MathEngine Karma™ Viewer

Developer Guide

MathEngine Karma Viewer. Developer Guide.

Contents

Chapter 1 • What Can the Karma Viewer Do?

Overview

The MathEngine Karma Viewer, or MeViewer, is a basic 3D rendering and interactive viewing tool that is included as part of the integrated Karma software package.

> **NOTE:** Karma does not require that this renderer be used. Any third party renderer can be used with dynamics and/or collision. The viewer is a visualisation tool that can be used by developers to display the virtual world they are building. It is certainly not designed to be used as a high quality renderer in a finished product.

Karma includes sample programs that demonstrate how to use the software. The purpose of MeViewer is to allow the sample programs - and any prototype game or test code that you write - to display a 3D scene and to allow interaction with the scene.

A straightforward extensible user interface is provided that includes camera operations such as pan, rotate, zoom, change the lighting and shading, stop and restart the animation.

While this guide provides a comprehensive description of MeViewer, it is recommended that you use it in conjunction with the supplied source code.

MeViewer has undergone substantial changes, that are reflected in the changed API, since the previous version. Please refer to *Chapter 2 • Programming the Viewer* for details about this.

Architecture

In order that applications using MeViewer look the same on all platforms, and to aid the rapid development of support for new platforms, MeViewer has been split into two layers. The front end, that provides the API, is platform independent. The only API call that executes platform specific code is `RRun`. This front end creates object geometries in data structures detailed in *Geometry Data Format* on page 30 that can be drawn by any platform supporting 3D. The back end is platform specific, and reads the data structures created by the front end, drawing them on screen in a well defined way (see *Back-End Interface* on page 34). User input is handled by the back end.

In the rest of this document, MeViewer will generally refer to the front end, whereas the term *renderer* will normally mean the back end.

Coordinate System

Like most graphics applications, MeViewer uses a *Left Handed* cartesian coordinate system, with x increasing from left to right as you view the screen, y from bottom to top, and z into the screen. Please note that this is purely for display purposes: Karma is independent of choice of handedness, and if you work consistently with a right-handed coordinate system, the transformation matrices generated will be suitable for display with a renderer which uses that system. Positions are specified by floating point values of type defined by `AcmeReal`. The exception to this is the 2D layer which uses coordinates from (0,0) in the top left corner of the screen to (640,448) in the bottom right.

Colors

All colors are specified as RGBA, except some used by lights, that are RGB. The RGBA model uses relative intensities varying between 0 and 1 for the primary colors red, green, and blue, and a transparency value alpha. An alpha value of 0 represents complete transparency regardless of the color values, and an alpha value of 1 complete opacity.

File Loading

MeViewer generates primitive geometries (such as spheres and boxes) procedurally, but user geometries (see *Object Geometry Files* on page 32) and textures (see *Textures* on page 24) are loaded from files. The locations in which MeViewer searches for files is specified in the `R_DefaultFileLocations` array defined in `MeViewer.c`.

In order to accelerate geometry loading, MeViewer caches the last object file that it loaded. If loading time is significant, objects using the same geometry file should be created together to minimize file reads.

Platforms Supported

The Viewer runs on the Sony PlayStation2, Windows 98, Windows NT, Windows 2000, Linux and IRIX. OpenGL is supported on all platforms except PlayStation2, and Direct3D on Windows 95/98 and Windows 2000. Note that version 3.2 (or above) of Mesa is required for Linux platforms and DirectX 7.0 (or above) is required for Direct3D on Windows platforms.

Building the Viewer into Applications

The Viewer is included as part of the compressed Karma archive, and its header files and libraries can be found in the `src/components/MeViewer2` subdirectory of the directory tree when you have uncompressed the archive that you have downloaded.

The source code of the example programs provided with Karma demonstrates how to use the Viewer.

Chapter 2 • Programming the Viewer

Overview

The MeViewer API (Application Programming Interface), is a set of functions designed to provide to Karma developers with the basic rendering functionality required to demonstrate their simulation code. It is not intended as a replacement for commercial 3D graphics libraries such as OpenGL or Direct3D.

Two libraries (`MeProfile` and `MeCommandLine`) will be used together with MeViewer but will not be covered in this manual. The source files and header files for these libraries can be found in `...\metoolkit\src\components\MeGlobals\src` and `...\metoolkit\include` respectively.

Using the Viewer in an Application

Let's look at the `RainbowChain.c` sample program from Karma. It provides a good example of how you can use the Viewer in your own programs. All of the following calls are described in more detail in the remainder of this chapter.

First Steps

You need first to include `MeViewer.h` and to declare a rendering context of type `RRender`, which is traditionally named `rc`. Then the application declares pointers to the two graphics structures of type `RGraphic` that will appear in the RainbowChain tutorial: a ball and a plane.

```
#include "MeViewer.h"

/* Render context */
RRender *rc;

/* graphics */
RGraphic *sphereG[NBALLS];
RGraphic *planeG;
```

Initializing the Renderer

In the main routine of RainbowChain, we pick up the renderer type (`-gl`, `-d3d`, `-null`, `-bench`, `-profile` all explained elsewhere in this chapter) from the command line:

```
/* Initialise rendering attributes. */

options = MeCommandLineOptionsCreate(argc, argv);
rc = RRenderContextCreate(options, 0, !MEFALSE);
MeCommandLineOptionsDestroy(options);
if (!rc)
   return 1;
```

Then we initialize the camera:

```
RCameraRotateElevation(rc, (MeReal)1.1);
RCameraRotateAngle(rc, (MeReal)0.2);
RCameraZoom(rc, 10);
```

Add a a visual performance bar:

```
RPerformanceBarCreate(rc);
```

Create the help system:

```
RRenderCreateUserHelp(rc, help, 1);
RRenderToggleUserHelp(rc);
```

And assign the keyboard call-back function:

```
RRenderSetActionNCallBack(rc, 2, Reset, 0);
```

Creating the Graphics

IFinally, we create the graphical objects, spheres for the chain and a plane for the floor.

```
for(i=0; i<NBALLS; i++)
{
    HSVtoRGB(((MeReal)i/(MeReal)NBALLS)*360, 1, 1, color);
    sphereG[i] = RGraphicSphereCreate(rc, ballRadius, color,
                                          McdModelGetTransformPtr(ball[i]));
}

color[0] = color[1] = color[2] = color[3] = 1;
planeG = RGraphicGroundPlaneCreate(rc, 24, 2, color, 0);
RGraphicSetTexture(rc, planeG, "checkerboard");
```

Running the Simulation

Then we call `RRun()` to run the simulation and render the graphics.

```
RRun(rc, tick, 0);
```

`RRun()` controls the event loop (the main loop) for the simulation. `tick` is a function which is called before rendering each frame, and in this application contains the code which updates the positions of each ball in the chain using Karma dynamics.

Terminating the Program

Calling `RRun()` sends the viewer into a loop, which—depending on your platform and your underlying graphics package—may not return. For safety, assume that your program will not return from `RRun()`.

To stop the program, the user presses <Esc> or clicks the close button, which results in a call to `exit()`, terminating your program. The effect is that no code placed after `RRun()` can execute.

So if you have any cleanup to do before your program terminates, put it in a function and register that function with `atexit()`. This way the Viewer will call that function before terminating the program. For example, here is a typical `cleanup()` function used in Karma:

```
RRenderContextDestroy(rc);
```

Register it with `atexit()` before calling `RRun()`:

```
atexit(cleanup);
  RRun(rc, tick);
/* No code placed after here will ever execute */
```

Render Contexts

A *Render Context* is a `RRender` structure (see the `MeViewerTypes.h` header file), that holds the state of a the viewer. Because there is no global render context, nearly all API calls will require a valid `RRender*` as their first parameter . Therefore the first task of an application using MeViewer is to create a render context

```
RRender * MEAPI RRenderContextCreate( MeCommandLineOptions* options,
                                      MeCommandLineOptions* overrideoptions,
                                      MeBool eat);
```

Creates a new render context. Fills the `RRender` with default values. Calls `RInit` in platform-dependent code. Return value is zero if creation or `RInit` fail. The `options` argument is an input from the command line, and options specified there are overridden by those specified in `overrideoptions`.

Here is an example of how to create a render context:

```
MeCommandLineOptions *options;
options = MeCommandLineOptionsCreate(argc, argv);
rc = RRenderContextCreate(options, 0, !MEFALSE);
MeCommandLineOptionsDestroy(options);
if (!rc) return 1;
```

```
int MEAPI RRenderContextDestroy( RRender *rc );
```

Cleans up and frees a Render Context

```
void MEAPI RRenderQuit( RRender *rc );
```

Tells the back-end to quit next frame. The argument `rc` is the render context to quit.

```
void MEAPI RRenderUpdateGraphicMatrices( RRender *rc );
```

Updates `RObjectHeader` matrices from `RGraphic` matrix pointers. Also used for timeout and pause graphic. Called each frame by the back-end. The argument `rc` is the render context all of whose `RGraphic`'s matrices are to be updated.

```
void MEAPI RRenderUpdateProjectionMatrices( RRender *rc );
```

Update the projection matrix.

Creating Objects

The platform-specific section of MeViewer only displays data in the format discussed in *Geometry Data Format* on page 30, which holds lists of triangles, grouped into objects each with associated color and texture properties. The format is based around the `RGraphic` structure. MeViewer provides functions to create `RGraphic` structures representing a number of primitive objects, and to create empty structures for application defined objects.

To create an `RGraphic` object from an object file:

```
RGraphic *MEAPI RGraphicCreate( RRender *rc, char *filename,
                               AcmeReal xScale, AcmeReal yScale,
                               AcmeReal zScale, const AcmeReal color[4],
                               MeMatrix4Ptr matrix, int is2D,
                               int bKeepAspectRatio );
```

Creates a `RGraphic` from object file. A new `RGraphic` is created using the parameters passed. The value `filename` specifies the object geometry file.The values `xScale`, `yScale` and `zScale` specifies the `x`, `y` and `z` scaling factor respectively. The value `color` specifies the object RGBA color. The value `matrix` is the pointer to the associated transformation matrix. The value `is2D` indicates if object is to be put in 2D render-list And finally, the value `bKeepAspectRatio` indicates if the aspect ratio is to be preserved when object is normalized

Or you may want to create an empty one:

```
RGraphic *MEAPI RGraphicCreateEmpty( int numVertices );
```

Allocates memory for `RGraphic`. The `numVertices` argument specifies how many vertices in `RGraphic` and should be multiple of 3. Fills in `numVertices` and pointers in `RGraphic.` This function returns a pointer to the resulting `RGraphic`, or `0` if it fails to do so.

Once you created a `RGraphic` object, you may delete or destroy it using:

```
void MEAPI RGraphicDestroy( RGraphic *rg );
```

Frees memory allocated for `RGraphic`. The argument `rg` is the `RGraphic` to destroy. This function also frees memory allocated for the associated `RObjectHeader` and vertex list.

```
void MEAPI RGraphicDelete( RRender *rc, RGraphic *rg, int is2D );
```

Removes `RGraphic` from list and then frees up memory. Hence, there is so no need to call `RGraphicDestroy` afterwards. The arguments `rc` and `rg` are the render context that the graphic is associated with, and the graphic to delete respectively. The `is2D` argument queries whether the graphic is in the 2D list.

Creating Primitives

The functions to create primitives are:

```
RGraphic * MEAPI RGraphic2DRectangleCreate( RRender *rc,
                                                AcmeReal orig_x,
                                            AcmeReal orig_y, AcmeReal width,
                                            AcmeReal height,
                                            const float color[4]);
```

Creates a `RGraphic` rectangle with the specified RGBA color. The argument `rc` is the render context into whose 2D list the resulting `RGraphic` is put. The arguments `orig_x` and `orig_y` are the x co-ordinate of the left edge of the rectangle and the y co-ordinate of the top edge of the rectangle respectively. The arguments `width` and `height` represent the x and y dimensions of the rectangle.This function returns a pointer to the resulting `RGraphic`, or `0` for failure. The associated `RGraphic` is added to the 2D render-list (see *Render Lists* on page 16).

```
RGraphic * MEAPI RGraphicGroundPlaneCreate ( RRender *rc,
                                             AcmeReal side_length,
                                             int triangles_per_side,
                                             const float color[4],
                                             AcmeReal y_position );
```

Creates a `RGraphic` representing a ground-plane. The plane is a square in x, z. The value `side_length` is the length of the side of the square and `triangles_per_side` specifies the number of triangles per side of the square. The value `y_position` sets the y position of the square.

```
RGraphic * MEAPI RGraphicBoxCreate( RRender *rc, AcmeReal width,
                                    AcmeReal height, AcmeReal depth,
                                    const float color[4],
                                    MeMatrix4Ptr matrix);
```

Creates a Box of width (x) `width`, height (y) `height` and depth (z) `depth`. The argument `color` is the RGBA color of the object and `matrix` is a pointer to the object's transformation matrix.

```
RGraphic * MEAPI RGraphicConeCreate( RRender *rc, AcmeReal radius,
                                     AcmeReal upper_height,
                                     AcmeReal lower_height,
                                     const float color[4],
                                     MeMatrix4Ptr matrix );
```

Creates a Cone. The argument `radius` is the radius of the base of the cone, `upper_height` is the length (z) from the origin to the apex and `lower_height` is the length (z) from the origin to the base. The variable `color` is the RGBA color of the object and `matrix` is a pointer to the object's transformation matrix.

```
RGraphic * MEAPI RGraphicCylinderCreate( RRender *rc,
                                          AcmeReal radius,
                                         AcmeReal height,
                                         const float color[4],
                                         MeMatrix4Ptr matrix);
```

Creates a Cylinder. The argument `radius` is the radius of the cylinder, `height` is length (z) of the cylinder, `color` is the RGBA color of the object, `matrix` is a pointer to the object's transformation matrix. The origin is located at the middle of the height (z).

```
RGraphic * MEAPI RGraphicLineCreate( RRender *rc,
                                    AcmeReal *origin, AcmeReal *end,
                                    const float color[4],
                                    MeMatrix4Ptr matrix);
```

Creates a Line, where `origin` is the start location of the line (x, y, z), end is the end location of the line (x, y, z), `color` is the RGBA color of the object and `matrix` is a pointer to the object's transformation matrix

```
RGraphic * MEAPI RGraphicSphereCreate( RRender *rc, AcmeReal radius,
                                       const float color[4],
                                       MeMatrix4Ptr matrix);
```

Creates a Sphere, where `radius` is the radius of the sphere, `color` is the RGBA color of the object and `matrix` is a pointer to the object's transformation matrix. The argument `rc` is the render context into whose 3D list the resulting `RGraphic` is put. This function returns a pointer to the resulting `RGraphic`, or 0 for failure.

```
RGraphic * MEAPI RGraphicSquareCreate( RRender *rc, AcmeReal side,
                                       const float color[4],
                                       MeMatrix4Ptr matrix);
```

Creates a Square, where `side` is the length of the side of the square, `color` is the RGBA color of the object and `matrix` is a pointer to the object's transformation matrix. The argument `rc` is the render context into whose 3D list the resulting `RGraphic` is put. This function returns a pointer to the resulting `RGraphic`, or 0 for failure.

```
RGraphic * MEAPI RGraphicTorusCreate( RRender *rc,
                                      AcmeReal innerRadius,
                                      AcmeReal outerRadius,
                                      const float color[4],
                                      MeMatrix4Ptr matrix);
```

Creates a Torus, where `radius` is the outer radius of the torus, `color` is the RGBA color of the object and `matrix` is a pointer to the object's transformation matrix. The argument `rc` is the render context into whose 3D list the resulting `RGraphic` is put. This function returns a pointer to the resulting `RGraphic`, or 0 for failure.

```
RGraphic * MEAPI RGraphicTextCreate( RRender *rc, char *text,
                                     AcmeReal orig_x, AcmeReal orig_y,
                                     const float color[4] );
```

Creates a `RGraphic` representing text. The `RGraphic` is placed in the 2D list. It uses “font” as the texture. The argument `rc` is the render context into whose 2D list the resulting `RGraphic` is placed, `text_in` is the text to display (this is parsed first, allowing for variable substitution to take place in RParseText). The values `orig_x` and `origin_y` represent the `x` co-ordinate of the left edge of the text and the `y` co-ordinate of the top edge of the text respectively. The argument `color` The RGBA color of the text. This function returns a pointer to the resulting `RGraphic`, or `0` for failure.

```
RGraphic * MEAPI RGraphicFrustumCreate( RRender *rc,
                                        AcmeReal bottomRadius,
                                        AcmeReal topRadius,
                                        AcmeReal bottom, AcmeReal top,
                                        int sides, const float color[4],
                                        MeMatrix4Ptr matrix );
```

Creates an arbitrary frustum. The argument `rc` is the render context into whose 3D list the resulting `RGraphic` is placed. The values `bottomRadius` and `topRadius` represent the radius of the approximation to a circle that forms the bottom and the top of the frustum respectively. The values `bottom` and `top` represent the z co-ordinates, in the frustum's reference frame, of the bottom and the top of the frustum respectively. The argument `sides` is the number of sides of the regular polygon at each end of the frustum, `color` is the RGBA color of the graphic and `matrix is` a pointer to the local-world transform for this graphic. This function returns a pointer to the resulting `RGraphic`, or `0` for failure.

Manipulating RGraphic Objects

```
int MEAPI RGraphicLineMoveEnds( RGraphic *lineG,
                               AcmeReal *origin, AcmeReal *end );
```

Moves the ends of an `RGraphic` which is a line. The argument `lineG` is a pointer to the RGraphic representing a line, `origin` is a three-vector containing the co-ordinates of the start of the line and `end` is the three-vector containing the co-ordinates of the end of the line. This function returns `0` for success or `1` for failure (an `MeWarning` will be printed before returning in this case).

```
void MEAPI RGraphicSetTransformPtr( RGraphic *g, MeMatrix4Ptr matrix );
```

Sets the transform pointer for an `RGraphic`. The argument `g` is the `RGraphic` in question and `matrix` is the transformation matrix to assign to g.

```
void MEAPI RGraphicSetColor( RGraphic *g, const float color[4] );
```

Sets the color of an `RGraphic`. The argument `g` is the `RGraphic` in question and `color` is the RGBA color to assign to the `RGraphic`. Note that this sets the ambient and diffuse components of the color and that the emissive and specular components are zeroed.

```
void MEAPI RGraphicGetColor( RGraphic *g, float color[4] );
```

Gets the color of an `RGraphic`. The argument `g` is the `RGraphic` in question and `color` is the returned ambient RGBA color of the `RGraphic`

```
void MEAPI RConvertTriStripToTriList( RGraphic* rg, RObjectVertex* strips,
int* stripSize, int* stripStart, int numStrips );
```

Converts a set of triangle strips to a list of triangles. Useful for back ends that do not support triangle strips, converting convex hulls and meshes to triangle lists. The parameter `rg` is ignored at present and `strips` is a pointer to the first vertex of the first strip to be converted. All vertices for each strip follow in one contiguous chunk. The argument `stripSize` is an array whose i^{th} element contains the number of vertices in the i^{th} strip, `stripStart` is an array whose i^{th} element contains the index in‘strips’ of the first vertex of the i^{th} strip and `numStrips` is the number of strips to process.

Render Lists

Graphics objects are considered to be of one of three types: 2D, 3D, and particle systems. When the renderer draws a frame, it walks through three linked lists, one for each type. Particle systems will be discussed in *Particle Systems* on page 27. The other two lists are of `RGraphic` objects. `RRender` holds the first element of each list, and the `RGraphics` themselves point to the next in the list.

```
void MEAPI RGraphicAddToList(RRender *rc, RGraphic *rg, int is2D);
```

Puts `RGraphic` into render-list in render context `rc`. The argument `is2D` determines whether `rg` is added to 3D or 2D list

If a `RGraphic` object has been created, but is not in a list, it will not be rendered. This provides a mechanism for *disabling* objects:.

```
void MEAPI RGraphicRemoveFromList( RRender *rc, RGraphic *rg, int is2D);
```

Removes `RGraphic` from render-list. The argument `is2D` specifies whether to look in 2D or 3D list

Menu System

The Viewer implements a simple menu system to simplify the use of an MeViewer2 application.

```
RMenu* MEAPI RMenuCreate( RRender* rc, const char* name );
```

Create a new on-screen menu. The argument `rc` is the render context that will display the menu and `name` is the title of the new menu. This function returns a pointer to the new menu.

```
void MEAPI RMenuDestroy( RMenu* rm );
```

Destroy a menu. The argument `rm` is the menu to destroy.

```
void MEAPI RRenderSetDefaultMenu( RRender *rc, RMenu* menu );
```

Make `menu` the default menu in a given render context. This means that this menu will be the one that appears when the menu key is pressed. The argument `rc` is the render context in which the menu is to be made the default.

```
void MEAPI RMenuDisplay( RMenu* rm );
```

Display a menu. The argument `rm` is the menu to display. Note that the render context in which this menu will be displayed is found out from rm.

```
void MEAPI RMenuAddToggleEntry( RMenu* rm, const char * name,
RMenuToggleCallback func, MeBool defaultValue );
```

Add a `toggle' entry to a menu. This is an entry type that has two states, on and off, which are toggled by pressing the button when the entry is highlighted. The callback is called with the new value for each change of state.

The argument `rm` is the menu to which the entry will be added, `name` is the text to be displayed for this menu entry and `func` is the function that will be called when this menu entry is selected. The value `defaultValue` is the initial value for this toggle when it is created

```
void MEAPI RMenuAddValueEntry( RMenu* rm, const char * name,
                               RMenuValueCallback func,
                               MeReal hi, MeReal lo,
                               MeReal increment, MeReal defaultValue );
```

Add a *value* entry to a menu. This is a menu entry type which provides a variable which can be modified by a specified stepsize between a minimum and maximum value by pressing the appropriate buttons when the entry is highlighted. The callback is called with the new value for each change of state.

The argument `rm` is the menu to which the entry will be added, `name` is the text to be displayed for this menu entry and `func` is the function to be called when the value is changed. The values `hi` and `lo` specify, respectively, the maximum value and minimum value this entry can take. The value `increment` is the amount by which the value is changed for each button press and `defaultValue` is the starting point for the value.

```
void MEAPI RMenuAddSubmenuEntry( RMenu* rm, const char * name,
                                 RMenu* submenu );
```

Add a *sub-menu* entry to a menu. This is an entry type that, when selected, opens another menu. The argument `rm` is the menu to which the entry will be added, `name` is the text that represents that menu entry and `submenu` is the menu that will be displayed when this entry is selected.

Help System

Any string that is recognized as a variable, but does not appear in the list above will be replaced by the text $<$unknown$>$.

```
void MEAPI RRenderCreateUserHelp( RRender *rc, char *help[],
                                  int arraySize );
```

Builds the RGraphic representing user help. The argument help is an array of null-terminated strings and arraySize is the number of elements in the array.

```
void MEAPI RRenderToggleUserHelp( RRender *rc );
```

Toggles the display of user help. Called by platform-specific back-end. Toggles the pause state of associated render context. The rc argument is the render context for which to toggle the display of the help text.

The text passed to this function will be put through RParseText, and should make use of the variables that this function substitutes to describe the controls for an application.

The Camera

The camera position is set using spherical polar coordinates to specify a position relative a specified look-at point. The spherical polar coordinates are the distance from the lookat point, the angle `theta` on the xz-plane anticlockwise from the z-axis, and elevation `phi` above the xz plane. Angles are specified in radians, `theta` in the range from $-\pi$ to π and `phi` in the range $-\pi/2$ to $\pi/2$. Whilst these values are held in the `RRender` structure, their values should not normallybe set or read directly, but rather through the use of the functions listed in this section.

```
void MEAPI RCameraSetLookAt( RRender *rc, const AcmeVector3 lookAt );
```

This sets the look-at point to the specified world position `lookAt`. The camera position is automatically updated.The argument `rc` is the render context of the camera whose lookat to be set.

```
void MEAPI RCameraGetLookAt( RRender *rc, AcmeVector3 camlookat );
```

The look-at point in world coordinates of the `rc` render context is stored into the `camlookat` vector.

```
void MEAPI RCameraSetView( RRender *rc,
                          AcmeReal dist, AcmeReal theta, AcmeReal phi );
```

This causes the camera's position to be calculated and updated from the `RRender` look-at point and the coordinates supplied. The `theta` and `phi` angles specify the angle and elevation in radians.

```
void MEAPI RCameraGetPosition( RRender *rc, AcmeVector3 campos );
```

The camera's position in world coordinates is calculated and stored into the `campos` vector.

Camera Movement

MeViewer provides a selection of functions for moving the camera by an incremental distance or angle. These are

- `RCameraZoom( RRender *rc, AcmeReal dist );`
- `RCameraPanX( RRender *rc, AcmeReal dist );`
- `RCameraPanY( RRender *rc, AcmeReal dist );`
- `RCameraPanZ( RRender *rc, AcmeReal dist );`
- `RCameraRotateAngle( RRender *rc, AcmeReal d_theta );`
- `RCameraRotateElevation( RRender *rc, AcmeReal d_phi );`

The argument `rc` is the render context whose camera is to be manipulated, `dist` is the displacement added to current camera distance, and `d_theta` and `d_phi` are the two rotation angles in spherical coordinates. Note that these functions will not allow the camera to get closer than `0.01f` from look-at. For more details, consult `MeViewer.h`.

```
void RCameraUpdate( RRender *rc );
```

Calculates the camera position and updates the camera matrix in the `RRender` structure from the look-at and spherical coordinates in that structure. The argument `rc` is the render context whose camera is to be manipulated.

It effectively synchronizes the various camera variables. It does not need to be called after using the other camera functions detailed in this section, but if any values are altered directly it should be called in order that the changes take effect.

Lighting

The number and type of lights in a render context is fixed. They are

- One ambient light source
- Two directional light sources
- One point light

The functions `RSwitchLightOn` and `RSwitchLightOff` switch lights on and off.

```
void RLightSwitchOn( RRender *rc, RRenderLight light);
```

Switch on `light`

```
void RLightSwitchOff( RRender *rc, RRenderLight light);
```

Switch off `light`

Ambient Lighting

The RGB value of the ambient light is held in the `m_rgbAmbientLightColor[3]` member of `RRender`. Any changes to this will take effect immediately (From the beginning of the next frame). The `m_bUseAmbientLight` member of `RRender` can be set to zero to disable the ambient lighting.

Directional Lighting

`RRender` holds two `RDirectLight` structures (`m_DirectLight1` and `m_DirectLight2`) that describe the directional lights. These contain the RGB values of the ambient, specular and diffuse components of the lights, as well as a 3-vector that defines the direction in which the light points. The light is active if the `m_bUseLight` member is non-zero.

If alterations to the `RDirectLight` structures are made after `RRun` has been called, then it is necessary to set the `m_bForceDirectLight1Update` or `m_bForceDirectLight2Update` members of `RRender` to a non-zero value to instruct the renderer to update the lighting for the next frame.

Point Light

RRender holds a single RPointLight structure, m_PointLight, that defines the point light. As with the directional lights, this structure holds the RGB values and active state of the light. In place of direction, the 3-vector m_Position[3] member defines the location of the light in world coordinates. The attenuation of the light is controlled by the members m_AttenuationConstant, m_AttenuationLinear and m_AttenuationQuadratic.

As with the directional lights, if alterations to m_PointLight are made after RRun, then m_bForcePointLightUpdate should be set to a non-zero value.

Textures

The Viewer supports a maximum of 25 different textures. These should be 128X128X24bpp Windows `.bmp` files. It also supports 256X256 images, but as these take 4 times the memory, one should take care to reduce the number of textures used appropriately -- this will not be enforced automatically.

Every time a new texture is specified for an object, an identifier is created for it. The files are loaded when `RRun` is called. This means that all textures should have an identifier before the call to `RRun`. See `RCreateTextureID` below for details.

```
int MEAPI RRenderTextureCreate( RRender *rc, char *filename )
```

Creates a Texture ID for `filename`. The argument `rc` is the render context into which the texture will be loaded and `filename` is the name of the texture file to attempt to load. Returns an ID for a texture filename or returns `-1` if all IDs are allocated.

```
int MEAPI RGraphicSetTexture( RRender *rc, RGraphic *rg, char *filename );
```

Sets the texture of a `RGraphic`. The argument `filename` specifies the name of the texture (the filename should not include the extension). Returns an ID for a texture filename or returns `-1` if all IDs are allocated.

Disabling an object's texture

To disable an object's texture, set its texture ID to `-1`. See *Geometry Data Format* on page 30 to find where this is stored. It may also be achieved by calling `RSetTexture` with an invalid filename, but this will have a larger overhead.

Controls

The Viewer provides ten buttons and joystick and mouse analog controls to an application. The application can assign a function to each of these using the following identifiers.

Button Press Controls

Exactly which key corresponds to which call-back being called is determined by the platform-specific layer. The call-back will be invoked only when the button is pressed, with the single argument specifying which render context is calling the function.

- `Up`
- `Up2`
- `Down`
- `Down2`
- `Left`
- `Left2`
- `Right`
- `Right2`

Each of these takes a `RRender*` render context and a function pointer as arguments:

```
void MEAPI RRenderSet*CallBack ( RRender *rc, RButtonPressCallBack func );
```

Sets the call-back for *.The wildcard symbol * refers to one of the above identifiers. The argument `rc` is the render context whose callback is to be set and `func` is the callback to assign to this button

Other call-back functions

```
void MEAPI RRenderSetActionNCallBack( RRender *rc, int N,
                                      RButtonPressCallBack func,
                                      void *userdata );
```

Sets the call-back for the Nth Action. The argument `rc` is the render context whose callback key is to be assigned. The value `N` represents the number of the callback which is to be set (usually from 0 to 5). The argument `func` is the function to call when the specified action key is pressed.

```
void MEAPI RRenderSetActionNKey ( RRender* rc, const unsigned int N,
                                  const char key );
```

Assigns a key to a given Action Callback. The argument `rc` is the render context whose callback key is to be assigned. The value `N` represents the number of the callback to assign the key to (usually in the range 2-5). The argument `key` is the key character to assign to the nth action callback.

Analog Controls

The analog call-back will be called when the platform-defined analog control has *changed* position. The arguments to the call-back specify the associated render context and the current (x, y) position of the controller. The range of these position values is *not* specified, and so the call-back should only use the difference in values between successive calls to be truly cross-platform compatible.

Here is the mouse function:

```
void MEAPI RRenderSetMouseCallBack( RRender *rc, RMouseCallBack func,
                                    void *userdata );
```

Sets the mouse call-back. The argument `rc` is the render context whose callback is to be set and `func` is the callback to assign to this button

Here is the joystick function:

```
void MEAPI RRenderSetJoystickCallBack( RRender *rc, RJoystickCallBack func
                                       void *userdata);
```

Sets the joystick call-back. The argument `rc` is the render context whose callback is to be set and `func` is the callback to assign to this button

Particle Systems

The Viewer supports an unlimited number of particle systems, each with an unlimited number of particles. The `RParticleSystem` objects can be considered like `RGraphic` objects, using a similar linked list. Particles are drawn as single textured triangles that always face the camera. Note that particle systems are not currently supported on PS2..

```
RParticleSystem* MEAPI RParticleSystemCreate ( RRender *rc,
                                               int numParticles,
                                               MeReal *positions,
                                               char *tex_filename,
                                               const float color[4],
                                               AcmeReal tri_size );
```

Allocates memory for a `RParticleSystem`, fills in the structure, and adds it to the particle system render-list in the specified render context. The `numParticles` argument specifies how many particles are defined in the `*positions` array. This array is of *`4-vectors`* that specify the position of each particle. The 4th element of this vector is not used. The texture filename `*tex_filename` should be specified as for `RSetTexture` (see *Textures* on page 24) and has the same implications as any other texture. The `tri_size` argument specifies the length of the side of the triangle that represents each particle in world coordinates. The return value will be zero if the system could not be created.

```
void MEAPI RParticleSystemAddToList( RRender *rc, RParticleSystem *ps )
```

This adds a particle system to the render context's render list. It is called automatically by `RParticleSystemCreate()`. The argument is `rc` the renderer to whose 3D list the particle system will be added and `ps` is the particle system to add.

```
void MEAPI RParticleSystemRemoveFromList( RRender *rc,
                                          RParticleSystem *ps );
```

This removes a particle system from the render context's render-list, so it will no longer be displayed. The argument is `rc` the renderer to whose 3D list the particle system will be removed and `ps` is the particle system to remove.

```
void MEAPI RParticleSystemDestroy( RParticleSystem *ps );
```

This will free the memory associated with the particle system `ps`. It should *not* be called on a particle system that is still in a render-list. The function `RRemovePSystemFromList()` should be used to remove the particle system from any render-lists before it is destroyed.

Performance Measurement

```
RPerformanceBar * MEAPI RPerformanceBarCreate( RRender *rc );
```

This function will add a performance bar to the render context `rc`. It is the responsibility of the renderer to update this bar with timings. This function returns a pointer to the newly created performance bar, or `0` if it fails to do so.

```
void MEAPI RPerformanceBarUpdate( RRender *rc, AcmeReal coltime,
                                  AcmeReal dyntime, AcmeReal rentime,
                                  AcmeReal idletime );
```

Updates the performance bar. This is called by the platform-specific back-end. The argument `rc` is the render context whose performance bar is to be updated. The argument `coltime` is the time taken by the collision detection update in the last frame, `dyntime` is the time taken by the dynamics simulation update in the last frame. The argument `rentime` is the time taken by the rendering in the last frame and `idletime` is the amount of idle time in the last frame (e.g. waiting for VSync).

To measure the frame per second (fps) of an application:

```
void MEAPI RRenderDisplayFps( RRender *rc, AcmeReal fps );
```

Creates the FPS RGraphic. The RGraphic is added to the 2D list.This is called by the platform-specific back-end. The argument `rc` The render context in which the frame rate will be displayed and `fps` is the current frame rate.

Utilities

```
void MEAPI RParseText( RRender *rc, char *text_in, char *text_out,
                       int outbuffersize );
```

Parses text, substituting text for variables. Variables defined by $ followed by capitals or numbers are substituted by text in `RRender`. The argument `rc` is the rendering context source for strings to be substituted. The argument `text_in` is the input string containing variables to be substituted and `text_out` is the output string returned with text substituted for variables. The value `outbuffersize` represents the amount of memory you allocated that is pointed to by `text_out` (i.e., maximum size of output string).

Any series of capital letters or numbers following a `$` character will be considered a variable for substitution. Those that are currently recognized are

- `$UP $DOWN $LEFT $RIGHT`
- `$UP2 $DOWN2 $LEFT2 $RIGHT2`
- `$ACTION1 $ACTION2 $ACTION3 $ACTION4 $ACTION5`
- `$APPNAME`

When these are found, they are replaced by the text held in the `RRender` structure. This allows the platform-specific renderer to provide names for the buttons and controls that are described in *Controls* on page 25. The application should set the `m_AppName` member of `RRender` to a null-terminated string that names the application.

The following functions are called by the `RGraphicLoad()` function:

```
void MEAPI RGraphicScale( RGraphic *rg,
                     AcmeReal xScale, AcmeReal yScale, AcmeReal zScale);
```

Scales a `RGraphic` using `xScale`, `yScale` and `zScale` as the multiplier for the `x`, `y` and `z` coordinates of each vertex respectively. Note that the `RGraphic` must be centered on `(0,0,0)` in model space.

```
void MEAPI RGraphicNormalize( RGraphic *rg, int bKeepAspectRatio );
```

Makes sure `RGraphic` lies in `[-1, 1]` on all axes and is centered on `(0,0,0)`. The value `bKeepAspectRatio` specifies whether to maintain aspect ratio when normalizing

```
void MEAPI RGraphicFillObjectBuffer( char *filename );
```

Fills `R_ObjectFileBuffer`. The argument `filename` is the name of the file to load. Makes sure that `R_ObjectFileBuffer` contains the contents of the specified file. `R_ObjectFileBuffer` is null terminated. Currently checks if `*filename` was the last file loaded.

Geometry Data Format

As well as being able to render primitives, the viewer can display any object described by a triangle list. These can be loaded from files, as covered in *Object Geometry Files* on page 32, or created at run-time by using the structures detailed below.

An object in MeViewer consists of a single `RObjectHeader` structure, followed by a number of `RObjectVertex` structures, each of which describes a single vertex in the triangle list. The `RGraphic` structure holds pointers to these as well as providing the linked-list mechanism.

Creating New Objects

The functions `RGraphicCreate()` and `RGraphicCreateEmpty()`, described in *Creating Objects* on page 10, should be used to allocate the memory and fill in the `RGraphic` structure for any object. It is possible to create an object with any number of vertices, but it should be borne in mind that only multiples of three will produce *valid* objects (remember that MeViewer uses triangle lists and not strips).

Consisting of merely a triangle list, an object is simply a set of triangles that share the same transformation matrix, color and texture.

The `RGraphic` structure holds pointers to the `RObjectHeader` and the first `RObjectVertex` in an object. The first `RObjectVertex` structure *must* immediately follow the `RObjectHeader` in memory, and so the pointer to it in `RGraphic` is purely for ease of programming vertex manipulation routines. The `m_pLWMatrix` member points to the transformation matrix for the object.

The member `m_nMaxNumVertices` should always be set to the maximum number of vertices that have memory allocated for them following the `RObjectHeader`. This is not necessarily the same as the number of vertices in the object, which is held in the `RObjectHeader`. This allows objects with varying numbers of vertices to be created with a minimum of memory allocation and copying.

Every graphical object in MeViewer has a single `RObjectHeader`. Its members are:

Member	Description
`m_Matrix`	This is filled in by the renderer, but should be set to the identity matrix if the `m_pLWMatrix` of the parent `RGraphic` is null.
`m_nNumVertices`	This indicates the number of RObjectVertex structures that make up the object. It *must* be a multiple of three.
`m_nTextureID`	The identifier for the object's texture. See *Textures* on page 24 for details. If it is -1 then the object is not textured.
`m_bIsWireFrame`	If this is non-zero, the renderer is requested to draw the object in wire-frame mode. This is not implemented on all platforms.

Member	Description
m_ColorAmbient	The RGBA ambient color of the object.
m_ColorDiffuse	The RGBA diffuse color of the object.
m_ColorEmissive	The RGBA emissive color of the object.
m_ColorSpecular	The RGBA specular color of the object.
m_SpecularPower	A value that indicates the shininess of the object.

RObjectVertex

Each vertex in a MeViewer object consists of a position (m_X, m_Y, m_Z), a normal (m_NX, m_NY, m_NZ) and a pair of texture coordinates (m_U, m_V). These can be updated at any time, and the changes will be reflected as soon as the next frame is drawn.

The normal should have a modulus of 1. Each texture coordinate should be between 0 and 1.

Object Geometry Files

The viewer provides the ability to load geometries from file, and indeed at the time of writing, all the primitives created by calls to `RCreate*` described in *Creating Primitives* on page 11 are loaded from disk. The file holds only vertex data, not object color or texture information. It is used to create the `RObjectVertex` structures described in *RObjectVertex* on page 31. To do this, the data required are

- The number of vertices described in the file that make up the object. This must be a multiple of three.
- For each vertex:
  - The position (x, y, z)
  - The normal (nx, ny, nz)
  - The texture coordinates (u, v)

File Format

The object geometry files are ASCII text files with the extension `.meg`. The parser in `MeViewer.c` simply looks for a series of numbers in order, each preceded by a colon ':' and followed by a space. However, it is recommended that files are created in the format described here.

On a new line, after any introductory comments (that must not contain any colons), the number of vertices must be specified as

```
vc:# /
```

where # is the number of vertices in the object, which must be a multiple of three.

Each vertex must specified on a new line. It is recommended that a blank line be left after each set of three vertices, so that the triangles are clearly defined. The vertices are specified as

```
x:# y:# z:# nx:# ny:# nz:# u:# v:# /
```

where # represents a floating-point value. This may be in any format recognised by the C-library function `atof()`. The normal should have modulus 1 and both u and v should be between 0 and 1.

This `RGraphicSave` function can be used to produce a viewer object geometry file from a valid `RGraphic` object that might, for example, have been created procedurally. This may also be used in conjunction with `RNormalizeGraphic` to normalize an object in an existing `.meg` file so that `RNormalizeGraphic` is not called every time the object is loaded.

```
int MEAPI RGraphicSave( RGraphic *rg, char *filename );
```

Creates a geometry file from a `RGraphic`. The argument `filename` is the name under which to save the geometry.

To load a file

```
RGraphic * MEAPI RGraphicLoad( char *filename );
```

Creates new `RGraphic` and fills in vertices from file. Note that `RObjectHeader` is not filled in. Returns a pointer to the resulting `RGraphic`, or `0` for failure.

Blender

Blender is a “free complete 3D animation suite”, available for download from www.blender.nl which provides access to its meshes using Python scripts. The file `blender2meg.py` is a script that exports Blender meshes in the viewer `.meg` format. Consult the Blender documentation for details of how to run the script. Note that texture coordinates are only exported if made ‘sticky’ in Blender.

Back-End Interface

As described in *Chapter 1 • What Can the Karma Viewer Do?*, MeViewer is designed to support any platform-specific *back-end* that is capable of drawing and lighting textured triangles. This chapter describes the responsibilities of a back-end to MeViewer and how it should interact with the platform-independent section.

Only two functions in the back-end will be called by `MeViewer`. These are pointed to by `RInit` and `RRun`. The pointers are set by `RRenderContextCreate`, which subsequently calls `RInit`. The function `RRun` is called by the application.

Initialization Function

The initialization function is that which is pointed to by `RInit` and shall be referred to simply as `RInit`. It takes a `RRender*` as its argument and returns an integer which should be zero for success, or non-zero to indicate failure.

```
int (*RInit) (RRender *)
```

`RInit` should initialize any data structures needed by the back-end including render surfaces. It should also remember the `RRender*` that is passed to it, and the back-end use that for all render context data.

`RInit` should also set the `RRender` members `m_ButtonText[]` to the names of the controls that will trigger the call-backs listed in *Controls* on page 25. The members are

- `m_ButtonText[0]` Up call-back trigger name (e.g. “Up Arrow”)
- `m_ButtonText[1]` Down call-back trigger name (e.g. “Down Arrow”)
- `m_ButtonText[2]` Left call-back trigger name (e.g. “Left Arrow”)
- `m_ButtonText[3]` Right call-back trigger name (e.g. “Right Arrow”)
- `m_ButtonText[4]` Up2 call-back trigger name (e.g. “W”)
- `m_ButtonText[5]` Down2 call-back trigger name (e.g. “S”)
- `m_ButtonText[6]` Left2 call-back trigger name (e.g. “A")
- `m_ButtonText[7]` Right2 call-back trigger name (e.g. “D”)
- `m_ButtonText[8]` Action1 call-back trigger name (e.g. “Space")
- `m_ButtonText[9]` Action2 call-back trigger name (e.g. “Enter”)
- `m_ButtonText[10]` AnalogX call-back trigger name (e.g. “Shift-Drag X”)
- `m_ButtonText[11]` AnalogY call-back trigger name (e.g. “Shift-Drag Y”)

RInit may also perform tasks such as printing platform-specific help to the console.

```
void (*RRun)( RRender *rc, RMainLoopCallBack func );
```

The function pointed to by RRun will be called after the render-context has been established, and the application is ready to enter the render-loop. The RRender* passed to it can be ignored, as it is not valid for this to be different to that passed to RInit.

Textures

As described in *Textures* on page 24, the texture files should be loaded into memory when RRun is called. The filenames of the textures are stored in the RRender member m_TextureList[25]. The index in the array corresponds to the texture ID as held in RObjectHeader structures, and the array element is either zero to indicate that no texture is associated with that ID, or a pointer to the filename null-terminated string. It cannot be assumed that used texture IDs are contiguous, so all 25 elements must be checked.

```
void MEAPI RBmpLoad( RRender* rc, char *filename, RImage *p_image,
                     int bRequireBGR );
```

Loads a 128*128*24 or 256*256*24 .bmp file. Fills in RImage struct, and creates 32bit bitmap at m_pImage of RImage. This allocates memory that must be freed later. The argument filename is the name of the file to attempt to load. The argument p_image will store the pointer to the block of memory containing the image after loading. Note that you should not allocate memory yourself and pass a pointer in - this will be ignored since the memory is allocated inside this function. The value bRequireBGR sets the output in format BGRA rather than the standard RGBA.

The image is in bottom-to-top, left-to-right 32bit format. If bRequiresBGR is zero then the color format is RGBA, otherwise it is BGRA.

It is the responsibility of RRun to free the memory pointed to by m_pImage when it has finished with it.

Profiling

The back-end is responsible for controlling MeProfile. Before rendering begins, it should call MeProfileStartTiming(), with the m_bProfiling member of RRender indicating whether MeProfile should log all data or just that for the previous frame. Within the render loop, a call to MeProfileStartFrame() should be made at the beginning and end of each frame, and calls to MeProfileStopTimers() and MeProfileEndFrame() at the end of each frame. Additionally, a MeProfileStartSection and MeProfileEndSection pair with the identifier *Rendering* should be put around the frame drawing code. Another section “Idle Time” should be put around any frame-rate locking code.

The timings from MeProfile should be used to call `RPerformanceBarUpdate()` with the times spent in the last frame on the sections relating to collision, dynamics, rendering and idle-time. Additionally, the back-end should calculate the frame-rate (in frames per second) and periodically call `RDisplayFPS` with this value.

When the back-end exits, if `m_bProfiling` is set, `MeProfileOutputResults` should be called. `MeProfileStopTiming` must always be called.

Camera Movement

The back-end should associate some controller input (e.g. mouse movement) with camera movement. It is entirely up to the back-end how this is implemented, but it should make use of the API calls described in *The Camera* on page 20.

User Help

A controller input (e.g. pressing F1) should result in a call to `RToggleUserHelp`.

Controller Call-backs

The back-end should associate ten button-press inputs with the ten call-backs. It should also associate an analog input with the analog call-back. The pointers to the functions that should be called are held in the `RRenderCallBacks` structure pointed to by the `m_pCallBacks` member of `RRender`. Before calling them, the back-end must check that they are non-zero, as they do not all have to be assigned by every application.

The button-press call-backs should be invoked whenever the associated buttons are pressed, with the back-end's render context being passed as the argument.

The analog callback should be called only when the (x, y) value of the controller is changed. The argument to the function should be the back-end's render context and the x and y values of the controller. These can be any floating point values that represent the position of the controller.

The Render Loop

The render loop should perform the following until the user quits:

- Swap front and back buffers.
- If the `m_bPause` member of the render context is zero, and the call-back passed as the argument to RRun is non-zero, then call the call-back.
- Call `RUpdateGraphicMatrices`.
- Draw the frame to the back-buffer (see *Drawing the Frame* on page 37).
- Check if the `m_bQuitNextFrame` member of the render-context is set. If it is, then quit next frame.
- Update the performance bar as described in *Profiling* on page 35.

Drawing the Frame

The back-end should provide a viewport with the aspect ratio of 640:448. As triangles are not z-sorted, alpha-blending should not be used except for 2D objects. Gouraud shading should be used by default.

Lighting and Shading

The details of the lighting structures are explained in *Lighting* on page 22. Every frame, the ambient light color should be set, and the ForceUpdate members of `RRender` checked to see if the other light sources need updating.

The Linked Lists

The particle systems, 3D `RGraphics` and 2D `RGraphics` are held in linked lists. Each `RGraphic` has a member `m_pNext` that points to the next in the list. Each `RParticleSystem` has an equivalent member called `m_pNextSystem`. The first element in each list is held in `RRender` (see the following sections for details). The last element in each list is the one whose *next* member is set to zero.

Particle Systems

- Lighting should be enabled.
- Z-comparison (depth testing) should be enabled.
- The model matrix and camera matrix should be identity.
- The projection matrix is the `m_ProjMatrix` member of the render context.
- For each member of the linked list starting with `m_pPS_First`
  - All drawing is in view coordinates.
  - Draw a triangle at each position in the RParticleSystem `m_Positions` member. This is a 4-vector, with the last element meaning nothing.
  - The triangle should be centered on the specified point with edge of the length specified in the `RParticleSystem` structure.
  - The triangle should be flat in z with the edge on (y-0.5*size) aligned to the x-axis.
  - It should be textured according to the ID for the system, with texture coordinates of (0,0), (0,1) and (1,0.5).

3D Objects

- Lighting should be enabled.
- Z-comparison (depth-testing) should be enabled.
- The projection matrix should be set to `m_ProjMatrix` in the render context.

- The camera matrix should be set to `m_CamMatrix` in the render context.
- For each `RGraphic` in the linked list starting with `m_pRG_First`
  - The model matrix should be set to the `m_Matrix` member of `RObjectHeader`.
  - Colors should be set as defined in `RObjectHeader`.
  - Texture should be set according to the ID in `RObjectHeader`.
  - Wire-frame mode may be set according to the `m_bIsWireFrame` member of `RObjectHeader`
  - The triangle list described in *Geometry Data Format* on page 30 should be drawn.

2D Objects

- Lighting should be set to full white ambient.
- Alpha blending should be enabled.
- Z-comparison (depth-testing) should be disabled.
- The projection matrix should be set to `m_ProjMatrix` in the render context.
- The camera matrix should be set to `m_CamMatrix2D` in the render context.
- Draw the `RGraphics` in the linked list starting with `m_pRG_First2D` in the same way as for 3D objects.