Desktop-M3G
チュートリアル

tueda@wolf.dog.cx

このチュートリアルは M3G API(JSR-184) の実装である Desktop-M3G の使い方を解説しています。基本的な 3D グラフィックスと OpenGL の知識がある事を前提にしています。つまりカメラ、透視投影、ローカル座標、ワールド座標、ビュー座標、モデルビュー変換、トライアングルストリップ、メッシュ、マテリアル、ライト、などの用語は既に知っているものとして説明無しに使用しています。やや応用的なスキンメッシュ、アニメーション、スケレタルアニメーション、ボーン、などは簡単な説明を付加していますが、それらの用語を始めて聞く読者には理解が難しいと思われるので専門の書籍を読む事をおすすめします。筆者のお薦めは Jason Gregory 氏の Game Engine Architecture です。

1　イントロダクション

1.1 M3G とは

M3G はモバイル機器向けに定義された 3D グラフィックス API の規格です。Java で記述され Java Platform, Micro Edition (Java ME) プラットフォーム上で動作します。Java Community Process によって JSR-184 として策定された。現在のバージョンは 1.1 です。

M3G は典型的には OpenGL ES と共に使われオブジェクト指向のハイレベルインターフェースを提供します。M3G はキャラクターアニメーションのためのスキンメッシュや、モーフメッシュを提供します。また M3G 規格のファイルフォーマットも定義します。

1.2 M3G の特徴

1.2.1 M3G の機能

M3G の提供する機能は図の通りです。

- カメラ、ライト、マテリアルなど基本描画機能
- アニメーションのためのキーフレームシーケンスおよびコントローラー
- スキンメッシュとモーフメッシュｎ
- M3G ファイルフォーマット
- 例外ベースのエラー処理

M3G は下位層として OpenGL ES を想定していますが Direct X などの他の API も利用可能です。ただし現時点では OpenGL ES 以外を利用する実装はありません。

1.2.2 M3G に含まれないもの

- シェーダープログラミング (*1)
- ポリゴン以外の高次曲面
- パーティクル
- ストリップ以外の指定方法
- 衝突判定および物理演算

(*1) 次の M3G バージョン 2.0 で利用可能になります。

一言で言えば M3G API を使う事で PlayStation2(R) レベルの 3D グラフィックスが記述できるようになります。

1.3 Desktop-M3G とは

Desktop-M3G は M3G API をデスクトップ用に C++ で実装したライブラリです。Linux および FreeBSD で利用可能です。ちなみに筆者の環境は Ubuntu です。MIT ライセンスで配布され自由に利用、変更可能です。Java と C++ の言語の違いから来る修正を除けば、可能な限りオリジナルの M3G API に忠実に実装しています。M3G 形式のファイルフォーマットは 100% 互換性があり、そのまま利用できます。

機能	M3G	Desktop-M3G
プラットフォーム	J2ME(モバイル)	Linux(デスクトップ)
実装言語	Java	C++
下位層	OpenGL ES	OpenGL

1.4 利用条件

Desktop-M3G を利用するには下記のライブラリが必要です。

- OpenGL 2.1 以上
- GLEW

このほかにウィンドウや OpenGL コンテキストを扱うのに、glut や Qt が必要です。

1.5 M3G の構成

M3G は 30 のクラスからなるオブジェクト指向の API です。そのうち半分の 15 は OpenGL をオブジェクト指向にラップしたものです。3 つがアニメーションのためのクラスで、9 つがシーンおよび構成物（メッシュなど）です。残り 3 つはその他ローダーなどです。M3G は OpenGL の上位層として完全に OpenGL を隠蔽します。従って M3G API を使用しているときは OpenGL を直接使用する事はできません。

2　シーンの描画

2.1 描画の基本

この章では実際に M3G API を使ったシーンの描画を解説します。

2.1.1 immediate モードと retained モード

M3G には 1 行 1 行レンダリングコマンドを実行する immediate モードと、シーングラフを作成し一括してレンダリングする retained モードがあります。現在の Desktop-M3G の実装では retained モードのみ対応しています。

2.1.2 シーンとシーンの描画

シーンは World クラスのオブジェクトを頂点とする木構造です。描画は Graphics3D クラスが行います。従って M3G での基本的な描画手順は、

- World を頂点とするシーングラフの作成
- Graphics3D に渡して描画

になります。コードではこのようになります。

```
  World* wld = new World;
Graphics3D* g3d = Graphics3D::getInstance();
g3d->render (wld);
```

Graphics3D クラスは new 演算子でインスタンス化せずに専用の getInstance() メソッドでインスタンス化します。シングルトン化されており常に同じオブジェクトが返ります。上記は完全に正しいコードですがこれだけでは何も表示されません。何か物体を表示するためにはシーンに (1) カメラおよび (2) メッシュを追加する必要があります。

2.2 カメラ

シーンにカメラオブジェクトを追加しアクティブにします。シーンには複数のカメラを設置できますが、同時にアクティブにできるのは 1 つだけです。

```
  Camera* cam = new Camera;
wld->addChild (cam);
wld->setActiveCamera (cam);
```

デフォルトのカメラは平行投影で z 軸負の方向を向いています。これを透視投影に変えるには setPerspective() メソッドを使います。

```
  Camera* cam = wld->getActiveCamera();
cam->setPerspective (45, h/w, 0.1, 100);
```

現在アクティブなカメラは getActiveCamera() メソッドで取得できます。setPerspective() の引数は OpenGL の glPerspective と同じ、fov、アスペクト比、near, far です。詳しくは M3G のリファレンスマニュアルをご覧ください。

2.3 メッシュ

M3G のプリミティブはメッシュのみです。トライアングル 1 個もメッシュとして描画します。あらゆる高次サーフェス（NURBS、Bezier パッチなど）およびパーティクルはありません。メッシュはデータを保持する VertexArray クラスと頂点としてデータを保持する VertexBuffer クラス、インデックスを保持する TriangleStrip クラスからなります。またマテリアルなどを保持する Appearance クラスも同時に渡します。

VertexArray,VertexBuffer,TriangleStrip,Mesh に関しては次章以降のチュートリアルをご覧ください。

2.4 最小限のシーンの描画

この章では４角形を表示する最小限のシーンを描画します。使用する M3G のクラスは World,Camera,Mesh ほかです。ここでは glut を使った完全なコードを示します。

現在コードは省略されています。

2.5 シーンの構成要素

シーンを構成する要素はすべて Node クラスを派生しています。以下のクラスは全て Node を基底クラスに持ち World クラスに挿入可能です。

Camera, Group, Light, Mesh, MorphingMesh, SkinnedMesh, Sprite3D

なおこのほかに World クラスも Node を派生しますが、World クラスに World クラスを挿入する事はできません。

3　M3G の機能

M3G ライブラリには 30 のクラスがあります。この章ではこの全てのクラスとそのメソッドについて解説します。

クラスは大きく分けて、 (1) (1)OpenGL を整理してオブジェクト指向風に変更したクラス 15 種類、(2) シーンノードとしてシーンを構成するクラス 9 種類、(3) アニメーションのためのクラス 3 種類、(4) その他 3 種類、です。

- OpenGL ラッパー： Appearance, Background, Image2D, Fog, CompositingMode, PolygonMode, VertexArray, VertexBuffer, IndexBuffer, TriangleStripArray, Graphics3D
- シーンノード： ObjectD, Transformable, Node, Group, Mesh, Sprite3D, World, SkinnedMesh, MorphingMesh
- アニメーション：AnimationTrack, AnimationController, KeframeSequence
- その他：Loader, RayIntersection, Transform

3.1 ライト

Light クラスはライトを表すシーンノードです。インスタンス化した後に World クラスに addChild() メソッドで追加します。ライトには環境光 (AMBIENT), 点光源 (OMNI), 平行光源 (DIRECTIONAL), スポットライト (SPOT) の 4 種類があります。ライトにはカラーと強度が設定できます。

```
  Light* lig = new Light;
wld->addChild (lig);
```

デフォルトは平行光源で Z 軸負の方向を向いています。カラーは白色 (1,1,1) で強度は 1.0 です。

3.1.1 共通設定

ライトの種類は setMode() メソッドで変更します。

```
  void  setMode (int mode);
```

mode には環境光 (AMBIENT), 点光源 (OMNI), 平行光源 (DIRECTIONAL), スポットライト (SPOT) のいずれかを指定します。

現在のライトの種類は getMode() メソッドで取得できます。

```
  int  getMode () const
```

ライトの色は setColor() メソッドで指定し、強度は setIntensity() メソッドで設定します。色と強度の積が最終的なライトの出力になります。

```
  void  setColor (int rgb)
void  setIntensity (float intensity)
```

引数の rgb には r(0-255),g(0-255),b(0-255) を rgb フォーマットで整数型にパックして渡します。intensity は強度を任意の浮動小数値で指定します。0 および負の値も指定可能です。

```
  rgb = (r<<16) || (g<<8) || (b<<0);
```

現在の色の取得には getColor() メソッドを、強度の取得には getIntensity() メソッドを使います。

```
  int  getColor () const
float  getIntensity () const
```

ここまでは全てのライトに共通です。

3.1.2 環境光

シーン全体を環境光の色で照らします。設定できる項目は共通設定の項のみです。

3.1.3 平行光源

シーンを無限遠方から平行に照らします。光は減衰しません。デフォルトで Z 軸負の方向を向いているので、これにアフィン変換を施して望みの方向に向けます。例えば Y 軸負の方向を向いた平行光源を作成する場合はこうなります。

```
  Light* lig = new Light;
lig->setMode (Light::DIRECTIONAL);
lig->postRotate (-90, 1,0,0);
```

3.1.4 点光源

シーンをある 1 点から照らします。距離によって減衰します。デフォルトでは原点に位置するのでアフィン変換を施して望みの位置に動かします。例えば Y 軸 (0,10,0) の位置に動かすにはこのようになります。

```
Light* lig = new Light;
lig->setMode (Light::OMNI};
lig->tranlate (0,10,0);
```

点光源の距離による減衰は setAttenuation() メソッドを使って制御できます。

```
  void setAttenuation (float constant, float linear, float quadratic);
```

減衰係数 α は引数の constant(c), linear(l), quadratic(q) を使って以下の式で計算されます。

$$\alpha = \frac{c + l \cdot r + q \cdot r^2}{2}$$

定数項 c および距離 r に比例する r、距離 r の 2 乗に比例する q を設定します。例えば物理的に正しい距離の 2 乗に反比例する点光源は setAttenuation() に constant=0, linear=0, quadratic=1 を指定します。

デフォルトは OpenGL と同じく constant=1, linear=0, quadratic=0 で、距離による減衰はなく一定の強さで照らされます。

現在の減衰項は getConstantAttenuation() メソッド、getLinearAttenuation() メソッド、getQuadraticAttenuation() メソッドで取得できます。

```
float  getConstantAttenuation () const
float  getLinearAttenuation () const
float  getQuadraticAttenuation () const
```

3.1.5 スポットライト

シーンをスポットライトで照らします。距離によって減衰します。スポットライトの位置と方向は点光源や平行光源と同じようにアフィン変換で制御します。スポット角度は setSpotAngle() メソッドで変更し、スポット指数は setSpotExponent() メソッドで変更します。

```
void  setSpotAngle (float angle)
```

```
void  setSpotExponent (float exponent)
```

引数の angle は中心線からライトの照らす一番外側までのカットオフ角度を 0〜90 ° の間で指定します。デフォルトは angle=45 ° です。exponent は中心から外周部に向けての減衰指数を 0〜128 の範囲で指定します。デフォルトは exponent=0.0 で減衰はなく一様に照らされます。

スポットライト効果 (β) は中心線からの角度と exponent(e) を使って以下の式で計算されます。

$$\beta = (\cos\theta)^e$$

カットオフ角度より大きな角度の領域は 0 です。

現在の設定値は getSplotAngle() メソッド、getSpotExponent() メソッドで取得できます。

```
float  getSpotAngle () const
float  getSpotExponent () const
```

3.2 バックグラウンド

背景の色および画像の設定は Background クラスで扱います。インスタンス化したのち World クラスの setBackground() メソッドで登録します。

```
Background* bg = new Background;
wld->setBackground (bg);
```

3.2.1 背景色

レンダリング前にフレームバッファーをクリアするかどうかの設定は setColorClearEnable() メソッドで行います。デフォルトは enable=yes で、黒色、透明 (0,0,0,0) でクリアされます。

```
void  setColorClearEnable (bool enable)
```

現在のカラバッファーのクリア設定は isColorClearEnabled() メソッドで取得できます。

```
bool  isColorClearEnabled () const
```

同様に Z バッファーのクリアは setDepthClearEnable() メソッドで行います。デフォルトは enable=yes で、Z バッファーの表現できる最大値でクリアされます。

```
void  setDepthClearEnable (bool enable);
```

現在のデプスバッファーのクリア設定は isDepthClearEnabled() メソッドで取得できます。

```
bool  isDepthClearEnabled () const
```

背景色の設定は setColor() メソッドで行います。背景画像が設定されていない場合はビューポート全体がこの色でクリアされます。画像が設定されていて繰り返しモードが BORDER の場合、画像の範囲外はこの色でクリアされます。デフォルトは黒色, 透明 (0,0,0,0) です。

```
void  setColor (int argb)
```

引数は argb 形式で指定します。

```
int argb = (a << 24) | (r << 16) | (g << 8) | (b << 0);
```

3.2.2 背景画像

背景を単一の色で塗りつぶす代わりに画像を使うには setImage() メソッドを使います。背景画像には jpeg か png が利用できます。現在の Desktop-M3G の実装では png のみサポートしています。デフォルトで背景画像は設定されていません。

```
void  setImage (Image2D *image)
```

引数の image には画像を設定済みの Image2D オブジェクトを渡します。

設定した画像は getImage() メソッドで取得できます。

```
Image2D *  getImage () const
```

デフォルトでは指定した画像がビューポート全体に拡大もしくは縮小されて表示されます。setCrop() メソッドで画像のクロップ領域（画像から一部分を切り出した矩形）を設定する事ができます。

```
void  setCrop (int crop_x, int rop_y, int width, int height)
```

4 つの引数はクロップ領域を指定します。(crop_x,crop_y) は画像の左上の点をピクセル単位で、(width,height) は大きさをピクセル単位で指定します。width,height には 0 以上の値を指定します。クロップ領域の指定は画像サイズ (0,0)〜(image_width-1,image_height-1) に限定されません。画像サイズを超える領域を指定した場合の取り扱い方法は setImageMode() メソッドで指定します。クロップ領域の使い方として例えば毎フレームクロップ領域を移動する事で背景画像をスクロールさせる事ができます。

現在のクロップ領域は getCropX(), getCropY(), getCropWidth(), getCropHeight(), メソッドで取得します。

```
int  getCropHeight () const
int  getCropWidth () const
int  getCropX () const
int  getCropY () const
```

setImageMode() メソッドは背景画像の繰り返しモードを設定します。

```
void  setImageMode (int mode_x, int mode_y)
```

引数の mode_x, mode_y にはそれぞれ X 方向と Y 方向のイメージモードを BORDER と REPEAT の 2 つから選択します。X 方向と Y 方向は独立して指定できます。BORDER は領域外を背景色で塗りつぶします。REPEAT は領域外を画像を繰り返します。画像を敷き詰めた仮想的なタイルから色を取得します。デフォルトは X 方向、Y 方向共に BORDER です。

現在のイメージモードは getImageModeX(), getImageModeY() メソッドで取得します。

```
int  getImageModeX () const
int  getImageModeY () const
```

3.3 カメラ

Camera クラスはカメラを表すシーンノードの構成要素の 1 つです。カメラはインスタンス化して World オブジェクトに追加したあと、アクティブ化すると有効になります。アクティブ化できるカメラは同時に 1 つだけです。

```
Camera* cam = new Camera;
wld->addChild (cam);
wld->setActiveCamera (cam);
```

3.3.1 カメラの種類

カメラは透視投影と平行投影か、もしくは任意の 4x4 の変換行列を指定できます。デフォルトは平行投影で、原点 (0,0,0) に位置し Z 軸負の方向を向いています。

3.3.2 平行投影

カメラを平行投影にするには setParallel() メソッドを使います。

```
void  setParallel (float height, float aspect_ratio, float near, float far)
```

引数で指定するパラメーターは OpenGL と同じです。第 1 引数には縦方向の視野領域 (top-bottom) を、第 2 引数にはアスペクト比 (width/height)、第 3 引数と第 4 引数で near クリッピング面と far クリッピング面を指定します。

現在のこれらのパラメーターは getProjection メソッドで取得できます。

```
int  getProjection (float *params) const
```

引数の params には最低 float4 つが書き込める領域を指定します。順番に height, aspect_ratio, near, far が書き込まれれます。

3.3.3 透視投影

カメラを透視投影にするには stePerspective() メソッドを使います。

```
void  setPerspective (float fovy, float aspect_ratio, float near, float far)
```

引数で指定するパラメーターは OpenGL と同じです。第 1 引数は縦方向の fov を角度 (degree) で、第 2 引数はアスペクト比 (width/height)、第 3 引数と第 4 引数で near クリッピング面と far クリッピング面を指定します。

現在のパラメーターは getProjection メソッドで取得できます。

```
int  getProjection (float *params) const
```

引数の params には最低 float4 つが書き込める領域を指定します。順番に fovy, aspect_ratio, near, far が書き込まれれます。

3.3.4 任意の射影変換

平行投影、透視投影以外にも任意の 4x4 行列を指定して射影変換を行う事ができます。変換行列は Transform オブジェクトを作成して渡します。ここで指定する行列は OpenGL の PROJECTION 行列に相当します。

```
void  setGeneric (const Transform &transorm)
```

引数には適切に設定した Transform 行列を渡します。射影行列は逆行列が計算できる必要があります。行列は内部でコピーして保存されます。従ってメソッド呼び出し後の変更は反映されません。

現在のパラメーターは getProjection メソッドで取得できます。

```
int getProjection (Transform* transform) const;
```

引数には結果を受け取る Transform オブジェクトのポインターを指定します。

3.3.5 カメラの位置、方向

カメラの位置と視線方向の変更は Camera クラスの基底クラスである Transformable クラスを使ってアフィン変換を指定して行います。デフォルトでは位置は原点 (0,0,0) で視線方向は Z 軸負の向き (0,0,-1) で up ベクトルは Y 軸正の向き (0,1,0) です。例えばここから位置を Z 軸上の点 (0,0,10) に移動するにはこのようにします。

```
Camera* cam = new Camera;
cam->translate (0,0,10);
```

通常のアフィン変換（回転、平行移動）を駆使して任意の方向にカメラを設定するのはかなり手間がかかります。Desktop-M3G では M3G 非標準ですが glut の glLookAt() 関数に相当する lookAt() メソッドを実装しています。

3.3.6 lookAt 関数

カメラ位置（from_x, from_y, from_z), 視線上の 1 点 (to_x, to_y, to_z)、up ベクトル (up_x, up_y, up_z) を指定してカメラの位置、向きをわかりやすく設定します。glut の gluLookAt() 関数に相当します。up ベクトルは正規化されている必要はありません。

```
void  lookAt (float from_x, float from_y, float from_z, float to_x, float to_y, float to_z, float up
```

3.4 フォグ

フォグは Fog クラスで制御します。フォグは距離に応じてポリゴンカラーとフォグカラーをブレンドして表示します。

ポリゴンの色 (Ci) とフォグの色 (Cf) は後述のブレンド係数 (f) を使って以下の式で計算されます。

$$C = fC_i + (1 - f)C_f$$

フォグはインスタンス化した後シーン (World) に追加します。

```
Fog* fog = new Fog;
wld->addChild (fog);
```

フォグには線形フォグと指数フォグの 2 種類があり setMode() メソッドで切り替えます。

```
void setMode (int mode);
```

mode には LINEAR か EXPONENTIAL を指定します。それぞれ LINEAR が線形フォグ、EXPONENTIAL が指数フォグを表します。デフォルトは LINEAR です。

現在のフォグの種類は getMode() メソッドで取得できます。

```
int getMode () const;
```

フォグの色は setColor() メソッドで指定します。

```
void setColor (int rgb);
```

引数の rgb にはフォグカラーを int 型で 0x00RRGGBB の形式で指定します。

現在のフォグカラーは getColor() メソッドで取得できます。

```
int getColor () const;
```

3.4.1 線形フォグ

ビュー座標での距離 (z) に比例してフォグカラーが変化するフォグです。フォグのブレンディング係数 (f) は以下の式で計算されます。

$$f = \frac{far - z}{far - near}$$

near と far は setLinear() メソッドで指定します。

```
void setLinear (float near, float far);
```

通常 $near < far$ ですが逆でもかまいません。その場合は物体が近づくにつれてフォグカラーが強く表示されます。near == far は指定できません。

現在の near,far は getNearDistance() メソッドと getFarDistance() メソッドを使って取得できます。

```
float getNearDistance () const;
float getFarDistance () const;
```

3.4.2 指数フォグ

ビュー座標での距離 (z) に対して指数的にカラーが変化するフォグです。フォグのブレンディング係数 (f) は以下の式で計算されます。

$$f = \exp^{-(density\dot{z})}$$

density は setDensity() メソッドで指定します。

```
void setDensity (float density);
```

density は 0 以上の浮動小数値を指定ます。ブレンディングの式は線形フォグの時と同じです。
現在の density は getDensity() メソッドを使って取得します。

```
float getDensity () const;
```

3.5 マテリアル

Material クラスはプリミティブの反射特性を制御します。シェーディングには OpenGL と同じフォンモデルを利用します。マテリアルはアピアランスの構成要素の 1 つで Appearance のクラスにセットして使います。

```
Matrial* mat = new Matrial;
app->setMaterial (mat);
```

マテリアル特性は環境光 (AMBIENT)、拡散光 (DIFFUSE)、鏡面光 (SPECULAR)、鏡面指数 (SHININESS)、放射光 (EMISSIVE) から構成されます。

3.5.1 カラーの設定

setColor() メソッドで各特性のカラーを設定します。

```
void  setColor (int target, int argb);
```

第 1 引数の targete には変更したいマテリアル特性を環境光 (AMBIENT)、拡散光 (DIFFUSE)、鏡面光 (SPECULAR)、放射光 (EMISSIVE) の中から 1 つを選択します。第 2 引数の argb には設定する色を指定します。フォーマットは各 1 バイトで argb の順番です。α 成分は DIFFUSE のみ有効で AMBIENT, SPECULAR, EMISSIVE の場合は無視されます。

```
int argb = (a << 24) | (r << 16) | (g << 8) | (b << 0);
```

デフォルトは下記の通りです。

AMBIENT	0x00333333
DIFFUSE	0xFFCCCCCC
SPECULAR	0x00000000
EMISSIVE	0x00000000

現在のカラーは getColor() メソッドで取得できます。

```
int  getColor (int target) const
```

引数の target には取得したいマテリアル特性を指定します。

鏡面反射に関してはこのほかに指数部の係数を setShininess() メソッドで指定します。デフォルトは shininess=0.0 です。

```
void  setShininess (float shininess)
```

現在の shininess パラメーターは getShininess() メソッドで取得できます。

```
float  getShininess () const;
```

3.5.2 バーテックスカラートラッキング

バーテックスカラートラッキングが有効な場合環境光 (AMBIENT) と拡散光 (DIFFUSE) は、VertexBuffer で設定されたデフォルトカラーか（もしあるなら）頂点毎に設定された頂点カラーで置き換えられます。頂点カラーを使う場合は setVertexColorTrackingEnable() メソッドを使って有効化します。デフォルトは enable=false です。この機能は現在実装されていません。

```
void  setVertexColorTrackingEnable (bool enable)
```

現在のバーテックスカラートラッキングの設定は isVertexColorTrackingEnabled() メソッドで取得します。

```
bool isVertexColorTrackingEnabled () const;
```

3.6 ポリゴンモード

ポリゴンに関する種々の設定は PolygonMode クラスで行います。設定できる項目は OpenGL とほとんど同じワインディング、カリング、シェーディング他です。OpenGL にあるポリゴンの輪郭線描画モードはありません。ポリゴンは必ず塗りつぶします。

```
PolygonMode* pmode = new PolygonMode;
app->setPolygonMode (pmode);
```

3.6.1 ワインディング

ポリゴンの表面の判定基準を setWinding() メソッドで行います。ポリゴンの表裏はカリング処理で使用されます。

```
void  setWinding (int mode)
```

mode には WINDING_CCW, WINDING_CW のうちから 1 つを選択します。それぞれ順に反時計回り、時計回りを” ポリゴンの表” と定義します。デフォルトは WINDING_CCW です。

現在のワインディングモードは getWinding() メソッドで取得できます。

```
int  getWinding () const
```

3.6.2 カリング

カリングの設定を setCulling() メソッドで行います。

```
void  setCulling (int mode)
```

mode には CULL_BACK, CULL_FRONT, CULL_NONE のうちから 1 つを指定します。それぞれ順に、背面カリング、前面カリング、カリング無しです。前面と背面の判定はワインディングの設定に依存します。デフォルトは CULLING_BACK です。

現在のカリングモードは getCulling() メソッドで取得できます。

```
int  getCulling () const
```

3.6.3 両面ポリゴン

setTwoSidedLightingEnable() メソッドはシェーディング計算を表面と裏面の両方で行うか表面のみで行うかを制御します。

```
void  setTwoSidedLightingEnable (bool enable)
```

典型的には背面カリングが有効なとき裏を向いたポリゴンは破棄されるので、シェーディングを行う必要はありません。デフォルトは enable=false で、表面のみのシェーディングされます。

現在の両面ライティングの設定は isTwoSidedLightingEnabled() メソッドで取得します。

```
bool  isTwoSidedLightingEnabled () const
```

3.6.4 シェーディング

フラットシェーディングとスムースシェーディングの選択を setShading() メソッドで行います。

```
void  setShading (int mode)
```

mode には SHADE_FLAT または SHADE_SMOOTH を指定します。SHADE_FLAT はフラットシェーディング、SHDE_SMMOTH はスムースシェーディングを行います。デフォルトは SHADE_SMOOTH です。

現在のシェーディングモードは getShading() メソッドで取得できます。

```
int  getShading () const
```

3.6.5 ローカルカメラライティング

setLocalCameraLightingEnable() メソッドで鏡面反射の計算で視点位置をカメラの位置（ローカル）で行うか、無限遠方で行うかを選択します。カメラ位置を無視して無限遠方で計算した方が処理は軽くなります。Desktop-M3G のデフォルトは enable=true です。

（注意）オリジナルの M3G ではデフォルトは enable=false ですが、現在デスクトップ用に売られている GPU ではローカルカメラライティングは問題になりません。

ローカルカメラライティングの設定は” ヒント” です。GPU によってはこれを無視する事があります。

```
void  setLocalCameraLightingEnable (bool enable)
```

現在のローカルカメラライティングの設定は isLocalCameraLightingEnabled() メソッデ注得できます。

```
bool  isLocalCameraLightingEnabled () const
```

3.6.6 透視変換補正

setPerspectiveCorrectionEnable() メソッドで色とテクスチャーの透視変換補正の有無を切り替えます。Desktop-M3G のデフォルトは enable=true です。

（注意）オリジナルの M3G ではデフォルトは enable=false ですが、現在デスクトップ用に売られている GPU では透視変換補正は問題になりません。

透視変換補正の設定は” ヒント” です。GPU によってはこれを無視する事があります。

```
  void setPerspectiveCorrectionEnable (bool enable);
```

現在の透視変換補正の設定は isPerspectiveCorrectionEnabled() メソッドで取得できます。

```
bool  isPerspectiveCorrectionEnabled () const
```

3.7 コンポジティングモード

ピクセル単位のコンポジット方法は CompositingMode クラスで設定します。ここで制御できるのは、デプステステト、αテスト、フレームバッファーへの書き込みおよびブレンド方法、デプスオフセットです。M3G はステンシルバッファー、アキュムレーションバッファーには対応していません。

3.7.1 デプステスト

setDepthTestEnable() メソッドはデプステストを有効化・無効化します。デプスバッファーそのものが有効化されていないときは無視されます。デフォルトは enable=true です。

```
void  setDepthTestEnable (bool enable)
```

M3G ではデプステストは常に OpenGL で言う LEQUAL です。つまりｚ値がデプスバッファーの値より小さい（手前にある）時、デプステストを通過します。

現在のデプステストの有効・無効は isDepthTestEnabled() メソッドで取得できます。

```
bool  isDepthTestEnabled () const
```

3.7.2 αテスト

setAlphatThreashold() メソッドはフラグメントのαテストの閾値を指定します。デフォルトは threshold=0.0 です。

```
void  setAlphaThreshold (float threshold)
```

M3G ではαテストは常に OpenGL で言う GEQUAL です。つまりフラグメントのα値が threashold 以上の時テストを通過します。指定した threshold より小さいαを持つフラグメントはフレームバッファーに書き込まれることなく棄却されます。threshold=0 を指定するとαテストそのものを無効化できます。

現在の閾値は getAlphaThreshold() メソッドで取得できます。

```
float  getAlphaThreshold () const
```

3.7.3 デプスバッファーへの書き込み

setDepthWriteEnable() メソッドでデプスバッファーへの書き込みを制御します。

```
void  setDepthWriteEnable (bool enable)
```

デフォルトは enable=true です。enable=false の時デプスバッファーへの書き込みは行われません。このメソッドはデプステストが行われるかどうかには関与しません。典型的には半透明物体の書き込み時にオフにされます。

現在のデプスバッファーへの書き込みの可否は isDepthWriteEnabled() メソッドで取得できます。

```
bool  isDepthWriteEnabled () const
```

3.7.4 カラーコンポーネントへの書き込み

setColorWriteEnable() メソッドでフレームバッファーのカラーコンポーネントへの書き込みを制御します。

```
void  setColorWriteEnable (bool enable)
```

カラーコンポーネントは r,g,b の 3 チャネル同時に制御し、チャネル毎に個別に制御する事はできません。α チャネルには影響しません。

現在のカラーコンポーネントへの書き込みの可否は isColorWriteEnalbed() メソッドで取得します。

```
bool  isColorWriteEnabled () const
```

3.7.5 α コンポーネントへの書き込み

setAlphaWriteEnable() メソッドでフレームバッファー α コンポーネントへの書き込みを制御します。カラーコンポーネントには影響しません。

```
void  setAlphaWriteEnable (bool enable)
```

現在の α コンポーネントへの書き込みの可否は isAlphaWriteEnalbed() メソッドで取得します。

```
bool  isAlphaWriteEnabled () const
```

3.7.6 ブレンディング方法

フラグメントをフレームバッファーへ書き込むときの演算方法を制御します。

```
void  setBlending (int mode)
```

引数の mode には REPLACE, ALPHA, ALPHA_ADD, MODULATE, MODULATE_x2 の中から 1 つを選択します。フラグメントを $C_s = (R_s, G_s, B, s, A_s)$、既存のフレームバッファーを $C_d = (R_d, G_d, B_d, A_d)$ とするとブレンド後のフレームバッファーの値は下記の通りになします。

REPLACE	$C_d = C_s$
ALPHA	$C_d = C_s A_s + C_d$
ALPHA_ADD	$C_d = C_s A_s + C_d(1 - A_s)$
MODULATE	$C_d = C_s C_d$
MODULATE_x2	$C_d = 2C_s C_d$

現在のブレンディングモードは getBlending() メソッドで取得できます。

```
int  getBlending () const
```

3.7.7 デプスオフセット

setDepthOffset() メソッドでデプスオフセットの設定を行います。デプスオフセットはデプスバッファーの有限の精度から生じる問題を解消するために、z 値にごくわずかなオフセットを足します。

```
void  setDepthOffset (float factor, float units)
```

下記の式を使って引数の factor と units から最終的なオフセット量を計算します。

$$offset = m \cdot factor + r \cdot units$$

r はデプスバッファーの最小精度で、m はトライアングルの z 方向の最大勾配です。

現在のデプスオフセットの設定は getDepthOffsetFactor() メソッドと getDepthOffsetUnits() メソッドで取得します。

```
float  getDepthOffsetFactor () const
float  getDepthOffsetUnits () const
```

3.8 テクスチャー

テクスチャーは Texuter2D クラスで管理します。ただし画像そのものは Image2D クラスが扱います。1D テクスチャーおよび 3D テクスチャーは M3G ではサポートしていません。Texuter2D クラスのコンストラクタで Image2D クラスのポインターを受け取ってインスタンス化します。テクスチャー番号を指定して Appearance クラスにセットします。マルチテクスチャーの枚数の上限は実装依存です。（クエリーの方法も書く）M3G ではテクスチャー画像に jpeg と png が使用できます。現在の Desktop-M3G の実装では png のみサポートしています。M3G では OpenGL2.1 を要求しているのでテクスチャーサイズは 2 の階乗である必要はありません。ただし一般的には GPU は 2 の階乗のサイズの方が高速です。

```
Image2D* img = new Image2D (IMAGE::RGBA, image.png);
Texture2D* tex = new Texture2D (img);
app->setTexutre(0, tex);
```

画像を後から差し替えるには setImage() メソッドを使います。

```
void  setImage (Image2D *image)
```

現在設定されている画像を取得するには getImage() メソッドを使います。

```
Image2D *  getImage () const
```

3.8.1 テクスチャーフィルター

setFiltering() メソッドでテクスチャーフィルターの選択を行います。M3G ではミップマップは（GPU が対応していれば）常に有効です。

```
void  setFiltering (int level_filter, int image_filter)
```

第 1 引数の image_filter は画像内のフィルターを FILTER_NEAREST, FILTER_LINEAR の中から選択します。FILTER __ NEAREST は最近傍画素を、FILTER_LINEAR は周辺 4 画素を線形補間します。第 2 引数の level_filter はミップマップ階層間のフィルターを、FILTER_BASE, FILTER_NEAREST, FILTER_LINEAR の中から選択します。FILTER_BASE は常にミップマップレベル 0 の画像を使用します。つまりミップマップ処理を行いません。

レベルフィルター	画像フィルター	説明	同等な OpenGL
BASE_LEVEL	NEAREST	ベースレベルをポイントサンプリング	NEAREST
BASE_LEVEL	LINEAR	ベースレベルをバイリニアサンプリング	LINEAR
NEAREST	NEAREST	1 番近いミップマップレベルをポイントサンプリング	NEAREST_MIPMAP_N
NEAREST	LINEAR	1 番近いミップマップレベルをバイリニアサンプリング	NEAREST_MIPMAP_N
LINEAR	NEAREST	2 つのミップマップレベルをポイントサンプリング	NEAREST_MIPMAP_L
LINEAR	LINEAR	2 つのミップマップレベルをバイリニアサンプリング	LINEAR_MIPMAP_LIN

現在のテクスチャーフィルターは getImageFilter() メソッドと getLevelFilter() メソッドを使って取得します。

```
int  getImageFilter () const
int  getLevelFilter () const
```

3.8.2 繰り返しモード

setWrapping() メソッドでテクスチャーの範囲外アクセスの繰り返しモードを指定します。

```
void  setWrapping (int wrap_s, int wrap_t)
```

横 (S) と縦 (T) は個別に指定でき、それぞれ WRAP_CLAMP, WRAP_REPEAT から選択します。WRAP_CLAMP は最外周ピクセルにクランプされます。WRAP_REPAT は画像がタイル状に繰り返されます。

現在の繰り返しモードは getWrappingS() メソッドと getWrappingT() メソッドで取得できます。

```
int  getWrappingS () const
int  getWrappingT () const
```

3.8.3 ブレンディングモード

setBlending() メソッドはテクスチャーカラーのブレンドの方法の制御方法を設定します。現在 RGB フォーマットおよび RGBA フォーマット以外の動作を確認していません。

```
void  setBlending (int func)
```

引数の func には FUNC_ADD, FUNC_BLEND, FUNC_DECAL, FUNC_REPLACE の中から 1 つを選択します。

テクスチャーフォーマット	REPLACE	MODULATE	DECAL	BLEND	ADD
ALPHA	$C_v = C_f$	$C_v = C_f$	未定義	$C_v = C_f$	$C_v = C_f$
	$A_v = A_f$	$A_v = A_f$	未定義	$A_v = A_f A_t$	$A_v = A_f A_t$
LUMINANCE	$C_v = C_t$	$C_v = C_f C_t$	未定義	$C_v = C_f(1 - C_t) + C_c C_t$	$C_v = C_f + C_t$
	$A_v = A_f$	$A_v = A_f$	未定義	$A_v = A_f$	$A_v = A_f$
LUM_ALPHA	$C_v = C_t$	$C_v = C_f C_t$	未定義	$C_v = C_f(1 - C_t) + C_c C_t$	$C_v = C_f + C_t$
	$A_v = A_f$	$A_v = A_f A_t$	未定義	$A_v = A_f A_t$	$A_v = A_f A_t$
RGB	$C_v = C_t$	$C_v = C_f C_t$	$C_v = C_t$	$C_v = C_f(1 - C_t) + C_c C_t$	$C_v = C_f + C_t$
	$A_v = A_f$	$A_v = A_f$	$A_v = A_f$	$A_v = A_f$	$A_v = A_f$
RGBA	$C_v = C_t$	$C_v = C_f C_t$	$C_v = C_t$	$C_v = C_f(1 - C_t) + C_c C_t$	$C_v = C_f + C_t$
	$A_v = A_t$	$A_v = A_f A_t$	$A_v = A_f$	$A_v = A_f A_t$	$A_v = A_f A_t$

現在のブレンディングモードは getBlending() メソッドで取得します。

```
int  getBlending () const
```

3.8.4 ブレンドカラー

setBlendColor() メソッドはブレンディングモード FUNC_BLEND で使用するブレンドカラーを設定します。現在この機能は実装されていません。

```
void  setBlendColor (int rgb)
```

引数のブレンドカラーは RGB フォーマットで指定します。

```
int rgb = (r << 16) | (g << 8) | (a << 0);
```

現在のブレンドカラーは getBlendColor() メソッドで取得できます。

```
int  getBlendColor () const
```

3.9 スプライト

Sprite3D はシーンの構成ノードの 1 つで 2D 画像を表示されます。スプライトはスクリーン空間に表示される矩形領域です。スプライトには後述のスケールドとアンスケールドの 2 種類があります。Desktop-M3G ではテクスチャーを張った板ポリゴンとして実装されています。インスタンス化した後 World クラスに追加します。

```
Sprite3D* spr = new Sprite3D (false, img, app);
wld->addChild (spr);
```

コンストラクタの引数は第 1 引数がスケールド、アンスケールドのフラグで、第 2 引数がスプライト画像として使う Image2D オブジェクトへのポインター、第 3 引数がアピアランスです。

3.9.1 スケールドスプライト

スプライトはコンストラクタでスケールド・スプライトとアンスケールドスプライトを選択します。スケールドは視点からの距離応じて大きさが変化するスプライトで、アンスケールドは距離にかかわらず常に一定サイズで表示されるスプライトです。

現在のスケールド、アンスケールドは isScaled() メソッドを使って取得できます。

```
bool  isScaled () const
```

3.9.2 スプライト画像

画像はコンストラクタで Image2D オブジェクトを渡します。画像を差し替えるには setImage() メソッドを使います。

```
void  setImage (Image2D *image)
```

第 1 引数には Image2D オブジェクトを渡します。

現在の画像は getImage() メソッドで取得します。

```
Image2D *  getImage () const
```

3.9.3 スプライトのアピアランス

setAppearance() メソッドでアピアランスをセットします。

```
void  setAppearance (Appearance *appearance)
```

現在のアピアランスは getAppearance() メソッドで取得できます。

```
Appearance *  getAppearance () const
```

3.9.4 クロップ領域

画像のクロップ領域を setCrop() メソッドを使って設定します。クロップ領域はピクセルで指定します。

```
void  setCrop (int cropX, int cropY, int width, int height)
```

現在のクロップ領域は getCropX(), getCropY(), getCropWidth(), getCropHeight() メソッドで取得します。

```
int  getCropHeight () const
int  getCropWidth () const
int  getCropX () const
int  getCropY () const
```

3.10 アピアランス

M3G ではオブジェクトの色、テクスチャー、コンポジット設定、フォグなど見え方に関する設定を全てまとめてアピアランス (Appearance) と呼びます。Appearance クラスは Material, Texture2D, PolygonMode, CompositingMode, Fog クラスを構成要素として持つホルダーです。

構成要素を新しくセットするメソッドは以下の通りです。何もセットしない場合は適当なデフォルトが使用されます。

```
void  setCompositingMode (CompositingMode *compositingMode)
void  setFog (Fog *fog)
void  setMaterial (Material *material)
void  setPolygonMode (PolygonMode *polygonMode)
void  setTexture (int index, Texture2D *texture)
```

テクスチャーのみマルチテクスチャーのためのテクスチャー番号の指定が必要です。現在設定されているクラスを取得するメソッドは以下の通りです。

```
CompositingMode *  getCompositingMode () const
Fog *  getFog () const
Material *  getMaterial () const
PolygonMode *  getPolygonMode () const
Texture2D *  getTexture (int index) const
```

3.10.1 レイヤー

M3G では描画はレイヤー順に行われます。すなわちレイヤー番号 0 に所属するメッシュ、スプライトが全て描画された後、レイヤー番号 1 のが描画されます。以下 2,3,…と続きます。同じレイヤー内では半透明な物体より不透明な物体が先に描画さます。ここでいう不透明な物体とは CompositingMode のブレンディングモードが REPLACE（デフォルト）の物を指します。それ以外は全て不透明な物体として扱われます。

現在この機能は実装されていません。描画順は保証されません。

```
void  setLayer (int layer);
```

引数の layer にはレイヤー番号を 0 以上の整数値で指定します。デフォルトは layer=0 で、最高プライオリティです。

現在所属するレイヤー番号は getLayer() メソッドで取得します。

```
int  getLayer () const
```

3.11 メッシュ

メッシュはシーンの構成要素の 1 つで Mesh クラスで扱います。メッシュは頂点データを取り扱う VertexBuffer クラスと面を定義するインデックス配列の IndexBuffer クラス、材質を定義する Appearance クラスを持ちます。

```
Mesh* msh = new Mesh (vertices, submeshes, indices, apps);
wld->addChild (msh);
```

3.11.1 コンストラクタ

頂点データと面を構成するインデックス配列およびアピアランスを指定してインスタンス化します。1 つのインデックス配列が 1 つのサブメッシュを定義し、1 つまたは複数のサブメッシュを同時に指定できます。

```
Mesh (VertexBuffer *vertices, int num_submesh, IndexBuffer **submeshes, int num_appearance, Appear
Mesh (VertexBuffer *vertices, IndexBuffer *submesh, Appearance *appearance)
```

現在のサブメッシュ数は getSubmeshCount() メソッドで取得できます。

```
int  getSubmeshCount () const
```

現在のインデックス配列を取得するには getIndexBuffer() メソッドを使用します。

```
IndexBuffer *  getIndexBuffer (int index) const
```

引数の index にはサブメッシュの番号を 0 から始まる整数値で指定します。
現在の頂点データを取得するには getIndexBuffer() メソッドを使います。

```
VertexBuffer *  getVertexBuffer () const
```

アピアランスは後から setAppearance() メソッドを使って変更する事ができます。NULL を指定するとデフォルトのアピアランスが使われます。

```
void  setAppearance (int index, Appearance *appearance)
```

第 1 引数の index にはサブメッシュの番号を指定します。第 2 引数の appearance にはセットしたい Appearace オブジェクトを指定します。NULL を指定するとデフォルトのアピアランスが使われます。
サブメッシュに定義されているアピアランスを取得します。

```
Appearance *  getAppearance (int index) const
```

第 1 引数の index にはサブメッシュの番号を指定します。

3.12 スキンメッシュ

キャラクターアニメーションのためのスキンメッシュは SkinnedMesh クラスが扱います。SkinnedMesh はシーンの構成要素です。

現在 Desktop-M3G では未実装です。

3.13 キーフレームシーケンス

アニメーションはキーフレームと呼ばれる、ある時刻に対するデータを連続して補完する事で実現されます。キーフレームの集合をキームレームシーケンスと呼び KeyframeSequence クラスで扱います。キーフレームは整数値の時刻 t と浮動小数点のデータからなります。キーフレームシーケンスは複数のキーフレームを持ち、任意時刻 t に対するデータの補間方法や、時刻の有効期間も定義します。

キーフレームは単なるデータの集合であり、データをどのように解釈するかは AnimationTrack クラスが行います。

キーフレームシーケンスの基本的な構築方法は、

- キーフレーム数を指定してキーフレームシーケンスの作成
- 有効範囲 (valid range) の設定
- 継続期間 (duration) の設定

キーフレームシーケンスはキーフレーム数と 1 フレーム当たりのコンポーネント数、キーフレーム間の補間方法を指定してインスタンス化します。

```
KeyframeSequence (int numKeyframes, int numComponents, int interpolation)
```

第 1 引数の num_keyframes にはキーフレームの数を、第 2 引数の num_components にはキーフレームの保持するコンポーネント数を、第 3 引数の interpolation には補間方法を STEP, LINEAR, SPLINE, SLERP, SQUAD の中から指定します。それぞれ STEP がステップ補間、線形補間、スプライン補間、クォータニオンのスプライン補間、クオータニオンの球面線形補間を表します。

設定されたキーフレーム数は getkeyframeCount() メソッドで取得します。

```
int  getKeyframeCount () const
```

設定されたのコンポーネント数は getComponentCount() メソッドで取得します。

```
int  getComponentCount () const
```

設定された補間方法は getInterpolationType() メソッドで取得します。

```
int  getInterpolationType () const
```

3.13.1 補間方法

補間方法は STEP, LINEAR, SPLINE, SLERP, SQUAD の中から選択します。

	STEP	ステップ補間	$v = v_i$
	LINEAR	線形補間	$v = (1 - s)V_i + sV_{i+1}$
ここに表	SPLINE	スプライン補間	略（M3G の仕様書を参照してください)
	SLERP	クォータニオンの球面線形補間	略（M3G の仕様書を参照してください)
	SQUAD	クォータニオンのスプライン補間	略（M3G の仕様書を参照してください)

3.13.2 キーフレームの設定

コンストラクタで指定したキーフレームの数だけキーフレーム毎に setKeyframe() メソッドを呼び出してデータを設定します。

```
void  setKeyframe (int index, int time, float *value)
```

引数の index は 0 から始まるキーフレーム番号です。存在しないフレーム番号を指定するとエラーになります。引数 time には時刻を 0 以上の整数値で指定し、引数 value にはデータへのポインターを渡します。データは M3G ライブラリ内部にコピーされます。セットされたキーフレームは setValidRange() で有効化される必要があります。

3.13.3 有効範囲の設定

セットされたキーフレームのうち有効なフレームを setValidRange() メソッドで指定します。単に setKeyframe でセットしただけでは有効になりません。

```
void  setValidRange (int first, int last)
```

引数の first と last は 0 から始まるフレーム番号を指定します。上限は getKyframeCount()-1 です。first と last は有効なフレームに含みます。存在しないフレーム番号を指定するとエラーになります。

現在の有効範囲は getValidRangeFirst() メソッド、getValidRangleLast() メソッドで取得できます。

```
int  getValidRangeFirst () const
int  getValidRangeLast () const
```

3.13.4 継続時間の設定

このキーフレームシーケンスの継続時間を setDuration() メソッドで指定します。継続時間とはこのキーフレームシーケンスが再生される長さ（時間）です。

```
void  setDuration (int duration)
```

引数の duration には 0 より大きな整数値を指定します。デフォルトは 0 です。

現在の継続時間は getDuration() メソッドで取得できます。

```
int  getDuration () const
```

3.13.5 繰り返しモード

このキーフレームシーケンスの繰り返しモードを setRepeatMode() で指定します。繰り返しは継続時間を超える時刻のアニメーションの再生方法に関係します。

```
void  setRepeatMode (int mode)
```

引数の mode は CONSTANT, LOOP から選択します。LOOP は継続時間のアニメーションを繰り返します。CONSTANT は繰り返し無しで継続時間を超える時刻には最後のフレームが使われます。

現在の繰り返しモードは getRepeatMode() メソッドで取得できます。

```
int  getRepeatMode () const
```

3.14 アニメーショントラック

AnimationTrack クラスは（単なるデータの集合である）キーフレームシーケンスを特定のアニメーション可能なプロパティに結びつけます。

3.15 コンストラクタ

AnimationTrack はキーフレームシーケンスとターゲットプロパティを指定します。

```
AnimationTrack (KeyframeSequence *sequence, int property)
```

コンストラクタには引数を 2 つ渡します。1 つは KeyframeSequence クラスへのポインタで、もう 1 つはそれを適用する property です。property は ALPHA, AMBIENT_COLOR, COLOR, CROP, DENSITY, DUFFUSE_COLOR, FAR_DISTANCE, FIELD_OF_VIEW, INTENSITY, MORPH_WEIGHTS, NEAR_DISTANCE, ORIENTATION, PICKABILITY, SCALE, SHININESS, SPECULAR_COLOR, SPOT_ANGLE, SPOT_EXPONENT, TRANSLATION, VISIBILITY の中から 1 つを選択します。指定できるターゲットプロパティとコンポーネント数、適応可能なクラスには制限があります。

プロパティ名	コンポーネント数	適応可能なクラス
ALPHA	1	Node, Background, Material, VertexBuffer
AMIBENT_COLOR	3	Material
COLOR	3	Light, Background, Fog, Texture2D, VertexBuffer
CROP	2,4	Sprite3D, Background
DENSITY	1	Fog
DIFFUSE_COLOR	3	Material
EMISSIVE_COLOR	3	Material
FAR_DISTANCE	1	Camera, Fog
FIELD_OF_VIEW	1	Camera
INTENSITY	1	LIght
MORPH_WEIGHTS	1	MorphingMesh
NEAR_DISTANCE	1	Camera, Fog
PICKABILITY	1	Node
SCALE	1,3	Trasformable
SHININESS	1	Material
SPECULAR_COLOR	3	Material
SPOT_ANGLE	1	Light
SPOT_EXPONENT	1	Light
TRANSLATION	3	Transformable
VISIBILITY	1	Node

アニメーショントラックにはアニメーションコントローラーをセットする必要があります。

```
void  setController (AnimationController *controller)
```

引数の controller には AnimationContoroller を指定します。１つの AnimationController を複数の AnimationTrack にセットできます。

現在設定されているアニメーションコントローラーは getController() メソッドで取得できます。

```
AnimationController *  getController () const
```

現在設定されているキーフレームシーケンスは getKeyframeSequence() メソッドで取得できます。

```
KeyframeSequence *  getKeyframeSequence () const
```

現在のアニメーション対象のプロパティは getTragetProperty() メソッドで取得できます。

```
int  getTargetProperty () const
```

p

3.16 アニメーションコントローラー

AnimationContoller はアニメーションの再生スピードなどを制御します。アニメーショントラックは必ずアニメーションコントローラーに結びつけられている必要があります。

```
AnimationController* ctlr = new AnimationController;
track->setAnimationController (ctlr);
```

3.16.1 アクティブ期間

このアニメーションが有効な期間を setActiveInterval() メソッドで指定します。

```
void  setActiveInterval (int start, int end)
```

このコントローラーがアクティブな期間をワールド時間 (start,end] で指定します。start は含み、end は含みません。このコントローラーに関連づけられたアニメーショントラックは、この期間以外は再生されません。start=end は特別な値で、指定すると全ての時刻で有効と見なされます。

現在のアクティブ期間は getActiveIntervalStart() メソッドと getActiveIntervalEnd() メソッドで取得できます。

```
int  getActiveIntervalStart () const
int  getActiveIntervalEnd () const
```

ある時刻 (t=world_time) でアクティブかどうかは isActiveInterval() メソッドで取得できます。この関数は M3G 非標準です。

```
bool  isActiveInterval (int world_time) const
```

3.16.2 再生速度

速度はこのコントローラーに関連づけられたアニメーショントラックの再生速度を変更します。再生速度は setSpeed() メソッドで変更します。

```
void  setSpeed (float speed, int world_time)
```

1 番目の引数の speed は再生速度です。デフォルトは speed=1.0 で、例えば 2.0 に設定するとアニメーションターゲットは 2 倍速で再生されます。2 番目の引数の world_time は現在のワールド時間を指定します。内部的にアニメーションを連続して再生するのに必要です。

現在の再生速度は getSpeed() メソッドで取得できます。

```
float  getSpeed () const
```

3.16.3 アニメーションウェイト

setWeight() メソッドでこのコントローラーのウェイトを設定します。ウェイトは 2 つのアニメーションをブレンドし、なめらかに遷移するのに使用されます。

```
void  setWeight (float weight)
```

引数の weight は 0 以上の浮動小数値を指定します。
現在のウェイト値は getWeight() メソッドで取得します。

```
float  getWeight () const
```

3.16.4 リファレンス時刻の設定

リファレンスとなるシーケンス時間を setPosition() メソッドを使って設定します。

```
void  setPosition (float sequence_time, int world_time)
```

第 2 引数には現在の world_time を指定します。これは内部的にアニメーションを連続して再生ために必要です。
現在の再生位置は getPosition() メソッドで取得できます。

```
float  getPosition (int world_time) const
```

現在のリファレンスのワールド時間は getRefWorldTime() メソッドで取得できます。

```
int  getRefWorldTime () const
```

3.17 Object3D

Object3D はシーンの構成要素（シーンオブジェクト）の基底となる抽象クラスです。全てのシーンの構成要素はこのクラスを継承します。

3.17.1 アニメーション

addAnimationTrack() メソッドでこのオブジェクトにアニメーションを追加します。

```
virtual void addAnimationTrack (AnimationTrack* animation_track);
```

引数には適切に設定された AnimationTrack オブジェクトを渡します。アニメーショントラックのターゲットプロパティが、このオブジェクトに適応可能でなければいけません。

既に設定したアニメーションを取り除くには removeAnimationTrack() メソッドを使います。

```
void removeAnimationTrack (AnimationTrack* animation_track);
```

引数には取り除きたい AnimationTrack オブジェクトを指定します。

3.17.2 ユーザー ID

全てのシーンオブジェクトは任意のユーザー ID を持ちます。ユーザー ID の設定および解釈はライブラリの使用者の自由です。ユーザー ID の設定には setUserID() メソッドを使います。デフォルトは 0 です。

```
void setUserID (int userID);
```

現在設定されているのユーザー ID を取得するには getUerID() メソッドを使います。

```
int getUserID () const;
```

3.17.3 ユーザーオブジェクト

全てのシーンオブジェクトは任意のユーザーオブジェクトを 1 つ持ちます。ユーザーオブジェクトはキー、バリューからなる任意のデータで、その設定と解釈はライブラリの使用者の自由です。この機能は現在実装されていません。

ユーザーオブジェクトを設定するには setUserObject() メソッドを使います。

```
void setUserObject (const char* name, void* value);
```

第 1 引数にはキーとなる文字列を、第 2 引数には値となるデータを渡します。

現在設定されているユーザーオブジェクトを取得するには getUserObject() メソッドを使います。

```
void* getUserObject () const;
```

3.17.4 デバッグ表示

M3G 非標準ですがシーンの構成要素を表すクラスは全てデバッグ用にの print() メソッドを持ちます。print() メソッドはそのクラスの情報のみを表示するので、基底クラスの情報も表示したい場合は明示的に基底クラスの名前で print() メソッドを修飾して呼び出してください。例えば Mesh クラスで全ての情報を表示するには以下のように書きます。

```
mesh->print (cout);
mesh->Node:: print (cout);
mesh->Transformable:: print (cout);
mesh->Object3D:: print (cout);
```

3.18 トランスフォーム可能なオブジェクト

シーンの構成要素のうちアフィン変換可能なものは Transformable クラスを基底クラスに持ちます。具体的には Node クラスと Texture2D クラスの２つが該当します。

3.18.1 基本的な概念

M3G ではトランスフォーメーションは４つの要素（平行移動、回転、拡大縮小、汎用 4x4 行列）を独立に保持・管理し、最後に掛け合わせます。掛け合わせる順番は、１：汎用 4x4 行列 (M)、２：拡大縮小 (S)、３：回転 (R)、４：平行移動 (T) の順番です。

$$p' = TRSMp$$

変換を指定するときもこの４つを別々に指定します。

3.18.2 平行移動

平行移動は translate() メソッドで指定します。

```
void translate (float tx, float ty, float tz);
```

引数には移動ベクトル (tx,ty,tz) を指定します。translate() メソッドは現在の T に指定した移動ベクトルを足します。

setTransform() メソッドは現在の T を破棄し指定した移動ベクトルで置き換えます。

```
void setTranslation (float tx, float ty, float tz);
```

引数には Transform オブジェクトを指定します。

現在の平行移動 (T) を取得するには getTranslation() メソッドを使います。

```
void getTranslation (float* xyz) const;
```

引数には結果を書き込む float3 つ分の領域を指すポインターを渡します。

3.19 回転

回転は postRotate() メソッドまたは preRotate() メソッドで行います。preRotate() メソッドは回転要素に指定された回転を左から乗算します。postRoate() メソッドは回転要素に指定された回転を右から乗算します。

```
void preRotate (float angle, float ax, float ay, float az);
void postRotate (float angle, float ax, float ay, float az);
```

第 1 引数の angle には回転角度を deree で指定します。第 2，3，4 引数で回転軸 (ax,ay,az) を指定します。回転軸は単位ユニットである必要はありません。

setOrientation() メソッドは現在の回転 (R) を破棄し指定した回転で置き換えます。

```
void setOrientation (float angle, float ax, float ay, float az);
```

引数は同様に角度と回転軸を指定します。

```
void getOrientation (float* angle_aixs) const;
```

3.19.1 拡大縮小

拡大縮小は scale() メソッドを使います。

```
void scale (float sx, float sy, float sz);
```

引数には X 方向、Y 方向、Z 方向の拡大縮小率を指定します。1.0 が現状維持です。
setScale() メソッドは現在の S を破棄し指定した拡大縮小率で置き換えます。

```
void setScale (float sx, float sy, float sz);
```

引数には同様に X,Y,Z 方向の拡大縮小率を指定します。
現在の拡大縮小 (S) を取得するにはは getScale() メソッドを使います。

```
void getScale (float* xyz) const;
```

引数には結果を書き込む float3 つ分の領域を指すポインターを渡します。

3.19.2 汎用 4x4 行列

M3G ではトランスフォームを表すのに平行移動、回転、拡大縮小の他に、任意の 4x4 行列 (M) を保持します。これはアフィン変換では行えない変換を指定するのに使います。

```
void setTransform (const Transform& transform);
```

引数には適切な 4x4 行列を設定した Transform オブジェクトを渡します。
現在の 4x4 行列 (M) を取得するには getTransform() メソッドを使います。

```
void getTransform (Transform* transform) const;
```

引数には結果を書き込む Transform オブジェクトを渡します。

3.20 ノード

Node はシーンに存在する全てのオブジェクトの抽象基底クラスです。

オブジェクトには Camera, Light, Group, World, Mesh, SkinnedMesh, MorphingMesh, Sprite3D があります。これらはインスタンス化された後 World クラスの addChild() メソッドでシーンに追加されます。

このノードの親ノードは getParent() メソッドで取得します。親ノードは Group もしくは World に addChild() で追加されたときに自動的に設定されます。

```
Node* getParent () const;
```

親が居ない場合は NULL が返ります。

3.20.1 スコープ

ノードには ” スコープ ” が設定でき、シーンの階層構造に関係なくノードをグルーピングできます。スコープは int 型の整数値で、ビット演算の AND を取ったときに 0 以外の値が存在するときに同じグループと見なされます。

$$scope_A \& scope_B ! = 0$$

デフォルトは-1 で、つまり全てのグループに所属します。グループ設定は、可視性カリング (visibility culling)、ライティング (Lighting)、ピッキング (pickig) に関係します。

スコープは setScope() メソッドで設定します。

```
void setScope (int scope);
```

現在の設定されているスコープは getScope() メソッドで取得します。

```
int getScope () const;
```

3.20.2 ノード α

ノードには setAlphaFactor() メソッドで α 値を設定する事ができます。α 値はシーン階層を伝搬します。

```
void setAlphaFactor (float alpha_factor);
```

引数の alpha_factor には α 値を [0,1] で指定します。α =0 に設定したノードは完全に不可視になります (レンダリングはされます)。デフォルトは α =1 です。

3.20.3 レンダリングフラグ

このノードがレンダリングされるかどうかを setRenderingEnable() メソッドで設定します。

```
void setRenderingEnable (bool enable);
```

現在の設定は isRenderingEnabled() メソッドで取得します。

```
bool isRenderingEnabled () const;
```

3.20.4 ピッキングフラグ

シーンに仮想的なレイを飛ばして交差するノードを見つける事をピッキングと呼びます。このノードがピッキング可能かどうかを setPickingEnable() メソッドで設定します。この機能は現在実装されていません。

```
void setPickingEnable (bool enable);
```

現在の設定は isPickingEnabled() メソッドで取得できます。

```
bool isPickingEnabled () const;
```

3.20.5 アライメント

ノードは他のノードに対して自動的に軸をそろえる事ができます。これをアライメントと呼びます。アライメントは例えばビルボードやヘッドライト効果を実現するために使用します。この機能は現在実装されていません。

アライメントは setAlignmentTarget() メソッドで設定します。この機能は現在実装されていません。

```
void setAlignment (Node* z_ref, int z_target, Node* y_ref, int y_target);
```

第 1 引数の z_ref には Z 軸のアライメント対象のノードを、z_target にはターゲットを NONE もしくは ORIGIN を設定します。NONE の場合は Z 軸のアライメントはなく第 1 引数は無視されます。ORIGIN の場合は Z 軸のアライメントを第 1 引数のノードに設定します。同様に第 3，第 4 引数は Y 軸に対するアライメントを設定します。X 軸に対する設定はありません。

現在のアライメント設定を取得するには getAlignmentReference() メソッドもしくは getAlignmentTarget() メソッドを使用します。

アライメントターゲットを取得するには getAlignmentTarget() メソッドを使用します。

```
int getAlignmentTarget (int axis) const;
```

引数の axis には Y_AXIS もしくは Z_AXIS を指定します。指定した軸のアライメントターゲットが返ります。NONE はアライメント無し、ORIGIN はその軸はノードにアライメントしています。

アライメントの参照ノードは getAlignmentReference() メソッドで取得します。

```
Node* getAlignmentReference (int axis) const;
```

引数の axis には Y_AXIS もしくは Z_AXIS を指定します。その軸がアライメントしている場合 Node が返ります。アライメントしていない（ターゲットが NONE の）場合は NULL が返ります。

アライメントの設定をした後に align() メソッドを呼ぶと、アライメントがそろいます。

```
void align (Node* reference);
```

引数 reference は設定されていない軸に対するアライメントを使用したい場合に指定します。これは動的にアライメントをそろえるために使用されます。

3.20.6 ノードトランスフォーム

このノードから指定されたノードへの座標変換行列を取得できます。それには getTransformTo() メソッドを使います。

```
bool getTransformTo (Node* target, Transform* transform) const;
```

第 1 引数には変換先のノードを、第 2 引数には結果を書き込む Transform オブジェクトを指定します。戻り値は変換行列が計算できた場合は true, 計算できない場合は false が返ります。

3.21 グループ

Group はシーンを構成するノードの１つで、複数の子のノードを１つにまとめシーンの階層構造を作成します。この階層構造はスキンアニメーションのスケルトンを構成するのにも使われます。Group オブジェクトはインスタンス化した後 World クラスに addChild() します。

```
Group* grp = new Group;
wld->addChild (grp);
```

3.21.1 子ノードの追加

Group に子ノードを追加するには addChild() メソッドを使います。

```
void addChild (Node* child);
```

既に保持していた子ノードの順番はこのメソッドの呼び出しによって変わるかもしれません。既にどこかの Group に属しているノードを追加する事はできません。また World クラスを追加する事はできません。

現在この Group オブジェクトが保持している子ノードの数を取得するには getChildCount() メソッドを使います。

```
int getChildCount () const;
```

現在この Group オブジェクトが保持している子ノードを取得するには getChild() メソッドを使います。

```
Node* getChild (int index) const;
```

引数には 0 から getChildCount()-1 の間の番号を指定します。

現在この Group オブジェクトが保持している子ノードを削除するには removeChild() メソッドを使います。

```
void removeChild (Node* child);
```

引数には削除したい子ノードを指定します。指定された子ノードが存在しない場合単に何もしません。既に保持していた子ノードの順番はこのメソッドの呼び出しによって変わるかもしれません。

3.21.2 ピッキング

ピッキングを行います。現在この機能は未実装です。

3.22 トランスフォーム

Transform はトランスフォームを表す 4x4 行列を抽象化したクラスです。デフォルトは単位行列に設定されています。Transform オブジェクトは原則（ポインターではなく）参照で渡され、ライブラリ内部でコピーされます。

Transform を表す 4x4 行列を取得するには get() メソッドを使います。

```
void get (float* matrix) const;
```

引数の matrix には float16 個分の領域へのポインターを渡します。行列は行優先 (row-major) で格納されます。

$$\begin{pmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \\ 12 & 13 & 14 & 15 \end{pmatrix}$$

明示的に 4x4 行列を設定するには set() メソッドを使います。

```
void set (const float* matrix);
void set (const Transform& transform);
```

引数には 16 個の float(row-major) で行列を指定するか、Transform オブジェクトを指定します。

setIdentity() メソッドはこの 4x4 行列 (M) を単位行列化します。

$$M = I$$

```
void setIdentity ();
```

invert() メソッドはこの 4x4 行列を逆行列化します。逆行列が計算できない場合は例外を発生します。

$$M' = M^{-}1$$

```
void invert ();
```

transopose() メソッドはこの 4x4 行列を転置します。

$$M' = M^t$$

```
void transpose ();
```

postTranslate() メソッドは引数で指定された平行移動を表す 4x4 行列 (T) を作成し、この 4x4 行列 (M) に右から乗算します。

$$M' = MT$$

```
void postTranslate (float tx, float ty, float tz);
```

引数には移動量 (tx,ty,tz) を指定します。

postRotate() メソッドは引数で指定された回転を表す 4x4 行列 (R) を作成し、この 4x4 行列 (M) に右から乗算します。

```
void postRotate (float angle, float ax, float ay, float az);
```

引数の angle には回転角度を degree で、ax,ay,az は回転軸 (ax,zy,az) を指定します。回転軸のベクトルは単位ベクトルである必要はありません。

postRotate() メソッドはクォータニオンで指定された回転を表す 4x4 行列 (R) を作成し、この 4x4 行列 (M) に右から乗算します。

$$M' = MR$$

```
void postRotateQuat (float qx, float qy, float qz, float qw);
```

(qx,qy,qz) がクォータニオンの虚数成分で、qw が実数成分です。

postScale() メソッドは引数で指定された拡大縮小を表す 4x4 行列 (S) を作成し、この 4x4 行列 (M) に右から乗算します。

$$M' = MS$$

```
void postScale (float sx, float sy, float sz);
```

引数には X,Y,Z 軸方向の拡大縮小率を指定します。1.0 が現状維持です。

postMultiply() メソッドは引数で指定された 4x4 行列 (T) を、この 4x4 行列 (M) に右から乗算します。

$$M' = MT$$

```
void postMultiply (const Transform& transform);
```

transform() メソッドは float4 個で表されたベクトル (v) を、この 4x4 行列 (M) を使って変換します。

$$v' = Mv$$

```
void transform (float* vectors) const;
```

引数の vectors は変換結果で上書きされます。

また transform() メソッドは VertexArray オブジェクトの持つデータを一括してこの行列 (M) で変換する事ができます。

```
void transform (VertexArray* in, float* out, bool w) const;
```

第 1 引数 in にはベクトルデータを、第 2 引数 out には結果を書き込む float の領域を、第 3 引数 w には VertexArray が 4 番目のコンポーネントを持たないときの補間方法を bool で指定します。w=true のとき 1 で補完され,w=false の時 0 で補完されます。2,3 番目のコンポーネントが存在しないときは 0 で補完されます。

3.23 バーッテックスアレイ

VertexArray クラスはデータを保持するためのクラスです。データは単なる数値の集合として扱われます。

１頂点は 2,3,4 コンポーネントからなります。１コンポーネントは 1 バイト (char) か 2 バイト (short) です。4 バイト型 (float?) はありません。バーテックスアレイのデータはそのまま参照されるのではなく、最終的なデータは bias と offset で演算したものが最終データとなります。bias と offset は VertexArray クラスの範囲外です。

3.23.1 コンストラクタ

VertexArray クラスは頂点数とコンポーネント数、コンポーネントサイズを指定してインスタンス化します。

```
VertexArray (int num_vertices, int num_components, int component_size);
```

第 1 引数の num_vertices には頂点数を、第 2 引数の num_components にはコンポーネント数を、第 3 引数の component_size にはコンポーネントサイズを指定します。頂点数は 1〜65535 まで対応しています。コンポーネント数は 2,3,4 に対応しています。コンポーネントサイズは 1 か 2 です。ライブラリ内部で確保された領域の値は未定義です。

現在の頂点数、コンポーネント数、コンポーネントタイプ（サイズ）は以下のメソッドで取得できます。

```
int getVertexCount () const;
int getComponentType () const;
int getComponentCount () const;
```

3.23.2 値の設定

作成した VertexArray オブジェクトには set() メソッドを使って値を設定します。

```
void set (int first_vertex, int num_vertices, char* values);
void set (int first_vertex, int num_vertices, short* values);
```

第 1 引数にはデータの開始位置を頂点番号で指定します。第 2 引数にはデータの個数を頂点数で指定します。第 3 引数にはセットしたいデータのポインターを指定します。コンポーネント数に応じて char 型と short 型があります。どちらも signed 型です。値はライブラリ内部に確保したメモリ領域にコピーされます。

現在設定されているデータは get() メソッドを使います。

```
void get (int first_vertex, int num_vertices, char* values) const;
void get (int first_vertex, int num_vertices, short* values) const;
```

第 1 引数にはデータの開始位置を頂点番号で指定します。第 2 引数にはデータの個数を頂点数で指定します。第 3 引数には結果を書き込むメモリ領域へのポインターを渡します。コンポーネント数に応じて char 型と short 型があります。どちらも signed 型です。

```
void get (int first_vertex, int num_vertices, float scale, float* bias, float* values) const;
```

3.24 バーテックスバッファー

VertexBuffer クラスは頂点座標、法線、テクスチャー座標、頂点カラーなどを保持・管理するクラスです。それらのデータを保持するために VertexArray クラスをメンバーに持ちます。通常 Mesh クラスのコンストラクタで渡します。

```
VerteBuffer* vbuf = new VertexBuffer;
Mesh* mesh = new Mesh (vbuf, index, appearance);
```

3.24.1 頂点座標

VertexBuffer に頂点座標をセットするには setPositions() メソッドを使います。

```
void setPositions (VertexArray* positios, float scale, float* bias);
```

第 1 引数には頂点座標として使うデータの入った VertexArray オブジェクトを渡します。コンポーネント数は 3 です。第 2，第 3 引数にはスケール乗数 (scale) とバイアス値 (bias) を渡します。バーテックスアレイ内の値を v とすれば最終的な値は以下の式で計算されます。

$$v' = scale * v + bias$$

頂点座標として使用している VertexArray オブジェクトを取得するには getPositions() メソッドを使います。

```
VertexArray* getPositions (float* scale_bias) const;
```

引数には

3.24.2 法線

VertexBuffer に法線をセットするには setNormals() メソッドを使います。

```
void setNormals (VertexArray* normals);
```

第 1 引数には法線として使うデータの入った VertexArray オブジェクトを渡します。コンポーネント数は 3 です。スケールとバイアスは固定で char と short 型が表現できる全領域が [-1,1] にマップされます。すなわちデータが char 型の場合 $scale = 2/255\, bias = 1/255$ で、short 型の場合は $scale = 2/65535\, bias = 1/65535$ に自動的にセットされます。

法線として使用している VertexArray オブジェクトを取得するには getNormals() メソッドを使います。

```
VertexArray* getNormals () const;
```

3.24.3 テクスチャー座標

VertexBuffer にテクスチャー座標をセットするには setTexCoords() メソッドを使います。

```
void setTexCoords (int index, VertexArray* tex_coords, float scale, float* bias);
```

第 1 引数にはテクスチャー番号、第 2 引数にはテクスチャー座標として使うデータの入った VertexArray オブジェクトを渡します。コンポーネント数は 1 または 2 です。コンポーネントサイズは 1 または 2 です。第 3, 第 4 引数にはスケール乗数 (scale) とバイアス値 (bias) を渡します。計算式は同じです。テクスチャー番号は 0 から始まる整数値で使用中の GPU が対応しているマルチテクスチャーの枚数-1 まで指定できます。

```
VertexArray* getTexCoords (int index, float* scale_bias) const;
```

現時のテクスチャー座標を取得するには getTexCoords() メソッドを使います。

```
VertexArray* getTexCoords (int index, float* scale_bias) const;
```

第 1 引数にはテクスチャー番号を指定します。第 2 引数には float2 個分のメモリ領域を指すポインターを渡します。スケール値とバイアス値がこの順で書き込まれます。

3.24.4 頂点カラー

VertexBuffer に頂点カラーを設定するには setColors() メソッドを使います。

```
void setColors (VertexArray* colors);
```

第 1 引数には法線として使うデータの入った VertexArray オブジェクトを渡します。コンポーネント数は 3 です。コンポーネントサイズは 1 です。スケールとバイアスは固定で char 型が表現できる全領域が [0,1] にマップされます。すなわち $scale = 1/255\, bias = 128/255$ に自動的にセットされます。

現在の頂点カラー配列を取得するには getColors() メソッドを使います。

```
VertexArray* getColors () const;
```

全頂点を同一カラーに設定する場合 setDefaultColor() メソッドが使えます。両方指定した場合は setColors() メソッドで指定した方が優先されます。

```
void setDefaultColor (int rgb);
```

引数の色には rgb フォーマットで指定します。

```
int rgb = (r << 16) | (r << 8) | (b << 0);
```

M3G の仕様では頂点カラーが使われるには Material クラスの頂点カラートラッキングが有効になっている必要があります。ただし現在の実装では頂点カラートラッキングの値にかかわらず、VertexBuffer に頂点カラーがセットされていれば使用されます。

現在のデフォルトの頂点カラーを取得するには getDefaultColor() メソッドを使います。

```
int getDefaultColor () const;
```

3.24.5 インデックスバッファー

IndexBuffer クラスは面を定義するインデックスに共通インターフェースを提供する抽象クラスです。インスタンス化される事はありません。現在では TriangleStripArray クラスがこの IndexBuffer クラスを継承します。

3.24.6 インデックス

インデックスは面を定義する頂点の番号の事です。

インデックスの数は getIndexCount() メソッドで取得します。

```
virtual int getIndexCount () const = 0;
```

インデックスは getIndices() メソッドで取得します。

```
virtual void getIndices (int* indices) = 0;
```

引数の indices にはインデックスの配列を書き込めるだけのメモリ領域へのポインターを指定します。

3.25 Graphics3D

Graphics3D クラスは描画コンテキストを取り扱うシングルトンクラスです。全ての描画は Graphics3D クラスを通して行われます。

典型的には以下の手順で行われます。

```
Graphics3D* g3d = Graphics3D::getInstance();
// ビューポートの設定
World* wld = new World;
// シーンの設定
g3d->render (wld);
```

Graphics3D はシングルトンクラス化されており new 演算子は呼び出してインスタンス化はできません。代わりに getInstance() メソッドを使います。M3G はマルチスレッドセーフではありません。M3G のクラス・メソッドは OpenGL コンテキストが使用可能な状態である事を要求します。OpenGL コンテキストが有効でない環境での利用は未定義です。

Java/MIDP のオリジナルの M3G では描画対象を指定するために render() メソッドの前に Graphics クラスを引数に持つ bindTarget() メソッドを呼び出します。C++ で書かれた Desktop-M3G では Graphics クラスに相当する標準的なクラスが存在しないので bindTarget() および releaseTarget() メソッドを呼ぶ必要はありません。

ただしこのあたりの実装に関しては OpenGL のグラフィックスコンテキストの標準化を見ながら随時変更する予定です。

3.25.1 イミィーディエイトモードとリテインドモード

M3G には大きくイミィーディエイトモード (immediate mode) とリテインドモード (retained mode) の 2 つがあります。イミィーディエイトモードは OpenGL と同じようにライト、カメラ等の設定を 1 つずつ設定していきます。速度的なメリットがないので現在 Desktop-M3G では対応していません。すなわち Graphic3D クラスのメソッドのうち、addLight(), clear(), getCamera(), getDepthRangeFar(), getDepthRangeNear(), getLight, getLightCount(), render(Node, Transform), render(VertexBuffer, IndexBuffer), render(VertexBuffer, IndexBuffer, Appearance, Transform, int), resetLights(), setCamera(), setDepthRagne(), setLight() を呼び出すと NotImplementedException 例外が発生します。リテインドモードは事前にシーングラフを作成し、それを Graphcs3D クラスに渡してレンダリングします。M3G で標準的に使われる手法です。render(World*) メソッドが該当します。

3.25.2 レンダリングターゲット

オリジナルの M3G では bindTarget() メソッドでレンダリングターゲットを指定し、releaseTarget() で解放します。指定できるターゲットは java.awt.Graphics か javax.microedition.lcdui.Graphics ですが、C/C++ ではこれらに相当する標準的な仕組みがないので呼び出す必要はありません。bindTarget()、releaseTarget() ともに呼び出すと例外を発生します。最終的に Desktop-M3G でレンダリングターゲットをどうするかは現在未定です。

bindTarget() で渡すレンダリングヒント（ANTIALIAS, DITHER, TRUE_COLOR, OVERWRITE) の扱いはは未定です。

3.25.3 ビューポートの設定

ビューポートの設定は setViewport() メソッドで行います。ビューポートはウィンドウ内の矩形領域の事で、NDC 空間 (-1,-1)〜(1,1) が伸張されここに表示されます。

```
void setViewport (int x, int y, int width, int height);
```

引数にはビューポートの左下の座標 (x,y) とサイズ (width,height) をピクセル数で指定します。

3.25.4 実装依存のプロパティ

いくつかの機能は実装または使用中の GPU によって制限されます。getProperties() メソッドは実装依存のプロパティ情報を取得します。プロパティは name(const char*),value(int) のペアで表されます。

```
std::vector<Property> properties = g3d->getProperty();
```

プロパティの名前を取得するには name() メソッドを、値を取得するには value() メソッドを使います。

3.26 トライアングルストリップ

TriangleStripArray クラスはトライアングルストリップ形式で面を定義します。トライアングルストリップ形式の解説は OpenGL などの解説書を参照してください。

3.26.1 コンストラクタ

TriangleStripArray クラスのコンストラクタには 2 つの形式があります。1 つ目はインデックスの配列とストリップ長を指定して作成する方法です。

```
TriangleStripArray (int* indices, int num_strips, int* strips);
```

第 1 引数の indices にはインデックスの配列を指定ます。インデックスは頂点番号を示す整数値です。TriangleStripArray クラスは複数のトライアングルストリップを同時に扱います。第 2, 第 3 引数にはストリップ長の配列とその配列長を指定します。TriangleStripArray クラスは指定されたストリップ長毎にインデックスの配列を切り出し、ストリップとします。

例えば、

```
int indices{} = {3,4,5,10,11,12,13};
strips{}      = {3,4}
TriangleStripArray* tris = new TriangleStripArray (indices, 2, strips);
```

はストリップ 1 (頂点番号 3,4,5) とストリップ 2（頂点番号 10,11,12,13）を作成します。

2 つめの形式は最初のインデックス番号とストリップ長を指定して作成する方法です。

```
TriangleStripArray (int first_index, int num_strips, int* strips);
```

第 1 引数の first_vertex には最初の頂点番号を指定します。第 2, 第 3 引数には同様にストリップ長の配列とその配列長を指定します。TriangleStripArray クラスは指定されたストリップ長毎に first_index から順番にインクリメントしながらインデックス配列を作成しストリップとします。

例えば、

```
strips{}      = {3,4}
TriangleStripArray* tris = new TriangleStripArray (10, 2, strips);
```

はストリップ 1（頂点番号 10,11,12）とストリップ 2（頂点番号 13,14,15,16）を作成します。

3.27 ワールド

World クラスはシーンを構成する最上位のコンテナクラスです。World は他のノードの子になる事ができません。シーンは World オブジェクトをルートとする木構造です。

World クラスを Graphics3D クラスの render() メソッドに渡す事で、そのシーンがレンダリングされます。

```
World* wld = new World;
g3d->render (wld);
```

3.27.1 カメラの設定

レンダリングするためには setActiveCamera() メソッドでカメラを１つアクティブ化する必要があります。

```
void setActiveCamera (Camera* camera);
```

カメラは事前にシーンノードとして World に挿入されている必要があります。詳細は Camera クラスの説明をご覧ください。

3.27.2 背景の設定

背景の設定は Background クラスが行います。デフォルト以外の背景を使う場合は setBackground() メソッドで設定します。詳細は Background クラスの説明をご覧ください。

```
void setBackground (Background* background);
```

3.27.3 ノードの追加

シーンノードの追加には addChild() メソッドを使います。ただし addChild() は World クラスの基底クラスである Group クラスで定義されたメソッドです。詳細は Group クラスの説明をご覧ください。

```
void addChild (Node* child);
```

3.28 ローダー

Desktop-M3G ではオリジナルの M3G で定義されたファイルデータをそのまま利用できます。主要な商用 3D グラフィクスソフトは全て M3G 形式で出力できるエクスポーターが存在します。フリーのものでは Blender が可能です。Loader クラスはインスタンス化できません。

3.28.1 ロード

M3G ファイルのロードには static 関数の load() メソッドを使います。

```
std::vector<Objecct*> scene = Loader:: loade ("scene.m3g");
```

引数にはローカルにある M3G ファイルを指定します。メモリ上に展開した M3G ファイルからロードするメソッドは現在未実装です。戻り値は Object のポインターへの std::vector 配列です。Object の順番は規定されていません。

この Object の配列から何らかの方法で” ルート” オブジェクトを見つけます。ルートとなるオブジェクトは M3G ファイルがシーン全体を保持している場合は World クラスであり、１体のモデルデータの場合は Mesh クラス（もしくはその派生クラス）です。Object の配列からルートオブジェクトを見つけるにはいくつかの方法があります。下記は World クラスを見つける例です。Mesh クラスも同じです。

- 見つかるまで順番に World クラスに dynamic_cast する
- World クラスに付けた ID を見つける

ID を付ける方法は、3D グラフィックソフト側で World クラスに固有の ID が付加されている事が条件です。

4　メッシュ

この章ではメッシュの概念と利用方法を解説します。
この章はまだ書かれていません。

5　アニメーション

この章ではアニメーションの概念と利用方法を解説します。
この章はまだ書かれていません。

6 付録

付録では Desktop-M3G の内部実装の補足説明を記します。

6.1 レンダリングパス

Desktop-M3G の実装ではシーンのレンダリングのために 3 回 render() 関数を呼びます。Graphics3D::render() 関数の中で自動的に行われるので、普通はこれを意識する必要はありません。それぞれのレンダリングパスは引数の pass=0,1,2 で区別します。

6.1.1 パス 0

カメラの設定と背景のクリアを行います。

6.1.2 パス 1

ライトの設定を行います。

6.1.3 パス 2

その他の描画対象のオブジェクトをレンダリングします。