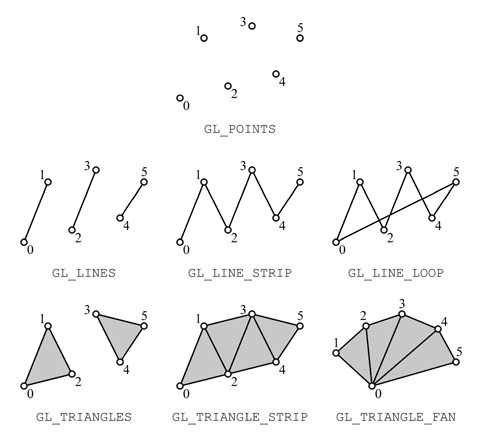
[OpenGL講座第02回]

OpenGLが描く3つの図形

# ポイントとライン、それにポリゴン

前回はOpenGLの初期化を行い、背景色を設定しました。今回は、OpenGLによる図形の描画について説明していきます。まずは、OpenGLがどんなものを描画できるのかを見てみましょう。  
現在、OpenGLが描画できるのは、次に示す7つの図形です。



「GL\_POINTS」は点の集まりです。1ドットの点を描くのに使います。  
「GL\_LINES」は直線の集まりで、「GL\_LINE\_STRIP」は連続した直線です。「GL\_LINE\_LOOP」は、GL\_LINE\_STRIPの最初と最後の点の間にも直線が引かれます。どれも、幅1ドットの直線を引くことができます。  
「GL\_TRIANGLES」は三角形の集まりです。「GL\_TRIANGLE\_STRIP」と「GL\_TRIANGLE\_FAN」は連続した三角形の集まりです。どちらも三角形をたくさん描くのに使いますが、描き方が異なっています。

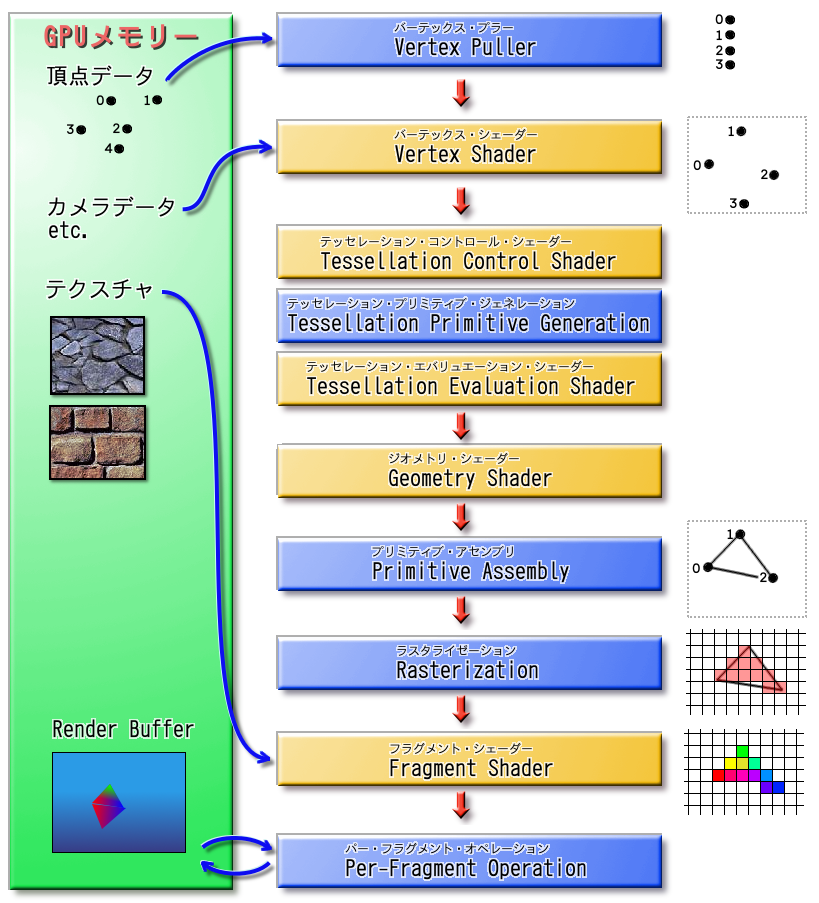
これらの図形のことを「プリミティブ」と呼びます(プリミティブというのは「原始的な」という意味の英単語)。見て分かるとおり、点と線、それから三角形以外のプリミティブはありません。そのような図形は、三角形を組み合わせて表現することになります。プリミティブを構成する点のことを「頂点」といいます。頂点の座標や色などのパラメータの集合を「頂点データ」といいます。世にあるゲームのキャラクターや背景のほとんどは、どんなに複雑な形状であっても、基本的にはこれらのプリミティブを組み合わせることで描画されています。

**[補足]** この他に「隣接頂点付きのプリミティブ」というものもあるのですが、本講義では扱いません。このタイプのプリミティブは「ジオメトリシェーダー」で使われます。

また、同じ頂点データであっても、プリミティブを変えることで異なる図形を描画することができます。

# グラフィックス・パイプライン

OpenGLはプリミティブを描画することできる、ということが分かりました。早速描画しよう！と行きたいところですが、その前に、OpenGLがどのような手順でプリミティブを描画するのかを説明します。  
OpenGLの描画は、それぞれが異なる処理を行う「ステージ」という単位に分けられています。ステージが順番に実行されることで、最終的にレンダー・バッファ(描画した画像を格納するメモリ領域)にプリミティブが描画されることになります。この、ステージが順番に実行される仕組みのことを「グラフィックス・パイプライン」と呼びます。  
グラフィックス・パイプラインは次のような構造になっています。



「石油パイプライン」というのを聞いたことがあるでしょう。金属製の筒(つつ)を、線のように長くつないだもので、石油はこの中を通って、ある地点から別の地点へと運ばれていきます。グラフィックス・パイプラインは名前の通り、石油ではなく画像を運びます。石油パイプラインとちょっと違うのは、画像がパイプラインの中を進んでいくごとに、少しずつ加工されていく点です。自動車工場なんかで製品がベルトコンベアの上を流れていって、それをちょっとずつ組み立てていく「ライン生産方式」に似ていますね。

## Vertex Puller(VP, ばーてっくす・ぷらー)

ここからは、パイプラインの動作を順番に説明します。  
Vertex Pullerはグラフィックス・パイプラインの最初のステージです。VPはGPUメモリからひとつずつ頂点データを取り出し、Vertex Shaderステージへ渡します。取り出されるのはVertex Shaderが実際に必要とする頂点データだけです。どの頂点データを取り出せばいいかを制御するには、「頂点アトリビュート」というパラメーターを設定します。

**[補足]** Vertex(ばーてっくす)は「頂点」という意味で、Pull(ぷる)は「引っ張る」という意味ですから、Vertex Pull-erは「頂点データを引っ張り出す装置」といった意味合いになります。

## Vertex Shader(VS, ばーってっくす・しぇーだー)

Vertex Shaderは最初のプログラム可能なステージです。VPで読み取られた全ての頂点データは、最初にこのステージで処理されます。VSでは、プリミティブの回転、拡大縮小、移動のほか、座標系の変換、頂点単位のライティングなど、様々な操作を行うことができます。VSは省略できません。

**[補足]** Shade(しぇーど)は「光を遮る、陰影を付ける」という意味です。ですから、Shade-rは「陰影を付ける装置」といった意味になります。

## Tessellation Control Shader (TCS, てっせれーしょん・こんとろーる・しぇーだー)

TCSは「テッセレーション」ステージの最初のステージです。テッセレーション・ステージは3つのステージで構成され、共同で「テッセレーション」と呼ばれる処理を行います。テッセレーションというのは「同じ模様を隙間なく敷き詰める」というような意味で、コンピューター・グラフィックスの世界では「描画する画像に細かなディテールを加えるために、プリミティブをより多くの小さなプリミティブに分割する」操作のことを指します。  
テッセレーションによって作られた小さなプリミティブは、通常はグラフィックス・パイプラインの中で作られ、グラフィックス・パイプラインの中で使用されます。そのため、通常のプリミティブと違い、CPU時間やGPUメモリを消費しません。そこで、例えば、入力データは低ポリゴンモデルにしておき、テッセレーション・ステージで高ポリゴンモデルを生成する、というふうに使います。用意する必要があるデータは低ポリゴンモデルのみなので、メモリが節約できるというわけです。

さて、TCSですが、新たに作成する小さなプリミティブの頂点データを、どこに、どのように作成するかを指定する作業を担当します。TCSはVSと同様にプログラム可能なステージです。テッセレーション・ステージは省略可能ですが、3つのステージはセットで扱われるため、3つ全てを使用するか、全て省略するかのいずれかになります。

本講義ではテッセレーションについては扱いません。そのため、TCSについてもこれ以上詳細な説明はしませんが、このようなステージがあるということは覚えておいてください。

## Tessellation Primitive Generation (TPG, てっせれーしょん・ぷりみてぃぶ・じぇねれーしょん)

TPGは「テッセレーション」ステージの2番目のステージです。これはVPと同じく固定機能のステージです。  
TPGはTCSからの情報を受け取り、実際に小さなプリミティブへと分割し、テッセレーションの最後のステージであるTESへと送ります。

## Tessellation Evaluation Shader (TES, てっせれーしょん・えばりゅえーしょん・しぇーだー)

TESは最後の「テッセレーション」ステージです。TESはTCS同様にプログラム可能なステージで、TPGから受け取ったプリミティブを変形し、細かなディテールを加える役割を持ちます。TPGから送られてきた小さなプリミティブは、元のプリミティブを分割しただけの状態なので、このステージで頂点の座標や色を調整します。

## Geometry Shader(GS, じおめとりー・しぇーだー)

GSは、ひとつのプリミティブの全頂点データを受け取り、0個以上のプリミティブの頂点データを生成して後段のステージへ送ります。このステージの特徴は、頂点データを増やすだけでなく、削除もできることです。  
プリミティブを増やすという意味では「テッセレーション」ステージと似ていますが、テッセレーションは「元のプリミティブを分割する」という処理に最適化され、大量のプリミティブを生成するのに向いているのに対して、GSは「プリミティブの頂点データから別のプリミティブを生成する、あるいは削除する」という、比較的小規模かつ汎用的な処理を得意としています。

GSはプログラム可能なステージです。また、テッセレーション・ステージと同様に省略することができます。本講義ではGSは扱いませんが、GSはパーティクルの描画や輪郭線の生成など、様々な用途で使われています。

## Primitive Assembly(PA, ぷりみてぃぶ・あせんぶりー)

PAは、前段のステージで変換された1つ以上の頂点データを蓄積し、プリミティブを構成できる数になった段階で、まとめて次のステージへ渡します。例えばGL\_TRIANGLESの場合は3つです。PAは固定機能のステージです。

## Rasterization(RS, らすたらいぜーしょん)

RSは、PAから送られたプリミティブをもとにフラグメントと呼ばれるピクセル片を生成し、後段のFragment Shaderへと送ります。RSは固定機能ステージです。  
各フラグメントは2～3つの頂点データを補間した値を持ちます。RSはまた、裏向きのプリミティブを除去したり(カリングといいます)、プリミティブの画面に表示されない部分を切り捨てたり (クリッピングといいます) します。  
また、RSは「ビューポート」というパラメータを使って、「クリッピング空間」から「スクリーン空間」への変換を行います。これらについては後の章で説明します。

## Fargment Shader(FS, ふらぐめんと・しぇーだー)

FSは最後のプログラム可能なステージで、RSより送られてきたフラグメントから、実際の色を計算します。VSがひとつの頂点データを受けとり、ひとつの頂点を送り出すように、FSもひとつのフラグメントを受け取り、ひとつの色を送り出します。FSは主に、フラグメント単位のライティングを行うために使用されます。

**[補足]** Fragment(ふらぐめんと)は「破片、断片」という意味です。OpenGLにおける「Fragment」とは、RSがプリミティブを分解したあとの、ピクセル単位の個々の領域を指す単語です。

## Per-Fragment Operations(PFO, ぱー・ふらぐめんと・おぺれーしょんず)

PFOはグラフィックス・パイプラインの最後のステージです。PFOは固定機能のステージで、FSから送られてきた色情報と深度情報をレンダー・バッファやデプス・ステンシル・バッファと呼ばれるメモリに書き込んでくれます。PFOはまた、FSからの色情報とレンダー・バッファの色を、さまざまな方法で合成する機能も備えています。  
このステージの結果がレンダー・バッファに書き込まれることになります。

## この章のまとめ

OpenGLでは、グラフィックス・パイプラインに頂点データや座標変換行列などのパラメータを送り込むことで、プリミティブを描画します。また、グラフィックス・パイプラインのパラメータを変えることで、同じ頂点データでも異なる表現を行うことが可能です。  
本講義では、実際に頂点シェーダーとフラグメントシェーダーのプログラムを作成してもらいます。

# 頂点バッファ

## 頂点データ型の定義

全てのプリミティブは頂点の集まりです。1章の図で、番号の振られた点のひとつひとつが頂点です。前述のとおり、OpenGLのプリミティブの種類は限定されていますが、頂点を定義するためのデータ構造は、ある程度自由に設計できるようになっています。  
各頂点には少なくとも16個のパラメータを割り当てることができます。それぞれのデータは8bitから64bitまで様々な型を使えます。プリミティブを描画するためには、まず頂点のデータ構造を設計し、その構造に従って頂点データを用意しなければなりません。そして、データ構造をOpenGLに登録します。頂点データはその情報に従ってグラフィックス・パイプラインへと送られます。

それでは、頂点データの構造を設計していきましょう。頂点データには、座標、頂点色、テクスチャ座標など、さまざまな要素が含まれます。これを、例えばfloat型の配列などで定義しても構わないのですが、どれがどの要素の値なのかが分かりにくいうえ、要素によって型を変えることができません。そこで、頂点データを表す構造体を定義することにします。Main.cppを開き、インクルード文の下に以下のコードを追加してください。

#include "GLFWEW.h"  
 #include <iostream>  
  
**+**/// 3Dベクトル型.  
**+**struct Vector3  
**+**{  
**+** float x, y, z;  
**+**};  
**+**  
**+**/// RGBAカラー型.  
**+**struct Color  
**+**{  
**+** float r, g, b, a;  
**+**};  
**+**  
**+**/// 頂点データ型.  
**+**struct Vertex  
**+**{  
**+** Vector3 position; ///< 座標  
**+** Color color; ///< 色  
**+**};  
**+**  
/\*\*  
\* エントリーポイント.  
\*/

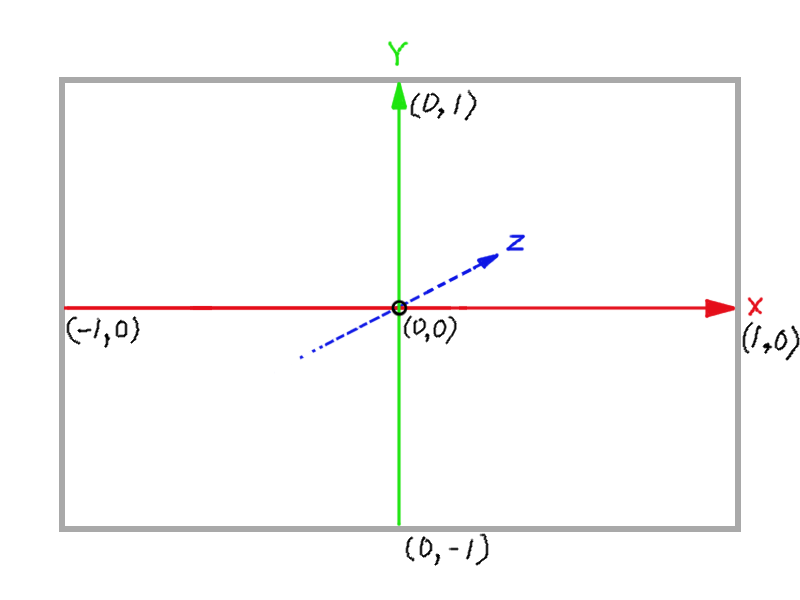
ここでは、ひとつの頂点データを表すVertexという構造体を定義しています(Vertexは「頂点」という意味の英単語です)。構造体の最初のメンバpositionは、頂点の座標を表しています。2つめのメンバcolorは頂点の色を表します。名前のままですね。それぞれのメンバの詳細は、Vector3とColorという構造体です。

## 頂点データを定義する

頂点データ型を使って、頂点データを定義しましょう。  
頂点データ型の定義の下に、次のコードを追加してください。

/// 頂点データ型.  
struct Vertex  
{  
 Vector3 position; ///< 座標  
 Color color; ///< 色  
};  
  
**+**/// 頂点データ.  
**+**const Vertex vertices[] = {  
**+** { {-0.5f, -0.5f, 0.5f}, {0.0f, 0.0f, 1.0f, 1.0f} },  
**+** { { 0.5f, -0.5f, 0.5f}, {0.0f, 1.0f, 0.0f, 1.0f} },  
**+** { { 0.0f, 0.5f, 0.5f}, {1.0f, 0.0f, 0.0f, 1.0f} },  
**+**};  
**+**  
/\*\*  
\* エントリーポイント.  
\*/

各列の最初の3つの値は頂点のX,Y,Z座標です。残りの4つはR,G,B,Aカラーです。  
頂点座標は「クリップ座標系」という空間内の位置を示します。クリップ座標系は、次に示すような、中心を原点(0,0)とする空間を定義します。



ここでは空間の大きさを±1としていますが、常にそうなるわけではありません。また、図ではZ軸が斜めに描かれていますが、実際には原点からまっすぐ奥へ向かっています。  
座標系については、次回以降の講義で説明します。

カラーは各色成分の量で、0.0～1.0の値をとります。例えば{1, 0, 0, 1}は「不透明な赤色」です。  
また、コンピュータグラフィックスのRGB成分は、色成分を重ねていくと明るさが上がる「加色混合法」が使われます。そのため、{1,1,1,1}は「不透明な白」、{0,0,0,1}は「不透明な黒」となります。

## VBOの作成

頂点データを格納するオブジェクトのことを「Vertex Buffer Object(VBO)」といいます。VBOはGPUメモリに作られるバッファの一種です。  
VBOを作成するコードは次のとおりです。この関数を、頂点データ定義の下に追加してください。

/// 頂点データ.  
const Vertex vertices[] = {  
 { {-0.5f, -0.5f, 0.5f}, {0.0f, 0.0f, 1.0f, 1.0f} },  
 { { 0.5f, -0.5f, 0.5f}, {0.0f, 1.0f, 0.0f, 1.0f} },  
 { { 0.0f, 0.5f, 0.5f}, {1.0f, 0.0f, 0.0f, 1.0f} },  
};  
  
**+**/\*\*  
**+**\* Vertex Buffer Objectを作成する.  
**+**\*  
**+**\* @param size 頂点データのサイズ.  
**+**\* @param data 頂点データへのポインタ.  
**+**\*  
**+**\* @return 作成したVBO.  
**+**\*/  
**+**GLuint CreateVBO(GLsizeiptr size, const GLvoid\* data)  
**+**{  
**+** GLuint vbo = 0;  
**+** glGenBuffers(1, &vbo);  
**+** glBindBuffer(GL\_ARRAY\_BUFFER, vbo);  
**+** glBufferData(GL\_ARRAY\_BUFFER, size, data, GL\_STATIC\_DRAW);  
**+** glBindBuffer(GL\_ARRAY\_BUFFER, 0);  
**+** return vbo;  
**+**}  
**+**  
/\*\*  
\* エントリーポイント.  
\*/

glGenBuffersはOpenGLが管理するメモリ領域を管理するオブジェクト(バッファ・オブジェクトと言います)を作成する関数です。最初の引数は作成するバッファ・オブジェクトの個数です。次の引数は、作成したバッファ・オブジェクトを格納する変数へのポインタです。

glBindBufferはバッファ・オブジェクトを特定の用途に割り当てます。GL\_ARRAY\_BUFFERは頂点データを示す定数です。指定できる用途は次のとおりです:

* GL\_ARRAY\_BUFFER: 頂点データ用バッファ
* GL\_ATOMIC\_COUNTER\_BUFFER: 排他的に操作されるカウンタ用バッファ
* GL\_COPY\_READ\_BUFFER: コピー操作の読み込み元バッファ
* GL\_COPY\_WRITE\_BUFFER: コピー操作の読み込み先バッファ
* GL\_DISPATCH\_INDIRECT\_BUFER: コンピュートシェーダーのインダイレクト処理用バッファ
* GL\_DRAW\_INDIRECT\_BUFFER: 頂点シェーダーのインダイレクト処理用バッファ
* GL\_ELEMENT\_ARRAY\_BUFFER: インデックスデータ用バッファ
* GL\_PIXEL\_PACK\_BUFFER: ピクセルデータ読み出し用バッファ
* GL\_PIXEL\_UNPACK\_BUFFER: ピクセルデータ書き出し用バッファ
* GL\_QUERY\_BUFFER: クエリ処理用バッファ
* GL\_SHADER\_STORAGE\_BUFFER: シェーダーから読み書きするためのバッファ
* GL\_TEXTURE\_BUFFER: テクスチャデータ用バッファ
* GL\_TRANSFORM\_FEEDBACK\_BUFFER: 座標変換結果を受け取るためのバッファ
* GL\_UNIFORM\_BUFFER: uniformデータ用バッファ

たくさんありますが、本講義で使うのはGL\_ARRAY\_BUFFER、GL\_ELEMENT\_ARRAY\_BUFFER、GL\_TEXTURE\_BUFFERの3つだけです。  
また、glBindBufferの第２引数に0を渡すことで、割り当てを解除できます。

glBufferDataは、バッファ・オブジェクトにデータを転送する関数です。最初の引数は、転送先のバッファの用途です。直前でvboをGL\_ARRAY\_BUFFER用に割り当てているので、結果的にvboへ転送されることになります。2つめの引数は転送バイト数です。3つめの引数は転送するデータへのポインタです。最後の引数は、転送先のバッファにどのようにアクセスするかに関するヒントを渡します。OpenGLはこのパラメータによって、バッファが最もパフォーマンスを発揮するような設定を行います。ヒントにはバッファへのアクセス頻度およびタイプによって、9種類の定数が定義されています(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)。

アクセス頻度には次の種類があります:

* STREAM: バッファは一度だけ転送され、僅かな回数だけ利用されます。
* STATIC: バッファは一度だけ転送され、何度も利用されます。
* DYNAMIC: バッファは何度も転送され、何度も利用されます。

アクセスのタイプには次の種類があります。

* DRAW: バッファはglBufferData等によってアプリケーションから更新され、OpenGLの描画処理のソースとして使われます。
* READ: バッファはOpenGLによって更新され、アプリケーションから読み出されます。
* COPY: バッファはOpenGLによって更新され、OpenGLの描画処理のソースとして使われます。

どのヒントを選択するかですが、頂点データを例にして考えてみましょう。  
頂点データは通常、初期化時に一度だけ転送され、描画のたびに利用されます。ということは、アクセス頻度は「STATIC」が妥当に思えますね。また、アプリケーションが転送し、OpenGLの描画処理で利用されることから、タイプは「DRAW」になりそうです。ということで、頂点データの場合はGL\_STATIC\_DRAWがよさそうです。  
しかし、例えばスプライトのように、頂点データを毎フレーム作成する場合はどうでしょう。何度も転送されることからアクセス頻度は「DYNAMIC」になります。タイプは「DRAW」のままでよさそうです。ということで、この場合はGL\_DYNAMIC\_DRAWが適当である、と考えられるわけです。

**[補足]** OpenGL 4.5からはglNamedBufferDataという関数が追加され、指定されたバッファ・オブジェクトに直接データを転送できるようになりました。この関数を使う場合、glBindBufferは不要です。

## VAOの作成

OpenGLでは、どの頂点データをどのシェーダーの入力に結びつける(バインドする)かを自由に設定できます。  
この設定をオブジェクトとしてまとめたものがVAOです。

さて、頂点データのバインドにはglSetVertexAttribPointer関数を使うのですが、これの引数の計算がちょっと面倒…。というのも、構造体メンバのオフセットを渡す必要があるのですが、これを合法的に得るには「offsetof」というプリプロセッサマクロを使うしかありません。しかも、そうやって得られる値は整数なのに、引数の型はポインタ型なので、キャストが必須なんですね。本当に面倒くさいですが、そうなっているので仕方がありません。では、CreateVBO関数定義の下に次のコードを追加してください。

GLuint CreateVBO(GLsizeiptr size, const GLvoid\* data)  
{  
 GLuint vbo = 0;  
 glGenBuffers(1, &vbo);  
 glBindBuffer(GL\_ARRAY\_BUFFER, vbo);  
 glBufferData(GL\_ARRAY\_BUFFER, size, data, GL\_STATIC\_DRAW);  
 glBindBuffer(GL\_ARRAY\_BUFFER, 0);  
 return vbo;  
}  
  
**+**/\*\*  
**+**\* Vertex Array Objectを作成する.  
**+**\*  
**+**\* @param vbo VAOに関連付けられるVBO.  
**+**\*  
**+**\* @return 作成したVAO.  
**+**\*/  
**+**GLuint CreateVAO(GLuint vbo)  
**+**{  
**+** GLuint vao = 0;  
**+** glGenVertexArrays(1, &vao);  
**+** glBindVertexArray(vao);  
**+** glBindBuffer(GL\_ARRAY\_BUFFER, vbo);  
**+** glEnableVertexAttribArray(0);  
**+** glVertexAttribPointer(0, sizeof(Vertex::position) / sizeof(float),  
**+** GL\_FLOAT, GL\_FALSE, sizeof(Vertex), (const　GLvoid\*)(offsetof(Vertex, position)));  
**+** glEnableVertexAttribArray(1);  
**+** glVertexAttribPointer(1, sizeof(Vertex::color) / sizeof(float),  
**+** GL\_FLOAT, GL\_FALSE, sizeof(Vertex), (const　GLvoid\*)(offsetof(Vertex, color)));  
**+** glBindVertexArray(0);  
**+** glDeleteBuffers(1, &vbo);  
**+** return vao;  
**+**}  
**+**  
/\*\*  
\* エントリーポイント.  
\*/

最初に、glGenVertexArrays関数でVAOを作成しています。次のglBindVertexArrayは、指定されたVAOをOpenGL「現在の処理対象」に設定する関数です。  
その次にglBindBuffer関数を呼んでいるのは、頂点アトリビュートを設定するには、事前に対応するVBOを割り当てておく必要があるためです。

**[補足]** このように、OpenGLでは、一見なんのつながりもなさそうな関数同士が密接に関係していることがよくあります。そして、個々の関数のドキュメントを読む以外に、関係があるかどうかを調べる方法はありません。

ここでは2つのOpenGL関数を呼んでいます。glEnableVertexAttribArrayは、指定した頂点アトリビュートを有効にする関数です。一度有効にしたら、glDisableVertexAttribArrayで無効化するまではずっと有効なままです。glVertexAttribPointerは、頂点データのどの部分を頂点シェーダーの入力先に割り当てるかを設定する関数です。  
それぞれの引数を見ていきましょう。最初の0は、バインド先の頂点アトリビュートのインデックスになります。

2つめの引数に出てくる「sizeof」というのは、クラスや構造体のバイト数を求めるC/C++の演算子です。  
float型は通常4バイトなので、sizeof(float)=4になります。また、VertexのpositionメンバはVector3型で、Vector3は3つのfloatを含んでいます。つまり、sizeo(Vector::position)=4×3=12です。  
そして、sizeof(Vertex::position)/sizeof(float)=12/4=3となり、最終的な引数の値は3ということになります。

3つめの引数はVertex型のサイズです、Vertex型はVector3とColorを含んでおり、それぞれのバイト数は12と16です。つまりVertexのサイズは28で、これが3つめの引数の最終的な値になります。

4つめの引数は2つの機能からなります。かっこの内側のoffsetofマクロが重要です。これは、第１引数の型に含まれる第２引数のメンバについて、第１引数の型内に配置されている位置(オフセット)を返すマクロです。  
このコードの場合、positionメンバはVertex型の最初のメンバなので、オフセットは0になります。また、colorメンバの場合はpositionメンバのサイズだけ後ろにずれるので、オフセットは12です。  
このプログラムではキャストを使っています。offsetofはsize\_t型を返しますが、glVertexAttribPointerはconst GLvoid\*型を要求するので、キャストして強制的に型を合わせているわけです。

**[補足]** OpenGLは暗黙的にconst GLvoid\*を整数値として扱っているようです。実際、現在のほとんどの処理系では問題なく動きます。しかし、実はC/C++言語の規格では、そうなる保証はありません。特に0は、特別にnullptrに変換可能なことになっていますが、ポインタとしての表現が数値の0と等しいかどうかは保証されません。過去にはnullptrとして0xB000000000を使っていたマシンなんかがあったりします。そのような骨董品ではOpenGLを動作させることはできないでしょうが、「コード上の0がnullptrとして扱われることと、nullptrの内部表現は無関係」ということだけは覚えておいてください。

さて、その次の関数はglBindVertexArrayです。引数に0を渡すことで、VAOのバインドを解除しています。最後のglDeleteBuffers関数は、VBOを削除する関数です。この関数はVBO以外にもさまざまなバッファ・オブジェクトの削除に使われます。この関数は、バッファをその場で削除するとは限りません。このコードのように、VAOに割り当てられている場合、単に「削除可能」というマークを付けるだけです。そして、全ての割り当てが解除されたとき、OpenGLは自動的にマークされたバッファを削除します。削除マークを付けない場合、削除の責任はプログラマが負うことになります。

# シェーダー

## シェーダー・コード

頂点バッファを作成する準備が出来たので、次はシェーダーを作成していきます。  
OpenGLのシェーダーはGLSL(OpenGL Shading Language:おーぷんじーえる・しぇーでぃんぐ・らんげーじ)という言語で記述します。GLSLはC言語を元にして、シェーダーにとって不要な機能を削除し、必要な機能や型を追加した言語です。普通のCコンパイラではGLSLをビルドできないので、OpenGLの機能を使ってビルドする必要があります。

最初に、シェーダー・コードを用意しましょう。一般的にはファイルから読み込むのですが、とりあえずMain.cppに直接書いていくことにします。頂点データ定義の下に、次のコードを追加してください。

/// 頂点データ.  
const Vertex vertices[] = {  
 { {-0.5f, -0.5f, 0.5f}, {0.0f, 0.0f, 1.0f, 1.0f} },  
 { { 0.5f, -0.5f, 0.5f}, {0.0f, 1.0f, 0.0f, 1.0f} },  
 { { 0.0f, 0.5f, 0.5f}, {1.0f, 0.0f, 0.0f, 1.0f} },  
};  
  
**+**/// 頂点シェーダー.  
**+**static const char\* vsCode =  
**+** "#version 410 \n"  
**+** "layout(location=0) in vec3 vPosition; \n"  
**+** "layout(location=1) in vec4 vColor; \n"  
**+** "layout(location=0) out vec4 outColor; \n"  
**+** "void main() { \n"  
**+** " outColor = vColor; \n"  
**+** " gl\_Position = vec4(vPosition, 1.0); \n"  
**+** "}";  
**+**  
**+**/// フラグメントシェーダー.  
**+**static const char\* fsCode =  
**+** "#version 410 \n"  
**+** "layout(location=0) in vec4 inColor; \n"  
**+** "out vec4 fragColor; \n"  
**+** "void main() { \n"  
**+** " fragColor = inColor; \n"  
**+** "}";  
**+** /\*\*  
\* Vertex Buffer Objectを作成する.  
\*  
\* @param size 頂点データのサイズ.  
\* @param data 頂点データへのポインタ.

## バージョン番号

各シェーダー・プログラムの先頭にあるのはシェーダーのバージョン指定です。  
OpenGLのバージョン番号とGLSLのバージョン番号、それから「#version」に指定する値の対応表を示します。

|  |  |  |
| --- | --- | --- |
| OpenGLバージョン | GLSLバージョン | #versionに指定する値 |
| 1.5 | 1.0 | **なし** |
| 2.0 | 1.1 | 110 |
| 2.1 | 1.2 | 120 |
| 3.0 | 1.3 | 130 |
| 3.1 | 1.4 | 140 |
| 3.2 | 1.5 | 150 |
| 3.3**以降** | 3.3**～** | 330**～** |

OpenGL 3.3以前は、OpenGLのバージョンが上がる毎にGLSLの小数点以下が1ずつ増えていました。3.3以降は、OpenGLとGLSLは同じバージョン番号を使うようになっています。  
そして、「#version」に指定する値は、GLSLのバージョン番号を100倍した値になっています。

## in/out(いん/あうと)修飾子

バージョン番号からmain関数の間にあるのは、各シェーダー・ステージへの入出力用の変数定義です。  
in修飾子のついた変数にはシェーダーへの入力が格納されます。  
out修飾子の突いた変数には、シェーダーが何らかの値を設定して、次のステージに処理を引き継ぎます。  
頂点シェーダーではvPositionとvColorが入力変数で、outColorが出力変数になります。フラグメントシェーダーではoutColorが入力変数で、fragColorが出力変数になります。

## layout(れいあうと)修飾子

いくつかの変数定義の先頭にあるlayoutというのは「layout修飾子」と呼ばれるもので、次の構文を持ちます。

layout(qualifier1 = value, qualifier2 = value, …)

layout修飾子には、変数の入出力先や属性を指定する機能があります。例えば入出力先を指定するには、シェーダー・コードのようにqualifier(くおりふぁいあー)の部分に「location(ろけーしょん)修飾子」を指定し、value(ばりゅー)の部分に「インデックス(何番目の入出力先を使うか)」を指定します。  
入出力先のインデックスは、各シェーダーの入力用と出力用で別々に指定しなければなりません。シェーダー・コードではvPositionとoutColorは同じ番号が指定されています。しかし、vPositionはin(入力)、outColorはout(出力)が指定されているので、番号が同じでも、違う場所を指しています。また、前段のシェーダーの出力先が後段のシェーダーの入力先になります。ですから、入出力先となる番号は、シェーダーのつながりを考慮して決めなければなりません。今回の場合、頂点シェーダーのoutColorとフラグメントーシェーダーinColorには同じ番号を指定していますので、頂点シェーダーからoutColorへと出力されたデータは、フラグメントシェーダーのinColorにコピーされます。このように、先に実行されるシェーダーの出力番号と、後から実行されるシェーダーの入力番号を合わせておくことはとても重要です。  
なお、頂点シェーダーには前段シェーダーがないので、入力の番号はglVertexAttribPointer関数を使って、通常のプログラムから指定しなければなりません。VPは、この関数で指定された番号の入力先変数に対して頂点データを引き渡します。

## vec3/vec4(べく・すりー/べく・ふぉー)型

これらはGLSLで定義されているベクター型です。それぞれ、3～4つのfloat型をまとめたものです。  
ベクター型はコンストラクタを持っていて、「vec4(vPosition,　1.0)」のようにして値を作成することができます。

## 頂点シェーダーのmain関数

頂点シェーダーのmain関数の内容はわずか2行です。入力変数vColorを出力変数outColorに、入力変数vPositionを謎の変数gl\_Position(じーえる・ぽじしょん)にコピーしています。  
GLSLはいくつかのグローバル変数を標準で提供していて、それらはプログラマが定義しなくても使うことができます。  
gl\_Positionはそのような変数のひとつです。ステージ間で頂点座標を受け渡すには、この変数に格納しなければならないと決められています。また、gl\_Positionはvec4型として定義されているので、コンストラクタでvec3型のvPosition変数からvec4型の一時変数を作成して渡しています。

## フラグメントシェーダーのmain関数

フラグメントシェーダーのmain関数の内容はたった1行、入力変数outColorを出力変数fragColorにコピーしているだけです。  
この出力変数の値が、フラグメントの色になります。

# シェーダーコードのコンパイル

シェーダーは文字列のままでは使えません。OpenGLの関数を組み合わせてビルドする必要があります。  
Main.cppを簡潔に保つためには、シェーダーをビルドする機能は別のファイルに分けるほうがいいでしょう。Srcフォルダに「Shader.h」という名前のヘッダー・ファイルを追加してください。  
追加したShader.hを開き、次のプログラムを追加してください。

**+**/\*\*  
**+**\* @file Shader.h  
**+**\*/  
**+**#ifndef SHADER\_H\_INCLUDED  
**+**#define SHADER\_H\_INCLUDED  
**+**#include <glew/glew.h>  
**+**  
**+**namespace Shader {  
**+  
+**GLuint Build(const GLchar\* vsCode, const GLchar\* fsCode);  
**+**  
**+**} // Shader namespace  
**+**  
**+**#endif // SHADER\_H\_INCLUDED

ヘッダファイルの次はC++ファイルを追加します。Srcフォルダに「Shader.cpp」という名前のC++ファイルを追加してください。  
さて、今回作成するコードではvectorクラスを使用します。そこで、まずはヘッダファイルをインクルードします。Shader.cppを開き、次のプログラムを追加してください。

**+**/\*\*  
**+**\* @file Shader.cpp  
**+**\*/  
**+**#include "Shader.h"  
**+**#include <vector>  
**+**#include <iostream>  
**+  
+**/\*\*  
**+**\* シェーダーに関する機能を格納する名前空間.  
**+**\*/  
**+**namespace Shader {  
**+  
+**} // Shader namespace

それでは、シェーダーをコンパイルするコードを追加していきましょう。Shader名前空間の内側に、次のコードを追加してください。

/\*\*  
\* シェーダーに関する機能を格納する名前空間.  
\*/  
namespace Shader {  
 **+**/\*\*  
**+**\* シェーダー・コードをコンパイルする.  
**+**\*  
**+**\* @param type シェーダーの種類.  
**+**\* @param string シェーダー・コードへのポインタ.  
**+**\*  
**+**\* @retval 0より大きい 作成したシェーダー・オブジェクト.  
**+**\* @retval 0 シェーダー・オブジェクトの作成に失敗.  
**+**\*/  
**+**GLuint Compile(GLenum type, const GLchar\* string)  
**+**{  
**+** GLuint shader = glCreateShader(type);  
**+** glShaderSource(shader, 1, &string, nullptr);  
**+** glCompileShader(shader);  
**+** GLint compiled = 0;  
**+** glGetShaderiv(shader, GL\_COMPILE\_STATUS, &compiled);  
**+** // コンパイルに失敗した場合、原因をコンソールに出力して0を返す.  
**+** if (!compiled) {  
**+** GLint infoLen = 0;  
**+** glGetShaderiv(shader, GL\_INFO\_LOG\_LENGTH, &infoLen);  
**+** if (infoLen) {  
**+** std::vector<char> buf;  
**+** buf.resize(infoLen);  
**+** if (static\_cast<int>(buf.size()) >= infoLen) {  
**+** glGetShaderInfoLog(shader, infoLen, NULL, buf.data());  
**+** std::cerr << "ERROR: シェーダーのコンパイルに失敗.\n" << buf.data() << std::endl;  
**+** }  
**+** }  
**+** glDeleteShader(shader);  
**+** return 0;  
**+** }  
**+** return shader;  
**+**}  
  
} // Shader namespace

シェーダーをコンパイルするには、まずglCreateShaderでシェーダー・オブジェクトを作成し、glShaderSourceで作成したシェーダー・オブジェクトにシェーダー・コードを関連付けます。そうしておいて、glCompileShader関数を呼び出すと、関連付けられたシェーダー・コードがコンパイルされ、シェーダー・オブジェクトにセットされるわけです。

さて、そこから下は、コンパイルに失敗した場合にその原因を表示する処理です。  
glGetShaderivは、シェーダー・オブジェクトの状態を取得する関数です。ここではGL\_COMPILE\_STATUSを指定して、コンパイル結果を取得しています。コンパイルに成功していればcompiled変数には0以外の値が入ります。失敗していたら0が入るので、より詳細な情報を取得します。  
glGetShaderivにGL\_INFO\_LENGTHを指定すると、詳細情報のバイト数を取得することができます。詳細情報が何バイトなのかは状況によるので、このようにして取得しなければなりません。  
詳細情報のバイト数が0の場合、詳細情報はありません。1以上の場合は何らかの情報が存在します。ここで、詳細情報を取得するためのバッファとしてstd::vectorを使っています。  
std::vector::resizeは、std::vectorのサイズを変更する関数です。そして、この関数が成功した場合、そのサイズは引数以上、つまりinfoLen以上になっているはずです。なっていなければ情報を格納することができないため、処理を終了します。十分な大きさがあれば、glGetShaderInfoLog関数でbufに詳細情報を取り込み、その内容を出力しています。

最後に、エラーがあった場合はglDeleteShaderでシェーダー・オブジェクトを削除し、0を返しています。  
コンパイルに成功していれば、作成したシェーダー・オブジェクトを返します。

# シェーダー・オブジェクトのリンク

シェーダー・オブジェクトを作成したら、それをリンクしてプログラム・オブジェクトにします。  
Complie関数定義の下に、次のコードを追加してください。

glDeleteShader(shader);  
 return 0;  
 }  
 return shader;  
}  
 **+**/\*\*  
**+**\* プログラム・オブジェクトを作成する.  
**+**\*  
**+**\* @param vsCode 頂点シェーダー・コードへのポインタ.  
**+**\* @param fsCode フラグメントシェーダー・コードへのポインタ.  
**+**\*  
**+**\* @retval 0より大きい 作成したプログラム・オブジェクト.  
**+**\* @retval 0 プログラム・オブジェクトの作成に失敗.  
**+**\*/  
**+**GLuint Build(const GLchar\* vsCode, const GLchar\* fsCode)  
**+**{  
**+** GLuint vs = Compile(GL\_VERTEX\_SHADER, vsCode);  
**+** GLuint fs = Compile(GL\_FRAGMENT\_SHADER, fsCode);  
**+**　　if (!vs || !fs) {  
**+** return 0;  
**+** }  
**+** GLuint program = glCreateProgram();  
**+** glAttachShader(program, fs);  
**+** glDeleteShader(fs);  
**+** glAttachShader(program, vs);  
**+** glDeleteShader(vs);  
**+** glLinkProgram(program);  
**+** GLint linkStatus = GL\_FALSE;  
**+** glGetProgramiv(program, GL\_LINK\_STATUS, &linkStatus);  
**+** if (linkStatus != GL\_TRUE) {  
**+** GLint infoLen = 0;  
**+** glGetProgramiv(program, GL\_INFO\_LOG\_LENGTH, &infoLen);  
**+** if (infoLen) {  
**+** std::vector<char> buf;  
**+** buf.resize(infoLen);  
**+** if (static\_cast<int>(buf.size()) >= infoLen) {  
**+** glGetProgramInfoLog(program, infoLen, NULL, buf.data());  
**+** std::cerr << "ERROR: シェーダーのリンクに失敗.\n"<< buf.data() << std::endl;  
**+** }  
**+** }  
**+** glDeleteProgram(program);  
**+** return 0;  
**+** }  
**+** return program;  
**+**}  
**+**  
} // Shader namespace

はじめに、先ほど作成したComplile関数を使ってシェーダー・コードからシェーダー・オブジェクトを作成しています。ここで失敗していたら0を返して終了します。次に、glCreateProgram関数でプログラム・オブジェクトを作成し、glAttachShader関数でシェーダー・オブジェクトをプログラム・オブジェクトに割り当てます。そして、glLinkProgram関数で割り当てたシェーダ・オブジェクト同士をリンクしています。

ところで、シェーダーを割り当てた直後にglDeleteShaderを呼んでいますね。この関数は、glDeleteBuffers関数と同様に働きます。つまり、シェーダーがプログラム・オブジェクトに割り当てられている場合は、単に「削除可能」というマークを付けるだけです。そして、シェーダー・オブジェクトがどのプログラム・オブジェクトにも割り当てられていない状態になったとき、OpenGLは自動的にマークされたシェーダー・オブジェクトを削除します。削除マークを付けない場合、削除の責任はプログラマが負うことになります。

さて、そのあとは、コンパイルのときと同様に、リンクに失敗した場合の原因の表示を行うコードが続きます。  
glGetProgramiv関数にGL\_LINK\_STATUSを指定すると、リンク結果が取得できます。そのあとはコンパイル失敗のコードと同様に、詳細情報を取得してその内容を出力しています。  
最後に、エラーがあった場合はglDeleteProgramでプログラム・オブジェクトを削除して0を返します。  
リンクに成功した場合は、作成したプログラム・オブジェクトを返します。

これで、シェーダーを作成する準備は整いました。

# 三角形を描く

## オブジェクトの作成

頂点データを定義し、シェーダーを使う準備も整いました。いよいよ、それらを使って図形を描画していきます。  
まずヘッダーファイルをインクルードします。Main.cppを開き、次のプログラムを追加してください。

#include "GLFWEW.h"  
**+**#include "Shader.h"  
 #include <iostream>  
  
/// 3Dベクトル型.  
struct Vector3  
{  
 float x, y, z;  
};

続いて、メインループの手前に、次のプログラムを追加してください。

GLFWEW::Window& window = GLFWEW::Window::Instance();  
if (!window.Init(800, 600, "OpenGL Tutorial")) {  
 return 1;  
}  
  
**+**const GLuint vbo = CreateVBO(sizeof(vertices), vertices);  
**+**const GLuint vao = CreateVAO(vbo);  
**+**const GLuint shaderProgram = Shader::Build(vsCode, fsCode);  
**+**if (!vbo || !vao || !shaderProgram) {  
**+** return 1;  
**+**}  
**+**  
// メインループ.  
while (!window.ShouldClose()) {  
glClearColor(0.1f, 0.3f, 0.5f, 1.0f);  
 glClear(GL\_COLOR\_BUFFER\_BIT | GL\_DEPTH\_BUFFER\_BIT);  
 　window.SwapBuffers();  
 }

ここまでに実装した関数を呼び出し、作成したオブジェクトを変数に格納しています。そして、いずれかの作成に失敗した場合は1を返してプログラムを終了します。

## 頂点データの描画

次は頂点データを描画します。メインループ内のglClear関数呼び出しの直後に、次のコードを追加してください。

// メインループ.  
while (!window.ShouldClose()) {  
glClearColor(0.1f, 0.3f, 0.5f, 1.0f);  
 glClear(GL\_COLOR\_BUFFER\_BIT | GL\_DEPTH\_BUFFER\_BIT);  
**+  
+** glUseProgram(shaderProgram);  
**+** glBindVertexArray(vao);  
**+** glDrawArrays(GL\_TRIANGLES, 0, sizeof(vertices)/sizeof(vertices[0]));  
**+**  
 　window.SwapBuffers();  
 }

glUseProgram関数は、描画に使用するプログラム・オブジェクトを設定する関数です。  
glBindVertexArray関数は、描画に使用するVAOを設定する関数です。なお、VBOはVAOによって自動的に設定されます(VAO作成時にVBOを指定していたことを思い出してください)。  
最後のglDrawArrays関数は、指定されたオブジェクトやデータを使って図形を描画します。  
この関数の最初の引数は描画する形状の種類です。1章で説明したように、OpenGLは頂点データを様々な方法で描画できます。ここではGL\_TRIANGLESを指定して、三角形プリミティブを描画させています。2つめの引数は、描画を開始する頂点データのインデックスです。ここでは0を指定して、最初の頂点データから描画するように指定しています。3つめの引数は、描画する頂点データの数です。ここではsizeof演算子を利用して、vertices変数の要素数を計算し、それを指定しています。

**[補足]** 「sizeof(配列名)/sizeof(配列名[0])」というのは、C/C++で配列の大きさを取得する慣用的なコードです。ここでは直接書いていますが、一般的には  
#define ARRAY\_SIZE(a) (sizeof(a)/sizeof((a)[0]))  
のようなマクロを定義してそれを使います。

# オブジェクトの削除

作成したオブジェクトは、プログラムを終了する前に削除しなければなりません。  
メインループの直後に、次のコードを追加してください。

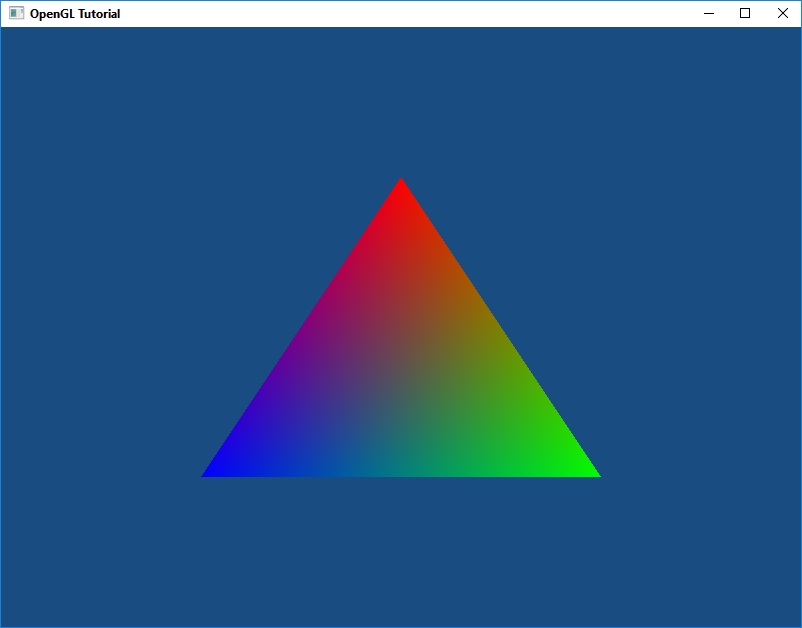
glDrawArrays(GL\_TRIANGLES, 0, sizeof(vertices)/sizeof(vertices[0]));  
  
 　 window.SwapBuffers();  
 }  
  
**+** glDeleteProgram(shaderProgram);  
**+** glDeleteVertexArrays(1, &vao);  
  
 return 0;  
 }

作成したときとは逆の順番でオブジェクトを削除していきます。なぜなら、後から作られたオブジェクトは、先に作られたオブジェクトを利用している可能性があり、その場合、先に作られたオブジェクトが存在していないと適切に終了処理を行えない可能性があるからです。

**[補足]** 「作成したときとは逆の順番でオブジェクトを削除する」ことは、プログラムを作成する上で一般的なルールです。依存関係の有無にかかわらず、このルールに従うことをお薦めします。

glDeleteProgram関数は、既に説明したようにプログラムオブジェクトを削除する関数です。  
glDeletVertexArrays関数は、VAOを削除する関数です。作成時と同様に、削除する個数を指定できます。

お疲れ様でした。これで今回の実装は完了です！  
ビルドして実行してみてください。次のような画面が表示されたでしょうか。



**[課題01]** 頂点データを変更して、三角形が黄色、紫色、水色で描かれるようにしてください。

**[課題02]** 頂点データを変更して、三角形を上下逆さまにしてください。

**[課題03]** 頂点データを追加して、三角形を3つ表示してください。

**[課題04]** 1章の図を参考にglDrawArrays関数の最初の引数を変更し、ビルドして実行してください。7種類のプリミティブがどのように表示されるかを確認できたら、GL\_TRIANGLESに戻してください。