KLab TECH BOOK

Vol. 1

KLab Tech Book Vol. 1

2017-10-22 版　　**KLab** 株式会社 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、個人的に技術書典 1・2 にて出版していた岡本が発起人となって KLab 株式会社の有志にて作成されたものです。KLab はスマートフォン向けのゲーム制作を主に行なっている会社ですが、業務に関連するノウハウのみならず、各々のエンジニアが個人的に研究したことなど、社内には書籍化できるようなネタがつつけば出てくるだろうなと感じていたので募集したところ、実際に CG・Emscripten・FPGA・スマホアプリビルドなど多様なトピックが集まりました。

内容は各章ごとに独立していますので、どの順番からお読みくださっても問題ありません。

楽しんでいただければさいわいです。

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。

ライセンス表記

表紙にて次の素材を使用しています。

Chalk Brush

Creative Commons Attributioin 4.0

Mattia Lo Bianco aka MattiaMc

L0067855 Oil painting frame

Creative Commons Attributioin 4.0

Wellcome Library, London

目次

はじめに **2**
お問い合わせ先 2
免責事項 2
ライセンス表記 2

第 1 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 **5**
1.1　3DCG の描画方法の基礎 5
1.2　パストレーシングと大域照明 7
1.3　レンダラーの紹介 10
1.4　表紙のシーンの解説 11
1.5　高速化の取り組み 11
1.6　Rust を選んだ理由 14
1.7　初めて Rust で実装した感想・苦労 14
1.8　まとめ 15

第 2 章　Sprache を CPS 変換 **16**
2.1　Sprache の紹介 16
2.2　CPS 変換とは 18
2.3　CPS 変換する上での問題 19
2.4　パフォーマンス比較 24
2.5　まとめ 25

第 3 章　Emscripten で動画再生する **26**
3.1　ブラウザでの動画再生 27
3.2　libvpx のビルド 28
3.3　動画ファイルの読み込みライブラリの作成 29
3.4　動画レンダラの作成 33
3.5　動画レンダラのビルド 36
3.6　まとめ 38

第 4 章　テキストマクロプロセッサ「M4」のチューリング完全性について　39
4.1　チューリング完全性の証明方法 39
4.2　実装方法 40
4.3　チューリング完全性の証明 43
4.4　終わりに 44

第 5 章　FPGA 初心者が試行錯誤しながら疑似乱数生成回路を作る話　45
5.1　本稿での環境・範囲 46
5.2　何を作るのか？ 46
5.3　まずは入出力 46
5.4　BBS 疑似乱数生成器を作る 48
5.5　人に優しいクロックの生成 49
5.6　目標達成の完成形 50
5.7　まとめ 51

第 6 章　家庭内ネットストーカーシステムを作った話　52
6.1　システム概要 52
6.2　ブロードバンドルータにカスタムファームウェアをインストールする . . 54
6.3　ブロードバンドルータの機種選定 54
6.4　MIPS ルータ上で mackerel-agent を動かす 55
6.5　独自プラグイン mackerel-plugin-iw の導入 56
6.6　mackerel-plugin-iw が表示する情報 57
6.7　本システムのキモさ 60
6.8　まとめ 61

第 7 章　とある KLab のスマホアプリのビルド事情　62
7.1　スマホアプリ開発、運営の現状 62
7.2　一般的なアプリのビルドでよくある問題 63
7.3　KLab のプロジェクト共通環境 64
7.4　ビルドに関わってきたプロジェクトの紹介 66
7.5　各プロジェクトのビルドフロー振り返り 66
7.6　おわりに 72
7.7　参考資料 73

著者　74

第 1 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

本誌の表紙の画像は、筆者自作の **Hanamaru レンダラー**[*1]でレンダリングしたものです。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す。パストレーシングは 3D の描画手法のひとつで、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を可能にします。本章の前半では、3DCG の描画方法に関する基礎に触れてから、パストレーシングの概要を紹介します。

また、Hanamaru レンダラーの実装には Rust というプログラミング言語を使用しています。本章の後半では、実装の概要と高速化への取り組みに焦点を当てていきます。筆者にとって Rust による実装は初めてでしたので、感想や苦労した点も紹介したいと思います。なお、後半の内容と関連して、筆者のブログに Hanamaru レンダラーの解説記事[*2]を掲載しています。もし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。

1.1 3DCG の描画方法の基礎

まずは 3DCG の描画方法に関する基礎に触れていきしましょう。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

レイトレーシング法はオフラインレンダリング向けの 3DCG 描画手法で、じっくりと時間をかけて 1 枚のフレームを描画するのが特徴です。リアルタイムに描画する必要がない映画の CG などに利用されています。本章で扱うパストレーシングもレイトレーシング法の一種です。

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

[*2] 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！ `http://gam0022.net/blog/2017/10/02/rtcamp5/`

レイトレーシングで物を描画する仕組みは、私たち人間が物を見る仕組みにとても似ています。そもそも、私たち人間はどうして物を**見る**ことができるのでしょうか？ 太陽や蛍光灯（以下、光源）から放たれた光線（以下、レイ）は、周囲の物体に衝突したり反射したりして人間の目に到達します。目に到達したレイは網膜上の視細胞によって受容され、その電気信号が脳に伝わることで、私たち人間は物を見ることができます。網膜上に並んだ無数の視細胞ひとつひとつが光を受容することで、脳は物の形や色を認識します。カメラで写真を撮るときには、カメラのイメージセンサーが人間の網膜に相当します。

この一連の流れをコンピュータ上でシミュレートするのがレイトレーシング法です。パソコンのディスプレイは目の網膜に相当し、視細胞がピクセルに相当します。ディスプレイ上のすべてのピクセルについて、一連のシミュレーションを行うことでシーンを描画します。たとえば、フル HD の画面解像度であれば $1920 \times 1080 = 2073600$ ピクセルあるため、約 200 万回のシミュレーションが必要です。レイトレーシングは膨大な計算量を必要とするので、リアルタイムに描画する必要のない映画などに利用されるのですね。

カメラから光源へのレイの追跡

光源から放たれた多くのレイのうち、人間の目（カメラ）に到達するレイはほんの一部に過ぎません。光源からカメラに至るまでにはさまざまな経路があるので、それらすべてをシミュレートするには時間がかかりそうですよね。多くのレイトレーシング法のアルゴリズムでは、光源からではなくカメラからレイを飛ばして、効率的にレイの経路をシミュレートしています。

レイの反射においては入射方向と反射方向を入れ替えても同じ結果になる性質があります。これをヘルムホルツの相反性といいます。この性質によって、カメラからレイを飛ばしても光源からレイを飛ばしたときと同じ結果になるのですね。ちなみに屈折はヘルムホルツの相反性が成り立たないため、屈折を実装するときには注意が必要です。

カメラからレイを飛ばしてシーンを描画する手順はとても単純です。カメラから飛ばした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。

ラスタライズ法

ラスタライズ法はリアルタイムレンダリング向けの 3DCG 描画手法で、主にゲームの CG などに利用されています。

ラスタライズ法では、3D モデルを三角形のポリゴンの集合に近似し、次の 2 ステップで 3D シーンを 2D スクリーンに描画します。

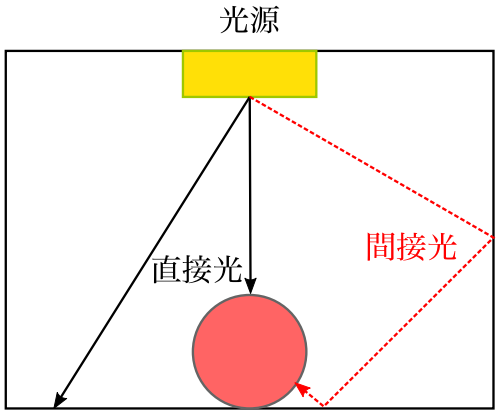
1. 3D 空間内のポリゴンの頂点を 2D スクリーン上の座標に投影する処理（頂点シェーダが担う処理）
2. 2D スクリーン上に配置されたポリゴンを塗りつぶすシェーディング処理（フラグ

メントシェーダが担う処理)

物体をたくさんの三角形の集合とみなし、三角形を対応する 2D のスクリーンに投影することによってレンダリングをします。GPU はラスタライズ法に特化した専用ハードウェアであるため、これらの処理を非常に高速に実行できます。

1.2 パストレーシングと大域照明

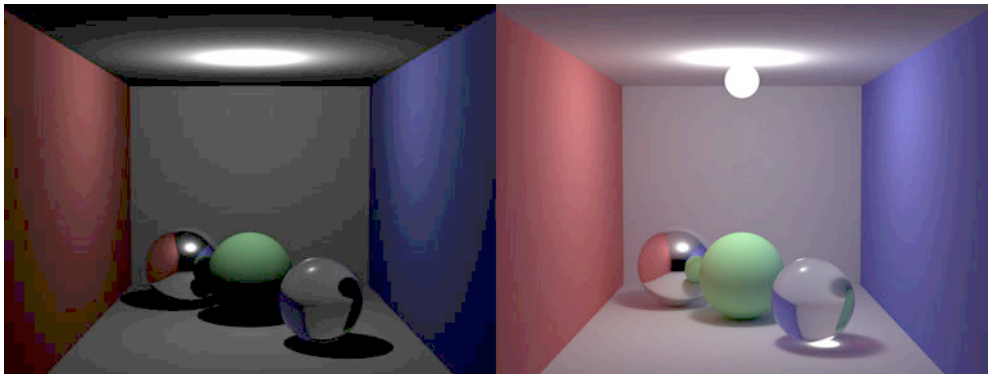
図 1.1 は狭い部屋の天井に蛍光灯の光源があり、床に球体のあるシーンを示しています。もっとも単純なレイトレーシング法では、図 1.1 の実線のように、光源から物体表面にまっすぐに到達した**直接光**しか扱えません。しかし現実世界では、光源を出た後に物体表面に直接到達せずに、他の物体に反射してから物体表面に到達する経路も存在します。図 1.1 の点線のように、壁や床に反射してから、球体の表面に到達するレイも存在することが分かるでしょう。直接光に対して、反射によって間接的に物体表面に到達するレイを**間接光**と呼びます。



▲図 1.1　直接光と間接光

本章で紹介するパストレーシングには「直接光」と「間接光」の両方を扱えるという強みがあります。直接光と間接光の双方を考慮したシェーディングを**大域照明**（グローバルイルミネーション）と呼びます。パストレーシングはレイトレーシング法の一種で、**大域照明**を実現できる手法です。

図 1.2 の左がレイトレーシングの結果（大域照明なし）、右がパストレーシングの結果（大域照明あり）です。左のレイトレーシングの結果では、球体の床側が暗く、間接光の大域照明が考慮されていません。一方、右のパストレーシングの結果では、球体の地面側も床などからの反射による間接光のある大域照明が考慮されています。ひと目で分かるように、パストレーシングの結果の方が明らかに写実的な仕上がりです。写実的なレンダリングを実現するためには、直接光に加えて間接光も考慮した大域照明が必要不可欠です。



▲図 1.2　（左）レイトレーシングの結果（直接光のみ = 大域照明なし）、（右）パストレーシングの結果（直接光＋間接光 = 大域照明あり）

パストレーシングでは間接光を扱うために、レイが何らかの物体に衝突した場合、物体表面の材質に応じて確率的にランダムな方向にレイを反射させます。カメラから出たレイが光源に到達すれば光の経路が存在するので、直接光か間接光かに関係なくシェーディングを行います。

このように 1 本の光の経路を探索する処理を**サンプリング**と呼びます。また、サンプリング時にレイの反射方向を決定するために用いる確率密度関数を **BRDF**（双方向反射率分布関数）と呼びます。サンプリングを何百回、何千回と繰り返し、サンプリング結果を加算平均したものが最終的なパストレーシングのレンダリング結果になります。

さまざまな材質（Material）を表現できる BRDF

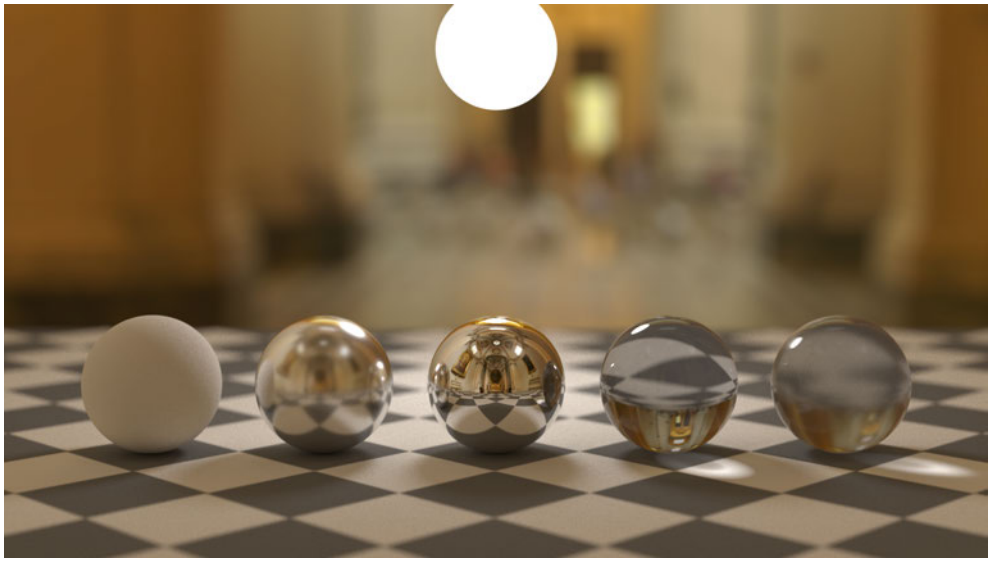
レイの反射方向を決定する **BRDF**（Bidirectional Reflectance Distribution Function, 双方向反射率分布関数）を変化させると、さまざまな**材質**（Material）を表現できます。

図 1.3 には、同じ色（白）・同じ大きさで材質だけが異なる 5 つの球体が描画されています。この材質の違いを、BRDF の違いで実現しています。

（1）完全拡散反射面は、石膏やチョークのような質感です。物体表面にある細かな凹凸により、入射した光がランダムな方向に拡散する材質をモデル化しています。BRDF としては一様分布の定数で表現されます。

（2）微小な凹凸のある鏡面は、金属のような質感です。今回は GGX と呼ばれる法線分布モデルを採用しました。Roughness というパラメータで物体表面の凹凸の程度を 0〜1 の範囲で制御できます。Roughness が 0 に近いほど（3）の表面が滑らかな完全鏡面反射に近づき、Roughness が 1 に近いほど表面がザラザラの鏡面になります。

（3）完全鏡面反射面は、表面に凹凸のまったくない滑らかな材質をモデル化したもので、鏡のような質感です。入射方向が決まれば、反射方向が 1 つに決まります。反射方向の一点が 1.0、それ以外は 0.0 の離散的な確率分布となります。



▲図 1.3　材質の紹介。左から順に（1）完全拡散反射面、（2）微小な凹凸のある鏡面、（3）完全鏡面反射面、（4）屈折面（ガラス面）、（5）微小な凹凸のある屈折面

（4）屈折面（ガラス面）は、光が屈折して物体内部を通過する材質をモデル化したもので、ガラスのような質感です。ただし、表面に入射したすべての光が屈折するわけではなく、一部の光は鏡面反射するので、図 1.3 をよく見ると、うっすらと周囲の景色が反射しています。鏡面反射する光の割合（反射率）はフレネルの式から計算できます。完全鏡面反射面と同様に、離散的確率分布の BRDF となります。（4）の球の近くの床を見ると、レンズと同じ原理によって光が集まって明るくなっている箇所があります。この集光模様は**コースティクス**と呼ばれます。コーティクスを観察できるのは、大域照明を考慮できている証拠です。

（5）微小な凹凸のある屈折面は、磨りガラスのような質感です。（4）の屈折面と（2）の GGX による法線分布を組み合わせたものです。

ちなみに（4）と（5）は**反射**（Reflectance）ではなく、**屈折**（Transmittance）であるため、厳密には **BTDF**（Bidirectional Transmittance Distribution Function, 双方向透過率分布関数）になります。反射と屈折の両方を扱う分布関数を **BSDF**（Bidirectional Scattering Distribution Function, 双方向散乱分布関数）と呼ぶこともあります。

レンダリング方程式とモンテカルロ積分

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、$L_0(x,\omega_0) = L_e + \int_\Omega f_s(x,\omega_i,\omega_0)\, L_i \cos\theta d\omega$ と定義します。

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を表します。

$\int_\Omega f_s(x, \omega_i, \omega_0)\ L_i \cos\theta d\omega$ 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。

大域照明を考慮したレンダリングの手法では、何らかの方法でこのレンダリング方程式を計算しています。レンダリング方程式を解くためには半球上の積分を計算する必要がありますが、これを解析的に計算することは困難です。

本節で紹介した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を用いて計算する手法といえます。モンテカルロ積分は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です。

1.3 レンダラーの紹介

Hanamaru レンダラーの主な機能を紹介します。ポリゴンとの衝突判定を BVH で高速化したり、薄レンズモデルによる**被写界深度**（レンズのピンボケ）を入れたり、**IBL**（Image Based Lighting, 全方位画像によるライティング）をしたりしました。機能一覧は次のとおりです。

- ポリゴンメッシュに対応
  - obj ファイルの読み込みに対応
  - BVH で衝突判定を高速化
- albedo / roughness / emission をテクスチャで指定可能
  - PBR 用のテクスチャ素材などを使える
- 薄レンズモデルによる被写界深度（レンズのピンぼけ）
- IBL
  - CubeMap に対応
- 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）

1.4 表紙のシーンの解説

表紙中心にあるモデルは、弊社 KLab のロゴをモチーフにしたオブジェです。第 5 章の『FPGA 初心者が試行錯誤しながら疑似乱数生成回路を作る話』を担当した於保がモデリングしました。前後左右どこから見ても「K」のマークに見えるような不思議な形状になっています。

周囲に散りばめられた宝石は、弊社の 3D デザイナーである下野がモデリングしました。実際のダイヤモンドは屈折率が非常に高く内部で全反射が起こりやすいため、複雑な輝きを発します。マテリアルには実際のダイヤモンドと同じ屈折率 2.42 を設定することで、ダイヤモンド独特の輝きを再現できました。

また左手前や右奥には、地球の形に光る球体を配置しました。被写界深度の実装により、実際のカメラのようにレンズ絞りの形状によるボケを再現できます。特に左手前の球体の大アンティル諸島あたりを見ると、綺麗に円形ボケしていますね。円形ボケを確認するために、あえて焦点が合わないように配置しました。

1.5 高速化の取り組み

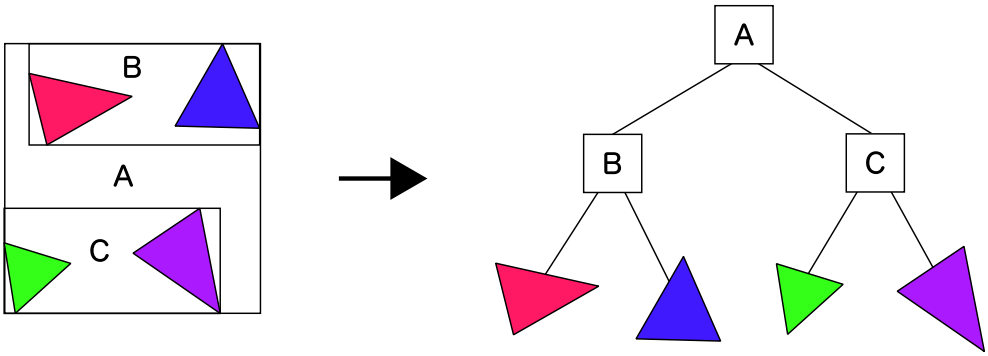
パストレーシング・レイトレーシングの高速化のアプローチはいくつかあります。本節では、Hanamru レンダラーで行った 3 つの高速化のアプローチを紹介します。

衝突判定の高速化

衝突判定はレイトレーシング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 1.4 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 1.4　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。

モンテカルロ積分のサンプリングの効率化

既に述べたように、パストレーシングはモンテカルロ法を使ってレンダリング方程式の解を求める手法です。このためモンテカルロ積分のサンプリングを効率化すれば、より少ないサンプリング数でノイズの少ない収束した結果を得られます。

サンプリングの効率化のための手法としては、重点的サンプリング、多重重点的サンプリング、Next Event Estimation などがあります。

今回はレンダリング方程式の cos 項に比例した重点的サンプリングのみを実装しました。レンダリング方程式の cos 項に比例した重点的サンプリングを行うと、確率密度関数で除算する処理でレンダリング方程式の cos 項を打ち消すことができ、パストレーシング実装をシンプルにできます。

今回のシーンでは IBL 光源が支配的であるため、cos 項に比例した重点的サンプリングのみで少ないサンプリング数でレンダリング結果が収束しましたが、光源が小さいシーンだと分散が大きいために収束が遅くなってしまいます。

並列処理

並列処理もレイトレーシングにおいてはとても重要です。たとえば 8 コアの CPU の環境では、並列処理をしないと計算リソースを全体の 1/8 しか利用できません。

Rust の標準ライブラリの `std::thread`でも並列化は実現できるのですが、CPU のコア数に応じてスレッドプールを作ったり、適切にスケジューリングしたりする部分はプログラマーが頑張る必要がありました。

ところが、Rayon[*3]という並列処理のライブラリを利用すれば、面倒なスケジューリングを自動でやってくれます。

リスト 1.1 は、サンプリング数を動的に変更できる並列化をしたレンダリングの関数です。Rayon では、`iter_mut()`の代わりに `par_iter_mut()`でイテレーターを作ると並列処理ができます。

▼リスト 1.1　Rayon によって並列化した render 関数

```
fn render(&mut self, scene: &SceneTrait, camera: &Camera,
    imgbuf: &mut ImageBuffer<Rgb<u8>, Vec<u8>>) {
  // 画面解像度
  let resolution = Vector2::new(imgbuf.width() as f64, imgbuf.height() as f64);

  // ピクセル数
  let num_of_pixel = imgbuf.width() * imgbuf.height();

  // レンダリングの途中結果を格納するためのバッファ
  let mut accumulation_buf = vec![Vector3::zero(); num_of_pixel as usize];

  // サンプリングごとに for ループ（サンプリングは並列処理しない）
  for sampling in 1..(self.max_sampling() + 1) {
    // Rayon の par_iter_mut() でピクセルごとの計算を並列処理
    accumulation_buf.par_iter_mut().enumerate().for_each(|(i, pixel)| {
      // ピクセルの添字 i を座標 x,y に変換
      let y = i as u32 / imgbuf.width();
      let x = i as u32 - y * imgbuf.width();

      // スクリーン座標をイメージセンサー上の座標に正規化
      let frag_coord = Vector2::new(x as f64, (imgbuf.height() - y) as f64);

      // 1 サンプリングの結果の画素値を計算するメソッドを呼び出し、その結果をバッファに加算
      *pixel += self.supersampling(scene, camera,
          &frag_coord, &resolution, sampling);
    });
  }

  // accumulation_buf を imgbuf にコピー
  Self::update_imgbuf(accumulation_buf, self.max_sampling(), imgbuf);
}
```

元のコードをほとんど変更せずに並列化できるのは素晴らしいですね。同じことを `std::thread`で書こうとすると、かなりコードが複雑になると思います。

[*3] https://github.com/nikomatsakis/rayon

1.6 Rust を選んだ理由

実は Hanamaru レンダラーを開発するまで Rust の経験は 0 でした。しかし調べているうちに、Rust はとてもレイトレーシングに向いていそうなことが分かりました。最終的には次の理由で Rust を採用することにしました。

- 実行時のパフォーマンスを重視している
  - 並列処理が簡単にできて、実行時のオーバヘッドが低いことはレイトレーシングには必須条件
- 高いポータビリティ
  - macOS でも Windows でも、コードの変更なしに動かせる
- Web ブラウザ上でも動かせる
  - WebAssembly の出力をサポートしている数少ない言語
- 弊社（KLab）の社員が 2 人も Rust でレイトレーシングしており、困ったら教えてもらおうと思った
  - 『Rust でレイトレーシング入門しました』[*4] by @trigott
  - 『hhatto/ray-tracing-in-one-weekend-in-rust』[*5] by @hhatto

1.7 初めて Rust で実装した感想・苦労

正直にいうと、Rust の学習コストはかなり高いと感じました。Rust の予習に 1 週間かけましたが、いざコードを書いてみると所有権や型のサイズが不定などのコンパイルエラーに阻まれ、心が折れそうになりました。特に所有権やライフタイムは自分にはまったく馴染みの無い概念で理解するのに苦労させられました。また所有権を理解できたとしても、所有権のルールを守りながらコードを書く難しさを痛感しました。

はじめこそ大変でしたが、慣れるうちにコンパイルエラーに阻まれることは少なくなりました。所有権を守るために設計から見直すこともありましたが、結果的に所有権を意識することでコードがスッキリしました。Rust は所有権という仕組みによってデータ競合を排除したり、ガーベジコレクタのない低オーバヘッドを実現したりしており、「学習コスト」対「安全性＋実行時パフォーマンス」のトレードオフだろうと感じました。C++ の経験がある人は『Rust は何が新しいのか（基本的な言語機能の紹介）』[*6]を読むと Rust の所有権の雰囲気を掴みやすいかと思います。

Rust には cargo というビルドツール兼パッケージ管理システムが標準搭載されていて、これはとても便利でした。`cargo run`を叩くだけで、自動的にパッケージをフェッ

[*4] https://qiita.com/kiritex/items/786dbed4b7a2a22cbb89
[*5] https://github.com/hhatto/ray-tracing-in-one-weekend-in-rust
[*6] https://imoz.jp/note/rust-functions.html

チし、依存関係を解決してビルドと実行ができます。開発前半と後半で Mac マシンから Windows マシンに開発環境を移行しましたが、cargo のおかげで macOS でも Windows でも同様にビルドができるので、開発環境を行き来するコストがまったくありませんでした。

Rust でレンダラーを開発して感じたことは、新しくプログラミング言語を習得する際のテーマとしてレイトレーシングの実装は大いに役に立つということです。もしレイトレーシングをすでに別の言語で実装した経験があったとしても、知らない言語でレイトレーシングを実装することで、パフォーマンスを意識した書き方や並列処理が身につくと強く感じました。もしレイトレーシングの実装が初めてであれば、実際に絵が出力されるので開発のモチベーションを維持しやすいですし、外部ライブラリに頼らず 3D のシーンを描画することで、コンピュータグラフィックスへの理解をかなり深めることができます。

1.8 まとめ

本章では、まず基礎的な 3DCG の描画方法について触れた後、パストレーシングの原理について簡単に説明しました。続いて筆者自作の Hanamaru レンダラーを紹介し、レイトレーシングやパストレーシングの高速化の取り組みについて触れました。また、初めて Rust で実装するにあたって感じたことや苦労したことを共有しました。本章を通じて、少しでもレイトレーシングに興味を持ってくだされば幸いです。

Sho HOSODA / @gam0022

第 2 章

Sprache を CPS 変換

C#のパーサーコンビネーター Sprache[*1]を継続渡しスタイル（CPS）に変換することで高速化できないかと試したのでその記録です。また、それを題材としてモナドとしての LINQ 式や、ランク N 多相についても取り上げます。

2.1 Sprache の紹介

Sprache はモナドを利用したパーサーコンビネーターライブラリーです。コンビネーターとは、関数を引数に取り関数を返す関数のことをいいます。Sprache を利用すれば、そのような関数を組み合わせてパーサーを作ることができます。

ちなみに、Sprache はドイツ語で「言語」という意味だそうです。

Sprache を使ったパーサーのサンプルを見てみましょう[*2]。“1+2+3”のような 10 進数の足し算の式をパースして同時に計算をするサンプルです。期待する入力はリスト 2.1 の EBNF で定義される文字列です。プログラムはリスト 2.2 です。

▼リスト 2.1　期待する入力の EBNF

```
decimal = (* 10 進数 *);
expression = decimal (+ decimal)*;
```

▼リスト 2.2　Sprache を使ったサンプル

```
using System;
using System.Linq;
using Sprache;

namespace SpracheSample
```

[*1] sprache/Sprache: Tiny C# Monadic Parser Framework https://github.com/sprache/Sprache

[*2] Sprache 2.1.0 を使用しています。

```
{
    class Program
    {
        static void Main(string[] args)
        {
            var addExpParser = from sa in Parse.Decimal                 // (1)
                               from a in Parse.Return(int.Parse(sa))    // (2)
                               from bs in (from plus in Parse.Char('+') // (3)
                                           from sb in Parse.Decimal     // (4)
                                           select int.Parse(sb))        // (5)
                                          .Many()                       // (6)
                                          .End()                        // (7)
                               select a + bs.Sum();                     // (8)

            var result = addExpParser.TryParse("1+2+3");
            Console.WriteLine(result.WasSuccessful);                   // => True
            if (result.WasSuccessful) Console.WriteLine(result.Value); // => 6

            result = addExpParser.TryParse("1+2-3");
            Console.WriteLine(result.WasSuccessful);                   // => False
        }
    }
}
```

それでは、順番に見て行きましょう。パーサーを定義してある部分は (1)〜(8) の部分です。LINQ を利用しています。

1. 10 進数をパースして、その結果の文字列を saに代入しています。Parse.Decimal は Parser<int>型です。
2. 文字列の saを intに変換しています。LINQ 内では新たに変数を導入するには from-in 構文を使うしかなさそうのなので、このような記述になっていますが、意味の上では a = int.Parse(sa)と同じです。ここまでで EBNF の decimal をパースしたことになります。Parse.Returnの型は Parser<T> Return<T>(T) です。
3. 3〜7 で EBNF の (+ decimal)*をパースします。パースした結果は IEnumerable<int>として bsに代入されます。3 では EBNF での + に相当します。
4. 1 と同じですね
5. + decimal のパース結果として decimal 部分に対応する int 値をパース結果として返しています。3〜5 の from plus ... select int.Parse(sb)は Parser<int>型です。
6. EBNF での*に相当します。Manyの型は、Parser<IEnumerable<T>> Many<T>(this Parser<T>)です。
7. Endは入力の終端にマッチすることを期待しています。Endはレシーバーをそのまま返します。3 の from bs in (...)の右辺が Parser<IEnumerable<int>>ということが分かったので、bsは IEnumerable<int>です。
8. addExpParserをパースした結果何が返るかを記述しています。これで、addExpParserは Parser<int>となります。

実行のしかたとその結果はそれ以降に記述しています。

パース結果をどう処理するかという部分が入っていますが、EBNF に対応するように書いていけることが分かったかと思います。

LINQ は IEnumerable専用の構文ではなく、Haskell の do 構文、Scala の for 構文と同じようにモナド用の構文なので、このような使い方ができます。モナド便利ですね。

2.2 CPS 変換とは

継続渡しスタイル（CPS）とは、関数の引数に継続を渡していくことで処理を繋げていくスタイルのことです。継続は、C-like な手続きプログラミングの言語ではコールバックとよく呼ばれます。

継続の例をリスト 2.3 にあげます。twoTimesと addTenの CPS 版がそれぞれ twoTimesCPSと addTenCPSです。twoTimesCPSの場合、「2 倍した値を取って続きをする関数」を contとして受け取っています。

▼リスト 2.3　通常のスタイルと継続渡しスタイル

```
// non-CPS
Func<int, int> twoTimes = n => 2 * n;
Func<int, int> addTen = n => n + 10;
Console.WriteLine(twoTimes(addTen(3)));

// CPS
Action<int, Action<int>> twoTimesCPS = (n, cont) => cont(2 * n);
Action<int, Action<int>> addTenCPS = (n, cont) => cont(n + 10);
addTenCPS(3, n => { twoTimesCPS(n, m => { Console.WriteLine(m); }); });
```

なんだかややこしくなっただけのような気がしますが、非 CPS のときに返り値のためにインスタンスを作る場合に、CPS 変換することでそれをしなくてよくなります。

たとえば、リスト 2.4 のような IntIntクラスがあるとします。そのとき、リスト 2.5 の twoTimesAndAddTen関数では times関数への引き渡しのために一時的に IntIntクラスのインスタンスを生成しています。ここでの興味は IntIntクラスのインスタンスを作ることではなく単に 2 つの整数を渡すことだけで、インスタンスを作ることは無駄です。

▼リスト 2.4　IntInt クラス

```
class IntInt
{
    public int Int1 { get; private set; }
    public int Int2 { get; private set; }
    public IntInt(int int1, int int2)
    {
        Int1 = int1; Int2 = int2;
    }
}
```

▼リスト 2.5　構築子に関する CPS

```
// non-CPS
Func<int, IntInt> twoTimesAndAddTen = n => new IntInt(2 * n, n + 10);
Func<IntInt, int> times = ii => ii.Int1 * ii.Int2;
Console.WriteLine(times(twoTimesAndAddTen(3)));

// CPS
Action<int, Action<int, int>> twoTimesAndAddTenCPS
    = (n, cont) => cont(2 * n, n + 10);
Action<int, int, Action<int>> timesCPS = (i1, i2, cont) => cont(i1 * i2);
twoTimesAndAddTenCPS(
    3,
    (i1, i2) => { timesCPS(i1, i2, n => { Console.WriteLine(n); }); }
);
```

CPS を使うと、twoTimesAndAddTenCPSのように 2 引数関数を継続としてもらいそれに 2 つの整数を渡せばよいので、中間の一時的なインスタンスを生成しなくてよくなります。これによって高速化と省メモリー化ができる場合があります。

ちなみに C#には匿名型があり、それを使えば var foo = new { field1 = 1, field2 = "two" };のように書けますが、右辺がラムダ式の代入文で varは使えません[*3]ので IntIntのようなクラス定義は必要です。

本章のメイントピックは、この技術を使って Sprache を書き換えていくというものです。

パーサーコンビネーターの実装に CPS を使って高速化することは、過去にも事例があり、Haskell では attoparsec[*4]というパーサーコンビネーターが CPS で実装され実際速いといわれています。今回、Sprache を変換するにあたって attoparsec のコードを参考にしました。

2.3　CPS 変換する上での問題

後は黙々と変換していけばいいはずと思っていたのですが、そう簡単にはいきませんでした。

まずは、どう変換していったのか紹介しましょう。リスト 2.6 が元の Parser型でリスト 2.7 が変換後の Parser型です[*5]。失敗時と成功時の継続を受け取る部分が追加されています。「2.2 CPS 変換とは」の例では継続は Action<T>のような型でしたが、今回は Func<T, S>のようになっています。これは最後までパースしたときに、CPS をやめ結果を返り値として返すようにするためです。Parser<T>が Action<T>になっていないのも

[*3] CS0815 https://msdn.microsoft.com/ja-jp/library/windows/apps/bb384140(v=vs.90)

[*4] bos/attoparsec: A fast Haskell library for parsing ByteStrings https://github.com/bos/attoparsec/

[*5] CPS 変換した Sprache https://github.com/kakkun61/Sprache/

同じ理由です。

▼リスト 2.6　元の Parser 型

```
public delegate IResult<T> Parser<out T>(IInput input);
```

▼リスト 2.7　CPS 変換後の Parser 型

```
public delegate IResult<object>
    Parser<out T>(IInput input, OnSuccess<T> onSuccess, OnFailure onFailure);
public delegate IResult<object> OnSuccess<in T>(T value, IInput remainder);
public delegate IResult<object>
    OnFailure(IInput remainder, string message, IEnumerable<string> expectations);
```

次は Orコンビネーターを例に実装を見てみましょう。Orコンビネーターは 1 つめのパーサーが失敗したときに、2 つめを試行するパーサーです。どちらかが成功するとそれが結果となり、両方とも失敗すると全体として失敗の結果となります。

▼リスト 2.8　元の Or コンビネーター

```
public static Parser<T> Or<T>(this Parser<T> first, Parser<T> second)
{
    return i =>
    {
        var fr = first(i);                                          // (1.1)
        if (!fr.WasSuccessful)                                      // (1.2)
        {
            return second(i).IfFailure(sf => DetermineBestError(fr, sf)); // (1.3)
        }

        return fr;
    };
}
```

▼リスト 2.9　CPS 変換後の Or コンビネーター

```
public static Parser<T> Or<T>(this Parser<T> first, Parser<T> second)
{
    return (input, onSuccess, onFailure) => {
        OnFailure onFailure_  =                                     // (2.1)
            (firstRemainder, firstMessage, firstExpectations) =>
                second(
                    input,
                    onSuccess,
                    DetermineBestError(
                        firstRemainder,
                        firstMessage,
                        firstExpectations,
                        onFailure
                    )
                );
        return first(input, onSuccess, onFailure_);                 // (2.2)
    };
}
```

変換前後とも返り値はパーサーなので、関数本体は return文のみとなります。return で返すものを見ると、前述のとおり、変換後は成功時・失敗時の継続を取るため引数が増えています。変換前では、まず 1 つめのパーサーを試行してその結果を得ています (1.1)。その成否で分岐しています (1.2)。DetermineBestErrorはエラーメッセージを作るユーティリティです (1.3)。変換後では、1 つめのパーサーを試行するのは (2.2) で、その前に「失敗したときに何をするのか」を構築しています (2.1)。

うまくいっているように見えます。どのようなときにうまくいかないのでしょうか？ リスト 2.7 を見たときに IResult<object>の objectとなっているのを見ておや？ と思った方はするどいです。そうです、Parser<T>の返り値は IResult<T>であるべきです。しかし、objectにして静的型検査を犠牲にしなければならない事情があるのです。それは Thenコンビネーター等で起こります。Thenコンビネーターはリスト 2.10 のように定義されます。Parser型がリスト 2.7 の定義のときでは、この Thenでうまく動きます。

▼リスト 2.10　Then コンビネーター

```
public static Parser<U> Then<T, U>(this Parser<T> first, Func<T, Parser<U>> second)
{
    return (i, onSuccess, onFailure) =>
    {
        OnSuccess<T> onSuccess_ =
            ((value, remainder) =>
                second(value)(remainder, onSuccess, onFailure)); // (1)
        return first(i, onSuccess_, onFailure);                  // (2)
    };
}
```

試しに Parserの定義をリスト 2.11 のようにしてみましょう。

▼リスト 2.11　IResult<T>とした Parser 型

```
public delegate IResult<T>
    Parser<T>(IInput input, OnSuccess<T> onSuccess, OnFailure<T> onFailure);
public delegate IResult<T> OnSuccess<T>(T value, IInput remainder);
public delegate IResult<T>
    OnFailure<T>(IInput remainder, string message,
                 IEnumerable<string> expectations);
```

リスト 2.10 と見比べてどこが問題になるか分かるでしょうか？ Visual Studio 2017 では (1) の second以降で次のようなエラーメッセージが出力されます。

> 型 'Sprache.IResult<U>' を 'Sprache.IResult<T>' に暗黙的に変換できません。明示的な変換が存在します (cast が不足していないかどうかを確認してください)

(2) の箇所では onFailureに対して次のエラーになります。

> 引数 3: は 'Sprache.OnFailure<U>' から 'Sprache.OnFailure<T>' へ変換することはできません。

どちらも型として Tを期待しているところが実際には Uになっていることが問題となっています。

型と実装のどちらかがまちがっているわけですが、この場合は型がまちがっています。Parser<T>の返り値は、場合によっては IResult<T>でなくてもいいはずです。というのも、CPS 変換後の Parser<T>はどんどん入れ子入れ子で呼んでいくわけですから、IResult<T>は組み合わせた最終のパーサーの結果と型が合っていればよいのです。その最終結果が入れ子の呼び出しを辿るように戻っていくのです。Parser<T>の Tと合うべきなのは OnSuccess<T>の Tです。

それならばと、リスト 2.12 のように、このパーサーの結果の型と全体での結果の型とを区別して Parser の型を付けた場合でも、結局は型が合いません。

▼リスト 2.12　このパーサーの結果の型と全体での結果の型とを区別した Parser の型

```
public delegate IResult<R>
    Parser<R, A>(IInput input, OnSuccess<R, A> onSuccess, OnFailure<R> onFailure);
public delegate IResult<R>
    OnSuccess<R, A>(R value, IInput remainder);
public delegate IResult<R>
    OnFailure<R>(IInput remainder, string message,
                 IEnumerable<string> expectations);

public static Parser<R, B> Then<R, A, B>(
    this Parser<R, A> first,
    Func<A, Parser<R, B>> second
)
{
    if (first == null) throw new ArgumentNullException(nameof(first));
    if (second == null) throw new ArgumentNullException(nameof(second));

    return (i, onSuccess, onFaulure) =>
    {
        OnSuccess<R, A> onSucc =
            ((value, remainder) => second(value)(remainder, onSuccess, onFaulure));
        return first(i, onSucc, onFaulure);
    };
}
```

どうすれば解決できるか、Haskell の CPS パーサーコンビネーター attoparsec のコードを見てみましょう。Thenにあたるものは (>>=)です。そのソースコードを、今回必要な部分が分かりやすいように改変するとリスト 2.13 のようになります。

▼リスト 2.13　attoparsec での (>>=)

```
(>>=) :: forall a b. Parser a -> (a -> Parser b) -> Parser b -- (1)

type Parser a = forall r.                                   -- (2)
                Input
              -> Failure   r
              -> Success a r
              -> IResult r
```

(1) で表しているのは、(>>=)関数（演算子）の型は、型引数として aと bを取って、引

数として Parser i aと a -> Parser i b を取って、Parser i bを返す関数型であるということです。リスト 2.12 と比べると C#にはある Rにあたるものがないこと以外は同じです。

(2) では Parser i a型の実際の型を定義しています（右辺に Parser aという別名を付けています)。全体の形としてはリスト 2.12 の Parserと同じです。異なるのは、また型引数の部分であり、attoparsec では aだけが型引数となっていて、rについては右辺で導入されています。つまり、Parser Intと型実引数を渡したあとでも rに関しては多相性を保ったままだということです。

この説明だと複雑で釈然と説明しにくいので、もっと簡単な例をリスト 2.14 に示します。この例では Fを呼ぶところで推論できないとエラーが出て、ではと Tの型を指定しようとすると intにしても stringにしても問題があることが理解できるでしょう。fの型は Fを呼ぶところでは一意に決めずに、そのもう 1 つ内側で多相のままにしておきたいわけです。

この「もう 1 つ内側」というのを表すのがリスト 2.13 の forall r.の部分です。これを内側で書くとそのレベルで多相にするということが表せます。一番外側を 1 としてどれだけ内側かを「ランク」という言葉で表し、それぞれの多相をランク 2 多相のように言います。

▼リスト 2.14　Id 関数を使った例

```
static void Main(string[] args) {
    F(Id);
}

T Id<T>(T t) { return t; }

void F<T>(Func<T, T> f) { // This is not sound!
    System.Console.WriteLine("{0}", f(1));
    System.Console.WriteLine("{0}", f("one"));
}
```

話を戻すと、まず Thenコンビネーターがうまく型付けできないかを探っていました。そして attoparsec を参考にするとランク 2 多相にすればいいことがわかりました。なのですが、もちろん C#にランク 2 多相はないので、リスト 2.12 の Rのところを object型にすることでリスト 2.7 のようになり、静的型検査をやめ動的型検査で対応することにしました。

もう 1 点問題があり、こちらは小さな問題ですが、Manyコンビネーターをまじめに CPS 変換したところコールスタックがあふれました。このため Manyはループを使った実装にしています。attoparsec では随所に inline 指定をしており、これによりスタックあふれを回避しているように思われます（未検証)。

2.4 パフォーマンス比較

さて、パーサー高速化をもくろむ話から CPS 変換の話になっていつの間にか型システムの多相性の話になっていましたが、無事変換できたのでパフォーマンスを計測してみましょう。計測対象プログラムは Sprache のサンプルとしてリポジトリーにあった `XmlParser`です。

CPS 変換によって内部は大きく変わりましたが、ユーザー向けのインターフェースはまったく変わらず、`XmlParser`の実装に手を付けずに動くのはよいところです。

ベンチマークをするにあたって次の 2 つが大変参考になりました。

- neue cc - C#のベンチマークドリブンで同一プロジェクトの性能向上を比較する方法[*6]
- C#で Type をキーにした Dictionary のパフォーマンス比較と最速コードの実装 - Grani Engineering Blog[*7]
  - このページ内の BenchmarkDotNet の使い方

元々のバージョンとして `https://github.com/kakkun61/Sprache` の c2cf535 リビジョン、変換後のバージョンとして 85c072e リビジョンを使用しました。

ベンチマーク結果は次のとおりです。C2cf535 が元々のバージョン、Current が変換後のバージョンです。

```
// * Summary *

BenchmarkDotNet=v0.10.9, OS=Windows 10 Redstone 2 (10.0.15063)
Processor=Intel Core i5-4440 CPU 3.10GHz (Haswell), ProcessorCount=4
Frequency=3026471 Hz, Resolution=330.4178 ns, Timer=TSC
  [Host]  : .NET Framework 4.7 (CLR 4.0.30319.42000), 32bit LegacyJIT-v4.7.2110.0
  LongRun : .NET Framework 4.7 (CLR 4.0.30319.42000), 32bit LegacyJIT-v4.7.2110.0

Job=LongRun  LaunchCount=3  TargetCount=100
WarmupCount=15

  Method |     Mean |     Error |    StdDev |   Median | Scaled |
-------- |---------:|----------:|----------:|---------:|-------:|
 C2cf535 | 2.541 ms | 0.0044 ms | 0.0224 ms | 2.537 ms |   1.00 |
 Current | 3.267 ms | 0.0043 ms | 0.0217 ms | 3.264 ms |   1.29 |

     Gen 0 | Allocated |
----------:|----------:|
  820.3125 |   2.47 MB |
 1097.6563 |    3.3 MB |
```

遅くなった上にメモリー使用量も増えてますね…… ということで、ボトルネックを探すために Visual Studio 2017 でプロファイルしたのですが、図 2.1 のような結果に。

[*6] `http://neue.cc/2017/08/20_557.html`
[*7] `http://engineering.grani.jp/entry/2017/07/28/145035`

関数名	呼び出し数	経過時間 % (子を含む)	経過時間 % (関数のみ)	平均経過時間 (子を含む)	平均経過時間 (関数のみ)
VirtCall:Sprache.Parser`1.Invoke(class Sprache.IInput,class Sprache.OnSuccess`1<!0>,class Sprache.OnFailure)	21,223	65.62	9.44	95.09	0.00

▲図 2.1　プロファイル結果

どの関数で作られた `Parser`型のクロージャーかというのが大切なのに、クロージャーは同一の型なら全部まとめられてしまうんですね。悲しい。はてさて、どうプロファイルしようかと悩んだのですが、VS 力のなさ（VS 使うのは今回がはじめて）と時間のなさで本記事はここでおしまいです。

計測を元にしない推測をすると、一時的な値を消せた代わりにクロージャーがたくさん作られるようになったことによる時間コストとメモリーコストが大きかったのではないかと考えています。Haskell の場合では、変数は再代入不可なので文脈への参照をクロージャーが保持しておく必要がなかった、またそれに加えて関数の参照透明性のため動的なクロージャー生成をしていないのではないかと考えています。

2.5　まとめ

- C#のモナドとしての LINQ 記法を紹介しました
- そのモナドを利用したパーサーコンビネーター Sprache の紹介をしました
- 継続渡しスタイル（CPS）とは何かと、その利点について説明しました
- Sprache を CPS 変換しました
- 多相性のランクと、C#の多相性の限界を見ました
- CPS 変換した Sprache のパフォーマンスを計測した結果、時間的空間的に悪化していました

うまいプロファイルのしかたを探してボトルネックを解消していきたいです。

岡本和樹 / @kakkun61

第 3 章

Emscripten で動画再生する

Emscripten は C/C++ コードを JavaScript に変換するコンパイラです[*1]。

Emscripten を使用すれば libpng や zlib のような C/C++ の資産を asm.js と呼ばれる JavaScript のサブセットに変換できます。ただ JavaScript に変換して実行するだけでなく、asm.js は型情報やさまざまな制限を加えて高速に動作するように設計されています。今日では asm.js の流れをくんだ WebAssembly が用いられるようになり、ネイティブに迫るパフォーマンスが出るようになりました。WebAssembly は Google Chrome、Microsoft Edge、Mozilla Firefox など多くの環境で動作するようになっています。asm.js と同様に Emscripten からコンパイル可能で、さらにはバイナリフォーマットであるため、asm.js に比べファイルサイズが小さくなります。それだけではなく、従来の JavaScript では使用できなかった SIMD（Single Instruction Multiple Data）なども使用できるようになっており、さらなるチューニングが可能です。こうなってくるとブラウザでどこまで実装できるのか、試したくなってくるのがプログラマではないでしょうか？ 本稿では Emscripten を使用してブラウザの動画再生機能を使わずに動画再生をしてみます。

[*1] `https://kripken.github.io/emscripten-site/index.html`

JavaScript と SIMD

SIMD とは並列処理の設計様式のひとつで、ひとつの命令で複数のデータを同時に演算することをいいます。たとえば 4 次元ベクトル A, B の加算処理を行う場合、4 次元ベクトルの成分 x, y, z, w をそれぞれ加算する必要があるため加算命令が 4 回必要となります。一方で Intel の SIMD CPU 拡張である SSE を使用すれば 1 命令でこれら 4 成分を加算できるため、高速にベクトルの加算処理を行えます。大量の頂点処理処理が必要なグラフィクス分野では非常に効果的なチューニングができます。SIMD 機能拡張は CPU によって命令セットが異なっており、スマートフォンでおなじみの ARM アーキテクチャでは NEON、Intel では SSE が使用可能です。Emscripten では SSE1, SSE2, SSE3, SSSE3 が使用可能で同じ Intel の SIMD 拡張である MMX や AVX は使用できません。かつてはこれらを JavaScript でも使用できるように SIMD.js という仕様が ECMAScript での標準化を目指して議論が続けられていましたが、標準化プロセスから外れてしまいました[a]。
ですが、SIMD は WebAssembly の仕様には今なお含まれており、Firefox Nightly など一部のブラウザで動作します。今後は JavaScript で SIMD が使いたい場合は SIMD.js ではなく、JavaScript から WebAssembly の SIMD API を呼ぶ形になるでしょう。

[a] `https://github.com/tc39/ecmascript_simd`

3.1 ブラウザでの動画再生

ブラウザには動画再生機能が実装されており、Video タグを使用すれば多くのブラウザで動画を簡単に再生することができます。再生できるコーデックはブラウザによって異なるのですが H.264、Ogg Theora、VP9 などが再生可能です。

本稿では Video タグを使用せずに Emscripten でビルドした動画ライブラリを使用して動画再生を行います。実装にあたり、最初は H.264 をブラウザで再生することを考えました。H.264 には多数の特許権が含まれており、使用に際してはライセンス料が必要となります。ですが、H.264 ライブラリである OpenH264[*2]を使用すればライセンス料を Cisco Systems 社が負担してくれるため、ある条件下においては無料で使用できます。ソースコードは修正 BSD ライセンスで公開されており、Emscripten でのビルドも可能です。しかし、OpenH264 は Cisco Systems 社によるビルド済みのバイナリのみが無料で

[*2] `http://www.openh264.org/`

利用可能となっており、Emscripten を使用したビルドはライセンス料が必要となる可能性があるようです[*3]。そのため WebM のコーデックとして用いられ、ライセンスフリーなコーデックとして開発されている VP9 及び、そのライブラリである libvpx を使用することにしました[*4]。

3.2 libvpx のビルド

libvpx をビルドするために Emscripten をインストールします。Emscripten の開発環境は Windows、Linux、Mac で準備できます[*5]。

本章では Ubuntu 16.04.3 LTS を使用します。

開発環境の準備ができたら libvpx のソースコードを clone します。

▼リスト 3.1　libvpx を clone

```
$ git clone https://www.webmproject.org/code/
```

次にビルドの準備として configure を実行します。configure のパラメータとしてビルドターゲットを指定できます。今回は generic-gnu をビルドターゲットとします。configure には Emscripten のツールである emconfigure を使用します。ですが、このままだとビルド時に一部箇所でエラーが発生してしまうためリスト 3.2 のように編集します。configure が成功すれば、あとはリスト 3.3 のように `make`コマンドを実行するだけでビルドが完了し、出力された libvpx.a をリンクして動画ファイルの読み込みライブラリを作成します。

▼リスト 3.2　configure を編集

```
--- a/build/make/configure.sh
+++ b/build/make/configure.sh
@@ -456,7 +456,7 @@ NM=${NM}

 CFLAGS  = ${CFLAGS}
 CXXFLAGS  = ${CXXFLAGS}
-ARFLAGS = -crs\$(if \$(quiet),,v)
+ARFLAGS = crs\$(if \$(quiet),,v)
 LDFLAGS = ${LDFLAGS}
 ASFLAGS = ${ASFLAGS}
 extralibs = ${extralibs}
```

[*3] http://www.openh264.org/faq.html
[*4] https://www.webmproject.org/code/
[*5] http://kripken.github.io/emscripten-site/docs/getting_started/downloads.html

▼リスト 3.3　libvpx を make する

```
$ emconfigure ../configure --target=generic-gnu
$ make
```

3.3　動画ファイルの読み込みライブラリの作成

WebGL を使ったレンダリングに先立ち、動画ファイルの読み込みライブラリを C/C++ で記述します。今回は動画コンテナとして libvpx のサンプルコードで使用されている IVF フォーマットを使用し、VP9 にエンコードします。IVF は非常にシンプルなフォーマットでリスト 3.4 のように FFmpeg で変換できるため、動画のテストに適しています。今回は Big Buck Bunny の Trailer[*6]をサンプルに使用しました。解像度は 720x400 です。

▼リスト 3.4　FFmpeg を使用した VP9 へのエンコード

```
$ ffmpeg -i trailer_400p.ogg -c:v libvpx-vp9 -an trailer_400p.ivf
```

まずは IVF をパース及び VP9 をデコードするクラスを作成し、JavaScript にそのインスタンスのポインタを渡す関数を作成します。作成する関数は `CreateReader`、`Next`、`GetBuffer`の 3 つです。動的リンクライブラリを作成する時と同様に JavaScript から呼び出す関数にはマングルを回避するために `extern "C"`を使用する必要があります。

`GetBuffer`で受け取ったテクスチャは YUV420 フォーマットであるため、Y の解像度に対して U と V の解像度は 1/4 です。YUV420 を RGB に変換する方法は弊社ブログにて Kyo Shinyuu が詳しく解説しています[*7]。

それぞれ解像度が異なるチャンネルをメモリの無駄なく配置するために、図 3.1 のように、下に Y、左上に U、右上に V を配置し 1 枚の 1 チャンネルアルファ画像としてしまいます。

[*6] https://peach.blender.org/
[*7] http://klabgames.tech.blog.jp.klab.com/archives/1054828175.html

▲図 3.1　1 チャンネルの YUV テクスチャに変換

▼リスト 3.5　libvpx のサンプルコードを参考に動画ファイルの読み込みライブラリを書く

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "vpx/vpx_decoder.h"

#include "tools_common.h"
#include "video_reader.h"
#include "vpx_config.h"

// サンプルコードが内部で呼び出すことがあるために定義しておく
void usage_exit(void) {
  exit(EXIT_FAILURE);
}

// IVF 読み込みクラス
class IVFReader
{
private:
    size_t frame_size;
    int32_t frame_cnt;
    vpx_codec_ctx_t codec;
    VpxVideoReader *reader;
    vpx_codec_iter_t iter;
```

```
    vpx_image_t *img;

    const uint8_t *frame;
    const VpxInterface *decoder;
    const VpxVideoInfo *info;

public:
    IVFReader(const char *inputFileName);
    bool Next();
    bool GetBuffer(uint8_t *buffer);
};

// 動画ファイルの読み込み
IVFReader::IVFReader(const char *inputFileName)
{
    frame_size = 0;
    frame_cnt = 0;
    reader = nullptr;
    frame = nullptr;
    decoder = nullptr;
    info = nullptr;
    iter = nullptr;
    img = nullptr;

    reader = vpx_video_reader_open(inputFileName);
    if (!reader)
        die("Failed to open %s for reading.", inputFileName);

    info = vpx_video_reader_get_info(reader);

    decoder = get_vpx_decoder_by_fourcc(info->codec_fourcc);
    if (!decoder)
        die("Unknown input codec.");

    printf("Using %s\n", vpx_codec_iface_name(decoder->codec_interface()));
    printf("%s: %dx%d\n", inputFileName, info->frame_width, info->frame_height);

    if (vpx_codec_dec_init(&codec, decoder->codec_interface(), NULL, 0))
        die_codec(&codec, "Failed to initialize decoder.");
}

// 1 フレームデコード
bool IVFReader::Next()
{
    if(reader == nullptr) {
        return false;
    }

    while(true) {
        if(img == nullptr ) {
            if(vpx_video_reader_read_frame(reader) == 0) {
                break;
            }
            frame_size = 0;
            frame = vpx_video_reader_get_frame(reader, &frame_size);
            if(vpx_codec_decode(
                &codec,
                frame,
                (unsigned int)frame_size,
                NULL, 0)
              )
                die_codec(&codec, "Failed to decode frame.");
        }
```

```
        if ((img = vpx_codec_get_frame(&codec, &iter)) != NULL)
        {
            ++frame_cnt;
            return true;
        }
    }

    printf("Processed %d frames.\n", frame_cnt);
    if (vpx_codec_destroy(&codec))
        die_codec(&codec, "Failed to destroy codec");

    vpx_video_reader_close(reader);

    reader = nullptr;

    return false;
}

// フレームを取得する
bool IVFReader::GetBuffer(uint8_t *dst) {
    if (!img)
    {
        return false;
    }

    int32_t plane;
    for (plane = 0; plane < 3; ++plane)
    {
        const uint8_t *buf = img->planes[plane];
        const int32_t stride = img->stride[plane];
        const int32_t w = vpx_img_plane_width(img, plane) *
                          ((img->fmt & VPX_IMG_FMT_HIGHBITDEPTH) ? 2 : 1);
        const int32_t h = vpx_img_plane_height(img, plane);
        int y;

        for (y = 0; y < h; ++y)
        {
            switch(plane) {
            case 0:
                // Y チャンネルをコピー
                memcpy(dst + w * (h - y - 1), buf, w);
                break;
            case 1:
                // U チャンネルをコピー
                memcpy(dst + (info->frame_width * info->frame_height)
                  + 2 * w * (h - y - 1), buf, w);
                break;
            case 2:
                // V チャンネルをコピー
                memcpy(dst + (info->frame_width * info->frame_height)
                  + w + 2 * w * (h - y - 1), buf, w);
                break;
            }

            buf += stride;
        }
    }

    return true;
}

extern "C" {
    // フレームをバッファにコピーする
```

```
    bool GetBuffer(IVFReader *pDecoder, uint8_t *buffer)
    {
        return pDecoder->GetBuffer(buffer);
    }

    // 動画ファイル名を指定してリーダーを作成する
    IVFReader* CreateReader(const char *inputFileName)
    {
        return new IVFReader(inputFileName);
    }

    // 1 フレームすすめる。
    bool Next(IVFReader *pDecoder)
    {
        return pDecoder->Next();
    }
}
```

3.4 動画レンダラの作成

デコードしたフレームのレンダリングには WebGL を使用し、こちらは JavaScript で記述します。WebGL API をそのまま使用するとコード量が多くなってしまうため今回は補助ライブラリである TWGL[*8]を使用します。

JavaScript コードはビルドされた JavaScript の先頭に連結される pre.js に記述します。pre.js はコンパイル時に`--pre-js`で指定し、Module オブジェクトを作成することにより WebAssembly の設定を記述できます。動画ファイルはブラウザ上の仮想ファイルシステムから読み出すため、HTTP サーバー上に配置した上で WebAssembly の実行前に呼び出される `preRun`関数内で仮想的なファイルパスを登録します。その他の処理は WebAssembly のコード実行後に実行される `postRun`関数に記述します。`Module.cwrap`を使用すれば WebAssembly 側の関数を呼び出せます。第 1 引数には呼び出す関数名、第 2 引数には戻り値の型、第 3 引数には関数に与える引数の型を指定します。今回は C++ における bool 型及びポインタ型も JavaScript では number 型としています。WebAssembly におけるポインタは `Module.HEAPU8`のオフセットであり、この性質を知っていれば JavaScript 側から WebAssembly 側の世界のメモリへ簡単にアクセスできます。リスト 3.6 では `GetBuffer`でデコードされたフレームの受取先メモリを確保するために `Module._malloc`でテクスチャ用のメモリを確保し、引数として与えるようにしました。

[*8] http://twgljs.org/

▼リスト 3.6　JavaScript から呼び出す pre.js

```
var Module = {
  preRun: function() {
      "use strict";
      // Web 上のパスと仮想ファイルシステムを結びつける
      FS.createFolder( '/', 'assets', true, false);
      FS.createPreloadedFile(
        '/assets',
        'trailer_400p.ivf',
        './assets/trailer_400p.ivf',
        true, false);
  },
  postRun: function() {
      "use strict";

      // WebAssembly 側の関数を取得
      var CreateReader = Module.cwrap('CreateReader', 'number', ['string']);
      var Next = Module.cwrap('Next', 'number', ['number']);
      var GetBuffer = Module.cwrap('GetBuffer', 'number', ['number', 'number']);

      // WebGL を初期化
      var gl = twgl.getWebGLContext(document.getElementById("c"));

      // シェーダをコンパイル
      var programInfo = twgl.createProgramInfo(gl, ["vs", "fs"]);

      // WebGL レンダリング用ポリゴンを作成
      var arrays = {
          position: [-1, -1, 0, 1, -1, 0, -1, 1, 0, -1, 1, 0, 1, -1, 0, 1, 1, 0],
      };
      var bufferInfo = twgl.createBufferInfoFromArrays(gl, arrays);

      var name = 'assets/trailer_400p.ivf'; // ファイル名
      var width = 720; // 画像の横幅
      var height = 600; // アルファ画像の高さ H + H/2 = 400 + 200
      var fps = 25.0;

      // テクスチャ用メモリの確保
      var texSize = width * height;
      var pTex = Module._malloc(texSize);
      var pBuffer = new Uint8Array(Module.HEAPU8.buffer, pTex, texSize);

      // テクスチャを作成
      // アルファチャネル 1 チャネルを使用します。
      var tex = twgl.createTexture(gl, {
          mag: gl.LINEAR,
          min: gl.LINEAR,
          format: gl.ALPHA,
          src: pBuffer,
          width: width,
          height: height
      });

      var prevFrame = -1;

      var decoder = CreateReader(name);

      function render(time) {
          twgl.resizeCanvasToDisplaySize(gl.canvas);
          gl.viewport(0, 0, gl.canvas.width, gl.canvas.height);

          var currentFrame = (fps * time * 0.001) | 0;
```

```
            if(prevFrame < currentFrame ) {
                // 次のフレームがあればフレームを取得して更新
                if(Next(decoder)) {
                    GetBuffer(decoder, pTex);
                    gl.texImage2D(gl.TEXTURE_2D,
                        0,
                        gl.ALPHA,
                        width,
                        height,
                        0,
                        gl.ALPHA,
                        gl.UNSIGNED_BYTE,
                        pBuffer);
                }
                prevFrame = currentFrame;
            }

            var uniforms = {
             resolution: [gl.canvas.width, gl.canvas.height], video: tex
            };

            gl.useProgram(programInfo.program);
            twgl.setBuffersAndAttributes(gl, programInfo, bufferInfo);
            twgl.setUniforms(programInfo, uniforms);
            twgl.drawBufferInfo(gl, bufferInfo);

            requestAnimationFrame(render);
        }

        requestAnimationFrame(render);
    }
};
```

最後に WebGL レンダリングを行う HTML を作成するとともに YUV を RGB に変換するフラグメントシェーダを記述します。1 枚のテクスチャに図 3.1 のように YUV が記録されているため、3 度サンプリングを行います。この際に各チャネルの境界をサンプリングしないように半テクセルオフセットをのせています。

▼リスト 3.7　動画を再生する HTML

```
<!DOCTYPE html>
<head>
    <meta charset="UTF-8">
    <script id="vs" type="text/vertex-shader">
    attribute vec4 position;
    uniform vec2 resolution;
    varying vec2 uv0;
    varying vec2 uv1;
    varying vec2 uv2;

    void main() {
        vec2 offset = 0.5 / resolution;
        vec2 uv = vec2(position.x, position.y) * 0.5 + 0.5;

        // それぞれ Y, U, V のフェッチ座標を計算する
        uv0 = vec2(uv.x, uv.y * 2.0 / 3.0);
        uv1 = vec2(uv.x * 0.5, (2.0 + uv.y) / 3.0 + offset.y);
        uv2 = vec2((uv.x + 1.0) * 0.5
          + offset.x, (2.0 + uv.y) / 3.0 + offset.y);
```

```
        gl_Position = position;
    }
    </script>

    <script id="fs" type="text/fragment-shader">
    precision mediump float;

    uniform sampler2D video;

    varying vec2 uv0;
    varying vec2 uv1;
    varying vec2 uv2;

    void main() {
        // YUV をサンプル
        float y = texture2D(video, uv0).a;
        float u = texture2D(video, uv1).a - 0.5;
        float v = texture2D(video, uv2).a - 0.5;

        // YUV から RGB に変換
        gl_FragColor = vec4(
          y + 1.402 * v,
          y - 0.334 * u - 0.714 * v,
          y + 1.772 * u, 1.0);
    }
    </script>
    <script src="./js/twgl-full.js"></script>
    <script src="./build/VPXReader.js"></script>
</head>

<body>
    <canvas id="c" width="720" height="400"></canvas>
</body>

</html>
```

3.5 動画レンダラのビルド

Emscripten でのビルドには CMake[*9]を使用することもできます。

その場合、emcmake コマンドを使用し、ジェネレータ（-G オプション）には"Unix Makefiles"を指定します。リンク時のパラメータを`-s WASM=1`とすると WebAssembly として出力できます。また関数をエクスポートするため`-s EXPORTED_FUNCTIONS`に関数名を列挙するのですが、このとき先頭に`_`をつけるのを忘れないようにしましょう。

▼リスト 3.8　CMakefile.txt の書き方

```
cmake_minimum_required(VERSION 3.0.0)
project(VPXReader VERSION 1.0.0)

# libvpx を参照する
include_directories(./libvpx)
include_directories(./libvpx/build)
```

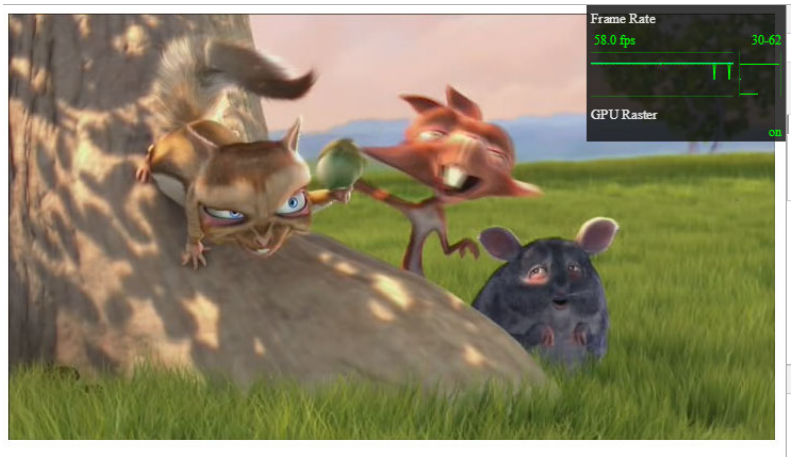
[*9] https://cmake.org/

```
link_directories(./libvpx/build)

# コンパイル時の最適化を有効に
set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -O3")

set(CMAKE_EXE_LINKER_FLAGS "\
${CMAKE_EXE_LINKER_FLAGS} \
--use-preload-plugins \
--pre-js ${CMAKE_SOURCE_DIR}/js/pre.js \
-s ALLOW_MEMORY_GROWTH=1 \
-s WASM=1 \
-s EXPORTED_FUNCTIONS=\"['_CreateReader', '_Next', '_GetBuffer']\" \
")

// コンパイルするソースコード。一部 libvpx のサンプルプログラムもコンパイルする。
add_executable(
  VPXReader
  src/VPXReader.cpp
  libvpx/video_reader.c
  libvpx/tools_common.c
  libvpx/ivfdec.c
)

# libvpx をリンクする
target_link_libraries(VPXReader vpx)
```

▼リスト 3.9　動画レンダラのビルド

```
$ emcmake cmake path/to/project -G "Unix Makefiles"
$ make
```

これで晴れてブラウザで動画が再生できるようになりました。著者の Core i7-2600 環境でも 60FPS でレンダリングできています。サンプルページ（https://nyamadan.github.io/IVF/index.html）で動画が再生されることを確認してみてください。フレームレートは落ちてしまいますが WebAssembly に対応していればスマートフォンブラウザでも動画を再生できます。



▲図 3.2　実行結果

3.6　まとめ

今回は少し駆け足でしたが、libvpx を使用して WebGL で VP9 の動画をレンダリングする方法を紹介しました。ブラウザにはすでに動画再生機能が実装されているため、この挑戦はあまり意味のないことのように思えるかもしれません。しかし、実用的な速度が出せたことや、RGB でなく YUV 色空間のデータが取得できるなど既存の仕組みでは難しかったことが可能となりました。JavaScript との連携もそこまで難しいものではないため動画に限らず今後はパフォーマンスの求められない場合は JavaScript、パフォーマンスが重要な場所では WebAssembly といった使い分けが進んでいくのかもしれません。新しいウェブの形にわくわくしてきませんか？

やまだ

第 4 章

テキストマクロプロセッサ「M4」のチューリング完全性について

M4 とは、1977 年にブライアン・カーニハンとデニス・リッチー（いわずと知れた K&R その人らである）によって作成されたテキストマクロプロセッサである[*1]。POSIX にも含まれる基本的なコマンドだが[*2]、読者諸氏の多くはこのコマンドを明示的に使ったことは無いだろう。しかしながら、間接的に M4 の恩恵を受けている人は少なくないはずだ。

`configure`スクリプト。数多くの OSS で採用されているこのビルド設定スクリプトは、GNU Autoconf により呼び出された M4 によって生成されている[*3]。他にも Sendmail の設定ファイルの生成に利用されるなど[*4]、縁の下の力持ちといったツールである。

登場から 40 年となるこのマクロプロセッサは、驚くべきことにチューリング完全性を備えている。つまり、コンピュータ黎明期に作られたプログラミング言語のひとつと言っても過言ではない。しかし、テキストマクロプロセッサ、すなわちテキストの置換という単純作業の繰り返しがチューリング完全になるとは俄に信じがたい。

この章ではそれを確かめるべく、M4 のチューリング完全性の証明を試みる。

4.1 チューリング完全性の証明方法

チューリング完全性の一般的な証明方法として、すでにチューリング完全であることがわかっている言語の処理系を実装するという手法がある。言い換えると、チューリング完全な言語で書かれたプログラムをすべて実行できるならば、それはチューリング完全である。

ここでは、チューリング完全な言語である Brainfuck[*5]（BF）を M4 で実装することに

[*1] http://wolfram.schneider.org/bsd/7thEdManVol2/m4/m4.pdf
[*2] http://pubs.opengroup.org/onlinepubs/9699919799/utilities/m4.html
[*3] https://www.gnu.org/software/autoconf/autoconf.html
[*4] http://www.sendmail.org/~ca/email/doc8.12/cf/m4/intro_m4.html
[*5] http://www.muppetlabs.com/~breadbox/bf/

より、M4 のチューリング完全性を証明する。

4.2 実装方法

この節では、M4 による BF の具体的な実装方法について解説する。なお、ここで扱う M4 は GNU M4[*6] とし、実装した M4 の完全なコードは GitHub にて公開しているので[*7]、適宜参照されたい。

BF コードの読み込みとマクロ列への変換

まず BF のコードは標準入力から受け取るものとし、“include”マクロを利用して/dev/stdinをまるごと取り込むようにした。

M4 では BF の命令に使われているような記号そのものをマクロとして処理できないため、取り込んだ BF コードを表 4.1 にしたがって、BF 命令 1 文字ずつ対応する文字列に変換し、マクロの列に変換した。この処理を行う M4 マクロをリスト 4.1 に示す。

▼表 4.1　BF 命令と変換文字列の対応表

	BF 命令	変換文字列
ポインタのインクリメント	>	‘’ip
ポインタのデクリメント	<	‘’dp
ポインタの指す値のインクリメント	+	‘’ic
ポインタの指す値のデクリメント	-	‘’dc
ポインタの指す値の表示	.	‘’pr
ループ開始点	[	‘’bn(‘
ループ終点	]	’)‘’ed

▼リスト 4.1　BF コードを読み込みマクロ列へ変換する

```
# 引数の 1 文字を対応する文字列に変換するマクロ
define(‘token’, ‘changequote({,})ifelse(
                {$1}, {>}, {‘’ip},
                {$1}, {<}, {‘’dp},
                {$1}, {+}, {‘’ic},
                {$1}, {-}, {‘’dc},
                {$1}, {.}, {‘’pr},
                {$1}, {[}, {‘’bn(‘},
                {$1}, {]}, {’)‘’ed}){}changequote’)

# 1 文字ずつ token マクロにかけるマクロ
# 引数の長さが 0 になるまで再帰的に呼び出す
define(‘parse’, ‘ifelse(
        eval(len(‘$*’)>0), 1,
        ‘token(substr(‘$*’,0,1))‘’parse(substr(‘$*’,1))’)’)
```

[*6] https://www.gnu.org/software/m4/m4.html
[*7] https://github.com/makiuchi-d/bfm4

```
# 標準入力をまるごと parse マクロにかけ、その結果を internal として定義する
define(‘internal’, parse(include(‘/dev/stdin’)))
```

ここで定義した“parse”マクロは、引数文字列の先頭 1 文字を取り出して“token”マクロに渡し、残りの文字列を自分自身に再び渡すという、再帰的なマクロである。

このマクロにより、たとえば“!”1 文字を表示する BF のプログラムを変換するとリスト 4.2 のようになる。

▼リスト 4.2　“!”を表示する BF プログラムと変換後のマクロ列

```
変換前：
>++++[-<++++++++>]<+.

変換後（改行は含まない）：
‘’ip‘’ic‘’ic‘’ic‘’ic‘’bn(‘‘’dc‘’dp‘’ic‘’ic‘’ic‘’ic‘’ic‘’ic‘’ic‘’ip’)
‘’ed‘’dp‘’ic‘’pr
```

ここで“‘’”は空文字列で、マクロ同士のデリミタとして働く。また、BF においてループを表す“[〜]”は、“bn(‘〜’)‘’ed”のように展開され、間に入るマクロ列がまるごとクォートされた文字列として bnマクロの引数になるように変換される。

ポインタとバイト配列

BF のメモリ空間は、30000 要素以上のバイト配列とその上を自由に動くポインタで構成される。

M4 には配列が無いため、バイト配列をそのまま実装することはできない。そこで、数字入りのマクロを動的に定義することで擬似的な配列とした。つまり、“_0_”、“_1_”、…“_30000_”といった具合のマクロを配列の要素として扱う。

ポインタは“ptr”というマクロを初期値 0として定義し、この定義を動的に書き換えることで実現した。

ポインタの値は“defn”マクロで“ptr”の定義を取り出せば参照でき、ポインタの指す値は“_”でポインタの値を挟んだマクロ名の定義を取り出せばよい。これらの処理の実装をリスト 4.3 に示す。

▼リスト 4.3　ポインタとバイト配列の実装

```
# ポインタの定義．0 で初期化
define(‘ptr’, ‘0’)

# 擬似的な配列からポインタの指す値を取り出すマクロ
define(‘cv’, ‘eval(defn(‘_’defn(‘ptr’)‘_’)+0)’)
```

“cv”マクロは“eval(〜+0)”と実装した。これにより、たとえば“_1_”が未定義だった場合その定義は空文字列になるので、“cv”は“eval(+0)”となり、“0”に展開さ

れる。バイト配列の要素の初期値は 0 クリアされているという BF の仕様をこのような形で実現した。

各命令に対応するマクロの実装

ここから BF の各命令に対応するマクロを実装していく。

ポインタのインクリメント・デクリメント（><）

前述のとおり、ポインタの操作は“ptr”マクロを動的に書き換えることで実現した。

▼リスト 4.4　ポインタのインクリメント・デクリメントの実装

```
# ポインタのインクリメント (>)
define(‘ip’, ‘define(‘ptr’, incr(defn(‘ptr’)))‘’’)

# ポインタのデクリメント ptr (<)
define(‘dp’, ‘define(‘ptr’, decr(defn(‘ptr’)))‘’’)
```

M4 では“define”マクロで既存の定義を上書きできることに加え、“incr”・“decr”という引数の値をインクリメント・デクリメントするマクロも用意されている。

これを利用して、「“defn(‘ptr’)”で取り出したポインタの値を“incr”マクロに渡し、その結果を新たな“ptr”として定義しなおす」という文字列に展開されるマクロとして“ip”を定義した。つまり、処理対象のテキストに“ip”が現れるたび、“define(‘ptr’, incr(defn(‘ptr’)))‘’”という文字列に置き換わり、それが処理される。

ポインタのデクリメントは“incr”の代わりに“decr”を使う以外は同じである。

ポインタの指す値のインクリメント・デクリメント（+-）

この命令も M4 の“incr”・“decr”を利用し、すでに定義した“cv”マクロで取り出した値を処理するようにした。

書き換えるべきマクロ名は、前述のとおり、“_”でポインタの値を挟んだ文字列である。

▼リスト 4.5　ポインタの指す値のインクリメント・デクリメントの実装

```
# ポインタの指す値のインクリメント (+)
define(‘ic’, ‘define(‘_’defn(‘ptr’)‘_’, incr(cv))‘’’)

# ポインタの指す値のデクリメント (-)
define(‘dc’, ‘define(‘_’defn(‘ptr’)‘_’, decr(cv))‘’’)
```

ループ（[]）

▼リスト 4.6　ループの実装

```
# ループ開始点（[）
define(‘bn’, ‘pushdef(‘s’, ‘ifelse(eval(cv>0), 1, ‘$1‘’s’)’)’)

# ループの終点（]）
define(‘ed’, ‘s‘’popdef(‘s’)’)
```

BF で“[ ]”に囲まれた部分に対応するマクロ列が、クォートされた文字列として“bn”マクロの引数になることはすでに解説した。

“bn”では“s”という名前の、「“cv”が 0 より大きい間、引数（$1）をそのまま展開してから“s”自身を呼び出す」という再帰的なマクロを定義している。そしてループ終点にあたる“ed”マクロにより、“bn”で定義された“s”が展開され、ループが始まる。

ここで“s”の定義には“define”ではなく“pushdef”を使っている。M4 では“pushdef”・“popdef”によりマクロの定義をスタックできる。これを利用し、多重ループでも“s”の定義を内側のループで上書きしてしまわないよう実装した。

ポインタの指す値の表示（.）

M4 には C 言語の“printf”同様のフォーマットで書式化できる“format”マクロがある。ポインタの指す値をそのまま“%c”で書式化する方法で実装した。

▼リスト 4.7　ポインタの指す値を表示するマクロの実装

```
define(‘pr’, ‘format(‘%c’, cv)’)
```

プログラムの実行

ここまでで BF の各命令に対応するマクロが定義できた。この後に BF プログラムから変換しておいたマクロ列、すなわち“internal”マクロを展開することで、マクロ列の中の命令マクロが順次展開され、プログラムが実行される。

4.3　チューリング完全性の証明

これにて M4 による BF の実装は完了である。実際に動作するかどうかはぜひ各々で検証してほしい。

さて、ここで BF に詳しい読者諸氏であれば、命令がひとつ足りないことに気づいたであろう。そう、標準入力から 1 文字読み込む“,”命令である。

今回の実装では標準入力を BF プログラムの入力としてしまっている都合上、“,”にあ

たる命令を実装することは不可能である。しかし、その命令によって入力したい文字がたとえば“A”ならば、“,”の代わりに“+”を 65 個並べることで代用できる。つまり、すべての“,”は別の BF コードに置き換えることが原理的に可能なので、この命令が無かったところでチューリング完全性が失われることはない。

このように、M4 により動作する BF のチューリング完全なサブセットを実装することができた。したがって、**テキストマクロプロセッサ「M4」はチューリング完全である。**

4.4 終わりに

広く使われているテキストマクロプロセッサといえば、C 言語のプリプロセッサを思い浮かべる人も多いだろう。しかし M4 と違い、C プリプロセッサ単体ではチューリング完全ではない。この差の要因のひとつは、マクロの再帰的な呼び出し（あるいはループ）ができるかどうかである。

テキストマクロプロセッサにチューリング完全性が必要かどうかはさておき、M4 はチューリング完全性を備えた強力なツールであることは理解していただけたと思う。そしてその歴史も長く広く使われており、これ以上ないほど安定している。

もし定型的なテキストを量産するような作業に出会った時、貧弱なテンプレートエンジンを自作する前に M4 のことを思い出してみてはいかがだろうか。書式はややまどろっこしいかもしれないが、やりたいことがすべてできることは証明済みだ。

まあそんなこと言っても積極的に使ったりしないですけどね。

Makiuchi Daisuke / @makki_d

第 5 章

FPGA 初心者が試行錯誤しながら疑似乱数生成回路を作る話

本稿では私が Maker Faire Tokyo 2017 に出展した FPGA を用いた疑似乱数生成回路を作る過程を解説します。

FPGA とは、Field-Programmable Gate Array の略で、チップ製造後に論理回路の構成を変更できるものです。ハードウェア記述言語（HDL）と呼ばれるプログラム言語的なものを使って、実装したい論理回路を記述し、その HDL をコンパイルしてできた構成情報のデータを PC などからチップに書き込むことで、任意の論理機能を実現可能なチップです。最近の FPGA は、HDL で記述した CPU を搭載してコントローラーにしたり、専用の演算回路を構成して処理の高速化を行ったりと多岐に渡る用途で使われています。ソフトウェアで処理するより省電力・高速にできたり、専用チップ（ASIC）と違い試行錯誤が楽であったりと、ソフトウェアとハードウェアの真ん中あたりの利点があることから、今後もさまざまな応用が期待されています。

筆者自身は FPGA について興味を持っていたものの、評価ボードの価格の高さやハードウェアのとっつきにくさから、手を付けあぐねていました。パラレルポートが必要とか無理ゲーかよと思ったこともありました。

そんな折に、$30 FPGA ボードという触れ込みの BeMicro Max10 Evaluation Kit のことを知ります。もうこの機会にやるしかないと思い、購入はしましたが、この評価ボードを知るきっかけになった、「$30 で始める FPGA」[*1]のスライドをひととおりやってみたのち、はたと、この次にどうすればいいのだろう？ と悩み、しばらく手つかずになってしまっていました。

そんな折、私も所属している Make 部という電子工作やモノづくりをしている社内サークルが、2016 年に引き続き 2017 年も Maker Faire Tokyo 2017 に出展することが決まりました。そこで、Maker Faire Tokyo 2017 での展示を目標に（締め切り駆動開発）、実際に自分で 0 から回路を HDL で書いて、FPGA をプログラムしてみようと考え、作ってみ

[*1] https://www.slideshare.net/yukitakatakemura/30fpga

た結果が本稿の内容です。

5.1 本稿での環境・範囲

使う FPGA ボードは BeMicro Max10 Evaluation Kit（以降、Max10 評価ボード）です。また、開発環境としては、Quartus Prime Version 16.1.0 Lite Edition を使います。Quartus の操作やどのように VerilogHDL をコンパイルして FPGA にアップロードするのかなどは、前述のスライドが詳しいので、そちらをご参照ください。本稿では、実現したい機能のための VerilogHDL の書き方に注目します。

5.2 何を作るのか？

今回は最終目標として暗号論的疑似乱数生成器として、Blum-Blum-Shub（以降、BBS）を実装してみます。アルゴリズムの詳細は Wikipedia[*2]などを参照してください。

実行する演算としては、次の漸化式をひたすら繰り返し、 x_n の下位ビットを出力するだけです。

$$x_{n+1} = {x_n}^2 \mod M$$

複数の算術演算を使い、組み合わせ回路、順序回路も必要なことから、ちょうどよい練習題材になるかと思い、挑戦してみることにしました。

5.3 まずは入出力

まずは簡単なロジックを作ってどう書いていけばいいのかを色々確かめていきます。Max10 評価ボードには 4 つのタクトスイッチと 8 つの LED が付いています。

▲図 5.1　BeMicro Max10 Evaluation Kit

[*2] https://ja.wikipedia.org/wiki/Blum-Blum-Shub

図 5.1 の写真で右端に LED とタクトスイッチが見えます。これを入出力として使えるようにしつつ、入力と出力の間に簡単なロジックを仕込んでみます。

一番最初はスイッチと LED を直結してみます。ボタンを押すと LED が点灯するという回路を作ってみます。筆者が前述のスライドにしたがって作ったプロジェクトでは BeMicro_MAX10_top.v という VerilogHDL ファイルに Max10 評価ボードの入出力ポートなどがすべて定義されていたので、おもむろにこのファイルに HDL を記述していきます。

確認したところ、タクトスイッチを表すポートは、PB[1]〜PB[4] として定義されています。また、LED については、USER_LED[1]〜USER_LED[8] として定義されています。そして、assign 構文で、入力と出力をつなぐことができます。そこで、リスト 5.1 のようなコードを一行記載して、コンパイルし、ボードにアップロードして確認してみます。

▼リスト 5.1　LChika.v

```
assign USER_LED[1] = PB[1];
```

SW1 のタクトスイッチを押すと D1 の LED が点灯することが確認できました。

でもちょっと確認してみましょう。リスト 5.2 の HDL を実行します。

▼リスト 5.2　Check.v

```
assign USER_LED[1] = 1;
assign USER_LED[2] = 0;
```

USER_LED[1] に対応する D1 は消灯で、USER_LED[2] に対応する D2 が点灯しています。

以上の確認から、Max10 評価ボードのスイッチと LED は、表 5.1 のような関係になっていると考えられます。

▼表 5.1　論理値と各デバイスの動作

	LED	タクトスイッチ
0	点灯	ON
1	消灯	OFF

筆者は FPGA による演算に興味があるので、次に 2 ビットの全加算器を作ってみます。タクトスイッチ 4 つを 2 つずつ入力として使い、2 ビット＋ 2 ビットの全加算器を作ってみます。

▼リスト 5.3　FullAdder1.v

```
assign USER_LED[1] = ~(~PB[1] ^ ~PB[3]);
assign USER_LED[2] = ~((~PB[2] ^ ~PB[4]) ^ (~PB[1] & ~PB[3]));
assign USER_LED[3] = ~((~PB[2] & ~PB[4]) |
                       ((~PB[2] ^ ~PB[4]) & (~PB[1] & ~PB[3])));
```

負論理と正論理の変換もあり冗長な書き方も多いですが、2 ビットの全加算器をビット演算だけで書き下すとこのような感じになります。さすがに、これではあんまりなので、wire を使ってみます。

▼リスト 5.4　FullAdder2.v

```
wire c0 = (~PB[1] & ~PB[3]);
wire c1 = (~PB[2] & ~PB[4]);
wire s0 = (~PB[1] ^ ~PB[3]);
wire s1 = (~PB[2] ^ ~PB[4]);

assign USER_LED[1] = ~s0;
assign USER_LED[2] = ~(s1 ^ c0);
assign USER_LED[3] = ~(c1 | (s1 & c0));
```

でも、実は足し算なら、こう書けばいいのですよね。

▼リスト 5.5　FullAdder3.v

```
assign USER_LED[3:1] = ~(~PB[2:1] + ~PB[4:3]);
```

複数の信号線をまとめるときには、[] カッコを使います。

5.4　BBS 疑似乱数生成器を作る

ここまで作ってきたのは、組み合わせ回路と言って、出力が現在の入力にだけしか依存しない論理回路でした。しかし、BBS 疑似乱数生成器を作るには順序回路という、出力が過去の入力にも依存するパターンの回路にする必要があります。早速 reg 指定子と always 構文を使って作ってみましょう。

▼リスト 5.6　BBSRandomGen1.v

```
reg [31:0] Xn;
reg [31:0] Xn_1;
reg [63:0] mul;
reg [63:0] res;

always@ (posedge SYS_CLK) begin
        mul[63:0] <= Xn[31:0] * Xn[31:0];
        res[63:0] <= mul[63:0] % 64'd4294957621;
        Xn[31:0] <= res[31:0];
end
```

```
// 計算した BBS 乱数の下位 7 ビットを LED に割り当て
assign USER_LED[7:1] = Xn[6:0];
```

しかし、LED はすべて点灯したままで、乱数が生成されているようには動いてくれません。LED の点滅による明るさの変化もないので、ロジックが期待したようには動いていないようです。これを動くようにしていきましょう。

5.5 人に優しいクロックの生成

まず現状のままだと、速すぎてちゃんと動いているかわかりません。オシロスコープなり、JTAG でデバッグなりあればいいのですが、取り急ぎ目デバッグで頑張るために考えていきます。本来、FPGA で演算をさせるメリットとして計算の速さがありますが、ここでは人に優しくいきましょう。数 Hz のクロックにして、人の目に見える速さで点滅するようにしてみます（リスト 5.7）。

▼リスト 5.7　slowclock.v

```
reg [31:0] counter;

wire SLWCLK;

always@ (posedge SYS_CLK) begin
        if(!PB[2])begin
                counter[31:0] <= 32'd0;
        end else begin
                counter[31:0] <= counter[31:0] + 32'd1;
        end
end

// カウンタの 23 ビット目をクロックとして取り出し、LED に接続する
assign SLWCLK = counter[22];
assign USER_LED[8] = SLWCLK;
```

これで点滅が人の目に確認できるくらい低周期のクロックを生成できました。ここで、always の中に if でタクトスイッチ 2 を押していたときにはクロックのカウンターが 0 になるように指定しています。これは、クロック生成回路のリセットをこのように行っているということです。論理回路では、すべての素子が並列に動作しているため、どのように初期状態を作るか、つまりリセットするかがとても重要になります。また、前節では、すべての素子が同時に動くという前提を考慮せず、ソフトウェアプログラムのように考えてしまったため、回路が期待する動作をしてくれませんでした。それを踏まえて、リセットの仕組みと状態遷移の機構を作ることでちゃんと期待する動作をする疑似乱数生成器を作成してみます。

5.6 目標達成の完成形

以上のことを踏まえ、実際に BBS 疑似乱数生成器を書いてみたものがリスト 5.8 です。細かい説明はコード内のコメントを参照してください。

▼リスト 5.8 BBSRandomGen2.v

```
reg [31:0] Xn;
reg [31:0] Xn_1;
reg [63:0] mul;
reg [63:0] res;
reg [1:0] status;
reg [31:0] counter;

wire SLWCLK;

// クロック生成部分　システムクロックから数 Hz のクロックを生成
always@ (posedge SYS_CLK) begin
        if(!PB[2])begin
                // リセット回路
                counter[31:0] <= 32'd0;
        end else begin
                // システムクロックの立ち上がりごとにカウンタをインクリメント
                counter[31:0] <= counter[31:0] + 32'd1;
        end
end
assign SLWCLK = counter[22];

// BBS 乱数計算部分のステートマシン 3 クロックで乱数を一つ計算
always@ (posedge SLWCLK) begin
        if(!PB[1])begin
                // リセット回路　PB[1] の押下でステータスと初期値をリセット
                status[1:0] <= 2'b00;
                Xn[31:0] <= 32'd9;
        end else begin
                // ステートによる条件分岐
                case(status[1:0])
                        2'b00:begin
                                // 二乗を計算してステートを更新
                                mul[63:0] <= Xn[31:0] * Xn[31:0];
                                status[1:0] <= 2'b01;
                        end
                        2'b01:begin
                                // M で mod を計算とステート更新
                                res[63:0] <= mul[63:0] % 64'd4294957621;
                                status[1:0] <= 2'b10;
                        end
                        2'b10:begin
                                // 下位 32 ビットを次の項に代入とステート更新
                                Xn[31:0] <= res[31:0];
                                status[1:0] <= 2'b00;
                        end
                        default:begin
                                status[1:0] <= 2'b00;
                        end
                endcase
        end
end
```

```
// BBS 乱数の下位 7 ビットを LED に配線、クロックも LED に配線
assign USER_LED[7:1] = Xn[6:0];
assign USER_LED[8] = SLWCLK;
```

5.7 まとめ

以上、初学者がいろいろなところに躓きながらも、ゼロからある働きをする論理回路をVerilogHDL で FPGA 上に実装する過程でした。

FPGA をベースとしたさまざまな応用は、今後のコンピューティングの世界で大きな役割を占めていくと思われます。今回は本当に初めの一歩ですが、今後も応用にチャレンジして有用なものを作り出したいと思います。

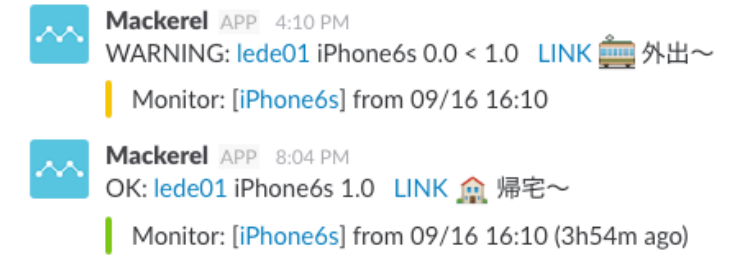
Suguru OHO / @ohomagic

第 6 章

家庭内ネットストーカーシステムを作った話

自宅の無線 LAN の通信状況をサーバ監視サービス Mackerel で監視してみたところ、予想以上に便利かつキモいシステムができました。

たとえば、家族の誰かが外出すると Slack に通知を飛ばすことができます。

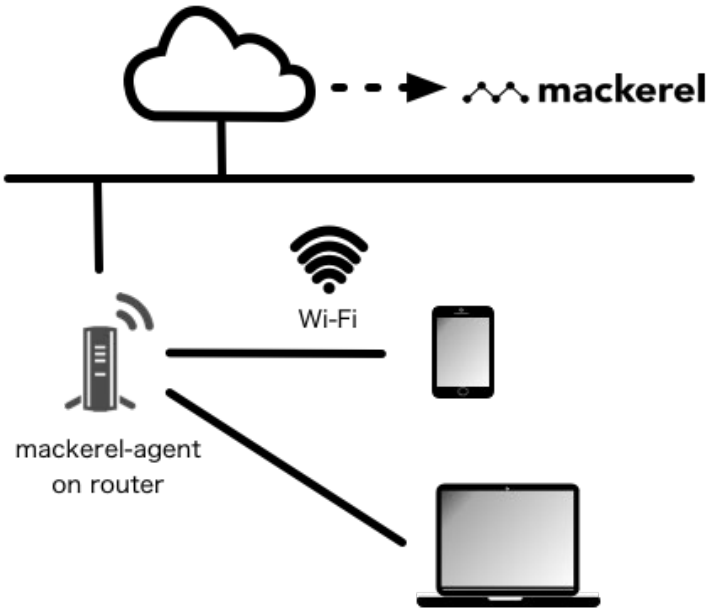


▲図 6.1　Slack 上に届く外出通知の例

同じことをしている人は多くないと思うので、その知見を紹介します。

6.1　システム概要

まずは筆者の自宅のネットワーク構成を紹介します（図 6.2）。



▲図 6.2　筆者の自宅のネットワーク構成

インターネットに接続しているブロードバンドルータがあり、無線でスマートフォンや PC がぶら下がっているような、ごく普通のネットワーク構成です。唯一変わっている点は、ブロードバンドルータ上で Mackerel エージェントが動いているということでしょう。

Mackerel とは

Mackerel は株式会社はてなが提供するサーバ監視サービスです[*1]。管理するサーバ台数が多い企業などを対象にした有料サービスですが、管理台数が少ない場合には無料で利用することもできます。

今回筆者が監視するサーバは自宅のブロードバンドルータ 1 台なので、無料プランで利用させてもらうことにしました。無料プランではログの保存期間が 1 日であるなどの制限はありますが、個人で利用する分には十分だと感じています。

後述するように、Mackerel エージェントのソースコードが GitHub 上で OSS 開発されている点や、プラグイン開発が容易である点もエンジニア目線ではありがたいと感じます。

また、LINE や Slack などアラートの通知先としてサポートされているサービスが充実しているのも便利な点です。監視対象が何らかのしきい値を超えた場合に管理者に通知を送るのが普通の使い方ではありますが、本稿の最初で紹介したように、外出と帰宅のよう

[*1] `https://mackerel.io/`

な個人的な情報の蓄積に使うこともできます。

6.2 ブロードバンドルータにカスタムファームウェアをインストールする

さて、読者の皆さんがご自宅で使っている無線 LAN ルータは何でしょうか？ 家電量販店で買った BUFFALO のルータだから Linux を動かせなんて言われても無理だよ、という方もいらっしゃるかもしれませんね。ところが、実は BUFFALO はじめ多くのメーカーの無線 LAN ルータの内部では Linux が動作しています。

ただし、メーカーが提供するファームウェア上の Linux は自由度が低く、任意のプログラムを動かすのには向きません。そこで、家庭用ブロードバンドルータのファームウェアごと書き換えて、自由度の高い Linux 環境を提供するプロジェクトがいくつか存在します。中でも一番知名度の高いプロジェクトが OpenWrt でしょう。

OpenWrt を導入すると家庭用のブロードバンドルータが DDNS・VPN・VLAN・QoS などに対応するようになります。ルータに SSH ログインできるようになるので、ネットワークトラブル時の原因切り分けにも便利です。

個人的に OpenWrt で一番便利だと感じる点は、常時起動の Linux サーバが家庭内に 1 個増える点です。家庭内の常時起動サーバを維持するのは案外手間で、重要な役割でもないかぎりいつの間にか退役していたりします[*2]。しかし、ブロードバンドルータの電源を落としっぱなしにするということはまずありませんから、長期運用が期待できる Linux サーバが家庭内に確保できるわけです。また、ブロードバンドルータは筐体や AC アダプタが家電っぽい雰囲気のものが多く、取り回しが楽なのも利点と言えるでしょう。

それだけでなく、OpenWrt では任意のソフトウェアをインストールすることもできます。今回はこれを利用して監視用エージェントの mackerel-agent を動かしてみたというわけです。

6.3 ブロードバンドルータの機種選定

ちなみに筆者の自宅では BUFFALO の有線 LAN ルータ BHR-4GRV のファームウェアを LEDE 17.01.3 に入れ替え、USB 無線 LAN 子機を接続してブロードバンドルータとして使っています[*3]。LEDE は 2016 年 5 月に母体である OpenWrt から fork したプロジェクトで、現状では OpenWrt の最新版と言えるようなディストリビューションです[*4]。

ちなみに、OpenWrt・LEDE がインストールできるブロードバンドルータの機種は市

[*2] 筆者の場合は Raspberry Pi 2 がそうでした

[*3] インストール手順など詳細は筆者による記事「BHR-4GRV に LEDE をインストールしたメモ」（`https://qiita.com/hnw/items/e1ec0cf5ca94be916433`）をご覧下さい

[*4] OpenWrt と LEDE は再合流が予定されています。LEDE という名前は 1 年後には存在しないかもしれません

場に出回っているもののうち、ごく一部です。動作しそうな機種をネットで調べてからブロードバンドルータを入手する必要があるのでご注意ください[*5]。

それは少し面倒だな、という方は、昨今なら Raspberry Pi 3 を無線 LAN ルータに仕立てるのが手っ取り早いかもしれません。

技適と OpenWrt・LEDE

Wi-Fi ルータで OpenWrt・LEDE を使おうという場合は電波法違反にならないよう注意が必要です。
というのも、ご存じの通り技適マークが付いていない無線機器を日本国内で使うのは違法ですが、技適を通っている無線機器のソフトウェア部分を改造した場合も電波法違反になる可能性があると考えられるためです。
そのため、ネット上で OpenWrt 情報を発信している日本人の方は、「電波暗室で実験した結果を報告しているだけで常用してるわけじゃないよ派」「日本に住んでいるとは誰も言っていないよ派」「有線 LAN ルータでしか遊んでないよ派」のいずれかの派閥に属している人が多い印象です。
本稿で紹介している有線 LAN ルータ +USB 無線 LAN 子機の構成であれば無線機器の改造に当たらないため技適の観点では合法だと考えていますが、もしかすると異なる観点もあるのかもしれません。

6.4 MIPS ルータ上で mackerel-agent を動かす

さて、ブロードバンドルータ上に LEDE をインストールして自由度の高い Linux 環境が構築できたわけですが、まだ問題があります。今回筆者が利用した BUFFALO のルータに搭載されている SoC[*6]は 32bit MIPS アーキテクチャです。実は、この環境で動く mackerel-agent のバイナリをビルドするのは一筋縄ではいきません。

OpenWrt・LEDE では SDK と呼ばれるクロスコンパイル環境を提供していますので、C/C++ プログラムであれば何の問題もなくバイナリをビルドすることができます[*7]。

しかし、mackerel-agent は Go で書かれています。本稿執筆時点の Go の最新バージョンである 1.9 は 32bit MIPS 対応が不完全で、浮動小数点数ユニット (FPU) 命令を含んだバイナリが作られてしまいます。しかし、ブロードバンドルータで採用されているよう

*5 「DD-WRT OpenWrt 適材適所で両方使いたい人向け @ ウィキ」や 2ch あらため 5ch の OpenWrt スレなどが参考になります
*6 System on a chip、ここでは CPU とほぼ同義
*7 通常ブロードバンドルータはメモリもストレージも少ないので、ターゲットマシン上でバイナリをビルドするのは非現実的です

な MIPS SoC は通常 FPU を搭載していません[*8]。FPU 命令を含まないバイナリを作るにはパッチを当てた Go 処理系でバイナリを自前でビルドする必要があるなど、ビルドするだけでも若干ハードルが高いのが現状です[*9]。

MIPS の softfloat 対応は Go 1.10 で導入予定ですので、半年もすればこの問題は解決しているはずですが、現状では筆者がビルドしたバイナリパッケージをインストールするのが一番楽だと思います。パッケージ配布ページ[*10]から最新版をダウンロードしてルータ上にコピーしてください。

```
# opkg update
# opkg install lede-17.01.3-ar71xx-mackerel-agent_0.46.0-4_mips_24kc.ipk
# uci set mackerel-agent.@mackerel-agent[0].apikey='APIKEY'
# uci commit
```

uci の仕組みを利用して API キーを登録しています。API キーは Mackerel 管理画面からコピー＆ペーストして下さい。

6.5 独自プラグイン mackerel-plugin-iw の導入

さて、ようやく本題の無線 LAN の監視について紹介します。せっかく mackerel-agent が MIPS ルータ上で動いたので無線 LAN の監視をしようとネットで検索してみたのですが、Mackerel で無線 LAN の利用状況を監視している事例は見つかりませんでした。

とはいえ、Mackerel にはプラグイン機構があるので、元データさえあれば監視対象を増やすのは簡単です。最近の Linux ディストリビューションであれば無線 LAN の利用状況は iw コマンドで確認できます。

```
# iw dev wlan1 station dump
Station 00:00:5e:f0:07:87 (on wlan1)
        inactive time:  1600 ms
        rx bytes:       6216368
        rx packets:     50296
        tx bytes:       4006367
        tx packets:     30159
        tx retries:     1159
        tx failed:      2
        rx drop misc:   17
        signal:         -50 [-49, -57] dBm
        signal avg:     -50 [-49, -56] dBm
        tx bitrate:     243.0 MBit/s MCS 14 40MHz
        rx bitrate:     216.0 MBit/s MCS 13 40MHz
        expected throughput:    53.557Mbps
```

[*8] 詳細は筆者によるの記事「Go 1.8 の MIPS32 バイナリをルータ上で動かしてみた（成功編）」（https://qiita.com/hnw/items/8703a0e776f8b6a60b45）をご覧下さい

[*9] 詳細は筆者による記事「Go の MIPS32 soft-float 対応が来たぞー（1.9 に入るとは言っていない）」（https://qiita.com/hnw/items/1cf89069b0d3e4118831）をご覧下さい

[*10] https://github.com/hnw/openwrt-packages-mackerel-agent/releases

```
        authorized:     yes
        authenticated:  yes
        associated:     yes
        preamble:       long
        WMM/WME:        yes
        MFP:            no
        TDLS peer:      no
        DTIM period:    2
        beacon interval:100
        short slot time:yes
        connected time: 195122 seconds
```

iw コマンドでは現時点でアクティブな Wi-Fi クライアントごとのネットワーク情報が取得できます。同じネットワークに 5 台のクライアントがぶら下がっている状態であれば同様の内容が 5 台分それぞれ表示されます。

今回、このうち「rx bytes」「tx bytes」「signal」の情報を蓄積するような Go スクリプト mackerel-plugin-iw を作成してみました[*11]。

インストールは mackerel-agent と同様に筆者の作成したバイナリパッケージを利用するのが良いでしょう。配布ページ[*12]から最新版パッケージがダウンロードできます。

```
# opkg install lede-17.01.3-ar71xx-mackerel-plugin-iw_0.1.0-1_mips_24kc.ipk
# /etc/init.d/mackerel-agent restart
```

6.6 mackerel-plugin-iw が表示する情報

iw コマンドの出力をグラフ化してみると多くの気づきがありました。改めて感じたのは、我々は多くの情報をまき散らしながら生きているのだなあ、ということです。個別のグラフから読み取れる情報を紹介していきます。

Wi-Fi セッションの生死がわかる

iw コマンドでは接続中の端末の MAC アドレスごとの通信状況が表示されます。つまり、MAC アドレスごとのエントリの有無を 1 分に 1 回記録することで、どの端末が現在ネットワークに参加しているかがわかります。mackerel-plugin-iw ではこれを記録・グラフ化してみました。

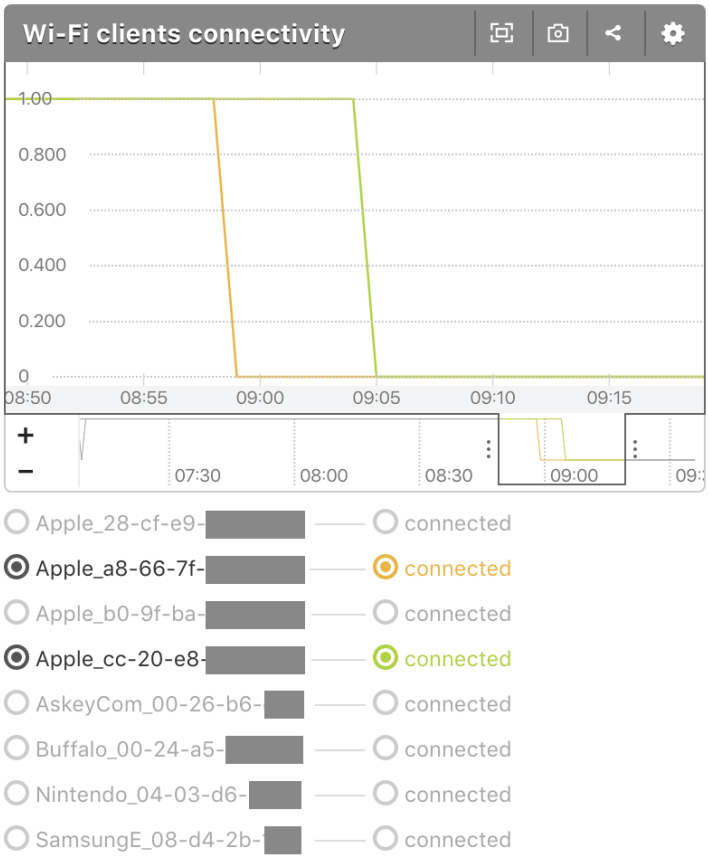
これにより、自宅にいるときにいつ PC を開いていたかがわかるようになりました。PC を開いている時間は常に Wi-Fi セッションが維持されるので、セッションが切断されたということは PC をスリープさせたということになります。

また、特定のスマートフォンを持った人が外出したかどうかの判断にも利用できます。というのも、iOS 端末は画面オフ時も Wi-Fi セッションが切れず、家の中にいる場合は常

[*11] https://github.com/hnw/mackerel-plugin-iw/
[*12] https://github.com/hnw/openwrt-packages-mackerel-plugin-iw/releases/

に Wi-Fi セッションが維持されるのです[*13]。筆者の自宅のルータでは 300 秒無通信状態になるとセッションが無効になるため、端末を持って外出した場合のみグラフ上の値が 0 になります。

図 6.3 は筆者が MacBook と iPhone を持って外出したときのグラフです。両者の Wi-Fi セッションが同時間帯に切れていることから、9 時頃に家を出たことがわかります。



▲図 6.3　iOS 端末と MacBook を持って外出したときの例

ちなみに、mackerel-plugin-iw では利便性向上のために MAC アドレスの前 3 桁からわかるベンダー名をラベルの先頭に追加しています。これによりどの端末の情報か判断しやすいですし[*14]、万一見慣れない端末がネットワークに参加していればすぐに気づくことができます。

[*13] Android 端末も iOS 同様に常時 Wi-Fi セッションを維持する機種が多いかと思いますが、一部機種では画面オフ時に Wi-Fi セッションを切る設定があるようです

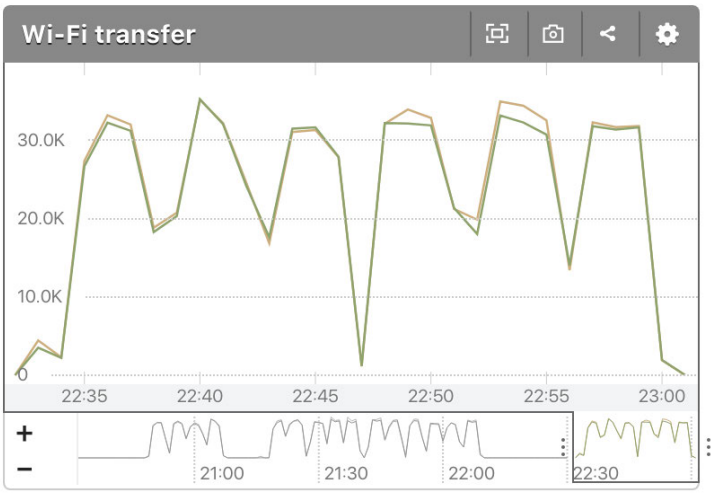
[*14] 図 6.3 の通り筆者の自宅には Apple 端末が多いためそれらの区別は困難ですが、たとえばベンダー名が「Nintendo」となっていればゲーム機であることは一目瞭然です

端末ごとのネットワーク転送量がわかる

iw コマンドでは送受信した累積バイト数がわかるので、これを 1 分ごとに記録することでネットワーク転送量をグラフ化することができます。これにより、各端末でどんな操作をしているか想像することができます。

たとえば YouTube で動画を見ている場合には転送量のグラフは横ばいになりますが、Web ページを閲覧している場合には凸凹が多いグラフになります。

他にもアプリケーションによっては特徴的なグラフになることがあります。図 6.4 のグラフは筆者が Nintendo Switch のスプラトゥーン 2 でナワバリバトルを 6 回遊んだときのものです。ナワバリバトルは同じ部屋にプレイヤーが 8 人集まるのを待って、3 分間の通信対戦を行うのが 1 セットなので、3 分間一定量の通信を行って 1 分ほど通信が落ち着くという繰り返しになっているのがわかります。



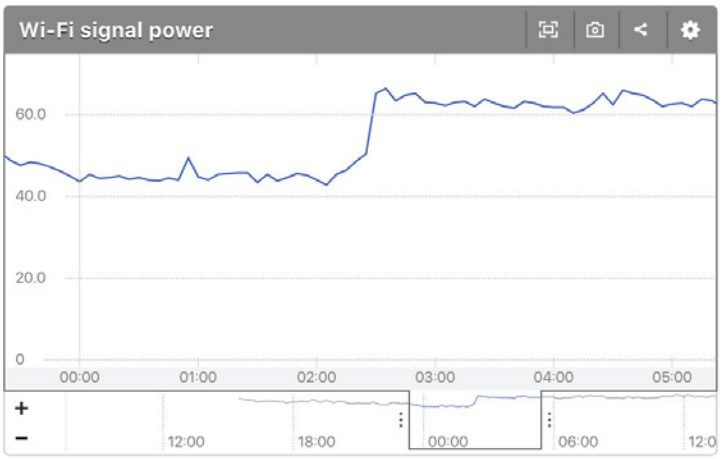
▲図 6.4　ナワバリバトル中のネットワーク転送量

電波強度から端末と親機の距離が推定できる

無線 LAN の電波強度は-30dB から-100dB といった値で表されます。この数字は小さくなればなるほど電波強度が弱く、親機と子機の距離が離れている可能性が高いという意味になります。この値も iw コマンドで取得できます。

電波は人体やその他でも遮蔽されるので正確な位置検出ができるわけではありませんが、Wi-Fi ルータと同じ部屋にいるか、それとも別の部屋にいるか、くらいは判断ができそうです。

図 6.5 のグラフは筆者が Wi-Fi 親機のある部屋でスマートフォンを操作していて、電池残量が減ってきたので隣室で USB 充電をはじめたときのものです。両者は直線距離ではそれほど変わらないのですが、親機と隣室の間に壁を挟んでいるせいか約-40dB[*15]から約-60dB と電波強度が大きく減衰していることがわかります。



▲図 6.5　Wi-Fi 親機のある部屋から隣室に移動した例

6.7　本システムのキモさ

ここまでの話だけであれば便利で面白いシステムができましたね、で済むかもしれません。しかし、ここから発展して少し怖い話があります。

家にいなくても家族の行動がわかってしまう

mackerel-plugin-iw で取得できる情報は、同じ無線 LAN にぶら下がっている全端末についてグラフ化されます。自分の端末の詳細な情報を自分が把握できるだけであれば何の問題もないのですが、同じ無線 LAN ネットワークを利用している家族の情報も取れてしまいます。家族とはいえ、自分の行動が筒抜けになるのは監視されているようで気持ちが良いものではありません。

また、本稿の最初で例示したように、家族の誰かが外出するたびに管理者に通知が飛ぶような設定もできるわけです。そこまでやると家族間とはいえストーカー呼ばわりされるかもしれません。

しかし、逆に言えば我々はそれだけ多くの情報をまき散らしながら生活しているという

[*15] 現在の Mackerel の仕様ではマイナスの値をグラフ化できないため正負反転して記録しています

ことの裏返しとも言えるでしょう。これらの情報が家族内でさえ知られて困るということなら、スマートフォンを使うこと自体がリスクだと言えるかもしれません。

無関係な外部の人間に行動を把握されるリスク

家族間でさえ知られると少し嫌な気持ちになるレベルの情報を我々はまき散らしながら生活している、という話を紹介してきましたが、これが無関係な外部の人間に知られるとしたらどうでしょうか。

実は無線 LAN には「モニターモード」というモードがあり、認証されていないアクセスポイントの通信も受け取ることができます。WPA2 などで暗号化されていれば通信の本文を解読することはできませんが、暗号化されていても転送量の大小は把握することができます。また MAC アドレスは暗号化の対象外なので、同一ネットワークに参加していなくても各 Wi-Fi 端末の識別や端末ごとの Wi-Fi のセッション生死の推定、さらにはベンダー名の把握まで可能です。

つまり、物理的に十分近い場所にいる人間であれば、赤の他人であっても本稿で紹介したようなグラフを作成することが可能なのです。ただし、電波強度だけは家の外からだと距離が遠かったり遮蔽物が多かったりして大した精度は出せないかもしれません。その点は少しだけ安心と言えるかもしれません。

何にせよ防犯的な観点で非常に怖い話で、考えただけでゾッとしますね。また、対策らしい対策がないのも恐ろしい点だと言えるでしょう。防犯用にダミーの無線通信を行うアプリケーションやガジェットを作るというアプローチも考えられますが、どれほど意味があるかは疑問です。

6.8 まとめ

OpenWrt と Mackerel を利用して少々キモい家庭内監視システムが作成できることを紹介しました。

改めて無線って怖いですね。現代の恐怖と言えばそこまでですが、まずは我々自身がそのリスクを知ることが大切ではないでしょうか。

Yoshio HANAWA / @hnw

第 7 章

とある KLab のスマホアプリのビルド事情

本章は、Unity を使ったスマホアプリ開発プロジェクトにおいてバイナリビルド環境・フローの改善に取り組む方々に向けたものです。筆者が KLab で約 2 年半のうちに取り組んできたいくつかのプロジェクトのビルド環境とフローについて紹介し、これらの事例を通じてどのアプローチで何が得られたか？ どのような状況が見えるようになったか？ についてまとめます。各ツールの詳細な機能や Tips はあまり紹介せず、ツールを利用する上で意識した点や、考え方を中心に紹介します。読者の皆様の取り組みに役立てていただけると幸いです。

それでは、早速見ていきましょう。

7.1 スマホアプリ開発、運営の現状

ビルドの失敗、遅延がもたらす悲劇

スマホアプリに限らず、アプリビルドの失敗、遅延によって起きる悲劇はさまざまです。たとえば、ビルドが遅延してリリースに間に合わなかった時の次のような状況を考えてみます。

ビルド担当者

「ビルドが突然うまくいかなくなってしまいました。このままでは予定どおりリリースできません。原因は調査中ですが、解決目処は立っていません」

プロジェクトマネージャー

「なんだって。今回のリリースには目玉の機能が入っていて、ユーザーにもすでに告知している。連動プロモーションにも費用をかけているし今日は絶対 App Store への申請をしたかったんだけど。でも目処が立ってないなら仕方ない。まずは延期を告知しよう」

企画担当者

「リリース延期の告知をしました。Twitter でのツイートも実施しました」

お客様

「リリースが延期だって。楽しみにしていたゲーム、イベントが遊べないのは辛い」

すべてはビルドに掛かっている

突っ込みどころはありますが、ビルドがうまくいかないことにはスマホゲームのサービス運営は成り立ちません。リリースが遅れるとゲームのイベントがずれる、費用をかけたプロモーションが中止になるといった影響を生む可能性があり、結果としてゲームを毎日遊んでくれているお客様の信頼を失ってしまうことにもつながりかねません。このため、ビルドの失敗や遅延はスマホアプリ開発・運営の天敵です。

7.2 一般的なアプリのビルドでよくある問題

安定的なビルドの重要性がわかったところで、ビルドにまつわる問題を具体的に見ていきましょう。

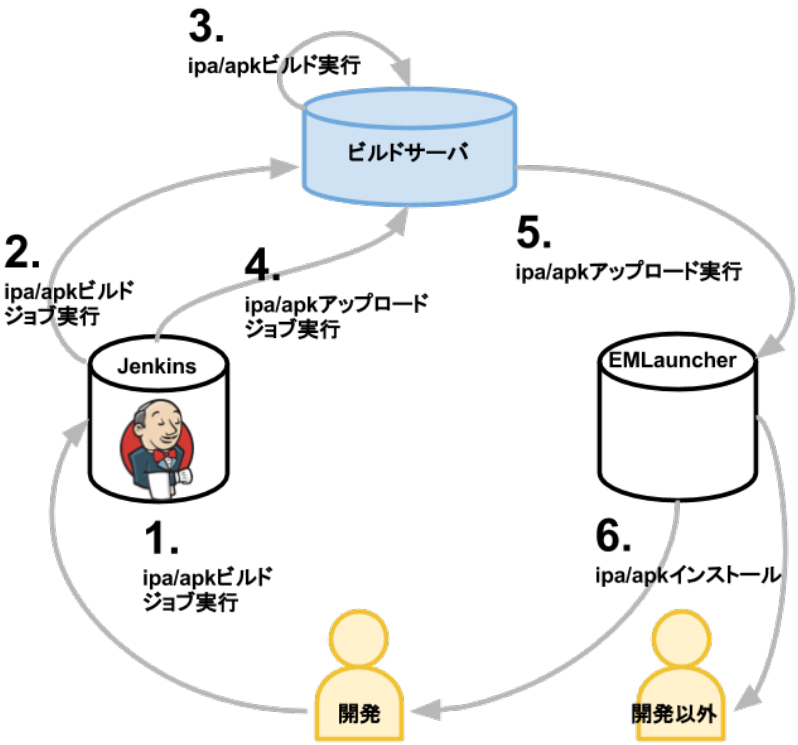
- ビルドがよく失敗する
  - ビルドエラーが起き、ひとつエラーを直してもまた別のエラーが出てしまう
- ビルド失敗時の原因把握に時間がかかる
  - ビルドエラーが起きた時のログが出力されていない
  - ビルドエラーが起きた時のログが多すぎて分析しにくい
- ビルドに時間がかかる
  - ビルドマシンのスペックが足りない
  - 必要のないビルドが何度もリクエストされている
- ビルドで問題が起きていることに気づかずその後の工程を進めてしまう
  - ビルドパラメータの確認がビルド開始時以外にできない、確認方法がない
- リリースビルドに切り替えた途端問題が発生する
  - アプリが起動しない、正常に動作しない
  - そもそもビルドが通らずバイナリが作れない
- ビルドできる人が限られている
  - ビルドパラメータの意味や、ビルドの仕組みを理解している人が少ない
  - ビルドの権限を持てる人がツールの制限などによって限られている

このようなビルドにまつわる問題は、プロジェクトの規模が大きくなるほど複数の問題を抱えやすく、結果として具体的な問題がどこにあるのかわかりづらくなりがちです。

ここで挙げた「よくある課題」に対して KLab ではどのように取り組んでいるのか？ここからは筆者が関わってきたいくつかのプロジェクトについて紹介していきます。

7.3 KLab のプロジェクト共通環境

各プロジェクト固有の話の前に、社内の多くのプロジェクトが共通で利用しているツールとフローについて紹介します。図 7.1 は KLab におけるゲーム開発プロジェクトの一般的なビルドフローを示したものです。



▲図 7.1　KLab のプロジェクトの一般的なビルドフロー

- プロジェクトのバージョン管理
  - メインは GitHub Enterprise（以下 GHE）[*1]を利用。ゲーム中で利用する画像や音声といったアセットは、別途 Subversion[*2]を利用することが多い

[*1] GitHub Enterprise `https://enterprise.github.com/home`
[*2] Subversion `https://subversion.apache.org`

- CI 環境
  - ほとんどのプロジェクトで Jenkins[*3]を利用。社内向けアプリ配布には内製サービスの EMLauncher[*4]を利用

小規模な検証プロジェクト以外では通常 Jenkins と EMLauncher を利用しているところが特徴といえます[*5]。

CI については、CircleCI のように iOS アプリのビルドに対応できるツールを使っているところもあるかもしれませんが KLab では検証、導入がまだこれからというところもあり、Jenkins を使っています。また、ビルドに使用するマシンはプロジェクトごとに社内に置いているのですが、この背景には iOS や Android のビルドに必要な証明書などを外部に出したくないということがあります。

EMLauncher と Apple Developer Enterprise Program の関係

KLab には、EMLauncher を利用して社内向けのアプリ配布、インストール、試遊を行う文化がありますが、この文化を下支えしているのが Apple Developer Enterprise Program です。

通常、iOS アプリの開発や App Store 上でのパブリッシングを行うためには Apple 社の Apple Developer Program に参加する必要がありますが、通常の Developer Program の注意点として、開発中のアプリをインストールできる端末台数に制限があります。[a]

Enterprise Program は、主に企業での利用を想定しており、契約した組織内の端末に限り端末数の制限なく開発中のアプリをインストール、テストできるメリットがあります。Enterprise Program には年 40,000 円近くの費用が掛かりますが[b]、バイナリや開発端末の管理が簡単になるところも魅力的ですので、意外と個人デベロッパーにもメリットがあるかもしれません。

[a] iPhone, iPad などの機種ごとに 100 台まで

[b] Apple Developer Enterprise Program 公式ページ `https://developer.apple.com/programs/enterprise/jp/`

[*3] Jenkins `https://jenkins.io`

[*4] EMLauncher `https://github.com/KLab/emlauncher`

[*5] 実際に私自身よく Jenkins や EMLauncher を利用していますが、あまりにも日常的に使っているため、ある日突然使用できなくなったとしたら正直頭を抱えてしまうと思います……

7.4 ビルドに関わってきたプロジェクトの紹介

次に、筆者が関わってきたプロジェクトを紹介します。

- パズルゲーム A
  - Unity で新規開発、リリースし、運営を続けていたプロジェクト。筆者は新規開発期に参加
  - プロジェクト最大人数は 40 名以上。ビルド関連の作業メンバーは 1, 2 名
- タイミングアクションゲーム B
  - Unity で新規開発、リリースし、運営を続けていたプロジェクト。筆者は新規開発期に参加
  - プロジェクト最大人数は 40 名以上。ビルド関連の作業メンバーは 1, 2 名
- アクションバトルゲーム C
  - Unity で新規開発、リリースし、運営を続けていたプロジェクト。筆者は運営期に参加
  - プロジェクト最大人数は 40 名以上。ビルド関連の作業メンバーは 1, 2 名
- 検証プロジェクト D
  - iOS, Android の新機能検証プロジェクト
  - プロジェクト最大人数は 2 名程度。ビルド関連の作業メンバーは 1 名

7.5 各プロジェクトのビルドフロー振り返り

要点ごとに、ビルドフローを見ていきます。具体例として、各プロジェクトではどうだったか？ その結果どういうメリットがあったのかを示していきます。

ビルド用のマシン

ビルドマシンは、プロジェクトごとに Mac Pro[*6]を用意して利用する場合が多いです。

パズルゲーム A では専用のビルドサーバーを用意せず、他のプロジェクトと共通のビルドサーバーを間借りしていました。このためか、普段は 10〜15 分程度で終わるビルドに 1 時間ほどかかることがありました。このような場合は、プロジェクトチームに対して都度説明が必要になっていたと記憶しています[*7]。

[*6] iOS は Xcode でのビルドが必須ですので、マシンのスペックを上げつつ、台数を最小限にする（＝費用を抑える）ためのマシンの選択肢として、自ずと Mac Pro に落ち着いた、と聞いています

[*7] 「他のプロジェクトのビルドが詰まっているみたいなので、テストプレイの MTG はリスケさせてください」のような

このようなことがあってからか、KLab の各プロジェクトでは今の所、ある程度の規模のプロジェクトになったら専用のビルドマシンを用意したほうが得策ということで、メインを専用ビルドマシン、サブとして共通ビルドマシンを利用する、といった形をとっています。

昨今はマシンの性能も上がっていますので、共通のビルドサーバーを利用することで一定ビルドにかかるハード的なコストを抑えることを期待できますが、プロジェクトに関わる人数が増えるほどひとつのプロジェクトのビルド頻度は上がります。また、iOS では異なるバージョンの Xcode でのビルドを同時並行して実行できないという現象も確認されていますので、ビルドに使用している Xcode のバージョンが異なるプロジェクト同士でビルドマシンを相乗りすることはあまり得策ではありません。

これは複数の Unity, Xcode をインストールする必要がある検証プロジェクトでも同様です。Unity, Xcode は共に 1〜10 数 GB 程度の空き容量を必要とするため、ビルドマシンの容量が逼迫することも多くありました。このため、小規模プロジェクトでも、検証プロジェクトであれば、独立したビルドマシンを用意することを検討する余地があります。

ビルドのジョブ

ジョブについては、多くのプロジェクトでターゲット OS（iOS/Android）ごとに分けたビルドジョブと、完成したバイナリを EMLauncher にアップロードするジョブを用意して、それらを順番に呼び出す上流のジョブを用意していました。

ひとつ特徴的だったのは、パズルゲーム A でリリース用のジョブを用意していたことです。これはボタンひとつでリリース用のビルドをできるもので、その甲斐あってか 10 回以上のリリースビルドでオペミスなしという実績を作っています[*8]。

パラメータビルドのオペレーションミスを減らす目的で使っていたものとしては、Jenkins の「Rebuild」機能もあります。これは一度実行したジョブと同じパラメータをコピー、再編集してビルドできるものですが、単純にビルドをやり直す以外にも意外と使えます。

たとえば iOS での申請時に、「署名だけ異なるビルドを作成する」場合です。Android アプリの場合は本番サーバー環境へアクセスするリリースバイナリをリリース前にテストできますが、iOS アプリの場合は iOS 申請に出せるバイナリを申請前にテストすることはできません。このため、署名のみを差し替えたビルドを作る需要が存在します。

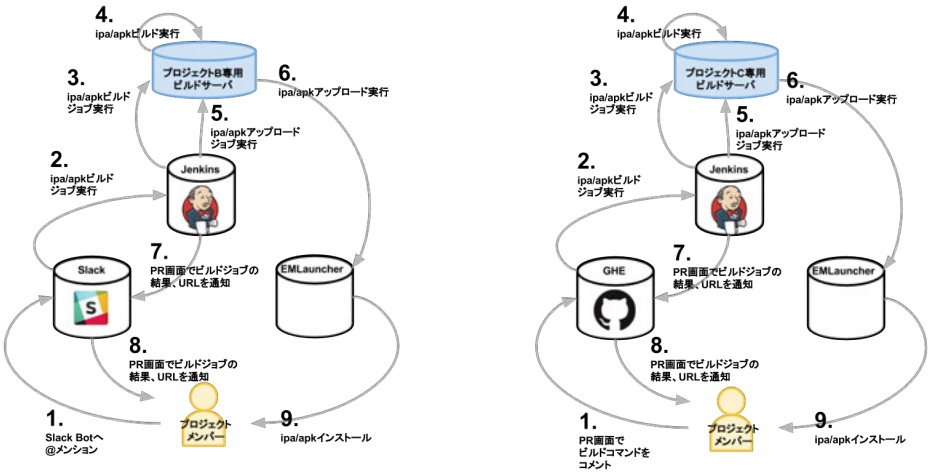
自動ビルドとはいえど、パラメータビルドになると一定人の手が入ってきますので、ビルドの仕組みとワークフローの作り込みによって単純なオペミスを未然に防ぐことも必要です。

[*8] 前提としてどのバージョンのバイナリでも同じビルドパラメータで作れるようなブランチ運用などのルール決め、ワークフローを整備する必要はありました

ビルドのトリガー

パズルゲーム A ではほとんどのメンバーが Jenkins を直接叩いていて、ビルドパラメータでバイナリを EMLauncher にアップロードするか否かを指定できるようになっていました。

特徴的なのはタイミングアクションゲーム B・アクションバトルゲーム C で、図 7.2 のように Slack, GHE 経由で Jenkins のジョブをフックするようにしていました。



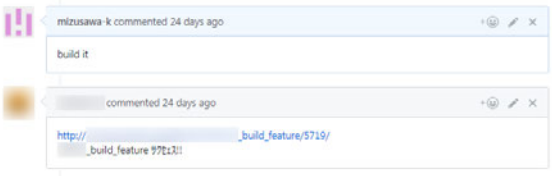
▲図 7.2　Slack, GHE と連携したプロジェクトのビルドフロー

タイミングアクションゲーム B では、Slack で特定のアカウントに向けてビルドコマンドを投げるとビルドが走るようになっており、Slack のチャンネルにいる人なら誰でもビルドできるようになっていました（図 7.3）。これによって開発職以外の人でもビルドが可能となり、パラメータ・文言を調整する企画職や、UI・演出の細かな調整を実機で確認したいアート職の人が自分の好きなタイミングで必要なバイナリを作り確認できるようになりました。

この方法は便利な反面、Slack でのビルドリクエストでは、パラメータが正しいフォーマットで送られていないことに気づかないケースも度々ありました。これは実際の運用状況を見ながらエラーチェックを充実させたり、分かりやすいエラーメッセージを Slack のメンションで返すようにすることで改善できそうです。

▲図 7.3　タイミングアクションゲーム B での Slack 経由でビルドした様子

アクションバトルゲーム C は GHE の PR 画面上で特定の文言をコメントするとビルドが走るようになっており（図 7.4)、こまかなパラメータの指定なしで簡単に実機確認用のビルドを作成できました[*9]。



▲図 7.4　アクションバトルゲーム C での GHE 経由でビルドした様子

[*9] GHE の PR 画面でビルドをリクエストする場合、ビルドのキューがたまりがちになる場合もままありました。必要以上にビルドリクエストが飛ばないように PR ルールを整備しておいたり、バイナリのターゲット OS が iOS, Android どちらでも問題ない場合はビルドジョブのキューの状況を見て、ジョブが空いている方を選択してビルドする仕組みを入れてみるのも一手だと思います

GitHub 上での PR 運用

Git を利用したチームでのビルドフローを考える上で、意外と重要で間接的に関わってくるものが PR の運用です。

たとえばタイミングアクションゲーム B では、PR で追加のコミットがプッシュされる度にそのブランチでのコンパイルが通るか否かを自動テストするようにしていて、さらにテストが通らなければ PR をマージできないように GitHub 上で設定していました。このため、PR マージ後のリリースブランチでのビルドエラーは比較的抑えられていました[*10][*11]。

また、自動テストの有無以外にも、PR 時点で実装者がどこまでテストするのか？ Unity エディタ上で確認するのか実機で確認するのか？ によっても、ビルドマシンにかかる負担が変わってきます。

実機確認を必須としたアクションバトルゲーム C では、PR 毎にビルドが実行されるためビルドマシンにかかる負担が多く、PR 毎の確認工数も掛かっていましたが、品質担保には一定繋がっていた印象です。

また、PR 時点では基本的に Unity エディタ上での確認のみをおこなっていた[*12]タイミングアクションゲーム B では、実機で確認すると予期せぬ現象が起きていた場合もありましたが、各実装者が持てるタスクの数はアクションバトルゲーム C と比較して多かった印象でした。

このあたりはコストと品質のトレードオフであり、どちらが良いというわけではありませんが、どのタイミングでどこまでチェックしておくか、PR がどのタイミングまでにマージされているべきか、などをチーム内できちんと事前共有しておくことは重要です。

接続先サーバーの設定

アプリ開発では接続先サーバーが開発用・ステージング用・本番用と複数に分かれているケースが多いです。

今回挙げた各プロジェクトでは、接続先サーバーの設定をソースコードなどのテキストファイルにあらかじめ定義して利用するか、Jenkins のパラメータで都度指定するかのい

[*10] 実際にはリリースビルドでなぜかいきなりビルドエラーが起きて iOS 申請前にバタバタした……という話もまれにあったのですが、逆にこのテストを通さずに PR のマージを進めていたらどれだけチームがビルドエラーに悩まされたのか……と思います。ビルド時点で出てくる問題を可能な限り「少なく」「小さく」するという意味で、PR マージ前にチェックできるに越したことはなさそうです

[*11] Unity でのシンボルの定義はプラットフォームごとの PlayerSettings や、rap などの定義ファイルで編集でき、Jenkins などでビルドする際にパラメータ変更することでビルドごとに必要な定義をできるようにしていますが、その使い分けがチーム内で周知されていないと思わぬところで足元をすくわれることがあったりします。本番アプリにデバッグ機能が入ったままリリースされてしまった、ということがないように:(

[*12] 流石にタイミングアクションパートや、課金機能などのネイティブ機能の確認は PR マージ前に、実装作業者にておこなっていました:)

ずれかの方法が取られていました。

テキストファイルにあらかじめ定義しておく方法のメリットはアドレスのコピペミスなどが起きにくいというところですが、その反面頻繁に接続先が変わったり作業者ごとに接続先が異なる場合には使いにくいです[*13]。

ふたつの方法を組み合わせて利用することもできますので、Jenkins や Slack などでビルド呼び出し時に指定するパラメータで接続先を指定できるようにしておくと何かと便利という印象です。

ちなみに、Jenkins を使用しない検証プロジェクト D のような場合は、特にパラメータを用意する必要はなく、ビルドの都度コードを書き換えるくらいでも十分な場合が多かった印象です。

開発用、リリース用バイナリの切り替え

一般的な iOS, Android アプリで開発用ビルドとリリース用ビルドは署名周りが異なります。

iOS ならプロビジョニングプロファイルとコード署名、Android ならキーストアのパスワードとパス類は各プロジェクトが個別のパラメータで設定するようにしていました。

一部のプロジェクトでは必要なパラメータをあらかじめスクリプトなどでまとめておいて、一括で設定できるようにする対応も進んでいるようですが、私が担当していた頃はそこまで手が回りませんでした。

急ぎでビルドが必要な場合にちょっとしたタイプミスなどでビルドを何回かやり直す羽目になると無用に神経をすり減らすので、署名周りの設定も事前にまとめておけるとより安心です。

ビルドに関するノウハウ共有

アプリの変化と共にビルドの手順も変化します。ビルド手順や詳細について細かくドキュメントを整備するとメンテナンスコストが大きいため、簡単な概要を wiki などにまとめておいて詳細は口頭あるいは Slack で確認するというケースがほとんどでした。

用途ごとに必要なジョブを構築したり、数多くのパラメータの意味をひとつひとつ伝授するコストがちょっと大きいと感じられる場合にも、ざっくりと用途ごとにパラメータをまとめておく程度でも大分安心感が違うようです。なお、ノウハウ共有という意味では、Slack や GHE などのコミュニケーションを取る場所でビルドのコマンドをポストするというのはエンジニア以外の方が自分が使うコマンドとの差分を比較する機会ができるということにもなりえますので、単にビルドができる以上のメリットがあります。

[*13] 接続先を変える度に接続先の定義を書き換えてコミットしなければならない場合、Git を使い慣れているメンバーのサポートも必要になりますし、Git に慣れているメンバーであっても正直手間だと感じられると思います

ビルド時間の短縮

マシンのスペックを上げる他にビルド時間を短縮する方法はないのか？ 気になった方もいらっしゃるかと思います。
今回は差分ビルドのような話題には触れませんでしたが、そうしたビルドそのものの見直し以外に、比較的簡単にビルド時間の短縮ができる方法として iOS 限定ではありますが[a]、Xcode で使用する CPU コア数を増やすというものがあります。
具体的な方法についてはここでは紹介しませんが、Web 記事でも取り上げられている方がいますし、Apple 公式でも触れられていますので、一度試してみても良いと思います。

[a] Unity 上で iOS プロジェクトをビルドする際も、IL2CPP ビルドには結構時間がかかるので同じような設定があると嬉しいのですが……

7.6 おわりに

筆者は、KLab 入社前は受託開発の会社で働いていましたが、納品をソースコードでおこなうことがほとんどだったり、アプリのバイナリビルドは完全に同僚にお任せしていました。

後述する参考資料に載せている書籍は、入社前にクライアントエンジニアなら読んでおかなくてはと思い購入したものでしたが、当時は正直、「何となく分かる気はするけど実感としてぴんと来ない」という内容でした。

しかし、KLab に入って複数のプロジェクトでビルドの経験を積んだ今となっては、「このケーススタディは身に覚えあるな」「うちではあまり見たことないけど、それはうちでこういうことができてるからかな。たまたまかもしれないけど」というように、実体験とセットで読み進められるようになっていることに気付きました。また、種々の課題に対しておおよそどのようなアプローチをできそうか見当をつけるられるようになったことも大きな成長だと思います。

KLab に入って深めた知識、経験に基づいてまとめた本章の内容が、読者の皆様のビルドの助けになれば幸いです。

Kinuko MIZUSAWA

7.7 参考資料

書籍

- 継続的デリバリー 信頼できるソフトウェアリリースのためのビルド・テスト・デプロイメントの自動化／アスキーメディアワークス
- 継続的インテグレーション入門／日経 BP
- Jenkins 実践入門 ——ビルド・テスト・デプロイを自動化する技術（WEB+DB PRESS plus）／技術評論社
- チーム開発実践入門 ——共同作業を円滑に行うツール・メソッド（WEB+DB PRESS plus）／技術評論社
- GitHub 実践入門 —— Pull Request による開発の変革（WEB+DB PRESS plus）／技術評論社

CEDEC 2015

- プロダクション分野のプラクティスについて CI や自動テストなどを中心に語るラウンドテーブル（https://cedil.cesa.or.jp/cedil_sessions/view/1307）
- チーム開発をスムーズにするために何をすべきか（https://cedil.cesa.or.jp/cedil_sessions/view/1385）
- 長期運営タイトルに後からパイプラインの自動化を導入した際の技術的 Tips（https://cedil.cesa.or.jp/cedil_sessions/view/1306）

CEDEC 2016

- ビルドエンジニアを「専任」でやってみた 〜「動けばいいや」は許しません！ 〜（https://cedil.cesa.or.jp/cedil_sessions/view/1444）
- チームのビルドパイプライン、錆びていませんか？ 〜Jenkins は定期的に組み換え、掃除しよう〜（https://cedil.cesa.or.jp/cedil_sessions/view/1477）

KLabGames TechBlog

- 新規アプリ開発プロジェクトでの制作物納品フロー（http://klabgames.tech.blog.jp.klab.com/archives/1063264564.html）

著者

第 1 章・表紙絵 Sho HOSODA / @gam0022

感情を表現する練習のために、ホラーゲーム実況をはじめました。

第 2 章・編集・表紙デザイン 岡本和樹 / @kakkun61

まんまとレンズ沼にはまりました。バズーカ。

第 3 章 やまだ

最近欲しいものは大きなパソコン机。

第 4 章 Daisuke MAKIUCHI / @makki_d

眼鏡っ娘が好きです。

第 5 章 Suguru OHO / @ohomagic

Google Home を買ったら 4 歳の息子が面白がって Google さんに色々聞いています。これがデジタルネイティブかと思いました。

第 6 章 Yoshio HANAWA / @hnw

そろそろスプラローラーを卒業したいと思いつつ、他のブキが全然うまくなりません。

第 7 章 Kinuko MIZUSAWA

ビルドを待ちながら Slack のログを眺めたり、音ゲーを嗜んだりするのが好きです。

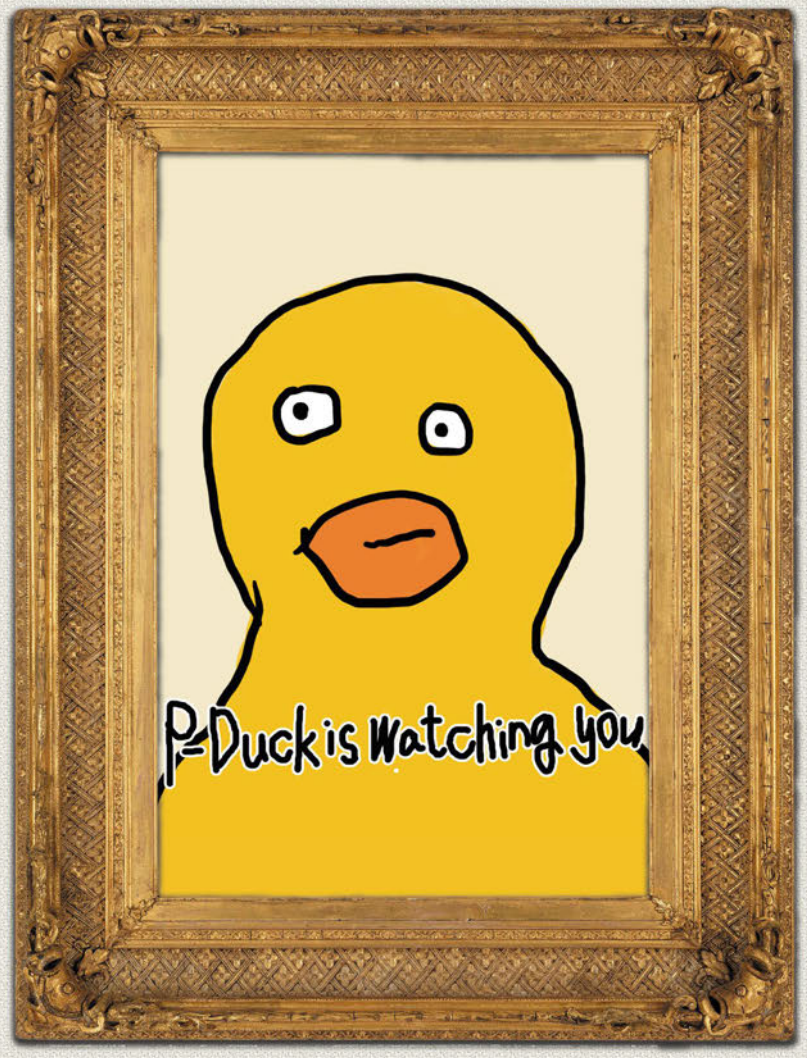
裏表紙キャラクター ngi

すだちくんが好きです。

KLab Tech Book Vol. 1

2017 年 10 月 22 日　技術書典 3 電子版（1.1）
著　者　KLab 株式会社
編　集　岡本和樹
発行所　KLab 株式会社
印刷所　日光企画

(C) 2017 KLab 株式会社


P-Duck is watching you