佐賀大学理工学部
知能情報システム学科

「高性能計算特論」
講義資料

担当者: 山下義行

R2 年度前期

目次（変更される場合あり）

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 OpenGL/GLSL を用いてその概要を学ぶ。

知能情報システム学科 2 年生科目「コンピュータグラフィックス」でも 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、まず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ことになった経緯が分かるはずである。次に GPGPU の専用環境として開発された OpenCL を取り上げて解説する。名前からも推察されるように OpenCL と OpenGL は密接に関連している。その辺りについても言及する予定である。

この講義では、Windows および MacOS 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。プログラムは、順次、Live Campus にアップロードするから、実際の動かしてほしい。

[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。

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。そのため、この講義テキストで示すプログラムの構造やプログラム片がそのホームページの内容に類似しているものがあることを事前に告知しておく。プログラムをそのまま利用した箇所ではその旨を記載する。

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平先生に感謝いたします。

[2] http://marina.sys.wakayama-u.ac.jp/~tokoi

第2章　OpenGLの概要

この章では OpenGL/GLSL を用いた CG プログラムの概要について解説する。

2.1　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**シェーダープログラム**からなる。一方だけでは動かない[1]。

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、通常のプログラム実行と同様に CPU 上で動作する。この講義ではその制御にグラフィックス API（Application Program Interface）として最もよく普及している OpenGL を用いる。

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]。

GLSL は、OpenGL Shading Language の略であり、OpenGL 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 API は用意されていなかったが、急速に開発が進む GPU を効果的に利用するために、OpenGL からほぼ 10 年遅れて仕様が策定され、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。

---

[1] 古い OpenGL ではシェーダープログラムを作成しない場合、システムが用意する固定機能のシェーダープログラムが自動的に仮定された。よって初心者はシェーダーを意識することなく CG プログラムを作ることができたが、現在の OpenGL では処理の多様性、高速性を強く意識するため、ユーザは必ずシェーダープログラムを用意せねばならない。結果として OpenGL は初心者には難易度が高いものになっている。

[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。

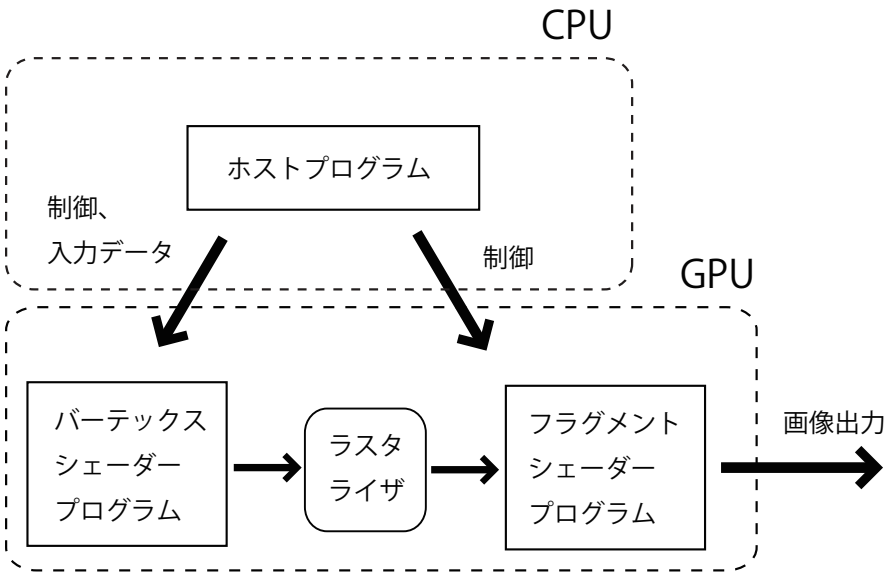

図 2.1: 3 種類のプログラムの関係

GLSL のプログラム（以下、シェーダープログラム）は OpenGL を用い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ー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ープログラムは、**バーテックスシェーダープログラム**と**フラグメントシェーダープログラム**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（図 2.1）。詳細は後に述べる。

結局、OpenGL/GLSL で 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
2. バーテックスシェーダープログラム
3. フラグメントシェーダープログラム

を書かねばならない。

実は、最近の CG 描画では図 2.1 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ー、テッセレータなどのプログラムを追加できる（追加は必須ではない）。また、計算に特化した計算シェーダープログラムも利用でき、これがこの講義の後半で述べる OpenCL に密接に関連している。

2.2　ホストプログラムの構造

OpenGL について注意すべき点は、OpenGL 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 — の処理に関する仕様であって、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 – の処理は OpenGL の仕様には含まれない。

コア部分に OpenGL という統一仕様があるとしても、コア以外の部分は オペレーティングシステム（OS）やそれに付随するウインドウシステムに強く依存するため、その仕様をひとつに決めることが難しい。本格的な CG ソフトウェアを開発する場合、そのソフトウェアの操作は搭 OS のウインドウシステムの機能を利用し、その OS 操作感に合うように作るべきである。その場合、OS 独自の処理を細かく実装する必要があり、他の種類の OS 上で全く同じプログラムが動くことは期待できない。

もし操作性にあまりこだわらず、手早く CG プログラムを作りたいだけならば、コア以外については様々な PC、OS に共通する必要最小限の機能のみを提供すればよいだろう。そのような目的の API として **GLUT**（OpenGL Utility Toolkit）がしばしば利用されてきた。この講義でも GLUT を用いる。CG の学習という目的ではそれで十分である。

GLUT は OpenGL の仕様が公開された時にほとんど時を同じくして提案された API であり、面倒なウインドウの生成などを簡単な関数呼び出しで実現できる。また描画の無限ループをサポートする関数が用意されており、API というよりも、プログラム全体の構造を決めるアプリケーションフレームワークを提供している。なお、GLUT は既に開発が終了して久しいライブラリであり、MacOS においては「deprecated」（サポートが廃止される可能性があり、使用が推奨されない）と警告されている。早晩利用できなくなると思われる。しかし新しい API — たとえば GLFW[5] — に乗り換える手間は決して小さくない。特に講義で学ぶだけのための面倒なインストール作業、そして講義終了後のアンインストール作業を強制することにいは躊躇する。その点を考慮し、結局、この講義では GLUT を使用する。

OpenGL 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、OpenGL/GLUT ライブラリの定型的な初期化、描画ウインドウの生成などを行う。

シェーダー実行可能プログラムの準備： シェーダープログラムのコンパイル･リンク、GPU への転送、登録を行う。

[5]講義担当者は OpenGL を使う研究用プログラムでは既に GLUT から GLFW に乗り換えた。

シェーダー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。

シェーダー実行可能プログラムの起動： シェーダー実行可能プログラムを起動する。

補足： 上記「シェーダー実行可能プログラムの準備」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（ソースコード）、オブジェクトプログラム（オブジェクトコード）、実行可能プログラム（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「シェーダープログラム」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「シェーダーソースプログラム」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「シェーダーオブジェクトプログラム」と呼ぶこととする。さらに、それを（それらを）リンクしたものを「シェーダー実行可能プログラム」と呼ぶこととする。

現在、シェーダーの機械語の仕様は、それがハードウェアに近い低レベルのものであれ、抽象度の高いレベルであれ、統一の仕様は策定されていない。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供給されている状況である。その結果、シェーダー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 OpenGL 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総じて遅くなる。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。もしシェーダー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（必要ならば数十分、数時間）を掛けて高度で複雑な最適化処理を実施できる[7]。

3. シェーダーソースプログラムをコンパイルして生成される機械語コードを見る（読む）ことができない[8]。よって、ユーザーは実際に GPU 上でどのような機械語コー

[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、膨大なプログラム解析時間を必要とする。

[7] たとえば Metal のシェーダープログラムは Xcode 環境で事前にコンパイルできる。

[8] リバースエンジニアリングを用いれば可能と思われるが、法的に問題があり、やるべきではない。

ドが実行されるのか知ることができず、シェーダーソースプログラムを書き直すなどのユーザレベルのプログラム最適化が難しい。

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。

OpenGL の仕様はプログラミング言語とは独立しており、様々なプログラミング言語から利用可能である。この講義では最も一般的な C++を用いる。

2.3　シェーダープログラムの構造

既に述べたように、ユーザは少なくとも 2 種類のシェーダー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ー上で動作し、以下のような役割を担っている。

バーテックス・シェーダー（vertex shader） ... 頂点（vertex）シェーダーともいう。3D CG では主に頂点の座標変換を行うため、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ーへの入力は、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。たとえば線分は 2 端点で定義され、三角形は 3 頂点で定義される。その違いに応じてバーテックスシェーダーの動作を簡単に変更できる。

計算結果はバーテックスシェーダーから出力され、ラスタライザに入力される。

フラグメント・シェーダー（fragment shader） DirectX ではピクセル（pixel=画素）シェーダーと呼ぶ。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ー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ーに入力される。フラグメントシェーダー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）は画像バッファ上の対応する画素へ書き込まれる。

シェーダープログラムは 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、ベクトルデータ型の拡張等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。

```
01 void display()
02 {
03     /* ここに描画処理 */
04 }
05
06 int main()
07 {
08     /* ここに初期化処理 */
09
10     glutDisplayFunc(display);     // 描画関数を callback 関数として登録
11     glutMainLoop();              // 無限ループで処理を繰り返すことの設定
12     return 0;
13 }
```

図 2.2: ホストプログラムのひな形（その 1）

2.4　OpenGL/GLSL プログラムの外形

GLUT を用いたホストプログラムの最も単純な形は図 2.2 の通りである。

GLUT では**コールバック**（callback）を用いて描画を行なう。そこでまず、関数呼び出し

```
10     glutDisplayFunc(display);     // 描画関数を callback 関数として登録
```

によって、関数 `display()` を GLUT の実行時管理テーブルへ登録しておく。次に、関数呼び出し

```
11     glutMainLoop();               // 無限ループで処理を繰り返すことの設定
```

によってプログラムは GLUT のイベント処理ループへ入る。このループでは、GLUT の実行時システムがウインドウの描画/再描画が必要と判断したとき（たとえばプログラム開始直後や描画ウインドウの状態が変化したとき）に `glutDisplayFunc()` によって事前に登録したコールバック関数（すなわち、`display()`）が自動的に呼び出されるようになっている。このため描画のタイミングをユーザがいちいち管理する必要はなく、プログラムを容易に作成できるようになっている（そもそもこの種の管理を一般ユーザに任せるのは難易度が高すぎる）。

上のひな形は一見すると分かりやすいが、実際には初期化処理を `main` 関数にベタ書きする形になっており、プログラム作法上あまり好ましくない。そこでこの講義では、システム関連の初期化を行う関数 `initSystem()` と被写体データの初期化を行う関数 `initData()` を定義し、ひな形を図 2.3 のように修正する。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。

```
01 void initSystem()
02 {
03     /* ここに OpenGL、GLUT、その他の初期化処理 */
04 }
05
06 void initData()
07 {
08     /* ここにプログラムで利用するデータの初期化処理 */
09 }
10
11 void display()
12 {
13    /* ここに描画処理 */
14 }
15
16 int main()
17 {
18     initSystem();                    // OpenGL、GLUT、その他の初期化
19     initData();                  // プログラムで利用するデータの初期化
20
21     glutDisplayFunc(display);   // 描画関数を callback 関数として登録
22     glutMainLoop();            // 無限ループで処理を繰り返すことの設定
23     return 0;
24 }
```

図 2.3: ホストプログラムのひな形（その 2）

2.5　講義の進め方

次章以降、実際の OpenGL/GLSL のプログラムを紹介していく。

個々のプログラムは、原則として、節毎にひとつのプログラムを紹介する。3 章で言えば、3.1 節、3.2 節でそれぞれひとつずつ、計 2 種類のプログラムを紹介する。

各節では、初めに節の冒頭にプログラムを全て示す。プログラムはいくつかのファイルに分かれている。ファイルの内容に全く変更がない場合に限り、そのファイルの内容の提示を省略する。変更があるファイルについては、わずかな変更であってもファイルの全内容を示す。

プログラム中の、コメントの付いている行が新たに導入された機能を用いている行である。その行については、節のそれ以降で解説する。コメントのついていない行は、前の章、前の節で既出の内容である。

各節では、プログラムの全体を眺めた後で、個々の機能の説明を行う。

第3章　スタートアップ

この章で作成するCG画像は、×印を二本の線分で描く図3.1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。

この章では2種類のプログラムを紹介する。ひとつは、OpenGL/GLSLの関数をそのまま順に呼び出すプログラムである。可読性は低いが、典型的なOpenGLのプログラムである。もうひとつは、OpenGLの機能をクラス化し、可読性を高めたものである。次章以降も、このクラス化されたプログラムをベースに議論を進めていく。

3.1　簡単な線画：最も単純なプログラム

図3.1のような単純な線画を描画するだけのために**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？」と批判されている。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と思われる。その点で言えば、最近、新しく仕様策定されたMetal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。

c〜図3.4はホストプログラムである。これは図2.3のひな形に当てはめて書かれている。バーテックスシェーダープログラムは図3.5である。フラグメントシェーダープログラムは図3.6の通りである。以下にそのプログラムの内容を解説していく。

補足：　OpenGLで定義されている関数の名前には必ず **gl** という接頭語が付く。同様に、GLUTで定義されている関数の名前には必ず **glut** という接頭語が付く。

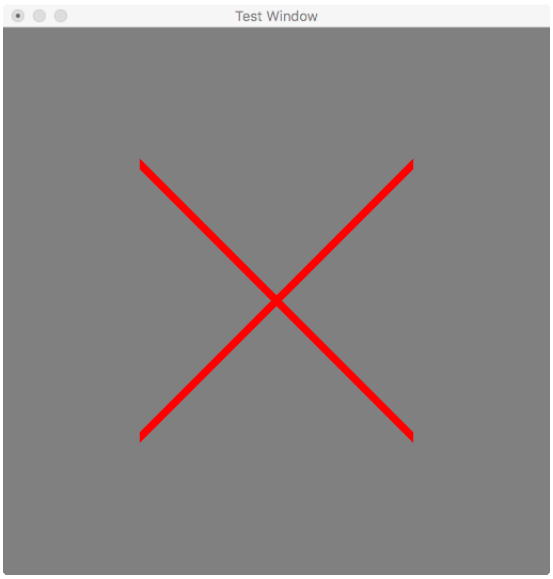

図 3.1: 画像例（その 1）

```
01 #include <iostream>
02 using namespace std;
03 #include <stdio.h>
04 #include <stdlib.h>
05
06 #if defined(WIN32)              // インクルードファイルを OS 毎に取捨選
択
07 #  pragma comment(lib, "glew32.lib")
08 #  include "glew.h"
09 #  include "glut.h"
10 #  include "glext.h"
11 #elif defined(__APPLE__) || defined(MACOSX)
12 #  include <GLUT/glut.h>
13 #else
14 #  define GL_GLEXT_PROTOTYPES
15 #  include <GL/glut.h>
16 #endif
17
18 void initSystem(int argc, char *argv[]);
                                    // OpenGL、GLUT、その他の初期化関数
19 void initData();              // プログラムで利用するデータの初期化関数
20 void display(void);                          // 1 枚の CG 画像の描画関数
21 GLuint compileProgram(GLenum type, const GLchar *file);
                                //シェーダープログラムのコンパイル関数
22
23 GLuint program;                     // シェーダープログラムの参照番号
24
25 void initSystem(int argc, char *argv[])
                                    // OpenGL、GLUT、その他の初期化関数
26 {
27     glutInit(&argc, argv);                            // GLUT の初期化
28     glutInitDisplayMode(GLUT_RGB|GLUT_SINGLE);
                                        // ウインドウの描画モードの設定
29     glutInitWindowSize(512,512);            // ウインドウサイズの設定
30     glutCreateWindow("Test Window");               // ウインドウの生成
31     glClearColor(0.5,0.5,0.5,1);       // ウインドウ内の背景色の設定
32
33 #if defined(WIN32)
34     glewInit();                        // Windows の場合、GLEW の初期化
35 #endif
36
37     GLuint vs = compileProgram(GL_VERTEX_SHADER, "shader.vert");
                           // バーテックスシェーダープログラムのコンパイル
38     GLuint fs = compileProgram(GL_FRAGMENT_SHADER, "shader.frag");
                         // フラグメント・シェーダープログラムのコンパイル
39
40     program = glCreateProgram();   // 空のシェーダープログラムの生成
41     glAttachShader(program, vs);
                               // バーテックスシェーダープログラムの追加
42     glAttachShader(program, fs);
                             // フラングメントシェーダープログラムの追加
```

図 3.2: ×印を描画するホストプログラム main.cpp（その 1、その 2 へ続く）

```
43
44      glLinkProgram(program); // シェーダープログラム、各種変数のリンク
45  }
46
47  const int NUM_POINTS = 4;                                   // 頂点の数
48
49  void initData()             // プログラムで利用するデータの初期化関数
50  {
51      float pos[2*NUM_POINTS];              // 2次元頂点を格納する配列
52
53      pos[0] = -0.5; pos[1] = -0.5;         // 左下の頂点 (-0.5,-0.5)
54      pos[2] = +0.5; pos[3] = +0.5;         // 右上の頂点 (+0.5,+0.5)
55      pos[4] = +0.5; pos[5] = -0.5;         // 右下の頂点 (+0.5,-0.5)
56      pos[6] = -0.5; pos[7] = +0.5;         // 左上の頂点 (-0.5,+0.5)
57
58      GLuint bufID;                       //バッファオブジェクトの参照番号
59      glGenBuffers(1, &bufID);                      // 参照番号の新規割付
60      glBindBuffer(GL_ARRAY_BUFFER, bufID);
                                  // その参照番号を配列バッファとして束縛
61
62      glBufferData(GL_ARRAY_BUFFER, sizeof(float)*2*NUM_POINTS,
                    pos, L_STATIC_DRAW);
                                 // 配列バッファに配列 pos のデータを転送
63      GLint p = glGetAttribLocation(program,"position");
                                    // 変数 position の位置 p の取得
64      glVertexAttribPointer(p, 2, GL_FLOAT, GL_FALSE, 0, 0);
                                                  // p の属性の設定
65      glEnableVertexAttribArray(p);                     // p の有効化
66
67      glLineWidth(10);                                  // 線幅の設定
68  }
69
70  void display(void)                          // 1 枚の CG 画像の描画関数
71  {
72      glUseProgram(program);                // プログラムを使用する宣言
73      glClear(GL_COLOR_BUFFER_BIT);  // 背景色によるウインドウのクリア
74      glDrawArrays(GL_LINES, 0, NUM_POINTS);
                                  // シェーダープログラムの実行開始
75      glFlush();                                  // 描画処理の強制実行
76  }
77
78  int main(int argc, char *argv[])
79  {
80      initSystem(argc,argv);             // OpenGL、GLUT、その他の初期化
81      initData();                   // プログラムで利用するデータの初期化
82
83      glutDisplayFunc(display);     // 描画関数を callback 関数として登録
84      glutMainLoop();             // 無限ループで処理を繰り返すことの設定
85      return 0;
86  }
```

図 3.3: ×印を描画するホストプログラム main.cpp（その 2、その 3 へ続く）

```
87 GLuint compileProgram(GLenum type, const GLchar *file)
88 {
89     FILE* fp = fopen(file, "r");        // シェーダーファイルの open
90
91 #define MAX_SHADER_FILE_SIZE 10000
92     GLchar ftext[MAX_SHADER_FILE_SIZE];
                                        // ファイルの内容を格納する配列
93
94     unsigned long n = fread(ftext, 1, MAX_SHADER_FILE_SIZE, fp);
                                                        // 読み込み
95     ftext[n] = '\0';                          // 末尾にヌル文字を設定
96     fclose(fp);                                  // ファイルの close
97
98     GLuint shader = glCreateShader(type);
                               // type 型の空のシェーダープログラムの作成
99     const GLchar* ftext_handle = ftext;           // ハンドルの設定
A0     glShaderSource(shader, 1, &ftext_handle, NULL);
                                        // ファイルの内容をハンドルで結合
A1
A2     glCompileShader(shader);                              // コンパイル
A3     return shader;
A4 }
```

図 3.4: ×印を描画するホストプログラム main.cpp（その 3）

```
01 #version 120
02
03 attribute vec2 position;                       // 線分データの各座標値
04
05 void main(void)
06 {
07     gl_Position = vec4(position, 0.0, 1.0);       // 描画位置の設定
08 }
```

図 3.5: ×印を描画するバーテックスシェーダープログラム shader.vert

```
01 #version 120
02
03 void main(void)
04 {
05     gl_FragColor = vec4(1.0, 0.0, 0.0, 0.0);// 赤色 (1,0,0) の描画
06 }
```

図 3.6: ×印を描画するフラグメントプログラム shader.frag

3.1.1　ホストプログラム

図 3.2 のプログラムを上から順に説明する。

インクルードファイルなど

図 3.2 冒頭の OpenGL 関連のインクルード・ファイルの宣言は、和歌山大学床井研究室のホームページにあるものをそのまま採用した。

大域変数の宣言

図 3.2 の大域宣言：

```
23 GLuint program;                    // シェーダープログラムの参照番号
```

は、コンパイル＆リンク後のシェーダー実行可能プログラムの参照番号（または管理番号）を保持するための変数の宣言である。参照番号は OpenGL のシステムによって自動的に割り当てられるから、ユーザーが具体的な番号を知る必要はない。ホストプログラムの複数の関数から参照されるため、これは大域化しておく。

注意：　「参照番号」に相当する、OpenGL のマニュアルの原語は “name”（直訳すると「名前」）であるが、実体は整数値であるため、無用な勘違いを避けるため、ここでは参照番号と訳すことにした。

`GLuint` は OpenGL が使用する unsigned int である。その実体は C/C++のそれと同じであるが、あえて別のデータ型名を用いることでそのデータがグラフィックス処理に用いられていることを明示し、プログラムの可読性を高めている。後に出てくる `GLfloat` なども同様である。

システムの初期化

ここでは関数 `initSystem()` について解説する。

```
27      glutInit(&argc, argv);                    // GLUT の初期化
```

上の 1 行は、GLUT ライブラリの初期化を行う。引数 `int argc, char *argv[]` はここでは使用しないが、慣例上、そのまま受け渡す。

```
28     glutInitDisplayMode(GLUT_RGB|GLUT_SINGLE);
                                        // ウインドウの描画モードの設定
```

上の 1 行は、描画ウインドウの設定である。GLUT_RGB は画像をフルカラーで表示することを意味し、GLUT_SINGLE は画像描画用フレームバッファを 1 枚だけ使用することを意味する。共にデフォルトの設定であるから実はこの関数呼び出しは不要である。後に CG アニメーションを行う場合には GLUT_SINGLE の代わりに GLUT_DOUBLE を指定する（アニメーションではダブルバッファを用いるため)。

```
29     glutInitWindowSize(512,512);          // ウインドウサイズの設定
```

上の 1 行は、描画ウインドウのサイズを 512 画素 × 512 画素 に設定する。

```
30     glutCreateWindow("Test Window");          // ウインドウの生成
```

上の 1 行は、”Test Window” というラベル名の付いたウインドウをコンピュータディスプレイ上に生成する [1]。

```
31     glClearColor(0.5,0.5,0.5,1);       // ウインドウ内の背景色の設定
```

上の 1 行は、画像の背景色、アルファ値（透明度）を $(r, g, b, a) = (0.5, 0.5, 0.5, 1)$（灰色、図 3.1 参照）に設定する。この時点では設定のみを行い、背景色による実際の塗りつぶしは描画用関数 display() の中の glClear() によって行う。なお、アルファ値はここでは使用しない。

```
33 #if defined(WIN32)
34     glewInit();                        // Windows の場合、GLEW の初期化
35 #endif
```

上の 3 行は、プログラムを Windows で実行する場合に必要である。glew は拡張ライブラリである。このライブラリはここでは陽には用いないが、この初期化を削除すると Windows での実行が正常に行われないため残しておく。

```
37     GLuint vs = compileProgram(GL_VERTEX_SHADER, "shader.vert");
                            // バーテックスシェーダープログラムのコンパイル
```

[1] glutCreateWindow() は int 型の戻り値を持つ。この戻り値はウインドウの参照番号である。ひとつのプログラムで複数のウインドウに画像描画するときにこの参照番号を利用し、ウインドウを切り替える。ウインドウを切り替える関数は void glutSetWindow(int win); である。

上の 1 行は、バーテックスシェーダープログラムのコンパイル処理である。関数 compileProgram() は図 3.4 に定義している通りだが、処理内容が複雑であるから、その詳細は後述する。この関数では第一引数にシェーダーの種類を OpenGL の定義する定数で指定する。バーテックスシェーダーの場合には GL_VERTEX_SHADER である。第二引数にはシェーダープログラムを格納するテキストファイル名を指定する。この講義ではバーテックスシェーダーのプログラムは "shader.vert" に格納するものと約束する [2]。その内容は図 3.5 の通りである。関数の戻り値はコンパイルされたシェーダーオブジェクトプログラムの参照番号である。具体的な参照番号を知る必要はない。

```
38      GLuint fs = compileProgram(GL_FRAGMENT_SHADER, "shader.frag");
                            // フラグメント・シェーダープログラムのコンパイル
```

上の 1 行は、フラグメントシェーダープログラムのコンパイル処理である。第一引数値がバーテックスシェーダーの場合と異なることに注意する。この講義ではフラグメントシェーダーのプログラムは "shader.frag" に格納すると約束する。ファイル "shader.frag" の内容は図 3.6 の通りである。

```
40      program = glCreateProgram();    // 空のシェーダープログラムの生成
```

上の 1 行は、空の プログラムを生成し、その参照番号を大域変数 program に格納する。program に格納される具体的な値を知る必要はない。

```
41      glAttachShader(program, vs);
                                  // バーテックスシェーダープログラムの追加
42      glAttachShader(program, fs);
                                // フラングメントシェーダープログラムの追加
```

上の 2 行は、シェーダーオブジェクトプログラム vs、fs を program にアタッチする（貼り付ける）。

```
44      glLinkProgram(program); // シェーダープログラム、各種変数のリンク
```

上の 1 行は、program にアタッチされたシェーダーオブジェクトプログラム間の入出力変数の矛盾などをチェックし、GPU 上で実行可能なシェーダー実行可能プログラムに仕上げる。

[2] 参考にした和歌山大学床井研究室のプログラムの名称をそのまま流用した。

データの初期設定

ここで考えている例題では、二本の線分で×印を描くだけ（図 3.1 参照）であるから、処理は比較的単純である。

まず、

```
47 const int NUM_POINTS = 4;                                    // 頂点の数
```

上の 1 行は、2 本の線分のそれぞれの両端点のために計 4 個の頂点を使用することを宣言する。頂点数は関数 `initData()` 、`display()` で参照するため、大域化しておく。

以下は、関数 `initData()` での処理である。

```
51      float pos[2*NUM_POINTS];                // 2 次元頂点を格納する配列
```

上の 1 行は、描画に必要な浮動小数点数値を格納する配列領域を確保する。なお、この例題では 2 次元座標値を扱うから、ひとつの頂点座標を保持するには `float` 型が 2 個必要である。頂点の個数は `NUM_POINTS` である。よって、必要とされる配列要素数は `2*NUM_POINTS` である。

注意： GPU では浮動小数点数値は倍精度（`double` 型）ではなく、単精度（`float` 型）で表現するのが一般的である [3]。何故ならば、CG 画像のサイズは縦横せいぜい数千画素である。各画素の輝度値は一般に RGB それぞれ 256 段階である。となれば、CG 画像の数値誤差が人の目に分からないためには、数値精度は 10 進数 4 桁程度が必要である。丸め誤差の発生等を考慮しても精度は 10 進数 6 桁もあれば足りるだろう。よって `float` 型で十分である。

次の 4 行

```
53      pos[0] = -0.5; pos[1] = -0.5;           // 左下の頂点 (-0.5,-0.5)
54      pos[2] = +0.5; pos[3] = +0.5;           // 右上の頂点 (+0.5,+0.5)
55      pos[4] = +0.5; pos[5] = -0.5;           // 右下の頂点 (+0.5,-0.5)
56      pos[6] = -0.5; pos[7] = +0.5;           // 左上の頂点 (-0.5,+0.5)
```

は、4 頂点の座標値を配列 `pos` に格納している。ここで配列中にデータを並べる順番には以下のルールがある。

[3] 最近では NVIDIA Telsa のように、 GPGPU のために倍精度計算を強化した GPU も開発・販売されている。

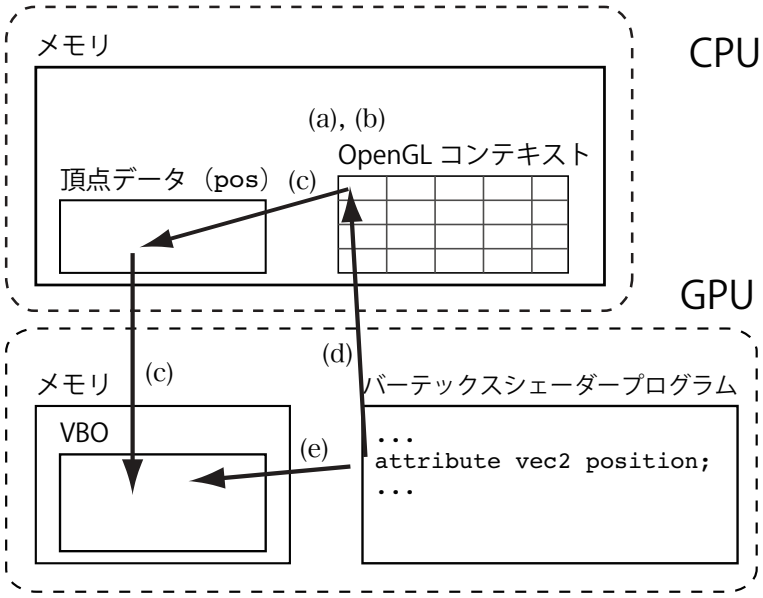

図 3.7: GPU メモリへの頂点データの転送：(a) OpenGL コンテキストにバッファオブジェクト `bufID` を生成する。(b) `bufID` を Vertex Buffer Object とみなし、これ以降の設定対象とする。(c) `bufID` に対応する GPU のメモリ領域を確保し、CPU のメモリからデータを転送する。(d) シェーダープログラムから特定の `attribue` 変数の場所 `p` を求める。(e) `bufID` と `p` を結合する。

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 [4] 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。

なお、OpenGL ではウインドウ上の座標は、ウインドウのアスペクト比に依らず、x 座標値は水平方向に $[-1, 1]$ の範囲、y 座標値は垂直方向に $[-1, 1]$ の範囲に固定されている。

[4] 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ープログラムでも逆順に取り扱う必要がある。

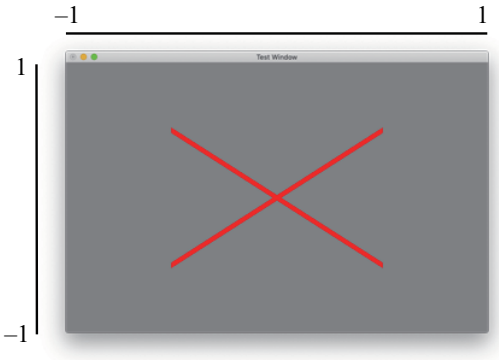

図 3.8: 横長のウインドウ上の座標

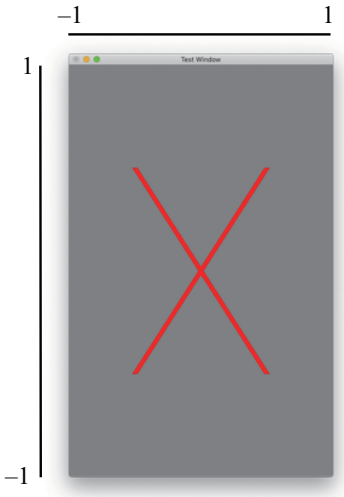

図 3.9: 縦長のウインドウ上の座標

よって図 3.8 のような横長のウインドウ、図 3.9 のような縦長のウインドウでは描画像も横長、縦長になる [5]。

ここからの 7 行は、配列 pos に設定した値を GPU に転送し、その値をバーテックスシェーダープログラムとリンクする処理である。この処理手順を図 3.7 を用いて説明する。**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済ます程度でよい。

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。GPU のメモリもその管理の対象である。

GPU のメモリには様々な種類のデータを格納するバッファオブジェクト（Buffer Object）を設けることができる。

```
58      GLuint bufID;                   //バッファオブジェクトの参照番号
```

上の 1 行は、ひとつのバッファオブジェクトの参照番号を保持する変数の宣言である。

```
59      glGenBuffers(1, &bufID);                // 参照番号の新規割付
```

上の 1 行は、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 3.7 で言えば、(a) の右上の表にひとつ項目が追加される。生成されたバッファオブジェクトの参照番号が変数 bufID に格納される。具体的な参照番号を知る必要は

[5] 既に説明したように、ウインドウサイズを変えるには関数 glutInitWindowSize() の実引数値を変えればよい

ない。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。上述したように、OpenGL ではこの参照番号をしばしば名前（name）と呼ぶが、実態は GLuint （＝ OpenGL で定義された符号なし整数＝通常の符号なし整数）であるから、このテキストでは参照番号という呼び方を用いる。

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「1 時点で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。そこで、

```
60      glBindBuffer(GL_ARRAY_BUFFER, bufID);
                                    // その参照番号を配列バッファとして束縛
```

上の 1 行は、参照番号 bufID のバッファオブジェクトについて、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している。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ーへの入力バッファであることを意味し、その種類のバッファを特に Vertex Buffer Object（VBO）と呼んでいる。

```
62      glBufferData(GL_ARRAY_BUFFER, sizeof(float)*2*NUM_POINTS,
                    pos, L_STATIC_DRAW);
                                    // 配列バッファに配列 pos のデータを転送
```

上の 1 行は、以下の処理をまとめて行う関数である。

1. 第 1 引数 GL_ARRAY_BUFFER 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している。直前の関数呼び出し glBindBuffer() において、参照番号 bufID が GL_ARRAY_BUFFER であると設定しているため、bufID がこの関数呼び出しの対象である（分かりにくい！）。

2. 第 2 引数 sizeof(float)*2*NUM_POINTS はバッファのサイズをバイトの単位で指定している。指定されたサイズのメモリ領域が GPU のメモリ内に確保される。

3. 第 3 引数 pos は、確保された GPU のメモリに CPU のメモリから転送するデータ（この場合、配列データ）の先頭アドレスを指定している。転送するデータのサイズは第 2 引数の値である。この第 3 引数に NULL を指定すると、データの転送は行わない。なお、CPU のメモリ上のデータを GPU のメモリへ転送する方法は他にもある [6] のだが、初学者には glBufferData() 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。

[6] 興味のある人は、glMapBuffer()、glUnmapBuffer()、glBufferSubData() などの関数を調べてみなさい。

4. 第 4 引数 GL_STATIC_DRAW は、確保する GPU のメモリがシェーダー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している。この宣言を間違えてもシェーダープログラムは（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_READ など 9 種類の引数が指定可能である。

次に、GPU に確保されたメモリ領域をシェーダープログラム（図 3.5）と結合するのが、以下である。

```
63      GLint p = glGetAttribLocation(program,"position");
                                          // 変数 position の位置 p の取得
```

上の 1 行は、第 1 引数に指定されたシェーダープログラム 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 "position" と同じものを探し、その位置（location）を変数 p へ代入する。位置の具体的な値を知る必要はない。もしその変数名が見つからない場合には関数の戻り値は負値となる（つまり実行時エラーである）。実際、図 3.5 のバーテックスシェーダープログラムには

```
03 attribute vec2 position;                    // 線分データの各座標値
```

という宣言があり、 "position" という名称を用いている。

次に、

```
64      glVertexAttribPointer(p, 2, GL_FLOAT, GL_FALSE, 0, 0);
                                                  // p の属性の設定
```

上の 1 行は、次の二つの処理を行う。

1. 第 1 引数 p で場所を指定したシェーダー実行可能プログラム中の arrribute 変数（直前の glGetAttribLocation() で指定した"position"という名前の変数）と現時点で設定対象となっている VBO（glBindBuffer() で指定した参照番号 bufID のバッファオブジェクト）を結合する。これによって、シェーダープログラムが実行されるときには、VBO の先頭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 2 は、対応する arrribute 変数が 2 個のコンポーネント（基本構成要素）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、GL_FALSE, 0, 0 でよい。

```
65      glEnableVertexAttribArray(p);                    // p の有効化
```

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

```
67      glLineWidth(10);                                 // 線幅の設定
```

上の 1 行は、線分を描く場合の線の太さを指定する。デフォルトでは 1.0 である。この関数呼び出しは本来、描画関数 display() に置くべきかもしれないが、今回の例題では一度設定した線分の太さを変えないため、initData() 内に置いた。

以上で、入力データの準備ができた。

描画処理

以下、描画関数 display() の処理について解説する。

```
72      glUseProgram(program);                   // プログラムを使用する宣言
```

上の 1 行は、参照番号 program のシェーダー実行可能プログラムをこれから実行することを宣言している。システムはこのプログラムを GPU 内にセットアップする。複数のシェーダー実行可能プログラムがある場合、プログラムを順次切り替えながら描画処理を進めていく。

```
73      glClear(GL_COLOR_BUFFER_BIT);  // 背景色によるウインドウのクリア
```

上の 1 行は、画像バッファを事前に設定した色でクリアする。実引数 GL_COLOR_BUFFER_BIT は画像のみをリセットすることを意味する。3D CG の場合には、デプス・バッファのリセットも必要であるが、ここでは用いない。

```
74      glDrawArrays(GL_LINES, 0, NUM_POINTS);
```

上の 1 行が描画の中核部分である。この 1 行は、直前の glUseProgram(program) で指定したシェーダー実行可能プログラムを起動する。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。

   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

   ここでは単純な線分を用いるため、GL_LINES を指定している。

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。

このプログラムの関数呼び出しでは、ひとつの線分（GL_POINTS）の描画に 2 頂点が必要である。その頂点を NUM_POINTS 個用いるから、線分の数は NUM_POINTS$/2 = 4/2 = 2$ である。

最後に、

```
75      glFlush();                                  // 描画処理の強制実行
```

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 glDrawArrays() 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。glDrawArrays() 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仕様になっている。実際にはそのような事態は複数台の 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、単体の PC で描画を行う場合には描画の遅延は起きない。しかし、描画を確実に行うために glFlush() を呼ぶことが OpenGL の流儀になっている。

main 関数の処理

main 関数の処理は既に 2.4 節で述べた通りである。

シェーダープログラムのコンパイル

上で説明を省略した関数 compileProgram() は図 3.4 のように用意する。以下、これをついて簡単に述べる。

まず、以下の 8 行

```
89     FILE* fp = fopen(file, "r");         // シェーダーファイルの open
90
91 #define MAX_SHADER_FILE_SIZE 10000
92     GLchar ftext[MAX_SHADER_FILE_SIZE];
                                        // ファイルの内容を格納する配列
93
94     unsigned long n = fread(ftext, 1, MAX_SHADER_FILE_SIZE, fp);
                                                        // 読み込み
95     ftext[n] = '\0';                              // 末尾にヌル文字を設定
96     fclose(fp);                                       // ファイルの close
```

は、ファイルの内容を文字配列 ftext に格納する。ただし、このプログラムは実装を端折っており、シェーダーファイルのサイズが 1 万文字（=MAX_SHADER_FILE_SIZE）を超える（通常のプログラムではまず超えないが）とプログラムがメモリ例外を起こす。

```
98     GLuint shader = glCreateShader(type);
                                  // type 型の空のシェーダープログラムの作成
```

上の 1 行は、第 1 引数の種類（GL_VERTEX_SHADER または GL_FRAGMENT_SHADER ）の、空のシェーダーオブジェクトプログラムを生成する。

```
99     const GLchar* ftext_handle = ftext;            // ハンドルの設定
```

上の 1 行は、シェーダーソースプログラムが格納されている配列の先頭アドレス ftext を一旦、変数 ftext_handle に格納する。アドレスを格納する変数を一般にハンドル（handle）と呼ぶ。

```
A0     glShaderSource(shader, 1, &ftext_handle, NULL);
                                        // ファイルの内容をハンドルで結合
```

上の 1 行は、ハンドル ftext_handle を介してソースファイルの文字列をシェーダーオブジェクトプログラムへ格納する。

```
A2     glCompileShader(shader);                                // コンパイル
```

上の 1 行は、シェーダーオブジェクトプログラムに格納されているシェーダーソースプログラムをコンパイルする。ソースプログラムに構文エラーがある場合には、ここでそのエ

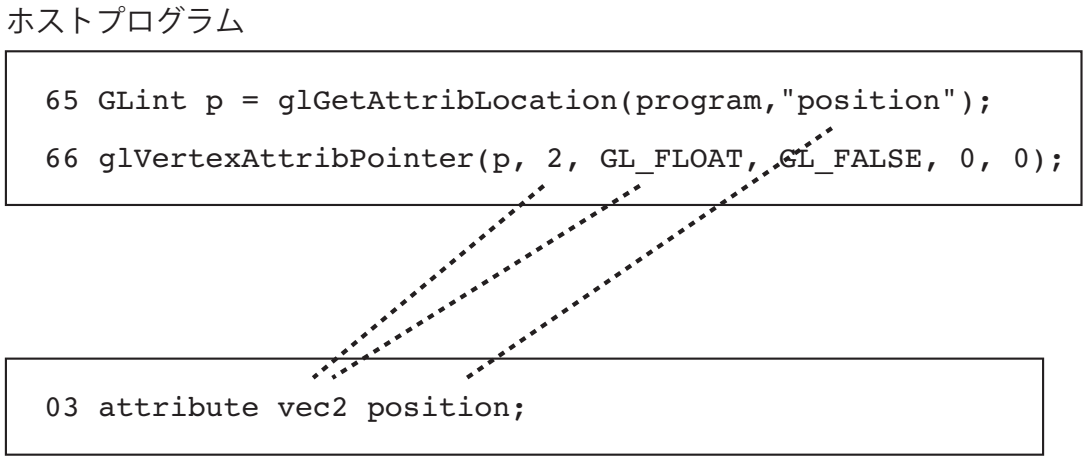

図 3.10: attribute 変数に関するホストプログラムとバーテックスシェーダープログラムの関係

ラーを検知し、プログラムの実行を強制中断すべきである。しかしここではとりあえず正常実行のプログラムの理解を優先するため、エラー検出/エラー処理は全て取り除いている。実際のプログラムにおいてエラー検出/エラー処理は必須であることは言うまでもない。エラー検出/エラー処理は 3.3 節で述べる。

3.1.2　バーテックスシェーダープログラム

　現時点の最新の OpenGL/GLSL の仕様はバージョン 4.6 であり、様々な新機能が盛り込まれている。しかし、この講義ではそれよりもはるかに以前のバージョン 1.2 を用いる。シェーダー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。

　図 3.5 が例題のバーテックスシェーダープログラムのソースである。主要部分の解説は以下の通りである。

```
01 #version 120
```

上の 1 行は、ここで用いる GLSL のバージョンが 1.2 であることを宣言している。

```
03 attribute vec2 position;                    // 線分データの各座標値
```

上の 1 行は、attribute 変数 `position` の宣言である。

　attribute 変数は、その値が関数 `initData()` で初期設定された配列バッファから供給されるように仕組まれた変数であり、キーワード attribute は大域変数にのみ修飾可能で

ある。この変数のデータは、関数 initData() で設定したバッファから自動的に供給される。ホストプログラムの以下の行

```
63      GLint p = glGetAttribLocation(program,"position");
                                        // 変数 position の位置 p の取得
```

が、ホストプログラムから供給先の attribute 変数 position の位置を求める関数呼び出しであった。データ型 vec2 はベクトル型と呼ばれ、float 型 2 個からなる。ホストプログラムの以下の行

```
64      glVertexAttribPointer(p, 2, GL_FLOAT, GL_FALSE, 0, 0);
                                                    // p の属性の設定
```

の第 2、第 3 引数がこれに対応する。position の中の各要素は、先頭から position.x、position.y で読み書き可能である。attribute 変数に関する、この辺の関係を図 3.10 に示す。attribute 変数の宣言がホストプログラムの設定と矛盾すると、プログラムの動作が正常に行われない。

```
07      gl_Position = vec4(position, 0.0, 1.0);         // 描画位置の設定
```

の一行は、頂点のウインドウ上の描画位置を設定する。左辺の vec4 型のシステム予約変数 gl_Position には、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

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

で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。なお、OpenGL では描画点が

$$-1.0 \leq X \leq 1.0, \quad -1.0 \leq Y \leq 1.0, \quad -1.0 \leq Z \leq 1.0, \quad w > 0$$

を満たすときに限り描画されると約束されている。描画位置はウインドウを $-1.0 \leq X \leq 1.0$、$-1.0 \leq Y \leq 1.0$ の矩形領域とするときの位置 (X, Y) である。Z の値は深さ値である。

上の式の右辺 vec4(position,0.0,1.0) について言えば

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

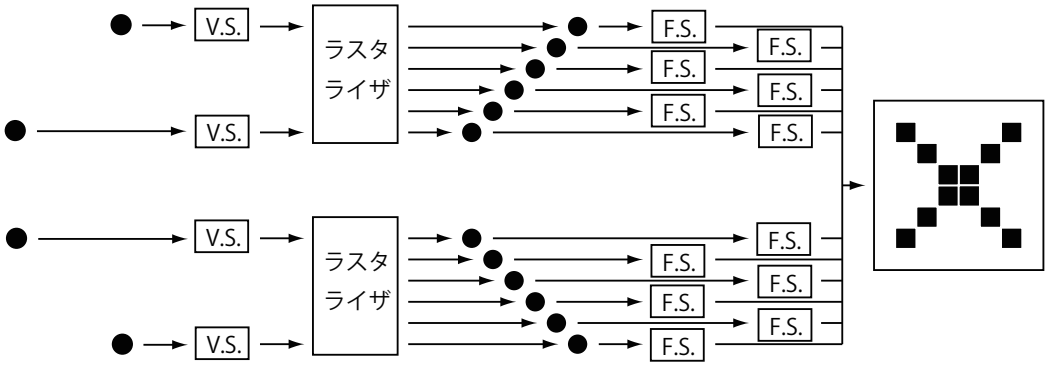

図 3.11: 線画を描く場合のシェーダーの動作例

を意味する（0.0, 1.0 が position.x, position.y の後ろに付く）。よって、通常の 3 次元空間の座標点は以下の通りである。

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

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ことになる。

3.1.3　フラグメントシェーダープログラム

図 3.6 が例題のフラゴメントシェーダープログラムのソースである。代入文：

```
05      gl_FragColor = vec4(1.0, 0.0, 0.0, 0.0);// 赤色 (1,0,0) の描画
```

の左辺の vec4 型の予約変数 gl_FragColor には、その頂点の RGB 輝度値とアルファ値を格納すると約束されている。ホストプログラムでアルファブレンド（透明度を指定した色の重ね合わせ）を有効にする指定[7]を行なっていないため、このプログラムではアルファ値は無視される。代入文の右辺の値は、RGB 輝度値が赤色 $(1.0, 0.0, 0.0)$ であるから、線分は赤一色に塗られる（図 3.1）。

3.1.4　シェーダープログラムの実行の様子

図 3.11 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。

[7]アルファブレンドを有効にするには、initSystem() で glEnable(GL_BLEND) を呼ぶ必要がある。

バーテックスシェーダーの動き この例題では、関数 `initData()` で 4 頂点の値を設定している。それらの各頂点がバーテックスシェーダープログラムの attribute 変数 `position` へ供給され、四つのバーテックスシェーダープログラムが**並列実行**される。バーテックスシェーダー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ーで求められた `gl_Position` の座標値は、入力バッファ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（`glDrawArrays()` 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する（実際にはひとつのラスタライザがパイプライン的な動作を行なっているだろうが、概念上は並列実行とみなされる）。

フラグメントシェーダーの動き 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ーが独立して並列実行される。

フラグメントシェーダーから `gl_FragColor` に出力された色情報は GPU のフレームバッファ集約されて、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 [8]。

[8] 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）。

3.2　簡単な線画：クラスの導入

前節で紹介したスタイルのプログラムは可読性がきわめて低い。そこでプログラムの一部をオブジェクト指向の手法でクラス化し、記述を整理する。処理内容は前節と同じであり、描画像も同じである。

新たに定義するクラスは以下の通りである。なおここではカプセル化を考慮しないため、`struct` キーワードで宣言する [9]。

`position2D` ... ２次元座標値を保持するための単純なオブジェクトのクラスである。メンバー関数（Java のメソッド）を持たないため、単なる構造体である。

`ArrayBuffer` ... attribute 変数に供給する配列データを保持するバッファオブジェクトを定義するクラスである。

`Shader` ... シェーダープログラムを統合管理するオブジェクトを定義するクラスである。

なお、この講義ではクラス間の階層構造は用いない。

[9] 本来はクラス定義を `class` キーワードで宣言すると、いちいち `public` 修飾するのが煩雑である。

```
01 #include <iostream>
02 using namespace std;
03 #include "stdio.h"
04 #include "stdlib.h"
05
06 #if defined(WIN32)
07 #  pragma comment(lib, "glew32.lib")
08 #  include "glew.h"
09 #  include "glut.h"
10 #  include "glext.h"
11 #elif defined(__APPLE__) || defined(MACOSX)
12 #  include <GLUT/glut.h>
13 #else
14 #  define GL_GLEXT_PROTOTYPES
15 #  include <GL/glut.h>
16 #endif
17
18 struct Position2D                  // 2次元座標値を保持するオブジェクト
19 {
20     float x;                                                  // x 成分
21     float y;                                                  // y 成分
22 };
23
24 struct ArrayBuffer {       // ひとつの配列バッファを保持するオブジェクト
25     GLuint bufID;                                    // バッファの参照番号
26     int size;                                          // バッファのサイズ
27
28     ArrayBuffer(float* data, int s, int n);         // コンストラクタ
29 };
30
31 struct Shader        // シェーダープログラムを管理・保持するオブジェクト
32 {
33     GLuint program;
34
35     Shader(const char* vsn, const char* fsn);      // コンストラクタ
36
37     void bindArrayBuffer(const char* vname, ArrayBuffer* ap);
                              // attribute 変数名と配列バッファの結合
38
39     void use();               // シェーダー実行可能プログラムの使用宣言
40     void run(GLenum mode, int n);                  // プログラムの実行
41
42     GLuint compileProgram(GLenum type, const GLchar *file);
                             // シェーダーソースプログラムのコンパイル
43     void buildProgram(const GLchar *vsn, const GLchar *fsn);
                                // シェーダー実行可能プログラムの構築
44 };
```

図 3.12: ×印を描画するヘッダープログラム `All.h`

```
01 #include "All.h"
02
03 ArrayBuffer::ArrayBuffer(float* data, int s, int n)
04 {
05     size = s;
06     glGenBuffers(1, &bufID);
07     glBindBuffer(GL_ARRAY_BUFFER, bufID);
08     glBufferData(GL_ARRAY_BUFFER, sizeof(float)*size*n,
                    data, GL_STATIC_DRAW);
09     glBindBuffer(GL_ARRAY_BUFFER, NULL);
                                    // bufID を配列バッファの設定から解放
10 }
```

図 3.13: ArrayBuffer クラスの実装プログラム Buffer.cpp

```
01 #include "All.h"
02
03 Shader::Shader(const char* vsn, const char* fsn)
04 {
05     buildProgram(vsn, fsn);
06 }
07
08 void Shader::bindArrayBuffer(const char* vname, ArrayBuffer* ap)
09 {
10     glBindBuffer(GL_ARRAY_BUFFER, ap->bufID);
11     GLint p = glGetAttribLocation(program, vname);
12     glVertexAttribPointer(p, ap->size, GL_FLOAT, GL_FALSE, 0, 0);
13     glEnableVertexAttribArray(p);
14 }
15
16 void Shader::use()
17 {
18     glUseProgram(program);
19 }
20
21 void Shader::run(GLenum mode, int n)
22 {
23     glDrawArrays(mode, 0, n);
24     glFlush();
25 }
```

図 3.14: Shader クラスの実装プログラム Shader.cpp（その 1、その 2 へ続く）

```
26 GLuint Shader::compileProgram(GLenum type, const GLchar *file)
27 {
28     FILE* fp = fopen(file, "r");
29
30 #define MAX_SHADER_FILE_SIZE 10000
31     GLchar ftext[MAX_SHADER_FILE_SIZE];
32
33     unsigned long n = fread(ftext, 1, MAX_SHADER_FILE_SIZE, fp);
34     ftext[n] = '\0';
35     fclose(fp);
36
37     GLuint shader = glCreateShader(type);
38     const GLchar* ftext_handler = (const GLchar*)&ftext;
39     glShaderSource(shader, 1, &ftext_handler, NULL);
40
41     glCompileShader(shader);
42     return shader;
43 }
44
45 void Shader::buildProgram(const GLchar *vsn, const GLchar *fsn)
46 {
47     GLuint vs = compileProgram(GL_VERTEX_SHADER, vsn);
48     GLuint fs = compileProgram(GL_FRAGMENT_SHADER, fsn);
49
50     program = glCreateProgram();
51     glAttachShader(program, vs);
52     glAttachShader(program, fs);
53
54     glLinkProgram(program);
55 }
```

図 3.15: Shader クラスの実装プログラム Shader.cpp（その 2）

```
01 #include "All.h"
02
03 Shader *sp;
04
05 void initSystem(int argc, char *argv[])
06 {
07     glutInit(&argc,argv);
08     glutInitDisplayMode(GLUT_RGB | GLUT_SINGLE);
09     glutInitWindowSize(512,512);
10     glutCreateWindow("Test Window");
11     glClearColor(0.5,0.5,0.5,1);
12
13 #if defined(WIN32)
14     glewInit();
15 #endif
16
17     sp = new Shader("shader.vert","shader.frag");
                                        // シェーダーオブジェクトの生成
18 }
19
20 const int NUM_POINTS = 4;
21
22 void initData()
23 {
24     Position2D pos[NUM_POINTS];
25
26     pos[0].x = -0.5; pos[0].y = -0.5;
27     pos[1].x = +0.5; pos[1].y = +0.5;
28     pos[2].x = +0.5; pos[2].y = -0.5;
29     pos[3].x = -0.5; pos[3].y = +0.5;
30     ArrayBuffer ab1((float*)pos, 2, NUM_POINTS);
                                      // 配列バッファオブジェクトの生成
31     sp->bindArrayBuffer("position", &ab1);
                                    // attribute 変数と配列バッファの結合
32     glLineWidth(10.0);
33 }
34
```

図 3.16: ×印を描画するホストプログラム main.cpp（その 1、その 2 へ続く）

```
35 void display(void)
36 {
37     sp->use();                          // シェーダプログラムの使用の宣言
38     glClear(GL_COLOR_BUFFER_BIT);
39     sp->run(GL_LINES, NUM_POINTS);      // シェーダープログラムの実行
40 }
41
42 int main(int argc, char *argv[])
43 {
44     initSystem(argc,argv);
45     initData();
46
47     glutDisplayFunc(display);
48     glutMainLoop();
49     return 0;
50 }
```

図 3.17: ×印を描画するホストプログラム main.cpp（その 2）

14 ページの図 3.5 と同じ

図 3.18: ×印を描画するバーテックスシェーダープログラム shader.vert

14 ページの図 3.6 と同じ

図 3.19: ×印を描画するフラグメントプログラム shader.frag

3.2.1　2 次元座標構造体：Position2D

図 3.12 に宣言される構造体 Position2D は 2 次元座標値を保持するために使用する。座標値に関する演算などは行わないため、メンバー関数は用意しない。

3.2.2　入力データ配列クラス：ArrayBuffer

同じく図 3.12 に宣言されるクラス ArrayBuffer は、GPU へ投入する入力データ、頂点データの配列をオブジェクト化して管理する。このクラスは図 3.2〜図 3.4 の複雑な処理をこのクラスの中にまとめて隠すことが目的である。

なお、この講義ではデータのカプセル化は行わないため、クラスの定義には class キーワードは用いず、struct を用いる。よってクラスのメンバーは全て外部に無条件で公開される。もし、class を用いるならば、クラス宣言の冒頭は以下のようにせねばならない。

```
class ArrayBuffer {
public:
    ... 以下、struct の場合と同じ
```

このクラスはコンストラクタのみを含むが、その仕様は以下の通りとする。

ArrayBuffer(float* data, int s, int n) ... 第 1 引数には、入力データが格納されている float 型配列を指定する。第 2 引数には、バーテックスシェーダーへの入力である attribute 変数に含まれる float 型データ [10] の個数を指定する。第 3 引数には、頂点の総数を指定する。

コンストラクタの実装は図 3.13 の通りである。メンバー変数 bufID、size は内部でのみ使用する変数である [11]。このクラスにはコンストラクタのみがあり、必要な全ての作業はオブジェクトを作る作業の中で行う。

3.2.3　シェーダークラス：Shader

同じく図 3.12 に宣言されているクラス Shader はシェーダープログラムをオブジェクト化して管理するものである。このクラスも図 3.2〜図 3.4 の複雑な処理をこのクラスの中にまとめて隠すことが目的である。

メンバー関数の内容は以下の通りである。

[10] attribute 変数は int 型を含んでもよいが、この講義ではそのような例は出てこない。

[11] 内部での使用のみならば、データを private 化して隠蔽するのが C++の正しいクラス定義作法である。が、ここではその点には関知しない方針である。

`Shader(const char* vsn, const char* fsn)` ... コンストラクタである。第 1、第 2 引数にバーテックスシェーダー、フラグメントシェーダーのソースファイル名を文字列で指定する。

`void use()` ... このシェーダー実行可能プログラムをこれ以降、使うことを宣言する。

`void bindArrayBuffer(const char* vname, ArrayBuffer* ap)` ... 第 1 引数に、このシェーダーのバーテックスシェーダーの中で宣言されている attribute 変数名を文字列で指定する。第 2 引数に、その attribute 変数へデータを供給する ArrayBuffer オブジェクトのアドレスを指定する。

`void run(GLenum mode, int n)` ... 描画を行う関数である。第 1 引数に描画モード（GL_LINES など）を指定する。第 2 引数に頂点数を指定する。

このクラスのメンバー関数の実装は図 3.14、図 3.15 の通りである。メンバー変数 program は内部でのみ使用する変数である。メンバー関数 compileProgram()、buildProgram() も内部でのみ使用する関数である。

3.2.4　初期関数、描画関数の変更

上記のクラス、オブジェクトを用いると、関数 initSystem()、initData()、display() はそれぞれ図 3.16、図 3.17 の通りに簡単化される。

3.2.5　バーテックスシェーダープログラム

本節はホストプログラムをクラス化、オブジェクト化する目的であるから、前節から変更しない。

3.2.6　フラグメントシェーダープログラム

やはり前節から変更しない。

3.2.7　シェーダープログラムの実行の様子

プログラムは、前節の図 3.11 と同様に実行される。

3.3　エラー処理

前節までは OpenGL/GLSL のプログラムを紹介するために、あえてエラー処理は省いてきた。しかし実際にはエラー処理のないプログラムはプログラムのデバッグを非常に困難にする。

図 3.14 の `Shader::compileProgram()` にコンパイルエラーが起きたことを検知する処理を追加したプログラムは、図 3.20 のようになる。コンパイルを行なった直後に

1. ステータスをチェックし

```
19      glGetShaderiv(shader, GL_COMPILE_STATUS, &status);
```

2. 問題があれば、次にエラーログのサイズを取得し、

```
24          glGetShaderiv(shader, GL_INFO_LOG_LENGTH, &infologLength);
```

3. エラーログを取得し、

```
34              glGetShaderInfoLog(shader, infologLength, &charsWritten,
35                                 infoLog);       // エラーログの取得
```

4. エラーログを表示し、

```
36              cerr << "Shader InfoLog:" << endl << infoLog << endl;
```

5. プログラムを強制中断する。

```
39          exit(1);                                        // 実行を強制中断
```

図 3.14 の `Shader::buildProgram()` にリンクエラーが起きたことを検知する処理を追加したプログラムは、図 3.21 のようになる。追加の内容は `Shader::compileProgram()` と同様である。

ホストプログラム中に書いた attribute 変数名とバーテックスシェーダーソースプログラム中に宣言した attribute 変数名が異なるエラーはありがちである。この場合には `Shader::bindArrayBuffer()` 内で求める attribute 変数の位置 `p` が負数となるエラーが起きる。それを検知するために図 3.14 の `Shader::bindArrayBuffer()` を図 3.22 のように変更する。

```
01 #define MAX_SHADER_FILE_SIZE 10000
02
03 GLuint Shader::compileProgram(GLenum type, const GLchar *file)
04 {
05     FILE* fp = fopen(file, "r");
06
07     GLchar ftext[MAX_SHADER_FILE_SIZE];
08     unsigned long n = fread(ftext, 1, MAX_SHADER_FILE_SIZE, fp);
09     ftext[n] = '\0';
10     fclose(fp);
11
12     GLuint shader = glCreateShader(type);
13     const GLchar* ftext_handler = (const GLchar*)&ftext;
14     glShaderSource(shader, 1, &ftext_handler, NULL);
15
16     glCompileShader(shader);
17
18     GLint status;
19     glGetShaderiv(shader, GL_COMPILE_STATUS, &status);
                                          // コンパイル結果のチェック
20     if (!status)                          // もしエラーがあるならば
21     {
22         cerr << "shader compile error: " << file << endl;
23         GLsizei infologLength;
24         glGetShaderiv(shader, GL_INFO_LOG_LENGTH, &infologLength);
                   // システムが一時的に保持するエラーログの長さを求める
25         GLchar * infoLog = new GLchar[infologLength];
                                       // ログの長さに応じた配列を生成
26         if (infoLog == NULL)
27         {                           // もし配列の生成に失敗したならば
28             cerr << "ERROR: Could not allocate InfoLog buffer\n"
29                  << endl;
30         }
31         else             // 配列の生成に成功したならば（通常はこちら）
32         {
33             int charsWritten  = 0;
34             glGetShaderInfoLog(shader, infologLength, &charsWritten,
35                                infoLog);         // エラーログの取得
36             cerr << "Shader InfoLog:" << endl << infoLog << endl;
                                                  // エラーログの表示
37             delete infoLog;
38         }
39         exit(1);                                  // 実行を強制中断
40     }
41
42     return shader;
43 }
```

図 3.20: コンパイルエラーをチェックし、エラーを発見したならばエラー出力を行い、プログラムを強制終了する Shader::compileProgram()

```
01 void Shader::buildProgram(const GLchar *vsn, const GLchar *fsn)
02 {
03     GLuint vs = compileProgram(GL_VERTEX_SHADER, vsn);
04     GLuint fs = compileProgram(GL_FRAGMENT_SHADER, fsn);
05
06     program = glCreateProgram();
07     glAttachShader(program, vs);
08     glAttachShader(program, fs);
09
10     glLinkProgram(program);
11
12     GLint status;
13     glGetProgramiv(program, GL_LINK_STATUS, &status);
                                                  // リンク結果のチェック
15     if (!status)                               // もしエラーがあるならば
16     {
17         cerr << "program link  error: " << endl;
18         GLsizei infologLength;
19         glGetProgramiv(program, GL_INFO_LOG_LENGTH, &infologLength);
                    // システムが一時的に保持するエラーログの長さを求める
20         GLchar * infoLog = new GLchar[infologLength];
                                        // ログの長さに応じた配列を生成
21         if (infoLog == NULL)
22         {                             // もし配列の生成に失敗したならば
23             cerr << "ERROR: Could not allocate InfoLog buffer\n"
24                  << endl;
25         }else{              // 配列の生成に成功したならば（通常はこちら）

26             int charsWritten  = 0;
27             glGetProgramInfoLog(program, infologLength,
28                                 &charsWritten, infoLog);
                                                    // エラーログの取得
29             cerr << "Program InfoLog:" << endl << infoLog << endl;
                                                    // エラーログの表示
30             delete infoLog;
31         }
32         exit(1);                                  // 実行を強制中断
33     }
34 }
```

図 3.21: リンクエラーをチェックし、エラー出力を行う Shader::buildProgram()

```
01 void Shader::bindArrayBuffer(const char* vname, ArrayBuffer* ap)
02 {
03     glBindBuffer(GL_ARRAY_BUFFER, ap->bufID);
04     GLint p = glGetAttribLocation(program,vname);
05     if(p < 0) {                          // もし位置が負値ならばエラー
06         cerr << "attribute name error: " << vname << endl;
07         exit(1);                                    // 実行を強制中断
08     }
09     glVertexAttribPointer(p, ap->size, GL_FLOAT, GL_FALSE, 0, 0);
10     glEnableVertexAttribArray(p);
11}
```

図　3.22:　attribute 変数名に関するエラーをチェックし、エラー出力を行う Shader::bindArrayBuffer()

第4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。

4.1　輝度の線形補間

バーテックスシェーダー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ダープログラムに入力するインターフェースに相当する。前章の例では attribute 変数として `position` のみを用い、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、頂点の輝度値を追加し、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ーの出力を線形補間してフラグメントシェーダーの入力とする仕組み（varying 変数）を新たに導入する。

この節の画像の出力例は図 4.1 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

$$B = \frac{|p_2 - p|}{|p_2 - p_1|} B_1 + \frac{|p - p_1|}{|p_2 - p_1|} B_2 \tag{4.1}$$

と求めることをいう。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。

図 4.1: 画像例（その 2、輝度値の線形補間）

31 ページの図 3.12 と同じ

図 4.2: 各種クラスを定義するヘッダープログラム `All.h`

32 ページの図 3.13 と同じ

図 4.3: `ArrayBuffer` クラスの実装プログラム `Buffer.cpp`

32 ページの図 3.14、33 ページの図 3.15 と同じ。ただし 3.3 節のエラー処理を追加すべき

図 4.4: `Shader` クラスの実装プログラム `Shader.cpp`

```
01 #include "All.h"
02
03 Shader *sp;
04
05 void initSystem(int argc, char *argv[])
06 {
07     glutInit(&argc,argv);
08     glutInitDisplayMode(GLUT_RGB | GLUT_SINGLE);
09     glutInitWindowSize(512,512);
10     glutCreateWindow("Test Window");
11     glClearColor(0.5,0.5,0.5,1);
12
13 #if defined(WIN32)
14     glewInit();
15 #endif
16
17     sp = new Shader("shader.vert","shader.frag");
18 }
19
20 const int NUM_POINTS = 4;
21
22 void initData()
23 {
24     Position2D pos[NUM_POINTS];
25
26     pos[0].x = -0.5; pos[0].y = -0.5;      // 0番目の頂点の座標位置
27     pos[1].x = +0.5; pos[1].y = +0.5;      // 1番目の頂点の座標位置
28     pos[2].x = +0.5; pos[2].y = -0.5;      // 2番目の頂点の座標位置
29     pos[3].x = -0.5; pos[3].y = +0.5;      // 3番目の頂点の座標位置
30
31     ArrayBuffer ab((float*)pos,2,NUM_POINTS);
32     sp->bindArrayBuffer("position",&ab);
33
34     float bri[NUM_POINTS];             // 4頂点の輝度値を格納する配列
35     bri[0] = 0.0;                          // 0番目の頂点の輝度値
36     bri[1] = 1.0;                          // 1番目の頂点の輝度値
37     bri[2] = 0.0;                          // 2番目の頂点の輝度値
38     bri[3] = 1.0;                          // 3番目の頂点の輝度値
39
40     ArrayBuffer ab2((float*)bri,1,NUM_POINTS); // 配列バッファを生成
41     sp->bindArrayBuffer("in_brightness",&ab2);
                                  // attribute変数と配列バッファを結合
42
43     glLineWidth(10.0);
44 }
45
```

図 4.5: 輝度値が滑らかに変化する×印を描画するホストプログラム main.cpp（その 1、その 2 へ続く）

```
46 void display(void)
47 {
48     sp->use();
49     glClear(GL_COLOR_BUFFER_BIT);
50     sp->run(GL_LINES, NUM_POINTS);
51 }
52
53 int main(int argc, char *argv[])
54 {
55     initSystem(argc,argv);
56     initData();
57
58     glutDisplayFunc(display);
59     glutMainLoop();
60     return 0;
61 }
```

図 4.6: 輝度値が滑らかに変化する×印を描画するホストプログラム main.cpp（その 2）

```
01 #version 120
02
03 attribute vec2 position;                // 頂点の座標位置が供給される
04 attribute float in_brightness;           // 頂点の輝度値が供給される
05
06 varying float out_brightness; // 頂点の輝度値をラスタライザへ渡す変数
07
08 void main(void)
09 {
10     gl_Position = vec4(position, 0.0, 1.0);        // 描画位置の設
定
11     out_brightness = in_brightness;                  // 輝度値の設
定
12 }
```

図 4.7: 輝度値が滑らかに変化する×印を描画するバーテックスシェーダープログラム shader.vert

```
01 #version 120
02
03 varying float out_brightness;
               // 画素の輝度値がラスタライザで線形補間されて供給される
04
05 void main(void)
06 {
07     gl_FragColor = vec4(out_brightness, 0.0, 0.0, 0.0);
                          // out_brightness の値を赤色の輝度値として利用
08 }
```

図 4.8: 輝度値が滑らかに変化する×印を描画するフラグメントシェーダープログラム shader.frag

4.1.1　ホストプログラム

ホストプログラムの変更は図 4.5 の関数 initData() のみである。新たに配列 bri を用意し、そこに 0.0（最低輝度＝真っ暗)、1.0（最大輝度）を格納している。

```
34      float bri[NUM_POINTS];              // 4 頂点の輝度値を格納する配列
35      bri[0] = 0.0;                             // 0 番目の頂点の輝度値
36      bri[1] = 1.0;                             // 1 番目の頂点の輝度値
37      bri[2] = 0.0;                             // 2 番目の頂点の輝度値
38      bri[3] = 1.0;                             // 3 番目の頂点の輝度値
```

なお、OpenGL/GLSL の輝度値は 0〜1 の範囲である。0〜255 ではないことに注意する。

そして、以下のように、その配列について ArrayBuffer オブジェクトを作り、それをシェーダーと結合する。“in_brightness”は、この配列に対応するバーテックスシェーダープログラムの attribute 変数の名前である。

```
40      ArrayBuffer ab2((float*)bri,1,NUM_POINTS); // 配列バッファを生成
41      sp->bindArrayBuffer("in_brightness",&ab2);
```

4.1.2　バーテックスシェーダープログラム

図 4.7 が変更後のバーテックスシェーダープログラムである。主な変更点は以下の箇所である。

```
04 attribute float in_brightness;           // 頂点の輝度値が供給される
...
06 varying float out_brightness; // 頂点の輝度値をラスタライザへ渡す変数
...
11      out_brightness = in_brightness;                  // 輝度値の設定
```

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ー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ー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ー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。

図 4.7 の main 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、たとえば以下では輝度値が反転する。

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

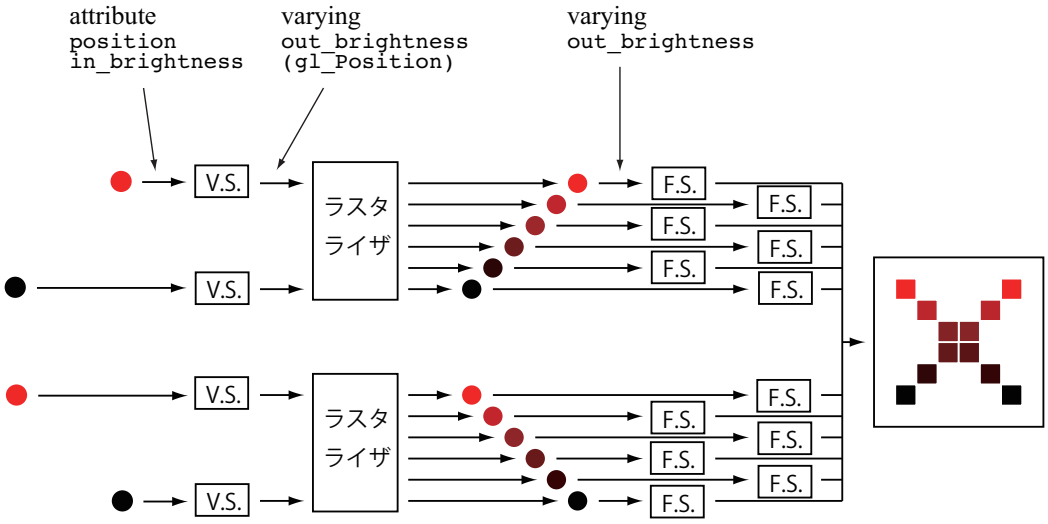

図 4.9: 輝度の線形補間の動作例

4.1.3　フラグメントシェーダープログラム

図 4.8 が変更後のフラグメントシェーダープログラムである。

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

```
03 varying float out_brightness;
```

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ー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ープログラムの varying 変数へ格納される。

バーテックスシェーダープログラムの `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ー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。

4.1.4　実行結果

図 4.9 に GPU の実行の様子を示す。

1　バーテックスシェーダー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。
2 この例では輝度値はそのままラスタライザに渡される。渡された輝度値は線分の両端点の
3 属性値と見なされ、線分の各画素について、その画素が線分内の位置に関する線形補間が
4 なされる。補間された値はフラグメントシェーダープログラムに入力される。それの値を
5 元に RGB カラーを作れば、図 4.1 のような画像が生成される。

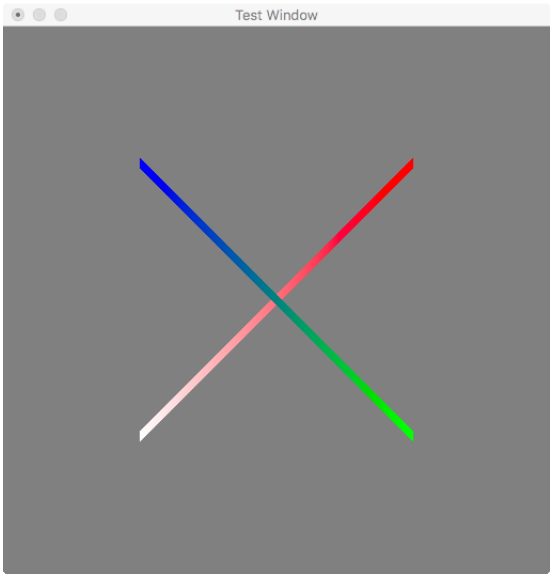

図 4.10: 画像例（その 3、RGB カラーの線形補間）

4.2　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 へ渡したが、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 4.10 である。線分の途中では両端点の 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

$$\vec{C} = \frac{|p_2 - p|}{|p_2 - p_1|}\vec{C}_1 + \frac{|p - p_1|}{|p_2 - p_1|}\vec{C}_2 \tag{4.2}$$

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。

```
01 #include <iostream>
02 using namespace std;
03 #include "stdio.h"
04 #include "stdlib.h"
05
06 #if defined(WIN32)
07 #  pragma comment(lib, "glew32.lib")
08 #  include "glew.h"
09 #  include "glut.h"
10 #  include "glext.h"
11 #elif defined(__APPLE__) || defined(MACOSX)
12 #  include <GLUT/glut.h>
13 #else
14 #  define GL_GLEXT_PROTOTYPES
15 #  include <GL/glut.h>
16 #endif
17
18 struct Position2D {
19     float x;
20     float y;
21 };
22
23 struct RGB {                                  // RGB カラーのための構造体
24     float r;                                                  // 赤の輝度値
25     float g;                                                  // 緑の輝度値
26     float b;                                                  // 青の輝度値
27 };
28
29 struct ArrayBuffer
30 {
31     GLuint    bufID;
32     int        size;
33
34    ArrayBuffer(float* data, int s, int n);
35 };
36
37 struct Shader
38 {
39     GLuint        program;
40
41     Shader(const char* vsn, const char* fsn);
42    void use();
43
44     void bindArrayBuffer(const char* vname, ArrayBuffer* ap);
45     void run(GLenum mode, int n);
46
47     GLuint compileProgram(GLenum type, const GLchar *file);
48     void buildProgram(const GLchar *vsfile, const GLchar *fsfile);
49 };
```

図 4.11: 各種クラスを定義するヘッダープログラム All.h

32 ページの図 3.13 と同じ

図 4.12: ArrayBuffer クラスの実装プログラム Buffer.cpp

32 ページの図 3.14、33 ページの図 3.15 と同じ。ただし 3.3 節のエラー処理を追加すべき

図 4.13: Shader クラスの実装プログラム Shader.cpp

```
01 #include "All.h"
02
03 Shader *sp;
04
05 void initSystem(int argc, char *argv[])
06 {
07     glutInit(&argc,argv);
08     glutInitDisplayMode(GLUT_RGB | GLUT_SINGLE);
09     glutInitWindowSize(512,512);
10     glutCreateWindow("Test Window");
11     glClearColor(0.5,0.5,0.5,1);
12
13 #if defined(WIN32)
14     glewInit();
15 #endif
16
17     sp = new Shader("shader.vert","shader.frag");
18 }
19
```

図 4.14: RGB カラーが滑らかに変化する×印を描画するホストプログラム main.cpp（その 1、その 2 へ続く）

```
20 const int NUM_POINTS = 4;
21
22 void initData()
23 {
24     Position2D pos[NUM_POINTS];
25
26     pos[0].x = -0.5; pos[0].y = -0.5;
27     pos[1].x = +0.5; pos[1].y = +0.5;
28     pos[2].x = +0.5; pos[2].y = -0.5;
29     pos[3].x = -0.5; pos[3].y = +0.5;
30
31     ArrayBuffer ab((float*)pos,2,NUM_POINTS);
32     sp->bindArrayBuffer("position",&ab);
33
34     RGB rgb[NUM_POINTS];
35     rgb[0].r = 1.0; rgb[0].g = 1.0; rgb[0].b = 1.0;          // 白
36     rgb[1].r = 1.0; rgb[1].g = 0.0; rgb[1].b = 0.0;          // 赤
37     rgb[2].r = 0.0; rgb[2].g = 1.0; rgb[2].b = 0.0;          // 緑
38     rgb[3].r = 0.0; rgb[3].g = 0.0; rgb[3].b = 1.0;          // 青
39
40     ArrayBuffer ab2((float*)rgb,3,NUM_POINTS); // 配列バッファの生成
41     sp->bindArrayBuffer("in_color",&ab2);
                                  // attribute 変数と配列バッファを結合
42
43     glLineWidth(10.0);
44 }
45
46 void display(void)
47 {
48     sp->use();
49     glClear(GL_COLOR_BUFFER_BIT);
50     sp->run(GL_LINES, NUM_POINTS);
51 }
52
53 int main(int argc, char *argv[])
54 {
55     initSystem(argc,argv);
56     initData();
57
58     glutDisplayFunc(display);
59     glutMainLoop();
60     return 0;
61 }
```

図 4.15: RGB カラーが滑らかに変化する×印を描画するホストプログラム main.cpp（その 2）

```
01 #version 120
02
03 attribute vec2 position;
04 attribute vec3 in_color;              // 頂点の RGB カラーが供給される
05
06 varying vec3 out_color;    // 頂点の RGB カラーをラスタライザへ渡す変数
07
08 void main(void)
09 {
10     gl_Position = vec4(position, 0.0, 1.0);
11     out_color = in_color;                          //RGB カラーの設定
12 }
```

図 4.16: RGB カラーが滑らかに変化する×印を描画するバーテックスシェーダープログラム shader.vert

```
01 #version 120
02
03 varying vec3 out_color;
              // 画素の RGB カラーがラスタライザで線形補間されて供給される
04
05 void main(void)
06 {
07     gl_FragColor = vec4(out_color, 0.0);
                               // out_color の値を RGB カラーとして利用
08 }
```

図 4.17: RGB カラーが滑らかに変化する×印を描画するフラグメントシェーダープログラム shader.frag

4.2.1　ホストプログラム

まず、RGB カラー値を保持する構造体を新たに図 4.11 の以下のように定義しておく。これは必須ではないが、プログラムの可読性向上には有効である。

```
23 struct RGB {                                    // RGB カラーのための構造体
24     float r;                                        // 赤の輝度値
25     float g;                                        // 緑の輝度値
26     float b;                                        // 青の輝度値
27 };
```

図 4.15 の関数 initData() には配列 rgb 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てみた。そして、その配列について ArrayBuffer オブジェクトを作り、それをシェーダーと結合している。“in_color”は、この配列に対応するバーテックスシェーダープログラムの attribute 変数の名前である。この辺は前節と同様である。

4.2.2　バーテックスシェーダープログラム

図 4.16 が変更後のバーテックスシェーダープログラムである。

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。さらに バーテックスシェーダー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ー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が、ただしデータ型は vec3、すなわち float 型が 3 個並ぶデータ型である。この vec3 型とホストプログラムの RGB 型は完全に一致するデータ型であることを注意する。

4.2.3　フラグメントシェーダープログラム

図 4.17 が変更後のフラグメントシェーダープログラムである。

プログラムにはバーテックスシェーダープログラムと同じ 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している。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（透明度） 0.0 を追加する。ただし、プログラムではアルファ値は利用しない設定になっているため、0.0 は単にサイズを合わせるためにのみ追加する。

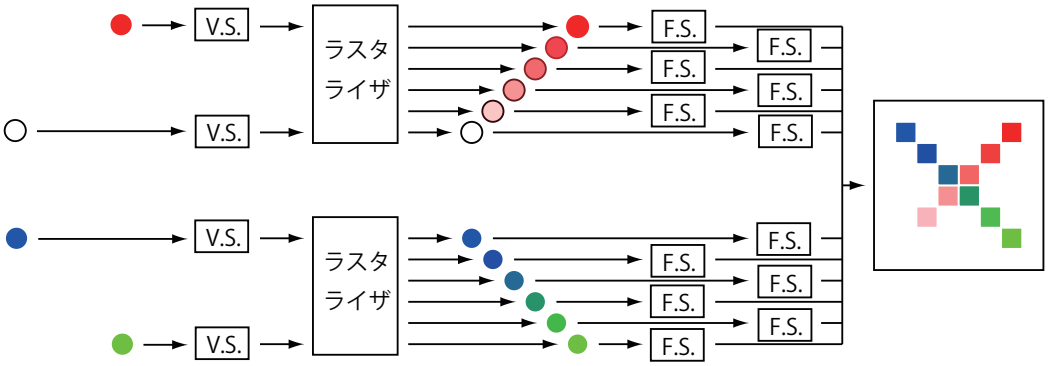

図 4.18: 輝度の線形補間の動作例

4.2.4　実行結果

図 4.18 に示すように、バーテックスシェーダー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ープログラムに入力される。

第5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とは、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描画技術である。素早く動く小さなキャラクターを背景の静止画の上にオーバーラップさせることで当時のコンピュータの演算・描画能力をカモフラージュする方法である。オーバーラップはメモリ上で行うのではなく、映像信号を出力する直前で信号合成によって行うため、コンピュータへの負荷は無い。“sprite”は妖精、小鬼を意味し、コンピュータ画面の上を激しく動き回るキャラクタという意味合いがある。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は行なっておらず、GPU のフレームメモリ上にスプライトの画像を直接描画している。現在の PC の演算・描画能力をもってすれば、その方法でも描画は十分に高速に可能である。

5.1　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 次元座標と線分の太さの情報が必要であった。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 5.1 の画像を生成するプログラムを紹介する。

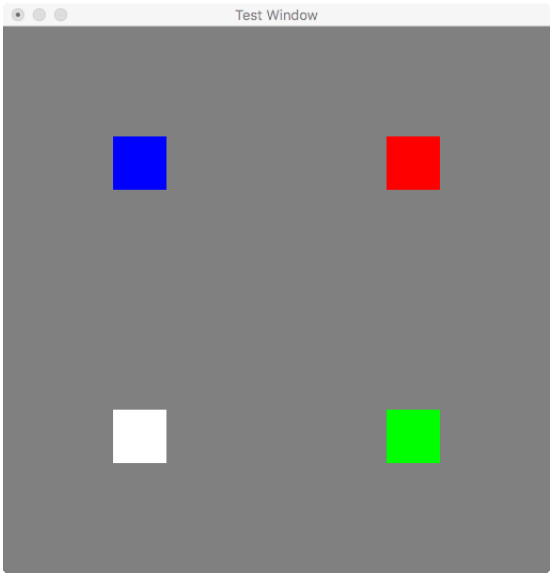

図 5.1: 画像例（その 4、ポイントスプライト）

51 ページの図 4.11 と同じ

図 5.2: 各種クラスを定義するヘッダープログラム `All.h`

32 ページの図 3.13 と同じ

図 5.3: `ArrayBuffer` クラスの実装プログラム `Buffer.cpp`

32 ページの図 3.14、33 ページの図 3.15 と同じ。ただし 3.3 節のエラー処理を追加すべき

図 5.4: `Shader` クラスの実装プログラム `Shader.cpp`

```
01 #include "All.h"
02
03 Shader *sp;
04
05 void initSystem(int argc, char *argv[])
06 {
07     glutInit(&argc,argv);
08     glutInitDisplayMode(GLUT_RGB | GLUT_SINGLE);
09     glutInitWindowSize(512,512);
10     glutCreateWindow("Test Window");
11     glClearColor(0.5,0.5,0.5,1);
12
13 #if defined(WIN32)
14     glewInit();
15 #endif
16
17     glEnable(GL_POINT_SPRITE); // ポイントスプライトを用いることを宣言
18     glEnable(GL_VERTEX_PROGRAM_POINT_SIZE);
   // ポイントスプライトの大きさをバーテックスシェーダーで指定することを宣言
19
20     sp = new Shader("shader.vert","shader.frag");
21 }
22
23 const int NUM_POINTS = 4;
24
25 void initData()
26 {
27     Position2D pos[NUM_POINTS];
28
29     pos[0].x = -0.5; pos[0].y = -0.5;
30     pos[1].x = +0.5; pos[1].y = +0.5;
31     pos[2].x = +0.5; pos[2].y = -0.5;
32     pos[3].x = -0.5; pos[3].y = +0.5;
33
34     ArrayBuffer ab((float*)pos,2,NUM_POINTS);
35     sp->bindArrayBuffer("position",&ab);
36
37     RGB rgb[NUM_POINTS];
38     rgb[0].r = 1.0; rgb[0].g = 1.0; rgb[0].b = 1.0;
39     rgb[1].r = 1.0; rgb[1].g = 0.0; rgb[1].b = 0.0;
40     rgb[2].r = 0.0; rgb[2].g = 1.0; rgb[2].b = 0.0;
41     rgb[3].r = 0.0; rgb[3].g = 0.0; rgb[3].b = 1.0;
42
43     ArrayBuffer ab2((float*)rgb,3,NUM_POINTS);
44     sp->bindArrayBuffer("in_color",&ab2);
45
46     glLineWidth(10.0);
47 }
48
```

図 5.5: 正方形のポイントスプライトを描画するホストプログラム main.cpp（その 1、その 2 へ続く）

```
49 void display(void)
50 {
51     sp->use();
52     glClear(GL_COLOR_BUFFER_BIT);
53     sp->run(GL_POINTS,NUM_POINTS);       // ポイントスプライトを描画
54 }
55
56 int main(int argc, char *argv[])
57 {
58     initSystem(argc,argv);
59     initData();
60
61     glutDisplayFunc(display);
62     glutMainLoop();
63     return 0;
64 }
```

図 5.6: 正方形のポイントスプライトを描画するホストプログラム main.cpp（その 2）

```
01 #version 120
02
03 attribute vec2 position;
04 attribute vec3 in_color;
05
06 varying vec3 out_color;
07
08 void main(void)
09 {
10     gl_Position = vec4(position,0.0,1.0);
11     out_color = in_color;
12     gl_PointSize = 50;           // ポイントスプライトの大きさを設定
13 }
```

図 5.7: 正方形のポイントスプライトを描画するバーテックスシェーダープログラム shader.vert

54 ページの図 4.17 と同じ

図 5.8: 正方形のポイントスプライトを描画するフラグメントシェーダープログラム shader.frag

5.1.1　ホストプログラム

図 5.5、図 5.6 が変更後のホストプログラムである。

ポイントスプライトを用いるには、まずそのことを関数呼び出し：

```
17      glEnable(GL_POINT_SPRITE); // ポイントスプライトを用いることを宣言
```

で宣言せねばならない。また、バーテックスシェーダープログラムでスプライトの大きさを表す予約変数 gl_PointSize に正数値を代入せねばならないから、その gl_PointSize を利用するために関数呼び出し：

```
18      glEnable(GL_VERTEX_PROGRAM_POINT_SIZE);
```

を事前に行う必要がある。この宣言をしない場合、バーテックスシェーダープログラムでの gl_PointSize への代入は無視され（コンパイルエラーは生じない）、値 1 が仮定される。

関数 initData() は頂点情報を定義するだけであるから、前節と全く同じでもよい。

関数 display() では、以下の通り、

```
53      sp->run(GL_POINTS,NUM_POINTS);          // ポイントスプライトを描画
```

ポイントスプライトの描画モード GL_POINTS でシェーダープログラムを起動する。

5.1.2　バーテックスシェーダープログラム

図 5.7 が変更後のバーテックスシェーダープログラムである。

gl_PointSize への代入文

```
12      gl_PointSize = 50;              // ポイントスプライトの大きさを設定
```

を追加している。この大きさの単位は画素である。gl_PointSize の最大有効値 [1] は処理系依存だが、ネット検索等の情報では 500 程度が多いようである。それよりも大きな値を代入しても最大値が有効になる。

5.1.3　フラグメントシェーダープログラム

フラグメントシェーダープログラムは前章から変更しない。

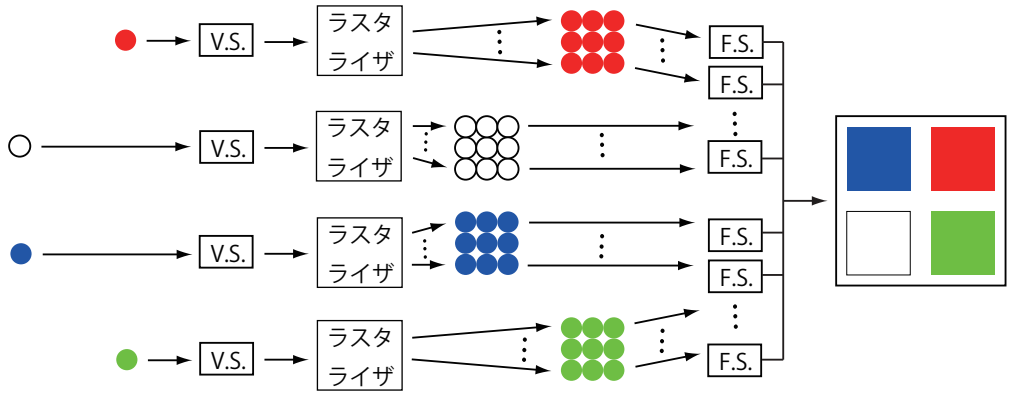

図 5.9: ポイントスプライトの動作例

5.1.4　実行結果

図 5.9 に実行の様子を図示する。バーテックスシェーダー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ーへ渡す。

[1]定数 GL_POINT_SIZE_MAX として定義されている。

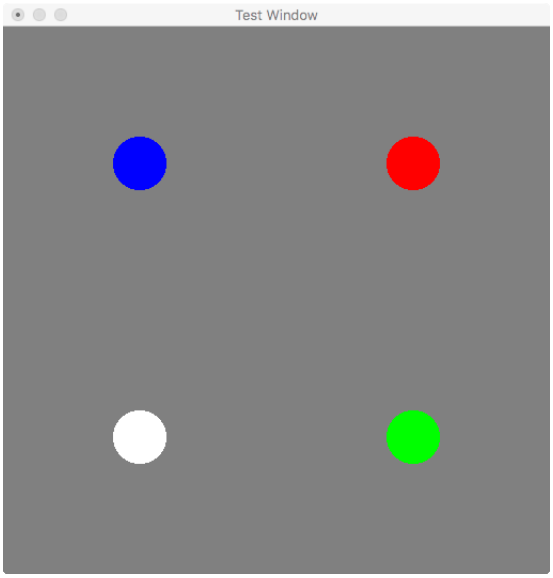

図 5.10: 画像例（その 5、ポイントスプライトが丸い場合）

5.2　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 5.10 のような円形を描画する方法を紹介する。きわめて簡単な手順である。

51 ページの図 4.11 と同じ

図 5.11: 各種クラスを定義するヘッダープログラム All.h

32 ページの図 3.13 と同じ

図 5.12: ArrayBuffer クラスの実装プログラム Buffer.cpp

32 ページの図 3.14、33 ページの図 3.15 と同じ。ただし 3.3 節のエラー処理を追加すべき

図 5.13: Shader クラスの実装プログラム Shader.cpp

59 ページの図 5.5、60 ページの図 5.6 と同じ

図 5.14: 円形のポイントスプライトを描画するホストプログラム main.cpp

60 ページの図 5.7 と同じ

図 5.15: 円形のポイントスプライトを描画するバーテックスシェーダープログラム shader.vert

```
01 #version 120
02
03 varying vec3 out_color;
04
05 void main(void)
06 {
07     vec2 xy = 2.0*gl_PointCoord-vec2(1.0,1.0);
        // 座標値を [0,1]x[0,1] の範囲から [-1,1]x[-1,1] の範囲へ座標変換
08     if(length(xy) < 1.0){  //原点から xy までの距離が 1 より小さいならば
09         gl_FragColor = vec4(out_color, 0.0);
10     }else{
11         discard;                                  // この画素を描画しない
12     }
13 }
```

図 5.16: 円形のポイントスプライトを描画するフラグメントシェーダープログラム shader.frag

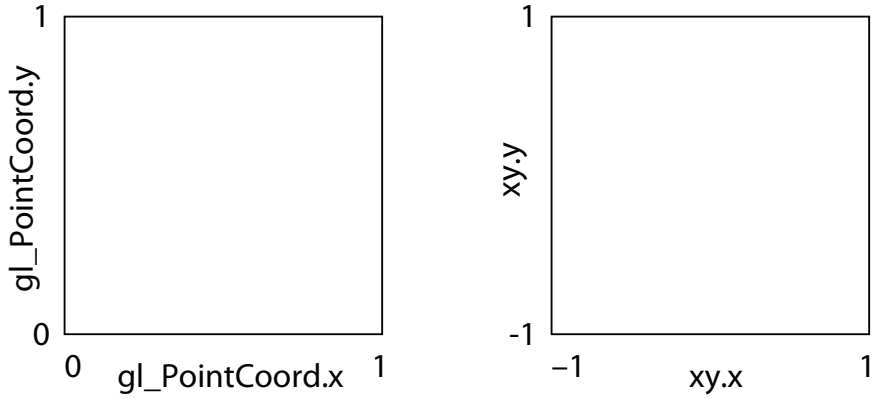

図 5.17: ポイントスプライトの正方形領域と gl_PointCoord、xy の座標値の関係

5.2.1　ホストプログラム

前節から変更しない。

5.2.2　バーテックスシェーダープログラムプログラム

前節から変更しない。

5.2.3　フラグメントシェーダープログラム

図 5.16 が変更後のフラグメントシェーダープログラムである。

このプログラム中の gl_PointCoord はフラグメントシェーダープログラムの vec2 型の予約変数である。ポイントスプライト描画時には正方形領域内の各画素の位置が、図 5.17 左のように、$[0,1] \times [0,1]$ の範囲の値としてラスタライザによって自動的に設定される。各フラグメントシェーダーは 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から、その位置に応じて色情報を計算すればよい。

図 5.16 のプログラムでは、まず、

```
07      vec2 xy = 2.0*gl_PointCoord-vec2(1.0,1.0);
```

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 xy へ計算している（図 5.17 を参照）。

次に、以下の一行

```
08      if(length(xy) < 1.0){  //原点から xy までの距離が 1 より小さいならば
```

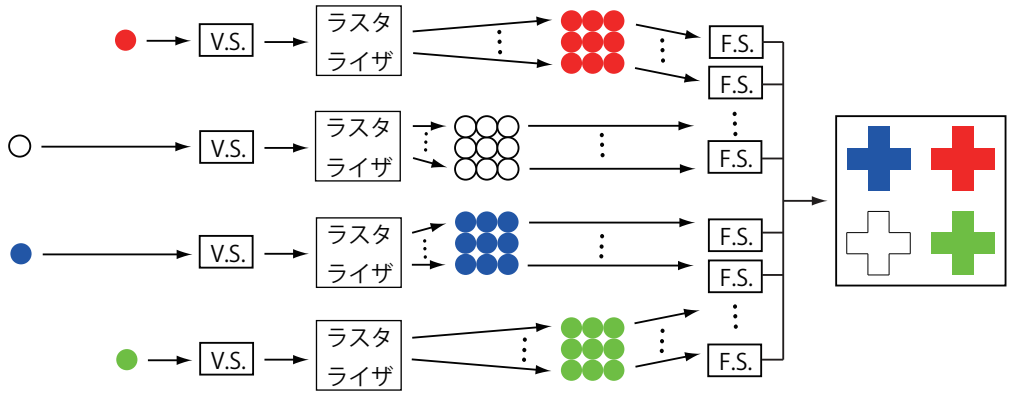

図 5.18: ポイントスプライトの動作例

は、xy の大きさ length(xy)（= |xy| =sqrt(xy.x*xy.x+xy.y*xy.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 vec2、vec3、vec4 型のベクトルの大きさを求める GLSL の組み込み関数である。

もし判定が真ならば、

```
09          gl_FragColor = vec4(out_color, 0.0);
```

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。「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

```
11          discard;                                // この画素を描画しない
```

に現れる discard は計算結果を無視（discard）することを表すフラグメントシェーダープログラムの予約語である。

5.2.4　実行結果

図 5.18 が実行の様子である。図 5.9 との違いは、フラグメントシェーダー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。

第6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方法は、線分やポイントスプライトとほとんど同様である。

6.1　簡単なポリゴン描画

ここでは、図 6.1 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 頂点にはそれぞれ赤、緑、白を割り当てた。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

$$\lambda_1 = \frac{|p - q_1|}{|p_1 - q_1|}, \quad \lambda_2 = \frac{|p - q_2|}{|p_2 - q_2|}, \quad \lambda_3 = \frac{|p - q_3|}{|p_3 - q_3|} \tag{6.1}$$

と定義する（図 6.2 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

$$\vec{C} = \lambda_1 \vec{C}_1 + \lambda_2 \vec{C}_2 + \lambda_3 \vec{C}_3 \tag{6.2}$$

とベクトル計算で求めることをいう。RGB カラー値だけではなく、任意の値が同様の線形補間できる。パラメータ $(\lambda_1, \lambda_2, \lambda_3)$ を重心座標（barycentric coordinate）と呼ぶ。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。

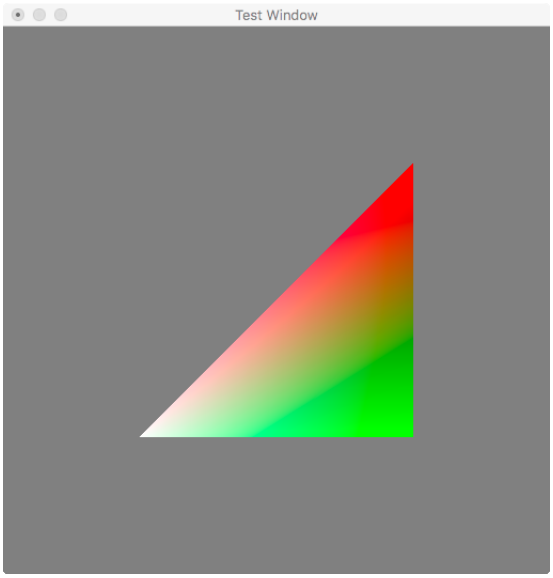

図 6.1: 画像例（その 6、一枚の三角形ポリゴンの塗りつぶしの場合）

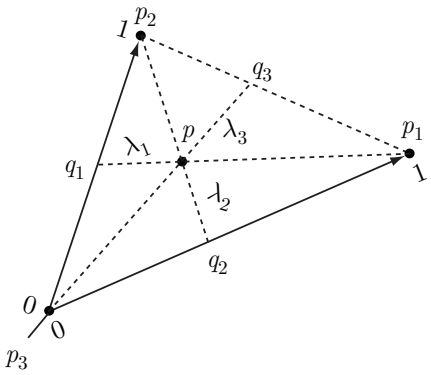

図 6.2: 重心座標系による RGB カラーの線形補間

31 ページの図 3.12 と同じ

図 6.3: 各種クラスを定義するヘッダープログラム `All.h`

32 ページの図 3.13 と同じ

図 6.4: `ArrayBuffer` クラスの実装プログラム `Buffer.cpp`

32 ページの図 3.14、33 ページの図 3.15 と同じ。ただし 3.3 節のエラー処理を追加すべき

図 6.5: Shader クラスの実装プログラム Shader.cpp

```
01 #include "All.h"
02
03 Shader *sp;
04
05 void initSystem(int argc, char *argv[])
06 {
07    glutInit(&argc,argv);
08     glutInitDisplayMode(GLUT_RGB|GLUT_SINGLE);
09     glutInitWindowSize(512,512);
10     glutCreateWindow("Test Window");
11     glClearColor(0.5,0.5,0.5,1);
12
13 #if defined(WIN32)
14     glewInit();
15 #endif
16
17 //  glEnable(GL_POINT_SPRITE);                // ポリゴン描画には不要
18 //  glEnable(GL_VERTEX_PROGRAM_POINT_SIZE);  // ポリゴン描画には不要
19
20     sp = new Shader("shader.vert","shader.frag");
21 }
22
```

図 6.6: 1 枚の三角形ポリゴンを描画するホストプログラム main.cpp（その 1、その 2 へ続く）

```
23 const int NUM_POINTS = 3;                          // 1 枚の三角形の 3 頂点
24
25 void initData()
26 {
27     Position2D pos[NUM_POINTS]; // これ以降のデータの定義は前節と同じ
28
29     pos[0].x = -0.5; pos[0].y = -0.5;
30     pos[1].x = +0.5; pos[1].y = +0.5;
31     pos[2].x = +0.5; pos[2].y = -0.5;
32
33     ArrayBuffer ab((float*)pos,2,NUM_POINTS);
34     sp->bindArrayBuffer("position",&ab);
35
36     RGB rgb[NUM_POINTS];
37     rgb[0].r = 1.0; rgb[0].g = 1.0; rgb[0].b = 1.0;
38     rgb[1].r = 1.0; rgb[1].g = 0.0; rgb[1].b = 0.0;
39     rgb[2].r = 0.0; rgb[2].g = 1.0; rgb[2].b = 0.0;
40
41     ArrayBuffer ab2((float*)rgb,3,NUM_POINTS);
42     sp->bindArrayBuffer("in_color",&ab2);
43
44 //  glLineWidth(10.0);                        // ポリゴン描画には不要
45 }
46
47 void display(void)
48 {
49     sp->use();
50     glClear(GL_COLOR_BUFFER_BIT);
51     sp->run(GL_TRIANGLES,NUM_POINTS);         // 三角形ポリゴンの描画
52 }
53
54 int main(int argc, char *argv[])
55 {
56     initSystem(argc,argv);
57     initData();
58
59     glutDisplayFunc(display);
60     glutMainLoop();
61     return 0;
62 }
```

図 6.7: 1 枚の三角形ポリゴンを描画するホストプログラム main.cpp（その 2）

```
01 #version 120
02
03 attribute vec2 position;
04 attribute vec3 in_color;
05
06 varying vec3 out_color;
07
08 void main(void)
09 {
10     gl_Position = vec4(position,0.0,1.0);
11     out_color = in_color;
12 //  gl_PointSize = 50;                          // ポリゴン描画には不要
13 }
```

図 6.8: 1 枚の三角形ポリゴンを描画するバーテックスシェーダープログラム shader.vert

```
01 #version 120
02
03 varying vec3 out_color;
04
05 void main(void)      // この処理は線分や正方形ポイントスプライトと同じ
06 {
07     gl_FragColor = vec4(out_color, 0.0);
08 }
```

図 6.9: 1 枚の三角形ポリゴンを描画するフラグメントシェーダープログラム shader.frag
（54 ページの図 4.17 に既出ではあるが、理解を容易にするために再掲する）

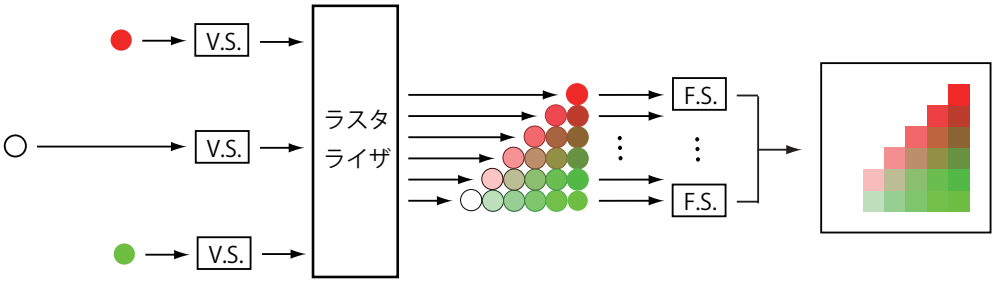

図 6.10: 三角形を描く場合の動作例

6.1.1　ホストプログラム

三角形は 3 頂点で定義されるから、データを準備する関数 `initData()` は図 6.7 のように変更する。前節までの例題では 4 頂点を定義していたが、単に頂点をひとつ減らしただけである。

線分やポイントスプライトのための処理は削除してもよい。ここではコメントアウトした。

図 6.7 の描画関数 `display()` では、描画モードを三角形ポリゴンのそれ `GL_TRIANGLES` に変更するだけでよい。

6.1.2　バーテックスシェーダープログラム

図 6.8 のバーテックスシェーダープログラムは前章までと同じである。ポイントスプライト関連をコメントアウトするだけでよい。

6.1.3　フラグメントシェーダープログラム

図 6.9 のフラグメントシェーダープログラムも前章の正方形ポイントスプライトを描画する場合と同じである。

6.1.4　実行結果

図 6.10 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ープログラムで処理される。3 頂点の情報はラスタライザに集約され、内点を含む画素点が求められる。なお、ラスタライザが集約する頂点の個数は、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 点を集約する。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個別のフラグメントシェーダーで色が計算され、それが画像となる。

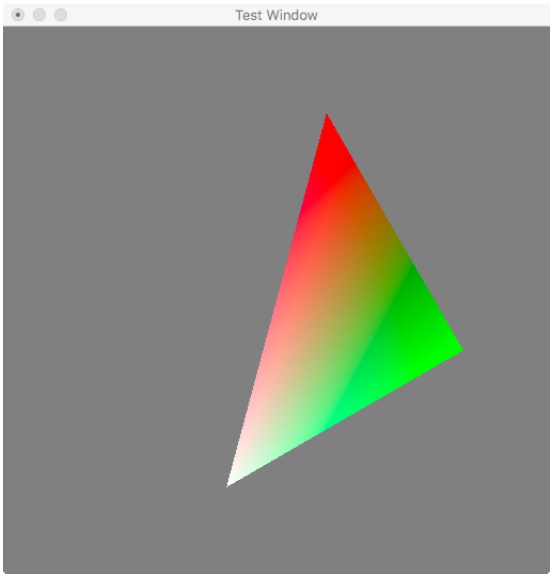

図 6.11: 画像例（その 7、三角形を 30 度回転させた場合）

6.2　シェーダープログラムへのパラメータの受け渡し（その 1）

実はホストプログラムからシェーダープログラムへのデータの受け渡しには二つの方法がある。

ひとつは、すでに述べたように、頂点データを attribute 変数としてバーテックスシェーダープログラムへ受け渡す方法である。この方法では、それぞれのバーテックスシェーダープログラムが異なる頂点データを受け取る。

もうひとつの方法は **uniform 変数**で受け渡す方法である。プログラム実行ではしばしば全てのバーテックスシェーダープログラム、全てのフラグメントシェーダープログラムが同じパラメータを受け取りたい場合がある。たとえば図 6.11 は、図 6.1 の三角形の全ての頂点の座標を反時計回りに一定角度（この場合は 30 度）だけ回転させた画像である。

座標の回転はバーテックスシェーダーで行うようにするとき、その回転角を全てのバーテックスシェーダープログラムで共有すべきである。**uniform 変数**はそのような用途に利用できる。`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ープログラム、フラグメントシェーダー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）。

概念図を図 6.12 に示す。uniform 変数の値はホストプログラムから設定する。シェーダープログラムからは uniform 変数の値を参照できるが、uniform 変数に値を代入するこ

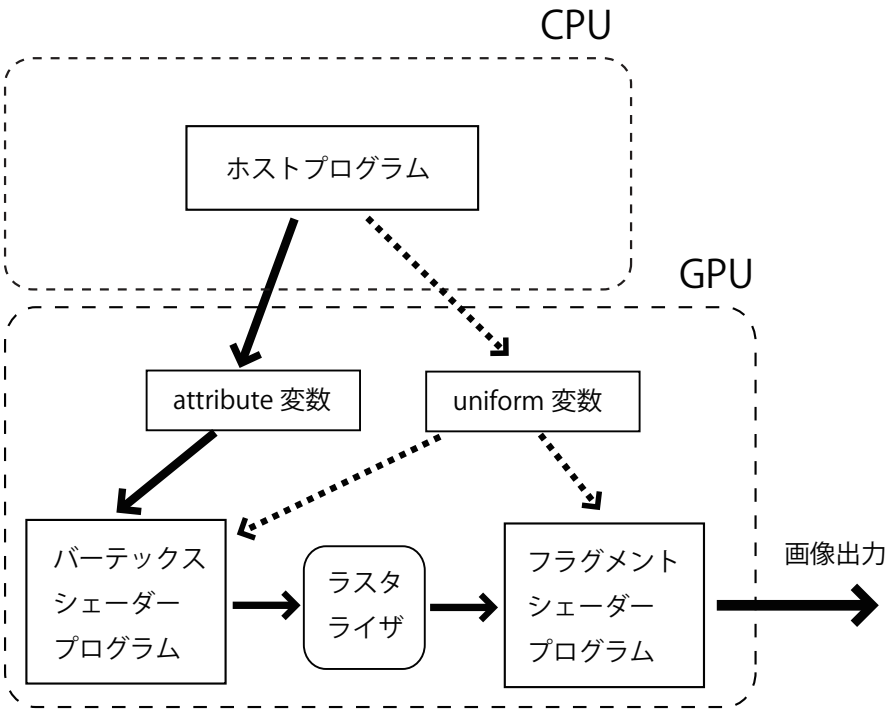

図 6.12: attribute 変数と uniform 変数の関係

とはできない[1]。

uniform 変数の具体的な利用法を以下に解説する。

[1]もしできるならば、複数のシェーダープログラムが uniform 変数へ異なる値を代入できることとなり、uniform 変数の値は不定となる。それは不都合であるから、シェーダープログラムからの代入はできない仕様となっている。

```
01 #include <iostream>
02 using namespace std;
03 #include "stdio.h"
04 #include "stdlib.h"
05
06 #if defined(WIN32)
07 #  pragma comment(lib, "glew32.lib")
08 #  include "glew.h"
09 #  include "glut.h"
10 #  include "glext.h"
11 #elif defined(__APPLE__) || defined(MACOSX)
12 #  include <GLUT/glut.h>
13 #else
14 #  define GL_GLEXT_PROTOTYPES
15 #  include <GL/glut.h>
16 #endif
17
18 struct Position2D {
19     float x;
20     float y;
21 };
22
23 struct RGB {
24     float r;
25     float g;
26     float b;
27 };
28
29 struct ArrayBuffer
30 {
31     GLuint    bufID;
32     int        size;
33
34    ArrayBuffer(float* data, int s, int n);
35 };
36
37 struct Shader
38 {
39     GLuint        program;
40
41     Shader(const char* vsn, const char* fsn);
42     void use();
43
44     void bindArrayBuffer(const char* vname, ArrayBuffer* ap);
45     void run(GLenum mode, int n);
46     void setFloat(const char* uname, float val);
                          // uniform 変数 uname に値 val を代入する関数
47
48     GLuint compileProgram(GLenum type, const GLchar *file);
49     void buildProgram(const GLchar *vsfile, const GLchar *fsfile);
50 };
```

図 6.13: 各種クラスを定義するヘッダープログラム All.h

32 ページの図 3.13 と同じ

図 6.14: ArrayBuffer クラスの実装プログラム Buffer.cpp

```
01 前略
02 ...
03 void Shader::setFloat(const char* uname, float val){
04     GLint p = glGetUniformLocation(program, uname);
                                              // uniform 変数の場所を取得
05     if(p < 0)  {                              // もし負数ならばエラー
06        cerr << "uniform name error: " << uname << endl;
07         exit(1);
08     }
09     glUniform1f(p,val);                     // uniform 変数に値を代入
10 }
```

図 6.15: Shader クラスの実装プログラム Shader.cpp（Shader::setFloat() のみ抜粋。前略部分は 32 ページの図 3.14、33 ページの図 3.15 と同じ。ただし 3.3 節のエラー処理を追加すべき）

```
01 #include "All.h"
02
03 Shader *sp;
04
05 void initSystem(int argc, char *argv[])
06 {
07     glutInit(&argc,argv);
08     glutInitDisplayMode(GLUT_RGB|GLUT_SINGLE);
09     glutInitWindowSize(512,512);
10     glutCreateWindow("Test Window");
11     glClearColor(0.5,0.5,0.5,1);
12
13 #if defined(WIN32)
14     glewInit();
15 #endif
16
17 //  glEnable(GL_POINT_SPRITE);
18 //  glEnable(GL_VERTEX_PROGRAM_POINT_SIZE);
19
20     sp = new Shader("shader.vert","shader.frag");
21 }
22
```

図 6.16: 1 枚の三角形ポリゴンを描画するホストプログラム main.cpp（その 1、その 2 へ続く）

```
23 const int NUM_POINTS = 3;
24
25 void initData()
26 {
27     Position2D pos[NUM_POINTS];
28
29     pos[0].x = -0.5; pos[0].y = -0.5;
30     pos[1].x = +0.5; pos[1].y = +0.5;
31     pos[2].x = +0.5; pos[2].y = -0.5;
32
33     ArrayBuffer ab((float*)pos,2,NUM_POINTS);
34     sp->bindArrayBuffer("position",&ab);
35
36     RGB rgb[NUM_POINTS];
37     rgb[0].r = 1.0; rgb[0].g = 1.0; rgb[0].b = 1.0;
38     rgb[1].r = 1.0; rgb[1].g = 0.0; rgb[1].b = 0.0;
39     rgb[2].r = 0.0; rgb[2].g = 1.0; rgb[2].b = 0.0;
40
41     ArrayBuffer ab2((float*)rgb,3,NUM_POINTS);
42     sp->bindArrayBuffer("in_color",&ab2);
43
44 //  glLineWidth(10.0);
45 }
46
47 #if defined(WIN32)          // この 4 行は M_PI の定義の読み込みのため
48 #define _USE_MATH_DEFINES
49 #endif
50 #include <math.h>
51
52 void display(void)
53 {
54     sp->use();
55     sp->setFloat("theta", 30*M_PI/180.0);      // theta に値を代入
56     glClear(GL_COLOR_BUFFER_BIT);
57     sp->run(GL_TRIANGLES,NUM_POINTS);
58 }
59
60 int main(int argc, char *argv[])
61 {
62     initSystem(argc,argv);
63     initData();
64
65     glutDisplayFunc(display);
66     glutMainLoop();
67     return 0;
68 }
```

図 6.17: 1 枚の三角形ポリゴンを描画するホストプログラム main.cpp（その 2）

```
01 #version 120
02
03 attribute vec2 position;
04 attribute vec3 in_color;
05
06 varying vec3 out_color;
07
08 uniform float theta;                              // uniform 変数の宣言
09
10 void main(void)
11 {
12     float x = cos(theta)*position.x-sin(theta)*position.y;
13     float y = sin(theta)*position.x+cos(theta)*position.y;
                                        // 頂点の座標を theta だけ回転
14
15     gl_Position = vec4(x,y,0.0,1.0);
16     out_color = in_color;
17 }
```

図 6.18: 1 枚の三角形ポリゴンを描画するバーテックスシェーダープログラム shader.vert

71 ページの図 6.9 と同じ

図 6.19: 1 枚の三角形ポリゴンを描画するフラグメントシェーダープログラム shader.frag

6.2.1　Shader クラスの拡張

図 6.13 のように、ホストプログラムから GPU のメモリ内の float 型 uniform 変数に値を代入するためのメンバー関数

```
void setFloat(const char* uname, float val);
```

を Shader クラスに新たに追加する。ここに、第 1 引数が uniform 変数名、第 2 引数が代入する値である。

その実装は図 6.15 の通りである。関数本体の代入文：

```
04      GLint p = glGetUniformLocation(program, uname);
```

は、バーテックスシェーダープログラムおよび（または）フラグメントシェーダープログラム中から uname と同じ変数名の uniform 変数を見つけ、その場所（location）を変数 p へ代入する。関数 glGet**Uniform**Location() は、3.1 節に出てきた glGet**Attrib**Location() 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 p の値が負値ならば、その変数名がバーテックスシェーダープログラムとフラグメントシェーダープログラムのどちらにも宣言されていないことを意味するから、その場合、エラーメッセージを出力し、プログラム全体を強制終了するように実装した。

関数呼び出し：

```
09      glUniform1f(p,val);                     // uniform 変数に値を代入
```

は uniform 変数の場所に float 値 val を代入する。glUniform1f() と類似した関数として、glUniform2f()、glUniform3f()、glUniform4f() 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 glUniform1i() 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。

6.2.2　ホストプログラム

このプログラムでは、uniform 変数 theta （θ のつもり）に三角形ポリゴンを回転させる角度（ラジアンモード）を格納する

そこで、図 6.17 の描画関数 display() では、その 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：

```
55      sp->setFloat("theta", 30*M_PI/180.0);       // theta に値を代入
```

を実行する。なお、円周率を表す定数 M_PI を使用するために、以下を display() の前に追加した。

```
47 #if defined(WIN32)          // この 4 行は M_PI の定義の読み込みのため
48 #define _USE_MATH_DEFINES
49 #endif
50 #include <math.h>
```

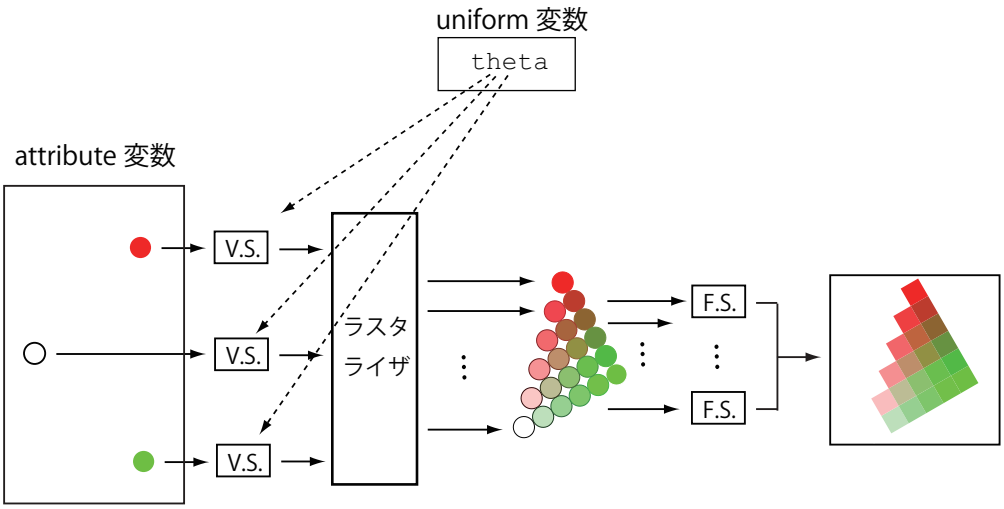

図 6.20: 三角形が回転する場合の動作例

6.2.3　バーテックスシェーダープログラム

バーテックスシェーダープログラムでは、回転角に対応する変数 theta を uniform 宣言する。

```
08 uniform float theta;                    // uniform 変数の宣言
```

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、回転後の座標値を gl_Position へ代入する。

なお、バーテックスシェーダープログラムで共有される theta について cos(theta)、sin(theta) を全てのバーテックスシェーダーで繰り返し実行するのはややコストが高い。ここでは行わないが、あらかじめホストプログラムで計算しておき、二つの unifrom 変数に cos(theta)、sin(theta) を格納する方が効率的である。

6.2.4　フラグメントシェーダープログラム

フラグメントシェーダープログラムは前節から変更しない。

6.2.5　実行結果

図 6.20 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ーで 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。

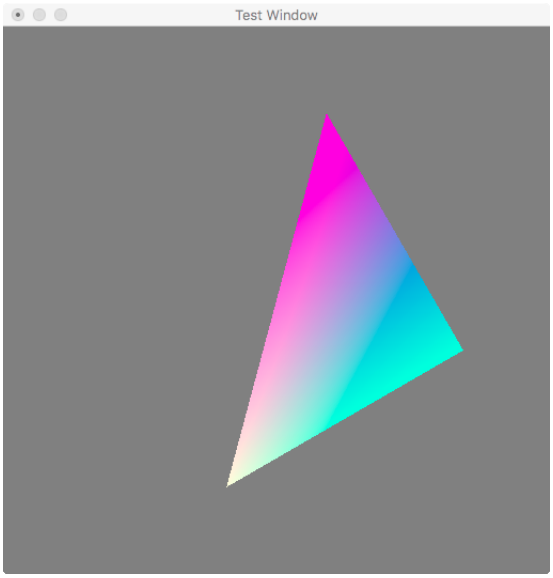

図 6.21: 画像例（その 8、三角形を 30 度回転させた場合）

6.3　シェーダー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ー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ープログラムでも全く同様に参照できる。図 6.21 は前節の `theta` をフラグメントシェーダープログラムからも参照して三角形の色を変えた描画例である。

76 ページの図 6.13 と同じ

図 6.22: 各種クラスを定義するヘッダープログラム All.h

32 ページの図 3.13 と同じ

図 6.23: ArrayBuffer クラスの実装プログラム Buffer.cpp

77 ページの図 6.15 と同じ

図 6.24: Shader クラスの実装プログラム Shader.cpp

77 ページの図 6.16、78 ページの図 6.17 と同じ

図 6.25: 1 枚の三角形ポリゴンを描画するホストプログラム main.cpp

79 ページの図 6.18 と同じ

図 6.26: 1 枚の三角形ポリゴンを描画するバーテックスシェーダープログラム shader.vert

```
01 #version 120
02
03 varying vec3 out_color;
04
05 uniform float theta;
06
07 void main(void)
08 {
09     gl_FragColor = vec4(out_color.r, out_color.g, cos(theta), 0.0);
                                        // 青の輝度値を cos(theta) に変更
10 }
```

図 6.27: 1 枚の三角形ポリゴンを描画するフラグメントシェーダープログラム shader.frag

6.3.1　ホストプログラム

ホストプログラムは前節と同じである。

6.3.2　バーテックスシェーダープログラム

バーテックスシェーダープログラムは前節と同じである。

6.3.3　フラグメントシェーダープログラム

フラグメントシェーダープログラムでは uniform 変数 `theta` を図 6.18 と同様に宣言し、参照できる。

図 6.27 では、以下のように

```
09      gl_FragColor = vec4(out_color.r, out_color.g, cos(theta), 0.0);
```

画素点の青色成分を `cos(theta)` と計算している。$\texttt{cos(theta)} = \cos(30° \cdot \pi/180) = 0.866..$ であるから、図 6.11 に比べてかなり青色成分が加わった画像になる。なお、`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` がそれであるまた `cos(theta)` は `theta` の値次第では負値となるが、負の輝度値は単に 0 と解釈される。

6.3.4　実行結果

図 6.28 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ープログラムに入力された色の青色成分が変更されて、画像バッファに出力される。

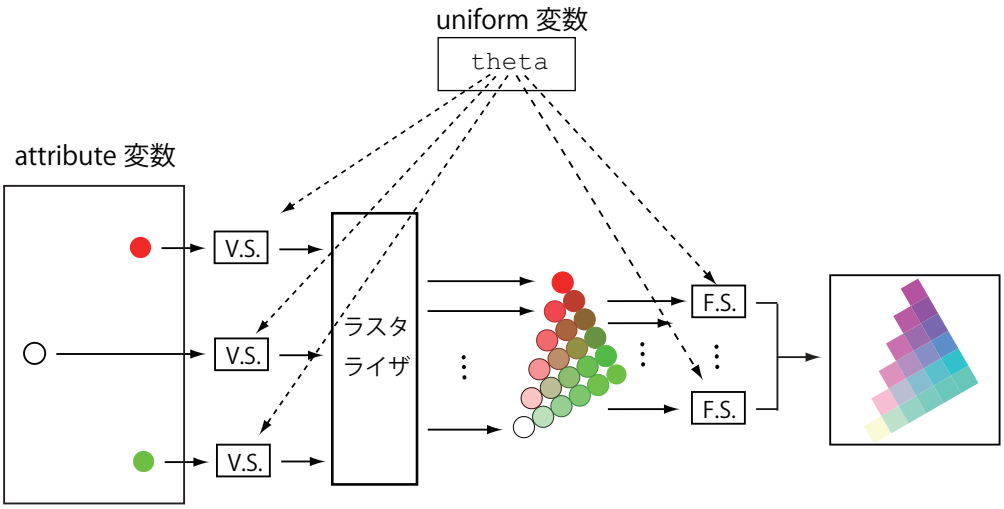

図 6.28: 三角形の色が変わる動作例

第7章　CGアニメーション

この章では、前節の三角形（図 6.21）が回転する CG アニメーションを行う。アニメーションを実現するためには、以下の二つの課題を解決せねばならない。

タイマー起動 アニメーションでは一定時間毎に再描画を行う必要がある。glut ではそれをコールバックを用いて実装する。なお、この方法はあくまでも glut の独自の方法であって、OpenGL プログラムをその他のフレームワーク（GLFW など）で実行する場合には異なる手法になることを注意する。

ダブルバッファ 1 枚の CG 画像の描画に必要な時間が数十ミリ秒以上に及ぶ場合、アニメーションがチラついて見えることがある。これは、ウインドウ上での描画の過程 — ウインドウが背景色でクリアされ、順に三角形や線分がウインドウに描画されていく過程 — の繰り返しがディスプレイ上に逐一表示されるからである。これを解決するために、リアルタイム CG アニメーションでは画像バッファを 2 枚用意し、交互に表示と描画を行う方法が一般的である。

ところで、全体のプログラムの様子がかなり分かりにくくなってきた。そこで前半のまとめを兼ねて、次ページ以降にプログラムを全て掲載する。ただし、あまりに長大なクラスの実装は省略する。

76 ページの図 6.13 と同じだが、再掲

```
01 #include <iostream>
02 using namespace std;
03 #include "stdio.h"
04 #include "stdlib.h"
05
06 #if defined(WIN32)
07 #  pragma comment(lib, "glew32.lib")
08 #  include "glew.h"
09 #  include "glut.h"
10 #  include "glext.h"
11 #elif defined(__APPLE__) || defined(MACOSX)
12 #  include <GLUT/glut.h>
13 #else
14 #  define GL_GLEXT_PROTOTYPES
15 #  include <GL/glut.h>
16 #endif
17
18 struct Position2D {
19     float x;
20     float y;
21 };
22
23 struct RGB {
24     float r;
25     float g;
26     float b;
27 };
28
29 struct ArrayBuffer
30 {
31     GLuint    bufID;
32     int        size;
33
34    ArrayBuffer(float* data, int s, int n);
35 };
36
37 struct Shader
38 {
39     GLuint        program;
40
41     Shader(const char* vsn, const char* fsn);
42     void use();
43
44     void bindArrayBuffer(const char* vname, ArrayBuffer* ap);
45     void run(GLenum mode, int n);
46     void setFloat(const char* uname, float val);
47
48     GLuint compileProgram(GLenum type, const GLchar *file);
49     void buildProgram(const GLchar *vsfile, const GLchar *fsfile);
50 };
```

図 7.1: 各種クラスを定義するヘッダープログラム `All.h`

32 ページの図 3.13 と同じだが、再掲

```
01 #include "All.h"
02
03 ArrayBuffer::ArrayBuffer(float* data, int s, int n)
04 {
05     size = s;
06     glGenBuffers(1, &bufID);
07     glBindBuffer(GL_ARRAY_BUFFER, bufID);
08     glBufferData(GL_ARRAY_BUFFER, sizeof(float)*size*n,
09                     data, GL_STATIC_DRAW);
10     glBindBuffer(GL_ARRAY_BUFFER, NULL);
11 }
```

図 7.2: ArrayBuffer クラスの実装プログラム Buffer.cpp

```
01 #include "All.h"
02
03 Shader::Shader(const char* vsn, const char* fsn)
04 {
05     buildProgram(vsn, fsn);
06 }
07
08 void Shader::bindArrayBuffer(const char* vname, ArrayBuffer* ap)
09 {
10     glBindBuffer(GL_ARRAY_BUFFER, ap->bufID);
11     GLint p = glGetAttribLocation(program,vname);
12     if(p < 0) {
13         cerr << "attribute name error: " << vname << endl;
14         exit(1);
15     }
16     glVertexAttribPointer(p, ap->size, GL_FLOAT, GL_FALSE, 0, 0);
17     glEnableVertexAttribArray(p);
18}
19
20 void Shader::use()
21 {
22     glUseProgram(program);
23 }
24
25 void Shader::run(GLenum mode, int n){
26     glDrawArrays(mode,0,n);
27     glutSwapBuffers();                              //glFlush() と置換
28 }
29
```

図 7.3: Shader クラスの実装プログラム Shader.cpp（その 1、その 2 へ続く）

```
30 void Shader::setFloat(const char* uname, float val){
31     GLint p = glGetUniformLocation(program, uname);
32     if(p < 0)  {
33        cerr << "uniform name error: " << uname << endl;
34         exit(1);
35     }
36     glUniform1f(p,val);
37 }
38
39 GLuint Shader::compileProgram(GLenum type, const GLchar *file)
40 {
41     FILE* fp = fopen(file, "r");
   ...

   中略、39 ページの図 3.20 の通り

   ...
42     return shader;
43 }
44
45 void Shader::buildProgram(const GLchar *vsn, const GLchar *fsn)
46 {
47     GLuint vs = compileProgram(GL_VERTEX_SHADER, vsn);
   ...

   中略、40 ページの図 3.21 の通り

   ...
48     glLinkProgram(program);
49 }
```

図 7.4: Shader クラスの実装プログラム Shader.cpp（その 2）

```
01 #include "All.h"
02
03 Shader *sp;
04
05 void initSystem(int argc, char *argv[])
06 {
07     glutInit(&argc,argv);
08     glutInitDisplayMode(GLUT_RGB | GLUT_DOUBLE);
09                                 // GLUT_SINGLE を GLUT_DOUBLE へ置換
10     glutInitWindowSize(512,512);
11     glutCreateWindow("Test Window");
12     glClearColor(0.5,0.5,0.5,1);
13
14 #if defined(WIN32)
15     glewInit();
16 #endif
17
18 //  glEnable(GL_POINT_SPRITE);
19 //  glEnable(GL_VERTEX_PROGRAM_POINT_SIZE);
20
21     sp = new Shader("shader.vert","shader.frag");
22 }
23
```

図 7.5: 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.cpp（その 1、その 2 に続く）

```
24 const int NUM_POINTS = 3;
25
26 void initData()
27 {
28     Position2D pos[NUM_POINTS];
29
30     pos[0].x = -0.5; pos[0].y = -0.5;
31     pos[1].x = +0.5; pos[1].y = +0.5;
32     pos[2].x = +0.5; pos[2].y = -0.5;
33
34     ArrayBuffer ab((float*)pos,2,NUM_POINTS);
35    sp->bindArrayBuffer("position",&ab);
36
37     RGB rgb[NUM_POINTS];
38     rgb[0].r = 1.0; rgb[0].g = 1.0; rgb[0].b = 1.0;
39     rgb[1].r = 1.0; rgb[1].g = 0.0; rgb[1].b = 0.0;
40     rgb[2].r = 0.0; rgb[2].g = 1.0; rgb[2].b = 0.0;
41
42     ArrayBuffer ab2((float*)rgb,3,NUM_POINTS);
43     sp->bindArrayBuffer("in_color",&ab2);
44
45 //  glLineWidth(10.0);
46 }
47
48 double theta = 0.0;                          // 回転角度を大域変数に保持
49
50 void display(void)
51 {
52     sp->use();
53     sp->setFloat("theta", theta);
54     glClear(GL_COLOR_BUFFER_BIT);
55     sp->run(GL_TRIANGLES, NUM_POINTS);
56     theta += 0.02;                                    // 回転角度を更新
57 }
58
59 void timer(int num) {                                 // タイマー関数
60     glutPostRedisplay(); // 再描画を glut のウインド管理システムに依頼
61     glutTimerFunc(50, timer, num+1);          // 次のタイマーをセット
62 }
63
64 int main(int argc, char *argv[])
65 {
66     initSystem(argc,argv);
67     initData();
68
69     glutDisplayFunc(display);
70     glutTimerFunc(50 , timer , 1);          // 初回のタイマーのセット
71     glutMainLoop();
72     return 0;
73 }
```

図 7.6: 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.cpp（その 2）

79 ページの図 6.18 と同じだが、再掲

```
01 #version 120
02
03 attribute vec2 position;
04 attribute vec3 in_color;
05
06 varying vec3 out_color;
07
08 uniform float theta;
09
10 void main(void)
11 {
12     float x = cos(theta)*position.x-sin(theta)*position.y;
13     float y = sin(theta)*position.x+cos(theta)*position.y;
14
15     gl_Position = vec4(x,y,0.0,1.0);
16     out_color = in_color;
17 }
```

図 7.7: 1 枚の三角形ポリゴンを描画するバーテックスシェーダープログラム shader.vert

84 ページの図 6.27 と同じだが、再掲

```
01 #version 120
02
03 varying vec3 out_color;
04
05 uniform float theta;
06
07 void main(void)
08 {
09     gl_FragColor = vec4(out_color.r, out_color.g, cos(theta), 0.0);
10 }
```

図 7.8: 1 枚の三角形ポリゴンを描画するフラグメントシェーダープログラム shader.frag

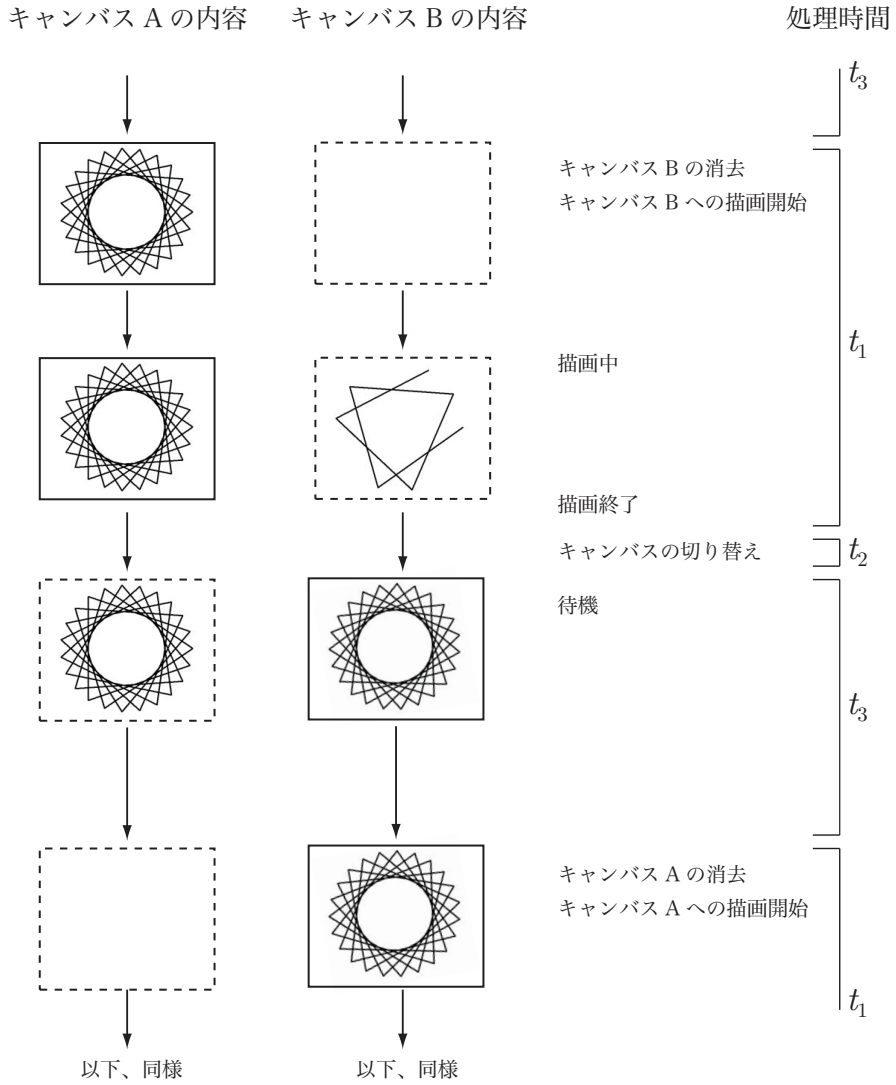

図 7.9: ダブルバッファ概念図

7.1　ポリゴンの回転のアニメーション

前章のプログラムを改造する。

7.1.1　ダブルバッファの実装

先述したように、一枚の画像バッファ用いる方法で CG アニメーションを行うと、その画像バッファが背景色でクリアされ、図形が描画される一部始終がディスプレイ上で見える。そのため、描画がよほど短時間で行われない限り、クリアされ、描画される様子がディスプレイ上で繰り返され、それが画面のチラつきの原因となる。それを避ける最も単純な

方法は、画像バッファを 2 枚用いる方法である。これをダブルバッファ法と呼び、CG アニメーションでは一般的に用いられている。この方法では、図 7.9 に示すように、ディスプレイ上に表示されている画像バッファと、ディスプレイ上には表示されず、描画に用いられる画像バッファを順次切り替えることで、クリア、描画のチラつきをディスプレイ上には見せない。

OpenGL ではこのダブルバッファ法をごく簡単な関数呼び出しで実行可能である。

まず、図 7.5 の関数呼び出し：

```
08      glutInitDisplayMode(GLUT_RGB | GLUT_DOUBLE);
```

の GL_DOUBLE によってダブルバッファを用いることを宣言する。

次に、図 7.6 の関数呼び出し：

```
55      sp->run(GL_TRIANGLES, NUM_POINTS);
```

の実装である図 7.3 の中の関数呼び出し：

```
27      glutSwapBuffers();                          //glFlush() と置換
```

が、2 枚の画像バッファの表示/非表示を瞬時に切り替える。display() が呼ばれ、新しい CG 画像が描画される度に上の glutSwapBuffers() が実行され、バッファが切り替わっていく。

これで画面のチラつきは解決する。

7.1.2　タイマー起動

CG 画像を、たとえば 1 秒間に 20 枚作ることを考えよう。そのとき、50 ミリ秒（= 1000 ミリ秒/20）毎に CG 画像を更新していく必要がある。画像を更新するタイミングを正確にその時間間隔に制御するために、GLUT ではタイマーを用意している。その一連の処理を図 7.10 を参考にしつつ解説する。

まず、図 7.6 の main 関数では glutMainLoop() を呼び出す前に関数呼び出し：

```
70      glutTimerFunc(50, timer, 1);          // 初回のタイマーのセット
```

を行う。この関数が呼び出されると、glut のタイマー管理システムのタイマーが起動し、第 1 引数で与えられた時間（単位はミリ秒）後に第 2 引数の関数を第 3 引数の実引数で自動的に呼び出すことが予約される。よってこの場合、システムは 50 ミリ秒後に timer(1) を呼び出す。

新たに作成する関数 timer()（図 7.6 の 59〜62 行目）では、まず、再描画のための関数呼び出し：

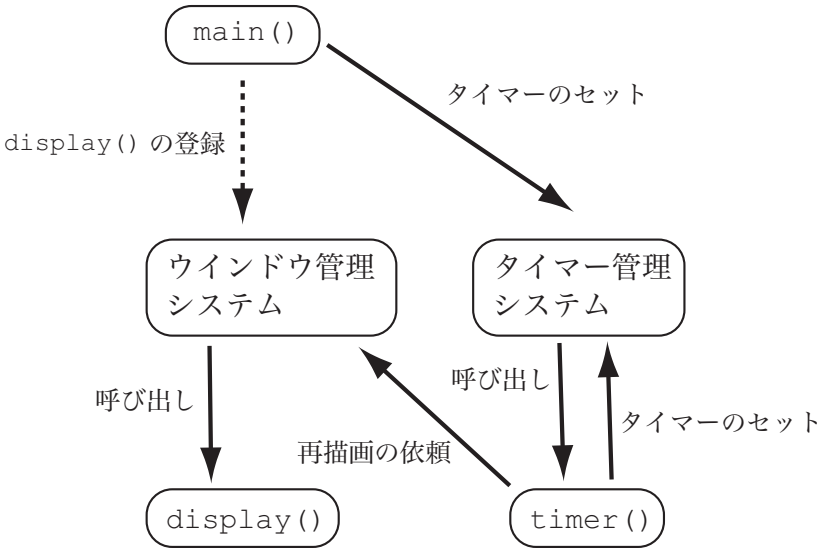

図 7.10: タイマー起動の概念図

```
60      glutPostRedisplay(); // 再描画を glut のウインド管理システムに依頼
```

を行う。この関数は、glut のウインドウ管理システムに再描画を依頼する関数である。ウインドウ管理システムは、あらかじめ登録された関数 `display()` を実行する。なお、関数 `timer()` の中から直接、描画関数 `display()` を呼び出すべきではない。描画処理は `glutMainLoop()` の後ろに居るウインドウ管理システムが管理しており、ユーザープログラムで勝手に描画を行うことはシステムに混乱、障害をもたらす可能性がある。

次に、

```
61      glutTimerFunc(50, timer, num+1);          // 次のタイマーをセット
```

によって、再度、50 ミリ秒後に `timer()` を呼び出すことを予約する。ただし、`main` 関数での呼び出し `glutTimerFunc(50, timer, 1)` とは異なり、第 3 引数を `num+1` にしている。よって次回に `timer()` が呼び出されたときには引数 `num` の値は 1 増える。つまり、`timer()` が呼び出される度に `num` の値は増えていくから、その値を見れば、`timer()` が過去に呼び出された回数を知ることができる。それを描画に利用することもできる。

このプログラムではタイマーが 50 ミリ秒毎に起動されるから、1 秒間に $1000/50 = 20$ 回の描画が行われる。すなわち、秒間描画枚数 FPS（Frames Per Sec）は 20 である。仮に 1000FPS を意図し、1 ミリ秒毎にタイマーが起動するように設定しても、もちろんそれは不可能である。

7.1.3　描画内容の更新

図 7.6 の関数 `display()` では大域変数 `theta` の値を関数呼び出し：

```
53      sp->setFloat("theta", theta);
```

によって、GPU の uniform 変数 `theta` へ代入している。その大域変数 `theta` の値は

```
56      theta += 0.02;                                  // 回転角度を更新
```

によって、0.02 ラジアンずつ増加する。よって、描画のたびに三角形が 0.02 ラジアンずつ回転していく。

上に述べたようにこのプログラムの FPS は 20 であるから、1 秒間で回転角度は 0.02 rad. $\times$ $20 \approx 23$ deg. だけ増える。よって三角形がちょうど 1 回転するには、$360/23 = 15.65$ 秒だけ掛かる計算となる。確認してほしい。

第8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。この画像を文字通りテクスチャ（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...」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つもりである。

8.1　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8.1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、...、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ープログラムでは、このテクスチャデータを $[0,1]$ の範囲の1次元座標値で参照する。

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。

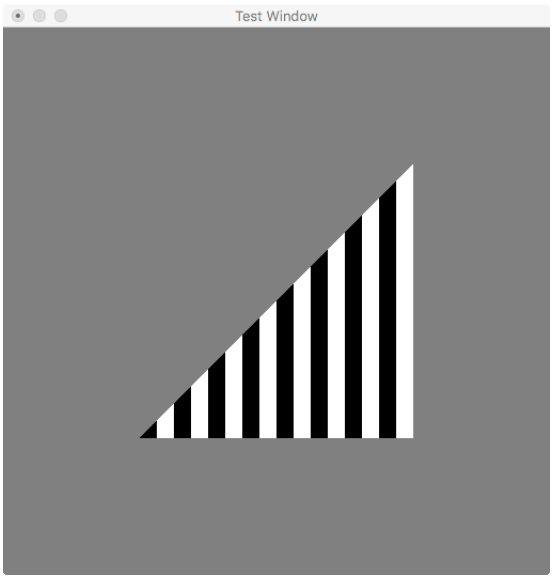

図 8.1: 画像例（その 9、1 次元テクスチャ利用の場合、CG アニメーションの最初の 1 枚に相当）

```
01 #include <iostream>
02 using namespace std;
03 #include "stdio.h"
04 #include "stdlib.h"
05
06 #if defined(WIN32)
07 #  pragma comment(lib, "glew32.lib")
08 #  include "glew.h"
09 #  include "glut.h"
10 #  include "glext.h"
11 #elif defined(__APPLE__) || defined(MACOSX)
12 #  include <GLUT/glut.h>
13 #else
14 #  define GL_GLEXT_PROTOTYPES
15 #  include <GL/glut.h>
16 #endif
17
18 struct Position2D
19 {
20     float x;
21     float y;
22 };
23
```

図 8.2: 各種クラスを定義するヘッダープログラム `All.h`（その 1、その 2 へ続く）

```
24 struct RGB
25 {
26     float r;
27     float g;
28     float b;
29 };
30
31 struct RGBA                 // RGB カラー値とアルファ値からなる構造体
32 {                           // テクスチャに使用
33     float r;
34     float g;
35     float b;
36     float a;
37 };
38
39 struct ArrayBuffer
40 {
41     GLuint    bufID;
42     int        size;
43
44     ArrayBuffer(float* data, int s, int n);
45 };
46
47 struct Texture1D                          // 1 次元テクスチャクラス
48 {
49     GLuint  texID;                            // テクスチャの参照番号
50     GLint   num;                              // テクスチャの装置番号
51
52     Texture1D(int tnum, void* data, int w);       // コンストラクタ
53 };
54
55 struct Shader
56 {
57     GLuint        program;
58
59     Shader(const char* vsn, const char* fsn);
60     void use();
61
62     void bindArrayBuffer(const char* vname, ArrayBuffer* ap);
63     void bindTexture(const char* vname, Texture1D* tp);
                                          // シェダーとテクスチャの結合
64     void run(GLenum mode, int n);
65     void setFloat(const char* uname, float val);
66
67     GLuint compileProgram(GLenum type, const GLchar *file);
68     void buildProgram(const GLchar *vsfile, const GLchar *fsfile);
69 };
```

図 8.3: 各種クラスを定義するヘッダープログラム `All.h`（その 2）

89 ページの図 7.2 と同じ

図 8.4: ArrayBuffer クラスの実装プログラム Buffer.cpp

```
01 Texture1D::Texture1D(int tnum, void* data, int w)
02 {
03     num = tnum;                              // テクスチャ装置番号の保存
04     glGenTextures (1, &texID);
05     glBindTexture(GL_TEXTURE_1D,texID);       // 参照番号の新規割付
06     glTexParameteri(GL_TEXTURE_1D, GL_GENERATE_MIPMAP,
07                     GL_FALSE);                 // ミップマップの設定
08     glTexParameteri(GL_TEXTURE_1D, GL_TEXTURE_MAG_FILTER,
09                     GL_NEAREST);       // データの読み込み方法の設定
10     glTexParameteri(GL_TEXTURE_1D, GL_TEXTURE_MIN_FILTER,
11                     GL_NEAREST);
                          // ミップマップからのデータの読み込み方法の設定
12     glTexImage1D(GL_TEXTURE_1D, 0, GL_RGBA, w, 0, GL_RGBA,
13                  GL_FLOAT, data);  // テクスチャデータの GPU への転送
14 }
```

図 8.5: 1 次元テクスチャクラス Texture1D のクラスの実装プログラム Textures.cpp

```
...
01 void Shader::bindTexture(const char* vname, Texture1D* tp)
02 {
03     glBindTexture(GL_TEXTURE_1D, tp->texID);
                                    // 1 次元テクスチャの設定処理の開始
04     GLint p = glGetUniformLocation(program, vname);
                                              // テクスチャの場所の取得
05     if(p < 0) {                               // もし負ならばエラー
06         cerr << "texture1d name error: "<< vname << endl;
07         exit(1);
08     }
09     glUniform1i(p, tp->num);          // テクスチャに装置番号を設定
10     glActiveTexture(GL_TEXTURE0+(tp->num));
                                    // このテクスチャを利用可能にする
11 }
...
```

図 8.6: Shader クラスの実装プログラム Shader.cpp（その他の部分は、89 ページの図 7.3、90 ページの図 7.4 と同じ）

```
01 #include "All.h"
02
03 Shader *sp;
04
05 void initSystem(int argc, char *argv[])
06 {
07     glutInit(&argc,argv);
08     glutInitDisplayMode(GLUT_RGB | GLUT_DOUBLE);
09     glutInitWindowSize(512,512);
10     glutCreateWindow("Test Window");
11     glClearColor(0.5,0.5,0.5,1);
12
13 #if defined(WIN32)
14     glewInit();
15 #endif
16
17 //  glEnable(GL_POINT_SPRITE);
18 //  glEnable(GL_VERTEX_PROGRAM_POINT_SIZE);
19
20     sp = new Shader("shader.vert","shader.frag");
21 }
22
23 const int NUM_POINTS = 3;
24
25 void initData()
26 {
27     Position2D pos[NUM_POINTS];
28
29     pos[0].x = -0.5; pos[0].y = -0.5;
30     pos[1].x = +0.5; pos[1].y = +0.5;
31     pos[2].x = +0.5; pos[2].y = -0.5;
32
33     ArrayBuffer ab((float*)pos,2,NUM_POINTS);
34     sp->bindArrayBuffer("position",&ab);
35
36     float u[NUM_POINTS];               // テクスチャ座標のための配列
37     u[0] = 0.0;
38     u[1] = 1.0;                        // 頂点への 1 次元座標値の代入
39     u[2] = 1.0;
40
41     ArrayBuffer ab3(u,1,NUM_POINTS); // 配列バッファオブジェクトの生成
42     sp->bindArrayBuffer("in_u",&ab3);      // attribute 変数との結合
43
```

図 8.7: 縞模様のテクスチャを描画するホストプログラム main.cpp（その 1、その 2 に続く）

```
44 const int TEXTURE_SIZE = 16;       // 1 次元テクスチャデータのサイズ
45     RGBA tex[TEXTURE_SIZE];             // テクスチャデータ用の配列
46     for(int i = 0; i < TEXTURE_SIZE; i++){
47         tex[i].r = i%2;
48         tex[i].g = i%2;                    // 0, 1, 0, 1, 0, 1, ...
49         tex[i].b = i%2;
50         tex[i].a = 0.0;
51     }
52     Texture1D t1(0, tex, TEXTURE_SIZE);
                                          // テクスチャオブジェクトの生成
53     sp->bindTexture("tex1", &t1);   // テクスチャuniform 変数との結合
54
55 //    glLineWidth(10.0);
56 }
57
58 void display(void)
59 {
60     sp->use();
61     sp->setFloat("theta", theta);
62     glClear(GL_COLOR_BUFFER_BIT);
63     sp->run(GL_TRIANGLES, NUM_POINTS);
64     theta += 0.02;
65 }
66
67 void timer(int num) {
68     glutPostRedisplay();
69     glutTimerFunc(500 , timer , num+1);   // 描画速度を 2FPS に変更
70 }
71
72 int main(int argc, char *argv[])
73 {
74    initSystem(argc,argv);
75     initData();
76
77     glutDisplayFunc(display);
78     glutTimerFunc(500 , timer , 1);       // 描画速度を 2FPS に変更
79     glutMainLoop();
80     return 0;
81 }
```

図 8.8: 縞模様のテクスチャを描画するホストプログラム main.cpp（その 2）

```
01 #version 120
02
03 attribute vec2 position;
04 attribute float in_u;                    // 1 次元テクスチャ座標値の入力
05
06 varying float out_u;                     // 1 次元テクスチャ座標値の出力
07
08 uniform float theta;
09
10 void main(void)
11 {
12     float x = cos(theta)*position.x-sin(theta)*position.y;
13     float y = sin(theta)*position.x+cos(theta)*position.y;
14     gl_Position = vec4(x, y, 0.0, 1.0);
15
16     out_u = in_u;              // 1 次元テクスチャ座標値を単純にコピー
17 }
```

図 8.9: 縞模様のテクスチャを描画するバーテックスシェーダープログラム shader.vert

```
01 #version 120
02
03 varying float out_u;                    // 1 次元テクスチャ座標値の入力
04
05 uniform sampler1D tex1;         // 1 次元テクスチャ型 uniform 変数の宣言
06
07 void main(void)
08 {
09     gl_FragColor = texture1D(tex1, out_u);
                                          // テクスチャデータの読み込み
10 }
```

図 8.10: 縞模様のテクスチャを描画するフラグメントシェーダープログラム shader.frag

8.1.1　RGBA 構造体

テクスチャデータの基本要素は様々なデータ型として定義できるのだが、ここでは最も基本的なデータ型である RGB カラー値およびアルファ値からなる float 型 4 個の 32bit データ型を用いることとする。そのために、RGBA 構造体を、図 8.3 の以下の部分：

```
31 struct RGBA
32 {
33     float r;
34     float g;
35     float b;
36     float a;
37 };
```

のように定義する。

8.1.2　1 次元テクスチャクラス Texture1D

利便性の観点からテクスチャをクラス定義し、オブジェクト化する。

図 8.3 の 47 行目〜53 行目はクラス Texture1D の定義である。コンストラクタ：

```
52     Texture1D(int tnum, void* data, int w);       // コンストラクタ
```

はテスクチャオブジェクトを生成する。引数の意味は以下の通りである。

tnum ... 0 以上の**テクスチャ装置番号**を指定する。装置番号はプログラマが自由に与えてよいが、通常は 0 から順番に用いる。

data ... テクスチャの内容を RGBA データの 1 次元配列として指定する。詳細は後に述べる。

w ... data の配列サイズを指定する。

このコンストラクタの実装は図 8.5 である。処理内容がやや複雑であって、授業でそこまで理解する必要はないようにも思うが、一応、順に解説していく。

まず、代入文：

```
03     num = tnum;
```

は、プログラマが指定する装置番号を Texture1D オブジェクトのメンバー変数 num に保存する。この装置番号はテクスチャをシェーダー実行可能プログラムと結合するとき（関数 Shader::bindTexture() の内部処理）に必要である。

関数呼び出し：

```
04      glGenTextures (1, &texID);
```

は、このテクスチャを OpenGL コンテキストに新規登録する。システムによって bufID に自動的に参照番号が割り振られるが、その具体値を知る必要はない。

関数呼び出し：

```
05      glBindTexture(GL_TEXTURE_1D,texID);        // 参照番号の新規割付
```

は、参照番号が texID の一次元テクスチャ（GL_TEXTURE_1D）についてこれ以降、設定を行うことを宣言する。OpenGL では複数の一次元テクスチャについて同時に設定することはできない仕様になっている。これは複数のバッファオブジェクトを同時に設定できないこと（3.1 節参照）と同じ事情であり、OpenGL の仕様の古臭さである。

関数呼び出し：

```
06      glTexParameteri(GL_TEXTURE_1D, GL_GENERATE_MIPMAP,
07                      GL_FALSE);                  // ミップマップの設定
```

は、設定中の 1 次元テクスチャについて、ミップマップを自動生成（GL_GENERATE_MIPMAP）しない（GL_FALSE）ことを設定する。ミップマップとは、テクスチャをポリゴンに張り付ける際のアンチエイリアシング（描画像のジャギー、ギザギザ感を軽減する処理）の仕組みと考えてよい。ここではミップマップは用いないため、自動生成は行わない。ミップマップの詳細は省略する。

なお、関数 glTexParameteri() は、テクスチャの各種設定を行う関数である。

関数呼び出し：

```
08      glTexParameteri(GL_TEXTURE_1D, GL_TEXTURE_MAG_FILTER,
09                      GL_NEAREST);        // データの読み込み方法の設定
```

は、テクスチャから値を読み出すときに最近傍（GL_NEAREST）の位置の値を読み出すことを指定する。これについては後に詳しく述べる。

関数呼び出し：

```
10      glTexParameteri(GL_TEXTURE_1D, GL_TEXTURE_MIN_FILTER,
11                      GL_NEAREST);
```

は、ミップマップの縮小テクスチャから値を読み出すときの RGBA 値の読み出し方を指定する。上述のようにここではミップマップを用いない [1] ため、単に GL_NEAREST（または GL_LINEAR）を指定する。詳細は述べない。

コンストラクタ内の最後の関数呼び出し：

[1] ミップマップを用いる場合には第三引数に GL_NEAREST_MIPMAP_NEAREST、GL_LINEAR_MIPMAP_NEAREST などを与える。

```
12      glTexImage1D(GL_TEXTURE_1D, 0, GL_RGBA, w, 0, GL_RGBA,
13                   GL_FLOAT, data);  // テクスチャデータの GPU への転送
```

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。実はこの関数 `glTexImage1D()` 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 1 次元テクスチャ `GL_TEXTURE_1D` である。

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。ミップマップを用いない場合には 0 である。

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、OpenGL 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 8 引数で渡すデータ配列の大きさを指定する。ここでは `w` である。

第 5 引数 ... 仕様による固定値 0 を指定する。

第 6 引数 ... シェーダー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 7 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 8 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` の中身の作り方は後に `initData()` の説明の中で述べる。

なお、これらの引数のうち、第 3、6、7 引数の指定方法に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。

[2] 実は講義担当者も完全に理解している訳ではない。詳細は OpenGL の公式サイト `https://www.khronos.org/opengl/` などで調べること。

8.1.3　Shader クラスの拡張

作成された Texture1D オブジェクトは、テクスチャが実際に利用されるシェーダー実行可能プログラムと結合されねばならない。図 8.6 のメンバー関数

```
63      void bindTexture(const char* vname, Texture1D* tp);
```

がそのための関数である。ここに、第 1 引数がシェーダープログラム内でのテクスチャ名、第 2 引数がテクスチャオブジェクトのアドレスである。この関数は、ちょうど配列バッファとシェーダー実行可能プログラムを結合するメンバー関数

```
    void Shader::bindArrayBuffer(const char* vname, ArrayBuffer* ap)
```

と類似するように設計した。

図 8.6 がその実装である。

関数本体の 1 行目の関数呼び出し：

```
03      glBindTexture(GL_TEXTURE_1D, tp->texID);
```

は、これからこのテクスチャの設定を行うことの宣言である。この関数呼び出しは TextureID のコンストラクタでも行った。

テクスチャは uniform 変数として参照される。そこで代入文：

```
04      GLint p = glGetUniformLocation(program, vname);
```

は、バーテックスシェーダープログラムおよび（または）フラグメントシェーダープログラム中から vname と同じ変数名の uniform 変数を見つけ、その場所（location）を変数 p へ代入する。

もし p の値が負値ならば、適切な場所を見つけられなかったエラーであるから、メッセージを出力し、プログラム全体を強制終了する。

次に、関数呼び出し：

```
09      glUniform1i(p, tp->num);            // テクスチャに装置番号を設定
```

は、uniform 変数の場所にテクスチャの装置番号 tp->num を設定する。前章の float 型 uniform 変数とは異なり、場所にテクスチャデータを直接代入することはしない。

関数呼び出し：

```
10      glActiveTexture(GL_TEXTURE0+(tp->num));
```

は、指定した装置番号のテクスチャを GPU で利用できるようにアクティブにする。ここに GL_TEXTURE0 は実装依存のオフセット値である。

以上で、テクスチャ利用の準備ができた。

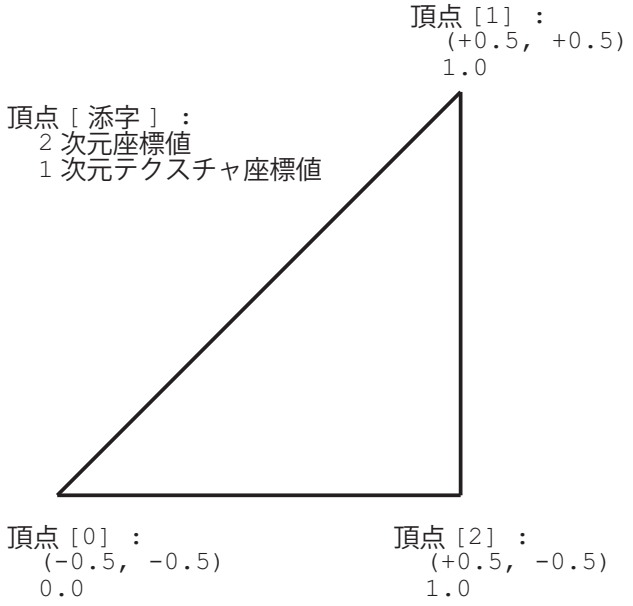

図 8.11: 三角形に 1 次元テクスチャを張る場合の動作例

8.1.4　ホストプログラム

図 8.7、図 8.8 が、クラス定義を除くホストプログラムである。関数 initData() を以下のように変更する。

頂点の座標値データは前章までと同様に設定する。今回の例題では頂点の RGB 値は設定しない。その代りに、3 頂点に対応する 1 次元テクスチャ座標を以下のように設定する。

```
36     float u[NUM_POINTS];                // テクスチャ座標のための配列
37     u[0] = 0.0;
38     u[1] = 1.0;
39     u[2] = 1.0;
40
41     ArrayBuffer ab3(u,1,NUM_POINTS); // 配列バッファオブジェクトの生成
42     sp->bindArrayBuffer("in_u",&ab3);       // attribute 変数との結合
```

なお、テクスチャ座標値は $[0, 1]$ の範囲で設定する（次章ではその条件を外すのだが)。三角形の頂点の 2 次元座標値とテクスチャ座標値の関係は図 8.11 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、テクスチャオブジェクトを生成し、シェーダープログラム中の uniform 変数 ”tex1” と結合する。

```
44 const int TEXTURE_SIZE = 16;                // 1 次元テクスチャサイズ
45     RGBA tex[TEXTURE_SIZE];              // テクスチャデータ用の配列
46     for(int i = 0; i < TEXTURE_SIZE; i++){
47         tex[i].r = i%2;
```

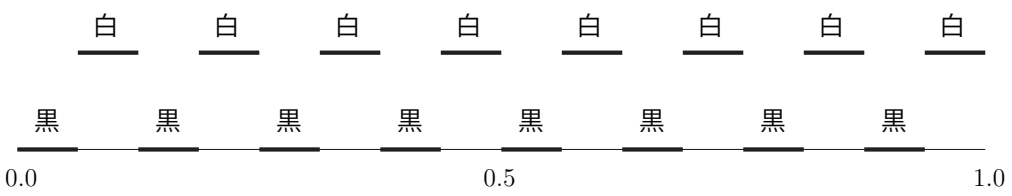

図 8.12: テクスチャ座標値とテクスチャデータ値の関係（その 1）

```
48          tex[i].g = i%2;                    // 0, 1, 0, 1, 0, 1, ...
49          tex[i].b = i%2;
50          tex[i].a = 0.0;
51      }
52      Texture1D t1(0, tex, TEXTURE_SIZE);
53      sp->bindTexture("tex1", &t1);   // テクスチャuniform 変数との結合
```

各 RGB 値は、 (0,0,0), (1,1,1), (0,0,0), (1,1,1), ... と設定されるから、つまり、白、黒、白、黒、... である。アルファ値は用いないから、常に 0 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 8.12 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、図 8.12 に示すように、その $[0, 1]$ の区間を 16 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ーで行う。

`initData()` 以外の関数での処理は、描画速度を 1 秒間に 2 コマ（2FPS）に変更した点を除けば、前章と同じである。

8.1.5　バーテックスシェーダープログラム

バーテックスシェーダープログラムは図 8.9 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。

8.1.6　フラグメントシェーダープログラム

フラグメントシェーダープログラムは図 8.10 の通りである。

バーテックスシェーダープログラムから出力された頂点のテクスチャ座標値はラスラタイザで各画素点について線形補間されてフラグメントシェーダーの varying 変数 `out_u` へ受け渡される。

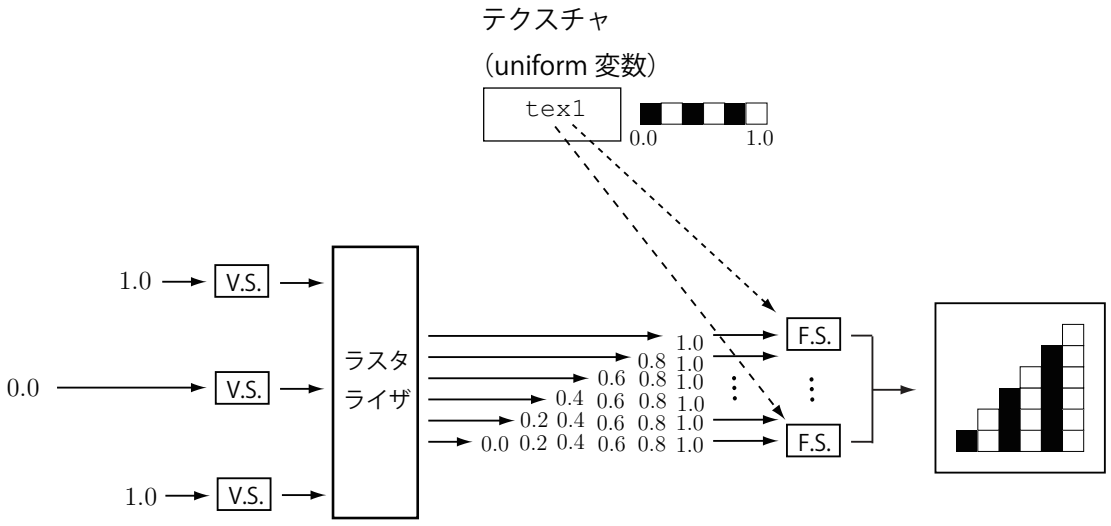

図 8.13: 三角形に 1 次元テクスチャを張る場合の動作例

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。また、1 次元テクスチャのデータ型は GLSL の組み込みデータ型 `sampler1D` と約束されている [3]。よってテクスチャデータの宣言は

```
05 uniform sampler1D tex1;          // 1 次元テクスチャ型 uniform 変数の宣言
```

となる。

画素の RGB 値の代入文：

```
09     gl_FragColor = texture1D(tex1, out_u);
```

の右辺 `texture1D(tex1, out_u)` は、テクスチャデータ `tex1` から座標値 `out_u` 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 8.12 の通りである。

8.1.7　実行結果

プログラムの実行の概念図が図 8.13 である。

配列バッファからバーテックスシェーダーの 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ーは、uniform 変

[3] より最近の OpenGL ではこの辺の仕様が激変しているので注意してほしい。この講義テキストではシェーダープログラミングの概念の理解および環境設定の手軽さを優先するため、あえて古いバージョンの OpenGL/GLSL を用いている。

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、それを画像バッファへ書き込む。

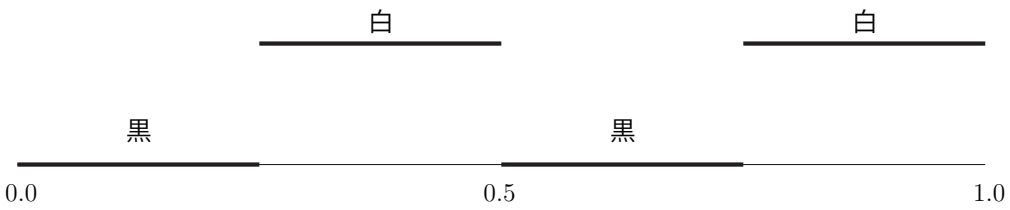

図 8.14: テクスチャ座標値とテクスチャデータ値の関係（その 2）

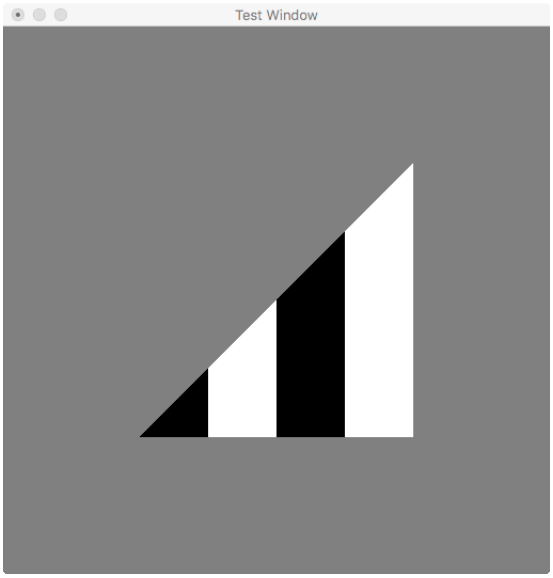

図 8.15: 画像例（その 10、テクスチャサイズの変更）

8.2　テクスチャデータサイズの変更

図 8.8 のプログラムのテクスチャオブジェクトを生成する部分；

```
44 const int TEXTURE_SIZE = 16;                 // 1 次元テクスチャサイズ
```

を

```
44 const int TEXTURE_SIZE = 4;                  // 1 次元テクスチャサイズ
```

へ変えてみる。つまりテクスチャデータの配列サイズを 16 から 4 へ変更する。これによって、座標値とその座標値の RGBA データ値の関係は図 8.14 のように代わり、結果、画像が図 8.15 のように変わる。

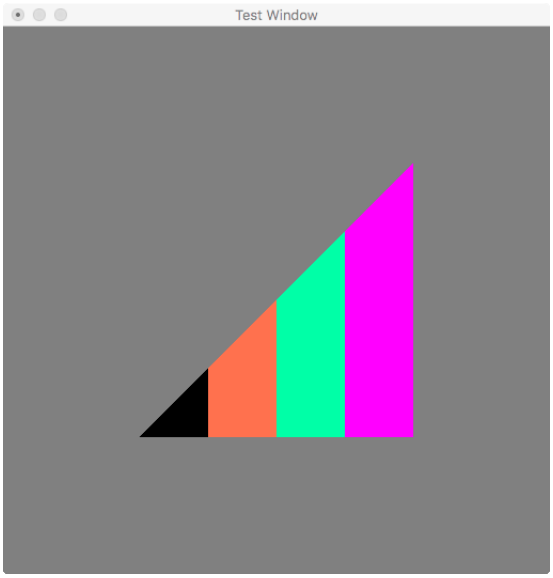

図 8.16: 画像例（その 11、カーラーテクスチャの利用）

8.3　カーラーデータの利用

次に、図 8.8 のプログラムの

```
46      for(int i = 0; i < TEXTURE_SIZE; i++){
47          tex[i].r = i%2;
48          tex[i].g = i%2;                    // 0, 1, 0, 1, 0, 1, ...
49          tex[i].b = i%2;
50          tex[i].a = 0.0;
51      }
```

を

```
46      for(int i = 0; i < TEXTURE_SIZE; i++){
47          tex[i].r = i%2;       // 0.0, 1.0,  0.0,  1.0, 0.0, 0.0, ...
48          tex[i].g = (i%3)/2.0; // 0.0, 0.5,  1.0,  0.0, 0.5, 1.0, ...
49          tex[i].b = (i%4)/3.0; // 0.0, 0.33, 0.67, 1.0, 0.0, 0.33, ..
50          tex[i].a = 0.0;
51      }
```

へ変更してみよう。図 8.16 の描画像が得られる。

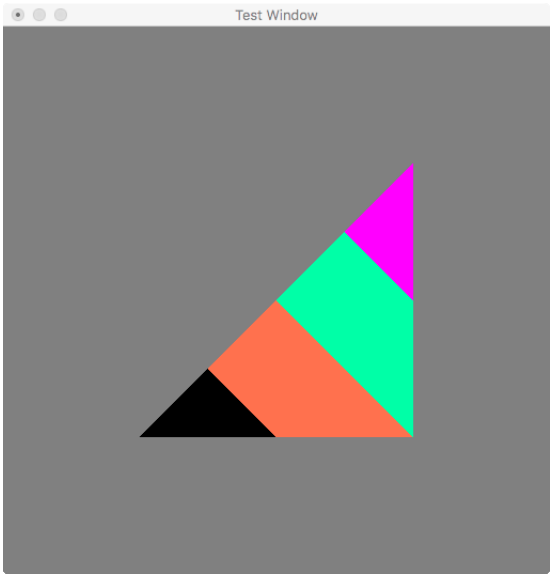

図 8.17: 画像例（その 12、テクスチャを傾斜させる変更）

8.4　テクスチャの傾斜

次に、図 8.7 のプログラムのテクスチャ座標値

```
37      u[0] = 0.0;
38      u[1] = 1.0;                          // 頂点への 1 次元座標値の代入
39      u[2] = 1.0;
```

の右辺を別の数値へ変更してみよう。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 8.17 のように変わる。三つの右辺の数値はどのようなものだろうか？

第9章　1次元テクスチャの高度な利用

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。

9.1　$[0,1]$ の範囲外のテクスチャ座標値：剰余計算

前章最後（8.4 節）の答えは、

```
37      u[0] = 0.0;
38      u[1] = 1.0;
39      u[2] = 0.5;
```

である。ここでこれを

```
37      u[0] = -1.0;        // ここを変更
38      u[1] =  2.0;        // ここを変更
39      u[2] =  0.5;
```

へ変更してみよう。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、右下がその中間値の 0.5 である。テクスチャ座標値として 0 よりも小さい値、1 よりも大きい値を含んでいる。この場合の描画像は図 9.1 である。

図 8.17 では（黒、橙、緑、桃）という色の帯が 1 回だけ現れるが、図 9.1 ではそれが 3 回繰り返している。この理由は、OpenGL/GLSL の暗黙の設定では 任意のテクスチャ座標値 $x \in [-\infty, +\infty]$ は剰余計算あるいは巡回計算：

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

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 [1]。x-x' のグラフを図示すると図 9.2 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 $[0,1]$ へ写像することは自然な発想である。

[1] ここでいう剰余計算とは、もし x が正数ならば x の小数点以下の部分を x' とみなし、x が 0 ならば $x' = 0$、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。

図 9.1: 画像例（その 13、テクスチャ座標値が $[0, 1]$ の範囲外の場合）

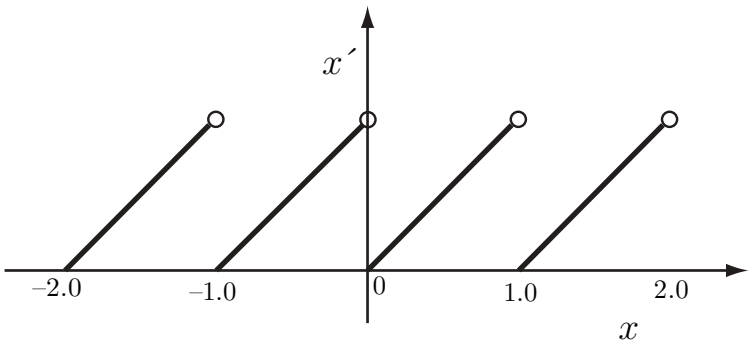

図 9.2: テクスチャ座標値の剰余計算

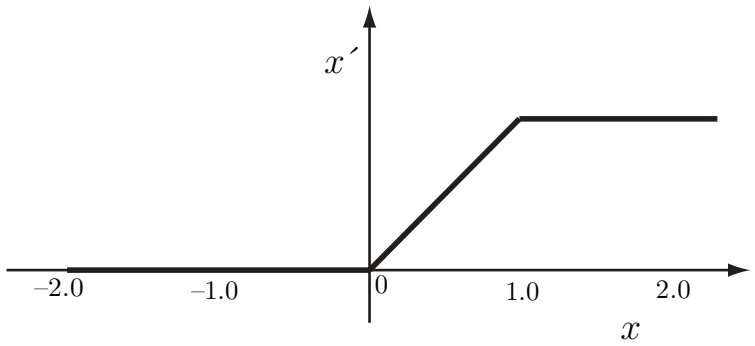

図 9.3: テクスチャ座標値の両端点固定

9.2　$[0, 1]$ の範囲外のテクスチャ座標値：端点固定

OpenGL/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。x-x' のグラフにすると、図 9.3 である。

これを行うには、図 8.5 のテクスチャクラス Texture1D のコンストラクタの

```
05      glBindTexture(GL_TEXTURE_1D,texID);        // 参照番号の新規割付
```

から

```
12      glTexImage1D(GL_TEXTURE_1D, 0, GL_RGBA, w, 0, GL_RGBA,
13                   GL_FLOAT, data);  // テクスチャデータの GPU への転送
```

の間に以下の関数呼び出しを加える。

```
        glTexParameteri(GL_TEXTURE_1D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
```

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_1D）の s 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の仮定されている。第 3 引数に繰り返し（GL_REPEAT）を指定している点に注意。

```
        glTexParameteri(GL_TEXTURE_1D, GL_TEXTURE_WRAP_S, GL_REPEAT);
```

これによって画像は図 9.4 のように変わる。

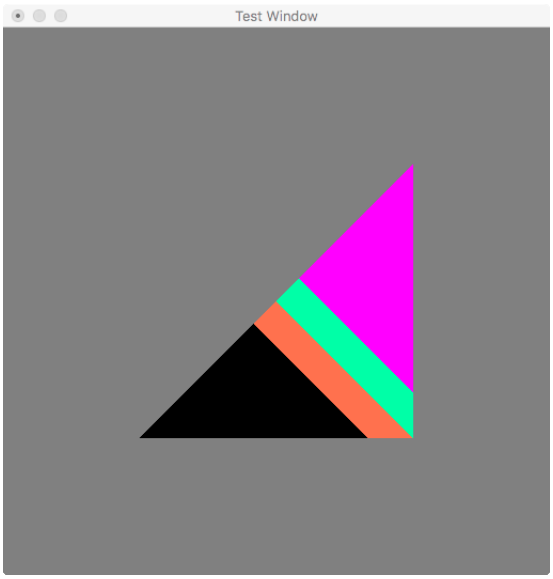

図 9.4: 画像例（その 14、範囲外のテクスチャ座標値を両端点の値にする）

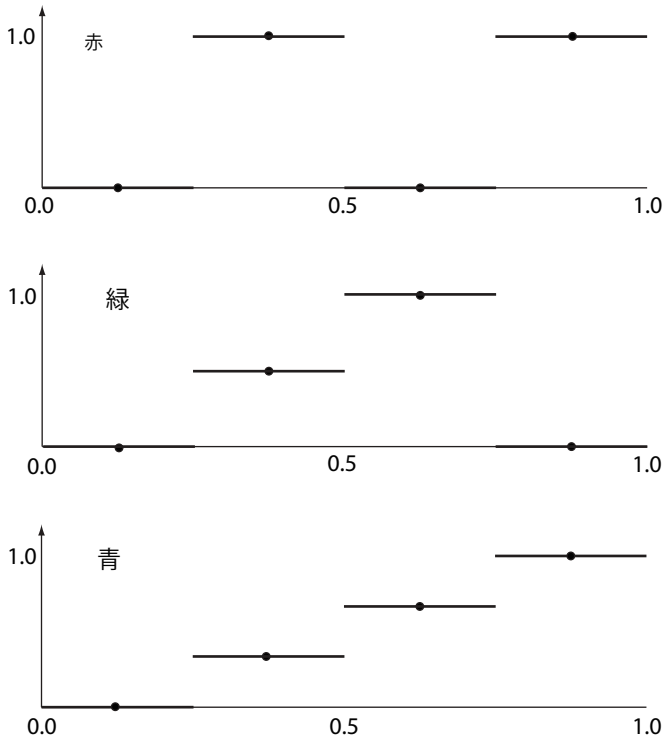

図 9.5: テクスチャ座標値とテクスチャデータ値の関係（その 3、最近傍の場合、上から赤、緑、青）

9.3　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テクスチャを滑らかに描画する機能が求められる。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。

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 9.5 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。たとえば座標値が 0.4 の場合、$(r, g, b) = (1.0, 0.5, 0.33)$ である。図中の黒丸 ● はその範囲の中心点を表す。

OpenGL/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 9.6 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している。

この機能を利用するには、図 8.5 のテクスチャクラス `Texture1D` のコンストラクタの中の以下の関数呼び出し：

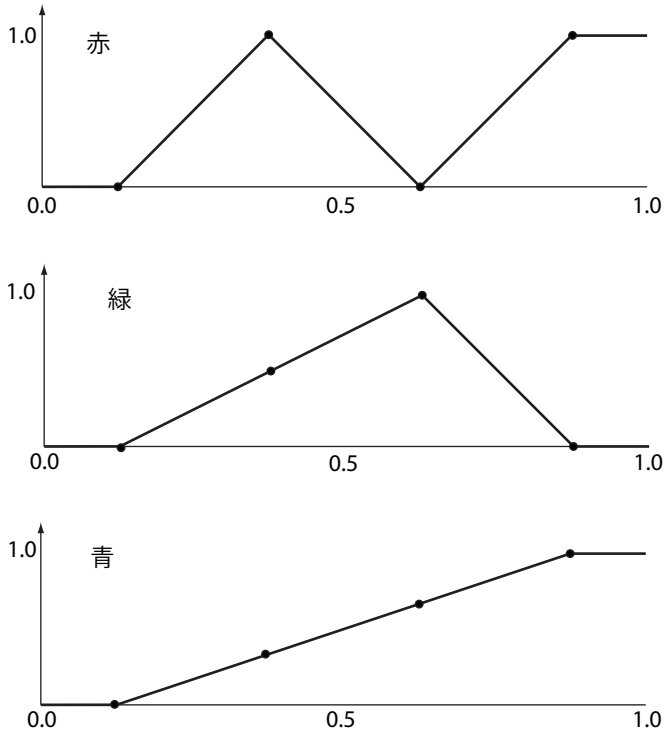

図 9.6: テクスチャ座標値とテクスチャデータ値の関係（その 4、線形補間の場合、上から赤、緑、青）

```
08     glTexParameteri(GL_TEXTURE_1D, GL_TEXTURE_MAG_FILTER,
09                     GL_NEAREST);        // データの読み込み方法の設定
```

の第 3 引数を以下のように変えてみる。つまり GL_NEAREST を GL_LINEAR へ変更する。

```
08     glTexParameteri(GL_TEXTURE_1D, GL_TEXTURE_MAG_FILTER,
09                     GL_LINEAR);        // データの読み込み方法の設定
```

これによって前節の画像（図 9.4）は図 9.7 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 9.7: 画像例（その 15、テクスチャデータを線形補間で求める）

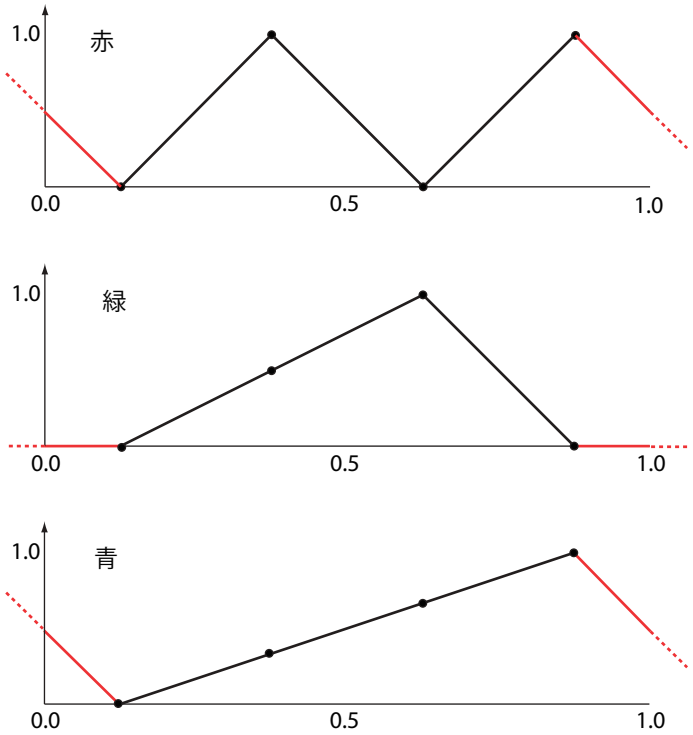

図 9.8: テクスチャ座標値とテクスチャデータ値の関係（その 5、剰余的線形補間の場合、上から赤、緑、青）

9.4　テクスチャデータの線形補間（その 2）

次に、9.2 節において導入した以下の文：

```
glTexParameteri(GL_TEXTURE_1D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
```

を削除（またはコメントアウト）する。または以下を加える。

```
glTexParameteri(GL_TEXTURE_1D, 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、図 9.8 のようなデータの読み込みとなる。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 9.9 の通りである。

図 9.9: 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）

第10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。

10.1　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10.1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。

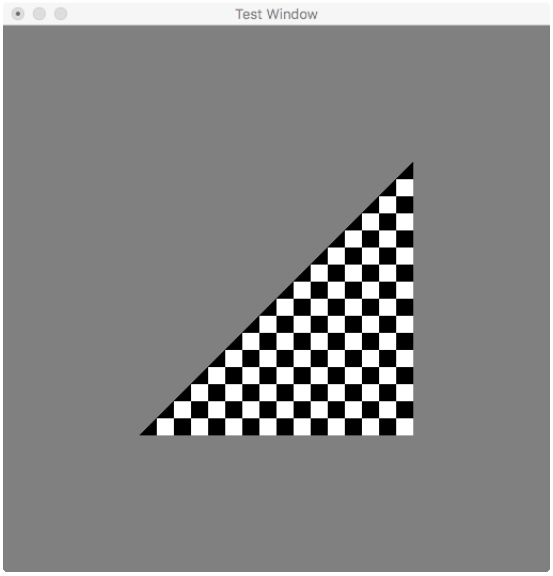

図 10.1: 画像例（その 17、2 次元テクスチャ利用の場合）

```
01 #include <iostream>
02 using namespace std;
03 #include "stdio.h"
04 #include "stdlib.h"
05
06 #if defined(WIN32)
07 #  pragma comment(lib, "glew32.lib")
08 #  include "glew.h"
09 #  include "glut.h"
10 #  include "glext.h"
11 #elif defined(__APPLE__) || defined(MACOSX)
12 #  include <GLUT/glut.h>
13 #else
14 #  define GL_GLEXT_PROTOTYPES
15 #  include <GL/glut.h>
16 #endif
17
18 struct Position2D
19 {
20     float x;
21     float y;
22 };
23
```

図 10.2: 各種クラスを定義するヘッダープログラム `All.h`（その 1、その 2 へ続く）

```
24 struct RGB
25 {
26     float r;
27     float g;
28     float b;
29 };
30
31 struct RGBA
32 {
33     float r;
34     float g;
35     float b;
36     float a;
37 };
38
39 struct ArrayBuffer
40 {
41     GLuint    bufID;
42     int        size;
43
44     ArrayBuffer(float* data, int s, int n);
45 };
46
47 struct Texture2D                              // 2 次元テクスチャクラス
48 {
49     GLuint  texID;                              // テクスチャの参照番号
50     GLint   num;                                // テクスチャの装置番号
51
52     Texture2D(int tnum, void* data, int w, int h);
                            // コンストラクタ。高さに関する引数 h が追加
53 };
54
55 struct Shader
56 {
57     GLuint        program;
58
59     Shader(const char* vsn, const char* fsn);
60     void use();
61
62     void bindArrayBuffer(const char* vname, ArrayBuffer* ap);
63     void bindTexture(const char* vname, Texture2D* tp);
                                          // シェダーとテクスチャの結合
64     void run(GLenum mode, int n);
65     void setFloat(const char* uname, float val);
66
67     GLuint compileProgram(GLenum type, const GLchar *file);
68     void buildProgram(const GLchar *vsfile, const GLchar *fsfile);
69 };
```

図 10.3: 各種クラスを定義するヘッダープログラム All.h（その 2）

89 ページの図 7.2 と同じ

図 10.4: ArrayBuffer クラスの実装プログラム Buffer.cpp

```
01 Texture2D::Texture2D(int tnum, void* data, int w, int h)
02 {
03     num = tnum;                                          // 前々章と同じ
04     glGenTextures (1, &texID);
05     glBindTexture(GL_TEXTURE_2D, texID);                 // 前々章と同様
06     glTexParameteri(GL_TEXTURE_2D, GL_GENERATE_MIPMAP,
07                     GL_FALSE);                           // 前々章と同様
08     glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
09                     GL_NEAREST);                         // 前々章と同様
10     glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
11                     GL_NEAREST);                         // 前々章と同様
12     glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, w, h, 0, GL_RGBA,
13                  GL_FLOAT, data);                        // 前々章と同様
14 }
```

図 10.5: 2 次元テクスチャクラス Texture2D のクラスの実装プログラム Textures.cpp

```
...
01 void Shader::bindTexture(const char* vname, Texture2D* tp)
02 {
03     glBindTexture(GL_TEXTURE_2D, tp->texID);         // 前々章と同様
04     GLint p = glGetUniformLocation(program, vname);// 前々章と同じ
05     if(p < 0) {                                      // 前々章と同じ
06         cerr << "texture1d name error: "<< vname << endl;
07         exit(1);
08     }
09     glUniform1i(p, tp->num);                         // 前々章と同じ
10     glActiveTexture(GL_TEXTURE0+(tp->num));          // 前々章と同じ
11 }
...
```

図 10.6: Shader クラスの実装プログラム Shader.cpp（その他の部分は、89 ページの図 7.3、90 ページの図 7.4 と同じ）

```
01 #include "All.h"
02
03 Shader *sp;
04
05 void initSystem(int argc, char *argv[])
06 {
07     glutInit(&argc,argv);
08     glutInitDisplayMode(GLUT_RGB | GLUT_DOUBLE);
09     glutInitWindowSize(512,512);
10     glutCreateWindow("Test Window");
11     glClearColor(0.5,0.5,0.5,1);
12
13 #if defined(WIN32)
14     glewInit();
15 #endif
16
17 //  glEnable(GL_POINT_SPRITE);
18 //  glEnable(GL_VERTEX_PROGRAM_POINT_SIZE);
19
20     sp = new Shader("shader.vert","shader.frag");
21 }
22
23 const int NUM_POINTS = 3;
24
25 void initData()
26 {
27     Position2D pos[NUM_POINTS];
28
29     pos[0].x = -0.5; pos[0].y = -0.5;
30     pos[1].x = +0.5; pos[1].y = +0.5;
31     pos[2].x = +0.5; pos[2].y = -0.5;
32
33     ArrayBuffer ab((float*)pos,2,NUM_POINTS);
34     sp->bindArrayBuffer("position",&ab);
35
36     Position2D uv[NUM_POINTS];          // テクスチャ座標のための配列
37     uv[0].x = 0.0; uv[0].y = 0.0;
38     uv[1].x = 1.0; uv[1].y = 1.0;       // 頂点への 2 次元テクスチャ
39     uv[2].x = 1.0; uv[2].y = 0.0;       // 座標値の設定
40
41     ArrayBuffer ab3((float*)uv,2 , NUM_POINTS);
                                        // 配列バッファオブジェクトの生成
42     sp->bindArrayBuffer("in_uv", &ab3);   // attribute 変数との結合
43
```

図 10.7: 一松模様のテクスチャを描画するホストプログラム main.cpp（その 1、その 2 へ続く）

```
44 const int TEXTURE_SIZE = 16;          // 2次元テクスチャデータのサイズ
45     RGBA tex[TEXTURE_SIZE][TEXTURE_SIZE]; // テクスチャデータ用の配列
46     for(int i = 0; i < TEXTURE_SIZE; i++)
47     {
48         for(int j = 0; j < TEXTURE_SIZE; j++)
49         {
50             tex[j][i].r = (i+j)%2;
51             tex[j][i].g = (i+j)%2;                  // 一松模様の計算
52             tex[j][i].b = (i+j)%2;
53             tex[j][i].a = 0.0;
54         }
55     }
56     Texture2D t2(0, tex, TEXTURE_SIZE, TEXTURE_SIZE);
                                              // テクスチャオブジェクトの生成
57     sp->bindTexture("tex2", &t2);   // テクスチャuniform変数との結合
58
59 //    glLineWidth(10.0);
60 }
61
62 double theta = 0.0;
63
64 void display(void)
65 {
66     sp->use();
67     sp->setFloat("theta", theta);
68     glClear(GL_COLOR_BUFFER_BIT);
69     sp->run(GL_TRIANGLES, NUM_POINTS);
70     theta += 0.02;
71 }
72
73 void timer(int num) {
74     glutPostRedisplay();
75     glutTimerFunc(500 , timer , num+1);
76 }
77
78 int main(int argc, char *argv[])
79 {
80    initSystem(argc,argv);
81     initData();
82
83     glutDisplayFunc(display);
84     glutTimerFunc(500 , timer , 1);
85     glutMainLoop();
86     return 0;
87 }
```

図 10.8: 一松模様のテクスチャを描画するホストプログラム main.cpp（その 2）

```
01 #version 120
02
03 attribute vec2 position;
04 attribute vec2 in_uv;                    // 2 次元テクスチャ座標値の入力
05
06 varying vec2 out_uv;                    // 2 次元テクスチャ座標値の出力
07
08 uniform float theta;
09
10 void main(void)
11 {
12     float x = cos(theta)*position.x-sin(theta)*position.y;
13     float y = sin(theta)*position.x+cos(theta)*position.y;
14     gl_Position = vec4(x, y, 0.0, 1.0);
15
16     out_u = in_u;              // 2 次元テクスチャ座標値を単純にコピー
17 }
```

図 10.9: 一松模様のテクスチャを描画するバーテックスシェーダープログラム shader.vert

```
01 #version 120
02
03 varying vec2 out_uv;                    // 2 次元テクスチャ座標値の入力
04
05 uniform sampler2D tex2;         // 2 次元テクスチャ型 uniform 変数の宣言
06
07 void main(void)
08 {
09     gl_FragColor = texture2D(tex2, out_uv);
                                           // テクスチャデータの読み込み
10 }
```

図 10.10:　一松模様のテクスチャを描画するフラグメントシェーダープログラム shader.frag

10.1.1　2 次元テクスチャクラス Texture2D

1 次元テクスチャクラス Texture1D と同様に定義する。Texture1D と Texture2D の違いは以下の点です。

まず、コンストラクタの引数が

```
52      Texture1D(int tnum, void* data, int w);       // コンストラクタ
```

から

```
52      Texture2D(int tnum, void* data, int w, int h);// コンストラクタ
```

へ、テクスチャデータの 2 次元目（高さ）のサイズを指定する引数が加わる。

コンストラクタの実装は図 8.5 から図 10.5 へ変更されるが、これもほとんど同じである。異なるのは、実引数のマクロ定数 GL_TEXTURE_1D を GL_TEXTURE_2D へ置換すること、コンストラクタ内の最後の関数呼び出し：

```
12      glTexImage1D(GL_TEXTURE_1D, 0, GL_RGBA, w, 0, GL_RGBA,
13                   GL_FLOAT, data);  // テクスチャデータの GPU への転送
```

を

```
12      glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, w, h, 0, GL_RGBA,
13                   GL_FLOAT, data);                    // 前々章と同様
```

へ置換することである。

10.1.2　Shader クラスの拡張

作成された Texture2D オブジェクトをシェーダー実行可能プログラムと結合するメンバー関数は図 10.2 の

```
63      void bindTexture(const char* vname, Texture2D* tp);
```

である。図 10.6 がその実装である。これも図 8.6 とほとんど同じである。異なるのは、実引数の GL_TEXTURE_1D を GL_TEXTURE_2D へ置換することだけである。

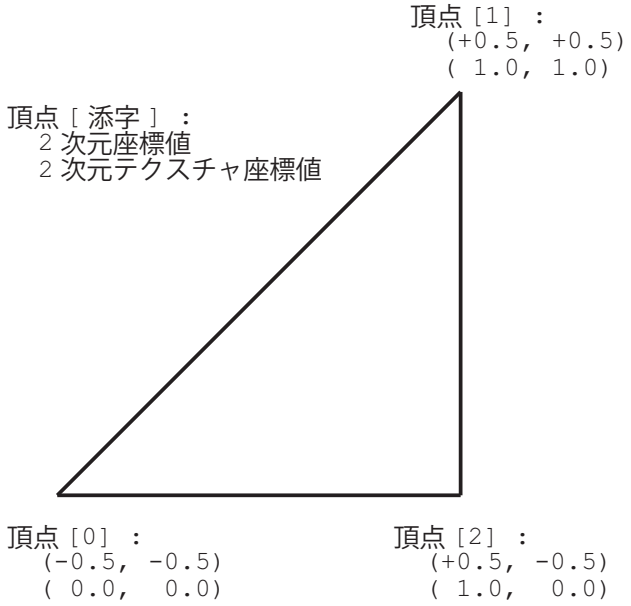

図 10.11: 三角形に 2 次元テクスチャを張る場合の動作例

10.1.3　ホストプログラム

図 10.7、図 10.8 が、クラス定義を除くホストプログラムである。関数 initData() を以下のように変更する。

頂点の座標値データは前章までと同様に設定する。2 次元テクスチャを用いるため、3 頂点に付随させるテクスチャ座標は 2 次元座標値である。そこでここではテクスチャデータとして Position2D オブジェクトの配列を用いる。

```
36     Position2D uv[NUM_POINTS];          // テクスチャ座標のための配列
37     uv[0].x = 0.0; uv[0].y = 0.0;
38     uv[1].x = 1.0; uv[1].y = 1.0;       // 頂点への 2 次元テクスチャ
39     uv[2].x = 1.0; uv[2].y = 0.0;       // 座標値の設定
40
41     ArrayBuffer ab3((float*)uv,2 , NUM_POINTS);
                                        // 配列バッファオブジェクトの生成
42     sp->bindArrayBuffer("in_uv", &ab3);   // attribute 変数との結合
```

テクスチャ座標値は上では $[0,1] \times [0,1]$ の範囲で設定しているが、前章で見たように範囲外の座標値でも問題ない。三角形の頂点の 2 次元座標値とテクスチャ座標値の関係は図 10.11 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。

```
44 const int TEXTURE_SIZE = 16;        // 2 次元テクスチャデータのサイズ
45     RGBA tex[TEXTURE_SIZE][TEXTURE_SIZE]; // テクスチャデータ用の配列
```

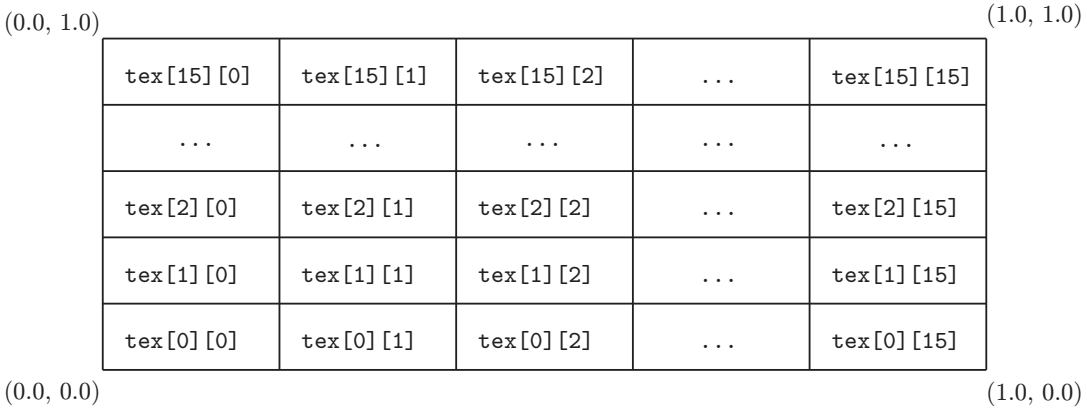

図 10.12: 2 次元テクスチャ空間におけるテクスチャデータの並び

```
46      for(int i = 0; i < TEXTURE_SIZE; i++)
47      {
48          for(int j = 0; j < TEXTURE_SIZE; j++)
49          {
50              tex[j][i].r = (i+j)%2;
51              tex[j][i].g = (i+j)%2;
52              tex[j][i].b = (i+j)%2;
53              tex[j][i].a = 0.0;
54          }
55      }
56      Texture2D t2(0, tex, TEXTURE_SIZE, TEXTURE_SIZE);
                                              // テクスチャオブジェクトの生成
57      sp->bindTexture("tex2", &t2);   // テクスチャuniform 変数との結合
```

なお、作成した各配列要素 tex[j][i] の 2 次元テクスチャ空間 $[0,1] \times [0,1]$ 上の対応位置は、図 10.12 のようになる。このことを踏まえてデータを作る必要がある。

10.1.4　バーテックスシェーダープログラム

バーテックスシェーダープログラムは図 10.9 の通りである。前章では float 型の 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、それをそのままラスタライザへ受け渡す。u と v はしばしばテクスチャ座標系で用いられる座標名である。

10.1.5　フラグメントシェーダープログラム

フラグメントシェーダープログラムは図 10.10 の通りである。

バーテックスシェーダープログラムから出力された頂点のテクスチャ座標値はラスラタイザで各画素点について線形補間されてフラグメントシェーダーの varying 変数 out_uv へ受け渡される。

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 と約束されている。よってテクスチャデータの宣言は

```
05 uniform sampler2D tex2;          // 2 次元テクスチャ型 uniform 変数の宣言
```

となる。

代入文：

```
09     gl_FragColor = texture2D(tex2, out_uv);
```

の右辺 texture2D(tex2, out_uv) は、テクスチャデータ tex2 から座標値 out_uv の位置の RGBA データを読み込む関数呼び出しである。

10.1.6　実行結果

プログラムの実行の概念図が図 10.13 である。

バッファからバーテックスシェーダーに入力される 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ーは、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、それをそのまま画像バッファへ書き込む。

次章以降では、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。

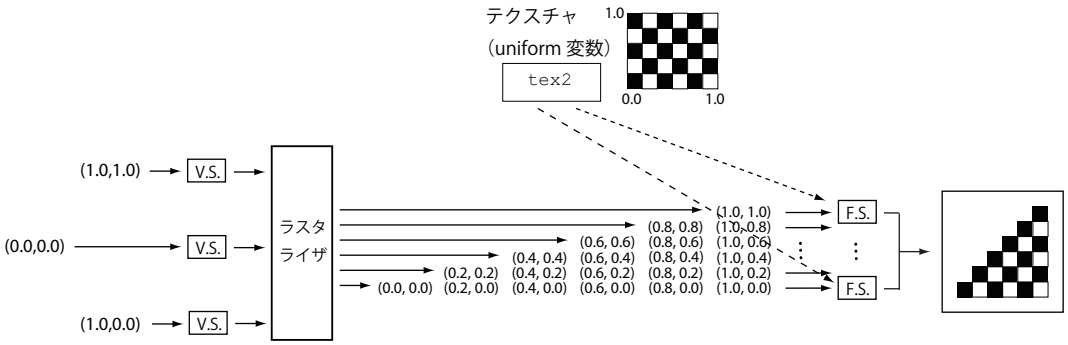

図 10.13: 三角形に 2 次元テクスチャを張る場合の動作例

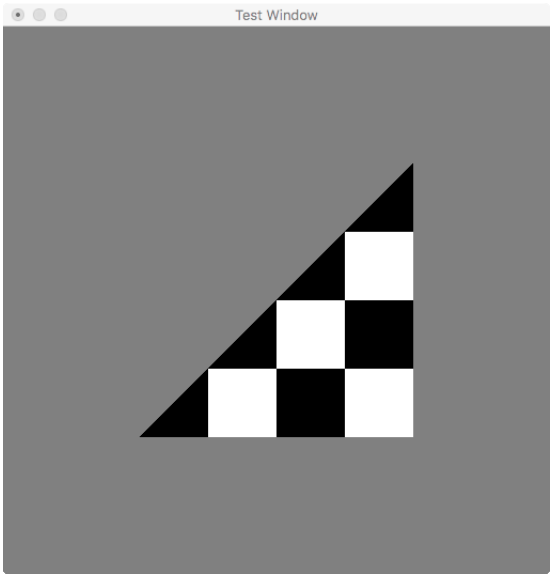

図 10.14: 画像例（その 18、テクスチャサイズの変更）

10.2　テクスチャデータサイズの変更

　図 10.8 のプログラムでは、テクスチャのサイズは
TEXTURE_SIZE × TEXTURE_SIZE = 16 × 16 であった。該当箇所：

```
44 const int TEXTURE_SIZE = 16;          // 2次元テクスチャデータのサイズ
```

を以下のように変えて、サイズを 4 × 4 へ変更すると、画像が図 10.14 のように変わる。

```
44 const int TEXTURE_SIZE = 4;          // 2次元テクスチャデータのサイズ
```

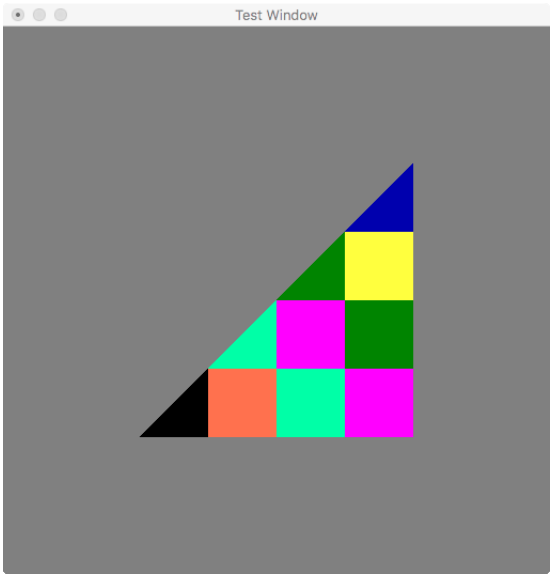

図 10.15: 画像例（その 19、カーラーテクスチャの利用）

10.3　カーラーデータの利用

次に、図 10.8 のテクスチャデータの設定部分：

```
46      for(int i = 0; i < TEXTURE_SIZE; i++)
47      {
48          for(int j = 0; j < TEXTURE_SIZE; j++)
49          {
50              tex[j][i].r = (i+j)%2;
51              tex[j][i].g = (i+j)%2;                  // 一松模様の計算
52              tex[j][i].b = (i+j)%2;
53              tex[j][i].a = 0.0;
54          }
55      }
```

を 8.3 節と同様に

```
46      for(int i = 0; i < TEXTURE_SIZE; i++)
47      {
48          for(int j = 0; j < TEXTURE_SIZE; j++)
49          {
50              tex[j][i].r = (i+j)%2;
51              tex[j][i].g = ((i+j)%3)/2.0;    // ここを変更
52              tex[j][i].b = ((i+j)%4)/3.0;    // ここを変更
53              tex[j][i].a = 0.0;
54          }
55      }
```

へ変更する。結果、図 10.15 の描画像が得られる。

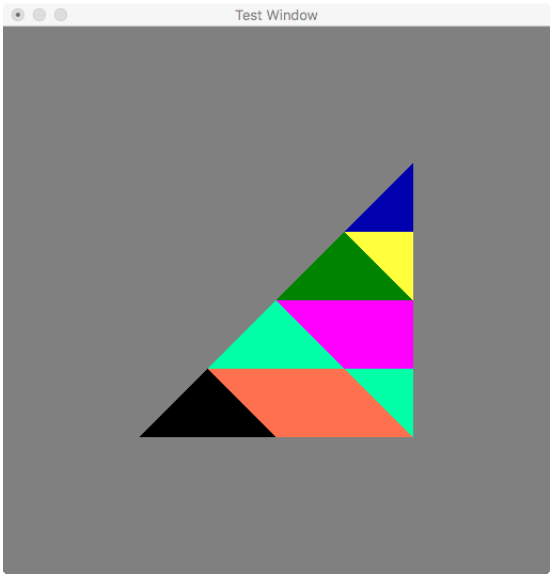

図 10.16: 画像例（その 20、テクスチャを傾斜させる変更）

10.4　テクスチャの傾斜

次に、図 10.7 のプログラムのテクスチャ座標値の設定：

```
37      uv[0].x = 0.0; uv[0].y = 0.0;
38      uv[1].x = 1.0; uv[1].y = 1.0;          // 頂点への 2 次元テクスチャ
39      uv[2].x = 1.0; uv[2].y = 0.0;          // 座標値の設定
```

を

```
37      uv[0].x = 0.0; uv[0].y = 0.0;
38      uv[1].x = 1.0; uv[1].y = 1.0;
39      uv[2].x = 0.5; uv[2].y = 0.0;     // uv[2].x を変更
```

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 10.16 のように変わる。

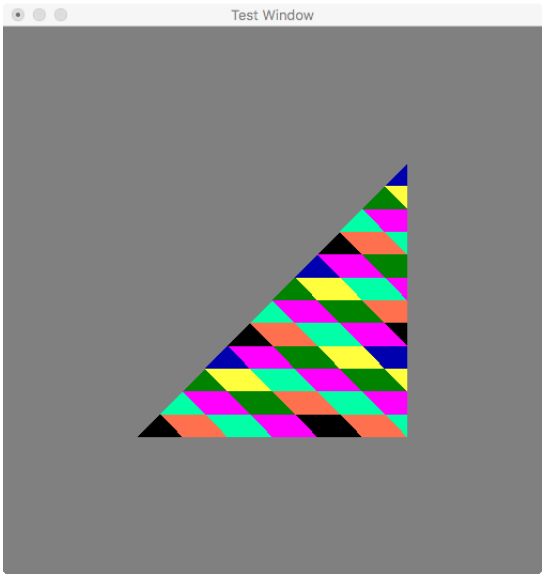

図 10.17: 画像例（その 21、テクスチャ座標値が $[0,1] \times [0,1]$ の範囲外の場合）

10.5　$[0,1]$ の範囲外のテクスチャ座標値：剰余計算

直前の変更

```
37      uv[0].x = 0.0; uv[0].y = 0.0;
38      uv[1].x = 1.0; uv[1].y = 1.0;
39      uv[2].x = 0.5; uv[2].y = 0.0;     // uv[2].x を変更
```

を

```
37      uv[0].x = -1.0; uv[0].y = -1.0;
38      uv[1].x =  2.0; uv[1].y =  2.0;
39      uv[2].x =  0.5; uv[2].y = -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 10.17 である。

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 $[0,1]$ の範囲に変換される（図 9.2 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。

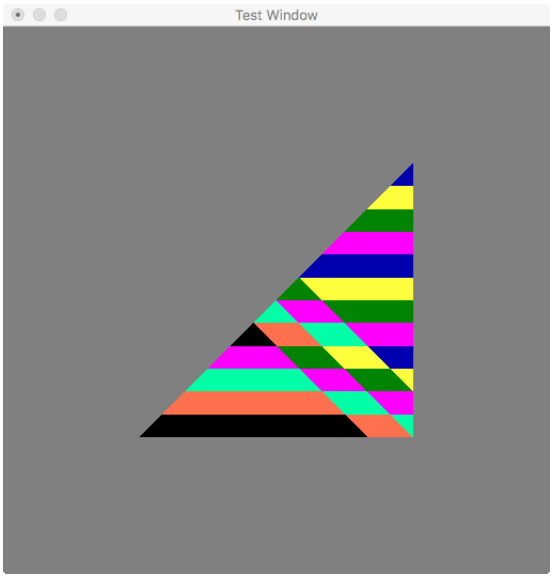

図 10.18: 画像例（その 22、範囲外のテクスチャ座標値を 1 次元目について両端点の値にする）

10.6　$[0, 1]$ 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 [1]。

端点固定を行うには、2 次元テクスチャのコンストラクタ（図 10.5）に以下を加える。

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S,
                GL_CLAMP_TO_EDGE);
```

これは、ただし、第 1 パラメータは 2 次元テクスチャであることを表す GL_TEXTURE_2D を指定する（GL_TEXTURE_1D ではない）。

これによって描画像は図 10.18 のように変わる。分かりにくいのだが、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 10.17 と図 10.18 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 次元目の座標に関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

[1] 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。

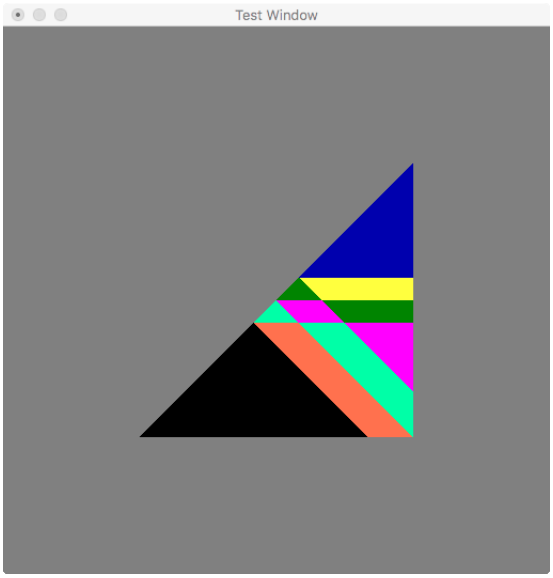

図 10.19: 画像例（その 23、範囲外のテクスチャ座標値を二つの次元とも両端点の値にする）

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 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 [2]。そこで、コンストラクタに

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T,
                GL_CLAMP_TO_EDGE);
```

を加えると、その結果は図 10.19 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。

[2]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。

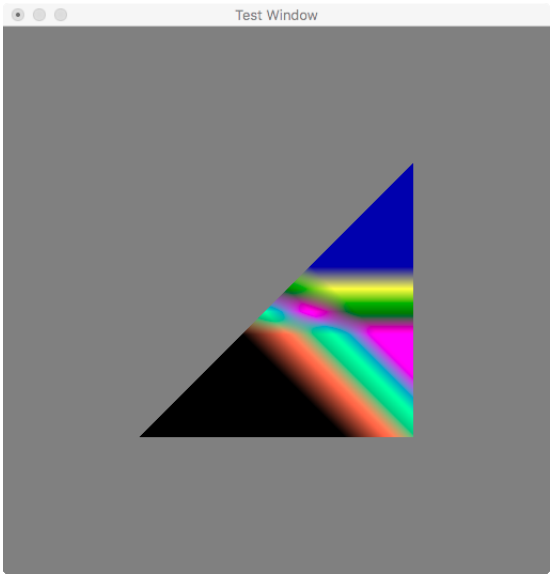

図 10.20: 画像例（その 24、テクスチャデータを線形補間で求める）

10.7　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、コンストラクタ（図 10.5）の中の設定：

```
08      glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
09                      GL_NEAREST);                    // 前々章と同様
```

の第 3 引数を以下のように変えてみる。

```
08      glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
09                      GL_LENEAR);
```

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 10.20 のように変わる。

これは 2 次元の線形補間であり、バイリニア補間（bilinear interpolation）とも呼ばれている方法である。すでに学部で講義で学んだ内容のはずだが、軽く復習しておく。

図 10.21 の正方形の 4 角の点 A、B、C、D に適当な値 V_A、V_B、V_C、V_D が与えられているものとする。その 4 角の値から、点 $P = (p_s, p_t)$（$0 \leq p_s, p_t \leq 1$）の点の値 V_P は以下のように補間される。

$$V_P = (1 - p_s) \cdot (1 - p_t) \cdot V_A + p_s \cdot (1 - p_t) \cdot V_B + (1 - p_s) \cdot p_t \cdot V_C + p_s \cdot p_t \cdot V_D$$

バイリニア補間を直感的に理解するために、式を変形すると

$$V_P = (1 - p_s) \left((1 - p_t) \cdot V_A + p_t \cdot V_C \right) + p_s \left((1 - p_t) \cdot V_B + p_t \cdot V_D \right)$$

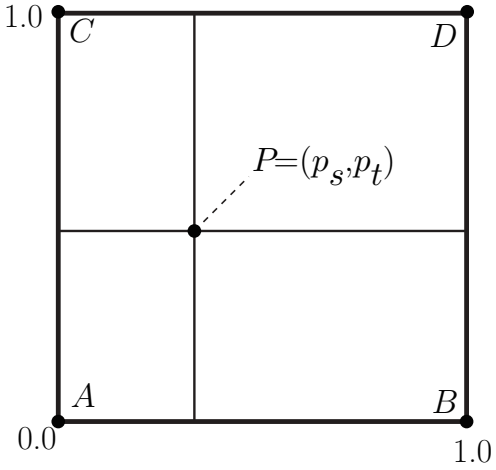

図 10.21: バイリニア補間の原理

となる。この場合、V_A と V_C について垂直方向の線形補間を行う。同様に V_B と V_D について垂直方向の線形補間を行う。その 2 回の線形補間の結果について水平方向の線形補間を行なって結果を得る。または

$$V_P = (1 - p_t)\left((1 - p_s) \cdot V_A + p_s \cdot V_B\right) + p_t\left((1 - p_s) \cdot V_C + p_s \cdot V_D\right)$$

の場合、水平方向の線形補間を 2 回、水平方向の線形補間を 1 回行う。

どの計算方法を用いても結果は同じである。

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。

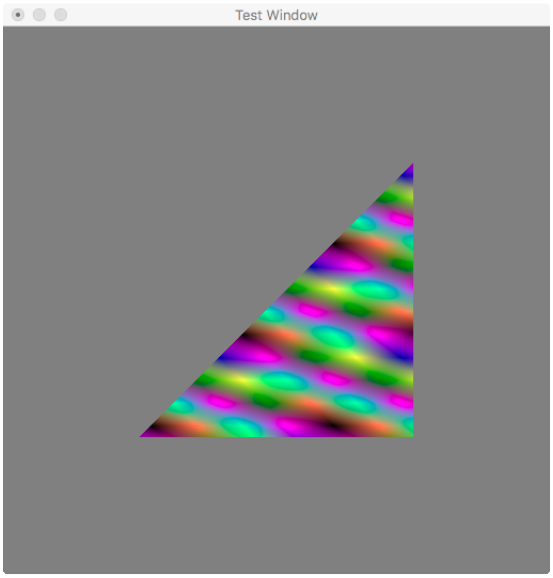

図 10.22: 画像例（その 25、巡回的なテクスチャデータを線形補間で求める）

10.8　テクスチャデータの線形補間（その 4）

次に、10.6 節において導入した以下の文：

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S,
                GL_CLAMP_TO_EDGE);
glTexParameteri(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

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S,
                GL_REPEAT);
glTexParameteri(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 10.22 を得る。

図 10.23: 元画像（理工学部 7 号館）

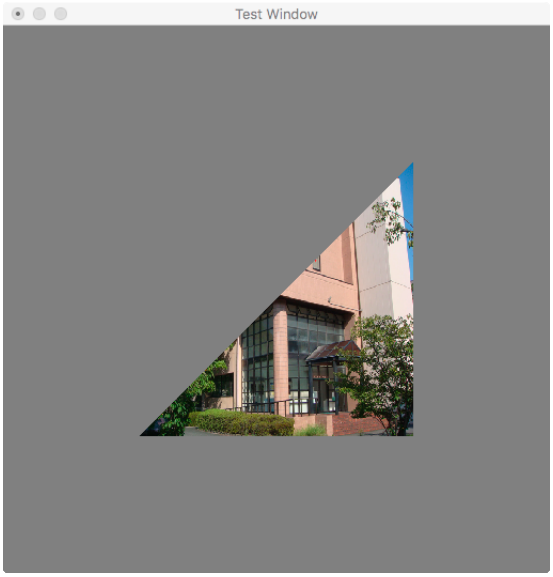

図 10.24: 画像例（その 26、理工学部 7 号館の画像を三角形に張る）

10.9　画像のマッピング

2 次元テクスチャマッピングを試すならば、やはり画像をマッピングしてみたい。

jpeg などの特定の画像フォーマットで格納されている画像をテクスチャとして利用するには、

1. その画像をプログラム中へ非圧縮の RGBA 型データの並びとして読み込み、
2. それをテクスチャデータとして GPU へ格納する。ここは前節までと同じ。

たとえば、図 10.24 は、理工学部 7 号館の画像（図 10.23）をその方法で取り込み、描画したものである。具体的には、図 10.8 の以下のプログラム片：

```
    RGBA tex[16][16];              // テクスチャデータの設定
    for(int i = 0; i < 16; i++){
        for(int j = 0; j < 16; j++){
            tex[j][i].r = (i+j)%2;
            tex[j][i].g = (i+j)%2;
            tex[j][i].b = (i+j)%2;
            tex[j][i].a = 0.0;
        }
    }
    Texture2D t2(0,tex,16,16);
    sp->bindTexture("tex2",&t2);
}
```

の部分を

```
    // 画像ファイルおよび画像の水平、垂直サイズの読み込み
    int imgw,imgh;
    RGBA* img = loadBMP("b7.bmp", imgw, imgh);

    Texture2D t2(0, img, imgw, imgh);
    sp->bindTexture("tex2",&t2);
}
```

などに置き換えればよい。ここに `loadBMP()` は講義担当者が作った bmp 画像を読み込むプログラム（図 10.25、図 10.26）である。bmp 画像は非常に単純な形式であるから、読み込みプログラムの自作は容易である。jpeg 画像の読み込みプログラムは片手間で自作できるほど簡単ではないが、そのような関数はインターネット上に多数、公開されているから、それを用いればよい。

　分かりにくいが、図 10.24 の 7 号館は図 10.23 の元画像の 7 号館よりもスリムに描画されている。その理由は、元画像は 600×459 のやや横長の画像なのだが、テクスチャ画像は縦横 1:1 の正方形で扱われるためである。

```
#include "All.h"

RGBA* loadBMP(char *filename, int &ww, int& hh) {
    FILE *file = fopen(filename, "rb");
    if(file == NULL)
    {
        fprintf(stderr,"BMP error 0: %s\n",filename);
        exit(1);
    }

    char    tmp1;
    short   tmp2;
    int     tmp4;
    RGBA* teximg;

    fread(&tmp1,1,1,file);
    if(tmp1 != 'B')
    {
        fprintf(stderr,"BMP error 1\n");
        exit(1);
    }
    fread(&tmp1,1,1,file);
    if(tmp1 != 'M')
    {
        fprintf(stderr,"BMP error 2\n");
        exit(1);
    }
    fread(&tmp4,4,1,file);
    fread(&tmp2,2,1,file);
    fread(&tmp2,2,1,file);
    fread(&tmp4,4,1,file);

    fread(&tmp4,4,1,file);
    switch(tmp4){
        case 40:
        {
            fread(&tmp4,4,1,file);
            int texwidth = ww = tmp4;
            fread(&tmp4,4,1,file);
            int texheight = hh = tmp4;
            fread(&tmp2,2,1,file);
            if(tmp2 != 1)
            {
                fprintf(stderr,"BMP error 2.2\n");
                exit(1);
            }
            fread(&tmp2,2,1,file);
            if(tmp2 != 24)
            {
```

図 10.25: BMP ファイルの読み込み bmp.cpp（その 1、その 2 へ続く）

```
                fprintf(stderr,"BMP error 2.3\n");
                exit(1);
            }
            fread(&tmp4,4,1,file);
            if(tmp4 != 0)
            {
                fprintf(stderr,"BMP error 2.4\n");
                exit(1);
            }
            fread(&tmp4,4,1,file);
            fread(&tmp4,4,1,file);
            fread(&tmp4,4,1,file);
            fread(&tmp4,4,1,file);
            fread(&tmp4,4,1,file);

            teximg = new RGBA[texwidth*texheight];
            if(teximg == NULL)
            {
                fprintf(stderr,"BMP new error1\n");
                exit(1);
            }

            int s = 3*texwidth;
            if(s%4 != 0) s += 4-s%4;
            unsigned char *buf = new unsigned char[s];
            for(int y = 0; y < texheight; y++)
            {
                fread(buf,s,1,file);
                for(int x = 0; x < texwidth; x++)
                {
                    int i = (x+y*texwidth);
                    teximg[i].r = float(buf[3*x+2])/256.0;
                    teximg[i].g = float(buf[3*x+1])/256.0;
                    teximg[i].b = float(buf[3*x  ])/256.0;
                    teximg[i].a = 0.0;
                }
            }
        }
        break;
        default:
        {
            fprintf(stderr,"BMP error 3\n");
            exit(1);
        }
    }
    fclose(file);
    return teximg;
}
```

図 10.26: BMP ファイルの読み込み bmp.cpp（その 2）

第11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ーおよび多数のフラグメントシェーダーが並列実行する。しかもシェーダー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。

この章以降で述べるテクスチャとフラグメントシェーダーを用いる並列計算はその自由度が高く、CUDAやOpenCLを用いた並列計算の先駆けになった。

テクスチャは配列のように扱うことができる。特にテクスチャからRGBA値を取得する（サンプリングする）方法を、線形補間（GL_LINEAR）ではなく、最近傍（GL_NEAREST）に設定する場合には、テクスチャ内に格納された実際のデータをそのまま読むことができるため、配列要素を読むイメージに非常に近い。前章のテクスチャは読み出し専用であったが、**もし何らかの方法でデータをテクスチャへ書き込むことができるならば**、配列要素データを読み込み、計算を行い、結果を配列要素へ書き込む一連の操作が可能になる。

結果、2次元配列上のデータ並列演算ができるだろうことは想像に難くない。実際、可能である。

OpenGLでは、高度な3D CGを可能するため — たとえば、鏡のあるシーンの3D CG描画や影のある3D CGを行うため — にテクスチャへのデータの書き込みが導入された。これを流用すれば任意のデータ並列演算ができる。その演算性能が圧倒的であったため、研究者らの注目を集め、その後、CUDAやOpenCLなどのGPGPU専用のプログラミング環境が開発されていく。

この章ではGPGPU草創期のOpenGLによるデータ並列計算の方法を紹介する。一見、古臭い内容とも言えるが、GPUを用いた高速計算の原点を知ることができる。また最近ではOpenCLのような並列計算環境ではなく、シェーダーでの並列計算が見直される原点回帰の動きもある。

この章のタイトルには「簡単な計算」と付けた。実際、計算内容は非常に簡単なものだが、それをGPUで実現するOpenGLの設定はそう簡単ではない。

```
#include "All.h"

void initSystem()
{
    /* ここにシステムパラメータの設定等の初期化処理 */
}

void initData()
{
    /* ここに入力データなどの初期化処理 */
}

void compute()                          // display() に相当
{
    /* ここに GPU 実行処理 */
}

void showResults()                  // 新たに導入
{
    /* ここに結果の出力等の処理 */
}

int main(int argc, char *argv[])
{
    initSystem(argc,argv);
    initData();
    compute();
    showResults();
    return 0;
}
```

図 11.1: GPGPU のためのホストプログラムのひな形

11.1　プログラムの外形の変更

これ以降は画像描画を行わないため、2 章の図 2.3 で設定したプログラムの全体構成を図 11.1 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の関数である。

`compute()` は、これまでの `display()` と同様の処理を行うが、描画を行わないため、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。

ここまで用いてきた `glutDisplayFunc()`、`glutMainLoop()` は描画専用の関数であるため、これ以降は用いない。

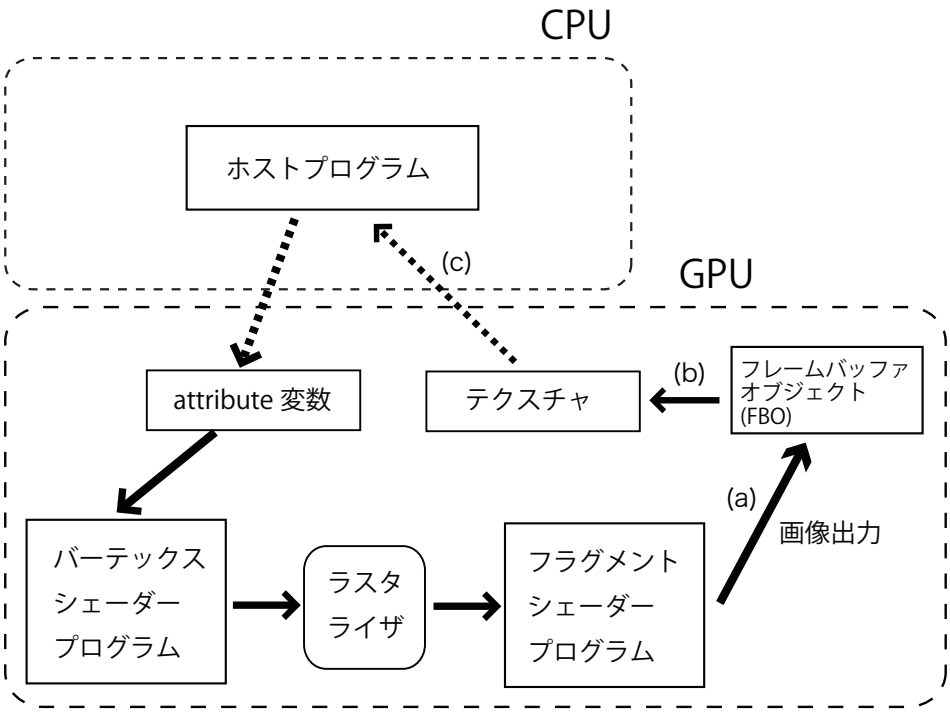

図 11.2: テクスチャへデータを代入する経路

11.2　テクスチャへの代入

この節では計算は行わず、テクスチャへデータを書き込めることを簡単なサンプルプログラムで確認する。

図 11.2 はテクスチャへデータを書き込む仕組みの概念図である。

OpenGL ではフレームバッファを仮想化/抽象化した機能を有する**フレームバッファオブジェクト**（以下、FBO）が利用できる。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。
2. フラグメントシェーダーの出力を FBO に接続する（図 11.2 の (a)）
3. FBO をテクスチャと接続する（図 11.2 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ーの計算した各画素の RGBA 値は、本来の画像バッファへは格納されず、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 11.2 の (c)）、ホストプログラムでその内容を確認する。

以下にこの動作をするプログラムを示す。これまで同様に、プログラムの後ろにコメントの付いている箇所のみが前章からの変更点である。

```
01 #include <iostream>
02 using namespace std;
03 #include <stdio.h>
04 #include <stdlib.h>
05
06 #if defined(WIN32)
07 #  pragma comment(lib, "glew32.lib")
08 #  include "glew.h"
09 #  include "glut.h"
10 #  include "glext.h"
11 #elif defined(__APPLE__) || defined(MACOSX)
12 #  include <GLUT/glut.h>
13 #else
14 #  define GL_GLEXT_PROTOTYPES
15 #  include <GL/glut.h>
16 #endif
17
18 struct Position2D
19 {
20     float x;
21     float y;
22 };
23
24 struct RGBA
25 {
26     float r;
27    float g;
28     float b;
29    float a;
30 };
31
32 struct ArrayBuffer
33 {
34    GLuint bufID;
35     int    size;
36
37     ArrayBuffer(float* data, int s, int n);
38 };
39
40 struct RWTexture2D              // 読み書き兼用 2 次元テクスチャクラス
41 {
42     GLuint  texID;                                          // 参照番号
43     GLint   num;                                            // 装置番号
44
45     RWTexture2D(int tnum, void* data, int w, int h);
                                                        // コンストラクタ
46     void readData(void* data, int w, int h);
                                        // GPU からホストへのデータの転送
47 };
```

図 11.3: 各種クラスを定義するヘッダープログラム All.h（その 1、その 2 へ続く）

```
48
49 struct Shader
50 {
51     GLuint program;
52
53     Shader(const char* vsn, const char* fsn);
54     void use();
55
56     void bindArrayBuffer(const char* vname, ArrayBuffer* ap);
57     void setFloat(const char* vname, float val);
58
59     void bindTextureR(const char* vname, RWTexture2D* tp);
        // 読み書き兼用 2 次元テクスチャを読み込み用としてシェーダーと結合
60     void bindTextureW(RWTexture2D* tp);
        // 読み書き兼用 2 次元テクスチャを書き込み用としてシェーダーと結合
61
62     void run(GLenum mode, int n);
63
64     GLuint compileProgram(GLenum type, const GLchar *file);
65     void buildProgram(const GLchar *vsfile, const GLchar *fsfile);
66 };
```

図 11.4: 各種クラスを定義するヘッダープログラム All.h（その 2）

89 ページの図 7.2 と同じ

図 11.5: ArrayBuffer クラスの実装プログラム Buffer.cpp

```
01 #include "All.h"
02
03 RWTexture2D::RWTexture2D(int tnum, void* data, int w, int h)
04 {                      //読み書き兼用 2 次元テクスチャのコンストラクタ
05     num = tnum;                    // 05 行〜15 行は前章までと全く同じ
06     glGenTextures (1, &texID);
07     glBindTexture(GL_TEXTURE_2D,texID);
08     glTexParameteri(GL_TEXTURE_2D, GL_GENERATE_MIPMAP,
09                     GL_FALSE);
10     glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
11                     GL_NEAREST);
12     glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
13                     GL_NEAREST);
14     glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, w, h, 0, GL_RGBA,
15                  GL_FLOAT, data);
16     glFramebufferTexture2D(GL_FRAMEBUFFER,
17                            GL_COLOR_ATTACHMENT0+(num),
18                            GL_TEXTURE_2D, texID, 0);
                      // フレームバッファからテクスチャへの出力経路の設定
19 }
20
21 void RWTexture2D::readData(void* data, int w, int h)
22 {                          // GPU からホストへのテクスチャデータの転送
23     glReadBuffer(GL_COLOR_ATTACHMENT0+(num));
                                                  // 転送元のテクスチャの指定
24     glReadPixels(0, 0, w, h, GL_RGBA, GL_FLOAT, data);// 転送の実施
25 }
```

図 11.6: 読み書き可能 2 次元テクスチャクラス RWTexture2D のクラスの実装プログラム Textures.cpp

```
01 ...
02 void Shader::run(GLenum mode, int n)
03 {
04     glDrawArrays(mode, 0, n);
05     glFlush();                              // glutSwapBuffer() を置換
06 }
07
08 void Shader::bindTextureR(const char* vname, RWTexture2D* tp)
09 {                     //引数のテクスチャを vname の uniform 変数に結合
10     glActiveTexture(GL_TEXTURE0+(tp->num));                  // 注 1
11     glBindTexture(GL_TEXTURE_2D,tp->texID);
12     GLint p = glGetUniformLocation(program, vname);
13     if(p < 0) {
14         cerr << "texture2d name error: "<< vname << endl;
15         exit(1);
16     }
17     glUniform1i(p, tp->num);
18 }
19
20 void Shader::bindTextureW(RWTexture2D* tp)
21 {                                  // 引数のテクスチャを出力先として指定
22     glDrawBuffer(GL_COLOR_ATTACHMENT0+(tp->num));
23 }
24 ...
```

図 11.7: Shader クラスの実装プログラム Shader.cpp（その他の部分は、89 ページの図 7.3、90 ページの図 7.4 と同じ）（注 1 については本文中で注釈する）

```
01 #include "All.h"
02
03 const int width = 5;                    //画面の幅＝テクスチャの幅を定数
04 const int height = 7;                   //画面の高さ＝テクスチャの高さ
05
06 Shader *sp;
07
08 void initSystem(int argc, char *argv[])
09 {
10     glutInit(&argc,argv);
11     glutInitDisplayMode(GLUT_RGB | GLUT_SINGLE);
12     glutInitWindowSize(width, height);     // ウインドウサイズの設定
13     glutCreateWindow("Test Window");
14     glClearColor(0.0, 0.0, 0.0, 0.0);
                                      // フレームバッファの 0 クリアを設定
15
16 #if defined(WIN32)
17     glewInit();
18 #endif
19
20     GLuint fb;
21     glGenFramebuffers(1, &fb);     // FBO を OpenGL コンテキストに登録
22     glBindFramebuffer(GL_FRAMEBUFFER, fb);
                                        // FBO をフレームバッファに結合
23
24     sp = new Shader("shader.vert","shader.frag");
25     sp->use();                               // シェーダーの使用の宣言
26 }
27
28 const int NUM_POINTS = 3;
29
30 RWTexture2D *texZp;            // 書き込み用 2 次元テクスチャのポインタ
31
32 void initData()
33 {
34     Position2D pos[NUM_POINTS];
35     pos[0].x = -0.5; pos[0].y = -0.5;        // 前章までと同じ形状の
36     pos[1].x = +0.5; pos[1].y = +0.5;        // 三角形ポリゴン
37     pos[2].x = +0.5; pos[2].y = -0.5;
38
39     ArrayBuffer ab((float*)pos, 2, NUM_POINTS);
40     sp->bindArrayBuffer("position", &ab);
41
42     texZp = new RWTexture2D(0, NULL, width, height);
                  // データを与えず（NULL)、読み書き兼用テクスチャを生成
43     sp->bindTextureW(texZp);          // テクスチャを書き込み用に設定
44 }
```

図 11.8: テクスチャへの代入を行うホストプログラム main.cpp（その 1、その 2 へ続く）

```
45
46 void compute(void)
47 {
48     glClear(GL_COLOR_BUFFER_BIT);       // フレームバッファの 0 クリア
49     sp->run(GL_TRIANGLES, NUM_POINTS);          // シェーダーの実行
50 }
51
52 void showResults()
53 {
54     glFinish();                  // シェーダープログラムの実行終了待ち
55
56     RGBA results[height][width];          // 転送先のメモリ領域を確保
57     texZp->readData(results, width, height);
                                // テクスチャの内容を results へ読み込み
58     for(int h = height-1; h >= 0; h--)                // 上から順に
59     {
60         for(int w = 0; w < width; w++)                // 左から順に
61         {
62             printf("(%3.1f,%3.1f) ",    // 読み込んだ個々の内容の表示
63                    results[h][w].r, results[h][w].g);
64         }
65         printf("\n");
66     }
67 }
68
69 int main(int argc, char *argv[])
70 {
71     initSystem(argc,argv);                // システムを初期化し、
72     initData();                           // データを初期化し、
73     compute();                            // 計算を実行し、
74     showResults();                        // 計算結果を表示する
75     return 0;
76 }
```

図 11.9: テクスチャへの代入を行うホストプログラム main.cpp（その 2）

```
01 #version 120
02
03 attribute vec2 position;
04
05 void main(void)
06 {
07     gl_Position = vec4(position, 0.0, 1.0);
                                  // 2 次元座標値をそのままラスタライザへ
08 }
```

図 11.10: テクスチャへの代入を行うバーテックスシェーダープログラム shader.vert

```
01 #version 120
02
03 void main(void)
04 {
05     gl_FragColor = gl_FragCoord;
                    // 画素位置をそのままフレームバッファ（＝書き込み用
                    // テクスチャ）へ書き込む
06 }
```

図 11.11: テクスチャへの代入を行うフラグメントシェーダープログラム shader.frag

11.2.1　読み書き兼用 2 次元テクスチャクラス RWTexture2D

テクスチャの取り扱いをできるだけ簡単にするため、読み書き兼用 2 次元テクスチャクラス RWTexture2D を定義し、テクスチャの処理の詳細をオブジェクト内にカプセル化する。

このテクスチャは読み込み専用または書き込み専用として機能するが、GPU の構造上、同時に読み込みかつ書き込みはできない。読み込み専用としてプログラムを実行した後に、書き込み専用に機能を切り替えることができる。そのような使い方は次章で行う。

読み込み機能の実装は前章の方法をほぼそのまま流用できる。ここでは主に書き込み機能を追加実装する。

まず、クラスの定義は図 11.3 のように設計する。コストラクタ：

```
45      RWTexture2D(int tnum, void* data, int w, int h);
```

の仕様は前章の 2 次元テクスチャと同じである。前章との違いはメンバー関数 readData() が新規に追加された点である。この関数：

```
46      void readData(void* data, int w, int h);
```

は、GPU 上のテクスチャに格納されているデータをホスト側の配列 data へ転送する。w、h はテクスチャの幅、高さである。

これらの実装は図 11.6 の通りである。

コンストラクタの実装は、最後の関数呼び出しを除いて前章の 2 次元テクスチャと全く同じである。最後の関数呼び出し：

```
16      glFramebufferTexture2D(GL_FRAMEBUFFER,
17                             GL_COLOR_ATTACHMENT0+(num),
18                             GL_TEXTURE_2D, texID, 0);
```

は、フレームバッファ（GL_FRAMEBUFFER）から num 番目の出力経路 [1] を介して参照番号 texID の 2 次元テクスチャ（GL_TEXTUR_2D）へ接続する。なお、この関数呼び出しの第 5 引数 0 にはミップマップのレベルを指定するが、この講義ではミップマップは用いないため、0 でよい。フレームバッファオブジェクト（FBO）とテクスチャの接続については既に図 11.2 で述べたが、より詳細に呼べると、実はこの接続は 2 段階の処理で行うことになっている。上の関数呼び出しは、FBO と各テクスチャの間に出力経路を構築するだけである。テクスチャが複数ある場合（次章の例題でそのような場合を扱う）には複数の経路が構築される。図 11.12 がそのイメージ図である。実際にデータを流す経路を選択する処理は、次項のメンバー関数 Shader::bindTextureW() で行う。

[1] OpenGL の言葉では color attachment という。num はテクスチャの装置番号である。実は、装置番号と color attachment の番号は同じである必要はないが、ここでは簡単のために同じ番号を用いる。

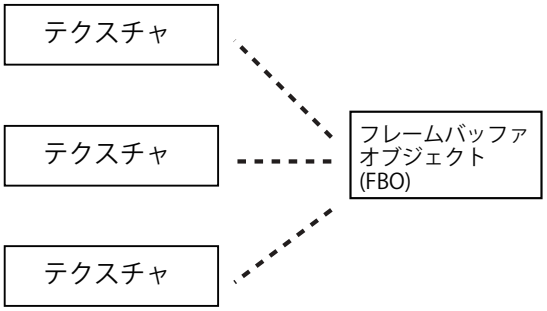

図 11.12: FBO から各テクスチャへの出力経路の構築

次に、メンバー関数 readData() は、このテクスチャに対応する GPU メモリ上のデータを CPU メモリ上の第 1 引数のアドレスへ転送する関数である。図 11.2 の (c) に相当する。関数呼び出し

```
23      glReadBuffer(GL_COLOR_ATTACHMENT0+(num));
```

は、num 番目の出力経路に接続されているテクスチャを読み出し元として指定する。num の値はコンストラクタで初期設定されていることに注意。関数呼び出し

```
24      glReadPixels(0, 0, w, h, GL_RGBA, GL_FLOAT, data);// 転送の実施
```

は、読み出し元テクスチャから配列 data へ実際にデータを読み出す。

11.2.2　Shader クラスの拡張

まず、図 11.7 のように、メンバー関数 Shader::run() では glutSwapBuffers() ではなく glFlush() を呼び出す。これは、前章までは CG アニメーションを行なっていたため、フレームバッファ2 枚（ダブルバッファ）を使い、描画毎にバッファを切り替えたが、この章以降では CG アニメーションは行わないからである。

次に、新規に導入したクラス RWTexture2D に関連する二つのメンバー関数 Shader::bindTextureR() と Shader::bindTextureW() を Shader クラスへ追加実装する。

関数 Shader::bindTextureR() はテクスチャを読み込み用としてシェーダーに結合するものであり、前章の図 10.6 の Shader::bindTexture() と同じ内容である。ただし、図 11.7 の

```
10      glActiveTexture(GL_TEXTURE0+(tp->num));                    // 注 1
```

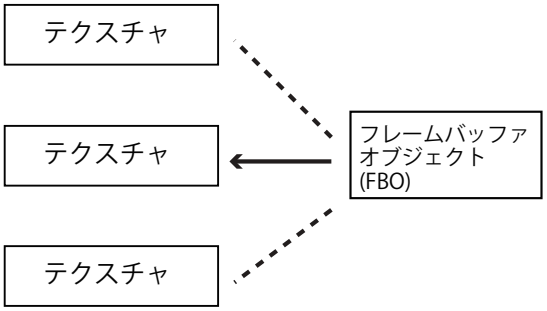

図 11.13: 出力経路の中のひとつを Shader::bindTextureW() で選択

は、次章のプログラムの正常な実行のために関数本体の冒頭へ移動する [2]。

関数 Shader::bindTextureW() はテクスチャを書き込み用として宣言するものである。その本体の関数呼び出し：

```
22      glDrawBuffer(GL_COLOR_ATTACHMENT0+(tp->num));
```

は、フレームバッファオブジェクト（FBO）からテクスチャへの出力経路のうち、tp->num 番目の出力経路を実際に選択する関数である。各テクスチャが生成されたときには、図 11.12 のように、FBO から各テクスチャへの出力経路が構築されている。上の関数は、ちょうど図 11.13 のように、その複数の経路からひとつの経路を選択するものである。シェーダープログラムを実行すればフラグメントシェーダーの出力がここで選択されたテクスチャへ書き込まれることになる。

11.2.3　ホストプログラム

図 11.8、図 11.9 がクラス以外のホストプログラム部分である。

まず、以下で計算のサイズを指定する

```
03 const int width = 5;                    //画面の幅＝テクスチャの幅を定数
04 const int height = 7;                   //画面の高さ＝テクスチャの高さ
```

テクスチャを用いた計算は 2 次元配列を用いた計算に相当する。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。描画面（＝ウインドウ）のサイズ、テクスチャのサイズには width、height を用いて同じサイズを指定する。

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ーの出力）に結合する処理である。図 11.2 の (a) に相当する。この部分は次章まで変更しないため、initSystem() に加えた。

[2] この実行位置の不具合に、次章部分の講義テキストの内容をチェックしているときに気づいた。

```
20      GLuint fb;
21      glGenFramebuffers(1, &fb);      // FBO を OpenGL コンテキストに登録
22      glBindFramebuffer(GL_FRAMEBUFFER, fb);
                                        // FBO をフレームバッファに結合
```

また、関数 initData() でのテクスチャの処理が前章よりも格段に複雑化する関係上、sp->use() を initSystem() の末尾に移動させる [3]。

```
25      sp->use();                              // シェーダーの使用の宣言
```

以下は、書き込み用テクスチャのポインタの宣言である。このポインタは、複数の関数で参照するため、大域化しておく。

```
30 RWTexture2D *texZp;              // 書き込み用 2 次元テクスチャのポインタ
```

以下は、書き込み用テクスチャの生成と設定である。

```
42      texZp = new RWTexture2D(0, NULL, width, height);
43      sp->bindTextureW(texZp);            // テクスチャを書き込み用に設定
```

なお、new RWTexture2D(0, NULL, width, height) の第 1 引数はテクスチャの装置番号、第 2 引数は初期データの格納されているメモリエリアの先頭アドレス、第 3、第 4 引数はテクスチャの幅、高さを指定する。装置番号は 0 以上の整数である [4]。第 2 引数が NULL の場合には、CPU から GPU へデータ転送を行わない。書き込み用テクスチャでは NULL を用いてよい。

関数 compute() は前章の display() と同様に実行可能シェーダープログラムを起動する。

そして関数 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。まず、以下でシェーダープログラムの実行の終了を待つ。

```
54      glFinish();                     // シェーダープログラムの実行終了待ち
```

関数 compute() の sp->run(GL_TRIANGLES, NUM_POINTS) はシェーダープログラムを起動するが、実行終了を待たない [5]。というのも、シェーダーの実行終了を待たずにホストは別の処理を行った方が効率がよいからである。しかし、計算結果をチェックするために

[3] 前章までは sp->use() は display() の冒頭に置いていた。

[4] 装置番号は 0 から順に付番しないとプログラムが正常動作しない場合がある。

[5] 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。終了を待つ場合、その間に CPU が遊んでしまうから実行効率を落とす。並列計算ではしばしばノンブロッキング型の処理が行われる。

は、シェーダーの実行が完全に終了している必要がある。glFinish() は実行が終了するまで関数呼び出しから戻ってこない関数である [6]。次に以下がテクスチャデータのホスト側への読み込みである。

```
56      texZp->readData(results, width, height);
```

配列 results にテクスチャデータが格納されているから、それを以下の形式で表示する。

```
61              printf("(%3.1f,%3.1f) ",          // 読み込んだ内容の表示
62                      results[h][w].r, results[h][w].g);
```

読みやすい整形された出力を見るために、ここでは cout ではなく、printf() を用いた [7]。なお、上の表示ではテクスチャの r 成分と g 成分しか表示していないことを注意する。テクスチャデータの値を見かけの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。

```
57      for(int h = height-1; h >= 0; h--)                  // 上から順に
```

11.2.4　バーテックスシェーダープログラム

図 11.10 がここでのバーテックスシェーダープログラムである。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。

11.2.5　フラグメントシェーダープログラム

図 11.11 がフラグメントシェーダープログラムである。

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ー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）に自動的に格納される**と約束されている。よって、その値の範囲は

$$(x\text{ 成分}, y\text{ 成分}) \in [0, \texttt{width}] \times [0, \texttt{height}]$$

である。より正確に言えば、各画素の中心の位置が格納されるため、それぞれ

$$\begin{aligned} x\text{ 成分} &= 0.5,\ 1.5,\ 2.5,\ \cdots,\ \texttt{width} - 0.5 \\ y\text{ 成分} &= 0.5,\ 1.5,\ 2.5,\ \cdots,\ \texttt{height} - 0.5 \end{aligned}$$

の離散値が格納される。例題では width= 5、height= 7 と設定しているから、図 11.14 の通りである。

[6] 対して、sp->run(GL_TRIANGLES, NUM_POINTS) の中で呼ばれている glFlush() がシェーダーを起動後、直ちに関数呼び出しから戻ってくる関数である。

[7] cout でも同様の出力が可能であるが、講義担当者がそれに慣れていない。

(0.5, 6.5)	(1.5, 6.5)	(2.5, 6.5)	(3.5, 6.5)	(4.5, 6.5)
(0.5, 5.5)	(1.5, 5.5)	(2.5, 5.5)	(3.5, 5.5)	(4.5, 5.5)
(0.5, 4.5)	(1.5, 4.5)	(2.5, 4.5)	(3.5, 4.5)	(4.5, 4.5)
(0.5, 3.5)	(1.5, 3.5)	(2.5, 3.5)	(3.5, 3.5)	(4.5, 3.5)
(0.5, 2.5)	(1.5, 2.5)	(2.5, 2.5)	(3.5, 2.5)	(4.5, 2.5)
(0.5, 1.5)	(1.5, 1.5)	(2.5, 1.5)	(3.5, 1.5)	(4.5, 1.5)
(0.5, 0.5)	(1.5, 0.5)	(2.5, 0.5)	(3.5, 0.5)	(4.5, 0.5)

図 11.14: 画像（出力用テクスチャ）上の gl_FragCoord の x 成分と y 成分：width= 5、height= 7 の場合

11.2.6　実行結果

図 11.15 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。

```
(0.0,0.0) (0.0,0.0) (0.0,0.0) (0.0,0.0) (0.0,0.0)
(0.0,0.0) (0.0,0.0) (0.0,0.0) (0.0,0.0) (0.0,0.0)
(0.0,0.0) (0.0,0.0) (0.0,0.0) (3.5,4.5) (0.0,0.0)
(0.0,0.0) (0.0,0.0) (2.5,3.5) (3.5,3.5) (0.0,0.0)
(0.0,0.0) (0.0,0.0) (2.5,2.5) (3.5,2.5) (0.0,0.0)
(0.0,0.0) (0.0,0.0) (0.0,0.0) (0.0,0.0) (0.0,0.0)
(0.0,0.0) (0.0,0.0) (0.0,0.0) (0.0,0.0) (0.0,0.0)
```

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。それ以外の部分は (0.0,0.0) であるから、書き込みは行われていない。

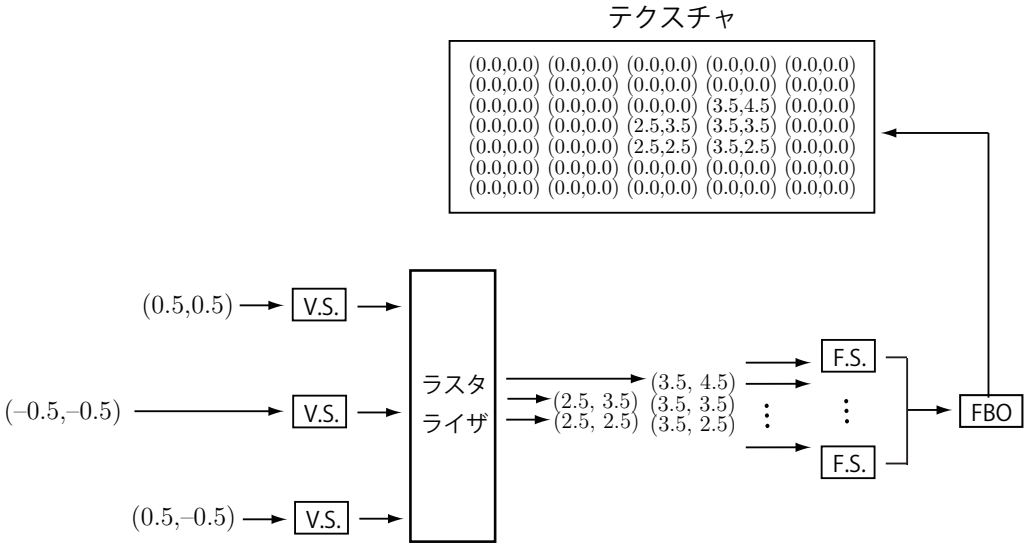

図 11.15: テクスチャへの代入の動作例

11.3　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ー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。以下がそのプログラムである。

154 ページの図 11.3、155 ページの図 11.4 と同じ

図 11.16: 各種クラスを定義するヘッダープログラム `All.h`

89 ページの図 7.2 と同じ

図 11.17: `ArrayBuffer` クラスの実装プログラム `Buffer.cpp`

156 ページの図 11.6 と同じ

図 11.18: 読み書き可能 2 次元テクスチャクラス `RWTexture2D` のクラスの実装プログラム `Textures.cpp`

157 ページの図 11.7 と同じ

図 11.19: `Shader` クラスの実装プログラム `Shader.cpp`（その他の部分は、89 ページの図 7.3、90 ページの図 7.4 と同じ）

```
01 #include "All.h"
02
03 const int width = 5;
04 const int height = 7;
05
06 Shader *sp;
07
08 void initSystem(int argc, char *argv[])
09 {
10     glutInit(&argc,argv);
11     glutInitDisplayMode(GLUT_RGB | GLUT_SINGLE);
12     glutInitWindowSize(width, height);
13     glutCreateWindow("Test Window");
14 //    glClearColor(0.0, 0.0, 0.0, 0.0);
                          //全域へ書き込むため、背景色の設定はもはや不要
15
16 #if defined(WIN32)
17     glewInit();
18 #endif
19
20     GLuint fb;
21     glGenFramebuffers(1, &fb);
22     glBindFramebuffer(GL_FRAMEBUFFER, fb);
23
24     sp = new Shader("shader.vert","shader.frag");
25     sp->use();
26 }
27
28 const int NUM_POINTS = 4;             // 頂点数を 4 へ変更
29
30 RWTexture2D *texZp;
31
32 void initData()
33 {
34     Position2D pos[NUM_POINTS];
35     pos[0].x = -1.0; pos[0].y = -1.0;              // 画面左下の位置
36     pos[1].x = +1.0; pos[1].y = -1.0;              // 画面右下の位置
37     pos[2].x = +1.0; pos[2].y = +1.0;              // 画面右上の位置
38     pos[3].x = -1.0; pos[3].y = +1.0;              // 画面左上の位置
                       // [-1.0,1.0]x[-1.0,1.0] を覆うように 4 頂点を設定
39
40     ArrayBuffer ab((float*)pos, 2, NUM_POINTS);
41     sp->bindArrayBuffer("position", &ab);
42
43     texZp = new RWTexture2D(0, NULL, width, height);
44     sp->bindTextureW(texZp);
45 }
46
```

図 11.20: テクスチャ全域への代入を行うホストプログラム main.cpp（その 1、その 2 へ続く）

```
47 void compute(void)
48 {
49 //    glClear(GL_COLOR_BUFFER_BIT);
                  //全域へ書き込むため、背景色での画面クリアはもはや不要
50     sp->run(GL_POLYGON, NUM_POINTS);// 描画モードを GL_POLYGON へ変更
51 }
52
53 void showResults()
54 {
55     glFinish();
56
57     RGBA results[height][width];
58     texZp->readData(results, width, height);
59     for(int h = height-1; h >= 0; h--)
60     {
61         for(int w = 0; w < width; w++)
62         {
63             printf("(%3.1f,%3.1f) ",
64                    results[h][w].r, results[h][w].g);
65         }
66         printf("\n");
67     }
68 }
69
70 int main(int argc, char *argv[])
71 {
72     initSystem(argc,argv);
73     initData();
74     compute();
75     showResults();
76     return 0;
77 }
```

図 11.21: テクスチャ全域への代入を行うホストプログラム main.cpp（その 2）

159 ページの図 11.10 と同じ

図 11.22: テクスチャ全域への代入を行うバーテックスシェーダープログラム shader.vert

160 ページの図 11.11 と同じ

図 11.23: テクスチャ全域への代入を行うフラグメントシェーダープログラム shader.frag

11.3.1　ホストプログラム

三角形を描画するモード（GL_TRIANGLES）ではなく、凸型図形を描画するためのモード（GL_POLYGON）を用いて正方形を描画する。

図 11.20 の

```
28 const int NUM_POINTS = 4;              // 頂点数を 4 へ変更
```

で、正方形の 4 頂点を用いることを宣言する。次に、

```
35      pos[0].x = -1.0; pos[0].y = -1.0;          // 画面左下の位置
36      pos[1].x = +1.0; pos[1].y = -1.0;          // 画面右下の位置
37      pos[2].x = +1.0; pos[2].y = +1.0;          // 画面右上の位置
38      pos[3].x = -1.0; pos[3].y = +1.0;          // 画面左上の位置
```

で、画面全体を覆う正方形の 4 頂点の座標値を画面の 4 隅になるように順に設定する。シェーダーの起動は

```
50      sp->run(GL_POLYGON, NUM_POINTS);// 描画モードを GL_POLYGON へ変更
```

と指定する。そうすると、4 頂点で囲まれた多角形（polygon）の内部が描画対象となる。結果、画面上の全画素についてフラグメントシェーダーが起動する。

11.3.2　実行結果

図 11.24 は実行のイメージである。showResults() の出力は以下の通りとなり、全域で代入が行われている。

```
(0.5,6.5) (1.5,6.5) (2.5,6.5) (3.5,6.5) (4.5,6.5)
(0.5,5.5) (1.5,5.5) (2.5,5.5) (3.5,5.5) (4.5,5.5)
(0.5,4.5) (1.5,4.5) (2.5,4.5) (3.5,4.5) (4.5,4.5)
(0.5,3.5) (1.5,3.5) (2.5,3.5) (3.5,3.5) (4.5,3.5)
(0.5,2.5) (1.5,2.5) (2.5,2.5) (3.5,2.5) (4.5,2.5)
(0.5,1.5) (1.5,1.5) (2.5,1.5) (3.5,1.5) (4.5,1.5)
(0.5,0.5) (1.5,0.5) (2.5,0.5) (3.5,0.5) (4.5,0.5)
```

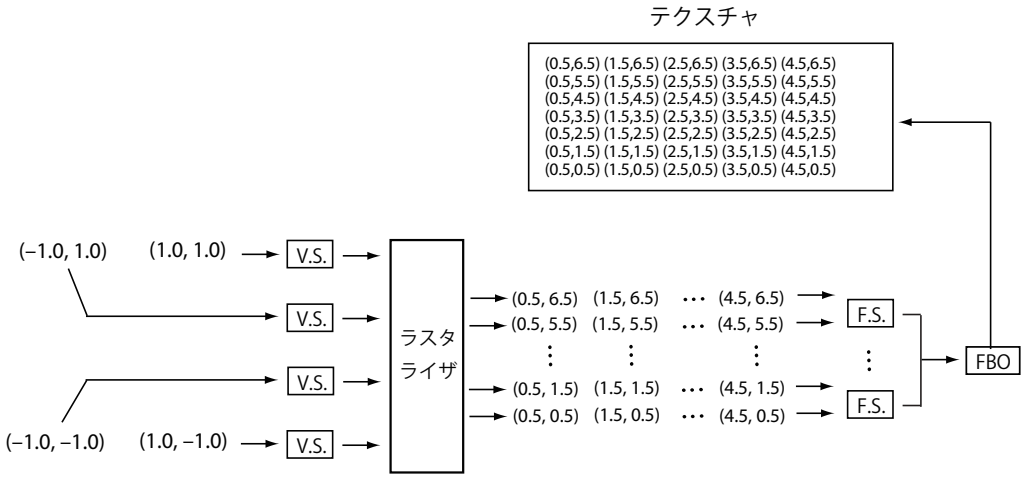

図 11.24: テクスチャ全域への代入の動作例

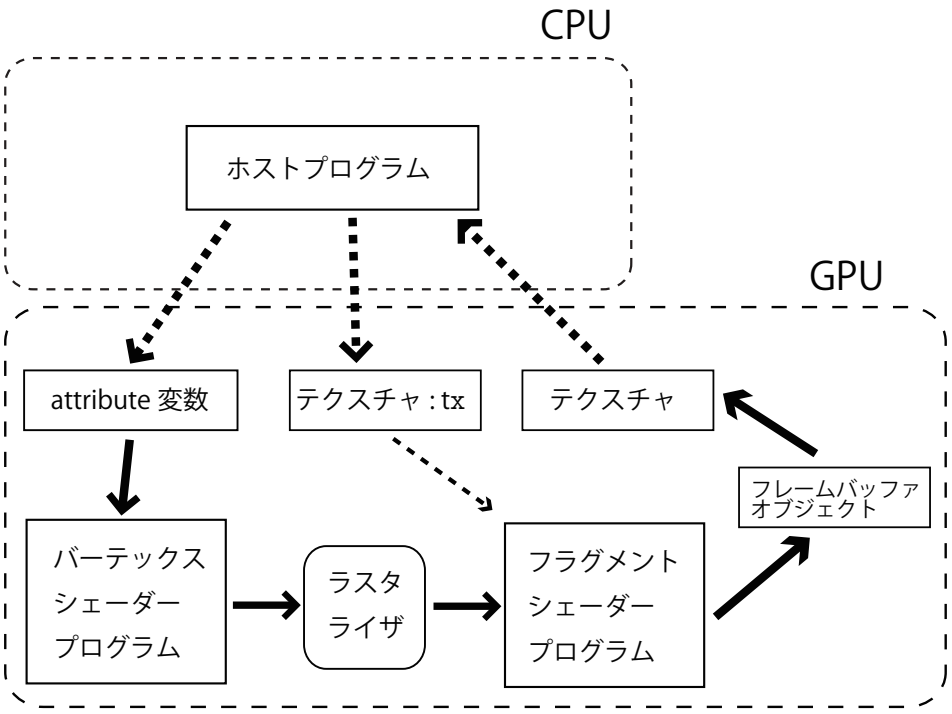

図 11.25: コピーの経路

11.4　テクスチャ間のデータのコピー

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];
```

という実行である。

GPU 内のデータの流れを図 11.25 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ー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り、11.2.1 節の読み書き可能テクスチャRWTexture2D でそれを行う。次に、そのテクスチャの内容を出力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。

154 ページの図 11.3、155 ページの図 11.4 と同じ

図 11.26: 各種クラスを定義するヘッダープログラム `All.h`

89 ページの図 7.2 と同じ

図 11.27: `ArrayBuffer` クラスの実装プログラム `Buffer.cpp`

156 ページの図 11.6 と同じ

図 11.28: 読み書き可能 2 次元テクスチャクラス `RWTexture2D` のクラスの実装プログラム `Textures.cpp`

157 ページの図 11.7 と同じ

図 11.29: `Shader` クラスの実装プログラム `Shader.cpp`（その他の部分は、89 ページの図 7.3、90 ページの図 7.4 と同じ）

```
01 #include "All.h"
02
03 const int width = 5;
04 const int height = 7;
05
06 Shader *sp;
07
08 void initSystem(int argc, char *argv[])
09 {
10     glutInit(&argc,argv);
11     glutInitDisplayMode(GLUT_RGB | GLUT_SINGLE);
12     glutInitWindowSize(width, height);
13     glutCreateWindow("Test Window");
14
15 #if defined(WIN32)
16     glewInit();
17 #endif
18
19     GLuint fb;
20     glGenFramebuffers(1, &fb);
21     glBindFramebuffer(GL_FRAMEBUFFER, fb);
22
23     sp = new Shader("shader.vert","shader.frag");
24     sp->use();
25 }
26
27 const int NUM_POINTS = 4;
28
29 RWTexture2D *texXp;           // 読み込み用２次元テクスチャのポインタ
30 RWTexture2D *texZp;           // 書き込み用２次元テクスチャのポインタ
31
32 RGBA x[height][width];        // 読み込み用テクスチャデータ配列
33
34 void initData()
35 {
36     Position2D pos[NUM_POINTS];
37     pos[0].x = -1.0; pos[0].y = -1.0;
38     pos[1].x = +1.0; pos[1].y = -1.0;
39     pos[2].x = +1.0; pos[2].y = +1.0;
40     pos[3].x = -1.0; pos[3].y = +1.0;
41
42     ArrayBuffer ab((float*)pos, 2, NUM_POINTS);
43     sp->bindArrayBuffer("position", &ab);
44
```

図 11.30: テクスチャのコピーを行うホストプログラム main.cpp（その 1、その 2 へ続く）

```
45      sp->setFloat("width", width);     // 画面の幅を uniform 変数に設定
46      sp->setFloat("height", height);// 画面の高さを uniform 変数に設定
47
48      for(int h = height-1; h >= 0; h--)
49      {
50          for(int w = 0; w < width; w++)
51          {
52              x[h][w].r = h+w+0.1;
53              x[h][w].g = h+w+0.2;  // 読み込み用サンプルデータの設定
54              x[h][w].b = h+w+0.3;
55              x[h][w].a = h+w+0.4;
56          }
57      }
58
59      texXp = new RWTexture2D(0, x, width, height);
                                              // 読み込み用テクスチャの生成
60      texZp = new RWTexture2D(1, NULL, width, height);
                                              // 書き込み用テクスチャの生成
61
62      sp->bindTextureR("tx", texXp);
              // 読み込み用テクスチャをシェーダーの uniform 変数 tx と結合
63      sp->bindTextureW(texZp);           // テクスチャを書き込み用に設定
64  }
65
66  void compute(void)
67  {
68      sp->run(GL_POLYGON, NUM_POINTS);
69  }
70
71  void showResults()
72  {
73      glFinish();
74
75      RGBA results[height][width];
76      texZp->readData(results, width, height);
77      for(int h = height-1; h >= 0; h--)
78      {
79          for(int w = 0; w < width; w++)
80          {
81              printf("%d %d r : %7.3f %7.3f\n",
82                     h, w, x[h][w].r, results[h][w].r);
83              printf("%d %d g : %7.3f %7.3f\n",          // 元のデータ
84                     h, w, x[h][w].g, results[h][w].g); // とコピー後
85              printf("%d %d b : %7.3f %7.3f\n",          // のデータを
86                     h, w, x[h][w].b, results[h][w].b); // 並べて表示
87              printf("%d %d a : %7.3f %7.3f\n",
88                     h, w, x[h][w].a, results[h][w].a);
89          }
90      }
91  }
```

図 11.31: テクスチャのコピーを行うホストプログラム main.cpp（その 2、その 3 へ続く）

```
92
93 int main(int argc, char *argv[])
94 {
95     initSystem(argc,argv);
96     initData();
97     compute();
98     showResults();
99     return 0;
A0 }
```

図 11.32: テクスチャのコピーを行うホストプログラム main.cpp（その 3）

159 ページの図 11.10 と同じ

図 11.33: テクスチャのコピーを行うバーテックスシェーダープログラム shader.vert

```
01 #version 120
02
03 uniform sampler2D tx;                          // 読み込み用テクスチャ
04
05 uniform float width;                           // テクスチャの幅
06 uniform float height;                          // テクスチャの高さ
07
08 void main(void)
09 {
10     vec2 texCoord = vec2(gl_FragCoord.x/width,  // gl_FragCoord.xy
11                          gl_FragCoord.y/height);// の値を正規化
12
13     gl_FragColor = texture2D(tx, texCoord); // テクスチャの読み込み
14 }
```

図 11.34: テクスチャのコピーを行うフラグメントシェーダープログラム shader.frag

11.4.1　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。これらは initData() と showResults() から使用される。

```
30 RWTexture2D *texZp;              // 書き込み用 2 次元テクスチャのポインタ
31
32 RGBA x[height][width];           // 読み込み用テクスチャデータ配列
```

さて、理由は後に述べるが、テクスチャをシェーダーで読み込むプログラムでは画面の幅と高さをシェーダーの uniform 変数に受け渡す必要がある。そこで図 11.31 の initData() の中で以下の処理を行う。

```
45      sp->setFloat("width", width);     // 画面の幅を uniform 変数に設定
46      sp->setFloat("height", height);// 画面の高さを uniform 変数に設定
```

次に、配列 x 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。

```
52              x[h][w].r = h+w+0.1;
53              x[h][w].g = h+w+0.2;  // 読み込み用サンプルデータの設定
54              x[h][w].b = h+w+0.3;
55              x[h][w].a = h+w+0.4;
```

たとえば x[2][3].r の場合、h+w+0.1$= 2 + 3 + 0.1 = 5.1$ となる。次に、以下で読み込み用テクスチャオブジェクトを x を用いて生成し、このオブジェクトをポインタ texXp に代入し、さらにこのオブジェクトをテクスチャunifrom 変数 tx に結びつける。

```
59      texXp = new RWTexture2D(0, x, width, height);
...
62    sp->bindTextureR("tx", texXp);
```

なお、59 行目から 63 行目：

```
59      texXp = new RWTexture2D(0, x, width, height);
60      texZp = new RWTexture2D(1, NULL, width, height);
61
62      sp->bindTextureR("tx", texXp);
63      sp->bindTextureW(texZp);          // テクスチャを書き込み用に設定
```

の実行順を

```
59      texXp = new RWTexture2D(0, x, width, height);
62      sp->bindTextureR("tx", texXp);

60      texZp = new RWTexture2D(1, NULL, width, height);
63      sp->bindTextureW(texZp);          // テクスチャを書き込み用に設定
```

などと変更した場合、プログラムは正常実行されない（かもしれない）。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ように思われ[8]、講義担当者は以下のルールで設定を行った。

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ーと結合する。

2. テクスチャの装置番号は 0 から順に使う。

関数 `showResults()` では

```
81              printf("%d %d r : %7.3f %7.3f\n",
82                     h, w, x[h][w].r, results[h][w].r);
```

コピー元の配列要素値 `x[h][w]` とシェーダーでコピーされた値 `results[h][w]` を並べて表示し、正しくコピーされたか否か目視確認する。

11.4.2　フラグメントシェーダー

フラグメントシェーダープログラムは図 11.34 のように変更する。

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意する。

```
10      vec2 texCoord = vec2(gl_FragCoord.x/width,  // gl_FragCoord.xy
11                           gl_FragCoord.y/height);// の値を正規化
```

何故ならば、`gl_FragCoord.xy` の値の範囲は

$$[0, \texttt{width}] \times [0, \texttt{height}]$$

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。OpenGL はもともと GPGPU を意識した仕様となっていないため、これは必要不可避の手間と理解する。

ところで、代入文：

```
13      gl_FragColor = texture2D(tx, texCoord); // テクスチャの読み込み
```

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 SIMD 演算を含むのである。

[8] 設定の順序のこの問題は講義担当者もまだよく理解できていない。

11.4.3　実行結果

showResults() の実行結果は以下の通りとなり、コピーは正しく行われていることが分かる。

```
0 6 r :   6.100   6.100
0 6 g :   6.200   6.200
0 6 b :   6.300   6.300
0 6 a :   6.400   6.400
1 6 r :   7.100   7.100
...
中略
...
3 0 a :   3.400   3.400
4 0 r :   4.100   4.100
4 0 g :   4.200   4.200
4 0 b :   4.300   4.300
4 0 a :   4.400   4.400
```

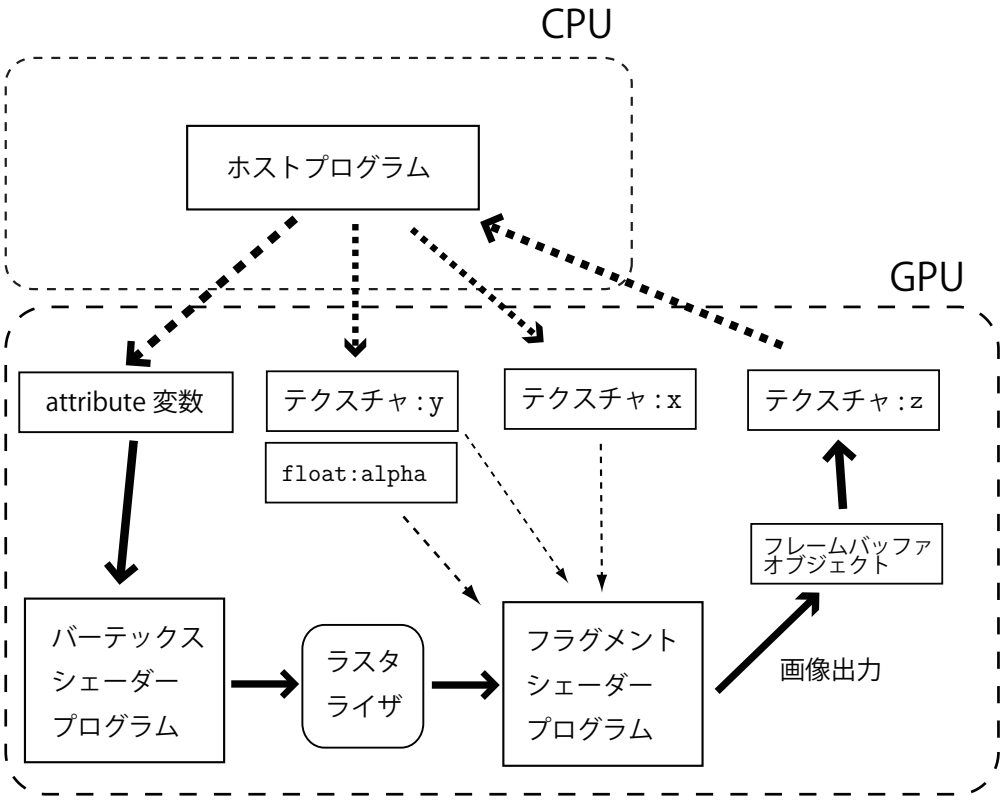

図 11.35: saxpy 計算の経路

11.5　テクスチャを用いた簡単な並列計算

テクスチャデータの読み書きができることを確認した。そこで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ってみよう。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 に由来している。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 [9]。

[9] 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。

154 ページの図 11.3、155 ページの図 11.4 と同じ

図 11.36: 各種クラスを定義するヘッダープログラム `All.h`

89 ページの図 7.2 と同じ

図 11.37: `ArrayBuffer` クラスの実装プログラム `Buffer.cpp`

156 ページの図 11.6 と同じ

図 11.38: 読み書き可能 2 次元テクスチャクラス `RWTexture2D` のクラスの実装プログラム `Textures.cpp`

157 ページの図 11.7 と同じ

図 11.39: `Shader` クラスの実装プログラム `Shader.cpp`（その他の部分は、89 ページの図 7.3、90 ページの図 7.4 と同じ）

```
01 #include "All.h"
02
03 const int width = 5;
04 const int height = 7;
05
06 Shader *sp;
07
08 void initSystem(int argc, char *argv[])
09 {
10     glutInit(&argc,argv);
11     glutInitDisplayMode(GLUT_RGB | GLUT_SINGLE);
12     glutInitWindowSize(width, height);
13     glutCreateWindow("Test Window");
14
15 #if defined(WIN32)
16     glewInit();
17 #endif
18
19     GLuint fb;
20     glGenFramebuffers(1, &fb);
21     glBindFramebuffer(GL_FRAMEBUFFER, fb);
22
23     sp = new Shader("shader.vert","shader.frag");
24     sp->use();
25 }
26
27 const int NUM_POINTS = 4;
28
29 RWTexture2D *texXp;           // 読み込み用２次元テクスチャのポインタ
30 RWTexture2D *texYp;           // 読み込み用２次元テクスチャのポインタ
31 RWTexture2D *texZp;           // 書き込み用２次元テクスチャのポインタ
32
33 float alpha = 0.5;            // 定数
34 RGBA x[height][width];        // 読み込み用テクスチャデータ配列
35 RGBA y[height][width];        // 読み込み用テクスチャデータ配列
36
37 void initData()
38 {
39     Position2D pos[NUM_POINTS];
40     pos[0].x = -1.0; pos[0].y = -1.0;
41     pos[1].x = +1.0; pos[1].y = -1.0;
42     pos[2].x = +1.0; pos[2].y = +1.0;
43     pos[3].x = -1.0; pos[3].y = +1.0;
44
45     ArrayBuffer ab((float*)pos, 2, NUM_POINTS);
46     sp->bindArrayBuffer("position", &ab);
47
```

図 11.40: saxpy 計算を行うホストプログラム main.cpp（その 1、その 2 へ続く）

```
48      sp->setFloat("width", width);
49      sp->setFloat("height", height);
50
51      sp->setFloat("alpha",alpha);  // saxpy の係数を uniform 変数に設定
52
53      for(int h = height-1; h >= 0; h--){
54          for(int w = 0; w < width; w++){
55              x[h][w].r = h+w+0.1;
56              x[h][w].g = h+w+0.2;
57              x[h][w].b = h+w+0.3;
58              x[h][w].a = h+w+0.4;
59
60              y[h][w].r = h-w+0.1;
61              y[h][w].g = h-w+0.2;  // 読み込み用サンプルデータの設定
62              y[h][w].b = h-w+0.3;
63              y[h][w].a = h-w+0.4;
64          }
65      }
66      texXp = new RWTexture2D(0, x, width, height);
                                              // 読み込み用テクスチャの生成
67      texYp = new RWTexture2D(1, y, width, height);
                                              // 読み込み用テクスチャの生成
68      texZp = new RWTexture2D(2, NULL, width, height);
                                              // 書き込み用テクスチャの生成
69
70      sp->bindTextureR("tx", texXp);
              // 読み込み用テクスチャをシェーダーの uniform 変数 tx と結合
71      sp->bindTextureR("ty", texYp);
              // 読み込み用テクスチャをシェーダーの uniform 変数 ty と結合
72      sp->bindTextureW(texZp);           // テクスチャを書き込み用に設定
73  }
74
75  void compute(void)
76  {
77      sp->run(GL_POLYGON, NUM_POINTS);
78  }
79
```

図 11.41: saxpy 計算を行うホストプログラム main.cpp（その 2、その 3 へ続く）

```
80 void showResults()
81 {
82     glFinish();
83
84     RGBA results[height][width];
85     texZp->readData(results, width, height);
86     for(int h = height-1; h >= 0; h--)
87     {
88         for(int w = 0; w < width; w++)
89         {  // ホストでの計算結果とシェーダーでの計算結果を並べて表示
90             printf("%d %d r : %7.3f %7.3f\n",
91                  h, w, alpha*x[h][w].r+y[h][w].r, results[h][w].r);
92             printf("%d %d g : %7.3f %7.3f\n",
93                  h, w, alpha*x[h][w].g+y[h][w].g, results[h][w].g);
94             printf("%d %d b : %7.3f %7.3f\n",
95                  h, w, alpha*x[h][w].b+y[h][w].b, results[h][w].b);
96             printf("%d %d a : %7.3f %7.3f\n",
97                  h, w, alpha*x[h][w].a+y[h][w].a, results[h][w].a);
98         }
99     }
A0 }
A1
A2 int main(int argc, char *argv[])
A3 {
A4     initSystem(argc,argv);
A5     initData();
A6     compute();
A7     showResults();
A8    return 0;
A9 }
```

図 11.42: saxpy 計算を行うホストプログラム `main.cpp`（その 3）

159 ページの図 11.10 と同じ

図 11.43: saxpy 計算を行うバーテックスシェーダープログラム `shader.vert`

```
01 #version 120
02
03 uniform sampler2D tx;                          // 読み込み用テクスチャ
04 uniform sampler2D ty;                          // 読み込み用テクスチャ
05
06 uniform float alpha;                           // saxpy 計算の定数値
07 uniform float width;                           // テクスチャの幅
08 uniform float height;                          // テクスチャの高さ
09
10 void main(void)
11 {
12     vec2 texCoord = vec2(gl_FragCoord.x/width,  // gl_FragCoord.xy
13                          gl_FragCoord.y/height);// の値を正規化
14
15
16     vec4 x = texture2D(tx,texCoord);          // テクスチャの読み込み
17     vec4 y = texture2D(ty,texCoord);          // テクスチャの読み込み
18
19     gl_FragColor = alpha*x+y;                 // saxpy 計算
20 }
```

図 11.44: saxpy 計算を行うフラグメントシェーダープログラム shader.frag

11.5.1　ホストプログラム

図 11.40〜図 11.42 がクラス定義以外のホストプログラムである。

まず、プログラムの構成上、以下の大域変数を宣言しておく。

```
29 RWTexture2D *texXp;            // 読み込み用 2 次元テクスチャのポインタ
30 RWTexture2D *texYp;            // 読み込み用 2 次元テクスチャのポインタ
31 RWTexture2D *texZp;            // 書き込み用 2 次元テクスチャのポインタ
32
33 float alpha = 0.5;              // 定数
34 RGBA x[height][width];          // 読み込み用テクスチャデータ配列
35 RGBA y[height][width];          // 読み込み用テクスチャデータ配列
```

関数 `initData()` には、`alpha` の値を uniform 変数としてシェーダーに受け渡す部分を追加する。

```
51      sp->setFloat("alpha",alpha);  // saxpy の係数を uniform 変数に設定
```

次に、前節の配列 `x` の初期設定に加えて、新たに配列 `y` に適当な値を初期設定する。

```
60              y[h][w].r = h-w+0.1;
61              y[h][w].g = h-w+0.2;  // 読み込み用サンプルデータの設定
62              y[h][w].b = h-w+0.3;
63              y[h][w].a = h-w+0.4;
```

そしてテクスチャの生成とシェーダーとの結合を行う。

```
66      texXp = new RWTexture2D(0, x, width, height);
67      texYp = new RWTexture2D(1, y, width, height);
68      texZp = new RWTexture2D(2, NULL, width, height);
69
70      sp->bindTextureR("tx", texXp);
71      sp->bindTextureR("ty", texYp);
72      sp->bindTextureW(texZp);          // テクスチャを書き込み用に設定
```

関数 `showResults()` のたとえば

```
88              printf("%d %d r : %7.3f %7.3f\n",
89                   h, w, alpha*x[h][w].r+y[h][w].r, results[h][w].r);
```

では、第 3 引数 `alpha*x[h][w].r+y[h][w].r` がホストでの saxpy の計算結果、第 4 引数 `results[h][w].r` がシェーダーでの計算結果である。両者を並べて表示する。

11.5.2　フラグメントシェーダープログラム

図 11.44 は saxpy 計算を行うフラグメントシェーダープログラムである。

uniform 変数 ty、alpha が追加されている。

gl_FragColor への代入文：

```
19      gl_FragColor = alpha*x + y;                 // saxpy 計算
```

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ーでは SIMD 計算が行われる。そして、多数のフラグメントシェーダーが並列に動作する – これをしばしば **SPMD**（Single Program Multiple Data）演算と呼ぶ – という図式になる。

11.5.3　実行結果

showResults() の実行結果は以下の通りとなり、CPU での計算と GPU での計算が一致することが確認できる。

```
6 0 r :   9.150   9.150
6 0 g :   9.300   9.300
6 0 b :   9.450   9.450
6 0 a :   9.600   9.600
6 1 r :   8.650   8.650
6 1 g :   8.800   8.800
...
中略
...
0 3 a :  -0.900  -0.900
0 4 r :  -1.850  -1.850
0 4 g :  -1.700  -1.700
0 4 b :  -1.550  -1.550
0 4 a :  -1.400  -1.400
```

以上で、本格的な GPGPU のための準備が整った。次節では大規模計算に挑戦する。

第12章　テクスチャを用いた大規模計算

前章の最後では1回の積和演算を行った。この程度では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。GPUによる計算の最も大きな問題点は

- CPUからGPUへ計算の素データを転送し、GPUからCPUへ計算結果を転送する処理に時間が掛かる

ことである。上記の処理を極力抑えることがGPUによる計算の要になるのだが、そのためにはCPUとGPUの間のデータ転送を長大な計算の初めと終わりにのみ行い、途中では極力、データ転送が要らないように工夫する必要がある。

12.1　テクスチャを用いたピンポン計算

次に考えるのは以下のようなsaxpy計算の繰り返しである。

```
for(int k = 0; k < N; 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。テクスチャによる計算では、ひとつのテクスチャを同時に読み込みと書き込み（参照と代入）の両方に使うことはできない。

そこで、この計算を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。都合上、Nは偶数と仮定する。

```
for(int k = 0; k < N/2; k++){
    z[i][j] = alpha*x[i][j]+y[i][j];    //前節の計算
    x[i][j] = alpha*z[i][j]+y[i][j];
}
```

ループ本体の1行目は前章の計算に他ならない。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12.1、図12.2である。図のtexXp、texYp、texZp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のポイン

タである。z[i][j] = alpha*x[i][j]+y[i][j] の計算では、それぞれがシェーダーの中で x、y、z 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 texXp、texYp、texZp は z、y、x としてアクセスされる。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。攻守（＝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。ただし、画面サイズ（＝テクスチャサイズ）は前章と同じく、まだ 5×7 の小さなままである。

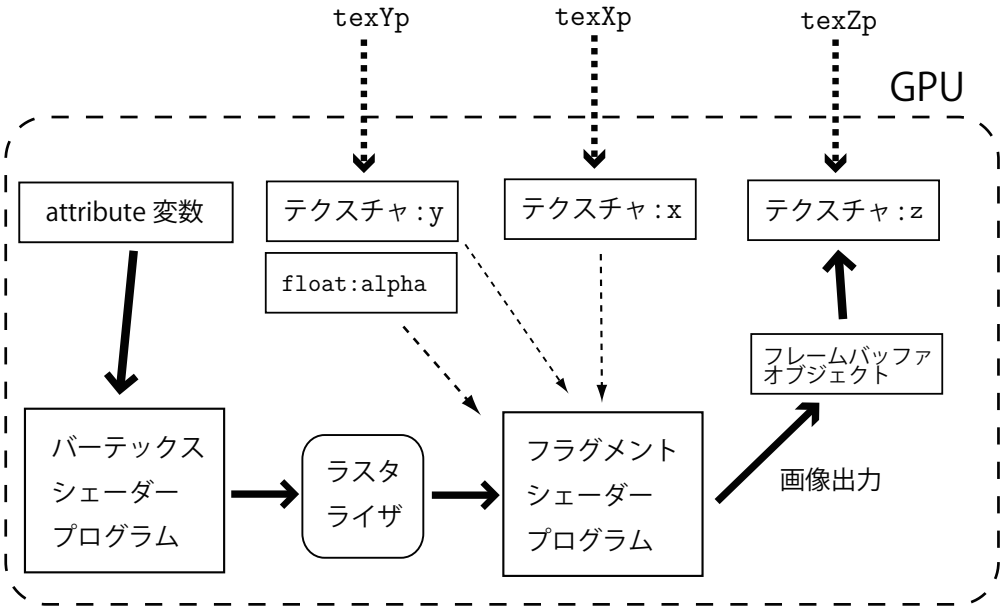

図 12.1: `z[i][j] = alpha*x[i][j]+y[i][j]` の計算経路

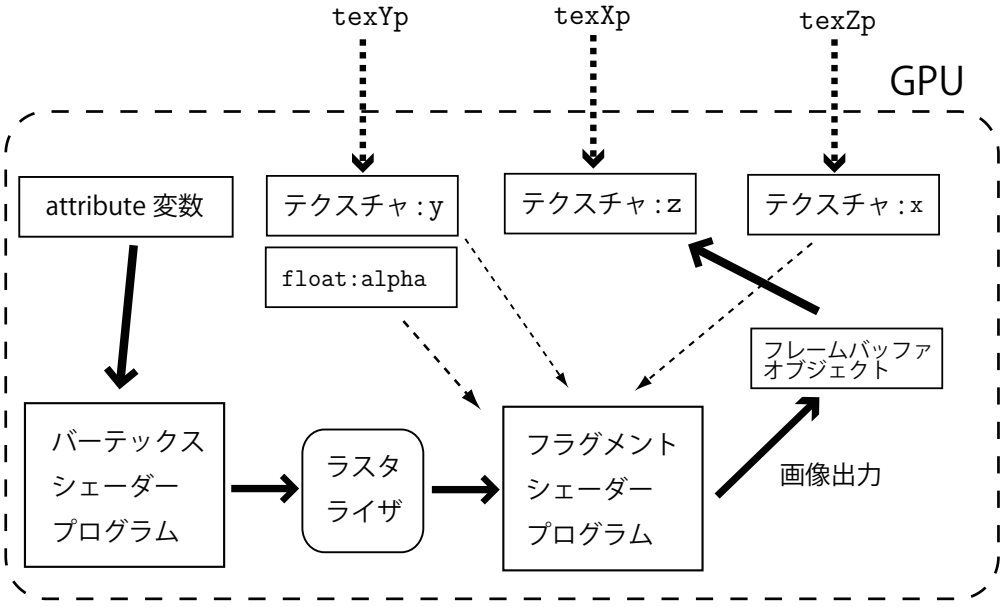

図 12.2: `x[i][j] = alpha*z[i][j]+y[i][j]` の計算経路

154 ページの図 11.3、155 ページの図 11.4 と同じ

図 12.3: 各種クラスを定義するヘッダープログラム `All.h`

89 ページの図 7.2 と同じ

図 12.4: `ArrayBuffer` クラスの実装プログラム `Buffer.cpp`

156 ページの図 11.6 と同じ

図 12.5: 読み書き可能 2 次元テクスチャクラス `RWTexture2D` のクラスの実装プログラム `Textures.cpp`

157 ページの図 11.7 と同じ

図 12.6: `Shader` クラスの実装プログラム `Shader.cpp`（その他の部分は、89 ページの図 7.3、90 ページの図 7.4 と同じ）

```
01 #include "All.h"
02 #include <math.h>                // 以下で fabs() を呼び出すために必要
03
04 const int width = 5;
05 const int height = 7;
06
07 Shader *sp;
08
09 void initSystem(int argc, char *argv[])
10 {
11     glutInit(&argc,argv);
12     glutInitDisplayMode(GLUT_RGB | GLUT_SINGLE);
13     glutInitWindowSize(width, height);
14     glutCreateWindow("Test Window");
15
16 #if defined(WIN32)
17     glewInit();
18 #endif
19
20     GLuint fb;
21     glGenFramebuffers(1, &fb);
22     glBindFramebuffer(GL_FRAMEBUFFER, fb);
23
24     sp = new Shader("shader.vert","shader.frag");
25     sp->use();
26 }
27
28 const int NUM_POINTS = 4;
29
30 RWTexture2D *texXp;
31 RWTexture2D *texYp;              // テクスチャオブジェクトへのポインタ
32 RWTexture2D *texZp;
33
34 float alpha = 0.5;
35 RGBA x[height][width];
36 RGBA y[height][width];          // 計算準備/計算結果の処理のための配列
37 RGBA z[height][width];
38 RGBA results[height][width];
39
40 void initData()
41 {
42     Position2D pos[NUM_POINTS];
43
44     pos[0].x = -1.0; pos[0].y = -1.0;
45     pos[1].x = +1.0; pos[1].y = -1.0;
46     pos[2].x = +1.0; pos[2].y = +1.0;
47     pos[3].x = -1.0; pos[3].y = +1.0;
48
49     ArrayBuffer ab((float*)pos,2,NUM_POINTS);
50    sp->bindArrayBuffer("position",&ab);
51
```

図 12.7: saxpy 計算を行うホストプログラム main.cpp（その 1、その 2 へ続く）

```
52      sp->setFloat("width", width);
53      sp->setFloat("height", height);
54
55      sp->setFloat("alpha",alpha);
56
57      for(int h = height-1; h >= 0; h--){
58          for(int w = 0; w < width; w++){
59              x[h][w].r = h+w+0.1;
60              x[h][w].g = h+w+0.2;
61              x[h][w].b = h+w+0.3;
62              x[h][w].a = h+w+0.4;
63
64              y[h][w].r = h-w+0.1;
65              y[h][w].g = h-w+0.2;
66              y[h][w].b = h-w+0.3;
67              y[h][w].a = h-w+0.4;
68          }
69      }
70
71      texXp = new RWTexture2D(0, x, width, height);
72      texYp = new RWTexture2D(1, y, width, height);
73      texZp = new RWTexture2D(2, NULL, width, height);
74
75      sp->bindTextureR("ty", texYp);
76  }
77
78  #define LOOPNUM 100                         // 繰り返し回数
79
80  void compute(void)
81  {
82      for(int n = 0; n < LOOPNUM/2; n++){          // 計算を繰り返す
83          sp->bindTextureR("tx", texXp);         // texXp を tx に結合
84          sp->bindTextureW(texZp);              // texZp を書き込み用に
85          sp->run(GL_POLYGON, NUM_POINTS);
86
87          sp->bindTextureR("tx", texZp);         // texZp を tx に結合
88          sp->bindTextureW(texXp);              // texXp を書き込み用に
89          sp->run(GL_POLYGON, NUM_POINTS);
90      }
91  }
92
```

図 12.8: saxpy 計算を行うホストプログラム main.cpp（その 2、その 3 へ続く）

```
93 void showResult()
94 {
95     glFinish();
96
97     texXp->readData(results,width,height);   // 計算結果を CPU へ転送
98
99     for(int n = 0; n < LOOPNUM/2; n++){      // CPU で同じ計算を行う
A0         for(int h = height-1; h >= 0; h--){
A1             for(int w = 0; w < width; w++){
A2                 z[h][w].r = alpha*x[h][w].r+y[h][w].r;
A3                 z[h][w].g = alpha*x[h][w].g+y[h][w].g;
A4                 z[h][w].b = alpha*x[h][w].b+y[h][w].b;
A5                 z[h][w].a = alpha*x[h][w].a+y[h][w].a;
A6
A7                 x[h][w].r = alpha*z[h][w].r+y[h][w].r;
A8                 x[h][w].g = alpha*z[h][w].g+y[h][w].g;
A9                 x[h][w].b = alpha*z[h][w].b+y[h][w].b;
B0                 x[h][w].a = alpha*z[h][w].a+y[h][w].a;
B1             }
B2         }
B3     }
B4
B5     double error = 0.0;
B6     for(int h = height-1; h >= 0; h--){
B7         for(int w = 0; w < width; w++){
B8             error += fabs(x[h][w].r-results[h][w].r);
B9             error += fabs(x[h][w].g-results[h][w].g); // 誤算の積算
C0             error += fabs(x[h][w].b-results[h][w].b);
C1             error += fabs(x[h][w].a-results[h][w].a);
C2         }
C3     }
C4     printf("%30.25le\n",error);               // 積算した誤差を表示
C5 }
C6
C7 int main(int argc, char *argv[])
C8 {
C9     initSystem(argc,argv);
D0     initData();
D1     compute();
D2     showResult();
D3     return 0;
D4 }
```

図 12.9: saxpy 計算を行うホストプログラム main.cpp（その 3）

159 ページの図 11.10 と同じ

図 12.10: saxpy 計算を行うバーテックスシェーダープログラム shader.vert

187 ページの図 11.44 と同じだが、重要であるから以下に再掲

```
01 #version 120
02
03 uniform sampler2D tx;
04 uniform sampler2D ty;
05
06 uniform float alpha;
07 uniform float width;
08 uniform float height;
09
10 void main(void)
11 {
12     vec2 texCoord = vec2(gl_FragCoord.x/width,
13                          gl_FragCoord.y/height);
14
15
16     vec4 x = texture2D(tx,texCoord);
17     vec4 y = texture2D(ty,texCoord);
18
19     gl_FragColor = alpha*x+y;
20 }
```

図 12.11: saxpy 計算を行うフラグメントシェーダープログラム `shader.frag`

12.1.1　ホストプログラム

次節以降の大規模計算では、巨大なサイズの配列を用いる。そのような配列は、通常、関数の局所変数として宣言できない [1]。

その点を考慮し、次節以降の準備のために、テクスチャオブジェクトを指すポインタ、テクスチャの初期データを格納する配列、計算結果を格納する配列は全て大域化しておく。図 12.7 の以下の部分がそれである。

```
30 RWTexture2D *texXp;
31 RWTexture2D *texYp;                 // テクスチャオブジェクトへのポインタ
32 RWTexture2D *texZp;
33
34 float alpha = 0.5;
35 RGBA x[height][width];
36 RGBA y[height][width];              // 計算準備/計算結果の処理のための配列
37 RGBA z[height][width];
38 RGBA results[height][width];
```

initData()（図 12.8）は、

```
      sp->bindTextureR("tx",texXp);
      sp->bindTextureW(texZp);
```

の 2 行が削除された点を除けば、前章、11.5 節の initData()（図 11.41）と同じである。

その削除された 2 行は図 11.41 の関数 compute() で実行するように変更する。すなわち、 compute() では、

```
82    for(int n = 0; n < LOOPNUM/2; n++){          // 計算を繰り返す
       ...
90    }
```

のループで saxpy を繰り返す。ループ本体の前半：

```
83        sp->bindTextureR("tx", texXp);        // texXp を tx に結合
84        sp->bindTextureW(texZp);             // texZp を書き込み用に
85        sp->run(GL_POLYGON, NUM_POINTS);
```

では図 12.1 の計算経路をセットアップし、実行する。後半：

```
87        sp->bindTextureR("tx", texZp);        // texZp を tx に結合
88        sp->bindTextureW(texXp);             // texXp を書き込み用に
89        sp->run(GL_POLYGON, NUM_POINTS);
```

[1] 関数呼び出しのスタックフレームがオーバーフローするから。

では図 12.2 の計算経路をセットアップし、実行する。

図 11.41 の showResults() は、GPU による計算結果が CPU による計算結果と等しいか否かをチェックする。まず、

```
97      texXp->readData(results,width,height);    // 計算結果を CPU へ転送
```

で GPU での計算結果を CPU へ転送する。次に、

```
99      for(int n = 0; n < LOOPNUM/2; n++){       // CPU で同じ計算を行う
A0          for(int h = height-1; h >= 0; h--){
A1              for(int w = 0; w < width; w++){
A2                  z[h][w].r = alpha*x[h][w].r+y[h][w].r;
...
B0                  x[h][w].a = alpha*z[h][w].a+y[h][w].a;
B1              }
B2          }
B3      }
```

では、saxpy 計算を CPU で行う。そして、

```
B5      double error = 0.0;
B6      for(int h = height-1; h >= 0; h--){
B7          for(int w = 0; w < width; w++){
B8              error += fabs(x[h][w].r-results[h][w].r);
B9              error += fabs(x[h][w].g-results[h][w].g); // 誤算の積算
C0              error += fabs(x[h][w].b-results[h][w].b);
C1              error += fabs(x[h][w].a-results[h][w].a);
C2          }
C3      }
C4      printf("%30.25le\n",error);
```

では、GPU での計算結果と CPU での計算結果の差分の絶対値の和を求め、それを表示している。

12.1.2　フラグメントシェダープログラム

フラグメントシェーダープログラムは 11.5 節のそれと同じであるが、内容の確認のために、図 12.11 に再掲した。

12.1.3　実行結果

講義担当者の PC の実行結果は以下である。

```
0.0000000000000000000000000e+00
```

誤差は 20 桁の精度で全く認められななかった。

```
#include <time.h>                // 時間関連のシステム関数が定義されている
double start_time,end_time;      // 大域変数として宣言しておく

...

    start_time = clock();

    ここに計算部分のプログラムを置く

    end_time = clock();
    double total_time = (end_time-start_time)/CLOCKS_PER_SEC;
```

図 12.12: 実行時間の計測方法

12.2　実行時間の計測

この節では実行時間の計測方法について解説する。

まず、実行時間（秒）を測定するプログラム部分を図 12.12 のように考える。ここに、clock() はプログラム実行開始直後からの経過時間を整数値 [2] で求めるシステム関数である。マクロ値 CLOCKS_PER_SEC は clock() の戻り値の時間単位における 1 秒間の大きさである。よって clock() によって取得した経過時間を CLOCKS_PER_SEC で割ることで秒の単位の経過時間を知ることができる。

[2] 厳密には標準ヘッダーファイル <time.h>に定義されている clock_t 型である。通常、clock_t は 32bit 整数値で実装されている。

12.3　演算性能

12.3.1　基本式

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、その実行系のおおよその GFLOPS 値を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。

$$N(W, H, L) = 2 \cdot 4 \cdot W \cdot H \cdot L \tag{12.1}$$

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ーでは 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。

```
    double gflops = (2*4*width*height*LOOPNUM) / (total_time * 1e9);
```

なお、実行時間の計測には様々な方法がある。ここで述べた方法はシステム関数を用いる方法であるが、他にはプロセッサのタイマーレジスタの値を読み出す方法などもある [3]。

12.3.2　GPU の演算速度の計測

GPU の演算速度を知るには `initData()` の中の

```
70      texXp = new RWTexture2D(0, x, width, height);
71      texYp = new RWTexture2D(1, y, width, height);
72      texZp = new RWTexture2D(2, NULL, width, height);
```

の直前から `showResults()` の中の

```
97      texXp->readData(results,width,height);    // 計算結果を CPU へ転送
```

の直後までに上のプログラム片を追加して計測する。

12.3.3　CPU の演算速度の計測

CPU の演算速度を知るには `showResults()` の中の

```
99      for(int n = 0; n < LOOPNUM/2; n++){       // CPU で同じ計算を行う
          ...
    }
```

のループの前後を挟み込む。

[3] タイマーレジスタを用いる場合、クロックサイクル精度で経過時間を計測できる。しかし、この章の議論ではその精度は不要である。

12.3.4　演算速度計測プログラム

前節、前々節の方法に従って、時間計測のプログラム片を追加したものが以下のプログラムである。

154 ページの図 11.3、155 ページの図 11.4 と同じ

図 12.13: 各種クラスを定義するヘッダープログラム `All.h`

89 ページの図 7.2 と同じ

図 12.14: `ArrayBuffer` クラスの実装プログラム `Buffer.cpp`

156 ページの図 11.6 と同じ

図 12.15: 読み書き可能 2 次元テクスチャクラス `RWTexture2D` のクラスの実装プログラム `Textures.cpp`

157 ページの図 11.7 と同じ

図 12.16: `Shader` クラスの実装プログラム `Shader.cpp`（その他の部分は、89 ページの図 7.3、90 ページの図 7.4 と同じ）

```
01 #include "All.h"
02 #include <math.h>
03 #include <time.h>                    // 時間計測ライブラリのヘッダー引用
04
05 const int width = 100;
06 const int height = 100;
07
08 Shader *sp;
09
10 void initSystem(int argc, char *argv[])
11 {
12     glutInit(&argc,argv);
13     glutInitDisplayMode(GLUT_RGB|GLUT_SINGLE);
14     glutInitWindowSize(width,height);
15     glutCreateWindow("Test Window");
16     glClearColor(0.0,0.0,0.0,0.0);
17
18 #if defined(WIN32)
19     glewInit();
20 #endif
21
22     GLuint fb;
23     glGenFramebuffers(1, &fb);
24     glBindFramebuffer(GL_FRAMEBUFFER, fb);
25
26     sp = new Shader("shader.vert","shader.frag");
27     sp->use();
28 }
29
30 const int NUM_POINTS = 4;
31
32 RWTexture2D *texXp;
33 RWTexture2D *texYp;
34 RWTexture2D *texZp;
35
36 float alpha = 0.5;
37 RGBA x[height][width];
38 RGBA y[height][width];
39 RGBA z[height][width];
40 RGBA results[height][width];
41
42 double start_clock, end_clock;       // 計測開始時間/終了時間の保持
44
```

図 12.17: saxpy 計算を行うホストプログラム main.cpp（その 1、その 2 へ続く）

```
45 void initData()
46 {
47     Position2D pos[NUM_POINTS];
48
49     pos[0].x = -1.0; pos[0].y = -1.0;
50     pos[1].x = +1.0; pos[1].y = -1.0;
51     pos[2].x = +1.0; pos[2].y = +1.0;
52     pos[3].x = -1.0; pos[3].y = +1.0;
53
54     ArrayBuffer ab((float*)pos,2,NUM_POINTS);
55     sp->bindArrayBuffer("position",&ab);
56
57     sp->setFloat("width", width);
58     sp->setFloat("height", height);
59
60     sp->setFloat("alpha",alpha);
61
62     for(int h = height-1; h >= 0; h--){
63         for(int w = 0; w < width; w++){
64             x[h][w].r = h+w+0.1;
65             x[h][w].g = h+w+0.2;
66             x[h][w].b = h+w+0.3;
67             x[h][w].a = h+w+0.4;
68
69             y[h][w].r = h-w+0.1;
70             y[h][w].g = h-w+0.2;
71             y[h][w].b = h-w+0.3;
72             y[h][w].a = h-w+0.4;
73         }
74     }
75
76     start_clock = clock();                  // GPU 計算の時間計測開始
77
78     texXp = new RWTexture2D(0, x, width, height);
79     texYp = new RWTexture2D(1, y, width, height);
80     texZp = new RWTexture2D(2, NULL, width, height);
81
82     sp->bindTextureR("ty", texYp);
83 }
84
```

図 12.18: saxpy 計算を行うホストプログラム main.cpp（その 2、その 3 へ続く）

```
85 #define LOOPNUM 100
86
87 void compute(void)
88 {
89     for(int n = 0; n < LOOPNUM/2; n++){
90         sp->bindTextureR("tx", texXp);
91         sp->bindTextureW(texZp);
92         sp->run(GL_POLYGON, NUM_POINTS);
93
94         sp->bindTextureR("tx", texZp);
95         sp->bindTextureW(texXp);
96         sp->run(GL_POLYGON, NUM_POINTS);
97     }
98 }
99
A0 void showResult()
A1 {
A2     glFinish();
A3
A4     texXp->readData(results,width,height);
A5
A6     end_clock = clock();                    // GPU 計算の時間計測終了
A7     double GPUtotal = (end_clock-start_clock)/CLOCKS_PER_SEC;
                                                  // GPU 計算時間
A8     double GPUgflops = (2.0*4.0*width*height*LOOPNUM)/(GPUtotal*1e9);
                                                 // GPU の演算性能
A9     printf("GPU: %15.10lf %15.10lf\n", GPUtotal, GPUgflops);
B0
B1     start_clock = clock();                  // CPU 計算の時間計測開始
B2     for(int n = 0; n < LOOPNUM/2; n++){
B3         for(int h = height-1; h >= 0; h--){
B4             for(int w = 0; w < width; w++){
B5                 z[h][w].r = alpha*x[h][w].r+y[h][w].r;
B6                 z[h][w].g = alpha*x[h][w].g+y[h][w].g;
B7                 z[h][w].b = alpha*x[h][w].b+y[h][w].b;
B8                 z[h][w].a = alpha*x[h][w].a+y[h][w].a;
B9
C0                 x[h][w].r = alpha*z[h][w].r+y[h][w].r;
C1                 x[h][w].g = alpha*z[h][w].g+y[h][w].g;
C2                 x[h][w].b = alpha*z[h][w].b+y[h][w].b;
C3                 x[h][w].a = alpha*z[h][w].a+y[h][w].a;
C4             }
C5         }
C6     }
C7     end_clock = clock();                    // CPU 計算の時間計測終了
C8     double CPUtotal = (end_clock-start_clock)/CLOCKS_PER_SEC;
                                                  // CPU 計算時間
C9     double CPUgflops = (2.0*4.0*width*height*LOOPNUM)/(CPUtotal*1e9);
                                                 // CPU の演算性能
D0     printf("CPU: %15.10lf %15.10lf\n", CPUtotal, CPUgflops);
```

図 12.19: saxpy 計算を行うホストプログラム `main.cpp`（その 3、その 4 へ続く）

```
D1      double error = 0.0;
D2      for(int h = height-1; h >= 0; h--){
D3          for(int w = 0; w < width; w++){
D4              error += fabs(x[h][w].r-results[h][w].r);
D5              error += fabs(x[h][w].g-results[h][w].g);
D6              error += fabs(x[h][w].b-results[h][w].b);
D7              error += fabs(x[h][w].a-results[h][w].a);
D8          }
D9      }
E0      printf("ERROR: %30.25le\n",error);
E1 }
E2
E3 int main(int argc, char *argv[])
E4 {
E5      initSystem(argc,argv);
E6      initData();
E7      compute();
E8      showResult();
E9      return 0;
F0 }
```

図 12.20: saxpy 計算を行うホストプログラム `main.cpp`（その 4）

159 ページの図 11.10 と同じ

図 12.21: saxpy 計算を行うバーテックスシェーダープログラム `shader.vert`

187 ページの図 11.44 と同じ

図 12.22: saxpy 計算を行うフラグメントシェーダープログラム `shader.frag`

表 12.1: テクスチャサイズと実行速度

	GPU		CPU	
サイズ	時間	速度	時間	速度
4^2	1.11E+00	1.16E−04	3.10E−05	4.12E+00
8^2	1.11E+00	4.62E−04	1.39E−04	3.68E+00
16^2	1.11E+00	1.85E−03	4.62E−04	4.44E+00
32^2	1.12E+00	7.34E−03	1.70E−03	4.81E+00
64^2	1.11E+00	2.94E−02	6.69E−03	4.90E+00
128^2	1.11E+00	1.18E−01	2.70E−02	4.82E+00
256^2	1.12E+00	4.68E−01	1.10E−01	4.76E+00
512^2	1.13E+00	1.86E+00	5.21E−01	4.02E+00
1024^2	1.13E+00	7.44E+00	2.20E+00	3.82E+00
2048^2	1.29E+00	2.60E+01	8.71E+00	3.86E+00
4096^2	1.69E+00	7.96E+01	3.39E+01	3.96E+00

時間の単位は秒、速度の単位は GFLOPS

12.3.5　測定結果

表 12.1 は、講義担当者の PC[4] を用いて `LOOPNUM` 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実行速度を求めたものである。同じ条件でグラフ化したものが図 12.23 である。

次に、表 12.2 は、`width`、`height` を共に 4096 に固定し、`LOOPNUM` を 10、100、...、100000 と変えながら実行時間と実行速度を求めたものである。表 12.1 の最下行と表 12.2 の 3 行目は同じ条件での実験だが、得られた時間の 3 桁目の値が異なることに気づく。それがこの実験の精度と考えてほしい。本来ならば、同じ実験を多数回繰り返して数値を精密に評価すべきだが、ここではそれを端折っている。

12.3.6　演算性能の評価

まず、表 12.1、表 12.2 から GPU の最大演算性能は約 100GFLOPS、CPU のそれは約 4.8GFLOPS 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。

[4]円筒形の Mac Pro Late 2013、3.7 GHz Quad-Core Intel Xeon E5、12 GB 1866 MHz DDR3、AMD FirePro D700 6144 MB

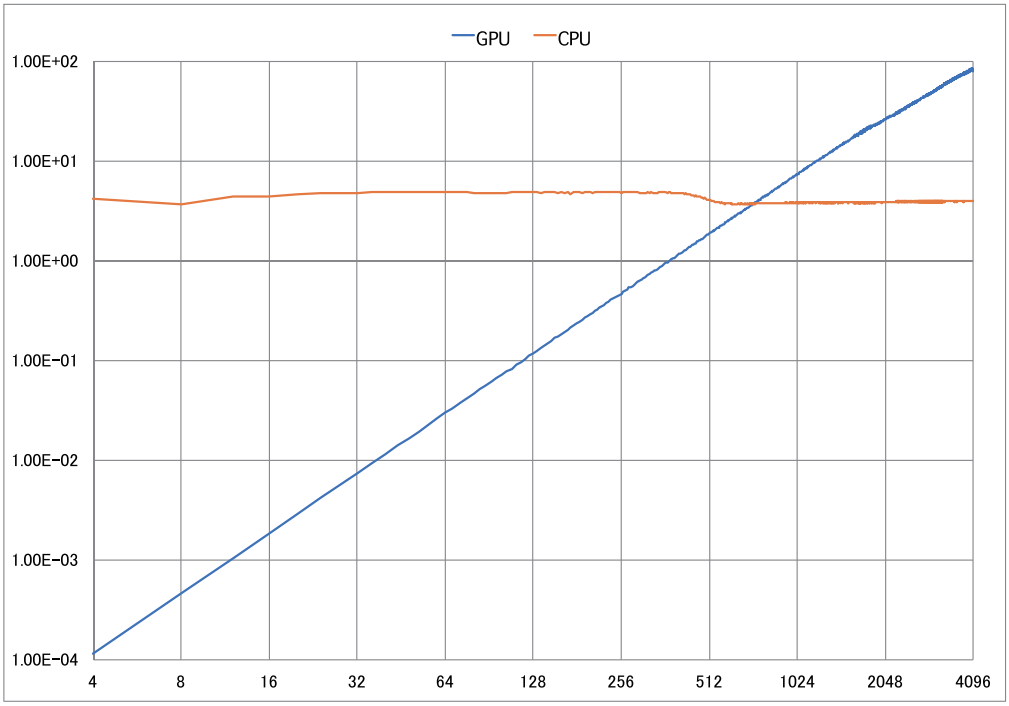

図 12.23: 表 12.1 のグラフ：横軸は画像の一辺の長さ、縦軸は演算性能（グラフ化に当たり、より詳細は実験を行なった。両軸とも対数目盛りであることに注意）

実験で用いた GPU のカタログ・ピーク性能は約 3.5TFLOPS（=3,500GFLOPS）であるから、それに比べればこの実験の 100GFLOPS はかなり小さいが、GPU のピーク性能を出し切るのは難しいと考えるべきである。

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

$$T = A + B \cdot N \tag{12.2}$$

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。

$$P = \frac{N}{T} = \frac{N}{A + B \cdot N} \tag{12.3}$$

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。

まず、GPU について表 12.2 の最下行では演算数が膨大であってオーバヘッド A_{GPU} が無視できると考えると、

$$\frac{N(4096, 100000)}{B_{\mathrm{GPU}} \cdot N(4096, 100000)} = 1.02 \times 10^{11}$$

表 12.2: 繰り返し回数と実行速度

	GPU		CPU	
回数	時間	速度	時間	速度
10	4.37E−01	3.07E+00	4.37E−01	3.07E+00
100	5.22E−01	2.57E+01	3.58E+00	3.74E+00
1000	1.68E+00	7.97E+01	3.41E+01	3.93E+00
10000	1.32E+01	1.02E+02	3.38E+02	3.97E+00
100000	1.32E+02	1.02E+02	3.41E+03	3.93E+00

時間の単位は秒、速度の単位は GFLOPS

よって

$$B_{\mathrm{GPU}} = 0.98 \times 10^{-11}$$

となる。

次に、同じ表 12.2 の 1 行目の結果に上の B_{GPU} を代入し、A_{GPU} を求めてみる。

$$\frac{N(4096, 10)}{A_{\mathrm{GPU}} + B_{\mathrm{GPU}} \cdot N(4096, 10)} = 3.07 \times 10^{9}$$

結果、

$$A_{\mathrm{GPU}} = 0.419$$

である。つまり、オーバーヘッドが約 0.4 秒かかることになる。コンピュータの実行時間において 0.4 秒は非常に大きい。この主な原因は、CPU から GPU への入力データの転送、GPU から CPU への計算結果の転送である。表 12.2 の実験ではトータルで 1GB 弱のデータを CPU と GPU の間でやりとりする必要があり [5]、この時間が非常に大きくなっていると考えられる。なお、A_{GPU} の計算では表 12.2 の 1 行目を用いたが、別の実験では転送データのサイズが異なるため A_{GPU} の値が変わりうることを注意しておく。

CPU については図 12.23 から演算性能はほぼ一定であると考えることができる。表 12.1、表 12.2 からその値を 4.0GFLOPS と置くと、以下の式が成り立つ。

$$\frac{N}{A_{\mathrm{CPU}} + B_{\mathrm{CPU}} \cdot N} = 4 \times 10^{9}$$

この式を解くと以下の通りである。

$$A_{\mathrm{CPU}} = 0.0, \qquad B_{\mathrm{CPU}} = 0.25 \times 10^{-9}$$

[5] 4096×4096 のテクスチャデータ — これは約 268M バイトのサイズになる — を 2 枚分、CPU から GPU へ送り、同じサイズのテクスチャデータ 1 枚分を GPU から CPU へ送るから、トータル 805MB である。

CPU であるから、オーバーヘッド A_{CPU} が無い（あるいは極めて小さいと考えられる）ことは納得できる。実験で用いた CPU のクロックサイクルは 3.7GHz であるから、クロックサイクル当たりの演算数 — これをしばしば IPC（instructions per cycle）と呼ぶ — は、4.0GFLOPS/3.7GHz = 1.08 となる。最近の CPU のピーク IPC は 2 以上であることが多いが、ピーク性能が出ることはまれであるから、1.08 はおおむね妥当な数値である。

12.4　より高速な計算

引き続き、前節の saxpy 計算を検討する。前節ではフラグメントシェーダー内（図 12.11）で 1 回の積和計算（詳しく言えば、4 並列の SIMD 型積和計算）

```
19      gl_FragColor = alpha*x+y;
```

を行い、これをピンポン計算で k 繰り返した。

前節では CPU、GPU 間のデータ転送のオーバーヘッドを論じたが、実はシェーダーの起動もコストが高い。そこで、1 回のシェーダーの計算に

```
    for(int k = 0; k < N; k++){
        x[i][j] = alpha*x[i][j]+y[i][j];
    }
```

を全て押し込む改造を行う。

プログラムは以下の通りである。

154 ページの図 11.3、155 ページの図 11.4 と同じ

図 12.24: 各種クラスを定義するヘッダープログラム `All.h`

89 ページの図 7.2 と同じ

図 12.25: `ArrayBuffer` クラスの実装プログラム `Buffer.cpp`

156 ページの図 11.6 と同じ

図 12.26: 読み書き可能 2 次元テクスチャクラス `RWTexture2D` のクラスの実装プログラム `Textures.cpp`

157 ページの図 11.7 と同じ

図 12.27: `Shader` クラスの実装プログラム `Shader.cpp`（その他の部分は、89 ページの図 7.3、90 ページの図 7.4 と同じ）

```
01 #include "All.h"
02 #include <math.h>
03 #include <time.h>
04
05 const int width = 4096;
06 const int height = 4096;
07
08 Shader *sp;
09
10 void initSystem(int argc, char *argv[])
11 {
12     glutInit(&argc,argv);
13     glutInitDisplayMode(GLUT_RGB|GLUT_SINGLE);
14     glutInitWindowSize(width,height);
15     glutCreateWindow("Test Window");
16
17 #if defined(WIN32)
18     glewInit();
19 #endif
20
21     GLuint fb;
22     glGenFramebuffers(1, &fb);
23     glBindFramebuffer(GL_FRAMEBUFFER, fb);
24
25     sp = new Shader("shader.vert","shader.frag");
26     sp->use();
27 }
28
29 const int NUM_POINTS = 4;
30
31 RWTexture2D *texXp;
32 RWTexture2D *texYp;
33 RWTexture2D *texZp;
34
35 float alpha = 0.5;
36 RGBA x[height][width];
37 RGBA y[height][width];
38 RGBA z[height][width];
39 RGBA results[height][width];
40
41 double start_clock, end_clock;
42
43 void initData()
44 {
45     Position2D pos[NUM_POINTS];
46
47     pos[0].x = -1.0; pos[0].y = -1.0;
48     pos[1].x = +1.0; pos[1].y = -1.0;
49     pos[2].x = +1.0; pos[2].y = +1.0;
```

図 12.28: saxpy 計算を行うホストプログラム main.cpp（その 1、その 2 へ続く）

```
50      pos[3].x = -1.0; pos[3].y = +1.0;
51
52      ArrayBuffer ab((float*)pos,2,NUM_POINTS);
53      sp->bindArrayBuffer("position",&ab);
54
55      sp->setFloat("width", width);
56      sp->setFloat("height", height);
57
58      sp->setFloat("alpha",alpha);
59
60      for(int h = height-1; h >= 0; h--){
61          for(int w = 0; w < width; w++){
62              x[h][w].r = h+w+0.1;
63              x[h][w].g = h+w+0.2;
64              x[h][w].b = h+w+0.3;
65              x[h][w].a = h+w+0.4;
66
67              y[h][w].r = h-w+0.1;
68              y[h][w].g = h-w+0.2;
69              y[h][w].b = h-w+0.3;
70              y[h][w].a = h-w+0.4;
71          }
72      }
73
74      start_clock = clock();
75
76      texXp = new RWTexture2D(0, x, width, height);
77      texYp = new RWTexture2D(1, y, width, height);
78      texZp = new RWTexture2D(2, NULL, width, height);
79
80      sp->bindTextureR("ty", texYp);
81  }
82
83  #define LOOPNUM 10000
84
85  void compute(void)
86  {
87      sp->setFloat("LOOPNUM",LOOPNUM);// 繰り返し回数をシェーダーに渡す
88
89      sp->bindTextureR("tx", texXp);        // texXp を読み込みに設定
90      sp->bindTextureW(texZp);              // texZp を書き込みに設定
91      sp->run(GL_POLYGON, NUM_POINTS);            // シェーダーの起動
92  }
93
94  void showResults()
95  {
96      glFinish();
97
98      texZp->readData(results,width,height);     // texZp から読み込み
99
```

図 12.29: saxpy 計算を行うホストプログラム main.cpp（その 2、その 3 へ続く）

```
A0      end_clock = clock();
A1      double GPUtotal = (end_clock-start_clock)/CLOCKS_PER_SEC;
A2      double GPUgflops = (2.0*4.0*width*height*LOOPNUM)/(GPUtotal*1e9);
A3      printf("GPU: %15.10lf %15.10lf\n",GPUtotal,GPUgflops);
A4
A5      start_clock = clock();
A6      for(int n = 0; n < LOOPNUM/2; n++){
A7          for(int h = height-1; h >= 0; h--)
A8          {
A9              for(int w = 0; w < width; w++)
B0              {
B1                  z[h][w].r = alpha*x[h][w].r+y[h][w].r;
B2                  z[h][w].g = alpha*x[h][w].g+y[h][w].g;
B3                  z[h][w].b = alpha*x[h][w].b+y[h][w].b;
B4                  z[h][w].a = alpha*x[h][w].a+y[h][w].a;
B5
B6                  x[h][w].r = alpha*z[h][w].r+y[h][w].r;
B7                  x[h][w].g = alpha*z[h][w].g+y[h][w].g;
B8                  x[h][w].b = alpha*z[h][w].b+y[h][w].b;
B9                  x[h][w].a = alpha*z[h][w].a+y[h][w].a;
C0              }
C1          }
C2      }
C3      end_clock = clock();
C4      double CPUtotal = (end_clock-start_clock)/CLOCKS_PER_SEC;
C5      double CPUgflops = (2.0*4.0*width*height*LOOPNUM)/(CPUtotal*1e9);
C6      printf("CPU: %15.10lf %15.10lf\n",CPUtotal,CPUgflops);
C7
C8      double error = 0.0;
C9      for(int h = height-1; h >= 0; h--)
D0      {
D1          for(int w = 0; w < width; w++)
D2          {
D3              error += fabs(x[h][w].r-results[h][w].r);
D4              error += fabs(x[h][w].g-results[h][w].g);
D5              error += fabs(x[h][w].b-results[h][w].b);
D6              error += fabs(x[h][w].a-results[h][w].a);
D7          }
D8      }
D9      printf("ERROR: %30.25le\n",error);
E0 }
E1
E2 int main(int argc, char *argv[])
E3 {
E4      initSystem(argc,argv);
E5      initData();
E6      compute();
E7      showResults();
E8      return 0;
E9 }
```

図 12.30: saxpy 計算を行うホストプログラム `main.cpp`（その 3）

159 ページの図 11.10 と同じ

図 12.31: saxpy 計算を行うバーテックスシェーダープログラム shader.vert

```
01 #version 120
02
03 uniform sampler2D tx;
04 uniform sampler2D ty;
05 uniform float alpha;
06
07 uniform float width;
08 uniform float height;
09
10 uniform float LOOPNUM;                                    // 繰り返し回数
11
12 void main(void)
13 {
14     vec2 texCoord = vec2(gl_FragCoord.x/width,
15                          gl_FragCoord.y/height);
16
17     vec4 x = texture2D(tx,texCoord);
18     vec4 y = texture2D(ty,texCoord);
19
20     for(int k = 0; k < int(LOOPNUM+0.5); k++)           // 繰り返し
21     {
22         x = alpha*x+y;                                  //saxpy 計算
23     }
24     gl_FragColor = x;              // 計算結果をフレームバッファへ出力
25 }
```

図 12.32: saxpy 計算を LOOPNUM 回行うフラグメントシェーダープログラム shader.frag

12.4.1　ホストプログラム

まず、図 12.29 の関数 compute() を以下のようにシェーダーの 1 回の起動に置き換える。なお、シェーダー内で繰り返し実行を行うため、LOOPNUM の値を uniform 変数としてシェーダーに渡す。LOOPNUM は int 型として受け渡す方がよいが、簡単のため[6]、ここでは LOOPNUM を浮動小数点数数値として受け渡す。

```
87      sp->setFloat("LOOPNUM",LOOPNUM);// 繰り返し回数をシェーダーに渡す
88
89      sp->bindTextureR("tx", texXp);          // texXp を読み込みに設定
90      sp->bindTextureW(texZp);                // texZp を書き込みに設定
91      sp->run(GL_POLYGON, NUM_POINTS);              // シェーダーの起動
```

上のプログラムでは、結果はポインタ textZp の指すテクスチャへ格納されるため、関数 showResults() の GPU から CPU へ計算結果を転送する箇所では、そのテクスチャからデータを読み込みように変更する。

```
98      texZp->readData(results,width,height);      // texZp から読み込み
```

12.4.2　フラグメントシェーダー

前節で

```
19      gl_FragColor = alpha*x+y;
```

であった部分を図 12.32 では

```
20      for(int k = 0; k < int(LOOPNUM+0.5); k++)             // 繰り返し
21      {
22          x = alpha*x+y;                                   //saxpy 計算
23      }
24      gl_FragColor = x;               // 計算結果をフレームバッファへ出力
```

に置き換える。ここに int(LOOPNUM+0.5) は、ホストプログラムから浮動小数点数数値として受け取った LOOPNUM を用いて正しい繰り返し回数を計算するための補正式である[7]。

[6] もしプログラミングの手間を厭（いと）わなければ、Shader::setFloat(float) と同様に Shader::setInt(int) を追加定義すればよい。

[7] たとえば、本来 100 を受け渡すつもりの LOOPNUM に小数点数 99.99999 が格納された場合、そのままでは小数点以下が切り捨てられ、99 回の繰り返しが行われる。丸め誤差を回避するため、0.5 を加えておく。

表 12.3: 繰り返し回数と実行速度

	GPU		CPU	
回数	時間	速度	時間	速度
10000	0.54E+00	2.37E+03	3.24E+02	3.95E+00

時間の単位は秒、速度の単位は GFLOPS

12.4.3　実行結果

図 12.3 は、4096 × 4096 の画面サイズについて 10,000 回繰り返した場合の実行結果である。この例では GPU が 2.37TFLOPS を達成している [8]。前節に書いたように、このプログラムを実行した PC の搭載する GPU の理論ピーク性能は 3TFLOPS である。よって、このプログラムでは理論ピーク性能の 80%の演算性能を達成したことになる。

この例は特殊であるが、工夫すれば理論ピークに近い性能を出すことができる。

[8] この実行例では GPU の計算時間が約 0.5 秒間である。この 0.5 秒の間、GPU はこのプログラムの実行に占有されるため、PC 上の他のプログラムの実行は見かけ上、ほとんど停止する。

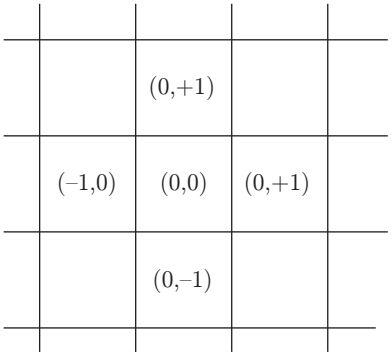

図 12.33: 画素の 4 近傍

12.5　テクスチャによる近傍計算

前節、前々節に紹介した計算は 2 次元データ内の個々の要素毎に行う計算であり、並列計算の典型的な例題である。要素毎（画素毎）に独立した計算が可能であるため、並列実行の効果は大きい。

実用的な計算では、隣接する要素を参照する場合もあるだろう。たとえば、以下は自データとその 4 近傍（図 12.33 参照）のデータによる平均値の計算である。この種の計算は画像フィルター等でしばしば見られる。

```
    for(int i = 0; i < N; i++){
        for(int j = 0; j < M; j++){
            z[i][j] = (x[i][j]+
                       x[i-1][j]+
                       x[i][j-1]+
                       x[i+1][j]+
                       x[i][j+1])/5.0;
        }
    }
```

この節ではこの計算を行う。なお、上のプログラムでは i が 0 のときの添え字 [i-1]＝[-1] 、i が N-1 のときの添え字 [i+1]＝[N] が不当な位置の配列アクセスとなり、不都合である。また、j の場合も同様である。これらの場合については後に述べる。

上の計算を行うプログラムを以下に示す。

154 ページの図 11.3、155 ページの図 11.4 と同じ

図 12.34: 各種クラスを定義するヘッダープログラム `All.h`

89 ページの図 7.2 と同じ

図 12.35: `ArrayBuffer` クラスの実装プログラム `Buffer.cpp`

156 ページの図 11.6 と同じ

図 12.36: 読み書き可能 2 次元テクスチャクラス `RWTexture2D` のクラスの実装プログラム `Textures.cpp`

157 ページの図 11.7 と同じ

図 12.37: `Shader` クラスの実装プログラム `Shader.cpp`（その他の部分は、89 ページの図 7.3、90 ページの図 7.4 と同じ）

```
01 #include "All.h"
02 #include <math.h>
03 #include <time.h>
04
05 const int width = 4096;
06 const int height = 4096;
07
08 Shader *sp;
09
10 void initSystem(int argc, char *argv[])
11 {
12     glutInit(&argc,argv);
13     glutInitDisplayMode(GLUT_RGB|GLUT_SINGLE);
14     glutInitWindowSize(width,height);
15     glutCreateWindow("Test Window");
16
17 #if defined(WIN32)
18     glewInit();
19 #endif
20
21     GLuint fb;
22     glGenFramebuffers(1, &fb);
23     glBindFramebuffer(GL_FRAMEBUFFER, fb);
24
25     sp = new Shader("shader.vert","shader.frag");
26     sp->use();
27 }
28
29 const int NUM_POINTS = 4;
30
31 RWTexture2D *texXp;
32 RWTexture2D *texZp;                    // texYp は用いないため、削除
33
34 RGBA x[height][width];
35 RGBA z[height][width];                 // y[][] は用いないため、削除
36 RGBA results[height][width];
37
38 double start_clock, end_clock;
39
40 void initData()
41 {
42     Position2D pos[NUM_POINTS];
43     pos[0].x = -1.0; pos[0].y = -1.0;
44     pos[1].x = +1.0; pos[1].y = -1.0;
45     pos[2].x = +1.0; pos[2].y = +1.0;
46     pos[3].x = -1.0; pos[3].y = +1.0;
47
48     ArrayBuffer ab((float*)pos,2,NUM_POINTS);
49     sp->bindArrayBuffer("position",&ab);
50
```

図 12.38: 近傍計算を行うホストプログラム main.cpp（その 1、その 2 へ続く）

```
51      sp->setFloat("width", width);
52      sp->setFloat("height", height);
53
54      for(int h = height-1; h >= 0; h--){
55          for(int w = 0; w < width; w++){
56              x[h][w].r = h+w+0.1;
57              x[h][w].g = h+w+0.2;
58              x[h][w].b = h+w+0.3;
59              x[h][w].a = h+w+0.4;
60          }
61      }
62
63      start_clock = clock();
64
65      texXp = new RWTexture2D(0, x, width, height);
66      texZp = new RWTexture2D(1, NULL, width, height);
67  }
68
69  #define LOOPNUM 10000
70
71  void compute(void)
72  {
73      for(int n = 0; n < LOOPNUM/2; n++){          // ピンポン計算
74          sp->bindTextureR("tx",texXp);
75          sp->bindTextureW(texZp);
76          sp->run(GL_POLYGON, NUM_POINTS);
77
78          sp->bindTextureR("tx",texZp);
79          sp->bindTextureW(texXp);
80          sp->run(GL_POLYGON, NUM_POINTS);
81      }
82  }
83
84  void showResults()
85  {
86      glFinish();
87
88      texXp->readData(results,width,height);
89
90      end_clock = clock();
91      double GPUtotal = (end_clock-start_clock)/CLOCKS_PER_SEC;
92      double GPUgflops = (5.0*4.0*width*height*LOOPNUM)/(GPUtotal*1e9);
                                                    // 2.0 を 5.0 へ変更
93      printf("GPU: %15.10lf %15.10lf\n",GPUtotal,GPUgflops);
94
```

図 12.39: 近傍計算を行うホストプログラム main.cpp（その 2、その 3 へ続く）

```
95      start_clock = clock();
96      for(int n = 0; n < LOOPNUM/2; n++){
97          for(int h = height-1; h >= 0; h--){
                // 検算、実行時間比較のため、近傍計算を CPU でも実行
98              int hm = (h == 0) ? height-1 : h-1 ;
99              int hp = (h == height-1) ? 0 : h+1 ;
A0              for(int w = 0; w < width; w++){
A1                  int wm = (w == 0) ? width-1 : w-1 ;
A2                  int wp = (w == width-1) ? 0 : w+1 ;
A3
A4                  z[h][w].r = (x[h][w].r+x[hp][w].r+x[hm][w].r+
A5                               x[h][wm].r+x[h][wp].r)/5.0;
A6                  z[h][w].g = (x[h][w].g+x[hp][w].g+x[hm][w].g+
A7                               x[h][wm].g+x[h][wp].g)/5.0;
A8                  z[h][w].b = (x[h][w].b+x[hp][w].b+x[hm][w].b+
A9                               x[h][wm].b+x[h][wp].b)/5.0;
B0                  z[h][w].a = (x[h][w].a+x[hp][w].a+x[hm][w].a+
B1                               x[h][wm].a+x[h][wp].a)/5.0;
B2              }
B3          }
B4          for(int h = height-1; h >= 0; h--){
B3              int hm = (h == 0) ? height-1 : h-1 ;
B4              int hp = (h == height-1) ? 0 : h+1 ;
B5              for(int w = 0; w < width; w++){
B6                  int wm = (w == 0) ? width-1 : w-1 ;
B7                  int wp = (w == width-1) ? 0 : w+1 ;
B8
B9                  x[h][w].r = (z[h][w].r+z[hp][w].r+z[hm][w].r+
C0                               z[h][wm].r+z[h][wp].r)/5.0;
C1                  x[h][w].g = (z[h][w].g+z[hp][w].g+z[hm][w].g+
C2                               z[h][wm].g+z[h][wp].g)/5.0;
C3                  x[h][w].b = (z[h][w].b+z[hp][w].b+z[hm][w].b+
C4                               z[h][wm].b+z[h][wp].b)/5.0;
C5                  x[h][w].a = (z[h][w].a+z[hp][w].a+z[hm][w].a+
C6                               z[h][wm].a+z[h][wp].a)/5.0;
C7              }
C8          }
C9      }
D0      end_clock = clock();
D1      double CPUtotal = (end_clock-start_clock)/CLOCKS_PER_SEC;
D2      double CPUgflops = (5.0*4.0*width*height*LOOPNUM)/(CPUtotal*1e9);
                                                    // 2.0 を 5.0 へ変更
D3      printf("CPU: %15.10lf %15.10lf\n",CPUtotal,CPUgflops);
D4
```

図 12.40: 近傍計算を行うホストプログラム main.cpp（その 3、その 4 へ続く）

```
D5      double error = 0.0;
D6      for(int h = height-1; h >= 0; h--){
D7          for(int w = 0; w < width; w++){
D8              error += fabs(x[h][w].r-results[h][w].r);
D9              error += fabs(x[h][w].g-results[h][w].g);
E0              error += fabs(x[h][w].b-results[h][w].b);
E1              error += fabs(x[h][w].a-results[h][w].a);
E2          }
E3      }
E4      printf("ERROR: %30.25le\n",error);
E5  }
E6
E7  int main(int argc, char *argv[])
E8  {
E9      initSystem(argc,argv);
F0      initData();
F1      compute();
F2      showResults();
F3      return 0;
F4  }
```

図 12.41: 近傍計算を行うホストプログラム main.cpp（その 4）

159 ページの図 11.10 と同じ

図 12.42: 近傍計算を行うバーテックスシェーダープログラム shader.vert

```
01 #version 120
02
03 uniform sampler2D tx;
04
05 uniform float width;
06 uniform float height;
07
08 void main(void)
09 {
10     vec2 texCoord00 = vec2((gl_FragCoord.x+0)/width,
11                            (gl_FragCoord.y+0)/height);  // ( 0, 0)
12
13     vec2 texCoordm0 = vec2((gl_FragCoord.x-1)/width,
14                            (gl_FragCoord.y+0)/height);  // (-1, 0)
15
16     vec2 texCoordp0 = vec2((gl_FragCoord.x+1)/width,
17                            (gl_FragCoord.y+0)/height);  // (+1, 0)
18
19     vec2 texCoord0m = vec2((gl_FragCoord.x+0)/width,
20                            (gl_FragCoord.y-1)/height);  // ( 0,-1)
21
22     vec2 texCoord0p = vec2((gl_FragCoord.x+0)/width,
23                            (gl_FragCoord.y+1)/height);  // ( 0,+1)
24
25     vec4 x00 = texture2D(tx,texCoord00);
26     vec4 xm0 = texture2D(tx,texCoordm0);
27     vec4 xp0 = texture2D(tx,texCoordp0);// それぞれの画素位置のデータ
28     vec4 x0m = texture2D(tx,texCoord0m);
29     vec4 x0p = texture2D(tx,texCoord0p);
30
31     gl_FragColor = (x00+xm0+xp0+x0m+x0p)/5.0;       // 平均値の計算
32 }
```

図 12.43: 近傍計算を行うフラグメントシェーダープログラム shader.frag

12.5.1　ホストプログラム

本質的には 12.1 節と変わりない。図 12.38〜図 12.41 のコメント箇所が変更箇所である。詳細は省略する。

12.5.2　フラグメントシェーダープログラム

近傍計算にはフラグメントシェーダープログラムを図 12.43 のように作ればよい。このプログラムの中でたとえば以下の行：

```
13      vec2 texCoordm0 = vec2((gl_FragCoord.x-1)/width,
14                             (gl_FragCoord.y+0)/height);  // (-1, 0)
```

は、添え字 [i-1][j] の計算である。この代入文の右辺の gl_FragCoord.x の値が 0 のときには (gl_FragCoord.x-1)/width は負値となり、メモリの不正アクセスが起きそうだが、この場合のテクスチャへのアクセスは 10.5 節で述べた通りであり、不正アクセスは起こらない。すなわち、テクスチャの設定において、

```
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
```

を指定したならば、[i-1][j] は [0][j] と同じ扱いになる（境界点、端点の値となる）。i が width-1 のときの [i+1][j] は [width-1][j] と同じ扱いになる。もし上の二行を指定しないならば（この節のプログラムはこれに当たる）、あるいは明示的に以下の指定：

```
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
```

を行うならば、i が 0 のときの [i-1][j] は [width-1][j] と同じ扱いになる。i が width-1 のときの [i+1][j] は [0][j] と同じ扱いになる。これは周期境界条件（periodic boundary condition）と呼ばれており、数値計算でしばしば設定される条件である。ここでの計算ではそれを従う。

12.5.3　実行速度

詳細な実験結果は省略するが、実行速度の様子は前節と同じ傾向にあった。

表 12.4 に、前節の表 12.2 の最下行と同じ条件において測定した実行時間と演算性能を示す。

表 12.4: 繰り返し回数と実行速度

	GPU		CPU	
回数	時間	速度	時間	速度
100000	2.11E+02	1.59E+02	9.57E+03	3.50E+00

時間の単位は秒、速度の単位は GFLOPS

フラグメントシェーダー内の演算数は、12.1 節のそれは 2 回（加算 1 回、乗算 1 回）であったのに対して、本節のそれは 5 回（加算が 4 回、除算 1 回）である。よって、図 12.39、図 12.40 の演算速度の計算式は

```
92      double GPUgflops = (5.0*4.0*width*height*LOOPNUM)/(GPUtotal*1e9);
...
D2      double CPUgflops = (5.0*4.0*width*height*LOOPNUM)/(CPUtotal*1e9);
```

と変更している。

12.5.4　演算性能の評価

まず、前節の四則演算数が式中に 2 回であったのに対して、本節のそれは 5 回であるから、単純計算では演算数が 2.5 倍に増えている。

GPU の演算性能は、前節が 102GFLOPS であったのに対してこの節は 159GFLOPS であるから、約 1.5 倍の性能を引き出したことになる。プログラムが演算を多く含めばその分だけシェーダー起動のオーバーヘッドの比率が減少することは前々節と前節の比較から分かっているから、1.5 倍はおおむね納得できる数値である。

CPU の演算性能は、前節が約 4GFLOPS であったのに対してこの節は 3.5GFLOPS である。CPU で数値が落ちた理由として、周期境界条件の取り扱いが考えられる。図 12.40 の

```
98              int hm = (h == 0) ? height-1 : h-1 ;
99              int hp = (h == height-1) ? 0 : h+1 ;
...
A1                  int wm = (w == 0) ? width-1 : w-1 ;
A2                  int wp = (w == width-1) ? 0 : w+1 ;
```

が境界部分の処理のための添え字の修正部分である。これら 3 項条件演算子 ？　：　が計算の負担になっていると思われる。実際、どんなハードウェア（GPU、CPU）においても条件判定は高速演算のネックになることが多い。

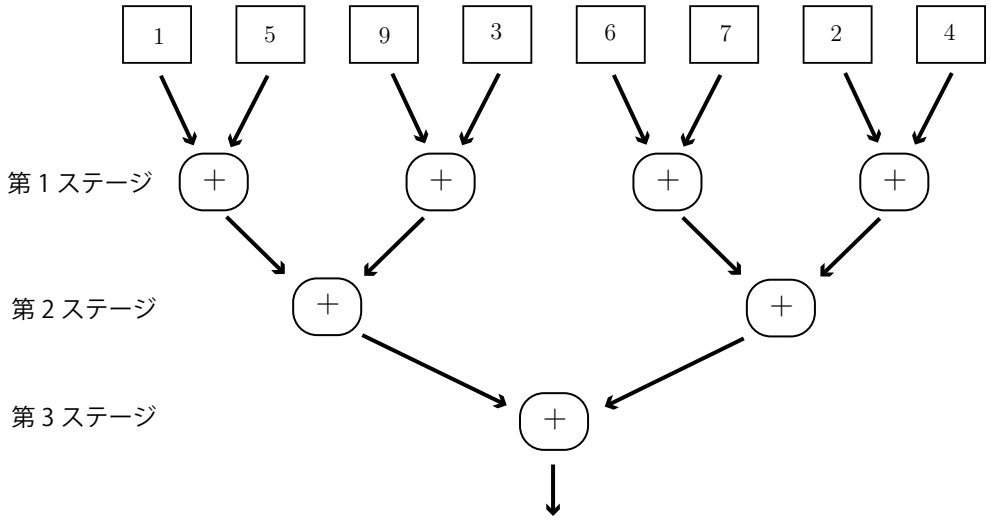

図 12.44: リダクション計算のカスケード実行

12.6　テクスチャによるリダクション計算

多数の値の合計値を求める計算、多数の値の最大/最小値を求めるような計算はリダクション（reduction、縮約）計算と呼ばれ、並列化の効果が出にくい種類の計算として知られている。

たとえば以下の合計値計算を考えよう。

$$1+5+9+3+6+7+2+4$$

この式の通常の計算順序は、加算演算子の左結合性から以下の通りと見なされる。

$$(((((((1+5)+9)+3)+6)+7)+2)+4)$$

しかし、加算は演算順序を変更できるため、以下の計算も可能である。

$$(((1+5)+(9+3))+((6+7)+(2+4)))$$

そうすると、$1+5$、$9+3$、$6+7$、$2+4$ が互いに独立した計算であるから、この四つの演算を並列実行できる。これを図式化したものが図 12.44 である。もし加算を並列計算できるならば、3 回分の計算時間で計 7 回の加算を実行できる。この形式の計算は、その形状からカスケード（cascade=滝）計算とも呼ばれている。

カスケード計算は 2 次元へ自然に拡張できる。この節ではそれを試みる。

まず、2 次元配列を用いる合計値計算を C 言語風に書けば以下の通りである。

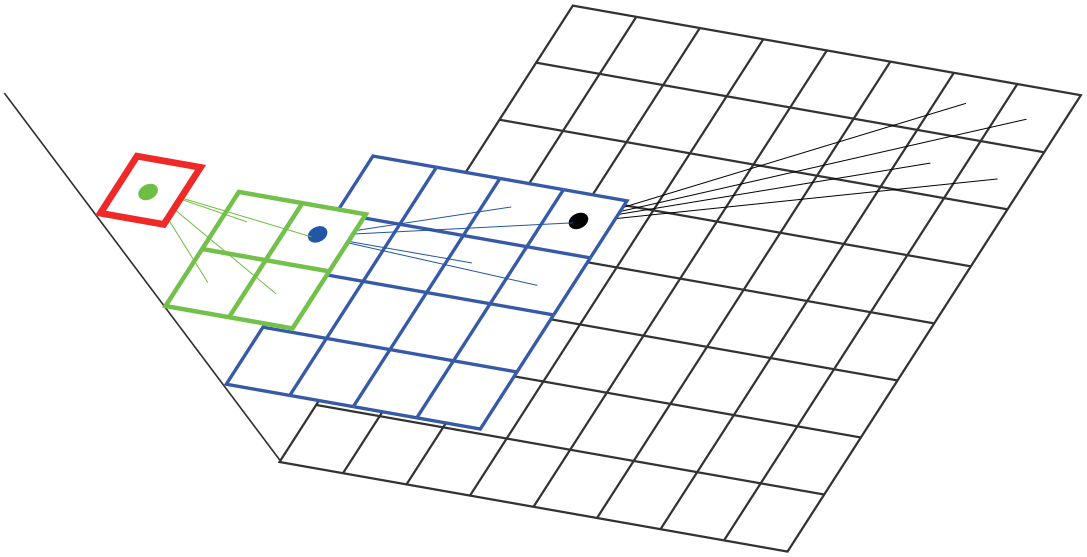

図 12.45: リダクション計算のカスケード実行

```
    float sum = 0.0;
    for(int j = 0; j < M; j++){
        for(int i = 0; i < N; i++){
            sum += z[i][j];
        }
    }
```

この計算も一見すると並列性がない。が、図 12.45 のように隣接する 4 要素毎に加算を行いながら、値を集約すれば、カスケード計算が可能である。1 次元のカスケード計算が二分木構造であったのに対して 2 次元のそれは四分木構造になる。

図 12.46 は、図 12.45 の最初の計算：最下層の黒いメッシュのテクスチャからひとつ上位の、サイズが 1/4 の青いメッシュを求めるステップを抜き出したものである。今、図 12.46 の青いメッシュの画素点 (m, n) に黒いメッシュの 4 点のデータの和を求めることを考える。(m, n) に対応する 4 点の位置は $(2m, 2n)$、$(2m+1, 2n)$、$(2m, 2n+1)$、$(2m+1, 2n+1)$、である。フラグメントシェーダーでは $(2m, 2n)$、$(2m+1, 2n)$、$(2m, 2n+1)$、$(2m+1, 2n+1)$ の位置の数値の和を求め、それを (m, n) の位置に書き込めばよい。ホストプログラムはこの計算を図 12.45 のように階層的に繰り返せばよい。この方法はデータサイズが大きい場合にはそれなりの高速化が期待できる。

以下、そのプログラムである。

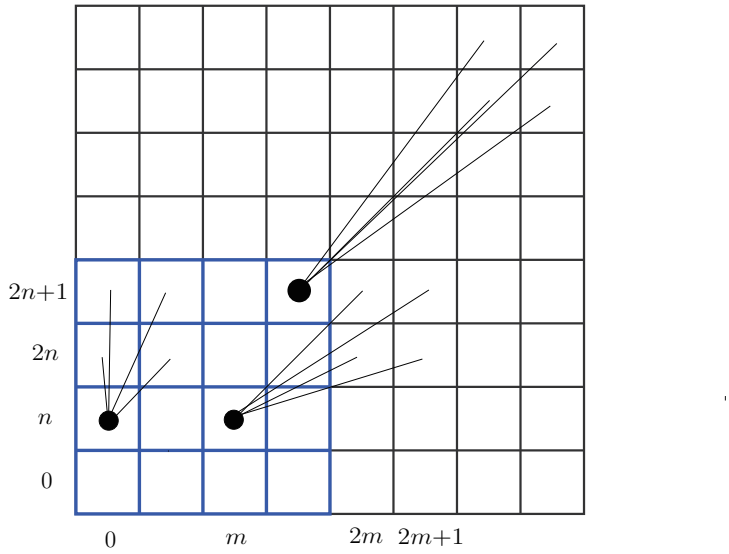

図 12.46: 2 次元リダクション計算のカスケード実行

```
01 #include <iostream>
02 using namespace std;
03 #include <stdio.h>
04 #include <stdlib.h>
05
06 #if defined(WIN32)
07 #  pragma comment(lib, "glew32.lib")
08 #  include "glew.h"
09 #  include "glut.h"
10 #  include "glext.h"
11 #elif defined(__APPLE__) || defined(MACOSX)
12 #  include <GLUT/glut.h>
13 #else
14 #  define GL_GLEXT_PROTOTYPES
15 #  include <GL/glut.h>
16 #endif
17
18 struct Position2D {
19     float x;
20     float y;
21 };
22
23 struct RGBA {
24     float r;
25     float g;
26     float b;
27     float a;
28 };
29
30 struct ArrayBuffer
31 {
32     GLuint    bufID;
33     int        size;
34
35     ArrayBuffer(float* data, int s, int n);
36     void moveData(float* data, int n);        // data を GPU へ再転送
37 };
38
39 struct RWTexture2D
40 {
41     GLuint  texID;
42     GLint   num;
43
44     RWTexture2D(int tnum, void* data, int w, int h);
45     void readData(void* data, int w, int h);
46 };
47
```

図 12.47: 各種クラスを定義するヘッダープログラム All.h（その 1、その 2 へ続く）

```
struct Shader
{
48      GLuint        program;
49
50      Shader(const char* vsn, const char* fsn);
51      void use();
52
53      void bindArrayBuffer(const char* vname, ArrayBuffer* ap);
54      void setFloat(const char* vname, float val);
55
56      void bindTextureR(const char* vname, RWTexture2D* tp);
57      void bindTextureW(RWTexture2D* tp);
58
59      void run(GLenum mode, int n);
60
61      GLuint compileProgram(GLenum type, const GLchar *file);
62      void buildProgram(const GLchar *vsfile, const GLchar *fsfile);
63 };
```

図 12.48: 各種クラスを定義するヘッダープログラム All.h（その 2）

```
01 ...
02 void ArrayBuffer::moveData(float* data, int n)
03 {
04      glBindBuffer(GL_ARRAY_BUFFER, bufID);
05      glBufferSubData(GL_ARRAY_BUFFER, 0, sizeof(float)*size*n, data);
                // data の内容を参照番号 bufID の GPU 上の配列バッファへ転送
06 }
07 ...
```

図 12.49: ArrayBuffer クラスの実装プログラム Buffer.cpp（その他の部分は 89 ページの図 7.2 と同じ）

156 ページの図 11.6 と同じ

図 12.50: 読み書き可能 2 次元テクスチャクラス RWTexture2D のクラスの実装プログラム Textures.cpp

157 ページの図 11.7 と同じ

図 12.51: Shader クラスの実装プログラム Shader.cpp（その他の部分は、89 ページの図 7.3、90 ページの図 7.4 と同じ）

```
01 #include "All.h"
02 #include <math.h>
03 #include <time.h>
04
05 const int width = 4096;
06 const int height = 4096;
07
08 Shader *sp;
09
10 ArrayBuffer* abp;                          // 頂点バッファへのポインタ
11
12 void initSystem(int argc, char *argv[])
13 {
14     glutInit(&argc,argv);
15     glutInitDisplayMode(GLUT_RGB|GLUT_SINGLE);
16     glutInitWindowSize(width,height);
17     glutCreateWindow("Test Window");
18
19 #if defined(WIN32)
20     glewInit();
21 #endif
22
23     GLuint fb;
24     glGenFramebuffers(1, &fb);
25     glBindFramebuffer(GL_FRAMEBUFFER, fb);
26
27     sp = new Shader("shader.vert","shader.frag");
28     sp->use();
29 }
30
31 const int NUM_POINTS = 4;
32 Position2D pos[NUM_POINTS];
33
34 RWTexture2D *texXp;
35 RWTexture2D *texZp;
36
37 RGBA x[height][width];
38
39 double start_clock, end_clock;
40
41 void initData()
42 {
43     pos[0].x = -1.0;     pos[0].y = -1.0;
44                          pos[1].y = -1.0;
45                           // 空白部分の配列要素は compute() で計算する
46     pos[3].x = -1.0;
47
48     abp = new ArrayBuffer((float*)pos, 2, NUM_POINTS);
49     sp->bindArrayBuffer("position", abp);
```

図 12.52: 近傍計算を行うホストプログラム main.cpp（その 1、その 2 へ続く）

```
50
51      sp->setFloat("width",width);
52      sp->setFloat("height",height);
53
54      for(int h = height-1; h >= 0; h--)
55       {
56          for(int w = 0; w < width; w++)
57          {
58              x[h][w].r = 1;
59              x[h][w].g = 2;
60              x[h][w].b = 3;
61              x[h][w].a = 4;
62
63          }
64      }
65
66      start_clock = clock();
67
68      texXp = new RWTexture2D(0, x, width, height);
69      texZp = new RWTexture2D(1, NULL, width, height);
70  }
71
72  void compute(void)
73  {
74      float offset;
75      for(int size = width; size > 1; )
76      {
77          offset = float(size)/float(width)-1.0;
78          pos[1].x = offset;
79          pos[2].x = offset;                  // 頂点データを動的に計算
80          pos[2].y = offset;
81          pos[3].y = offset;
82          abp->moveData((float*)pos,NUM_POINTS); // 頂点データの再設定
83
84          sp->bindTextureR("tx",texXp);           // texXp を読み込み
85          sp->bindTextureW(texZp);                // texZp を書き込み
86          sp->run(GL_POLYGON, NUM_POINTS);        // シェーダーの起動
87
88          size /= 2;                              // 描画領域を半分に
89
```

図 12.53: リダクション計算を行うホストプログラム main.cpp（その 2、その 3 へ続く）

```
90          offset = float(size)/float(width)-1.0;
91          pos[1].x = offset;
92          pos[2].x = offset;                    // 頂点データを動的に計
算
93          pos[2].y = offset;
94          pos[3].y = offset;
95          abp->moveData((float*)pos,NUM_POINTS); // 頂点データの再設定
96
97          sp->bindTextureR("tx",texZp);             // texZp を読み込み
98          sp->bindTextureW(texXp);                  // texXp を書き込み
99          sp->run(GL_POLYGON,NUM_POINTS);           // シェーダーの起動
A0
A1          size /= 2;                                // 描画領域を半分に
A2      }
A3 }
A4
A5 void showResults()
A6 {
A7      glFinish();
A8
A9      RGBA result;
B0      texXp->readData(&result,1,1); // 最終的な計算結果は float 型 1 個
B1
B2      end_clock = clock();
B3      double GPUtotal = (end_clock-start_clock)/CLOCKS_PER_SEC;
B4      double GPUgflops = (4.0*width*height)/(GPUtotal*1e9);
B5      printf("GPU: %15.10lf %15.10lf\n",GPUtotal,GPUgflops);
B6
B7      start_clock = clock();
B8      float result2r = 0.0,
B9            result2g = 0.0,
C0            result2a = 0.0;
C1      for(int h = height-1; h >= 0; h--)
C2      {
C3          for(int w = 0; w < width; w++)
C4          {
C5              result2r += x[h][w].r;
C6              result2g += x[h][w].g;
C7              result2b += x[h][w].b;
C8              result2a += x[h][w].a;
C9          }
D0      }
D1      end_clock = clock();
D2      double CPUtotal = (end_clock-start_clock)/CLOCKS_PER_SEC;
D3      double CPUgflops = (4.0*width*height)/(CPUtotal*1e9);
D4      printf("CPU: %15.10lf %15.10lf\n",CPUtotal,CPUgflops);
D5
D6 }
D7
```

図 12.54: リダクション計算を行うホストプログラム main.cpp（その 3、その 4 へ続く）

```
D8 int main(int argc, char *argv[])
D9 {
E0     initSystem(argc,argv);
E1     initData();
E2     compute();
E3     showResults();
E4     return 0;
E5 }
```

図 12.55: リダクション計算を行うホストプログラム main.cpp（その 4）

159 ページの図 11.10 と同じ

図 12.56: リダクション計算を行うバーテックスシェーダープログラム shader.vert

```
01 #version 120
02
03 uniform sampler2D tx;
04
05 uniform float width;
06 uniform float height;
07
08 void main(void)
09 {
10     vec2 texCoord = 2*(gl_FragCoord.xy-vec2(0.5));
11     vec2 delta = vec2(0.5/width, 0.5/height);
12     vec2 texCoord00 = vec2(texCoord.x/width,
13                            texCoord.y/height) + delta;
14     vec2 texCoord10 = vec2((texCoord.x+1)/width,
15                            texCoord.y/height) + delta;
16     vec2 texCoord01 = vec2(texCoord.x/width,
17                            (texCoord.y+1)/height) + delta;
18     vec2 texCoord11 = vec2((texCoord.x+1)/width,
19                            (texCoord.y+1)/height) + delta;
20
21     vec4 x00 = texture2D(tx,texCoord00);
22     vec4 x10 = texture2D(tx,texCoord10);  // 合算する 4 個のデータを
23     vec4 x01 = texture2D(tx,texCoord01);  // テクスチャから読み込み
24     vec4 x11 = texture2D(tx,texCoord11);
25
26     gl_FragColor = x00+x10+x01+x11;       // 合算
27 }
```

図 12.57: リダクション計算を行うフラグメントシェーダープログラム shader.frag

12.6.1　ArrayBuffer クラスの拡張

クラス定義のヘッダーファイル All.h は図 12.47、図 12.48 の通りである。

メンバー関数 ArrayBuffer::moveData() を新規に導入している。これは、配列バッファに格納する頂点データを CPU から GPU へ再転送するための関数である。図 12.45 で説明したように、リダクション計算では計算領域（描画領域）を狭めながら計算を繰り返す。その計算領域の再設定を ArrayBuffer::moveData() で行う。

ArrayBuffer::moveData() の実装は図 12.49 である。ここに OpenGL の関数呼び出し：

```
05      glBufferSubData(GL_ARRAY_BUFFER, 0, sizeof(float)*size*n, data);
                // data の内容を参照番号 bufID の GPU 上の配列バッファへ転送
```

は、すでに GPU 内に領域確保されている頂点バッファに頂点データを一斉転送する [9]。

12.6.2　ホストプログラム

クラス定義以外のホストプログラムは図 12.52〜図 12.54 である。

まず、この計算では、頂点バッファは initData() と compute() の両方からアクセスするため、以下のように大域化した。

```
10 ArrayBuffer* abp;                              // 頂点バッファへのポインタ
```

initSystem() は変更ない。

関数 initData() では頂点データの x 座標値、y 座標値が -1 の箇所のみ設定している。

```
43      pos[0].x = -1.0;      pos[0].y = -1.0;
44                            pos[1].y = -1.0;
45                             // 空白部分の配列要素は compute() で計算する
46      pos[3].x = -1.0;
```

というもの、図 12.45、図 12.46 に示したように、計算を行う矩形領域の左辺、底辺は固定のままだが、右辺は徐々に左側へ移動し、上辺は徐々に下方へ移動しながらリダクション演算を行うため、関数 initData() での初期設定では左辺、底辺に関する座標値のみを設定している。右辺、上辺の座標値は関数 compute() 内で設定する。また、ピンポン計算を行うため、テクスチャとシェーダープログラムの関連付けを行う関数呼び出し sp->bindTextureR()、sp->bindTextureW() も全て compute() へ移動させた。

リダクション計算の中核は関数 compute() である。

---

[9] 一斉転送ではなく、glMapBuffer()[10]、glUnmapBuffer()[11] を用いることも可能であるが、それについての解説は省略する。

図 12.46 を思い出そう。まず、画像全体を表す矩形領域は、

$$\text{左下の座標点：}(-1,\ -1),\quad \text{右上の座標点：}(1,\ 1)$$

である。それに対して、青いメッシュの矩形領域は、縦横の辺の長さが半分になるから

$$\text{左下の座標点：}(-1,\ -1),\quad \text{右上の座標点：}(0,\ 0)$$

である。さらに緑のメッシュの矩形領域は、

$$\text{左下の座標点：}(-1,\ -1),\quad \text{右上の座標点：}(0.5,\ 0.5)$$

である。これを繰り返し、その矩形領域に含まれる画素点が 1 個になるまで計算を繰り返す。関数 compute() の for ループ：

```
75     for(int size = width; size > 1; )
```

は、矩形のサイズを半分に減じながら実行する繰り返しである。

プログラム辺：

```
77          offset = float(size)/float(width)-1.0;
78          pos[1].x = offset;
79          pos[2].x = offset;                  // 頂点データを動的に計算
80          pos[2].y = offset;
81          pos[3].y = offset;
```

はその矩形領域の右辺と上辺の座標値の設定を行なっている部分である。

設定された頂点座標値は

```
82          abp->moveData((float*)pos,NUM_POINTS); // 頂点データの再設定
```

によって GPU へ転送する。

一旦、矩形が設定されたならば、関数呼び出し：

```
84          sp->bindTextureR("tx",texXp);           // texXp を読み込み
85          sp->bindTextureW(texZp);                // texZp を書き込み
86          sp->run(GL_POLYGON, NUM_POINTS);        // シェーダーの起動
```

は、テクスチャの設定、描画の実行を行う。この一連の実行で図 12.45 の黒いメッシュの 4 画素点の値の和は、対応する青いメッシュの画素点に格納される。

OpenGL による GPGPU の基本はピンポン計算であるから、関数 compute() の後半では、2 枚のテクスチャの役割を入れ替え、矩形の一片のサイズを半分に減じて、同様の計算を繰り返す。このピンポン計算を矩形に含まれる画素点が 1 個になるまで繰り返す。

関数 showResults() は、計算結果、実行時間を出力するプログラムである。

GPU の計算結果は GPU のメモリ内のテクスチャの左下の 1 画素分の求められているから、

```
A9      RGBA result;
B0      texXp->readData(&result,1,1); // 最終的な計算結果は float 型 1 個
```

によって CPU の 1 画素分の変数 result に転送している。CPU による合計計算は、単純な二重ループで実装した。

この計算に必要な総演算数は、4.0*width*height である [12]。

12.6.3　シェーダープログラム

バーテックスシェーダーのソースプログラムに変更はない。

フラグメントシェーダーのソースプログラムは図 12.57 である。

図 12.46 で見たように、フラグメントシェーダーの画素点が (m, n) であるとき、テクスチャ上の $(2m, 2n)$、$(2m+1, 2n)$、$(2m, 2n+1)$、$(2m+1, 2n+1)$ の位置の値の和を求めねばならない。そこで、座標値を保持する変数 texCoord、texCoord00、texCoord10、texCoord01、texCoord00 を用いて、テクスチャ上の座標を計算している。プログラム中の定数 0.5 は座標値を正確に計算するための調整用である。

12.6.4　計算結果

詳細は述べないが、GPU による計算結果と CPU による計算結果が異なる場合がある。

これは GPU のリダクション計算と CPU の単純な二重ループの計算の計算順序の違いに依る。float 型は有効桁数がせいぜい 7 桁程度しかないため、7 桁以上大きさの異なる数の加算では小さい方の数が無視される（情報落ちが起きる）。単純な二重ループによる積算ではそれが起きやすいが、階層的なリダクション計算ではそれが起きにくい。

12.6.5　実行速度

図 12.5 は、GPU による計算と CPU による計算の実行時間、演算性能である。テクスチャサイズは 4096^2 とした。前節までの計算では繰り返しを多数回（= LOOPNUM 回）行うことで演算性能を稼ぐことができたが、この節のリダクション演算では合計値を 1 回計算するだけの時間を測定した。

[12] 正確に言えば、4.0*(width*height-1) であるが、width、height が十分大きいとして -1 を無視した。

表 12.5: リダクション計算の実行時間と実行速度

GPU		CPU	
時間	速度	時間	速度
1.69E-01	4.00E-01	2.13E-02	3.15E+00

時間の単位は秒、速度の単位は GFLOPS

12.6.6　演算性能の評価

図 12.5 によれば、GPU の演算性能は約 0.4GFLOPS であるが、CPU のそれは 3.15GFLOPS であった。

総演算数が少ない上に単純な計算ではないことから、GPU の性能が CPU の約 1/10 であることは仕方がない。むしろ、1/10 で済んだことを良しとすべきである。特殊な目的でない限り、リダクション計算はそれほど頻度の高い計算ではなく、多用されることはない。リダクション計算が全体の計算の中でそれほどネックにならないことを確認できたと考えるべきである。

第13章　OpenCLによる並列計算

前章、前々章ではOpenGLを用いたGPGPUを見てきた。2000年以降、この手法での大規模計算が評判となり、徐々に広がりをみせるが、如何せん、OpenGLの知識がないとプログラミングができない難しさがあった。そこで、OpenGLの知識を前提としない、しかもより広範な応用を見据えたGPGPU向けプログラム開発環境が作られることになる。CUDAやOpenCLがそれである。

この講義テキストは、OpenCLの簡単なプログラムを概説して終わることとする。

13.1　saxpy計算

この節では、11.5節で解説した簡単なsaxpy計算のプログラムをOpenCLで記述した場合を紹介する。プログラムの詳細な解説はしない。ここで紹介するプログラムは、「日曜研究室」さん（`https://peta.okechan.net/blog`)）の記事「いまXcodeでOpenCLをはじめる多分いちばん簡単かもしれない方法」を参考に、この講義テキストに合わせて改変したものである。プログラムの読みやすさのため、エラー処理は全て削除した[1]。

図13.1〜図13.5がOpenCLのホストプログラムである。前章までのプログラムに合わせるため、関数`initSystem()`、`initData()`、`compute()`、`showResults()`にOpenCLの各動作を割り振ってみた。これは、11.5節の図11.40〜図11.42のOpenGLのホストプログラムに相当する。

図13.6はOpenCLのカーネルプログラムである。これは、11.5節の図11.44のOpenGLのフラグメントシェーダープログラムに相当する。

図13.1〜図13.5、図13.6を眺めて気づくことは、OpenCLは比較的容易にGPGPUのプログラムを書けるように工夫されているものの、実際には図11.40〜図11.42、図11.44のプログラムに比べてプログラムサイズはそう変わらない点である。GPGPUを行うために最低限準備すべき処理はOpenGL、OpenCLでほとんど共通しており、実は準備の煩雑さはそれほど変わらないのである。

[1] よって、警告、エラーが全く出ないため、ここのプログラムを基点としてOpenCLのプログラムを開発するのには無理がある。

OpenGL と OpenCL の違いは、ここのプログラムでは見ることはできないが、OpenCL には GPU を使いこなすための様々なチューニングパラメータが用意されており、それを適切に設定することで、OpenGL では達成できないレベルの高速化が可能になる点である。それについて述べる余裕はないので、興味を持った人は各自で調べてほしい。

```
01 #include <iostream>
02 #include <vector>
03 #include <OpenCL/opencl.h>
04 using namespace std;
05
06 #define PLATFORM_MAX 4
07 #define DEVICE_MAX 4
08 cl_int err = CL_SUCCESS;
09
10 cl_device_id devices[DEVICE_MAX];
11 cl_context ctx;
12 cl_program program;
13 cl_kernel kernel;
14 cl_command_queue q;
15
16 void initSystem()
17 {
18     // プラットフォーム一覧を取得，見つかった情報を印字
19     cl_platform_id platforms[PLATFORM_MAX];
20     cl_uint platformCount;
21     clGetPlatformIDs(PLATFORM_MAX,
22                      platforms,
23                      &platformCount);
24     for (int i = 0; i < platformCount; i++) {
25         char vendor[100] = {0};
26         char version[100] = {0};
27         clGetPlatformInfo(platforms[i],
28                           CL_PLATFORM_VENDOR,
29                           sizeof(vendor),
30                           vendor,
31                           nullptr);
32         clGetPlatformInfo(platforms[i],
33                           CL_PLATFORM_VERSION,
34                           sizeof(version),
35                           version,
36                           nullptr);
37         cout << "Platform id: " << platforms[i]
38              << ", Vendor: " << vendor
39              << ", Version: " << version << endl;
40     }
41
42     // デバイス一覧を取得、見つかった情報を印字
43     cl_uint deviceCount;
44     clGetDeviceIDs(platforms[0],
45                    CL_DEVICE_TYPE_GPU,
46                    DEVICE_MAX,
47                    devices,
48                    &deviceCount);
```

図 13.1: saxpy 計算を行う OpenCL のホストプログラム `main.cpp`（その 1、その 2 へ続く）

```
49      cout << deviceCount << " device(s) found." << endl;
50      for (int i = 0; i < deviceCount; i++) {
51          char name[100] = {0};
52          size_t len;
53          clGetDeviceInfo(devices[i],
54                          CL_DEVICE_NAME,
55                          sizeof(name),
56                          name,
57                          &len);
58          cout << "Device id: " << i
59               << ", Name: " << name << endl;
60      }
61
62      // コンテキストの作成
63      ctx = clCreateContext(nullptr,
64                            1,
65                            devices,
66                            nullptr,
67                            nullptr,
68                            &err);
69
70      // コンパイル済み cl プログラムの読み込み
71      const char* bitcode_path = "OpenCL/kernel.cl.gpu_32.bc";
72      size_t len = strlen(bitcode_path);
73      program = clCreateProgramWithBinary(
74                      ctx,
75                      1,
76                      devices,
77                      &len,
78                      (const unsigned char**)&bitcode_path,
79                      nullptr,
80                      &err);
81
82      // プログラムのビルド
83      clBuildProgram(program,
84                     1,
85                     devices,
86                     nullptr,
87                     nullptr,
88                     nullptr);
89
90      // カーネルの作成
91      kernel = clCreateKernel(program, "saxpy", &err);
92  }
93
94  const int N = 1000;
95
```

図 13.2: saxpy 計算を行う OpenCL のホストプログラム `main.cpp`（その 2、その 3 へ続く）

```
96 cl_mem device_memx;
97 cl_mem device_memy;
98 cl_mem device_memz;
99
A0 vector<float> x(N);
A1 vector<float> y(N);
A2 vector<float> z(N);
A3
A4 void initData()
A5 {
A6     // データを用意
A7     for (int i = 0; i < N; i++)
A8     {
A9         x[i] = float(i);
B0         y[i] = float(i+1);
B1     }
B2
B3     // デバイスメモリを確保しつつデータをコピー
B4     device_memx =
B5         clCreateBuffer(ctx,
B6                        CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
B7                        sizeof(float) * N,
B8                        x.data(),
B9                        &err);
C0     device_memy =
C1         clCreateBuffer(ctx,
C2                        CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
C3                        sizeof(float) * N,
C4                        y.data(),
C5                        &err);
C6     device_memz =
C7         clCreateBuffer(ctx,
C8                        CL_MEM_WRITE_ONLY | CL_MEM_ALLOC_HOST_PTR,
C9                        sizeof(float) * N,
D0                        NULL,
D1                        &err);
D2
D3     // カーネルの引数をセット
D4     clSetKernelArg(kernel, 0, sizeof(cl_mem), &device_memx);
D5     clSetKernelArg(kernel, 1, sizeof(cl_mem), &device_memy);
D6     clSetKernelArg(kernel, 2, sizeof(cl_mem), &device_memz);
D7     float alpha = 0.5;
D8     clSetKernelArg(kernel, 3, sizeof(float), &alpha);
D9 }
E0
```

図 13.3: saxpy 計算を行う OpenCL のホストプログラム `main.cpp`（その 3、その 4 へ続く）

```
E1 void compute()
E2 {
E3     // コマンドキューの作成
E4     q = clCreateCommandQueue(ctx, devices[0], 0, &err);
E5
E6     // カーネルの実行
E7     size_t global = N;
E8     clEnqueueNDRangeKernel(q,
E9                            kernel,
F0                            1,
F1                            nullptr,
F2                            &global,
F3                            nullptr,
F4                            0,
F5                            nullptr,
F6                            nullptr);
F7 }
F8
F9 void showResults(){
G0     // 結果を読み込み
G1     clEnqueueReadBuffer(q,
G2                         device_memz,
G3                         CL_TRUE,
G4                         0,
G5                         sizeof(float) * N,
G6                         z.data(),
G7                         0,
G8                         nullptr,
G9                         nullptr);
H0
H1     // 結果の印字
H2     for (int i = 0; i < N; i++)
H3     {
H4         cout << i << " : " << z[i] << endl;
H5     }
H6 }
H7
```

図 13.4: saxpy 計算を行う OpenCL のホストプログラム `main.cpp`（その 4、その 5 へ続く）

```
H8 int finSystem()
H9 {
I0     // コマンドキューの解放
I1     clReleaseCommandQueue(q);
I2
I3     // デバイスメモリを解放
I4     clReleaseMemObject(device_memx);
I5     clReleaseMemObject(device_memy);
I6
I7     // カーネルの解放
I8     clReleaseKernel(kernel);
I9
J0     // プログラムの解放
J1     clReleaseProgram(program);
J2
J3     // コンテキストの解放
J4     clReleaseContext(ctx);
J5
J6     return EXIT_SUCCESS;
J7 }
J8
J9 int main(int argc, const char * argv[])
K0 {
K1     initSystem();
K2     initData();
K3     compute();
K4     showResults();
K5     return finSystem();
K6 }
```

図 13.5: saxpy 計算を行う OpenCL のホストプログラム `main.cpp`（その 5）

```
01 kernel void saxpy(__global float* x,
02                   __global float* y,
03                   __global float* z,
04                   const float alpha)
05 {
06     int i = get_global_id(0);
07
08     z[i] = alpha*x[i]+y[i];
09 }
```

図 13.6: saxpy 計算を行う OpenCL のカーネルプログラム `kernel.cl`

第14章　まとめ

この講義では、OpenGL/GLSL を用いた描画の話から始まった。シェーダー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、シェーダー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 や OpenCL といった GPGPU 専用の環境が利用できるから、この講義テキストの内容が GPGPU の主流になることはなく、むしろ古い技術とみなされる。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ひとつの答えになっていると考える。