**ライブラリー**:

OpenGL

glm

GLFW

GLAD

stb\_image

期間: 理解してから作成するまでは１ヶ月。それぞれの実行ファイルは1日ぐらいです。

**解説:**

OpenGLとは、ハードウェアの持つ3次元コンピュータグラフィックス（3DCG）関連の演算機能をソフトウェアから呼び出すための規約を定めたインターフェース（API）標準の一つ。もともと旧Silicon Graphics社（SGI）が自社ソフトウェア向けに開発していたIRIS GLをオープン化したもので、業界団体のクロノス・グループ（Khronos Group）が仕様の標準化を行っている

http://e-words.jp/w/OpenGL.html

OpenGLを使うのに別のライブラリーが必要です。このプロジェクトはGLFWでウィンドウズや描画などを行うライブラリーとGLADでOSやウィンドウズのバージョンを簡単に指定して使えるライブラリーを使います。

始める前にGLFWとGLADを初期化しないといけません。

glfwInit();

glfwWindowHint(GLFW\_CONTEXT\_VERSION\_MAJOR, 3);

glfwWindowHint(GLFW\_CONTEXT\_VERSION\_MINOR, 3);

glfwWindowHint(GLFW\_OPENGL\_PROFILE, GLFW\_OPENGL\_CORE\_PROFILE);

glfwInit()でGLFWを初期化してから、glfwWindowHint()のメソッドでOpenGLのバージョンを指定します。

そして、ウィンドウを初期化します。

//Create window

window = glfwCreateWindow(screenwidth, screenheight, "LearningOpenGL", NULL, NULL);

if (window == NULL)

{

std::cout << "Failed to create GLFW window" << std::endl;

glfwTerminate();

return false;

}

glfwMakeContextCurrent(window);

glfwSetFramebufferSizeCallback(window, framebuffer\_size\_callback);

glfwCreateWindow()、メソッド名の通りにウィンドウを作るメソッドです。GLFW\* windowの変数に入れて、エラーが起これば(window == NULL) cout(出力変数)でエラー報告を表示して、glfwTerminate()でglfwのデータを消します。glfwMakeContextCurrent()でレンダリングコンテキスト、ウィンドウことに保持されます。glfwSetFramebufferSizeCallback()はウィンドウのサイズを大きくしたりする関数です。

if (!gladLoadGLLoader((GLADloadproc)glfwGetProcAddress))

{

std::cout << "Failed to Initialize GLAD" << std::endl;

return false;

}

gladライブラリーですべてのglfwメソッドを初期化します。Falseの値が返ったら、エラーメッセージを表示します。glfwGetProcAddressは使用しているOSのバージョンを返してくれます。

//Setting Viewport

glViewport(0, 0, screenwidth, screenheight);

glViewPort()でウィンドウサイズを指定します。

**while**(!glfwWindowShouldClose(window))

{

glfwSwapBuffers(window);

glfwPollEvents();

}

ウィンドウを保持たいため、while文でもしglfwWindowClose()ウィンドウが閉じられなければ、回します。while文の中にglfwSwapBuffers()のメソッドでカラーバッファを映していきます。glfwPollEvents()の関数はウィンドウの状態(マウスやキーボードなど)を取ります。

すべてを組み合わせたら、Initializeの関数

bool Initialize(unsigned int screenwidth, unsigned int screenheight)

{

//Initialize GLFW

glfwInit();

//Setting GLFW version (3)

glfwWindowHint(GLFW\_CONTEXT\_VERSION\_MAJOR, 3);

glfwWindowHint(GLFW\_CONTEXT\_VERSION\_MINOR, 3);

glfwWindowHint(GLFW\_OPENGL\_PROFILE, GLFW\_OPENGL\_CORE\_PROFILE);

//Create window

window = glfwCreateWindow(screenwidth, screenheight, "LearningOpenGL", NULL, NULL);

if (window == NULL)

{

std::cout << "Failed to create GLFW window" << std::endl;

glfwTerminate();

return false;

}

glfwMakeContextCurrent(window);

glfwSetFramebufferSizeCallback(window, framebuffer\_size\_callback);

if (!gladLoadGLLoader((GLADloadproc)glfwGetProcAddress))

{

std::cout << "Failed to Initialize GLAD" << std::endl;

return false;

}

//Setting Viewport

glViewport(0, 0, screenwidth, screenheight);

return true;

}

**while**(!glfwWindowShouldClose(window))

{

glfwSwapBuffers(window);

glfwPollEvents();

}

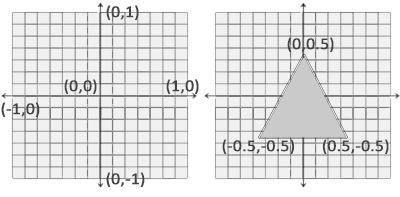
glfwTerminate();

**Vertex and Fragment**



<https://learnopengl.com/Getting-started/Hello-Triangle>

上記の図はグラフィックパイプラインを表しています。まずは頂点変換をします。そして、Shape Assemblyで形を作って、ジオメトリシェーダーはピクセルシェーダーに渡されるオブジェクト内の頂点の集合を加工します。Rasterizationでフラグメントを作ります。Fragment Shaderには最後のカラーを決めます。最後のTest and Blendingはユーザーにより起動すかどうかでやります。例えばDepth Testing(奥行)。

**GLSL**

https://learnopengl.com/Getting-started/Hello-Triangle

上記の図はOpenGLによる頂点の座標です。Vertex Shaderを説明します。

まずは書きたい形の頂点を決めます。例えば

float vertices[] =

{

-0.5f, -0.5f, 0.0f,

0.5f, -0.5f, 0.0f,

0.0f, 0.5f, 0.0f

};

そして、バッファIDを作ります。

glGenBuffers(1, &VBO);

glBindBuffer(GL\_ARRAY\_BUFFER, VBO);

glBufferData(GL\_ARRAY\_BUFFER, sizeof(vertices), vertices, GL\_STATIC\_DRAW);

OpenGLは描画するのにvertex shaderとfragment shaderが必要です。Vertex と Fragment シェダーの中身はGLSLで書きます。GLSL(OpenGL Shading Language) はOpenGLの主の言語です。つまり、OpenGLはGLSLという言語を使っています。GLSLはC言語と似ています。

例:　Vertex Shader

#version 330 core

layout(location = 0) in vec3 aPos;

layout(location = 1) in vec3 aNormal;

layout(location = 2) in vec2 aTexCoords;

out vec3 FragPos;

out vec3 Normal;

out vec2 TexCoords;

uniform mat4 model;

uniform mat4 view;

uniform mat4 projection;

void main()

{

FragPos = vec3(model \* vec4(aPos,1.0));

Normal = mat3(transpose(inverse(model)))\*aNormal;

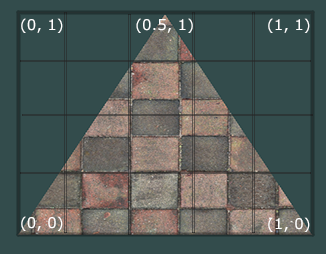
TexCoords = aTexCoords;

gl\_Position = projection \* view \* vec4(FragPos,1.0);

}

キーワードを説明すると layout修飾子 というのはロケーションを指定する修飾子です。一つの頂点配列は描画する頂点だけではなく、テキスチャーの座標なども含めています。だから、layout修飾子で指定すれば便利です。in は入力という意味で、outは出力という意味です。uniformはグローバル変数として使っています。外部ファイルから指定できます。例えば、glUniform1i(glGetUniformLocation(ID, name.c\_str()), (int)value);でintの値を指定できます。

**テキスチャー**



<https://learnopengl.com/Getting-started/Textures>

テキスチャーの座標は左下から(0,0)になります。テキスチャーの座標は普通の座標と違って、(s,t,r)になります。テキスチャーを使うためには頂点シェダーと似ていて、まずはIDを作らないとといけません。

glGenTextures(1, &texture);

glBindTexture(GL\_TEXTURE\_2D, mTextureID);

glTexImage2D(GL\_TEXTURE\_2D, 0, format, mWidth, mHeight, 0, format, GL\_UNSIGNED\_BYTE, image);

glGenerateMipmap(GL\_TEXTURE\_2D);

//set the texture wrapping/filtering options

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_S, GL\_REPEAT);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_T, GL\_REPEAT);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MIN\_FILTER, GL\_LINEAR\_MIPMAP\_LINEAR);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MAG\_FILTER, GL\_LINEAR);

glBindTextureでglGenTexturesでつくったIDをとうろくします。そして、glTexImage2Dでロードした写真でテキスチャーを生成します。glGenerateMipmapはテキスチャーを遠いところ見るときと近いところ見るときのテキスチャーを分けて、生成します。近く見るときは細かく見えますが、遠いところから見るとボヤっとします。glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_S, GL\_REPEAT);とglTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_T, GL\_REPEAT);は座標をラッピングします。そして、ミップマップを使っているため Linearを設定します。glActiveTexture(GL\_TEXTURE0)してから、glBindTexture(GL\_TEXTURE\_2D, mTextureID)でGL\_TEXTURE\_2Dとテクスチャオブジェクトを結合します。

**ライティング**

**光源の色**

**環境光 (ambient light)は**自然光の乱反射です。例えば、太陽や照明器具などです。

void main()

{

float ambientStrength = 0.1;

vec3 ambient = ambientStrength \* lightColor;

vec3 result = ambient \* objectColor;

FragColor = vec4(result, 1.0);

}

計算するには ambientstrength(環境光の強さ) の値を決めて、lightColor(光源の色)とobjectColor(オブジェクトの色)を掛けて、Fragment Shaderで出力します。

**拡散光(diffuse light)は**光が物体とぶつかって反射する。物体から反射した光は拡散光といいます。

//diffuse

vec3 norm = normalize(Normal);

vec3 lightDir = normalize(LightPos - FragPos);

float diff = max(dot(norm,lightDir),0.0);

vec3 diffuse = diff \* lightColor;

計算するには法線ベクトルが必要です。光が法線ベクトルとの角度を求めないといけないからです。光と法線ベクトルとの角度により光の強さが変わります。lightDirは光の方向、光源からオブジェクトまでです。ベクトルのドットプロダクトを計算して、光と法線ベクトルとの角度を計算します。

**鏡面光(specular light)は** 光が物体とぶつかるとき，光はある特定方向に乱反射します。鏡のような完全反射の場合には乱反射せず，スネルの法則にしたがって反射しますが，普通のつや有り物体の場合には，完全反射に準じた方向に乱反射します。

float specularStrength = 0.5;

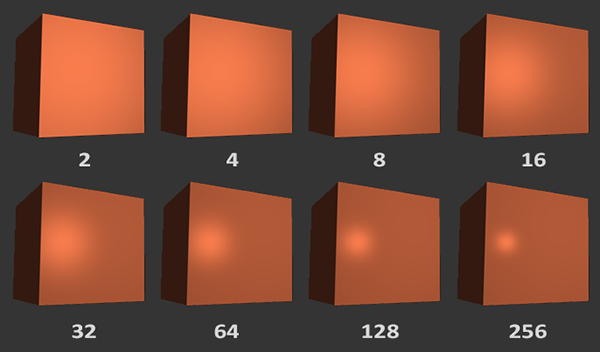
vec3 viewDir = normalize(- FragPos);// the viewer is always at (0,0,0) in view-space, so viewDir is (0,0,0) - Position => -Position

vec3 reflectDir = reflect(-lightDir,norm);

//calculate specular component

float spec = pow(max(dot(viewDir,reflectDir),0.0),32);

vec3 specular = specularStrength \* spec \* lightColor;



鏡面光は反射したベクトルを求めて、計算します。viewDirは見る角度によって光の反射が違うため、見ている方向からオブジェクトまでになります。光と法線ベクトルとの角度で、反射された光を求めて、reflectで反射光を計算します。ハイライトの強さはドットプロダクトを計算し、32乗します。図のように何乗により強さが変わります。

光源の位置・方向

平行光線 (directional light) 光の方向は同じ方向しかありません。距離関係なく、遠くても強さは同じです。例えば太陽。

**vec3 lightDir = normalize(-light.direction);**

ポイントライト(point light) 光源から離れると光が弱くなります。例えば、電球です。

//Calculate attenuation

float distance = length(light.position - fragPos);

float attenuation = 1.0 / (light.constant + light.linear \* distance + light.quadratic \* (distance \* distance));

//combine result

vec3 ambient = light.ambient \* vec3(texture(material.diffuse,TexCoords).rgb);

vec3 diffuse = light.diffuse \* diff \* vec3(texture(material.diffuse,TexCoords).rgb);

vec3 specular = light.specular \* spec \* vec3(texture(material.specular,TexCoords).rgb);

ambient \*= attenuation;

diffuse \*= attenuation;

specular \*= attenuation;

return (ambient + diffuse + specular);

減衰を計算しないといけません。減衰の計算式は

1.0/constant+linear\*distance+quadratic\*distance\*distance

減衰を光の色と掛けて、全部足します。