Initialization and Termination for Applications Using Wild Magic 5

David Eberly
Geometric Tools, LLC
http://www.geometrictools.com/

Created: August 20, 2010

Contents

1 Introduction

Wild Magic 5 (WM5) has various subsystems that need to be created and initialized before dependent code may be executed. For example, if you have enabled the WM5 memory system by exposing the `WM5_USE_MEMORY` conditional define, then this system must be initialized. Any data structures created before the `main` entry function must be destroyed after the `main` function exits.

Some initialization can occur after the `main` function is entered but before the application actually starts running. For example, WM5 maintains a directory-path system that allows you to specify directories to search for data files, much like the DOS PATH environment variable. The directories must be established before the application attempts to load data files. When the application finishes running, any data structures created before the run must be destroyed.

This document summarizes the initialization and termination that occurs during a WM5 3D application. If you choose to include WM5 in your own 3D application layer, the last section of this document summarizes what you must do minimally to ensure that WM5 runs properly.

2 Pre-main Registration

The entry point into a WM5 application is the `main` function that is part of any standard C or C++ run-time library. This is true even for the 3D applications that use Microsoft's Win32 API; that is, the entry function `WinMain` is never used. The `main` function is implemented in the source file

```
GeometricTools/WildMagic5/LibApplications/Wm5Application.cpp
```

The same `main` function is called even when the application is a console application.

An application might want certain actions to be taken before `main` is entered. I call this phase the `pre-main execution`. Some actions might also be desired after `main` is exited. I call this phase the `post-main execution`.

Pre-main execution is caused by the side effects of global variables declared at file scope. For example, in a C++ environment, a global class object has its constructor called pre-main. In the same example, post-main execution includes a call to that object's destructor. In a C environment, post-main behavior is obtained by the `atexit` function.

It is well known that the order of pre-main/post-main function calls depends on the compiler and linker; the order in which source files are compiled and linked is not generally predictable. An application layer might very well contain a subsystem to allow the user to control the order, but such subsystems are tedious to write and maintain. Rather than fight this battle in WM5, the only supported pre-main/post-main behavior is to *register* functions that are to be called before the application starts running (initialization) and after the application finishes running (termination). The initialization/termination calls actually occur within the `main` entry function.

The order in which functions are registered is not guaranteed due to the same problem mentioned previous about compiler and linker dependencies. However, the developer may control initialization/termination dependencies by arranging for one such function to call another that it is dependent on. It is only natural that the developer be responsible for maintaining the directed acyclic graph of dependencies. If you have

dependencies, the developer is responsible for ensuring that the initialization and termination function for a class be called at most once.

The pre-main/post-main execution is managed by the class `InitTerm` in the `LibCore` project. The file `Wm5InitTerm.h` contains the class definition and macros for a class to use when it wants initialization before the application starts running and/or when it wants termination after the application finishes running. If a class wants initialization, the macros are used as shown next.

```
// In MyClass.h file:
class MyClass
{
    WM5_DECLARE_INITIALIZE;
    <remainder of class body goes here>;
};
WM5_REGISTER_INITIALIZE(MyClass);

// In MyClass.cpp file:
WM5_IMPLEMENT_INITIALIZE(MyClass);
```

The macros are

```
//----------------------------------------------------------------------------
#define WM5_DECLARE_INITIALIZE \
public: \
    static bool RegisterInitialize (); \
    static void Initialize (); \
private: \
    static bool msInitializeRegistered
//----------------------------------------------------------------------------
#define WM5_IMPLEMENT_INITIALIZE(classname) \
bool classname::msInitializeRegistered = false; \
bool classname::RegisterInitialize () \
{ \
    if (!msInitializeRegistered) \
    { \
        InitTerm::AddInitializer(classname::Initialize); \
        msInitializeRegistered = true; \
    } \
    return msInitializeRegistered; \
}
//----------------------------------------------------------------------------
#define WM5_REGISTER_INITIALIZE(classname) \
static bool gsInitializeRegistered_##classname = \
    classname::RegisterInitialize ()
//----------------------------------------------------------------------------
```

The `WM5_DECLARE_INITIALIZE` macro declares two static functions and a static data member. The function `RegisterInitialize` is part of the pre-main registration and has a body that is defined by the *implement* macro used in the `cpp` file. The actual call to `RegisterInitialize` occurs *at least once* because the

WM5_REGISTER_INITIALIZE is used in the h file that is included in at least one source file. Generally, the h file is included (directly or indirectly) in multiple source files, so RegisterInitialize has the potential to be called multiple times. The static data member is used to ensure that the registration function is called *at most once*. Similar macros exist for termination.

At first glance, this appears to be an inefficient way of registering the function; after all, we know that we include MyClass.h in MyClass.cpp, which should force the registration to occur. For the initialization and termination systems, we could do use the more efficient method. However, the streaming system uses the same mechanism to register the class factory functions. If a developer creates new streamable classes, places these in a library, and the application source files do not include *all* the h files from that library, the non-included h files will cause some class factories functions *not to be registered*. In fact, this problem showed up in NetImmerse/Gamebryo with streaming because we had developed the game engine to consist of multiple libraries. As it turns out, the inefficiency of the registration system is not a bottleneck in application run time.

Notice that the only responsibility of RegisterInitialize is to add the MyClass::Initialize function to a static array of initialization functions in InitTerm. These functions are called just before the application is told to start running. Similarly, the only responsibility of RegisterTerminate is to add the MyClass::Terminate function to a static array of termination functions in InitTerm. These functions are called just after the application finishes running.

The author of MyClass is responsible for implementing the MyClass::Initialize and MyClass::Terminate functions.

The InitTerm::AddInitializer and InitTerm::AddTerminator functions insert function pointers into arrays of a fixed size (currently 512 elements per array). If your application requires larger arrays, you must increase the value of InitTerm::MAX_ELEMENTS accordingly. In the Debug configuration, if an attempt is made to insert too many items, an assertion is triggered with a diagnostic message indicating you must increase the enumerate value and recompile the libraries. This approach ensures that no dynamic allocation occurs pre-main because of the needs of InitTerm.

3 The Main Function

The main entry function is implemented in Wm5Application.cpp. This is a platform-independent function common to all WM5 applications regardless of operating system and windowing system. The responsibility of main is the following:

1. Initialize the Memory system, if it is enabled.
2. Read the value of the environment variable WM5_PATH. This is the path to where WM5 is installed, generally the path leading to the GeometricTools/WildMagic5 subdirectory.
3. Call the initialization functions that were registered pre-main.
4. Insert relevant directories into the path system for use by WM5 applications. Directories include those to compiled shaders (*.wmfx), scene object files (*.wmof), texture images (*.wmtf), vertex and index buffers (*.wmvf), and image files (*.im). An automatically maintained path that was not part of Wild Magic 4 is the *application path*, stored in the class-static variable Application::ThePath. This subdirectory contains the project file and source code for the project.

5. Create a command parser that stores and processes the command-line parameters (if any) passed to the application.
6. Call the `Application::Run` function. At this time all data and initialization that the application requires is in place. The remaining steps listed next occur after `Application::Run` returns from execution.
7. Destroy the command parser.
8. Remove the various data directories from the system path.
9. Call the termination functions that were registered pre-main.
10. Terminate the `Memory` system, if it is enabled.

3.1 The Memory::Initialize Function

The WM5 memory system keeps track of allocations, deallocations, and the file/line on which each such operation occurs. WM5 has smart pointers, as did WM4, but WM4 smart pointers were designed so that each `Object` maintained its own reference counter. The reference counters to WM5 `Object`s are instead stored externally, in a class-static map that is managed by class `Memory`. The function `Memory::Initialize` includes the creation of this map.

The memory system also allows you to hook in your own memory allocation and deallocation. Another variation of the `Memory::Initialize` function allows you to specify an alternate allocator and deallocator.

3.2 The Environment::Initialize Function

A new class has been added to Wild Magic, namely, `Environment`. This replaces a portion of the WM4 `System` class (this class is obsolete). `Environment` supports reading environment variables, one of those being `WM5_PATH`. The class also allows you to insert and remove directories from a list of directories that is used for searching for files to open during application run time.

The function `Environment::Initialize` function simply creates the class-static list of directories, although insertion and removal functions will create the list in a lazy manner if necessary during program execution. In fact, this initializer is called by the `InitTerm::ExecuteInitializers` function, so there is not an explicit call to it in `main`.

After initialization, the `main` program inserts directories specific to the WM5 sample applications. If you do not use the WM5 application layer, you can omit these directories if your application does not need them.

3.3 The Application::ThePath Variable

This is a class-static variable that is new to Wild Magic. Its value is intended to be the path to the subdirectory that contains your application project file and source code. To ensure that this variable is correctly set, the derived-class application constructor must specify the subdirectory that contains the project. For example, in `SampleGraphics/BillboardNodes`, the constructor is

```
BillboardNodes::BillboardNodes ()
    :
    WindowApplication3("SampleGraphics/BillboardNodes",0, 0, 640, 480,
        Float4(0.9f, 0.9f, 0.9f, 1.0f)),
        mTextColor(1.0f, 1.0f, 1.0f, 1.0f)
{
}
```

The subdirectory is specified in the first argument of the base-class constructor and the path is relative to the path stored by the environment variable `WM5_PATH`.

3.4 The Command-Line Parser

Class `Application` defined in `Wm5Application.h` contains a static data member, `TheCommand`, which is a pointer to an object of type `Command`. This class is implemented in `Wm5Command.h` and `Wm5Command.cpp` in the `LibApplications` project. It is a simple class that stores the command-line parameters as an array of strings and parses them according to the rules in Command Line Parsing.

3.5 The Application::Run Function

Class `Application` is implemented in `Wm5Application.h` and `Wm5Application.cpp`. It declares a static function pointer called `Run`. `Application`-derived classes implement a function to which this pointer is directed. An application is either a *console application* having no need for a window with drawing surface (it can use a text-only console window), a *2D window application* for 2D graphics-based applications, or a *3D window application* for 3D graphics-based applications. The class `ConsoleApplication` encapsulates the console application. Both 2D and 3D window applications have some common behavior, which is encapsulated in `WindowApplication`. The 2D window applications are encapsulated by `WindowApplication2` and the 3D window applications are encapsulated by `WindowApplication3`.

Class `Application` also has a static pointer member, called `TheApplication`, that points to the unique instance of the application. Although WM5 sample applications use one window per application, the restriction to a unique instance does not prevent you from having multiple windows in an application.

The file `Wm5ConsoleApplication.h` contains macros that are used in pre-main initialization and post-main termination. These must be declared in the final application source code. The macros are

```
//----------------------------------------------------------------------------
#define WM5_CONSOLE_APPLICATION(classname) \
WM5_IMPLEMENT_INITIALIZE(classname); \
WM5_IMPLEMENT_TERMINATE(classname); \
\
void classname::Initialize () \
{ \
    Application::Run = &ConsoleApplication::Run; \
    TheApplication = new0 classname(); \
} \
```

```
\
void classname::Terminate () \
{ \
    delete0(TheApplication); \
}
//----------------------------------------------------------------------------
```

These are initializer and terminator functions that are registered pre-main as discussed previously. The initializion hooks up the `Run` pointer to the correct application type and creates an instance of the application. Similar macros for initialization and termination in windowed-applications are in `Wm5WindowApplication.h`.

For example, in the `SampleGraphics/BillboardNodes` application, you will see in the file `BillboardNodes.h` effectively

```
class BillboardNodes : public WindowApplication3
{
    WM5_DECLARE_INITIALIZE;
    WM5_DECLARE_TERMINATE;
    <other stuff>;
};
WM5_REGISTER_INITIALIZE(BillboardNodes);
WM5_REGISTER_TERMINATE(BillboardNodes);
```

In `Billboards.cpp` you will see

```
WM5_WINDOW_APPLICATION(BillboardNodes);
```

The call to `Application::Run` in the `main` entry function occurs after all initializers are called, so in fact `Application::Run` is a nonnull function pointer and may be dereferenced.

The `Run` function for a console application is

```
int ConsoleApplication::Run (int numArguments, char** arguments)
{
    ConsoleApplication* theApp = (ConsoleApplication*)TheApplication;
    return theApp->Main(numArguments, arguments);
}
```

It executes the application `Main` entry function in a way similar to what you have seen for the `main` entry function. However, `ConsoleApplication::Main` is a pure virtual function, so any application derived from `ConsoleApplication` must declare and implement `Main`. For example, see `SamplePhysics/SimplePendulum`. The application may choose to use `Application::TheCommand` for parsing command-line parameters or it may process directly the inputs to `Main`.

The `Run` function for a window application is

```
int WindowApplication::Run (int numArguments, char** arguments)
{
```

```
    WindowApplication* theApp = (WindowApplication*)TheApplication;
    return theApp->Main(numArguments, arguments);
}
```

so it has exactly the same structure as that of `ConsoleApplication`. However, `WindowApplication::Main` is not a pure virtual function. The primary window of the application must be created, but this process is specific to the operating system (MS Windows, Mac OS X) and/or windowing system (X-Windows on Linux). Thus, each platform must specifically implement `WindowApplication::Main` according to its needs. For example, if you look at the `Wm5WinApplication.cpp` file for a Microsoft Win32 application (DirectX 9 or OpenGL), there is an implementation of `WindowApplication::Main` as well as other functions that support the window (event handling, keyboard, mouse, and so on). You will notice that `WindowApplication::Main` function creates a window using the Win32 API, creates a renderer, and interfaces to the unique application instance CodeApplication::TheApplication for initialization, message handling, idle loop, and termination.

When `Main` terminates, control is returned to the `main` entry function, followed by clean-up code to free up resources and memory.

3.6 The Environment::Terminate Function

After `Application::Run` terminates, all directories are removed from the path system. The array of directory strings itself is deleted by the call to `Environment::Terminate`. In fact, this terminator is called by the `InitTerm::ExecuteTerminators` function, so there is not an explicit call to it in `main`.

3.7 The Memory::Terminate Function

If the WM5 memory system is enabled, you must terminate it. The `Memory::Terminate` function writes to a text file any memory that was allocated by the system and has not yet been deallocated. The generated report has information about the address, the number of bytes, the dimension (singleton, 1D-array, 2D-array, and so on), and the name of the source file and line number where the allocation occurred.

After the report is generated, the class-static map is destroyed.

4 Post-main Cleanup

As designed, no freeing of resources or deallocations of memory should occur during post-main execution. If you are lucky, a third-party library will also attempt to free its resources and memory before the post-main execution.

5 Example: Extension to a WGL Application not using WM5 Application

The outline shown next illustrates what you must do in your own WGL application layer that uses 3D graphics.

```
// Pre-main execution occurs first as always.  Then 'main' is called...
int main (int numArguments, char** arguments)
{
#ifdef WM5_USE_MEMORY
    // To specify your own allocator and deallocator, change this function
    // call to Memory::Initialize(yourAllocator, yourDeallocator).  See the
    // Memory class in LibCore/Memory/Wm5Memory.h for the signatures of these
    // functions.
    Memory::Initialize();
#endif

    // The Wild Magic 5 application layer depends on the directory structure
    // structure that ships with the libraries.  You need to create the
    // WM5_PATH environment variable in order for the applications to find
    // various data files.
    Application::WM5Path = Environment::GetVariable("WM5_PATH");
    if (Application::WM5Path != "")
    {
        Application::WM5Path += "/";
    }
    // else:  If your application needs WM5_PATH to be set, you need to
    // take any necessary action to let the user know the environment
    // variable does not exist.

    // Execute any registered pre-main initializers.
    InitTerm::ExecuteInitializers();

    // You can add any paths you like.  Syntax is
    //   Environment::InsertDirectory("MyDirectory");

    // *** BEGIN PLATFORM-SPECIFIC WINDOW/RENDERER CREATION ***
    //
    // OpenGL uses a projection matrix for depth in [-1,1].  For a DirectX 9
    // application, change the input to Camera::PM_DEPTH_ZERO_TO_ONE.
    Camera::SetDefaultDepthType(Camera::PM_DEPTH_MINUS_ONE_TO_ONE);

    // Setup for creating a Win32 window goes here.
    int windowWidth = *;
    int windowHeight = *;
    // ... other setup goes here ...
    HWND handle = CreateWindow(...);

    // Create a Windows OpenGL (WGL) renderer.  Change parameters as needed.
    // If you want multisampling, see Wm5WinApplication.cpp for details.
    // For creation of a DirectX 9 renderer, see Wm5WinApplication.cpp.
    Texture::Format colorFormat = Texture::TF_A8R8G8B8;
    Texture::Format depthStencilFormat = Texture::TF_D24S8;
    int numMultisamples = 0;
```

```
RendererInput input;
input.mWindowHandle = handle;
input.mPixelFormat = 0;
input.mRendererDC = 0;
input.mDisableVerticalSync = true;  // set to false if you want sync to retrace
Renderer* renderer = new0 Renderer(input, windowWidth, windowHeight,
    colorFormat, depthStencilFormat, numMultisamples);
//
// *** END PLATFORM-SPECIFIC WINDOW/RENDERER CREATION ***

// *** BEGIN APPLICATION-SPECIFIC LOGIC ***
//
// This work occurs in WindowApplication::OnInitialize.  YOU
// DO NOT HAVE TO CREATE A CAMERA AT THIS TIME.  However, you must
// have some camera attached to the renderer for any drawing.
Camera* camera = new0 Camera();
renderer->SetCamera(camera);
Float4 clearColor = <your choice>;
renderer->SetClearColor(clearColor);

// This work occurs in the DerivedApplication::OnInitialize.
float fieldOfView = <your choice>;  // degrees
float aspectRatio = ((float)windowWidth)/((float)windowHeight);
float near = <your choice>;
float far = <your choice>;
camera->SetFrustum(fieldOfView, aspectRatio, near, far);
APoint camPosition(...);  // eye point
AVector camDVector(...);  // direction of view
AVector camUVector(...);  // up vector
AVector camRVector = camDVector.Cross(camUVector);  // right vector
camera->SetFrame(camPosition, camDVector, camUVector, camRVector);

<create scene graphs and anything else you need>;
<call someScene->Update() as needed>;

// There can be multiple scene graphs.  The code below is for a
// single scene.  It is possible to compute visible sets for multiple
// scenes and combine them.  ALL THIS IS REALLY APPLICATION-SPECIFIC
// LOGIC.
Culler culler;
culler.SetCamera(camera);
culler.ComputeVisibleSet(scene);

do
{
    <Run your application code>;  // uses message pump, idle loop
}
until_finished;
```

```
    // This work occurs in DerivedApplication::OnTerminate.
    <destroy all resources and memory you used>;

    // This work occurs in WindowApplication3::OnTerminate.
    renderer->SetCamera(0);
    delete0(camera);
    //
    // *** END APPLICATION-SPECIFIC LOGIC ***

    // *** BEGIN PLATFORM-SPECIFIC WINDOW/RENDERER DESTRUCTION ***
    //
    // For destruction of a DirectX 9 renderer, see Wm5WinApplication.cpp.
    delete0(renderer);

    // This is actually handled in the message pump, but I place it here for
    // completeness.
    DestroyWindow(handle);
    //
    // *** END PLATFORM-SPECIFIC WINDOW/RENDERER DESTRUCTION ***

    // Remove all directories from the path system.
    Environment::RemoveAllDirectories();

    // Execute any registered post-main terminators.
    InitTerm::ExecuteTerminators();

#ifdef WM5_USE_MEMORY
    // Report memory leaks.
    Memory::Terminate("MemoryReport.txt");
#endif

    return 0;
}
```

6 Example: Extension to an AGL or GLX Application not using WM5 Application

The platform-specific code of the previous section must be replaced for each of these platforms (Macintosh or Linux). Specifically, the window creation and destruction, the renderer creation and destruction, and the message pump (events, input-device handling, idle loop, and so on) must be modified. See the files `Wm5AglApplication.cpp` and `Wm5GlxApplication.cpp` for examples of how I do this.