Initialization and Termination for Applications Using Wild Magic 4

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

Created: May 28, 2009

Contents

1 Introduction

Wild Magic has various subsystems that need to be created and initialized before dependent code may be executed. For example, the loading and parsing of shader programs (`*.wmsp` files) requires that a `Renderer` object be created. This creation sets various static class data that are shared by all renderers. Another example is the streaming system, which requires that factory functions be available at run time to allow you to create objects when loading `*.wmof` files. 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, Wild Magic 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 Wild Magic 4 3D application. If you choose to include Wild Magic 4 in your own 3D application layer, the last section of this document summarizes what you must do minimally to ensure that Wild Magic 4 runs properly.

2 Pre-main Registration

The entry point into a Wild Magic 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.

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 Wild Magic, 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 `Main` in the `LibGraphics` project. The file `Wm4MainMCR.h` contains 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
{
    WM4_DECLARE_INITIALIZE;
    <remainder of class body goes here>;
};
WM4_REGISTER_INITIALIZE(MyClass);

// In MyClass.cpp file:
WM4_IMPLEMENT_INITIALIZE(MyClass);
```

The macros are

```
//----------------------------------------------------------------------------
#define WM4_DECLARE_INITIALIZE \
public: \
    static bool RegisterInitialize (); \
    static void Initialize (); \
private: \
    static bool ms_bInitializeRegistered
//----------------------------------------------------------------------------
#define WM4_IMPLEMENT_INITIALIZE(classname) \
bool classname::ms_bInitializeRegistered = false; \
bool classname::RegisterInitialize () \
{ \
    if (!ms_bInitializeRegistered) \
    { \
        Main::AddInitializer(classname::Initialize); \
        ms_bInitializeRegistered = true; \
    } \
    return ms_bInitializeRegistered; \
}
//----------------------------------------------------------------------------
#define WM4_REGISTER_INITIALIZE(classname) \
static bool gs_bInitializeRegistered_##classname = \
    classname::RegisterInitialize ()
//----------------------------------------------------------------------------
```

The `WM4_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 `WM4_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*.

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 `Main`. 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 `Main`. 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 `Main::AddInitializer` and `Main::AddTerminator` functions use a lazy dynamic allocation of the static arrays of function pointers. Alternatively, if you know an upper bound on the number of initializers and terminators, you could declare the arrays using fixed element sizes rather than dynamically allocating them. This is useful if you want to ensure that no dynamic allocations occur in your application pre-main.

3 The Main Function

The `main` entry function is implemented in `Wm4Application.cpp`. This is a platform-independent function, thereby common to all Wild Magic applications regardless of operating system and windowing system. The responsibility of `main` is the following:

1. Create the "system path" data structure. This stores various data directories for use by the application. Read the value of the environment variable `WM4_PATH`. This is the path to the where Wild Magic 4 is installed, generally the path leading to the `WildMagic4` subdirectory and usually the prefix of any data directory.
2. Call the initialization functions that were registered pre-main.
3. Insert various data directories into the system path for use by the Wild Magic applications to find scene graph files (`*.wmof`), texture images (`*.wmif`), and shader programs (`*.wmsp`).
4. Create a command parser that stores and processes the command-line parameters (if any) passed to the application.
5. 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 when `Application::Run` returns.

6. Destroy the command parser.
7. Remove the various data directories from the system path.
8. Call the termination functions that were registered pre-main.
9. Destroy the global application object.
10. Destroy the system path.

3.1 The System::Initialize Function

The creation of the system path data structure is managed by the `System::Initialize` function, which is found in the `LibFoundation` project. This data structure is simply an array of strings. The `System` functions that interface the path are

```
static int GetDirectoryQuantity ();
static const char* GetDirectory (int i);
static bool InsertDirectory (const char* acDirectory);
static bool RemoveDirectory (const char* acDirectory);
static void RemoveAllDirectories ();
```

The names and inputs of the function are self-explanatory. The return value of `InsertDirectory` is `true` when the directory is actually inserted (the input is not already in the array of strings). The return value of `RemoveDirectory` is `true` when the input is actually removed (the input is in the array of strings at the time of the call).

The `System::Initialize` functoin also has the responsibility to read the value of the environment variable `WM4_PATH` and store it in the static data member `System::WM4_PATH`. If this variable has not been set by the developer, `System::WM4_PATH` is a string of length zero. The `main` function asserts this is not the case (and has a long comment about why the assertion was triggered).

3.2 The Main::Initialize Function

Previously I had described the pre-main registration system. This is only one of three responsibilities of `Main::Initialize`. In a system that uses *smart pointers* (automatic handling of reference-counted objects), it is useful to verify that all reference-counted objects have been destroyed. If they have not, the developer has overlooked proper clean-up of the application.

The class `Object` supports smart pointers (this is a heavy-weight class that will be factored in Wild Magic 5 to allow reference counting without having to drag along streaming and other subsystems). The static member `Object::InUse` is a hash table of the objects that still exist in memory. The hash table is allocated only when the first `Object` is created (lazy allocation). `Main::Initialize` asserts that the hash table pointer is null before the initializers are called; that is, the pre-main execution should never create `Object`-derived objects.

The second part of `Main::Initialize` is to create "catalogs" of images, textures, vertex programs, and pixel programs. These are used for sharing of such objects in the application. Although the objects stored

in the catalogs are `Object`-derived, the catalogs do not adjust the reference counts; that is, the catalogs are just containers for (standard) pointers and perform no actions on the objects pointed to. Although useful, I plan on revisiting the catalog system in Wild Magic 5 and determining a simpler method for sharing (I had written such a system in 2XL game code).

The third part of `Main::Initialize` is to call the initializer functions. Once called, there is no need to keep the array of function pointers in memory, so the array is deleted. Initializers can create referenced-counted objects, so it is possible that `Object::InUse` is not null. The static member `Main::ms_iStartObjects` is set to the number of reference-counted objects that exists (possibly zero). This number is used in `Main::Terminate` to verify that all reference-counted objects have been destroyed.

3.3 The Command-Line Parser

Class `Application` defined in `Wm4Application.h` contains a static data member, `TheCommand`, which is a pointer to an object of type `Command`. This class is implemented in `Wm4Command.h` and `Wm4Command.cpp` in the `LibFoundation` 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.

3.4 The Application::Run Function

Class `Application` is implemented in `Wm4Application.h` and `Wm4Application.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 Wild Magic 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 `Wm4ApplicationMCR.h` contains two macros, one for console applications and one for window applications, that must be declared in the final application source code. The macros are

```
//----------------------------------------------------------------------------
#define WM4_CONSOLE_APPLICATION(classname) \
WM4_IMPLEMENT_INITIALIZE(classname); \
\
void classname::Initialize () \
{ \
    Application::Run = &ConsoleApplication::Run; \
    TheApplication = WM4_NEW classname; \
}
//----------------------------------------------------------------------------
#define WM4_WINDOW_APPLICATION(classname) \
```

```
WM4_IMPLEMENT_INITIALIZE(classname); \
\
void classname::Initialize () \
{ \
    Application::Run = &WindowApplication::Run; \
    TheApplication = WM4_NEW classname; \
}
//----------------------------------------------------------------------------
```

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

```
class BillboardNodes : public WindowApplication3
{
    WM4_DECLARE_INITIALIZE;
    <other stuff>;
};
WM4_REGISTER_INITIALIZE(BillboardNodes);
```

Unlike what was discussed previously, the application macros mentioned in this section are used for the implementation because it is already known what the initialization semantics must be. In `Billboards.cpp` you will see

```
WM4_WINDOW_APPLICATION(BillboardNodes);
```

The call to `Application::Run` in the `main` entry function occurs after all initializers are called, so in fact the function pointer is valid and may be dereferenced. Note that the actual `main` function tests for a non-null `Application::Run` just in case the developer forgot to use the `WM4_*_APPLICATION` macro in his source file.

The `Run` function for a console application is

```
int ConsoleApplication::Run (int iQuantity, char** apcArgument)
{
    ConsoleApplication* pkTheApp = (ConsoleApplication*)TheApplication;
    return pkTheApp->Main(iQuantity,apcArgument);
}
```

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 iQuantity, char** apcArgument)
```

```
{
    WindowApplication* pkTheApp = (WindowApplication*)TheApplication;
    return pkTheApp->Main(iQuantity,apcArgument);
}
```

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 `WglApplication` project, there is a file `Wm4WglApplication.cpp` that implements `WindowApplication::Main`, as well as other functions that support the window (event handling, keyboard, mouse, and so on). You will notice that the WGL `WindowApplication::Main` function creates a window using the Win32 API, creates a WGL 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.

3.5 The Main::Terminate Function

The `Main::Terminate` function reverses the steps in `Main::Initialize`. Firstly, it queries the `Object` class to determine how many referenced-counted objects exist, storing the number in `Main::ms_iFinalObjects`. If the developer cleaned up the application properly (usually via Application::Terminate), then the values of `ms_iStartObjects` and `ms_iFinalObjects` should match; the termination code asserts that this is true.

Secondly, `Main::Terminate` calls all the termination functions. Once called, there is no need to keep the array of function pointers in memory, so the array is deleted. If any referenced-counted objects were created during the initializer calls, they should have been destroyed during the terminator calls.

Thirdly, the catalogs are destroyed. Because no reference counting is used, the catalogs are simply deleted.

Finally, a check is made that the number of reference-counted objects in memory is zero, which is the desired behavior since on entry to `Main::Initialize`, there were no reference-counted objects in memory.

The `Object::InUse` hash table is no longer needed, so it too is deleted. The idea is to deallocate as soon as possible everything a Wild Magic application has allocated in order to help with memory leak tracking.

If there is a mismatch in starting and final reference-counted objects, a text file called `AppLog.txt` is written to disk. (I need to arrange for this to happen only during development, just in case there is no disk storage.) Each `Object` has a unique ID (member `m_uiID`) when it is created. This ID is written to the log file, along with the class names of the objects. You can set conditional breakpoints in the `Object` constructor, triggered by the unique ID being one for an object that has "leaked".

3.6 The System::Terminate Function

After `Application::Run` terminates, all directories were removed from the system path. The array of directory strings itself is deleted by the call to `System::Terminate`. As mentioned in the previous subsection, Wild Magic was designed so that all allocated memory has been deallocated after the call to `System::Terminate`.

If you build the configurations for `Debug Memory` or `Release Memory` (i.e. you use the memory manager and tracking), a report is generated about any remaining memory that has not been deallocated. If there are remaining allocations in a well structured Wild Magic application, those allocations can be associated only with run-time or third-party libraries.

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 Wild Magic Application

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

```
// Pre-main execution occurs firt as always.  Then 'main' is called...
int main (int quantity, char* arguments)
{
    // Initialize directory-path handling.  You do not have to use the
    // WM4_PATH environment variable.  If you do, you should
    //  'assert(System::WM4_PATH[0])' after the Initialize() call.
    System::Initialize();

    // You can add any paths you like.  If you want to use the
    // WildMagic4 folder as the root for data diretories, use the
    // code that is in my 'main'.
    <Call System::InsertDirectory(...) for all desired directories>;

    // Initialize bookkeeping for reference counts, create catalogs, call
    // initializer functions.
    Main::Initialize();

    // *** BEGIN PLATFORM-SPECIFIC WINDOW/RENDERER CREATION ***
    //
    <Setup for creating a Win32 window goes here>;
    int windowWidth = *;
    int windowHeight = *;
    HWND hWnd = CreateWindow(...);

    // Create a Windows OpenGL (WGL) renderer.
    WglRenderer* renderer = new WglRenderer
    (
```

```
    hWnd,
    FrameBuffer::FT_FORMAT_RGBA,
    FrameBuffer::DT_DEPTH_24,
    FrameBuffer::ST_STENCIL_8,
    FrameBuffer::BT_BUFFERED_DOUBLE,
    FrameBuffer::MT_SAMPLING_NONE,
    windowWidth,
    windowHeight,
    0  // indicates window/renderer created once only (no multisampling)
);

// This work occurs in WindowApplication::OnInitialize.  You do not have
// to set a clear color (in fact, you can arrange never to call clear as
// long as your 'world' fills the window).  You do have to set the
// Program static data members, because they are used to load and parse
// shader programs.
Program::RendererType = renderer->GetExtension();  // "ogl"
Program::CommentChar = renderer->GetCommentCharacter();  // '#'
//
// *** 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 = new Camera;
renderer->SetCamera(camera);

// This work occurs in the DerivedApplication::OnInitialize.
camera->SetFrustum(...);
camera->SetFrame(...);
<create scene graphs and anything else you need>;
<call someScene->UpdateGS() as needed>;
<call someScene->UpdateRS() 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);
    delete camera;
    //
    // *** END APPLICATION-SPECIFIC LOGIC ***

    // *** BEGIN PLATFORM-SPECIFIC WINDOW/RENDERER DESTRUCTION ***
    //
    // This work occurs in WindowApplication::OnTerminate.
    Program::RendererType = "";
    Program::CommentChar = 0;
    delete renderer;

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

    // Terminate bookkeeping for reference counts, destroy catalogs, call
    // terminator functions.
    Main::Terminate();

    // Terminate directory-path handling.
    System::RemoveAllDirectories();
    System::Terminate();
}
```