佐賀大学理工学研究科

「高性能計算特論」
講義資料

担当者: 山下義行

R4 年度後期

目次

第1章　はじめに

1.1　講義の概要

この講義では、データを高速に処理する手法を解説する。特に最近、急速に性能が高まっているGPU（Graphics Processing Unit[1]）を用いた高速計算について二つの視点から解説を行う。

ひとつはコンピュータグラフィックス（CG）の視点である。言うまでもなく、GPU は CG の高速描画のためのデバイスとして開発されてきた。CG の描画技法のひとつであるラスタライズ法は定型的な処理であるため、ハードウェアで高速化することに向いており、実際に GPU はラスタライズ法をハードウェアで実装している。この講義では WebGL を用いてその概要を学ぶ。

学部専門科目「コンピュータグラフィックス演習」（旧カリ「コンピュータグラフィックス」）において CG を解説した。しかし、そこで述べた内容は 2D CG や 3D CG の基礎であり、GPU を用いた CG 描画についてはほとんど述べていない。この講義では逆に 2D CG や 3D CG については極力触れず、GPU の利用方法を中心に解説する。

この講義のもうひとつの視点は、CG に限らない一般のデータ並列計算である。大量の定型的な計算を GPU 上で行う方法が検討され始め、現在ではそれ専用のプログラミング環境が適用されている。これを一般に GPGPU（General-Purpose computing on Graphics Processing Units）と呼ぶ。この講義では、テクスチャを用いたラスタライズ法を一般計算に拡張する方法を紹介する。GPU が高速計算と結びつく経緯が分かるはずである。

WebGL は Web ブラウザ上で実行される OpenGL/GLSL のプログラムである。ベース言語はブラウザ上でプログラムを実行できるプログラミング言語 JavaScript である。最新のブラウザならば、どのブラウザであってもほぼ実行可能であり、Windows、MacOS などの OS を問わない。言語処理系のインストールも不要である。この講義で実際に動作するプログラムを示しながら解説を進めていく。講義テキストを執筆するに当たり、講義担当者が調べた限りでは、この授業で紹介するプログラムは Windows、MacOS の主要な Web ブラウザで正常に実行できている。

OpenGL の機能を調べるに当たり、和歌山大学システム工学部デザイン情報学科床井研究室のホームページ[2]を参考にした。ホームページを通じて貴重な情報をご教示いただいたことを床井浩

[1]術語として “Graphics Processing Unit” ではなく、“Graphical ...” を用いている論文も散見される。
[2]`http://marina.sys.wakayama-u.ac.jp/~tokoi`

1　平先生に感謝いたします。

第2章　JavaScriptの概要

2.1　ハローワールド（その1）

JavaScript は、そのプログラムが含まれた HTML ファイルを Web ブラウザで開くとプログラムが実行される方式の非常に手軽なプログラミング言語である。Web ブラウザの上から GPU を操作する WebGL の関数群の呼び出しはこの JavaScript から行う。

図 2.1（4 ページ）は、JavaScript のハーローワールドの表示および四角形の描画を行う JavaScript を含む HTML ファイルの例である。これをそのまま適当な Web ブラウザで開くと、図 2.2（4 ページ）の描画が表示される。

このプログラムの冒頭：

```
onload = function() {
    ...
};
```

は、= の右辺で無名関数を定義し、その参照（reference、つまりメモリアドレスのようなもの）を左辺の変数、JavaScript の `window` オブジェクトのイベントハンドラである `onload` に格納することを表している。このイベントハンドラの参照する関数は、HTML に定義されている全てのリソースをブラウザに読み込んだ後に実行されると約束されている。つまり、準備が万端整った後にプログラムが実行される仕組みである。

HTML ではブラウザの表示を遅延させないために、通常は読み込んだリソースを他のリソースの読み込み終了を待たずに表示する仕組みになっている。この仕組みはユーザを待たせない表示には適する。しかしリソースの読み込みを待たないため、JavaScript のプログラム実行に支障をきたす場合があり、JavaScript ではしばしば `onload = ...` が利用される[1]。

さて、関数の内部の冒頭：

```
let canvas = document.getElementById("canvas");
```

[1] なお、`onload` は正確には JavaScript の `Window` オブジェクトのメンバー変数（インスタンス変数）である。JavaScript のプログラムは、そのプログラムを含む HTML ファイルに対応付けられた `Window` オブジェクト内で動作するという暗黙の設定のため、いちいち `Window` オブジェクトを指定する必要はない。

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<script>
    onload = function(){
            let canvas = document.getElementById("canvas");
            let context = canvas.getContext("2d");

            context.font = "24px Arial, meiryo, sans-serif" ;
            context.fillText( "Hello World!", 50, 50) ;

            context.fillRect(20, 80, 100, 50);
    }
</script>
</head>

<body>
<canvas id="canvas"></canvas>
</body>
</html>
```

図 2.1: ハローワールドと長方形を描く HTML/JavaScript のプログラム（その 1）

図 2.2: 図 2.1 の実行結果

では、HTML ファイルの下部のキャンバス部分（実際に描画が行われる部分）：

```
<canvas id="canvas"></canvas>
```

を id 名をキーにして、オブジェクトとして取得する。次に、

```
            let context = canvas.getContext("2d");
```

では、描画方法（"2d" = 2 次元グラフィックス描画の意味）を指定して、キャンバスのコンテキストを取得する。そして、その後に、文字列の描画：

```
context.font = "24px Arial, meiryo, sans-serif" ;
context.fillText( "Hello World!", 50, 50) ;
```

1

2 および、四角形の描画：

```
context.fillRect(20, 80, 100, 50);
```

3

4 を行う。

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<script src="./main.js" type="text/javascript"></script>
</head>

<body>
<canvas id="canvas"></canvas>
</body>
</html>
```

図 2.3: ハローワールドと長方形を描く HTML のプログラム（その 2）

```
onload = function(){
    let canvas = document.getElementById("canvas");
    let context = canvas.getContext("2d");

    context.font = "24px Arial, meiryo, sans-serif" ;
    context.fillText( "Hello World!", 50, 50) ;

    context.fillRect(20, 80, 100, 50);
}
```

図 2.4: ハローワールドと長方形を描く JavaScript のプログラム（その 2、main.js）

2.2　ハローワールド（その 2）

JavaScript のプログラムを HTML ファイルに埋め込むのではなく、別のファイルに用意することもできる。

図 2.3（6 ページ）には、JavaScript のプログラムをファイル名 `"./main.js"`（HTML ファイルと同じディレクトリ内の `main.js`）で指定している。そして JavaScript のファイルは、図 2.4（6 ページ）のように別途用意する。

この方法でプログラムを複数のファイルに分割して作成すれば、ひとつのファイルが長大にならず読み易い。

このテキストでは、次章以降、「（その１）」ではなく、「（その２）」の方式でプログラムを作成する。

第3章　WebGLの概要

WebGL を用いたプログラムの具体例は次章以降と示すとし、この章では WebGL を用いた CG プログラムの概要を述べる。

3.1　プログラムの全体構成

GPU を用いるプログラムは、CPU 上で動作する**ホストプログラム**と GPU 上で動作する**シェーダプログラム**[1]からなる。

ホストプログラムは GPU を制御するためのプログラムであり、通常の JavaScript プログラムの実行と同様に CPU 上で動作される。GPU を制御するための API（Application Programming Interface）には WebGL を用いる。これは OpenGL の Web ブラウザ版である。

OpenGL は、かつて最先端の CG 専用コンピュータの開発・販売を行っていたシリコン・グラフィックス社が社内用に開発したグラフィックス・ライブラリの仕様を 1993 年に公開したものである。仕様の抽象度が高く、様々なグラフィックス装置への移植が比較的容易であったため、普及が進んだ。抽象度の高さは逆に個々のグラフィックス装置の性能を引き出すことを阻害する要因ともなるため、たとえばマイクロソフトの DirectX を利用した場合に比べ、描画速度がやや遅いという欠点もある[2,3,4]。

シェーダプログラムは GPU の上で実行されるプログラムである。WebGL では GLSL を用いる。GLSL は、OpenGL Shading Language の略であり、OpenGL（この授業では WebGL）と対になって GPU 上のプログラムを記述するプログラミング言語である。1990 年代の公開された当初の OpenGL には GPU を操作する機能は提供されていなかった。しかし、急速に機能拡張し、高性能化する GPU を効果的に利用するために、OpenGL の公開からほぼ 10 年遅れて GLSL の

[1] shader のカタカナ表記として「シェーダ」を用いるか、「シェーダ」を用いるか迷いました。JIS の規定では前者ですが、最近、後者が増えているとのこと。このテキストでは前者を採用します。

[2] DirectX はマイクロソフトが Windows に提供するものであるため、マクロソフト自身が精力的にサポートしている点も大きい。それに対して、OpenGL のサポートはおざなりになりがちである。

[3] 最近では、最新の GPU を想定し、かなり低レベルの操作ができるグラフィックス API として、Vulkan や Metal が提案されている。講義担当者は最近は Metal を用いている。

[4] OpenGL が DirectX に比べて本当に遅いのか、実はネット検索しても確たる情報は出てこない。そこで、山下研の卒業研究で実際に同じ構造のプログラムの実行速度を比較する研究を行い、OpenGL がやや遅いことを実証した。

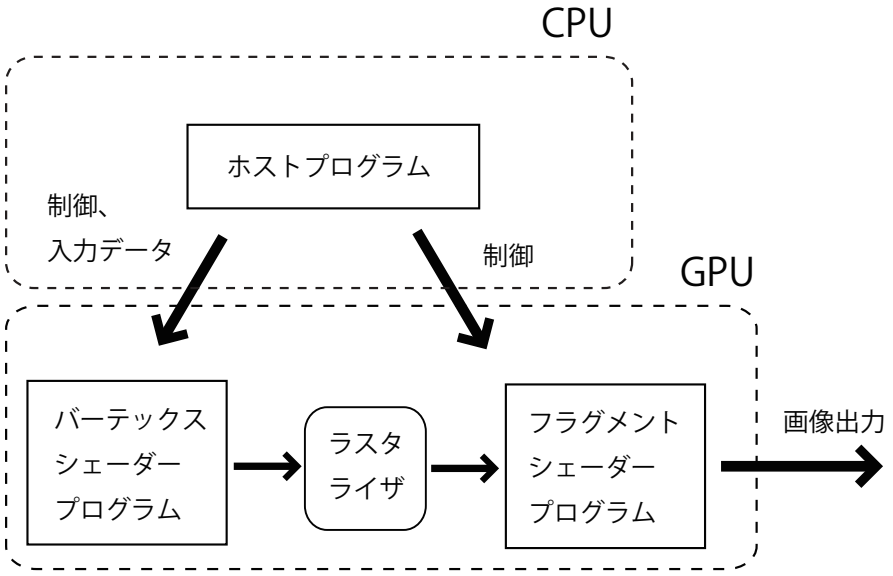


図 3.1: 3 種類のプログラムの関係

仕様が策定された。これによって GPU を高水準プログラミング言語レベルで利用できるようになった。

図 3.1（8 ページ）は、WebGL のプログラムの構成図である。GLSL のプログラム（以下、シェーダプログラム）は WebGL が動作するホストプログラムの管理下で実行される。GPU は CG の様々な処理をパイプライン的に実行しており、パイプラインの各ステージ毎にシェーダプログラムを作成する必要がある。この講義で用いるシェーダプログラムは、**バーテックスシェーダプログラム**と**フラグメントシェーダプログラム**の 2 種類である。この二つがライスタライザを挟んで 3 段のパイプラインを成している。具体例は後に述べる。

結局、WebGL/GLSL を用いて CG 描画を行う場合、プログラマは少なくとも以下の 3 種類のプログラム

1. ホストプログラム（JavaScript ＋ WebGL の関数群）
2. バーテックスシェーダプログラム（GLSL）
3. フラグメントシェーダプログラム（GLSL）

を書かねばならない。

実は最近の CG 描画では図 3.1 のような単純な構成では不十分であり、これにジオメトリシェーダ（geometry shader）[5]、テッセレータ（tessellator）などの別機能のシェーダを追加できる（追加

[5]ジオメトリシェーダは並列計算を阻害する動作があることが知られるようになり、最近ではほとんど用いられない。

は必須ではない)。また、計算に特化した計算シェーダ（コンピュートシェーダ、 compute shader）も利用できるようになってきている。

WebGL の機能、構造は全て OpenGL のそれを引き継いでおり、この講義テキストで学ぶ内容は OpenGL でも基本的に通用する。ただし、この講義テキストで解説する WebGL（正確には WebGL ver. 1）は、最新の OpenGL の仕様からは２０年近く遅れた仕様である。

WebGL の最新仕様は ver. 2 である。しかし現時点では、いくつかの Web ブラウザではこの仕様をサポートしていないものもあるため、この講義テキストでは WebGL ver. 1 を採用することとする。

注意： 上に述べたように、WebGL の仕様は OpenGL の旧版の仕様である。そのため、これ以降の解説で「WebGL では...」と書かれている箇所の多くは「旧版の OpenGL では...」と言い換えてもよいし、逆もそうである。

3.2　ホストプログラムの構造

WebGL はグラフィックス処理のコア部分 — たとえば画面上に線分を引く、三角形を描画する — に関する仕様である。コアに付随する処理 — たとえばディスプレイ上にウインドウを開く/閉じる／移動する、タイマー割り込みを用いる – などは WebGL の仕様には含まれない。

コア以外の部分は、使用する Web ブラウザと JavaScript に依存するが、ほとんどの Web ブラウザで共通仕様（HTML5）が採用されており、同じ描画が可能である。少なくとも講義担当者が、Windows、MacOS 上の 3 種類の Web ブラウザでこのテキストで解説するプログラムを実行した限りでは同じ描画、同じ計算が可能であった。

WebGL を用いるホストプログラムの一般的な処理手順と処理内容は以下の通りである。

実行時環境の初期化： まず、WebGL の定型的な初期化、描画ウインドウの生成などを行う。

シェーダ実行可能コードの準備： シェーダプログラムのコンパイル・リンク、GPU への転送、登録を行う。

シェーダへの入力データの準備： 実際に描画する被写体のデータをホストプログラムから GPU へ転送するなどの準備を行う。

シェーダ実行可能コードの起動： シェーダ実行可能コードを起動する。

補足：　上記「シェーダ実行可能コードの準備」についてやや長い補足解説を行う。

まず、プログラムには、ソースプログラム、オブジェクトコード、実行可能コードの三種類があることを思い出そう。ここまで「シェーダプログラム」と呼んでいたものは暗にソースプログラムのことを指していた。これ以降、混乱を避ける場合には **シェーダソースプログラム** という呼ぶ方も用いることとする。それをコンパイルしたものを **シェーダオブジェクトコード** と呼ぶこととする。さらに、それをリンクしたものを **シェーダ実行可能コード** と呼ぶこととする。

現在、GLSL シェーダの機械語の統一仕様は策定されていない。策定できないと考えるべきかもしれない。そのため、それぞれの GPU のための専用コンパイラはそれぞれのデバイスドライバと対になってベンダーから供給されている状況である。結果、シェーダソースプログラムのコンパイルは、Visual Studio や Xcode のような統合開発環境に組み込まれて実行されるのではなく、ホストプログラムから WebGL のコンパイラを関数呼び出しする簡易的な方式で実装されている。この方式の問題点は以下の通りである。

1. アプリケーションプログラム実行の中でコンパイルされるため、アプリケーションプログラムの立ち上がりが遅くなる。たとえばゲームならば、実際にゲームがスタートするまでにユーザは多少待たされることになり、好ましくない。

2. 上記のユーザーの待ち時間の短縮のために、コンパイラは高度なコード最適化処理[6]を実施できない。もしシェーダソースプログラムを事前にコンパイルできるならば、十分な時間を掛けて高度で複雑な最適化処理を実施できる[7]。

実行時コンパイルのこれらの問題点は広く認識されているが、全面的な解決には至っていない。

3.3　シェーダプログラムの構造

既に述べたように、ユーザは少なくとも 2 種類のシェーダソースプログラムを作成する必要がある。それらはそれぞれのシェーダ上で動作し、以下のような役割を担っている。

バーテックス・シェーダ（vertex shader） ... 頂点（vertex）シェーダともいう。3D CG では主に頂点の座標変換をこのシェーダで行うため、この名称が付いている。実際には座標変換に限らず、自由な計算ができるため、アイディア次第では GPU の面白い使い方ができる。

ホストプログラムからバーテックスシェーダへの入力は、各頂点の位置、色、その他の付随情報であり、ユーザが自由に設計できる。

計算結果はバーテックスシェーダから出力され、ラスタライザに入力される。

[6]機械語の命令を並べ換えるなどの処理をいう。計算の流れを解析する必要があり、通常、複雑な解析処理を必要とする。
[7]たとえば Metal のシェーダプログラムは Xcode 環境で事前にコンパイルできる。

```
function initSystem() {
    /* ここに WebGL その他の初期化処理 */
}

function initData() {
    /* ここにプログラムで利用するデータの初期化処理 */
}

function display() {
   /* ここに描画処理 */
}

window.onload = function(){
    initSystem();
    initData();
    display();
};
```

図 3.2: ホストプログラムのひな形

フラグメント・シェーダ（fragment shader） DirectX ではピクセル（pixel=画素）シェーダと呼ぶ。主にラスタライズで求められた各画素の色の計算を行う。

　バーテックスシェーダから出力された頂点情報は、ラスタライザを経由して各画素毎の情報へ線形補間されて、フラグメントシェーダに入力される。フラグメントシェーダではその入力情報を元に個々の各画素の色（と必要ならば深度情報）を計算する。色（と深度情報）はフレームバッファ上の対応する画素へ書き込まれる。

　シェーダプログラムは GLSL で記述するが、GLSL は C 言語とほとんど同じ基本構文を持つ言語であるから、C 言語に慣れている者にはプログラミングのストレスがほとんどない。C 言語と異なるのは、データの入出力に関連する特殊なルール、ベクトルデータ型導入などが付加されている点である。詳細は次章以降に例題を用いて丁寧に解説していく。

3.4　WebGL/GLSL プログラムの外形

　このテキストでは、WebGL を用いる JavaScript のホストプログラムの形状を図 3.2（11 ページ）の通りとする。プログラムの読みやすさのため、システム関連の初期化を行う関数 `initSystem()`、被写体データの初期化を行う関数 `initData()`、 シェーダプログラムを起動し、GPU で描画を行う関数 `display()` を分けて定義する。この講義では一貫してこの形のプログラムを用いることとする。具体例は次節以降で解説する。

第4章　スタートアップ：簡単な線画

この章で作成する CG 画像は、バツ印（×）を二本の線分で描く図 4.1 である。この単純な画像を作成するプログラムを丁寧に紹介していく。

これ以降の解説を読むと、図 4.1 のような単純な線画を描画するためだけに**うんざり**するような量のプログラムを書かねばならないことに驚くだろう。しかし高度なグラフィックス処理には避けて通れないと理解してほしい。その点では OpenGL（＝ WebGL）だけではなく、DirectX、Metal なども同様である。

量が多いだけではなく、OpenGL のプログラムは分かりにくい。しばしば OpenGL は「**何故、このように複雑でわかりにくい仕様**を採用しているのか？」と批判される。基本仕様が策定されたのが 30 年前であり、当時のコンピュータのアーキテクチャの制限を反映したものであり、またさまざまなハードウェアで実行できるように抽象化されたためである。その点で言えば、最近、新しく仕様策定された Metal などはすっきりとしており、比較的理解し易い仕様になっている。

なお、WebGL は一部、オブジェクト指向風の仕様を取り入れており、本家の OpenGL よりも少しだけ読み易いプログラムになっている。

4.1　全体のプログラム

図 4.2（14 ページ）、図 4.3（20 ページ）、図 4.8（33 ページ）がホストプログラムである。特に図 4.2 がトップレベルの主要部であり、図 3.2 のひな形に当てはめて書かれている。図 4.3 は、ホストからシェーダへデータを転送するための設定部分である。図 4.8 は、これについては章末に述べるが、シェーダプログラムのコンパイル、リンクの処理を行うプログラム部分である。

バーテックスシェーダプログラム、フラグメントシェーダプログラムは、HTML ファイルのリソースとして定義する。その HTML ファイルは図 4.5（26 ページ）である。

補足（JavaScript の関数定義）：　JavaScript プログラム中では、関数は以下の構文で定義する。

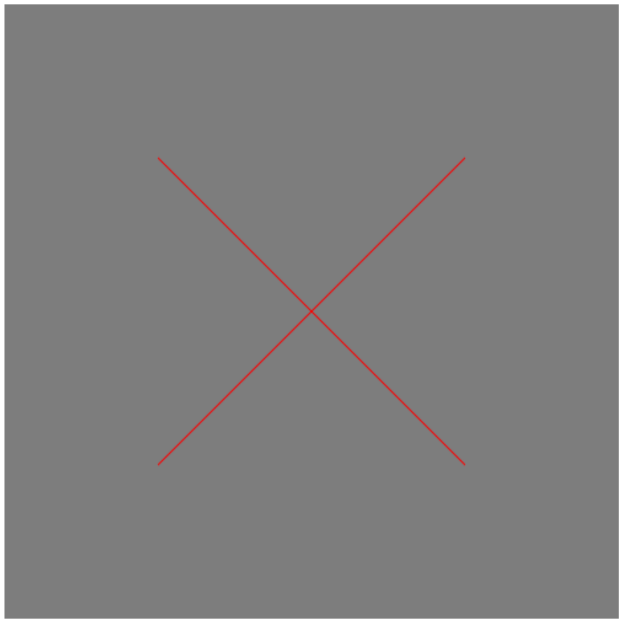


図 4.1: 画像例（その 1）

```
function 関数名（仮引数リスト）{
    関数本体
}
```

4.2　ホストプログラム：主要部

4.2.1　大域変数の宣言

図 4.2 冒頭のグローバル宣言：

```
let gl;                                          // グラフィックスコンテキスト
```

は、グラフィックスコンテキストを保持する変数の宣言である。このグラフィックスコンテキストはプログラム中のほぼ全域からアクセスされるため、グローバル宣言する。

次の行の

```
let program;                                     // シェーダ実行可能コード
```

```
let gl;                                          // グラフィックスコンテキスト
let program;                                        // シェーダ実行可能コード

function initSystem() {                             // 描画システムの初期設定
    let c = document.getElementById('canvas');          // キャンバスを取得
    c.width = 500; c.height = 500;                // キャンバスのサイズを設定

    gl = c.getContext('webgl');     // WebGL 用グラフィックスコンテキストを取得

    gl.clearColor(0.5, 0.5, 0.5, 1.0);                        // 背景色を指定

    buildProgram('vs', 'fs');              // シェーダ実行可能コードのビルド
    gl.useProgram(program);          // ビルドしたプログラムを利用可能にする
}

let NUM_POINTS = 4;                                              // 頂点数の設定

function initData() {                                 // 描画データの初期設定
    let position = [                            // 2 次元頂点データを配列に設定
        -0.5, -0.5,                                                  // 左下
        +0.5, +0.5,                                                  // 右上
        +0.5, -0.5,                                                  // 右下
        -0.5, +0.5                                                   // 左上
    ];

    let buffer = buildArrayBuffer(position);  // 配列から頂点バッファを生成

    bindArrayBuffer(buffer, 'position', 2, program);
                    // buffer を シェーダ内の attribute 変数 position と結合
}

function display() {                                               // 描画を行う
    gl.clear(gl.COLOR_BUFFER_BIT);             // キャンバスを背景色でクリア
    gl.drawArrays(gl.LINES, 0, NUM_POINTS);
                               // 頂点バッファの先頭（0）から NUM_POINTS-1
                               // までの頂点を利用して線画を描画

    gl.flush();                                          // GPU に描画処理を催促
}

window.onload = function(){                           // main 関数に相当する
    initSystem();                                    // 描画システムの初期化
    initData();                                       // 描画データの初期化
    display();                                                    // 描画
};
```

図 4.2: ×印を描画する JavaScript ホストプログラム（主要部、main.js）

は、コンパイル＆リンク後のシェーダ実行可能コードを参照する変数の宣言である。これも全域からアクセスされるため、グローバル宣言する。

補足（JavaScript のデータ型宣言について）：　JavaScript は動的型付の言語であるため、変数の宣言では変数にデータ型を付けない。よって上の 2 行では、let キーワードの後に単に変数名を与えている。

補足（JavaScript の var、let、const）：　let は、直後に宣言された識別子が変数であることを示すものである。同様のものとして、var キーワードがある。var と let の違いは変数のスコープである。一般に、C/C++と同等のスコープを持つ変数は let で宣言されたものになる。var はブロック（{ }）の中で宣言されていても、ブロックの外でもその変数にアクセスできるという広いスコープを持っており、C/C++の感覚からは広過ぎる。通常、let ではなく、var が必要な局面はほとんど無いから、このテキストでは専ら let を用いる。const は定数宣言（より正確に言えば、その変数に代入された値を二度と更新できない変数[1]の宣言）である。

4.2.2　描画システムの初期設定

ここでは関数 initSystem() について解説する。

冒頭の

```
let c = document.getElementById('canvas');          // キャンバスを取得
```

では、前章で述べたように、HTML ファイルのキャンバス部分を取得する。次の行：

```
c.width = 500; c.height = 500;                  // キャンバスのサイズを設定
```

では、そのキャンバスの横、縦のサイズを画素で設定する。さらに、次の行：

```
gl = c.getContext('webgl');     // WebGL 用グラフィックスコンテキストを取得
```

では、グラフィックスコンテキストを取得する。前章では、引数として "2d" を指定したが、ここでは 'webgl' を指定する。'webgl' のグラフィックスコンテキストは WebGL の様々な機能を受け付ける。

[1] const 宣言された変数にオブジェクトを初期代入した場合、変数へ別のオブジェクトを再代入することはできませんが、オブジェクト内の属性の変更は可能です。

補足（JavaScript の文字列）　紛らわしいが、JavaScript では文字列をシングルクォーテーション ’〜’ でもダブルクォーテーション "〜"でも記述できる。よって、 ’webgl’ は、 "webgl" でもよい。

さらに、

```
gl.clearColor(0.5, 0.5, 0.5, 1.0);                    // 背景色を指定
```

は、キャンバスの背景色を設定する。設定のみであり、この時点では背景色で塗りつぶすことはしない。引数 0.5，0.5，0.5 は RGB 輝度値を指定している。0 が最低輝度、1 が最大輝度のため、この指定は灰色（図 4.1 参照）である。4 番目の引数値 1.0 は不透明度（やはり、0〜1 で指定）である。1 は完全不透明であり、直前の輝度値を上書きする[2]。なお、clearColor() は WebGL のグラフィックスコンテキスト gl のメソッドである。通常の OpenGL では、これは関数 glClearColor() として提供されているものである。

次の行：

```
buildProgram(’vs’, ’fs’);              // シェーダ実行可能コードのビルド
```

はシェーダソースプログラムをコンパイルし、バーテックスシェーダオブジェクトコードとフラグメントシェーダオブジェクトコードをリンクし、実行可能コードを作成し、グローバル変数 program に格納する。関数 buildProgram() については後述する。引数 ’vs’, ’fs’ は HTML ファイル中の文字列リソースの id であり、それぞれバーテックスシェーダソースプログラム、フラグメントシェーダソースプログラムを表す。これについても後述する。

最後の行：

```
gl.useProgram(program);          // ビルドしたプログラムを実行可能にする
```

は、作成されたシェーダ実行可能コードを実行可能になるように設定する。通常、GPU の上では複数のシェーダプログラムは実行させ、複雑な CG 描画を行う。その場合、実行するプログラムの設定を次々と変えながら実行していく。それが上の 1 行である。

[2]不透明度が 1.0 よりも小さい場合、下地が透ける色で塗ることに相当する。

4.2.3　描画データの初期設定

まず、以下の 1 行：

```
let NUM_POINTS = 4;                                    // 頂点数の設定
```

は、描画に用いる頂点の数を設定しています。この情報は関数 `display()` でも参照するため、グローバルに宣言する。

関数 `initData()` の冒頭：

```
    let position = [                              // 2 次元頂点データを配列に設定
        -0.5, -0.5,                                                    // 左下
        +0.5, +0.5,                                                    // 右上
        +0.5, -0.5,                                                    // 右下
        -0.5, +0.5                                                     // 左上
    ];
```

は、四つの 2 次元座標値を 1 次元配列に順に格納したものである。このデータを GPU へ転送し、描画に用いる。

配列中にデータを並べる順番には以下の単純なルールがある。

1. 2 次元座標値は、以下の図のように、x 座標値、y 座標値をこの順番で[3]配列要素に連続して格納する。

...	x 座標値	y 座標値	...

2. 線分を表すときには、以下の図のように、始点の 2 次元座標値、終点の 2 次元座標値を配列要素に連続して格納する。

...	始点の 2 次元座標値	終点の 2 次元座標値	...

結果、ひとつの線分を表すデータの並びは以下でよい。

...	始点の x 座標値	始点の y 座標値	終点の x 座標値	終点の y 座標値	...

二つの線分ならば、それらデータを配列に連続して配置することになるから、4 頂点を定義する必要がある。

[3]厳密に言えば、逆順に並べてもよいが、その場合にはバーテックスシェーダプログラムでも逆順に取り扱う必要がある。

プログラムの次の行：

```
let buffer = buildArrayBuffer(position);  // 配列から頂点バッファを生成
```

は、頂点データを格納した配列 position のデータを用いて、GPU 上の配列 — これをバーテックスバッファオブジェクト（vertex buffer object、VBO）と呼ぶ — を準備する。

そして、最後の行：

```
bindArrayBuffer(buffer, 'position', 2, program);
                  // buffer を シェーダ内の attribute 変数 position と結合
```

では、VBO をシェーダプログラム中の attribute 変数 position と結合する。この 2 行の実際の処理はかなり複雑であるため、二つの関数にまとめた。詳細は後述する。

4.2.4　描画処理

描画処理関数 display() の最初の処理はキャンバスの初期化である。

```
gl.clear(gl.COLOR_BUFFER_BIT);              // キャンバスを背景色でクリア
```

次にシェーダプログラムの起動を行う。

```
gl.drawArrays(gl.LINES, 0, NUM_POINTS);
                            // 頂点バッファの先頭（0）から NUM_POINTS-1
                            // までの頂点を利用して線画を描画
```

関数呼び出し gl.drawArray() — 文字通り「配列を描く」 — がシェーダプログラムを起動する。描画の中核部分である。引数の意味は以下の通り。

1. 第 1 引数には描画する被写体の形状を指定する。たとえば

   ポイントスプライトの場合 ... gl.POINTS を指定する。後章で用いる予定。

   線分の場合 ... gl.LINES、gl.LINE_STRIP、gl.LINE_LOOP などを指定する。詳細は省略。

   三角板の場合 ... gl.TRIANGLES、gl.TRIANGLE_STRIP、gl.TRIANGLE_FAN などを指定する。詳細は省略。後章で用いる予定。

ここでは単純な線分を用いるため、gl.LINES を指定する。

2. 第 2 引数には、事前に登録した頂点データの配列について、描画を開始する頂点の位置を指定する。通常は関数 initData() で設定した配列の先頭から描画するから 0 を指定する。少し凝ったプログラムでは 0 以外の位置を指定することもある。

3. 第 3 引数には、描画に用いる頂点の数を指定する。今回の例題では頂点数は NUM_POINTS である。

最後に、

```
gl.flush();                                    // GPU に描画処理を催促
```

上の 1 行は、glDrawArrays() による描画を強制する。というのも、OpenGL の仕様では直前の gl.drawArrays() は描画を予約する関数であって、描画を実行する関数ではない。gl.drawArrays() を呼び出したからといって直ちに描画が開始されるとは限らないのである。しかし、実際にはそのような描画の遅延が生じる特殊な状況は複数台の PC がサーバー/クライアントの関係でネットワークを介して描画処理を行うような場合にのみ考えられるから、単体の PC で描画を行う場合には通常は描画の遅延は起きない。上の 1 行は OpenGL の慣用と考えてよい。

補足：線の太さを変えることはできるか？:　図 4.1（13 ページ）の図の線の太さを変えることはできない。以前のネイティブな OpenGL では可能であった。現在も一部のシステムでは可能かもしれないが、最近の OpenGL などのグラフィックスライブラリの考え方は「太い線分を描きたいならば細長い長方形を用いなさい」となっており、gl.LINES ではシステム標準の太さの線しか描くことができない。

```
function buildArrayBuffer(data) {          // 頂点バッファオブジェクトの作成
    let arrayBuffer = gl.createBuffer();           // 空のオブジェクトの作成

    gl.bindBuffer(gl.ARRAY_BUFFER, arrayBuffer);      // オブジェクトを束縛

    gl.bufferData(gl.ARRAY_BUFFER,                 // 配列 data の内容を GPU へ
                  new Float32Array(data),          // コピーし、頂点バッファ
                  gl.STATIC_DRAW);                 // オブジェクトを作成

    gl.bindBuffer(gl.ARRAY_BUFFER, null);                     //  束縛を外す

    return arrayBuffer;                           // 作成したオブジェクトを戻す
}

function bindArrayBuffer(arrayBuffer, name, s, program) {
    gl.bindBuffer(gl.ARRAY_BUFFER, arrayBuffer);      // オブジェクトを束縛

    let location = gl.getAttribLocation(program, name);
                                   // バーテックスシェーダ内の name と同じ
                                   // 名前の attribute 変数の GPU 内の位置を取得

    gl.vertexAttribPointer(location, s, gl.FLOAT, false, 0, 0);
                                      // location の頂点バッファの属性を設定

    gl.enableVertexAttribArray(location);
                                   // location の頂点バッファを利用可能にする
}
```

図 4.3: ×印を描画する JavaScript ホストプログラム（頂点バッファ関連、util.js）

4.3　ホストプログラム：頂点バッファの設定

この節では、前節の以下の 2 行：

```
    let buffer = buildArrayBuffer(position);  // 配列から頂点バッファを生成

    bindArrayBuffer(buffer, 'position', 2, program);
                    // buffer を シェーダ内の attribute 変数 position と結合
```

の内部処理について解説する。ここは難解である。非常に不自然で分かりにくい処理であるから、**あまり悩まず、さっさと次へ行く程度の軽い気持ち**で読み進めてよい。

関数 buildArrayBuffer()、bindArrayBuffer() の定義は図 4.3 の通りである。その処理手順、内容を図 4.4 を用いて説明する。なお、以下の説明中の「OpenGL」は「WebGL」に読み替えてよい。

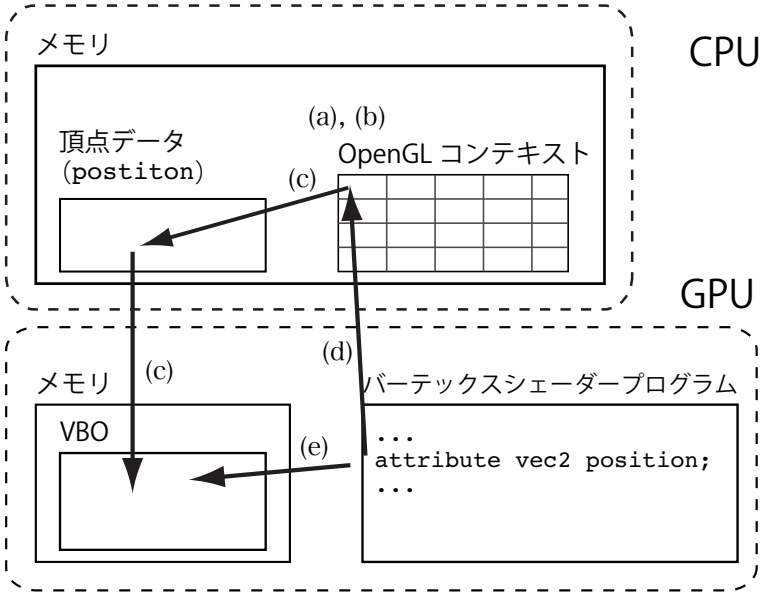


図 4.4: GPU メモリへの頂点データの転送：(a) OpenGL コンテキストにバッファオブジェクト arrayBuffer を生成する。(b) arrayBuffer を Vertex Buffer Object とみなし、これ以降の設定対象とする。(c) arrayBuffer に対応する GPU のメモリ領域を確保し、CPU のメモリからデータを転送する。(d) シェーダプログラムから特定の attribue 変数の場所 location を求める。(e) arrayBuffer と location を結合する。

4.3.1　VBO の作成

OpenGL ではプログラムの実行に関わる様々な情報を OpenGL コンテキストとして一元管理している。ただし、プログラマはこのコンテキストに直接にはアクセスできず、関連する OpenGL の関数群を用いてのみ間接的にアクセスできる。GPU のメモリ上のデータも OpenGL コンテキストの管理対象であり、OpenGL ではこのデータをバッファオブジェクト（Buffer Object）と呼んでいる。関数 buildArrayBuffer() の定義の 1 行目：

```
let arrayBuffer = gl.createBuffer();          // 空のオブジェクトの作成
```

は、OpenGL コンテキスト内にひとつのバッファオブジェクトを新たに生成する操作である。図 4.4 に一連の処理を図示しているが、この関数呼び出しは図の (a) に相当する。上のプログラムでは、生成されたバッファオブジェクトは変数 arrayBuffer から参照できる。なお、まだこの時点では GPU のメモリ上にバッファ領域は確保されてはいない。

補足（ネイティブな OpenGL）： WebGL ではないネイティブな OpenGL ではベース言語を C 言語としている。よって、オブジェクト（オブジェクト指向プログラミングでいうところのオブジェクト）を利用できず、バッファの管理はバッファID で行っている。そのため、ネイティブな OpenGL のプログラムはこのテキストのプログラムよりもさらに読みにくいものになっている。OpenGL のプログラムの読みにくさの主因は、OpenGL の仕様が 30 年前のコンピュータの状況をベースに作られていることにある。

バッファオブジェクトにはその用途によって様々な種類があり、種類毎に設定する方法が異なる。不便なことに OpenGL では「同時にはひとつのバッファオブジェクトしか設定の対象にできない」という仕様になっている。それを踏まえて、次の 1 行を見てみよう。

```
gl.bindBuffer(gl.ARRAY_BUFFER, arrayBuffer);       // オブジェクトを束縛
```

この 1 行は、バッファオブジェクト arrayBuffer について、それを gl.ARRAY_BUFFER という種類のバッファとしてこれ以降の設定の対象にすることを指定する（図 4.4 (b) 参照）。gl.ARRAY_BUFFER は、そのバッファオブジェクトがバーテックスシェーダへの入力配列バッファであることを意味する。この種類のバッファを、前節で触れたように、VBO（バーテックスバッファオブジェクト）と呼んでいる。

次の行：

```
gl.bufferData(gl.ARRAY_BUFFER,                  // 配列 data の内容を GPU へ
              new Float32Array(data),           // コピーし、頂点バッファ
              gl.STATIC_DRAW);                  // オブジェクトを作成
```

は、以下の処理をまとめて行う関数である（図 4.4 (c) 参照）。

1. 第 1 引数 gl.ARRAY_BUFFER にはこの関数呼び出しで対象とするバッファの種類を指定する。直前の関数呼び出し gl.bindBuffer() において、 arrayBuffer が gl.ARRAY_BUFFER であると設定しているため、 arrayBuffer がこの関数呼び出しの設定の対象である（分かりにくい！）。

2. 第 2 引数には CPU の配列から GPU のバッファオブジェクトへデータを転送する際の、CPU 側の数値配列を指定する。

   上の 1 行の場合、data は引数で渡された数値の配列である。この数値の具体的なデータ型は規定されていない。これを 32bit 浮動小数点数値の配列へ変換するのが、new Float32Array(data)

である。通常、GPU が扱う浮動小数点数値データは 32bit である[4]から、data が整数型の場合、あるいは 64bit 浮動小数点数値型の場合などには個々の配列要素についてデータ型変換が必要である。

なお、CPU のメモリ上のデータを GPU のバッファオブジェクトへ転送する方法は他にもある[5]のだが、初学者には gl.bufferData() を用いる方法が最も単純で変な誤解が生じにくいと思われる。

3. 第 3 引数 gl.STATIC_DRAW は、確保する GPU のメモリがシェーダプログラム実行時には主にデータの読み出しに利用されることを宣言する。この宣言を間違えてもシェーダプログラムは（たぶん）正常に動作するが、データの読み出しに無駄な時間が掛かる場合がある。引数として、他に gl.STREAM_DRAW、gl.STATIC_COPY、gl.DYNAMIC_DRAW が指定可能である。

関数 buildArrayBuffer() の処理の最後は以下：

```
gl.bindBuffer(gl.ARRAY_BUFFER, null);                    //  束縛を外す
```

である。第 2 引数に null を指定することで、gl.ARRAY_BUFFER に関する設定を終了させる。

4.3.2　VBO とシェーダの結合

関数 buildArrayBuffer() は VBO の設定を行うものだったが、bindArrayBuffer() は VBO をシェーダと結合させ、シェーダプログラム実行の準備を行う。

1 行目：

```
gl.bindBuffer(gl.ARRAY_BUFFER, arrayBuffer);       // オブジェクトを束縛
```

は、既に前節で述べたように、これから arrayBuffer について gl.ARRAY_BUFFER として設定を行うことの表明である。

次の行：

```
let location = gl.getAttribLocation(program, name);
                                  // バーテックスシェーダ内の name と同じ
                                  // 名前の attribute 変数の GPU 内の位置を取得
```

[4] CG で使う数値に 64bit の数値精度は必要ない。
[5] 興味のある人は、gl.mapBuffer()、gl.unmapBuffer()、gl.bufferSubData() などの関数を調べてみなさい。

は、第 1 引数に指定されたシェーダ実行可能コード program において attribute 宣言された変数の中で変数名が第 2 引数の文字列 name と同じものを探し、その位置（location）を変数 location へ代入する（図 4.4 (d) 参照)。location の具体的な値を知る必要はない。もしその名前の変数が見つからない場合には関数の戻り値は null となる（つまり実行時エラーである)。

すぐ後に解説する図 4.5（26 ページ）のバーテックスシェーダプログラムには

```
attribute vec2 position;                                    // 2 次元頂点位置
```

という宣言があり、 "position" という変数名を用いている。上の関数呼び出しはこの position の位置を求めている。

次の行：

```
gl.vertexAttribPointer(location, s, gl.FLOAT, false, 0, 0);
                                        // location の頂点バッファの属性を設定
```

は、次の二つの処理を行う（図 4.4 (e) 参照)。

1. 第 1 引数 location で指定するシェーダ実行可能コード中の arrribute 変数の位置と、現時点で設定対象となっている VBO（gl.bindBuffer() で指定したバッファオブジェクト）を結合する。これによって、シェーダプログラムが実行されるときには、VBO から順にデータが取り出され、それがそれぞれの arrribute 変数に代入されることになる（ここも地味に分かりにくい！)。

2. 第 2 引数から第 6 引数は、VBO の先頭から順にデータが取り出す方法を指定する。

   (a) 第 2 引数 s は、対応する arrribute 変数が s 個のコンポーネント（基本構成要素、第 3 引数を参照のこと）を持つことを意味する。

   (b) 第 3 引数 gl.FLOAT は、前出のコンポーネントが float 型であることを意味する。他に、GL_BYTE、GL_UNSIGNED_BYTE、GL_SHORT、 GL_UNSIGNED_SHORT、GL_INT、GL_UNSIGNED_INT などが指定可能である。

   (c) 第 4、第 5、第 6 引数の説明は省略する。高度に凝った処理を行わない限り、false, 0, 0 でよい。

最後の行：

```
gl.enableVertexAttribArray(location);
                                // location の頂点バッファを利用可能にする
```

は、第 1 引数 location で位置を指定した arrribute 変数を利用可能にする。

以上が VBO の設定である。

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<script src="./main.js" type="text/javascript"></script>
<script src="./util.js" type="text/javascript"></script>
<script src="./compileLink.js" type="text/javascript"></script>

<script id="vs" type="text/plain">
// ここにバーテックスシェーダソースプログラムを記述

    attribute vec2 position;                                    // 2次元頂点位置

    void main(void){
        gl_Position = vec4(position, 0.0, 1.0); // そのままラスタライザへ渡す
    }
</script>

<script id="fs" type="text/plain">
// ここにフラグメントシェーダソースプログラムを記述

    void main(void){
        gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0);                // 赤を出力
    }
</script>

</head>
<body>
<canvas id="canvas"></canvas>
</body>
</html>
```

図 4.5: ×印を描画する HTML/シェーダプログラム

4.4　シェーダプログラム

シェーダソースプログラムは JavaScript のプログラムにとっては単なる文字列である。そこでこのテキストではシェーダソースプログラムを以下のように取り扱う。

1. シェーダのソースプログラムを HTML ファイルに `script` 要素として埋め込み、
2. `script` 要素の `id` 属性をキーとしてシェーダプログラムを JavaScript にオブジェクトとして取り込み、
3. そのオブジェクトの `text` メンバーとしてシェーダプログラムを文字列として取得する。

上の方針で記述された HTML ファイルが図 4.5（26 ページ）である。このファイルの 4 行目の `script` 要素は図 4.2（14 ページ）の JavaScript プログラムである。このファイルの 5 行目の `script` 要素は図 4.3（20 ページ）の JavaScript プログラムである。このファイルの 6 行目の `script` 要

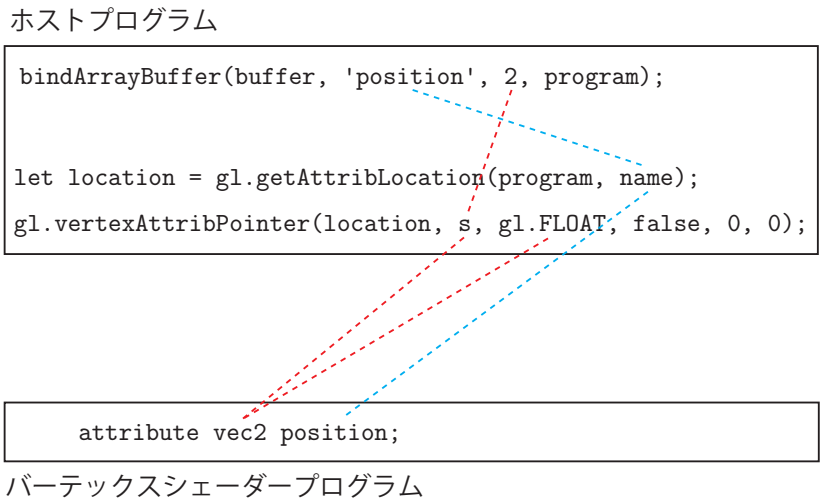


図 4.6: attribute 変数に関するホストプログラムとバーテックスシェーダプログラムの関係

素は図 4.8（33 ページ）の JavaScript プログラムであり、後節で解説する。これらのファイルは HTML ファイルと同じディレクトリに `.js` ファイルとして格納する。その下の `id="vs"`の `script` 要素がバーテックスシェーダソースプログラム、 `id="fs"`の `script` 要素がフラグメントシェーダソースプログラムである。これら `script` 要素の `type` 属性として `"text/plain"` を指定する。

現時点の最新の OpenGL/GLSL の仕様は ver. 4 であり、様々な新機能が盛り込まれている。しかし、このテキストの WebGL の仕様は OpenGL の ver. 2 相当であり、機能が低い。しかし、シェーダプログラミングの基礎を理解するにはそれで十分である。

以下、この節の例題のバーテックス/フラグメントシェーダプログラムについて概説するが、具体的なシェーダプログラムの実行の様子は 4.5 節（30 ページ）で述べる。ここでは、プログラムの概要を理解する程度とし、4.5 節で正確に理解する方がよい。

4.4.1　バーテックスシェーダプログラム

バーテックスシェーダにはホストプログラムから与えられる 4 個の頂点のそれぞれの処理内容を記述する。

図 4.5 のバーテックスシェーダプログラムの冒頭：

```
attribute vec2 position;   // 2 次元頂点位置
```

は、attribute 変数 position の宣言である。データ型 vec2 はベクトル型と呼ばれ、float 型 2 個からなることを宣言している。この宣言はホストプログラムのバッファの設定と整合していなければならない。その点について、先に解説された事項を思い出そう。ホストプログラム（図 4.3）の以下の行：

```
let location = gl.getAttribLocation(program, name);
                                  // バーテックスシェーダ内の name と同じ
                                  // 名前の attribute 変数の GPU 内の位置を取得

gl.vertexAttribPointer(location, s, gl.FLOAT, false, 0, 0);
                                     // location の頂点バッファの属性を設定
```

が、attribute 変数 position のデータを設定する関数であった。ここに変数 name には "position" が格納されており、s には 2 が格納されている。つまり、

1. 変数名が position であることは、gl.getAttribLocation(program, name) の第 2 引数が "position" であることと整合している。
2. 変数のデータ型が vec2 型であることは、gl.vertexAttribPointer(location, s, gl.FLOAT, false, 0, 0) の第 2、第 3 引数が s, gl.FLOAT であることと整合している。

この整合関係を図 4.6 にまとめた。

次に、バーテックスシェーダプログラムの main 関数は

```
gl_Position = vec4(position, 0.0, 1.0); // そのままラスタライザへ渡す
```

の 1 行のみからなる。これは、この頂点のキャンバス上の描画位置を設定するものである。左辺の gl_Position は、vec4 型のシステム定義変数であり、頂点のウインドウ上の座標位置を 4 次元同次座標系上の位置

$$(x, y, z, w)$$

として格納すると約束されている。この座標値は通常の 3 次元空間では

$$(X, Y, Z) = (\frac{x}{w}, \frac{y}{w}, \frac{z}{w})$$

と解釈される。描画位置はウインドウを水平方向 $[-1.0, 1.0]$、垂直方向 $[-1.0, 1.0]$ の矩形領域とするときの位置 (X, Y) である。Z の値は深さ値（正確には正規化された深さ値）である。$Z < -1.0$, $1.0 < Z$ の場合には描画されない。

上の式の右辺 vec4(position,0.0,1.0) は、

$$(x, y, z, w) = (\texttt{position.x},\ \texttt{position.y},\ 0.0,\ 1.0)$$

を意味する。つまり二つの数値 0.0, 1.0 が position.x, position.y の後ろに付く。これを通常の 3 次元空間の座標点に変換すると以下の通りである。

$$(X, Y, Z) = (\frac{\texttt{position.x}}{1.0},\ \frac{\texttt{position.y}}{1.0},\ \frac{0.0}{1.0}) = (\texttt{position.x},\ \texttt{position.y},\ 0.0)$$

よって

$$-1.0 \leq \texttt{position.x} \leq 1.0, \quad -1.0 \leq \texttt{position.y} \leq 1.0$$

ならば、この頂点はウインドウ上に描画される。

補足（ベクトル型データの合成）： GLSL のプログラムではベクトル型 vec2、vec3、textttvec4 を利用できる。それぞれ、float 型数値が 2 個、3 個、4 個からなる 2 次元、3 次元、4 次元ベクトルデータを表す。

ベクトル型の要素は、.x、.y、.z、.w で取り出すことができます。これはベクトルを座標値と解釈した場合である。ベクトルを RGBA データと解釈することもできて、その場合には、要素は .r、.g、.b、.a で取り出すこともできます。

ベクトル型は、全要素を連ねて記述し、たとえば

```
vec2 v2 = vec2(2.0, 3.0);
vec3 v3 = vec3(1.0, 2.0, 3.0);
vec4 v4 = vec4(1.0, 2.0, 3.0, 4.0);
```

とデータを作ることもできるが、低次のベクトル型データを用いて、高次のベクトル型データを作ることもできる。たとえば以下は上と同じ効果を持ち、しかもプログラムは読み易い。

```
vec2 v2 = vec2(2.0, 3.0);
vec3 v3 = vec3(1.0, v2);
vec4 v4 = vec4(v3, 4.0);
```

ベクトルの要素の入れ替えもできる。たとえば

```
vec4 w4 = vec4(v4.y, v4.z, v4.x, v4.w);
```

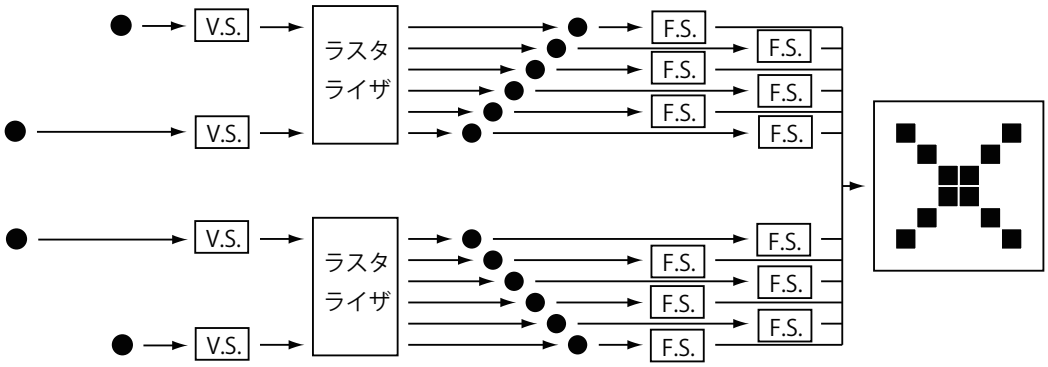


図 4.7: 線画を描く場合のシェーダの動作例

あるいは

```
vec4 w4 = vec4(v4.g, v4.b, v4.r, v4.a);
```

4.4.2　フラグメントシェーダプログラム

フラグメントシェーダではラスタライザが求めた各画素の色を決定する。

図 4.5 の `main` 関数は代入文：

```
gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0);                    // 赤を出力
```

のみからなる。左辺の `gl_FragColor` は `vec4` 型のシステム予約変数 であり、この変数にはその画素の RGB 輝度値とアルファ値（不透明度）を格納すると約束されている。代入文の右辺の値は、RGB 輝度値は赤色 $(1.0, 0.0, 0.0)$ であるから、線分は赤一色に塗られる（図 4.1 参照）。4 番目の不透明度は 1.0 であるから、完全に不透明であり、下地が透けて見えることはない。

4.5　プログラムの実行の様子

図 4.7 はここまで解説してきたプログラムの、特に GPU 内での実行の様子を図示したものである。

バーテックスシェーダの動き　この例題では、関数 `initData()` で 4 頂点の値を設定する。それらの各頂点がバーテックスシェーダプログラムの attribute 変数 `position` へ供給され、四つ

のバーテックスシェーダプログラムがそれぞれ**並列実行**される。バーテックスシェーダ間のデータのやり取りは一切なく、全く独立に並列実行される。

ラスタライザの動き　バーテックスシェーダで求められた gl_Position の座標値は、入力バッファと同じ順序で並べられ、その先頭から二つずつがペアにされて、ウインドウ上の線分の始点と終点と見なされる（gl.drawArrays() 関数の第一引数で gl.LINES を指定したため、そのペアを作る動作が起きる）。

頂点のペアはラスタライザで線分を構成する画素点にラスタライズされる。ペアは二つあるから、それぞれに対応するラスタライザが独立に**並列実行**される。

フラグメントシェーダの動き　ラスタライザから出力された各画素点についてフラグメントシェーダが独立して**並列実行**される。

フラグメントシェーダから gl_FragColor に出力された色情報は GPU のフレームバッファに集約されて、一枚の画像になる。

今回の例題では、2 本の線分がちょうど交わる画素点では重複して色情報がフレームバッファへ出力される。そのとき、その画素に実際に塗られる色は実行のタイミングに依存する[6]。

このように GPU を用いた描画処理は各シェーダ、ラスタライザで並列処理になっている。また、シェーダ、ラスタライザの処理は部分的に重なって実行されるパイプライン処理になっている。GPU では、並列処理とパイプライン処理によって高速描画を実現している、

[6] 3D CG の場合には、画素点の深さ情報によって実際に描画する色を選択する（視点から被写体までの距離の短い方の色を選択し、描画する）。

4.6　ホストプログラム：シェーダプログラムのコンパイル、リンク

順序が大幅に逆順になったが、この章の最後の話題はシェーダプログラムのコンパイルとリンクである。コンパイル、リンクを最後にした理由は、GPU の実行の理解を最優先にしたためである。関連プログラムは、図 4.8（33 ページ） である。

4.6.1　シェーダプログラムのコンパイル

シェーダソースプログラムをコンパイルする関数は、図 4.8 の下方　compileProgram() である。

引数の type には、それがバーテックスシェーダ用プログラムなのか（gl.VERTEX_SHADER）、フラグメントシェーダ用プログラムなのか（gl.FRAGMENT_SHADER）、の区別を指定する。同じく、id には、HTML ファイルの id 属性値を指定する。

関数本体の冒頭：

```
let source = document.getElementById(id);
                          // HTML ファイル中から id に相当するリソースを取得
if (!source) { return; }                  // 取得できなければ null を返す
```

では、リソースを取得し、取得確認を行う。次に、

```
let shader = gl.createShader(type);     // 空のオブジェクトコードを作成

gl.shaderSource(shader, source.text);          // ソースプログラムの文字
                                               // 列を shader に接続

gl.compileShader(shader);                                    // コンパイル
```

は、ソースプログラムをコンパイルし、オブジェクトコードを作る一連の作業である。ソースプログラムにコンパイルエラーがあったか否かを以下でチェックし、エラーが無ければ、オブジェクトコードを戻り値とする。エラーがある場合には、それを表示し、プログラマに知らせる。

```
if (gl.getShaderParameter(shader, gl.COMPILE_STATUS)){ // エラーチェック
    return shader;        // エラーが無いならばオブジェクトコードを戻す
} else {
    alert(gl.getShaderInfoLog(shader));             // エラー内容の表示
}
```

```
// html ファイル中の id 名が vsid、fsid のリソースを文字列として読み込んで
// シェーダ実行可能コードをビルド
function buildProgram(vsid, fsid) {
    let vs = compileProgram(gl.VERTEX_SHADER, vsid);
                        // バーテックスシェーダソースプログラムのコンパイル
    let fs = compileProgram(gl.FRAGMENT_SHADER, fsid);
                        // フラグメントシェーダソースプログラムのコンパイル

    program = gl.createProgram();               // 空の実行可能コードを作成

    gl.attachShader(program, vs);
                          // バーテックスシェーダのオブジェクトコードを接続
    gl.attachShader(program, fs);
                          // フラグメントシェーダのオブジェクトコードを接続

    gl.linkProgram(program);                                        // リンク

    if (!gl.getProgramParameter(program, gl.LINK_STATUS)){ // エラーチェック
        alert(gl.getProgramInfoLog(program));           // エラー内容の表示
    }
}

// シェーダのタイプ（バーテックス/フラグメント）とリソース id から
// オブジェクトコードを戻す
function compileProgram(type, id){
    let source = document.getElementById(id);
                            // HTML ファイル中から id に相当するリソースを取得
    if (!source) { return; }                // 取得できなければ null を返す

    let shader = gl.createShader(type);     // 空のオブジェクトコードを作成

    gl.shaderSource(shader, source.text);         // ソースプログラムの文字
                                                  // 列を shader に接続

    gl.compileShader(shader);                                       // コンパイル

    if (gl.getShaderParameter(shader, gl.COMPILE_STATUS)){ // エラーチェック
        return shader;        // エラーが無いならばオブジェクトコードを戻す
    } else {
        alert(gl.getShaderInfoLog(shader));             // エラー内容の表示
    }
}
```

図 4.8:　×印を描画する javaScript ホストプログラム（シェーダ実行可能コード作成関連、compileLink.js）

4.6.2　リンク

コンパイルされたオブジェクトコードはリンクされるが、それが図 4.8 の上方 buildProgram() である。

冒頭の 2 行：

```
    let vs = compileProgram(gl.VERTEX_SHADER, vsid);
                        // バーテックスシェーダソースプログラムのコンパイル
    let fs = compileProgram(gl.FRAGMENT_SHADER, fsid);
                        // フラグメントシェーダソースプログラムのコンパイル
```

は、バーテックス/フラグメントシェーダをそれぞれコンパイルし、オブジェクトコードを取得する部分である。コンパイルに成功したならば、以下のように二つのプログラムをリンクし、実行可能コードを作成する。

```
    program = gl.createProgram();                  // 空の実行可能コードを作成

    gl.attachShader(program, vs);
                          // バーテックスシェーダのオブジェクトコードを接続
    gl.attachShader(program, fs);
                          // フラグメントシェーダのオブジェクトコードを接続

    gl.linkProgram(program);                                          // リンク
```

実行可能コードはグローバル変数 program に格納される。リンクエラー（たとえば、バーテックスシェーダとフラグメントシェーダで受け渡すデータが整合しないなど）をチェックする部分が末尾にある。

4.7　補足：この章のプログラムのリスト

プログラム読解の便利のために、以下にこの章の最後で解説したプログラムのリストを載せる。

```
<!-- HTML -->
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<script src="./main.js" type="text/javascript"></script>
<script src="./util.js" type="text/javascript"></script>
<script src="./compileLink.js" type="text/javascript"></script>

<script id="vs" type="text/plain">
// ここにバーテックスシェーダソースプログラムを記述

    attribute vec2 position;                                    // 2次元頂点位置

    void main(void){
        gl_Position = vec4(position, 0.0, 1.0); // そのままラスタライザへ渡す
    }
</script>

<script id="fs" type="text/plain">
// ここにフラグメントシェーダソースプログラムを記述

    void main(void){
        gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0);                // 赤を出力
    }
</script>

</head>
<body>
<canvas id="canvas"></canvas>
</body>
</html>
```

```
// main.js
let gl;                                          // グラフィックスコンテキスト
let program;                                     // シェーダ実行可能コード

function initSystem() {                          // 描画システムの初期設定
    let c = document.getElementById('canvas');   // キャンバスを取得
    c.width = 500; c.height = 500;               // キャンバスのサイズを設定

    gl = c.getContext('webgl');     // WebGL 用グラフィックスコンテキストを取得

    gl.clearColor(0.5, 0.5, 0.5, 1.0);                    // 背景色を指定

    buildProgram('vs', 'fs');                // シェーダ実行可能コードのビルド
    gl.useProgram(program);         // ビルドしたプログラムを利用可能にする
}

let NUM_POINTS = 4;                                       // 頂点数の設定

function initData() {                            // 描画データの初期設定
    let position = [                           // 2 次元頂点データを配列に設定
        -0.5, -0.5,                                                 // 左下
        +0.5, +0.5,                                                 // 右上
        +0.5, -0.5,                                                 // 右下
        -0.5, +0.5                                                  // 左上
    ];

    let buffer = buildArrayBuffer(position);  // 配列から頂点バッファを生成

    bindArrayBuffer(buffer, 'position', 2);
                    // buffer を シェーダ内の attribute 変数 position と結合
}

function display() {                                          // 描画を行う
    gl.clear(gl.COLOR_BUFFER_BIT);             // キャンバスを背景色でクリア
    gl.drawArrays(gl.LINES, 0, NUM_POINTS);
                                // 頂点バッファの先頭（0）から NUM_POINTS-1
                                // までの頂点を利用して線画を描画

    gl.flush();                                       // GPU に描画処理を催促
}

window.onload = function(){                          // main 関数に相当する
    initSystem();                                   // 描画システムの初期化
    initData();                                      // 描画データの初期化
    display();                                                  // 描画
};
```

```
// util.js
function buildArrayBuffer(data) {          // 頂点バッファオブジェクトの作成
    let arrayBuffer = gl.createBuffer();           // 空のオブジェクトの作成

    gl.bindBuffer(gl.ARRAY_BUFFER, arrayBuffer);      // オブジェクトを束縛

    gl.bufferData(gl.ARRAY_BUFFER,                // 配列 data の内容を GPU へ
                  new Float32Array(data),         // コピーし、頂点バッファ
                  gl.STATIC_DRAW);                // オブジェクトを作成

    gl.bindBuffer(gl.ARRAY_BUFFER, null);                    //  束縛を外す

    return arrayBuffer;                          // 作成したオブジェクトを戻す
}

function bindArrayBuffer(arrayBuffer, name, s) {
    gl.bindBuffer(gl.ARRAY_BUFFER, arrayBuffer);      // オブジェクトを束縛

    let location = gl.getAttribLocation(program, name);
                                    // バーテックスシェーダ内の name と同じ
                                    // 名前の attribute 変数の GPU 内の位置を取得

    gl.vertexAttribPointer(location, s, gl.FLOAT, false, 0, 0);
                                       // location の頂点バッファの属性を設定

    gl.enableVertexAttribArray(location);
                                    // location の頂点バッファを利用可能にする
}
```

```
// compileLink.js

// html ファイル中の id 名が vsid、fsid のリソースを文字列として読み込んで
// シェーダ実行可能コードをビルド
function buildProgram(vsid, fsid) {
    let vs = compileProgram(gl.VERTEX_SHADER, vsid);
                        // バーテックスシェーダソースプログラムのコンパイル
    let fs = compileProgram(gl.FRAGMENT_SHADER, fsid);
                        // フラグメントシェーダソースプログラムのコンパイル

    program = gl.createProgram();               // 空の実行可能コードを作成

    gl.attachShader(program, vs);
                         // バーテックスシェーダのオブジェクトコードを接続
    gl.attachShader(program, fs);
                         // フラグメントシェーダのオブジェクトコードを接続

    gl.linkProgram(program);                                    // リンク

    if (!gl.getProgramParameter(program, gl.LINK_STATUS)){ // エラーチェック
        alert(gl.getProgramInfoLog(program));           // エラー内容の表示
    }
}

// シェーダのタイプ（バーテックス/フラグメント）とリソース id から
// オブジェクトコードを戻す
function compileProgram(type, id){
    let source = document.getElementById(id);
                           // HTML ファイル中から id に相当するリソースを取得
    if (!source) { return; }                // 取得できなければ null を返す

    let shader = gl.createShader(type);         // 空のオブジェクトコードを作成

    gl.shaderSource(shader, source.text);          // ソースプログラムの文字
                                                   // 列を shader に接続

    gl.compileShader(shader);                                   // コンパイル

    if (gl.getShaderParameter(shader, gl.COMPILE_STATUS)){ // エラーチェック
        return shader;        // エラーが無いならばオブジェクトコードを戻す
    } else {
        alert(gl.getShaderInfoLog(shader));             // エラー内容の表示
    }
}
```

第5章　線形補間

この章ではラスタライザによる線形補間機能を解説する。

5.1　輝度の線形補間

バーテックスシェーダプログラムに宣言する attribute 変数は、ホストプログラムで準備したデータをシェーダプログラムに接続するインターフェースになっている。前章の例では attribute 変数として `position` を用い、そこに配列バッファから頂点の 2 次元座標位置を供給した。この節では、各頂点に輝度値を追加し、ラスタライザで線分内の各画素の輝度を滑らかに変化させる。この際、バーテックスシェーダの出力をラスタライザを経由してフラグメントシェーダの入力とする仕組み：**varying 変数**を新たに用いる。

この節の画像の出力例は図 5.1 である。

なお、ここでいう線形補間とは、線分の両端点 p_1、p_2 の輝度値を B_1、B_2 とするとき、線分上の画素点 p の輝度値 B を

$$B = \frac{|p_2 - p|}{|p_2 - p_1|} B_1 + \frac{|p - p_1|}{|p_2 - p_1|} B_2 \tag{5.1}$$

と求めることをいう。この方法で計算の手間を極力省き、輝度値にグラデーションを付ける。GPU ではこの線形補間をハードウェアで計算するため、きわめて高速な処理が可能である。

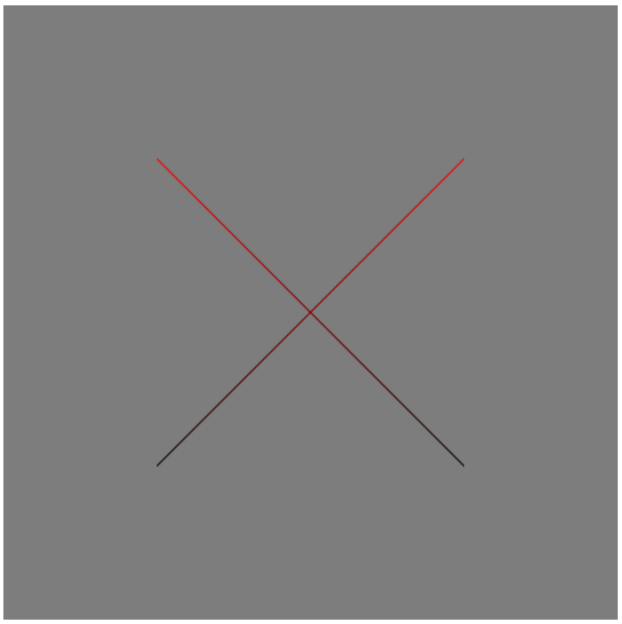


図 5.1: 画像例（その 2、輝度値の線形補間）

```
// main.js
let gl;
let program;

function initSystem() {
    ** 前章と同じ **
}

let NUM_POINTS = 4;

function initData() {                                   // 描画データの初期設定
    let position = [
        -0.5, -0.5,                                                 // 左下
        +0.5, +0.5,                                                 // 右上
        +0.5, -0.5,                                                 // 右下
        -0.5, +0.5                                                  // 左上
    ];

    let buffer1 = buildArrayBuffer(position);

    bindArrayBuffer(buffer1, 'position', 2);

    let brightness = [                                  // 輝度値を新規追加
        0.0,                                                        // 左下
        1.0,                                                        // 右上
        0.0,                                                        // 右下
        1.0                                                         // 左上
    ];

    let buffer2 = buildArrayBuffer(brightness);  // 配列から頂点バッファを生成

    bindArrayBuffer(buffer2, 'in_brightness', 1);
            // buffer2 を シェーダ内の attribute 変数 in_brightness と結合
}

function display() {                                                // 描画を行う
    ** 前章と同じ **
}

window.onload = function(){                             // main 関数に相当する
    ** 前章と同じ **
};
```

図 5.2: 輝度値が滑らかに変化する×印を描画する JavaScript ホストプログラム main.js（その 1、その 2 へ続く）

5.1.1　ホストプログラム

ホストプログラムの変更は図 5.2 の関数 `initData()` のみである。新たに配列 `brightness` を用意し、そこに 0.0（最低輝度＝真っ暗） と 1.0（最大輝度）を格納する。

```
    let brightness = [                                      // 輝度値を新規追加
        0.0,                                                        // 左下
        1.0,                                                        // 右上
        0.0,                                                        // 右下
        1.0                                                         // 左上
    ];
```

なお、WebGL では輝度値は $[0, 255]$ の範囲ではなく、$[0.0, 1.0]$ の範囲であることを注意する。

そして、以下のように、この配列について VBO（バーテックスバッファオブジェクト） を作り、それをシェーダと結合する。文字列 “in_brightness”は、この配列に対応するバーテックスシェーダプログラムの attribute 変数の名前である。

```
    let buffer2 = buildArrayBuffer(brightness);  // 配列から頂点バッファを生成

    bindArrayBuffer(buffer2, 'in_brightness', 1);
              // buffer2 を シェーダ内の attribute 変数 in_brightness と結合
```

5.1.2　バーテックスシェーダプログラム

図 5.3 の上方が変更後のバーテックスシェーダプログラムである。主な変更点は以下の箇所である。

```
    attribute float in_brightness;                    // attribute 変数の追加
    ...
    varying float out_brightness;                // varying 変数（出力）の追加
    ...

        out_brightness = in_brightness;                // 輝度値を単純コピー
```

まず、プログラムには輝度値を格納する attribute 変数 `in_brightness` の宣言を追加する。さらにバーテックスシェーダプログラムで計算した輝度値をラスタライザを介してフラグメントシェーダプログラムへ受け渡す変数 `out_brightness` を宣言する。バーテックスシェーダからの出力に対応する変数は varying 変数と呼ばれ、変数宣言の前にキーワード `varying` を付けて宣言する。`main`

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<script src="./main.js" type="text/javascript"></script>
<script src="./util.js" type="text/javascript"></script>
<script src="./compileLink.js" type="text/javascript"></script>

<script id="vs" type="text/plain">
    attribute vec2 position;
    attribute float in_brightness;                    // attribute 変数の追加

    varying float out_brightness;               // varying 変数（出力）の追加

    void main(void)
    {
        gl_Position = vec4(position, 0.0, 1.0);

        out_brightness = in_brightness;               // 輝度値を単純コピー
    }
</script>

<script id="fs" type="text/plain">
    precision highp float;                         // float を高精度として宣言

    varying float out_brightness;               // varying 変数（入力）の追加

    void main(void)
    {
        gl_FragColor = vec4(out_brightness, 0.0, 0.0, 1.0);
                                       // 赤の輝度値を out_brightness に変更
    }
</script>


</head>
<body>
<canvas id="canvas"></canvas>
</body>
</html>
```

図 5.3: 輝度値が滑らかに変化する×印を描画する HTML/シェーダプログラム

関数では単に in_brightness の値を out_brightness に代入するだけである。単なる代入であるが、これは省略できない。単なる代入ではない例として、たとえば以下の代入文を用いれば輝度値が反転することができる。

```
out_brightness = 1.0-in_brightness;
```

5.1.3　フラグメントシェーダプログラム

図 5.3 の下方が変更後のフラグメントシェーダプログラムである。

まず、新規に記載される以下：

```
precision highp float;                          // float を高精度として宣言
```

は、フラグメントシェーダにおける float の数値計算の最低精度を指定するものである。highp は、high precision（高精度）の計算（32bit 演算）を意味する。他に mediump ＝中精度（16bit 演算）、lowp ＝低精度（10bit 固定小数点演算）が指定可能である。指定精度が低い方が演算速度は大きくなると期待できるが、実際には中精度を指定してもシステムが高精度演算を行う場合もあり、指定精度.vs. 速度の関係はそれほど単純ではない。ここでは高画質を優先する観点から highp を指定する。

プログラムにはバーテックスシェーダプログラムと同じ名前の varying 変数の宣言：

```
varying float out_brightness;                   // varying 変数（入力）の追加
```

を加えねばならない（さもなくばリンクエラーとなる）。これによって、バーテックスシェーダプログラムから出力された out_brightness の値がラスタライザを介して線形補間され、フラグメントシェーダプログラムの varying 変数へ格納される。

バーテックスシェーダプログラムの out_brightness は線分の両端点の値（0.0 または 1.0）であるが、フラグメントシェーダプログラムのそれは線形補間された各画素の値になる。ラスタライザが、out_brightness の両端点の値、その両端点の画素位置、それ以外の各画素の位置の情報から、各画素における out_brightness の値を自動的に線形補間する。この補間には GPU の補間専用ハードウェアを用いるため、高速な補間ができる。

main 関数では、線形補間された out_brightness を赤色の輝度値として RGB カラーを作っている。

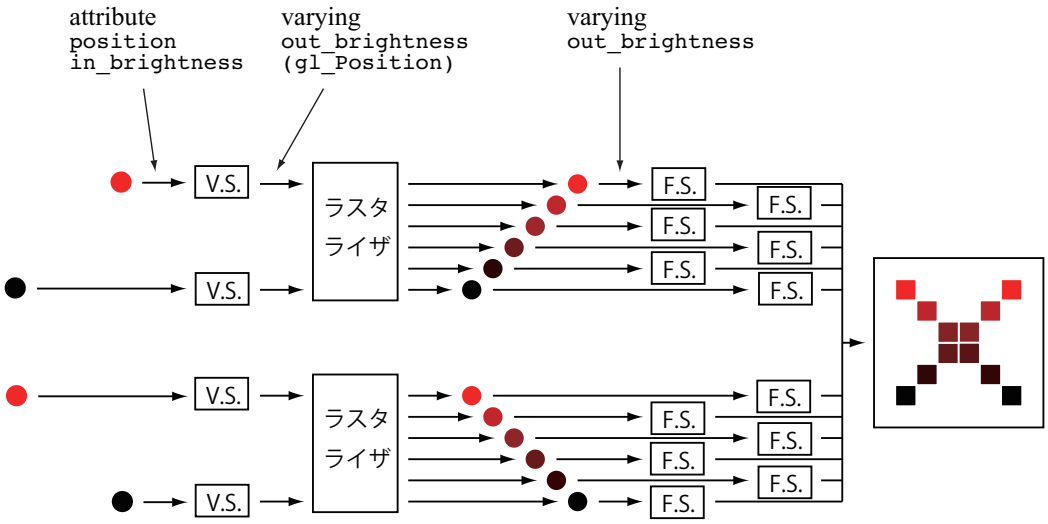


図 5.4: 輝度の線形補間の動作例

5.1.4　実行の様子

図 5.4 に GPU の実行の様子を示す。

バーテックスシェーダプログラムに入力される時点で頂点に輝度値が追加されている。この例では輝度値はそのままラスタライザに渡される。ラスタライザは、両端点の情報から線分内の各画素の情報を線形補間する。補間された値はフラグメントシェーダプログラムに入力される。それの値を元に RGB カラーを作れば、図 5.1 のような画像が生成される。

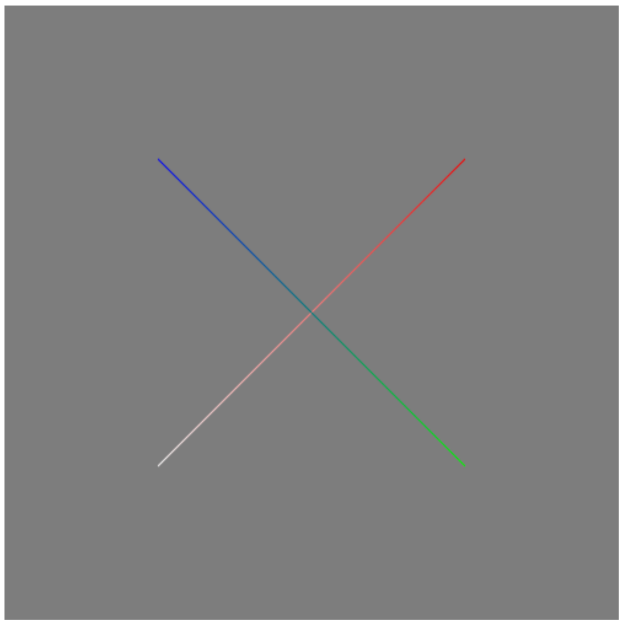


図 5.5: 画像例（その 3、RGB カラーの線形補間）

5.2　RGB カラーの線形補間

この節は前節の応用である。

前節の例では `float` 型の輝度値を attribute 変数として GPU へ渡した。その attribute 変数に格納できるデータのサイズは、2 個、3 個、4 個の `float` 型の並びに変更することもできる。この節では、頂点の RGB カラー値を attribute 変数に格納してみる。

画像の出力例は図 5.5 である。両端点の RGB カラーが線形補間され、色がグラデーションをなしている。

なお、ここでいう線形補間とは、線分の両端点 p_1、p_2 の RGB カラー値を $\vec{C}_1$、$\vec{C}_2$ という 3 次元ベクトル値とするとき、画素点 p の RGB カラー値 $\vec{C}$ を

$$\vec{C} = \frac{|p_2 - p|}{|p_2 - p_1|}\vec{C}_1 + \frac{|p - p_1|}{|p_2 - p_1|}\vec{C}_2 \tag{5.2}$$

とベクトル計算で求めることをいう。繰り返しになるが、GPU では線形補間をハードウェアで実行するため、きわめて高速な処理が可能である。

5.2.1　ホストプログラム

図 5.6（47 ページ）の関数 `initData()` には配列 `color` を用意し、そこに 白、赤、緑、青の RGB カラーをこの順番に格納した。

```
// main.js
let gl;
let program;

function initSystem() {
    ** 前章と同じ **
}

let NUM_POINTS = 4;

function initData() {                                   // 描画データの初期設定
    let position = [
        -0.5, -0.5,                                                     // 左下
        +0.5, +0.5,                                                     // 右上
        +0.5, -0.5,                                                     // 右下
        -0.5, +0.5                                                      // 左上
    ];

    let buffer1 = buildArrayBuffer(position);

    bindArrayBuffer(buffer1, 'position', 2);

    let color = [                                           // RGB 値を新規追加
        1.0, 1.0, 1.0,                                              // 左下、白
        1.0, 0.0, 0.0,                                              // 右上、赤
        0.0, 1.0, 0.0,                                              // 右下、緑
        0.0, 0.0, 1.0                                               // 左上、青
    ];

    let buffer2 = buildArrayBuffer(color);    // 配列から頂点バッファを生成

    bindArrayBuffer(buffer2, 'in_color', 3);
                    // buffer2 を シェーダ内の attribute 変数 in_color と結合
}

function display() {
    ** 前章と同じ **
}

window.onload = function(){
    ** 前章と同じ **
};
```

図 5.6: RGB カラーが滑らかに変化する×印を描画する JavaScript ホストプログラム main.js

```
let color = [                                          // RGB 値を新規追加
    1.0, 1.0, 1.0,                                          // 左下、白
    1.0, 0.0, 0.0,                                          // 右上、赤
    0.0, 1.0, 0.0,                                          // 右下、緑
    0.0, 0.0, 1.0                                           // 左上、青
];
```

そして、その配列について ArrayBuffer オブジェクトを作り、それをシェーダと結合する。以下：

```
bindArrayBuffer(buffer2, 'in_color', 3);
                // buffer2 を シェーダ内の attribute 変数 in_color と結合
```

の第 2 引数値 “in_color” は、この配列に対応するバーテックスシェーダプログラムの attribute 変数の名前である。第 3 引数値 3 は、RGB 値は三つの float 型データからなるためである。

5.2.2　バーテックスシェーダプログラム

図 5.7（49 ページ）が変更後のシェーダプログラムである。

図の上方、バーテックスシェーダプログラムには輝度値を格納する attribute 変数 in_color を宣言する。前節の in_brightness は float 型であったが、in_color は float 型データ三つからなるため、vec3 型である。

さらに バーテックスシェーダプログラムで計算した RGB カラー値をラスタライザを介してフラグメントシェーダプログラムへ受け渡す varying 変数 out_color を宣言する。この辺の仕組みは前節と同じである。

5.2.3　フラグメントシェーダプログラム

図 5.7 の下方、フラグメントシェーダプログラムにはバーテックスシェーダプログラムと同じ varying 変数の宣言を加える。main 関数では、ラスタライザで線形補間された out_color を画素の色として出力する。なお、out_color は vec3 型であり、gl_FragColor は vec4 型であるから、データを整合させるために、不足分のアルファ値（不透明度） 1.0 を追加した。

5.2.4　実行の様子

図 5.8 に示すように、バーテックスシェーダプログラムに入力される時点で頂点に RGB カラー値が追加されている。この例では、RGB カラー値をそのままラスタライザに渡す。ラスタライザ

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<script src="./main.js" type="text/javascript"></script>
<script src="./util.js" type="text/javascript"></script>
<script src="./compileLink.js" type="text/javascript"></script>

<script id="vs" type="text/plain">
    attribute vec2 position;
    attribute vec3 in_color;                          // attribute 変数の変更

    varying vec3 out_color;                     // varying 変数（出力）の変更


    void main(void) {
        gl_Position = vec4(position, 0.0, 1.0);

        out_color = in_color;                           // RGB 値を単純コピー
    }
</script>

<script id="fs" type="text/plain">
    precision highp float;

    varying vec3 out_color;                     // varying 変数（入力）の変更

    void main(void) {
        gl_FragColor = vec4(out_color, 1.0);
                                                // RBB 値を out_color から取得
    }
</script>


</head>
<body>
<canvas id="canvas"></canvas>
</body>
</html>
```

図 5.7: RGB カラーが滑らかに変化する×印を描画する HTML/シェーダプログラム

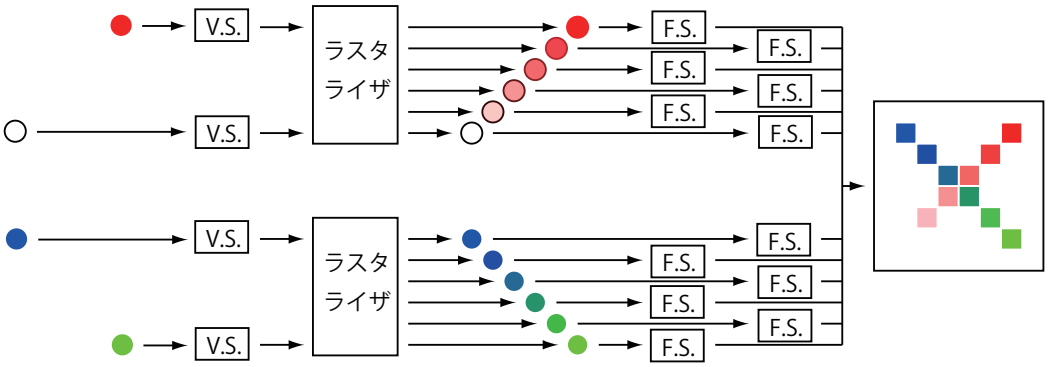


図 5.8: 輝度の線形補間の動作例

は渡された RGB カラー値を線分の両端点の属性値と見なされ、その値が線分の各画素について線形補間される。補間された値はフラグメントシェーダプログラムに入力される。

5.3　補足：この章のプログラムのリスト

プログラム読解の便利のために、以下にこの章の最後で解説したプログラムのリストを載せる。

```
<!-- HTML -->
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<script src="./main.js" type="text/javascript"></script>
<script src="./util.js" type="text/javascript"></script>
<script src="./compileLink.js" type="text/javascript"></script>

<script id="vs" type="text/plain">
    attribute vec2 position;
    attribute vec3 in_color;                          // attribute 変数の変更

    varying vec3 out_color;                     // varying 変数（出力）の変更


    void main(void) {
        gl_Position = vec4(position, 0.0, 1.0);

        out_color = in_color;                          // RGB 値を単純コピー
    }
</script>

<script id="fs" type="text/plain">
    precision highp float;

    varying vec3 out_color;                     // varying 変数（入力）の変更

    void main(void) {
        gl_FragColor = vec4(out_color, 1.0);
                                                // RBB 値を out_color から取得
    }
</script>


</head>
<body>
<canvas id="canvas"></canvas>
</body>
</html>
```

```
// main.js
let gl;
let program;

function initSystem() {
    ** 前章と同じ **
}

let NUM_POINTS = 4;

function initData() {                                   // 描画データの初期設定
    let position = [
        -0.5, -0.5,                                                 // 左下
        +0.5, +0.5,                                                 // 右上
        +0.5, -0.5,                                                 // 右下
        -0.5, +0.5                                                  // 左上
    ];

    let buffer1 = buildArrayBuffer(position);

    bindArrayBuffer(buffer1, 'position', 2);

    let color = [                                           // RGB 値を新規追加
        1.0, 1.0, 1.0,                                          // 左下、白
        1.0, 0.0, 0.0,                                          // 右上、赤
        0.0, 1.0, 0.0,                                          // 右下、緑
        0.0, 0.0, 1.0                                           // 左上、青
    ];

    let buffer2 = buildArrayBuffer(color);    // 配列から頂点バッファを生成

    bindArrayBuffer(buffer2, 'in_color', 3);
                    // buffer2 を シェーダ内の attribute 変数 in_color と結合
}

function display() {
    ** 前章と同じ **
}

window.onload = function(){
    ** 前章と同じ **
};
```

```
// util.js  前章から変更なし
function buildArrayBuffer(data) {          // 頂点バッファオブジェクトの作成
    let arrayBuffer = gl.createBuffer();           // 空のオブジェクトの作成

    gl.bindBuffer(gl.ARRAY_BUFFER, arrayBuffer);      // オブジェクトを束縛

    gl.bufferData(gl.ARRAY_BUFFER,                // 配列 data の内容を GPU へ
                  new Float32Array(data),         // コピーし、頂点バッファ
                  gl.STATIC_DRAW);                // オブジェクトを作成

    gl.bindBuffer(gl.ARRAY_BUFFER, null);                    //  束縛を外す

    return arrayBuffer;                          // 作成したオブジェクトを戻す
}

function bindArrayBuffer(arrayBuffer, name, s) {
    gl.bindBuffer(gl.ARRAY_BUFFER, arrayBuffer);      // オブジェクトを束縛

    let location = gl.getAttribLocation(program, name);
                                    // バーテックスシェーダ内の name と同じ
                                    // 名前の attribute 変数の GPU 内の位置を取得

    gl.vertexAttribPointer(location, s, gl.FLOAT, false, 0, 0);
                                       // location の頂点バッファの属性を設定

    gl.enableVertexAttribArray(location);
                                    // location の頂点バッファを利用可能にする
}
```

```
// compileLink.js　前章から変更なし

// html ファイル中の id 名が vsid、fsid のリソースを文字列として読み込んで
// シェーダ実行可能コードをビルド
function buildProgram(vsid, fsid) {
    let vs = compileProgram(gl.VERTEX_SHADER, vsid);
                        // バーテックスシェーダソースプログラムのコンパイル
    let fs = compileProgram(gl.FRAGMENT_SHADER, fsid);
                        // フラグメントシェーダソースプログラムのコンパイル

    program = gl.createProgram();               // 空の実行可能コードを作成

    gl.attachShader(program, vs);
                         // バーテックスシェーダのオブジェクトコードを接続
    gl.attachShader(program, fs);
                         // フラグメントシェーダのオブジェクトコードを接続

    gl.linkProgram(program);                                    // リンク

    if (!gl.getProgramParameter(program, gl.LINK_STATUS)){ // エラーチェック
        alert(gl.getProgramInfoLog(program));           // エラー内容の表示
    }
}

// シェーダのタイプ（バーテックス/フラグメント）とリソース id から
// オブジェクトコードを戻す
function compileProgram(type, id){
    let source = document.getElementById(id);
                           // HTML ファイル中から id に相当するリソースを取得
    if (!source) { return; }                // 取得できなければ null を返す

    let shader = gl.createShader(type);         // 空のオブジェクトコードを作成

    gl.shaderSource(shader, source.text);          // ソースプログラムの文字
                                                   // 列を shader に接続

    gl.compileShader(shader);                                   // コンパイル

    if (gl.getShaderParameter(shader, gl.COMPILE_STATUS)){ // エラーチェック
        return shader;        // エラーが無いならばオブジェクトコードを戻す
    } else {
        alert(gl.getShaderInfoLog(shader));             // エラー内容の表示
    }
}
```

第6章　ポイントスプライトの描画

この節では、線分ではなく、ポイントスプライト（point sprite）を描画する。

ポイントスプライトは、CPU の演算・描画能力が十分でない 1980 年代から 90 年代の PC やゲーム機において、ディスプレイ画面上を素早く動き回るゲームのキャラクタをスムーズに描画するために開発された手法である。素早く動く小さなキャラクタを背景の静止画の上にオーバーラップさせるのだが、オーバーラップはメモリ上の演算で行うのではなく、映像信号を出力するときにハードウェアによる信号合成によって行うため、コンピュータへの負荷は無く、当時のコンピュータの演算・描画能力不足をカモフラージュできた。“sprite”は妖精、小鬼を意味し、コンピュータ画面の上をせわしなく動き回るキャラクタという意味合いがある。

現在の WebGL（= OpenGL）のポイントスプライトの描画では信号合成によるオーバラップではなく、GPU のフレームメモリ上にスプライトの画像を直接描画するが、現在の GPU の描画能力ならば、演算によっても十分に高速な描画が可能である。

6.1　簡単なポイントスプライト描画

線分の描画では両端点の 2 次元座標が必要であった。それに対して、ポイントスプライトの描画に必要な情報はそのスプライトの中心点の 2 次元座標とスプライトの大きさである。そのことを踏まえ、図 6.1 の画像を生成するプログラムを紹介する。前章からの変更はたった二箇所である。

6.1.1　ホストプログラム

図 6.2 がこの節のホストプログラムである。変更箇所は描画関数 `display()` の 1 行のみである。

まず、以下：

```
gl.drawArrays(gl.LINES, 0, NUM_POINTS);
```

は、前章の GPU を起動する関数呼び出しであった。これをこの節では、

```
gl.drawArrays(gl.POINTS, 0, NUM_POINTS);              // gl.POINTS に変更
```

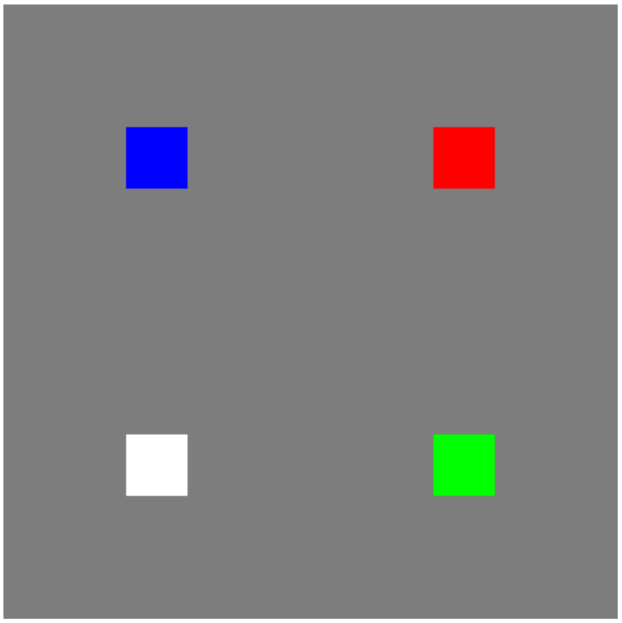


図 6.1: 画像例（その 4、ポイントスプライト）

と変更する。前章での第 1 引数の gl.LINES は始点、終点を与えて線分を描画するモードを意味したが、この節の gl.POINTS は各頂点を中心とするポイントスプライトを描画するモードを意味する。

JavaScript プログラムの変更は上の一箇所である。

6.1.2　バーテックスシェーダプログラム

もう一箇所の変更はバーテックスシェーダプログラムにある。図 6.3 がこの節のシェーダプログラムである。変更箇所は以下の代入文の追加である。

```
gl_PointSize = 50.0;                    // ポイントスプライトの大きさの指定
```

左辺の gl_PointoSize は、float 型のシステム定義変数であり、ここにポイントスプライトの一辺の長さを代入するものと約束されている。長さの単位は画素である。この代入処理を行わない場合、暗黙に 1 画素の長さと見なされ、ディスプレイ上でほとんど見えない。

6.1.3　フラグメントシェーダプログラム

フラグメントシェーダプログラムは前章から変更しない。

```
// main.js
let gl;
let program;

function initSystem() {
    ** 前章と同じ **
}

let NUM_POINTS = 4;

function initData() {
    ** 前章と同じ **
}

function display() {
    gl.clear(gl.COLOR_BUFFER_BIT);
    gl.drawArrays(gl.POINTS, 0, NUM_POINTS);              // gl.POINTSに変更
    gl.flush();
}

window.onload = function(){
    ** 前章と同じ **
};
```

図 6.2: 正方形のポイントスプライトを描画する JavaScript ホストプログラム `main.js`

6.1.4　実行の様子

図 6.4 に実行の様子を図示する。バーテックスシェーダプログラムからラスタライザへポイントサイズ N が渡されると、ラスタライザは頂点 (x, y) の周りに $N \times N$ 画素の正方形領域を計算し、その中の個々の画素点をフラグメントシェーダへ渡す。

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<script src="./main.js" type="text/javascript"></script>
<script src="./util.js" type="text/javascript"></script>
<script src="./compileLink.js" type="text/javascript"></script>

<script id="vs" type="text/plain">
    attribute vec2 position;
    attribute vec3 in_color;

    varying vec3 out_color;


    void main(void) {
        gl_Position = vec4(position, 0.0, 1.0);
        out_color = in_color;
        gl_PointSize = 50.0;            // ポイントスプライトの大きさの指定
    }
</script>

<script id="fs" type="text/plain">
    precision highp float;
    varying vec3 out_color;

    void main(void) {
        gl_FragColor = vec4(out_color, 1.0);
    }
</script>


</head>
<body>
<canvas id="canvas"></canvas>
</body>
</html>
```

図 6.3: 正方形のポイントスプライトを描画する HTML/シェーダプログラム

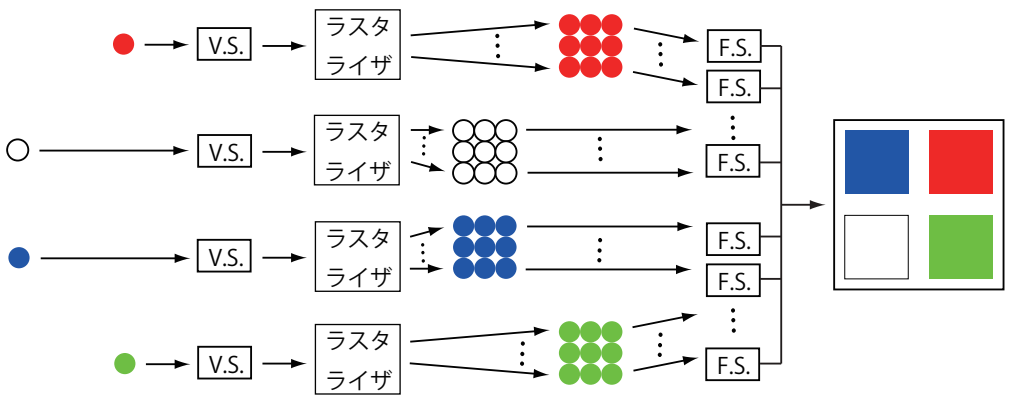


図 6.4: ポイントスプライトの動作例

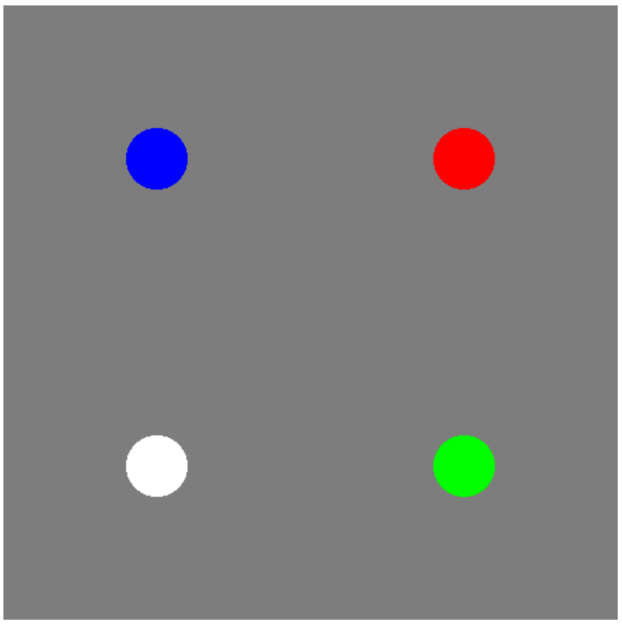


図 6.5: 画像例（その 5、ポイントスプライトが丸い場合）

6.2　円形のポイントスプライト

ポイントスプライトの形状はプログラムで自由に変更できる。ここでは正方形ではなく、図 6.5 のような円形を描画する方法を紹介する。

6.2.1　ホストプログラム

前節から変更しない。

6.2.2　バーテックスシェーダプログラム

前節から変更しない。

6.2.3　フラグメントシェーダプログラム

図 6.6 が変更後のフラグメントシェーダプログラムである。

このプログラム中の `gl_PointCoord` はフラグメントシェーダプログラムの `vec2` 型の予約変数である。ポイントスプライト描画時には、各フラグメントシェーダは `gl_PointCoord` によってポイントスプライト上の自身の画素の位置を知ることができるようになっている。

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<script src="./main.js" type="text/javascript"></script>
<script src="./util.js" type="text/javascript"></script>
<script src="./compileLink.js" type="text/javascript"></script>

<script id="vs" type="text/plain">
    attribute vec2 position;
    attribute vec3 in_color;

    varying vec3 out_color;


    void main(void) {
        gl_Position = vec4(position, 0.0, 1.0);
        out_color = in_color;
        gl_PointSize = 50.0;
    }
</script>

<script id="fs" type="text/plain">
    precision highp float;
    varying vec3 out_color;

    void main(void) {
        vec2 localPos = 2.0*gl_PointCoord-vec2(1.0,1.0);       // 座標変換
        if(length(localPos) < 1.0){    // もし原点からの距離が 1 未満ならば
            gl_FragColor = vec4(out_color, 1.0);       // この画素に色を塗る
        }else{
            discard;                                   // 画素に色を塗らない
        }
    }
</script>

</head>
<body>
<canvas id="canvas"></canvas>
</body>
</html>
```

図 6.6: 円形のポイントスプライトを描画する HTML/シェーダプログラム

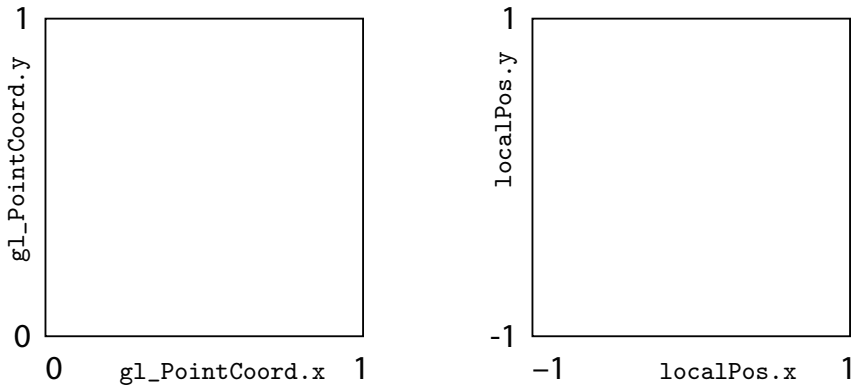


図 6.7: ポイントスプライトの正方形領域と gl_PointCoord、localPos の座標値の関係

前節で見たように、ラスタライザはバーテックスシェーダプログラムで指定した画素サイズの正方形内の各画素を求め、各画素についてフラグメントシェーダを起動する。各画素の位置は、図 6.7 の左図のように、$[0,1]\times[0,1]$ の範囲の値としてラスタライザによって gl_PointCoord に自動的に設定される。そこで、図 6.6 のプログラムでは、まず、

```
vec2 localPos = 2.0*gl_PointCoord-vec2(1.0,1.0);       // 座標変換
```

によって、その座標値を $[0,1]\times[0,1]$ の領域から $[-1,1]\times[-1,1]$ へ変更し、vec2 型変数 localPos へ計算する（図 6.7 の右図を参照）。たとえば gl_PointCoord =(0,0) ならば、localPos $=(-1,-1)$ であり、gl_PointCoord =(0.5,1) ならば、localPos $=(0,1)$ となる。

次に、以下：

```
if(length(localPos) < 1.0){     // もし原点からの距離が 1 未満ならば
```

は、localPos の大きさ length(localPos)（$=$ |localPos| $=$ sqrt(localPos.x*localPos.x+localPos.y*localPos.y)）が 1.0 よりも小さいか否か（すなわち半径 1 の円の内側か否か）の判定である。なお、length() はベクトルの大きさを求める GLSL の組み込み関数である（引数の型は vec2、vec3、vec4 のいずれも可）。

もし判定が真ならば、

```
gl_FragColor = vec4(out_color, 1.0);       // この画素に色を塗る
```

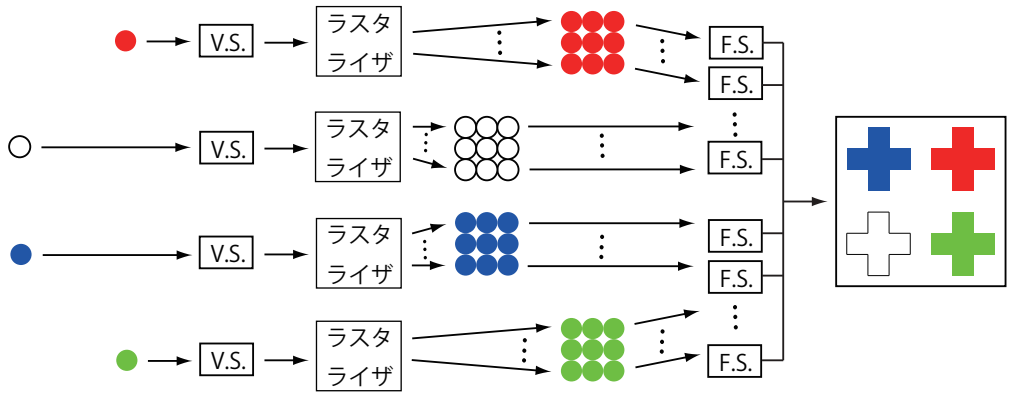


図 6.8: ポイントスプライトの動作例

によって、out_color の色をその画素に描画する。

　さもなくば描画しない。ここに「描画しない」とは、黒色を描画することではなく、フレームバッファへ色情報を書き込まないことを意味する。else 部の

```
discard;                                    // 画素に色を塗らない
```

は、計算結果を無視（discard）することを意味するフラグメントシェーダプログラムの予約語である。

6.2.4　実行の様子

　図 6.8 が実行の様子である。図 6.4 との違いは、フラグメントシェーダの値をフレームバッファに書き込まない場合があることである。

6.3　補足：この章のプログラムのリスト

プログラム読解の便利のために、以下にこの章の最後で解説したプログラムのリストを載せる。

```
<!-- HTML -->
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<script src="./main.js" type="text/javascript"></script>
<script src="./util.js" type="text/javascript"></script>
<script src="./compileLink.js" type="text/javascript"></script>

<script id="vs" type="text/plain">
    attribute vec2 position;
    attribute vec3 in_color;

    varying vec3 out_color;


    void main(void) {
        gl_Position = vec4(position, 0.0, 1.0);
        out_color = in_color;
        gl_PointSize = 50.0;
    }
</script>

<script id="fs" type="text/plain">
    precision highp float;
    varying vec3 out_color;

    void main(void) {
        vec2 localPos = 2.0*gl_PointCoord-vec2(1.0,1.0);       // 座標変換
        if(length(localPos) < 1.0){    // もし原点からの距離が 1 未満ならば
            gl_FragColor = vec4(out_color, 1.0);       // この画素に色を塗る
        }else{
            discard;                                   // 画素に色を塗らない
        }
    }
</script>

</head>
<body>
<canvas id="canvas"></canvas>
</body>
</html>
```

```
// main.js
let gl;
let program;

function initSystem() {
    let c = document.getElementById('canvas');
    c.width = 500; c.height = 500;

    gl = c.getContext('webgl');

    gl.clearColor(0.5, 0.5, 0.5, 1.0);

    buildProgram('vs', 'fs');
    gl.useProgram(program);
}

let NUM_POINTS = 4;

function initData() {
    let position = [
        -0.5, -0.5,                                          // 左下
        +0.5, +0.5,                                          // 右上
        +0.5, -0.5,                                          // 右下
        -0.5, +0.5                                           // 左上
    ];

    let buffer1 = buildArrayBuffer(position);

    bindArrayBuffer(buffer1, 'position', 2);

    let color = [
        1.0, 1.0, 1.0,                                       // 左下
        1.0, 0.0, 0.0,                                       // 右上
        0.0, 1.0, 0.0,                                       // 右下
        0.0, 0.0, 1.0                                        // 左上
    ];

    let buffer2 = buildArrayBuffer(color);

    bindArrayBuffer(buffer2, 'in_color', 3);
}

function display() {
    gl.clear(gl.COLOR_BUFFER_BIT);
    gl.drawArrays(gl.POINTS, 0, NUM_POINTS);             // gl.POINTS に変更
    gl.flush();
}

window.onload = function(){
    initSystem();
    initData();
    display();
};
```

```
// util.js  前章から変更なし
function buildArrayBuffer(data) {          // 頂点バッファオブジェクトの作成
    let arrayBuffer = gl.createBuffer();           // 空のオブジェクトの作成

    gl.bindBuffer(gl.ARRAY_BUFFER, arrayBuffer);      // オブジェクトを束縛

    gl.bufferData(gl.ARRAY_BUFFER,                // 配列 data の内容を GPU へ
                  new Float32Array(data),         // コピーし、頂点バッファ
                  gl.STATIC_DRAW);                // オブジェクトを作成

    gl.bindBuffer(gl.ARRAY_BUFFER, null);                    //  束縛を外す

    return arrayBuffer;                          // 作成したオブジェクトを戻す
}

function bindArrayBuffer(arrayBuffer, name, s) {
    gl.bindBuffer(gl.ARRAY_BUFFER, arrayBuffer);      // オブジェクトを束縛

    let location = gl.getAttribLocation(program, name);
                                    // バーテックスシェーダ内の name と同じ
                                    // 名前の attribute 変数の GPU 内の位置を取得

    gl.vertexAttribPointer(location, s, gl.FLOAT, false, 0, 0);
                                       // location の頂点バッファの属性を設定

    gl.enableVertexAttribArray(location);
                                    // location の頂点バッファを利用可能にする
}
```

```
// compileLink.js　前章から変更なし

// html ファイル中の id 名が vsid、fsid のリソースを文字列として読み込んで
// シェーダ実行可能コードをビルド
function buildProgram(vsid, fsid) {
    let vs = compileProgram(gl.VERTEX_SHADER, vsid);
                        // バーテックスシェーダソースプログラムのコンパイル
    let fs = compileProgram(gl.FRAGMENT_SHADER, fsid);
                        // フラグメントシェーダソースプログラムのコンパイル

    program = gl.createProgram();               // 空の実行可能コードを作成

    gl.attachShader(program, vs);
                         // バーテックスシェーダのオブジェクトコードを接続
    gl.attachShader(program, fs);
                         // フラグメントシェーダのオブジェクトコードを接続

    gl.linkProgram(program);                                    // リンク

    if (!gl.getProgramParameter(program, gl.LINK_STATUS)){ // エラーチェック
        alert(gl.getProgramInfoLog(program));           // エラー内容の表示
    }
}

// シェーダのタイプ（バーテックス/フラグメント）とリソース id から
// オブジェクトコードを戻す
function compileProgram(type, id){
    let source = document.getElementById(id);
                           // HTML ファイル中から id に相当するリソースを取得
    if (!source) { return; }                // 取得できなければ null を返す

    let shader = gl.createShader(type);         // 空のオブジェクトコードを作成

    gl.shaderSource(shader, source.text);          // ソースプログラムの文字
                                                   // 列を shader に接続

    gl.compileShader(shader);                                   // コンパイル

    if (gl.getShaderParameter(shader, gl.COMPILE_STATUS)){ // エラーチェック
        return shader;        // エラーが無いならばオブジェクトコードを戻す
    } else {
        alert(gl.getShaderInfoLog(shader));             // エラー内容の表示
    }
}
```

第7章　ポリゴンの描画

この章では三角形の内部を塗りつぶすプログラムを紹介する。

3 次元空間内の任意の被写体は、その表面を多角形（ポリゴン＝ polygon）で覆うことができる。任意の多角形は複数枚の三角形を組み合わせて表すことができる。このことから 3D CG では三角形を描画の基本図形（プリミティブ）とみなしており、これをしばしば三角形ポリゴン、三角ポリゴン（triangular polygon）、あるいは単にポリゴンと呼ぶ。

三角形ポリゴンの描画プログラムは、線分やポイントスプライトのそれと同じ構造である。

7.1　簡単なポリゴン描画

ここでは図 7.1 の、1 枚の三角形を描画する方法を紹介する。三角形を描画するという点だけが前章までと異なり、それ以外については前章までの議論と大差ない。

前節のプログラムを流用するため、三角形の 3 頂点には前節の赤、緑、白の頂点を割り当てた。三角形の内点の色は、線分の場合と同様に、3 頂点から線形補間によって求められる。ここに三角形内部の線形補間とは、三角形の 3 頂点 p_1、p_2、p_3 と任意の内点 p について、パラメータ $(\lambda_1, \lambda_2, \lambda_3)$ を

$$\lambda_1 = \frac{|p - q_1|}{|p_1 - q_1|}, \quad \lambda_2 = \frac{|p - q_2|}{|p_2 - q_2|}, \quad \lambda_3 = \frac{|p - q_3|}{|p_3 - q_3|} \tag{7.1}$$

と定義する（図 7.2 参照）とき、3 頂点の RGB カラー値 $\vec{C}_1$、$\vec{C}_2$、$\vec{C}_3$ から内点 p の RGB カラー値 $\vec{C}$ を

$$\vec{C} = \lambda_1 \vec{C}_1 + \lambda_2 \vec{C}_2 + \lambda_3 \vec{C}_3 \tag{7.2}$$

とベクトル計算で求めることをいう。パラメータ $(\lambda_1, \lambda_2, \lambda_3)$ を重心座標（barycentric coordinate）と呼ぶ。実は RGB カラー値だけではなく、任意の値が同様に線形補間できる。GPU ではこの線形補間をハードウェアで実行するため、きわめて高速な処理が可能である。

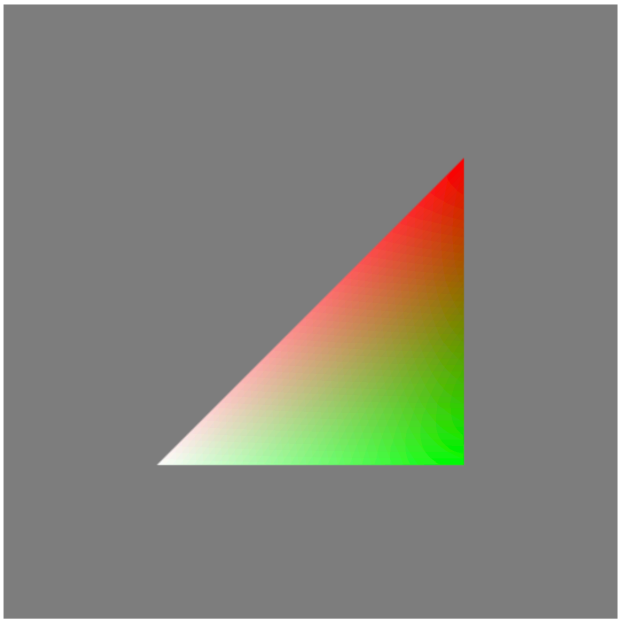


図 7.1: 画像例（その 6、一枚の三角形ポリゴンの塗りつぶしの場合）

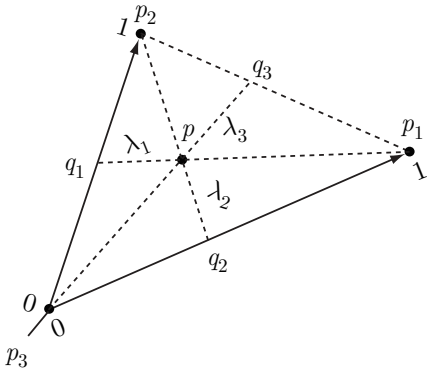


図 7.2: 重心座標系による RGB カラーの線形補間

```
// main.js
let gl;
let program;

function initSystem() {
    ** 前章から変更なし **
}

let NUM_POINTS = 3;                              // 使用する頂点数を 3 の倍数に変更

function initData() {
    ** 前章から変更なし **
}

function display() {
    gl.clear(gl.COLOR_BUFFER_BIT);
    gl.drawArrays(gl.TRIANGLES, 0, NUM_POINTS);       // gl.TRIANGLES へ変更
    gl.flush();
}

window.onload = function(){
    ** 前章から変更なし **
};
```

図 7.3: 1 枚の三角形ポリゴンを描画する JavaScript ホストプログラム `main.js`

7.1.1　ホストプログラム

図 7.3 がこの節のホストプログラムである。変更箇所は描画関数 `display()` の以下の 2 行のみである。

ひとつは、以下の通り、頂点数を 3 の倍数 — ここでは 3 — へ変更する。1 枚の三角形を定義する頂点数は 3 だからである。

```
let NUM_POINTS = 3;                              // 使用する頂点数を 3 の倍数に変更
```

もちろん、三角形を 2 枚描画する場合には頂点数は 6 であるが、ここでは 1 枚のみの描画を行う。

もうひとつは描画モードの変更である。前々章では線分を描画するために `gl.LINES` を用いた。前章ではポイントスプライトを描画するために `gl.POINTS` を用いた。この章では三角形を描画するために `gl.TRIANGLES` に変更する。

```
    gl.drawArrays(gl.TRIANGLES, 0, NUM_POINTS);       // gl.TRIANGLES へ変更
```

7.1.2　バーテックスシェーダプログラム

図 7.4 がこの節のシェーダプログラムである。前章のポイントスプライトの 1 例目の描画プログラム（図 6.3（58 ページ））から以下の 1 行を削除するだけの変更である。

```
        //gl_PointSize = 50.0;                                    // 削除
```

7.1.3　フラグメントシェーダプログラム

フラグメントシェーダプログラム（図 7.4）は前章のポイントスプライトの 1 例目の描画プログラム（図 6.3）と同じである。

7.1.4　実行の様子

図 7.5 が実行の様子である。

各頂点はそれぞれ個別のバーテックスシェーダプログラムで処理される。3 頂点の情報はラスタライザでまとめられ、内点を含む画素点が求められる。なお、ラスタライザがまとめる頂点の個数は、描画関数 display() で指定する描画モードによって決まる。このプログラムでは GL_TRIANGLES を指定したため、連続する 3 点をまとめて三角形の 3 頂点と見なす。描画モードにはこれまでに紹介したもの以外で、GL_LINE_STRIP、GL_LINE_LOOP、GL_TRIANGLE_STRIP、GL_TRIANGLE_FAN が指定できる。詳細は各自で調べてほしい。

ラスタライザが求めた各画素点についてはそれぞれのフラグメントシェーダでその画素の色が計算され、それがフレームバッファに反映されて 1 枚の画像となる。

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<script src="./main.js" type="text/javascript"></script>
<script src="./util.jsw" type="text/javascript"></script>
<script src="./compileLink.js" type="text/javascript"></script>

<script id="vs" type="text/plain">
    attribute vec2 position;
    attribute vec3 in_color;

    varying vec3 out_color;


    void main(void) {
        gl_Position = vec4(position, 0.0, 1.0);
        out_color = in_color;
        //gl_PointSize = 50.0;                                    // 削除
    }
</script>

<script id="fs" type="text/plain">
    precision highp float;                    // 前章 1 例目と同じプログラム
    varying vec3 out_color;

    void main(void) {
        gl_FragColor = vec4(out_color, 1.0);
    }
</script>

</head>
<body>
<canvas id="canvas"></canvas>
</body>
</html>
```

図 7.4: 1 枚の三角形ポリゴンを描画する HTML/シェーダプログラム

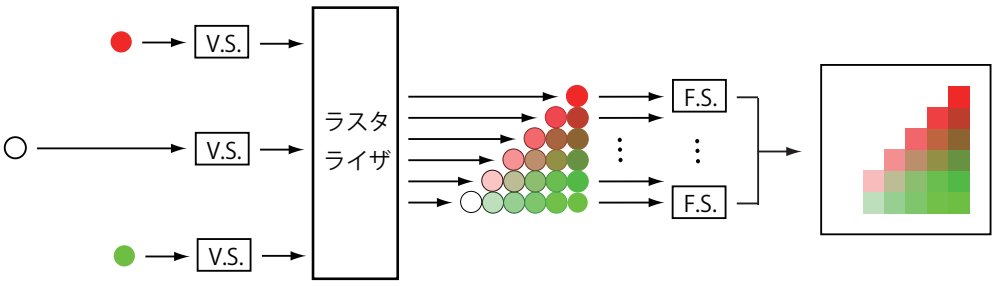


図 7.5: 三角形を描く場合の動作例

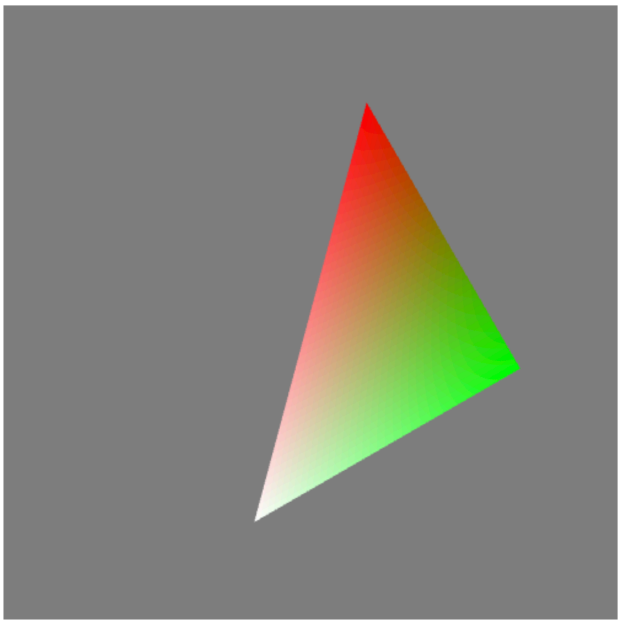


図 7.6: 画像例（その 7、三角形を 30 度回転させた場合）

7.2　シェーダプログラムへのパラメータの受け渡し（その 1）

データを CPU（ホストプログラム）から GPU（シェーダプログラム）へ受け渡す方法には、頂点データを attribute 変数としてバーテックスシェーダプログラムへ受け渡す方法以外にもうひとつ、**uniform 変数** で受け渡す方法がある。この節ではこれを用いる。

uniform（＝同一の、均一の）が意味するように、この方法では同じデータを全てのシェーダプログラムへ受け渡すことができる。たとえば図 7.6 は、図 7.1 の三角形の全ての頂点の座標を反時計回りに 30 度回転させた画像である。回転角度は全てのバーテックスシェーダで共有せねば、形状が歪んでしまう。uniform 変数はそのような同一データの共有に利用される。

uniform 変数は GPU のメモリ上に記憶領域が確保され、バーテックスシェーダプログラム、フラグメントシェーダプログラムからは読み込み専用の大域変数として利用できる（読み込み専用であるから、グローバルに定義された定数値と考えてもよい）。

uniform 変数を含むプログラム実行の概念図を図 7.7 に示す。uniform 変数の値はホストプログラムから設定する。シェーダプログラムからは uniform 変数の値を参照できるが、uniform 変数に値を代入することはできない。もしそれができるならば、複数のシェーダプログラムが uniform 変数へ異なる値を代入できることとなり、結果、uniform 変数の値は不定となり、不都合である。

以下、図 7.6 の三角形の回転を行うプログラムを解説する。主な改造点は、uniform 変数をシェーダに設定する関数の追加と利用である。

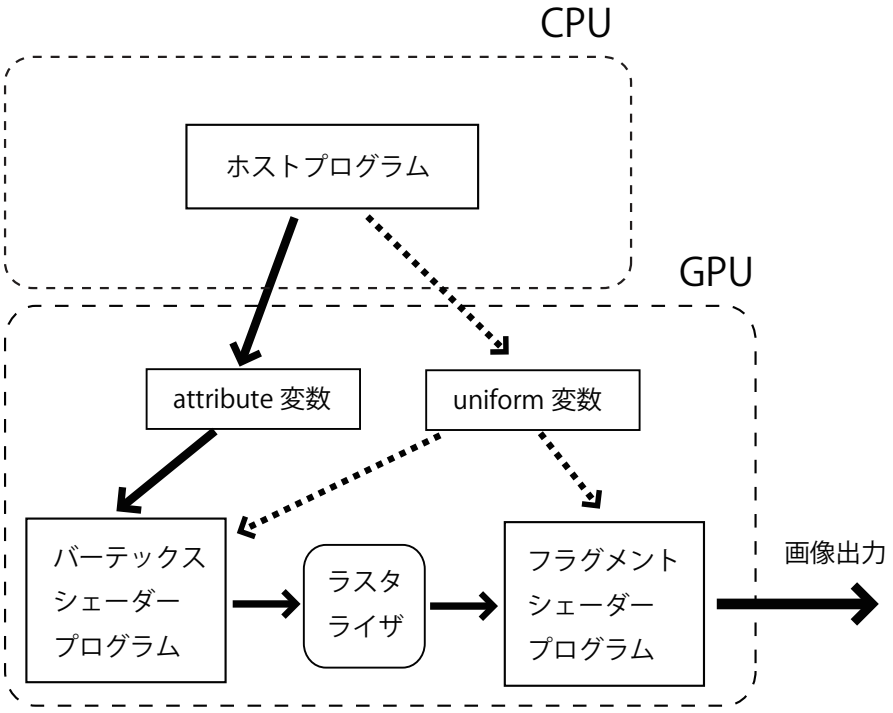


図 7.7: attribute 変数と uniform 変数の関係

7.2.1　バーテックスシェーダプログラム

uniform 変数の説明の都合上、ホストプログラムを示す前に、図 7.8（75 ページ）にバーテックスシェーダプログラムを示す。

バーテックスシェーダプログラムでは、回転角に対応する変数 theta を uniform 宣言している。

```
uniform float theta;                    // 回転角度が格納された変数
```

この変数には、ホストプログラムで設定した 30*M_PI/180.0 が格納されている。

その変数を用いて attribute 変数として入力された 2 次元座標 position を回転させる変換を行う。

```
        float x = cos(theta)*position.x-sin(theta)*position.y; // 座標の
        float y = sin(theta)*position.x+cos(theta)*position.y; // 回転
```

回転後の座標値 (x, y) は代入文：

```
        gl_Position = vec4(x, y, 0.0, 1.0);
```

によって gl_Position へ代入する。

なお、cos(theta)、sin(theta) を個々のバーテックスシェーダで繰り返し実行するのは実際には無駄である。あらかじめホストプログラムで cos(theta)、sin(theta) を 1 回だけ計算して

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<script src="./main.js" type="text/javascript"></script>
<script src="./util.jsw" type="text/javascript"></script>
<script src="./compileLink.js" type="text/javascript"></script>

<script id="vs" type="text/plain">
    attribute vec2 position;
    attribute vec3 in_color;

    varying vec3 out_color;

    uniform float theta;                          // 回転角度が格納された変数

    void main(void) {
        float x = cos(theta)*position.x-sin(theta)*position.y; // 座標の
        float y = sin(theta)*position.x+cos(theta)*position.y; // 回転

        gl_Position = vec4(x, y, 0.0, 1.0);
        out_color = in_color;
    }
</script>

<script id="fs" type="text/plain">
    precision highp float;                         // 前節と同じプログラム
    varying vec3 out_color;

    void main(void) {
        gl_FragColor = vec4(out_color, 1.0);
    }
</script>

</head>
<body>
<canvas id="canvas"></canvas>
</body>
</html>
```

図 7.8: 1 枚の三角形ポリゴンを 30° 回転させて描画する HTML/シェーダプログラム

```
function setUniformFloat(name, value) {
    let location = gl.getUniformLocation(program, name);
    gl.uniform1f(location, value);
}
```

図 7.9: uniform 変数に値をセットする関数：util.js に追加

おき、その値を二つの unifrorm 変数に格納する方が効率的であるが、ここではプログラム理解の容易さを優先するため、三角関数値をあえてバーテックスシェーダで計算している。

7.2.2　フラグメントシェーダプログラム

フラグメントシェーダプログラムは前節から変更しない。

7.2.3　ホストプログラム

図 7.8 の uniform 変数の値をセットするのは、ホストプログラムである。それを行う関数をファイル util.js に図 7.9（76 ページ）のように用意する。ここに、setUniformFloat(name, value) の第 1 引数が uniform 変数名、第 2 引数が代入する値である。関数本体の代入文：

```
let location = gl.getUniformLocation(program, name);
```

は、バーテックスシェーダプログラムおよび（または）フラグメントシェーダプログラム中から name と同じ変数名の uniform 変数を見つける。関数 gl.getUniformLocation() は、gl.getAttribLocation() に類似の機能であるが、前者は uniform 変数の、後者は attribute 変数の場所を見つける関数である。もし location の値が負値（−1）ならば、その変数名がバーテックスシェーダプログラムとフラグメントシェーダプログラムのどちらにも宣言されていない、あるいは（ここ注意！）宣言されているが、プログラム内で参照されていない[1]ことを意味する。本来は、これに対するエラー処理部を記述すべきだが、簡単のため、図 7.9 ではそれをサボっている。

関数呼び出し：

```
gl.uniform1f(location, value);
```

は uniform 変数の場所に値 val を代入する。gl.uniform1f() と類似した関数として、gl.uniform2f()、gl.uniform3f()、gl.uniform4f() があり、これらはそれぞれ vec2、vec3、vec4 型の uniform 変数へ値を代入する関数である。また、関数 gl.uniform1i() は int 型 uniform 変数へ値を代入する。それらの詳細は各自で調べてほしい。

[1] OpenGL のコンパイラは、宣言されていても参照されていない uniform 変数を（GPU のメモリを不当に圧迫する）迷惑な変数として存在を抹消する処置が行われることがある。

```
// main.js
let gl;
let program;

function initSystem() {
    ** 前章から変更なし **
}

let NUM_POINTS = 3;

function initData() {
    ** 前章から変更なし **
}

function display() {
    setUniformFloat('theta', 30*Math.PI/180.0);                    // 追加
    gl.clear(gl.COLOR_BUFFER_BIT);
    gl.drawArrays(gl.TRIANGLES, 0, NUM_POINTS);
    gl.flush();
}

window.onload = function(){
    ** 前章から変更なし **
};
```

図 7.10: 1 枚の三角形ポリゴンを描画する JavaScript ホストプログラム `main.js`

uniform 変数への実際の値の設定は、シェーダ実行前ならばどの時点で行っても良い。ここでは描画関数内で行うこととする。図 7.10（77 ページ）がそのプログラムである。

`display()` の冒頭で、uniform 変数 `theta` へ角度 30 度（ラジアンモードでは $30^\circ \cdot \pi/180$）を代入する関数呼び出し：

```
setUniformFloat('theta', 30*Math.PI/180.0);                    // 追加
```

を実行する。

7.2.4　実行の様子

図 7.11 が実行の様子である。attribute 変数として入力された座標値はバーテックスシェーダで uniform 変数 `theta` の値だけ回転される。その回転された座標値がラスタライザへ入力されて、回転された三角形の画素が求められる。

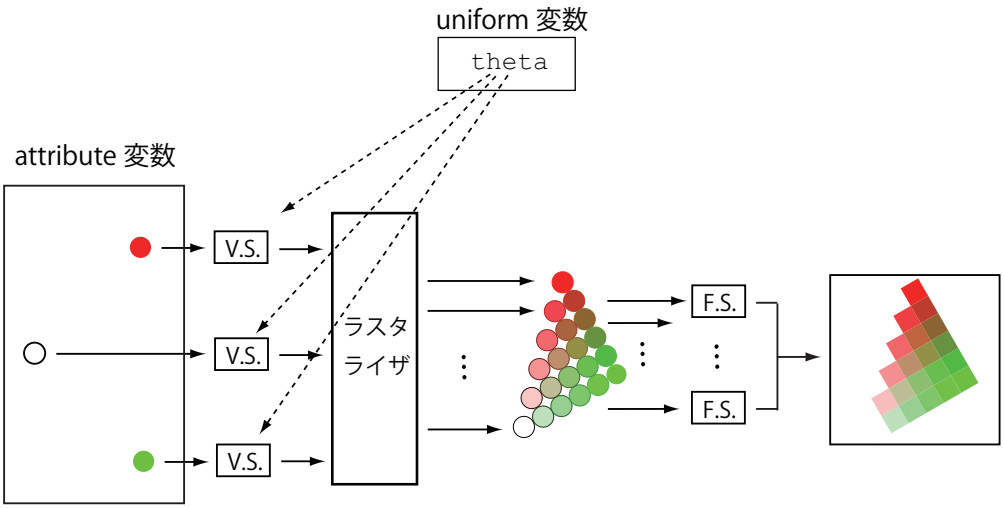


図 7.11: 三角形が回転する場合の動作例

図 7.12: 画像例（その 8、三角形を 30 度回転させた場合）

7.3　シェーダプログラムへのパラメータの受け渡し（その 2）

バーテックスシェーダプログラムで参照する uniform 変数の値はフラグメントシェーダプログラムでも全く同様に参照できる。図 7.12 は前節の `theta` をフラグメントシェーダプログラムから青色の輝度値として流用し、三角形の色を少し変えた描画例である。

7.3.1　ホストプログラム

前節と同じである。

7.3.2　バーテックスシェーダプログラム

前節と同じである。

7.3.3　フラグメントシェーダプログラム

フラグメントシェーダプログラム（図 7.13（80 ページ）の後半）では uniform 変数 `theta` を図 7.8 と同様に宣言している。その uniform 変数の参照：

```
        gl_FragColor = vec4(out_color.r, out_color.g, cos(theta), 1.0);
                                                  // 青色の輝度値を theta から計算
```

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<script src="./main.js" type="text/javascript"></script>
<script src="./util.js" type="text/javascript"></script>
<script src="./compileLink.js" type="text/javascript"></script>

<script id="vs" type="text/plain">
    attribute vec2 position;
    attribute vec3 in_color;

    varying vec3 out_color;

    uniform float theta;

    void main(void) {
        float x = cos(theta)*position.x-sin(theta)*position.y;
        float y = sin(theta)*position.x+cos(theta)*position.y;

        gl_Position = vec4(x, y, 0.0, 1.0);
        out_color = in_color;
    }
</script>

<script id="fs" type="text/plain">
    precision highp float;
    varying vec3 out_color;

    uniform float theta;                                  // uniform 変数の宣言

    void main(void) {
        gl_FragColor = vec4(out_color.r, out_color.g, cos(theta), 1.0);
                                                  // 青色の輝度値を theta から計算
    }
</script>

</head>
<body>
<canvas id="canvas"></canvas>
</body>
</html>
```

図 7.13: 1 枚の三角形ポリゴンを 30° 回転させて描画する HTML/シェーダプログラム

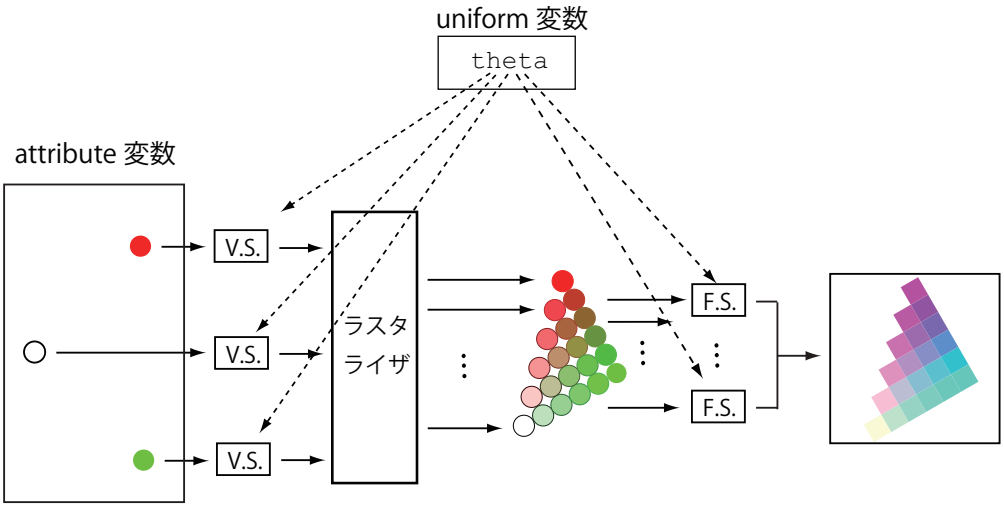


図 7.14: 三角形の色が変わる動作例

では、画素点の青色成分を `cos(theta)` と計算する。`cos(theta)` $= \cos(30^\circ \cdot \pi/180) = 0.866..$ であるから、図 7.6 に比べてかなり青色成分が加わった画像になる。なお、**この計算には単に uniform 変数を使った例という以上の意味はない**ことを注意しておく。

`vec3` 型の変数の 3 成分は、`.x`、`.y`、`.z` で参照できるが、`.r`、`.g`、`.b` で参照することもできる。上の代入文の `out_color.r, out_color.g` がそれである。`cos(theta)` の値は `theta` の値次第では負値となりうるが、負の輝度値は単に 0 と解釈される。

7.3.4　実行の様子

図 7.14 が実行の様子である。ラスタライザからフラグメントシェーダプログラムに入力された色の青色成分が変更されて、フレームバッファに出力される。

7.4　補足：この章のプログラムのリスト

プログラム読解の便利のために、以下にこの章の最後で解説したプログラムのリストを載せる。

```
<!-- HTML -->
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<script src="./main.js" type="text/javascript"></script>
<script src="./util.js" type="text/javascript"></script>
<script src="./compileLink.js" type="text/javascript"></script>

<script id="vs" type="text/plain">
    attribute vec2 position;
    attribute vec3 in_color;

    varying vec3 out_color;

    uniform float theta;

    void main(void) {
        float x = cos(theta)*position.x-sin(theta)*position.y;
        float y = sin(theta)*position.x+cos(theta)*position.y;

        gl_Position = vec4(x, y, 0.0, 1.0);
        out_color = in_color;
    }
</script>

<script id="fs" type="text/plain">
    precision highp float;
    varying vec3 out_color;

    uniform float theta;           // 回転角度を色の一様な変化にも流用

    void main(void)
    {
        gl_FragColor = vec4(out_color.r, out_color.g, cos(theta), 1.0);
    }
</script>

</head>
<body>
<canvas id="canvas"></canvas>
</body>
</html>
```

```
// main.js
let gl;
let program;

function initSystem() {
    let c = document.getElementById('canvas');
    c.width = 500; c.height = 500;

    gl = c.getContext('webgl');

    gl.clearColor(0.5, 0.5, 0.5, 1.0);

    buildProgram('vs', 'fs');
    gl.useProgram(program);
}

let NUM_POINTS = 3;

function initData() {
    let position = [
        -0.5, -0.5,                                                    // 左下
        +0.5, +0.5,                                                    // 右上
        +0.5, -0.5,                                                    // 右下
        -0.5, +0.5                                                     // 左上
    ];

    let buffer1 = buildArrayBuffer(position);

    bindArrayBuffer(buffer1, 'position', 2);

    let color = [
        1.0, 1.0, 1.0,                                                 // 左下
        1.0, 0.0, 0.0,                                                 // 右上
        0.0, 1.0, 0.0,                                                 // 右下
        0.0, 0.0, 1.0                                                  // 左上
    ];

    let buffer2 = buildArrayBuffer(color);

    bindArrayBuffer(buffer2, 'in_color', 3);
}

function display() {
    setUniformFloat('theta', 30*Math.PI/180.0);
    gl.clear(gl.COLOR_BUFFER_BIT);
    gl.drawArrays(gl.TRIANGLES, 0, NUM_POINTS);
    gl.flush();
}

window.onload = function(){
    initSystem();
    initData();
    display();
};
```

```
// util.js
function buildArrayBuffer(data) {          // 頂点バッファオブジェクトの作成
    let arrayBuffer = gl.createBuffer();           // 空のオブジェクトの作成

    gl.bindBuffer(gl.ARRAY_BUFFER, arrayBuffer);      // オブジェクトを束縛

    gl.bufferData(gl.ARRAY_BUFFER,                // 配列 data の内容を GPU へ
                  new Float32Array(data),         // コピーし、頂点バッファ
                  gl.STATIC_DRAW);                // オブジェクトを作成

    gl.bindBuffer(gl.ARRAY_BUFFER, null);                    //  束縛を外す

    return arrayBuffer;                          // 作成したオブジェクトを戻す
}

function bindArrayBuffer(arrayBuffer, name, s) {
    gl.bindBuffer(gl.ARRAY_BUFFER, arrayBuffer);      // オブジェクトを束縛

    let location = gl.getAttribLocation(program, name);
                                    // バーテックスシェーダ内の name と同じ
                                    // 名前の attribute 変数の GPU 内の位置を取得

    gl.vertexAttribPointer(location, s, gl.FLOAT, false, 0, 0);
                                       // location の頂点バッファの属性を設定

    gl.enableVertexAttribArray(location);
                                    // location の頂点バッファを利用可能にする
}


function setUniformFloat(name, value) {
    let location = gl.getUniformLocation(program, name);
    gl.uniform1f(location, value);
}
```

```
// compileLink.js　前章から変更なし

// html ファイル中の id 名が vsid、fsid のリソースを文字列として読み込んで
// シェーダ実行可能コードをビルド
function buildProgram(vsid, fsid) {
    let vs = compileProgram(gl.VERTEX_SHADER, vsid);
                        // バーテックスシェーダソースプログラムのコンパイル
    let fs = compileProgram(gl.FRAGMENT_SHADER, fsid);
                        // フラグメントシェーダソースプログラムのコンパイル

    program = gl.createProgram();               // 空の実行可能コードを作成

    gl.attachShader(program, vs);
                         // バーテックスシェーダのオブジェクトコードを接続
    gl.attachShader(program, fs);
                        // フラグメントシェーダのオブジェクトコードを接続

    gl.linkProgram(program);                                    // リンク

    if (!gl.getProgramParameter(program, gl.LINK_STATUS)){ // エラーチェック
        alert(gl.getProgramInfoLog(program));           // エラー内容の表示
    }
}

// シェーダのタイプ（バーテックス/フラグメント）とリソース id から
// オブジェクトコードを戻す
function compileProgram(type, id){
    let source = document.getElementById(id);
                           // HTML ファイル中から id に相当するリソースを取得
    if (!source) { return; }                // 取得できなければ null を返す

    let shader = gl.createShader(type);         // 空のオブジェクトコードを作成

    gl.shaderSource(shader, source.text);          // ソースプログラムの文字
                                                   // 列を shader に接続

    gl.compileShader(shader);                                   // コンパイル

    if (gl.getShaderParameter(shader, gl.COMPILE_STATUS)){ // エラーチェック
        return shader;        // エラーが無いならばオブジェクトコードを戻す
    } else {
        alert(gl.getShaderInfoLog(shader));             // エラー内容の表示
    }
}
```

第8章　CGアニメーション

この章では、前節の三角形（図 7.12）が回転する CG アニメーションを行う。プログラムの改造は簡単である。ベース言語の JavaScript にアニメーションを実現するための機能が用意されているから、それを用いればよい。

8.1　ポリゴンの回転のアニメーション

ここでは前章の三角形ポリゴンを描画する CG アニメーションを行う。

重要なのは `window` オブジェクトのメソッド `requestAnimationFrame()` の利用である。このメソッドは、Javascript のプログラムが実行されている PC のディスプレイのリフレッシュレートに合わせて引数で与えたメソッドを実行する予約を行う。いわゆるコールバックメソッドの登録である。

プログラム例を見ると、図 8.1（87 ページ）の下の方：

```
requestAnimationFrame(display);              // display の実行の登録 (1)
```

が、`requestAnimationFrame()` の 1 回目の実行である。つまり、引数のメソッド `display()` を実行する予約をここで行なっている。多くの場合、ディスプレイのリフレッシュレートは 1/60 秒であるから、上の `requestAnimationFrame()` の実行の後、1/60 秒後に `display()` が実行されることとなる。

図 8.1 の `display()` の定義は

```
function display(time) {                                    // 引数を追加
```

となっており、前章までと異なり、新たに引数 `time` が追加されている。実は `requestAnimationFrame()` の引数として登録されたメソッド（この場合、`display()`）にはそのメソッドが実行された時刻が引数として渡される約束となっている。`display()` の新たな引数 `time` はその時刻を受け取るた

```
// main.js
let gl;
let program;

function initSystem() {
    ** 前章から変更なし **
}

let NUM_POINTS = 3;

function initData() {
    ** 前章から変更なし **
}

function display(time) {                                    // 引数を追加
    setUniformFloat('theta', 0.05*time*Math.PI/180.0);     // 回転角の設定
    gl.clear(gl.COLOR_BUFFER_BIT);
    gl.drawArrays(gl.TRIANGLES, 0, NUM_POINTS);
    gl.flush();
    requestAnimationFrame(display);            // display の実行の登録 (2)
}

window.onload = function(){
    initSystem();
    initData();
    requestAnimationFrame(display);            // display の実行の登録 (1)
};
```

図 8.1: 三角形ポリゴンの回転アニメーションを行うホストプログラム main.js

めのものである。この時刻はシステムが定義する初期時刻からの経過時間（ミリ秒単位）である。よって時刻そのものはシステム依存である。

図 8.1 のプログラムでは引数 time を用いて、

```
    setUniformFloat('theta', 0.05*time*Math.PI/180.0);     // 回転角の設定
```

と、回転角度を計算している。time の値に比例して回転角度が増える。time の値の単位はミリ秒であるから、1 秒後には time は 1000 増える。よって 0.05*time は一秒で 50 増えることとなり、三角形は 1 秒間に 50° 回転する。1 回転するのに掛かる時間は $360/50 = 7.2$ 秒である。

CG アニメーションを行うには、次々と描画を繰り返さねばならない。メソッド display() の末尾の

```
    requestAnimationFrame(display);            // display の実行の登録 (2)
```

は、次の描画を予約するものである。つまり、 `display()` の中で、次の `display()` の実行を予約し、ディスプレイのリフレッシュレート毎の連続描画を実現している。

　この章のプログラムの変更はわずかであるから、恒例の章末のプログラムリストの再掲は省略する。

第9章　1次元テクスチャの利用

より高度なCG描画では三角形ポリゴンの表面に画像を貼り付け、ポリゴンに豊かな質感を与え、表現力を高める。この画像をテクスチャ（texture、質感の意味）と呼ぶ。テクスチャの張り付け処理をテクスチャマッピングと呼ぶ。「画像を貼り付け...」と書いたが、これは典型的なテクスチャマッピングの例であって、実はテクスチャデータは画像に限らない。1次元〜3次元の形状のデータの並び（実装上は配列）をテクスチャとしてGPU内で参照できる。

この章では手始めに1次元テクスチャを利用した様々なプログラミング手法を解説する。一般にテクスチャを用いるプログラムは前章までのプログラムから難易度が上がるから、丁寧にゆっくりと解説していく。

9.1　縞模様のマッピング

この節で目的とするCGアニメーションは図9.1の縞模様を張った三角形である。前章を引き継ぎ、この三角形が回転するCGアニメーションを行う。この縞模様を実現するために、色が

黒、白、黒、白

の順に繰り返すRGBA値の配列を用意する。これが1次元テクスチャデータである。フラグメントシェーダプログラムでは、このテクスチャデータを $[0,1]$ の範囲の1次元座標値を与えて参照する。

以下、これまでと同様にプログラムを示し、その後に、前章からの変更箇所を順次、解説する。プログラム中のコメントの付いている箇所に注意してほしい。

9.1.1　ホストプログラム

図9.2（90ページ）、図9.3（91ページ）が、ホストプログラムの主要部分である。

まず、メソッド `initSystem()` に以下を追加する。これは、WebGLのテクスチャデータの設定に `float` 型を用いる場合に必要となる拡張機能の追加設定である[1]。

[1] 今後、この機能は標準装備されることとなるだろうが、現時点では拡張が必要である。

図 9.1: 画像例（その 9、1 次元テクスチャ利用の場合、CG アニメーションの最初の 1 枚に相当）

```
let gl;
let program;

function initSystem() {
    let c = document.getElementById('canvas');
    c.width = 500; c.height = 500;

    gl = c.getContext('webgl');

    gl.clearColor(0.5, 0.5, 0.5, 1.0);

    let ext = gl.getExtension('OES_texture_float') ||           // 機能拡張
              gl.getExtension('OES_texture_half_float');        // 機能拡張
    if (ext == null) {                              // 機能拡張に失敗したならば
        alert('float texture not supported');       // アラートを表示して
        return;                                     // リターン
    }

    buildProgram('vs', 'fs');
    gl.useProgram(program);
}
```

図 9.2: 縞模様のテクスチャを描画する JavaScript ホストプログラム `main.js`（その 1、その 2 へ続く）

```
let NUM_POINTS = 3;

function initData() {
    let position = [                                    //頂点の 2 次元座標値
        -0.5, -0.5,                                                  // 左下
        +0.5, +0.5,                                                  // 右上
        +0.5, -0.5                                                   // 右下
    ];
    let buffer1 = buildArrayBuffer(position);
    bindArrayBuffer(buffer1, 'position', 2);

    let u_value = [                          // 頂点の 1 次元テクスチャ座標値
        0.0,                                                         // 左下
        1.0,                                                         // 右上
        1.0                                                          // 右下
    ];
    let buffer2 = buildArrayBuffer(u_value);
    bindArrayBuffer(buffer2, 'in_u', 1);

    let TEXTURE_SIZE = 4;                                  // 2 の冪乗に設定
    let rgbaData = new Float32Array([
        0.0, 0.0, 0.0, 1.0,                                // 黒
        1.0, 1.0, 1.0, 1.0,                                // 白
        0.0, 0.0, 0.0, 1.0,                                // 黒
        1.0, 1.0, 1.0, 1.0                                 // 白
    ]);
    let texture = buildTexture(rgbaData, TEXTURE_SIZE);
                                          // テクスチャオブジェクトの生成
    bindTexture(texture, 'tex1', 0);
                         // texture を シェーダ内の uniform 変数 tex1 と結合
}

function display(time) {
    ** 前章と同じ **
}

window.onload = function() {
    ** 前章と同じ **
};
```

図 9.3: 縞模様のテクスチャを描画する JavaScript ホストプログラム main.js（その 2）

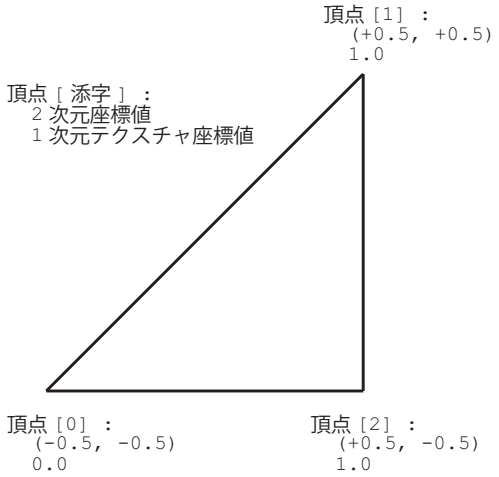


図 9.4: 三角形の各頂点に付随する attribute 変数の情報

```
    let ext = gl.getExtension('OES_texture_float') ||           // 機能拡張
              gl.getExtension('OES_texture_half_float');        // 機能拡張
    if (ext == null) {                              // 機能拡張に失敗したならば
        alert('float texture not supported');       // アラートを表示して
        return;                                     // リターン
    }
```

initSystem() のそれ以外の部分は前章までと同じである。

initData() の変更は図 9.3 の通りである。

頂点の座標値データは前章までと同様に設定する。

この節の例題では頂点に RGB 値は設定しない。その代りに、三角形の 3 頂点にそれぞれ **1 次元テクスチャ座標値**を以下のように設定する。三角形の頂点の色は、この座標値から参照されるテクスチャの RGB 値である。

```
    let u_value = [                                 // 頂点の 1 次元テクスチャ座標値
        0.0,                                                    // 左下
        1.0,                                                    // 右上
        1.0                                                     // 右下
    ];
```

なお、テクスチャ座標値は $[0, 1]$ の範囲で設定する（その条件は次章で外すのだが）。頂点の 2 次元座標値と 1 次元テクスチャ座標値の関係は図 9.4 の通りとなる。

次に、以下のようにテクスチャデータを配列に作成し、その配列を基にしてメソッド buildTexture() を用いてテクスチャオブジェクトを生成し、メソッド bindexture() を用いてそのオブジェクト

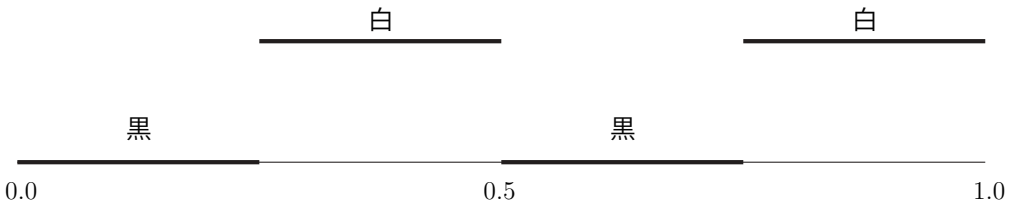


図 9.5: テクスチャ座標値とテクスチャデータ値の関係（その 1）

をシェーダプログラム中の uniform 変数 ”tex1” と結合する。このとき、このテクスチャにテクスチャ番号 0 を割り当てる。なお、メソッド buildTexture()、bindTexture() の中の処理については後述する。

```
let TEXTURE_SIZE = 4;                                    // 2 の冪乗に設定
let rgbaData = new Float32Array([
    0.0, 0.0, 0.0, 1.0,                                  // 黒
    1.0, 1.0, 1.0, 1.0,                                  // 白
    0.0, 0.0, 0.0, 1.0,                                  // 黒
    1.0, 1.0, 1.0, 1.0                                   // 白
]);
let texture = buildTexture(rgbaData, TEXTURE_SIZE);
                                          // テクスチャオブジェクトの生成
bindTexture(texture, 'tex1', 0);
                      // texture を シェーダ内の uniform 変数 tex1 と結合
```

テクスチャ配列の各 RGB 値は、 (0.0, 0.0, 0.0), (1.0, 1.0, 1.0), (0.0, 0.0, 0.0), (1.0, 1.0, 1.0) と設定しているから、つまり、白、黒、白、黒 である。アルファ値は常に 1.0（不透明）を設定する。

さて、この例題プログラムでのテクスチャ座標値とテクスチャデータの関係は図 9.5 に示す通りである。テクスチャ座標の範囲は $[0, 1]$ であるから、その区間を 4 分割した各区間の色が上に設定した色に対応する。座標値から色を求める操作はフラグメントシェーダで行う。

ここでテクスチャの利用に関して以下の制限があることを注意する。

- **テクスチャサイズ（上のプログラムでは TEXTURE_SIZE） は 2 の冪乗数 2, 4, 8, 16, ... でなければならない**。

この制約は本家 OpenGL の最新バージョンではもはや課されていないが、WebGL では依然として残っている。近い将来、この制約は WebGL からも消えるだろうが、現時点でテクスチャサイズを 2 の冪乗以外に設定した場合、シェーダプログラムでテクスチャデータの読み込みを行っても RGBA 値として $(0, 0, 0, 0)$ が読み込まれるだけである。

```
function buildTexture(data, w) {
    let texture = gl.createTexture();          // テクスチャオブジェクトの生成

    gl.bindTexture(gl.TEXTURE_2D, texture);           // テクスチャのバインド

    // テクスチャへデータを格納
    gl.texImage2D(gl.TEXTURE_2D,                          // テクスチャの種類
                  0,                                   // 縮小テクスチャのレベル
                  gl.RGBA,                           // テクスチャデータの内部形式
                  w,                                        // テクスチャの幅
                  1,                                       // テクスチャの高さ
                  0,                                       // システムの固定値
                  gl.RGBA,                               // 読み出すデータの形式
                  gl.FLOAT,                          // テクスチャデータの要素の形式
                  data);                                  // テクスチャ用配列データ

    // テクスチャデータの取り出し方法を指定
    gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER, gl.NEAREST);
    gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.NEAREST);

    gl.bindTexture(gl.TEXTURE_2D, null);       // テクスチャのバインドを無効化

    return texture;
}

function bindTexture(texture, name, num) {
    let location = gl.getUniformLocation(program, name);       // 場所の取得
    gl.uniform1i(location, num);                  // テクスチャに装置番号を格納

    gl.activeTexture(gl.TEXTURE0+num); // 装置番号のテクスチャをアクティブ化

    gl.bindTexture(gl.TEXTURE_2D, texture);           // テクスチャのバインド
}
```

図 9.6: 1 次元テクスチャの操作に関するメソッド：util.js に追加

テクスチャオブジェクトを生成するメソッド buildTexture()、そのオブジェクトをシェーダプログラムに結合するメソッド bindexture() は、図 9.6（94 ページ）のように作る。

まず、buildTexture() の冒頭：

```
    let texture = gl.createTexture();          // テクスチャオブジェクトの生成
```

では、空（から）のテクスチャオブジェクトをとりあえず作る。次に、

```
    gl.bindTexture(gl.TEXTURE_2D, texture);           // テクスチャのバインド
```

では、これから 2 次元テクスチャオブジェクトの設定を行うこと、およびその設定対象が texture であることを宣言している。この gl.bind...() 系のメソッドとして、以前に gl.bindBuffer() を解説したが、これから設定を始めることの先触れを行うという点で gl.bindTexture() と gl.bindBuffer() は類似している。

テクスチャの具体的な設定は以下のメソッド呼び出しからである。

```
gl.texImage2D(gl.TEXTURE_2D,                    // テクスチャの種類
```

ここに **2D** は 2 次元であることを意味する。すなわちこの設定は 2 次元テクスチャに関する設定である。1 次元でなく、2 次元を用いる理由は、本家の OpenGL には 1 次元テクスチャを設定する関数（たとえば glTextImage1D()）があるが、WebGL には無いためである。1 次元テクスチャは利用頻度が低いこと、2 次元テクスチャは 1 次元テクスチャの機能を包含することから省スペースの観点から 1 次元テクスチャを省略しやものと思われる。しかし、テクスチャ理解に 1 次元テクスチャを触ってみることは有意義と講義担当者は考えるから、ここでは WebGL の 2 次元テクスチャ機能を用いて 1 次元テクスチャを取り扱うこととする。

さて、メソッド呼び出し gl.texImage2D(gl.TEXTURE_2D, ...) は、ホストプログラム上で作成したテクスチャデータに対応するメモリ領域を GPU に確保し、そのデータをホストから GPU へ転送する。実はこのメソッドは多様な機能を有しており、使いこなすにはかなり深い理解が必要である。しかしここでは引数について以下に概説するのみにとどめる[2]。

第 1 引数 ... テクスチャの種類を指定する。ここでは 2 次元テクスチャ gl.TEXTURE_2D である。他にキューブ環境テクスチャに対応する gl.TEXTURE_CUBE_MAP_POSITIVE_X などが使用できるが、省略する。

第 2 引数 ... データをミップマップの縮小テクスチャを転送する場合のレベル（level of detail）を指定する。ここではミップマップを用いない場合の 0 を指定する。

第 3 引数 ... テクスチャデータの内部形式の基本要素数を指定する。要素数を数値で指定するよりも、WebGL のマクロで指定する方がよい。ここでは最も一般的な RGBA 形式 gl.RGBA（要素数は 4）である。

第 4 引数 ... 第 9 引数で渡す 2 次元テクスチャデータの幅を指定する。ここでは w である。

第 5 引数 ... 第 9 引数で渡す 2 次元テクスチャデータの高さを指定する。ここでは、2 次元テクスチャを 1 次元テクスチャとして利用するから、高さは 1 である。

[2]実は講義担当者も完全に理解している訳ではない。

第 6 引数 ... システム予約の固定値 0 を指定する。

第 7 引数 ... シェーダプログラムで読み出すピクセルデータの形式を指定する。ここでは RGBA 形式 gl.RGBA である。

第 8 引数 ... テクスチャデータの要素の形式を指定する。ここでは float 型（gl.FLOAT）である。

第 9 引数 ... ホストプログラム中のテクスチャデータが格納されているメモリの先頭アドレスを指定する。ここでは引数として受け取る data である。data の内容は 関数 initData() の説明の通りである。

なお、これらの引数のうち、第 3、第 7、第 8 引数の指定方法には様々なバリエーションがある。標準的な float 型の RGBA 形式以外を用いる場合には特に注意が必要である。この授業ではこれについて深く理解する必要はないし、他の形式を利用することはしない。

次に、

```
// テクスチャデータの取り出し方法を指定
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER, gl.NEAREST);
```

は、テクスチャから値を読み出すときに（特にテクスチャがディスプレイ上で拡大されて表示されるときに）、最近傍（gl.NEAREST）の位置の RGBA 値を読み出すことを指定する。最近傍以外の選択肢として線形補間も可能である。これについては後に述べる。

その次の行：

```
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.NEAREST);
```

は、テクスチャがディスプレイ上で縮小されて表示されるときの RGBA 値の読み出し方を指定する。これも最近傍（gl.NEAREST）の位置の RGBA 値を読み出すことを指定する。

メソッド呼び出し：

```
gl.bindTexture(gl.TEXTURE_2D, null);       // テクスチャのバインドを無効化
```

は、このテクスチャに関する設定の終了を意味する。

以上でテクスチャオブジェクトの設定は終了し、オブジェクトを戻り値とする。

次に、関数 bindTexture() の冒頭の 2 行：

```
let location = gl.getUniformLocation(program, name);       // 場所の取得
gl.uniform1i(location, num);                    // テクスチャに装置番号を格納
```

は、名前が name であるようなテクスチャのシェーダプログラム内の位置（location）を求め、その位置にテクスチャの装置番号を格納する。gl.getUniformLocation() と gl.gl.uniform...() を用いる方法は、7.2 節（73 ページ）と同じである。

次の 1 行：

```
gl.activeTexture(gl.TEXTURE0+num); // 装置番号のテクスチャをアクティブ化
```

は、装置番号 num のテクスチャの設定を開始することを WebGL の管理システムに宣言する。メソッド名に active... とあるが、GPU 内のテクスチャには全く作用を及ぼさず、単に管理システムへ宣言しているだけである。この辺の仕様が、WebGL（そして大元の OpenGL）を**非常に分かりにくい！**ものにしている。

そしてその次の 1 行：

```
gl.bindTexture(gl.TEXTURE_2D, texture);            // テクスチャのバインド
```

で、直前にアクティブ（設定開始）にした装置番号 num のテクスチャをテクスチャオブジェクト texture に結合する。

以上でテクスチャの利用準備が完了した。

9.1.2　バーテックスシェーダプログラム

バーテックスシェーダプログラムは図 9.7（98 ページ）の前半の通りである。前章では RGB カラー値を attribute 変数 in_color で受け渡したが、ここでは float 型の 1 次元テクスチャ座標値を attribute 変数を in_u で受け取り、そのままラスタライザへ受け渡す。そしてラスタライザは線形補間した 1 次元テクスチャ座標値を各画素のフラグメントシェーダへ受け渡す。

9.1.3　フラグメントシェーダプログラム

フラグメントシェーダプログラムは図 9.7 の後半の通りである。

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<script src="./main.js" type="text/javascript"></script>
<script src="./util.js" type="text/javascript"></script>
<script src="./compileLink.js" type="text/javascript"></script>

<script id="vs" type="text/plain">
    attribute vec2 position;
    attribute float in_u;                              // 1 次元テクスチャ座標値

    varying float out_u;       // ラスタライザへ受け渡す 1 次元テクスチャ座標値

    uniform float theta;

    void main(void) {
        float x = cos(theta)*position.x-sin(theta)*position.y;
        float y = sin(theta)*position.x+cos(theta)*position.y;

        gl_Position = vec4(x, y, 0.0, 1.0);
        out_u = in_u;
    }
</script>

<script id="fs" type="text/plain">
    precision highp float;
    varying float out_u; // ラスタライザから受け渡された 1 次元テクスチャ座標値

    uniform sampler2D tex1;                                  // 2 次元テクスチャ


    void main(void) {
        gl_FragColor = texture2D(tex1, vec2(out_u,0.0));
                                                  //テクスチャデータの読み込み
    }
</script>

</head>
<body>
<canvas id="canvas"></canvas>
</body>
</html>
```

図 9.7: 縞模様のテクスチャを描画する HTML/シェーダプログラム

上にも触れたが、テクスチャデータは uniform 変数として参照される。2 次元テクスチャのデータ型は GLSL の組み込みデータ型 sampler2D と約束されており、テクスチャデータの宣言は

```
uniform sampler2D tex1;                                    // 2 次元テクスチャ
```

となる。

画素の RGB 値の代入文：

```
gl_FragColor = texture2D(tex1, vec2(out_u, 0.0));
                                                  //テクスチャデータの読み込み
```

の右辺 texture2D(tex1, vec2(out_u, 0.0)) は、テクスチャデータ tex1 から 2 次元座標値 vec2(out_u, 0.0) の位置の RGBA データを求める関数呼び出しである。第 1 引数 out_u が水平方向の座標値、第 2 引数が 0.0 が垂直方法の座標値である。垂直方向のテクスチャの幅は 1 （図 9.6 の以下の指定：

```
                1,                                         // テクスチャの高さ
```

を思い出そう）であるから、第 2 引数値は $[0, 1]$ の範囲のどんな値であっても結果は変わらない。

座標値と色の対応関係は、既に述べたように、図 9.5 の通りである。

9.1.4　実行の様子

プログラムの実行の概念図が図 9.8 である。

配列バッファからバーテックスシェーダの attribute 変数に入力された 1 次元座標値はそのままラスタライザへ受け渡される。ラスタライザは、3 頂点からなる三角形の内側の全ての画素点について座標値を線形補間する。フラグメントシェーダは、uniform 変数に格納されている 1 次元テクスチャデータ tex1 の中から座標値に対応する RGBA 値を読み込み、それをフレームバッファへ書き込む。

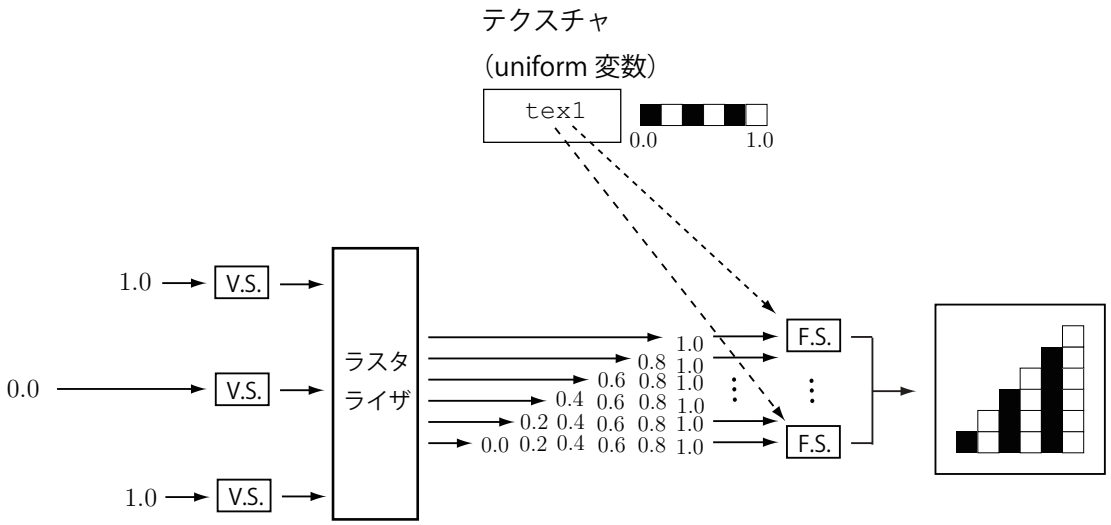


図 9.8: 三角形に 1 次元テクスチャを張る場合の動作例

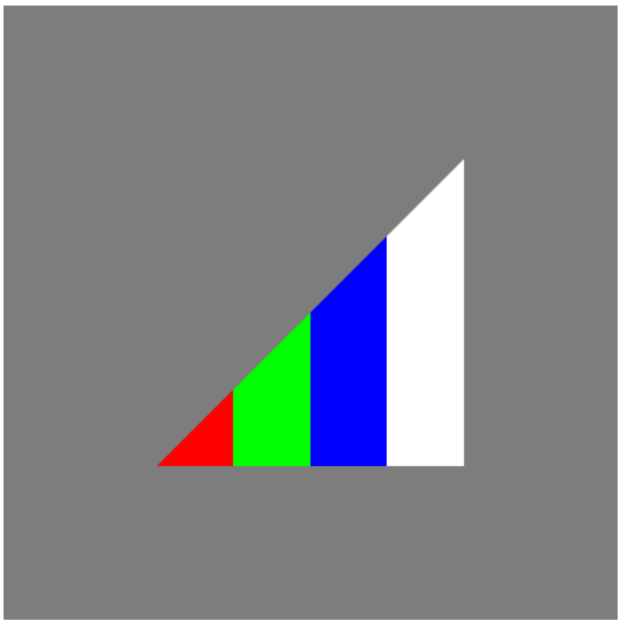


図 9.9: 画像例（その 11、カーラーテクスチャの利用）

9.2　カーラーデータの利用

次に、図 9.3 のプログラムの

```
let rgbaData = new Float32Array([
    0.0, 0.0, 0.0, 1.0,                                    // 黒
    1.0, 1.0, 1.0, 1.0,                                    // 白
    0.0, 0.0, 0.0, 1.0,                                    // 黒
    1.0, 1.0, 1.0, 1.0                                     // 白
]);
```

を

```
let rgbaData = new Float32Array([
    1.0, 0.0, 0.0, 1.0,                                    // 赤
    0.0, 1.0, 0.0, 1.0,                                    // 緑
    0.0, 0.0, 1.0, 1.0,                                    // 青
    1.0, 1.0, 1.0, 1.0                                     // 白
]);
```

へ変更してみよう。図 9.9 の描画像が得られる。

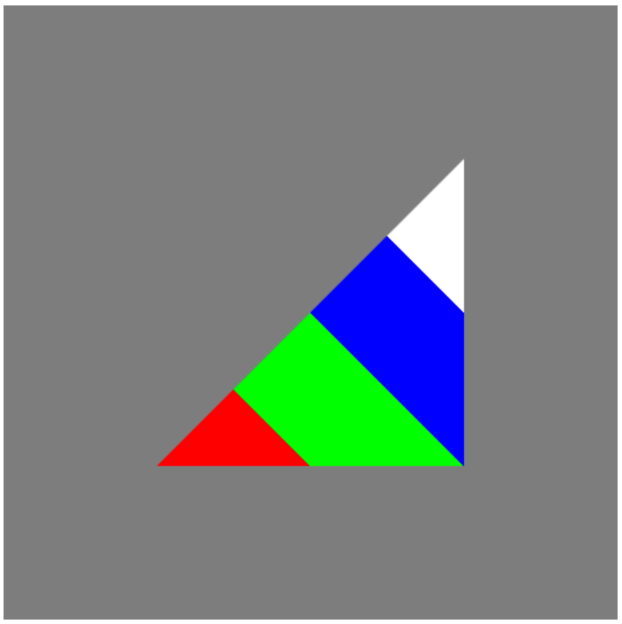


図 9.10: 画像例（その 12、テクスチャを傾斜させる変更）

9.3　テクスチャの傾斜

次に、図 9.3 のプログラムのテクスチャ座標値

```
    let u_value = [                                 // 頂点の 1 次元テクスチャ座標値
        0.0,                                                        // 左下
        1.0,                                                        // 右上
        1.0                                                         // 右下
    ];
```

の右辺を以下のように変えてみる。

```
    let u_value = [                                 // 頂点の 1 次元テクスチャ座標値
        0.0,                                                        // 左下
        1.0,                                                        // 右上
        0.5                                       // 右下, 1.0 -> 0.5 へ変更
    ];
```

これによって三角形の各画素が参照するテクスチャデータの位置が変わるから、画像は図 9.10 のように変わる。

9.4　補足：この章のプログラムのリスト

プログラム読解の便利のために、以下にこの章の最後で解説したプログラムのリストを載せる。

```
<!-- HTML -->
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<script src="./main.js" type="text/javascript"></script>
<script src="./util.js" type="text/javascript"></script>
<script src="./compileLink.js" type="text/javascript"></script>

<script id="vs" type="text/plain">
    attribute vec2 position;
    attribute float in_u;                                   // 1 次元テクスチャ座標値

    varying float out_u;       // ラスタライザへ受け渡す 1 次元テクスチャ座標値

    uniform float theta;

    void main(void) {
        float x = cos(theta)*position.x-sin(theta)*position.y;
        float y = sin(theta)*position.x+cos(theta)*position.y;

        gl_Position = vec4(x, y, 0.0, 1.0);
        out_u = in_u;
    }
</script>

<script id="fs" type="text/plain">
    precision highp float;
    varying float out_u; // ラスタライザから受け渡された 1 次元テクスチャ座標値

    uniform sampler2D tex1;                                     // 2 次元テクスチャ

    void main(void) {
        gl_FragColor = texture2D(tex1, vec2(out_u,0.0));
                                                    //テクスチャデータの読み込み
    }
</script>

</head>
<body>
<canvas id="canvas"></canvas>
</body>
</html>
```

```
// main.js
let gl;
let program;

function initSystem() {
    let c = document.getElementById('canvas');
    c.width = 500; c.height = 500;

    gl = c.getContext('webgl');

    gl.clearColor(0.5, 0.5, 0.5, 1.0);

    let ext = gl.getExtension('OES_texture_float') ||           // 機能拡張
              gl.getExtension('OES_texture_half_float');        // 機能拡張
    if (ext == null) {                              // 機能拡張に失敗したならば
        alert('float texture not supported');       // アラートを表示して
        return;                                     // リターン
    }

    buildProgram('vs', 'fs');
    gl.useProgram(program);
}

let NUM_POINTS = 3;

function initData() {
    let position = [                                    //頂点の 2 次元座標値
        -0.5, -0.5,                                                 // 左下
        +0.5, +0.5,                                                 // 右上
        +0.5, -0.5                                                  // 右下
    ];
    let buffer1 = buildArrayBuffer(position);
    bindArrayBuffer(buffer1, 'position', 2);

    let u_value = [                             // 頂点の 1 次元テクスチャ座標値
        0.0,                                                        // 左下
        1.0,                                                        // 右上
        0.5                                         // 右下, 1.0 -> 0.5 へ変更
    ];
    let buffer2 = buildArrayBuffer(u_value);
    bindArrayBuffer(buffer2, 'in_u', 1);

    let TEXTURE_SIZE = 4;                                   // 2 の冪乗に設定
    let rgbaData = new Float32Array([
        1.0, 0.0, 0.0, 1.0,                                         // 赤
        0.0, 1.0, 0.0, 1.0,                                         // 緑
        0.0, 0.0, 1.0, 1.0,                                         // 青
        1.0, 1.0, 1.0, 1.0                                          // 白
    ]);
    let texture = buildTexture(rgbaData, TEXTURE_SIZE);
                                            // テクスチャオブジェクトの生成
    bindTexture(texture, 'tex1', 0);
                          // texture を シェーダ内の uniform 変数 tex1 と結合
}

function display(time) {
setUniformFloat('theta', 0.05*time*Math.PI/180.0);
    gl.clear(gl.COLOR_BUFFER_BIT);
```

```
    gl.drawArrays(gl.TRIANGLES, 0, NUM_POINTS);
    gl.flush();
    requestAnimationFrame(display);
}

window.onload = function() {
    initSystem();
    initData();
    requestAnimationFrame(display);
};
```

```
// util.js
function buildArrayBuffer(data) {          // 頂点バッファオブジェクトの作成
    let arrayBuffer = gl.createBuffer();           // 空のオブジェクトの作成

    gl.bindBuffer(gl.ARRAY_BUFFER, arrayBuffer);      // オブジェクトを束縛

    gl.bufferData(gl.ARRAY_BUFFER,                 // 配列 data の内容を GPU へ
                  new Float32Array(data),          // コピーし、頂点バッファ
                  gl.STATIC_DRAW);                 // オブジェクトを作成

    gl.bindBuffer(gl.ARRAY_BUFFER, null);                     //  束縛を外す

    return arrayBuffer;                           // 作成したオブジェクトを戻す
}

function bindArrayBuffer(arrayBuffer, name, s) {
    gl.bindBuffer(gl.ARRAY_BUFFER, arrayBuffer);      // オブジェクトを束縛

    let location = gl.getAttribLocation(program, name);
                                    // バーテックスシェーダー内の name と同じ
                                    // 名前の attribute 変数の GPU 内の位置を取得

    gl.vertexAttribPointer(location, s, gl.FLOAT, false, 0, 0);
                                       // location の頂点バッファの属性を設定

    gl.enableVertexAttribArray(location);
                                    // location の頂点バッファを利用可能にする
}


function setUniformFloat(name, value) {
    let location = gl.getUniformLocation(program, name);
    gl.uniform1f(location, value);
}


function buildTexture(data, w) {
    let texture = gl.createTexture();         // テクスチャオブジェクトの生成

    gl.bindTexture(gl.TEXTURE_2D, texture);           // テクスチャのバインド

    // テクスチャへデータを格納
    gl.texImage2D(gl.TEXTURE_2D,                              // テクスチャの種類
                  0,                                       // 縮小テクスチャのレベル
                  gl.RGBA,                              // テクスチャデータの内部形式
                  w,                                          // テクスチャの幅
                  1,                                         // テクスチャの高さ
                  0,                                         // システムの固定値
                  gl.RGBA,                                 // 読み出すデータの形式
                  gl.FLOAT,                             // テクスチャデータの要素の形式
                  data);                                   // テクスチャ用配列データ

    // テクスチャデータの取り出し方法を指定
    gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER, gl.NEAREST);
    gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.NEAREST);

    gl.bindTexture(gl.TEXTURE_2D, null);      // テクスチャのバインドを無効化

```

```
    return texture;
}

function bindTexture(texture, name, num) {
    let location = gl.getUniformLocation(program, name);      // 場所の取得
    gl.uniform1i(location, num);                 // テクスチャに装置番号を格納

    gl.activeTexture(gl.TEXTURE0+num); // 装置番号のテクスチャをアクティブ化

    gl.bindTexture(gl.TEXTURE_2D, texture);           // テクスチャのバインド
}
```

```
1  // compileLink.js
2
3  // html ファイル中の id 名が vsid、fsid のリソースを文字列として読み込んで
4  // シェーダ実行可能コードをビルド
5  function buildProgram(vsid, fsid) {
6      let vs = compileProgram(gl.VERTEX_SHADER, vsid);
7                           // バーテックスシェーダソースプログラムのコンパイル
8      let fs = compileProgram(gl.FRAGMENT_SHADER, fsid);
9                           // フラグメントシェーダソースプログラムのコンパイル
10
11     program = gl.createProgram();                // 空の実行可能コードを作成
12
13     gl.attachShader(program, vs);
14                            // バーテックスシェーダのオブジェクトコードを接続
15     gl.attachShader(program, fs);
16                            // フラグメントシェーダのオブジェクトコードを接続
17
18     gl.linkProgram(program);                                        // リンク
19
20     if (!gl.getProgramParameter(program, gl.LINK_STATUS)){ // エラーチェック
21         alert(gl.getProgramInfoLog(program));            // エラー内容の表示
22     }
23 }
24
25 // シェーダのタイプ（バーテックス/フラグメント）とリソース id から
26 // オブジェクトコードを戻す
27 function compileProgram(type, id){
28     let source = document.getElementById(id);
29                               // HTML ファイル中から id に相当するリソースを取得
30     if (!source) { return; }                 // 取得できなければ null を返す
31
32     let shader = gl.createShader(type);          // 空のオブジェクトコードを作成
33
34     gl.shaderSource(shader, source.text);          // ソースプログラムの文字
35                                                    // 列を shader に接続
36
37     gl.compileShader(shader);                                       // コンパイル
38
39     if (gl.getShaderParameter(shader, gl.COMPILE_STATUS)){ // エラーチェック
40         return shader;         // エラーが無いならばオブジェクトコードを戻す
41     } else {
42         alert(gl.getShaderInfoLog(shader));              // エラー内容の表示
43     }
44 }
45
```

第10章　1次元テクスチャのさまざまな機能

この章では 1 次元テクスチャマッピングに関する様々な性質を紹介する。

10.1　$[0, 1]$ の範囲外のテクスチャ座標値：剰余計算

前章最後（9.3 節）において、テクスチャ座標を以下のように設定した。

```
let u_value = [
    0.0,
    1.0,
    0.5
];
```

ここで、これを

```
let u_value = [
    -1.0,                                                    // 変更
    2.0,                                                     // 変更
    0.5
];
```

へ変更してみよう。前章ではテクスチャ座標値は $[0, 1]$ の範囲を考えてきた。しかし、ここではその範囲外を与える。つまり、三角形の左下のテクスチャ座標値は -1.0、右上のそれが 2.0 である。これに伴って三角形内の各点のテクスチャ座標値は図 10.1（110 ページ）のように変更され、描画像は図 10.2（111 ページ）である。

図 9.10（102 ページ）では、赤、緑、青、白の色の帯が 1 回だけ現れたが、図 10.2 ではそれが 3 回繰り返している。この理由は、WebGL/GLSL の暗黙の設定では 任意のテクスチャ座標値 $x \in [-\infty, +\infty]$ は、剰余計算（別の呼び方で巡回計算）：

$$x' = \mod(x, 1.0)$$

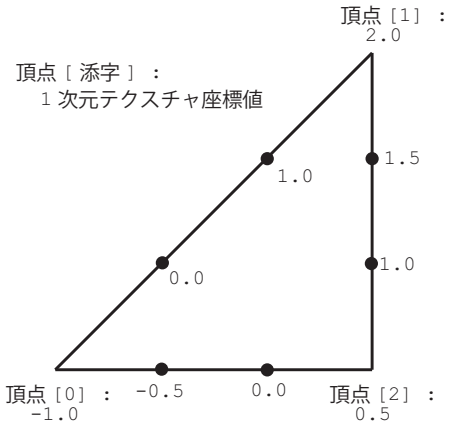


図 10.1: 三角形の各頂点に付随する attribute 変数の情報：テクスチャ座標値が $[0, 1]$ の範囲外にある場合

によって $x' \in [0, 1]$ に変換されてデータを読み込むように約束されているからである。なお、もし x が負数ならば x の小数点以下の部分に 1 を加えた数を x' とみなす計算である。x と x' の関係をグラフで図示すると図 10.3（111 ページ）の通りである。

　テクスチャマッピングはパターンを繰り返し張り付けることを意図している技術であるから、任意のテクスチャ座標値を剰余的（あるいは巡回的）に $[0, 1]$ へ写像することはまったく自然である。

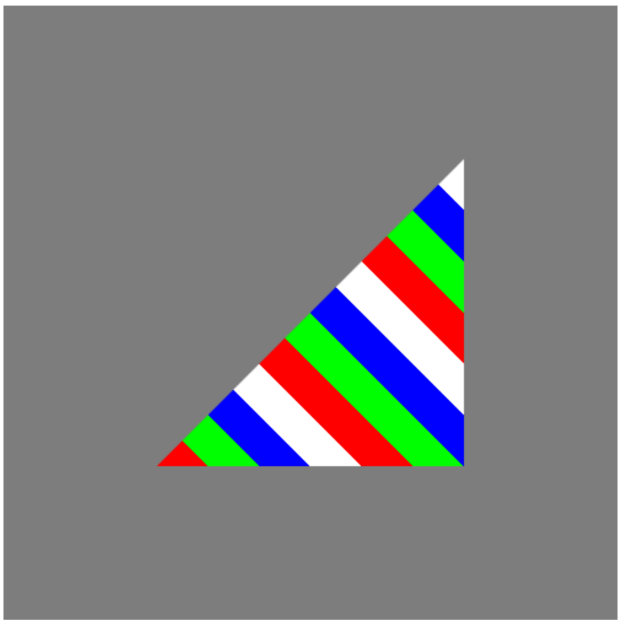


図 10.2: 画像例（その 13、テクスチャ座標値が $[0, 1]$ の範囲外の場合）

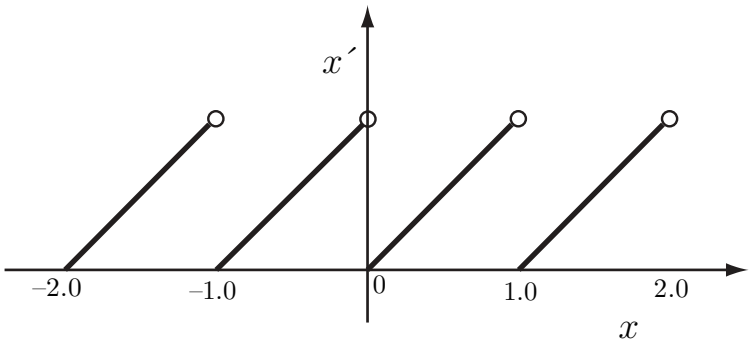


図 10.3: テクスチャ座標値の剰余計算

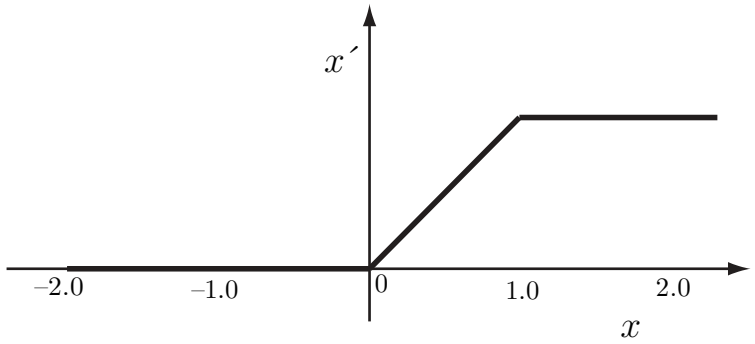


図 10.4: テクスチャ座標値の両端点固定

10.2　$[0, 1]$ の範囲外のテクスチャ座標値：端点固定

WebGL/GLSL では前節の剰余計算を $x < 0$ のときに $x' = 0$、$x > 1$ のときに $x' = 1$ に写像する計算方法も可能である。この写像をここでは端点固定と呼ぶ。x と x' の関係をグラフで図示すると図 10.4（112 ページ）である。これによって画像は図 10.5（113 ページ）のように変わる。

これを行うには、関数 `buildTexture()` を図 10.6（113 ページ）のように変更する。ポイントは、テクスチャ座標値が範囲外の場合の、以下の設定：

```
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_S, gl.CLAMP_TO_EDGE);
                                        // 範囲外を端点の値に固定する設定
```

である。ここに、三つの引数の意味は、設定中の一次元テクスチャ（`gl.TEXTURE_2D`）の S 座標方向の折り返し（`gl.TEXTURE_WRAP_S`）を両端に固定する（ `gl.CLAMP_TO_EDGE`）ことを表している。

実は、上の関数呼び出しを加えない場合には以下の設定が暗黙に仮定されている。

```
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_S, gl.REPEAT);
                                        // 範囲外を端点の値に固定する設定
```

第 3 引数に繰り返し（`gl.REPEAT`）（repeat=繰り返し）を指定している点に注意してほしい。

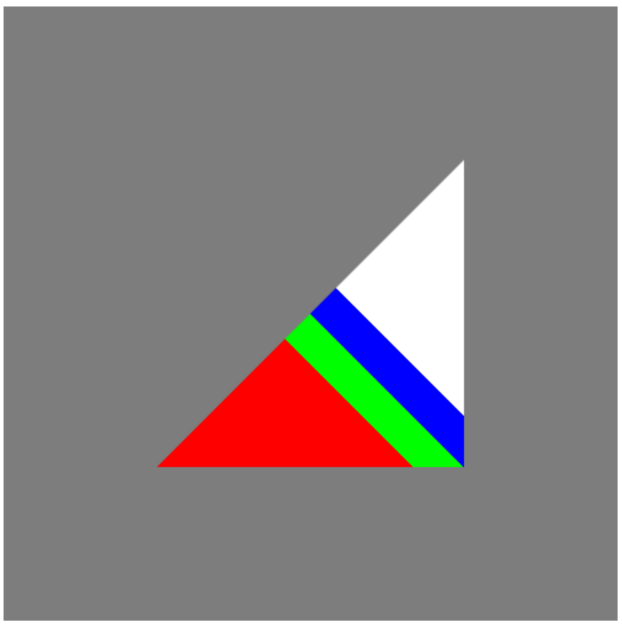


図 10.5: 画像例（その 14、範囲外のテクスチャ座標値を両端点の値にする）

```
function buildTexture(data, w) {
    let texture = gl.createTexture();


    gl.bindTexture(gl.TEXTURE_2D, texture);

    gl.texImage2D(gl.TEXTURE_2D,
                  0,
                  gl.RGBA,
                  w,
                  1,
                  0,
                  gl.RGBA,
                  gl.FLOAT,
                  data);

    gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER, gl.NEAREST);
    gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.NEAREST);

    gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_S, gl.CLAMP_TO_EDGE);
                                              // 範囲外を端点の値に固定する設定

    gl.bindTexture(gl.TEXTURE_2D, null);

    return texture;
}
```

図 10.6: 範囲外のテクスチャ座標値を両端点の値にする `buildTexture()`

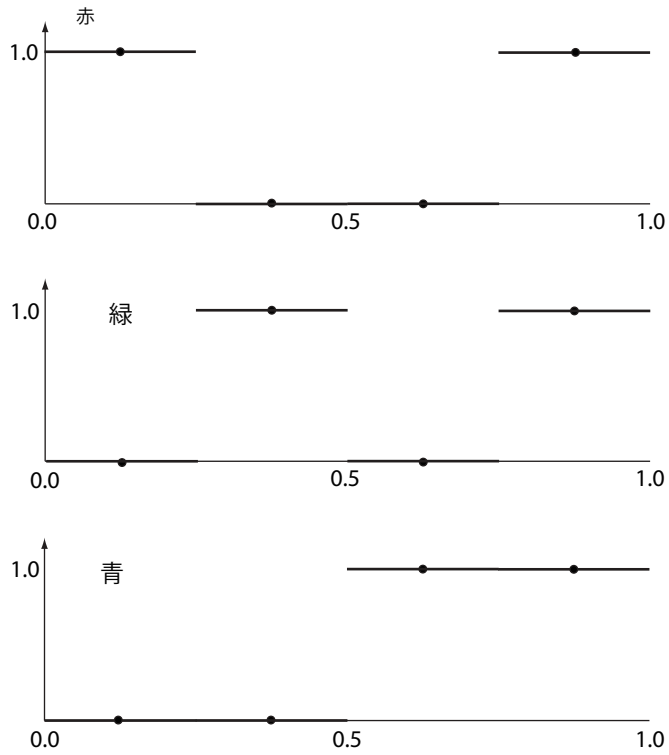


図 10.7: テクスチャ座標値とテクスチャデータ値の関係（その 3、最近傍の場合、上から赤、緑、青）

10.3　テクスチャデータの線形補間（その 1）

高品位な画像生成のためには、テクスチャを滑らかに描画する機能が必要である。GPU には離散的なテクスチャデータから線形補間で連続的に変化する RGBA 値を求めるハードウェアが搭載されている[1]。

前節まで用いてきたテクスチャ座標値とテクスチャデータ値（RGB 値）との関係は図 10.7（114 ページ）のように図示することができる。$[0, 1]$ の範囲を `TEXTURE_SIZE`（=4）に分割しており、それぞれの範囲に対応する RGB 値が読み込まれる。たとえばテクスチャ座標値が 0.4 の場合、$(r, g, b) = (0.0, 1.0, 0.0)$ である。図中の黒丸 ● はその範囲の中心点を表す。

WebGL/GLSL では、範囲の中心点（● の位置）を線分で結んだ折れ線グラフをテクスチャデータ値に用いる設定も可能である。図 10.8（115 ページ）はそのグラフである。線分で結ぶとは 2 点間のデータを線形荷重平均することであり、GPU ではこれをハードウェアで高速演算できる[2]。

この機能を利用するには、図 10.6 の関数 `buildTexture()` の中の以下の関数呼び出し：

[1] こうしてみると GPU は至るところで線形補間が可能である。
[2] 線形補間を高速演算できるが、しかし補間しない場合に比べれば計算コストがやや高くなることは避けられない。

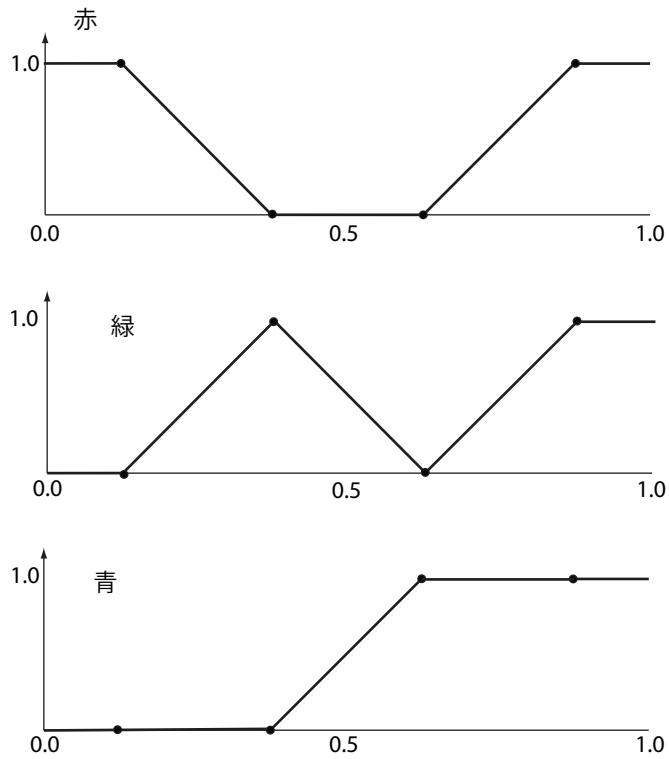


図 10.8: テクスチャ座標値とテクスチャデータ値の関係（その 4、線形補間の場合、上から赤、緑、青）

```
    gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER, gl.NEAREST);
    gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.NEAREST);
```

の第 3 引数を以下のように変える（図 10.10（116 ページ）参照）。

```
    // 第 3 引数を NEAREST から LINEAR に変更
    gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER, gl.LINEAR);
    gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.LINEAR);
```

つまり gl.NEAREST を gl.LINEAR へ変更する。これによって前節の画像（図 10.5）は図 10.9（116 ページ）のように変わる。なお、前節の画像は端点固定の設定であるから、最左の黒丸の左側（テクスチャ位置にして、$x < 0.125$）、最右の黒丸の右側（$x > 0.875$）ではそれぞれの端点の RGBA 値となる。

図 10.9: 画像例（その 15、テクスチャデータを線形補間で求める）

```
function buildTexture(data, w) {
    let texture = gl.createTexture();

    gl.bindTexture(gl.TEXTURE_2D, texture);

    gl.texImage2D(gl.TEXTURE_2D,
                  0,
                  gl.RGBA,
                  w,
                  1,
                  0,
                  gl.RGBA,
                  gl.FLOAT,
                  data);

    // 第 3 引数を NEAREST から LINEAR に変更
    gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER, gl.LINEAR);
    gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.LINEAR);

    gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_S, gl.CLAMP_TO_EDGE);

    gl.bindTexture(gl.TEXTURE_2D, null);

    return texture;
}
```

図 10.10: テクスチャ値を線形補間する `buildTexture()`

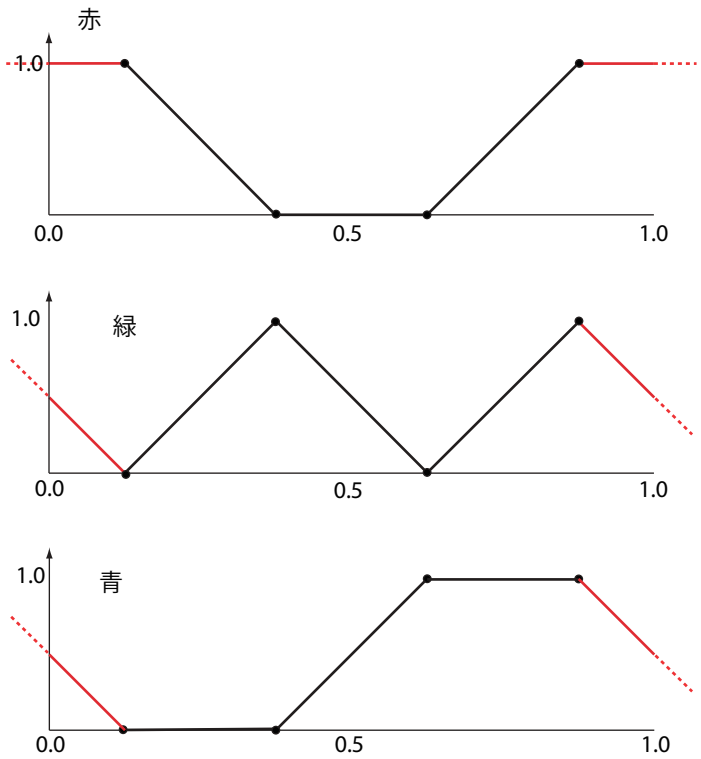


図 10.11: テクスチャ座標値とテクスチャデータ値の関係（その 5、剰余的線形補間の場合、上から赤、緑、青）

10.4　テクスチャデータの線形補間（その 2）

次に、10.2 節において導入した以下の文：

```
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_S, gl.CLAMP_TO_EDGE);
```

を削除（またはコメントアウト）する。または以下のように第 3 引数を変更する。

```
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_S, gl.REPEAT);
```

すなわち、テクスチャを剰余的に読み込む設定へ戻す。結果として、データの読み込み方法は図 10.11（117 ページ）のようになる。グラフの両端の赤い線の部分、最左の黒丸の左側（テクスチャ位置にして、$x < 0.125$）、最右の黒丸の右側（$x > 0.875$）の取り扱いが前節から変更される。

描画像は図 10.12（118 ページ）の通りである。

図 10.12: 画像例（その 16、巡回的なテクスチャデータが線形補間される）

10.5　補足：この章のプログラムのリスト

プログラム読解の便利のために、以下にこの章の最後で解説したプログラムのリストを載せる。

```
<!-- HTML -->
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<script src="./main.js" type="text/javascript"></script>
<script src="./util.js" type="text/javascript"></script>
<script src="./compileLink.js" type="text/javascript"></script>

<script id="vs" type="text/plain">
    attribute vec2 position;
    attribute float in_u;

    varying float out_u;

    uniform float theta;

    void main(void) {
        float x = cos(theta)*position.x-sin(theta)*position.y;
        float y = sin(theta)*position.x+cos(theta)*position.y;

        gl_Position = vec4(x, y, 0.0, 1.0);
        out_u = in_u;
    }
</script>

<script id="fs" type="text/plain">
    precision highp float;
    varying float out_u;

    uniform sampler2D tex1;

    void main(void)
    {
        gl_FragColor = texture2D(tex1, vec2(out_u,0.0));
    }
</script>

</head>
<body>
<canvas id="canvas"></canvas>
</body>
</html>
```

```
// main.js
let gl;
let program;

function initSystem() {
    let c = document.getElementById('canvas');
    c.width = 500; c.height = 500;

    gl = c.getContext('webgl');

    gl.clearColor(0.5, 0.5, 0.5, 1.0);

    let flg = (gl.getExtension('OES_texture_float') != null) &&
              (gl.getExtension("OES_texture_float_linear") != null);
    if (!flg) {
        alert('float texture not supported');
        return;
    }

    buildProgram('vs', 'fs');
    gl.useProgram(program);
}

let NUM_POINTS = 3;

function initData() {
    let position = [
        -0.5, -0.5,                                                    // 左下
        +0.5, +0.5,                                                    // 右上
        +0.5, -0.5                                                     // 右下
    ];

    let buffer1 = buildArrayBuffer(position);
    bindArrayBuffer(buffer1, 'position', 2);

    let u_value = [
        -1.0,                                                          // 変更
        2.0,                                                           // 変更
        0.5
    ];

    let buffer2 = buildArrayBuffer(u_value);
    bindArrayBuffer(buffer2, 'in_u', 1);

    let TEXTURE_SIZE = 4;
    let rgbaData = new Float32Array([
        1.0, 0.0, 0.0, 1.0,
        0.0, 1.0, 0.0, 1.0,
        0.0, 0.0, 1.0, 1.0,
        1.0, 1.0, 1.0, 1.0
    ]);
    let texture = buildTexture(rgbaData, TEXTURE_SIZE);
    bindTexture(texture, 'tex1', 0);

}

function display(time) {
setUniformFloat('theta', 0.05*time*Math.PI/180.0);
```

```
    gl.clear(gl.COLOR_BUFFER_BIT);
    gl.drawArrays(gl.TRIANGLES, 0, NUM_POINTS);
    gl.flush();
    requestAnimationFrame(display);
}

window.onload = function() {
    initSystem();
    initData();
    requestAnimationFrame(display);
};
```

```
// util.js
function buildArrayBuffer(data) {          // 頂点バッファオブジェクトの作成
    let arrayBuffer = gl.createBuffer();           // 空のオブジェクトの作成

    gl.bindBuffer(gl.ARRAY_BUFFER, arrayBuffer);      // オブジェクトを束縛

    gl.bufferData(gl.ARRAY_BUFFER,                // 配列 data の内容を GPU へ
                  new Float32Array(data),         // コピーし、頂点バッファ
                  gl.STATIC_DRAW);                // オブジェクトを作成

    gl.bindBuffer(gl.ARRAY_BUFFER, null);                    //  束縛を外す

    return arrayBuffer;                          // 作成したオブジェクトを戻す
}

function bindArrayBuffer(arrayBuffer, name, s) {
    gl.bindBuffer(gl.ARRAY_BUFFER, arrayBuffer);      // オブジェクトを束縛

    let location = gl.getAttribLocation(program, name);
                                    // バーテックスシェーダ内の name と同じ
                                    // 名前の attribute 変数の GPU 内の位置を取得

    gl.vertexAttribPointer(location, s, gl.FLOAT, false, 0, 0);
                                       // location の頂点バッファの属性を設定

    gl.enableVertexAttribArray(location);
                                    // location の頂点バッファを利用可能にする
}


function setUniformFloat(name, value) {
    let location = gl.getUniformLocation(program, name);
    gl.uniform1f(location, value);
}


function buildTexture(data, w) {
    let texture = gl.createTexture();


    gl.bindTexture(gl.TEXTURE_2D, texture); // テクスチャをバインドする

    // テクスチャへイメージを適用
    gl.texImage2D(gl.TEXTURE_2D,
                  0,
                  gl.RGBA,
                  w,
                  1,
                  0,
                  gl.RGBA,
                  gl.FLOAT,
                  data);

    gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER, gl.LINEAR);
    gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.LINEAR);

//    gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_S, gl.CLAMP_TO_EDGE);

```

```
    // テクスチャのバインドを無効化
    gl.bindTexture(gl.TEXTURE_2D, null);

    return texture;
}

function bindTexture(texture, name, num) {
    let location = gl.getUniformLocation(program, name);
    gl.uniform1i(location, num);
    gl.activeTexture(gl.TEXTURE0+num);
    gl.bindTexture(gl.TEXTURE_2D, texture);
}
```

```
// compileLink.js

// html ファイル中の id 名が vsid、fsid のリソースを文字列として読み込んで
// シェーダ実行可能コードをビルド
function buildProgram(vsid, fsid) {
    let vs = compileProgram(gl.VERTEX_SHADER, vsid);
                        // バーテックスシェーダソースプログラムのコンパイル
    let fs = compileProgram(gl.FRAGMENT_SHADER, fsid);
                        // フラグメントシェーダソースプログラムのコンパイル

    program = gl.createProgram();               // 空の実行可能コードを作成

    gl.attachShader(program, vs);
                         // バーテックスシェーダのオブジェクトコードを接続
    gl.attachShader(program, fs);
                         // フラグメントシェーダのオブジェクトコードを接続

    gl.linkProgram(program);                                    // リンク

    if (!gl.getProgramParameter(program, gl.LINK_STATUS)){ // エラーチェック
        alert(gl.getProgramInfoLog(program));           // エラー内容の表示
    }
}

// シェーダのタイプ（バーテックス/フラグメント）とリソース id から
// オブジェクトコードを戻す
function compileProgram(type, id){
    let source = document.getElementById(id);
                           // HTML ファイル中から id に相当するリソースを取得
    if (!source) { return; }                // 取得できなければ null を返す

    let shader = gl.createShader(type);         // 空のオブジェクトコードを作成

    gl.shaderSource(shader, source.text);          // ソースプログラムの文字
                                                   // 列を shader に接続

    gl.compileShader(shader);                                   // コンパイル

    if (gl.getShaderParameter(shader, gl.COMPILE_STATUS)){ // エラーチェック
        return shader;        // エラーが無いならばオブジェクトコードを戻す
    } else {
        alert(gl.getShaderInfoLog(shader));             // エラー内容の表示
    }
}
```

第11章　2次元テクスチャの利用

2次元テクスチャは1次元テクスチャの自然な拡張であるから、前々章、前章の議論がほぼそのまま成り立つ。この章ではそれを順に確認していく。

11.1　市松模様のマッピング

2次元テクスチャを用いた最初の例題として図11.1（126ページ）のような市松模様（チェッカーボードパターン）を三角形で張ったものを考える。前章までと同様に、この三角形がゆっくりと回転する。

以下が、そのプログラムである。これ以降、1次元テクスチャを用いることはないため、ここでは前章の1次元テクスチャに関する記述を全て2次元テクスチャに置き換えることとする。プログラムリストの分量は多いが、変更箇所はコメントのある部分のみである。

11.1.1　ホストプログラム

図11.2（127ページ）が、ホストプログラムの主要部分である。`initData()`の一部が変更されている。

頂点の座標値データは前章までと同様に設定する。

頂点の1次元テクスチャ座標値は、以下のように2次元テクスチャ座標値に変更する。

```
let uv_value = [                                    // 2次元テクスチャ座標値
    0.0, 0.0,                                                   // 左下
    1.0, 1.0,                                                   // 右上
    1.0, 0.0                                                    // 右下
];
```

テクスチャ座標の各要素値は $[0,1]$ の範囲で設定するのが基本だが、その範囲外の場合には前章と同じ議論となる。詳細は後節で述べる。3頂点の2次元座標値と2次元テクスチャ座標値の関係は図11.3（128ページ）の通りとなる。

次に、以下のようにテクスチャデータを配列に作成する。

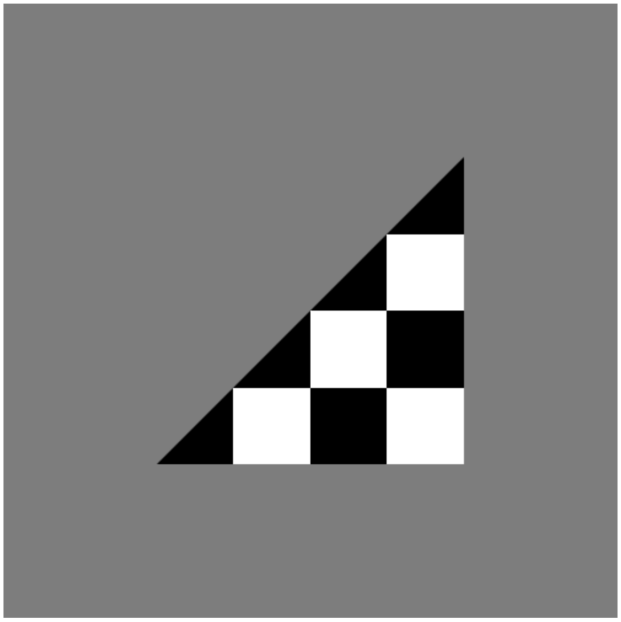


図 11.1: 画像例（その 17、2 次元テクスチャ利用の場合）

```
let TEXTURE_SIZE = 4;
rgbaData = new Float32Array(4*TEXTURE_SIZE*TEXTURE_SIZE);
for (let i = 0; i < TEXTURE_SIZE; i++) {
    for (let j = 0; j < TEXTURE_SIZE; j++) {
        let k = i+j*TEXTURE_SIZE;
        rgbaData[4*k+0] = (i+j)%2;
        rgbaData[4*k+1] = (i+j)%2;
        rgbaData[4*k+2] = (i+j)%2;
        rgbaData[4*k+3] = 1.0;
    }
}
```

1 次元テクスチャの場合（図 9.3（91 ページ）参照）、配列の全要素を列挙した。しかし、2 次元テクスチャでは要素数が多すぎるため、2 重ループで定義することとした。なお、2 次元テクスチャには 2 次元配列を用いるのが自然であるが、JavaScript の 2 次元配列は WebGL とは相性が悪い[1]ため、1 次元配列に 2 次元テクスチャデータを格納した。

2 次元テクスチャと配列の対応関係を図 11.4（128 ページ）に示す。テクスチャデータは RGBA の 4 要素からなり、その 4 要素がみだりから右へ水平方向に TEXTURE_SIZE 個（=4 個）だけ並び、その並びが下から上へ垂直方向に TEXTURE_SIZE 個（=4 個）並んで、2 次元テクスチャデータの

[1] JavaScript の 2 次元配列は、配列の配列（1 次元配列オブジェクトが配列状に並んでいるもの）という高次構造を持つ。一方、WebGL の想定する配列は C/C++の 2 次元配列（つまり、データは 1 次元的に並んでいるもの）であり、両者は整合しない。

```
let gl;
let program;

function initSystem() {
    ** 前章と同じ **
}

let NUM_POINTS = 3;

function initData() {
    let position = [
        -0.5, -0.5,                                                    // 左下
        +0.5, +0.5,                                                    // 右上
        +0.5, -0.5                                                     // 右下
    ];
    let buffer1 = buildArrayBuffer(position);
    bindArrayBuffer(buffer1, 'position', 2);

    let uv_value = [                                  // 2 次元テクスチャ座標値
        0.0, 0.0,                                                      // 左下
        1.0, 1.0,                                                      // 右上
        1.0, 0.0                                                       // 右下
    ];
    let buffer2 = buildArrayBuffer(uv_value);
    bindArrayBuffer(buffer2, 'in_uv', 2);

    let TEXTURE_SIZE = 4;
    rgbaData = new Float32Array(4*TEXTURE_SIZE*TEXTURE_SIZE);
    for (let i = 0; i < TEXTURE_SIZE; i++) {
        for (let j = 0; j < TEXTURE_SIZE; j++) {
            let k = i+j*TEXTURE_SIZE;
            rgbaData[4*k+0] = (i+j)%2;
            rgbaData[4*k+1] = (i+j)%2;
            rgbaData[4*k+2] = (i+j)%2;
            rgbaData[4*k+3] = 1.0;
        }
    }
    let texture = buildTexture(rgbaData, TEXTURE_SIZE, TEXTURE_SIZE);
    bindTexture(texture, 'tex1', 0);
}

function display(time) {
    ** 前章と同じ **
}

window.onload = function() {
    ** 前章と同じ **
};
```

図 11.2: 縞模様のテクスチャを描画する JavaScript ホストプログラム `main.js`

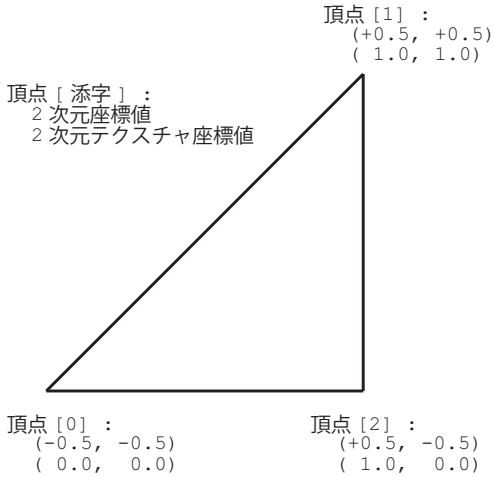


図 11.3: 三角形に 2 次元テクスチャを張る場合の動作例

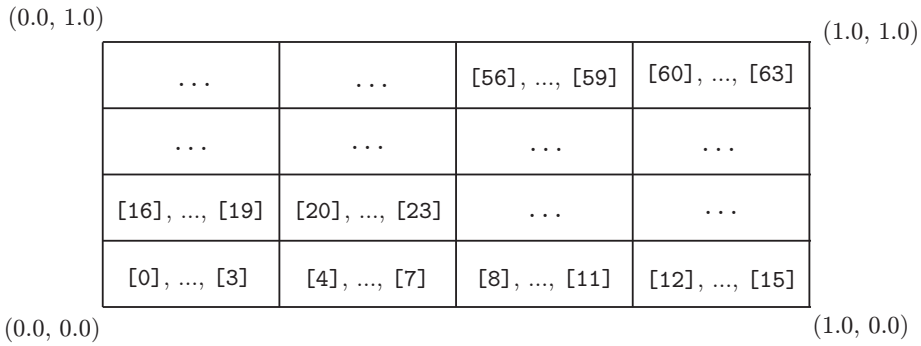


図 11.4: 2 次元テクスチャ空間における配列データの並び

全体を構成している。

　完成した 2 次元テクスチャデータは以下によってシェーダプログラムと結合される。

```
let texture = buildTexture(rgbaData, TEXTURE_SIZE, TEXTURE_SIZE);
bindTexture(texture, 'tex1', 0);
```

これは 1 次元テクスチャの場合と同様であるが、buildTexture() の第 3 引数にテクスチャデータの高さ方向の大きさを指定する点が 1 次元の場合と異なる。

　テクスチャオブジェクトを生成するメソッド buildTexture() は図 11.5（129 ページ）のように、高さに関する指定を 1 から h に変更するだけである。テクスチャオブジェクトをシェーダプログラムに結合するメソッド bindexture() は前章と同じである。

```
function buildTexture(data, w, h) {                                    // h を追加
    let texture = gl.createTexture();

    gl.bindTexture(gl.TEXTURE_2D, texture);

    // テクスチャへイメージを適用
    gl.texImage2D(gl.TEXTURE_2D,
                  0,
                  gl.RGBA,
                  w,
                  h,                                                   // 1 を h に置換
                  0,
                  gl.RGBA,
                  gl.FLOAT,
                  data);

    gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER, gl.NEAREST);
    gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.NEAREST);

    gl.bindTexture(gl.TEXTURE_2D, null);

    return texture;
}

function bindTexture(texture, name, num) {
    ** 前章と同じ **
}
```

図 11.5: 2 次元テクスチャの操作に関するメソッド：util.js を変更

11.1.2　バーテックスシェーダプログラム

バーテックスシェーダプログラムは図 11.6（130 ページ）の前半の通りである。前章では 1 次元テクスチャ座標値を attribute 変数 in_u で受け渡したが、ここでは 2 次元テクスチャ座標値を attribute 変数を in_uv で受け取り、そのままラスタライザへ受け渡す。そしてラスタライザは線形補間された 2 次元テクスチャ座標値を各画素のフラグメントシェーダへ受け渡す。

11.1.3　フラグメントシェーダプログラム

フラグメントシェーダプログラムは図 11.6（130 ページ）の後半の通りである。

画素の RGB 値の代入文：

```
        gl_FragColor = texture2D(tex1, out_uv);
```

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<script src="./main.js" type="text/javascript"></script>
<script src="./util.js" type="text/javascript"></script>
<script src="./compileLink.js" type="text/javascript"></script>

<script id="vs" type="text/plain">
    attribute vec2 position;
    attribute vec2 in_uv;

    varying vec2 out_uv;

    uniform float theta;

    void main(void) {
        float x = cos(theta)*position.x-sin(theta)*position.y;
        float y = sin(theta)*position.x+cos(theta)*position.y;

        gl_Position = vec4(x, y, 0.0, 1.0);
        out_uv = in_uv;
    }
</script>

<script id="fs" type="text/plain">
    precision highp float;
    varying vec2 out_uv;

    uniform sampler2D tex1;

    void main(void)
    {
        gl_FragColor = texture2D(tex1, out_uv);
    }
</script>

</head>
<body>
<canvas id="canvas"></canvas>
</body>
</html>
```

図 11.6: 市松模様のテクスチャを描画する HTML/シェーダプログラム

の右辺 texture2D(tex1, out_uv) は、テクスチャデータ tex1 から 2 次元座標値 out_uv の位置の RGBA データを求める関数呼び出しである。

座標値と色の対応関係は、既に述べたように、図 9.5（93 ページ）の通りである。

11.1.4　実行の様子

プログラムの実行の概念図が図 11.7（132 ページ）である。

バッファからバーテックスシェーダに入力される 2 次元座標値はそのままラスタライザへ受け渡される。ラスタライザは、3 頂点からなる三角形の内側の全ての画素点について 2 次元座標値を線形補間する。フラグメントシェーダは、uniform 変数に格納されている 2 次元テクスチャデータ tex2 の中から座標値に対応する RGBA 値を読み込み、それをそのままフレームバッファへ書き込む。

次節以降では、前章で 1 次元テクスチャに行った変更を 2 次元テクスチャでも試す。

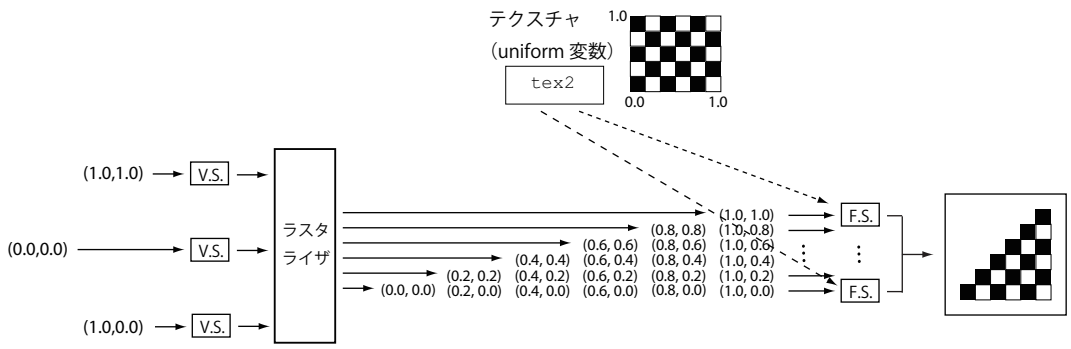


図 11.7: 三角形に 2 次元テクスチャを張る場合の動作例

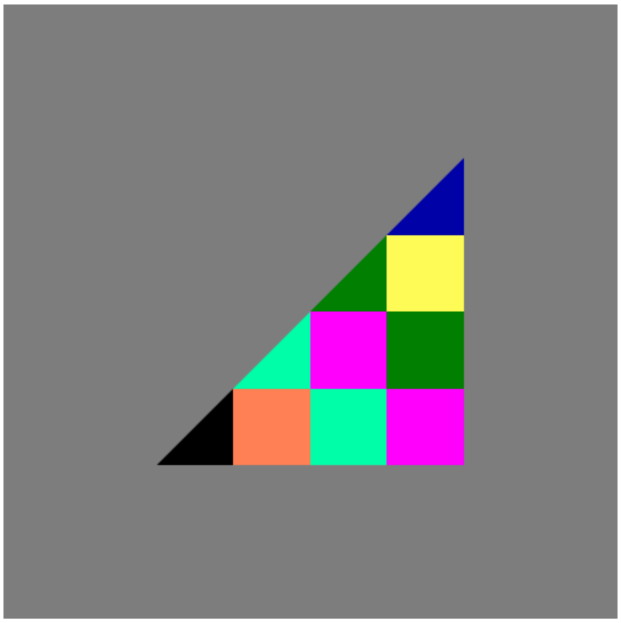


図 11.8: 画像例（その 19、カーラーテクスチャの利用）

11.2　カーラーデータの利用

次に、図 11.2（127 ページ）のテクスチャデータの設定部分：

```
    for (let i = 0; i < TEXTURE_SIZE; i++) {
        for (let j = 0; j < TEXTURE_SIZE; j++) {
            let k = i+j*TEXTURE_SIZE;
            rgbaData[4*k+0] = (i+j)%2;
            rgbaData[4*k+1] = (i+j)%2;
            rgbaData[4*k+2] = (i+j)%2;
            rgbaData[4*k+3] = 1.0;
        }
    }
```

を

```
    for (let i = 0; i < TEXTURE_SIZE; i++) {
        for (let j = 0; j < TEXTURE_SIZE; j++) {
            let k = i+j*TEXTURE_SIZE;
            rgbaData[4*k+0] = (i+j)%2;
            rgbaData[4*k+1] = ((i+j)%3)/2;
            rgbaData[4*k+2] = ((i+j)%4)/3;
            rgbaData[4*k+3] = 1.0;
        }
    }
```

へ変更する。右辺の式に結果、図 11.8（133 ページ）の描画像が得られる。

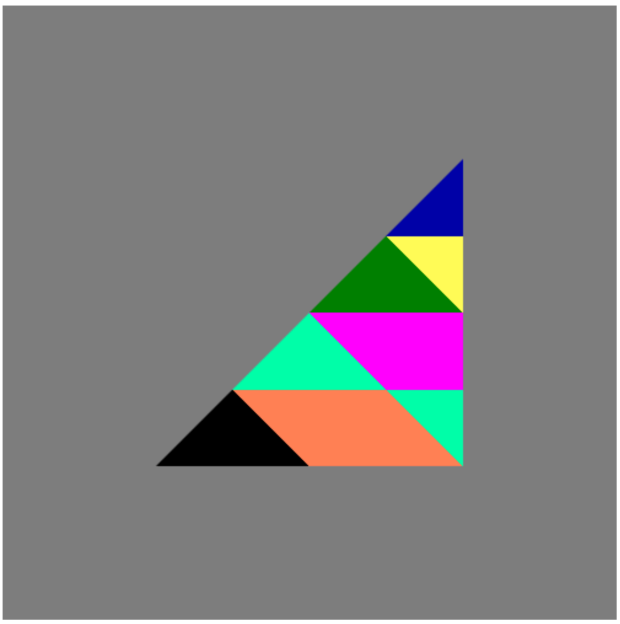


図 11.9: 画像例（その 20、テクスチャを傾斜させる変更）

11.3　テクスチャの傾斜

次に、図 11.2（127 ページ）のプログラムのテクスチャ座標値の設定：

```
    let uv_value = [                                    // 2 次元テクスチャ座標値
        0.0, 0.0,                                                   // 左下
        1.0, 1.0,                                                   // 右上
        1.0, 0.0                                                    // 右下
    ];
```

を

```
    let uv_value = [
        0.0, 0.0,
        1.0, 1.0,
        0.5, 0.0
    ];
```

へ変更してみよう。これによって三角形の各画素が参照するテクスチャデータの位置が変わるから、画像は図 11.9（135 ページ）のように変わる。

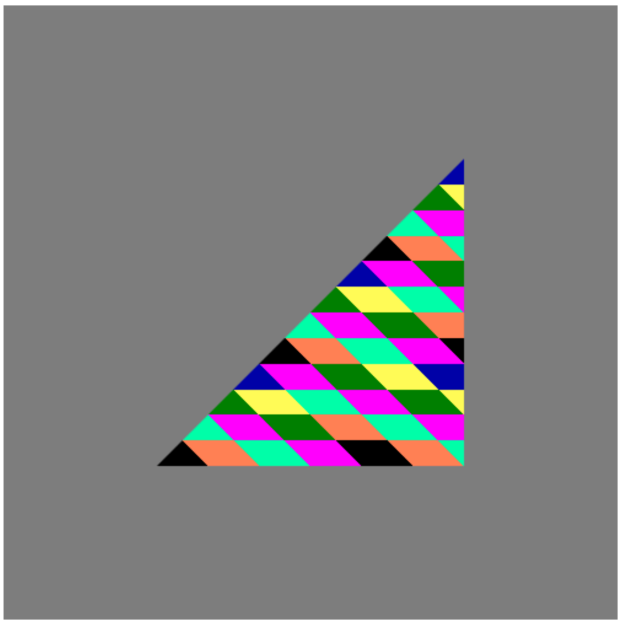


図 11.10: 画像例（その 21、テクスチャ座標値が $[0,1] \times [0,1]$ の範囲外の場合）

11.4　$[0,1]$ の範囲外のテクスチャ座標値：剰余計算

さらにプログラムを改造する。

前節の

```
    let uv_value = [
        0.0, 0.0,
        1.0, 1.0,
        0.5, 0.0
    ];
```

を

```
    let uv_value = [
        -1.0, -1.0,
        2.0, 2.0,
        0.5, -1.0
    ];
```

へ変更する。これによってテクスチャ座標値が $[0,1] \times [0,1]$ の範囲外になる。描画された画像は図 11.10（136 ページ）である。

1　既に 1 次元テクスチャの説明で述べたが、テクスチャ座標値は剰余計算によって $[0, 1]$ の範囲に
2 変換される（図 10.3（111 ページ）参照）。2 次元テクスチャの個々の座標値についても同様である。

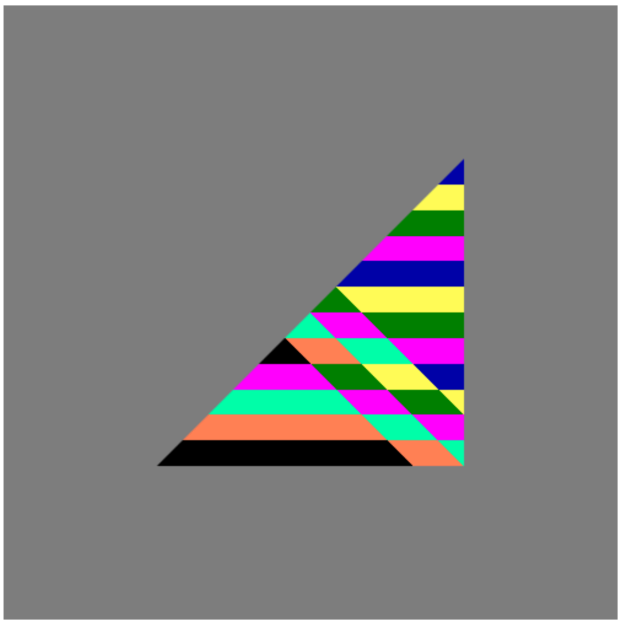


図 11.11: 画像例（その 22、範囲外のテクスチャ座標値を 1 次元目について両端点の値にする）

11.5　$[0, 1]$ の範囲外のテクスチャ座標値：端点固定

前章において、$[0, 1]$ の範囲の取り扱い法には剰余計算による方法と端点値に固定する方法の 2 種類があることを説明した[2]。

端点固定を行うには、まず前章同様に、関数 `buildTexture()` に以下を加えればよい。

```
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_S, gl.CLAMP_TO_EDGE);
```

これによって描画像は図 11.11（138 ページ）のように変わる。分かりにくいが、左下から右上への斜め 45 度の方向が 1 次元目、垂直方向が 2 次元目である。図 11.10（136 ページ）と図 11.11（138 ページ）を比較すると、斜め 45 度方向ではパターンの繰り返し（剰余計算）が無くなっている。しかし垂直方向には繰り返しが見られる。

上の関数呼び出し中のマクロ定数 `gl.TEXTURE_WRAP_S` は、2 次元テクスチャの 1 次元目の座標を指定するパラメータである。2 次元座標系に用いる座標名には xy、uv、st などが用いられるが、マクロ定数 `gl.TEXTURE_WRAP_S` の末尾の `S` が st 座標系の 1 次元目を表すものである。

[2]細かく分けると、実は全部で 3〜5 種類（OpenGL のバージョン、種類によって異なる）があるが、ここでは代表的な二つを紹介する。

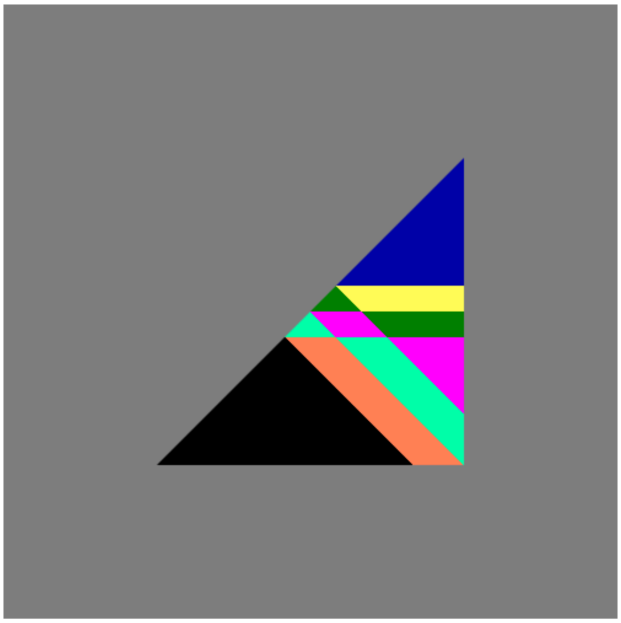


図 11.12: 画像例（その 23、範囲外のテクスチャ座標値を二つの次元とも両端点の値にする）

2 次元目の座標に関するマクロ定数は gl.TEXTURE_WRAP_T である。末尾の T が 2 次元目を表している[3]。そこで、関数 buildTexture() に

```
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_T, gl.CLAMP_TO_EDGE);
```

を加えると、その結果は図 11.12（139 ページ）である。2 次元目の方向である垂直方向でもパターンの繰り返しが無くなっている。

[3]ちなみに、OpenGL では 3 次元テクスチャの 3 番目の座標については R を用いる。S、T の次は U になりそうだが、U は u-v 座標系の座標名と混乱するから避けたのだろう。

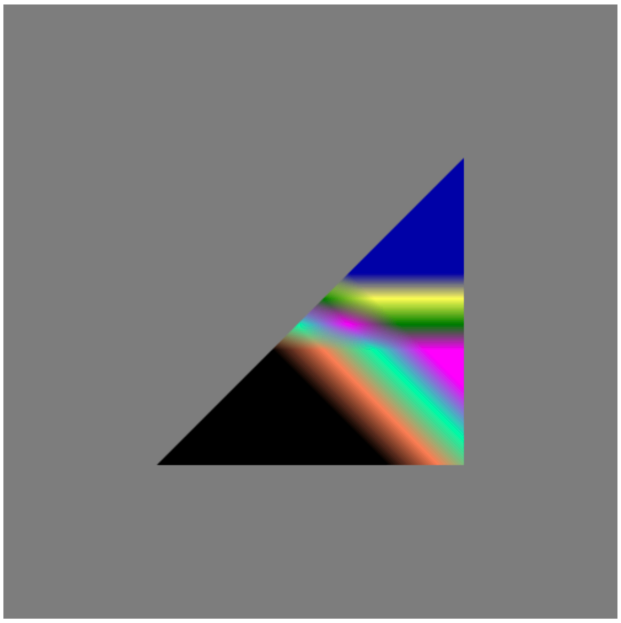


図 11.13: 画像例（その 24、テクスチャデータを線形補間で求める）

11.6　テクスチャデータの線形補間（その 3）

1 次元テクスチャの場合と同様に、関数 buildTexture() の中の設定：

```
    gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER, gl.NEAREST);
    gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.NEAREST);
```

の第 3 引数を以下のように変えてみる。

```
    gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER, gl.LINEAR);
    gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.LINEAR);
```

3 番目の引数を gl.NEAREST から gl.LINEAR へ変更したことに注意する。これによって画像は図 11.13（140 ページ）のように変わる。

GPU ではテクスチャについての線形補間をハードウェアで高速演算できる。

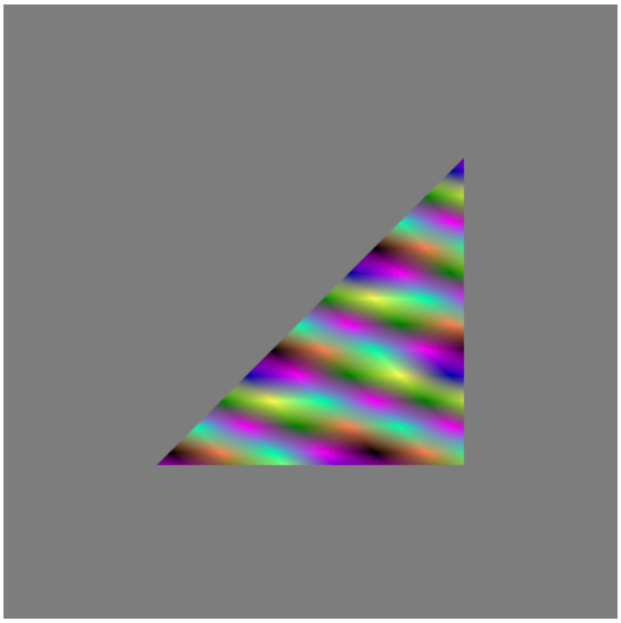


図 11.14: 画像例（その 25、巡回的なテクスチャデータを線形補間で求める）

11.7　テクスチャデータの線形補間（その 4）

次に、11.5 節において導入した以下の文：

```
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_S, gl.CLAMP_TO_EDGE);
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_T, gl.CLAMP_TO_EDGE);
```

を削除（またはコメントアウト）する。なお、上の一文を削除することは、明示的には以下の指定と同様である

```
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_S, gl.REPEAT);
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_T, gl.REPEAT);
```

これによって端点固定がふたたび剰余計算に戻る。結果、図 11.14（141 ページ）を得る。

11.8　補足：この章のプログラムのリスト

プログラム読解の便利のために、以下にこの章の最後で解説したプログラムのリストを載せる。

```
<!-- HTML -->
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<script src="./main.js" type="text/javascript"></script>
<script src="./util.js" type="text/javascript"></script>
<script src="./compileLink.js" type="text/javascript"></script>

<script id="vs" type="text/plain">
    attribute vec2 position;
    attribute vec2 in_uv;

    varying vec2 out_uv;

    uniform float theta;

    void main(void) {
        float x = cos(theta)*position.x-sin(theta)*position.y;
        float y = sin(theta)*position.x+cos(theta)*position.y;

        gl_Position = vec4(x, y, 0.0, 1.0);
        out_uv = in_uv;
    }
</script>

<script id="fs" type="text/plain">
    precision highp float;
    varying vec2 out_uv;

    uniform sampler2D tex1;

    void main(void) {
        gl_FragColor = texture2D(tex1, out_uv);
    }
</script>

</head>
<body>
<canvas id="canvas"></canvas>
</body>
</html>
```

```
// main.js
let gl;
let program;

function initSystem() {
    let c = document.getElementById('canvas');
    c.width = 500; c.height = 500;

    gl = c.getContext('webgl');

    gl.clearColor(0.5, 0.5, 0.5, 1.0);

    let flg = (gl.getExtension('OES_texture_float') != null) &&
              (gl.getExtension('OES_texture_float_linear') != null);
    if (!flg) {
        alert('float texture not supported');
        return;
    }

    buildProgram('vs', 'fs');
    gl.useProgram(program);
}

let NUM_POINTS = 3;

function initData() {
    let position = [
        -0.5, -0.5,                                                    // 左下
        +0.5, +0.5,                                                    // 右上
        +0.5, -0.5                                                     // 右下
    ];

    let buffer1 = buildArrayBuffer(position);
    bindArrayBuffer(buffer1, 'position', 2);

    let uv_value = [
        -1.0, -1.0,
        2.0, 2.0,
        0.5, -1.0
    ];

    let buffer2 = buildArrayBuffer(uv_value);
    bindArrayBuffer(buffer2, 'in_uv', 2);

    let TEXTURE_SIZE = 4;
    rgbaData = new Float32Array(4*TEXTURE_SIZE*TEXTURE_SIZE);
    for (let i = 0; i < TEXTURE_SIZE; i++) {
        for (let j = 0; j < TEXTURE_SIZE; j++) {
            let k = i+j*TEXTURE_SIZE;
            rgbaData[4*k+0] = ((i+j)%2);
            rgbaData[4*k+1] = ((i+j)%3)/2;
            rgbaData[4*k+2] = ((i+j)%4)/3;
            rgbaData[4*k+3] = 1.0;
        }
    }
    let texture = buildTexture(rgbaData, TEXTURE_SIZE, TEXTURE_SIZE);
    bindTexture(texture, 'tex1', 0);

```

```
}

function display(time) {
setUniformFloat('theta', 0.05*time*Math.PI/180.0);
    gl.clear(gl.COLOR_BUFFER_BIT);
    gl.drawArrays(gl.TRIANGLES, 0, NUM_POINTS);
    gl.flush();
    requestAnimationFrame(display);
}

window.onload = function() {
    initSystem();
    initData();
    requestAnimationFrame(display);
};
```

```
// util.js
function buildArrayBuffer(data) {          // 頂点バッファオブジェクトの作成
    let arrayBuffer = gl.createBuffer();           // 空のオブジェクトの作成

    gl.bindBuffer(gl.ARRAY_BUFFER, arrayBuffer);      // オブジェクトを束縛

    gl.bufferData(gl.ARRAY_BUFFER,                // 配列 data の内容を GPU へ
                  new Float32Array(data),         // コピーし、頂点バッファ
                  gl.STATIC_DRAW);                // オブジェクトを作成

    gl.bindBuffer(gl.ARRAY_BUFFER, null);                    //  束縛を外す

    return arrayBuffer;                          // 作成したオブジェクトを戻す
}

function bindArrayBuffer(arrayBuffer, name, s) {
    gl.bindBuffer(gl.ARRAY_BUFFER, arrayBuffer);      // オブジェクトを束縛

    let location = gl.getAttribLocation(program, name);
                                    // バーテックスシェーダ内の name と同じ
                                    // 名前の attribute 変数の GPU 内の位置を取得

    gl.vertexAttribPointer(location, s, gl.FLOAT, false, 0, 0);
                                       // location の頂点バッファの属性を設定

    gl.enableVertexAttribArray(location);
                                    // location の頂点バッファを利用可能にする
}


function setUniformFloat(name, value) {
    let location = gl.getUniformLocation(program, name);
    gl.uniform1f(location, value);
}


function buildTexture(data, w, h) {
    let texture = gl.createTexture();


    gl.bindTexture(gl.TEXTURE_2D, texture); // テクスチャをバインドする

    // テクスチャへイメージを適用
    gl.texImage2D(gl.TEXTURE_2D,
                  0,
                  gl.RGBA,
                  w,
                  h,
                  0,
                  gl.RGBA,
                  gl.FLOAT,
                  data);

    gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER, gl.LINEAR);
    gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.LINEAR);

//    gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_S, gl.CLAMP_TO_EDGE);
//    gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_T, gl.CLAMP_TO_EDGE);
```

```
1
2       // テクスチャのバインドを無効化
3       gl.bindTexture(gl.TEXTURE_2D, null);
4
5       return texture;
6   }
7
8   function bindTexture(texture, name, num) {
9       let location = gl.getUniformLocation(program, name);
10      gl.uniform1i(location, num);
11      gl.activeTexture(gl.TEXTURE0+num);
12      gl.bindTexture(gl.TEXTURE_2D, texture);
13  }
```

```
// compileLink.js

// html ファイル中の id 名が vsid、fsid のリソースを文字列として読み込んで
// シェーダ実行可能コードをビルド
function buildProgram(vsid, fsid) {
    let vs = compileProgram(gl.VERTEX_SHADER, vsid);
                          // バーテックスシェーダソースプログラムのコンパイル
    let fs = compileProgram(gl.FRAGMENT_SHADER, fsid);
                          // フラグメントシェーダソースプログラムのコンパイル

    program = gl.createProgram();                // 空の実行可能コードを作成

    gl.attachShader(program, vs);
                           // バーテックスシェーダのオブジェクトコードを接続
    gl.attachShader(program, fs);
                           // フラグメントシェーダのオブジェクトコードを接続

    gl.linkProgram(program);                                        // リンク

    if (!gl.getProgramParameter(program, gl.LINK_STATUS)){ // エラーチェック
        alert(gl.getProgramInfoLog(program));           // エラー内容の表示
    }
}

// シェーダのタイプ（バーテックス/フラグメント）とリソース id から
// オブジェクトコードを戻す
function compileProgram(type, id){
    let source = document.getElementById(id);
                             // HTML ファイル中から id に相当するリソースを取得
    if (!source) { return; }                // 取得できなければ null を返す

    let shader = gl.createShader(type);          // 空のオブジェクトコードを作成

    gl.shaderSource(shader, source.text);          // ソースプログラムの文字
                                                   // 列を shader に接続

    gl.compileShader(shader);                                       // コンパイル

    if (gl.getShaderParameter(shader, gl.COMPILE_STATUS)){ // エラーチェック
        return shader;        // エラーが無いならばオブジェクトコードを戻す
    } else {
        alert(gl.getShaderInfoLog(shader));             // エラー内容の表示
    }
}
```

第12章　テクスチャを用いた簡単な計算

GPU による描画処理では多数のバーテックスシェーダおよび多数のフラグメントシェーダが並列実行する。しかもシェーダプログラムはプログラミング可能であるから、それをうまく活用すれば描画に限らず様々な並列計算ができるのではないかと期待できる。

ここでのポイントはテクスチャの利用である。テクスチャは配列のように扱うことができる。事実、テクスチャデータは配列上に作成したから、テクスチャを用いた演算で配列演算も可能と予想される。この章では簡単な計算プログラムでそれを確認する。この章の内容が、最近の GPGPU につながっていく。

12.1　テクスチャを用いた計算

この節では、同じ形状（同じ縦横サイズ）の 2 次元テクスチャを 2 枚用意し、それらテクスチャ間の対応する画素の加算を行い、結果をフレームバッファに描画するプログラムを考える。

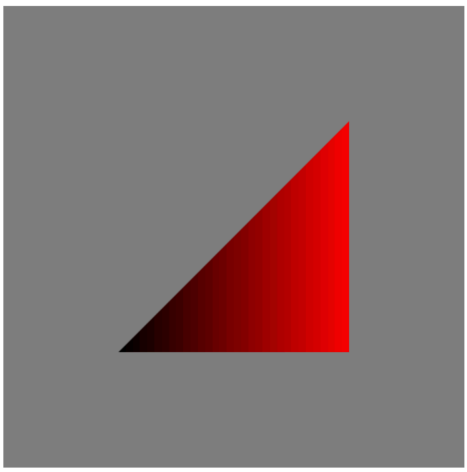
まず図 12.1（149 ページ）(a)、(b) がそれぞれの 2 次元テクスチャを前章の方法で描画したものである。そして図 12.1（149 ページ）(c) がテクスチャの加算を実行したものである。この加算はフラグメントシェーダの上で実行する。つまり、フラグメントシェーダで 2 枚のテクスチャのデータを取得し、加算し、その結果をフレームバッファに出力する。これは加算の並列計算である。

12.1.1　ホストプログラム

図 12.2（150 ページ）、図 12.3（151 ページ）がホストプログラムである。2 枚のテクスチャを用いる点が主な変更箇所である。

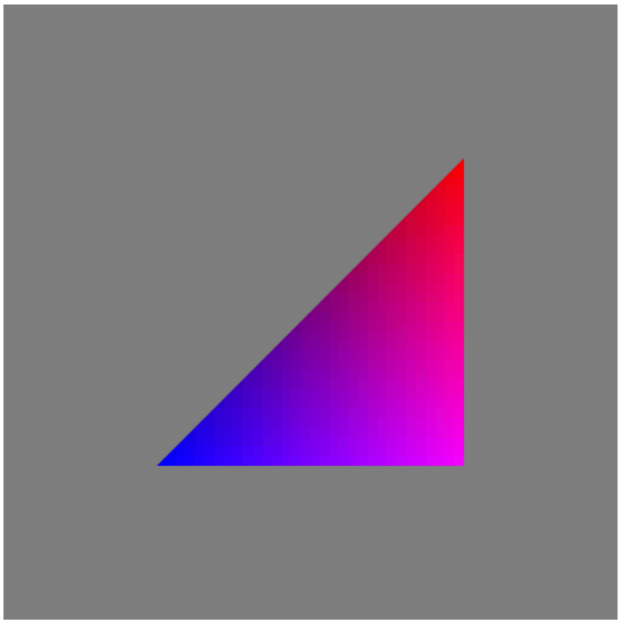
図 12.1（149 ページ）(a) の三角形に貼られたテクスチャは図 12.2（150 ページ）の関数 `initData()` 内の以下の部分で作成する。

```
rgbaData1[4*k+0] = i/TEXTURE_SIZE;
rgbaData1[4*k+1] = 0;
rgbaData1[4*k+2] = 0;
rgbaData1[4*k+3] = 1.0;
```



(a) テクスチャ1

(b) テクスチャ2



(c) (a) と (b) の加算

図 12.1: 二つのテクスチャの加算

```
// main.js
let gl;
let program;

function initSystem() {
  ** 前章と同じ **
}

let NUM_POINTS = 3;

function initData() {
    let position = [
        -0.5, -0.5,                                                    // 左下
        +0.5, +0.5,                                                    // 右上
        +0.5, -0.5                                                     // 右下
    ];
    let buffer1 = buildArrayBuffer(position);
    bindArrayBuffer(buffer1, 'position', 2);

    let uv_value = [
        0.0, 0.0,
        1.0, 1.0,
        1.0, 0.0
    ];
    let buffer2 = buildArrayBuffer(uv_value);
    bindArrayBuffer(buffer2, 'in_uv', 2);
```

図 12.2: テクスチャの加算と描画を行う JavaScript ホストプログラム `main.js`（その 1、その 2 へ続く）

つまり R（赤）の輝度値を水平方向（左から右へ `i` の増える方向）にグラデーションした画像である。図 12.1（149 ページ）(b) の三角形に貼られたテクスチャは図 12.2（150 ページ）の以下である。

```
            rgbaData1[4*k+0] = i/TEXTURE_SIZE;
            rgbaData1[4*k+1] = 0;
            rgbaData1[4*k+2] = 0;
            rgbaData1[4*k+3] = 1.0;
```

つまり B（青）の輝度値を垂直方向（上から下へ `j` の減る方向）にグラデーションした画像である。

テクスチャオブジェクトはその直後に作り、シェーダプログラムと結合する。

```
    let TEXTURE_SIZE = 32;
    let rgbaData1 = new Float32Array(4*TEXTURE_SIZE*TEXTURE_SIZE);
    let rgbaData2 = new Float32Array(4*TEXTURE_SIZE*TEXTURE_SIZE);

    for (let i = 0; i < TEXTURE_SIZE; i++) {
        for (let j = 0; j < TEXTURE_SIZE; j++) {
            let k = i+j*TEXTURE_SIZE;
            rgbaData1[4*k+0] = i/TEXTURE_SIZE;
            rgbaData1[4*k+1] = 0;
            rgbaData1[4*k+2] = 0;
            rgbaData1[4*k+3] = 1.0;

            rgbaData2[4*k+0] = 0;
            rgbaData2[4*k+1] = 0;
            rgbaData2[4*k+2] = 1.0-j/TEXTURE_SIZE;
            rgbaData2[4*k+3] = 1.0;
        }
    }
    let texture1 = buildTexture(rgbaData1, TEXTURE_SIZE, TEXTURE_SIZE);
    let texture2 = buildTexture(rgbaData2, TEXTURE_SIZE, TEXTURE_SIZE);

    bindTexture(texture1, 'tex1', 0);
    bindTexture(texture2, 'tex2', 1);
}

function display(time) {
    ** 前章と同じ **
}

window.onload = function() {
    ** 前章と同じ **
};
```

図 12.3: テクスチャの加算と描画を行う JavaScript ホストプログラム `main.js`（その 2）

```
    texture1 = buildTexture(rgbaData1, TEXTURE_SIZE, TEXTURE_SIZE);
    texture2 = buildTexture(rgbaData2, TEXTURE_SIZE, TEXTURE_SIZE);

    bindTexture(texture1, 'tex1', 0);
    bindTexture(texture2, 'tex2', 1);
```

12.1.2　バーテックスシェーダプログラム

図 12.4（152 ページ）がシェーダプログラムである。前半部のバーテックスシェーダプログラムは前章と同じである。

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<script src="./main.js" type="text/javascript"></script>
<script src="./util.js" type="text/javascript"></script>
<script src="./compileLink.js" type="text/javascript"></script>

<script id="vs" type="text/plain">
    attribute vec2 position;
    attribute vec2 in_uv;

    varying vec2 out_uv;

    uniform float theta;

    void main(void) {
        float x = cos(theta)*position.x-sin(theta)*position.y;
        float y = sin(theta)*position.x+cos(theta)*position.y;

        gl_Position = vec4(x, y, 0.0, 1.0);
        out_uv = in_uv;
    }
</script>

<script id="fs" type="text/plain">
    precision highp float;
    varying vec2 out_uv;

    uniform sampler2D tex1;
    uniform sampler2D tex2;

    void main(void)
    {
        vec4 data1 = texture2D(tex1,out_uv);
        vec4 data2 = texture2D(tex2,out_uv);

        gl_FragColor = data1+data2;
    }
</script>

</head>
<body>
<canvas id="canvas"></canvas>
</body>
</html>
```

図 12.4: テクスチャの加算と描画を行う HTML/シェーダプログラム

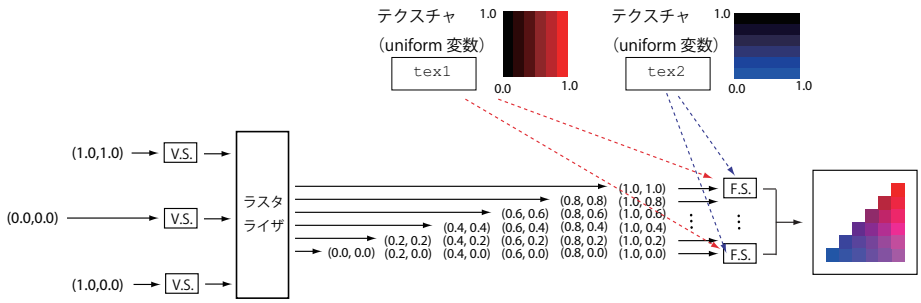


図 12.5: テクスチャの加算と描画を行うシェーダの動作例

12.1.3　フラグメントシェーダプログラム

図 12.4（152 ページ）の後半部のフラグメントシェーダプログラムでは、まず、2 枚のテクスチャを uniform 変数としてフラグメントシェーダから参照できるようにする。

```
uniform sampler2D tex1;
uniform sampler2D tex2;
```

次に、実際のテクスチャからデータを取り出す。

```
vec4 data1 = texture2D(tex1,out_uv);
vec4 data2 = texture2D(tex2,out_uv);
```

そして、以下の代入文でデータ間の加算演算を行い、画像として出力する。

```
gl_FragColor = data1+data2;
```

12.1.4　シェーダプログラムの実行の様子

図 12.5（153 ページ）に、この節のプログラムの実行の様子を示す。2 枚のテクスチャから読み まれた値がフラグメントシェーダで加算されて、出力される。

重要なのは、フラグメントシェーダは並列に実行される点である。GPU 内の多数のプロセッサで独立して並列に実行されるため、ホスト側の CPU で繰り返し実行される場合の数倍、数十倍の高速実行ができると期待される。もちろん、実際に高速実行を実現にするには様々な問題をクリアせねばならない。次節以降に踏み込んで議論を進めていく。

```
function initSystem() {
    /* ここに WebGL その他の初期化処理 */
}

function initData() {
    /* ここにプログラムで利用するデータの初期化処理 */
}

function compute() {                                    // display() から変更
   /* ここに計算処理 */
}

function showResults() {                                  // 新たに導入
   /* ここに計算結果の出力等の処理 */
}

window.onload = function(){
    initSystem();
    initData();
    compute();
    showResults();
};
```

図 12.6: GPGPU のためのホストプログラムのひな形

12.2　プログラムの外形の変更

ここから最終章までは処理を計算に特化し、描画は一切行わない。そこで、3.4 項（11 ページ）の図 3.2（11 ページ）で設定したプログラムの全体構成を図 12.6（154 ページ）のように変更する。ここに、

- `initSystem()`、`initData()` は、これまでとほぼ同じ役割を担う関数となる。
- `compute()` は、これまでの `display()` と同様に、GPU の起動直前の設定および起動を行うが、描画は行わないため、関数名を変えた。
- `showResults()` は、計算結果を表示するために新たに導入する関数である。

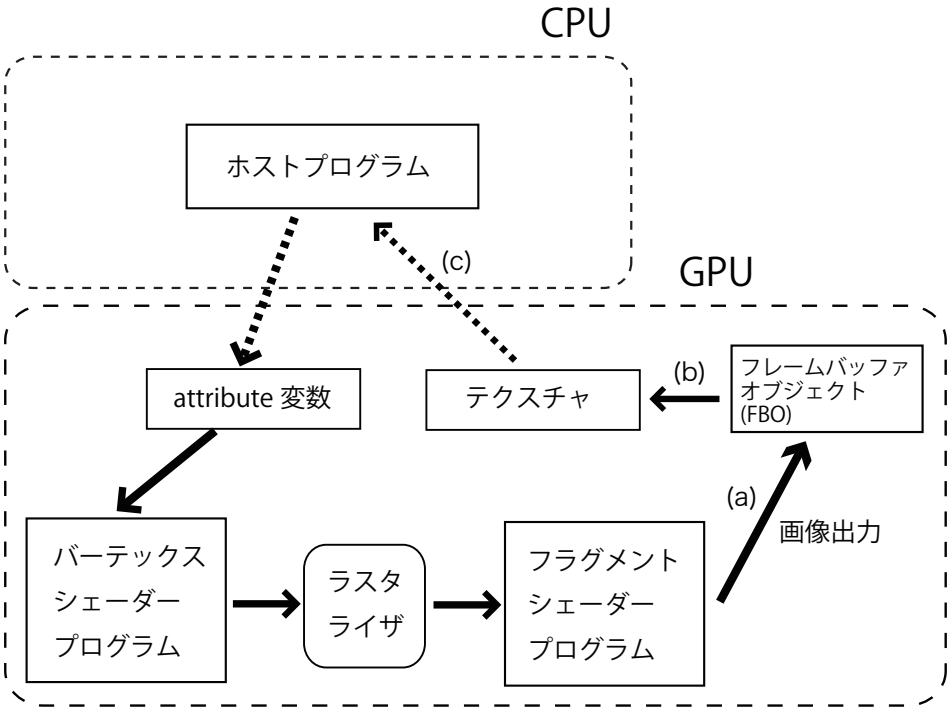


図 12.7: テクスチャへデータを代入する経路

12.3　テクスチャへの代入

12.1 節のプログラムの大きな問題は計算結果を取り出せないことである。結果がホスト側へ取り出せないならば、何の役にも立たない。

実は WebGL には（もちろん本家の OpenGL にも）フラグメントシェーダの計算結果をフレームバッファにではなく、テクスチャに書き込む機能が用意されている。この機能は GPGPU のために導入された訳ではなく、たとえば鏡のある 3D シーンにおいて鏡に写った映像を含めてシーンを正確に描画するために導入されたものである[1]。また、WebGL（と OpenGL）にはテクスチャデータをホストへ取り出す機能も用意されている。これらの機能を用い、計算結果を取り出す。

この節ではそれら機能を簡単なサンプルプログラムで確認する。

図 12.7（155 ページ）はテクスチャへデータを書き込む処理の概念図である。WebGL ではフレームバッファを仮想化/抽象化した機能オブジェクトを**フレームバッファオブジェクト**（以下、FBO）と呼ぶ。これを用いてテクスチャをフレームバッファに見せかける。手順は以下の通りである。

1. まず、FBO をひとつ生成する。

[1]鏡のある 3D シーンの描画では、まず鏡に映る画像を求め、その画像を一旦、メモリに格納する。格納された画像は再度、読み込まれ、3D シーンの中に組み込まれる。この「一旦、メモリに格納する」ために、計算結果をフレームバッファではなく、テクスチャに書き込む機能が導入された。

2. フラグメントシェーダの出力を FBO に接続する（図 12.7（155 ページ）の (a)）

3. FBO をテクスチャと接続する（図 12.7 の (b)）

4. 通常の描画処理を行う。これによって、フラグメントシェーダの計算した各画素の RGBA 値は、フレームバッファへは格納されず、上のテクスチャの対応する画素へ格納される。

5. テクスチャデータを CPU のメモリへ読み出し（図 12.7 の (c)）、ホストプログラムでその内容を確認する。

12.3.1　ホストプログラム

図 12.8（157 ページ）、図 12.9（158 ページ） がホストプログラムの主要部分である。

まず、以下で計算のサイズを指定する

```
let width = 8;
let height = 8;
```

この章では動作確認が目的であるから、非常に小さなサイズを指定した。上のサイズはテクスチャのサイズであり、かつウインドウサイズでもある。

関数 `initSystem()` の以下の部分は、フレームバッファオブジェクト（FBO）を作成し、それをフレームバッファ（フラグメントシェーダの出力）に結合する処理である。図 12.7（155 ページ）の (a) に相当する。この部分はこれ以降も変更しないため、`initSystem()` に加えておく。

```
    let frameBuffer = gl.createFramebuffer();
    gl.bindFramebuffer(gl.FRAMEBUFFER, frameBuffer);
```

関数 `initData()` の以下の部分は、書き込み用テクスチャの生成と設定である。

```
    let texture = buildTexture(null, width, height);
    bindWTexture(texture);
```

関数呼び出し `buildTexture(null, width, height)` の第 1 引数が `null` の場合、CPU から GPU へデータ転送を行わない。関数呼び出し `bindWTexture(texture)` （`bind` と `Texture` の間の `W` に注意）は引数のテクスチャを書き込み用に設定する関数である。これは図 12.7（155 ページ）の (b) に相当する。`W` は write に意味するものとして命名した。関数の中身は後に述べる。

```
// main.js
let gl;
let program;

let width = 8;
let height = 8;

function initSystem() {
    let c = document.getElementById('canvas');
    c.width = width; c.height = height;

    gl = c.getContext('webgl');

    gl.clearColor(0.0, 0.0, 0.0, 1.0);

    let flg = (gl.getExtension('OES_texture_float') != null) &&
              (gl.getExtension("OES_texture_float_linear") != null);
    if (!flg) {
        alert('float texture not supported');
        return;
    }

    let frameBuffer = gl.createFramebuffer();
    gl.bindFramebuffer(gl.FRAMEBUFFER, frameBuffer);

    buildProgram('vs', 'fs');
    gl.useProgram(program);
}

let NUM_POINTS = 3;

function initData() {
    let position = [
                    -0.5, -0.5,
                    +0.5, +0.5,
                    +0.5, -0.5
                ];
    let buffer1 = buildArrayBuffer(position);
    bindArrayBuffer(buffer1, 'position', 2);

    let texture = buildTexture(null, width, height);
    bindWTexture(texture);
}
```

図 12.8: テクスチャへの代入を行う JavaScript ホストプログラム `main.js`（その 1、その 2 へ続く）

```
function compute() {
    gl.clear(gl.COLOR_BUFFER_BIT);
    gl.drawArrays(gl.TRIANGLES, 0, NUM_POINTS);
    gl.finish();
}

function showResults() {
    let result = new Float32Array(4*width*height);
    readFromTexture(result, width, height);
    let str = "";
    for (let h = height-1; h >= 0; h--) {
        for (let w = 0; w < width; w++) {
            let k = 4*(h*width+w);
            str = str +
                  " " + w + "," + h + ": " +
                  result[k]+" "+
                  result[k+1]+" "+
                  result[k+2]+" "+
                  result[k+3]+"\n";
        }
    }
    alert(str);
}
window.onload = function(){
    initSystem();
    initData();
    compute();
    showResults();
};
```

図 12.9: テクスチャへの代入を行う JavaScript ホストプログラム main.js（その 2）

関数 compute() は前章の display() と同様に実行可能シェーダプログラムを起動する。本体中の関数呼び出し：

```
    gl.finish();
```

は、GPU の実行終了を待つものである。というのも、シェーダプログラムを起動する gl.drawArrays() は実行終了を待たないで呼び出し処理を終えるからである[2]。通常の GPU の処置ではホストはシェーダの実行終了を待たず、その間に別の処理を行った方が効率がよい。しかし、計算結果をチェックするためには、シェーダの実行が完全に終了している必要がある。gl.finish() は実行が終了するまで関数呼び出しから戻ってこない[3]。

[2]実行の終了を待たずに制御が戻る関数呼び出し処理をノンブロッキング型処理と呼ぶ。
[3]これに対して gl.flush() は GPU にシェーダの起動を催促し、直ちに関数呼び出しから戻ってくる関数である。

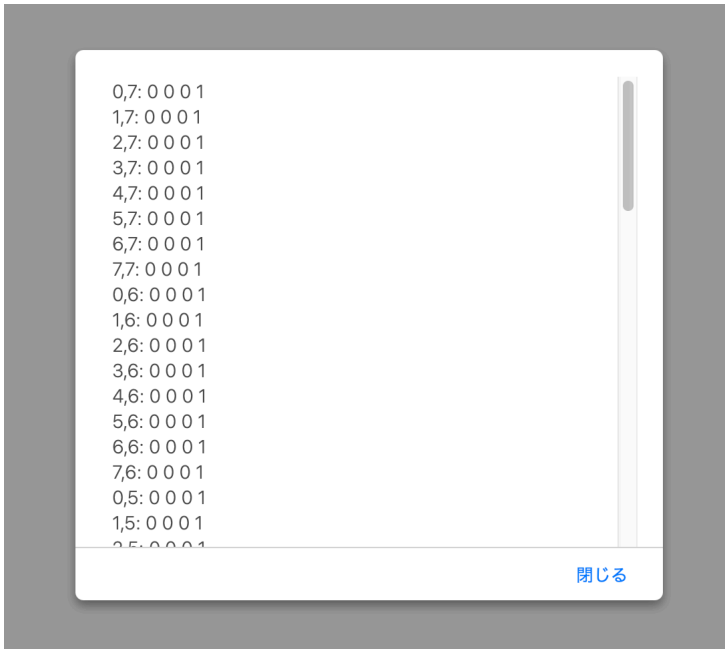


図 12.10: 警告ボックスへの出力例：MacOS、Safari の場合

関数 `showResults()` は GPU のテクスチャに書き込まれた計算結果をホスト側へ転送し、その内容を標準出力へ表示する。以下は、テクスチャデータをホスト側のメモリへ読み込む処理である。

```
let result = new Float32Array(4*width*height);
readFromTexture(result, width, height);
```

1 行目では転送先に必要なメモリをホスト側に用意する。そして 2 行目で、メモリバッファ（配列）`result` に第 2、第 3 引数で与えた量のデータを転送する。これは図 12.7（155 ページ）の (c) に相当する。

その次の 2 重ループでは転送したデータを警告ボックスに表示する。

```
let str = "";
for (let h = height-1; h >= 0; h--) {
    for (let w = 0; w < width; w++) {
        let k = 4*(h*width+w);
        str = str +
              " " + w + "," + h + ": " +
              result[k]+" "+
              result[k+1]+" "+
              result[k+2]+" "+
              result[k+3]+"\n";
    }
}
alert(str);
```

末尾の alert() が、警告ボックスに文字列を表示させる関数である。JavaScript では、値の出力/表示には console.log() を用いることが多いが、その場合、表示内容を確認するために、いちいちコンソールウインドウを開かねばならず、手間が掛かる。そこでここでは警告ボックスを用いることとする。図 12.10（159 ページ）がこの節のプログラムで現れる警告ボックスの例である。出力例の各行は

列番号, 列番号：R の値 G の値 B の値 A の値

を順に出力する。なお出力では、テクスチャデータの値を上から順に表示するため、ループ変数 h は height-1 から 0 まで下に向かって変化させる。実際の出力の検証は後に行う。

上の解説に出てきた関数 bindWTexture()、readFromTexture() の内容は図 12.11（161 ページ） の通りである。なお、図 12.11（161 ページ） には、書き出し用の bindWTexture() の逆の読み込み用の bindRTexture() も載せたが、これは前章までの bindTexture() と全く同じ処理を行う。

bindWTexture() は、メソッド gl.framebufferTexture2D() を呼び出しているに過ぎない。このメソッドはテクスチャをフレームバッに結合するためのメソッドである。WebGL では第 4 引数の texture 以外は実質、図 12.11（161 ページ）に載っている値に固定である[4]。このメソッドを実行することで任意のテクスチャを出力用に設定可能である（図 12.12（162 ページ））。後節では、複数の出力用テクスチャを切り替えながら、より複雑な処理を行うことになる。

readFromTexture() も、実体はメソッド gl.readPixels() である。このメソッドは、出力に割り当てられた 2 次元テクスチャデータ内の任意の位置（第 1、第 2 引数で指定）の任意の大きさ（第 3、第 4 引数で指定）の任意形式（第 5、第 6 引数で指定）の矩形データを第 7 引数の配列へ格

[4] WebGL の最新版である WebGL 2 では引数の自由度が増えているが、それについては省略する。

```
function bindRTexture(texture, name, num) {
    let location = gl.getUniformLocation(program, name);
    gl.uniform1i(location, num);

    gl.activeTexture(gl.TEXTURE0+num);
    gl.bindTexture(gl.TEXTURE_2D, texture);
}

function bindWTexture(texture) {
    gl.framebufferTexture2D(gl.FRAMEBUFFER,
                            gl.COLOR_ATTACHMENT0,
                            gl.TEXTURE_2D,
                            texture,                  // テクスチャを指定
                            0);
}

function readFromTexture(data, w, h) {
    gl.readPixels(0, 0,
                  w, h,
                  gl.RGBA, gl.FLOAT,
                  data);
}
```

図 12.11: テクスチャへの代入を行うための関数群：util.js に追加

納する。ここでは、全データを転送するため、第 1、第 2 引数は左下の位置 $(0, 0)$、第 3、第 4 は全体の大きさ (width, height)、第 5 引数は RGBA 値（= gl.RGBA）、第 6 引数は単精度浮動小数点データ（= gl.FLOAT）を指定する。

12.3.2　バーテックスシェーダプログラム

図 12.13（163 ページ）の前半がここでのバーテックスシェーダプログラムである。入力された 2 次元座標値をそのままラスタライザへ出力する。

12.3.3　フラグメントシェーダプログラム

図 12.13（163 ページ）の後半がフラグメントシェーダプログラムである。

代入文右辺の gl_FragCoord は vec4 型の読み込み専用の GLSL 予約変数であり、当該フラグメントシェーダが担当している画像上の**画素位置がこの変数の x 成分、y 成分（あるいは s 成分、t 成分、または r 成分、g 成分でもよい）に自動的に格納される**。よって、その値の範囲は

$$(x \text{ 成分}, y \text{ 成分}) \in [0, \texttt{width}] \times [0, \texttt{height}]$$

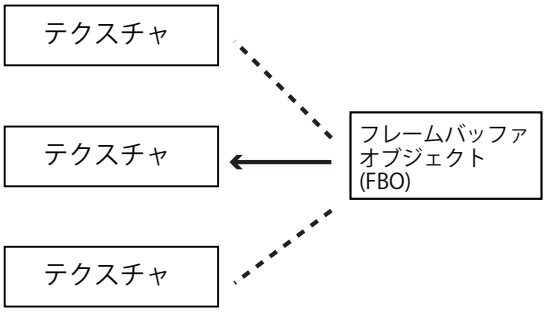


図 12.12: 出力経路の中のひとつを `bindWTexture()` で選択

である。正確に言えば、各画素の中心の位置が格納されるため、それぞれ

$$x \text{ 成分} \quad = \quad 0.5,\ 1.5,\ 2.5,\ \cdots,\ \texttt{width} - 0.5$$

$$y \text{ 成分} \quad = \quad 0.5,\ 1.5,\ 2.5,\ \cdots,\ \texttt{height} - 0.5$$

の離散値が格納される。例題では `width`= 8、`height`= 8 と設定しているから、図 12.14（164 ページ）の通りである。

12.3.4　実行の様子

図 12.15（165 ページ）がこの節のプログラムの動作イメージである。実際の `showResults()` の出力は以下の通りである。

```
 0,7: 0 0 0 1
 1,7: 0 0 0 1
 2,7: 0 0 0 1
 3,7: 0 0 0 1
 4,7: 0 0 0 1
 5,7: 0 0 0 1
 6,7: 0 0 0 1
 7,7: 0 0 0 1
 0,6: 0 0 0 1
 1,6: 0 0 0 1
 2,6: 0 0 0 1
 3,6: 0 0 0 1
 4,6: 0 0 0 1
 5,6: 0 0 0 1
 6,6: 0 0 0 1
 7,6: 0 0 0 1
 0,5: 0 0 0 1
 1,5: 0 0 0 1
 2,5: 0 0 0 1
 3,5: 0 0 0 1
 4,5: 0 0 0 1
 5,5: 5.5 5.5 0.5 1
```

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<script src="./main.js" type="text/javascript"></script>
<script src="./util.js" type="text/javascript"></script>
<script src="./compileLink.js" type="text/javascript"></script>

<script id="vs" type="text/plain">
    attribute vec2 position;

    void main(void)
    {
        gl_Position = vec4(position, 0.0, 1.0);
    }
</script>

<script id="fs" type="text/plain">
    precision highp float;

    void main(void)
    {
        gl_FragColor = gl_FragCoord;
    }
</script>

</head>
<body>
<canvas id="canvas"></canvas>
</body>
</html>
```

図 12.13: テクスチャへの代入を行う HTML/シェーダプログラム

```
6,5: 0 0 0 1
7,5: 0 0 0 1
0,4: 0 0 0 1
1,4: 0 0 0 1
2,4: 0 0 0 1
3,4: 0 0 0 1
4,4: 4.5 4.5 0.5 1
5,4: 5.5 4.5 0.5 1
6,4: 0 0 0 1
7,4: 0 0 0 1
0,3: 0 0 0 1
1,3: 0 0 0 1
2,3: 0 0 0 1
3,3: 3.5 3.5 0.5 1
4,3: 4.5 3.5 0.5 1
5,3: 5.5 3.5 0.5 1
6,3: 0 0 0 1
7,3: 0 0 0 1
0,2: 0 0 0 1
```

(0.5, 7.5)	(1.5, 7.5)	(2.5, 7.5)	…	(7.5, 7.5)
…	…	…	…	…
(0.5, 2.5)	(1.5, 2.5)	(2.5, 2.5)	…	(7.5, 2.5)
(0.5, 1.5)	(1.5, 1.5)	(2.5, 1.5)	…	(7.5, 1.5)
(0.5, 0.5)	(1.5, 0.5)	(2.5, 0.5)	…	(7.5, 0.5)

図 12.14: 画像（出力用テクスチャ）上の gl_FragCoord の x 成分と y 成分：width= 8、height= 8 の場合

```
 1,2: 0 0 0 1
 2,2: 2.5 2.5 0.5 1
 3,2: 3.5 2.5 0.5 1
 4,2: 4.5 2.5 0.5 1
 5,2: 5.5 2.5 0.5 1
 6,2: 0 0 0 1
 7,2: 0 0 0 1
 0,1: 0 0 0 1
 1,1: 0 0 0 1
 2,1: 0 0 0 1
 3,1: 0 0 0 1
 4,1: 0 0 0 1
 5,1: 0 0 0 1
 6,1: 0 0 0 1
 7,1: 0 0 0 1
 0,0: 0 0 0 1
 1,0: 0 0 0 1
 2,0: 0 0 0 1
 3,0: 0 0 0 1
 4,0: 0 0 0 1
 5,0: 0 0 0 1
 6,0: 0 0 0 1
 7,0: 0 0 0 1
```

図 12.15（165 ページ）に、各座標値での x 成分、y 成分を 2 次元上に並べている。ちょうど三角形に対応する部分のみに、画素の座標値が格納されていることが分かる。逆にそれ以外の部分には書き込みは行われていない。

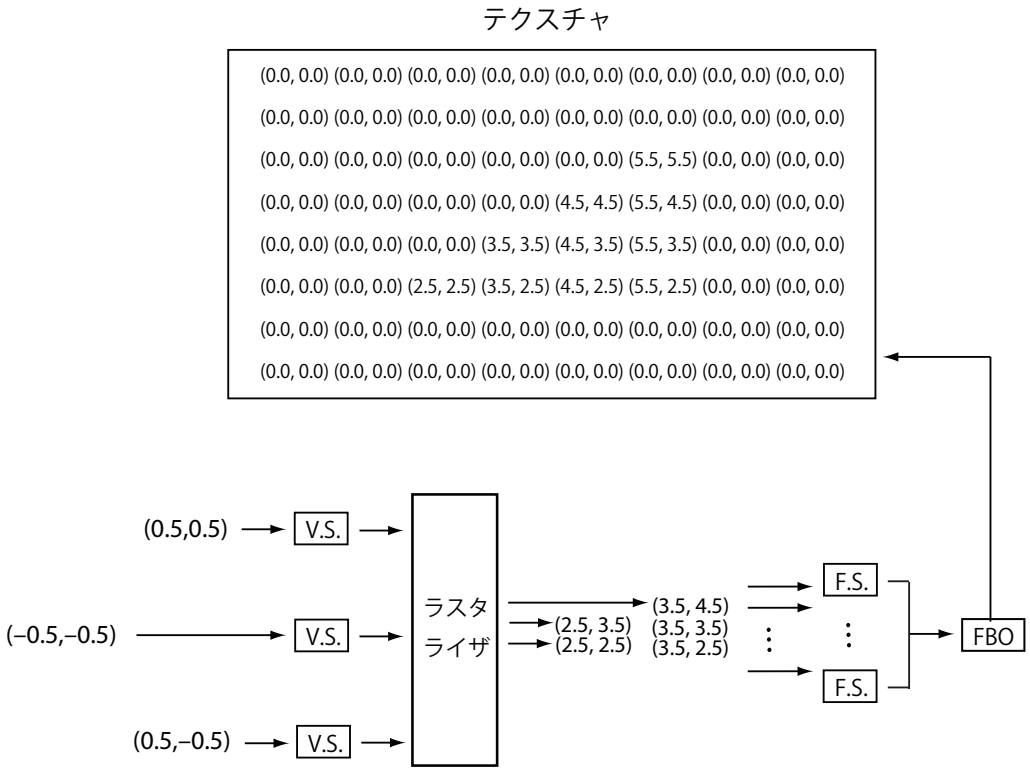


図 12.15: テクスチャへの代入の動作例

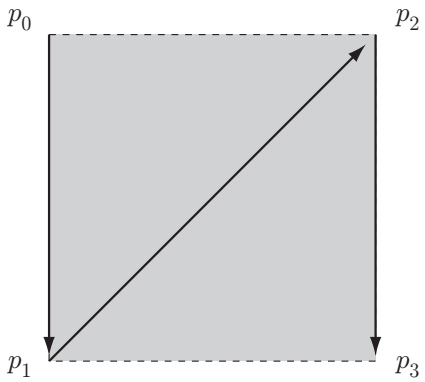


図 12.16: triangle strip による矩形の構成

12.4　テクスチャ全域への代入

前節の例では、三角形領域にのみ代入が行われた。しかし計算目的ではテクスチャの全画素へ代入が行われるべきである。

このためのプログラムの改造は容易である。バーテックスシェーダから出力される座標点群が、画像の全域 $[-1, 1] \times [-1, 1]$ を覆うように設定すればよい。全域を覆うためにここでは 2 枚の三角形を用い、矩形を構成する。以下がそのプログラムである。

12.4.1　ホストプログラム

矩形のテクスチャ全体で計算を行うため、1 枚の三角形を描画するモード（gl.TRIANGLES）ではなく、複数の近接する三角形を描画するモード（gl.TRIANGLE_STRIP）を用いて正方形を描画する。それを図 12.17（167 ページ）のプログラムについて解説していく。

まず、正方形の 4 頂点を用いることを宣言する。

```
let NUM_POINTS = 4;
```

次に、以下のように、画面全体を覆う正方形の 4 頂点の座標値を設定する。

```
    let position = [
        -1.0, +1.0,                                          // p0
        -1.0, -1.0,                                          // p1
        +1.0, +1.0,                                          // p2
        +1.0, -1.0                                           // p3
    ];
```

```
// main.js
let gl;
let program;

let width = 8;
let height = 8;

function initSystem() {
    ** 前節と同じ **
}

let NUM_POINTS = 4;

function initData() {
    let position = [
        -1.0, +1.0,                                                    // p0
        -1.0, -1.0,                                                    // p1
        +1.0, +1.0,                                                    // p2
        +1.0, -1.0                                                     // p3
    ];

    let buffer1 = buildArrayBuffer(position);
    bindArrayBuffer(buffer1, 'position', 2);

    let texture = buildTexture(null, width, height);
    bindWTexture(texture);
}

function compute() {
    gl.drawArrays(gl.TRIANGLE_STRIP, 0, NUM_POINTS); // 描画モードを変更
    gl.finish();
}

function showResult() {
    ** 前節と同じ **
}
window.onload = function(){
    ** 前節と同じ **
};
```

図 12.17: テクスチャ全体を覆う正方形を描画する JavaScript ホストプログラム `main.js`

ここに頂点 p_0、...、p_3 は図 12.16（166 ページ）の配置である。

さて、関数 compute() では、gl.TRIANGLE_STRIP を指定してシェーダプログラムを起動する。

```
gl.drawArrays(gl.TRIANGLE_STRIP, 0, NUM_POINTS); // 描画モードを変更
```

gl.TRIANGLE_STRIP では、頂点配列中の連続する 3 頂点から順次、三角形を構成し、描画していく。この節のプログラムで言えば、まず 3 頂点 p_0、p_1、p_2 で 1 枚の三角形を構成する。次に、要素を一つずらし、3 頂点 p_1、p_2、p_3 で次の三角形を作る。このようにして、4 頂点について 2 枚の三角形を構成できる。その 2 枚の三角形は図 12.16（166 ページ）通りに正方形を覆うこととなる。

ところで、画像の全域（全ての画素）に対して画素値の書き込みを行うから、ここまで必ず行なってきたシェーダプログラム起動直前の画像のクリア：

```
gl.clear(gl.COLOR_BUFFER_BIT);
```

は、もはや不要である。図 12.17（167 ページ）からはこれを削除し、これ以降の全てのプログラムにおいても削除する。

この節のプログラムの変更は以上である。

12.4.2　実行の様子

図 12.18（169 ページ）は実行のイメージである。

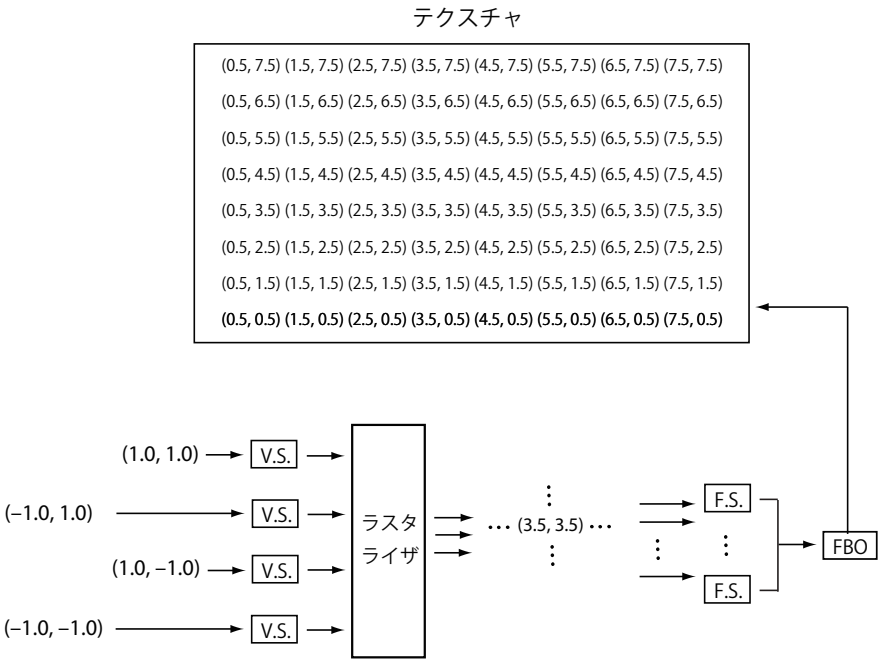


図 12.18: テクスチャ全域への代入の動作例

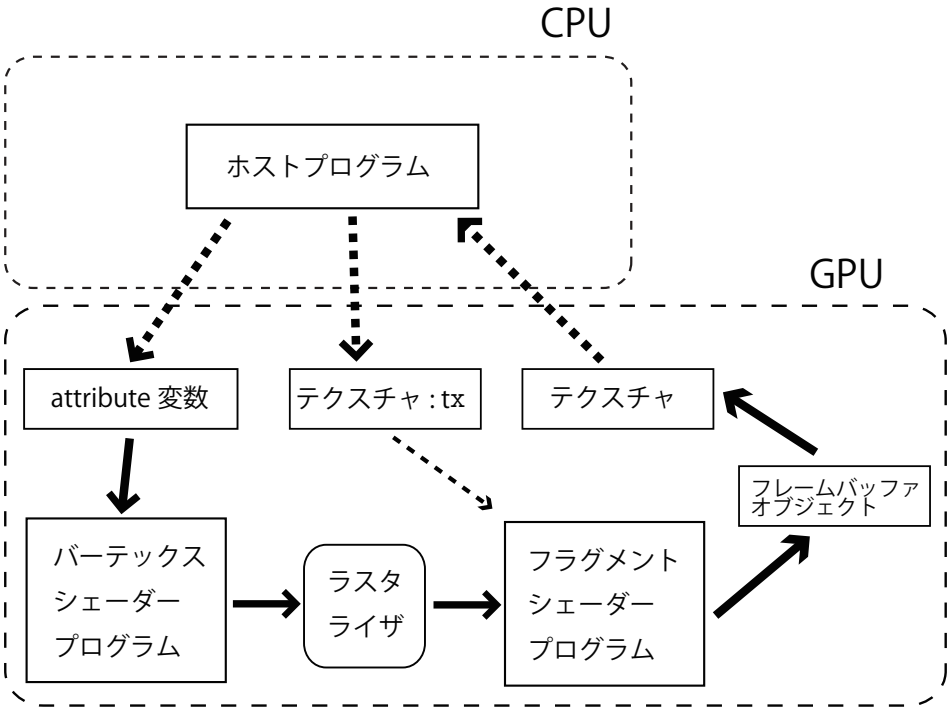


図 12.19: コピーの経路

12.5　テクスチャ間のコピー

　前節まででテクスチャに値を代入できることを確認した。

　そこで次にテクスチャ間でデータの単純なコピーを確認する。C のプログラムに表すならば、2 次元配列 x、z について

```
z[i][j] = x[i][j];
```

と相当する実行である。

　GPU 内のデータの流れを図 12.19（170 ページ）のように設定する。まず、ホストプログラムによってフラグメントシェーダプログラム内のテクスチャ x を適当な初期値でセットアップする。この方法は前章の通りである。次に、そのテクスチャの内容を出力用テクスチャへ画素毎に格納すればよい。具体的には以下の通りである。

12.5.1　ホストプログラム

まず、以下の大域変数を追加宣言しておく。これは initData() と showResults() から使用される。

```
let xData;                                                    // 入力用テクスチャ
```

さて、理由は後に述べるが、テクスチャをシェーダで読み込むプログラムでは画面の幅と高さをシェーダの uniform 変数に受け渡す必要がある。そこで図 12.20（172 ページ）の initData() の中で以下の処理を行う。

```
    setUniformFloat('width', width);
    setUniformFloat('height', height);
```

次に、配列 xData に適当な初期値を設定する。ここでは配列要素の値が適当に散らばる以下の設定を用いる。

```
            xData[k+0] = w+h+0.1;
            xData[k+1] = w+h+0.2;
            xData[k+2] = w+h+0.3;
            xData[k+3] = w+h+0.4;
```

次に、以下で入力用テクスチャオブジェクト textureX を xData を用いて生成し、このオブジェクトをテクスチャunifrom 変数 tx に結びつける。

```
    let textureX = buildTexture(xData, width, height);
    ...
    bindRTexture(textureX, 'tx', 0);
```

なお、この辺り：

```
    let textureX = buildTexture(xData, width, height);
    let textureZ = buildTexture(null, width, height);

    bindRTexture(textureX, 'tx', 0);
    bindWTexture(textureZ);
```

```
let gl;
let program;

let width = 8;
let height = 8;

function initSystem() {
    ** 前節と同じ **
}

let NUM_POINTS = 4;

let xData;                                                  // 入力用テクスチャ

function initData() {
    let position = [
        -1.0, +1.0,
        -1.0, -1.0,
        +1.0, +1.0,
        +1.0, -1.0
    ];
    let buffer1 = buildArrayBuffer(position);
    bindArrayBuffer(buffer1, 'position', 2);

    setUniformFloat('width', width);
    setUniformFloat('height', height);

    xData = new Float32Array(4*width*height);               // 入力データの準備
    for (let w = 0; w < width; w++) {
        for (let h = 0; h < height; h++) {
            let k = 4*(h*width+w);
            xData[k+0] = w+h+0.1;
            xData[k+1] = w+h+0.2;
            xData[k+2] = w+h+0.3;
            xData[k+3] = w+h+0.4;
        }
    }

    let textureX = buildTexture(xData, width, height);
    let textureZ = buildTexture(null, width, height);

    bindRTexture(textureX, 'tx', 0);
    bindWTexture(textureZ);
}
```

図 12.20: テクスチャ間でデータをコピーする JavaScript ホストプログラム `main.js`（その 1、その 2 へ続く）

```
function compute() {
    ** 前節と同じ **
}

function showResults() {
    let result = new Float32Array(4*width*height);
    readFromTexture(result, width, height);
    let error = 0;
    for (let h = 0; h < height; h++) {
        for (let w = 0; w < width; w++) {
            let k = 4*(h*width+w);
            for (let i = 0; i < 4; i++) {
                error += Math.abs(result[k+i]-xData[k+i])/Math.abs(xData[k+i]);
            }
        }
    }
    alert(error/4/width/height);
}

window.onload = function(){
    ** 前節と同じ **
};
```

図 12.21: テクスチャ間でデータをコピーする JavaScript ホストプログラム main.js（その 2）

の実行順を

```
    let textureX = buildTexture(xData, width, height);
    bindRTexture(textureX, 'tx', 0);

    let textureZ = buildTexture(null, width, height);
    bindWTexture(textureZ);
```

などと変更した場合、プログラムは正常実行されないことを注意する。この辺りの設定の順序は非常にセンシティブなところがある。原則として以下のルールで設定を行う。

1. テクスチャオブジェクトの生成を先にまとめて行い、その後にまとめてシェーダと結合する。
2. テクスチャの装置番号は 0 から順に使う。

図 12.21（173 ページ）の関数 showResults() では、以下の式：

```
                error += Math.abs(result[k+i]-xData[k+i])/Math.abs(xData[k+i]);
```

で、シェーダプログラムの出力値と元の入力値の相対誤差を合算していく。そして、合算終了後に alert(error/4/width/height) で相対誤差の平均値を出力する。その値が 0 でなければコピーは正常に行われていないこととなる。

12.5.2　バーテックスシェーダプログラム

図 12.22（175 ページ）の前半がここでのバーテックスシェーダプログラムである。前節から変更はない。

12.5.3　フラグメントシェーダ

フラグメントシェーダプログラムは図 12.22（175 ページ）の後半部である。

ここでテクスチャ tx にアクセスするときの座標値の計算式が以下であることに注目したい。

```
vec2 texCoord = vec2(gl_FragCoord.x/width,
                     gl_FragCoord.y/height);
```

何故ならば、gl_FragCoord.xy の値の範囲は

$$[0, \mathtt{width}] \times [0, \mathtt{height}]$$

であり、テクスチャ座標の範囲は

$$[0, 1] \times [0, 1]$$

である。これを整合させるのが上の計算式である。余計な手間であるが、WebGL は（と言うよりも本家の OpenGL は）元々 GPGPU を前提とした仕様となっていないため、一手間が要る。

次に代入文：

```
gl_FragColor = texture2D(tx,texCoord);
```

では 4 個の float 型のデータ（RGBA 輝度値の 4 要素）が同時にコピーされることに注意する。ひとつの実行文で複数のデータを同時処理することを **SIMD**（Single Instruction Multiple Data）演算と呼ぶ。GPU による計算は、代入操作のみならず、四則演算も、SIMD 演算を含む。

12.5.4　実行の様子

誤差の平均値が 0 であることを確認できる。

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<script src="./main.js" type="text/javascript"></script>
<script src="./util.js" type="text/javascript"></script>
<script src="./compileLink.js" type="text/javascript"></script>

<script id="vs" type="text/plain">
    attribute vec2 position;

    void main(void)
    {
        gl_Position = vec4(position, 0.0, 1.0);
    }
</script>

<script id="fs" type="text/plain">
    precision highp float;

    uniform sampler2D tx;

    uniform float width;
    uniform float height;

    void main(void) {
        vec2 texCoord = vec2(gl_FragCoord.x/width,
                             gl_FragCoord.y/height);

        gl_FragColor = texture2D(tx,texCoord);
    }
</script>

</head>
<body>
<canvas id="canvas"></canvas>
</body>
</html>
```

図 12.22: テクスチャ間でデータをコピーする HTML/シェーダプログラム

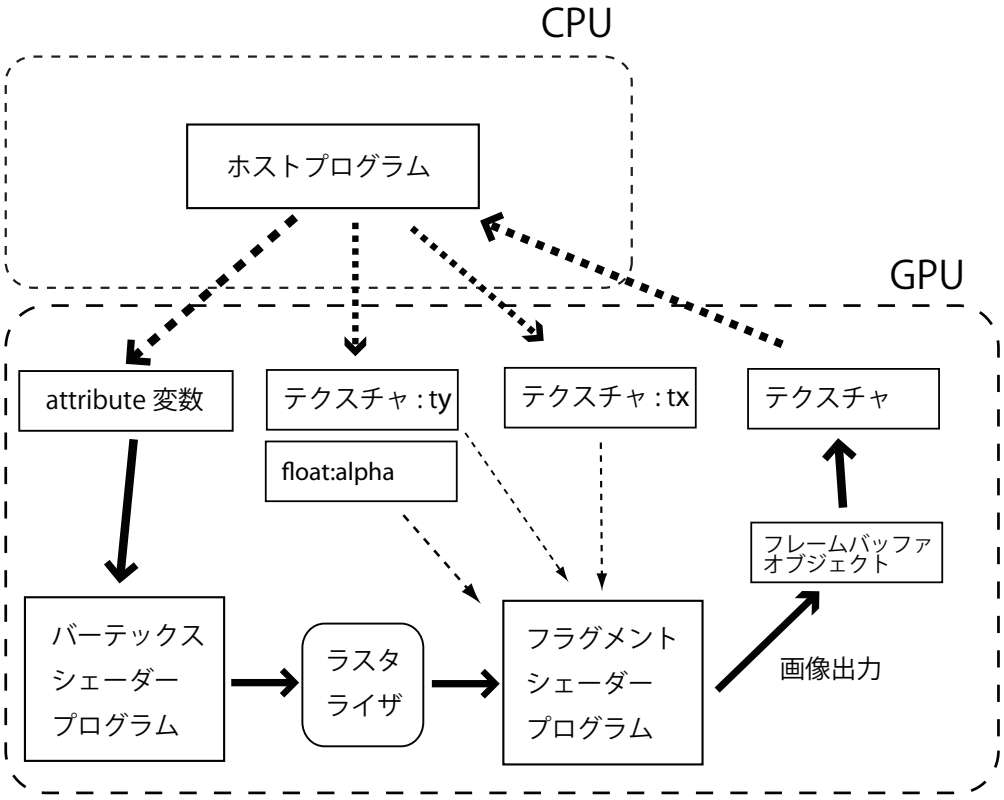


図 12.23: saxpy 計算の経路

12.6　テクスチャを用いた簡単な並列計算

テクスチャによる計算の準備はできた。ここでは簡単な計算を行ってみる。x、y、z を適当な 2 次元配列、alpha を適当な定数とするとき、

```
z[i][j] = alpha*x[i][j]+y[i][j];
```

の形式の積和計算を行う。この計算を double 型（double precision）で行うとき、この式をしばしば daxpy と呼ぶ。これは double、alpha、x、plus、yに由来する。同じく float 型（single precision）で計算するときには saxpy = single、alpha、x、plus、y と呼ぶ。GPU を用いる計算では一般に saxpy を実行できる[5]。

GPU を用いた計算経路のイメージは図 12.23（176 ページ）の通りである。三つのテクスチャとひとつの定数を uniform 変数として実装する。

```
let gl;
let program;

let width = 8;
let height = 8;
let alpha = 0.9;

function initSystem() {
    ** 前節と同じ **
}

let NUM_POINTS = 4;

let xData;
let yData;

function initData() {
    let position = [
        -1.0, +1.0,
        -1.0, -1.0,
        +1.0, +1.0,
        +1.0, -1.0
    ];
    let buffer1 = buildArrayBuffer(position);
    bindArrayBuffer(buffer1, 'position', 2);

    setUniformFloat('width', width);
    setUniformFloat('height', height);
    setUniformFloat('alpha', alpha);

    xData = new Float32Array(4*width*height);
    yData = new Float32Array(4*width*height);

    for (let w = 0; w < width; w++) {
        for (let h = 0; h < height; h++) {
            let k = 4*(h*width+w);
            xData[k+0] = w+0.1;
            xData[k+1] = w+0.2;
            xData[k+2] = w+0.3;
            xData[k+3] = w+0.4;

            yData[k+0] = h+0.1;
            yData[k+1] = h+0.2;
            yData[k+2] = h+0.3;
            yData[k+3] = h+0.4;
        }
    }
```

図 12.24: saxpy の計算を行う JavaScript ホストプログラム `main.js`（その 1、その 2 へ続く）

```
    let textureX = buildTexture(xData, width, height);
    let textureY = buildTexture(yData, width, height);
    let textureZ = buildTexture(null, width, height);

    bindRTexture(textureX, 'tx', 0);
    bindRTexture(textureY, 'ty', 1);
    bindWTexture(textureZ);
}

function compute() {
    ** 前節と同じ **
}

function showResults() {
    let result = new Float32Array(4*width*height);
    readFromTexture(result, width, height);
    let error = 0;
    for (let h = 0; h < height; h++) {
        for (let w = 0; w < width; w++) {
            let k = 4*(h*width+w);
            for (let i = 0; i < 4; i++) {
                let val =alpha*xData[k+i]+yData[k+i];
                error += Math.abs(result[k+i]-val)/Math.abs(val);
            }
        }
    }
    alert(error/4/width/height);
}

window.onload = function(){
    ** 前節と同じ **
};
```

図 12.25: saxpy の計算を行う JavaScript ホストプログラム `main.js`（その 2）

12.6.1　ホストプログラム

図 12.24（177 ページ）、図 12.25（178 ページ）がホストプログラムである。

まず、プログラムの構成上、複数の関数から参照する以下の大域変数を宣言しておく。

```
let alpha = 0.9;
```

```
let xData;
let yData;
```

[5]ハイエンドの GPU には `double` 型演算器を搭載するものもあるが、普及型 GPU は `float` 型演算器しか搭載していない。一般に CG で `double` 計算は不要だからである。

関数 initData() には、alpha の値を uniform 変数としてシェーダに受け渡す部分を追加する。

```
    setUniformFloat('alpha', alpha);
```

次に、配列 xData、yData に適当な値を初期設定する。

```
            xData[k+0] = w+0.1;
            xData[k+1] = w+0.2;
            xData[k+2] = w+0.3;
            xData[k+3] = w+0.4;

            yData[k+0] = h+0.1;
            yData[k+1] = h+0.2;
            yData[k+2] = h+0.3;
            yData[k+3] = h+0.4;
```

そしてテクスチャの生成とシェーダとの結合を行う。

```
    let textureX = buildTexture(xData, width, height);
    let textureY = buildTexture(yData, width, height);
    let textureZ = buildTexture(null, width, height);

    bindRTexture(textureX, 'tx', 0);
    bindRTexture(textureY, 'ty', 1);
    bindWTexture(textureZ);
```

関数 showResults() では、以下のように、GPU での saxpy の計算値と CPU での saxpy の計算値の差の絶対値の合計を計算する。

```
            error += Math.abs(result[k+0]-(alpha*xData[k+0]+yData[k+0]))+
```

上のプログラム片の result[k+0] が GPU での計算値、alpha*xData[k+0]+yData[k+0] が CPU での計算値である。

12.6.2　バーテックスシェーダプログラム

図 12.26（180 ページ）の前半がここでのバーテックスシェーダプログラムである。前節から変更はない。

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<script src="./main.js" type="text/javascript"></script>
<script src="./util.js" type="text/javascript"></script>
<script src="./compileLink.js" type="text/javascript"></script>

<script id="vs" type="text/plain">
    attribute vec2 position;

    void main(void)
    {
        gl_Position = vec4(position, 0.0, 1.0);
    }
</script>

<script id="fs" type="text/plain">
    precision highp float;

    uniform sampler2D tx;
    uniform sampler2D ty;

    uniform float alpha;
    uniform float width;
    uniform float height;

    void main(void)
    {
        vec2 texCoord = vec2(gl_FragCoord.x/width,
                             gl_FragCoord.y/height);

        vec4 x = texture2D(tx,texCoord);
        vec4 y = texture2D(ty,texCoord);

        gl_FragColor = alpha*x+y;
    }
</script>

</head>
<body>
<canvas id="canvas"></canvas>
</body>
</html>
```

図 12.26: saxpy の計算を行う HTML/シェーダプログラム

12.6.3　フラグメントシェーダプログラム

図 12.26（180 ページ）は saxpy 計算を行うフラグメントシェーダプログラムである。

まず、uniform 変数 `ty`、`alpha` が追加されている。

`gl_FragColor` への代入文：

```
gl_FragColor = alpha*x+y;
```

では、実際には以下の SIMD 計算

```
gl_FragColor.r = alpha*x.r + y.r;
gl_FragColor.g = alpha*x.g + y.g;
gl_FragColor.b = alpha*x.b + y.b;
gl_FragColor.a = alpha*x.a + y.a;
```

が行われていることは前節に述べた通りである。

GPU の計算では、ひとつのフラグメントシェーダでは SIMD 計算が行われる。そして、多数のフラグメントシェーダの並列実行 – これはしばしば **SPMD**（Single Program Multiple Data）演算と呼ばれる – がその上にある。つまり、GPU での計算は、SIMD と SPMD の 2 段構えになっている。

12.6.4　実行の様子

前節同様に、`showResults()` では GPU と CPU の計算値の平均を計算し、警告ボックに表示する。その結果は、講義担当者の PC では

$$3.0903946849573404e-8$$

であった。よって GPU の計算値と CPU の計算値が完全に一致する訳ではない。これはよく知られた事実である[6]。ここで `float` 型（単精度浮動小数点数型）の有効桁数は 2 進数 24 桁（10 進数で約 7 桁）であることを考え合わせれば、上の誤差は仮数部の最下位 bit 周辺の僅かな誤差であると理解できる。

結論として、GPU と CPU の計算結果はほとんど一致していると言っていいだろう。

[6]以前の GPU は IEEE 準拠でなかったため、GPU の計算値と CPU の計算値に差が生じることが一般的であった。最近では GPU の浮動小数点数演算器も IEEE 準拠になっており、計算結果は同じになる。しかし実際には積和演算の処理方法の違いでわずかに差が生じる場合があることが知られている。ただし、本講義の誤差がそれに該当するか否かは不明である。

12.7　補足：この章のプログラムのリスト

プログラム読解の便利のために、以下にこの章の最後で解説したプログラムのリストを載せる。

```
<!-- HTML -->
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<script src="./main.js" type="text/javascript"></script>
<script src="./util.js" type="text/javascript"></script>
<script src="./compileLink.js" type="text/javascript"></script>

<script id="vs" type="text/plain">
    attribute vec2 position;

    void main(void)
    {
        gl_Position = vec4(position, 0.0, 1.0);
    }
</script>

<script id="fs" type="text/plain">
    precision highp float;

    uniform sampler2D tx;
    uniform sampler2D ty;

    uniform float alpha;
    uniform float width;
    uniform float height;

    void main(void)
    {
        vec2 texCoord = vec2(gl_FragCoord.x/width,
                             gl_FragCoord.y/height);

        vec4 x = texture2D(tx,texCoord);
        vec4 y = texture2D(ty,texCoord);

        gl_FragColor = alpha*x+y;
    }
</script>

</head>
<body>
<canvas id="canvas"></canvas>
</body>
</html>
```

```
// main.js
let gl;
let program;

let width = 8;
let height = 8;
let alpha = 0.9;

function initSystem() {
    let c = document.getElementById('canvas');
    c.width = width; c.height = height;

    gl = c.getContext('webgl');

    let flg = (gl.getExtension('OES_texture_float') != null) &&
              (gl.getExtension("OES_texture_float_linear") != null);
    if (!flg) {
        alert('float texture not supported');
        return;
    }

    let frameBuffer = gl.createFramebuffer();
    gl.bindFramebuffer(gl.FRAMEBUFFER, frameBuffer);

    buildProgram('vs', 'fs');
    gl.useProgram(program);
}

let NUM_POINTS = 4;

let xData;
let yData;

function initData() {
    let position = [
        -1.0, +1.0,
        -1.0, -1.0,
        +1.0, +1.0,
        +1.0, -1.0
    ];

    let buffer1 = buildArrayBuffer(position);
    bindArrayBuffer(buffer1, 'position', 2);

    setUniformFloat('width', width);
    setUniformFloat('height', height);
    setUniformFloat('alpha', alpha);

    xData = new Float32Array(4*width*height);
    yData = new Float32Array(4*width*height);

    for (let w = 0; w < width; w++) {
        for (let h = 0; h < height; h++) {
            let k = 4*(h*width+w);
            xData[k+0] = w+0.1;
            xData[k+1] = w+0.2;
            xData[k+2] = w+0.3;
            xData[k+3] = w+0.4;
```

```
1
2               yData[k+0] = h+0.1;
3               yData[k+1] = h+0.2;
4               yData[k+2] = h+0.3;
5               yData[k+3] = h+0.4;
6           }
7       }
8
9       let textureX = buildTexture(xData, width, height);
10      let textureY = buildTexture(yData, width, height);
11      let textureZ = buildTexture(null, width, height);
12
13      bindRTexture(textureX, 'tx', 0);
14      bindRTexture(textureY, 'ty', 1);
15      bindWTexture(textureZ);
16  }
17
18  function compute() {
19      gl.drawArrays(gl.TRIANGLE_STRIP, 0, NUM_POINTS);
20      gl.finish();
21  }
22
23  function showResults() {
24      let result = new Float32Array(4*width*height);
25      readFromTexture(result, width, height);
26      let error = 0;
27      for (let h = 0; h < height; h++) {
28          for (let w = 0; w < width; w++) {
29              let k = 4*(h*width+w);
30              for (let i = 0; i < 4; i++) {
31                  let val = alpha*xData[k+i]+yData[k+i];
32                  error += Math.abs(result[k+i]-val)/Math.abs(val);
33              }
34          }
35      }
36      alert(error/4/width/height);
37  }
38
39  window.onload = function(){
40      initSystem();
41      initData();
42      compute();
43      showResults();
44  };
```

```
// util.js
// util.js
function buildArrayBuffer(data) {          // 頂点バッファオブジェクトの作成
    let arrayBuffer = gl.createBuffer();           // 空のオブジェクトの作成

    gl.bindBuffer(gl.ARRAY_BUFFER, arrayBuffer);      // オブジェクトを束縛

    gl.bufferData(gl.ARRAY_BUFFER,                // 配列 data の内容を GPU へ
                  new Float32Array(data),         // コピーし、頂点バッファ
                  gl.STATIC_DRAW);                // オブジェクトを作成

    gl.bindBuffer(gl.ARRAY_BUFFER, null);                    //  束縛を外す

    return arrayBuffer;                           // 作成したオブジェクトを戻す
}

function bindArrayBuffer(arrayBuffer, name, s) {
    gl.bindBuffer(gl.ARRAY_BUFFER, arrayBuffer);      // オブジェクトを束縛

    let location = gl.getAttribLocation(program, name);
                                    // バーテックスシェーダ内の name と同じ
                                    // 名前の attribute 変数の GPU 内の位置を取得

    gl.vertexAttribPointer(location, s, gl.FLOAT, false, 0, 0);
                                       // location の頂点バッファの属性を設定

    gl.enableVertexAttribArray(location);
                                    // location の頂点バッファを利用可能にする
}


function setUniformFloat(name, value) {
    let location = gl.getUniformLocation(program, name);
    gl.uniform1f(location, value);
}


function buildTexture(data, w, h) {
    let texture = gl.createTexture();

    gl.bindTexture(gl.TEXTURE_2D, texture); // テクスチャをバインドする

    // テクスチャへイメージを適用
    gl.texImage2D(gl.TEXTURE_2D,
                  0,
                  gl.RGBA,
                  w,
                  h,
                  0,
                  gl.RGBA,
                  gl.FLOAT,
                  data);

    gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER, gl.NEAREST);
    gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.NEAREST);

    // テクスチャのバインドを無効化
    gl.bindTexture(gl.TEXTURE_2D, null);
```

```
1
2       return texture;
3   }
4
5   function bindRTexture(texture, name, num) {
6       let location = gl.getUniformLocation(program, name);
7       gl.uniform1i(location, num);
8       gl.activeTexture(gl.TEXTURE0+num);
9       gl.bindTexture(gl.TEXTURE_2D, texture);
10  }
11
12  function bindWTexture(texture) {
13      gl.framebufferTexture2D(gl.FRAMEBUFFER, gl.COLOR_ATTACHMENT0, gl.TEXTURE_2D, texture, 0);
14  }
15
16  function readFromTexture(data, w, h) {
17      gl.readPixels(0, 0, w, h, gl.RGBA, gl.FLOAT, data);
18  }
```

```
// compileLink.js

// html ファイル中の id 名が vsid、fsid のリソースを文字列として読み込んで
// シェーダ実行可能コードをビルド
function buildProgram(vsid, fsid) {
    let vs = compileProgram(gl.VERTEX_SHADER, vsid);
                        // バーテックスシェーダソースプログラムのコンパイル
    let fs = compileProgram(gl.FRAGMENT_SHADER, fsid);
                        // フラグメントシェーダソースプログラムのコンパイル

    program = gl.createProgram();               // 空の実行可能コードを作成

    gl.attachShader(program, vs);
                         // バーテックスシェーダのオブジェクトコードを接続
    gl.attachShader(program, fs);
                         // フラグメントシェーダのオブジェクトコードを接続

    gl.linkProgram(program);                                    // リンク

    if (!gl.getProgramParameter(program, gl.LINK_STATUS)){ // エラーチェック
        alert(gl.getProgramInfoLog(program));           // エラー内容の表示
    }
}

// シェーダのタイプ（バーテックス/フラグメント）とリソース id から
// オブジェクトコードを戻す
function compileProgram(type, id){
    let source = document.getElementById(id);
                           // HTML ファイル中から id に相当するリソースを取得
    if (!source) { return; }                // 取得できなければ null を返す

    let shader = gl.createShader(type);         // 空のオブジェクトコードを作成

    gl.shaderSource(shader, source.text);          // ソースプログラムの文字
                                                   // 列を shader に接続

    gl.compileShader(shader);                                   // コンパイル

    if (gl.getShaderParameter(shader, gl.COMPILE_STATUS)){ // エラーチェック
        return shader;        // エラーが無いならばオブジェクトコードを戻す
    } else {
        alert(gl.getShaderInfoLog(shader));             // エラー内容の表示
    }
}
```

第13章　テクスチャを用いた大規模計算

前章最後のプログラムでは、画素当たり 1 回の積和演算（saxpy 計算）を行った。しかし、その程度の計算量では GPU による計算のメリットはなく、CPU で計算する方が高速である。そこでこの章ではもっと大量の計算処理に挑戦し、実際に実行速度を測定する。

13.1　テクスチャを用いたピンポン計算

ここで考えるのは、全ての i、j についての、以下のような saxpy 計算の繰り返しである。

```
for(int k = 0; k < L; k++){
    x[i][j] = alpha*x[i][j]+y[i][j];
}
```

上の代入文の問題は左辺と右辺に同じ配列要素 x[i][j] を用いている点である。しかし、前章で述べたように、テクスチャによる計算ではひとつのテクスチャを読み込みと書き込み（参照と代入）の両方に同時に使うことはできない。そこで、この計算を WebGL（OpenGL）で行うためには代替策としてプログラムを以下のように変更する。簡単のため、L は偶数と仮定する。

```
for(int k = 0; k < L/2; k++){
    z[i][j] = alpha*x[i][j]+y[i][j];
    x[i][j] = alpha*z[i][j]+y[i][j];
}
```

ループ本体の 1 行目 z[i][j] = alpha*x[i][j]+y[i][j]; は前章の saxpy 計算に他ならない。2 行目も x と z を置き換えれば前章と同じである。

この二つの代入文に関する GPU 内の計算経路はそれぞれ図 13.1（189 ページ）、図 13.2（189 ページ）である。図の textureX、textureY、textureZ はホストプログラムから見た各テクスチャオブジェクトである。z[i][j] = alpha*x[i][j]+y[i][j] の計算では、それぞれがシェーダの中で tx、tty、tz としてアクセスされる。次の x[i][j] = alpha*z[i][j]+y[i][j] の計算では textureX と textureZ はシェーダ内では tz、tx として入れ替えてアクセスされる。図の二つの計算経路を交互にセットアップしながら GPU の実行を繰り返していく。これを**ピンポン計算**（ping pong computation）とも呼ぶ。読み書きを切り替えながら実行するイメージである。

プログラムは以下の通りである。

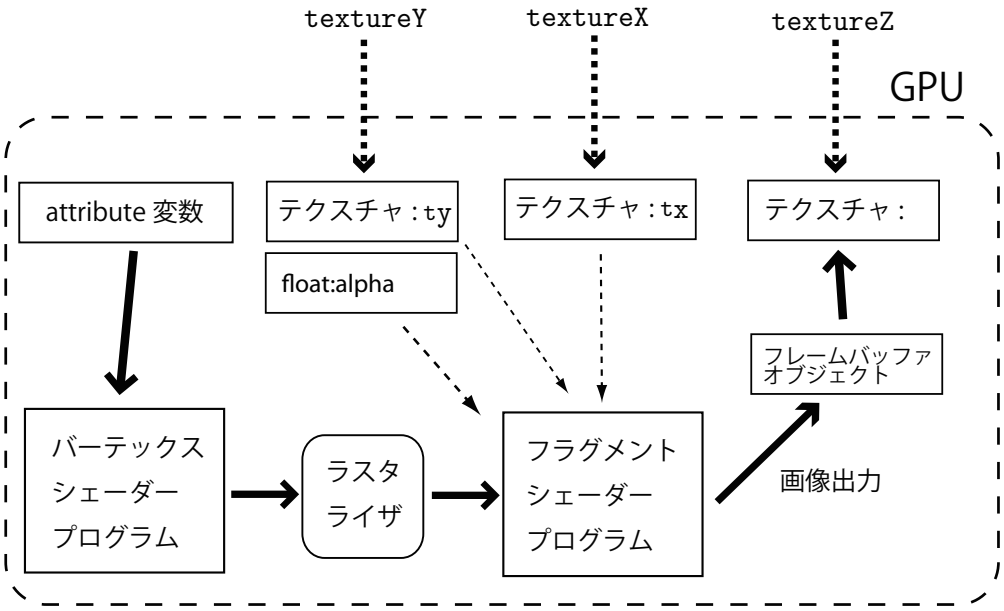


図 13.1: z[i][j] = alpha*x[i][j]+y[i][j] の計算経路

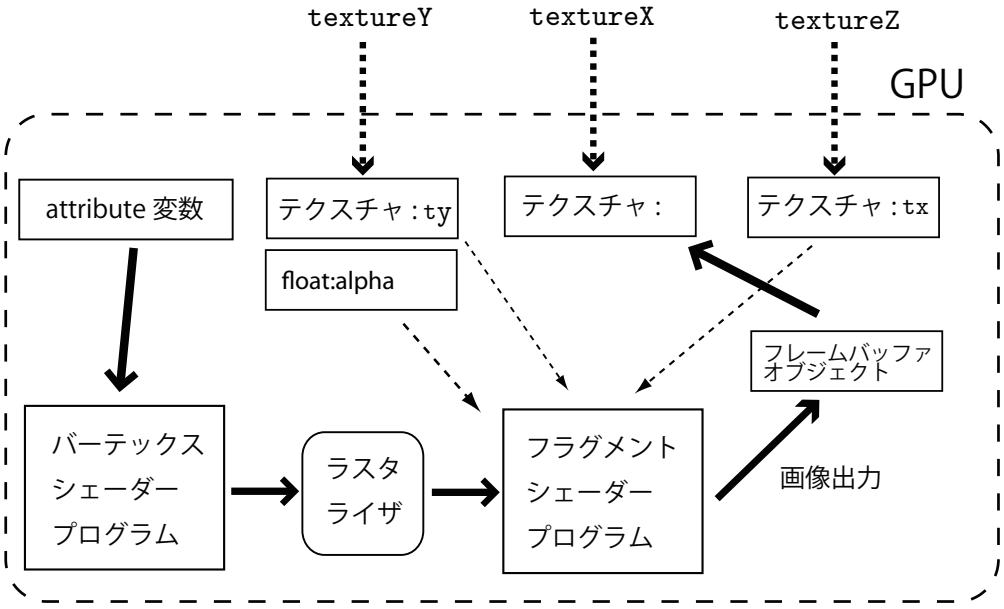


図 13.2: x[i][j] = alpha*z[i][j]+y[i][j] の計算経路

13.1.1　ホストプログラム

図 13.3（191 ページ）、図 13.4（192 ページ）、図 13.5（193 ページ）がホストプログラムである。

まず、計算のサイズを大きくした。

```
let width = 256;
let height = 256;
```

次に、ここで使用する 3 枚のテクスチャは関数 initData()、compute() で参照されるため、以下のように大域宣言しておく。

```
let textureX;
let textureY;
let textureZ;
```

関数 initData() の処理は 12.6 節（176 ページ）とほとんど同じである。唯一異なるのは、テクスチャをシェーダプログラムと結合する bindRTexture()、bindWTexture() を initData() から削除し、compute() へ移動した点である。

その compute() では、シェーダプログラムの実行を繰り返す。まず、図 13.1（189 ページ）に対応する以下を実行する。

```
        bindRTexture(textureX, 'tx', 0);
        bindRTexture(textureY, 'ty', 1);
        bindWTexture(textureZ);

        gl.drawArrays(gl.TRIANGLE_STRIP, 0, NUM_POINTS);
```

次に、図 13.2（189 ページ）に対応する以下を実行する。

```
        bindRTexture(textureZ, 'tx', 2);
        bindRTexture(textureY, 'ty', 1);
        bindWTexture(textureX);

        gl.drawArrays(gl.TRIANGLE_STRIP, 0, NUM_POINTS);
```

これを loop2 回繰り返し、最後に gl.finish() を実行し、全体の実行を終える。なお、途中の gl.drawArrays() の連続実行の途中に gl.finish() は不要である。

```
let gl;
let program;

let width = 256;
let height = 256;
let alpha = 0.9;

function initSystem() {
    ** 前章と同じ **
}

let NUM_POINTS = 4;

let textureX;
let textureY;
let textureZ;

function initData() {
    let position = [
        -1.0, +1.0,
        -1.0, -1.0,
        +1.0, +1.0,
        +1.0, -1.0
    ];
    let buffer1 = buildArrayBuffer(position);
    bindArrayBuffer(buffer1, 'position', 2);

    setUniformFloat('width', width);
    setUniformFloat('height', height);
    setUniformFloat('alpha', alpha);

    let xData = new Float32Array(4*width*height);
    let yData = new Float32Array(4*width*height);

    for (let w = 0; w < width; w++) {
        for (let h = 0; h < height; h++) {
            let k = 4*(h*width+w);
            xData[k+0] = w+0.1;
            xData[k+1] = w+0.2;
            xData[k+2] = w+0.3;
            xData[k+3] = w+0.4;

            yData[k+0] = h+0.1;
            yData[k+1] = h+0.2;
            yData[k+2] = h+0.3;
            yData[k+3] = h+0.4;
        }
    }
```

図 13.3: ピンポン計算を行う JavaScript ホストプログラム `main.js`（その 1、その 2 へ続く）

```
    textureX = buildTexture(xData, width, height);
    textureY = buildTexture(yData, width, height);
    textureZ = buildTexture(null, width, height);
}

let loop2 = 10;

function compute() {
    for (let i = 0; i < loop2; i++) {
        bindRTexture(textureX, 'tx', 0);
        bindRTexture(textureY, 'ty', 1);
        bindWTexture(textureZ);

        gl.drawArrays(gl.TRIANGLE_STRIP, 0, NUM_POINTS);

        bindRTexture(textureZ, 'tx', 2);
        bindRTexture(textureY, 'ty', 1);
        bindWTexture(textureX);

        gl.drawArrays(gl.TRIANGLE_STRIP, 0, NUM_POINTS);

    }
    gl.finish();
}

function checkResult() {
    let resultGPU = new Float32Array(4*width*height);
    readFromTexture(resultGPU, width, height);

    let resultCPU = new Float32Array(4*width*height);
    for (let h = 0; h < height; h++) {
        for (let w = 0; w < width; w++) {
            let k = 4*(h*width+w);
            for (let i = 0; i < 3; i++) {
                let x = w+0.1*(i+1);
                let y = h+0.1*(i+1);
                for (let i = 0; i < 2*loop2; i++) {
                    x = alpha*x+y;
                }
                resultCPU[k+i] = x;
            }
        }
    }
```

図 13.4: ピンポン計算を行う JavaScript ホストプログラム `main.js`（その 2、その 3 へ続く）

```
    let error = 0;
    for (let h = 0; h < height; h++) {
        for (let w = 0; w < width; w++) {
            let k = 4*(h*width+w);
            for (let i = 0; i < 3; i++) {
                error += Math.abs(resultGPU[k+i]-resultCPU[k+i]);
                         /Math.abs(resultCPU[k+i]);
            }
        }
    }
    alert(error/4/height/width);
}


window.onload = function(){
    initSystem();
    initData();
    compute();
    checkResult();
};
```

図 13.5: ピンポン計算を行う JavaScript ホストプログラム main.js（その 3）

関数 checkResults() ではまず、冒頭のプログラム片：

```
    let resultGPU = new Float32Array(4*width*height);
    readFromTexture(resultGPU, width, height);
```

で GPU の実行結果を取り出す。

次に、同じ計算を CPU で行うのが以下の部分である。

```
    let resultCPU = new Float32Array(4*width*height);
    for (let h = 0; h < height; h++) {
        for (let w = 0; w < width; w++) {
            let k = 4*(h*width+w);
            for (let i = 0; i < 3; i++) {
                let x = w+0.1*(i+1);
                let y = h+0.1*(i+1);
                for (let i = 0; i < 2*loop2; i++) {
                    x = alpha*x+y;
                }
                resultCPU[k+i] = x;
            }
        }
    }
```

さらに GPU での計算値と CPU での計算値の比較（差の絶対値の平均の計算）を行う部分が以下である。

```
let error = 0;
for (let h = 0; h < height; h++) {
    for (let w = 0; w < width; w++) {
        let k = 4*(h*width+w);
        for (let i = 0; i < 3; i++) {
            error += Math.abs(resultGPU[k+i]-resultCPU[k+i])
                     /Math.abs(resultCPU[k+i]);
        }
    }
}
alert(error/4/height/width);
```

13.1.2　バーテック/フラグメントシェーダプログラム

12.6 節（176 ページ）と同じものを利用できる。

13.1.3　実行の様子

図 13.3（191 ページ）、図 13.4（192 ページ）に載せているように

```
let width = 256;
let height = 256;
...
let loop2 = 10;
```

の条件で実行したところ、講義担当者の PC での平均相対誤差は以下の通りであった。

```
1.5461998589054O6e-7
```

誤差は `width`、`height` の値にはほぼ依存せず、`loop2` の増加と共に誤差も増加する傾向が見られる。長い連鎖の計算では誤差が累積するからである。

```
let before;
let after;                                          // 大域変数として宣言しておく

...

    before = Date.now();

    ここに計算部分のプログラムを置く

    after = Date.now();
    alert(0.001*(after-before));                              // 時間差を表示
```

図 13.6: 実行時間の計測方法

13.2　演算性能の計測

13.2.1　実行時間の計測

この節では JavaScript における実行時間の計測方法について概説する。

JavaScript において実行時間を測定する方法はいくつか考えられるが、今回は図 13.6（195 ページ）のようなプログラムを用いる。

`Date.now()` は `Date` クラスの静的メソッドであり、 UTC の 1970 年 1 月 1 日 00:00:00 からメソッド呼び出し時までの経過時間をミリ秒で返す。よって、時間を計測したい前後の `Date.now()` の差分を求めれば、それが実行時間（ミリ秒単位）である。表示時には 0.001 倍すれば、秒単位へ変換できる。

13.2.2　演算性能の基本式

1GFLOPS は、1 秒当たりに 10^9 回の加減乗算を実行できることを指す。除算は加減乗算に比べて演算コストが一桁ほど高いため、通常、演算性能を論じる際には用いない。

おおよその GFLOPS 値は、プログラムに含まれる総演算数を実行時間で割ることで求めることができる。この節の saxpy のくり返し計算の総演算数 $N(W, H, L)$ は以下の通りである。

$$N(W, H, L) = 2 \cdot 4 \cdot W \cdot H \cdot L \tag{13.1}$$

ここに因子 2 は、SAXPY 計算では加算と乗算の 2 回の演算を行うためである。因子 4 は、フラグメントシェーダでは R、G、B、A の 4 個分の計算を一度に行う SIMD 計算を意味する。W、H はテクスチャの水平方向、垂直方法のサイズ、L は繰り返し回数である。よって、このプログラムでの演算性能は以下のプログラム片で計算できる。

```
(2*4*width*height*2*loop2) / (0.001*(after-before) * 1e9);
```

13.3　ピンポン計算の演算性能

13.1 節（188 ページ）のプログラムに時間計測のプログラム片を追加し、PC の演算性能を測定する。

13.3.1　ホストプログラム

GPU の演算性能を知るに、図 13.7（198 ページ） のように、入力データを CPU から GPU へ転送する直前から計算結果を GPU から CPU へ転送した直後までの実行時間を測る。つまり、以下の区間である。

```
beforeGPU = Date.now();                          // GPU の実行時間計測開始

textureX = buildTexture(xData, width, height);
textureY = buildTexture(yData, width, height);
textureZ = buildTexture(null, width, height);

....

let resultGPU = new Float32Array(4*width*height);
readFromTexture(resultGPU, width, height);

afterGPU = Date.now();                           // GPU の実行時間計測終了
```

そして、演算性能を前節の式に基づいて以下のように計算出力する。

```
let durationGPU = 0.001*(afterGPU-beforeGPU);
alert(durationGPU + " " + 2*4*width*height*2*loop2/durationGPU/1e9);
```

CPU の演算性能も同様である。

```
** 前略 **

let beforeGPU;                                          // GPU 実行の時間計測用
let afterGPU;                                           // 同上
let beforeCPU;                                          // CPU 実行の時間計測用
let afterCPU;                                           // 同上

function initData() {
    ** 中略 **

    beforeGPU = Date.now();                            // GPU の実行時間計測開始

    textureX = buildTexture(xData, width, height);
    textureY = buildTexture(yData, width, height);
    textureZ = buildTexture(null, width, height);
}

** 中略 **

function compute() {
    ** 中略 **
}

function checkResults() {
    let resultGPU = new Float32Array(4*width*height);
    readFromTexture(resultGPU, width, height);

    afterGPU = Date.now();                             // GPU の実行時間計測終了

    beforeCPU = Date.now();                            // GPU の実行時間計測開始
    let resultCPU = new Float32Array(4*width*height);
    for (let h = 0; h < height; h++) {
        ** 中略 **
    }
    afterCPU = Date.now();                              // CPU の実行時間計測開始

    let durationGPU = 0.001*(afterGPU-beforeGPU);
    alert(durationGPU + " " + 2*4*width*height*2*loop2/durationGPU/1e9);

    let durationCPU = 0.001*(afterCPU-beforeCPU);
    alert(durationCPU + " " + 2*4*width*height*2*loop2/durationCPU/1e9);
}

** 後略 **
```

図 13.7: 演算性能を調べる JavaScript ホストプログラム `main.js`

表 13.1: テクスチャサイズと演算性能（WebGL、Safari、iMac 27 inch (2017)）

	GPU		CPU	
サイズ	時間	性能	時間	性能
4^2	1.97	0.000130	0.001	0.256
8^2	1.97	0.000520	0.002	0.512
16^2	1.98	0.00207	0.005	0.819
32^2	1.98	0.00827	0.014	1.17
64^2	1.98	0.0330	0.047	1.39
128^2	1.98	0.132	0.182	1.44
256^2	1.99	0.527	0.716	1.46
512^2	1.99	2.11	2.83	1.48
1024^2	2.10	8.01	11.3	1.49
2048^2	3.71	18.08	45.24	1.48
4096^2	13.3	20.2	–	–

時間の単位は秒、速度の単位は GFLOPS

表 13.2: 演算性能（OpenGL、iMac 27 inch (2017)）

	GPU		CPU	
サイズ	時間	性能	時間	性能
2048^2	2.20	30.5	13.9	4.84

時間の単位は秒
速度の単位は GFLOPS

13.3.2　測定結果

表 13.1 は、講義担当者の PC 上で Web ブラウザ Safari を用いて、`loop2` を 1000 に固定し、`width`、`height` を共に 4、8、...、4096 と変えながら実行時間と演算性能を求めたものである。それをグラフ化したものが図 13.8（200 ページ）である。

まず、表 13.1 から GPU の最大演算性能は約 20GFLOPS、CPU のそれは約 1.5GFLOPS である。大規模計算では GPU の性能が圧倒的に高いことがわかる。逆に小規模な計算では CPU が高く、GPU は非常に低い。計算規模に応じて両者を使い分けるべきである。

実験で用いた PC に搭載された GPU（AMD Radeon Pro580）のカタログ・ピーク性能は約 6TFLOPS（=6,000GFLOPS）である。一般に GPU のカタログ・ピーク性能を実際の計算プログラムで出すことはほとんどできないが、それにしてもこの実験の 20GFLOPS はかなり小さい。その理由として、WebGL の場合、Web ブラウザ自体が計算専用ソフトウェアではなく、計算以外の処理が多数あること、インタプリタ言語である JavaScript の実行速度の遅さ等が考えられる。参

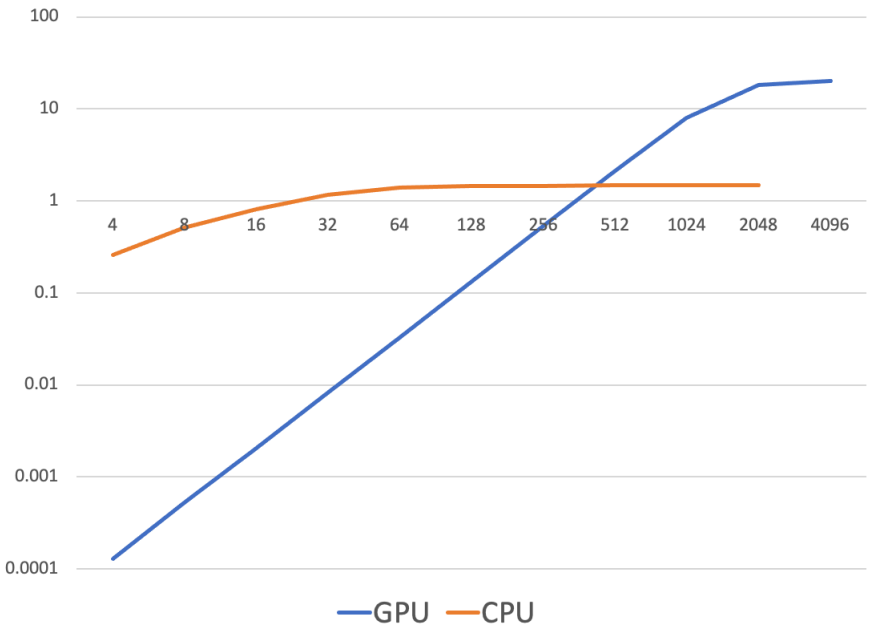


図 13.8: 表 13.1 のグラフ：横軸は画像の一辺の長さ、縦軸は演算性能（グラフ化に当たり、より詳細は実験を行なった。両軸とも対数目盛りであることに注意）

表 13.3: 演算性能（WebGL、Firefox、iMac 27 inch (2017)）

	GPU		CPU	
サイズ	時間	性能	時間	性能
2048^2	3.81	17.6	46.7	1.44

時間の単位は秒
速度の単位は GFLOPS

考までに、同じ計算を WebGL ではなく、OpenGL と C++で実装した場合の測定結果を表 13.2 に示す。GPU 性能で WebGL の場合の約 1.5 倍、CPU 性能で WebGL の場合の約 3 倍の性能を出している。

また、同じ PC 上であっても、Web ブラウザが異なれば、性能が違ってくる。表 13.3 は、Firefox を用いた場合の測定結果である。Safari に比べてやや性能が劣っている。

少し詳しい解析のために、総演算数 N の計算にかかる計算時間 T を単純な N の線形近似で

$$T = A + B \cdot N \tag{13.2}$$

で表すことを考えよう。ここに A、B は正定数である。A は計算のオーバーヘッド、B は 1 回の演算にかかるコストと考えられる。総演算数 N を実行時間 T で割った値 P は演算性能（単位時

間に実行できる演算数）を表すが、それは次式の通りである。

$$P = \frac{N}{T} = \frac{N}{A + B \cdot N} \tag{13.3}$$

ここで検討している例題について定数 A、B を求めてみる。

最小二乗法を用いるまでもないだろう。まず、GPU について表 13.1 の 1 行目ではほぼオーバヘッド A_{GPU} のみと考えると、

$$A_{\mathrm{GPU}} = 1.97$$

となる。次に、表 13.1 の 2048^2 のサイズについて

$$3.71 = 1.97 + B_{\mathrm{GPU}} \cdot (2 \cdot 4 \cdot 2048^2 \cdot 2 \cdot 1000)$$

を解くと、

$$B_{\mathrm{GPU}} = 2.59 \times 10^{-11}$$

となる。なお、4096^2 の測定結果は線形補間からやや逸脱する数値になっているため、除外する。逸脱の理由は計算サイズが大きくなりすぎて、キャッシュ溢れ等の影響で演算性能が相対的に落ちているためと思われる。

CPU については、その性質から

$$A_{\mathrm{CPU}} = 0.0$$

とおく。そして表 13.1 の 2048^2 のサイズについて

$$45.24 = B_{\mathrm{CPU}} \cdot (2 \cdot 4 \cdot 2048^2 \cdot 2 \cdot 1000)$$

を解くと、

$$B_{\mathrm{CPU}} = 6.74 \times 10^{-10}$$

となる。

以上をまとめると、GPU、CPU の演算性能はそれぞれ以下の通りである。

$$\begin{aligned} P_{\mathrm{GPU}} &= \frac{N}{1.97 + 2.59 \times 10^{-11} \cdot N} \rightarrow 38.6 \text{ GFLOPS} \quad (N \rightarrow \infty) \\ P_{\mathrm{CPU}} &= \frac{N}{6.74 \times 10^{-10} \cdot N} = 1.48 \text{ GFLOPS} \end{aligned}$$

13.4　より高速な計算

この節の例題がこの講義の最後の例題である。

前章ではフラグメントシェーダ内で 1 回の積和計算（詳しく言えば、4 並列の SIMD 型積和計算）

```
gl_FragColor = alpha*x+y;
```

を行い、これをピンポン計算で L 回繰り返した。

前節最後に CPU、GPU 間のデータ転送のオーバーヘッドを論じたが、実はシェーダの起動も大きなオーバーヘッドである。そこで、シェーダを繰り返し実行するのではなく、1 回のフラグメントシェーダの計算の中にループ：

```
for (int k = 0; k < L; k++) {
    x = alpha*x+y;
}
```

を作り、全てここに押し込む改造を行う。

プログラムは以下の通りである。

13.4.1　ホストプログラム

まず、図 13.10（204 ページ）のように、3 枚のテクスチャの設定および関数 `compute()` は、12.6 節（176 ページ）の場合（あるいは 12.7 節（182 ページ）の方が見やすい）の単純な設定に戻す。

13.4.2　バーテックスシェーダプログラム

バーテックスシェーダプログラムは図 13.12（206 ページ）の前半だが、前章と同じである。

13.4.3　フラグメントシェーダ

フラグメントシェーダプログラムは図 13.12（206 ページ）の後半である。

前節で

```
gl_FragColor = alpha*x+y;
```

であった部分を図 13.12（206 ページ）では

```
let gl;
let program;

let width = 2048;
let height = 2048;
let alpha = 0.9;

function initSystem() {
    let c = document.getElementById('canvas');
    c.width = width; c.height = height;

    gl = c.getContext('webgl');

    gl.clearColor(0.0, 0.0, 0.0, 1.0);

    let flg = (gl.getExtension('OES_texture_float') != null) &&
              (gl.getExtension("OES_texture_float_linear") != null);
    if (!flg) {
        alert('float texture not supported');
        return;
    }

    let frameBuffer = gl.createFramebuffer();
    gl.bindFramebuffer(gl.FRAMEBUFFER, frameBuffer);

    buildProgram('vs', 'fs');
    gl.useProgram(program);
}

let NUM_POINTS = 4;

let beforeGPU;
let afterGPU;
let beforeCPU;
let afterCPU;

function initData() {
    let position = [
        -1.0, +1.0,
        -1.0, -1.0,
        +1.0, +1.0,
        +1.0, -1.0
    ];
    let buffer1 = buildArrayBuffer(position);
    bindArrayBuffer(buffer1, 'position', 2);
```

図 13.9: saxpy の高速計算を行う JavaScript ホストプログラム `main.js`（その 1、その 2 へ続く）

```
    setUniformFloat('width', width);
    setUniformFloat('height', height);
    setUniformFloat('alpha', alpha);

    let xData = new Float32Array(4*width*height);
    let yData = new Float32Array(4*width*height);

    for (let w = 0; w < width; w++) {
        for (let h = 0; h < height; h++) {
            let k = 4*(h*width+w);
            xData[k+0] = w+0.1;
            xData[k+1] = w+0.2;
            xData[k+2] = w+0.3;
            xData[k+3] = w+0.4;

            yData[k+0] = h+0.1;
            yData[k+1] = h+0.2;
            yData[k+2] = h+0.3;
            yData[k+3] = h+0.4;
        }
    }

    beforeGPU = Date.now();

    let textureX = buildTexture(xData, width, height);
    let textureY = buildTexture(yData, width, height);
    let textureZ = buildTexture(null, width, height);

    bindRTexture(textureX, 'tx', 0);
    bindRTexture(textureY, 'ty', 1);
    bindWTexture(textureZ);
}

let loop2 = 1000;

function compute() {
    gl.drawArrays(gl.TRIANGLE_STRIP, 0, NUM_POINTS);
    gl.finish();
}

function checkResults() {
    let resultGPU = new Float32Array(4*width*height);
    readFromTexture(resultGPU, width, height);
    let afterGPU = Date.now();
```

図 13.10: saxpy の高速計算を行う JavaScript ホストプログラム `main.js`（その 2、その 3 へ続く）

```
    let beforeCPU = Date.now();

    let resultCPU = new Float32Array(4*width*height);
    for (let h = 0; h < height; h++) {
        for (let w = 0; w < width; w++) {
            let k = 4*(h*width+w);
            for (let i = 0; i < 3; i++) {
                let x = w+0.1*(i+1);
                let y = h+0.1*(i+1);
                for (let i = 0; i < 2*loop2; i++) {
                    x = alpha*x+y;
                }
                resultCPU[k+i] = x;
            }
        }
    }
    afterCPU =Date.now();

    let durationGPU = 0.001*(afterGPU-beforeGPU);
    alert(durationGPU + " " + 2*4*width*height*2*loop2/durationGPU/1e9);

    let durationCPU = 0.001*(afterCPU-beforeCPU);
    alert(durationCPU + " " + 2*4*width*height*2*loop2/durationCPU/1e9);
}

window.onload = function(){
    initSystem();
    initData();
    compute();
    checkResults();
};
```

図 13.11: saxpy の高速計算を行う JavaScript ホストプログラム `main.js`（その 3）

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<script src="./main.js" type="text/javascript"></script>
<script src="./util.js" type="text/javascript"></script>
<script src="./compileLink.js" type="text/javascript"></script>

<script id="vs" type="text/plain">
    attribute vec2 position;

    void main(void)
    {
        gl_Position = vec4(position, 0.0, 1.0);
    }
</script>

<script id="fs" type="text/plain">
    precision highp float;

    uniform sampler2D tx;
    uniform sampler2D ty;

    uniform float alpha;
    uniform float width;
    uniform float height;

    void main(void)
    {
        vec2 texCoord = vec2(gl_FragCoord.x/width,
                             gl_FragCoord.y/height);

        vec4 x = texture2D(tx,texCoord);
        vec4 y = texture2D(ty,texCoord);

        for (int k = 0; k < 2000; k++) {
            x = alpha*x+y;
        }
        gl_FragColor = x;
    }
</script>

</head>
<body>
<canvas id="canvas"></canvas>
</body>
</html>
```

図 13.12: saxpy の高速計算を行う HTML/シェーダプログラム

表 13.4: 演算性能（WebGL、Safari、iMac 27 inch (2017)）

	GPU		CPU	
サイズ	時間	性能	時間	性能
2048^2	0.205	327	45.1	1.49

時間の単位は秒
速度の単位は GFLOPS

表 13.5: 演算性能（OpenGL、iMac 27 inch (2017)）

	GPU		CPU	
サイズ	時間	性能	時間	性能
2048^2	0.116	579	14.0	4.78

時間の単位は秒
速度の単位は GFLOPS

```
        for (int k = 0; k < 2000; k++) {
            x = alpha*x+y;
        }
        gl_FragColor = x;
```

に置き換える。ここにループ回数の定数値 2000 をそのままプログラムに埋め込んでいるのは、現時点での WebGL/GLSL の言語仕様が貧弱で、そこに uniform 変数を用いるなどができないためである。

13.4.4　測定結果

図 13.4（207 ページ）は、2048 × 2048 の画面サイズについて 2,000 回繰り返した場合の実行結果である。

この実行例では GPU が 300GFLOPS を超える性能を達成している。CPU との差は明白である。この例はやや特殊であるが、工夫すればかなり高い演算性能を出すことができることが分かる。参考までに図 13.5（207 ページ）には同じ計算を OpenGL と C++で実装し、実行した場合の測定結果を示す。GPU では 600GFLOPS 近い性能を達成している。

第14章　まとめ

この講義では、WebGL/GLSL を用いた描画の話から始まった。GPU のシェーダを用いたプログラムは本質的に並列プログラムであり、その並列性を描画を通して直感的に理解することを試みた。次にテクスチャを導入し、シェーダで配列データを読み込むことが可能であることを示した。さらにテクスチャへの書き込みを方法を紹介し、これを用いて GPU で数値計算する方法を示した。

現在は CUDA などの GPGPU 専用の環境が利用できるから、この講義テキストの数値計算方法が GPGPU の主流になることはない。しかし、「なぜグラフィックス専用ハードウェアが高速計算に利用できるのか」という素朴な問いへの答えを教えてくれる。

（了）