[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！

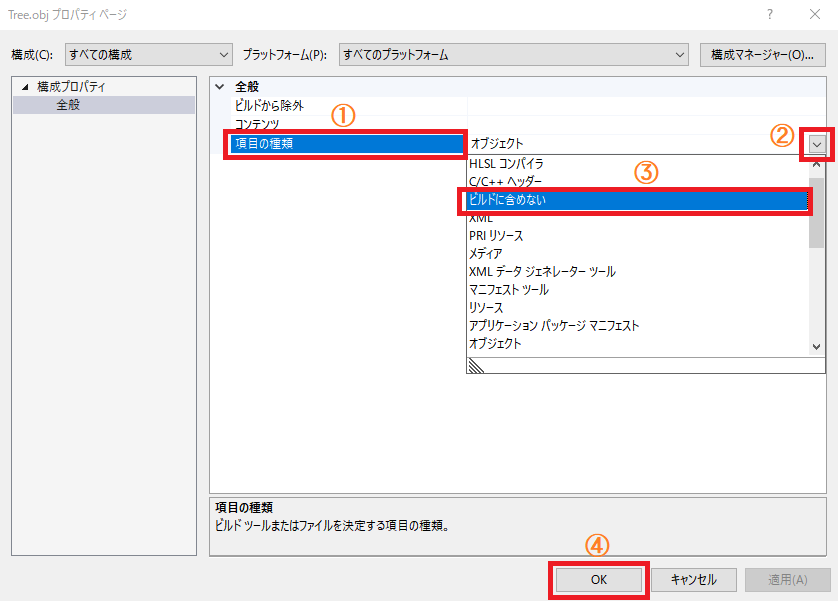
OBJを選んた理由は、単純さと対応しているモデリング・ツールの多さです。単純ゆえにあまり凝ったことはできませんが、問題ありません。なぜなら、今回作成するのは「基本的なモデル・データを読み込む機能」だけだからです。

# 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。 |

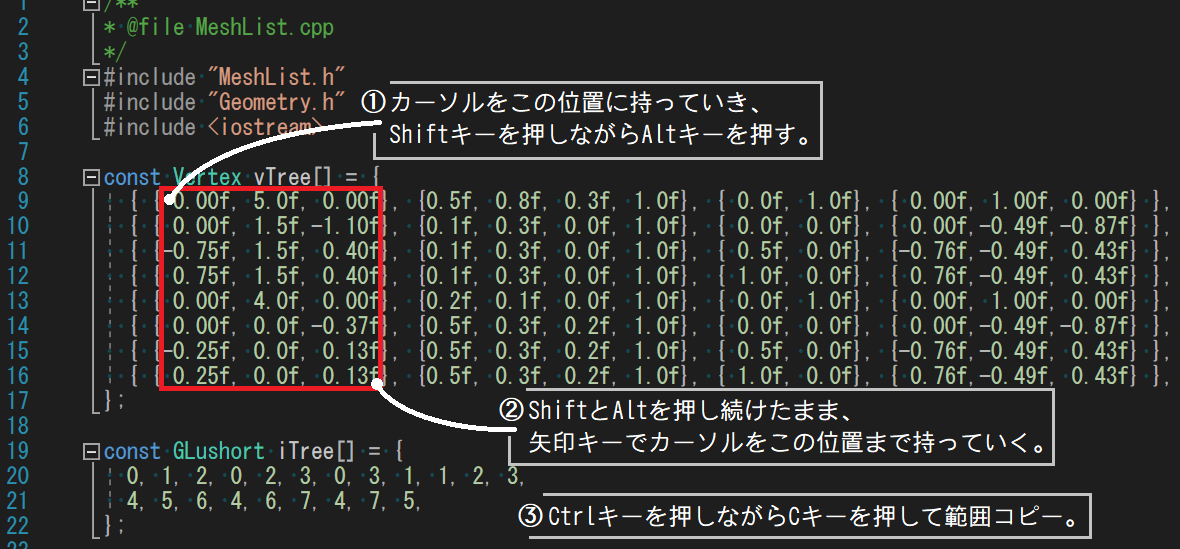
これらに加えて、コメント記号として「#(しゃーぷ)」を使うことができ、「#」ヵら改行までがコメントとして扱われます。なお、OpenGLの頂点データと異なり、OBJファイル・フォーマットでは頂点を構成する座標、テクスチャ座標、法線を個別に定義しなければなりません。

## 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

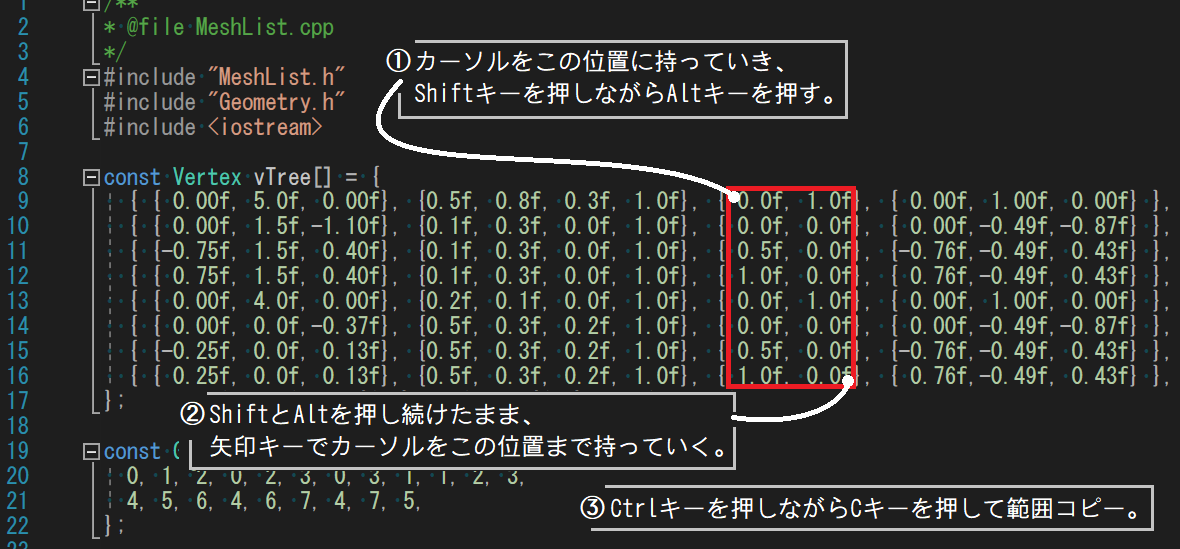
OBJファイル・フォーマットの浮動小数点数は、末尾に「f」を付けることができません。また、要素の区切りはカンマではなくスペースですので、これらを次のように修正してください。

#  
 # 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, &f[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";  
+ positionList.push\_back({0, 0, 0});  
+ }  
+ 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";  
+ texCoordList.push\_back({0, 0});  
+ }  
+ if (normalList.empty()) {  
+ std::cerr << "WARNING: " << path << "には法線の定義がありません.\n";  
+ normalList.push\_back({0, 1, 0});  
+ }  
+  
 // 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、ダミーのデータをひとつ追加します。

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;  
+Color color = { 1, 1, 1, 1 };  
-const int v = faceList[i].v - 1;  
+int v = faceList[i].v - 1;  
+if (v < 0 || v >= (int)positionList.size()) {  
+ color = { 0.5f, 0, 0, 1 };  
+ v = 0;  
+}  
-const int vt = faceList[i].vt - 1;  
+int vt = faceList[i].vt - 1;  
+if (vt < 0 || vt >= (int)texCoordList.size()) {  
+ color = { 0.5f, 0, 0, 1 };  
+ vt = 0;  
+}  
-const int vn = faceList[i].vn - 1;  
+int vn = faceList[i].vn - 1;  
+if (vn < 0 || vn >= (int)normalList.size()) {  
+ color = { 0.5f, 0, 0, 1 };  
+ vn = 0;  
+}  
 vertex.position = positionList[v];  
-vertex.color = { 1,1,1,1 };  
+vertex.color = color;  
 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、誤った値の代わりに0を使うようにします。それぞれのリストは少なくとも1つの要素を持っているので、0は常に有効な添字になります。さらに、エラーがあった部分は頂点カラーを「暗めの赤」にして、実際に表示されたときにエラーがあったことがわかるようにしています。

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。

**[課題 03]** 家のOBJファイルのvt文を全て削除して保存し、実際にエラーが見つかることを確認してください。エラーが確認できたら元の状態に戻してください。削除する前にhouse.objのコピーを作っておくと簡単に元に戻せるでしょう。

# Blenderを使ってみよう

## Blenderとは

OBJファイルを作成するには、3Dモデル作成専用のアプリケーション(以降はモデリング・ツールと呼びます)を使うのが簡単です。しかし、ゲーム用のモデルデータを作成できるモデリング・ツールとなると、あまり選択肢はありません。例えばWindows 10には「Paint 3D」という簡単な3Dシーンを作成するアプリが用意されています。しかし、Paint 3DはOBJファイルを扱えませんし、テクスチャをファイルに書き出す機能もありません。また、Visual Studioにも3Dモデルを作成できる機能があります。こちらはOBJファイルを扱えるのですが、テクスチャを操作する機能がまったくありません。  
他には、3DCやVectaryといった、インストール不要でブラウザ上でモデリングが出来るツールがあります。しかし、どれも単体では力不足で、特にテクスチャを扱う機能は全くないか、あっても簡易的なものにとどまります。

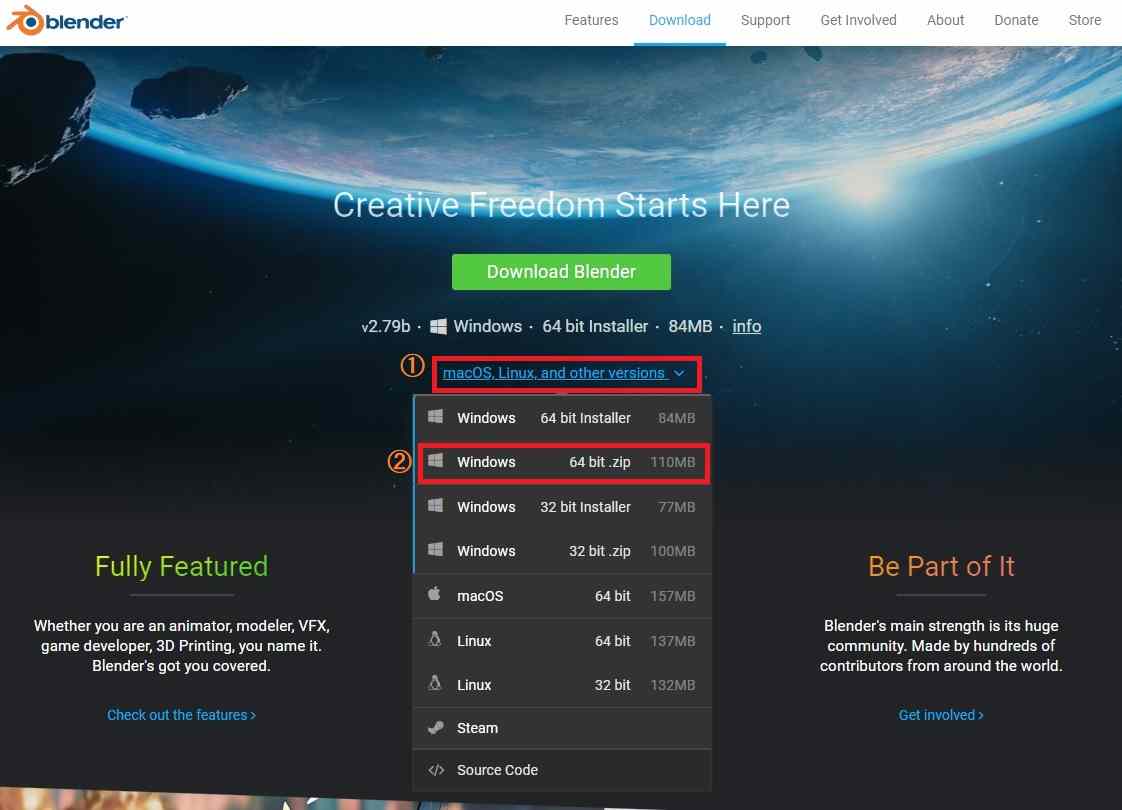
いくつかのツールを試した結果、ちゃんとインストールするタイプのモデリング・ツールを使うしかない、という結論に達しました。とはいえ、有名なツールのほとんどは非常に高価です。いくつかは学生向けに期間限定のフリーバージョンを提供してはいるものの、グラフィックデザインを目指すわけでもないので、そこまでの機能は必要ありません。  
そこで、「Blender(ぶれんだー)」というフリーのモデリング・ツールを使うことにします。

## Blenderのダウンロード

インターネット・ブラウザを開き、アドレスバーに次のURLを入力してください。

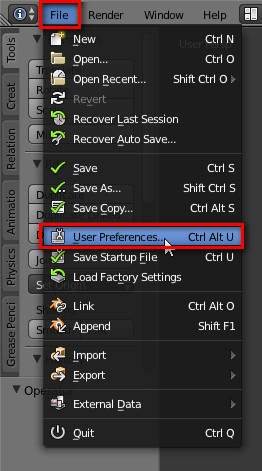
**https://www.blender.org/download/**

すると、次の画面が表示されます。

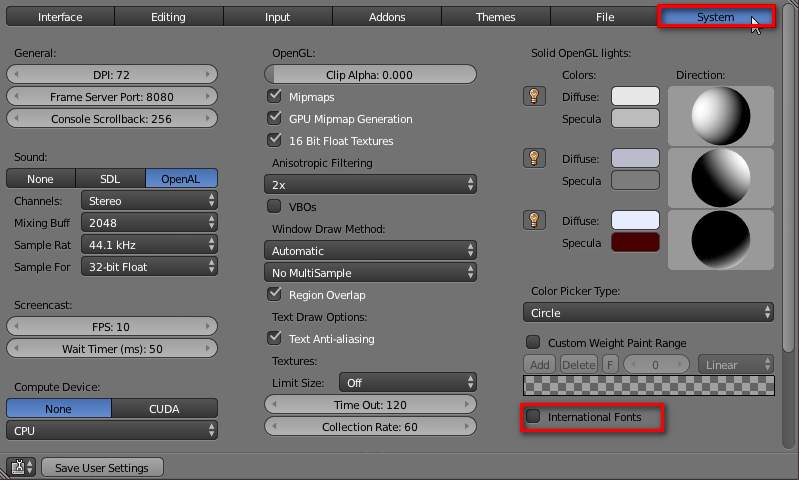


「macOS, Linux, and other versions」と書かれている部分をクリックすると(①)、ダウンロードできるファイルのリストが表示されます。このリストから「Windows 64bit zip」と書かれている部分をクリックしてください(②)。すると、ファイルのダウンロードが始まります。  
ダウンロードが終わったら、ダウンロードしたzipファイルをダブルクリックし、「blender-2.79b-windows64」と書かれたフォルダをコピーし、Dドライブ(あるいは他の容量に余裕のあるドライブ)に貼り付けてください。  
貼り付けが完了したら、貼り付けたblender-2.79b-windows64フォルダを開きます。このフォルダにある「blender」あるいは「blender.exe」というファイルがBlenderアプリケーションです。このファイルをダブルクリックすると、Blenderが起動します。毎回blender.exeを探すのが面倒な場合は、デスクトップに「ショートカット」を作成するといいでしょう。やり方は、Altキーを押しながらblender.exeをデスクトップにドラッグ&ドロップするだけです。

## Blenderを日本語化する

起動直後のBlenderは英語モードになっています。英語のままだと意味が分からないメニューも多いと思いますので日本語モードにしましょう。  
まず、左上の「File」メニューをクリックしてください。するとファイルメニューが開きますので、中程にある「User Preference(ゆーざー・ぷりふぁれんす)」をクリックしてください。

すると、ユーザー設定ウィンドウが開きますので、右上の「System」をクリックしてください。



クリックするとウィンドウの内容が「システム設定」に切り替わります。そうしたら、ウィンドウの右下にある「International Fonts」という文字の横にあるチェックボックスをクリックしてチェックを入れてください。もし画面に見えていない場合は、右端の「スクロール・バー」をドラッグして内容をスクロールしてください。

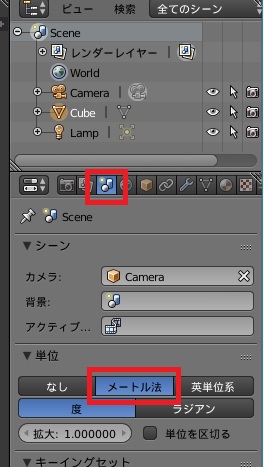
チェックを入れると、文字の下にいくつかの項目が表示されます。そのうちの「English(English)」という文字をクリックすると、利用可能な言語のリストが表示されます。



リストの中から「Japanese(日本語)」を選択してください。使用する言語を変更したら、「language」の下にある「Interface(いんたーふぇいす)」と「Toolchips(つーる・ちっぷ)」ボタンをクリックしてください。このボタンを押すことで、ボタンに対応するメニューが日本語化されます。  
ただし、「New Data(新規データ)」ボタンだけは押さないでください。このボタンを押すと、新規作成したモデルの名前が日本語になります。しかし、メニュー等とは異なり、Blenderのモデル操作プログラムは日本語への対応が不十分なようです。そのため、モデルの名前に日本語が混じっていると、おかしな動作をする可能性があります。そういうことなので、「New Data」ボタンは押さないほうがいいです。

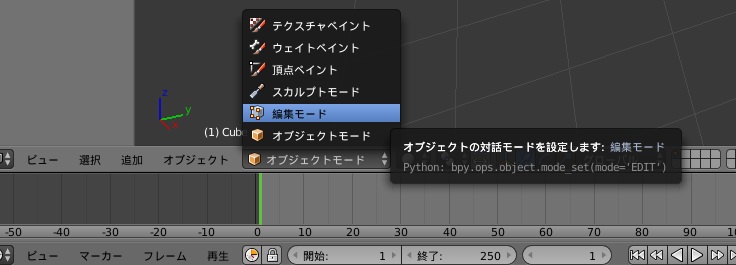
これで変更は完了しました。左下にある「**ユーザー設定の保存**」ボタンをクリック設定を保存し、ウィンドウの「X」ボタンをクリックしてウィンドウを閉じてください。

## 距離の単位を設定する

Blenderの初期状態は距離の単位が設定されていません。設定しなくてもモデルの作成自体に影響はないのですが、データを保存する際に単位が不正確だと、意図しない大きさで保存されてしまうことがあります。  
単位を設定するには、右側にある「シーン」アイコンをクリックしたあと、単位欄で「メートル法」のボタンをクリックするだけです。

## 編集モードに切り替える

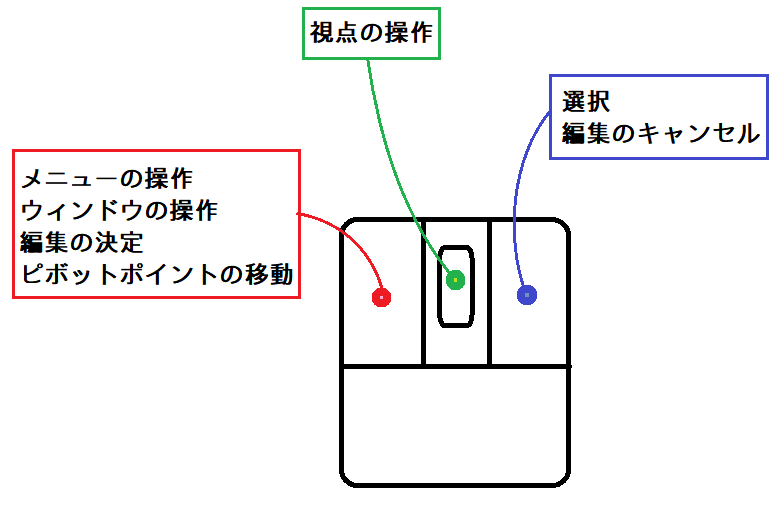
Blenderを起動した直後は「オブジェクトモード」という状態になっています。オブジェクトモードは、モデル単位で編集を行うためのモードです。頂点単位の編集を行うためには「編集モード」に切り替えなくてはなりません。モードの切り替えは左下にあるモード名をクリックし、切り替えたいモード名を選択するだけです。



また、編集モードは頻繁に使うので、Tabキーを押すことで現在のモードと編集モードを交互に切り替えることができます。それでは、いずれかの方法で「編集モード」に切り替えてください。

## Blenderの基本操作

Blenderの操作は3ボタンマウスとキーボードで行います。トラックパッドでは中央ボタンの操作ができないため、Blenderを操作することは難しいでしょう。安物でいいのでマウスを用意してください。  
基本的なマウスの使い方は次のとおりです。



## 視点の操作

視点の操作は中央ボタンとホイールで行います。操作は次の4種類です。

|  |  |
| --- | --- |
| 中央ボタンドラッグ | 視点の回転移動 |
| ホイール回転 | 視点の前進・後退 |
| Shift+中央ボタンドラッグ | 視点の右移動・左移動 |
| Shift+ホイール回転 | 視点の上昇・下降 |

ここで、ひとつ注意してほしいことがあります。Blenderの座標系は**「X軸が横方向、Y軸が奥行き方向、Z軸が高さ方向」**だということです。OpenGLとは異なるので気をつけてください。

## 編集対象の選択

右クリックで編集対象の頂点(または辺、面)を選択します。また、CキーあるいはBキーを押すと範囲選択モードに入ることができます。範囲選択モードでは、複数の頂点をまとめて選択できます。

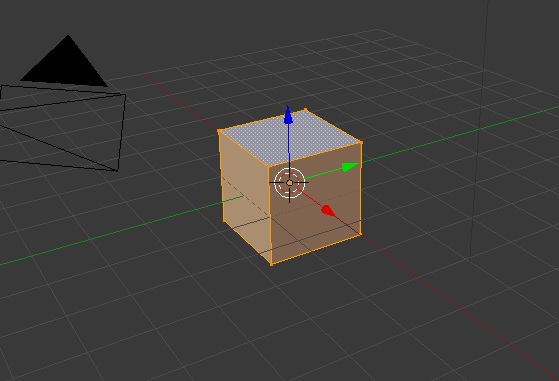
|  |  |
| --- | --- |
| 選択 | 右クリック |
| 選択解除 | Shift+右クリック |
| 範囲選択モード | Cキー(円形)、Bキー(矩形) |
| 範囲選択モード解除 | 左クリック, ESCキー |
| 範囲選択モード中の選択 | 左ボタンドラッグ |
| 範囲選択モード中の選択解除 | Shift+左ボタンドラッグ |
| すべて選択 | Aキー |
| すべて選択解除 | Aキー |
| 選択中の頂点から辿れる全ての頂点を選択(リンク選択) | Ctrl+Lキー |

## 編集の取り消しとやりなおし

取り消しはWindows標準のCtrl+Zキーですが、やり直しはCtrl+Y**ではなくて**、Ctrl+Shift+Zです。

|  |  |
| --- | --- |
| 取り消し | Ctrl+Zキー |
| やり直し | Ctrl+Shift+Zキー |

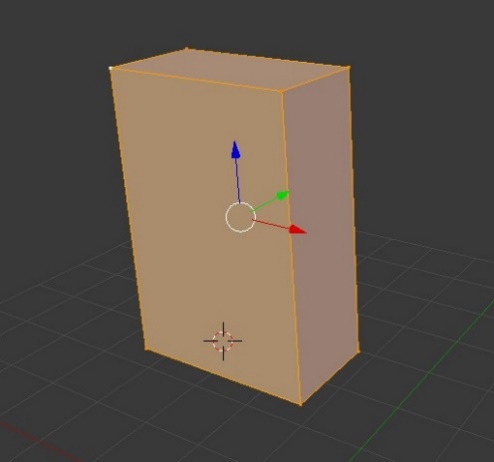
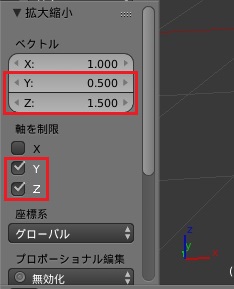
## 図形の移動

それでは、Minecraftの人間に毛が生えたようなモデルを作ってみましょう。まずは、頂点を移動させます。Aキーを押してすべての頂点を選択してください。すると、以下のように立方体の中心に赤緑青の矢印が表示されます。この矢印は「マニピュレータ」といいます。この矢印のいずれかを左ボタンでドラッグすると、矢印の軸に沿って選択している頂点を移動させることができます。上向きの矢印(Z軸)をドラッグして上に移動させてみましょう。

移動を行うと、左下の小さな領域に操作した内容が表示されます。この領域には最後の編集操作の履歴が表示されるようになっていて、ここを直接編集することで、編集結果を調整することができます。正確な距離を移動させる、といった操作はこの部分で行います。なお、移動操作は「Gキー」でも開始できます。Gキーによる移動は、軸とは無関係に好きな位置に移動できます(GはGrab(ぐらぶ=「つかむ」という意味)の頭文字)。

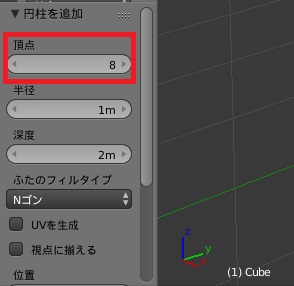
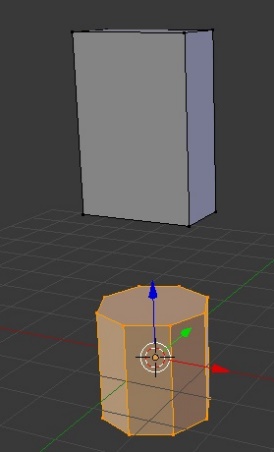
それでは、右下の「Z」の部分の数字を左クリックし、キーボードから「4.5」という数値を入力してください。単位をメートル法に変更しているので、この操作で移動する距離は4.5mになります。なお、「4,5」のようにドットではなくカンマを書いてしまうと、カンマより前が無視されて「5」になってしまうので注意しましょう。

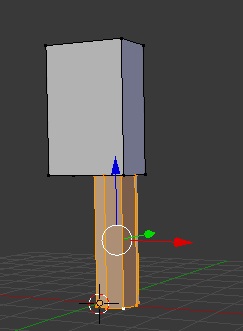
## 胴体を作る(図形の拡大・縮小)

この立方体が胴体になります。この図形を変形させて胴体らしくしていきましょう。図形を拡大縮小するにはSキーを押してマウスを動かします(SはScale(すけーる)の頭文字)。特定の軸だけ拡大縮小するには、左下の操作内容の表示部分にあるX,Y,Zのチェックボックスをチェックします。ここではYとZにチェックを入れ、Yの数値を0.5、Zの数値を1.5にしてください。厚みが半分になり、少し縦長になると思います。

## 足を追加する(図形の追加)

次は別の図形を追加しましょう。図形を追加するには左下の「追加」メニューをクリックするか、Shift+Aキーを押します。すると追加できる図形のリストが表示されるので、好きな図形をクリックすると、その図形が追加されます。追加される場所は「3Dカーソル」という、左の画像のように黒い十字と紅白の円を組みあせた図形の位置になります。3Dカーソルは左クリックするとその場所に移動してしまいます。原点に戻すには、左下のメニューから「メッシュ > スナップ > カーソル→原点」を選択してください。

3Dカーソルを原点に戻したら、「追加」メニューから「円柱」を選択してください。原点に円柱が追加されると思います。ただ、そのままでは頂点が多すぎます。少し減らしましょう。左下の操作内容欄を上にスクロールさせると、「頂点」という項目があります。この数値を「8」にしてください。これでかなり頂点数を減らせるでしょう。

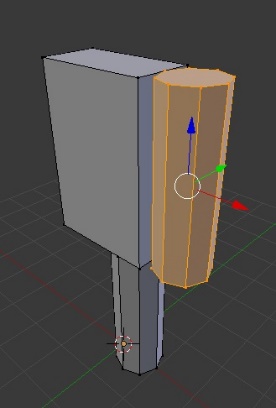
次は円柱を細くします。Sキーを押してXとYを0.5倍、Zを1.5倍にしてください。細くしたら、今度は上矢印(青色)を左ボタンドラッグして上の直方体とつなげてください。1.5m移動させればぴったりの位置になると思います。さらに右矢印(赤色)を左ボタンドラッグして0.5m移動させてください。  
左の画像のようになっていればO.K.です。

## 図形の削除

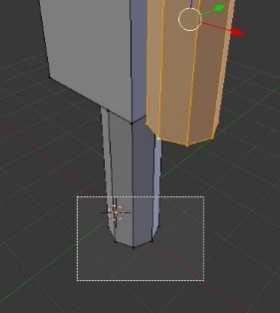
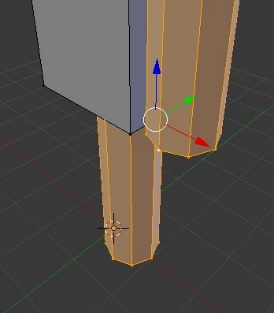
図形を削除するにはXキーかDelキーを押します。すると、削除する対象のリストが表示されます。図形全体を選択しているときは「頂点」「辺」「面」のどれを選んでも図形が削除されます。図形の一部分だけを選択しているときは、消したい要素を選んで消すことができます。

## 腕を作る(図形のコピー)

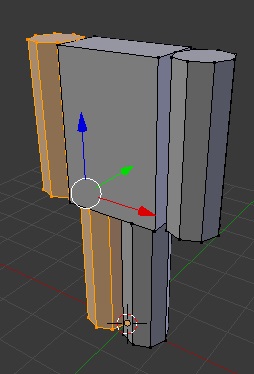
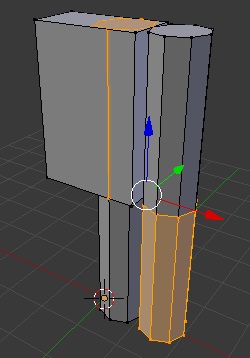
次は足をコピーして腕を作りましょう。選択している図形をコピーするにはShift+Dキーを使います(DはDuplicateの頭文字)。コピーしたあとは自動的に移動モードになり、マウスを動かして好きな位置で左クリックするとその位置にコピーされます。コピー元と同じ位置に配置したい場合は右クリックで移動モードをキャンセルします。移動をキャンセルしてもコピー自体は行われます。いずれの場合でも、コピー元の図形の選択状態は解除され、コピーした図形だけが選択状態になります。

それでは、Shift+Dキーで円柱をコピーしてください。コピーしたらすぐに右クリックで移動をキャンセルしましょう。うっかり左クリックしてしまった場合はCtrl+Zキーで操作を取り消して、Shift+Dキーでコピーするところからやり直してください。  
キャンセルしたら、上向き矢印(青色)を左ボタンドラッグして、矢印の向いている方向に3m移動させてください。次に右向き矢印(赤色)を左ボタンドラッグして、矢印の向いている方向に0.5m移動させ、左の画像のように、コピーした円柱が直方体の右側に配置されるようにしてください。

## 反対側の腕と足を作る(図形の回転)

図形を回転させる前に、下側の円柱も選択します。まずBキーを押して範囲選択モードに入り、円柱の下端を左ボタンドラッグして、いくつかの頂点を選択頂点に追加してください。  
追加したら、Ctrl+Lキーを押して円柱全体を選択対象に加えます。

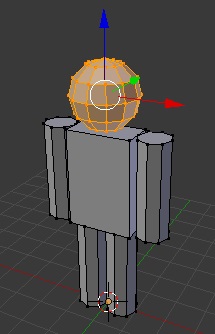
次に、Shift+Dキーで選択している2つの円柱をコピーします。コピーしたらすぐに右クリックでキャンセルしてください。これで準備完了です。

それでは、図形を回転させましょう。  
図形を回転させるにはRキーを押します(Rotate(ろーてーと)の頭文字)。マウスを動かすと図形がぐるぐる回転します。このときの回転軸は、視点の向きになっています。しかし、視点の向きを軸にした回転では、思い通りの回転を行うことが難しいです。そこで、適当に左クリックで回転を決定したあと、左下の操作履歴欄の角度を「180°」にし、軸を制御のZにチェックを入れてください。これで、Z軸に対してぴったり180°回転させることができます。

回転させたら、赤い矢印を左ボタンドラッグして2つの円柱を直方体の反対側に移動させてください。移動距離はぴったり「-2m」です。  
左の画像のようになっていればO.K.です。

## 頭部を作る

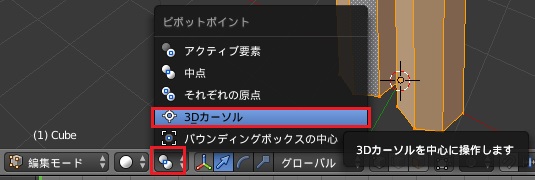
最後に頭部を追加します。頭部は球状にしましょう。Shift+Aキーを押して、図形リストから「UV球」を選択してください。UV球もそのままでは頂点が多すぎます。左下の編集履歴で「セグメント」を12に、「リング」を6にしてください。

そして上向き矢印(青色)を左ボタンドラッグして、上に7m移動してください。左の中央の画像のようになっていればO.K.です。

ところで、頂点数をけちったために、あまり球状には見えなくなってしまいました。しかしこれは、「面のシェーディング」を切り替えれば改善できます。シェーディングを切り替えるには、ウィンドウの左にある「シェーディング/UV」タブを左クリックし、「面」のところの「スムーズ」ボタンを押します。頭部の角ばった雰囲気がなくなればO.K.です。

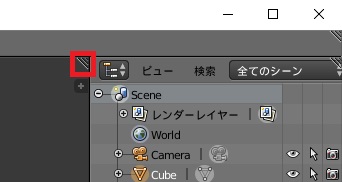
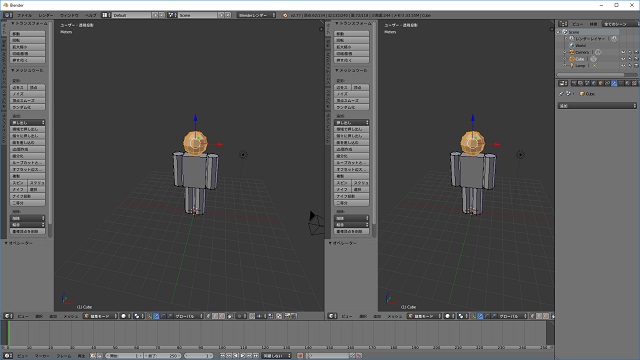
## 図形のサイズを調整する

作成した人間の身長は8mもあります。これでは巨人なので、相応の身長に縮めましょう。ウィンドウ下部にある2つの白い球が重なったようなアイコンをクリックすると、「ピボットポイント」という「編集操作の原点」になる点を変更できます。一覧から3Dカーソルを選択してください。



ここで3Dカーソルがワールド原点にない場合、Shift+Sキーで3Dカーソルを原点に移動してください。  
3Dカーソルが原点にあることを確認したら、Sキーを押して身長が170cmになるように縮小しましょう。ぴったり170cmにするには、X,Y,Zの全ての拡大率ベクトルに0.2125を設定します(1.7/8=0.2125)。なお、4桁以上の少数を入力した場合、表示は4桁目を四捨五入したものになりますが、編集操作は入力した値がそのまま反映されます。  
これでモデルは完成です。次はモデルにテクスチャを割り当てて、色を塗っていきましょう。

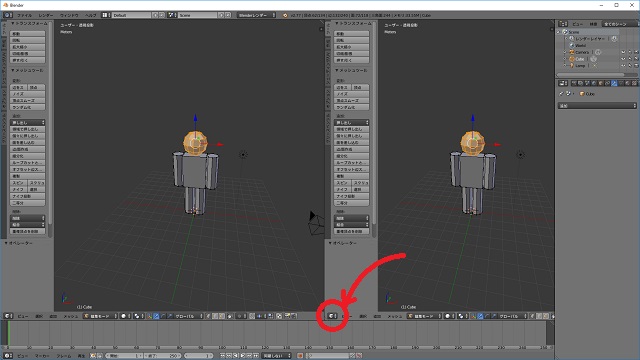
## ウィンドウの分割

テクスチャを追加するには、テクスチャ操作用のウィンドウを表示しなくてはなりません。そこで、ウィンドウを分割し、一報をテクスチャようにしようと思います。それぞれのウィンドウの右上には小さな斜線部分があり、この部分をドラッグするとウィンドウを分割することができます。

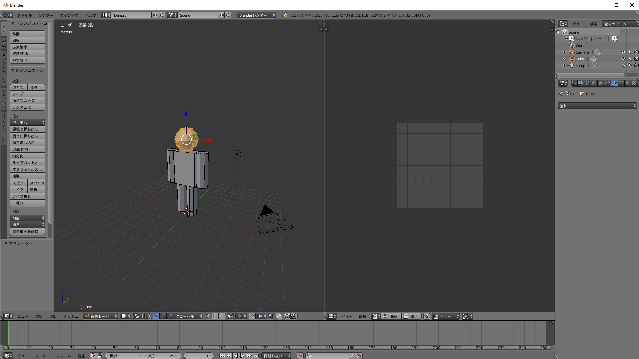
この斜線を左に左ボタンドラッグすればウィンドウが左右に分割され、下に左ボタンドラッグすれば上下に分割されます。また、分割したウィンドウを統合するには、右に左ボタンドラッグするか(左右分割の場合)、上に左ボタンドラッグします(上下分割の場合)。

では、この斜線部分を左にドラッグしてウィンドウを分割し、右の画像のようにしてください。

## 表示内容(エディター・タイプ)の変更

分割しただけでは同じ内容のウィンドウが2つに増えるだけです。この図形を表示するィンドウは「3D ビュー」と言いますが、テクスチャを編集するには「UV画像エディター」に変更しなければなりません。エディター・タイプを変更するには、ウィンドウの左下にある「エディター・タイプ・アイコン」を左クリックします。このアイコンは、現在のエディター・タイプを示しています。

それでは、エディター・タイプ・アイコンを左クリックしてください。すると、選択可能なエディター・タイプがリスト表示されますので、「UV画像エディター」を選択してください。表示内容が左下のように変化すればO.K.です。



## テクスチャを追加する



テクスチャを追加するにはウィンドウ下部の「新規」と書かれたボタンを押します。すると、左の画像のように、追加するテクスチャの名前や大きさを設定するウィンドウが開きます。開いたら、名前を「human」、幅を「512」、高さを「512」に変更してください。変更したら「OK」ボタンを押すと、テクスチャが追加されます。

## テクスチャを割り当てる

ところで、Blenderでは頂点ごとに異なるテクスチャを割り当てることができます。現在はたまたま選択中だった頭部にのみhumanテクスチャが割り当てられており、胴体や腕、足には割り当てられていません。これらにも同じテクスチャを割り当てます。やり方は簡単です。まず、マウスカーソルを3Dビュー側に持っていき、Aキーを2回押して全ての頂点を選択します。次に、UV画像エディターの下部にあるメニューから、「新規」ボタンの左にあるアイコンをクリックしてください。

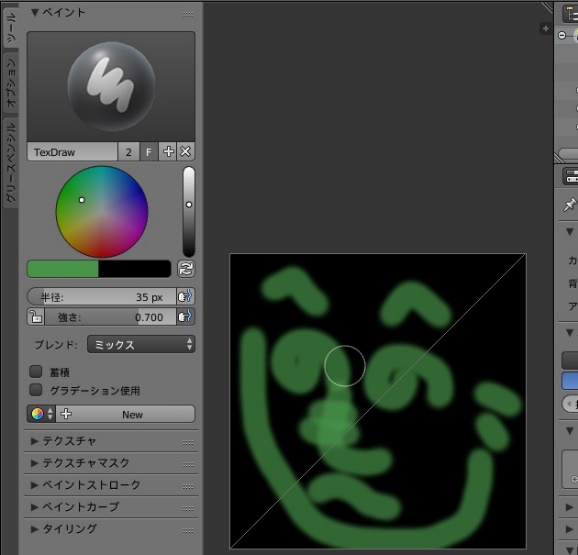
すると、選択可能なテクスチャの一覧が表示されますので、「human」を選びます。これでテクスチャの割り当ては完了です。

## テクスチャに色を塗る

それでは、テクスチャに色を塗ってみましょう。UV画像エディターの下部にあるメニューを見てください。「ビュー」と書かれている部分があると思います。これはUV画像エディターの操作モードを示しています「ビュー」は画像閲覧モードです。



この「ビュー」をクリックすると選択可能な操作モードの一覧が表示されます。一覧から「ペイント」を選択してください。これでテクスチャに色をぬることができます。黒い画像の上で左ボタンドラッグすると色が塗れると思います。

ペンの太さや種類、色を変更するにはTキーを押します。すると、左の画像のようなツールが表示されます。円形のグラデーション部分をクリックすると色を変えられます。その右の白黒グラデーションをクリックすると明るさを変えられます。その下の「半径」の数値を変更すると、ペンの太さを変えられます。ツールを閉じるにはTキーを押します。

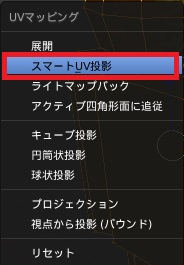
## 3Dビューでテクスチャを確認する

色を塗ったテクスチャがどのように表示されるかを確認するには、3Dビューの描画モードを切替えます。  
3Dビューの下部のメニューを見てください。編集モードと書かれている部分の右側に「白い球」のアイコンがあると思います。この部分をクリックすると、選択可能な描画モードの一覧が表示されます。



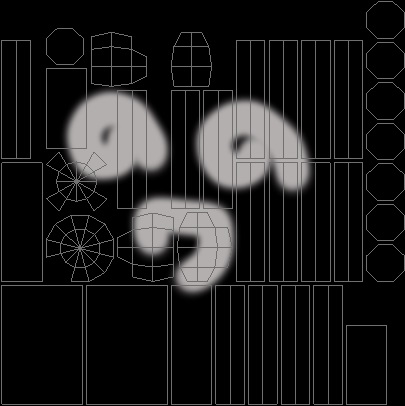
一覧から「テクスチャ」を選んでください。すると、テクスチャが貼られた状態で図形が表示されるようになります。

## UVマッピング

描画モードを変更しても、図形は真っ黒でうまくテクスチャが貼られているようには見えないと思います。これは、適切なUV座標を設定していないからです。幸いなことに、Blenderには一発でいい感じのUVを設定してくれる機能が用意されています。それが「UVマッピング」機能です。この機能を使うには「Uキー」を押します。

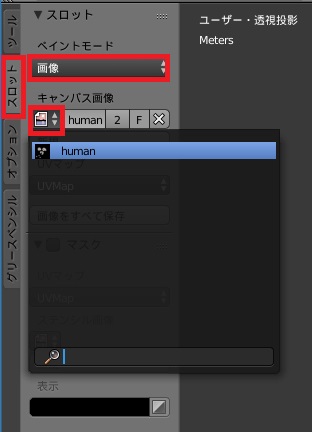
すると、UV座標を設定する方法の一覧が表示されますので、「スマートUV投影」を選択してください。

そうするとさらに、UVマッピングの設定を行うウィンドウが出ます。このウィンドウの「島の余白」という部分の数値を「0.05」にしてください。

「島の余白」は、異なる図形間のUV座標の間隔を制御します。間隔が狭すぎると、塗った色が無関係な図形にもれ出す可能性があります。逆に広すぎると、テクスチャの殆どの部分が余白になり、無駄が多くなります。この数値はテクスチャの大きさによりますが、512x512では0.05くらいが適当でしょう。

「スマートUV投影」を実行すると、UV画像エディターの表示が左の画像のように変化します。テクスチャ上に表示された細い白線は、スマートUV投影によって設定されたモデルのUV座標です。

## 図形に直接ペイントする

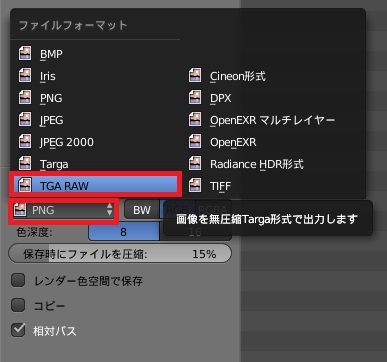
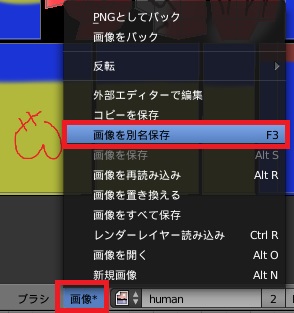
スマートUV投影されたテクスチャにペイントするのは、モデルの特定の場所に色を塗るためには、テクスチャのどこを塗ればいいのかが分かりにくいため、少し難しいです。これも幸いなことに、Blenderには図形を直接ペイントできる機能が備わっています。3Dビュー下部の「編集モード」の文字をクリックし、表示されたモード一覧から「テクスチャペイント」を選択してください。

テクスチャペイントを選択すると、図形が紫色になると思います。これはペイント対象のテクスチャが選択されていないためです。ペイント対象のテクスチャを選択するには、左側にある「スロット」タブをクリックし、「ペイントモード」を「画像」に変更し、そしてキャンパス画像を描画したい画像に設定します。今回はキャンパス画像に「human」を設定してください。

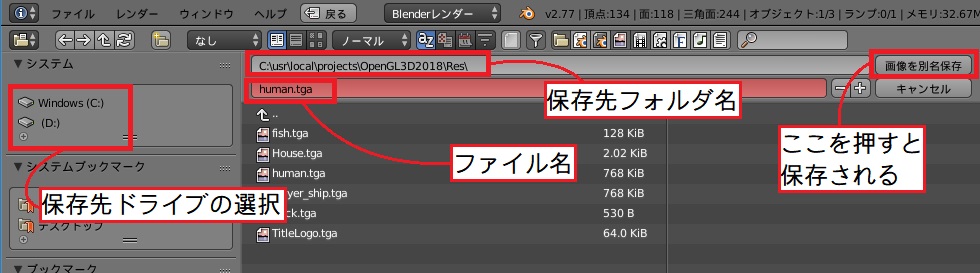
これで図形に直接ペイントできるようになりました。3Dビュー内で左ボタンドラッグして、図形に色を塗ってみてください。図形に色が塗られ、すぐにUV画像エディターのほうにも反映されることが分かると思います。

## テクスチャの保存

色塗りが済んだらテクスチャをファイルに保存しましょう。UV画像エディター下部のメニューから「画像 > 画像を別名保存」を選択してください。すると、ファイル名と保存先を選択する画面が表示されます。左下の「画像を別名保存」と書かれた部分でファイル形式を選択できます。「PNG」と書かれた部分をクリックしてファイル形式一覧を表示し、「TGA RAW」を選択してください。

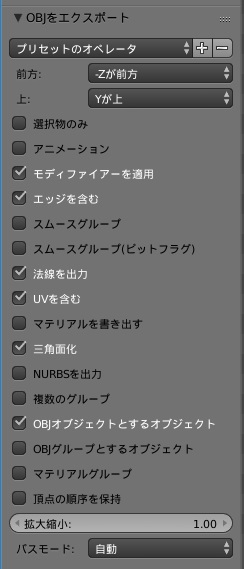


保存先フォルダの選択とファイル名の変更は上部のテキストボックスで行います。なお、Blenderのフォルダ選択はシングルクリックで操作します。いつもの癖でダブルクリックしないようにしてください。



保存先は、みなさんのプロジェクトのResフォルダにしてください。ファイル名は「human.tga」のままで大丈夫です。保存先のResフォルダを選択したら、右上の「画像を別名保存」というボタンを押すとテクスチャが保存されます。

## 図形を保存する

最後に図形を保存しましょう。まずBlender専用の形式で保存します。ウィンドウ上部のメニューから「ファイル > 保存」を選択するか、Ctrl+Sキーを押すと、blendファイルを保存するウィンドウが開きます。このウィンドウは基本的にはテクスチャの保存と同じです。保存先フォルダを選択し、ファイル名を設定してから「Blenerファイルを保存」ボタンを押すだけです。新しいモデルを作成したときは、ファイル名が「untitled.blend」になっていますので、「human.blend」などに変更してください。一度ファイルを保存すれば、以後はCtrl+Sを押すだけで自動的に上書き保存されます。

BlenderファイルはBlenderの様々な設定を全て記録したものです。モデルを調整するにはBlenderでこのファイルを開きます。しかし、わたしたちが本当に必要としているのはOBJファイルですから、続けてOBJファイルを保存していきましょう。  
OBJファイルを保存するには、ウィンドウ上部のメニューから「ファイル > エクスポート > Wavefront(.obj)」を選択します。すると、テクスチャファイルの保存やBlenderファイルの保存と同じようなウィンドウが開きます。

ところで、このテキストで作成したOBJファイル読み込みプログラムは、必要な情報が適切に設定されたOBJファイルしか正しく読み込めません。そこで、読み込みプログラムのためにBlenderのエクスポート設定を調整します。ウィンドウの左下に、左の画像と同じ領域があると思います。その部分を、この画像と同じ状態にしてください。この状態でOBJファイルを保存すれば、OBJファイル読み込みプログラムで読み込むことが可能です。  
保存先フォルダはみなさんのプロジェクトのResフォルダにしてください。  
ファイル名は「human.obj」としてください。

**[課題 04]** human.tgaとhuman.objを読み込み、家や木などと重ならない位置に表示してください。

## OBJファイルのインポート

BlenderにOBJファイルを読み込むには、上部のメニューから「ファイル > インポート > Wavefront(.obj)」を選択します。すると、ファイルを保存するときとよく似たファイル選択ウィンドウが表示されますので、読み込ませたいOBJファイルを選択します。右上の「OBJをインポート」ボタンを押すと、選択したOBJファイルが読み込まれます。

「インポート」は現在のシーンにモデルを追加することに注意してください。何もない状態にするには、上部のメニューから「ファイル > 新規」を選択し、「スタートアップファイルの再読み込み」をクリックします。すると、立方体がひとつだけ表示されたシーンになりますので、この立方体を選択してXキーで削除し、単位をメートル法に設定し直します。  
これで何もない状態になります。

**[課題 05]** Blenderにhouse.objをインポートし、「煙突的なもの」を追加してエクスポートしてください。