 完全鏡面反射
  - 金属面（GGX の法線分布モデル）
  - ガラス面（GGX の法線分布モデル）
- Image Based Lighting（IBL）
- テクスチャによる albedo / roughness / emission の指定
- 薄レンズモデルによる被写界深度（レンズのピンぼけ）



▲図 7.2　自作の Hanamru レンダラーによる描画結果

Hanamaru レンダラーの作成過程や各機能について KLab Tech Book Vol. 1 と筆者のブログで詳しく紹介しています。興味がありましたら、そちらもぜひご覧いただければ嬉しいです。

レイトレ合宿 5 に参加して、Rust でパストレーシングを実装しました！

`https://gam0022.net/blog/2017/10/02/rtcamp5/`

レイトレ合宿 6 参加報告 前編（準備編）

`https://gam0022.net/blog/2018/09/18/rtcamp6-part1/`

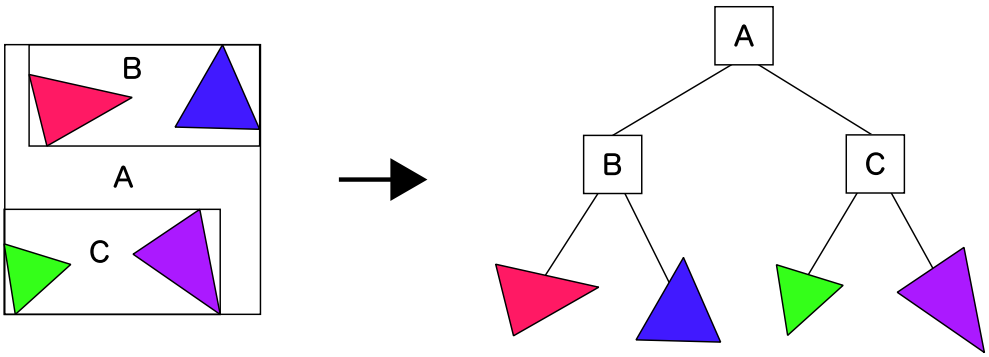
7.3 BVH による衝突判定の高速化

衝突判定はレイトレの実行時間のほとんどを占めるため、これを高速化するのは非常に効果的です。衝突判定の高速化としては **BVH**（Bounding Volume Hierarchy）を実装しました。シーン内のすべてのオブジェクトを総当りで衝突判定せずに、空間分割することで衝突判定の回数を減らします。BVH は前回の記事でも紹介しましたが、とても効果が高い高速化手法であるため、改めて紹介します。

BVH による高速化の効果はシーンに依存し、シーンが大規模になるほど高速化の効果は大きくなります。シーンに含まれるポリゴンの数を n としたとき、BVH なしの総当たりで衝突判定する場合、計算量のオーダーは $O(n)$ となります。一方、BVH を用いる場合は $O(log\ n)$ になります。つまり、シーンに含まれるポリゴンの数が多い大規模なシーンになるほど BVH は効果的です。

BVH はオブジェクトのバウンディングボックスを入れ子にした二分木の構造です。BVH の例を図 7.3 に示します。A が根ノードかつ内部ノードで、B と C は葉ノードです。

BVH を構築する手法はさまざまですが、今回はバウンディングボックスの最長辺を軸に選んで要素数で 2 分割する実装にしました。まずは全オブジェクトを内包するバウンディングボックスをつくり、それを根ノードとします。次に、バウンディングボックスの一番長い辺を選んで、その辺を軸として内包するオブジェクトの数が均等になるように二等分します。二等分されたオブジェクトそれぞれでノードを作り、さらに分割をする手順を再帰的に繰り返し、ノードが持つオブジェクト数が 4 以下になったら、そのノードを葉ノードとして分割を終了します。



▲図 7.3　BVH の例。オブジェクトのバウンディングボックスを入れ子にした二分木の構造

BVH の構築ができたら、いよいよ衝突判定です。BVH との衝突判定はよくある二分木に対する走査と同じです。まずは根ノードのバウンディングボックスと衝突判定を行い、もし交差していればその子ノードと衝突判定を行い、交差していなければ走査を打ち切る、という手順を再帰的に繰り返すと、いずれは交差するオブジェクトを持つ葉ノード

に到達します。あとは葉ノードが持つオブジェクトに対して総当りで衝突判定を行うだけです。単純な BVH 構築の方法でしたが、総当りと比較すると十分に高速化できました。

シーンを構成するオブジェクトとして、ポリゴンメッシュ、球体、立方体の 3 種類に対応しました。シーンを構成するオブジェクトに対する BVH だけでなく、ポリゴンメッシュを構成するポリゴンに対しても BVH を構築しました。

7.4 式を整理して不要な計算を省く高速化

式を整理して不要な計算を省くことによる高速化を紹介します。「7.1 パストレーシングと計算時間」の繰り返しとなりますが、パストレーシングはレンダリング方程式をモンテカルロ積分によって解く手法でしたね。レンダリング方程式は、光の伝達・挙動を記述した方程式であり、次の式で定義されます。

$$L_0(x, \omega_0) = L_e + \int_\Omega f_s(x, \omega_i, \omega_0) L_i \cos\theta d\omega$$

L_0 は、ある点 x から ω_0 方向に放出される放射輝度（=レイのもつ光のエネルギー）を表します。L_e は、 x が光源の場合に ω_0 方向に放出される放射輝度となります。$f_s(x, \omega_i, \omega_0)$ は x において ω_i と ω_o に対する **BSDF**（Bidirectional Scattering Distribution Function）です。BSDF は、簡単にいうと「ある方向から来た光が別の方向にどれくらい反射（透過）するか」を表しています。

$\int_\Omega f_s(x, \omega_i, \omega_0) L_i \cos\theta d\omega$ は半球上の積分となっており、x に入射するあらゆる方向からの放射輝度を集めるという意味です。つまり、ある点 x の放射輝度が、 x 自身が放つ光と周囲から集めた光のエネルギーの総和であることを示しています。

レンダリング方程式の積分の被積分関数は再帰的で高次元かつ非連続的であるため、解析的に解くことは困難です。そこで、パストレーシングではモンテカルロ積分によって数値的にレンダリング方程式を解きます。レンダリング方程式をモンテカルロ積分をつかって書き直すと次の式となります。

$$\hat{L}_0(x, \omega_0) = L_e + \frac{1}{N} \sum_{i=1}^{N} \frac{f_s(x, \omega_i, \omega_0) L_i \cos\theta}{p(\omega_i)}$$

$\hat{L}_0$ は L_0 の推定値を表します。N はモンテカルロ積分のサンプリング数であり、$p(\omega_i)$ は ω_i 方向を選ぶ確率密度関数です。

以上を踏まえると、モンテカルロ積分において、ある 1 サンプリングの重み w_i は次の式となります。

$$w_i = \frac{f_s(x, \omega_i, \omega_0) \cos\theta}{p(\omega_i)}$$

愚直に w_i を計算する場合、BSDF である $f_s(x, \omega_i, \omega_0)$ 、確率密度関数である $p(\omega_i)$ 、$\cos\theta$ という 3 つの項の計算が必要になります。

しかし、特定のマテリアルの場合にはこのような計算を省略できます。たとえば完全拡散面において cos 項に応じた重点的サンプリングをした場合を考えてみましょう。

まず $f_s(x, \omega_i, \omega_0)$ は次の式になります。 ρ は物体の反射率です。

$$f_s(x, \omega_i, \omega_0) = \frac{\rho}{\pi}$$

そして cos 項に応じた重点的サンプリングする場合、 $p(\omega_i)$ は次の式になります。

$$p(\omega_i) = \frac{\cos\theta}{\pi}$$

以上から w_i を整理すると、次の式のように単なる定数である ρ に簡略化できます。

$$w_i = \frac{f_s(x, \omega_i, \omega_0)\cos\theta}{p(\omega_i)} = \frac{\frac{\rho}{\pi}\cos\theta}{\frac{\cos\theta}{\pi}} = \rho$$

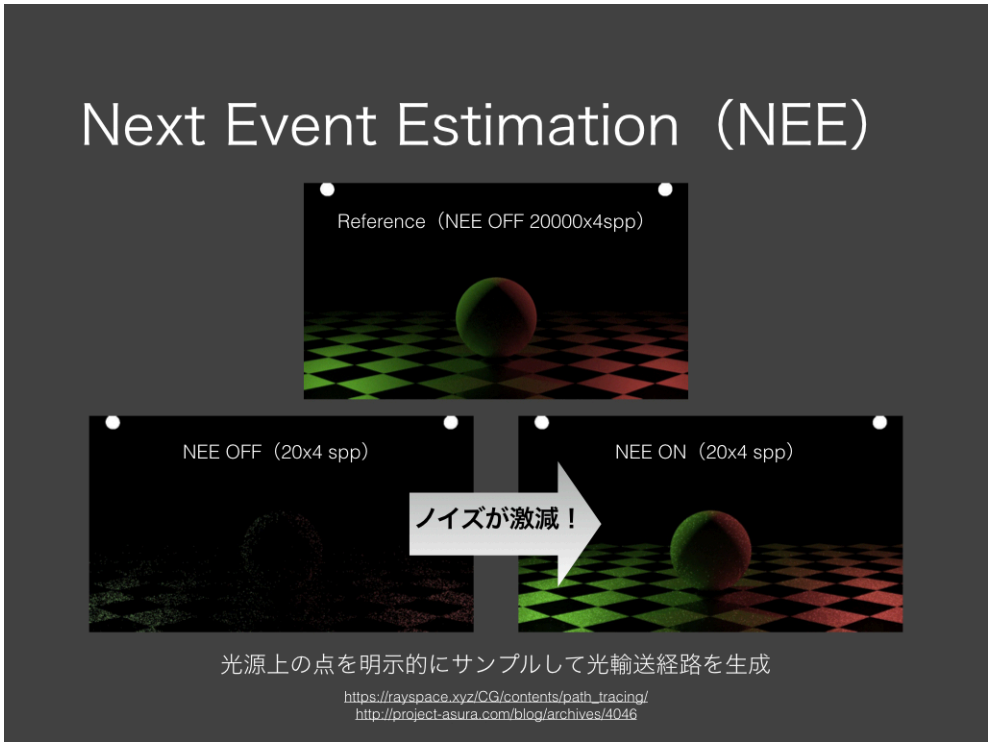
完全拡散面だけでなく、完全鏡面反射でも同様に w_i を整理することで ρ に簡略化できます。GGX といったもう少し複雑な BSDF の場合は定数にはなりませんが、愚直に 3 項を計算するよりも計算を削減できます。Hanamaru レンダラーの実装ではマテリアルのインターフェースとしてサンプリング方向と w_i をセットで返す関数を定義しました。このように式を整理することで不要な計算を省く高速化を紹介しました。

7.5 Next Event Estimation（NEE）の実装

NEE（Next Event Estimation）と呼ばれるパストレーシングのサンプリングを効率化する手法を実装しました。

光源が小さいシーンでは、BSDF による重点的サンプリングだけではなかなか光源にヒットしません。そのため、短い計算時間ではノイズだらけの結果になってしまいます。

そこで、光源の表面上の点を明示的にサンプリングして光転送経路を生成することで、効率的なサンプリングを行う手法が NEE です。



▲図 7.4　Next Event Estimation の効果

図 7.4 は同じサンプリング数で NEE 実装前と NEE 実装後を比較した結果です。左の NEE 実装前では、ほとんどのレイは光源にヒットしないために全体的に暗く、偶然にも光源にヒットした箇所が極端に明るいためノイズが目立ちます。右の NEE 実装後では、直接光源をサンプリングしているため、どの箇所も明るくノイズも目立ちません。上は長時間をかけて計算したリファレンスの結果です。NEE 実装後の結果とリファレンスの結果はほとんど一致しており、正しい結果を保ちつつ高速化できたことが分かります。

NEE の理論と実装についてはいくつかの資料[*3][*4][*5]を参考にさせていただきました。Sphere の光源を NEE に対応させるために必要な「球面上に一様分布した点を選ぶ処理」は渡辺さんの資料[*6]の「2.4 単位球面に一様分布する点」を参考にさせていただきました。

極座標ではなく (z, ϕ) で球面を表現するとシンプルに計算できます。

$$0 \leq z \leq 1, \quad 0 \leq \phi \leq 2\pi$$

[*3] パストレーシング - Computer Graphics - memoRANDOM https://rayspace.xyz/CG/contents/path_tracing/
[*4] パストレーシング / Path Tracing - Speaker Deck fhttps://speakerdeck.com/shocker_0x15/path-tracing
[*5] レイトレ再入門 - ☆ PROJECT　ASURA ☆ http://project-asura.com/blog/archives/4046
[*6] 一様乱数を使う http://apollon.issp.u-tokyo.ac.jp/~watanabe/pdf/prob.pdf

$$x = \sqrt{1 - z^2} cos\phi$$

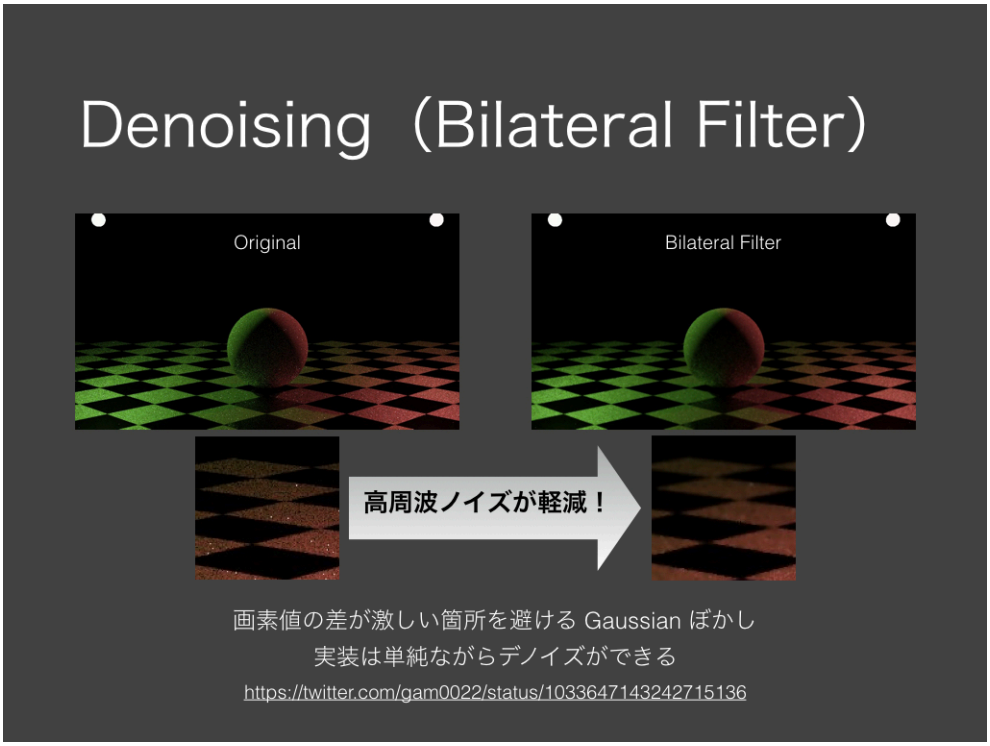
$$y = \sqrt{1 - z^2} sin\phi$$

$$z = z$$

7.6　デノイズの実装

十分な時間がかけられずにサンプリング数が不足したパストレーシングの結果には高周波なノイズが発生します。デノイズはレンダリング結果に 2D 的なフィルタリング処理を行うことでノイズを軽減する処理です。

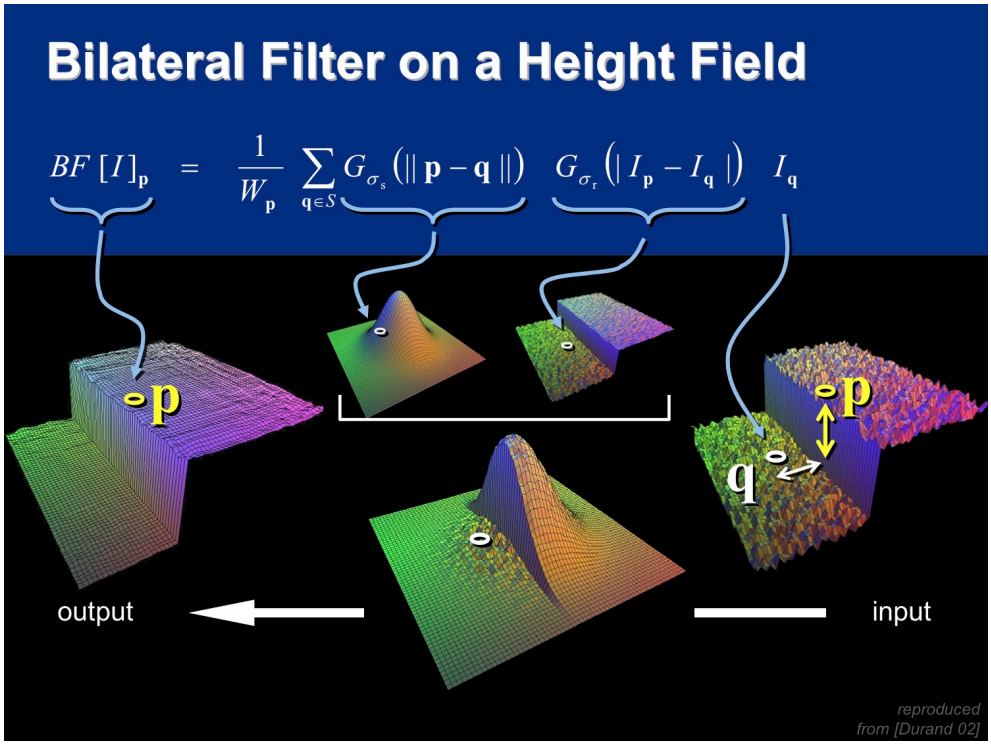
今回はデノイズの中でも、もっとも単純そうな **Bilateral Filter** を実装しました。



▲図 7.5　デノイズの効果

図 7.5 はデノイズなしとデノイズありを比較した結果です。デノイズを行うことでノイズを除去できたことが見て取れます。

Bilateral Filter を簡単に解説します。平滑化フィルター（ぼかしフィルター）として有名な **Gaussian Blur** では「空間的な重み」に基づいて周囲のピクセルを混ぜ合わせて平滑化を行います。Gaussian Blur では全体的にぼやけてしまうので、エッジ部分を保持するように重みを変化させたものが Bilateral Filter です。Bilateral Filter では「空間的な重み」と「ピクセル値の差による重み」を掛け合わせた重みを用います。



▲図 7.6　「空間的な重み」と「ピクセル値の差による重み」を組み合わせる

図 7.6 は"A Gentle Introduction to Bilateral Filtering and its Applications" SIGGRAPH 2007 の"Fixing the Gaussian Blur": the Bilateral Filter[*7]の 7 ページ目から引用しました。

$G_{\sigma_s}(|||p-q|||)$ は距離をパラメータとしたガウス関数なので「空間的な重み」となります。

$G_{\sigma_r}(|I_p-I_q|)$ は画素値の差をパラメータとしたガウス関数なので「ピクセル値の差による重み」です。

この 2 つの重みを組み合わせることでエッジ部分を保持しながら平滑化ができます。1 つ補足すると、上の式の σ_r を無限大にするとガウス関数の性質上、「ピクセル値の差によ

[*7] http://people.csail.mit.edu/sparis/bf_course/slides/03_definition_bf.pdf

る重み」が一様な分布になります。つまり σ_r を無限大にすると Gaussian Blur になります。このような知識を念頭に置いておくと、パラメータ調整に役に立つでしょう。

今回は簡単なデノイズを実装しましたが、今後の展望としては BCD[*8]などもっと品質の高いデノイズに挑戦したいです。

7.7 Rust 環境の最新化

実行速度を重視した言語を採用することやコンパイラの最適化を利用することも高速化では重要です。幸いにも Rust は実行速度を重視しているため、多くのケースで C++ と遜色ないパフォーマンスを発揮できます。Rust は新しい言語だけあって取り巻く環境の変化も激しく、古いバージョンの Rust ではパフォーマンスに関する改善が取り込まれていないことがあります。そのため、コンパイラの最適化の恩恵を最大限に受けるためには最新版に追従する必要があります。この節では Rust 環境を最新化する方法を紹介します。

Rust コンパイラのバージョンアップ

Rust のコンパイラのバージョンを 0.20.0 から 1.28.0 にアップデートしたところ、コードをまったく書き換えずに約 3 倍の高速化ができました。メジャーバージョンを上げると劇的にパフォーマンスが変わることがあるのですね。1.27[*9]で導入された SIMD の最適化の効果が大きいと予想していますが、今回はバージョンの上がり幅が大きいので、複数の理由で高速化したと思われます。

Rust を最新の安定バージョンに上げる手順は非常に簡単で `rustup update`を実行するだけです。

```
# 最新の安定バージョンに上げる
$ rustup update

# バージョンの確認
$ rustc -V
```

依存ライブラリのバージョンアップ

今回は速度に変化ありませんでしたが、依存ライブラリもバージョンアップしました。ここでは Rayon という並列化のライブラリのバージョンを上げる例を紹介します。

まず `Cargo.toml`を編集します。

[*8] BCD - Bayesian Collaborative Denoiser for Monte-Carlo Rendering `https://github.com/superboubek/bcd`

[*9] Announcing Rust 1.27 `https://blog.rust-lang.org/2018/06/21/Rust-1.27.html`

```
-rayon = "0.8.2"
+rayon = "1.0"
```

そして次のコマンドを叩くと、Cargo.tomlで指定された中で最新のバージョンに依存ライブラリがアップデートされます。

```
$ cargo update
```

依存ライブラリの最新バージョンを調べる際に **cargo-outdated**[*10]というツールが役に立ちました。cargo-outdatedは次のコマンドからインストールできます。

```
$ cargo install cargo-outdated
```

cargo outdatedを実行すると、依存ライブラリのバージョン情報を一括で調査できます。

Project
: 現在のバージョン

Compat
: 現在の Cargo.tomlのままで cargo updateからインストール可能なバージョン

Latest
: 最新バージョン

```
$ cargo outdated
Name                                Project  Compat  Latest   Kind    Platform
----                                -------  ------  ------   ----    --------
fuchsia-zircon->bitflags            1.0.4    ---     Removed  Normal  ---
fuchsia-zircon->fuchsia-zircon-sys  0.3.3    ---     Removed  Normal  ---
rand                                0.3.22   ---     0.5.5    Normal  ---
rand->fuchsia-zircon                0.3.3    ---     Removed  Normal  cfg(target_os = "fuchsi
rand->libc                          0.2.43   ---     Removed  Normal  cfg(unix)
rand->rand                          0.4.3    ---     Removed  Normal  ---
rand->winapi                        0.3.5    ---     Removed  Normal  cfg(windows)
winapi->winapi-i686-pc-windows-gnu  0.4.0    ---     Removed  Normal  i686-pc-windows-gnu
winapi->winapi-x86_64-pc-windows-gnu 0.4.0   ---     Removed  Normal  x86_64-pc-windows-gnu
```

7.8 まとめ

本章では、パストレーシングの高速化についてアプローチの異なる 5 つの手法を紹介しました。

[*10] https://github.com/kbknapp/cargo-outdated

1 つ目にレイと物体との衝突判定の回数を減らす方法を取り上げました。BVH というデータ構造を用いて空間分割することで、シーン内の全オブジェクトに対して総当りで衝突判定を行うよりも、効率的に衝突判定ができます。

2 つ目に実行時の不要な計算を洗い出して、全体の処理を減らす方法について述べました。これはパストレーシングに限らず、さまざまな高速化課題に応用が利く方法です。

3 つ目はサンプリングのアルゴリズムの改良です。BSDF に応じた重点的サンプリングと光源方向へのサンプリングを組み合わせることによって、少ないサンプリング数でも効率的にパストレーシングの計算を行い、ノイズを軽減できます。

4 つ目はレンダリング結果に 2D 的なフィルタリング処理を行うことでノイズを軽減させるデノイズです。今回はデノイズの手法として Bilateral Filter を実装しました。

5 つ目にコンパイラの最適化を利用する方法を紹介しました。今回は極端な例かもしれませんが、Rust を最新バージョンにすることで約 3 倍の大幅な高速化ができました。

最後に、本章では取り上げませんでしたが、コンピューターの性質を深く理解することも非常に重要です。たとえば、並列化をする際にメモリのキャッシュヒット率を意識した戦略に変えることでも速度の向上が期待できます。

パストレーシングをはじめとしたレイトレはとてもシンプルなアルゴリズムですが、高速化を突き詰めていくとさまざまな工夫の方法が考えられて面白いですし、得られる並列化などの知見は CG 以外の分野でも応用できます。

みなさんもレイトレや、その高速化に挑戦してみませんか？

第 8 章

ヘッドレス Chrome でリボ払いを回避している話

Yoshio HANAWA / @hnw

8.1 はじめに

この章では私の仕事とは一切関係のない、趣味のプログラミングとクレジットカードの話をします。

私は DC カード Jizile というクレジットカードを利用しています。このカードはポイント還元率が 1.5% と比較的高いのが魅力ですが、一方でリボ払い[*1]専用カードなので使い方を間違えると事故につながる可能性もあるようなカードです。

このカードではリボ返済額の上限が 5 万円になっており、月の利用額が 5 万円を超えるとリボ払いの金利が発生します。ただし、5 万円以上使った月でも Web サイトから支払い方法を変更すれば全額一括返済に変更することができます。逆にいうと、月に 1 回 Web サイトにアクセスしないとリボ払いの金利手数料が発生してしまいます。

私は忘れっぽい性格なので、手動で Web サイトにアクセスする運用を続けていたらいずれ大金を支払うことになるのは明らかでした[*2]。そこで、この作業をヘッドレス Chrome で自動化して Raspberry Pi 上で定期実行するような仕組みを作りました。最近の Chrome にはヘッドレスモードが実装されており、ウインドウシステムのない環境でもフル機能の Web ブラウザを動かすことができるようになっているのです。

本稿ではこの仕組みの作り方と、作業を通じて得た気づきを紹介します。

[*1] クレジットカードで支払った額のうち設定額を超えた分が自動的に借金になるような仕組み。仕組みを知らずに借金を増やしてしまうと非常に危険。

[*2] 実際、本稿で紹介する仕組みを作るまでは怖くて滅多に使わないカードでした。

8.2 実行環境

今回の仕組みは自宅で運用しています。実行環境は次のとおりです。

- Raspberry Pi 2 Model B
  - Raspbian 9.4
  - Node.js v8.11.1
  - Puppetter 1.7.0 （後述）
  - Chromium[*3] 70
  - crond

cron により月に 1 回ヘッドレス Chrome が起動しているというわけです。

自宅で運用する理由は、クラウド上にクレジットカード会社の ID・パスワードを置きたくなかったからです。昨今自宅サーバを持っている人は減っている印象ですが、こういう用途のためにも Raspberry Pi を自宅サーバにしておくのは悪くない選択肢だと感じています。

8.3 Puppeteer（ぱぺってぃあ）とは

すでに紹介したとおり、最近の Chrome は単体でヘッドレス動作が可能ですが、自動操作をするには操作用のライブラリも必要になります。

こうしたライブラリはいくつか存在するようですが、個人的に一番有望だと感じているのが Node.js の API である Puppeteer[*4]です。すでにユーザーも多いですし、かなりアクティブにメンテナンスされています。Chrome DevTools チームがメンテナンスしているので長期サポートが期待できるのも良いところだといえるでしょう。

8.4 Puppeteer のインストール

Puppeteer のインストールは簡単です。作業用のディレクトリに移動してから次のように打つだけです[*5]。

```
$ npm install puppeteer
```

これでカレントディレクトリの node_modules以下に必要なファイルがインストールさ

[*3] Chrome のオープンソース版プロジェクト。Puppeteer から見ると大きな違いはないので、本稿では両者をほとんど区別せず使っています。

[*4] https://github.com/GoogleChrome/puppeteer

[*5] インストールには Node.js と npm が必要です。事前に apt などでインストールしておいてください。

れます。動作に必要な Chromium も合わせてインストールされるので、すぐに試せるのは非常に良いと感じます。

Rapsberry Pi でのインストール

Raspberry Pi 上で動かす場合は `npm install`一発で終了とはいきません。というのも、`npm install`でインストールされる Chromium が x86-64 バイナリなので、ARM アーキテクチャの Raspberry Pi では動作しないのです。

そこで自前で Chrome/Chromium を用意する必要があるのですが、自前でインストールする場合は dev 版がオススメです。というのも、Puppeteer の機能追加やバグ修正の際に Chrome 本体を修正していることがあるので、古い Chrome では動かないことがあるのです。

実際、私は最初 Chrome 68（当時の最新リリース版）と Puppeteer 1.7.0 の組み合わせで使っていたのですが、Puppeteer の一部機能が動作しないトラブルが発生し、最終的に Chromium 70（当時の最新 dev 版）を使ってようやく全機能を動かすことができました。

dev 版の入手方法ですが、Chromium チームが dev 版 Chromium の ARM バイナリを apt リポジトリ形式で提供しているので、これを利用しましょう。

```
deb http://ppa.launchpad.net/chromium-team/dev/ubuntu xenial main
deb-src http://ppa.launchpad.net/chromium-team/dev/ubuntu xenial main
```

このファイルを `/etc/apt/sources.list.d/chromium-dev.list` として作成すれば apt 経由で ARM バイナリをインストールできます。

また、自前でインストールした Chromium を Puppeteer から使うには、次のように `puppeteer.launch()`のオプションでフルパスを渡してやる必要があります。

```
  const browser = await puppeteer.launch({
    executablePath: '/usr/bin/chromium-browser'
  });
```

8.5 Puppeteer でのコード例

すでに紹介したとおり、Puppeteer は Node.js のライブラリです。次に Puppeteer のコード例を紹介します。

```
const puppeteer = require('puppeteer');
(async () => {
  const browser = await puppeteer.launch();
  const page = await browser.newPage();
  await page.goto('https://www.example.com/');
```

```
    await page.waitForSelector('input[name="userid"]', {visible: true})
          .then(el => el.type('USER_ID'));
    await page.type('input[name="password"]', 'PASSWORD');
    await page.click('input[type="image"]');
  })();
```

このように page.type()と page.click()を使ってページ遷移していき、ページ遷移のタイミングで page.waitForSelector()で次ページの要素の出現を待つ、というところだけ把握すれば一定の処理はすぐに書き始められるはずです。また、細かい部分については Puppeteer 公式のマニュアル[*6]が非常に参考になります。

Puppeteer ではほぼすべての API が非同期（async）関数として提供されているため、あちこちに awaitが登場するようなコードになります。人によっては「自分の知ってる JavaScript じゃない…」と思うかもしれません。

async・awaitというのは ES2017 で導入された非同期処理の構文です。私も今回が初見だったので慣れるまで大変でしたが、慣れてくると「全部 await入れれば逐次処理っぽく書けるでしょ[*7]」くらいの雑な境地に達することができます。

8.6 Puppeteer の良いところ

私は過去に PhantomJS[*8]を使ったりスクリプト言語のスクレイピングライブラリを使ったりしてきましたが、それらと比べても Puppeteer は便利に思えます。Puppeteer の良い点を紹介します。

ブラウザの全機能を使えるのが最大の強み

銀行やクレジットカード会社などの金融系サイトだと普通の<a>リンクで良さそうな部分まで JavaScript で実装されていたりします。HTML の解釈しかできないスクレイピングライブラリでこうしたサイトを扱うのは面倒なのですが、本物のブラウザを利用している Puppeteer なら問題なく遷移できます。

他にも CSS、Canvas、SVG なども本物のブラウザと同じように動作します。

普通のブラウザと同じことが同じようにできるので、ツールを使い分けたり特別な工夫をしたりする必要がないのが最大のメリットだといえるでしょう。

メモリ 1GB 程度のマシンでも動く

普通のサイトを 1 タブで閲覧するだけであれば、メモリ 300MB もあれば問題なく動きます。メモリ 1GB の Raspberry Pi でも余裕を持って動くのは嬉しいところですね。

実際にはメモリより CPU の方がボトルネックになりやすい印象があります。広告が山

[*6] https://github.com/GoogleChrome/puppeteer/blob/master/docs/api.md

[*7] とはいえハマったときは真面目に調べることになり、結局 Promise の復習をする羽目になります。

[*8] ヘッドレスブラウザの代名詞的存在でしたが、現在はメンテナンスされていません。

ほど設置されていたり、JavaScript+Canvas でゴリゴリ描画するようなサイトでは CPU 負荷が厳しくなりがちですので、これが問題になるようであれば良いマシンを使った方がよいでしょう。

実際のブラウザの様子を見ながら開発できる

Chrome をヘッドレス動作させるのが Puppeteer のデフォルトの挙動ですが、次のようにオプション指定すればウインドウありで起動することもできます。

```
const browser = await puppeteer.launch({headless: false});
```

これにより、実際のページ上で Chrome のデベロッパーツールを使いながらセレクタを特定できるので、開発中は非常に便利です。また、期待通りに動かなかった場合のデバッグ時にも活躍します。

ブラウザのスクリーンショットが撮れる

Puppeteer スクリプトは運用に入ってからが本番だといえます。というのも、対象の Web サイト側がいつまでも同じ挙動である保証がないので、ある日突然動かなくなることが十分考えられるからです。

その意味で、ヘッドレス動作中のログは非常に重要です。思いつくログは全部残しておくとトラブル時に役立ちます。また、下手なログより断然役立つのがスクリーンショットです。以前とサイトデザインが変わった、なんてことがすぐにわかります。

ちなみに、私は Puppeteer で例外を catch したら例外の詳細とその時点のスクリーンショットを Slack に転送しています。スクリーンショット込みで時系列順に並んでいるとログとして非常に見やすいので、Slack への転送するのはオススメです。

8.7 おわりに

Web サイトからしかできない処理というのは案外多いものです。

私は今回作った仕組み以外に、銀行振込の自動化にも Puppeteer を使っています。いわゆる金融系のサービスは自動化のメリットが大きい割に API 公開されないので、Puppeteer の活躍範囲は広いように思います。

もちろん、金融系以外の Web サイトでも便利に使えるツールだと思います。私の隣の席の同僚はコマンドラインからお弁当の注文をするのに Puppeteer を利用しており、なかなか気に入っているようです。

本稿でご紹介したとおり、試すまでのハードルは非常に低いツールなので、読者の皆さんも日々の生活の改善に使ってみてはいかがでしょうか。

著者

第 1 章 Kinuko MIZUSAWA

某アイドルリズムゲームのイベントやら何やらと原稿のスケジュールが重なり正直死ぬ思いでしたが何とか原稿を形に出来て良かったです。今年は 3D 関係を頑張っていきたいと思います。

第 2 章 黒井春人

平成最後の技術書典。ちゃんと原稿が書けてよかったです。

第 3 章 Daisuke TAKAI

技術書典 5 当日は大阪でとあるライブに参加してます。

第 4 章 Yoshihiro KASHIMA

Microsoft Excel が好きです。

第 5 章 Daisuke MAKIUCHI / @makki_d

眼鏡っ娘が好きです。

第 6 章　Junichi KIKUCHI / @jukey17

初参加でバタバタしましたがガッツリと書けたと思います！ スプラトゥーン 2 楽しいです！

第 7 章 Sho HOSODA / @gam0022

今年はレイトレ飛躍の年でしたね！ Microsoft が DirectX Raytracing を発表し、NVIDIA も GeForce RTX を発売して、リアルタイムレンダリングの世界にも羽ばたいたレイトレの今後に目が離せません。

第 8 章 Yoshio HANAWA / @hnw

クレジットカードが好きです。30 枚あるので引っ越し時の手続きが怖いです。

表紙　あつち

幼女描くの楽しかったです。

木こり担当 ひらた

褐色はいいぞ

装丁 山浦

つづりには気を付けよう

PDF 版表紙 きくた

眼鏡女子イイ…！

編集 於保 俊 / @ohomagic

今回は編集と取りまとめ役での参加です。

KLab Tech Book Vol. 3

2018 年 10 月 8 日　技術書典 5 版（1.0）
著　者　KLab 技術書サークル
編　集　於保 俊
発行所　KLab 技術書サークル
印刷所　日光企画