

MathEngine™ Viewer Developer's Guide

Alpha v0.0.5 – August 2000

© 2000 MathEngine PLC. All rights reserved.

MathEngine Viewer Developer's Guide.

MathEngine is a registered trademark and the MathEngine logo is a trademark of MathEngine PLC. All other trademarks contained herein are the properties of their respective owners.

This document is protected under copyright law. The contents of this document may not be reproduced or transmitted in any form, in whole or in part, or by any means, mechanical or electronic, without the express written consent of MathEngine PLC. This document is supplied as a manual for the MathEngine Viewer. Reasonable care has been taken in preparing the information it contains. However, this document may contain omissions, technical inaccuracies, or typographical errors. MathEngine, PLC does not accept responsibility of any kind for customers' losses due to the use of this document. The information in this manual is subject to change without notice.

Printed in Canada.

Contents

1 What Can the MathEngine Viewer Do?

Overview	2
Platforms Supported	2
Installation Procedure	2
Building the MathEngine Viewer into Applications	3

2 Using the MathEngine Viewer

Overview	6
User Interface	7
MathEngine Viewer Commands	7
Control Keys	7
Mouse Controls	8
Notes for Windows and UNIX	8
Notes for Sony Playstation2	8
Searching the Code for User Interface Details	8
Command-line Options	11
Embedding the Viewer in an Application	12
First Steps	12
Mainline Logic	12
Running the Simulation	13
Terminating the Program	14
Programming Notes	15
Render Contexts (Worlds)	15
Multiple Render Contexts	16
Lights, Camera...	16
Texturing (OpenGL only)	17
Performance Statistics	18
Reference	20

Chapter 1 • What Can the MathEngine Viewer Do?

Overview

The MathEngine Viewer is a simple 3D rendering and interactive viewing tool that is included with MathEngine products such as the MathEngine Dynamics Toolkit and the MathEngine Collision Toolkit.

NOTE: MathEngine products are designed to work well with any third-party renderer.

MathEngine products include sample programs that demonstrate the power and simplicity of the product. The purpose of the MathEngine Viewer is to allow the sample programs—and any prototype game or test code that you write—to display a 3D scene and to allow a user interact with these programs.

It provides a simple user interface—pan, rotate, zoom, change the lighting and shading, stop and restart the animation, and so forth—which you can easily add to.

Platforms Supported

The MathEngine Viewer runs on Windows 95/98, Windows NT, Windows 2000, Linux, Irix, and Sony PlayStation2. It allows you to use either OpenGL or Direct3D as the underlying graphics package and also provides some display support for PS2. Here is the distribution:

Platform	OpenGL	Direct3D	Renderware
Windows 95/98	x	x	
Windows NT	x		
Windows 2000	x	x	
Linux	x		
Irix	x		
PlayStation2			x

Installation Procedure

The MathEngine Viewer will be installed automatically when you install the MathEngine product.

Building the MathEngine Viewer into Applications

After installing a MathEngine product that uses the MathEngine Viewer, you can find the Viewer's header files and libraries in the `src/components/MeViewer` subdirectory of the product's directory tree.

For an example on how to build the MathEngine Viewer into a game or other application, have a look at any of the example programs provided.

Chapter 2 • Using the MathEngine Viewer

Overview

This chapter explains the MathEngine Viewer's user interface and explains how a programmer can embed the Viewer in an application.

User Interface

Most example programs and demonstration programs use the MathEngine Viewer both for rendering and to provide a standard user interface for viewing the rendered scene and controlling the simulation.

MathEngine Viewer Commands

For any program that uses the MathEngine Viewer, you can usually manipulate the rendered scene by using the standard viewer commands below:

Control Keys

Control Keys	Action
cursors	move camera
ctrl+cursors	move light
F1	Application-specific Help
F2	change shade model
F3	toggle texture
F4	toggle axis display
F5	toggle stats display
F6	toggle text
F7	toggle camera info
F8	toggle display lists
F9	toggle fullscreen
F10	toggle pause
F11	toggle mouse manipulation
F12	toggle renderer help
<Esc>	quit

Mouse Controls

Mouse Controls	Action
drag in center	orbit camera
drag near left edge	pan camera up/down
drag near bottom edge	pan camera left/right
drag near right edge	pan camera towards/away
drag near top edge	zoom camera in/out
SHIFT + pick graphic	translate object left/right and up/down
CTRL + pick graphic	translate object towards/away
ALT + pick graphic	rotate object

Notes for Windows and UNIX

- The standard commands are displayed automatically by the MathEngine Viewer, usually at the beginning of an application’s execution.
- F1 displays Help provided by the application program, usually for keyboard commands such as <Space> or <Enter>.
- F12 provides Help for the MathEngine Viewer itself.
- F1 and F12 stop the simulation while they display the Help. Press F1 or F12 again to resume the simulation.
- Not all programs support the pick graphics commands.
- Many demonstration and sample programs involve throwing balls from the camera towards objects in the scene. You can manipulate the camera in order to aim before you throw.
- Some programs override the default Viewer commands. Press F1 to check for any overrides and to receive application-specific Help.
- Clicking the right-mouse button on a rendered scene provides a menu of commands.

Notes for Sony Playstation2

The Sony PlayStation2 is normally used with a control pad, not a mouse and keyboard. Most of the controls listed above are not supported. You can, however, rotate the camera by using the left and right buttons on the control pad.

Searching the Code for User Interface Details

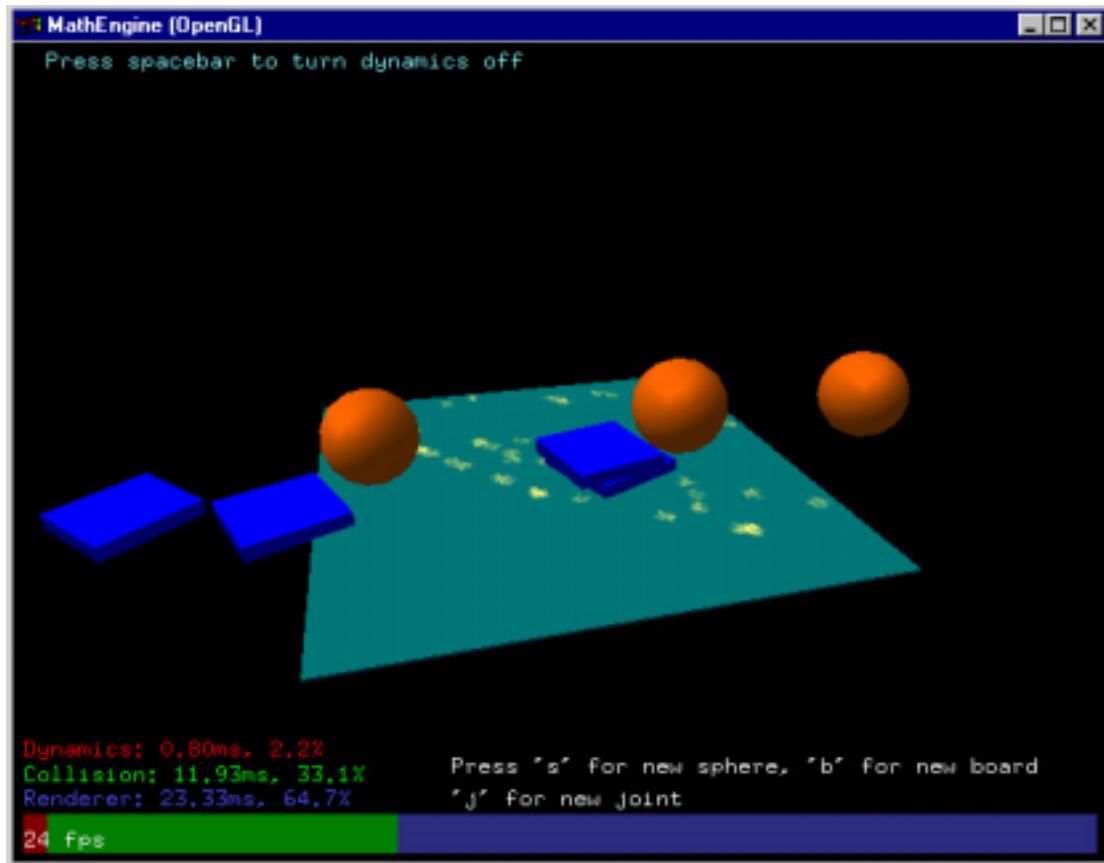
For the demo programs and example programs that are distributed with MathEngine Toolkits: if the program uses the MathEngine Viewer, you can see how application-specific commands are implemented by editing the source file of the program, and searching for RUseKey or RUsePadKey.

Here, for example, is how cd_topple implements its commands:

```
#ifndef PS2
    RUseKey('s',  stepEvolve);
    RUseKey('a',  toggleAutoEvolve);
    RUseKey(' ',  shoot);
    RUseKey('k',  killAll);
    RUseKey('w',  wakeAll);
    RUseKey('f',  toggleFriction);
    RUseKey('\r', reset);
#else

    RUsePadKey( PADUp,  stepEvolve);
    RUsePadKey( PADcircle,  shoot);
    RUsePadKey( PADRleft,  killAll);
    RUsePadKey( PADRdown,  wakeAll);
    RUsePadKey( PADtriangle, toggleFriction);
    RUsePadKey( PADSsquare, reset);

#endif
```



The mouse and keyboard controls listed above allow you to pan, zoom, pause, and so forth.

If you enter F12, you get an abbreviated version of the MathEngine Viewer Help information, with some commands missing.

If you enter F1, you get any application-specific help provided by the application program itself.

NOTE: Whenever Help is displayed, the simulation pauses. To resume the simulation hit the Help Key (either F1 or F12) again.

You (the application programmer) can also display and process other commands on the graphics screen (such as "<Enter> to restart simulation"). As well, you can display performance statistics (shown above) at the bottom left corner.)

<Esc> closes both windows and exits the program.

Command-line Options

Any application that you build using the MeViewer interprets the following command-line options:

Option	Description
-gl	OpenGL graphics (default)
-d3d	Direct3D graphics
-null	No graphics: the simulation will run with no graphics window.
-bench	Benchmarks: the simulation will run with no graphics window, but will display statistics in the console window. The application will terminate after a fixed time.
-timeout [n]	Application exits after n frames
-profile [n] [m]	Profiles for n frames after m frames of settletime

Embedding the Viewer in an Application

Let's look at the Bounce sample program from the Dynamics Toolkit. It gives a good example of how you can use the Viewer in your own programs.

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 Bounce tutorial: a ball and a plane.

```
#include "Mdt.h"
#include "MeViewer.h"

/* Render context */
RRender *rc;

/* ball graphic */
RGraphic *sphereG;

/* plane graphic */
RGraphic *planeG;
```

Mainline Logic

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

```
int main(int argc, const char **argv)
{
    const RRenderType renderer = RParseRenderType(&argc,&argv);
    .
    .
    .
}
```

We use the renderer type later on, when we initialize the Viewer:

```
/* initialise rendering attributes */
rc = RNewRenderContext(renderer, kRQualitySmooth);
```

Then we initialize the camera:

```
rc->m_cameraOffset = 40;
rc->m_cameraAngle = 0.1f;
rc->m_cameraElevation = 1.4;
rc->m_cameraLookAt[0] = 4;
```

```
RUpdateCamera();
```

We assign the <Enter> key and Space Bar key to two call-back functions, Reset and shoot .

```
RUseKey('\r',Reset);
RUseKey(' ', shoot);
```

Then we assign a function called help to become the application's help routine (called by F1, as in any MeViewer application) and we create the title of the application's window.

```
RCreateUserHelp(help, 2);
rc->m_title = "Bounce V.1.0 tutorial - www.mathengine.com";
```

Next, we create the two graphical objects, a sphere corresponding to a ball and a plane corresponding to a surface (e.g., a floor) on which the ball may fall. Note that the plane is really one of the faces of a cube:

```
sphereG = RCreateSphere(rc, 2, color, 0);
RSetTexture(sphereG, "ME_ball3");
planeG = RCreateCube(rc, 24, (MeReal)(0.2), 24, color, 0);
RSetTexture(planeG, "checkerboard");
```

Running the Simulation

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

```
RRun(rc, Tick);
```

RRun() controls the event loop (the main loop) for the simulation. Here is pseudo-code for what happens in Bounce:

```
While the user has not pressed the <esc> key [to exit the simulation])
{
  If the user enters '\r', call Reset() [callback function]
  If the user enters ' ', call shoot() [callback function]
  Call Tick() [callback function] to evolve the simulation and update
                                   the graphic transforms
  Draw the graphics
}
```

Finally, once the user has quit the simulation by pressing <Esc>, we free up all memory used by the Viewer:

```
RDeleteRenderContext(rc);
```

And that's it.

Terminating the Program

Calling `RRun()` sends the MathEngine 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 for MathEngine Toolkits:

```
void cleanup(void)
{
    MdtWorldDestroy(world); //clean up dynamics
    free(keaMemoryArea); //clean up solver work area
    RDeleteRenderContext(rc); //clean up MeViewer
}
```

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

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

Programming Notes

Render Contexts (Worlds)

To create a render context (a “world”), use this function:

```
RRender* RNewRenderContext (RRenderType, RRenderQuality);
```

Here is what the render context's structure looks like:

```
typedef struct RRender
{
    RGraphic          *m_first; /* linked list of graphics */
    void              *m_font;
    int               m_OGLfont;
    float             m_textColor[3];
    float             m_backgroundColor[3];
    RRenderQuality    m_quality; /* shading attributes */
    RKeyCallback      m_keyFuncs[NUM_KEYS];
    RKeyCallback      m_padkeyFuncs[NUM_PAD_BUTTONS];
    RMouseCallback    m_mouseFunc;
    RMotionCallback   m_motionFunc;

    /*
     * Window attributes.
     */
    char              *m_title;
    int               m_windowWidth;
    int               m_windowHeight;
    int               m_windowID;
    int               nm_ctrlDown;

    /*
     * Booleans.
     */
    int               m_showTexture;
    int               m_showAxis;
    int               m_showLight;
    int               m_showText;
    int               m_showHelp;
    int               m_showUserHelp;
    int               m_showInfo;
    int               m_showFPS;
    int               m_useDisplayLists;
    int               m_pausedDynamics;
```

```

int            m_initialized;
int            m_ended;

unsigned int    m_paddata; /* Key down info store for joypad */
/*
    Stats display variables.
*/
float          m_timePerFrame;
float          m_renderTime;
float          m_dynamicsTime;
float          m_collisionTime;
float          m_totalTime;
Timer          *m_dynamicsTimer;
Timer          *m_collisionTimer;
Timer          *m_renderTimer;

/*
    Camera and Light Parameters.
*/
AcmeReal       m_zClipFar;
AcmeReal       m_lightPos[3];
AcmeReal       m_cameraPos[3];
AcmeReal       m_cameraLookAt[3];

/*
    There five are used by the standard keyboard camera and light
    positioning functions change them to alter initial positions.
*/
AcmeReal       m_cameraOffset;
AcmeReal       m_cameraAngle;
AcmeReal       m_cameraElevation;
AcmeReal       m_lightAngle;
AcmeReal       m_lightElevation;
} RRender;

```

Multiple Render Contexts

You can create more than one render context, if needed. To switch between them use:

```
void    RSetRenderContext    (RRender* context);
```

Lights, Camera...

Here are the light and camera fields again:

```

AcmeReal      m_zClipFar;
AcmeReal      m_lightPos[3];
AcmeReal      m_cameraPos[3];
AcmeReal      m_cameraLookAt[3];
AcmeReal      m_cameraOffset; /* there five are used by the */
AcmeReal      m_cameraAngle; /* standard keyboard camera and */
AcmeReal      m_cameraElevation; /* and light positioning functions*/
AcmeReal      m_lightAngle; /* change them to alter initial */
AcmeReal      m_lightElevation; /* positions */

```

The units are in meters or radians. Here are some further notes about them:

Field Name	Default Value, Other Notes
m_zClipFar;	Distance to the far clipping plane. Any objects beyond the plane will not be rendered.
m_cameraPos[3];	Set by a call to RUpdateCamera(). See below.
m_cameraLookAt[3];	Default: {0, 0, 0}, i.e. the camera is pointing at the origin.
m_cameraOffset;	Distance from the origin. Default: 10. User interface: bound to the '+' and '-' keys.
m_cameraAngle;	The azimuth: the rotation around the origin, in the horizontal plane, starting along the z-axis, increasing counter-clockwise when viewed from above. Default: 0. User interface: bound to the cursor keys.
m_cameraElevation;	Angle of elevation above the z-axis. Default: 0. User interface: bound to the cursor keys.
m_lightAngle;	Default: $\text{PI}/4$.
m_lightElevation;	Default: $\text{PI}/4$.
m_lightPos	Default: Null

The vector that defines the position of the camera is m_cameraPos. You can set it directly, or call RUpdateCamera() to set the value of the vector based on m_cameraOffset, m_cameraAngle, and m_cameraElevation.

Texturing (OpenGL only)

With OpenGL, you can apply texture to a model by using the following:

```
RSetTexture (RGraphic* object, char* fileName);
```

In Windows, the texture file must be a bitmap (*.bmp) file. In any Unix system, the texture file must be a tiff file. The filename must not include the extension.

Performance Statistics

The Viewer has functions that allow you to track how much time different parts of your application consume. Here is how to use them:

```
void Tick(RRender* rc)
{
    RStopTimer(kRRender);
    RStartTimer(kRCollision);
    /* update collision space*/
    RStopTimer(kRCollision);
    RStartTimer(kRDynamics);
    /* evolve dynamics */
    MdtWorldStep(world, step);
    RStopTimer(kRDynamics);
    RStartTimer(kRRender);
}
```

The Viewer displays the statistics in the graphics window. F5 toggles the display of statistics. Like the display of any other text, displaying the graphics slows down the simulation.

This is very useful to get continuous visual idea of how long the application is spending doing rendering, dynamics and collision. This method is restricted to these three areas. If you want an average overall time you can use the -bench command line parameter.

However, there is a third and better way to time your code using the -profile command line parameter. This will give a much more detailed breakdown of time spent in different sections of code, specifically collision, scene partitioning, constraint creation, constraint solving and rendering, although the user is free to add as many sections as they want. Here are some examples of it's usage:

```
bridge -profile 100 - profiles first 100 frames of bridge example including rendering
bridge -profile 100 50 - profiles 100 frames of bridge including rendering after 50
                        frames of settle time.
bridge -null -profile 50 - profiles first 50 frames of bridge with no rendering.
```

In fact this command line option is making use of more general profiling code which resides in the MeGlobals lib. In this way a user is free to add their own code sections to be timed. All you have to do is `#include "MeProfile.h"` which is in MeGlobals/include. The code will look like this:

```
#include "MeProfile.h"
...
MeProfileStartSection("My code section",0);
...
MeProfileEndSection("My code section");
```

The second parameter of `MeProfileStartSection()` is `autorun`. If it is 0, profiling will continue until explicitly stopped with a call to `MeProfileEndSection()`. If it is 1 profiling will continue until the next call to `MeProfileStartSection()`. Hence it is clear that code sections can be concurrent, therefore don't expect the percentages in the output display to add up to 100%, unless you really are timing all the code sequentially.

To get output in a gnuplot format:

```
MeProfileSetOutputParameters(kMeProfileOutputGnuplot);
```

Reference

For reference information about the MathEngine Viewer's functions, structures, and so forth, read the header file.