M3G チュートリアル

バージョン 1.1 対応
2010.11.11 版

上田 孝 (tueda@wolf.dog.cx)

目次

第1章　概要

M3G とは

M3G は携帯電話向けに策定された 3D グラフィックスを記述するための API です。Java 言語のモバイル用機能セット(J2ME)で利用可能です。M3G-1.0 が 2005 年に JSR-184 として Java community process(JCP)によって策定されました。2007 年に現在の最新バージョンであるバージョン 1.1 にアップデートされています。同種の API として OpenGL(や DirectX)があるにもかかわらずなぜ JCP は新たな API を定義したのでしょうか。Java 言語からは M3G を使わなくても OpenGL Java bindings for OpenGL(JOGL)を利用して OpenGL を使うことができます。M3G には OpenGL と比べて以下の利点があります。

- シンプル
- シーングラフ
- アニメーション
- スキニング
- ファイルフォーマット

OpenGL のシーン記述が記述順に依存する手続き型(ステートマシーン)だったに対して、M3G は記述順番に依存しないオブジェクト指向なシーングラフを基本としています。M3G は OpenGL では曖昧だった「シーンの記述」と「レンダリング」を完全に分離しました。ユーザーは順番に依存せず自由にシーンを記述し、その後一括して高速にレンダリングすることができます。同様に「アニメーション」も分離しています。また OpenGL にはない機能としてキーフレームアニメーションとキャラクターアニメーションのためのスキニングおよび独自のファイルフォーマットが定義されました。従来は OpenGL を使ってゲームを作成しようと考えた時に、このあたりの機能は自作するか公開されている別のライブラリを利用するしかありませんでした。本職のゲームプログラマーはともかくアマチュアがこれらの機能を自分で実装するのは難しいものがあります。公開されているライブラリを利用するにしても用途に適応したライブラリを探し、その使い方を学習するコストは無視できません。M3G ではこららの機能はすべて標準で提供されます。従って一度学習すれば普遍的に利用可能です。M3G は 3D ゲームを開発する上で PlayStation(R)2 のゲームと同程度の表現力を有しています。なお M3G ライブラリは GPU にアクセスするための下位層として多くの場合 OpenGL ES を使用します。OpenGL をハードウェアに命令する API として内部で使用し、M3G を人間にやさしい API としてユーザーに提供します。またこれまで OpenGL には DirectX における.x ファイルのような定番のファイルフォーマットが存在しませんでした。M3G では.m3g ファイルという形式を定義しています。これはすでに多くのツールでサポートされておりすぐに利用可能です。

M3G のソフトウェアスタック

M3G ライブラリの構成は図 1 のようになります。現在ある M3G ライブラリは GPU にアクセスするために OpenGL ES を利用しています。下位層は必ずしも OpenGL でなくてもよく DirectX や GPU メーカー独自の API でも動作します。M3G ライブラリは Java もしくは C++で実装され提供されます。

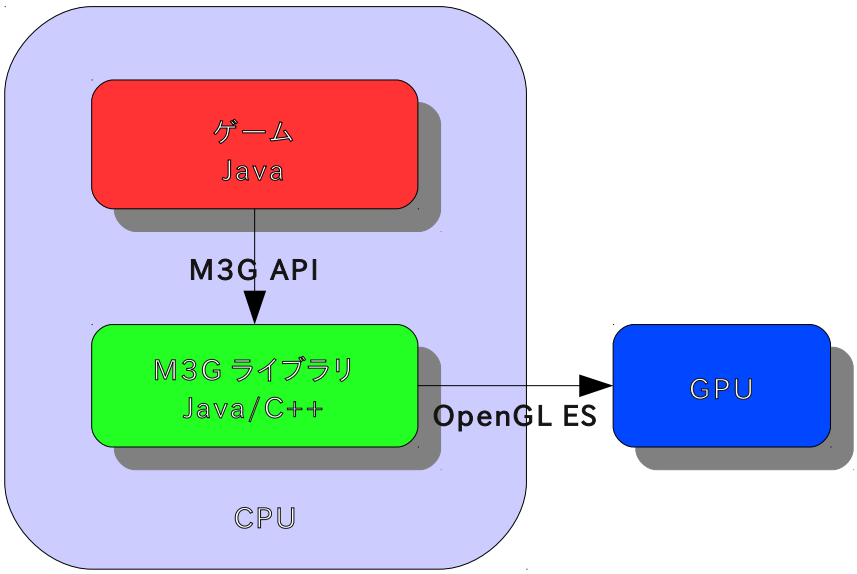


図 1:M3G の構成

M3G を構成する 30 クラス

M3G は図 2 の全 30 のクラスからなります。こららのクラスは大きく(1)OpenGL のラッパー、(2)シーングラフ、(3)アニメーション、(4)ファイルローダーにわける事ができます。

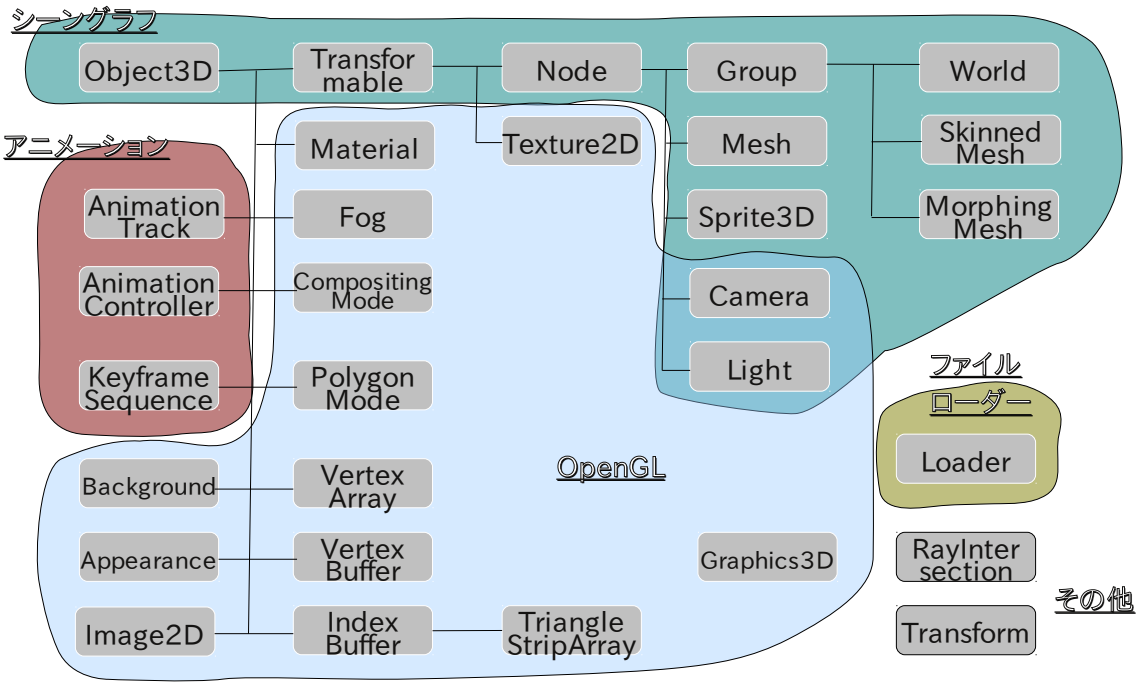


図 2:M3G を構成する 30 クラス

OpenGL のラッパーに関する解説は主に第 5,6 章を、シーングラフの解説は第 4 章を、アニメーションに関する解説は第 7 章を、ファイルフォーマットの解説は第 8 章を参照して下さい。

Desktop-M3G とは

Desktop-M3G は M3G API のオープンソースな実装です。オリジナルの M3G が Java 言語のモバイル用プラットフォーム(J2ME)をターゲットにしているのに対して、Desktop-M3G はデスクトップ用 Linux で C++言語をターゲットにしています。Desktop-M3G の実装は C++で行われており、ユーザーは C++から使用するだけでなくバインダーを使って Ruby や Java(J2SE)からも使用することが可能です。オリジナルと比べて Java と C++という言語の違いから来る若干の API の違いがあります。またオリジナルがモバイル向けの比較的非力な CPU･GPU を想定しているのに対して高性能の CPU･GPU の使用を想定しています。貧弱な環境向けの機能、例えばイミィーディエイトモードは意図的に実装していません。いくつかのデフォルトパラメーターをデスクトップ用途に適した数値に上方修正しています。詳しくは付録 A を参照して下さい。開発は主に Google Project Hosting 上の Desktop-M3G のページで行っています。

- http://code.google.com/p/desktop-m3g/

また関連プロジェクトとして Ruby-M3G と Java-M3G があります。

- http://code.google.com/p/ruby-m3g/
- http://code.google.com/p/java-m3g/

2010 年 11 月の段階でバージョンは 0.3.1 です。Desktop-M3G を動作させるには下記の環境が必要です。

- Linux 系 OS (作者は Ubuntu 上で開発・テストしています)
- OpenGL 1.3 以上
- libGL, libpng, libjpeg, libz などのライブラリ

C++の Desktop-M3G を多言語から使うためのバインダーである Ruby-M3G と Java-M3G については上記の開発用ウェブページを参照して下さい。

第2章　描画の基本

最小の描画コード

この章では M3G API を使った 3D シーンの描画方法の基本的な手順を解説します。ある程度 3D グラフィックスの基本を理解している必要があります。すなわち「カメラ」「シーン」「ポリゴン」「ビューポート」などの概念を理解している必要があります。M3G では次の手順でシーンを描画します。

1. シーングラフの作成
2. Graphics3D オブジェクトの取得
3. シーンのレンダリング

順を追って説明していきます。まず最初に描画したい 3D シーンを作成します。シーングラフは描画したいシーンを表す木構造のグラフです。シーングラフの頂点ノードは必ず World クラスです。すなわち 1 つの World オブジェクトが 1 つのシーンを表します。独立した複数のシーンが同時に存在可能で、シーンは互いに影響を及ぼしません。

```
World* wld = new World;
```

インスタンス化した直後の World オブジェクトはすでに有効なシーンです。当然シーンには何も含みません。

次に Graphics3D オブジェクトのインスタンスを取得します。Graphics3D クラスは描画デバイス(GPU)を抽象化したようなクラスで、シーンを描画するにはこのクラスのオブジェクトが必要です。Graphics3D クラスはシングルトン化されているので getInstance() 関数を使ってただ 1 つのインスタンスを取得します。

```
Graphic3D* g3d = Graphics3D:: getInstance ();
```

最後にこの Graphics3D クラスの render()関数を World オブジェクトを引数にして呼び出すと、そのシーンが描画され画面に表示されます。

```
g3d->render (wld);
```

この 3 行が M3G の描画に必要な最小限のコードです。ただしこの 3 行は完全に正しいコードですがまだ 3D の描画に必要なビューポートとカメラの設定がないので何も表示されず、呼び出すと”Active camera is NULL.”というメッセージとともに IllegalStateException 例外が発生します。このように M3G では何らかの異常を検出すると例外を発生します。発生する例外の種類については第 9 章を参照して下さい。

```
$ ./a.out
terminate called after throwing an instance of 'm3g::IllegalStateException'
  what():  Graphics3D.cpp:render Active camera is NULL.
```

次の節ではビューポートを設定し、シーンにカメラとメッシュを追加して意味のあるシーンが描画されるようにします。

板ポリゴンの表示

意味のあるシーンを描画するためには前節の 3 行に加えて以下の手順が必要です。

1. ビューポートの設定
2. カメラの追加
3. メッシュの追加

この節では順番に 3 つの手順を追加し、画面に図 3 の 4 角形の板ポリゴンを表示します。

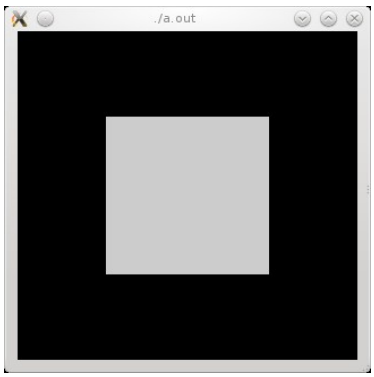


図 3:板ポリゴンの表示

ビューポートの設定

まず初めにビューポートの設定を行います。ビューポートはレンダリングした結果をウィンドウに張り付けるときの表示領域です。通常はウィンドウ全体に設定します。ビューポートの設定は Graphics3D クラスの setViewport()関数で行います。ウィンドウサイズがピクセル数で(width,height)としてウィンドウ全体に表示するためには、以下の設定にします。

```
g3d->setViewport (0, 0, width, height);
```

ウィンドウは左上が(0,0)で右下が(width-1, height-1)です。

カメラの追加

次にシーンにカメラを追加します。シーンには複数のカメラを挿入することができ、そのうちの 1 つをアクティブカメラに指定する必要があります。このアクティブカメラから見たシーンが最終的にレンダリングされて表示されます。次のコードは Camera クラスをインスタンス化してシーングラフに追加し、World クラスの setActiveCamera()関数を使ってアクティブカメラに設定しています。

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

デフォルトのカメラは座標位置(0,0,0)で z 軸負の方向を向いています。アップベクトルは(0,1,0)です。射影行列は単位行列です。このままでも表示出来ますがせっかくなので標準的なカメラ設定に変更します。ここでは透視投影カメラで視野角 45°、アスペクト比 width/height 、ニアクリップ 0.1、ファークリップ 100 に設定します。最後にカメラの位置を Z 軸方向に+5 だけ移動します。カメラはデフォルトで Z 軸負の方向を向いているので、これで原点付近(Z=0)を見ているカメラが作成できます。

```
cam->setPerspective (45, (float)width/height, 0.1, 100);
cam->translate (0,0,5);
```

なおシーングラフは記述順に依存しません。render()関数に渡したときの状態で描画されます。

メッシュの追加

最後に板ポリゴンを表すシーンノードとして Mesh オブジェクトを作成します。Mesh ノードはおもに頂点情報を保持する VertexBuffer クラスと、その見え方を定義する Appearance クラスの 2 つに分けられます。これらのオブジェクトの作成は若干面倒なのでここでは簡単に解説するにとどめ、詳しい解説は第 5 章と第 6 章に譲ります。

板ポリゴンは 4 つの頂点(1,-1,0), (1,1,0), (-1,-1,0), (-1,1,0)をこの順で結んだトライアングルストリップで作成します。ストリップ長は 4 で 2 つのトライアングルからなります。アピアランスはデフォルトをそのまま用います。初めに頂点座標を保持する VertexArray オブジェクトを作成します。VertexArray は引数に頂点数=4、1 頂点あたりのコンポーネント数=3、1 コンポーネントのバイト数=4 を指定してインスタンス化し、値を set()関数を使ってセットします。

```
VertexArray* positions         = new VertexArray (4, 3, 4);
float        positions_value[] = {1,-1,0, 1,1,0, -1,-1,0, -1,1,0};
positions->set (0, 4, positions_value);
```

作成した VertexArray はただの float 型のデータの配列にすぎないので、これを座標位置として VertexBuffer オブジェクトに setPositions()関数を使ってセットします。この時 M3G の仕様上必ずスケール値とバイアス値を同時に指定します。最終的な値は VertexArray にセットされた値にこのスケール、バイアスをかけた値です。詳しくは第 5 章を参照してください。ここではセットした値をそのまま使用するので scale=1, bais={0,0,0}を指定しています。

```
VertexBuffer* vertices = new VertexBuffer;
float scale   = 1;
float bias[3] = {0,0,0};
vertices->setPositions (positions, scale, bias);
```

次に頂点を結んでトライアングルストリップを形成します。ここではトライアングルストリップは 1 本で長さ 4、インデックスは{0,1,2,3}を指定しています。

```
int strips[1] = {4};
int indices[4] = {0,1,2,3};
TriangleStripArray* tris = new TriangleStripArray (indices, 1, strips);
```

TriangleStrtipArray は複数のストリップを一度に指定できますが、ここでは 1 本しか使用していません。

そして見え方を定義する Appearance オブジェクトを作成します。

```
Appearance* app = new Appearance;
```

ここでは全てフォルトを使用するので new でインスタンス化したものをそのまま使用します。

以上で VertexBuffer オブジェクト, TriangleStripArray オブジェクト, Appearance オブジェクトの 3 オブジェクトがインスタンス化できたので、Mesh オブジェクトを作成しシーングラフに追加します。カメラが座標位置(0,0,5)から原点付近を見ているので、このまま特に移動も回転も行いません。

```
Mesh* mesh = new Mesh (vertices, tris, app);
wld->addChild (mesh);
```

以上で全ての準備は完了です。World オブジェクトを引数にして Graphics3D クラスの render()関数を呼べば 4 角形の板ポリゴンが表示されます。手順をもう 1 度まとめると下記になります。

```
Graphic3D* g3d = Graphics3D:: getInstance ();
// ビューポートの設定
World* wld = new World;
// Camera オブジェクトを追加
// Mesh オブジェクトを追加
g3d->render (wld);
```

全ての設定はシーンが render()関数によって描画されるまでに完了している必要があります。設定は呼び出し順に依存しないので順不同で行えます。このシンプルな板ポリゴンの表示の完全なサンプルが Desktop-M3G のパッケージには含まれています。${DESKTOP-M3G}/sample/test-01-simple を参考にして下さい。

補足: OpenGL の方が簡単のような?

正直このサンプルのようにトライアングルを 1、2 つ描くだけなら OpenGL を使った方が簡単に書けます。これは M3G API がゲームのような数千ポリゴンからなるメッシュをさらに 10 体同時にレンダリングするような状況を想定しているからです。そのような状況下では M3G の方が扱いやすさ、速度共に優れています。

モデルデータのロード

前節までのようにプログラマーがコード上で 0 からメッシュを作りシーングラフを構築してもいいのですが、一般的にはゲーム開発においてモデルデータやシーンを作るのはデザイナーもしくはレベルデザイナーの領分でありプログラマーがプログラムするのものではありません。M3G では M3G(JSR-184)形式と呼ばれるファイルフォーマットを定義しています。このフォーマットは MAYA や MAX などの主要 DCC ツールですでにサポートされています。M3G 形式のファイルは Loader クラスのスタティック関数 load()を使ってロードします。load()関数の戻り値は Object3D(ほぼすべてのクラスの基底クラス)オブジェクトのポインターの配列で、そこから欲しいクラスのインスタンスを探して使用します。シーンをロードしたのであれば World オブジェクトを探すことになります。ここでは例として MilkSphape3D で作ったシーンファイル”mokujin-kun-1.m3g”をロードして表示するコードを示します。

```
vector<Object3D*> objs = Loader:: load (”mokujin-kun-1.m3g”);
for (int i = 0; i < (int)objs.size(); i++) {
    wld = dynamic_cast<World*>(objs[i]);
    if (wld) {
        cout << ”Find world !¥n”;
        break;
    }
}
```

このようにして作成したシーングラフ(World オブジェクト)は前節でプログラマーが 1 つ 1 つコード上で作成したものと等価です。レンダリングは同様に Graphics3D クラスの render()関数にそのまま渡します。

```
g3d->render (wld);
```

このように M3G では DCC ツールからエクスポートしたシーンを非常に簡単にレンダリングすることができます。API の標準のみでツールからエクスポートしたシーンをレンダリングできるのが特徴の 1 つです。図 4 は MilkShape3D で作成した”木人君 1 号”を Loader クラスを使ってロードして描画した例です。

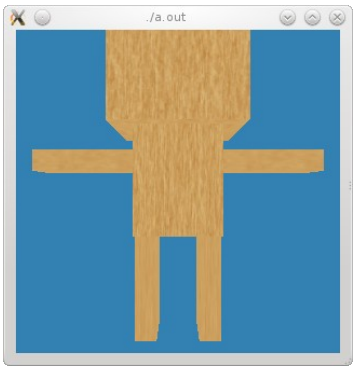


図 4:木人君 1 号

Loader クラスの詳しい使い方は第 8 章を参照して下さい。

第3章　基本コンセプト

この章では M3G API を使った描画の基本コンセプトを解説します。一般的な 3D グラフィックスの知識があることを前提としています。オリジナルの M3G にはシーンノードを 1 つ 1 つ渡してレンダリングするイミーディエイト(immediate mode)とシーン全体を一括してレンダリングするリテインドモード(retained mode)があります。Desktop-M3G ではリテインドモードのみをサポートし、イミーディエイトモードに関する関数呼び出しはすべて NotImplementedException 例外を発生します。このチュートリアルも全てリテインドモードを前提に書かれています。

M3G の描画では以降で解説される「Graphics3D」「レンダリング」「シーングラフ」「アニメーション」「アライメント」という 5 つの概念を知っておく必要があります。この章ではさらにほぼ全てのクラスの基底となる Object3D クラスの解説と一般的な Image2D クラス、Transformable クラスの解説を行います。また一時的なパラメーターの受け渡しに使われる RayIntersection クラス, Transform クラスの解説を行います。これらのクラスの解説は最初に読むときは飛ばしても構いません。

Graphics3D

Graphics3D クラスは描画デバイス(GPU)を抽象化したクラスで、全ての描画はこのクラスが行います。Graphic3D はシングルトン化されておりただ 1 つのインスタンスをスタティックな getInstance()関数で取得します。

```
static Graphics3D* getInstance ();
```

戻り値は Graphics3D オブジェクトのポインターです。このポインターを delete してはいけません。

描画した 3D シーンはウィンドウのビューポートに貼り付けられます。ビューポートの設定は setViewport()関数を使用します。ビューポートの設定は必須です。

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

引数の x と y は 0 以上の整数値、width と height は 1 以上の整数値で指定します。最大のビューポートサイズは使用中の GPU によって制限される可能性があります。通常はウィンドウ全体をビューポートにするのでウィンドウサイズを(width, height)とすればビューポートは(0,0,width,height)を指定します。

現在設定されているビューポートを取得するには getViewportX(), getViewportY(),

getViewportWidth(), getViewportHeight()関数を使用します。

```
int getViewportHeight () const;
int getViewportWidth () const;
int getViewportX () const;
int getViewportY () const;
```

戻り値は現在のビューポートの(x,y,width,height)です。

リテインドモードでのシーンのレンダリングには render()関数を使用します。

```
void render (World* world) const;
```

引数は描画したい World オブジェクトへのポインターです。この関数を呼ぶと指定のシーンをレンダリングする OpenGL コマンドが一斉に GPU に送られます。通常はかなり重い処理になります。途中で何らかのエラーが発生した場合は例外が発生するので try～catch で囲むなりして適切に対処してください。この関数が例外を発生せずに返ってきた場合レンダリングは成功しています。

いくつかのパラメーターは使用中の描画デバイス(GPU)の能力に依存します。例えばテクスチャーユニットの数や一度に使用できるライトの数などです。これらの制限されたプロパティの値を取得するには getProperties()関数を使用します。プロパティは名前と制限値のペアです。

```
std::map<const char*, int> getProperties () const;
```

戻り値がプロパティの std::map です。現在の Desktop-M3G の実装では適当にそれらしい値を返しているだけで、使用中の GPU の能力を反映していません。現在の定義されているプロパティ値は表 1 のとおりです。

プロパティ名	Desktop-M3G の返す値	解説
supportAntialiasing	1	アンチエイリアスをサポートしている場合 1、そうでない場合 0
supportTrueColor	1	32bit カラーをサポートしている場合 1、そうでない場合 0
supportDithering	1	ディザー処理をサポートしている場合 1、そうでない場合 0
supportMipmapping	1	ミップマップをサポートしている場合 1、そうでない場合 0
supportPerspectiveCorrection	1	テクスチャーの透視変換補正をサポートしている場合 1、そうでない場合 0
supportLocalCameraLighting	1	ローカルカメラライティングをサポートしている場合 1、そうでない場合 0
maxLights	8	一度に使用可能なライトの個数
maxViewportWith	2048	ビューポートの幅の最大値
maxViewportHeight	2048	ビューポートの高さの最大値
maxViewportDimension	2048	ビューポートの幅と高さの最大値の最小値
maxTextureDimension	2048	テクスチャーサイズの幅と高さの最大値の最小値
maxSpriteCropDimension	2048	スプライトのクロップ領域のサイズの幅と高さの最大値の最小値
maxTransformsPerVertex	65535	1 頂点あたりに関連付けられるボーンの数
numTextureUnits	4	同時に使用できるテクスチャーの枚数

表 1:プロパティー覧

以下の関数はイミーディエイトモードで使用しますが Desktop-M3G ではイミーディエイトモードには対応していません。今後実装する予定はありません。これらの関数の呼び出しはすべて NotImplementedException 例外が発生します。

```
int addLight (Light* light, const Transform& transform);
```

```
void clear (Background* background);
Camera* getCamera (Transform* transform);
float getDepthRangeFar () const;
float getDepthRangeNear () const;
int getHints () const;
Light* getLight (int index, Transform* transform) const;
int getLightCount () const;
void* getTarget () const;
bool isDepthBufferEnabled () const;
void render (Node* node, const Transform& transform) const;
void render (VertexBuffer* vertices, IndexBuffer* triangles, Appearance* apperance,
                    Transform& transform, int scope=-1) const;
void resetLights ();
void setCamera (Camera* camera, const Transform& transform);
void setDepthRange (float near, float far);
void setLight (int index, Light* light, const Transform& transform);
```

レンダリングターゲットのバインド

オリジナルの M3G では Grapchis3D クラスはレンダリングターゲットをバインド/アンバインドする関数として bindTarget()/releaseTarget()を定義しています。この関数はレンダリングターゲットを指定するとともに OpenGL コンテキストを使用可能にします。この機能は J2ME に依存しておりデスクトップ用の OpenGL と C++では類似の機能を実装する方法がありません。現在の Desktop-M3G の実装では bindTarget()/releaseTarget()の両関数は実装されていません。レンダリングターゲットはフレームバッファー固定で、OpenGL コンテキストの作成には別途 glut などを用います。

```
void bindTarget (void* target, bool depth_buffer_enabled=true, int hints=0);
void releaseTarget ();
```

シーングラフ

前節で見たように Graphics3D オブジェクトの render()関数に World オブジェクトを渡すことでレンダリングを行います。M3G では「シーンの記述」と「レンダリング」は完全に分離しています。シーンは render()関数に渡されるまで一切描画されません。シーングラフは World クラスを頂点とするノードの木構造です。シーング

ラフの一例を図 5 に示します。

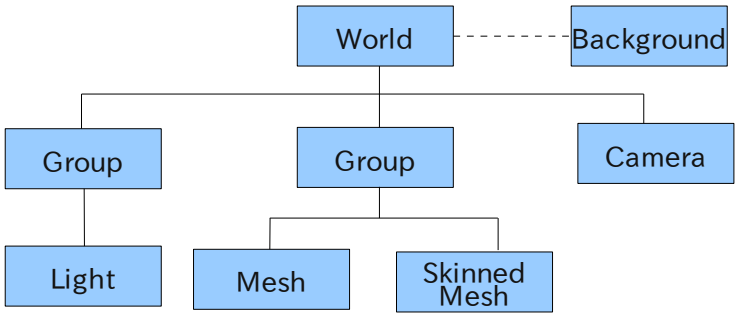


図 5:シーングラフの一例

現在定義されているシーンノードは Camera, Light, Group, Mesh, SkinnedMesh, MorphingMesh, Sprite3D, World の 8 種類です。Group クラスおよび World クラスのみが子ノードを保有でき階層構造を構成します。M3G ではこれらのノードを使って描画したい 3D シーンを構築します。シーンノードに共通する機能はこの後に続く節を、特定のノードに固有な機能は第 4 章シーングラフを参照して下さい。また World オブジェクトには背景として Background オブジェクト(シーンノードではない)を接続できます。

アニメーション

「シーンの記述」と「レンダリング」が分離していたように「アニメーション」も分離されています。シーングラフ全体をアニメートさせるには World クラスの animate()関数を使用します。

```
wld->animate (wold_time);
```

この関数を呼び出すと World オブジェクト以下の全ノードとその構成要素は(アニメーションが設定されていれ

ば)world_time に相当する時刻の値に変更されます。時刻を段階的に変化させることでなめらかなアニメーションが表示されます。全てのアニメーション可能なプロパティの一覧は第 7 章アニメーションにあります。

アライメント

アライメントはあるノードを別のノードに自動的に整列させる機能です。例えば 3D でビルボードと呼ばれる常にカメラの方向を向いたオブジェクトの実装に使われます。「レンダリング」や「アニメーション」が分離していたように「アライメント」も分離しています。ユーザーは World クラスの align()関数を呼びシーングラフおよびその構成ノードを整列させます。アライメントがまったく設定されていなければ align()関数を呼ぶ必要はありません。

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

この関数を呼び出すとシーンの全ノードはあらかじめ設定されているアライメント対象か、もしくは引数 reference で指定されたノードに整列されます。詳しくは第 4 章を参照して下さい。

Object3D

Object3D は全ての基本となるクラスで、特殊な 4 クラス(Graphics3D, Loader, RayIntersection, Transform)を除く全てのクラスの基底となっています。このクラスは M3G オブジェクトに共通の機能を提供します。

オブジェクトには setUserID()関数を使って任意のユーザー ID を付与できます。

```
void setUserID (int user_id);
```

引数 user_id はユーザー ID を表す int 型の整数値です。ユーザー ID の利用方法はユーザーに一任されています。レンダリングには一切影響しません。find()関数を使って特定のユーザー ID を持つオブジェクトを検索可能です。デフォルトは 0 です。

現在設定されているユーザー ID を取得するには getUserID()関数を使用します。

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

戻り値はユーザー ID です。

Object3D には setUserObject()関数を使って任意のデータを付与できます。

```
void setUserObject (void* user_object);
```

引数の user_object には任意のデータへのポインターを渡します。このポインターが指す先のメモリ領域のサイズおよび使い方は M3G ライブラリは一切関与しません。通常 M3G ではコピー渡しが原則ですが、ユーザーオブジェクトに限りポインターを保有するのみとなります。デフォルトは NULL です。

このポインターは後で getUserObject()関数で取得できます。

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

戻り値は setUserObject()で設定したユーザーオブジェクトです。

特定のユーザー ID を持つオブジェクトは find()関数を使って検索できます。このオブジェクトおよび、そこから到達できる Object3D オブジェクト(例えばシーングラフの子ノード、スキンメッシュのスケルトン、アピアランスの構成要素など)が検索対象になります。

```
virtual Object3D* find (int user_id) const;
```

引数の user_id には検索したいユーザー ID を指定します。このオブジェクトとそこから到達できる Object3D オブジェクトを検索し、もし一致すればそのオブジェクトへのポインターを返します。同じユーザー ID を持つオブジェクトが複数ある場合は、そのうちのどれか 1 つが返ります。どれが返るかは不定です。

この Object3D オブジェクトに関連付けられているオブジェクトを取得するには getReferences()関数を使用します。

```
virtual int getReferences (Object3D** references) const;
```

引数 references は結果を書き込む Object3D のポインターの配列です。結果を書き込むのに十分な大きさを確保してある必要があります。引数が NULL の場合は書き込みません。戻り値はオブジェクトの個数です。通常この関数は 2 回呼ばれます。1 回目は引数を NULL にして個数を取得してメモリを確保し、2 回目で確保した配列へ結果を書き込みます。

Object3D とその派生クラスは、コピーコンストラクターと代入演算子がプライベート宣言されているのでコピーする事ができません。代わりに dupliate()関数を呼び出して複製します。複製には必ず duplicate()関数を使用します。この関数はメンバー変数の単なるコピーとは少し異なる動作を行います。

```
virtual Object3D* duplicate () const;
```

戻り値は複製された Object3D オブジェクトです。複製元のオブジェクトは一切変更されません。複製のルールは以下のとおりです。

1. 全てのメンバー変数をコピーする。
2. Node の親ノードは NULL にする。
3. Group の全ての子ノードを duplicate()する。
4. SkinnedMesh のスケルトンノードを duplicate()する。

原則として複製元の M3G オブジェクトから参照されている M3G オブジェクトはコピーされません。従って複製元と複製先の 2 つのオブジェクトが同時に参照する事になります。例えば Mesh オブジェクトを複製しても、それが持つ VertexBuffer オブジェクトと Appearance オブジェクトは複製されません。シーングラフの子ノードに限り再帰的に複製されます。これには Group の子ノードのみならず SkinnedMesh のスケルトン階層も含まれます。なお GPU 側にデータを保存する VertexArray クラスの場合、GPU 上のデータ(vertex buffer object)も複製され、複製後は 2 つのまったく別のデータとして扱われます。ユーザーオブジェクトポインターで指し示している先はコピーされません。

アニメーショントラックを追加するには addAnimationTrack()関数を使用します。

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

引数 animation_track は追加したいアニメーショントラックです。アニメーショントラックとはアニメーションのもっとも基本となる単位です。アニメーションの追加・削除はこのトラック単位で行います。アニメーションの詳細については第 7 章を参照して下さい。

アニメーショントラックを取り除くには removeAnimationTrack()関数を使用します。

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

引数 animation_track はアニメーショントラックです。引数が NULL もしくはこのオブジェクトが保有していないアニメーショントラックの場合は安全に無視されます。

現在設定されているアニメーショントラックの数を取得するには getAnimationTrackCount()関数を使用します。

```
int getAnimationTrackCount () const;
```

戻り値は現在の付加されているアニメーショントラックの総数です。

アニメーショントラックを取得するには getAnimationTrack()関数を使用します。

```
AnimationTrack* getAnimationTrack (int index) const;
```

引数 index はアニメーショントラックのインデックスを指定します。これは 0 から getAnimationTrackCount()-1 までが有効です。戻り値は対応するアニメーショントラックです。インデックスと AnimationTrack オブジェクトの対応付けは変わる可能性があります。

Transformable

座標変換可能なクラスはすべて Transformable クラスを継承しています。具体的には全てのシーンノードの基底クラスである Node クラスと、Texture2D クラスの 2 つが継承しています。

M3G では座標変換を(1)平行移動要素 T(tx,ty,tz), (2)回転要素 R(angle, axis(ax,ay,az)), (3)スケール要素 S(sx,sy,sz), (4)任意変換の M(4x4 行列)の重ね合わせで表現します。元の座標を V(x,y,z,w), 変換後の座標を V'(x',y',z',w')とすると下記の式で計算されます。

$$V' = TRSM\ V$$

すなわちまず最初に任意の 4x4 行列要素 M で変換され、次にスケール要素 S で変換され、回転要素 R で変換され、最後に平行移動要素 T で変換されます。TRSM の 4 要素はそれぞれ別個に管理され、変換時に順番に適応されます。TRSM の順で適応されるので必ず「回転してから移動」になります。通常これが問題になることはありません。なお回転は内部的にはクォータニオンで管理していますが、ユーザーが気にする必要はありません。

補足: 座標変換の表現方法

座標変換をどのように表現するかはライブラリ・アプリケーションによって異なります。TRS(とM)を別個に管理しS,R,Tの順番でかけるM3Gの方式は簡潔で十分な表現力があり、よい選択だと思われます。

それぞれTRSMのデフォルトは以下のとおりです。

- T = (0,0,0)
- R = (0, (0,0,0))
- S = (1,1,1)
- M = 単位行列

デフォルトのTRSMの変換を行うと結果的に”そのまま”です。

平行移動要素(T)を変更するにはsetTranslation()関数を使用します。この関数は以前のT要素を破棄し引数で指定されたT要素(tx,ty,tz)に変更します。

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

引数は新しいT要素(tx,ty,tz)です。

現在のT要素を破棄せずに新しいT要素を追加にはtranslate()関数を使用します。古いT要素に引数のT要素を足した値が新しいT要素になります。

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

引数は追加したいT要素(tx,ty,tz)です。

現在のT要素を取得するにはgetTranslation()関数を使用します。

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

引数 xyz は結果を書き込む float3 つ分のメモリ領域です。T 要素(tx,ty,tz)がこの順に書き込まれます。引数が NULL の場合は例外を発生します。

回転要素(R)を変更するには setOrientation()関数を使用します。回転軸と回転角度は angle-axis 形式で指定します。この関数は以前の R 要素を破棄し引数で指定された R 要素(angle, axis(ax,ay,az))に変更します。

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

引数は回転角度(angle)、回転軸(ax,ay,az)です。回転軸は正規化されている必要はありません。回転軸に(0,0,0)を指定すると例外を発生します。ただし angle=0, axis=(0,0,0)の場合のみ受け入れます。

> 補足:
>
> R 要素をセットする関数の名前が setRotation()ではなく setOrientation()なのはなぜなのだろうか。

現在の R 要素を破棄せずに新しい R 要素を付け足すには preRotate()関数か postRotate()関数を使用します。以前の R 要素に引数の R 要素を付け加えた値が新しい R 要素になります。

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

引数は angle-axis 形式の回転角度(angle)・回転軸(ax,ay,az)です。制限は setOrientation()の時と同じです。2 つの関数の違いは引数の R 要素を右からかける(preRotate)か左からかける(postRotate)かです。

$$V' = R\,R_{pre}\,V$$
$$V' = R_{post}\,R\,V$$

引数の R 要素を先に適応したい場合は pre を、後に適応したい場合は post を使用します。一般的に 2 つの回転 A,B は適応する順番によって結果が異なります。

現在の R 要素を取得するには getOrientation()関数を使用します。

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

引数 angle_axis は結果を書き込む float4 つ分のメモリ領域です。angle-axis 形式で書き込まれます。0 番目が角度(angle)で度数で書き込まれます。1、2、3 番目が回転軸(ax,ay,az)です。回転軸は正規化されています。引数が NULL の場合は例外を発生します。

スケール要素(S)を変更するには setScale()関数を使用します。この関数は以前の S 要素を破棄し引数で指定された S 要素(sx,sy,sz)に変更します。

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

引数は新しい S 要素(tx,ty,tz)です。

現在の S 要素を破棄せずに新しい S 要素を付け足すには scale()関数を使用します。古い S 要素に引数の S 要素をかけた値が新しい S 要素になります。

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

引数は追加したい S 要素(sx,sy,sz)です。

現在の S 要素を取得するには getScale()関数を使用します。

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

引数 xyz は結果を書き込む float3 つ分のメモリ領域です。それぞれ sx, sy, sz が順に書き込まれます。引数が NULL の場合は例外を発生します。

4x4 行列要素(M)を変更するには setTransform()関数を使用します。この関数は以前の M 要素を破棄し引数で指定された 4x4 行列に変更します。

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

引数は新しい 4x4 行列を表す Transform オブジェクトです。Transform クラスについては後に続く項を参照して下さい。

現在の M 要素を破棄せずに新しい M 要素を付け加える API はありません。

現在の M 要素を取得するには getTransform()関数を使用します。

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

引数 transform は結果を書き込む Transform オブジェクトへのポインターです。引数が NULL の場合は例外を発生します。

原則 4x4 行列の M 要素はユーザーがセットし、ライブラリ側では関与しません。例外的に Desktop-M3G では唯一 Camera クラスの M3G 非標準の関数 lookAt()が M 要素を設定します。

Transform

Transform クラスは座標変換を表す 4x4 行列を保有します。ます。このクラスは関数引数などで使われ、使用後は破棄される寿命の短いオブジェクトです。このように一時的な利用を想定しているクラスは Object3D を派生していません。

Transform クラスのコンストラクタは引数なしで呼び出します。

```
Transform ();
```

インスタンス化直後の Transform オブジェクトの保有する 4x4 行列は単位行列です。

現在の 4x4 の変換行列を取得するには get()関数を使用します。

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

引数 matrix は結果を書き込む float16 個の配列へのポインターです。行優先で格納されます。引数が NULL の場合は例外を発生します。4x4 行列の並びを下式とすると、

$$\text{4x4行列} = \begin{pmatrix} m_{00} & m_{01} & m_{02} & m_{03} \\ m_{10} & m_{11} & m_{12} & m_{13} \\ m_{20} & m_{21} & m_{22} & m_{23} \\ m_{30} & m_{31} & m_{32} & m_{33} \end{pmatrix}$$

引数の matrix には下式の順番で書き込まれます。

$$matrix = [m_{00}\ m_{01}\ m_{02}\ m_{03}\ m_{10}\ m_{11}\ m_{12}\ m_{13}\ m_{20}\ m_{21}\ m_{22}\ m_{23}\ m_{30}\ m_{31}\ m_{32}\ m_{33}]$$

この Transform オブジェクトの変換行列を変更するには set()関数を使用します。16 個の float 配列で指定する形式と Transform オブジェクトで指定する形式の 2 つが存在します。

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

1 番目の形式では引数の matrix に設定したい 4x4 行列を 16 個の float で行優先で指定します。NULL の場合は例外を発生します。2 番目の形式では既に存在する Transform オブジェクトを渡します。

RayIntersection

RayIntersection クラスはピッキングの結果を収納する一時的な利用を想定しているクラスです。このクラスは Object3D を継承しません。ユーザーが直接パラメーターを設定することはありません。ピッキングは Group クラスの pick()関数が行い、結果を RayIntersection オブジェクトに格納して返します。pick()関数の詳細は第 4 章を参照して下さい。

RayIntersection クラスのコンストラクタは引数なしで呼び出します。

```
RayIntersection ();
```

インスタンス化直後の値はすべてデフォルトです。この状態では何も意味を持ちません。このオブジェクトはピッキングの結果として pick()関数によって値が設定されてから意味を持ちます。

ピッキングによって検出されたノードを取得するには getIntersected()関数を使用します。ピッキングはレイ光線によって最初にヒットしたピッキング可能なノードを返します。

```
Node* getIntersected () const;
```

戻り値はヒットしたノードへのポインターです。何もヒットしなかった場合は NULL が返ります。

ヒットした地点の法線を取得するには getNormalX(), getNormalY(), getNormalZ()関数を使用します。

```
float getNormalX () const;
```

```
float getNormalY () const;
float getNormalZ () const;
```

戻り値は衝突地点の持つ正規化された法線(nx,ny,nz)です。デフォルトは(0,0,1)です。

衝突地点のテクスチャー座標を取得するには getTextureS(), getTextureT()関数を使用します。

```
float getTextureS (int index) const;
float getTextureT (int index) const;
```

引数 index は 0 から始まる取得したいテクスチャーユニット番号です。使用していないテクスチャーユニット番号の指定は例外を発生します。戻り値はテクスチャー座標(s,t)です。[0,1)の範囲の値が返ります。

ヒットしたノードのサブメッシュインデックスを取得するには getSubmeshIndex()関数を使用します。

```
int getSubmeshIndex () const;
```

戻り値はサブメッシュのインデックスです。サブメッシュについては第 4 章のメッシュの項を参照して下さい。

ピッキングに使用したピック光線は getRay()関数で取得します。

```
void getRay (float* ray) const;
```

引数 ray は結果を書き込む 6 個の float の配列へのポインターです。順番にピック光線の起点(ox,oy,oz)と方向(dx,dy,dz)がこの順で書き込まれます。方向は正規化されています。

ピック光線の起点から衝突地点までの距離を取得するには getDistance()関数を使用します。

```
float getDistance () const;
```

戻り値は衝突地点までの距離です。ピック光線は半直線なので必ず 0 以上の値を返します。

Image2D

Image2D クラスは 2 次元画像を表すクラスです。Texture2D と Sprite3D から使用されます。

Image2D クラスのコンストラクタは 3 種類あります。

```
Image2D (int format, int width, int height);
Image2D (int format, int width, int height, void* pixels);
Image2D (int format, int width, int height, unsigned char* pixels, void* palette);
```

すべてに共通する引数 format は表 2 の中から画像フォーマットを指定します。

フォーマット	バイト/ピクセル (bpp)	説明
ALPHA	1	A 値のみ
LUMINANCE	1	輝度値のみ
LUMINANCE_ALPHA	2	輝度値 + α 値
RGB	3	R,G,B の 3 色
RGBA	4	R,G,B の 3 色 + α 値

表 2:画像フォーマット一覧

引数の width には画像の幅を, height には高さをピクセル数で指定します。format, width, height のみを指定する 1 番目の形式のコンストラクタは、指定されたフォーマット・サイズの空の画像を作成します。各ピクセル値は未定義です。2 番目の形式のコンストラクタは pixels から画像データをコピーします。pixels は画像の初期化に必要な十分データ(width * height * bpp)を指している必要があります。画像データは内部的に確保したメモリに直ちにコピーされます。コンストラクタから処理が返った後は pixels を開放してかまいません。3 番目の形式はにパレット化された画像を使用します。pixels は画像データそのものではなくパレットのインデックス(0～255)の配列を指します。palette は 255 個の色の集合を指します。最終的な色は下記の式に従って計算されます。

```
Color[i] = palette[pixels[i]]
```

画像データは内部的に確保したメモリにコピーされるので pixels および palette はコンストラクタの呼び出しが返ったら開放してかまいません。なお Desktop-M3G の実装では内部的にはパレットを使わず普通の画像と同じように処理します。パレット化画像を使うメリットはありません。

初期化を行わない 1 番目の形式の画像をミュータブル(mutable)、初期化を伴う 2、3 番目の形式の画像をイミュータブル(immutable)と呼びます。イミュータブルな画像はインスタンス後は変更出来ません。ミュータブ

ルかイミュータブルかを取得するには isMutable()関数を使用します。

```
bool isMutable () const;
```

戻り値はミュータブルの場合が true,イミュータブルの場合が false です。

画像のフォーマットを取得するには getFormat()関数を使用します。

```
int getFormat () const;
```

戻り値は表 2 の画像フォーマットのいずれかでコンストラクタで指定したフォーマットが返ります。

画像のサイズを取得するには getWidth()関数と getHeight()関数を使用します。

```
int getWidth () const;
int getHeight () const;
```

それぞれ戻り値が画像の幅と高さのピクセル数です。

ミュータブルな画像にデータをセットするには set()関数を使用します。

```
void set (int x, int y, int width, int height, void* image);
```

画像の一部分(x,y,width,height)を引数 image からコピーして変更します。image は十分な大きさ(width * height * bpp)が必要です。画像は左上が原点(0,0)です。元画像の領域を越える書き込みは例外を発生します。

Image2D オブジェクトは Loader クラスの load()関数で PNG 画像もしくは JPEG 画像をロードして取得することも出来ます。通常この方法がもっとも手軽に画像ファイルから Image2D オブジェクトを作成する方法です。

```
Image2D* img = dynamic_cast<Image2D*>(Loader::load ( “image.png” )[0]);
```

詳しくは第 8 章 Loader クラスの項を参照して下さい。

第4章　シーングラフ

シーングラフは World クラスを頂点とする木構造のグラフです。レンダリング対象はすべてシーンに属しています。現在 M3G で定義されているシーンノードは Camera, Light, Mesh, SkinnedMesh, MorphingMesh, Sprite3D, Group, World の 8 種類です。これらは全て Node クラスを継承するシーンノードです。その基底クラスである Object3D クラスと Transformable クラスの解説は第 3 章を参照してください。

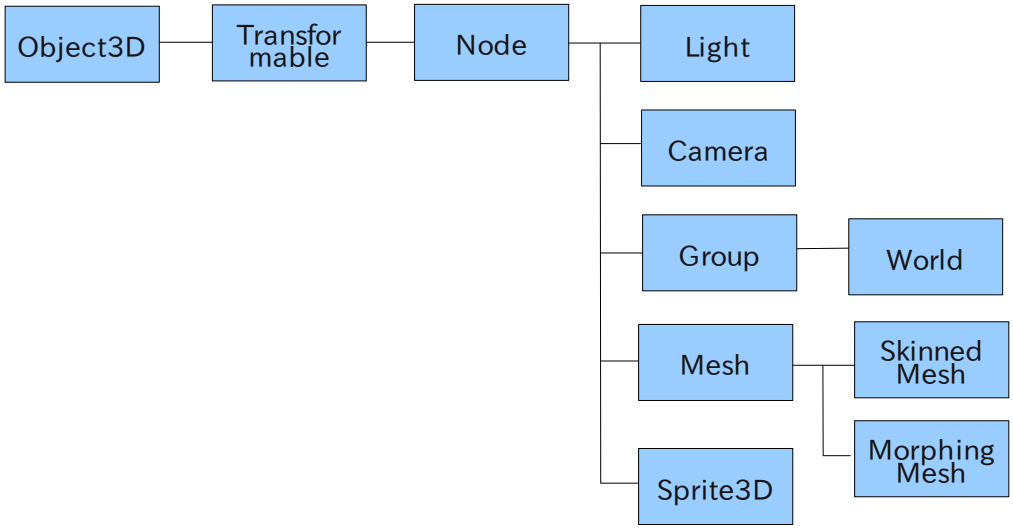


図 6:シーンノードのクラス階層

Node

Node はシーングラフを構成するノードを表す抽象クラスです。すべてのシーンノードはこのクラスを継承します。抽象クラスなのでインスタンス化はされません。ノードで共通の処理を行います。

スコープを設定するには setScope()関数を使用します。スコープは INT 型の整数値で AND を取った結果が 0 以外のときに”そのスコープに含まれ”ます。スコープはレンダリングやピッキングの制御に使われます。デフォルトのスコープは-1(0xffffffff)です。従って必ず全てのスコープに含まれます。

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

引数 scope は新しいスコープです。デフォルトは-1 です。

現在の設定されているスコープを取得するには getScope()関数を使用します。

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

戻り値はスコープです。

ノード α (不透明度)の変更を行うには setAlphaFactor()関数を使用します。

```
void setAlphaFactor (float alpha);
```

引数の alpha には新しいノード α 値を[0,1]で指定します。デフォルトは 1(完全不透明)です。ノード α はこのノードがレンダリングされる時の基本的な不透明度を表します。0 に設定するとその物体は完全に透明になります。レンダリング時の α は自ノードからたどって一番上の親(通常は World)までノード α 値を全て掛け合わせた値が使用されます。これを利用すると World クラスのノード α を制御することで画面全体をフィードイン、フォードアウトさせることが出来ます。こうして求めた α は最終的に拡散色の α 成分に乗算されます。

現在のノード α を取得するには getAlphaFactor()関数を使用します。

```
float getAlphaFactor () const;
```

戻り値はノード α です。

ノードのレンダリングフラグを変更するには setRenderingEnable()関数を使用します。

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

引数 enable が true の時は描画され、false の時は描画されません。デフォルトは true です。

現在のレンダリングフラグを取得するには isRenderingEnabled()関数を使用します。

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

戻り値はレンダリングフラグです。

ノードのピッキングフラグを変更するには setPickingEnable()関数を使用します。

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

引数 enable が true の時はピッキング対象となり、false の時はピッキングされません。デフォルトは true です。

現在のピッキングフラグを取得するには isPickingEnabled()関数を使用します。

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

戻り値はピッキングフラグです。

アライメントとはあるノードの 1 軸もしくは 2 軸を自動的に別のノードの軸と同期させる機能です。例えば Sprite3D ノードの Z 軸を常にカメラの Z 軸にアライメントさせると、このスプライトは常にカメラの方角を向きます。これはビルボードと呼ばれるテクニックです。自動的なアライメントの調整は Node::align()関数によって実行されます。アライメントの一例を図 6 に示します。これは XZ 平面を Y 軸正の方向から見た図で Sprite3D ノードと Camera ノードの 2 つが存在します。Sprite3D ノードの Z 軸を Camera ノードの Z 軸に合わせるアライメントを設定し、align()関数を呼ぶと自動的に図の右側のように Sprite3D の Z 軸が Camera の Z 軸に整合します。

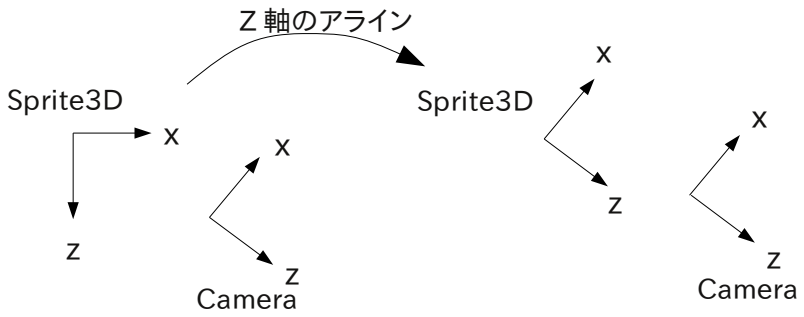


図 7:アライメントの一例

アライメントを設定するには setAlignment()関数を使用します。

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

引数の z_ref には Z 軸を合わせたい対象ノードを指定します。z_target には ORIGIN もしくは NONE を指定します。ORIGIN は指定のノードにアライメントします。NONE はアライメントを消去します。既に設定済みのアライメントがある場合新しいアライメントに置き換えられます。Y 軸に関しても同様です。アライメントは必ず先に Z 軸に対して調整し、その後に Y 軸に対して調整します。X 軸は設定できません。

アライメント対象は NULL のままにしておき、Node::align()の呼び出しのときに対象ノードを指定することも可能です。

Camera

Camera クラスは現実世界のカメラと同様に、描画の視点、方向、視野角などを定義します。シーンには複数の Camera オブジェクトが存在するかもしれません。その中から 1 つをレンダリングに使うアクティブカメラに指定します。レンダリングを行うためにはアクティブカメラの設定が必須です。カメラには透視投影と平行投影、それから任意の射影行列を指定できるジェネリックの 3 種類が定義されています。

Camera のコンストラクタは引数なしで呼び出します。

```
Camera ();
```

デフォルトはジェネリックで、射影変換は単位行列に設定されています。位置は原点(0,0,0)で Z 軸負の方向(0,0,-1)を向いています。up ベクトルは(0,1,0)です。

カメラを平行投影に変更するには setParallel()関数を使用します。

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

引数の height にはカメラの視野領域のカメラ空間での高さを指定します。aspect_ratio にはアスペクト比(横/縦)を指定します。near と far はそれぞれ視野領域のカメラ空間でのニア面とファー面を指定します。

カメラを透視投影に変更するには setPerspective()関数を使用します。

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

引数の fovy には Y 方向の視野角を度数で指定します。aspect_ratio にはアスペクト比(横/縦)を指定します。near と far はそれぞれ視野領域のカメラ空間でのニア面とファー面を指定します。

カメラをジェネリックに変更するには setGeneric()関数を使用します。これはデフォルトのカメラ設定です。

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

引数 transform は射影変換を表す Transform クラスです。任意の 4x4 行列を指定可能です。

設定したカメラパラメーターを取得するには getProjection()関数を使います。この関数は引数の違う 2 種類が存在します。平行投影と透視投影の場合は 1 番目の形式を、ジェネリックの場合は 2 番目の形式を使用します。

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

1 番目の形式の引数の params には結果を書き込む float4 個分の配列へのポインターを指定します。params には setParallel()もしくは setPerspective()の引数で渡したパラメーター(fovy(or height), aspect_ratio, near, far)がそのまま書き込まれます。2 番目の形式の引数 transform は結果を書き込む Transform オブジェクトへのポインターです。setGeneric()の引数で指定た射影行列が書き込まれます。戻り値はどちらの形式でもカメラの種類で GENERIC,PARALLEL,PERSPECTIVE のどれか 1 つが返ります。

カメラの位置・姿勢を制御するには Camera クラスの基底クラスである Transformable クラスのメンバー関数を使用するのが基本です。これは第 3 章で解説があります。ただしこれらの関数のみを使って任意の位置、方向に設定するのはなかなか難しいものがあるので、Desktop-M3G では補助関数として M3G 非標準の lookAt()関数を提供しています。これは glut の glutLookAt()関数に相当します。

```
void lookAt (float from_x, float from_y, float from_z,
                     float to_x, float to_y, float to_z,
                     float up_x, float up_y, float up_z);
```

引数は glutLookAt()関数と同様にカメラの位置(from_x, from_y, from_z)、注視点(to_x, to_y, to_z)、アップベクトル(up_x, up_y, up_z)を指定します。lookAt()関数はこれらのパラメーターから 4x4 の変換行列を作成し Transformable クラスの M 要素に設定します。Transformable の TRSM 要素の解説は第 3 章を参照して下さい。

カメラはレンダリングする前に World クラスの setActiveCamera()関数を使ってアクティブカメラに設定する必要があります。

```
Camera* cam = new Camera;
// ここでカメラの設定
```

```
Wld.addChild (cam);
wld.setActiveCamera (cam);
```

レンダリング結果はこのアクティブカメラから見た画像になります。

Light

Light クラスはシーン中の光源をあらわすシーンノードです。ライトには環境光(AMBIENT)、平行光源(DIRECTIONAL)、点光源(OMNI)、スポットライト(SPOT)の 4 種類が定義されています。最終的にレンダリングされる物体の色はライトとマテリアルの相互作用によって決まります。

ライトノードのコンストラクタは引数なしで呼び出します。

```
Light ();
```

デフォルトは平行光源(DIRECTIONAL)で位置は原点(0,0,0)、方向は Z 軸負の方向(0,0,-1)を向いています。光源のカラーは 0x00ffffff (白色)で強度は 1 です。減衰パラメーターは(1,0,0)、スポット角度、強度は 45 度と 0 に設定されています。

ライトの種類の設定には setMode()関数を使用します。

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

引数の mode には表 3 の AMBIENT, DIRECTIONAL, OMNI, SPOT のいずれかを指定します。デフォルトは DIRECTIONAL です。

ライトの種類	解説
AMBIENT	環境光
DIRECTIONAL	平行光源
OMNI	点光源
SPOT	スポットライト

表 3:ライトの種類

ライトのカラーの変更には setColor()関数を使用します。

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

引数は RGB を int 型で指定します。デフォルトは 0x00ffffff(白色)です。

ライトの強度の変更には setIntensity()関数を使用します。

```
void setIntensity (float intensity);
```

引数 intensity は強度を表す float 値です。デフォルトは 1.0 です。intensity は 0 もしくは負の値も受け付けます。負の値のライトで照らされた物体は暗くなります。使用中の GPU によっては負の値のライトの効果が異なるかもしれません。

距離による減衰の設定は setAttenuation()関数で行います。

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

減衰係数は constant(c), linear(l), quadratic(q)の 3 つのパラメーターからなり、距離を d として以下の式で計算されます。

$$減衰係数 = \frac{1}{c + ld + qd^2}$$

デフォルトは(c,l,q)=(1,0,0)なので減衰係数は 1 となり、距離による減衰はありません。c,l,q に負の値を設定することはできません。また 3 係数すべてを同時に 0 にすることはできません。光源が環境光の時は減衰係数は無視されます。

> 補足: 点光源なのに減衰なし?
>
> 現実世界の点光源の強度は距離の 2 乗に比例して減衰します。従って減衰係数は(0,0,1)が物理的に正しい値となりますが、これだと距離による減衰が強すぎて、レンダリング時に物体がシェーディングされた結果[0,1]の値に収まるような光源強度に設定するのを難しくします。グラフィックツールでは点光源でも減衰なしの(1,0,0)を使うことが多いようです。

現在の減衰係数の設定は以下の関数で取得可能です。

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

戻り値はそれぞれ減衰係数の計算式中のパラメーター constant, linear, quadratic です。

スポットライトの角度を設定するには setSpotAngle()関数を使用します。

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

引数の angle はスポットライトの角度(中心軸から外側まで)を度数(degree)で指定します。デフォルトは 45 度です。有効な値は[0,90](0 と 90 を含む)です。スポット角度はライトの種類がスポットライト(SPOT)の場合のみ有効で、それ以外の場合は無視されます。

設定したスポット角度を取得するには getSpotAngle()関数を使います。

```
float getSpotAngle () const;
```

戻り値はスポットライトの角度です。

スポットライトの指数係数を設定するには setSpotExponent()関数を使用します。

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

引数の exponent はスポットライトの計算で使われる指数係数を[0,128]で指定します。デフォルトは 0 です。

スポット指数係数はライトの種類がスポットライト(SPOT)の時のみ有効で、その他の場合は無視されます。

スポット指数係数を取得するには getSpotExponent()関数を使用します。

```
float getSpotExponent () const;
```

戻り値はスポットライトの指数係数です。

ライトのオン、オフは基底クラスの Node::setRenderingEnable()関数を使用します。オフの状態のライトは物体に一切影響を与えません。

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

またライトが影響を与えるのは、ライトのスコープと同じスコープのノードのみです。デフォルトのスコープは-1 なのでライトは全てのノードに影響を与えます。特定のノードのみに影響を与えたい場合等はスコープを制御します。スコープについては第 3 章を参照して下さい。

Mesh

Mesh クラスは物体を表すもっとも基本的なシーンノードです。トライアングル 1 つ表示する場合でもこのメッシュを使います。M3G で表示される物体は Mesh クラスかその派生クラスである MorphingMesh クラス、SkinnedMesh クラスのいずれかです。Mesh クラスは変形しない剛体の物体をあらわすクラスです。変形を伴うキャラクターアニメーションには SkinnedMesh クラスを、モーフィングには MorphingMesh クラスを使用します。

メッシュクラスをインスタンス化するには頂点データ(VertexBuffer オブジェクト)と、頂点をつないで面を形成するインデックスデータ(IndexBuffer オブジェクト)、それに見え方の定義するアピアランス(Appearance オブジェクト)の 3 つが必要です。1 つの VertexBuffer にたいして IndexBuffer と Appearance は同時に複数組み合わせることができます。このとき 1 つの IndexBuffer があらわす面の集合をサブメッシュと呼びます。

VertexBuffer と IndexBuffer は第 5 章を、Appearance は第 6 章を参照して下さい。ここではそれらのクラスが適切にインスタンス化されているものとします。

Mesh クラスのインスタンスは VertexBuffer, IndexBuffer, Appearance オブジェクトを引数に取ります。

```
Mesh (VertexBuffer*  vertices,
            IndexBuffer*   submesh,
            Appearance*  appearance);
Mesh (VertexBuffer*  vertices,
            int                    num_submesh,
```

```
                IndexBuffer**  submeshes,
                Appearance** appearances);
```

1 番目の形式は 1 つの VertexBuffer に 1 つの IndexBuffer, Appearance を指定してインスタンス化します。サブメッシュは 1 つです。2 番目の形式は 1 つの VertexBuffer に対して複数の IndexBuffer, Appearance を指定してインスタンス化します。複数のサブメッシュを持ちます。どちらの形式を使用しても機能に差はありません。インスタンス化した時に使用した VertexBuffer と IndexBuffer を後で別のものに差し替えることはできません。Appearance のみ差し替えが可能です。これは色違いの 2P キャラクターを表示する時などに使われます。

設定されている VertexBuffer を取得するには getVertexBuffer()関数を使用します。

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

戻り値は VertexBuffer オブジェクトへのポインターです。

サブメッシュの数を取得するには getSubmeshCount()関数を使用します。

```
int getSubmeshCount () const;
```

戻り値はサブメッシュの数です。

設定されている IndexBuffer を取得するには getIndexBuffer()関数を使用します。

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

引数の index は 0 から getSubmeshCount()-1 までが有効です。戻り値は IndexBuffer オブジェクトへのポインターです。通常はその派生クラスである TriangleStripArray オブジェクトを指します。

同様にアピアランスを取得するには getAppearance()関数を使用します。

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

引数の index は 0 から getSubmeshCount()-1 までが有効です。

アピアランスを変更するには setAppearance()関数を使用します。

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

引数の index はサブメッシュの番号を指定します。0 から getSubmeshCount()-1 までが有効です。
appearance には設定したい Appearance オブジェクトのポインターを指定します。NULL を指定するとデフォルトのアピアランスが使用されます。

Mesh クラスは変形を伴わないのでインスタンス化した後に固有の操作(メンバー関数)を必要としません。メッシュノード全体を動かすには基底クラスの Transformable クラスのメンバー関数を使用します。

SkinneMesh

SkinnedMesh クラスは Mesh クラスにスケレタルアニメーション用のボーンを追加したシーンノードです。
Mesh クラスから派生しています。このクラスの頂点データ(スキン)はボーンに追従して移動・変形します。キャラクターアニメーションに使用されます。SkinnedMesh クラスの詳細については第 5 章モデリングを参照してください。

MorphingMesh

MorphingMesh クラスは Mesh クラスにモーフィング用のモーフターゲットを追加したシーンノードです。Mesh クラスから派生しています。このクラスは複数の頂点データ(モーフターゲット)を取り、そのターゲット間をなめらかに遷移します。MorphingMesh クラスの詳細については第 5 章モデリングを参照して下さい。

Sprite3D

スプライト 3D は 3D 空間上で 2 次元の画像を扱うためのシーンノードの 1 つです。Desktop-M3G の実装では単なるテクスチャー付きの板ポリゴンとして実装されています。スプライトは必ずカメラの方向を向き縦横の辺が Y 軸と X 軸に沿って配置されます。unscale なスプライトはスクリーン空間上で常に一定の大きさで表示されます。これはゲーム画面でスコアやゲーム情報などを表示するのに使われます。scale なスプライトはワールド空間上で一定の大きさを持ち、従ってカメラからの距離に応じて大きさが変化します。これは例えばワールド空間上の特定の位置にアイテムを表示するのに使われたりします。スプライトでは画像の一部のみを表示することができます。この表示する領域をクロップ領域と呼びます。デフォルトでは画像全体が表示されます。クロップ領域を時間で変化させることで例えば画像のスクロールを実装することができます。

Sprite3D のコンストラクタはスケーリングフラグ、表示する画像、アピアランスを引数に取ります。

```
Sprite3D (bool scaled, Image2D* image, Appearance* appearance);
```

第 1 引数の scaled はスケール・アンスケールを指定します。image には表示したい画像を、appearance には見え方を定義するアピアランスを指定します。Sprite3D では Appearance の持つ CompositingMode 要素と Fog 要素のみ有効で残りの要素は無視されます。

スケールフラグを取得するには isScaled()関数を使用します。

```
bool isScaled () const;
```

戻り値は scaled なときは true で、unscaled の時は false です。

表示画像を差し替えるには setImage()関数を使用します。

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

引数 image で指定された画像がこのスプライトで表示されるようになります。image に NULL は指定できません。この関数が呼ばれるとクロップ領域は画像全域にリセットされます。

設定されている画像の取得は getImage()関数で行います。

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

戻り値は Image2D オブジェクトへのポインターです。

アピアランスを差し替えるには setAppearance()関数を使用します。

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

引数 appearance には新しい Appearance オブジェクトを指定します。引数に NULL を指定するとデフォルトのアピアランスが使われるようになります。

クロップ領域を設定するには setCrop()関数を使用します。デフォルトのクロップ領域は画像全域です。

```
void setCrop (int crop_x, int crop_y, int crop_width, int crop_height);
```

クロップ領域の設定は表示したい領域の左上の座標(crop_x,crop_y)と幅(crop_width)、高さ

(crop_height)で指定します。画像の原点(0,0)は左上で右下が(width-1,height-1)です。クロップ領域の設定は元画像の領域(0,0,width,height)外を指定できます。例えばクロップ領域として(0,0,width*2, height)を指定すると X 方向に 2 回繰り替えした画像が表示されます。また幅と高さにマイナスを指定すると反転します。例えばクロップ領域として(0,0,-width, height)を指定すると左右反転した画像が表示されます。画像領域外の描画指定は必ずリピートになりボーダーカラーのようなものは設定できません。

現在設定されているクロップ領域を取得するには getCropX(), getCropY(), getCropWidth(), getCropHeight()関数を使用します。

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

戻り値はクロップ領域(crop_x, crop_y, crop_width, crop_height)です。

Group

Group ノード(とその派生クラスである World ノード)は子ノードを取れる唯一のシーンノードです。シーンは Group ノードによって階層化されます。またこのクラスは SkinnedMesh クラスの持つスケルトン階層のボーンとしても使用されます。

Group クラスのコンストラクタは引数なしで呼び出します。

```
Group ();
```

現在保持している子ノードの個数を取得するには getChildCount()関数を使用します。

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

戻り値はこのオブジェクトが直に保持している子ノードの数です。孫ノード以下は数に入りません。

この Group ノードに子ノードを追加するには addChild()関数を使用します。

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

引数の child には追加したい子ノードを指定します。child に NULL は指定できません。またすでに他のノードの子ノードになっているノードも追加できません。World クラスは特殊なクラスなので追加できません。自分自身を子ノードとして追加することはできません。

子ノードを取得するは getChild()関数を使用します。

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

引数 index は 0 から getChildCount()-1 まで有効です。子ノードのインデックスと Node オブジェクトの対応は addChild()または removeChild()したタイミングで変わる可能性があります。子ノードの並び順に依存したプログラムを書いてはいけません。

子ノードは removeChild()で削除することができます。

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

引数 child には削除したい子ノードのポインターを渡します。不正な引数、例えば NULL や子ノードとして存在しないノードは安全に無視されます。この関数は例外を発生しません。

ピッキングを行うには pick()関数を使います。pick 関数は用途別に 2 つの形式があります。ピッキングはこの Group ノードからたどれる全ての子ノードを再帰的にピックします。通常これは Group クラスを継承した World クラスで行いシーングラフ全体をピックします。

一番目の形式は現在見えているスクリーン座標上の座標を指定してピックします。通常これはマウスで物体を選択するのに使います。

```
bool pick (int scope, float x, float y, const Camera* camera,
                    RayIntersection* ri) const;
```

第 1 引数の scope は検索したい物体のスコープを指定します。スコープの詳細については第 3 章を参照してください。すべてのスコープを検索するには-1 を指定します。第2第 3 引数はピック対象のスクリーン座標(x,y)をピクセル数で指定します。スクリーン上に見えている範囲は(0,0,width-1,height-1)ですが、それ以外の領域も指定可能です。第 4 引数の camera にはピック時に使うカメラを指定します。通常これはアクティブカメラを指定します。他の(現在レンダリングには使用していない)カメラも利用可能です。5 番目の引数は結果を受け取る RayIntersection オブジェクトへのポインターです。RayIntersection クラスはピックされたノードや、ピックに使われたレイ光線の情報、それに衝突した地点の法線、テクスチャー座標などの情報を持ちます。詳しくは第 3 章を参照して下さい。戻り値は物体にヒットした場合は true、ヒットしなかった場合は false です。

2 番目の形式は任意の位置・方向でレイ光線を飛ばしてピックします。座標系はこの Group ノードのものが使われます。

```
bool pick (int scope, float ox, float oy, float oz,
                  float dx, float dy, float dz, RayIntersection* ri) const;
```

引数はレイ光線の位置(ox,oy,oz)と方向(dx,dy,dz)をこの Group クラスと同じ座標系で指定します。例えば World クラスでピックした場合座標系はワールド座標系になります。結果を書き込む RayIntersection オブジェクトへのポインターと戻り値は 1 番目の形式と同じです。

スプライト 3D のピックは特殊で、scale なスプライトのみピック可能です。unscaled なスプライトはピックされません。

> 補足: Desktop-M3G での実装
>
> 現在の実装だとピック光線と全てのトライアングルに対して交差判定を計算しているのでとても遅いです。将来的にはバウンディングボックスまたはスフィアを実装して高速化する予定です。

World

World クラスは Group クラスを継承する特別なクラスです。World クラスは親ノードを持たず全てのノード階層の一番頂上に位置します。1 つの World オブジェクトが 1 つのシーンを表します。World クラスは Graphics3D クラスの render()関数に渡してレンダリングできる唯一のクラスです。World クラスにはシーンをレンダリングするために 1 つのアクティブなカメラを設定する必要があります。また任意で Background オブジェクト(シーンノードではない)を接続できます。

World クラスのコンストラクタは引数なしで呼び出します。

```
World ();
```

カメラをアクティブに設定するには setActiveCamera()関数を使用します。

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

引数 camera は Camera オブジェクトのポインターです。この関数を呼ぶ時点では Camera オブジェクトはシーン中に存在する必要はありませんが、レンダリング時にはシーン中に存在している必要があります。レンダリングするにはシーン中のカメラのうち 1 つが”アクティブ”に設定されている必要があります。

現在アクティブなカメラを取得するには getActiveCamera()関数を使用します。

```
Camera* getActiveCamera () const;
```

戻り値はアクティブな Camera オブジェクトのポインターです。アクティブなカメラが無い場合は NULL が返ります。

背景として Background オブジェクトを設定するには setBackground()関数を使用します。

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

引数 background は設定したい Background オブジェクトへのポインターです。引数が NULL の場合はデフォルトの背景(画像なし、黒色)に設定されます。

現在設定されている Background オブジェクトを取得するには getBacckground()関数を使用します。

```
Background* getBackground () const;
```

戻り値は Background オブジェクトのポインターです。設定されてない場合は NULL が返ります。
Background クラスに関しては次の項を参照して下さい。

Background

Background オブジェクトは World オブジェクトに接続し背景を設定します。Background クラスは Node クラスから派生していないので、シーンノードではありません。

Background クラスのコンストラクタは引数なしで呼び出します。

```
Background ();
```

フレームバッファー(カラーバッファー)のクリアの設定を変更するには setColorClearEnable()関数を使用し

ます。true だとフレームバッファーはレンダリングに先立ち背景色でクリアされます。

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

引数 enable が true の時はクリアされ、false の時はクリアされません。デフォルトは true です。

現在の設定を取得するには isColorClearEnabled()関数を使用します。

```
bool isColorClearEnabled () const;
```

戻り値はカラークリアーが有効なら true, 無効なら false です。

デプスバッファーのクリアの設定を変更するには setDepthClearEnable()関数を使用します。デプスバッファーはレンダリングに先立ち 1.0(もっとも奥の値)でクリアされます。なお M3G ではデプスバッファーの有効範囲は[0,1]でクリア値も 1.0 で固定です。

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

引数 enable が true の時はクリアされ、false の時はクリアされません。デフォルトは true です。

現在の設定を取得するには isDepthClearEnabled()関数を使用します。

```
bool isDepthClearEnabled () const;
```

戻り値はデプスクリアーが有効なら true, 無効なら false です。

背景色を変更するには setColor()関数を使用します。

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

引数の argb には背景色を int 型の ARGB 形式で指定します。デフォルトは 0x00000000(黒)です。A 成分はフレームバッファーの形式に依存します。

現在設定されている背景色を取得するには getColor()関数を使用します。

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

戻り値は背景色です。

背景画像を設定するには setImage()関数を使用します。

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

引数の image には表示したい画像の Image2D オブジェクトのポインターを指定します。画像の幅と高さは 2 の階乗である必要があります。NULL を指定すると画像なし、つまり背景色で塗りつぶされます。デフォルトは NULL です。この関数を呼び出すとクロップ領域が画像全域(0,0,width,height)に設定されます。

背景画像はある部分だけ切り取って表示できます。これをクロップ領域と呼びます。背景のクロップ領域にアニメーションを付けると背景スクロールを実装することができます。クロップ領域の設定は setCrop()関数で行います。

```
void setCrop (int crop_x, int crop_y, int width, int height);
```

引数は画像の左上を(0,0)としてクロップ領域を(crop_x, crop_y)～(crop_x + width, crop_y + height)に設定します。width と height は 0 より大きなピクセル数で指定します。クロップ領域は背景画像の領域の外を指定してもかまいません。その場合の動作は後述のイメージモードの設定に依存します。デフォルトは画像全域です。

現在設定されているクロップ領域の取得には getCropX(), getCropY(), getCropWidth(), getCropHeight()を使用します。

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

戻り値はクロップ領域の(x,y,width,height)です。

イメージモードの設定は setImageMode()関数で行います。

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

引数はそれぞれ x 方向と y 方向のイメージモードを BORDER もしくは REPEAT から選択します。BORDER は元画像の領域外を背景色で塗りつぶします。REPEAT は背景画像を繰り返します。

イメージモードを取得するには getImageModeX(), getImageModeY()関数を使用します。

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

戻り値はそれぞれＸ方向、Ｙ方向のイメージモードです。

第5章　モデリング

この章では M3G API を使ったモデリングの解説を行います。M3G では 3D 空間内で表示される物体(モデル)1 つが、1 つの Mesh オブジェクトまたはその派生クラスのオブジェクトに相当します。メッシュはトライアングルストリップと呼ばれるトライアングルの一筆書きで構成されています。1 本のトライアングルストリップの事をサブメッシュと呼ぶ事があります。1 つのメッシュは複数のサブメッシュ(トライアングルストリップ)と複数のアピアランスを持つ事ができます。

Mesh は大きく(1)頂点データ(VertexBuffer)、(2)頂点インデックス(TriangleStripArray)、(3)アピアランス(Appearance)からなります。(1)の VertexBuffer クラスは座標、法線、頂点カラー、テクスチャー座標の 4 種類のデータを保持します。これらのデータはそれぞれデータ保持用の VertexArray オブジェクトで保存されます。(2)の TriangleStripArray クラスは頂点を接続し面(トライアングルストリップ)を作成します。(3)の Appearance は見え方を定義する Material, Fog, CompositingMode, FogMode を保持します。Mesh クラスはこれら 3 つ(VertexBuffer,TriangleStripArray, Appearance)を統合して、シーン中に物体を定義します。以下の節でこれら 3 つの詳しい解説があります。

VertexArray

VertexArray クラスはデータを保持するための汎用的なクラスです。このクラスはデータを保存するのが目的で、そのデータの持つ意味については関与しません。例えば保存しているデータが(1,1,1)として、これが法線なのかカラーなのかについては関与しません。すべてのデータは内部的に確保されたメモリーにコピーされます。Desktop-M3G の場合は同時に GPU に転送されます。従ってセットした後は直ちに元データを開放してかまいません。VertexArray に関しては次節の”データのエンコード”を理解することが重要です。データは 1 バイト(char 型)か 2 バイト(short 型)か 4 バイト(float 型)で保存されます。unsigned 型はありません。

> 補足: VertexArray のデータの型
>
> M3G1.1 の仕様では VertexArray クラスの持つデータは char 型と short 型のみで float 型がありません(2.0 で仕様に入りました)。1.1 が策定された当時の携帯電話はコストのかかかる float 型の使用がまだ一般的でなかったためです。Desktop-M3G では 2.0 の仕様を先取りする形で float 型を取り込んでいます。互換性を重要視する場合は使用してはいけません。

データのエンコード

VertexArray クラスにセットされたデータは(M3G 内部で)使用されるときに必ず scale 値と bias 値で変調されます。

$$value' = value * scale + bias[i]$$

scale 値はコンポーネントで共通で、bias 値はコンポーネントごとに指定します。変調しない場合は scale=1, bias={0,0,0}(コンポーネント数が 3 の場合)です。例えば char 型(-128～127)のデータを(-1.0～1.0)にマッピングするには scale=2/255, bias=1/255 を使用します。

$$value' = (value * 2 + 1)/255$$

このようなエンコードを使う理由は上記のように[-1.0, 1.0]の範囲しか使用しないデータの場合は、そのまま 4 バイトの float 型で保持すると無駄が多いからです。精度がさほど必要とされない状況では 1 バイトの char 型で十分に表すことが出来ます。これはしばしば法線の表現に使われます。なお scale, bias の値は VertexArray クラスが直接保存するのではなく VertexBuffer クラスが必要に応じて保有しています。

VertexArray クラスのコンストラクタは(1)頂点数、(2)コンポーネント数、(3)コンポーネントサイズの 3 つを指定して呼び出します。

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

引数の num_vertices には 1 以上の頂点数を指定します。num_components には 1 頂点あたりのコンポーネント数を 2,3,4 の中から指定します。component_size にはデータのサイズ(型)を 1,2,4 バイトの中から指定します。それぞれ char 型、short 型、float 型を表します。例えば 1 頂点あたり 3 要素(x,y,z)を持ち 100 頂点数を float 型で指定する場合は num_vertices = 100, num_components = 3, component_size = 4 を指定します。インスタンス化直後のデータは未定義です。

設定されている頂点数を取得するには getVertexCount()関数を使用します。

```
int getVertexCount () const;
```

戻り値はコンストラクタで指定した頂点数です。

設定されているコンポーネント数を取得するには getComponentCount()関数を使用します。

```
int getComponentCount () const;
```

戻り値はコンストラクタで指定したコンポーネント数です。2,3,4 のいずれかが返ります。

設定されているコンポーネントサイズ(型)を取得するには getVertexCount()関数を使用します。

```
int getComponentType () const;
```

戻り値はコンストラクタで指定したコンポーネントサイズ(型)です。1,2,4 バイトのいずれかが返ります。

VertexArray にデータをセットするには set 関数を使います。コンポーネントサイズに応じて 3 種類存在します。

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

引数の first_vertex 番目の頂点から first_vertex + num_vertices 番目の頂点まで values の指すから値を取得してコピーします。頂点のインデックスは 0 から getVertexCount()-1 までの範囲が有効です。
values にはコンポーネントサイズが 1 バイトの時は char 型を、2 バイトの時は short 型を、4 バイトの時は float 型を使用します。値は直ちに内部で確保したバッファーにコピーされるのでこの呼び出しが返ったら即開放して構いません。Desktop-M3G ではデータをさらに GPU にも転送します。
セットされたデータを取得するには get()関数を使用します。set()関数と同様コンポーネントサイズに応じて 3 種類存在します。

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

引数の意味は set()関数の時と同じで first_vertex 番目の頂点から first_vertex + num_vertices 番目の頂点まで、今度は逆に values の指すメモリ領域に書き込まれます。values は十分な大きさのメモリ(頂点数 x コンポーネント数 x コンポーネントサイズ)を確保していなければいけません。
VertexArray クラスにはデータの設定・取得以外の機能はありません。

VertexBuffer

VertexBuffer クラスは頂点のデータ(位置、法線、カラー、テクスチャー座標)を保持する VertexArray オブジェクトを持つコンテナクラスです。

VertexBuffer クラスのコンストラクタは引数なしで呼び出します。

```
VertexBuffer ();
```

位置データをセットするには setPositions()関数を使用します。位置データは必須です。

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

引数の positions には位置情報を持つ VertexArray オブジェクトへのポインターを渡します。コンポーネント数は 3 です。scale と bias にはこの VertexArray クラスの持つデータを変調するのに使用されるスケール値、バイアス値を指定します。バイアス値はコンポーネント数個の float の配列へのポインターです。
設定されている位置情報を取得するには getPositions()関数を使用します。

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

引数 scale_bias には結果を書き込むメモリ領域を指定します。コンポーネント数は必ず 3 なので最低 4 個の float 領域が必要です。0 番目に scale 値が書き込まれ、1、2、3 番目に bias 値が書き込まれます。
scale_bias が NULL の場合は何も書き込まれません。戻り値は位置情報を持つ VertexArray オブジェクトへのポインターです。設定されていない場合は NULL が返ります。
法線データををセットするには setNormals()関数を使用します。

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

引数の normals には法線データを持つ VertexArray オブジェクトへのポインターを渡します。コンポーネント数は 3 です。法線データの x,y,z 成分はそれぞれ[-1,1]の範囲の値で指定します。char 型もしくは short 型の場合その型で表せる一番小さな負の整数値が-1.0 に、一番大きな正の整数値が 1.0 にマッピングされます。float 型の場合は[-1.0, 1.0]でそのままマッピングされます。scale,bias の値は下記の表のとおりに自動設定されるのでユーザーが指定する事はありません。

データの型	データの値	scale 値(自動設定)	bias 値(自動設定)
char	-128 ～ 127	2/255	1/255
short	-32768 ～ 32767	2/65535	1/65535
float	-1.0 ～ 1.0	1	0

表 4:法線の指定

設定されている法線データを取得するには getNormals()関数を使用します。

```
VertexArray* getNormals (float* scale_bias=0) const;
```

引数の scale_bias には(自動設定された)スケール・バイアス値を書き込むメモリ領域を指定します。float4 個分のメモリ領域が必要です。0 番目に scale 値が、1,2,3 番目 bias 値が書き込まれます。scale_bias が NULL 場合は何も書き込まれません。戻り値は法線データを持つ VertexArray オブジェクトへのポインターです。設定されていない場合は NULL が返ります。

> 補足: 自動設定されるスケール・バイアス値
>
> オリジナルの M3G API は引数の scale_bias がなく、自動設定された scale, bias 値を取得する手段がありません。値は表の通り自明ですが Desktop-M3G の実装では引数の scale_bias を拡張してライブラリ側から取得できるようにしています。

頂点カラーデータをセットするには setColors()関数を使用します。頂点カラーを有効にするとマテリアルを無視して頂点に設定された色が使用されます。頂点カラーを有効にするには第 6 章アピアランスの Material の項を参照して下さい。Material:: setVertexColorTrackingEnable()関数を使って頂点カラーを有効にする必要があります(ここで頂点カラーのデータをセットしただけでは有効になりません)。

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

引数の colors には頂点カラーデータを持つ VertexArray オブジェクトへのポインターを渡します。コンポーネ

ント数は 3 または 4 です。コンポーネントサイズは 1(char 型)です。頂点カラーとして使用する場合に限りデータを unsigned char 型[0, 255]で確保し、char 型にキャストしてライブラリに渡します。

```
unsigned char colors[3] = {255, 255, 255};    // 白
varray->set (0, 1, (char*)colors);
```

ライブラリはこれを scale=1/255, bias=0 で変調し[0, 1]にマッピングします。

$$c' = c * 1/255 + 0$$

頂点カラーデータでは scale, bias 値は下記の表どおりに自動で設定されます。

データ型	データの値	scale 値(自動設定)	bias 値(自動設定)
unsigned char	0 ～ 255	1/255	0/255

表 5:頂点カラーの指定

設定されている頂点カラー情報を取得するには getColors()関数を使用します。

```
VertexArray* getColors (float* scale_bias=0) const;
```

引数の scale_bias には自動設定された scale, bais 値が書き込まれます。4 個または 5 個の float 領域が必要です。0 番目に scale 値が書き込まれ、1,2,3,4 番目 bias 値が書き込まれます。scale_bias が NULL 場合は何も書き込まれません。戻り値は頂点カラー情報を持つ VertexArray オブジェクトへのポインターです。設定されていない場合は NULL が返ります。

補足: オリジナルとの違い

Java 言語の 1 バイト型は byte 型のみで符号なしバージョンがありません。従ってオリジナルの M3G ではカラーを[-128, 127]で指定します。これは色の表現として非常に違和感がありC++言語の Desktop-M3G では[0, 255]に変更しています。

テクスチャー座標データをセットするには setTexCoords()関数を使用します。テクスチャー座標はマテリアルでテクスチャーが有効にされた時に参照されます。テクスチャーについては第 6 章を参照して下さい。

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

引数の index には 0 から始まるテクスチャーユニット番号を指定します。GPU が対応していれば 4 つまで使用可能です。引数の tex_coords にはテクスチャー座標情報を持つ VertexArray オブジェクトへのポインターを渡します。コンポーネント数は 2 です。scale と bias にはデータを変調するのに使用されるスケール値、バイアス値を指定します。バイアス値はコンポーネント数個の float の配列へのポインターです。

設定されているテクスチャー座標データを取得するには getTexCoords()関数を使用します。

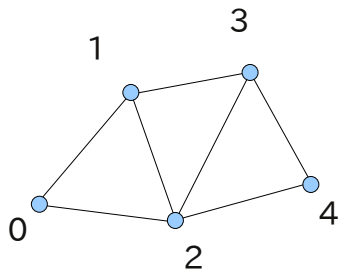
```
VertexArray* getTexCoords (int index, float* scale_bias) const;
```

引数の index には 0 から始まるテクスチャーユニット番号を指定します。引数 scale_bias には結果を書き込むメモリ領域を指定します。最低 3 個の float 領域が必要です。0 番目に scale 値が、1,2 番目に bias 値が書き込まれます。scale_bias が NULL の場合は何も書き込まれません。戻り値は位置情報を持つ VertexArray オブジェクトへのポインターです。設定されていない場合は NULL が返ります。

TriangleStripArray

IndexBuffer クラスは頂点の集合を線で接続し面を形成します。TriangleStripArray クラスは IndexBuffer クラスから派生しトライアングルのストリップと呼ばれる一筆書きで面の集合を定義します。M3G ではすべての面をトライアングルストリップで表し、これ以外の IndexBuffer の派生クラスはありません。すべての面を一回で一筆書きするのは不可能なのでこのクラスは複数のストリップをまとめて扱います。M3G

ではトライアングルストリップ以外の面形式、例えばトライアングルファンやクアドストリップには対応していません。



図形 1:トライアングルストリップの例

TriangleStripArray クラスのコンストラクタは以下の 2 つの形式があります。

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

引数の num_strips はストリップの本数を指定します。引数の strips はストリップの長さを並べた 1 次元配列です。indices は全てのストリップで使用する頂点インデックスを並べた 1 次元配列です。以下の例ではストリップ 2 本でそれぞれ長さ(頂点数)が 8 のストリップを使って TriangleStripArray オブジェクトを作成しています。

```
int indices[16] = {0,1,2,3,4,5,6,7, 8,9,10,11,12,13,14,15}
int num_strips = 2;
int strips = {8, 8};
TriangleStripArray* tris = new TriangleStripArray (indices, num_strips, strips);
```

2 番目の形式は頂点番号を配列で明示的に指定する代わりに first_index から自動的に+1 づつインクリメントしながら使用します。頂点番号を順番に使用するときはこちらの形式のコンストラクタを使用した方がいいかもしれません。Desktop-M3G の実装ではどちらを使用しても内部的に違いはありません。

この TraiangleStripArray クラスが定義する面の個数を取得するには getFaceCount()関数を使用します。

```
virtual int getFaceCount () const;
```

戻り値は面の総数です。これは全てのストリップが定義する面の合計です。

指定した頂点インデックスの個数を取得するには getIndexCount()関数を使用します。

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

戻り値は頂点インデックスの個数です。これはコンストラクタで指定した strips 配列の合計値です。

1 つの面あたりの頂点数を取得するには getFaceVertexCount()関数を使用します。

```
virtual int getFaceVertexCount () const;
```

戻り値は頂点数です。トライアングルなので必ず 3 が返ります。

面を構成するインデックスを取得するには getFaceVertexIndex()関数を使用します。

```
virtual void getFaceVertexIndex (int face_index, int* vertex_indices) const;
```

引数 face_index には 0 から始まる面インデックスを指定します。最大値は getFaceCount()-1 です。vertex_indices には結果を書き込むメモリ領域です。getFaceVertexCount()個の float の配列を指している必要があります。トライアングルなら必ず 3 です。

Mesh

作成した VertexBuffer オブジェクトと IndexBuffer オブジェクトを Mesh クラスのコンストラクタに渡してインスタンス化します。モデリングの手順をまとめると、

1. 位置データを持った VertexArray オブジェクトを用意する
    1. 同様に法線データを用意する(オプション)
    2. 同様に頂点カラーデータを用意する(オプション)
    3. 同様にテクスチャー座標データを用意する(オプション)
2. VertexBuffer オブジェクトにセットする
3. 面を構成する TriangleStripArray オブジェクトを作成する
4. アピアランスを定義する Appearance オブジェクトを作成する

5. VertexBuffer オブジェクト, TriangleStripArray オブジェクト, Appearance オブジェクトを使って Mesh ノードを作成しシーンに登録する

となります。

これらの手順を踏まえて頂点数 4、ストリップ 1 本で図 8 の単純な 4 角形を書くコードは以下の通りです。

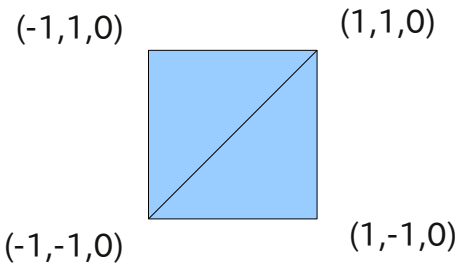


図 8:単純な 4 角形

```
VertexArray* positions         = new VertexArray (4, 3, 2);
short        positions_value[] = {1,-1,0, 1,1,0, -1,-1,0, -1,1,0};
positions->set (0, 4, positions_value);

float scale   = 1;
float bias[3] = {0,0,0};
VertexBuffer* vertices = new VertexBuffer;
vertices->setPositions (positions, scale, bias);

int strips[1] = {4};
int indices[] = {0,1,2,3};
TriangleStripArray* tris = new TriangleStripArray (indices, 1, strips);

Appearance* app = new Appearance;

Mesh* mesh = new Mesh (vertices, tris, app);
wld->addChild (mesh);
```

このコードの実行結果は図 9 のとおりです。

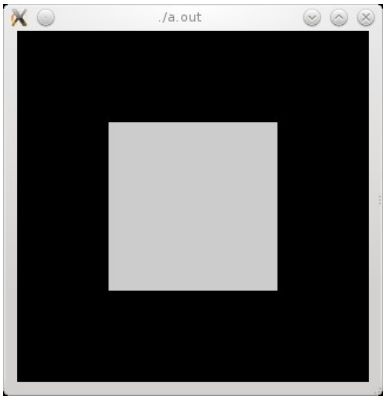


図 9:単純な 4 角形のレンダリング

SkinnedMesh

SkinnedMesh クラスは静的なメッシュを表す Mesh クラスにボーンと呼ばれる構造を追加したシーンノードです。主にゲームのキャラクターを描画するのに使われます。Mesh の頂点はスキンと呼ばれボーンの変形に追従して変形します。これをスケレタルアニメーションと呼びます。

スケレタルアニメーション

SkinnedMesh クラスは通常の Mesh に加えてスケルトンと呼ばれるボーン階層を保有しています。ボーンとは描画されない仮想的な概念で人体における骨と同じものです。ただし正確に言うとプログラムの中で存在するのはボーンではなく関節を表すジョイントです。ジョイントを 2 つ結んだ線がボーンになります。ボーンとジョイントはしばしば混同され区別無く使われます。通常ユーザーレベルで気にする必要はありません。M3G では 1 つのボーン(ジョイント)を 1 つの Group クラスがあらわします。Group はシーンノードとしての Group をそのまま使用します。スケルトンは 1 つのボーンを頂点とする木構造のグラフです。通常頂点のボーンは人体の”へそ”に置かれます。そして”へそ”から手先、足先、首に向かってボーンを伸ばしていく事が多いようです。

ボーンとスキンの変形

ボーンに影響を受ける頂点の集合のことをスキンと呼びます。スキンはボーンの移動、回転に追従して動きます。スキンは頂点ごとに(1)どのボーンに影響を受けるか(インデックス)、(2)どのぐらい影響を受けるか(ウェイト)を持ちます。通常腕などの関節付近のスキンは上腕と下椀の両方のボーンに影響を受けるように値が設定されています。これによりスキンは腕のボーンの変形に追従してなめらかに変形します。

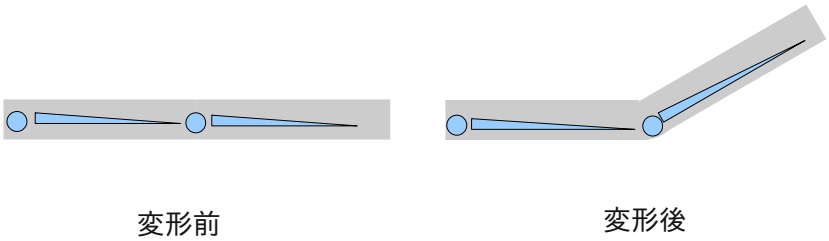


図 10:ボーンの変形とスキンの追従

図 10 はボーンの変形とスキンの追従の様子を模式化したものです。○△印がボーンをあらわします。灰色の領域がスキン(頂点の集合)で、近くのボーンに関連付けられています。関節部分は両方のボーンに関連付けられています。このようにスキンはボーンの変形に追従して移動します。

SkinnedMesh

SkinnedMesh クラスのコンストラクタは引数に Mesh クラスのコンストラクタと同じ VertexBuffer オブジェクト、IndexBuffer オブジェクト, Appearance オブジェクトに加えて Group ノードのスケルトンを取ります。Mesh クラスと同様に IndexBuffer と Appearance は複数指定できます。

```
SkinnedMesh (VertexBuffer* vertices,
                              int                    num_submesh,
                              IndexBuffer**  submeshes,
                              Appearance** apperarances,
                              Group*            skeleton);
SkinnedMesh (VertexBuffer* vertices,
                             IndexBuffer*   submeshes,
```

```
                    Appearance*  apperarances,
                    Group*             skeleton);
```

インスタンス化直後の姿勢をバインドポーズと呼びます。キャラクターの場合多くは腕を横にまっすぐ伸ばしたT字型を採用することが多いと思われます。バインドポーズはスケルトンに設定されたアニメーションに追従して変形します。適切に設定されたアニメーションを再生することでキャラクターは歩いたり走ったりする事ができます。

設定したスケルトンを取得するには getSkeleton()関数を使用します。

```
Group* getSkeleton () const;
```

戻り値はスケルトンのルートノードです。

SkinnedMesh をインスタンス化した直後はまだボーンとスキンの関係(インデックス、ウェイト)が記述されていません。この状態でスケルトンを動かしてもスキンは追従しません。ボーンとスキンを関係付けるには addTransform()関数を使います。

```
void addTransform (Node* bone, int weight, int first_vertex, int num_vertices);
```

引数の first_vertex 番目の頂点から num_vertices 個の頂点を bone ノードにたいしてウェイト値 weight で関連付けます。ボーンはスケルトンで指定したボーン階層に含まれている必要があります。ウェイト値は 1 以上の正の整数値で指定します。レンダリング時にウェイト値は関連付けられたボーン全ての間で[0,1]に正規化されます。

設定されたウェイト値を取得するには getBoneVertices()関数を使用します。

```
int getBoneVertices (Node* bone, int* vertex_indices, float* weights) const;
```

この関数は引数の bone ノードに関連付けられている全ての頂点インデックスとウェイトを取得します。第2引数の vertex_indices と第 3 引数の weights はそれぞれ問い合わせ結果を書き込むメモリ領域です。このボーンが影響を及ぼす頂点の数と同じだけの配列を確保してある必要があります。NULL が指定された場合は書き込みません。通常この関数は 2 回呼び出します。つまり 1 回目は vertex_indices と weights を NULL にして呼び出し、戻り値から影響を及ぼす頂点数を取得しメモリを確保します。2 回目は確保したメモリを渡してインデックスとウェイトを取得します。ウェイト値は[0,1]に正規化されています。1 つの頂点が影響を受ける全てのボーンのウェイト値を合計すると 1 になります。

ボーン変換行列を取得するには getBoneTransform()関数を使用します。この関数は SkinnedMesh ノードのローカル座標系から指定のボーン座標系への変換を行う 4x4 行列を取得します。スキンメッシュに関係す

るボーンの変形およびスキンの追従は全てライブラリ内部で自動的に処理されるので、通常ユーザーがこの関数を呼んでボーン変換行列を取得する必要はありません。

```
void getBoneTransform (Node* bone, Transform* transform) const;
```

第 1 引数の bone には対象となるボーンを指定します。第2引数は結果を書き込む Transform オブジェクトを指定します。NULL は指定できません。

補足: ボーン変換行列とは

ここで言うボーン変換行列とはローカル座標系をボーン座標系に変換するのに使用される 4x4 行列(バインドポーズ)を指します。このあたりの用語はツール・書籍類で統一されていないので特に注意が必要です。一般的にスケレタルアニメーションはこの行列を使ってボーン空間に移動した後、カレントポーズを使って変形後のローカル座標の位置を計算します。詳しくは Jason Gregory の Game Engine Architecture をご覧下さい。筆者が今まで読んだスケレタルアニメーションの解説の中ではこの本が最良です。

補足: 頂点データとしてボーンの情報(インデックス、ウェイト)を持たせたら?

多くの 3D ライブラリでは法線やテクスチャー座標と同様に頂点データとしてボーンのインデックスとウェイト値を持たせているようです。M3G で言えば VertexBuffer クラスに新しいボーン情報用の VertexArray を持たせます。API 上も実装上もその方が簡単なのですが、なぜ M3G1.1 でこのような仕様が採用されたのかはよく分かりません。ちなみに現在ドラフト段階の M3G2.0 の仕様書を読むと上記の一般的な方式に変更されたようです。

スケルトンのボーンとして使われる Group ノードはシーン中で使われる Group ノードと同じものです。スケルトンとして渡された Group ノードの下には任意のノードを挿入可能です。挿入されたノードはレンダリング時に子ノードと同様に描画されます。これは例えばゲームでキャラクター(SkinnedMesh クラス)が手に武器(Mesh クラス)を持つ時に使われます。武器を変更する場合はこの Mesh クラスのみを差し替えます。

MorphingMesh

MorphigMesh クラスは Mesh クラスにモーフィング機能を追加したシーングラフノードです。複数のモーフタッゲットを合成して表示します。典型的には顔の表情をフォーフィングしたり、簡易のキャラクターアニメーションを作成するのに使われます。

モーフィングの仕組み

モーフィングは複数のターゲット(メッシュ)をウェイト値をかけて合成します。ウェイト値を変化させることでターゲット A からターゲット B になめらかに形状が変化します。モーフターゲットは複数いくらでも指定可能です。

MorphingMesh

MorphingMesh クラスのコンストラクタは Mesh クラスに必要な VertexBuffer オブジェクト, IndexBuffer オブジェクト, Appearance オブジェクトに加えてモーフターゲット用の VertexBuffer オブジェクトを取ります。IndexBuffer オブジェクトと Appearance オブジェクトを 1 つ、または複数とるのも Mesh クラスと同様です。

```
MorphingMesh (VertexBuffer*    base,
              int                      num_target,
              VertexBuffer**  targets,
              int                      num_submesh,
              IndexBuffer**    submeshes,
              Appearance**   appearances);

MorphingMesh (VertexBuffer*    base,
              int                      num_target,
              VertexBuffer**  targets,
              IndexBuffer*     submesh,
              Appearance*    appearance);
```

引数の base は基本メッシュとなる VertexBuffer オブジェクトを渡します。モーフタゲットの個数は num_target で指定します。モーフターゲットとなる頂点データの配列は targets で指定します。頂点インデックス(submesh)とアピアランス(appearance)は基本メッシュとターゲットメッシュで共通に使われます。つまりトポロジーは必ず等しくなります。基本メッシュの VertexBuffer に存在する VertexArray 要素はターゲットの方にも存在し、コンポーネント数と頂点数が等しくなければなりません。scale と bias 値は基本メッシュの持つ値が使用されます。

設定されたモーフターゲットの個数を取得するには getMorphTargetCount() 関数を使用します。

```
int getMorphTargetCount () const;
```

戻り値はモーフターゲットの個数です。

設定されたモーフターゲットを取得するには getMorphTarget()関数を使います。

```
VertexBuffer* getMorphTarget (int index) const;
```

引数の index は 0 から getMorphTargetCount()-1 までが有効です。

MorphingMesh オブジェクトをインスタンス化した直後はウェイトが設定されてないのでモーフィングできません。このあとモーフターゲット毎にウェイト値を設定する必要があります。ウェイト値を設定するには setWeights()関数を使用します。

```
void setWeights (int num_weights, float* weights);
```

float の配列の引数 weights から num_weights 個の float 値をコピーしてウェイトに設定します。ウェイトの個数は原則ターゲットの個数と同じです。それより少ない場合は 0 で補完されます。多い場合は余剰分は無視されます。全てのモーフターゲットのウェイト値を足すと 1 になるように正規化されているべきですが、必須ではありません。ウェイト値には負の値や 1 を越える値を設定してもかまいません。デフォルトのウェイト値は全て 0 です。

設定したウェイト値は getWeights()関数で取得できます。

```
void getWeights (float* weights) const;
```

引数は weights は結果を書き込む float の配列へのポインターです。モーフターゲットの個数分だけ書き込まれます。

第6章　アピアランス

M3G では色やテクスチャーなど物体の”見え方”に関する項目を総称してアピアランスと呼びます。Appearance クラスは見え方を制御するアピアランス要素(Material, Texture2D, CompositingMode, PolygonMode, Fog)を保持するコンテナクラスです。Mesh などの物体はこれらのアピアランス要素を変更することで幾何情報はそのままに色などの見え方だけ一括して変更することができます。

Material

Material クラスは物体の基本的な色を決定するアピアランス要素です。M3G では OpenGL の固定パイプラインをそのまま利用します。ライトとマテリアルの相互作用(照光計算)はフォンシェーディング・モデルに従って計算されます。シェーディング後のカラーは以下の式で計算されます。

$$頂点の色 = k_{emissive} + k_{ambient} I_{ambient} + \sum k_{diffuse} I_{diffuse} (L \cdot N) + k_{specular} * I_{specular} (V \cdot R)^{shininess}$$

マテリアルの反射係数は k で表記します。発光係数、環境係数、拡散係数、鏡面係数の 4 種類が存在します。

$k_{emissive}$:=発光係数
$k_{ambient}$:=環境係数
$k_{diffuse}$:=拡散係数
$k_{specular}$:=鏡面係数

ライトのカラーは I で表記します。ライトには環境光、拡散光、鏡面光の 3 種類が存在します。

$I_{ambient}$:=環境光
$I_{diffuse}$:=拡散光
$I_{specular}$:=鏡面光

点光源(OMNI)と平行光源(DIRECTIONAL)のライトカラーがそのまま拡散光と鏡面光として使われます。”拡散成分のみを持った光源”は出来ません。ライトの種類の環境光(AMBIENT)のライトカラーが環境光として使われます。

残りの N,R,L,V は方向ベクトルを表します。

N :=*法線ベクトル*
R :=*ライトの反射方向へ向かうベクトル*
L :=*ライトの入射方向へ向かうベクトル*
V :=*ビュー方向へ向かうベクトル*

ベクトルは全てシェーディング地点を起点とする正規化されたベクトルです。

shininess は鏡面指数で、大きいほど鏡面反射が鋭くなります。鏡面成分の計算は使用中の GPU によって多少違う計算式で計算される可能性があります。

Material クラスは上記の計算式のマテリアルの各項目、$k_{emissive}$(発光係数), $k_{ambient}$(環境係数), $k_{diffuse}$(拡散係数), $k_{specular}$(鏡面係数), shinisess(鏡面指数係数)を制御します。各要素は RGB3 色で表されます。拡散色のみ A 成分を持ち、これが物体の不透明度 α となります。

Material クラスのコンストラクタは引数なしで呼び出します。

```
Material ();
```

デフォルトのマテリアルは下記の色(係数)に設定されています。

$k_{emissive}$ =000000000
$k_{ambient}$ =0x00333333
$k_{diffuse}$ =0xffcccccc
$k_{specular}$ =0x00000000
$shininess$ =0

注意点はデフォルトの拡散色は白(0xffffffff)ではなく灰色(0xffcccccc)な事です。0xcc=204 なのでデフォルトは 80%の光を反射します。デフォルトマテリアルで描画すると想定したよりも暗く表示される可能性があります。

マテリアルのカラーを変えるには setColor()関数を使用します。

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

引数の target には AMBIENT, DIFFUSE, EMISSIVE, SPECULAR の中から 1 つを指定します。argb には設定したいカラーを int 型で指定します。各チャンネルは 0～255 で指定し 1 つの整数値にパックします。

```
argb = a << 24 | r << 16 | g << 8 | b << 0
```

A 成分は拡散色でのみ有効です。ほかのターゲットの場合は無視されます。

現在設定されているカラーを取得するには getColor()関数を使用します。

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

引数の target には同様に AMBIENT, DIFFUSE, EMISSIVE, SPECULAR の中から 1 つを指定します。

戻り値がそのカラーです。

鏡面項の計算で使われる指数係数を変更するには setShininess()関数を使用します。

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

引数の shininess には[0,128]の範囲の値を指定します(0 と 128 を含む)。値が大きいほど鏡面反射成分は鋭くなります。デフォルトは 0 です。

現在設定されている指数係数を取得するには getShininess()関数を使用します。

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

戻り値は鏡面指数係数です。

ライティングが有効な時は、頂点の色は前述のフォンシェーディングによって計算された値が使用されますが、頂点カラーと呼ばれる機能を有効にするとライトやマテリアルの設定を無視して頂点に設定された色がそのまま頂点の色になります。頂点カラーの詳細は第 5 章の VertexBuffer に関する解説を参照して下さい。頂点カラーを有効にするには setVertexColorTrackingEnable()関数を使用します。

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

引数 enable が true のときは有効化されます。false の時は無効化されます。デフォルトは false です。

頂点カラーを有効にすると、この物体に関してのライティング計算は行われず設定された頂点カラーがそのまま使用されます。

現在の頂点カラーの設定を取得するには isVertexColorTrackingEnabled()関数を使用します。

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

戻り値は有効な時は true で、無効な時は false です。

Texture2D

Texture2D はテクスチャー設定に関するアピアランス要素です。使用している GPU が対応していればテクスチャーは同時に 4 枚まで使用可能です。

Texture2D クラスはテクスチャーとして使用する画像を持った Image2D オブジェクトを引数にしてインスタンス化します。

```
Texture2D (Image2D* image);
```

引数の image は Image2D オブジェクトのポインターを渡します。NULL はエラーとなります。テクスチャーとして使用する画像のサイズは縦横ともに 2 の階乗でなければいけません。それ以外のサイズは GPU が対応していたとしても受け付けません。

設定した画像を差し替えるには setImage()関数を使用します。

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

引数の image は設定したい Image2D オブジェクトのポインターです。NULL は指定できません。画像サイズは 2 の階乗でなければいけません。

設定されている画像を取得するには getImage()関数を使用します。

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

戻り値は Image2D オブジェクトのポインターです。

テクスチャーのブレンド関数を設定するには setBlending()関数を使用します。ブレンド関数は下地の色(シェーディング計算の実行結果)にテクスチャーの色をどのように適応するかを決定します。

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

引数の func には表 6 に示す FUNC_BLEND、FUNC_DECAL、FUNC_MODULATE、FUNC_REPLACE の中から 1 つを選択します。デフォルトでは FUNC_MODULATE です。FUNC_REPLACE は下地を無視してテクスチャーで置き換えます。FUNC_DECAL は下地の上にテクスチャーを張り付けます。テクスチャーの透明な部分だけ下地が透けて見えます。FUNC_MODULATE はもっとも一般的なテクスチャー関数で下地とテクスチャーでモジュロ演算を行い変調した色を出力します。FUNC_ADD は加算合成で下地にテクスチャーを加算します。GL_BLEND は下地の色と後述のブレンドカラーをテクスチャーの色で分配して出力します。これらのテクスチャーブレンド関数は基本的に OpenGL のそれと同じです。計算式はテクスチャーのフォーマットによっても多少異なります。詳細な計算方法は表 6 を参照して下さい。図中でカラー成分(A を含まず)は C で、α 値は A で表記されます。下地の色(フラグメントカラー)は添字として f を付けて C_f と A_f と表記されます。同様にテクスチャーの色は C_t と A_t と表記されます。ブレンド後の色は C_v と A_v で表記されます。

	ALPHA	LUMINANCE	LUMINANCE_ ALPHA	RGB	RGBA
FUNC_REPLACE	$C_v=C_f$ $A_v=A_t$	$C_v=C_t$ $A_v=A_f$	$C_v=C_t$ $A_v=A_t$	$C_v=C_t$ $A_v=A_f$	$C_v=C_t$ $A_v=A_t$
FUNC_DECAL	undefined	undefined	undefined	$C_v=C_t$ $A_v=A_f$	$C_v=C_f(1-A_t)+C_tA_t$ $A_v=A_f$
FUNC_MODULATE	$C_v=C_f$ $A_v=A_fA_t$	$C_v=C_fC_t$ $A_v=A_f$	$C_v=C_fC_t$ $A_v=A_fA_t$	$C_v=C_fC_t$ $A_v=A_f$	$C_v=C_fC_t$ $A_v=A_fA_t$
FUNC_ADD	$C_v=C_f$ $A_v=A_fA_t$	$C_v=C_f+C_t$ $A_v=A_f$	$C_v=C_f+C_t$ $A_v=A_fA_t$	$C_v=C_f(1-C_t)+C_cC_t$ $A_v=A_f$	$C_v=C_f(1-C_t)+C_cC_t$ $A_v=A_fA_t$
FUNC_BLEND	$C_v=C_f$ $A_v=A_fA_t$	$C_v=C_f(1-C_t)+C_cC_t$ $A_v=A_f$	$C_v=C_f+C_t$ $A_v=A_fA_t$	$C_v=C_f(1-C_t)+C_cC_t$ $A_v=A_f$	$C_v=C_f(1-C_t)+C_cC_t$ $A_v=A_fA_t$

表 6:ブレンド関数

マルチテクスチャーの場合は図 11 のようにテクスチャーユニット毎にブレンド関数を設定します。

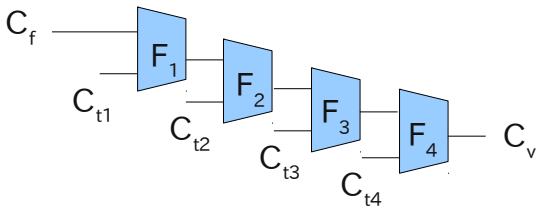


図 11:マルチテクスチャー

現在設定されているブレンド関数を取得するには getBlending()関数を使用します。

```
int getBlending () const;
```

戻り値は表 6 のブレンド関数のいずれかです。

フィルタリング関数を設定するには setFiltering()関数を使用します。

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

引数の level_filter は LOD 階層(ミップマップ)のフィルタリングモードを FILTER_BASE_LEVEL, FILTER_NEAREST, FILTER_LINEAR の中から選択します。BASE_LEVEL を選択するとミップマップ処理を行いません。NEAREST は一番近い LOD 階層の画像を使用します。LINEAR は現在の詳細度の両側の LOD 階層の画像を線形補間して使用します。ミップマップの処理はすべて GPU(もしくはドライバー)が行います。使用中の GPU によってはここでの設定に関わらず BASE_LEVEL が使われることがあります。引数の image_filter には画像内のフィルタリングモードを NEAREST と LINEAR の中から選択します。NEAREST は一番近いテクセルの値をそのまま使用します。LINEAR は現在のテクスチャー座標の周辺 4 テクセルの線形補間された値を使用します。

レベルフィルター	説明
BASE_LEVEL	LOD を使用しません
NEAREST	もっとも近い LOD 階層を使用します
LINEAR	LOD 階層の近景側と遠景側の 2 階層を線形補間します

表 7:レベルフィルター

イメージフィルター	説明
NEAREST	最近傍ピクセルの値を使用します
LINEAR	周辺 4 ピクセルの値を線形補間します

表 8:イメージフィルター

デフォルトは level_filter = FILTER_BASE_LEVEL, image_filter = FILTER_NEAREST_LEVEL に設定されています。つまり LOD を使用せず、最近傍ピクセルの値を使用します。画質はよくありません。通常は LOD を使用し線形補間を行うよう変更すべきです。

現在のフィルタリング関数を取得するには getLevelFilter(), getImageFilter()関数を使用します。

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

それぞれ表 7、表 8 のフィルタリング関数を返します。

テクスチャーのラッピングモードを切り替えるには setWrapping()関数を使用します。

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

引数にはそれぞれ s 方向、t 方向のラッピングモードを WRAP_CLAMP、WRAP_REPEAT の中から選択します。ラッピングモードはテクスチャー座標(s,t)が(0,0)～(1,1)の範囲外の場合の処理設定です。
WRAP_CLAMP は一番近い画像テクセルの値を使用します。これは最外周のテクセルを引き伸ばした事と同じです。WARP_REPEAT は画像がタイル状に繰り返し並んでいる物として扱います。

ラッピングモード	説明
WRAP_CLAMP	最外周ピクセルの値を使用します
WRAP_REPEAT	画像を仮想的に繰り返します

表 9:ラッピングモード

デフォルトは s 方向、t 方向ともに WRAP_REPEAT です。

現在設定されているラッピングモードを取得するには getWrappingS(), getWrappingT()関数を使用します。

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

それぞれ s 方向、t 方向のラッピングモードを表 9 の定数値で返します。

ブレンドカラーを設定するには setBlendColor()関数を使います。ブレンドカラーはブレンド関数が FUNC_BLEND の時に使用されます。それ以外の時は影響はありません。

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

引数 rgb はブレンドカラーを RGB 形式で指定します。A 要素は安全に無視されます。デフォルトのブレンドカラーは 0x000000 です。

現在設定されているブレンドカラーを取得するには getBlendColor()関数を使用します。

```
int getBlendColor () const;
```

戻り値がブレンドカラーです。

Fog

Fog はフォグ要素を表すアピアランスの構成要素です。フォグは物体に対してカメラからの距離に応じたフォグカラーをブレンドします。フォグには線形フォグ(LINEAR)と指数フォグ(EXPONENTIAL)があります。両者のフォグ係数は以下の計算式で計算されます。

線形フォグ

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

指数フォグ

$$f = e^{-dz}$$

z はカメラ空間でのカメラから物体までの距離です。d(指数係数)、near(ニア距離)、far(ファー距離)は設定可能な任意の定数値です。フォグ係数 f は[0,1]にクランプされ、物体のカラー(C_s)とフォグカラー(C_f)のブレンドに使用されます。

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

Fog クラスのコンストラクタは引数なしで呼び出します。

```
Fog ();
```

フォグの種類を設定するには setMode()関数を使用します。

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

引数の mode には LINEAR(線形フォグ)と EXPONENTIAL(指数フォグ)のどちらかを選択します。デフォル

トは LINEAR です。

現在のモードを取得するには getMode()関数を使用します。

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

戻り値は LINEAR もしくは EXPNENTIAL が返ります。

フォグカラーを設定するには setColor()関数を使用します。

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

引数 rgb には設定したいフォグカラーを RGB 形式で指定します。A 要素は無視されます。デフォルトのフォグカラーは 0x00000000 です。

現在設定されているフォグカラーを取得するには getColor()関数を使用します。

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

戻り値は RGB 形式のフォグカラーでです。

線形フォグの係数を設定するには setLinear()関数を使用します。

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

引数には線形フォグの計算で使われるニア距離(near)とファー距離(far)を指定します。デフォルトは near=0, far=1 です。これらの係数はモードが LINEAR の場合のみ有効です。z=near の時にフォグ係数は 1.0 になりフォグカラーは加算されず、z=far の時にフォグ係数は 0.0 になり物体は完全にフォグカラーのみで描画されます。通常は near < far ですが、そうでなくても問題ありません。near == far はエラーです。

現在の設定値を取得するには getNearDistance(), getFarDistance()関数を使用します。

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

戻り値はそれぞれニア距離、ファー距離です。

指数フォグの係数を設定するには setDensity()関数を使用します。

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

引数 density には指数係数を 0 以上の実数で指定します。値が大きい方がフォグが濃くなります。この係数はフォグモードが EXPONENTIAL の場合のみ有効です。デフォルトは 1.0 です。

CompositingMode

CompositingMode クラスはピクセル単位の特殊処理を集めたアピアランス要素です。デプステストの可否や各種バッファーへの書き込みの制御、デプスオフセットなどの処理を行います。

CompositingMode クラスのコンストラクタは引数なしで呼び出します。

```
CompositingMode ();
```

フラグメントをフレームバッファー(カラーバッファー)へ書き込む時のブレンド処理を変更するに setBlending()関数を使用します。

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

引数の mode には 表 10 の REPLACE、ALPHA、ALPHA_ADD、MODULATE、MODULATE_X2 の中から 1 つを選択します。デフォルトは REPLACE です。図中で C はカラー(A 成分を含む)を表し、A は α 成分のみを表します。添字の s はソースを表し、d はデスティネーションを表します。ソースはフラグメントカラーでデスティネーションはフレームバッファーです。

書き込みモード	説明
REPLACE	$C_d = C_s$
ALPHA	$C_d = C_s A_s + C_d(1 - A_s)$
ALPHA_ADD	$C_d = C_s A_s + C_d$
MODULATE	$C_d = C_s C_d$
MODULATE_X2	$C_d = 2C_s C_d$

表 10:ブレンドモード

REPLACE 以外のブレンドモードは特殊効果を得るのに使用されます。例えば半透明物体の描画にはブレンドモードを ALPHA に設定しフレームバッファーとフラグメントカラーを α 値で合成します。

現在のブレンドモードを取得するには getBlending()関数を使用します。

```
int getBlending () const;
```

戻り値は表 10 のブレンドモードです。

デプステストの有効・無効の設定を変更するには setDepthTestEnable()関数を使用します。

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

引数 enable にはテストを行う場合は true を、行わない場合は false を指定します。デフォルトは true です。デプステストは必ず OpenGL で言う GL_LEQUAL で実行されます。すなわち現在のデプスバッファーの値よりｚが等しいか手前にある場合デプステストを通過し、それ以外の場合は拒絶されます。フラグメントのｚは視体積のニアクリップ面をｚ=0、ファークリップ面をｚ=1 として、その中間の値を持ちます。初期状態でデプスバッファーは最奥の値 1.0 に初期化されています。

デプステストを行うには M3G の使用に先立ちデプスバッファーが有効化されている必要があります。例えば glut を使用している場合 glutInitDisplayMode()関数を呼び出すときにデプスバッファーを有効にします。

```
glutInitDisplayMode (GLUT_RGBA | GLUT_DEPTH);
```

デプスバッファーが有効化されていない状態でデプステストを有効にした時の動作は未定義です。

> 補足: デプスバッファーが有効化されていないときのデプステストの動作
>
> 筆者が手元のマシンで試したところ NVIDIA 系のチップでは動作し ATI 系のチップで動作しませんでした。必ずしも想定外のコードで動作した(ように見える)NVIDIA の方が良い分けではありません。個人的には間違えたコードを書いたら間違えた結果が得られる ATI の動作の方が好きです。

現在のデプステストの有効・無効の確認は isDepthTestEnabled()関数を使用します。

```
bool isDepthTestEnabled () const;
```

戻り値は有効な時 true、無効な時 false です。

デプステストを通過したフラグメントの、デプスバッファーへの書き込みは setDepthWriteEnable()関数で制御します。

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

引数 enable が true の時は書き込みを許可し、false の時は不許可にします。デフォルトは true です。例えば半透明物体を描画をする時はデプスバッファーへの書き込みを禁止にします。

現在のデプスバッファーへの書き込みを確認するには isDepthWriteEnabled()関数を使用します。

```
bool isDepthWriteEnabled () const;
```

戻り値は書き込みが許可されている時は true で、不許可の時は false です。

カラーバッファー(RGB)への書き込みは setColorWriteEnalbe()関数で制御します。この関数は RGB の

みを制御し A チャネルは無関係です。A 成分の制御には後述の setAlphaWriteEnable()関数を使用します。

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

引数 enable が true の時はカラーバッファー(RGB)へ書き込まれ、false の時は書き込まれずに棄却されます。デフォルトは true です。

現在のカラーバッファー(RGB)への書き込み許可の取得は isColorWriteEnabled()関数を使用します。

```
bool isColorWriteEnabled () const;
```

戻り値は書き込みが許可されている時は true で、不許可の時は false です。

カラーバッファー(A)への書き込み許可の変更は setAlphaWriteEnable()関数を使用します。この関数は RGB チャネルとは無関係です。

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

引数が true の時はカラーバッファー(A)へ書き込まれ、false の時は書き込まれずに棄却されます。デフォルトは true です。

現在のカラーバッファー(A)への書き込み許可を取得するには isAlphaWriteEnabled()関数を使用します。

```
bool isAlpaWriteEnabled () const;
```

戻り値は書き込みが許可されている時は true で、不許可の時は false です。

α テストの設定を変更するには setAlphaThreshold()関数を使用します。α テストは OpenGL で言う所の GL_GEQUAL で行います。すなわちフラグメントの α 成分が、設定値と等しいかそれ以上の時受け入れられ、それ以外の時は棄却されます。

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

引数 threshold は α テストのしきい値を[0,1]で指定します。0 に設定すると α テスト自体行われず全てのピクセルが常に受け入れられます。デフォルトは threshold=0 です。

現在設定されている α テストのしきい値を取得するには getAlphaThreshold()関数を使用します。

```
float getAlphaThreshold () const;
```

戻り値はしきい値です。

デプスバッファーの精度や計算精度の問題から描画時に極めて接近した 2 つの物体の前後関係が正しく計算されず、表示が乱れることがあります(z ファイティング)。この問題を避けるためにデプスオフセットと呼ばれる機能が用意されています。

デプスオフセットの設定には setDpethOffset()関数を使用します。

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

オフセット値は引数の factor と units を使用して下記の式で計算されます。

$$offset = m * factor + r * units$$

m と r は自動で決定される定数です。m はトライアングルの z 方向の最大勾配で、r はデプスバッファーの最小精度です。それぞれ factor と units に積算されます。計算された offset はウィンドウ座標でデプステストの前にフラグメントの z に加算されます。

現在のオフセット係数を取得するには getDepthOffsetFactor(), getDpethOffsetUnits()関数を使用します。

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

戻り値は上記オフセット計算で使われる factor, units です。

PolygonMode

PolygonMode はポリゴンレベルでの特殊処理を集約したアピアランス要素です。主にカリング・シェーディング方式の設定を行います。

PolygonMode クラスのコンストラクタは引数なしで呼び出します。

```
PolygonMode ();
```

カリングの設定を行うには setCulling()関数を使用します。

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

引数 mode には CULL_BACK, CULL_FRONT, CULL_NONE の中から 1 つを選択します。一般的に言って物体が凸型で閉じている場合裏面を描画する必要はありません。CULL_BACK は背面をカリングし表面のみを描画します。CULL_FRONT は逆に表面をカリングして裏面のみを描画します。CULL_NONE はカリングを行わず両面を描画します。デフォルトは CULL_BACK です。

現在のカリングモードを取得するには getCulling()関数を使用します。

```
int getCulling () const;
```

戻り値はカリングモードのいずれかです。

ワインディングモードの変更するには setWinding()関数を使用します。

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

引数の mode には WINDING_CCW か WINDING_CW を指定します。ワインディングとはポリゴンの表面を定義の仕方です。面を構成する頂点を順に辿って時計回りだった時に”表”とするのが WINDIG_CW で、反時計回りの時に”表”とするのが WINDING_CCW です。デフォルトは WINDING_CCW です。すなわちトライアングルの 3 頂点がスクリーン空間で見て反時計まわりで指定された時が表と考えます。このポリゴンの表裏は頂点に設定された法線とは何も関係ありません。

現在のワインディングモードの取得には getWinding()関数を使用します。

```
int getWinding () const;
```

戻り値はワインディングモードの定数値のいずれかです。

ポリゴンのシェーディングモードの変更には setShading()関数を使います。

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

引数の mode には SHADE_FLAT か SHADE_SMOOTH を指定します。SHADE_FLAT はフラットシェーディングを行いトライアングル全体を 1 色で塗りつぶします。SHADE_SMOOTH はスムースシェーディング(グーローシェーディング)を行いトライアングルの 3 頂点のシェーディングの結果をなめらかに補間します。デフォルトは SHADE_SMOOTH です。

現在のシェーディングモードを取得するには getShading()関数を使用します。

```
int getShading () const;
```

戻り値はシェーディングモードの定数値です。

両面ライティングの設定を変更するには setTwoSidedLightingEnable()関数を使用します。通常裏面はカリングによって棄却されるのでライティングを計算する必要がありません。両面ライティングを有効にすると裏面でもライティング計算を行います。デフォルトは無効です。

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

引数 enable が true の時は両面ライティングを有効にします。false なら無効にします。デフォルトは false です。

現在の両面ライティングの設定を取得するには isTwoSidedLightingEnabled()関数を使用します。

```
bool isTwoSidedLightingEnabled () const;
```

戻り値は有効なら true、無効なら false です。

テクスチャーマッピングの透視変換補正の設定を変更するには setPerspectiveCorrectionEnable()関数を使用します。透視変換補正を無効にすると高速だが正しくない計算を行います。

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

引数 enable は true で透視変換補正を有効化します。false は無効化します。デフォルトは true です。この関数はヒントです。使用中の GPU はこれを無視する可能性があります。

実際に有効化・無効化されるかは使用中の GPU に依存します。

> 補足: テクスチャーの透視変換補正
>
> 現在販売されている GPU は全て透視変換補正を行い、そもそもこの補正を無効化することはできません。この関数は既に時代遅れです。ちなみに初代 PlayStation(R)の GPU は透視変換補正を行いませんでした。

ローカルカメラの設定を変更するには setLocalCameraLightingEnable()関数を使います。ローカルカメラとはライティング計算の時にカメラを実際のカメラ位置ではなく無限遠方に置くことを意味します。この時カメラ方向を向くビューベクトルが(0,0,1)に単純化され高速に計算できます。デフォルトでローカルカメラは有効です。

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

引数で true を指定するとローカルカメラライティングは有効になります。false で無効化されます。デフォルトは true です。この関数はヒントです。使用中の GPU はこれを無視する可能性があります。

現在のローカルカメラライティングの設定を取得するには isLocalCameraLightingEnabled()関数を使用します。

```
bool isLocalCameraLightingEnabled () const;
```

戻り値は有効の時 true で、無効の時 false です。

Appearance

Appearance クラスはその構成要素(コンポーネント)である Material, Texture2D, CompositingMode, PolygonMode, Fog を保持するコンテナクラスです。

Appearance クラスのコンストラクタは引数なしで呼び出します。インスタンス化直後は全てのコンポーネント

でデフォルトが適応されます。

```
Appearance ();
```

マテリアル要素を設定するには setMaterial()関数を使用します。

```
void setMaterial (Material* material);
```

引数 material には設定したい Material オブジェクトへのポインターを指定します。NULL を指定するとデフォルトのマテリアルが使われます。デフォルトは NULL です。

現在設定されている Material 要素を取得するには getMaterial()関数を使用します。

```
Material* getMaterial () const;
```

戻り値は現在の Material オブジェクトへのポインターです。Material が設定されていない場合は NULL が返ります。

テクスチャーを設定するには setTexture()関数を使用します。

```
void setTexture (int index, Texture2D* texture);
```

引数の index にはテクスチャーユニットの 0 から始まるインデックスを指定します。テクスチャーユニットは 4 つまで使用可能です。ただし使用中の GPU によっては 4 以下までしか対応しないことがあります。テクスチャーユニットは必ず 0 番から順に使用します。例えば 0,1 番目を使用せずに2,3 番目のテクスチャユニットを使用したときの動作は未定義です。引数の texture には使用する Texture2D オブジェクトへのポインターを指定します。NULL を指定するとそのテクスチャーユニットは使われません。デフォルトでは全てのテクスチャーは NULL に設定されています。

現在設定されているテクスチャーを取得するには getTexture()関数を使います。

```
Texture2D* getTexture (int index) const;
```

引数 index にはテクスチャーユニットの 0 から始まるインデックスを指定します。戻り値は使用中の Texture オブジェクトのポインターです。

PolygonMode 要素を設定するには setPolygonMode()関数を使用します。

```
void setPolygonMode (PolygonMode* polygon_mode);
```

引数 polygon_mode には設定したい PolygonMode オブジェクトへのポインターを渡します。NULL を設定するとデフォルトの設定が使用されます。デフォルトは NULL です。

現在設定されている PolygonMode オブジェクトを取得するには getPolygonMode()関数を使用します。

```
PolygonMode* getPolygonMode () const;
```

戻り値は現在の PolygonMode オブジェクトのポインターです。設定されていない場合は NULL が返ります。

CompositingMode 要素を設定するには setCompositingMode()関数を使用します。

```
void setCompositingMode (CompositingMode* compositing_mode);
```

引数 compositing_mode には CompositingMode オブジェクトへのポインターを渡します。NULL を渡すとデフォルトの設定が使われます。デフォルトは NULL です。

現在設定されている CompositingMode 要素を取得するには getCompositingMode()関数を使用します。

```
CompositingMode* getCompositingMode () const;
```

戻り値は CompositingMode オブジェクトのポインターです。設定されていない場合は NULL が返ります。

Fog を設定するには setFog()関数を使用します。

```
void setFog (Fog* fog);
```

引数には Fog オブジェクトへのポインターを渡します。NULL を渡すとデフォルトの設定が使われます。

現在設定されている Fog 要素を取得するには getFog()関数を使用します。

```
Fog* getFog () const;
```

戻り値は Fog オブジェクトのポインターです。設定されていない場合は NULL が返ります。

レイヤーは物体のレンダリングの順番をユーザーが明示的に制御するのに使用されます。レイヤーは[-63,63] (-63 と 63 を含む)の整数値で、番号の小さい方から順番にレンダリングされます。 つまり-63 のレイヤー番号を持つノードが一番始めに描画され、63 のノードが一番最後に描画されます。デフォルトのレイヤー番号は 0 番です。また同一番号レイヤーでは必ず不透明物体(CompositingMode が REPLACE なもの)が先に描画され、半透明物体(同 REPLACE 以外のもの)が後で描画されます。

レンダリンレイヤーの設定には setLayer()関数を使用します。

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

引数 layer にはレイヤー番号[-63,63]の整数値を指定します。デフォルトのレイヤー番号は 0 です。

現在のレイヤー番号を取得するには getLayer()関数を使用します。

```
int getLayer () const;
```

戻り値は設定されているレイヤー番号です。

第7章　アニメーション

この章では M3G API を使ったアニメーションの方法を解説します。アニメーションの最も基本となる単位はアニメーショントラックです。アニメーショントラックはデータの時間変化の記録(キーフレーム)と、そのデータの適応先(ターゲットプロパティ)の 2 つを保有します。オブジェクトにアニメーションを追加するとは、オブジェクトにアニメーショントラックを追加する事と等価です。アニメーショントラックの対象プロパティは後の animate() 関数の呼び出しによって時刻 t の時の値に変更されます。

通常アニメーションはシーンノード全体で共通で制御するので World クラスの animate()関数を呼び出します。例えば下記のコードはシーン全体を時刻 100 の値にアニメートします。

```
wld->animate (100);
```

ワールド時刻(world_time)はシーン全体の設定時刻です。単位は任意の整数値です。M3G では時刻 1 が現実世界の何秒、何分に相当するかは規定していません。ゲームでは 1 時刻を 1msec にする事が多いように思います。animate()関数はアニメーションが設定されていないオブジェクトに対しては何も影響を与えません。

アニメーションを付加する大まかな手順は下のとおりです。

1. AnimationController オブジェクトの作成
2. KeyframeSequence オブジェクトの作成
3. AnimationTrack オブジェクトの作成
4. AnimationTrack オブジェクトの追加

この章では KeyframeSequence, AnimationController, AnimationTrack の 3 クラスを解説します。

KeyframeSequence

KeyframeSequence クラスは複数のキーフレームデータを保有するとともに、キーフレーム間の補完方法、繰り返しモードを設定します。

KeyframeSequence のコンストラクタは以下の形式で呼び出します。

```
KeyframeSequence (int num_keyframes, int num_components, int interpolation);
```

引数の num_keyframes にはキーフレーム数を指定します。num_components には 1 キーフレームに含まれる値の要素数(コンポーネント数)を指定します。これはアニメーションターゲットによって異なります。例えば平行移動なら x,y,z の 3 要素、不透明度 α なら 1 要素です。Interpolation にはキーフレーム間の補完方法を LINEAR, SLERP, SPLINE, SQUAD, STEP の中から選択します。

Interpolation	解説
SLERP	球面線形補間(クォータニオン)
SPLINE	スプライン補間
SQUAD	スプライン補間(クォータニオン)
STEP	1 つ前のキーフレームをそのまま使用
LINEAR	線形補間

表 11:キーフレームの補間方式

この KefyrameSequence クラスに設定されている総キーフレーム数を取得するには getKeyframeCount()関数を使用します。

```
int getKeyframeCount () const;
```

戻り値は総キーフレーム数です。コンストラクタで指定した時のキーフレーム数が返ります。

キーフレームあたりのコンポーネント数を取得するには getComponentCount()関数を使用します。

```
int getComponentCount () const;
```

戻り値はコンストラクタで指定したコンポーネント数です。

キーフレームにデータを設定するには setKeyframe()関数を使用します。

```
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 getKeyframeCount()-1 まで有効です。キーフレームデータには時刻(time)と値(value)を設定します。time は 0 から始まる整数値です。時刻の単位は特に規定されていません。ゲームだと msec とすることが多いように感じます。value はコンストラクタで指定したコンポーネント数個の float の配列を指すポインターです。データは直ちに内部的に確保した領域にコピーされるので、この関数が返ったら即開放して構いません。キーフレームは値をセットしただけではまだ有効ではありません。このあと期間(duration)と有効範囲(valid range)を設定する必要があります。

一連のキーフレームシーケンスが表すアニメーションの長さを設定するには setDuration()関数を使用します。

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

引数の duration には長さを時刻で指定します。このアニメーションは[0, duration)の時刻で有効です。この区間以外の時刻に設定されたキーフレームは無視されます。デフォルトは dration=0 なのでユーザーは必ずこの関数を呼び出して有効期間を設定する必要があります。

現在設定されれている有効期間を取得するには getDuration()関数を使用します。

```
int getDuration () const;
```

戻り値は有効期間です。

有効範囲(valid range)を設定するには setValidRange()関数を使用します。セットしたキーフレームはこの関数を呼び出して有効にする必要があります。

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

引数の[first, last]のキーフレームが有効になります(first, last 共に有効範囲に含む)。first と last は 0 から getKeyframeCount()-1 までの整数値を指定します。この範囲に含まれないキーフレームは無視されます。

デフォルトは first=0, last=0 なのでキーフレームはすべて無効になっています。ユーザーはこの関数を呼んで有効範囲を設定する必要があります。この関数は例えば一連のキーフレームシーケンスをサブシーケンスに分けて使用するときに有用です。100 個のキーフレームを 20 個ずつ 5 個のアニメーションセットに分けて、状況に応じてアニメーションセットを切り替えたりする事ができます。

現在設定されている有効範囲を取得するには getValidRangeFirst()、getValidRangeLast()関数を使用します。

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

それぞれ有効範囲の[first, last]が返ります。

リピートモードを変更するには setRepeatMode()関数を使用します。

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

引数の mode は CONSTANT または LOOP を選択します。この関数は有効期間(duration)の範囲外の時の時刻のアニメーションを定義します。CONSTANT は最後のキーフレームを再生します。つまりキャラクターは最後のキーフレームの値のまま停止します。LOOP は 0 に折り返します(時刻が負の値の時は逆方向に)。つまり有効期間のアニメーションを繰り返し再生します。デフォルトは CONSTANT です。

現在設定されているリピートモードを取得するには getRepeatMode()関数を使用します。

```
int getRepeatMode () const;
```

戻り値はリピートモードで CONSTANT か LOOP のどちらかです。

AnimationTrack

AnimationTrack は KeyframeSequence オブジェクトを特定のターゲットプロパティに結びつけます。アニメーショントラックはアニメーションの最も基本となる単位で、オブジェクトにアニメーションを追加するとは、addAnimationTrack()関数で AnimationTrack オブジェクトを追加することを意味します。アニメーションの対象となる属性値をターゲットプロパティと呼びます。例えば TLANSLATION は 3D 空間内の平行移動属性を指します。

```
node->addAnimationTrack (anim_track);
```

AnimationTrack クラスのコンストラクタは以下の形式で呼び出します。

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

引数の sequence にはアニメーションのデータセットである KeyframeSequence オブジェクトを指定します。property にはその適応先を表 11 の中から指定します。sequence には propety に適したコンポーネント数のデータを保持している必要があります。またインスタンス化した直後のアニメーショントラックはアニメーションコントローラーがセットされておらずまだ無効です。必ず後述の setController()関数で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。

プロパティ	コンポーネント数	設定できるクラス	解説
ALPHA	1	Node, Background, Material	ノードの不透明度または色の A 成分。補間された値は[0,1]にクランプされます。
AMBIENT_COLOR	3	Material	マテリアルの環境係数。補間された値は[0,1]にクランプされます。
COLOR	3	Light, Background, Fog, Texture2D, VertexBuffer	色。α チャネルは含めません。補間された値は[0,1]にクランプされます。α チャネルは別途 ALPHA プロパティで指定します。
CROP	2, 4	Background, Sprite3D	クロップ領域。コンポーネント数は 2(x,y)もしくは 4(x,y,width,height)
DENSITY	1	Fog	指数フォグの時のフォグ密度
DIFFUSE_COLOR	3	Material	マテリアルの拡散係数。補間された値は[0,1]にクランプされます。A 成分は別途 ALPHA プロパティで指定します。
EMISSIVE_COLOR	3	Material	マテリアルの発光係数。補間された値は[0,1]にクランプされます。
FAR_DISTANCE	1	Camera, Fog	カメラの far 係数もしくは線形フォグの far 係数。
FIELD_OF_VIEW	1	Camera	カメラの fov 係数。透視投影の時は[0,180]にクランプされます。平行投影の時は 0 より大きな最小の数にクランプされます。
INTENSITY	1	Light	ライトの強度。
MORPH_WEIGHTS	V	MorphingMesh	モーフィングメッシュのウェイト値。モーフターゲットの数を N とすれば if V < N v[i] I < V 0 V < I < N if v > N v[i] = N[i] I < N 残りは無視
NEAR_DISTANCE	1	Camera, Fog	カメラの near 係数もしくは線形フォグの near 係数。
ORIENTATION	4	Transformable	回転要素 R。クオータニオン。 V[0] = ベクトル成分の i 要素

			V[1] = ベクトル成分の j 要素 V[2] = ベクトル成分の k 要素 V[3] = スカラー成分の w 要素
PICKABILITY	1	Node	ピック可能フラグ。
SCALE	1,3	Transformable	スケール要素 S。1 要素だけ指定された場合は同じ値を 3 軸に適応する。
SHININESS	1	Material	鏡面項の指数係数。[0,128]にクランプされる。
SPECULAR_COLOR	3	Material	鏡面係数。[0,1]にクランプされる
SPOT_ANGLE	1	Light	スポットライトの角度係数。[0,90]にクランプされる。
SPOT_EXPONENT	1	Light	スポットライトの指数係数。[0,128]にクランプされる。
TRANSLATION	3	Tarnsformable	平行移動要素 T。
VISIBILITY	1	Node	レンダリング可能フラグ。

表 12:ターゲットプロパティー覧

> 補足: クォータニオンはどうやって計算すればいいのか
>
> M3G の仕様では回転アニメーションを付加するのにクォータニオンで指定することになっていますが、クォータニオンは M3G の規格には含まれていません。Desktop-M3G では内部で使用している Quaternion クラスを使用できますが推奨はしません。外部の数学ライブラリを使用してください。

この AnimationTrack オブジェクトのキーフレームシーケンスを取得するには getKeyframeSequence() 関数を使用します。

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

戻り値は設定されている KeyframeSequence オブジェクトです。

この AnimationTrack オブジェクトのターゲットプロパティを取得するには getTargetProperty()関数を使用します。

```
int getTargetProperty () const;
```

戻り値は表 11 のプロパティのいずれかです。

AnimationTrack オブジェクトには再生速度などをコントロールする AnimationController オブジェクトを関連付ける必要があります。コントローラーが関連付けられていないアニメーショントラックは無視されます。

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

引数 controller は AnimationController オブジェクトのポインターです。NULL の場合は以前の値がクリアされ、このアニメーショントラックは無視されます。

現在設定されているコントローラーを取得するには getController()関数を使用します。

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

戻り値はコントローラーです。設定されていない場合は NULL が返ります。

AnimationController

AnimationController クラスは主にアニメーションの再生速度を制御します。有効なアニメーショントラックは必ず AnimationController オブジェクトに関連づけられている必要があります。

シーケンス時間とワールド時間

アニメーションに重要な概念としてシーケンス時間とワールド時間があります。シーケンス時間はそのアニメーションが持つローカルな時間軸です。キーフレームシーケンスの時刻 t はシーケンス時間です。ワールド時間はシーン全体で使用する時刻です。

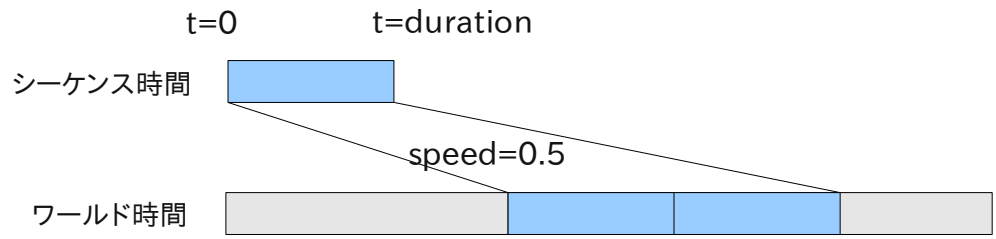


図 12:シーケンス時間とワールド時間

図 12 はシーケンス時間とワールド時間の対応を示した図です。アニメーションはシーケンス時間で t=[0,duration]の期間で定義されています。ライブラリはあるワールド時刻 t'に対応するシーケンス時間 t を計算し、キーフレームから値を補間しプロパティの値を変更します。図中では speed=0.5 なのでアニメーションはワールド時間で見ると 2 倍に引き伸ばされています(スロー再生)。ワールド時間の特定の時刻 t'をシーケンス時間の特定の時刻 t に合わせることが出来ます。この時の時間合わせに使用した時刻をワールドリファレンス時間、シーケンスリファレンス時間と呼びます。

AnimationController

AnimationController クラスのコンストラクタは引数なしで呼び出します。

```
AnimationController ();
```

このコントローラーが有効な期間をアクティブ区間(Active Interval)と呼びます。コントローラーに関連付けられているアニメーショントラックはアクティブ区間の間でのみ効果を発します。

アクティブ区間を設定するには setActiveInterval()関数を使用します。

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

引数の[start, end)の期間がアクティブになります(start を含み、end を含まない)。デフォルトは start=0, end=0 です。start と end を同じ値にすると特殊な設定になり全期間で有効となります。

現在設定されているアクティブ区間を取得するには getActiveIntarvalStart()、getActiveIntervalEnd()関数を使用します。

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

戻り値はそれぞれアクティブ区間の[start, end)です。

アニメーションの再生スピードを変更するには setSpeed()関数を使用します。

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

引数 speed には再生速度を指定します。標準は 1.0 です。0 はアニメーションを停止します。負の値を指定すると逆方向に再生します。引数の world_time には現在のワールド時間を指定します。これはアニメーションが連続して変化するように操作するのにライブラリ内部で必要になるパラメーターでユーザーが直接気にするものではありません。現在のワールド時間を指定して下さい。デフォルトの再生速度は speed=1.0 です。

現在設定されている再生速度を取得するには getSpeed()関数を使用します。

```
float getSpeed () const;
```

戻り値は再生速度です。

アニメーションウェイトの変更には setWeight()関数を使用します。

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

引数の weight に設定したいウェイト値を 0 以上で指定します。weight=0 はこのコントローラーを無効化します。デフォルトは weight=1.0 です。ウェイト値はアニメーションのブレンドに使用されます。

現在設定されているウェイト値を取得するには getWegith()関数を使用します。

```
float getWeight () const;
```

戻り値はウェイトです。

シーケンス時間とワールド時間の同期をとるには setPosition()関数を使用します。この時のワールド時間をリファレンスワールド時間と呼びます。これはアニメーションシーケンスを任意の位置にジャンプするのに使用します。

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

この関数はアニメーションの sequence_time を world_time に一致させます。例えば world_time に現在時刻を指定し sequence_time に任意の時間を指定すると、再生中のアニメーションを任意の位置にジャンプすることができます。デフォルトは sequence_time=0, world_time=0 です。つまりアニメーションシーケンスの先頭位置がワールド時間の 0 に一致しています。

現在設定されているリファレンスのワールド時間を取得するには getRefWorldTime()関数を使用します。

```
int   getRefWorldTime () const;
```

戻り値はワールドリファレンス時間です。

任意のワールド時間に対応するシーケンス時間を取得するには getPosition()関数を使用します。

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

引数の world_time にはワールド時間を指定します。戻り値はこの時刻に対するシーケンス時間です。

第8章　モデルの作成とロード

M3G では専用のファイルフォーマットを規定しています。通常 M3G 形式のファイルの拡張子は.m3g が使われます。ファイルにはモデルデータ、モーションデータ、シーングラフ等をすべて含みます。ユーザーは Loader クラスを使ってこの M3G 形式のファイルをロードし利用することができます。

M3G 形式のモデルの作成

主要な DCC ツールでは標準もしくは別途エクスポーターを購入する事で M3G 形式でファイルを出力することができます。商用ソフトで現在対応していると思われるのは以下の 4 つです。

- MAYA(R)
- MAX(R)
- LightWave(R)
- MilkShape3D(R)

フリーソフトでは Blender が対応しています。

- Blender(R)

Blender を使えばフリーで試すことが出来ますが筆者が試したところ安定してなかったので、M3G の開発では約 3000 円で買える MilkShape3D を利用しています。MilkShape3D 用の M3G エクスポーターは Sony Ericsson 社がフリーで公開しているものをダウンロードして使用しています。Desktop-M3G で動作を確認しているのは MilShape3D で出力した M3G ファイルのみです。

M3G Exporter for Milkshape3D

MilkShape3D 用の M3G Exporter は Sony Ericsson 社によって提供されています。ダウンロードは下記の URL から行ってください。

- http://developer.sonyericsson.com/wportal/devworld/downloads/download/dw-91536-semcm3gexporter10milkshape3d

ダウンロードしたファイルを解凍するとインストーラーが出現します。実行して後は手順に従ってください。

図 13:M3G Exporter for MilkShape3D のインストール画面

MilkShape3D

MilkShape3D は chumbalum soft 社によって開発・販売されているモデリングソフトです。低価格でありながらゲーム開発で使うのに十分なモデリング機能を有しています。特に対応しているエクスポーターの種類の多さには定評があります。MilkShape3D は下記の URL からオンラインで購入できます。詳しい購入方法はサイトをご覧ください。

- http://chumbalum.swissquake.ch/

M3G 形式で出力するには「File」ー>「Export」ー>「Mobile 3D Graphics」を選択しエクスポートします。

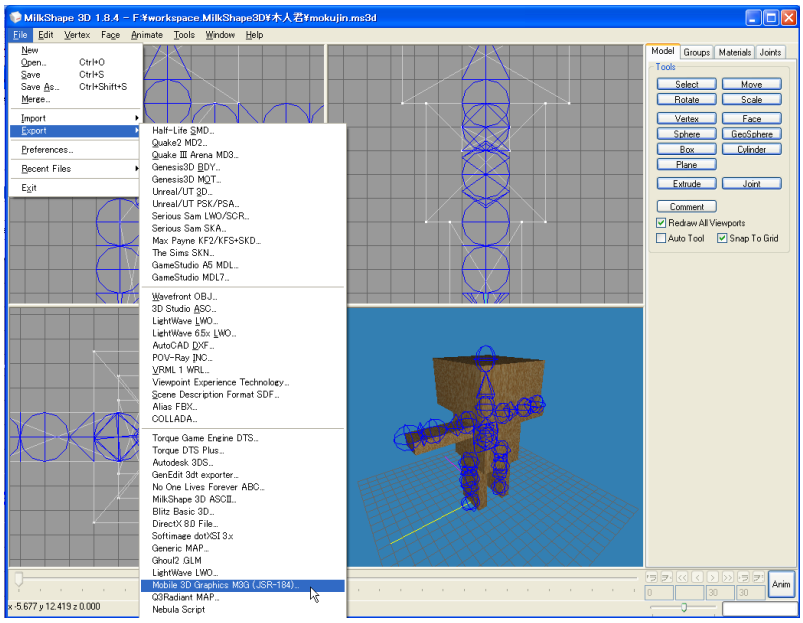


図 14:MilkShape3D の M3G エクスポーター

図 14 はテスト用データの「木人君 1 号」をエクスポートしている所です。

Loader

作成した M3G ファイルは Loader クラスの load 関数を使ってロードします。この関数はスタティックです。

```
static std::vector<Object3D*> load (const char* file_name);
```

引数の file_name にはファイル名を指定します。Loader クラス M3G 形式のファイルか、PNG 画像もしくは JPEG 画像をロード可能です。画像ファイルはサイズが 2 の階乗である必要があります。戻り値は 1 次元の Object3D のポインターの配列です。Object3D はほぼすべてのクラスの基底クラスで、雑際にはこのポインターは Texture2D オブジェクトや World オブジェクトを指しています。シーングラフ全体をロードしたい場合はこの配列の中から欲しい World クラスを探します。

```
objs = Loader::load ("simple.m3g");
for (int i = 0; i < (int)objs.size(); i++) {
    wld = dynamic_cast<World*>(objs[i]);
```

```
        if (wld)
            break;
    }
```

また上記の方法とは別に(DCC ツールが対応していれば)M3G オブジェクトに一意な ID を付加しておいて、ID で識別する方法もあります。ただし MilkShape3D は ID を付加しないのでこの方法は使えません。

```
objs = Loader::load ("simple.m3g");
for (int i = 0; i < (int)objs.size(); i++) {
    if (objs[i]->getUserID() == WORLD_ID) {
        wld = dynamic_cast<World*>(objs[i]);
        break;
}
```

> 補足: Object3D 配列の順番
>
> M3G では通常シーングラフの一番下(枝)のノードから上に向かって格納されます。つまりルートノード(通常 World)が一番最後に格納されます。従って通常は配列の一番後ろのオブジェクトが World クラスです。for ループは 0 から+1 方向に回すよりも size()-1 から-1 方向に回した方が速く見つかります。ただし最後が必ず World クラスであると仮定してはいけません。

また load 関数は PNG や JPEG などの画像ファイルを直接ロードし、Image2D オブジェクトを返す事ができます。

```
    img = dynamic_cast<Image2D*>(Loader::load ("test.png")[0]);
```

引数で指定する画像は PNG フォーマットもしくは JPEG フォーマットです。JPEG 画像のロードはオプションです。実装によっては対応していません。戻り値の配列のサイズは 1 でインデックスの 0 番が Image2D オブジェクトへのポインターです。

また load 関数はメモリ上に展開したデータからのロードにも対応しています。これは暗号化した状態でファイルに保存し、メモリ上に M3G ファイルを復号化してからロードする使い方を想定しています。

```
static std::vector<Object3D*> load (int length, const char* p, int offset);
```

引数の length にはメモリ上の長さをバイト数で指定します。p がメモリへのポインター。offset が p から有効な M3G ファイルが始まるオフセットです。戻り値は Object3D のポインターの配列で 1 番目の形式の load 関数と同じです。

Loader クラスには load()関数以外のメンバー関数はありません。またコンストラクタはプライベート宣言されているのでインスタンス化はできません。

第9章　エラーハンドリング

例外

M3G ではエラー処理として C++の例外機構を使用しています。ライブラリ内部で何らかのエラーを検出した場合、即座に例外を発生して処理を戻します。ユーザーは適切に try～catch を設定しエラーを処理して下さい。投げられる例外は M3G の仕様書で定義された 7 種類と Desktop-M3G で独自に定義した 3 種類があります。それぞれの例外の発生条件は全て M3G リファレンスにてドキュメント化されているのでそちらを参照してください。例外はすべて std::exception を継承し、ファイル名と関数名と発生理由を含むメッセージを持ちます。

- ArithmeticException
  - 逆行列の計算など主に数学的な計算処理中のエラーで発生します。
- IllegalArgumentException
  - 引数が不正な時に発生します。
- IllegalStateException
  - 処理を行うための状態が不正な時に発生します。
- IndexOutOfBoundsException
  - 配列の添字が範囲外の時に発生します。
- IOException
  - ファイルの読み込み時のエラーで発生します。
- NullPointerException
  - ポインターが NULL の時に発生します。
- SecurityException
  - アクセス権限が無いときに発生します。Desktop-M3G では発生しません。
- NotImplementedException
  - M3G 非標準。仕様上存在するが実装されていない機能を呼び出したときに発生します。イミーディエイトモード関係の呼び出しで発生します。実装の予定はありません。
- OpenGLException
  - M3G 非標準。OpenGL まわりのエラーで発生します。典型的には OpenGL の呼び出し準備が整っていないか、ハードウェアが必要な機能を実装していない時に発生します。Desktop-M3G の必要要件を確認して下さい。
- InternalException
  - M3G 非標準。このエラーは通常では絶対に発生しません。もしこの例外を補足したらお手数ですが開発元までお知らせ下さい。

下記は M3G ファイルをロードしようとしてファイルが見つからず IOException が発生している例です。

```
$ a.out
terminate called after throwing an instance of 'm3g::IOException'
what():  Loader.cpp:load Can't open the file, name=test.m3g.
```

例えばこのように try～catch で例外を補足し、適切に対処してください。

```
vector<Object3D*> objs;
try {
    objs = Loader:: load ("test.m3g");
} catch (const m3g::IOException& e) {
    cout << "File not found!! ¥n";
    cout << e.msg << "¥n";
    return 0;
}
```

デバッグ表示

全てのクラスはデバッグ用にオブジェクトの情報を表示する M3G 非標準のメンバー関数 print()を持っています。この関数の名前は言語によって異なります。

C++	std::ostream& print (std::ostream& out) const; std::ostream& operator<< (const T& t);
Java	String toString ()
Ruby	inspect

これらの関数はオブジェクトの情報をコンソールに表示(C++)するか、その文字列を返し(Java,Ruby)ます。これらの文字列をデバッグ用途以外に使用してはいけません。また文字列をパースしてそれに依存した処理を行ってはいけません。返される文字列は予告無く変更される可能性があります。

以下の表示例はあるシーンの World オブジェクトの出力結果です。これを見ると直接の子ノードが 3 つ

(Group,Camera,Light)、アクティブカメラが[1]のカメラ、バックグランドオブジェクトはなし、などがわかります。

```
World:
  active camera = [1]
  background    = NOT FOUND
  [0] : Group: 1 nodes  [0] : SkinnedMesh: 1657 vertices, 37 bones
  [1] : Camera:   type="PERSPECTIVE", fovy=37.8493, aspect_ratio=1.5, near=0.1,
  far=1000
  [2] : Light: mode=DIRECTIONAL, color=0xffffff, intensity=1, attenuation=1,0,0,
  spot=45,0
```

表示される情報はそのクラスが直接保有するデータだけです。基底クラスのデータは含みません。全てのデータを表示するには下記のように基底クラスも含めて全ての print()関数を明示的に呼ぶ必要があります。

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

第10章　　付録

付録 A：オリジナルの M3G API と Desktop-M3G の違い

この章ではオリジナルの M3G API と Desktop-M3G の実装で異なる点を列挙します。const など Java 言語と C++言語の違いから来る些細な違いについては省略しています。

配列のサイズ指定

java の配列はそれ自身からサイズを取得できますが、C++のポインターからはサイズが取得できないので別途サイズも渡しています。

```
Java:    void setWeights (float[] weights)
                              ↓
C++:     void setWeights (int num_weights, float* weights);
```

VertexArray の float 型

オリジナルの M3G 1.1 では VertexArray のデータ型は char 型と short 型のみで float 型がありません。メモリ使用量を気にする必要がないデスクトップ用に使いやすさを重視して float 型を拡張しています。なお次の M3G 2.0 で仕様に float 型が入る予定です。

```
void set (int first_vertex, int num_vertices, const float* values);
```

VertexBuffer の scale, bias 値

オリジナルの M3G では VertexBuffer クラスで自動設定される scale, bais 値を取得する方法がありません。自明なので必要ないと言えば必要ないのですが API の対称性を重視し取得出来るように拡張しています。

```
VertexArray* getColors (float* scale_bias=0) const;
```

VertexBuffer のカラーの指定

オリジナルの M3G ではカラーを Java のバイト型(-128, 127)で指定します。Java の 1 バイト型には unsigned 型がなく、これはやむを得ないのですが極めて直感に反します。C++では unsinged char 型(0, 255)で指定するように変更しました。

Graphics3D のイミィーディエイトモード

オリジナルの M3G にはシーン全体を一括して描画する”リテインドモード”と、1 つずつノードを指定して描画する”イミーディエイトモード”の 2 つがあります。イミーディエイトモードはメモリの節約になりますが、コードが複雑になり M3G の利点を損ないます。イミーディエイトモードは時代遅れです。あえて使用する理由はありません。Desktop-M3G ではリテインドモードのみ対応しています。

Graphics3D の bindTarget()関数/releaseTarget()関数

オリジナルの M3G 規格では OpenGL コンテキストの作成・削除も仕様に含まれています。bindTarget()と releaseTarget()関数がこれに相当します。デスクトップ用ではそれらの統一された方法が存在しないため、Desktop-M3G に両関数はありません。レンダリングターゲットはフレームバッファー固定です。OpenGL コンテキストの作成とウィンドウの作成は glut などの外部ライブラリが行います。

VertexBuffer の頂点座標とテクスチャー座標で 1 バイト型

VertexBuffer の頂点座標(positions)とテクスチャー座標(tex_coords)で、オリジナルの M3G では許される 1 バイト型のデータタイプが Desktop-M3G では例外を発生します。これは OpenGL ES ではない OpenGL が 1 バイト型を許容しないからです。なおロード関数内でファイルフォーマット中に 1 バイト型のデータを見つけた場合 2 バイト型に自動的に変換します。

Camera の lookAt()関数

Camera の位置制御を楽に行うために glut の glutLookAt に相当する関数を独自拡張しています。オリジナルの M3G にこの関数はありません。移植性を重視する場合は使用してはいけません。なおここで設定したパ

ラメーターは TRS 要素ではなく M 要素にセットされます。

```
void lookAt (float from_x, float from_y, float from_z,
                           float to_x, float to_y, float to_z,
                           float up_x, float up_y, float up_z);
```

print()関数

すべてのクラスはメンバー関数の print()を持ち、オブジェクト内部データを表示できます。これはデバッグ用途に拡張した関数で M3G 非標準です。表示されるデータは予告なく変更される可能性があります。なお cout にリダイレクトしても表示可能です。

```
virtual std::ostream& print (std::ostream& out) const;
std::ostream& operator<< (std::ostream& out, const m3g::Camera& c);
```