KLab Tech Book vol.04

// Argument Clinicを使ってみよう
// Rustで世界を統一する
// QRコードマニアックス— 数字・英数字・漢字モード
// Unity ×レイマーチングによる映像制作の実践手法
// 自作キーボード入門
// 機械学習APIの力でCAPTCHAを破る
// Unityでの条件付きコンパイルシンボル定義にエディタ拡張を活用する
// デジカメで撮影した写真の正確なタイムスタンプを推測する

KLab Tech Book vol.04

KLab学園

ここは、あなたたちの世界とちょっとだけ近くて、でも遠い別世界です

日本の人口減少、AIやロボティクスの技術発達により
いつの間にか無駄のない合理性が求められる社会になりました
独自文化や娯楽が衰退していく中
私たちは「心躍る体験を創出する人を育てる」を
基本理念とした学園で活動に励んでいます

「KLab学園」は日文科、国際科、創造科の3つの学科があり
私たちは創造科の2年生！
1年生の頃、最初は何をやっていいかわからなかったり
日文科や国際科と衝突することも多かったけれど
その経験は自分の自信に繋がりました
この人に私たちの作品を届けたい！と決めてから
毎日がワクワクするような日々を送っています

このメッセージが届いているかはわかりませんが
いつか時間すら超えて
あなたに私たちの作品を届けられるといいなー…と、思います！

- 時乃 天音

レベルデザイナー
時乃 天音
ときの　あまね

観察がクセ
ワクワクを常に探している

ロボット使い
糸繰 叶
いとくり　かな

ロボや機械の体調がわかる
機械オイルソムリエ

ハッカー
一道 千桜
いちどう　ちはる

痕跡を残さずに去る
幻のハッカー<Ghost>

子供プログラマー
外池 雛子
そといけ　ひなこ

とあるゲームとの出会いにより
プログラマーの道に

ケモ少女
ニーナ ルナール

耳や尻尾がすぐ反応してしまう
いたずら大好き

図書委員
星月 美兎
ほしつき　みと

本の知識が豊富
目線を合わせて会話する

KLab Tech Book Vol. 4

2019-04-14 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 4 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。業務に少しだけ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なく、唯一の統一感が、表紙が全員眼鏡っ子という本書ですが、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。

知的好奇心に駆動される「楽しさ」は、我々エンジニアが技術や知識を追求する原動力です。その楽しさが少しでも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

はじめに **2**
お問い合わせ先 2
免責事項 2

第 1 章 Argument Clinic を使ってみよう **7**
1.1 これは何？ 7
1.2 最初の一歩 7
1.3 コンバーター - 引数、戻り値の Python, C 間の自動変換 10
1.4 引数コンバーターを自作する 10
1.5 既存のコードに合わせた出力を得る 12
1.6 おわりに 13

第 2 章 Rust で世界を統一する **14**
2.1 はじめに: Rust/Peridot とは 14
2.2 プラットフォームに依存する部分をなんとかする: NativeLinker トレイト編 15
2.3 プラットフォームに依存する部分をなんとかする: cradle 編 17
2.4 生ポインタと共に 18
2.5 cargo と Xcode と Gradle と 20
2.6 プラットフォーム固有のカスタマイズを可能にする 22
2.7 終わりに 23
2.8 出典 24

第 3 章 QR コードマニアックス — 数字・英数字・漢字モード **25**
3.1 QR コードのモード 25
3.2 8 ビットバイトモード 26
3.3 数字モード 26
3.4 英数字モード 27
3.5 漢字モード 29

3.6 まとめ 30

第 4 章 Unity ×レイマーチングによる映像制作の実践手法 32
4.1 Tokyo Demo Fest について 33
4.2 題材『WORMHOLE』の概要 33
4.3 レンダリング 33
4.4 距離関数によるモデリング 35
4.5 演出の実装 40
4.6 音楽との同期方法 44
4.7 おわりに 45

第 5 章 自作キーボード入門 46
5.1 組み立て編 46
5.2 無線化編 47
5.3 おわりに 50

第 6 章 機械学習 API の力で CAPTCHA を破る 51
6.1 はじめに 51
6.2 CAPTCHA とは 51
6.3 CAPTCHA 破りの正攻法 52
6.4 手軽に CAPTCHA を破りたい！ 53
6.5 ノイズ除去で認識精度を上げる 55
6.6 reCAPTCHA を使おう 56
6.7 まとめ 57

第 7 章 Unity での条件付きコンパイルシンボル定義にエディタ拡張を活用する 58
7.1 本章で作成するシンボル情報編集エディタ拡張の概要 59
7.2 シンボル編集エディタ拡張の利用イメージ 60
7.3 シンボル編集エディタ拡張の実装 62
7.4 いざ、ビルド 69
7.5 おわりに 72

第 8 章 デジカメで撮影した写真の正確なタイムスタンプを推測する 74
8.1 デジカメと Android 74
8.2 要望：デジカメの時計がズレるのをなんとかしたい 75
8.3 システム概要検討 78
8.4 OpenMemories SDK を利用したアプリ開発 80
8.5 スマフォとカメラを接続し、カメラの時刻情報を取得する 82
8.6 まとめ 88

8.7 本章で取り上げなかった要検討事項 88

著者 **90**

第 1 章

Argument Clinic を使ってみよう

Shunsuke Ito / @fgshun

1.1　これは何？

CPython の標準ライブラリのうち C で書かれたもののコードには「`/*[clinic input]`」から始まるコメントが散見されます。これはコードを読み書きする人向けに書かれたものではなく、Argument Clinic というツールに対する入力です。

Argument Clinic は CPython にて C/C++ でモジュールを書く時に使えるメソッド作りに特化した何か、いわゆる DSL です。メソッドの引数解釈、シグネチャ付けといった退屈な仕事を肩代わりしてくれます。

出力されるコードは CPython の内部構造と密結合したものです。ドキュメントが用意されておらず、将来の互換性が担保されていない関数・マクロが使われる可能性もあります。逆にいえば、出力されるコードが CPython の進化にあわせて改善されることで、新バージョンの恩恵[*1]を受けることもできます。標準ライブラリにも使われているこのツールを使ってみましょう。

1.2　最初の一歩

Argument Clinic は CPython 3.4 以降であればソースに含まれています。パスは `Tools/clinic/clinic.py` 。ツールに対する入力はコメントとして C のソースに埋め込む形をとります（リスト 1.1）。

▼リスト 1.1　空の clinic への入力

[*1] たとえば Python 3.6 から追加された関数呼び出し規約 METH_FASTCALL

```
/*[clinic input]
[clinic start generated code]*/
```

このファイルを clinic.py で 1 度処理すると start generated code の直後に生成されたコードが追加されます（リスト 1.2）。

▼リスト 1.2　空の clinic への入力に対する空の出力

```
/*[clinic input]
[clinic start generated code]*/
/*[clinic end generated code: output=da39a3ee5e6b4b0d input=da39a3ee5e6b4b0d]*/
```

複数回実行しても入力、つまり input, start 間が同じであれば、出力 start, end 間は変化しません。なお出力された箇所を直接書き換えるのはツールの想定するところではなく、もし書き換えてもツールの再実行時点で内容が失われる旨の警告が出ます。

入力を繰り返し編集し、望みの出力が得られるようにしていくのが clinic.py を使ったメソッドの作成工程です。

早速ひとつ作ってみましょう。なるべく簡単そうなものをといったところで次のようなものにしてみましょう。単純な + 演算です（リスト 1.3）。

▼リスト 1.3　Python で書かれた関数例

```
def add(a, b):
  """add a to b"""
  return a + b
```

C で書くとリスト 1.4 のようになるでしょうか。

▼リスト 1.4　C で書かれた関数例

```
PyObject *
spam_add(PyObject *module, PyObject *args, PyObject *kwargs)
{
  PyObject *a, *b:
  static char *keywords[] = {"a", "b", NULL};
  if (!PyArg_ParseTupleAndKeywords(args, kwargs, "OO", keywords, a, b)) {
    return NULL;
  }

  return PyNumber_Add(a, b);
}
```

これに相当するメソッド作りを Argument Clinic ツールで行ってみます。まずモジュール名の宣言を、そしてモジュールにどのような名前のメソッドを作るのか、その引数は、 docstring はどのようにしたいのかを記述します（リスト 1.5）。

▼リスト 1.5　spam.c - clinic を使った関数の宣言の例

```
/*[clinic input]
module spam
[clinic start generated code]*/

/*[clinic input]
spam.add

  a: object
  b: object

add a to b
[clinic start generated code]*/
```

するとリスト 1.6 のような出力が得られます。

▼リスト 1.6　clinic からの出力

```
static PyObject *
spam_add_impl(PyObject *module, PyObject *a, PyObject *b)
/*[clinic end generated code: output=841f3d07aa0b7744 input=e8d7719e11a46e14]*/
```

end の直後に

▼リスト 1.7　clinic を使った関数の実装の例

```
{
  return PyNumber_Add(a, b);
}
```

と、C で書いた時の引数周りの変数宣言と PyArg_ParseTuple を取り除いたものをおけば対応は概ね完了します。

あれ？　作りたかった関数は spam_add であったはずなのにでき上がったのは spam_add_impl ですね。でも心配はいりません。先ほどの空の入力に対する出力とは異なり追加の出力 clinic/spam.c.h ができ上がっています（リスト 1.8）。

▼リスト 1.8　別ファイルに切り出された出力 clinic/spam.c.h - 一部抜粋

```
PyDoc_STRVAR(spam_add__doc__,
"add($module, /, a, b)\n"
"--\n"
"\n"
"add a to b");

#define SPAM_ADD_METHODDEF    \
  {"add", (PyCFunction)spam_add, METH_FASTCALL|METH_KEYWORDS, spam_add__doc__},

static PyObject *
spam_add_impl(PyObject *module, PyObject *a, PyObject *b);

static PyObject *
spam_add(PyObject *module, PyObject *const *args, Py_ssize_t nargs,
    PyObject *kwnames)
{
  // 略。引数の解釈に成功したら spam_add_impl を呼ぶコードが生成される
```

この追加で生成されたコードには docstring 、 PyMethodDef 定義のために使えるマク

ロ、 _impl の関数宣言、そして spam_add の実装[*2]が含まれています。というわけでこれを include し、 PyMethodDef に出力された SPAM_ADD_METHODDEF マクロを含めることで今度こそ対応完了です。

1.3 コンバーター - 引数、戻り値の Python, C 間の自動変換

先ほどは PyObject* を受け取りそして返す関数を作りました。しかし、 C/C++ でモジュールを書いているのですから C の型への変換を行う処理は頻出することとなるでしょう。引数および戻り値で C の型を用いるには、たとえばリスト 1.9 のようにします。

▼リスト 1.9　long 型を引数にとり double 型を返す例

```
/*[clinic input]
spam.div -> double

  a: long
  b: long

divide a by b
[clinic start generated code]*/

static double
spam_div_impl(PyObject *module, long a, long b)
/*[clinic end generated code: output=92d70cf68fdf86d3 input=28d08239ebebed11]*/
```

これにより引数に渡された PyObject* を long に変換する前処理と失敗時のエラー送出、そして戻り値とした double を PyObject* に変換する後処理が追加され、変換後の値に対応する関数宣言が生成されます。なので変換はツールに任せて本来の処理の実装に専念できます。

引数と戻り値の変換処理は別の仕組みであるため、対応している型および書き方が異なります。それぞれの対応するコンバーターについては clinic.py --converters を実行することで確認できます。Argument Clinic は既存の書き方をされたコードすべてに対応するべく作られているせいか、戻り値の変換より引数の変換の方が多くのケースに対応しています。PyArg_ParseTupleでできることが多彩であるからですね。

1.4 引数コンバーターを自作する

では、独自の変換処理を何度も繰り返し適用する必要が生じた場合は Argument Clinic を自分好みに作り変えなければならないのでしょうか。いいえ、引数のコンバーターを自

[*2] Python 3.7 付属の clinic.py での出力。関数のシグネチャが PyCFunction と異なるのは METH_FASTCALL 呼び出し規約が使われているからです。この規約がまだ存在しなかった Python 3.5 以前の clinic.py を使用した場合はまた異なる出力となります。出力されたコードが過去および将来のバージョンで動作する保証はありません。

作して追加できます。

まず clinic.py を見ると引数のコンバーターは CConverter を継承して実装されていることがわかります（リスト 1.10）。

▼リスト 1.10　clinic.py にある long 引数のコンバーター

```
class long_converter(CConverter):
  type = 'long'
  // 後略
```

次に [python input] です。 Argument Clinic への入力には、これまでに紹介した [clinic input] 記法だけではなく Python コードも利用可能です。リスト 1.11 のように使います。

▼リスト 1.11　python input 使用例

```
/*[python input]
import platform
print('// Python {}'.format(platform.python_version()))
print('// {!r}'.format(CConverter))
[python start generated code]*/
// Python 3.7.2
// <class '__main__.CConverter'>
/*[python end generated code: output=b034f54fc66aabf4 input=733f05da31622073]*/
```

標準出力に書いたものがそのまま出力に出てくるので、Python で C のコードを書いたりすることもできます。ここで実行されている Python から Argument Clinic 中で定義されている変数 CConverter が見えていることに注目してください。つまり、[python input] 内にて自作のコンバーターを実装すれば clinic.py を書き換えることなくそれを追加することができるのです。

では、作り方です。 CConverter を継承したクラスを作り、 type クラス変数に変換後の型を、 converter クラス変数に変換用の関数名を設定します。クラス名は _converter で終わるようにしてください。この命名規則に従うことでコンバータとしての登録処理が済むようになっています。変換用の関数は、 PyObject* と変換結果を渡す変換後の型のポインタを引数にとり、変換に成功したら 1 を、失敗したら 0 を返すように作ります。たとえば何でも long 0 に変換するコンバータはリスト 1.12 のようになります。

▼リスト 1.12　何でも long 0 に変換する自作コンバーター

```
static int
_zero_converter(PyObject *arg, long *value)
{
  *value = 0;
  return 1;
}

/*[python input]
class Zero_converter(CConverter):
  type = 'long'
  converter = '_zero_converter'
```

```
[python start generated code]*/
```

PyObject *arg を無視せずに PyLong_AsLong などで変換、値域の確認を行えばもう少し意味のあるコンバータになるでしょう。こうしてできた Zero_converter をつかうには末尾の _converter を取り除いた名前 Zero を使います。

▼リスト 1.13　自作コンバーターの使用

```
/*[clinic input]
spam.zero

  value: Zero
[clinic start generated code]*/

static PyObject *
spam_zero_impl(PyObject *module, long value)
/*[clinic end generated code: output=7f394e205a5ccf15 input=531247c2ffd91669]*/
```

1.5　既存のコードに合わせた出力を得る

ここまではゼロからライブラリを書くという前提で説明をしていました。ところで、Argument Clinic は、既存の C/C++ 製ライブラリに適用して引数解釈処理を取り除いたり docstring 付けを行うためにも使えます。むしろこちらが本来の用途です。このため既存のコードに合うように出力を調整するための機能をいくつか持っています。それを紹介します。

まずは生成される関数名の変更方法です。関数名はデフォルトでは モジュール名_クラス名_関数名_impl という名前がつくようになっているため、モジュール階層が深い、クラス名が長いといった場合不必要に長い名前になってしまいます。これを変更することができます。メソッド名の後ろに as 関数名 を付加します。リスト 1.14 では spam モジュールの Mycls のメソッド eggs を作る際に spam_Mycls_eggs_impl と命名されるところを縮めています。

▼リスト 1.14　関数名の変更

```
/*[clinic input]
spam.Mycls.eggs as Mycls_eggs
[clinic start generated code]*/

static PyObject *
Mycls_eggs_impl(MyclsObject self)
/*[clinic end generated code: output=f7ef008fb9f29a91 input=e18a6dad0392a2aa]*/
```

次は Python 上での引数名と C/C++ 上での引数名の分離です。 Python 上での引数名の後ろに as C 上での変数名 を加えます。これは言語による予約語の違いによる問題の回避にも使えます。リスト 1.15 は void という名の引数を持った関数を用意しようとしたものの C の予約語と被ったため void_obj と呼び変えて回避しているものです。

▼リスト 1.15　引数名の変更、予約語の回避

```
/*[clinic input]
spam.ham
    void as void_obj: object
[clinic start generated code]*/

static PyObject *
spam_ham_impl(PyObject *module, PyObject *void_obj)
/*[clinic end generated code: output=db5655fd02283ef5 input=580ce19e4b0350e0]*/
```

最後に生成されるコードの位置の調整です。デフォルトでは `[clinic input]` を書いたファイルそのものへの出力は最低限に留めて、`clinic/.c.h` という別のファイルへの出力を行うようになっています。この挙動を `output preset` で変更することができます。block プリセットを用いると、別ファイルへの出力を止め `start generated` 直下にすべてまとめて出力するようになります。また buffer プリセットを用いると出力を溜めておいて `dump buffer` で任意の位置に出力することができるようになります（リスト 1.16）。

▼リスト 1.16　生成コードの位置の調整

```
/*[clinic input]
output preset buffer
[clinic start generated code]*/
/*[clinic end generated code: output=da39a3ee5e6b4b0d input=531a13e5785536fb]*/

// 中略

/*[clinic input]
dump buffer
[clinic start generated code]*/
// clinic/.c.h に出力されていた自動生成コードがここに出力されるようになる
```

1.6　おわりに

Argument Clinic ツールの使い方について駆け足ながら解説しました。CPython の拡張モジュールを書いてみる際のおともにどうでしょう。そして標準ライブラリのコードを読む際の役に立てばまた幸いです。

第 2 章

Rust で世界を統一する

Shinya Naganuma / @Pctg_x8

趣味で製作しているゲームエンジン「Peridot」では、開発言語として Rust をメインで使用しつつ、一切のユーザーサイドのコード変更なしに Windows/macOS/Linux/Android/iOS など多岐にわたるプラットフォームへのバイナリビルドを行えるような環境を整えています。

本章では、クロスプラットフォーム環境を整備する際に行った自動化、抽象化などの手法や概念について解説をしたいと思います。

2.1 はじめに: Rust/Peridot とは

Rust は Mozilla が支援しているプログラミング言語で、効率的かつ安全性の高いプログラムを書けるようにすることを目的としています。

Rust がもつ特徴として、C/C++ などで発生するデータ競合やぶら下がりポインタによるメモリリーク、不正な参照外しなどの危険なコードをコンパイラレベルで検知し、実行時エラーを事前に防ぐことができるような仕組み（ライフタイムおよび借用）を持っていることが挙げられます。**借用**（参照）には厳しい制限が課されており、特に可変な借用（`&mut`）は一つの変数に対してただ一つしか同時に持つことができない[*1]という非常に厳しい制限が課されています。これがレキシカルスコープで判定されるため Rust が難しい言語だと評される要因になっていますが、このような厳しい制限を課すことで Rust を使っている限り複数スレッドからのデータ書き込みに起因する不思議なバグに悩まされることはありません[*2]。

[*1] 他の参照は不変であろうと可変であろうと制限される

[*2] `unsafe` を使っていない場合に限る

また、Rust では基本的に全ての値は参照で渡すか、所有権を譲渡する（いわゆるムーブする）ようになっています。コピーは明示的です。コピーの必要性を意識させる言語設計のため、自然と不要なコピーを発生させることがなくなり比較的高速なプログラムを書くことができるようになっています。

先述の、自然と効率的なプログラムが書けるという特性のため、効率が重要視されるゲーム、広くはインタラクティブメディアの分野における開発言語としてもっとも適した選択肢となり得ます。Rust のこの特性に注目し、内部をほぼ全て Rust で書き上げることでミドルウェアとしてのオーバーヘッドを最小限にすることを目的としたゲームエンジン、それが Peridot です。

Rust はコンパイラバックエンドとして LLVM を採用しており、LLVM によるターゲットレベルでの最適化や複数プラットフォームのサポートを実現しています。Windows/Linux、x86/x86_64 など PC プラットフォームはもちろんのこと、Android（ARMv7/ARM64）や iOS へのビルドにも対応しています[*3]。

2.2 プラットフォームに依存する部分をなんとかする: NativeLinker トレイト編

単一のコードで複数のプラットフォーム向けのバイナリを出そうとする場合、ほとんどの言語ではプラットフォーム固有の事情に頭を悩まされます。各種プラットフォームが提供している API が違う他、macOS/iOS/Android ではアプリがサンドボックス化された環境で実行されるためファイルの扱いが異なるなどかなり微妙な差もうまく吸収する必要があります。ここをうまく抽象化することで、その上のユーザーコードではプラットフォームを透過的に扱うことができるようになります。

Peridot では `NativeLinker` というトレイトを用意することで抽象化を図りました。

トレイトとはすごく平たく言うと `interface` と等価な機能を提供するものです。その構造体が

- 何ができるのか
- どういう用途で使われるのか
- どういった特徴を持っているか

などを記述します（リスト 2.1）。

▼リスト 2.1　NativeLinker の定義

[*3] ただし完全に動くことは保証されていない 詳細は https://forge.rust-lang.org/platform-support.html

```
pub trait NativeLinker
{
    type AssetLoader: PlatformAssetLoader;
    type RenderTargetProvider: PlatformRenderTarget;
    type InputProcessor: InputProcessPlugin;

    /// アセット（画像やサウンドなどのリソース）を
    /// ストレージなどから読み込む役割を持ったオブジェクトを返す
    fn asset_loader(&self) -> &Self::AssetLoader;

    /// プラットフォーム固有の描画ターゲット（ウィンドウなど）を抽象化する
    /// オブジェクトを返す。具体的には、VkSurfaceKHR の生成やクライアントエリアの
    /// サイズの取得などを受け持っている
    fn render_target_provider(&self) -> &Self::RenderTargetProvider;

    /// プラットフォーム固有の入力ソースをハンドリングするオブジェクトを返す
    /// 具体的には、Windows であれば WM_INPUT を適切に処理したものになるし、
    /// Linux であれば大抵の場合で udev/uevent を適切にハンドリングするものになる
    fn input_processor_mut(&mut self) -> &mut Self::InputProcessor;
}
```

トレイトがインターフェイスと違うところは、**関連する型**（関連型）を内部に持つことができる点です。ジェネリクスのようですが若干違っていて、例えばジェネリクスの場合は違う型引数を渡せば違うものとして認識されますが、関連型の場合は変えても NativeLinker であることに変わりはないため、ある型に実装されている NativeLinker は常に一つです（リスト 2.2）。

▼リスト 2.2　関連型とジェネリクスの違い

```
// 関連型
pub trait AssocType
{
    type T;

    fn get(&self) -> &Self::T;
}
// ジェネリクス
pub trait GenericType<T>
{
    fn get(&self) -> &T;
}

pub struct Implementor(i32, i64);

// ジェネリクスは複数回実装できる GenericType<i32> != GenericType<i64>だから
impl GenericType<i32> for Implementor
{
    fn get(&self) -> &i32 { &self.0 }
}
impl GenericType<i64> for Implementor
{
    fn get(&self) -> &i64 { &self.1 }
}

// 一方で、関連型の場合は一回しか実装できない impl〜の部分が一緒なので、
// そこを見るのがわかりやすい
impl AssocType for Implementor
{
    type T = i32;

    fn get(&self) -> &i32 { &self.0 }
}
/* コメントを外すとコンパイルエラー
impl AssocType for Implementor
{
```

```
    type T = i64;

    fn get(&self) -> &i64 { &self.1 }
}
*/
```

ジェネリクスでなく関連型を使用する利点として、ジェネリクスでの境界指定を簡潔にできる点があります（リスト 2.3）。

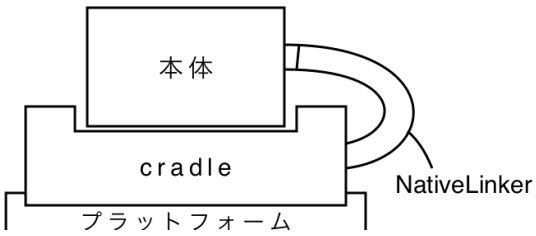
▼リスト 2.3　境界指定の例

```
// ジェネリクスの場合、全部指定する必要がある
pub trait TooManyGenericParams<A, B, C, D, E, F, G, ..., Z> { ... }
pub struct Container<T>
    where T: TooManyGenericParams<i32, u32, i64, ..., String>
{
    ...
}

// 関連型の場合、境界として必要な最小限の記述に抑えられる
pub trait TooManyAssocTypes
{
    type A;
    type B;
    ...
    type Z;
}
pub struct Container<T>
    where T: TooManyAssocTypes<B = i32>
{
    ...
}
```

2.3　プラットフォームに依存する部分をなんとかする: cradle 編

残る問題は、この NativeLinker をどこでユーザーコード側に受け渡すのか、です。トレイトは単純な特性の定義であり、実体となる型が別に必要となります。先述した通り NativeLinker はプラットフォーム依存部分の共通操作を抽象化するためのものですので、実体となる型はプラットフォームごとのものとなります。Peridot では、これを含む**土台となるコード**（cradle）を用意し、そこからユーザーコードに NativeLinker を受け渡す構造をとっています。ユーザーコードと cradle はビルドの直前にソースコードレベルでリンクするようにします。こうすることで、必要であればソースコードのレベルでインライン展開を促すことが可能になり、コンパイラによる積極的かつ高度な最適化を可能にしています。

ここまで解説した、Peridot におけるユーザーコードと cradle の概念図を図 2.1 に示します。



▲図 2.1　NativeLinker,cradle, ユーザーコード（本体）の概念図

2.4　生ポインタと共に

先述の cradle 部分のコードでは、プラットフォームと密接に関わる必要があるため**生ポインタ**（`*const T`/`*mut T`）に触れることは避けられません。Rust の安全性の大部分を保証しているのは厳しい借用ルールおよびライフタイム制約であり、これを守っている以上は未定義動作を踏むことはありませんが、生ポインタには適用されません。扱いには十分気をつける必要があります。

生ポインタを扱う場合、その大部分を `unsafe`というブロックに入れる必要があります。これにより非安全なコードを分離できるため、バグの原因となる箇所を特定することが簡単になります。また、`unsafe`内はあらゆるコンパイラチェック[*4]をすり抜ける超危険地帯となっています。そのため、`unsafe`の範囲はできるだけ小さく保つべきとされています。

Rust において生ポインタ内の値にアクセスする方法は 4 つあります。基本的に全て `unsafe`です。

1. 普通にデリファレンスする。値が `Copy`を実装していない場合は所有権の移譲ができずコンパイルエラーになるため、使えるところはかなり限られる。
2. `<*const T>::as_ref`または`<*mut T>::as_mut`を使い、`Option<&T>`や `Option<&mut T>`に変換する。
3. `NonNull<T>`を使う。中身を取り出す処理（`NonNull<T>::as_ref`/`NonNull<T>::as_mut`）はいずれも `unsafe`。
4. `Box<T>::from_raw`を使う。

Windows における COM インターフェイスを取り回す、ウィンドウハンドルを Rust 側で管理するなど「持っている間は必ず null にならず、Rust 側で解放の責任を負う（つまり所有している）」データの場合、3. の `NonNull<T>`は非常に良い選択肢です。`NonNull<T>`は内部表現が生ポインタ（`*mut T`）と同じためメモリ消費のオーバーヘッドがないこと、また生ポインタを取り出す操作が `unsafe`ではないため、生ポインタを要求してく

[*4] ただし Borrow Checker を除く

る関数に渡しやすくなります（リスト 2.4）。

▼リスト 2.4　COM インターフェイスを Rust でラップする例。このようなイディオムは"newtype"と呼ばれている

```
pub struct ID3D12Device(NonNull<ID3D12Device>);
impl ID3D12Device
{
    pub fn new() -> Option<Self>
    {
        // NonNull<T>::new(*mut T) は Option<T>を返すので、Option の
        // コンビネータで綺麗に書ける
        NonNull::new(unsafe { D3D12CreateDevice(...) }).map(ID3D12Device)
    }
    ...
}
```

一方で、「受け取った生ポインタの中身を、所有しなくていいので（呼び出し側が解放することがわかっているので）扱いたい」といった場合は 2. の`<*const T>::as_ref`および`<*mut T>::as_mut`が最適です。これらのメソッドは、null の場合には`None`を返すなどの最小限のガードは入っているものの`unsafe`とされています。これは、生ポインタの指している先が有効なデータであることを保証できないためです。このパターンにマッチするものの例として、Android でユーザー側のエントリポイントとして最初に呼ばれる`android_main`関数が挙げられます。この関数はネイティブ側のグルーコードから必要な情報が入った構造体へのポインタが渡され、かつそれは呼び出し側で解放することになっているため、今回のパターンにぴったり当てはまります（リスト 2.5）。

▼リスト 2.5　Peridot の android_main（一部）

```
#[no_mangle]
pub extern "C" fn android_main(app: *mut android::App)
{
    let app = unsafe { app.as_mut().expect("null app") };
    app.on_app_cmd = Some(appcmd_callback);
    ...
}
```

あるいは、Windows におけるウィンドウコールバックもこのパターンが当てはまるでしょう。ウィンドウハンドルに関連付けたデータの所有権は、大抵の場合生成した関数またはクラスが持つため、データの所有をしない`as_ref`/`as_mut`を使用するのが適切な選択肢となります（リスト 2.6）。

▼リスト 2.6　ウィンドウコールバックでの例

```
extern "system"
fn window_callback(hwnd: HWND, msg: UINT, wparam: WPARAM, lparam: LPARAM)
    -> LRESULT
{
    match msg
    {
        ...
        WM_SIZE =>
        {
```

```
            let obj = unsafe
            {
                (GetWindowLongPtrA(hwnd, GWL_USERDATA) as *mut Holder)
                    .as_mut().expect("null obj")
            };
            ...
        },
        ...
    }
}
```

ここで述べた選択肢以外を使うことはあまりないと思われます。特に 4 番目の `Box<T>::from_raw`を使う方法はかなり特殊で、「Rust から他言語側にポインタを渡し、最後に他言語側からそれを受け取って Rust 側で解放処理をする」以外の用途がほとんどありません。malloc で確保されたポインタを `Box<T>`で解放することは諸般の事情[*5]で大抵の場合できないため、自動で解放させるために `Box<T>`を使うことはできません。

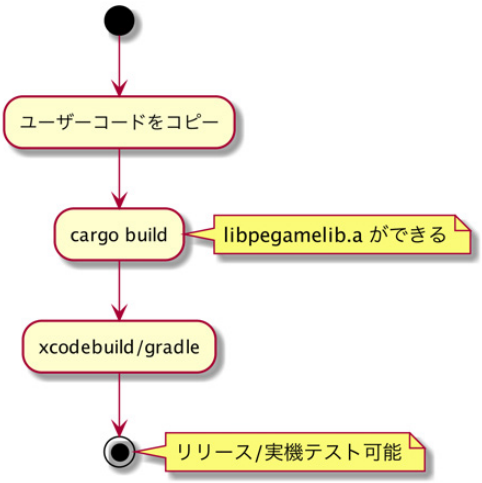
2.5 cargo と Xcode と Gradle と

Windows や Linux、サンドボックス環境を利用しない（コンソール上で動く）macOS アプリケーションであれば、Rust の標準ビルドシステムである cargo のみでバイナリ出力が可能です。一方、AppStore で配布できる形式の macOS アプリケーションや iOS/Android で動くバイナリを出力するには、バイナリのビルドに加えて署名などの操作が必要なため、xcodebuild や Gradle などの専用のビルドシステムを使用することがほぼ必須となります。当然ですが、これらのビルドシステムでは Rust のコードをコンパイルできませんので、

1. cargo で Rust のコードを静的ライブラリにコンパイル
2. その後、xcodebuild や Gradle などで配布可能パッケージを作成

といった手順を踏みます。具体的には、Android では NativeActivity を使用したパッケージを作成し、macOS/iOS では Swift で記述された、いわゆる Bootstrap と呼ばれるコードと静的リンクを行って一つのパッケージを作成します（図 2.2）。

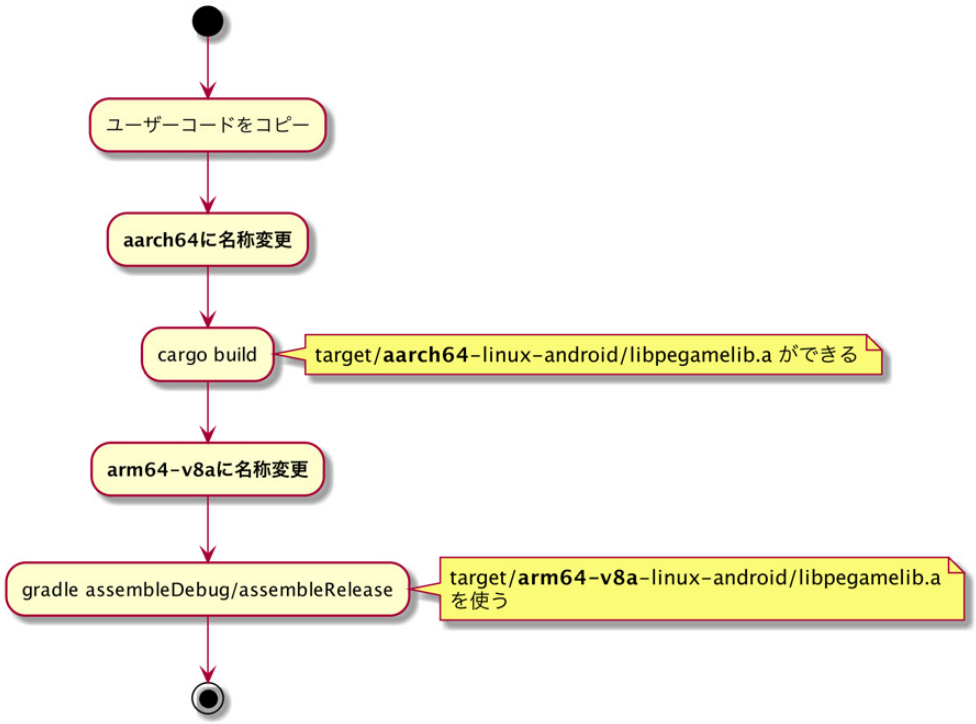
[*5] jemalloc という libc の malloc とは異なるアロケータをデフォルトで使用するため、互換性がない



▲図 2.2　統合ビルドパイプライン簡易チャート

cargo で出力した.a ファイルは `target/<target-triple>/`以下に出力されるため、target-triple の一部が設定される変数（`ANDROID_ABI`など）を使用することで柔軟にビルドターゲットを変更することが可能です。可能ではありますが少し問題がありまして、ARM64 向けのビルドを行う場合、cargo では ABI の表記が aarch64 になるのですが、Gradle（CMake）では arm64-v8a になってしまい、両者で互換性がありません。これらの差をビルドパイプラインのスクリプトで吸収する必要があります。

具体的には、cargo でビルドする前に cargo の出力ディレクトリを aarch64-linux-android に、Gradle に処理を渡す前に arm64-v8a-linux-android にそれぞれ名前を変更するようにします（図 2.3)。cargo でのビルド前に出力ディレクトリの名前を変更しておかないと、cargo は前回のビルド結果がないものとして処理してしまうので毎回フルビルドが発生してしまいます。



▲図 2.3　Android でのビルドパイプライン

2.6　プラットフォーム固有のカスタマイズを可能にする

Android ではアプリ名などを外部の xml ファイルを使って言語別に設定することが可能です。また、Android や iOS で配布可能なバイナリを生成できるようにする場合、そのパッケージに与えられる ID は一意である必要があります。Rust 単体ではアプリ名を変更することもパッケージ ID を変更することも不可能なため、これらのカスタマイズ処理はビルドパイプラインのスクリプトで行う必要があります。Peridot では bash 環境向けと PowerShell 環境向けのスクリプトが存在しますが、PowerShell 側のスクリプトを例として紹介します（リスト 2.7）。

▼リスト 2.7　Android ビルドスクリプト (一部)

```
param(
    [parameter(Mandatory=$true)][String]$UserlibDirectory,
    [parameter(Mandatory=$true)][String]$AssetDirectory,
    [parameter(Mandatory=$true)][String]$AppPackageID
)

...
```

```
function RewriteAndroidFiles {
    $template = Get-Content $ScriptPath\apkbuild\app\build-template.gradle
    $template = $template.Replace("**APKAPPID**", "'$AppPackageID'")
    $template = $template.Replace("**ASSETDIR**", `
        "'$((Resolve-Path $AssetDirectory).Path.Replace("\", "\\"))'")
    $template | Set-Content $ScriptPath\apkbuild\app\build.gradle

    $template = Get-Content `
        $ScriptPath\apkbuild\app\src\main\AndroidManifest-template.xml
    $template = $template.Replace("**APKAPPID**", "$AppPackageID")
    $template |
        Set-Content $ScriptPath\apkbuild\app\src\main\AndroidManifest.xml `
        -Encoding UTF8
}

Sync-Userlib $UserlibDirectory $ScriptPath
robocopy $UserlibDirectory\android-res `
        $ScriptPath\apkbuild\app\src\main\res /mir
RewriteAndroidFiles

...
```

`RewriteAndroidFiles`関数のなかで、あらかじめ用意された build.gradle と AndroidManifest.xml のテンプレートを編集しています。実のところ、これは Unity がやっていることとほとんど同じです。ビルドスクリプトの引数として`-AppPackageID`を指定することで、指定されたパッケージ ID を build.gradle と AndroidManifest.xml に設定します。

AndroidManifest.xml で使用する文字列リソースなどのリソース定義については、ユーザーコード内 android-res フォルダ以下をそのままミラーリングするようにしています。こちらも Unity の仕組みを参考にしました。独自形式を用意せず Android の仕組みに乗ることの利点として、ミドルウェアが簡潔になることと、今後新しくプラットフォームに対応する事になった際に既存の独自形式にロックインされてプラットフォームの良さを最大限活かせなくなるような事態を避けられることが挙げられます。また、プラットフォームの標準レイアウトに従うことで、すでに存在するノウハウをある程度適用できるという利点もあります。

2.7 終わりに

本章で紹介した取り組みにより、Rust コードとしては一切手を加えることなく複数のプラットフォームに対して展開できるバイナリを作成することが可能になりました。すでに Rust 自体は複数プラットフォーム向けのコードを出力できますので、

1. プラットフォーム別の API/ABI をうまく抽象化する
2. それ以外の部分（署名やアセットなど）をシェルスクリプトで処理する

の 2 つを追加して補助してやることで目標を達成することが可能です。

本章では、複数の技術を組み合わせて目標を達成する話の一例を紹介しました。本章が、複雑な自動化機構を考える際の参考になれば幸いです。

2.8 出典

The Rust Programming Language

https://doc.rust-lang.org/book/

Rust By Example

https://doc.rust-lang.org/rust-by-example/

Rust Forge

https://forge.rust-lang.org/

第 3 章

QR コードマニアックス — 数字・英数字・漢字モード

Daisuke Makiuchi / @makki_d

QR コード[*1]は、いまや誰もが知っている 2 次元バーコードとなりました。日本国内でもこの数年で何社も QR コードを利用した決済サービスを提供しはじめており、大規模なキャッシュバックキャンペーンなどでも世間を賑わせています。

ところで、この QR コードについてみなさんはどれだけ知っていますか？ 決済サービス以外にも URL の共有に利用されることが多いため、何らかの文字列を表現できるものだと思っている人が多いと思います。実際そのとおりで、QR コードには主に文字列情報が格納されています。

そこからも想像できるとおり、QR コード自体に決済機能があるわけではありません。QR コードから取引情報の文字列を読み取ったアプリケーションが、その情報を元に個別の処理を行っています。つまり、決済サービスにとって QR コードである必要はまったくないのです。読み取り精度の高さや集積度、表現できる文字種のバランスがよいため、QR コードが選ばれているのでしょう。

この章では、QR コードに文字列がどのように格納されているか具体的に見ていきたいと思います。

3.1 QR コードのモード

QR コードにはデータの種類を示す「モード」があり、モードを切り替えながらデータを格納するようになっています。また、データをどのようなビット列に符号化するのかも、モードごとに決められています。

ここでは代表的な 4 つのモードについて、具体的にどのようなデータがどのようなビット列として表現されるのかを紹介していきます。

[*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

3.2 8 ビットバイトモード

まず最初に紹介するのは 8 ビットバイトモードです。その名のとおり 1 バイトをそのまま 8 ビットとして表現するモードで、後述する他のモードに比べて空間効率がよくないモードです。そのかわり、原理的にどんなデータでも格納することができます。

ひとつ注意すべき点として、デフォルトの文字コード解釈が実装によってまちまちだということです。実は、JIS の規格（JIS X 0510）にはデフォルトの文字コードは ISO-8859-1（Latin1）と書かれていますが、ISO の規格（ISO/ICE 18005:2000）では JIS X 0201（JIS8）と書かれていて、混乱の元になっています。

どちらの文字コードも ASCII コード部分は互換がありますので[*2]、その範囲内の文字だけを使うのであれば問題ありません。非 ASCII コードを含む文字列を格納する場合は、ECI モード[*3]によって文字コードを明示したほうがよいです。

3.3 数字モード

このモードでは、0〜9 の数字だけからなる文字列が格納できます。格納する文字列を 3 文字ずつに区切り、それぞれを 10 ビットで表現します[*4]。つまり、1 文字あたり 3.33 ビットという超高密度で記録できるモードです。

たとえば「1234567890」を符号化してみましょう。まず 3 文字ずつ、"012" "345" "678" "9" に分割してから、それぞれを 10 ビットで表現します。モードを表すヘッダも含めるとリスト 3.1 のようなビット列を格納することになります。

▼リスト 3.1　数字モードでのビット列

```
0001            (数字モード)
0000001010      (文字数=10)
0000001100      (012)
0101011001      (345)
1010100110      (678)
1001            (9)
```

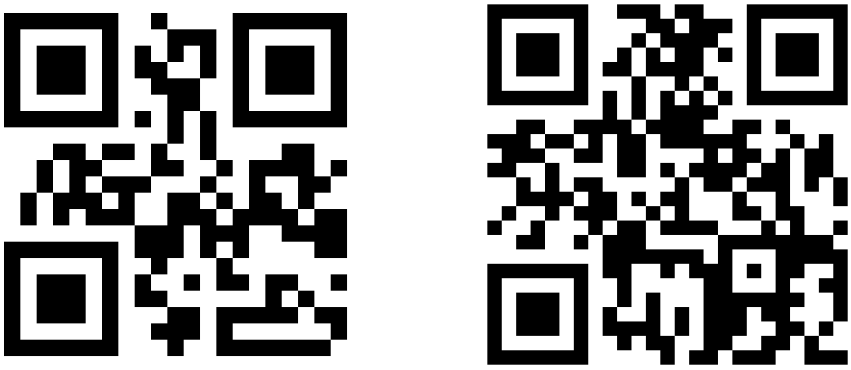
8 ビットバイトモードで同じ文字列を表す場合、それぞれの数字を ASCII コードで表現するのでリスト 3.2 のようなビット列を格納することになります。

▼リスト 3.2　8 ビットバイトモードでのビット列

[*2] 厳密には、5C を「\」とするか「¥」とするかという違いがあります
[*3] 拡張チャネル解釈（Extended Channel Interpretations）モード。詳細は省きますが、これによって文字コードを指定できます
[*4] 末尾に余った桁がある場合、2 文字なら 7 ビット、1 文字なら 4 ビットで表します

```
0100            (8 ビットバイトモード)
00001010        (データ長=10)
00110000        (ASCII ‘0’)
00110001        (ASCII ‘1’)
00110010        (ASCII ‘2’)
00110011        (ASCII ‘3’)
00110100        (ASCII ‘4’)
00110101        (ASCII ‘5’)
00110110        (ASCII ‘6’)
00110111        (ASCII ‘7’)
00111000        (ASCII ‘8’)
00111001        (ASCII ‘9’)
```

これらの QR コードを図 3.1 に示します。8 ビットバイトモードでは数字モードよりも格納するデータ量が多いため、最小のドット数の QR コードに収まらず、1 段階ドット数の多いものにしました[*5]。



▲図 3.1　数字モードの QR コード（左）と 8 ビットバイトモードの QR コード（右）

3.4　英数字モード

このモードは、数字に加えてアルファベットの大文字、スペース、8 つの記号[*6]の計 45 種の文字を格納できるモードです。

45 種類の文字を 0〜44 にマッピングし、先頭から 2 文字ずつ、次のように計算した値を格納していきます[*7]。

[*5] エラー訂正レベルを下げることでも格納データ量は増やせますが、読み取りエラー耐性は下がります

[*6] $ % * + - . / : の 8 種類

[*7] 1 文字余った場合は、その文字を最後に 6 ビットで表現します

1 文字目の値 $*$ 45 + 2 文字目の値

この値の最大値は $45 * 45 = 2025$ なので 11 ビットに収まります。つまり 1 文字あたり 5.5 ビットとなり、8 ビットバイトモードにくらべて 45% 以上多くの情報を格納できます。

それでは、同じサイズ、同じエラー訂正レベルの QR コードに「QR-CODE/」を何回格納できるか、英数字モードと 8 ビットバイトモードで比べてみましょう。

英数字モードでは「QR-CODE/」はリスト 3.3 のように符号化されます。

▼リスト 3.3　英数字モードでのビット列

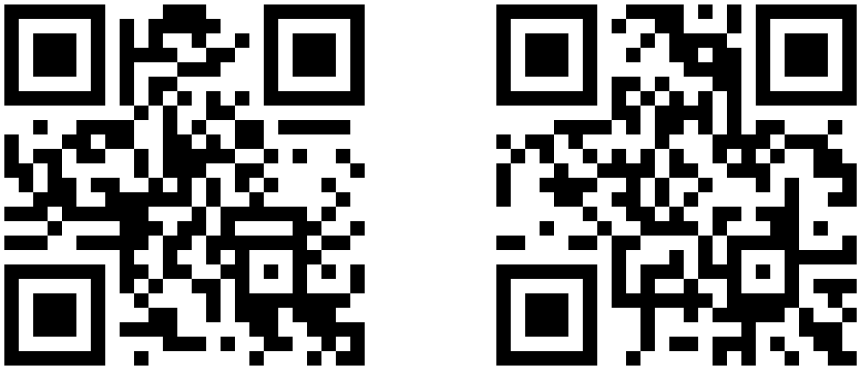
```
10010101101     (‘Q’=26, ‘R’=27, 26*45+27=1197)
11101000001     (‘-’=41, ‘C’=12, 41*45+12=1857)
10001000101     (‘O’=24, ‘D’=13, 24*45+13=1093)
01010100001     (‘E’=14, ‘/’=43, 14*45+43=673)
```

一方 8 ビットバイトモードでは、ASCII コードをそのままリスト 3.4 のように符号化します。

▼リスト 3.4　8 ビットバイトモードでのビット列

```
01010001        (ASCII ‘Q’)
01010010        (ASCII ‘R’)
00101101        (ASCII ‘-’)
01000011        (ASCII ‘C’)
01001111        (ASCII ‘O’)
01000100        (ASCII ‘D’)
01000101        (ASCII ‘E’)
00101111        (ASCII ‘/’)
```

これを格納した QR コードを図 3.2 に示します。読み取っていただくと分かるとおり、英数字モードでは「QR-CODE/QR-CODE/QR-C」まで格納できていますが、8 ビットバイトモードでは「QR-CODE/QR-COD」までしか格納できません。



▲図 3.2　英数字モードの QR コード（左）と 8 ビットバイトモードの QR コード（右）

3.5　漢字モード

不思議に思うかもしれませんが、世界中で使われている QR コードの規格には、日本語の文字を格納するためのモードがあります。QR コードを発明したのは株式会社デンソーの開発部門（現：株式会社デンソーウェーブ）で、日本生まれなのです。

Shift_JIS の 2 バイト文字は、1 バイト目が 81〜9Fと E0〜EF、2 バイト目が 40〜FC[*8]の範囲にコードされています。1 バイト目の値を詰めると 00〜2Eで表すことができ、また 2 バイト目から 40を引くと 00〜BCで表すことができます。1 バイト目から計算した値に C0を掛けたものを 2 バイト目から計算した値に足して記録すると、その値を C0で割った商から 1 バイト目、余りから 2 バイト目を取り出すことができます。

このモードでは EBBFまでの文字を格納することができ[*9]、最大の値は 2A*C0 + BF-40 = 1FFFなので、13 ビットで表すことができます。つまり、8 ビットバイトモードに比べて 1 文字あたり 3 ビットお得になります。

それでは、「技術書典」を漢字モードで格納してみましょう。それぞれの文字を符号化するとリスト 3.5 のようになります。

▼リスト 3.5　漢字モードでのビット列

[*8] 7Fも使われていませんが説明を簡単にするために省いています

[*9] この範囲を超える文字は漢字モードで格納することができませんが、JIS 第 2 水準漢字までを十分にカバーしています

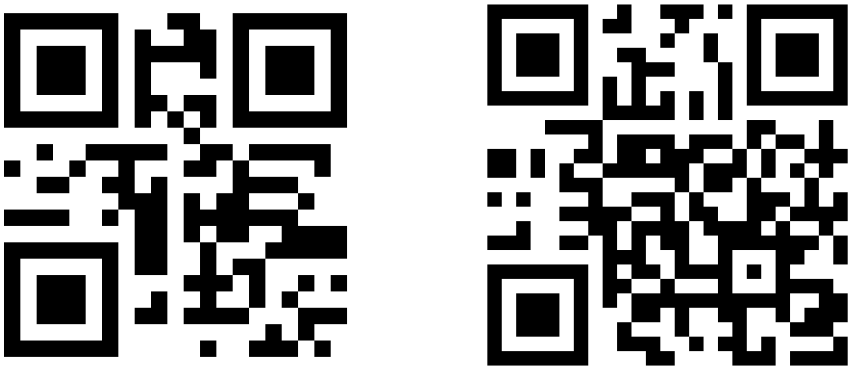
```
1000              (漢字モード)
00000100          (文字数=4)
0011110011010     (‘技’=8B5A, (8B-81)*C0+(5A-40)=079A)
0101010110000     (‘術’=8F70, (8F-81)*C0+(70-40)=0AB0)
0101011010001     (‘書’=8F91, (8F-81)*C0+(91-40)=0AD1)
0110110010100     (‘典’=9354, (93-81)*C0+(54-40)=0D94)
```

これまで同様、8 ビットバイトモードで符号化した場合はリスト 3.6 のようになります。文字コードは UTF-8 でもよいのですが、漢字モードに合わせて Shift_JIS にしました。ECI モードで Shift_JIS であることを明示もしています。

▼リスト 3.6　8 ビットバイトモードでのビット列

```
0111                    (ECI モード)
00010100                (Shift_JIS=20)
0100                    (8 ビットバイトモード)
00001000                (データ長=8)
1000101101011010        (‘技’=8B5A)
1000111101110000        (‘術’=8F70)
1000111110010001        (‘書’=8F91)
1001001101010100        (‘典’=9354)
```

これらを格納した QR コードを図 3.3 に示します。8 ビットバイトモードのデータ量は最小ドット数の QR コードには収まらないので、ドット数の多いものに格納しています。



▲図 3.3　漢字モードの QR コード（左）と 8 ビットバイトモードの QR コード（右）

3.6　まとめ

今回は、QR コードにはデータをより効率よく格納するためのモードが備わっていることを紹介しました。

しかし現在広く使われている URL を格納するという用途には、これらの空間効率のよいモードは表現できる文字種が足りず、8 ビットバイトモードを使うしかありません。

そして、QR コードといえば URL という使われ方があまりにも浸透してしまったため、8 ビットバイトモードにしか対応していない手抜き実装な QR コードリーダーも存在するらしいです。さらにいえば、8 ビットバイトモードにはデフォルトの文字コードが規格によって異なるという問題もあります。

誰もが知っている QR コードですが、本当に誰でも読み取れるようにするためには、これらの事情を考慮しなくてはいけません。それでも QR コードの読み取りやすさと十分な表現力は、これからもさまざまなサービスで利用されていくでしょう。

QR コード関連の実装をする際にはちょっとだけ思い出していただけると幸いです。

某コミュニケーションツールの QR コードリーダー

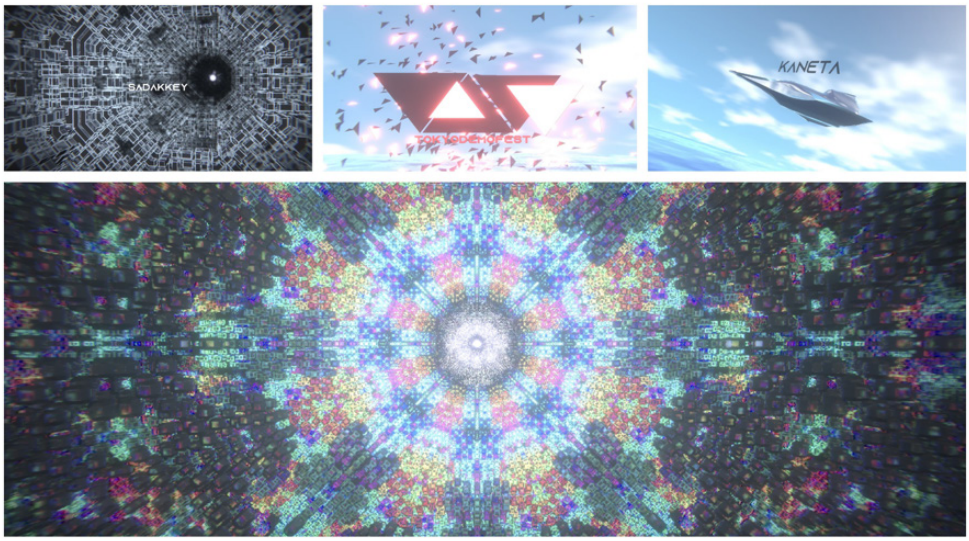
おそらく多くの日本人のスマートフォンに入っているであろう某コミュニケーションツールにも QR コードリーダーが付属しており、これを常用している人も少なからずいるのではないでしょうか。
しかし、その QR コードリーダーでこの章に掲載した QR コードを読み取ると、対応していないフォーマットとしてエラーになると思います。
こちらで調べた限りでは、その QR コードリーダーは「http://」など特定の文字列で始まっているもの以外はエラーとして扱うようです。ご注意ください。

第 4 章

Unity ×レイマーチングによる映像制作の実践手法

Sho HOSODA / @gam0022

本章では、筆者が Tokyo Demo Fest 2018[*1]で発表したリアルタイム CG 作品『WORMHOLE』[*2]を題材にして、Unity と**レイマーチング**を組み合わせた映像制作技術を解説します。



▲図 4.1　WORMHOLE by gam0022 & sadakkey

[*1] Tokyo Demo Fest 2018 公式サイト http://tokyodemofest.jp/2018/
[*2] 『WORMHOLE』の YouTube 動画 https://www.youtube.com/watch?v=k5MotEfghjQ

4.1 Tokyo Demo Fest について

Tokyo Demo Fest（以下、TDF）は、日本国内で唯一のデモパーティです。コンピュータを用いて作成された楽曲や映像作品をデモと呼び、デモに関心のある人々が一堂に会してコンペティションを行ったり、技術を共有したりといったイベントをデモパーティと呼びます。

4.2 題材『WORMHOLE』の概要

「ワームホールによる空間移動」をコンセプトとして、不思議な球体がワームホールを介して非現実なデジタル空間と水平線の広がる自然空間を行き来するリアルタイムの CG 作品です。

不思議な球体がトンネルを進んでいくと、周囲を明滅する光がだんだんとモノクロからカラフルに変わっていきます。トンネルの最奥にあるワームホールへ近づくほど明滅はだんだんと激しくなっていき、ホワイトアウトとともにワームホールを越えると、球体は海上に出現します。その後、球体はじわじわと歪んでいき、戦闘機へと形を変えます。変形中の不思議な球体の上には、私が尊敬するデモシーナーの名前を表示しました。これはグリーティングと呼ばれるデモシーンにおける慣習です。戦闘機はパーティクルを放ちながら海上を進み、パーティクルが一瞬だけ TDF のロゴを形作ります。そして戦闘機は元の球体に変形し、突如現れたワームホールに吸い込まれるようにして冒頭のトンネルのシーンに戻っていきます。

TDF2018 にて、さだきちさん（@sadakkey）とチーム（映像：@gam0022 / サウンド：@sadakkey）を組み、本作品を発表しました。Windows 実行ファイル形式のデモ作品のコンペティションである Combined Demo Compo にて、本作品が 1 位に選ばれました！

実装ならびに制作には Unity を利用しました。詳細は後述しますが、Timeline, TextMeshPro, Chinemachine, PostProcessingStack v2 といった Unity 2018.2 の新機能も活用しています。GitHub にソースコード[*3]を公開していますので、ご興味のある方はそちらもご確認いただければと思います。

4.3 レンダリング

映像の大部分はレイマーチングで描画し、パーティクルやグリーティングのテキストなどのレイマーチングが苦手とする部分はラスタライザで描画するというハイブリッドなレンダリング方式を採用しました。

レイマーチングは、距離関数（3D 空間上の座標を入力すると、物体への最短距離を出

[*3] 『WORMHOLE』のソースコード `https://github.com/gam0022/unity-demoscene`

力する関数）で定義されたシーンに対し、レイの交差を判定する手法です。描画する形状を距離関数からプロシージャルに定義できるため、3D のモデリングなしに 3D シーンを描画できます。容量制限がある部門では大変メリットの大きい特徴です。

加えて、距離関数のパラメータを変更することで描画する形状を制御できるので、レイマーチングでアニメーション表現をすることも可能です。一般的に 3D の映像作品は、モデリングを始め、ボーンやスキンウェイトの設定といった数多くの作業を経て作り上げられるものですが、短いコードから 3D の映像作品を生み出せるのがレイマーチングの魅力と言えるでしょう（もちろん表現のセンスや形状を思い通りに制御するには数学の知識が必須ですが……）。

レイマーチングの基礎知識については、筆者のスライド[*4]にて詳しく紹介しています。

Unity を使う理由

Unity を使う大きな理由として、トライアンドエラーやパメラメータ調整などのイテレーションを高速に回せる点があります。

Unity の ShaderLab でレイマーチングのシェーダーを実装することで、インスペクタからパラメータを微調整しながら見た目を調整できます。

SceneView から好きな視点でレイマーチングによるオブジェクトの形状を確認できるため、距離関数によるモデリング作業のイテレーションを高速化できます。

Unity の通常の映像制作のためのインスペクタや SceneView 等の基本機能は、レイマーチングによる映像制作においても、とても便利で役に立ちます。

ディファードレンダリングによる Unity とレイマーチングの統合

レイマーチングのシェーディングには Unity 標準のディファードレンダリングを利用することにしました。そうすることで、G バッファの書き込みまでを実装すれば、それ以降のライティングの処理を Unity の標準のシェーダーに任せることができます。簡単に言ってしまえば、Unity でサポートされる全種類のライトやグローバルイルミネーションに対応するライティング処理を、あえて自分で実装しなくて済むという工数削減のメリットがあります。

Unity でディファードレンダリングによるレイマーチングを実現するにあたり、@hecomi さんの uRaymarching[*5]を利用させていただきました。uRaymarching は距離関数と G バッファに値を書き込む部分を実装すれば、簡単にレイマーチングができる便利なシェーダーテンプレートです。

他にも、鏡面反射による周囲の映り込みに、Unity 標準の ReflectionProbe を配置して

[*4] シェーダだけで世界を創る！ three.js によるレイマーチング https://www.slideshare.net/shohosoda9/threejs-58238484

[*5] uRaymarching https://github.com/hecomi/uRaymarching

実現しています。ReflectionProbe を配置するだけで苦労せずに反射がレイマーチングにも反映されるのは、ディファードレンダリングによって Unity のレンダリング機能と美しく統合できたためです。

Full Screen Quad の実装方法

uRaymarching の話に関連して、Full Screen Quad の実装方法について紹介します。

uRaymarching では CommandBuffer で Full Screen Quad を表示させていましたが、スクリプトによる制御は最小限にして Unity の Editor モードの挙動を安定させたかったので、別のアプローチをとってみました。

Editor ツールで BoudingBox を巨大にして Frustum Culling を無効にした Quad を静的生成しました[*6]。

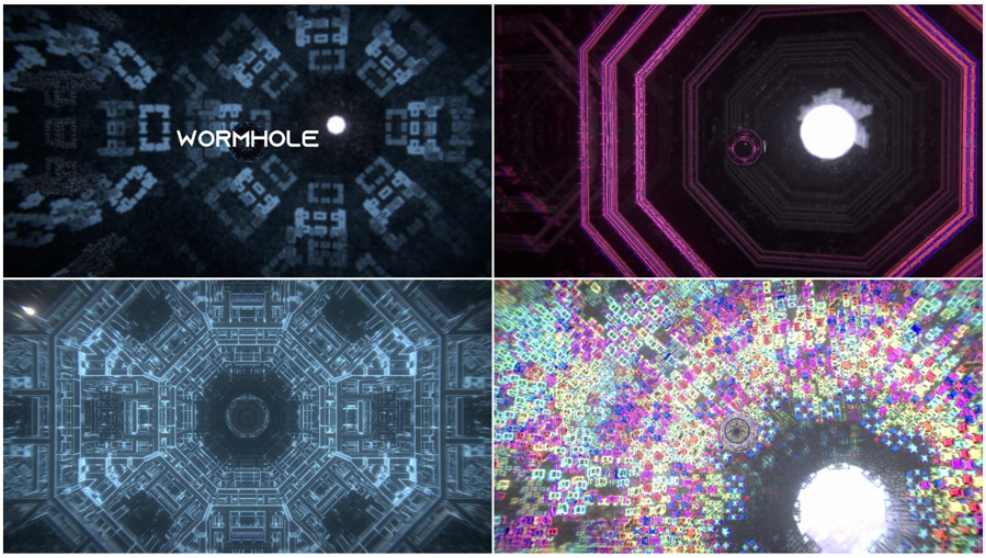
これによって時々レイマーチング部分が動かないトラブルを回避できました。また、本作品のように Full Screen Quad が必要なレイマーチングのワールドが複数存在して、時間によって切り替わる表現のためには、MeshRenderer の enable の切り替えで制御できる単純な仕組みの方が好都合でした。

4.4 距離関数によるモデリング

本作品に登場するオブジェクトの多くは距離関数によってプロシージャルにモデリングしました。本節では距離関数によるモデリングについて解説します。

[*6] Raymarching Quad Mesh Creator `https://github.com/gam0022/unity-demoscene/pull/10`

トンネル



▲図 4.2　様々に変化するトンネル

トンネルは Menger sponge という有名なフラクタル図形をベースにしています。リスト 4.1 のように、ベースとなる Menger sponge の距離関数に対して回転の fold のテクニックを利用して万華鏡のように見せたり、mod をつかった図形の繰り返しのテクニックを適用しました。

回転の fold は筆者のブログの[*7]の記事で紹介しています。

図 4.2 の 4 種類の画像はいずれも同じ距離関数によるトンネルの様子です。パラメータを変化させることで形状や色などを演出に合わせて変更できるようにしました。

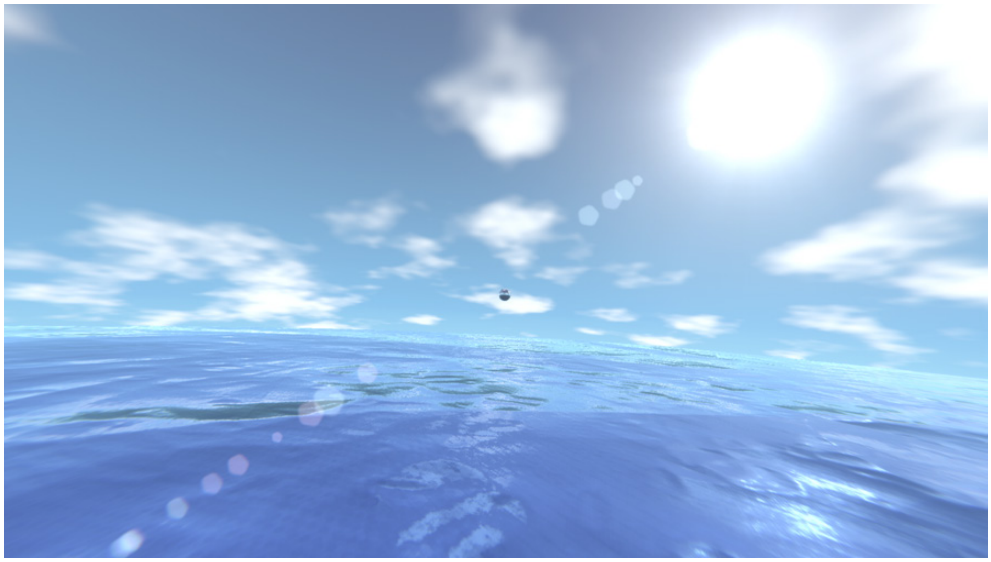
▼リスト 4.1　トンネルの距離関数

```
// Menger sponge の距離関数の定義
float dMenger(vec3 z0, vec3 offset, float scale) {
    vec4 z = vec4(z0, 1.0);
    for (int n = 0; n < 4; n++) {
        z = abs(z);

        if (z.x < z.y) {
            z.xy = z.yx;
        }

        if (z.x < z.z) {
```

[*7] 距離関数の fold（折りたたみ）による形状設計 https://gam0022.net/blog/2017/03/02/raymarching-fold/

```
            z.xz = z.zx;
        }

        if (z.y < z.z) {
            z.yz = z.zy;
        }

        z *= scale;
        z.xyz -= offset * (scale - 1.0);

        if (z.z < -0.5 * offset.z * (scale - 1.0)) {
            z.z += offset.z * (scale - 1.0);
        }
    }
    return (length(max(abs(z.xyz) - vec3(1.0, 1.0, 1.0), 0.0)) - 0.05) / z.w;
}

// 2D の回転行列の生成
float2x2 rotate(in float a) {
    float s = sin(a), c = cos(a);
    return float2x2(c, s, -s, c);
}

// 回転 fold
// https://www.shadertoy.com/view/Mlf3Wj
float2 foldRotate(in float2 p, in float s) {
    float a = PI / s - atan2(p.x, p.y);
    float n = PI2 / s;
    a = floor(a / n) * n;
    p = mul(rotate(a), p);
    return p;
}

inline float DistanceFunction(float3 pos) {
    pos -= float3(2.0, 2.0, 2.0);

    // mod をつかった図形の繰り返し
    pos = Repeat(pos, 4.0);

    // Z 座標に応じた回転
    pos.xy = mul(pos.xy, rotate(pos.z * _MengerTwistZ));

    // 回転 fold の適用
    pos.yx = foldRotate(pos.yx, _MengerFold);

    return dMenger(pos, _MengerOffset, _MengerScale);
}
```

海面



▲図 4.3　海面

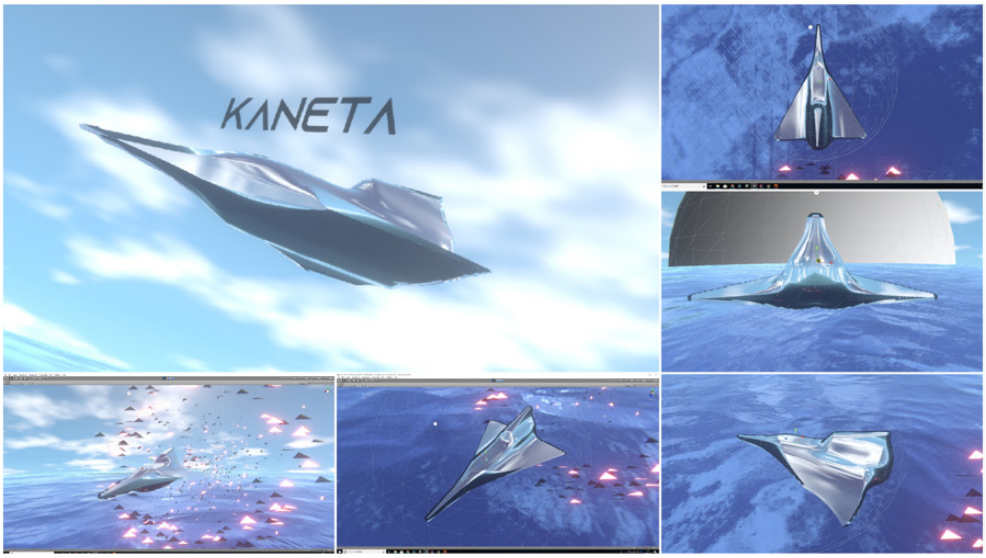
図 4.3 の海面は平面として衝突判定を行い、ノーマルマップだけで波が立っているように見せています。こちらは以前に実装した WebGL 作品[*8]と同じアプローチの軽量化方法です。

ところで、上記の記事の作品と異なり、本作品では LOD（Level of Detail）を一切行っておりません。カメラワーク的に海面に近づかないため、そもそも LOD が必要なかったのと、マーチングループ中でテクスチャのフェッチをすると Unity のシェーダーのコンパイルが激重になる現象を回避するためです。

海面の質感は、G バッファに書き込むパラメータの調整だけで再現しました。ディファードレンダリングなので不透明オブジェクトとして当然ライティングされているのですが、どことなく海中を感じさせるような半透明の質感を擬似的に再現できたのではないかと思います。

[*8] 正解するカドの「カド」をレイマーチングでリアルタイム描画する https://gam0022.net/blog/2017/06/30/raymarching-kado/

戦闘機



▲図 4.4　戦闘機

図 4.4 の戦闘機は距離関数でモデリングしました。

リスト 4.2 のように、3 つの Box の大きさを cos/sin/abs 等で調整しつつ、smoothmin によるメタボールで Box を融合することで、流線形の SF っぽい戦闘機をモデリングしました。

また、フラグメントシェーダーの負荷軽減のために Object Space Raymarching[*9]を行いました。Full Screen Quad を使わずに戦闘機と同じ大きさの Sphere を配置し、Sphere のシェーダーでレイマーチングをしています。上記の画像をよく見ると Sphere のワイヤーフレームを確認できると思います。

▼リスト 4.2　戦闘機の距離関数

```
float _Beat;
float _PlaneRate;

inline float DistanceFunction(float3 pos) {
    float r = 0.05;
    float sphere = Sphere(pos, r);

    float3 plane_offset = float3(0.0, 0.0, 0.0);

    float fbm = 0.2 * cos(PI2 * _Beat / 6) + 0.01 * cos(4.0 * _Beat) +
        0.0025 * cos(16.0 * _Beat);
```

[*9] http://i-saint.hatenablog.com/entry/2015/08/24/225254

```
    pos.xy = mul(pos.xy, rotate(fbm));

    // 胴体の Box
    float nz = pos.z + 0.5;
    float w = 0.03 * abs(sin(0.5 + 2.8 * nz));
    float3 body_size = float3(w, w, 0.4);
    float body = sdBox(pos - float3(0.0, -0.08 * nz, 0.0) - plane_offset,
        body_size);

    // 主翼の Box
    float3 wing_size = float3(0.3, 0.01 * cos(abs(6.0 * pos.x)),
        0.2 * cos(abs(5.0 * pos.x)));
    float wing = sdBox(pos - float3(0.0, -0.06, -0.1 - 0.9 * abs(pos.x)) -
        plane_offset, wing_size);

    // 垂直尾翼の Box
    float3 vwing_size = float3(0.005, 0.1 - 0.3 * nz, 0.1);
    float vwing = sdBox(pos - float3(0.0, 0.06, -0.3) - plane_offset,
        vwing_size);

    // 胴体、主翼、垂直尾翼の 3 つを、smoothmin によるメタボールで合成する
    float plane = sminCubic(body, wing, 0.2);
    plane = sminCubic(plane, vwing, 0.2);

    return mix(sphere, plane, _PlaneRate);
}
```

4.5 演出の実装

演出には TextMeshPro とカスタムシェーダーを組み合わせ、Unity Timeline の Animation Track や Custom Track、パーティクルシステムなどを利用しました。

TextMeshPro によるフォントのレンダリング

冒頭の文字をパラパラと出現させたり消滅させたりする演出は、TextMeshPro とカスタムシェーダーを組み合わせて実装しました。

TextMeshPro は、SDF（Signed Distance Field, 文字の輪郭までの距離を画素値にした画像）をつかって高品質にフォントをレンダリングできるアセットです。Package Manager や AssetStore から無料で入手可能です。

フォントはプロシージャルではなくテクスチャを使用しています。TextMeshPro の Editor ツールを利用して Azonix font のデータ[*10]から SDF のフォントのアトラステクスチャを生成しました。

生成したアトラステクスチャは TextMeshPro のシェーダーでレンダリングしています。

次のような簡単な文字の出現と消滅のエフェクトを、TextMeshPro の標準シェーダーの一部を改造して実装しました。

この演出に関する解説を Unity #2 Advent Calendar 2018 の 19 日目の記事[*11]で行い

[*10] https://www.fontspace.com/mixofx/azonix

[*11] カスタムシェーダーで TextMeshPro に独創的な演出を加える https://qiita.com/gam0022/items/

ました。

Animation Track vs Custom Track

Unity の Timeline ではトラックを自作することができます（以降、自作トラックのことを Custom Track と書きます)。

Custom Track の実装はそれなりに工数がかかります。たとえば、クリップのパラメータを 1 つでも増やすと複数箇所に変更が発生します。工数が限られている場合や試行錯誤しながら色々なパータンを作る場合には、Animation Track では実現できないのかを事前に確認することをお勧めします。

本作品では、レイマーチングのシェーダーのパラメータの制御は Animation Track を利用し、アニメーションでは制御できない TextMeshPro の文字列指定においてのみ Custom Track を利用する方針としました。

パーティクル

パーティクルは Unity の ParticleSystem を利用しました。

図 4.5 はポストエフェクトと Skybox を OFF にした状態でパーティクルをワイヤーフレーム表示したものです。

▲図 4.5　パーティクルのワイヤーフレーム表示

f3b7a3e9821a67a5b0f3

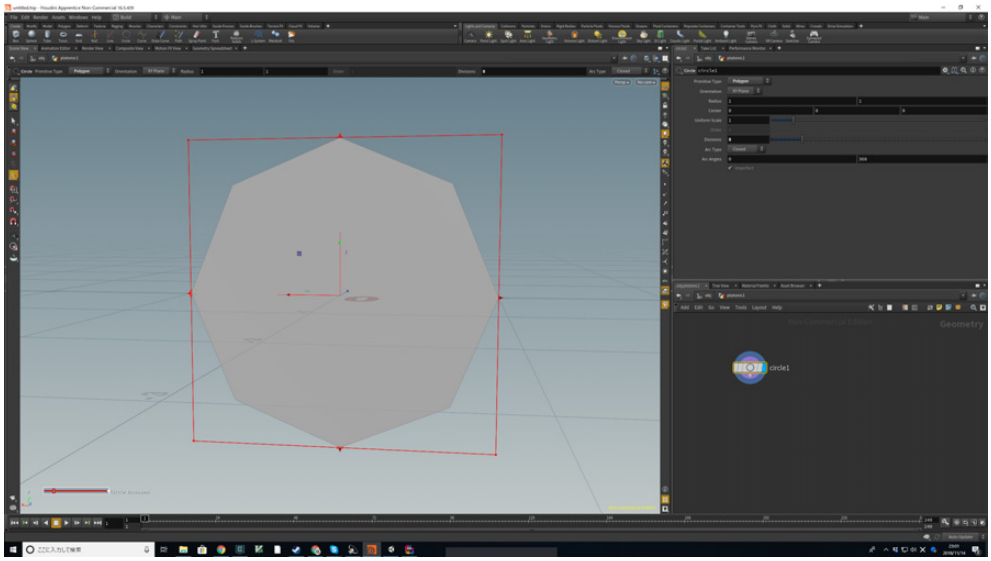
パーティクルの形状は 5 種類でしたが、パーティクル用のモデルは 1 種類しか用意しませんでした。四角形の Quad をフラグメントシェーダーで discard して形状を変化させました。すべてのパーティクルを 1 マテリアルで表現できるので、全パーティクルを 1 ドローコールで描画できました。

4 種類のパーティクルが同時に登場する演出では、Custom Vertex Streams[*12]を用いてランダム値をシェーダーに渡し、シェーダーで形状の切り替えを行いました。

ワームホールの実装

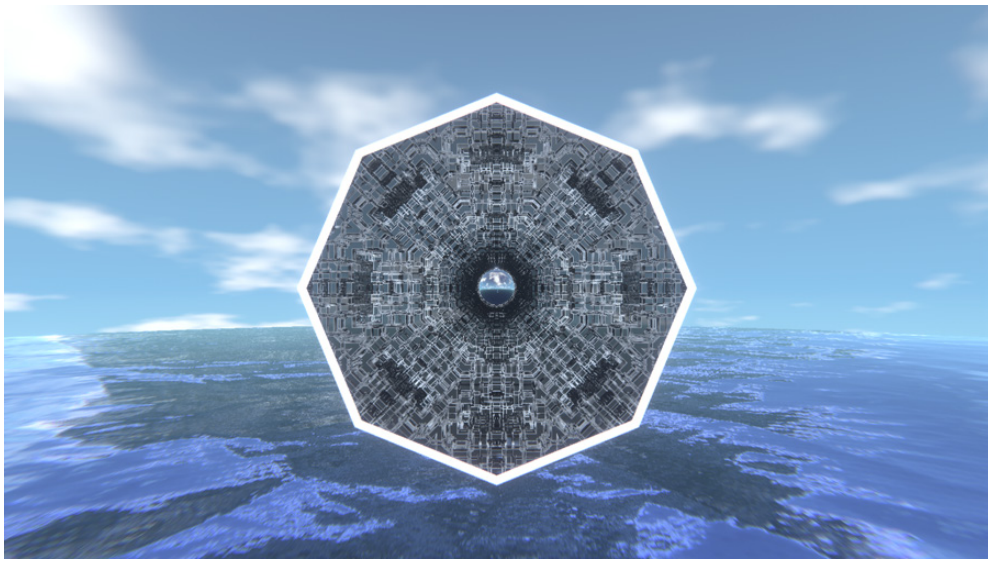
「ワームホールの中身だけ別の世界になる」演出にも戦闘機と同じ Object Space Raymarching の仕組みを利用しました。

まず図 4.6 のように Houdini でワームホールの八角形のポリゴンメッシュを作成しました。



▲図 4.6　Houdini で八角形のポリゴンをモデリングする様子

[*12] 【Unity】Shuriken Particle「Custom Vertex Streams」https://goisagi-517.hatenablog.com/entry/2018/05/15/011845



▲図 4.7　別空間へ繋がるワームホールの演出

八角形のメッシュを描画するシェーダーで Object Space Raymarching を行えば、図 4.7 の別の世界と繋がる演出ができます。

ところが、カメラの原点からレイを進めると、別世界が 3D の立体映像のように飛び出してしまうという罠にハマってしまいました。この問題はレイを物体の表面から進めることで回避できました。

ワームホールの内側は現在の世界（図 4.7 では海の世界）のレイマーチングのシェーダーを無効にしたかったので、Stencil を利用しようとしたのですが、Unity のディファードレンダリングでは Stencil の利用が制限されていました。Unity の公式マニュアル[*13]に次のような記述があります。

> deferred レンダリングパスでレンダリングするオブジェクトのためのステンシル機能はいくらか制限されます。それらの 2 つのステージの間、シェーダーで定義されるステンシルステートは無視され、最終的なパスの間に考慮されるだけです。

そこで、Depth テストと RenderQueue による制御で Stencil を代用しました。

ReflectionProbe の映り込みによる演出

2 度目のワームホール出現時（2:05〜）に海面が黒く侵食されていく演出があります。

これは、ワームホールの向こう側の景色が ReflectionProbe に映り込み、Unity のライティング機能によって自動的に水面に反映された結果です。意図的に演出したものではな

[*13] ShaderLab: ステンシル `https://docs.unity3d.com/ja/current/Manual/SL-Stencil.html`

く偶然の産物でしたが、気に入ったのでこのまま採用しました。

揺らぎ

揺らぎは 2 箇所で利用しました。単純でコストもかからない工夫ですが、効果は大きいと感じました。

カメラに fbm ノイズを加えて手ブレ感を出すことで臨場感が生まれました。

それから、戦闘機を cos 波で振り子のように左右に揺らしています。戦闘機の動き自体は Z 軸に直進するだけなのですが、機体の揺れとカメラワークによって旋回しているような雰囲気が出ているのではないでしょうか。

4.6 音楽との同期方法

本節では音楽との同期方法について紹介します。

作曲については、さだきちさんのブログ記事[*14]をご覧いただければと思います。

ビート単位でのシェーダー制御

シェーダーの入力をビートにし、演出を「ビート単位」で制御することで、映像と音楽を同期させました。時間単位（秒単位）で制御するよりも、BPM 変更に柔軟に対応できるというメリットがあります。

秒数（`time`）を特定の BPM（`bpm`）のビート（`beat`）に変換するには `beat = time * bpm / 60` を計算します。

カメラのカット切り替えやパーティクルの同期

カメラのカットやパーティクルのエミットのタイミングといったシェーダーで制御していない部分は、音楽に合わせて Timeline のクリップを手動で配置する必要がありました。

こちらは音楽を 120BPM で制作していただいたことで、かなり楽に解決できました。

120BPM では、1 ビートが 0.5 秒となります。

4 分の 4 拍子であれば 1 小節の長さが 2 秒となるため、カメラのカット切り替えを 2 秒単位にすると音楽と映像が自然に同期します。同様に、4 分の 3 拍子であればカット切り替えを 1.5 秒単位にすればよいわけです。

パーティクルは、エミット間隔を 0.5 秒ごとに設定することで音楽とタイミングを合わせています。

[*14] Tokyo Demo Fest2018 の Demo Compo 優勝作品の解説〜サウンド編〜 `http://klabgames.creative.blog.jp.klab.com/archives/14415590.html`

4.7 おわりに

『WORMHOLE』の映像を作るための取り組みや手法について技術的な視点で解説しました。

上記のとおり、『WORMHOLE』の制作には Unity の機能やライブラリを多く利用しています。巨人の肩の上に立つことで表現の部分に注力でき、3 週間弱という短い制作期間の中で完成度の高い作品に仕上げることができました。

Unity には初心者〜上級者まで様々なレベルの方を対象とした資料や教材があります。『WORMHOLE』では使用しませんでしたが、たくさんのアセットも用意されています。デモシーンに興味はあるもののハードルが高そうで踏みとどまっている方や、レンダリング技術の学習に挫折してしまった方に、Unity でもデモ作品を作成できることをお伝えしたいです。また、日頃の業務で Unity を利用している方に、自分でも作れそうな身近なものとしてデモシーンに興味を持ってもらえれば嬉しいです。『WORMHOLE』が新たなデモシーナーを生み出すきっかけとなれば幸いです。

第 5 章

自作キーボード入門

Suguru OHO / @ohomagic

先日、秋葉原の遊舎工房[*1]さんで HelixPico[*2]を購入し、ついに自作キーボード道に入門することとなりました。本稿ではその顛末を書いていこうと思います。

5.1 組み立て編

今回作成したのは、キットで販売している HelixPico です。ロープロファイルキースイッチ対応の分離型で小型のキーボードです。組み立てもしやすく入門には適切かと思ったのと、薄くて小さくて軽くて持ち運びが容易という個人的に重視するポイントを押さえていたので、これを選びました。

キースイッチは、Kailh ロープロファイルスイッチ[*3]の白軸にしました。これもカチカチ系メカニカルスイッチ好きとしては外せないポイントです。キートップは刻印ありの黒です。これも好みです。こうして自分の好きなようにカスタマイズできるのも、自作キーボードの魅力ですね。

組み立ては遊舎工房さんの店舗の組み立てコーナーでしました。ハンダゴテなどの道具もひととおり揃っていて、とても快適に製作できました。制作所要時間は、およそ 3 時間ほどでした。電子工作にはそれなりには慣れているので、ほとんど迷わずに完成しました。小さな SMD（表面実装部品）もないので（唯一の SMD は大きめのスピーカーのみ）、難易度としては非常に簡単な方でしょう。ただ、部品点数が多く、ダイオードなどの方向を間違えてはいけないものもあるので、テスターをお借りしてひとつひとつ確認しながらすすめました。

[*1] 遊舎工房さん https://yushakobo.jp/

[*2] 遊舎工房さんの HelixPico 商品ページ https://yushakobo.jp/shop/helixpico/

[*3] 遊舎工房さんの Kailh ロープロファイルスイッチ商品ページ https://yushakobo.jp/shop/pg1350/

> **組み立てコーナーにいろんな人がいて面白かった話**
>
> その日は平日の午後で、最初のうちは客は自分一人だったのですが、しばらくすると満員になり驚きました。鬼のように表面実装部品満載の超小型キーボードを組み立てる人や、修理しに来た人、僕のようにはじめてキーボードを作りに来た人など、さまざまな人がいました。印象的だったのが、海外の人が多かったことです。僕が、これが俺の初自作キーボードだ！ と完成した HelixPico を見せると、次はどんなキーボードを作るんだいとすかさず煽られたのは忘れません。

5.2 無線化編

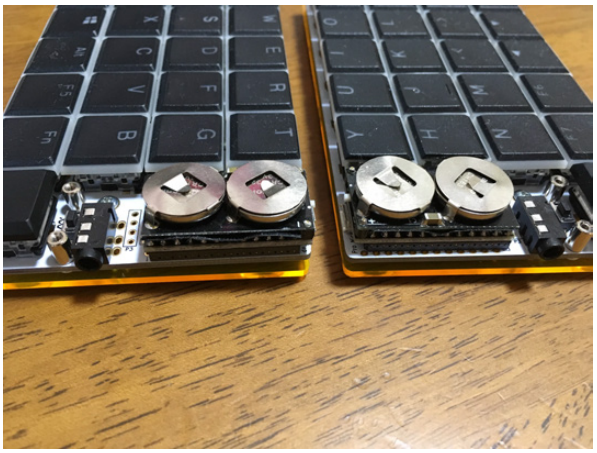
ハードウェアの変更

次に、このキーボードを完全無線化してみました。

▲図 5.1　キーボードの全体イメージ

完全無線化のためには、キットに付属しているマイコンボード（ProMicro）の換装と電源の付加が必要です。BLE Micro Pro[*4]という Bluetooth 無線化機能をもった互換コントローラーに変更し、ボタン電池を載せることのできる電源ボードを追加しました。この電池基板には表面実装のダイオードなどがあり、意外と時間がかかりました。また、BLE Micro Pro 基板に接続する際に、基板同士が短絡しそうだったので、ビニルテープを貼り絶縁しました。幸い、目立たないように上手く収まりました。

[*4] BLE Micro Pro のリポジトリ `https://github.com/sekigon-gonnoc/BLE-Micro-Pro`



▲図 5.2　BLE Micro Pro と電池基板の実装

QMK ファームウェアのビルドと書き込み

ファームウェアも無線対応のものが必要です。HelixPico はマイコンボードに QMK ファームウェア[*5]があらかじめ書き込まれたキットで販売されていましたが、そもそもマイコンボードを換装したので新しいボードに HelixPico 用の無線化したファームウェアを書き込む必要があります。今回は、BLE Micro Pro の作者により無線化のためにカスタマイズされた QMK[*6]を使います。

まずは Windows 上に QMK のビルド環境を構築します。いろいろ調べてみると、msys2 と gitbash が干渉するなどの話があり、それならと、gitbash をアンインストールして、なるべくクリーンな msys2 環境を構築しました。QMK のインストールは README のとおり次のようにしました。

▼リスト 5.1　QMK のインストール

```
$ git clone --depth 1 -b nrf52 https://github.com/sekigon-gonnoc/qmk_firmware.git
$ cd qmk_firmware
$ util/qmk_install.sh
```

いろいろ聞かれますが、大体 Yes 的な確認をして進めました。他にもいくつか必要なツールなどがあるので、BLE Micro Pro 用 QMK の README を参考にしてダウンロードしてインストールします。

[*5] QMK ファームウェアのリポジトリ `https://github.com/qmk/qmk_firmware`
[*6] BLE Micro Pro 用ファームウェアのリポジトリ `https://github.com/sekigon-gonnoc/qmk_firmware`

HelixPico 対応

QMK ファームウェアは汎用の自作キーボード用ファームウェアですが、対象キーボードの選択はビルド時に行います。無線化 QMK ファームウェアの方には無線化された HelixPico 用のビルド設定がないので、既存のサンプルを修正して HelixPico に対応させます。

今回はなるべく簡単に対応するべく、無線化 QMK ファームウェアのサンプルとして同梱されている Helix BLE 用のコードを、Pico 用に少しだけ改変して動くようにしてみます。Helix BLE 用の設定は `qmk_firmware/keyboards/helix_ble/`のディレクトリにあります。

`/keyboards/helix_ble/keymaps/default/rules.mk`の 4 行目を次のように修正して、`HELIX_ROWS`を 4 にしてあげれば、ほぼ HelixPico のキー配置となります。

▼リスト 5.2　Helix Pico 対応のための変更

```
-HELIX_ROWS = 5              # Helix Rows is 4 or 5
+HELIX_ROWS = 4              # Helix Rows is 4 or 5
```

この変更を加えてからビルドし、キーボードに書き込みます。

▼リスト 5.3　ファームウェアのビルド

```
$ export NRFSDK15_ROOT=~/qmk_utils/nRF5_SDK_15.0.0_a53641a
$ make helix_ble/master:nrfutil
$ make helix_ble/slave:nrfutil
```

ファームウェアのビルドは master と slave の 2 回行う必要がありました。make によるビルドの過程で、あらかじめ USB 接続したキーボードへのファームウェアの書き込みまで行ってくれます。

ファームウェアの書き込みができたら Bluetooth での接続を試してみます。まずは接続先の PC の方で Bluetooth 機器の探索モードにします。デフォルトでは、アドバタイジングを開始するコマンド `restart_advertising_wo_whitelist()`が `Adjust+Alt`に割り当てられているので、押下してペアリングを開始します。後は通常の Bluetooth キーボードと同様につながります。

▲図 5.3　無線化 HelixPico が Bluetooth のペアリング画面に表示されたところ

5.3　おわりに

キーボード自作面白いですね。3D プリンタとかも最近では気軽に使えるようになってきているので、さらに変なものも作れそうです。キースイッチをはんだ付けなしで差し替えることができる部品などもあり、非常に興味をそそられます。自分の使う道具を自分で好きなように作るということにも魅かれますし、当分まったりと遊んでいこうと思います。

あなたもおひとつ自作いかがですか？

第 6 章

機械学習 API の力で CAPTCHA を破る

Yoshio HANAWA / @hnw

6.1　はじめに

bot プログラムによる機械的・連続的な入力を防ぐ目的で開発された CAPTCHA という技術があります。人間にしか解けないような問題を解かせて、bot の入力を排除するような仕組みの総称です。

CAPTCHA は誕生当初こそ有用な技術でしたが、もはや並の CAPTCHA では bot を防げない時代になりつつあります[*1]。プログラム技術、特に機械学習の技術が発達した結果、大半の問題で人間よりも機械の方が賢くなってしまったためです。

一方で、機械学習の知識がない普通のエンジニアからすると、CAPTCHA を破るプログラムを一から書けといわれても途方に暮れてしまいます。筆者もそんな一人ですが、機械学習 API を使えば機械学習の知識が無くても CAPTCHA が破れるのではないか？ と考えて試してみました。本稿ではその成果を紹介します。

6.2　CAPTCHA とは

すでに紹介したとおり、CAPTCHA は人間と機械を区別する技術です。アカウント作成の自動化やパスワードリスト攻撃など、bot がビジネス上の脅威になるような場合に利用されます。

CAPTCHA という名前でピンと来ない人でも、Web サイトの会員登録ページやログインページなどで図 6.1 のようなものを見たことがあるのではないでしょうか。これが CAPTCHA です。

[*1] 後述しますが、reCAPTCHA v2 および v3 であれば現在でも bot 対策として有効です。

▲図 6.1　CAPTCHA の例

図 6.1 の CAPTCHA は画像中のテキストを認識して ASCII 文字列を入力させるものです。最近では与えられた画像から自動車だけを選ぶなど、テキスト認識以外の CAPTCHA も増えてきましたが、初期の CAPTCHA はすべてテキスト認識でした。

CAPTCHA の歴史は古く、世界初の CAPTCHA は 1997 年に AltaVista[*2]で開発されました。当時の AltaVista は検索対象の URL を登録するフォームを持っていたのですが、SPAM bot からの大量登録を受けており、これに対抗するためのものでした。

CAPTCHA 誕生当初から画像中のテキストを機械的に読み取られる危険性については検討されていたようで、AltaVista では OCR[*3]のマニュアルに書いてあった認識精度を改善する方法を逆手に取り、次のような画像を作れば OCR 対策になると考えました。

- 複数のフォントを混在させる
- 背景が均一でない
- 文字を歪ませたり回転させたりする

OCR プログラムはそもそも出版物の読み取りを前提としているので、出版物ではありえない条件のテキスト認識は困難だったと考えられます。実際、初期の CAPTCHA は bot プログラムの大半を除去できていたようです。

6.3　CAPTCHA 破りの正攻法

CAPTCHA のテキスト認識は 2000 年頃の技術では困難でしたが、2019 年現在の機械学習の専門家にとって難しい問題ではありません。機械学習の知識がある人が CAPTCHA 破りで最高の精度を出そうと考えた場合、次のような手順でモデル構築をすることになります。

1. 教師データの元となる CAPTCHA 画像を大量に集める（おそらく数百から数千）
2. 必要に応じて画像のノイズ除去など前処理を行う
3. 1 文字ごとに画像を切り出し、文字ごとにラベル付けを行う
4. 必要に応じてデータの水増し（data argumentation）を行う

[*2] 今は亡き検索エンジン。Google 登場以前は最大手のひとつだった。
[*3] Optical Character Reader。印刷された文章を読み取ってテキストファイルにする装置。かなり古くから製品が作られている

5.「学習」：モデル構築を行う。大量の GPU 計算が必要。
6.「評価」：精度を評価する。イマイチだったら 1 または 2 からやり直し。

つまり、教師データの収集・作成を人力で行い、一定以上の計算リソースを投入して、画像処理の知識も駆使すれば確実に破れる、ということになります。とはいえ、教師データを作るコストとチューニングのコストは専門家でもタダにはなりません。専門家が取り組むにしても、人の時間と GPU 時間とそれぞれ一定のコストがかかるというわけです。

6.4 手軽に CAPTCHA を破りたい！

さて、私のように機械学習分野の専門知識もなく、よい GPU も持っていない人間が CAPTCHA 破りに取り組む方法はあるのでしょうか。

ひとつは既存の OCR ソフトウェアを使うことです。定評のある OCR ソフトウェアとして Tesseract OCR[*4]という OSS があります。実際に試したわけではないので CAPTCHA 破りに使えるかはわかりませんが、状況次第では有力な選択肢になるはずです。

さらにお手軽な方法として、機械学習 API を利用する手があります。クラウド事業者の多くがテキスト認識 API を提供しており[*5]、いずれも安価に利用することができます。また、環境構築のコストがほぼゼロなのも API のよいところでしょう。

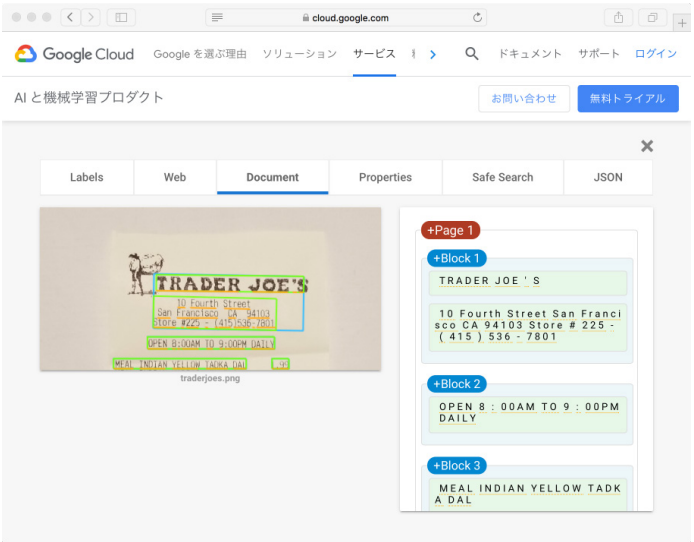
これらのツールや API では CAPTCHA を破るような使い方は想定していないはずですから、過度の期待はできません。とはいえ、多少精度が低くても手軽に CAPTCHA 破りができることがわかれば、それはそれで価値があるはずです。そのような考えから、今回 Cloud Vision API を使った CAPTCHA 破りに挑戦してみました。

Cloud Vision API とは

Cloud Vision API は Google Cloud Platform （GCP）上で提供されている画像分析 API で、画像のカテゴリ分類、テキスト認識など複数の機能をもつ API です。

この API は GCP にアカウントがある人なら誰でも利用できます。また、デモ利用であればアカウント無しで試すこともできます。Cloud Vision API のページ（`https://cloud.google.com/vision/`）がデモページになっているので、興味のある読者の方は試してみてください[*6]。

[*4] `https://github.com/tesseract-ocr/tesseract`
[*5] たとえば AWS の Amazon Rekognition （`https://aws.amazon.com/jp/rekognition/`）や Azure の Computer Vision API （`https://azure.microsoft.com/ja-jp/services/cognitive-services/computer-vision/`）など。
[*6] デモ利用では言語のヒント情報を与えることができないので、API を使った方が認識精度は上がります。



▲図 6.2　Cloud Vision API のデモページでテキスト認識を試した例

気になる利用料金ですが、1000API 呼び出しあたり 1.5 ドルです。CAPTCHA のように短いテキストに対して使うには若干高く思えますが、本来は書籍 1 ページをスキャンしてテキスト化するような用途で使うものですから、むしろ破格と言ってよいでしょう。

API 単体で CAPTCHA は破れるか

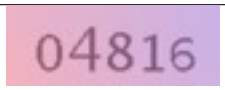
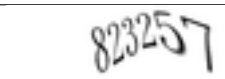
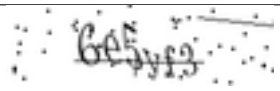
CAPTCHA 破りをするには対象となる CAPTCHA が必要です。今回は表 6.1 の 6 サイトについて CAPTCHA 画像を 20 個ずつ集め、Vision API でテキスト認識してみました。その結果が表 6.2 です。

サイト A とサイト D については正解率が 100% 近い結果になりました。サイト A はパッと見で簡単そうなので納得ですが、サイト D は一見難しそうに見えるので意外な結果です。両サイトとも数字しか出現しないので、その分簡単なのかもしれません。

サイト B とサイト C は約 50% の正解率になりました。機械学習の専門家ならもっと精度を上げられそうですが、CAPTCHA 破りとしては十分な精度です。というのも、CAPTCHA を使った入力フォームでは通常リトライができるので、1 回の正解率が 50% なら 5 回以下で突破できる確率は 96.8% ということになるからです。

サイト E とサイト F についてはほぼ 0% で、実用的とは言いがたい結果になりました。この 2 サイトについては次章で深追いしてみましょう。

▼表 6.1　対象サイトと CAPTCHA 画像例

	画像例
サイト A	
サイト B	
サイト C	
サイト D	
サイト E	
サイト F	

▼表 6.2　Cloud Vision API による CAPTCHA 破りの結果

	正解率
サイト A	100%
サイト B	55%
サイト C	55%
サイト D	95%
サイト E	0%
サイト F	10%

6.5　ノイズ除去で認識精度を上げる

一般に、機械学習では入力データのノイズ除去や正規化といった前処理も重要になってきます。特に今回は OCR 機能を利用しているので、元画像を OCR 向きの画像に加工することで精度改善ができそうです。

「6.4 手軽に CAPTCHA を破りたい！」で精度が出なかったサイト E・サイト F の画像に共通することとして、文字だけでなくドットや直線などのノイズが加えられているという特徴があります。このノイズを除去できれば正解率が上がるかもしれません。

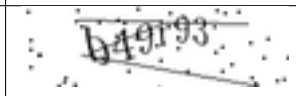
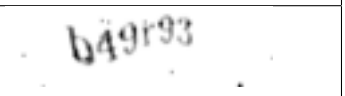
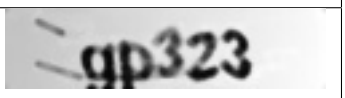
そのような考えから、OpenCV を使ってノイズ除去プログラムを作成してみました[*7]。OpenCV といえば顔認識のライブラリだと考えている人も多いかもしれませんが、実際

[*7] https://github.com/hnw/cloudvision-captcha-solver

は画像処理全般のライブラリで、さまざまな画像処理関数が用意されています。

与えられた画像を加工した結果が表 6.3 です。試行錯誤を繰り返した結果、サイト E はメディアンフィルターを 2 回適用するのが一番よさそうだと考えました。また、サイト F についてはモルフォロジー変換を行っています。どちらも文字は読みにくくなっているものの、ノイズの大半が除去できました。

▼表 6.3　ノイズ除去の適用前・適用後の画像例

	元画像	ノイズ除去後
サイト E		
サイト F		

すべての画像についてノイズ除去プログラムを適用した上で Cloud Vision API を使ってみたところ、20%〜25% ほど精度が向上しました （表 6.4)。実用性の観点ではもう少し改善したいところです。

▼表 6.4　ノイズ除去の適用前・適用後の画像例

	元画像	ノイズ除去後
サイト E	0%	20%
サイト F	10%	35%

このように、サイトごとに特化したノイズ除去を行えば認識精度を改善できることがわかりました。問題に特化した前処理を行った上で汎用の機械学習 API を利用するような使い方は今後どんどん増えていくはずですから、その意味でも面白い結果が得られたように思います。

6.6　reCAPTCHA を使おう

本稿では破る側の視点で CAPTCHA について考えてきましたが、サイト運営者の視点では CAPTCHA をどう考えるべきなのでしょうか。

まず言えることは、CAPTCHA を自作してはいけないということでしょう。本稿で示したとおり、いまや簡単なテキスト認識 CAPTCHA であれば私のような門外漢でも容易に破れる状況であり、知識を持った攻撃者の前では大抵の CAPTCHA は無力です。現在でも有効な CAPTCHA を作れるのは一部の専門家だけと考えるべきでしょう。

では、bot を防ぎたい場合に現実解はあるのでしょうか。

実は機械学習のプロでも破るのが難しい CAPTCHA を Google 社が無料で提供してい

ます。それが reCAPTCHA v2 および v3 です[*8]。reCAPTCHA v2（図 6.3）は画像を選ばせるタイプの CAPTCHA ですが、機械学習方面の知見が盛り込まれており[*9]、日々改良されています。また、v3 はページ遷移に注目する CAPTCHA で、ページ遷移の情報だけを基に人間かどうかを判定するようです。アカウント大量作成 bot のようなものは人間の挙動とは随分違うはずですから、効果は大きそうですね。

▲図 6.3　reCAPTCHA v2 の例

6.7　まとめ

- Cloud Vision API で CAPTCHA が破れた
  - 単純なテキスト認識 CAPTCHA なら正解率 100%
- 元の画像を加工して OCR が得意そうな画像に書き換えると精度が上がった
  - ノイズ除去は大事
- CAPTCHA を自作してはいけない
  - 専門知識なし GPU なしで CAPTCHA が破れる時代
  - reCAPTCHA を使うべき

[*8] `https://developers.google.com/recaptcha/`

[*9] 機械学習モデルを騙すテクニック（Adversarial example）が使われていたり、機械学習モデルの成績が悪い画像をわざわざ混ぜ込んだりしている気配を感じます [要出典]

第 7 章

Unity での条件付きコンパイルシンボル定義にエディタ拡張を活用する

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。前職では iOS アプリ（Objective-C）や PSVita のゲーム（C++）などの開発を経験し、KLab に入ってからは Unity（C#）での開発、特にビルドのマシンやジョブの管理や、OS ネイティブ周りの機能を担当しています。

私が Unity の C#でのゲーム開発に移ってやりづらさを感じたのは、プリプロセッサによる条件付きコンパイルのシンボル定義です。

C++ などでは、リスト 7.1 のように、SYMBOL1 が定義されていたら SYMBOL2 と SYMBOL3 を定義するようなことがよく行われます。この方法のメリットは、シンボルの定義をカテゴリなどの大きな単位でまとめて定義できる点です。

▼リスト 7.1　C++ などの場合のシンボル定義

```
#if SYMBOL1

#define SYMBOL2
#define SYMBOL3

#endif
```

条件付きコンパイルでは、たとえば機能の実装コードをシンボル定義の有無で切り変わるようにしておくことで、ビルド時にシンボル定義を切り替えるだけで機能を簡単に入れ替えられるようになります。しかし、バイナリのビルド時に必要なシンボルを何個も把握しなくてはならないのは、あまり良い状態ではありません。大きな括りでシンボルを定義して、ごそっと機能を入れ替えできるのがリスト 7.1 の利点です。

一方、Unity でプリプロセッサで適用可能なシンボルを定義する方法は次の 3 つです。

1. Player Settings で各プラットフォーム毎に定義する[*1]（リスト 7.2）
2. /Assets/csc.rspファイルで定義しておく[*2]（リスト 7.3）
3. C#ファイルで直接定義する（リスト 7.4）

▼リスト 7.2　PlayerSettings の記述

```
SYMBOL1;SYMBOL2;SYMBOL3;...
```

▼リスト 7.3　/Assets/csc.rspファイルの記述

```
-define:SYMBOL1;SYMBOL2;SYMBOL3;...
```

▼リスト 7.4　C#ファイルの記述

```
#define SYMBOL1
#define SYMBOL2
#define SYMBOL3
```

しかし、これらの選択肢では、リスト 7.1 のようにカテゴリなどの単位でまとめてシンボルを切り替えることができません。1 と 2 ではすべてのシンボルを列挙する必要がありますし、3 の場合、#defineで定義したシンボルが適用されるのは、定義を記述したファイルに限定されてしまいます。

そんな訳で、少々前置きが長くなりましたが、本章では Unity でプリプロセッサのシンボル定義を柔軟に扱うためのエディタ拡張機能を作ってみたいと思います。早速、進めて行きましょう。

7.1　本章で作成するシンボル情報編集エディタ拡張の概要

本章で作成するシンボル編集エディタ拡張の概要は以下のようになります。

動作環境

- Unity 2018.3.6f
- Mac OS 10.4 (Mojave)

[*1] Unity のメニューの File > Build Settings > Player Settings で開いたインスペクタ画面の Scripting Define Symbols という項目です。

[*2] Unity 5 系では mcp.rspファイルという名前でした。Unity 2018 系で mcp.rspファイルを配置してビルドすると、obsolete（廃止）というエラーが出ます。

特徴

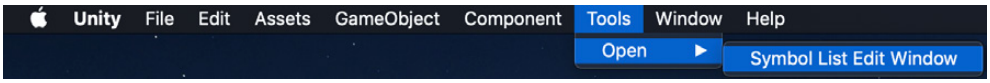
- エディタ拡張のウィンドウの中でシンボルの定義を列挙できる
- あるシンボル（カテゴリ）が定義されるときだけ定義されるシンボルを記述できる
- 列挙しているシンボル定義を有効にしたり無効にしたりできる
- シンボルの情報は外部ファイルに保存して永続化しておける
- 任意のタイミングで `csc.rsp`ファイルへの書き出しができる

7.2　シンボル編集エディタ拡張の利用イメージ

シンボル編集エディタ拡張の利用イメージについて書いていきます。

シンボル編集ウィンドウを開く

Unity のメニューでウィンドウを選択すると、シンボル編集ウィンドウを開くことができます。



▲図 7.1　Unity のメニューでシンボル編集ウィンドウを開く時の表示

ウィンドウを開くとシンボル情報が読み込まれ、定義中のシンボルを全て確認することができます。

▲図 7.2　シンボル編集ウィンドウの表示

この時、カテゴリとなるシンボル（上部タブ）と、そのシンボルが有効になっている時

のみ定義されるシンボル（左側リスト）がそれぞれ表示され確認することができます。

カテゴリを追加する

シンボルの名前とコメントを書いて、追加ボタンを押下すると、カテゴリとなるシンボル情報を追加することができます。

▲図 7.3　カテゴリ追加 UI 表示

シンボルを追加する

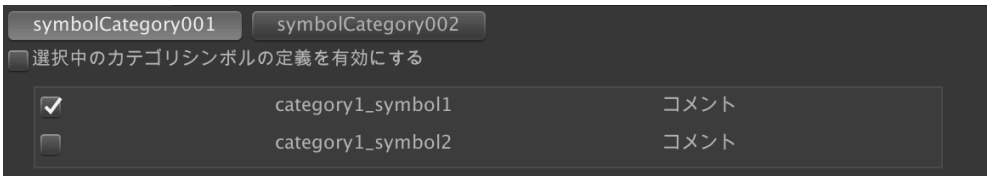
シンボルの名前とコメントを書いて、追加ボタンを押下すると、選択中のカテゴリに紐付けてシンボル情報を追加することができます。

▲図 7.4　シンボル追加 UI 表示

カテゴリ、シンボルの有効状態を変更する

カテゴリとシンボルは、それぞれチェックボックスで有効状態を変更することができます。

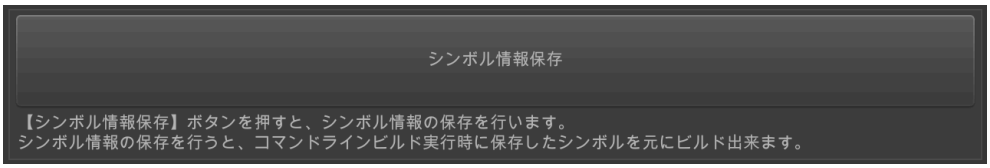
タブでカテゴリを選択することで、選択したカテゴリに紐付いているシンボルが表示されます。「選択中のカテゴリシンボルの定義を有効にする」チェックボックスでカテゴリの有効状態を切り替えることができ、またカテゴリに紐付いたシンボル個々についても一つずつチェックボックスで切り替えることができます。



▲図 7.5　カテゴリとカテゴリでまとめて定義しているシンボル群の個別チェックボックス表示

シンボル情報を保存する

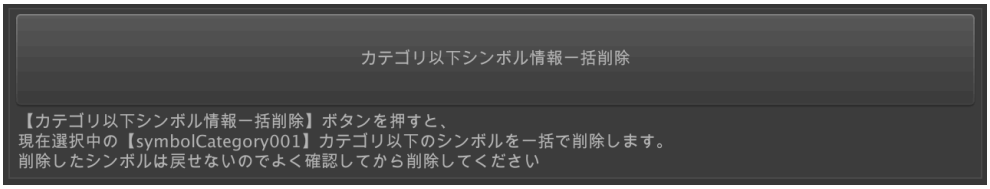
シンボル情報保存ボタンを押すと、外部ファイル csc.rspにシンボルの情報が保存され、エディタ拡張で編集した情報をもとにビルドすることができるようになります。



▲図 7.6　シンボル情報保存ボタン表示

カテゴリを削除する

カテゴリ削除ボタンを押すと、カテゴリ以下で定義されているシンボルを丸ごと削除して定義を掃除することができます。



▲図 7.7　カテゴリ削除ボタン表示

このボタンを押すと、紐づけられているシンボルも全て削除されます。

7.3　シンボル編集エディタ拡張の実装

実際にエディタ拡張を実装して行きます。

外部ファイルの形式

シンボル情報をまとめておくファイルは、エクセルやテキストエディタで編集しやすいCSV 形式にしました。このファイルの構成は以下の通りです。

- カテゴリのシンボルを定義しておくカテゴリシンボル定義ファイル
  - ファイルの数は本エディタ拡張機能を利用するプロジェクトごとに一つ
- カテゴリの中で定義されているシンボルを定義しておくシンボル定義ファイル
  - ファイルの数は本エディタ拡張機能を利用するプロジェクトで定義したカテゴリの数だけ

ファイルの具体例は次のようになります。

▼リスト 7.5　カテゴリシンボル定義ファイル symbolCategoryList.csvの具体例

```
name,enabled,comment
symbolCategory001,true, コメント
symbolCategory002,true, コメント
```

▼リスト 7.6　シンボル定義ファイル symbolCategory001.csvの具体例

```
name,enabled,comment
category1_symbol1,true, コメント
category1_symbol2,false, コメント
```

フォーマットの特徴は一列目がシンボル名、二列目が有効状態（true or false)、三列目がシンボルの意味を説明するためのコメントとなっているところです。

csc.rspファイルは上記のファイル郡の内容を元にビルド前に上書き保存する形になります。

クラスの分担

今回は以下のようなクラスに分けて実装しました。

- ウィンドウクラス
- モデルクラス
- マネージャクラス

ウィンドウクラスの実装

シンボル情報を編集するウィンドウクラス、SymbolListEditWindow の実装を紹介します。

ウィンドウクラスはエディタ拡張でウィンドウを実装するときに使用する EditorWindow を継承させます。そして、Unity のメニューからウィンドウを開くための `Open`メソッドと、ウィンドウ内の UI の表示と挙動を処理する `OnGUI`メソッドを実装します。

このクラスとメソッドを抜粋したものをリスト 7.7 に示します。

▼リスト 7.7　ウィンドウクラスの実装コード：ウィンドウメニュー表示メソッド

```
/// <summary>
/// #define で定義されているシンボルを一覧で表示するウィンドウを管理するクラス
/// </summary>
public sealed class SymbolListEditWindow : EditorWindow
{
    /// <summary>
    /// ウィンドウを開きます
    /// </summary>
    [MenuItem("Tools/Open/Symbol List Edit Window")]
    static void Open()
    {
        // ウィンドウ表示前にシンボル情報をロード
        SymbolListManager.LoadSymbolSystemFile();

        GetWindow<SymbolListEditWindow>("Symbol List Edit");
    }

    /// <summary>
    /// ウィンドウを表示します
    /// </summary>
    void OnGUI()
    {
        EditorGUILayout.BeginHorizontal();
        {
            // スクロールビューの表示を開始します
            mScrollPos = EditorGUILayout.BeginScrollView(
                                mScrollPos,
                                GUILayout.Height(position.height)
                                );

            EditorGUILayout.BeginVertical();

            OnGUICategorySymbol();                  // カテゴリ UI 表示
            OnGUISymbol();                          // シンボル UI 表示
            EditorGUILayout.EndVertical();

            // スクロールビューの表示を終了します
            EditorGUILayout.EndScrollView();

            EditorGUILayout.BeginVertical();
            OnGUIAddCategory();                     // カテゴリ追加 UI 表示
            OnGUIAddSymbol();                       // シンボル追加 UI 表示
            OnGUIRun();                             // 削除・保存 UI 表示
            EditorGUILayout.EndVertical();
        }

        EditorGUILayout.EndHorizontal();
    }

    // ... 略...
```

ウィンドウの実装をしていく際には、基本的に UI をグループ化するように EditorGUILayout.BeginVertical()と EditorGUILayout.EndVertical()で挟む形で記述していきました。コード中の OnGUIXXX()メソッドも OnGUI()メソッドと同じような要領の実装となりますので、実装の詳細は割愛します。

モデルクラスの実装

シンボル情報を管理するモデルクラス、SymbolStatusModel の実装がリスト 7.8 です。このクラスが、前述した CSV ファイル一行分のデータを保持するクラスになります。

▼リスト 7.8　モデルクラスの実装コード：公開プロパティとコンストラクタ

```
namespace EditorScript
{
    /// <summary>
    /// シンボルのステータスと名前の情報を持つモデルクラス
    /// </summary>
    public class SymbolStatusModel
    {
        /// <summary>
        /// シンボルの名前
        /// </summary>
        /// <value>The name.</value>
        public string name { get; set; }

        /// <summary>
        /// シンボルのステータス
        /// </summary>
        /// <value><c>true</c> if enabled; otherwise, <c>false</c>.</value>
        public bool enabled { get; set; }

        /// <summary>
        /// シンボルのコメント
        /// </summary>
        /// <value>The comment.</value>
        public string comment { get; set; }

        /// <summary>
        /// コンストラクタ
        /// </summary>
        /// <param name="name">Name.</param>
        /// <param name="enabled">If set to <c>true</c> enabled.</param>
        /// <param name="comment">Comment.</param>
        public SymbolStatusModel(string name, bool enabled, string comment)
        {
            this.name = name;
            this.enabled = enabled;
            this.comment = comment;
        }
    }
}
```

マネージャークラスの実装

シンボル情報をファイルから読み込み、モデルクラスを生成して管理するマネージャクラスを実装していきます。

SymbolListManagerクラスと、ファイルの読み書き両方で共通利用するインターフェースをリスト 7.9 のように定義します。

▼リスト 7.9　マネージャクラス：ファイルの読み書きで利用する共通インターフェース

```
/// <summary>
/// シンボル情報の CSV ファイルのカラム種別
/// </summary>
enum SymbolColumnType
{
    Name = 0,
    Enabled,
    Comment,
}

/// <summary>
/// シンボルを一覧管理するマネージャークラス
/// </summary>
public static class SymbolListManager
{
    /// <summary>
    /// シンボルリストを配置しているディレクトリのパス
    /// </summary>
    /// <value>The symbol list csv file directory path.</value>
    public static string SymbolListCsvFileDirectoryPath
    {
        get => Application.dataPath + "/Data/";
    }

    /// <summary>
    /// カテゴリシンボルリストのファイルパス
    /// </summary>
    /// <value>The category list csv file path.</value>
    public static string CategoryListCsvFilePath
    {
        get => SymbolListCsvFileDirectoryPath + "/symbolCategoryList.csv";
    }

    /// <summary>
    /// シンボルのカテゴリリスト
    /// </summary>
    public static List<SymbolStatusModel> SymbolCategoryList
    {
        get; private set;
    } = new List<SymbolStatusModel>();

    /// <summary>
    /// シンボルのディクショナリ
    /// </summary>
    public static Dictionary<string,
                             List<SymbolStatusModel>> SymbolDictionary
    {
        get; private set;
    } = new Dictionary<string, List<SymbolStatusModel>>();

    // ... 略...
```

シンボル情報をファイルから読み込む処理はリスト 7.10 のようになります。

▼リスト 7.10　マネージャクラス：シンボル情報ファイルの読み込み

```
    /// <summary>
    /// シンボルのファイルを読み込みます
    /// </summary>
    public static void LoadSymbolSystemFile()
    {
```

```
        // 初期化
        SymbolDictionary.Clear();
        SymbolCategoryList.Clear();

        // csv ファイルを開く：カテゴリシンボル
        SymbolCategoryList = LoadSymbolFile(CategoryListCsvFilePath);

        // csv ファイルを開く：シンボル
        for (var i = 0; i < SymbolCategoryList.Count; i++)
        {
            var path = SymbolListCsvFileDirectoryPath
                            + SymbolCategoryList[i].name
                            + ".csv";
            SymbolDictionary.Add(
                    SymbolCategoryList[i].name,
                    LoadSymbolFile(path));
        }
    }

    /// <summary>
    /// 指定したファイルパスからシンボル情報を読み込みます
    /// </summary>
    /// <returns>The symbol file.</returns>
    /// <param name="path">Path.</param>
    static List<SymbolStatusModel> LoadSymbolFile(string path)
    {
        var list = new List<EditorScript.SymbolStatusModel>();

        // csv ファイルを開く：シンボル
        using (var sr = new System.IO.StreamReader(path))
        {
            var row = 0;

            // ストリームの末尾まで繰り返す
            while (!sr.EndOfStream)
            {
                // ファイルから一行読み込む
                var line = sr.ReadLine();

                row++;

                // 一行目ならスキップ
                if (row == 1) continue;

                // 読み込んだ一行をカンマ毎に分けて配列に格納する
                var values = line.Split(',');

                var entity = new SymbolStatusModel(
                            values[(int)SymbolColumnType.Name],
                            System.Convert.ToBoolean(
                                    values[(int)SymbolColumnType.Enabled]),
                            values[(int)SymbolColumnType.Comment]
                        );

                list.Add(entity);
            }
        }

        return list;
    }
```

シンボル情報をファイルに書き込む処理をリスト 7.11 に示します。CSV ファイルへの保存と csc.rspファイルへの書き出しの両方を行っています。

▼リスト 7.11　マネージャクラス：シンボル情報の保存

```
        /// <summary>
        /// シンボルのファイルを保存します
        /// </summary>
        public static void SaveSymbolFile()
        {
            SaveSymbolCsvFile();

            SaveSymbolRspFile();
        }

        /// <summary>
        /// シンボル情報を CSV ファイルに保存します
        /// </summary>
        public static void SaveSymbolCsvFile()
        {
            // カテゴリシンボルの保存
            SaveSymbolFile(SymbolListCsvFileDirectoryPath, SymbolCategoryList);

            // シンボルの保存
            foreach (var categoryList in SymbolCategoryList)
            {
                var symbolList = SymbolDictionary[categoryList.name];
                var path = SymbolListCsvFileDirectoryPath
                               + "/" + categoryList.name +".csv";
                SaveSymbolFile(path, SymbolCategoryList);
            }
        }

        /// <summary>
        /// シンボル情報を rsp ファイルに保存します
        /// </summary>
        public static void SaveSymbolRspFile()
        {
            var path = Application.dataPath + "/csc.rsp";

            var isFirstAdd = true;
            var symbolDefineString = "-define:";

            foreach (var categorySymbol in SymbolCategoryList)
            {
                // カテゴリの分
                if (categorySymbol.enabled)
                {
                    symbolDefineString = symbolDefineString
                                            + (isFirstAdd ? "" : ";")
                                            + categorySymbol.name;
                    isFirstAdd = false;
                }
                else
                {
                    continue;
                }

                // シンボルの分
                var symbolList = SymbolDictionary[categorySymbol.name];

                foreach (var symbol in symbolList)
                {
                    if (symbol.enabled)
                    {
                        symbolDefineString = symbolDefineString
                                            + (isFirstAdd ? "" : ";")
                                            + symbol.name;
                        isFirstAdd = false;
                    }
                }
            }

            File.WriteAllText(path, symbolDefineString);
        }

        /// <summary>
```

```
        /// 指定したファイルパスにシンボル情報を保存します
        /// </summary>
        /// <returns>The symbol file.</returns>
        /// <param name="path">Path.</param>
        static void SaveSymbolFile(string path,
                                   List<SymbolStatusModel> symbolList)
        {
            // csv ファイルを開く：シンボル
            using (var sr = new System.IO.StreamWriter(path))
            {
                foreach (var symbol in symbolList)
                {
                    string str = symbol.name
                        + ","
                        + symbol.enabled.ToString()
                        + ","
                        + symbol.comment;
                    sr.WriteLine(str);
                }
            }
        }
```

7.4 いざ、ビルド

このようにエディタ拡張を作成したら、あとはビルド時にシンボルを切り替えられるようにしなければなりません。今回は有効になっているシンボルを csc.rspに列挙することでコンパイラに伝えることにします。Unity ではビルドの前処理や後処理のためのフックが提供されているので、そのタイミングで csc.rspにシンボルを書き込めばよさそうです。

くわえて、Unity でのビルドは GUI で行うだけでなく、コマンドライン (CUI) でできるようにしておくと CI 連携もできて便利ですので、ここでは CUI でのビルドについて紹介しようと思います。CUI でビルドする場合は、ビルドクラスを作り、ビルドメソッドをコマンドラインから呼び出すようにします。

今回はビルドクラスを BinaryBuilderとして作成し、ビルドの前処理、後処理のインターフェイスを実装していきます。前処理は IPreprocessBuildWithReport.OnPreprocessBuildとして実装します。これをリスト 7.12 に示します。

OnPreprocessBuildの中では csc.rps更新後適当な C#ファイルも合わせて更新していますが、これは OnPreprocessBuildの中で csc.rpsを更新しただけではシンボルが反映されないために行っています。Unity の公式ドキュメントいわく、

> .rsp ファイルに変更を行うたびに、それを有効にするためには再コンパイルを行う必要があります。これを行うにはスクリプトファイル（ .js または .cs ）を更新または再インポートするだけです。

とのことでしたので、Unity の公式ドキュメントの記述に即した対応になります。

▼リスト 7.12　ビルド前処理

```
/// <summary>
/// バイナリを生成するビルダークラス
/// </summary>
public class BinaryBuilder : IPreprocessBuildWithReport,
                             IPostprocessBuildWithReport
{
    /// <summary>
    /// ビルド前に Unity から呼ばれる処理
    /// </summary>
    /// <param name="report">Report.</param>
    public void OnPreprocessBuild(BuildReport report)
    {
        Debug.Log("OnPreprocessBuild");

        // シンボル情報ロード
        SymbolListManager.LoadSymbolSystemFile();

        // csc.rsp ファイルに書き込み
        SymbolListManager.SaveSymbolRspFile();

        // 適当な C#ファイルを更新
        // ここで扱っている C#ファイルはコメントのみで実行コードなし
        var inFilePath = Application.dataPath + "/Editor/CommentOnly.cs";
        var outFilePath = Application.dataPath + "/Editor/CommentOnly2.cs";
        File.Copy(inFilePath, outFilePath, true);
    }
    // ... 略...
```

ビルドの後処理は IPostprocessBuildWithReport.OnPostprocessBuildです。リスト 7.13 のように、ビルドの後始末として csc.rspファイルを削除しています。

▼リスト 7.13　ビルド後処理

```
    /// <summary>
    /// ビルド後に Unity から呼ばれる処理
    /// </summary>
    /// <param name="report">Report.</param>
    public void OnPostprocessBuild(BuildReport report)
    {
        Debug.Log("OnPostprocessBuild");

        File.Delete(Application.dataPath + "/csc.rsp");
    }
```

CUI からビルドの処理を呼び出すメソッドをリスト 7.14 に示します。BuildIosまたは BuildAndroidメソッドを呼び出します。

▼リスト 7.14　ビルドの開始メソッド

```
    /// <summary>
    /// iOS ビルドの出力パス
    /// </summary>
    /// <value>The output location ios path.</value>
    static string OutputLocationIosPath
    {
        get { return Application.dataPath + "/iOSBuild"; }
    }

    /// <summary>
    /// Android ビルドの出力パス
    /// </summary>
    /// <value>The output location ios path.</value>
    static string OutputLocationAndroidPath
```

```
{
    get { return Application.dataPath + "/AndroidBuild/ESPApp.apk"; }
}

/// <summary>
/// iOS をターゲットにビルドします
/// 主にコマンドラインからの実行を想定しています
/// </summary>
public static void BuildIos()
{
    Debug.Log("Build start, iOS");

    var buildOptions = new BuildOptions();
    var buildTargetGroup = BuildTargetGroup.iOS;
    var buildTarget = BuildTarget.iOS;
    EditorUserBuildSettings.SwitchActiveBuildTarget(
        buildTargetGroup,
        buildTarget
    );

    Build(BuildTarget.iOS, OutputLocationIosPath, buildOptions);
}

/// <summary>
/// Android をターゲットにビルドします
/// 主にコマンドラインからの実行を想定しています
/// </summary>
public static void BuildAndroid()
{
    Debug.Log("Build start, Android");

    var buildOptions = new BuildOptions();
    var buildTargetGroup = BuildTargetGroup.Android;
    var buildTarget = BuildTarget.Android;
    EditorUserBuildSettings.SwitchActiveBuildTarget(
        buildTargetGroup,
        buildTarget
    );

    Build(
        BuildTarget.Android,
        OutputLocationAndroidPath,
        buildOptions
    );
}

/// <summary>
/// 指定したターゲットでビルドを実行します
/// </summary>
/// <param name="target">Target.</param>
/// <param name="locationPath">Location path.</param>
/// <param name="buildOptions">Build options.</param>
static void Build(BuildTarget target,
                  string locationPath,
                  BuildOptions buildOptions)
{
    BuildReport report = BuildPipeline.BuildPlayer(
        GetAllScene(),
        locationPath,
        target,
        buildOptions
    );
    BuildSummary summary = report.summary;

    if (summary.result == BuildResult.Succeeded)
    {
        Debug.Log("Build succeeded: " + summary.totalSize + " bytes");
    }

    if (summary.result == BuildResult.Failed)
    {
        Debug.Log("Build failed");
```

```
            }
        }

        /// <summary>
        /// ビルドを行うシーンファイルを取り出します
        /// </summary>
        /// <returns>scene paths</returns>
        static string[] GetAllScene()
        {
            return EditorBuildSettings.scenes.Where(
                x => x.enabled).Select(x => x.path
            ).ToArray();
        }
```

このようにビルドクラスのメソッドを定義しておき、コマンドラインからは次のように呼び出します。

```
$ /Applications/Unity/Unity.app/Contents/MacOS/Unity
-batchmode
-quit
-logFile /Users/mizusawa-k/Documents/Project/EditorScriptProject/build.log
-executeMethod EditorScript.BinaryBuilder.BuildIos
```

```
$ /Applications/Unity/Unity.app/Contents/MacOS/Unity
-batchmode
-quit
-logFile /Users/mizusawa-k/Documents/Project/EditorScriptProject/build.log
-executeMethod EditorScript.BinaryBuilder.BuildAndroid
```

7.5 おわりに

以上、この章では Unity でプリプロセッサで適用可能なシンボルを定義するエディタ拡張について、その具体的な実装と使い方を紹介してきました。

最終的にはトータルで 700 行以上ものプログラムになり、本章の執筆でエディタ拡張開発デビューした身であることを除いても、C/C++ でのプリプロセッサによる柔軟なシンボル定義の存在の大きさを感じざるをえません。

本章の内容が、読者の皆様の Unity 開発の一助となれば嬉しいです。

ビルド時に適用されているシンボルを確認する方法

ここまで、ビルド時にシンボルを定義する仕組みを紹介してきました。では、ビルド時に実際に定義され適用されているシンボルを確認する方法はあるのでしょうか？
少なくとも、Unity が提供している API の中には無いようです。ただし、Unity 2018 で類似の働きを持つものに次の API があります。

> クラス：
> EditorUserBuildSettings
> パッケージ：
> UnityEditor
> プロパティ：
> public static string[] activeScriptCompilationDefines;
> Unity のドキュメント上での説明：
> コンパイラの DEFINE 命令

ドキュメントによると、API が呼び出された時点で有効になっているシンボルを文字列の配列として取得できるようなのですが、筆者が確認したところ（Unity 2018.3.6f)、コマンドラインでビルドコマンドを呼び出した際、`csc.rsp`に記載したシンボルが取得したリストに含まれていませんでした。
どうも、この API で取得できるのはプラットフォームごとの PlayerSettings にある Scripting Define Symbols の文字列配列のようです。このため、同 API を利用する場合は留意しておいた方が良さそうです。

第 8 章

デジカメで撮影した写真の正確なタイムスタンプを推測する

本章は、Android を搭載するソニーのデジカメ上で動作するアプリを試作し、カメラ性能の限界を突破する可能性を研究・検討するものです。具体的には、デジカメの内蔵時計がズレやすいという厄介な問題に対して、補助アプリの開発によって内蔵時計がズレても困らない仕組みを確立することを目指します。

本章では、Java や Kotlin のコードをなんとなく雰囲気で読める（正確に読解できる必要はありません）ソニー製カメラ利用者を読者として想定しています。

8.1　デジカメと Android

Android とデジカメの微妙な関係

Android は世界で最も広く普及したモバイル用 OS です。2003 年創業の Android 社を 2005 年に Google 社が買収し、携帯電話（スマフォ）用 OS として発展を遂げたことは周知の事実ですが、初期の Android がデジカメをメインターゲットとして開発されていたことはあまり知られていません。

さて、2010 年頃までにスマフォ市場が急速に成長したことは、特にコンパクトデジカメ（コンデジ）業界にとって脅威でもありました。カメラとしてのレンズ・センサー・ズーム性能によって一定以上の品質の写真を撮れるとはいえ、コンデジのメイン利用者であるライトユーザー層の「手軽に撮って、簡単に加工、素早く共有」という需要を満足するものではなかったためです。

デジカメ業界の競争が年々激化する中で、Android 経済圏との融合を図る製品開発もおこなわれました。Samsung GALAXY Camera（2012 年）、同 2（2014 年）、Nikon COOLPIX S800c（2012 年）といった製品が Android 搭載を前面に打ち出して販売されました。これらはいずれも通常の Android スマフォと同じように Google Play ストアからアプリをインストールする構造で、特に SNS への写真投稿や画像加工という面での付

加価値を狙った製品であったと解釈できます。

残念ながらいずれも大きな売上を記録することなく、歴史の中に埋もれていきました。

近年は、スマフォからカメラを遠隔操作して好きな設定・タイミングで写真を撮り、スマフォへデータを転送して加工するという分業スタイルの連携が主流です[*1][*2]。

ソニーの Android 搭載カメラ

あまり宣伝されていませんが、実はソニーのカメラには Android を搭載する[*3]ものが多く存在します。これらのうち 2017 年頃までに発売された機種では、PlayMemories Camera Apps[*4]と呼ばれるアプリを専用ストアから入手し、カメラの機能を拡張できます。

ストア自体はいまひとつ盛り上がりに欠けており、掲載アプリ数は 33、最後にアプリが追加されたのは 2017 年 3 月と、少々寂しい状態です。

ソニーのカメラ用非公式アプリ開発環境

カメラを使っていて不便を感じたらどうするか。新しいカメラを買うのが正攻法ですが、先立つ物がなければ我慢することになります。

しかし我々は開発者なので、目の前の不便を解消する仕組みの自作を可能な範囲で模索したいところです。PlayMemories Camera Apps のアプリストアが下火だからといって、出来ることがないわけではありません。カメラの動作を調査して構築された OpenMemories[*5]と呼ばれる非公式の SDK が存在します。

普段利用しているカメラがどこまで普通の Android なのか、そして自前アプリから何ができて何ができないのか、気になりますね。試しにやってみましょう。

8.2 要望：デジカメの時計がズレるのをなんとかしたい

本章の著者は、デジカメの内蔵時計がズレて困るケースがそれなりにあるのでなんとかしたい、と日々思っています。これが今回取り上げる、カメラに搭載された Android サブシステムを使って解決したい課題です。まずは、曖昧な要望をブレイクダウンして明確化するところから手を付けてみましょう。

[*1] https://developer.sony.com/ja/develop/cameras/ Camera Remote API beta SDK - Sony Developer World

[*2] https://www.itmedia.co.jp/news/articles/1903/06/news109.html キヤノン、デジカメを遠隔操作できる SDK と API 提供（2019/3/6）

[*3] 正確には、カメラの機械制御や電子制御を担うメインシステムに加えて Android プラットフォームをサブシステムとして相乗りさせる構造です

[*4] https://www.sony.jp/support/software/playmemories-camera-apps/

[*5] https://github.com/ma1co/OpenMemories-Framework

デジカメの時計がズレてしんどい問題

デジカメの時計はよくズレます。著者の手元にあるデジカメの時計は、気付くと 5 分から 10 分はズレています。ズレの主な原因としては、内蔵時計に使われる省電力水晶発振子モジュール（RTC）の精度が低いことや、カメラ筐体が高温環境に曝されることが考えられます。

時計がズレると困る場面

正式な記録用写真を撮るわけでもないのだからデジカメの時計は気にしないで良い、と思っていた時期もありましたが、割と実用上厄介なシーンがあります。

ひとつめは、イベント写真を複数人で撮って共有アルバムへアップロードする時で、デジカメの時計がばらばらだと時系列がちぐはぐになる、というものです。カンファレンスや結婚式の共有アルバムで一人だけ写真のタイムスタンプがずれていて恥ずかしい思いをした方もいるのではないでしょうか（著者は経験あります）。

ふたつめは、一人でも複数のデジカメを併用あるいはデジカメとスマフォを使い分けて撮影する時で、これも結果としてアルバムの時系列が崩壊する、というものです。とりわけ旅行先で簡単な撮影はスマフォ、しっかりした撮影はデジカメを利用するケースが多いものですが、デジカメの時計がズレていると後でアルバムを整理する際に脳が混乱します。

これらを防ぐライフハックとして、一連の撮影に加えてスマフォの時計をデジカメで撮影しておき、その写真を基準として同じカメラで撮影したすべての写真にタイムスタンプ差分を後から適用する、というものがあります。

理屈は分かるのですが、著者は残念ながらタイムスタンプ基準写真を撮影しそびれることが多いです。さらに、デジカメとスマフォで並列して撮影した写真の中からデジカメで撮影した分のみを選び出して時刻を補正する、という手順自体が結構手間です。デジカメで撮影した写真をなるべくスムーズにスマフォへ転送し、意識せずクラウドサービスへバックアップしてから整理したいという欲求と相容れない手続きでもあります。

身近で正確な時計：スマフォ

スマフォの時計には信頼感があります。

なぜスマフォの時計はズレないのでしょうか。ズレない部品があるなら、それをデジカメにも使えばいいのに、と思いませんか。

スマフォは一般的にデジカメよりもかなり大容量のバッテリーを搭載するため、主電源を切っている間でも RTC への電力供給に余裕がある点は指摘できます。

それでも実際のところ、スマフォの内蔵時計は頻繁にズレています。頻繁にズレますが、頻繁に補正しています。携帯電話網や Wi-Fi へ接続されているスマフォの場合、携帯電話基地局やネットワーク時刻サーバなど豊富な高精度ソースから時刻情報を得られるた

め、利用者が時計のズレを感じることはほとんどありません。

デジカメの自動時計あわせ機能

地球上で通信に依存せず幅広く利用できる高精度な時刻情報のソースとして、GPS 電波が挙げられます[*6]。一時期、デジカメに GPS 受信モジュールを搭載して写真に正確な位置情報・タイムスタンプを記録するという製品開発の流れがありました。しかしカメラの起動から GPS 測位完了までにかかる時間が長いという難点があり、部品コストと消費電力の割に撮影のリアルタイム性要求とマッチしなかったためか、近年のカメラにはあまり搭載されていません。

代わりに、正確なスマフォの時計と位置情報をカメラ側へと取り込む傾向にあります。

2013 年頃発売の機種にも、スマフォ用の連携アプリに接続したタイミングでカメラ内の時計を同期するものがあります（DSC-QX シリーズ）[*7]。残念ながら、この方法の問題点は「新たに撮影する写真のタイムスタンプは補正されるが、すでに撮影した写真のタイムスタンプはズレたまま」というところです。1 ヶ月間の誤差が 2-3 秒程度におさまる RTC であれば十分に実用範囲ですが、気付いたら数分ズレているような RTC に対しては不十分です。かといって、カメラの電源投入後、撮影前にスマフォと接続して最新の時刻を取得する、という使い方は本末転倒で、やはり写真は撮りたいと思った瞬間に撮りたいものです[*8]。

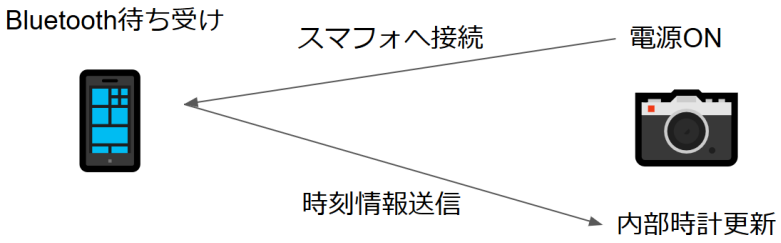
このような問題に対処するためか、2018 年以降に発売された高級機ではカメラ起動時にスマフォへ Bluetooth で接続して現在時刻と位置情報を受け取る機能を持つものもあります（図 8.1）[*9]。この方法の強みは、写真を撮影するためにカメラの電源を入れた時点で、利用者が意識する必要もなく時計が同期することです。是非この仕組みを搭載したカメラへ乗り換えたいところですが、いかんせん買い換えるには値が張ります。

[*6] 電波時計や地デジの放送波といったソースも思いつきますが、国ごとの事情差が大きいためユニバーサルとは言えません

[*7] `https://www.sony.jp/ServiceArea/impdf/pdf/44773110M.w-JP/jp/contents/TP0000219635.html` DSC-QX10/QX100 日時設定と時計合わせについて

[*8] 例示した DSC-QX シリーズは「レンズスタイルカメラ」という本体ディスプレイすら持たない特殊なモデルで、スマフォとの密な連携を前提とするカメラであったからこその機能といえます

[*9] `https://www.sony.jp/support/software/playmemories-mobile/operation/location/android/` 画像にスマートフォンの位置情報を記録する（Android）



▲図 8.1　カメラが Bluetooth 経由でスマフォから時刻情報を取得するパターン

要件まとめ

まとめると、著者が満たしたい要件は次のとおりです。

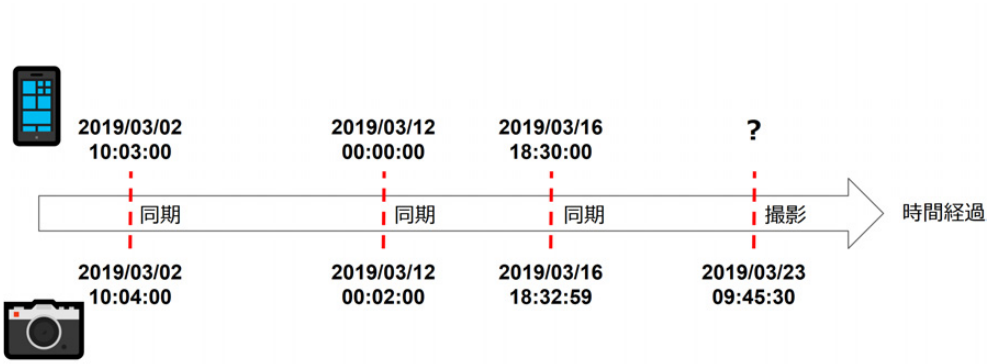
- Bluetooth 時刻同期機能を持たないデジカメで撮影した写真のタイムスタンプを、なるべく正確な値（2 秒精度）へと補正する
- 時刻補正は写真のクラウドへの同期（バックアップ）前に完了する
- デジカメの時計を手動で調節しない
- 写真を撮り始める前の明示的なスマフォ連携による時計合わせをおこなわない

8.3　システム概要検討

前節でまとめた要件を満たす仕組みを検討します。デジカメの内蔵時計はズレるに任せて放置しつつ、正確な時刻情報を持つスマフォとうまく連携することで撮影済み写真の正確なタイムスタンプを推測できるか否か、がポイントです。

GPS 非搭載デジカメの秒精度タイムスタンプを推測する手法

基本的なアイディアは、あらかじめスマフォとカメラそれぞれの主観的なタイムスタンプ（UNIX 時間の値）をセットで複数回記録しておき、カメラで撮影した写真の持つ撮影日時メタデータ（Exif）を入力値としてスマフォ側のタイムスタンプを補完して求める、というシンプルなものです（図 8.2）。



▲図 8.2　撮影時タイムスタンプの推測方法

まず、スマフォアプリとカメラ内アプリを連携させて、ある瞬間の両者の主観的なタイムスタンプの組をなるべく高い精度で取得して記録します。写真の撮影後にもタイムスタンプの組を記録したいので、日常的な写真撮影のオペレーション内で自然に利用する機能の一部にタイムスタンプ記録を盛り込むことが望ましいです。また、カメラを複数台保有して使い分けることも考えられるため、タイムスタンプの組を記録する際にはカメラ機種とシリアル番号のペアを取得してスマフォ側で仕分けする必要があります。

続いて、ある写真の撮影日時メタデータ（Exif）を入力値として、蓄積したタイムスタンプの組を利用して撮影時の正確なタイムスタンプを推測します。具体的には、対象タイムスタンプを挟む直前・直後の 2 サンプル、直後のものが存在しなければ最近傍の 2 サンプルをもとにした線形補間（一次補完）を想定します。RTC の定常的なズレは水晶発振子の性質によるもので、線形補間で十分に高い精度を出せると期待できるためです。また、高温環境下に曝すことで発生すると考えられるズレは頻度が低く、かつ全体のズレに占める割合も小さいものと想定します。

タイムゾーンの扱いに関する懸念

前節で述べたシンプルなタイムスタンプ推測手法において、カメラ側のタイムスタンプに関しては一筋縄ではいかない事情があります。

少なくとも 2015 年ごろまでに発売されたカメラにおける写真メタデータはローカル時刻主義です。つまり、保存される JPEG ファイルの中にはローカル時刻しか記録されず、当該時刻が協定世界時（UTC）でいつにあたるのか（=UNIX 時間）を知ることはできません。撮影時点のタイムゾーン情報が残されていれば UNIX 時間の計算は容易ですが、これも残りません。このため、カメラ内の設定でタイムゾーンを変更すると、手順後半で補完の入力値として利用する「撮影日時メタデータ」自体が大きくズレることになりま

す[*10]。

この状態でタイムスタンプの推測処理をおこなうと、線形補間の際に本来の記録時刻と異なる基準点を選ぶ可能性があるため、望ましくありません。

暫定対策として、カメラ側のタイムゾーンは固定することを前提に、スマフォ側のタイムゾーン変化情報を適宜取得、別途記録して最終的なタイムスタンプ推測時に適用する必要がありそうです[*11]。

PoC（コンセプト実証）プログラムの構築手順

タイムスタンプを正確に推測する手法の目処がついたので、次節以降で実際に本要件を満たすための PoC プログラムを構築します。情報の少ない SDK なので、次のような手順で基盤部分から順に積み上げることにします。

- スマフォとカメラを Wi-Fi で接続する
- スマフォアプリとカメラとの間の通信基盤を整備する
- スマフォアプリからのリクエストによってカメラの型番/シリアル番号と時刻情報を取得する
  - UNIX 時間を算出するのに十分な情報を得られるか否かを確認する

実用上はカメラからスマフォへ転送した写真（JPEG ファイル）のメタデータを編集してクラウドサービスへアップロードする機能も必要ですが、これは正確な補正情報さえ揃っていれば通常の Android アプリの文脈で完結して開発できるため、本章では扱いません。

8.4 OpenMemories SDK を利用したアプリ開発

OpenMemories フレームワークを利用したカメラ用アプリ開発は公式にサポートされた方法ではありませんが、本章では研究・検討を目的として簡単に解説します。

ソニーの Android 搭載カメラの概略

まず気になるのは、カメラに搭載されているのがどの程度普通の Android で、どのような API セットを利用できるか、という点です。

搭載バージョンは Android 2.3.7 または Android 4 系です。

標準的な Android SDK の API 群（標準 API）に加えて、`com.sony`名前空間に配置さ

[*10] SD カードに記録されるファイルの更新日時情報から UNIX 時間を取り出せる可能性はありますが、この情報は連写時に軽く見積もって 10 秒程度はズレることが想定されるため、一次情報として扱うには適しません

[*11] SD カード上のタイムスタンプを用いてカメラ側のタイムゾーン情報を得られる可能性については別途検討とします

れたクラス群（独自 API）を利用できます。標準 API の中には動作がおかしいものもあり、それらは独自 API を使った代替実装を検討することになります。実際に使える手札を探し当てるためにクラスのスタブ定義[*12]を読みつつ試行錯誤する必要があります。独自 API にはオートフォーカスの開始・停止やシャッターを切るといった撮影系の API に加えて、各種操作キーに関するイベント処理、レンズの着脱イベント処理なども含まれます。

カメラ機種と対応 API バージョンのリストは `https://github.com/ma1co/OpenMemories-Framework/blob/master/docs/Cameras.md` に掲載されています。2016 年 10 月製品発表の機種を最後にアプリストアからの追加インストール機能が削除されたため、その後に発表された機種は動作対象外です。なお、この表で「API version」と書かれた項目が、各機種で実装している独自 API のバージョンを指します。標準 API の API Level とは関係ありません。

アプリケーションのバイナリ形式としては、一般的な Android アプリと同じく apk ファイルを利用できます。

開発機の確保

なにぶん非公式 SDK を利用した開発ですので、最悪の場合にはカメラが起動しなくなることを覚悟すべきでしょう。もちろん、メーカーによる保証も失われると考えるべきです。このため、普段使いのカメラとは別に開発用の機材（カメラ）を入手します。Android 搭載機は基本的にハイエンド機種であるため値崩れしにくいのですが、DSC-HX60V という 2014 年に発売されたコンデジは API version 2.4 というある程度新しい独自 API セットに対応しつつ中古品を 2 万円前後で入手できます。開発機として使い倒す Android 端末としては手頃といえます。

学習に適したデモプロジェクト

PMCADemo（`https://github.com/ma1co/PMCADemo`）というプロジェクトを出発点にするのが手っ取り早いです。このデモプロジェクト内にはカメラの制御からメディアアクセス方法まで、かなり実用性の高いサンプルコードが揃っています。

Android Studio の最新安定版（2019/3/13 時点）である 3.3.2 で本プロジェクトを開くと、非推奨の記述や対象ビルドツールが古いことなど数点の警告やエラーが出力されます。ビルドを通すために粛々とコードを書き換える必要がありますが、本筋から外れるため省略します。

重要箇所として、独自 API のスタブクラスを `compileOnly`でモジュールのコンパイル時に見えるようにする必要があります（リスト 8.1）。

[*12] `https://github.com/ma1co/OpenMemories-Framework/tree/master/stubs`

▼リスト 8.1　build.gradle ファイルの compileOnly 記述

```
dependencies {
...
  compileOnly 'com.github.ma1co.OpenMemories-Framework:stubs:-SNAPSHOT'
...
}
```

開発機へのインストールと実行

apk ファイルの生成に成功したら、続いて開発機へインストールします。インストールには、有志によって開発された pmca-gui という GUI アプリを利用します。PC とカメラを USB ケーブルで接続し、pmca-gui アプリの"Install app"タブを操作して apk ファイルをインストールします。

インストールしたアプリはカメラ側メニューの「アプリ一覧」に表示されます。カメラを操作してアプリを実行し、PMCADemo アプリの初期 Activity が開いて実行可能なデモの一覧が表示されれば成功です。

あとは各種デモを実行したり、該当コードを読んだりすることで SDK への理解が深まります。

8.5　スマフォとカメラを接続し、カメラの時刻情報を取得する

SDK の扱い方をある程度把握したところで、いよいよ実際のアプリの骨組みを成す PoC 版の構築に取り掛かります。といっても難しい部分はすでに先人の手で切り拓かれているので、それに沿ってサクサクと各機能を組み立てるイメージです。

Wi-Fi 経由のスマフォ接続

ソニーのカメラの写真転送機能を利用した経験のある方はご存知だと思いますが、これらは接続先の Wi-Fi アクセスポイントをカメラに設定して通信するモードと、カメラがスマフォからの接続を受け付けるモードを使い分けられます。

通常、無線 LAN といえば無線 LAN ルータに代表されるアクセスポイントへ各端末が接続し、アクセスポイントを介して通信をおこなうものですが、外出先での利用に不向きです。これに対して、カメラとスマフォの間のデータ転送によく利用されるのは Wi-Fi Direct と呼ばれる技術で、2 台以上の端末のうち 1 台がアクセスポイントの代わりになるモードで待ち受けて、他の端末がそこへ接続することでデータ通信を可能にするものです。

OpenMemories では簡単に Wi-Fi Direct での待受が可能で、PMCADemo アプリの WifiDirectActivity に完全動作するサンプルコードがあります。つまり、Wi-Fi 経由でス

マフォから接続を受ける部分に関しては apk ファイルを生成して動作確認するだけでおしまいです。

カメラが TCP サーバとして機能することの確認

Wi-Fi での接続を確立できたら、続いて簡単なデータ送受信を試みます。

なんと、この手順も PMCADemo をビルドした時点でほとんど完了しています。PMCADemo の WifiDirectActivity 内では、Activity を開始した時点で HttpServer というクラスを利用して簡易的な HTTP サーバを開始します。

HttpServer の実体は NanoHTTPD という Web サーバライブラリで、簡単なやり取りが可能です。スマフォまたは PC からカメラへ Wi-Fi 接続し、`http://192.168.122.1:8080/` へアクセスすると、「Hello World!」と出力されます[*13]。カメラが TCP サーバとして機能することも、デモアプリの動作確認のみで検証できました。

メッセージング基盤の選択と疎通確認

前項で、簡易 HTTP サーバを介してスマフォとカメラがメッセージをやり取りできることを確認しました。このまま HTTP をベースとしてアドホックにコードを書いて Web サービスを実装することも可能ですが、扱うメッセージの性質にあわせて基盤を検討すべきタイミングでもあります。

メッセージング基盤の検討

ネットワーク経由のデータ交換という文脈で JSON が栄華を極める現代ですが、スマフォとカメラの間でやり取りするメッセージには写真データに代表されるバイナリデータが多用されることを考えると、REST+JSON という選択肢は避けるのが無難です。もちろん、コマンドや軽いデータのやり取りを REST+JSON でおこないつつ、重いデータをバイナリ転送するパスを切り分けるのも定石のひとつですが、スマフォとカメラの間での通信回数が増加する分だけ不利です。

今風にかっこよく protobuf/gRPC で組む、といきたい気もしますが、ここでサーバ役も Android アプリであることを思い出す必要があります。gRPC は HTTP/2 の上に構築されたメッセージング基盤ですが、Android 用の HTTP/2「サーバ」ライブラリの定番は著者の知る限り存在しません。

このようなケースには、クライアント・サーバ共に各言語での実装が充実していてバイナリデータも扱えるポータブルなメッセージング基盤である Thrift[*14]が適しています。

[*13] 少なくともソニーのカメラにおいては Wi-Fi Direct でカメラ側が持つ IP アドレスは 192.168.122.1 だとハードコードしてよさそうです

[*14] `https://thrift.apache.org/`

Thrift を使った基本的なメッセージング

Thrift の書式やルールはともかく、Thrift の扱い方自体は簡単です。.thrift というメッセージ定義ファイルを書き、コマンドでコードを生成し、実装部分を書き、使うというシンプルな 4 ステップです。

さっそく、定義ファイルの作成とサーバ実装、そして Android クライアントからカメラ側のサーバを呼び出すまでをやってみます。

ここで構築するのは、整数値 2 つを渡すとそれらの積を返すという単純なサービスです（リスト 8.2）。

▼リスト 8.2　hello.thrift ファイル

```
service MultiplicationService
{
  i32 multiply(1:i32 n1, 2:i32 n2)
}
```

この内容を hello.thrift ファイルへと保存し、thrift-compiler[*15]でコードを生成します。

```
$ thrift --gen java hello.thrift
```

続いてサーバとクライアントをそれぞれ書き、動作を確認します。下準備としてカメラアプリとスマフォアプリの両方のプロジェクトで Java 用の Thrift ライブラリを取り込み（リスト 8.3）、また生成済みの Java コードを Android Studio のプロジェクト内へと適宜取り込みます。

▼リスト 8.3　build.gradle の Thrift ライブラリ依存関係記述

```
dependencies {
  ...
  implementation 'org.apache.thrift:libthrift:0.9.1'
  ...
}
```

今回は先にカメラ側（サーバ）での計算コードとサーバ起動コードを書きます。

サーバ側の計算コードはリスト 8.4 のとおりです。

▼リスト 8.4　単純な積算をおこなうサービスの実装

```
class MultiplicationHandler() : MultiplicationService.Iface {
  override fun multiply(n1: Int, n2: Int): Int = n1 * n2
}
```

[*15] Homebrew では thrift パッケージ、Debian/Ubuntu では thrift-compiler パッケージでインストールできます

サーバの開始部分は、一旦 PMCADemo の WifiDirectActivity の groupCreated メソッドの続きに書きます（リスト 8.5)。

▼リスト 8.5　Thrift サーバの起動

```
protected fun groupCreated(configuration: DirectConfiguration) {
  log("Group created!!")
  log("SSID: " + configuration.ssid)
  log("Key: " + configuration.preSharedKey)
  log("Server URL: http://" + MY_IP_ADDRESS + ":" + HttpServer.PORT + "/")
  // ここから Thrift サーバ初期化部分
  handler = MultiplicationHandler()
  processor = MultiplicationService.Processor(handler)
  val runnable = Runnable {
    try {
      val serverTransport = TServerSocket(9090)
      val server = TSimpleServer(
          TServer.Args(serverTransport).processor(processor))
      log("Starting the simple server...")
      server.serve()
    } catch (e: Exception) {
      log(e.stackTrace[0].toString())
    }
  }
  Thread(runnable).start()
  // ここまで Thrift サーバ初期化部分
}
```

一方、スマフォアプリ側では画面上のボタンをタップした際にサーバ（カメラアプリ）へ接続して計算処理を実行し、結果を画面上に配置した TextView へ表示するようにしてみます（リスト 8.6)。

▼リスト 8.6　スマフォアプリから Thrift サービスへ接続して計算処理を実行

```
btn.setOnClickListener {
  thread {
    val transport = TSocket("192.168.122.1", 9090)
    transport.open()
    val protocol = TBinaryProtocol(transport)
    val client = MultiplicationService.Client(protocol)
    val answer = client.multiply(6, 7)
    runOnUiThread {
      textView.text = answer.toString()
    }
  }
}
```

カメラ側でアプリを起動し、スマフォから同アクセスポイントへ接続してボタンを押すと、「42」という計算結果が表示されます。これで、Thrift を用いたメッセージング基盤を確立できました。

スマフォからの要求によりカメラ内の時刻を返す

時刻情報は、PMCADemo に含まれる Time サンプルを参考にできます。`TimeUtil#getCurrentCalendar`メソッドで得られる `PlainCalendar`のインスタンスに時刻情報が

含まれることがわかります。

残念ながらこのデータは秒精度です。ミリ秒精度ぐらいのデータを直接取得できれば嬉しいところですが、贅沢はいえません。PlainCalendarがローカル時間のタイムスタンプ（秒単位）を保持し、PlainTimeZoneがタイムゾーン情報を持ちます。タイムゾーンはGMT(UTC) に対する差分と夏時間差分がそれぞれ分単位で取得できます。

Thrift でタイムスタンプとタイムゾーンの値をほぼそのまま取得し、受け取ったスマフォアプリ側で加工するのが正攻法でしょう。

リスト 8.7 のような Thrift サービスを定義します。

▼リスト 8.7　om.thrift タイムスタンプ取得用サービス定義

```
struct TimestampSource
{
  1:i32 year,
  2:i32 month,
  3:i32 day,
  4:i32 hour,
  5:i32 minute,
  6:i32 second,
  7:i32 gmtDiff,
  8:i32 summerTimeDiff
}

service OMService
{
  TimestampSource getCurrentTimestamp()
}
```

Thrift で単純に構造体（struct）を定義し、その取得用サービスを定義します。

```
$ thrift --gen java om.thrift
```

で Java のコードを生成し、Android Studio 内へと適宜取り込みます。

TimestampSource を返すサービス（カメラ側）はリスト 8.8 のように実装します。

▼リスト 8.8　カメラの現タイムスタンプ返却

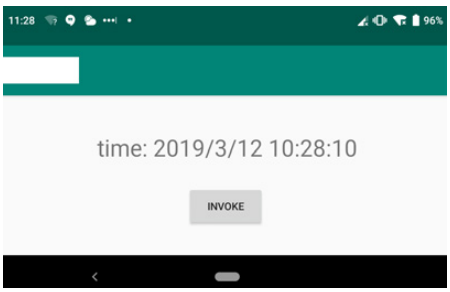
```
class OMHandler : OMService.Iface {
  override fun getCurrentTimestamp(): TimestampSource {
    val cal = TimeUtil.getCurrentCalendar()
    val tz = TimeUtil.getCurrentTimeZone()
    return TimestampSource(
        cal.year, cal.month, cal.day, cal.hour, cal.minute, cal.second,
        tz.gmtDiff, tz.summerTimeDiff
    )
  }
}
```

呼び出しコード（スマフォアプリ側）もシンプルです（リスト 8.9)。

▼リスト 8.9　カメラのタイムスタンプを Wi-Fi 経由で取得

```
btn.setOnClickListener {
  thread {
    val transport = TSocket("192.168.122.1", 9090)
    transport.open()
    val protocol = TBinaryProtocol(transport)
    val client = OMService.Client(protocol)
    val cal = client.currentTimestamp
    val timestr = "time: %d/%d/%d %d:%d:%d".format(
        cal.year, cal.month, cal.day, cal.hour, cal.minute, cal.second)
    runOnUiThread {
      textView.text = timestr
    }
  }
}
```

無事にタイムスタンプを取得できました（図 8.3）。



▲図 8.3　カメラのタイムスタンプ取得結果

カメラアプリの特殊事情を勘案しつつ、タイムスタンプの補正に必要なデータを得られたことになります。この先の、取得したデータをスマフォアプリ側で加工・保存・検索する部分は、完全に普通の Android アプリなので今回は省略します。

OpenMemories 基盤に基づくアプリ開発の特殊な点

OpenMemories Framework を利用する範囲では、開発中のアプリの APK ファイルを通常の Android 端末やエミュレータへインストールして普通に実行できます。これは同フレームワークが独自 API に対する非カメラ用の代替実装を持つためです。フレームワークでラッピングされていない、独自 API を直接利用するコードを含む場合には事前検証が困難です。
アプリ開発上の留意点として、USB ケーブルをつなぎっぱなしにして adb でログを確認しながら開発、というわけにいかないことが挙げられます。カメラ設定の編集アプリをインストールして内部の ADB 機能有効化・Wi-Fi の常時有効化設定を施すことで、Wi-Fi 経由の adb 接続が可能ですが、本章で扱ったような Wi-Fi Direct の接続/切断を多用するアプリのデバッグには不向きです。原始的な方法ですが、SD カードや FlashAir へログを書き出して適宜 PC へと取り込むのが無難でしょう。

8.6 まとめ

デジカメの内蔵時計がズレやすいという問題に対して、カメラとスマフォアプリを連携させてタイムスタンプ記録を作成することで正確な時刻の推測ができることを展望しました。そして、ソニー製カメラが搭載する Android サブシステムを利用したアプリ開発の枠組みにおいて、OpenMemories SDK を利用して本問題を解決するための最低限のデータ取得系を構築できることを示しました。

8.7 本章で取り上げなかった要検討事項

本章では要求精度を 2 秒程度としましたが、可能であれば秒未満の精度を追求したいところです。最後に精度向上のためのヒントを数点記述します。

PlainCalendar の精度改善

本章で紹介した PlainCalendar のインスタンスは秒単位のデータしか持ちません。このままでは 1 秒未満の精度を望むこともできませんが、カメラ側のアプリに手を加えることで、この情報を補強できます。Android 標準の API を利用し、カメラ起動からの経過ミリ秒（起動後ミリ秒）を取得できるため、比較的簡単に実装可能です。

たとえば 50ms 間隔で 20 回程度 `TimeUtil#getCurrentCalendar`を呼び出すと、どこかで秒の変化するタイミングがあります。このタイミングで起動後ミリ秒（baseMillisec）

を記録し、スマフォアプリから時刻情報を要求されたタイミングで取得した起動後ミリ秒（lastMillisec）から引いた差分を 1000 で除した余り[*16]を返すことで、おおむね 100ms 精度のタイムスタンプを返せることになります[*17]。

ネットワーク遅延

PC からカメラへ Wi-Fi 接続して ping を打ってみると、PC とカメラが至近距離にあっても応答時間のばらつきが大きいことがわかります。手元の DSC-HX60V では、2ms から 100ms 超までのばらつきがありました。

タイムスタンプの精度を追求するうえで、このばらつきは厄介な問題です。100ms のうち往路と復路で均等に時間がかかっているのか否かも分かりません。

この点については、複数回の試行によりばらつきの均質化を図ること、一定の閾値を超えて戻ってきた応答を棄却すること、などが有効です。本格的にやると NTP[*18]を再実装することになりそうなので、ほどほどにするのが良いでしょう。

うるう秒（leap second）

UNIX 時間はうるう秒を適用済みの時計における UTC1970 年 1 月 1 日 0 時 0 分 0 秒からの経過秒数を示すものであり、実際の経過秒数とは一致しません。うるう秒が挿入または削除されるタイミングでは、20 時間程度かけて時刻サーバの配信データをゆるやかに引き伸ばしたり縮めたりすることが一般的です。この間にスマフォの時計は（同期がおこなわれるたび）少しずつずらされていって辻褄が合うことになりますが、ともかくその日を境に 1 秒間の認識差が生じます。結果的にスマフォとカメラのタイムスタンプ組を取得するタイミングによっては誤差が 1 秒拡大するため、なんらかのポリシーを定めて対応する必要があります。

[*16] 文章にすると分かりづらいですが、単純な `val millisec = (lastMillisec - baseMillisec) % 1000`という剰余計算です

[*17] これが 50ms 精度にならないのはサンプリング定理によります。機種によってどの程度の精度まで耐えられるかを自動調節するのも一手でしょう

[*18] `https://ja.wikipedia.org/wiki/Network_Time_Protocol`

著者

第 1 章 Shunsuke Ito / @fgshun
のんびり、まったり

第 2 章 Shinya Naganuma / @Pctg_x8
人生二期では猫として生きたい

第 3 章 Daisuke Makiuchi / @makki_d
眼鏡っ娘が好きです

第 4 章 Sho HOSODA / @gam0022
日本最強のグラフィックエンジニア（自称）

第 5 章 Suguru OHO / @ohomagic
長時間のフライトを快適に過ごせるライフハックを知りたいです

第 6 章 Yoshio HANAWA / @hnw
機械学習とライトに付き合いたい

第 7 章 Kinuko Mizusawa
何とかこんとか... 表紙が尊いです...

第 8 章 @muo_jp
次回は令和元年の技術書典でお会いしましょう。

あつち
奥の二人とロボットを担当しました。楽しかったです！ 平成最後のメガネっ娘。

たかぎ
とても良いメンバーと挑戦的な制作に参加できて楽しかったです！

木

熱量が高いメンバーと制作できてとても刺激的でした！ この学院に入学するために願書かかなきゃ…

二本足おたまじゃくし

可愛い女の子が集まって仲良さそうなの良いなぁと思いました。制作に関われて楽しかったです！

福田

背景楽しかったです。

みなみせんぱい

柵のみですいません。完成度が高くて大変尊いものが…（拝）

ヤマウラ

タイトルロゴなどの表紙デザインや衣装デザイン、一部キャラデザなどデザイン周りを担当しましたメンバーの力で春らしいはつらつとした表紙になったかと！

電子版ダウンロード

http://klabgames.tech.blog.jp.klab.com/techbook/KLabTechBook_Vol4.pdf

KLab Tech Book Vol. 4

2019 年 4 月 14 日　技術書典 6 版（1.0）
著　者　KLab 技術書サークル
編　集　牧内 大輔、於保 俊
発行所　KLab 技術書サークル
印刷所　日光企画

(C) 2019 KLab 技術書サークル