〇〇くんのために一所懸命書いたものの
結局〇〇くんの卒業に間に合わなかった
GLFW による OpenGL 入門

目次

第1章 はじめに

1.1 本書の目的

本書は OpenGL (http://www.opengl.org) と呼ばれるグラフィックス表示のための**アプリケーションプログラムインタフェース** (Application Program Interface, API) を使用して、グラフィックスアプリケーションを作成する方法について解説します。

ただし、OpenGL 自体はグラフィックスハードウェアを制御する機能しか持っていません。グラフィックスアプリケーションを作成するには、**コンピュータグラフィックス** (Computer Graphics, CG) の理論の知識も必要になります。そのため実際のアプリケーションソフトウェア開発では、CG の各種の理論や手法を実装した**ミドルウェア**がよく用いられます。

しかし、本書では直接 OpenGL の API を使ってアプリケーションソフトウェアを作成する方法を解説します。このため本書では、基礎的な CG の理論の解説も行います。これは、仮にミドルウェアを用いてグラフィックスアプリケーションを開発するとしても、それを使いこなすためには CG の理論の理解が不可欠だと考えるからです。

本書が想定する読者は、学校の授業などで C あるいは C++ を勉強したものの、自分では実際にプログラムを組んだ経験があまりないという、そう、そこの君！○○君 (実在の人物) のことだよ。そのため、本書ではプログラミング言語として C++ を用いますが、使用する C++ の機能は構造体レベルの簡単なクラス定義と iostream や vector などの基本的なライブラリにとどめ、いわゆる “better C” として C++ を使用します。

1.2 OpenGL

三次元コンピュータグラフィックス (3D Computer Graphics, 3D CG) を使った画面表示は、現在ではパソコンやゲーム機のみならず、スマートフォンやカーナビなど、幅広い領域で利用されています。しかし 3D CG の処理は、現在のパソコンの CPU の高い計算能力をもってしても、負荷の高いものになります。特に、リアルタイム 3D CG によって快適な応答性能を得るには、やはり専用ハードウェアによる支援が不可欠になります。

この専用ハードウェア、いわゆる**グラフィックスハードウェア**は、パソコンの CPU に匹敵するか、それ以上の複雑さを持っています。したがって、これを有効に活用するために、グラフィックスハードウェアの機能を抽象化し、アプリケーションプログラムがグラフィックス表示を行うために必要な機能を整理する、高機能な**グラフィックスライブラリ**が使用されます。

グラフィックスライブラリは、通常コンピュータのハードウェア全体を制御するオペレーティングシステム (OS) の機能の一部として提供されます。アプリケーションプログラムはこれを介してグラフィックスハードウェアを制御するので、このようなグラフィックスライブラリはアプリケーションプログラムインタフェース、API と呼ばれます。パソコンの OS として最も普及している Microsoft 社の Windows には、一般的な二次元のグラフィックス表示を行う Graphics Device Interface (GDI) や、二次元グラフィックスおよびリアルタイム 3D CG の表示機能などを包含した DirectX (3D CG 部分は Direct3D) という API が用意されています。

ところが Windows には、これらとは別に、OpenGL と呼ばれるグラフィックス表示用の API が用意されています。OpenGL は Microsoft 社が DirectX を用意する以前から Windows に組み込まれていた、リアルタイム 3D CG に対応したグラフィックス API です。

この OpenGL は、最初シリコングラフィックス社 (Silicon Graphics, Inc.、後に Silicon Graphics International Corp., SGI) により開発されました。同社はもともと IRIX と呼ばれる UNIX 系 OS を搭載した、エンジニアリングワークステーション (Engineering Work Station, EWS) と呼ばれるコンピュータのメーカーでした。OpenGL は同社の EWS のグラフィックス表示に用いられていたグラフィックスライブラリ (GL、OpenGL と区別するために IRIS GL と呼ばれることがあります) を、プラットホーム (ハードウェアや OS などのコンピュータの基盤) に依存する部分を分離して再実装したものです。その後 OpenGL はオープンソースソフトウェアとして公開され、現在は Khronos Group (http://www.khronos.org) によって規格が策定されています。

EWS は主にコンピュータ支援設計 (Computer Aided Design, CAD) などの技術的用途に用いられるコンピュータです。しかし、パソコンの性能の向上により、この目的にもパソコンが使用されるようになりました。その結果、EWS で動作していたアプリケーションソフトウェアをパソコンに移行する必要性が生じ、そのために Windows 上にも OpenGL が移植されました。

当時はこれら以外のグラフィックスライブラリもいくつか存在しましたが、パソコン用 OS の Windows による寡占化が進んだ結果、グラフィックスライブラリも DirectX と OpenGL の二つ

以外は実質的に淘汰されてしまいました。このうち DirectX は Microsoft 社の専有物のため、OS として Windows を採用しているパソコン以外で共通に使用できるグラフィックスライブラリは、現在では事実上 OpenGL (およびその後継の Vulkan[1]) しかありません。しかし、これは言い換えれば、Windows パソコンを含むコンピュータ関連機器のほとんどが、3D CG 表示用の API として OpenGL (または、組み込み機器向けの OpenGL ES) を採用しているとも言えます。

OpenGL がこのように広く使われるようになった背景には、もちろん OpenGL しか選択肢がなかったこともありますが、その仕様がプラットホームから独立していることも大きな要因でしょう。これにより OpenGL はさまざまな機器に導入されました。もともと OpenGL が動作していた IRIX などの UNIX 系 OS やオープンソースで開発されている Linux の画面表示に用いられる X Window System に組み込まれているほか、Apple 社のパソコンの OS である Mac OS X にも OpenGL が導入されています。またパソコンに限らず、スマートフォンの OS である Apple 社の iOS や Google 社の Android も、組み込みシステム向けの OpenGL ES を採用しています。

1.3 GLFW

1.3.1 ツールキット

OpenGL はプラットホームに依存しないグラフィックス API ですが、これをアプリケーションプログラムから使用するには、やはりプラットホームごとに異なる「お膳立て」が必要になります。しかし、その手順もそれなりに面倒なものになるため、それを包み隠して簡単に使えるようにした**ツールキット**がいくつも提案されています。

中でも GLUT (OpenGL Utility Toolkit) は、OpenGL を開発した Silicon Graphics のエンジニア (当時) が作った、使いやすいツールキットです。また、この GLUT はマルチプラットホームに対応しているため、これを使ったソースプログラムは Unix / Linux、Windows、Mac OS X の間で共通にすることができます。GLUT は OpenGL の初期の頃に作られたものですが、OpenGL の学習や OpenGL を使った簡単なプログラムの作成を手軽に始めることができます。

しかし、オリジナルの GLUT は既に長い間メンテナンスされていません。代わりに GLUT 互換の freeglut (http://freeglut.sourceforge.net) というツールキットが開発されています。しかし、少なくとも本書の執筆時点では、これらは Mac OS X には対応していません。また、Mac OS X には以前から標準で GLUT が搭載されていましたが、これは OpenGL 2.1 にしか対応しておらず、Mac OS X のバージョン 10.7 (Lion) 以降で使用可能になった OpenGL 3.2 の Core Profile[2] や、10.9 (Mavericks) 以降で使用可能になった OpenGL 4.1 を (公式には) 使用することができません[3]。加えて、この 10.9 では、ついに GLUT の使用自体が非推奨になりました。

---

[1] 2016 年 2 月に OpenGL の後継の API である Vulkan (ヴァルカン) が Khronos Group により策定されました。
[2] OpenGL の過去のバージョンとの互換性を維持しない設定。
[3] gl3w というツールを使えば GLUT で OpenGL バージョン 3.2 以降の機能を使用できます。

1.3.2 GLFW の概要

GLUT が使えないとなると、代わりのものを探す必要があります。OpenGL に対応していて GLUT と同様にマルチプラットホームで使用できるツールキットには、FLTK (http://www.fltk.org/) をはじめ SDL (https://www.libsdl.org/)、Qt (https://www.qt.io/)、などさまざまなものがあります。中でも Qt (キュート) は非常に高機能なツールキットであり、CG 関連のいくつかの主要なアプリケーションソフトウェアがこれを使って開発されています。これら以外にも、C++ に対応した openFrameworks (http://openframeworks.cc/) や Cinder (https://libcinder.org/) も非常に高機能なツールキットです。これらは “Creative Coding”、すなわち、表現としての「創造的なプログラミング」の領域で盛んに利用されています。

しかし、OpenGL の学習のためにあれこれ試したり、ちょっとしたプログラムを書いたりするには、Qt などはちょっと大きすぎる気がします。そこで、GLUT の代わりになる簡単で小さなツールキットとして、GLFW (http://www.glfw.org/) があります。

上記の GLFW のホームページでは、GLFW はウィンドウを作成し、OpenGL のコンテキストを作って、入力 (デバイス) を管理する、無料の、オープンソースの、マルチプラットホームのライブラリであると説明されています。ライセンスには zlib/libpng license を採用しています。

1.3.3 GLFW の特徴

GLFW は次のような特徴を持っています。

- **非常にコンパクトである**

OpenGL と組み合わせて使うツールキットの中では非常にコンパクトであり、OpenGL のウィンドウを管理するための最小限の機能を提供しています。

- **マルチプラットホームである**

GLUT と同様に Windows / Mac OS X / Linux でソースプログラムを共通化できます。

- **OpenGL のバージョンやプロファイルが指定できる**

Mac OS X バージョン 10.7 (Lion) 以降では OpenGL のバージョン 3.2 の Core Profile、10.9 (Mavericks) 以降では OpenGL のバージョン 4.1 を指定することができます。

- **最初からダブルバッファリングになっている**

ダブルバッファリングはアニメーション表示を行うための必須の機能ですが、GLFW ではこれが標準で有効になっています。GLFW のバージョン 3.1 からは、シングルバッファのモードに切り替えることもできるようになっています。

● イベントループを自分で書く

OpenGL などによるグラフィックス表示では、OS からの描画要求 (**再描画イベント**) にもとづいて、くり返し描画処理を行う必要があります。このくり返しを**イベントループ**といいます。このループにおいて描画処理を行った後に次の再描画イベントが発生するまで待つようにすれば、マウスやキーボードの操作によって画面表示を更新する対話的なアプリケーションソフトウェアを作成することができます。一方、描画処理を一定の時間間隔で行うことにより、画面表示が時間とともに更新されるアニメーション表示を行うことができます。

● ポーリング方式とコールバック方式のどちらにも対応している

GLFW のプログラミングは、どのようなイベント (マウスのドラッグやキーボードのタイプなど) が発生したのかをイベントループの中で調べて (ポーリング) 対応する処理を記述する方式が基本です。しかし、必要に応じてイベントごとに実行する関数 (コールバック関数) を登録する方式で記述することもできます。

● マルチウィンドウやマルチモニタに対応している

OpenGL のレンダリングコンテキストを管理する機能を持っており、マルチウィンドウやマルチモニタに対応したアプリケーションソフトウェアを作成することができます。

● 入力デバイスの取り扱い方法が異なる

キーボードからの入力は GLUT のように文字として得ることができるほか、Shift キーなどの文字のキー以外のものを含む特定のキーの状態を調べることもできます。また、マウスホイールやジョイスティックのデータを取得することもできます。

● GLFW にない機能

一方 GLUT にあって GLFW にない機能もいくつかあります。たとえば Cube や Sphere、Teapot のような図形を表示する機能は省かれています。またビットマップフォントをレンダリングする機能もありません。ポップアップメニューを表示する機能も用意されていません。

最初からダブルバッファリングになっていることや、キーボードやマウス、ジョイスティックの扱い方を見ると、GLFW は GLUT に比べてかなりゲーム向きに作られているように思われます。また GLUT にあって GLFW に無い機能の多くは、現在の OpenGL では非推奨となった機能を使っています。

第2章 準備

2.1 準備するもの

2.1.1 実行環境

本書では実際にプログラムを作成しながら学習するので、パソコンが必要です。また、OpenGL のバージョン 3.2 以降の API を用いるので、NVIDIA GeForce 8 シリーズ以降、ATI RADEON HD シリーズ以降のビデオカード、あるいはグラフィックス機能を内蔵している Intel の第 7 世代の CPU (Ivy Bridge, Core i7-3770 など) や AMD の APU (A シリーズ) が必要です。

2.1.2 ソフトウェア開発環境

想定する OS は Windows 7 以降、Mac OS X 10.7 (Lion) 以降、X Window System (X11) を備えた Linux です。これらの上で、コンパイラ・リンカ、テキストエディタ等のソフトウェア開発環境を使用してプログラムを作成します。

Windows では Visual Studio Community 2015 を使用し、32bit (x86) 版のプログラムを作成することを想定しています。個人ユーザであれば、Visual Studio Community 2015 は無料で使用することができます (https://www.visualstudio.com/ja-jp/products/visual-studio-community-vs.aspx)。ただし C++ を使用するには、セットアップ時に「カスタム」を選んで、「プログラミング言語」にある「Visual C++」と「Windows 開発と Web 開発」にある「ユニバーサル Windows アプリ開発ツール」を選択する必要があります (セットアップ後は「プログラムと機能」から変更できます)。

Mac OS X では Xcode と Command Line Tools を使用します。Xcode は AppStore から無料で入手できます。Command Line Tools は、現在は Xcode に同梱されています。“Xcode” メニューから “Open Developer Tool” の “More Developer Tools” を選ぶと表示される Web サイトからも入手できます (無料の開発者ユーザ登録が必要です)。

Linux では標準的な C++ コンパイラとテキストエディタ、および make コマンドの使用を想定しています。また、ソフトウェア開発環境として Geany (https://www.geany.org/) を使用する例についても解説します。

このほか、OpenGL を使用したプログラムの作成を補助するツールキット GLFW バージョン 3 および OpenGL の拡張機能を利用可能にするための補助ライブラリ GLEW を使用します。

2.1.3 OpenGL

OpenGL はどのプラットホームにも標準で組み込まれているので、改めて用意する必要はありません。ただし、前述の通り OpenGL のバージョン 3.2 以降に対応したグラフィックスハードウェアと、そのドライバソフトウェアがインストールされている必要があります。

また Linux (X Window System) では、別途 Mesa (http://www.mesa3d.org) 関連のパッケージや、使用しているグラフィックスハードウェアのベンダーが提供するドライバソフトウェアのインストールが必要になる場合があります。Intel の CPU 内蔵グラフィックスハードウェアのドライバは https://01.org/linuxgraphics から入手することができます。

2.1.4 GLFW

GLFW のプロジェクトのダウンロードページ (http://www.glfw.org/download.html) からソースプログラムおよびコンパイルされたバイナリファイルがダウンロードできます。最新版のバージョンは 3 ですが、これは以前のバージョン 2 とは互換性がないので注意してください。これをコンパイルして、使用するコンピュータにインストールします。Windows に対しては、既にコンパイルされたバイナリファイルも用意されています。

2.1.5 GLEW

本書のプログラムでは、OpenGL と GLFW のほかに、GLEW (The OpenGL Extension Wrangler Library, http://glew.sourceforge.net) というライブラリも使用します。これはグラフィックスハードウェアの拡張機能を使用可能にするためのものです。特に Windows では、Windows がもともとサポートしている OpenGL のバージョンが 1.1 のために、グラフィックスハードウェアがそれ以降のバージョンに対応したものであっても、そのままでは新しい機能を使用することができません。そこで GLEW を使って、グラフィックスハードウェアが持つ全ての機能をアプリケーションプログラムから使えるようにします。

2.2 GLFW のインストール

2.2.1 Windows

GLFW は、Visual Studio Community 2015 では NuGet を使ってプロジェクトに組み込むことができるので、通常は以下の手順は必要ありません。NuGet を使う方法は後で説明します。

GLFW を Visual Studio 自体に組み込めば、個々のプロジェクトに GLFW を組み込まなくても、プログラムを作成することができます。GLFW のプロジェクトで配布しているバイナリファイル (コンパイル済みのファイル) をダウンロードしてください。

GLFW のダウンロードページ (http://www.glfw.org/download.html) の “32-bit Windows binaries”

のボタンをクリックすれば、glfw-3.X.Y.bin.WIN32.zip (X, Y は数字) というファイルがダウンロードされます。これを右クリックで選択して「すべて展開」を選び、デスクトップ等の適当な場所に展開してください。この中のフォルダやファイルを以下のフォルダ (プロジェクトのプロパティの Target Platform Version が 8.1 の場合) に移動あるいはコピーしてください。

● 32bit 版 Windows の場合

include フォルダにある GLFW フォルダ

```
C:¥Program Files¥Windows Kits¥8.1¥Include¥um
```

lib-vc2015 フォルダにある glfw3.lib ファイル

```
C:¥Program Files¥Windows Kits¥8.1¥Lib¥winv6.3¥um¥x86
```

● 64bit 版 Windows の場合

include フォルダにある GLFW フォルダ

```
C:¥Program Files (x86)¥Windows Kits¥8.1¥Include¥um
```

lib-vc2015 フォルダにある glfw3.lib ファイル

```
C:¥Program Files (x86)¥Windows Kits¥8.1¥Lib¥winv6.3¥um¥x86
```

2.2.2 Mac OS X

GLFW のバージョン 3 は HomeBrew (http://brew.sh) または MacPorts (http://www.macports.org) からパッケージがインストールすることができます。Fink (http://www.finkproject.org) には、本書の執筆時点では見当たりませんでした。HomeBrew では以下の手順でインストールできます。

```
$ brew tap homebrew/versions
$ brew install glfw3
```

ソースファイルからも、以下の手順で簡単にインストールできます。

● インストール

GLFW のプロジェクトのダウンロードページ (http://www.glfw.org/download.html) からソースファイルの ZIP ファイル glfw-3.X.Y.zip (X, Y は数字) をダウンロードしてデスクトップに置き、それをダブルクリックして展開してください。glfw-3.X.Y というディレクトリが作成されます。

次にターミナルを開き、以下のコマンドを順に実行してください (管理者権限が必要です)。'$' はシェルのプロンプトを表します。なお、ファイルは /usr/local 以下にインストールされます。

```
$ cd ~/Desktop/glfw-3.X.Y
$ mkdir build
$ cd build
$ cmake ..
$ make
$ sudo make install
```

● アンインストール

前述の手順でインストールした GLFW は、その際に build ディレクトリの中に作成される Makefile を使ってアンインストールできます (管理者権限が必要です)。

```
$ cd ~/Desktop/glfw-3.X.Y/build
$ sudo make uninstall
```

2.2.3 Linux

GLFW のバージョン 3 は、Ubuntu (http://www.ubuntu.com)、Fedora (http://fedoraproject.org) にはパッケージが用意されています。また、OpenSUSE (http://www.opensuse.org) では Tumbleweed に official release として、Leap 42.1 では unstable package としてインストールできます。ソースファイルからも、以下の手順で簡単にインストールできます。

● インストール

GLFW のプロジェクトのダウンロードページ (http://www.glfw.org/download.html) からソースファイルの ZIP ファイル glfw-3.X.Y.zip (X, Y は数字) ダウンロードしてください。

次にターミナルを開き、以下のコマンドを順に実行してください (管理者権限が必要です)。'$' はシェルのプロンプトを表します。なお、ファイルは /usr/local 以下にインストールされます。

```
$ unzip glfw-3.X.Y.zip
$ cd glfw-3.X.Y
$ mkdir build
$ cd build
$ cmake ..
$ make
$ sudo make install
```

● アンインストール

前述の手順でインストールした GLFW は、その際に build ディレクトリの中に作成される Makefile を使ってアンインストールできます (管理者権限が必要です)。

```
$ cd glfw-3.X.Y/build
$ sudo make uninstall
```

2.3 GLEW のインストール

2.3.1 Windows

ソフトウェア開発環境として Visual Studio Community 2015 を使用する場合は、GLEW も NuGet によりプロジェクトに組み込むことができますので、以下の手順は必要ありません。

パソコンの管理者権限を持っていれば，GLEW を Visual Studio 自体に組み込むことにより、個々のプロジェクトに GLEW を組み込む手間を省くことができます。GLEW のプロジェクトで配布しているバイナリファイル (コンパイル済みのファイル) をダウンロードしてください。

GLEW のプロジェクトのページ (http://glew.sourceforge.net) の “Windows 32-bit and 64-bit” の文字をクリックすれば、glew-1.X.Y-win32.zip (X, Y は数字) というファイルがダウンロードされます。これを右クリックで選択して「すべて展開」を選び、デスクトップ等の適当な場所に展開してください。この中のファイルを以下のフォルダ (プロジェクトのプロパティの Target Platform Version が 8.1 の場合) に移動あるいはコピーしてください。

● 32bit 版 Windows の場合

include¥GL フォルダにある glew.h glxew.h wglew.h ファイル

```
C:¥Program Files¥Windows Kits¥8.1¥Include¥um¥gl
```

lib¥Release¥Win32 フォルダにある glew32.lib glew32s.lib ファイル

```
C:¥Program Files¥Windows Kits¥8.1¥Lib¥winv6.3¥um¥x86
```

bin¥Release¥Win32 フォルダにある glew32.dll ファイル

```
C:¥Windows¥System32
```

● 64bit 版 Windows の場合

include フォルダにある glew.h glxew.h wglew.h ファイル

```
C:¥Program Files (x86)¥Windows Kits¥8.1¥Include¥um¥gl
```

lib¥Release¥Win32 フォルダにある glew32.lib glew32s.lib ファイル

```
C:¥Program Files (x86)¥Windows Kits¥8.1¥Lib¥winv6.3¥um¥x86
```

bin¥Release¥Win32 フォルダにある glew32.dll ファイル

```
C:¥Windows¥SysWOW64
```

2.3.2 Mac OS X

本書の執筆時点では、GLEW のパッケージは MacPorts (http://www.macports.org)、HomeBrew (http://brew.sh)、Fink (http://www.finkproject.org) のいずれのプロジェクトにも用意されていました。ソースファイルからは以下の手順でインストールできます。

● インストール

GLEW のプロジェクトのページ (http://glew.sourceforge.net) からソースファイルの ZIP ファイル glew-1.X.Y.zip (X, Y は数字) をダウンロードしてデスクトップに置き、それをダブルクリックして展開してください。glew-1.X.Y というディレクトリが作成されます。

次にターミナルを開き、以下のコマンドを順に実行してください (管理者権限が必要です)。‘$’ はシェルのプロンプトを表します。なお、ファイルはデフォルトでは /usr にインストールされま

す。インストール先を変更するには、make コマンドの引数で GLEW_DEST にインストール先を設定してください (GLFW と同じディレクトリにインストールすることを勧めます)。

```
$ cd ~/Desktop/glew-1.X.Y
$ make GLEW_DEST=/usr/local
$ sudo make GLEW_DEST=/usr/local install
```

● アンインストール

前述の手順でインストールした GLEW は、その際に build ディレクトリの中に作成される Makefile を使ってアンインストールできます (管理者権限が必要です)。GLEW_DEST にインストールの時に指定したインストール先を指定してください。

```
$ cd ~/Desktop/glew-1.X.Y
$ sudo make GLEW_DEST=/usr/local uninstall
```

2.3.3 Linux

本書の執筆時点では、Fedora (http://fedoraproject.org)、Ubuntu (http://www.ubuntu.com)、OpenSUSE (http://www.opensuse.org) のいずれのディストリビューションにも、GLEW のパッケージが用意されていました。ソースファイルからは、以下の手順でインストールできます。

● インストール

GLEW のプロジェクトのページ (http://glew.sourceforge.net) からソースファイルの TGZ ファイル glew-1.X.Y.tgz (X, Y は数字) をダウンロードし、適当なディレクトリで展開してください。

次にターミナルを開き、以下のコマンドを順に実行してください (管理者権限が必要です)。'$' はシェルのプロンプトを表します。なお、ファイルはデフォルトでは /usr にインストールされます。インストール先を変更するなら、make コマンドの引数で GLEW_DEST にインストール先を設定してください (GLFW と同じディレクトリにインストールすることを勧めます)。

```
$ tar xzf glew-1.X.Y.tgz
$ cd glew-1.X.Y
$ make GLEW_DEST=/usr/local
$ sudo make GLEW_DEST=/usr/local install
```

● アンインストール

前述の手順でインストールした GLEW は、その際に build ディレクトリの中に作成される Makefile を使ってアンインストールできます (管理者権限が必要です)。GLEW_DEST にインストールの時に指定したインストール先を指定してください。

```
$ cd glew-1.X.Y
$ sudo make GLEW_DEST=/usr/local uninstall
```

第3章　プログラムの作成

3.1　ソフトウェア開発環境

プログラムの作成はテキストエディタとコンパイラ・リンカなどの言語処理系さえあれば可能ですが、現在はテキストエディタやコンパイラ・リンカ、デバッガなどをひとまとめにした、統合開発環境 (Integrated Development Environment, IDE) が一般的に用いられます。ここでは各プラットホームにおいてよく使われるソフトウェア開発環境について説明します。

3.1.1　Windows

● プロジェクトの新規作成

ここでは Visual Studio Community 2015 を例にして説明します。まず、NuGet パッケージマネージャーを最新の状態にします。Visual Studio Community 2015 を起動し、「ツール」メニューの「拡張機能と更新プログラム」を選んでください。左の「更新プログラム」から「Visual Studio ギャラリー」を選んで、「NuGet パッケージマネージャー」の更新プログラムがあれば、それを選んで「更新」してください。その後 Visual Studio Community 2015 を再起動し、「新しいプロジェクトを作成します。新しいプロジェクトを作成するには、「ファイル」のメニューの「新規作成」から「プロジェクト」を選ぶか、図 1 の矢印のところをクリックしてください。

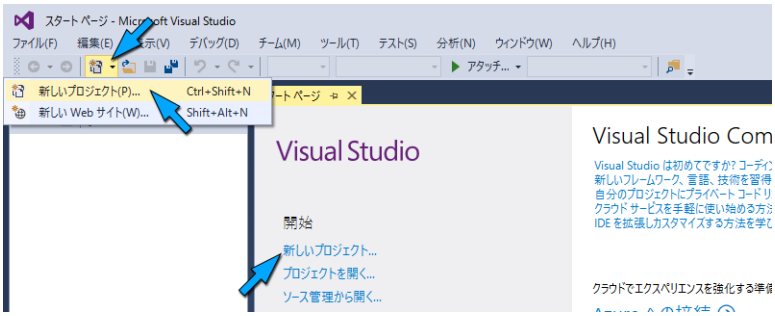

図 1 プロジェクトの作成

インストール済みのテンプレートから「Visual C++」の「General」を選び、「Empty Project」を選んで作成するプログラムの「名前」を設定した後、「OK」をクリックしてください (図 2)。なお、ここでは一つのソリューションに一つのプロジェクトしか作らないので、「ソリューションの

ディレクトリを作成」のチェックは外しても構いません。

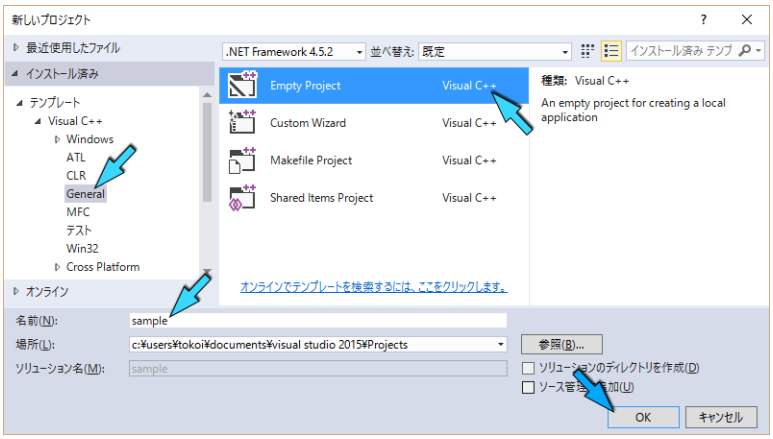

図 2 「Empty Project」の選択

これで空のプロジェクトが作成されます (図 3)。

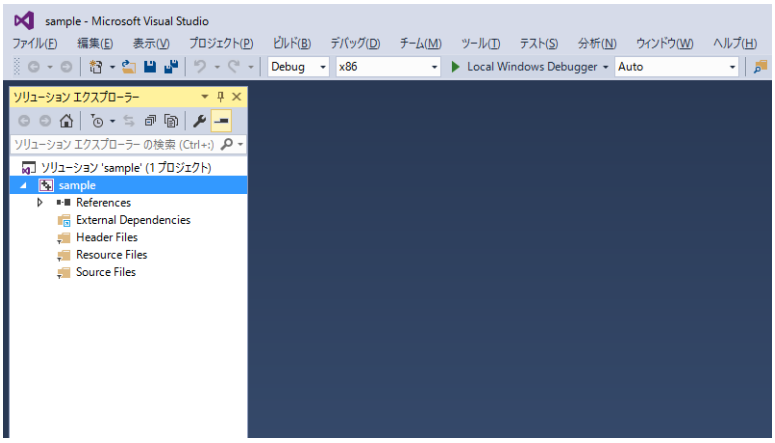

図 3 空のプロジェクト

● NuGet パッケージの管理

このプロジェクトに NuGet を使って GLFW と GLEW を組み込みます。「プロジェクト」のメニューから「NuGet パッケージの管理」を選んでください (図 4)。

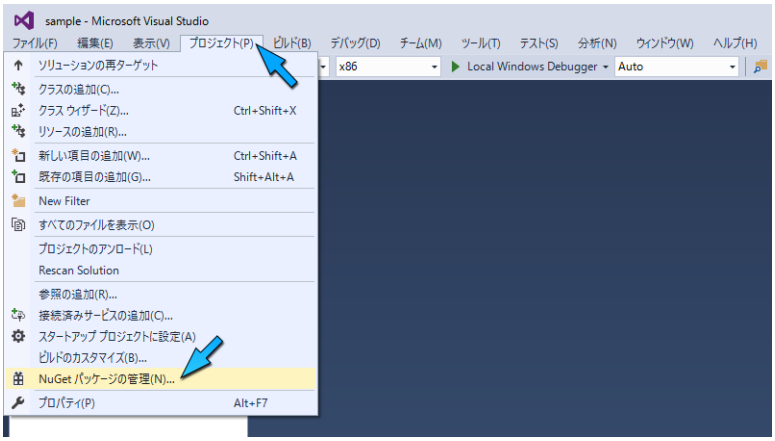

図 4 「NuGet パッケージの管理」の選択

NuGet のウィンドウの上部にある「参照」をクリックし、右側の「パッケージソース」の欄で nuget.org を選択した後、「参照」の下の欄に “glfw” を入力して改行してください (図 5)。GLFW に関連したパッケージがいくつか見つかると思います。これらは (Microsoft ではない) 複数のサードパーティによって提供されており、パッケージによってメンテナンス状況が異なる場合があります。バージョンや説明を確認して、最適なものを選んでインストールしてください。ここでは 3.1.2 というバージョンを使用しています。

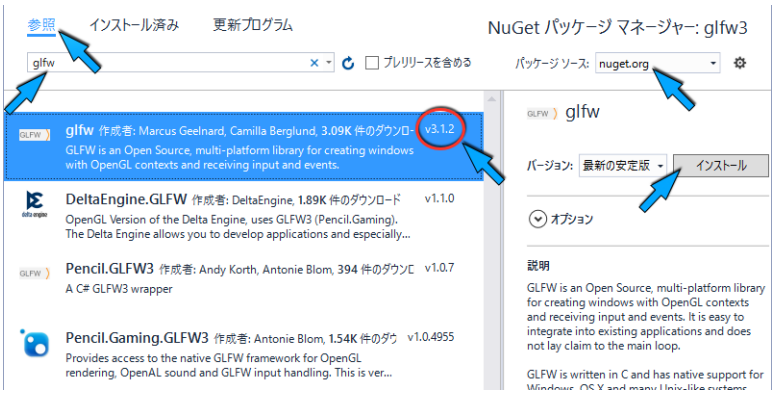

図 5 NuGet で GLFW のパッケージをインストール

同様にして、GLEW もインストールしてください (図 6)。ここでは 1.12.0 というバージョンを使用しています。

図 6 NuGet で GLEW のパッケージをインストール

● **プロジェクトのプロパティ**

作成するプログラムに GLFW と GLEW のライブラリファイルをリンクする設定を行います。「プロジェクト」のメニューの「プロパティ」を選択してください (図 7)。

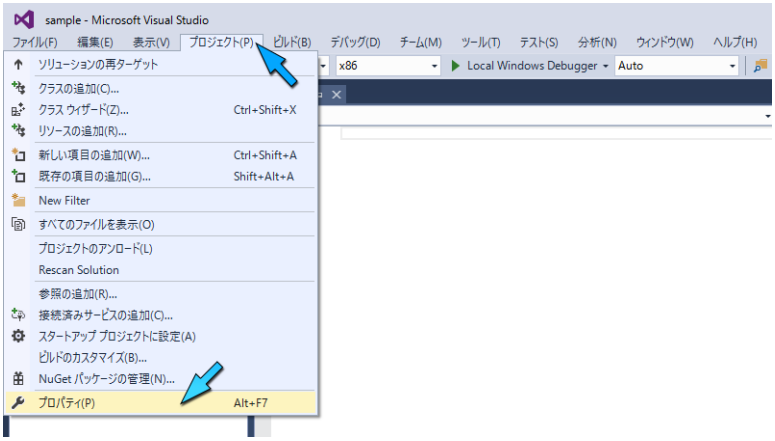

図 7 プロジェクトの「プロパティ」の選択

構成プロパティの「Linker」の項目にある「Input」を選択して、「Additional Dependencies」の欄の右端の▽をクリックして現れる「<編集...>」をクリックしてください (図 8)。

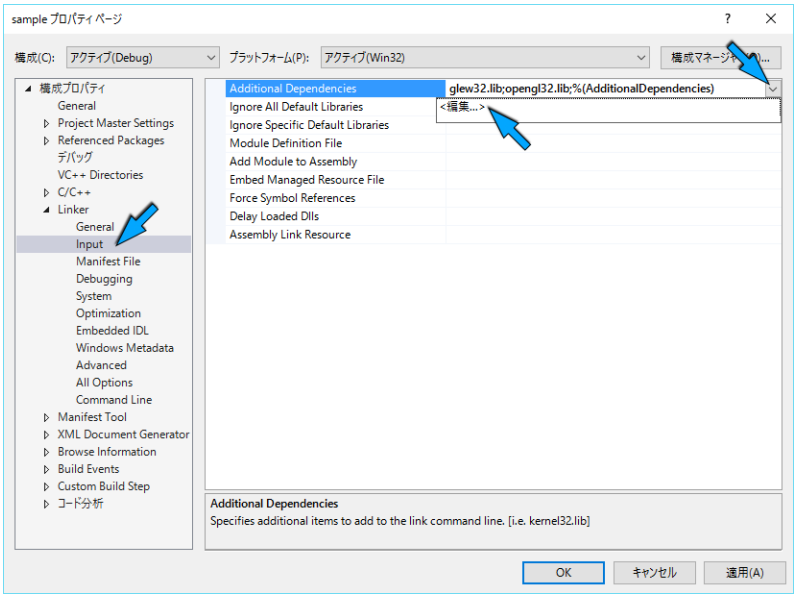

図 8 プロジェクトの「プロパティ」の設定

図 8 の上部にある「構成」では、デバッグのときに用いる「Debug」と、デバッグが完了して最終的なプログラムを作成するときに用いる「Release」を切り替えることができます。「Release」構成でプログラムをビルド (コンパイルやリンク等の一連の処理を経て目的のプログラムを作成する作業) すると最適化により効率の良いプログラムが生成されますが、不要なコードが削除されるなどしてソースプログラムとコンパイルした結果が一致しなくなり、デバッグが難しくなります。ここで「全ての構成」を選べば、両方の構成に同じ設定を適用することができます。

「Additional Dependencies」の一番上の欄に opengl32.lib を入力します (図 9)。自分で GLFW や GLEW をインストールした場合や選んだ NuGet パッケージによっては、ほかに glfw3.lib あるいは glew32.lib、またはこの両方が必要になる場合があります。これはパッケージのドキュメントやビルド時のエラー等で判断してください。最後に「OK」をクリックしてください。

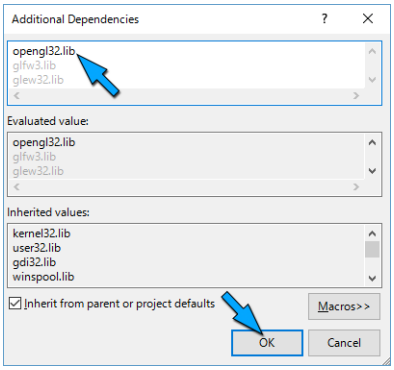

図 9 「Additional Dependencies」の設定

補足: ソースプログラムへの設定の埋め込み

前述の Additional Dependencies の設定は、ソースプログラムに次の 1 行を追加すれば省くこ

とができます。ソースプログラムが複数ある場合は、その中のどれか一つに追加してください。

```
#pragma comment(lib, "opengl32.lib")
```

この場合も自分で GLFW や GLEW をインストールしたときや選んだ NuGet パッケージによって glfw3.lib や glew32.lib が必要になる場合があるので、次の 2 行のいずれか、あるいは両方を必要に応じて追加してください。

```
#pragma comment(lib, "glfw3.lib")
#pragma comment(lib, "glew32.lib")
```

これにより、GLFW と GLEW のファイルを「システムのディレクトリ」に置き、ソースプログラムの冒頭でこれらの設定を行えば、プロジェクトを新規作成するたびに NuGet を使ってこれらを組み込んだり「プロジェクトのプロパティ」を設定したりする手間を省くことができます。

また、「Win32 コンソールアプリケーション」のプロジェクトで作成したプログラムは、実行すると OpenGL のウィンドウのほかに「コンソールウィンドウ」が開きます。コンソールウィンドウはデバッグ時にメッセージ等を表示するのに有用ですが、これを開きたくない場合は次の内容をソースプログラムに追加してください (続けて 1 行で入力してください)。

```
#pragma comment(linker, "/subsystem:¥"windows¥" /entry:¥"mainCRTStartup¥"")
```

補足: ライブラリファイルのリンクについて

ライブラリのリンク方法には、スタティックリンクとダイナミックリンクという二つの方法があります。スタティックリンクは、ライブラリファイルに登録されている関数をプログラムのビルド時にプログラム自体にリンクする (組み込む) 方法です。一方ダイナミックリンクは、ライブラリファイルを別に用意しておき、プログラムの実行時にそのライブラリファイルに登録されている関数を呼び出す方法です。この別に用意したライブラリファイルのことを、**ダイナミックリンクライブラリ** (Dynamic Link Library, DLL) といいます。

ライブラリファイルに登録されている関数は複数のプログラムから利用されるため、スタティックリンクを行うと同じ関数のコピーが異なるプログラム内に存在することになり、メモリやディスクなどが無駄に使われます。これに対してダイナミックリンクでは、関数の実体は一つになるため、スタティックリンクのような無駄がありません。また、ライブラリが更新されたときは DLL を入れ替えるだけで済み、スタティックリンクのようにリンクし直す必要がありません。

このようにメリットの多いダイナミックリンクですが、ダイナミックリンクでは作成したプログラム本体のほかに、リンクしている DLL を「コマンドの検索パス」に含まれるディレクトリ、あるいはプログラムを実行する際の「作業ディレクトリ」に置く必要があります。また、ビルド時と実行時の DLL のバージョンの不一致によるトラブルを起こすこともあります。

前述の設定では、GLFW についてはスタティックリンクライブラリをリンクしていますが、

GLEW はダイナミックリンクライブラリをリンクしています。GLEW についてもスタティックリンクライブラリをリンクするには、glew.h を #include する前に GLEW_STATIC を #define し、glew32.lib の代わりに glew32s.lib をリンクしてください。

```
#define GLEW_STATIC
#include <GL/glew.h>
```

ただし、Visual C++ においてこれらのスタティックリンク用のライブラリをリンクすると、Visual C++ によって暗黙的にリンクされる、ほかのライブラリと競合しているという警告が表示されることがあります。この警告はダイナミックリンクを行えば抑制することができます。

GLFW をダイナミックリンクするには glfw3.lib の代わりに glfw3dll.lib をリンクします (表 1)。そして glfw3.dll と glew32.dll を「コマンドの検索パス」に含まれるディレクトリか、作成したプログラムを実行する際の「作業ディレクトリ」に置いてください。前者の場合は glfw3.dll と glew32.dll を「システムのディレクトリ」である C:¥Windows¥System32 (32bit 版 Windows) あるいは C:¥Windows¥SysWOW64 (64bit 版 Windows) に置くか、「システムの詳細設定[4]」で「システム環境変数」の Path に glfw3.dll と glew32.dll を置いたディレクトリを追加してください。

表 1 リンクするライブラリ

	記号定数	リンクするライブラリ	DLL
スタティックリンク	GLEW_STATIC	glfw3.lib glew32s.lib	(なし)
ダイナミックリンク	(なし)	glfw3dll.lib glew32.lib	glfw3.dll glew32.dll

● ソースファイルの追加

プログラムのソースファイルを作成します。「プロジェクト」のメニューから「新しい項目の追加」を選んでください (図 10)。

[4] Windows 7:「スタートメニュー」から「コントロールパネル」「システムとセキュリティ」「システム」の順に選んで「システムの詳細設定」を選択します。Windows 8: デスクトップから「チャーム」を開き「設定」「PC 情報」の順に選んで「システムの詳細設定」を選択します。Windows 10: スタートメニューから「設定」の「システム」を開き一番下にある「システム情報」を選んで「システムの詳細設定」を選択します。

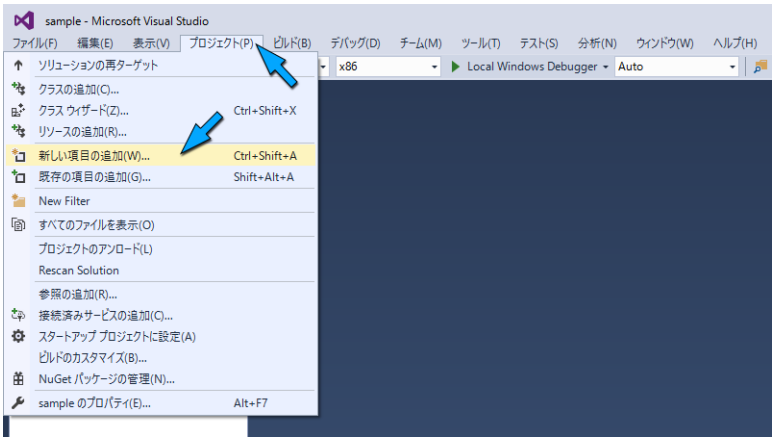

図 10 新しい項目の追加

「新しい項目の追加」のウィンドウ (図 11) の左側のペインで「Visual C++」を選び、中央のペインで「C++ ファイル (.cpp)」を選んでください。また、その下の「名前」の欄にソースファイルのファイル名を入力してください。その後、「追加」をクリックしてください。

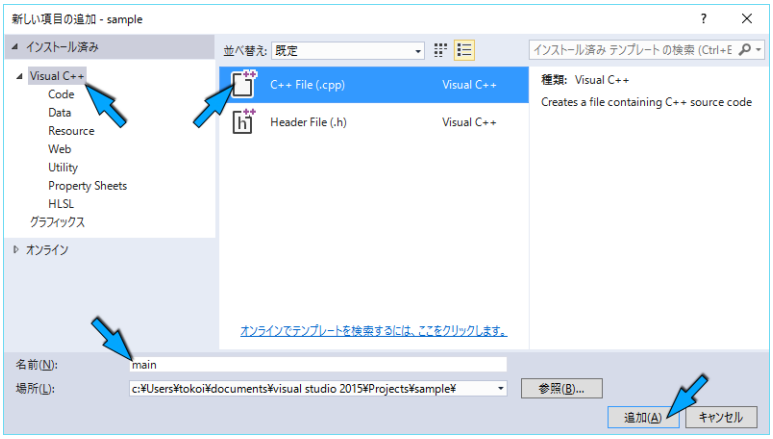

図 11 C++ のソースファイルの追加 (新規作成)

テキストエディタのウィンドウ (図 12) が開きます。ここにソースプログラムを入力します。

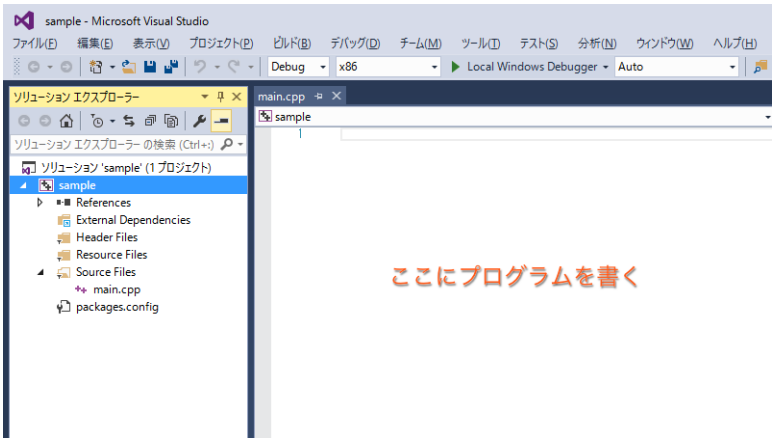

図 12 ソースプログラムの編集

3.1.2 Mac OS X

● プロジェクトの新規作成

Mac OS X では Xcode のバージョン 8 の例について説明します。Xcode を起動し、スプラッシュウィンドウ (図 13) の “Create a new Xcode project” をクリックするか、“File” メニューの “New” から “Project” を選んでください。

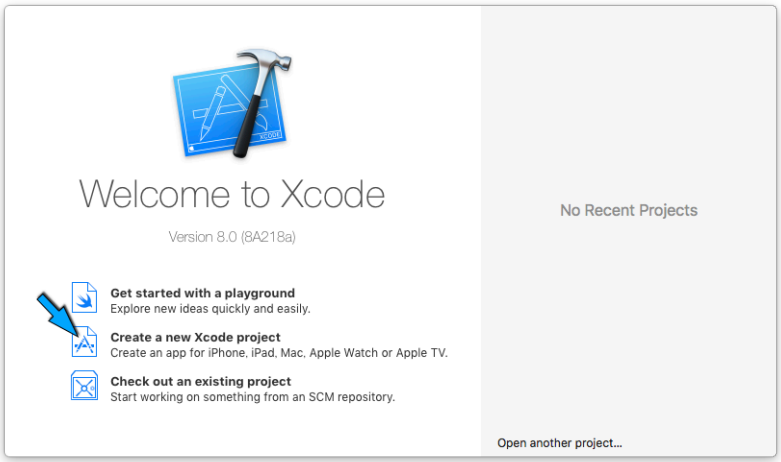

図 13 スプラッシュウィンドウ

プロジェクトのテンプレートとして “macOS” の “Application” から “Command Line Tool” を選び、“Next” をクリックしてください (図 14)。

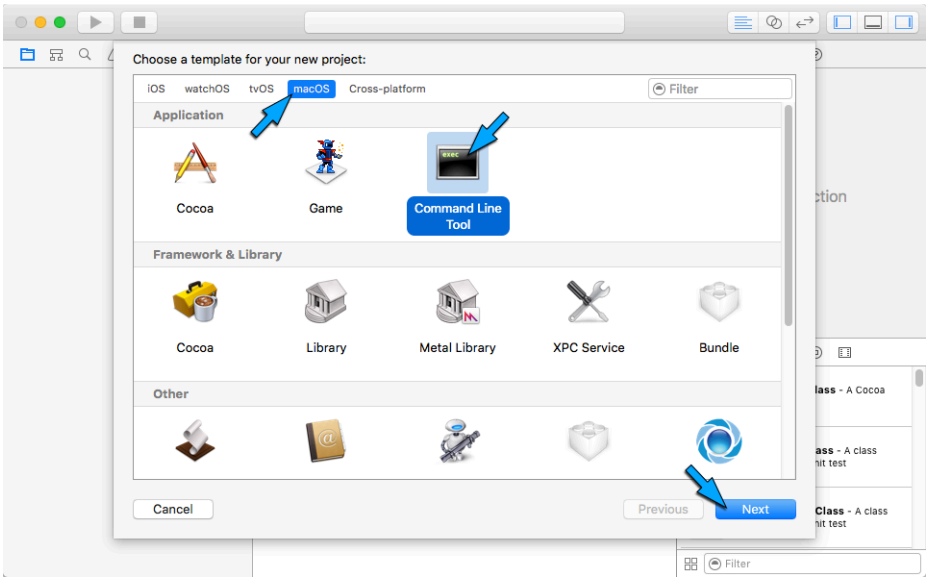

図 14 プロジェクトのテンプレートの選択

プロジェクトのオプションの “Product Name” を設定してください。“Organization Name” や “Company Identifier” は既定値が設定されています。これらは必要に応じて変更してください。

“Type” にはもちろん C++ を選んでください。その後 “Next” をクリックしてください (図 15)。

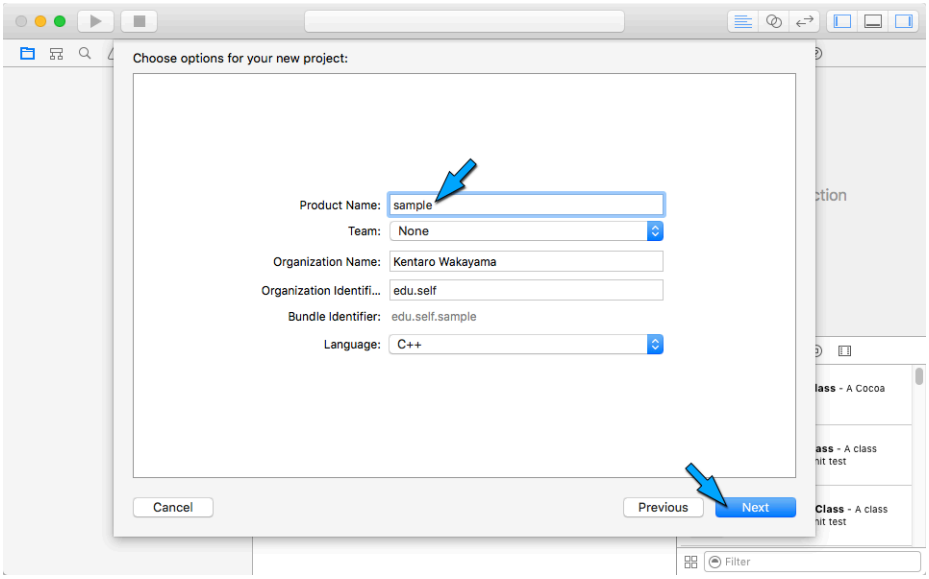

図 15 プロジェクトのオプションの設定

プロジェクトのディレクトリを作成する場所を指定します。適当なところを選んで、“Create” をクリックしてください (図 16)。

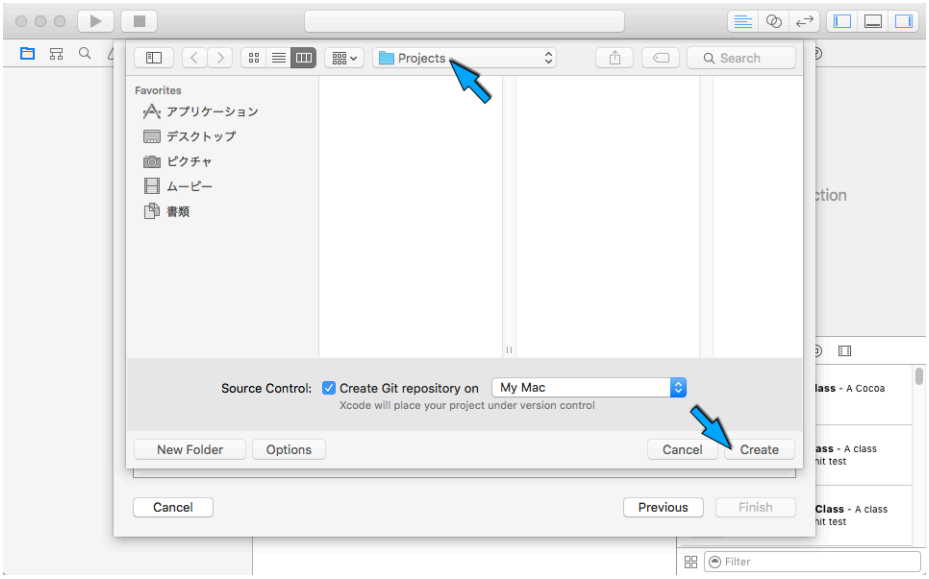

図 16 プロジェクトのディレクトリの保存先

● プロジェクトの設定

まず、ヘッダファイルとライブラリの検索パスを設定します。図 17 のウィンドウの左側にあるプロジェクト名をクリックし、その右のポップアップメニューから “Targets” を選んでください。次に中央部の “Build Settings” を選択します。ここには設定項目が大量にあるので、検索窓に “search” などを入力して “Header Search Paths” という項目を探し、その右側をダブルクリックし

ます。すると入力ウィンドウがポップアップしますから、左下の “+” をクリックして欄を追加し、そこに GLFW と GLEW のヘッダファイルをインストールした場所 (/usr/local/include) を設定してください。

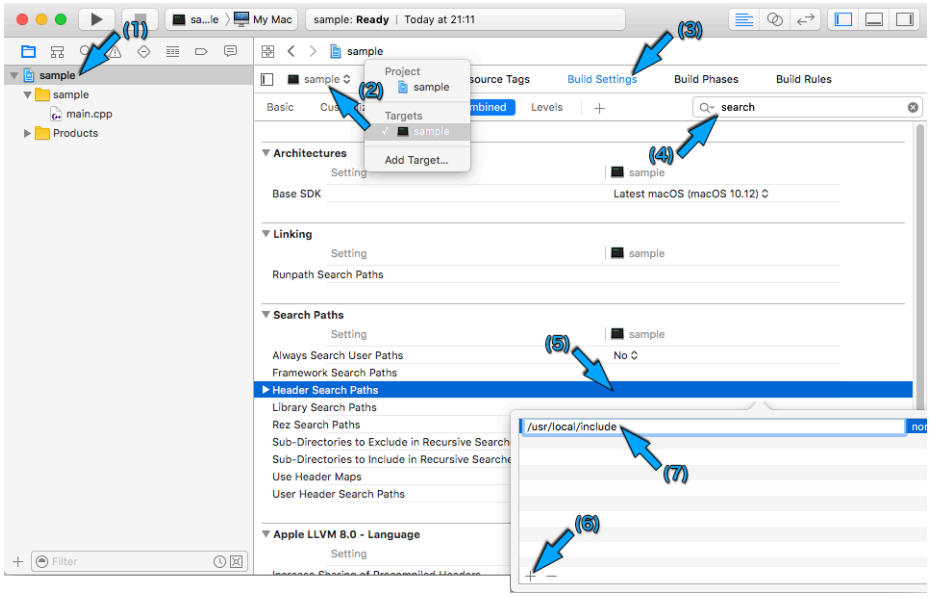

図 17 ヘッダファイルの検索パスの設定

ポップアップしたウィンドウは ESC キーをタイプするか、そのウィンドウ以外のところをクリックすれば消えます。次に “Library Search Paths” の右側 (図 18) をダブルクリックし、同様に GLFW と GLEW のライブラリファイルをインストールした場所 (/usr/local/lib) を設定します。

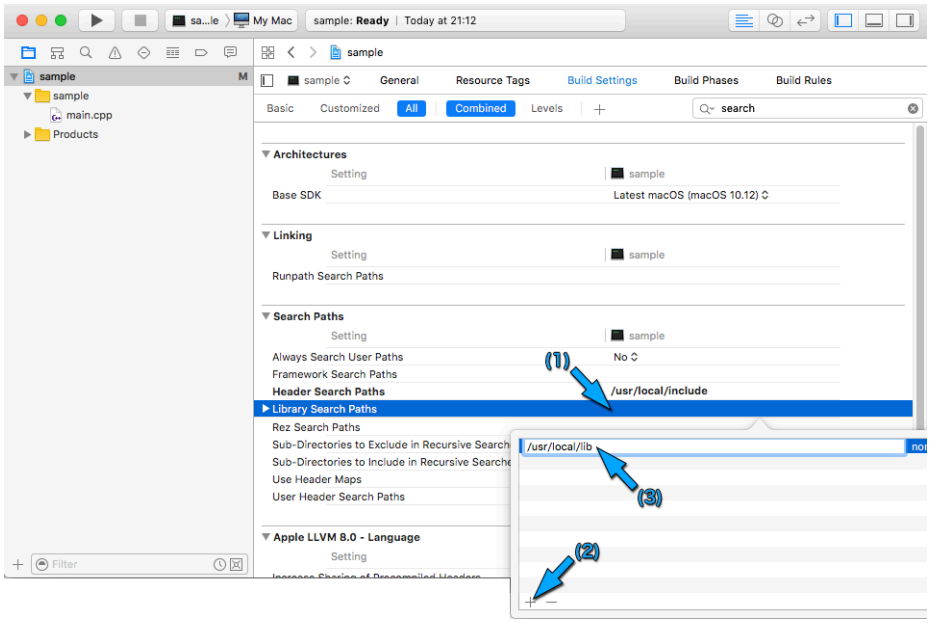

図 18 ライブラリファイルの検索パスの設定

リンクするライブラリとフレームワークを設定します。検索窓に “linker” などを入力して “Other Linker Flags” という項目を探し、その右側をダブルクリックします (図 19)。入力ウィン

ドウがポップアップしたら、左下の “+” をクリックして欄を追加し、そこに以下の内容を入力してください。これは 1 行で入力しても、一つずつ欄を作っても構いません。

```
-lglfw3 -lGLEW  -framework OpenGL -framework CoreVideo -framework IOKit -framework Cocoa
```

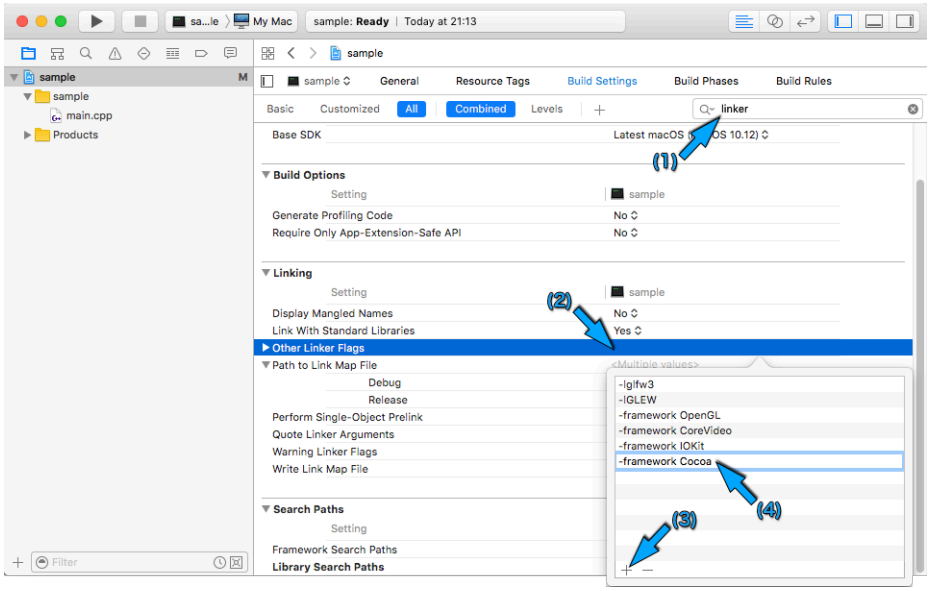

図 19 リンクするライブラリとフレームワークの指定

このテンプレートでは “main.cpp” というファイル名のソースファイルが自動的に作成されます (図 20)。これに main() 関数が定義されています。

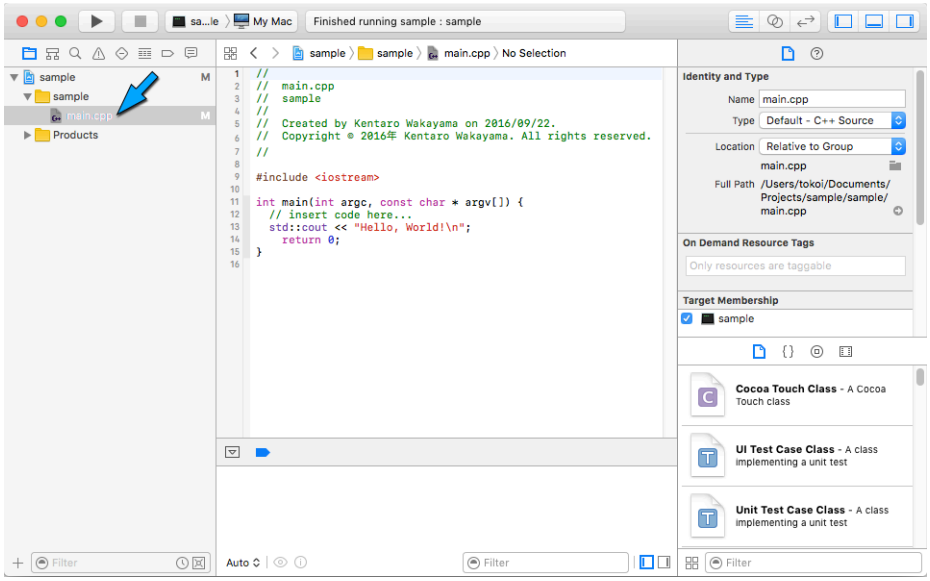

図 20 ソースプログラムの編集

● Makefile を作る

Xcode を使用せず、シェルでコマンドを使ってプログラムを作成する場合は、Makefile を用意

しておくと手間が省けます。まず、mkdir コマンドなどを使って、ソースファイルを置く空のディレクトリを一つ作成してください。‘$’ はシェルのプロンプトを表します。

```
$ mkdir sample
```

次に、テキストエディタを使って以下の内容のファイルを Makefile というファイル名で作成し、このディレクトリに保存してください。また、このファイルの行頭の空白 (網かけの部分) には、スペースではなくタブを使ってください。

```
CXXFLAGS   = -g -Wall -I/usr/local/include
LDLIBS     = -L/usr/local/lib -lglfw3 -lGLEW  -framework OpenGL ¥
    -framework CoreVideo -framework IOKit -framework Cocoa
OBJECTS    = $(patsubst %.cpp,%.o,$(wildcard *.cpp))
TARGET     = sample

.PHONY: clean

$(TARGET): $(OBJECTS)
    $(LINK.cc) $^ $(LOADLIBES) $(LDLIBS) -o $@

clean:
    -$(RM) $(TARGET) $(OBJECTS) *~ .*~ core
```

このファイルを作成しておけば、このディレクトリで make コマンドを実行することにより、ソースプログラムがコンパイル・リンクされて、実行プログラムが作成されます。

3.1.3 Linux

● Geany を使う

Linux にもさまざまな統合開発環境があり、また、本書の執筆時点ではどれが主流というわけでもなさそうです。テキストエディタとコマンドラインコンパイラだけで開発することも可能なのですが、ここではシンプルで軽量な開発環境の Geany (http://www.geany.org) の例を紹介します。

ターミナルを開き、次のコマンドで Geany を起動して、テキストファイルを新規作成します。この例では main.cpp というファイル名のソースファイルを作成します。‘$’ はシェルのプロンプトを表します。

```
$ geany main.cpp
```

あるいは、Geany の「ファイル」メニューから新規作成することもできます。C++ のソースファイルであれば、「テンプレートから新規作成」の main.cxx を選びます (図 21)。拡張子が “.cxx” になってしまいますが、Linux ではこれも C++ のソースファイルの拡張子として認識されます。

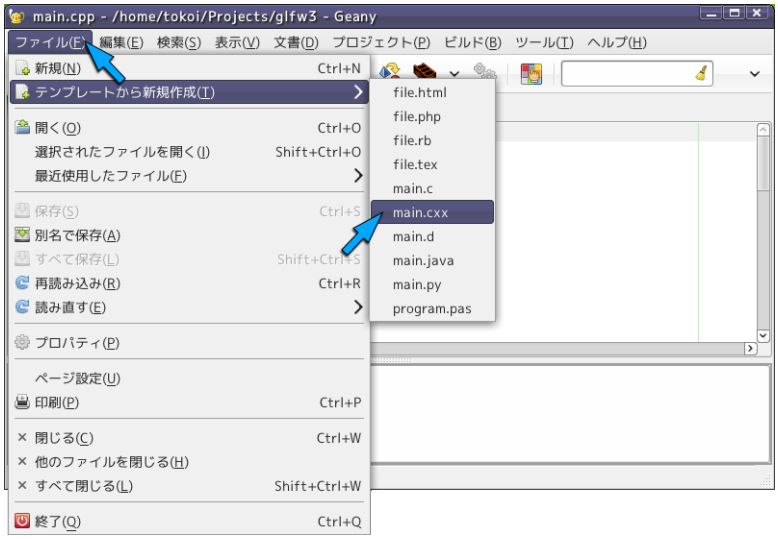

図 21 ソースファイルの新規作成

Geany の「ビルド」メニューから「ビルドコマンドを設定」を選んでください (図 22)。

図 22 ビルドコマンドの設定ウィンドウの呼び出し

「ビルド」ラベルのコマンドの欄 (図 23) に、既に入力されているものの後にスペースをあけて次の内容を追加してください。これは途中で改行せずに、一行で入力してください。-lm は GLFW、GLEW、あるいは OpenGL では利用されませんが、本書のプログラムでは数学ライブラリを使うので、ここで追加しておきます。

```
-lglfw3 -lGLEW -lGL -lXrandr -lXinerama -lXcursor -lXi -lXxf86vm -lX11 -lpthread
-lrt -lm
```

設定が終わったら「OK」をクリックしてください。

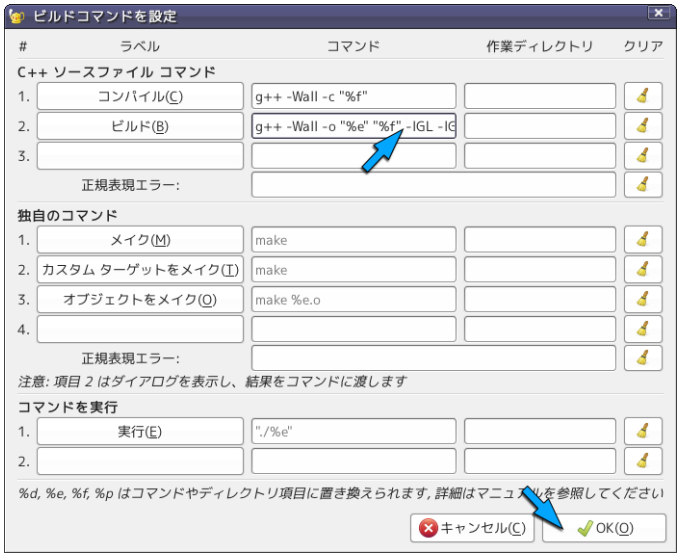

図 23 ビルドコマンドを設定

● Makefile を作る

Geany などの統合開発環境を使用せず、シェルでコマンドを使ってプログラムを作成する場合は、Makefile を用意しておくと手間が省けます。まず、mkdir コマンドなどを使って、ソースファイルを置く空のディレクトリを作成してください。'$' はシェルのプロンプトを表します。

```
$ mkdir sample
```

次に、テキストエディタを使って以下の内容のファイルを Makefile というファイル名で作成し、このディレクトリに保存してください。また、このファイルの行頭の空白 (網かけの部分) には、スペースではなくタブを使ってください。

```
CXXFLAGS   = -g -Wall
LDLIBS     = -lglfw3 -lGLEW -lGL -lXrandr -lXinerama -lXcursor -lXi ¥
    -lXxf86vm -lX11 -lpthread -lrt -lm
OBJECTS    = $(patsubst %.cpp,%.o,$(wildcard *.cpp))
TARGET     = sample

.PHONY: clean

$(TARGET): $(OBJECTS)
	$(LINK.cc) $^ $(LOADLIBES) $(LDLIBS) -o $@

clean:
	-$(RM) $(TARGET) $(OBJECTS) *~ .*~ core
```

このファイルを作成しておけば、このディレクトリで make コマンドを実行することにより、ソースプログラムがコンパイル・リンクされて、実行プログラムが作成されます。

3.2 ソースプログラムの作成

3.2.1 処理手順

GLFW を使ったプログラムの処理手順を以下に示します。

(1) GLFW を初期化する (glfwInit())
(2) ウィンドウを作成する (glfwCreateWindow())
(3) ウィンドウが開いている間くり返し描画する (glfwWindowShouldClose())
(4) ダブルバッファリングのバッファの入れ替えを行う (glfwSwapBuffers())
(5) ウィンドウが閉じたら終了処理を行う (glfwTerminate())

● メインプログラム (main.cpp)

最初に「最小の」C++ のプログラムを考えてみます。ソースファイル名を main.cpp として、以下の網かけの部分を (ソフトウェア開発環境の) テキストエディタに打ち込んでください。C++ では、main() 関数の return 文を省略すると、0 を return することになります。

```
int main()
{
}
```

このプログラムは、プログラムのエントリポイント (プログラムの実行を開始する場所) である main() 関数しかなく、その中身も空なので、実行しても何も起こらずに終了します。

■ (サンプルプログラム step00)

3.2.2 GLFW を初期化する

main() 関数に GLFW の初期化処理を追加します。

● メインプログラム (main.cpp) の変更点

ソースプログラムの冒頭で GLFW のヘッダファイル GLFW/glfw3.h を #include し、main() 関数の最初の部分、すなわちプログラムの実行開始直後に glfwInit() 関数を実行します。これにより、このプログラムで OpenGL を使用するための準備が行われます。glfwInit() 関数の戻り値が GL_FALSE のときは GLFW の初期化に失敗していますから、エラーメッセージを出してプログラムを終了するようにしておきます。

```
#include <iostream>
#include <GLFW/glfw3.h>

int main()
```

```
{
  // GLFW を初期化する
  if (glfwInit() == GL_FALSE)
  {
    // 初期化に失敗した
    std::cerr << "Can't initialize GLFW" << std::endl;
    return 1;
  }
}
```

int glfwInit(void)

GLFW を初期化します。ほかの全ての GLFW の関数を実行する前に実行する必要があります。初期化に成功すれば GL_TRUE (値は 1)、失敗すれば GL_FALSE (値は 0) を返します。

3.2.3 ウィンドウを作成する

GLFW の初期化が成功したら、glfwCreateWindow() 関数を使ってウィンドウを作成します。

● メインプログラム (main.cpp) の変更点

glfwCreateWindow() 関数の戻り値は**レンダリングコンテキスト**と呼ばれるウィンドウ固有の情報を保持するオブジェクトのポインタです。これが NULL ならウィンドウの作成に失敗しているので、エラーメッセージを表示してプログラムを終了するようにしておきます。なお、この時点では既に glfwInit() による GLFW の初期化が完了しているので、プログラムを終了する直前、すなわち return 文の前で、glfwTerminate() 関数を実行して GLFW の終了処理を行います。

```
#include <iostream>
#include <GLFW/glfw3.h>

int main()
{
  // GLFW を初期化する
  if (glfwInit() == GL_FALSE)
  {
    // 初期化に失敗した
    std::cerr << "Can't initialize GLFW" << std::endl;
    return 1;
  }

  // ウィンドウを作成する
  GLFWwindow *const window(glfwCreateWindow(640, 480, "Hello!", NULL, NULL));
  if (window == NULL)
  {
    // ウィンドウが作成できなかった
    std::cerr << "Can't create GLFW window." << std::endl;
    glfwTerminate();
    return 1;
  }
}
```

GLFWwindow *glfwCreateWindow(int width, int height, const char *title, GLFWmonitor *monitor, GLFWwindow *share)

GLFW のウィンドウを作成します。戻り値は作成したウィンドウのハンドルです。ウィンドウが開けなければ NULL を返します。

width

作成するウィンドウの横幅の画素数で、0 より大きくなければなりません。

height

作成するウィンドウの高さの画素数で、0 より大きくなければなりません。

title

作成するウィンドウのタイトルバーに表示する文字列です。文字コードは UTF-8 です。

monitor

ウィンドウをモニタ (ディスプレイ) の全面に表示するとき (フルスクリーンモード)、表示するモニタを指定します。フルスクリーンモードでなければ NULL を指定します。

share

引数 share にほかのウィンドウのハンドルを指定すれば、そのウィンドウとテクスチャなどのリソースを共有します。NULL を指定すれば、リソースの共有は行いません。

void glfwTerminate(void)

GLFW の終了処理を行います。glfwInit() 関数で GLFW の初期化に成功した場合は、プログラムを終了する前に、この関数を実行する必要があります。この関数は GLFW で作成した全てのウィンドウを閉じ、確保した全てのリソースを解放して、プログラムの状態を glfwInit() 関数で初期化する前に戻します。この後に GLFW の機能を使用するには、再度 glfwInit() 関数を実行しなければなりません。

3.2.4 作成したウィンドウを処理対象にする

glfwCreateWindow() 関数を何回も呼び出せば、一つのプログラムで複数のウィンドウを作成することができます。しかし、複数のウィンドウに同時に図形を描画することはできません。図形の描画を行う前に、これから描画を行うウィンドウを指定する必要があります。

● メインプログラム (main.cpp) の変更点

glfwCreateWindow() 関数の戻り値のポインタ (ハンドル) を glfwMakeContextCurrent() 関数の引数に指定して、そのウィンドウのレンダリングコンテキストを処理の対象にします。開いたウィンドウに対する設定や図形の描画などは、この後に行います。

```
#include <iostream>
#include <GLFW/glfw3.h>
```

```
int main()
{
  // GLFW を初期化する
  if (glfwInit() == GL_FALSE)
  {
    // 初期化に失敗した
    std::cerr << "Can't initialize GLFW" << std::endl;
    return 1;
  }

  // ウィンドウを作成する
  GLFWwindow *const window(glfwCreateWindow(640, 480, "Hello!", NULL, NULL));
  if (window == NULL)
  {
    // ウィンドウが作成できなかった
    std::cerr << "Can't create GLFW window." << std::endl;
    glfwTerminate();
    return 1;
  }

  // 作成したウィンドウを OpenGL の処理対象にする
  glfwMakeContextCurrent(window);
}
```

void glfwMakeContextCurrent(GLFWwindow *const window)

引数 window に指定したハンドルのウィンドウのレンダリングコンテキストをカレント (処理対象) にします。レンダリングコンテキストは描画に用いられる情報で、ウィンドウごとに保持されます。図形の描画はこれをカレントに設定したウィンドウに対して行われます。

window

OpenGL の処理対象とするウィンドウのハンドルを指定します。

3.2.5 OpenGL の初期設定

glfwMakeContextCurrent() 関数で OpenGL による描画を行うウィンドウを指定すれば、ようやく OpenGL の機能が使用できるようになります。

● メインプログラム (main.cpp) の変更点

ここでは glClearColor() 関数により表示領域を消去する色 (背景色) を設定します。この最初の三つの引数は、塗りつぶす色を (赤, 緑, 青) の光の三原色の割合で表します。ここでは白 (1, 1, 1) にしています。第 4 引数は不透明度を表します。ここでは透明 (0) にしています。なお、関数名が gl〜 で始まるものは OpenGL の API です (GLFW の関数名は glfw〜 で始まります)。

```
#include <iostream>
#include <GLFW/glfw3.h>

int main()
{
  // GLFW を初期化する
```

```
  if (glfwInit() == GL_FALSE)
  {
    // 初期化に失敗した
    std::cerr << "Can't initialize GLFW" << std::endl;
    return 1;
  }

  // ウィンドウを作成する
  GLFWwindow *const window(glfwCreateWindow(640, 480, "Hello!", NULL, NULL));
  if (window == NULL)
  {
    // ウィンドウが作成できなかった
    std::cerr << "Can't create GLFW window." << std::endl;
    glfwTerminate();
    return 1;
  }

  // 作成したウィンドウを OpenGL の処理対象にする
  glfwMakeContextCurrent(window);

  // 背景色を指定する
  glClearColor(1.0f, 1.0f, 1.0f, 0.0f);
}
```

void glClearColor(GLclampf R, GLclampf G, GLclampf B, GLclampf A)

glClear(GL_COLOR_BUFFER_BIT) でウィンドウを塗り潰す色を指定します。

R，G，B

それぞれ赤色、緑色、青色の成分の強さを示す GLclampf 型 (float 型と等価) の値で、0～1 の値を持ちます (図 24)。1 が最も明るく、この三つにそれぞれ 0, 0, 0 を指定すれば黒色、1, 1, 1 を指定すれば白色になります。

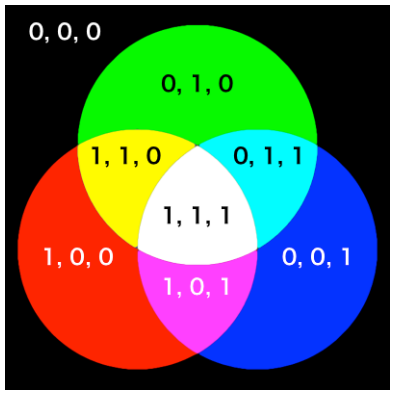

図 24 R, G, B の値と色

A

アルファ値と呼ばれ、OpenGL では不透明度として扱われます (0 で透明、1 で不透明)。

補足: OpenGL のデータ型

グラフィックスハードウェア (GPU) が内部的に使用しているデータの書式がプラットホーム

(CPU) と一致しているとは限らないので、OpenGL で取り扱うデータには独自のデータ型が割り当てられています (表 2)。

しかし、基本的には、そのデータ型は CPU、あるいは使用するプログラミング言語のものに対応付けられています。たとえば GLubyte 型は C 言語あるいは C++ 言語の unsigned char 型に対応し、GLfloat 型は float 型に対応しています。また GLclampf も float 型のデータに対応しますが、これはこのデータが OpenGL で使用されるときに、負の値は 0 に、1 を超える正の値は 1 として扱われることを示します (データ自体がそういう特性を持つわけではありません)。

なお、GLhalf 型に対応する CPU 側のデータ型は存在しないため、CPU 側でこの値をそのまま計算などに用いることはできません。しかし、グラフィックスハードウェアから取り出して CPU 側で保持しておくことは可能です。

表 2 OpenGL のデータ型

OpenGL のデータ型	最小ビット数	説明
GLboolean	1	論理値
GLbyte	8	符号付き二進整数（二の補数表現）
GLubyte	8	符号なし二進整数
GLchar	8	文字列中の文字
GLshort	16	符号付き二進整数（二の補数表現）
GLushort	16	符号なし二進整数
GLint	32	符号付き二進整数（二の補数表現）
GLuint	32	符号なし二進整数
GLint64	64	符号付き二進整数（二の補数表現）
GLuint64	64	符号なし二進整数
GLsizei	32	非負の二進整数で表したサイズ
GLenum	32	二進整数で表した列挙子
GLintptr	※	符号付き二進整数（二の補数表現）
GLsizeiptr	※	非負の二進整数で表したサイズ
GLsync	※	同期オブジェクトのハンドル
GLbitfield	32	ビットフィールド
GLhalf	16	符号なしの値に符号化された半精度浮動小数点数
GLfloat	32	単線度浮動小数点数
GLclampf	32	[0, 1] にクランプされた単精度浮動小数点数
GLdouble	64	倍線度浮動小数点数
GLclampd	64	[0, 1] にクランプされた倍精度浮動小数点数

※ ポインタ（アドレス）を保持するのに必要なビット数

3.2.6 メインループ

ウィンドウを開いても、このままではすぐに main() 関数の最後に到達して、プログラムが終了してしまいます。そこで、ウィンドウが閉じられなければプログラムが終了しないように、while

によって処理を繰り返します。

● メインプログラム (main.cpp) の変更点

この while ループはウィンドウが開いている間、継続します。ウィンドウが閉じられたかどうかは glfwWindowShouldClose() 関数で調べることができます。

このループの中では、最初に glClear() 関数を使って画面の表示領域を消去します。その後、そこに OpenGL により図形の描画を行います。描画が終わったら glfwSwapBuffers() 関数を実行して、図形を描画したカラーバッファと現在図形を表示しているカラーバッファを入れ替えます。この処理は**ダブルバッファリング**といいます。

最後に、このプログラムが次に何をすべきか判断するために、この時点で発生しているイベントを調査します。glfwWindowShouldClose() によるウィンドウを閉じるべきかどうかの判断も、このイベントの調査にもとづいて行われます。イベントの調査には、マウスなどで操作する対話的なアプリケーションソフトウェアの場合は、イベントが発生するまで待つ glfwWaitEvents() 関数を用います。これに対して、時間とともに画面の表示を更新するアニメーションなどの場合は、イベントの発生を待たない glfwPollEvents() 関数を用います (表 3)。

```
#include <iostream>
#include <GLFW/glfw3.h>

int main()
{
  // GLFW を初期化する
  if (glfwInit() == GL_FALSE)
  {
    // 初期化に失敗した
    std::cerr << "Can't initialize GLFW" << std::endl;
    return 1;
  }

  // ウィンドウを作成する
  GLFWwindow *const window(glfwCreateWindow(640, 480, "Hello!", NULL, NULL));
  if (window == NULL)
  {
    // ウィンドウが作成できなかった
    std::cerr << "Can't create GLFW window." << std::endl;
    glfwTerminate();
    return 1;
  }

  // 作成したウィンドウを OpenGL の処理対象にする
  glfwMakeContextCurrent(window);

  // 背景色を指定する
  glClearColor(1.0f, 1.0f, 1.0f, 0.0f);

  // ウィンドウが開いている間繰り返す
  while (glfwWindowShouldClose(window) == GL_FALSE)
  {
```

```
    // ウィンドウを消去する
    glClear(GL_COLOR_BUFFER_BIT);

    //
    // ここで描画処理を行う
    //

    // カラーバッファを入れ替える
    glfwSwapBuffers(window);

    // イベントを取り出す
    glfwWaitEvents();
  }
}
```

int glfwWindowShouldClose(GLFWwindow *const window)

window に指定したウィンドウを閉じる必要があるとき、戻り値は非 0 になります。

void glClear(GLbitfield mask)

ウィンドウを塗り潰します。mask には塗り潰すバッファを指定します。フレームバッファは色を格納するカラーバッファのほか、隠面消去処理に使うデプスバッファ、図形の型抜きを行うステンシルバッファなどの複数のバッファで構成されており (図 25)、これらが一つのウィンドウに重なっています。mask に GL_COLOR_BUFFER_BIT を指定したときは、カラーバッファだけを glClearColor() 関数で指定した色で塗り潰します。

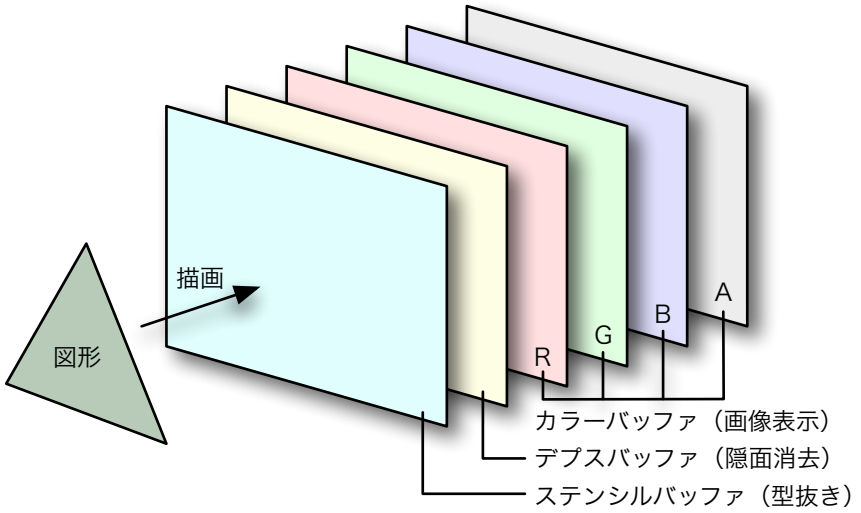

図 25 フレームバッファの構成

void glfwSwapBuffers(GLFWwindow *const window)

window に指定したウィンドウのカラーバッファを入れ替えます。図形を描画した後にこの関数を実行しなければ、描画したものは画面に表示されません。

void glfwWaitEvents(void)

マウスの操作などのイベントの発生を待ちます。イベントが発生したら、それを記録してプログラムの実行を再開します。この関数はメインのループ以外で実行すべきではありません。

void glfwPollEvents(void)

マウスの操作などのイベントを取り出し、それを記録します。この関数はプログラムを停止させないので、アニメーションのように連続して画面表示を更新する場合に使用します。

表 3 イベントの取り出し

glewWaitEvents()	glewPollEvents()
イベントが発生するまで待つ (プログラムの実行を停止する)。イベントが発生すれば、最初のイベントを取り出してプログラムの実行を再開する。	イベントが発生していれば、それを取り出す。イベントの有無にかかわらず、次に進む (プログラムの実行を停止しない)。

補足: バッファについて

バッファは一般に緩衝装置と訳されます。これは何か二つのものが接続されており、一方からもう一方に何らかの影響を与えるような状況にあるとき、この二つの間に入って影響の仲立ちをするもののことをいいます。

OpenGL におけるバッファは、データを次の処理に引き渡すために用いるメモリのことを指します。OpenGL ではいろいろな種類のバッファを使用しますが、ここで説明しているバッファは、描画した図形を画面に表示するために用いるフレームバッファのカラーバッファです。

図形はフレームバッファのカラーバッファに描画されます。このカラーバッファの内容が読み出されて画面に表示されます。ここでフレームバッファへの描画と読み出しを同時に行うと、表示にちらつきが発生してしまいします。ダブルバッファリングはカラーバッファを二つ用意しておいて、一方を表示している間にもう一方に描画する手法です。描画が完了した時点でこの二つのカラーバッファを入れ替える (図 26) ことで、ちらつきの発生を抑えます。

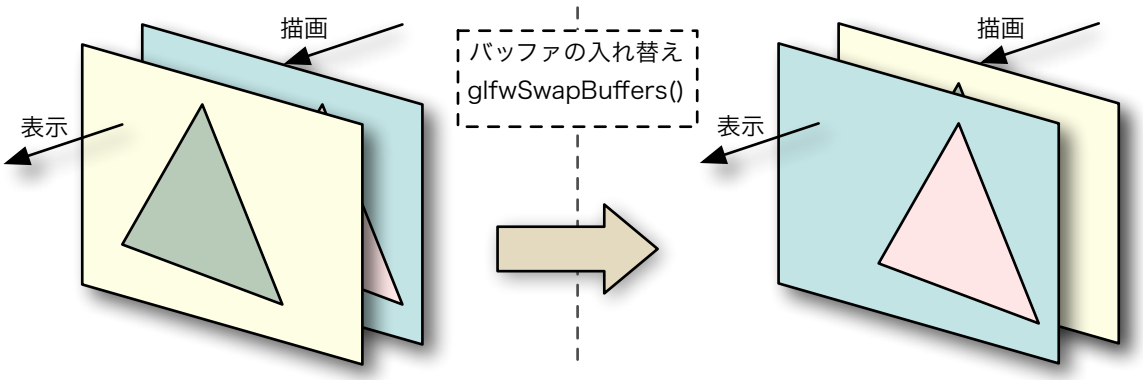

図 26 ダブルバッファリング

補足: イベントについて

画面上で一つのウィンドウの上に別のウィンドウが重なっているとき、上のウィンドウが閉じられたときには、隠されていた下のウィンドウの表示内容を描き直す必要があります。また対話的なアプリケーションソフトウェアでは、マウスなどの操作に対応した処理を随時実行する必要があります。このように、ある処理の実行のきっかけとなる出来事を、**イベント**といいます。

発生したイベントに対応する処理の実行方法には、画面表示のたびに glfwWaitEvents() 関数や glfwPollEvents() 関数を用いてイベントを取り出す方法 (ポーリング方式) と、特定のイベントが発生したときに実行する関数をあらかじめ登録しておく方法 (コールバック方式) があります。

3.2.7 終了処理

glfwInit() 関数による初期化に成功していれば、プログラムを終了する前に glfwTerminate() 関数を実行する必要があります。

● メインプログラム (main.cpp) の変更点

main() 関数の最後の return 文の直前に glfwTerminate() 関数の呼び出しを追加します。

```
#include <iostream>
#include <GLFW/glfw3.h>

int main()
{
  // GLFW を初期化する
  if (glfwInit() == GL_FALSE)
  {
    // 初期化に失敗した
    std::cerr << "Can't initialize GLFW" << std::endl;
    return 1;
  }

  // ウィンドウを作成する
  GLFWwindow *const window(glfwCreateWindow(640, 480, "Hello!", NULL, NULL));
  if (window == NULL)
  {
    // ウィンドウが作成できなかった
    std::cerr << "Can't create GLFW window." << std::endl;
    glfwTerminate();
    return 1;
  }

  // 作成したウィンドウを OpenGL の処理対象にする
  glfwMakeContextCurrent(window);

  // 背景色を指定する
  glClearColor(1.0f, 1.0f, 1.0f, 0.0f);

  // ウィンドウが開いている間繰り返す
  while (glfwWindowShouldClose(window) == GL_FALSE)
```

```
  {
    // ウィンドウを消去する
    glClear(GL_COLOR_BUFFER_BIT);

    //
    // ここで描画処理を行う
    //

    // カラーバッファを入れ替える
    glfwSwapBuffers(window);

    // イベントを取り出す
    glfwWaitEvents();
  }

  glfwTerminate();
}
```

3.2.8 atexit() による glTerminate() の実行

プログラムの終了時には必ず glTerminate() 関数を実行する必要があります。そのため、このプログラムでは main() 関数の最後の return 文の前のほか、glfwCreateWindow() 関数のエラー処理のところでも実行しています。

しかし、この後プログラムを追加していくと、ほかの場所でもプログラムを終了させなければならない場合が発生するかも知れません。そのためにプログラムのあちこちで glTerminate() 関数を実行することは無駄に思われますし、呼び出しを忘れる可能性もあります。そこで、プログラムの終了時に必ず glTerminate() 関数が実行されるように、atexit() 関数を用います。

● メインプログラム (main.cpp) の変更点

プログラムの冒頭で atexit() 関数を定義しているヘッダファイル cstdlib を #include し、glfwInit() 関数が成功した後で、atexit() 関数により glfwTerminate() 関数を登録します。また、既に main() 関数に埋め込んでいた glfwTerminate() 関数は削除します。

```
#include <cstdlib>
#include <iostream>
#include <GLFW/glfw3.h>

int main()
{
  // GLFW を初期化する
  if (glfwInit() == GL_FALSE)
  {
    // 初期化に失敗した
    std::cerr << "Can't initialize GLFW" << std::endl;
    return 1;
  }

  // プログラム終了時の処理を登録する
  atexit(glfwTerminate);
```

```
  // ウィンドウを作成する
  GLFWwindow *const window(glfwCreateWindow(640, 480, "Hello!", NULL, NULL));
  if (window == NULL)
  {
    // ウィンドウが作成できなかった
    std::cerr << "Can't create GLFW window." << std::endl;
    return 1;
  }

  // 作成したウィンドウを OpenGL の処理対象にする
  glfwMakeContextCurrent(window);

  // 背景色を指定する
  glClearColor(1.0f, 1.0f, 1.0f, 0.0f);

  // ウィンドウが開いている間繰り返す
  while (glfwWindowShouldClose(window) == GL_FALSE)
  {
    // ウィンドウを消去する
    glClear(GL_COLOR_BUFFER_BIT);

    //
    // ここで描画処理を行う
    //

    // カラーバッファを入れ替える
    glfwSwapBuffers(window);

    // イベントを取り出す
    glfwWaitEvents();
  }
}
```

int atexit(void (*function)(void))

atexit() 関数は、引数 function に指定した関数を、プログラム終了時に実行するよう登録します。atexit() 関数を複数回呼び出して、複数の関数を登録することができます (少なくとも 32 個の関数が登録できます)。その場合、関数は登録した順の逆順に実行されます。戻り値として、関数の登録に成功した時は 0、失敗した時は 0 以外の値を返します。

3.2.9 GLEW の初期化

前節までで、とりあえず OpenGL による描画を行うための環境が整いました。このままでも OpenGL による図形の描画は可能です。しかし、本書で用いる OpenGL の機能のいくつかは、これだけでは使用できない場合があります。

これは、Windows において標準的に用意されている OpenGL のバージョンが 1.1 であり、本書が対象としている OpenGL のバージョン 3.2 に対してかなり古いものであるためです。また Mac OS X では、そのままでは OpenGL 2.1 に対応した Compatibiliy Profile が使用され、OpenGL 3.2 以降の機能を使用するには明示的に Core Profile に切り替える必要があります (Mac OS X

10.7 以降)。さらに、それに合わせて使用するするヘッダファイルも切り替える必要があります。

そこで、サポートされていない OpenGL の機能を有効にし、プラットホームによるソースプログラムの違いを吸収するために、ここで GLEW を導入します。

● メインプログラム (main.cpp) の変更点

GLEW のヘッダファイル GL/glew.h を GLFW のヘッダファイル GLFW/glfw3.h の**前**で #include し、GLFW のウィンドウを作成して、そこへの描画を指定した**後**に、GLEW の初期化を行う glewInit() 関数を追加します。なお、本書の執筆時点では、glewInit() 関数を実行する前に “glewExperimental = GL_TRUE;” という代入を行わないと、GLFW と GLEW を組み合わせたときに一部の API が有効になりませんでした。

```
#include <cstdlib>
#include <iostream>
#include <GL/glew.h>
#include <GLFW/glfw3.h>

int main()
{
  // GLFW を初期化する
  if (glfwInit() == GL_FALSE)
  {
    // 初期化に失敗した
    std::cerr << "Can't initialize GLFW" << std::endl;
    return 1;
  }

  // プログラム終了時の処理を登録する
  atexit(glfwTerminate);

  // ウィンドウを作成する
  GLFWwindow *const window(glfwCreateWindow(640, 480, "Hello!", NULL, NULL));
  if (window == NULL)
  {
    // ウィンドウが作成できなかった
    std::cerr << "Can't create GLFW window." << std::endl;
    return 1;
  }

  // 作成したウィンドウを OpenGL の処理対象にする
  glfwMakeContextCurrent(window);

  // GLEW を初期化する
  glewExperimental = GL_TRUE;
  if (glewInit() != GLEW_OK)
  {
    // GLEW の初期化に失敗した
    std::cerr << "Can't initialize GLEW" << std::endl;
    return 1;
  }
```

```
  // 背景色を指定する
  glClearColor(1.0f, 1.0f, 1.0f, 0.0f);
```

GLenum glewInit(void)

ハードウェアやドライバには用意されているにも関わらず、プラットホームではサポートされていない OpenGL の機能を有効にし、プログラムから呼び出せるようにします。戻り値は、処理に成功した時は GLEW_OK (値は 0) を返します。失敗した時は 0 以外の値を返します。

3.2.10 OpenGL のバージョンとプロファイルの指定

本書では OpenGL 3.2 以降の機能を使用してプログラムを作成します。そのため、OpenGL のバージョンやプロファイルを指定してウィンドウを作成します。これは glfwCreateWindow() 関数でウィンドウを作成する前に、glfwWindowHint() 関数を用いて行います。

● メインプログラム (main.cpp) の変更点

```
int main()
{
  // GLFW を初期化する
  if (glfwInit() == GL_FALSE)
  {
    // 初期化に失敗した
    std::cerr << "Can't initialize GLFW" << std::endl;
    return 1;
  }

  // プログラム終了時の処理を登録する
  atexit(glfwTerminate);

  // OpenGL Version 3.2 Core Profile を選択する
  glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
  glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 2);
  glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT, GL_TRUE);
  glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);

  // ウィンドウを作成する
  GLFWwindow *const window(glfwCreateWindow(640, 480, "Hello!", NULL, NULL));
  if (window == NULL)
  {
    // ウィンドウが作成できなかった
    std::cerr << "Can't create GLFW window." << std::endl;
    return 1;
  }

  // 作成したウィンドウを OpenGL の処理対象にする
  glfwMakeContextCurrent(window);

  // GLEW を初期化する
  glewExperimental = GL_TRUE;
  if (glewInit() != GLEW_OK)
  {
```

```
    // GLEW の初期化に失敗した
    std::cerr << "Can't initialize GLEW" << std::endl;
    return 1;
  }

  // 背景色を指定する
  glClearColor(1.0f, 1.0f, 1.0f, 0.0f);
```

void glfwWindowHint(int target, int hint)

この後に glfwCreateWindow() 関数によって作成するウィンドウの特性を設定します。デフォルトの設定に戻すには glfwDefaultWindowHints() を呼び出します。

target

ヒントを設定する対象。以下のものが指定できます (一部を抜粋)。

GLFW_RED_BITS, GLFW_GREEN_BITS, GLFW_BLUE_BITS, GLFW_ALPHA_BITS

それぞれカラーバッファの赤色・緑色・青色・アルファに割り当てるビット数を hint に指定します。デフォルトはいずれも 8 です。

GLFW_DEPTH_BITS

デプスバッファに割り当てるビット数を hint に指定します。デフォルトは 24 です。

GLFW_STENCIL_BITS

ステンシルバッファに割り当てるビット数を hint に指定します。デフォルトは 8 です。

GLFW_SAMPLES

hint にマルチサンプル時のサンプル数を指定します。0 を指定するとマルチサンプルが無効になります。デフォルトは 0 です。

GLFW_STEREO

hint に GL_TRUE を指定すればステレオモードになります。デフォルトは GL_FALSE です。これを GL_TRUE にできるかどうかは、ハードウェアに依存します。

GLFW_CLIENT_API

hint に GLFW_OPENGL_ES_API を指定すれば OpenGL ES の API を使用します。デフォルトは GLFW_OPENGL_API です。

GLFW_CONTEXT_VERSION_MAJOR

OpenGL のメジャーバージョン番号を hint に指定します。バージョンが 3.2 ならば 3 です。デフォルトは 1 です。

GLFW_CONTEXT_VERSION_MINOR

OpenGL のマイナーバージョン番号を hint に指定します。バージョンが 3.2 ならば 2 です。デフォルトは 1 です。

GLFW_OPENGL_FORWARD_COMPAT

OpenGL の Forward Compatible Profile (前方互換プロファイル、古い機能が使えない) を使う場合は、hint に GL_TRUE を指定します。デフォルトは GL_FALSE です。

GLFW_OPENGL_PROFILE

使用する OpenGL のプロファイルを指定します。Compatible Profile を使う場合は hint に GLFW_OPENGL_COMPAT_PROFILE 、 Core Profile を使う場合は hint に GLFW_OPENGL_CORE_PROFILE を指定します。デフォルトは 0 で、この場合はシステムの設定に依存します。

グラフィックスハードウェアが glfwWindowHint() 関数で指定した特性に対応していなければ、その後の glfwCreateWindow() 関数の実行は失敗してウィンドウは開かれません。

3.2.11 作成したウィンドウに対する設定

一方、作成したウィンドウに対する設定は、glfwCreateWindow() 関数によるウィンドウの作成に成功した後に行います。ここでは glfwSwapInterval() 関数によって、ダブルバッファリングにおけるバッファの入れ替えタイミングを設定します。

● メインプログラム (main.cpp) の変更点

```
// GLEW を初期化する
glewExperimental = GL_TRUE;
if (glewInit() != GLEW_OK)
{
  // GLEW の初期化に失敗した
  std::cerr << "Can't initialize GLEW" << std::endl;
  return 1;
}

// 垂直同期のタイミングを待つ
glfwSwapInterval(1);

// 背景色を指定する
glClearColor(1.0f, 1.0f, 1.0f, 0.0f);
```

void glfwSwapInterval(int interval)

ダブルバッファリングにおける、カラーバッファの入れ替えのタイミングを指定します。

interval

少なくとも interval に指定した回数だけディスプレイの垂直同期タイミング (V-Sync、帰線消去期間) を待ってから、カラーバッファの入れ替えを行います。普通は 1 を指定します。0 を指定するとディスプレイの垂直同期タイミングを待たなくなるため、数値の上では fps (frame per second) が上昇しますが、完全な画面表示が行われるわけではありません。

■ サンプルプログラム step01

3.3 プログラムのビルドと実行

ソースプログラムをコンパイルしてオブジェクトプログラムを生成し、リンクによりオブジェクトプログラム同士やライブラリを結合して実行プログラムを生成する一連の作業をビルドといいます。これは簡単なプログラムであればコンパイラのコマンドを用いて実行することができますが、プログラムが複雑になるとコンパイラのコマンドだけでビルドするのは手間がかかります。そこで、統合開発環境や make コマンドなどの補助的な手段を用いてビルドします。

3.3.1 Windows

Visual Studio Community 2015 でビルドするには、「ビルド」メニューから「ソリューションのビルド」を選んでください (図 27)。その下の「“プロジェクト名” のビルド」でも構いません。

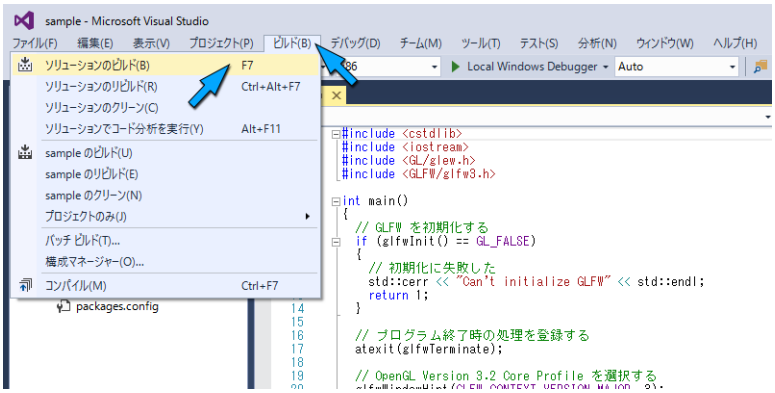

図 27 Visual Studio Community 2015 によるプログラムのビルド

プログラムの実行は、「デバッグメニュー」の「デバッグ開始」を選ぶか、ファンクションキーの F5 をタイプしてください。ツールバーにある「緑色の右向き三角▶」をクリックしてもデバッグ実行を開始します。ソースプログラムの修正後、ビルドせずにプログラムを実行した時は、先にビルドを実行します。

なお、ソリューション構成が「Debug」のとき、ビルド時に「LINK : warning LNK4098: defaultlib 'MSVCRT' はほかのライブラリの使用と競合しています。/NODEFAULTLIB:library を使用してください。」という警告が出ることがあります。これはここでは無視してください。GLFW と GLEW をダイナミックリンクすれば、この警告を抑制できます。

3.3.2 Mac OS X

Xcode では、左上の黒い右向き三角▶をクリックすれば、プログラムのビルドと実行を続けて行います (図 28)。Command キーを押しながら R のキーをタイプしても同様です。

図 28 Xcode によるプログラムのビルドと実行

コマンドラインでビルドする場合は、c++ コマンドに -I/usr/local/include -L/usr/local/lib -lglfw3 -lGLEW -framework OpenGL -framework CoreVideo -framework IOKit -framework Cocoa オプションを追加してください。'$' はシェルのプロンプトを表します。

```
$ c++ main.cpp -g -Wall -I/usr/local/include -L/usr/local/lib -lglfw3 -lGLEW ¥
-framework OpenGL -framework CoreVideo -framework IOKit -framework Cocoa
```

さすがに長過ぎるので、Makefile を作って make コマンドを実行するのが賢明だと思います。Makefile を用意していれば、make コマンドを実行するだけです。

```
$ make
```

3.3.3 Linux

Geany の場合は、「ビルド」メニューの「ビルド」を選んでください (図 29)。ビルドされたプログラムを実行するには、同じ「ビルド」メニューの「実行」を選んでください。これらはそれぞれファンクションキーの F9 と F5 でも実行できます。また、ツールバー上にも対応するボタンがあります。

図 29 Geany によるプログラムのビルドと実行

コマンドラインでビルドする場合は、c++ コマンドに -lglfw3 -lGLEW -lGL -lXrandr -lXinerama -lXcursor -lXi -lXxf86vm -lX11 -lpthread -lrt -lm オプションを追加してください。'$' はシェルのプロンプトを表します。

```
$ c++ main.cpp -g -Wall -lglfw3 -lGLEW -lGL -lXrandr -lXinerama -lXcursor -lXi ¥
-lXxf86vm -lX11 -lpthread -lrt -lm
```

長いので、Makefile を作って make コマンドを使用することを勧めます。Makefile を用意していれば、make コマンドを実行するだけです。

```
$ make
```

● 実行結果

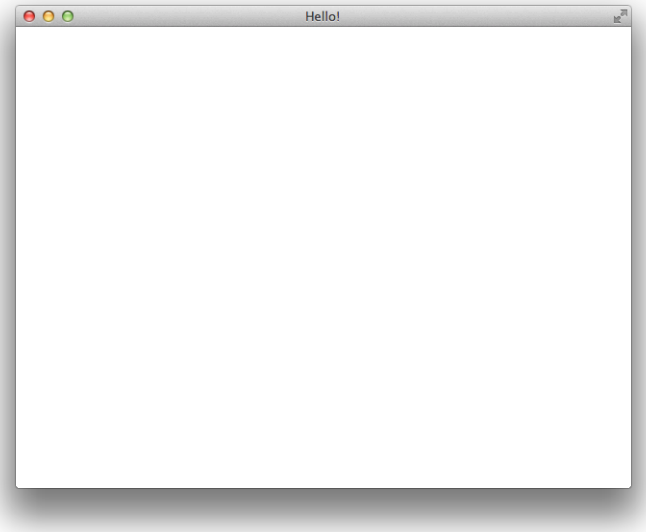

図 30 実行結果 (Mac OS X の場合)

第4章　プログラマブルシェーダ

4.1　画像の生成

OpenGL において描画する図形のデータは、線分の端点や多角形の頂点の、位置や色などの情報です。この情報をもとに図形の画面上での位置や色を決定し、そこに図形を描きます (図 31)。

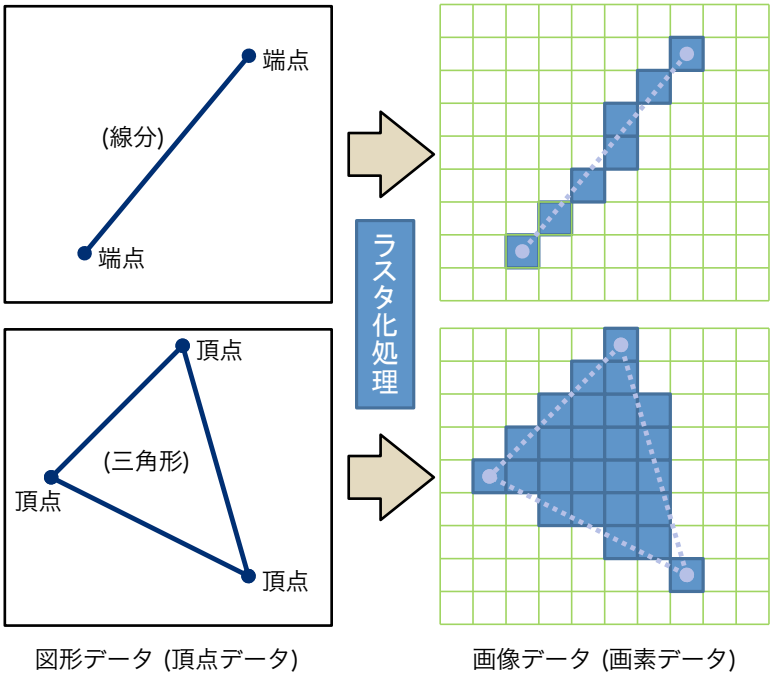

図 31 画像の生成

図形は画素で構成された画像データとして画面上に描かれます。それには、頂点の情報から画素の情報への変換処理が必要になります。この処理を**ラスタ化処理**あるいは**ラスタライズ**といい、そのためのハードウェアを**ラスタライザ**といいます。グラフィックスハードウェアによる図形の描画は、このラスタ化処理をはさんで、以下の三つの手順によって行われます (図 32)。

なお、このように処理を複数のステージ (処理の段階) に分けて、各ステージがデータを順送りする処理の形態のことを、**パイプライン処理**といいます。また、この図形描画のためのパイプライン処理を、**レンダリングパイプライン**といいます。

(1)　頂点の画面上での位置を決定する (頂点処理)

(2)　頂点の位置と図形の種類をもとに画素の情報を生成する (ラスタ化処理)

(3)　画素の色を決定して画像を生成する (画素処理)

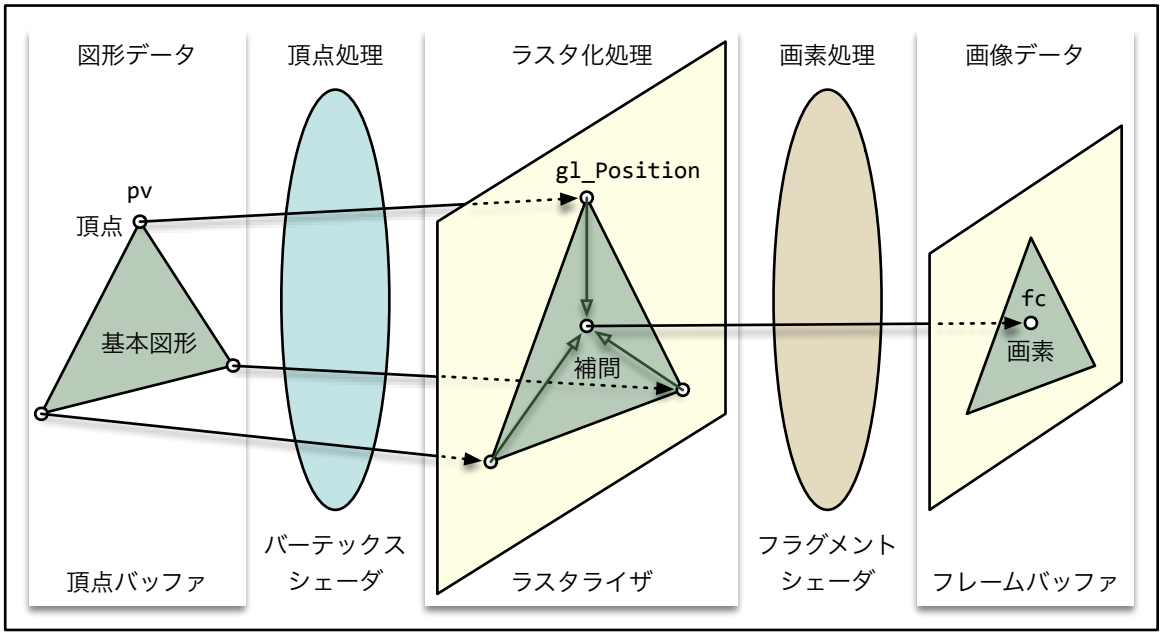

図 32 レンダリングパイプライン

この頂点処理や画素処理は、かつては決まった (変更できない) 処理を実行するハードウェアによって行われていました。これは固定機能シェーダと呼ばれます。そしてグラフィックス表示の品質に対する要求が多様化するにつれ、このハードウェアも多機能化し、実数計算機能を搭載するなど、CPU に匹敵する複雑さを持つようになりました。このことから、このようなグラフィックスハードウェアは GPU (Graphics Processing Unit) と呼ばれるようになりました。

しかし、このグラフィックス表示の品質に対する要求には際限がありませんから、そのひとつひとつに対してハードウェアに機能を追加して対応しようとしても限界があります。そこで、この頂点処理と画素処理のハードウェアをプログラム可能にして、機能をプログラムにより実装できるようにすることが考えられました。これを**プログラマブルシェーダ**といい、このプログラミングに用いるプログラミング言語を**シェーディング言語**といいます。

OpenGL で使用できるシェーディング言語には、OpenGL 自体の機能である GLSL (OpenGL Shading Language) のほか、NVIDIA が開発した Cg (C for Graphics) があります。また、DirectX では HLSL (High Level Shading Language) というシェーディング言語が使用できます。なお、Cg は DirectX でも使用できます。本書ではシェーディング言語として GLSL を使用します。

プログラマブルシェーダは、通常いくつかのシェーダを組み合わせて構成されます。OpenGL の場合、頂点処理を担当するものを**バーテックスシェーダ**といい、画素処理を担当するものを**フラグメントシェーダ**といいます。なお、これは最も単純な構成であり、これらのほかにもいくつかの異なる機能を持つシェーダを組み合わせる場合があります。

4.2 シェーダプログラム

4.2.1 シェーダプログラムの作成手順

GLSL のシェーダプログラムを利用する手順は、次のようになります。

(1) glCreateProgram() 関数によってプログラムオブジェクトを作成します。
(2) glCreateShader() 関数によってバーテックスシェーダとフラグメントシェーダのシェーダオブジェクトを作成します。
(3) glShaderSource() 関数によって、作成したそれぞれのシェーダオブジェクトに対してソースプログラムを読み込みます。
(4) glCompileShader() 関数によって読み込んだソースプログラムをコンパイルします。
(5) glAttachShader() 関数によってプログラムオブジェクトにシェーダオブジェクトを組み込みます。
(6) glLinkProgram() 関数によってプログラムオブジェクトをリンクします。

以上の処理によりシェーダのプログラムオブジェクトが作成されますから、図形を描画する前に glUseProgram() 関数を実行して、図形の描画にこのシェーダプログラムを使うようにします。

4.2.2 シェーダのソースプログラム

● **バーテックスシェーダのソースプログラム** point.vert

バーテックスシェーダのソースプログラムとして、ここでは次のものを用います。このプログラムは、入力された図形データのひとつひとつの頂点に対して実行されます。

```
#version 150 core
in vec4 position;
void main()
{
  gl_Position = position;
}
```

この 1 行目の #version の行は、使用する GLSL のバージョンを指定します。150 は OpenGL 3.2 の GLSL のバージョンである 1.5 を表し、core は Core Profile であることを示します。

2 行目は、レンダリングパイプラインの前のステージからこのシェーダプログラムに送られたデータを受け取るために用いる **in 変数**を、position という変数名で宣言しています。バーテックスシェーダの in 変数には、CPU から送られた図形データの一つの頂点のデータが格納されます。この頂点のデータを**頂点属性** (attribute) といい、この in 変数を特に **attribute 変数**といいます。

vec4 はこの変数のデータ型が 32bit 浮動小数点型 (float 型) の 4 要素のデータからなるベクトル型であることを示します。ベクトル型には、このほかに 2 要素の vec2、3 要素の vec3 が

あります。ベクトルの各要素は、C 言語の構造体と同様にドット演算子 “.” を用いて変数名の後に { .x, .y, .z, .w }、{ .s, .t, .p, q } あるいは { .r, .g, .b, .a } を付けることにより、参照や代入を行うことができます (表 4)。

表 4 GLSL のデータ型

変数のデータ型	内容	ベクトル型の要素
float	単精度浮動小数点 1 個	.x .y .z .w
vec2	単精度浮動小数点 2 個	.r .g .b .a
vec3	単精度浮動小数点 3 個	.s .t .p .q
vec4	単精度浮動小数点 4 個	

凡例

vec4 pv;	vec4 型の変数宣言
pv.xy	pv の第1,2要素, vec2 型
pv.rgb	pv の第1,2,3要素, vec3 型
pv.q	pv の第4要素, float 型
pv.yx	pv の第1,2要素の順序を入れ替えたもの, vec2 型
pv.brg	pv の第1,2,3要素を3,1,2の順に並べ替え, vec3 型

シェーダプログラムは C や C++ 言語同様 main() 関数から実行を開始します。ただし、シェーダプログラムの main() 関数は、引数を持たず、値を戻すこともありません。したがって、関数の定義は void main() になります。

頂点属性は CPU から一旦 GPU が管理する**頂点バッファオブジェクト**と呼ばれるメモリに格納します。その後 CPU から**描画命令 (ドローコール)** を送ると、GPU は頂点バッファオブジェクトから頂点ごとに頂点属性を取り出して attribute 変数に格納し、バーテックスシェーダのプログラムを実行します。

gl_Position は GLSL の組み込み変数で、この変数に代入した値がパイプラインの次のステージ (ラスタライザなど) に送られます。バーテックスシェーダは必ずこの変数に値を代入しなければなりません。このプログラムでは attribute 変数 position をそのまま gl_Position に代入していますから、図形の描画に指定された頂点属性をそのまま次のステージのラスタライザに送ります。

● フラグメントシェーダのソースプログラム point.frag

フラグメントシェーダのソースプログラムには、次のものを用います。このプログラムは出力する画像のひとつひとつの画素に対して実行されます。

```
#version 150 core
out vec4 fragment;
void main()
```

```
{
  fragment = vec4(1.0, 0.0, 0.0, 1.0);
}
```

この 2 行目は、フラグメントの色の出力先の out 変数を fragment という変数名で宣言しています。out 変数に代入したデータは、レンダリングパイプラインの次のステージに送られます。フラグメントシェーダの out 変数に代入した値は、フレームバッファのカラーバッファに格納されます。

vec4() は四つの数値を vec4 型に変換 (キャスト) します。出力変数 fragment には色を表す R (赤)、G (緑)、B (青)、および A (不透明度) の 4 要素のベクトルを代入します。もしフレームバッファに値を格納しない (画素を描かない) なら、フラグメントシェーダで discard 命令を実行します。

4.2.3 プログラムオブジェクトの作成

シェーダのソースプログラムを GPU で実行するには、それらをコンパイル・リンクして、シェーダのプログラムオブジェクトを作成する必要があります。まず、glCreateProgram() 関数で空のプログラムオブジェクト (program) を作成します。

```
  const GLuint program(glCreateProgram());
```

GLuint glCreateProgram(void)

プログラムオブジェクトを作成します。戻り値は作成されたプログラムオブジェクト名 (番号) で、作成できれば 0 でない正の整数、作成できなければ 0 を返します。

4.2.4 シェーダオブジェクトの作成

次に、シェーダのソースプログラムをコンパイルして、シェーダオブジェクトを作成します。ここではバーテックスシェーダを例に説明します。フラグメントシェーダのシェーダオブジェクトも、同じ手順で作成します。

まずソースプログラムの各行を文字列にして、次のように配列に格納します。この 1 行目の #version の行のように、行頭が '#' の行の行末には '¥n' が必要です。

```
  static const GLchar *vsrc[] =
  {
    "#version 150 core¥n",
    "in vec4 position;",
    "void main()",
    "{",
    "  gl_Position = position;",
    "}",
  };
```

そして以下の手順により、ソースプログラムからシェーダオブジェクト (vobj) を作成します。

sizeof vsrc / sizeof vsrc[0] により配列変数 vsrc の要素数 (行数) を求めています。なお、これはバーテックスシェーダの場合です。フラグメントシェーダのシェーダオブジェクトを作成する場合は、GL_VERTEX_SHADER を GL_FRAGMENT_SHADER に書き換えます。

```
const GLuint vobj(glCreateShader(GL_VERTEX_SHADER));
glShaderSource(vobj, sizeof vsrc / sizeof vsrc[0], vsrc, NULL);
glCompileShader(vobj);
```

シェーダのソースプログラムが単一の文字列なら、行数を 1 として、その文字列のポインタを格納した変数のポインタを用います。ただし、この書き方の場合は各行の行末に '¥n' を入れないと、エラーメッセージの行番号がどれも 2 になってしまいます。

```
static const GLchar *vsrc =
  "#version 150 core¥n"
  "in vec4 position;¥n"
  "void main()¥n"
  "{¥n"
  "  gl_Position = position;¥n"
  "}¥n";
```

この場合は次のようにしてソースプログラムの文字列を読み込みます。

```
const GLuint vobj(glCreateShader(GL_VERTEX_SHADER));
glShaderSource(vsrc, 1, &vsrc, NULL);
glCompileShader(vobj);
```

GLuint glCreateShader(GLenum shaderType)

シェーダオブジェクトを作成します。戻り値は作成されたシェーダオブジェクト名 (番号) で、作成できれば 0 でない正の整数、作成できなければ 0 を返します。

shaderType

作成するシェーダの種類を指定します。バーテックスシェーダのシェーダオブジェクトを作成する場合は GL_VERTEX_SHADER、フラグメントシェーダのシェーダオブジェクトを作成する場合は GL_FRAGMENT_SHADER を指定します。

void glShaderSource(GLuint shader, GLsizei count, const GLchar **string, const GLint *length)

シェーダのソースプログラムを読み込みます。

shader

glCreateShader() 関数の戻り値として得たシェーダオブジェクト名。

count

引数 string に指定した配列の要素数。

string

シェーダのソースプログラムの文字列の配列。行頭が '#' の行の行末には '¥n' が必要です。

length

引数 length が NULL (0) でなければ、length の各要素には string の対応する要素の文字列の長さ (文字数) を格納します。また、この length の要素に負の値を格納したときは、string の対応する要素の文字列の終端がヌル文字 ('¥0') になっている必要があります。

void glCompileShader(GLuint shader)

シェーダオブジェクトに読み込まれたソースファイルをコンパイルします。

shader

コンパイルするシェーダオブジェクト名。

シェーダのソースプログラムのコンパイルに成功したなら、それをプログラムオブジェクトに組み込みます。プログラムオブジェクトに組み込んだシェーダオブジェクトは、ほかに使うあてがなければもう不要なので、ここで削除マークをつけておきます。

```
glAttachShader(program, vobj);
glDeleteShader(vobj);
```

void glAttachShader(GLuint program, GLuint shader)

プログラムオブジェクトにシェーダオブジェクトを組み込みます。

program

シェーダオブジェクトを組み込むプログラムオブジェクト名。

shader

組み込むシェーダオブジェクト名。

void glDetachShader(GLuint program, GLuint shader)

プログラムオブジェクトからシェーダオブジェクトを取り外します。

program

シェーダオブジェクトを取り外すプログラムオブジェクト名。

shader

取り外すシェーダオブジェクト名。

void glDeleteShader(GLuint shader)

シェーダオブジェクトに削除マークを付けます。

shader

削除マークを付けるシェーダオブジェクト名。

void glDeleteProgram(GLuint program)

プログラムオブジェクトに削除マークを付けます。

program

削除マークを付けるプログラムオブジェクト名。

glDeleteShader() 関数で削除マークを付けたシェーダオブジェクトは、そのシェーダオブジェクトが全てのプログラムオブジェクトから glDetachShader() 関数によって取り外されたときに削除されます。またプログラムオブジェクトが削除されるときには、それに組み込まれているシェーダオブジェクトは自動的に取り外されます。したがって、削除マークが付けられたシェーダオブジェクトは、それを組み込んだプログラムオブジェクトが全て削除された時点で削除されます。

プログラムオブジェクトに削除マークを付けるには、glDeleteProgram() 関数を用います。削除マークを付けたプログラムオブジェクトは、どのレンダリングコンテキストでも使用されなくなったときに削除されます。

4.2.5 プログラムオブジェクトのリンク

プログラムオブジェクトに必要なシェーダオブジェクトを組み込んだら、プログラムオブジェクトのリンクを行います。その前に、図形の頂点の位置を受け取る attribute 変数 (変数名 position) の場所 (index) と、フラグメントシェーダから出力するデータを格納する変数 (変数名 fragment) の出力先 (colorNumber) を指定しておきます。これらはいずれも 0 以上の整数です。

CPU 側のプログラムと GPU 側のシェーダプログラムの間のデータの受け渡しは、このような番号を介して行います。これは GPU のハードウェアのレジスタ番号に相当します。

```
glBindAttribLocation(program, 0, "position");
glBindFragDataLocation(program, 0, "fragment");
glLinkProgram(program);
```

void glBindAttribLocation(GLuint program, GLuint index, const GLchar *name)

バーテックスシェーダのソースプログラム中の attribute 変数の場所を指定します。attribute 変数は、バーテックスシェーダのソースプログラム内では in 変数として宣言されます。

program

attribute 変数の場所を調べるプログラムオブジェクト名。

index

引数 name に指定した attribute 変数の場所 (番号) を、0 以上の整数で指定します。デフォルトでは 0 が設定されています。

name

バーテックスシェーダのソースプログラム中の attribute 変数の変数名の文字列。

void glBindFragDataLocation(GLuint program, GLuint colorNumber, const char *name)

フラグメントシェーダのソースプログラム中の out 変数にカラーバッファを割り当てます。

program

フラグメントの出力変数 (out 変数) の場所を調べるプログラムオブジェクト名。

index

引数 name に指定した out 変数の場所 (番号) を、0 以上の整数で指定します。デフォルトでは 0 が設定されています。0 は標準のフレームバッファのカラーバッファを示します。

name

フラグメントシェーダのソースプログラム中の out 変数の変数名の文字列。

void glLinkProgram(GLuint program)

program に指定したプログラムオブジェクトをリンクします。これが成功すれば、シェーダプログラムが完成します。

program

リンクするプログラムオブジェクト名。

4.2.6 プログラムオブジェクトを作成する手続き

以上の手順を、次のように createProgram() という関数関数にまとめます。

● メインプログラム (main.cpp) の変更点

createProgram() を、main.cpp の main() 関数より前で定義します。ソースプログラムの文字列が NULL の時は、シェーダオブジェクトを作成しないようにします。

```
#include <cstdlib>
#include <iostream>
#include <GL/glew.h>
#include <GLFW/glfw3.h>

// プログラムオブジェクトを作成する
//   vsrc: バーテックスシェーダのソースプログラムの文字列
//   fsrc: フラグメントシェーダのソースプログラムの文字列
GLuint createProgram(const char *vsrc, const char *fsrc)
{
  // 空のプログラムオブジェクトを作成する
  const GLuint program(glCreateProgram());

  if (vsrc != NULL)
  {
    // バーテックスシェーダのシェーダオブジェクトを作成する
    const GLuint vobj(glCreateShader(GL_VERTEX_SHADER));
    glShaderSource(vobj, 1, &vsrc, NULL);
    glCompileShader(vobj);

    // バーテックスシェーダのシェーダオブジェクトをプログラムオブジェクトに組み込む
    glAttachShader(program, vobj);
    glDeleteShader(vobj);
  }
```

```
  if (fsrc != NULL)
  {
    // フラグメントシェーダのシェーダオブジェクトを作成する
    const GLuint fobj(glCreateShader(GL_FRAGMENT_SHADER));
    glShaderSource(fobj, 1, &fsrc, NULL);
    glCompileShader(fobj);

    // フラグメントシェーダのシェーダオブジェクトをプログラムオブジェクトに組み込む
    glAttachShader(program, fobj);
    glDeleteShader(fobj);
  }

  // プログラムオブジェクトをリンクする
  glBindAttribLocation(program, 0, "position");
  glBindFragDataLocation(program, 0, "fragment");
  glLinkProgram(program);

  // 作成したプログラムオブジェクトを返す
  return program;
}
```

4.2.7 シェーダプログラムの使用

図形の描画を行う前に、glUseProgram() 関数で使用するプログラムオブジェクトを指定します。

```
  glUseProgram(program);
```

void glUseProgram(GLuint program)

図形の描画に使用するプログラムオブジェクトを指定します。

program

図形の描画に使用するプログラムオブジェクト名。0 を指定すると、どのプログラムオブジェクトも使用されなくなります。

● メインプログラム (main.cpp) の変更点

main() 関数では、バーテックスシェーダとフラグメントシェーダのそれぞれのソースプログラムの文字列を用意し、GLEW の初期化を行った後に createProgram() を実行します。そして図形の描画を行う前に、この関数の戻り値として得たプログラムオブジェクト名を glUseProgram() 関数の引数に指定して、このシェーダプログラムの使用を開始します。

```
int main()
{
  《省略》

  // GLEW を初期化する
  glewExperimental = GL_TRUE;
  if (glewInit() != GLEW_OK)
  {
    // GLEW の初期化に失敗した
```

```
    std::cerr << "Can't initialize GLEW" << std::endl;
    return 1;
  }

  // 背景色を指定する
  glClearColor(1.0f, 1.0f, 1.0f, 0.0f);

  // バーテックスシェーダのソースプログラム
  static const GLchar vsrc[] =
    "#version 150 core¥n"
    "in vec4 position;¥n"
    "void main()¥n"
    "{¥n"
    "  gl_Position = position;¥n"
    "}¥n";

  // フラグメントシェーダのソースプログラム
  static const GLchar fsrc[] =
    "#version 150 core¥n"
    "out vec4 fragment;¥n"
    "void main()¥n"
    "{¥n"
    "  fragment = vec4(1.0, 0.0, 0.0, 1.0);¥n"
    "}¥n";

  // プログラムオブジェクトを作成する
  const GLuint program(createProgram(vsrc, fsrc));

  // ウィンドウが開いている間繰り返す
  while (glfwWindowShouldClose(window) == GL_FALSE)
  {
    // ウィンドウを消去する
    glClear(GL_COLOR_BUFFER_BIT);

    // シェーダプログラムの使用開始
    glUseProgram(program);

    //
    // ここで描画処理を行う
    //

    // カラーバッファを入れ替える
    glfwSwapBuffers(window);

    // イベントを取り出す
    glfwWaitEvents();
  }
}
```

■ サンプルプログラム step02

補足: glUseProgram() 関数を実行する位置

glUseProgram(program); は、使用するシェーダプログラムが一つしかなければ、while ループの前に置くことができます。シェーダプログラムを描画命令のたびに切り替えて使う場合は、この

while ループの中に置いてください。なお、シェーダプログラムが不要になれば glUseProgram(0); を実行しますが、シェーダプログラムは使いっぱなしでも構いません。

4.2.8 エラーメッセージの表示

ここまではシェーダのコンパイルやリンクの時にエラーチェックを行っていませんでした。GLSL といえどもプログラミング言語なので、書き間違えればエラーが発生します。その時、エラーメッセージがわからなければ、間違いを見つけることが難しくなります。

● メインプログラム (main.cpp) の変更点

そこで glGetShaderiv() 関数を使ってコンパイル時のエラーをチェックし、エラーが発生していれば glGetShaderInfoLog() でログを取り出して、エラーメッセージを表示する関数を作成します。関数名は printShaderInfoLog() とします。この関数の戻り値は、エラーが発生しなければ GL_TRUE、発生すれば GL_FALSE にします。

なお、この関数では C++ の標準テンプレートライブラリに含まれる vector を使用しているので、main.cpp の冒頭で vector を #include します。

```
#include <cstdlib>
#include <iostream>
#include <vector>
#include <GL/glew.h>
#include <GLFW/glfw3.h>

// シェーダオブジェクトのコンパイル結果を表示する
//   shader: シェーダオブジェクト名
//   str: コンパイルエラーが発生した場所を示す文字列
GLboolean printShaderInfoLog(GLuint shader, const char *str)
{
  // コンパイル結果を取得する
  GLint status;
  glGetShaderiv(shader, GL_COMPILE_STATUS, &status);
  if (status == GL_FALSE) std::cerr << "Compile Error in " << str << std::endl;

  // シェーダのコンパイル時のログの長さを取得する
  GLsizei bufSize;
  glGetShaderiv(shader, GL_INFO_LOG_LENGTH , &bufSize);

  if (bufSize > 1)
  {
    // シェーダのコンパイル時のログの内容を取得する
    std::vector<GLchar> infoLog(bufSize);
    GLsizei length;
    glGetShaderInfoLog(shader, bufSize, &length, &infoLog[0]);
    std::cerr << &infoLog[0] << std::endl;
  }

  return static_cast<GLboolean>(status);
}
```

void glGetShaderiv(GLuint shader, GLenum pname, GLint *params)

シェーダオブジェクトの情報を取り出します。

shader

情報を取り出すシェーダオブジェクト。

pname

シェーダオブジェクトから取り出す情報の種類。以下のものが指定できます。

GL_SHADER_TYPE

shader に指定したシェーダオブジェクトのシェーダの種類 (GL_VERTEX_SHADER, GL_FRAGMENT_SHADER) を調べて *params に格納します。

GL_DELETE_STATUS

shader に指定したシェーダオブジェクトに glDeleteShader() 関数によって削除マークが付けられているかどうかを調べて、削除マークがついていれば GL_TRUE、ついていなければ GL_FALSE を *params に格納します。

GL_COMPILE_STATUS

shader に指定したシェーダオブジェクトのコンパイルが成功したかどうかを調べて、成功していれば GL_TRUE、失敗していれば GL_FALSE を *params に格納します。

GL_INFO_LOG_LENGTH

shader に指定したシェーダオブジェクトのコンパイル時に生成されたログの長さを調べて *params に格納します。ログがなければ 0 を格納します。

GL_SHADER_SOURCE_LENGTH

shader に指定したシェーダオブジェクトのソースプログラムの長さを調べて *params に格納します。ソースプログラムがなければ 0 を格納します。

params

取り出した情報の格納先。

void glGetShaderInfoLog(GLuint shader, GLsizei maxLength, GLsizei *length, GLchar *infoLog)

シェーダオブジェクトのコンパイル時のログを取り出します。

shader

ログを取り出すシェーダオブジェクト。

maxLength

取り出すログの最大の長さ。引数 infoLog に指定するログの格納先の大きさは、これより小さくなければなりません。

length

取り出したログの実際の長さの格納先。

infoLog

取り出したログの格納先。

同様に、リンクの際には glGetProgramiv() を使ってエラーの発生をチェックし、エラーが発生していれば glGetShaderInfoLog() でログを取り出して、エラーメッセージを表示します。

なお、リンクが成功しても作成されたシェーダのプログラムオブジェクトが実行できない場合があります。これは描画を実行する前に glValidateProgram() を使って実行可能かどうかを調べて glGetProgramiv() で判定することができますが、ここではその処理は省略します。

```
// プログラムオブジェクトのリンク結果を表示する
//   program: プログラムオブジェクト名
GLboolean printProgramInfoLog(GLuint program)
{
  // リンク結果を取得する
  GLint status;
  glGetProgramiv(program, GL_LINK_STATUS, &status);
  if (status == GL_FALSE) std::cerr << "Link Error." << std::endl;

  // シェーダのリンク時のログの長さを取得する
  GLsizei bufSize;
  glGetProgramiv(program, GL_INFO_LOG_LENGTH, &bufSize);

  if (bufSize > 1)
  {
    // シェーダのリンク時のログの内容を取得する
    std::vector<GLchar> infoLog(bufSize);
    GLsizei length;
    glGetProgramInfoLog(program, bufSize, &length, &infoLog[0]);
    std::cerr << &infoLog[0] << std::endl;
  }

  return static_cast<GLboolean>(status);
}
```

void glGetProgramiv(GLuint program, GLenum pname, GLint *params)

プログラムオブジェクトの情報を取り出します。

program

情報を取り出すプログラムオブジェクト。

pname

プログラムオブジェクトから取り出す情報の種類。以下のものが指定できます。これら以外にもありますが、ここでは割愛します。

GL_DELETE_STATUS

program に指定したプログラムオブジェクトに glDeleteProgram() 関数によって削除マークが付けられているかどうかを調べて、削除マークがついていれば GL_TRUE、ついていなければ GL_FALSE を *params に格納します。

GL_LINK_STATUS

program に指定したプログラムオブジェクトのリンクが成功したかどうかを調べて、成功していれば GL_TRUE、失敗していれば GL_FALSE を *params に格納します。

GL_VALIDATE_STATUS

program に指定したプログラムオブジェクトに対する glValidateProgram() 関数による検証結果を、*params に格納します。*params にはプログラムオブジェクトが現在の OpenGL の状態で実行可能なら GL_TRUE、実行できなければ GL_FALSE が格納されます。

GL_INFO_LOG_LENGTH

program に指定したプログラムオブジェクトのリンク時に生成されたログの長さを調べて *params に格納します。ログがなければ 0 を格納します。

GL_ATTACHED_SHADERS

program に指定したプログラムオブジェクトに組み込まれているシェーダオブジェクトの数を調べて *params に格納します。

params

取り出した情報の格納先。

void glValidateProgram(GLuint program)

プログラムオブジェクトが現在の OpenGL の状態で実行可能かどうかを検証します。結果は glGetProgramiv() 関数の引数 pname に GL_VALIDATE_STATUS を指定して取り出します。

program

検証するプログラムオブジェクト。

void glGetProgramInfoLog(GLuint program, GLsizei maxLength, GLsizei *length, GLchar *infoLog)

シェーダオブジェクトのコンパイル時のログを取り出します。

program

ログを取り出すプログラムオブジェクト。

maxLength

取り出すログの最大の長さ。引数 infoLog に指定した格納先の大きさを超えてはなりません。

length

取り出したログの実際の長さ (infoLog に格納された長さ) の格納先。

infoLog

ログの格納先。格納先の大きさは引数 maxLength よりも大きくなければなりません。

createProgram() 関数において、シェーダのコンパイル直後のシェーダオブジェクトに対して printShaderInfoLog() を使ってエラーをチェックします。またリンク直後のプログラムオブジェク

トに対して printProgramInfoLog() を使ってエラーをチェックします。もしリンクに失敗していれば作成したプログラムオブジェクトを削除し、0 を返します。

```
// プログラムオブジェクトを作成する
//   vsrc: バーテックスシェーダのソースプログラムの文字列
//   fsrc: フラグメントシェーダのソースプログラムの文字列
GLuint createProgram(const char *vsrc, const char *fsrc)
{
  // 空のプログラムオブジェクトを作成する
  const GLuint program(glCreateProgram());

  if (vsrc != NULL)
  {
    // バーテックスシェーダのシェーダオブジェクトを作成する
    const GLuint vobj(glCreateShader(GL_VERTEX_SHADER));
    glShaderSource(vobj, 1, &vsrc, NULL);
    glCompileShader(vobj);

    // バーテックスシェーダのシェーダオブジェクトをプログラムオブジェクトに組み込む
    if (printShaderInfoLog(vobj, "vertex shader"))
      glAttachShader(program, vobj);
    glDeleteShader(vobj);
  }

  if (fsrc != NULL)
  {
    // フラグメントシェーダのシェーダオブジェクトを作成する
    const GLuint fobj(glCreateShader(GL_FRAGMENT_SHADER));
    glShaderSource(fobj, 1, &fsrc, NULL);
    glCompileShader(fobj);

    // フラグメントシェーダのシェーダオブジェクトをプログラムオブジェクトに組み込む
    if (printShaderInfoLog(fobj, "fragment shader"))
      glAttachShader(program, fobj);
    glDeleteShader(fobj);
  }

  // プログラムオブジェクトをリンクする
  glBindAttribLocation(program, 0, "position");
  glBindFragDataLocation(program, 0, "fragment");
  glLinkProgram(program);

  // 作成したプログラムオブジェクトを返す
  if (printProgramInfoLog(program))
    return program;

  // プログラムオブジェクトが作成できなければ 0 を返す
  glDeleteProgram(program);
  return 0;
}
```

■ サンプルプログラム step03

4.2.9　シェーダのソースプログラムを別のファイルから読み込む

前のプログラムでは、GLSL で書いたシェーダのソースプログラムを文字列の形で C++ のソースプログラムに埋め込みました。この方法はシェーダのソースプログラムと C++ のソースプログラムを一つのファイルにまとめることができるので便利と言えば便利ですが、シェーダのソースプログラムを文字列で表さなければならないのは、やっぱり面倒です。

● メインプログラム (main.cpp) の変更点

そこで、シェーダのソースプログラムを別のファイルにして、C++ のプログラムから実行時に読み込むようにします。こうすると、シェーダのソースプログラムに変更を加えたときに C++ のプログラムをコンパイルし直さずにすみます。

main.cpp の createProgram() 関数の後に、readShaderSource() という関数を追加します。この関数ではファイルの入出力を行いますので、main.cpp の冒頭で fstream を #include します。

```
#include <cstdlib>
#include <iostream>
#include <fstream>
#include <vector>
#include <GL/glew.h>
#include <GLFW/glfw3.h>

《省略》

// プログラムオブジェクトを作成する
//   vsrc: バーテックスシェーダのソースプログラムの文字列
//   fsrc: フラグメントシェーダのソースプログラムの文字列
GLuint createProgram(const char *vsrc, const char *fsrc)
{
  《省略》
}

// シェーダのソースファイルを読み込んだメモリを返す
//   name: シェーダのソースファイル名
//   buffer: 読み込んだソースファイルのテキスト
GLchar *readShaderSource(const char *name, std::vector<GLchar> &buffer)
{
  // ファイル名が NULL だった
  if (name == NULL) return false;

  // ソースファイルを開く
  std::ifstream file(name, std::ios::binary);
  if (file.fail())
  {
    // 開けなかった
    std::cerr << "Error: Can't open source file: " << name << std::endl;
    return false;
  }

  // ファイルの末尾に移動し現在位置（＝ファイルサイズ）を得る
  file.seekg(0L, std::ios::end);
```

```
  GLsizei length = static_cast<GLsizei>(file.tellg());

  // ファイルサイズのメモリを確保
  buffer.resize(length + 1);

  // ファイルを先頭から読み込む
  file.seekg(0L, std::ios::beg);
  file.read(buffer.data(), length);
  buffer[length] = '¥0';

  if (file.fail())
  {
    // うまく読み込めなかった
    std::cerr << "Error: Could not read souce file: " << name << std::endl;
    file.close();
    return false;
  }

  // 読み込み成功
  file.close();
  return true;
}
```

さらに、この関数を使ってシェーダのソースファイルを読み込み、createProgram() 関数を使ってプログラムオブジェクトを作成する関数 loadProgram() を、この後ろに追加します。

```
// シェーダのソースファイルを読み込んでプログラムオブジェクトを作成する
//   vert: バーテックスシェーダのソースファイル名
//   frag: フラグメントシェーダのソースファイル名
GLuint loadProgram(const char *vert, const char *frag)
{
  // シェーダのソースファイルを読み込む
  std::vector<GLchar> vsrc;
  const bool vstat(readShaderSource(vert, vsrc));
  std::vector<GLchar> fsrc;
  const bool fstat(readShaderSource(frag, fsrc));

  // プログラムオブジェクトを作成する
  return vstat && fstat ? createProgram(vsrc.data(), fsrc.data()) : 0;
}
```

main() 関数では、createProgram() を loadProgram() に置き換えます。また、バーテックスシェーダのソースプログラムの文字列 vsrc とフラグメントシェーダのソースプログラムの文字列 fsrc は削除します。

```
int main()
{
  《省略》

  // 背景色を指定する
  glClearColor(1.0f, 1.0f, 1.0f, 0.0f);

  // プログラムオブジェクトを作成する
  const GLuint program(loadProgram("point.vert", "point.frag"));
```

```
  // ウィンドウが開いている間繰り返す
  while (glfwWindowShouldClose(window) == GL_FALSE)
  {
    《省略》
  }
}
```

● シェーダのソースファイル

シェーダのソースファイルは、C++ のソースファイルとは別に作成します。バーテックスシェーダのソースファイルのファイル名は point.vert、フラグメントシェーダのソースファイルのファイル名は point.frag にします (それぞれの拡張子は適当です)。これらを C++ のソースファイルと同じディレクトリ (フォルダ) に保存してください。

バーテックスシェーダのソースファイル point.vert

```
#version 150 core
in vec4 position;
void main()
{
  gl_Position = position;
}
```

フラグメントシェーダのソースファイル point.frag

```
#version 150 core
out vec4 fragment;
void main()
{
  fragment = vec4(1.0, 0.0, 0.0, 1.0);
}
```

■ サンプルプログラム step04

補足: Visual Studio によるシェーダのソースファイルの作成

シェーダのソースファイルも C++ のソースファイルと同じテキストファイルなので、C++ のソースファイルと同じ手順 (図 10、図 11) で作成します。ただし、ここで使っている .vert や .frag という拡張子のファイルを作成する項目は Visual Studio にはないので、ファイル名は拡張子まで含めて指定する必要があります (図 33)。

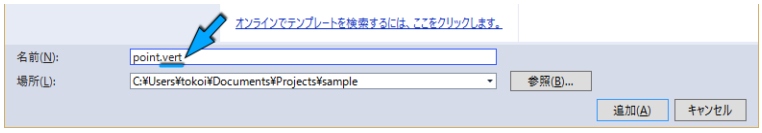

図 33 Visual Studio におけるシェーダのソースファイルのファイル名の指定

補足: Xcode によるシェーダのソースファイルの作成

Xcode でプロジェクトにシェーダのソースファイルを追加するには、“File” メニューから “New” を選び、その先の “File…” の項目を選んでください (図 34)。

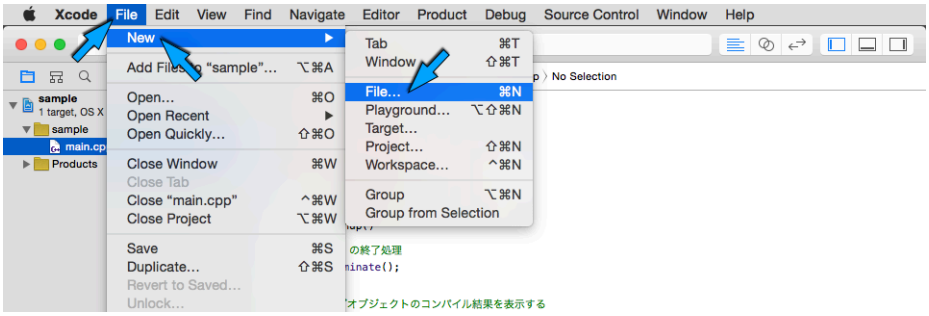

図 34 Xcode でのファイルの新規作成

シェーダのソースファイルは、テキストファイルとして作成します。“OS X” の “Other” にある “Empty” を選んで、“Next” をクリックします (図 35)。

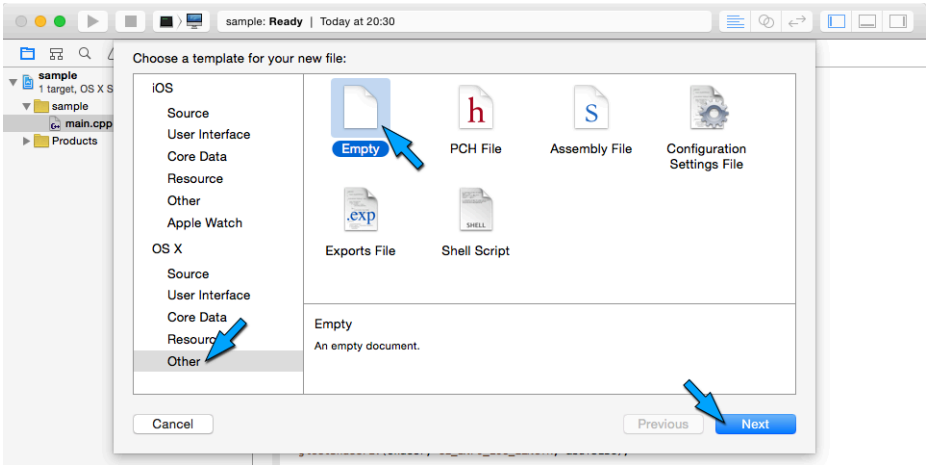

図 35 Xcode での空のファイルの作成

“Save as” に指定するファイル名は .vert や .frag という拡張子まで含めて指定してください (図 36)。その後、“Create” をクリックすれば、空のファイルがプロジェクトに追加されます。これにシェーダのソースプログラムを入力します。

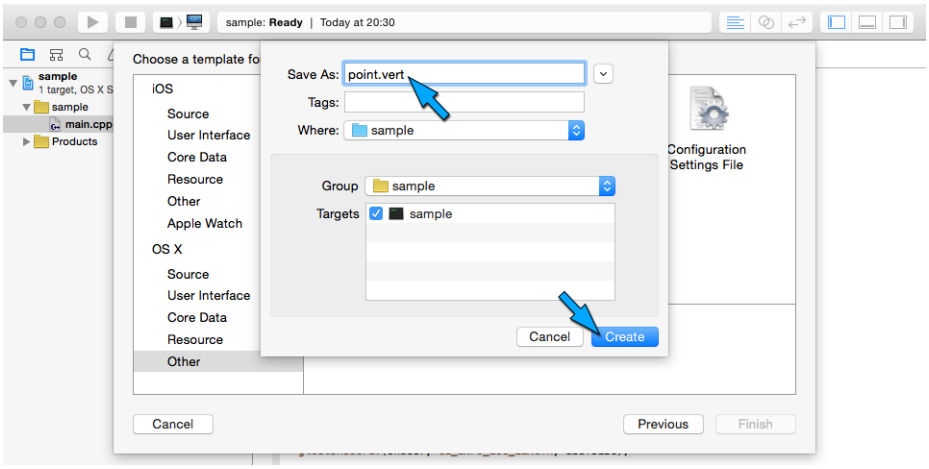

図 36 Xcode でのシェーダのソースファイルのファイル名の指定

Xcode を使ってビルドした場合、実行ファイルは C++ のソースファイルとは異なる場所に保存されます。また、そこはプログラムを Xcode 内で実行する時の作業ディレクトリになります。このため、シェーダのソースファイルを C++ のソースファイルと同じディレクトリに置いた場合は、プログラム中のシェーダのソースファイルのファイル名にそのディレクトリのパスを含めないと、読み込むことができません。

そこで、プログラムを Xcode 内で実行したときの作業ディレクトリが、シェーダや C++ のソースファイルを置いたディレクトリになるよう、Xcode の設定を変更します。Xcode の左上の“Set Active Scheme”から“Edit Scheme”を選んでください (図 37)。

図 37 スキームの設定

左のペインの“Run プロジェクト名”を選び、上の“Options”を選択して“Working Directory”にチェック☑を入れてください。その後、その下の欄の右側のフォルダのアイコンをクリックすれば、フォルダ選択のダイアログが現れますので、シェーダや C++ のソースファイルを置いたディレクトリを選択してください。あるいは、この欄に $PROJECT_DIR を設定すれば、Xcode のプロジェクトのディレクトリが使用されます。最後に“OK”をクリックしてください (図 38)。

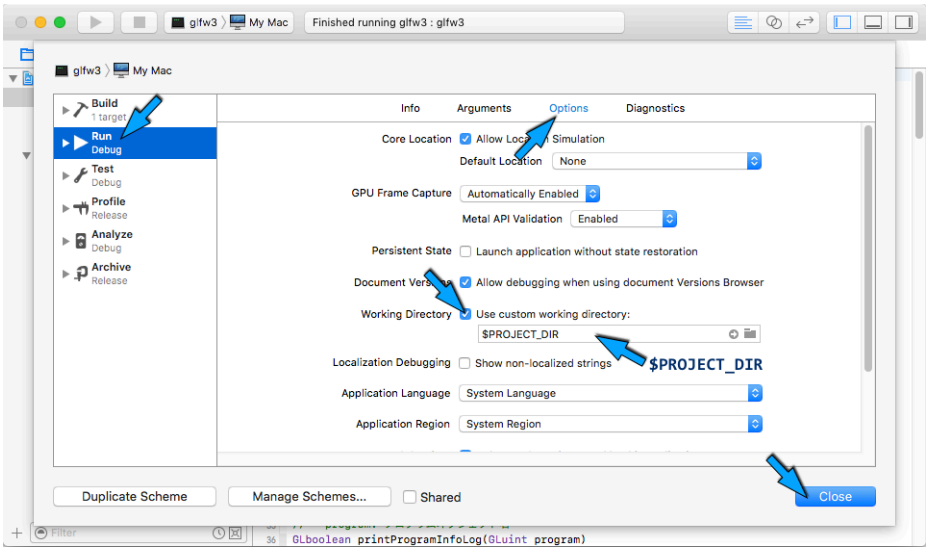

図 38 作業ディレクトリ (Working Directory) の設定

第5章 図形の描画

5.1 OpenGL の図形データ

OpenGL では、図形のデータは頂点の位置や色、法線ベクトルなどの頂点属性 (attribute) で表されます (4.2.2)。図形を描画するには、この頂点属性を一旦 GPU 側のメモリに転送します。この GPU 側のメモリを**頂点バッファ**といいます。また、この頂点バッファオブジェクトを管理する機構を、**頂点バッファオブジェクト** (Vertex Buffer Object, VBO) といいます。

図形の描画に用いる頂点属性には、位置や色のほか、法線ベクトルやテクスチャ座標など、必要に応じて様々なものが与えられます。したがって実際に図形を描画するには、複数の頂点属性、すなわち頂点バッファオブジェクトを組み合わせる必要があります。この組み合わせを管理するために、**頂点配列オブジェクト** (Vertex Array Object, VAO) を用います。

5.2 図形データの描画

5.2.1 図形データの描画手順

OpenGL では、以下の手順で図形を描画します。

(1) glGenVertexArrays() で頂点配列オブジェクトを作成する。
(2) 作成した頂点配列オブジェクトを glBindVertexArray() で結合する
(3) glGenBuffers() で頂点バッファオブジェクトを作成する。
(4) 作成した頂点バッファオブジェクトを glBindBuffer() で結合する
(5) glBufferData() で頂点バッファオブジェクトにデータ (頂点属性) を転送する
(6) 頂点バッファオブジェクトを glVertexAttribPointer() で attribute 変数に関連づける
(7) 頂点配列オブジェクトを結合して描画命令 (ドローコール) を実行する

頂点配列オブジェクトを結合 (glBindVertexArray()) した後、頂点バッファオブジェクトを結合 (glBindBuffer()) することによって、頂点配列オブジェクトに頂点バッファオブジェクトが組み込まれます。この後は頂点配列オブジェクトを結合すれば、組み込まれた頂点バッファオブジェクトや attribute 変数との対応付けが有効になります。

5.2.2 頂点バッファオブジェクトの作成

頂点配列オブジェクトに図形データを登録します。図形は 2 点 (-0.5, -0.5) と (0.5, 0.5) を対角の頂点とする四角形にします。これを Vertex という構造体に格納します。

```
// 頂点属性
struct Vertex
{
  // 位置
  GLfloat position[2];
};

// 図形データ
static const Vertex vertex[] =
{
  { -0.5f, -0.5f },
  {  0.5f, -0.5f },
  {  0.5f,  0.5f },
  { -0.5f,  0.5f }
};
```

頂点バッファオブジェクトを作成し、このデータをそこに送ります。頂点バッファオブジェクトは GPU 側に確保したデータの保存領域を管理します。転送するデータのサイズは、Vertex 型の頂点データが 4 個なので、4 * sizeof (Vertex) になります。

```
  GLuint vbo;
  glGenBuffers(1, &vbo);
  glBindBuffer(GL_ARRAY_BUFFER, vbo);
  glBufferData(GL_ARRAY_BUFFER, 4 * sizeof (Vertex), vertex, GL_STATIC_DRAW);
```

void glGenBuffers(GLsizei n, GLuint *buffers)

頂点バッファオブジェクトを作成します。

n

作成する頂点バッファオブジェクトの数。

arrays

作成した頂点バッファオブジェクト名 (番号) を格納する配列。少なくとも引数 n に指定した数以上の要素を持つ必要があります。

void glBindBuffer(GLenum target, GLuint buffer)

頂点バッファオブジェクトを結合して使用可能にします。

target

頂点バッファオブジェクトの結合対象。GL_ARRAY_BUFFER, GL_COPY_READ_BUFFER, GL_COPY_WRITE_BUFFER, GL_ELEMENT_ARRAY_BUFFER, GL_PIXEL_PACK_BUFFER, GL_PIXEL_UNPACK_BUFFER, GL_TEXTURE_BUFFE, GL_UNIFORM_BUFFE, GL_TRANSFORM_FEEDBACK_BUFFER が指定できます。頂点バッファオブジェクトの内

容を描画に使う頂点属性として使う場合は、GL_ARRAY_BUFFER を指定します。

buffer

結合する頂点バッファオブジェクト名。0 なら現在の結合を解除します。

void glBufferData(GLenum target, GLsizeiptr size, const GLvoid *data, GLenum usage)

頂点バッファオブジェクトのメモリを確保し、そこにデータ (頂点属性) を転送します。

target

データを転送する頂点バッファオブジェクトの結合対象。

size

GPU 側に確保する頂点バッファオブジェクトのサイズ。

data

頂点バッファオブジェクトに転送するデータが格納されている CPU 側のデータのポインタ。データが格納されているメモリ (配列変数等) のサイズは少なくとも引数 size バイトある必要がある。なお、これが 0 なら GPU 側に頂点バッファオブジェクトに使うメモリの確保だけを行い、データの転送は行わない。

usage

この頂点バッファオブジェクトの使われ方。GL_STREAM_DRAW、GL_STREAM_READ、GL_STREAM_COPY 、 GL_STATIC_DRAW 、 GL_STATIC_READ 、 GL_STATIC_COPY 、GL_DYNAMIC_DRAW、GL_DYNAMIC_READ、GL_DYNAMIC_COPY が指定できます。これは GPU の動作を最適化するためのヒントとして利用されます。

STREAM

データの保存領域には一度だけ書き込まれ、高々数回使用されます。

STATIC

データの保存領域には一度だけ書き込まれ、何度も使用されます。

DYNAMIC

データの保存領域には繰り返し書き込まれ、何度も使用されます。

DRAW

データの保存領域の内容はアプリケーションによって書き込まれ、描画のためのデータとして用いられます。

READ

データの保存領域の内容はアプリケーションからの問い合わせによって OpenGL (GPU) 側から読み出され、アプリケーション側に返されます。

COPY

データの保存領域の内容はアプリケーションからの指令によって OpenGL (GPU) 側から読み出され、描画のためのデータとして用いられます。

5.2.3 頂点バッファオブジェクトと attribute 変数の関連付け

頂点バッファオブジェクトに格納されているデータ (頂点属性) は、バーテックスシェーダの in 変数として宣言される attribute 変数 (4.2.2) を介して取り出します。attribute 変数がデータを取り出す頂点バッファオブジェクトは、glVertexAttribPointer() 関数を用いて指定します。そして glEnableVertexAttribArray() 関数によって、この attribute 変数を有効にします。

```
glVertexAttribPointer(0, 2, GL_FLOAT, GL_FALSE, 0, 0);
glEnableVertexAttribArray(0);
```

void glVertexAttribPointer(GLuint index, GLint size, GLenum type, GLboolean normalized, GLsizei stride, const GLvoid *pointer);

図形の描画時に attribute 変数が受け取るデータの格納場所と書式を指定します。

index

シェーダプログラムのリンク時に glBindAttribLocation() 関数で指定した、データを受け取る attribute 変数の場所 (index)。このシェーダプログラムでは vertex の唯一のメンバ position の index に 0 を指定していますから、ここでは 0 を指定しています。

size

attribute 変数が受け取る一個のデータのサイズを指定します。1、2、3、4 の値が指定できます。一個のデータが二次元 (x, y) なら 2、三次元 (x, y, z) なら 3 を指定します。このプログラムの図形データは二次元なので、ここでは 2 を指定しています。

type

attribute 変数が受け取る (pointer で示された先の) データ型を指定します。GL_BYTE、GL_UNSIGNED_BYTE、GL_SHORT、GL_UNSIGNED_SHORT、GL_INT、GL_UNSIGNED_INT、GL_FLOAT、GL_DOUBLE が指定できます。このプログラムの図形データは GLfloat 型なので、ここでは GL_FLOAT を指定しています。

normalized

GL_TRUE なら type が固定小数点型 (GL_BYTE、GL_UNSIGNED_BYTE、GL_SHORT、GL_UNSIGNED_SHORT、GL_INT、GL_UNSIGNED_INT) のとき、その値をそのデータ型で表現可能な最大値で正規化します。GL_FALSE なら正規化しません。ここでは正規化を行わないので、GL_FALSE を指定しています。

stride

attribute 変数が受け取る (pointer で示された先の) データの配列の、要素間の間隔を指定します。0 ならデータは密に並んでいるものとして扱います。ここでは 0 を指定しています。

pointer

attribute 変数が受け取るデータが格納されている場所を指定します。バイト単位のオフセットをポインタにキャストして渡します。頂点バッファオブジェクトの先頭から取り出すなら

0 を指定します。バイト単位のオフセットをポインタに直すには、offset を int 型の変数として、static_cast<char *>(0) + offset あるいは (char *)0 + offset とします。

void glEnableVertexAttribArray(GLuint index)

attribute 変数を有効にします。

index

有効にする attribute 変数の場所。

5.2.4 頂点配列オブジェクトの作成

頂点配列オブジェクトは、glGenVertexArrays() 関数を使って作成します。作成した頂点配列オブジェクトを使用するときは、glBindVertexArray() 関数を実行します。

```
GLuint vao;
glGenVertexArrays(1, &vao);
glBindVertexArray(vao);
```

void glGenVertexArrays(GLsizei n, GLuint *arrays)

頂点配列オブジェクトを作成します。

n

作成する頂点配列オブジェクトの数。

arrays

作成した頂点配列オブジェクト名 (番号) を格納する配列。少なくとも引数 n に指定した数以上の要素を持つ必要があります。

void glBindVertexArray(GLuint array)

頂点配列オブジェクトを結合して使用可能にします。

array

結合する頂点配列オブジェクト名。0 なら現在の結合を解除します。

● **頂点配列オブジェクトのクラス Object**

以上の手続きを定義したクラス Object を、Object.h というヘッダファイルに作成します (これ以降で作成するヘッダファイルの内容は、main.cpp に埋め込んでも構いません)。この private メンバ変数には、頂点配列オブジェクト名の vao と頂点バッファオブジェクト名の vbo を登録します。これらはこのクラス内でしか使わないので、private メンバにします。#pragma once は同じヘッダファイルが複数回読み込まれないようにするための「おまじない」です。

```
#pragma once

// 図形データ
```

```
class Object
{
  // 頂点配列オブジェクト名
  GLuint vao;

  // 頂点バッファオブジェクト名
  GLuint vbo;
```

public メンバには頂点属性の構造体 Vertex、コンストラクタ、デストラクタ、および描画を行うメソッドを定義します。コンストラクタでは頂点配列オブジェクトを作成した後、引数に与えられた頂点の位置の要素数 (次元) size とその頂点の数 vertexcount をもとに頂点バッファオブジェクトを作成し、それに頂点データ vertex を転送して頂点配列オブジェクトに組み込みます。

```
public:

  // 頂点属性
  struct Vertex
  {
    // 位置
    GLfloat position[2];
  };

  // コンストラクタ
  //   size: 頂点の位置の次元
  //   vertexcount: 頂点の数
  //   vertex: 頂点属性を格納した配列
  Object(GLint size, GLuint vertexcount, const Vertex *vertex)
  {
    // 頂点配列オブジェクト
    glGenVertexArrays(1, &vao);
    glBindVertexArray(vao);

    // 頂点バッファオブジェクト
    glGenBuffers(1, &vbo);
    glBindBuffer(GL_ARRAY_BUFFER, vbo);
    glBufferData(GL_ARRAY_BUFFER,
      vertexcount * sizeof (Vertex), vertex, GL_STATIC_DRAW);

    // 結合されている頂点バッファオブジェクトを in 変数から参照できるようにする
    glVertexAttribPointer(0, size, GL_FLOAT, GL_FALSE, 0, 0);
    glEnableVertexAttribArray(0);
  }
```

デストラクタでは、コンストラクタで作成した頂点配列オブジェクトと頂点バッファオブジェクトを忘れずに削除します。これは仮想関数にしておきます。

```
  // デストラクタ
  virtual ~Object()
  {
    // 頂点配列オブジェクトを削除する
    glDeleteBuffers(1, &vao);

    // 頂点バッファオブジェクトを削除する
```

```
    glDeleteBuffers(1, &vbo);
  }
```

ただし、これだとインスタンスのコピーが複数作られたとき、そのうちのどれか一つでも削除されると、それらの間で共有されている頂点配列オブジェクトや頂点バッファオブジェクトオブジェクトが削除されてしまいます。そうすると残りのインスタンスが使えなくなってしまうので、コピーコンストラクタと代入演算子を private メンバにしてインスタンスのコピーを禁止します。

```
private:

  // コピーコンストラクタによるコピー禁止
  Object(const Object &o);

  // 代入によるコピー禁止
  Object &operator=(const Object &o);
```

図形を描画するときは、あらかじめ glBindVertexArray() により頂点配列オブジェクトを結合しておく必要があるので、この処理を行うメソッド bind() を用意しておきます。これはインスタンスを変更しないので、const メソッドにします。

```
public:

  // 頂点配列オブジェクトの結合
  void bind() const
  {
    // 描画する頂点配列オブジェクトを指定する
    glBindVertexArray(vao);
  }
};
```

5.2.5 描画の実行

描画する図形データを保持した頂点配列オブジェクトを glBindVertexArray() 関数で指定し、glDrawArrays() 関数で基本図形の種類 (図 39) と頂点の数を指定して描画します。

```
    glBindVertexArray(vao);
    glDrawArrays(GL_LINE_LOOP, 0, 4);
```

void glDrawArrays(GLenum mode, GLint first, GLsizei count)

頂点配列を用いて図形を描画します。

mode

描画する基本図形の種類 (図 39)。

first

描画する頂点の先頭の番号。頂点バッファオブジェクトの先頭の頂点から描画するなら 0。

count

描画する頂点の数。たとえば四角形なら 4。

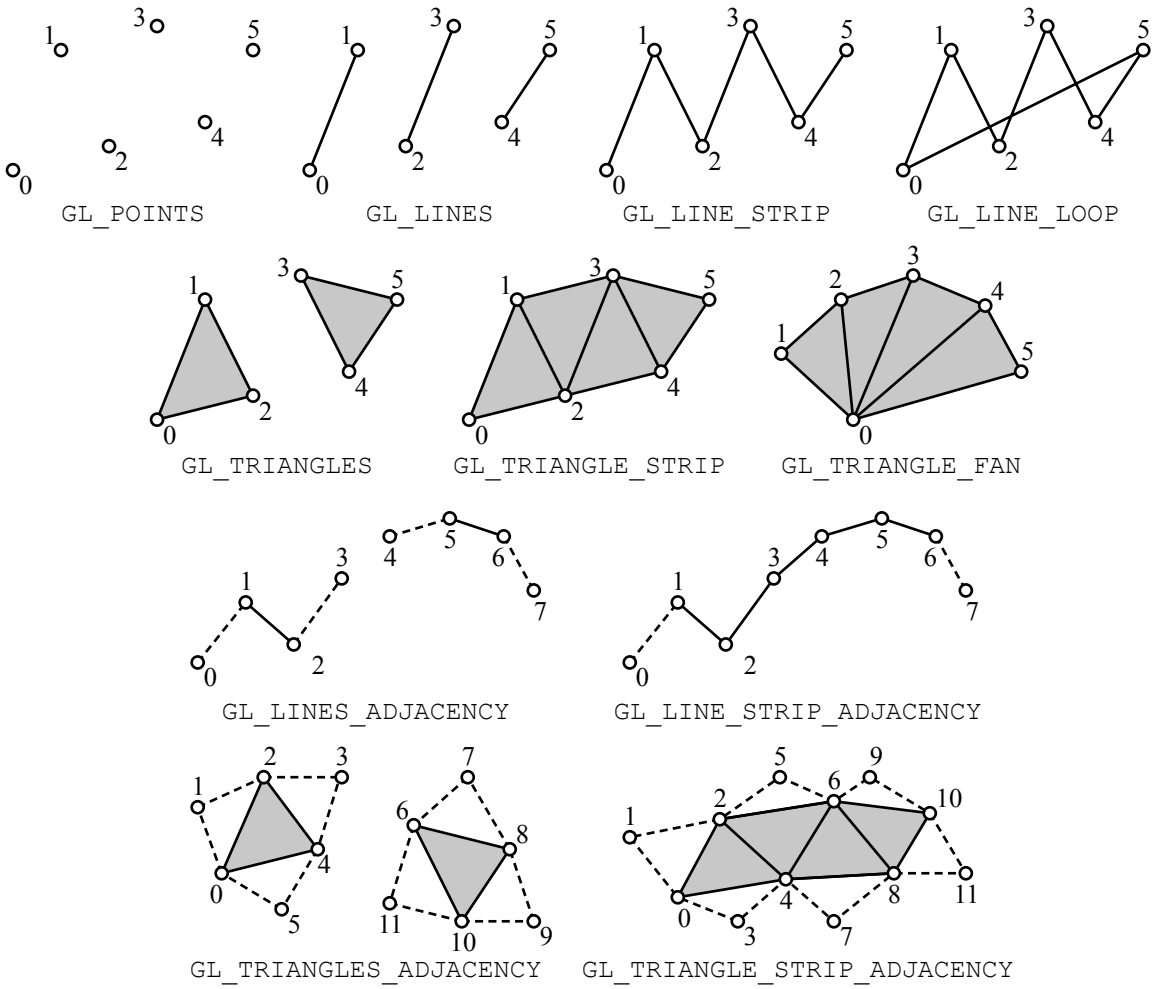

図 39 OpenGL の基本図形

● 図形の描画を行うクラス Shape

実際に図形の描画を行う Shape クラスを、Shape.h というファイルで定義します。このクラスは Object クラスのインスタンスを参照します。この参照にはスマートポインタ shared_ptr を使います。インスタンスのポインタを格納する変数 object を shared_ptr にしておけば、同じインスタンスを参照しているポインタが全て削除 (delete) された時に、インスタンス自体が削除されます。こうすることにより、コピーコンストラクタや代入によるインスタンスのコピーが可能になります。これを使うには、標準テンプレートライブラリの memory を #include します。

その後、Object.h を #include します。このほか、描画の時には頂点の数が必要になるので、これを保持する vertexcount というメンバも用意します。このメンバは派生クラスからも参照するので、protected にします。

```
#pragma once
#include <memory>

// 図形データ
#include "Object.h"
```

```
// 図形の描画
class Shape
{
  // 図形データ
  std::shared_ptr<const Object> object;

protected:

  // 描画に使う頂点の数
  GLsizei vertexcount;
```

このコンストラクタは引数で受け取った頂点属性を使って Object クラスのインスタンスを生成し、それにより object を初期化します。本体はありません。

```
public:

  // コンストラクタ
  //   size: 頂点の位置の次元
  //   vertexcount: 頂点の数
  //   vertex: 頂点属性を格納した配列
  Shape(GLint size, GLsizei vertexcount, const Object::Vertex *vertex)
    : object(new Object(size, vertexcount, vertex))
    , vertexcount(vertexcount)
  {
  }
```

描画処理では、先に基底クラス Object のメソッド bind() を呼び出して描画に使う頂点配列オブジェクトを結合したあと、描画を実行します。実際に描画を行うメソッド execute() は仮想関数にして、このクラスから派生したクラスでオーバーライドできるようにしておきます。

```
  // 描画
  void draw() const
  {
    // 頂点配列オブジェクトを結合する
    object->bind();

    // 描画を実行する
    execute();
  }

  // 描画の実行
  virtual void execute() const
  {
    // 折れ線で描画する
    glDrawArrays(GL_LINE_LOOP, 0, vertexcount);
  }
};
```

● メインプログラム (main.cpp) の変更点

図形の描画は Shape クラスのインスタンスを生成して行いますが、そのポインタをスマートポインタ unique_ptr にします。unique_ptr は shared_ptr と違って複数のポインタが同じインスタ

ンスを指すことはありませんが、ポインタが削除された時にはインスタンスも自動的に削除 (delete) されます。これを使うために、ここでも memory を #include します。

また、Shape クラスを定義しているヘッダファイル Shape.h を main.cpp の冒頭で #include します。その後、main() 関数の前に矩形の頂点の位置データ rectangleVertex を置き、main() 関数でこれを使って Shape クラスのインスタンスを作成します。これはスマートポインタ unique_ptr の変数 shape に格納します。ループの中でこの draw() メソッドを呼び出します。

```
#include <cstdlib>
#include <iostream>
#include <fstream>
#include <vector>
#include <memory>
#include <GL/glew.h>
#include <GLFW/glfw3.h>
#include "Shape.h"

《省略》

// 矩形の頂点の位置
const Object::Vertex rectangleVertex[] =
{
  { -0.5f, -0.5f },
  {  0.5f, -0.5f },
  {  0.5f,  0.5f },
  { -0.5f,  0.5f }
};

int main()
{
  《省略》

  // 背景色を指定する
  glClearColor(1.0f, 1.0f, 1.0f, 0.0f);

  // プログラムオブジェクトを作成する
  const GLuint program(loadProgram("point.vert", "point.frag"));

  // 図形データを作成する
  std::unique_ptr<const Shape> shape(new Shape(2, 4, rectangleVertex));

  // ウィンドウが開いている間繰り返す
  while (glfwWindowShouldClose(window) == GL_FALSE)
  {
    // ウィンドウを消去する
    glClear(GL_COLOR_BUFFER_BIT);

    // シェーダプログラムの使用開始
    glUseProgram(program);

    // 図形を描画する
    shape->draw();

    // カラーバッファを入れ替える
    glfwSwapBuffers(window);
```

```
    // イベントを取り出す
    glfwWaitEvents();
  }
}
```

■ サンプルプログラム step05

● 実行結果

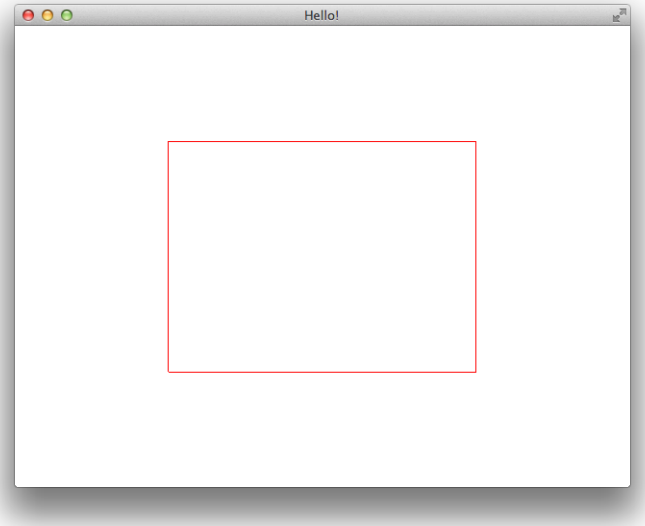

図 40 実行結果

第6章 マウスとキーボード

6.1 ウィンドウとビューポート

6.1.1 ビューポート変換

ここまでに作成したプログラムは、二点 (-0.5, -0.5) と (0.5, 0.5) を対角の頂点とする正方形のはずです。ところが表示されたウィンドウに描かれた図形は、幅と高さが一致していません。

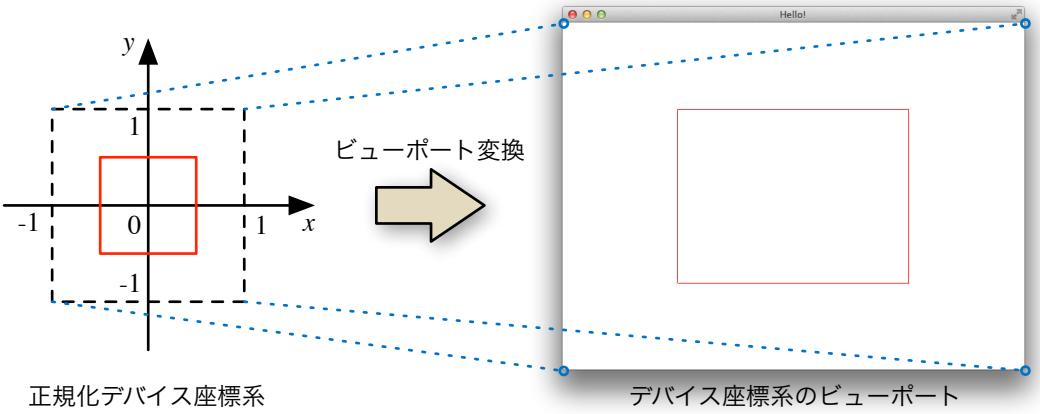

図 41 図形データと描画された図形

これは図 41 のように、図形データを定義している空間が (-1, -1) と (1, 1) を対角の頂点とする正方形となっていて、その領域が表示されたウィンドウ全体に投影されているからです。このウィンドウのサイズを変更すると、図形もそれに合わせて変形します (図 42)。

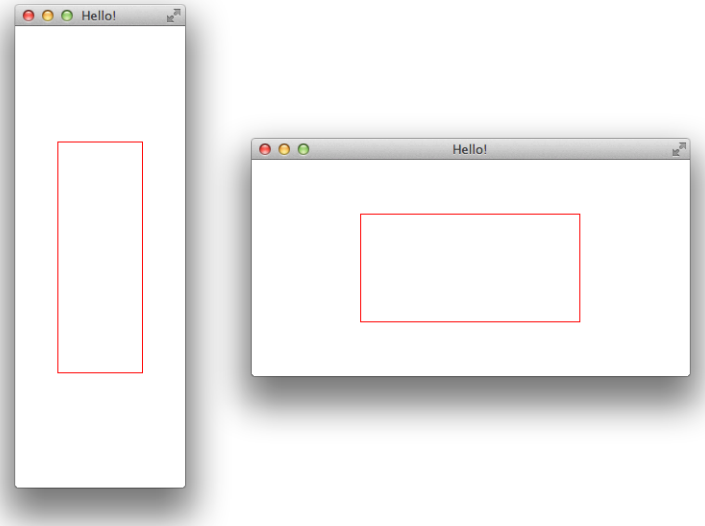

図 42 ウィンドウのサイズ変更による描画された図形の変形

この図形データを定義している空間の座標系を**正規化デバイス座標系** (Normalized Device Coordinate, NDC)、ウィンドウ上の座標系を**デバイス座標系** (Device Coordinate) といい、デバイス座標系上で図形の表示を行う領域を**ビューポート** (Viewport) といいます。glfwCreateWindow() 関数によりウィンドウを作成した直後は、ビューポートはウィンドウ全体に設定されています。

このように正規化デバイス座標系上の点の位置から、そのデバイス座標系上のビューポート内での位置を求める座標変換のことを、**ビューポート変換** (Viewport Transformation) といいます。

ビューポートの設定は、glViewport() 関数で行います。試しに main.cpp の main() 関数で、開いたウィンドウに対して図 43 のようにビューポートを設定してみてください。

```
  // 背景色を指定する
  glClearColor(1.0f, 1.0f, 1.0f, 0.0f);

  // ビューポートを設定する
  glViewport(100, 50, 300, 300);

  // プログラムオブジェクトを作成する
  const GLuint program(loadProgram("point.vert", "point.frag"));
```

void glViewport(GLint x, GLint y, GLsizei w, GLsizei h)

デバイス座標系上にビューポートを設定します。

x, y

ビューポートの左下隅の位置を指定します。デバイス座標系の原点は開いたウィンドウの左下隅にあり、そこからの相対位置を画素数で指定します。

w, h

ビューポートの幅と高さを画素数で指定します。w に負の数を指定すると、描画する図形の左右が反転します。h に負の数を指定すると、描画する図形の上下が反転します。

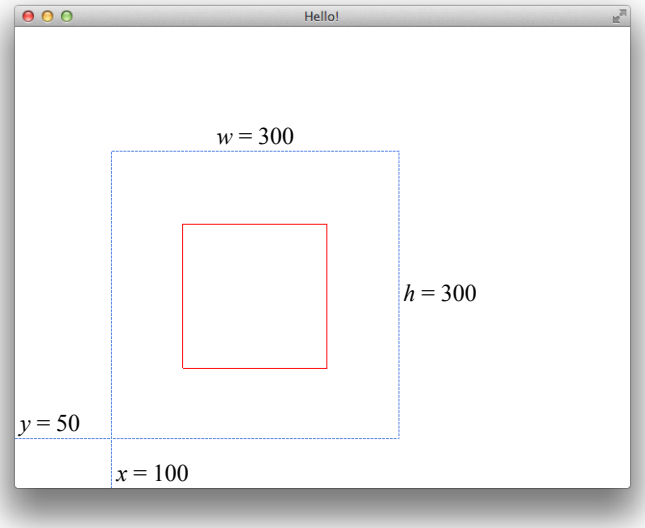

図 43 ビューポートの設定

このようにビューポートを正方形にすると、その中に描かれる図形の縦横比が正規化デバイス座標系上での縦横比と一致します。ただし、この例のように描画のループの前でビューポートを設定してしまうと、ウィンドウのサイズを変更したときに、その結果がビューポートに反映されず、それ以降は期待通りの描画が行われなくなることがあります。この対処法は後述します。

6.1.2 クリッピング

今度は main.cpp で定義している図形の頂点の位置を、次のように変更してみてください。

```
// 矩形の頂点の位置
const Object::Vertex rectangleVertex[] =
{
  { -0.5f, -0.5f },
  {  1.5f, -0.5f },
  {  1.5f,  1.5f },
  { -0.5f,  1.5f }
};
```

こうすると、図形の右上の頂点が (-1, -1) と (1, 1) を対角の頂点とする正方形の領域からはみ出ます。したがって、この図形を描画すると、ビューポートからはみ出た部分は図 44 のように刈り取られ、描画されません。この処理を**クリッピング** (Clipping) といいます。

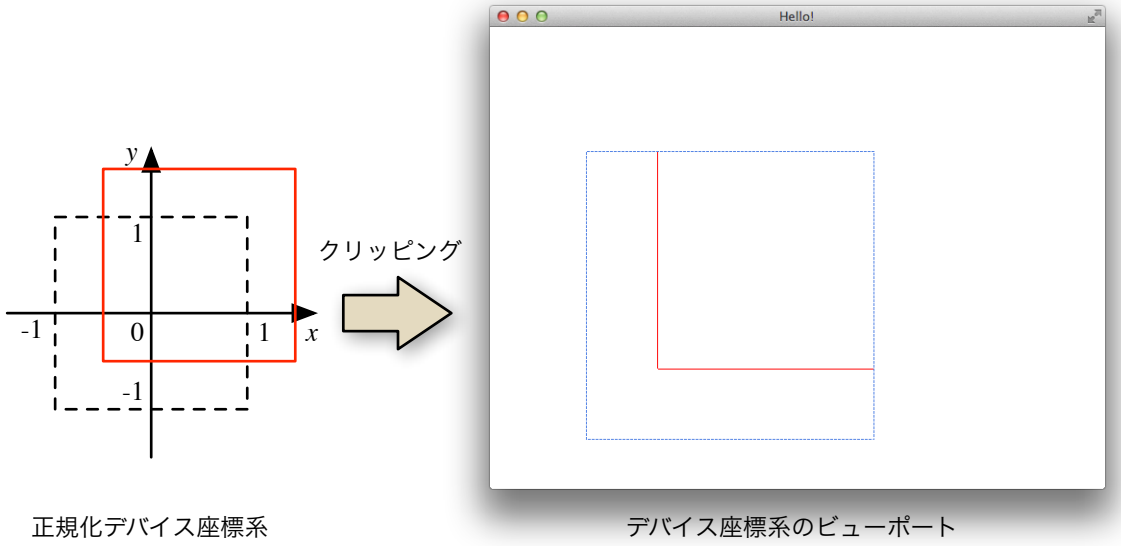

図 44 クリッピング

この結果から分かるように、図形は正規化デバイス座標系において (-1, -1) と (1, 1) を対角の頂点とする正方形の内部にある部分しか描かれません。この領域は**クリッピング空間**と呼ばれ、正規化デバイス座標系は**クリッピング座標系**とも呼ばれます。

したがって、任意の大きさの空間に配置された図形の描画を行いたい場合は、その空間のうち画面上に表示する部分をクリッピング空間にはめ込む座標変換を行う必要があります。

6.1.3 ビューポートの設定方法

glViewport() によるビューポートの設定を描画のループの前でしか行なっていなければ、その後のウィンドウのサイズの変更がビューポートに反映されません。かといって、描画のループの中で描画のたびにウィンドウのサイズを調べてビューポートを設定するのは、ウィンドウのサイズの変更がそれほど頻繁に行われる処理ではないことを考えると、無駄が多い気がします。

そこで、ウィンドウのサイズが変更された時だけ glViewport() を実行して、ビューポートの設定を行うようにします。glfwSetWindowSizeCallback() 関数を用いれば、ウィンドウのサイズが変更されたときに実行する関数 (コールバック関数) を設定することができます。

● ウィンドウ処理のクラス Window

この処理を追加するために、GLFW によるウィンドウに関する処理を次の Window というクラスにまとめます。これは Window.h というヘッダファイルに作成します。この private メンバ変数の window には、開いたウィンドウのハンドル (識別子) を保持します。

```
#pragma once

// ウィンドウ関連の処理
class Window
```

```
{
  // ウィンドウのハンドル
  GLFWwindow *const window;
```

メンバ変数 window は、コンストラクタにおいて、glfwCreateWindow() 関数の戻り値として得られるウィンドウのハンドルで初期化します。glfwMakeContextCurremt() 関数や GLEW の初期化、glfwSwapInterval() 関数もコンストラクタ内で実行します。

```
public:

  // コンストラクタ
  Window(int width = 640, int height = 480, const char *title = "Hello!")
    : window(glfwCreateWindow(width, height, title, NULL, NULL))
  {
    if (window == NULL)
    {
      // ウィンドウが作成できなかった
      std::cerr << "Can't create GLFW window." << std::endl;
      exit(1);
    }

    // 現在のウィンドウを処理対象にする
    glfwMakeContextCurrent(window);

    // GLEW を初期化する
    glewExperimental = GL_TRUE;
    if (glewInit() != GLEW_OK)
    {
      // GLEW の初期化に失敗した
      std::cerr << "Can't initialize GLEW" << std::endl;
      exit(1);
    }

    // 垂直同期のタイミングを待つ
    glfwSwapInterval(1);
  }
```

デストラクタではウィンドウを閉じる glfwDestroyWindow() 関数を実行します。そのほか、ウィンドウを閉じる判定をする shouldClose() やダブルバッファリング時にバッファの入れ替えを行う swapBuffers() も定義しておきます。

```
  // デストラクタ
  virtual ~Window()
  {
    glfwDestroyWindow(window);
  }

  // ウィンドウを閉じるべきかを判定する
  int shouldClose() const
  {
    return glfwWindowShouldClose(window);
  }

  // カラーバッファを入れ替えてイベントを取り出す
```

```
  void swapBuffers()
  {
    // カラーバッファを入れ替える
    glfwSwapBuffers(window);

    // イベントを取り出す
    glfwWaitEvents();
  }
};
```

void glfwDestroyWindow(GLFWwindow *const window)

指定されたウィンドウとそのコンテキストを破棄します。この関数が実行されると、このウィンドウに設定されているコールバック関数は呼び出されなくなります。

window

破棄するウィンドウのハンドル。

このほか、このクラスのメンバ関数として、glfwWindowShouldClose() 関数の戻り値を返す shouldClose() や、バッファを入れ替えてイベントを取り出す swapBuffers() も定義しておきます。

● ウィンドウのサイズ変更時に実行する関数の登録

ウィンドウのサイズを変更した時に実行する関数は、glfwSetWindowSizeCallback() 関数を使って次のように設定します。実行する関数名は、ここでは resize() とします。引数の window は対象のウィンドウのハンドルです。

```
  glfwSetWindowSizeCallback(window, resize);
```

GLFWwindowsizefun glfwSetWindowSizeCallback(GLFWwindow *const window,
    GLFWwindowsizefun cbfun)

指定されたウィンドウのサイズが変更されたときに実行するコールバック関数を指定します。戻り値として以前に設定されていたコールバック関数のポインタか、コールバック関数が設定されていなければ NULL を返します。

window

サイズ変更時にこのコールバック関数を呼び出すのウィンドウのハンドル。

cbfun

実行する関数のポインタ。NULL なら現在設定されているコールバック関数を削除します。

引数 cbfun に設定する関数は、次の形式で定義します。ここでは関数名を resize() とします。

```
void resize(GLFWwindow *const window, int width, int height)
{
  《省略》
}
```

window

サイズが変更されたウィンドウのハンドル。

width, height

それぞれサイズ変更後のウィンドウの幅と高さ。

● Window クラス (Window.h) の変更点

この処理を先ほど定義したクラス Window に組み込みます。resize() を静的メンバ関数としてこのクラスに追加し、コンストラクタで glfwSetWindowSizeCallback() 関数を実行して、これをコールバック関数として登録します。resize() ではウィンドウ全体をビューポートに設定します。メンバ関数をコールバック関数に使う場合は、静的メンバ関数である必要があります。

この関数で行うビューポートの設定などの処理は、一般に最初に Window を開いたときにも実行すべきものです。したがって、コンストラクタの最後でも resize() を呼び出します。

```
  // コンストラクタ
  Window(int width = 640, int height = 480, const char *title = "Hello!")
    : window(glfwCreateWindow(width, height, title, NULL, NULL))
  {
     《省略》

    // 垂直同期のタイミングを待つ
    glfwSwapInterval(1);

    // ウィンドウのサイズ変更時に呼び出す処理の登録
    glfwSetWindowSizeCallback(window, resize);

    // 開いたウィンドウの初期設定
    resize(window, width, height);
  }

  《省略》

  // カラーバッファを入れ替えてイベントを取り出す
  void swapBuffers()
  {
    // カラーバッファを入れ替える
    glfwSwapBuffers(window);

    // イベントを取り出す
    glfwWaitEvents();
  }

  // ウィンドウのサイズ変更時の処理
  static void resize(GLFWwindow *const window, int width, int height)
  {
    // ウィンドウ全体をビューポートに設定する
    glViewport(0, 0, width, height);
  }
};
```

● メインプログラム (main.cpp) の変更点

Window クラスの定義を記述したヘッダファイル Window.h を、main.cpp の冒頭で #include します。そして glfwWindowHint() による設定の後の処理を、Window クラスのインスタンス (window) の生成に置き換えます。これで図形を表示するウィンドウが作成されます。GLEW の初期化はこのインスタンスの生成時に行われますので、この部分からは削除します。また、ループの継続条件やダブルバッファリングの処理を、それぞれそのインスタンスのメンバ関数である shouldClose() と swapBuffers() に置き換えます。

```
#include <cstdlib>
#include <iostream>
#include <fstream>
#include <vector>
#include <memory>
#include <GL/glew.h>
#include <GLFW/glfw3.h>
#include "Window.h"
#include "Shape.h"

《省略》

int main()
{
  《省略》

  // OpenGL Version 3.2 Core Profile を選択する
  glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
  glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 2);
  glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT, GL_TRUE);
  glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);

  // ウィンドウを作成する
  Window window;

  // 背景色を指定する
  glClearColor(1.0f, 1.0f, 1.0f, 0.0f);

  // プログラムオブジェクトを作成する
  const GLuint program(loadProgram("point.vert", "point.frag"));

  // 図形データを作成する
  std::unique_ptr<const Shape> shape(new Shape(2, 4, rectangleVertex));

  // ウィンドウが開いている間繰り返す
  while (window.shouldClose() == GL_FALSE)
  {
    // ウィンドウを消去する
    glClear(GL_COLOR_BUFFER_BIT);

    // シェーダプログラムの使用開始
    glUseProgram(program);

    // 図形を描画する
```

```
    wireRectangle.draw();

    // カラーバッファを入れ替えてイベントを取り出す
    window.swapBuffers();
  }
}
```

■ サンプルプログラム step06

6.1.4 表示図形の縦横比を維持する

表示図形の縦横比 (**アスペクト比**) が、ウィンドウの縦横比に影響されないようにする方法を考えます。画面上のウィンドウの幅を w、高さを h とし、高さを基準にした時、ウィンドウの縦横比 $aspect$ は $aspect = w / h$ になります。

したがって、ウィンドウ上での図形の縦横比を本来の縦横比と一致させるには、正規化デバイス座標系上での図形の横幅を、本来の図形の横幅に対して $1 / aspect$ 倍します (図 45)。なお、ここでは本来の図形を定義している座標系を**ワールド座標系**といいます。

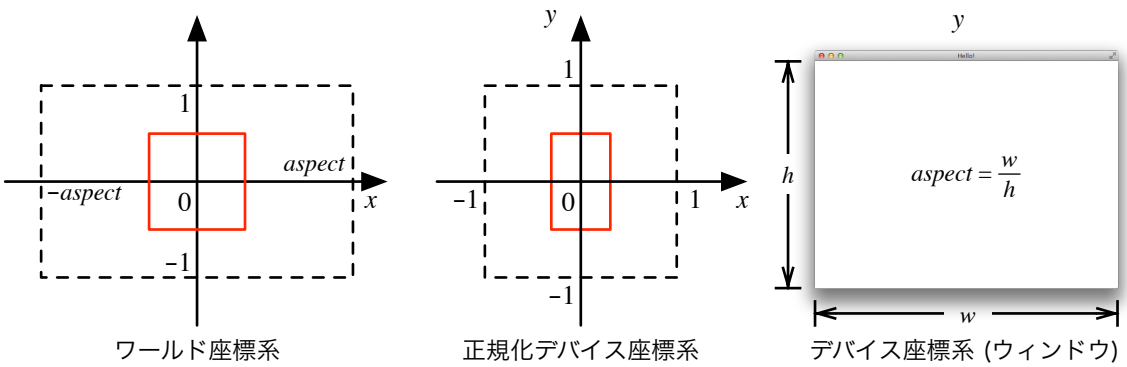

図 45 ウィンドウの縦横比 *aspect* をワールド座標に反映

● Window クラス (Window.h) の変更点

Window クラスに float 型のメンバ変数 aspect を追加します。aspect は、ウィンドウのサイズが変更されたときに、それに合わせて更新する必要があります。このクラスでは、ウィンドウのサイズが変更されたときにメンバ関数 resize() を実行するようにしていますから、そこで更新することができます。resize() はコンストラクタの最後でも呼び出していますから、これによりウィンドウを開いたときに aspect に値が設定されます。

ところが、resize() はコールバック関数として使うために静的メンバ関数にしています。静的メンバ関数はインスタンスを生成していない (すなわち、メンバ変数にメモリが割り当てられていない) 時にも呼び出せるかわりに、その中でメンバ変数を参照することができません。静的メンバ関数内でメンバ変数を参照するためには、そのインスタンスの実体 (割り当てられたメモリ) の場所を知る必要があります。

そのために、コンストラクタで glfwSetWindowUserPointer() 関数により this ポインタ、すなわ

ち生成されたインスタンス自体を指すポインタを、GLFW 側で記録しておきます。そして、コールバック関数として起動された静的メンバ関数において、その引数に渡されるウィンドウのハンドルをもとに、glfwGetWindowUserPointer() 関数を使って記録した this ポインタを取り出します。インスタンスのメンバ変数は、このポインタを使ってアクセスすることができます。

このほか、この変数の値をクラス外から参照できるように、getAspect() のようなメンバ関数 (アクセサ) を用意しておきます。

```
// ウィンドウ関連の処理
class Window
{
  // ウィンドウのハンドル
  GLFWwindow *const window;

  // 縦横比
  GLfloat aspect;

public:

  // コンストラクタ
  Window(int width = 640, int height = 480, const char *title = "Hello!")
    : window(glfwCreateWindow(width, height, title, NULL, NULL))
  {
  《省略》

    // 垂直同期のタイミングを待つ
    glfwSwapInterval(1);

    // このインスタンスの this ポインタを記録しておく
    glfwSetWindowUserPointer(window, this);

    // ウィンドウのサイズ変更時に呼び出す処理の登録
    glfwSetWindowSizeCallback(window, resize);

    // 開いたウィンドウの初期設定
    resize(window, width, height);
  }

  《省略》

  // 縦横比を取り出す
  GLfloat getAspect() const { return aspect; }

  // ウィンドウのサイズ変更時の処理
  static void resize(GLFWwindow *const window, int width, int height)
  {
    // ウィンドウ全体をビューポートに設定する
    glViewport(0, 0, width, height);

    // このインスタンスの this ポインタを得る
    Window *const
      instance(static_cast<Window *>(glfwGetWindowUserPointer(window)));

    if (instance != NULL)
    {
```

```
      // このインスタンスが保持する縦横比を更新する
      instance->aspect =
        static_cast<GLfloat>(width) / static_cast<GLfloat>(height);
    }
  }
};
```

void glfwSetWindowUserPointer(GLFWwindow *const window, void *pointer)

window に指定したウィンドウに対して pointer に指定したユーザ定義の任意のポインタを記録します。ウィンドウが破棄されるまで、この値が保持されます。初期値は NULL です。

window

ポインタを記録する対象のウィンドウのハンドル。

pointer

記録するポインタ。

void glfwGetWindowUserPointer(GLFWwindow *const window)

window に指定したウィンドウに対して記録されているユーザ定義のポインタを取り出します。初期値は NULL です。

window

記録されたポインタを取り出す対象のウィンドウのハンドル。

● バーテックスシェーダのソースプログラム point.vert の変更点

このメンバ変数 aspect の値をシェーダプログラムに渡して、バーテックスシェーダで処理される頂点の位置を変更します。aspect の値は一つの図形の描画中に変更されることはないので、これにはシェーダの uniform 変数を用います。in 変数 (4.2.2) が一つの頂点ごとに異なる情報を保持しているのに対し、uniform 変数は一回の描画命令で使用される全ての頂点から共通して参照される値を保持します。

このプログラムでは、描画する図形の頂点位置がバーテックスシェーダの in 変数 position に格納されています。vec4 は四つの float 型の要素を持つベクトルのデータ型で、position は position.x、position.y、position.z、および position.w の要素を持っています。ただし、このプログラムは図 40 の二次元図形を描画するため、CPU 側のプログラムでは頂点の x 座標値と y 座標値だけを設定しています。これらはそれぞれ position.x と position.y に格納されます。残りの position.z には 0、p.w には 1 がデフォルト値として格納されています。

この頂点位置 position に vec4 などのベクトル型同士のかけ算は、対応する要素同士の積を要素とする同じ型のベクトルになります。たとえば、a と b が vec4 型の変数のとき、a * b は vec4(a.x * b.x, a.y * b.y, a.z * b.z, a.w * b.w) になります。したがって、この X 座標値だけを 1 / aspect 倍するには、vec4(1.0 / aspect, 1.0, 1.0, 1.0) という vec4 型の値を乗じます。これは position との乗算を省略して、gl_Position = vec4(position.x / aspect, position.yzw); という書き方もできます。

```
#version 150 core
uniform float aspect;
in vec4 position;
void main()
{
  gl_Position = position * vec4(1.0 / aspect, 1.0, 1.0, 1.0);
}
```

● uniform 変数 aspect の設定

uniform 変数には、描画を行う前に CPU 側のプログラムで値を設定します。そのために、プログラムオブジェクトにおける uniform 変数の場所 (index) を、CPU 側のプログラムで調べておきます。これは glGetUniformLocation() 関数で行います。そして glUseProgram() でシェーダプログラムを有効にした後で、その index に対して glUniform*() 関数を使って値を設定します。float 型の単一の変数を設定するなら glUniform1f() 関数を用います。

```
int main()
{
  《省略》

  // 背景色を指定する
  glClearColor(1.0f, 1.0f, 1.0f, 0.0f);

  // プログラムオブジェクトを作成する
  const GLuint program(loadProgram("point.vert", "point.frag"));

  // uniform 変数の場所を取得する
  cosnt GLint aspectLoc(glGetUniformLocation(program, "aspect"));

  // 図形データを作成する
  std::unique_ptr<const Shape> shape(new Shape(2, 4, rectangleVertex));

  // ウィンドウが開いている間繰り返す
  while (window.shouldClose() == GL_FALSE)
  {
    // ウィンドウを消去する
    glClear(GL_COLOR_BUFFER_BIT);

    // シェーダプログラムの使用開始
    glUseProgram(program);

    // uniform 変数に値を設定する
    glUniform1f(aspectLoc, window.getAspect());

    // 図形を描画する
    wireRectangle.draw();

    // カラーバッファを入れ替えてイベントを取り出す
    window.swapBuffers();
  }
}
```

■ サンプルプログラム step07

GLint glGetUniformLocation(GLuint program, const GLchar *name)

program に指定したプログラムオブジェクトの中で使われている name に指定した uniform 変数の場所を探します。戻り値は uniform 変数の index で、見つからなければ -1 を返します。

program

uniform 変数を探すプログラムオブジェクト名。

name

探す uniform 変数名の文字列。

void glUniform1f(GLint location, GLfloat v0)

現在使用中のシェーダプログラムの location に指定した index の float 型の uniform 変数に GLfloat 型の値 v0 を設定します。ほかに vec2 型に設定する glUniform2f()、vec3 型に設定する glUniform3f()、vec4 型に設定する glUniform4f() があります。

location

値を設定する float 型の uniform 変数の index。

v0

設定する GLfloat 型の値。

補足: in 変数と uniform 変数

in 変数がシェーダプログラムの前段 (バーテックスシェーダの場合は頂点バッファオブジェクト) からのデータの受け取りに使われ、頂点ごとに異なる値が設定されるのに対し、uniform 変数は 1 回の描画命令ごとに設定され、すべての頂点で同じ値に設定されます (図 46)。

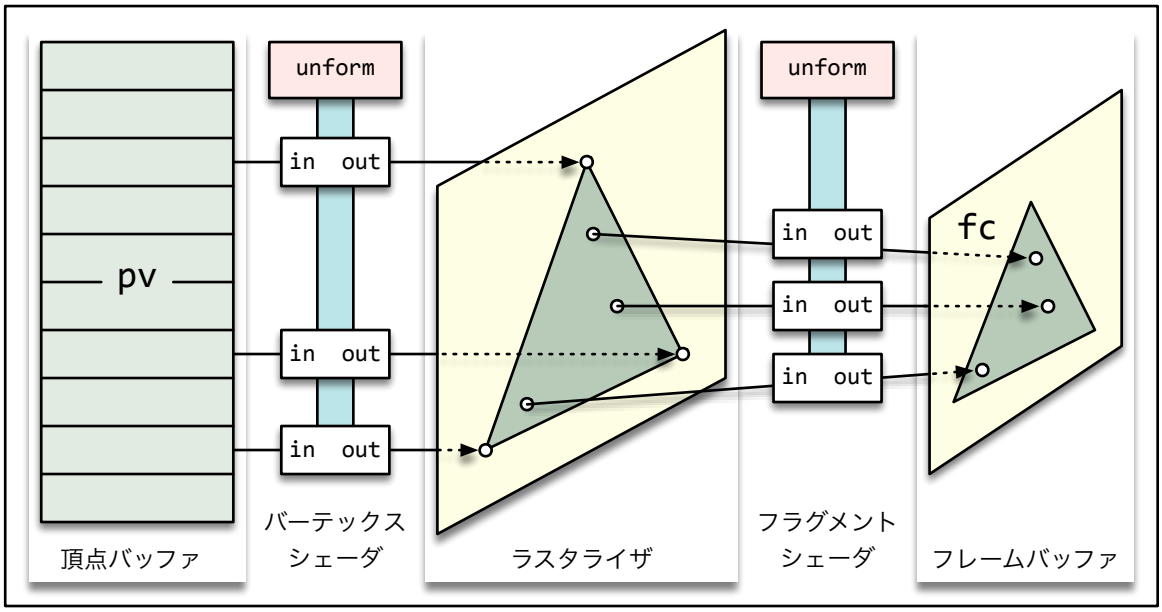

図 46 in 変数と uniform 変数

6.1.5　表示図形のサイズを固定する

前節では、ワールド座標系の高さをクリッピング空間の高さと一致させ、幅方向だけを縦横比によりスケーリングしていました。ここでは表示図形のサイズがウィンドウのサイズに影響されず、常に同じ大きさで表示されるようにする方法を考えます。

そのために、表示しようとする図形の画面上での大きさを決めておきます。図形を定義しているワールド座標系上において長さ 1 の線分が、画面上に長さ s (画素) で表示されるとします (図 47)。これはワールド座標系に対するデバイス座標系の拡大率です。

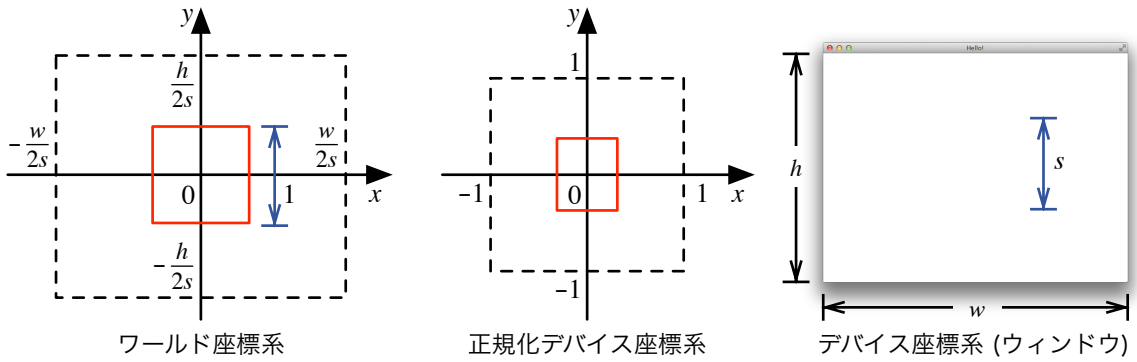

図 47 デバイス座標系のワールド座標系に対する拡大率 s

● Window クラス (Window.h) の変更点

図形を表示するウィンドウの画面上の幅を w、高さを h としたとき、このウィンドウにぴったり収まるワールド座標系の領域は $\pm w/2s$、$\pm h/2s$ の範囲になります (図 47)。これを正規化デバイス座標系の ± 1 の領域にスケーリングするので、ワールド座標系に対する正規化デバイス座標系の拡大率は、これらの逆数になります。描画するウィンドウの幅 w と高さ h を保持する二つの要素の配列のメンバ変数 size と、ワールド座標系に対するデバイス座標系の拡大率 s を保持するメンバ変数 scale を、Window クラスに追加します。

```
// ウィンドウ関連の処理
class Window
{
  // ウィンドウのハンドル
  GLFWwindow *const window;

  // ウィンドウのサイズ
  GLfloat size[2];

  // ワールド座標系に対するデバイス座標系の拡大率
  GLfloat scale;

public:
```

scale には、ここではワールド座標系上において 1 の長さを画面上では 100 (画素) の長さで

表示するものとして、100 で初期化しておきます。scale はウィンドウのサイズが変更されたときに更新する必要があるので、resize() で設定します。resize() は Window クラスのコンストラクタでも呼び出しているので、scale の初期値はそこで設定されます。

このほか、scale のポインタを取り出すメンバ関数 getScale() を追加しておきます。

```
  // コンストラクタ
  Window(int width = 640, int height = 480, const char *title = "Hello!")
    : window(glfwCreateWindow(width, height, title, NULL, NULL))
    , scale(100.0f)
  {
    《省略》

    // 垂直同期のタイミングを待つ
    glfwSwapInterval(1);

    // このインスタンスの this ポインタを記録しておく
    glfwSetWindowUserPointer(window, this);

    // ウィンドウのサイズ変更時に呼び出す処理の登録
    glfwSetWindowSizeCallback(window, resize);

    // 開いたウィンドウの初期設定
    resize(window, width, height);
  }

  《省略》

  // ウィンドウのサイズを取り出す
  const GLfloat *getSize() const { return size; }

  // ワールド座標系に対するデバイス座標系の拡大率を取り出す
  GLfloat getScale() const { return scale; }

  // ウィンドウのサイズ変更時の処理
  static void resize(GLFWwindow *const window, int width, int height)
  {
    // ウィンドウ全体をビューポートに設定する
    glViewport(0, 0, width, height);

    // このインスタンスの this ポインタを得る
    Window *const
      instance(static_cast<Window *>(glfwGetWindowUserPointer(window)));

    if (instance != NULL)
    {
      // 開いたウィンドウのサイズを保存する
      instance->size[0] = static_cast<GLfloat>(width);
      instance->size[1] = static_cast<GLfloat>(height);
    }
  }
};
```

● バーテックスシェーダのソースプログラム point.vert の変更点

このプログラムの場合は、x 方向と y 方向の二つの方向について、別々に拡大率を設定します。二つの要素を保持する vec2 型の uniform 変数 size と float 型の uniform 変数 scale を用意し、先ほどの aspect と置き換えます。これらを vec4(2.0 * scale / size, 1.0, 1.0) として、ワールド座標系に対する正規化デバイス座標系の拡大率の vec4 型のベクトルを作ります。

```
#version 150 core
uniform vec2 size;
uniform float scale;
 in vec4 position;
void main()
{
  gl_Position = position * vec4(2.0 * scale / size, 1.0, 1.0);
}
```

● uniform 変数 size と scale の設定

uniform 変数 size と scale に、それぞれ Window クラスのメンバ変数 size と scale の値を設定します。まず、これらの uniform 変数の index を glGetUniformLocation() により求めます。そして glUseProgram() でシェーダプログラムを有効にした後に、それぞれの index を用いて、vec2 型の size は glUniform2fv() 関数を用いて、float 型の scale は glUniform1f() を用いて指定します。メンバ変数の size はポインタなので、値をポインタで指定する glUniform2fv() を使います。

```
int main()
{
  《省略》

  // 背景色を指定する
  glClearColor(1.0f, 1.0f, 1.0f, 0.0f);

  // プログラムオブジェクトを作成する
  const GLuint program(loadProgram("point.vert", "point.frag"));

  // uniform 変数の場所を取得する
  const GLint sizeLoc(glGetUniformLocation(program, "size"));
  const GLint scaleLoc(glGetUniformLocation(program, "scale"));

  // 図形データを作成する
  std::unique_ptr<const Shape> shape(new Shape(2, 4, rectangleVertex));

  // ウィンドウが開いている間繰り返す
  while (window.shouldClose() == GL_FALSE)
  {
    // ウィンドウを消去する
    glClear(GL_COLOR_BUFFER_BIT);

    // シェーダプログラムの使用開始
    glUseProgram(program);

    // uniform 変数に値を設定する
```

```
    glUniform2fv(sizeLoc, 1, window.getSize());
    glUniform1f(scaleLoc, window.getScale());

    // 図形を描画する
    wireRectangle.draw();

    // カラーバッファを入れ替えてイベントを取り出す
    window.swapBuffers();
  }
}
```

void glUniform2fv(GLint location, GLsizei count, const GLfloat *value)

現在使用中のシェーダプログラムの location に指定した index の vec2 型の uniform 変数に value に指定した GLfloat 型の 2 要素以上の配列のポインタを設定します。ほかに vec2 型に設定する glUniform2fv()、vec3 型に設定する glUniform3fv()、vec4 型に設定する glUniform4fv() があります。

location

値を設定する vec2 型の uniform 変数の index。

count

設定する uniform 変数が配列のとき、その要素数。配列でなければ 1。

value

設定する値を格納した GLfloat 型の配列。配列の要素数は count×2 以上必要。

■ サンプルプログラム step08

● 実行結果

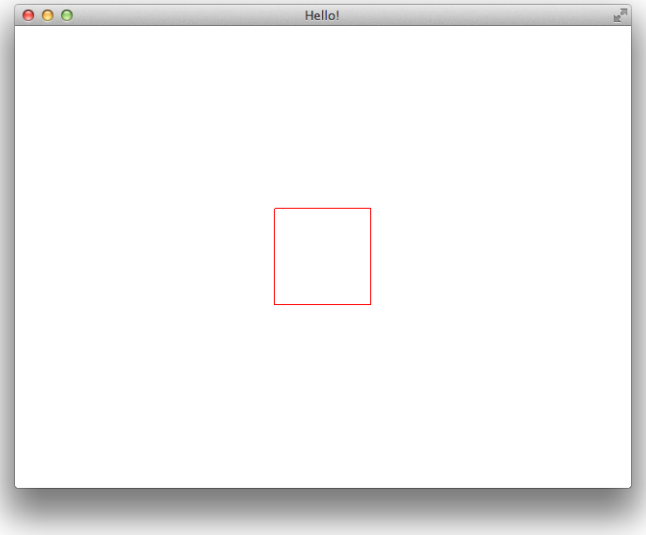

図 48 ウィンドウのサイズ変更に対して表示図形のサイズを固定

6.2 マウスで図形を動かす

6.2.1 マウスカーソルの位置の取得

表示している図形がマウスに追従して動くようにします。そのためには、ウィンドウ上のマウスカーソルの位置を調べる必要があります。これには、glfwSetCursorPosCallback() によってマウスカーソルが動いたときに呼び出すコールバック関数を設定する方法 (コールバック方式) と、glfwWaitEvents() や glfwPollEvents() でイベントを取り出した後に glfwGetCursorPos() 関数によって得る方法 (ポーリング方式) の二通りの方法があります。ここでは後者のポーリング方式による方法について説明します。

● Window クラス (Window.h) の変更点

Window クラスに図形の位置を保持する二つの要素の配列のメンバ変数 location を追加します。この値をバーテックスシェーダの uniform 変数 location に設定します。

```
// ウィンドウ関連の処理
class Window
{
  《省略》

  // ワールド座標系に対するデバイス座標系の拡大率
  GLfloat scale;

  // 図形の正規化デバイス座標系上での位置
  GLfloat location[2];

public:
```

location にはコンストラクタで初期値として 0 を設定しておきます。

```
  // コンストラクタ
  Window(int width = 640, int height = 480, const char *title = "Hello!")
    : window(glfwCreateWindow(width, height, title, NULL, NULL))
    , scale(100.0f)
  {
    《省略》

    // 開いたウィンドウの初期設定
    resize(window, width, height);

    // 図形の正規化デバイス座標系上での位置の初期値
    location[0] = location[1] = 0.0f;
  }
```

マウスの移動などで発生したイベントはメンバ関数 swapBuffers() で glfwWaitEvents() 関数を呼び出して取り出していますから、その後に glfwGetCursorPos() 関数を使ってマウスカーソルの位置を調べます。なお、glfwWaitEvents() はイベントが発生するまでプログラムを停止させます。

取り出したマウスカーソルの位置 (x, y) の、正規化デバイス座標系における位置を求めます。マウスカーソルの位置はデバイス座標系における座標値ですから、x と y のそれぞれを画面上のウィンドウの幅 *w* と高さ *h* で割り、それを 2 倍して 1 引く ([0, 1] の範囲を [-1, 1] の範囲に変換する) ことで、その正規化デバイス座標系上の位置が得られます (図 47)。

ただし、マウスカーソルの座標系の原点はウィンドウの左上にあって、正規化デバイス座標系とは上下が反転しているため、この y 座標については符号を反転します。これらをメンバ変数 location に代入します。

```
  // カラーバッファを入れ替えてイベントを取り出す
  void swapBuffers()
  {
    // カラーバッファを入れ替える
    glfwSwapBuffers(window);

    // イベントを取り出す
    glfwWaitEvents();

    // マウスカーソルの位置を取得する
    double x, y;
    glfwGetCursorPos(window, &x, &y);

    // マウスカーソルの正規化デバイス座標系上での位置を求める
    location[0] = static_cast<GLfloat>(x) * 2.0f / size[0] - 1.0f;
    location[1] = 1.0f - static_cast<GLfloat>(y) * 2.0f / size[1];
  }
```

このほか、location のポインタを取り出すメンバ関数 getLocation() を追加しておきます。

```
  // ワールド座標系に対するデバイス座標系の拡大率を取り出す
  GLfloat getScale() const { return scale; }

  // 位置を取り出す
  const GLfloat *getLocation() const { return location; }

  // ウィンドウのサイズ変更時の処理
  static void resize(GLFWwindow *window, int width, int height)
  {
    《省略》
  }
};
```

void glfwGetCursorPos(GLFWwindow *window, const double *xpos, const double *ypos)

window で指定したウィンドウの左上を原点としたマウスカーソルの位置を *xpos と *ypos に得ます。なお、位置の整数値を得たい時は floor() 関数を使用してください。直接整数型にキャストすると、負の値のときに正しい値を得られません。

window

マウスカーソルの位置を取得するウィンドウのハンドル。

x, y

ウィンドウの左上を原点としたマウスカーソルの位置。

● バーテックスシェーダのソースプログラム point.vert の変更点

バーテックスシェーダのソースプログラム point.vert に、uniform 変数 location の宣言を追加します。この location を vec4 に変換して gl_Position に加えます。

```
#version 150 core
uniform vec2 size;
uniform float scale;
uniform vec2 location;
in vec4 position;
void main()
{
  gl_Position = position * vec4(2.0 * scale / size, 1.0, 1.0)
    + vec4(location, 0.0, 0.0);
}
```

● uniform 変数 location の設定

uniform 変数 location に Window クラスのメンバ変数 location の値を設定します。scale の場合と同様に glGetUniformLocation() を使って uniform 変数 location の index を得ます。そして glUseProgram() でシェーダプログラムを有効にしたあ glUniform2fv() 関数を使って、その index にメンバ変数 location の値を設定します。これも要素数が 2 の配列なので、glUniform2fv() ではそのポインタを指定します。

```
int main()
{
  《省略》

  // プログラムオブジェクトを作成する
  const GLuint program(loadProgram("point.vert", "point.frag"));

  // uniform 変数の場所を取得する
  const GLint sizeLoc(glGetUniformLocation(program, "size"));
  const GLint scaleLoc(glGetUniformLocation(program, "scale"));
  const GLint locationLoc(glGetUniformLocation(program, "location"));

  // 図形データを作成する
  std::unique_ptr<const Shape> shape(new Shape(2, 4, rectangleVertex));

  // ウィンドウが開いている間繰り返す
  while (window.shouldClose() == GL_FALSE)
  {
    // ウィンドウを消去する
    glClear(GL_COLOR_BUFFER_BIT);

    // シェーダプログラムの使用開始
    glUseProgram(program);
```

```
    // uniform 変数に値を設定する
    glUniform2fv(sizeLoc, 1, window.getSize());
    glUniform1f(scaleLoc, window.getScale());
    glUniform2fv(locationLoc, 1, window.getLocation());

    // 図形を描画する
    wireRectangle.draw();

    // カラーバッファを入れ替えてイベントを取り出す
    window.swapBuffers();
  }
}
```

■ サンプルプログラム step09

6.2.2 マウスボタンの操作の取得

前節のプログラムでは、マウスのボタンを押さなくてもマウスを動かすだけで図形が動いていました。ここでは、マウスのボタンを押している間だけ、図形が動くようにこれを変更します。

マウスのボタンの操作も glfwSetMouseButtonCallback() を使って設定したコールバック関数により調べることができますが、ここでは glfwWaitEvents() や glfwPollEvents() でイベントを取り出した後に glfwGetMouseButton() 関数によって得る方法を説明します。

● Window クラス (Window.h) の変更点

glfwGetMouseButton() 関数は引数に指定したボタンが押されている時は GLFW_PRESS (1)、押されていない時は GLFW_RELEASE (0) を返すので、GLFW_RELEASE でないときにマウスカーソルの位置を取得し、図形の位置 location を更新します。

```
// ウィンドウ関連の処理
class Window
{
  《省略》

  // カラーバッファを入れ替えてイベントを取り出す
  void swapBuffers()
  {
    // カラーバッファを入れ替える
    glfwSwapBuffers(window);

    // イベントを取り出す
    glfwWaitEvents();

    // マウスの左ボタンの状態を調べる
    if (glfwGetMouseButton(window, GLFW_MOUSE_BUTTON_1) != GLFW_RELEASE)
    {
      // マウスの左ボタンが押されていたらマウスカーソルの位置を取得する
      double x, y;
      glfwGetCursorPos(window, &x, &y);

      // マウスカーソルの正規化デバイス座標系上での位置を求める
```

```
      location[0] = static_cast<GLfloat>(x) * 2.0f / size[0] - 1.0f;
      location[1] = 1.0f - static_cast<GLfloat>(y) * 2.0f / size[1];
    }
  }

  《省略》
};
```

void glfwGetMouseButton(GLFWwindow *window, int button)

window で指定したウィンドウにおいて button に指定したマウスのボタンが押されていれば GLFW_PRESS (1)、押されていないときは GLFW_RELEASE (0) を返します。

window

マウスのボタンの状態を調べる対象のウィンドウのハンドル。

button

状態を調べるマウスのボタン。

GLFW_MOUSE_BUTTON_1 と GLFW_MOUSE_BUTTON_LEFT は左ボタン、

GLFW_MOUSE_BUTTON_2 と GLFW_MOUSE_BUTTON_RIGHT は右ボタン、

GLFW_MOUSE_BUTTON_3 と GLFW_MOUSE_BUTTON_MIDDLE は中ボタン、

GLFW_MOUSE_BUTTON_4 ～ GLFW_MOUSE_BUTTON_8 はマウスに依存、

GLFW_MOUSE_BUTTON_LAST は GLFW_MOUSE_BUTTON_8 と同じ。

■ サンプルプログラム step10

6.2.3 マウスホイールの操作の取得

マウスホイールによって図形を拡大縮小できるようにします。マウスホイールの操作を取得するには、glfwSetScrollCallback() 関数を使ってマウスホイールを操作したときに呼び出すコールバック関数を設定します。なお、GLFW のバージョン 3 にはポーリングによってマウスホイールの操作を取得する方法は用意されていません。

● Window クラス (Window.h) の変更点

図形の拡大縮小は、ワールド座標系に対するデバイス座標系の拡大率 s を変化させることで実現します。s を変化させた時は scale も更新する必要がありますから、現在開いているウィンドウの大きさを保持するメンバ変数 size と、size と s から scale を求めるプライベートメソッド updateScale() を追加します。この変更に伴い、resize() では引数 width と height を size に保存するようにし、scale を更新している部分を updateScale() の呼び出に置き換えます。

また、コンストラクタで glfwSetScrollCallback() 関数を使ってマウスホイールを操作したときに呼び出すメンバ関数 wheel() を登録します。

```
// ウィンドウ関連の処理
class Window
{
  《省略》

public:

  // コンストラクタ
  Window(int width = 640, int height = 480, const char *title = "Hello!")
    : window(glfwCreateWindow(width, height, title, NULL, NULL))
    , scale(100.0f)
  {
    《省略》

    // ウィンドウのサイズ変更時に呼び出す処理の登録
    glfwSetWindowSizeCallback(window, resize);

    // マウスホイール操作時に呼び出す処理の登録
    glfwSetScrollCallback(window, wheel);

    // このインスタンスの this ポインタを記録しておく
    glfwSetWindowUserPointer(window, this);

    // 開いたウィンドウの初期設定
    resize(window, width, height);

    // 図形の正規化デバイス座標系上での位置の初期値
    location[0] = location[1] = 0.0f;
  }
```

GLFWscrollfun glfwSetScrollCallback(GLFWwindow *const window, GLFWscrollfun cbfun)

指定されたウィンドウに対してマウスホイールが操作されたときに実行するコールバック関数を指定します。戻り値として以前に設定されていたコールバック関数のポインタか、コールバック関数が設定されていなければ NULL を返します。

window

対象のウィンドウのハンドル。

cbfun

実行する関数のポインタ。NULL なら現在設定されているコールバック関数を削除します。

引数 cbfun に設定する関数は、次の形式で定義します。ここでは関数名を wheel() とします。

```
void wheel(GLFWwindow *const window, double x, double y)
{
  《省略》
}
```

window

サイズが変更されたウィンドウのハンドル。

x, y

以前のマウスホイールの操作によるイベントの発生時点からのマウスホイールの操作量。

この関数 wheel() は resize() 同様コールバック関数として使いますから、静的メンバ関数にします。したがって、これも glfwGetWindowUserPointer() を使ってインスタンスの this ポインタを取り出して、インスタンスのメンバ変数にアクセスします。

この引数の x と y は以前のマウスホイールの操作からの操作量ですから、絶対量が必要なら、これを累積します。ここではワールド座標系に対するデバイス座標系に対する拡大率を保持しているメンバ変数 s に y を積算し、そのあとメンバ関数 updateScale() を呼び出してメンバ変数 scale を更新します。

なお、この x は Apple の Magic Mouse のように二次元のスクロールが可能なマウスでは変化しますが、ホイールの回転軸が固定されている場合は常に 0 になっています。

```
  // ウィンドウのサイズ変更時の処理
  static void resize(GLFWwindow *window, int width, int height)
  {
    《省略》

    if (instance != NULL)
    {
      // 開いたウィンドウのサイズを保存しておく
      instance->size[0] = width;
      instance->size[1] = height;

      // ワールド座標系に対する正規化デバイス座標系の拡大率を更新する
      instance->updateScale();
    }
  }

  // マウスホイール操作時の処理
  static void wheel(GLFWwindow *window, double x, double y)
  {
    // このインスタンスの this ポインタを得る
    Window *const
      instance(static_cast<Window *>(glfwGetWindowUserPointer(window)));

    if (instance != NULL)
    {
      // ワールド座標系に対するデバイス座標系の拡大率を更新する
      instance->s += static_cast<GLfloat>(y);
    }
  }
};
```

■ サンプルプログラム step11

6.3 キーボードで図形を動かす

6.3.1 ESC キーでプログラムを終了する

キーボード操作で図形を移動させる前に、ESC キーをタイプしたらプログラムが終了するようにしてみましょう。特定のキーが押されているかどうかを調べるには、glfwGetKey() 関数を用います。この関数は引数に指定したキーが押されていれば GLFW_PRESS (1)、押されていなければ GLFW_RELEASE (0) を返します。

● Window クラス (Window.h) の変更点

Window クラスでウィンドウを閉じるべきかどうかを調べているメンバ関数 shoudClose() を、以下のように変更します。glfwWindowShouldClose(window) はウィンドウを閉じるべきときには非 0 の値を返し、glfwGetKey(window, GLFW_KEY_ESCAPE) も ESC キーが押されたときに 1 を返すので、論理演算の論理和 “||” で結果を求めています。これはビット演算の論理和 “|” でも結果は同じでが、論理演算であれば左の項が真 (非 0) なら右の項は実行しないので、場合によっては無駄な処理を省くことができます。でも多分、ここではほとんど差はないと思います。

```
// ウィンドウ関連の処理
class Window
{
  《省略》

  // ウィンドウを閉じるべきかを判定する
  int shouldClose() const
  {
    return glfwWindowShouldClose(window) || glfwGetKey(window, GLFW_KEY_ESCAPE);
  }

  《省略》
};
```

int glfwGetKehy(GLFWwindow *window, int key)

window で指定したウィンドウにおいて key に指定したキーボードのキーが押されていれば GLFW_PRESS (1)、押されていないときは GLFW_RELEASE (0) を返します。

window

キーの状態を調べる対象のウィンドウのハンドル。

key

状態を調べるキー。

GLFW_KEY_SPACE (空白)、GLFW_KEY_APOSTROPHE (‘)、GLFW_KEY_COMMA (,)、GLFW_KEY_MINUS (-)、GLFW_KEY_PERIOD (.)、GLFW_KEY_SLASH (/)、GLFW_KEY_0 ～ GLFW_KEY_9、GLFW_KEY_SEMICOLON (;)、GLFW_KEY_EQUAL (=)、

GLFW_KEY_A ～ GLFW_KEY_Z、GLFW_KEY_LEFT_BRACKET “[”
GLFW_KEY_BACKSLASH “¥”、GLFW_KEY_RIGHT_BRACKET “]”、
GLFW_KEY_GRAVE_ACCENT (`)、GLFW_KEY_ESCAPE (Esc)、GLFW_KEY_ENTER、
GLFW_KEY_TAB、GLFW_KEY_BACKSPACE (Bs)、GLFW_KEY_INSERT、
GLFW_KEY_DELETE (Del)、GLFW_KEY_RIGHT (→)、GLFW_KEY_LEFT (←)、
GLFW_KEY_DOWN (↓)、GLFW_KEY_UP (↑)、GLFW_KEY_PAGE_UP、
GLFW_KEY_PAGE_DOWN、GLFW_KEY_HOME、GLFW_KEY_END、
GLFW_KEY_CAPS_LOCK、GLFW_KEY_SCROLL_LOCK、GLFW_KEY_NUM_LOCK
GLFW_KEY_PRINT_SCREEN、GLFW_KEY_PAUSE、GLFW_KEY_F1 ～ GLFW_KEY_F25
GLFW_KEY_KP_0 ～ GLFW_KEY_KP_9 (テンキー 数字)、
GLFW_KEY_KP_DECIMAL (テンキー .)
GLFW_KEY_KP_DIVIDE (テンキー /)、GLFW_KEY_KP_MULTIPLY (テンキー *)、
GLFW_KEY_KP_SUBTRACT (テンキー -)、GLFW_KEY_KP_ADD (テンキー +)、
GLFW_KEY_KP_ENTER (テンキー Enter)、GLFW_KEY_KP_EQUAL (テンキー =)、
GLFW_KEY_LEFT_SHIFT、GLFW_KEY_LEFT_CONTROL、GLFW_KEY_LEFT_ALT
GLFW_KEY_LEFT_SUPER、GLFW_KEY_RIGHT_SHIFT、GLFW_KEY_RIGHT_CONTROL
GLFW_KEY_RIGHT_ALT、GLFW_KEY_RIGHT_SUPER、GLFW_KEY_MENU、
GLFW_KEY_LAST は GLFW_KEY_MENU と同じ。

6.3.2 矢印キーで図形を移動する

この glfwGetKey() を使って、キーボードの矢印キーで図形を動かすようにします。

● Window クラス (Window.h) の変更点

矢印キーは GLFW_KEY_RIGHT、GLFW_KEY_LEFT、GLFW_KEY_DOWN、GLFW_KEY_UP の四つです。Window クラスのメンバ関数 swapBuffers() の中にある glfwWaitEvents() でイベントを取り出した後、そのイベントにおけるキーボードのキーの状態を調べます。

```
// ウィンドウ関連の処理
class Window
{
  《省略》

  // カラーバッファを入れ替えてイベントを取り出す
  void swapBuffers()
  {
    // カラーバッファを入れ替える
    glfwSwapBuffers(window);

    // イベントを取り出す
    glfwWaitEvents();
```

```
    // キーボードの状態を調べる
    if (glfwGetKey(window, GLFW_KEY_LEFT) != GLFW_RELEASE)
      location[0] -= scale[0] / s;
    else if (glfwGetKey(window, GLFW_KEY_RIGHT) != GLFW_RELEASE)
      location[0] += scale[0] / s;
    if (glfwGetKey(window, GLFW_KEY_DOWN) != GLFW_RELEASE)
      location[1] -= scale[1] / s;
    else if (glfwGetKey(window, GLFW_KEY_UP) != GLFW_RELEASE)
      location[1] += scale[1] / s;

    // マウスの左ボタンの状態を調べる
    if (glfwGetMouseButton(window, GLFW_MOUSE_BUTTON_1) != GLFW_RELEASE)
    {
      《省略》
    }
  }

  《省略》
};
```

scale はワールド座標系に対する正規化デバイス座標系の拡大率ですから、このそれぞれの要素をワールド座標系に対するデバイス座標系の拡大率 s で割れば、デバイス座標系 (画面) 上の 1 画素分の長さの正規化デバイス座標系上の長さが得られます。したがって、scale[0] / s および scale[1] / s を正規化デバイス座標系上の位置 location のそれぞれの要素に足せば、図形を 1 画素分移動することができます。

● スムーズに動かす

ところが、この方法では図形はスムーズに動いてくれません。矢印キーを押した瞬間に図形は 1 画素分移動し、離した瞬間にまた 1 画素分移動します。矢印キーを押し続けていればキーリピート機能が働いて図形は連続的に動き出しますが、それでも滑らかではありません。

これはキーボードのキーのイベントが、キーを押した瞬間と離した瞬間しか発生しないからです。glfwWaitEvents() はイベントが発生した時にプログラムの実行を再開しますが、キーを押し続けている状態ではキーリピート機能が働くまでイベントは発生しませんから、それまでプログラムが停止したままになります。

そういうことなら、glfwWaitEvents() の代わりに、イベントの発生を待たない glfwPollEvents() を用いればいいことになります。実際、glfwWaitEvents() を glfwPollEvents() に置き換えれば、図形は矢印キーを押した瞬間からスムーズに動き出すようになります。

```
// ウィンドウ関連の処理
class Window
{
  《省略》

  // カラーバッファを入れ替えてイベントを取り出す
  void swapBuffers()
```

```
  {
    // カラーバッファを入れ替える
    glfwSwapBuffers(window);

    // イベントを取り出す
    glfwPollEvents();

    // キーボードの状態を調べる
    if (glfwGetKey(window, GLFW_KEY_LEFT) != GLFW_RELEASE)
      location[0] -= scale[0] / s;
    else if (glfwGetKey(window, GLFW_KEY_RIGHT) != GLFW_RELEASE)
      location[0] += scale[0] / s;
    if (glfwGetKey(window, GLFW_KEY_DOWN) != GLFW_RELEASE)
      location[1] -= scale[1] / s;
    else if (glfwGetKey(window, GLFW_KEY_UP) != GLFW_RELEASE)
      location[1] += scale[1] / s;

    // マウスの左ボタンの状態を調べる
    if (glfwGetMouseButton(window, GLFW_MOUSE_BUTTON_1) != GLFW_RELEASE)
    {
      《省略》
    }

  《省略》
};
```

● 消費電力をケチる

しかし、これはこれでまた別の問題があります。glfwPollEvents() はプログラムを停止させないので、キーボードのキーを操作しなくても、画面表示を繰り返し行ってしまいます。アニメーションを行っている場合はこれで問題ないのですが、画面表示に変化がないのに画面表示を繰り返し行っていると、ほかの処理の足を引っ張ってしまいますし、無駄な電力も消費します。

そこでイベントの取り出しを、キーを押している間は glfwPollEvents() で行い、キーを押していない時は glfwWaitEvents() で行うようにします。この切り替えを、glfwSetKeyCallback() 関数により登録したキーを操作したときに呼び出されるコールバック関数で行うことにします。

まず、キーボードの状態を保持するメンバ変数 keyStatus を追加し、これを GLFW_RELEASE で初期化します。

```
// ウィンドウ関連の処理
class Window
{
  《省略》

  // ワールド座標系に対する正規化デバイス座標系の拡大率の更新
  void updateScale()
  {
    scale[0] = s * 2.0f / static_cast<GLfloat>(size[0]);
    scale[1] = s * 2.0f / static_cast<GLfloat>(size[1]);
  }

  // キーボードの状態
```

```
  int keyStatus;

public:

  // コンストラクタ
  Window(int width = 640, int height = 480, const char *title = "Hello!")
    : window(glfwCreateWindow(width, height, title, NULL, NULL))
    , scale(100.0f)
    , keyStatus(GLFW_RELEASE)
  {
    《省略》
  };
```

カラーバッファの入れ替え時にメンバ変数 keyStatus を調べて、それがキーを押していないことを示す GLFW_RELEASE であれば glfwWaitEvents() を呼び出し、そうでなければ glfwPollEvents() を呼び出します。

```
  // カラーバッファを入れ替えてイベントを取り出す
  void swapBuffers()
  {
    // カラーバッファを入れ替える
    glfwSwapBuffers(window);

    // イベントを取り出す
    if (keyStatus == GLFW_RELEASE)
      glfwWaitEvents();
    else
      glfwPollEvents();

    《省略》
  }
```

また、キーボードの操作時に呼び出すメンバ関数 keyboard() を glfwSetKeyCallback() 関数によりコールバック関数として登録します。

```
    // ウィンドウのサイズ変更時に呼び出す処理の登録
    glfwSetWindowSizeCallback(window, resize);

    // マウスホイール操作時に呼び出す処理の登録
    glfwSetScrollCallback(window, wheel);

    // キーボード操作時に呼び出す処理の登録
    glfwSetKeyCallback(window, keyboard);

    // このインスタンスの this ポインタを記録しておく
    glfwSetWindowUserPointer(window, this);

   // 開いたウィンドウの初期設定
    resize(window, width, height);

    // 図形の正規化デバイス座標系上での位置の初期値
    location[0] = location[1] = 0.0f;
  }
```

GLFWkeyfun glfwSetKeyCallback(GLFWwindow *const window, GLFWkeyfun cbfun)

指定されたウィンドウに対してキーボードのキーが操作されたときに実行するコールバック関数を指定します。戻り値として以前に設定されていたコールバック関数のポインタか、コールバック関数が設定されていなければ NULL を返します。

window

対象のウィンドウのハンドル。

cbfun

実行する関数のポインタ。NULL なら現在設定されているコールバック関数を削除します。

引数 cbfun に設定する関数は、次の形式で定義します。ここでは関数名を keyboard() とします。

```
void keyboard(GLFWwindow *const window, int key, int scancode,
  int action, int mods)
{
  《省略》
}
```

window

キーボード操作の対象となったウィンドウのハンドル。

key

操作されたキー。これは glfwGetKey() の引数 key に指定するものと同じです。

scancode

操作されたキーのスキャンコード。この値はプラットホームに依存しています。

action

キーを押したときには GLFW_PRESS (1)、離したときには GLFW_RELEASE (0)、キーリピート機能が働いたときには GLFW_REPEAT (2) が格納されます。

mods

key と同時に押した Shift などのモディファイア (修飾) キー。Shift キーが同時に押されていれば GLFW_MOD_SHIFT (0x0001)、Ctrl キーは GLFW_MOD_CONTROL (0x0002)、ALT キーは GLFW_MOD_ALT (0x0004)、Windows キーや Command キーなどの Super キーは GLFW_MOD_SUPER (0x0008) キーで、複数のモディファイアキーを同時に押しているときは、これらのビット論理和が格納されます。

この関数もコールバック関数として使いますから、静的メンバ関数にします。したがって、これも glfwGetWindowUserPointer() を使ってインスタンスの this ポインタを取り出して、インスタンスのメンバ変数にアクセスします。ここでは引数 action に格納されたキーの状態をインスタンスのメンバ変数 keyStatus に代入します。

```
  // ウィンドウのサイズ変更時の処理
  static void resize(GLFWwindow *window, int width, int height)
  {
    《省略》
  }

  // マウスホイール操作時の処理
  static void wheel(GLFWwindow *window, double x, double y)
  {
    《省略》
  }

  // キーボード操作時の処理
  static void keyboard(GLFWwindow *window, int key, int scancode,
    int action, int mods)
  {
    // このインスタンスの this ポインタを得る
    Window *const
      instance(static_cast<Window *>(glfwGetWindowUserPointer(window)));

    if (instance != NULL)
    {
      // キーの状態を保存する
      instance->keyStatus = action;
    }
  }
};
```

■ サンプルプログラム step12