コンピュータアニメーション特論

第3回　幾何形状データの読み込み

九州工業大学 情報工学研究院　尾下真樹

幾何形状データの読み込み

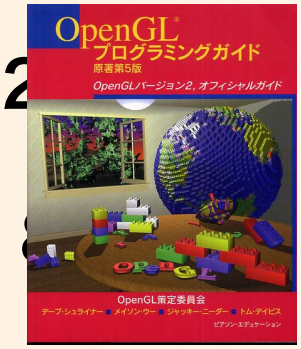
- 幾何形状データ
  - 多くのソフトウェアでは、幾何形状データ（モデルデータ）が必要となる
  - 通常、幾何形状データはあらかじめ作成されて、ファイルに格納されている
    - ソース中に直接モデルデータを記述するのは現実的ではない
  - ファイルからの読み込み処理が必要となる
  - 幾何形状データの描画も必要

今日の内容

- 復習：ポリゴンモデルの描画
- 幾何形状データ
- ファイル形式
- データ構造の定義と描画処理
- ファイル読み込み処理の作成
  - Cによる読み込み処理の実装
  - C++による読み込み処理の実装
- 頂点配列の利用、高度な幾何形状データ処理

参考書

- OpenGLの定番の本
  - OpenGLプログラミングガイド（赤本），12
    円
  - OpenGLリファレンスマニュアル（青本），
    円
    - ピアソン・エデュケーション出版
    - 中・上級者向け



- 学部の授業の演習資料
  - http://www.cg.ces.kyutech.ac.jp/lect
    /
  - システム創成情報工学科 「コンピュータ
    グラフィックスS」 の講義・演習資料
  - OpenGLの使い方を段階的に学べる演習資料

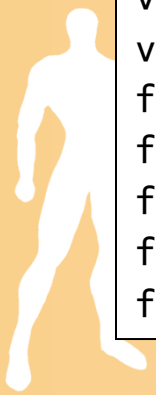
幾何形状データファイルの例

```
# Sample Obj Data (Pyramid)

mtllib pyramid.mtl
usemtl green
v    0.0    1.0    0.0
v    1.0   -0.8    1.0
v    1.0   -0.8   -1.0
v   -1.0   -0.8   -1.0
v   -1.0   -0.8    1.0
vn   0.9    0.4    0.0
vn   0.0    0.4   -0.9
vn  -0.9    0.4    0.0
vn   0.0    0.4    0.9
vn   0.0   -1.0    0.0
f   1//1   2//1   3//1
f   1//2   3//2   4//2
f   1//3   4//3   5//3
f   1//4   5//4   2//4
f   5//5   4//5   3//5   2//5
```

```
# Sample Obj Data (Pyramid)

newmtl green
Ka 0 0 0
Kd 0.3 0.8 0.3
Ks 0.9 0.9 0.9
Ns 20
```

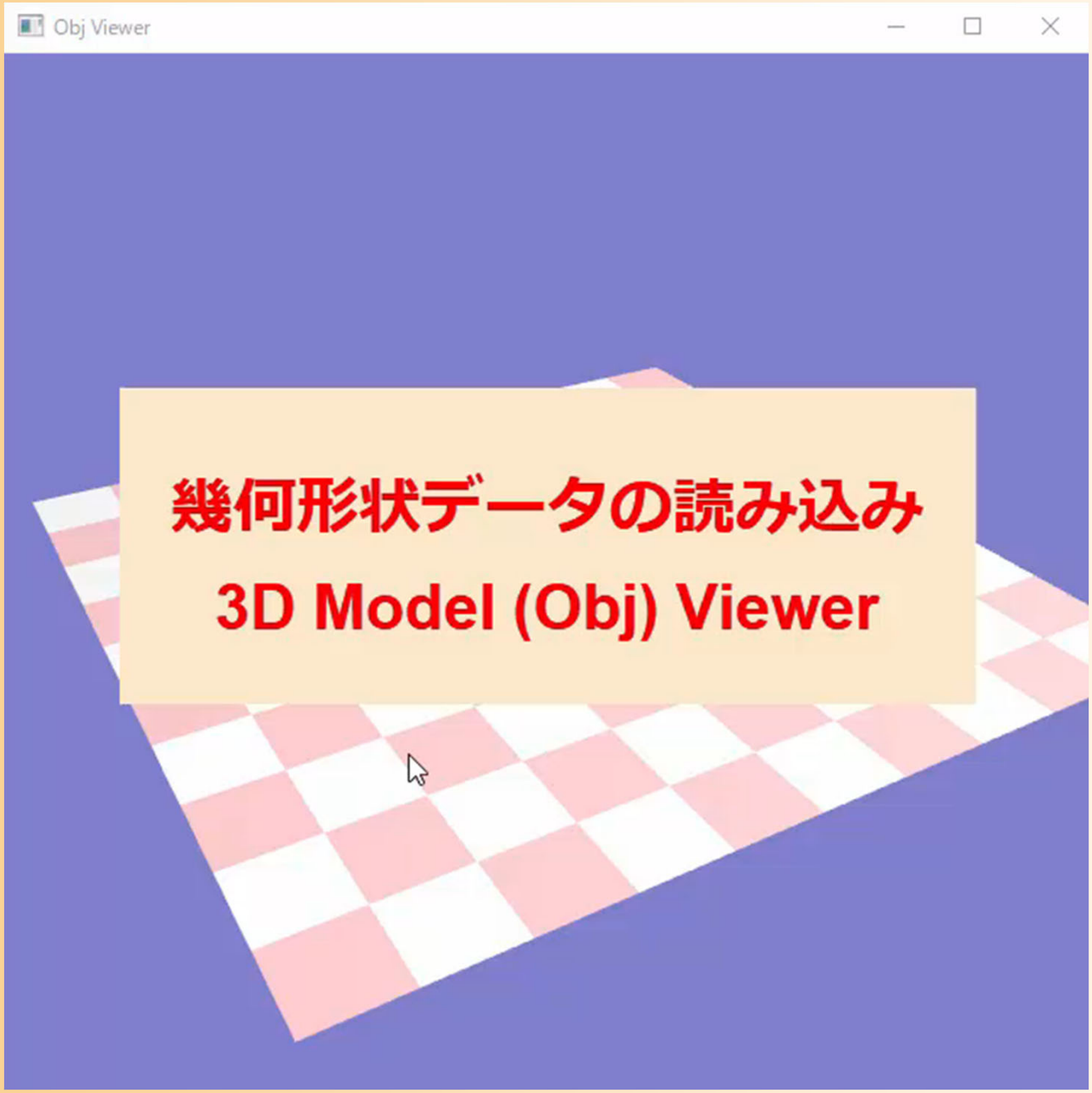
四角すいの形状データファイル
(pyramid.obj)(左)

四角すいの材質データファイル
(pyramid.mtl)(上)



デモプログラム

- 幾何形状データの読み込みと描画
  - 幾何形状データ（obj形式ファイル）を読み込んで描画
  - マウスドラッグによる視点操作も可能（前回の講義で扱った内容）


Obj Viewer
幾何形状データの読み込み
3D Model (Obj) Viewer

今日の内容

- 復習：ポリゴンモデルの描画
- 幾何形状データ
- ファイル形式
- データ構造の定義と描画処理
- ファイル読み込み処理の作成
  - Cによる読み込み処理の実装
  - C++による読み込み処理の実装
- 頂点配列の利用、高度な幾何形状データ処理

ポリゴンモデルの描画

ポリゴンモデルの描画

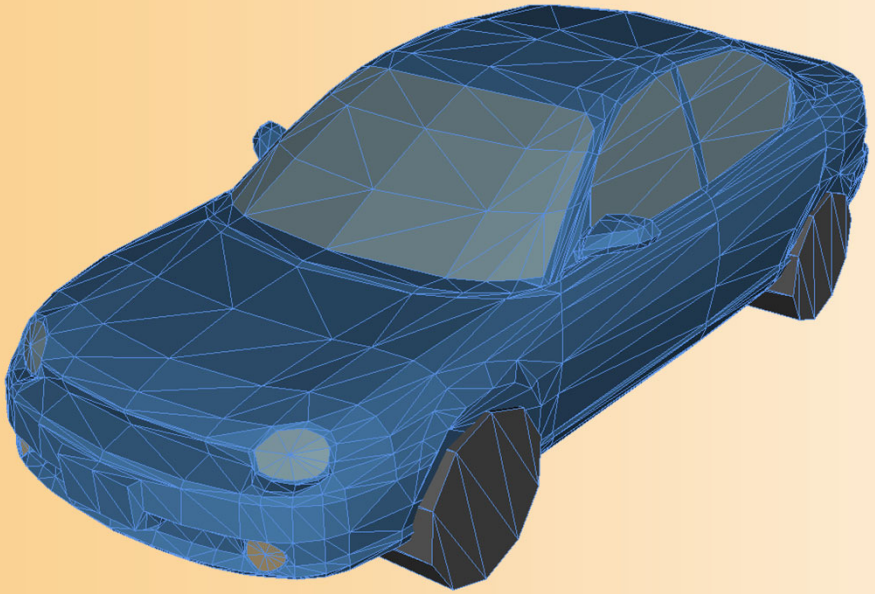
- 復習：レンダリング・パイプライン
- ポリゴンの描画
- ポリゴンモデルの描画

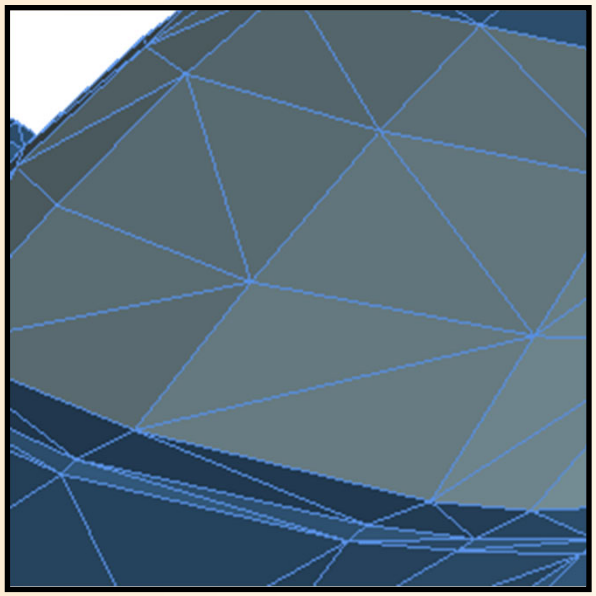
グラフィックスライブラリの機能

- 基本的なレンダリングの機能を提供
  - ポリゴンモデルによるモデリング
  - Ｚバッファ法によるレンダリング
  - スクリーン座標系への座標変換の適用
  - 局所照明モデルによるシェーディング
  - テクスチャマッピングの適用
- プログラマブルシェーダ（GPUプログラミング）により、座標変換やシェーディングの処理を拡張可能（本講義では説明は省略）

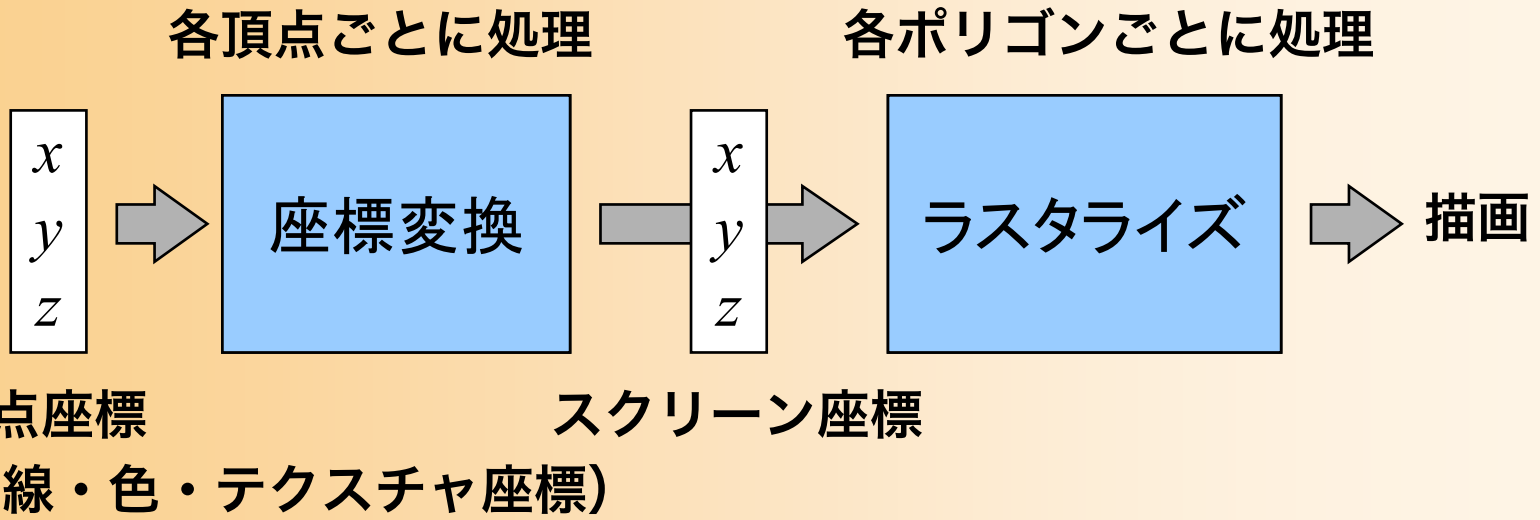
ポリゴンモデル

- 物体の表面の形状を、多角形（ポリゴン）の集まりによって表現する方法
  - 最も一般的なモデリング技術
  - 本講義の演習でも、ポリゴンモデルを扱う



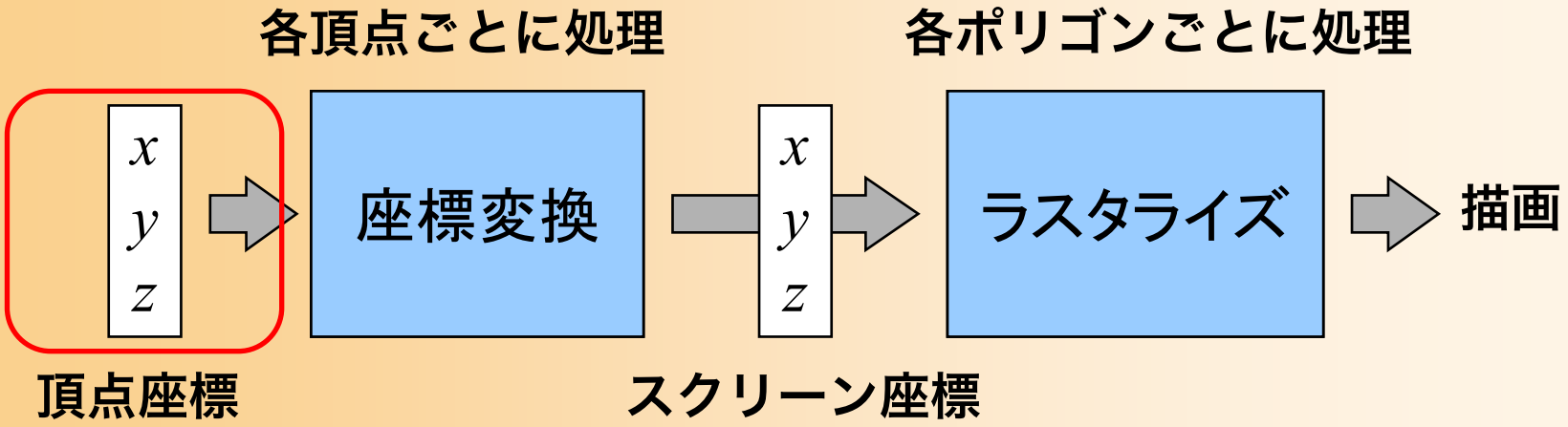


レンダリング・パイプライン



- レンダリング・パイプライン（ビューイング・パイプライン、グラフィックス・パイプライン）
  - 入力されたデータを、流れ作業（パイプライン）で処理し、順次、画面に描画
  - ポリゴンのデータ（頂点データの配列）を入力
  - いくつかの処理を経て、画面上に描画される

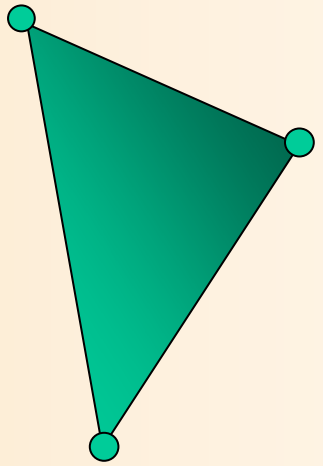
描画データの入力



- 物体の情報を入力
  - ポリゴンを構成する頂点の座標・法線・色・テクスチャ座標などを入力
- 表面の素材などを途中で変える場合は、適宜設定を変更

ポリゴンデータ

- ポリゴンの持つ情報
  - 各頂点の情報
    - 座標
    - 法線
    - 色
    - テクスチャ座標
  - 面の向き
    - 頂点の順番によって面の向きを表す
    - 視点と反対向きの面は描画しない（背面除去）
    - 法線と面の向きは別なので注意が必要



ポリゴンモデルの描画

- 復習：レンダリング・パイプライン
- ポリゴンの描画
- ポリゴンモデルの描画

ポリゴンの描画

- glBegin（） ～ glEnd（） を使用

```
glBegin( プリミティブの種類 );
        この間にプリミティブを構成する頂点データを指定
glEnd();
```

※ 頂点データの指定は、一つの関数で、ポリゴンを構成するデータの一つのみを指定するようになっている

- プリミティブの種類
  - GL_POINTS（点）、GL_LINES（線分）、GL_TRIANGLES（三角面）、GL_QUADS（四角面）、GL_POLYGON（ポリゴン）、他

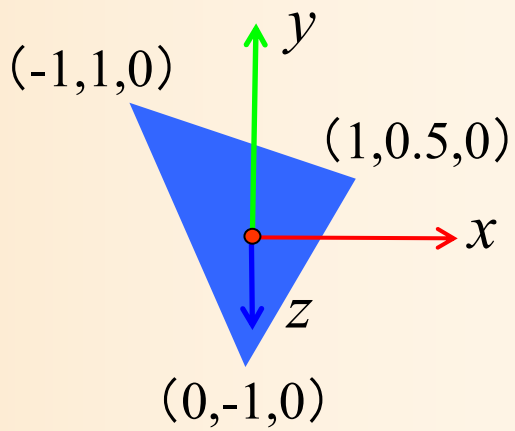
頂点データの指定

- glColor3f（ r, g, b ）
  - これ以降の頂点の色を設定
- glNormal3f（ nx, ny, nz ）
  - これ以降の頂点の法線を設定
- glVertex3f（ x, y, z ）
  - 頂点座標を指定
  - 色・法線は、最後に指定したものが使用される

ポリゴンの描画の例（1）

- 1枚の三角形を描画
  - 各頂点の頂点座標、法線、色を指定して描画
  - ポリゴンを基準とする座標系（モデル座標系）で頂点位置・法線を指定

```
glBegin( GL_TRIANGLES );
        glColor3f( 0.0, 0.0, 1.0 );
        glNormal3f( 0.0, 0.0, 1.0 );
        glVertex3f(-1.0, 1.0, 0.0 );
        glVertex3f( 0.0,-1.0, 0.0 );
        glVertex3f( 1.0, 0.5, 0.0 );
glEnd();
```

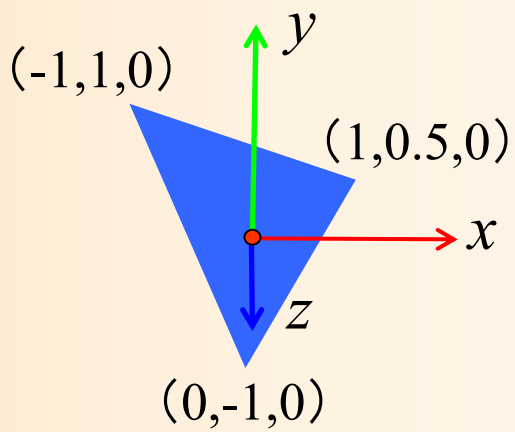


GL_TRIANGLES が指定されているので、3つの頂点をもとに、1枚の三角面を描画（6つの頂点が指定されたら、2枚描画）

ポリゴンの描画の例（1）

- 頂点の色・法線は、頂点ごとに指定可能
  - 指定しなければ、最後に指定したものが使われる

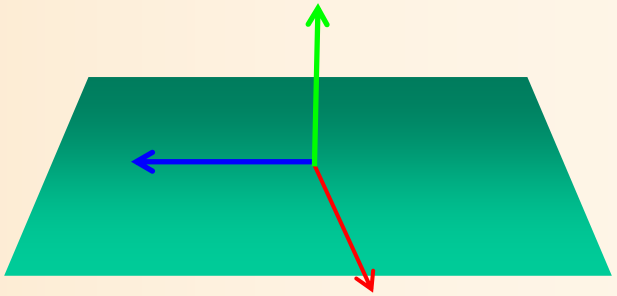
```
glBegin( GL_TRIANGLES );
        glColor3f( 0.0, 0.0, 1.0 );
        glNormal3f( 0.0, 0.0, 1.0 );
        glVertex3f(-1.0, 1.0, 0.0 );

        glColor3f( 0.0, 0.0, 1.0 );
        glNormal3f( 0.0, 0.0, 1.0 );
        glVertex3f( 0.0,-1.0, 0.0 );

        glColor3f( 0.0, 0.0, 1.0 );
        glNormal3f( 0.0, 0.0, 1.0 );
        glVertex3f( 1.0, 0.5, 0.0 );
glEnd();
```



ポリゴンの描画の例（2）

- １枚の四角形として地面を描画
  - 各頂点の頂点座標、法線、色を指定して描画
  - 真上（0,1,0）を向き、水平方向の長さ10の四角形

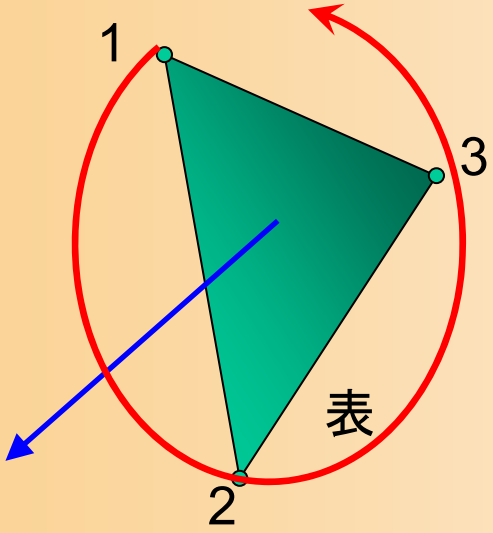
```
// 地面を描画
glBegin( GL_POLYGON );
        glNormal3f( 0.0, 1.0, 0.0 );
        glColor3f( 0.5, 0.8, 0.5 );

        glVertex3f( 5.0, 0.0, 5.0 );
        glVertex3f( 5.0, 0.0,-5.0 );
        glVertex3f(-5.0, 0.0,-5.0 );
        glVertex3f(-5.0, 0.0, 5.0 );
glEnd();
```

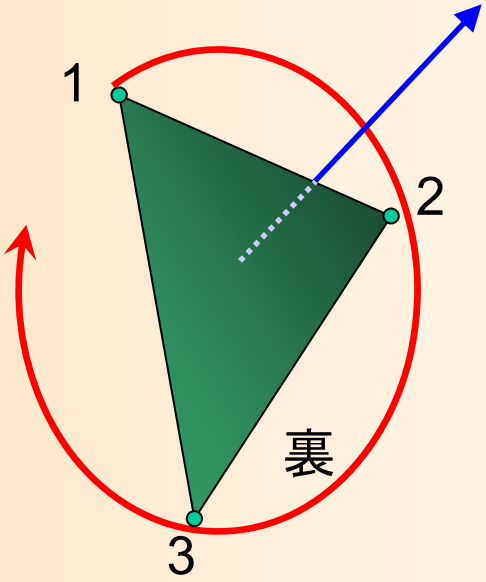


GL_POLYGON が指定されているので、n 個の頂点をもとに、n 角面を描画
（1度に1枚しか描画できない）

ポリゴンの向き

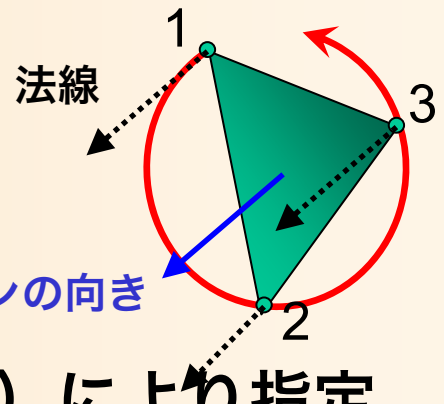
- 頂点の順番により、ポリゴンの向きを決定
  - 表から見て反時計回りの順序で頂点を与える
    - どちらの順序を表とするかは設定で変更できる
  - 視点と反対向きの面は描画しない（背面除去）
    - 頂点の順序を間違えると描画されないため要注意





法線とポリゴンの向き

- OpenGLでは、法線とポリゴンの向きは、独立の扱い（要注意）
- 法線
  - 頂点ごとに、関数（glNormal3f（））により指定
  - 光のモデルにより色を計算するために使用
- ポリゴンの向き
  - ポリゴンを描画するとき、頂点の順序により指定
  - 背面除去の判定に使用

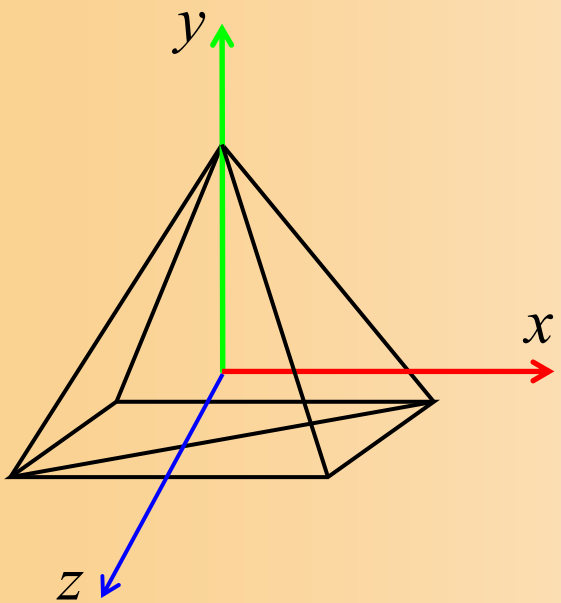


ポリゴンモデルの描画

- 復習：レンダリング・パイプライン
- ポリゴンの描画
- ポリゴンモデルの描画

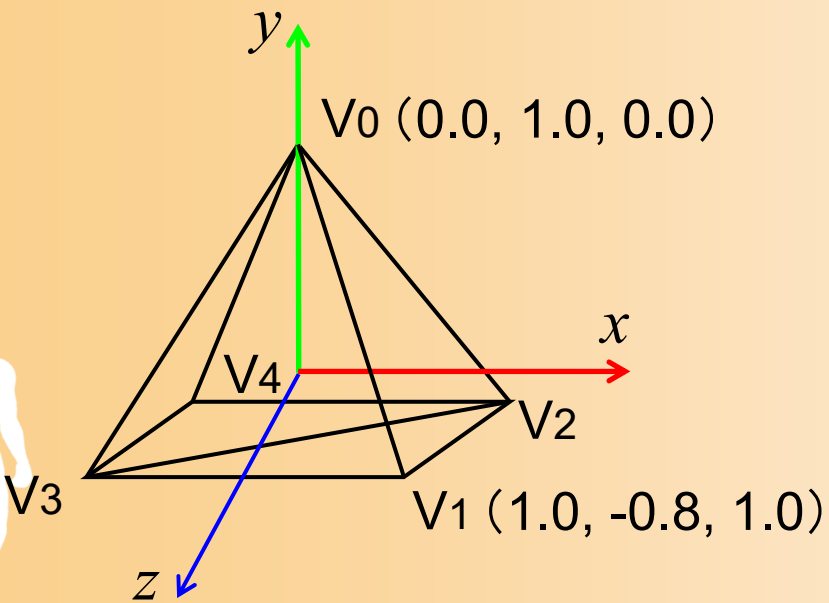
ポリゴンモデルの例（1）

- 四角すいを三角面の集まりとして描画
  - ５個の頂点と６枚の三角面により構成
  - 底面（四角形）は２枚の三角形に分割して表す



ポリゴンモデルの例（2）

- 四角すいを構成する頂点と三角面
  - 頂点座標
  - 三角面の頂点、面の法線



頂点座標

V0 (0.0, 1.0, 0.0)

V1 (1.0, -0.8, 1.0)

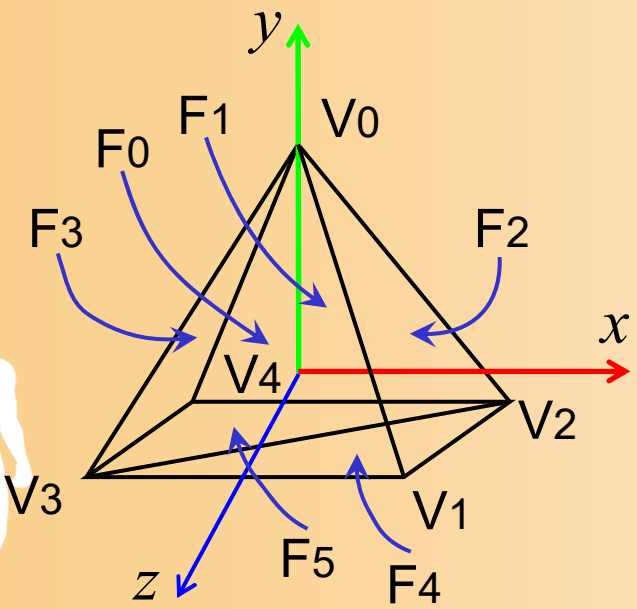
V2 (1.0, -0.8, -1.0)

V3 (-1.0, -0.8, 1.0)

V4 (-1.0, -0.8, -1.0)

ポリゴンモデルの例（3）

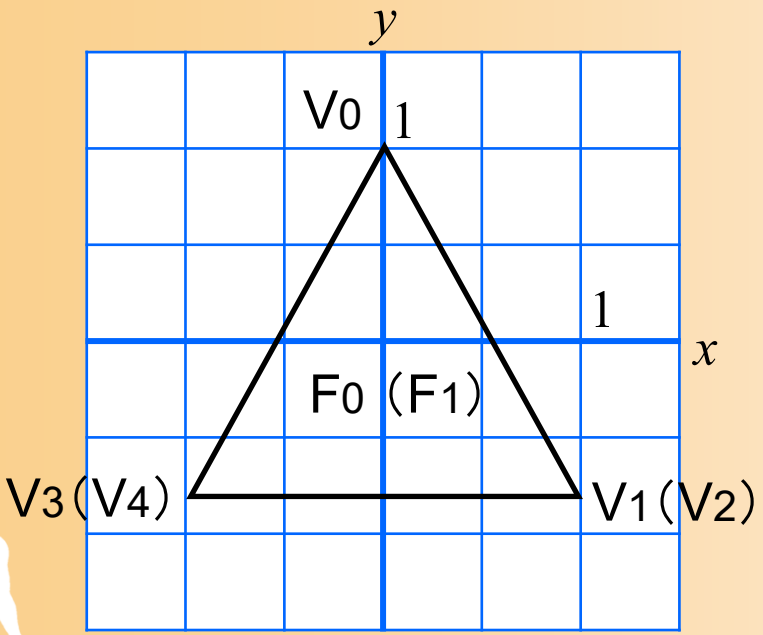
- 四角すいを構成する頂点と三角面
  - 頂点座標
  - 三角面の頂点、面の法線

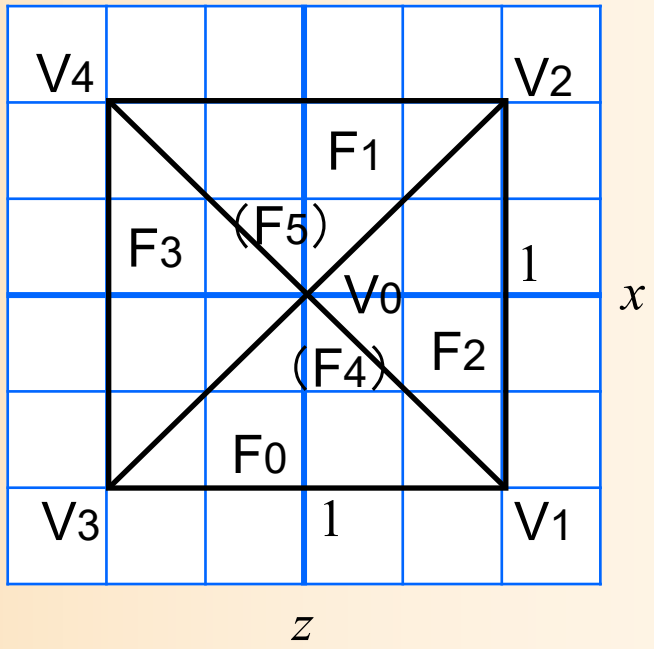


三角面	法線
F0 { V0, V3, V1 }	{ 0.0, 0.53, 0.85 }
F1 { V0, V2, V4 }	{ 0.0, 0.53, -0.85 }
F2 { V0, V1, V2 }	{ 0.85, 0.53, 0.0 }
F3 { V0, V4, V3 }	{ -0.85, 0.53, 0.0 }
F4 { V1, V3, V2 }	{ 0.0, -1.0, 0.0 }
F5 { V4, V2, V3 }	{ 0.0, -1.0, 0.0 }

三面図（1）

- xy平面、xz平面

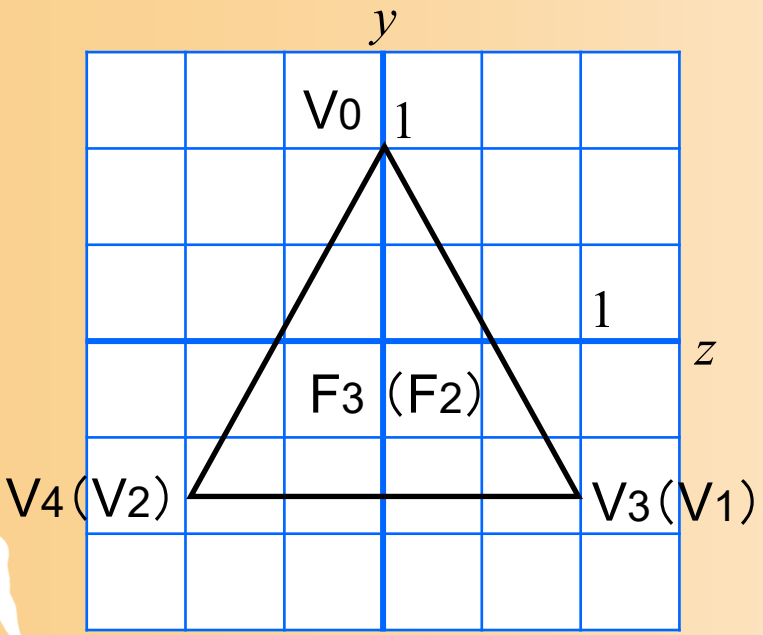




※ 括弧付きの頂点・面は裏側の頂点・面を表す

三面図（2）

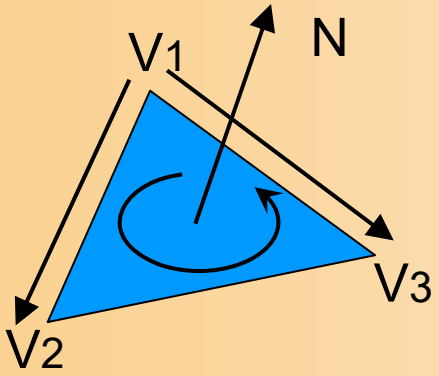
- yz平面



※ 括弧付きの頂点・面は裏側の頂点・面を表す

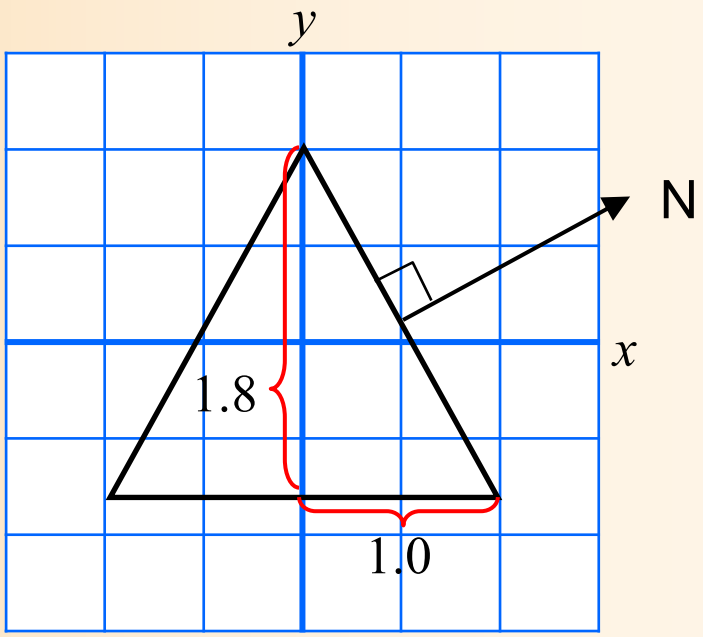
面の法線の計算方法

- ポリゴンの２辺の外積から計算できる



$$N = (V_3 - V_1) \times (V_2 - V_1)$$

長さが１になるように正規化



- 四角すいの場合、
  断面で考えれば、
  より簡単に求まる

ポリゴンモデルの描画方法

- いくつかの描画方法がある
  - プログラムからOpenGLに頂点データを与える方法として、いろいろなやり方がある
- 形状データの表現方法の違い
  - 頂点データのみを使う方法と、頂点データ＋面インデックスデータを使う方法がある
  - 後者の方が、データをコンパクトにできる
- OpenGLへのデータの渡し方の違い
  - OpenGLの頂点配列の機能を使うことで、より高速に描画できる

ポリゴンモデルの描画方法

- 方法1: glVertex() 関数に直接頂点座標を記述
  - 頂点データ（直接記述）、頂点ごとに渡す
- 方法2: 頂点データの配列を使用
  - 頂点データ、頂点ごとに渡す
- 方法3: 頂点データと面インデックスの配列を使用
  - 頂点データ＋面インデックス、頂点ごとに渡す
- 方法4: 頂点配列を使用
  - 頂点データ、OpenGLにまとめて渡す
- 方法5: 頂点配列と面インデックス配列を使用
  - 頂点データ＋面インデックス、 OpenGLにまとめて渡す

方法1 最も基本的な描画方法

- サンプルプログラムと同様の描画方法
  - glVertex 関数の引数に直接、頂点座標を記述
  - ポリゴン数×各ポリゴンの頂点数の数だけglVertex関数を呼び出す

四角すいの描画（1）

- 四角すいを描画する関数

```
void  renderPyramid1()
{
        glBegin( GL_TRIANGLES );
                // +Z方向の面
                glNormal3f( 0.0, 0.53, 0.85 );
                glVertex3f( 0.0, 1.0, 0.0 );
                glVertex3f(-1.0,-0.8, 1.0 );
                glVertex3f( 1.0,-0.8, 1.0 );

                ......
                以下、残りの5枚分のデータを記述
                ......
        glEnd();
}
```

ポリゴンモデルの描画方法

- 方法1: glVertex() 関数に直接頂点座標を記述
  - 頂点データ（直接記述）、頂点ごとに渡す
- 方法2: 頂点データの配列を使用
  - 頂点データ、頂点ごとに渡す
- 方法3: 頂点データと面インデックスの配列を使用
  - 頂点データ＋面インデックス、頂点ごとに渡す
- 方法4: 頂点配列を使用
  - 頂点データ、OpenGLにまとめて渡す
- 方法5: 頂点配列と面インデックス配列を使用
  - 頂点データ＋面インデックス、 OpenGLにまとめて渡す

方法2 頂点データの配列を使用

- 配列を使う方法
  - 頂点データを配列として定義しておく
  - glVertex() 関数の引数として配列データを順番に与える
- 利点
  - モデルデータが配列になってるので扱いやすい

頂点データの配列を使用（1）

- 配列データの定義

```
// 全頂点数
const int  num_full_vertices = 18;

// 全頂点の頂点座標
static float  pyramid_full_vertices[][ 3 ] = {
    { 0.0, 1.0, 0.0 }, {-1.0,-0.8, 1.0 }, { 1.0,-0.8, 1.0 },
    ・・・・
    {-1.0,-0.8,-1.0 }, { 1.0,-0.8,-1.0 }, {-1.0,-0.8, 1.0 } };

// 全頂点の法線ベクトル
static float  pyramid_full_normals[][ 3 ] = {
    { 0.00, 0.53, 0.85 }, { 0.00, 0.53, 0.85 }, { 0.00, 0.53, 0.85 },
    ・・・・
    { 0.00,-1.00, 0.00 }, { 0.00,-1.00, 0.00 }, { 0.00,-1.00, 0.00 } };
```

頂点データの配列を使用（2）

- 各頂点の配列データを呼び出す

```
void  renderPyramid2()
{
        int  i;
        glBegin( GL_TRIANGLES );
        for ( i=0; i<num_full_vertices; i++ )
        {
                glNormal3f( pyramid_full_normals[i][0],
                            pyramid_full_normals[i][1],
                            pyramid_full_normals[i][2] );
                glVertex3f( pyramid_full_vertices[i][0],
                            pyramid_full_vertices[i][1],
                            pyramid_full_vertices[i][2] );
        }
        glEnd();
}
```

各頂点ごとに繰り返し

法線・頂点を指定（i番目の頂点のデータを指定）

ポリゴンモデルの描画方法

- 方法1: glVertex() 関数に直接頂点座標を記述
  - 頂点データ（直接記述）、頂点ごとに渡す
- 方法2: 頂点データの配列を使用
  - 頂点データ、頂点ごとに渡す
- 方法3: 頂点データと面インデックスの配列を使用
  - 頂点データ＋面インデックス、頂点ごとに渡す
- 方法4: 頂点配列を使用
  - 頂点データ、OpenGLにまとめて渡す
- 方法5: 頂点配列と面インデックス配列を使用
  - 頂点データ＋面インデックス、 OpenGLにまとめて渡す

ここまでの方法の問題点

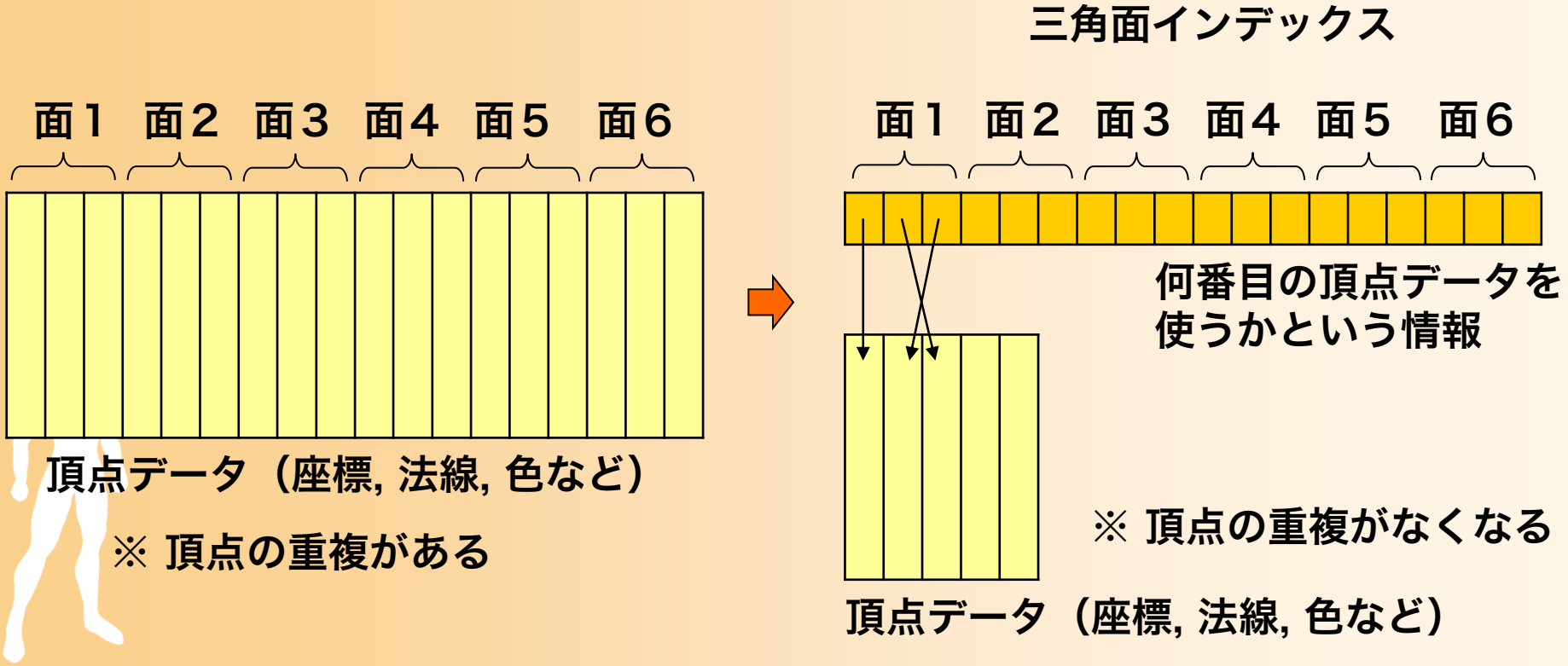
- 別の問題点
  - ある頂点を複数のポリゴンが共有している時、各ポリゴンごとに同じ頂点のデータを何度も記述する必要がある
    - 例：四角すいの頂点数は５個だが、これまでの方法では、三角面数６×３個＝１８個の頂点を記述する必要がある
  - OpenGLは、与えられた全ての頂点に座標変換などの処理を適用するので、同じモデルを描画する時でも、なるべく頂点数が少ない方が高速

方法3 三角面インデックスを使用

- 頂点とポリゴンの情報を別々の配列に格納
  - 頂点データの数を最小限にできる
- 描画関数では、配列のデータを順に参照しながら描画
- 必要な配列（サンプルプログラムの例）
  - 頂点座標（x,y,z）×頂点数
  - 三角面を構成する頂点番号（v0,v1,v2）×三角面数
  - 三角面の法線（x,y,z）×三角面数

三角面インデックス

- 頂点データの配列と、三角面インデックスの配列に分けて管理する



配列を使った四角すいの描画（1）

- 配列データの定義

```
const int  num_pyramid_vertices = 5;    // 頂点数
const int  num_pyramid_triangles = 6;   // 三角面数

// 角すいの頂点座標の配列
float  pyramid_vertices[ num_pyramid_vertices ][ 3 ] = {
          { 0.0, 1.0, 0.0 }, { 1.0,-0.8, 1.0 }, { 1.0,-0.8,-1.0 }, ・・・・・・
};

// 三角面インデックス（各三角面を構成する頂点の頂点番号）の配列
int  pyramid_tri_index[ num_pyramid_triangles ][ 3 ] = {
          { 0,3,1 }, { 0,2,4 }, { 0,1,2 }, { 0,4,3 }, { 1,3,2 }, { 4,2,3 }
};

// 三角面の法線ベクトルの配列（三角面を構成する頂点座標から計算）
float  pyramid_tri_normals[ num_pyramid_triangles ][ 3 ] = {
          { 0.00, 0.53, 0.85 }, // +Z方向の面
          ・・・・・・
```

配列を使った四角すいの描画（2）

- 配列データを参照しながら三角面を描画

```
void  renderPyramid3()
{
    int  i, j, v_no;
    glBegin( GL_TRIANGLES );
    for ( i=0; i<num_pyramid_triangles; i++ )
    {
        glNormal3f( pyramid_tri_normals[i][0], [i][1], [i][2] );
        for ( j=0; j<3; j++ )
        {
            v_no = pyramid_tri_index[ i ][ j ];
            glVertex3f( pyramid_vertices[ v_no ][0], [ v_no ][1], 
        }
    }
    glEnd();
}
```

各三角面ごとに繰り返し

三角面の各頂点ごとに繰り返し

面の法線を指定（i番目の面のデータを指定）

頂点番号を取得（i番目の面のj番目の頂点が、何番目の頂点を使うかを取得）

頂点座標を指定（v_no番目の頂点のデータを指定）

ポリゴンモデルの描画方法

- 方法1: glVertex() 関数に直接頂点座標を記述
  - 頂点データ（直接記述）、頂点ごとに渡す
- 方法2: 頂点データの配列を使用
  - 頂点データ、頂点ごとに渡す
- 方法3: 頂点データと面インデックスの配列を使用
  - 頂点データ＋面インデックス、頂点ごとに渡す
- 方法4: 頂点配列を使用
  - 頂点データ、OpenGLにまとめて渡す
- 方法5: 頂点配列と面インデックス配列を使用
  - 頂点データ＋面インデックス、 OpenGLにまとめて渡す

ここまでの方法の問題点

- 問題点
  - 頂点ごとに glVertex(), glNormal() 関数を呼び出す必要がある
  - 一般に関数呼び出しにはオーバーヘッドがかかるので、なるべく関数呼び出しの回数は少なくしたい
- OpenGLの頂点配列の機能を使えば、この問題を解決できる

頂点配列を使った描画方法

- 頂点配列
  - 配列データを一度に全部 OpenGL に渡して描画を行う機能
  - 頂点ごとに OpenGL の関数を呼び出して、個別にデータを渡す必要がなくなる
    - 処理を高速化できる
    - 渡すデータの量は同じでも、頂点配列を利用することで、処理を高速化できる
  - 実用では頂点配列の機能を使うのが一般的
    - 携帯端末用の OpenGL ES では、glBegin・glEnd関数は使えないため、頂点配列の使用が必須となる

方法4 頂点配列を使った描画方法

- 頂点配列を使った描画の手順
  1. 頂点の座標・法線などの配列データを用意
  2. OpenGLに配列データを指定（配列の先頭アドレス）
     - glVertexPointer（）関数、glNormalPointer（）関数、等
  3. どの配列データを使用するかを設定
     - 頂点の座標、色、法線ベクトル、テクスチャ座標など
     - glEnableClientState（）関数
  4. 配列の使用する範囲を指定して一気に描画
     - 配列データをレンダリング・パイプラインに転送
     - glDrawArrays（）関数

頂点配列を使用した描画（1）

- 配列データの定義（方法2と同じ）

```
// 全頂点数
const int  num_full_vertices = 18;

// 全頂点の頂点座標
static float  pyramid_full_vertices[][ 3 ] = {
    { 0.0, 1.0, 0.0 }, {-1.0,-0.8, 1.0 }, { 1.0,-0.8, 1.0 },
    ・・・・
    {-1.0,-0.8,-1.0 }, { 1.0,-0.8,-1.0 }, {-1.0,-0.8, 1.0 } };

// 全頂点の法線ベクトル
static float  pyramid_full_normals[][ 3 ] = {
    { 0.00, 0.53, 0.85 }, { 0.00, 0.53, 0.85 }, { 0.00, 0.53, 0.85 },
    ・・・・
    { 0.00,-1.00, 0.00 }, { 0.00,-1.00, 0.00 }, { 0.00,-1.00, 0.00 } };
```

頂点配列を使用した描画（2）

- 頂点配列の機能を利用して描画

```
void  renderPyramid()
{
        glVertexPointer( 3, GL_FLOAT, 0, pyramid_full_vertices );
        glNormalPointer( GL_FLOAT, 0, pyramid_full_normals );

        glEnableClientState( GL_VERTEX_ARRAY );
        glEnableClientState( GL_NORMAL_ARRAY );

        glDrawArrays( GL_TRIANGLES, 0, num_full_vertices );
}
```

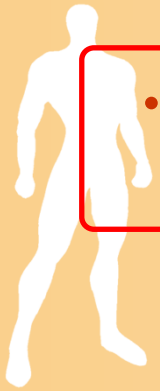
配列の先頭アドレスを
OpenGLに渡す

設定した配列を
有効化

設定した配列を使って
複数のポリゴンを描画

ポリゴンモデルの描画方法

- 方法1: glVertex() 関数に直接頂点座標を記述
  - 頂点データ（直接記述）、頂点ごとに渡す
- 方法2: 頂点データの配列を使用
  - 頂点データ、頂点ごとに渡す
- 方法3: 頂点データと面インデックスの配列を使用
  - 頂点データ＋面インデックス、頂点ごとに渡す
- 方法4: 頂点配列を使用
  - 頂点データ、OpenGLにまとめて渡す
- 方法5: 頂点配列と面インデックス配列を使用
  - 頂点データ＋面インデックス、 OpenGLにまとめて渡す



方法5 頂点配列＋インデックス配列

- 頂点配列に加えて三角面インデックスを指定し、OpenGLに一度にデータを渡して描画する方法
- 描画の手順
  - 基本的には、先ほどの頂点配列を使用する場合と同様に、配列データを設定する
  - 最後の描画時に、三角面インデックスを指定して描画
    - glDrawArrays()関数の代わりに glDrawElements()関数を使用

頂点配列を使用した描画（1）

- 配列データの定義（方法3と同じ）

```
const int  num_pyramid_vertices = 5;    // 頂点数
const int  num_pyramid_triangles = 6;   // 三角面数

// 角すいの頂点座標の配列
float  pyramid_vertices[ num_pyramid_vertices ][ 3 ] = {
          { 0.0, 1.0, 0.0 }, { 1.0,-0.8, 1.0 }, { 1.0,-0.8,-1.0 }, ・・・・・・
};

// 三角面インデックス（各三角面を構成する頂点の頂点番号）の配列
int  pyramid_tri_index[ num_pyramid_triangles ][ 3 ] = {
          { 0,3,1 }, { 0,2,4 }, { 0,1,2 }, { 0,4,3 }, { 1,3,2 }, { 4,2,3 }
};

// 三角面の法線ベクトルの配列（三角面を構成する頂点座標から計算）
float  pyramid_tri_normals[ num_pyramid_triangles ][ 3 ] = {
          { 0.00, 0.53, 0.85 }, // +Z方向の面
          ・・・・・・
```

頂点配列を使用した描画（2）

- 頂点配列の機能を利用して描画

```
void  renderPyramid()
{
        glVertexPointer( 3, GL_FLOAT, 0, pyramid_full_vertices );
        glNormalPointer( GL_FLOAT, 0, pyramid_full_normals );

        glEnableClientState( GL_VERTEX_ARRAY );
        glEnableClientState( GL_NORMAL_ARRAY );

        glDrawElements( GL_TRIANGLES, num_pyramid_triangles * 3,
                                GL_UNSIGNED_INT, pyramid_tri_index );
}
```

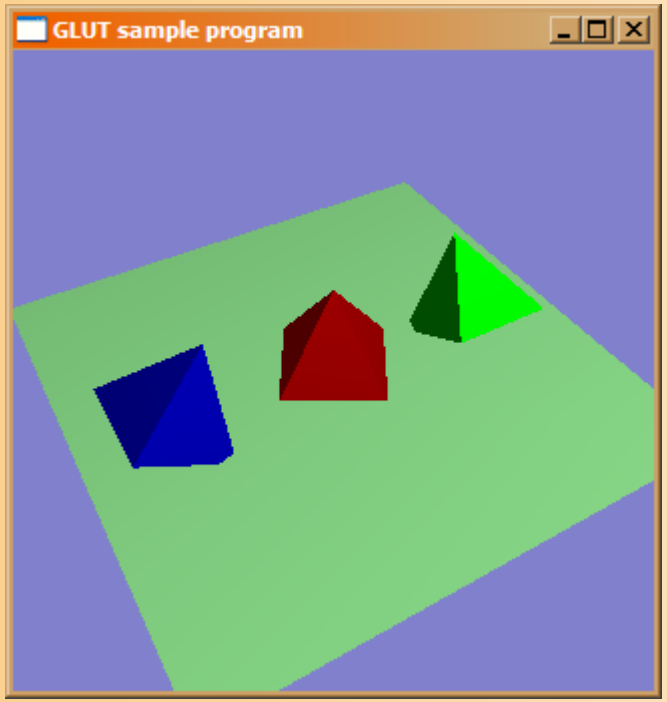

pyramid_tri_index で指定した頂点番号の配列に従って、頂点データを参照し、複数のポリゴンを描画

この方法の問題点

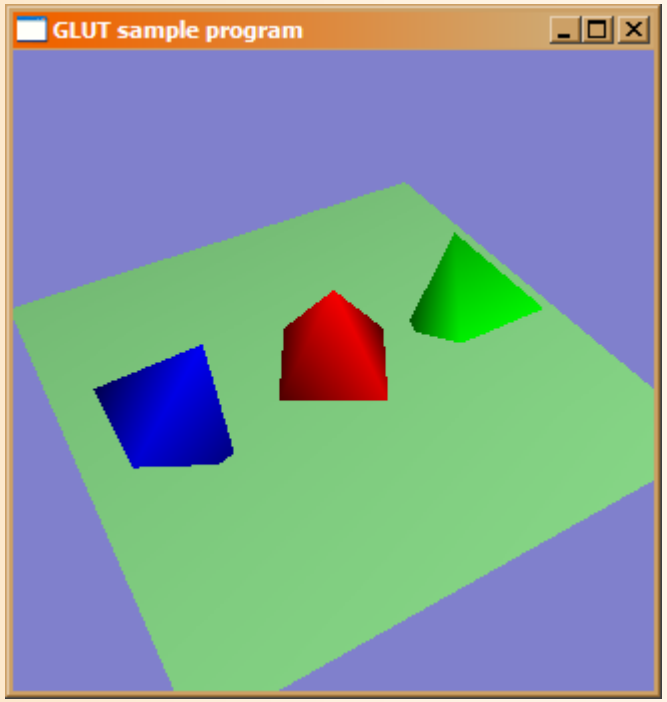
- 頂点の法線の問題
  - 隣接するポリゴンが共有する頂点を一つにまとめることができる
    - 頂点データの数を減らすことができる
  - ただし、頂点の座標だけでなく、法線やテクスチャ座標も一緒にする必要がある
    - 同じ頂点で、面ごとに法線を変えるようなことはできない
    - どうしても別の法線にしたければ、頂点データを分ける必要がある

２種類の法線による結果の例

同一頂点でも面ごとに
異なる法線を使用



同一頂点には全部の面で
同じ法線を使用



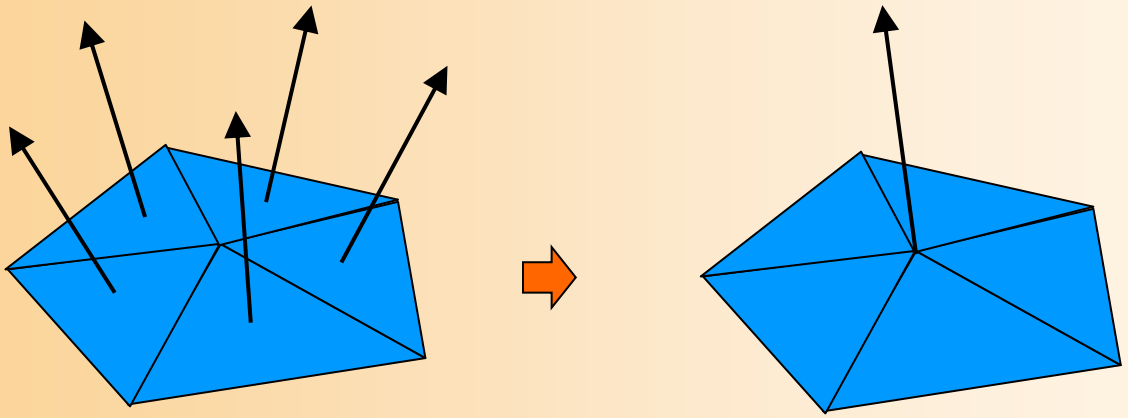
※ 面と面の境界がおなじ色になる

頂点の法線の使い分け

- 頂点を共有する面全部で共通の法線
  - 人体などの一般的な物体では、シェーディングによって表面をなめらかに見せるため、頂点を共有する面全部で頂点の法線を共通にするのが普通
  - 法線データは頂点の数だけ必要
- 面ごとに別々の法線
  - 今回の例の四角すいのように、角のある物体については、面ごとに法線を分けるのが自然
  - 法線データは面の数（×３）だけ必要になる
- 両者の混在は困難なのでどちらかに決めて適用

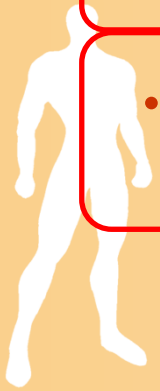
頂点の法線の計算方法

- 頂点の法線を自動的に計算する方法
  - 頂点の法線を全て零ベクトルにする
  - それぞれの面ごとに面の法線を計算
    - 面の法線を、面を構成する全頂点の法線に加算
  - 最後に、それぞれの頂点の法線を正規化する

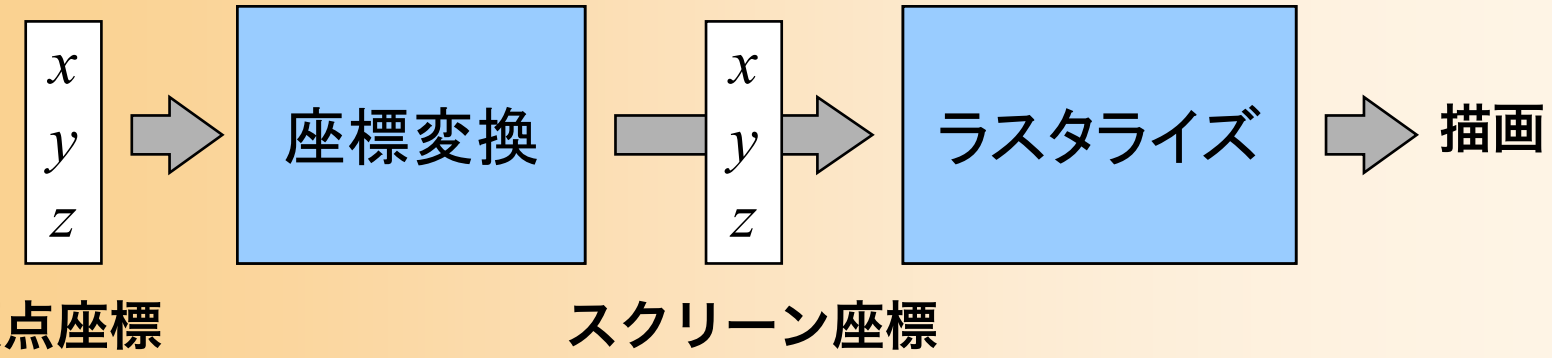


ポリゴンモデルの描画方法（まとめ）

- 方法1: glVertex() 関数に直接頂点座標を記述
  - 頂点データ（直接記述）、頂点ごとに渡す
- 方法2: 頂点データの配列を使用
  - 頂点データ、頂点ごとに渡す
- 方法3: 頂点データと面インデックスの配列を使用
  - 頂点データ＋面インデックス、頂点ごとに渡す
- 方法4: 頂点配列を使用
  - 頂点データ、OpenGLにまとめて渡す
- 方法5: 頂点配列と面インデックス配列を使用
  - 頂点データ＋面インデックス、 OpenGLにまとめて渡す



補足: レンダリングの高速化



- レンダリング時にボトルネックとなりうる処理
  - ラスタライズ
  - 頂点データの座標変換、頂点データの転送
  - 各描画関数の呼び出し
- 環境やプログラムによりボトルネックは異なる

レンダリングの高速化の問題

- 頂点データの座標変換・転送の問題
  - 同じ頂点は共有するなど、データ量を減らすことで高速化できる
- ラスタライズの問題
  - Ｚソートなどと併用することで、高速化できる
- 関数呼び出しのオーバーヘッドの問題
  - 頂点配列・インデックス配列を使うことで高速化

今日の内容

- 復習：ポリゴンモデルの描画
- 幾何形状データ
- ファイル形式
- データ構造の定義と描画処理
- ファイル読み込み処理の作成
  - Cによる読み込み処理の実装
  - C++による読み込み処理の実装
- 頂点配列の利用、高度な幾何形状データ処理

幾何形状データ

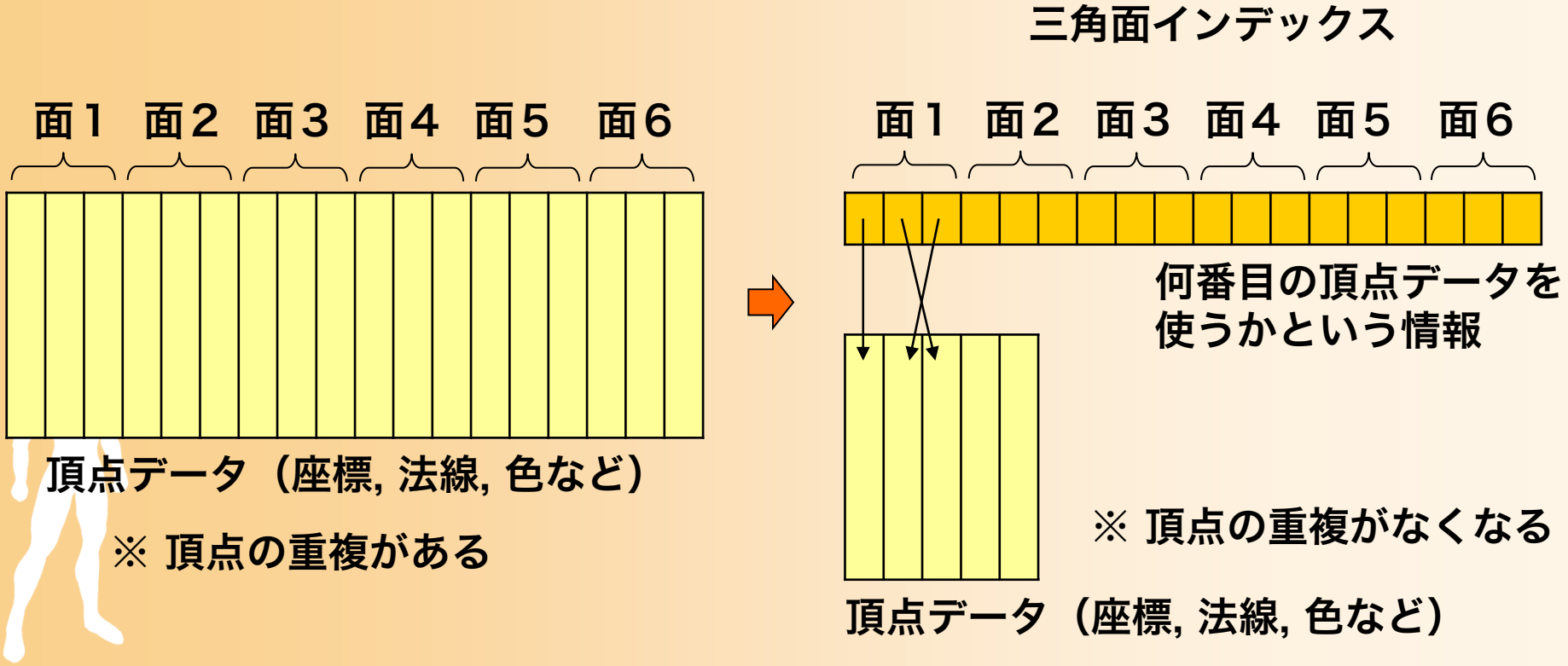
幾何形状データ

- 一般的なポリゴンモデルデータ
  - 頂点データ
    - 頂点座標、法線ベクトル、色、テクスチャ座標
  - 面（ポリゴン）データ
    - 面（ポリゴン）を構成する頂点の組
    - 面の向き（頂点の順序により表す）

※ 効率化のために、頂点データとポリゴンデータを分けて管理するのが一般的

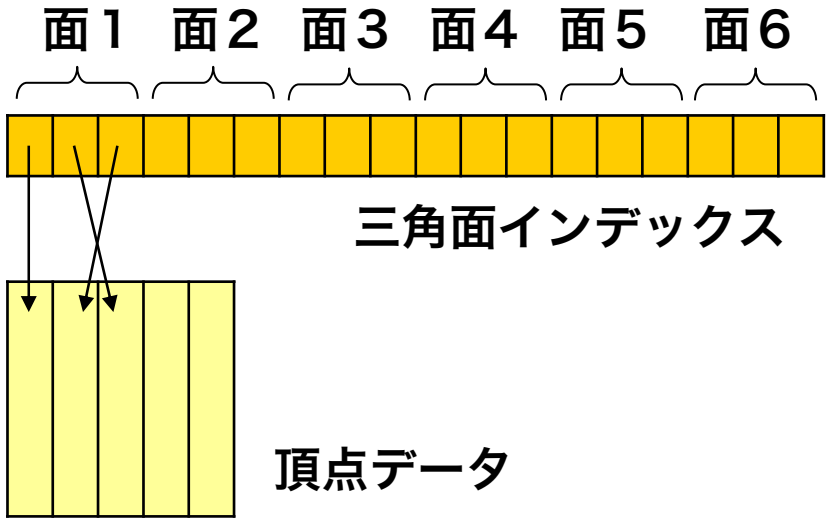
形状データの描画方法（復習）

- 頂点データの配列と、三角面インデックスの配列に分けて管理する



幾何形状データの表現例

```
// ベクトルデータ
struct  Vector
{
        float   x, y, z;
};

// 幾何形状データ
struct  Geometry
{
        int         num_vertices;   // 頂点数
        Vector *    vertices;       // 頂点座標配列
        Vector *    normals;        // 法線ベクトル配列
        Vector *    colors;         // カラー配列

        int         num_triangles;  // 三角面数
        int *       triangles;      // 三角面の頂点番号配列
};
```



幾何形状データ

- 頂点データ
  - 頂点座標（x, y, z）、法線ベクトル（nx, ny, nz）
  - 頂点カラー（r, g, b）、テクスチャ座標（u, v）
- ポリゴンの種類
  - 三角形のみ or 四角形のみ or 三角形と四角形 or 一般の多角形 （四角形・多角形は三角面に分割可能、ただしデータ量は増える）
- マテリアル情報
  - テクスチャ画像（ファイル名）、各種反射特性

ファイル形式

ファイル入出力機能作成のポイント

- 採用するファイル形式の決定
  - 既存のファイル形式 or 独自ファイル形式
- 幾何形状のデータ構造の決定
  - 頂点配列を使った描画に対応するかどうか
  - 対応するポリゴンの種類 （三角形のみ or 四角形のみ or 三角形と四角形 or 多角形）
- 読み込み処理の実装
  - 何らかの補助ライブラリを利用 or C/C++の標準関数のみを用いて実装

ファイル形式の種類

- アニメーションソフト（モデリングソフト）ごとに独自の形式がある
- 多くのアニメーションソフトは、他の一般的な形式でも インポート / エクスポート 可能
  - 各ソフトのデータをフルに保存するためには専用の形式を使う必要がある
  - 基本的な形状データであればどの形式でも大体表現可能
  - ただし、実際のデータ表現の仕方はファイル形式によってかなり異なる

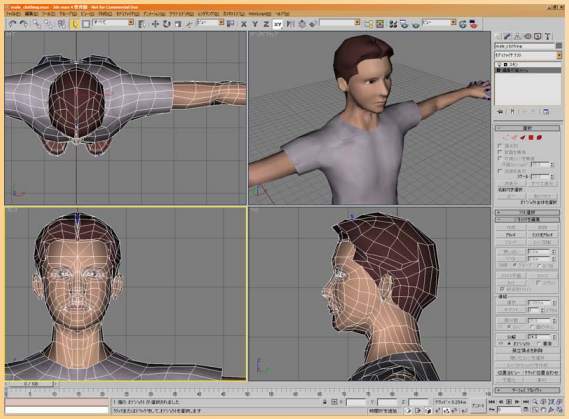
ファイル形式の選択

- 既存のファイル形式を利用
  - 読み込み関数を作成するだけで良い
  - ソフトがサポートしていない機能は使えない
  - ファイルサイズや読み込み速度などの点では、効率が悪くなる
- 独自のファイル形式を定義
  - 読み込み関数に加えて、アニメーションソフト用の書き出しプラグインを作成する必要がある
  - 同じく編集用プラグインを追加することで、特別な機能にも対応可能

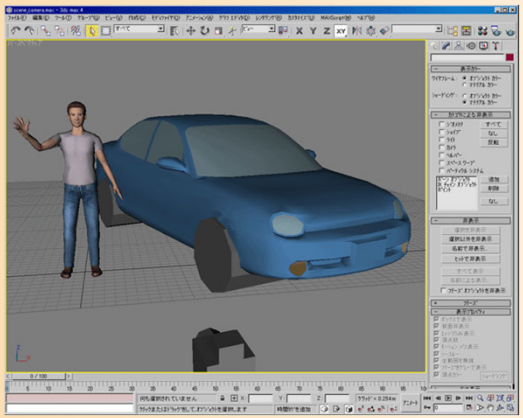
市販ソフトウェアの利用（復習）

- 既存ソフトウェアと組み合わせたプログラミング

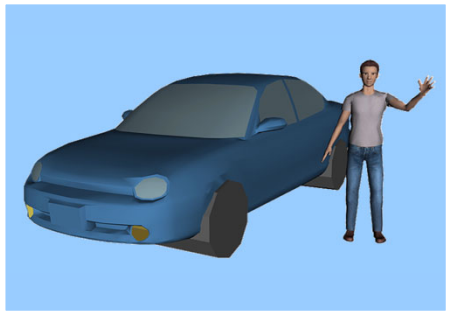
モデリング



レイアウト

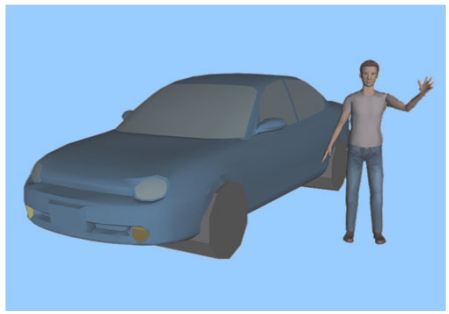


レンダリング



高品質な描画

高速な描画



形状データ

シーンデータ
動作データ

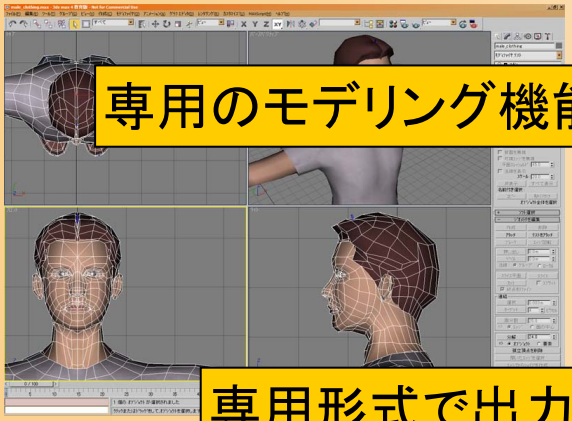
プログラム

ファイルからデータを読み込み、必要に応じて動きを生成しながら、リアルタイムにレンダリング

市販ソフトウェアの利用（復習）

・プラグインによる拡張が可能

モデリング

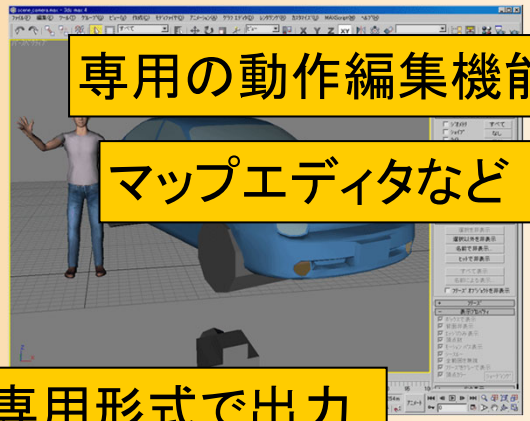


専用のモデリング機能

専用形式で出力

形状データ

レイアウト



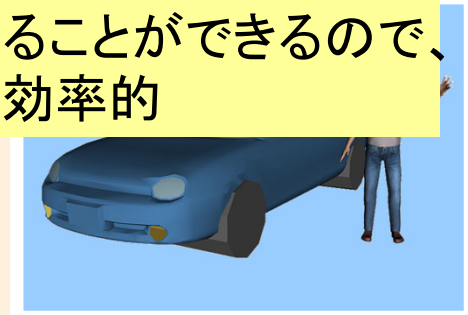
専用の動作編集機能

マップエディタなど

専用形式で出力

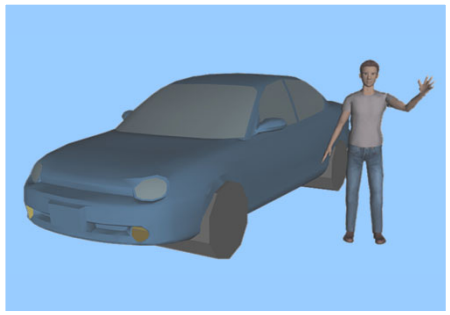
シーンデータ
動作データ

使い慣れたソフトウェアに、必要な機能だけを追加することができるので、効率的



高品質な描画

高速な描画



プログラム

映画制作・ゲーム制作などでは、各プロダクションごとに、独自のプラグインを多数使用

既存のファイル形式の読み込み

- ファイル形式の選択基準
  - 自分の使うソフトからエクスポートしやすいか
  - バイナリ形式 or アスキー形式（テキスト形式）
    - 一般にファイルフォーマットは公開されていないので、バイナリ形式は対応が困難
    - ファイルフォーマットが分かっていれば、バイナリ形式の方が読み込み処理の実現は容易な場合もある
  - 階層構造をサポートするかどうか
    - 一つの物体の形状だけではなく、多関節体・シーン情報・カメラ情報などを格納できるが、データ形式は複雑になる
  - 補助ライブラリやファイル形式の情報があるかどうか
- データ構造の定義、読み込み・描画処理の実装

よく使われるファイル形式（1）

- obj
  - Wavefront|Alias Maya、アスキー
- DXF
  - Autodesk AutoCAD、アスキー
- VRML（Virtual Reality Modeling Language）
  - アスキー、シーンの階層構造も表現可能
- max
  - 3ds max、バイナリ、シーンの階層構造も表現可能
- x
  - Direct X、バイナリ
  - Direct X を使えば 容易に読み込み・描画可能

よく使われるファイル形式（2）

- dea
  - COLLADA、アスキー
  - XMLベースのデータ交換用フォーマット
  - COLLDA ライブラリを使うことで読み書き可能
- fbx
  - Autodesk MotionBuilder（旧 Kaydara FilmBox）、バイナリ
  - データ交換用フォーマットとして広く利用されている
- glTF 2.0
  - アスキー＋バイナリ
  - 最近開発されている、モデルデータ＋シェーダの標準フォーマット
  - ドワンゴにより拡張された、人体モデル用の VRM 形式もある

ファイル読み込み処理の作成

- 本講義では、obj 形式を使用することにする
  - 元々は Alias|Wavefront Maya のデータ形式
  - 基本的な形状データのやり取りに使われる
  - テキスト形式、比較的単純な形式
  - 材質情報を表す mtl 形式と組で使われる
- Obj 形式のファイルの例
  - 四角すい （pyramid.obj）
  - 車 （car.obj）

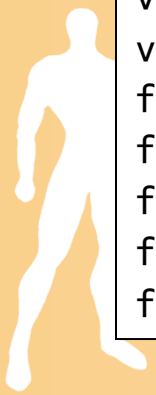
Obj形式のファイルの例

```
# Sample Obj Data (Pyramid)

mtllib pyramid.mtl
usemtl green
v    0.0    1.0    0.0
v    1.0   -0.8    1.0
v    1.0   -0.8   -1.0
v   -1.0   -0.8   -1.0
v   -1.0   -0.8    1.0
vn   0.9    0.4    0.0
vn   0.0    0.4   -0.9
vn  -0.9    0.4    0.0
vn   0.0    0.4    0.9
vn   0.0   -1.0    0.0
f   1//1   2//1   3//1
f   1//2   3//2   4//2
f   1//3   4//3   5//3
f   1//4   5//4   2//4
f   5//5   4//5   3//5   2//5
```

```
# Sample Obj Data (Pyramid)

newmtl green
Ka 0 0 0
Kd 0.3 0.8 0.3
Ks 0.9 0.9 0.9
Ns 20
```

四角すいの形状データファイル（pyramid.obj）（左）

四角すいの材質データファイル（pyramid.mtl）（上）



Obj形式（1）

- 詳しい定義はウェブなどにある情報を参照
  - 3D Format などのキーワードでウェブ検索すると、各種フォーマットの情報が見つかる （以下は例）
    - http://www.dcs.ed.ac.uk/home/mxr/gfx/3d-hi.html
  - 非公式な情報や古い情報も混じっている可能性があるので注意
- 以降、最低限必要な形式を説明

Obj形式（2）

- 形状データ
  - １行が１つのデータ（頂点・法線・面など）を表す
  - 各行の先頭の文字列（単語）によって、何のデータを表しているかが決まる
  - # で始まる行はコメント
  - v *x y z*　頂点座標
  - vn *nx ny nz*　法線ベクトル
  - vt *tx ty tz*　テクスチャ座標

Obj形式（3）

- 形状データ（続き）
  - f　*v1/t1/n1　v2/t2/n2　v3/t3/n3 …*
    ポリゴン（各頂点ごとの頂点座標番号/テクスチャ座標番号/法線ベクトル番号）
    - テクスチャ座標や法線ベクトルは空の場合もある
    - Obj形式では、各番号は１から始まるので注意

Obj形式（4）

- 形状データ（続き）
  - mtllib　*ファイル名*
    材質情報をファイルから読み込む
  - usemtl　*マテリアル名*
    後に続くポリゴンのマテリアルを指定する
  - g　*グループ名*
    後に続くポリゴンのグループ名を指定する
    （ポリゴンをグループ分けするときに使用）

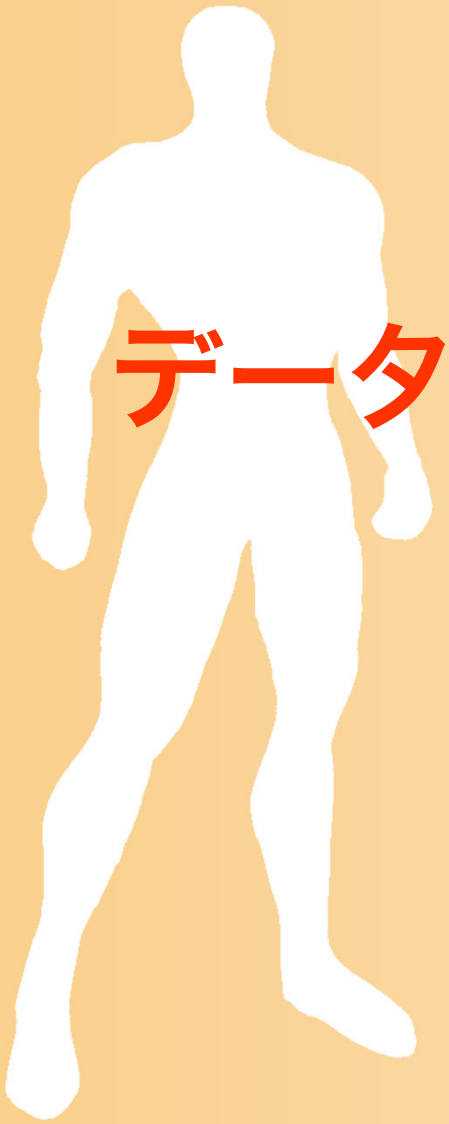
Obj形式（5）

- 材質データ（.mtl ファイル）
  - newmtl　マテリアル（材質）名
    新しい材質の定義を開始する
  - Ka *r g b*　環境光に対する反射特性
  - Kd *r g b*　拡散反射光に対する反射特性
  - Ks *r g b*　鏡面反射光に対する反射特性
  - Ns *s*　鏡面反射光の働く角度
  - map_Kd　反射光に使用するテクスチャ画像名

    - OpenGL の glColor 関数を呼ぶと、デフォルトでは、ka kd に相当するパラメタが変更される

今日の内容

- 復習：ポリゴンモデルの描画
- 幾何形状データ
- ファイル形式
- データ構造の定義と描画処理
- ファイル読み込み処理の作成
  - Cによる読み込み処理の実装
  - C++による読み込み処理の実装
- 頂点配列の利用、高度な幾何形状データ処理

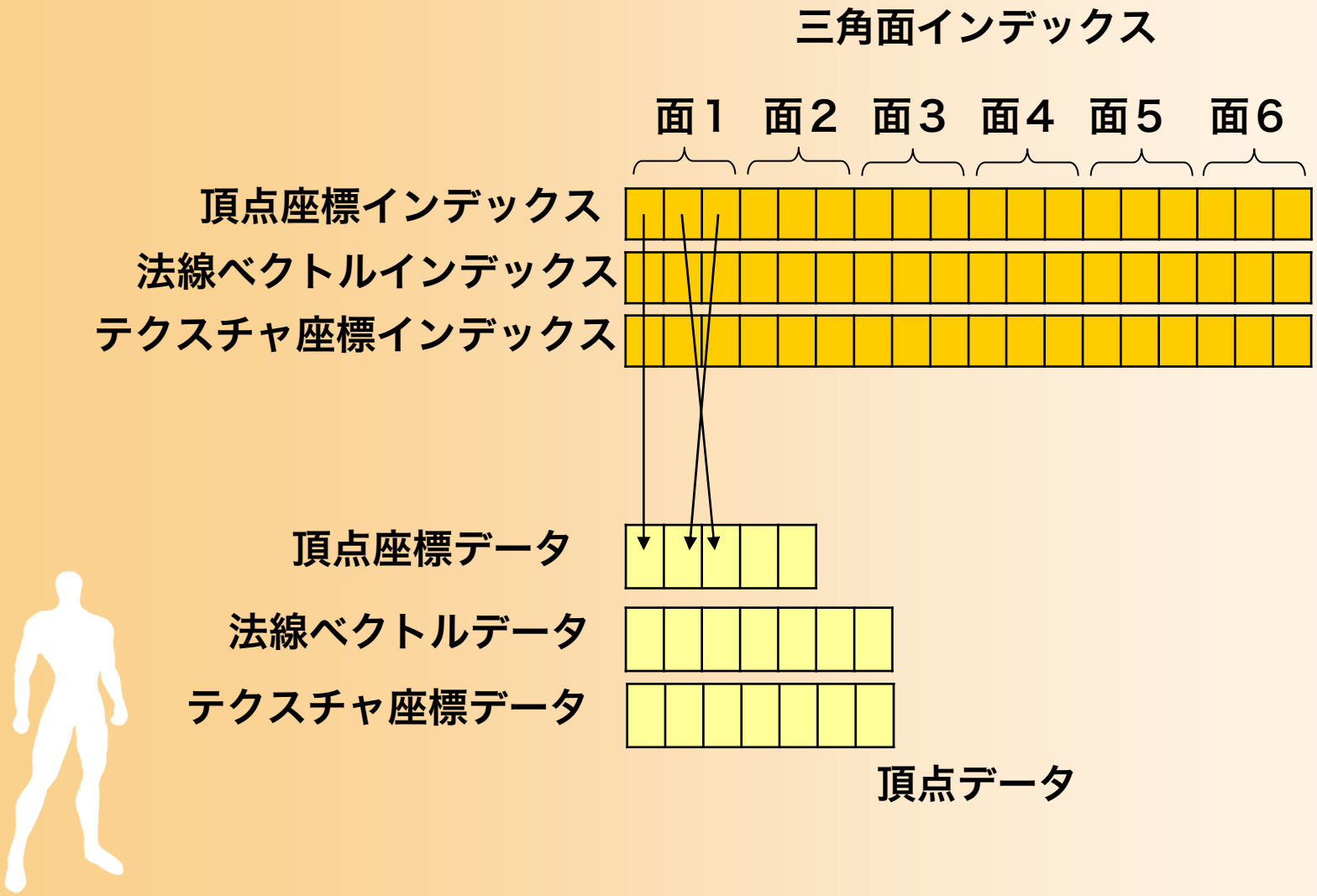


データ構造の定義と描画処理

データ構造の定義例

```
// 幾何形状データ(Obj形式用)
struct  Obj
{
        int         num_vertices;   // 頂点数
        Vector *    vertices;       // 頂点座標配列

        int         num_normals;    // 法線ベクトル数
        Vector *    normals;        // 法線ベクトル配列

        int         num_textures;   // テクスチャ座標数
        Vector *    textures;       // テクスチャ座標配列

        int         num_triangles;  // 三角面数
        int *       tri_v_no;       // 三角面の頂点座標番号の配列
        int *       tri_vn_no;      // 三角面の法線ベクトル番号の配列
        int *       tri_vt_no;      // 三角面のテクスチャ座標番号の配列
        Mtl *       tri_material;   // 三角面の素材の配列
};
```

データ構造の定義例


三角面インデックス
面１
面２
面３
面４
面５
面６
頂点座標インデックス
法線ベクトルインデックス
テクスチャ座標インデックス
頂点座標データ
法線ベクトルデータ
テクスチャ座標データ
頂点データ

データ構造の定義例

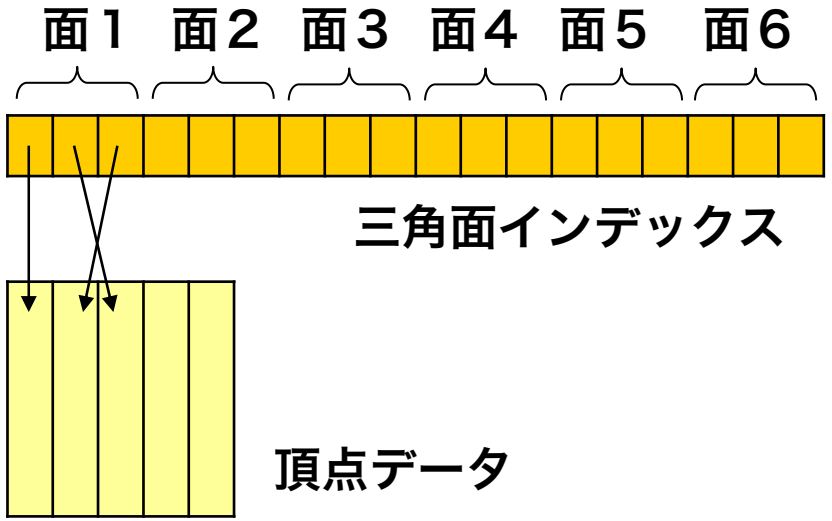
- Obj形式では任意のポリゴンを表現できるが、ここでは三角面のみを扱うことにする
  - 四角形以上の面が入力された場合は、複数の三角面に分割する
    - 一般に、n角形は、n-2個の三角計に分割できる
  - データ量は増えるが、全て三角面として処理できるので、データ構造や処理を単純にできる

頂点配列の利用

- 今回の定義例の問題点
  - そのままでは OpenGL の頂点配列の機能を利用できない
  - OpenGL の頂点配列を使うためには、各頂点ごとに頂点座標・法線ベクトル・テクスチャ座標をまとめる必要があるため
  - このように、既存のファイル形式の内部表現が自分の望む形式とは異なることはよくあるので、必要に応じて読み込み後に変換する

幾何形状データの表現例（比較用）

```
// ベクトルデータ
struct  Vector
{
        float   x, y, z;
};

// 幾何形状データ
struct  Geometry
{
        int         num_vertices;  // 頂点数
        Vector *    vertices;      // 頂点座標配列
        Vector *    normals;       // 法線ベクトル配列
        Vector *    colors;        // カラー配列
        int         num_triangles; // 三角面数
        int *       triangles;     // 三角面の頂点番号配列
};
```



頂点配列の利用

- 解決方法
  - 頂点配列を使うのをあきらめる
  - 頂点配列に適したデータ構造に変換する
    - 最初はObj形式のデータ構造で読み込み、後で変換
    - 最初から頂点配列に適したデータ構造で読み込み
- 本講義では
  - ひとまず、頂点配列を使わない単純なデータ構造を使用
  - 頂点配列を使用する場合に、どのような改良が必要になるかを説明

幾何形状データの描画

- 描画の手順
  - GL_TRIANGLES
  - 各面ごとに描画（num_triangles枚）
    - 材質が変更になっていれば、カラーの変更（glColor3f）などを実行
      - 毎回行うと無駄なので、前の面と異なる場合のみ実行
    - 面の各頂点データを OpenGL に渡す
      - tri_v_no, tri_vn_no, tri_vt_no からデータ番号を取得
      - glVector3f, glNormal3f などの関数を呼び出す

描画プログラム

```
void  RenderObj( Obj * obj )
{
    int  i, j, no;
    Mtl *  curr_mtl = NULL;

    glBegin( GL_TRIANGLES );
    for ( i=0; i<obj->num_triangles; i++ )
    {
        // マテリアルの切り替え
        if ( ( obj->num_materials > 0 ) && ( obj->tri_material[ i ] != curr_mtl ) )
        {
            curr_mtl = obj->tri_material[ i ];
            glColor3f( curr_mtl->kd.r, curr_mtl->kd.g, curr_mtl->kd.b );
        }
```

最後に描画した三角面の素材情報

各三角面ごとに繰り返し

現在の素材と異なる場合のみ設定

ここでは、環境光・拡散反射光の特性のみを変更

描画プログラム

```
            // 三角面の各頂点データの指定
            for ( j=0; j<3; j++ )
            {
                        // 法線ベクトル
                        no = obj->tri_vn_no[ i*3 + j ];
                        const Vector &  vn = obj->normals[ no ];
                        glNormal3f( vn.x, vn.y, vn.z );

                        // 頂点座標
                        no = obj->tri_v_no[ i*3 + j ];
                        const Vector &  v = obj->vertices[ no ];
                        glVertex3f( v.x, v.y, v.z );
            }
      }
      glEnd();
}
```

ファイル読み込み処理の作成

読み込み処理の作成

- C/C++ でのファイル読み込み
  - プログラミングが結構難しいので慣れが必要
- ファイル読み込みの方法
  - stdio を使用 （C標準関数、C++からも使える）
  - iostream を使用 （C++標準関数、>> 演算子）
- 読み込んだ文字列の解析の方法
  - 自分で文字列を解析 or 外部ライブラリを利用
  - scanf 系の関数を使用 （書式付き読み込み）
  - tokenizer を使用 （単語の切り分けの標準関数）

読み込み処理の作成（続き）

- 可変長のデータの読み込みの必要性
  - 頂点・ポリゴン数があらかじめ分かる場合
    - 最初に必要な大きさのデータ領域を確保すれば良いので、可変長データの扱いは必要ない
  - 最後まで読み込まないと分からない場合
    - 可変長データの扱いを考慮する必要がある
- 可変長のデータの扱い方
  - あらかじめかなり大きめの領域を確保しておき、領域が足りなくなったらあきらめる
    - 任意ファイルに対応できず、効率も悪いので良くない
  - データを読み込みながら領域を動的に広げる

一般的な読み込みの実現方法（1）

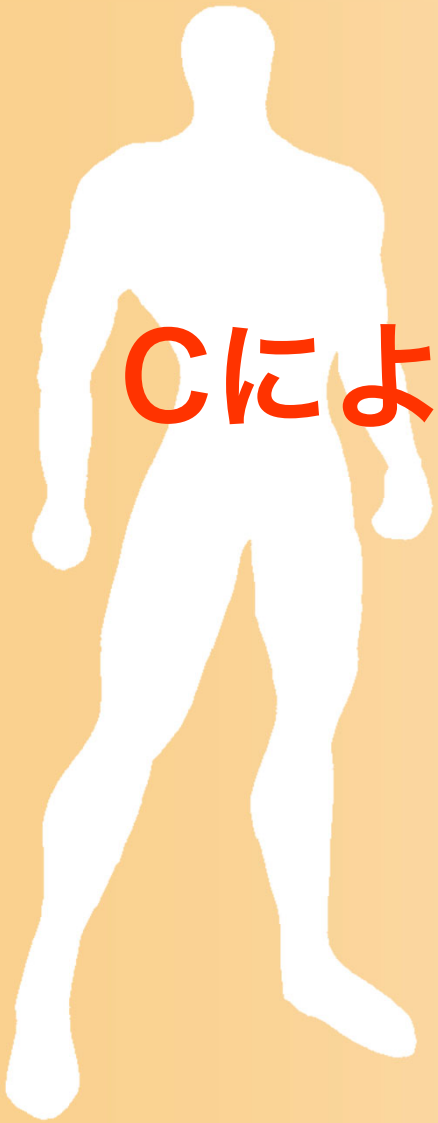
- 形状データに限らず、一般にデータ入出力は必須
- 自分で独自のファイル形式を定義
  - 機能を限定することで読み込みの行いやすい形式、または、効率の良い形式を定義できる
- yacc の利用
  - LR文法を使ってファイルの文法を定義すると、解析用のプログラムを出力してくれるコンパイラ・コンパイラ
  - メンテナンスが大変、現在は使われていない

一般的な読み込みの実現方法（2）

- XMLの利用
  - アスキー形式、タグを使ったデータを記述
  - 書き出し、読み込みのライブラリが多数存在
  - 階層構造を持った複雑なデータにも対応可能
- JSON の利用
  - アスキー形式、JavaScript オブジェクトのデータを記述
  - 多くのプログラミング言語から読み込める
- XMLやJSONを利用するためには、これらにもとづく独自形式を定義する必要がある

読み込み処理の例

- obj形式の読み込みのサンプルプログラム
  - 頂点の読み込み、一部のポリゴン・マテリアルの読み込みのみ
    → 完全版は各自作成してみること
- C/C++ での読み込み処理のサンプル
  - studio や iostream を使ったファイルの読み込み
  - tokenizer を使った文字列の解析
  - STL（Standard Template Library）を使った可変長データの扱い



Cによる読み込み処理の実装

読み込み処理のサンプル

- C での obj 形式の読み込みのサンプル
  - stdio を使ったファイルの読み込み
  - scanf系の標準関数を使った文字列の解析
  - 可変長のデータの読み込みへの対応
    - あらかじめ十分大きな配列を確保しておくことで対応
    - 確保済みの配列の大きさを超えるサイズには未対応
  - C/C++ 標準関数の使い方については、この講義では説明しないので、各自調べること
    - Visual Studio のヘルプでかなり詳しい説明がある

デモプログラム

- 幾何形状データの読み込みと描画
  - 幾何形状データ（obj形式ファイル）を読み込んで描画
  - マウスドラッグによる視点操作も可能（前回の講義で扱った内容）

サンプルプログラム

- obj_viewer.cpp, obj.h, obj.cpp
  - デモプログラムのもとになるプログラム
  - objファイルの読み込みや描画は実装済み
- WavefrontObj.h, WavefrontObj.cpp
  - 参考用 C++ のサンプルプログラム、後程説明
- プログラムの解説は、講義のウェブページの演習資料を参照
- 開発環境は、第1回の講義の説明や、講義のウェブページの資料を参照

サンプルプログラムの構成

- obj.h
  - obj形式の幾何形状データを表す構造体や読み込み・描画関数の定義
- obj.cpp
  - obj形式の幾何形状データの読み込み・描画関数の実装
- obj_viewer.cpp
  - 上記の構造体・関数を使用するメインプログラム
  - GLUT + OpenGL を使用

サンプルプログラム解説（1）

- obj.cpp
  - 28～181行が、読み込み処理を行う関数
    - ファイル名を引数として受け取り、読み込んだオブジェクトのポインタを返す
  - 45～58行で、オブジェクトを初期化し、読み込み結果の格納に使うデータ領域を大きめに確保している
  - 40行で、ファイルを開く
    - アスキー形式、読み込みモード
  - 61行以降、ファイルから1行ずつ読み込みながら、処理

サンプルプログラム解説（2）

```
// バッファ長（サイズは適当）
#define  BUFFER_LENGTH      1024
#define  MAX_VECTOR_SIZE    4096
#define  MAX_TRIANGLE_SIZE  4096
#define  MAX_MTL_SIZE       32

//  Objファイルの読み込み
Obj *  LoadObj( const char * filename )
{
        FILE *  fp;
        char    line[ BUFFER_LENGTH ];
        char    name[ BUFFER_LENGTH ];
        int  i, j;
        Vector  vec;
        int  v_no[4], vt_no[4], vn_no[4];
        int  count;
        Mtl *  curr_mtl = NULL;
```

領域を確保する頂点数・三角面数を
あらかじめ定数として指定

ファイル名を引数として受け取り、
読み込んだオブジェクトを返す

サンプルプログラム解説（3）

```
// ファイルを開く
fp = fopen( filename, "r" );
if ( fp == NULL )    return  NULL;

// Obj構造体を初期化（ひとまず固定サイズの配列を割り当てる）
Obj *  obj = new Obj();
obj->num_vertices = 0;
obj->num_normals = 0;
obj->num_tex_coords = 0;
obj->vertices = new Vector[ MAX_VECTOR_SIZE ];
obj->normals = new Vector[ MAX_VECTOR_SIZE ];
obj->tex_coords = new Vector[ MAX_VECTOR_SIZE ];
obj->num_triangles = 0;
obj->tri_v_no = new int[ MAX_TRIANGLE_SIZE * 3 ];
obj->tri_vn_no = new int[ MAX_TRIANGLE_SIZE * 3 ];
obj->tri_vt_no = new int[ MAX_TRIANGLE_SIZE * 3 ];
obj->tri_material = new Mtl*[ MAX_TRIANGLE_SIZE * 3 ];
obj->num_materials = 0;
obj->materials = NULL;
```

アスキー形式、読み込みモード
以降、fp を使ってファイルにアクセス

頂点に関する配列を確保
（頂点座標、法線ベクトル、テクスチャ座標）

三角面に関する配列を確保
（頂点番号、法線ベクトル番号、
テクスチャ座標番号、素材番号）

サンプルプログラム解説（4）

- obj.cpp
  - 64行以降の if 文では、strncmp（）関数を使って、読み込んだ行の先頭が、Obj形式の各コマンドかどうかを判定
    - 頂点データ（92行以降）に関しては、１文字目と２文字目を見て判定
  - 頂点データやポリゴンデータの解析は、sscanf（）関数を使用
    - 今回のプログラムでは、以下の形式のみ対応
      f　*v0//n0　v1//n1　v2//n2*
      （*v0~v1*には頂点番号、*n0~n2*には法線番号が入る）

サンプルプログラム解説（5）

```
// ファイルから1行ずつ読み込み
while ( fgets( line, BUFFER_LENGTH, fp ) != NULL )
{
    // マテリアルの読み込み
    if ( strncmp( line, "mtllib", 6 ) == 0 )
    {
        // テキストを解析
        sscanf( line, "mtllib %s", name );

        // 指定されたファイル名のマテリアルデータを読み込み
        if ( strlen( name ) > 0 )
            LoadMtl( name, obj );
    }

    // マテリアルの変更
    if ( strncmp( line, "usemtl", 6 ) == 0 )
    {
        .......
```

ファイルから一行読み込み、line に格納
ファイルの末端まで到達したら終了

行の先頭の6文字が「mtllib」であれば、対応した解析処理を実行
以下、obj形式の各コマンドごとに、同様の判定と処理を行う

sscanf関数を使って解析

材質ファイル（mtl形式）の読み込みは、別に作成する LoadMtl 関数を呼び出し。
LoadMtl関数の実装方法は、基本的にLoadObje関数と同様なので、説明は省略

サンプルプログラム解説（6）

```
// 頂点データの読み込み
if ( line[0] == 'v' )
{
    // 法線ベクトル（vn）
    if ( line[1] == 'n' )
    {
        // テキストを解析
        sscanf( line, "vn %f %f %f", &vec.x, &vec.y, &vec.z );

        // 法線ベクトル配列の末尾に格納
        obj->normals[ obj->num_normals ] = vec;
        obj->num_normals ++;
    }
    // テクスチャ座標（vt）
    else if ( line[1] == 't' )
    {
        .......
```

ここでは vn, vt, v をまとめて処理

sscanf関数を使って解析
（入力がフォーマットに従っていると仮定）

読み込んだデータを配列に格納

サンプルプログラム解説（7）

```
// ポリゴンデータの読み込み
if ( line[0] == 'f' )
{
    // テキストを解析（三角形・テクスチャ座標なしの場合）
    count = sscanf( line, "f %i//%i %i//%i %i//%i", &v_no[0], &vn_no[0],
                    &v_no[1], &vn_no[1], &v_no[2], &vn_no[2] );
    // 解析に成功したらポリゴンデータを記録
    if ( count == 6 )
    {
        i = obj->num_triangles * 3;
        for ( j=0; j<3; j++ )
        {
            obj->tri_v_no[ i+j ] = v_no[ j ] - 1;
            obj->tri_vn_no[ i+j ] = vn_no[ j ] - 1;
            obj->tri_vt_no[ i+j ] = vt_no[ j ] - 1;
        }
        obj->tri_material[ obj->num_triangles ] = curr_mtl;
        obj->num_triangles ++;
    }
```

sscanf関数を使って解析
（入力が想定したフォーマットに従っていると仮定）

今回の三角面の先頭インデックス番号

サンプルプログラム解説（8）

- obj.cpp
  - 170行以降、読み込んだデータサイズに合わせて、新しい配列を確保して、読み込んだデータをそちらにコピーし、読み込みに使用した大きな配列は解放
    - 全配列に対して、同様の処理を行う

サンプルプログラム解説（9）

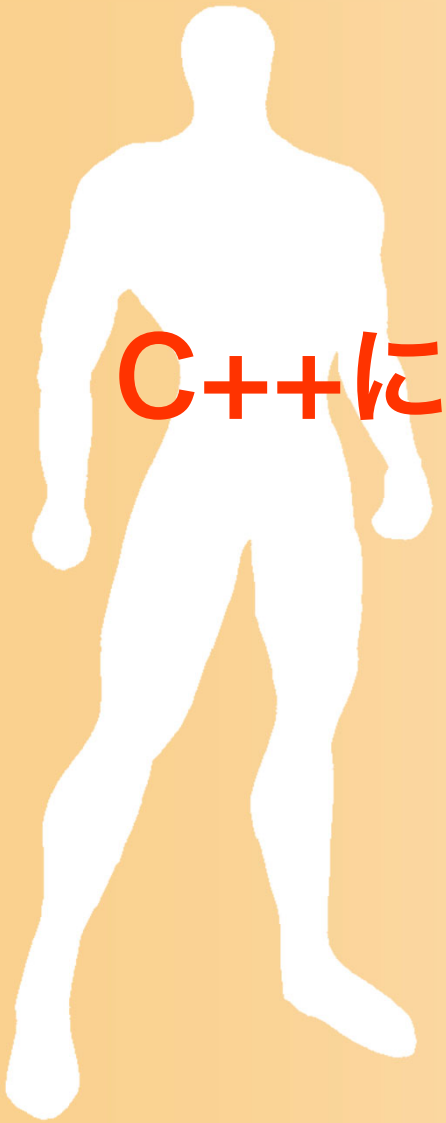
```
    };
    // ここまでで、ファイルの全行の解析が終了

    // 必要な配列を確保しなおす
    Vector *  new_array;
    new_array = new Vector[ obj->num_vertices ];
    memcpy( new_array, obj->vertices, sizeof( Vector ) * obj->num_vertices );
    delete[]  obj->vertices;
    obj->vertices = new_array;

    .......

    // ファイルを閉じる
    fclose( fp );

    // 読み込んだオブジェクトデータを返す
    return  obj;
}
```

サンプルプログラム解説（10）

- メインプログラム（obj_view.cpp）
  - 基本的に、前回の視点操作のプログラムと同じ
  - 形状データを格納する変数を追加
  - キーボードが押されたときの処理に、ファイル読み込みの処理を追加
    - キーが押されたときに呼ばれるGLUTのコールバック関数を追加
    - ファイル選択ダイアログの表示は、Windows API を使用
  - 描画処理で、読み込んだ形状データを描画

サンプルプログラムの問題点

- scanf 系関数では、ファイル形式のゆらぎに対応できない
  - 特にポリゴンデータの解析が苦しい
- 可変長データの読み込みにきちんと対応していない
  - 読み込みの途中で領域が不足したら動的に領域を拡張するような処理を追加すると、かなり複雑になってしまう



C++による読み込み処理の実装

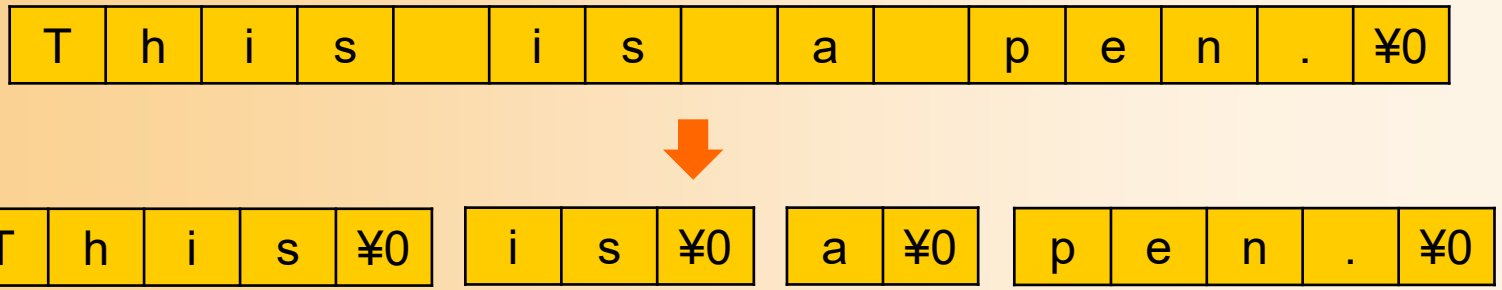
読み込み処理の改良

- C++ での obj 形式の読み込みのサンプル
  - ファイルの読み込み
    - iostream を使った方法
  - 文字列の解析
    - tokenizer を使う方法
  - 可変長データの読み込み
    - STL（Standard Template Library）を使う方法
- 必ずしも C++ を使う必要はない
  - C でも同じような工夫は可能
- 外部ライブラリを利用する方法もある

Tokenizer による文字列解析

- strtok 関数
  - 文字列切り出し関数
  - 1回目　tokenizer(文字列, 区切り文字集合);
    2回目以降　tokenizer(NULL, 区切り文字集合);
  - もとの文字列を破壊するので注意

例 :



STL

- STL（Standard Template Library）
  - 可変長配列（vector）、リスト（list）、２分木による集合（set）やインデックス（map）などのコンテナを扱う標準ライブラリ
  - テンプレートライブラリ
    - vector< float > array; のように、格納するデータの型を指定して変数を定義でき、任意の型に対応できる
    - 可変長配列（vector）は、array[i] などのように、配列と同様に利用できる（演算子オーバーロード）
    - メモリ管理（領域の確保・解放）も自動的に行われる

STLの使用方法（1）

- 可変長配列　vector< 型 >
  - [番号] により、通常の配列と同様に要素を参照
  - size() により、配列のサイズを取得
  - resize(), assign() などにより、配列のサイズの変更や初期化
  - push_back() で、最後尾に要素を追加
- 文字列型 string
  - vector< char > を使って実現
  - c_str() で、const char * 型の文字列を取得

STLの使用方法（2）

- STL では、要素に順番にアクセスするために、反復子（iterator）を使用する
  - 反復子（iterator）を使うことで、コンテナの種類に関わらず、共通の処理を実装できる
  - 反復子（iterator）に対して、ポインタと同様の操作ができる
    - * 演算子で、反復子（iterator）が指す値を取得
    - ++ 演算子で、次の要素に進む


```
list< int >  data;
list< int >::iterator   it = data.front()
while ( it != data.end() ) {
    printf( “%d “, *it );   it ++;  }
```

STLの使用方法（2）

- STL では、要素に順番にアクセスするために、反復子（iterator）を使用する
  - 反復子（iterator）を使うことで、コンテナの種類に関わらず、共通の処理を実装できる
  - 反復子（iterator）に対して、ポインタと同様の操作ができる
    - * 演算子で、反復子（iterator）が指す値を取得
    - ++ 演算子で、次の要素に進む


```
list< int >  data;
for ( list< int >::iterator  it = data.front();  it != data.end();  i++ )
    printf( “%d “, *it );
```

STLの使用方法（3）

- インデックス　map，multimap
  - 以下のように、キー（この場合は文字列）を使って、キーに対応するデータ（この場合は Object *）を格納・検索できる

```
map< string, Object * >  index;
Object *  object = NULL;

// 追加
index[ “car” ] = object;

// 検索
map< string, Object * >::iterator   i = index.find( “car” );
if ( i != index.end() )
    object = (*i).second
```

STLの使用方法（3）

- インデックス　map，multimap
  - 以下のように、キー（この場合は文字列）を使って、キーに対応するデータ（この場合は Object *）を格納・検索できる

```
map< string, Object * >  index;
Object *  object = NULL;

// 追加
map< string, Object * >::value_type  data( “car”, object );
index.insert( data );

// 検索
map< string, Object * >::iterator   i = index.find( “car” );
if ( i != index.end() )
    object = (*i).second
```

STLの使用方法（4）

- インデックス　map，multimap
  - 以下のように、キーの範囲内の値を全て取得することもできる

```
multimap< int, Object * >  index;

// 範囲検索（キーの値が10以上、30以下の全データを探索）
multimap< int, Object * >::iterator   l = index.lower_bound( 10 );
multimap< int, Object * >::iterator   u = index.upper_bound( 30 );
for (map< int, Object * >::iterator  l = l; i != u; i++ )
    object = (*i).second;
```


  - ソートに用いる比較関数を指定できる
  - multimapは、一つのキーに対して、複数のデータを格納できる

サンプルプログラム解説（1）

- WavefrontObj.h
  - データ構造（WavefrontObj クラス）を定義
    - 頂点・ポリゴンデータには STL の可変長配列を使用
- WavefrontObj.cpp
  - 18行以降が、読み込み処理を行うコンストラクタ
    - 読み込んだ結果はメンバ変数に格納
  - 29行で、ファイルを開く
    - iostreamを使用、アスキー形式での読み込みモード
  - 33行以降、ファイルから1行ずつ読み込みながら、処理

サンプルプログラム解説（2）

```
WavefrontObj::WavefrontObj( const char * file_name )
{
    ifstream  file;
    char      line[ BUFFER_LENGTH ];
    char *    token;    char *    data;
    Group *     curr_group = NULL;    Material *  curr_mtl = NULL;
    vector< int >   face_data;

    // ファイルのオープン
    file.open( file_name, ios::in );
    if ( file.is_open() == 0 )  return; // ファイルが開けなかったら終了

    // ファイルを先頭から1行ずつ順に読み込み
    while ( ! file.eof() )
    {
        // 1行読み込み、先頭の単語を取得
        file.getline( line, BUFFER_LENGTH );
        token = strtok( line, " " );
```

コンストラクタ
指定されたファイルから幾何形状データを読み込み、初期化

読み込んだ単語（解析する単語）のアドレスを格納する変数

ファイルの末端に到達するまで繰り返し

最初の呼び出しなので、引数には行の先頭のアドレスを指定

サンプルプログラム解説（3）

- WavefrontObj.cpp
  - 37行で、tokenizer を使用し、読み込んだ行の先頭の単語（最初の空白までの文字列）を取得
  - 44行以降では、取得した単語に応じて処理
  - 頂点データやポリゴンデータの解析は、引き続き tokenizer を使用し、以降の数字を取得
    - 区切り文字として、空白や / を使用（65行など）
    - ポリゴンデータは、テクスチャ頂点番号が省略されることがあるので、特別な判定を追加（68行）
  - 読み込んだデータは、STLの可変長配列（vector）に順次格納（push_back（）関数）

サンプルプログラム解説（4）

```
// 点データの読み込み
if ( *token == 'v' )
{
        Vertex  v;
        data = strtok( NULL, " " );  v.x = data ? atof( data ) : 0.0;
        data = strtok( NULL, " " );  v.y = data ? atof( data ) : 0.0;
        data = strtok( NULL, " " );  v.z = data ? atof( data ) : 0.0;

        token ++;
        if ( *token == 't' )
                t_coords.push_back( v );
        else if ( *token == 'n' )
                normals.push_back( v );
        else
                vertices.push_back( v );
        continue;
}
.........
```

二つ目以降の単語なので、引数にはNULLを指定
（前回指定した文字列が引き続き解析される）

dataがNULLでなければ、文字列を数値に変換して記録

配列にデータを追加
（もとのタグの2文字目に応じて、
どの配列に追加するかを決定）

サンプルプログラム解説（5）

```
// 面データの読み込み
if ( *token == 'f' )
{
    face_data.clear();
    while ( (data = strtok( NULL, " /" )) != NULL )
    {
        // テクスチャ番号が省略された時('/' が続いた時)は -1 を設定
        if ( *(data - 1) == '/' )
            face_data.push_back( -1 );

        // Obj形式では頂点番号は1から始まるので -1して0からの番号に変換
        face_data.push_back( atoi( data ) - 1 );
    }
    Face *  face = new Face();
    face->group = curr_group;     face->material = curr_mtl;
    face->data = face_data;
    faces.push_back( face );
    ........
```

二つ目以降の単語なので、引数にはNULLを指定
（前回指定した文字列が引き続き解析される）

区切り（最後のスペースor/）の前も
/ であれば、/ が続いていると判定

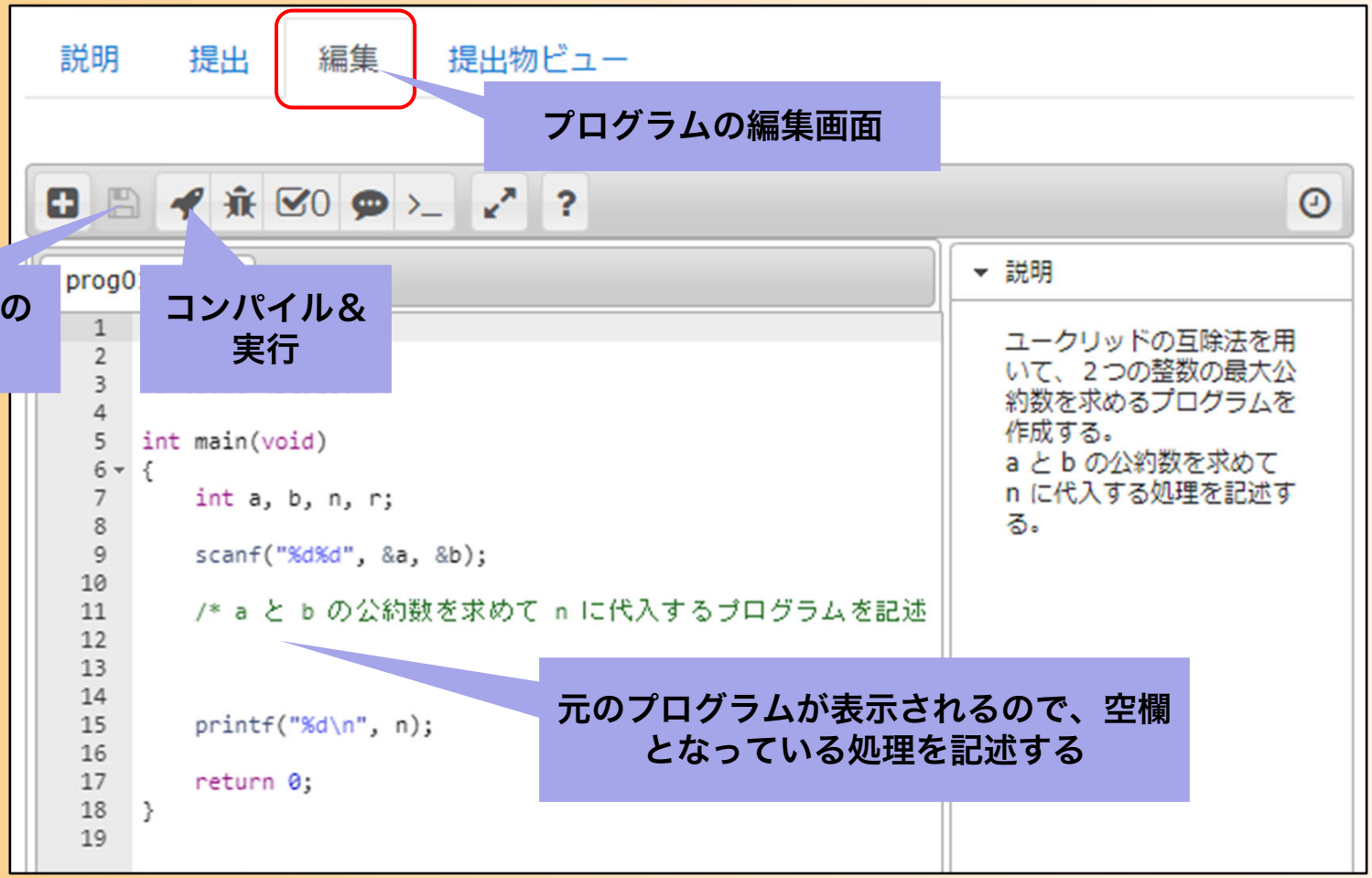
配列にデータを追加

演習問題

- 今回は、Moodle のプログラミング演習問題も受験する
  - VPL (Virtual Programming Lab) の機能を利用
  - 指定された処理を実現するように、与えられたプログラムの空欄を記述する
  - VPL 上で、コンパイル・実行、評価を行う
  - 評価まで行い、正しく実行されることを確認

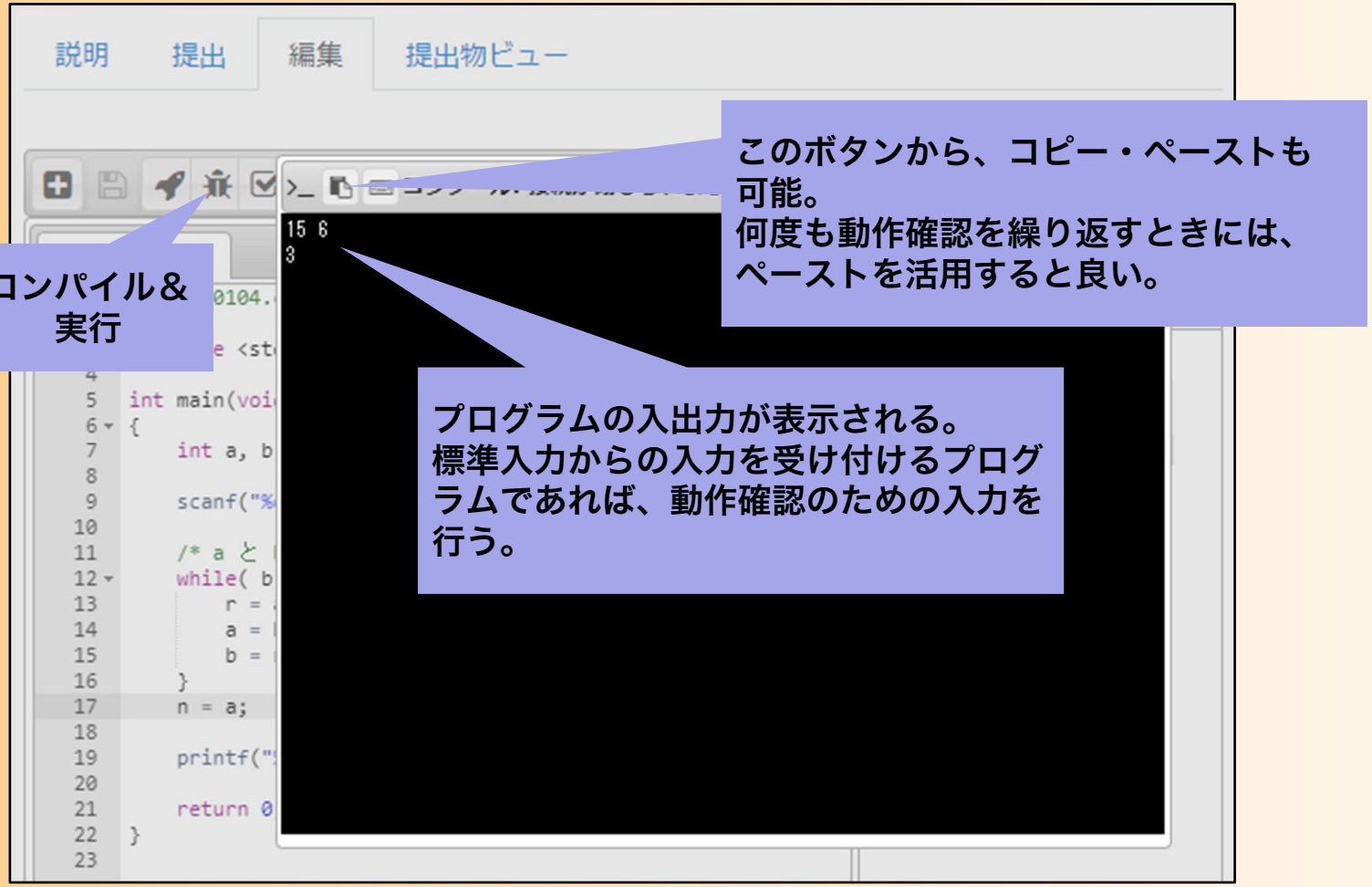
演習手順（1）

- VPL でのプログラムの編集



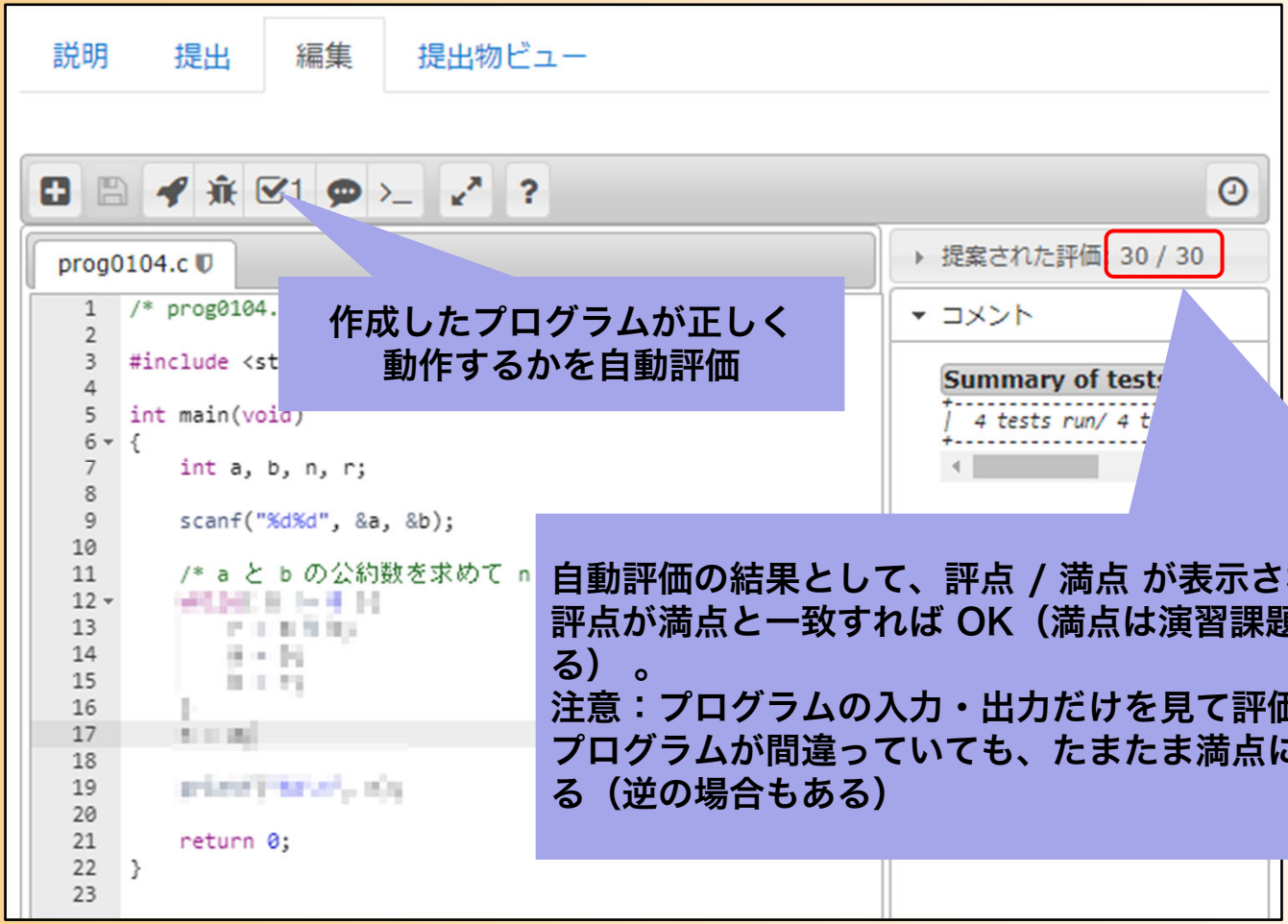
演習手順（3）

- VPL でのプログラムの実行



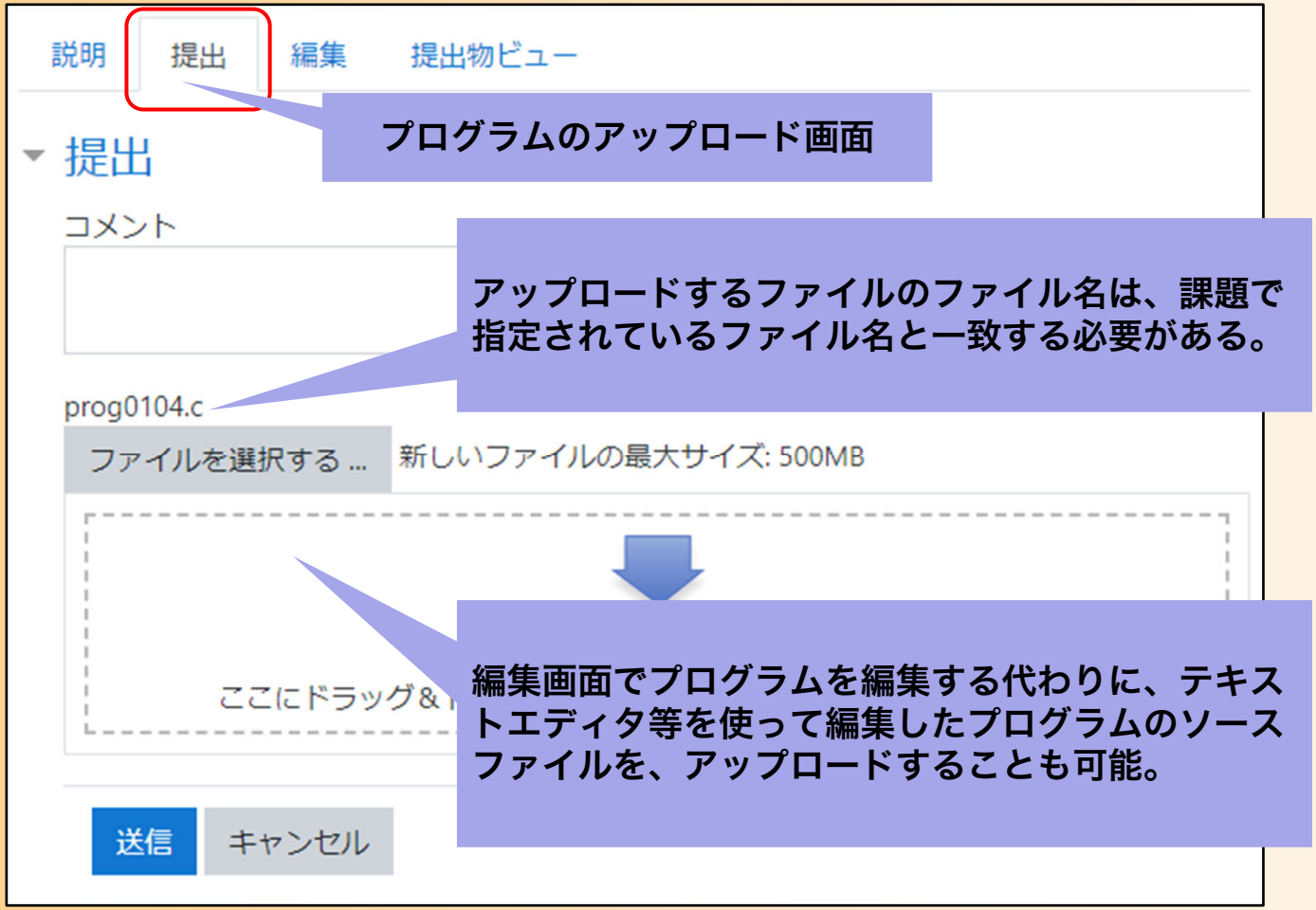
演習手順（4）

- VPL でのプログラムの評価



演習手順（5）

- 参考：VPL でのプログラムのアップロード



今日の内容

- 復習：ポリゴンモデルの描画
- 幾何形状データ
- ファイル形式
- データ構造の定義と描画処理
- ファイル読み込み処理の作成
  - Cによる読み込み処理の実装
  - C++による読み込み処理の実装
- 頂点配列の利用、高度な幾何形状データ処理

頂点配列の利用

頂点配列を使った描画（復習）

- 頂点配列
  - 配列データを一度に全部 OpenGL に渡して描画を行う機能
  - 頂点ごとに OpenGL の関数を呼び出して、個別にデータを渡す必要がなくなる
    - 処理を高速化できる
    - 渡すデータの量は同じでも、頂点配列を利用することで、処理を高速化できる
  - 実用では頂点配列の機能を使うのが一般的
    - 携帯端末用の OpenGL ES では、glBegin・glEnd関数は使えないため、頂点配列の使用が必須となる

頂点配列を使った描画方法（復習）

- 頂点配列を使った描画の手順
  1. 頂点の座標・法線などの配列データを用意
  2. OpenGLに配列データを指定（配列の先頭アドレス）
     - glVertexPointer（）関数、glNormalPointer（）関数、等
  3. どの配列データを使用するかを設定
     - 頂点の座標、色、法線ベクトル、テクスチャ座標など
     - glEnableClientState（）関数
  4. 配列の使用する範囲を指定して一気に描画
     - 配列データをレンダリング・パイプラインに転送
     - glDrawArrays（）関数

頂点配列を使用した描画（1）

- 配列データの定義

```
const int  num_pyramid_vertices = 5;    // 頂点数
const int  num_pyramid_triangles = 6;   // 三角面数

// 角すいの頂点座標の配列
float  pyramid_vertices[ num_pyramid_vertices ][ 3 ] = {
          { 0.0, 1.0, 0.0 }, { 1.0,-0.8, 1.0 }, { 1.0,-0.8,-1.0 }, ・・・・・・
};

// 三角面インデックス（各三角面を構成する頂点の頂点番号）の配列
int  pyramid_tri_index[ num_pyramid_triangles ][ 3 ] = {
          { 0,3,1 }, { 0,2,4 }, { 0,1,2 }, { 0,4,3 }, { 1,3,2 }, { 4,2,3 }
};

// 三角面の法線ベクトルの配列（三角面を構成する頂点座標から計算）
float  pyramid_tri_normals[ num_pyramid_triangles ][ 3 ] = {
          { 0.00, 0.53, 0.85 }, // +Z方向の面
          ・・・・・・
```

頂点配列を使用した描画（2）

- 頂点配列の機能を利用して描画

```
void  renderPyramid()
{
        glVertexPointer( 3, GL_FLOAT, 0, pyramid_full_vertices );
        glNormalPointer( GL_FLOAT, 0, pyramid_full_normals );

        glEnableClientState( GL_VERTEX_ARRAY );
        glEnableClientState( GL_NORMAL_ARRAY );

        glDrawElements( GL_TRIANGLES, num_pyramid_triangles * 3,
                                GL_UNSIGNED_INT, pyramid_tri_index );
}
```

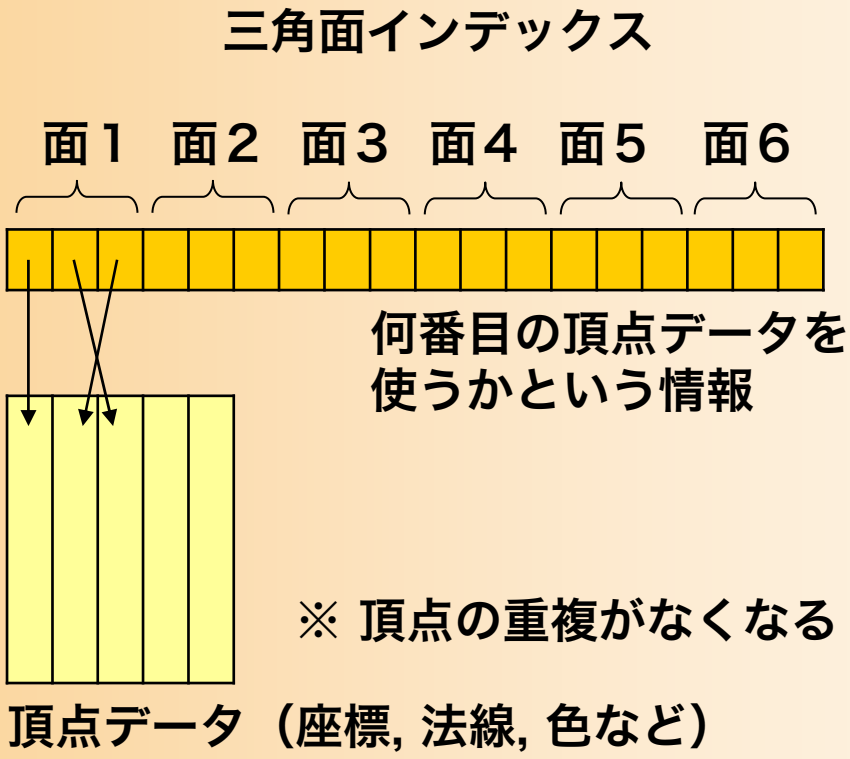

pyramid_tri_index で指定した頂点番号の配列に従って、頂点データを参照し、複数のポリゴンを描画

頂点配列の利用

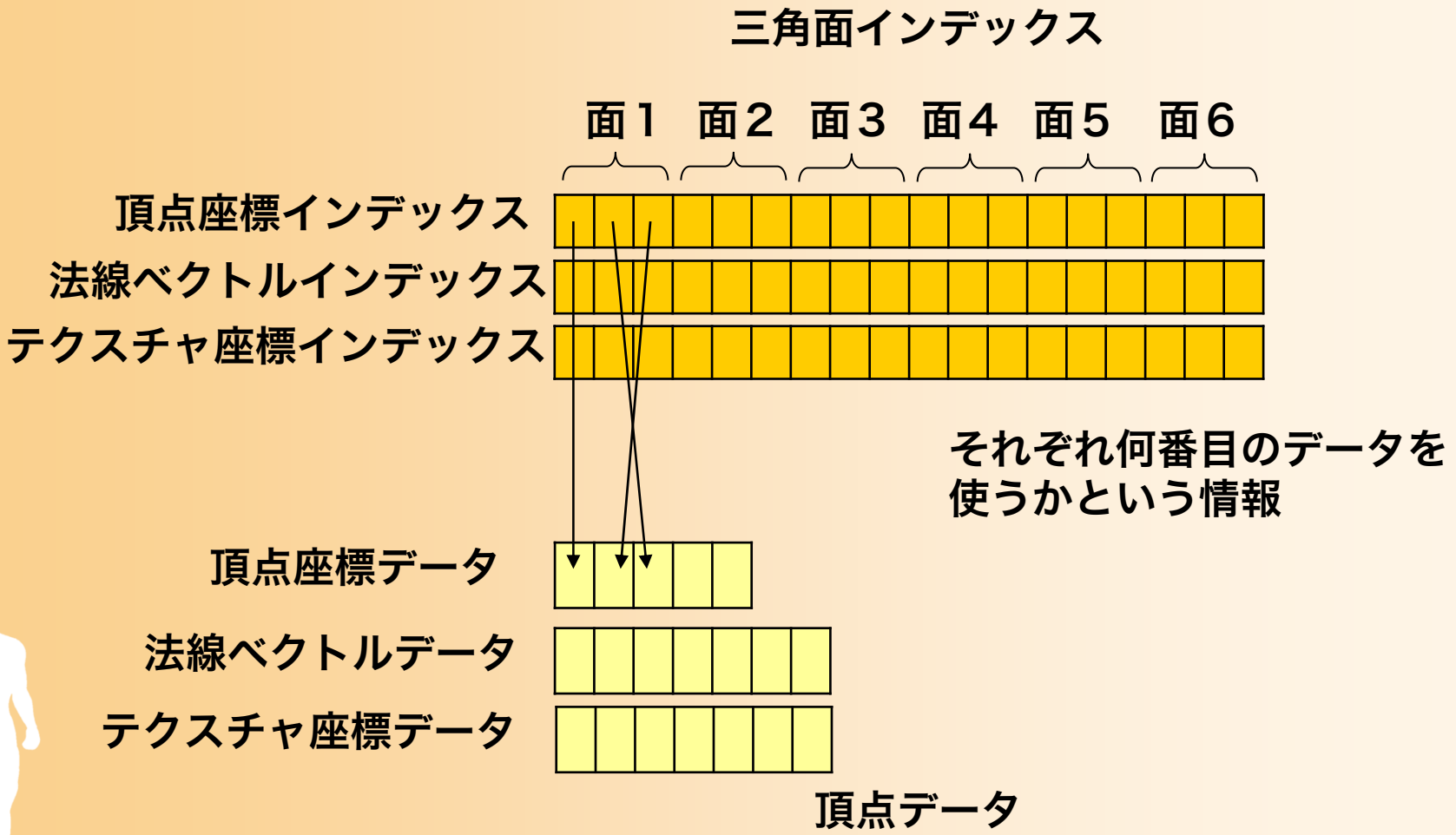
- OpenGL の頂点配列を使うためには、各頂点ごとに頂点座標・法線ベクトル・テクスチャ座標をまとめる必要がある
- データ構造の変換が必要
  - 以下、変換方法の例を説明

頂点配列の利用（1）：頂点配列を使うためのデータ構造

- 頂点配列の機能を利用するためには、頂点ごとにデータをまとめる必要がある

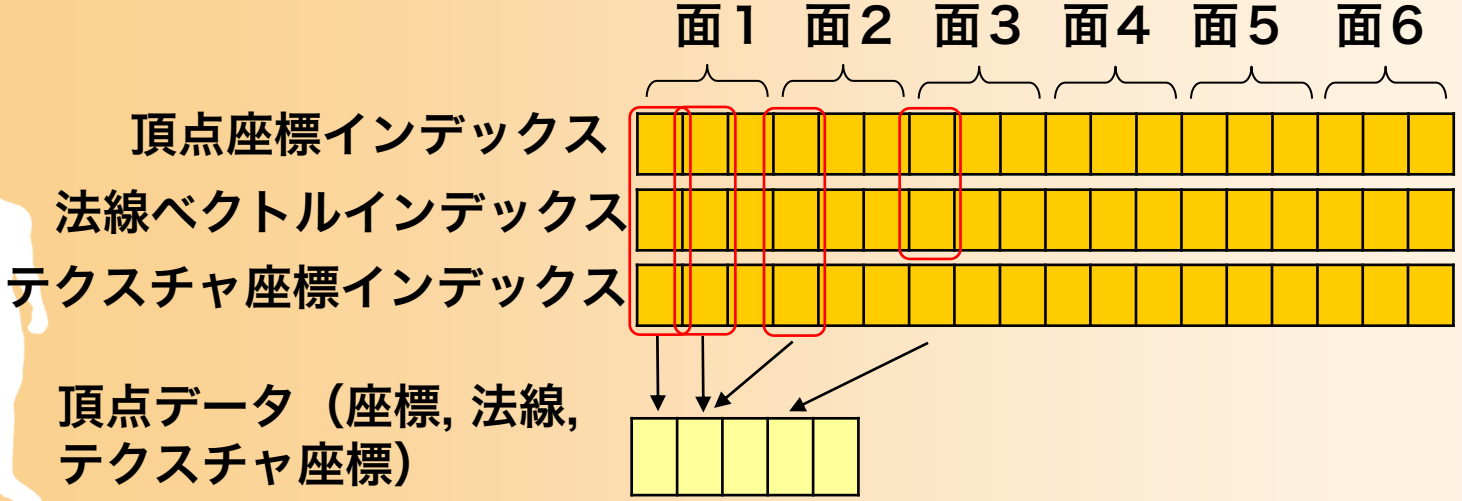


頂点配列の利用（2）：Obj形式のデータ構造



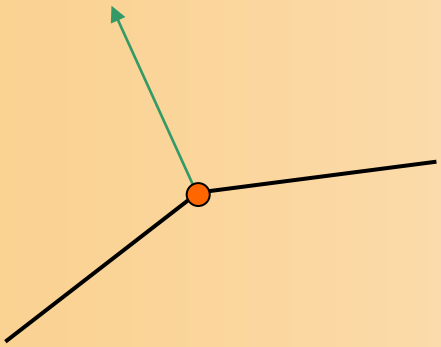
頂点配列の利用（3）：データ構造の変換

- 共通の頂点座標・法線・テクスチャ座標を利用している頂点があれば共通化して利用
  - 頂点座標が同じでも、法線・テクスチャ座標が異なれば、別の頂点データとして扱う
  - 再構成された頂点番号に応じて、三角面インデックスを作成

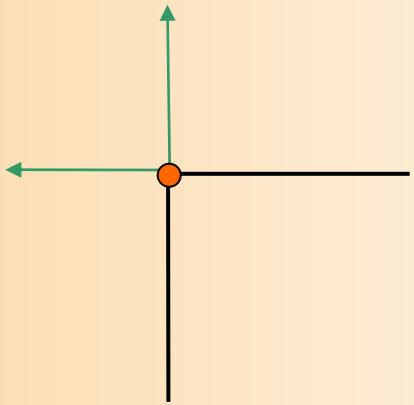


頂点の共通化

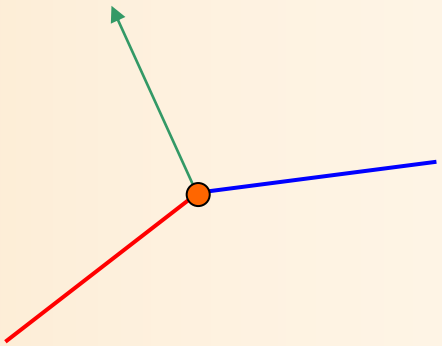
- 共通の頂点座標・法線・テクスチャ座標
  - なめらかな面の場合は、通常、これらの情報は共通になる
  - 角にあたる頂点では法線ベクトルが異なる
  - 模様が変わる頂点ではテクスチャ座標が異なる



なめらかな面では、隣接面で共通の頂点データを使用



角にあたる頂点



左右の面同士で模様が変わる頂点

高度な幾何形状データ処理

幾何形状データ処理（1）

- 今回は、最も基本（必須）となる、ファイルからの幾何形状データ読み込みについて学習
- 幾何形状データ処理（モデリングの技術）に関しては、他にも多くの技術がある
  - 本科目では扱わない
- 形状データの生成・編集に使われるオフライン処理の一部は、モデリングソフトウェアに組み込まれているため、自分でプログラムを作成しなくとも、利用可能

幾何形状データ処理（3）

- サブディビジョンサーフェス（Subdivision Surface、細分割曲面）
  - ポリゴンモデルを繰り返し細分割していき、曲面に近い滑らかなポリゴンモデルを生成
    - ポリゴン枚数が極端に増えるため、リアルタイム処理用のポリゴンモデルの生成に使うのは難しい？
- 幾何形状の簡略化
  - 粗いポリゴンモデルに変換
  - LOD（Level of Detail）：視点からの距離に応じて離れている物体を動的に簡略化して描画

幾何形状データ処理（3）

- 点群からの幾何形状モデル生成
  - レーザセンサ等により計測した、実物体の表面の点データの集合から、ポリゴンモデルを生成
    - 穴への対応、複数の計測データの統合、モデルの簡略化などの課題
- 幾何形状モデルの作成・変形
  - 制作者が意図する形状を直観的・対話的に作成できるモデリングシステム
  - 初心者でも容易にモデリングが行えるシステム
  - 自然な形状の生成・変形

まとめ

- 復習：ポリゴンモデルの描画
- 幾何形状データ
- ファイル形式
- データ構造の定義と描画処理
- ファイル読み込み処理の作成
  - Cによる読み込み処理の実装
  - C++による読み込み処理の実装
- 頂点配列の利用、高度な幾何形状データ処理

次回予告

- キーフレームアニメーション
  - 行列・ベクトルを扱うプログラミング
  - 位置補間
    - 線形補間、Hermit曲線、 Bézier曲線、B-Spline曲線
  - 向きの補間
    - オイラー角
    - 四元数と球面線形補間