様式６

申請先学部　　工学部　採択番号　No.工７

# 令和６年度大阪大学未来基金「学部学生による自主研究奨励事業」研究成果報告書

| | | | | | |
|---|---|---|---|---|---|
| ふりがな<br>氏名 | わだ　はると<br>和田　遥大 | 学部<br>学科 | 工学部<br>電子情報工学科 | 学年 | ２年 |
| ふりがな<br>共同<br>研究者氏名 | | 学部<br>学科 | | 学年 | 年 |
| | | | | | 年 |
| | | | | | 年 |

| | | | |
|---|---|---|---|
| アドバイザー教員<br>氏名 | わたなべ　たかし<br>渡辺　尚 | 所属 | 大阪大学大学院<br>情報科学研究科 |

| | |
|---|---|
| 研究課題名 | WebAssembly による仮想現実実行環境の開発 |
| 研究成果の概要 | 研究目的、研究計画、研究方法、研究経過、研究成果等について記述すること。必要に応じて用紙を追加してもよい。(先行する研究を引用する場合は、「阪大生のためのアカデミックライティング入門」に従い、盗作剽窃にならないように引用部分を明示し文末に参考文献リストをつけること。) |

## 1.緒論

### 1.1 研究背景・動機

米国 Meta 社の Meta Quest シリーズや、中国 ByteDance 社の Pico シリーズといった、スタンドアロン、つまりそれ以外の機器なしに仮想現実や複合現実、いわゆる VR や AR を実行できるヘッドマウントディスプレイ(以下 HMD)が誰でも安価で購入できるようになった。これにより、ハードウェアが手に入らないという障壁が小さくなり、VRChat をはじめとする、いわゆる”VRSNS”が流行し、2024 年 1 月には VRChat の同時アクセス数が 10 万人を記録するほどとなっている。(※1)

しかしながら、仮想現実に存在する障壁は入手性の部分に限らない。特に、ストレージが 256GB 程度のスタンドアロン HMD では、インストールできるアプリケーションの数に限界があり、インストールは VR アプリケーションの実行上の大きな障壁となりうる。

そのような背景を踏まえ、ブラウザ上で仮想現実・複合現実環境が実行可能な WebXR という技術を用いれば、インストール不要で仮想現実の体験ができ、その障壁を取り除けるのではないかと考えた。

しかしながら、現状の WebXR の実行環境はブラウザ上のため、JavaScript による実行のみである。JavaScript のようなインタプリタ型の言語では C や Rust のようなコンパイラ型の言語に比べると処理速度が遅い。そのため計算量が多く、速度が必要な仮想現実の実行にはあまり向いていない。そこで、私は近年ブラウザ上で実行可能となり「第四の言語」と呼ばれている(※2)、JavaScript と比較して高速な演算が可能な WebAssembly という技術を用いて WebXR を実行し、高速化することできないかと考え、今回の研究を行った。

### 1.2 研究環境

● ハードウェア

HMD: Meta Quest 3(SoC: Qualcomm Snapdragon XR2+ Gen1)(図１)

図１ 使用した Meta Quest 3

ウェブサーバ：さくらの VPS(CPU: 仮想 3Core, メモリ: 2GB, ストレージ: SSD 100GB)
大阪サーバ(大阪第 3 ゾーン)

- ソフトウェア

プログラミング言語: Rust()、JavaScript(WebAssembly との比較検証用)
ライブラリ: wasm-bindgen(v0.2.93), wasm-bindgen-futures(v0.4.43),futures(v0.3.31),web-sys(v0.3.70), js-sys(v0.3.70), gl-matrix(v0.0.2)

WebAssembly 用の言語には、Rust を選択した。WebAssembly にコンパイルできる言語はさまざまあるが、Rust はこれらの言語の中で最も高速な C/C++と同等のパフォーマンスを持っており、またメモリ安全性やエラーハンドリングが充実しており、デバッグもしやすいと考えたため、Rust を選択した。

### 1.3 研究目的

- WebAssembly を用いた WebXR の実行方法を確立する。
- JavaScript で WebAssembly の実装と等価な実装を行い、実行速度(fps)と使用メモリ量、ファイルサイズの比較を行う。

## 2.研究内容

### 2.1 実装

WebAssembly は、ブラウザ上で実行可能であるものの、DOM 操作等のブラウザ API とやり取りする部分は単体では実行できず、JavaScript の入力を介した実行が必要となる。そのため、wasm-bindgen や web-sys といった、JavaScript の関数を WebAssembly 内で実行可能にするようなライブラリを用いて実行した。(図２)

```
let body = document.body().expect("document doesn't have body.");
let button = document.create_element("button")?.dyn_into::<HtmlButtonElement>()?;
button.set_inner_text("Start WebXR");
let button_clone = button.clone();
let onclick_func = Closure::wrap(Box::new(move ||{
    button_clone.set_inner_text("WebXR is starting...");
    loop{
        let Err(_) = button_tx.try_send(()) else{
            break;
        };
    }
})as Box<dyn FnMut()>);
button.set_onclick(Some(onclick_func.as_ref().unchecked_ref::<js_sys::Function>()));
let _ = body.append_child(&button)?;
```

図２　Rust から JavaScript の関数を呼び出し、WebXR 実行用のボタンを作成する処理

3D レンダリングエンジンは最初、WebGPU を用いて、より高速化を求めた実装をしようとしたものの、残念ながら WebXR の API は WebGL/WebGL2 を前提としたものとなっており、WebGPU でのレンダリング結果を使用することは難しかった。そのため、WebGL2 を使用して実装を行った。GLSL を用いて、フラグメントシェーダとバーテックスシェーダを実装し、fetch API を用いてシェーダを取得、コンパイルした。そして、表示したい 3D 空間上の形状と色を、インターリーブドバッファに入れ、表示する実装をした。ここでインターリーブドバッファーを用いることで頂点と色のデータがメモリ上で近い位置に配置され、実行速度が速くなるよう工夫した。

WebXR の API については、HMD が接続されたブラウザ環境でしか動作ができないため、まずは WebXR に対応した環境であるかを確認し、対応していれば WebXR を動作させるという形で実装した。また、WebXR の API は複合現実や、HMD がない環境での動作にも対応するためにいくつかのモードが用意されているが、今回は仮想現実のみを実行したいため、”immersive-vr”モードでセッションを開始し、WebGL2 のデータと接続した。

そして、最終的に HMD に出力する描画処理は、HMD の２つの目に対して、それぞれビュー変換行列を計算し、描画を行う処理を行った。描画は再帰的に行うように実装しなければならなかったが、Rust においてクロージャは再帰的な処理はできない。そこで、クロージャを自分自身の参照を許すスマートポインタである RefCell でラップし、更にそれを再帰的に実行するため複数の所有者を許すスマートポインタである Rc でラップすることで、この問題を解決した。(図３)

```
let animation_loop = Rc::new(RefCell::new(None::<Closure<dyn FnMut(f64,XrFrame)>>));
let mut fps_tracker = Logger::new(performance);
//初期状態はNone
//RefCellはClosureを後で自分自身を参照できるようにするためのラッパー
//Rcは複数の所有者を持つためのスマートポインタ

let animation_loop_clone = Rc::clone(&animation_loop);
let session_clone = xrsession.clone();
*animation_loop.borrow_mut() = Some(Closure::wrap(Box::new(move |time: f64, frame: XrFrame|{
    fps_tracker.track_frame();
    fps_tracker.log_fps();
    fps_tracker.log_memory_usage();
    render_frame(time, &frame, &reference_space, &session_clone, &gl, &program);
    session_clone.request_animation_frame(animation_loop_clone.borrow().as_ref().unwrap().as_ref().unchecked_ref::<js_sys::Function>());
}) as Box<dyn FnMut(f64,XrFrame)>));

//最初のアニメーションフレームをリクエスト
let _animation_frame_request_id = xrsession.request_animation_frame(animation_loop.borrow().as_ref().unwrap().as_ref().unchecked_ref::<js_sys::Function>());
```

図３　RefCell と Rc を用いた再帰的なクロージャを作成し、描画ループを作成する処理

以上のような流れで、Rust,WebAssembly で計算した結果から WebGL2, WebXR の API へ指令を出す形で処理を行い、WebXR の一連の処理を実装した。まとめると、図４のようになっている。

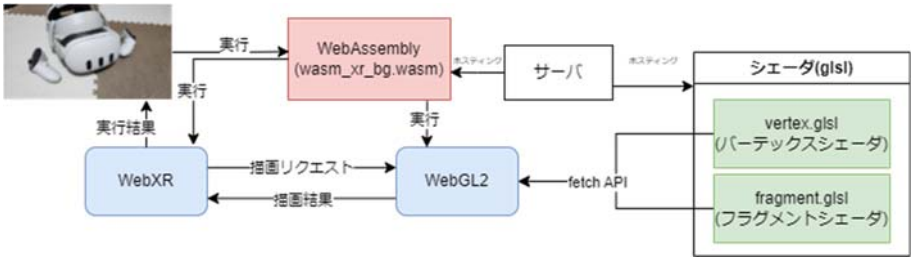


図４ 今回実装したシステム

また、これは基本的にプログラムでの処理結果を JavaScript の関数を介して表示しているので、JavaScript でも等価な処理を実装した。

### 2.2 実行結果

上記のように実装を行うことで、WebAssembly を介した WebXR は実際に動作し、3D 空間上に２色の積み上げられた箱が現れた。実際に、HMD をつけて動くと、物体もそこにあるように視点内で動き、WebXR が WebAssembly 上で実際に動作していることが確認できた。

### 2.3 JavaScript との速度比較

次に、実行速度については、描画のループにかかった時間を３分間実行して計測し、平均の fps(frame per second)を算出した。そして、メモリの使用量は JavaScript の performance を用い、算出した。これを５回行い、その平均を算出した。

また、WebAssembly の最適化については wasm-opt でデフォルトで使われる最適化レベル s(サイズについての最適化)を選択した。

- 計測結果：実行速度

WebAssembly:平均 89.9326 fps

JavaScript:平均　87.484fps

どちらも 90fps に近いのは、Quest3 のリフレッシュレートが 90Hz であるのに合わせていると考えられる。ここで示したように、JavaScript と WebAssembly 間では 2fps 程度の差が出た。この差の原因を調べてみると、図 5 を見るとわかるように JavaScript で実行した際の最初の fps が 0.1~0.2 となっており、以降は 90 に近い値を推移している。実際、WebXR のセッション開始ボタンを押してから、実行されて仮想現実が表示されるまでの時間は WebAssembly のほうが速いことが体感でも感じられた。

| | 1s | 2s | 3s | 4s | 5s | 6s | 7s | 以降の平均 |
|---|---|---|---|---|---|---|---|---|
| WebAssembly | 89.98 | 90.00 | 89.99 | 89.99 | 89.99 | 89.99 | 90.00 | 89.95 |
| JavaScript | 0.18 | 87.87 | 89.98 | 89.98 | 88.21 | 89.93 | 89.94 | 87.54 |

図 5　最初の 7 秒間の fps 推移

- 計測結果：使用ヒープ

WebAssembly: 9.54MB
JavaScript: 9.54MB
このように、使用しているヒープのサイズは変わらず、実行中も変化しなかった。

- 計測結果：ファイルサイズ

WebAssembly: 106797B
JavaScript: 8196B
ファイルサイズでは、このように WebAssembly が圧倒的に大きい。

## 3 考察

実行速度については、どちらも不便がない程度の速度が出ていた。初期の処理速度に関しては、プログラムから実行する行列計算等の処理は主に 3D オブジェクトを出す部分などの初期化の部分で発生していたため、WebAssembly と JavaScript の差がその部分で出たのだと考えられる。以降の部分では、動いたりする 3D モデルがなかったため、計算量が少なく、大きな差が出なかったのだと考えられる。そのため、頻繁にシーンが切り替わるゲームであればこの速度差の恩恵があると考えられる。ファイルサイズは、比較的サイズが小さくなる最適化を入れてもかなりの差が出たため、この点においては JavaScript に優位性があると考えられる。

## 4 結論

今回は、Rust と WebAssembly を用いて WebXR を実行するという新しい試みを行い、見事動作をし、速度面では多少の優位性を見せることができた。しかしながら、開発面では WebAssembly のブラウザでの WebAssembly の実行は、コンパイルの手間があるため、面倒だった。また、WebAssembly では使えない言語機能やライブラリも多く存在した。また、WebXR や WebGL のフレームワークがなく、現状すべて手で実装する必要があったため、すべての処理を書くとかなり長いコードになってしまった。また、今回は断念したが、今後は WebGPU を用いた WebXR の実行に再度挑戦し、レンダリング処理から改善することで、さらに高速で軽量な WebXR の実行に挑みたいと考えている。

## 5 謝辞

本研究においては、アドバイザー教員の渡辺尚先生、ならびに事務職員の髙橋あみさんには本プロジェクトの採択に向けた研究計画書へのアドバイスから、予算執行等までご協力いただき、一人なが

らここまで研究をすることができました。ありがとうございます。

**参考文献**

※１　“VRChat API Metrics”の Current Online Players における 2024/1/1 14:28:00 のデータより (URL:https://metrics.vrchat.community/?orgId=1&viewPanel=2&from=now-2y&to=now&refresh=30s )

※２　W3C 公式ページより( https://www.w3.org/press-releases/2019/wasm/ )