[OpenGL 3D 2018 第01回]

OpenGLの鼓動

# OpenGLについて

OpenGLは、クロノス・グループ(Khronos Group)というアメリカの非営利団体が策定しているコンピューターグラフィックスAPIです。  
DirectXは基本的にWindows専用ですが、OpenGLはWindows以外にもmacOS、UNIX、Linux、FreeBSDなどさまざまな環境で使用できます(クロスプラットフォームといいます)。  
現在、OpenGLの最新バージョンは4.6(2018.2.11現在)です。しかし、すべての環境で最新バージョンが動作するとは限りません。そこで、ちょっと古いノートパソコンでも動作すると思われる、バージョン4.1を使用して解説していきます。

OpenGL はバージョン3.1においてそれまで使われていた固定機能のグラフィックスAPIを全て捨て去り、シェーダによるグラフィックスAPIに一本化されました。本講義で使用するバージョン4.1もシェーダのみが使用可能です。

## 表記について

この講義文書では、みなさんが実際に書き写す必要のあるコードは次のように薄い灰色の背景で示します。

// これは書き写す.

以前のサンプルコードに追加したり削除した部分がある場合、行頭にプラスとマイナスの記号を付けます。

**-**// 先頭にマイナス記号が付いているのは、削除する行です.  
**+**// プラス記号が付いているのは追加する行です.  
 // どちらも付いていない場合は変更しない行です.

また、書き写す必要のないサンプルコードや特記事項は、以下のように黒枠で囲んで示すことにします。

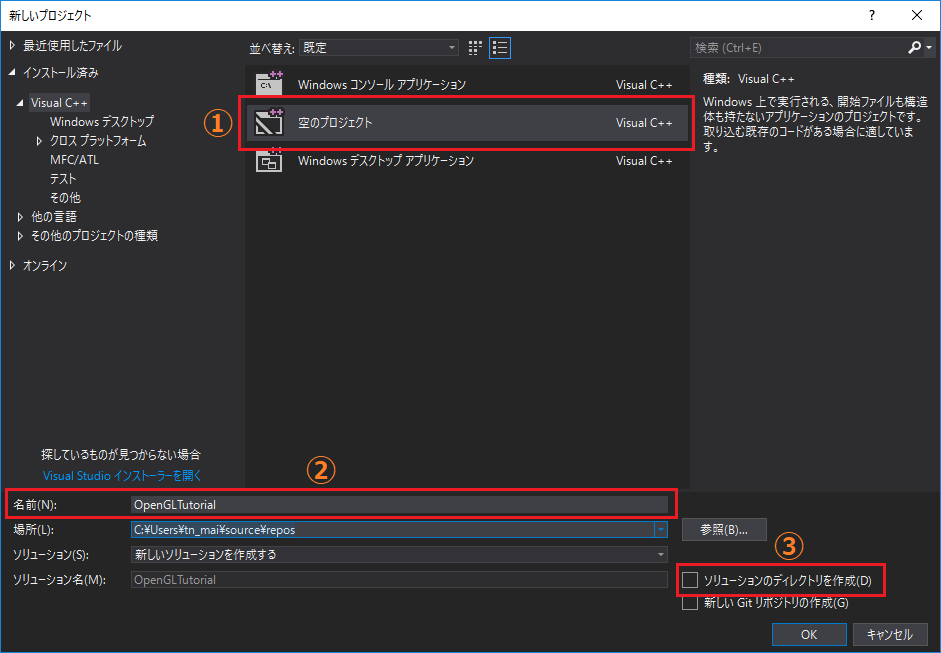
**[補足]** これは書き写さない。

# プロジェクトの作成

Visual Studio 2017を起動してください。そして、メニューから「ファイル→新規作成→プロジェクト」を選択しましょう。すると「新しいプロジェクト」ウィンドウが開きます。ウィンドウ左側パネルの「Visual C++」をクリックし、中央のテンプレートリストから「空のプロジェクト」を選択します。「空のプロジェクト」では、Visual Studioは必要最小限のファイルとフォルダだけを作成します。ソースファイルやヘッダファイルは自分でプロジェクトに追加することになります。

次に、これから作成するプロジェクトに名前をつけます。プロジェクト名は、プロジェクトを格納するフォルダや、作成される実行ファイルの名前にも使われます。講義資料ではOpenGLTutorialとしていますが、好きな名前を付けてもらって構いません。

それから、右下に「ソリューションのディレクトリを作成」というチェックボックスがあると思います。もしチェックが付いているなら、クリックしてチェックを外しておいてください。名前を決め、チェックが外れていることを確認したら「OK」ボタンをクリックしてプロジェクトを作成しましょう。



**[補足]** ソリューションは複数のプロジェクトをまとめて扱うための機能です。ソリューション用のフォルダを作ると、その中に複数のプロジェクト用フォルダを配置することができ、管理しやすくなります。しかし、今回のようにひとつのプロジェクトしか使わない場合、ソリューション用フォルダは冗長になりますから、作成しないほうがいいでしょう。

# Main.cppを追加する

それでは、最初のソースファイルを追加していきましょう。  
Visual Studioの右側に「ソリューション エクスプローラー」パネルが開いていると思います。もし見つからないときは、メニューから「表示→ソリューション エクスプローラー」を選択して「ソリューション エクスプローラー」パネルを開いてください。

「ソースファイル」フィルターを右クリックして右クリックメニューを開き、「追加→新しい項目」を選択して「新しい項目の追加」ウィンドウを開きます。右側パネルは「Visual C++」、中央パネルは「C++ファイル(.cpp)」が選択された状態になっていると思います。別の項目が選択されていた場合は、「Visual C++」「C++ファイル(.cpp)」をクリックして選択状態にして下さい。

ウィンドウ下部の「名前」テキストボックスの内容を「Source.cpp」から「Main.cpp」に変更します。さらに、「場所」テキストボックスの内容はプロジェクトのルートフォルダになっていると思います。これを「プロジェクトのルートフォルダ\Src」となるように変更しましょう。

D:\Your\Project\Folder\OpenGLTutorial\

↓

D:\Your\Project\Folder\OpenGLTutorial\**Src**

変更したら「追加」ボタンをクリックします。「ソリューション エクスプローラー」パネルに「Main.cpp」が追加されていることを確認して下さい。わざわざフォルダを作るのは、Visual Studioが作るファイルと、自分が作るファイルを、簡単に区別できるようにするためです。

それでは、最初のプログラムを書いていきます。「Main.cpp」をダブルクリックして開き、次のコードを入力して下さい。

**+**/\*\*  
**+**\* @file Main.cpp  
**+**\*/  
**+**  
**+**/\*\*  
**+**\* エントリーポイント.  
**+**\*/  
**+**int main()  
**+**{  
**+** return 0;  
**+**}

入力が終わったら、ツールバーの「ローカルWindowsデバッガー」をクリック(あるいはメニューから「デバッグ→デバッグの開始」をクリック)してビルド・起動してみましょう。正しく入力できていれば、出力ウィンドウに

プログラム '[5956] OpenGLTutorial.exe' はコード 0 (0x0) で終了しました。

のような文章が表示されると思います。エラーが表示された場合は、コードを確認・修正してもういちどビルド・起動して下さい。

今度は成功したでしょうか？おめでとうございます！Win32コンソールアプリケーションが作成できました！

# OpenGLの初期化

「OpenGLはクロスプラットフォームなライブラリである」と説明しました。しかし、その初期化方法はプラットフォームごとに違っています。その理由は、ウィンドウの作成方法や画面に描画する方法がプラットフォームによって異なっているからです。OpenGLは共通のグラフィックス機能を提供するのが主な目的なので、初期化に関しては各プラットフォーム上で動作するOpenGL実装の提供者に一任されています。

せっかくのクロスプラットフォームライブラリなのに、これではとても不便です。そう思った人は世の中に何人もいて、この問題をなんとかしてくれるとても便利なライブラリを作ってくれています。本講義でもそういったライブラリを使っていきますが、一方で、どのようなコードで初期化が行われるかを全く知らない、というのもあまりよろしくありません。そこで、初期化の内容をかいつまんで説明することにします。

**※この章のコードは説明用なので実装しないでください**

## OpenGLコンテキスト

OpenGLの初期化は2段階からなり、最初は「OpenGLコンテキスト」というものを作成します。  
OpenGLコンテキストは、OpenGL実行におけるあらゆる状態を保持します。また、実行環境による差異を吸収する役割も持っています。Windowsでは、OpenGLコンテキストを作成し、ウィンドウを表示するコードは次のようになります。

#include <Windows.h>  
#include <GL/GL.h>  
#pragma comment (lib, "opengl32.lib")  
  
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam);  
HGLRC oglContext;  
  
// エントリーポイント.  
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,  
 LPSTR lpCmdLine, int nShowCmd)  
{  
 WNDCLASS wc = {};   
 wc.lpfnWndProc = WndProc;  
 wc.hInstance = hInstance;  
 wc.hbrBackground = reinterpret\_cast<HBRUSH>(COLOR\_BACKGROUND);  
 wc.lpszClassName = L"OpenGLVersionCheck";  
 wc.style = CS\_OWNDC;  
 if (!RegisterClass(&wc)) {  
 return 1;  
 }  
 CreateWindowW(  
 wc.lpszClassName,L"OpenGL Version Check", WS\_OVERLAPPEDWINDOW|WS\_VISIBLE,  
 0, 0, 640, 480, 0, 0, hInstance, 0);  
  
 MSG msg;  
 while (GetMessage(&msg, nullptr, 0, 0) > 0) {  
 DispatchMessage(&msg);  
 }  
 return 0;  
}  
  
// ウィンドウプロシージャ.  
LRESULT CALLBACK WndProc(  
 HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)  
{  
 switch (message) {  
 case WM\_CREATE: {  
 PIXELFORMATDESCRIPTOR pfd = {};  
 pfd.nSize = sizeof(PIXELFORMATDESCRIPTOR);  
 pfd.nVersion = 1;  
 pfd.dwFlags = PFD\_DRAW\_TO\_WINDOW | PFD\_SUPPORT\_OPENGL | PFD\_DOUBLEBUFFER;  
 pfd.iPixelType = PFD\_TYPE\_RGBA;  
 pfd.cColorBits = 32;  
 pfd.cDepthBits = 24;  
 pfd.cStencilBits = 8;  
 pfd.iLayerType = PFD\_MAIN\_PLANE;  
  
 HDC hDC = GetDC(hWnd);  
 int pixelFormat = ChoosePixelFormat(hDC, &pfd);  
 SetPixelFormat(hDC, pixelFormat, &pfd);  
 oglContext = wglCreateContext(hDC);  
 wglMakeCurrent(hDC, oglContext);  
 ReleaseDC(hWnd, hDC);  
 MessageBoxA(  
 0, reinterpret\_cast<const char\*>(  
 glGetString(GL\_VERSION)), "OPENGL VERSION", 0);  
 break;  
 }  
 case WM\_DESTROY:  
 wglDeleteContext(oglContext);  
 PostQuitMessage(0);  
 break;  
 default:  
 return DefWindowProc(hWnd, message, wParam, lParam);  
 }  
 return 0;  
}

## 関数ポインタの取得

次に、必要なOpenGL関数へのポインタをグラフィックス・ドライバなどから取得します。  
WindowsではwglGetProcAddress関数を使用します。wglGetProcAddress関数は、引数として渡されたOpenGL関数名に対応する関数を検索し、その関数ポインタを返してくれます。また、対応する関数が見つからない場合はnullptrを返します。

まず、次のように、関数ポインタを格納する変数を定義します。

typedef const GLubyte\* (APIENTRY \*PFNGLGETSTRINGIPROC)(GLenum, GLuint);  
PFNGLGETSTRINGIPROC glGetStringi;

そして、wglGetProcAddress関数でポインタを取得し、その変数に格納します。

glGetStringi =  
 reinterpret\_cast<PFNGLGETSTRINGIPROC>(wglGetProcAddress("glGetStringi"));

この作業を、アプリケーションで必要とされる全ての関数に対して行うのですが…、OpenGLで定義された関数は100個以上あるんです。使う関数だけだとしても、いちいち関数ポインタを取得するコードを書くのは非常に面倒です。そこで、この作業にも便利なライブラリを使っていくことにします。

# ライブラリのインストール

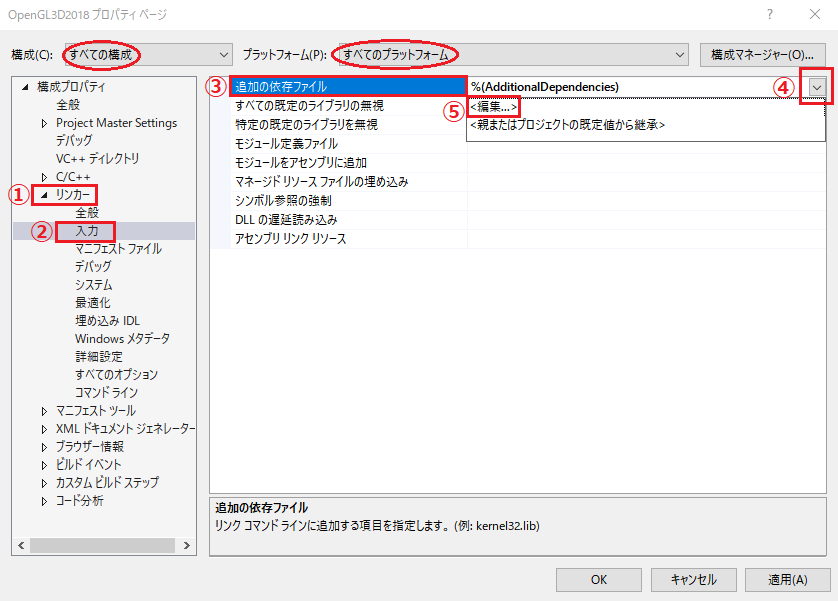
## リンカーの設定

まず最初に、OpenGLの基本的な関数を使えるようにします。Windowsでは、リンカーへの入力にopengl32.libというファイルを追加することで、基本的な関数が使えるようになります。

**[補足]** ソースコードをコンパイルすると「オブジェクトファイル」と呼ばれるものが生成されます。リンカーは複数のオブジェクトファイルをつなげて、一つの実行ファイルを作り出すプログラムです。

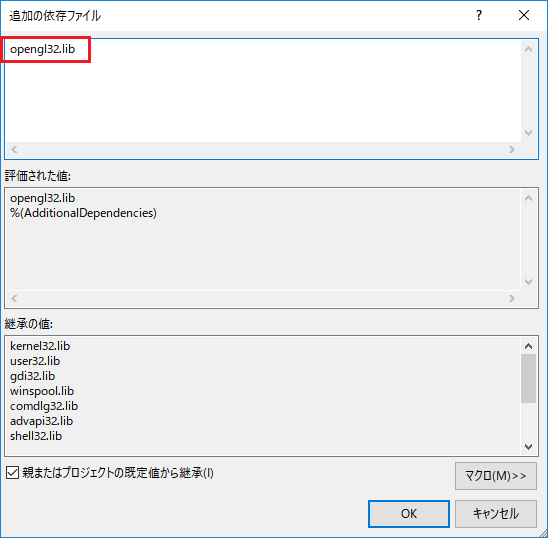
「ソリューションエクスプローラー」の「OpenGLTutorial」プロジェクトの上にカーソルを持っていき「右クリック→プロパティ」を選択します。すると、「OpenGLTutorial プロパティページ」というダイアログボックスが開くと思います。  
上段にある「構成」ドロップダウンリストから「すべての構成」を選択します。同様に「プラットフォーム」ドロップダウンリストから「すべてのプラットフォーム」を選択してください。  
これによって、プロジェクトの全てのビルド設定に対して同じプロパティを設定できるようになります。

次に、左側のツリービューから「リンカー」を展開し(左にある三角形をクリックすると展開できます)、「入力」を選択します(①、②)。右ペインに移り、「追加の依存ファイル」を選択すると、右端に下向き矢印のついたボタンが現れます(③)。これをクリックするとドロップダウンリストが開くので、「<編集…>」を選択してください(④、⑤)。すると「追加の依存ファイル」ダイアログボックスが開きます。



ダイアログボックスの上段にあるテキストウィンドウでカーソルが点滅していると思います。カーソルが別の場所に出ていた場合、上段のテキストウィンドウをクリックしてカーソルの位置を変更します。

最初の行に「opengl32.lib」と入力し、「OK」ボタンを押して「追加の依存ファイル」ダイアログボックスを閉じます。さらに、「OpenGLTutorial プロパティページ」ダイアログボックスの「OK」ボタンを押して、このダイアログボックスも閉じてください。



これでリンカーの設定は完了です。

## GLFWとGLEW

それでは、初期化を肩代わりしてくれる便利なライブラリをインストールしましょう。  
今回は、OpenGLコンテキストの作成には「GLFW」というライブラリ、関数ポインタの取得には「GLEW」というライブラリを使うことにしました。この2つはOpenGLの世界では広く使われているライブラリなので、Webで情報を検索する場合も、それほど苦労はしないと思います。

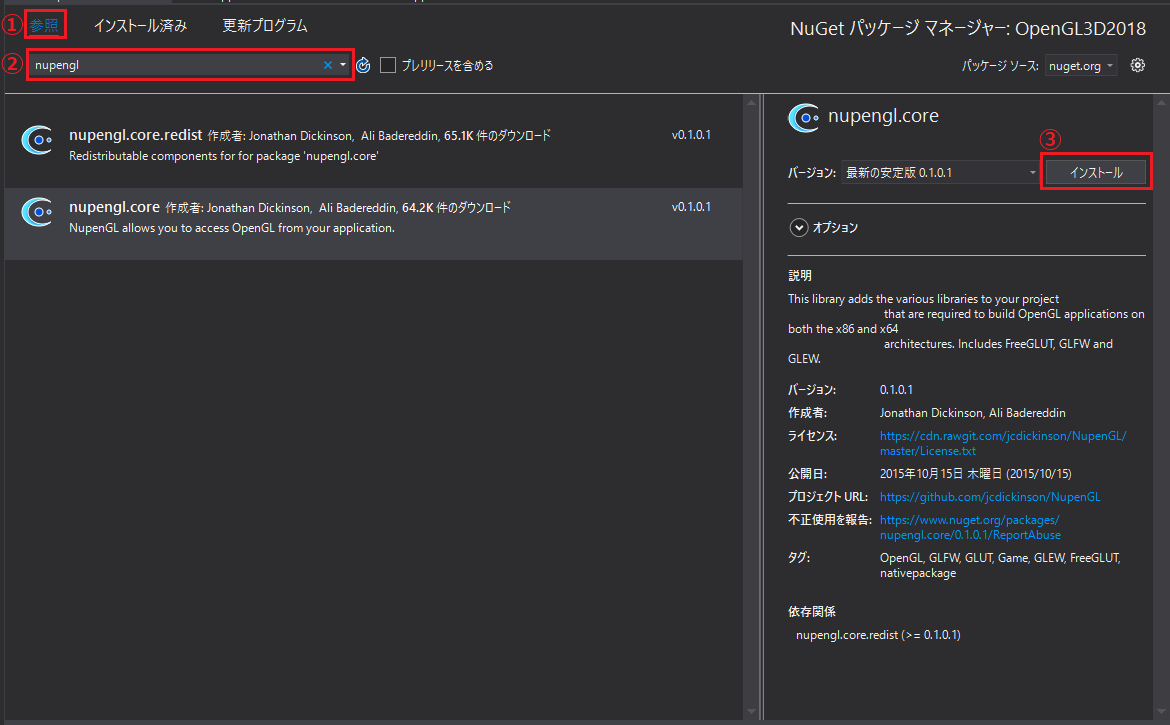
## NuGetパッケージマネージャ

さて、インストール方法ですが、今回はVisual Studioの機能であるNuGetパッケージマネージャを使います。それぞれのライブラリのWebサイトからダウンロードして、手動でインストールすることもできますが、NuGetを使うほうが簡単で、手間もかかりません。GLFWとGLEWをまとめてダウンロードしてインストールできる「nupengl.core」というNuGetパッケージがありますので、今回はこれを使うことにします。

メニューバーから「プロジェクト→NuGetパッケージの管理」を選択してください。「NuGetパッケージマネージャー」というウィンドウが開くと思います。  
左上にある「参照」という文字をクリックすると、左ペインにずらっとパッケージのリストが並びます(①)。  
どんどんスクロールさせていけば、nupengl.coreを見つけられるはずなので、がんばって探してください！

…見つかりませんか？

では、「参照」のすぐ下にある検索テキストボックスにnupenglと入力してみてください(②)。さくっと候補が表示されることでしょう(出てこなかったら綴りを確認してくださいね)。  
さて、検索すると、nupengl.core.registとnupengl.coreの2つが表示されると思いますnupengl.coreのほう(registのついてないほう)をクリックしてください。  
そうすると、右側に「インストール」ボタンが表示されます(③)。このボタンをクリックすると「変更の確認」ダイアログボックスが開き、本当にインストールするかをたずねてきますので、「OK」ボタンを押してください。選んだパッケージのインストールが始まります。



インストールが完了すると、左ペインのnupengl.coreのアイコンの右下にチェックマークが付いていると思います。  
これでパッケージのインストールは完了しました。「NuGet-ソリューション」ウィンドウを閉じてください。

**[補足]** nupengl 1.0.1に含まれるGLFWのバージョンは3.0.4、GLEWのバージョンは1.10.0となっています。これらは少し古いものなので、OpenGLの最新の機能には対応していません。そのような機能を使うにはそ、れぞれの公式サイトから最新版をダウンロードして、手動でインストールする必要がありますが、今回の講義範囲からは外れますので説明はしません。

# ライブラリの初期化

## ヘッダファイルのインクルード

それでは、ライブラリを使ってOpenGLを初期化していきましょう。まずはGLFWとGLEWのヘッダファイルをインクルードします。次のコードをMain.cppの先頭にあるブロックコメントの直後に追加してください。

/\*\*  
\* @file Main.cpp  
\*/  
**+**#include <GL/glew.h>  
**+**#include <GLFW/glfw3.h>  
**+**#include <iostream>  
  
/\*\*  
\* エントリーポイント.  
\*/

glew.hはGLEWの機能を使うためのヘッダファイルです。glfw3.hはGLFWの機能を使うためのヘッダファイルで、3というのはGLFWのバージョンを示しています。  
この2つのヘッダファイルの順番は重要です。OpenGLにはgl.hとglext.hという標準のヘッダファイルがあるのですが、上記のコードではそれらをインクルードしていません。なぜかというと、glew.hにgl.hおよびglext.hの内容が全て含まれているからです。これはあまり行儀の良い実装とはいえませんが、様々なプラットフォームに対応させる過程で、標準のヘッダでは対応しきれないケースがあったためと思われます。面倒なことに、glfw3.hのほうは内部でgl.hをインクルードしようとします。そのため、glfw3.hを先にインクルードしてしまうと、gl.hとglew.hが競合し、ビルドは失敗に終わります。  
こういう理由があるので、常にglew.hを先にインクルードするように気をつけなければなりません。

**[補足]** インクルード順を強制するのは、見つけにくいバグのもとになりやすいため、基本的には悪い習慣です。ヘッダファイルの設計する際は、インクルード順に依存することがないように心がけてください。一つの指針は、まず自分で書いたヘッダをインクルードし、次にライブラリのヘッダをインクルードすることです。

最後のiostreamヘッダは、エラーメッセージ等をコンソールウィンドウに出力するために使います。

## エラーコールバック

GLFWには、エラーが発生した場合に指定した関数を呼び出すしくみがあります。  
iostreamインクルード文の下に、次のコードを追加してください。

#include <GL/glew.h>  
#include <GLFW/glfw3.h>  
#include <iostream>  
  
**+**/\*\*  
**+**\* GLFWからのエラー報告を処理する.  
**+**\*  
**+**\* @param error エラー番号.  
**+**\* @param desc エラーの内容.  
**+**\*/  
**+**void ErrorCallback(int error, const char\* desc)  
**+**{  
**+** std::cerr << "ERROR: " << desc << std::endl;  
**+**}  
**+**  
/\*\*  
\* エントリーポイント.  
\*/

この関数は、GLFWから渡されたエラーメッセージを標準エラー出力に表示するだけの簡単なものです。  
実際、エラーコールバックでは、それ以外にできることはほとんどありません。

次に、ErrorCallback関数をGLFWに設定します。main関数の先頭に、次のコードを追加してください。

int main()  
{  
**+** glfwSetErrorCallback(ErrorCallback);  
**+**  
 return 0;  
}

glfwSetErrorCallback関数は、コールバックをGLFWに設定します。  
GLFWのほとんどの関数は、GLFWを初期化してからでないと呼び出すことはできないのですが、この関数はその数少ない例外です。

## GLFWの初期化

それでは、GLFWの初期化を実装していきましょう。glfwSetErrorCallback関数呼び出しの下に、次のコードを追加してください。

int main()  
{  
glfwSetErrorCallback(ErrorCallback);  
  
**+** // GLFWの初期化.  
**+** if (glfwInit() != GL\_TRUE) {  
**+** return 1;  
**+** }  
**+** GLFWwindow\* window =  
**+**  glfwCreateWindow(800, 600, "OpenGL Tutorial", nullptr, nullptr);  
**+** if (!window) {  
**+** glfwTerminate();  
**+** return 1;  
**+** }  
**+** glfwMakeContextCurrent(window);  
**+**  
 return 0;  
}

glfwInit関数は、OpenGL及びGLFWの初期化を行う関数です。成功するとGL\_TRUE、失敗するとGL\_FALSEを返します。GL\_TRUE/GL\_FALSEというのは、OpenGLの真偽値マクロです。

glfwCreateWindow関数は、グラフィックス描画用のウィンドウを作成します。引数は、描画範囲の縦横のピクセル数、ウィンドウのタイトルバーに表示される文字列、そして、フルスクリーンモード用のパラメータです。  
今回はウィンドウモードで実装を進めるため、フルスクリーンモード用のパラメータにはnullptrを渡しています。  
この関数は成功するとGLFWwindowオブジェクトへのポインタを返します。失敗するとnullptrを返します。  
nullptrが返された場合、glfwTerminate関数を呼び出して終了します。

glfwTerminate関数は、GLFWを終了させます。glfwInit関数呼び出しに成功した場合、アプリケーションを終了するまえに呼び出さなければなりません。

glfwMakeContextCurrent関数は、指定したウィンドウに対応するOpenGLコンテキストを描画対象に設定します。

## GLEWの初期化

GLFWの初期化に成功したら、続いてGLEWを初期化します。glfwMakeContextCurrent関数呼び出しの下に、次のコードを追加してください。

if (!window) {  
 glfwTerminate();  
 return 1;  
}  
glfwMakeContextCurrent(window);  
  
**+** // GLEWの初期化.  
**+** if (glewInit() != GLEW\_OK) {  
**+**  std::cerr << "ERROR: GLEWの初期化に失敗しました." << std::endl;  
**+**  glfwTerminate();  
**+**  return 1;  
**+** }  
**+**  
 return 0;  
}

glewInit関数は、GLEWを初期化します。成功するとGLEW\_OK、失敗するとそれ以外の値を返します。  
GLEWはGLFWとは直接の関係を持たないため、エラーがあってもGLFWのエラーコールバックは呼ばれません。そのため、明示的にエラーメッセージを出力しています。  
また、初期化に失敗した場合はglfwTerminateを呼び出してGLFWを終了させ、アプリケーションの終了に備えています。  
GLEWには終了させる関数がありません。というのも、GLEWは複雑な終了処理を必要とするオブジェクトを作成しないため、デフォルトの終了方法で十分だからです。

# メインループ

GLFWとGLEWの初期化が終わったら、次は毎フレーム実行されるメインループを実装します。  
GLEWの初期化コードの下に、次のコードを追加してください。

// GLEWの初期化.  
if (glewInit() != GLEW\_OK) {  
 std::cerr << "ERROR: GLEWの初期化に失敗しました." << std::endl;  
 glfwTerminate();  
 return 1;  
}  
  
**+** // メインループ.  
**+** while (!glfwWindowShouldClose(window)) {  
**+**  glClearColor(0.1f, 0.3f, 0.5f, 1.0f);  
**+** glClear(GL\_COLOR\_BUFFER\_BIT | GL\_DEPTH\_BUFFER\_BIT);  
**+** 　　glfwPollEvents();  
**+** glfwSwapBuffers(window);  
**+** }  
**+**  
 return 0;  
}

glfwWindowShouldClose関数は、ウィンドウを閉じるべきかどうかを調べる関数です。名前のまんまですね。引数で渡されたウィンドウに対して、OSなどから終了要求が来ていなければ0を、来ていれば0以外を返します。  
GLFWで作成したウィンドウが1つだけなら、終了判定はこの関数を見るだけで十分です。本講義では扱いませんが、もし複数のウィンドウを作成した場合は、最後のウィンドウ、あるいはメインウィンドウが閉じられたかどうかを判定する必要があるでしょう。

glClearColor関数は、glClear関数でバックバッファを消去するときに使われる色を指定します。引数はRGBAの各要素を0.0～1.0で表した値です。上記のコードでは、暗い青色を指定しています。

glClear関数は、実際にバックバッファを消去する関数です。引数には、消去するバッファの種類を示すGL\_???\_BUFFER\_BITフラグの論理和を指定します。上記のコードでは、カラーバッファと深度バッファを消去するように指定しています。

glfwPollEvents関数は、GLFWがOSから送られたイベントを処理するための関数です。GLFWでは、この関数を定期的に呼び出すか、あるいはglfwWaitEvents関数でイベントの発生を待機するか、いずれかの方法でOSからのイベントを処理する必要があります。

glfwSwapBuffers関数は、フレームバッファの表示側と描画側を入れ替えます。フレームバッファというのは絵を描画するためのメモリです。近代的なハードウェアでは、現在の画面を表示中に次の画面を描画し、描画が終わったら表示画面と描画画面を切り替えることで、描画中の画面をユーザーに見せないようにするのが一般的です。そのためには、少なくとも2枚のフレームバッファが必要となります。2枚の場合をダブルバッファ、3枚の場合をトリプルバッファを呼びます。枚数を増やしていくと、フレームレートを安定させやすくなりますが、そのかわり、ユーザーの入力が画面に反映されるまで、時間がかかるようになってしまいます。なお、「バックバッファ」というのは、描画側のフレームバッファを指す用語です。表示側のフレームバッファは「フロントバッファ」と呼ばれます。  
GLFWはダブルバッファを採用しています。

# 終了処理

最後は終了処理の実装です。メインループのコードの下に、次のコードを追加してください。

// メインループ.  
while (!glfwWindowShouldClose(window)) {  
 glClearColor(0.1f, 0.3f, 0.5f, 1.0f);  
 glClear(GL\_COLOR\_BUFFER\_BIT | GL\_DEPTH\_BUFFER\_BIT);  
　　glfwPollEvents();  
 glfwSwapBuffers(window);  
}  
  
**+** // GLFWの終了.  
**+** glfwTerminate();  
**+**  return 0;  
}

終了処理はglfwTerminate関数を呼び出すだけです。  
これで最小限のOpenGLアプリケーションの実装は完了です。ビルドして実行してください。  
黒いコンソールウィンドウと、青いウィンドウが表示されたら成功です。

**[課題01]** 画面を塗りつぶす色を変更してください。

# 描画デバイスと対応バージョンを出力する

OpenGLの関数を使って、描画デバイスの名前と、デバイスが対応しているOpenGLのバージョンを取得し、コンソールウィンドウに出力してみましょう。  
メインループの手前に、次のコードを追加してください。

std::cerr << "ERROR: GLEWの初期化に失敗しました." << std::endl;  
 glfwTerminate();  
 return 1;  
}  
 **+** // OpenGLの情報をコンソールウィンドウへ出力する.  
**+** const GLubyte\* renderer = glGetString(GL\_RENDERER);  
**+** std::cout << "Renderer: " << renderer << std::endl;  
**+** const GLubyte\* version = glGetString(GL\_VERSION);  
**+** std::cout << "Version: " << version << std::endl;  
**+** // メインループ.  
while (!glfwWindowShouldClose(window)) {  
 glClearColor(0.1f, 0.3f, 0.5f, 1.0f);  
 glClear(GL\_COLOR\_BUFFER\_BIT | GL\_DEPTH\_BUFFER\_BIT);

glGetString関数は、実行環境においてOpenGLコンテキストに関連付けられたいくつかの情報を取得する関数です。取得できる情報は次のとおりです:

**GL\_VENDOR**: 実行環境のOpenGL実装について、責任を持つ会社名.

**GL\_RENDERER**: OpenGLを描画するグラフィックスハードウェア名、またはドライバ名.

**GL\_VERSION**: グラフィックスハードウェア、またはドライバが対応している最大のOpenGLバージョン.

**GL\_SHADING\_LANGUAGE\_VERSION**: グラフィックスハードウェア、またはドライバが対応している最大のGLSLバージョン.

**GL\_EXTENSIONS**: グラフィックスハードウェア、またはドライバが対応しているOpenGL拡張名を、スペースで区切って並べたもの.

上記のコードを追加したら、ビルドして実行してみてください。コンソールウィンドウに、グラフィックスハードウェア名とOpenGLのバージョンが表示されると思います。

**[注意]** コンソールウィンドウは他のウィンドウの後ろに隠れてしまうことがあります。見当たらないときは、手前に表示されているウィンドウを移動させてみてください。

**[課題02]** glGetString関数を使い、GL\_RENDERER、GL\_VERSION以外の情報を取得して、その結果をコンソールウィンドウに表示してください。

# GLFWとGLEWをC++のクラスにまとめる

## ヘッダファイルの追加

ここまでのコードを、C++のクラスとして再設計してみましょう。  
まずは新しいクラスのために、ヘッダファイルを追加します。  
「ソリューションエクスプローラー」の「ヘッダーファイル」フィルターを右クリックして右クリックメニューを開きます。そして「追加→新しい項目」を選択すると、「新しい項目の追加」ダイアログボックスが開きます。中央ペインから「ヘッダーファイル(.h)」を選択し、下部の名前テキストボックスの内容を「GLFWEW.h」に変更します(安直な名前ですね…)。そして、場所コンボボックスの内容がSrcフォルダを指していることを確認してください。指していなかったら、テキストを直接編集するか、「参照」ボタンを押してSrcフォルダを選択してください。  
名前と場所を確認したら、「追加」ボタンを押すとファイルが追加されます。

それでは、GLFWEW.hを開き、次のコードを追加してください。

**-**#pragma once **+**/\*\*  
**+**\* @file GLFWEW.h  
**+**\*/  
**+**#ifndef GLFWEW\_H\_INCLUDED  
**+**#define GLFWEW\_H\_INCLUDED  
**+**#include <GL/glew.h>  
**+**#include <GLFW/glfw3.h>  
**+**  
**+**namespace GLFWEW {  
**+**  
**+**/\*\*  
**+**\* GLFWとGLEWのラッパークラス.  
**+**\*/  
**+**class Window  
**+**{  
**+**public:  
**+** static Window& Instance();  
**+** bool Init(int w, int h, const char\* title);  
**+** bool ShouldClose() const;  
**+** void SwapBuffers() const;  
**+**  
**+**private:  
**+** Window();  
**+** ~Window();  
**+** Window(const Window&) = delete;  
**+** Window& operator=(const Window&) = delete;  
**+**  
**+** bool isGLFWInitialized = false;  
**+** bool isInitialized = false;  
**+** GLFWwindow\* window = nullptr;  
**+**};  
**+**  
**+**} // namespace GLFWEW  
**+**  
**+**#endif // GLFWEW\_H\_INCLUDED

このコードでは、名前空間GLFWEWの中にWindowクラスを定義しています。

Windowクラスは、デフォルトコンストラクタとデストラクタをprivateとして宣言しています。また、コピーコンストラクタとコピー代入演算子を削除指定することで、コピーを禁止しています。GLFWやGLEWを管理するオブジェクトを複数作れるようになっていると、初期化処理や終了処理がオブジェクトの数だけ実行されてしまう危険性があります。そこで、Windowクラスの利用者がオブジェクトを作成・コピーすることを禁止しているのです。とはいえ、それだけだとWindowクラスを利用することができません。ですから、Instance関数をstatic関数として定義し、この関数のなかでオブジェクトを作成し、管理するようにします。このような仕組みを「シングルトン」と呼びます。

本講義では「#pragma once」のかわりに「インクルード・ガード」という技法を使っていくので、「#pragma once」を削除しています。そうする理由は、「#pragma onceはC++標準の機能ではないから」ということに尽きます。標準の機能にならないのにはいろいろな理由があるのですが、本講義では割愛します。対して、インクルード・ガードは#pragma onceと同じくファイルの多重インクルードを防ぐことができ、さらにC++標準の機能で実現されているので、あらゆる環境で気兼ねなく使うことができます。実際、多くのプログラムが#pragma onceを避けてインクルード・ガードを採用しています。使い方を学んで損はないと思います。  
さて、インクルード・ガードは次の形をしています。

#ifndef ファイル固有の名前  
#defineファイル固有の名前  
  
…多重インクルードを防ぎたいプログラム…  
  
#endif

これがどのように働くのか、ヘッダファイルA.hとソースファイルB.cppを例に説明します。  
A.hはインクルード・ガードされており、そのファイル固有の名前は「A\_H\_INCLUDED」だとしましょう。A.hがB.cppに初めてインクルードされたとき、A\_H\_INCLUDEDという名前は未定義です。そのため、#ifndefの条件(指定された名前が未定義なら#endifまでを実行)が成立し、次の行の#defineによってA\_H\_INCLUDEDが定義されます。A.hがもう一度B.ppにインクルードされたとき、A\_H\_INCLUDEDは最初のインクルードのときに定義されているので、今度は#ifndefは不成立となり、#endifまでの内容は無視されます。

## ソースファイルの追加

ヘッダファイルと同様にして、「GLFWEW.cpp」ファイルをSrcフォルダに追加してください。  
追加したらGLFWEW.cppファイルを開き、次のコードを追加します。

**+**/\*\*  
**+**\* @file GLFWEW.cpp  
**+**\*/  
**+**#include "GLFWEW.h"  
**+**#include <iostream>  
**+**  
**+**/// GLFWとGLEWをラップするための名前空間.  
**+**namespace GLFWEW {  
**+**  
**+**} // namespace GLFWEW

次に、名前空間の中にMain.cppで定義したエラーコールバックのコードをコピーしてください。これは、この後の実装で必要となります(最終的にはMain.cppのエラーコールバックは削除します)。

/// GLFWとGLEWをラップするための名前空間.  
namespace GLFWEW {  
  
**+**/\*\*  
**+**\* GLFWからのエラー報告を処理する.  
**+**\*  
**+**\* @param error エラー番号.  
**+**\* @param desc エラーの内容.  
**+**\*/  
**+**void ErrorCallback(int error, const char\* desc)  
**+**{  
**+** std::cerr << "ERROR: " << desc << std::endl;  
**+**}  
**+** } // namespace GLFWEW

## Instance関数

それでは、Windowクラスの各関数を実装していきましょう。最初はInstance関数です。  
GLFWEW名前空間の内側に、次のコードを追加してください。

void ErrorCallback(int error, const char\* desc)  
{  
 std::cerr << "ERROR: " << desc << std::endl;  
}  
 **+**/\*\*  
**+**\* シングルトンインスタンスを取得する.  
**+**\*  
**+**\* @return Windowのシングルトンインスタンス.  
**+**\*/  
**+**Window& Window::Instance()  
**+**{  
**+** static Window instance;  
**+** return instance;  
**+**}  
**+** } // namespace GLFWEW

Windowをstatic変数としてひとつだけ定義し、その参照を返しています。  
C++におけるシングルトンは、基本的にはこのように実装されます。

## コンストラクタ

次はコンストラクタです。Instance関数定義の下に、次のコードを追加してください。

Window& Window::Instance()  
{  
 static Window instance;  
 return instance;  
}  
  
**+**/\*\*  
**+**\* コンストラクタ.  
**+**\*/  
**+**Window::Window()  
**+**{  
**+**}  
**+** } // namespace GLFWEW

コンストラクタでは特に何もしません。

## デストラクタ

続いて、デストラクタを実装します。

Window::Window()  
{  
}  
  
**+**/\*\*  
**+**\* デストラクタ.  
**+**\*/  
**+**Window::~Window()  
**+**{  
**+** if (isGLFWInitialized) {  
**+** glfwTerminate();  
**+** }  
**+**}  
**+** } // namespace GLFWEW

GLFWが初期化されていた場合のみ、glfwTerminate関数を呼び出すようになっています。  
現在のコードでは何度もglfwTerminate関数を呼び出していますが、このように、デストラクタに終了責任を持たせておけば、それ以外の場所には書かないで済ませることができます。書き忘れることがなくなるので、バグを起こしにくくなるわけです。

## Init関数

初期化関数の内容は、基本的にはMain.cppで実装した初期化コードのコピーです。  
そこに、誤って複数回の初期化が行われないようにする処理を追加しています。

Window::~Window()  
{  
 if (isGLFWInitialized) {  
 glfwTerminate();  
 }  
 }  
  
**+**/\*\*  
**+**\* GLFW/GLEWの初期化.  
**+**\*  
**+**\* @param w ウィンドウの描画範囲の幅(ピクセル).  
**+**\* @param h ウィンドウの描画範囲の高さ(ピクセル).  
**+**\* @param title ウィンドウタイトル(UTF-8の0終端文字列).  
**+**\*  
**+**\* @retval true 初期化成功.  
**+**\* @retval false 初期化失敗.  
**+**\*/  
**+**bool Window::Init(int w, int h, const char\* title)  
**+**{  
**+** if (isInitialized) {  
**+** std::cerr << "ERROR: GLFWEWは既に初期化されています." << std::endl;  
**+** return false;  
**+** }  
**+** if (!isGLFWInitialized) {  
**+** glfwSetErrorCallback(ErrorCallback);  
**+** if (glfwInit() != GL\_TRUE) {  
**+** return false;  
**+** }  
**+** isGLFWInitialized = true;  
**+** }  
**+**  
**+** if (!window) {  
**+** window = glfwCreateWindow(w, h, title, nullptr, nullptr);  
**+** if (!window) {  
**+** return false;  
**+** }  
**+** glfwMakeContextCurrent(window);  
**+** }  
**+**  
**+** if (glewInit() != GLEW\_OK) {  
**+** std::cerr << "ERROR: GLEWの初期化に失敗しました." << std::endl;  
**+** return false;  
**+** }  
**+**  
**+** // OpenGLの情報をコンソールウィンドウへ出力する.  
**+** const GLubyte\* renderer = glGetString(GL\_RENDERER);  
**+** std::cout << "Renderer: " << renderer << std::endl;  
**+** const GLubyte\* version = glGetString(GL\_VERSION);  
**+** std::cout << "Version: " << version << std::endl;  
**+  
+** isInitialized = true;  
**+** return true;  
**+**}  
**+** } // namespace GLFWEW

## ShouldClose関数

これは、glfwWindowShouldClose関数のラッパーです。  
glfwWindowShouldClose関数を呼び、その戻り値が0でなければtrueを、0ならばfalseを返しています。

**[補足]** ラッパーというのは、関数やクラス、ライブラリを薄く包むようなコードを示す表現です。サランラップとかクレラップとか、ほかにも「商品のラッピング」なんて言いますよね。この「ラップ」に、「～する者」という意味の「-er」をつけたものです。

// OpenGLの情報をコンソールウィンドウへ出力する.  
 const GLubyte\* renderer = glGetString(GL\_RENDERER);  
 std::cout << "Renderer: " << renderer << std::endl;  
 const GLubyte\* version = glGetString(GL\_VERSION);  
 std::cout << "Version: " << version << std::endl;  
 isInitialized = true;  
 return true;  
}  
  
**+**/\*\*  
**+**\* ウィンドウを閉じるべきか調べる.  
**+**\*  
**+**\* @retval true 閉じる.  
**+**\* @retval false 閉じない.   
**+**\*/  
**+**bool Window::ShouldClose() const  
**+**{  
**+** return glfwWindowShouldClose(window) != 0;  
**+**}  
**+** } // namespace GLFWEW

## SwapBuffers関数

この関数は、glfwSwapBuffers関数とglfwPollEvents関数のラッパーです。この2つの関数は毎フレーム呼びだすべきものです。わざわざ分けて呼び出す必要はないため、この関数にまとめています。

bool Window::ShouldClose() const  
{  
 return glfwWindowShouldClose(window) != 0;  
}  
  
**+**/\*\*  
**+**\* フロントバッファとバックバッファを切り替える.  
**+**\*/  
**+**void Window::SwapBuffers() const  
**+**{  
**+** glfwPollEvents();  
**+** glfwSwapBuffers(window);  
**+**}  
**+** } // namespace GLFWEW

これでGLFW/GLEWをクラスにまとめることができました。  
ここで、「ソリューションエクスプローラー」のGLFWEW.cppを右クリックし、右クリックメニューを開いて「コンパイル」を選択してみてください。コンパイルが正常終了したら、正しく実装できています。失敗していたら、エラーメッセージをよく見てコードを修正してください。そして、もう一度コンパイルします。これを、正常終了するまで繰り返してください。

GLFW/GLEWには他にも有用な機能がありますが、それは今後必要になったときに、このクラスへ追加していくことにしましょう。

# Main.cppの修正

## インクルード文の修正

作成したGLFW/GLEWライブラリのラッパークラスを使って、Main.cppを書き換えていきます。  
まずはインクルードしているヘッダファイルを変更します。  
GL/glew.hとGLFW/glfw3.hのインクルード文を削除し、GLFWEW.hをインクルードしてください。  
iostreamのほうは、今後のログ出力のために残しておくことにします。修正したインクルード文のコードは次のようになります。

/\*\*  
\* @file Main.cpp  
\*/  
**-**#include <GL/glew.h>  
**-**#include <GLFW/glfw3.h> **+**#include "GLFWEW.h"  
 #include <iostream>  
/\*\*  
\* GLFWからのエラー報告を処理する.  
\*

## エラーコールバックの削除

エラー処理はGLFWEW.cppに実装しているので、Main.cppからはエラーコールバックのコードを削除してください。

#include "GLFWEW.h"  
#include <iostream>  
  
**-**/\*\*  
**-**\* GLFWからのエラー報告を処理する.  
**-**\*  
**-**\* @param error エラー番号.  
**-**\* @param desc エラーの内容.  
**-**\*/  
**-**void ErrorCallback(int error, const char\* desc)  
**-**{  
**-** std::cerr << "ERROR: " << desc << std::endl;  
**-**}  
**-**  
/\*\*  
\* エントリーポイント.  
\*/

## 初期化コードの修正

次に、GLFW/GLEWの初期化コードをごっそりと削除し、以下のコードに置き換えてください。

int main()  
{  
**-** glfwSetErrorCallback(ErrorCallback);  
**-  
-** // GLFWの初期化.  
**-** if (glfwInit() != GL\_TRUE) {  
**-** return 1;  
**-** }  
**- .**  
**- .**  
**- .**  
**-** // GLEWの初期化.  
**-** if (glewInit() != GLEW\_OK) {  
**-**  std::cerr << "ERROR: GLEWの初期化に失敗しました." << std::endl;  
**-**  glfwTerminate();  
**-**  return 1;  
**-** }  
**-**  
**-** // OpenGLの情報をコンソールウィンドウへ出力する.  
**-** const GLubyte\* renderer = glGetString(GL\_RENDERER);  
**-** std::cout << "Renderer: " << renderer << std::endl;  
**-** const GLubyte\* version = glGetString(GL\_VERSION);  
**-** std::cout << "Version: " << version << std::endl; **+** GLFWEW::Window& window = GLFWEW::Window::Instance();  
**+** if (!window.Init(800, 600, "OpenGL Tutorial")) {  
**+**  return 1;  
**+** }  
  
// メインループ.  
while (!glfwWindowShouldClose(window)) {

初期化コードは全てGLFWEW::Window::Init関数に移動させたので、Main.cppにわずかなコードを書くだけでGLFWとGLEWを初期化することができます。

## メインループの修正

最後に、メインループを次のように変更します。

// メインループ.  
**-** while (!glfwWindowShouldClose(window)) {  
**+** while (!window.ShouldClose()) {  
glClearColor(0.1f, 0.3f, 0.5f, 1.0f);  
 glClear(GL\_COLOR\_BUFFER\_BIT | GL\_DEPTH\_BUFFER\_BIT);  
**-** glfwPollEvents();  
**-**  glfwSwapBuffers(window);  
**+** 　 window.SwapBuffers();  
 }  
  
**-** // GLFWの終了.  
**-** glfwTerminate();  
**-** return 0;  
 }

glfwPollEvents関数とglfwSwapBuffers関数がなくなり、かわりにGLFWEW::Window::SwapBuffers関数を呼んでいます。また、メインループ直後のglfwTerminate関数呼び出しも削除しています。この関数はGLFWEW::Windowクラスのデストラクタが実行してくれるからです。

これでMain.cppの修正は完了です。ビルドして実行してみてください。  
修正前と同じように動作していたら成功です。