General Game Programming With Mercury

General Concepts and Notes:

Game programming is not like normal linear programming. Everything is done based on game cycles. Each portion of each screen or manager is designed to run for a brief period of time, do what it's supposed to do and return.

There are two primary portions to any Mercury game:
1: Managers
2: Screens/Actors

The only changes that you need to make to main() are adding your managers. Managers are portions of the game that are instantiated independently from individual screens. They are used often throughout the execution of the program.

Screens are classes that can be dynamically loaded or unloaded at any point during the program's execution. They are what actually control the graphics.

Screens in Mercury are self-registering. This means if you so much as link in a screen, Mercury is aware of its existence. Screens also abstract from MercuryObject so any functions you can perform on objects like tweening, moving, etc. you can also perform on a screen.

Let's examine a sample "hello world" program:

ScreenHelloWorld.h

```
#ifndef _SCREEN_TITLE_H
#define _SCREEN_TITLE_H

#include "MercuryScreen.h"

class ScreenHelloWorld : public MercuryScreen
{
public:
	ScreenHelloWorld( ) { }
	ScreenHelloWorld( const CString & name ):
			MercuryScreen( name ) {m_name = name;}
	virtual void Init();
private:
	MercurySprite m_sprHello;
};

#endif
//This code: Public Domain 2005 Charles Lohr
```

ScreenHelloWorld.cpp

```
1   #include "global.h"
2   #include "ScreenHelloWorld.h"
3
4   REGISTER_SCREEN_CLASS( ScreenHelloWorld )
5
6   void ScreenHelloWorld::Init()
7   {
8         m_sprHello.SetName( "HelloSprite" );
9         m_sprHello.LoadPNG( "HelloWorld.png" );
10        AddObject( &m_sprHello );
11        m_sprHello.Tweening.AddCommand("linear,2;rotationz,360");
12        MercuryScreen::Init();
13  }
14  //This code: Public Domain 2005 Charles Lohr
```

Hopefully this .h looks similar to other classes you have worked on in C++. It has a header file containing the class definition and a C++ file containing the definition of your class "ScreenHelloWorld." Screens are forced to contain two constructors even if there is no code used in them. This is to allow multiple and different instantiations of the same screens (much more advanced and will not be covered in this tutorial). You can copy and paste these two constructors.

ScreenHelloWorld abstracts from MercuryScreen which contains all of the necessary functionality for a stand-alone screen class.

It only contains one abstracted function here (Init, line 12). This is what gets called once the Screen Manager has loaded your screen and is ready to let it start operating. **You should minimize the code you put in your constructor in favor of putting code in Init.**

It is generally suggested you either put a copyright or public domain notice at the bottom of your code. (line 18).

Now, looking at the .cpp, it should look similar but with some differences from what you are used to. You are strongly encouraged to include "global.h" because it contains a number of useful compiler directives, pragma's and necessary code for the Mercury Engine. Sometimes your code will compile warning-free without it, sometimes it won't.

We then introduce a macro used by Mercury here "`REGISTER_SCREEN_CLASS.`" This macro does everything necessary to make the Screen Manager aware of the existence of your screen and gives it the ability to instantiate it with nothing more than the name of your class. This is useful for modular programming. It will allow you to have a lot more dynamic control over the loading and unloading of screens.

In the Init code for our screen, we do five things.

1. Name our sprite – This is very useful for debugging code when you become more advanced. It's also a mechanism to help find memory leaks. When

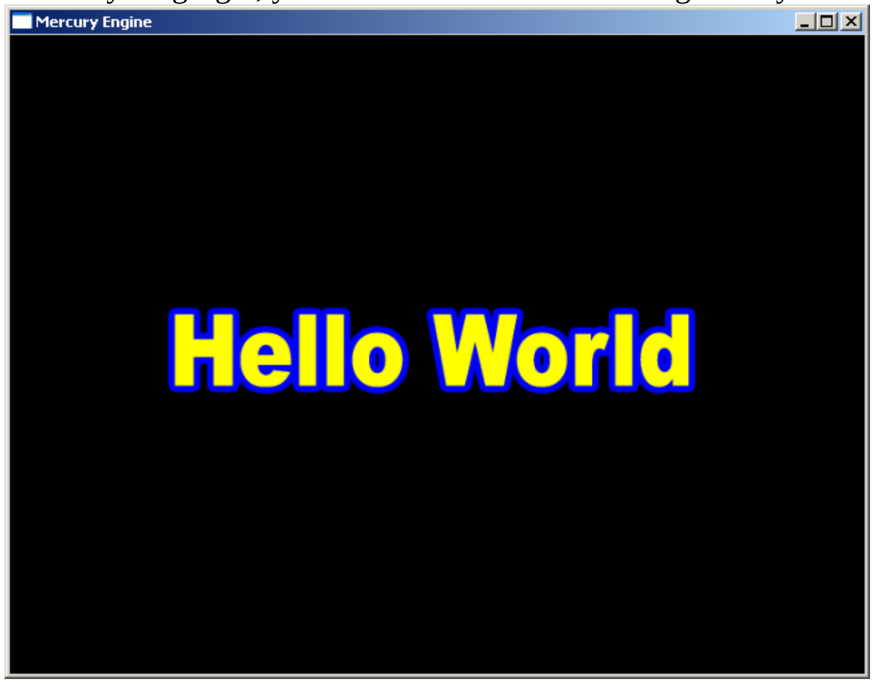
everything's named, if you find yourself having too many of a certain object, you know exactly where to look.

2. Load a PNG into our sprite – Currently, mercury supports only two graphic formats: BMP and PNG. Since it loads the given image as a texture, it must be square in shape with it's width and height a value of 2^n. IE Mercury only accepts textures of sizes: 1x1, 2x2, 4x4, 8x8, 16x16, 32x32, 64x64, etc…
3. Add Object to the screen – This essentially tells Mercury to handle drawing and updating of this object. It makes it unnecessary for us to have to worry about drawing in order, or making sure our object gets updated and makes it unnecessary for us to have an overloaded Draw() or Update() function.
4. Add tweening to the object – This gives the object a set of commands to follow. Linear is a tween type, it says to move from the current state to the defined state in a linear manner. It takes on one parameter, the amount of time to spend tweening. In this case it's 2 seconds. The command it to set the rotation Z to 360 degrees. Z is the axis that comes out at the camera, and therefore by rotation relative to the Z axis, the sprite will appear to spin, one full turn to end up being oriented correctly when it's done.
5. Tell MercuryScreen to Init – Hopefully you'll remember from your computer science classes that you need to call the base class's abstracted function manually.

You will also need to tell Mercury to put your screen on. You can simply put the following command in your main file (Mercury.cpp) after the Texture Manager gets instanciated:

```
	SCREENMAN->AddScreen("ScreenHelloWorld" );
```

If you've done everything right, you should receive the following when you run:


Mercury Engine
Hello World