Game Architecture

Engine Support Systems

Every game engine requires some low-level support systems that manage mundane tasks, such as starting up and shutting down the game, configure game features, managing memory usage, access to the file system, providing access to the different types of assets the game may use.

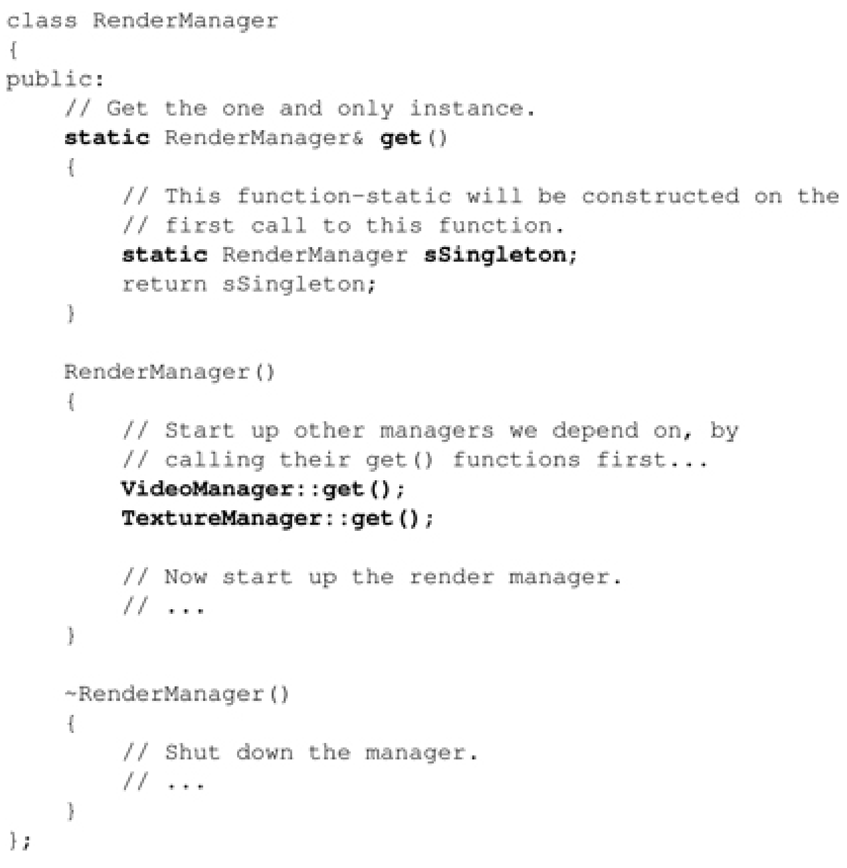
Subsystem Startup and Shutdown

In C++, global and static objects are constructed before the program's entry point (`main()`). However, these constructors are called in a totally unpredictable order. The destructors of global and static class instances are called after `main()` returns, and once again they are called in an unpredictable order. This method is not desireable for initializing and shutting down the subsystems of a game engine.

This is unfortunate because a common pattern for implementing major subsystems, such as the ones that make up the game engine, are defined as singleton classes (often called a manager).

Construction on Demand

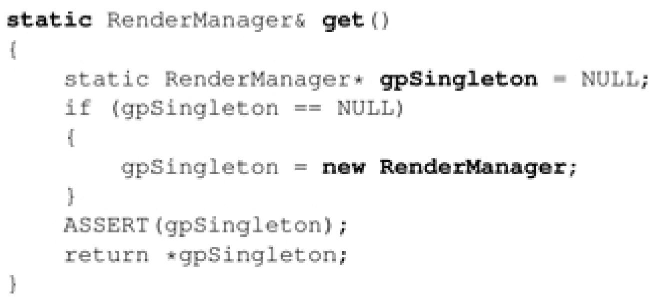
There is a little trick that can be employed to time the creation of static variables. A static variable that is declared within a function will not be constructed before `main()`, but rather on the first invocation of that function. So if our global singleton is function-static, we can control the order of construction of our global singletons as seen in Figure 1.



```
class RenderManager
{
public:
    // Get the one and only instance.
    static RenderManager& get()
    {
        // This function-static will be constructed on the
        // first call to this function.
        static RenderManager sSingleton;
        return sSingleton;
    }

    RenderManager()
    {
        // Start up other managers we depend on, by
        // calling their get() functions first...
        VideoManager::get();
        TextureManager::get();

        // Now start up the render manager.
        // ...
    }

    ~RenderManager()
    {
        // Shut down the manager.
        // ...
    }
};
```

Figure 1: Controlled singleton creation in C++

Many software engineering textbooks suggest using the following to create the singleton utilizing the heap as shown in Figure 2.



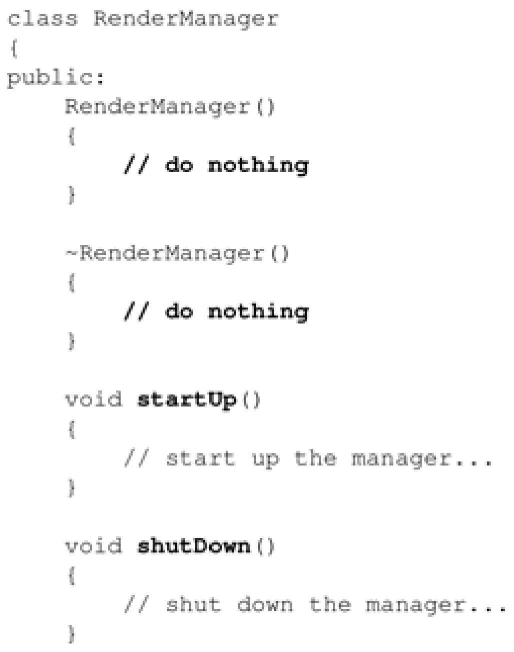
```
static RenderManager& get()
{
    static RenderManager* gpSingleton = NULL;
    if (gpSingleton == NULL)
    {
        gpSingleton = new RenderManager;
    }
    ASSERT(gpSingleton);
    return *gpSingleton;
}
```

Figure 2: Modified controlled singleton creation

This still does not give us a way to control the destruction order though. That is a problem because what if a manager that is dependent on another manager is released back to memory before itself. For example `RenderManager` is dependent on `TextureManager`, if `TextureManager` is released first, `RenderManager` may try to call `TextureManager` which would result in a crash.

A Simple Approach That Works

The simplest to control destruction of these singletons is to ignore the constructors and destructors and create our own that will be manually called as shown in Figure 3.



```
class RenderManager
{
public:
    RenderManager()
    {
        // do nothing
    }

    ~RenderManager()
    {
        // do nothing
    }

    void startUp()
    {
        // start up the manager...
    }

    void shutDown()
    {
        // shut down the manager...
    }
```

Figure 3: Brute force method of creating and destroying singletons in C++

A full example is shown below in Figure 4.

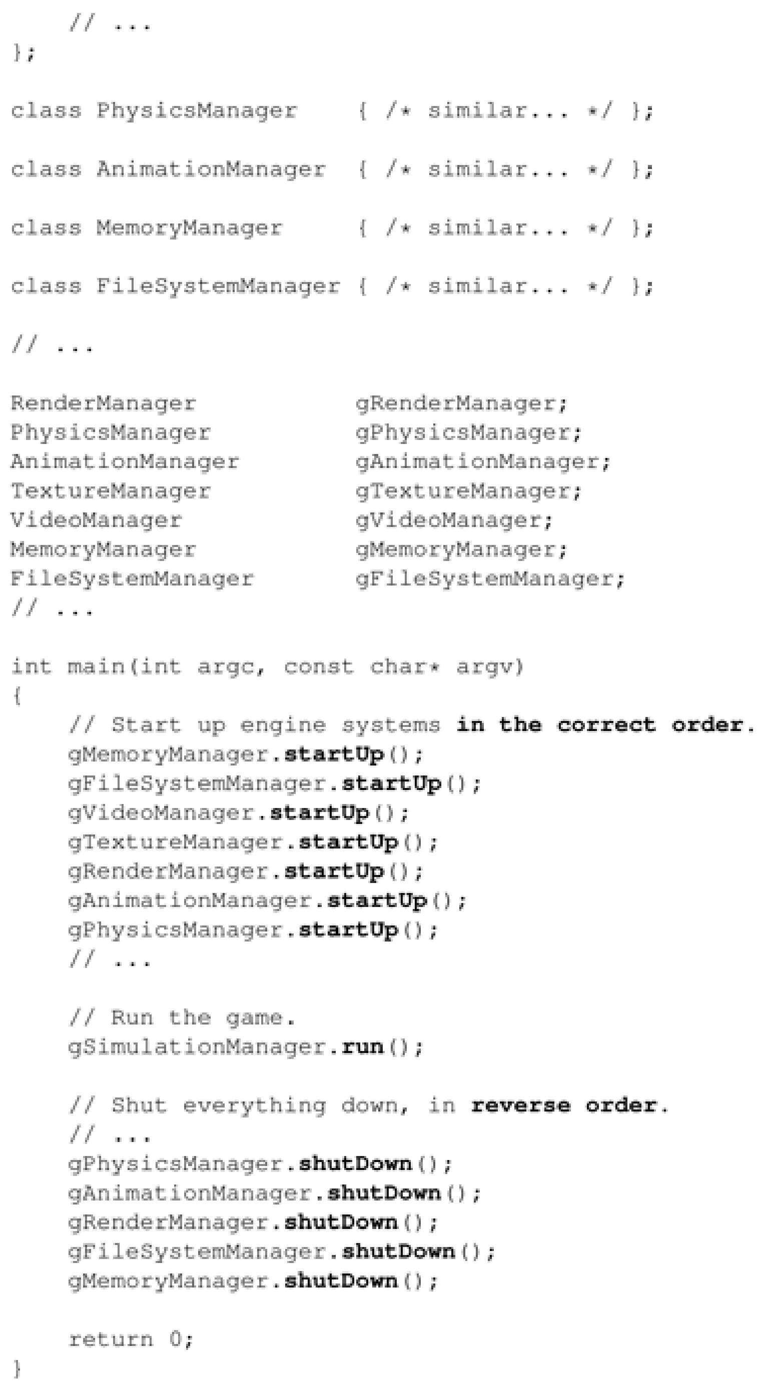
OGRE - An Example

Everything in OGRE is controlled by a singleton object `Ogre::Root`. It contains pointers to every other subsystem in OGRE and manages their creation and destruction. This makes it very easy for a programmer to start up OGRE - just a new instance of `Ogre::Root` and you're done. Examples are shown in Figures 5, 6, **??**, and **??**.

OGRE provides a template `Ogre::Singleton` base class from which all its singleton (manager) classes derive.

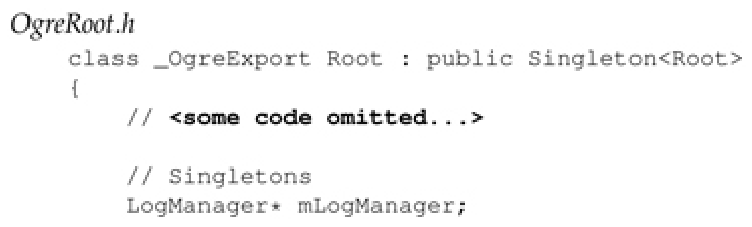
Memory Management

Memory affects performance in two ways: 1. *Dynamic memory allocation* via `malloc()` or C++'s global operator `new` is a very sow operation. We can improve the performance of our code by either avoiding dynamic allocation altogether or by making use of custom memory allocators that greatly reduce allocation cost. 2. On modern CPUs, the performance of software is often dominated by its *memory access patterns*. Data that is located in small, contiguous blocks of memory can be operated on much more efficiently by the CPU than if the same data were to be spread out accross a wide range of memory addresses.



```
    // ...
};

class PhysicsManager    { /* similar... */ };

class AnimationManager  { /* similar... */ };

class MemoryManager     { /* similar... */ };

class FileSystemManager { /* similar... */ };

// ...

RenderManager           gRenderManager;
PhysicsManager          gPhysicsManager;
AnimationManager        gAnimationManager;
TextureManager          gTextureManager;
VideoManager            gVideoManager;
MemoryManager           gMemoryManager;
FileSystemManager       gFileSystemManager;
// ...

int main(int argc, const char* argv)
{
    // Start up engine systems in the correct order.
    gMemoryManager.startUp();
    gFileSystemManager.startUp();
    gVideoManager.startUp();
    gTextureManager.startUp();
    gRenderManager.startUp();
    gAnimationManager.startUp();
    gPhysicsManager.startUp();
    // ...

    // Run the game.
    gSimulationManager.run();

    // Shut everything down, in reverse order.
    // ...
    gPhysicsManager.shutDown();
    gAnimationManager.shutDown();
    gRenderManager.shutDown();
    gFileSystemManager.shutDown();
    gMemoryManager.shutDown();

    return 0;
}
```

Figure 4: A full example of using the brute method to start and stop singletons



OgreRoot.h

```
    class _OgreExport Root : public Singleton<Root>
    {
        // <some code omitted...>

        // Singletons
        LogManager* mLogManager;
```

Figure 5: OgreRoot.h class declaration

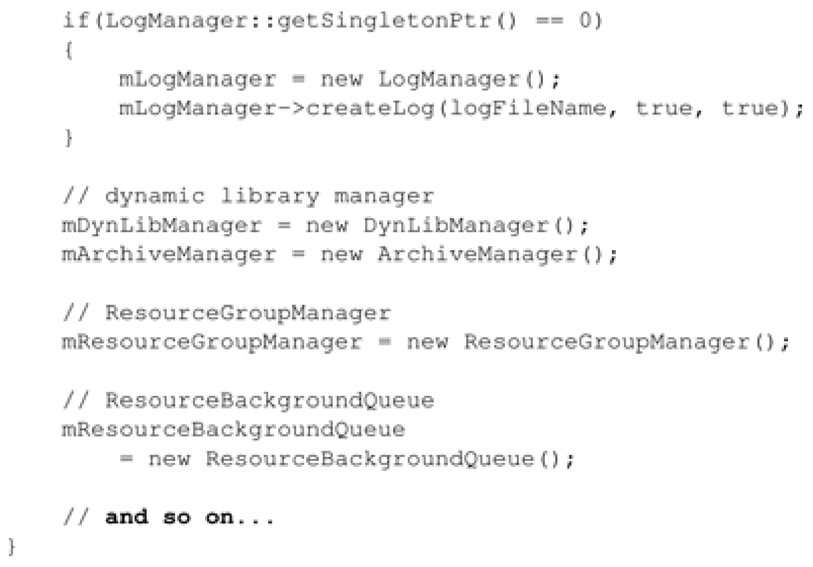
```
    ControllerManager* mControllerManager;
    SceneManagerEnumerator* mSceneManagerEnum;
    SceneManager* mCurrentSceneManager;
    DynLibManager* mDynLibManager;
    ArchiveManager* mArchiveManager;
    MaterialManager* mMaterialManager;
    MeshManager* mMeshManager;
    ParticleSystemManager* mParticleManager;
    SkeletonManager* mSkeletonManager;
    OverlayElementFactory* mPanelFactory;
    OverlayElementFactory* mBorderPanelFactory;
    OverlayElementFactory* mTextAreaFactory;
    OverlayManager* mOverlayManager;
    FontManager* mFontManager;
    ArchiveFactory *mZipArchiveFactory;
    ArchiveFactory *mFileSystemArchiveFactory;
    ResourceGroupManager* mResourceGroupManager;
    ResourceBackgroundQueue* mResourceBackgroundQueue;
    ShadowTextureManager* mShadowTextureManager;

    // etc.
};
```

Figure 6: OgreRoot.h member variables

OgreRoot.cpp

```
    Root::Root(const String& pluginFileName,
               const String& configFileName,
               const String& logFileName) :
        mLogManager(0),
        mCurrentFrame(0),
        mFrameSmoothingTime(0.0f),
        mNextMovableObjectTypeFlag(1),
        mIsInitialised(false)
    {
        // superclass will do singleton checking
        String msg;

        // Init
        mActiveRenderer = 0;
        mVersion
            = StringConverter::toString(OGRE_VERSION_MAJOR)
            + "."
            + StringConverter::toString(OGRE_VERSION_MINOR)
            + "."
            + StringConverter::toString(OGRE_VERSION_PATCH)
            + OGRE_VERSION_SUFFIX + " "
            + "(" + OGRE_VERSION_NAME + ")";
        mConfigFileName = configFileName;

        // create log manager and default log file if there
        // is no log manager yet
```

Figure 7: OgreRoot.cpp Constructor



```
        if(LogManager::getSingletonPtr() == 0)
        {
            mLogManager = new LogManager();
            mLogManager->createLog(logFileName, true, true);
        }

        // dynamic library manager
        mDynLibManager = new DynLibManager();
        mArchiveManager = new ArchiveManager();

        // ResourceGroupManager
        mResourceGroupManager = new ResourceGroupManager();

        // ResourceBackgroundQueue
        mResourceBackgroundQueue
            = new ResourceBackgroundQueue();

        // and so on...
    }
```

Figure 8: OgreRoot.cpp Methods

Optimizing Dynamic Memory Allocation

The dynamic memory allocation via `malloc()` or `new` and `delete` operators is typically very slow. The high cost can be attributed to two main factors. First, a heap allocator is a general purpose facility, so it must be written to handle any allocation size, from one byte to one gigabyte. This requires a lot of management overhead. Second, on most operating systems a call to `malloc()` or `free()` must first context-switch from user mode into kernel mode, process the request and then context-switch back to the program. These context switches can be very expensive. One rule of thumb in game development is:

> Keep heap allocations to a minimum, and never allocate from the heap within a tight loop.

Stack Based Allocators Many games allocate memory in a stack-like fashion. When a new game level is loaded, memory is allocated for it. Once the level has been loaded, little or no dynamic memory allocation is taken place. At the end of the level, the data is unloaded and all of its memory can be freed. It makes a lot of sense to use a stack-like data structure for these kinds of memory allocations.

A *stack allocator* (Figure 9 is easy to implement. We allocate a large block of memory using `malloc()` or a global `new`, or by declaring a global array of bytes. A pointer to the top of the stack is maintained, and every time you make a request for memory, a pointer is moved to keep track of the top. Everything below that pointer is considered to be used.

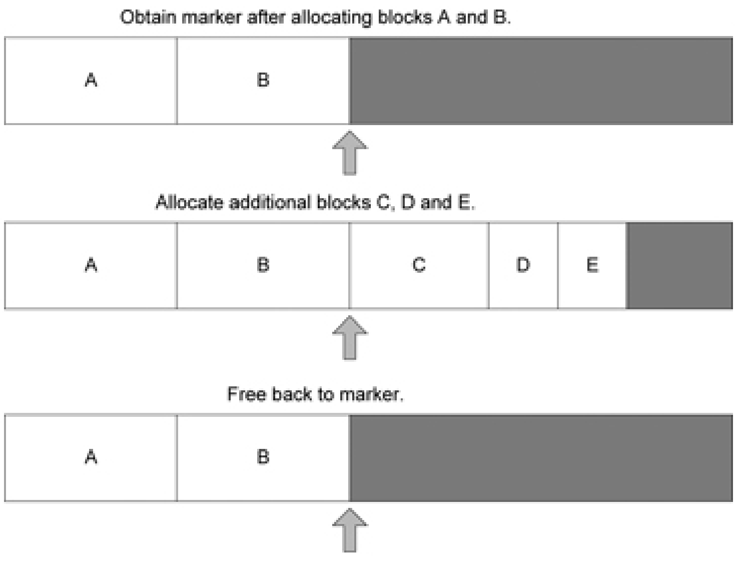

Obtain marker after allocating blocks A and B.
A
B
Allocate additional blocks C, D and E.
A
B
C
D
E
Free back to marker.
A
B

Figure 9: A visual representation of a stack allocator

It is important when using this kind of memory allocation, you cannot delete items in any order. The memory must be freed in the opposite order that it came in (last in first out, like a stack).

Figure shows an outline of what a stack allocator class should contain. The private section is left empty

!(A class declaration for a stack allocator class)[https://learning.oreilly.com/library/view/game-engine-architecture/9781466560017/ima