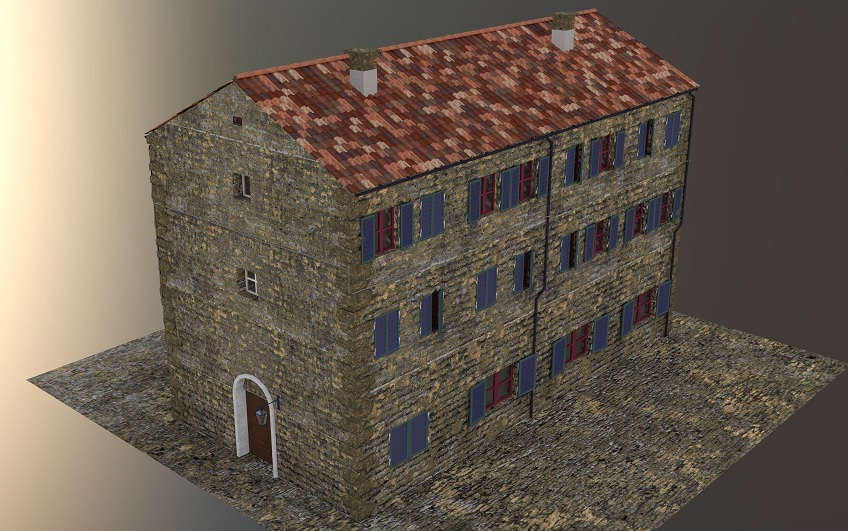
[OpenGL 3D 2018 第04回]

テクスチャー・パワー！メイク・アップ！

# 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(texture)は「織物の出来栄え、手触り。物の質感」という意味です。物体の表面の色や凹凸、柔らかさといったものを表します。次の画像の床や壁などは、わずかな数のプリミティブで作られた単なる平面なのですが、石模様のテクスチャを貼り付けることで、あたかも本当に石のモデルが置かれているかのように見せています。



## Texture.hの作成

ここからは、実際にテクスチャ・マッピングのプログラムを作っていきます。  
テクスチャを操作する関数は、専用のファイルに作っていくことにしましょう。ということで、ソリューションエクスプローラーの「ソースファイル」フィルターを右クリックして、SrcフォルダにTexture.hというヘッダーファイルを作成してください。作成したTexture.hを開き、次のプログラムを追加してください。

**+**/\*\*  
**+**\* @file Texture.h  
**+**\*/  
**+**#ifndef TEXTURE\_H\_INCLUDED  
**+**#define TEXTURE\_H\_INCLUDED  
**+**#include <GL/glew.h>  
**+**  
**+**namespace Texture {  
**+**  
**+**GLuint CreateImage2D(GLsizei width, GLsizei height,　const GLvoid\* data);  
**+**  
**+**} // namespace Texture  
**+**  
**+**#endif // TEXTURE\_H\_INCLUDED

最初にGL/glew.hをインクルードします。  
そして、Texture(てくすちゃ)名前空間を作成して、その中にCreateImage2D(くりえいと・いめーじ・つーでぃー)という名前の関数を宣言しました。これがテクスチャを作成する関数になります。  
テクスチャの作成にはさまざまな情報が必要なのですが、今回はテクスチャの幅と高さ、そして画像データの3つだけを引数として、残りは関数の中で設定することにしました。

## Texture.cppの作成

今度はソースファイルを作ります。ソリューションエクスプローラーの「ソースファイル」フィルターを右クリックして、SrcフォルダにTexture.cppというCPPファイルを作成してください。  
作成したTexture.cppを開き、次のプログラムを追加してください。

**+**/\*\*  
**+**\* @file Texture.cpp  
**+**\*/  
**+**#include "Texture.h"  
**+**#include <iostream>  
**+  
+**/// テクスチャ関連の関数やクラスを格納する名前空間.  
**+**namespace Texture {  
**+  
+**} // namespace Texture

Texture.hは先ほど作成したヘッダーファイルですね。もうひとつのiostreamのほうは、エラーメッセージを出力するときのためにインクルードしています。

## 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。最初はコメントと雛形から書いていきましょう。Texture名前空間の中に、次のプログラムを追加してください。

#include "Texture.h"  
 #include <iostream>  
  
/// テクスチャ関連の関数やクラスを格納する名前空間.  
 namespace Texture {  
  
**+**/\*\*  
**+**\* 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 data テクスチャデータへのポインタ.  
**+**\*  
**+**\* @retval 0以外 作成したテクスチャ・オブジェクトのID.  
**+**\* @retval 0 テクスチャの作成に失敗.  
**+**\*/  
**+**GLuint CreateImage2D(GLsizei width, GLsizei height, const GLvoid\* data)  
**+**{  
**+** GLuint id;  
**+  
+** return id;  
**+**}  
**+**  
} // namespace Texture

雛形が書けたら、CreateImage2D関数の中に次のプログラムを追加してください。

GLuint CreateImage2D(GLsizei width, GLsizei height, const GLvoid\* data)  
{  
GLuint id;  
**+** glGenTextures(1, &id);  
**+** glBindTexture(GL\_TEXTURE\_2D, id);  
**+** glTexImage2D(GL\_TEXTURE\_2D,  
**+** 0, GL\_RGBA8, width, height, 0, GL\_RGBA, GL\_UNSIGNED\_BYTE, data);  
**+** const GLenum result = glGetError();  
**+** if (result != GL\_NO\_ERROR) {  
**+** std::cerr << "ERROR: テクスチャの作成に失敗(0x" << std::hex << result << ").";  
**+** glBindTexture(GL\_TEXTURE\_2D, 0);  
**+** glDeleteTextures(1, &id);  
**+** return 0;  
**+** }  
**+** glBindTexture(GL\_TEXTURE\_2D, 0); return id;  
}

OpenGLでテクスチャを作成するには、glGenTextures(じーえる・じぇん・てくすちゃーず)関数を使います。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を設定しています。2つめの引数は、作成したテクスチャ・オブジェクトのIDが格納される変数へのポインタです。

次に、作成したテクスチャ・オブジェクトをテクスチャ操作用のバインディング・ポイントに割り当てます。これにはglBindTexture(じーえる・ばいんど・てくすちゃ)関数を使います。最初の引数には、割り当て先となるバインディング・ポイントを指定します。OpenGLは一般的な二次元画像だけでなく、一次元や三次元、あるいはそれ以外の画像を扱うことができ、それぞれ専用のバインディング・ポイントが用意されています。ここでは、二次元画像用のバインディング・ポイントを指定しています。2つめの引数は、割り当てるテクスチャ・オブジェクトのIDです。0を設定した場合は、OpenGLに最初から用意されているテクスチャ(デフォルト・テクスチャといいます)が割り当てられます。事実上の「割り当て解除」と考えてもらって構いません。

**[補足]** デフォルト・テクスチャに画像を設定することは可能ですが、やってはいけません。デフォルト・テクスチャは、同時にひとつしかテクスチャを扱えなかったOpenGL 1.0時代の名残だからです。デフォルト・テクスチャは特別なので、glDeleteTexturesで削除することもできません。また、他のglGen系関数と同様に、glGenTextures関数はも絶対に0を返しません。これらの事実は、0番を「テクスチャが設定されていないことを示す数値」として扱うことを可能にします。0番をこの用途で使うためにも、デフォルト・テクスチャに何かを設定するべきではありません。

テクスチャを割り当てたら、glTexImage2D(じーえる・てっくす・いめーじ・つーでぃ)関数でテクスチャのパラメーターと画像データをGPU用メモリに転送します。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の宣言は次のとおりです。

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

***target*(たーげっと)**: 転送先となるバインディング・ポイントです。glBindTextureと同じ値を指定します。

***level*(れべる):** 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大きさの違うテクスチャを使い分ける機能のことです。画像が遠くに表示される場合は縮小して表示しなければなりませんが、事前に縮小済みの画像を用意しておけば、縮小にかかる時間を減らせるというわけです。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。

**[補足]** MIPはラテン語の「*multum in parvo*」の略称です。これは「小さな入れ物に、たくさんの物が入っている」というような意味だそうです。

***internalFormat*(いんたーなる・ふぉーまっと)**:  
画像がどのようにGPUメモリに格納されるかを指定します。一部の色成分だけを格納するデータ形式を指定した場合、格納されていない色成分を読み出した結果は0になります。例えばGL\_R8を指定して真っ赤なテクスチャを作ったとします。このテクスチャのRGB成分を読み出すと(1.0, 0.0, 0.0)になるということです。  
よく使われる形式はGL\_R8, GL\_RGBA8, GL\_RGBA16F, GL\_RGB565などです。

***width*(うぃす)**:  
テクスチャの幅をピクセル数で指定します。最小値は1です。最大値は実行環境次第ですが、OpenGL 4.0までは1024以上、OpenGL 4.1からは16384以上であることが保証されています。

***height*(はいと)**:  
テクスチャの高さをピクセル数で指定します。最小値と最大値はwidthと同じです。

***border*(ぼーだー)**:  
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以降は使用禁止になりました)。

***forma*t(ふぉーまっと)**:  
転送元の(CPUメモリにある)画像のデータ形式を指定します。internalFormatにない色成分が含まれる場合、その色要素は無視されます。例えばinternalFormatにGL\_R8、formatにGL\_RGBを指定した場合、画像のG成分とB成分は無視され、R成分だけが使われます。  
反対に、internalFormatがformatにない色成分を含んでいる場合、その色成分の値は0になります。例えばinternalFormatにGL\_RGB8、formatにGL\_REDを指定した場合、GとB成分は0になります。  
よく使われる形式はGL\_RED, GL\_RGBA, GL\_BGRAなどです。

***type*(たいぷ)**:  
転送元の(CPUメモリにある)画像が、どのように色を格納しているかを指定します。色の成分の格納方法によって、2通りの指定方法があります。ひとつが「各色成分の大きさ」を示す場合、もうひとつが、「すべての色成分を合計した大きさ」を示す場合です。例えば画像データの色成分が各色8ビット(=1バイト)で格納されている場合、GL\_UNSIGNED\_BYTEを指定します。色成分が各色5ビットで、それを16ビット(=2バイト)のデータに結合した値が格納されている場合、GL\_UNSIGNED\_SHORT\_5\_5\_5\_1を指定します。

***data*(でーた)**:  
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(あとからデータを描き込んで使用する場合に指定します)。

今回作成するテクスチャのデータ形式は、1ピクセルが赤、緑、青、不透明度の4要素からなっていて、各要素のビット数は8ですから、formatにはGL\_RGBA、typeにはUNSIGNED\_BYTEを指定しています。また、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。formatにGL\_BGRAを指定すると、B,G,R,Aの順序になります。internalFormatは、CPUメモリの形式と一致させたかったのでGL\_RGBA8を選択しました。CPUメモリとGPUメモリの形式を合わせておくと、データ形式を変換しなくて済むため転送が速く終わります。特に理由がなければ、形式は揃えておくといいでしょう。なお、formatとinternalFormtに指定できる値には似た名前が付いていますが、全く別のものです。混同しがちなので注意してください。

さて、glTexImage2D関数のあとは、glGetError(じーえる・げっと・えらー)関数で、テクスチャの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(じーえる・のー・えらー)が返されます。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、バインドを解除してテクスチャを削除したあと、0を返しています(0はデフォルト・テクスチャを指す番号でしたね)。ところで、エラーメッセージを出力するときに「std::hex(えすてぃーでぃー・へっくす)」というものを使っています。これは、「IO(あい・おー)マニピュレーター」と呼ばれるもので、「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(えすてぃーでぃー・でっく)」を使います。

glDeleteTextures(じーえる・でりーと・てくすちゃーず)関数は、テクスチャ・オブジェクトを削除する関数です。引数はglGenTextures関数と同じです。

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。  
エラー処理プログラムの下に、次のプログラムを追加してください。

if (result != GL\_NO\_ERROR) {  
 std::cerr << "ERROR: テクスチャの作成に失敗(0x" << std::hex << result << ").";  
glBindTexture(GL\_TEXTURE\_2D, 0);  
 glDeleteTextures(1, &id);  
 return 0;  
}  
 **+**// テクスチャのパラメータを設定する. **+**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);  
**+** glBindTexture(GL\_TEXTURE\_2D, 0);  
  
 return id;

パラメータを設定する関数にはglTexParameterf/glTexParamteri/glTexParamterfv/glTexParamteriv(じーえる・てっくす・ぱらめーた・[えふ/あい/えふ・ぶい/あい・ぶい])の4種類があり、パラメータの種類によって使い分けます。とはいえ、今回設定するパラメータはひとつの整数値で表現できるものばかりなのでglTexParameteriだけを使っています。少数の値を使うパラメータの場合はglTexParamterf、ひとつではなく複数の値を必要とする場合はglTexParamterfvやglTexParamterivを使います。

さて、glTexParamteri関数の最初の引数は、見てのとおりバインディング・ポイントの指定です。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、ここでは次に示す5種類を設定しています。

***GL\_TEXTURE\_MAX\_LEVEL*(じーえる・てくすちゃ・まっくす・れべる)**: 使用するミップマップの最大レベルです。初期値は1000です。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*(じーえる・てくすちゃ・みん・ふぃるたー)**: テクスチャを縮小表示する場合の読み取り方法です(minは「minify(小さくする)」の略です)。これには以下の6種類があります。

*GL\_NEAREST(じーえる・にあれすと)*  
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(じーえる・りにあ)*  
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(じーえる・にあれすと・みっぷまっぷ・にあれすと)*  
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(じーえる・りにあ・みっぷまっぷ・にあれすと)*  
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(じーえる・にあれすと・みっぷまっぷ・りにあ)*  
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(じーえる・りにあ・みっぷまっぷ・りにあ)*  
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です。名前にMIPMAPが含まれる値はミップマップを設定したテクスチャ用なので、ミップマップを使わない場合はGL\_NEARESTかGL\_LINEARに変更しなければなりません。

***GL\_TEXTURE\_MAG\_FILTER(じーえる・てくすちゃ・まぐ・ふぃるたー)***: テクスチャを拡大表示する場合の読み取り方法です(magは「」)。GL\_TEXTURE\_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。実はこのパラメータの初期値はGL\_LINEARなので、今回の場合は設定しなくても大丈夫です。しかし、GL\_TEXTURE\_MIN\_FILTERだけ変更してGL\_TEXTURE\_MAG\_FILTERの設定を忘れる、といったミスをしないようにペアで設定しておくほうがよいでしょう。

***GL\_TEXTURE\_WRAP\_S(じーえる・てくすちゃ・らっぷ・えす)***  
***GL\_TEXTURE\_WRAP\_T(じーえる・てくすちゃ・らっぷ・てぃー)***: 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(OpenGLではテクスチャ座標系の軸を表すのにXとYではなくSとTを使います)。それぞれ、次の4種類のうちから指定できます。

*GL\_CLAMP\_TO\_EDGE(じーえる・くらんぷ・とぅ・えっじ)*  
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(じーえる・くらんぷ・とぅ・ぼーだー)*  
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(じーえる・りぴーと)*  
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。パラメータを設定しない場合はこれが使われます。

*GL\_MIRRORED\_REPEAT(じーえる・みらーど・りぴーと)*  
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ので、GL\_CLAMP\_TO\_EDGEに変更しています。

パラメータを設定するプログラムを追加したら、テクスチャ作成関数は完成です。

## 画像データからテクスチャを作成する

それでは、Texture::CreateImage2D関数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Texture.hをインクルードしてください。

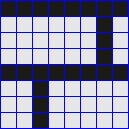
#include "GLFWEW.h"  
 #include "Shader.h"  
**+**#include "Texture.h"  
 #include <glm/gtc/matrix\_transform.hpp>  
 #include <iostream>

Texture.hをインクルードしたことで、先ほどプログラムしたテクスチャ作成関数が使えるようになります。

続いて、画像データを定義し、そのデータを使ってテクスチャを作成します。  
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。

const GLint matMVPLoc = glGetUniformLocation(shaderProgram, "matMVP");  
 if (matMVPLoc < 0) {  
 std::cerr << "ERROR: uniform変数'matMVP'の位置を取得できません.\n";  
 return 1;  
 }  
  
**+**// テクスチャを作成する.  
**+**const int imageWidth = 8; // 画像の幅.  
**+**const int imageHeight = 8; // 画像の高さ.  
**+**const uint32\_t B = 0xff'00'00'00; // 黒.  
**+**const uint32\_t W = 0xff'ff'ff'ff; // 白.  
**+**const uint32\_t imageData[imageWidth \* imageHeight] = {  
**+** W, W, B, W, W, W, W, W,  
**+** W, W, B, W, W, W, W, W,  
**+** W, W, B, W, W, W, W, W,  
**+** B, B, B, B, B, B, B, B,  
**+** W, W, W, W, W, W, B, W,  
**+** W, W, W, W, W, W, B, W,  
**+** W, W, W, W, W, W, B, W,  
**+** B, B, B, B, B, B, B, B,  
**+**};  
**+**GLuint texId = Texture::CreateImage2D(imageWidth, imageHeight, imageData);  
**+**if (!texId) {  
**+** return 1;  
**+**}  
**+**  
 // メインループ.  
 while (!window.ShouldClose()) {  
 glEnable(GL\_DEPTH\_TEST);

imageData(いめーじ・でーた)配列は幅8ピクセル、高さ8ピクセルの画像データです。1ピクセルは32ビットなので、データを定義するには0xffffffffのような数値を64回も書かなければなりません。それでは手間がかかりますし、見づらいプログラムになってしまいます。そこで数値に名前を付け、その名前でデータを作ることにしました。上記のプログラムでは、黒色にB(Blackの頭文字)、白色にW(Whiteの頭文字)と名付け、BとWで模様を描いています。なお、C++14から数値の区切り記号として「'(シングル・クォーテーション)」が使えるようになりましたので、これを使って桁数を把握しやすくしています。こうして定義したimageDataは次のような模様になります。



画像データの用意ができたので、これを使ってテクスチャを作りましょう。テクスチャを作成するには、画像の幅と高さ、それに画像データをTexture::CreateImage2D関数の引数に設定するだけです。戻り値が0以外の場合、その値は有効なテクスチャ・オブジェクトのIDです。  
作成に失敗した場合は0が返されますので、1を返して終了しています。

## テクスチャを削除する

作成したテクスチャ・オブジェクトが不要になったら削除しなければなりません。  
プログラム・オブジェクトを作成する処理の上に、次のプログラムを追加してください。

glUniformMatrix4fv(matMVPLoc, 1, GL\_FALSE, &matMVP[0][0]);  
 glDrawElementsBaseVertex(meshList[2].mode, meshList[2].count,  
 GL\_UNSIGNED\_SHORT, meshList[2].indices, meshList[2].baseVertex);  
 }  
  
 window.SwapBuffers();  
 }  
  
**+**glDeleteTextures(1, &texId);  
 glDeleteProgram(shaderProgram);  
 glDeleteVertexArrays(1, &vao);  
  
 return 0;

テクスチャ・オブジェクトを削除するにはglDeleteTextures関数を使うのでしたね。

# テクスチャ座標の追加

## テクスチャ座標

モデルにテクスチャを貼り付けるには、頂点にテクスチャ内の位置を割り当てなければなりません。これを「テクスチャ座標」といいます。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、追加していきましょう。

さて、二次元画像の場合、当然ですがテクスチャ座標も二次元(2D)になります。テクスチャ座標は「テクスチャ座標系」で指定します。OpenGLのテクスチャ座標系は画像の左下が原点(0,0)で、画像の右上が(1,1)になっています。  
テクスチャ座標系の各軸はU,VまたはS,Tと呼ばれます(X,Yではありません)。そのため「UV座標」とか「ST座標」と呼ばれることもあります。とはいえ、U=S=X, V=T=Yであって、UVもSTも意味はXYと同じなので、実際のプログラムでは2Dベクトルとして扱います。

**[補足]** テクスチャ座標系の軸がXYではないのは、頂点位置を示す座標と混同しないためです。W,X,Y,Zは頂点座標で使われているため、その直前にあるU,Vが選ばれました。UVとSTの違いは、単純に名前の規則としてどちらを使うか、ということです。今日、一般的にはUVが使われますが、GLSLではSTを使います。STが選ばれた理由には、「UVに続く文字が使用済みのため次元を増やせない」などいくつかあるのですが、コンピューターグラフィックスの歴史的な事情によるところが大きいです。

## 2Dベクトル型を追加する

そういうわけで、まずはテクスチャ座標用の2Dベクトル型を追加しましょう。  
Main.cppのVector3構造体の手前に、次のプログラムを追加してください。

#include "GLFWEW.h"  
#include "Shader.h"  
#include <glm/gtc/matrix\_transform.hpp>  
#include <iostream>  
  
**+**/// 2Dベクトル型. **+**struct Vector2  
**+**{  
**+** float x, y;  
**+**};  
**+**  
/// 3Dベクトル型. struct Vector3  
 {  
 float x, y, z;  
 };

## 頂点データ型にテクスチャ座標を追加する

次に、追加した2Dベクトル型を使って、テクスチャ座標を追加します。  
Vertex構造体に、次のプログラムを追加してください。

/// 頂点データ型. struct Vertex  
 {  
 Vector3 position; ///< 座標.  
 Color color; ///< 色.  
**+** Vector2 texCoord; ///< テクスチャ座標.  
 };

テクスチャ座標としてtexCoord(てっくす・こーど)メンバ変数を追加しました。なお、texCoordという名前は、テクスチャ座標の英語名texture coordinates(てくすちゃ・こーでぃねーつ)の略です。

## 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配列にある家の頂点データを、次のプログラムのように変更してください(やっていることは、末尾にテクスチャ座標を付け加えただけです)。

// 家  
**-** { { 2.8f, 0.0f, 3.0f}, {0.4f, 0.3f, 0.2f, 1.0f} },  
**-** { { 3.0f, 4.0f, 3.0f}, {0.6f, 0.5f, 0.3f, 1.0f} },  
**-** { { 0.0f, 6.0f, 3.0f}, {0.5f, 0.4f, 0.2f, 1.0f} },  
**-** { {-3.0f, 4.0f, 3.0f}, {0.6f, 0.5f, 0.3f, 1.0f} },  
**-** { {-2.8f, 0.0f, 3.0f}, {0.4f, 0.3f, 0.2f, 1.0f} },  
**-**  
**-** { {-2.8f, 0.0f,-3.0f}, {0.4f, 0.3f, 0.2f, 1.0f} },  
**-** { {-3.0f, 4.0f,-3.0f}, {0.6f, 0.5f, 0.3f, 1.0f} },  
**-** { { 0.0f, 6.0f,-3.0f}, {0.5f, 0.4f, 0.2f, 1.0f} },  
**-** { { 3.0f, 4.0f,-3.0f}, {0.6f, 0.5f, 0.3f, 1.0f} },  
**-** { { 2.8f, 0.0f,-3.0f}, {0.4f, 0.3f, 0.2f, 1.0f} },  
**-**  
**-** { { 3.0f, 4.0f, 3.0f}, {0.2f, 0.1f, 0.1f, 1.0f} },  
**-** { { 0.0f, 6.0f, 3.0f}, {0.3f, 0.2f, 0.2f, 1.0f} },  
**-** { {-3.0f, 4.0f, 3.0f}, {0.2f, 0.1f, 0.1f, 1.0f} },  
**-** { {-3.0f, 4.0f,-3.0f}, {0.2f, 0.1f, 0.1f, 1.0f} },  
**-** { { 0.0f, 6.0f,-3.0f}, {0.3f, 0.2f, 0.2f, 1.0f} },  
**-** { { 3.0f, 4.0f,-3.0f}, {0.2f, 0.1f, 0.1f, 1.0f} },  
**+** { { 2.8f, 0.0f, 3.0f}, {0.4f, 0.3f, 0.2f, 1.0f}, { 0.0f, 0.0f } },  
**+** { { 3.0f, 4.0f, 3.0f}, {0.6f, 0.5f, 0.3f, 1.0f}, { 0.0f, 0.7f } },  
**+** { { 0.0f, 6.0f, 3.0f}, {0.5f, 0.4f, 0.2f, 1.0f}, { 0.5f, 1.0f } },  
**+** { {-3.0f, 4.0f, 3.0f}, {0.6f, 0.5f, 0.3f, 1.0f}, { 1.0f, 0.7f } },  
**+** { {-2.8f, 0.0f, 3.0f}, {0.4f, 0.3f, 0.2f, 1.0f}, { 1.0f, 0.0f } },  
**+**  
**+** { {-2.8f, 0.0f,-3.0f}, {0.4f, 0.3f, 0.2f, 1.0f}, { 0.0f, 0.0f } },  
**+** { {-3.0f, 4.0f,-3.0f}, {0.6f, 0.5f, 0.3f, 1.0f}, { 0.0f, 0.7f } },  
**+** { { 0.0f, 6.0f,-3.0f}, {0.5f, 0.4f, 0.2f, 1.0f}, { 0.5f, 1.0f } },  
**+** { { 3.0f, 4.0f,-3.0f}, {0.6f, 0.5f, 0.3f, 1.0f}, { 1.0f, 0.7f } },  
**+** { { 2.8f, 0.0f,-3.0f}, {0.4f, 0.3f, 0.2f, 1.0f}, { 1.0f, 0.0f } },  
**+**  
**+** { { 3.0f, 4.0f, 3.0f}, {0.2f, 0.1f, 0.1f, 1.0f}, { 0.0f, 0.0f } },  
**+** { { 0.0f, 6.0f, 3.0f}, {0.3f, 0.2f, 0.2f, 1.0f}, { 0.5f, 0.0f } },  
**+** { {-3.0f, 4.0f, 3.0f}, {0.2f, 0.1f, 0.1f, 1.0f}, { 1.0f, 0.0f } },  
**+** { {-3.0f, 4.0f,-3.0f}, {0.2f, 0.1f, 0.1f, 1.0f}, { 1.0f, 1.0f } },  
**+** { { 0.0f, 6.0f,-3.0f}, {0.3f, 0.2f, 0.2f, 1.0f}, { 0.5f, 1.0f } },  
**+** { { 3.0f, 4.0f,-3.0f}, {0.2f, 0.1f, 0.1f, 1.0f}, { 0.0f, 1.0f } },  
  
 // 岩

本来は他のモデルの頂点データにもテクスチャ座標を追加しなければなりませんが、手間がかかるのでとりあえず家だけ設定することにしました。なお、設定していない頂点のテクスチャ座標は0になります。

これでテクスチャ座標の追加は完了です。

# シェーダーの変更

## 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。では、どうすればいいのかというと、テクスチャ座標をシェーダーに送り、シェーダーは送られた座標を使ってテクスチャを読み取るようにしなければなりません。  
テクスチャ座標がシェーダーに送られるようにするには、VAOを作成するときに、頂点データの中のテクスチャ座標データを、シェーダーのバインディング・ポイントに割り当てます。バインディング・ポイントへの割り当てにはglVertexAttribPointer関数を使うのでした。  
それでは、Main.cppのCreateVAO関数に、次のプログラムを追加してください。

GLuint CreateVAO(GLuint vbo, GLuint ibo)  
{  
 GLuint vao = 0;  
 glGenVertexArrays(1, &vao);  
 glBindVertexArray(vao);  
 glBindBuffer(GL\_ARRAY\_BUFFER, vbo);  
 glBindBuffer(GL\_ELEMENT\_ARRAY\_BUFFER, ibo);  
 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)));  
**+** glEnableVertexAttribArray(2);  
**+** glVertexAttribPointer(2, sizeof(Vertex::texCoord) / sizeof(float),  
**+** GL\_FLOAT, GL\_FALSE, sizeof(Vertex), (const　GLvoid\*)(offsetof(Vertex, texCoord)));  
 glBindVertexArray(0);  
 glDeleteBuffers(1, &vbo);  
 glDeleteBuffers(1, &ibo);  
 return vao;  
}

追加したプログラムでは、バインディング・ポイントの番号が2になっていることに気をつけてください。

続いて、シェーダーへの入力を追加します。頂点データが送られるのは頂点シェーダーです。次のように頂点シェーダーの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てください。

/// 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=2) in vec2 vTexCoord; \n"  
 "layout(location=0) out vec4 outColor; \n"  
 "uniform mat4x4 matMVP; \n"  
 "void main() { \n"

## シェーダーでテクスチャを読み込む

テクスチャ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、テクスチャ座標が頂点シェーダーまで届くようにしたので、今度はこれをフラグメントシェーダまで届けましょう。  
まず、頂点シェーダのoutColor変数定義の下に、出力用変数を追加してください。

/// 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=2) in vec2 vTexCoord; \n"  
 "layout(location=0) out vec4 outColor;"  
**+** "layout(location=1) out vec2 outTexCoord; \n"  
 "uniform mat4x4 matMVP; \n"  
 "void main() { \n"

次に、main関数内でvTexCoordをoutTexCoordにコピーします。  
vColorをoutColorにコピーしているプログラムの下に、次のプログラムを追加してください。

/// 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=2) in vec2 vTexCoord; \n"  
 "layout(location=0) out vec4 outColor;"  
"layout(location=1) out vec2 outTexCoord; \n"  
 "uniform mat4x4 matMVP; \n"  
"void main() { \n"  
 " outColor = vColor; \n"  
**+** " outTexCoord = vTexCoord; \n"  
 " gl\_Position = matMVP \* vec4(vPosition, 1.0); \n"  
 "}";

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に次のプログラムを追加してください。

/// 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"  
**+** "layout(location=1) in vec2 inTexCoord; \n"  
**+** "uniform sampler2D texColor; \n"  
 "out vec4 fragColor; \n"  
 "void main() { \n"

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sampler2D(さんぷらー・つーでぃ)は、「サンプラー」という「テクスチャ読み込み用オブジェクト」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が2Dテクスチャを読み込むためのサンプラー型です。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。2Dテクスチャを読み込むにはsampler2D、3Dテクスチャの場合はsampler3Dというように、テクスチャの種類によって使えるサンプラー型が決められています。

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関数を、次のように変更してください。

/// 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"  
"layout(location=1) in vec2 inTexCoord; \n"  
"uniform sampler2D texColor; \n"  
 "out vec4 fragColor; \n"  
 "void main() { \n"  
**-** " fragColor = inColor; \n"  
**+** " fragColor = inColor \* texture(texColor, inTexCoord); \n"  
 "}";

テクスチャを読み込むには、texture関数を使います。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、**サンプラーに割り当てられたテクスチャ**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。戻り値の型はvec4です。

ベクトル型の乗算は、以下に示すように、お互いの要素を個別に乗算した結果になります。

vec4 a = vec4(1, 2, 3, 4);  
vec4 b = vec4(8, 7, 6, 5);  
vec4 c = a \* b;  
vec4 d;  
d.x = a.x \* b.x;  
d.y = a.y \* b.y;  
d.z = a.z \* b.z;  
d.w = a.w \* b.w;  
このとき、変数cとdの値は等しい

これは乗算だけでなく、どの四則演算でも同じです。  
上記のプログラムでは、texture関数の戻り値とinColorを乗算しています。

# プリミティブにテクスチャを貼り付ける

## サンプラーの位置を取得する

テクスチャを作成し、テクスチャ座標を追加し、シェーダーも書き換えました。いよいよテクスチャを使う時です。

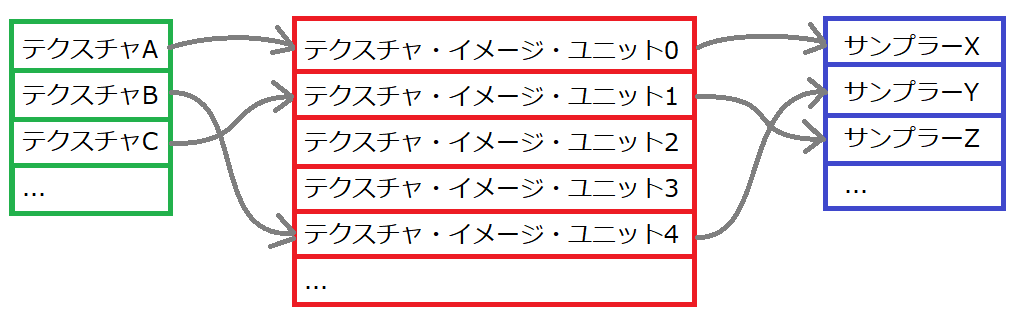
ところで、シェーダを書き換えるとき、「サンプラーに割り当てられたテクスチャ」と言いました。これを言い換えると、テクスチャを読み込むには、そのテクスチャをサンプラーに割り当てる必要がある、ということです。テクスチャをサンプラーに割り当てるには、割当先となるサンプラーが、シェーダー内の何番目のuniform変数なのかが分からなければなりません。サンプラーはuniform変数なので、行列のときと同じglGetUniformLocation関数で位置を調べることができます。

行列の位置を調べるプログラムの下に、次のプログラムを追加してください。

**+**// uniform変数の位置を取得.  
 const GLint matMVPLoc = glGetUniformLocation(shaderProgram, "matMVP");  
 if (matMVPLoc < 0) {  
 std::cerr << "ERROR: uniform変数'matMVP'の位置を取得できません.\n";  
 return 1;  
 }  
**+**glUseProgram(shaderProgram);  
**+**const GLint texColorLoc = glGetUniformLocation(shaderProgram, "texColor");  
**+**if (texColorLoc >= 0) {  
**+** glUniform1i(texColorLoc, 0);  
**+**}  
**+**glUseProgram(0);  
  
 // テクスチャを作成する.  
 const int imageWidth = 8; // 画像の幅.  
 const int imageHeight = 8; // 画像の高さ.

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(じーえる・ゆにふぉーむ・わん・あい)関数は、1つのint型の値をuniform変数に設定する関数です。サンプラーに使うと、サンプラーを指定した番号の「テクスチャ・イメージ・ユニット」に割り当てることができます。「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。テクスチャとサンプラー、そしてテクスチャ・イメージ・ユニットは、次の図で示すように別々に割り当てることができます。



OpenGLにおいて、作れるテクスチャの数を制限するのはメモリの量だけです。しかし、テクスチャ・イメージ・ユニットの数はGPUによって決まっています。加えて、シェーダーごとに使える数の制限があります。OpenGL 4.1以降、フラグメント・シェーダーでは少なくとも16個のテクスチャ・イメージ・ユニットが使えることになっています(GPUによっては16個以上使える場合があります)。この制限を超える数のテクスチャを使うときは、プログラムで適宜テクスチャを割り当て直すことになります。

なお、uniform変数に値を設定するには、設定するシェーダー・プログラムを使用中にしておく必要があります。そこで、サンプラーを割り当てる直前に対象のシェーダー・プログラムを使用中にして、割り当てが終わったら未使用に戻すようにしています。シェーダーを使用中にするには、glUseProgram(じーえる・ゆーず・ぷろぐらむ)関数を使います。未使用に戻すには、この関数の引数に0を設定します。

## テクスチャを割り当てる

テクスチャを表示するための最後の仕事は、テクスチャをテクスチャ・イメージ・ユニットに割り当てる作業です。  
木のモデルを表示するプログラムの手前に、次のプログラムを追加してください。

// 座標変換行列を作成する.  
 const glm::mat4x4 matProj =  
 glm::perspective(glm::radians(45.0f), 800.0f / 600.0f, 0.1f, 500.0f);  
 const glm::mat4x4 matView =  
 glm::lookAt(viewPos, glm::vec3(0, 0, 0), glm::vec3(0, 1, 0));  
  
 glBindVertexArray(vao);  
  
**+**glActiveTexture(GL\_TEXTURE0);  
**+**glBindTexture(GL\_TEXTURE\_2D, texId);  
**+**  
 const float treeCount = 10; // 木の本数.  
const float radius = 8; // 木を植える円の半径.  
 for (float i = 0; i < treeCount; ++i) {  
 const float x = std::cos(3.14f \* 2 / treeCount \* i) \* radius;  
 const float z = std::sin(3.14f \* 2 / treeCount \* i) \* radius;

glActiveTexture(じーえる・あくてぃぶ・てくすちゃ)関数は、指定したテクスチャ・イメージ・ユニットをテクスチャ関数の処理対象として設定する関数です。テクスチャ・イメージ・ユニットはGL\_TEXTURE0～GL\_TEXTURE31のいづれかの名前で指定します。この関数でテクスチャ・イメージ・ユニットを選んだあと、glBindTexture関数を使うと、指定したテクスチャが選択されたテクスチャ・イメージ・ユニットに割り当てられます。

## テクスチャの割り当てを解除する

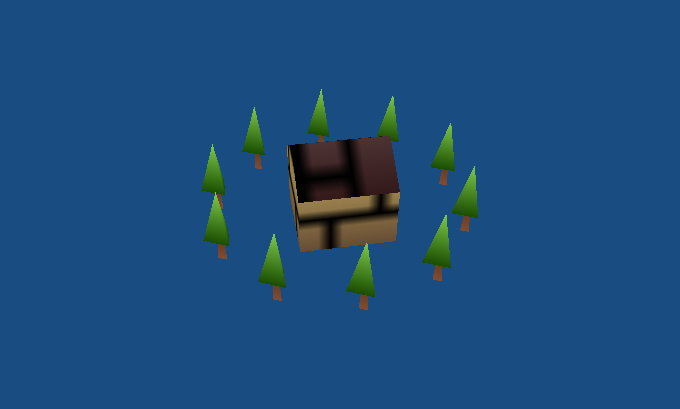
テクスチャを使い終わったら、すみやかに割り当てを解除します。理由は、テクスチャが割り当てられている場合、jそのテクスチャがシェーダーで実際に使われるかどうかに関わらず、GPUがテクスチャを読み込む準備処理を行う場合があるからです。これは無駄な処理時間なので、使わないテクスチャ・イメージ・ユニットにはデフォルト・テクスチャを割り当てておきます。

それでは、モデルを描画するプログラムの下に、次のプログラムを追加してください。

glUniformMatrix4fv(matMVPLoc, 1, GL\_FALSE, &matMVP[0][0]);  
 glDrawElementsBaseVertex(meshList[2].mode, meshList[2].count,  
 GL\_UNSIGNED\_SHORT, meshList[2].indices, meshList[2].baseVertex);  
 }  
  
**+** glActiveTexture(GL\_TEXTURE0);  
**+** glBindTexture(GL\_TEXTURE\_2D, 0);  
**+**  
 window.SwapBuffers();  
 }  
  
 glDeleteTextures(1, &texId);  
 glDeleteProgram(shaderProgram);  
 glDeleteVertexArrays(1, &vao);

割り当ての解除はglActiveTexture関数で解除するテクスチャ・イメージ・ユニットを選択して、glBindTexture関数に0を設定するだけです。テクスチャを作成するときに説明したように、0はデフォルト・テクスチャを示す番号なので、解除といっても実際には「テクスチャ・イメージ・ユニットにデフォルト・テクスチャを割り当てる」という動作になります。

プログラムが書けたら、**ビルドして実行してください。**次のような画像が表示されると思います。



## テクスチャ・フィルター

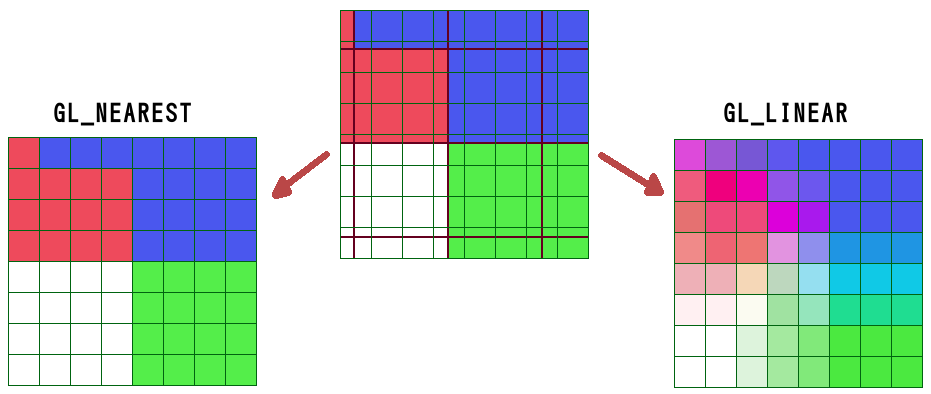
ここで、テクスチャのパラメータにどのような効果があるのか、実際に変更しながら確認していきます。  
まずはフィルタの設定から見ていきます。フィルタにはGL\_NEAREST(じーえる・にあれすと)かGL\_LINEAR(じーえる・りにあ)のいずれかを設定します。現在はGL\_LINEARになっているので、GL\_NEARESTに変更してみましょう。  
Texture.cppのCreateImage2D関数を、次のように変更してください。

// テクスチャのパラメータを設定する.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\_MIN\_FILTER, GL\_NEAREST);  
**+**glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MAG\_FILTER, GL\_NEAREST);  
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);

プログラムを変更したら、**ビルドして実行してください。**  
フィルタをGL\_NEAREST(じーえる・にあれすと)にしたことで、下記の画像のように色の境界がはっきりすると思います。



次の図は、フィルタの種類によってフラグメントの色がどう変わるかを示したものです。



太い枠線で区切られた矩形はテクスチャのピクセルを示しています。細い枠線はフラグメントを示しています。  
GL\_NEARESTは各フラグメントの中心が指している色だけが使われます。GL\_LINEARは、フラグメントに近い4つのピクセルを選び、それらのピクセルまでの距離の比で、色を混ぜ合わせます。2種類のフィルタの使い分けですが、基本的にはGL\_LINEAR(じーえる・りにあ)を使ってください。そして、GL\_LINEARではあまり綺麗な表示にならなかったときだけGL\_NEARESTを試してみるのがいいでしょう。

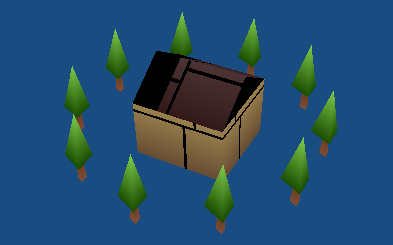
## 0.0～1.0の範囲をこえるテクスチャ座標

2.1節では「テクスチャ座標系は画像の左下が(0,0)、右上が(1,1)」と説明しました。では、テクスチャ座標に0未満や1より大きい数値を指定したらどうなるのでしょうか。

答えは「テクスチャのパラメータによる」です。試しに、パラメータを変えずに、家モデルのテクスチャ座標を変更したらどうなるかを見てみましょう。  
家モデルの頂点データを、次のように変更してください。

// 家  
**-** { { 2.8f, 0.0f, 3.0f}, {0.4f, 0.3f, 0.2f, 1.0f}, { 0.0f, 0.0f } },  
**-** { { 3.0f, 4.0f, 3.0f}, {0.6f, 0.5f, 0.3f, 1.0f}, { 0.0f, 0.7f } },  
**-** { { 0.0f, 6.0f, 3.0f}, {0.5f, 0.4f, 0.2f, 1.0f}, { 0.5f, 1.0f } },  
**-** { {-3.0f, 4.0f, 3.0f}, {0.6f, 0.5f, 0.3f, 1.0f}, { 1.0f, 0.7f } },  
**-** { {-2.8f, 0.0f, 3.0f}, {0.4f, 0.3f, 0.2f, 1.0f}, { 1.0f, 0.0f } },  
**-**  
**-** { {-2.8f, 0.0f,-3.0f}, {0.4f, 0.3f, 0.2f, 1.0f}, { 0.0f, 0.0f } },  
**-** { {-3.0f, 4.0f,-3.0f}, {0.6f, 0.5f, 0.3f, 1.0f}, { 0.0f, 0.7f } },  
**-** { { 0.0f, 6.0f,-3.0f}, {0.5f, 0.4f, 0.2f, 1.0f}, { 0.5f, 1.0f } },  
**-** { { 3.0f, 4.0f,-3.0f}, {0.6f, 0.5f, 0.3f, 1.0f}, { 1.0f, 0.7f } },  
**-** { { 2.8f, 0.0f,-3.0f}, {0.4f, 0.3f, 0.2f, 1.0f}, { 1.0f, 0.0f } },  
**-**  
**-** { { 3.0f, 4.0f, 3.0f}, {0.2f, 0.1f, 0.1f, 1.0f}, { 0.0f, 0.0f } },  
- { { 0.0f, 6.0f, 3.0f}, {0.3f, 0.2f, 0.2f, 1.0f}, { 0.5f, 0.0f } },  
**-** { {-3.0f, 4.0f, 3.0f}, {0.2f, 0.1f, 0.1f, 1.0f}, { 1.0f, 0.0f } },  
**-** { {-3.0f, 4.0f,-3.0f}, {0.2f, 0.1f, 0.1f, 1.0f}, { 1.0f, 1.0f } },  
**-** { { 0.0f, 6.0f,-3.0f}, {0.3f, 0.2f, 0.2f, 1.0f}, { 0.5f, 1.0f } },  
**-** { { 3.0f, 4.0f,-3.0f}, {0.2f, 0.1f, 0.1f, 1.0f}, { 0.0f, 1.0f } },  
**+** { { 2.8f, 0.0f, 3.0f}, {0.4f, 0.3f, 0.2f, 1.0f}, {-2.0f,-2.0f } },  
**+** { { 3.0f, 4.0f, 3.0f}, {0.6f, 0.5f, 0.3f, 1.0f}, {-2.0f, 0.8f } },  
**+** { { 0.0f, 6.0f, 3.0f}, {0.5f, 0.4f, 0.2f, 1.0f}, { 0.0f, 2.0f } },  
**+** { {-3.0f, 4.0f, 3.0f}, {0.6f, 0.5f, 0.3f, 1.0f}, { 2.0f, 0.8f } },  
**+** { {-2.8f, 0.0f, 3.0f}, {0.4f, 0.3f, 0.2f, 1.0f}, { 2.0f,-2.0f } },  
**+**  
**+** { {-2.8f, 0.0f,-3.0f}, {0.4f, 0.3f, 0.2f, 1.0f}, {-2.0f,-2.0f } },  
**+** { {-3.0f, 4.0f,-3.0f}, {0.6f, 0.5f, 0.3f, 1.0f}, {-2.0f, 0.8f } },  
**+** { { 0.0f, 6.0f,-3.0f}, {0.5f, 0.4f, 0.2f, 1.0f}, { 0.0f, 2.0f } },  
**+** { { 3.0f, 4.0f,-3.0f}, {0.6f, 0.5f, 0.3f, 1.0f}, { 2.0f, 0.8f } },  
**+** { { 2.8f, 0.0f,-3.0f}, {0.4f, 0.3f, 0.2f, 1.0f}, { 2.0f,-2.0f } },  
**+**  
**+** { { 3.0f, 4.0f, 3.0f}, {0.2f, 0.1f, 0.1f, 1.0f}, {-2.0f,-2.0f } },  
**+** { { 0.0f, 6.0f, 3.0f}, {0.3f, 0.2f, 0.2f, 1.0f}, { 0.0f,-2.0f } },  
**+** { {-3.0f, 4.0f, 3.0f}, {0.2f, 0.1f, 0.1f, 1.0f}, { 2.0f,-2.0f } },  
**+** { {-3.0f, 4.0f,-3.0f}, {0.2f, 0.1f, 0.1f, 1.0f}, { 2.0f, 2.0f } },  
**+** { { 0.0f, 6.0f,-3.0f}, {0.3f, 0.2f, 0.2f, 1.0f}, { 0.0f, 2.0f } },  
**+** { { 3.0f, 4.0f,-3.0f}, {0.2f, 0.1f, 0.1f, 1.0f}, {-2.0f, 2.0f } },  
  
 // 岩

テクスチャ座標を4倍して2を引いてみました。  
変更したら、**ビルドして実行してください。**以下のような画像が表示されたら成功です。

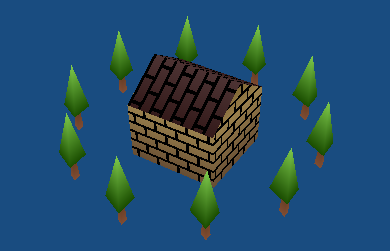


なんだか奇妙な表示ですが、これはラップ・モードGL\_CLAMP\_TO\_EDGE(じーえる・くらんぷ・とぅ・えっじ)の効果です。

さて、今度はテクスチャ作成関数を次のように変更してください。

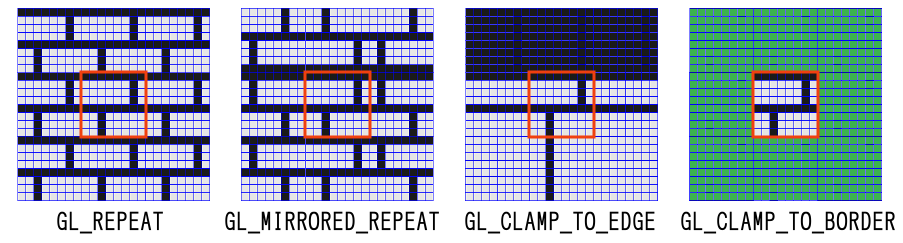
// テクスチャのパラメータを設定する.glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MAX\_LEVEL, 0);  
 glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MIN\_FILTER, GL\_NEAREST);  
 glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MAG\_FILTER, GL\_NEAREST);  
**-**glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_S, GL\_CLAMP\_TO\_EDGE);  
**-**glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_T, GL\_CLAMP\_TO\_EDGE);  
**+**glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_S, GL\_REPEAT);  
**+**glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_T, GL\_REPEAT);  
glBindTexture(GL\_TEXTURE\_2D, 0);

変更したら、**ビルドして実行してください。**以下のような画像が表示されたら成功です。



レンガ造りの家のようになりましたね。これがGL\_REPEATの効果です。

テクスチャ座標が0.0～1.0をこえる場合、次の図で示すようにラップ・モードによって異なる見た目になります。



上図の赤枠が0.0～1.0の範囲です。テクスチャ座標がこの範囲の外側にあるとき、ラップ・モードがGL\_REPEATのときは同じ画像が無限に繰り返されます。GL\_MIRRORED\_REPEATの場合、鏡に写したような画像が無限に繰り返されます。GL\_CLAMP\_TO\_EDGEでは、画像の端のピクセルだけが無限に繰り返されます。そして、GL\_CLAMP\_TO\_BORDERは、「ボーダー・カラー」という別のパラメータで設定した色になります。  
ボーダー・カラーを設定するにはglTexParameterfv関数の第2引数にGL\_TEXTURE\_BORDER\_COLOR(じーえる・てくすちゃ・ぼーだー・からー)という値を設定し、第3引数にRGBAの順で色データを格納したfloat型配列へのポインタを設定します。

**[課題01]** テクスチャの模様を変更してください。

**[課題02]** 木と岩の頂点データに、テクスチャ座標を設定してください。

**[課題03]** 木のモデルのためのテクスチャを作成して、そのテクスチャを使って木を表示してください。