The Mercury Game Engine

(Version 0.2 **Preliminary** Design Document)

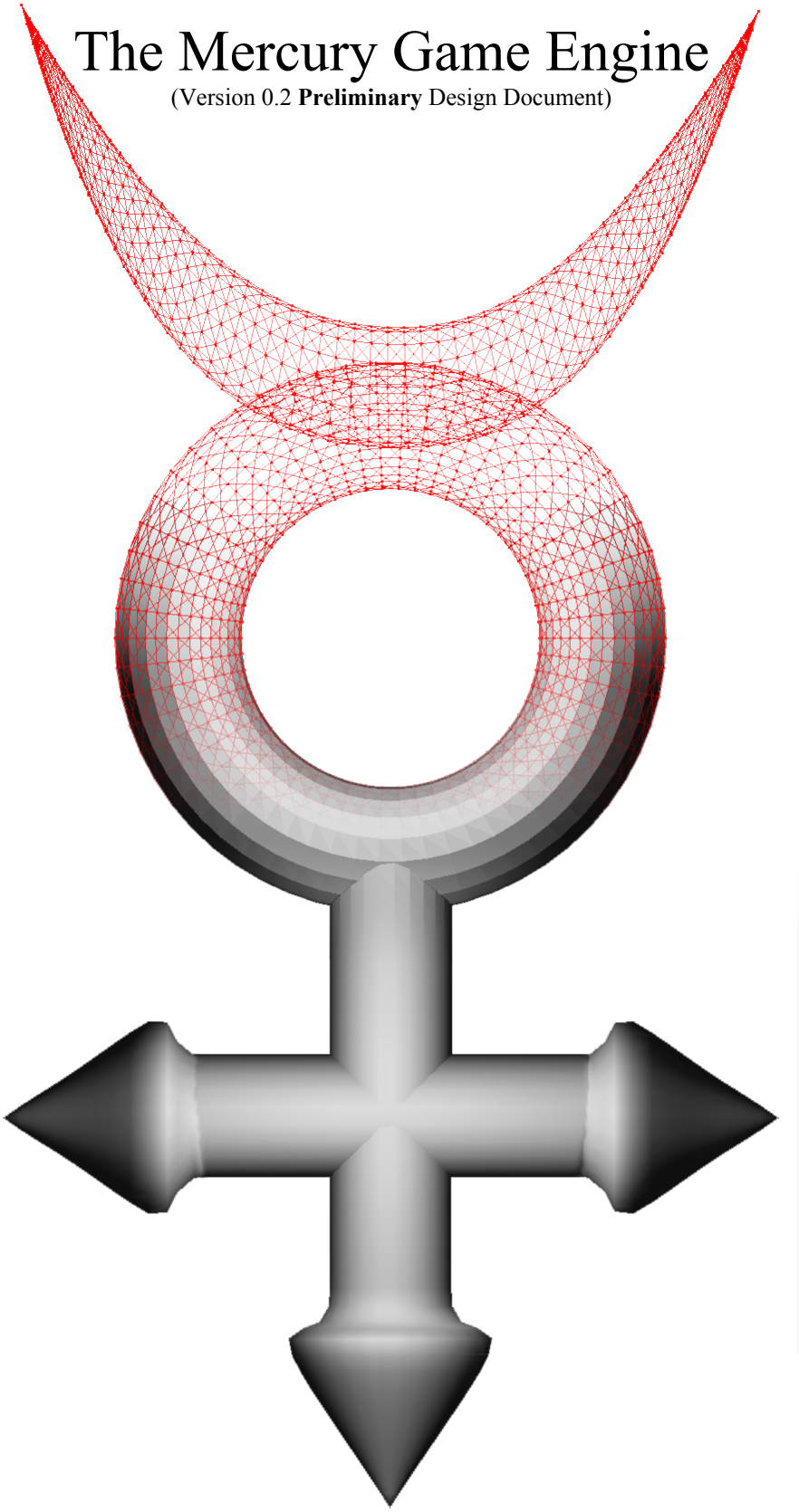

Table of Contents

General Information

The Mercury Game Engine or just Mercury for short is under the following license. Use of it is not limited to the following license depending on the intended use of the end product.

Parts of the code, namely the crash handler are derived from StepMania, which uses the VirtualDub system created by Avrey Lee. Those parts are Copyright Avrey Lee, Glenn Maynard, and Chris Danford. All content is either under the MIT-X11 license or public domain. Both licenses are completely compatible with the BSD license that Mercury is under.

Mercury makes use of the following libraries and if you choose to use their functionality in Mercury, you must comply with their terms and conditions.

OpenGL – BSD License
Open Dynamics Engine – BSD License
Zlib – Zlib License (mostly compliant with the FreeBSD License)
LibPNG – LibPNG License (mostly compliant with the FreeBSD License)
FreeType – The Freetype License or The GPL License.
OpenAL – The LGPL License
PlayStation 2 Support – The Free Academic License 2.0*

*The Free Academic License 2.0 Does NOT permit commercial use of its material.

In short, **you may** use Mercury in **closed source, commercial production without permission**. You may modify it how you see fit without any prior written consent. However, it would be kind to let us know of your project.

General Description

Mercury is a C++, Open Source, Multi-platform, Multi-threaded Game Engine that runs fully featured on Linux and Windows®. It is 64-bit x86 compatible and can operate on 32, 64 and 128-bit processors. Mercury compiles in GCC3.2 - 4.1 in Linux, dev-C++/minGW, Visual Studio® 6, .Net 2003, and .Net 2005 in Windows®.

Mercury isn't just a graphics library but a full programming environment. In general code designed to be run in Mercury needs not to make calls to the system or any functions that would be platform dependent. Despite this, Mercury should remain small and applications that utilize it should not have to suffer exaggerated load times for features they do not use. Mercury's registration system allows different features to be compiled in without modification of other code. If there is no need for physics, the ODE classes can simply not be linked and ODE will not be used.

Mercury is designed around the open source fundamentals. Most of what you will see supported in Mercury and used with Mercury is Open. We feel that you should be able to develop a great game without having to purchase proprietary tools and systems. We as the designers of Mercury believe that it is important that software move toward open source in general. We do not require, but strongly encourage derivative projects to become open source as well.

When a programmer uses Mercury they should not need to make use of system dependent external libraries such as Microsoft's .NET®. All programmers are strongly encouraged to make use entirely of functionality already included in Mercury. Programmers also do not need to write their own classes to do menials activities with data. Structures like matrices, vectors, hash tables, etc. are already written for the user and plug into Mercury.

Mercury has re-written some of the STL data types for added functionality as well as improved performance. Users should exclusively use MStrings, MVectors, and MDeques instead of their STL counterparts.

Mercury's primary goal is speed. Mercury sacrifices some of the benefits that are gained by clean coding when a noticeable performance increase can be found for the end user. You can see this in many places in Mercury where you can see duplicate code, extra use of inline functions, macros, and private data members being made public without accessors. Assembly code for common math functions is compiled in Windows for extra performance is an example of the paramount of speed in Mercury.

Code-wise Mercury is laid out similar to StepMania's Rage engine by Glenn Maynard and Chris Danford. More information about StepMania and its engine may be found at www.stepmania.com.

Current Features

Mercury has a variety of notable features and tools available to the programmer. They can be split up in to four basic groups: Code Features, Implemented Features, Built-In Features, and Tools.

Code Features:

- Highly portable and robust code; can run on:
  - Linux (most flavors)
  - Microsoft Windows® 98, ME, 2k, XP
  - ReactOS [1]
  - Sony PlayStation® 2 [2]
- Cross-platform support, can compile in:
  - Microsoft Visual Studio 6
  - Microsoft Visual Studio .Net 2003
  - Microsoft Visual Studio .Net 2005
  - Bloodshed Dev-C++
  - GCC 3.2 through 4.1
- Strong object-orientation, most entities in Mercury abstract from MercuryObject, most core functionality can be acquired through use of global singletons.
- Basic RTTI when manually used. This prevents overhead due to RTTI when unnecessary, however allows the user to identify objects when necessary.
- Object registration, Command and Message based system. Objects, singletons, external code, etc. can talk to objects in Mercury through a clean, unified system.
- Automatic Object Registration allows you to do nothing more than link in objects and spawn them by nothing more than a string that could be read from the theme or anywhere else.
- Useful existing framework for use with graphics applications, included but not limited to the following:
  - Matrix Class
  - Vector/Point Class
  - Quaternion Class
  - Color Class
  - Unified standard fast math functions, COS, SIN, ABS, etc...

1 Sound does not work
2 Sound does not work and Video is not complete

    - Matrix/Vector/Point Utility functions
- Self-contained file system. Allowing for common reading of files, assets and other data from any of the following transparent to its use:
    - Zip Files
    - Uncompressed package files
    - Memory Files
    - Compiled-in zip archive files
    - Ordinary files off of the hard drive
- Theme system to allow for easy multiple language/locale support stacking of skins and easy aggregation of game assets.
- Log system to provide a unified generalized output for logging purposes, so output can be seen both in console and logged to disk or whatever device is pertinent to the architecture.
- Memory Manager to allow for memory accounting. when in debug mode the memory manager can be turned on and any leaked ram will be shown along with the file and line number of the statement that leaked it.
- Loading of Models
- Loading of Images
- Loading of Text Boxes
- Particle engine
- Orthographic and Perspective view
- Built in data structures for extra performance and functionality:
    - MString (to replace the STL's string)
    - MVector (to replace the STL's vector)
    - MDeque (to replace the STL's Deque, Queue, and Stack)
    - MHash (string-based hash table)
- General Purpose Timer/Clock
- General Thread Class
- Utility functions for most system-dependent and extra-featured uses
    - SAFE_DELETE/FREE
    - Sleep
    - Basic String Processing
    - Assertion
- And a ton of examples!

Implemented Features:

- Open Dynamics Engine

  With ease, use the in-engine map editor to create and modify a scene to your liking, then simply describe the way each object should behave when physics is applied, select which objects can be interacted with and play!

- PNG and BMP support

  Transparently load BMP and PNG files. Support for extra file formats is easy to add. PNG and BMP are two of the most widely used lossless formats available and allow you to save your graphic asset in most programs for use in Mercury.

- Zlib support

  Use Zip files interchangeably with the native filesystem. This allows you to steamroll updates or general assets into a single zip for distribution and use. Heck, you can just compile the zip file into your end .exe file for a live demo at your next interview. No bulky installers, or messy unzipping multiple files to a desktop to let your potential employer see your work.

- OpenGL

  Use the most widespread open graphics library for the end display of your project and get the benefits of system acceleration and identical functionality on many different platforms. Also, Mercury supports GLSL on a per-material basis, allowing you to use pixel shading to it's full potential.

- DbgHelp

  If your program crashes on an end user, instead of getting an unfriendly error box with little to no indication as to what has gone wrong, Mercury can output a nice clean crash log with the reason for the crash and a call stack with the names of all the symbols of the path that lead to the crash.

- FreeType

  Use your .TTF fonts and see them in their full glory. FreeType allows you to blow your text up and keep all of your edges clean and smooth. When using FreeType fonts, a user can change their screen resolution and Mercury will automatically re-render all of the fonts displayed on the screen.

- OpenAL

  Make use of that player's 5.1 Surround system or that pair of headphones the user is wearing to it's full potential. Seamlessly you can attach sounds to objects and they will be presented to the user where they should be and fully using their hardware.

Built-In Features:

- Comprehensive Crash Handler

  The crash handler in Mercury operates in Windows and Linux, 32 and 64 bit. If the program performs an ASSERT, FAIL, or ASSERT_M, the reason for the crash will

be printed on the first line of the crash report. Otherwise, say if the program writes to address 0x00000004, or another dumb move, the crash handler will present the reason for the crash: "Segmentation Fault – Unmapped Address 0x00000004" as well as a stack showing all of the functions that were called leading up to the crash.

- Debug Memory Accounting System

  Before you should release any program using Mercury, you should turn on the _DEBUG_MEMORY flag in debug mode. This will cause mercury to record every instance of memory allocation as well as deallocation that uses new, delete, malloc, calloc, realloc and free. When the program execution is complete, a list of every allocation that was not deleted will be presented. Included in it is the absolute address of the memory, as well as the file name and line number the allocation command was on.

- Dynamic Shape Creation

  Create shapes without having to import them from a model. This lets you put placeholder artwork in your application without spending hardly any time at all.

- INI file support

  Make your program more dynamic with the INI file support that Mercury includes. You can make a new MercuryINI object, and load or save it to a file. Gain access to sections and keys through a variety of accessors.

- Mercury Model Format

  Mercury has its own binary model format. While this does not allow for manual modification of models, it does provide a significant reduction in loading time. Besides, who, realistically is going to edit a 20,000 polygon mesh in notepad? The Mercury model format supports bones, vertex-weights, quaternion-based animations, multiple meshes and materials.

- Software Renderer

  On systems where OpenGL does not exist or fails to operate properly there is a backup plan! The Renderer flag can be set in Mercury.ini to read SWC instead of OGL and get the user going. The software renderer is considerably faster than using Mesa on Linux systems, as it foregos many of the features that are not terribly important to using Mercury.

- Unified device-independent input system

  Use one common interface system for all systems. This means, without modification to your code you can be using the analog stick on your Sony DualShock® Controller or the mouse at someone's computer and you couldn't care less. The A button on a keyboard or the X button on a PS One controller, they're all the same interface method. Heck, you can even let the user remap the buttons however they see fit and there's no nasty input code you have to write for your own program.

- Plain-text command/tweening system

  Mercury allows you to compose the way objects should behave through plain text strings. It processes these at run time and for commonly used commands, i.e. the

motion of a particle on a firework it allows caching of commands, thereby meaning it processes the plain text once, and just runs the commands whenever necessiary. An example of an Object Command is: "x,0;linear,2;x,100;" which would cause a smooth motion of whatever object it is applied to to move from 0,0,0 to 100,0,0 over the course of two seconds.

- Copper UI System

  Mercury comes with a fairly easy-to-use and fully customizable UI interface called Copper.

Tools

- HGPCK archiver

  Archive your files in a HGPCK file instead of a zip file if initial loading time is an issue. Unlike .ZIP files, the entire file index is stored at the beginning of the file, which makes archive enumeration take a fraction of the time.

- MakeOBJ

  Take a .ZIP file and turn it into an OBJ file for use with Mercury. This allows you to compile content into your very EXE. You can get a product in the end that is a stand-alone exe file that contains all content and libraries necessary to run. This is ideal for resume, demos and Internet-distributable applications.

- ModelOptimizer

  Optimize your models for display. The model optimizer currently tears apart meshes whenever possible to minimize calculation and modification of vertex data. It also removes "invisible" bone weights and animation effects.

- MS3D Export Plugin

  Use Chumbalum Soft's MilkShape 3D® to import your models from virtually any format edit your models and then export them to Mercury's native binary model format keeping all bones, meshes, materials and animations in tact.

- OgreXML to HGMDL

  Take in any format that the OgreXML converter can read such as Maya or 3D Studio Max and import it with all of its features directly in the Mercury's native binary model format.

- Doxygen-ready config file

  Use the copy of Doxygen in CVS and instantly produce graphs and documentation for the whole of Mercury including whatever new content you have added.

Goals

Among other things, Mercury aims to become a more complete game engine by adding support in the following areas. We hope to encompass as much as possible to allow users to do whatever they want without having to worry about importing extra DLLs and code they don't need. We hope to provide the following in the coming year:

- Networking Support

- Better Pixelshading support
- DirectX Support
- Better PlayStation 2 Support
- Mac OSX Support
- LUA Implementation
- Self-Compiled Scripting Language
- A high-performance red-black tree to replace the STL's Map.
- More Texture format support
- More Importing tools

Coding Standards

Mercury uses coding standards similar to those of StepMania. We believe in the following coding standards: Tabs, not spaces; spaces between parenthesis; m_ prefix to class member functions; comments of any style are acceptable. Magic numbers should not be used.

Examples:

```
///Demonstration Class
class A : public class B
{
public:
	A() : m_i(0) { }

	inline bool IsFive() { if( m_i == 5 ) return true; else
		return false; }

	///Returns if the value is six.
	inline bool IsSix() { return (m_i == 6); }

	///Perform a loop
	void DoLoop();

	int m_i;
};

void A::DoLoop()
{
	for( int j = 0; j < m_i; ++j )
		LOG->Info( ssprintf( "%d\n", j ) );
}
```

Notice that comments relating to something that should be documented in doxygen have three forward slashes. This is how doxygen looks for class information. Also pay attention to the parenthesis use as well as the curly brace use. Inline functions have their curly brace and code on the same line, if the line gets too long, it can move to the next line. When in CPPs, function definitions should be on a line by themselves.

Usage-wise we are a little funny. We follow these general guidelines: Minimize use of the STD, meaning <iostream>, <vector>, <deque>, <sstream> should not be used. Instead of using streams people should use ssprintf(), instead of vectors, use MVector; instead of deques, stacks and queues, use MDeque; instead of printing out to the console, use LOG. Never use commands like MessageBox, or other windows-dependent commands, unless writing a driver.

Minimize #includes in your .h files, keep them for your .cpp files. This decreases compile time and makes it easier to understand what is being included in what order. There can be serious issues if something like windows.h is included in the .h of one of your device drivers. For example: windows.h defines LoadImage as LoadImageA. This means LoadImage will be mislabeled in parts of mercury that have the LoadImage function.

Developer and Resource Information

NOTE: ALL INFORMATION IN THIS SECTION IS SUBJECT TO CHANGE

Mercury's current home is on SourceForge.net. The current URL is: http://hgengine.sf.net. Mercury is in a SourceForge CVS repository. In order to get a copy of Mercury, you can get it via the instructions found at http://sourceforge.net/cvs/?group_id=140522. The module for Mercury is "Mercury." If you want to browse our CVS you can look at it here http://hgengine.cvs.sourceforge.net/hgengine/Mercury/. If you would like the link to the SourceForge page, it is here http://sf.net/projects/hgengine.

Windows Users:
Once you have Mercury checked out, you will notice a series of Visual Studio and Bloodshed Dev-C++ project files in the src folder. You should select the build to "Release" as the other builds may not work. Compile and run your application in the root Mercury/ folder, not in the Release/ folder or whever it compiles it. Running it elsewhere may result in an inability to find appropriate DLLs. If it finds all of the DLLs, mercury will start but will show nothing other than a black screen.

Linux Users:
Once you have Mercury checked out, you will have to run the following commands:

```
sh autogen.sh
./configure
make
```

Then, you will have a file called "mercury" in your src folder. Run it from your main Mercury/ folder. The actual executable is located in src/.libs/. Use on Linux requires the following:

- LibSDL
- ODE
- OpenAL
- Xorg/X11
- Zlib
- LibPNG

Major Entities

Mercury has a series of major entities or objects that people generally interface with. Since the functions and data types they contain change often and for the most part, they will not be described here. For information regarding them, you may click on the [Doxy] link for each. Note that this is not a complete list. It only contains global objects that make sense for the end-user to interface with.

DISPLAY

[Doxy] Mercury has an interface to the display driver that can be accessed via the DISPLAY singleton. In general, until you get into more advanced programming, most access to the DISPLAY singleton is unnecessary as most of what you want to do can be done simply by using separate objects. It is a very powerful tool that can be used when writing your own custom render cycle for an object.

FILEMAN

[Doxy] The Mercury File Manager "FILEMAN" is how you should access all files. You should avoid using regular I/O functions i.e. fopen, fstream, etc. since using them will not allow your program to function using the virtual file systems. FILEMAN gives you a one-stop way to get to any MercuryFile objects you may need to use or a listing of a folder. The file manager also understands prefixes of "FILE:" "MODEL:"and "GRAPHIC:" where it will use the theme system to find your file.

PREFSMAN

[Doxy] PREFSMAN is a global handle to Mercury.ini that allows you to obtain general preferences for the whole of the application. Most information core to Mercury is stored here, such as the renderer, list of input devices, logging information, key mappings, sound drivers, theme list and graphics output information. You may store core information relating to your program here. Information such as the last mode played, or the last user name of the player or mouse sensitivity would make sense to go in this file.

INPUTMAN

[Doxy] The Input Manager allows objects to receive information regarding peripheral input. A user can both inquire as to the current state of some device, i.e. A pointing device or if a key is pressed. A user can also subscribe to be informed about events such as a button being pressed or released via the message manager.

LOG

[Doxy] The log allows programmers to output general information, trace information and warnings in a meaningful manner. If you are running in debug mode, anything logged will also be sent to the console for easy debugging.

MESSAGEMAN

[Doxy] The Message Manager allows singletons, static code and objects to talk to all other objects as well as call commands. This system allows any entity to subscribe to messages and whenever one of those messages is raised, code within that entity will be called. Examples of messages could be “mappedinput” or “quit.” The message manager can both broadcast messages for immediate dispatchment, as well as post messages for dispatch upon the next program cycle. Messages can be posted from any thread, allowing for inter-thread coordination.

THEME

[Doxy] The theme manager provides access to a polymorphic, multiple-inheritance theme system. This allows the programmer to have their code get game content (graphics, models, files) even as much as information such as an item's HP or a person's hair color from a Theme. The theme allows users to define multiple fall backs. IE If you have a special character in your story, it may want to inherit traits such as HP or behavior from a more general character class. Any of the values or graphics gotten from one of the theme values can be overridden by another theme. You can load a second theme, say a German theme on top of the default theme. You can specify the order in which to overlay themes. IE: Christmas => German => Default.

OBJECTREGISTER

[Doxy] The object register keeps a table of all of the currently created objects in Mercury. If you are using scripting or need to acquire control of an object you don't have the pointer to, you can give its name to the object register, and it will return a pointer to that object. You can then even inquire as to the object's type using GetType().

BENCHMARK

[Doxy] One of the last user-functional singletons in Mercury is the benchmark. You can put BENCHMARK.Begin() and End() calls around any code you want to analyze the performance of. When the program is done execution, it will display a list of all of the run benchmarks, the number of times the benchmark was started and stopped, and the average amount of time that was spent inside of one of the calls. This is a alternative to code profiling. A user may put a benchmark statement inside of a conditional statement so that it will only get run when a condition is met, unlike profiling.

Utility

Mercury has a variety of utility functions to increase programmer productivity. These utilities can be split into four basic groups below:

Global Namespace General Utility

The global namespace has a series of utility functions. Most are included in global.h and MercuryUtil.h. They include:

- SAFE_DELETE(x) – Delete X if X is nonzero, and set it to zero.

- SAFE_DELETE_ARRAY(x) – Same as previous, but for arrays
- SAFE_FREE(x) – Same as previous but does not call destructor
- MVPtr – Integral type for a void * (changes based on word width)
- nextPow2(x) – Get the value of the next power of two higher than x, IE 5 will return 8, 8 will return 16.
- makePow2(x) – Get the value of the next power of two equal to or higher than x, IE 5 will be 8 and 8 will return 8
- long DumpFromFile(const MString & filename, char * & data) - Dump the data in file filename to data. Data will be allocated appropriately, you must delete it. It will return a value less than zero if it fails, otherwise it will return the number of bytes read.
- bool DumpToFile(const MString & filename, const char * data, long bytes) - Dump the string in data with length bytes to the file with name filename. Returns true if it was successful, false if not.
- void FileToMString(const MString & filename, MString & data) - Counterpart to DumpFromFile, except uses an MString.
- void MStringToFile(const MString & filename, const MString & data) - Counterpart to DumpToFile, except uses an Mstring.
- bool FileExists(const MString & sFileName) – Return true if the file by the given name exists, else return false.
- long BytesUntil(const char* strin, const char * termin, long start, long slen, long termlen) - Returns the number of bytes until the character in strin is a character in the list of tokens in termin.
- long BytesNUntil(const char* strin, const char * termin, long start, long slen, long termlen) - Returns the number of bytes until the character in strin is a *not* character in the list of tokens in termin.
- void SplitStrings(const MString & in, MVector < MString > & out, const char * termin, const char * whitespace, long termlen, long wslen) - Split in into separate strings in out using termin and whitespace as guidelines as to how to split the string.
- T Clamp(in, min, max) - Clamp the value of in between min and max. This could be used in a situation where a player must be forcibly bound inside of some parameter.
- Swap32(n) - Return n byte-swapped (useful for changing endians)
- Swap16(n) - Identical to Swap32 except for shorts, not longs.
- ASSERT_M(condition, message) - If the condition assertion fails, present message as the reason for failure.
- ASSERT(condition) - If the assertion fails, notify the user in a crash.
- FAIL(message) - Cause a program fail, and present the user with the message for failure.
- REGISTER_OBJECT_TYPE(x) - Register the given object type in the type register so that objects of that type can be spawned.
- REGISTER_SCREEN_CLASS(x) - Register the given screen type so that the screen can be set based on values in metrics.ini.
- REGISTER_ODE_OBJECT_CLASS(x) - Register the given ODEObjectLoadable type as an ODE class so it can be included with different maps.

- CLASS_RTTI(CLASSNAME, PARENTCLASS) - Give your class RTTI. Put this in your class definition.
- REGISTER_STATEMENT_TO_COMMAND(command, code) - Register the statement in "code" to be a command, it can be any old expression.
- REGISTER_FUNCTION_TO_COMMAND(command, function) - Register the function in function to be a command, it can be any function of the format CMDCallback.
- KeyMappingWithCode(x, y) - register a key mapping of name x to the key input system. If there is not already a key by that name in Mercury.ini, it will add it with the default initial binding of y. y is in the format of [device]-[key] IE 0-65 is the A button on a keyboard.

Global Namespace Math Utility

There are a series of math utility defines and functions in the global namespace for mathematical use. They are as follows:

- DEGRAD – Multiply degrees by this value to get radians.
- RADDEG – Multiply radians by this value to get degrees.
- Q_PI – Pi (floating point)
- SIN(x), COS(x), ATAN2(x, y), ASIN(x), ACOS(x), SQRT(x), TAN(x), ABS(x), SQ(x) - Perform the given operation on x except do it in the best way for the given architecture. For example: on Windows, typecasting sin to float is faster than the sinf function. If you want to do math with a function that exists in this list, use its instance in this list!
- DotProduct(x , y) - Perform the dot product of x onto y.
- MercuryPoint Rotate2DPoint(float fAngle, MercuryPoint pIn) - Rotate the given point pIn around the Z axis.
- void VectorIRotate (const MercuryPoint & in1, MercuryMatrix &in2, MercuryPoint & out) – Transform a vector in1 by the reverse of a matrix in2, but only in the linear region (treat the matrix as a 3x3).
- void VectorRotate (const MercuryPoint & in1, const MercuryMatrix &in2, MercuryPoint & out) – Same as VectorIRotate except forward, not reverse.
- void VectorMultiply(MercuryMatrix &m, const MercuryPoint &p, MercuryPoint &out) - Multiply a given point p by a matrix m.
- void InvertMatrix(MercuryMatrix &in, MercuryMatrix & out) - Perform a matrix inversion on in. Note that this is VERY slow.
- MQuaternion SLERP(const MQuaternion &a, const MQuaternion &b, float t) - Spherically interpolate two quaternions a and by by an amount t.

Math Types Utility

Mercury has types for Quaternions[Doxy], Vectors/Points[Doxy], Matrices[Doxy] and Colors[Doxy] along with public member functions to perform all necessary math functions on them.

Data Types Utility

Mercury has rewritten certain base types in order to have increased performance functionality and standardization within mercury over the STL. They are as follows:

- MString [Doxy]
  Mercury's string class is a class whose definition is strongly based upon StdString, a class by Joe O'Leary. It's designed to be VERY fast though. In general it can be used interchangeably with the STL's string.
- MVector [Doxy]
  Mercury's vector class is a work-alike to the STL's vector class, however it does not require iterators to perform basic insertion and removal from it, instead it uses integers for all accessing.
- MDeque [Doxy]
  Mercury's deque is a work-alike to the STL's, however it does not have an iterator internal to it. One should use MDequeIterator for an iterator. It is a drop in replacement for queue, deque and stack.
- MHash [Doxy]
  String-indexed hash table. This has no STL counterpart. It exists for the purpose of providing extremely high speed access to string-indexed data types in situations where there will not be a ridiculously large number of elements.

Class Layout

All things that appear on the screen in Mercury must abstract from MercuryObject. In fact, virtually everything that is a real entity in Mercury abstracts from MercuryObject. This includes screens, text and pretty much everything else. You should abstract from it too. You should visit the Doxygen information for MercuryObject [Doxy] and MercuryScreen [Doxy].

In general objects have a clear distinction between regular objects and screens. Regular objects are added to screens. Objects can be added to other objects as well. When an object is added to a screen, whether to put the object in as a perspective object (3D view) or an orthographic object (2D view) must be specified. When something happens to a parent object, its children inherit the effect. IE if a person is moved 30 units on Z, all of his body parts will be moved as well.

MercuryObject has the following virtual functions that it would make sense to overload:

- void Init( )
  This gets called after the creation of objects. Objects should not have initialization code in their constructor to allow for future multithreading processes. In here you should load any objects on your screen or object and set up the environment how you would like.
- void SetName(const MString & sName)
  This gets called any time the name of the object is changed. Normally objects don't need to override this.
- void Update(const float fDtime)
  This gets called on every update, it has the amount of time that has occurred since the previous update in fDtime. This allows you to do any necessary modification to your children or other game play.
- void Message(int Message, PStack & data, const Mstring & sName)
  Receive a message you have registered for. Notice that Message in this case is an integer, instead of a string. You must first register for messages and provide

an int and the name. Your ints should be #defined so that you can use a switch statement for the message number.

- void Render()
  Access to render is usually not necessary unless you do something that is required in a custom render cycle, such as modifying the camera. Most update code should be performed in update.

Commonly Used Objects

Mercury has a number of fairly common and useful objects that you can use in your screen or object. You may even find an object you'd like to abstract from.

- CopperWindow [Doxy]
- MercuryCamera [Doxy]
- MercuryLight [Doxy]
- MercuryLoadableModel [Doxy]
- MercuryObject [Doxy]
- MercuryODEObjectLoadable [Doxy]
- MercuryODEWorld [Doxy]
- MercuryParticleField [Doxy]
- MercuryScreen [Doxy]
- MercuryShape [Doxy]
- MercurySoundObject [Doxy]
- MercurySprite [Doxy]
- MercuryText [Doxy]
- ScreenOutdoors [Doxy]