[OpenGL 3D 2018 第09回]

May the OBJ File be with you.

「OBJファイルが共にあらんことを。」

# ファイル・フォーマットの選択

## OBJファイルについて

そこで、何らかのアプリケーションを利用して頂点データやインデックスデータを作ることを考えます。3次元モデルを作るためのさまざまなアプリがあります。

## モデリングツール

メッシュリストをクラスにしたことで、モデルを追加しやすくなりました。そうはいっても、手作業で頂点データやインデックスデータを作るのは大変です。簡単な多角形を組み合わせた程度のモデルであれば、手作業で作ることも難しくはないでしょう。しかし、数千、数万の頂点データを定義しなければならないとしたら？　そのような大量のデータを手作業で書くのは、不可能とは言いませんがとても現実的とは思えません。そこで、通常はモデル作成用のアプリケーションを使ってモデルを作りします。このようなアプリケーションを「モデリング・ツール」などと呼びます。

ここで問題となるのは、モデリング・ツールによって、作成されるデータの形式が異なることです。しかも、それらの形式はモデリング・ツール自身が使うために定義されているので、ほかのプログラムから使うことができません。そのため、モデリング・ツールで出力したデータをプログラムで使用できる形式に変換する必要があります。

## ファイル・フォーマット

ところで、データの形式にはどのようなものがあるのでしょうか。一般的に、モデリング・ツールがモデル・データを扱う際は、はそれぞれ専用の形式を使います。この形式のことを「ファイル・フォーマット」といいます。

|  |  |
| --- | --- |
| ファイル・フォーマット | アプリケーション |
| MA/MB | Maya |
| 3DS | 3DS Max |
| POV | POV-Ray |
| LWO | LightWave 3D |
| XSI | Softimage |

また、それぞれのモデリング・ツールには得手不得手があります。となれば、得意分野ごとにツールを切替えられると便利ですよね。そんなときのために、ツール間でデータをやりとりできる汎用のファイル・フォーマットが存在します。これらのうちいくつかは、もともとは上記のファイル・フォーマットと同じく特定のアプリケーションのために定義されたものが多いのですが、仕様が公開されていたり、比較的単純な形式であるなどの理由で、多くのモデリング・ツールが対応しています。そのため、データを汎用のファイル・フォーマットで保存して他のツールで開くことができるわけです。代表的な汎用ファイル・フォーマットには次のようなものがあります。

**OBJ(おぶじぇ):**  
 Advanced Visualizerというツールのために開発されたフォーマットです。非常に単純なファイル・フォーマットで、頂点カラーが定義できないなどの制約があります。しかし、定義された時期が古いため長く使われていること、仕様が公開されていること、単純であることの3点から、ほとんどのモデリング・ツールが読み書きに対応しています。最も普及した汎用ファイル・フォーマットといえるでしょう。

**STL(えす・てぃー・える):**  
 3D Systems社の3D CAD用フォーマットです。これも古くからある比較的単純なファイル・フォーマットで、OBJ形式と同様に多くのモデリング・ツールが対応しています。

**PLY(ぴー・える・わい):**  
スタンフォード大学が3Dスキャナー(現実の物体からモデル・データを作成する装置)用に開発したファイル・フォーマットです。OBJ形式と比較するとやや複雑なためか、直接対応しているモデリングツールはあまり多くありません。とはいえ、これも古くからあるファイル・フォーマットなので、検索すれば大抵のツール用のプラグイン(=拡張機能)を見つけることができるでしょう。

**COLLADA(こらだ):**  
SCE(現SIE)がPS3/PSP用に開発したフォーマットです。比較的最近に開発されたこともあり、複雑なデータも扱えます。そのぶんフォーマットも複雑ですが、仕様が公開されていることから多くのモデリング・ツールが対応しています。また、公式GitHubに読み書き用ソースコードが公開されています。

**X3D(えっくす・すりー・でぃー)**:  
ネットワーク上で3Dデータをやりとりするために開発されたファイル・フォーマットです。こちらも比較的最近に開発されたもので、複雑なデータを扱えます。仕様が公開されているものの、まだ直接対応しているモデリングツールは多くありません。いくつかのツールではプラグインが開発されているようです。

**FBX(えふ・びー・えっくす):**  
Filmboxというツールのために開発されたファイル・フォーマットです。3Dシーン全体を記録できるように作られているため、非常に複雑です。詳細な仕様は公開されていませんが、FBXを読み書きするためのライブラリが提供されていて、それを使えば比較的簡単にデータを読み書きできます。そのため、多くのモデリングツールが対応しています。

## フォーマットの選択

ひととおりファイル・フォーマットを挙げましたので、今回使うファイル・フォーマットとしてどれを選ぶか考えてみましょう。まず、モデリング・ツール専用のファイル・フォーマットは除外しましょう。それらは他のアプリケーションから使われることなど全く考えておらず、仕様も公開されていないため、まず仕様を詳しく調べることから始めなくてはならないからです。そうなると、あとは汎用ファイルフォーマットからどれを選ぶかです。まずは簡単なファイル・フォーマットと複雑なファイル・フォーマットを比較してみましょう。

簡単なファイル・フォーマットの利点は、当然ながらプログラムを書くのが簡単だということです。欠点は、頂点カラーのように対応していない機能があることです。  
複雑なフォーマットの利点は、考えられるほとんど全ての機能に対応していることです。また、モデル・データ以外にもアニメーションやモデルの配置情報などを含むことができます。そのかわり、それらを適切に判別して読み込まなければなりません。ライブラリやソースコードが提供されているなら、この工程は多少簡単になるかもしれません。

さてさて、これらの利点と欠点を考慮したうえで、選ばれたのは(ここでドラムロール)…

「**OBJ**」ファイル・フォーマット！君に決めた！

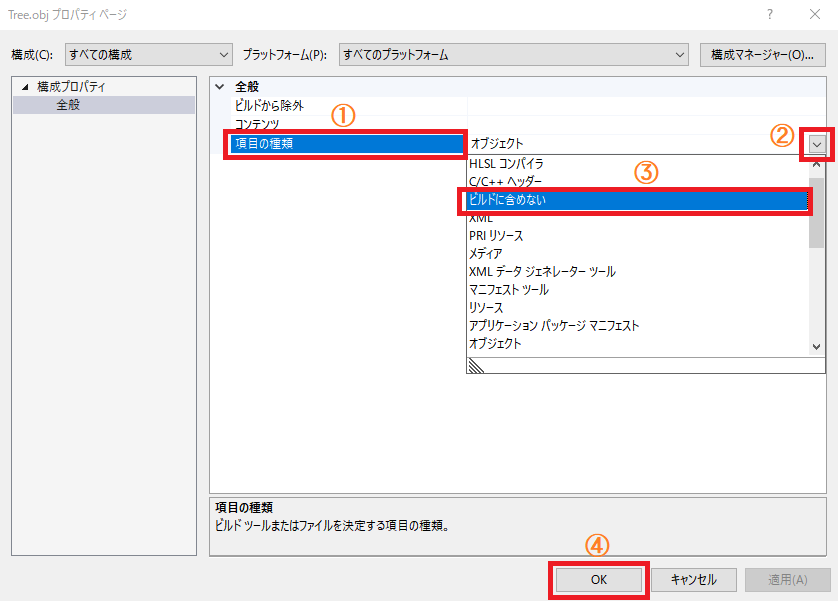
OBJを選んた理由は、単純さと対応しているモデリング・ツールの多さです。単純ゆえにあまり凝ったことはできませんが、問題ありません。なぜなら、今回作成するのは「基本的なモデル・データを読み込む機能」だけだからです。

# OBJファイルを作成する

## OBJファイルを追加する

OBJファイルはテキスト・ファイルです。文法も難しくありませんので、テキスト・エディタで簡単に編集することができます。ファイル・フォーマットの仕様では、点・線・三角形だけでなく、多角形や曲線、曲面を定義する能力を持っていますが、今回は三角形のみを読み込めるようにしていきます。

とにかくOBJファイルを作ってみましょう。ソリューションエクスプローラーの「リソースファイル」フィルターの上で右クリックして「追加→新しい項目」を選択してください。次に、右側のリストから「ユーティリティ」を、中央のリストから「テキストファイル」を選択します。そして、場所テキストを「プロジェクトフォルダ\Res」に、名前テキストを「Tree.obj」に変更してから「追加」ボタンを押すと、OBJファイルが作成されます。ソリューションエクスプローラーに「Tree.obj」が追加されると思います。  
次に、追加されたTree.objを右クリックして「右クリックメニュー」を表示し、下の方にある「プロパティ」を選んでください。すると、以下のように「プロパティ」ウィンドウが表示されます。



最初は「項目の種類」が「オブジェクト」になっていると思います。これを「ビルドに含めない」に変更します。まず「項目の種類」をクリックしてください(①)。すると、右側に下向き矢印のボタンが表示されるので、これをクリックします(②)。種類のリストが表示されますので、「ビルドに含めない」をクリックして種類を変更してください(③)。最後にOKボタンを押すと変更が反映されます(④)。

「項目の種類」を変更しなければならないのには理由があります。実はVisual Studioは「.obj」という拡張子を「ビルドの過程で作られる作業用ファイル」に割り当てています。そのため、Tree.objもプログラムとして扱われてしまうのです。しかし、もちろんTree.objはプログラムではありませんから、コンピューターは混乱してビルドに失敗してしまいます。そこで、プロパティを設定して「このファイルはプログラムではないよ。だからビルドに含めてはいけないよ。」と教えてあげなければならないのですね。

## OBJファイルの文法

モデル・データをOBJファイルに書き込むためには、4つの文法を覚えなくてはなりません。すなわち、頂点座標を定義する「v」と、テクスチャ座標を定義する「vt」、法線を定義する「vn」、そして面を定義する「f」です。

|  |  |
| --- | --- |
| **v** *x y z* | 頂点の座標を定義します。vはvertexの頭文字。x,y,zは頂点のX,Y,Z座標を表す浮動小数点数です。各要素はスペースで区切られます。 |
| **vt** *u v* | テクスチャ座標を定義します。vtはvertex textureの頭文字。u, vは頂点のテクスチャ座標を表す浮動小数点数です。各要素はスペースで区切られます。 |
| **vn** *i j k* | 法線を定義します。vnはvertex normalの頭文字。I,j,kは頂点の法線のX,Y,Z座標を表す浮動小数点数です。各要素はスペースで区切られます。 |
| **f** *v1/vt1/vn1 v2/vt2/vn2 v3/vt3/vn3* | 多角形による面を定義します。fはfaceの頭文字。v1,vt1,vn1は面を構成する頂点座標、テクスチャ座標、法線の番号です。各番号はスラッシュ(/)で分離されます。番号とスラッシュの間にスペースを入れてはいけません。番号は1番から始まります。多角形の頂点の数だけこの組を記述します。各頂点はスペースで区切られます。 |

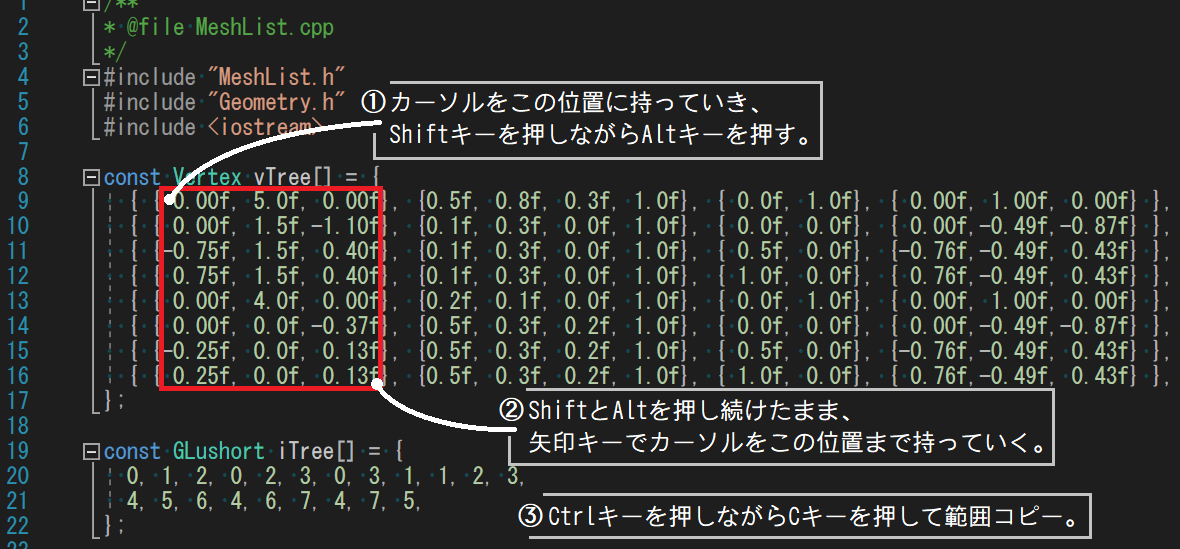
これらに加えて、コメント記号として「#(しゃーぷ)」を使うことができ、「#」ヵら改行までがコメントとして扱われます。なお、OpenGLの頂点データと異なり、OBJファイル・フォーマットでは頂点を構成する座標、テクスチャ座標、法線を個別に定義しなければなりません。

## OBJファイルに頂点座標を定義する

それでは、木のモデル・データをOBJファイルに定義しましょう。まずはコメントで、このファイルがどんなモデルを格納しているのかを記しておきます。Tree.objに次のテキストを追加してください。

+#  
+# Tree.obj  
+#

最初は頂点座標を定義します。MeshList.cppを開き、次の画像のようにShiftキーとAltキーを使って頂点座標の部分だけをコピーしてください。

  
[頂点座標をコピーする]

そしてコピーしたデータをTree.objに貼り付けてください。

#  
 # Tree.obj  
 #  
+  
+ 0.00f, 5.0f, 0.00f  
+ 0.00f, 1.5f,-1.10f  
+-0.75f, 1.5f, 0.40f  
+ 0.75f, 1.5f, 0.40f  
+ 0.00f, 4.0f, 0.00f  
+ 0.00f, 0.0f,-0.37f  
+-0.25f, 0.0f, 0.13f  
+ 0.25f, 0.0f, 0.13f

これは頂点の座標なので、次のように先頭に「v」を付与します。コピーのときと同様にShiftキーとAltキーを組み合わせて先頭の縦一列を選択し、v、スペースの順でキーを押すだけです。このとき、先頭のマイナス記号を消してしまわないように気をつけてください。

#  
 # Tree.obj  
 #  
  
- 0.00f, 5.0f, 0.00f  
- 0.00f, 1.5f,-1.10f  
--0.75f, 1.5f, 0.40f  
- 0.75f, 1.5f, 0.40f  
- 0.00f, 4.0f, 0.00f  
- 0.00f, 0.0f,-0.37f  
--0.25f, 0.0f, 0.13f  
- 0.25f, 0.0f, 0.13f  
+v 0.00f, 5.0f, 0.00f  
+v 0.00f, 1.5f,-1.10f  
+v -0.75f, 1.5f, 0.40f  
+v 0.75f, 1.5f, 0.40f  
+v 0.00f, 4.0f, 0.00f  
+v 0.00f, 0.0f,-0.37f  
+v -0.25f, 0.0f, 0.13f  
+v 0.25f, 0.0f, 0.13f

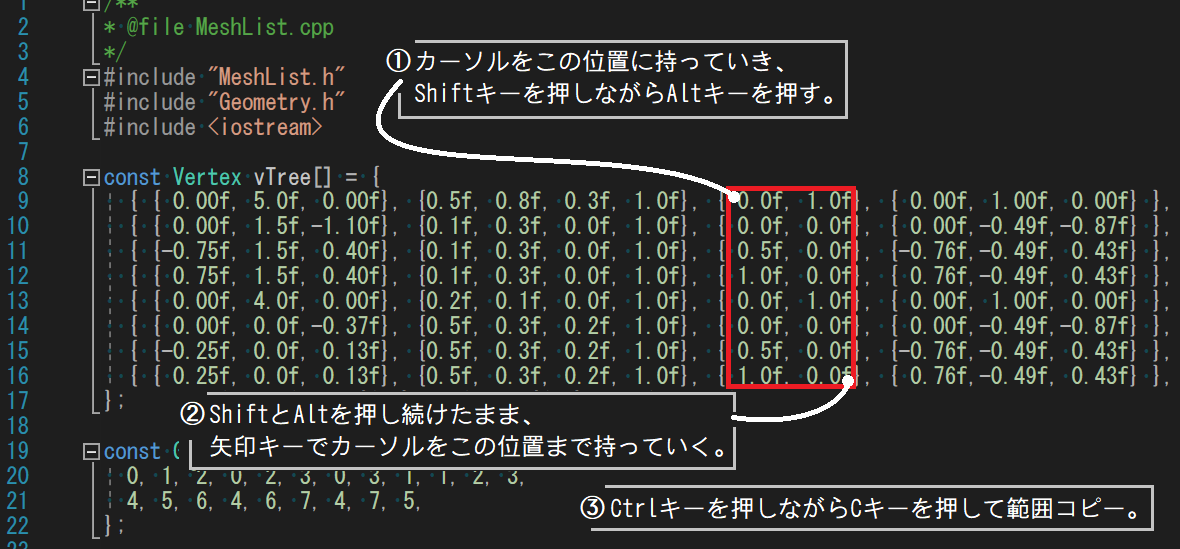
OBJファイル・フォーマットの浮動小数点数は、末尾に「f」を付けることができません。また、要素の区切りはカンマではなくスペースですので、これらを次のように修正してください。

#  
 # Tree.obj  
 #  
  
-v 0.00f, 5.0f, 0.00f  
-v 0.00f, 1.5f,-1.10f  
-v -0.75f, 1.5f, 0.40f  
-v 0.75f, 1.5f, 0.40f  
-v 0.00f, 4.0f, 0.00f  
-v 0.00f, 0.0f,-0.37f  
-v -0.25f, 0.0f, 0.13f  
-v 0.25f, 0.0f, 0.13f  
+v 0.00 5.0 0.00  
+v 0.00 1.5 -1.10  
+v -0.75 1.5 0.40  
+v 0.75 1.5 0.40  
+v 0.00 4.0 0.00  
+v 0.00 0.0 -0.37  
+v -0.25 0.0 0.13  
+v 0.25 0.0 0.13

スペースの数はいくつでも構いませんので、見やすくなるように数を調節するといいでしょう。  
これで頂点の座標を定義することができました。

## OBJファイルにテクスチャ座標を定義する

続いてテクスチャ座標を定義します。MeshList.cppを開き、頂点座標のときと同様にShiftキーとAltキーを使って、テクスチャ座標の部分だけをコピーしてください。

  
[テクスチャ座標をコピーする]

Tree.objを開き、コピーしたテクスチャ座標を頂点座標の下に貼り付けてください。

v 0.00 5.0 0.00  
 v 0.00 1.5 -1.10  
 v -0.75 1.5 0.40  
 v 0.75 1.5 0.40  
 v 0.00 4.0 0.00  
 v 0.00 0.0 -0.37  
 v -0.25 0.0 0.13  
 v 0.25 0.0 0.13  
+  
+ 0.0f, 1.0f  
+ 0.0f, 0.0f  
+ 0.5f, 0.0f  
+ 1.0f, 0.0f  
+ 0.0f, 1.0f  
+ 0.0f, 0.0f  
+ 0.5f, 0.0f  
+ 1.0f, 0.0f

テクスチャ座標には先頭に「vt」が必要です。また、頂点座標と同様に「f」やカンマは削除します。ということで、テクスチャ座標の部分を次のように変更してください。Shift+Altを活用しましょう。

v 0.00 5.0 0.00  
 v 0.00 1.5 -1.10  
 v -0.75 1.5 0.40  
 v 0.75 1.5 0.40  
 v 0.00 4.0 0.00  
 v 0.00 0.0 -0.37  
 v -0.25 0.0 0.13  
 v 0.25 0.0 0.13  
  
- 0.0f, 1.0f  
- 0.0f, 0.0f  
- 0.5f, 0.0f  
- 1.0f, 0.0f  
- 0.0f, 1.0f  
- 0.0f, 0.0f  
- 0.5f, 0.0f  
- 1.0f, 0.0f  
+vt 0.0 1.0  
+vt 0.0 0.0  
+vt 0.5 0.0  
+vt 1.0 0.0  
+vt 0.0 1.0  
+vt 0.0 0.0  
+vt 0.5 0.0  
+vt 1.0 0.0

これで、テクスチャ座標を定義することができました。

## OBJファイルに法線を定義する

続いて法線を定義しましょう。頂点座標やテクスチャ座標と同様に、MeshList.cppから法線データをコピーし、次のようにTree.objに貼り付けてください。

vt 0.0 1.0  
 vt 0.0 0.0  
 vt 0.5 0.0  
 vt 1.0 0.0  
 vt 0.0 1.0  
 vt 0.0 0.0  
 vt 0.5 0.0  
 vt 1.0 0.0  
  
+ 0.00f, 1.00f, 0.00f  
+ 0.00f,-0.49f,-0.87f  
+-0.76f,-0.49f, 0.43f  
+ 0.76f,-0.49f, 0.43f  
+ 0.00f, 1.00f, 0.00f  
+ 0.00f,-0.49f,-0.87f  
+-0.76f,-0.49f, 0.43f  
+ 0.76f,-0.49f, 0.43f

法線の定義の先頭は「vn」です。頂点座標やテクスチャ座標と同様に「f」やカンマは削除しなければなりません。法線を次のように変更してください。

vt 0.0 1.0  
 vt 0.0 0.0  
 vt 0.5 0.0  
 vt 1.0 0.0  
 vt 0.0 1.0  
 vt 0.0 0.0  
 vt 0.5 0.0  
 vt 1.0 0.0  
  
- 0.00f, 1.00f, 0.00f  
- 0.00f,-0.49f,-0.87f  
--0.76f,-0.49f, 0.43f  
- 0.76f,-0.49f, 0.43f  
- 0.00f, 1.00f, 0.00f  
- 0.00f,-0.49f,-0.87f  
--0.76f,-0.49f, 0.43f  
- 0.76f,-0.49f, 0.43f  
+vn 0.00 1.00 0.00  
+vn 0.00 -0.49 -0.87  
+vn -0.76 -0.49 0.43  
+vn 0.76 -0.49 0.43  
+vn 0.00 1.00 0.00  
+vn 0.00 -0.49 -0.87  
+vn -0.76 -0.49 0.43  
+vn 0.76 -0.49 0.43

これで法線の定義は完了です。

## OBJファイルに三角形を定義する

モデルを定義するためには、頂点のデータだけでは不完全です。モデルがどのような面で構成されているのかを定義しなければなりません。これは、OpenGLのインデックスデータにあたるものです。OBJファイル・フォーマットも頂点データのインデックスによって多角形を定義するのですが、違い2点あります。

ひとつは、頂点座標、テクスチャ座標、法線に個別のインデックスを指定するところです。例えば、頂点座標と法線は同じでテクスチャ座標だけが異なる2つの頂点があるとします。OpenGLでは完全な頂点データが必要なので、頂点座標と法線は重複して定義しなければなりません。対してOBJファイルでは、テクスチャ座標だけを2つ定義すれば済みます。

もうひとつは、インデックスが1番から始まる点です。C/C++言語やOpenGLでは0番から始まるので、OBJファイルでもうっかり0番を使ってしまいがちです。注意してください。

これらを踏まえて、面のデータを定義しましょう。まずはMeshList.cppにあるiTree配列からインデックスデータをコピーしてください。

vn 0.00 1.00 0.00  
 vn 0.00 -0.49 -0.87  
 vn -0.76 -0.49 0.43  
 vn 0.76 -0.49 0.43  
 vn 0.00 1.00 0.00  
 vn 0.00 -0.49 -0.87  
 vn -0.76 -0.49 0.43  
 vn 0.76 -0.49 0.43  
  
+ 0, 1, 2, 0, 2, 3, 0, 3, 1, 1, 2, 3,  
+ 4, 5, 6, 4, 6, 7, 4, 7, 5,

OBJファイル・フォーマットでは、1行にひとつの三角形しか定義できません。そこで、次のようにインデックスを3つずつの組に分けてください。

vn 0.00 -0.49 -0.87  
 vn -0.76 -0.49 0.43  
 vn 0.76 -0.49 0.43  
  
- 0, 1, 2, 0, 2, 3, 0, 3, 1, 1, 2, 3,  
- 4, 5, 6, 4, 6, 7, 4, 7, 5,  
+ 0, 1, 2,  
+ 0, 2, 3,  
+ 0, 3, 1,  
+ 1, 2, 3,  
+ 4, 5, 6,  
+ 4, 6, 7,  
+ 4, 7, 5,

面の定義の先頭には「f」が必要です。また、他の定義と同様に区切りはカンマではなくスペースですから、Tree.objのインデックスの定義を次のように変更してください。

vn 0.00 -0.49 -0.87  
 vn -0.76 -0.49 0.43  
 vn 0.76 -0.49 0.43  
  
- 0, 1, 2,  
- 0, 2, 3,  
- 0, 3, 1,  
- 1, 2, 3,  
- 4, 5, 6,  
- 4, 6, 7,  
- 4, 7, 5,  
+f 0 1 2  
+f 0 2 3  
+f 0 3 1  
+f 1 2 3  
+f 4 5 6  
+f 4 6 7  
+f 4 7 5

次に、頂点のインデックス番号をずらします。インデックスはOpenGLでは0から、OBJファイルでは1から数えますので、OpenGLからコピーしてきた全ての数字を1増やさなければなりません。面の定義を次のように変更してください。

vn 0.00 -0.49 -0.87  
 vn -0.76 -0.49 0.43  
 vn 0.76 -0.49 0.43  
  
-f 0 1 2  
-f 0 2 3  
-f 0 3 1  
-f 1 2 3  
-f 4 5 6  
-f 4 6 7  
-f 4 7 5  
+f 1 2 3  
+f 1 3 4  
+f 1 4 2  
+f 2 3 4  
+f 5 6 7  
+f 5 7 8  
+f 5 8 6

最後に、頂点座標、テクスチャ座標、法線の3つに個別のインデックスを割り当てます。といっても、今回の場合はOpenGLのデータを元にしているため、全て同じインデックスになります。ですから「1」と書いてあるなら「1/1/1」に、「7」と書いてあれば「7/7/7」のようにしていきます。それでは、面の定義を次のように変更してください。なお、この場面でもShift+Altによる矩形選択が便利です。縦1列をコピーして貼り付けると簡単に変更できます。

vn 0.00 -0.49 -0.87  
 vn -0.76 -0.49 0.43  
 vn 0.76 -0.49 0.43  
  
-f 1 2 3  
-f 1 3 4  
-f 1 4 2  
-f 2 3 4  
-f 5 6 7  
-f 5 7 8  
-f 5 8 6  
+f 1/1/1 2/2/2 3/3/3  
+f 1/1/1 3/3/3 4/4/4  
+f 1/1/1 4/4/4 2/2/2  
+f 2/2/2 3/3/3 4/4/4  
+f 5/5/5 6/6/6 7/7/7  
+f 5/5/5 7/7/7 8/8/8  
+f 5/5/5 8/8/8 6/6/6

これで木のモデルをOBJファイルにすることができました。  
OBJファイル・フォーマットの構造を、なんとなく理解してもらえているといいのですが。

# OBJファイルを読み込む

## OBJファイルを読み込む関数を宣言する

この章では、前の章で作成したOBJファイルを読み込んで、メッシュデータにする機能を作成します。OBJファイルを読み込む関数の宣言を追加することから始めましょう。  
MeshList.hを開き、次のプログラムを追加してください。

bool Allocate();  
 void Free();  
 void Add(const Vertex\* vBegin, const Vertex\* vEnd,  
 const GLushort\* iBegin, const GLushort\* iEnd);  
+ bool AddFromObjFile(const char\* path);  
  
 void BindVertexArray();  
 const Mesh& Get(size\_t index) const;

関数名はAddFromObjFile(あど・ふろむ・おぶじぇ・ふぁいる)としました。引数は「OBJファイル名」で、戻り値として「メッシュの追加に成功したかどうか」を返します。

## ヘッダファイルのインクルード

次にファイルを読み込むために必要なヘッダファイルをインクルードします。MeshList.cppを開き、次のプログラムを追加してください。

/\*\*  
 \* @file MeshList.cpp  
 \*/  
 #include "MeshList.h"  
+#include <fstream>  
+#include <string>  
+#include <stdio.h>  
+#include <math.h>  
 #include <iostream>

まず、ファイル操作のためにfstream(えふ・すとりーむ)ヘッダが必要ですね。それから、行単位の読み込みにstd::stringを使いたいので、string(すとりんぐ)ヘッダをインクルードしています。stdio.h(すたんだーど・あい・おー・どっと・えいち)とmath.h(ます・どっと・えいち)には、読み込んだファイルを解析し、頂点データとインデックスデータに変換するために使う関数が含まれています。

## AddFromFile関数を定義する

ここからAddFromFile関数の定義を書いていきます。この関数は少し長くなるので、雛形から始めることにします。MeshList.cppのAddメンバ関数定義の下に、次のプログラムを追加してください。

tmpVertices.insert(tmpVertices.end(), vBegin, vEnd);  
 tmpIndices.insert(tmpIndices.end(), iBegin, iEnd);  
 }  
  
+/\*\*  
+\* OBJファイルからメッシュを読み込む.  
+\*  
+\* @param path 読み込むOBJファイル名.  
+\*  
+\* @retval true 読み込み成功.  
+\* @retval false 読み込み失敗.  
+\*/  
+bool MeshList::AddFromObjFile(const char\* path)  
+{  
+ return true;  
+}  
+  
 /\*\*  
 \* 描画に使用するVAOを設定する.  
 \*/  
 void MeshList::BindVertexArray()

## ファイルを開く

まずはファイルの読み込み処理を追加しましょう。AddFromObjFile関数に、次のプログラムを追加してください。

bool MeshList::AddFromObjFile(const char\* path)  
 {  
+ // ファイルを開く.  
+ std::ifstream ifs;  
+ ifs.open(path);  
+ if (!ifs.is\_open()) {  
+ std::cerr << "ERROR: " << path << "を開けません\n";  
+ return false;  
+ }  
+  
 return true;  
 }

std::ifstreamはstd::basic\_ifstream<char>の別名(using宣言したもの)です。テキストファイルの読み書きは頻繁に使われるので、標準C++ライブラリに最初から定義されているのです。そしてopen関数でファイルを開きます。なお、今回はテキストファイルとして開きますので、引数にはファイル名だけ設定しています。もしファイルを開けなかったら(is\_open関数がfalseを返したら)、エラーメッセージを出力し、戻り値をfalseにして終了します。

## データ格納用の変数を準備する

次に、読み取ったデータを格納するための変数を準備します。ファイルを開くプログラムの下に、次のプログラムを追加してください。

bool MeshList::AddFromObjFile(const char\* path)  
 {  
 // ファイルを開く.  
 std::ifstream ifs;  
 ifs.open(path);  
 if (!ifs.is\_open()) {  
 std::cerr << "ERROR: " << path << "を開けません\n";  
 return false;  
 }  
  
+ // データ読み取り用の変数を準備.  
+ struct Face {  
+ int v;  
+ int vt;  
+ int vn;  
+ };  
+ std::vector<Face> faceList;  
+ std::vector<Vector3> positionList;  
+ std::vector<Vector2> texCoordList;  
+ std::vector<Vector3> normalList;  
+ faceList.reserve(1000);  
+ positionList.reserve(1000);  
+ texCoordList.reserve(1000);  
+ normalList.reserve(1000);  
+  
 return true;  
 }

Face(ふぇいす)構造体は、OBJファイル・フォーマットの「f」要素を表します。そして、「f」要素を格納するためのvector型変数がfaceList(ふぇいす・りすと)です。positionList(ぽじしょん・りすと)、texCoordList(てっくす・こーど・りすと)、normalList(のーまる・りすと)は、それぞれv, vt, vnの要素を格納するためのvector型変数です。これらの変数は、reserveメンバ関数でとりあえず1000個分のメモリを確保するようにしました。

**[補足]** 確保視するメモリの量はアプリケーションに依存します。市販ゲームで使われるようなモデルを読み込むなら、1万から10万は必要でしょう。今回はそこまで大きなモデルは扱わないので1000程度で十分でしょう。

## ファイルを行単位で読み込む

ここから、いよいよOBJファイルを読み込んでいきます。データ格納用変数を準備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。

indexSetList.reserve(1000);  
 positionList.reserve(1000);  
 texCoordList.reserve(1000);  
 normalList.reserve(1000);  
  
+ // ファイルからモデルのデータを読み込む.  
+ while (!ifs.eof()) {  
+ std::string line;  
+ getline(ifs, line);  
+ }  
+  
 return true;  
 }

eof(いー・おー・えふ)メンバ関数は、読み込み位置がファイルの終端(end of file)に到達したかどうかを返します。このメンバ関数とwhile文を組み合わせた

while(!ifs.eof()) {  
 読み込み処理  
}

という書き方は、テキストファイルを読み込む処理の基本形です。

さて、C++でファイルを読み込むにはいくつかの方法がありますが、今回はgetline(げっと・らいん)関数を使うことにしました。getlineは、1つめの引数で指定されたファイルから1行読み取って、2つめの引数に格納する関数です。

## 読み込んだテキストを調べる

次に、読み込んだ行の文法を調べ、読み取ったデータを文法に対応する変数に格納する処理を追加しましょう。getline関数の下に、次のプログラムを追加してください。

// ファイルからモデルのデータを読み込む.  
 while (!ifs.eof()) {  
 std::string line;  
 getline(ifs, line);  
+  
+ // 読み取り処理用の変数を定義する.  
+ Vector3 v;  
+ Vector2 vt;  
+ Vector3 vn;  
+ Face f[3];  
+  
+ // データを文法に対応する変数に格納する.  
+ if (sscanf\_s(line.data(), "v %f %f %f", &v.x, &v.y, &v.z) == 3) {  
+ positionList.push\_back(v);  
+ } else if (sscanf\_s(line.data(), "vt %f %f", &vt.x, &vt.y) == 2) {  
+ texCoordList.push\_back(vt);  
+ } else if (sscanf\_s(line.data(), "vn %f %f %f",  
+ &vn.x, &vn.y, &vn.z) == 3) {  
+ const float length = sqrt(vn.x \* vn.x + vn.y \* vn.y + vn.z \* vn.z);  
+ vn.x /= length;  
+ vn.y /= length;  
+ vn.z /= length;  
+ normalList.push\_back(vn);  
+ } else if (sscanf\_s(line.data(), "f %d/%d/%d %d/%d/%d %d/%d/%d",  
+ &f[0].v, &f[0].vt, &f[0].vn,  
+ &f[1].v, &f[1].vt, &f[1].vn,  
+ &f[2].v, &f[2].vt, &v[2].vn) == 9) {  
+ faceList.push\_back(f[0]);  
+ faceList.push\_back(f[1]);  
+ faceList.push\_back(f[2]);  
+ }  
 }  
  
 return true;  
 }

まずいくつかの変数を定義しています。v,vt,vn,fは、名前の通りv,vt,vn,f文法のデータを取得するための変数です。

変数を定義したら、文法を調べてデータを取得していきます。このときに使うのはsscanf\_s(えす・すきゃん・えふ・えす)関数です。sscanf\_s関数は、最初の引数で指定した文字列に、2つめの引数で指定された形式の文字列が含まれているかを調べ、含まれていれば、特定のルールに従って3つめ以降の変数にデータを格納します。sscanf\_sはstdio.hで宣言されています。

**[補足]** sscanf\_sという名前は「**s**tring **scan f**ormat **s**ecure」を省略したものです。無理やり日本語に翻訳するなら「文字列を書式付きで走査する関数の安全なバージョン」となるでしょうか。

「特定のルール」というのは、printf関数でおなじみの「%f」や「%d」のことです。printf関数では%fや%dを「変数の値を文字として表示する」ために使いました。sscanf\_s関数ではその逆に、行の中の%fや%dに相当する部分について「文字を値に変換して変数に格納する」ために使います。  
上記のプログラムの最初のsscanf\_s関数では「v %f %f %f」という形式の文字列が引数になっています。まず文字「v」があり、空白をはさんで浮動小数点数が3つ続く、という形式です。これはOBJファイル・フォーマットの頂点座標の文法そのものです。そして、3つの浮動小数点数が3つめ以降の変数に格納されます。

例として、Tree.objの最初の頂点座標を見てみましょう。これは「v 0.00 5.0 0.00」というテキストでした。この行がline変数に読み込まれ、sscanf\_s関数の最初の引数として渡されると、変数v.xには0.0、v.yには5.0、v.zには0.0が格納されることになります。

行が文法に一致したかどうかはsscanf\_s関数の戻り値で確認できます。sscanf\_s関数の戻り値は、正常に変換・代入された変数の数です。最初のsscanf\_sの場合だと、3つの「%f」を使っていますから、文法が一致して正常に変換・代入が行われた場合は「3」が返ってくるはずです。それ以外の数値が返ってきた場合は違う文法の行だということです。

文法が一致し、変数に値が格納されたら、その値を対応するデータ格納用変数に追加します。  
頂点座標(v)とテクスチャ座標(vt)、それと面(f)はそのままpush\_back関数に設定するだけでいいのですが、法線(vn)は単位ベクトルに変換しなければなりません。なぜかというと、OBJファイル・フォーマットの法線は単位ベクトルであることを要求されていないからです。しかしOpenGLでは単位ベクトルになっていないと都合が悪いため、格納された数値をベクトルの長さで割って単位ベクトルにしてからpush\_back関数に設定します。ベクトルの長さは各要素の2乗を足し合わせた値の平方根です。平方根の計算にはsqrt(すくえあ・るーと)関数を使います。sqrtはmath.hに定義されています。

## データ変換用の変数を準備する

すべての行を走査し終えると、eof関数がtrueを返すようになり、while文を抜けます。これでOBJファイルの読み込みは完了です。続いて、読み込んだデータを頂点データとインデックスデータに変換していきます。  
まずは変換用の変数を準備しましょう。while文の閉じ波括弧の下に、次のプログラムを追加してください。

faceList.push\_back(f[0]);  
 faceList.push\_back(f[1]);  
 faceList.push\_back(f[2]);  
 }  
 }  
  
+ // 頂点データとインデックスデータ用の変数を準備.  
+ std::vector<Vertex> vertices;  
+ std::vector<GLushort> indices;  
+ vertices.reserve(faceList.size());  
+ indices.reserve(faceList.size());  
+  
 return true;  
 }

変換した頂点データをためておくためのvertices変数と、変換したインデックスデータをためておくためのindices変数を定義し、それぞれfaceListと同じ数だけ要素を格納できるようにメモリを確保しておきます。なぜfaceListと同じ数にしたのかは、次に追加するプログラムで判明します。

## 読み込んだ値を変換する

続いて、データ変換処理を追加します。変換用変数を準備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。

// 頂点データとインデックスデータを格納する変数を準備.  
 std::vector<Vertex> vertices;  
 std::vector<GLushort> indices;  
 vertices.reserve(faceList.size());  
 indices.reserve(faceList.size());  
  
+ // モデルのデータを頂点データとインデックスデータに変換する.  
+ for (size\_t i = 0; i < faceList.size(); ++i) {  
+ // インデックスデータを追加する.  
+ indices.push\_back(i);  
+  
+ // 頂点データを追加する.  
+ Vertex vertex;  
+ const int v = faceList[i].v – 1;  
+ const int vt = faceList[i].vt – 1;  
+ const int vn = faceList[i].vn – 1;  
+ vertex.position = positionList[v];  
+ vertex.color = { 1,1,1,1 };  
+ vertex.texCoord = texCoordList[vt];  
+ vertex.normal = normalList[vn];  
+ vertices.push\_back(vertex);  
+ }  
+  
 return true;  
 }

今回は、面の頂点ごとに頂点データとインデックスデータを作ります。この方法は、頂点とインデックスが1対1で対応しているため、インデックスデータを作る意味がありません。そのかわり、比較的簡単にプログラムを書くことができます。

インデックスデータは0から順番に追加していくだけです。OBJファイル・フォーマットの表裏の定義は、OpenGLと同じく頂点の順序が反時計回りのときに表になるようになっています。ですから、特に変換の必要はありません。  
インデックスデータに比べると、頂点データはすこし複雑です。まずfaceListから頂点座標のインデックス、テクスチャ座標のインデックス、法線のインデックスの3つを取り出し、v,vt,vnの各変数に代入しておきます。このとき、数値から1を引いています。理由はOBJファイル・フォーマットはインデックスが1番から始まるからです。そのため、C/C++言語の配列の添字として使うためには1を引く必要があるのです。

インデックスを取得したら、それらを使って配列からデータを取り出し、対応するメンバ変数に格納します。OBJファイル・フォーマットには頂点カラーがないので、color変数には{1,1,1,1}を代入しています。

**[補足]** この頂点カラーの代入で使っている記法は、C++11から導入された「リスト初期化」と呼ばれるものです。この記法のおかげで、配列や構造体の初期化と同じ書き方で代入ができるようになりました。

全てのメンバ変数に値を設定したら、push\_back関数で頂点データを追加します。  
あとはこれを、すべての面の頂点に対して行うだけです。

## 頂点データとインデックスデータをメッシュとして追加する

最後に、変換した頂点データとインデックスデータをメッシュとして追加します。  
データの変換を行うfor文の閉じ波括弧の下に、次のプログラムを追加してください。

vertex.position = positionList[v];  
 vertex.color = { 1,1,1,1 };  
 vertex.texCoord = texCoordList[vt];  
 vertex.normal = normalList[vn];  
 vertices.push\_back(vertex);  
 }  
  
+ Add(vertices.data(), vertices.data() + vertices.size(),  
+ indices.data(), indices.data() + indices.size());  
+  
 return true;  
 }

メッシュを追加する機能は既に作成済みなので、やるべきことはAdd関数を呼ぶだけです。  
これでOBJファイルを読み込んでデータを変換し、メッシュとして追加することができるようになりました。

## AddFromObjFile関数を使う

それでは、作成したOBJファイル読み込み関数を使ってみましょう。  
MeshList::Allocate関数にあるメッシュを追加しているプログラムを、次のように変更してください。

// メッシュを追加.  
- Add(std::begin(vTree), std::end(vTree), std::begin(iTree), std::end(iTree));  
+ AddFromFile("Res/Tree.obj");  
 Add(std::begin(vHouse), std::end(vHouse), std::begin(iHouse), std::end(iHouse));  
 Add(std::begin(vRock), std::end(vRock), std::begin(iRock), std::end(iRock));  
 Add(std::begin(vGround), std::end(vGround), std::begin(iGround), std::end(iGround));  
  
 // VAOを作成する.  
 GLuint vbo = CreateVBO(tmpVertices.size() \* sizeof(Vertex), tmpVertices.data());  
 GLuint ibo = CreateIBO(tmpIndices.size() \* sizeof(GLushort), tmpIndices.data());  
 vao = CreateVAO(vbo, ibo);

プログラムが書けたら、**ビルドして実行してください。**  
色だけが白く変化した木のモデルが表示されていれば成功です。

**[課題01]** 家のモデルを「House.obj」という名前のOBJファイルにしてください。

**[課題02]** 家のモデルの追加処理について、Add関数ではなくAddFromObjFile関数を使うように置き換えてください。

# 頂点データの重複をなくす

## 作成した頂点データの数はいくつ？

木の頂点は8個だけだったはずです。しかし、OBJファイルを読み込むと21個の頂点データが作られます。  
「え、本当？」  
「いやいや、8個しかデータ用意してないんだから8個でしょ。」  
まあ、疑うのは無理もありません。では、実際にいくつの頂点データが作られているのかを出力してみましょう。  
AddFromObjFile関数の末尾付近に次のプログラムを追加してください。

vertex.position = positionList[v];  
 vertex.color = { 1,1,1,1 };  
 vertex.texCoord = texCoordList[vt];  
 vertex.normal = normalList[vn];  
 vertices.push\_back(vertex);  
 }  
  
+ std::cout << "INFO: " << path <<  
+ " [頂点数=" << vertices.size() << " インデックス数=" << indices.size() << "]\n";  
  
 Add(vertices.data(), vertices.data() + vertices.size(),  
 indices.data(), indices.data() + indices.size());  
  
 return true;  
 }

プログラムが書けたら、**ビルドして実行してください。**  
コンソールウィンドウに、読み込んだOBJファイルの名前、頂点数、インデックス数が出力されていたら成功です。表示された数値を見ると、頂点数とインデックス数が等しいこと、どちらも本来の頂点データの数より大きいことが確認できると思います。OBJファイルには8個の頂点しか書き込んでいないはずなのに、一体どういうことなんでしょう。

## 同じ頂点かどうかを調べるには

原因は、OBJファイルには三角形が7つ定義されていて、AddFromObjFile関数ではその全てについて新しい頂点データを作っているからです。それでもモデルは表示できるので、当面は問題になることはないでしょう。しかし、これではインデックスデータを作る意味がありません。そこで、既に同じ頂点データが作られている場合は、新しく頂点データを作らずに既存の頂点データを使うように、プログラムを修正しましょう。

しかし、2つの頂点データが「等しい」とはどういう状態なのでしょう？　答えは、「頂点座標が等しく、テクスチャ座標が等しく、法線が等しい」状態です。ここで「頂点座標が等しい」というのは、もちろん頂点のX,Y,Z座標が全て等しいことを指しているのですが、より正確には「頂点座標のインデックスが等しい」状態だということもできるでしょう。インデックスが等しければ、同じ頂点座標を参照することになるからです。ということは、「頂点座標のインデックス、テクスチャ座標のインデックス、法線のインデックスの3つが全て等しい」ならば、同じ頂点データだと言ってよさそうです。

## f文法の値から対応する頂点データを見つけるには

ところで、vertices変数にはf文法の値を頂点データに変換したものが格納されます。そのため、vertices変数だけを見ても、元になったf文法の値は分かりません。これでは同じ頂点かどうかを調べられませんよね。そこで、vertices変数と対になるvector型変数を定義して、対応するf文法の値を格納することにしましょう。  
MeshList.cppを開き、AddFromObjFile関数に次のプログラムを追加してください。

// 頂点データとインデックスデータ用の変数を準備.  
 std::vector<Vertex> vertices;  
 std::vector<GLushort> indices;  
+std::vector<Face> faceToVertexList;  
 vertices.reserve(faceList.size());  
 indices.reserve(faceList.size());  
+faceToVertexList.reserve(faceList.size());  
  
 // モデルのデータを頂点データとインデックスデータに変換する.  
 for (size\_t i = 0; i < faceList.size(); ++i) {

F文法の値を格納する変数はfaceToVertexList(ふぇいす・とぅ・ばーてっくす・りすと=面から頂点へのリスト)という名前にしてみました。  
それではfaceToVertexListに値を格納しましょう。AddFromObjFile関数に次のプログラムを追加してください。

// モデルのデータを頂点データとインデックスデータに変換する.  
 for (size\_t i = 0; i < faceList.size(); ++i) {  
 // インデックスデータを追加する.  
 indices.push\_back(i);  
  
+ // 面データを追加する.  
+ faceToVertexList.push\_back(faceList[i]);  
+  
 // 頂点データを追加する.  
 Vertex vertex;  
 const int v = faceList[i].v – 1;  
 const int vt = faceList[i].vt – 1;  
 const int vn = faceList[i].vn – 1;  
 vertex.position = positionList[v];  
 vertex.color = { 1,1,1,1 };  
 vertex.texCoord = texCoordList[vt];  
 vertex.normal = normalList[vn];  
 vertices.push\_back(vertex);  
 }

これで、vertices変数のN番目の頂点データがどのようなf文法から変換されたのかを知りたければ、faceToVertexList変数のN番目の値を見れば分かるようになりました。

## 頂点データの重複を見つける

続いて、頂点データの重複をなくします。for文の中で、変換元はfaceList[i]の値です。もしfaceToVertexListのN番目の値がfaceList[i]と等しければ、新しい頂点データを作らずとも、vertices変数のN番目の頂点データを使うことができます。

モデルのデータを頂点データとインデックスデータに変換するプログラムを、次のように変更してください。

// モデルのデータを頂点データとインデックスデータに変換する.  
 for (size\_t i = 0; i < faceList.size(); ++i) {  
+ // 重複する頂点データがあるか調べる.  
+ size\_t n = 0;  
+ for (; n < faceToVertexList.size(); ++n) {  
+ if (faceToVertexList[n].v == faceList[i].v &&  
+ faceToVertexList[n].vt == faceList[i].vt &&  
+ faceToVertexList[n].vn == faceList[i].vn) {  
+ // 重複する頂点データを見つけた.  
+ break;  
+ }  
+ }  
+  
 // インデックスデータを追加する.  
 indices.push\_back(i);  
  
　 // 面データを追加する.  
 　faceToVertexList.push\_back(faceList[i]);

faceToVertexListの中に重複する頂点データが存在する場合、if文の条件が成立してbreak文が実行され、for文を抜けます。その結果、nにはfaceToVertexList.size()より小さい値が格納された状態になります。また、このときのnの値は重複する頂点データのインデックスになっています。一方、一度もbreak文が実行されなければ、for文が終わった時のnの値はfaceToVertexList.size()以上になります。  
つまり、nの値を見れば、重複する頂点データがあったかどうか、あったなら、それは何番目の頂点データなのかということが分かるのですね。

## 重複している場合とそうでない場合を分ける

それでは、nの値によって、新しい頂点を作るかどうかの場合分けを行いましょう。  
重複する頂点データがあるか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。

// モデルのデータを頂点データとインデックスデータに変換する.  
 for (size\_t i = 0; i < faceList.size(); ++i) {  
 // 重複する頂点データがあるか調べる.  
 size\_t n = 0;  
 for (; n < faceToVertexList.size(); ++n) {  
 if (faceToVertexList[n].v == faceList[i].v &&  
 faceToVertexList[n].vt == faceList[i].vt &&  
 faceToVertexList[n].vn == faceList[i].vn) {  
 // 重複する頂点データを見つけた.  
 break;  
 }  
 }  
  
+ if ( n < faceToVertexList.size()) {  
+ // 重複する頂点データが見つかったら、見つかった頂点データのインデックスを追加する.  
+ indices.push\_back((GLushort)n);  
+ } else {  
- // インデックスデータを追加する.  
+ // 重複する頂点データが見つからなければ、新しい頂点データを作成し、そのインデックスを追加する.  
- indices.push\_back(i);  
+ indices.push\_back((GLushort)vertices.size());  
  
　 // 面データを追加する.  
 　 faceToVertexList.push\_back(faceList[i]);  
  
 // 頂点データを追加する.  
 Vertex vertex;  
 const int v = faceList[i].v – 1;  
 const int vt = faceList[i].vt – 1;  
 const int vn = faceList[i].vn – 1;  
 vertex.position = positionList[v];  
 vertex.color = { 1,1,1,1 };  
 vertex.texCoord = texCoordList[vt];  
 vertex.normal = normalList[vn];  
 vertices.push\_back(vertex);  
+ }  
 }  
  
 std::cout << "INFO: " << path <<  
 " [頂点数=" << vertices.size() << " インデックス数=" << indices.size() << "]\n";

重複している頂点データが見つかった場合、nはその頂点データのインデックスになっています。ですから、やるべきことはnをindices変数の末尾に追加するだけです。  
見つからなかった場合は、これまでどおり新しい頂点データを作ります。ただし、もはや変数iを頂点データのインデックスとして使うことはできません。なぜなら、重複した頂点データを作らなくしたので、作成した頂点データの数と変数iの値が一致するとは限らなくなったからです。そこで、変数iの代わりに「作成済みの頂点データの数」をインデックスとして追加するように変更しています。

これで、OBJファイルを読み込んだときに、最小限の頂点データだけが作成されるようになりました。  
**ビルドして実行してください。**同じようにモデルが表示されているにもかかわらず、コンソールウィンドウに表示される頂点数が、インデックス数より小さくなっていれば成功です。

## 間違ったデータを見つける

OBJファイルはただのテキストファイルなので、時には間違った数値を書いてしまうこともあるでしょう。そんなとき、現在のプログラムの挙動は、おかしなモデルを表示するか、前触れもなく停止してしまうかのいずれかです。  
これはちょっと不親切なので、間違ったデータを見つけるプログラムを追加しましょう。OBJファイルを読み込むプログラムと、変換用の変数を準備するプログラムの間に、次のプログラムを追加してください。

faceList.push\_back(f[0]);  
 faceList.push\_back(f[1]);  
 faceList.push\_back(f[2]);  
 }  
 }  
  
+ // データ不足の場合は作成中止.  
+ if (positionList.empty()) {  
+ std::cerr << "WARNING: " << path << "には頂点座標の定義がありません.\n";  
+ Add(nullptr, nullptr, nullptr, nullptr);  
+ return false;  
+ }  
+ if (texCoordList.empty()) {  
+ std::cerr << "WARNING: " << path << "にはテクスチャ座標の定義がありません.\n";  
+ Add(nullptr, nullptr, nullptr, nullptr);  
+ return false;  
+ }  
+ if (normalList.empty()) {  
+ std::cerr << "WARNING: " << path << "には法線の定義がありません.\n";  
+ Add(nullptr, nullptr, nullptr, nullptr);  
+ return false;  
+ }  
+  
 // 頂点データとインデックスデータ用の変数を準備.  
 std::vector<Vertex> vertices;  
 std::vector<GLushort> indices;  
 std::vector<Face> faceToVertexList;  
 vertices.reserve(faceList.size());  
 indices.reserve(faceList.size());  
 faceToVertexList.reserve(faceList.size());

頂点座標、テクスチャ座標、法線のいずれかのデータが全く読み込まれなかった、つまりOBJファイルに適切なデータが設定されていなかった場合は、そのOBJファイルはモデルデータとして不完全です。そのような場合はメッセージを表示して変換を中止します。このとき、空のメッシュデータを追加することで、後に続くメッシュデータのインデックスがずれてしまわないようにしています。

データが定義されている場合でも油断はできません。v,vt,vn文法に誤った値が入っていても、表示されるモデルが少しおかしくなるだけです。しかしf文法のインデックスが定義されているデータの範囲を超えていると、プログラムが止まってしまう可能性があります。そこで、f文法の値を調べるプログラムを追加しましょう。頂点データを追加するプログラムを、次のように変更してください。

// 頂点データを追加する.  
 Vertex vertex;  
-const int v = faceList[i].v - 1;  
+int v = faceList[i].v - 1;  
+if (v < 0 || v >= (int)positionList.size()) {  
+ std::cerr << "WARNING: 不正なvインデックス(" << v << ")\n";  
+ v = 0;  
+}  
-const int vt = faceList[i].vt - 1;  
+int vt = faceList[i].vt - 1;  
+if (vt < 0 || vt >= (int)texCoordList.size()) {  
+ std::cerr << "WARNING: 不正なvtインデックス(" << vt << ")\n";  
+ vt = 0;  
+}  
-const int vn = faceList[i].vn - 1;  
+int vn = faceList[i].vn - 1;  
+if (vn < 0 || vn >= (int)normalList.size()) {  
+ std::cerr << "WARNING: 不正なvnインデックス(" << vn << ")\n";  
+ vn = 0;  
+}  
 vertex.position = positionList[v];  
 vertex.color = { 1,1,1,1 };  
 vertex.texCoord = texCoordList[vt];  
 vertex.normal = normalList[vn];  
 vertices.push\_back(vertex);  
 faceToVertexList.push\_back(faceList[i]);

v,vt,vnの値が0未満だったり、対応する変数に格納されている要素数より大きい場合、それは誤ったデータです。そのようなデータを見つけたら、メッセージを出力して注意を促します。そして誤った値の代わりに0を使うようにします。正しい値は不明なので、とりあえず0を設定して様子を見ます。

プログラムが書けたら、**ビルドして実行してください。**  
もっとも、今回追加したプログラムは、正常なデータには何の影響も及ぼしませんから、見た目の違いは全くないのですが。