

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

くらげ～

2018 年 2 月 23 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？

II 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 ヶ月やっています。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 調べてたのでくるぜ！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も侮れない。ちなみにこやつは CGAPI(コ

ンピュータ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 によって軽量の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... 別に何もありません....

## 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な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 などでも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、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 とともに OpenGL が導入されていて、さらに iOS や Android にも組み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いで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 に代わるものとして Qt(キョウト) 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、他の言語で使う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 OpenGL 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 LightColor[] = { 0.2f, 0.2f, 0.8f, 1.0f };
40:     static const GLfloat Lightpos0[] = { 0.0f, 3.0f, 5.0f, 1.0f };
41:     static const GLfloat Lightpos1[] = { 5.0f, 3.0f, 0.0f, 1.0f };
42:
43:
44:     static void DrawCube()
45:     {
46:         glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT_AND_DIFFUSE,
47:                     CubeMaterial);
48:
49:         glBegin(GL_QUADS);
50:         for (size_t i = 0; i < 6; ++i)
51:         {
52:             glNormal3dv(CubeNormal[i]);
53:             for (size_t j = 0; j < 4; ++j)
54:             {
55:                 glVertex3dv(CubeVertex[CubeFace[i][j]]);
56:             }
57:         }
58:         glEnd();
59:     }
60:
61:     int main()
62:     {
63:         if (glfwInit() == GL_FALSE)
64:         {
65:             std::cerr << "Error_initialize_Glfw" << std::endl;
66:             exit(EXIT_FAILURE);
67:             return 1;
68:         }
69:
70:         glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 4);
71:         glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 6);
72:         glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT, GL_TRUE);
73:

```

```

74:     window = glfwCreateWindow(width, height, "SampleCube", NULL, NULL);
75:
76:     if (window == NULL)
77:     {
78:         std::cerr << "      ErrorcreateGLFWwindow." << std::endl;
79:         glfwTerminate();
80:         exit(EXIT_FAILURE);
81:         return 1;
82:     }
83:
84:     glfwMakeContextCurrent(window);
85:     glEnable(GL_DEPTH_TEST);
86:     glEnable(GL_LIGHTING);
87:     glEnable(GL_LIGHT0);
88:     glEnable(GL_LIGHT1);
89:     glEnable(GL_CULL_FACE);
90:     glCullFace(GL_FRONT);
91:     glLightfv(GL_LIGHT1, GL_SPECULAR, LightColor);
92:     glLightfv(GL_LIGHT1, GL_DIFFUSE, LightColor);
93:     // #82ffe6 is HTML color
94:     glClearColor(0.509f, 1.0f, 0.901f, 1.0f);
95:
96:     while (glfwWindowShouldClose(window) == GL_FALSE)
97:     {
98:
99:         glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
100:        glLoadIdentity();
101:
102:
103:        int now_width, now_height;
104:        glfwGetFramebufferSize(window, &now_width, &now_height);
105:        glViewport(0, 0, now_width, now_height);
106:        gluPerspective(30.0, (double)now_width / (double)now_height
            , 1.0, 100.0);
107:        glTranslated(0.0, 0.0, -2.0);
108:        gluLookAt(3.0, 5.0, 4.5, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
109:        glLightfv(GL_LIGHT0, GL_POSITION, Lightpos0);
110:        glLightfv(GL_LIGHT1, GL_POSITION, Lightpos1);
111:
112:        DrawCube();

```

```

113:
114:         glfwSwapBuffers(window);
115:         glfwPollEvents();
116:     }
117:
118:     glfwTerminate();
119:     return 0;
120: }

```

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` と一緒に 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_initialize 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_initialize 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 initialize 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のでしょうか？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、何が言いたいのか？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知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ー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。ちなみにフルスクリーンにする方法は 2 つほどあって

モニタの横幅と高さを引数にする方法

```
int width, height;
GLFWmonitor* monitor = glfwGetPrimaryMonitor();
glfwGetMonitorPhysicalSize(monitor, width, height);
window = glfwCreateWindow(width, height, "SampleCube", NULL, NULL);
//マクロは外してください
```

glfwGetPrimaryMonitor() で 1 枚目のモニターを取得して、glfwGetMonitorPhysicalSize() でモニタの物理サイズを取得します。