[OpenGL講義 第05回]

テクスチャ・ファイル・パワー！メイク・アップ！

# テクスチャ読み込み関数を作る

## 画像ファイルの形式を選ぶ

これ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、比較的簡単にテクスチャにできるTGA(てぃー・じー・えー)という形式のファイルを使うことにしました。TGAは複数のデータ形式を扱えますが、最初は一番簡単な「圧縮なし32ビット形式」を読み込めるようにして、それから残りの形式にも対応していくことにしましょう。

**[補足]** TGAは「**T**ruevision **G**raphics **A**dapter(とぅるー・びじょん・ぐらふぃっくす・あだぷたー)」の略です。

## 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。  
CreateImage2D関数の宣言の下に、次のプログラムを追加してください。

namespace Texture {  
  
 GLuint CreateImage2D(GLsizei width, GLsizei height, const GLvoid\* data);  
+GLuint LoadImage2D(const char\* path);  
  
 } // namespace Texture

## ヘッダファイルのインクルード

それでは、LoadImage2D関数を定義していきましょう。まずは、ファイル操作で使うヘッダファイルをインクルードします。Texture.cppを開き、次のプログラムを追加してください。

#include "Texture.h"  
+#include <stdint.h>  
+#include <vector>  
+#include <fstream>  
 #include <iostream>

stdint.h(えすてぃーでぃー・いんと・へっだ)は、ビット数が指定された整数型を定義しているヘッダファイルです。そのような整数型は、画像データのように厳密にビット数が一致することが要求される処理で使います。  
vector(べくたー)は読み込みバッファを作るためにインクルードしています。  
fstream(えふ・すとりーむ)はファイル操作用のクラスを定義しているヘッダファイルです。ファイル名の先頭にある「f」は「file(ふぁいる)」の頭文字です。

## TGAフォーマットの構造

TGAファイルの先頭には、画像の情報が格納されている部分があります。  
これをTGAファイルヘッダといい、次のような構造になっています。

|  |  |  |  |
| --- | --- | --- | --- |
| **名前** | **オフセット** | **バイト数** | **格納されている情報** |
| ID Length (あいでぃー・れんぐす) | 0 | 1 | イメージIDデータのバイト数. |
| Colormap Type (からーまっぷ・たいぷ) | 1 | 1 | カラーマップの有無. 0 = 無 1 = 有 |
| Image Type (いめーじ・たいぷ) | 2 | 1 | 画像記録形式. 0 = 画像データ無し 1 = 圧縮なし、インデックス・カラー画像 2 = 圧縮なし、カラー画像 3 = 圧縮なし、白黒画像 9 = 圧縮あり、インデックス・カラー画像 10 = 圧縮あり、カラー画像 11 = 圧縮あり、白黒画像 |
| First Entry Index (ふぁーすと・えんとりー・ いんでっくす) | 3 | 2 | 画像データのインデックス0に対応するカラーマップのインデックス. カラーマップ無しの場合は未使用(常に0). |
| Colormap Length (からーまっぷ・れんぐす) | 5 | 2 | カラーマップに登録されている色の数. カラーマップ無しの場合は未使用(常に0). |
| Colormap Entry Size (からーまっぷ・ えんとりー・さいず) | 7 | 1 | カラーマップに登録されている色ひとつのビット数. カラーマップ無しの場合は未使用(常に0). |
| X-origin (えっくす・おりじん) | 8 | 2 | X座標をずらすピクセル数. |
| Y-origin (わい・おりじん) | 10 | 2 | Y座標をずらすピクセル数. |
| Image Width (いめーじ・うぃす) | 12 | 2 | 画像の横のピクセル数 |
| Image Height (いめーじ・はいと) | 14 | 2 | 画像の縦のピクセル数 |
| Pixel Depth (ぴくせる・でぷす) | 16 | 1 | 画像の1ピクセルのビット数. |
| Image Descriptor (いめーじ・ですくりぷた) | 17 | 1 | 画像の格納方向および1ピクセルのアルファ要素のビット数. |

TGAファイルでは、この18バイトのヘッダの直後に、イメージID(画像の種類などを示す文字列)、カラーマップ(インデックス・カラー形式の画像で使われるパレット)が続き、さらにその直後に画像データが格納されています。  
ファイルによってはイメージIDやカラーマップが存在せず、いきなり画像データが格納されていることもあります。

**[補足]** インデックス・カラーは、色に番号をつけておき、ピクセルの色を直接記録するかわりに、その番号を記録する形式です。普通の画像では1ピクセルに16～32ビットのデータが必要となりますが、番号をつける色の数を256種類だけに限定した場合、1ピクセルに8ビットあれば画像を記録できるので、データ量を小さくすることができるのです。もちろん、画像に必要な色数が256種類で足りる場合に限りますが。

## TGAヘッダを読み込む

なんとなくTGAファイルの構造が分かったところで、読み込み関数の作成を始めましょう。  
CreateImage2D関数定義の下に、次のプログラムを追加してください。

return id;  
 }  
  
+/\*\*  
+\* ファイルから2Dテクスチャを読み込む.  
+\*  
+\* @param path 2Dテクスチャとして読み込むファイル名.  
+\*  
+\* @retval 0以外 作成したテクスチャ・オブジェクトのID.  
+\* 0 テクスチャの作成に失敗.  
+\*/  
+GLuint LoadImage2D(const char\* path)  
+{  
+ // TGAヘッダを読み込む.  
+ std::basic\_ifstream<uint8\_t> ifs;  
+ ifs.open(path, std::ios\_base::binary);  
+ uint8\_t tgaHeader[18];  
+ ifs.read(tgaHeader, 18);  
+  
+}  
+  
 } // namespace Texture

C++でファイルを読み込むにはstd::basic\_ifstream(えすてぃーでぃー・べーしっく・あい・えふ・すとりーむ)というクラスを使います。このクラスはstd::vectorクラスと同様のクラス・テンプレートで、テンプレート・パラメータで読み書きするデータの種類を指定できるようになっています。今回は「uint8\_t(ゆー・いんと・はち・てぃー)」型を指定してます。これは「8ビットの符号なし(**u**nsigned)整数(**int**eger)型(**t**ype)」を意味します。

ファイルを開くにはopen(おーぷん)メンバ関数を使います。最初の引数は「開くファイル名」です。引数のpath(ぱす)には読み込むファイル名が格納されているので、これを設定しています(pathには「経路、道筋」といった意味があります)。2つめの引数によって、開いたファイルの扱い方を指定します。この引数に設定できるのはstd::ios\_base::openmode(えすてぃーでぃー・あい・おー・えす・べーす・おーぷん・もーど)列挙型の値の組み合わせです。  
この列挙型には以下のメンバが用意されています。

|  |  |
| --- | --- |
| In(いん) | ファイルを読み込み用に開きます。ifstreamでは自動的に付与されます。 |
| out(あうと) | ファイルを書き込み用に開きます。 |
| ate(あっと・えんど) | 読み書き位置をファイルの末尾に移動します。 |
| app(あぺんど) | 書き込みが常にファイルの末尾に対して行われるようにします。 |
| trunc(とらんく) | ファイルを開いた直後に、内容を全て削除します。 |
| binary(ばいなり) | ファイルをバイナリ・ファイルとして操作します。 |

この列挙型にはいろいろな値が定義されていますが、よく使うのはbinaryくらいです。それ以外の値を使う機会はあまりないでしょう。

さて、C/C++でファイルを操作するときは、そのファイルが「テキスト・ファイル」なのか「バイナリ・ファイル」なのかを指定しなければなりません。何もしなければ全てのファイルは「テキスト・ファイル」として扱われますので、バイナリ・ファイルとして扱いたいときはstd::ios\_base::binaryを指定する必要があります。

「テキスト・ファイル」は人間が直接読み書きするためのものです。ファイルの内容は行単位で扱われ、各行は改行記号によって区切られます。そのため、「行単位で読み込む」といった操作が可能になります。基本的には、人間が読める文字(および改行記号などの制御記号)だけが記録されます。このため、数値も文字列で表され、例えば数値の「10000」を表すために5バイトのデータが必要になります。

「バイナリ・ファイル」は人間ではなくコンピューターが直接扱うためのものです。ですから、データとして表せるあらゆるものを含むことができます。改行記号も他の文字と同じく単なるビット列として扱われます。ですから、例えば数値の「10000」を表すのに必要なデータ量は2バイトだけです。

ただし、「テキスト・ファイルに文字として認識されないデータを含めてはならない」というルールはありません。この分類は人間がテキスト・データを扱いやすくするためのもので、コンピューターにとって二種類のファイルに違いはありません。実際、プログラムにおいてファイルの種類が影響するのは「改行の扱い」だけです。

ファイルを読み込むにはread(りーど)メンバ関数を使います。最初の引数は読み込み先のバッファへのポインタで、次の引数が読み込むデータの数となっています。今回はuint8\_t型を使っているので「データ数 = バイト数」になりますね。TGAヘッダは18バイトなので、この引数にも18を指定しています。

## イメージID領域をスキップする

TGAヘッダと画像データの間には、イメージIDとカラーマップという2つのデータが存在する可能性があります。  
当面はどちらも不要なので、読み飛ばすことにしましょう。  
TGAヘッダを読み込むプログラムの下に、次のプログラムを追加してください。

std::basic\_ifstream<uint8\_t> ifs;  
 ifs.open(path, std::ios\_base::binary);  
 uint8\_t tgaHeader[18];  
 ifs.read(tgaHeader, 18);  
  
+ // イメージIDを飛ばす.  
+ ifs.ignore(tgaHeader[0]);  
+  
 }  
  
 } // namespace Texture

ファイルの一部を読み飛ばすにはignore(いぐのあ)メンバ関数を使います。引数として「読み飛ばすデータ数」を設定すると、その数だけデータを読み飛ばしてくれます。イメージIDの長さはTGAヘッダの0バイト目に格納されていますので、それを引数に設定するだけです。

## カラーマップをスキップする

続いてカラーマップを読み飛ばします。イメージIDをスキップ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。

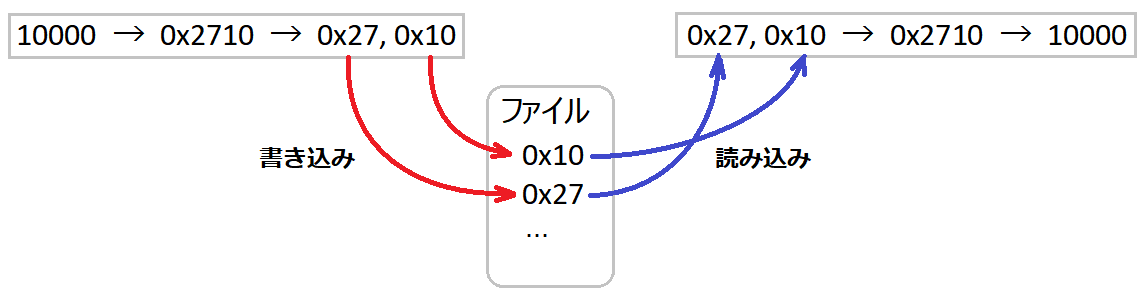
// イメージIDを飛ばす.  
 ifs.ignore(tgaHeader[0]);  
  
+ // カラーマップを飛ばす.  
+ if (tgaHeader[1]) {  
+ const int colorMapLength = tgaHeader[5] + tgaHeader[6] \* 0x100;  
+ const int colorMapEntrySize = tgaHeader[7];  
+ const int colorMapSize = colorMapLength \* colorMapEntrySize / 8;  
+ ifs.ignore(colorMapSize);  
+ }  
+  
 }  
  
 } // namespace Texture

カラーマップの有無は、TGAヘッダの1バイト目を見れば分かります。ここに格納されている数値が0なら、カラーマップはありませんので、何もする必要はありません。1なら、カラーマップのバイト数を計算して読み飛ばす必要があります。カラーマップのバイト数は、TGAヘッダにあるカラーマップ・レングスとカラーマップ・エントリー・サイズという2つのデータから計算することができます。

ところで、TGAファイルヘッダにはバイト単位で情報が詰め込まれています。ですから、データの読み出しもバイト単位で行わなければなりません。カラーマップ・エントリー・サイズは1バイトのデータなのでそのまま使うことができますが、カラーマップ・レングスは2バイトです。2バイトのデータってどう扱ったらいいんでしょう？  
これは、データがどうやって格納されているかが分かれば大丈夫です。TGAファイルの場合、数値の格納方法には「リトル・エンディアン」という方法を使っています。リトル・エンディアンは、数値データをバイト単位に分けて、桁の小さい方から1バイトずつ順番にファイルに格納する方法です。これを数値に戻すには、1バイトずつ取り出して元の順番に戻します。

例として10000という数値を考えてみましょう。10000を16進数にすると0x2710となります(「0x」は16進数であることを示すために付けています)。まず、これを0x2700と0x10の2つの部分に分けます。0x2700を0x100で割ると0x27になります。これで10000を2つの1バイトデータに分けることができました。リトル・エンディアンは小さい桁側から格納していく方法なので、ファイルには0x10, 0x27の順番で格納されます。

次に、このファイルをuint8\_t型の配列に読み込んだとしましょう。配列名をarrayとすると、array[0]には0x10、array[1]には0x27が格納されることになります。この2つから元の0x2710を作り出すには、2番めの0x27を0x100倍して足し合わせるだけです。



これで、カラーマップに登録されている色数と、ひとつの色のビット数が分かりました。あとはこのふたつを掛け合わせるだけです。ただし、カラーマップ・エントリー・サイズに格納されているのは色のビット数なので、8で割ってバイト数に変換する必要があります。

バイト数が計算できればあとは簡単です。イメージIDのときと同様にignore(いぐのあ)メンバ関数を使ってカラーマップを読み飛ばすだけです。

**[補足]** 通常は「圧縮なし32ビット」形式にカラーマップは存在しません。

## 画像データを読み込む

カラーマップの後ろには画像データがあります。これを読み込みましょう。  
カラーマップをスキップ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。

// カラーマップを飛ばす.  
 if (tgaHeader[1]) {  
 const int colorMapLength = tgaHeader[5] + tgaHeader[6] \* 0x100;  
 const int colorMapEntrySize = tgaHeader[7];  
 const int colorMapSize = colorMapLength \* colorMapEntrySize / 8;  
 ifs.ignore(colorMapSize);  
 }  
  
+ // 画像データを読み込む.  
+ const int width = tgaHeader[12] + tgaHeader[13] \* 0x100;  
+ const int height = tgaHeader[14] + tgaHeader[15] \* 0x100;  
+ const int pixelDepth = tgaHeader[16];  
+ const int imageSize = width \* height \* pixelDepth / 8;  
+ std::vector<uint8\_t> buf(imageSize);  
+ ifs.read(buf.data(), imageSize);  
+  
 }  
  
 } // namespace Texture

画像データのバイト数は、TGAヘッダのイメージ・ウィスとイメージ・ハイト、それにピクセル・デプスの3つの数値から計算できます。イメージ・ウィスとイメージ・ハイトは2バイトのデータなので、カラーマップ・レングスと同じ方法で数値に戻します。ピクセル・デプスもカラーマップ・エントリー・サイズと同様にビット数なので8で割ってバイト数にします。あとは3つを掛け合わせるだけです。

画像データはstd::vector<uint8\_t>型の変数に読み込みます。変数名はbuf(ばふ)としました(buffer(ばっふぁ)の略)。さて、画像データを読み込むには、読み込み先となるbufの大きさを、画像データのバイト数に合わせなければなりません。いくつか方法があるのですが、今回はコンストラクタに大きさを指定する方法を使いました。このコンストラクタは、vectorを引数で指定された大きさに変更します。  
buf変数を作ったら、readメンバ関数を使って画像データを読み込みます。

**[補足]** std::vector型は、newやdeleteを書かなくても自由な大きさの配列を扱えるので、とても便利です。積極的に使っていきましょう。

## 画像データからテクスチャを作成する

最後に、読み込んだデータからテクスチャを作成します。  
画像データを読み込むプログラムの下に、次のプログラムを追加してください。

// 画像データを読み込む.  
 const int width = tgaHeader[12] + tgaHeader[13] \* 0x100;  
 const int height = tgaHeader[14] + tgaHeader[15] \* 0x100;  
 const int pixelDepth = tgaHeader[16];  
 const int imageSize = width \* height \* pixelDepth / 8;  
 std::vector<uint8\_t> buf(imageSize);  
 ifs.read(buf.data(), imageSize);  
  
+ // 読み込んだ画像データからテクスチャを作成する.  
+ return CreateImage2D(width, height, buf.data());  
 }  
  
 } // namespace Texture

テクスチャの作成はCreateImage2D関数を呼び出すだけです。必要なデータは揃っているので、それらを引数に設定するだけの簡単なお仕事ですね。

ひとまずは、テクスチャ・ファイルを読み込むプログラムは完成とします。

# TGAテクスチャの作成

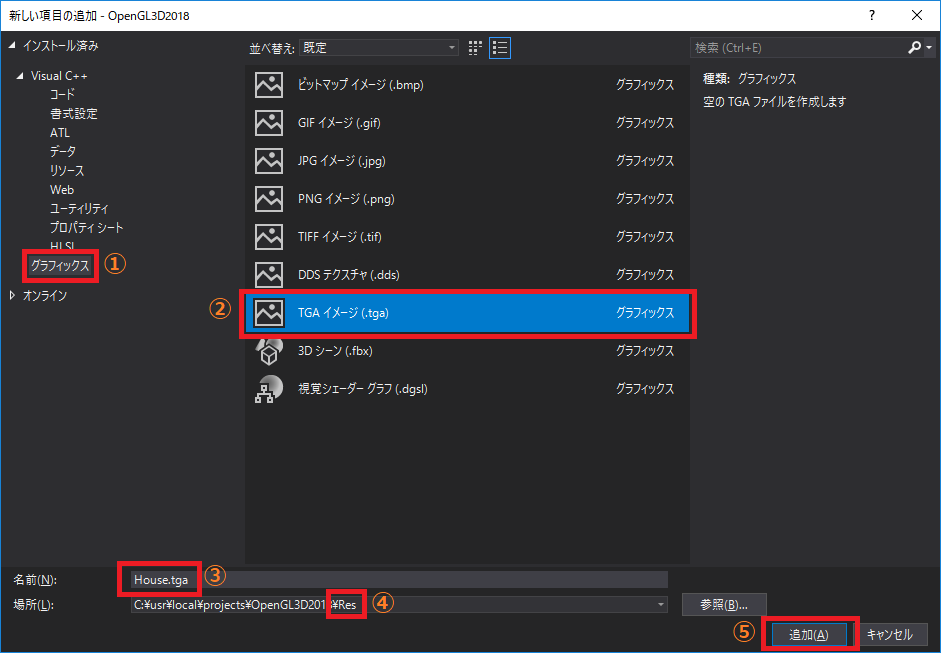
## TGAテクスチャを作るには

前章ではTGAファイルを読み込めるようにしました。あとは関数を呼び出すだけですが、肝心のTGAファイルがないと意味がありません。そこで、簡単なTGAファイルを作ってみましょう。

画像をTGAファイルとして保存する機能は、ほとんどの2D画像作成アプリケーションに備わっています。  
例えばGIMPの場合、画像にアルファ・チャンネルがある状態にして「エクスポート」を選択し、拡張子を「tga」に変えて「Export」ボタンを押します。エクスポート・オプションが開くので、「RLE complession」のチェックを外し、もう一度「Export」ボタンを押せば、「圧縮なし32ビット形式」のTGAファイルとして保存されます。

## Visual StudioでTGAファイルを作る

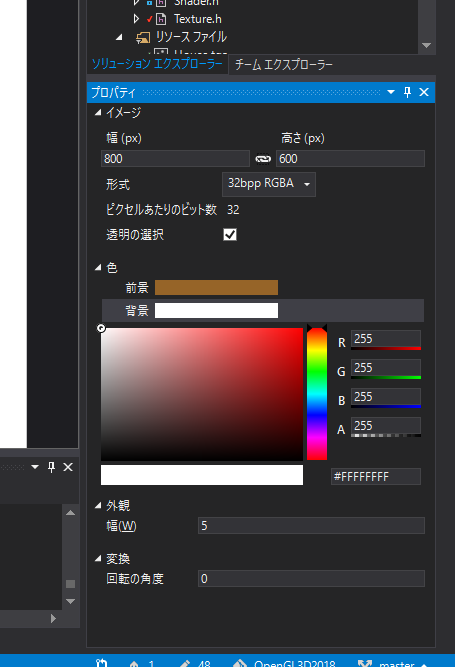
GIMP以外で手っ取り早いのは、VIsual Studioの画像編集機能を使うことです。基本的な機能しか備えていないため、高度な画像編集をすることはできませんが、動作確認用のテクスチャを作成する程度ならば十分です。ちょっと作成してみましょう。  
ソリューション・エクスプローラーの「リソースファイル」フィルタの上で右クリック、「追加→新しい項目」を選択して「新しい項目の追加」ウィンドウを開いてください。



そうしたら、左側のリストから「グラフィックス」を選択して(①)、次に中央のリストの「TGAイメージ(.tga)」を選びます(②)。名前テキストボックスでファイル名を「House.tga」にします(③)。場所テキストボックスはプロジェクト・フォルダを指しているかプロジェクトのSrcフォルダを指していると思いますので、これを「プロジェクト・フォルダ￥Res」となるように変更します。上図のプロジェクト・フォルダは「C:￥usr￥local￥projects￥OpenGL3D2018」となっていますが、これは筆者の環境がそうなっているだけですので、皆さん自身のプロジェクト・フォルダ名で置き換えてください。最後に、「追加」ボタンを押すと、TGAファイルが追加されます。

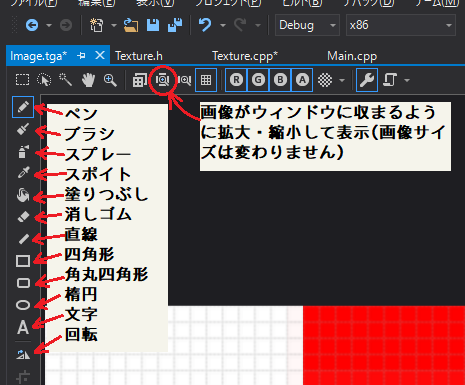
## TGAファイルを編集する

作成したばかりのTGAファイルは800x600ピクセルの白い画像になっています。最初から大きな画像を編集するのは大変なので小さくしましょう。  
「表示」メニューをクリックし「その他のウィンドウ」にある「プロパティウィンドウ」を選択してください。  
すると、下図のようなウィンドウが開きます(既に開いている場合はこの操作は不要です)。

最初に、幅と高さの中間にある鎖の記号をクリックして、鎖が外れた状態にしてください。こうすることで、幅と高さを個別に変更できるようになります。

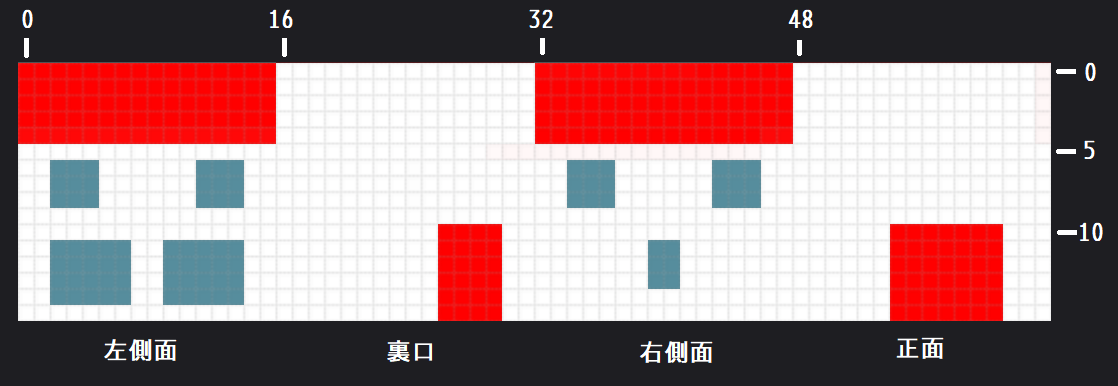
次に幅を64、高さを16にします。変更を確定するには、テキストボックスとは別の場所をクリックするか、Enterキーを押してください。

大きさを編集したら画像編集を始めましょう。編集のためのツールは画像ウィンドウの左上にあります。

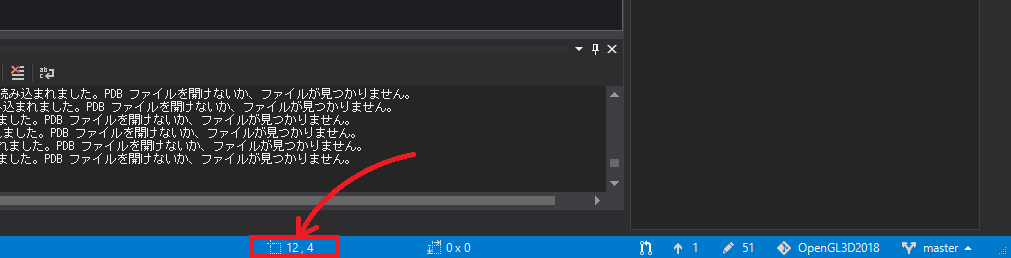
横一列に並んだアイコンの中央付近に、四方を壁で囲まれた虫眼鏡アイコンがありますので、まずはこれをクリックしてください。すると、画像がウィンドウいっぱいに拡大表示されます。

画像を編集するには縦一列に並んだアイコンを使います。色やペンの太さはプロパティ・ウィンドウで変えることができます。  
なお、「四角形」ツールや「楕円」ツールは輪郭しか描けませんので、内側を塗りつぶしたいときは「塗りつぶし」ツールを併用してください。

それでは、ツールを使って以下のような画像を作成してください。  
白い部分が壁、赤い部分が屋根と扉、灰色の部分が窓のつもりで描きましたが、まあ適当に描いてもらって構いません。



カーソルの位置はVisual Studioの一番下にある「ステータス・バー」に表示されます。編集の際は参考にしてください。また、カーソル位置の右にある数値は、選択範囲の大きさです。範囲選択ツールの使用時に数値が変化します。



そうそう、画像を編集したら保存を忘れないように。

**[補足]** GIMPやPhotoShopなどを使って家のテクスチャ用の画像を作成する場合は、大きさも内容も自由に作ってもらって構いません。

# TGAファイルを読み込む

## 用意したTGAファイルを読み込む

2章で用意したTGAファイルを読み込みます。  
Main.cppを開き、テクスチャ作成プログラムの下に、次のプログラムを追加してください。

const GLuint imageData[tw \* th] = {  
 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(tw, th, imageData);  
+ GLuint texHouse = Texture::LoadImage2D("Res/House.tga");  
  
 // メインループ.  
 while (!window.ShouldClose()) {  
 glEnable(GL\_DEPTH\_TEST);

うまくプログラムが動いてくれれば、texHouseに家のテクスチャ・オブジェクトのIDが格納されるはずです。  
続いて、テクスチャの切り替えを行います。家のモデルを表示するプログラムに、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[0].mode, meshList[0].count,  
 GL\_UNSIGNED\_SHORT, meshList[0].indices, meshList[0].baseVertex);  
 }  
  
 {  
+ glActiveTexture(GL\_TEXTURE0);  
+ glBindTexture(GL\_TEXTURE\_2D, texHouse);  
+  
 const glm::mat4x4 matModel = glm::translate(glm::mat4(1), glm::vec3(0, 0, 0));  
 const glm::mat4x4 matMVP = matProj \* matView \* matModel;  
 glUniformMatrix4fv(matMVPLoc, 1, GL\_FALSE, &matMVP[0][0]);  
 glDrawElementsBaseVertex(meshList[1].mode, meshList[1].count,  
 GL\_UNSIGNED\_SHORT, meshList[1].indices, meshList[1].baseVertex);  
 }

最後に、作成したテクスチャを破棄します。  
メインループの下に、次のプログラムを追加してください。

window.SwapBuffers();  
 }  
  
+glDeleteTextures(1, &texHouse);  
 glDeleteTextures(1, &texId);  
 glDeleteProgram(shaderProgram);  
 glDeleteVertexArrays(1, &vao);

プログラムが書けたら**ビルドして実行してください。**  
TGAファイルが家のテクスチャとして使われるはずです。

## 家のモデルを修正する(その1)

テクスチャは表示されたものの、思ったような絵にはなっていないと思います。なぜなら、新しいテクスチャを用意したのに、テクスチャ座標は何も変えていないからです。そこで、テクスチャ座標を修正しましょう。  
vertices配列にある家の頂点データを、次のように変更してください。なお、変更を行単位で示す都合で全て書き換えているように見えますが、実際に変更しているのはテクスチャ座標だけですので、そこだけ修正すれば大丈夫です。実際に変更した部分の背景は黄色にしてあります。

// 家  
- { { 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 } },  
  
+ { { 2.8f, 0.0f, 3.0f}, {0.4f, 0.3f, 0.2f, 1.0f}, { 0.0f, 1.0f } },  
+ { { 3.0f, 4.0f, 3.0f}, {0.6f, 0.5f, 0.3f, 1.0f}, { 0.0f, 0.31f } },  
+ { { 0.0f, 6.0f, 3.0f}, {0.5f, 0.4f, 0.2f, 1.0f}, { 0.125f, 0.0f } },  
+ { {-3.0f, 4.0f, 3.0f}, {0.6f, 0.5f, 0.3f, 1.0f}, { 0.25f, 0.31f } },  
+ { {-2.8f, 0.0f, 3.0f}, {0.4f, 0.3f, 0.2f, 1.0f}, { 0.25f, 1.0f } },  
+  
+ { {-2.8f, 0.0f,-3.0f}, {0.4f, 0.3f, 0.2f, 1.0f}, { 0.5f, 1.0f } },  
+ { {-3.0f, 4.0f,-3.0f}, {0.6f, 0.5f, 0.3f, 1.0f}, { 0.5f, 0.31f } },  
+ { { 0.0f, 6.0f,-3.0f}, {0.5f, 0.4f, 0.2f, 1.0f}, { 0.625f, 0.0f } },  
+ { { 3.0f, 4.0f,-3.0f}, {0.6f, 0.5f, 0.3f, 1.0f}, { 0.75f, 0.31f } },  
+ { { 2.8f, 0.0f,-3.0f}, {0.4f, 0.3f, 0.2f, 1.0f}, { 0.75f, 1.0f } },  
+  
+ { { 3.0f, 4.0f, 3.0f}, {0.2f, 0.1f, 0.1f, 1.0f}, { 0.25f, 0.69f } },  
+ { { 0.0f, 6.0f, 3.0f}, {0.3f, 0.2f, 0.2f, 1.0f}, { 0.25f, 1.0f } },  
+ { {-3.0f, 4.0f, 3.0f}, {0.2f, 0.1f, 0.1f, 1.0f}, { 0.25f, 0.69f } },  
+ { {-3.0f, 4.0f,-3.0f}, {0.2f, 0.1f, 0.1f, 1.0f}, { 0.25f, 0.69f } },  
+ { { 0.0f, 6.0f,-3.0f}, {0.3f, 0.2f, 0.2f, 1.0f}, { 0.25f, 1.0f } },  
+ { { 3.0f, 4.0f,-3.0f}, {0.2f, 0.1f, 0.1f, 1.0f}, { 0.25f, 0.69f } },  
  
 // 岩

変更したら**ビルドして実行してください。**  
前よりは見られるモデルになっているはずです。

…オモッテタノト チガウヨウナ？  
屋根や扉の色は赤色で、窓は灰色っぽかったはず…。それに、家の側面に扉らしきものが表示されています。、画像の左側面はこんな絵ではなかったはずです。どうやら、まだおかしい部分があるようですね…。

## テクスチャのformatを指定する

色がおかしいのは、glTexImage2D関数でテクスチャを作成するときに、適切なformat引数を設定していないからです。実はTGAファイルの画像データは「青、緑、赤、不透明」の順番で記録されているのです。ところが、今のテクスチャ作成関数は常にGL\_RGBAを設定しています。GL\_RGBAは「赤、緑、青、不透明」の順番を示します。つまり、画像データの赤色の情報が青色として、青色の情報が赤色として使われているわけです。

色の順番を解決するのは簡単です。GL\_BGRAという定数に置き換えるだけです。とはいえ、単純に置き換えるてしまうと、今度は前回作成したテクスチャの赤と青が逆転してしまいます。そこで、CreateImage2D関数にformat引数を追加して、関数を呼び出す側が形式を選択できるようにしましょう。

最初に行うのは、関数宣言にformat引数を追加することです。Texture.hを開き、次のプログラムを追加してください。

#ifndef TEXTURE\_H\_INCLUDED  
 #define TEXTURE\_H\_INCLUDED  
 #include <GL/glew.h>  
  
 namespace Texture {  
  
-GLuint CreateImage2D(GLsizei width, GLsizei height, const GLvoid\* data);  
+GLuint CreateImage2D(GLsizei width, GLsizei height, const GLvoid\* data,  
+ GLenum format);  
 GLuint LoadImage2D(const char\* path);  
  
 } // namespace Texture  
  
 #endif // TEXTURE\_H\_INCLUDED

同様にして、関数定義にもformat引数を追加します。Texture.cppを開き、CreateImage2D関数を次のように変更してください。

/\*\*  
 \* 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 data テクスチャデータへのポインタ.  
+\* @param format 転送元画像のデータ形式.  
 \*  
 \* @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 CreateImage2D(GLsizei width, GLsizei height, const GLvoid\* data,  
+ GLenum format)  
 {  
 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);  
+ 0, GL\_RGBA8, width, height, 0, format, GL\_UNSIGNED\_BYTE, data);  
 const GLenum result = glGetError();  
 if (result != GL\_NO\_ERROR) {  
 std::cerr << "ERROR: テクスチャの作成に失敗(0x" << std::hex << result << ").";

変更点は、format引数を追加していること、それから、追加した引数をglTexImage2Dに設定しているところです。

続いて、LoadImage2D関数のなかでCreateImage2Dを呼び出している部分を、次のように変更してください。

std::vector<uint8\_t> buf(imageSize);  
 ifs.read(buf.data(), imageSize);  
  
 // 読み込んだ画像データからテクスチャを作成する.  
- return CreateImage2D(width, height, buf.data());  
+ return CreateImage2D(width, height, buf.data(), GL\_BGRA);  
}  
  
} // namespace Texture

CreateImage2Dの最後の引数としてGL\_BGRAを設定するようにしただけです。

最後に、main関数のCreateImage2D呼び出しを修正します。  
Main.cppを開き、テクスチャを作成しているプログラムを次のように変更してください。

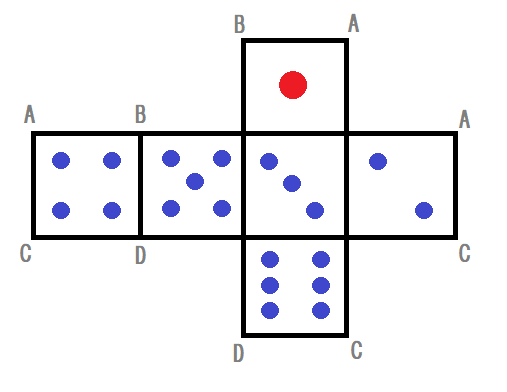
W, W, W, W, W, W, B, W,  
 W, W, W, W, W, W, B, W,  
 B, B, B, B, B, B, B, B,  
 };  
- GLuint texId = Texture::CreateImage2D(tw, th, imageData);  
+ GLuint texId = Texture::CreateImage2D(tw, th, imageData, GL\_RGBA);  
 GLuint texHouse = Texture::LoadImage2D("Res/House.tga");  
  
 // メインループ.  
 while (!window.ShouldClose()) {  
 glEnable(GL\_DEPTH\_TEST);

こちらも、CreateImage2Dの最後の引数としてGL\_RGBAを設定するようにしただけです。

プログラムが書けたら、**ビルドして実行してください。**家のテクスチャが正しい色で表示されていたら成功です。

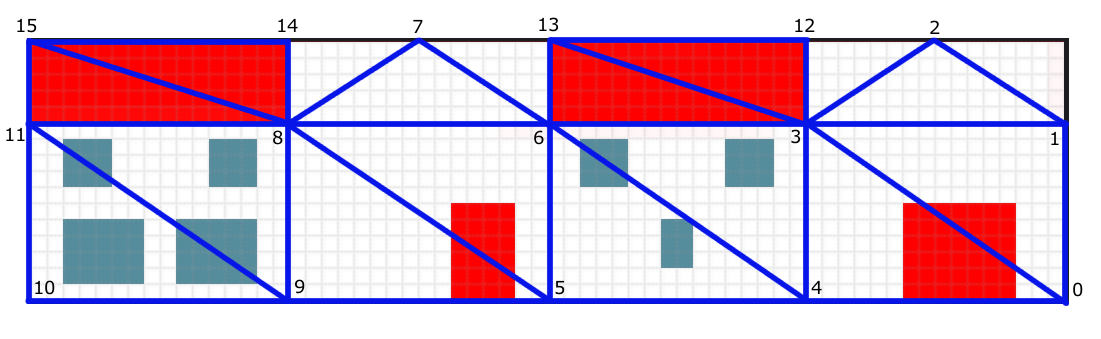
## 家のモデルを修正する(その2)

色は直りましたが、まだ左側面は考えていたのとは違う画像が表示されています。こうなってしまう原因は「モデルの表面を二次元に展開すると、ひとつの頂点が複数のテクスチャ座標に現れることがある」からです。例えば、サイコロモデル用のテクスチャを作るとします。それは以下のような画像になるでしょう。

この図では、サイコロモデル上の同じ頂点に対応するテクスチャ座標にA～Dの記号を当ててあります。AとCは3箇所、BとDは2箇所に現れていますね。  
問題は、ひとつの頂点データには、ひとつのテクスチャ座標を割り当てるのが基本だということです。そのため、なにか工夫をしない限り、A～Dのような頂点を表現することはできません。

この問題を解決する方法はいくつか考えられますが、最も簡単な方法は「テクスチャ座標の数だけ頂点をコピーする」ことです。

この方針に沿って、頂点データとインデックスデータを作り直してみましょう。  
まずは画像を元にして展開図を考えます。これは次の画像のようになるでしょう。



図の各頂点には対応するインデックスを振ってあります。  
それでは、この図を元に頂点データを修正します。家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, 1.0f } },  
 { { 3.0f, 4.0f, 3.0f}, {0.6f, 0.5f, 0.3f, 1.0f}, { 0.0f, 0.31f } },  
 { { 0.0f, 6.0f, 3.0f}, {0.5f, 0.4f, 0.2f, 1.0f}, { 0.125f, 0.0f } },  
 { {-3.0f, 4.0f, 3.0f}, {0.6f, 0.5f, 0.3f, 1.0f}, { 0.25f, 0.31f } },  
 { {-2.8f, 0.0f, 3.0f}, {0.4f, 0.3f, 0.2f, 1.0f}, { 0.25f, 1.0f } },  
  
 { {-2.8f, 0.0f,-3.0f}, {0.4f, 0.3f, 0.2f, 1.0f}, { 0.5f, 1.0f } },  
 { {-3.0f, 4.0f,-3.0f}, {0.6f, 0.5f, 0.3f, 1.0f}, { 0.5f, 0.31f } },  
 { { 0.0f, 6.0f,-3.0f}, {0.5f, 0.4f, 0.2f, 1.0f}, { 0.625f, 0.0f } },  
 { { 3.0f, 4.0f,-3.0f}, {0.6f, 0.5f, 0.3f, 1.0f}, { 0.75f, 0.31f } },  
 { { 2.8f, 0.0f,-3.0f}, {0.4f, 0.3f, 0.2f, 1.0f}, { 0.75f, 1.0f } },  
  
- { { 3.0f, 4.0f, 3.0f}, {0.2f, 0.1f, 0.1f, 1.0f}, { 0.25f, 0.69f } },  
- { { 0.0f, 6.0f, 3.0f}, {0.3f, 0.2f, 0.2f, 1.0f}, { 0.25f, 1.0f } },  
- { {-3.0f, 4.0f, 3.0f}, {0.2f, 0.1f, 0.1f, 1.0f}, { 0.25f, 0.69f } },  
- { {-3.0f, 4.0f,-3.0f}, {0.2f, 0.1f, 0.1f, 1.0f}, { 0.25f, 0.69f } },  
- { { 0.0f, 6.0f,-3.0f}, {0.3f, 0.2f, 0.2f, 1.0f}, { 0.25f, 1.0f } },  
- { { 3.0f, 4.0f,-3.0f}, {0.2f, 0.1f, 0.1f, 1.0f}, { 0.25f, 0.69f } },  
+ { { 2.8f, 0.0f, 3.0f}, {0.4f, 0.3f, 0.2f, 1.0f}, { 1.0f, 1.0f } },  
+ { { 3.0f, 4.0f, 3.0f}, {0.6f, 0.5f, 0.3f, 1.0f}, { 1.0f, 0.31f } },  
+ { { 0.0f, 6.0f, 3.0f}, {1.0f, 1.0f, 1.0f, 1.0f}, { 0.25f, 0.0f } },  
+ { { 0.0f, 6.0f,-3.0f}, {1.0f, 1.0f, 1.0f, 1.0f}, { 0.5f, 0.0f } },  
+ { { 0.0f, 6.0f,-3.0f}, {1.0f, 1.0f, 1.0f, 1.0f}, { 0.75f, 0.0f } },  
+ { { 0.0f, 6.0f, 3.0f}, {1.0f, 1.0f, 1.0f, 1.0f}, { 1.0f, 0.0f } },  
  
 // 岩

極力データを変更しないで済むようにしたので、変更するのは後半の6頂点だけです。ここは屋根の頂点データだったのですが、壁の頂点のコピーと屋根の頂点に置き換えたため、ほぼ行全体が変更されています。

続いて、インデックスデータを次のように変更してください。

const GLushort indices[] = {  
 // 木  
 0, 1, 2, 0, 2, 3, 0, 3, 1, 1, 2, 3,  
 4, 5, 6, 4, 6, 7, 4, 7, 5,  
  
 // 家  
 0, 1, 3, 3, 4, 0, 1, 2, 3,  
 5, 6, 8, 8, 9, 5, 6, 7, 8,  
- 9, 8, 1, 1, 0, 9,  
+ 9, 8, 11, 11, 10, 9,  
 4, 3, 6, 6, 5, 4,  
- 15, 14, 11, 11, 10, 15,  
- 12, 11, 14, 14, 13, 12,  
+ 3, 12, 13, 13, 6, 3,  
+ 8, 14, 15, 15, 11, 8,  
  
 // 四角形  
 0, 1, 2, 2, 3, 0,  
 4, 5, 6, 7, 8, 9,  
 };

変更が済んだら**ビルドして実行してください。**  
今度こそ、どの面にも考えたとおりのテクスチャが表示されることでしょう。

# 対応するテクスチャの形式を増やす

## 圧縮なし16ビット形式

Visual Studioで作成できるTGAファイルには3種類の形式があります。ひとつは既に作成した「圧縮なし32ビット形式」で、Visual Studioのプロパティウィンドウには「32bpp RGBA」と表示されます。残る2つは「圧縮なし16ビット形式」と「圧縮なし8ビット白黒形式」で、それぞれ「16bpp BGRA5551」と「8bpp Gray」と表示されます。この章では残りの2つを読み込めるようにしていきます。

まずは「圧縮なし16ビット形式」からやっていきましょう。  
Visual StudioでHouse.tgaを開き、プロパティウィンドウで形式を「16bpp BGRA 5551」に変更して保存してください。

**[補足]** GIMPやPhotoShopには16ビット形式で保存する機能はありません。しかし、24ビット形式やインデックス・カラー形式が選べ、さらに圧縮の有無を選択して保存することができます。Visual Studioに、これらの機能はありません。

ファイル形式を変更したら、**プログラムを実行してください。**多分、おかしな色の家が表示されると思います。プログラムにはなんの変更も加えていないので、間違った情報をもとにテクスチャが作られてしまうためです。

## テクスチャのtypeを指定する

正しい情報を設定するには、formatに加えてtypeも指定できなければなりません。formatのときと同様に、CreateImage2D関数にtype引数を追加しましょう。Texture.hを開き、CreateImage2D関数の宣言を次のように変更してください。

#ifndef TEXTURE\_H\_INCLUDED  
 #define TEXTURE\_H\_INCLUDED  
 #include <GL/glew.h>  
  
 namespace Texture {  
  
 GLuint CreateImage2D(GLsizei width, GLsizei height, const GLvoid\* data,  
- GLenum format);  
+ GLenum format, GLenum type);  
 GLuint LoadImage2D(const char\* path);  
  
 } // namespace Texture  
  
 #endif // TEXTURE\_H\_INCLUDED

同様にして、関数定義にもtype引数を追加します。Texture.cppを開き、CreateImage2D関数を次のように変更してください。

/\*\*  
 \* 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 data テクスチャデータへのポインタ.  
 \* @param format 転送元画像のデータ形式.  
+\* @param type 転送元画像のデータ格納形式.  
 \*  
 \* @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,  
- GLenum format)  
+ GLenum format, GLenum type)  
 {  
 GLuint id;  
 glGenTextures(1, &id);  
 glBindTexture(GL\_TEXTURE\_2D, id);  
 glTexImage2D(GL\_TEXTURE\_2D,  
- 0, GL\_RGBA8, width, height, 0, format, GL\_UNSIGNED\_BYTE, data);  
+ 0, GL\_RGBA8, width, height, 0, format, type, 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 << ").";

続いて、LoadImage2D関数のなかでCreateImage2Dを呼び出している部分を、次のように変更してください。

std::vector<uint8\_t> buf(imageSize);  
 ifs.read(buf.data(), imageSize);  
  
 // 読み込んだ画像データからテクスチャを作成する.  
- return CreateImage2D(width, height, buf.data(), GL\_BGRA);  
+ GLenum type = GL\_UNSIGNED\_BYTE;  
+ if (tgaHeader[16] == 16) {  
+ type = GL\_UNSIGNED\_SHORT\_1\_5\_5\_5\_REV;  
+ }  
+ return CreateImage2D(width, height, buf.data(), GL\_BGRA, type);  
}  
  
} // namespace Texture

16ビット形式かどうかはTGAヘッダのピクセル・デプスを調べれば分かります。ここには1ピクセルのビット数が記録されているので、記録されている値が16だったら16ビット形式です。  
TGAファイルの16ビット形式は次のようになっています。

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| A | R | | | | | G | | | | | B | | | | |

**Bit**

**Color**

この図の上段は、ビット単位の位置、下段はそのビットに割り当てられている色の種類です。  
このビット配置に対応するOpenGLの定数は「GL\_UNSIGNED\_SHORT\_1\_5\_5\_5\_REV(じーえる・あんさいんど・しょーと・いち・ご・ご・ご・れぶ)」です(REVはreverse(りばーす)の略)。そこで、ピクセル・デプスが16の場合はtypeにGL\_UNSIGNED\_SHORT\_1\_5\_5\_5\_REVを設定し、それ以外の場合はGL\_UNSIGNED\_BYTEを設定するようにしています。

最後に、main関数のCreateImage2D呼び出しを修正します。  
Main.cppを開き、テクスチャを作成しているプログラムを次のように変更してください。

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(tw, th, imageData, GL\_RGBA);  
+ GLuint texId = Texture::CreateImage2D(tw, th, imageData, GL\_RGBA, GL\_UNSIGNED\_BYTE);  
 GLuint texHouse = Texture::LoadImage2D("Res/House.tga");  
  
 // メインループ.  
 while (!window.ShouldClose()) {  
 glEnable(GL\_DEPTH\_TEST);

CreateImage2Dの最後の引数としてGL\_UNSIGNED\_BYTEを追加しただけです。

プログラムが書けたら、**ビルドして実行してください。**家のテクスチャが正しい色で表示されていたら成功です。

## アラインメント(Alignment)

これで16ビット形式も読み込めるようになった、と言いたいところですが、実はまだ完璧ではありません。  
再びVisual StudioでHouse.tgaファイルを開き、横幅を「65」にして保存してください。  
保存したら**プログラムを実行してください。**テクスチャが「ななめ」にくずれていませんか？

こうなってしまう原因は、OpenGLが画像データを読み取るときに「あるルール」を使っているためです。そのルールとは「**データを4バイト単位で読み取る**」というものです。この読み取り単位のことを「アラインメント(Alignment, 整列)」といいます。OpenGLがこのようなルールを使っているのは、一般にコンピュータというものは、

**「一度に読み書きするバイト数が大きいほど速く読み書きができる」**

ように作られているからです。

一般的な画像データは、横一列のピクセルを記録したら次の列のピクセルを記録する、というようになっています。横幅が4で割り切れるバイト数ならなんの問題もありません。例えば32ビット形式の場合、1ピクセルは4バイトなので必ず4で割り切れます。ところが16ビット形式では、横幅が奇数のときに余りができてしまいます。  
余りの部分は無視されるので、1列ごとに1ピクセルずれていってしまうのです。

この問題の対策方法は2つあります。ひとつは、4で割り切れないバイト数の画像を読み込まないことです。もうひとつは、アラインメントを変更することです。  
実際の開発では、前者の方法を取ることが基本となります。読み書きが遅くなることは、ユーザーに悪い印象を与える可能性があるからです。対して、画像の横幅に制限を設けるのは、グラフィックス・デザイナーにちょっと気をつけてもらえば済みます(実際、経験を積んだデザイナーは常に横幅が4バイト単位の画像を作ってくれます)。

とはいえ、今回はデザイナーに頼らずに、横幅が4バイトで割り切れない画像を読み込めるようにしてみましょう。CreateImage2D関数に、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,  
 　　GLenum format, GLenum type)  
　{  
 　GLuint id;  
 　glGenTextures(1, &id);  
 　glBindTexture(GL\_TEXTURE\_2D, id);  
+ glPixelStorei(GL\_UNPACK\_ALIGNMENT, 1);  
 　 glTexImage2D(GL\_TEXTURE\_2D, 0, GL\_RGBA8, width, height, 0, format, type, data);  
+ glPixelStorei(GL\_UNPACK\_ALIGNMENT, 4);  
 　 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 << ").";

GPUメモリに画像データを転送するとき、そのアラインメントの設定はglPixelStorei(じーえる・ぴくせる・すとあ・あい)関数で行います。この関数の最初の引数にGL\_UNPACK\_ALIGNMENT(じーえる・あんぱっく・あらいんめんと)という定数を設定すると、GPUメモリへデータを転送するときの、転送元データのアラインメントを指定できます。アラインメントはふたつめの引数で設定します。注意点として、アラインメントとして設定できるのは1, 2, 4, 8のいずれかだけです。それ以外の数値を設定しようとするとエラーになり無視されます。  
上記のプログラムでは、GPUメモリへ画像データを転送する前にアラインメントを1にし、どんな横幅の画像でも正しく転送されるようにしています。そして、画像データの転送を終えた後は4に戻します。変更したままにしておくと、あとで別の関数がGPUメモリにデータを転送しようとしたときに、意図せずに速度が遅くなってしまうためです。そういう関数を使っていて「なんか転送が遅いですぅ」ってなったときに、実はその原因が「全然関係ないCreateImage2D関数」にあるなんて、普通は気づきませんよね。元に戻すのは、そういうことが起きないようにするためなんです。それから、4という数値については、OpenGLの初期値が4だからです。

プログラムが書けたら、**ビルドして実行してください。**  
もうテクスチャがななめに傾くことはないはずです。

## 圧縮なし8ビット白黒形式

続いて、Visual Studioで作成できる3つめの形式を読み込めるようにしていきます。  
Visual StudioでHouse.tgaを開き、プロパティウィンドウで形式を「8bpp Gray」に変更して保存してください。

さて、圧縮なし8ビット白黒形式では、画像データとして白黒の濃度を示す数値だけが格納されます。各数値は8ビットで表されます。

**[補足]** TGAファイル仕様に白黒の場合のビット数は明示されていないようです。もっとも、最初のTGAファイル仕様が決められたのは1984年のことです。当時のコンピューターの性能では、16ビット以上の白黒データの存在を考慮する意味はなかったでしょう。

白黒画像では1ピクセルにひとつの要素(どの程度白いか、という情報)だけが格納されています。OpenGLにおいて、要素がひとつのデータ形式はGL\_RED(じーえる・れっど)で表されます。また、各要素は8ビット、つまり1バイトですから、データの格納形式はGL\_UNSIGNED\_BYTEになります。  
これを踏まえてプログラムを書いていきましょう。LoadImage2D関数に次のプログラムを追加してください。

std::vector<uint8\_t> buf(imageSize);  
 ifs.read(buf.data(), imageSize);  
  
 // 読み込んだ画像データからテクスチャを作成する.  
 GLenum type = GL\_UNSIGNED\_BYTE;  
+ GLenum format = GL\_BGRA;  
+ if (tgaHeader[2] == 3) {  
+ format = GL\_RED;  
+ }  
 if (tgaHeader[16] == 16) {  
 type = GL\_UNSIGNED\_SHORT\_1\_5\_5\_5\_REV;  
 }  
- return CreateImage2D(width, height, buf.data(), GL\_BGRA, type);  
+ return CreateImage2D(width, height, buf.data(), format, type);  
}  
  
} // namespace Texture

TGAヘッダのイメージ・タイプが「3」ならば白黒画像なので、formatをGL\_REDにしています。それ以外はGL\_BGRAのままです。

これで圧縮なし8ビット白黒形式の画像を読み込めるようになりました。しかし、このまま実行すると家が真っ赤になってしまいます。理由は簡単で、データには赤の要素しかないのに、テクスチャの形式にはGL\_RGBA8を設定しているからです。ならば、internalFormatをGL\_R8(じーえる・あーる・はち)にすればいいのかというと、これも赤くなるのを止められません。OpenGLとGLSLの初期設定が、R要素しかないテクスチャをシェーダーで読み取るときは、R要素以外は0として読み取られる仕様になっているからです。ですから、「R要素だけを持ったテクスチャ」を作るだけではうまくいかないのです。

## スウィズル(Swizzle)

こんなときに便利なのが「スウィズル(Swizzle、かきまぜる)」という機能です。この機能を使うと、シェーダーでテクスチャを読み取る時、データの順序を入れ替えることができます。RGBAの順序で読み取られるところを、RRRRのようにできるということです。

それでは、CreateImage2D関数に次のプログラムを追加してください。

// 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\_REPEAT);  
 glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_T, GL\_REPEAT);  
+　　// 1要素の画像データの場合、(R,R,R,1)として読み取られるように設定する.  
+ if (format == GL\_RED) {  
+ const GLint swizzle[] = { GL\_RED, GL\_RED, GL\_RED, GL\_ONE };  
+ glTexParameteriv(GL\_TEXTURE\_2D, GL\_TEXTURE\_SWIZZLE\_RGBA, swizzle);  
+ }  
  
 glBindTexture(GL\_TEXTURE\_2D, 0);  
  
 return id;  
 }

スウィズルを設定するにはglTexParameteriv(じーえる・てっくす・ぱらめーたー・あい・ぶい)関数を使います。この関数はglTexParameteri関数の最後の引数を、GLint型配列へのポインタを設定するように変えたバージョンです。この関数のふたつめの引数にGL\_TEXTURE\_SWIZZLE\_RGBAという定数を設定し、みっつめの引数に順序を格納した配列へのポインタを設定します。すると、シェーダーでtexture関数を使ってテクスチャを読み出すとき、この機能で設定した順番に従ってデータが読み出される、という仕組みです。  
配列には4つの要素があり、先頭から順番にRとして読み取られるデータ、Rとして読み取られるデータ、Bとして(略)、Aとして(略)、となっています。ここで、Aとして読み取られるデータは「GL\_ONE(じーえる・わん)」になっていますね。これは「常に1が読み取られる」という意味になります。同様に「GL\_ZERO(じーえる・ぜろ)」を使うと「常に0が読み取られる」ように設定できます。白黒画像には不透明度は存在しないので、A要素が常に「不透明」になるように1を設定しているのです。

プログラムが書けたら、**ビルドして実行してください。**頂点データに色が設定してあるので白黒にはなりませんが、濃淡だけが変化しているように見えていれば成功です。

## 圧縮なし24ビット形式

Visual Studioでは作成できませんが、PhotoShopやGIMPなどでは作成されることの多い形式にも対応したほうが便利でしょう。  
LoadImage2D関数に、次のプログラムを追加してください。

if (tgaHeader[2] == 3) {  
 format = GL\_RED;  
 }  
- if (tgaHeader[16] == 16) {  
+ if (tgaHeader[16] == 24) {  
+ format = GL\_BGR;  
+ } else if (tgaHeader[16] == 16) {  
 type = GL\_UNSIGNED\_SHORT\_1\_5\_5\_5\_REV;  
 }

圧縮なし24ビット形式は、32ビット形式からアルファ要素をなくしただけです。ですから、formatをGL\_BGRに変更するだけで対応できます。

**[補足]** 本当は、読み込む形式に合わせてinternalFormatも調整するべきなのですが、当面は問題ないので、今回は対応しないことにしました。

# 画像の上下を反転する

## イメージ・デスクリプタ(Image Descriptor)のひみつ

えー、ここでみなさんに大事なお知らせがあります…。  
実は…  
Visual Studioで作ったTGAファイルは…  
「上下逆さま」になっているのです！

ナ、ナンダッテー！＞ΩΩΩ

残念ながら本当です。TGAファイルには、画像データがどの向きで格納されているかを示すデータがあり、その値によって「上から下」に格納されているのか「下から上」なのかが決まります。そして、「上から下」の場合は左上が原点、「下から上」の場合は左下が原点になります。以前説明したようにOpenGLのテクスチャ座標系は左下が原点です。そのため、TGAファイルが「上から下」で画像を記録している場合、原点の位置が違うために画像が逆さまになるわけです。

「逆さま？家のテクスチャは普通に表示されてたよ？」

そう思うのはもっともですが、ここでさらに残念なお知らせがあります。実は、家モデルのテクスチャ座標も逆さまになっていたのです！　どちらも逆さまなので、問題がないように見えているわけですね。「これはこれであり」と考えることもできますが、しかし、逆さまを正しいとしてしまうと、今度は「下から上」で作られたテクスチャを正しく表示できません。もちろん、全てのテクスチャを「上から下」で作り「逆さまこそ正しい状態だ」とするなら構いませんが、インターネットから画像を持ってきたり、他の人に絵を描いてもらうときに困りますよね。

そこで、「上から下」だった場合は画像の上下を入れ替える処理を追加します。  
LoadImage2D関数に、次のプログラムを追加してください。

std::vector<uint8\_t> buf(imageSize);  
 ifs.read(buf.data(), imageSize);  
  
+ // 画像データが「上から下」で格納されている場合、上下を入れ替える.  
+ if (tgaHeader[17] & 0x20) {  
+ const int lineSize = width \* pixelDepth / 8;  
+ std::vector<uint8\_t> tmp(imageSize);  
+ std::vector<uint8\_t>::iterator source = buf.begin();  
+ std::vector<uint8\_t>::iterator destination = tmp.end();  
+ for (int i = 0; i < height; ++i) {  
+ destination -= lineSize;  
+ std::copy(source, source + lineSize, destination);  
+ source += lineSize;  
+ }  
+ buf.swap(tmp);  
+ }  
+  
 GLenum type = GL\_UNSIGNED\_BYTE;  
 GLenum format = GL\_BGRA;

画像データがどの方向で格納されているかは「イメージ・デスクリプタ」を見れば分かります。  
イメージ・デスクリプタをビット単位で見ると、次のような構造になっています。

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| ピクセルのアルファ要素のビット数 | | | | 0 = 左から右 1 = 右から左 | 0 = 下から上 1 = 上から下 | 未使用(常に0) | |

**bit**

今回注目するのは第5ビットです。この部分に画像の上下方向が記録されているからです。あるビットが1かどうかを調べる最も簡単な方法は、「&」でビット積(AND演算)を計算することです。AND演算は、2つの値の両方のビットが1のときだけ1、それ以外は0になる計算方法です。ですから、第5ビットだけ1になっている数値(0x20)とイメージ・デスクリプタとをAND演算すれば、イメージ・デスクリプタの第5ビットが1のときは0x20になり、そうでなければ0x00になる、という仕組みです。

「上から下」だということが分かったら、画像の上下を反転させます。これは、vector型の変数tmp(てんぷ)に上下反転させたデータを作り、最後にbufとtmpを入れ替えるという方法で実現しています。  
反転させる方法はとても単純です、bufの先頭から1列ずつ、tmpの末尾のほうへとコピーしていくだけです。  
まず、1列のバイト数を計算してlineSize(らいん・さいず)という変数に格納します。  
次に、画像サイズと同じ大きさのvector型の変数tmp(てんぷ)を作成します。この変数に逆さまのデータを作っていきます。その次は、bufの先頭とtmpの終端を指す、「iterator(イテレータ)」という型の変数を作っています。

イテレータという名前は「iterate(いてれーと、反復する)」という単語にerを付与したものです。C++言語では、メモリや配列の中の位置を示すオブジェクトのことを「イテレータ」と呼びます。ですから、ポインタもイテレータの一種です。vector型のイテレータを取得するには、begin(びぎん)メンバ関数かend(えんど)メンバ関数を使います。beginは「先頭の位置」を指すイテレータを返します。endは「終端の位置」を指すイテレータを返します。source(ソース)イテレータの初期化にはbufのbeginメンバ関数を使っているので、bufの先頭を指すことになります。同様にdestination(ですてぃねーしょん)イテレータは、tmpの終端を指した状態になります。

**[補足]** sourceは「源泉、原因、出どころ」という意味です。destinationは「目的地、あて先」という意味です。

イテレータを作ったら、for文を使って画像データを1列ずつコピーしていきます。列の数は画像の高さと等しいので、ループ変数iは0からスタートしてheight未満なら1ずつ増えていく、というようにしています。

では、for文の中を見ていきましょう。最初にdestinationから1列のバイト数を引いています。この時点ではdestinationはコピー先となる列の終端を指しているからです。コピーするためには列の先頭が必要なので、先頭を指すようにlineSizeバイト手前に移動させています。  
destinationの位置を更新したらデータをコピーします。データのコピーにはstd::copy(えすてぃーでぃー・こぴー)関数を使います。この関数の最初の2つの引数は、コピーしたい範囲の先頭と終端の位置で、これがコピー元になります。そして3つめがコピー先の先頭の位置です。一度にコピーするのは1列分なので、コピー元の先頭はsource、終端はsource + lineSizeになります。そしてコピー先はdestinationです。  
1列分のコピーが終わったら、sourceが次の列の先頭を指すようにlineSizeを足します。  
これをheight回繰り返すと、tmpには逆さまになったデータが作られるわけですね。

for文が終了し、逆さまデータが完成したら、それをbufに設定します(逆さまデータはtmpに作っていたのでしたね)。これにはswap(すわっぷ)メンバ関数を使います。swapは、引数で指定したオブジェクトと中身を入れ替える関数です。この関数を実行した後は、bufに逆さまデータ、tmpに元のデータが入った状態になります。

これで画像の上下は正しくなりました。

**[補足]** イメージ・デスクリプタには左右方向を指定するビットもあります。過去のハードウェアではどちら向きで記録するかがバラバラだったためです。しかし、もはや「右から左」以外で記録するハードウェアはまず見かけません。そのような画像を作成するツールも見かけなくなりました。ですから、対応する必要はないと思います(興味があれば挑戦してみてください)。

## 家のテクスチャ座標を逆転させる

画像だけが正しくなっても、家モデルのテクスチャ座標がそのままでは表示がおかしくなっただけでおわってしまいます。そこで課題です。課題01が完了したら、上下どちら向きのTGAファイルであっても正しく表示できるようになるでしょう。

**[課題01]** 上下が正しいテクスチャを使って以前と同様の絵が表示されるように、家モデルのテクスチャ座標を修正してください。

**[課題02]** 岩のテクスチャを作成して、岩がそのテクスチャで表示されるようにしてください。