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

(draft 版)

和歌山大学システム工学部

床井浩平

学校の授業で C 言語や C++ 言語は習ったけど、自分自身で積極的にプログラムを書いてきたっていう経験があまりないという学生さんに、どうやって OpenGL を教えたらいいのか悩んだ挙句に、このチュートリアルを書きました。

私は以前、というか、もう 20 年くらい前から、学生さんに OpenGL を教えようとしてきました。初期の OpenGL は描画アルゴリズムを組み込んだ「固定機能」のパイプラインとして実装されていたので、その機能を組み合わせて「こうすれば絵が出るよ」みたいな教え方ができました。

しかし、今の OpenGL は機能の大部分を「プログラマブル」にしてしまったので、どんなアルゴリズムで描くかは自分で考えてプログラミングしないといけません。つまり、今の OpenGL を用いたグラフィックスプログラミングでは、CG の理論に関する知識が必須なのです。

ここで、もう一つ問題が出てきます。学校の授業でしかプログラミングをしたことがない人の中には、サンプルプログラムはいじることはできても、何もないところから自分でコードを作り出すということは不得手だという方がいらっしゃいます。この場合、教科書的に CG の知識を説明しただけでは、それをコードとして記述することができない可能性があります。

正直、私は悩みました。そういう知識は、今は Unity や Unreal Engine 4、Lumberyard、chidori、OROCHI 4、CryEngine などのミドルウェアに組み込まれているので、それらを使えば「手早く」映像を表示させることができます。これは、特に初心者において重要な要素です。だから、学校の授業でしかプログラミングをしたことがない人に OpenGL のような低水準の API によるグラフィックスプログラミングを強いるのは、もうナンセンスなんじゃないかと思ったりもします。

ですが、私にも意地があります。もう「老害」と言われても構いません。

他人の書いたプログラムを写すこと、すなわち「写経」は、プログラミングの初心者がプログラミングスキルを向上するのに有効な手法の一つだと言われます。もちろん、ただ写すだけでなく、写しているプログラムが何をしているのか、どういう理由でコードが書かれているのかを読み解きながら写します。私は、それを CG の理論を参照しつつ、「何もないところから」やってもらおうと考えました。すなわち、学生さんにスクラッチ[1]で書いていただくのです。

そんなやり方が正しいかどうかわからないのですけど、自分は悩んだ挙句にこの方法しか思いつきませんでした。はっきり言って、面倒くさいやり方だと思います。その分、解説は丁寧に書こうと思いました。それでも分かりにくかったり、説明が必要な事項を見落としていたりするかもしれません。そういうときは、すみません、ご自身でググって頂けますでしょうか。

「写経」するソースプログラムは別にテキストファイルの形で用意しますが、「写経」することが目的ですので、タイプミスが見つからないときなどの参考にとどめてくださいますようお願いします。また、このソースプログラムは説明に必要な最小限のことしか実装していません。もし誤りがある、あるいは機能が不足していることにお気づきになられましたら、ご自身で修正・拡張してくださることを期待しております。何卒よろしくお願いいたします。

---

[1] 言うまでもありませんが、プログラミング言語の「Scratch」を使えという意味ではありません。

謝辞

このチュートリアルの作成にあたっては，伊藤兎さま，西山信之さま，中山雅紀さま，嶋田有紀さま，今給黎隆さま，丸山晴さま，Torikun さま (@Toriatama_TAR)，まてぃさま (@rkmathi)、うさ m さま (@usam_code44)，kaage さま (@ageprocpp) ほか皆さまから数多くのアドバイスやアイデアを頂きました．また，日常的にも非常にお世話になっております．この場を借りて，みなさまに御礼申し上げます．

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 はグラフィックスハードウェアを制御するための基本的な機能しか持っていません。何らかの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 array、vector、およびスマートポインタなどの基本的なものにとどめ、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。実は 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 系の 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、プラットフォーム (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、パソコンの性能が向上したことで、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、他にもいくつかのグラフィックスライブラリが存在していました。しかし、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、現在 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、事実上 OpenGL (および、その後継の Vulkan[2]) しかありません。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フォ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 macOS 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 はプラットフォ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、やはりプラットフォームごとに異なる「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 はマルチプラットフォ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、macOS 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 はもう長い間メンテナンスされていません。代わりに GLUT と互換性のある freeglut (http://freeglut.sourceforge.net) というツールキットが開発されています。ただし、少なくとも本書の執筆時点では、これは macOS には対応していません。また、macOS 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、macOS (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[3] 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[4]。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。

---

[2] 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。
[3] OpenGL の過去のバージョンとの互換性を維持しない設定。
[4] 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 が使えないなら、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フォ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。また、これらの他に C++ に対応した openFrameworks (http://openframeworks.cc/) や Cinder (https://libcinder.org/)、C# に対応した OpenTK (https://opentk.github.io/) も学習が容易で高機能なツールキットです。これらは “Creative Coding”、すなわち、表現のための「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。そこで本書では、GLUT の代わりになる簡単で小さなツールキットとして、本書では GLFW (http://www.glfw.org/) を採用します。

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、マルチプラットフォ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。

● マルチプラットフォームである

GLUT と同様に Windows / macOS / 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

macOS (Mac OS X) のバージョン 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 (P. 39) 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**) にしたがって、繰り返し描画処理を行う必要があります。この繰り返しを**イベントループ**といいます。このイベントループにおいて描画処理を行った後、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、そこに対応する処理を記述する**ポーリング**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**コールバック**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 では表示に関わる情報や状態を**レンダリングコンテキスト**と呼ばれるデータに保持します。GLFW はこ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。

● **ユーザ定義のポインタを保存できる**

ユーザ定義の任意のポインタをウィンドウと関連づけて保存する機能を持っています。これを使えば C++ の this ポインタが保存できるので、静的メンバ関数にする必要があるコールバック関数からも静的でないメンバを参照することができ、クラス化が容易になります。

● **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 キーや Ctrl 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。

● **GLUT にあって 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。ただし、省かれた機能のうちいくつかは、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 世代以降の 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、macOS (Mac OS X) のバージョン 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 2017 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 2017 は無料で使用することができます (https://www.visualstudio.com/ja/vs/community/)。ただし C++ を使用するには、インストーラで「C++ によるデスクトップ開発」を選択する必要があります (図 1)。

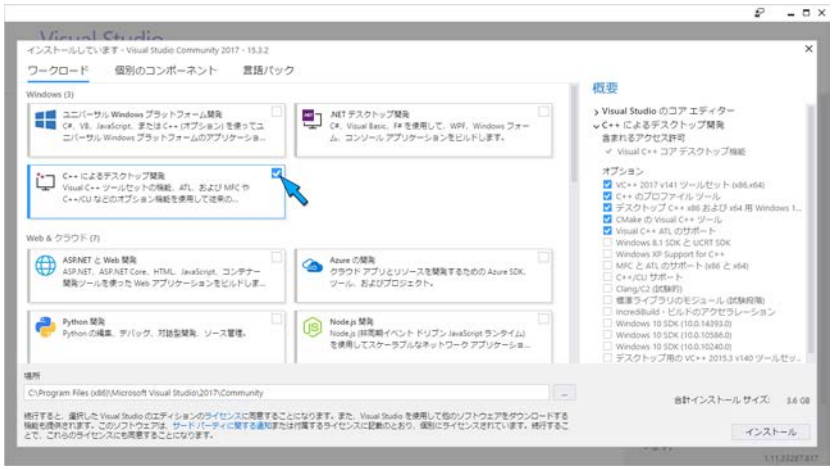


図 1 Visual Studio Community 2017 のインストーラ

macOS 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 はどのプラットフォ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。ただし本書の内容を実行するには、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。なお 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 の他に、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

Visual Studio Community 2017 では、GLFW と GLEW を NuGet を使ってプロジェクトに組み込むことができますが[5]、ここでは自分でソースファイルからビルドする方法を説明します。まず CMake (https://cmake.org/) を入手します。ダウンロードページ (https://cmake.org/download/) から Windows 用のインストーラ (cmake-X.Y.Z-win32-x86.msi など、X, Y, Z は数字) をダウンロードし、これを実行して CMake をインストールしてください。この CMake を使って、GLFW のビルドに使う Visual Studio のソリューションファイルを生成します。

次に、GLFW のソースファイルを入手します。ソースファイルのパッケージは、プロジェクトのサイト (http://www.glfw.org/) の “Download GLFW 3.X.Y” のボタン (X, Y は数字) をクリックすれば、glfw-3.X.Y.zip というファイル名でダウンロードできます。このファイルを選択して右クリックのメニューから「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。

先ほどインストールした CMake の GUI 版 (cmake-gui) を起動し、“Where is the source code:” の欄の右にある Browse Source のボタンをクリックして、GLFW のソースファイルを展開したフォルダを指定してください。また、 “Where to build the binaries:” の欄の右の Browse Build のボタンをクリックしてソースファイルと同じフォルダを開き、そこに “build” というフォルダを作って、そこを選択してください (図 2)。

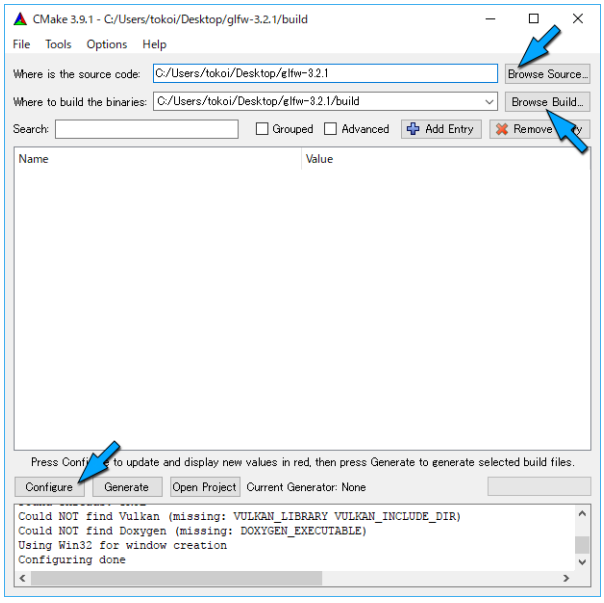


図 2 GLFW のプロジェクトを生成する場合の CMake のフォルダの設定

その後 Configure をクリックすると、どのバージョンの Visual Studio のプロジェクト (ソリュ

[5] インストーラ (図 1) の「個別のコンポーネント」のタブで「NuGet パッケージ マネージャー」をインストールしておく必要があります。

ーション) ファイルを生成するか尋ねてきます (図 3)。Visual Studio Community 2017 用のものを生成する場合は、“Visual Studio 15 2017” を選択してください。なお、これは 32bit (x86, Win32) のライブラリファイルを作るプロジェクトファイルを生成します。64bit (x64, Win64) 用のライブラリファイルを作る場合は、“Visual Studio 15 2017 Win64” を選択してください。

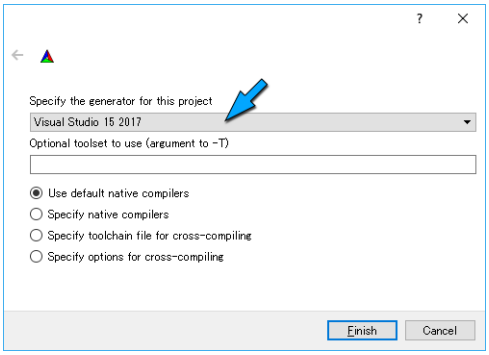


図 3 生成するプロジェクトファイルの種類の選択

Configure に成功すると生成するソリューションファイルの設定のカスタマイズ項目が表示されます。この中の CMAKE_INSTALL_PREFIX には GLFW のインストール先を指定します。このインストール先は任意ですが、ここでは C:¥OpenGL にインストールすることにします。

CMAKE_INSTALL_PREFIX の右側の欄の右端の…をクリックして C:¥ を開き、OpenGL というフォルダを作って、そこを選択してください (図 4)。これ以外の GLFW_BUILD_DOCS や GLFW_BUILD_EXAMPLES、GLFW_BUILD_TESTS は本書では使用しないので、チェックを外しても問題ありません。

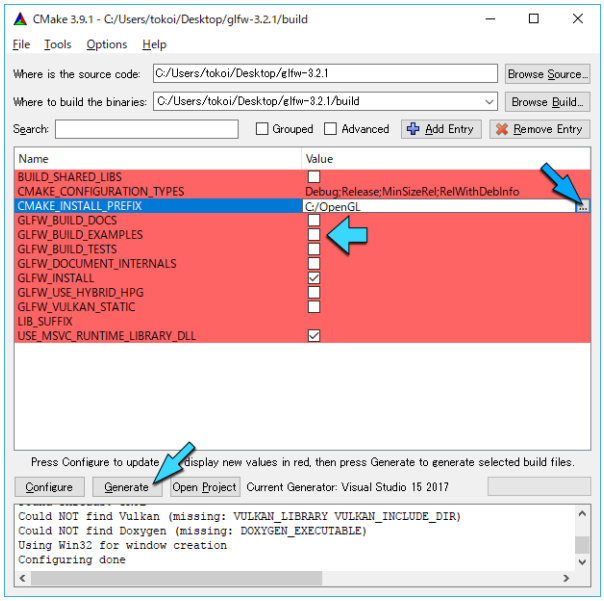


図 4 GLFW のプロジェクト設定のカスタマイズ

最後に Generate のボタンをクリックすれば、“Where to build the binaries:” の欄に指定したフォルダに Visual Studio Community 2017 のプロジェクトファイル GLFW.sln が作成されます。この

ファイルをダブルクリックするか、CMake の Open Project ボタンをクリックすれば、Visual Studio Community 2017 が起動します。

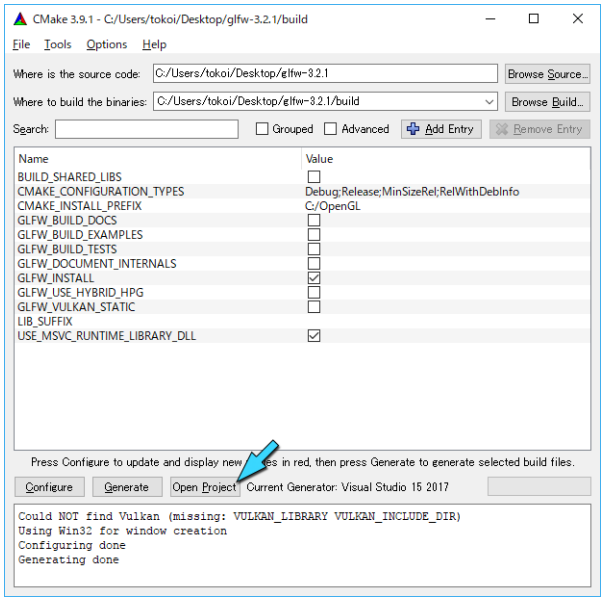


図 5 GLFW のプロジェクト (ソリューション) ファイルを開く

Visual Studio Community 2017 が起動したら、ソリューションエクスプローラーの “INSTALL” のプロジェクトを選択し、**右クリック**のメニューから「ビルド」を選んでください (図 6)。これで GLFW のライブラリファイルが作成され、ヘッダファイル等とともにインストール先に指定したフォルダ C:¥OpenGL にコピーされます。

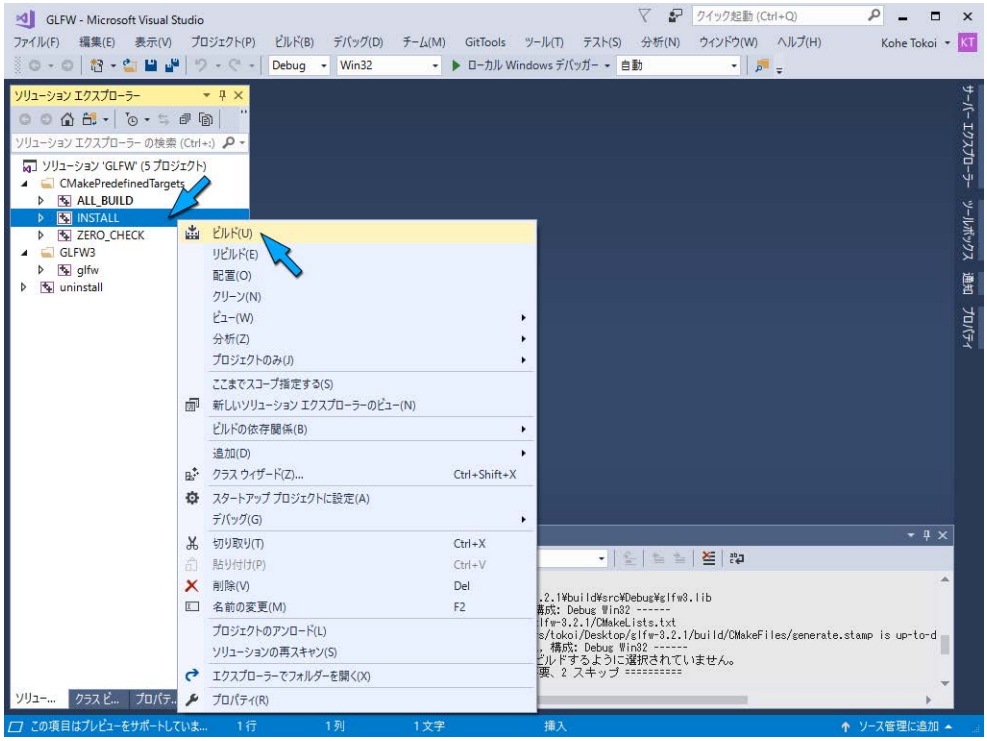


図 6 GLFW のビルドとインストール

2.2.2 macOS

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

GLEW についても、ここでは自分でソースファイルからビルドする方法を説明します。CMake は既にインストールされているとします。

まず GLEW のソースファイルを入手します。ソースファイルのパッケージは、プロジェクトのサイト (http://glew.sourceforge.net/) の "Downloads" の "Source" の "ZIP" の文字をクリックすれば、glew-2.X.Y.zip (X, Y は数字) というファイル名でダウンロードできます。このファイルを選択して右クリックのメニューから「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。

次に、CMake の GUI 版 (cmake-gui) を起動し、"Where is the source code:" の欄の右にある Browse Source のボタンをクリックして、GLEW のソースファイルを展開したフォルダの中にある build フォルダの中の cmake フォルダを指定してください。また、"Where to build the binaries:" の欄の右の Browse Build のボタンをクリックしてソースファイルと同じフォルダを開き、その中にある "build" フォルダを選択してください (図 7)。

その後 Configure をクリックし、生成するプロジェクト (ソリューション) ファイルの Visual Studio のバージョンに "Visual Studio 15 2017" を選択すれば、32bit (x86, Win32) のライブラリファイルを作るプロジェクトファイルを生成します。64bit (x64, Win64) 用のライブラリファイルを作る場合は、ここで "Visual Studio 15 2017 Win64" を選択してください。

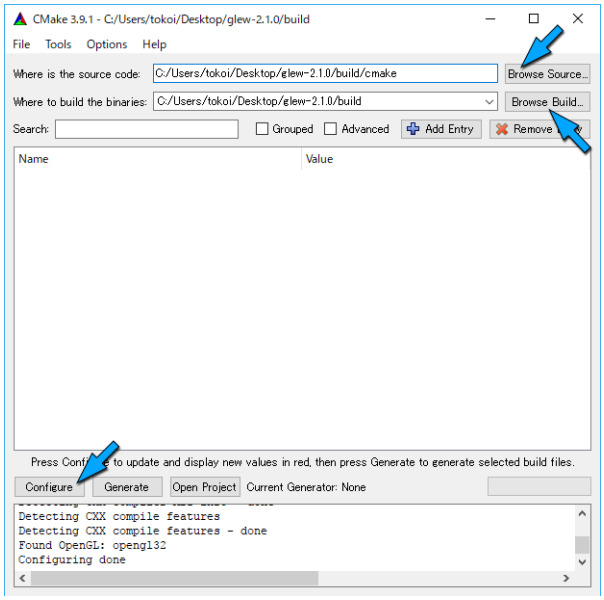


図 7 GLEW のプロジェクトを生成する場合の CMake のフォルダの設定

Configure に成功すると生成するソリューションファイルの設定のカスタマイズ項目が表示されます。この中の CMAKE_INSTALL_PREFIX には GLFW のインストール先を指定します。これには GLFW と同じ場所を指定してください。ここでは C:¥OpenGL にインストールします。

CMAKE_INSTALL_PREFIX の右側の欄の右端の…をクリックして C:¥ を開き、OpenGL というフォルダを選択してください (図 8)。

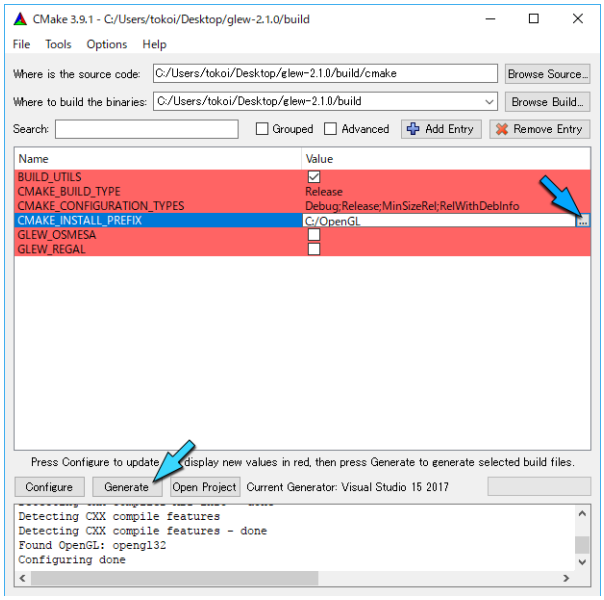


図 8 GLEW のプロジェクト設定のカスタマイズ

最後に Generate のボタンをクリックすれば、“Where to build the binaries:” の欄に指定したフォルダに Visual Studio Community 2017 のプロジェクトファイル GLFW.sln が作成されます。このファイルをダブルクリックするか、CMake の Open Project ボタンをクリックすれば、Visual

Studio Community 2017 が起動します。

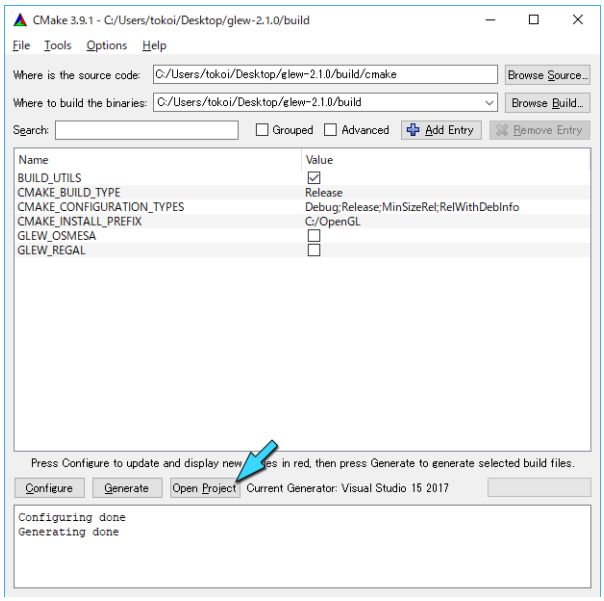


図 9 GLEW のプロジェクト (ソリューション) ファイルを開く

Visual Studio Community 2017 が起動したら、ソリューションエクスプローラーの “INSTALL” のプロジェクトを選択し、**右クリック**のメニューから「ビルド」を選んでください (図 6)。これで GLFW のライブラリファイルが作成され、ヘッダファイル等とともにインストール先に指定したフォルダ C:¥OpenGL にコピーされます。

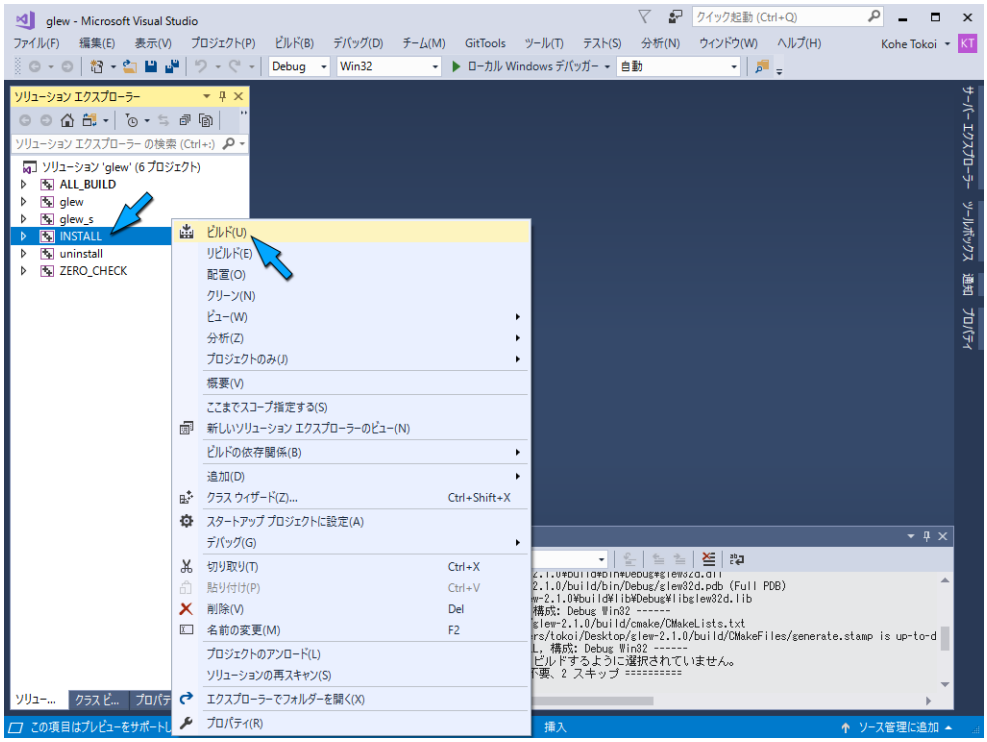


図 10 GLEW のビルドとインストール

2.3.2 macOS

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/)、および 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-2.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-2.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-2.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 にインストールのときに指定したインストール先を指定してください。

```
$ cd ~/Desktop/glew-2.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/)、および 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-2.X.Y.tgz
$ cd glew-2.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 にインストールのときに指定したインストール先を指定してください。

```
$ cd glew-2.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フォ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**

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」を選んでください (図 11)。

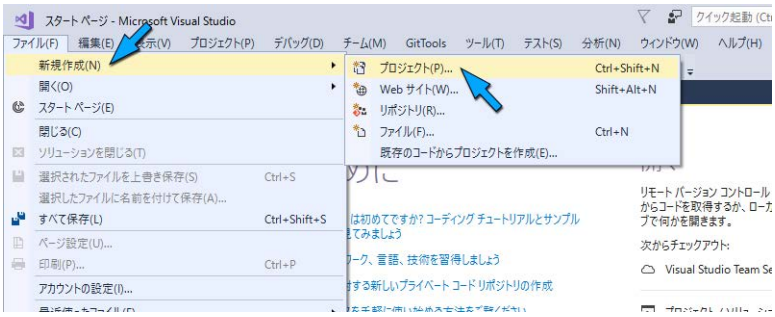


図 11 プロジェクトの作成

インストール済みのテンプレートの「Visual C++」にある「空のプロジェクト」を選び、作成するプログラムの「名前」(図 12 では “sample”)を設定した後「OK」をクリックしてください。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。

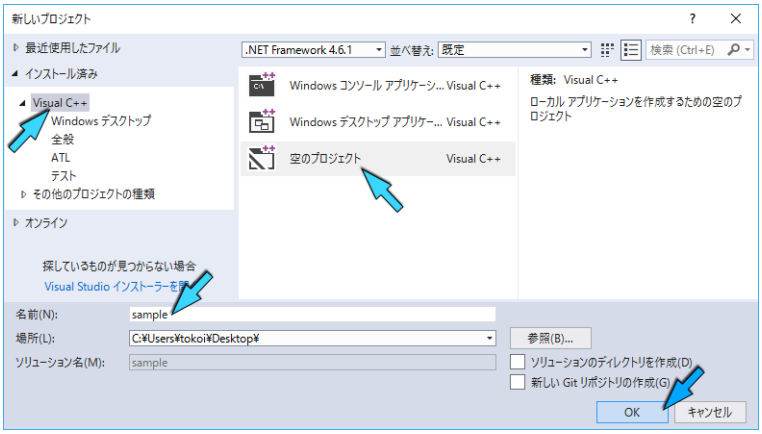


図 12 「Empty Project」の選択

これで空のプロジェクトが作成されます (図 13)。

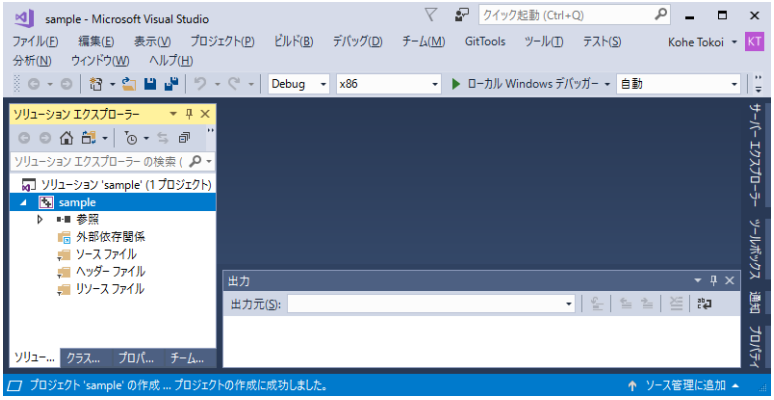


図 13 空のプロジェクト

● **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 14)。

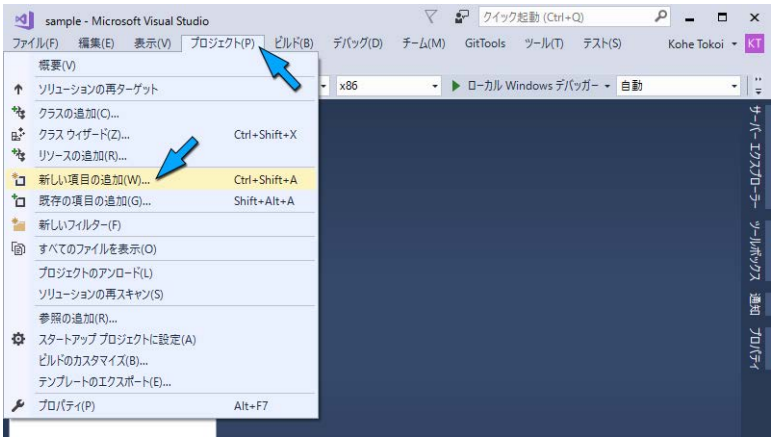


図 14 新しい項目の追加

「新しい項目の追加」のウィンドウ (図 15) 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。

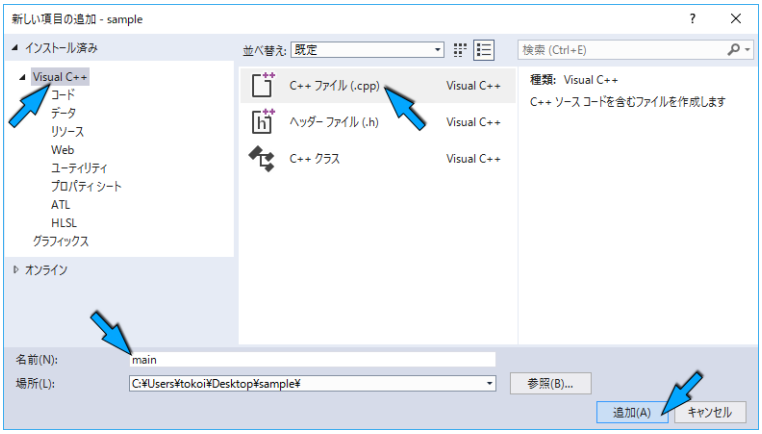


図 15 C++ のソースファイルの追加 (新規作成)

テキストエディタのウィンドウ (図 16) が開きます。ここにソースプログラムを入力します。

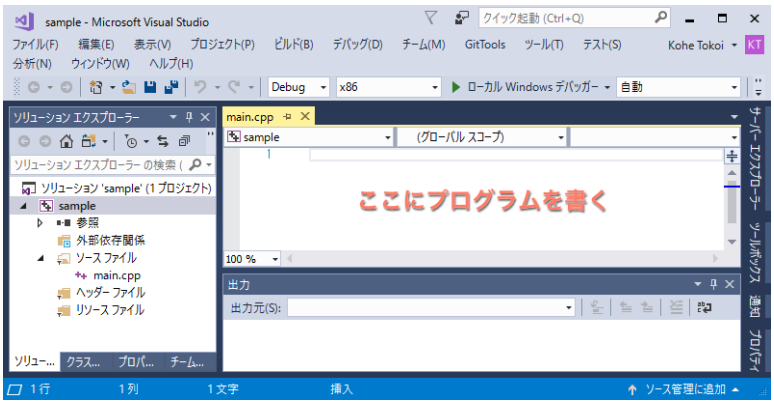


図 16 ソースプログラムの編集

● **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、および OpenGL 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 17)。

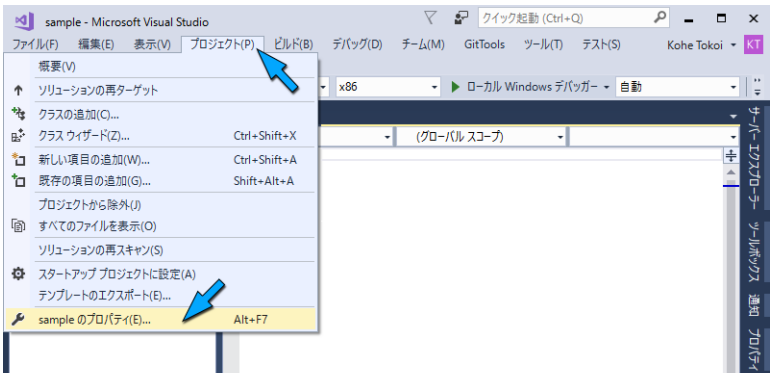


図 17 プロジェクトの「プロパティ」の選択

プロパティページの「C/C++」の項目にある「全般」を選択して、「追加のインクルードディレクトリ」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 18)。

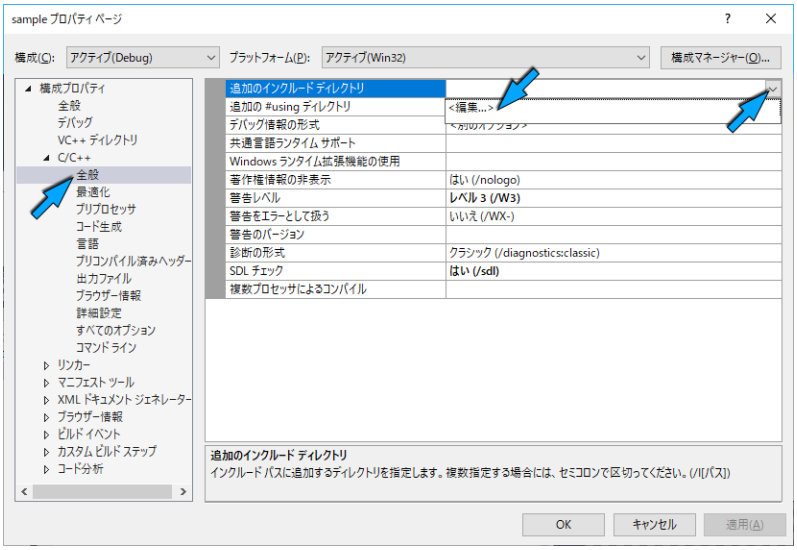


図 18 プロジェクトの「プロパティ」の設定

「フォルダの追加」のアイコンをクリックしたあと…をクリックしてください。

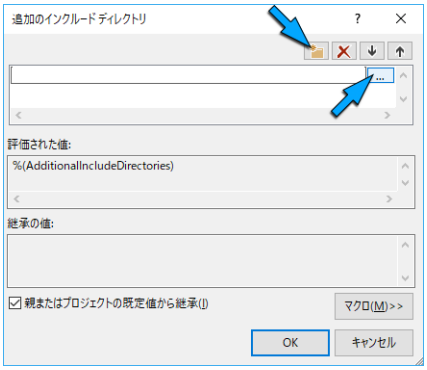


図 19 追加のインクルードディレクトリ

「追加のインクルードディレクトリ」には、GLFW や GLEW をインストールしたディレクトリ (C:¥OpenGL) の中の include ディレクトリを選択します。

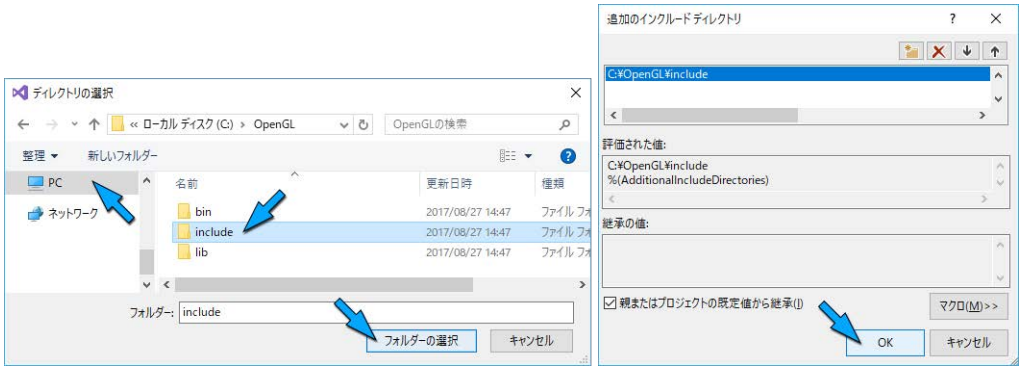


図 20 追加のインクルードディレクトリの選択

次にプロパティページの「C/C++」の項目にある「プリプロセッサ」を選択して、「プリプロセッサの定義」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 21)。

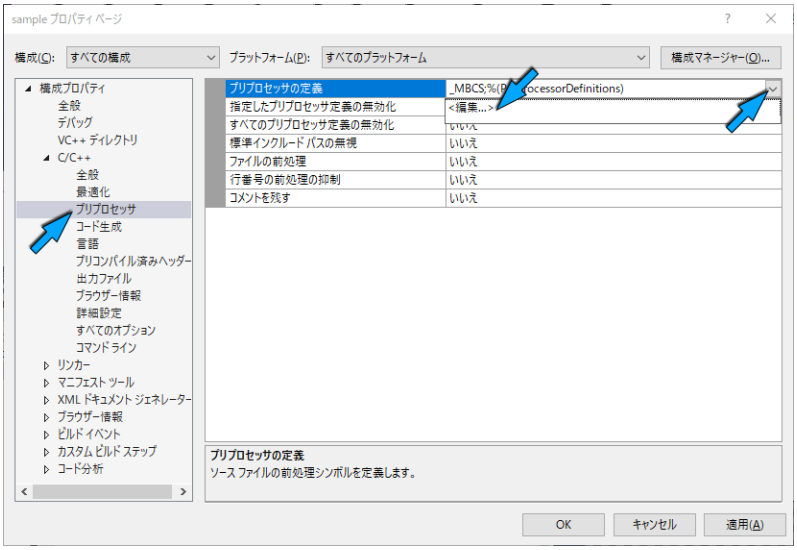


図 21 プリプロセッサの定義の編集

「プリプロセッサの定義」に GLEW_STATIC を入力して「OK」をクリックします (図 22)。

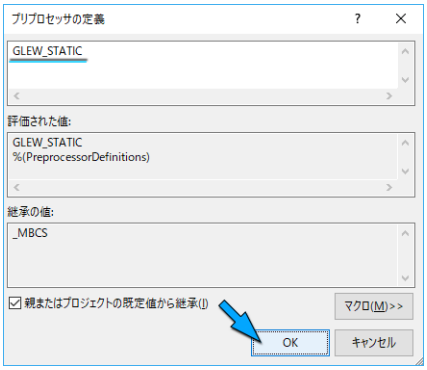


図 22 プリプロセッサの定義に GLEW_STATIC を追加

今度はプロパティページの「リンカー」の項目にある「全般」を選択して、「追加のライブラリディレクトリ」の欄の右端の▽をクリックして「<編集...>」をクリックしてください (図 23)。

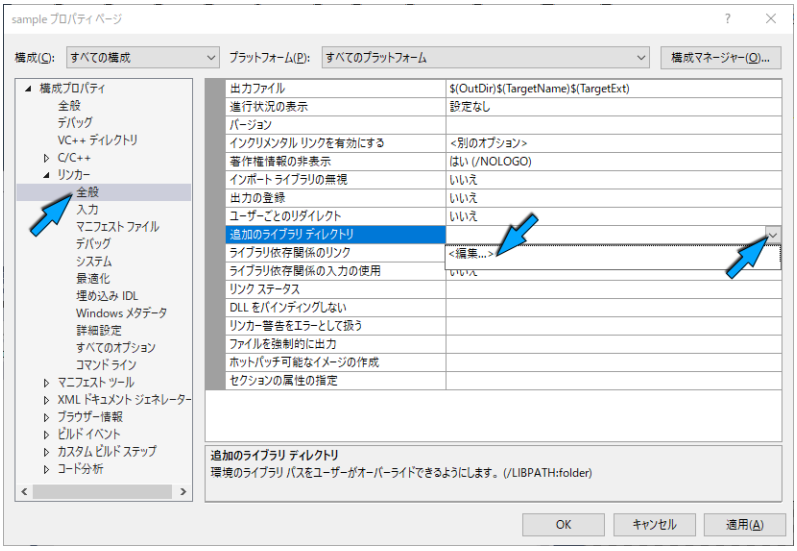


図 23 追加のライブラリディレクトリの選択

「フォルダの追加」のアイコンをクリックしたあと … をクリックしてください。

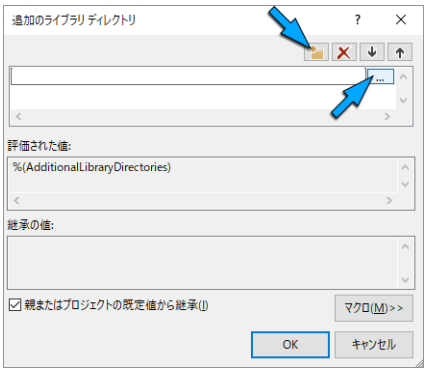


図 24 追加のライブラリディレクトリ

「追加のライブラリディレクトリ」には、GLFW や GLEW をインストールしたディレクトリ (C:¥OpenGL) の中の lib ディレクトリを選択します。

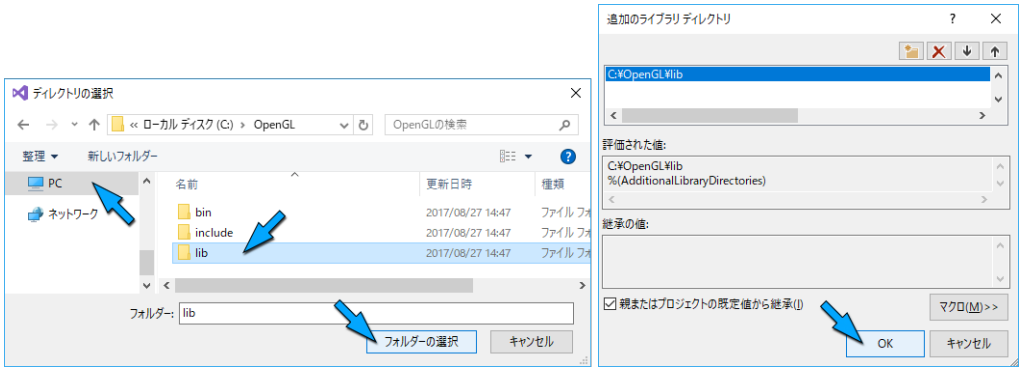


図 25 追加のライブラリディレクトリの選択

最後にプロパティページの「リンカー」の項目にある「入力」を選択して、「追加の依存ファイル」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 26)。

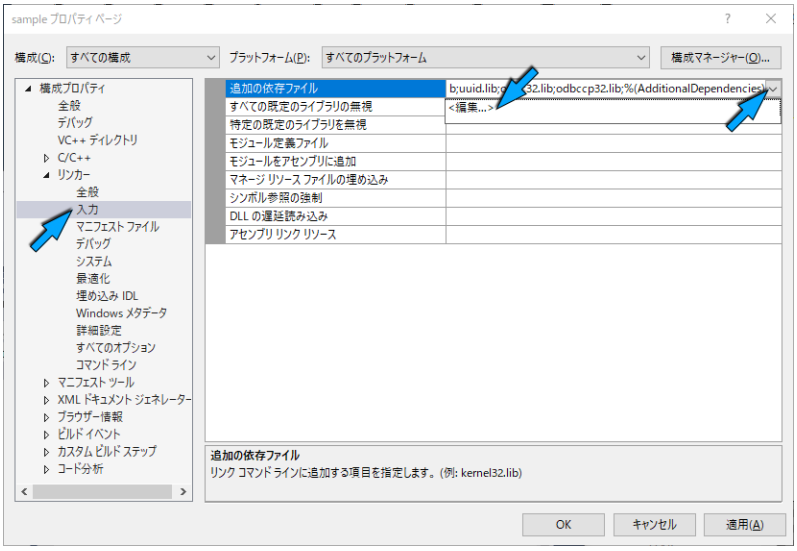


図 26 追加の依存ファイルの選択

「追加の依存ファイル」に opengl32.lib、glfw3.lib、および libglew32d.lib を入力して「OK」をクリックしてください (図 27)。

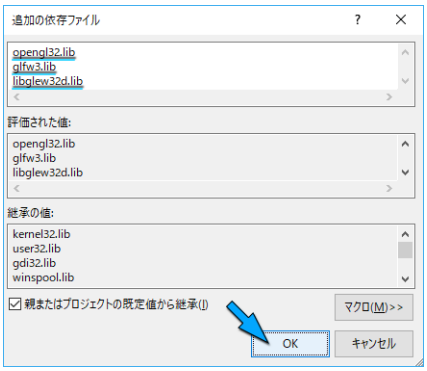


図 27 追加の依存ファイルの入力

補足：「構成」と「プラットフォーム」

図 18 のプロパティページの上部にある「構成」は、デバッグのときに用いる「Debug」と、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。

また「プラットフォーム」は、一般的な PC では Win32 (32bit) と x64 (64bit) を切り替えることができます。ただし、x64 に切り替えてビルドしたプログラムを実行するには、オペレーティングシステム (Microsoft Windows) が 64bit 版である必要があります。

前節で GLFW と GLEW を Debug 構成の 32bit で作成している場合は、ここの設定もそれに

合わせる必要があります。もし Release 構成や 64bit のプログラムを作成したい場合は、GLFW と GLEW もそれに合わせて作成し、それぞれ異なるフォルダに配置する必要があります。

補足：ソースプログラムへの設定の埋め込み

前述の「追加の依存ファイル」の設定は、ソースプログラムに次の 3 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")
#pragma comment(lib, "glfw3.lib")
#pragma comment(lib, "libglew32d.lib")
```

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 のウィンドウの他に「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:s¥"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が共有されるため、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、この方法では作成したプログラム本体の他にリンクしている 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。

これを避けるために、前述の設定では GLFW と GLEW をスタティックリンクしています。このうち GLEW をスタティックリンクする場合には、プリプロセッサの定義に GLEW_STATIC を追加する必要があります (図 22)。この設定は glew.h を #include する前に GLEW_STATIC を

#define すれば省くことができます。

```
#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、他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 と GLEW をダイナミックリンクする場合は、それぞれ glfw3.lib の代わりに glfw3dll.lib、libglew32d.lib の代わりに glew32d.lib をリンクします (表 1)。そして glfw3.dll と glew32d.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 glew32d.dll を、32bit 版 Windows では C:¥Windows¥System32、64bit 版 Windows では C:¥Windows¥SysWOW64 に置くか、「システムの詳細設定[6]」で「システム環境変数」の Path に glfw3.dll と glew32d.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 libglew32d.lib	(なし)
ダイナミックリンク	(なし)	glfw3dll.lib glew32d.lib	glfw3.dll glew32d.dll

なお、GLFW や GLEW のバイナリパッケージでは、ライブラリファイルは「Release」構成でビルドされています。また、GLEW のスタティックリンクライブラリのファイル名は glew32s.lib、ダイナミックリンクライブラリのファイル名は glew32.lib と glew32.dll になっています。

3.1.2 macOS

● プロジェクトの新規作成

macOS 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 28) 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。

[6] 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 28 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 29)。

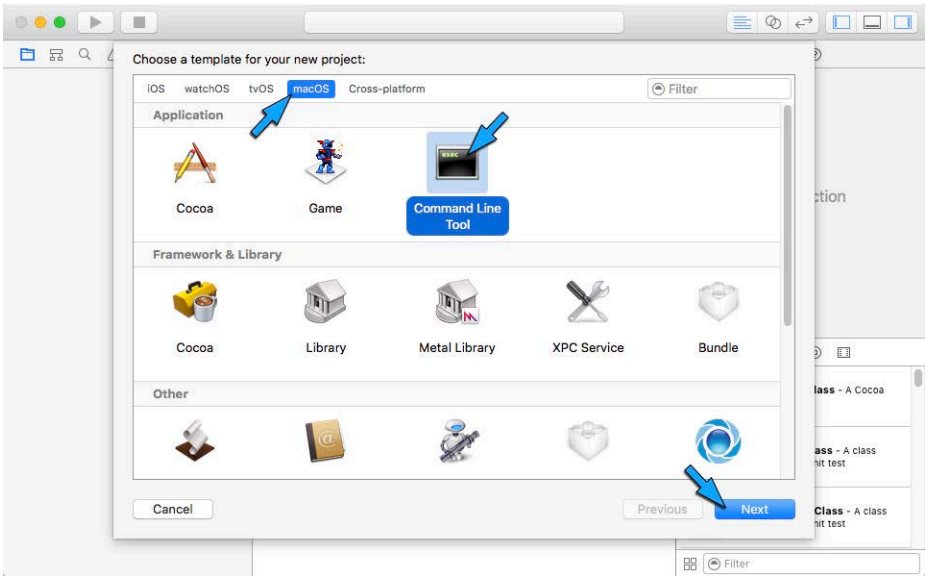


図 29 プロジェクトのテンプレートの選択

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 30)。

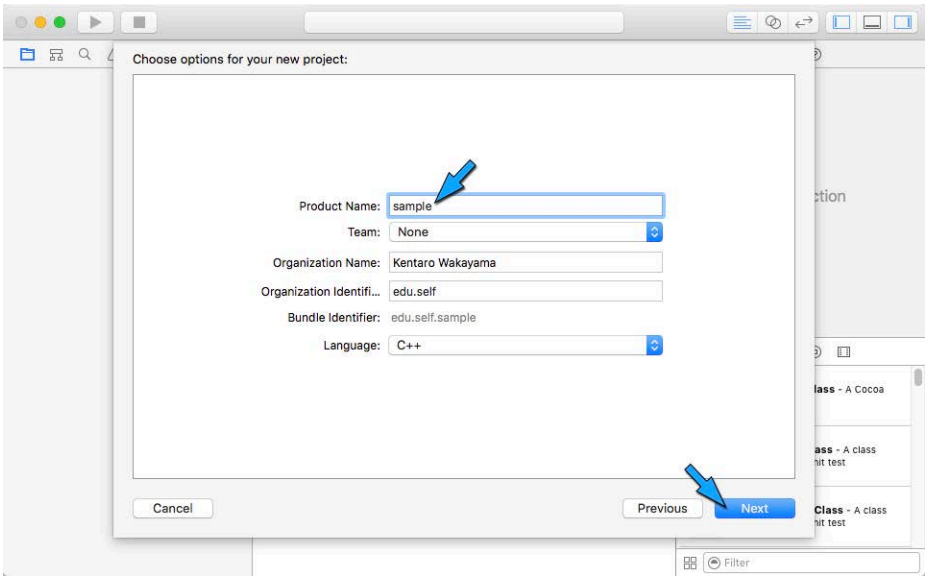


図 30 プロジェクトのオプションの設定

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 31)。

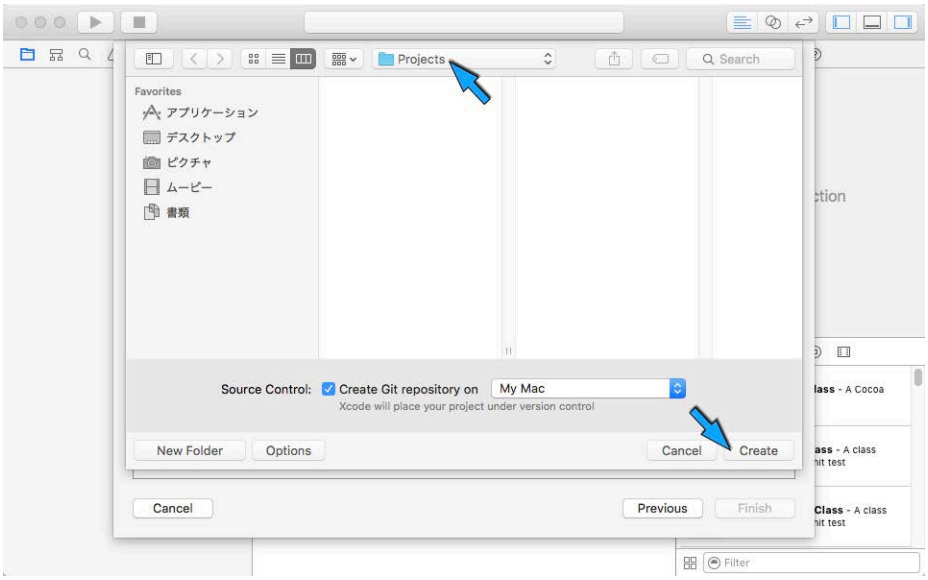


図 31 プロジェクトのディレクトリの保存先

● **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 32 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 (1)、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 (2)。次に中央部の “Build Settings” を選択します (3)。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 (4) “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 (5)。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 (6)、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 (7)。

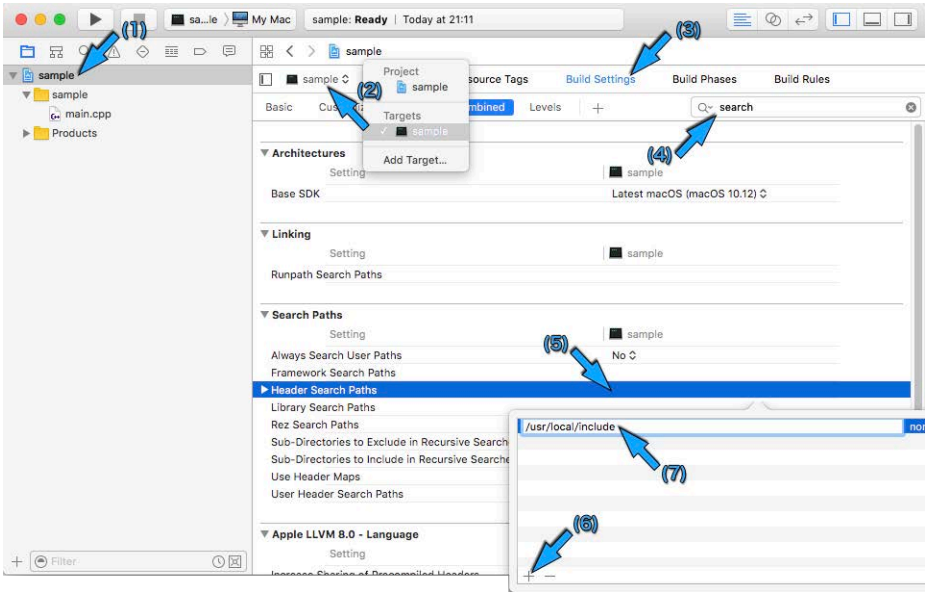


図 32 ヘッダファイルの検索パスの設定

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 33) をダブルクリックし (1)、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 (2)(3)。

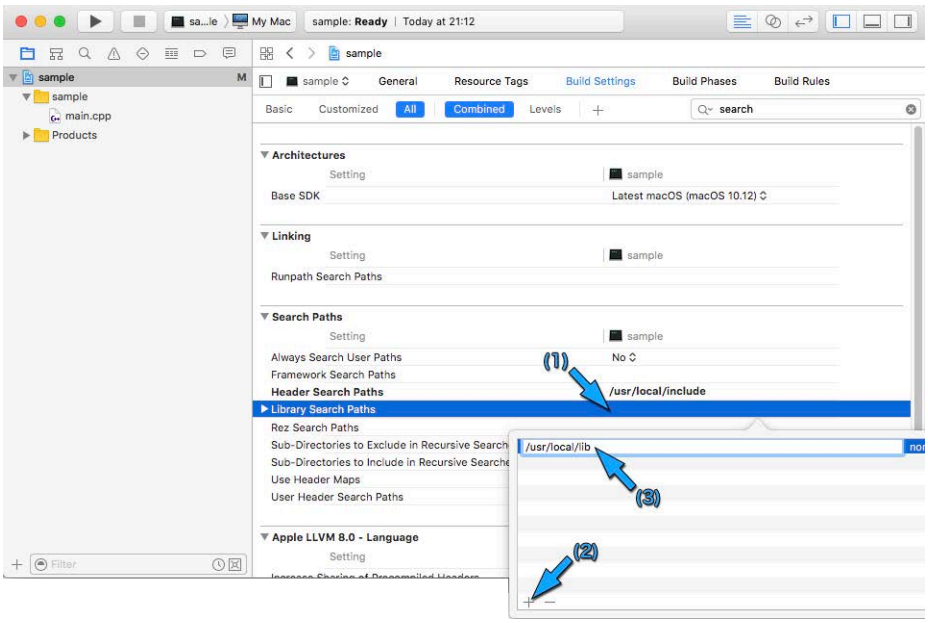


図 33 ライブラリファイルの検索パスの設定

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 34)。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
```

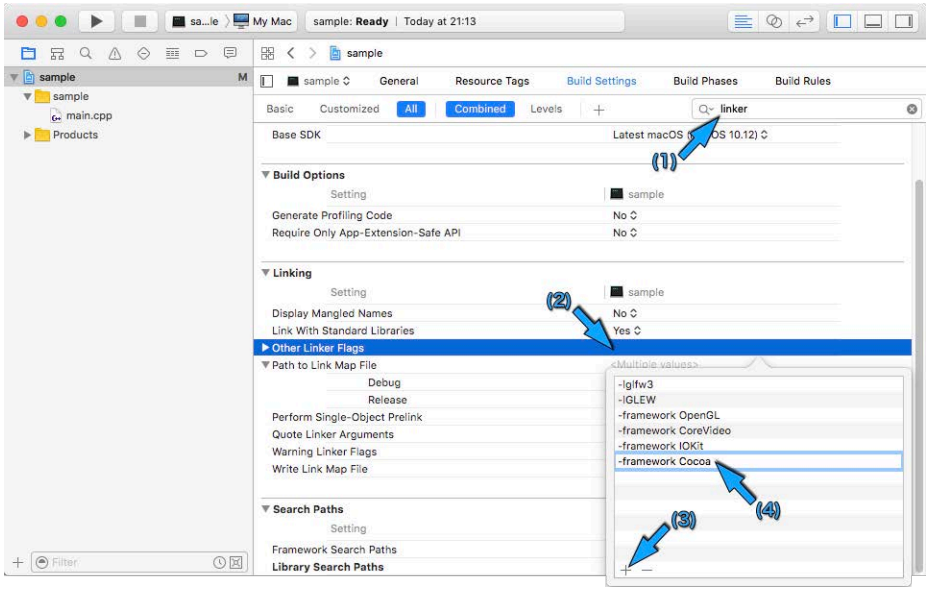


図 34 リンクするライブラリとフレームワークの指定

最後に、コンパイラのバージョンを設定します。本書で作成するプログラムは、一部に C++11 (ISO 標準 ISO/IEC 14882:2011) の機能を使っているため、検索窓に compiler などを入力して C++ Language Dialect を探し (1)、そこ (2) で “C++11 [-std=c++11]” を選びます (3)。

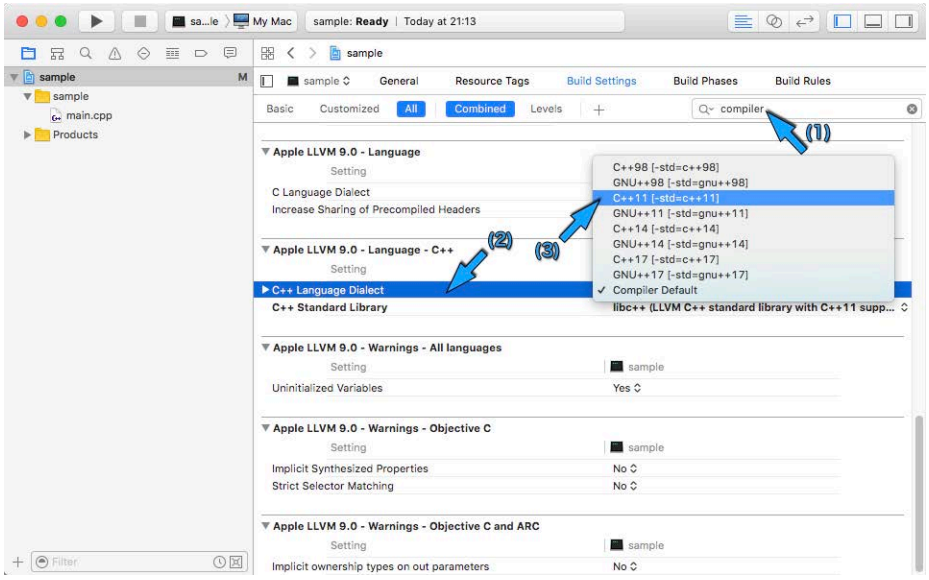


図 35 リンクするライブラリとフレームワークの指定

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 36)。これに main() 関数が定義されています。

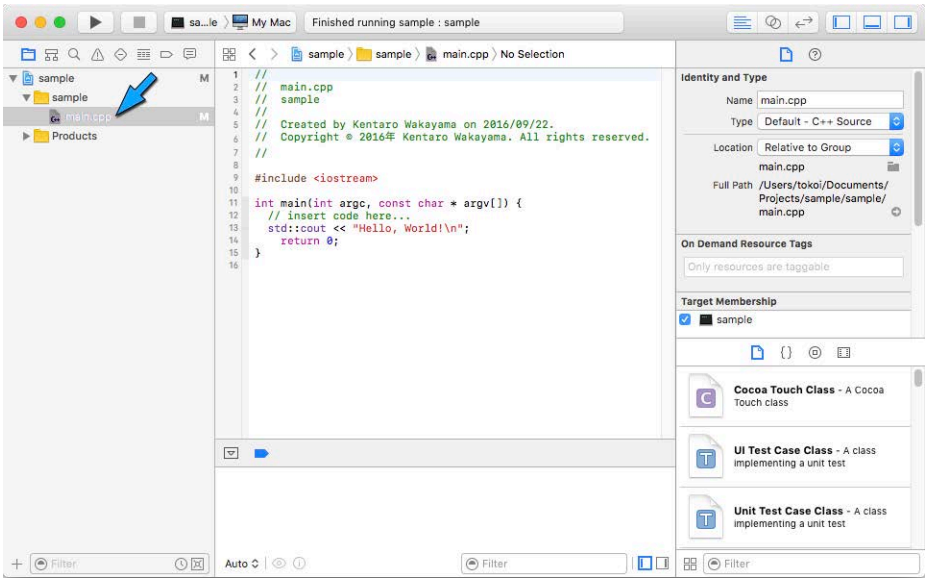


図 36 ソースプログラムの編集

● 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 -std=c++11 -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 37)。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。

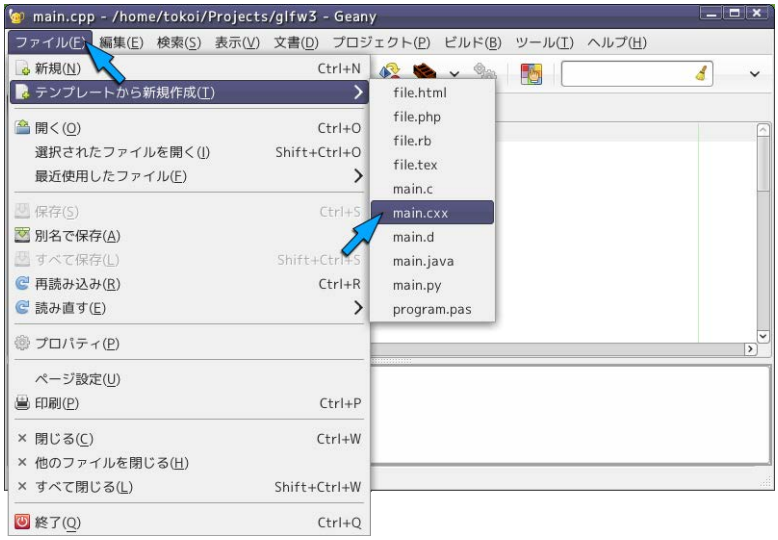


図 37 ソースファイルの新規作成

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 38)。

図 38 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 39) 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。設定が終わったら「OK」をクリックしてください。

```
-std=c++11 -lglfw3 -lGLEW -lGL -lXrandr -lXinerama -lXcursor -lXi -lXxf86vm -lX11
-lpthread -lrt -lm -ldl
```

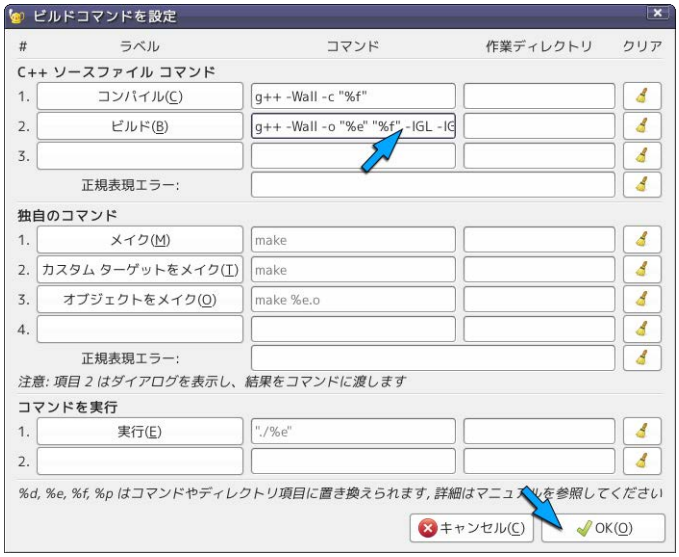


図 39 ビルドコマンドを設定

● 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 -std=c++11
LDLIBS  = -lglfw3 -lGLEW -lGL -lXrandr -lXinerama -lXcursor -lXi ¥
        -lXxf86vm -lX11 -lpthread -lrt -lm -ldl
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繰り返し描画する (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()
{
}
```

■ サンプルプログラム step00

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。

3.2.2 GLFW の初期化

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。他の全ての 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 ウィンドウの作成

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。

```
#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;
    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 に他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。

3.2.4 描画するウィンドウの指定

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。この後に開いたウィンドウに対する設定や図形の描画などを行います。

```
#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;
  }

  // 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。

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。

```
#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(GLfloat R, GLfloat G, GLfloat B, GLfloat 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 GLfloat 型 (float 型と等価) の 0～1 の値。1 が最も明るく、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 (図 40)。

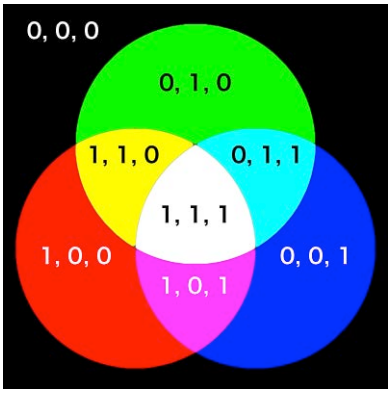


図 40 R, G, B の値と色

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フォーム (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 型に対応します。なお、OpenGL 4.2 では GLclampf 型と GLclampd 型は廃止され、それぞれ GLfloat、GLdouble に置き換えられました (表 2 の <注>)。

また、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 <注>
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 <注>

※ 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 による描画ループは、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 (フレームバッファのカラーバッファ, 図 41) 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** (図 42) 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、このとき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 41)、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。

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。待ち時間の最大値を引数に秒で指定できる glfwWaitEventsTimeout(double sec) もあります。

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。

補足：バッファについて

バッファは一般に緩衝器と訳され、一般に二つのものの一方からもう一方に何らかの影響を与える状況にあるとき、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 41)。

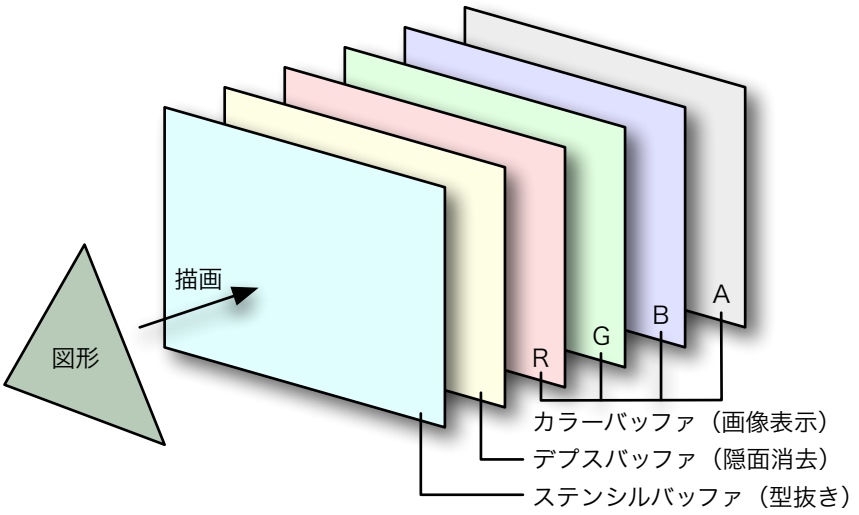


図 41 フレームバッファの構成

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。この描画が完了し、かつ直前の画面表示が完了した時点 (垂直同期タイミング、V-Sync) でこの二つのカラーバッファを入れ替える (図 42) ことで、ちらつきの発生を抑えます。

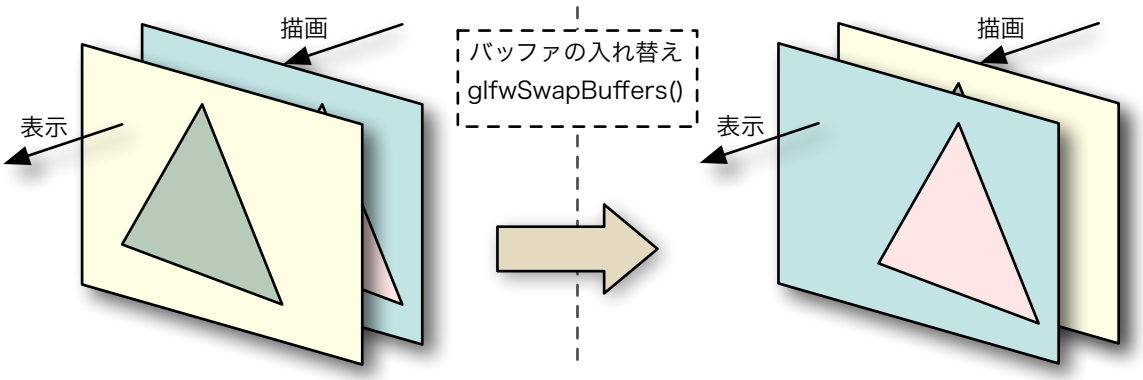


図 42 ダブルバッファリング

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。

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：関数名について

関数名が gl〜 で始まるものは OpenGL の API です。一方、間数名が glfw〜 で始まるものは GLFW の関数、glew〜 で始まるものは GLEW の関数です。

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。そのためにはプログラムの最後で glfwTerminate() 関数を呼び出します。ところが、このプログラムでは glfwCreateWindow() が失敗した場合にも、プログラムを終了するようにしています。当然、このときも glfwTerminate() を呼び出す必要があります。

さらに、このあとプログラムを追加していくと、他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() 関数を呼び出す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。

```
#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ときは 0、失敗したときは 0 以外の値を返します。

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。

3.2.8 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 macOS 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。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、プラットフォ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 より前で #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、プラットフォ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、処理に成功すれば GLEW_OK (値は 0) を返します。失敗すれば 0 以外の値を返します。

3.2.9 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_RESIZABLE

hint に GL_TRUE を指定すればマウス等でウィンドウのサイズが変更できるようになります。デフォルトは GL_TRUE です。GL_FALSE を設定していても、glfwSetWindowSize() 関数によってプログラム中からウィンドウのサイズを変更することは可能です。

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

使用する OpenGL の最低バージョン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 3.2 以降の機能しか使用しないなら 3 です。デフォルトは 1 です。

GLFW_CONTEXT_VERSION_MINOR

使用する OpenGL の最低バージョン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 以降の機能しか使用しないなら 2 です。デフォルトは 0 です。

GLFW_OPENGL_FORWARD_COMPAT

OpenGL のバージョン 3.0 以降において、Forward Compatible Profile (古い機能を削除した前方互換プロファイル) を使用する場合は、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。hint には、OpenGL 3.0 より前の古い機能をすべて含む Compatible Profile を使う場合は GLFW_OPENGL_COMPAT_PROFILE、古い機能をサポートしない Core Profile を使う場合は GLFW_OPENGL_CORE_PROFILE を指定します。デフォルトは GLFW_OPENGL_ANY_PROFILE 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。

3.2.10 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);
```

■ サンプルプログラム step01

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

カラーバッファを入れ替える際に待つディスプレイの垂直同期タイミング (V-Sync) の最小回数。通常は 1。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、数値の上ではフレームレート (frame per second, fps) が上昇することがあるが、完全な画面表示が行われるわけではない。

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 2017 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 43)。その下の「“プロジェクト名” のビルド」でも構いません。

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ときは、先にビルドを実行します。

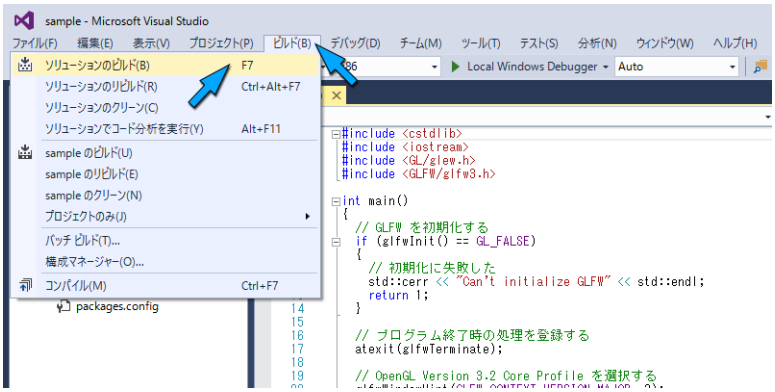


図 43 Visual Studio Community 2017 によるプログラムのビルド

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' は他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 macOS

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 44)。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 44 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 -std=c++11 -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 -std=c++11 -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 45)。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 45 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 -std=c++11 -lglfw3 -lGLEW -lGL -lXrandr -lXinerama -lXcursor -lXi -lXxf86vm -lX11 -lpthread -lrt -lm -ldl 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 -std=c++11 -lglfw3 -lGLEW -lGL -lXrandr -lXinerama ¥
-lXcursor -lXi -lXxf86vm -lX11 -lpthread -lrt -lm -ldl
```

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

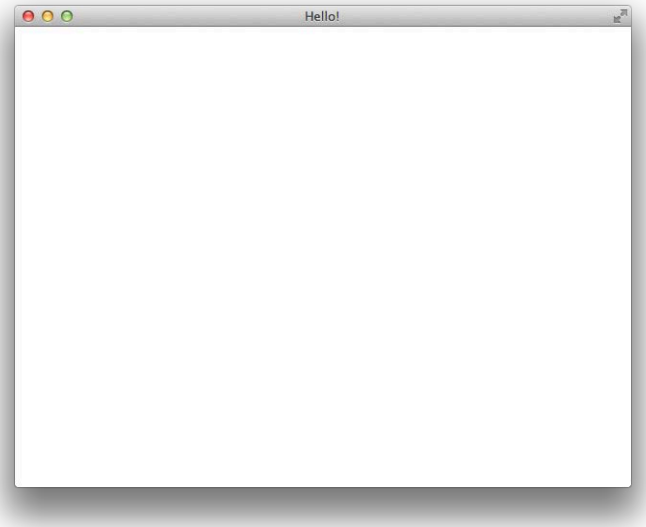


図 46 開いたウィンドウ (macOS の場合)

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 47)。

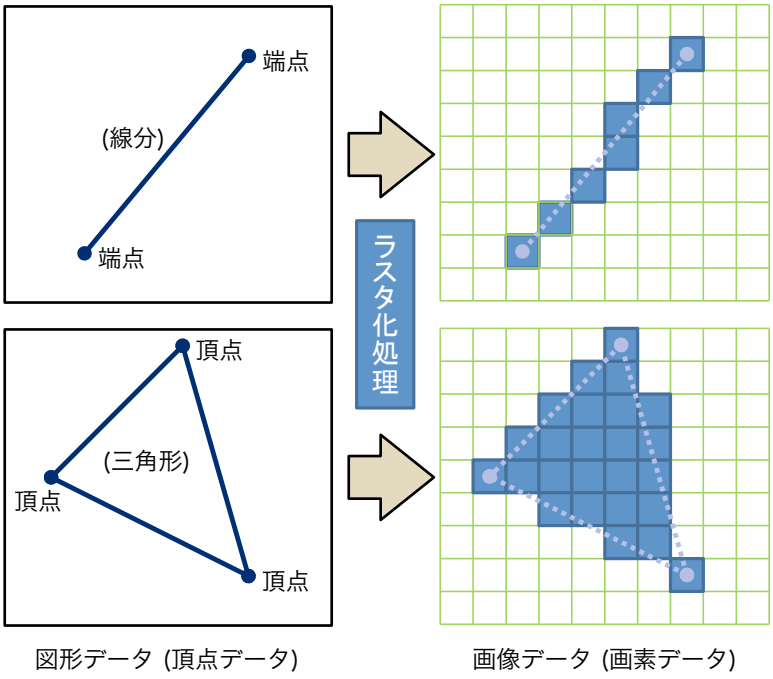


図 47 画像の生成

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 48)。

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)

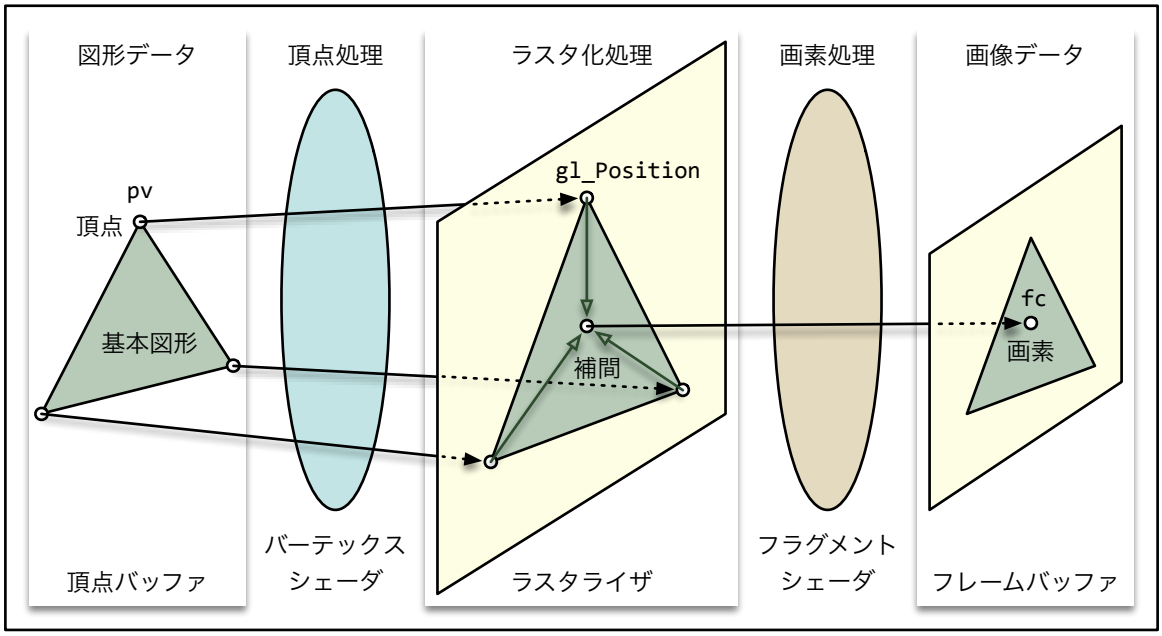


図 48 レンダリングパイプライン

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**といいます。そして、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、これらの他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

GLSL は C あるいは C++ 言語に似ており、/* … */ で挟まれた範囲や // 以降はコメントとして扱われるほか、C 言語のプリプロセッサの機能 (#define など) も使えます。

● バーテックスシェーダのソースプログラム point.vert

描画する図形の頂点の位置をそのままレンダリングパイプラインの次のステージに送るなら、バーテックスシェーダのソースプログラムは次のようになります。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 以降で使用できる GLSL のバージョン 1.5 を使用することを表し、core は OpenGL 3.0 より前のバージョンの機能をサポートしない Core Profile を用いることを表します。

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 **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、この他に 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)。この要素は組み合わせて使うことができ、一部の要素だけを使用したり、その順序を入れ替えて使用したりできます。この機能は swizzling といいます。

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	

要素のアクセス

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処理を実行します。

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 (ここではラスタライザ) 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

ラスタライザは描画する図形の各画素について、フラグメントシェーダの処理を実行します。画素に赤色を設定するなら、フラグメントシェーダのソースプログラムは次のようになります。

```
#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を作成し、program という定数に設定します。

```
  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、'#' を含む行の行末には '¥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。

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 (番号)。

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。

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 (番号)。

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、他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 (番号)。

shader

組み込むシェーダオブジェクト名 (番号)。

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 (番号)。

shader

取り外すシェーダオブジェクト名 (番号)。

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 (番号)。

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。

なお、プログラムオブジェクト自体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。

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 (番号)。

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 attribute 変数の場所 (番号) 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 (番号)。

index

引数 name に指定した attribute 変数の場所を示す 0 以上の整数。デフォルトは 0。

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 (番号)。

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 (番号)。

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() 関数より前に定義します。ソースプログラムの文字列が NULL のときは、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 (番号)。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() を実行してプログラムオブジェクトを作成します。そして図形の描画を行う前に、この関数の戻り値のプログラムオブジェクト名 (番号) 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。なお、この場合はシェーダのソースプログラムの文字列定数 vsrc と fsrc はコンパイル時に決定できるので、constexpr で修飾します。

```
int main()
{
  《省略》

  // 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 constexpr 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 constexpr 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とき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。そのとき、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るので、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
bool 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。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 14、図 15) 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 49)。

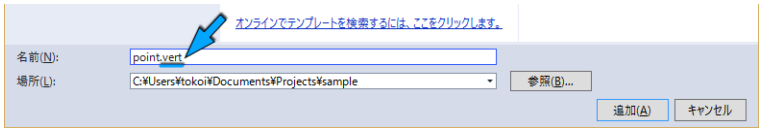


図 49 Visual Studio におけるシェーダのソースファイルのファイル名の指定

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 50)。テキストファイルを作成するので、“OS X” の “Other” にある “Empty” を選んで、“Next” をクリックします (図 51)。

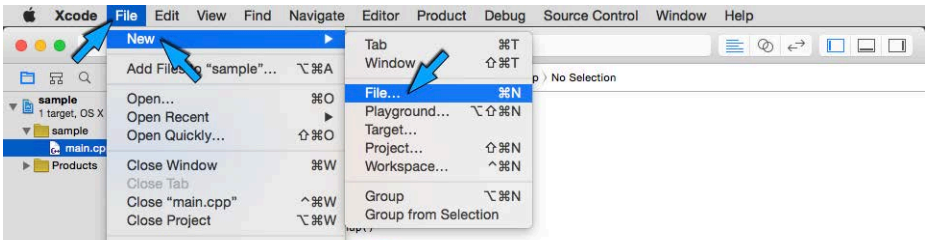


図 50 Xcode でのファイルの新規作成

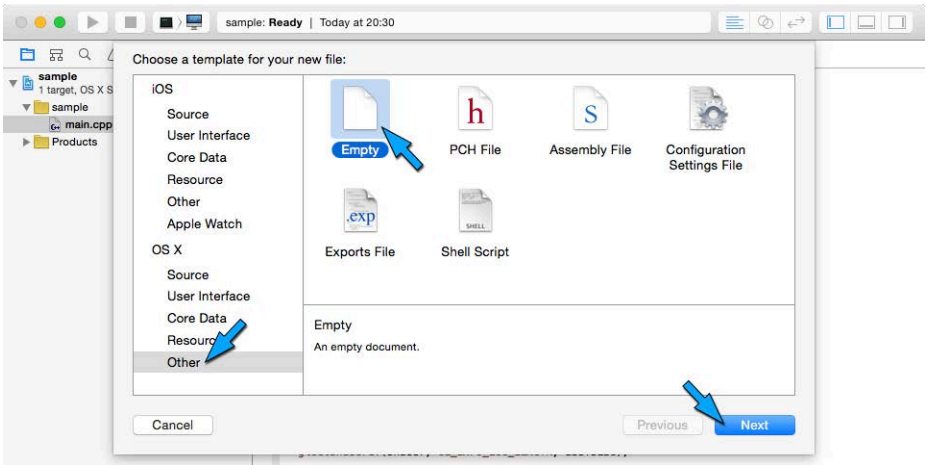


図 51 Xcode での空のファイルの作成

“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 52)。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。

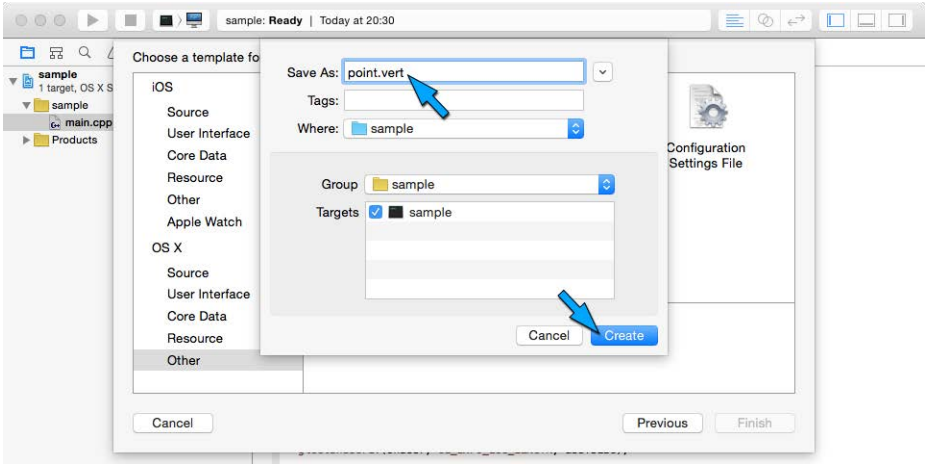


図 52 Xcode でのシェーダのソースファイルのファイル名の指定

補足：Xcode の作業ディレクトリの設定

Xcode を使ってビルドした場合は、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 内で実行するとき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 53)。

図 53 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るので、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。この欄に $PROJECT_DIR を設定すれば、プロジェクトのフォルダを移動しても、常に Xcode のプロジェクトファイルのあるフォルダが作業ディレクトリとして使用されます。最後に “OK” をクリックしてください (図 54)。

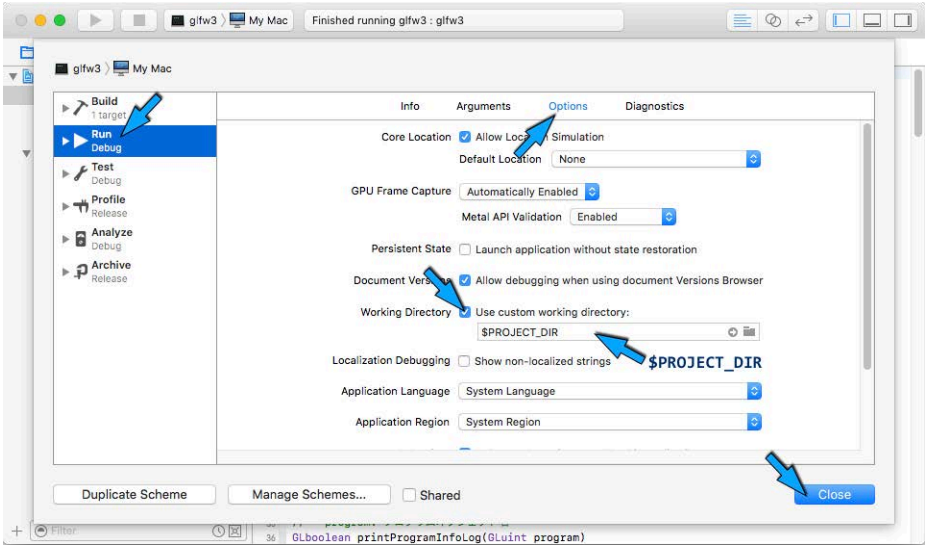


図 54 作業ディレクトリ (Working Directory) の設定

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 constexpr 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 型の頂点のデータが 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 (番号)。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。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 (番号)。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 (Object.h)

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。このメンバ変数には、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。なお #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
#include <GL/glew.h>

// 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、それに頂点のデータ 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, GLsizei 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
    glDeleteVertexArrays(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に、そのうちのどれか一つを削除すると、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るほか、残りのインスタンスを削除するときに既に削除されたバッファオブジェクトを削除しようとして、OpenGL のエラーを引き起こしてしまいます。そこ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 (コピーしようとするとコンパイルエラーになります)。

```
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 55) 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 55)。

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。

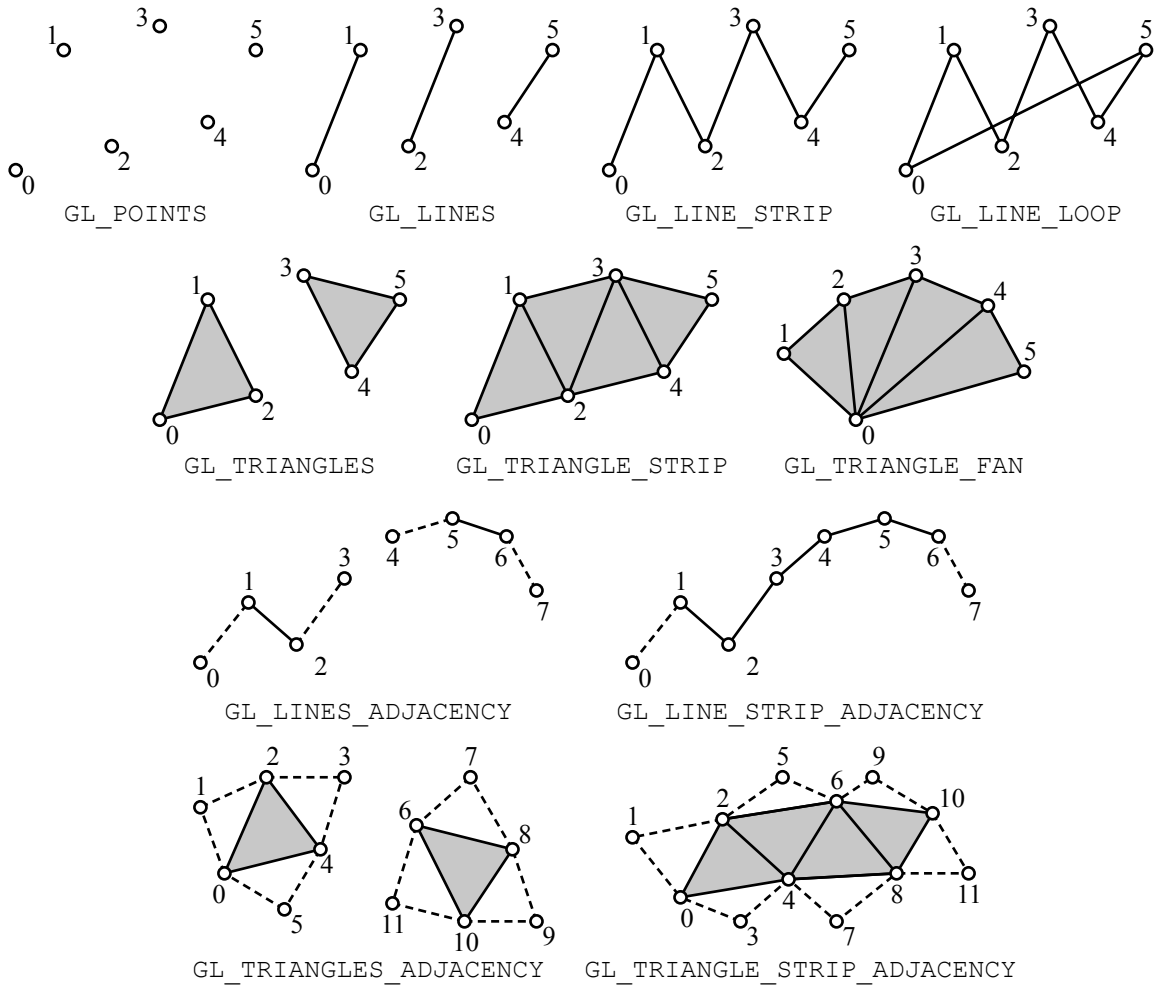


図 55 OpenGL の基本図形

● 図形の描画を行うクラス Shape (Shape.h)

実際に図形の描画を行う Shape クラスを、Shape.h というファイルに定義します。このクラスは描画する Object クラスのインスタンス (図形データ) を指すポインタを保持します。このポインタを保持するメンバ変数 object はスマートポインタ shared_ptr にします。

object を shared_ptr にしておけば、同じ Object クラスのインスタンスを参照している Shape クラスのインスタンスがすべて削除されたときに、そのインスタンス自体が削除されます (図 56)。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。これを使うために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、描画のとき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。これは変更することがないので、const にします。また、このメンバは派生クラスからも参照するので、protected にしておきます。

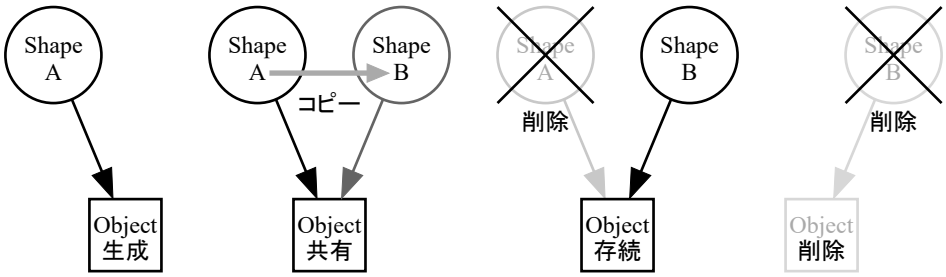


図 56 shared_ptr の動作

```
#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
  const GLsizei vertexcount;
```

このコンストラクタでは、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。また、vertexcount メンバは const なので、ここで初期化しておく必要があります。なお、このコンストラクタに本体はありません。

```
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ときに、それが指すインスタンスも自動的に削除 (delete) してくれます。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、main() 関数より前に矩形の頂点の位置データ rectangleVertex を追加します。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
constexpr 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

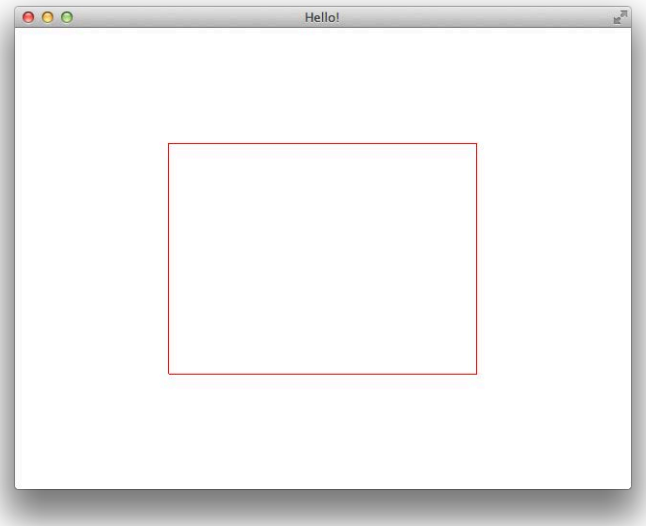


図 57 矩形の表示

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。

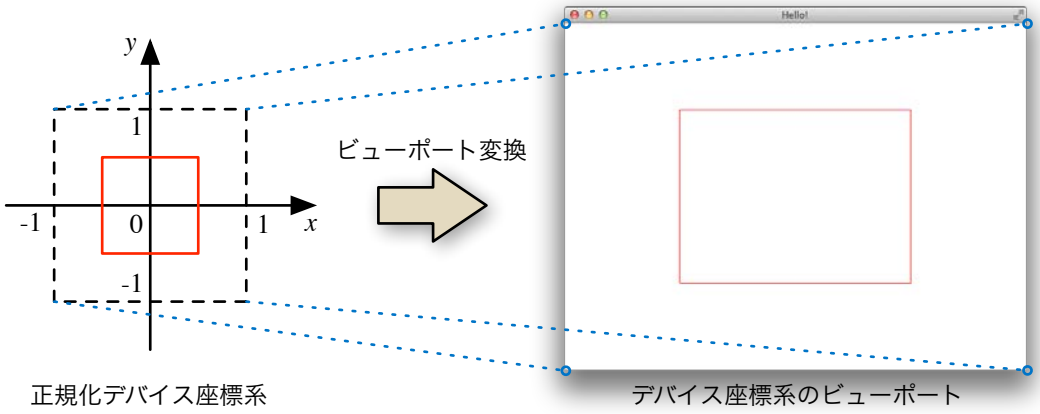


図 58 図形データと描画された図形

これは図 58 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 59)。

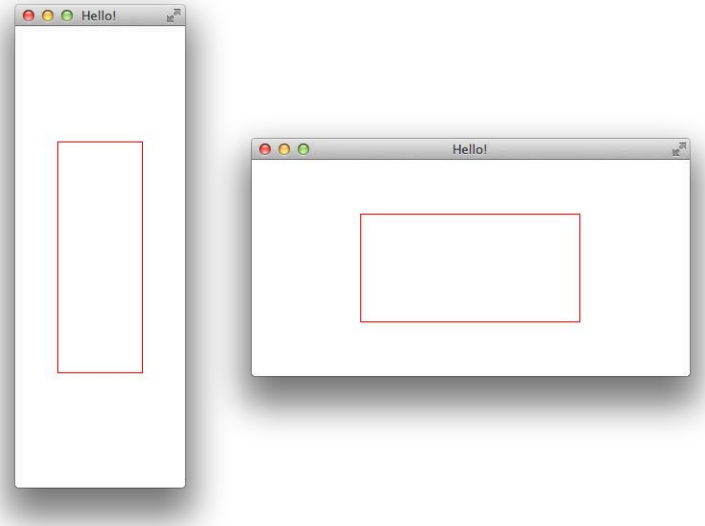


図 59 ウィンドウのサイズ変更による描画された図形の変形

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 60 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。

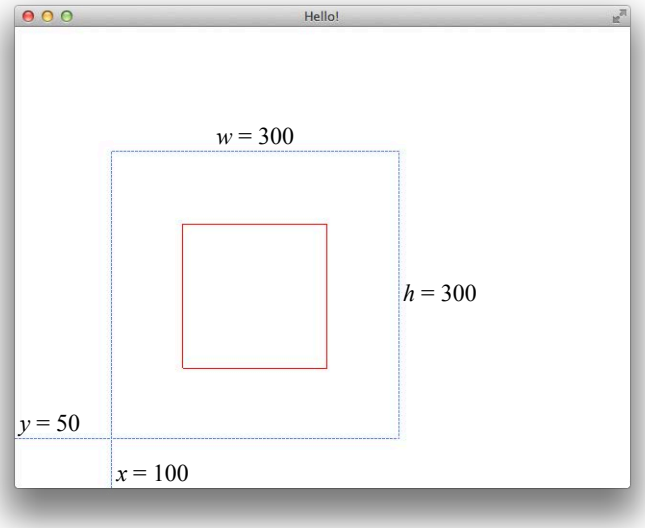


図 60 ビューポートの設定

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
constexpr 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 61 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。

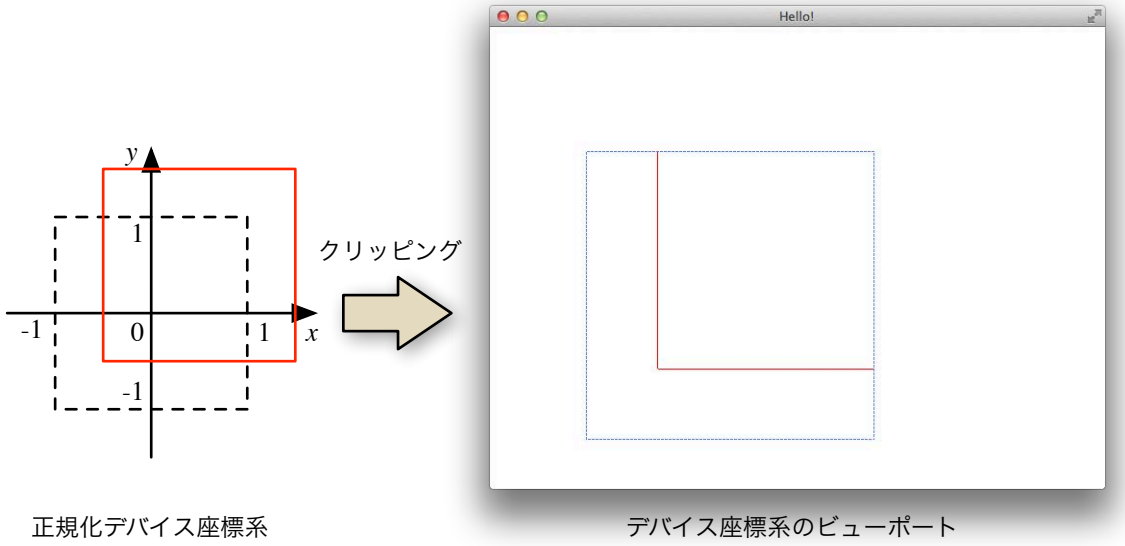


図 61 クリッピング

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 (Window.h)

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

```
// 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。また、このクラスの bool 演算子をオーバーライド (operator bool() 関数を定義) して、glfwWaitEvents() 関数でイベントを取り出した後、glfwWindowShouldClose() 関数に論理否定演算子 (!) を付けて戻り値を反転し，ウィンドウを閉じる必要がなければ true を返すようにします。こうすると、このクラスのインスタンス自体を使って描画ループの継続判定を行うことができます。このほか、バッファを入れ替えてイベントを取り出すメンバ関数 swapBuffers() も定義しておきます。

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

```
  // 描画ループの継続判定
  explicit operator bool()
  {
    // イベントを取り出す
    glfwWaitEvents();

    // ウィンドウを閉じる必要がなければ true を返す
    return !glfwWindowShouldClose(window);
  }

  // ダブルバッファリング
  void swapBuffers() const
  {
    // カラーバッファを入れ替える
    glfwSwapBuffers(window);
  }
};
```

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。

● ウィンドウのサイズ変更時に実行する関数の登録

次に、ウィンドウのサイズを変更した時に実行するコールバック関数を登録します。実行する関数名を、ここでは resize() とします。以下に Window.h の変更点を示します。

コンストラクタで glfwSetWindowSizeCallback() 関数を実行して、関数 resize() 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。

この関数で行うビューポートの設定などの処理は、通常は最初に 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);
  }
```

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。

コールバック関数としてメンバ関数を使う場合は、静的メンバ関数である必要があります。そのため、関数 resize() を静的メンバ関数として追加します。

resize() が呼び出された時には、引数 width と height にサイズ変更後のウィンドウのサイズが与えられます。しかし、glVierport() で設定するビューポートのサイズには、フレームバッファのサイズを用いる必要があります。そこで、glfwGetFramebufferSize() 関数を使って、現在のフレームバッファのサイズを調べて、これを glViewport() に指定します[7]。

```
// ダブルバッファリング
void swapBuffers()
{
  // カラーバッファを入れ替える
  glfwSwapBuffers(window);
}

// 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)
{
```

[7] フレームバッファのサイズは通常ウィンドウのサイズと一致していることが多いのですが、Retina ディスプレイを搭載した Mac では、これらが異なっている場合があります。なお、glfwWindowSizeSizeCallback() の代わりに glfwFramebufferSizeSizeCallback() を用いれば、フレームバッファのサイズが変更された時に呼び出す関数を指定できます。これによって呼び出された関数の引数 width と height には、フレームバッファのサイズ (画素数) が与えられます。また、画面上のウィンドウのサイズを調べるには、glfwGetWindowSize() が使います。

```
    // フレームバッファのサイズを調べる
    int fbWidth, fbHeight;
    glfwGetFramebufferSize(window, &fbWidth, &fbHeight);

    // フレームバッファ全体をビューポートに設定する
    glViewport(0, 0, fbWidth, fbHeight);
  }
};
```

void glfwGetFramebufferSize (GLFWwindow *const window, int *width, int *height)

指定されたウィンドウに割り当てられているフレームバッファのサイズを調べます。

window

フレームバッファのサイズを調べるウィンドウのハンドル。

width, height

調べたフレームバッファのサイズの、それぞれ幅と高さを格納する先のポインタ。NULL なら何も格納しません。

● 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の生成に置き換えます。このインスタンスを window とします。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るので、この部分からは削除します。また、ループの継続条件をこの window に置き換え、ダブルバッファリングの処理をメンバ関数の 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)
  {
    // 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
    window.swapBuffers();
  }
}
```

■ サンプルプログラム step06

6.1.4 表示図形の縦横比の固定

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、高さを基準にしたとき、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 62)。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。

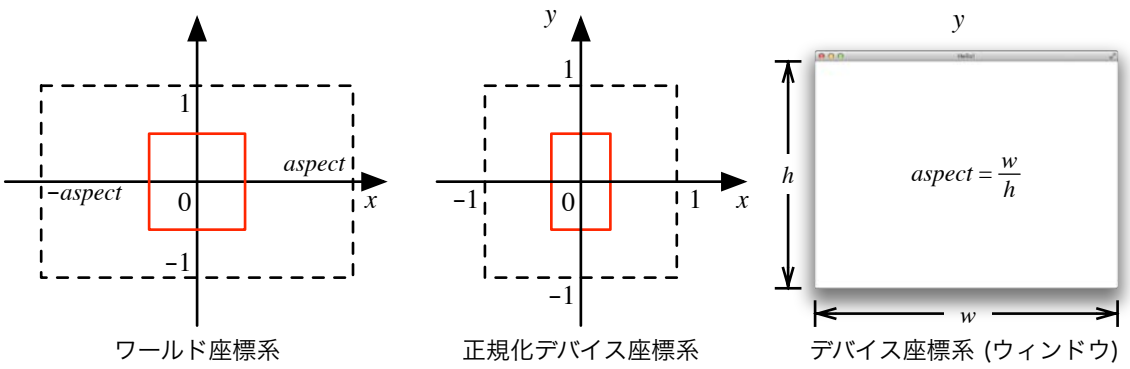


図 62 ウィンドウの縦横比 *aspect* をワールド座標に反映

● 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、aspect は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、最初に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) とき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。

そこで、コンストラクタで this の値、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 glfwSetWindowUserPointer() 関数によりウィンドウのハンドルに対応づけて記録しておきます。

```
// 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);

   《省略》
  }
```

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。

一方コールバック関数として起動された静的メンバ関数では、引数に与えられた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 の値を取り出します。インスタンスのメンバ変数には、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。

```
  // ウィンドウのサイズ変更時の処理
  static void resize(GLFWwindow *const window, int width, int height)
  {
    // フレームバッファのサイズを調べる
    int fbWidth, fbHeight;
    glfwGetFramebufferSize(window, &fbWidth, &fbHeight);

    // フレームバッファ全体をビューポートに設定する
    glViewport(0, 0, fbWidth, fbHeight);

    // 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);
    }
  }

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

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 (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 57 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。この乗算を省略して、gl_Position = vec4(position.x / aspect, position.yzw); と書くこともできます。

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

● 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 (番号) 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
  const 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)
  {
    // 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
    shape->draw();

    // カラーバッファを入れ替える
    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 (番号)。

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。他に 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 変数の場所。

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 63)。

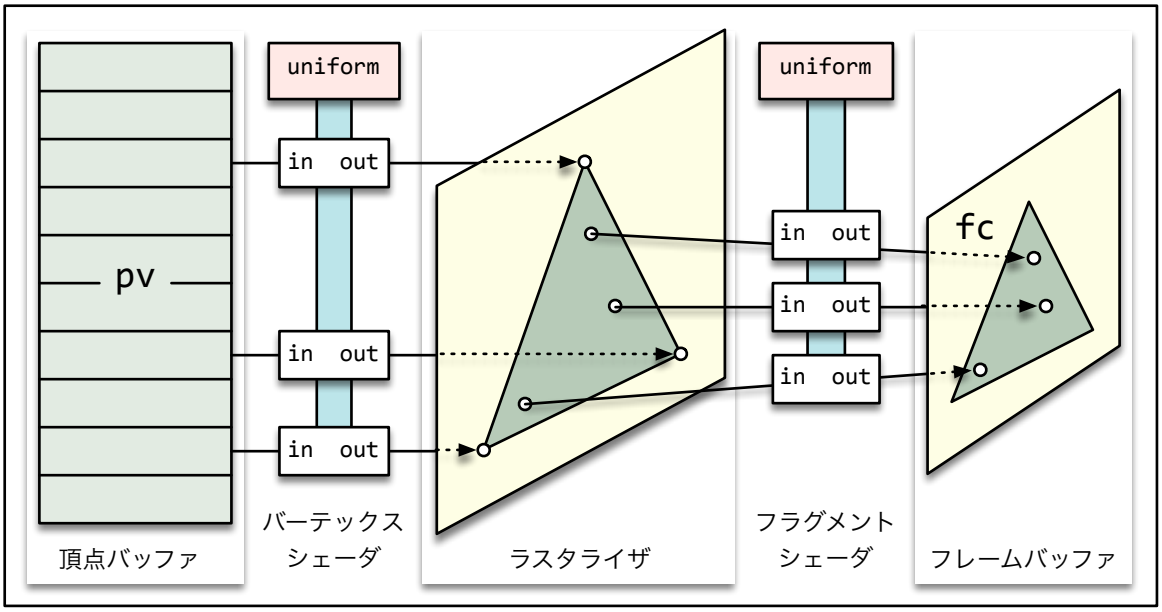


図 63 in 変数と uniform 変数

6.1.5 表示図形のサイズの固定

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 64)。これはワールド座標系からデバイス座標系に投影する際の拡大率です。

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 $\pm w/2s$、$\pm h/2s$ の範囲です (図 64)。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、ワールド座標系から正規化デバイス座標系に投影する際の拡大率は、これらの逆数になります。

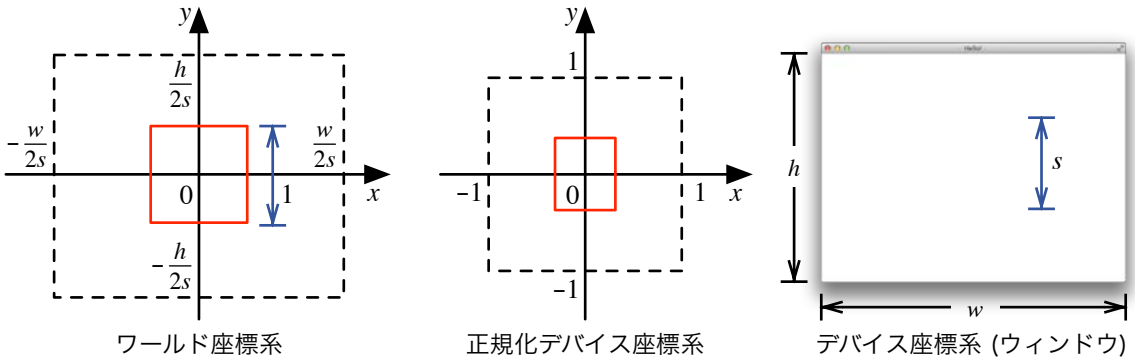


図 64 ワールド座標系に対するデバイス座標系の拡大率 s

● Window クラス (Window.h) の変更点

このスケーリングのために、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、縦横比 aspect の代わりに 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 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。また、ウィンドウのサイズはウィンドウのサイズの変更時に呼び出される resize() の引数で得られますから、そこでメンバ変数 size に格納します。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、size の初期値はそこで設定されます。このほか、size のポインタを取り出すメンバ関数 getSize() や、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》

    // 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変更時の処理
  static void resize(GLFWwindow *const window, int width, int height)
  {
    // フレームバッファのサイズを調べる
    int fbWidth, fbHeight;
```

```
    glfwGetFramebufferSize(window, &fbWidth, &fbHeight);

    // フレームバッファ全体をビューポートに設定する
    glViewport(0, 0, fbWidth, fbHeight);

    // 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);
    }
  }

  // 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; }
};
```

● バーテックスシェーダ (point.vert) の変更点

ウィンドウの縦横比 aspect (y 方向に対する x 方向の拡大率) と y 方向の拡大率としていた 1 を、それぞれ拡大率 *s* の 2 倍をウィンドウの幅 *w* で割ったものとウィンドウの高さ *h* で割ったものに置き換えます (図 64)。*s* は float 型の uniform 変数 scale に格納されています。また *w* と *h* は、それぞれ vec2 型の uniform 変数 size の x 成分 size.x と y 成分 size.y に格納されています。したがって、この除算は vec2(2.0 * scale / size.x, 2.0 * scale / size.y) となりますが、これは 2.0 * scale / size と書くことができます。これに z 成分と w 成分の拡大率 1 を補った vec4 型のベクトル vec4(2.0 * scale / size, 1.0, 1.0) が、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になります。

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

● 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》

  // 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)
  {
    // 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
    shape->draw();

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

■ サンプルプログラム step08

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。他に 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。

● 実行結果

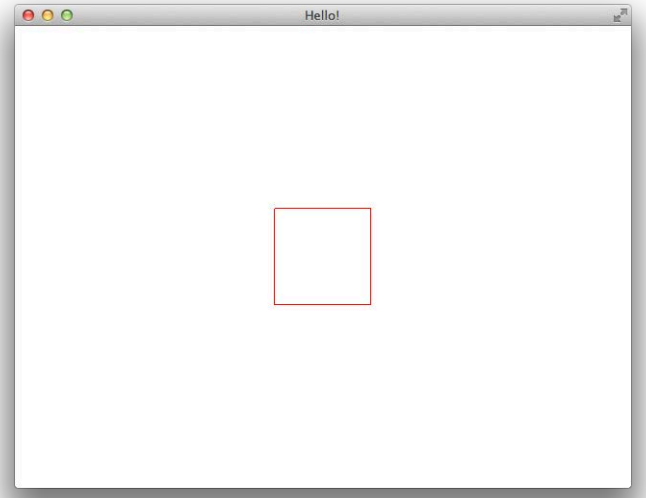


図 65 ウィンドウのサイズ変更に対して表示図形のサイズを固定

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 クラスに図形の正規化デバイス座標系上の位置を保持する配列のメンバ変数 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:
```

コンストラクタで location の各要素に初期値として 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), location{ 0.0f, 0.0f }
  {
    《省略》
  }
```

マウスの移動などで発生したイベントは bool 演算子で 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 64)。

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。

```
  // 描画ループの継続判定
  explicit operator bool()
  {
    // 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];

    // ウィンドウを閉じる必要がなければ true を返す
    return !glfwWindowShouldClose(window);
  }
```

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。

```
  // 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》
  }

  // 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; }

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

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ときは 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 (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 = vec4(2.0 * scale / size, 1.0, 1.0) * position
    + 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と、glUniform2fv() 関数によりその index にメンバ変数 location の値を設定します。これも要素数が 2 の配列を vec2 型の uniform 変数に設定するので値の設定には glUniform2fv() を用い、データはポインタで指定します。

```
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)
  {
    // 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
    shape->draw();

    // カラーバッファを入れ替える
    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。これを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ときは GLFW_PRESS (1)、押されていないときは 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》

  // 描画ループの継続判定
  explicit operator bool()
  {
    // 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];
    }

    // ウィンドウを閉じる必要がなければ true を返す
    return !glfwWindowShouldClose(window);
  }

  《省略》
};
```

■ サンプルプログラム step10

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。

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、図 64 の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* はメンバ変数 scale に格納しているので、これをマウスホイールによって変化させます。そのために、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), location{ 0.0f, 0.0f }
  {
    《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);
  }
```

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() と同様に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。ここではメンバ変数 scale に y を累積します。

なお、この引数の 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》
  }

  // 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->scale += static_cast<GLfloat>(y);
    }
  }

  《省略》
};
```

■ 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 クラスの bool 演算子の定義において、でウィンドウを閉じるべきかどうかの判定に加えて、ESC キーが押されていないかどうかの判定を追加します。ESC キーが押されていないときに true なるよう glfwGetKey(window, GLFW_KEY_ESCAPE) の戻り値を反転し、ウィンドウを閉じるべきではなく、かつ、ESC キーも押されていないときに true を返すようにします。

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

  // 描画ループの継続判定
  explicit operator bool()
  {
    // イベントを取り出す
    glfwWaitEvents();

    《省略》

    // ウィンドウを閉じる必要がなければ true を返す
    return !glfwWindowShouldClose(window) && !glfwGetKey(window, GLFW_KEY_ESCAPE);
  }
```

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

正規化デバイス座標系のクリッピング空間の幅は 2、高さは 2 であり、この領域内に描かれた図形が画面上の幅 w 画素、高さ h 画素の表示領域 (ビューポート) に描かれます (図 64)。したがって、画面上の 1 画素の長さのワールド座標系での長さは、幅方向が $2/w$、高さ方向が $2/h$ になります。w と h はメンバの配列変数 size に格納されています。

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。キーが押されている間 1 フレームあたり 1 画素動かすのであれば、押されているキーにしたがって location[0] を 2.0f / size[0] だけ増減するか、location[1] を 2.0f / size[1] だけ増減します。

```
// 描画ループの継続判定
explicit operator bool()
{
```

```
    // 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] -= 2.0f / size[0];
    else if (glfwGetKey(window, GLFW_KEY_RIGHT) != GLFW_RELEASE)
      location[0] += 2.0f / size[0];
    if (glfwGetKey(window, GLFW_KEY_DOWN) != GLFW_RELEASE)
      location[1] -= 2.0f / size[1];
    else if (glfwGetKey(window, GLFW_KEY_UP) != GLFW_RELEASE)
      location[1] += 2.0f / size[1];

    《省略》
  }

  《省略》
};
```

● 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とき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。

それでは困る場合は、glfwWaitEvents() の代わりにイベントの発生を待たない glfwPollEvents() を用います。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》

  // 描画ループの継続判定
  explicit operator bool()
  {
    // 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] -= 2.0f / size[0];
    else if (glfwGetKey(window, GLFW_KEY_RIGHT) != GLFW_RELEASE)
      location[0] += 2.0f / size[0];
    if (glfwGetKey(window, GLFW_KEY_DOWN) != GLFW_RELEASE)
      location[1] -= 2.0f / size[1];
    else if (glfwGetKey(window, GLFW_KEY_UP) != GLFW_RELEASE)
```

```
      location[1] += 2.0f / size[1];

    《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、他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ときは glfwWaitEvents() で行うようにします。glfwSetKeyCallback() 関数を用いれば、任意のキーを操作したときに実行するコールバック関数を指定できるので、その中でこの切り替えを行います。

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》

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

  // 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), location{ 0.0f, 0.0f }, 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。

```
  // 描画ループの継続判定
  explicit operator bool()
  {
```

```
    // 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);
  }
```

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。この値はプラットフォ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。

```
  // 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;
    }
  }

  《省略》
};
```

■ サンプルプログラム step12

第7章 座標変換

7.1 頂点処理

7.1.1 図形表示の手順

OpenGL による図形表示では、正規化デバイス座標系上に設定されているクリッピング空間からはみ出た部分はクリッピングされ、画面には表示されません。したがって、任意の大きさの図形を表示するには、その図形をクリッピング空間に収まるように拡大縮小と平行移動を行う必要があります。第 6 章では、描画しようとする図形が単純な二次元図形であることから、これを場当たり的な方法で行っていました。

しかし、描画しようとする図形が複雑な構造を持っていたりすると、このようなアプローチではすぐに訳が分からなくなってしまいそうです。それに OpenGL は、本来は三次元図形の表示を行う API です。三次元の図形を二次元の空間である画面上に表示するには、さらに投影という処理も必要になります。

この投影を含めた三次元図形表示については、既に標準的な手順が考えられています。一般にグラフィックス表示の対象となる図形のデータは、複数の「部品」で構成されています。これを空間中に「配置」することによって、シーンを構成します (図 66)。この「部品の配置」のための座標変換を**モデル変換**といいます。次に、部品が配置されたシーンをカメラの位置から見たときの座標に変換します。これを**ビュー変換**といいます。そして最後に投影変換を行って、シーンをスクリーンの正規化デバイス座標系上に投影します。

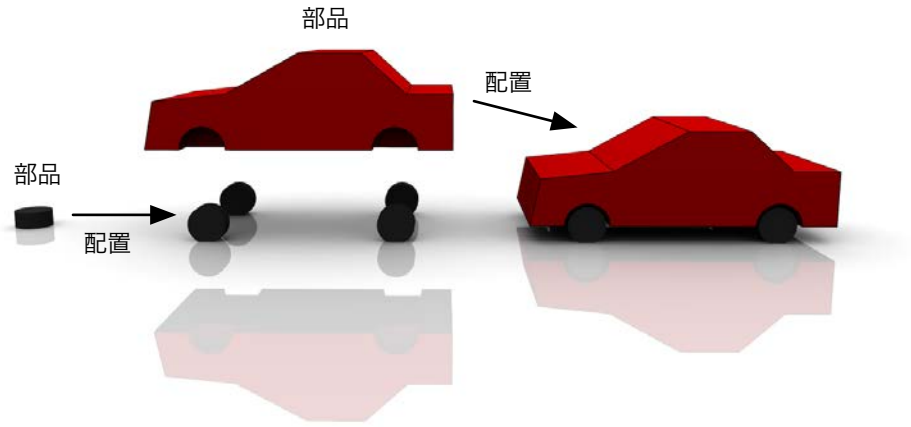


図 66 部品の配置

このように、座標変換はモデル変換→ビュー変換→投影変換の順に行われるので、これも一つの論理的なパイプラインと見なすことができます。このため、これを**ジオメトリパイプライン**と呼ぶことがあります。かつては、この処理をジオメトリエンジンと名付けられた専用のハードウェアで行っていました (さらにその前は CPU によるソフトウェア処理でした)。

現在の GPU では、ジオメトリパイプラインをバーテックスシェーダ上にシェーダプログラムとして実装します。バーテックスシェーダは頂点バッファオブジェクトに格納されている頂点属性を受け取り、これらの座標変換により正規化座標系上の座標値を求めて、それを次のステージに送ります。

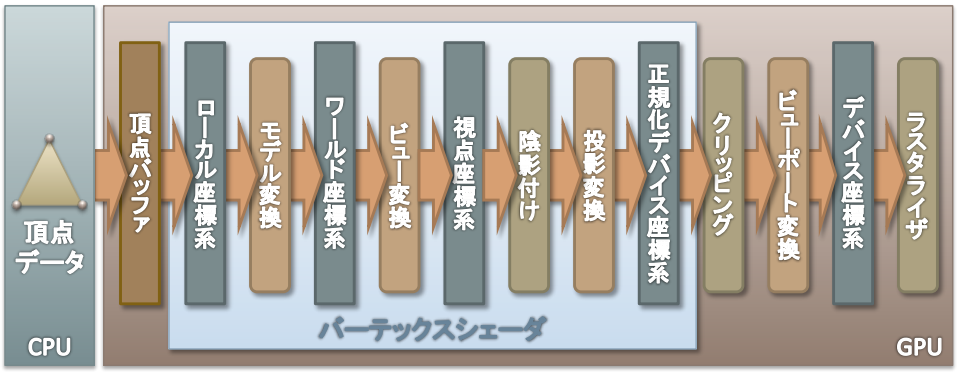


図 67 頂点処理のパイプライン

7.1.2 モデル変換

シーンを構成する部品 (オブジェクトと呼ばれます) は、個々に独自の座標系で定義されています。この座標系を**ローカル座標系**といいます。シーンを構成するには、この部品のほか、視点や光源などを、ワールド座標系と呼ばれる単一の座標系に配置します。モデル変換は、このローカル座標系からワールド座標系への座標変換を行います (図 68)。

モデル変換は部品ごとに設定します。部品間に骨格のような階層構造があれば、この変換も階層的に合成して実行します。モデル変換後はすべての部品がワールド座標系上に存在します。

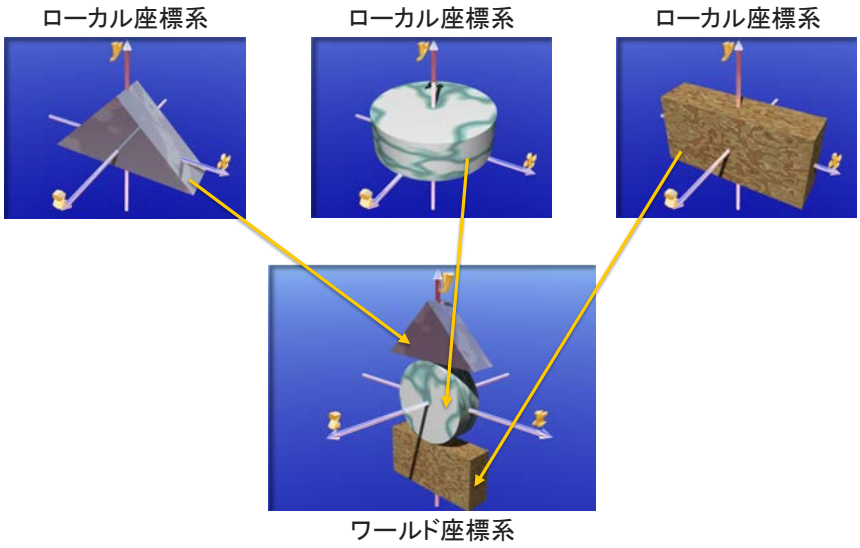


図 68 モデル変換

7.1.3 ビュー変換

ワールド座標系で構築されたシーンの視点位置から見た映像を作成するために、視点を基準とする視点座標系にシーン全体を座標変換します。ビュー変換はこのワールド座標系から視点座標系への変換を行います (図 69)。なお、モデル変換とビュー変換は通常一つの座標変換に合成されます。この合成変換を**モデルビュー変換**といいます。

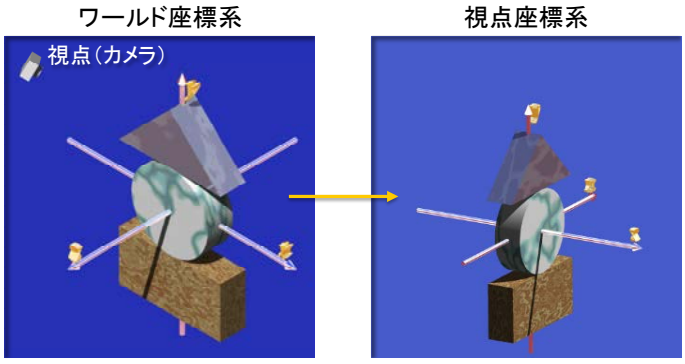


図 69 ビュー変換

図 70 にローカル座標系、ワールド座標系、視点座標系、およびスクリーンの関係を示します。

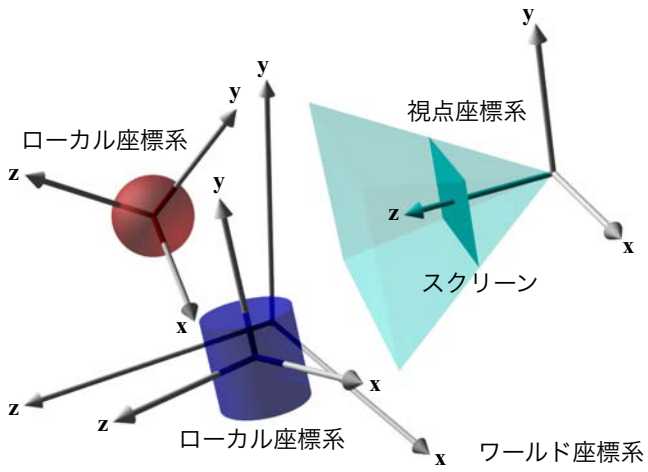


図 70 各座標系の関係

7.1.4 投影変換

視点座標系に配置されたシーンをスクリーンに投影する変換を**投影変換** (Projection) といいます。ディスプレイの表示領域は有限ですから、そこに映るシーンの空間も限定されます。この空間を**視体積** (View Volume) といいます。投影変換は、この空間を、中心が原点にあり一辺の長さが 2 の立方体の空間である**標準視体積** (Canonical View Volume) に変形します (図 71)。これは三次元のクリッピング領域であり、**クリッピング空間** (Clipping Volume) とも呼ばれます。

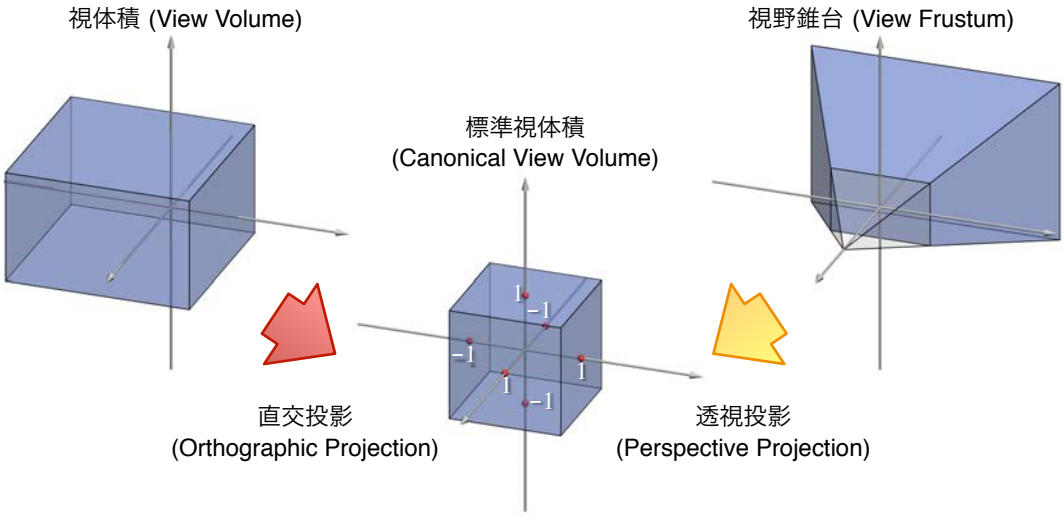


図 71 投影変換

標準視体積への変換は、変換前の視体積に直方体を用いる場合と、四角錐台を用いる場合の二通りがあります。直方体を用いれば**直交投影** (Orthogonal Projection) となります。一方、四角錐台を用いれば**透視投影** (Perspective Projection) となります (図 72)。なお、この四角錐台の視体積は、特に**視錐台** (View Frustum) と呼ばれます。

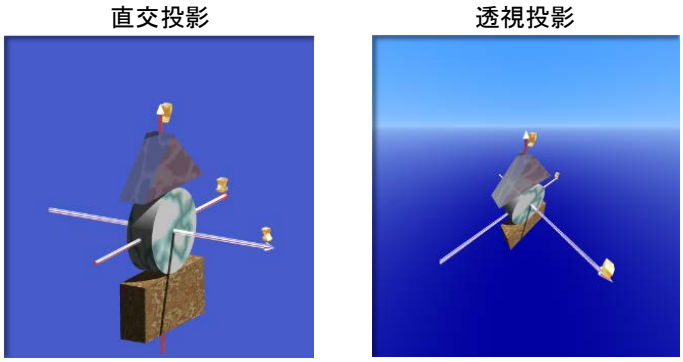


図 72 直交投影と透視投影

7.2 同次座標

7.2.1 アフィン変換

第 6 章で行っていた拡大縮小などの線形変換と平行移動の組み合わせは、一般にアフィン変換と呼ばれます。一次元のアフィン変換は次のように表すことができます。

$$x' = ax + b \tag{1}$$

これは x を a 倍して、b だけ平行移動します。同様に、二次元、三次元の場合は、それぞれ次のようになります。

$$\begin{aligned} x' &= a_{xx}x + a_{yx}y + b_x \\ y' &= a_{xy}x + a_{yy}y + b_y \end{aligned} \tag{2}$$

$$\begin{aligned} x' &= a_{xx}x + a_{yx}y + a_{zx}z + b_x \\ y' &= a_{xy}x + a_{yy}y + a_{zy}z + b_y \\ z' &= a_{xz}x + a_{yz}y + a_{zz}z + b_z \end{aligned} \tag{3}$$

行列を使うと、式 (3) を次のように書くことができます。

$$\begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} = \begin{pmatrix} a_{xx} & a_{yx} & a_{zx} \\ a_{xy} & a_{yy} & a_{zy} \\ a_{xz} & a_{yz} & a_{zz} \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} + \begin{pmatrix} b_x \\ b_y \\ b_z \end{pmatrix} \tag{4}$$

7.2.2 同次座標の導入

アフィン変換は式 (4) に示すように、線形変換の結果を平行移動したものとして表すことができます。しかし、積と和の組み合わせでは、複数のアフィン変換の合成変換を表すことが面倒になります。そこで、x, y, z にもう一つ数を追加して、次のように表すことを考えてみます。

$$\begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \begin{pmatrix} a_{xx} & a_{yx} & a_{zx} & b_x \\ a_{xy} & a_{yy} & a_{zy} & b_y \\ a_{xz} & a_{yz} & a_{zz} & b_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} \tag{5}$$

この x',y',z' も式 (3) と同じものになります。このように表せば、変換の合成を行列の積のみで表すことができます。

同次座標 (斉次座標) は、式 (5) のように、表現しようとする空間の次元より一つ多い数の組で座標 (点の位置) を表すものです。二次元なら三つの数、三次元なら四つの数で座標を表します。通常の三次元の座標 (以降は**標準座標**と呼ぶことにします) を (x^*, y^*, z^*) と表すとき、それと、その同次座標 (x, y, z, w) との間には、同次座標の定義により次の関係があります。

$$\begin{aligned} x^* &= \frac{x}{w} \\ y^* &= \frac{y}{w} \\ z^* &= \frac{z}{w} \end{aligned} \tag{6}$$

したがって、標準座標が (x, y, z) であれば、その同次座標は $(x, y, z, 1)$ となります。また、同次座標の四つ目の要素 w が 0 に近づけば、その標準座標は (x, y, z) 方向の無限遠に向かって移動します。同次座標 $(x, y, z, 0)$ は (x, y, z) の方向の無限遠点です。

$$\begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} \Rightarrow (x, y, z) \text{の位置} \tag{7}$$

$$\begin{pmatrix} x \\ y \\ z \\ 0 \end{pmatrix} \Rightarrow (x, y, z) \text{の無限遠点} \tag{8}$$

同次座標にスカラー値を掛けても、標準座標は変わりません。

$$a \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix} = \begin{pmatrix} ax \\ ay \\ az \\ aw \end{pmatrix} \Rightarrow \left(\frac{ax}{aw}, \frac{ay}{aw}, \frac{az}{aw}\right) = \left(\frac{x}{w}, \frac{y}{w}, \frac{z}{w}\right) \tag{9}$$

CG では、ある点から別の点に向かう方向単位ベクトルを求めなければならないことがよくあります。いま、二点 $\mathbf{P}_0, \mathbf{P}_1$ の位置をそれぞれ同次座標 $\mathbf{P}_0 = (x_0, y_0, z_0, w_0)$, $\mathbf{P}_1 = (x_1, y_1, z_1, w_1)$ で表すとき、$\mathbf{P}_0$ から $\mathbf{P}_1$ に向かうベクトルは、それぞれの標準座標を求めて差を求めます。

$$\frac{\mathbf{P}_1}{w_1} - \frac{\mathbf{P}_0}{w_0} = \begin{pmatrix} x_1/w_1 \\ y_1/w_1 \\ z_1/w_1 \\ 1 \end{pmatrix} - \begin{pmatrix} x_0/w_0 \\ y_0/w_0 \\ z_0/w_0 \\ 1 \end{pmatrix} = \begin{pmatrix} x_1/w_1 - x_0/w_0 \\ y_1/w_1 - y_0/w_0 \\ z_1/w_1 - z_0/w_0 \\ 0 \end{pmatrix} \tag{10}$$

これだと w_0 か w_1 のどちらかが 0 のときに計算できないので、両辺に $w_0 w_1$ を掛けて通分します。

$$w_0\mathbf{P}_1 - w_1\mathbf{P}_0 = \begin{pmatrix} w_0x_1 \\ w_0y_1 \\ w_0z_1 \\ w_0w_1 \end{pmatrix} - \begin{pmatrix} w_1x_0 \\ w_1y_0 \\ w_1z_0 \\ w_1w_0 \end{pmatrix} = \begin{pmatrix} w_0x_1 - w_1x_0 \\ w_0y_1 - w_1y_0 \\ w_0z_1 - w_1z_0 \\ 0 \end{pmatrix} \tag{11}$$

式 (11) は引き算により四つ目の要素が 0 になるので、無限遠点となります。この計算方法では、$\mathbf{P}_0$ と $\mathbf{P}_1$ のどちらか一方の四つ目の要素が 0、すなわち無限遠にあっても場合分けを行う必要がありません。また、この計算はここまで割り算を必要としないので、処理速度の点でも有利です。これを正規化して、照明計算などに用います。

なお、同次座標 (x_i, y_i, z_i, w_i) の和は、それが表す標準座標 (x_i^*, y_i^*, z_i^*) の w_i を重みととした重み付き平均になります。

$$\sum_{i=1}^{N} \begin{pmatrix} x_i \\ y_i \\ z_i \\ w_i \end{pmatrix} = \sum_{i=1}^{N} \begin{pmatrix} w_ix_i^* \\ w_iy_i^* \\ w_iz_i^* \\ w_i \end{pmatrix} = \begin{pmatrix} \sum w_ix_i^* \\ \sum w_iy_i^* \\ \sum w_iz_i^* \\ \sum w_i \end{pmatrix} \Rightarrow \left(\frac{\sum w_ix_i^*}{\sum w_i}, \frac{\sum w_iy_i^*}{\sum w_i}, \frac{\sum w_iz_i^*}{\sum w_i} \right) \tag{12}$$

7.3 変換行列

7.3.1 同次座標の座標変換

式 (5) のアフィン変換では、ベクトルや行列の一部に 0 や 1 の定数が入っていました。ここではより一般的な同次座標による座標変換について考えてみます。同次座標で表された点の位置 $\mathbf{v}$ の変換行列 $\mathbf{M}$ による座標変換 $\mathbf{v}' = \mathbf{M}\mathbf{v}$ は、式 (16) のように表すことができます。

$$\mathbf{v} = \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix} \tag{13}$$

$$\mathbf{v}' = \begin{pmatrix} x' \\ y' \\ z' \\ w' \end{pmatrix} \tag{14}$$

$$\mathbf{M} = \begin{pmatrix} m_0 & m_4 & m_8 & m_{12} \\ m_1 & m_5 & m_9 & m_{13} \\ m_2 & m_6 & m_{10} & m_{14} \\ m_3 & m_7 & m_{11} & m_{15} \end{pmatrix} \tag{15}$$

$$\begin{pmatrix} x' \\ y' \\ z' \\ w' \end{pmatrix} = \begin{pmatrix} m_0 & m_4 & m_8 & m_{12} \\ m_1 & m_5 & m_9 & m_{13} \\ m_2 & m_6 & m_{10} & m_{14} \\ m_3 & m_7 & m_{11} & m_{15} \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix} \tag{16}$$

このベクトル $\mathbf{v}'$ の各要素 (x', y', z', w') は次式により求められます。

$$
\begin{array}{rcl}
x' &=& m_0\,x + m_4\,y + m_8\,z + m_{12}\,w \\
y' &=& m_1\,x + m_5\,y + m_9\,z + m_{13}\,w \\
z' &=& m_2\,x + m_6\,y + m_{10}\,z + m_{14}\,w \\
w' &=& m_3\,x + m_7\,y + m_{11}\,z + m_{15}\,w
\end{array}
\tag{17}
$$

したがって、ベクトル **v**' の各要素は、行列 **M** の各行を要素とするベクトルと **v** の内積になります。たとえば、$x' = (m_0\ m_4\ m_8\ m_{12}) \cdot (x\ y\ z\ w)^T$ となります (図 73)。

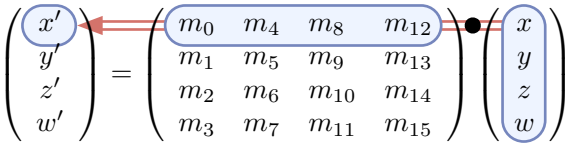


図 73 行列とベクトルの積

● OpenGL の変換行列

式 (15) に示した 4 行 4 列の変換行列は OpenGL の座標変換の基本となるもので、GPU で座標変換を行うために、CPU 側から GPU 側に頻繁に渡されます。ただし、行列を配列変数に格納する際には、配列の要素の順序は行列の要素の順序を**見かけ上**転置したものになります。

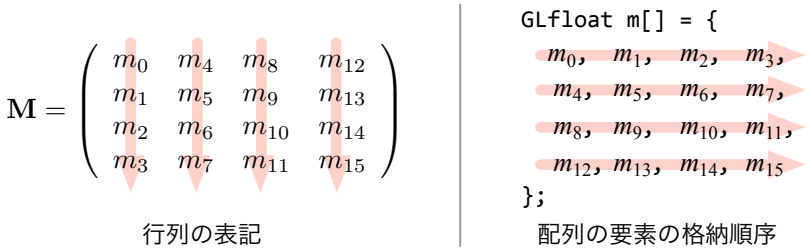


図 74 行列の表記と配列の要素の格納順序

● 変換行列のクラス Matrix (Matrix.h)

このような変換行列の取り扱いには glm (OpenGL Mathematics, https://glm.g-truc.net/) というライブラリがよく用いられますが、ここでは自分で簡単なものを作ります。

この変換行列を取り扱うクラス Matrix を、Matrix.h というヘッダファイルに定義します。標準テンプレートライブラリの copy や fill を使うので、argorithm を #include します。private メンバの配列 matrix に変換行列を保持します。

```
#pragma once
#include <algorithm>
#include <GL/glew.h>

// 変換行列
class Matrix
{
  // 変換行列の要素
  GLfloat matrix[16];
```

デフォルトコンストラクタでは何もしません。この他に、変換行列の内容を配列で初期化するためのコンストラクタを用意します。また、変換行列の要素を参照する [ ] 演算子や，内容を保持している配列のポインタを取り出す data() というメソッドも用意しておきます。

```
public:

  // コンストラクタ
  Matrix() {}

  // 配列の内容で初期化するコンストラクタ
  //   a: GLfloat 型の 16 要素の配列
  Matrix(const GLfloat *a)
  {
    std::copy(a, a + 16, matrix);
  }

  // 行列の要素を右辺値として参照する
  const GLfloat &operator[](std::size_t i) const
  {
    return matrix[i];
  }

  // 行列の要素を左辺値として参照する
  GLfloat &operator[](std::size_t i)
  {
    return matrix[i];
  }

  // 変換行列の配列を返す
  const GLfloat *data() const
  {
    return matrix;
  }
};
```

7.3.2 単位行列

単位行列は対角成分が 1、残りの成分が 0 の正方行列です。

$$\mathbf{I} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{18}$$

● 単位行列を作るメソッドの追加 (Matrix.h)

Matrix クラスに単位行列を設定する loadIdentity() というメソッドを追加します。これはメンバの配列 matrix のすべての要素を一旦 0 にし、その後に対角要素に 1 を代入します。また、それを使って単位行列を作る identity() というメソッドを追加します。これはインスタンスを生成せずに呼び出せるように static メソッドにします。

```
#pragma once
#include <algorithm>
#include <GL/glew.h>

// 変換行列
class Matrix
{
  《省略》

  // 変換行列の配列を返す
  const GLfloat *data() const
  {
    return matrix;
  }

  // 単位行列を設定する
  void loadIdentity()
  {
    std::fill(matrix, matrix + 16, 0.0f);
    matrix[ 0] = matrix[ 5] = matrix[10] = matrix[15] = 1.0f;
  }

    // 単位行列を作成する
  static Matrix identity()
  {
    Matrix t;
    t.loadIdentity();
    return t;
  }
};
```

7.3.3 平行移動

点の位置を、現在の位置から $\mathbf{t}=(t_x, t_y, t_z)$ 離れたところに平行移動する変換行列 $\mathbf{T}(t_x, t_y, t_z)$ は、次のようになります。

$$\mathbf{T}(\mathbf{t}) = \mathbf{T}(t_x, t_y, t_z) = \begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{19}$$

標準座標 (x, y, z) に対するこの変換行列による変換は、次のようになります。

$$\begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x + t_x \\ y + t_y \\ z + t_z \\ 1 \end{pmatrix} \tag{20}$$

$\mathbf{T}(7, 8, 0)$ という変換は、図 75 のような平行移動になります。

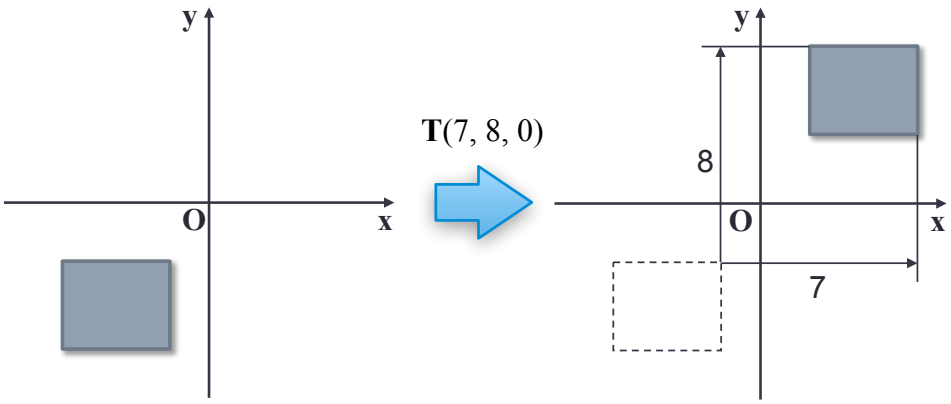


図 75 平行移動

同次座標の四つ目の要素が 0 のときは、この変換の影響を受けません。

$$\begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 0 \end{pmatrix} = \begin{pmatrix} x \\ y \\ z \\ 0 \end{pmatrix} \tag{21}$$

● 平行移動の変換行列を作るメソッドの追加 (Matrix.h)

Matrix クラスに引数 x, y, z で与えられた分だけ平行移動する変換行列を作るメソッドを追加します。平行移動の変換行列は式 (19) より式 (15) の変換行列の要素の m_{12} に x、m_{13} に y、m_{14} に z を代入したものなので、Matrix 型の変換行列を単位行列で初期化した後、メンバの配列変数 matrix のこれらの要素に値を代入します。これもインスタンスを生成せずに呼び出せるように static メソッドにします。

```
// 変換行列
class Matrix
{
  《省略》

  // 単位行列を設定する
  void loadIdentity()
  {
    《省略》
  }

  // 単位行列を作成する
  static Matrix identity()
  {
    《省略》
  }

  // (x, y, z) だけ平行移動する変換行列を作成する
  static Matrix translate(GLfloat x, GLfloat y, GLfloat z)
  {
    Matrix t;
```

```
    t.loadIdentity();
    t[12] = x;
    t[13] = y;
    t[14] = z;

    return t;
  }
};
```

7.3.4 拡大縮小

点の位置を、原点を中心に $\mathbf{s} = (s_x, s_y, s_z)$ 倍する変換行列 $\mathbf{S}(s_x, s_y, s_z)$ は、次のようになります。

$$\mathbf{S}(\mathbf{s}) = \mathbf{S}(s_x, s_y, s_z) = \begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{22}$$

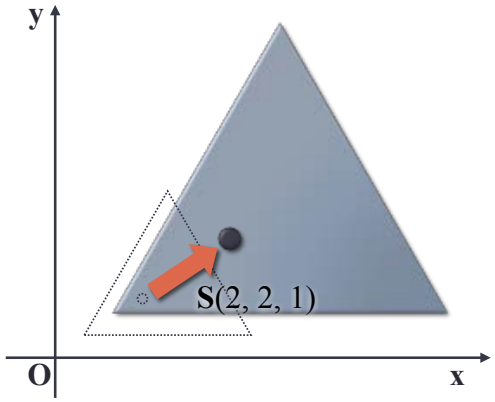


図 76 拡大縮小

拡大率を $s_x, = s_y, = s_z = a$ とした場合と w 要素の拡大率を $1/a$ とした場合とでは、標準座標は等しくなります。

$$\begin{pmatrix} a & 0 & 0 & 0 \\ 0 & a & 0 & 0 \\ 0 & 0 & a & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} ax \\ ay \\ az \\ 1 \end{pmatrix} \Rightarrow (ax,\ ay,\ az) \tag{23}$$

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1/a \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x \\ y \\ z \\ 1/a \end{pmatrix} \Rightarrow (ax,\ ay,\ az) \tag{24}$$

● 拡大縮小の変換行列を作成するメソッドの追加 (Matrix.h)

平行移動と同様に、Matrix クラスに引数 x, y, z で与えられた分だけ拡大縮小する変換行列を作るメソッドを追加します。こ拡大縮小の変換行列は、式 (22) より式 (15) の変換行列の要素の m_0 に x、m_5 に y、m_{10} に z を代入したものなので、Matrix 型の変換行列を単位行列で初期化し

た後、メンバの配列変数 matrix のこれらの要素に値を代入します。れもインスタンスを生成せずに呼び出せるように static メソッドにします。

```
// 変換行列
class Matrix
{
  《省略》

  // (x, y, z) だけ平行移動する変換行列を作成する
  static Matrix translate(GLfloat x, GLfloat y, GLfloat z)
  {
    《省略》
  }

  // (x, y, z) 倍に拡大縮小する変換行列を作成する
  static Matrix scale(GLfloat x, GLfloat y, GLfloat z)
  {
    Matrix t;

    t.loadIdentity();
    t[ 0] = x;
    t[ 5] = y;
    t[10] = z;

    return t;
  }
};
```

7.3.5 せん断

せん断は形状の異なる部分に異なる方向に力をかけたときに生じる変形のことをいいます。

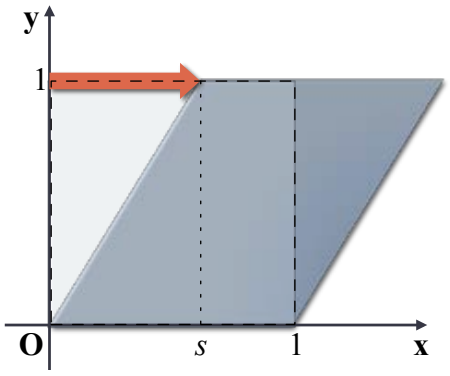


図 77 せん断

これは座標値のある軸の値に係数をかけたものを別の軸に加えることによって実現できます。図 77 では y 座標値に s をかけたものを x 座標値に足しており、式 (25) で表されます。

$$\mathbf{H}_{xy}(s) = \begin{pmatrix} 1 & s & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{25}$$

この変換は x, y, z の各軸に対してそれぞれ二つずつ定義されるので、全部で六つあります。

$$\mathbf{H}_{xy}(s) = \begin{pmatrix} 1 & s & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad \mathbf{H}_{yz}(s) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & s & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad \mathbf{H}_{zx}(s) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ s & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$\mathbf{H}_{yx}(s) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ s & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad \mathbf{H}_{zy}(s) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & s & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad \mathbf{H}_{xz}(s) = \begin{pmatrix} 1 & 0 & s & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{26}$$

これらのメソッドは、ここでは特に用意しませんが、演習として、これまでの例にならって実装してみてください。

7.3.6 回転

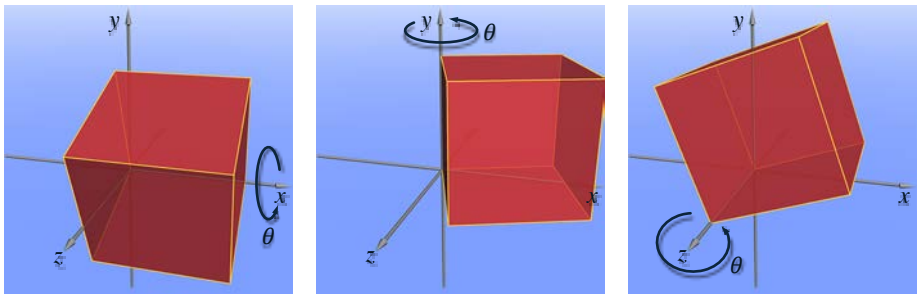
● 座標軸中心の回転

次の変換は点の位置をそれぞれ x、y、z の座標軸を中心に回転します。

$$\mathbf{R}_x(\theta) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{27}$$

$$\mathbf{R}_y(\theta) = \begin{pmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{28}$$

$$\mathbf{R}_z(\theta) = \begin{pmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{29}$$



x 軸中心の回転 $\mathbf{R}_x(\theta)$　y 軸中心の回転 $\mathbf{R}_y(\theta)$　z 軸中心の回転 $\mathbf{R}_z(\theta)$

図 78 座標軸中心の回転

これらのメソッドは、ここでは特に用意しませんが、あるとオイラー変換 (7.4.6) の実装など

に使えて便利なので、これまでの例にならって実装してみてください。

● 任意の軸中心の回転

方向余弦 (l, m, n) を軸として θ 回転する変換行列は、次式により求めることができます。

$$\mathbf{R}(l,m,n,\theta) = \begin{pmatrix} l^2+(1-l^2)\cos\theta & lm(1-\cos\theta)-n\sin\theta & ln(1-\cos\theta)+m\sin\theta & 0 \\ lm(1-\cos\theta)+n\sin\theta & m^2+(1-m^2)\cos\theta & mn(1-\cos\theta)-l\sin\theta & 0 \\ ln(1-\cos\theta)-m\sin\theta & mn(1-\cos\theta)+l\sin\theta & n^2+(1-n^2)\cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad (30)$$

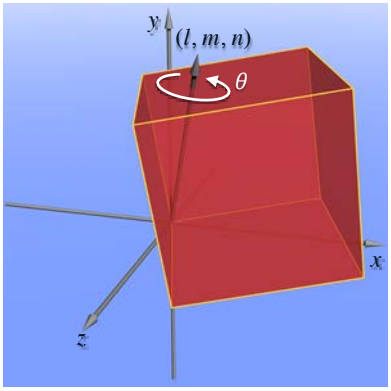


図 79 任意の軸中心の回転

● 任意の軸中心の回転の変換行列を作成するメソッドの追加 (Matrix.h)

式 (30) の変換行列を作るメソッドを Matrix クラスに追加します。このメソッドは引数 x, y, z で与えられたベクトルを軸に引数 a だけ回転する変換行列を返します。数学ライブラリ関数 sqrt() を使うので、cmath を #include します。なお、回転軸の長さが 0 の時はエラーも出さずに何も入っていない変換行列を返しますが、気になる人はエラー処理を追加してください。

```
#pragma once
#include <cmath>
#include <algorithm>
#include <GL/glew.h>

// 変換行列
class Matrix
{
  《省略》

  // (x, y, z) を軸に a 回転する変換行列を作成する
  static Matrix rotate(GLfloat a, GLfloat x, GLfloat y, GLfloat z)
  {
    Matrix t;
    const GLfloat d(sqrt(x * x + y * y + z * z));

    if (d > 0.0f)
    {
```

```
      const GLfloat l(x / d), m(y / d), n(z / d);
      const GLfloat l2(l * l), m2(m * m), n2(n * n);
      const GLfloat lm(l * m), mn(m * n), nl(n * l);
      const GLfloat c(cos(a)), c1(1.0f - c), s(sin(a));

      t.loadIdentity();
      t[ 0] = (1.0f - l2) * c + l2;
      t[ 1] = lm * c1 + n * s;
      t[ 2] = nl * c1 - m * s;
      t[ 4] = lm * c1 - n * s;
      t[ 5] = (1.0f - m2) * c + m2;
      t[ 6] = mn * c1 + l * s;
      t[ 8] = nl * c1 + m * s;
      t[ 9] = mn * c1 - l * s;
      t[10] = (1.0f - n2) * c + n2;
    }

    return t;
  }
};
```

7.4 変換の合成

7.4.1 複数の変換の組み合わせ

複数の変換の合成は、変換行列の積で表すことができます。たとえば、図 80 のように x 方向に 4 移動した後、y 方向に 3 移動する変換は、式 (31) に示す単一の変換行列で表すことができます。

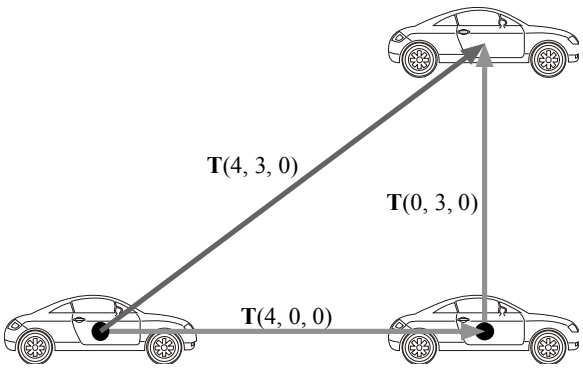


図 80 変換の合成

$$\begin{aligned}\mathbf{T}(0,3,0)\mathbf{T}(4,0,0) &= \begin{pmatrix}1&0&0&0\\0&1&0&3\\0&0&1&0\\0&0&0&1\end{pmatrix}\begin{pmatrix}1&0&0&4\\0&1&0&0\\0&0&1&0\\0&0&0&1\end{pmatrix} \\ &= \begin{pmatrix}1&0&0&4\\0&1&0&3\\0&0&1&0\\0&0&0&1\end{pmatrix} = \mathbf{T}(4,3,0)\end{aligned} \tag{31}$$

● 行列の積を求める演算子

Matrix クラスにおいて積の演算子 * を次のようにオーバーライドして、変換の合成に用いる行列の積を求めるようにします。なお、この行列の各要素の乗算の順序は、OpenGL では行列が配列変数に見かけ上転置した形で格納されていることを考慮しています。また、これはオブジェクト自体を変更しないので、const メソッドにします。

```
// 乗算
Matrix operator*(const Matrix &m) const
{
  Matrix t;

  for (int j = 0; j < 4; ++j)
  {
    for (int i = 0; i < 4; ++i)
    {
      const int ji(j * 4 + i);

      t[ji] = 0.0f;
      for (int k = 0; k < 4; ++k)
        t[ji] += matrix[k * 4 + i] * m[j * 4 + k];
    }
  }

  return t;
}
```

ただし、これは三重のループになっているので、性能が少し気になります。たとえば次のように書けば、ループを一重にできます。

```
// 乗算
Matrix operator*(const Matrix &m) const
{
  Matrix t;

  for (int i = 0; i < 16; ++i)
  {
    const int j(i & 3), k(i & ~3);

    t[i] =
      matrix[ 0 + j] * m[k + 0] +
      matrix[ 4 + j] * m[k + 1] +
      matrix[ 8 + j] * m[k + 2] +
      matrix[12 + j] * m[k + 3];
  }

  return t;
}
```

7.4.2 剛体変換

平行移動と回転は、立体の形状に影響を与えません。そのため、この二つを組み合わせた変換

は剛体変換と呼ばれます。

$$\mathbf{M} = \mathbf{T}(\mathbf{t})\mathbf{R}(l, m, n, \theta) = \begin{pmatrix} r_{00} & r_{10} & r_{20} & t_x \\ r_{01} & r_{11} & r_{21} & t_y \\ r_{02} & r_{12} & r_{22} & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{32}$$

したがって、この $\mathbf{M}$ は回転の変換行列 $\bar{\mathbf{R}}$ と平行移動のベクトル $\mathbf{t}$ で構成されます。

$$\bar{\mathbf{R}} = \begin{pmatrix} r_{00} & r_{10} & r_{20} \\ r_{01} & r_{11} & r_{21} \\ r_{02} & r_{12} & r_{22} \end{pmatrix}, \ \mathbf{t} = \begin{pmatrix} t_x \\ t_y \\ t_z \end{pmatrix} \ \Rightarrow \ \mathbf{M} = \begin{pmatrix} \bar{\mathbf{R}} & \mathbf{t} \\ \mathbf{0}^T & 1 \end{pmatrix} \tag{33}$$

7.4.3 任意の点を中心にした回転

7.3.6 で示した回転の変換は、いずれも原点を通る軸を中心としたものでした。任意の点を点を通る軸を中心に回転するには、一旦その点を原点に移す平行移動を行い、回転した後に元の位置に戻すという処理を行います。これは次の三つの変換の合成変換 $\mathbf{M} = \mathbf{T}(\mathbf{p})\,\mathbf{R}_z(\theta)\,\mathbf{T}(-\mathbf{p})$ です。

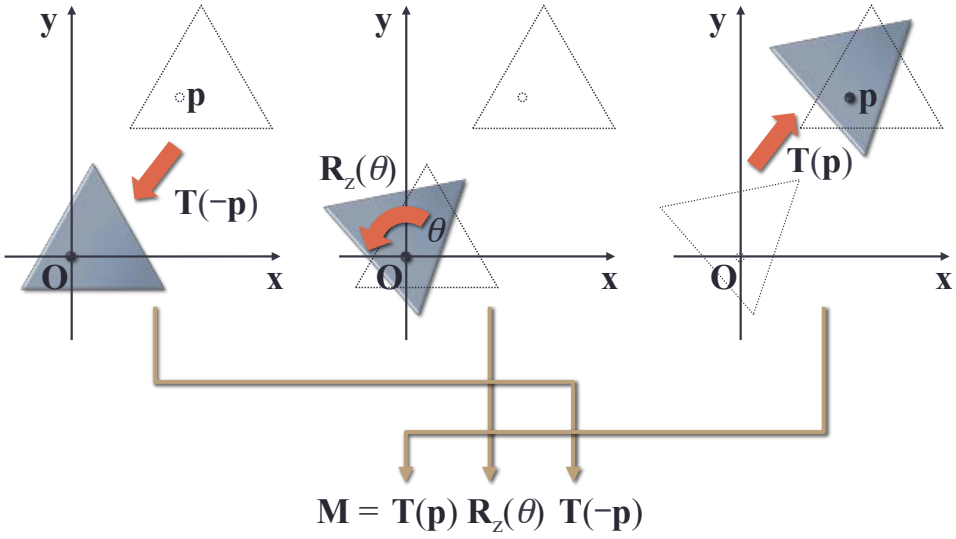


図 81 任意の点を中心とする回転

7.4.4 任意の点を中心にした拡大縮小

7.3.4 で示した拡大縮小は、原点を基準にしています。そのため、拡大縮小を行うとする図形が原点から離れていると、図形全体の位置も変ってしまいます。

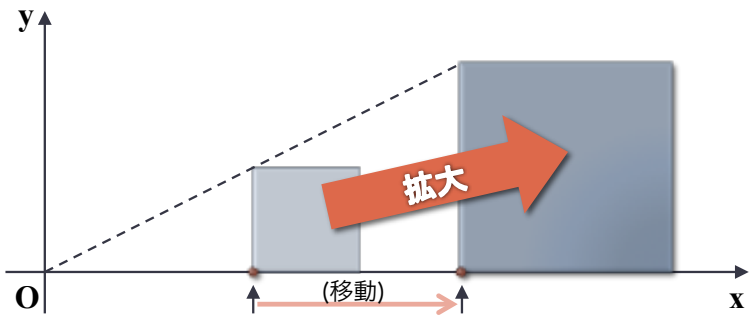


図 82 拡大縮小に伴う図形の移動

図形を、その図形の位置で拡大するためには、拡大縮小を行う際の基準点を図形の位置に移す必要があります。これは基準点が原点になるように図形を平行移動し、その後に拡大縮小して、元の位置に戻すことで実現します。これは次の三つの変換の合成変換 $\mathbf{M} = \mathbf{T}(\mathbf{p})\,\mathbf{S}(\mathbf{s})\,\mathbf{T}(-\mathbf{p})$ です。

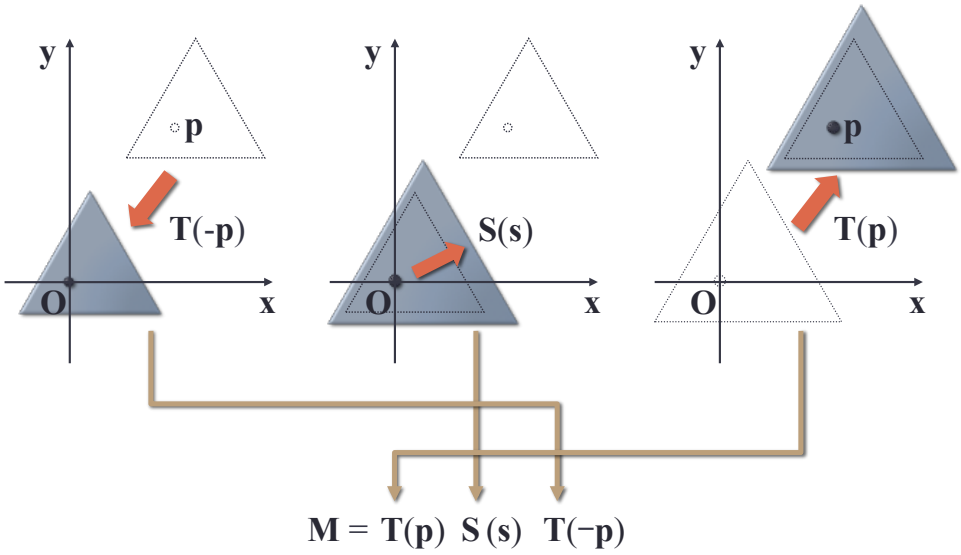


図 83 任意の基準点による拡大縮小

7.4.5 特定方向への拡大縮小

7.3.4 で示した拡大縮小は、また x、y、z のそれぞれの軸方向に対する拡大縮小しか行うことができません。任意の方向に対して拡大縮小を行うためには、その方向を一旦 x、y、z 軸の方向に合わせてから拡大縮小を行い、それから元の向きに戻します。いま、軸ベクトル $(\mathbf{x}\,\mathbf{y}\,\mathbf{z})$ をそれぞれ $(\mathbf{r}\,\mathbf{s}\,\mathbf{t})$ 方向に回転する変換行列を $\mathbf{F}$、s_r、s_s、s_t をそれぞれ x、y、z 軸方向の拡大率とする拡大縮小の変換行列 $\mathbf{S}$ とするとき、この変換は $\mathbf{M} = \mathbf{F}\mathbf{S}\mathbf{F}^T$ となります。

$$\mathbf{F} = \begin{pmatrix} \mathbf{r} & \mathbf{s} & \mathbf{t} & \mathbf{0} \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{34}$$

$$\mathbf{S} = \begin{pmatrix} s_r & 0 & 0 & 0 \\ 0 & s_s & 0 & 0 \\ 0 & 0 & s_t & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{35}$$

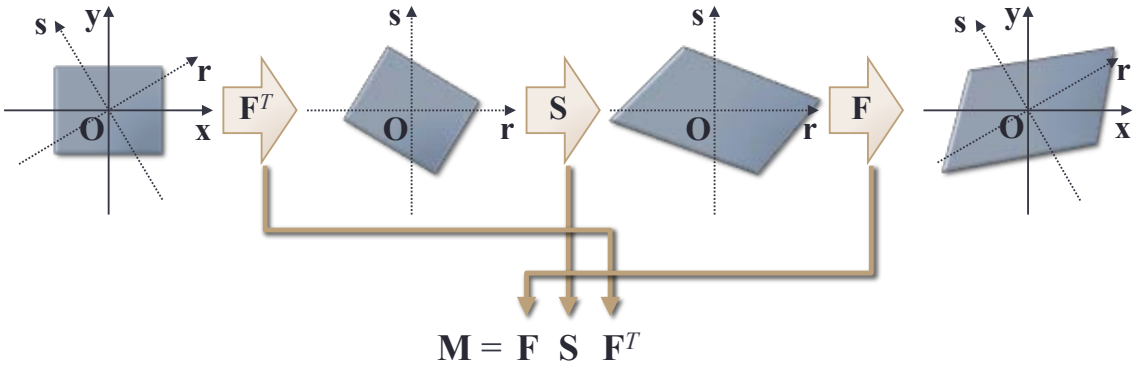


図 84 特定方向への拡大縮小

7.4.6 オイラー変換

原点を中心とした任意の回転は、x, y, z の三つの軸中心の回転を合成して表すことができます。この回転を表す各軸中心の回転角の組を**オイラー角**といい、それらによって表される回転の変換を**オイラー変換**といいます。

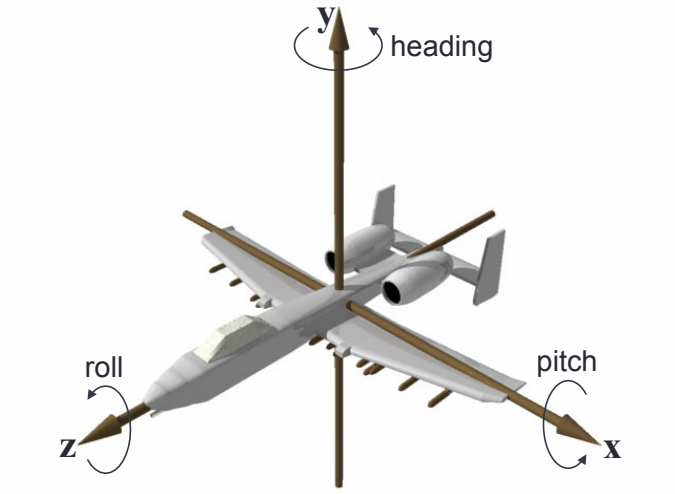


図 85 オイラー角

合成変換は変換行列の積で表されますが、行列の積には交換法則が成り立たちません。したがってオイラー変換の結果は、各軸中心の回転を合成する順序によって変化します。ここでは、z 軸中心の回転 (roll, bank) → x 軸中心の回転 (pitch) → y 軸中心の回転 (heading, yaw) の順に変換するものとします。また、それぞれの角度を r, p, h で表します。

- r: roll, bank (z 軸中心の回転角)
- p: pitch (x 軸中心の回転角)

- h: heading, yaw (y 軸中心の回転角)

このとき、オイラー変換 **E**(*h*, *p*, *r*) は次式で表されます。

$$
\begin{aligned}
\mathbf{E}(h,p,r) &= \mathbf{R}_y(h)\mathbf{R}_x(p)\mathbf{R}_z(r) \\
&= \begin{pmatrix} \cos h & 0 & \sin h & 0 \\ 0 & 1 & 0 & 0 \\ -\sin h & 0 & \cos h & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos p & -\sin p & 0 \\ 0 & \sin p & \cos p & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \cos r & -\sin r & 0 & 0 \\ \sin r & \cos r & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \\
&= \begin{pmatrix} \sin h \sin p \sin r + \cos h \cos r & \sin h \sin p \cos r - \cos h \sin r & \sin h \cos p & 0 \\ \cos p \sin r & \cos p \cos r & -\sin p & 0 \\ \cos h \sin p \sin r - \sin h \cos r & \cos h \sin p \cos r + \sin h \sin r & \cos h \cos p & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}
\end{aligned} \tag{36}
$$

● ジンバルロック

オイラー角のうち、$p = \pi/2$ の場合を考えます。このとき $\sin p = 1, \cos p = 0$ になるので、式 (36) は次のようになります。

$$
\mathbf{E}(h,\pi/2,r) = \begin{pmatrix} \sin h \sin r + \cos h \cos r & \sin h \cos r - \cos h \sin r & 0 & 0 \\ 0 & 0 & -1 & 0 \\ \cos h \sin r - \sin h \cos r & \cos h \cos r + \sin h \sin r & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{37}
$$

これは、加法定理により、次のように書き換えられます。

$$
\mathbf{E}(h,\pi/2,r) = \begin{pmatrix} \cos(h-r) & \sin(h-r) & 0 & 0 \\ 0 & 0 & -1 & 0 \\ -\sin(h-r) & \cos(h-r) & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{38}
$$

この回転は *h* - *r* という単一の角度で決定される、一つの軸による回転です。このため、*r* と *h* のどちらを変化させても同じ回転になり、自由度が一つ減ってしまいます。$p = \pi/2$ の回転によって *r* の回転軸 (z 軸) が *h* の回転軸と一致します。この現象を**ジンバルロック**といいます。

● 回転変換行列からオイラー角の算出

ある回転変換行列 **M** がオイラー変換 **E**(*h*, *p*, *r*) にもとづくものであったとします。

$$
\mathbf{M} = \begin{pmatrix} m_0 & m_4 & m_8 & m_{12} \\ m_1 & m_5 & m_9 & m_{13} \\ m_2 & m_6 & m_{10} & m_{14} \\ m_3 & m_7 & m_{11} & m_{15} \end{pmatrix} = \mathbf{E}(h,p,r) \tag{39}
$$

m_1 と m_5 の関係から、*r* を求めることができます。

$$\left.\begin{array}{l} m_1 = \cos p \sin r \\ m_5 = \cos p \cos r \end{array}\right\} \quad \rightarrow \quad r = \texttt{atan2}(m_5, m_1) \tag{40}$$

同様に、m_8 と m_{10} の関係から、h を求めることができます。

$$\left.\begin{array}{l} m_8 = \sin h \cos p \\ m_{10} = \cos h \cos p \end{array}\right\} \quad \rightarrow \quad h = \texttt{atan2}(m_{10}, m_8) \tag{41}$$

p は m_9 から求めることができます。

$$m_9 = -\sin p \rightarrow \quad p = \texttt{asin}(-m_9) \tag{42}$$

ただし、$m_1 = m_5 = 0$ のときは $\cos p = 0$ なので、$p = \pm \pi / 2$ となります。これはジンバルロックが発生している状態であり、式 (40) や式 (41) で r や h を求めることができません。その場合は、たとえば $h = 0$ とし、m_0 と m_{14} の関係から、h を求めることができます。

$$\left.\begin{array}{l} m_0 = \cos(h \mp r) \\ m_4 = \sin(h \mp r) \end{array}\right\} \quad \rightarrow \quad h = 0, r = -\texttt{atan2}(m_0, m_4) \tag{43}$$

7.4.7 変換の順序

変換の合成は行列の積で表されますが、行列の積は交換の法則が成り立ちません。したがって、同じ変換行列を合成した場合でも、その順序が異なれば、合成変換の結果は異なります。図 86 の例では、正方形を回転してから x 軸方向に引き伸ばすとひし形になりますが、x 軸方向に引き伸ばしてから回転すると回転した長方形になります。

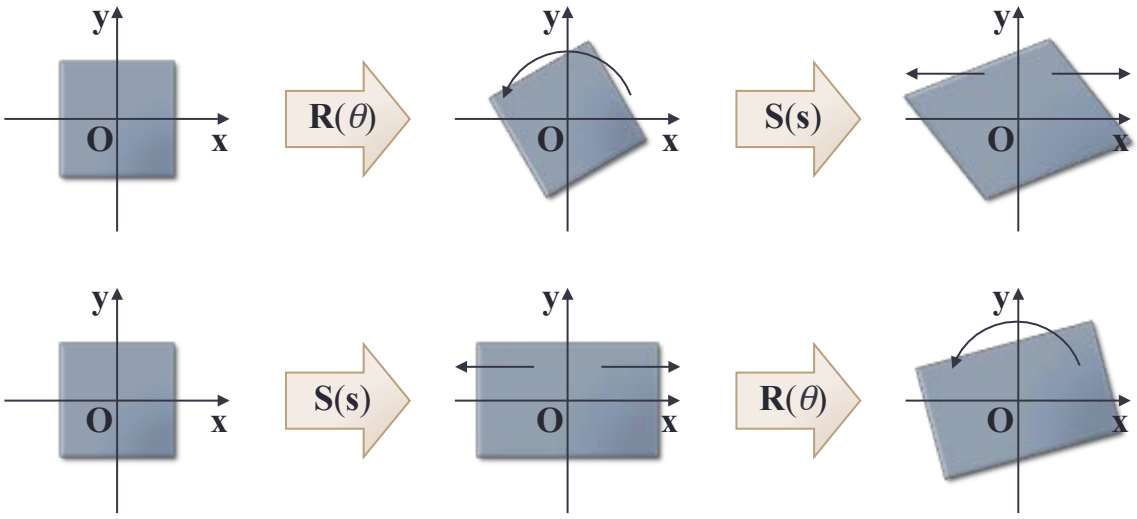


図 86 変換の順序による合成変換の結果の違い

7.5 変換行列を使った座標変換

それでは、第 6 章で作成したプログラムを、変換行列を使った座標変換で実装してみます。

● バーテックスシェーダ (point.vert) の変更点

座標変換はバーテックスシェーダで行いますから、作成した変換行列をシェーダプログラムに渡す必要があります。これまでは size、scale、および location という三つの uniform 変数を使ってきましたが、これを model という一つの mat4 型の uniform 変数に置き換えます。

バーテックスシェーダにおける座標変換は、変換行列を格納した uniform 変数を頂点位置に乗じるだけです。これを gl_Position に代入します。

```
#version 150 core
uniform mat4 model;
in vec4 position;
void main()
{
  gl_Position = model * position;
}
```

表 5 4 行 4 列の行列のデータ型

変数のデータ型	内容
mat4	単精度浮動小数点 4 行 4 列の行列

要素のアクセス

mat4 mc;	mat4 型の変数宣言
mc[0][0] = 1.0; mc[3] = vec4(0.0);	mc の 0 行 0 列の一つの要素に 1.0 を代入する mc の 3 行目の四つの要素に全部 0.0 を代入する

● メインプログラム (main.cpp) の変更点

Matrix クラスの定義を記述したヘッダファイル Matrix.h を、main.cpp の冒頭で #include します。

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

main() 関数では、これまで使っていたバーテックスシェーダの uniform 変数 size、scale、および location の代わりに、uniform 変数 model の場所をプログラムオブジェクトから取り出します。

```
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 modelLoc(glGetUniformLocation(program, "model"));

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

これまでシェーダで行っていた拡大縮小 (scale / size) を、scale() メソッドを使って変換行列に置き換えて、Matrix 型の変数 scaling に格納します。また、position による平行移動の変換も translate() メソッドを使って変換行列に置き換えて、Matrix 型の変数 translation に格納します。

この二つを乗算し、合成した変換を Matrix 型の変数 model に格納します。そして model のメンバの配列 matrix のポインタを data() メソッドで取り出して、glUniformMatrix4fv() 関数を使ってバーテックスシェーダの uniform 変数 model に設定します。

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

    // 拡大縮小の変換行列を求める
    const GLfloat *const size(window.getSize());
    const GLfloat scale(window.getScale() * 2.0f);
    const Matrix scaling(Matrix::scale(scale / size[0], scale / size[1], 1.0f));

    // 平行移動の変換行列を求める
    const GLfloat *const position(window.getLocation());
    const Matrix translation(Matrix::translate(position[0], position[1], 0.0f));

    // モデル変換行列を求める
    const Matrix model(translation * scaling);

    // uniform 変数に値を設定する
    glUniformMatrix4fv(modelLoc, 1, GL_FALSE, model.data());

    // 図形を描画する
    shape->draw();

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

```
  }
}
```

■ サンプルプログラム step13

void glUniformMatrix4fv(GLint location, GLsizei count, GLboolean transpose, 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 mat4 型の uniform 変数に GLfloat 型の配列変数 value の値を設定します。

location

値を設定する mat4 型の uniform 変数の index。

count

設定する uniform 変数が mat4 型の配列のとき、その要素数。配列でなければ 1。

transpose

配列変数を転置して格納する場合は GL_TRUE、転置しない場合は GL_FALSE。したがって、これを GL_TRUE にすれば、配列の要素の順序を変換行列と一致させることができる (図 74 参照)。

value

設定する GLfloat 型の配列変数。mat4 型は 4 行 4 列の float 型の要素を持つ行列なので、GLfloat 型の要素を count × 16 個もつ配列を指定する。

7.6 直交座標系の変換

7.6.1 基底ベクトルの変換

互いに直交する三つの単位ベクトル **i**', **j**', **k**' を軸とする座標系上の点の位置 **p** を、この座標系と原点を共有する、別の直交する三つの単位ベクトル **i**, **j**, **k** を軸とする座標系に移す変換を求めます。この変換は回転の変換になります。それぞれの座標系における **p** の座標値を (x', y', z') と (x, y, z) とするとき、**p** は次のように表すことができます。

$$\mathbf{p} = x\mathbf{i} + y\mathbf{j} + z\mathbf{k} = x'\mathbf{i}' + y'\mathbf{j}' + z'\mathbf{k}' \tag{44}$$

この式は、行列を用いて次のように書くことができます。

$$\begin{pmatrix} \mathbf{i} & \mathbf{j} & \mathbf{k} \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} \mathbf{i}' & \mathbf{j}' & \mathbf{k}' \end{pmatrix} \begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} \tag{45}$$

したがって (x, y, z) は、次式で求めることができます。

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} \mathbf{i} & \mathbf{j} & \mathbf{k} \end{pmatrix}^{-1} \begin{pmatrix} \mathbf{i}' & \mathbf{j}' & \mathbf{k}' \end{pmatrix} \begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} \tag{46}$$

ここで行列 **(i j k), (i' j' k')** は直交行列ですから、これらの逆行列は転置行列と等しくなります。

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} \mathbf{i} & \mathbf{j} & \mathbf{k} \end{pmatrix}^{T} \begin{pmatrix} \mathbf{i}' & \mathbf{j}' & \mathbf{k}' \end{pmatrix} \begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} \tag{47}$$

しがたがって、次の変換行列 **M** は *(x', y', z')* を *(x, y, z)* に回転します。

$$\mathbf{M} = \begin{pmatrix} \mathbf{i} & \mathbf{j} & \mathbf{k} \end{pmatrix}^{T} \begin{pmatrix} \mathbf{i}' & \mathbf{j}' & \mathbf{k}' \end{pmatrix} = \begin{pmatrix} \mathbf{i}\cdot\mathbf{i}' & \mathbf{i}\cdot\mathbf{j}' & \mathbf{i}\cdot\mathbf{k}' \\ \mathbf{j}\cdot\mathbf{i}' & \mathbf{j}\cdot\mathbf{j}' & \mathbf{j}\cdot\mathbf{k}' \\ \mathbf{k}\cdot\mathbf{i}' & \mathbf{k}\cdot\mathbf{j}' & \mathbf{k}\cdot\mathbf{k}' \end{pmatrix} \tag{48}$$

7.6.2 座標軸の回転

ここで、この M について別の (ちょっとくどい) 説明をします。いま、互いに直交する三つの単位ベクトルを **r, s, t** とします。

$$\mathbf{r} = \begin{pmatrix} r_x \\ r_y \\ r_z \end{pmatrix},\ \mathbf{s} = \begin{pmatrix} s_x \\ s_y \\ s_z \end{pmatrix},\ \mathbf{t} = \begin{pmatrix} t_x \\ t_y \\ t_z \end{pmatrix} \tag{49}$$

また、x 軸、y 軸、z 軸のそれぞれの軸方向の単位ベクトル **x, y, z** は次のように表されます。

$$\mathbf{x} = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix},\ \mathbf{y} = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix},\ \mathbf{z} = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} \tag{50}$$

x, y, z を **r, s, t** に一致させる回転の変換行列を **M** とします。

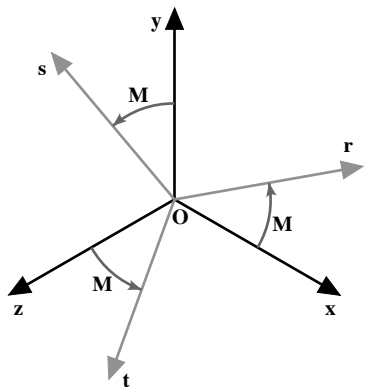


図 87 座標軸の回転

$$\begin{cases} \mathbf{r} = \mathbf{M}\mathbf{x} \\ \mathbf{s} = \mathbf{M}\mathbf{y} \\ \mathbf{t} = \mathbf{M}\mathbf{z} \end{cases} \tag{51}$$

これを行列で表すと、次のようになります。

$$\begin{pmatrix} \mathbf{r} & \mathbf{s} & \mathbf{t} \end{pmatrix} = \mathbf{M} \begin{pmatrix} \mathbf{x} & \mathbf{y} & \mathbf{z} \end{pmatrix} \tag{52}$$

この右辺の **(x y z)** は単位行列ですから、行列 **(r s t)** そのものが、求める **M** になります。

$$\begin{pmatrix} r_x & s_x & t_x \\ r_y & s_y & t_y \\ r_z & s_z & t_z \end{pmatrix} = \mathbf{M} \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} = \mathbf{M} \tag{53}$$

r, s, t を **x, y, z** に一致させる変換行列 $\mathbf{M}^{-1}$ は、次のようにして求めることができます。

$$\mathbf{M}^{-1} = \begin{pmatrix} r_x & s_x & t_x \\ r_y & s_y & t_y \\ r_z & s_z & t_z \end{pmatrix}^{-1} = \begin{pmatrix} r_x & s_x & t_x \\ r_y & s_y & t_y \\ r_z & s_z & t_z \end{pmatrix}^{T} = \begin{pmatrix} r_x & r_y & r_z \\ s_x & s_y & s_z \\ t_x & t_y & t_z \end{pmatrix} = \begin{pmatrix} \mathbf{r}^T \\ \mathbf{s}^T \\ \mathbf{t}^T \end{pmatrix} \tag{54}$$

7.6.3 ベクトルの方向転換

ある単位ベクトル **u** の向きを、単位ベクトル **v** の方向に向ける変換 **M** を求めます。**u** と **v** の両方に直交するベクトルを **n** とします。

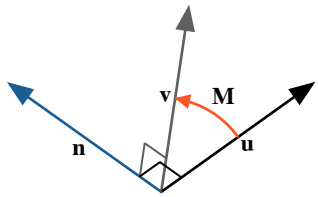


図 88 ベクトルの方向転換

n は **u** と **v** の外積により求めることができます。

$$\mathbf{n} = \frac{\mathbf{u} \times \mathbf{v}}{\|\mathbf{u} \times \mathbf{v}\|} \tag{55}$$

さらに **u** と **n** に直交するベクトル **l** と、**v** と **n** に直交するベクトル **m** を求めます。

$$\mathbf{l} = \frac{\mathbf{u} \times \mathbf{n}}{\|\mathbf{u} \times \mathbf{n}\|} = \frac{\mathbf{u} \times (\mathbf{u} \times \mathbf{v})}{\|\mathbf{u} \times (\mathbf{u} \times \mathbf{v})\|} \tag{56}$$

$$\mathbf{m} = \frac{\mathbf{v} \times \mathbf{n}}{\|\mathbf{v} \times \mathbf{n}\|} = \frac{\mathbf{v} \times (\mathbf{u} \times \mathbf{v})}{\|\mathbf{v} \times (\mathbf{u} \times \mathbf{v})\|} \tag{57}$$

これにより、求める変換 **M** は **(u, n, l)** を基底ベクトル (座標軸) とする空間を、**(v, n, m)** を基底ベクトルとする空間に回転する変換になります。

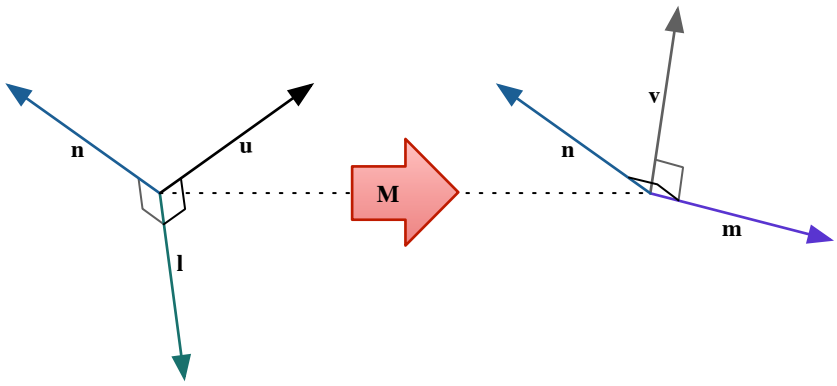


図 89 座標軸の回転

$$
\begin{aligned}
\mathbf{M}_u &= \begin{pmatrix} \mathbf{u} & \mathbf{n} & \mathbf{l} \end{pmatrix} = \begin{pmatrix} \mathbf{u} & \dfrac{\mathbf{u}\times\mathbf{v}}{\|\mathbf{u}\times\mathbf{v}\|} & \dfrac{\mathbf{u}\times(\mathbf{u}\times\mathbf{v})}{\|\mathbf{u}\times(\mathbf{u}\times\mathbf{v})\|} \end{pmatrix} \\
\mathbf{M}_v &= \begin{pmatrix} \mathbf{v} & \mathbf{n} & \mathbf{m} \end{pmatrix} = \begin{pmatrix} \mathbf{v} & \dfrac{\mathbf{u}\times\mathbf{v}}{\|\mathbf{u}\times\mathbf{v}\|} & \dfrac{\mathbf{v}\times(\mathbf{u}\times\mathbf{v})}{\|\mathbf{v}\times(\mathbf{u}\times\mathbf{v})\|} \end{pmatrix} \\
\mathbf{M} &= \mathbf{M}_v\mathbf{M}_u^{-1} = \mathbf{M}_v\mathbf{M}_u^{T}
\end{aligned}
\tag{58}
$$

7.7 ビュー変換

7.7.1 視点の位置の移動

7.5 までに作成したプログラムでは、図形データの頂点に x 座標値と y 座標値のみを指定して、xy 平面上の二次元の図形を表示していました。しかし、本章で見てきた通り OpenGL 自体は同次座標を使って三次元の処理を行っています。このため、この図形の頂点の座標値は、実際には $z=0$ の三次元の座標値として扱われています。

そこで、ここでは図形を描画する際の視点の位置を三次元空間中の別の場所に移して、そこから見た図形を描くことを考えてみます。ビュー変換は図形を視点の位置と方向から見た座標系 (視点座標系) に移す座標変換です。

ここで自分が画面を正面から見ているとき、自分の視点が空間中のどこにあるか考えてみます。画面上で x 軸は画面の右方向、y 軸は画面の上方向に向いています。z 軸は x 軸と y 軸の両方と直交していますから、画面に対して垂直になっているはずです。OpenGL では右手系 (図 90 の左) の座標系を用いますから、z 軸は画面から見ている人の方に向かって伸びています。

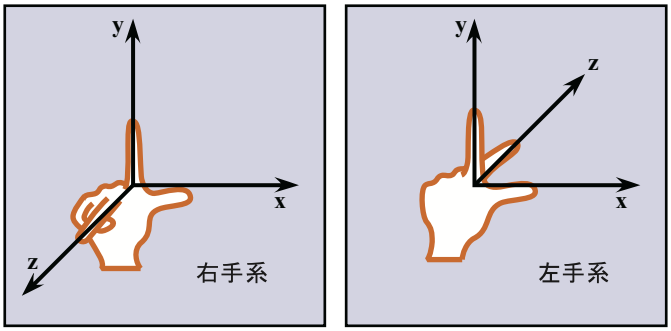


図 90 右手系と左手系

すなわち視点は原点にあり、視線は z 軸の負の方向を向いています。ビュー変換は三次元空間中に置かれた視点の位置を原点とし、視線を z 軸の負の方向としたときの、図形の位置を求める処理です。

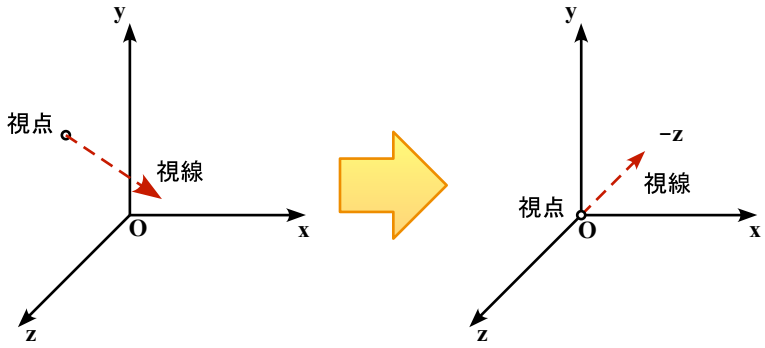


図 91 ビュー変換

● ビュー変換行列を作成するメソッドの追加 (Matrix.h)

視点の位置、目標点の位置、および視野の上方向のベクトルを指定してビュー変換行列を作成するメソッド lookat() を Matrix クラスに追加します。視点の位置は $\mathbf{e} = (e_x, e_y, e_z)$、視線上にある目標点の位置は $\mathbf{g} = (g_x, g_y, g_z)$、視野の上方向のベクトルを $\mathbf{u} = (u_x, u_y, u_z)$ とします。上方向を y 軸方向とするなら、これに (0, 1, 0) を設定します。これもインスタンスを生成せずに呼び出せるように static メソッドにします。

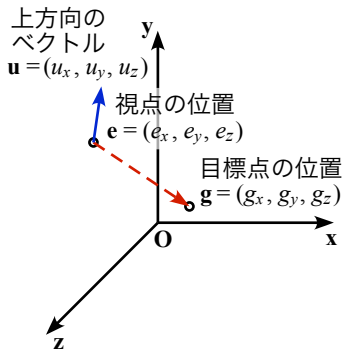


図 92 視点の設定

```
// ビュー変換行列を作成する
static Matrix lookat(
  GLfloat ex, GLfloat ey, GLfloat ez,   // 視点の位置
  GLfloat gx, GLfloat gy, GLfloat gz,   // 目標点の位置
  GLfloat ux, GLfloat uy, GLfloat uz)   // 上方向のベクトル
{
```

まず、視点 $\mathbf{e} = (e_x, e_y, e_z)$ を原点 $\mathbf{O} = (0, 0, 0)$ に平行移動する変換行列 $\mathbf{T}_v$ を作ります。

$$\mathbf{T}_v = \begin{pmatrix} 1 & 0 & 0 & -e_x \\ 0 & 1 & 0 & -e_y \\ 0 & 0 & 1 & -e_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{59}$$

```
  // 平行移動の変換行列
  const Matrix tv(translate(-ex, -ey, -ez));
```

視点の位置 **e** を原点に移すと、目標点の位置は目標点の位置は **g**−**e** になります。これと上方向のベクトル **u** から、視点を原点とした座標系である視点座標系の軸ベクトルを求めます。これを **(r s t)** とします。

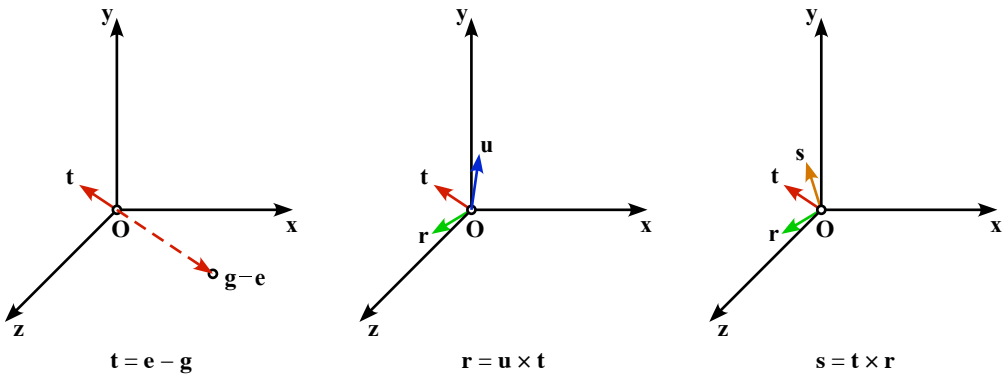


図 93 視点座標系の軸ベクトル (r s t)

視点座標系における z 軸に相当するベクトル $\mathbf{t} = (t_x, t_y, t_z)$ は、視点方向の逆ベクトル **e** - **g** になります。視点座標系の x 軸に相当するベクトル $\mathbf{r} = (r_x, r_y, r_z)$ は、上方向のベクトル **u** と **t** との外積 **u** × **t** により求めます。視点の座標系の y 軸に相当するベクトル $\mathbf{s} = (s_x, s_y, s_z)$ を、ベクトル **t** とベクトル **r** の外積 **t** × **r** で求めます。

$$
\begin{aligned}
\mathbf{t} = \mathbf{e} - \mathbf{g} &= \begin{pmatrix} e_x - g_x \\ e_y - g_y \\ e_z - g_z \end{pmatrix} = \begin{pmatrix} t_x \\ t_y \\ t_z \end{pmatrix} \\
\mathbf{r} = \mathbf{u} \times \mathbf{t} &= \begin{pmatrix} u_y t_z - u_z t_y \\ u_z t_x - u_x t_z \\ u_x t_y - u_y t_x \end{pmatrix} = \begin{pmatrix} r_x \\ r_y \\ r_z \end{pmatrix} \\
\mathbf{s} = \mathbf{t} \times \mathbf{r} &= \begin{pmatrix} t_y r_z - t_z r_y \\ t_z r_x - t_x r_z \\ t_x r_y - t_y r_x \end{pmatrix} = \begin{pmatrix} s_x \\ s_y \\ s_z \end{pmatrix}
\end{aligned} \tag{60}
$$

```
    // t 軸 = e - g
    const GLfloat tx(ex - gx);
    const GLfloat ty(ey - gy);
    const GLfloat tz(ez - gz);

    // r 軸 = u x t 軸
    const GLfloat rx(uy * tz - uz * ty);
    const GLfloat ry(uz * tx - ux * tz);
    const GLfloat rz(ux * ty - uy * tx);

    // s 軸 = t 軸 x r 軸
    const GLfloat sx(ty * rz - tz * ry);
    const GLfloat sy(tz * rx - tx * rz);
    const GLfloat sz(tx * ry - ty * rx);
```

得られた視点座標系の軸ベクトル (**r s t**) をそれぞれ正規化し、式 (54) にもとづいて式 (61) に示す回転の変換行列 $\mathbf{R}_v$ を作ります。

$$
\mathbf{R}_v = \begin{pmatrix} \dfrac{\mathbf{r}^T}{|\mathbf{r}|} & 0 \\ \dfrac{\mathbf{s}^T}{|\mathbf{s}|} & 0 \\ \dfrac{\mathbf{t}^T}{|\mathbf{t}|} & 0 \\ 0 \quad 0 \quad 0 & 1 \end{pmatrix} = \begin{pmatrix} \dfrac{r_x}{|\mathbf{r}|} & \dfrac{r_y}{|\mathbf{r}|} & \dfrac{r_z}{|\mathbf{r}|} & 0 \\ \dfrac{s_x}{|\mathbf{s}|} & \dfrac{s_y}{|\mathbf{s}|} & \dfrac{s_z}{|\mathbf{s}|} & 0 \\ \dfrac{t_x}{|\mathbf{t}|} & \dfrac{t_y}{|\mathbf{t}|} & \dfrac{t_z}{|\mathbf{t}|} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{61}
$$

まず、ベクトル **s** の長さを調べます。これが 0 でなければ、ベクトル **r** とベクトル **t** の長さは、ともに 0 ではありません。このとき、これらを正規化して、4×4=16 要素の配列変数 rv の左上の 3×3 の要素に格納します。

```
    // s 軸の長さのチェック
    const GLfloat s2(sx * sx + sy * sy + sz * sz);
    if (s2 == 0.0f) return tv;

    // 回転の変換行列
    Matrix rv;
    rv.loadIdentity();

    // r 軸を正規化して配列変数に格納
    const GLfloat r(sqrt(rx * rx + ry * ry + rz * rz));
    rv[ 0] = rx / r;
```

```
    rv[ 4] = ry / r;
    rv[ 8] = rz / r;

    // s 軸を正規化して配列変数に格納
    const GLfloat s(sqrt(s2));
    rv[ 1] = sx / s;
    rv[ 5] = sy / s;
    rv[ 9] = sz / s;

    // t 軸を正規化して配列変数に格納
    const GLfloat t(sqrt(tx * tx + ty * ty + tz * tz));
    rv[ 2] = tx / t;
    rv[ 6] = ty / t;
    rv[10] = tz / t;
```

平行移動の変換行列 $\mathbf{T}_v$ に回転の変換行列 $\mathbf{R}_v$ を乗じて、ビュー変換行列を求めます。

$$\mathbf{M}_v = \mathbf{R}_v\mathbf{T}_v \tag{62}$$

```
  // 視点の平行移動の変換行列に視線の回転の変換行列を乗じる
  return rv * tv;
}
```

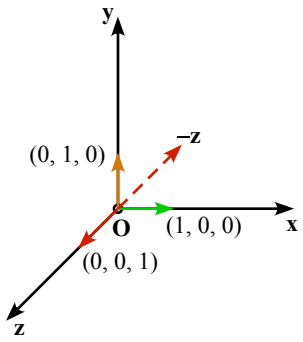


図 94 回転の変換適用後の視点座標系の座標軸

● メインプログラム (main.cpp) の変更点

lookat() メソッドを使ってビュー変換行列を求め、Matrix 型の変数 view に格納します。これをモデル変換 model に乗じて合成し、Matrix 型の変数 modelview に格納します。この二つの変換はまとめて実行されることが多く、**モデルビュー変換**と呼ばれます。そこでシェーダの uniform 変数の変数名も、model から modelview に変更しておきます。

```
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 modelviewLoc(glGetUniformLocation(program, "modelview"));
```

```
  // 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)
  {
    《省略》

    // モデル変換行列を求める
    const Matrix model(translation * scaling);

    // ビュー変換行列を求める
    const Matrix view(Matrix::lookat(0.0f, 0.0f, 0.0f, -1.0f, -1.0f, -1.0f,
      0.0f, 1.0f, 0.0f));

    // モデルビュー変換行列を求める
    const Matrix modelview(view * model);

    // uniform 変数に値を設定する
    glUniformMatrix4fv(modelviewLoc, 1, GL_FALSE, modelview.data());

    // 図形を描画する
    shape->draw();

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

この例では視点の位置は原点 (0, 0, 0) とし、そこから (-1, -1, -1) にある目標点を見ています。視点を原点に置いているのは、この段階で視点の位置を変更すると、図形が視体積の外に出てしまう可能性があるからです。視点を任意の位置に配置するには、投影変換を行う必要があります。

● バーテックスシェーダ (point.vert の変更点

バーテックスシェーダの uniform 変数 model を modelview に変更します。

```
#version 150 core
uniform mat4 modelview;
in vec4 position;
void main()
{
  gl_Position = modelview * position;
}
```

■ サンプルプログラム step14

● 実行結果

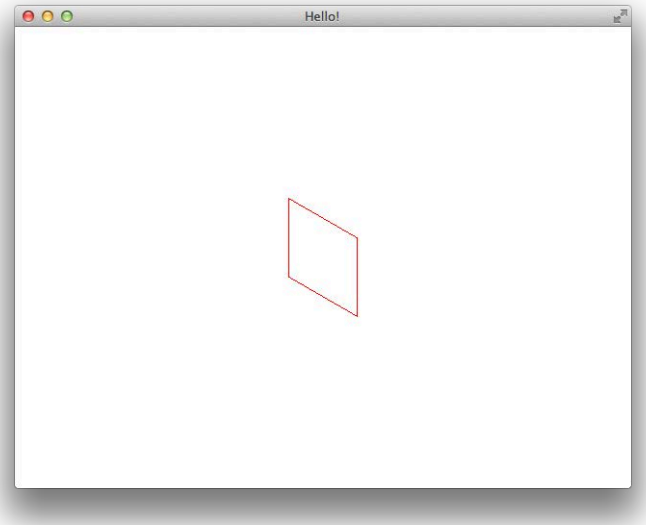


図 95 ビュー変換を実装した実行結果

ウィンドウ上でマウスをドラッグすると、図形がこの空間の xy 平面上を移動します。

7.8 投影変換

7.8.1 標準視体積

二次元の図形表示では、画面上の表示領域 (ビューポート) に正規化デバイス座標系のクリッピング領域内のものが表示されます。この領域には**深度** (奥行き) 方向にも表示可能な範囲が設定されており、図 96 に示す各軸 [-1, 1] の範囲の立方体の空間になっています。これを**標準視体積** (Canonical View Volume) といいます。画面には、この xy 平面上への投影像が表示されます。

このため、三次元空間の任意の領域に置かれた物体を画面に表示するには、表示しようとする領域を、この標準視体積に収める必要があります。これは投影変換 (7.1.4) により行います。

投影変換の方法には、距離に関係なく遠くのものも近くのものも同じ大きさで表示される直交投影と、遠いものほど小さく表示される透視投影があります。これら以外にも、たとえば遠いものほど大きく表示したり、魚眼レンズのように歪ませて表示したりする投影方法などもありますが、ここでは取り扱いません。

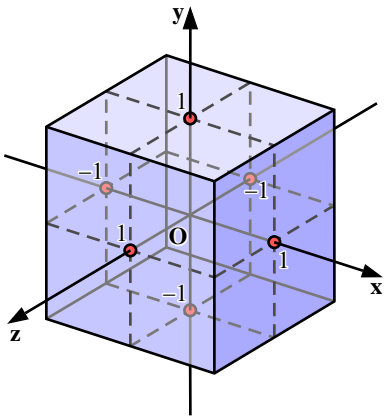


図 96 標準視体積

7.8.2 直交投影

● 直交投影の手順

三次元空間中の任意の領域を標準視体積に収めるために、直交投影ではワールド座標系 (6.1.4) 上の 2 点 (*left*, *bottom*) および (*right*, *top*) を対角の頂点とする図形の表示領域の他に、深度方向の範囲 (*near*, *far*) も指定します。

いま、視点が原点にあり、視線が xy 平面に垂直な z 軸上の負の方向を向いている (右手系) とします。*near* は表示を行うもっとも視点に視線に垂直な平面の深度 (xy 平面からの距離) であり、*far* は最も遠方にある平面の深度です。これらによって決まる直方体の領域を、**視体積** (View Volume) といいます。

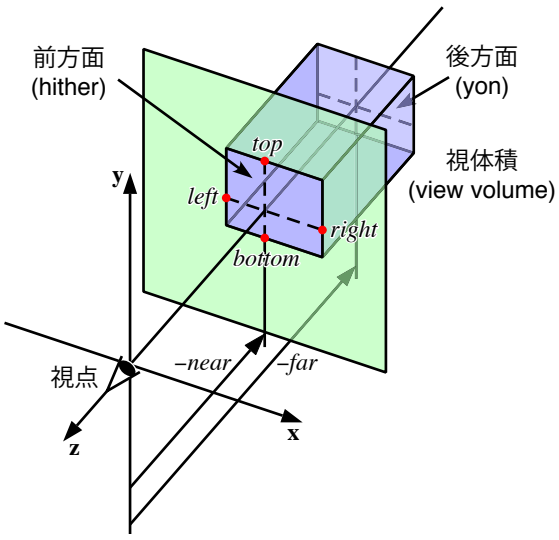


図 97 視体積

この視点に近いほうの平面を**前方面** (hither) 、遠い方の面を**後方面** (yon) といいます。ビュー変換によって視点は z 軸方向の負の方向に向いていますから、前方面と後方面の z 値は、それぞれ *−near*、*−far* になります。

この視体積内の図形を標準視体積に収めるには、視体積の中心が原点になるように平行移動し、一辺の長さが 2 になるように拡大縮小します。

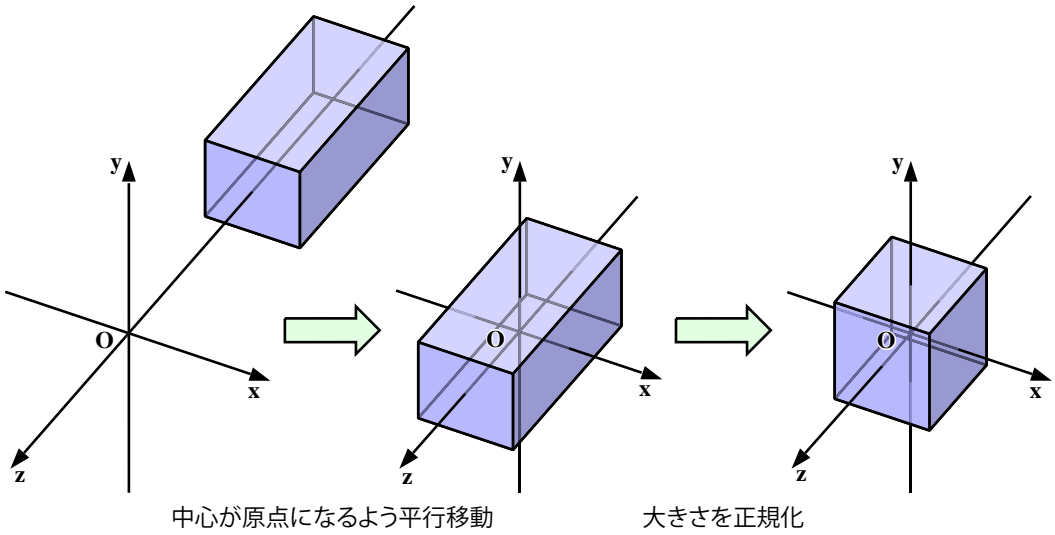


図 98 直交投影の手順

● **中心が原点になるよう平行移動**

まず、視体積の中心の位置を求め、そこが原点になるように平行移動する変換行列を求めます。視体積の中心は ((*right* + *left*) / 2, (*top* + *bottom*) / 2, -(*far* + *near*) / 2) です。

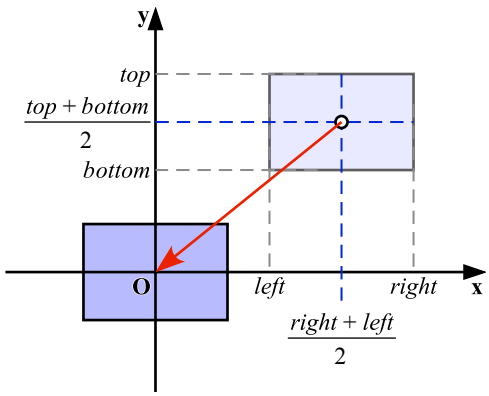


図 99 中心が原点となるように平行移動

この変換行列は平行移動の変換行列 (19) より、(63) 式のようになります。*far* と *near* に負号が付いているので、Z に関しては逆に負号が無くなっています。

$$\mathbf{M}_{centering} = \begin{pmatrix} 1 & 0 & 0 & -\dfrac{right+left}{2} \\ 0 & 1 & 0 & -\dfrac{top+bottom}{2} \\ 0 & 0 & 1 & \dfrac{far+near}{2} \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{63}$$

● 大きさを正規化

視体積の幅、高さ、および深度を求め、それを一辺の長さが 2 になるように拡大縮小する変換行列を求めます。視体積の幅、高さ、深度は、それぞれ *right* – *left*、*top* – *bottom*、–(*far* – *near*) です。

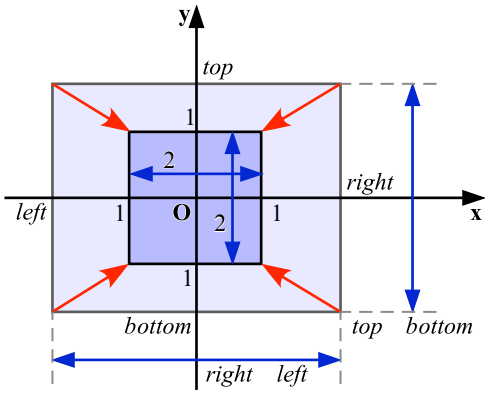


図 100 一辺の長さが 2 になるよう拡大縮小

この変換行列は (22) 式の拡大縮小の変換行列より (64) 式のようになります。これも *far* と *near* に負号が付いているので、Z に関しては負号が付いています。

$$\mathbf{M}_{scaling} = \begin{pmatrix} \frac{2}{right-left} & 0 & 0 & 0 \\ 0 & \frac{2}{top-bottom} & 0 & 0 \\ 0 & 0 & -\frac{2}{far-near} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{64}$$

● 直交投影変換行列

直交投影変換行列 $\mathbf{M}_o$ は、この二つの変換行列 $\mathbf{M}_{centering}$ と $\mathbf{M}_{scaling}$ を乗じたものです。

$$\begin{aligned} \mathbf{M}_o &= \mathbf{M}_{scaling}\mathbf{M}_{centering} \\ &= \begin{pmatrix} \frac{2}{right-left} & 0 & 0 & -\frac{right+left}{right-left} \\ 0 & \frac{2}{top-bottom} & 0 & -\frac{top+bottom}{top-bottom} \\ 0 & 0 & -\frac{2}{far-near} & -\frac{far+near}{far-near} \\ 0 & 0 & 0 & 1 \end{pmatrix} \end{aligned} \tag{65}$$

● 直交投影変換行列を作成するメソッドの追加 (Matrix.h)

Matrix クラスにビュー変換行列を作るメソッド orthogonal() を追加します。引数に指定した視体積 (*left*, *bottom*)、(*right*, *top*)、(*near*, *far*) から式 (65) によって変換行列を求めます。これもイン

スタンスを生成せずに呼び出せるように static メソッドにします。

```
  // 直交投影変換行列を作成する
  static Matrix orthogonal(GLfloat left, GLfloat right,
    GLfloat bottom, GLfloat top,
    GLfloat zNear, GLfloat zFar)
  {
    Matrix t;
    const GLfloat dx(right - left);
    const GLfloat dy(top - bottom);
    const GLfloat dz(zFar - zNear);

    if (dx != 0.0f && dy != 0.0f && dz != 0.0f)
    {
      t.loadIdentity();
      t[ 0] = 2.0f / dx;
      t[ 5] = 2.0f / dy;
      t[10] = -2.0f / dz;
      t[12] = -(right + left) / dx;
      t[13] = -(top + bottom) / dy;
      t[14] = -(zFar + zNear) / dz;
    }

    return t;
  }
```

● メインプログラム (main.cpp) の変更点

投影変換行列をバーテックスシェーダに渡すために、モデルビュー変換行列 modelview とは別の uniform 変数を用意します。変数名は projection にします。

```
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 modelviewLoc(glGetUniformLocation(program, "modelview"));
  const GLint projectionLoc(glGetUniformLocation(program, "projection"));

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

ウィンドウのサイズの変更に対する表示図形の縦横比の維持 (6.1.4) やサイズの固定 (6.1.5) のような処理は、一般には投影変換の際に行われます。そこで前方面の表示領域として、図 64 に示す $left = -w / 2s$、$right = w / 2s$、$bottom = -h / 2s$、$top = h / 2s$ を設定します。

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

    // 直交投影変換行列を求める
    const GLfloat *const size(window.getSize());
    const GLfloat scale(window.getScale() * 2.0f);
    const GLfloat w(size[0] / scale), h(size[1] / scale);
    const Matrix projection(Matrix::orthogonal(-w, w, -h, h, 1.0f, 10.0f));
```

これに伴って、モデルビュー変換行列には平行移動とビュー変換だけを設定します。ビュー変換では、目標点を図形の位置の (初期値の) 原点に設定すれば、視点位置は任意の位置に設定できます。ただし、目標点は視点から見て zNear (1) から zFar (10) の間にある必要があります。

```
    // モデル変換行列を求める
    const GLfloat *const location(window.getLocation());
    const Matrix model(Matrix::translate(location[0], location[1], 0.0f));

    // ビュー変換行列を求める
    const Matrix view(Matrix::lookat(3.0f, 4.0f, 5.0f, 0.0f, 0.0f, 0.0f,
      0.0f, 1.0f, 0.0f));

    // モデルビュー変換行列を求める
    const Matrix modelview(view * model);
```

変換行列 projection の内容をシェーダプログラムの uniform 変数 projection に設定して描画します。

```
    // uniform 変数に値を設定する
    glUniformMatrix4fv(projectionLoc, 1, GL_FALSE, projection.data());
    glUniformMatrix4fv(modelviewLoc, 1, GL_FALSE, modelview.data());

    // 図形を描画する
    shape->draw();

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

● バーテックスシェーダ (point.vert) の変更点

バーテックスシェーダに mat4 型の uniform 変数 projection を追加し、それをモデルビュー変換後の頂点の座標値に乗じます。

```
#version 150 core
uniform mat4 modelview;
uniform mat4 projection;
in vec4 position;
void main()
{
```

```
  gl_Position = projection * modelview * position;
}
```

■ サンプルプログラム step15

● 実行結果

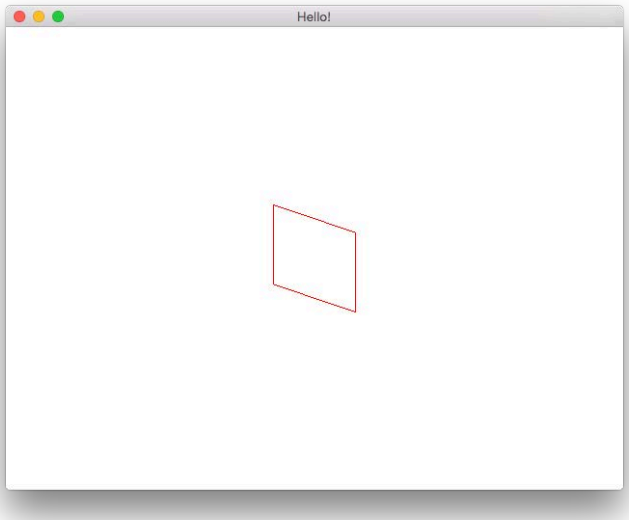


図 101 直交投影変換を実装した実行結果

7.8.3 透視投影

透視投影も直交投影と同様に、前方面上の 2 点 (*left*, *bottom*) および (*right*, *top*) を対角の頂点とする図形の表示領域と、深度方向の範囲 (*near*, *far*) を指定します。ただし透視投影の場合は、視点を頂点とし後方面を底面とする四角錐を前方面で切り落とした視錐台 (View Frustum) の内部にある図形を標準視体積に収めます。

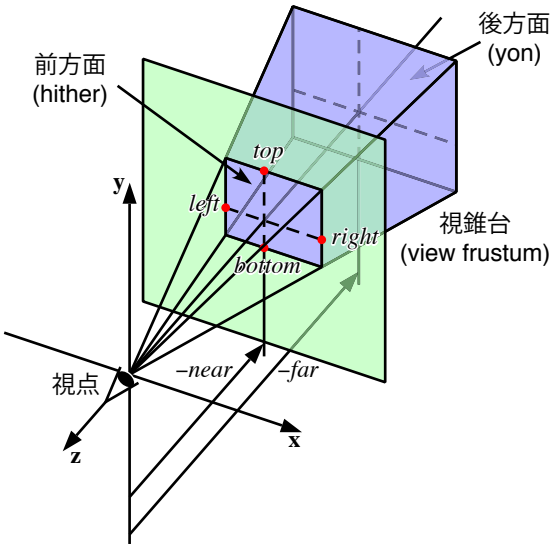


図 102 視錐台

この視錐台内の図形を標準視体積に収めるには、視錐台の中心軸が視線 (z 軸) と一致するようにせん断変形し、それを透視投影した後、中心が原点になるように平行移動し、最後に一辺の長さが 2 になるように拡大縮小します。

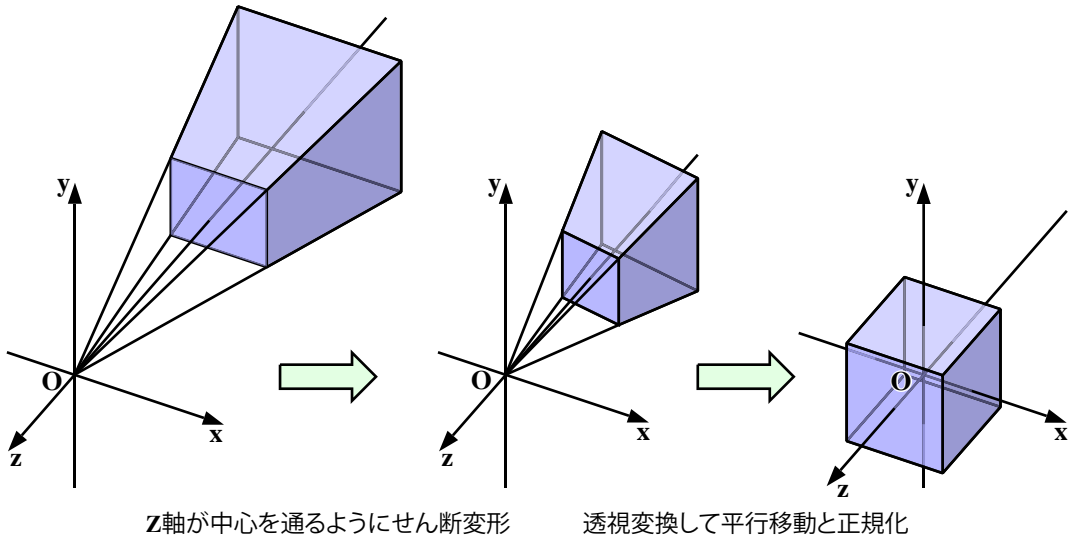


図 103 透視投影の手順

● **せん断変形**

透視投影の場合は、前方面上の図形の表示領域の中心を z 軸が通るようにするために、視錐台全体をせん断変形します。前方面は *–near* の位置にあり、その上の表示領域の中心は ((*right* + *left*) / 2, (*top* + *bottom*) / 2) にありますから、z = –1 における移動量は ((*right* + *left*) / 2*near*, (*top* + *bottom*) / 2 *near*) になります。

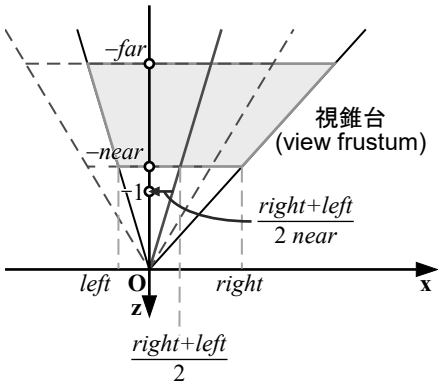


図 104 せん断変形変換

$$\mathbf{M}_{shear} = \begin{pmatrix} 1 & 0 & \dfrac{right+left}{2\,near} & 0 \\ 0 & 1 & \dfrac{top+bottom}{2\,near} & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{66}$$

● 透視投影

透視投影では、物体の見かけの大きさが距離 (深度, z 値) に反比例します。(*x*, *y*, *z*) の位置にある点は *–near* の位置にある前方面の (*–near x* / *z*, *–near y* / *z*) の位置に投影されます。z 値に関しては、□ の透視深度を参照してください。

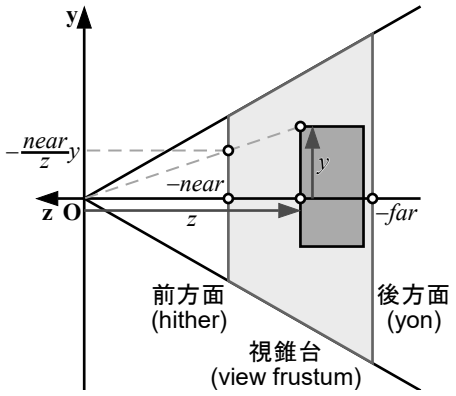


図 105 透視投影

$$\mathbf{M}_{perspective} = \begin{pmatrix} near & 0 & 0 & 0 \\ 0 & near & 0 & 0 \\ 0 & 0 & \frac{far + near}{2} & far\,near \\ 0 & 0 & -1 & 0 \end{pmatrix} \tag{67}$$

● 透視投影変換行列

透視投影変換行列 $\mathbf{M}_p$ は、この二つの変換行列 $\mathbf{M}_{perspective}$ と $\mathbf{M}_{shear}$ の積に、大きさを正規化する変換行列 $\mathbf{M}_{scaling}$ を乗じたものです。

$$\begin{aligned} \mathbf{M}_p &= \mathbf{M}_{scaling}\mathbf{M}_{perspective}\mathbf{M}_{shear} \\ &= \begin{pmatrix} \frac{2\,near}{right - left} & 0 & \frac{right + left}{right - left} & 0 \\ 0 & \frac{2\,near}{top - bottom} & \frac{top + bottom}{top - bottom} & 0 \\ 0 & 0 & -\frac{far + near}{far - near} & -\frac{2\,far\,near}{far - near} \\ 0 & 0 & -1 & 0 \end{pmatrix} \end{aligned} \tag{68}$$

● 透視投影変換行列を作成するメソッドの追加 (Matrix.h)

Matrix クラスに透視投影変換行列を作るメソッド frustum() を追加します。引数に指定した視体積 (*left*, *bottom*)、(*right*, *top*)、(*near*, *far*) から式 (68) によって変換行列を求めます。これもインスタンスを生成せずに呼び出せるように static メソッドにします。

```
// 透視投影変換行列を作成する
static Matrix frustum(GLfloat left, GLfloat right,
  GLfloat bottom, GLfloat top,
```

```
    GLfloat zNear, GLfloat zFar)
  {
    Matrix t;
    const GLfloat dx(right - left);
    const GLfloat dy(top - bottom);
    const GLfloat dz(zFar - zNear);

    if (dx != 0.0f && dy != 0.0f && dz != 0.0f)
    {
      t.loadIdentity();
      t[ 0] = 2.0f * zNear / dx;
      t[ 5] = 2.0f * zNear / dy;
      t[ 8] = (right + left) / dx;
      t[ 9] = (top + bottom) / dy;
      t[10] = -(zFar + zNear) / dz;
      t[11] = -1.0f;
      t[14] = -2.0f * zFar * zNear / dz;
      t[15] = 0.0f;
    }

    return t;
  }
```

● メインプログラム (main.cpp) の変更点

Matrix 型の変数 projection に直交投影変換行列を設定していた orthogonal() を frustum() に置き換えます。前方面の領域 (*left* = -*w* / 2*s*, *right* = *w* / 2*s*, *bottom* = -*h* / 2*s*, *top* = *h* / 2*s*, 図 64) や深度の範囲はそのまま使います。こうすると図が直交投影よりもかなり小さくなってしまいますが、マウスのホイールで拡大することができます。

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

    // 透視投影変換行列を求める
    const GLfloat *const size(window.getSize());
    const GLfloat scale(window.getScale() * 2.0f);
    const GLfloat w(size[0] / scale), h(size[1] / scale);
    const Matrix projection(Matrix::frustum(-w, w, -h, h, 1.0f, 10.0f));

    // モデル変換行列を求める
    const GLfloat *const location(window.getLocation());
    const Matrix model(Matrix::translate(location[0], location[1], 0.0f));
```

■ サンプルプログラム step16

● 実行結果

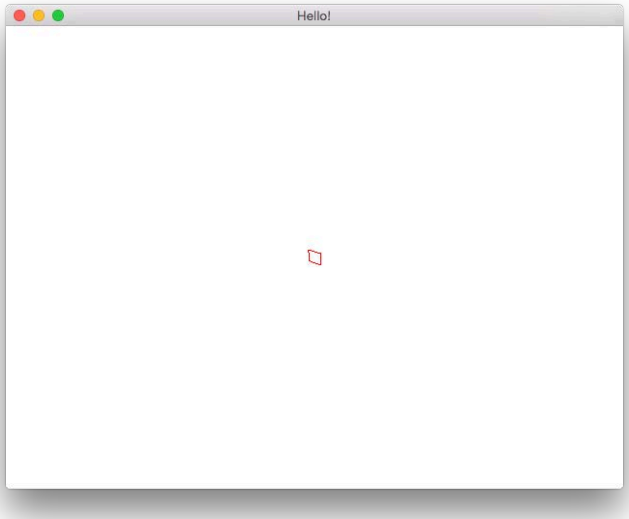


図 106 透視投影変換を実装した実行結果

7.8.4 透視深度

透視投影では、投影像の大きさが深度 (z 値) に反比例します。透視投影後の深度にも、元の深度の逆数を用います。これにより、透視投影後の位置の精度は前方面に近いほど高く、前方面から離れるほど低くなります。このような深度を**透視深度**といいます。

ただし深度が逆数だと、前方面に近いほど大きく、遠くなるにつれて小さくなってしまいます。これは隠面消去処理 (8.4) の際に都合が良くありません。また、この深度を標準視体積 (7.1.4) に収める必要もあります。

そこで、深度 (z 値) の逆数に $-far\ near$ を乗じ、これから $(far + near) / 2$ を減じて 0 点が中央に来るように平行移動します。そして最後に式 (64) の $\mathbf{M}_{scaling}$ により $-2 / (far - near)$ 倍して、$[-1, 1]$ の範囲に正規化します。これによって「視点から遠いほど大きな値」になります。

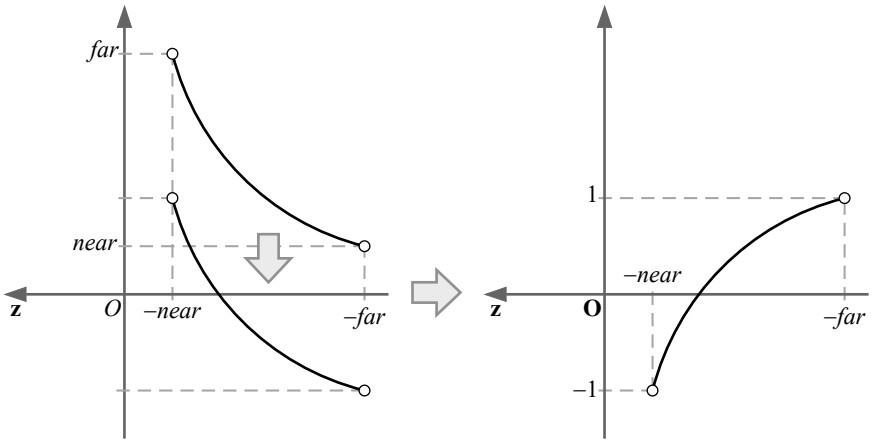


図 107 透視深度の正規化

以上により、この透視投影は式 (69) で表されます。

$$\begin{aligned} x^* &= -\frac{near}{z}x \\ y^* &= -\frac{near}{z}y \\ z^* &= -\frac{far\,near}{z} - \frac{far+near}{2} \end{aligned} \tag{69}$$

この変換を行列で表すと、式 (67) の $\mathbf{M}_{perspective}$ になります。標準座標 (x, y, z) にこの変換を適用すれば、式 (70) より式 (71) に示す同時座標が得られます。

$$\begin{pmatrix} x' \\ y' \\ z' \\ w' \end{pmatrix} = \begin{pmatrix} near & 0 & 0 & 0 \\ 0 & near & 0 & 0 \\ 0 & 0 & \frac{far+near}{2} & far\,near \\ 0 & 0 & -1 & 0 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} \tag{70}$$

$$\begin{aligned} x' &= near\,x \\ y' &= near\,y \\ z' &= \frac{far+near}{2}z + far\,near \\ w' &= -z \end{aligned} \tag{71}$$

これから標準座標 (x^*, y^*, z^*) を求めれば、式 (69) が得られます。

7.8.5 投影面上の線形補間

いま図 108 において、xz 平面上の点 $A = (x_a, z_a)$ および $B = (x_b, z_b)$ が $z = -near$ の位置にある前方面に投影された位置を、それぞれ C および D とします。

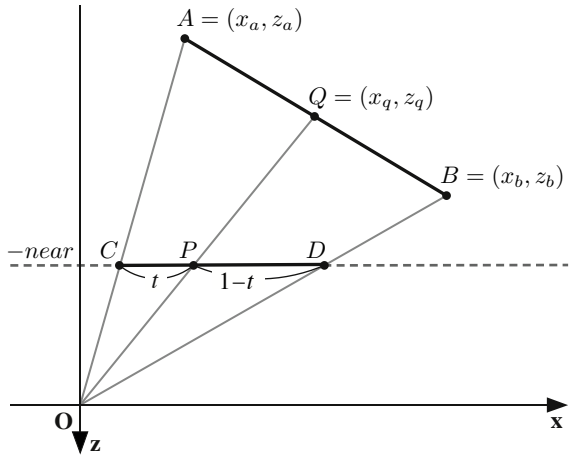


図 108 投影面上の線形補間

この線分 CD を $t : 1-t$ で内分する点を P とするとき、P を AB 上に逆透視投影した点の位置 $Q = (x_q, z_q)$ は、式 (72) により求めることができます。

$$\begin{cases} x_q = \dfrac{\dfrac{x_a}{z_a}(1-t)+\dfrac{x_b}{z_b}t}{\dfrac{1}{z_a}(1-t)+\dfrac{1}{z_b}t} \\ z_q = \dfrac{1}{\dfrac{1}{z_a}(1-t)+\dfrac{1}{z_b}t} \end{cases} \tag{72}$$

式 (72) の Q の z 成分 z_q は、A の z 成分 z_a と B の z 成分 z_b の**逆数を線形補間したものの逆数**になります。ラスタ化処理 (4.1) では画面上の画素の位置における深度を求める必要がありますが、これは透視深度を線形補間したものの逆数として得られます。

Q の x 成分 x_q の分子は、この点の z = 1 の投影面上での位置を線形補間により求めたものです。一方、分母は Q の透視深度、すなわち z_q の逆数です。

7.8.6 画角と縦横比にもとづく透視投影変換行列

視錐台の指定による透視投影変換行列の設定は、汎用性は高いものの、直感的には理解しづらいものがあります。そこで、カメラのレンズの y 方向の**画角** (*fovy*) と画面の**縦横比** (*aspect ratio*) を使って透視投影変換行列を設定する方法がしばしば用いられます。

式 (73) の f は画角 *fovy* で上下端が±1 の投影面に投影した時の、投影面までの距離 (焦点距離) です。したがって $z = -near$ の位置にある前方面上の表示領域は、$-bottom = top = near / f$、$-left = right = aspect\ near / f$ となります。これを式 (68) に代入すれば、式 (74) が得られます。

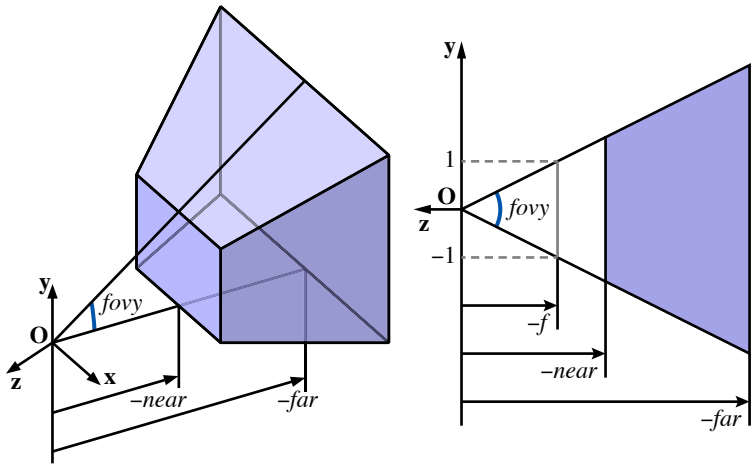


図 109 画角と縦横比にもとづく透視投影変換行列

$$f = \frac{1}{\tan\left(\dfrac{fovy}{2}\right)} = \cot\left(\frac{fovy}{2}\right) \tag{73}$$

$$\mathbf{M}_p = \begin{pmatrix} \dfrac{f}{aspect} & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & -\dfrac{far+near}{far-near} & -\dfrac{2\,far\,near}{far-near} \\ 0 & 0 & -1 & 0 \end{pmatrix} \tag{74}$$

● 画角をもとに透視投影変換行列を作成するメソッドの追加 (Matrix.h)

Matrix クラスに画角を使って透視投影変換行列を作るメソッド perspective() を追加します。引数に指定した画面の y 方向の画角 *fovy*、画面の縦横比 *aspect*、および前方面と後方面の位置 (*near*,*far*) から式 (74) によって変換行列を求めます。これもインスタンスを生成せずに呼び出せるように static メソッドにします。

```
// 画角を指定して透視投影変換行列を作成する
static Matrix perspective(GLfloat fovy, GLfloat aspect,
  GLfloat zNear, GLfloat zFar)
{
  Matrix t;
  const GLfloat dz(zFar - zNear);

  if (dz != 0.0f)
  {
    t.loadIdentity();
    t[ 5] = 1.0f / tan(fovy * 0.5f);
    t[ 0] = t[5] / aspect;
    t[10] = -(zFar + zNear) / dz;
    t[11] = -1.0f;
    t[14] = -2.0f * zFar * zNear / dz;
    t[15] = 0.0f;
  }

  return t;
}
```

● メインプログラム (main.cpp) の変更点

画角 fovy には Window クラスのメンバ変数 scale を代用します。このため、マウスのホイールがズーム操作に相当することになります。scale の初期値は 100 なので、これが $60° = \pi/6$ になるよう $0.6\pi/180 \fallingdotseq 0.01$ を乗じて fovy とします。縦横比 aspect には画面上のウィンドウの高さ h に対する幅 w の割合を用います。

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

  // 透視投影変換行列を求める
  const GLfloat *const size(window.getSize());
  const GLfloat fovy(window.getScale() * 0.01f);
  const GLfloat aspect(size[0] / size[1]);
  const Matrix projection(Matrix::perspective(fovy, aspect, 1.0f, 10.0f));
```

```
    // モデル変換行列を求める
    const GLfloat *const location(window.getLocation());
    const Matrix model(Matrix::translate(location[0], location[1], 0.0f));
```

■ サンプルプログラム step17

● 実行結果

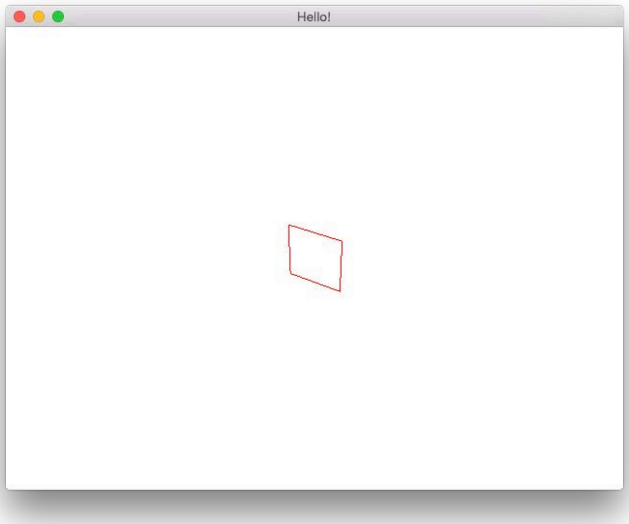


図 110 透視投影変換を実装した実行結果

第8章 形状の表現

8.1 三次元図形の描画

これまでは xy 平面上の二次元図形 (正方形) を描いてきました。実際には、これは三次元空間中の $z=0$ の平面上の図形です。ここでは三次元の図形を描きます。

8.1.1 一筆書きによる描画

図 111 の正八面体を線分で描くことを考えます。図形は基本図形 (図 55) に GL_LINE_LOOP (閉じた折れ線) を指定して、glDrawArrays() により描画しています (5.2.5)。この場合、頂点のデータを一筆書きの要領で描画する順に並べておく必要があります。図 111 の正八面体を一筆書きするなら、例えば⓪①③④⓪②③⑤①②④⑤番の順に頂点をたどり、最後に⓪番に戻ります。

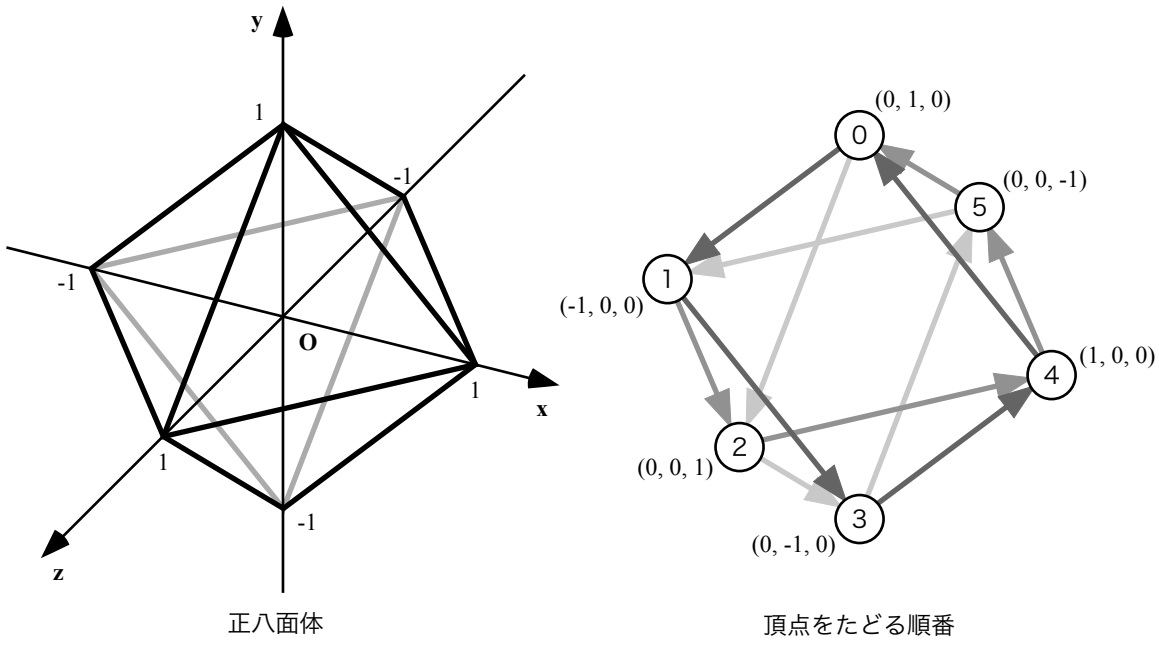


図 111 描画する正八面体

● Object クラス (Object.h) の変更点

頂点の位置が三次元なので、Object クラスの中で定義している Vertex 構造体のメンバ position の要素数を 3 に変更します。

```
// 図形データ
class Object
{
  《省略》

  // 頂点属性
  struct Vertex
  {
    // 位置
    GLfloat position[3];
  };
```

● メインプログラム (main.cpp) の変更点

main() 関数の前に八面体の頂点の位置のデータ octahedronVertex を追加し、これを使って Shape クラスのインスタンスを作成します。これをループの中で描画します。この位置データの 1 頂点あたりの要素数は 3、頂点数は 12 になります。

```
// 八面体の頂点の位置
constexpr Object::Vertex octahedronVertex[] =
{
  {  0.0f,  1.0f,  0.0f },
  { -1.0f,  0.0f,  0.0f },
  {  0.0f, -1.0f,  0.0f },
  {  1.0f,  0.0f,  0.0f },
  {  0.0f,  1.0f,  0.0f },
  {  0.0f,  0.0f,  1.0f },
  {  0.0f, -1.0f,  0.0f },
  {  0.0f,  0.0f, -1.0f },
  { -1.0f,  0.0f,  0.0f },
  {  0.0f,  0.0f,  1.0f },
  {  1.0f,  0.0f,  0.0f },
  {  0.0f,  0.0f, -1.0f }
};

int main()
{
  《省略》

  // 図形データを作成する
  std::unique_ptr<const Shape> shape(new Shape(3, 12, octahedronVertex));

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

■ サンプルプログラム step18

● 実行結果

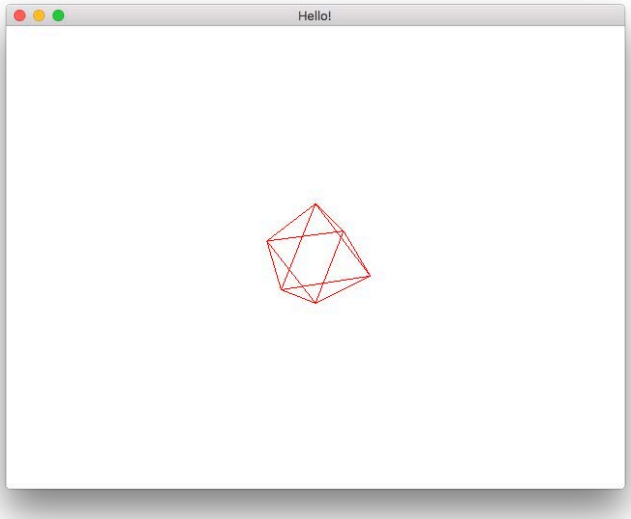


図 112 正八面体の線画による描画

8.1.2 インデックスを用いた描画

次に図 113 の立方体 (正六面体) を描いてみます。ところが、この図形は一筆書きできないため、描画する基本図形 (図 55) として八面体のように GL_LINE_LOOP を使うことができません。そこで、ここでは GL_LINES を使うことにします。

GL_LINES は二つの頂点を結んで一本の独立した線分を描きます。図 113 の六面体の 12 本の稜線を描くには、合計で 24 個の頂点属性が必要になります。実際の頂点は 8 個しかありませんから、個々の頂点はそれぞれ 3 回使われることになります。

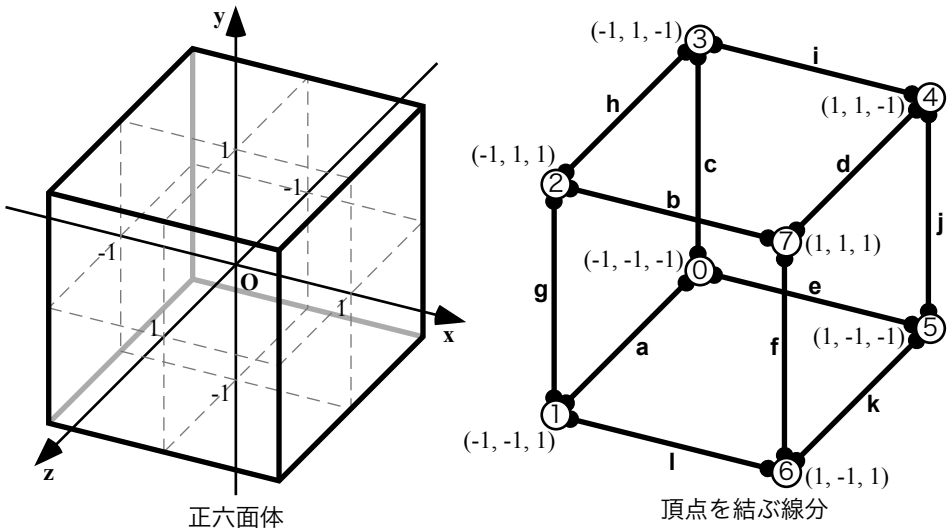


図 113 描画する正六面体

しかし、頂点が 8 個しかないのに 24 個の頂点を使うのはもったいないように思えます。そこ

で 8 個の頂点の位置の情報 (頂点属性) と、どの頂点を結んで 12 本の線分を描くかという**インデックス**を組み合わせて、この六面体の図形を表現します。図 113 の六面体のデータは、表 6 の頂点表と稜線表により表されます。ちなみに、頂点の位置の情報は**計量情報**あるいは**幾何情報**（ジオメトリ情報)、インデックスの情報は**位相情報**（トポロジ情報）と呼ぶことがあります。

表 6 頂点表と稜線表

頂点	位置		
	x	y	z
0	-1	-1	-1
1	-1	-1	1
2	-1	1	1
3	-1	1	-1
4	1	1	-1
5	1	-1	-1
6	1	-1	1
7	1	1	1

頂点表

稜線	インデックス	
	始点	終点
a	1	0
b	2	7
c	3	0
d	4	7
e	5	0
f	6	7
g	1	2
h	2	3
i	3	4
j	4	5
k	5	6
l	6	1

稜線表

● Object クラス (Object.h) の変更点

頂点のインデックスを使って図形を描画する場合は、これも頂点バッファオブジェクトに格納し、それを頂点配列オブジェクトに組み込みます。まず Object クラスにインデックスの頂点バッファオブジェクト名 (番号) を保持する ibo を private メンバに追加します。

```
// 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;

  // インデックスの頂点バッファオブジェクト
  GLuint ibo;
```

次に、コンストラクタの引数に頂点のインデックスの数 (描画する頂点数) indexcount と頂点のインデックスの配列 index を追加し、コンストラクタ内でインデックスの頂点バッファオブジェ

クトを作成します。これらの引数にはデフォルト引数を設定しておきます。これにより、これまでのコンストラクタの呼び出しに対するソースプログラム上の互換性を保つことができます。

頂点のインデックスに用いるバッファオブジェクトは、glBufferData() の第 1 引数 target に GL_ELEMENT_ARRAY_BUFFER を指定して作成します。この第 2 引数 size には確保する頂点バッファオブジェクトのサイズを指定します。indexcount が 0 だとこれが 0 になってしまいますが、エラーにはなりません。同様に第 3 引数の index が NULL の場合は glBufferData() はデータの転送を行いませんが (OpenGL は C 言語向けの API なので、仕様上は nullptr ではなく NULL が使用されています)、これもエラーにはなりません。

この頂点バッファオブジェクトも、このクラスの頂点配列オブジェクトに組み込まれますが、頂点のインデックスは attribute 変数としては参照しないので、この頂点バッファオブジェクトには attribute 変数を割り当てません。

```
  // 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
  //   indexcount: 頂点のインデックスの要素数
  //   index: 頂点のインデックスを格納した配列
  Object(GLint size, GLsizei vertexcount, const Vertex *vertex,
    GLsizei indexcount = 0, const GLuint *index = NULL)
  {
    // 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);

    // インデックスの頂点バッファオブジェクト
    glGenBuffers(1, &ibo);
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, ibo);
    glBufferData(GL_ELEMENT_ARRAY_BUFFER,
      indexcount * sizeof (GLuint), index, GL_STATIC_DRAW);
  }
```

デストラクタではインデックスの頂点バッファオブジェクトを忘れずに削除します。

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

```
    // 頂点バッファオブジェクトを削除する
    glDeleteBuffers(1, &vbo);

    // インデックスの頂点バッファオブジェクトを削除する
    glDeleteBuffers(1, &ibo);
  }
```

● Shape クラス (Shape.h) の変更点

Object クラスのコンストラクタに引数を追加したので、これを参照している Shape クラスも変更します。コンストラクタの引数に頂点のインデックスの数 (描画する頂点数) indexcount と頂点のインデックスの配列 index を追加し、Object クラスのコンストラクタの引数に追加します。Object クラスの場合と同様に、Shape クラスに追加した引数にデフォルト引数を設定し、これまでのコンストラクタの呼び出しに対するソースプログラム上の互換性を保ちます。

```
  // 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
  //   indexcount: 頂点のインデックスの要素数
  //   index: 頂点のインデックスを格納した配列
  Shape(GLint size, GLsizei vertexcount, const Object::Vertex *vertex,
    GLsizei indexcount = 0, const GLuint *index = NULL)
    : object(new Object(size, vertexcount, vertex, indexcount, index))
    , vertexcount(vertexcount)
  {
  }
```

インデックスを使って図形の描画を行うクラス ShapeIndex (ShapeIndex.h)

次に、六面体を線画で描画するクラス ShapeIndex を、ShapeIndex.h というヘッダファイルに Shape クラスを継承して定義します。インデックスを使って描画するにはインデックスの数が必要になるので、これを保持する indexcount という const メンバを用意します。これも派生クラスから参照するので、protected にします。

```
#pragma once

// 図形の描画
#include "Shape.h"

// インデックスを使った図形の描画
class ShapeIndex
  : public Shape
{
protected:

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

このコンストラクタでは、引数で受け取った頂点属性を使って基底クラスの Shape のコンス

トラクタを実行します。これにより、Object クラスのインスタンスが生成されます。const メンバの indexcount もここで初期化します。なお、このコンストラクタに本体はありません。

```
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
  //   indexcount: 頂点のインデックスの要素数
  //   index: 頂点のインデックスを格納した配列
  ShapeIndex(GLint size, GLsizei vertexcount, const Object::Vertex *vertex,
    GLsizei indexcount, const GLuint *index)
    : Shape(size, vertexcount, vertex, indexcount, index)
    , indexcount(indexcount)
  {
  }
```

描画を実行する execute() メソッドオーバーライドして、glDrawElements() を呼び出します。

```
  // 描画の実行
  virtual void execute() const
  {
    // 線分群で描画する
    glDrawElements(GL_LINES, indexcount, GL_UNSIGNED_INT, 0);
  }
};
```

void glDrawElements(GLenum mode, GLsizei count, GLenum type, const GLvoid *indices)

インデックスを参照して頂点配列による図形の描画を行います。

mode

描画する基本図形の種類 (図 55)。

count

描画する頂点の数。たとえば四角形なら 4。

type

インデックスのデータ型。インデックスが GLuint 型なら GL_UNSIGNED_INT。

indices

インデックスのデータが格納されている場所。インデックスの頂点バッファオブジェクトの先頭から使用するなら 0。

● メインプログラム (main.cpp) の変更点

ShapeIndex クラスを定義しているヘッダファイル ShapeIndex.h を main.cpp の冒頭で #include します。

```
#include <cstdlib>
#include <iostream>
```

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

また、main() 関数の前に表 6 の頂点表の六面体の頂点の位置のデータ cubeVertex と稜線表のインデックスのデータ wireCubeIndex を追加します。

```
// 六面体の頂点の位置
constexpr Object::Vertex cubeVertex[] =
{
  { -1.0f, -1.0f, -1.0f },  // (0)
  { -1.0f, -1.0f,  1.0f },  // (1)
  { -1.0f,  1.0f,  1.0f },  // (2)
  { -1.0f,  1.0f, -1.0f },  // (3)
  {  1.0f,  1.0f, -1.0f },  // (4)
  {  1.0f, -1.0f, -1.0f },  // (5)
  {  1.0f, -1.0f,  1.0f },  // (6)
  {  1.0f,  1.0f,  1.0f }   // (7)
};

// 六面体の稜線の両端点のインデックス
constexpr GLuint wireCubeIndex[] =
{
  1, 0, // (a)
  2, 7, // (b)
  3, 0, // (c)
  4, 7, // (d)
  5, 0, // (e)
  6, 7, // (f)
  1, 2, // (g)
  2, 3, // (h)
  3, 4, // (i)
  4, 5, // (j)
  5, 6, // (k)
  6, 1  // (l)
};
```

main() 関数では ShapeIndex クラスのコンストラクタの引数に六面体の頂点の位置 cubeVertex と稜線の両端点のインデックス wireCubeIndex を指定して、インスタンスを一つ生成します。描画する頂点の数は実際の頂点の数の 8 ではなく、描画に使われる線分の両端点の合計、すなわちインデックスの要素数の 24 になります。これをループの中で描画します。

```
int main()
{
  《省略》

  // 図形データを作成する
  std::unique_ptr<const Shape> shape(new ShapeIndex(3, 8, cubeVertex,
```

```
    24, wireCubeIndex));

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

■ サンプルプログラム step19

● 実行結果

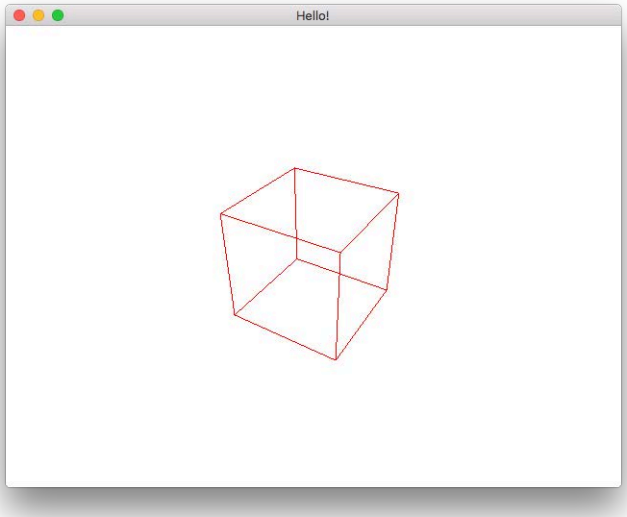


図 114 正六面体の線画による描画

8.2 頂点色の指定

画面に表示する図形の色は、フラグメントシェーダにおいて、フレームバッファのカラーバッファに割り当てた out 変数 (0) に色を代入することにより決定されます。フラグメントシェーダはバーテックスシェーダの出力を画素の位置で補間した値を受け取る (4.1) ことができるので、バーテックスシェーダの入力、すなわち頂点属性に色のデータを追加することで、描画する図形の色を指定することができます。

● Object クラス (Object.h) の変更点

頂点属性として位置の他に色を使いますから、Vertex 構造体に色のデータを保持するメンバ color を追加します。(赤, 緑, 青) の三原色で表すので、この要素数は 3 にします。

```
// 図形データ
class Object
{
```

```
《省略》

// 頂点属性
struct Vertex
{
  // 位置
  GLfloat position[3];

  // 色
  GLfloat color[3];
};
```

頂点属性のデータ構造を変更したので、コンストラクタで行なっている頂点バッファオブジェクトと attribute 変数との関連づけ (5.2.3) も変更します。位置は場所が 0 の attribute 変数に関連づけていますから、色は場所が 1 の attribute 変数に関連づけることにします。

この引数 vertex に指定したメモリには、頂点の位置のデータのメンバ position と色のデータのメンバ color が図 115 のように格納されています。このとき、頂点ごとのデータの間隔 stride は vertex[0].position から vertex[1].position までの間隔ですから、配列変数 vertex の一つの要素の大きさになります。これは sizeof (Vertex) で得られます。これは color も同じです。

一方、vertex の先頭のデータ (vertex[0]) の各メンバが格納されている位置は、position メンバは vertex->position、color メンバは vertex->color です。ただし、この vertex は CPU 上のメモリを指しているのに対して、データの格納先の頂点バッファオブジェクトは GPU 上のメモリです。後者の先頭は 0 なので、0 を Vertex 型のポインタ (Vertex *) にキャスト (型変換) することにより、offset はそれぞれ static_cast<Vertex *>(0)->position、static_cast<Vertex *>(0)->color で得られます。なお、position メンバは Vertex 型の先頭にあるので、static_cast<Vertex *>(0)->position は実質的には 0 です。

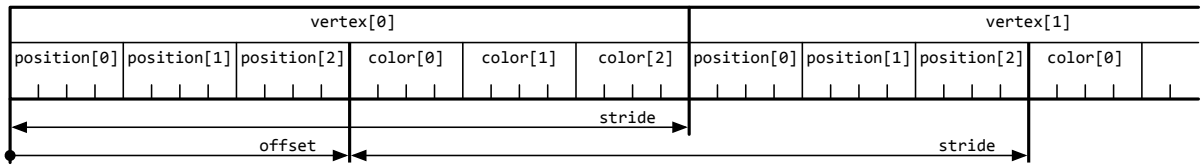


図 115 頂点データの構造体 Vertex 中の位置データ position と色データ color の配置

```
// 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
//   indexcount: 頂点のインデックスの要素数
//   index: 頂点のインデックスを格納した配列
Object(GLint size, GLsizei vertexcount, const Vertex *vertex,
  GLsizei indexcount, const GLuint *index)
{
  // 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, sizeof (Vertex),
      static_cast<Vertex *>(0)->position);
    glEnableVertexAttribArray(0);
    glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, sizeof (Vertex),
      static_cast<Vertex *>(0)->color);
    glEnableVertexAttribArray(1);
```

● メインプログラム (main.cpp) の変更点

頂点属性として位置の他に色を受け取るために、バーテックスシェーダの attribute 変数 color を追加します。color の attribute 変数の場所は、前述の通り 1 にします。

```
// 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());

   《省略》

  // プログラムオブジェクトをリンクする
  glBindAttribLocation(program, 0, "position");
  glBindAttribLocation(program, 1, "color");
  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;
}
```

main() 関数の前に置いた六面体の頂点のデータ cubeVertex に色のデータを追加します。色は (赤, 緑, 青) の明るさをそれぞれ 0 ～ 1 の値で設定します。(0, 0, 0) で黒、(1, 1, 1) で白、(0.8, 0, 0) で少し暗めの赤になります。

```
// 六面体の頂点の位置と色
constexpr Object::Vertex cubeVertex[] =
{
  { -1.0f, -1.0f, -1.0f,  0.0f,  0.0f,  0.0f },  // (0)
  { -1.0f, -1.0f,  1.0f,  0.0f,  0.0f,  0.8f },  // (1)
  { -1.0f,  1.0f,  1.0f,  0.0f,  0.8f,  0.0f },  // (2)
```

```
  { -1.0f,  1.0f, -1.0f,  0.0f,  0.8f,  0.8f },  // (3)
  {  1.0f,  1.0f, -1.0f,  0.8f,  0.0f,  0.0f },  // (4)
  {  1.0f, -1.0f, -1.0f,  0.8f,  0.0f,  0.8f },  // (5)
  {  1.0f, -1.0f,  1.0f,  0.8f,  0.8f,  0.0f },  // (6)
  {  1.0f,  1.0f,  1.0f,  0.8f,  0.8f,  0.8f }   // (7)
};

// 六面体の稜線の両端点のインデックス
constexpr GLuint wireCubeIndex[] =
{
  《省略》
};
```

● バーテックスシェーダ (point.vert) の変更点

バーテックスシェーダでは、頂点属性に追加した頂点色のデータを受け取る in 変数 color を宣言します。in 変数はシェーダプログラムがレンダリングパイプラインの前のステージデータを受け取る変数で、バーテックスシェーダでは頂点バッファオブジェクトのデータが格納されます。ここでは、これをそのまま out 変数 vertex_color に代入します。out 変数に格納された内容は、レンダリングパイプラインの次のステージ (この場合はラスタライザ) に送られます。

```
#version 150 core
uniform mat4 modelview;
uniform mat4 projection;
in vec4 position;
in vec4 color;
out vec4 vertex_color;
void main()
{
  vertex_color = color;
  gl_Position = projection * modelview * position;
}
```

● フラグメントシェーダ (point.frag) の変更点

フラグメントシェーダでは、バーテックスシェーダからラスタライザに送られ、ラスタライザによって補間された頂点色を受け取る in 変数 vertex_color を宣言します。これはバーテックスシェーダの out 変数と対応している必要があります。これをそのまま out 変数 fragment に代入して、フラグメントシェーダの次のステージであるフレームバッファに出力します。

なお、この vertex_color の値はラスタライザによって補間されているため、この変数の内容はバーテックスシェーダで代入された値と同じになるとは限りません。したがって、このようにしてシェーダプログラム間で受け渡しされる変数を、特に varying 変数と呼びます。

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

```
  fragment = vertex_color;
}
```

■ サンプルプログラム step20

このようにして頂点に色をつけると、図 116 のように線分にグラデーションが付きます。一つの線分には二つの端点がありますから、線分上の色はそれらを**補間**したものになります。

● **実行結果**

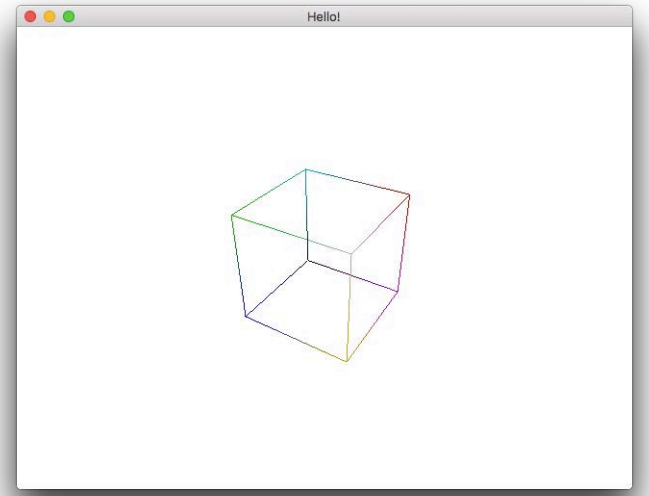


図 116 頂点に色をつけた場合の描画

補足：頂点色を補間しないようにする

線分にグラデーションをつけないなら、両端点の色を同じにするか、varying 変数の宣言に flat という修飾子を付けます。この場合は最後に描画に使われた頂点 (終点) の色が使われます。

● **バーテックスシェーダ**

```
#version 150 core
uniform mat4 modelview;
uniform mat4 projection;
in vec4 position;
in vec4 color;
flat out vec4 vertex_color;
void main()
{
  vertex_color = color;
  gl_Position = projection * modelview * position;
}
```

● フラグメントシェーダ

```
#version 150 core
flat in vec4 vertex_color;
out vec4 fragment;
void main()
{
  fragment = vertex_color;
}
```

● 実行結果

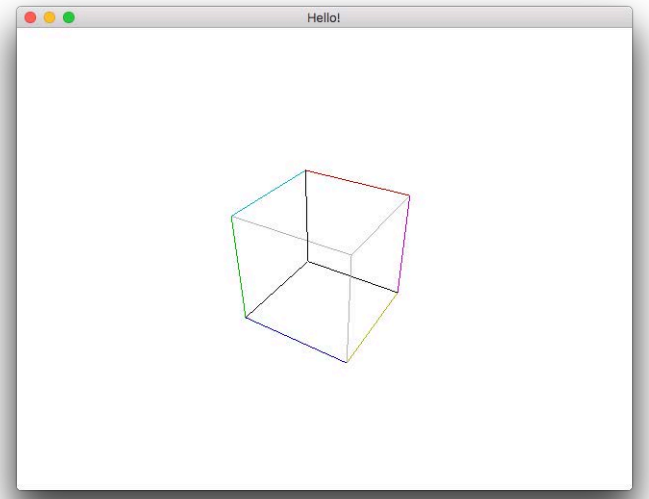


図 117 頂点色を補間しない場合の描画結果

補足：attribute 変数の場所の指定

attribute 変数の場所 (番号) は、前述のようにシェーダのプログラムオブジェクトをリンクする直前に glBindAttribLocation() で指定する方法の他に、リンク時には指定せずに (その場合は適当な場所が自動的に割り振られます) リンク後の任意の時点で glGetAttribLocation() により得ることができます。

GLint glGetAttribLocation (GLuint program, 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 変数の場所を調べます。戻り値は attribute 変数 name の場所です。name に指定した attribute 変数が program に指定したプログラムオブジェクトの中に見つからなければ -1 を返します。

program

attribute 変数の場所を調べるプログラムオブジェクト名 (番号)。

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。

GLSL のバージョン 3.3 以降 (#version 330) であれば、このバーテックスシェーダの attribute 変数の場所やフラグメントシェーダの出力変数の場所は、シェーダのソースプログラム中で変数を宣言する際に layout 修飾子を使って指定することができます。また、この機能は GLSL のバージョン 3.2 でも GPU の拡張機能として用意されている場合があります。その場合はバーテックスシェーダのソースプログラムの冒頭に #extension GL_ARB_explicit_attrib_location: enable を付ければ、この機能を使用できるようになります。

● バーテックスシェーダ

```
#version 150 core
#extension GL_ARB_explicit_attrib_location: enable
uniform mat4 modelview;
uniform mat4 projection;
layout (location = 0) in vec4 position;
layout (location = 1) in vec4 color;
out vec4 vertex_color;
void main()
{
  vertex_color = color;
  gl_Position = projection * modelview * position;
}
```

● フラグメントシェーダ

```
#version 150 core
#extension GL_ARB_explicit_attrib_location: enable
in vec4 vertex_color;
layout (location = 0) out vec4 fragment;
void main()
{
  fragment = vertex_color;
}
```

8.3 図形の塗りつぶし

8.3.1 三角形の描画

GL_LINES や GL_LINE_STRIP、GL_LINE_LOOP はいずれも線分の描画を行います (図 55)。図形を塗りつぶす場合には、GL_TRIANGLES や GL_TRIANGLE_STRIP、GL_TRIANGLE_FAN などを使用します。いずれも三角形を描くので、図形のデータは三角形に分割されている必要があります。図 113 の正六面体を描くには、例えば図 118 の三角形の組み合わせで表現します。

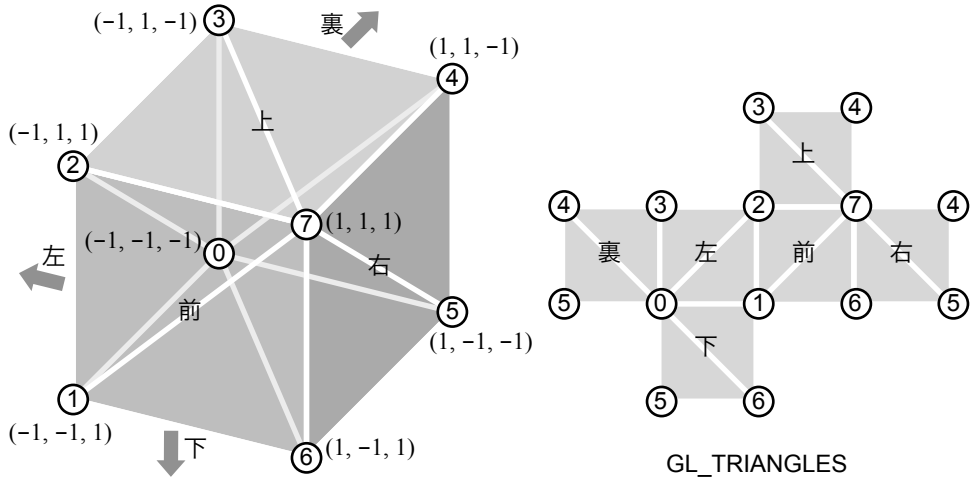


図 118 三角形で表現した正六面体

このデータは表 7 の頂点表と三角形表のように表すことができます。

表 7 頂点表と三角形表

頂点	位置		
	x	y	z
0	-1	-1	-1
1	-1	-1	1
2	-1	1	1
3	-1	1	-1
4	1	1	-1
5	1	-1	-1
6	1	-1	1
7	1	1	1

頂点表

面	インデックス					
	三角形 (1)			三角形 (2)		
左	0	1	2	0	2	3
裏	0	3	4	0	4	5
下	0	5	6	0	6	1
右	7	6	5	7	5	4
上	7	4	3	7	3	2
前	7	2	1	7	1	6

三角形表

● インデックスを使って三角形による描画を行うクラス SolidShapeIndex (SolidShapeIndex.h)

インデックスを使って六面体を 12 枚の三角形で描画するクラス SolidShapeIndex を、SolidShapeIndex.h というヘッダファイルに ShapeIndex クラスを継承して定義します。このクラスは ShapeIndex クラスの execute() メソッドのオーバーライドだけを行います。このコンストラクタの引数は、そのまま ShapeIndex のコンストラクタに渡します。また、三角形を描画するので、execute() で実行する glDrawElements() の第一引数 mode を GL_TRIANGLES にします。

```
#pragma once

// インデックスを使った図形の描画
#include "ShapeIndex.h"

// インデックスを使った三角形による描画
```

```
class SolidShapeIndex
  : public ShapeIndex
{
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
  //   indexcount: 頂点のインデックスの要素数
  //   index: 頂点のインデックスを格納した配列
  SolidShapeIndex(GLint size, GLsizei vertexcount, const Object::Vertex *vertex,
    GLsizei indexcount, const GLuint *index)
    : ShapeIndex(size, vertexcount, vertex, indexcount, index)
  {
  }

  // 描画の実行
  virtual void execute() const
  {
    // 三角形で描画する
    glDrawElements(GL_TRIANGLES, indexcount, GL_UNSIGNED_INT, 0);
  }
};
```

● メインプログラム (main.cpp) の変更点

SolidShapeIndex クラスを定義しているヘッダファイル SolidShapeIndex.h を main.cpp の冒頭で #include します。

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

main() 関数より前に、六面体を塗りつぶす三角形の頂点のインデックス (表 7 の三角形表) のデータ solidCubeIndex を追加します。そして main() 内でこれと六面体の頂点の位置 cubeVertex を使って SolidShapeIndex クラスのインスタンスを生成します。データの頂点の数は 8 個ですが、描画する頂点の数 (インデックスの数) は三角形が 12 枚なので 36 個になります。

```
// 六面体の面を塗りつぶす三角形の頂点のインデックス
constexpr GLuint solidCubeIndex[] =
{
  0,  1,  2,  0,  2,  3, // 左
  0,  3,  4,  0,  4,  5, // 裏
```

```
  0,  5,  6,  0,  6,  1, // 下
  7,  6,  5,  7,  5,  4, // 右
  7,  4,  3,  7,  3,  2, // 上
  7,  2,  1,  7,  1,  6  // 前
};

int main()
{
  《省略》

  // 図形データを作成する
  std::unique_ptr<const Shape> shape(new SolidShapeIndex(3, 8, cubeVertex,
    36, solidCubeIndex));

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

■ サンプルプログラム step21

● 実行結果

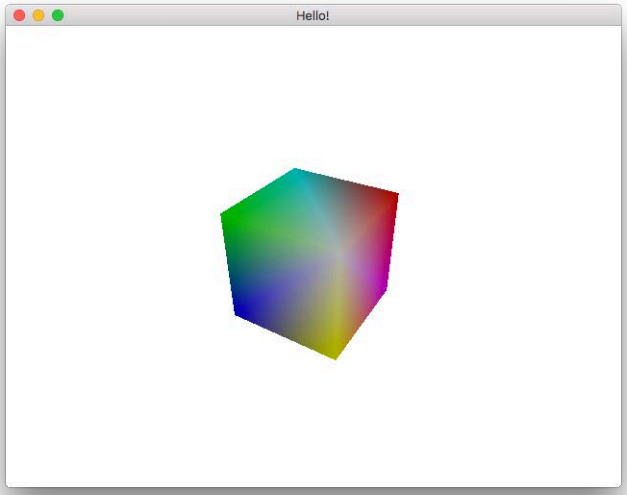


図 119 三角形を使って描画した結果

8.3.2 面単位の色の塗り分け

三角形の頂点に色をつけた場合は、図 119 のように、三角形の内部が頂点色を補間した色で塗りつぶされます。しかし、これでは図形が立方体なのにもかかわらず、面の境界がはっきりしないものになってしまいます。そこで、面ごとに異なる色を付けようと思います。

ところが、図 118 のように頂点を複数の面で共有していると、面ごとに頂点色を変えることができません。頂点色も位置と同じく頂点属性なので、位置が同じでも色が違えば、頂点属性は異

なるものになります。

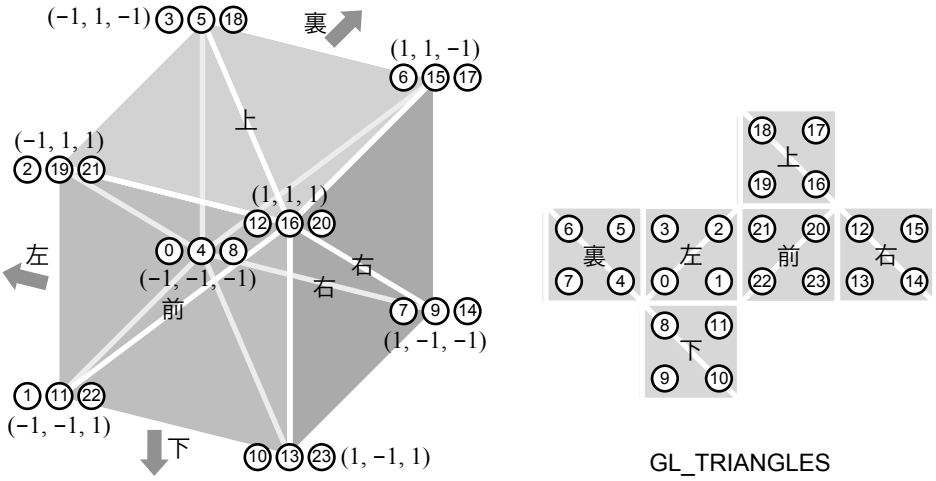


図 120 インデックスを使って六面体の各面を GL_TRIANGLES で表す場合

このような立方体を描くには、例えば図 120 のように一つの面を独立した 2 枚の三角形で描くという方法が考えられます。このとき、データの頂点の数は 4 頂点×6 面=24 個になります。描画する頂点の数は 36 個です。

● メインプログラム (main.cpp) の変更点

main() 関数より前に、面ごとに色を変えた六面体の頂点属性のデータ solidCubeVertex を追加します。また、三角形の頂点のインデックスのデータ solidCubeIndex も、それに合わせて面ごとに独立する (同じ頂点番号を持たない) ように変更します。

```
// 面ごとに色を変えた六面体の頂点属性
constexpr Object::Vertex solidCubeVertex[] =
{
  // 左
  { -1.0f, -1.0f, -1.0f,  0.1f,  0.8f,  0.1f },
  { -1.0f, -1.0f,  1.0f,  0.1f,  0.8f,  0.1f },
  { -1.0f,  1.0f,  1.0f,  0.1f,  0.8f,  0.1f },
  { -1.0f,  1.0f, -1.0f,  0.1f,  0.8f,  0.1f },

  // 裏
  {  1.0f, -1.0f, -1.0f,  0.8f,  0.1f,  0.8f },
  { -1.0f, -1.0f, -1.0f,  0.8f,  0.1f,  0.8f },
  { -1.0f,  1.0f, -1.0f,  0.8f,  0.1f,  0.8f },
  {  1.0f,  1.0f, -1.0f,  0.8f,  0.1f,  0.8f },

  // 下
  { -1.0f, -1.0f, -1.0f,  0.1f,  0.8f,  0.8f },
  {  1.0f, -1.0f, -1.0f,  0.1f,  0.8f,  0.8f },
  {  1.0f, -1.0f,  1.0f,  0.1f,  0.8f,  0.8f },
  { -1.0f, -1.0f,  1.0f,  0.1f,  0.8f,  0.8f },

  // 右
  {  1.0f, -1.0f,  1.0f,  0.1f,  0.1f,  0.8f },
```

```
  {  1.0f, -1.0f, -1.0f,  0.1f,  0.1f,  0.8f },
  {  1.0f,  1.0f, -1.0f,  0.1f,  0.1f,  0.8f },
  {  1.0f,  1.0f,  1.0f,  0.1f,  0.1f,  0.8f },

  // 上
  { -1.0f,  1.0f, -1.0f,  0.8f,  0.1f,  0.1f },
  { -1.0f,  1.0f,  1.0f,  0.8f,  0.1f,  0.1f },
  {  1.0f,  1.0f,  1.0f,  0.8f,  0.1f,  0.1f },
  {  1.0f,  1.0f, -1.0f,  0.8f,  0.1f,  0.1f },

  // 前
  { -1.0f, -1.0f,  1.0f,  0.8f,  0.8f,  0.1f },
  {  1.0f, -1.0f,  1.0f,  0.8f,  0.8f,  0.1f },
  {  1.0f,  1.0f,  1.0f,  0.8f,  0.8f,  0.1f },
  { -1.0f,  1.0f,  1.0f,  0.8f,  0.8f,  0.1f }
};

// 六面体の面を塗りつぶす三角形の頂点のインデックス
constexpr GLuint solidCubeIndex[] =
{
   0,  1,  2,  0,  2,  3, // 左
   4,  5,  6,  4,  6,  7, // 裏
   8,  9, 10,  8, 10, 11, // 下
  12, 13, 14, 12, 14, 15, // 右
  16, 17, 18, 16, 18, 19, // 上
  20, 21, 22, 20, 22, 23  // 前
};
```

main() 関数では solidCubeVertex と solidCubeIndex を使って SolidShapeIndex クラスのインスタンスを生成します。

```
int main()
{
  《省略》

  // 図形データを作成する
  std::unique_ptr<const Shape> shape(new SolidShapeIndex(3, 24, solidCubeVertex,
    36, solidCubeIndex));

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

■ サンプルプログラム step22

● 実行結果

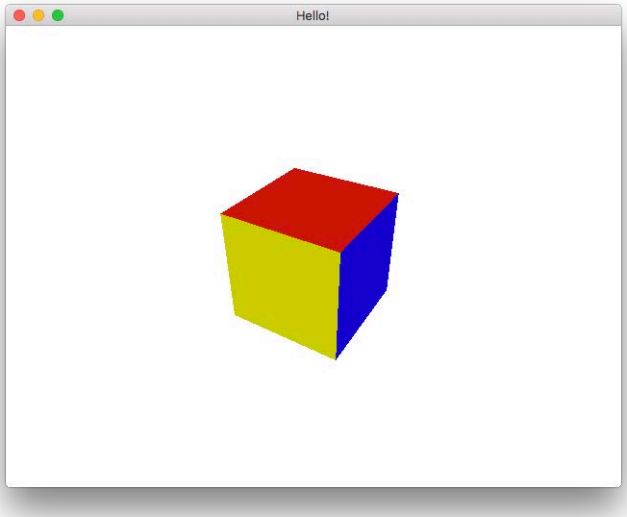


図 121 面ごとに色を変えて描画した結果

8.3.3 インデックスを使わずに描画

面ごとに色を変えた六面体を、今度はインデックスを使わずに描くことを考えます。この場合、全ての頂点を独立して扱うので、データの頂点の数は描画する頂点の数と同じ 36 個になります。

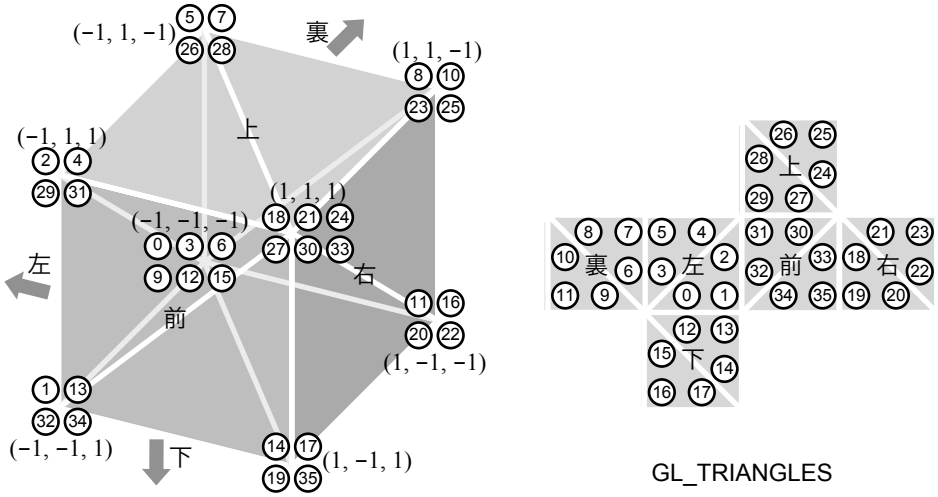


図 122 インデックスを使わずに六面体の各面を GL_TRIANGLES で表す場合

● 複数の三角形による描画を行うクラス SolidShape (SolidShape.h)

今度はこの六面体を、インデックスを使わずに 12 枚の三角形で描画するクラス SolidShape を、SolidShape.h というヘッダファイルに Shape クラスを継承して定義します。このクラスは Shape クラスの execute() メソッドのオーバーライドだけを行います。したがってコンストラクタは引数をそのまま Shape のコンストラクタに渡します。また、三角形を描画するので、execute()

メソッドで実行する glDrawArrays() の第一引数 mode を GL_TRIANGLES にします。

```
#pragma once

// 図形の描画
#include "Shape.h"

// 三角形による描画
class SolidShape
  : public Shape
{
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
  SolidShape(GLint size, GLsizei vertexcount, const Object::Vertex *vertex)
    : Shape(size, vertexcount, vertex)
  {
  }

  // 描画の実行
  virtual void execute() const
  {
    // 三角形で描画する
    glDrawArrays(GL_TRIANGLES, 0, vertexcount);
  }
};
```

● メインプログラム (main.cpp) の変更点

SolidShape クラスを定義しているヘッダファイル SolidShape.h を main.cpp の冒頭で #include します。

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

六面体の頂点属性のデータ solidCubeVertex を次のように修正します。各面の最初の頂点のデータと 3 番目の頂点のデータを、ともに最後の頂点のデータの直前に挿入します。インデックスを使ったときは、各面の 2 点を二つの三角形で共有していました。GL_TRIANGLES ではインデックスを使わないと頂点を共有できないので、その分頂点のデータが増えてしまいます。また、

インデックスのデータ solidCubeIndex も気色悪いのでこれに合わせて修正していますが、これは使わないので放置していても構いません。

```
// 面ごとに色を変えた六面体の頂点属性
constexpr Object::Vertex solidCubeVertex[] =
{
  // 左
  { -1.0f, -1.0f, -1.0f,  0.1f,  0.8f,  0.1f },
  { -1.0f, -1.0f,  1.0f,  0.1f,  0.8f,  0.1f },
  { -1.0f,  1.0f,  1.0f,  0.1f,  0.8f,  0.1f },
  { -1.0f, -1.0f, -1.0f,  0.1f,  0.8f,  0.1f },
  { -1.0f,  1.0f,  1.0f,  0.1f,  0.8f,  0.1f },
  { -1.0f,  1.0f, -1.0f,  0.1f,  0.8f,  0.1f },

  // 裏
  {  1.0f, -1.0f, -1.0f,  0.8f,  0.1f,  0.8f },
  { -1.0f, -1.0f, -1.0f,  0.8f,  0.1f,  0.8f },
  { -1.0f,  1.0f, -1.0f,  0.8f,  0.1f,  0.8f },
  {  1.0f, -1.0f, -1.0f,  0.8f,  0.1f,  0.8f },
  { -1.0f,  1.0f, -1.0f,  0.8f,  0.1f,  0.8f },
  {  1.0f,  1.0f, -1.0f,  0.8f,  0.1f,  0.8f },

  // 下
  { -1.0f, -1.0f, -1.0f,  0.1f,  0.8f,  0.8f },
  {  1.0f, -1.0f, -1.0f,  0.1f,  0.8f,  0.8f },
  {  1.0f, -1.0f,  1.0f,  0.1f,  0.8f,  0.8f },
  { -1.0f, -1.0f, -1.0f,  0.1f,  0.8f,  0.8f },
  {  1.0f, -1.0f,  1.0f,  0.1f,  0.8f,  0.8f },
  { -1.0f, -1.0f,  1.0f,  0.1f,  0.8f,  0.8f },

  // 右
  {  1.0f, -1.0f,  1.0f,  0.1f,  0.1f,  0.8f },
  {  1.0f, -1.0f, -1.0f,  0.1f,  0.1f,  0.8f },
  {  1.0f,  1.0f, -1.0f,  0.1f,  0.1f,  0.8f },
  {  1.0f, -1.0f,  1.0f,  0.1f,  0.1f,  0.8f },
  {  1.0f,  1.0f, -1.0f,  0.1f,  0.1f,  0.8f },
  {  1.0f,  1.0f,  1.0f,  0.1f,  0.1f,  0.8f },

  // 上
  { -1.0f,  1.0f, -1.0f,  0.8f,  0.1f,  0.1f },
  { -1.0f,  1.0f,  1.0f,  0.8f,  0.1f,  0.1f },
  {  1.0f,  1.0f,  1.0f,  0.8f,  0.1f,  0.1f },
  { -1.0f,  1.0f, -1.0f,  0.8f,  0.1f,  0.1f },
  {  1.0f,  1.0f,  1.0f,  0.8f,  0.1f,  0.1f },
  {  1.0f,  1.0f, -1.0f,  0.8f,  0.1f,  0.1f },

  // 前
  { -1.0f, -1.0f,  1.0f,  0.8f,  0.8f,  0.1f },
  {  1.0f, -1.0f,  1.0f,  0.8f,  0.8f,  0.1f },
  {  1.0f,  1.0f,  1.0f,  0.8f,  0.8f,  0.1f },
  { -1.0f, -1.0f,  1.0f,  0.8f,  0.8f,  0.1f },
  {  1.0f,  1.0f,  1.0f,  0.8f,  0.8f,  0.1f },
  { -1.0f,  1.0f,  1.0f,  0.8f,  0.8f,  0.1f }
};

// 六面体の面を塗りつぶす三角形の頂点のインデックス
```

```
constexpr GLuint solidCubeIndex[] =
{
   0,  1,  2,  3,  4,  5, // 左
   6,  7,  8,  9, 10, 11, // 裏
  12, 13, 14, 15, 16, 17, // 下
  18, 19, 20, 21, 22, 23, // 右
  24, 25, 26, 27, 28, 29, // 上
  30, 31, 32, 33, 34, 35  // 前
};
```

solidCubeVertex の要素数 (頂点数) が 36 個に増えたので、main() 関数で SolidShapeIndex クラスのインスタンスを生成しているところの頂点の数も修正します。

```
int main()
{
  《省略》

  // 図形データを作成する
  std::unique_ptr<const Shape> shape(new SolidShapeIndex(3, 36, solidCubeVertex,
    36, solidCubeIndex));

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

■ サンプルプログラム step23

実は、上記のプログラムでは solidCubeIndex のインデックスの順序が solidCubeVertex の頂点の順序と一致しているので、SolidShapeIndex クラスの代わりに SolidShape クラスを用いても、問題なく描画されます。しかし、この後の都合上、このままにしておきます。

8.4 隠面消去

立体図形を表示では、他の面に隠された面を描かないようにしないと、例えば遠くにあるものが近くのものよりも手前に描かれてしまうということが起こります。他の面に隠された面を描かない (除去する・消去する) 処理のことを、**隠面消去処理**といいます。

8.4.1 図形の回転

隠面消去処理の説明の前に、この立方体が回転するアニメーションを表示するよう、プログラムを変更します。

● メインプログラム (main.cpp) の変更点

main() 関数のループの直前で glfwSetTime() 関数を使ってタイマーを 0 にセットし、そこか

らの経過時間を glfwGetTime() で取得して、それをそのまま回転角に使います。この回転角から y 軸中心に回転する変換行列を求め、それをモデル変換行列に乗じます。glfwGetTime() が返す経過時間の単位は秒なので、立方体は 2π ≒ 6.3 秒かけて一周します。

```
int main()
{
  《省略》

  // 図形データを作成する
  std::unique_ptr<const Shape> shape(new SolidShapeIndex(3, 8, cubeVertex,
    36, solidCubeIndex));

  // タイマーを 0 にセット
  glfwSetTime(0.0);

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

    // モデル変換行列を求める
    const GLfloat *const location(window.getLocation());
    const Matrix r(Matrix::rotate(static_cast<GLfloat>(glfwGetTime()),
      0.0f, 1.0f, 0.0f));
    const Matrix model(Matrix::translate(location[0], location[1], 0.0f) * r);

    《省略》
  }
}
```

void glfwSetTime(double time)

GLFW のタイマーに時間をセットします。

time

セットする時間、単位は秒。

double glfwGetTime()

GLFW のタイマーの現在時刻を戻り値として返します。単位は廟です。

● Window クラス (Window.h) の変更点

プログラムを立方体が回転するアニメーションにしたので、何もしなくても立方体が回転するように消費電力をケチる処理 (P. 111) を無効にします。Window.h で定義している Window クラスの bool 演算子においてイベントの取得方法をイベントの発生を待つ glfwWaitEvents() と待たない glfwPollEvents() を切り替えていたのを、glfwPollEvents() だけに戻します。

```
  // 描画ループの継続判定
  explicit operator bool()
  {
    // イベントを取り出す
```

```
    glfwPollEvents();

    《省略》
}
```

■ サンプルプログラム step24

8.4.2 背面カリング

このプログラムを実行すると、最初は図 123 (a) のように正しい六面体のように描かれます。しかし、この六面体を回転させると、途中で正しい六面体として表示されなくなります。

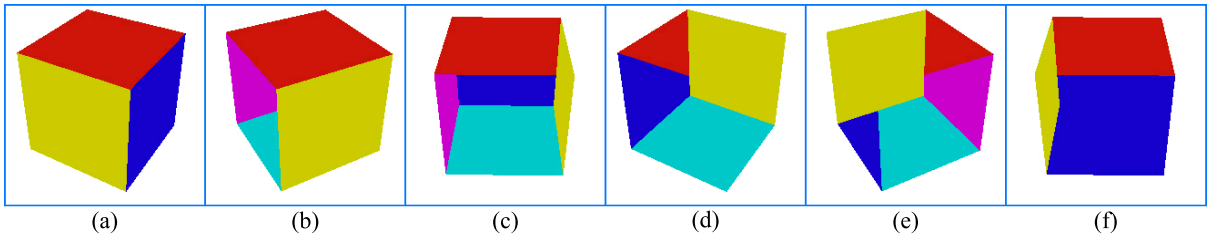


図 123 六面体を一回転させた時の表示

そこで、六面体が図 123 (d) の向きにあるとき、六面体の各面をデータの順で描くと、図 124 のようになります。このように多面体の各面をデータの順に重ね書きしていくだけでは、面の見え隠れが正しく表現できません。

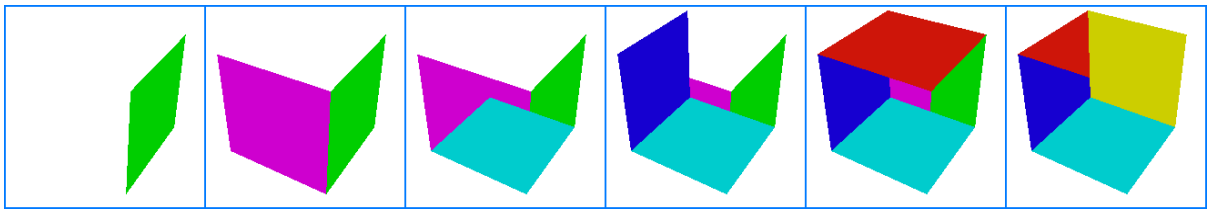


図 124 図 123 (d) を一面ずつ描画

そこで、六面体の各面に表裏を設定します。多面体のデータの面の表裏は、その面を構成する頂点の順番で表すことができます。多面体の表を向いている面を見たとき、その面の頂点が左回りに並んでいれば、その面は視点に対して表を向けているとします (図 125)。

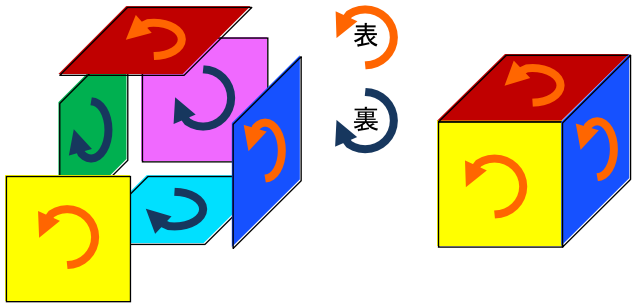


図 125 面に表裏を設定する

このとき、視点から裏側が見えている面を描かないようにすれば、立方体のような凸多面体の場合は、このような問題を避けることができます (図 127)。この処理を、**背面カリング** (Backface Culling) といいます。

三角形の場合、この判定は次のようにして行うことができます。左手系の座標系において、三角形の頂点のスクリーン上の位置ベクトルを $\mathbf{P}_0$、$\mathbf{P}_1$、$\mathbf{P}_2$ とし、$\mathbf{P}_0$ から $\mathbf{P}_1$ に向かうベクトルを $\mathbf{V}_1$、$\mathbf{P}_0$ から $\mathbf{P}_2$ に向かうベクトルを $\mathbf{V}_2$ とするとき、$\mathbf{V}_1$、$\mathbf{V}_2$ の外積 $\mathbf{V}_1 \times \mathbf{V}_2$ の Z 成分が正なら $\mathbf{P}_0$、$\mathbf{P}_1$、$\mathbf{P}_2$ は左回り (表面)、負なら右回り (裏面) になっています。

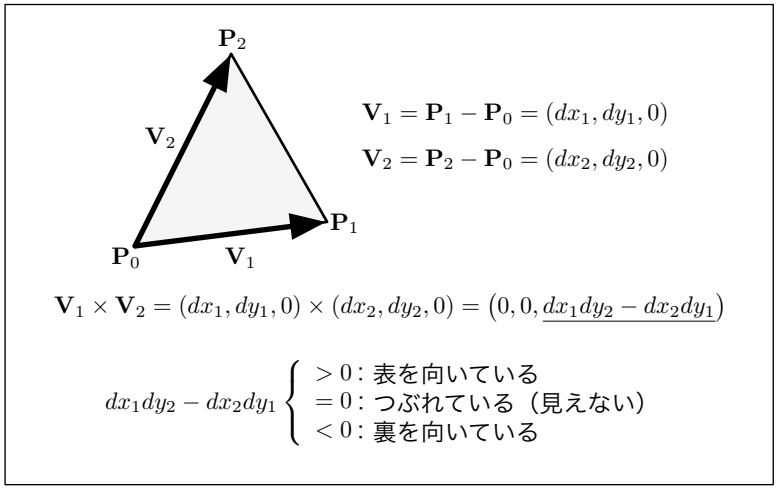


図 126 三角形の表裏判定

● **メインプログラム** (main.cpp) **の変更点**

OpenGL にはこの判定機能が組み込まれており、以下の設定により有効にすることができます。

```
int main()
{
  《省略》

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

  // 背面カリングを有効にする
  glFrontFace(GL_CCW);
  glCullFace(GL_BACK);
  glEnable(GL_CULL_FACE);

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

  《省略》

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

```
    }
}
```

■ サンプルプログラム step25

void glFrontFace(GLenum mode)

視点に対して表側を向いている面 (表面) の条件を引数 mode に指定します。

mode

GL_CCW なら見て頂点が左回り (反時計回り、Counterclockwise) の面を表とし、GL_CW なら頂点が右回り (時計回り、Clockwise) の面を面とする。デフォルトは GL_CCW。

void glCullFace(GLenum mode)

表面を削除するか、裏面を削除するかを引数 mode に指定します。

mode

表面を削除するなら GL_FRONT、裏面を削除するなら GL_BACK。デフォルトは GL_BACK。

これらの設定の後、glEnable(GL_CULL_FACE) を実行すれば、背面カリングが有効になります。背面カリングを無効にするには glDisable(GL_CULL_FACE) を実行します。

補足：背面カリングのデフォルト値

デフォルトでは背面カリングは無効になっています。また、デフォルトでは glFrontFace() が GL_CCW、glCullFace() が GL_BACK になっているので、多面体の外から面を見たときに頂点が左回りになっていれば、glEnable(GL_CULL_FACE) だけで背面カリングが有効になります。

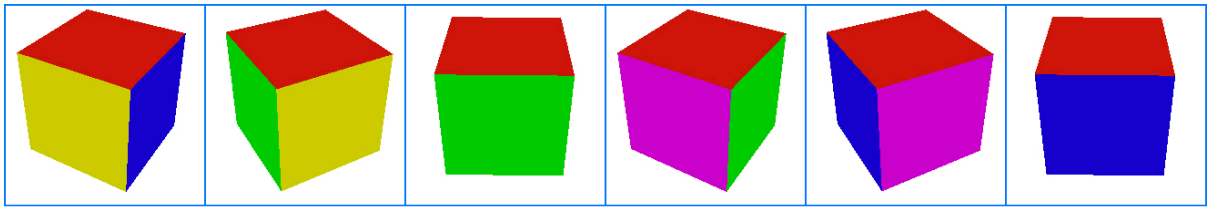


図 127 背面カリングを行なった場合

8.4.3 デプスバッファ法

背面カリングは凸な多面体に対しては有効ですが、多面体に凹部があったり、凸な多面体でも複数あったりする場合 (図 128) には、期待した表示が得られないことがあります。

図 128 背面カリングがうまくいかない例

● メインプログラム (main.cpp) の変更点

描く図形を二つにします。二つ目の図形は一つ目の図形に対して $(0, 0, 3)$ の位置に配置します。この座標変換を一つ目の図形のモデルビュー変換の前に適用すれば、二つ目の図形が一つ目の図形と一緒に動くようになります。

```
int main()
{
  《省略》

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

    // uniform 変数に値を設定する
    glUniformMatrix4fv(projectionLoc, 1, GL_FALSE, projection.data());
    glUniformMatrix4fv(modelviewLoc, 1, GL_FALSE, modelview.data());

    // 図形を描画する
    shape->draw();

    // 二つ目のモデルビュー変換行列を求める
    const Matrix modelview1(modelview * Matrix::translate(0.0f, 0.0f, 3.0f));

    // uniform 変数に値を設定する
    glUniformMatrix4fv(modelviewLoc, 1, GL_FALSE, modelview1.data());

    // 二つ目の図形を描画する
    shape->draw();

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

■ サンプルプログラム step26

● 実行結果

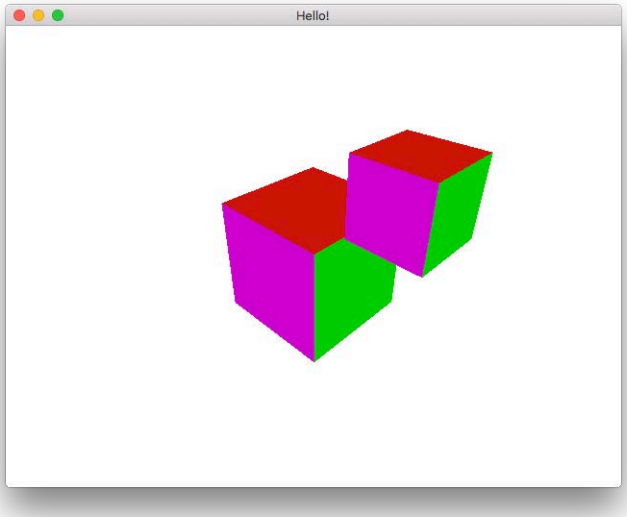


図 129 複数の図形の前後関係が正しく表現されない場合

このプログラムを実行すると、回転の途中で二つの図形の前後関係がおかしくなるところがあります (図 129)。これが常に正しく表示されるようにするには、他の面に隠されて見えない面の領域を表示しないようにする必要があります。このような隠面消去処理にはさまざまな手法がありますが、本書が対象としているパソコンのグラフィックスハードウェアでは、**デプスバッファ法**あるいは **Z バッファ法**と呼ばれる手法が採用されています。

デプスバッファ法は物体表面の**ポリゴン** (多角形) をラスタ化 (4.1 節) するときに、画素単位に奥行きを比較して、最も手前にあるポリゴンの色を画素の色とする手法です。この処理では、画面上の表示領域と同じ画素数の配列を別に用意します。この配列にはラスタ化された多角形の各画素における**奥行き** (デプス、深度、Z 値) を格納するので、これを**深度バッファ**、**デプスバッファ**、あるいは **Z バッファ**と呼びます。

● デプスバッファ法の手順

まず、デプスバッファのすべての要素に、その画素が取りうる深度の最大値を格納しておきます。図 130 では、最大の深度を「9」にしています。

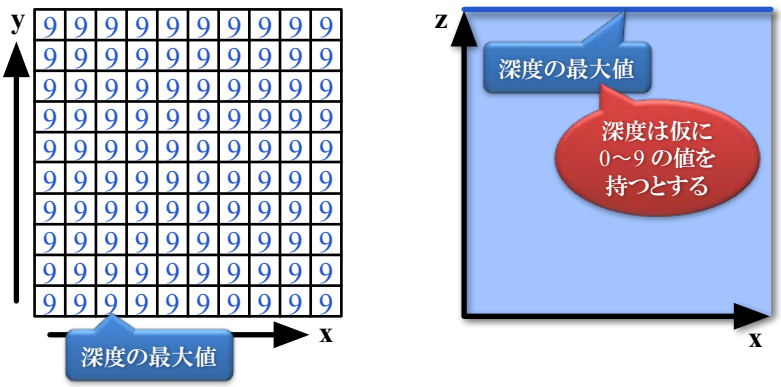


図 130 デプスバッファの初期化

ここに 1 枚目のポリゴンを描画します。このポリゴンの深度は「3」であるとします。デプスバッファの内容はそれより大きな「9」なので、このポリゴンの画素は全て表示されます。また、ポリゴンを表示した画素の位置のデプスバッファの内容は、このポリゴンの深度の「3」に更新されます (図 131)。

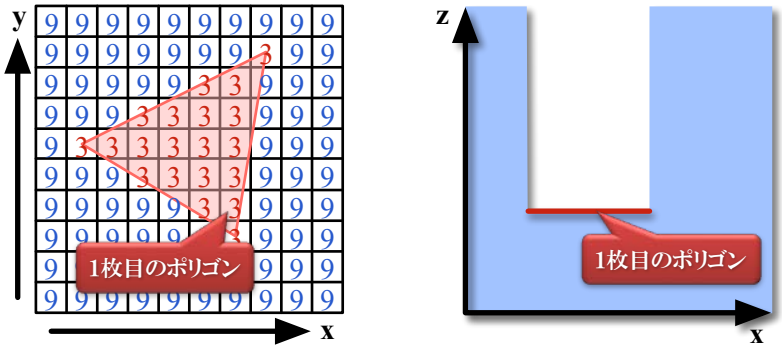


図 131 1 枚目のポリゴンの描画

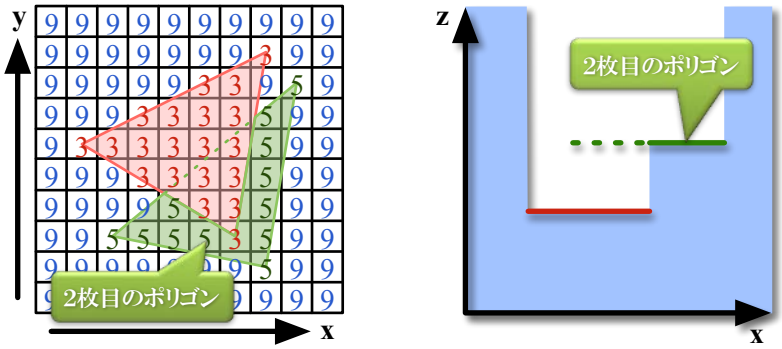


図 132 2 枚目のポリゴンの描画

次に、2 枚目のポリゴンを描画します。このポリゴンの深度は「5」であるとします。この場合、デプスバッファの深度がそれより大きい「9」の画素ではこのポリゴンが表示され、デプスバッファの内容が「5」に更新されます。一方、デプスバッファの内容がそれより小さな「3」の画素で

はこのポリゴンは表示されず、深度も「3」のまま変更されません (図 132)。これにより画面には視点に最も近いポリゴンの画素が表示され、その奥にあるポリゴンの画素が隠されます。

● メインプログラム (main.cpp) の変更点

OpenGL にはこの機能が組み込まれており、以下の設定により有効にすることができます。デプスバッファの初期化するには、glClear() の引数に GL_DEPTH_BUFFER_BIT を追加します。

```
int main()
{
  《省略》

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

  // 背面カリングを有効にする
  glFrontFace(GL_CCW);
  glCullFace(GL_BACK);
  glEnable(GL_CULL_FACE);

  // デプスバッファを有効にする
  glClearDepth(1.0);
  glDepthFunc(GL_LESS);
  glEnable(GL_DEPTH_TEST);

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

  《省略》

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

    《省略》
  }
}
```

■ サンプルプログラム step27

void glClearDepth(GLdouble depth), void glClearDepthf(GLfloat depth)

glClear(GL_DEPTH_BUFFER_BIT) でデプスバッファに設定する深度を引数 depth に指定します。glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT) のようにすれば、カラーバッファとデプスバッファを同時に塗りつぶすことができます。

depth

デプスバッファに設定する値。0〜1 の値を設定でき、初期値は 1。

void glDepthFunc(GLenum func)

奥行きの比較関数を引数 func に指定します。

func

GL_NEVER (ポリゴンを表示しない), GL_LESS (ポリゴンの深度がデプスバッファより小さければ表示する), GL_EQUAL (ポリゴンの深度がデプスバッファと等しければ表示する), GL_LEQUAL (ポリゴンの深度がデプスバッファより小さいか等しければ表示する), GL_GREATER (ポリゴンの深度がデプスバッファより大きければ表示する), GL_NOTEQUAL (ポリゴンの深度がデプスバッファと等しくなければ表示する), GL_GEQUAL (ポリゴンの深度がデプスバッファより大きいか等しければ表示する), GL_ALWAYS (ポリゴンを常に表示する) が指定できる。デフォルトは GL_LESS。

これらの設定の後 glEnable(GL_DEPTH_TEST) を実行すれば、デプスバッファ法による隠面消去処理が有効になります。隠面消去処理を無効にするには glDisable(GL_DEPTH_TEST) を実行します。デフォルトでは隠面消去処理は無効になっています。また、デフォルトでは glClearDepth() が 1、glDepthFunc() が GL_ LESS になっているので、glEnable(GL_DEPTH_TEST) だけで隠面消去処理が有効になります。

● **実行結果**

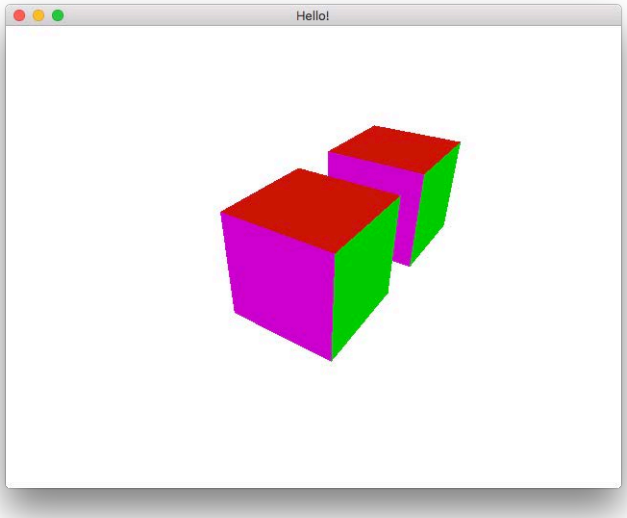


図 133 隠面消去処理を有効にした場合

第9章 陰影付け

9.1 二色性反射モデル

陰影付け (シェーディング、shading) は、物体の表面に光を当てた時の、反射光の強さを計算する処理のことをいいます (図 134)。このとき光源の位置、強度 (明るさ / 色) などの特性、物体表面の光の反射特性 (色・つや)、および視点の位置と、視点に届く反射光の強度との関係を表したものを**陰影付けモデル (シェーディングモデル**、shading model) といいます。

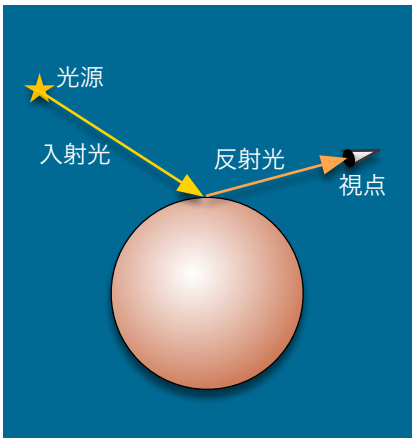


図 134 入射光と反射光

光源から発射された光線 (入射光) は物体表面で正反射光と屈折光に分かれます (図 135)。

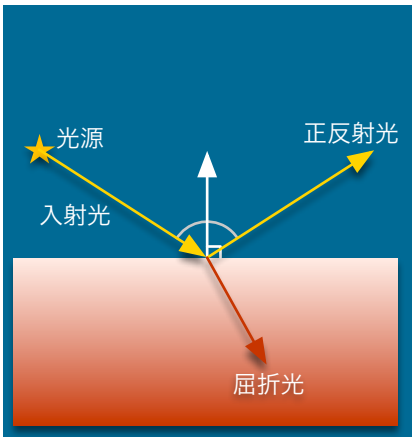


図 135 正反射光と屈折光

このうち屈折光は物体内部に侵入した後、再び外部に放射されます。この光はあらゆる方向に拡散して放射されるので、**拡散反射光** (diffuse reflection) と呼ばれます。一方、正反射光は物体表面で入射光の正反射方向を軸とする方向に拡散して放射され、その一部が視点に届きます。この光は物体表面で鏡のように反射するので、**鏡面反射光** (specular reflection) と呼ばれます。

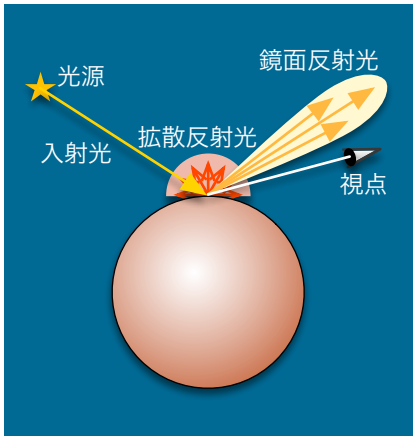


図 136 二色性反射モデル

視点には、この拡散反射光と鏡面反射光を足したものが到達します。このように、反射光をこれらの二つの成分に分けてとらえたモデルを**二色性反射モデル**といいます。陰影付けでは、これに間接光など光源からの直接光以外の成分である**環境光** (ambient light) の反射光を加えます。

9.1.1 拡散反射光

物体が透明なら、屈折光はその中を直進します。しかし物体が不透明な非金属なら、屈折光はその中で散乱と吸収を繰り返し、入射点から再び物体の外部に放射されると考えます (図 137)。このとき放射される光は物体内での散乱によって指向性を失い、物体の外部の全ての方向に均等に拡散すると考えます。これを拡散反射光といいます。また、拡散反射光の色成分は入射光の色成分のうち物体内で吸収されずに残ったものになるため、物体の色が付いています。

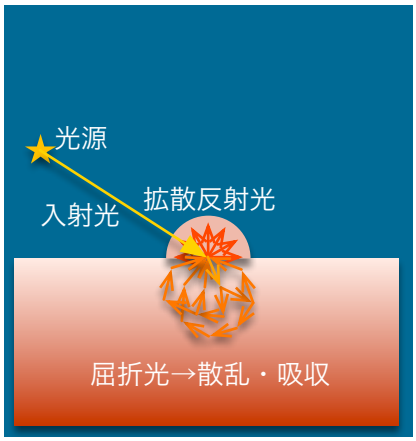


図 137 拡散反射光

拡散反射光の強度は入射光の強度に比例します。さらに入射光の強度は入射光の密度に比例します。入射点の近傍の微小な領域では、入射光は平行光線とみなすことができるので、その幅を d、入射角を θ_i とすれば、入射光の照射範囲は $d / \cos\theta_i$ となります。したがって入射光の密度は入射角 θ_i の余弦 $\cos\theta_i$ に比例します。これを **Lambert の余弦法則**といいます。物体表面の法線ベクトル $\mathbf{N}$ と光源方向のベクトル $\mathbf{L}$ がともに単位ベクトルなら、$\cos\theta_i = \mathbf{N} \cdot \mathbf{L}$ です。

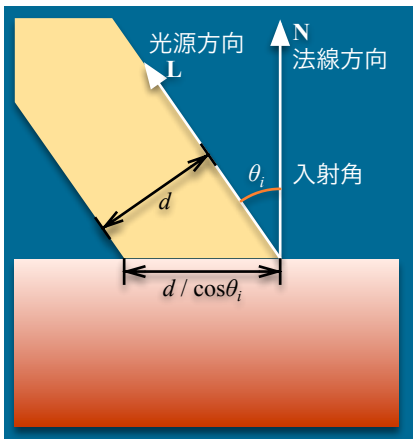


図 138 入射光の密度

いま、L_{diff} を光源強度の拡散反射光成分 (現実の光源では拡散反射光成分を分離することはできませんが、実装上このような取り扱いをします)、K_{diff} を物体表面の材質の拡散反射係数とするとき、この物体表面における拡散反射光強度 I_{diff} は次式で求められます。L_{diff}、K_{diff}、および I_{diff} は光の三原色 R、G、B の三つの要素を持っており、$\otimes$ はその要素ごとの積を表す演算子です。

$$I_{diff} = \cos\theta_i K_{diff} \otimes L_{diff} = (\mathbf{N} \cdot \mathbf{L})\, K_{diff} \otimes L_{diff} \tag{75}$$

ここで $\mathbf{N}$ と $\mathbf{L}$ のなす角 θ_i が $\pi/2$ 以上、すなわち $\mathbf{N} \cdot \mathbf{L}$ が負のときは、光源が物体表面の裏側にあるので、拡散反射光強度は 0 にします。$\max(x, y)$ は x と y の大きい方を返す関数です。このモデルによる陰影付けの例を、図 139 に示します。

$$I_{diff} = \max(\mathbf{N} \cdot \mathbf{L}, 0)\, K_{diff} \otimes L_{diff} \tag{76}$$

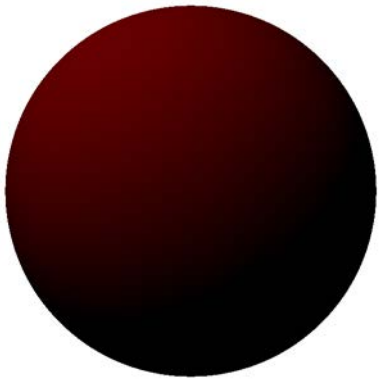


図 139 拡散反射光による陰影の例

● メインプログラム (main.cpp) の変更点

これまでは描画する図形 (六面体) の頂点属性 solidCubeVertex の個々の頂点に頂点色を指定していました。これをその頂点における法線ベクトルに書き換えます。

```
// 面ごとに法線を変えた六面体の頂点属性
constexpr Object::Vertex solidCubeVertex[] =
{
  // 左
  { -1.0f, -1.0f, -1.0f, -1.0f,  0.0f,  0.0f },
  { -1.0f, -1.0f,  1.0f, -1.0f,  0.0f,  0.0f },
  { -1.0f,  1.0f,  1.0f, -1.0f,  0.0f,  0.0f },
  { -1.0f, -1.0f, -1.0f, -1.0f,  0.0f,  0.0f },
  { -1.0f,  1.0f,  1.0f, -1.0f,  0.0f,  0.0f },
  { -1.0f,  1.0f, -1.0f, -1.0f,  0.0f,  0.0f },

  // 裏
  {  1.0f, -1.0f, -1.0f,  0.0f,  0.0f, -1.0f },
  { -1.0f, -1.0f, -1.0f,  0.0f,  0.0f, -1.0f },
  { -1.0f,  1.0f, -1.0f,  0.0f,  0.0f, -1.0f },
  {  1.0f, -1.0f, -1.0f,  0.0f,  0.0f, -1.0f },
  { -1.0f,  1.0f, -1.0f,  0.0f,  0.0f, -1.0f },
  {  1.0f,  1.0f, -1.0f,  0.0f,  0.0f, -1.0f },

  // 下
  { -1.0f, -1.0f, -1.0f,  0.0f, -1.0f,  0.0f },
  {  1.0f, -1.0f, -1.0f,  0.0f, -1.0f,  0.0f },
  {  1.0f, -1.0f,  1.0f,  0.0f, -1.0f,  0.0f },
  { -1.0f, -1.0f, -1.0f,  0.0f, -1.0f,  0.0f },
  {  1.0f, -1.0f,  1.0f,  0.0f, -1.0f,  0.0f },
  { -1.0f, -1.0f,  1.0f,  0.0f, -1.0f,  0.0f },

  // 右
  {  1.0f, -1.0f,  1.0f,  1.0f,  0.0f,  0.0f },
  {  1.0f, -1.0f, -1.0f,  1.0f,  0.0f,  0.0f },
  {  1.0f,  1.0f, -1.0f,  1.0f,  0.0f,  0.0f },
  {  1.0f, -1.0f,  1.0f,  1.0f,  0.0f,  0.0f },
  {  1.0f,  1.0f, -1.0f,  1.0f,  0.0f,  0.0f },
  {  1.0f,  1.0f,  1.0f,  1.0f,  0.0f,  0.0f },

  // 上
  { -1.0f,  1.0f, -1.0f,  0.0f,  1.0f,  0.0f },
  { -1.0f,  1.0f,  1.0f,  0.0f,  1.0f,  0.0f },
  {  1.0f,  1.0f,  1.0f,  0.0f,  1.0f,  0.0f },
  { -1.0f,  1.0f, -1.0f,  0.0f,  1.0f,  0.0f },
  {  1.0f,  1.0f,  1.0f,  0.0f,  1.0f,  0.0f },
  {  1.0f,  1.0f, -1.0f,  0.0f,  1.0f,  0.0f },

  // 前
  { -1.0f, -1.0f,  1.0f,  0.0f,  0.0f,  1.0f },
  {  1.0f, -1.0f,  1.0f,  0.0f,  0.0f,  1.0f },
  {  1.0f,  1.0f,  1.0f,  0.0f,  0.0f,  1.0f },
  { -1.0f, -1.0f,  1.0f,  0.0f,  0.0f,  1.0f },
  {  1.0f,  1.0f,  1.0f,  0.0f,  0.0f,  1.0f },
  { -1.0f,  1.0f,  1.0f,  0.0f,  0.0f,  1.0f }
};
```

頂点属性の頂点色を法線ベクトルに置き換えたので、関数 createProgram() で場所を取り出している頂点色の uniform 変数の変数名 color を normal に変更します。

```
// 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());

   《省略》

  // プログラムオブジェクトをリンクする
  glBindAttribLocation(program, 0, "position");
  glBindAttribLocation(program, 1, "normal");
  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;
}
```

● Object クラス (Object.h) の変更点

Object クラス (Object.h) でも main.cpp の変更に合わせて、頂点属性の構造体 Vertex のメンバ変数 color を normal に変更します。

```
// 図形データ
class Object
{
   《省略》

public:

  // 頂点属性
  struct Vertex
  {
    // 位置
    GLfloat position[3];

    // 法線
    GLfloat normal[3];
  };

  // 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
  //   indexcount: 頂点のインデックスの要素数
```

```
  //   index: 頂点のインデックスを格納した配列
  Object(GLint size, GLsizei vertexcount, const Vertex *vertex,
    GLsizei indexcount, const GLuint *index)
  {
     《省略》

    // 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, sizeof (Vertex),
      static_cast<Vertex *>(0)->position);
    glEnableVertexAttribArray(0);
    glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, sizeof (Vertex),
      static_cast<Vertex *>(0)->normal);
    glEnableVertexAttribArray(1);
```

● バーテックスシェーダ (point.vert) の変更点

バーテックスシェーダの in 変数 color を normal に変更します。これは法線ベクトルなので、データ型は vec3 にします。また out 変数 vertex_color を Idiff に変更します。このデータ型も vec3 にします。このほか、光源の位置の同次座標 Lpos、光源強度の拡散反射光成分 Ldiff および拡散反射係数 Kdiff をそれぞれ定数で与えます。vec3(1.0) は vec3(1.0, 1.0, 1.0) と同じです。

ローカル座標系における頂点の位置 position にモデルビュー変換行列 modelview を乗じて、視点座標系の頂点位置 P を求めておきます。このプログラムでは頂点の座標値の第 4 要素 (w 要素) を指定していませんが (glVertexAttribPointer() の第 2 引数 size には 3 を指定している)、バーテックスシェーダの in 変数の第 4 要素には 1 があらかじめ設定されています。

この頂点における光源方向のベクトル L は、光源の位置 Lpos から頂点の位置 P を引いて求めます。これらはともに同次座標なので、7.2.2 節の式 (11) のとおり通分してから減算し、その結果の xyz 成分を正規化します。これと頂点の法線ベクトル normal から、式 (76) により拡散反射光強度 Idiff を求めます。dot() はベクトルの内積を求める GLSL の組込関数です。Idiff は out 変数なので、画素の位置で補間されてフラグメントシェーダに渡されます。

```
#version 150 core
uniform mat4 modelview;
uniform mat4 projection;
const vec4 Lpos = vec4(0.0, 0.0, 5.0, 1.0);
const vec3 Ldiff = vec3(1.0);
const vec3 Kdiff = vec3(0.6, 0.6, 0.2);
in vec4 position;
in vec3 normal;
out vec3 Idiff;
void main()
{
  vec4 P = modelview * position;
  vec3 L = normalize((Lpos * P.w - P * Lpos.w).xyz);
  Idiff = max(dot(normal, L), 0.0) * Kdiff * Ldiff;
  gl_Position = projection * P;
}
```

float dot(genType x, genType y)

ベクトル x とベクトル y の内積を戻り値として返す、GLSL の組み込み関数です。genType は引数のデータ型が任意であることを示します。

x, y

内積を求める二つのベクトル。二つのベクトルのデータ型は一致している必要がある。

genType normalize(genType v)

ベクトル v を正規化したベクトルを戻り値として返す、GLSL の組み込み関数です。

v

正規化するベクトル。戻り値のデータ型は引数のデータ型と一致する。

genType max(genType x, genType y) または genType max(genType x, float y)

ベクトル x と ベクトル (もしくはスカラー) y を要素ごとに比較し、大きい方の要素からなるベクトルを戻り値として返す、GLSL の組み込み関数です。

x, y

比較するベクトル。y が float 型の時は、この y と x の個々の要素を比較する。

これらの関数は **GLSL の組込関数**です。C や C++ 言語の関数ではありません。

● **フラグメントシェーダ (point.frag) の変更点**

バーテックスシェーダからは out 変数 Idiff に拡散反射光強度を求めていましたから、フラグメントシェーダの in 変数 vertex_color は Idiff に変更します。このデータ型はバーテックスシェーダの out 変数の Idiff と同じ vec3 です。これを vec4 に型変換し、アルファ値の第 4 要素に 1 (不透明) を設定して、フラグメントの色の out 変数 fragment に代入します。

```
#version 150 core
in vec3 Idiff;
out vec4 fragment;
void main()
{
  fragment = vec4(Idiff, 1.0);
}
```

■ **サンプルプログラム step28**

● 実行結果

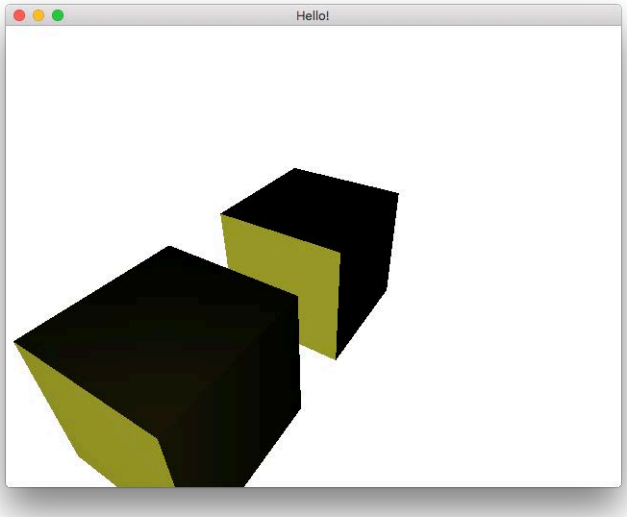


図 140 拡散反射光による陰影付けを行なった場合

9.1.2 法線ベクトルの回転

図 140 は物体が回転しても、陰影はあまり変化しません。これは物体が回転して各面の向きが変化しているのに、その法線ベクトルが変化していないためです。陰影付けを正しく行うためには、物体の回転に合わせて、法線ベクトルも回転する必要があります。

面の向きと法線ベクトルを一致させるには、法線ベクトルを回転後の頂点の位置から求めるか、あるいは法線ベクトルにも物体の回転の変換を適用する必要があります。物体を回転するたびに法線ベクトルを計算するのは少し計算コストが高いので、ここでは後者の法線ベクトルに対する回転の変換について考えます。

いま、**M** をモデルビュー変換行列 (P. 118) とします。法線ベクトル (x, y, z) を同次座標で表した $(x\ y\ z\ 0)^T$ にこれを乗じれば、頂点の座標と同じ変換を法線ベクトルにも適用できます。

$$\begin{pmatrix} x' \\ y' \\ z' \\ w' \end{pmatrix} = \mathbf{M} \begin{pmatrix} x \\ y \\ z \\ 0 \end{pmatrix} = \begin{pmatrix} a_{xx} & a_{yx} & a_{zx} & b_x \\ a_{xy} & a_{yy} & a_{zy} & b_y \\ a_{xz} & a_{yz} & a_{zz} & b_z \\ c_x & c_y & c_z & d \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 0 \end{pmatrix} \tag{77}$$

これより変換後の法線ベクトル (x', y', z') は、次式で求めることができます。

$$\begin{aligned} x' &= a_{xx}x + a_{yx}y + a_{zx}z \\ y' &= a_{xy}x + a_{yy}y + a_{zy}z \\ z' &= a_{xz}x + a_{yz}y + a_{zz}z \end{aligned} \tag{78}$$

これはベクトル (x, y, z) に **M** の第 4 行と第 4 列を取り除いた小行列 $\mathbf{M}_4$ を乗じており、**M**

に含まれる平行移動の影響は受けません。しかし、$\mathbf{M}_4$ に特定方向への拡大縮小やせん断変形が含まれている場合には、法線ベクトルを正しく変換できません。

例えば、図 141 (a) の図形に対して **M** により x 方向に縮小する変換を行う際に、その小行列 $\mathbf{M}_4$ をこの図形の法線ベクトル **N** にそのまま乗じると、変換後のベクトル $\mathbf{M}_4\mathbf{N}$ は同図 (b) のように図形と直交しなくなります。そのため、同図 (c) のように変換後のベクトル **GN** が変換後の図形と直交する変換 **G** を、$\mathbf{M}_4$ をもとに決める必要があります。

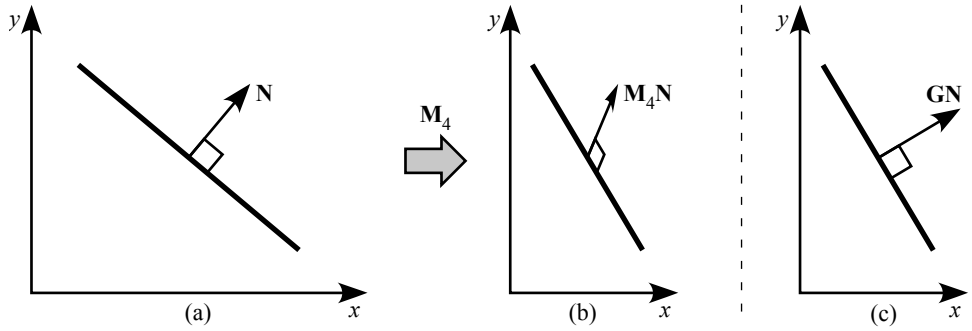


図 141 法線ベクトルの変換

このような変換 **G** は、$\mathbf{M}_4$ の随伴行列 (adjoint matrix) を転置したものになります。

$$\mathbf{G} = (\mathrm{adj}\,(\mathbf{M}_4))^T \tag{79}$$

随伴行列は $\mathbf{M}_4$ の各要素の余因子の行列式を、その要素の転置した位置に置いて求めることができます。

$$\mathbf{M}_4 = \begin{pmatrix} a_{xx} & a_{yx} & a_{zx} \\ a_{xy} & a_{yy} & a_{zy} \\ a_{xz} & a_{yz} & a_{zz} \end{pmatrix}$$

a_{xx}とa_{xx}の余因子
a_{yx}とa_{yx}の余因子

(80)

例えば、a_{xx} の余因子の行列式 d_{xx} と a_{yx} の余因子の行列式 d_{yx} は、次のようになります。

$$d_{xx} = \begin{vmatrix} a_{yy} & a_{zy} \\ a_{yz} & a_{zz} \end{vmatrix},\ d_{yx} = \begin{vmatrix} a_{xy} & a_{zy} \\ a_{xz} & a_{zz} \end{vmatrix} \tag{81}$$

それらの他の要素の余因子についても同様に行列式を求めれば、随伴行列は次式で得られます。

$$\mathrm{adj}\,(\mathbf{M}_4) = \begin{pmatrix} d_{xx} & -d_{xy} & d_{xz} \\ -d_{yx} & d_{yy} & -d_{yz} \\ d_{zx} & -d_{zy} & d_{zz} \end{pmatrix} \tag{82}$$

なお、$\mathbf{M}_4$ の逆行列 $\mathbf{M}_4^{-1}$ は随伴行列をその行列式 $|\mathbf{M}_4|$ で割った $\mathbf{M}_4^{-1} = \mathrm{adj}(\mathbf{M}_4) / |\mathbf{M}_4|$ です。したがって法線ベクトルの変換行列 **G** は、$\mathbf{G} = (\mathbf{M}_4^{-1})^T$ とすることもできます。もし $\mathbf{M}_4$ が回転などの直交行列であれば、随伴行列はその転置行列となるので、$\mathbf{G} = \mathbf{M}_4$ とすることができます。

● Matrix クラス (Matrix.h) の変更点

法線ベクトルの変換行列を求めるメソッド getNormalMatrix() を、Matrix クラス (Matrix.h) に追加します。このメソッドはオブジェクトを変更しませんから、const メソッドにします。

```
// 変換行列
class Matrix
{
  // 変換行列の要素
  GLfloat matrix[16];

public:

  《省略》

  // 変換行列の配列を返す
  const GLfloat *data() const
  {
    return matrix;
  }

  // 法線ベクトルの変換行列を求める
  void getNormalMatrix(GLfloat *m) const
  {
    m[0] = matrix[ 5] * matrix[10] - matrix[ 6] * matrix[ 9];
    m[1] = matrix[ 6] * matrix[ 8] - matrix[ 4] * matrix[10];
    m[2] = matrix[ 4] * matrix[ 9] - matrix[ 5] * matrix[ 8];
    m[3] = matrix[ 9] * matrix[ 2] - matrix[10] * matrix[ 1];
    m[4] = matrix[10] * matrix[ 0] - matrix[ 8] * matrix[ 2];
    m[5] = matrix[ 8] * matrix[ 1] - matrix[ 9] * matrix[ 0];
    m[6] = matrix[ 1] * matrix[ 6] - matrix[ 2] * matrix[ 5];
    m[7] = matrix[ 2] * matrix[ 4] - matrix[ 0] * matrix[ 6];
    m[8] = matrix[ 0] * matrix[ 5] - matrix[ 1] * matrix[ 4];
  }

  《省略》
};
```

● メインプログラム (main.cpp) の変更点

モデルビュー変換行列から法線ベクトルの変換行列を求めて、シェーダプログラムの uniform 変数に設定します。この uniform 変数の変数名は normalMatrix とします。シェーダプログラムから、この変数の場所を normalMatrixLoc という変数に取り出しておきます。

```
int main()
{
  《省略》

  // uniform 変数の場所を取得する
  const GLint modelviewLoc(glGetUniformLocation(program, "modelview"));
  const GLint projectionLoc(glGetUniformLocation(program, "projection"));
  const GLint normalMatrixLoc(glGetUniformLocation(program, "normalMatrix"));
```

モデルビュー変換行列から法線ベクトルの変換行列を求め、unform 変数 normalMatrix に設定します。法線ベクトルの変換行列は 3 行 3 列の 9 要素ですから、求めた変換行列の格納先の配列変数 normalMatrix のサイズをこれに合わせます。また、uniform 変数 normalMatrix は mat3 型ですから、glUniformMatrix3fv() を使って値を設定します。

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

    // 法線ベクトルの変換行列の格納先
    GLfloat normalMatrix[9];

    // モデルビュー変換行列を求める
    const Matrix modelview(view * model);

    // 法線ベクトルの変換行列を求める
    modelview.getNormalMatrix(normalMatrix);

    // uniform 変数に値を設定する
    glUniformMatrix4fv(projectionLoc, 1, GL_FALSE, projection.data());
    glUniformMatrix4fv(modelviewLoc, 1, GL_FALSE, modelview.data());
    glUniformMatrix3fv(normalMatrixLoc, 1, GL_FALSE, normalMatrix);

    // 図形を描画する
    shape->draw();

    // 二つ目のモデルビュー変換行列を求める
    const Matrix modelview1(modelview * Matrix::translate(0.0f, 0.0f, 3.0f));

    // 二つ目の法線ベクトルの変換行列を求める
    modelview1.getNormalMatrix(normalMatrix);

    // uniform 変数に値を設定する
    glUniformMatrix4fv(modelviewLoc, 1, GL_FALSE, modelview1.data());
    glUniformMatrix3fv(normalMatrixLoc, 1, GL_FALSE, normalMatrix);

    // 二つ目の図形を描画する
    shape->draw();

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

● バーテックスシェーダ (point.vert) の変更点

mat3 型の uniform 変数 normalMatrix を追加し、これを in 変数 normal に乗じます。これを GLSL の組み込み関数 normalize() を使って正規化し、vec3 型の変数 N に格納します。拡散反射光強度 Idiff は、この N と光源方向のベクトル L との内積により求めます。

```
#version 150 core
uniform mat4 modelview;
```

```
uniform mat4 projection;
uniform mat3 normalMatrix;
const vec4 Lpos = vec4(0.0, 0.0, 5.0, 1.0);
const vec3 Ldiff = vec3(1.0);
const vec3 Kdiff = vec3(0.6, 0.6, 0.2);
in vec4 position;
in vec3 normal;
out vec4 Idiff;
void main()
{
  vec4 P = modelview * position;
  vec3 N = normalize(normalMatrix * normal);
  vec3 L = normalize((Lpos * P.w - P * Lpos.w).xyz);
  Idiff = max(dot(N, L), 0.0) * Kdiff * Ldiff;
  gl_Position = projection * P;
}
```

■ サンプルプログラム step29

● 実行結果

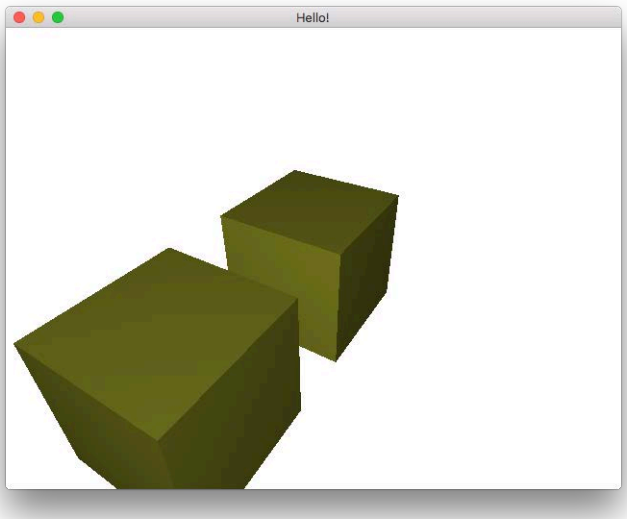


図 142 法線ベクトルを変換した結果

9.1.3 鏡面反射光

鏡面反射光は、入射光のうち物体の内部に進入せず、物体表面で入射光の正反射方向に反射した成分です (図 143)。しかし、ここでは光源を点として扱っているため、この反射光は正反射方向から少しでもずれた方向では観測されないことになります。また、仮に正反射方向から観測したとしても、光源が点であれば大きさを持たないため、やはり見えないことになります。

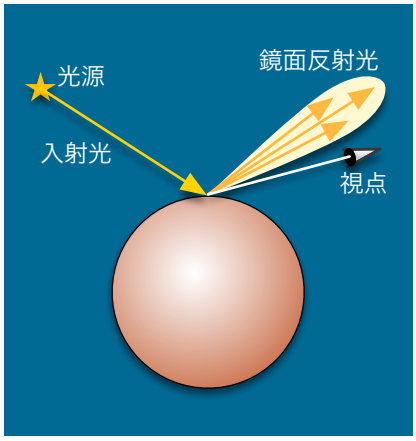


図 143 鏡面反射光

このように光源を点とみなすと矛盾が発生します。この矛盾は光源の大きさや形を考慮すれば避けることができますが、それは計算時間のかかる処理になります。そのため、この光源を点とみなした簡易な陰影付けモデルは、基本的な陰影付け手法として頻繁に利用されています。

とはいえ、反射光が存在するのに見えないというのも困りものです。そこで、正反射光が物体表面の微小な凹凸によって拡散し、視点にはその一部が届くと考えます。この光は物体表面で反射し物体の内部に侵入しないので、物体の色には影響されず、光源の色がそのまま反映されます。

正反射方向の単位ベクトル **R** は、光源方向の単位ベクトルを **L**、物体表面の法線単位ベクトルを **N** とするとき (図 144)、次式で求めることができます。

$$\mathbf{R} = 2(\mathbf{L} \cdot \mathbf{N}) - \mathbf{L} \tag{83}$$

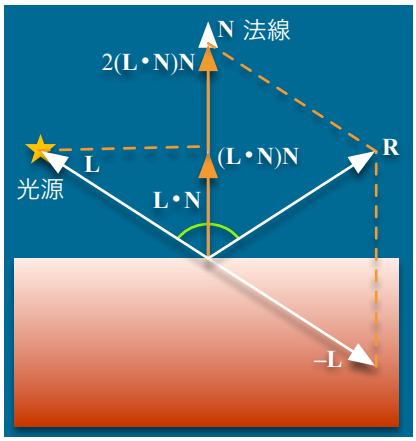


図 144 正反射方向のベクトル R

視点方向の単位ベクトルを **V** とするとき、この物体表面における鏡面反射光強度 I_{spec} は、次式のように **R** と **V** の内積のべき乗を用いて求めます。ここで θ_r は **R** と **V** のなす角 (図 145)、L_{spec} は光源強度の鏡面反射光成分 (これも現実の光源では鏡面反射光成分を分離することはできませんが、実装上このような取り扱いをします)、K_{spec} は物体表面の材質の鏡面反射係数とします。また、べき指数の K_{shi} は**輝き係数** (shininess) と呼ばれます。

$$I_{spec} = \cos^{K_{shi}} \theta_r K_{spec} \otimes L_{spec} = (\mathbf{R} \cdot \mathbf{V})^{K_{shi}} K_{spec} \otimes L_{spec} \tag{84}$$

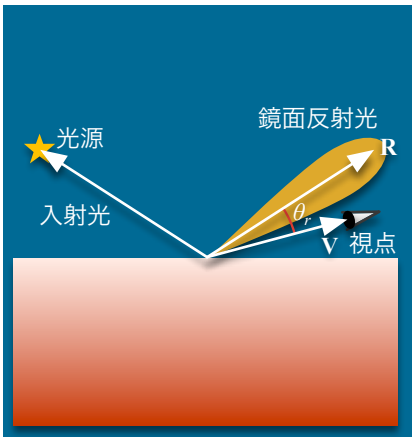


図 145 鏡面反射光強度の算出

これも **R** と **V** のなす角 θ_r が $\pi/2$ 以上、すなわち **R・V** が負のときは光源が物体表面の裏側にあるので、鏡面反射光強度は 0 にします。

$$I_{spec} = \max(\mathbf{R} \cdot \mathbf{V}, 0)^{K_{shi}} K_{spec} \otimes L_{spec} \tag{85}$$

これは **Phong の陰影付けモデル**と呼ばれます。このモデルによる鏡面反射光による陰影は、図 146 のようになります。平面に対する陰影は、入射角にかかわらず同心円状に分布します。

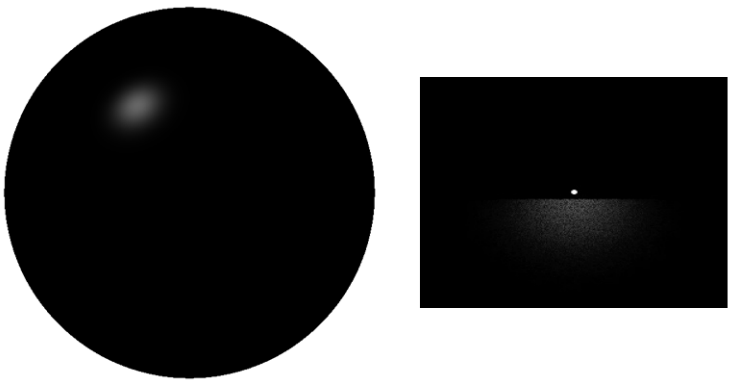


図 146 Phong の陰影付けモデルによる鏡面反射光の陰影の例

輝き係数 K_{shi} は、大きいほど鏡面反射光が鋭くなります。また鏡面反射係数 K_{spec} は、非金属の材質では光源の色をそのまま反映するために、一般的にはグレーに設定します。

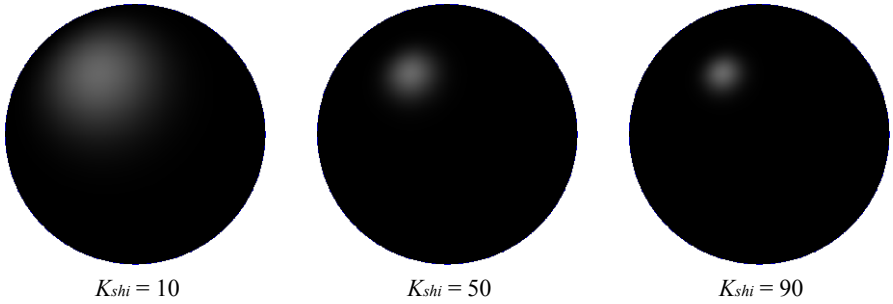


図 147 輝き係数の制御

θ_r を偏角として $\cos^{Kshi}\theta_r$ を極座標のグラフで表すと、図 148 のようになります。

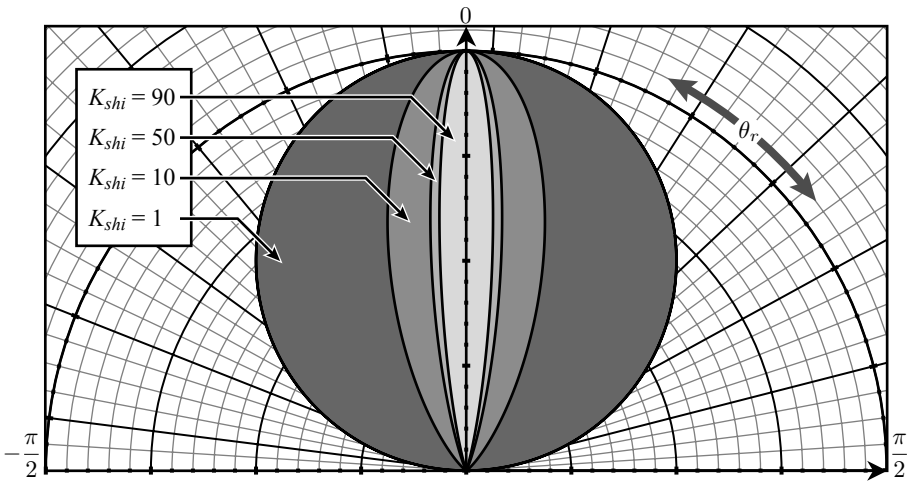


図 148 $\cos^{Kshi}\theta_r$

● **バーテックスシェーダ** (point.vert) **の変更点**

バーテックスシェーダに vec3 型の out 変数 Ispec を追加します。これに鏡面反射光強度を代入して、補間値をフラグメントシェーダに渡します。このほか、光源強度の鏡面反射光成分 Lspec および鏡面反射係数 Kspec、輝き係数 Kshi をそれぞれ定数で与えます。

視点方向のベクトル V は、視点座標系であれば頂点の位置ベクトルの逆ベクトルになりますから、頂点の位置 P の xyz 成分を正規化して符号を反転します。また、光源方向のベクトル L の正反射方向のベクトル R は、GLSL の組み込み関数 reflect() を使って求めることができます。このとき光源方向のベクトル L の向きは、式 (83) に対して反対になります。なお、これらの負号は後の dot(R, V) で打ち消されるので、反転しなくても結果は同じです。

この入射光の正反射方向のベクトル R と視点方向のベクトル V の内積を求め、式 (85) にもとづいて鏡面反射光強度を求め、Ispec に代入します。

```
#version 150 core
uniform mat4 modelview;
uniform mat4 projection;
uniform mat3 normalMatrix;
const vec4 Lpos = vec4(0.0, 0.0, 5.0, 1.0);
```

```
const vec3 Ldiff = vec3(1.0);
const vec3 Lspec = vec3(1.0);
const vec3 Kdiff = vec3(0.6, 0.6, 0.2);
const vec3 Kspec = vec3(0.3, 0.3, 0.3);
const float Kshi = 30.0;
in vec4 position;
in vec3 normal;
out vec3 Idiff;
out vec3 Ispec;
void main()
{
  vec4 P = modelview * position;
  vec3 N = normalize(normalMatrix * normal);
  vec3 L = normalize((Lpos * P.w - P * Lpos.w).xyz);
  Idiff = max(dot(N, L), 0.0) * Kdiff * Ldiff;
  vec3 V = -normalize(P.xyz);
  vec3 R = reflect(-L, N);
  Ispec = pow(max(dot(R, V), 0.0), Kshi) * Kspec * Lspec;
  gl_Position = projection * P;
}
```

genType reflect(genType I, genType N)

法線ベクトルが N の点に入射するベクトル I の正反射方向のベクトルを戻り値として返す、GLSL の組み込み関数です。I – 2.0 * dot(N, I) * N を返します。

I

入射ベクトル。

N

入射点の法線ベクトル。

フラグメントシェーダ (point.frag) の変更点

バーテックスシェーダの out 変数 Ispec の補間値を受け取る in 変数 Ispec をフラグメントシェーダに追加します。これと Idiff との和を vec4 に型変換し、アルファ値の第 4 要素に 1 (不透明) を設定して、フラグメントの色の out 変数 fragment に代入します。

```
#version 150 core
in vec3 Idiff;
in vec3 Ispec;
out vec4 fragment;
void main()
{
  fragment = vec4(Idiff + Ispec, 1.0);
}
```

■ サンプルプログラム step30

鏡面反射光に関しては、視点方向のベクトル **V** と正反射方向のベクトル **R** となす角 θ_r の代わりに、視点方向のベクトル **V** と光源方向のベクトル **L** の中間ベクトル **H** が物体表面の法線

単位ベクトル **N** となす角 θ_h を用いるモデルもあります (図 149)。

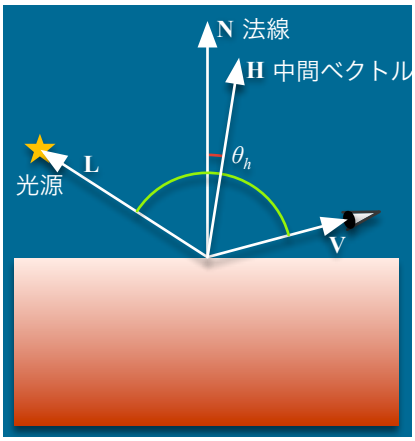


図 149 中間ベクトルを用いた鏡面反射光強度の算出

$$\mathbf{H} = \frac{\mathbf{L} + \mathbf{V}}{|\mathbf{L} + \mathbf{V}|} \tag{86}$$

$$I_{spec} = \cos^{K_{shi}} \theta_h K_{spec} \otimes L_{spec} = (\mathbf{N} \cdot \mathbf{H})^{K_{shi}} K_{spec} \otimes L_{spec} \tag{87}$$

これも **N** と **H** のなす角 θ_h が $\pi/2$ 以上、すなわち **N**・**H** が負のときは光源が物体表面の裏側にあるので、鏡面反射光強度は 0 にします。

$$I_{spec} = \max(\mathbf{N} \cdot \mathbf{H}, 0)^{K_{shi}} K_{spec} \otimes L_{spec} \tag{88}$$

これは **Blinn-Phong の陰影付けモデル**あるいは **Blinn の陰影付けモデル**と呼ばれます。このモデルによる鏡面反射光による陰影は、のようになります。球面に対する陰影は K_{shi} が同じであれば Phong の陰影付けモデルより若干広がりますが、平面に対する陰影は入射角が浅いときに縦方向 (前後方向) に伸びたものになります。

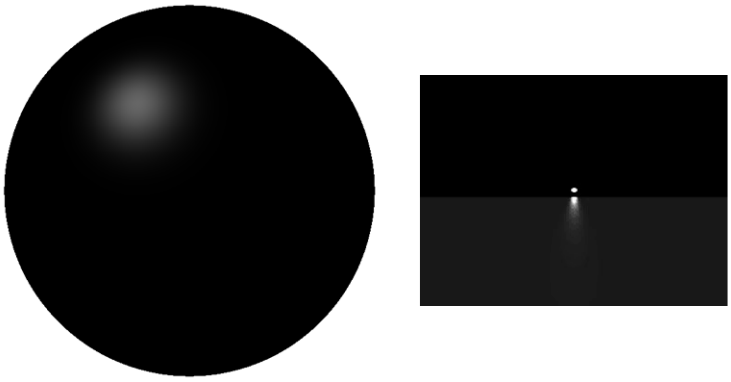


図 150 Blinn-Phong の陰影付けモデルによる鏡面反射光の陰影の例

Blinn-Phong の陰影付けモデルは、物体表面の微小な凹凸による反射をより精密にモデル化し

ようとする微小面理論 (microfacet theory) の考え方を、Phong の陰影付けモデルに反映したものといえます。微小面理論では物体表面の凹凸を多数の微小な反射面の集まりであるととらえ、その方向 (法線ベクトル) **n** が光源方向のベクトル **L** と視点方向のベクトル **V** の中間の方向 (中間ベクトル) **H** と一致する微小面が、光源の光を視点に届ると考えます (図 151)。

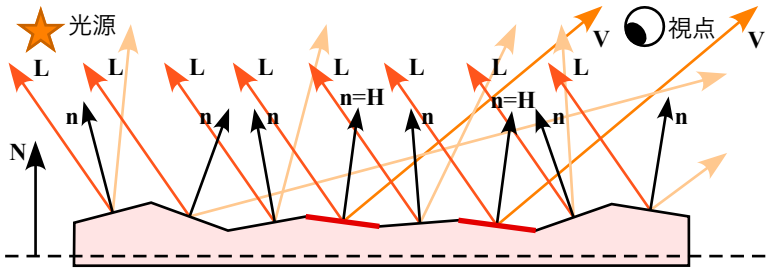


図 151 微小面理論の考え方

したがって、反射率は **n** が **H** と一致する確率で決定されます。つや消しのような不規則な凹凸では、**n** は土台の法線ベクトル **N** を平均値とした正規分布でモデル化できます。Blinn-Phong のモデルは、それを正規分布と形の似た $\cos^{Kshi}\theta_h$ で置き換えたものと解釈することができます。

● バーテックスシェーダ (point.vert) の変更点

光源方向のベクトル L の正反射方向のベクトル R の代わりに、L と視点方向のベクトル V との中間ベクトル H を式 (86) により求めます。これは V を求めるときに符号を反転せず、H を求めるときに normalize(L – V) としても構いません。この中間ベクトル H と法線ベクトル N の内積を求め、式 (88) にもとづいて鏡面反射光強度を求め、Ispec に代入します。

```
#version 150 core
《省略》
void main()
{
  vec4 P = modelview * position;
  vec3 N = normalize(normalMatrix * normal);
  vec3 L = normalize((Lpos * P.w - P * Lpos.w).xyz);
  Idiff = max(dot(N, L), 0.0) * Kdiff * Ldiff;
  vec3 V = -normalize(P.xyz);
  vec3 H = normalize(L + V);
  Ispec = pow(max(dot(N, H), 0.0), Kshi) * Kspec * Lspec;
  gl_Position = projection * P;
}
```

■ サンプルプログラム step31

9.1.4 環境光の反射光

拡散反射光や鏡面反射光は、光源から直接届いた入射光、すなわち直接光の反射光です。しかし物体の表面には、光源から直接届かず、他の面で反射されて届く間接光や、天空光のような大

域的な光源からの光も届きます。ところが、一般に間接光の経路は複雑であり、それを正確に見積もることは容易ではありません。物体表面で反射した光が、めぐりめぐって再び同じ物体表面に到達することも考えられます。この場合、非常に長い計算時間がかかってしまいます。

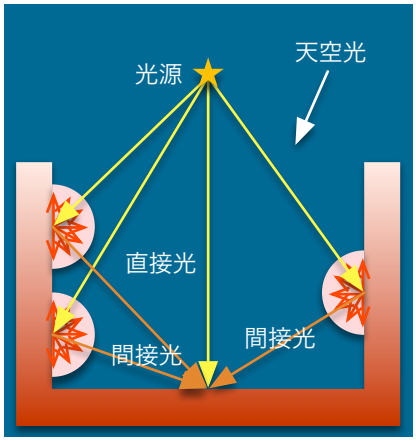


図 152 直接光と間接光

そこで、もう間接光を見積もることは諦めて、定数で表すという近似を行います。これが**環境光**です。環境光の強度を L_{amb}、環境光に対する反射係数を K_{amb} とするとき、この物体表面における環境光の反射光強度 I_{amb} は次式で求められます。

$$I_{amb} = K_{amb} \otimes L_{amb} \tag{89}$$

非金属の材質では、一般に K_{amb} は拡散反射光の反射係数 K_{diff} と同じにします。また L_{amb} には一般に光源強度の拡散反射光成分 L_{diff} や鏡面反射光成分 L_{spec} に比べて小さな値を設定します。大きな値を設定すると、陰影の失われた表示になります。

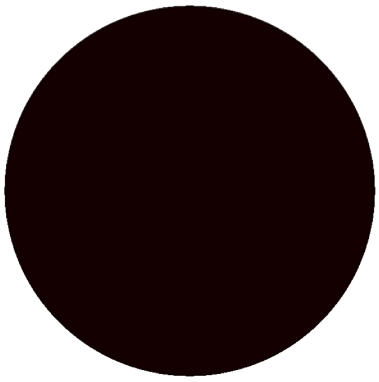


図 153 環境光の反射光による陰影の例

● **バーテックスシェーダ (point.vert) の変更点**

バーテックスシェーダに環境光強度 Lamb および環境光に対する反射係数 Kamb をそれぞれ定数で与えます。一般に環境光強度 Lamb には、光源強度の拡散反射光成分 Ldiff を減衰させた

値を設定します。また 環境光に対する反射係数 Kamb には拡散反射係数 Kdiff と同じ値を設定します。この Kamb と Lamb を乗じて、環境光の反射光強度 Iamb を求めます。

なお、この Iamb は out 変数にせず、Idiff にそのまま加えます。したがって、Kamb に Kdiff と同じ値を設定するなら、Kamb を別に定めずに Idiff = (max(dot(N, L), 0.0) * Ldiff + Lamb) * Kdiff; とすることもできます。

```
#version 150 core
《省略》
const vec4 Lpos = vec4(0.0, 0.0, 5.0, 1.0);
const vec3 Lamb = vec3(0.2);
const vec3 Ldiff = vec3(1.0);
const vec3 Lspec = vec3(1.0);
const vec3 Kamb = vec3(0.6, 0.6, 0.2);
const vec3 Kdiff = vec3(0.6, 0.6, 0.2);
const vec3 Kspec = vec3(0.3, 0.3, 0.3);
const float Kshi = 30.0;
《省略》
void main()
{
  vec4 P = modelview * position;
  vec3 N = normalize(normalMatrix * normal);
  vec3 L = normalize((Lpos * P.w - P * Lpos.w).xyz);
  vec3 Iamb = Kamb * Lamb;
  Idiff = max(dot(N, L), 0.0) * Kdiff * Ldiff + Iamb;
  vec3 V = -normalize(P.xyz);
  vec3 H = normalize(L + V);
  Ispec = pow(max(dot(N, H), 0.0), Kshi) * Kspec * Lspec;
  gl_Position = projection * P;
}
```

■ サンプルプログラム step32

9.1.5 反射光強度

このように反射光強度は、拡散反射光強度 I_{diff}、鏡面反射光強度 I_{spec}、環境光の反射光強度 I_{amb} を合計して得られます。

$$I_{tot} = I_{amb} + I_{diff} + I_{spec} \tag{90}$$

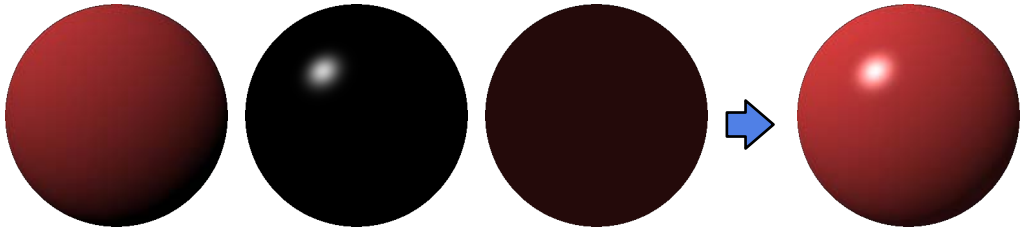


図 154 反射光強度

9.1.6　球の描画

これまで描いてきた立方体は反射面が平面なので、鏡面反射光が観測される視点の方向が限られてしまいます。そこで、ここで立方体に代えて球を描くようにしてみます。ただし、OpenGL には曲面を直接描く機能はないので、多面体で近似することになります。その場合でも、隣接するポリゴンと頂点を共有していれば、図 119 のように隣接するポリゴンの境界で色がなだらかに変化するので、境界線を目立ちにくくすることができます。これは**スムーズシェーディング**と呼ばれるテクニックです。スムーズシェーディングについては 9.4 節で解説します。

図 155 (a) の球の表面を同図 (b) のように緯度と経度でそれぞれ等角度に分割します。これを展開して、同図 (c) のような格子にします。この格子は緯線と経線が直角に交わっており、それぞれは等間隔になっています。地球儀をこのような形で展開して世界地図を描く描き方は、正距円筒図法と呼ばれます。

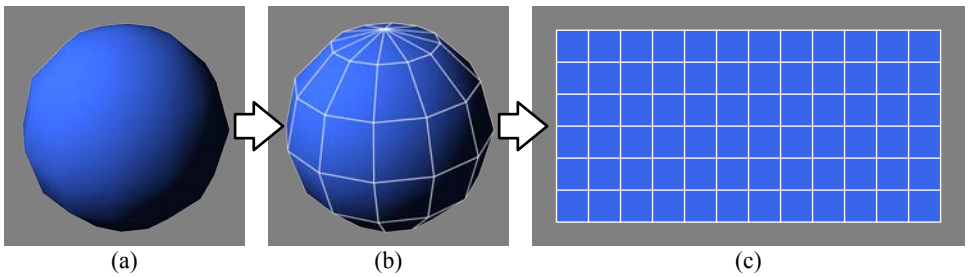


図 155 球の展開

図 156 のように、一辺の長さが 1 の正方形を、横方向に *slices* 個、縦方向に *stacks* 個のマス目に分割します。この格子の各交点 (格子点) の位置 (s, t) は、経度方向の格子点の番号を i、緯度方向の格子点の番号を j とすると、それぞれ次のように求められます。

$$s = \frac{i}{slices},\ t = \frac{j}{stacks} \tag{91}$$

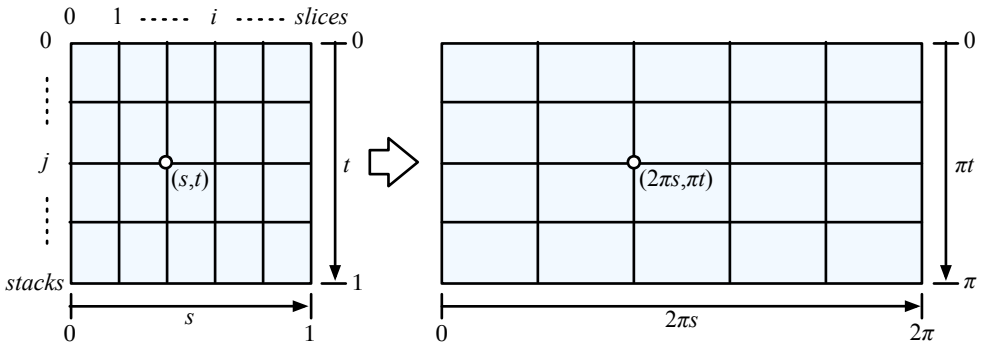


図 156 格子の作成

これをもとに、半径 1 の単位球面上の点の位置 (x, y, z) を求めます (図 157)。

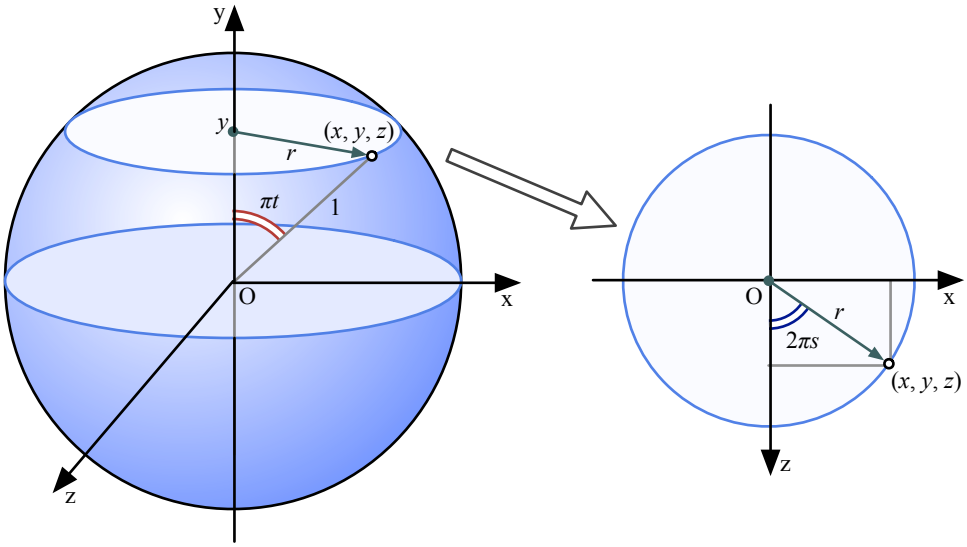


図 157 格子点位置の算出

● メインプログラム (main.cpp) の変更点

ここでは三角関数を使うので、main.cpp の冒頭で cmath を #include しておきます。また、これまで表示していた立方体の頂点属性は定数で表していましたが、この方法で球を表現しようとすると大変になります。そこで、頂点属性をプログラムで生成することにします。main() 関数に、以下のプログラムを追加します。頂点属性は配列ではなく、vector に格納します。

```
#include <cmath>
《省略》

int main()
{
  《省略》

  // uniform 変数の場所を取得する
  const GLint modelviewLoc(glGetUniformLocation(program, "modelview"));
  const GLint projectionLoc(glGetUniformLocation(program, "projection"));
  const GLint normalMatrixLoc(glGetUniformLocation(program, "normalMatrix"));

  // 球の分割数
  const int slices(16), stacks(8);

  // 頂点属性を作る
  std::vector<Object::Vertex> solidSphereVertex;
```

まず、(s, t) の格子点における高さ y と、y における単位球の断面の半径 r を求めます。

$$\begin{aligned} y &= \cos(\pi t) \\ r &= \sin(\pi t) \end{aligned} \tag{92}$$

```
  for (int j = 0; j <= stacks; ++j)
  {
```

```
    const float t(static_cast<float>(j) / static_cast<float>(stacks));
    const float y(cos(3.141593f * t)), r(sin(3.141593f * t));
```

j と stacks は整数 (int 型) なので、ここで “j / stacks” としてしまうと整数の除算になってしまうので、実数 (float 型) に変換してから除算します。“3.141593f * t” は πt です。

次に、この断面の円周上の点の位置の x と z を求めます。

$$\begin{aligned} z &= r\cos(2\pi s) \\ x &= r\sin(2\pi s) \end{aligned} \tag{93}$$

```
    for (int i = 0; i <= slices; ++i)
    {
      const float s(static_cast<float>(i) / static_cast<float>(slices));
      const float z(r * cos(6.283185f * s)), x(r * sin(6.283185f * s));
```

求めた x、y、z を使って、頂点属性をひとつ作ります。頂点属性のデータ型の Object::Vertex は 3 要素の位置座標と 3 要素の法線ベクトルで構成されています (9.1.1 節)。原点を中心とする単位球では、球面上の任意の点における単位法線ベクトルがその点の位置と一致するので、位置と法線ベクトルに同じ値を設定します。これを、頂点属性を格納する vector に追加します。

```
      // 頂点属性
      const Object::Vertex v = { x, y, z, x, y, z };

      // 頂点属性を追加する
      solidSphereVertex.emplace_back(v);
    }
  }
```

つぎに、インデックスのデータを作成します。このプログラムで使っている SolidShapeIndex クラス (8.3.1) では図形を GL_TRIANGLES、すなわち独立した三角形で描画しますから、格子を三角形に分割して考えます。この格子の横 i 番目、縦 j 番目の格子点の番号 k_0 は、格子点の横方向の数が $slices + 1$ ですから (図 160)、次式で求めることができます。

$$k_0 = (slices + 1)\, j + i \tag{94}$$

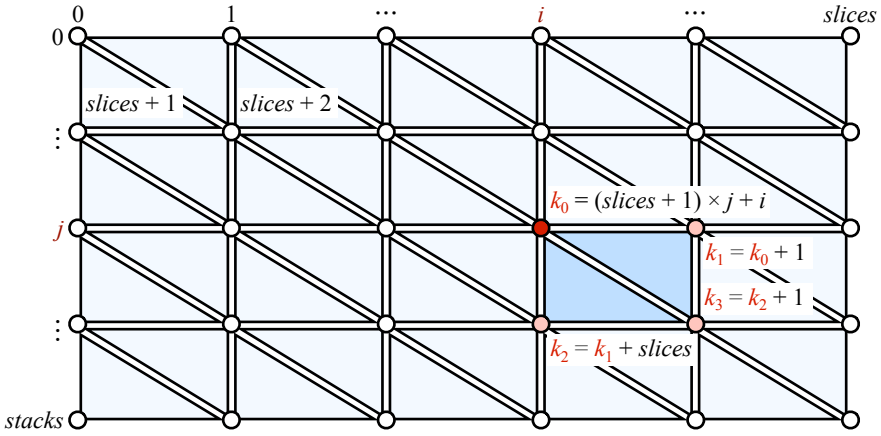


図 158 三角形分割された格子と格子点の番号

また、k_0 を左上の格子点とするマス目の残りの格子点の番号は、それぞれ以下のようになります。これをすべてのマス目について求め、二つの三角形の頂点のインデックスとして格納します。このインデックスにも配列ではなく、vector を用います。

$$
\begin{aligned}
k_1 &= k_0 + 1 \\
k_2 &= k_1 + slices \\
k_3 &= k_2 + 1
\end{aligned}
\tag{95}
$$

```
  // インデックスを作る
  std::vector<GLuint> solidSphereIndex;

  for (int j = 0; j < stacks; ++j)
  {
    const int k((slices + 1) * j);

    for (int i = 0; i < slices; ++i)
    {
      // 頂点のインデックス
      const GLuint k0(k + i);
      const GLuint k1(k0 + 1);
      const GLuint k2(k1 + slices);
      const GLuint k3(k2 + 1);
```

これらの格子点の番号を、このマス目の左下と右上の三角形の頂点のインデックスとして、インデックスを格納する vector に追加します。

```
      // 左下の三角形
      solidSphereIndex.emplace_back(k0);
      solidSphereIndex.emplace_back(k2);
      solidSphereIndex.emplace_back(k3);

      // 右上の三角形
      solidSphereIndex.emplace_back(k0);
      solidSphereIndex.emplace_back(k3);
      solidSphereIndex.emplace_back(k1);
```

```
    }
  }
```

この二つの vector を使って、SolidShapeIndex クラスのオブジェクトを作成します。vector クラスに格納されているデータの配列のポインタを取り出すには、data() というメソッドを使います。また、それに格納されているデータの数は、size() というメソッドで調べられます。ただし、size() の戻り値は size_t 型であり、これは 64bit のオペレーティングシステムでは int などより大きな数を表すことができるため、これを GLsizei 型 (int 型相当) に変換します。

```
  // 図形データを作成する
  std::unique_ptr<const Shape> shape(new SolidShapeIndex(3,
    static_cast<GLsizei>(solidSphereVertex.size()), solidSphereVertex.data(),
    static_cast<GLsizei>(solidSphereIndex.size()), solidSphereIndex.data()));

  // タイマーを 0 にセット
  glfwSetTime(0.0);
```

■ サンプルプログラム step33

● 実行結果

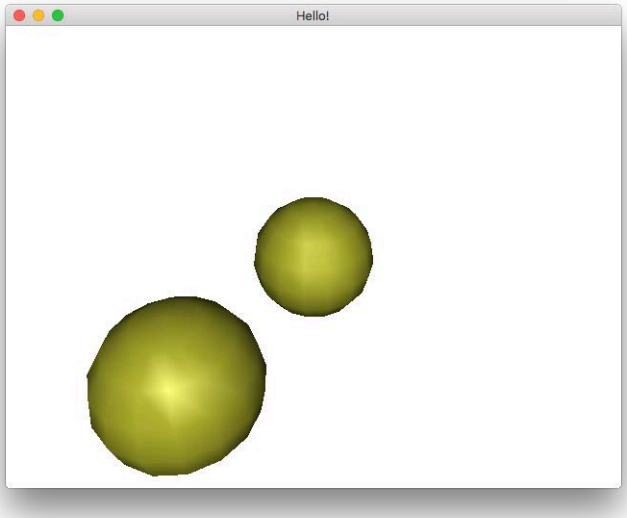


図 159 多面体で近似した球の描画

ポリゴンの境界がかなり目立ちますが、これは slices と stacks を増やすか、あるいは 9.4 節で解説する Phong のスムーズシェーディングにより解消することができます。

9.1.7 光源の種類

これまで、入射光は**平行光線**として扱ってきました。3D CG で用いられる最も基本的な光源のモデルには、この他に**点光源**と**スポットライト**があります。

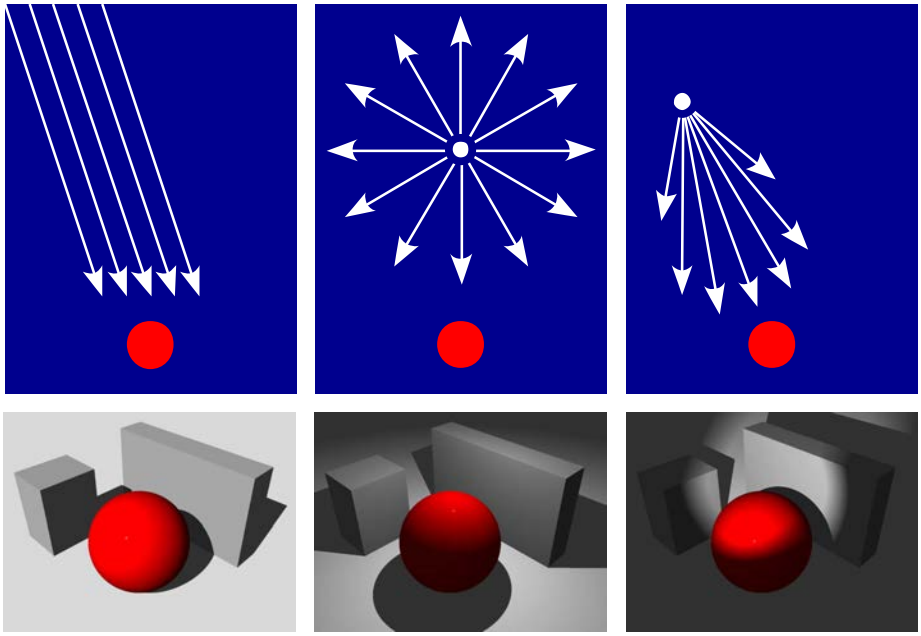


図 160 基本的な光源

点光源は光源が物体から見て有限の位置にあり、すべての方向に均等に光を放射しているものです。点光源と同じく光源が点ながら特定の方向に光を放射しているものは、**スポットライト**と呼ばれます。なお、平行光線は光源が無限の彼方にあると考えますから、これも物体から見れば光源は大きさを持っていない点である、ということになります。

● **平行光線**

平行光線は光源の位置の同次座標の w 要素を 0 にするだけです。物体の頂点における光線の方向ベクトルは、光源位置から頂点の位置を引いたものですが、これを式 (11) の同次座標の減算で行えば、点光源と平行光線を区別することなく取り扱うことができます。

● **点光源**

点光源では、光源により照明される面積が光源からの距離の二乗に比例します。このため、物体表面における入射光の密度は距離の二乗に反比例することになります。したがって点光源では、入射光強度を光源からの距離の二乗で割る必要があります。

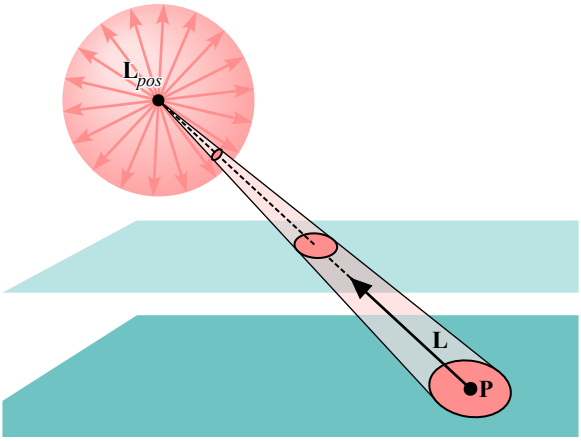


図 161 点光源

光源の位置を $\mathbf{L}_{pos}$、物体表面上の点の位置を $\mathbf{P}$ とすると、点 $\mathbf{P}$ における光源との距離による減衰 d は、以下により求められます。

$$d = \frac{1}{|\mathbf{L}_{pos} - \mathbf{P}|^2} \tag{96}$$

したがって、これを考慮した反射光強度は、d を拡散反射光強度と環境光の反射光強度に乗じて、次のように求めることができます。

$$I_{tot} = I_{amb} + d\,(I_{diff} + I_{spec}) \tag{97}$$

ただし、この減衰は非常に急なため、光源が物体から少しでも離れると、途端に物体が暗くなってしまいます。CG のシーン中に点光源が一つだけあると言うのは、例えば部屋の中にろうそくが 1 本あるような状況に相当します。このとき、ろうそくの炎は強く輝いているのに部屋は全然明るくない、ということが起こることがあります。

現実の照明環境では、光源が複数存在することがほとんどですし、間接光も光源として別の物体の照明に寄与します。そのため、一つの点光源の移動によって、離れたところの照明が急激に暗くなるということは起こりません。しかし、そのような複雑な光源環境の正確な再現には、長い計算時間がかかります。

そこで、距離の二乗に反比例する値の代わりに実用上は距離に反比例する値を用いたり、距離によらず一定の値を用いたりすることがあります。これらの三つの距離による減衰を一つにまとめて、次式で表します。

$$d = \frac{1}{s_c + s_l\,|\mathbf{L}_{pos} - \mathbf{P}| + s_q\,|\mathbf{L}_{pos} - \mathbf{P}|^2} \tag{98}$$

ここで s_c は距離によらない場合の減衰係数、s_l は距離に反比例する場合の減衰係数、s_q は距離の二乗に反比例する減衰係数です。この (s_c, s_l, s_q) の組を**距離減衰係数**といいます。$(s_c, s_l, s_q) =$

(0, 0, 1) なら距離の二乗に反比例、(0, 1, 0) なら距離に反比例、(1, 0, 0) なら一定になります。

● スポットライト

スポットライトの光の投射方向を $\mathbf{L}_{dir}$ とするとき、その配光分布 c_{spot} には $\mathbf{L}_{dir}$ と光源方向の逆ベクトル $-\mathbf{L}$ との内積のべき乗を用います。べき指数 s_{exp} はスポットライトの広がりを制御する係数で、これが大きいほどスポットライトは鋭くなります。

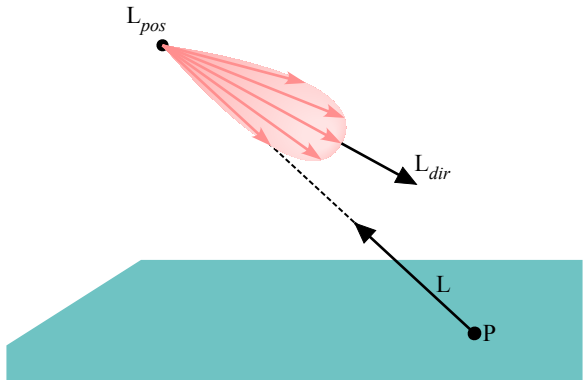


図 162 スポットライト

$$c_{spot} = \max\left(-\mathbf{L} \cdot \mathbf{L}_{dir}, 0\right)^{s_{exp}} \tag{99}$$

反射光強度は、距離減衰を考慮した反射光強度 (式 (97)) にこれを乗じて求めます。

$$I_{tot} = c_{spot}\left\{I_{amb} + d\left(I_{diff} + I_{spec}\right)\right\} \tag{100}$$

式 (99) の配光分布は、スポットライトの中心から離れるにしたがって、明るさが徐々に低下します。実際のスポットライトを真似るなら、中心から一定の範囲の明るさはあまり変化せず、それより外側は急激に暗くした方が良いように思われます。そこでスポットライトの方向 $\mathbf{L}_{dir}$ に対する角度 θ_s が一定の角度 θ_u を超えた方向では、光源の明るさを 0 にしてしまうという方法があります (式 (101))。

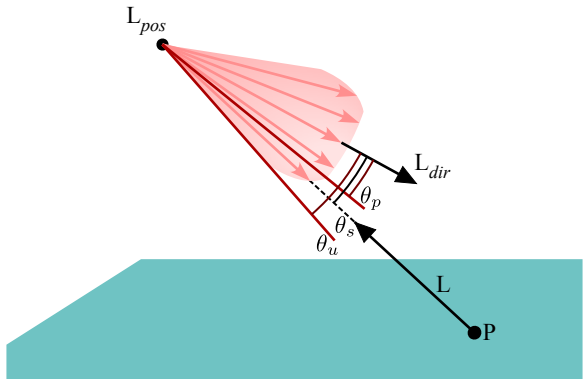


図 163 スポットライトのバリエーション

$$c_{spot} = \begin{cases} \cos^{s_{exp}} \theta_s & \theta_s \leq \theta_u \\ 0 & \theta_s > \theta_u \end{cases} \tag{101}$$

ただし、この方法でも角度 θ_s が θ_u より小さい範囲では、明るさが徐々に変化してしまいます。そこで、一定の角度 θ_p までは明るさを一定にし、それを超えてから明るさを徐々に下げ、θ_u を超えれば 0 にするという方法もあります。

$$c_{spot} = \begin{cases} 1 & \cos\theta_s \geq \cos\theta_p \\ \cos^{s_{exp}} \theta_s & \cos\theta_u < \cos\theta_s < \cos\theta_p \\ 0 & \cos\theta_s \leq \cos\theta_u \end{cases} \tag{102}$$

現実の照明器具の配光分布 (配光特性) は複雑な形をしており、CG においても、その実測値を用いて精密な陰影を求めることが可能です。しかし一般には、これらのような単純な分布がよく用いられます。

● 照明方程式

光源が複数あるときは、反射光は光源ごとの反射光の総和になります。光源の数が n 個あるときは、次のようになります。ここで I^k_{amb}、I^k_{diff}、I^k_{spec} は、それぞれ k 番目の光源による環境光の反射光強度、拡散反射光強度、鏡面反射光強度です。

$$I_{tot} = \sum_{k=1}^{n} \left\{ I^k_{amb} + I^k_{diff} + I^k_{spec} \right\} \tag{103}$$

さらに、k 番目の光源の距離減衰係数を d^k、配光分布を c^k_{spot} を考慮すれば、反射光強度は次式で求めることができます.

$$I_{tot} = \sum_{k=1}^{n} c^k_{spot} \left\{ I^k_{amb} + d^k \left(I^k_{diff} + I^k_{spec} \right) \right\} \tag{104}$$

これに、これらのどの光源にも依存しない環境光 (背景光) L_{glob} と、物体自体が発光する場合の発光強度 (自己発光) L_{emi} をついかして、最終的な照明方程式を得ます。

$$I_{tot} = K_{amb} \otimes L_{glob} + L_{emi} + \sum_{k=1}^{n} c^k_{spot} \left\{ I^k_{amb} + d^k \left(I^k_{diff} + I^k_{spec} \right) \right\} \tag{105}$$

補足：リアルな陰影計算

式 (105) の照明方程式 (陰影付けモデル) は、光源を点とみなし、拡散反射光は入射光の入射点から放出されるとしています。これは、光源をそのように取り扱うことで、陰影付けの計算量を大幅に削減できるからです。これによりゲームなどの対話的なアプリケーションに求められる良好な応答速度が得られますが、陰影のリアリティは限定的なものになってしまいます。

物体内部に進入した入射光は散乱と吸収を繰り返したのち、現実には入射点の周囲から放射されます (図 164)。この放射される光の強度は放射点が入射点から離れるに従って指数関数的に減少するため、視点の位置が物体表面から十分遠ければ、相対的に入射点と放射点の位置が一致すると見なすことができます。しかし、物体表面の光の透過率が高く、視点が物体表面に近ければ、物体に透明感が出てきます。このような反射は**表面下散乱** (Sub-surface scattering) と呼ばれます。

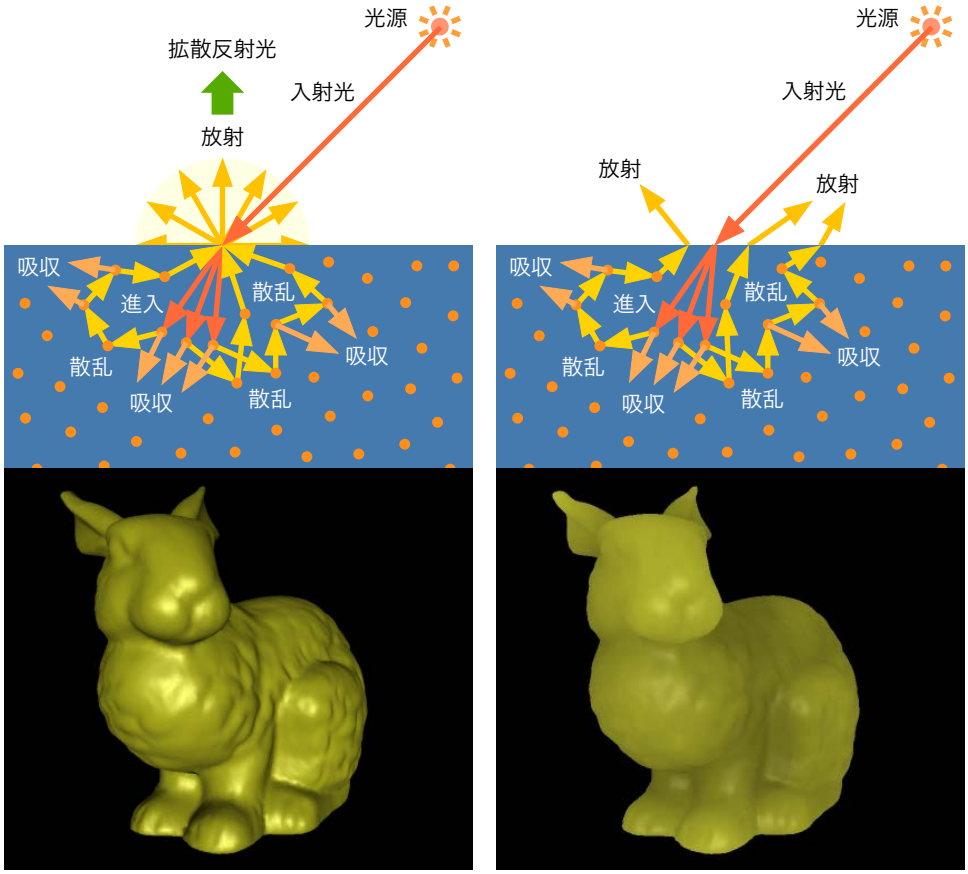


図 164 表面下散乱の有無

また、環境光の反射光強度を定数で表すと言うのも大胆な近似です。この場合、環境光しか存在しない状況では、陰影は変化のない平板なものになってしまいます。しかし間接光 (相互反射) を考慮すれば、光の届きにくい込み入ったところは暗くなります (図 165)。

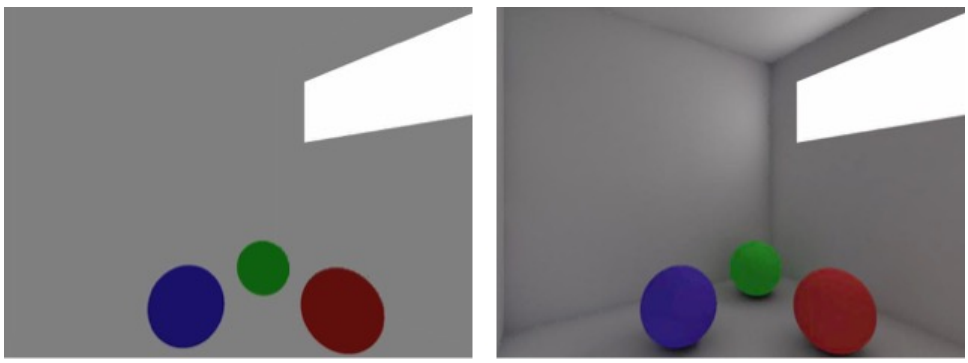


図 165 環境光を定数にした場合と間接光を考慮した場合

間接光を考慮した陰影付けモデルは、**大域照明** (Global Illumination) モデルと呼ばれます。大域照明モデルでは、拡散反射面からの間接光によって、別の面にその色がつく**カラーブリーディング** (Color Bleeding) という現象の再現が可能です。(まるさんありがとう)

大域照明モデルでは、大きさを持った光源を取り扱うこともできます。ここでは影を取り扱っていませんが、光源が点であれば影の境界はくっきりしたものになるのに対し、光源の大きさを考慮すれば影の境界は影が物体から離れるにしたがってぼけたものになります (図 166)。また、空全体が一様な明るさを持つ光源 (天空光) による陰影を求めることもできます。

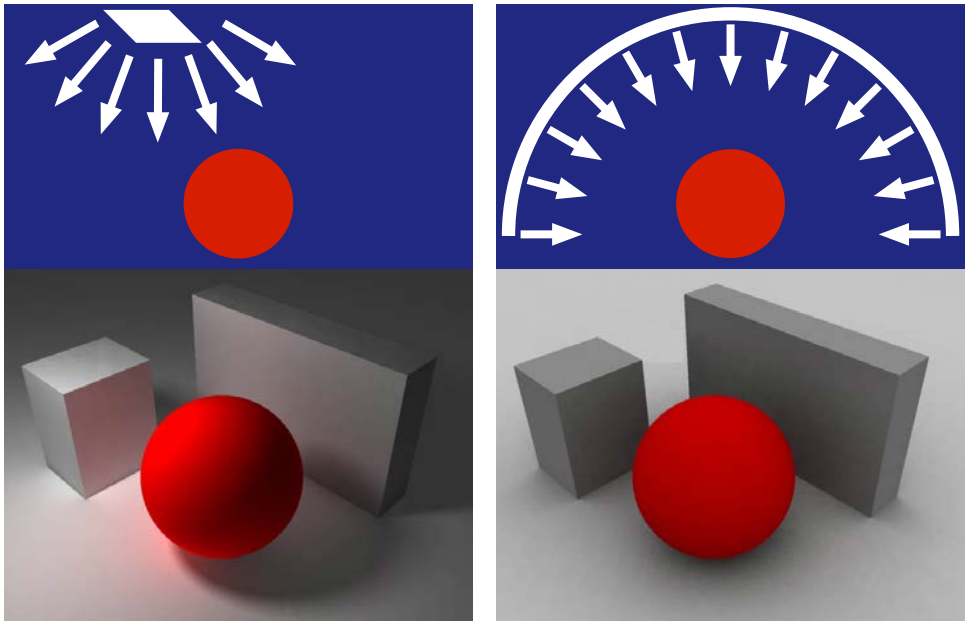


図 166 面積を持った光源と天空光

大域照明モデルは反射光強度の算出に多くの要素を考慮する必要があるため、計算時間が長くかかります。そのため、OpenGL が対象とするリアルタイム 3D CG への導入には、多くの困難が伴います。このリアリティとリアルタイム性にはトレードオフの関係があります。

しかし、現在では GPU の高性能化や効率的な計算手法の開発によって、非常にリアルな陰影をリアルタイムに再現できるようになってきています。また、これらの機能はゲームエンジンなどのミドルウェアに組み込まれており、手軽に利用できるようになっています。

9.2 光源のデータ

まず、光源のデータをメインプログラムから設定できるようにします。

9.2.1 光源のデータの uniform 変数による設定

これまで、陰影付けに使う光源や材質のデータは、シェーダのソースプログラムの中に定数で記述していました。これでは描画する物体や面ごとに色を変えたり、プログラムの実行中に光源の色や位置を変えたりすることが難しくなります。

そこで、これらもデータとして準備し、描画時にメインプログラム (C++ のプログラム) からシェーダプログラムに与えるようにします。これらも図形の描画中 (一回の描画命令の実行中) に変更することはないので、uniform 変数に格納してシェーダプログラムに渡します。

● **バーテックスシェーダ** (point.vert) **の変更点**

シェーダプログラムの光源のデータの定数を、uniform 変数に変更します。データ型と**初期化並び**はそのままにしておきます。この初期化並びは、その uniform 変数にメインプログラムから値が設定されなかった場合の初期値 (デフォルト値) になります。

```
#version 150 core
uniform mat4 modelview;
uniform mat4 projection;
uniform mat3 normalMatrix;
uniform vec4 Lpos = vec4(0.0, 0.0, 5.0, 1.0);
uniform vec3 Lamb = vec3(0.2);
uniform vec3 Ldiff = vec3(1.0);
uniform vec3 Lspec = vec3(1.0);
const vec3 Kamb = vec3(0.6, 0.6, 0.2);
const vec3 Kdiff = vec3(0.6, 0.6, 0.2);
const vec3 Kspec = vec3(0.3, 0.3, 0.3);
const float Kshi = 30.0;
```

この状態でプログラムを実行しても (シェーダプログラムだけを変更したときはビルドする必要はありません)、以前と変わらない表示が得られるはずです。

● **メインプログラム** (main.cpp) **の変更点**

それでは、これらの uniform 変数の値をメインプログラムから設定してみましょう。まず、光源の uniform 変数の場所を glGetUniformLocation() を使って取り出しておきます。

```
  // uniform 変数の場所を取得する
  const GLint modelviewLoc(glGetUniformLocation(program, "modelview"));
  const GLint projectionLoc(glGetUniformLocation(program, "projection"));
  const GLint normalMatrixLoc(glGetUniformLocation(program, "normalMatrix"));
  const GLint LposLoc(glGetUniformLocation(program, "Lpos"));
  const GLint LambLoc(glGetUniformLocation(program, "Lamb"));
  const GLint LdiffLoc(glGetUniformLocation(program, "Ldiff"));
  const GLint LspecLoc(glGetUniformLocation(program, "Lspec"));
```

また、これらの uniform 変数に設定する uniform 変数のデータを別に用意します。光源の色は、青の成分と緑の成分を下げて少し赤くしてみましょう。(まてぃさんありがとう)

```
  // 図形データを作成する
  std::unique_ptr<const Shape> shape(new SolidShapeIndex(3,
    static_cast<GLsizei>(solidSphereVertex.size()), solidSphereVertex.data(),
    static_cast<GLsizei>(solidSphereIndex.size()), solidSphereIndex.data()));

  // 光源データ
```

```
  static constexpr GLfloat Lpos[] = { 0.0f, 0.0f, 5.0f, 1.0f };
  static constexpr GLfloat Lamb[] = { 0.2f, 0.1f, 0.1f };
  static constexpr GLfloat Ldiff[] = { 1.0f, 0.5f, 0.5f };
  static constexpr GLfloat Lspec[] = { 1.0f, 0.5f, 0.5f };

  // タイマーを 0 にセット
  glfwSetTime(0.0);
```

そして glUseProgram() でシェーダプログラムを選択した後、図形を描画する前に、これらの uniform 変数に値を設定します。

```
    // モデルビュー変換行列を求める
    const Matrix modelview(view * model);

    // 法線ベクトルの変換行列を求める
    modelview.getNormalMatrix(normalMatrix);

    // uniform 変数に値を設定する
    glUniformMatrix4fv(projectionLoc, 1, GL_FALSE, projection.data());
    glUniformMatrix4fv(modelviewLoc, 1, GL_FALSE, modelview.data());
    glUniformMatrix3fv(normalMatrixLoc, 1, GL_FALSE, normalMatrix);
    glUniform4fv(LposLoc, 1, Lpos);
    glUniform3fv(LambLoc, 1, Lamb);
    glUniform3fv(LdiffLoc, 1, Ldiff);
    glUniform3fv(LspecLoc, 1, Lspec);

    // 図形を描画する
    shape->draw();
```

■ サンプルプログラム step34

● 実行結果

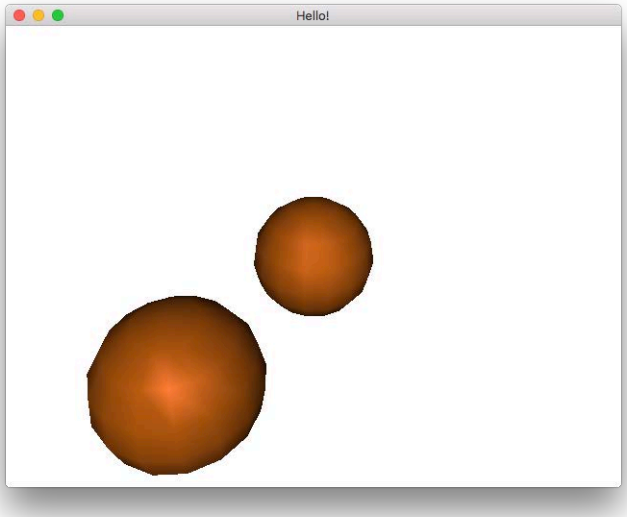


図 167 光源の色を変更した結果

9.2.2 光源位置の変更

光源の位置は、現時点では物体の頂点の座標値を視点座標系に変換したものに対して設定しています。つまり、光源の位置も視点座標系にあるため、現時点の設定のように $(0, 0, 5)$ であれば視点座標系の z 軸上、すなわち視線上にあります。このため、この光源による陰影は、「ヘッドライト」のようなものになっています。

この光源の位置をワールド座標系上に固定するには、光源の位置にビュー変換行列をかけたものを用いて陰影付け処理を行います。この処理はシェーダプログラム上でも可能ですが、光源のモデル変換行列を物体とは独立に設定するために、メインプログラム側であらかじめビュー変換行列をかけてシェーダプログラムに渡すことにします。

● **ベクトルのデータ型 Vector (Vector.h)**

これまで座標値は配列変数に格納していましたが、これをメインプログラム側で変換行列のクラス Matrix と組み合わせるために、新たに Vector というデータ型を用意します。標準ライブラリの vector と紛らわしくてごめんなさい。これを Vector.h というヘッダファイルに定義します。

この Vector はクラスや構造体ではなく、標準ライブラリの array を用いて定義します。そのために、ヘッダファイル array を Vector.h の冒頭で #include します。この他に、Vector と組み合わせる Matrix クラスを定義している Matrix.h も #include します。

array は vector と同様に連続的にメモリを確保しますが、要素数の動的な拡張は行わず、通常の配列と同様に要素数は固定になります。同次座標や RGBA で表現する色は要素数がともに 4 なので、それらの値を格納する 4 要素の array の別名として Vector を宣言します。

```
#pragma once
#include <array>

// 変換行列
#include "Matrix.h"

// ベクトル
using Vector = std::array<GLfloat, 4>;
```

そして、演算子 '*' をオーバーライドしてクラス Matrix と Vector との積を計算する関数を定義します。Matrix クラスの data() メソッドは変換行列を格納しているメンバの配列のポインタを返しますから、[ ] 演算子により個々の要素の値を取り出すことができます。それを Vector の各要素に乗じます。

この乗算の結果は Vector 型の変数 t に格納します。この個々の要素も配列と同じように [ ] 演算子により参照することができますが、変数名 t そのものはポインタではなくデータ全体を示します。したがって return t; により、この関数の戻り値 ('*' 演算子による乗算の結果) は Vector 型のデータ全体が返されます。

```
// 行列とベクトルの乗算
//   m: Matrix 型の行列
//   v: Vector 型のベクトル
Vector operator*(const Matrix &m, const Vector &v)
{
  Vector t;

  for (int i = 0; i < 4; ++i)
  {
    t[i] = m[i] * v[0] + m[i + 4] * v[1] + m[i + 8] * v[2] + m[i + 12] * v[3];
  }

  return t;
}
```

● メインプログラム (main.cpp) の変更点

この Vector.h をメインプログラムの冒頭で #include します。

```
#include <cmath>
#include <cstdlib>
#include <iostream>
#include <fstream>
#include <vector>
#include <memory>
#include <GL/glew.h>
#include <GLFW/glfw3.h>
#include "Window.h"
#include "Matrix.h"
#include "Vector.h"
#include "Shape.h"
#include "ShapeIndex.h"
#include "SolidShapeIndex.h"
#include "SolidShape.h"
```

uniform 変数に渡す光源の位置のデータ型を Vector 型に変更します。配列変数ではないので宣言に [ ] をつける必要はありませんが、初期化並びは同じです。

```
  // 光源データ
  static constexpr Vector Lpos = { 0.0f, 0.0f, 5.0f, 1.0f };
  static constexpr GLfloat Lamb[] = { 0.2f, 0.1f, 0.1f };
  static constexpr GLfloat Ldiff[] = { 1.0f, 0.5f, 0.5f };
  static constexpr GLfloat Lspec[] = { 1.0f, 0.5f, 0.5f };
```

最後に、この光源の位置データ Lpos にビュー変換の行列 view を乗じます。これは Vector.h で定義した operator*() 関数により実行されます。この戻り値の Vector 型は array の別名なので、array の data() メソッドを使って格納されているデータ本体へのポインタを取り出します。

```
    // uniform 変数に値を設定する
    glUniformMatrix4fv(projectionLoc, 1, GL_FALSE, projection.data());
    glUniformMatrix4fv(modelviewLoc, 1, GL_FALSE, modelview.data());
    glUniformMatrix3fv(normalMatrixLoc, 1, GL_FALSE, normalMatrix);
    glUniform4fv(LposLoc, 1, (view * Lpos).data());
```

```
    glUniform3fv(LambLoc, 1, Lamb);
    glUniform3fv(LdiffLoc, 1, Ldiff);
    glUniform3fv(LspecLoc, 1, Lspec);
```

■ サンプルプログラム step35

● 実行結果

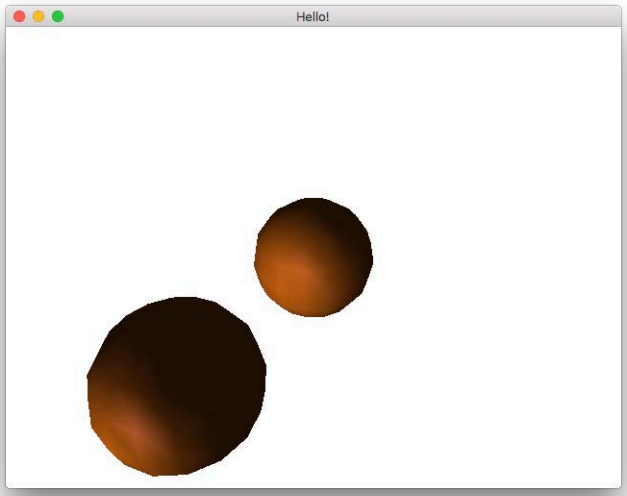


図 168 光源の位置を変更した結果

9.2.3 複数の光源

このシーンに光源を追加してみましょう。光源が複数ある時の反射光強度は、一つ一つの光源による反射光強度の合計になります (式 (103))。

● バーテックスシェーダ (point.vert) の変更点

まず、光源のデータを受け取る uniform 変数を配列にします。配列の要素数は定数値でなければいけません。

```
#version 150 core
uniform mat4 modelview;
uniform mat4 projection;
uniform mat3 normalMatrix;
const int Lcount = 2;
uniform vec4 Lpos[Lcount];
uniform vec3 Lamb[Lcount];
uniform vec3 Ldiff[Lcount];
uniform vec3 Lspec[Lcount];
const vec3 Kamb = vec3(0.6, 0.6, 0.2);
const vec3 Kdiff = vec3(0.6, 0.6, 0.2);
const vec3 Kspec = vec3(0.3, 0.3, 0.3);
const float Kshi = 30.0;
in vec4 position;
```

```
in vec3 normal;
out vec3 Idiff;
out vec3 Ispec;
```

Idiff と Ispec にそれぞれ拡散反射光強度と鏡面反射光強度の合計を求めます。そのために、まずこれらの変数を 0 にしておきます。なお、環境光の反射光強度は拡散反射光強度に合算しますので、個別に合計を求めることはしません。

```
void main()
{
  vec4 P = modelview * position;
  vec3 N = normalize(normalMatrix * normal);
  vec3 V = -normalize(P.xyz);
  Idiff = vec3(0.0);
  Ispec = vec3(0.0);
```

光源ごとに拡散反射光強度と鏡面反射光強度を求め、それぞれ Idiff と Ispec に加算します。光源のデータの uniform 変数は配列にしたので、[ ] を付けて個々の要素を指定します。

```
  for (int i = 0; i < Lcount; ++i)
  {
    vec3 L = normalize((Lpos[i] * P.w - P * Lpos[i].w).xyz);
    vec3 Iamb = Kamb * Lamb[i];
    Idiff += max(dot(N, L), 0.0) * Kdiff * Ldiff[i] + Iamb;
    vec3 H = normalize(L + V);
    Ispec += pow(max(dot(N, H), 0.0), Kshi) * Kspec * Lspec[i];
  }
  gl_Position = projection * P;
}
```

● **メインプログラム** (main.cpp) **の変更点**

光源のデータを追加します。x 方向 (8, 0, 0) から白い光 (0.9, 0.9, 0.9) を当ててみましょう。環境光の強度は (0.1, 0.1, 0.1) くらいにしてみます。光源の位置 Lpos は配列にします。

```
  // 光源データ
  static constexpr int Lcount(2);
  static constexpr Vector Lpos[] = { 0.0f, 0.0f, 5.0f, 1.0f, 8.0f, 0.0f, 0.0f, 1.0f };
  static constexpr GLfloat Lamb[] = { 0.2f, 0.1f, 0.1f, 0.1f, 0.1f, 0.1f };
  static constexpr GLfloat Ldiff[] = { 1.0f, 0.5f, 0.5f, 0.9f, 0.9f, 0.9f };
  static constexpr GLfloat Lspec[] = { 1.0f, 0.5f, 0.5f, 0.9f, 0.9f, 0.9f };
```

光源の位置にビュー変換行列 view をかけたものを、uniform 変数 Lpos に値を設定します。uniform 変数の Lpos も配列にしていますが、その個々の要素は、Lpos の場所 LposLoc からの相対位置にります。つまり、Lpos[0] の場所は LposLoc に、Lpos[1] の場所は LposLoc＋1 です。

配列変数に保持されている複数のデータを一度に uniform 変数の配列に格納する場合は、格納するデータの数を glUniform*fv() の第 2 引数に指定します。

```
    // uniform 変数に値を設定する
    glUniformMatrix4fv(projectionLoc, 1, GL_FALSE, projection.data());
    glUniformMatrix4fv(modelviewLoc, 1, GL_FALSE, modelview.data());
    glUniformMatrix3fv(normalMatrixLoc, 1, GL_FALSE, normalMatrix);
    for (int i = 0; i < Lcount; ++i)
      glUniform4fv(LposLoc + i, 1, (view * Lpos[i]).data());
    glUniform3fv(LambLoc, Lcount, Lamb);
    glUniform3fv(LdiffLoc, Lcount, Ldiff);
    glUniform3fv(LspecLoc, Lcount, Lspec);
```

■ サンプルプログラム step36

● 実行結果

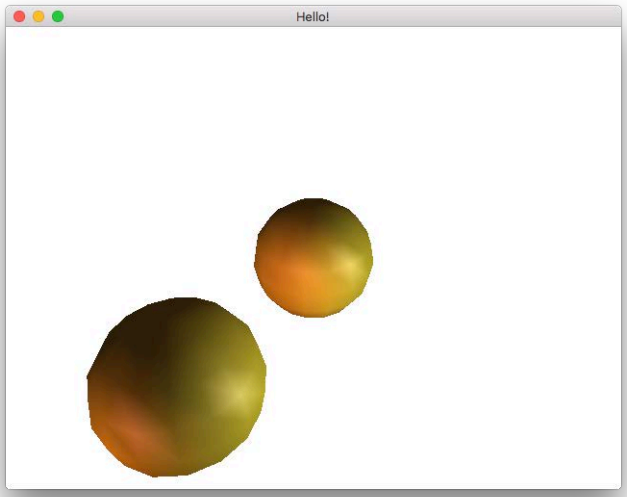


図 169 複数の光源を用いた陰影付け

バーテックスシェーダのソースプログラム point.vert では、光源の数 Lcount を定数にしています。これも uniform 変数にしてメインプログラムから渡せば、このシェーダプログラムで任意の数の光源を扱うことができるようになります。しかし、シェーダプログラムの配列変数の要素数は定数でなければならないので、要素数を多めに宣言しておくなどの工夫が必要になります。また、シェーダプログラム内の繰り返しは、繰り返しの回数が定数でないと、効率が非常に悪くなることがあります。

9.3 材質のデータ

材質のデータも光源のデータと同様に uniform 変数にして、メインプログラムからシェーダプログラムに渡すようにします。しかし、複数の図形に対して同じデータが用いられることが多い光源のデータに対して、材質のデータは描画する図形ごとに変更され、図形のデータ (頂点配列オブジェクト) とは別々に管理されるのが普通です。このとき、材質のデータを光源のデータと同じように uniform 変数に直接結びつけてしまうと、材質を切り替えるたびにシェーダプログラ

ムの uniform 変数の場所を指定して値を設定する必要があります。また、このためにシェーダプログラムの uniform 変数の場所を意識する必要があり、材質のデータを図形のデータと独立して管理することが難しくなります。

9.3.1 ユニフォームバッファオブジェクトと結合ポイント

光源や材質などのように uniform 変数に設定するデータを、頂点属性と同様にあらかじめ GPU 上のバッファオブジェクトに置いておくことができます。このバッファオブジェクトは**ユニフォームバッファオブジェクト** (Uniform Buffer Object, UBO) といいます。これと uniform 変数を、uniform 変数の場所に依存しない**結合ポイント** (Bounding Point, BP) を介して間接的に結びつけることにより、前述の問題を回避することができます (図 170)。

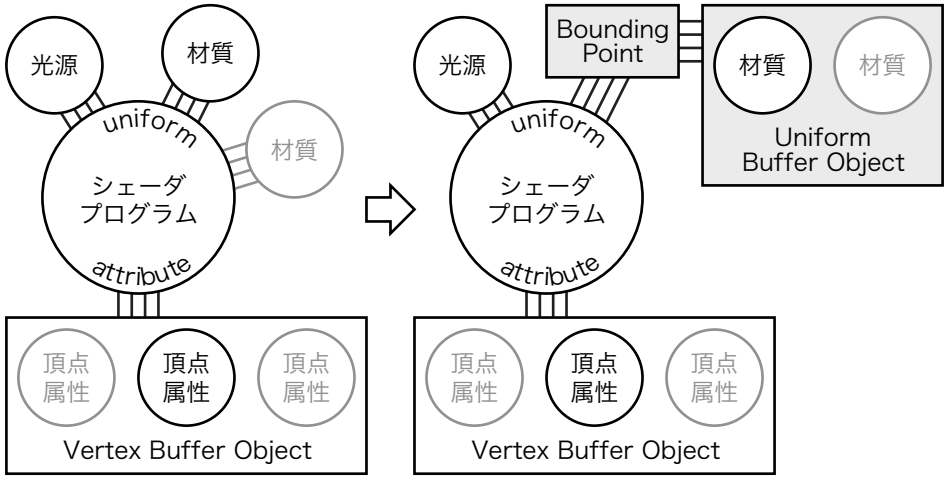


図 170 Uniform Buffer Object と Bounding Point

● **バーテックスシェーダ** (point.vert) **の変更点**

まず、定数で与えていたデータを、uniform 変数で受け取るようにします。ただし、ユニフォームバッファオブジェクトを用いるので、個々に uniform 変数にするのではなく、全部をまとめて uniform **ブロック**にします。layout (std140) は、この uniform ブロックのメモリレイアウトが、uniform ブロックの構造 (定義) と一致することを保証します。

```
#version 150 core
uniform mat4 modelview;
uniform mat4 projection;
uniform mat3 normalMatrix;
const int Lcount = 2;
uniform vec4 Lpos[Lcount];
uniform vec3 Lamb[Lcount];
uniform vec3 Ldiff[Lcount];
uniform vec3 Lspec[Lcount];
layout (std140) uniform Material
{
  vec3 Kamb;
```

```
  vec3 Kdiff;
  vec3 Kspec;
  float Kshi;
};
in vec4 position;
in vec3 normal;
out vec3 Idiff;
out vec3 Ispec;
```

● 材質の構造体 Material (Material.h)

材質のデータの構造体 Material を、Material.h というファイルに定義します。この構造体のメンバの構成は、バーテックスシェーダのソースプログラム point.vert の uniform ブロックと一致したものにします。その際、vec3 型の uniform 変数は 16 バイト境界に、float 型の uniform 変数は 4 バイト境界に置かれるため、それぞれのメンバに alignas(16) あるいは alignas(4) を指定します。uniform ブロックのメモリレイアウトについては後ほど解説します。(兎さんありがとう)

```
#pragma once
#include <array>
#include <GL/glew.h>

// 材質データ
struct Material
{
  // 環境光の反射係数
  alignas(16) std::array<GLfloat, 3> ambient;

  // 拡散反射係数
  alignas(16) std::array<GLfloat, 3> diffuse;

  // 鏡面反射係数
  alignas(16) std::array<GLfloat, 3> specular;

  // 輝き係数
  alignas(4) GLfloat shininess;
};
```

● ユニフォームバッファオブジェクトのクラス Uniform (Uniform.h)

材質とは別に、ユニフォームバッファオブジェクトのクラス Uniform を Uniform.h というファイルに定義します。

ところで、ユニフォームバッファオブジェクトなどのバッファオブジェクトをコンストラクタで作成してデストラクタで削除するようにすると、このインスタンスのコピーが複数存在し、それらが同じバッファオブジェクトを参照しているときに、そのどれか一つの削除によってバッファオブジェクトが削除されてしまい、残りのインスタンスからはアクセスできなくなってしまったり、残りのインスタンスが削除されたときにすでに削除されたバッファオブジェクトを削除しようとして OpenGL のエラーを引き起こしたりします。

そのため、頂点配列オブジェクトや頂点バッファオブジェクトを参照するクラスでは、コピー

コンストラクタや代入演算子を private にして、インスタンスのコピー自体を禁止していました(P. 79)。ユニフォームバッファオブジェクトでもその手法を採用することができますが、ここではスマートポインタ shared_ptr (図 56) を使って、すべてのインスタンスのコピーが削除されるまでバッファオブジェクトが削除されないようにするテクニックを紹介します。

また、Material 構造体の構造は uniform ブロックの構造を反映したものになるため、異なる陰影付けの方式を採用するなどして uniform ブロックの構造が変更されると、Material 構造体の構造も変更しなければなりません。そこで Uniform クラスはテンプレートクラスにして、ユニフォームブロックの構造の変更に対応できるようにします。

```
#pragma once
#include <memory>
#include <GL/glew.h>

// ユニフォームバッファオブジェクト
template <typename T>
class Uniform
{
```

template <typename T> によって、それに続くクラスは T という（仮の）データ型をもとに定義されます。この private メンバに実際にユニフォームバッファオブジェクトを参照する別の構造体 UniformBuffer を定義します。このコンストラクタでユニフォームバッファオブジェクトを作成し、デストラクタでそれを削除します。このときコンストラクタの引数の型を T 型のポインタ T* とし、その引数の内容を作成したユニフォームバッファオブジェクトに格納します。

```
  struct UniformBuffer
  {
    // ユニフォームバッファオブジェクト名
    GLuint ubo;

    // コンストラクタ
    //   data: uniform ブロックに格納するデータ
    UniformBuffer(const T *data)
    {
      // ユニフォームバッファオブジェクトを作成する
      glGenBuffers(1, &ubo);
      glBindBuffer(GL_UNIFORM_BUFFER, ubo);
      glBufferData(GL_UNIFORM_BUFFER,
        sizeof (T), data, GL_STATIC_DRAW);
    }

    // デストラクタ
    ~UniformBuffer()
    {
      // ユニフォームバッファオブジェクトを削除する
      glDeleteBuffers(1, &ubo);
    }
  };
```

Uniform クラスの private メンバには、他に UniformBuffer 構造体のポインタをスマートポイ

ンタ shared_ptr として保持します。

```
  // バッファオブジェクト
  const std::shared_ptr<const UniformBuffer> buffer;
```

そして Uniform クラスのコンストラクタでは、引数を用いて UniformBuffer 構造体のインスタンスを一つ生成し、上記のメンバ buffer を初期化します。

Uniform クラスのインスタンスをコピーすると、この buffer の内容もコピーされ、コピーされたインスタンスは同一の UniformBuffer 構造体のインスタンスを参照することになります。仮に buffer が通常のポインタ変数であれば、buffer に格納されているポインタを明示的に delete しない限り、そのポインタが指している実体は削除されません。したがって、Uniform クラスのインスタンスが一つ削除されても、それが参照している UniformBuffer 構造体のインスタンスは削除されないため、ユニフォームバッファオブジェクトも削除されません。

しかし、それではすべてのインスタンスが削除されても UniformBuffer 構造体のインスタンスが削除されないことになってしまいます。そこで buffer をスマートポインタ shared_ptr にすることにより、最後のインスタンスが削除されるときに UniformBuffer 構造体のインスタンスも削除されるようになります。

また、Uniform クラスの変数を一つ宣言するごとに一つの UniformBuffer 構造体のインスタンスが生成されますが、その変数に別の Uniform クラスのインスタンスを代入した場合、もともと参照していた (変数宣言時に生成された) UniformBuffer 構造体のインスタンスは削除されます。この結果、変数への代入により使われないバッファオブジェクトが残ることも防止できます。

なお、この引数のデフォルト引数に NULL を設定しておけば、デフォルトコンストラクタによりデータを設定しないユニフォームバッファオブジェクトが作成されます。コンストラクタやデストラクタの内部では、この他に特に処理することはありません。

```
public:

  // コンストラクタ
  //   data: uniform ブロックに格納するデータ
  Uniform(const T *data = NULL)
    : buffer(new UniformBuffer(data))
  {
  }

  // デストラクタ
  virtual ~Uniform()
  {
  }
```

上記のようにコンストラクタの引数に NULL を指定したときなどのために、あとからユニフォームバッファオブジェクトにデータを転送するメソッドも作成しておきます。

```
  // ユニフォームバッファオブジェクトにデータを格納する
  //   data: uniform ブロックに格納するデータ
  void set(const T *data) const
  {
    glBindBuffer(GL_UNIFORM_BUFFER, buffer->ubo);
    glBufferSubData(GL_UNIFORM_BUFFER, 0,
      sizeof (T), data);
  }
```

void glBufferSubData(GLenum target, GLintptr offset, GLsizeiptr size, const GLvoid *data)

バッファオブジェクトの指定した場所にデータを転送します。

target

データを転送するバッファオブジェクトの結合対象。

offset

データを転送するバッファオブジェクトの転送先の位置。

size

GPU 側に転送するデータ (data) のサイズ。

data

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) のサイズは引数 size バイト以上必要。

既に存在するバッファオブジェクトへのデータの転送には、上記の glBufferSubData() 以外に、glMapBuffer() を用いる方法があります。この説明は、本書では割愛します。

ユニフォームバッファオブジェクトに転送された材質のデータを、シェーダプログラムの uniform ブロックのメンバの変数から参照するために、ユニフォームバッファオブジェクトを結合ポイントに結合します。

```
  // このユニフォームバッファオブジェクトを使用する
  //   bp: 結合ポイント
  void select(GLuint bp) const
  {
    // 材質に設定するユニフォームバッファオブジェクトを指定する
    glBindBufferBase(GL_UNIFORM_BUFFER, bp,
      buffer->ubo);
  }
};
```

void glBindBufferBase(GLenum target, GLuint index, GLuint buffer)

引数 buffer に指定されたバッファオブジェクトを引数 target に指定されたターゲットの配列の index 番の結合ポイントに結合します。

target

結合ポイントに結合するバッファオブジェクトの結合対象。

index

バッファオブジェクトを結合する結合ポイントの番号。結合ポイントの番号は 0 または正の整数値で最大値は少なくとも 36 の整数。

buffer

結合ポイントに結合するバッファオブジェクト。

● メインプログラム (main.cpp) の変更点

このヘッダファイル Material.h と Uniform.h を main.cpp の冒頭で #include します。

```
#include <cmath>
#include <cstdlib>
#include <iostream>
#include <fstream>
#include <vector>
#include <memory>
#include <GL/glew.h>
#include <GLFW/glfw3.h>
#include "Window.h"
#include "Matrix.h"
#include "Vector.h"
#include "Shape.h"
#include "ShapeIndex.h"
#include "SolidShapeIndex.h"
#include "SolidShape.h"
#include "Uniform.h"
#include "Material.h"
```

glGetUniformBlockIndex() を使って、シェーダプログラムから uniform ブロックの場所を取得します。これを glUniformBlockBinding() により結合ポイント (ここでは 0 番を使用) に結びつけておきます。この結合ポイントの番号を介することにより、シェーダプログラムの uniform 変数の場所を意識せずにシェーダプログラムの uniform ブロックの変数にデータを設定できます。

```
  // uniform 変数の場所を取得する
  const GLint modelviewLoc(glGetUniformLocation(program, "modelview"));
  const GLint projectionLoc(glGetUniformLocation(program, "projection"));
  const GLint normalMatrixLoc(glGetUniformLocation(program, "normalMatrix"));
  const GLint LposLoc(glGetUniformLocation(program, "Lpos"));
  const GLint LambLoc(glGetUniformLocation(program, "Lamb"));
  const GLint LdiffLoc(glGetUniformLocation(program, "Ldiff"));
  const GLint LspecLoc(glGetUniformLocation(program, "Lspec"));

  // uniform block の場所を取得する
  const GLint materialLoc(glGetUniformBlockIndex(program, "Material"));

  // uniform block の場所を 0 番の結合ポイントに結びつける
  glUniformBlockBinding(program, materialLoc, 0);

  // 球の分割数
  const int slices(16), stacks(8);
```

void glUniformBlockBinding(GLuint program, GLuint uniformBlockIndex,
GLuint uniformBlockBinding)

結合ポイントにシェーダプログラムの uniform ブロックの場所を結合します。

program

結合ポイントに結合する uniform ブロックを含む、シェーダプログラムのプログラムオブジェクト名 (番号)。

uniformBlockIndex

program で指定したプログラムオブジェクトから取り出された uniform ブロックの場所。

uniformBlockBinding

このシェーダプログラムの uniform ブロックに結合する結合ポイントの番号。結合ポイントの番号は 0 または正の整数値で最大値は少なくとも 36 の整数。

次に、Material 構造体の変数を作成し、その初期化並びに色データを設定します。また、ユニフォームバッファオブジェクトのテンプレートクラス Uniform のテンプレート引数に Material 構造体を指定して、そのインスタンスを生成します。

このテンプレート引数に Material 構造体を指定した Uniform クラスのコンストラクタは、引数として Material 構造体のポインタを受け取ります。したがって、そのインスタンスの配列の material は、初期化並びに与えられた Material 構造体のポインタ &color[0] と &color[1] のそれぞれを引数として、要素ごとにコンストラクタを呼び出して初期化されます。なお、この初期化はコンパイル時には実行できないので、配列 material は constexpr では修飾できません。

```
// 光源データ
static constexpr int Lcount(2);
static constexpr Vector Lpos[] = { 0.0f, 0.0f, 5.0f, 1.0f, 8.0f, 0.0f, 0.0f, 1.0f };
static constexpr GLfloat Lamb[] = { 0.2f, 0.1f, 0.1f, 0.1f, 0.1f, 0.1f };
static constexpr GLfloat Ldiff[] = { 1.0f, 0.5f, 0.5f, 0.9f, 0.9f, 0.9f };
static constexpr GLfloat Lspec[] = { 1.0f, 0.5f, 0.5f, 0.9f, 0.9f, 0.9f };

// 色データ
static constexpr Material color[] =
{
  //      Kamb               Kdiff              Kspec        Kshi
  { 0.6f, 0.6f, 0.2f,  0.6f, 0.6f, 0.2f,  0.3f, 0.3f, 0.3f,  30.0f },
  { 0.1f, 0.1f, 0.5f,  0.1f, 0.1f, 0.5f,  0.4f, 0.4f, 0.4f,  60.0f }
};

const Uniform<Material> material[] = { &color[0], &color[1] };

// タイマーを 0 にセット
glfwSetTime(0.0);
```

最後に、図形を描画する直前で select メソッドを使って、このユニフォームバッファオブジェクトを uniform ブロックの結合ポイント (0 番) に結合します。

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

    // 図形を描画する
    material[0].select(0);
    shape->draw();

     《省略》

    // 二つ目の図形を描画する
    material[1].select(0);
    shape->draw();

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

■ サンプルプログラム step37

● 実行結果

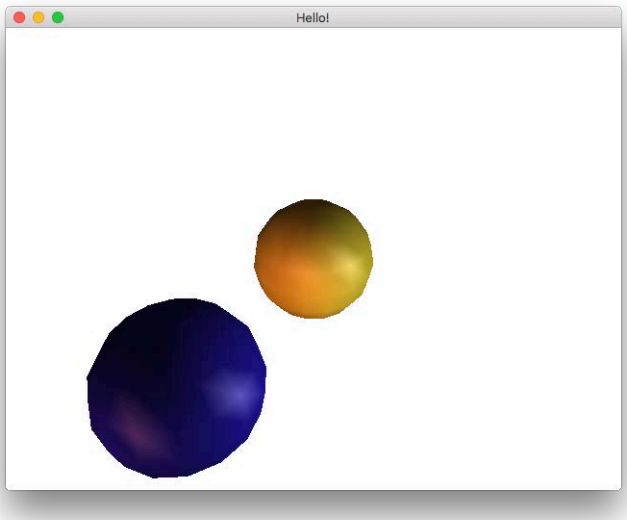


図 171 図形ごとに色を変えて描画

9.3.2 uniform ブロックのメモリレイアウトについて

uniform ブロック内の変数には、ユニフォームバッファオブジェクトの内容がコピーされるため、データが個々の uniform 変数に期待通り格納されるためには、uniform ブロックの構造に合わせてユニフォームバッファオブジェクト上にデータを配置する必要があります。

uniform ブロックの layout 修飾子は、このメモリレイアウトを指定するものです。これには次のものがあります。

packed

メモリレイアウトは実装に依存します。シェーダプログラムの最適化によって、使われていない uniform ブロックの要素が削除される場合もあります。これがデフォルトです。

shared

packed と同様メモリレイアウトは実装に依存します。異なるシェーダプログラム間で同じメモリレイアウトとなることが保証されます。

std140

メモリレイアウトは uniform ブロックの定義に従います。このため、C / C++ の構造体やクラスのメンバに対応させることが容易になります。ただし、配列の要素の間隔は常に 16 バイトになります。Material 構造体は、このレイアウトを採用しています。

また、各データ型のアライメントは、float が 4、vec2 が 8、vec3 が 16、vec4 が 16 です。したがって、このそれぞれのデータの開始位置は、バッファオブジェクトの開始位置を 0 として、この数で割り切れる場所である必要があります。例えば、Material の uniform ブロックのメモリレイアウトは、図 172 のようになります。

GLSL

```
layout (std140) uniform Material
{
  vec3 Kamb;
  vec3 Kdiff;
  vec3 Kspec;
  float Kshi;
};
```

C++

```
struct Material
{
  alignas(16) std::array<GLfloat, 3> ambient;
  alignas(16) std::array<GLfloat, 3> diffuse;
  alignas(16) std::array<GLfloat, 3> specular;
  alignas(4) GLfloat shininess;
};
```

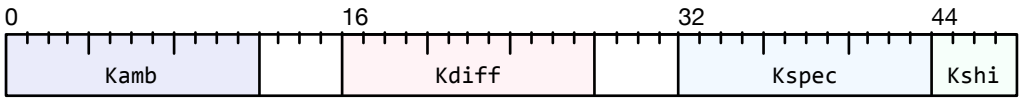


図 172 Material の uniform ブロックのメモリレイアウト

vec3 のアライメントは 16 ですから、Kamb、Kdiff、Kspec の先頭の位置はいずれも 16 の整数倍の場所に配置され、それらの間に 4 の隙間が空いてしまいます。これに対して float のアライメントは 4 ですから、Kshi は Kspec の直後に配置されます。

したがって、Kshi を Kamb と Kdiff の間に移せば、隙間が一つ埋まります。しかし、そのような最適化はあまり効果がありません。後述しますが、実はユニフォームブロック自体が GPU のメモリ上で 64 や 256 など 2 べき乗のより大きなアライメントで配置されます。このユニフォームブロックは隙間を含め全体で 48 の大きさを持つので、仮に blocksize が 64 であれば、GPU 上では後ろに 16 の隙間が空いてしまうことになります。

なお、OpenGL Wiki https://www.khronos.org/opengl/wiki/Interface_Block_(GLSL) によれば、std140 のレイアウトにおいて、実装によっては vec3 のデータの配置が間違っていることがあるようです。そのため、構造体/配列を手作業で詰め込んで、vec3 を使用しないことが推奨されています。

9.3.3 単一のユニフォームバッファオブジェクトを使う

一つの材質ごとに一つのユニフォームバッファオブジェクトを割り当てる代わりに、一つのユニフォームバッファオブジェクト上に複数の材質のデータを格納することもできます。

● Uniform クラス (Uniform.h) の変更点

UniformBuffer 構造体に GLsizeiptr 型のメンバ blocksize を追加し、 それに T 型の uniform ブロックを格納可能なバッファオブジェクト上の最小サイズを格納します。blocksize は uniform ブロックのアライメント GL_UNIFORM_BUFFER_OFFSET_ALIGNMENT の整数倍でなければなりません。また、コンストラクタの引数に確保する uniform ブロックの数 count を追加します。

GL_UNIFORM_BUFFER_OFFSET_ALIGNMENT はシステム固有の大域パラメータで、このようなパラメータは GetIntegerv() などの glGet 関数群を用いて取得することができます。これは一度だけ取得すれば十分なので blocksize は static メンバにすることも可能ですが、取得は OpenGL の初期化以降に行う必要があります。glGet 関数群の説明は割愛します。

```
// ユニフォームバッファオブジェクト
template <typename T>
class Uniform
{
  struct UniformBuffer
  {
    // ユニフォームバッファオブジェクト名
    GLuint ubo;

    // ユニフォームブロックのサイズ
    GLsizeiptr blocksize;

    // コンストラクタ
    //   data: uniform ブロックに格納するデータ
    //   count: 確保する uniform ブロックの数
    UniformBuffer(const T *data, unsigned int count)
    {
      // ユニフォームブロックのサイズを求める
      GLint alignment;
      glGetIntegerv(GL_UNIFORM_BUFFER_OFFSET_ALIGNMENT, & alignment);
      blocksize = (((sizeof (T) - 1) / alignment) + 1) * alignment;
```

このユニフォームバッファオブジェクトには、データを blocksize の間隔をあけて格納しなければならないので (図 173)、glBufferData() の第 2 引数 size には確保するユニフォームバッファオブジェクトのサイズとして blocksize の uniform ブロックの数 count 個分を指定します。また、第 3 引数 data に指定する転送するデータには、ここでは NULL を指定して、データを転送しないでおきます。データの転送は、この後に glBufferSubData() を用いて、一つずつ転送します。その際には、データの転送先を blocksize ずつずらします。

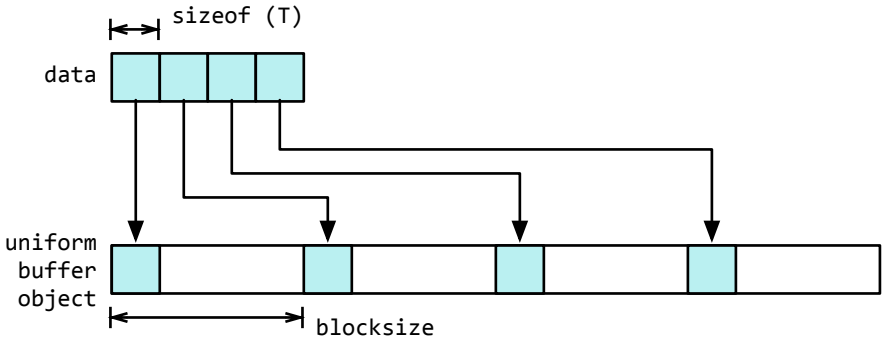


図 173 ユニフォームバッファオブジェクトへの複数のデータの格納

```
    // ユニフォームバッファオブジェクトを作成する
    glGenBuffers(1, &ubo);
    glBindBuffer(GL_UNIFORM_BUFFER, ubo);
    glBufferData(GL_UNIFORM_BUFFER,
      count * blocksize, NULL, GL_STATIC_DRAW);
    for (unsigned int i = 0; i < count; ++i)
    {
      glBufferSubData(GL_UNIFORM_BUFFER, i * blocksize,
        sizeof (T), data + i);
    }
  }

  《省略》
};
```

Uniform クラスのコンストラクタの引数に、確保する uniform ブロックの数 count を追加し、それをそのまま UniformBuffer クラスのコンストラクタに使用します。

```
  《省略》

public:

  // コンストラクタ
  //   data: uniform ブロックに格納するデータ
  //   count: 確保する uniform ブロックの数
  Uniform(const T *data = NULL, unsigned int count = 1)
    : buffer(new UniformBuffer(data, count))
  {
  }

  // デストラクタ
  virtual ~Uniform()
  {
  }
```

このユニフォームバッファオブジェクトにデータを格納する位置は、blocksize の整数倍になります。これも UniformBuffer クラスのコンストラクタと同様に、glBufferSubData() を使って、データを一つずつ転送します。

```
  // ユニフォームバッファオブジェクトにデータを格納する
  //   data: uniform ブロックに格納するデータ
```

```
  //   start: データを格納する uniform ブロックの先頭位置
  //   count: データを格納する uniform ブロックの数
  void set(const T *data, unsigned int start = 0, unsigned int count = 1) const
  {
    glBindBuffer(GL_UNIFORM_BUFFER, buffer->ubo);
    for (unsigned int i = 0; i < count; ++i)
    {
      glBufferSubData(GL_UNIFORM_BUFFER, (start + i) * buffer->blocksize,
        sizeof (T), data + i);
    }
  }
```

複数の uniform ブロックが格納されたユニフォームバッファオブジェクトの、特定の uniform ブロックを結合ポイントに結合するには、glBindBufferBase() の代わりに glBindBufferRange() を使用します。結合する uniform ブロックのユニフォームバッファオブジェクト上の位置は、これも blocksize の整数倍になります。

なお、glBindBufferBase() は glBindBufferRange() の offset を 0、size をバッファオブジェクト全体のサイズとした場合と等価です。

```
  // このユニフォームバッファオブジェクトを使用する
  //   bp: 結合ポイント
  //   i: 結合する uniform ブロックの位置
  void select(GLuint bp, unsigned int i = 0) const
  {
    // 結合ポイントにユニフォームバッファオブジェクトを結合する
    glBindBufferRange(GL_UNIFORM_BUFFER, bp,
      buffer->ubo, i * buffer->blocksize, sizeof (T));
  }
};
```

void glBindBufferRange(GLenum target, GLuint index, GLuint buffer, GLintptr offset, GLsizeiptr size))

引数 buffer に指定されたバッファオブジェクトを引数 target に指定されたターゲットの配列の index 番の結合ポイントに結合します。

target

結合ポイントに結合するバッファオブジェクトの結合対象。

index

バッファオブジェクトを結合する結合ポイントの番号。結合ポイントの番号は 0 または正の整数値で最大値は少なくとも 36 の整数。

buffer

結合ポイントに結合するバッファオブジェクト。

offset

結合ポイントに結合するバッファオブジェクト buffer 中の開始位置。これはシステム固有の大域パラメータである GL_UNIFORM_BUFFER_OFFSET_ALIGNMENT の整数倍である

必要がある。

size

結合ポイントに結合するバッファオブジェクト上のサイズ。

● メインプログラム (main.cpp) の変更点

このままでもプログラムは実行できますが、それだと材質ごとに独立したユニフォームバッファオブジェクトを持つことに変わりはありません。そこで、これを次のように書き換えます。

テンプレート引数に Material 構造体を指定した Uniform クラスのインスタンスは、配列ではなく単一の引数とします (配列の場合は要素ごとに独立したユニフォームバッファオブジェクトを保持していました)。そのコンストラクタの引数に材質の色データとその数を指定します。

```
// 色データ
static constexpr Material color[] =
{
  //      Kamb               Kdiff              Kspec        Kshi
  { 0.6f, 0.6f, 0.2f,  0.6f, 0.6f, 0.2f,  0.3f, 0.3f, 0.3f,  30.0f },
  { 0.1f, 0.1f, 0.5f,  0.1f, 0.1f, 0.5f,  0.4f, 0.4f, 0.4f,  60.0f }
};

const Uniform<Material> material(color, 2);

// タイマーを 0 にセット
glfwSetTime(0.0);
```

材質の指定は配列の添字ではなく、select() メソッドの引数で行います。

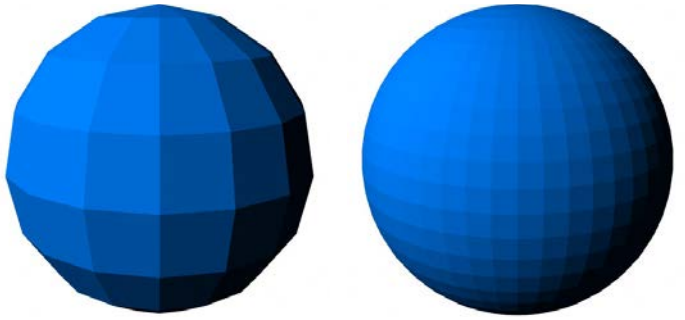
```
  // 図形を描画する
  material.select(0, 0);
  shape->draw();

  《省略》

  // 二つ目の図形を描画する
  material.select(0, 1);
  shape->draw();
```

■ サンプルプログラム step38

この実行結果は step37 と変わりません。

9.4 スムーズシェーディング

OpenGL には曲面を表示する機能はありません。OpenGL に限らず、ほとんどのグラフィックスハードウェアは、直接曲面を生成する機能を持っていません。そのため、なめらかな曲面を表示するには多数の面 (ポリゴン) を持つ多面体で近似しなければなりません。しかし、境界が知覚できないほどポリゴンの多い多面体では、データ量が膨大になってしまいます。



面の数が少ない時　　面の数が多い時

ポリゴンの境界が目立ってしまう理由に、明度差のある境界において明るい方はより明るく、暗い方はより暗く見えるという人間の視覚の特性があります。これは**マッハ効果** (図 175) と呼ばれ、このために境界がより目立ってしまいます。

図 174 曲面の多面体による近似

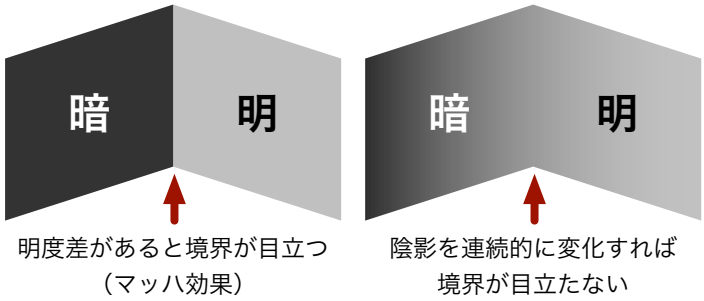


図 175 マッハ効果

9.4.1 頂点色の補間 (グーローの方法)

境界付近の明度差を下げれば、この効果を抑制できます。そこで、境界を横切る方向に明度を連続的に変化させます。OpenGL では三角形の頂点に設定されている頂点属性の、三角形の内部における補間値を得ることができるため、頂点における色を補間するようにすれば、三角形の内部の色を連続的に変化させることができます (図 176)。

そこで、隣接する三角形で頂点属性を共有するなどして頂点の色を同じにすれば、ポリゴンの境界を目立たなくすることができます。このような手法を**スムーズシェーディング**といい、特に頂点における色を補間する方法を**グーロー (Gouraud) のスムーズシェーディング**と呼びます。

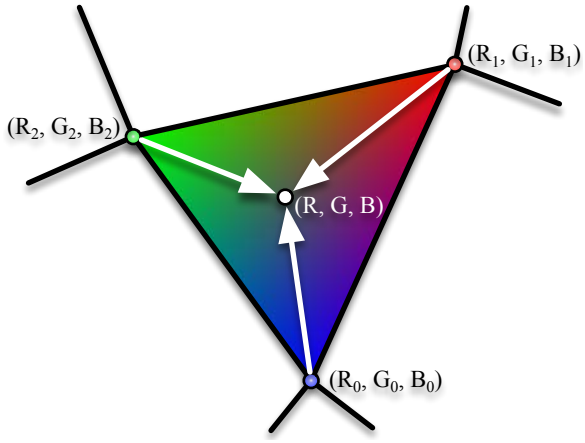


図 176 頂点色の補間

頂点の色は、頂点において陰影付け処理を行って決定します。球の場合、球の表面の一点における法線ベクトルは、球の中心からその点に向かうベクトルになります。しかし、任意の多面体では、通常、頂点の法線ベクトルは明らかではありません。

このため曲面を近似しようとする多面体の頂点の法線は、その頂点を共有しているポリゴンの法線ベクトルを合成して求めるのが一般的です (図 177)。

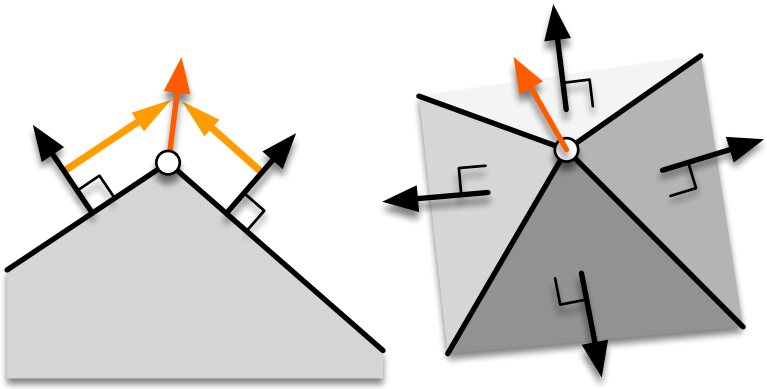


図 177 頂点の法線ベクトル

ただ、この方法では光源や視点の位置によって、ハイライト (鏡面反射光) が消失することがあります (図 178)。また、ポリゴンの数が少ないときのハイライトの形状の品質もあまりよくありません。この現象はサンプルプログラム step 33 以降でも観測できます。

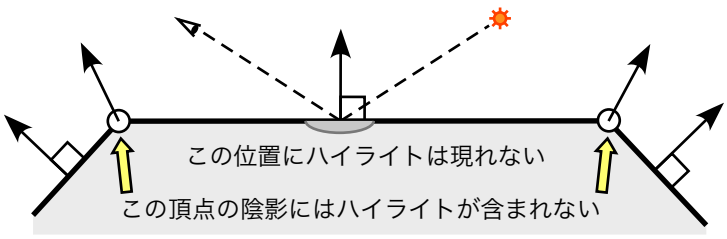


図 178 ハイライトの消失

9.4.2 法線ベクトルの補間 (フォンの方法)

グーローのスムーズシェーディングで行われる頂点の色の補間は単純なハードウェアで実現できるため、性能の面ではスムーズシェーディングを行わない場合 (フラットシェーディング) と遜色はありません。しかし、得られる画像に前述のような品質の問題があります。

これは、頂点の色の代わりに法線ベクトルを補間し (図 179)、画素単位に陰影付けを行えば解消することができます (図 180)。この**方法はフォン (Phong) のスムーズシェーディング**と呼ばれます。この方法は画素ごとに多数の実数計算を行うので、頂点の色を補間するより処理に時間がかかります。しかし、最近の GPU の実数計算の性能はスーパーコンピュータにも活用されるほど非常に高いので、気にする必要はないかもしれません。

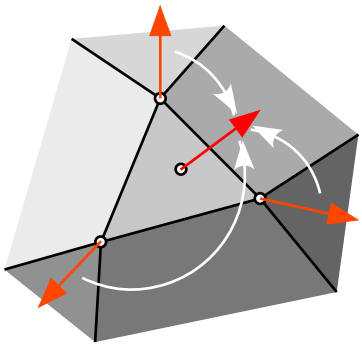


図 179 法線ベクトルの補間

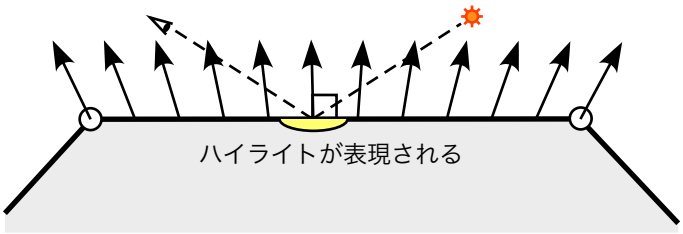


図 180 ハイライトの再現

実は、この方法も完全ではなく、例えば図 181 の左のような図形では、陰影に凹凸の形状が反映されません。これは同図右のように、形状を工夫することによって回避できます。

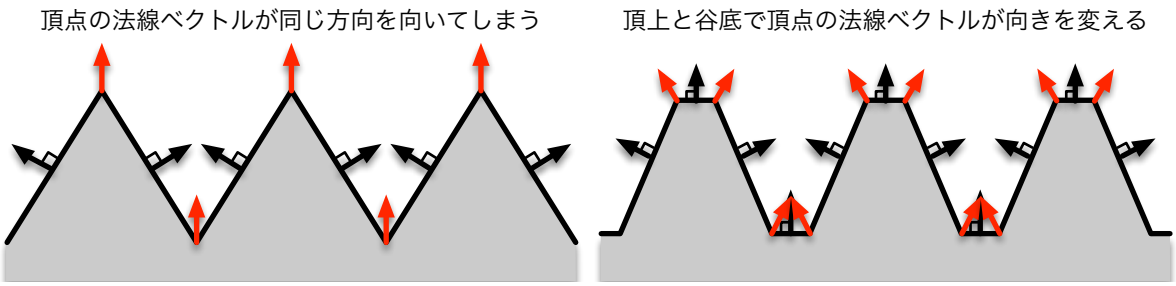


図 181 スムーズシェーディングできない形状とその対策

● バーテックスシェーダ (point.vert) の変更点

鏡面反射光をフラグメントシェーダで計算するようにしてみます。そのために、鏡面反射光強度 Ispec をバーテックスシェーダから出力する代わりに、頂点の位置 P と頂点の法線ベクトル N を出力するようにします。P と N を out 変数にして、Ispec と Ispec を求めている式を削除します。Ispec を求めるのに使っている Lspec も使わないので削除します。

```
#version 150 core
uniform mat4 modelview;
uniform mat4 projection;
uniform mat3 normalMatrix;
const int Lcount = 2;
uniform vec4 Lpos[Lcount];
uniform vec3 Lamb[Lcount];
uniform vec3 Ldiff[Lcount];
layout (std140) uniform Material
{
  vec3 Kamb;
  vec3 Kdiff;
  vec3 Kspec;
  float Kshi;
};
in vec4 position;
in vec3 normal;
out vec3 Idiff;
out vec4 P;
out vec3 N;
void main()
{
  P = modelview * position;
  N = normalize(normalMatrix * normal);
  Idiff = vec3(0.0);
  for (int i = 0; i < Lcount; ++i)
  {
    vec3 L = normalize((Lpos[i] * P.w - P * Lpos[i].w).xyz);
    vec3 Iamb = Kamb * Lamb[i];
    Idiff += max(dot(N, L), 0.0) * Kdiff * Ldiff[i] + Iamb;
  }
  gl_Position = projection * P;
}
```

● フラグメントシェーダ (point.frag) の変更点

フラグメントシェーダで鏡面反射光を計算するので、鏡面反射光強度 Ispec の計算に使う uniform 変数 Lpos や Lspec をバーテックスシェーダと同様に宣言します。また材質の uniform ブロックもここにコピーします。

```
#version 150 core
const int Lcount = 2;
uniform vec4 Lpos[Lcount];
uniform vec3 Lspec[Lcount];
layout (std140) uniform Material
```

```
{
  vec3 Kamb;
  vec3 Kdiff;
  vec3 Kspec;
  float Kshi;
};
```

鏡面反射光強度 Ispec の代わりに頂点の位置 P と頂点の法線ベクトル N を in 変数で受け取ります。これらを用いて視点方向のベクトル V や光源方向のベクトル L、そして中間ベクトル H を求め、鏡面反射光強度 Ispec を求めます。ちなみに、これらはバーテックスシェーダのものを移すだけです。

```
in vec3 Idiff;
in vec4 P;
in vec3 N;
out vec4 fragment;
void main()
{
  vec3 V = -normalize(P.xyz);
  vec3 Ispec = vec3(0.0);
  for (int i = 0; i < Lcount; ++i)
  {
    vec3 L = normalize((Lpos[i] * P.w - P * Lpos[i].w).xyz);
    vec3 H = normalize(L + V);
    Ispec += pow(max(dot(normalize(N), H), 0.0), Kshi) * Kspec * Lspec[i];
  }
  fragment = vec4(Idiff + Ispec, 1.0);
}
```

■ サンプルプログラム step39

● 実行結果

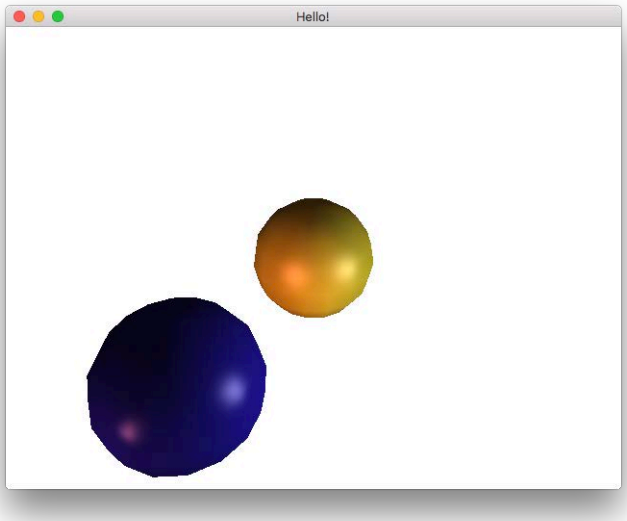


図 182 鏡面反射光を画素単位に計算した結果

● バーテックスシェーダ (point.vert) の変更点

サンプルプログラム step39 の実行結果より、鏡面反射光強度の計算だけをフラグメントシェーダに移せば、ハイライトの品質は改善されることがわかります。これは実行回数の多いフラグメントシェーダでの計算量をできるだけ増やさないという点では有効ですが、拡散反射光強度の計算もフラグメントシェーダに移してしまった方がシェーダプログラム自体は簡単になります。

この場合、バーテックスシェーダでは頂点の位置 P と頂点の法線ベクトル N を out 変数で出力するだけになります。拡散反射光強度 Idiff の出力や、陰影づけの計算を行わないので光源の uniform 変数や材質の uniform ブロックも不要になります。

```
#version 150 core
uniform mat4 modelview;
uniform mat4 projection;
uniform mat3 normalMatrix;
in vec4 position;
in vec3 normal;
out vec4 P;
out vec3 N;
void main()
{
  P = modelview * position;
  N = normalize(normalMatrix * normal);
  gl_Position = projection * P;
}
```

● フラグメントシェーダ (point.frag) の変更点

一方フラグメントシェーダには、拡散反射光強度 Idiff の計算に必要な uniform 変数を追加し、Idiff を算出します。ちなみに、これらもバーテックスシェーダのものを移すだけです。

```
#version 150 core
const int Lcount = 2;
uniform vec4 Lpos[Lcount];
uniform vec3 Lamb[Lcount];
uniform vec3 Ldiff[Lcount];
uniform vec3 Lspec[Lcount];
layout (std140) uniform Material
{
  vec3 Kamb;
  vec3 Kdiff;
  vec3 Kspec;
  float Kshi;
};
in vec4 P;
in vec3 N;
out vec4 fragment;
void main()
{
  vec3 V = -normalize(P.xyz);
  vec3 Idiff = vec3(0.0);
  vec3 Ispec = vec3(0.0);
```

```
  for (int i = 0; i < Lcount; ++i)
  {
    vec3 L = normalize((Lpos[i] * P.w - P * Lpos[i].w).xyz);
    vec3 Iamb = Kamb * Lamb[i];
    Idiff += max(dot(N, L), 0.0) * Kdiff * Ldiff[i] + Iamb;
    vec3 H = normalize(L + V);
    Ispec += pow(max(dot(normalize(N), H), 0.0), Kshi) * Kspec * Lspec[i];
  }
  fragment = vec4(Idiff + Ispec, 1.0);
}
```

■ サンプルプログラム step40

● 実行結果

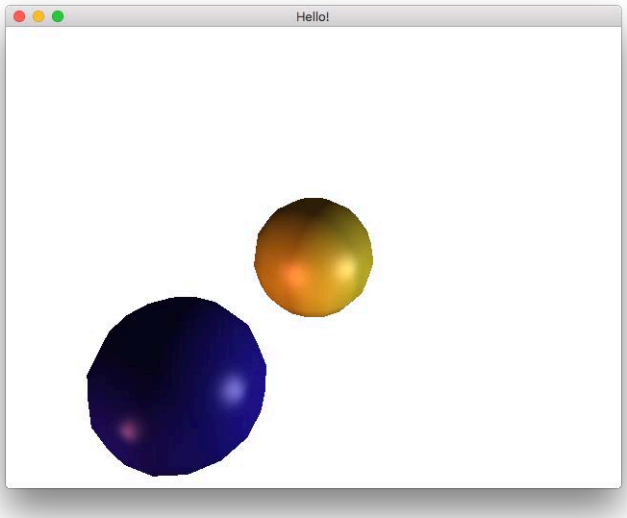


図 183 鏡面反射光と拡散反射光を画素単位に計算した結果

（つづく）