コンピュータグラフィックスＳ 演習資料

OpenGL & GLUT の基本関数の説明

九州工業大学 情報工学部 システム創成情報工学科
講義担当：尾下真樹

1. はじめに

本資料では、演習で用いる OpenGL & GLUT の基本的な関数を簡単に説明している。
各関数をどのように使うかは、演習資料を参照すること。
さらに詳しい各関数の定義や利用方法が知りたい場合は、講義で紹介している参考書などを読むこと。

2. GLUT の関数

2.1. 初期化

以下は、GLUT の初期化や設定を行うための関数である。

```
void    glutInit( int argc, char ** argv );
```

引数を解析して各種設定を行う。glut はアプリケーションを起動する時にコマンドラインからオプション付きで起動すると、そのオプションに応じてウィンドウサイズやモードを設定できる機能を持っている。今回の演習では基本的に使わない。

```
void    glutInitDysplayMode( unsigned int mode );
```

引数 mode でウィンドウの画面モードを指定する。オプションには複数指定可能で、各オプションの論理和によって指定する。基本的にはサンプルで使用している設定で十分。

```
void    glutInitWindowSize( int width, int height );
```

GLUT ウィンドウの初期サイズを指定する。

```
void    glutWindowPosition( int x, int y );
```

GLUT ウィンドウの初期表示位置を指定する。

```
int    glutCreateWindow ( char * title );
```

GLUT ウィンドウを生成する。
引数 title には、ウィンドウのタイトルバーに表示する文字列を指定する。
戻り値は、ウィンドウの ID を返す。この ID は、複数のウィンドウを生成した時に、それらを識別するために使用する（今回の演習では使用しない）。

```
void    glutMainLoop();
```

GLUT のイベント処理ループに入る。

2.2. コールバック関数の登録

```
void    glutDisyplayFunc( void (*func)( void ) );
```

表示コールバック関数を登録する。
表示コールバック関数は、ウィンドウを再描画する必要が生じた時に呼ばれる。

```
void    glutReshapeFunc( void (*func)( int width, int height ) );
```

ウィンドウサイズ変更時に呼ばれるコールバック関数を登録する。
コールバック関数は次のような引数・戻り値で定義される。

```
    void (*func)( int width, int height );
```

ウィンドウのサイズが変更された時には、カメラ座標系からウィンドウ座標系への透視変換行列を再設定する必要があるので、そのための処理をここに記述する。

```
void    glutIdleFunc( void (*func)( void ) );
```

アイドルコールバック関数を登録する。
アイドルコールバック関数は、他に処理すべきイベントがない時に継続的に呼ばれる。
アニメーションなどの常に行うべき処理をアイドルコールバック関数に記述する。

```
void    glutMouseFunc( void (*func)( int button, int state, int x, int y ) );
```

マウスコールバック関数を登録する。
コールバック関数は次のような引数・戻り値で定義される。

```
    void (*func)( int button, int state, int x, int y );
```

このコールバック関数は、マウスのボタンが押された、または、離されたされた時に呼び出される。
引数の button にはボタンの種類（GLUT_LEFT_BUTTON, GLUT_MIDDLE_BUTTON, GLUT_RIGHT_BUTTON のどれか）、state にはボタンの状態（GLUT_UP, GLUT_DOWN）、x,y にはマウスカーソルのスクリーン上の座標が入る。

```
void    glutMotionFunc( void (*func)( int x, int y ) );
void    glutPassiveMotionFunc( void (*func)( int x, int y ) );
```

マウス移動コールバック関数を登録する。glutMotionFunc() はドラッグされた状態でマウスが移動する時、glutPassiveMotionFunc() はドラッグされていない状態でマウスが移動する時に呼ばれるコールバック関数を設定する。
マウスのドラッグ開始・終了時には必ずマウスイベントが発生するので、上のマウスコールバック関数が呼ばれる。

```
void    glutKeyboardFunc( void (*func)( unsigned char key, int x, int y ) );
```

キーボードコールバック関数を登録する。コールバック関数は次のような引数・戻り値で定義される。
    void (*func)(unsigned char key, int x, int y);
このコールバック関数は、キーボードのキーが押された時に呼び出される。
引数の key には、押されたキーの文字コードが入る。カーソルキーの押下については、GLUT_KEY_LEFT, GLUT_KEY_RIGHT, GLUT_KEY_UP, GLUT_KEY_DOWN などの値を使って判定できる。

2.3. 画面の再描画を行うための関数

視点やオブジェクトの位置や向きの値を変更したときなど、画面を再描画したいときには、以下の関数を呼び出す。

```
void    glutPostRedisplay();
```

GLUT に描画コールバック関数の呼び出しを要求する。
この関数が呼ばれると、GLUT は次に処理が空いた時に描画コールバック関数を呼び出す。
アイドルコールバック関数内で物体やカメラの位置を移動させたときなど、画面を再描画したいときにはこの関数を実行すると良い。
この関数を使わずに、直接表示コールバック関数を呼び出してはいけない。タイミングによっては、正しく描画されない可能性がある。

2.4. 基本オブジェクトの描画

GLUT では、立方体や球などの基本的なオブジェクトを描画するための補助関数が用意されている。

```
void    glutWireCube( GLdouble size );
void    glutSolidCube( GLdouble size );
```

立方体を描画する。引数 *size* には、立方体の大きさを指定する。
glutWireCube() はワイヤーフレーム（線分のみ）、glutSolidCube() はポリゴンで立方体を描画する。

```
void    glutWireSphere( GLdouble radius, GLdouble slices, GLdouble stacks );
void    glutSolidSphere( GLdouble radius, GLdouble slices, GLdouble stacks );
```

球をポリゴンモデルで近似して描画する。引数 *size* には、球の半径を指定する。引数 *slices, stacks* には、それぞれ球を円周方向・上下方向にいくつに分割することでポリゴンモデルに近似するかを指定する。引数 *slices, stacks* の値が大きい程、より球に近いオブジェクトが描画される。

```
void    glutWireCone( GLdouble radius, GLdouble height, GLdouble slices, GLdouble stacks );
void    glutSolidCone( GLdouble radius, GLdouble height, GLdouble slices, GLdouble stacks );
```

円すいをポリゴンモデルで近似して描画する。引数 *radius, height* には、円すいの半径・高さを指定する。引数 *slices, stacks* は、球の場合と同様。

その他、glutSolidIcosahedron(), glutSolidCone(), glutSolidCone(), glutSolidCone(), などがある。

これらの関数を用いる場合も描画の仕組みは同じであり、これらの関数の内部では glBegin()～glEnd() 呼び出しが実行されている。

2.5. 文字の描画

GLUT の関数を使用して画面に文字を描画できる。

```
void    glutBitmapCharacter( void * font, int character );
```

現在の描画位置に１文字描画する。
引数 *font* には、使用するフォントを指定する。フォントには、GLUT_BITMAP_8_BY_13, GLUT_BITMAP_9_BY_15（以上、固定幅フォント）, GLUT_BITMAP_TIMES_ROMAN_10, GLUT_BITMAP_TIMES_ROMAN_24, GLUT_BITMAP_HELVETICA_10, GLUT_BITMAP_HELVETICA_12, GLUT_BITMAP_HELVETICA_18（プロポーショナルフォント）が使用できる。
引数 *font* には、描画する文字の文字コードを指定する。一度に１文字しか描画できない。

表示する位置は、次のような関数を使用して設定できる。サンプルプログラムのように射影行列を平行投影に設定している場合は、Ｚ座標は関係ないため２次元のバージョンでかまわない。

```
void    glRasterPos2i( GLint x, GLint y );
void    glRasterPos2f( GLfloat x, GLfloat y );
void    glRasterPos3i( GLint x, GLint y, GLint z );
void    glRasterPos3i(GLfloat x, GLfloat y, GLfloat z );
```

文字などを描画する位置を指定する。関数の形式は、glVertex～() 関数と同じ。関数の末尾で引数の次元と型が決まる。

なお、描画する文字のピクセル幅は次の関数を使って調べることができる。プロポーショナルフォントの場合、文字によって必要なピクセル幅が異なるので、文字列全体での文字幅を調べたければ、１文字ずつ文字幅を取得して合計する必要がある。この関数を使うことによって、例えば文字列を画面の右端に寄せて描画したりといったレイアウトが実現できる。

```
int    glutBitmapWidth ( void * font, int character );
```

glutBitmapCharacter() 関数を使用して文字を描画した時のピクセル幅を返す。
引数は glutBitmapCharacter() 関数と同じ。

3. OpenGL の関数

OpenGL の関数は大きく 2 つに分けられる。
- 描画のオプションや座標変換行列などを設定するための関数
- 実際に図形を描画する関数

OpenGL の関数は、関数名が全て gl～で始まる。glu～で始まる関数は、OpenGL の補助ライブラリ（OpenGL Utility Library）の関数であり、OpenGL の関数を組み合わせたり引数を内部で変換したりすることによって、より使いやすいインターフェースを提供するものである。
一方、glut～で始まる関数は、GLUT（OpenGL Utility Toolkit）の関数である。前回の資料で説明したように、GLUT の関数は通常は OpenGL には含まれておらず、GLUT をインストールしないと使えないことに注意する（制御端末室の PC には既に GLUT がインストールされているのでインストールは不要）。

また、OpenGL には、同じ機能の関数でも引数の種類に応じていくつかのバージョンが存在する。
例えば、glVertex3f(*x*, *y*, *z*), glVertex3d(*x*, *y*, *z*) には、それぞれ頂点座標(*x*, *y*, *z*)の各座標値を float 型で指定するか、double 型で指定するかの違いがある。当然、double 型の関数を使用した方が値の精度は高くなるが、float 型を使用した方が処理は高速になるので、必要に応じて使い分ける。

3.1. 座標変換行列の設定

OpenGL はレンダリング・パイプラインで使用する変換行列の情報を内部に持っている。OpenGL の関数を呼び出すことで変換行列を設定できる。一度設定された変換行列は次に変更されるまで有効であり、その間に実行された描画関数の頂点座標には設定した変換行列が常に適用される。
変換行列には、モデルビュー変換行列（モデル座標系からカメラ座標系への変換行列）、射影変換行列（カメラ座標系からスクリーン座標系への変換行列）、テクスチャ変換行列（テクスチャ座標に適用される変換行列）の 3 種類がある。
座標変換を設定する前に、まず、3 種類の変換行列のうちこれからどの変換行列の変更を行うかを次の関数で指定する。

```
void    glMatrixMode( GLenum mode );
```

これからどの行列の変更を行うかを指定する。
引数 *mode* には、GL_MODELVIEW, GL_PROJECTION, GL_TEXTURE のいずれかを指定する。
GL_MODELVIEW はモデルビュー変換行列（モデル座標系からカメラ座標系への変換行列）、GL_PROJECTION は射影変換行列（カメラ座標系からスクリーン座標系への変換行列）、GL_TEXTURE はテクスチャ変換行列（テクスチャ座標に適用される変換行列）を意味する。

テクスチャ変換行列はあまり使われないので、以下では、主に使用するモデルビュー変換行列・射影変換行列のそれぞれについて、設定の方法を説明する。
サンプルプログラムでは、31 行目と 82 行目で GL_MODELVIEW, GL_PROJECTION が使用されている。

3.1.1. モデルビュー変換行列の設定

モデルビュー変換行列は、モデル座標系からカメラ座標系への変換行列を設定する。
設定には次のような関数を使用する。

```
void    glLoadIdentity( void );
```

現在の変換行列を単位行列で初期化する。

void glTranslatef(GLfloat *x*, GLfloat *y*, GLfloat *z*);
void glTranslated(GLdouble *x*, GLdouble *y*, GLdouble *z*);

(*x*, *y*, *z*) 平行移動するような平行移動行列を計算して、現在の変換行列に掛ける。

void glRotatef(GLfloat *angle*, GLfloat *x*, GLfloat *y*, GLfloat *z*);
void glRotated(GLdouble *angle*, GLdouble *x*, GLdouble *y*, GLdouble *z*);

回転軸 (*x*, *y*, *z*) を中心に *angle* 度回転するような回転行列を生成して、現在の変換行列に掛ける。

基本的に変換行列設定の関数は、現在の変換行列を置き換えるのではなく、現在の変換行列に対して新しく生成した変換行列を掛ける仕様になっている。従って、関数を呼び出す度に、どんどん変換行列が掛けられていく形になる。

$$\begin{pmatrix} & & & \\ & \mathbf{A} & & \\ & & & \\ & & & \end{pmatrix}\begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}=\begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix}$$

座標変換式を上式のように表した時、新しく生成された変換行列 $\mathbf{A}_i$ はもとの変換行列 $\mathbf{A}$ に対して右側に順にかけられていく。従って、実際の行列演算は、プログラム内で後から指定した変換行列の方から順番に掛けられることになる（何故このような順番になっているかは理由があるので後述する）。
プログラム内で、変化行列が $\mathbf{A}_1,\cdots,\mathbf{A}_n$ の順番で指定されたとすると、全体での変換行列 $\mathbf{A}$ は下記のようになる。

$$\mathbf{A}=\mathbf{A}_1\mathbf{A}_2\mathbf{A}_3\cdots\mathbf{A}_n$$

上記の平行移動や回転の変換行列をかける以外に、カメラ位置と目標位置を指定して変換行列を指定する関数も用意されている。カメラや目標の位置が与えられているときにはこちらの関数の方が便利である。

void gluLookAt(GLdouble *eye_x*, GLdouble *eye_y*, GLdouble *eye_z*, GLdouble *center_x*, GLdouble *center_y*, GLdouble *center_z*, GLdouble *up_x*, GLdouble *up_y*, GLdouble *up_z*);

カメラ位置(*eye_x*, *eye_y*, *eye_z*)から目標位置(*center_x*, *center_y*, *center_z*)の方を向くような変換行列を計算して、現在の変換行列に計算する。上方向の向きは(*up_x*, *up_y*, *up_z*)で指定する。

また、変換行列を直接設定する関数も用意されている。上記の関数を使用せずに、自分で平行移動や回転の変換行列を行って設定することも可能である。複雑なモデル座標系やカメラ座標系を実現したい時など、場合によっては自分で行列演算を行った方が良いケースもある。

void glLoadMatrixf(const GLfloat * *m*);
void glLoadMatrixd(const GLdouble * *m*);

現在の変換行列を 16 個の要素を指定した配列で初期化する。引数 *m* には、要素数が 16 個の配列を与える。
ただし、引数 *m* を GLdouble *m*[4][4]; のように４×４の行列として定義した時、*m*[*i*][*j*]は変換行列の *j*

行 *i* 列の要素の値を表す（*i* 行 *j* 列ではない点に注意が必要）。

void　glMultMatrixf(const GLfloat * *m*);
void　glMultMatrixd(const GLdouble * *m*);

現在の変換行列に、16 個の要素で指定した変換行列を右側からかける。引数 *m* の定義については、glLoadMatrixf(),glLoadMatrixd()と同じ。

途中で変換行列の状態を記録しておくことが可能である。OpenGL は、内部に、変換行列のためのスタックを持っている。glPushMatrix() 関数や glPopMatrix() 関数を使用することで、現在の変換行列をスタックに退避したり、スタックから変換行列を取得したりすることができる。

void　glPushMatrix (void);

現在の変換行列をスタックに退避する。

void　glPopMatrix (void);

スタックから変換行列を取得して、現在の変換行列として設定する。

glPushMatrix() 関数を使用してスタックにつんだら、必ずその後で glPopMatrix() 関数を呼んでスタックから行列を取得するようにする。glPopMatrix() 関数を呼ばずに glPushMatrix() 関数だけを何度も呼んでいると、スタックがあふれてしまってエラーになる。プログラミングを行う時は、関数の呼び忘れがないように、glPopMatrix() 関数と glPushMatrix() 関数を組にして同時に記述するようにすると良い。

3.1.2. 射影変換行列の設定

射影変換行列はカメラ座標系からスクリーン座標系への変換行列を設定する。
上で説明した平行移動や回転などを使うのではなく、射影変換行列を生成する専用の関数が用意されている。

void　glFrustum(GLdouble *left*, GLdouble *right*, GLdouble *bottm*, GLdouble *top*, GLdouble *near*, GLdouble *far*);

透視射影のための行列を設定する（行列を作成して現在の変換行列にかける）。
視界の錐台形を与えると自動的に透視投影行列が計算される。引数は図を参照。

ただし、glFrustum()は引数 *near* の距離によって視野角が大きく変化するため、望むような設定が難しいと

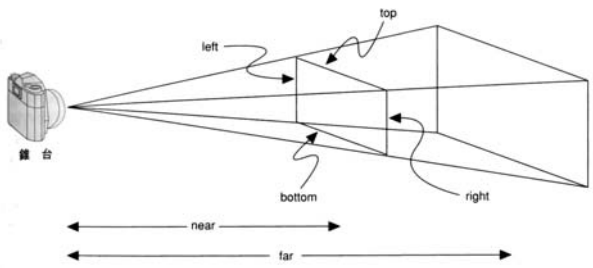

図 1　glFrustum() による透視射影変換の設定

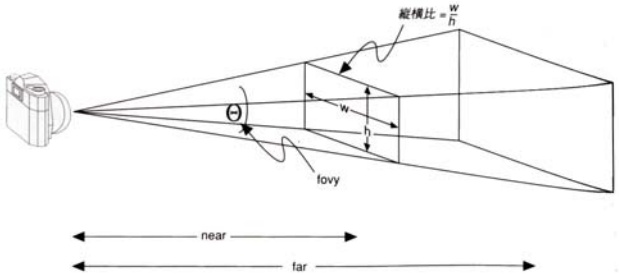

図 1　glPerspective() による透視射影変換の設定

いう問題がある。そこで、glFrustum() よりも使いやすい関数として、gluPerspective()が用意されている。

void　gluPerspective(GLdouble *fovy*, GLdouble *aspect*, GLdouble *near*, GLdouble *far*);
左右対称な透視射影行列を設定する（行列を作成して現在の変換行列にかける）。

なお、射影変換行列とは直接関係はないが、同時に設定されることが多いので、ビューポートの設定関数について以下に説明しておく。
glViewport() は、ウィンドウ内のどの領域に描画を行うかを設定する関数である。ウィンドウの描画領域の広さと同じサイズを指定した場合は、ウィンドウ全体に描画される。設定によっては、例えば、画面の上半分にだけ描画するといったことも可能である。

void　glViewport(GLint *x*, GLint *y*, GLsizei *width*, GLsizei *height*);
画面内の描画領域をスクリーン座標の左上位置と幅・高さで指定する。

実際には、射影変換行列とビューポートの両方が決まって初めて、カメラ座標系からスクリーン座標への変換行列が決定されることになる。
一般に、ウィンドウのサイズが変更された時には、射影変換行列やビューポートを再設定する必要がある。サンプルプログラムでは、コールバック関数 reshape()の内部で、まず glViewport() 関数を使用してウィンドウ全体を描画領域に指定し、その後、gluPerspective() 関数を使用して視野角を 45 ° に設定している。
なお、glFrustum()や gluPerspective ()は、他の関数と同様に、現在の変換行列を置き換えるものではなく、現在の変換行列に生成した変換行列を掛けるものなので、通常はこれらの関数を呼び出す前に関数 glLoadIdentity()を呼んで現在の変換行列を初期化する必要があることに注意する。

なお、透視射影変換を行わず、カメラ座標系の頂点座標をスクリーン座標系に平行に投影する平行射影行列を設定することも可能である。
平行射影変換は、次のような関数で指定する。

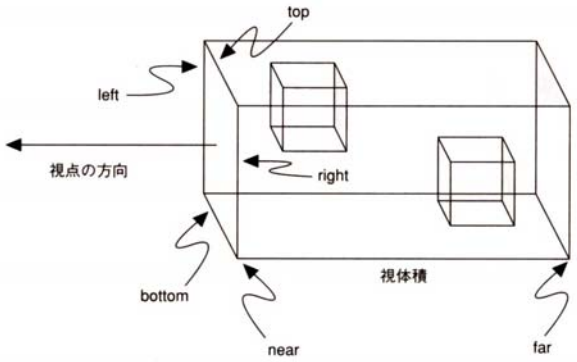

図 1　glOrtho() による平行射影変換の設定

void　glOrtho(GLdouble *left*, GLdouble *right*, GLdouble *bottm*, GLdouble *top*, GLdouble *near*, GLdouble *far*);
平行射影のための行列を設定する（行列を作成して現在の変換行列にかける）。

平行射影変換は、図面をレンダリングする時など、透視変換を行いたくないような場合に用いられる。

3.2. オブジェクトの描画

3.2.1. 画面のクリア

描画を行う前には、画面をまずクリアする必要がある。

void　glClear(GLbitfield *mask*);

バッファをクリアする。どのバッファをクリアするかどうかは、各バッファを表す GL_COLOR_BUFFER_BIT, GL_DEPTH_BUFFER_BIT, GL_ACCUM_BUFFER_BIT, GL_STENCIL_BUFFER_BIT の論理和によって指定する。
通常は、GL_COLOR_BUFFER（画面の色情報）、GL_DEPTH_BUFFER（Ｚバッファ）の２つをクリアする。

クリアする時の背景色は、次の関数を使用して設定する。

void　glClearColor(CLclampf *red*, GLclampf *green*, GLclampf *blue*, GLclamp *alpha*);

カラーバッファのクリア時に塗りつぶす背景色を指定する。
引数 *red, green, blue* で色を設定する。引数 *alpha* は透明度の指定である。透明度は通常は 1.0（不透明）を指定する。各引数の値が 0 以下や 1 以上である場合は、[0,1]の範囲に強制的に設定される。

3.2.2. プリミティブの描画

ポリゴンや線分や線などのプリミティブ（最も基本的な図形）を描画する方法について説明する。
プリミティブを描画する場合は、まず関数 glBegin()を呼んでプリミティブの描画を開始することを宣言する。その後、そのプリミティブを構成する頂点データを順次与えることで、そのプリミティブの形状を指定する。プリミティブの全ての頂点データの指定が終わったら、最後に関数 glEnd()を呼んでここまでがプリミティブのデータであることを宣言する。

void　glBegin(GLenum *type*);

プリミティブの描画の開始。プリミティブの種類を引数 *type* で指定する。
引数 *type* の種類には、GL_POINTS（点を描画）、GL_LINES（線分を描画）、GL_TRIANGLES（三角面を描画）、GL_POLYGON（ポリゴンを描画）などがある。

void　glEnd();

プリミティブの描画の終了。

複数形になっていることからも分かるように、GL_POINTS, GL_LINES, GL_TRIANGLES などを指定した場合は、１つの glBegin()～glEnd()の中に複数のプリミティブのデータを記述できる。
例えば、GL_TRIANGLES はの場合は、3n 個の頂点データを記述することで、n 個の三角面が描画される。一方、GL_POLYGON の場合は、１つの glBegin()～glEnd()で１枚のポリゴンを表すので、各ポリゴンごとに glBegin()～glEnd()を呼び出す必要がある。

各頂点のデータは、次のような関数を使って指定する。

```
void    glColor3f( GLfloat r, GLfloat g, GLfloat b );
void    glColor3d( GLdouble r, GLdouble g, GLdouble b );
```

これ以降の頂点の色を(*r*, *g*, *b*)とする。

```
void    glNormal3f( GLfloat nx, GLfloat ny, GLfloat zn );
void    glNormal3d( GLdouble nx, GLdouble ny, GLdouble zn );
```

これ以降の頂点の法線を(*nx*, *ny*, *zn*) とする。

```
void    glVertex3f( GLfloat x, GLfloat y, GLfloat z );
void    glVertex3d( GLdouble x, GLdouble y, GLdouble z );
```

頂点座標 (*x*, *y*, *z*) の頂点データを送る。頂点の色・法線・テクスチャ座標については、最後に指定した値が使用される。

関数名の末尾の 3f や 3d は、それぞれ 3 個の float 型、3 個の double 型の引数を取ることを表している。この他にも、glVertex2f(), glVertex4f() などの関数も存在する。
サンプルプログラムの例では 6 つの頂点データを与えて、2 枚の三角面を描画している。この例では三角面ごとに頂点の色や法線の色を与えているが、これらの値を各頂点ごとに変えることもできる。

3.3. 機能設定

OpenGL は描画時に適用される多くの機能を持っている。それらの各機能は、必要に応じてオン・オフを切り替えることが可能である。使わない機能はオフを設定することで、処理を高速化できる。

```
void    glEnable( GLenum capability );
```

指定した機能を有効にする。
引数 *capability* には、GL_LIGHTING, GL_COLOR_MATERIAL , GL_DEPTH_TEST, GL_CULL_FACE などの値を指定することができる。
指定可能な機能の種類は、細かいものも含めると全部で 40 以上ある。

```
void    glDisable( GLenum capability );
```

指定した機能を無効にする。

初期設定ではほとんどの機能はオフになっているため、必要な機能は明示的にオンに設定する必要がある。サンプルプログラムでは、GL_LIGHTING（光源による影響を考慮して頂点色を求める）, GL_COLOR_MATERIAL（指定した頂点色を有効にする）, GL_DEPTH_TEST（Ｚテストを有効にする）, GL_CULL_FACE（背面除去を有効にする）などの機能を有効にしている。

また、各機能ごとの個別の設定を行う関数も用意されている。下に挙げているのは、背面除去の対象とする面を設定する関数である。

```
void    glCullFace( GLenum mode );
```

背面除去の対象とする面を指定する。
引数 *mode* には、GL_FRONT, GL_BACK, GL_FRONT_AND_BACK を指定できる。

3.4. 光源の設定

描画されるポリゴンの頂点の色は、光源と物体の素材（頂点の色）によって決まる。
OpenGL では、複数の光源を設定することができる。光源の情報は、以下の関数を使用して設定する。

```
void    glLightf( GLenum light_no, GLenum pname, GLfloat param );
void    glLighti( GLenum light_no, GLenum pname, GLint param );
void    glLightfv( GLenum light_no, GLenum pname, GLfloat * param );
void    glLightiv( GLenum light_no, GLenum pname, Glint * param );
```

光源の情報を設定する。上の２つは１次元のスカラ情報を設定するための関数で、下の２つは３・４次元のベクトル情報を配列を使って設定するための関数である。
引数 *light_no* は、設定する光源の番号で、GL_LIGHT0, GL_LIGHT1, …, GL_LIGHT7 を指定する。
引数 *pname* は、設定する光源の情報の種類で、GL_AMBIENT, GL_DIFFUSE, GL_SPECULAR, GL_POSITION, GL_SPOT_DIRECTION, GL_SPOT_DIRECTION, GL_SPOT_EXPONENT, GL_SPOT_CUTOFF, GL_CONSTANT_ATTENUATION, GL_LINEAR_ATTENUATION, GL_QUADRATIC_ATTENUATION を指定する。
引数 *param* は、設定する値、または、配列を指定する。

GL_AMBIENT, GL_DIFFUSE, GL_SPECULAR は、それぞれ環境光・拡散光・鏡面反射光を設定する。
光源の位置（GL_POSITION）の設定では、光源の位置（方向）を４次元ベクトル（*x,y,z,w*）で設定する。この時、*w* 座標を 1.0 に設定すると点光源になり、（*x,y,z*）で指定された位置に点光源が置かれる。一方、*w* 座標を 0.0 に設定すると平行光源になり、(*x,y,z*)で指定された方向から平行に光が来るものとして計算される。平行光源の方が点光源よりも計算量は少なくなる。
GL_SPOT_DIRECTION, GL_SPOT_EXPONENT, GL_SPOT_CUTOFF を設定することで、点光源にスポットライト効果を設定できる。デフォルトでは、スポットライト効果はなしに設定されている。スポットライト効果については、本演習では詳細は省略する。
また、GL_CONSTANT_ATTENUATION, GL_LINEAR_ATTENUATION, GL_QUADRATIC_ATTENUATION を設定することで光の減衰を表現できる。減衰のパラメタを設定すると、光源から離れるほど光源による効果は小さくなる。デフォルトでは、減衰はなしに設定されている。減衰についても本演習では詳細は省略する。

光源処理では、RGB それぞれの成分ごとに、下の式によって頂点の色が決定される。

$$Color = L_{\mathrm{ambient}} \cdot M_{\mathrm{ambient}} + \max\{\ \mathbf{l}\cdot\mathbf{n},\ 0\ \}\ L_{\mathrm{diffuse}} \cdot M_{\mathrm{diffuse}} + \max\{\ \mathbf{s}\cdot\mathbf{n},\ 0\ \}^{M_{\mathrm{specular\_factor}}}\ \ L_{\mathrm{specular}} \cdot M_{\mathrm{specular}}$$

ベクトル **n** は頂点の法線、**l** は頂点から光源へのベクトル（平行光源の場合は光源の方向）、**s** は頂点から視線へのベクトルとベクトル **l** を足して正規化したベクトルである。$L_{\mathrm{ambient}}, L_{\mathrm{diffuse}}, L_{\mathrm{specular}}$ は、それぞれ光源の環境光成分、拡散反射光成分、鏡面反射光成分を表す。$M_{\mathrm{ambient}}, M_{\mathrm{diffuse}}, M_{\mathrm{specular}}, M_{\mathrm{specular\_factor}}$ は、物体の環境特性、拡散反射特性、鏡面反射特性、鏡面反射係数を表す。
上式から分かるように、拡散光成分は、頂点の法線と光の方向が一致しているほど明るくなる。また、鏡面反射光成分は、法線と入射光の角度と法線と視線方向の角度が一致しているほど明るくなる。複数の光源がある場合は、上式がそれぞれの光源ごとに加算される。

$M_{\text{ambient}}, M_{\text{diffuse}}$ には、glColor3f()で設定された色が使われる。$M_{\text{specular}}, M_{\text{specular\_factor}}$ にはデフォルト値の $(1.0,1.0,1.0)$, 1.0 がそれぞれ使われる。glMaterial() 関数を使用することで、物体特性のそれぞれの値を個別に設定することが可能になる。本演習では詳しい具体例は省略する。

光源を使用する場合は、前回の資料で説明した glEnable() 関数を使用して、照明処理（GL_LIGHTING）、及び、それぞれの光源（GL_LIGHT0, GL_LIGHT1, …）を有効にする必要がある。光源の数が増えるほど処理が遅くなるので、なるべく最小限の光源を使用するようにし、使わない光源は glDisable() 関数で無効に設定する。
なお、ここで使用される OpenGL の光源処理は局所的な直接照明モデルである。他の物体による影や、周囲の物体からの反射などは、通常では一切考慮されないので、これらの効果を加えたい場合は自分で何らかの処理を追加する必要がある。

3.5. 頂点配列の使用

OpenGL を使って３次元図形を描画するための方法として、glBegin()～glEnd()の間に glVertex()などの関数を呼び出してそれぞれの頂点ごとに座標・法線・色などのデータを指定する方法がある。ここでは、このように各頂点ごとに関数を呼び出してデータを設定するのではなく、頂点のデータを配列を使って一度で指定する方法を紹介する。一般に、関数呼び出しのためにオーバーヘッドがかるので、特に頂点数が多いモデルを描画する場合は、こちらの方法を用いた方が効率的である。また、頂点のデータを配列に格納することで、これまでのように glVertex()関数などの引数に直接頂点のデータを記述する方式と比較して、頂点のデータの管理が容易になり、プログラムも短くなるという利点もある。

```
void   glVertexPointer( GLint size, GLenum type, GLsizei stride, const GLvoid * pointer );
```

頂点座標の配列を指定する関数。
引数 *size* は、頂点の座標の次元を指定する。通常は３次元ベクトルなので、３を指定する。
引数 *type* は、頂点座標の型を指定し、GL_FLOAT, GL_DOUBLE, GL_INT などから一つを指定する。
引数 *stride* は、各頂点ごとのデータ間のオフセット。今回のサンプルプログラムのように隙間なく配列にデータが格納されている場合は０を指定する。
引数 *pointer* は、頂点座標の配列のポインタ（アドレス）を指定する。配列の先頭からデータを使用する場合は、配列の変数名をそのまま指定すれば良い。

glVertexPointer()関数は、あくまで配列の先頭アドレスを指定するだけであることに注意する。実際の配列データの転送は、実際に描画命令が呼び出されて、何個の頂点データを使用するかが決まってから行われる。
頂点の法線ベクトル、色、テクスチャ座標などの配列も、同様の関数を使用して指定する。

```
void   glNormalPointer( GLenum type, GLsizei stride, const GLvoid * pointer );
void   glColor( GLint size, GLenum type, GLsizei stride, const GLvoid * pointer );
void   glTexCoordPointer( GLint size, GLenum type, GLsizei stride, const GLvoid * pointer );
```

頂点データの配列を指定する関数。上から順に、頂点の法線ベクトル、色、テクスチャ座標の配列を指定する。引数は、glVertexPointer()と同じ。ただし、glNormalPointer()に関しては、法線の次元は必ず３と決まっているので引数 *size* は不要である。

実際にどの配列を有効にするかは、以下の関数を使用して設定する。

```
void   glEnableClientState( GLenum array_type );
void   glDisableClientState( GLenum array_type );
```

有効化 or 無効化する配列を指定する。引数 *array_type* には、GL_VERTEX_ARRAY, GL_NORMAL_ARRAY, GL_COLOR_ARRAY, GL_TEXTURE_COORD_ARRAY などから一つを指定する。

別のところで指定した配列の情報が残っている可能性があるので、使わない配列については一通り無効化した方が安全である。
実際の描画は、次の関数を使用する。

void glDrawArrays(GLenum *mode*, GLint *first*, GLsizei *count*);

頂点配列のデータを順番に使用して図形を描画する。
引数 *mode* には、描画する図形の種類を指定する。GL_POINTS（点を描画）、GL_LINES（線分を描画）、GL_TRIANGLES（三角面を描画）、GL_POLYGON（ポリゴンを描画）など、glBegin()関数と同様の引数が指定できる。
引数 *first*, *count* には、それぞれ、配列の何番目のデータから、何個のデータを使用するかを指定する。

また、配列データを直接指定するのではなく、インデックスを使用して配列データを参照しながら図形を描画するための関数もある。こちらの関数を使用することで、共通する頂点のデータの重複を防ぐことができるので、頂点データの配列のサイズを小さくすることができ、結果的に処理を効率化することができる。

void glDrawElements(GLenum *mode*, GLint *count*, GLenum *type*, void * *indices*);

インデックスを使用して頂点配列のデータを順番に参照しながら図形を描画する。
引数 *mode* には、glDrawArrays(), glBegin()と同じく図形の種類を指定する。
引数 *count* には、インデックスの配列のサイズを指定する。
引数 *type* には、インデックスの型を GL_UNSIGNED_INT, GL_UNSIGNED_SHORT, GL_UNSIGNED_BYTE のいずれかで指定する。
引数 *indices* には、インデックスの配列の先頭アドレスを指定する。

その他に、頂点配列のデータをひとつづつ呼び出す glArrayElement(Glint *no*); 関数がある。この関数は、上の関数のように配列に格納されている図形全体を描画するのではなく、一部のデータのみを選んで描画したいときに使用する。通常はあまり使うことはないので、詳しい説明は省略する。

3.6. テクスチャマッピング

テクスチャマッピングを使用すると、画像をポリゴンに貼り付けることができる。
テクスチャマッピングを使用する場合は、まず、テクスチャ画像を生成（定義）する必要がある。あらかじめテクスチャ画像のピクセルデータを配列に格納した上で、以下の glTexImage2D() 関数を呼び出し、使用するテクスチャ画像を設定する。

void glTexImage2D(GLenum *target*, GLint level, GLenum *int_format*, GLsizei *width*, GLsizei *height*, GLsizei *border*, GLenum *format*, GLenum *type*, const GLvoid * *pixels*);

頂点座標の配列を指定する関数。
引数 *target* は、作成するテクスチャの種類で、通常は GL_TEXTURE_2D を指定する。
引数 *level* は、テクスチャの解像度で、ミップマップ機能を使用するために指定する。ミップマップ機能を使用すると、粗い画像と詳細な画像を設定しておいて、適切な方を使用することができる。ミップマップを使用しない場合は、0 を指定する。

引数 *int_format* は、テクスチャの内部形式を指定する。引数の種類については、下記の引数 *format* の説明を参照。ここで指定する内部形式は、必ずしも画像データと同じでなくとも良い。
引数 *width* と *height* は、画像データのサイズを指定する。
引数 *border* は、境界の幅を指定する。通常は 0 で良い。
引数 *format* は、引数 *pixels* で渡す画像データのデータ形式を指定する。例えば、各ピクセルが RGB のデータを持つ場合は、GL_RGB を指定する。RGB に加えて透明度の情報を持つ場合は、GL_RGBA を指定する。
引数 *type* は、引数 *pixels* で渡す画像データの各ピクセルのデータ型（サイズ）を指定する。例えば、各ピクセルが 1 バイト（RGB で 3 バイト=24 ビット）のデータを持つ場合は、GL_UNSIGNED_BYTE を指定する。
引数 *pixels* には、引数 *format* と引数 *type* で指定した形式で画像データを格納したメモリのアドレスを指定する。

OpenGL は、画像ファイルの読み込みはサポートしていないので、BMP, GIF, JPEG などの画像ファイルを読み込んで配列に格納する処理については、別のライブラリを使用するか、自分で処理を作成する必要がある。サンプルプログラムでは、24 ビット形式の BMP 画像ファイルを読み込む関数を作成している。
テクスチャマッピングを行うための処理方法は、glTexParameteri() 関数を使用して設定する。

```
void    glTexParameteri( GLenum target, GLenum pname, GLint param );
```

テクスチャマッピングの処理方法を指定する関数。
引数 *target* は、設定する処理を指定する。引数 *pname* には、その処理方法を指定する。引数 *param* は、そのときの数値パラメタを指定する。引数 *target* の種類によって、数値パラメタを持たなければ、引数 *param* は省略可能。
引数 *target* が GL_TEXTURE_MAG_FILTER や GL_TEXTURE_MIN_FILTER のときは、テクスチャを拡大縮小しながら貼り付けるときの計算方法を設定し、引数 *pname* には GL_NEAREST や GL_LINEAR などの方法を指定する。
引数 *target* が GL_TEXTURE_WRAP_S や GL_TEXTURE_WRAP_T のときは、テクスチャの繰り返し貼り付けを行うかどうかを設定し、引数 *pname* には GL_REPEATE のときは、テクスチャ座標が 1 よりも大きな値になったら繰り返し画像を貼り付ける（画像を複数枚並べて貼り付ける）、GL_CLAMP のときは、繰り返し貼り付けは行わない。

テクスチャを使用してポリゴンを描画する場合は、glEnable(GL_TEXTURE_2D) 関数を呼び出し、テクスチャ処理を有効にする。テクスチャの使用が終わったら、glDisable() 関数で無効にする。

最後に、テクスチャ画像をどのように適用するかを設定する。

```
void    glTexEnv( GLenum target, GLenum pname );
void    glTexEnvf( GLenum target, GLenum pname, GLfloat param );
void    glTexEnvfv( GLenum target, GLenum pname, GLfloat * param );
```

テクスチャの適用方法を指定する関数。
引数 *target* は、通常 GL_TEXTURE_ENV を指定する。
引数 *pname* には、テクスチャの適用方法を指定する。指定可能な種類は GL_DECAL, GL_REPLACE, GL_MODULATE, GL_BLEND である。GL_REPLACE を指定すると、ポリゴンの色は無視して、テクスチャ画像を単純に貼り付ける。そのため、光源処理は行われず、光の効果を表すことはできない。GL_DECAL が最も一般的な適用方法で、テクスチャ画像のアルファ値に応じて、もとのポリゴン色とテクスチャ画像を合成する。テクスチャ画像がアルファ値を持たない場合は、GL_REPLACE

と同じ働きになる。
GL_MODULATE と GL_BLEND は、テクスチャを画像として使用するのではなく、テクスチャに設定されている値に応じて、ポリゴンの色を変化させるために使用できる。GL_MODULATE は、テクスチャ画像の値に応じて、ポリゴンの明るさを変化させる。GL_BLEND は、ある色を指定して、テクスチャ画像の値に応じて、ポリゴンの色と指定した色をブレンドする。

以上の流れでテクスチャが設定されていれば、これまでの図形描画と同じやり方で、頂点座標・色・法線に加えて、頂点のテクスチャ座標を与えれば、テクスチャ画像が貼り付けて表示される。

```
void    glTexCoord2f( GLfloat u, GLfloat v );
void    glTexCoord2d( GLdouble u, GLdouble v );
```

頂点のテクスチャ座標 (*u*, v) を指定する。テクスチャ座標は、テクスチャ画像の範囲を 0.0< *u* <1.0, 0.0< *v* <1.0 とした時の (*u*, v) 値で指定する。

複数のテクスチャを使用する場合は、テクスチャを切り替えながら描画を行うことになる。ただし、テクスチャを切り替える度に glTexImage2D () 関数を呼び出していると、毎回テクスチャを初期化することになり無駄な処理時間がかかる。そこで、glBindTexture() 関数を利用することで、一度作成したテクスチャを記録しておき、改めて呼び出して使用することができる。
glBindTexture() 関数については、今回は詳しい説明は省略する。
なお、glBindTexture() 関数を使用したとしても、テクスチャの切り替えには多少のオーバーヘッドがかかるので、なるべく同じテクスチャ画像を使用する図形はまとめて描画するようにして、なるべくテクスチャ切り替えの回数を減らす方が処理速度が速くなる。