GMTL Programmer's Guide

Generic Math Template Library

Allen Bierbaum,
Kevin Meinert,
Ben Scott,

$Date: 2002-07-15 19:52:31 $

Table of Contents

Introduction

GMTL stands for (G)eneric (M)ath (T)emplate (L)ibrary. It is a math library designed to be high-performance, extensible, and generic. The design is based upon discussion with many experts in the field of computer graphics and virtual reality and is the culmination of many previous graphics math library efforts. GMTL gives the graphics programmer several core math types and a rich library of graphics/ math operations on those types.

Design

The design of GMTL allows extensibility while mantaining a stable core. Core data types are separated from operations to improve encapsulation [Meyars]. This allows anyone to write their own math routines to extend or replace parts of the GMTL. This feature allows a very stable core set of math primitives that seldom change due to extensions, maintainance, or programmer error.

All math primitives in GMTL use generic programming techniques [ModernC++] to give the programmer many options to define their data. For example, matrices and vectors can be any dimension and any type. GMTL suffers no loss of performance due to these generalities because the parameter choices made are bound at *compile time*.

Implementation

GMTL is implemented using generic programming and template metaprogramming [ModernC++][GenerativeProgramming]. Generic programming allows selection by the user of size and type information for all data types in GMTL. For example, the generic Matrix type allows a programmer to select between any size (N x M) and any datatype (float, double, int...). The selection of these parameters is done through *template parameters*. To ease the use of these parameters, the system declares several typedefs that capture commonly used options.

Requested data types are statically bound and optimized by the compiler. The operations supplied with GMTL are implemented generically using a technique called *template metaprogramming*. Template metaprogramming allows things such as loops to be unrolled and conditionals to be evaluated *by the compiler*. Things such as loops and conditionals are evaluated statically, rather than at runtime. In addition, advanced optimizations can be performed that do this such as eliminate temporary variables and other intermediate computations. The result is compiled code that can behave as fast (or faster) then using traditional hand-coding methods such as loop unrolling, etc..

Testing

GMTL has an integrated test suite included in the source code distribution. The suite tests GMTL for correctness as well as performance degradation. The GMTL developers have put much time and effort into the test suite because we think that it will ensure that the code stays stable when changes are made, and that changes don't introduce performance hits. The bottom line is, if any behaviour changes in GMTL we want to know about it before it bites us. As a result of this philosophy, any contributions to GMTL also need to be well tested. Submissions will not be accepted without tests for correctness and performance.

The GMTL API

The GMTL API has two aspects you should keep in mind. The *data* types, and the *operations* on the data.

All data types and operations are defined in the `gmtl` namespace. Thus all types must be prefixed with the `gmtl::` scope or a `using gmtl;` command can be used to bring all of the GMTL functionality into the local scope.

Supplied GMTL Math Types

GMTL comes with many math data types: Vec, Point, Matrix, Quat, Coord, Sphere. The only member functions allowed within each class are:

- Constructor, Copy Constructor, and Destructor
- assignment operator - the compiler defines this automatically
- Set/Get member functions
- bracket (or paren) - operator for data element access.
- getData - function to retrieve a pointer to internal data.
- DataType - a typedef for the internal data format, useful for generic programming.

Additionally many of the types have predefined typedefs available for commonly used types. For example instead of typing gmtl::Matrix<4, 4, float>, a user could instead use the gmtl::Matrix44f typedef.

Filenames for each math type are always [PrimitiveType].h. For example, documentation on the gmtl::Quat type is located in Quat.h.

Supplied GMTL Operations

In Table 1. Mathematical operations supplied with GMTL are arranged into files by the following categories. we illustrate how the operations in GMTL are grouped and specified. Use this table for quick reference. Using the information here, such as file and function names, you can then go to the GMTL programmer reference for specific information. Alternatively you can look in the specified header files for documentation.

Table 1. Mathematical operations supplied with GMTL are arranged into files by the following categories.

Category	File	What you might find there	Discussion
Mathematical Operations	[PrimitiveType]Ops.h	• type operator*(type, type) • normalize(type) • bool invert(type)	Implements fundamental mathematical operations such as +, -, *, invert, dot product.
Geometric Transformations	Xform.h	• void xform(result, a, b) • type operator*(a, b)	Transforms a * b, stores into result.
Creational (Factory Functions)	Generate.h	• type makeTrans<type>(type) • type makeRot<type>(rad, x, y, z) • type makeRot<type>(x, y, z, rotation_order)	A "make" function should be thought of as a constructor. Any time a temporary is needed, use a "make" function. Make functions are *creational* in nature, and most takes data as input to "seed" that creation. The result can be a clone or a conversion from the input data. The "make" functions all create a *temporary* object and return it "out the back". The "make" functions are convenient in certain cases. For example to construct a translation matrix on one line of code: Matrix44f mat(makeTrans<Matrix44f>(1, 2, 3)); See [Patterns] for discussion on Factory. NOTE: Since the functions are inlined, a smart compiler should be able to optimize out the temporary making these perform fast. Beware, when using a dumb compiler (or debug mode), these functions will be slower (because of the temporary) than the get/set functions.
Setters (and Getters)	Generate.h	• getRot(type, result_rad, result_x, result_y, result_z) • setTrans(type, result_vec) • getScale(type, result_scale)	Setter functions all take some input, and write to some output. A "Getter" is simply a Setter except backward, and is why we refer to them all as "Setters".

Category	File	What you might find there	Discussion
			A Setter extracts information from const data, and writes information to the non-const data. Sometimes like in the case of x,y,z a group of inputs is considered one input. A Setter never returns a temporary "out the back", and because of this is usually more efficient than their "make" Creational counterpart. Following object oriented style, the first parameter is always the object being set (for set's) or read (for get's).
Conversions	Convert.h	convert(src, dest)	Convert one type to another. For example, convert Matrix to Quat, Quat to Matrix, and others... Convert functions are purely functional in nature since they do not particularly "belong" to either class. The first parameter is always the source (read-only), and the second is the destination of the converstion (writable).
Comparisons	[PrimitiveType]Ops.h	• bool operator==(type, type) • bool operator!=(type, type) • bool isEqual(type, type, tol)	Compare similar types. Compare functions always take two objects, and return a boolean. The special isEqual, takes a third parameter to specify a tolerance. Each of these three functions are defined for every type in GMTL.
OStream Outputs (operator<<)	Output.h	• std::ostream& operator<<(std::ostream&, type)	There is one operator<< defined for each Math type in GMTL.
C Math Abstraction	Math.h	• type Math::sin(type) • type Math::aTan2(type) • type Math::isEqual(type)	Anything you would find in the C math library (such as sinf, sin, cosf, cos, fabsf, or fabs, etc...) is templated here using only one name each. For example, instead of using ::sin() for 64bit float and ::sinf() for 32bit float, you can use the templated gmtl::Math::sin() function with allows the compiler to autodetect the datatype passed to it and select sinf or sin appropriately.

Category	File	What you might find there	Discussion
			NOTE: All items in the GMTL C Math abstraction are in the "gmtl::Math::" namespace.
Collision Detection	Intersection.h	• bool intersect(a, b)	Test whether a intersects b.
Bounding Volumes	Containment.h	• []	Builders of bounding volumes around geometric types such as Point, Sphere.
Template Metaprogramming Utilities	Meta.h	• class Type2Type	Template metaprogramming utilities for use in optimization of generic programming used throughout GMTL.

References

[Patterns] Erich Gamma. Richard Helm. Ralph Johnson. John Vlissides. *Design Patterns*. Elements of Reusable Object-Oriented Software. Addison Wesley . 1995. Copyright © 1995 Addison Wesley Longman, Inc.

[ModernC++] Andrei Alexandrescu. *Modern C++ Design*. Generic Programming and Design Patterns Applied. Addison Wesley . 2001. Copyright © 2001 Addison Wesley.

[GenerativeProgramming] Ulrich Eisenecker. Krzysztof Czarnecki. *Generative Programming*. Methods, Tools, and Applications. Addison Wesley Pub. Co.. 2000. Copyright © 2000 Addison Wesley.

[Meyars] Scott Meyars. “C++ Users Journal”. How Non-Member Functions Improve Encapsulation. CMP. February 2000.