# 君はゲームプログラマーになりたいか？<br>目指してるのは楽な道のりじゃないぜ

くらげ～

2018 年 2 月 28 日

# 1　君の名は？

この上の君というの著作者だから心配する必要はない！

## 1.1　ん？じゃあ君は宇宙人か？

1I の JOKEN の中で生きている (執筆時) くらげ～です。ん？人間なのにくらげ～とかみずみずしい名前しあがって、さてはてめえ宇宙人だな？いや、別にただの一般人です。ま、プログラミングは 8 年、セキュリティは 3 年、人工知能は 3 ヵ月やってます。まあ、中身を詳しく言うときりがないので割愛しますが、何年やってきたかはどうでもいいんだよ！やる気さえあればなんでもできる！ (白目) 僕のモットーは広く深く、やり方は基礎を完全に理解してから、一気 に深く掘り下げていきます。それで勉強の効率化を図っています。僕の場合は基礎が定着 していないと応用をしないので、最初の方は成長が遅いです。小学校の時がそうやった。

僕は別にゲームクリエイターになるつもりは現時点ではないが、スキルを身に着ける上 で学習しながら OpenGL や Vulkan でオブジェクト配置したり GLSL(OpenGL Shading Language) で minecraft の影 mod(現時点では影処理しか書けない) を作ったりして楽し んでいる。ちなみに、影処理は Shadow mapping って調べたらでてくるぜ！その他にも いろいろしているが... まあ、そんなことはどうでもいい。ゲームプログラマーというものは知っているか？

# 2　え？ゲームプログラマー？え、めっちゃかっこええやん

この世には Unity や UNREAL ENGINE 4 など様々なゲームエンジンがあって、そ れで物理現象などを考えずにクリックで重力を起こせたり, さらにはオブジェクトを配置 したりできる。これらのゲームエンジンだけでゲームを作成することができる。ただ、君 はそこで満足するのか？ん？そこで満足していいと思ってるのか？

## 2.1　基礎を知らないままゲームを作るのか？

ゲームを作成するだけなら Unity や UNREAL ENGINE 4 の勉強をすればいい。で も、それはゲームクリエイターであってゲームプログラマーではない。ただ、これらの ゲームエンジンを使ってはいけないとは誰も言っていない。ゲームを作成するうえで時間 短縮、楽することは重要なことである。俺はゲームエンジンを使うことは推奨する人だ。 だが、どのように描画されているのか知らないままゲームエンジンでゲームを作成してい ると、ゲーム作成の基礎が分かっていないことになるよな？な？だから、ちょっとでも足 を突っ込んでみて、ゲームエンジンの中身までわからなくても、その初歩的なことでもわ かればそれでも十分だよな！ (この資料に全部書くとページが何枚あっても足りないのは 秘密)

# 3　本題に入る前に 2 次元/3 次元グラフィックスプログラミ ング API の紹介をするぜ

まずは先ほど出てきた OpenGL からだ。こやつはクロノス・グループとかいうちょっとかっこいい名前をしているところが開発元だ。名前がかっこいいなみにやってることも侮れない。ちなみにこやつは CGAPI(コ

ンピューターグラフィックスアプリケーションプログラミングインターフェイス) である。

まあ、わけわからない言葉がたくさん出てきたが、不安がる必要はない。最初から分かる天才なんてこの世に存在しねえ。OpenGL ってのは Java みたいに対応 OS が幅広いのが特徴だ。ただ、C++ みたいに簡単に学習できるような言語ではない少し複雑な API である。つい最近までは 3D ゲームの作成などはこの API が主流であった (今も主流かも知 れない)。C++ と併用するのが多い。

次は DirectX というものだ。僕はこの API は少ししか触ったことがないので具体的にはいえないが。唯一いえることは、Windows のみでしか動作しないことだ。Windows だけで動作させる場合ならこの選択肢もありだろう。

その次に Vulkan というものだ。こいつもクロノス・グループによって、開発されている。OpenGL の上位互換といってもいい API だ。ただ OpenGL より難しいが、低レベルな API によって軽量な処理が可能になっている。今後主流になっていくであろう API であろう。

その他にもいろいろあるかもしれないが、俺は知らない。

## 4　おい、さっさと本題いってくれよ

まあまあ、待て待て。今回紹介するのは OpenGL だが。同じ開発元だよな？じゃあ、何が違うんだ？

A:同じ開発元だからなんだって言いたいんだ。

me:いや... 別に何もないです....

## 5　OpenGL と Vulkan の違いってなんだよ？

何が違うかといわれても一見はほとんど違いはない。ただ簡単に言うと Vulkan の方が多くのベンダーをサポートして効率的に GPU のリソースを使うことができる。何が Vulkan の方が難しいか言うと、OpenGL ではドライバによってメモリマネジメントや、スレッドマネジメント、コマンドバッファの生成や更新を自動でするが、Vulkan はそれをアプリケーション側でしないといけない。そこが難しい理由だ。ただ、勉強できないほど難しいものでもないので気になった人は勉強してみよう。そして Vulkan で描画したものは OpenGL 内で使うことができるぞ！ただ、今回は紙媒体ということもあり、Vulkan のソースコードは長いので、資料に書くことができません！皆さんも読む気なくしますよね？

# 6 OpenGL

まずは OpenGL で立方体を描画したいのだが、そもそも OpenGL ってなんなの？？

## 6.1 OpenGL とは

OpenGL とは, 先ほど出たようにクロノス・グループ (khronos Group) が策定、開発していて、2/3 次元 CGAPI である。

## 6.2 OpenGL とは何者だ！

ここ最近、ゲームなどで 3D ゲームなどが増えてきました。これは三次元コンピュータグラフィックス (3DComputerGraphics、3DCG) の技術を使っています。”Virtual Youtuber”などは 3DCG などを使って、キャラクターの外観を作成し、web カメラや今は生産終了されている Kinect などでモーションキャプチャーを行って、キャラクターを動かしたりしています。いま一番身近にある技術なのではないのかなと思っています。ただ、”Virtual Youtuber”や 3D ゲームが生まれる前からも、OpenGL は広く使われていました。一つの大きな要因として OpenGL はプラットフォームに依存しないからです。プラットフォームに依存しないことにより Linux の X Window System に組み込まれていたり、Windows と macOS ともに OpenGL が導入されていて、さらに iOS や Android にも組込みシステム向けの OpenGL ES などが導入されています。Nintendo Switch も Vulkan や OpenGL がサポートされています。

## 6.3 OpenGL を動かすためには

OpenGL は「プラットフォームに依存しない CGAPI(コンピュータグラフィック API)」であるが、やはりプラットフォームごとに少し工夫する必要があります。また、この工夫はちょっとめんどくさいのでそれを気にしなくて簡単に使うことができるキットがこの世には存在する。

## 6.4 GLUT

GLUT とは先ほどのキットの一種である。OpenGL の初期のころに作られたもので、僕も最初はこれを使って勉強していた。しかし、この GLUT には欠点があり、長期間メンテナンスされていないし、macOS ver10.9 からは GLUT の使用が非推奨となっています。このキットは Unix 系 OS で使うことができるぞ！

## 6.5 GLFW

GLUT に代わるものして Qt(キュート) などたくさんありますが、やはり手軽に学習できるという点で GLFW をお勧めします。GLFW はクロスプラットフォーム (マルチプラットフォーム) で、C 言語で書かれているが、他の言語で使用することができるバインディングがある。GLFW には OpenGL のバージョンやプロファイルが指定できたりマルチモニタに対応していたり、いろいろ機能があるが説明しているときりがないので今回は省く。さらに、この offline では GLFW で描画するぜ！ちなみにゲームを OepnGL と C 言語系で作るときは SDL の方が多機能でゲームプログラマー御用達なので、そっちを使うことをお勧めする。

## 6.6　立方体を描画するぞ！

OpenGL を使って図形を表示するソースコードを見たりプログラムを書いたりするにあたって、幾何学の存在は逃せません。とくに三次元空間のオブジェクトを扱うには、幾何学の知識が必要になります。

まずはソースコードを見てみよう！

```
1:  #define GLFW_INCLUDE_GLU
2:  #include <iostream>
3:  #include <GLFW/glfw3.h>
4:  #define width 1280
5:  #define height 960
6:
7:  static GLFWwindow*  window;
8:
9:  static const GLdouble CubeVertex[][3] = {
10:         { 0.0, 0.0, 0.0 },
11:         { 1.0, 0.0, 0.0 },
12:         { 1.0, 1.0, 0.0 },
13:         { 0.0, 1.0, 0.0 },
14:         { 0.0, 0.0, 1.0 },
15:         { 1.0, 0.0, 1.0 },
16:         { 1.0, 1.0, 1.0 },
17:         { 0.0, 1.0, 1.0 }
18: };
19:
20: static const GLint CubeFace[][4] = {
21:         { 0, 1, 2, 3 },
22:         { 1, 5, 6, 2 },
23:         { 5, 4, 7, 6 },
24:         { 4, 0, 3, 7 },
25:         { 4, 5, 1, 0 },
26:         { 3, 2, 6, 7 }
27: };
28:
29: static const GLdouble CubeNormal[][3] = {
30:         { 0.0, 0.0,-1.0 },
31:         { 1.0, 0.0, 0.0 },
32:         { 0.0, 0.0, 1.0 },
33:         { -1.0, 0.0, 0.0 },
34:         { 0.0,-1.0, 0.0 },
```

```
35:         { 0.0, 1.0, 0.0 }
36: };
37:
38: static const GLfloat CubeMaterial[] = { 0.8f, 0.35f, 0.4f, 1.0f };
39: static const GLfloat Lightpos0[] = { 0.0f, 3.0f, 5.0f, 1.0f };
40:
41:
42: static void DrawCube()
43: {
44:         glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT_AND_DIFFUSE,
               CubeMaterial);
45:
46:         glBegin(GL_QUADS);
47:         for (size_t i = 0; i < 6; ++i)
48:         {
49:                 glNormal3dv(CubeNormal[i]);
50:                 for (size_t j = 0; j < 4; ++j)
51:                 {
52:                         glVertex3dv(CubeVertex[CubeFace[i][j]]);
53:                 }
54:         }
55:         glEnd();
56: }
57:
58: int main()
59: {
60:
61:         if (glfwInit() == GL_FALSE)
62:         {
63:         std::cerr << "Error␣initilize␣GLFW" << std::endl;
64:         exit(EXIT_FAILURE);
65:         return 1;
66:         }
67:
68:         glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 4);
69:         glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 6);
70:         glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT, GL_TRUE);
71:
72:         window = glfwCreateWindow(width, height, "SampleCube", NULL, NULL);
73:
```

```
74:         if (window == NULL)
75:         {
76:                 std::cerr << "    ErrorcreateGLFWwindow." << std::endl;
77:                 glfwTerminate();
78:                 exit(EXIT_FAILURE);
79:                 return 1;
80:         }
81:
82:         glfwMakeContextCurrent(window);
83:         glEnable(GL_DEPTH_TEST);
84:         glEnable(GL_LIGHTING);
85:         glEnable(GL_LIGHT0);
86:         glEnable(GL_CULL_FACE);
87:         glCullFace(GL_FRONT);
88:         // #82ffe6 is HTML color
89:         glClearColor(0.509f, 1.0f, 0.901f, 1.0f);
90:
91:         while (glfwWindowShouldClose(window) == GL_FALSE)
92:         {
93:
94:                 glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
95:                 glLoadIdentity();
96:
97:
98:                 int now_width, now_height;
99:                 glfwGetFramebufferSize(window, &now_width, &now_height);
100:                glViewport(0, 0, now_width, now_height);
101:                gluPerspective(30.0, (double)now_width / (double)now_height
                        , 1.0, 100.0);
102:                glTranslated(0.0, 0.0, -2.0);
103:                gluLookAt(3.0, 5.0, 4.5, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
104:                glLightfv(GL_LIGHT0, GL_POSITION, Lightpos0);
105:
106:                DrawCube();
107:
108:                glfwSwapBuffers(window);
109:                glfwPollEvents();
110:        }
111:
112:        glfwTerminate();
```

```
113:         return 0;
114: }
```

この上記のコードは OpenGL(GLFW) における立方体の描画プログラムだ。早速見ていこう！

まず#define だ#define は C 言語をベースとするプリプロセッサ命令で、マクロ処理を担っている。ちなみに#define GLFW_INCLUDE_GLU は GLFW3.0 以降にデフォルトで GLU ヘッダーが含まれなくなったので、インクルードしています。#include は#define と一緒で C 言語をベースとするプリプロセッサ命令で。簡単に言うと別のソースファイルをまとめて処理してくれる。#include<GLFW/glfw3.h>が今回使う GLFW のソースファイルだ。これがないと GLFW を使うことができない。#define width 1280 は横幅 #define height 960 は高さをマクロしている。ここでは初期の window size として扱っている。

## 6.7 ポリゴンの表示

さあここから本場所に入っていく！ C++ では int main(引数) ※引数が void は省略可; がプログラム実行時に一番最初に呼び出される関数です。まず最初にすべきことは GLFW を初期化しないといけません。GLFW を初期化するのは以下のコードで初期化することができます

```
if( !glfwInit() ) return 1;
```

僕のソースコードでは

```
if (glfwInit() == GL_FALSE)
{
        std::cerr << "Error␣initilize␣GLFW" << std::endl;
        exit(EXIT_FAILURE);
        return 1;
}
```

このようになっていますが、あまり変わりはありません。!と== false は同じ意味で、また false と GL_FALSE も意味合い的にはそこまで違いはありません。また、

```
std::cerr << "Error␣initilize␣GLFW" << std::endl;
```

と書いているのは、標準エラー出力を出した方がエラー内容が分かりやすいので、デバッグ時などに有用です。初期化に失敗した場合は

```
Error initilize GLFW
```

と表示させ、初期化に失敗したとを知らせます。

```
return 1;
```

で 1 を返して、プログラムが異常終了をします。
ちなみに余談なんですが

Listing 1　sample.cpp

```
int main()
{
        return 0;
}
```

というソースコードがあって以下のコマンドを実行します

Linux 系の OS の場合は

```
$ g++ sample.cpp
$ ./a.out && echo "SUCCESS"
```

Windows の場合は

```
$ g++ sample.cpp
$ ./a.exe && echo "SUCCESS"
```

そうすると出力はどうなるでしょうか？では、次に下記のソースコードで同じコマンドを実行してみましょう。

Listing 2　sample.cpp

```
int main()
{
        return 1;
}
```

さて、どうなるでしょうか。一つ目のプログラムでコマンドを実行した場合、出力は

```
SUCCESS
```

となりますが、2 つ目プログラムでコマンドを実行した場合、出力はありません。.... で、何が言いたいの？まあ、return の有用性を知ってもらいたかっただけです。まあ、C++ を普通に勉強するならいらない知識かもしれない。

```
glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 4);
glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 6);
glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT, GL_TRUE);
```

この 3 つは別に必要はないですが、window 作成時に、コンテキストやスワップチェインなどの情報を指定します。各環境によって動作する OpenGL のバージョンが変わってくる可能性があるので、

```
glfwDefaultWindowHint();
```

を書くといいでしょう、これはデフォルトの情報を指定してくれます。

```
window = glfwCreateWindow(width, height, "SampleCube", NULL, NULL);
```

では window を作成する上で情報を指定します。データ型としては

```
glfwCreateWindow(int, int,const char, GLFWmonitor , GLFWwindow );
```

で構成されていて、第 1 引数は横幅、第 2 引数は高さ、第 3 引数は表示させるタイトル、第 4 引数は表示させるモニター、第 5 引数はリソースを共有するコンテキストです。また、GLFWwindow 型のポインタを返します。ちなみにフルスクリーンにする方法があって、

```
glfwCreateWindow(width, height, glfwGetPrimaryMonitor(), NULL );
```

glfwGetPrimaryMonitor() で 1 枚目のモニターを取得して、第 4 引数に渡しています。また、2 枚目や 3 枚目のモニターをフルスクリーンにしたい場合は

```
glfwCreateWindow(width, height, glfwGetMonitors(2), NULL );
```

で、フルスクリーンにさせることができます。これも GLFW の初期化同様に、window の生成に失敗したら 1 を返す必要があります。僕のソースコードでは

```
if (window == NULL)
{
        std::cerr << "Error␣create␣GLFW␣window." << std::endl;
        glfwTerminate();
        exit(EXIT_FAILURE);
        return 1;
}
```

となっていますが

```
if (!window)
{
        glfwTerminate();
        return 1;
}
```

で OK です。なぜ glfwTerminate(); が必要なのかというと、glfwInit 関数を呼び出した後は必ず呼び出す必要があるからです。そして、glfwMakeContextCurrent を使ってカレントにします。

```
glfwMakeContextCurrent(window);
```

OpenGL の API を使う前には、必ず OpenGL コンテキストを取得しなければならないのでこのコードは必須です。他のコンテキストをカレントするか、もしくはウィンドウが破棄 (主に終了動作) されるまで、カレントコンテキストはこの状態を維持し続けます。
OpenGL の API を使えるようになったので、まずは OpenGL を初期化していきます。まずはデプスバッファの有効化です。

```
glEnable(GL_DEPTH_TEST);
```

で有効化できるのですが.... デプスバッファって何？

デプスバッファとは Z バッファとも呼ばれますが、深度情報を用いて物体の描画処理を省略して高速化するための技術です。3DCG では、平面に描画するためには空間内に配置された物体の頂点や境界線、表面のうち見えている部分だけを描画しなければなりません。手前にある他の物体などに隠されて見えてない部分の描画を省略する必要があるから有効にしています。今回は 1 つの物体しか描画していませんが、2 つ 3 つ描画するときに必要なものです。

その次に、ライティング処理を有効化します。

```
glEnable(GL_LIGHTING);
```

これで頂点カラーに証明パラメーターを使用するようになりました。そして光源を有効化します。

```
glEnable(GL_LIGHT0);
```

でライト 0 が有効化されました。

```
glEnable(GL_CULL_FACE)
```

描画処理を軽量化します。これではポリゴンの片面表示を有効化しています
表面だけ描きたいのなら

```
glCullFace(GL_BACK);
```

裏面だけを描きたいのなら

```
glCullFace(GL_FRONT);
```

両面を描くときは、

```
glDisable(GL_CULL_FACE);
```

を一応書いた方がいいでしょう。そして、glClearColor() で glViewPort で指定した範囲を塗りつぶします。ちなみに glClearColor は RGBA で指定できるのですが、少し気を付けなければならない点があります。256 段階の数字を扱うのではなく係数で扱います。すなわち、1.0 が 255 で 0.0 が 0 となります。なので#82ffe6 は rbga(130,255,230,1.0) になって係数にするには、$y = \frac{x}{255}$ に当てはめて計算します。すると

```
glClearColor(0.509f, 1.0f, 0.901f, 1.0f);
```

こう書けます！！ GLFW では描画する際に while でループを書かなければなりません。

```
while(!glfwWindowShouldClose())
```

でできます。glClear() を用いれば、バッファを初期化できます。

```
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

GL_COLOR_BUFFER_BIT は色バッファの初期化、GL_DEPTH_BUFFER_BIT はデプスバッファ (Z バッファ) を初期化します。

```
glLoadIdentity();
```

をコールして変換行列を単位行列で初期化します。これで描画毎に座標系をリセットしています。

```
int now_width, now_height;
glfwGetFramebufferSize(window, &now_width, &now_height);
```

now_width と now_height の int 型を宣言して、glfwGetFramebufferSize() でフレームバッファサイズを取得します。そして、

```
glViewport(0, 0, now_width, now_height);
```

で glClearColor で設定された色を glViewport の範囲を塗りつぶします。

```
gluPerspective(30.0, (double)now_width / (double)now_height, 1.0, 100.0);
```

gluPerspective() では視野角、奥行きの最大、最小距離を描画します。この範囲外は例外を除いて基本的には描画されません。第 1 引数には縦の視野角 (度数指定)、第 2 引数には縦に対する横方向の視野角、第 3 引数には一番近い Z 座標を指定して、第 4 引数には一番遠い Z 座標を指定します

```
glTranslated(0.0, 0.0, -2.0);
```

glTranslated() では x,y,z に移動量を渡すと、平行移動します。この場合 z 軸方向に-2.0 平行移動します。そして、カメラの位置と姿勢を設定します

```
gluLookAt(3.0, 5.0, 4.5, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
```

左に引数から順にカメラの X 座標,Y 座標,Z 座標, 注視点の X 座標, 注視点の Y 座標, 注視点の Z 座標、カメラの上方向が X 軸に対してどこか, カメラの上方向が Y 軸に対してどこか, カメラの上方向が Z 軸に対してどこかを指定します。第 7,8,9 引数はベクトルを指定します。

```
glLightfv(GL_LIGHT0, GL_POSITION, Lightpos0);
```

第 2 引数で GL_POSITION が定数とされているので、光源を設置します。Lightpos0 の座標に設置します、その後に DrawCube() を呼び出します。

## 6.8 DrawCube()

まずはオブジェクトの質を設定します

```
glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT_AND_DIFFUSE, CubeMaterial);
```

第 1 引数には質感を適用する面を指定します。GL_FRONT_AND_BACK は両面を指定しています。そして第 2 引数では反射光を指定します、GL_AMBIENT_AND_DIFFUSE になっているので、拡散反射係数と鏡面反射係数の両方を保持しています。次に描く処理を書きます。glBegin() から始まって glEnd() の間を処理します。その次に glNormal3dv() で法線ベクトルの設定を行います。オブジェクトが都のように光が反射するかどうかはこれによって決まります。最後に glVertex3dv() で配列のポインタを与えて頂点を指定します。for とかでソースコードの解読が困難になっていますが、可視化すると

```
CubeNormal: { 0.0 ,0.0 ,-1.0 }

------------------------------
CubeVertex: { 0.0 ,0.0 ,0.0 }
CubeVertex: { 1.0 ,0.0 ,0.0 }
CubeVertex: { 1.0 ,1.0 ,0.0 }
CubeVertex: { 0.0 ,1.0 ,0.0 }
------------------------------

CubeNormal: { 1.0 ,0.0 ,0.0 }

------------------------------
CubeVertex: { 1.0 ,0.0 ,0.0 }
CubeVertex: { 1.0 ,0.0 ,1.0 }
CubeVertex: { 1.0 ,1.0 ,1.0 }
CubeVertex: { 1.0 ,1.0 ,0.0 }
------------------------------

CubeNormal: { 0.0 ,0.0 ,1.0 }

------------------------------
CubeVertex: { 1.0 ,0.0 ,1.0 }
CubeVertex: { 0.0 ,0.0 ,1.0 }
CubeVertex: { 0.0 ,1.0 ,1.0 }
CubeVertex: { 1.0 ,1.0 ,1.0 }
------------------------------

CubeNormal: { -1.0 ,0.0 ,0.0 }

------------------------------
CubeVertex: { 0.0 ,0.0 ,1.0 }
CubeVertex: { 0.0 ,0.0 ,0.0 }
CubeVertex: { 0.0 ,1.0 ,0.0 }
CubeVertex: { 0.0 ,1.0 ,1.0 }
------------------------------

CubeNormal: { 0.0 ,-1.0 ,0.0 }

------------------------------
```

```
CubeVertex: { 0.0 ,0.0 ,1.0 }
CubeVertex: { 1.0 ,0.0 ,1.0 }
CubeVertex: { 1.0 ,0.0 ,0.0 }
CubeVertex: { 0.0 ,0.0 ,0.0 }
-----------------------------

CubeNormal: { 0.0 ,1.0 ,0.0 }

-----------------------------
CubeVertex: { 0.0 ,1.0 ,0.0 }
CubeVertex: { 1.0 ,1.0 ,0.0 }
CubeVertex: { 1.0 ,1.0 ,1.0 }
CubeVertex: { 0.0 ,1.0 ,1.0 }
-----------------------------
```

glNormal3dv() には CubeNormal:の後の配列を、glVertex3dv() には CubeVertex：の後の配列を与えて頂点を指定しています。

```
glfwSwapBuffers(window);
glfwPollEvents();
```

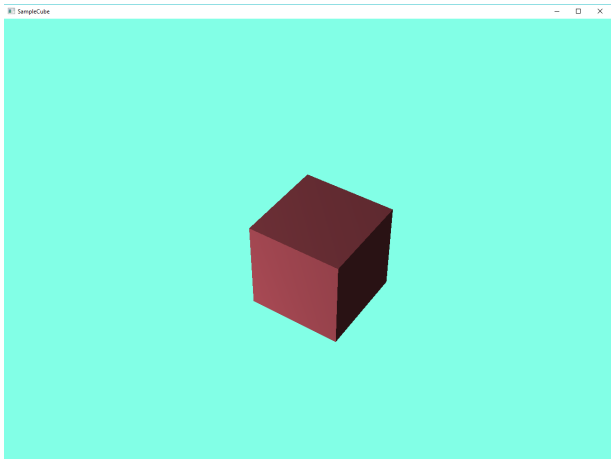
そして glfwSwapBuffers() と glfwPollEvents() で描画用バッファをスワップします。

## 6.9 GLFW の終了

glfwWindowShouldClose(window) が TRUE になるとループが終わるため、GLFW の終了処理が必要です。

```
glfwTerminate();
return 0;
```

glfwTerminate はプログラムの最後に呼び出さないといけないおまじないです。そして return 0; でプログラムが正常終了したことを伝えます。ちなみに今回作った図形はこちらです




## 7　さいごに

今回は offline で Vulkan とか OpenGL で立方体を回転させてみたり、変形させてみたりはしていませんが。OpenGL や vulkan ではそんなこともできます。僕は楽しいからプログラミングをやっています。プログラミングはたのしいですよ？！ joken ではプログラミングを学ぶことができたりします！ (最初のような感じで書こうと思ったが... 案外難しかった)

## 参考文献

[1] ”GLFW Documentation”
http://www.glfw.org/documentation.html
http://www.glfw.org/docs/latest/index.html
http://www.glfw.org/docs/latest/pages.html
http://www.glfw.org/docs/latest/modules.html
[2] ”OpenGL 4.6 API Reference Guide”
https://www.khronos.org/files/opengl46-quick-reference-card.pdf
[3] ”GLUT による「手抜き」OpenGL 入門”
・座標データの参考にしました。
https://tokoik.github.io/opengl/libglut.html#8