5-1-2010

A plug-in based tool for numerical grid generation

Wali Akram Aziz

A PLUG-IN BASED TOOL FOR NUMERICAL GRID GENERATION

By

Wali Akram Aziz Jr.

A Thesis
Submitted to the Faculty of
Mississippi State University
in Partial Fulfillment of the Requirements
for the Degree of Master's of Science
in Computational Engineering
in the Department of Computational Engineering

Mississippi State, Mississippi

May 2010

A PLUG-IN BASED TOOL FOR NUMERICAL GRID GENERATION

By

Wali Akram Aziz Jr.

Approved:

Michael Remotigue Associate Research Professor (Major Professor)	Seongjai Kim Professor of Mathematics (Committee Member)
Ioana Banicescu Professor of Computer Science (Committee Member)	Mark Janus Associate Professor of Aerospace Engineering (Graduate Coordinator)
Sarah A. Rajala Dean of the College of Engineering	

Name: Wali Akram Aziz Jr.

Date of Degree: May 1, 2010

Institution: Mississippi State University

Major Field: Computational Engineering

Major Professor: Dr. Michael G. Remotigue

Title of Study: A PLUG-IN BASED TOOL FOR NUMERICAL GRID GENERATION

Pages in Study: 102

Candidate for Degree of Master of Science

The presented research summarizes (1) the development of a rapid prototyping framework, (2) the application of advance meshing algorithms, data structures, programming languages and libraries toward the field of numerical surface-water modeling (NSWM), (3) the application of (2) in (1), and (4) a real world application. The result of the research was the development of a prototype grid generator tool, the Mesh Generation and Refinement Tool (MGRT). MGRT supports a customizable interface and plug-and-play functionality through the use of plug-ins and incorporates a plug-in based topology/geometry system. A detailed explanation of the data structures, algorithms, and tools used to construct the MGRT are presented. Additionally, the construction of a mesh of Mobile Bay is presented. This represents a real world application of the MGRT. This tool provides many benefits over current tools in NSWM, which include faster meshing and the ability the use any grid generator that can be plugged-in.

DEDICATION

I would like to dedicate this research to

My parents, Wali Aziz Sr. and Carolyn Aziz,

Maryam Husband and family,

Archie and Mary Plump,

The Bright family,

The Bounds family,

Especially to my Wife and Best Friend Gabrielle Aziz,

Our wonderful daughter, Marielle Sephora Aziz

And I can't forget our cats Jazz, Ricky, and Isis.

ACKNOWLEDGMENTS

I would like to acknowledge the Northern Gulf Institute for providing funding for this research. I would also like to acknowledge Dr. Vladimir Alarcon of the Mississippi State University's Geosystems Research Institute for his interest in this research. Dr. Alarcon has co-authored two journal articles on this work and has provided valuable feedback time and time again. I also like to acknowledge my Major Professor, Dr. Remotigue, for allowing me to explore and learn the ins and outs of mesh generation, graphical user interfaces, topology, and computational geometry. I really enjoyed having the opportunity to try new ideas out and to see why they did or did not work. Lastly, I would like to acknowledge my parents Wali Aziz Sr. and Carolyn Aziz. Mom and dad thanks for your constant support and guidance. It has been a long journey lasting four years. I have seen many of my classmate leave before me, but I am on a different path. The knowledge, which I have acquired through this research and through my time spent working on this degree, has become an invaluable resource. And for everyone I did not mention thanks.

TABLE OF CONTENTS

Page

DEDICATION ii

ACKNOWLEDGMENTS iii

LIST OF FIGURES vi

CHAPTER

I. INTRODUCTION 1

II. NUMERICAL GRID GENERATION OVERVIEW 6

2.2 EDGE GRID GENERATION 7
2.3 UNSTRUCTURED GRID GENERATION 10
2.3.1 DELAUNAY TRIANGULATIONS 10
2.3.2 ADVANCING-FRONT POINT-INSERTIONS/ LOCAL RECONNECTION 13

III. DESIGN OVERVIEW 17

IV. IMPLEMENTATION 20

4.1 TOPOLOGY 20
4.1.1 GRID TOPOLOGY MOELING DATA STRUCTURE 20
4.1.2 MODIFICATIONS TO THE GRID TOPOLOGY MODEL 23
4.1.3 CURVE AND SURFACE FACTORY METHODS 25
4.2 STORAGE 27
4.2.1 DATABASE 28
4.2.2 QT INTERFACES 28
4.2.3 DATABASE READER, DATABASE WRITER, AND INTERACTIVE OBJECTS 29
4.3 PLUGIN SYSTEM 30
4.3.1 PLUG-IN CREATION 30
4.3.2 TYPES OF PLUG-INS 31
4.3.2.1 GENERIC MODEL INTERFACE 31
4.3.2.2 FUNCTION INTERFACE 32
4.3.2.3 DOCK MODULE INTERFACE 32

4.3.2.4 FILE READER INTERFACE 32
4.3.2.5 FILE WRITER INTERFACE 32
4.3.2.6 GRAPHICS DISPLAY INTERFACE 33
4.3.2.7 TOOLBAR INTERFACE 33
4.3.2.8 GTM CURVE INTERFACE 33
4.3.2.9 GTM SURFACE INTERFACE 34
4.4 GRAPHICAL USER INTERFACE 34

V. AN APPLICATION 41

5.1 INTRODUCTION 41
5.2 2.5D GRID GENERATION 44
5.3 MOBILE BAY INSTRUCTIONS 46
5.4 RESULTING GRID 48

VI. DEVELOPMENT TOOLS 53

VII. CONCLUSIONS 56

REFERENCES 58

APPENDIX

A. PLUG-IN API 61

B. PLUG-IN WALKTHROUGH 94

LIST OF FIGURES

FIGURE		Page
2.1	Edge Grid Generation	7
2.2	Voronoi Diagram and Delaunay Triangulation	11
2.3	Point Insertion with the Bowyer/Watson Algorithm	13
2.3	Edge Swapping	15
3.1	MGRT Major Component Relations	18
3.2	MGRT Core Library Components	19
4.1	GTM Topology Hierarchy	23
4.2	MGRT Topology Hierarchy	25
4.3	MGRT Storage Hierarchy	28
4.4	Available Dock Window and Toolbar locations	35
4.5	Overall View of the MGRT GUI	37
4.6	MGRT Mode Viewer and View Controls	38
4.7	MGRT Drop Down Menus	41
4.8	Plug-in Information Dialog	42
5.1	Study area. Coastline surrounding Mobile Bay, Alabama, USA	42
5.2	Conversion of National Oceanic and Atmospheric Administration-National Geophysical Data Center (NOAA-NGDC) coastline data for MGRT ingestion.	42
5.3	Grey Scale of Elevation of Mobile Bay, Alabama (USA)	43

5.4 Nine Point Stencil Used in Inverse Distance Weighting.....45

5.5 Overall View of Mobile Bay Grid.....48

5.6 Top View of Mobile Bay Grid 49

5.7 Middle View of Mobile Bay Grid 50

5.8 Lower Left View of Mobile Bay Grid.....51

5.9 Lower Right View of Mobile Bay Grid 52

CHAPTER I

INTRODUCTION

Numerical modeling of physical phenomenon is a popular form of engineering analysis. These models typically come in the form of a system of partial differentiable equations (PDEs). These systems of PDEs can be approximated algebraically over some discrete field of structured or unstructured points and elements. Popular problems of this class include computational fluid dynamics, environmental fluid dynamics, and computational electro-magnetics.

Observations and research of Tannehill [10] have determined that, one of the first steps in computing a numerical solution to the equations that describe a physical process is the construction of a grid. The physical domain must be covered with a mesh, so that discrete volumes or elements are identified where the conservation laws can be applied. A well-constructed grid greatly improves the quality of the solution, and conversely, a poorly constructed grid is a major contributor to a poor result. In many applications, difficulties with the numerical simulation can be traced to poor grid quality.

Numerical surface water modeling, like other numerical simulations, requires meshes of good quality, also. Many of the tools in this field have been based on older methods of grid generation and lack automation. Users of these tools do not have many alternatives for meshing.

A common tool for mesh generation in surface water modeling is SMS [13]. SMS is a commercial tool jointly developed by Brigham Young University and the US Army Corp. of Engineers. This tool is the fullest featured meshing tool on the market for surface water modeling. It has many automated features; however, it lacks many automated meshing algorithms such as those found in SolidMesh [26]. SMS's mesh generation routines require the user to move nodes, and many times, it meshes a convex hull of the problem domain, which requires the user to manually delete elements outside of the problem domain. Current generation mesh generators such as Advancing-Front/Local-Reconnection (AFLR) code [1][2] do not have this issue. Furthermore, SMS lacks many of the tried and proven edge distribution functions such as the hyperbolic sine and hyperbolic tangent distribution functions that allow for changes in boundary point spacing. There is a large amount of hands on manipulations of the discrete edge distributions with SMS. It is not uncommon for the user to manually specify each boundary node location in SMS. Many advanced meshing tools have the ability to apply edge distributions or point spacings globally. The user would set an ideal spacing between boundary nodes, and the mesh generator would then fill in other boundary points based on that criterion.

The motivation for this research is to demonstrate the capabilities of advanced meshing algorithms, data structures, and programming languages and libraries toward the field and to demonstrate these capabilities in a rapid prototyping framework. The goal of this thesis is to present the development and application of this research.

A prototype grid generation tool, called the Mesh Generation and Refinement Tool (MGRT), was created to satisfy these goals. The MGRT is a cross-platform, plug-in

based tool coded in the C++ programming language. MGRT has been developed using Qt4 [14][15][16], a comprehensive C++ application framework, which includes an object oriented graphical user interface (GUI) and Standard Template Library (STL) compatible container class-libraries and tools for cross-platform development. Through the uses of its plug-in system, the MGRT features customizable interfaces and functionality. It can host plug-ins, for meshing routines, file import and export routines, dock windows, toolbars, specialized graphics displays, or any other procedure, which require access to the MGRT topology data structures.

This paper presents the design and the application of the MGRT. The presented application is for the creation of a 2.5D unstructured grid of Mobile Bay (Alabama, USA) that will be used for hydrodynamics modeling. The unstructured meshing algorithm used in this particular application is Advancing-Front/Local-Reconnection (AFLR) [1][2]. This research shows results of grids created with MGRT.

After the introduction, Chapter II will discuss some background information on unstructured grid generation. It will detail the overall procedure for creating a grid. Then, the process of generating the edge grid will be discussed. An edge grid is the discretization of a boundary curve, which is used in the process of generating the unstructured grid. The next section discusses Delaunay tessellations, the Bowyer/Watson algorithm, local reconnection, and advancing front point placement. These sections all lead up to a discussion of the advancing front with local reconnection algorithm (AFLR) algorithm created by Marcum [1] [2]. The procedure followed by AFLR is outlined for a two dimensional case.

Chapter III outlines the design of the MGRT. It exposes the three major components of the tool and explains how they work together. Those major components are the Core Library, the Plug-ins, and the Graphical User Interface. Then there is a brief break down of the Core Library's components, which are the backbone of the tool.

Chapter IV outlines and explains the Topology Model, Storage Model, Plug-in System, and Graphical User Interface of the MGRT.

Chapter IV Section 1 outlines and discusses the data structures used in the MGRT. It details the Grid Topology Modeling (GTM) data structure created by Gaither [8], and details the modifications to the GTM data structures, which lead to the Modified Grid Topology Modeling (MGTM) data structures. It explains the hierarchical topology; and how the topology components interact to form the higher-level topology items.

Chapter IV Section 2 presents the Storage model used in the MGRT. It presents the abstraction model used and an overview of the components of the abstraction. This chapter also explains why the abstraction components were created and how the MGRT uses them.

The plug-in system is discussed in Chapter IV Section 3. Presented here is an overview of the MGRT plug-in system. It discusses the different types of plug-in components; and the purpose and interface for each.

Presented in Chapter IV Section 4 is an overview of the MGRT Graphical User Interface (GUI). This chapter explains the various mounting locations for toolbars and dock module plug-ins. It discusses how plug-in items are displayed in the GUI. Furthermore, it details the default GUI components and their use.

Chapter V presents the results of this research. It explains how the data set of the Mobile Bay was obtained and was modified for use with the MGRT. Then 2.5D Grid Generation and its implementation as a plug-in are discussed. Then the AFLR plug-in is discussed. After which, a step-by-step instruction to create the grid using the MGRT is presented. Finally, the resulting grid is presented.

Chapter VI presents the tools used in the development of the MGRT. The tools when used with the Qt Application Framework provided an integrated working environment.

The final chapter, Chapters VII, concludes this thesis. Chapter VII presents concluding remarks of the results and of the research.

CHAPTER II

NUMERICAL GRID GENERATION OVERVIEW

The MGRT uses a boundary representation (B-Rep) of geometry. A B-Rep is a piece of geometry composed of edges, vertices, loops, and faces, which together describe the boundaries of the geometry. This research will limit the discussion to B-Reps in two dimensions. As such, the highest dimensional object is of dimension two in Real and Parametric Space. To create a grid using such a representation we must follow the following procedure.

For this study there are six steps to create a grid.

1. Define geometry.
2. Apply the grid control information. Point spacings are applied to edges and edge growths are applied to loops (Gaither, et.al.[5][6][7][8]).
3. Compute edge discretizations.
4. Compute the two-dimensional grid. Grids can be computed using a Delaunay triangulation algorithm or an advancing front algorithm.
5. Evaluate the two-dimensional grid quality. If the quality metrics identify problems, iterate back to the grid control phase to solve any problems with the grid (Gaither [5]).
6. Apply elevation data to the grid

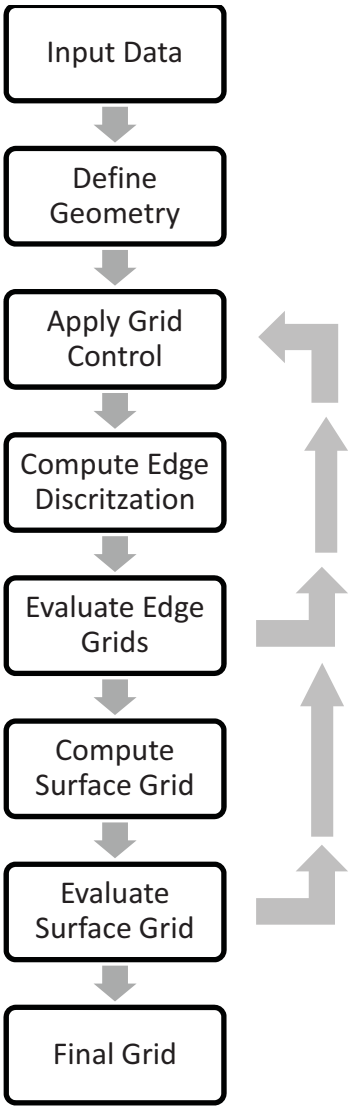

Figure 2.1 Edge Grid Generation

2.2 EDGE GRID GENERATION

Edge grid generation is the process of discretizing a continuous curve. In this study, edge grid generation on curve (edge) geometry, i.e. shoreline data, is performed in parametric space. According to Remotigue [17], it has been acknowledged that a parametric geometry description and associated operations will aid in maintaining geometric fidelity within the grid generation procedure. Operating in the parametric

range allows for a more compact storage and uniform access of edge grid information. Instead of storing three double precision numbers representing coordinates in real space (R^3), one double precision number can be used. Furthermore, it abstracts knowledge of the edge's shape when being computed. Coordinates in real space can be quickly calculated by mapping the parametric coordinates to a curve's geometry.

The MGRT has four techniques for generating edge grids: uniform, two-sided hyperbolic sine, two-sided hyperbolic tangent, and one-dimensional advancing front. Thompson [11] calculates a uniform stretching of a given number of points on a curve using a Lagrange interpolation. See Equation 2.1 for details.

$$s\left(\frac{\xi}{I}\right) = \sum_{n=2}^{N} \phi_N\left(\frac{\xi}{I}\right)[r_n - r_0] \times I \tag{2.1}$$

The initial steps to calculating a hyperbolic sine or hyperbolic tangent point distribution are as follows Vinokur [4] and Thompson [11]:

$$A = \sqrt{\frac{\Delta s_1}{\Delta s_2}}. \tag{2.2}$$

$$B = \frac{1}{I\sqrt{(\Delta s_1 \Delta s_2)}}. \tag{2.3}$$

In Equation 2.4, delta must be solved iteratively. Using Newton's method and initial guess of $\delta = 2B.$, delta can be found within five iterations.

$$B = \frac{\sinh\delta}{\delta}. \tag{2.4}$$

Use Equation 2.5 in Equation 2.7 to solve for a hyperbolic tangent distribution. Similarly, use Equation 2.6 in Equation 2.7 to solve for hyperbolic sine distribution.

$$u\left(\frac{\xi}{I}\right)=\frac{1}{2}\left(1+\frac{\tanh\left(\delta\times\left(\frac{\xi}{I}-\frac{1}{2}\right)\right)}{\tanh\left(\frac{\delta}{2}\right)}\right). \tag{2.5}$$

$$u\left(\frac{\xi}{I}\right)=\frac{1}{2}\left(1+\frac{\sinh\left(\delta\times\left(\frac{\xi}{I}-\frac{1}{2}\right)\right)}{\sinh\left(\frac{\delta}{2}\right)}\right). \tag{2.6}$$

$$s\left(\frac{\xi}{I}\right)=\frac{u\left(\frac{\xi}{I}\right)}{A+(1-A)\times u\left(\frac{\xi}{I}\right)}. \tag{2.7}$$

Gaither [8] and Marcum [1] [2] describe a general method of producing a one-dimensional advancing front edge grid. The grid is created by setting a point-spacing at each end of an edge. Then, points are inserted from the endpoints inward. The spacing between the points is weighted and smoothed, such that the final distribution varies seamlessly between the endpoints. The final number of points is unknown until all the points are inserted.

The one-dimensional advancing front method of producing edge grids is ideal for unstructured mesh generation, since edges are not point matched within boundary loops. The other methods are good when a specific number of points on an edge are required like in structured grid generation. Structured grid generation constructs two-dimensional grids based on a computational square. The number of points must the same on opposing sides of the computational square.

2.3 UNSTRUCTURED GRID GENERATION

2.3.1 DELAUNAY TRIANGULATIONS

A popular method of unstructured grid generation is Delaunay Triangulation. Delaunay creates a triangulation of an existing set of points. Mavriplis [18], Tannehill [10], and Thompson [3] describe Delaunay as having the following properties, which are beneficial in grid generation:

- A valid Delaunay triangulation does not contain any other points except those contained in the triangulation.
- The closest points are always connected.
- In a valid Delaunay triangulation maintains the empty circum-circle property. This property states that no point in the triangulation can be contained inside the circum-circle of any triangle.
- The dual of a Delaunay triangulation is a Voronoi Tessellation ([3] [18]). A Voronoi Dual is the smallest containment region of a point in the triangulation. It is obtained from a Delaunay triangulation by connecting line segments from each triangles centroid to the mid points of the segments, which make up the triangles edges. (See Figure 2.2)
- Delaunay triangulations implicitly maintain a minimum angle criterion. As such, it incurs the largest minimum angle for all triangular elements states Mavriplis [18]. Thus, a Delaunay triangulation is expected to result in "well-shaped" elements, without very small angles (Mavriplis [18]).
- Delaunay triangulations can be extended to n-dimensions.

- For the purpose of mesh generation, the grid point generation and the triangulation are decoupled (Tannehill [10]).

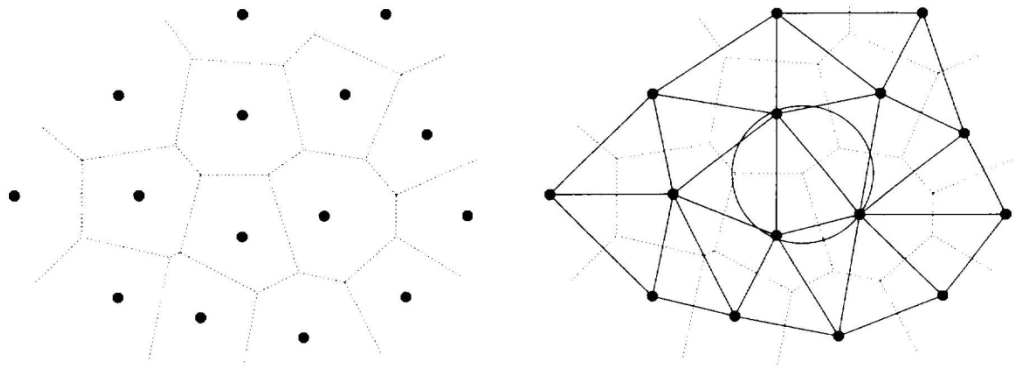

Figure 2.2 Voronoi Diagram and Delaunay Triangulation

The Bowyer-Watson Algorithm is a popular algorithm used to construct Delaunay triangulations. It is an incremental point insertion method, which adds points sequentially to an existing valid Delaunay triangulation (Mavriplis [18] and Tannehill [10]). For this algorithm, the initial triangulation must be large enough to enclose all points to be inserted. When a point is inserted into the triangulation, a search is performed to find all triangles whose circum-circle encloses the newly inserted point. These triangles are deleted and a convex hull is left. This convex region is then re-triangulated using the newly inserted point and the points forming the hull. Connecting the inserted point with each point on the hull forms the new elements. This iterative process continues until all desired points are inserted. The resulting triangulation is a valid Delaunay triangulation. Mavriplis [18] states that the Bowyer-Watson algorithm has a worst-case complexity of O (N^2). The worst case can be avoided by randomizing the order of point insertions. However, for unstructured mesh generation, near linear

O(N) performance has been reported for both two- and three-dimensional applications (Mavriplis [18]). Baker (1987) studied the Bowyer insertion scheme and has shown that the method is based upon two theorems stated by Tannehill [10].

Theorem 1: Given a Delaunay triangulation T of a planar set of points S, introduce a new point $p \in S$ and remove all triangles that fail the Delaunay circle test. All the edges of the Delaunay cavity are visible form point *p*.

Theorem 2: The re-triangulation of a Delaunay cavity, by joining the point *p* to each of the boundary points of the cavity, is Delaunay.

A Delaunay triangulation can be constructed using Bowyer/Watson using the following algorithm.

1. Generate a set of boundary points.
2. Create an initial valid Delaunay triangulation of a geometric simplex, such as a triangle or rectangle, which completely encloses the entire domain.
3. Insert a mesh point into the existing triangulation, and delete the first triangle that fails the circum-circle test. This will be the cell where the point is inserted.
4. Initiate a search of the neighboring cells to determine if any other neighbors have violated circum-circles. If so, the neighbor cell is deleted, and that cell's neighbors are searched in a breadth-first search style until the list of deleted cells is compiled.

5. Establish new connectivity by connecting the newly inserted point the boundary points of the cavity created by the deleted cells. Add each of the new cells the list of valid triangles.

Repeat this procedure, starting with step 3, until all the points are inserted.

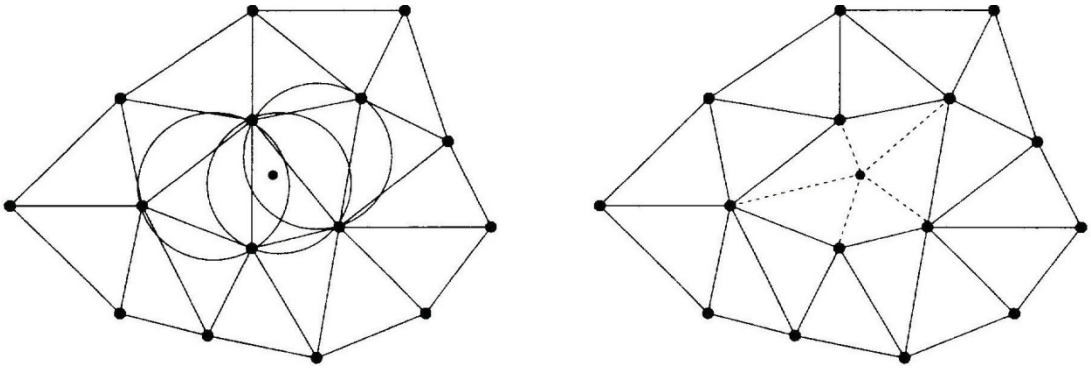

Figure 2.3 Point Insertion with the Bowyer/Watson Algorithm

2.3.2 ADVANCING-FRONT POINT-INSERTIONS/ LOCAL RECONNECTION

The Advancing Front/Local Reconnection (AFLR) grid generation code created by Marcum [1] [2] is the meshing routine used in this study. AFLR differs from traditional advancing front methods in the sense that it creates meshes by forming new elements in an existing mesh using a combination of advancing front point insertion and local reconnection. The scheme used by AFLR allows the grid to maintain validity throughout the grid generation process (i.e. no negative areas). There are no particular properties, such as Delaunay, which has to be maintained, and it allows for optimal point placement. The grids produced by AFLR are composed of predominately "well-shaped" elements.

Grid generation codes, like AFLR [1] [2], use methods like the Bowyer-Watson algorithm to insert boundary points in to a grid. The Bowyer-Watson algorithm performs well in this respect, but the method has to have a set of points to insert. AFLR uses an advancing-front point placement strategy to develop a list of points to be inserted. Calculating candidate point locations based on the grid's front generates these points. The front is the edges of elements, which have already been created and are exposed to the region of elements, which were not created using inserted points. The edges are used to determine the ideal point locations to form new elements. These locations are filtered for proximity. The passing candidate points are inserted into the grid using the Bowyer-Watson algorithm. Initially, the front is the set of edges composing the internal and external boundaries of the grid. Edges are removed from the front as new elements are formed from inserted points. The exposed edges of new elements are added to the front. The front can be visualized as a set of moving boundaries, which expand until they collide. At which point, there are no more edges on the front.

Local Reconnection is also called edge swapping. It is the process of improving grid quality by swapping the edges of adjacent triangles to satisfy some criteria. Three other criteria may be used to optimize elements locally: (1) minimize the maximum angle, (2) maximize the minimum angle, or (3) satisfy a Delaunay criterion for the pair of triangles. In the AFLR code, a local reconnection step is performed each time after the Bowyer-Watson algorithm inserts all of its candidate nodes for a given front.

> Mavriplis [18] states, "For each pair of triangles in the mesh which forms a convex quadrilateral, the original triangulation is compared to the alternative triangulation obtained by swapping the position of the internal diagonal. […] If

the alternative triangulation is found to better optimize the triangulation criterion, then the diagonal is swapped. By iterative application of this simple diagonal swapping primitive over the entire mesh, the triangulation is eventually transformed into a more optimal triangulation."

Mavriplis [18] goes on to state that the complexity of the edge swapping procedure is $O(N \log N)$, where N is the number of vertices.

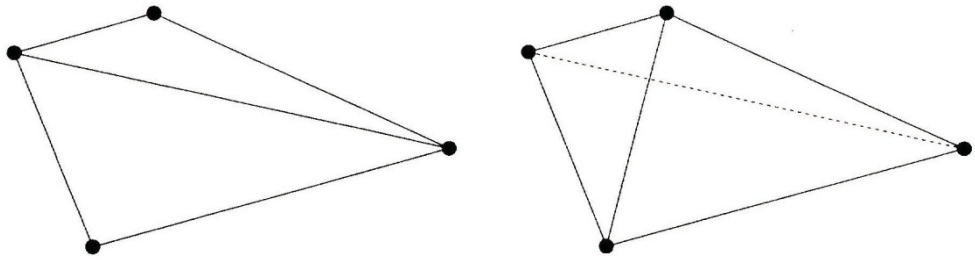

Figure 2.4 Edge Swapping

A simplified two-dimensional AFLR procedure can be constructed from the literature of Marcum [1] [2] and Mavriplis [18] as the following.

1. Generate boundary curve grids for the given configuration.
2. Obtain a valid triangulation of the boundary points and recover all boundary edges.
3. Delete elements outside the external boundary.
4. Assign a point distribution function to each initial boundary point.
5. Initialize the front with the boundaries.
6. Create a new point for each active element.
7. For each new point, search for the element which contains it.

8. Interpolate the point distribution function for the new point from the containing element (Marcum [1] [2]).
9. Reject new points that are too close to an existing point or another new point (Marcum [1] [2]).
10. Directly insert each accepted new point using Bowyer/Watson algorithm.
11. For each active element, compare the reconnection criterion for all allowable connectivity with adjacent elements and reconnect using the most optimal connectivity.
12. Repeat the local reconnection process until no elements are reconnected.
13. Update the front.
14. Repeat steps 6-12, until no new points are created or accepted.
15. Smooth the coordinates of the field grid point.

Repeat the reconnection steps 11 and 12 with all elements activated.

CHAPTER III

DESIGN OVERVIEW

The MGRT is designed to take advantage of current generation data structures, algorithms, programming, languages, and libraries. The MGRT is a cross-platform tool, which has been coded in the C++ programming language. C++ was chosen for its robustness and for use of its standard template library (STL). Furthermore, the tool uses Qt4, a comprehensive C++ application framework, which includes an object oriented graphical user interface (GUI) and STL compatible container class-libraries and tools for cross-platform development [15]. Through the uses of a plug-in framework, the MGRT features customizable interfaces and functionality. Through its design, it can host plug-ins, for meshing routines, file import and export routines, dockable windows, toolbars, specialized graphics displays, or any other procedure, which require access to the MGRT topology data structures.

With modularity in mind, the MGRT consists of three major components, the Core Library, the GUI, and Plug-in components. The core library is the bonding mechanism for the tool. The objective of the core is to serve as a central repository of common data structures and routines for the GUI and Plug-in components. This library is linked statically to the other components to simplify cross-platform builds. This simplification is significant on Microsoft Windows' build of the MGRT. This is due to the Windows' requirement of *__declspec(dllimport)* and *__declspec(dllexport)* on all

declarations based on the type of build, linking, or compiling. As such, the GUI and Plug-in components store the compiled objects of the Core Library locally. The Core Library needs to only be compiled once. If local changes are made to either GUI or Plug-in components, the Core does not have to be recompiled.

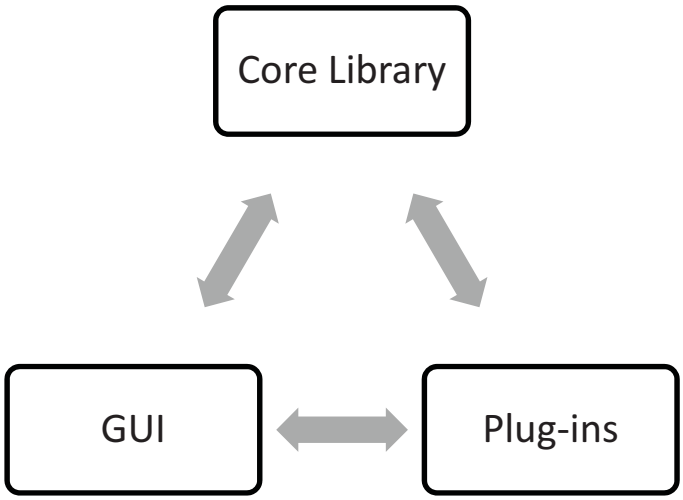

Figure 3.1 MGRT Major Component Relations

As mentioned earlier, the Core Library is the glue that holds the MGRT together. It contains common data structures and routines for the topology model, the storage mechanism, and the plug-in interfaces. See Figure 3.1. The idea is that if both GUI and Plug-in component link to the same library, the two components when dynamically linked at runtime should not have any interference when passing data. The components of the Core Library are discusses in detail in Chapter IV.

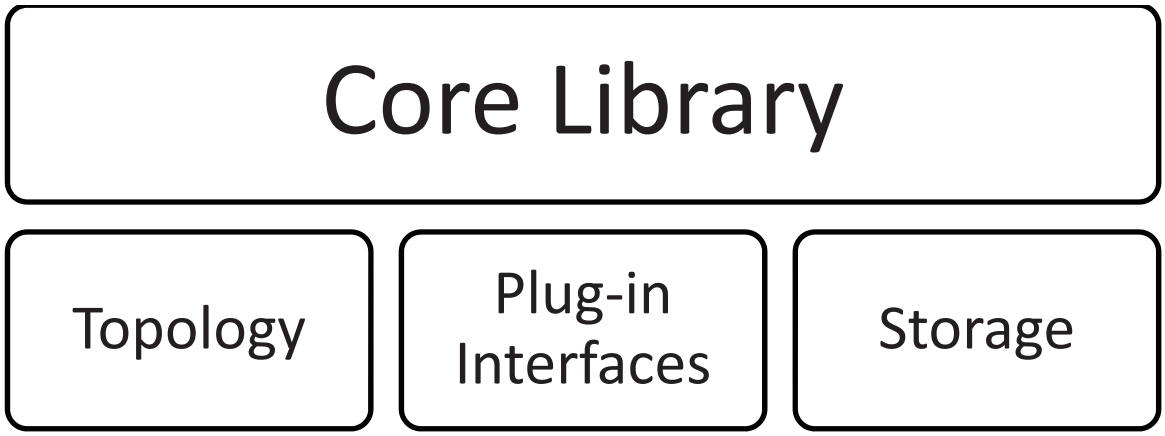

Figure 3.2 MGRT Core Library Components

CHAPTER IV

IMPLEMENTATION

The following sections outline and discuss the three major components of the core library and the present MGRT's graphical user interface. The first section 4.1 discusses the topology model. The topology model is the mechanism MGRT uses to determine how geometry is connected and oriented. Section 4.2 discusses how data is stored. It, also, discusses the storage abstractions used in the tool. Section 4.3 outlines the available plug-in interfaces. This section explains how the plug-in are loaded and how the interfaces are used. Finally, section 4.4 discusses the graphic user interface. It outlines how GUI component are organized in the tool, and presents pictures of the MGRT's GUI.

4.1 TOPOLOGY

4.1.1 GRID TOPOLOGY MOELING DATA STRUCTURE

The MGRT utilizes the Grid Topology Modeling data structure (GTM) data structure. The GTM defines a hierarchy of topology and geometry elements. According to Gaither [5][6][7][8], the GTM allows for fast and efficient adjacency traversals, and provides a space efficient for storing grid data. Furthermore, it has been successfully implemented in two different production grid generation systems, SolidMesh [8][26] and GUM-B [8][16].

According to Gaither [8], the GTM is based on the radial-edge non-manifold (RENM) data structured design. Furthermore, Gaither [8] continues on to state that the RENM has been used by every CAD system that uses a boundary representation non-manifold data structure for solid modeling applications. The efficiency of the GTM adjacency traversals, like the RENM from which it is based on, is linear within each data structure (Gaither [8]).

The GTM data structure consists of the following elements: shell, block, face, loop, edge, vertex, face-use, edge-use, vertex-use, point, curve, and surface. Furthermore, it defines four levels of abstraction, from the top down: Volume Topology, Surface Topology, Geometry Abstraction, and Geometry Levels. According to Gaither [8] the GTM data structures are defined as follows:

- Shell: A shell is an oriented volume bounded by any number of faces. It is an element of both unstructured and hybrid grid topologies. A face is connected to a shell through a face-use. The face-use's orientation is determined by the direction of the associated surface geometry's normal direction. All surface normals point consistently either inwards or outwards from the interior of the shell's volume.
- Block: A block is a right-handed (positive Jacobian) oriented volume bounded by six (logical) faces. Blocks are elements in structured multi-block grid topologies. A face is connected to a block through a face-use. A face-use's orientation is determined from its location on the block and the orientation of the face geometry. Orientations from one face to another on a block are implied from the location on the block.

- Face: A face is an orientable 2D abstraction of a surface. A face can be bounded by either four edges associated with the derived curves form the original surface geometry, or by any number of non-degenerate oriented trimming loops that are defined as sets of edges with associated parametric curves on the surface. Any number of face-uses that share the face determines a face's orientation.
- Loop: A loop is an oriented collection of non-degenerate edges in either 2D space or the parametric space of a parent-trimmed surface. Loops run in a counter-clockwise direction whether in 2D space or in parametric space of the surface.
- Edge: An edge is an orientable 1D abstraction of a curve. An edge is connected to a face either directly to a face-use are one of four derived edges of the original surface, or through a loop as part of a trimming loop. Any number of associated edge-uses determines an edge's orientation. An edge is orientable, though not oriented; it is the uses of an edge, which are oriented.
- Vertex: A vertex is an abstraction of a point in space.
- Edge-use: An edge use is an element that defines the use of an edge by a face or loop. The edge-use provides adjacency information for the edge, defines the orientation of the edge with respect to the face or loop that is using it, and stores the edge grid information with respect to the face or loop.
- Face-use: A face use is an element that defines the use of a face by a block or shell. The face-use provides adjacency information for the face, defines the

orientation of the face with respect to the shell or block that is using it, and stores the face grid information.

- Vertex-use: A vertex-use is an element that defines the use of a vertex by an edge. The vertex-use provides adjacency information for the vertex and stores the vertex grid information.

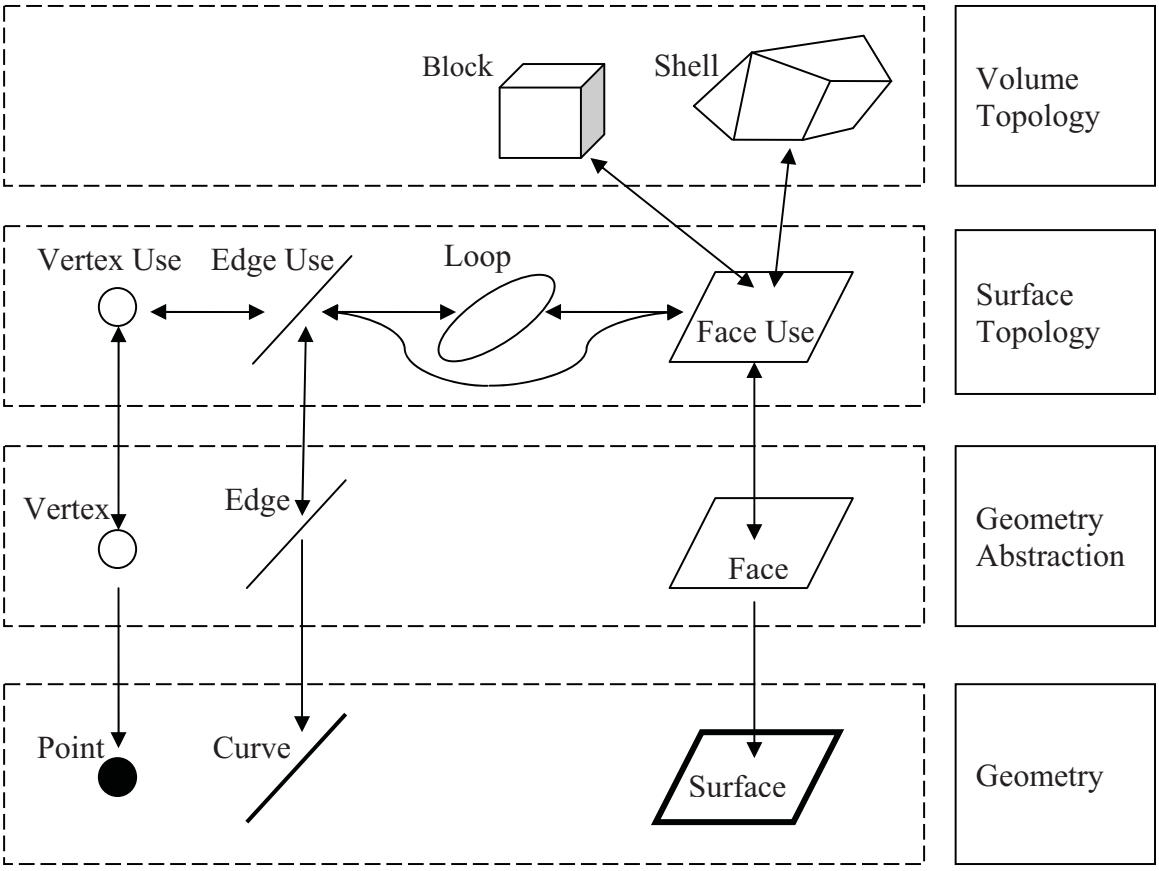

Figure 4.1 GTM Topology Hierarchy

4.1.2 MODIFICATIONS TO THE GRID TOPOLOGY MODEL

The GTM's hierarchy has an additional level of abstraction, the Geometry Components layer. This level shows that the curve and surface elements of the GTM are

only encapsulations of interfaces to access lower level geometry definitions. With this said, the definitions of curves and surfaces are modified to be more of an interface or an encapsulation of their lower level counter parts. Additionally, the Volume Topology layer will be ignored since the implementation for this work is in two dimensions.

Definitions of Modified GTM elements:

- Curve: A curve is a simplified interface for higher-level abstraction to access curve information. Its stored curve definition and a curve data objects define it.
- Surface: A surface is a uniform interface for higher levels of abstractions to access surface information. Its stored surface definition and surface data objects define it.
- Curve Definition: A curve definition is an element, which answers queries about a curve. It interprets data in curve data elements to do this. A curve definition stores no data of its own; it only provides functionally to answer queries about a curve data element.
- Surface Definition: A surface definition is an element, which answers queries about a surface. It interprets data in surface data elements to do this. A surface definition stores no data of its own; it only provides functionally to answer queries about a surface data element.
- Curve Data: A curve data element is a container object, which provides a common and generic storage mechanism for curves. A curve data object without a definition is only a collection of integers and double precision numbers. A curve definition interprets the data to answer queries.

- Surface Data: A surface data element is a container object, which provides a common and generic storage mechanism for curves. As such, it is only a collection of integers and double precision numbers. A surface definition element interprets the data to answer queries.

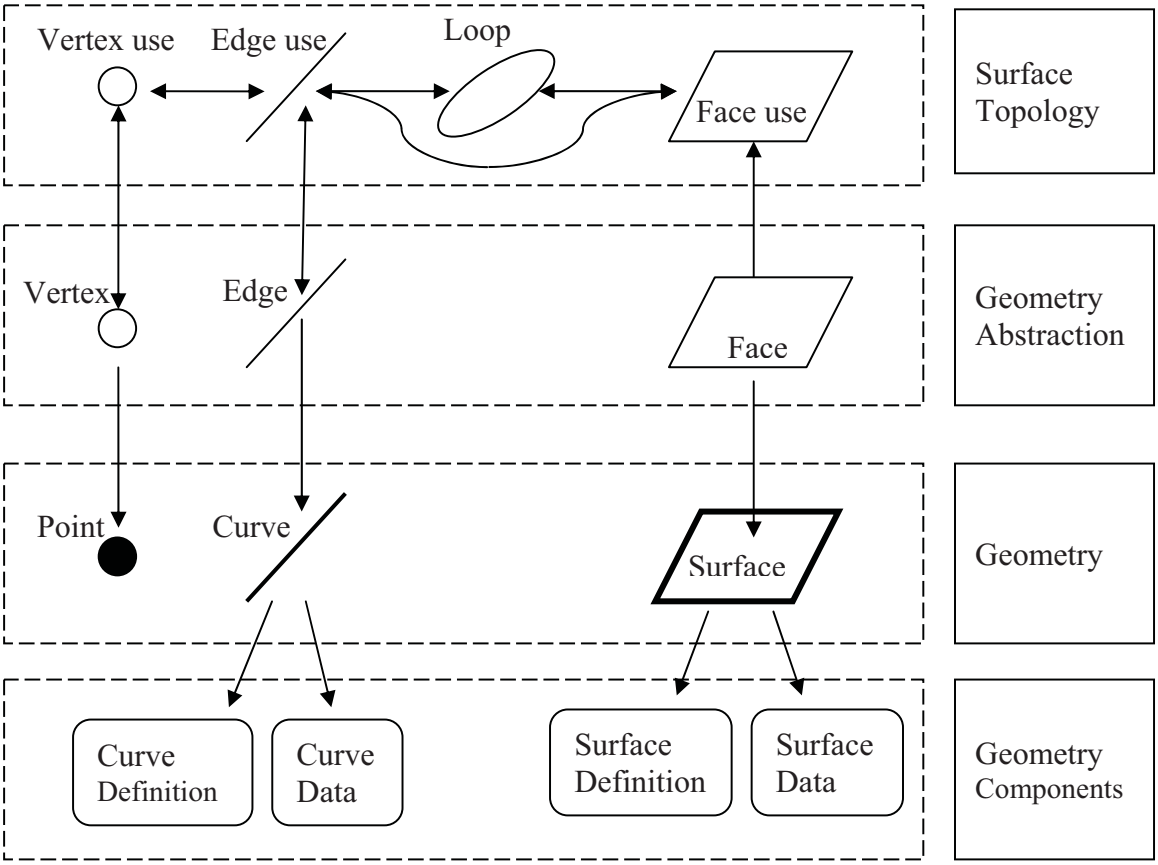

Figure 4.2 MGTM Topology Hierarchy

4.1.3 CURVE AND SURFACE FACTORY METHODS

The curves and surfaces are created with two factory methods. These methods construct curves and surfaces from a definition and a data component. When the MGRT

is started, the curve and surface definitions available to the program are stored and indexed by a unique name. As a curve or surface is constructed, the factory method uses a data object that specifies the name of its associated definition. The method then recalls a definition by looking up its name. If the definition cannot be found, the curve or surface is not created. Otherwise, the curve or surface is created. For more information on the plug-in based curves and surfaces refer to sections 5.4, 6.2.8, and 6.2.9

4.2 STORAGE

While the GTM handles geometry and topology of the MGRT, it does not address referencing and storing data. The generic item storage class was created to fill this gap. It converts pointers of inserted items into void pointers. The stored pointer is assigned type and reference identification numbers. The type-id is used to determine what type the void pointer points to, such that the void pointer can be correctly re-casted. The reference-id is used as a unique identifier of each item within a type.

The generic storage class stores the data in vectors of maps, such that items can be quickly accessed using type and reference number or a type and a pointer. The type and reference number are typically used to retrieve a pointer. A type and a pointer can be used to look up a reference number.

A list of available reference numbers is maintained in a minimum priority queue. As an item is removed, its reference number is added to the queue and its pointer is removed from the storage maps. When inserting an item, if the queue is not empty, a reference number is taken from the top of the queue and is used for the newly inserted item. Using the priority queue in this way reduces the values of reference numbers for new items after multiple deletions. This ensures that available reference numbers are filled before any new reference numbers are added.

The Generic Storage class is the lowest level of access in the storage abstraction method used by the MGRT. Figure 5.1 shows the full storage abstraction strategy.

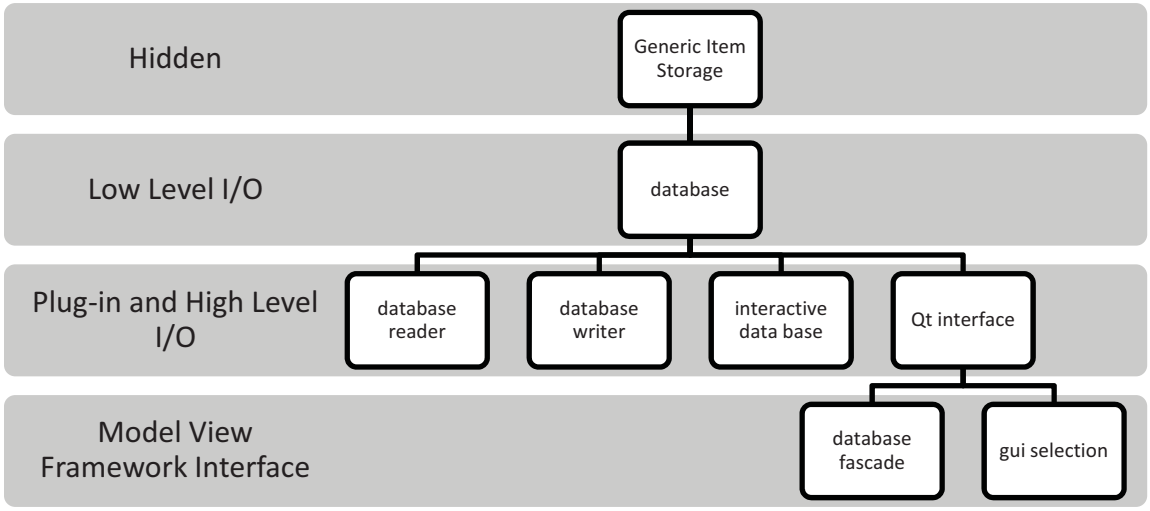

Figure 4.3 MGRT Storage Hierarchy

4.2.1 DATABASE

The database is an abstraction of the Generic Storage class. It provides a simpler interface than the Generic Item Storage Class. Furthermore, it performs topology specific actions for inserted data. For example, vertices are checked for proximity to existing vertices when being inserted into the database; and loops are updated when attached edge-uses become updated. This class provides more control and functionality than the Plug-in and High Level I/O objects.

4.2.2 QT INTERFACES

The database facade uses Qt's built in model view framework to interact with data in the database. It is used to display data with Qt's various item view (i.e. the tree view).

This makes it easily used with the Qt's GUI or any other component using the Qt model view framework. As such, it provides an end user a list, table, or tree representation of stored data and its associated properties. The user can select an item here and adjust the parameters of the data. All operations to the data the database façade are communicated to all items connected to the storage system. The communication is done by communicating changes down to the lowest level in the hierarchy. Then the changes are communicated back down to the other components.

The GUI selection class like the database facade is, also, part of the model view framework. As such, it serves as an intermediary between the model view framework and the database.

4.2.3 DATABASE READER, DATABASE WRITER, AND INTERACTIVE OBJECTS

There are three classes used by plug-ins to interact with the database, the *Write-to-Database* class, the *Read-from-Database* class, and the *Interactive Database Object* class. Data transfer was done in this manner to provide a simple and straightforward interface to plug-ins and to hide unnecessary and dangerous implementation details of the database class. In addition, these classes have signaling mechanisms, which communicate information between the plug-in and the database.

The *Write-to-Database* class has the ability to add items to the database. The *Read-from-Database* class can retrieve items from the database by type and selection status. The *Interactive Database Object* is a merger of the *Read-from-Database* and *Write-to-Database* classes. It was designed to be used with plug-ins, which required more information that what was provided with the *Read-from-Database* and *Write-to-*

Database classes. The *Write-to-Database* and *Interactive Database Object* also, have access to the surface and curve factory methods. These are provided for the creation and insertion of curves and surfaces, which can be inserted into the *Database*.

4.3 PLUGIN SYSTEM

A flexible plug-in system is the best way to respond to changing software requirements. To this end, the Qt framework has a plug-in architecture that allows the MGRT to support plug-ins [12] [16]. The advantage of the plug-in system is that the plug-in source code can be compiled separately from the MGRT source code, which alleviates the need to rebuild the tool whenever functionality is to be added or removed. Furthermore, a plug-in is plug-and-play. Meaning that a plug-in can be added by placing it in the MGRT's plug-in folder; likewise, a plug-in can be removed by removing it from the folder. These plug-ins can be meshing routines, file import and export routines, dockable windows, toolbars, specialized graphics displays, or any other procedure which require access to the MGRT's GTM data structures.

4.3.1 PLUG-IN CREATION

To create a plug-in, the plug-in and the program using it need a defined method of communication. In Qt, this is achieved by using an interface [16]. An interface is an abstract C++ class that contains a set of functions which are used to communicate between the MRGT and plug-in. This is implemented by defining the same interface in the MGRT's source code, and in the plug-in's source code. This is simplified by creating a single header file containing the interface and both codes link to it.

The plug-in is compiled to a shared or static library. The library can then be placed in the MGRT's plug-in folder. On startup the MGRT searches the plug-ins folder for possible plug-ins. It attempts to load any file that "looks like" a plug-in by trying to match the objects the libraries found with its known list of interfaces. If the interfaces match, it continues loading the plug-in. The Qt documentation [14] [15] [16] outlines the minute details of creating plug-ins.

4.3.2 TYPES OF PLUG-INS

There currently are nine unique plug-in interfaces in the MGRT. As previously mentioned, an interface is the common ground which the main code and plug-in communicate. The interfaces are purely abstract classes. A plug-in's implementation defines how an interface is used. Furthermore, a plug-in can use more than one interface to compound functionality. The nine unique interfaces are the *generic model, function, dock module, file reader, file writer, graphic display, toolbar, curve* and the *surface* plug-in interfaces. The interfaces provide data access to the plug-ins through the use of three classes: a *Write-to-Database* class, a *Read-from-Database* class, and an *Interactive Database Object* class. Data transfer was done in this manner to provide a simple and straightforward interface to plug-ins and to hide unnecessary and dangerous implementation details of the database class.

4.3.2.1 GENERIC MODEL INTERFACE

The *generic model* interface is a stripped down interface, which can only be passed a *Read-from-Database* and *Write-to-Database* object. It is defined as a simple, no frills interface for testing plug-in functionality.

4.3.2.2 FUNCTION INTERFACE

The *function interface*, when loaded with a menu item with the plug-in's name is added to the plug-in menu of the MGRT. If this item is clicked, the function defined by the implementation of this interface is called. This interface uses *Read-from-Database* and *Write-to-Database* objects. If desired, it can show a modal dialog when called.

4.3.2.3 DOCK MODULE INTERFACE

The *dock module interface*, adds a dockable window to the MGRT's interface. The dock window can be assigned which locations it can and cannot dock in its implementation. This interface provides a plug-in with *Read-from-Database* and *Write-to-Database* objects.

4.3.2.4 FILE READER INTERFACE

The *file reader interface* is used to add new file formats for reading to the MGRT. It adds file data to the generic storage with a *Write-to-Database* object. Furthermore, it provides the MGRT with the file extensions it can read. These extensions can be used as filters when loading files. They also are used in a map for fast check to determine if a file can be read. This interface, also, defines a more comprehensive check to determine if a file can be read. This check, which is called before the file name, is passed along to the interface's file reading method.

4.3.2.5 FILE WRITER INTERFACE

The file *writer interface* is similar to the *file reader*. It is used to add new file formats for writing to the MGRT. It writes file data using a *Read-from-Database* object.

Furthermore, it provides the MGRT with the file extensions it can write. These extensions can be used as file filters when specifying a file to write. Furthermore, when a filter is not selected, the extensions of the file is checked against all know extensions. Possible known extensions that match, the MGRT attempts to read the file with each until a successful read.

4.3.2.6 GRAPHICS DISPLAY INTERFACE

The *graphic display interface* is used to plug-in new main displays for the MGRT. While the interface is called graphic display, the plugged-in display does not have to be graphics (OpenGL) based. The display can be any GUI item or set of GUI items the plug-in defines. This interface provides a plug-in with an *Interactive Database Object*.

4.3.2.7 TOOLBAR INTERFACE

The *toolbar interface* gives a plug-in the ability to add a toolbar to the MGRT's interface. The buttons on a plugged-in toolbar can call dialogs or other GUI items defined by the plug-in. This interface provides the plug-in with *Read-from-Database* and *Write-to-Database* objects.

4.3.2.8 GTM CURVE INTERFACE

The *curve plug-in interface* is used to add new curve definition types as defined in the modified GTM. The new curve type is added to a curve factory method. A curve of the defined type can be created with the factory method if a provided curve data object matches in type with the curve definition.

4.3.2.9 GTM SURFACE INTERFACE

The *surface plug-in interface* is used to add new surface definition types as defined in the modified GTM. The new surface type is added to a surface factory method. A surface of the defined type can be created with the factory method if a provided surface data object matches in type with the surface definition.

4.4 GRAPHICAL USER INTERFACE

The Graphical User Interface (GUI) is composed of menus, toolbars, dock windows, status bar, and the Central Area. As shown in Figure 7.1, menu resides at the top of the MGRT window, the status bar at the lower. The Qt documentation [15] states, "Toolbars contain collections of buttons and other widgets that the user can access to perform actions. They can be moved between the areas at the top, left, right, and bottom of the central area of a main window. Any toolbar can be dragged out of its toolbar area, and floated as an independent tool palette." The Qt Whitepaper [15] further states, "Dock windows are windows that the user can move inside a toolbar area or from one toolbar area to another. The user can undock a dock window and make it float on top of the application, or minimize it."

Dock windows, toolbar, and menu items can be added to MGRT using plug-ins. The toolbar plug-in interface must be implemented to add a toolbar. Dock windows and toolbars can be toggled on and off by clicking the items name in the menu *View-> Item Name* or right clicking in the toolbar area and clicking the Item Name. The function and graphic display plug-in interfaces support adding menu items to the plug-in menu. The function plug-in interface adds menu items to the

submenu -> plug-ins -> routines. Clicking on a submenu item here will launch a dialog if the plug-in supports it or it would invoke the routines the plug-in implements. These dialogs are defined in the implementation of the plug-in. As such, plug-in designers can use it as a GUI front end to the routines the plug-in implements. The graphic display plug-in interface adds a submenu item to -> *plug-ins -> Main Display*. Clicking a submenu item here changes the Central Area to a plug-in defined display. This display can be switched back to the default display by clicking *->Plug-ins -> Main Display -> Default*. The Default menu item is always present.

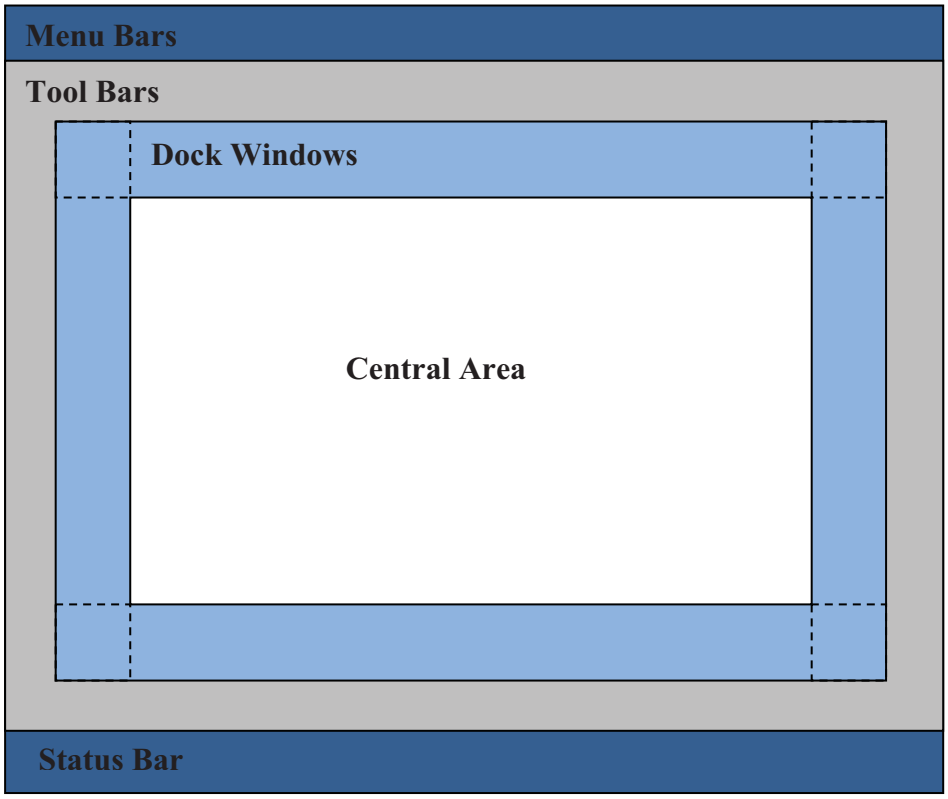

Figure 4.4 Available Dock Window and Toolbar Locations

General information about plug-ins can be found in the *Help->About Plug-ins.* Clicking this menu item launches a dialog displaying a name and description of all loaded plug-ins. This dialog sorts items by the plug-in interface that they implement. See Figure 7.5 for an example of this dialog.

Default GUI items include the *File Menu*, the *Edit Menu*, the *Help Menu*, the *File New/Save Tool Bar*, the *Model Viewer Dock Window*, the *Graphic Control Dock Window*, and the default central graphic display. The *File Menu* launches dialog to load or save files or exit the program. These dialogs support can support any file type given that a file reader/writer plug-in for the file type is provided. The file reader/writer plug-in add it supported file types' file suffix – extensions to the dialogs file filters, such that only the files with the given extension are displayed. The *Edit Menu* is disabled in the MGRT. The *File New/Save Toolbar* is a short cut to the corresponding items in the File Menu.

The *Model View Dock Window* uses the Model View Framework Interfaces, described in Chapter 5, to display a tree view of topology items. This view of the data can be used to select/deselect topology items and to display general data about the topology item via tooltip pop ups and brief status bar messages.

The Graphic Control Dock Window and the default central graphic display are coupled together. Changes to the view in the graphic display reflect in the control window and vice versa. As such, the *Graphic Control Window* is a feasible method for navigating the display. The default display uses OpenGL to display the stored topology. It can be used to select/deselect items, to create a point, or to change the display options. Figures 7.2-7.4 shows these GUI items.

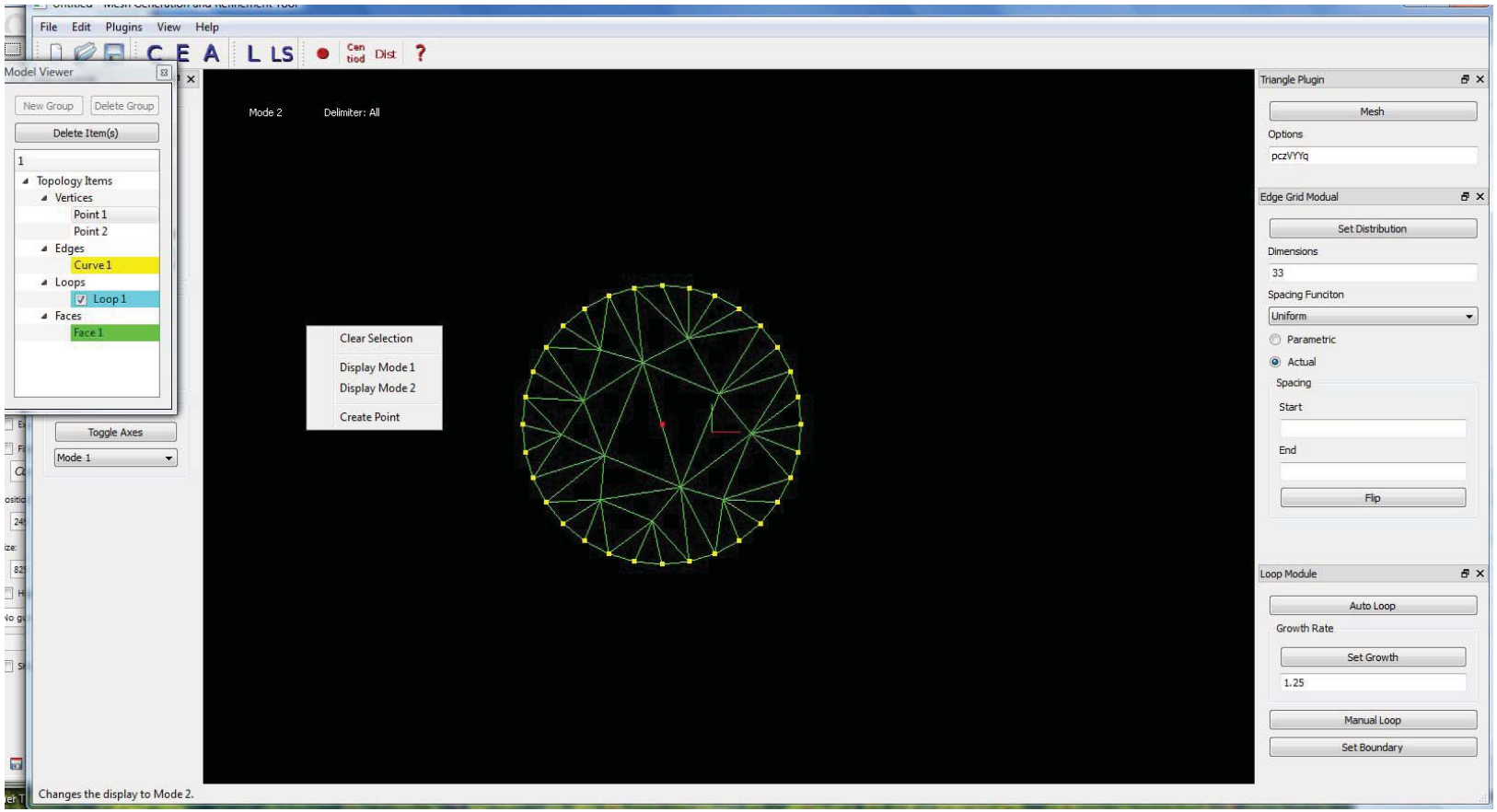

Figure 4.5 Overall View of the MGRT GUI

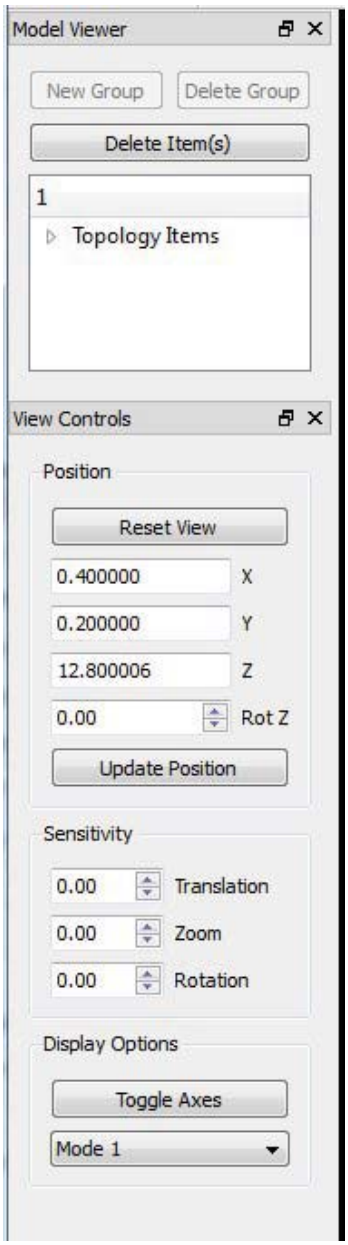

Figure 4.6 MGRT Model Viewer and View Controls

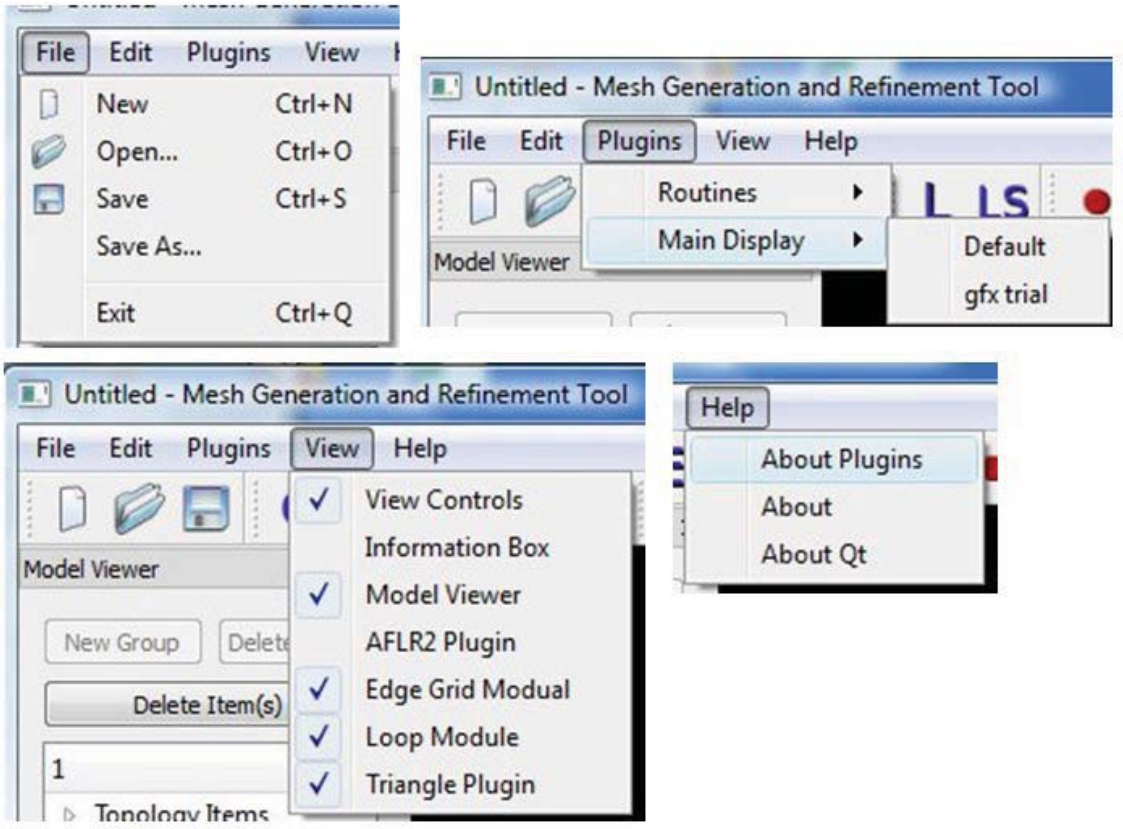

Figure 4.7 MGRT Drop Down Menus

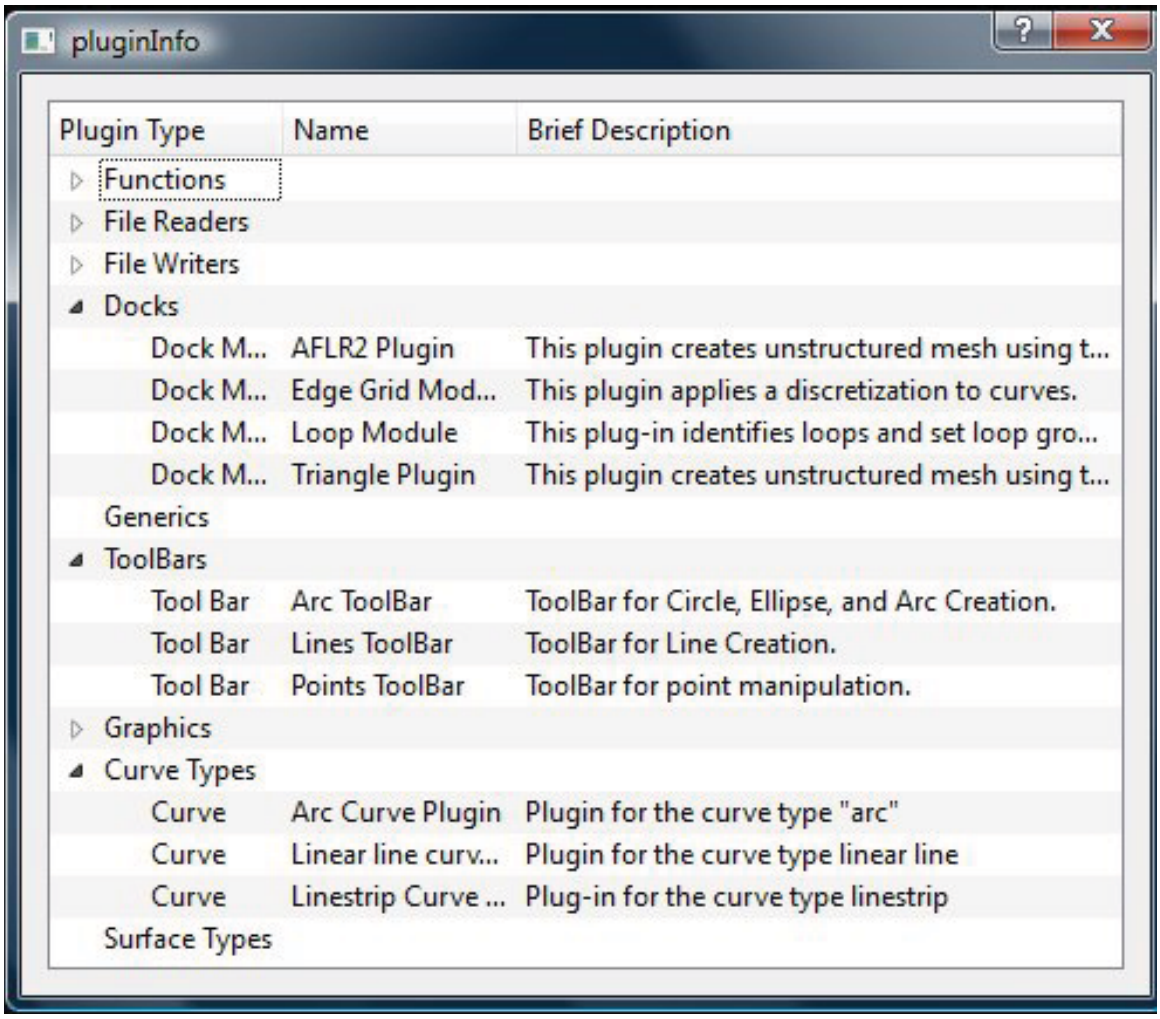

Figure 4.8 Plug-in Information Dialog

CHAPTER V

AN APPLICATION

5.1 INTRODUCTION

For generating a computational grid using MGRT, the primary required data is the specification of the geometrical characteristics of the computational domain (manually or in ASCII format). The study area in this research is Mobile Bay, Alabama, USA (Figure 8.1). The Northern Gulf Institute at Mississippi State University was interested on developing future studies on the hydrodynamics of the estuary located in coastal Alabama. Several numerical models are candidates for estimating the transport of water and sediments. [9]

Although coastline data for United States can be easily downloaded from public domain databases, they are provided in geographical coordinates and data formats that do not allow easy projection of the data. Figure 8.2 illustrates the steps followed for re-projecting the coastline data from geographical coordinates to Universal Transverse Mercator coordinates (UTM). [9].

A gray scale elevation map of mobile bay is shown in Figure 8.3.

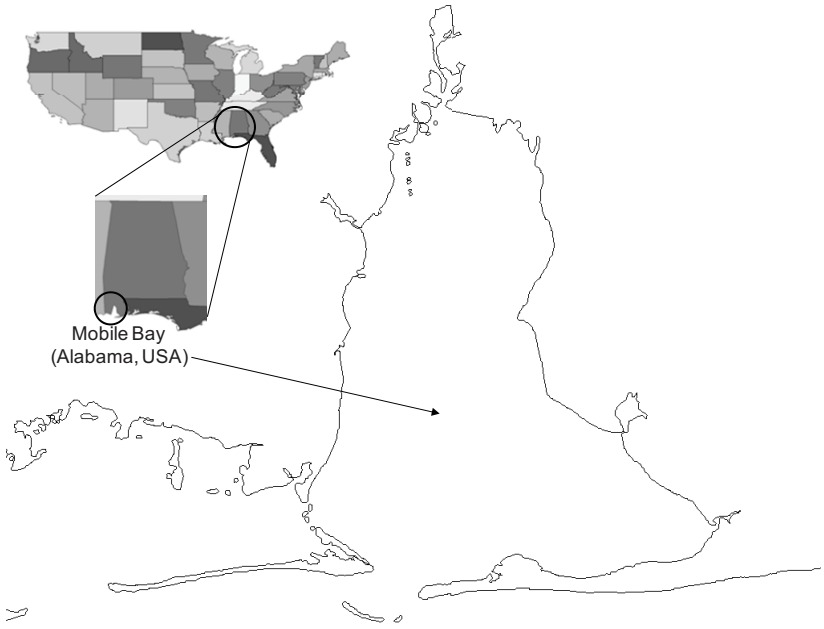

Figure 5.1 Study area. Coastline surrounding Mobile Bay, Alabama, USA

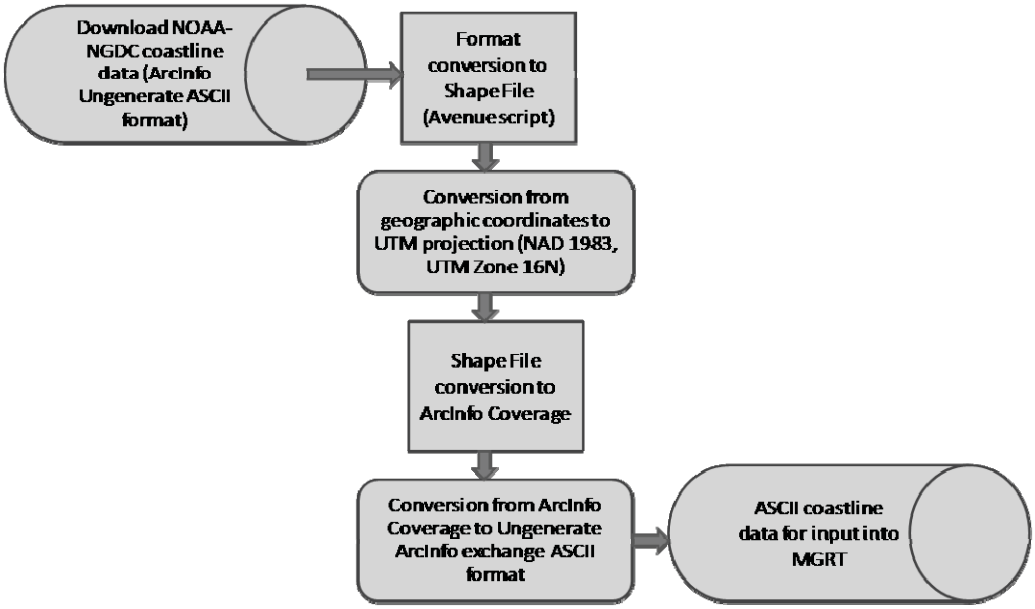

Figure 5.2 Conversion of National Oceanic and Atmospheric Administration-National Geophysical Data Center (NOAA-NGDC) coastline data for MGRT ingestion.

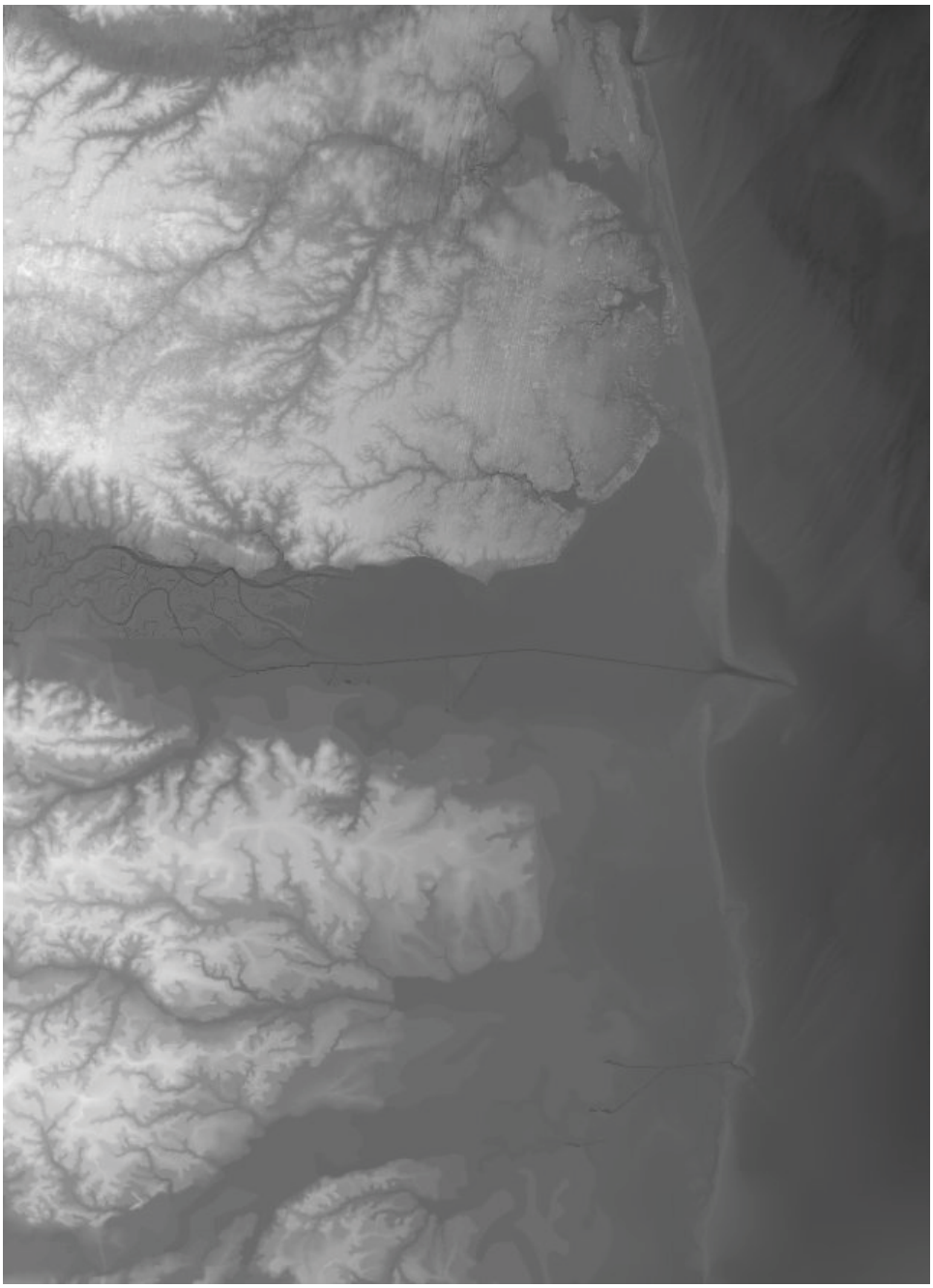

Figure 5.3 Grey Scale of Elevation of Mobile Bay, Alabama (USA)

5.2 2.5D GRID GENERATION

A two and a half dimensional (2.5D) grid is a grid, which is constructed in two dimensions, but has a functional value for a third dimension. In this study the grid is created using a two-dimensional unstructured grid generation algorithm. The third dimensional value must be set in order to satisfy the half dimension. The functional third dimensional values are set based on elevation and bathymetry data of the Mobile Bay. This data is read from a digital elevation map (DEM), a common geographic information system (GIS) data format. The DEM files are interpreted using a C++ library called GDAL [19]. GDAL converts the DEM data into a geo-referenced raster where the data values at each location represent an elevation. A raster can be thought of as an image, and each pixel in the image has an associated elevation. An elevation is assigned to each pixel center in a grid using inverse distance weighting inside a nine point stencil. Inverse distance weighting was chosen as the interpolation method because it is simple, and because it can provide a more accurate elevation for multiple points, which lie in the same raster pixel. See Equation 8.1 for the Inverse distance weighing formula.

$$Z = \frac{\sum_{i=1}^{9}\left(1/d_i\right)\vec{R}_i}{\sum_{i=1}^{9}\left(1/d_i\right)}. \tag{8.1}$$

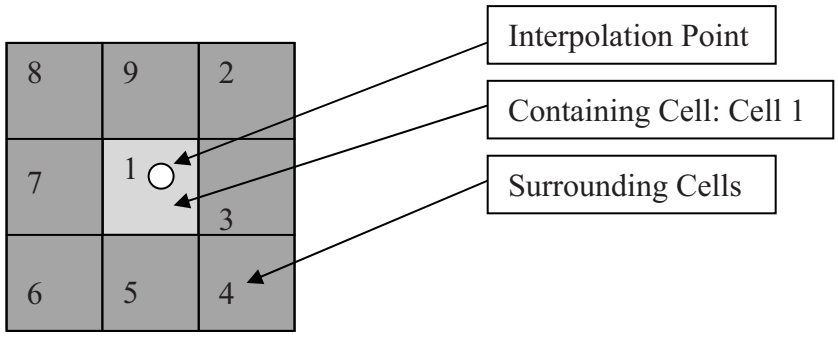

Figure 5.4 Nine Point Stencil Used in Inverse Distance Weighing

The GDAL Grid Interpolation Plug-in provides a usable implementation of Equation 8.1 using a nine-point stencil to interpolate elevation for use in the grid. The plug-in reads elevation data into a regularly spaced grid. The GDAL geographical information systems library is used to read the various file formats that contain elevation data [19]. Each node in this grid has an elevation. To create a 2.5D grid, a two dimensional grid, which lies within the boundaries of the elevation grids, is created. The elevation grid acts as a background grid. As such, the two-dimensional grids vertex locations overlap the cells of the background grid. From the overlapped location, the data in the background grid is interpolated to each vertex. This plug-in implements the *Function* Plug-in Interface, as such it adds a menu item to the Plug-in Menu of the MGRT. This menu item has no GUI and only invokes the interpolation routine.

The AFLR2 plug-in can be used to generate grids of triangular or quadrilateral elements or mix of the two. These grids can be used with the GDAL Interpolation plug-in to create 2.5D grids. This plug-in provides a GUI front end to the command line mesh generator AFLR Marcum [1] [2].

5.3 MOBILE BAY INSTRUCTIONS

To create a mesh for mobile bay, first launch the MGRT. This can be accomplished by running the command *mgrt* located in the bin directory of the MGRT installation directory.

On startup, the MGRT searches the plug-ins folder for possible plug-ins. It attempts to load any file that "looks like" a plug-in by trying to match the objects the libraries found with its know list of interfaces. If the interfaces match, it continues loading the plug-in. The details of which plug-ins were loaded can be found by clicking the *Help Menu --> About Plug-ins*. A dialog pops ups showing all loaded plug-ins by category.

We can load the bay's geometry by clicking *File->open* and selecting the correct file from the file open dialog.

Press *F4* to center the view on the loaded Geometry.

From this point, we can add any additional geometry, such as lines to close the boundary of the bay. Plug-ins for points and lines can be used for this.

A global edge distribution can be set using the *Edge Grid Plug-in*. To set the global distribution, set the spacing type to real spacing, set the spacing function to *Global spacing*, and set your real spacing in the start location. Press *Set Distribution*.

Now, the loops of the mesh must be defined. This is an automated process with the loop module plug-in. A simple press of the auto loop button detects all loops. It is assumed that each edge belongs to only one loop for this procedure to work.

Select the outermost loop. Then press the *Boundary Loop Button* to store it as the boundary loop and all other loops as internal.

Next, we create the mesh by opening the AFLR plug-in, and pressing the *mesh button*. The AFLR plug-in uses the loop topology information to create the mesh. Unwanted internal loops can be eliminated from the input of the AFLR by unchecking it in the tree view of the items.

To set the bathymetry for the mesh, we use the GDAL grid interpolation plug-in. This plug-in used by selecting the mesh and clicking *Plug-ins->Routines->GDAL 2.5D Mesh Interpolation*. A file dialog will appear. Set the input file to the Bathymetry data file for the Bay.

Finally, the mesh can be saved to disk with the 2dm/3dm file I/O plug-in. Select *File->Save*. Set a desired filename. The extension 3dm can be added or the filter can be selected to save the file correctly.

The time required to create the mesh from start to finish is approximately five to ten minutes.

5.4 RESULTING GRID

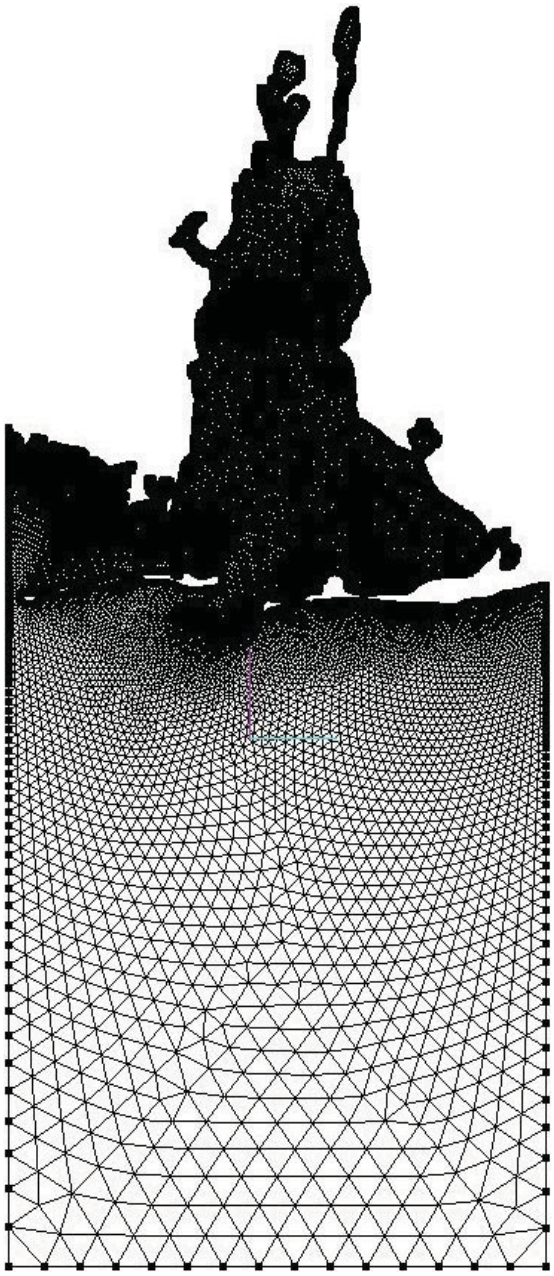

Figure 5.5 Overall View of Mobile Bay Grid

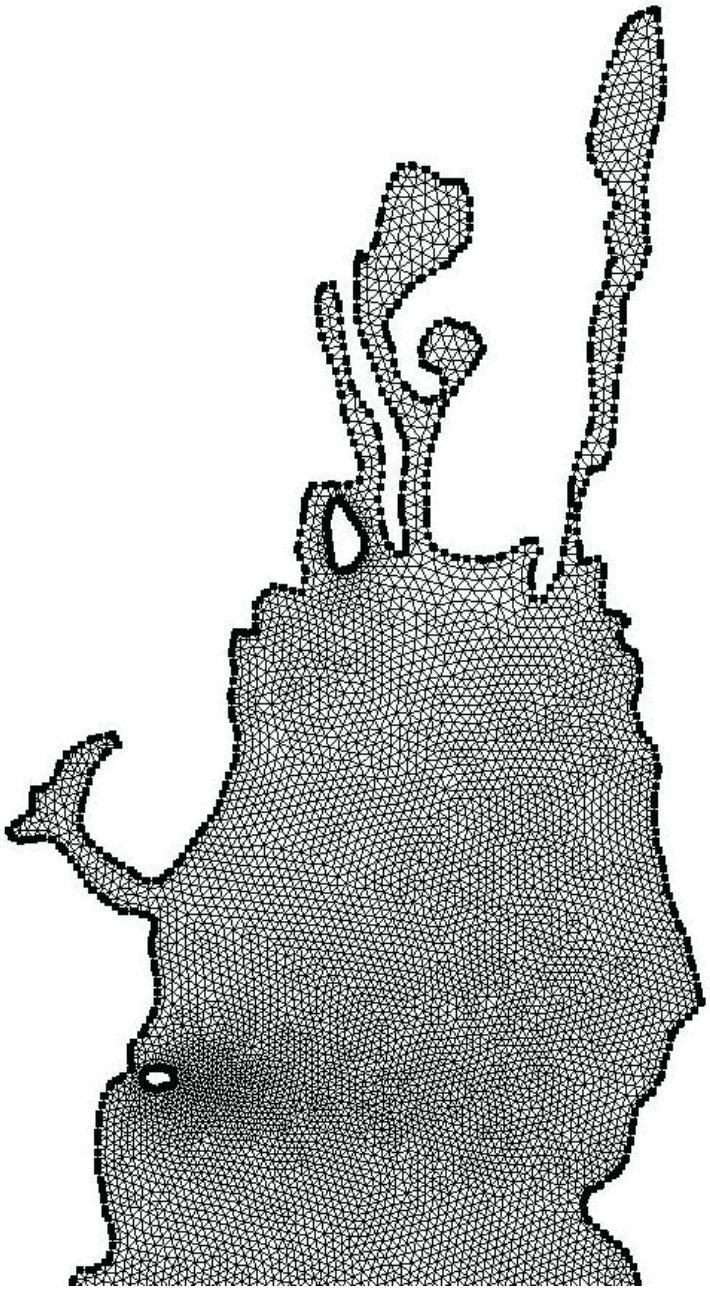

Figure 5.6 Top View of Mobile Bay Grid

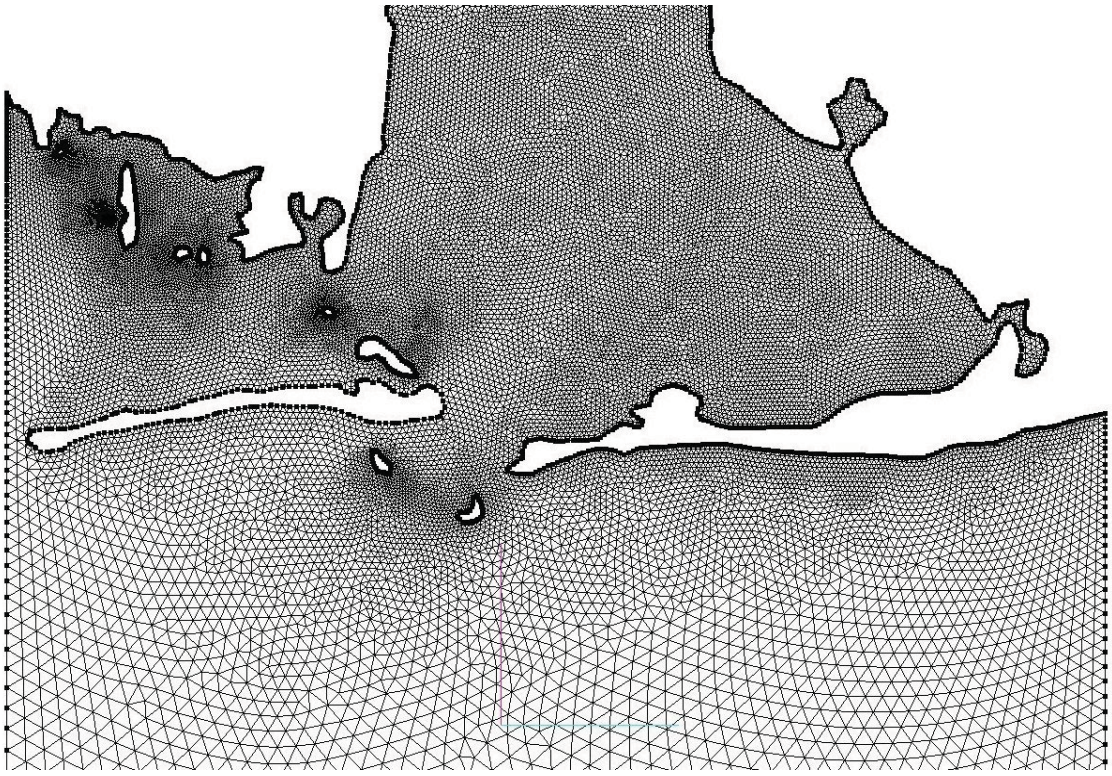

Figure 5.7 Middle View of Mobile Bay Grid

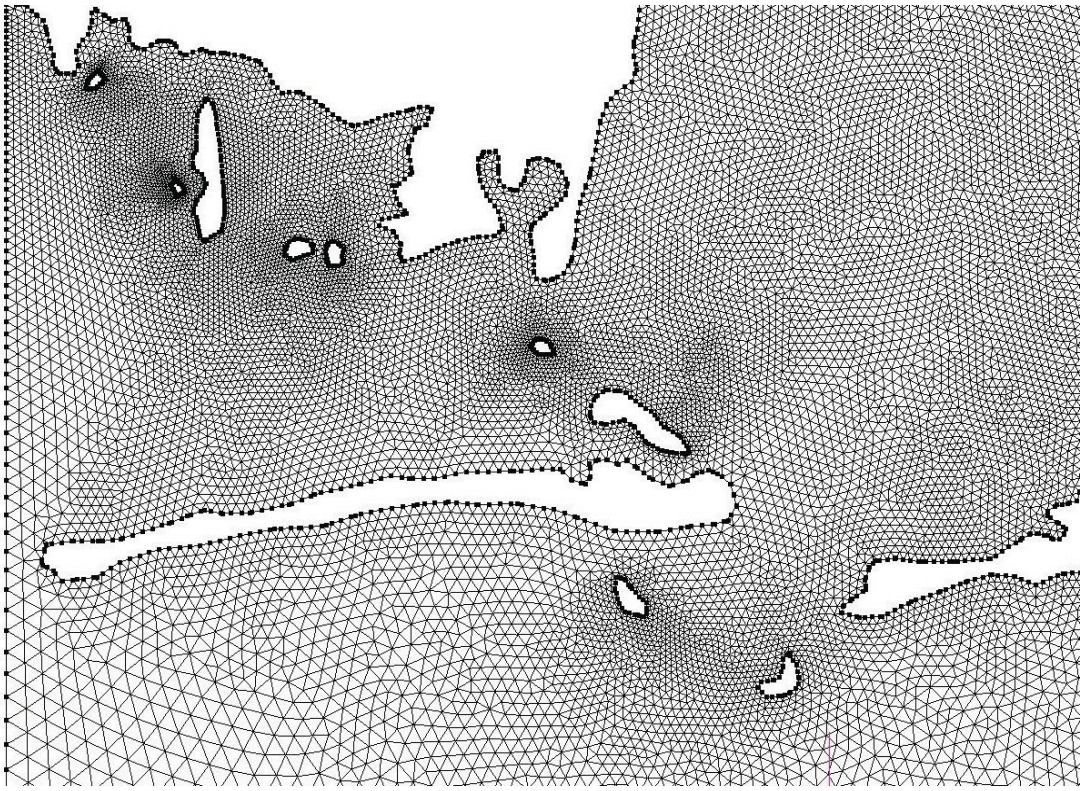

Figure 5.8 Lower Left View of Mobile Bay Grid

Figure 5.9 Lower Right View of Mobile Bay Grid

CHAPTER VI

DEVELOPMENT TOOLS

The MGRT uses the Qt API, developed by Trolltech. Qt is a C++ toolkit for cross platform application development. This means that a single source code can be complied to run natively on Windows, Mac, Linux, and the major UNIX variants. Qt is available in two forms: Open source and Commercial [14]. The two variants are very similar. The major exceptions are that Commercial version has better support, can be closed source, and has Microsoft Visual Studio integration [14]. The MGRT has been coded using the open Source version of Qt. Some notable applications that use Qt are SMS, the KDE desktop environment, Adobe Photoshop Album, and Google Earth [14].

Initially, all of the coding for the MGRT was done in text editors like vim; since, the Microsoft Visual Studio integration package is only available in Commercial versions of Qt. Programming in using a text editor is tedious and time consuming. To simplify coding, Eclipse, an open source integrated development environment, was used in the development of the MGRT [20].

The C/C++ Development Tools (CDT) plug-in was used with Eclipse [20] [25]. The CDT plug-in adds functionality related to C and C++ code development to Eclipse [25]. Additionally, Trolltech provides a freely available integration package for Eclipse. Using the mentioned plug-ins streamlined MGRT development.

Additionally, a Doxygen plug in for Eclipse has been used to document the code. Doxygen is a documentation system for C, C++, and various other languages. The Eclipse plug-in for Doxygen is called Eclox [22]. The Doxygen documentation [21] states that: It can generate an on-line documentation browser (in HTML) and/or an off-line reference manual from a set of documented source files. There is also support for generating output in RTF (MS-Word), PostScript, hyperlinked PDF, compressed HTML, and UNIX man pages. The documentation is extracted directly from the source, which makes it much easier to keep the documentation consistent with the source code. As stated by Doxygen [21]:

> You can configure Doxygen to extract the code structure from undocumented source files. This is very useful to quickly find your way in large source distributions. You can also visualize the relations between the various elements by means of including dependency graphs, inheritance diagrams, and collaboration diagrams, which are all generated automatically.

The GNU C and C++ compilers were used to build the MGRT source code. GNU GCC is available on all UNIX systems and a port, called Mingw32, is available for Window [23][24]. While the compilers across the various platforms are all considered a variant of GNU GCC, the different versions behave differently. For example, compiling libraries on Linux is well documented, but there are different steps and commands that have to be issued in order to compile and link libraries on a windows machine using Mingw32. However, the GNU compiler is still the best suited for building of the MGRT.

It is freely available; Trolltech recommends it; and the Eclipse CDT is setup to use it out of the box.

CHAPTER VII

CONCLUSIONS

The objectives behind this research were to develop a rapid prototyping framework, demonstrate current grid generation technology for the use of creating grids for numerical surface-water modeling, demonstrate that technology in the prototyping framework, and apply the research to a real world application. The MGRT accomplishes all set objectives.

The main component of the MGRT is its topology data structures. The GTM data structure has been modified to be a plug-in based topology data structure, the MGTM. The MGTM abstracts curves and surfaces into definitions and standardized storages containers. This allows new types curves and surface types to be used in the grid generation process.

Furthermore, the MGRT uses the Qt libraries and OpenGL. The Qt libraries are used for the MGRT's graphical user interface and plug-in system. The MGRT effectively uses its plug-in system to extend its topology data structures and core functionality. More specifically, the plug-in system allows for the creation and use of specific tools and routines without revising the MGRT code.

Several development tools and programming libraries were used with much success. These tools and libraries shorten development time significantly. Additionally, several issues concerning their usage were identified and addressed.

A real world application of the MGRT was presented. The result of the application of the tool was the construction of an unstructured mesh of Mobile Bay, Alabama, USA. This application of the tool required the use of several plug-ins, such as the GDAL grid interpolation plug-in and the various File I/O plug-ins. Additionally, a plug-in for the AFLR grid generation code was used.

REFERENCES

[1] D. L. Marcum and N. P. Weatherill, *Unstructured Grid Generation Using Iterative Point Insertion and Local Reconnection*, AIAA Journal, Vol. 33, No. 9, pp 1619-1625, September 1995.

[2] D. L. Marcum, "Unstructured Grid Generation Using Automatic Point Insertion and Local Reconnection," in *The Handbook of Grid Generation,* edited by J.F. Thompson, B. Soni, and N.P. Weatherill, CRC Press, p. 18-1, 1998.

[3] J. F. Thompson, Z. U. Warsi, and C. W. Mastin, *Numerical Grid Generation: Foundations and Applications*. Elsevier North-Holland, Inc.

[4] Vinokur M., 1980, "On One-Dimensional Stretching Functions for Finite-Difference Calculations," Ph.D. Thesis, University of Santa Clara, Santa Clara, CA, USA, October 1980.

[5] Gaither, A., "A Topology Model for Numerical Grid Generation," Proceedings of the 4th International Conference on Numerical Grid Generation in Computational Fluid Dynamics and Related Fields, Swansea, Wales, 1994, pp.247-258.

[6] Gaither, A., Gaither, K., Jean, B., Remotigue, M., and Whitmire, J., et al, "The National Grid Project: A System Overview," Proceedings of Workshop on Surface Modeling, Grid Generation and Related Issues in Computational Fluid Dynamics (CFD) Solutions, NASA CP 3291, 1995, pp 423-446.

[7] Gaither, A., Jean, B., Remotigue, M., and Whitmire, J., "NGP: Defining a Grid Generation Paradigm Based on NURBS and Solid Modeling Topology," Proceedings of the 5th International Conference on Numerical Grid Generation in Computational Field Simulations, Starkville, MS, USA, 1996, pp. 393-402.

[8] Gaither, A., "A Boundary Representation Solid Modeling Data Structure for General Numerical Grid Generation," Master Thesis, Mississippi State University, December 1997.

[9] Aziz, W., Alarcon V. J., McAnally, W., Martin, J., Cartwright, J., 2008. An Application Of The Mesh Generation And Refinement Tool To Mobile Bay, Alabama, USA. Proceedings ICCMSE 2008, American Institute of Physics.

[10] Tannehill, J., Anderson, D., Pletcher, R., "Computational Fluid Mechanics and Heat Transfer, Second Edition," Taylor and Francis. 1997 pp. 679-708.

[11] Thompson J., Soni, B., and Weatherill, N., "Handbook of Grid Generation," CRC-Press. 1st Ed. 1998.

[12] Triangle. A Two-Dimensional Quality Mesh Generator and Delaunay Triangulator [internet]. University of California at Berkeley: Computer Science Division; [cited 2008 January 23]. Available from: http://www.cs.cmu.edu/~quake/triangle.html

[13] Zundel, A. K., (2005). "Surface-water Modeling System reference manual – Version 9.0," Brigham Young University Environmental Modeling Research Laboratory, Provo, UT.

[14] Qt-Trolltech [internet]. RedWood City, California: US Office; [cited 2008 January 23]. Available from: http://trolltech.com/products/qt

[15] Qt-4.3-Whitepaper. Trolltech ASA. 2007.

[16] Blanchette J., Summerfield M., "C++ GUI Programming with Qt4", 2nd Ed., Prentice Hall PTR. 2008.

[17] Remotigue, M., "Structured Grid Technology to enable flow simulation in an integrated system environment," Ph D. Dissertation, Mississippi State University, December 1999.

[18] Marivplis, D., "Unstructured mesh generation and adaptivity," tech. rep., ICASE 1995. ICASE Report No. 95-26.

[19] GDAL. GDAL: GDAL – Geospatial Data Abstraction Library [internet]. [cited 2008, October 15]. Available from: http://www.gdal.org.

[20] Eclipse [internet]. Portland, Oregon: Portland Office; [cited 2008 January 23]. Available from: http://www.eclipse.org

[21] Doxygen [internet]. Dimitri van Heesch; [cited 2008 January 23]. Available from: http://www.stack.nl/~dimitri/doxygen/

[22] Eclox [internet]. [cited 2008 January 23]. Available from: http://home.gna.org/eclox/

[23] GCC, the GNU Compiler Collection [internet]. [cited 2008 January 25]. Available from: http://gcc.gnu.org/

[24] MinGW [internet]. [cited 2008 January 25]. Available from: http://www.mingw.org/

[25] Eclipse C/C++ Development Toolkit [internet]. [cited 2008 January 25]. Available from: http://www.eclipse.org/cdt/

[26] SolidMesh: 3D User's Manual [internet]. [cited 2010 Feburary 15]. http://www.simcenter.msstate.edu/docs/solidmesh/

APPENDIX A

PLUG-IN API

This appendix introduces a few of the C++ data structures needed to create a plug-in. To fully understand this introduction, it is expected the reader be familiar with object-oriented programming, C++, programming with templates, and the Qt API. The Qt API is documented in [14] [15] [16].

The class introductions sections highlight the public members of the important classes used by the plug-ins and explains their use. Private members are not discussed; since, programmers cannot access them.

The class name introduces each section. If the class is a derived class, its base class is given, next. Then, a list of all the class's public members is presented. After this list, the section ends with detailed descriptions of each public member.

Class Reference: curve_data

Brief Description

The curve_data class provides a generic data object for a curve as described by the MGTM data structure. This generic data object is used to provide data for all curves. The data is not set in any particular fashion. It is left up to the factory methods and/or definition objects to set and use the data. All of the data needed to define a curve, except for a function object, are in this data structure. Keeping this information centralized simplifies file I/O and curve construction from saved data.

Public Members

- `curve_data(std::string Type = "un_set");`
- `curve_data(const curve_data& that);`
- `curve_data& operator = (const curve_data& that);`
- `virtual ~curve_data();`
- `template<typename OutputIter> void getMvInts(OutputIter out) const;`
- `template<typename FwdIter> void setMvInts(FwdIter begin, FwdIter end);`
- `template<typename OutputIter> void getMvDoubles(OutputIter out) const;`
- `template<typename FwdIter> void setMvDoubles(FwdIter begin, FwdIter end);`
- `std::string getMvType() const;`
- `void setMvType(std::string Type);`
- `unsigned getId() const;`
- `void setId(unsigned id);`

Detailed Description

```
curve_data(std::string Type = "un_set");
```

The default constructor take a string with the name of the definition associated with the data. If a name is not provided, the name “un_set” is used.

```
curve_data(const curve_data& that);
```

The copy constructor clones the data in a given curve data structure.

```
curve_data& operator = (const curve_data& that);
```

The assignment operator clones the data in a given curve data structure.

```
virtual ~curve_data();
```

The destructor destroys any allocated memory.

```
template<typename OutputIter> void getMvInts(OutputIter out) const;
```

The template member ***getMvInts*** retrieves all of the stored integer data. The output iterator typically used in MGRT is a ***std::back_inserter***. The back inserter appends data to the end of a STL compliant container.

```
template<typename FwdIter> void setMvInts(FwdIter begin, FwdIter end);
```

The template member ***setMvInts*** inserts integer data using input iterators. In addition, calling ***setMvInts*** clears any stored integers data.

```
template<typename OutputIter> void getMvDoubles(OutputIter out) const;
```

The template member ***getMvDoubles*** retrieves all of the stored double precision float data. The output iterator typically used in MGRT is a ***std::back_inserter***. The back inserter appends data to the end of a STL compliant container.

```
template<typename FwdIter> void setMvDoubles(FwdIter begin, FwdIter
end);
```

The template member ***setMvDoubles*** inserts integer data using input iterators. In addition, calling ***setMvDoubles*** clears any stored double precision float data.

```
std::string getMvType() const;
```

The member function ***getMvType*** returns the name of the data's definition.

```
void setMvType(std::string Type);
```

The member function setMvType set the name of the data's definition.

```
unsigned getId() const;
```

The member function ***getId*** returns a unique identification number associated with the data.

```
void setId(unsigned  id);
```

The member function ***setId*** sets a unique identification number associated with the data.

Class Reference: curve_definition

Brief Description

The curve definition class is an abstract interface for a defining curve definition objects. This class is purely virtual. An implementation of the interface must provide concrete implementations of all of the methods.

Public Members:

- `virtual int para_to_real(const double para_coords,`
- `curve_data cdata, double* real_coords) = 0;`
- `virtual int real_to_para(double* real_coords,`
- `curve_data cdata, double* para_coords) = 0;`
- `virtual std::string type() const = 0;`

Detailed Description

```
virtual int para_to_real(const double para_coords, curve_data cdata,
double* real_coords) = 0;
```

The member functions ***para_to_real*** transforms a parametric input, the variable ***para_coords***, and sets the values of ***real_coords*** to the corresponding real space coordinates. The variable para_coords is expected to be a value in the interval [0,1]. The array ***real_coords*** is an array of size three. The return value of this function is expected to be one if the operation was successful, and zero if otherwise.

```
virtual int real_to_para(double* real_coords, curve_data cdata, double*
para_coords) = 0;
```

The member function ***real_to_para*** is not used in the MGRT and is reserved for future use. As its name implies, ***real_to_para*** was included to convert a real space coordinate on the curve to its parametric equivalent.

```
virtual std::string type() const = 0;
```

The member function ***type*** returns a string identifying the name of the definition.

Class Reference: surface_data

Brief Description

The surface_data class provides a generic data object for a surface as described by the MGTM data structure. This generic data object is used to provide data for all curves. The data is not set in any particular fashion. It is left up to the factory methods and/or definition objects to set and use the data. All of the data needed to define a curve, except for a function object, are in this data structure. Keeping this information centralized simplifies file I/O and curve construction from saved data.

Public Members

- `surface_data(std::string Type = "un_set");`
- `surface_data (const surface & that);`
- `surface_data& operator = (const surface_data& that);`
- `virtual ~surface_data();`
- `template<typename OutputIter> void getMvInts(OutputIter out) const;`
- `template<typename FwdIter> void setMvInts(FwdIter begin, FwdIter end);`
- `template<typename OutputIter> void getMvDoubles(OutputIter out) const;`
- `template<typename FwdIter> void setMvDoubles(FwdIter begin, FwdIter end);`
- `std::string getMvType() const;`
- `void setMvType(std::string Type);`
- `unsigned getId() const;`
- `void setId(unsigned id);`

Detailed Description

```
surface_data(std::string Type = "un_set");
```

The default constructor take a string with the name of the definition associated with the data. If a name is not provided, the name “un_set” is used.

```
surface_data(const curve_data& that);
```

The copy constructor clones the data in a given surface data structure.

```
surface_data& operator = (const surface_data& that);
```

The assignment operator clones the data in a given surface data structure.

```
virtual ~surface_data();
```

The destructor destroys any allocated memory.

```
template<typename OutputIter> void getMvInts(OutputIter out) const;
```

The template member ***getMvInts*** retrieves all of the stored integer data. The output iterator typically used in MGRT is a ***std::back_inserter***. The back inserter appends data to the end of a STL compliant container.

```
template<typename FwdIter> void setMvInts(FwdIter begin, FwdIter end);
```

The template member ***setMvInts*** inserts integer data using input iterators. In addition, calling ***setMvInts*** clears any stored integers data.

```
template<typename OutputIter> void getMvDoubles(OutputIter out) const;
```

The template member ***getMvDoubles*** retrieves all of the stored double precision float data. The output iterator typically used in MGRT is a ***std::back_inserter***. The back inserter appends data to the end of a STL compliant container.

```
template<typename FwdIter> void setMvDoubles(FwdIter begin, FwdIter
end);
```

The template member ***setMvDoubles*** inserts integer data using input iterators. In addition, calling ***setMvDoubles*** clears any stored double precision float data.

```
std::string getMvType() const;
```

The member function ***getMvType*** returns the name of the data's definition.

```
void setMvType(std::string Type);
```

The member function ***setMvType*** set the name of the data's definition.

```
unsigned getId() const;
```

The member function ***getId*** returns a unique identification number associated with the data.

```
void setId(unsigned  id);
```

The member function ***setId*** sets a unique identification number associated with the data.

Class Reference: surface_definition

Brief Description

Like the curve definition class, the surface definition class is an abstract interface for a defining surface definition objects. This class is purely virtual. An implementation of the interface must provide concrete implementations of all of the methods.

Public Members

- `virtual int para_to_real(const double* para_coords, surface_data sdata, double* real_coords) = 0;`
- `virtual int real_to_para(double* real_coords, surface_data sdata, double* para_coords) = 0;`
- `virtual std::string type() const = 0;`
- `virtual int para_to_real(const double* para_coords, surface_data sdata, double* para_coords) = 0;`

Detailed Description

```
virtual int para_to_real(const double* para_coords, surface_data sdata,
double* real_coords) = 0;
```

The member functions ***para_to_real*** transforms a parametric input, the array ***para_coords***, and sets the values of ***real_coords*** to the corresponding real space coordinates. The array ***para_coords*** is an array of size two, and the array ***real_coords*** is an array of size three. The array ***para_coords*** is the parametric coordinates (u,v), which is defined on the interval ([0,1],[0,1]). The return value of ***para_to_real*** is expected to be one if the operation was successful, and zero if otherwise.

```
virtual int real_to_para(double* real_coords, surface_data sdata,
double* para_coords) = 0;
```

The member function ***real_to_para*** is not used in the MGRT and is reserved for future use. The member function ***real_to_para*** was included to convert a real space coordinate on the surface to its parametric equivalent.

```
virtual std::string type() const = 0;
```

The member function ***type*** returns a string identifying the name of the definition.

Class Reference: triplet

Brief Description

Inherits: **template<typename T1 typename T2> std::pair<T1,T2>**

Template Types: **template<typename T1, typename T2, typename T3>**

The triplet template class holds three objects of arbitrary type. This class is sub-classed from std::pair to allow for simple low-level conversions to std::pair.

Public Members

- **typedef T1** first_type;
- **typedef T2** second_type;
- **typedef T3** third_type;
- **first_type** *first*;
- **second_type** *second*;
- **third_type** *third*;
- **triplet**();
- **triplet**(**const T1**& *x*, **const T2** &*y*, **const T3** &*z*);
- **template**<**typename U**, **typename V**, **typename W**> **triplet**(**const** triplet<**U**, **V**, **W**> &t);

Detailed Description

typedef T1 first_type;

First_type is a typedef for the first templated type. The typedef is a part of the STL API.

typedef T2 second_type

Second type is a typedef for the second tempated type. This typedef is a part of the STL API.

typedef T3 third_type;

Thrid_type is a typedef for the third template type. The typedef is used to follow STL Reference style.

first_type *first*;

The member ***first*** is the stored first piece of data of type ***first_type***.

second_type *second*;

The member second is the stored second piece of data of type ***second_type***.

```
third_type third;
```

The member third is the stored third piece of data (which the class is named for) of type ***third_type***.

```
triplet();
```

The default constructor sets the values of the stored three stored members using their default constructors.

```
triplet(const T1& x, const T2 &y, const T3 &z);
```

The copy constructor copies data from a triplet with the same template arguments.

```
template<typename U, typename V, typename W> triplet(const triplet<U,
V, W> &t);
```

This copy constructor copies data from a triplet with template types convertible to the current triplet's template types.

Class Reference: WriteToDB

Brief Description

This class provides low level write privileges to a database object. Also see the ***ReadFromDB*** class, ***InteractiveDBObject*** class.

Public Members

- `WriteToDB(database* pDataBase, curveManager* cManager = NULL, surfaceManager* sManager = NULL );`
- `WriteToDB(const WriteToDB &that);`
- `WriteToDB& operator = (const WriteToDB &that);`
- `virtual ~WriteToDB();`
- `template<typename FwdIter> void write(FwdIter begin, FwdIter end);`
- `template<typename FwdIter, typename Inserter>`
- `void encapuslateBulk(FwdIter begin, FwdIter end, Inserter encappedRec);`
- `void setCurveManager(curveManager* cManager);`
- `curveManager* getCurveManager() const;`
- `void setSurfaceManager(surfaceManager* sManager);`
- `surfaceManager* getSurfaceManager() const;`

Signals:

- `void drawRequest();`
- `void sendText( const QString & );`
- `void updateItem( const std::pair<int,int>& );`
- `void itemUpdated( const std::pair<int,int> & );`

Detailed Description

```
WriteToDB(database* pDataBase, curveManager* cManager = NULL,
surfaceManager* sManager = NULL );
```

Constructs a ***WriteToDB*** which uses a given database, curve manager, and surface manager. If the curve or surface manager is *NULL* the ***WriteToDB*** object cannot provide the plug-in using the object the *NULL* manager. The MGRT typically handles the construction of this type of object.

```
WriteToDB(const WriteToDB &that);
```

Constructs a ***WriteToDB*** that is a copy of a given ***WriteToDB***.

```
WriteToDB& operator = (const WriteToDB &that);
```

Clones a ***WriteToDB*** from a given ***WriteToDB***.

```
virtual ~WriteToDB();
```

Destroys the ***WriteToDB*** and frees any allocated resources.

```
template<typename FwdIter> void write(FwdIter begin, FwdIter end);
```

Expects FwdIter to be an iterators with type std::pair<int, void*>. The integer value in the pair an enumerated type defining type of the pointer. The void pointer is a pointer pointing to stored data. All pointers can be or are implicitly casted to void a pointer. It is best to use the encapsulate functions to put data in the correct format. Refer to the ***encapsulateBulk*** member function (following this one) and the encapsulate functions for methods to encapsulate a pointer.

```
void encapuslateBulk(FwdIter begin, FwdIter end, Inserter encappedRec);
```

This function converts new items to a std::pair<int,void*>. The input to this function is defined by the iterators ***begin*** and ***end***, and their type, ***FwdIter***, is expected to be a pointer to a MGTM data type. The type ***Inserter*** is expected to be an inserter iterator with type std::pair<int,void*>, where the integer is the enumerated type, and the void pointer is the pointer to a topology item. This function converts an inputted range of topology items to a std::pair<int,void*> and stores the results in the output iterator ***encappedRec***. This function is useful when converting newly allocated topology items the format expected by the WriteToDB ***write*** function. Note: this function will convert only one type at a time.

```
void setCurveManager(curveManager* cManager);
```

This function sets the ***curveManager*** used to build curves from curve definitions.

```
curveManager* getCurveManager() const;
```

Retrieves the curveManager used to build curve from curve definitions and curve data.

```
void setSurfaceManager(surfaceManager* sManager);
```

This function sets the surfaceManager used to build surface from surface definitions.

```
surfaceManager* getSurfaceManager() const;
```

Retrieves the ***surfaceManager*** used to build surface from surface definitions and surface data.

```
void drawRequest();
```

Emitting this signal notifies MGRT to refresh its graphics.

```
void sendText( const QString & );
```

Emitting this signal sends a text to be printed out via the MGRT information box.

```
void updateItem( const std::pair<int,int>& );
```

This outgoing signal takes the argument std::pair<int,int>, where the first argument is the enumeration of the type, and the second is the identification number of the item assigned to it by MGRT. Emitting this signal will notify MGRT that the item has been modified and to notify all other plug-ins and components of the change.

```
void itemUpdated( const std::pair<int,int> & );
```

This is an incoming signal informing the component using ***WriteToDB*** that the item described in the std::pair<int,int> has been updated. In the argument type ***std::pair<int,int>,*** the first integer is the enumeration of the type, and the second is the identification number of the item assigned to it by MGRT.

Class Reference: ReadFromDB

Brief Description

This class provides low level reading privileges from a database object. Also see the WriteToDB class and the InteractiveDBObject class.

Public Members:

- `ReadFromDB( database* pDataBase );`
- `ReadFromDB( const ReadFromDB &that );`
- `ReadFromDB& operator =( const ReadFromDB &that );`
- `virtual ~ReadFromDB()`
- `template<typename Insrtr>`
- `void readOnlySelected( Insrtr inserter, const int type );`
- `template<typename Insrtr> void read( Insrtr inserter, const int type );`

Signals:

- `void drawRequest();`
- `void sendText( const QString & );`
- `void updateItem( const std::pair<int,int>& );`
- `void itemUpdated( const std::pair<int,int> & );`

Detailed Description

`ReadFromDB( database* pDataBase );`

Constructs a ***ReadFromDB,*** which uses a given database. The MGRT typically handles the construction of this type of object.

`ReadFromDB( const ReadFromDB &that );`

Constructs a ***ReadFromDB*** that is a copy of a given ***ReadFromDB***.

`ReadFromDB& operator =( const ReadFromDB &that );`

Clones a ***ReadFromDB*** from a given ***ReadFromDB***.

`virtual ~ReadFromDB()`

Destroys the ***ReadFormDB*** and frees any allocated resources.

`template<typename Insrtr> void readOnlySelected( Insrtr inserter, const int type );`

Retrieves only items of the given *type,* which have been selected. The template type ***Insrtr*** is expected to be an iterator with a type ***triplet<int,int,void*>***. The argument *type* is an enumeration of the objects type that is being retrieved. The first integer in the triplet is an enumeration of the objects type. The second integer is the objects identification number assigned to the object by MGRT. The void pointer is the objects pointer. This pointer can be cast to the correct type using the enumeration as a guide to which type the pointer should be casted.

```
template<typename Insrtr> void read( Insrtr inserter, const int type );
```

Retrieves all items of the given *type*. The template type ***Insrtr*** is expected to be an iterator with a type ***triplet<int,int,void*>***. The argument *type* is an enumeration of the objects type that is being retrieved. The first integer in the triplet is an enumeration of the objects type. The second integer is the objects identification number assigned to the object by MGRT. The void pointer is the objects pointer. This pointer can be cast to the correct type using the enumeration as a guide to which type the pointer should be casted.

```
void drawRequest();
```

Emitting this signal notifies MGRT to refresh its graphics.

```
void sendText( const QString & );
```

Emitting this signal sends a text to be printed out via the MGRT information box.

```
void updateItem( const std::pair<int,int>& );
```

This outgoing signal takes the argument std::pair<int,int>, where the first integer is the enumeration of the type, and the second is the identification number of the item assigned to it by MGRT. Emitting this signal will notify MGRT that the item has been modified and to notify all other plug-ins and components of the change.

```
void itemUpdated( const std::pair<int,int> & );
```

This is an incoming signal informing the component using a ReadFromDB object that the item described in the std::pair<int,int> has been updated. In the argument type std::pair<int,int>, the first integer is the enumeration of the type, and the second is the identification number of the item assigned to it by MGRT.

Class Reference: InteractiveDBObject

Brief Description

This class is an improved combination of the WriteToDB and ReadToDB classes. This class can perform all of the operations of WriteToDB and ReadToDB. In addition to this, the InteractiveDBObject adds more functionality through Qt signals and slots. This class can be connected to plug-in code to notify the database of changes to stored items, or the plug-in code can be notified when other components changes stored items.

Public Members:

- `InteractiveDBObject( database* pDataBase, curveManager* cManager = NULL, surfaceManager* sManager = NULL );`
- `InteractiveDBObject( const InteractiveDBObject &that );`
- `InteractiveDBObject& operator =( const InteractiveDBObject &that );`
- `virtual ~InteractiveDBObject();`
- `template<typename Insrtr>`
- `void readOnlySelected( Insrtr inserter, const int type );`
- `template<typename Insrtr> void read( Insrtr inserter, const int type );`
- `Triplet<int,int,void*> retrieveItem( const record2 rec2 );`
- `template<typename FwdIter, typename Inserter>`
- `void write(FwdIter begin,FwdIter end, Inserter outputRecords);`
- `template<typename FwdIter, typename Inserter>`
- `void encapuslateBulk(FwdIter begin, FwdIter end, Inserter encappedRec);`
- `void setCurveManager(curveManager* cManager);`
- `curveManager* getCurveManager() const;`
- `void setSurfaceManager(surfaceManager* sManager);`
- `surfaceManager* getSurfaceManager() const;`

Signals:

- `void sendText( const QString & );`
- `void select_target( const std::pair<int,int> & );`
- `void select_source( const std::pair<int,int> & );`
- `void deselect_target( const std::pair<int,int> & );`
- `void deselect_source( const std::pair<int,int> & );`

- `void selectMultiple_target( const std::set< std::pair<int,int> > & );`
- `void selectMultiple_source( const std::set< std::pair<int,int> > & );`
- `void deselectMultiple_target( const std::set< std::pair<int,int> > & );`
- `void deselectMultiple_source( const std::set< std::pair<int,int> > & );`
- `void clearSelection_target( );`
- `void clearSelection_source( );`
- `void newAdditions_target( );`
- `void newDeletions_target( );`
- `void update_target( const std::pair<int,int> & );`
- `void update_source( const std::pair<int,int> & );`
- `void refresh_target();`
- `void refresh_source();`

Detailed Description

```
InteractiveDBObject( database* pDataBase, curveManager* cManager =
NULL, surfaceManager* sManager = NULL );
```

Constructs an **InteractiveDBObject** which uses a given database, curve manager, and surface manager. If the curve or surface manager is NULL the **InteractiveDBObject** object cannot provide the plug-in using the object the NULL manager. The MGRT typically handles the construction of this type of object.

```
InteractiveDBObject ( const InteractiveDBObject &that );
```

Constructs an ***InteractiveDBObject*** that is a copy of a given ***InteractiveDBObject***.

```
InteractiveDBObject & operator =( const InteractiveDBObject &that );
```

Clones an **InteractiveDBObject** from a given **InteractiveDBObject.**

```
virtual ~ InteractiveDBObject ()
```

Destroys the ***InteractiveDBObject*** and frees any allocated resources.

```
template<typename Insrtr> void readOnlySelected( Insrtr inserter, const
int type );
```

Retrieves only items of the given *type,* which have been selected. The template type ***Insrtr*** is expected to be an iterator with a type ***triplet<int,int,void*>***. The argument *type* is an enumeration of the objects type that is being retrieved. The first integer in the triplet

is an enumeration of the objects type. The second integer is the objects identification number assigned to the object by MGRT. The void pointer is the objects pointer. This pointer can be cast to the correct type using the enumeration as a guide to which type the pointer should be casted.

```
template<typename Insrtr> void read( Insrtr inserter, const int type );
```

Retrieves all items of the given *type*. The template type ***Insrtr*** is expected to be an iterator with a type ***triplet<int,int,void*>***. The argument *type* is an enumeration of the objects type that is being retrieved. The first integer in the triplet is an enumeration of the objects type. The second integer is the objects identification number assigned to the object by MGRT. The void pointer is the objects pointer. This pointer can be cast to the correct type using the enumeration as a guide to which type the pointer should be casted.

```
template<typename FwdIter> void write(FwdIter begin, FwdIter end);
```

Expects ***FwdIter*** to be an iterators with type std::pair<int, void*>. The integer value in the pair an enumerated type defining type of the pointer. The void pointer is a pointer pointing to stored data. All pointers can be or are implicitly casted to void a pointer. It is best to use the encapsulate functions to put data in the correct format. Refer to the ***encapsulateBulk*** member function (following this one) and the encapsulate functions for methods to encapsulate a pointer.

```
void encapuslateBulk(FwdIter begin, FwdIter end, Inserter encappedRec);
```

This function converts new items to a ***std::pair<int,void*>.*** The input to this function is defined by the iterators ***begin*** and ***end***, and their type, ***FwdIter***, is expected to be a pointer to a MGTM data type. The type ***Inserter*** is expected to be an inserter iterator with type std::pair<int,void*>, where the integer is the enumerated type, and the void pointer is the pointer to a topology item. This function converts an inputted range of topology items to a std::pair<int,void*> and stores the results in the output iterator ***encappedRec***. This function is useful when converting newly allocated topology items the format expected by the WriteToDB ***write*** function. Note: this function will convert only one type at a time.

```
void setCurveManager(curveManager* cManager);
```

This function sets the curveManager used to build curves from curve definitions.

```
curveManager* getCurveManager() const;
```

Retrieves the ***curveManager*** used to build curve from curve definitions and curve data.

```
void setSurfaceManager(surfaceManager* sManager);
```

This function sets the ***surfaceManager*** used to build surface from surface definitions.

```
surfaceManager* getSurfaceManager() const;
```

Retrieves the ***surfaceManager*** used to build surface from surface definitions and surface data.

```
void select_target( const std::pair<int,int> & );
```

When this signal is received, it informs the component using the ***InteractiveDBObject*** that an item has been selected. The selected item is identified by a ***std::pair<int,int>***. The first integer is the item's enumerated type and the second is the item's identification number assigned to it by the MGRT.

```
void select_source( const std::pair<int,int> & );
```

This signal should be emitted when a single item has been selected. The selected item is identified by a ***std::pair<int,int>***. The first integer is the item's enumerated type and the second is the item's identification number assigned to it by the MGRT.

```
void deselect_target( const std::pair<int,int> & );
```

When this signal is received, it informs the component using the InteractiveDBObject that an item has been deselected. The deselected item is identified by a std::pair<int,int>. The first integer is the item's enumerated type and the second is the item's identification number assigned to it by the MGRT.

```
void deselect_source( const std::pair<int,int> & );
```

This signal should be emitted when a single item has been deselected. The deselected item is identified by a std::pair<int,int>. The first integer is the item's enumerated type and the second is the item's identification number assigned to it by the MGRT.

```
void selectMultiple_target( const std::set< std::pair<int,int> > & );
```

This signal is received. It informs the component using the InteractiveDBObject that multiple items have been selected. The selected items are in a std::set of std::pairs.

```
void selectMultiple_source( const std::set< std::pair<int,int> > & );
```

This signal should be emitted when a multiple items has been deselected. The deselected items are identified by a std::pair<int,int>. The first integer is the item's enumerated type and the second is the item's identification number assigned to it by the MGRT.

```
void deselectMultiple_target( const std::set< std::pair<int,int> > & );
```

This signal is received. It informs the component using the InteractiveDBObject that multiple items have been deselected. The deselected items are in a std::set of std::pairs.

```
void deselectMultiple_source( const std::set< std::pair<int,int> > & );
```

This signal should be emitted when a multiple item has been deselected. The deselected items are identified by a std::pair<int,int>. The first integer is the item's enumerated type and the second is the item's identification number assigned to it by the MGRT.

```
void clearSelection_target( );
```

This signal is received. It informs the component that the MGRT selection has been cleared.

```
void clearSelection_source( );
```

The signal should be emitted when the MGRT selection should be cleared.

```
void newAdditions_target( );
```

This signal is received, when new items have been added to the MGRT storage from another component. This signal can be used to inform the component that new data is available.

```
void newDeletions_target( );
```

This signal is received, when items have been deleted from the MGRT storage.

```
void update_target( const std::pair<int,int> & );
```

This is an incoming signal informing the component using ***WriteToDB*** that the item described in the std::pair<int,int> has been updated.

```
void update_source( const std::pair<int,int> & );
```

This outgoing signal takes the argument ***std::pair<int,int>,*** where the first argument is the enumeration of the type, and the second is the identification number of the item assigned to it by MGRT. Emitting this signal will notify MGRT that the item has been modified and to notify all other plug-ins and components of the change.

```
void refresh_target();
```

This signal is received. It notifies the component that it the MGRT has issued a refresh.

```
void refresh_source();
```

Emitting this signal notifies MGRT to refresh its graphics.

Class Reference: Plugin_Base

Brief Description

This is the base class for every plug-in interface. Every plug-in must provide a concrete implementation to ***getName*** and ***getDescription***. The function ***getName*** returns the name of the plug-in, and ***getDescription*** returns a description of the plug-in.

Public Members:

- `virtual std::string getName() = 0;`
- `virtual std::string getDescription() = 0;`

Detailed Description

```
virtual std::string getName() = 0;
```

Virtual function which must be overloaded to return the plug-in's name.

```
virtual std::string getDescription() = 0;
```

Virtual function which must be overloaded to return a description of the plug-in.

Class Reference: Generic_Model_Plugin

Brief Description

Inherits: **Plugin_Base**

The generic model interface is a stripped down interface, which can only be passed a read-from and write-to database object. It is defined as a simple, no frills interface for testing plug-in functionality.

Public Members:

- `virtual std::string getName() = 0;`
- `virtual std::string getDescription() = 0;`
- `virtual void read( ReadFromDB ) = 0;`
- `virtual void write ( WriteToDB ) = 0;`

Detailed Description

```
virtual std::string getName() = 0;
```

Virtual function which must be overloaded to return the plug-in's name.

```
virtual std::string getDescription() = 0;
```

Virtual function which must be overloaded to return a description of the plug-in.

```
virtual void read( ReadFromDB ) = 0;
```

This function tests the read capability.

```
virtual void write ( WriteToDB ) = 0;
```

This function tests the write capability.

Class Reference: Function_Plugin

Brief Description

Inherits: **Plugin_Base**

When the Function_Plugin is loaded a menu item with the plug-in's name is added to the plug-in menu of the MGRT. If this item is clicked, the function defined by the implementation of this interface is called. This interface uses read-from and write-to database objects. If desired, it can show a modal dialog when called. And the dialog can be used as a GUI front end to the function.

Public Members:

- `virtual std::string getName() = 0;`
- `virtual std::string getDescription() = 0;`
- `virtual void func(ReadFromDB, WriteToDB) = 0;`
- `virtual QDialog* instance(QWidget* parent, ReadFromDB reader, WriteToDB writer) = 0;`
- `virtual bool haveGui() = 0;`

Detailed Description

```
virtual std::string getName() = 0;
```

Virtual function which must be overloaded to return the plug-in's name.

```
virtual std::string getDescription() = 0;
```

Virtual function which must be overloaded to return a description of the plug-in.

```
virtual void func(ReadFromDB, WriteToDB) = 0;
```

If the plug-in does not have a GUI this function is called.

```
virtual QDialog* instance(QWidget* parent, ReadFromDB reader, WriteToDB writer) = 0;
```

This function is called if the plug-in has a GUI. This function should spawn a dialog containing the plug-in's GUI.

```
virtual bool haveGui() = 0;
```

This function should return true if the plug-in has a GUI.

Class Reference: DockModual_Plugin

Brief Description

Inherits: **Plugin_Base**

The dock module interface, adds a dockable window to the MGRT's interface. The dock window can be assigned to specific locations, and it can specify a default location. This interface provides a plug-in with read-from and write-to database objects.

Public Members:

- `virtual std::string getName() = 0;`
- `virtual std::string getDescription() = 0;`
- `virtual QDockWidget* instance(QWidget* parent, ReadFromDB reader, WriteToDB writer) = 0;`
- `virtual Qt::DockWidgetAreas dockableAreas() = 0;`
- `virtual Qt::DockWidgetArea defaultDockArea() = 0;`

Detailed Description

```
virtual std::string getName() = 0;
```

Virtual function which must be overloaded to return the plug-in's name.

```
virtual std::string getDescription() = 0;
```

Virtual function which must be overloaded to return a description of the plug-in.

```
virtual QDockWidget* instance(QWidget* parent, ReadFromDB reader,
WriteToDB writer) = 0;
```

This function returns an instance of a dock widget created by the plug-in.

```
virtual Qt::DockWidgetAreas dockableAreas() = 0;
```

This function returns a flag with all the areas the dock widget can dock with the main window.

```
virtual Qt::DockWidgetArea defaultDockArea() = 0;
```

This function returns a flag with the default dock area in the main window.

Class Reference: FileReader_Plugin

Brief Description

Inherits: **Plugin_Base**

The file reader interface is used to add new file formats for reading to the MGRT. It adds file data to the generic storage with a write-to database object. Furthermore, it provides the MGRT with the file extensions it can read. These extensions can be used as filters when loading files. They also are used in a map for fast check to determine if a file can be read. This interface, also, defines a more comprehensive check to determine if a file can be read. This check is called before the file name is passed along to the interface's file reading method.

Public Members

- `virtual std::string getName() = 0;`
- `virtual std::string getDescription() = 0;`
- `virtual void read( std::string filename, WriteToDB writer ) = 0;`
- `virtual bool canRead( std::string filename ) = 0;`
- `virtual std::string readFilter() = 0;`
- `virtual std::string readFileExts() = 0;`

Detailed Description

```
virtual std::string getName() = 0;
```

Virtual function which must be overloaded to return the plug-in's name.

```
virtual std::string getDescription() = 0;
```

Virtual function which must be overloaded to return a description of the plug-in.

```
virtual void read( std::string filename, WriteToDB writer ) = 0;
```

This function when called tells the plug-in which to file to read in. The MGRT code passes in the file name and a ***WriteToDB***.

```
virtual bool canRead( std::string filename ) = 0;
```

This function test if the plug-in can read a file.

```
virtual std::string readFilter() = 0;
```

This function returns a string with a file filter. The filter has the following format: “Format Type (*.ext1, *ext2, *extN) “

```
virtual std::string readFileExts() = 0;
```

This function returns a string with all the file extensions. The string has the following format “ext1;ext2;extN”.

Class Reference: FileWriter_Plugin

Brief Description

Inherits: **Plugin_Base**

The file writer interface is similar to the File Reader. It is used to add new file formats for writing to the MGRT. It writes file data using a read-from database object. Furthermore, it provides the MGRT with the file extensions it can write. These extensions can be use as file filters when specifying a file to write. The extensions are also used to in a map to determine which plug-in to use if an extension filter has not been specified.

Public Members:

- `virtual std::string getName() = 0;`
- `virtual std::string getDescription() = 0;`
- `virtual bool write( std::string filename, ReadFromDB reader ) = 0;`
- `virtual std::string writeFilter() = 0;`
- `virtual std::string writeFileExts() = 0;`

Detailed Description

```
virtual std::string getName() = 0;
```

Virtual function which must be overloaded to return the plug-in's name.

```
virtual std::string getDescription() = 0;
```

Virtual function which must be overloaded to return a description of the plug-in.

```
virtual bool write( std::string filename, ReadFromDB reader ) = 0;
```

This function when called tells the plug-in which to file to write out. The MGRT code passes in the file name and a ***ReadFromDB***.

```
virtual std::string writeFilter() = 0;
```

This function returns a string with a file filter. The filter has the following format: “Format Type (*.ext1, *ext2, *extN) “

```
virtual std::string writeFileExts() = 0;
```

This function returns a string with all the file extensions. The string has the following format “ext1;ext2;extN”.

Class Reference: GraphicDisplay_Plugin

Brief Description

Inherits: **Plugin_Base**

The graphic display interface is used plug-in new main displays for the MGRT. While the interface is called graphic displays, the plugged-in display does not have to be graphics (OpenGL) based. The display can be any GUI item or set of GUI items the plug-in defines. This interface provides a plug-in with an interactive database object.

Public Members:

- `virtual std::string getName() = 0;`
- `virtual std::string getDescription() = 0;`
- `virtual QWidget* instance(QWidget* parent, InteractiveDBObject idbo) = 0;`

Detailed Description

```
virtual std::string getName() = 0;
```

Virtual function which must be overloaded to return the plug-in's name.

```
virtual std::string getDescription() = 0;
```

Virtual function which must be overloaded to return a description of the plug-in.

```
virtual QWidget* instance(QWidget* parent, InteractiveDBObject idbo) =
0;
```

This function returns a widget which can be used for a central widget in the main window.

Class Reference: ToolBar_Plugin

Brief Description

Inherits: **Plugin_Base**

The toolbar interface gives a plug-in the ability to add a toolbar to the MGRT's interface. The buttons on a plugged-in toolbar can call dialogs or other GUI items defined by the plug-in. This provides interface provides the plug-in with read-from and write-to database objects.

Public Members:

- `virtual std::string getName() = 0;`
- `virtual std::string getDescription() = 0;`
- `virtual QToolBar* instance(QWidget* parent, ReadFromDB reader, WriteToDB writer) = 0;`

Detailed Description

```
virtual std::string getName() = 0;
```

Virtual function which must be overloaded to return the plug-in's name.

```
virtual std::string getDescription() = 0;
```

Virtual function which must be overloaded to return a description of the plug-in.

```
virtual QToolBar* instance(QWidget* parent, ReadFromDB reader,
WriteToDB writer) = 0;
```

This function returns a toolbar to be used in the main window.

Class Reference: Curve_Plugin

Brief Description

Inherits: **Base_Plugin**

The curve plug-in interface is used to add new curve definition types as defined in the modified GTM. The new curve type is added to a singleton curve factory class. A curve of the defined type can be created with the factory class if a provided curve data object matches in type with the curve definition.

Public Members:

- `virtual std::string getName() = 0;`
- `virtual std::string getDescription() = 0;`
- `virtual curve_definition* definition() = 0;`

Detailed Description

`virtual std::string getName() = 0;`

Virtual function which must be overloaded to return the plug-in's name.

`virtual std::string getDescription() = 0;`

Virtual function which must be overloaded to return a description of the plug-in.

`virtual curve_definition* definition() = 0;`

This function returns a pointer to the curve definition associated with the plug-in. This function must be overloaded to return a custom definition.

Class Reference: Surface_Plugin

Brief Description

Inherits: Plugin_Base

The surface plug-in interface is used to add new surface definition types as defined in the modified GTM. The new surface type is added to a singleton surface factory class. A surface of the defined type can be created with the factory class if a provided surface data object matches in type with the surface definition.

Public Members:

- `virtual std::string getName() = 0;`
- `virtual std::string getDescription() = 0;`
- `virtual surface_definition* definition() = 0;`

Detailed Description

`virtual std::string getName() = 0;`

Virtual function which must be overloaded to return the plug-in's name.

`virtual std::string getDescription() = 0;`

Virtual function which must be overloaded to return a description of the plug-in.

`virtual surface_definition* definition() = 0;`

This function returns a pointer to the surface definition associated with the plug-in. This function must be overloaded to return a custom definition.

APPENDIX B

PLUG-IN WALKTHROUGH

This appendix presents a walk through of the io2dm plug-in class. To fully understand this walkthrough, the reader is expected be familiar with object-oriented programming, C++, and the Qt API. The Qt API is documented in [14] [15] [16].

This appendix will use the following format.

```
C++ code is in boxes like this
```

Descriptions and explanations will precede its associated code.

The current file is denoted as a section tile. The word File suffixed with a colon is followed by the file name. For example:

File: *example_file.txt*

File: i*o2dm.hxx*

This file begins with the inclusion of the Qt libraries and the path to the interface header.

```
#ifndef IO2DM_HXX
#define IO2DM_HXX

#include <QtGui>
#include <QObject>
#include <string>

#include "plugin_interfaces/mdb_interface.h"
```

Here we define our class. All plug-ins must be publicly derived from ***QObject***. The additional base classes are the plug-in interfaces we are implementing. It is allowable to inherit from multiple interfaces as seen here.

Qt requires that the Q_OBJECT macro be declared in all classes using signals and slots. The Q_INTERFACES macro declares which plug-in interfaces are being implemented. Both of these are required.

We now implement all the member functions from the abstract base classes (Base_Plugin, FileWriter_Plugin, and FileReader_Plugin).

```
class io2DM : public QObject, public FileWriter_Plugin, public
FileReader_Plugin
{
    Q_OBJECT
    Q_INTERFACES(FileWriter_Plugin FileReader_Plugin)
```

```
    Q_OBJECT
    Q_INTERFACES(FileWriter_Plugin FileReader_Plugin)
```

```
public:
	std::string getName();
	std::string getDescription();

	void read( std::string filename, WriteToDB writer );
	bool canRead( std::string filename );
	std::string readFilter();
	std::string readFileExts();

	bool write( std::string filename, ReadFromDB reader );
	std::string writeFilter();
	std::string writeFileExts();
};

#endif
```

That is all that is required in the plug-ins prototype. We must now define the actual functions. We can do this in the same file or a separate *.cpp or *.cxx file. Those familiar with C++ should be familiar with the header file and source file concept.

File: i*o2dm.cpp*

Start by including the required headers for the implementation. In this example, the needed standard C++ and Qt headers, along with io2dm.hxx are included.

```
#include "io2dm.hxx"
#include <fstream>
#include <QTextStream>
#include <numeric>
#include <algorithm>
#include <functional>
```

This file uses namespace *std* since we will be using many functions in that namespace. This is not required. It was done to save a few keystrokes. Additionally, declaring this namespace here will not affect the namespaces of other plug-ins or MGRT components.

These functions return the plug-in's name and description.

```
using namespace std ;
```

```
std::string io2DM::getName()
{
	return "2DM/3DM plugin";
}
std::string io2DM::getDescription()
{
	return "Plug-in for reading and writing 2DM/3DM formatted
meshes.";
}

void io2DM::read( std::string filename, WriteToDB writer )
{
	QFile file(filename.data());
	QTextStream strm;
	file.open(QFile::ReadOnly | QFile::Text);
	strm.setDevice(&file);

	// Implementation details omitted...
```

Once we create our objects we encapsulate them in a std::pair<int,void*> with the encapsulate functions. Then we write the objects to the database using a writeToDB object class, which was passed in to the function.

```
	std::pair<int, void *> record = encapsulate( myCreatedObject );

	std::vector< std::pair<int, void *> > tempVec;
	tempVec.push_back(record);

	writer.write(tempVec.begin(),tempVec.end());
}
```

```
bool io2DM::canRead( std::string filename )
{
	QFile file(filename.data());
	bool readable = false;
	QFileInfo info(file);
	if(info.suffix().contains("2dm"))
		readable = true;
	else if(info.suffix().contains("3dm"))
		readable = true;
	return readable;
}
```

```
std::string io2DM::readFilter()
{
	return "SMS 3DM FORMAT (*.2dm, *.3dm)";
}
std::string io2DM::readFileExts()
{
	return "2dm;3dm";
}
```

The write function checks the file name for the extension *.3dm*. If the file name does not end in *.3dm*, it appends on.

```
bool io2DM::write( std::string filename, ReadFromDB reader )
{
	if ( filename.find(".3dm") == std::string::npos )
		filename.append(".3dm");
	std::fstream outfile(filename.data(), std::ios::out);
```

Data can be retrieved from the database using the ReadFromDB class pass into the function.

Here we are retrieving all faces and inserting those values in to a ***std::vector<triplet<int,int,void*> >***. The triplet contains the type enumeration in the first member, the face's identification number in the second, and a pointer to the faces in the

```
	std::vector< triplet<int, int, void*> > recVec;
	reader.read(std::back_inserter(recVec), enumFace);

// Implementation details omitted...

	if(success)
		return true;
	else
		return false;
}
```

third.

If we have written the file out successfully, the function return true; otherwise, the function returns false.

```
std::string io2DM::writeFilter()
{
      return "SMS FORMAT (*.2dm *.3dm)";
}

std::string io2DM::writeFileExts()
{
      return "3dm;;2dm";
}
```

Like the reading filters the plug-in provides filters for its supported file types.

At the end of the source file the following statement is included. The macro exports the plug-in's name and associates it with the plug-in's class. The first argument is the plug-in target name, which must match the target in the plug-in's project file. The second argument is the plug-in's class name. Project files are discussed in the next section.

File: i*o2dm.pro*

Please see the Qt documentation for specific details regarding project files.

```
TEMPLATE = lib
CONFIG += plugin \
    warn_on
```

```
#ifdef QT_NO_DEBUG
Q_EXPORT_PLUGIN2(pnp_io2DM, io2DM)
#else
Q_EXPORT_PLUGIN2(pnp_io2DM_debug, io2DM)
#endif
```

```
INCLUDEPATH += $MGRTCORELIBRARYPATH/ \
    .
DEPENDPATH += . \
    $MGRTCORELIBRARYPATH /
```

The template is set to library and the configuration is set to plug-in. The warn_on flag stops the compile sequence if there is a problem with the source code.

The current path name and the core library path is included and added to the dependency path.

The project file specifies the headers and source files and the core library.

```
HEADERS += io2dm.hxx
SOURCES = io2dm.cpp
LIBS += -L$MGRTDIR/lib/ \
    -lmgrtCore
```

The target or the plug-in's file name is set as well as the destination directory.

```
debug:TARGET = pnp_io2dm_debug
!debug:TARGET = pnp_io2dm
DESTDIR = $MGRTDIR/plugins

# install =
target.path = $MGRTDIR/plugins
```