[OpenGL講義 第04回]

もようをつけよう

# テクスチャマッピング

プリミティブごとに色を付けることができれば、理論上はあらゆる物体を表現できます。  
しかし、物体の表面の細かな模様や凹凸まで全てポリゴンで表現しようとすると、信じられない数のプリミティブが必要になってしまいます。そこで考案されたのが、プリミティブに画像を貼り付ける「テクスチャマッピング」という技法です。

# テクスチャクラス

## クラス定義

OpenGLでテクスチャを作成したり破棄するには、glGenTexturesとglDeleteTextures関数を使います。  
破棄のタイミングを常に気にしなければならないのは面倒なので、std::shared\_ptrというクラスを使って自動化することにしましょう。shared\_ptrは、自身とそのコピーが全て破棄されたとき、自動的にdeleteを呼んでくれるクラスです。shared\_ptrのようなクラスは、通常のポインタより高度な管理機能を備えていることから「スマート(賢い)ポインタ」と呼ばれます。

それでは、SrcフォルダにTexture.hとTexture.cppを作成してください。  
作成したTexture.hを開き、次のコードを追加します。

/\*\*  
\* @file Texture.h  
\*/  
#ifndef TEXTURE\_H\_INCLUDED  
#define TEXTURE\_H\_INCLUDED  
#include <GL/glew.h>  
#include <memory>  
  
class Texture;  
using TexturePtr = std::shared\_ptr<Texture>; ///< テクスチャポインタ.  
  
/\*\*  
\* テクスチャクラス.  
\*/  
class Texture  
{  
public:  
 static TexturePtr Create(  
 int width, int height, GLenum iformat, GLenum format, const void\* data);  
  
 GLuint Id() const { return texId; }  
 GLsizei Width() const { return width; }  
 GLsizei Height() const { return height; }  
  
private:  
 Texture() = default;  
 ~Texture();  
 Texture(const Texture&) = delete;  
 Texture& operator=(const Texture&) = delete;  
  
 GLuint texId = 0;  
 int width = 0;  
 int height = 0;  
};  
  
#endif

最初に幾つかのヘッダをインクルードします。  
OpenGLの基本的な型や機能を利用したい場合、通常はgl.hをインクルードします。しかし、GLEWを使う場合はgl.hのかわりにglew.hを使うべきです。この講義ではGLEWを使っているため、glew.hをインクルードしています。  
memoryヘッダはstd::shared\_ptrを使うためのヘッダです。

さて、Textureクラスの実装ですが、GLFWEWクラスのように、コンストラクタやデストラクタをprivateメンバとしています。直接作成することはできないので、作成用のstaticメンバ関数を用意しました。  
また、テクスチャの情報を取得する3つのメンバ関数は、内容がシンプルなので、ここで定義してしまいます。

なお、OpenGLにおいて、有効なテクスチャを指すIDは0以外の値を取ることが保証されています。そして、0はテクスチャがないことを示すことになっています。そこで、texIdを0で初期化しています。  
そして、テクスチャがない場合の幅と高さは0で初期化しています。

## ヘッダのインクルード

今度はTextureクラスのメンバ関数を実装していきましょう。  
Texture.cppを開き、次のコードを追加してください。

/\*\*  
\* @file Texture.cpp  
\*/  
#include "Texture.h"  
#include <iostream>

Texture.hは先ほど作成したヘッダファイルですね。iostreamヘッダはエラー出力用にインクルードしています。

## デストラクタ

まずはデストラクタを実装します。  
コンストラクタ定義の下に、次のコードを追加してください。

/\*\*  
\* デストラクタ.  
\*/  
Texture::~Texture()  
{  
 if (texId) {  
 glDeleteTextures(1, &texId);  
 }  
}

デストラクタでは、texIdが0以外だったらglDeeteTextures関数を呼び出すようになっています。  
こうしておけば、Textureクラスが破棄されたときには、それが指しているテクスチャも自動的に破棄されるというわけです。なお、デストラクタが走った後のオブジェクトにアクセスすることはできないので、texIdに0を代入する必要はありません。

## テクスチャ作成関数

テクスチャ作成関数は少し長いので、ちょっとずつ実装していきます。  
まずはデストラクタ定義の下に、次のコードを追加してください。

/\*\*  
\* 2Dテクスチャを作成する.  
\*  
\* @param width テクスチャの幅(ピクセル数).  
\* @param height テクスチャの高さ(ピクセル数).  
\* @param iformat テクスチャのデータ形式.  
\* @param format アクセスする要素.  
\* @param data テクスチャデータへのポインタ.  
\*  
\* @return 作成に成功した場合はテクスチャポインタを返す.  
\* 失敗した場合はnullptr返す.  
\*/  
TexturePtr Texture::Create(  
 int width, int height, GLenum iformat, GLenum format, const void\* data)  
{  
 struct Impl : Texture {};  
 TexturePtr p = std::make\_shared<Impl>();  
  
 **//<--- ここにコードを追加していきます --->**  
  
 return p;  
}

最初の行で関数内クラスを定義していますね。これ、なんの役に立つんでしょう？  
実はこれ、privateなコンストラクタを持つクラスをshared\_ptrに持たせるためのトリックなんです。  
shared\_ptr自体はTextureクラスとは無関係な独立したクラスです。そのため、Textureクラスのprivateメンバにアクセスすることができません。一方、関数内クラスは、関数または関数を所有するクラスのメンバとみなされます。ImplクラスはTextureのメンバ関数内で定義されているのでTextureクラスのメンバとして扱われます。その結果、Textureクラスのprivateメンバを呼び出すことができるわけです。さらにImplクラス自身のコンストラクタはpublicなので、当然shared\_ptrはそれを呼び出すことができる、というからくりです。  
なお、std::make\_sharedというのはshared\_ptrオブジェクトを作成する関数です。

これでTextureオブジェクトが作成できるようになったので、次はテクスチャを作成します。  
make\_shared関数呼び出しの下に、次のコードを追加してください。

struct Impl : Texture {};  
 TexturePtr p = std::make\_shared<Impl>();  
 **+**p->width = width;  
**+**p->height = height;  
**+**glGenTextures(1, &p->texId);  
**+**glBindTexture(GL\_TEXTURE\_2D, p->texId);  
**+**glTexImage2D(  
**+** GL\_TEXTURE\_2D, 0, iformat, width, height, 0, format, GL\_UNSIGNED\_BYTE, data);  
**+**const GLenum result = glGetError();  
**+**if (result != GL\_NO\_ERROR) {  
**+** std::cerr << "ERROR テクスチャ作成に失敗: 0x" << std::hex << result << std::endl;  
**+** return {};  
**+**}  
  
 return p;

最初の2行は幅と高さをコピーしているだけです。これは簡単ですね。  
そこから先が、実際にテクスチャを作成するコードです。まずglGenTextures関数でテクスチャオブジェクトを作成します。この関数は、これまで見てきたglGenBuffersやglGenVertexArrays関数とよく似ています。最初の引数は、作成するテクスチャオブジェクトの数です。今回は1つだけ作成するので1を渡しています。2つめの引数は、作成したテクスチャオブジェクトのIDを受け取る変数へのポインタです。

次に、glBindTexture関数を使い、作成したオブジェクトを特定のテクスチャ処理スロット(OpenGLではターゲットと呼びます)に割り当てています。最初の引数には割当先となるターゲットを指定します。OpenGLにはテクスチャ用のターゲットが複数あり、個別にテクスチャを割り当てることができます。ここでは、2Dテクスチャをターゲットにしています。2つめの引数は、割り当てるテクスチャのIDです。

最後に、glTexImage2Dでテクスチャの属性と画像データを転送します。引数が多いので、関数の宣言を見ながらひとつづつ説明します。glTexImage2Dの宣言を次に示します。

void glTexImage2D(GLenum target, GLint level, GLint internalFormat,  
 　GLsizei width, GLsizei height, GLint border, GLenum format,  
 　GLenum type, const GLvoid \* data);

***target***: 転送先となるターゲットです。glBindTextureと同じ値を指定します。

***level***: ターゲットに設定するミップマップレベルです。ミップマップ(MIP map)というのは、オブジェクトとカメラの距離に応じて解像度の違うテクスチャを使い分ける機能のことです。0は距離が最も近い場合で、数字が大きくなるほど、より遠い場合に使われるテクスチャということになります。レベルの最大値はハードウェアやドライバによって違いますが、8か16が多いです。

NOTE: MIPはラテン語の「*multum in parvo*」の略称です。これは「小さなものにたくさんのものが入っている」というような意味だそうです。

***internalFormat***: 転送元および転送先のデータ形式を指定します。

***width***: テクスチャの幅をピクセル数で指定します。

***height***: テクスチャの高さをピクセル数で指定します。

***border***: 常に0を指定してください(OpenGL2.xまでは境界線の有無を指定していましたが、3.0以降使われなくなりました)。

***format***: OpenGLが実際に処理する要素を指定します。例えばinternalFormatがGL\_RGB8でformatがGL\_REDの場合、読み書きを行うときには赤以外の要素は無視されます。ただし、internalFormatにない要素を追加することはできません。例えばinternalFormatがGL\_RGB8の場合、GL\_RGBAは指定できません。

***type***: データの読み書きに使うフォーマットを指定します。例えばinternalFormatがGL\_RGB8の場合、バイト単位で読み書きし、値の範囲は0-255なのでGL\_UNSIGNED\_BYTEにします。

***data***: 転送元データへのポインタです。nullptrを指定すると、何が描かれているか分からないテクスチャを作ることができます(描画先に使いたい場合に指定します)。

glTexImage2D関数のあとは、glGetError関数で作成に成功したかどうかを調べています。成功していればGL\_NO\_ERRORが返されるはずです。それ以外が返された場合は作成失敗なので、エラーメッセージを出して空のポインタを返しています。エラーメッセージの作成で「std::hex」というものを使っています。これは、「IOマニピュレータ」と呼ばれるもののひとつで、「ここから先は整数を16進数で表示せよ」という指示です。10進数に戻すには「std::dec」を使います。

NOTE: エラー時にglDeleteTextures関数を呼んでいないことに注意してください。glDeleteTexturesはTexturePtrが自動的に呼び出してくれるのでしたね。

テクスチャが作成できたら、さらにglTexImage2Dでは設定できないいくつかのパラメータを設定します。

if (result != GL\_NO\_ERROR) {  
 std::cerr << "ERROR テクスチャ作成に失敗: 0x" << std::hex << result << std::endl;  
 return {};  
}  
  
**+**glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MAX\_LEVEL, 0);  
**+**glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MIN\_FILTER, GL\_LINEAR);  
**+**glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MAG\_FILTER, GL\_LINEAR);  
**+**glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_S, GL\_CLAMP\_TO\_EDGE);  
**+**glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_T, GL\_CLAMP\_TO\_EDGE);  
  
 return p;

パラメータを設定する関数にはglTexParameterf/glTexParamteri/glTexParamterfv/glTexParamterivの4種類があり、パラメータの表現型によって使い分けます。今回は全てひとつの整数値で表現できるのでglTexParameteriを使っていますが、0.0～1.0のような値を必要とする場合はglTexParamterf、複数の値を必要とする場合はglTexParamterfvやglTexParamterivを使います。  
例えば境界色を指定するGL\_TEXTURE\_BORDER\_COLORは、RGBAの4要素を必要とし、それぞれ0.0~1.0の範囲になるのでglTexParameterfvを使うことになります。

さて、glTexParamteri関数の最初の引数は、見てのとおりターゲットの指定です。2つめの引数が設定するパラメータの名前で、3つめがそのパラメータに設定される値になっています。  
パラメータにはさまざまなものがありますが、ここでは以下の5種類を設定しています。

***GL\_TEXTURE\_MAX\_LEVEL***: 使用するミップマップの最大レベルです。初期値は1000なので、ミップマップを使用しない場合は0を指定する必要があります。

***GL\_TEXTURE\_MIN\_FILTER***: テクスチャを縮小表示する場合の読み取り方法です。これには以下の6種類があります。

*GL\_NEAREST*  
読み取るテクスチャ座標に最も近い1ピクセルを読み取ります。  
ミップマップがある場合、最小レベルが使われます。

*GL\_LINEAR*  
読み取るテクスチャ座標に最も近い4ピクセルを読み取り、距離による加重平均を取ります。  
ミップマップがある場合、最小レベルが使われます。

*GL\_NEAREST\_MIPMAP\_NEAREST*  
カメラからの距離に最も合うミップレベルのテクスチャを選び、GL\_NEAREST方式で読み取ります。

*GL\_LINEAR\_MIPMAP\_NEAREST*  
カメラからの距離に最も合うミップレベルのテクスチャを選び、GL\_LINEAR方式で読み取ります。

*GL\_NEAREST\_MIPMAP\_LINEAR*  
カメラからの距離が最も近いミップレベルのテクスチャを2つ選び、それぞれGL\_NEAREST方式で読み取ったあと、カメラからの距離による加重平均を取ります。

*GL\_LINEAR\_MIPMAP\_LINEAR*  
カメラからの距離が最も近いミップレベルのテクスチャを2つ選び、それぞれGL\_LINEAR方式で読み取ったあと、カメラからの距離による加重平均を取ります。

初期値はGL\_NEAREST\_MIPMAP\_LINEARなので、ミップマップを使わない場合はGL\_NEARESTかGL\_LINEARに変更しておきます。

***GL\_TEXTURE\_MAG\_FILTER***: テクスチャを拡大表示する場合の読み取り方法です。MIN\_FILTERと似ていますが、指定できるのはGL\_NEARESTとGL\_LINEARの2種類だけです。実は初期値もGL\_LINEARなのでこれは不要なのですが、MIN\_FILTERとMAX\_FILTERはペアで扱うほうが間違いを起こしにくいため設定しています。

***GL\_TEXTURE\_WRAP\_S***  
***GL\_TEXTURE\_WRAP\_T***: 0.0未満または1.0以上のテクスチャ座標が指定された場合の、横及び縦方向の座標の扱い方です。それぞれ、次の4種類のうちから指定できます。

*GL\_CLAMP\_TO\_EDGE*  
 0.0未満は0.0、1.0以上は1.0に切り捨てられます。

*GL\_CLAMP\_TO\_BORDER*  
 0.0未満、1.0以上の座標は、全てボーダーカラーという色のピクセルとして扱います。

*GL\_REPEAT*  
 整数部を無視します。例えば3.5は0.5、-1.3は0.7と同じ扱いになります。

*GL\_MIRRORED\_REPEAT*  
 GL\_REPEATと同様に整数部を無視しますが、整数部が奇数の場合は座標を反転します。例えば2.6は0.6になりますが、1.6の場合は0.4になります。

初期値はGL\_REPEATです。しかし、1枚絵を貼り付ける場合、フィルタにGL\_LINEARを指定していると、境界のピクセルが反対側のピクセルと混ざってしまうという問題が起きるため、GL\_CLAMP\_TO\_EDGEに変更しています。

最後に、テクスチャの割り当てを解除して終了です。  
パラメータ設定コードの下に、次のコードを追加してください。

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_S, GL\_CLAMP\_TO\_EDGE);  
glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_T, GL\_CLAMP\_TO\_EDGE);  
  
**+**glBindTexture(GL\_TEXTURE\_2D, 0);  
  
 return p;

これでテクスチャ作成関数は完成です。

# テクスチャ座標の追加

## 頂点データ型にテクスチャ座標メンバを追加

テクスチャは、テクスチャ座標によって参照されます。現在の頂点データにはテクスチャ座標がありませんので、これを追加していきましょう。

さて、2Dテクスチャの場合、当然ですがテクスチャ座標も2Dになります。なお、OpenGLの場合、テクスチャ座標系では左下が原点で、右上が+方向になっています。各軸の呼び方はX,Yではなく、U,V(またはS,T)です。

NOTE: UVとSTの違いは、基本的には名前の規則としてどちらを使うか、ということです。一般にはUVが使われますが、GLSLではSTです。

自分で2Dベクトル型を作成してもいいのですが、前回追加したGLMライブラリのクラスを使うほうが簡単です。ついでなので、座標やカラーもGLMのクラスに置き換えてしまいましょう。  
それでは、Main.cppに移動し、Vertex構造体を次のように書き換えてください。

**-**/// 3Dベクトル型. **-**struct Vector3  
**-**{  
**-** float x, y, z;  
**-**};  
**-  
-**/// RGBAカラー型. **-**struct Color  
**-**{  
**-** float r, g, b, a;  
**-**};  
  
 struct Vertex  
 {  
**!** glm::vec3 position; ///< 座標.  
**!** glm::vec4 color; ///< 色.  
**+** glm::vec2 texCoord; ///< テクスチャ座標.  
 };

テクスチャ座標としてtexCoordメンバ変数が追加され、Vector3をglm::vec3に、Colorをglm::vec4に書き換えています。書き換えたら、Vector3とColor構造体定義を削除しておいてください。なお、texCoordという名前はtexture coordinate(=テクスチャ座標)を縮めたものです。

## 頂点データにテクスチャ座標を追加

続いて、各頂点のテクスチャ座標を指定していきます。  
vertices変数の各頂点データの末尾にテクスチャ座標を加えて、次のコードのようにしてください。

const Vertex vertices[] = {  
**!** { {-0.5f, -0.3f, 0.5f}, {0.0f, 0.0f, 1.0f, 1.0f}, { 0.0f, 0.0f} },  
**!** { { 0.3f, -0.3f, 0.5f}, {0.0f, 1.0f, 0.0f, 1.0f}, { 1.0f, 0.0f} },  
**!** { { 0.3f, 0.5f, 0.5f}, {0.0f, 0.0f, 1.0f, 1.0f}, { 1.0f, 1.0f} },  
**!** { {-0.5f, 0.5f, 0.5f}, {1.0f, 0.0f, 0.0f, 1.0f}, { 0.0f, 1.0f} },  
  
**!** { {-0.3f, 0.3f, 0.1f}, {0.0f, 0.0f, 1.0f, 1.0f}, { 0.0f, 1.0f} },  
**!** { {-0.3f, -0.5f, 0.1f}, {0.0f, 1.0f, 1.0f, 1.0f}, { 0.0f, 0.0f} },  
**!** { { 0.5f, -0.5f, 0.1f}, {0.0f, 0.0f, 1.0f, 1.0f}, { 1.0f, 0.0f} },  
**!** { { 0.5f, -0.5f, 0.1f}, {1.0f, 0.0f, 0.0f, 1.0f}, { 1.0f, 0.0f} },  
**!** { { 0.5f, 0.3f, 0.1f}, {1.0f, 1.0f, 0.0f, 1.0f}, { 1.0f, 1.0f} },  
**!** { {-0.3f, 0.3f, 0.1f}, {1.0f, 0.0f, 0.0f, 1.0f}, { 0.0f, 1.0f} },  
 };

これでテクスチャ座標の追加は完了です。

# シェーダの変更

## VAOにテクスチャ座標を追加する

頂点データにテクスチャ座標を追加するだけでは何も起こりません。  
テクスチャ座標をシェーダに渡して、テクスチャからピクセルを読み出す必要があります。  
頂点データをシェーダの入力に割り当てるには、SetVertexAttribPointer関数を使うのでしたね。  
CreateVAO関数に移動し、次のように3つめのSetVertexAttribPointer関数呼び出しを追加してください。

SetVertexAttribPointer(0, Vertex, position);  
 SetVertexAttribPointer(1, Vertex, color);  
**+**SetVertexAttribPointer(2, Vertex, texCoord);  
 glBindVertexArray(0);

シェーダにほうにも入力を追加します。頂点データが送られるのは頂点シェーダです。ですから、次のように頂点シェーダのvColor変数定義の下に、テクスチャ座標のための入力用変数を追加します。

"layout(location=0) in vec3 vPosition;"  
 "layout(location=1) in vec4 vColor;"  
**+**"layout(location=2) in vec2 vTexCoord;"  
 "layout(location=0) out vec4 outColor;"

## テクスチャを読み込む

テクスチャのピクセルを読み込むのはフラグメントシェーダの仕事になります。いま、テクスチャ座標は頂点シェーダまでは来ているので、まずはこれをフラグメントシェーダに送りましょう。  
まず、次のように頂点シェーダのoutColor変数定義の下に出力用変数を追加します。

"layout(location=0) out vec4 outColor;"  
**+**"layout(location=1) out vec2 outTexCoord;"  
 "uniform mat4x4 matMVP;"

次に、main関数内でvTexCoordをoutTexCoordにコピーします。  
vColorをoutColorにコピーしているコードの下に、次のコードを追加してください。

"outColor = vColor;"  
**+**"outTexCoord = vTexCoord;"  
 "gl\_Position = matMVP \* vec4(vPosition, 1.0);"

これで、フラグメントシェーダの1番目の入力用変数にテクスチャ座標が送られるようになりました。

続いて、フラグメントシェーダでテクスチャ座標を受け取ります。  
フラグメントシェーダのinColor変数定義の下に、次のコードを追加してください。

"layout(location=0) in vec4 inColor;"  
**+**"layout(location=1) in vec2 inTexCoord;"  
**+**"uniform sampler2D colorSampler;"  
 "out vec4 fragColor;"

これで、inTexCoordにテクスチャ座標が格納されます。そして、その下にあるのは、テクスチャ読み込み用のオブジェクト(このようなオブジェクトを「サンプラー」といいます)の定義です。最初のuniformは、アプリケーションから渡される変数を示す修飾子でしたね。次にあるsamper2Dは2Dテクスチャを読み込むための型です。sampler2D型の変数はuniform変数か、関数の引数にしかできません。また、テクスチャは対応するサンプラー型でなければ読み込めません(例えば2Dテクスチャを読み込むにはsampler2Dを使わなければなりません)。

最後に、テクスチャを読み込んで頂点色に掛け合わせます。フラグメントシェーダのmain関数内のコードを、次のコードで置き換えてください。

"void main() {"  
**!**" fragColor = inColor \* texture(colorSampler, inTexCoord);"  
 "}";

このコードでは、texture関数を使ってテクスチャを読み込み、inColorと掛け合わせています。texture関数の最初の引数は読み込みに使うサンプラーです。2つめの引数は読み込むテクスチャ座標です。  
texture関数が実行されると、サンプラーに結びつけられたテクスチャから指定したテクスチャ座標のピクセルデータが読み取られ、関数の戻り値として返されます。

# テクスチャを作成する

テクスチャ座標の準備ができたら、Textureクラスを使ってテクスチャを作成します。  
まずはヘッダファイルをインクルードしましょう。Main.cppに移動し、GLFWEW.hのインクルード文の下に、Texture.hのインクルード文を追加してください。

#include "GLFWEW.h"  
**+**#include "Texture.h"  
 #include <glm/gtc/matrix\_transform.hpp>

次に、glEnable関数呼び出しの手前に次のコードを追加してください。

if (!vbo || !ibo || !vao || !shaderProgram) {  
 return 1;  
 }  
  
**+**/// テクスチャデータ.  
**+**static const uint32\_t textureData[] = {  
**+** 0xffffffff, 0xffcccccc, 0xffffffff, 0xffcccccc, 0xffffffff,  
**+** 0xff888888, 0xffffffff, 0xff888888, 0xffffffff, 0xff888888,  
**+** 0xffffffff, 0xff444444, 0xffffffff, 0xff444444, 0xffffffff,  
**+** 0xff000000, 0xffffffff, 0xff000000, 0xffffffff, 0xff000000,  
**+** 0xffffffff, 0xff000000, 0xffffffff, 0xff000000, 0xffffffff,  
**+**};  
**+**TexturePtr tex = Texture::Create(5, 5, GL\_RGBA8, GL\_RGBA, textureData);  
**+**if (!tex) {  
**+** return 1;  
**+**}  
  
 glEnable(GL\_DEPTH\_TEST);

textureDataは5x5ピクセルのテクスチャデータです。このサイズ情報はTexture::Create関数の最初の2つの引数として渡されます。  
データ形式はGL\_RGBA8です。この形式は下位ビットからR,G,B,Aの順番で各色8bitのデータが格納されていることを示しています。また、読み込み形式にはGL\_RGBAを指定しています。この形式は、全ての要素が下位ビットからR,G,B,Aの順番で読み出されます。データ形式と読み出し形式は似た名前が付いていますが、全く別のパラメータです。混同しがちなので注意してください。  
作成に失敗した場合、nullptrが返されます。その場合は1を返して終了しています。

テクスチャの作成はこれだけです。

# テクスチャを使う

## サンプラーの位置を取得する

テクスチャ座標を追加し、テクスチャも作成しました。これでようやくプリミティブにテクスチャを貼ることができます。  
ところで、シェーダを書き換えるとき、「サンプラーに結びつけられたテクスチャ」と言いました。つまり、テクスチャを読み込むには、サンプラーに結びつける必要がある、ということです。

テクスチャとサンプラーに結びつけるには、まずシェーダ内にあるサンプラーの位置を知る必要があります。  
サンプラーはuniform変数なので、行列のときと同じくglGetUniformLocation関数が使えます。

メインループ内の行列データ転送処理の下に、次のコードを追加してください。

glUniformMatrix4fv(matMVPLoc, 1, GL\_FALSE, &matMVP[0][0]);  
 }  
**+**const GLint colorSamplerLoc = glGetUniformLocation(shaderProgram, "colorSampler");  
 glBindVertexArray(vao);

## サンプラーとテクスチャを結びつける

続いて、取得に成功した場合にテクスチャを結びつける処理を加えます。  
glGetUniformLocation関数呼び出しの下に、次のコードを追加してください。

const GLint colorSamplerLoc = glGetUniformLocation(shaderProgram, "colorSampler");  
**+**if (colorSamplerLoc >= 0) {  
**+** glUniform1i(colorSamplerLoc, 0);  
**+** glActiveTexture(GL\_TEXTURE0);  
**+** glBindTexture(GL\_TEXTURE\_2D, tex->Id());  
**+**}  
 glBindVertexArray(vao);

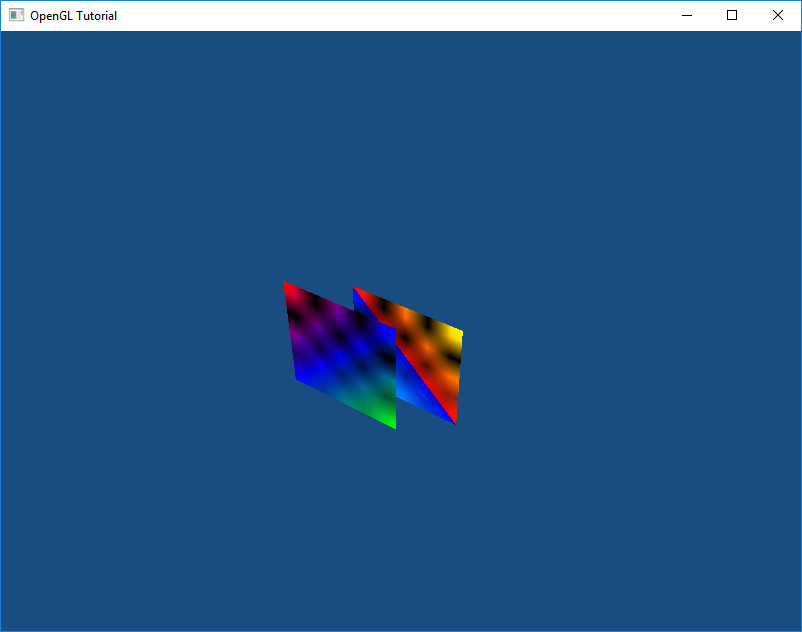
glGetUniformLocation関数が成功した場合、0以上の値を返すのでしたね。

さて、glUniform1i関数(関数名の最後の部分は「いち・あい」です。「える・あい」ではありません)は、1つのint型の値をuniform変数に設定する関数です。サンプラーの場合、使用するテクスチャイメージユニットの番号を指定します。テクスチャイメージユニットとは、OpenGLのテクスチャ処理装置です。サンプラーとテクスチャは、テクスチャイメージユニットを介して結びつけられるようになっています。

glActiveTextureは指定したテクスチャイメージユニットを現在のテクスチャ処理の対象にする関数です。  
テクスチャイメージユニットはGL\_TEXTURE0~31という名前で指定します。

最後に、glBindTextureで、glActiveTextureによって処理対象になっているテクスチャイメージユニットに、テクスチャを割り当てます。

これで、ようやくテクスチャが貼れるようになりました。  
実行すると、次のように表示されると思います。



# テクスチャ読み込み関数の実装

## 画像ファイルの形式を選ぶ

ここまではコードに埋め込んだデータからテクスチャを作成していました。しかし、画像ファイルから作成できるほうがなにかと便利です。そこで、画像ファイルをテクスチャとして読み込む関数を作ることにしましょう。

画像ファイルといっても、いろいろなフォーマットがあります。よく使われるのはJPGやPNGですが、これらは圧縮されているため、読み込んですぐにテクスチャとして使うということができません。そこで、今回は比較的簡単にテクスチャにできるBMPを読み込むことにします。BMPファイル自体にもさまざまなデータ形式がありますが、簡単にするため、24bit無圧縮に限定します。また、BMPファイルは横１列のバイト数が4の倍数でない場合、4の倍数になるように詰め物(パディングといいます)を追加する仕様になっています。OpenGLの初期状態では、テクスチャも同じ仕様になっているため、データを変更しなくても読み込むことができます。しかし、有効な画像データかどうかを調べるために、幅と高さから画像データのサイズを計算する際は、この詰め物を考慮しなければなりません。

NOTE: 例えば横幅7ピクセルの画像の場合、7ピクセルx3バイト(=24bit)は21バイトです。21は4の倍数ではないので、21以上で最小の4の倍数である24が実際の横１列のバイト数になります。

## 標準のファイル操作関数を使用可能にする

C言語にはファイルを操作する関数が存在するので、ファイルの読み込みにはそれらを使うことができます。ところがMicrosoftは「簡単にセキュリティ上問題のある使いかたができてしまう」という理由で、これらの関数をコンパイルできないようにしてしまいました。しかし、この措置は移植性を考慮したアプリケーションでは致命的なので、正常な状態に戻す方法が用意されています。これは次の手順になります。

まず、ソリューションエクスプローラーのプロジェクト名を右クリックし、メニューの一番下にある「プロパティ」を選択してプロパティページを開きます。開いたら、構成を「すべての構成」、プラットフォームを「すべてのプラットフォーム」にします。次に、ウィンドウ右ペインの「C/C++」をダブルクリックして展開し、「プリプロセッサ」を選択します。そして、「プリプロセッサの定義」を選択してさらに右端の下向き矢印を選択、表示された<編集…>を選択すると、「プリプロセッサの定義」ウィンドウが開きます。  
この一番上のテキストボックスに「\_CRT\_SECURE\_NO\_WARNINGS」と打ち込み、「OK」ボタンを押します。プロパティページに戻るので、さらに「OK」ボタンを押すと変更が適用されます。

これで、Microsoftのおせっかいを止めることができました。

## 関数宣言を追加する

まずはTexture.hを開き、テクスチャファイル読み込み関数を宣言します。  
Texture::Create関数宣言の下に、次のコードを追加してください。

static TexturePtr Create(  
 int width, int height, GLenum iformat, GLenum format, const void\* data);  
**+**static TexturePtr LoadFromFile(const char\*);

## ヘッダファイルのインクルード

まずはファイルの読み込み処理で必要になるヘッダファイルをインクルードしましょう。  
Texture.cppを開き、iostreamヘッダの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 "Texture.h"  
 #include <iostream>  
**+**#include <vector>  
**+**#include <stdint.h>  
**+**#include <stdio.h>  
**+**#include <sys/stat.h>

vectorは読み込みバッファのために使います。stdint.hをインクルードするとint8\_tなどのビット幅指定型が使えるようになります。そして、stdio.hとstat.hはファイル操作関数を使うためにインクルードしています。

## BMPフォーマットの構造

BMPファイルの先頭には、BMPファイルであることを示す小さなデータが入っています。  
これをビットマップファイルヘッダといいます。ビットマップファイルヘッダは次のような構造になっています。

|  |  |  |
| --- | --- | --- |
| **オフセット** | **バイト数** | **格納されている情報** |
| 0 | 2 | ファイルタイプ(Windows用BMPでは常に’B’,’M’) |
| 2 | 4 | ファイルのバイト数 |
| 6 | 2 | 将来のための予約領域 |
| 8 | 2 | 将来のための予約領域 |
| 10 | 4 | ファイルの先頭から画像データまでのバイト数 |

ビットマップファイルヘッダの直後には、画像に関する情報が格納されています。しかし、実はBMPファイルにはいくつかのバージョンがあるんです。面倒なことに、バージョンによってこの情報の内容や順序が違っています。とはいえ、実際に使われているのは、Windows用バージョン1の場合がほとんどです。このバージョンは次のように定義されています。

|  |  |  |
| --- | --- | --- |
| **オフセット** | **バイト数** | **格納されている情報** |
| 14 | 4 | 画像情報のバイト数(40バイト) |
| 18 | 4 | 横のピクセル数 |
| 22 | 4 | 縦のピクセル数 |
| 26 | 2 | 画像の枚数(常に1) |
| 28 | 2 | 1ピクセルのビット数(1,4,8,16,24,32のいずれか) |
| 30 | 4 | 圧縮方式(0なら無圧縮) |
| 34 | 4 | 画像データのバイト数 |
| 38 | 4 | 水平解像度(1mあたりのピクセル数) |
| 42 | 4 | 垂直解像度(1mあたりのピクセル数) |
| 46 | 4 | カラーパレットのうち、実際に使われる色数 |
| 50 | 4 | カラーパレットのうち、画像を再現する上で重要だとされる色数 |

さて、上記のとおり、BMPファイルの情報はバイト単位で情報を詰め込んでいます。そのため、そのまま構造体として定義してしまうと、ビルド環境によってはメンバのオフセットが一致しません(実際に、Window環境では一致しません)。そこで、バイト列から数値を復元する関数を作成し、それを使って数値を取得することにします。  
stat.hのインクルード文の下に、次のコードを追加してください。

/\*\*  
\* バイト列から数値を復元する.  
\*  
\* @param p バイト列へのポインタ.  
\* @param offset 数値のオフセット.  
\* @param size 数値のバイト数(1～4).  
\*  
\* @return 復元した数値.  
\*/  
uint32\_t Get(const uint8\_t\* p, size\_t offset, size\_t size)  
{  
 uint32\_t n = 0;  
 p += offset;  
 for (size\_t i = 0; i < size; ++i) {  
 n += p[i] << (i \* 8);  
 }  
 return n;  
}

BMPファイルの数値情報は、下位から8ビットずつ順番に格納されています。これをアプリケーションで扱える数値に復元するには、8ビットずつ取り出して元のビット位置に戻していきます。

## ファイルを読み込む

BMPファイルの構造が分かったところで、読み込み関数を実装していきます。  
Texture::Create関数定義の下に、次のコードを追加してください。

/\*\*  
\* ファイルから2Dテクスチャを読み込む.  
\*  
\* @param filename ファイル名.  
\*  
\* @return 作成に成功した場合はテクスチャポインタを返す.  
\* 失敗した場合はnullptr返す.  
\*/  
TexturePtr Texture::LoadFromFile(const char\* filename)  
{  
 **//<--- ここにコードを追加していきます --->**  
}

では、少しずついきましょう。

続いて、stat関数でファイルサイズを取得します。  
関数の先頭に、次のコードを追加してください。

TexturePtr Texture::LoadFromFile(const char\* filename)  
 {  
**+** //ファイルサイズを取得する. **+** struct stat st;  
**+** if (stat(filename, &st)) {  
**+** return {};  
**+** }

stat関数の最初の引数はファイルパスです。2つめの引数は、ファイルの情報を格納する変数へのポインタです。この変数の型は、関数名と同じstatという構造体です。stat構造体にはさまざまなメンバ変数がありますが、これを全て説明するにはファイルシステムについての理解が必要です。しかし、今回必要なのはファイルサイズを示すst\_sizeだけです。  
stat関数が情報の取得に成功した場合は0、失敗した場合は-1が返されるので、失敗した場合は空のオブジェクトを返しています。

NOTE: 関数と構造体(またはクラス)には同じ名前をつけることができます。ただし、名前を使うときは関数名が優先されます。そのため、名前を構造体(またはクラス)として使うときは、名前の前にstruct/classを置いて、構造体(またはクラス)であることを明示しなければなりません。

ファイルサイズが手に入ったら、BMPファイルの情報を保持できるサイズかを調べます。  
ファイルサイズ取得処理の下に、次のコードを追加してください。

if (stat(filename, &st)) {  
 return {};  
}  
**+**// ファイルサイズがBMPファイルの情報を保持できるサイズ未満なら空のオブジェクトを返す. **+**const size\_t bmpFileHeaderSize = 14; // ビットマップファイルヘッダのバイト数  
**+**const size\_t windowsV1HeaderSize = 40; // ビットマップ情報ヘッダのバイト数.  
**+**if (st.st\_size < bmpFileHeaderSize + windowsV1HeaderSize) {  
**+** return {};  
**+**}

Windows用バージョン1のBMPファイルには、少なくとも14+40バイトが必要です。この数値未満の場合、BMPファイルではないので、空のオブジェクトを返して関数を終了します。

サイズ判定をパスした場合、ファイルを開きます。  
関数の先頭に、次のコードを追加してください。

if (st.st\_size < bmpFileHeaderSize + windowsV1HeaderSize) {  
 return {};  
}  
**+**// ファイルを開く. **+**FILE\* fp = fopen(filename, "rb");  
**+**if (!fp) {  
**+** return {};  
**+**}

fopen関数は、ファイルを開くまたは作成する関数です。最初の引数はファイルパス、2つめの引数は、動作モードを指定する文字列です。”r”は「読み取り専用で開く」、”b”は「バイナリファイルとして開く」ことを意味します。戻り値はFILE型のポインタです。FILE型はファイルを表現する型です。fopenがファイルを開くのに失敗した場合、nullptrが返されます。その場合は空のオブジェクト(実体はnullptrです)を返しています。

ファイルが開けたら、次は読み込みです。

if (!fp) {  
 return {};  
}  
**+**// ファイルを読み込む. **+**std::vector<uint8\_t> buf;  
**+**buf.resize(st.st\_size);  
**+**const size\_t readSize = fread(buf.data(), 1, st.st\_size, fp);  
**+**fclose(fp);  
**+**if (readSize != st.st\_size) {  
**+** return {};  
**+**}

まず、vector型の変数bufを定義し、resize関数でbufにファイルサイズ分の領域を確保しています。  
次に、fread関数でファイルを読み込みます。この関数は次のように定義されています。

size\_t fread(void\*　*ptr*, size\_t *size*, size\_t *count*, FILE\* *stream*);

***ptr***: 読み込み先バッファへのポインタです。

***size***: 読み込む要素のサイズをバイト数で指定します。

***count***: 読み込む要素数です。実際に読み込まれるのはsize\*countバイトになります。

***stream***: 読み込むファイルを示すFILE型ポインタ。

fread関数の戻り値は、読み込まれた要素数です。

コードではbuf.data()でbufの先頭アドレスを取得し、それを読み込み先バッファにしています。  
また、ファイルを読み込む場合、sizeを1、countをファイルサイズにする場合がほとんどです。というのも、ファイル全体が正しく読み込めたかを調べるには、ファイルサイズと読み込まれたバイト数を比較するのが簡単だからです。fread関数の戻り値は「読み込まれた要素数」なので、要素のサイズが1なら「要素数＝バイト数」となるわけですね。

読み込みが済んだらファイルを閉じます。ファイルを閉じるには、fclose関数にFILE型ポインタを渡します。fclose関数が実行されたあとでFILE型ポインタにアクセスするとエラーになるので注意してくださいね。

ファイルを閉じた後、ファイル全体が読み込まれたかをチェックし、読み込まれていなかったら空のオブジェクトを返して終了しています。

## 本当にBMPファイルなの？

続いて、読み込んだデータが本当にBMPファイルかどうかを検査します。  
ファイル読み込み処理の下に、次のコードを追加してください。

if (readSize != st.st\_size) {  
 return {};  
}  
**+**// BMPファイルでなければ空のオブジェクトを返す. **+**const uint8\_t\* pHeader = buf.data();  
**+**if (pHeader[0] != 'B' || pHeader[1] != 'M') {  
**+** return {};  
**+**}

Windows用BMPファイルの先頭2バイトは常に’B’と’M’です。これが一致しなければ、少なくともWindows用BMPファイルではないということなので、空のオブジェクトを返して関数を終了します。

## BMPファイルの情報を取得する

先頭2バイトが一致したら、読み込んだファイルがBMPである可能性が出てきます。  
さらに情報を取得して、詳しく検査していきます。  
先頭2バイトを検査するコードの下に、次のコードを追加してください。

if (pHeader[0] != 'B' || pHeader[1] != 'M') {  
 return {};  
 }  
**+**// BMPファイルの情報を取得する. **+**const size\_t offsetBytes = Get(pHeader, 10, 4);  
**+**const uint32\_t infoSize = Get(pHeader, 14, 4);  
**+**const uint32\_t width = Get(pHeader, 18, 4);  
**+**const uint32\_t height = Get(pHeader, 22, 4);  
**+**const uint32\_t bitCount = Get(pHeader, 28, 2);  
**+**const uint32\_t compression = Get(pHeader, 30, 4);  
**+**const size\_t pixelBytes = bitCount / 8;  
**+**if (infoSize != windowsV1HeaderSize || bitCount != 24 || compression){  
**+** return {};  
**+**}

読み込んだデータはただのバイト列なので、そこから情報を取得するために、7.4節で実装したGet関数を使います。各情報とオフセットとサイズは7.4節に書いてあるので、それを引数に渡しています。  
また、ここではBMPファイルのチェックと、テクスチャの作成に必要な情報しか取得していません。使わない情報を取得するのは無駄だからです。

情報を取得したら、情報ヘッダのサイズが正しいかを調べます。違っていたら、このファイルはWindowバージョン1ではないので空のオブジェクトを返して終了します。また、今回は24bit無圧縮形式だけを読み込みたいので、bitCountが24でなかったり、compressionが0でない場合も同様に終了します。

続いて、画像データが正しく格納されているかを検査します。  
詳細データを検査するコードの下に、次のコードを追加してください。

if (infoSize != windowsV1HeaderSize || bitCount != 24 || compression){  
 return {};  
}  
**+**const size\_t stride = ((width \* pixelBytes + 3) / 4) \* 4; // 実際の横バイト数.  
**+**const size\_t imageSize = stride \* height;  
**+**if (buf.size() < offsetBytes + imageSize) {  
**+** return {};  
**+**}

まず横1列のバイト数を求めます。単純に考えると「横ピクセル数x1ピクセルのバイト数」ですが、BMPファイルの仕様上は4の倍数となるように切り上げる必要があるのでしたね。整数の切り上げ計算には定番のやりかたがあります。それは「「切り上げたい倍数-1」を加算し、倍数で割って、倍数を掛ける」方法です。上記のstride変数の計算ではこの方法を使っています。  
横のバイト数が求まったら、あとは高さを掛けるだけで画像データのバイト数を求めることができます。

画像データは、ビットマップファイルヘッダの「画像データまでのバイト数」の位置から始まります。それに、画像データのバイト数を加えた位置が、画像データの終端です。この終端がbufに含まれていなければ、画像データは途中で切れているか、取得した情報が間違っています。その場合、空のオブジェクトを返して関数を終了します。

## テクスチャを作成する！

ここまでの検査をくぐり抜けた場合、ファイルは高確率でWindowバージョン1で24bit無圧縮のBMP画像データです。あとは、これをCreate関数に渡せばテクスチャが作成できるはずです！  
それでは、画像データのサイズ検査コードの下に、次のコードを追加してください。

if (buf.size() < offsetBytes + imageSize) {  
 return {};  
}  
**+**return Create(width, height, GL\_RGB8, GL\_BGR, buf.data() + offsetBytes);

必要な情報は全て作成済みなので、テクスチャの作成はCreate関数を呼び出すだけです。  
読み込んだBMPは24bit形式なので、データ形式はGL\_RGB8としています。  
しかし、フォーマットをGL\_BGRとしていることに気をつけてください。これは、BMPファイルが色データをB, G, Rの順で記録しているためです。

# BMPファイルを読み込む

## BMPファイルを用意する

それでは、早速BMPファイルからテクスチャを作成してみましょう！  
と、それにはBMPファイルが必要ですね。PhotoShopやWindow Paint、その他の適当なツールでBMPファイルを作成してください。Visual Studioにも簡単なグラフィックツールが付いているので、それを使うのもいいでしょう。  
作成したBMPファイルはプロジェクトフォルダの直下にResフォルダを作成し、そこに入れておいてください。ファイル名は自由につけてもらって構いません。

いずれにしても、画像の横幅を4の倍数にすることと、保存するときに24bit無圧縮形式を選ぶことを忘れないようにしてください。

## 用意したBMPファイルを読み込む

さて、BMPファイルが用意できたら、それを読み込みます。  
Main.cppを開き、テクスチャ作成コードを次のように書き換えてください。

**-**static const uint32\_t textureData[] = {  
**-** 0xffffffff, 0xffcccccc, 0xffffffff, 0xffcccccc, 0xffffffff,  
**-** 0xff888888, 0xffffffff, 0xff888888, 0xffffffff, 0xff888888,  
**-** 0xffffffff, 0xff444444, 0xffffffff, 0xff444444, 0xffffffff,  
**-** 0xff000000, 0xffffffff, 0xff000000, 0xffffffff, 0xff000000,  
**-** 0xffffffff, 0xff000000, 0xffffffff, 0xff000000, 0xffffffff,  
**-**};  
**-**TexturePtr tex = Texture::Create(5, 5, GL\_RGBA8, GL\_RGBA, textureData);  
**+**TexturePtr tex = Texture::LoadFromFile("Res/Sample.bmp");  
 if (!tex) {  
 return 1;  
 }

現在のテクスチャ作成処理を削除して、BMPファイルの読み込み処理を追加しています。このコードではテクスチャ名を「Sample.bmp」としていますが、これはみなさんが付けた名前に変更してくださいね。

さて、これでみなさんの作成したテクスチャがプリミティブに貼り付けられて表示されるはずです。  
ビルドして実行してみてください。

## テクスチャをそのまま表示する

テクスチャは表示されたものの、プリミティブの色と混ざって妙な色合いになっています。  
テクスチャの色をそのまま表示するには、プリミティブの頂点カラーを真っ白にするだけです。  
vertices変数を次のように変更してください。

const Vertex vertices[] = {  
**!** { {-0.5f, -0.3f, 0.5f}, {1.0f, 1.0f, 1.0f, 1.0f}, {0.0f, 0.0f} },  
**!** { { 0.3f, -0.3f, 0.5f}, {1.0f, 1.0f, 1.0f, 1.0f}, {1.0f, 0.0f} },  
**!** { { 0.3f, 0.5f, 0.5f}, {1.0f, 1.0f, 1.0f, 1.0f}, {1.0f, 1.0f} },  
**!** { {-0.5f, 0.5f, 0.5f}, {1.0f, 1.0f, 1.0f, 1.0f}, {0.0f, 1.0f} },

とりあえずひとつめのプリミティブの色データだけを全て1.0fで埋めてみました。

変更したら、ビルドして実行してみてください。  
変更したプリミティブのほうは、画像ファイルがそのまま表示されると思います。

# やってみよう

* GL\_TEXTURE\_MIN\_FILTER, GL\_TEXTURE\_MAG\_FILTERの値を変更して、表示がどう変わるか確認してみましょう。
* GL\_TEXTURE\_WRAP\_S, GL\_TEXTURE\_WRAP\_Tの値を変更して、表示がどう変わるか確認してみましょう。
* glTexImage2D関数のformat引数をGL\_REDやGL\_GREEN,GL\_BLUEなどに変更して、表示がどう変わるか確認してみましょう。