DESIGN AND DEVELOPMENT OF AN EXTENSIBLE, INTERCHANGEABLE COMPONENT ARCHITECTURE FOR OPEN- SOURCE GEOGRAPHIC INFORMATION SYSTEMS

by

Harold A. Dunsford Jr.

A Dissertation

Submitted in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy in Engineering and Applied Sciences

In the Department of Geosciences

Idaho State University

May 2010

In presenting this thesis in partial fulfillment of the requirements for an advanced degree at Idaho State University, I agree that the Library shall make it freely available for inspection. I further state that permission for extensive copying of my thesis for scholarly purposes may be granted by the Dean of the Graduate School, Dean of my academic division, or by the University Librarian. It is understood that any copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Signature _____

Date _____

To the Graduate Faculty:

The members of the committee appointed to examine the thesis of Harold Dunsford Jr. find it satisfactory and recommend that it be accepted.

_____
Dr. Daniel P. Ames,
Major Advisor

_____
Dr. Sylvio Mannel,
Committee Member

_____
Dr. Paul Link,
Committee Member

_____
Dr. Steve Chiu,
Committee Member

_____
Dr. Chikashi Sato,
Committee Member

_____
Dr. Tom Jackson,
Graduate Faculty Representative

Table of Contents

Abstract 1
1. Introduction 3
2. Original Algorithms Developed 5
2.1. Interface Architecture 6
2.2. Kd-tree Generalization 10
2.3. Fast In-Memory PitFill 17
2.3.1. Dry Upwards Implementation 20
2.3.2. Pit Handling Improvement 23
2.3.3. Piecewise Implementation 28
2.3.4. Pitfill Summary 30
3. GIS and Open Source Software 32
3.1. Open Source Software 33
3.2. Open Source GIS 37
3.2.1. Organizations 37
3.2.2. GRASS 38
3.2.3. GeoTools 40
3.2.4. SharpMap 41
3.2.5. NASA World Wind 41
3.3. MapWindow 43
3.3.1. History 43
4. Project Requirements and Considerations 45
4.1. Data Formats 48
4.1.1. Temporal GIS 50
4.1.2. File Formats 51
4.1.3. Database Access 52
4.1.4. Web Service Consumption 53
4.2. Symbology 53
4.2.1. Symbolizers and Thematic Layers 53
4.2.2. Selections 54
4.2.3. Categories and Schemes 56
4.2.4. Polygon Triangulation 56
4.3. Topology 57
4.4. Map Projections and Coordinate Systems 59
4.5. Components 64
4.6. Tools and Model Building 65
5. Methods 66
5.1. Development Tools and Procedures 66
5.2. Code Conventions 68
5.3. 2D Vector Drawing 71
5.3.1. GraphicsPath vs. DrawLines 72
5.3.2. Point Culling 73
5.3.3. Geometric Features vs. Vertices 76
5.3.4. Extended Buffers and Multi-Layer Buffers 79
5.3.5. Line Overlap Styles 82
5.3.6. Vertex Storage 83

5.3.7. Anti-aliasing 84
6. Component Architecture Design 86
6.1. Geometry Relationships 86
6.1.1. Overlay Operations: 90
6.2. Data Management Architecture 92
6.2.1. Point 92
6.2.2. Extents 93
6.2.3. Shapes 93
6.2.4. Shapefile 94
6.2.5. Grid 96
6.2.6. GridHeader 99
6.2.7. Image 100
6.2.8. Extension Methods 101
6.3. Symbology Architecture 104
6.3.1. Layers 105
6.3.2. Rendering Layers 107
6.3.3. Categories 108
6.3.4. Schemes 110
6.3.5. Point Symbol Classes 113
6.3.6. Line Symbol Classes 113
6.3.7. Polygon Symbol Classes 114
6.3.8. Raster Symbology Classes 115
7. Implementation and Demonstration 117
7.1. Assembling a Map Project 117
7.1.1. Step 1: Start a New C# Application 117
7.1.2. Step 2: Add MapWindow Components 118
7.1.3. Step 3: Add Menu and Status Strips 121
7.1.4. Step 4: Add the Map 123
7.1.5. Step 5: Add the Legend and Toolbox 124
7.1.6. Step 6: Link It All Together 126
7.2. Simplifying Australia Data Layers 129
7.2.1. Step 1: Download Data 129
7.2.2. Step 2: Calculate Polygon Areas 129
7.2.3. Step 3: Compute Centroids 132
7.2.4. Step 4: Sub-sample by Attributes 133
7.2.5. Step 5: Export Layer to a File 135
7.2.6. Step 6: Repeat with Roads Layer 135
7.2.7. Step 7: Select Political Bounds by Clicking 136
7.2.8. Step 8: Apply Labeling 137
7.3. Programmatic Point Symbology 138
7.3.1. Add a Point Layer 139
7.3.2. Simple Symbols 142
7.3.3. Character Symbols 144
7.3.4. Image Symbols 148
7.3.5. Point Categories 152
7.3.6. Compound Point symbols 154

7.4. Programmatic Line Symbology 155
7.4.1. Adding Line Layers 155
7.4.2. Simple Line symbols 156
7.4.3. Outlined Symbols 156
7.4.4. Unique Values 157
7.4.5. Custom Categories 158
7.4.6. Compound Lines 159
7.4.7. Line Decorations 161
7.5. Programmatic Polygon Symbology 162
7.5.1. Add Polygon Layers 162
7.5.2. Simple Patterns 164
7.5.3. Gradients 165
7.5.4. Individual Gradients 166
7.5.5. Multi-Colored Gradients 168
7.5.6. Custom Polygon Categories 169
7.5.7. Compound Patterns 170
7.6. Programmatic Labels 171
7.6.1. Field Name Expressions 172
7.6.2. Multi-Line Labels 173
7.6.3. Translucent Labels 174
Programmatic Raster Symbology 175
7.7. 175
7.7.1. Download Data 175
7.7.2. Add a Raster Layer 178
7.7.3. Control Category Range 182
7.7.4. Shaded Relief 183
7.7.5. Predefined Schemes 184
7.7.6. Edit Raster Values 184
7.7.7. Quantile Breaks 186
8. Conclusion 187
References 188
Appendix A: MapWindow 6.0 User's Manual 199
1. End User Tutorial – Getting Started with MapWindow 6.0 199
1.1. Introduction 199
1.1.1. The MapWindow 6 Main Interface 199
1.1.2. The MapWindow 6 Main Menu 200
1.1.3. The MapWindow 6 Toolbar 201
1.1.4. The MapWindow 6 Legend 204
1.2. Symbology Controls 205
1.2.1. Point Symbolizer Dialog 206
1.2.2. The Line Symbol Dialog 213
1.2.3. The Polygon Symbolizer Properties Dialog 218
1.2.4. Raster Symbology 222
1.3. Legend Context Menu 223
1.3.1. Remove Layer 224
1.3.2. Zoom to Layer 224

1.3.3. View Attributes 224
1.3.4. Attribute Table Editor 225
1.3.5. Set Dynamic Visibility 228
1.3.6. Labeling 228
1.3.7. Selection 232
1.3.8. Data 232
1.3.9. Properties 232
1.4. Layer Properties Dialog 233
1.4.1. Coloring 234
1.4.2. Values 234
1.4.3. Statistics and Graphing 238
1.4.4. Statistics 239
1.4.5. Graphing 241
1.4.6. Unique Field 242
1.5. Print Layout 245
1.5.1. Print Layout Main Window 245
1.5.2. Print Layout Menu 246
1.5.3. Print Layout Toolbar 250
1.5.4. Print Layout Contents Window 253
1.5.5. Print Layout Properties Window 253
1.5.6. Map Properties 254
1.5.7. North Arrow Properties 255
1.5.8. Legend Properties 256
1.5.9. Scale Bar Properties 258
1.5.10. Text Box Properties 260
1.5.11. Rectangle Properties 261
1.5.12. Bitmap Properties 262
1.6. Toolbox 262
Appendix B: Vector Drawing Problems and Solutions 266

List of Figures

Figure 1: Lines of Code by Namespace 4
Figure 2: Occlusion relationship to scale (Ted Dunsford original) 14
Figure 3: 2D kd-tree (http://en.wikipedia.org/wiki/Kd-tree) 15
Figure 4: 3D kd-tree (http://en.wikipedia.org/wiki/Kd-tree) 16
Figure 5: Hydrologic Watershed 17
Figure 6: The Illustration of a Pit 18
Figure 7: Completely Flooded Starting Point 20
Figure 8: Perimeter Cells “Dried” out 20
Figure 9: Higher Elevation Interior 21
Figure 10: Drain Higher Neighbors 21
Figure 11: Dried Neighboring Cells. 22
Figure 12: Only Pit Cells Remain 23
Figure 13: Raster Scan Method 23
Figure 14: Enclosed Valley Super-Pit 25
Figure 15: Addressing Pit Cells 26
Figure 16: All High Water Levels Replaced 26
Figure 17: Final Result 27
Figure 18: Default Propagation of Values 27
Figure 19: Less Redundant Propagation. 28
Figure 20: Subsections with Overlap 29
Figure 21: Performance Comparison 31
Figure 22: MapWindow 4.7 43
Figure 23: MapWindow 6.0 (2D map) 45
Figure 24: Continental US UTM Zones 59
Figure 25: Mercator vs. Transverse Mercator 60
Figure 26: Projection Coordinate System Class Diagrams 61
Figure 27: Oblate Spheroid 62
Figure 28: ITransform Interface 63
Figure 29: Randomly Generated Lines 72
Figure 30: Real Road Data Drawing Times 73
Figure 31: DrawLines vs. GraphicsPath 73
Figure 32 : Duplicate Point Elimination Drawing Times 75
Figure 33: Drawing Times as a function of Number of Lines Drawn. 76
Figure 34: Drawing Times for Coordinate Objects or Double Arrays. 77
Figure 35: Memory Usage as a Function of the Number of Shapes for Feature Objects. 77
Figure 36: Direct Drawing vs. Multi-layer Stencil 80
Figure 37: Extended Single Buffer vs. Multiple Layer Buffers 80
Figure 38: Extension vs. Multi-Layer Buffer Drawing Times 81
Figure 39: Two different overlap Styles 82
Figure 40: Buffer.BlockCopy vs. BitConverter.ToDouble 84
Figure 41: No Anti-Aliasing 85
Figure 42: Anti-Aliasing 85
Figure 43: Anti-Aliasing Performance 86
Figure 44: Intersection Matrix 87
Figure 45: Clipped Texas Rivers 90

Figure 46: Coordinate Class 92
Figure 47 : Envelope Class 93
Figure 48: Feature Class 94
Figure 49: FeatureSet 96
Figure 50: Raster Class 98
Figure 51: RasterBounds Class 100
Figure 52: ImageData and WorldFile Classes 101
Figure 53: Raster Extension Methods 102
Figure 54: Feature Extension Methods 103
Figure 55: Symbolizer Class Diagram 104
Figure 56: Thematic Layers 106
Figure 57: Feature Layer Classes 107
Figure 58: Layer, Scheme, and Category in Legend 109
Figure 59: Feature Category Architecture 110
Figure 60: Feature Scheme Classes 111
Figure 61: Available Feature Editor Settings 112
Figure 62: Point Symbol Class Diagram 113
Figure 63: Stroke Class Hierarchy 114
Figure 64: Pattern Class Diagram 115
Figure 65: Raster Symbology Classes 116
Figure 66: New Project Dialog 118
Figure 67: Add MapWindow Tab 119
Figure 68: Choose Items 119
Figure 69: Browse 120
Figure 70: Select MapWindow.dll 120
Figure 71: MapWindow Tools 121
Figure 72: Drag a MenuStrip 122
Figure 73: Add Status and Tool Strips 122
Figure 74: Add a SplitContainer Control 123
Figure 75: Add a Map 123
Figure 76: Add a Tab Control 124
Figure 77 : Add Legend 125
Figure 78: Add a Toolbox 125
Figure 79: Link Map 126
Figure 80: Link Tool Strip 127
Figure 81: Census Data 128
Figure 82: Add Data 130
Figure 83: Sydney Polygon 130
Figure 84: Calculate Areas 131
Figure 85: Newly Added Areas 132
Figure 86: Calculate Centroid 132
Figure 87: Too Many Cities 133
Figure 88: Area > .01 134
Figure 89: Create Layer 134
Figure 90: Export Layer 135
Figure 91: Primary Roads 135

Figure 92: Select Major Areas 136
Figure 93: Activate Labeling 137
Figure 94: Apply Labels 137
Figure 95: Applied Labels 138
Figure 96: C# Code - Add Point Layer 139
Figure 97: Feature Set and Features 140
Figure 98: Extension Method 141
Figure 99: C# Code - Yellow Star Symbolizer 143
Figure 100: Yellow Star 143
Figure 101: C# Code - Black Outline 143
Figure 102: Yellow Stars with Outlines 144
Figure 103: Fonts in Control Panel 145
Figure 104: Right Click to Install New Font 146
Figure 105: Switch to Characters 146
Figure 106: Choose Military 147
Figure 107: C# Code - Blue Character Symbols 147
Figure 108: Military Plane Characters 148
Figure 109: Add a New Item 149
Figure 110: Images.resx Resource 150
Figure 111: Add an Existing File 150
Figure 112: Rename to Tiger 151
Figure 113: C# Code - Image Point Symbol 151
Figure 114: Tiger Images 152
Figure 115: Large Area Cities 152
Figure 116: C# Code - Multiple Category Symbols 153
Figure 117: City Categories by Area 153
Figure 118: C# Code - Quantile Classification 154
Figure 119: Quantile Area Categories 154
Figure 120: C# Code - Multi-Layer Point Symbols 154
Figure 121: Blue Circles with Yellow Stars 155
Figure 122: C# Code - Add Line Layer 155
Figure 123: Add Line Layer 156
Figure 124: C# Code - Color Roads Brown 156
Figure 125: Brown Lines 156
Figure 126: C# Code - Black Outlined Lines 157
Figure 127: Yellow Roads with Outlines 157
Figure 128: C# Code - Unique Value Classification 158
Figure 129: Roads with Unique Values 158
Figure 130: C# Code - Multiple Line Categories 159
Figure 131: Custom Line Categories 159
Figure 132: C# Code - Multi-layer Lines 161
Figure 133: Multi-Stroke Railroads 161
Figure 134: C# Code - Lines with Decorations 162
Figure 135: Lines with Star Decorations 162
Figure 136: C# Code - Add Polygon Layer 163
Figure 137: Add Major Boundaries 163

Figure 138: C# Code - Light Blue Polygons 164
Figure 139: Blue Fill Only 165
Figure 140: C# Code - Polygon Outlines 165
Figure 141: With Blue Border 165
Figure 142: C# Code – Gradient Polygon Fill 166
Figure 143: Continuous Blue Gradient 166
Figure 144: Major Boundaries Fields 167
Figure 145: C# Code - Unique Value Polygon Classification 167
Figure 146: Individual Gradients 168
Figure 147: C# Code - Polygon Color Ramp 168
Figure 148: Unique Cool Colors with Gradient 169
Figure 149: C# Code - Custom Polygon Categories 170
Figure 150: Custom Categories 170
Figure 151: C# Code - Hatch Pattern Polygons 171
Figure 152: Hatch patterns 171
Figure 153: C# Code – Label Features 171
Figure 154: Polygons with Labels 172
Figure 155: C# Code - Centered Field Labels 172
Figure 156: Field Name Labels 173
Figure 157: C# Code - Multiline Labels 173
Figure 158: Multi-Line Labels 174
Figure 159: Conditional Labeling Expressions 175
Figure 160: Translucent Labels 175
Figure 161: Earth Explorer 177
Figure 162: Export Data 178
Figure 163: Save as Bgd 178
Figure 164: C# Code - Add Raster Layer 178
Figure 165: Default Raster 179
Figure 166: Statistics 180
Figure 167: Zoom To Categories 181
Figure 168: Raster Cell Value Histogram After Zoom 181
Figure 169: After Adjustments 182
Figure 170: C# Code - Multiple Category Rasters 182
Figure 171: Programmatically Restricted Range 183
Figure 172: C# Code - Shaded Relief 183
Figure 173: With Lighting 183
Figure 174: C# Code - Predefined Schemes 184
Figure 175: Glaciers 184
Figure 176: C# Code - Crop Extreme Raster Values 185
Figure 177: Default Symbology of Fixed Raster 186
Figure 178: After Fixing Raster 186
Figure 179: C# Code - Quantile Breaks 187
Figure 180: Quantile Breaks 187
Figure 181: MapWindow 6 Main Interface 200
Figure 182: File Menu 201
Figure 183: MapWindow 6 Toolbar 202

Figure 184: FeatureIdentifier Window 204
Figure 185: MapWindow 6 Legend Window 205
Figure 186: Point Symbolizer Dialog Window 207
Figure 187: Symbol Type Options 208
Figure 188: Color Window 210
Figure 189: Select Line Symbol Window 213
Figure 190: Line Symbol Window 214
Figure 191: Stroke Type Options 215
Figure 192: Cartographic Stroke Type Tabs 215
Figure 193: Pattern Type Options 219
Figure 194: Gradient Coloring Tool 221
Figure 195: Polygon Pattern Types 222
Figure 196: CollectionPropertyGrid Window 223
Figure 197: Right Click Menu 224
Figure 198: Attribute Table Editor 225
Figure 199: Attribute Table Editor Toolbar 226
Figure 200: Expression Editor 227
Figure 201: Feature Labeler Window 229
Figure 202: Feature Labeler Tabs 230
Figure 203: Layer Properties Window 233
Figure 204: Values Portion of the Layer Properties Window 235
Figure 205: Layer Properties Values Toolbar 237
Figure 206: Statistics and Graphing Portion of the Layer Properties Window 238
Figure 207: DataValue Method of Interval Snapping 240
Figure 208: None Method of Interval Snapping 240
Figure 209: Graph Display 242
Figure 210: Unique Properties Field for Point and Line Data Types 243
Figure 211: Unique Properties Field for Polygon Data Types 243
Figure 212: Raster with out HillShading on the left and with HillShading on the right 244
Figure 213: Unique Properties Field for Raster Data Types 244
Figure 214: Print Layout File Menu 246
Figure 215: Print Layout Page Setup Window 247
Figure 216: Print Layout Select Menu 248
Figure 217: Print Layout View Menu 249
Figure 218: Print Layout Map Properties 254
Figure 219: Print Layout North Arrow Properties 255
Figure 220: Print Layout Legend Properties 256
Figure 221: Print Layout Fonts Window 258
Figure 222: Print Layout Scale Bar Properties 258
Figure 223: Print Layout Text Box Properties 260
Figure 224: Print Layout Rectangle Properties 261
Figure 225: Print Layout Bitmap Properties 262
Figure 226: Toolbox 263
Figure 227: Reproject Features Window 264
Figure 228: Select a Projection Window. 265
Figure 229: Two different overlap Styles 270

Figure 230: Old Polygon Vertex Order 277

List of Tables

Table 1: Number of Lines Kept 74
Table 2: Buffer Memory Requirements 82
Table 3: Geometry Relate Definitions 88
Table 4: Relate Expressions and Illustrations 89
Table 5: Overlay Operations and Definitions 91

Abstract

This dissertation presents the design and development of a modular, extensible, and interchangeable component architecture for a geographic information system (GIS). Each of the new architectural strategies, the ontological choices, and several new algorithms are examined in detail. A working, early stage open source project serves as the platform within which the principles and concepts presented in this work are tested. The project uses a series of components that each act independently and can be recombined in different ways. This dissertation directly exposes the technical challenges, especially with regards to performance, surrounding the creation of a working geographic mapping system. The result is a two dimensional mapping component that can draw vectors and images at speeds that are comparable to other professional 2D drawing software. We also created a 3D component that uses DirectX for drawing, but there the challenge that was resolved was how to preserve the necessary precision from 64 bit double precision coordinates in a drawing environment that only supports 32 bit floating point coordinates. A great focus was placed on designing the library in such a way that other developers could use it as a software development kit to construct their own geospatial software. While many open source GIS applications exist, no other open source project that we know of offers this kind of ability for the .Net environment. The interface based architecture is specifically designed to be extended by future developers. The actual nature of the interface based extensibility architecture is something that is not really replicated in any other open source GIS package. The result is that most objects that are used by our framework can be exchanged for different objects that follow the

same rules, enabling our software to work directly with classes that do not exist yet, without us having to re-compile the project.

The underlying motivation for the project was to create the GIS framework that could be used in order to address scientific problems in hydrology. One of the outcomes of this study was the development of a very fast method for filling pits. The method allows for piecewise evaluation of rectangular blocks in memory, which is a unique challenge because of the dependencies that exist between blocks. It also introduces some optimizations that improve the algorithm in performance by about a factor of 2 over our implementation of the fastest known previously documented algorithm. Our application also provides a working C# implementation of that algorithm, rather than just the theoretical process.

Light Detection And Ranging (LiDAR) datasets also represent a significant problem for GIS systems to display the points effectively. The major problem with working with these large datasets is that the file format can contain billions of points. This much information is impossible to load into memory all at once, and would also be extremely cumbersome to draw onto the screen when every single point is rendered. We propose a novel kd-tree based data structure that would allow these points to be stored on the disk in a fashion that allows not only the elimination of points outside the 2D extent, but also allows the view to eliminate points that would overlap. This dissertation leaves much of actual implementation of this solution to future development, but a working prototype was created and tested, giving very good overall performance for the purpose of displaying the points.

1. Introduction

The central contribution of this dissertation is the creation of the MapWindow 6.0 GIS framework. This is a working C# code base that can be used to create custom GIS software. It is freely available for download for the general public at http://mapwindow6.codeplex.com. It is fully implemented and currently in the beta testing stage. It is already being used in the HydroDesktop project funded by the Consortium of Universities for Advancement of Hydrolic Science (CUAHSI) and the Data for Environmental Modeling (D4EM) project funded by the Environmental Protection Agency (EPA). The software was created using a model where, from the very beginning, every line of code was available for public scrutiny using the open source repository system Tortoise SVN (http://tortoisesvn.tigris.org/). The latest version of the source code actually uses a newer, much more efficient repository system called Mercurial (http://mercurial.selenic.com/).

The code itself consists of over 82,000 lines of executable lines of code. After removing the contributions from other developers and the net topology suite, there are still approximately 64,000 lines of executable code that were created by this author for the purposes of running the software. Figure 1 Shows the lines of code in a breakdown according to the different tasks that are required to support the GIS activities. The drawing namespace, for example is concerned with symbology and organizing content to appear in thematic layers that can have specific categories.

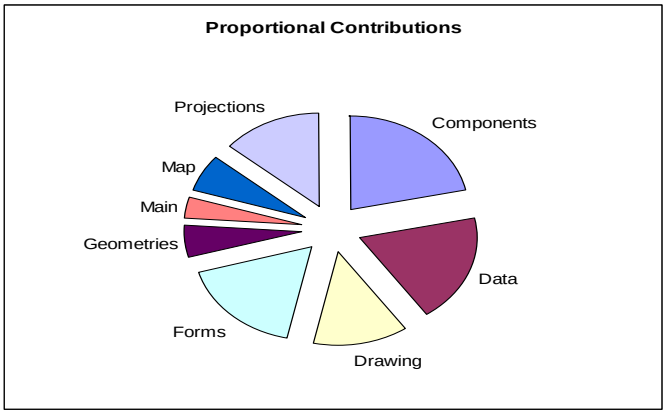


Figure 1: Lines of Code by Namespace

In addition to the raw lines of code, the project also includes comments in the form of inline Extensible Markup Language (XML) that can also appear in Intellisense, which will be described in more detail in section 5.2: Code Conventions. More than 250,000 words of comments were created by this author, or basically enough to fill a 1000 page novel. The GIS system was designed with the experience gained from the earlier versions of the MapWindow GIS project (Ames et al., 2008). The work from this dissertation is licensed through the GNU Lesser General Public License (http://www.gnu.org/copyleft/lesser.html) which is well suited for serving as a starting point for future academic, commercial, or open software development. GNU originated as an operating system that was similar to Unix, but contained no Unix code. The acronym is recursive and stands for "Gnu's Not Unix". Subsequently, the specific license for this kind of open source software gained the name GNU General Public License (GPL) and GNU Lesser General Public License (LGPL)

The MapWindow 6.0 library introduces a series of interfaces that allow any component to be replaced by a different component so long as it can satisfy the basic methods or properties on the interface. It is written in C# to support a strong need for open GIS software that can be used directly by .NET developers. The core of the project

is entirely C#, and so entirely managed and common language runtime (CLR) compliant, allowing it to be referenced by other .NET languages. It targets the 3.5 .NET Framework to allow the code to be directly run in Linux and Macintosh environments using the Mono framework. The project includes separate 2D and 3D map controls that can be used interchangeably, as well as other supporting controls like a legend control, a toolbar and status bar that automatically work together with the map component. The project includes methods from a C# port of the Java Topology Suite in order to perform the basic overlay and intersection operations. This library includes interfaces for each of the geometry classes so that future classes can be used directly with the existing geoprocessing methods, even if they have a different internal structure or extended temporal capabilities. Finally, extensibility interfaces allow new data providers, tool providers, and layer providers that extend the functionality of the system. Even this extensibility itself has been encapsulated in a single component, so that if another developer wants to enable MapWindow 6.0 plugins to work with their project, all they have to do is drag the ApplicationManager component onto their project and set some properties.

The remainder of this dissertation presents original geospatial algorithms and key scientific contributions of the work (Chapter 2), followed by an introduction to GIS and open source software (Chapter 3), design and development of the new MapWindow 6 software (Chapters 4-7), and finally general conclusions (Chapter 8).

2. Original Algorithms Developed

This section is devoted to novel innovations in the interface architecture, K-D tree generalization and fast, piecewise pit-filling algorithms developed as part of this dissertation. With any research project, the special focus is usually the entire topic of the

study. Any innovations that are achieved are generally directly required for the study to progress. In this project, however, there were several incidental achievements or discoveries that are each publishable accomplishments. The first and third have already been published, and the second only needs some formalization before being submitted. Each of these discoveries will be discussed in detail.

2.1.Interface Architecture

Using interfaces to facilitate the extensibility architecture is a topic that has been published (Dunsford & Ames, 2008) and represents a new way of managing extensibility. Much of the success of the MapWindow project has been its easy-to-use plug-in programming environment allowing third-party developers to quickly extend the application using languages such as VB.NET and C#. The current plug-in extensibility architecture in MapWindow 4 uses an “application-wide” design in which each plug-in implements an interface that provides direct interaction with and access to all elements and components in the full MapWindow application. This application wide model provides a “single interface” approach that has proven quite usable, but has some critical deficiencies.

Specifically, developers who only need to use one or two components should not be required to implement such a large and inclusive interface. More specifically, a plug-in that only provides a new data type, or one that only provides a new geoprocessing tool, should not necessarily require implementing an interface with hooks to the GIS map, legend, or toolbar. Indeed, every custom GIS application is generally different enough that no one plug-in interface would be suitable for every conceivable configuration of modular components. The architectural solutions introduced in this paper (and

implemented in the source code of the MapWindow 6.0 project) address improving the extensibility model itself by creating a series of plug-in manager components (and a much more granular set of interfaces), each one handling a narrow focus of extended functionality.

Presumably, the easier it is to extend a GIS platform, the more usable it will be for scientists and engineers who are not programmers. To cater to this need, proprietary software applications (e.g., ArcGIS) support the writing of plug-ins that extend the capabilities of the software without having to re-compile the binaries or be expert programmers. Many open-source GIS software projects also benefit from a plug-in environment because it offers a layer of insulation between the core application required by all users, and the newest, least stable code that is actively being developed. It is therefore useful to include the concept of plug-in extensibility into a modular framework that mirrors the modular component development itself. MapWindow GIS is only weakly copyleft, allowing the use of components themselves in both open and closed source applications, and so designing the plug-in architecture itself to be modular gives greater flexibility to developers using the MapWindow developer tools. (Greenberg & Fitchett. 2001; Greenberg, 2007)

Some key interfaces under development are as follows. Data Interfaces including *IRaster* and *IFeatureSet* as generic forms of raster and vector data. *IFeatureSet* is comprised of *IFeature* objects consisting of attribute data as well as geometric data. These data interfaces are not rendered directly, rather for drawing, an *ILayer* interface is introduced to bring together existing datasets with *ISymbolizers*. The symbolizer interface specified colors, drawing styles, and other visual characteristics. Layers are organized

into a collection called an *IMapFrame*, which also specifies a visual extent (2D) or perspective (3D). Topology is controlled through a set of classes derived from the Java Topology Suite that satisfy the Open GIS Consortium (OGC) simple feature architecture (ISO19107:2003). Geometry classes including *Envelopes*, *Points*, *LineStrings*, *Polygons* and various *Multi-Geometries* have each been created, and both abbreviated and complete interfaces have been developed for each of the classes. The *IBasicGeometry* interfaces convey all the information needed to create a feature, but do not require topological methods. Implementing these basic interfaces from a new data format is trivial compared to attempting to re-implement the entire suite of topology methods (Boissonnat, 1998). To make it easier to use topology methods when starting with an abbreviated interface, each of the existing geometry classes have been equipped with constructors that require the basic version of the same type.

Some developers may wish to extend one aspect of the project themselves, but not want to support additional plug-ins. Other developers might want to open the options for data handling, supporting them through a plug-in like capability, while preventing any extensions that provide tools or custom drawing layers from being used. The model that has been developed to support this wide range of abilities is called a manager component. Rather than having a single block of code in the MapWindow 6.0 application itself, a manager is a general description that includes several different components, each with a corresponding add-in interface that it manages. Each manager can then be configured at design time in order to specify the other components with which it should connect through its properties. The manager then knows enough about the application to adjoin some kind of pre-defined application interface to the various plug-ins. To use the

manager, developers drag a manager component onto their application. Managers themselves are components but do not have a visible control, so they will appear at the bottom of the designer window, just like a *DataAdapter* or *ImageList* component. The properties of the manager component can be customized in the property window, enabling or disabling the various techniques for supporting plug-in extensibility.

Plug-in developers can focus on the provider rather than the manager. Supporting a new data file format to support shapefiles, for instance, would require implementing the *IFeatureSetProvider* interface. A provider supplies the methods for opening and saving an *IFeatureSet*. The *IFeatureSet* can use the existing *FeatureSet* class, but alternately could be a completely new class that simply answers the same questions.

In summary, this project has developed a plug-in extensibility architecture that is modular in order to be used side by side with modular components. It uses a component-like drag and drop behavior that should be very familiar to visual developers. The project makes this operation possible by using interfaces for each of its classes. The interfaces allow for interchangeability, can restrict access to components, and consist of smaller interfaces that can be used where implementing the entire class is not necessary. These interfaces then allow for the addition of run-time classes identified by using system-reflection. Because the new classes implement pre-defined interfaces, other components in the architecture can interact with the new classes as if they had a programmatic reference to them. Because the entire project has been built to use interfaces, future developers can exchange literally any part of it with their own classes as long as the classes implement the same interfaces.

The steps being taken by this project show an exciting new trend for community code. While mapping projects are abundant, projects that set themselves up to provide components for use by other projects are rare, and projects that allow those components to be used for free by developers of proprietary software are rarer still. This approach may yet prove to be naïve, as many advocates of the GNU general public license (GNU, 2007) warn. However, producing easy-to-use components that are specifically designed for an environment used predominantly by proprietary developers would be largely useless unless those proprietary developers could use the software. Attempting to create tools that can be used for multiple purposes also presents new problems and perspectives that can lead to solutions that benefit the entire community.

2.2.Kd-tree Generalization

The problem is that drawing hundreds of thousands of point symbols to the screen is impractical, but this quantity is routinely encountered in software that is designed for geographic information systems. Some examples would include private well data for a state, which can number in the millions of points. (EPA, 2008) Other examples include newer Light Detection and Ranging (LIDAR) sample points, which store individual elevation readings as point clouds at a resolution measured in centimeters for areas measured in square kilometers, resulting in billions of points (Haugerud et al., 2003) Furthermore, most symbolic representations of points are larger than a single pixel, and are more likely to be drawn as such. Drawing more than a few thousand point symbols to the screen creates enough overlap that most of the drawing time is spent drawing redundant symbols. Eliminating points that are outside the extents is typically very fast because the points can be organized into binary tree structures that can eliminate all the

members that are outside the extents in log(n) time. (Hjaltason & Samat, 2002; Robinson, 1981) This paper investigates how to create a similar index that prevents overlapping points from being drawn.

Algorithms that reduce drawing complexity are most important to computer visualization. Rendering simplifications is usually concerned with preventing the drawing of hidden surfaces in 3D space (McKenna, 1987; Mulmuley, 1989). In GIS, two dimensional points are often displayed by the millions, but all spread out over the same two-dimensional surface (Panse et al., 2006). Any low-level improvements in the basic behavior for GIS software will ultimately be appreciated by a wide range of engineers and scientists using this software on a regular basis.

The most intuitive method for checking if a point would overlap an existing point on the screen involves drawing the points one by one, and checking each new point against previously drawn points to ensure that it is not in the same place. This overlap constraint has simply been expressed as the expression in equation 1. In this equation, the notation "bi" represents the x and y position of the i^{th} point. (Panse et al., 2006)

$$i \neq j \Rightarrow bi \neq bj \quad \forall i, j \in \{1, \ldots, N-1\} \qquad \text{[Eqn. 1]}$$

This section will make use of what is known as "Big O" notation, which is an approximation of the running time in terms of the number of elements being considered. The number of members is considered to be n. O(n) would represent something where the performance graph, as a plot, would be a straight line that increased as a linear function of n. $O(n^2)$, on the other hand, would represent a performance curve where the time increases as the square of n. Since testing for overlapping points tends to behave like a nearest neighbor evaluation, it has an unsophisticated performance of around $O(n^2)$

(Cleary, 1979), though methods exist that improve on that performance (Angiulli, 2007; Berchtold et al, 1997). The overlap test would normally have to be recalculated each time the scale was changed because the pixel distance between the points changes as a function of the viewing scale relative to their geographic separations. Because the performance of even the fastest nearest neighbor (NN) tests (when performed on the entire set) are worse than O(n), for very large values the check can become slower than simply drawing the points. Building an index that can prevent overlap must be useable at different scales. At large scales, more elimination occurs from points lying outside the drawing area, and so it is better to have an index that can rapidly eliminate those points.

Currently, several popular GIS software applications do not try to solve this problem at all, and simply draw all the points. Computer vision problems and solutions have become progressively more focused on solving problems that specifically involve polygon representations in three dimensions. These methods have benefitted from the development of new, high-performance hardware solutions. Existing methods of occlusion testing also seem to focus on cases where one polygon falls between the camera and another polygon (An et al., 2002; Bittner, 2002; Brodsky et al., 1995; Fuchs et al., 1980; Mulmuley, 1989; Snyder & Lengyel, 1998). The previous authors do not attempt to address the issue of what happens when all the polygons are organized on a plane preventing that kind of occlusion testing. Two-dimensional optimizations do exist for extent-checking, (Beckmann et al., 1990; Cheng et al., 1992; Henrich, 1996; Robinson, 1981) but are generally optimized only to consider if a point falls outside the view without checking to see if the point would be redundant with one already drawn to the screen.

Many algorithms exist to subdivide two-dimensional geographic space in order to speed up extents testing. For instance, a common approach is the use of Quad-Trees. (Aref & Samet, 1997; Berg, et al., 1997; Lightstone & Mitra, 1997; Proietti, 1999). This sort of spatial decomposition can be especially useful for generating meshes from collections of points (Bern et al., 1999). Through the course of this project a new kd-tree-based occlusion test has been devised that can eliminate points that are known to overlap. Creating trees like quad-trees or kd-trees that are balanced can be complicated and often requires complicated coloring schemes (Eppstein et al., 2007; Guibas & Sedgewick, 1978; Stout & Warren, 1986). Kd-trees are especially dangerous to attempt to balance because the typical method of rotation will not work since the dimensions change with each generation. Building a tree in such a way that it will be balanced can slow down the time it takes to build a tree, and so there has been much work into attempting to speed up this construction (Bern et al., 1999; Hjaltason & Samet, 2002; Hsiao, 1996). Kd-trees are not strictly required to be perfectly balanced, and should exhibit close to optimal behavior so long as the points are added to the tree randomly (Cannady, 1978).

A similar process can be applied to lines and polygons by replacing an overlap criterion with a Duglass-Pucker Line Simplification criterion. In either case, the optimal rendering would only draw the points or vertices that contribute visually discernable content to the screen. In either case, this characteristic depends on the zoom distance from the camera. Mathematically, the scale at which a point becomes unhidden corresponds to the geographic distance between those points. The scale calculations is illustrated in Figure 2.

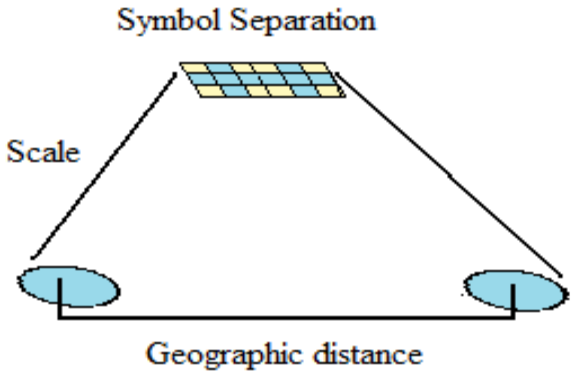


Figure 2: Occlusion relationship to scale (Ted Dunsford original)

Equation 2 is used to determine the scale at which any two points occlude one another.

Geographic Distance = (Pixel Distance * Resolution) / Scale [Eqn. 2]

A method for mathematically evaluating the criterion exists so that a "scale" at which a particular point becomes visible can be recorded in advance. Where points are concerned, the threshold for simplification is controlled by actual overlap in pixels, but in the case of lines, this is represented by the Douglas-Peucker (DP) line-simplification criterion. This criterion can then be stored with each point, effectively as a third dimension or Z value.

The problem of eliminating points that are outside of the desired rectangular extents can be addressed in several ways. Even a direct test with every coordinate is faster than sending points that are outside the visible extents to the drawing method. A kd-tree in the X and Y dimensions can make this selection process require a small fraction of the time because it can carve away large sections of the tree in a single step. Kd-trees are a sub-class of binary trees (Bayer, 1972) in which the plane of bisection cycles through the dimensions with each step. So first, it would subdivide the space horizontally, and then vertically. In the case illustrated in Figure 3, the first division was

calculated from the x-axis, and the median point forms a sub-division that extends across the entire range of Y values. Figure 3 shows a graphical representation of a 2D kd-tree.

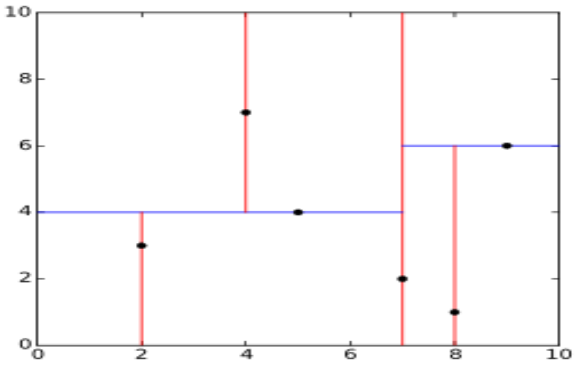


Figure 3: 2D kd-tree (http://en.wikipedia.org/wiki/Kd-tree)

In Figure 3, the dots represent actual data values stored in the tree. The lines represent the planes of division where the two dimensional space is divided. Only the first point added to the tree actually subdivides the entire space. Subsequent points subdivide only the region that they fall within. The k in kd-trees specifies the number of dimensions that exist in the search key (Bentley, 1975). They have also seen some more recent success at allowing some dynamic addition and removal from these trees (Bentley, 1990). This project incorporates the idea of occlusion into the kd-tree itself. The first two dimensions are representative of the two dimensional position of the point on a plane, or its geographic coordinates. The third dimension, however, is not elevation, but rather a new criterion that is being introduced here that can be used to rapidly eliminate overlapping points. In essence, the kd-tree can be extended with an additional dimension. An example of a 3d-tree is shown in Figure 4.

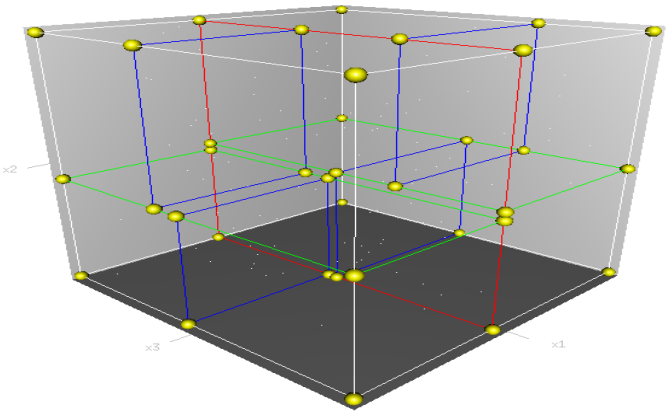


Figure 4: 3D kd-tree (http://en.wikipedia.org/wiki/Kd-tree)

The serious advantage of a kd-tree approach to storing points is that they can be culled without having to test every point. If there are a million available points, but only 1000 need to be drawn, it is still slow to consider each of the million points and make a decision of whether each one needs to be drawn. So the great boon of a kd-tree is that entire sections that are simply outside of the view range can be carved off without ever being examined. The ingenuity of this new approach is to do a one-time calculation in advance to determine a third criterion. This third criterion is simply added to the vertices, and a third dimension is added to the kd-tree. Normally when view is magnified to the entire extents, it will not matter that the points in a 2D kd-tree are present, because they cannot be eliminated from the drawing list since they are all in the bounds of the screen. However, with the addition of the new criterion, the points that are occluded by other points can be culled rapidly without testing every single one. The fact that many points can be skipped at once is accomplished by the binary structure of the tree, but the reason they can be skipped safely is because they are being hidden by a point that has already been drawn on the screen.

The advantage of culling points based on an overlap criteria is that when the view is zoomed out, a large percentage of points can be culled as a result of generalization, and

when zoomed in, a large percentage of points can be culled because they are outside of the extents. The result is that no matter what the zoom level, only a few points ever need be drawn.

2.3.Fast In-Memory PitFill

The GIS analysis that involves hydrology, specifically watershed delineation, requires implementing several algorithms that are commonly used by hydrologists to delineate watersheds from digital elevation models (DEMs). Figure 5 illustrates the basic principal of a watershed, which can be described as the region within which all the water will be directed to a single point.

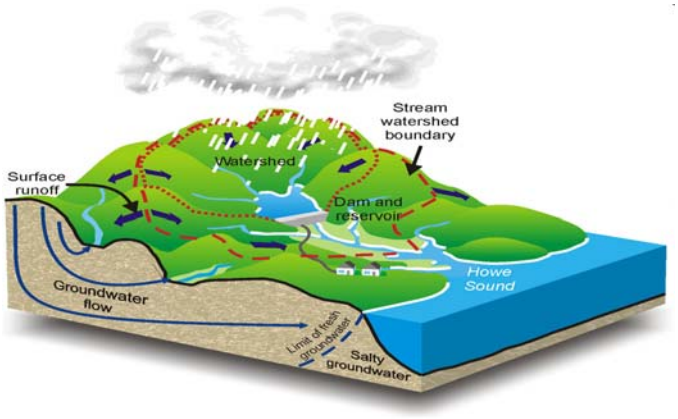


Figure 5: Hydrologic Watershed

One significant obstacle to this process involves a pit-filling process, which has been problematically slow for a long time. The ideas introduced in this section have been partially published in a conference proceeding (Dunsford & Ames, 2007), and represent an optimization on this problem. The basic idea of a pit is a depression that is deeper in the center than it is on any of the edges. Figure 6 Illustrates an example of a pit.

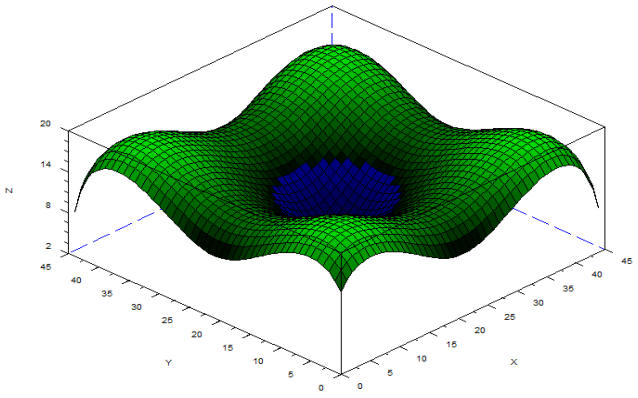


Figure 6: The Illustration of a Pit

If water is poured on this pit, it would pool up in the pit until it eventually poured over the edge. While in nature water frequently simply sinks into an underground water table, it is common for the depth of the water table to be determined by features visible in the surface terrain (Davis & Cornwell, 1998). The use of terrain models to predict the flow of water relies on the idea that water will flow down hill (Hwang & Houghtalen, 1996). Strict downward moving techniques encounter a problem with pits because in nature this water almost always makes it to the streams and rivers eventually.

Historically, elevation has been mapped using contour plots of equal elevations, and those, in turn, have been traditionally used to delineate where water will likely travel. This is called watershed delineation (Alarcon & O'Hara, 2006) and is one of two areas where a *PitFill* algorithm became widely used. As geospatial software becomes more affordable and prevalent, the techniques that accompany that software play an increasing role in the fields that use geographic information. At the time of this study, even though contour maps can still be created, the raw form of elevation information is typically stored in a Digital Elevation Model (DEM). DEMs are scalar fields, where each value represents the average elevation for a rectangular region. They have many uses,

especially involving watershed delineation, and drainage modeling, for example (Soille et al., 2003).

In hydrology, *PitFill* is used in order to delineate watersheds. In soil sciences, *PitFill* was first used to study the Depression Storage Capacity (DSC) for materials on a much smaller scale. While hydrologic models typically work in the range of each cell being 30 meters or more, the soil sciences typically are studying cells only 1 mm in size. Regardless of the application, it became apparent that a computational method was necessary. While the algorithms used by professional software packages are generally proprietary, there have been numerous publications on the topic, and improvements in the technique can be followed with them. Some of the published methods run faster than the commercial software, which might be using out-of-date approaches.

Previous efforts to solve the problem involve several important steps. Starting with Marks, Dozier, and Frew in 1984, a very basic technique was introduced that involved a 3 x 3 filter for detecting pits, but it frequently missed important larger pits that could not be easily identified by the initial pass of the 3 x 3 filter. In 1988 Jenson and Domingue published a fairly long and involved process. This process involved not finding pits for their own sake, but rather as an incidental result of a more involved watershed delineation process, and had a time complexity of about $O(N^2)$, which is very slow for large DEM grids that can have millions of cells. The same year other code was being published with similar performance characteristics to determining catchment areas (Martz, 1998). In 1997, Dr. David Tarboton implemented watershed delineation methods that were incorporated into the earlier versions of MapWindow (Tarboton, 1997). Planchon and Darboux introduced an algorithm in 2002 that has a fast “Dry Upwards”

method that works about as quickly as a paint method for a large part of the image (Plancheon and Darboux, 2002).

2.3.1. Dry Upwards Implementation

The first step in implementing this process is to start by "marking" the entire region as though it had water at a level that is higher than the highest point of elevation.

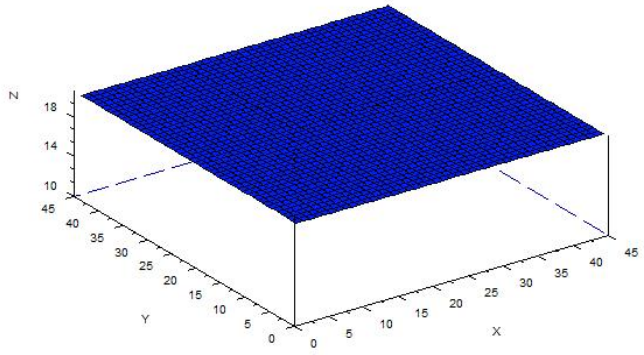


Figure 7: Completely Flooded Starting Point

The objective is to allow water to "drain" from the cells in this raster, revealing the natural level of the terrain beneath. To start with, only the outside edges are considered. The rule is designed so that any cell on the perimeter is allowed to drain outwards, regardless of the elevation of that cell.

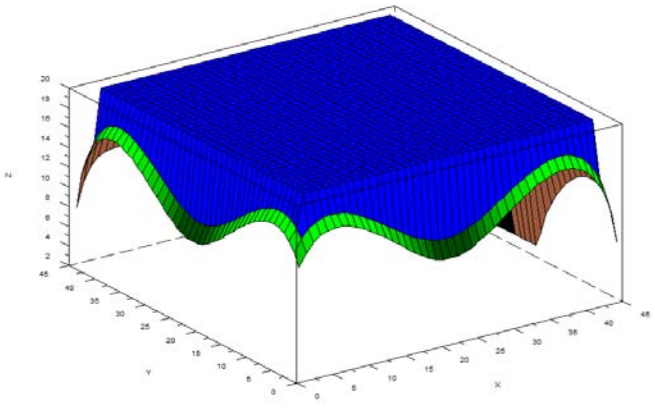


Figure 8: Perimeter Cells "Dried" out

When considering the next cells within, they when compared with the recently dried border cells; they can be either higher or lower than the neighboring cells that have already been dried out. Figure 9 shows the condition where the inner cells have a higher elevation than the cells that are at the edge of the raster.

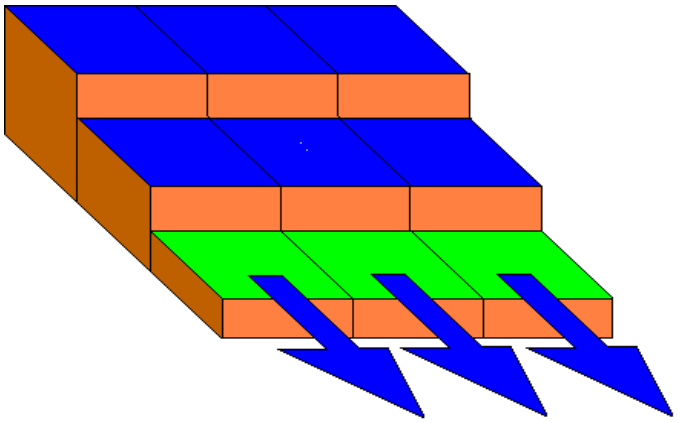


Figure 9: Higher Elevation Interior

As the water is "drained" the output cell is marked as having been successfully drained because it is the same elevation as the original cell. In the case where interior cells are higher than the border cells, the water is allowed to drain via the neighboring cells.

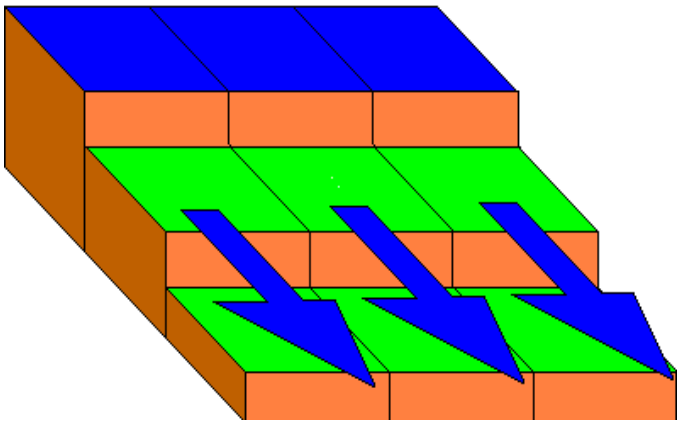


Figure 10: Drain Higher Neighbors

Consideration of each of the neighbors of the perimeter cells can be visualized by

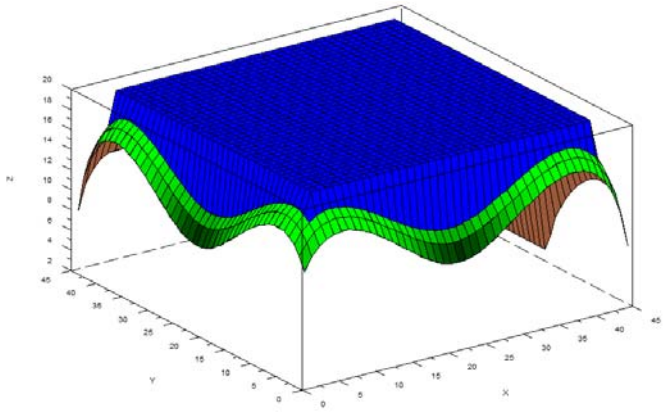


Figure 11: Dried Neighboring Cells.

This process is allowed to continue until eventually all the untreated cells that remain in the image are classified as some kind of pit. Because the process described only considers the neighbors of cells as they are dried, and only considers those neighbors if they are still wet, the process is very efficient. Up to this point, the algorithm introduced in this research replicates this very efficient process. The only distinction between our implementation and the published recursive algorithm is that we use a cue for storing cells that need to be considered. A recursive process may work, but relies on a very small amount of memory available in the call stack, and generally makes extremely poor use of that memory by having duplicates of any local variables that exist in the method itself. This almost certainly creates a "Stack Overflow" exception in all but the smallest of grids. This was replaced then by a simple cue that stores the "wet" cells that need to be considered. It is possible for a wet cell to be added to the cue more than once, so a check is added. If the cell from the cue is being considered has an output value that is the same as the input value, then this cell is skipped. Otherwise, it checks each of the 8 neighbors to see if any of those cells are still marked as "wet" by having an output value that is larger than the original elevation. Any cells that are still wet and also have a higher elevation are then added to the cue. Cells that are wet but have a lower

elevation are ignored. Eventually Figure 12 illustrates the situation where all of the remaining wet cells have elevations that are below the neighboring dry cells.

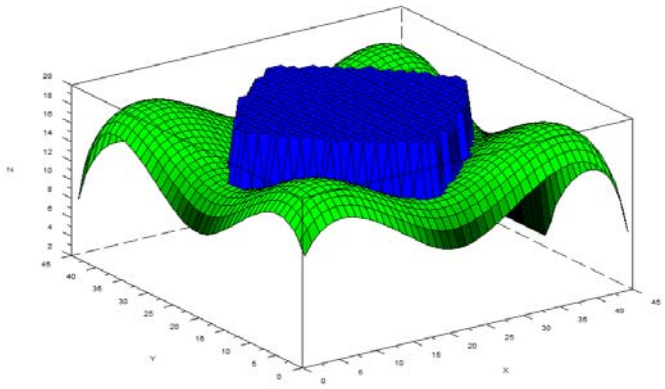


Figure 12: Only Pit Cells Remain

2.3.2. Pit Handling Improvement

The question then remains as to how best to handle the remaining elevations. In rare cases this might be a large lake or basin, but most of the time, the discovered pits will be small and scattered. The approach proposed by Plancheon and Darboux is to perform many successive raster scans in different directions. If the scan encounters a wet cell, and that wet cell has a neighbor with a water level or dry elevation that is lower than itself, it is allowed to drain to that level.

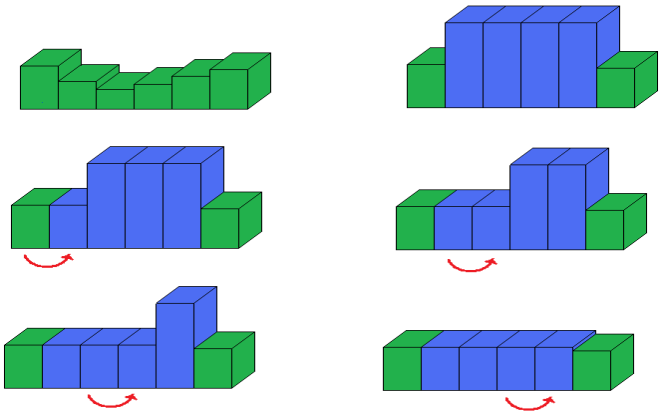


Figure 13: Raster Scan Method

The advantage of the process is that the raster handling doesn't require storing any values in memory. However, this is in direct contradiction to the earlier process, which requires keeping track of the active cells that are being dried out. Meanwhile, there is a noticeable problem in performance for this stage. While before, each cell was being considered only once, now suddenly every cell is being considered a minimum of eight times (one for each cardinal direction) and potentially dozens or hundreds of times for complicated winding river channel shapes.

This dissertation explores a two-part improvement on this original strategy (Dunsford & Ames, 2007). The first improvement eliminates this highly redundant and recursive step for addressing the cells that have been classified as pits. Instead, we simply allow the same process to continue, but with slightly different rules. Instead of only allowing cells to "drain" completely, we allow them to drain to the level of a neighboring cell. In the original dry-upwards method, when considering a cell in the cue, if the cell is already dry, we skip it. If the cell is wet, then each of the neighbors was considered. If the neighbor is wet and higher than the current elevation, the neighboring cell is then added to the cue. The new process has to deal with potentially different elevations, and so the neighboring cell is added to the cue, but instead of simply storing the row and column information, we also store the water level. Then, instead of simply checking to see if the cell is already dry, we check the output elevation against the water level stored in the cue. If the water level in the output cell is lower or equal to the elevation stored in the cue, we skip the cell. Otherwise, we consider each of its neighbors, adding them to the cue if they store water levels that are higher than the current level. Ultimately, this process might discover some high elevation cells, like an

island in the lake, where the original dry-upwards process can be allowed to continue. If dry cells are discovered, then the dry-upwards process is allowed to run as much as possible. This adaptation handles the cases of pits in a river valley. It may be possible for an entire watershed to be unable to "dry-upwards" because of a relatively small pit in the valley. The Plancheon and Darboux method would then require raster scanning the entire watershed, which would be incredibly inefficient, and the situation is not unlikely to occur in naturally occurring elevation data. Figure 14 shows an example of one scenario where this would be true.

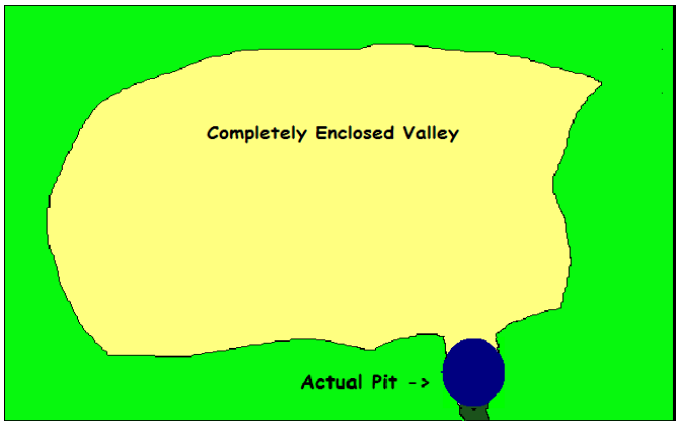


Figure 14: Enclosed Valley Super-Pit

By passing back and forth between the regular dry-upwards process and the modified process, eventually, all the cells in the pits will obtain the best values. Figure 15 shows the elevation values propagating through the pit region. The cyan cells represent the optimal value. The red cells represent border elevations that are higher than the optimal value.

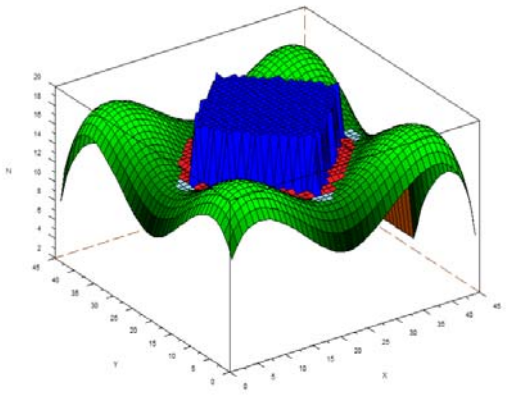


Figure 15: Addressing Pit Cells

Figure 16 shows the image of the terrain after all of the original high-water values have been replaced.

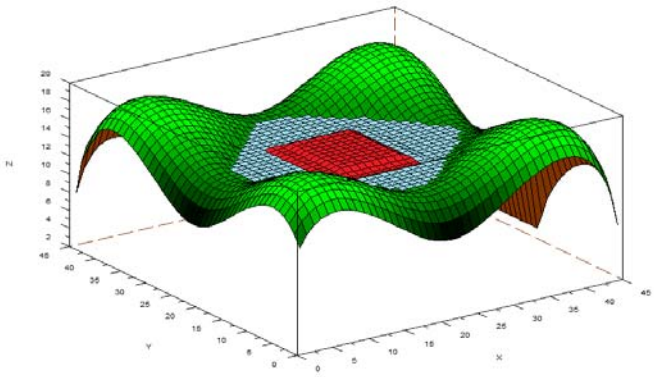


Figure 16: All High Water Levels Replaced

The process will not end here, however, because the red cells are recording elevations that are higher than their cyan neighbors. Eventually the optimal cyan value is allowed to spread through the entire pit.

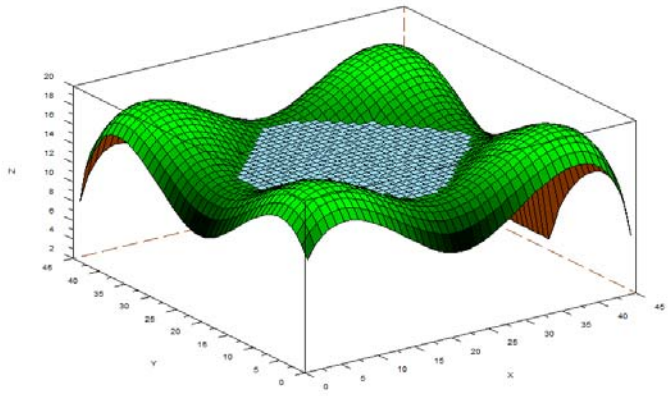


Figure 17: Final Result

There is a subtle alteration to the original dry-upwards rules that we added in order to improve the efficiency of this portion. The default propagation of values through the pit might look like what is shown in Figure 18. The result is that rather than each cell only being considered a single time, each cell may be considered in a worst case scenario once for each different elevation value on the edge of the pit. This can lead to a fairly redundant process. The pits are typically much smaller than the size of the grid itself, and therefore this method is still less redundant than the Plancheon and Darboux method.

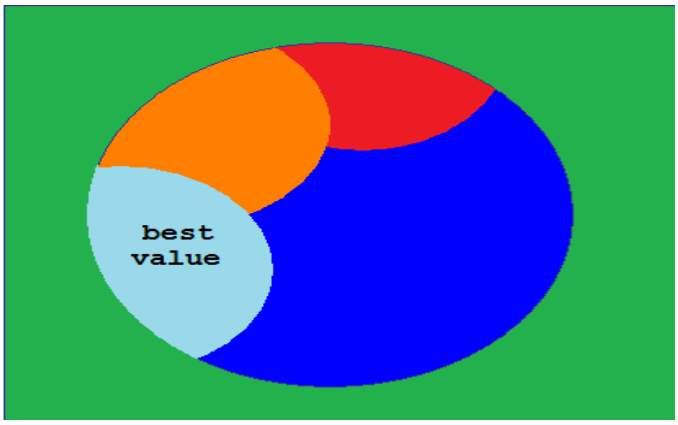


Figure 18: Default Propagation of Values

An optimization that we introduced to reduce the redundancy was extremely subtle. Remember that the process we described earlier considers each of the neighbors, adding them to the cue if their levels are higher than the water level of the current cell. We added a subtle check. If one of the neighbors has a water level below the current

level for the cell, then instead of adding all of the immediate neighbors to the list, this value is added to the list using the lower value from its neighbor. The effect of this is subtle. It means that we will spend much less time spreading non-optimal values. The reason is that the optimal values will spread not only on their own turn, but also during any of turns of the neighbors while considering the cue. Meanwhile, the non-optimal values that are immediately discoverable as non-optimal don't spread at all. Allowing values to spread out of turn creates a process that fills pits like Figure 19.

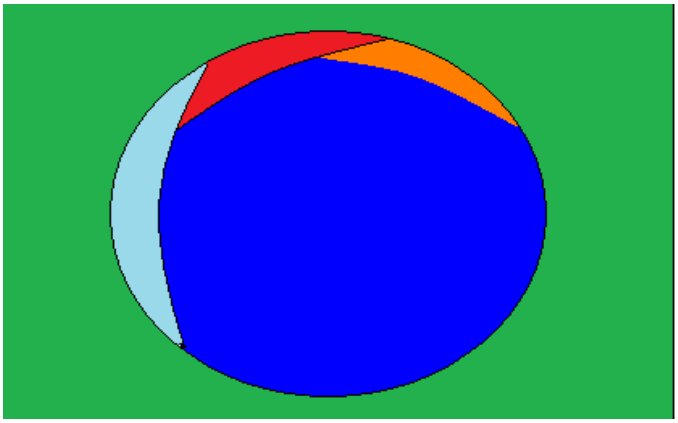


Figure 19: Less Redundant Propagation.

2.3.3. Piecewise Implementation

The final problem that we addressed was the memory criteria. Because we are using an in memory cue, and also because we are loading the terrain raster into memory for consideration, there is an upper limit on the size of the raster that we can consider using this approach. The solution is to break the process into rectangular grids. There should be exactly one cell worth of overlap on the interior portions of the separate section.

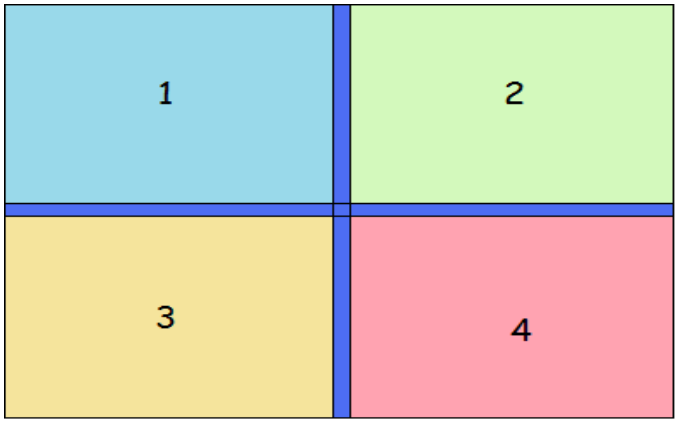


Figure 20: Subsections with Overlap

The algorithm has dependencies. Cells in the blue section might eventually drain through the green and pink sections in Figure 20. However, we cannot simply allow water to drain freely from the interior edges. The solution is to create an output file that keeps track of the output elevation values of the overlap regions. We used two rasters. One that has as many rows as our total source raster, but only as many columns as we have overlaps horizontally. The other has as many columns as the original raster, but only as many rows as the number of vertical overlaps. When considering the quadrant 1 in Figure 20, for instance, water is allowed to flow freely from the top and left exterior borders. The interior borders, however, record the final elevation of the water levels after being allowed to drain from the exterior borders of this quadrant. When the adjacent green quadrant 2 is considered, water is allowed to drain through either the top or right edges as with the first section, but water is also allowed to drain through the interior edge. Since we have recorded the working water level for each of those cells, they can be considered along with the exterior borders as the starting point for the dry-upwards or pit-filling methods, which both use a flood-fill like propagation through the new quadrant. It is now possible that cells that were recorded as a pit on the shared border are allowed to flow through one of the other directions. If this happens, then the quadrant is marked to

be revisited. Revisiting a quadrant only considers the changed cells. The modified pit-handling algorithm, or else the dry-upwards algorithm will use the cells that have changed on the cell border as the starting point. These can either be cached or else a test can be performed on the perimeter cells, comparing the overlap cells in the border file with the output cells. This may be slower, but will consume less memory.

This is a fairly straight forward treatment for attempting to perform the algorithm in a piecewise fashion. However, it should be pointed out that without considerable alteration the original Plancheon and Darboux process would not have worked well in a piecewise approach. The raster-scan method would have to be performed across the entire composite raster over and over again. The performance in such a case would be much slower than the method proposed in this paper for enormous terrain grids. Also, there would have been no strategy for storing intermediate values in order to track dependencies between raster grids. In our case we necessarily revisit some grids, but when revisiting a grid, we only need to revisit the small portion of cells that are still wet, and don't need to re-consider the entire grid surface.

2.3.4. Pitfill Summary

The *PitFill* method used by the Terraflow group in 2003 (Arge et al., 2003) is directly comparable in performance to the proprietary ArcInfo calculations and also able to handle large grids. The method created in this study is called the fast, piece-wise algorithm. However, the new fast piece-wise method introduced as part of this dissertation is approximately 20 times faster than either of these methods, making it more than a full order of magnitude different from previous solutions.

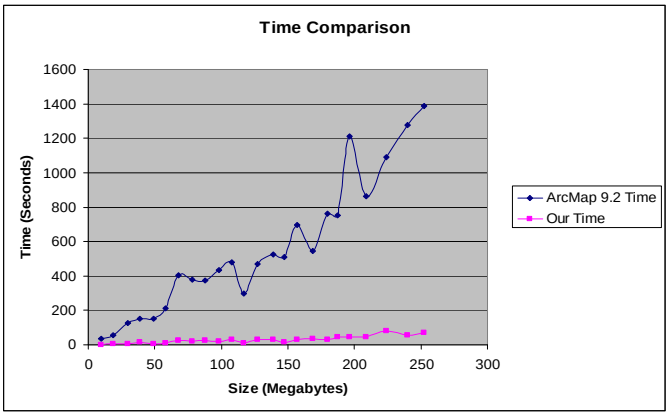


Figure 21: Performance Comparison

For grids that are too large to fit in memory, the *piece-wise* algorithm becomes more applicable. The fast, piece-wise method required approximately half an hour to process grids in the gigabyte scale instead of approximately 30 days for Terraflow, or not even being possible for ArcInfo.

The first change necessary to make the new algorithm work in an O(N) piecewise process is to remove the raster-scan element from the analysis. Instead of performing a raster-scan process to find pits, the dry-upward process is simpliy continued across the pit. Like a paint algorithm, each cell affects its neighbors. Like ripples on a pond, successively lower values spread across a pit. Because of a special ordering of the algorithm, the lowest value propagates significantly faster around the edges of a pit than it does the center. This means that even without knowing the actual dimensions of the pit in advance, the amount of time that is spent propagating larger values into the pit is minimized. The benefit is that a raster scan is unnecessary to search for pits and nor is it needed to re-consider the cells once they have a proven escape path for water to flow.

The solution presented here uses a method of recording values at the edges of a window and allows the process to be run for a given window, starting at an edge. When a window is tested adjacent to a previously processed region, the shared interior edge can

be treated like an outside edge. This method allows block portions of the image to be processed separately. It is possible that a block may require processing more than once. As such a secondary treatment starts with the updated cells on the interior edge and only needs to process cells that need to be re-calculated, rather than every cell in the given block. The final processing approximates O(N) performance, since each cell is only considered once.

O(N) is an example of notation referred to as “big O” notation by software developers. Processes that are called sequentially usually scale linearly. That is, if a method is called 100 times, it will take 100 times longer than calling the method once. Because complex methods may work with members more than once, the scaling may not be linear. Big O notation reflects the approximate order of magnitude relative to the number of members that the algorithm works against. In this case, when the grid is broken into more pieces, the performance deteriorates slightly because of the increasing need to re-process squares, but even then only the windowed regions would be processed a few times at most. The worst case performance is about $O(N^{1.5})$ because there are sometimes large craters that have several passes before the best value is found. However, in most cases, the performance will be better approximated as O(N).

3. GIS and Open Source Software

Many areas of environmental modeling, civil engineering and geosciences all depend on software that can render and analyze spatially organized information. (Bolstad, 2008) There are numerous examples of GIS software applications that are contemporary with this study, and there has been a recent increase in the number of open source GIS projects. An illustration of the numbers can be seen at FreeGIS.org where the

applications listed on that website alone include 349 free GIS software packages using 36 different license agreements and more than 15 programming languages (www.FreeGIS.org). The role and function of these software packages can range from exposing data access methods (eg., PostGIS, GDAL), to performing two-dimensional topology calculations (e.g., Net Topology Suite, GeoTools), to desktop GIS applications (e.g., GRASS, QGIS, MapWindow), to web mapping services (e.g., MapServer, uDig, Open Layers). Even among open source licenses, there are different levels of obligation. The General Public License (GNU 2007) comes with the most restrictions on what derivative works can do with the software, forcing anything that uses the open source material to in turn become open source themselves. Licenses like Mozilla (Mozilla Foundation) fall at the other end of the spectrum, allowing commercial use of the software so long as proper reference is made to the inclusion of the open source components (Steiniger & Bocher, 2008).

3.1.Open Source Software

The term open source software (OSS) gets its name from exposing the source code before it is compiled into binaries or machine language. “Some of the most striking features are that OSS development is created from the contributions of volunteer programmers, that these programmers only associated with each other through informal communities, that the resulting software products are made available for free, and that this unconventional development method is able to reproduce software of high complexity and extraordinary quality” (Bitzer & Philipp, 2006)

Many agencies, especially government agencies, which desire to freely disseminate information, are looking to software solutions that are open source and

transparent. These solutions are cheaper to develop, freely distributable, and can be used as foundations for future projects. They also provide transparency for improved accountability (von Krogh & Spaeth, 2007). The Environmental Protection Agency's (EPA's) Better Assessment Science Integrating Point and Non Point Sources (BASINS) project and the Federal Emergency Management Agency's (FEMA's) Digital Flood Insurance Rate Map (DFIRM) project are both examples of environmental regulators' increasing interest in transparent software. The EPA also does not want to design special purpose software on a framework that will vanish completely when a commercial vendor decides to stop supporting it.

Open source software presents many economic advantages over proprietary software. It is cheaper to write, maintain, distribute and update because all those tasks are shared by the entire community using the software (Waring & Maddocks, 2005). Open source software solutions are certainly cheaper to the users, who routinely download the software for free (Fitzgerald, 2004). Agencies that need specialized software often require an entire team of programmers to develop the application. The alternative is working with existing software that is frequently a poor fit to their specific needs and expensive to keep properly licensed. The open source solution can offer a valuable alternative to these options. Agencies can hire developers to work on the project, but have the developers focus on areas that are especially important to the agencies, while still benefiting from the existing code supplied by the community (Weber, 2004).

These initial obvious economic advantages often come with a hidden cost. Many times open source software is written for an unconventional platform, e.g., Linux. While the cost of the platform is free, it frequently requires companies to hire a specialist in

order to install and maintain the software. Free applications may not be as well documented or completely functional, requiring a greater cost for training or learning to use the software. While some technical support may exist in the form of forums or direct contact with the developers, in general, the agencies publishing open source software do not have the resources or infrastructure to support the large call-centers or user-training seminars that proprietary software is able to support from the profits.

Open source software is highly reusable (Blind & Edler, 2003). As long as the code is modular, fully functional pieces of the code can be used or interchanged with newer, more advanced versions. Entire core libraries can be used over and over again. A library for making Audio Video Interleave (AVI) files can be shared by a wide variety of software, not just the original project that the library was created to support.

One of the most significant improvements that open source software offers over closed source solutions is that open source software is transparent. Transparency means that every line of code can be analyzed directly to see if it is working correctly. It becomes especially effective with some form of version repository.

It is widely thought that expensive software captivates the cutting edge. Certainly for video games that is the case. However, scientific software is unlike a video game because what is being represented is usually cutting edge science, not cutting edge graphics. Indeed, interested individuals world wide can participate in the areas that most capture their interest, enhancing innovation (de Joode, 2004). With open source solutions, scientific contributors can benefit the selective community immediately, without waiting for the next version of a public software release, which can take years.

Further, the developers in open source projects tend to interact through a community. Developing software has an inherent complexity. A principal known as Brook's Law states that "Adding manpower to a late software project makes it later" (Brooks, 1995, p. 25). Simply put, division of labor is less effective with complex, interdependent tasks. In order for a new developer to be brought up to speed, normally productive members must devote their time to teaching the new arrival. The interaction points between the parts of the software that are written by different people tend to add complexity to the project as well. The end result is that adding developers tends to add extra complexity, which frequently slows down a project that is already behind. Raymond describes software projects as either following an orderly development plan, like building a cathedral, or else as a wild collection of different contributions, like a bazaar (Raymond, 2001). Open source projects frequently start out like a cathedral, but over time, they end up combining the works of so many different contributors that they become more like a bazaar. This collection of different contributors with very different perspectives or specific areas of expertise would otherwise never be in contact with each other. "To work successfully, the geographically dispersed individuals as well as small groups of developers must have well functioning and open communication channels between each other, especially as the developers do not usually meet face-to-face" (Abrahamsson, Salo & Ronkainen 2002, p. 78). "Beyond the source code itself, open source projects tend to allow direct access to all software development artifacts such as requirements, design, open issues, rationale, development team responsibilities and schedules" (Robbins et al., 2005, p. 3).

3.2.Open Source GIS

At the time of this study, there are hundreds of small, geospatial software projects, and a few of those build on a history that is decades old. There are also several organizations that tend to the distribution of similar projects over the internet (e.g., FreeGIS.org, SourceForge.net). It is impractical to discuss the complete scope of existing projects, but a few examples of the organizations and existing software projects are discussed here to fill in a picture of the extent and nature of existing projects.

3.2.1. Organizations

In addition to individual projects, there are organizations that see to the adoption of abandoned source code as well as developing standards that can be applied across many different projects. The Open Geospatial Consortium (OGC), founded in 1994, provides standards that attempt to improve the ability of geospatial software to work together, and increase the familiarity for users working with different GIS projects by specifying common nomenclature. The consortium members also specify open data formats and web protocols to improve the ability of different programs to directly share information. Other organizations like the Open Source Geospatial Foundation (OSGeo) provide financial, organizational, and legal support to the broader open source geospatial community. OSGeo was formed in 2006 and possesses goals that extend beyond software, including exposing government geospatial data and hosting education and training. The consortium members also maintain source code that is no longer being updated by the authors, allowing useful code bases to be expanded and updated by future developers. While largely community based, the members currently maintain more than 20 GIS programs (e.g., GeoTools, GDAL/OGR, gvSIG, GRASS GIS, Quantum GIS,

MapServer, OpenLayers, MapGuide) but a more complete listing can be found on their website (www.OsGeo.org).

3.2.2. GRASS

Successful open source projects like the Geographical Resources Analysis Support System (GRASS) (Neteler & Mitasova, 2008) accumulate additional libraries and capabilities over time. GRASS is currently an accumulation of decades of source code contributions made to their repository of C language libraries. GRASS is a combination of several separate libraries that can work with geographic raster and vector datasets for visualization, image processing, and analysis (Blazek et al., 2002). GRASS adopted the GNU GPL General Public License, (see http://www.gnu.org) in 1999. The difference between this license and other more tolerant licenses is that it places strict limitations on derivative work that would use the software. It forbids proprietary software from linking to or mixing with its software. It also prevents modifications (or derivative work) to be taken private and not returned to the public, and prevents release changes from being released under a different license. By way of comparison, the GNU Lesser General Public License (LGPL) does allow linking. Licenses like BSD, Mozilla and MIT are even less strict, and allow release changes to occur under different licenses and allow modifications to be taken private (Steiniger & Bocher, 2008).

GRASS supports both a Graphical User Interface (GUI) for their software as well as allowing almost all of their processes to work through command-line execution. It is fairly straight forward for developers to extend the GRASS environment because the source code is available and they can simply write their own library files before compiling the new project. An example of this process might be adding new libraries that

can compute solar radiation incident to the Earth's surface (Hofierka & Suri, 2002). One disadvantage for Windows users is that the entire project was developed for the Linux operating system. Projects written for the Linux environment can be compiled in Windows if enough support libraries are also downloaded and built into the project. In the case of GRASS, about 28 steps are required before compiling GRASS on a Windows machine, most of which involve downloading and unpacking applications or libraries. Once the necessary dependencies are downloaded, developers still can not compile those libraries from Visual Studio, but rather must use C compilers external to the .NET developer interface.

The strength of GRASS is that it has a longevity that proves it is a system that works. There are also efforts to connect these libraries with other open source projects like the R statistics package (Bivand & Neteler, 2000). Though the learning curve is steep, it is clear that the GRASS libraries provide a significant amount of functionality that otherwise would only be available through expensive proprietary software. The source code repository is extensive enough that it can be thought of as the textbook example for a wide range of analysis tools. The project also brings together many open source ventures that are built and maintained independently from the GRASS development team. One example is the GDAL libraries, which are maintained by Frank Warmerdam. Another is the PROJ.4 library, which handles projections. The GRASS libraries are also consumed by alternate desktop GUI software programs such as Quantum GIS (QGIS).

3.2.3. GeoTools

GeoTools (http://geotools.codehaus.org) is a java based toolkit that other developers can use for geospatial operations. The community of software developers working on the project has a large number of members and the toolkit provides access to much of the topological, projection and data handling. Libraries built on top of that package can focus on the user interface and project specific tasks. The Geotools libraries use OGC ontology and the GNU LGPL. The innovative, object oriented geometries correspond to the Simple Feature Specification (ISO19107:2003) that describe vector data stored as geometric objects. The object oriented behavior of these libraries mimics the geometric terminology of features, so that a polygon becomes an object that contains the coordinates and some derived characteristics of that polygon. The polygon object also exposes properties, like an Envelope, which describes the rectangular extents that completely contains the polygon. Methods are also exposed that use the polygon as an automatic argument. The most important of the methods are the *relate* and the *overlay* methods. For example, if a road is represented by a LineString class, then the intersection method on either the road or a county polygon will produce a new LineString that represents the road shape, but cropped to fit inside the polygon.

GeoTools is written for the Java language. Java was originally developed by James Gosling at Sun Microsystems and released in 1995. It is a free development library, is completely object oriented and supports built in memory management and multi-platform support (Gosling et al., 2000). Java is incompatible with most other languages, however, so that developers wishing to use the GeoTools library will most likely also have to be developing in Java. This incompatibility is contrasted by

Microsoft's COM platform that allows almost any windows-based language to work directly with libraries produced in almost any other windows-based language. The COM platform introduces some performance considerations, which will be discussed further in section 2.

3.2.4. SharpMap

SharpMap (http://www.codeplex.com/SharpMap) is a C# project protected by the GNU LGPL. The developers of the project are also working from a port of the Java Topology Suite to the C# language. Their principal objective, as far as the 1.0 version of their software was capable of supporting, was to extend the topology suite with sufficient visualization tools for developing web controls. There are projected efforts for a 2.0 version that can extend their project to have full desktop functionality. However, the core of the project that supports projections and topology operations is obtained through the Net Topology Suite, which is effectively the same starting point as the MapWindow 6.0 project. SharpMap is poorly documented and does not have a very strong user base. To date there is no mention of any effort to extend the project to work with 3D maps.

3.2.5. NASA World Wind

The NASA World Wind project (http://worldwind.arc.nasa.gov/) is a DirectX-based three dimensional globe view of the earth that supports displaying vector and raster data. It supports both C# and Java based source code, and the C# code is compatible with the .NET framework and CLR compliant. World Wind's license allows derivative commercial work to be created from it without forcing the entire derivative work to become open-source. It supports 3D visualization through the DirectX libraries. It is

already set up to support many data formats, including Shapefiles, Jpeg images, Bitmap images, and others.

The globe interface is especially useful for 3D visualization and resembles Google Earth (http://earth.google.com/) or Microsoft's Virtual Earth (http://www.microsoft.com/virtualearth/). Google Earth and Microsoft Virtual Earth are both proprietary, client-side applications that allow for the on-demand access to large repositories of image data. The data, largely purchased from satellite image providers such as DigitalGlobe are copyrighted by Google or Microsoft respectively. The data that are available through the NASA World Wind project, on the other hand, is part of the public domain and free to use and modify.

The disadvantage with programs that focus on a globe view is that while globes are especially effective at rendering 3D terrain data and showing off stunning satellite views, the shape of the globe and three dimensional terrain makes it somewhat challenging to perform actual GIS analysis. The advantage of a simple, 2D overhead view is that users can very quickly establish their bearings, easily select vector features for editing and control the editing process. Selecting or editing in three dimensions is possible but generally requires a much more sophisticated interface that includes multiple views and detailed ability to control those views. The underlying focus of their project is display, and not analysis. There appears to be no real plan to modularize the software. While it is possible to write a "plug-in" for the globe application, the aspects of the project that are accessible from that interface are limited, and the possible functionality that a plug-in can provide is not open ended.

3.3.MapWindow

3.3.1. History

The MapWindow GIS (http://www.mapwindow.org/) (Ames D. et al., 2008) has versions that predate this project. The active version (discounting the project for this dissertation) is the 4.8 version, which is a two dimensional desktop GIS application with approximately 10,000 downloads per month. Figure 22 shows an example of MapWindow 4.7.

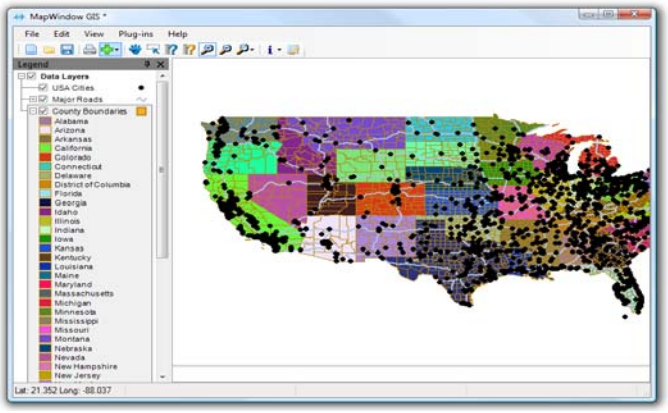


Figure 22: MapWindow 4.7

Funding for the project is currently supplied by the U.S. EPA BASINS project, FEMA's DFIRM project, and lately the NSF funded (CUAHSI-HIS) project (Consortium of Universities for the Advancement of Hydrologic Science, Inc.) The MapWindow project was first established in 1998 at Utah Water Research Lab in Logan as an alternative to using MapObjects LT 1.0. The major issue, of course, was that the proprietary controls, though free to use, prevented users from modifying the underlying data, thus rendering it only usable as a viewer. The requirements of the Utah Water Research Lab at the time included being able to dynamically alter vector data, or access the low level grids. The early MapWindow team created the core MapWinGIS.ocx component, an ActiveX control that could provide low level access to the data formats

that developers could then rapidly turn into successful projects. As the number of GIS developers re-using the ActiveX map increased, frequently these developers would need the same common features that surround the map such as a legend control or selection, or even simply buttons to change the mode that the map is in. Each project that used the map tended to duplicate the same project design over and over again. Each project added a legend, status bar, table viewer and so on. To simplify things, instead of starting over each time, a single project was created that was highly extensible. That is, the starting point for the project already included many standard GIS display, editing and data access functionality, which could be extended to perform new custom tasks. The fully developed project is called MapWindow 4.x. There is a user's manual for this software (Watry, G. 2007), as well as one for working with the programming library (Ames. D. P., 2007) and it is described among other open source applications as part of the Open Source Geospatial (OSGeo) organization (Ames, D.P., Michaelis, C., & Dunsford H., 2007)

At the time of this dissertation, the MapWindow project is open source and being used by both users and developers. While the original project was focused around an ActiveX control (an older Microsoft technology exclusive to the windows platform), the new ideals are to remove any reference to COM libraries like the ActiveX control. The MapWindow6 project is focused on developing modular components written in C#. Since everything written for MapWindow 6.0 is in a managed .NET language, it will be easier to use the same library with other operating systems or for web applications. Mono supports .NET framework code, and is not restricted to the windows platform, which

means that as long as the project is entirely written in .NET, it will be portable to other languages.

The MapWindow 6.0 version of the project began in the summer of 2007. In GIS there is a frequent need to study topology. The tools that were provided by the geoprocessing library were being re-developed by the author to include a more complete, object oriented vector topology toolkit. In the fall of 2007, the team, consisting at the time of Dr. Daniel Ames and the author, was directed to a toolkit that already existed, and had even been ported to C#. The introduction of the Net Topology Suite into the project would require such a serious re-thinking of the underlying objects that it proved beneficial to start a new version of MapWindow. The first step in the process was adding new interfaces for the NTS classes. Ultimately, the classes proved insufficient for rendering performance, and so are created strictly for the topologic analysis that they are designed to do.

4. Project Requirements and Considerations

The basic idea is to support two and three dimensional GIS. An example layout for the two dimensional version of the controls is featured in Figure 23.

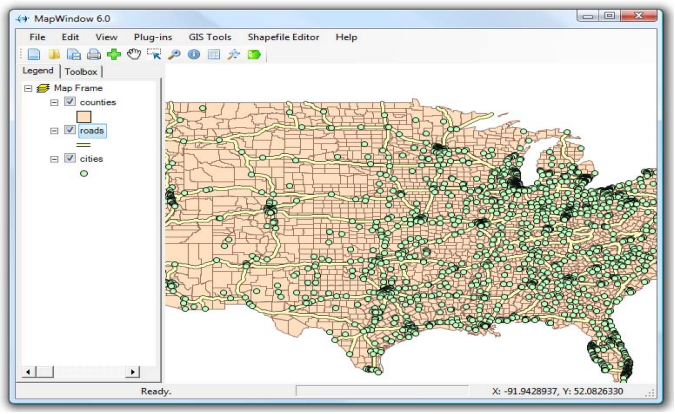


Figure 23: MapWindow 6.0 (2D map)

The primary research objective for this project is the design and development of an extensible, interchangeable, component-based architecture for open-source geographic information systems. The framework introduced by this project is designed to support a toolkit for engineering applications, especially involving hydrology. The research contributions for this dissertation follow along the lines of either solving architectural problems of software design, or else optimizing methods that are important to either the GIS facet of the software or the hydrology toolkit aspects. In order for this dissertation project to be able to satisfy the goals of acting as a geographic information system, it must have some basic capabilities: to read and write geographic datasets, display those datasets, allow for analysis of those datasets, and ultimately supply the user interfaces so that end users can directly access all of these features. One additional criterion for this particular project includes supporting extended data formats, processes, and plug-in behaviors through an innovative interface based add-in architecture. Other criteria include optimizations that make it more feasible to render large vector datasets and analysis tools specific to hydrology.

- *Read and Write Data*
  - Vector, Raster and Image formats
  - File, Database and Web Data Sources
  - Temporal & 3D Support
- *Display Symbolic Representations*
  - Vector & Raster Symbolizers
  - Thematic Layers
  - Feature Selection
  - Cartographic Categories or Schemes
  - Triangulation for 3D
- *Analyze Topologic Relationships*

- Relate
- Overlay
- Topology Algorithms
- Watershed Delineation Algorithms

- *Map Capabilities*
  - Zoom, Pan
  - Selection
  - Vertex Editing
- *Legend Capabilities*
  - Remove or Reorder Layers
  - Access Layer Properties
  - Open Attribute Table
  - Check or uncheck visibility
  - Access Symbology Editors
- *Other User Interfaces*
  - Toolbox Component
  - Attributes Table Editor
  - Raster and Vector Symbology Editors
  - Temporal Controls Component

The following sections detail aspects of the entirely new GIS libraries that have been constructed in C#. While some long term goals remain incomplete at the time of this dissertation, the project consists of over 82,000 executable lines of code, with approximately 18,000 lines of code originating from the incorporated methods of the Net Topology Suite that handle topological analysis, or from other co-developers on the project. The methods, properties, events and arguments are all commented in a way that will export them to an xml file and provide information to developers through IntelliSense. The xml file of comments for the 2D project is more than 250,000 words. The 3D extension is comparatively light weight. It consists of about 6000 executable

lines of code and includes about 16,685 words (about 66 pages) worth of comments. By design, the 3D module is a light weight optional library because of the DirectX requirement, which is not portable to other platforms.

4.1.Data Formats

Data formats are handled as one of three different types: vectors, rasters, or images. Vector formats are complex, but basically combine some kind of geometry, like a polygon, with non-spatial attributes. An example would be to associate a polygon representing a county with the non-spatial data of its population. A single instance of this pairing is known as a feature. So in the data handling there is a class called a *Feature* that can represent a pairing between a geometry and *DataRow* of attributes. A *DataRow* is simply one row from a .NET *DataTable*. The geometry can be a *Point*, *LineString*, *Polygon*, *MultiPoint*, *MultiLineString* or *MultiPolygon*. In older formats, multipart polygons could be either solid polygons or holes cut into the polygon and the part or hole classification was encoded in the file format by using clockwise or counter-clockwise ordering for the vertices. In the OGC format, however, each polygon stores a list of holes, where holes are defined as being contained by the polygon. Each polygon also stores a single outer shell. In order to create an island chain, like the state of Hawaii, several separate polygons must be used. The need for joining separate polygons into a single feature is the reason why the geometry for a single feature might be a *MultiPolygon*. More recently designed vector data formats also tend to include some method of spatial indexing that will be discussed in detail in the special section on kd-tree generalization.

Raster formats can store values such as doubles, floats, integers, shorts or bytes in a two-dimensional grid that has rows and columns. Each separate grid is a single band.

Raster data formats follow the idea that a single raster can be either a single band of image data, or else it can consist of several independent bands of data, but each band in turn is also a raster. For example, a hyper-spectral image might have recorded the same image in hundreds of different spectral wavelengths. Each spectral frequency would appear as a separate band in the dataset. The human eye can be tricked into perceiving color with three bands, taken from the red, green and blue parts of the spectrum. Any more spectral bands represent too much color information to be encoded in a color image, so hyper-spectral images often take advantage of false coloring spectral bands that are normally not visible to the naked eye. Similarly, raster datasets that include elevations or other non-image data need to be paired with false coloring. This sort of false coloring will be explained in the Symbology section. The point of the matter is that raster formats require this extra step before the data can be drawn to the screen.

A different class, known as an *ImageData* class, represents an image that is either black and white or color, but which is already an image, and so does not need any sort of symbolic coloring to be applied to the data before it can be drawn to the screen. An Image class already exists as a .NET data type, so the name has been modified to be *ImageData* for this project to prevent naming conflicts. An *ImageData* class provides simple access to georeferenced data formats. It also provides access to a *WorldFile* class that is capable of producing a very simple text file with the information necessary to the georeferenced image. It is important to provide a means to access the WorldFile information because, if the image is modified, it is likely that the file storing its georeferencing content must be updated as well.

4.1.1. Temporal GIS

There has been a large amount of recent interest in the inclusion of temporal analysis in GIS. Practical applications include everything from modeling discrete historical changes of polygons in time (Robertson et al., 2007) to supporting climate models and flood prediction models, all of which gain incredible power by combining information in both time and space. Historical temporal modeling is strictly focused on time. This temporal interest places the focus on techniques such as autoregressive moving average (ARMA) models, spectral analysis and seasonal models to forecast quantities strictly using temporal information (Brockwell & Davis, 1996). More recent efforts tend to group time and space together into either database datasets (Bernard et al., 1998; Dai et al., 2005) or temporo-spatial data cubes or other indexing methods (Gatalsky & Andrienko, 2004; Botea et al., 2008; Mokbel & Aref, 2007). It is important, therefore, to support time as a potential data format, while not necessarily requiring time from older spatial datasets, or space from older temporal formats. This extension of the coordinate to include time is accomplished through an *ICoordinate* interface. The X, Y and Z values can be adjusted to work with a temporal query because, unlike simple fields, they provide access to properties. For the temporal coordinate, a new property for time, T is added, and depending on this T value, the values returned for X, Y or Z can be very different (Pang and Shi, 2002). Supporting versatile coordinates that can support time then sets the beginning stage for datasets that can handle representations in time or space or both, and can then run models that depend on both time and space (Pissinou, Radev & Makki, 2001). To support modeling that involves both time and space, it is important to be able to develop spatial constraints that are dependent on time (Brodsky et al., 1995).

4.1.2. File Formats

While it is part of the design plan to include a few basic classes that can read and write to standard geographic data formats, the principal objective is to create a series of interfaces that enable data provider plug-ins to extend the number of data formats with which other components of the software can interact. From the perspective of higher level programmers who use a Raster object, for instance, the file handling should be largely behind the scenes and not really part of their principal concern. Hiding the business logic is achieved by inspecting the file extension of the filename specified by an “Open” command and then opening the correct type of raster internally. The most basic file formats supported are the .shp or ESRI Shapefile format, which uses a specification that is now part of the public domain. MapWindow 6.0 also supports GML (Google Markup Language) which is frequently also known as KML for (Keyhole Markup Language). These formats are useful for viewing shapes in Google earth or other professional globe displays. The most basic raster format in MapWindow 6.0 is the .bgd or binary grid format, which was developed for use with earlier generations of the MapWindow project. The interfaces that allow data access, however, are able to support functionality that can extend beyond the original functionality for existing classes that implement the interfaces. For instance, the .shp file is limited by its lack of topological information, so there is no intrinsic record of which shapes share boundaries. It also does not include any binary search trees, so typically the shapes are accessed by checking the extents one by one, which is slower than being able to rule out a large portion of the shapes at once through a binary tree format. The .bgd file is limited in that it is only able to store a single band in a given file. However significant these limitations are, the file formats are

important for backward compatibility with data formats that were used by earlier generations of the MapWindow product. A basic Image handling toolkit exists in the .NET framework providing access to many important data formats like bmp, jpg, tif and others. However, the file format code for images must extend that capability with the ability to work with world files and therefore save and load georeferencing information with an image. One of the first extensions to be added was a library called GDAL, which can work with many image data formats. This library is unmanaged, and therefore may be undesirable to some developers. Therefore, rather than including the complicated build of GDAL libraries and their dependencies as part of the core application, the libraries are instead added as an optional data-provider extension.

4.1.3. Database Access

Database access is a powerful tool that is useful in .NET. However, the most prominent database format, namely the ESRI GeoDatabase that uses an access database, is a proprietary format. It is part of the goals of this project to provide interfaces that are versatile enough to easily allow others to work with database formats that they feel are important. Some formats that are being considered include PostGIS, ODBC (Microsoft SQL Server and others), Oracle Spatial and OLEDB (Microsoft Access). Support for some formats, such as Oracle Spatial, can be achieved rapidly by using the OGR library. This library is unmanaged, like the GDAL library, and would expose a large number of alternate formats. However, because it is unmanaged, it is likely that it will be added through an optional extension in the same way that the GDAL image library extension is being supported.

4.1.4. Web Service Consumption

Web mapping services (WMS) are accessed through browsers or other client software like MapWindow. The client requests a visual image for a particular geographic region. The resulting image can then be downloaded on demand. An example is Google Earth. Even though Google Earth is a desktop globe application, it pulls information to show imagery for very specific locations from a web service. Web feature services (WFS) are slightly more sophisticated. Instead of responding to a request with a visual image, they provide the data in some form of vector format. This format then consists of a dataset that can be displayed provided that the software is capable of displaying vector data formats. It is a future goal of the project to be able to consume both web mapping services and web feature services. The number of actual services is very large, so it is not the objective of the early project to include all of them. Instead, the project will incorporate at least one example of each type of service as part of the improvements developed before the MapWindow conference in April 2010.

4.2.Symbology

4.2.1. Symbolizers and Thematic Layers

Part of the OGC convention is that a class that stores the specific display characteristics that can modify how a feature or raster appears be referred to as a symbolizer. These symbolizers can have different behaviors depending on the character of the data being represented. In the case of points, there are characteristics such as symbol size or the use of specific text characters to represent the points that have no corollary in lines and polygons. Therefore, symbolizers have sub-classes that expose the unique characteristics for each. Line symbolizers, for instance, allow the user to specify

line width while polygon symbolizers use a line symbolizer to describe their borders. There are also common descriptive characteristics that are shared across points, lines and polygons, and these characteristics are grouped into a feature symbolizer. The FeatureSymbolizer class specifies common characteristics like fill color, whether or not borders are visible, and whether to draw symbols with sizes that are measured in screen pixels or geographically referenced sizes (e.g., 10 pixels or 10 miles). For rasters, the symbolizer calculates a color to use by categorizing the cell value. The result is a bitmap image that is created from the raster values. Actual drawing is kept reasonably fast because the bitmap is only created once the symbolic characteristics are updated.

In the architecture used for this project, a symbolizer is paired with one of the data formats like a Raster or FeatureSet in order to create a thematic layer. A thematic layer is a single set of data that is comprised of many instances of the same type. For instance a roads layer might be a set of LineStrings. Each road may, by itself, be comprised of several parts, but all of the parts for a single road are named the same thing. The combination of all the roads in the state of Idaho, for instance, could be considered to be a single thematic layer of roads. These layers are generally drawn in coordination with one another to show thematic representations of geographic areas.

4.2.2. Selections

Though there are many advantages in working with thematic layers that allow bulk interactions with many features at once, one weakness is that users need a separate mechanism to allow them to work with a single feature, or even a single vertex. These selection capabilities are controlled by changing the drawing mode for the map. When a user changes the mode, instead of zooming or panning in response to mouse clicks, the

behavior of the map adjusts to allow selection of an individual feature or set of features. A selected feature is represented by changing the visible characteristics (i.e., its Symbology). Having selected a feature, it now becomes possible to perform calculations or other interactions that only apply to the selected feature or features. It is customary to use a cyan color to indicate that a shape is selected, but the selection color is currently customizable by both developers programmatically and end-users.

The next level of selection is vertex selection. Normally, the individual vertices that make up a line or polygon are not shown. However, when editing or creating a shape, it is important to include a technique that allows interactions with individual vertices. This extra level of control is not yet implemented, but is planned as part of the final product, and is being written by Darrel Brown. The symbolic characteristics for these selected vertices will likely be a setting that is controlled at the level of the entire application. As such, it would be consistent across all layers, categories, selection states and shapes. It is desirable to insulate the data in common applications from being accidentally changed. To that end, it is necessary to only have vertex manipulation exposed after first activating the mode in an editor menu; possibly only after launching an easy-to-use toolbar with editing options. One option is to place the map into a shape building mode where each click inserts a new vertex in a sequence of the clicks. Another option places the map in a mode where existing features can be selected for editing by clicking on the feature, which in turn exposes the vertices of that feature. These exposed vertices do not allow themselves to be selected, or highlighted. Therefore, in the application settings a symbol for both the normal vertices (for instance a red .NET) and the selected vertices (for instance a cyan dot) is needed.

4.2.3. Categories and Schemes

Cartographic representation requires the ability to use the attribute values to organize the thematic layer into separate, visually distinct groups. For instance, it might be desirable to use larger circles to represent cities with larger populations. A collection of these groups, or categories, creates a scheme where the color or shape or other visual characteristics are chosen to represent the values in a specific attribute field. In the case of rasters, the scheme calculates the color of the pixel by analyzing the cell value, while in the case of feature layers, the colors are chosen according to values of a specific field in the associated attribute table. Whether the technique is creating a different symbol for each unique field value, or spread across a range of values, the categories themselves can be expressed as a combination of a symbolizer with a filter expression. Internal to the drawing of layers, a *DrawingFilter* will subdivide the features by using a string filter expression that is specified in each category. The filter expression uses square brackets to enclose field names. The complexity of this expression is restricted to SQL like expressions, but internally uses the *DataTable.Select* method, which is a part of the .NET Framework. At the time of this dissertation the symbolic architecture, as well as the user interface controls, have been completed for both raster and vector data models.

4.2.4. Polygon Triangulation

Drawing points, lines or polygons in two dimensions is easily accomplished through the standard GDI+ library in .NET. Drawing lines and points, or even textured tiles with images for either of those types is fairly straight forward with DirectX (Miller, 2004). The challenge comes when complex, two-dimensional polygons need to be represented as a series of triangles. Unlike open GL, DirectX does not support built in

methods that allow the programmer to break a complex polygon into triangles (Luna, 2003). It is therefore a part of this research to investigate adequate methods to bring polygon triangulation into the managed C# world. While theoretically, this triangulation can be done in linear time (Chazelle, 1991), implementing the technique can be difficult (Lamot & Balik, 1999). Instead it is more likely that this project will begin by implementing one of the simpler, slightly slower, working methods. (Held, 1998; Tarjan, 1998). Similarly, digital elevation models that should be drawn as surfaces are also ultimately represented as triangles. This triangular representation means that a part of the project must include an investigation into techniques for deconstructing these regular grids into triangles. The naïve method is to create two triangles for each grid square. Creating such small triangles rapidly produces triangle lists that are too large to render effectively, so there is a definite advantage to exploring mechanisms to take terrain datasets and triangulate them more efficiently. This can generally be done by representing areas with very small changes in elevation with larger triangles, while abrupt or steeper changes get described with fewer triangles. (Chazelle et al., 1996; Daming et al., 2005; Hartmann, 1998)

4.3.Topology

Topology is the study of how geographic features relate to one another in space. The specific algorithms that are used to calculate these relationships belong to a topic known as computational topology (Boissonnat & Yvinec, 1995; Chazelle et. al, 1996). Many of the simpler, two-dimensional algorithms have been organized by the OGC into a set of standard methods to test relationships and calculate new geometries from a limited set of methods. The *Relate* methods include: *Contains*, *CoveredBy*, *Covers*, *Crosses*,

Disjoint, *Intersects*, *Touches*, *Overlap*, *Relate*, *Touches* and *Within*. The *Overlay* methods consist of *Difference*, *Intersection*, *SymmetricDifference*, and *Union*. The significance of these methods is that they are all defined not just for *LineStrings*, for instance, but rather for any geometry, allowing for a wide range of functionality in the combinations of these basic methods. These methods are the principal contribution from the Net Topology Suite code. A fairly extensive library of underlying algorithms is required in order to successfully implement the entire scope of these topology methods.

The challenges of implementing these methods mean that using geometries as the core data objects would require developers that wanted to extend the data providing capabilities to implement the entire range of topological calculations. Instead, this project uses the *IBasicGeometry* interfaces to encapsulate the vertex coordinate information without any of the more sophisticated geometry functions. Each of the MapWindow geometries, on the other hand, can perform the entire range of topological methods. A static method on the *Geometry* class handles the logic necessary to decide how to correctly equip a basic geometry with the topology function. Because an *IGeometry* interface inherits from the *IBasicGeometry* interface, the MapWindow library automatically loads a *Geometry* as the *IBasicGeometry* on each feature. In such a case, the static method simply casts the object as an *IGeometry*. In the case where a basic geometry has been provided without also being an *IGeometry*, the static method on the *Geometry* class can create a new object and copy the vertex information from the original.

4.4.Map Projections and Coordinate Systems

In order to be effective at representing locations on the Earth, it is useful to have a basic understanding of geodesy, the science of measuring the shape of the Earth, and map projections, the transformation of coordinate locations from the Earth's curved surface onto flat maps (Bolstad 2008). In addition, there are several other parameters that combine with that information in order to define a specific coordinate system. As an example, the Universal Transverse Mercator (UTM) coordinate systems all use one basic transform that controls how map points from a position on the earth are drawn to a position on a flat map. However, because the transform produces significant distortions away from the central longitudinal position, the world is divided up into 60 separate north and south zones. Figure 24 shows the northern zones in the area of the continental United States.

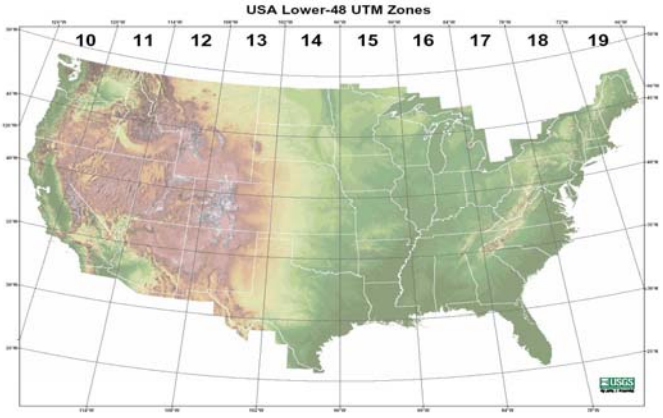


Figure 24: Continental US UTM Zones

A specific zone, using the UTM transform, establishes a specific coordinate system. A coordinate system effectively defines the location of the X and Y axis, the units of measure, and several other parameters that establish an exact mapping from geographic units to page units. A map projection, such as Transverse Mercator, defines

the mathematical transform that is being used, but is usually not enough to define a coordinate system. Figure 25 depicts both the Mercator and the Transverse Mercator projections.

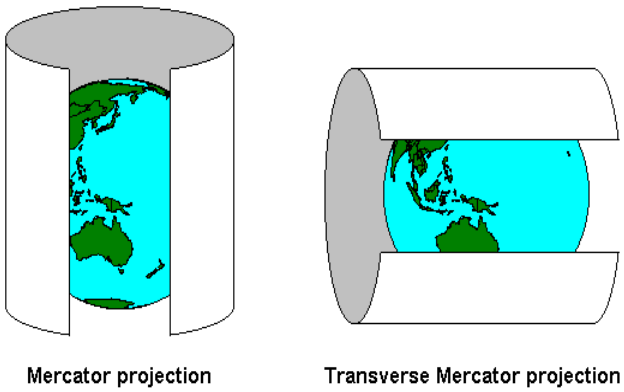


Figure 25: Mercator vs. Transverse Mercator

In this example, the Transverse Mercator projection can be visualized by wrapping the flat paper of the map around the world to form a horizontal cylinder. Projecting positions from the surface of the globe onto a flat sheet provides a relatively accurate mapping for the positions on the globe that are located close to the sheet, but as illustrated in the figure above; there are places on the earth that can not be mapped to the paper if that paper takes the form of a cylinder.

While there are several methods for storing and working with projections, the two most commonly used formats are Well Known Text (WKT) and Proj4. In this project, the string form from either format is interpreted and stored in several parameters that make up the *ProjectionInfo* class. The transformations themselves are accomplished using an extensible interface, so that new mathematical transforms can be added. Figure 26 shows the class diagrams for the classes responsible for recording a coordinate system.

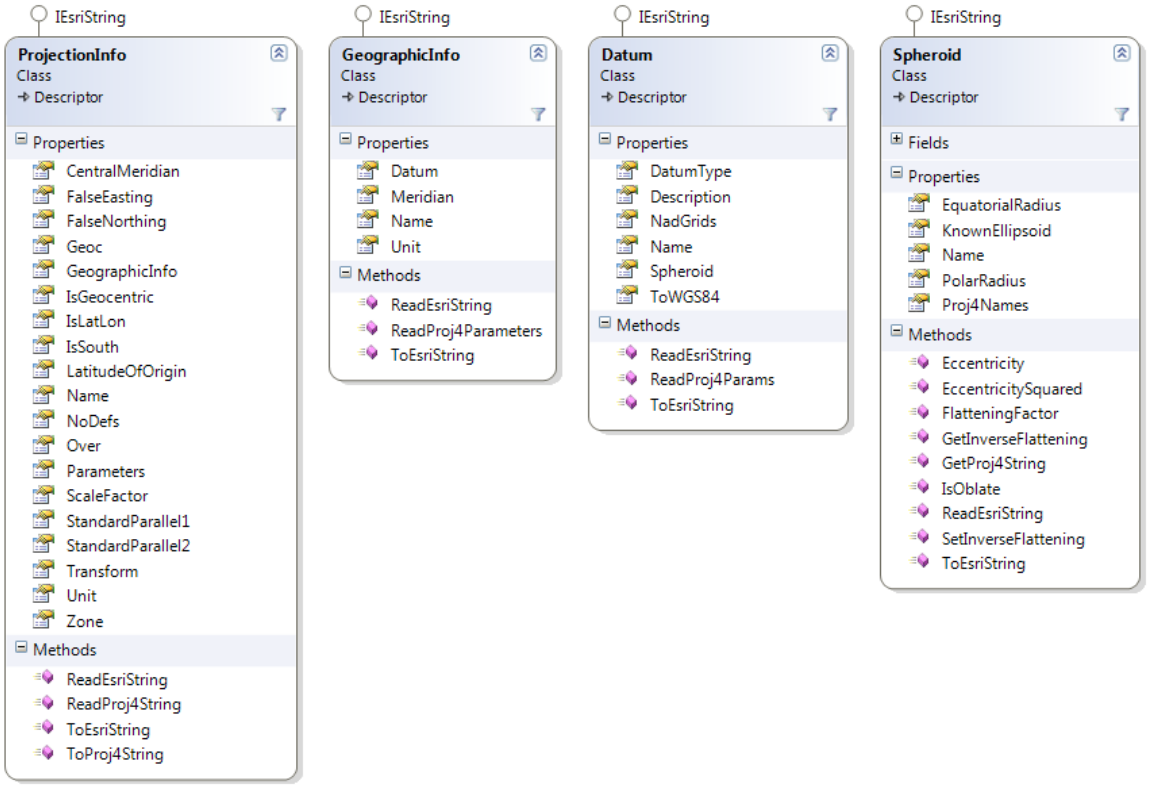


Figure 26: Projection Coordinate System Class Diagrams

The projection classes give progressively more information, but each class surrounds a basic idea that is important to geodesy. The most fundamental description is the choice of spheroid. Since the radius of the earth is slightly greater at the equator than at the poles, an oblate spheroid, often called an ellipsoid, is a more mathematically precise model for the shape of the earth. The exact parameters used, however, can change. The two most important parameters are the semi-major axis (the measure of the radius at the equator) and the flattening factor (the ratio that controls how squashed the shape appears to be). Figure 27 shows an oblate spheroid.

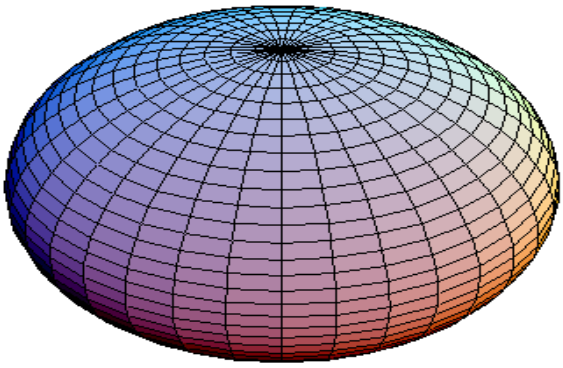


Figure 27: Oblate Spheroid

Maps that were created more recently than 1984 will almost all use the WGS84 definition of the spheroid, which has a radius of 6,378,137 meters, and supports a flattening factor of 1/298.257223563, which is about .335%. However, the earth is not a perfect ellipsoid either. The mathematical model that best matches a local area might be slightly different than the best fit for the entire earth. In addition, many maps and coordinate systems were defined using models that existed before 1984, and so it is necessary to support different models in order to support these older datasets.

A geodetic datum is a set of reference points on the Earth's surface against which position measurements are made, and (often) an associated model of the shape of the Earth. Some datum types are based on a grid mapping system, which defines the reference points that are used based on which cell of the grid the latitude and longitude fall into. Other datum types use a universal calculation that works for the whole world. The WGS84 datum is used as a standard. In order to change from one datum to another, the approach is that all the datums store the transition from their own coordinates to WGS84 and back. The calculations will either use 4 or 7 parameters if they are global transforms, and can require whole tables of data in the case of a grid transform. The combination of a datum and the angular units of measure and the meridian being used are

enough to define the geographic coordinate system. In other words, if the X and Y coordinates are simply expressed as decimal degrees, the repositioning of those coordinates onto a flat screen automatically creates a Mercator projection. This introduces very noticeable distortions, especially near the northern latitudes. However, it does not require the explicit specification of a projection transform.

The *ProjectionInfo* class specifies anything that extends beyond the geographic coordinate system as well as specifying the geographic coordinate system itself. Every projected coordinate system also uses a geographic coordinate system. The geographic coordinate system is necessary because in order to convert map points from a 3D shape to a plane, it is first necessary to decide on the 3D model. One key characteristic is the *Transform* property. The Transform can technically be any implementation of the *ITransform* interface. Figure 28 shows the *ITransform* interface, which defines the *Forward* and *Inverse* methods for actually transforming coordinates.

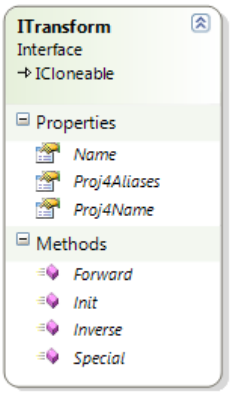


Figure 28: ITransform Interface

The *Init* method of the transform gives the algorithms a chance to perform a one-time initialization based on the projection information (all the parameters stored on the *ProjectionInfo* class). After this initialization, the forward and inverse transforms will continue to assume that all the coordinates are using the same parameters for the

projection. The *Proj4Name* represents the string that will be used when writing this transform to a *Proj4String*. Proj4 is a very well known projection library written in C that is released under an MIT license, and is essentially part of the public domain. For this dissertation, the C code has been transferred to C# for 55 of the most commonly used and shared projections between Proj4 and ArcGIS.

4.5.Components

One option for designing a re-usable plug-in based application is to use an application wide strategy that gives programmatic access to the entire application at once. This was the strategy in the MapWindow 4 generation software. The plug-in capability itself, however, is not modular, and only works when using the application as a whole. For instance, if a developer wants a custom application that has several maps separated on tab controls, or wants charts and graphs to share some of the screen space with a map, the layout would be nearly impossible to create from an existing layout on an application that has a plug-in architecture.

Instead, the idea of MapWindow 6 is to bring closer together the developers that are used to working with plug-in interfaces and the ones that are used to dragging the white box of the activeX control onto a form. With MapWindow 6, developers can drag and drop any of the components onto a form and in a few quick clicks, they can build a fully working GIS. On the other hand, any of those components is completely optional, so developers could arrange the layout and functionality to their choosing. Each of the components is also interchangeable as far as the other components are concerned. The legend will work regardless of whether a developer is working with a 2D Map, a 3D

Map, or some completely new type of map, so long as it implements the *IBasicMap* interface.

The plug-in extensibility is being encapsulated into a component that can be dragged and dropped onto a new project. Having a component to manage extensibility means that if developers want to create a completely new application that can access special data formats from GDAL, all the developers need to do is drag and drop an *ApplicationManager* onto their project. In the designer they can then specify the directory that has the GDAL data provider after which the *ApplicationManager* is able to open images from any of the file formats handled by GDAL. With the architecture designed as part of this dissertation, future developers can easily pick and choose the aspects of the core MapWindow project that are most useful for them. They can also exchange any of the existing components with completely new components, as long as the new components implement the same interface.

4.6.Tools and Model Building

In addition to opening geographic datasets and displaying them, the project must also undergo a large number of processes that involves the interactions between various features, raster grids, or selections. All of these will be controlled through an extensible design that will allow completely new functions to be introduced as processes. It will also be possible to group several processes together into a single process in the model-builder interface. The model-builder interface itself is not part of the scope for this dissertation. However, the success of the model-builder interface relies on algorithms and extensibility design ideas that are being developed as part of this dissertation.

Some of the tools that have been developed as part of this project are especially tailored to perform methods in computational hydrology such as watershed delineation (Davis & Cornwell, 1998), but there is also interest in extending the tools to include methods that are important to geosciences. *Geostatistics*, for instance, requires a broad range of statistical tools such as Kriging, spatial simulation and variography tools in order to use a collection of point samples to make accurate predictions about the unknown areas in between. (Goovaerts, 1997). These techniques are quite useful for spatial datasets that may have nothing to do with their traditional uses of mineral prospecting. For instance they have been successfully used to model orographic relationships between elevation and precipitation in arid regions (Havesi et al., 1992). Full implementations of the ultimate hydrology toolkits and geostatistical analysis fall outside the range of this dissertation, but may be proposed for post-doctoral work in the field.

5. Methods

5.1.Development Tools and Procedures

The project is being developed in C#, which is a language that is built on Microsoft's Common Language Runtime (CLR) environment. Specifically the Academic version of Visual Studio 2008 is being used and the library targets the 3.5 version of the .NET Framework. This version allows for the greatest interoperability between operating systems because it is supported when it is interpreted by Mono. After each significant change, the latest version of the source code is submitted to a repository using an open source tool called Tortoise SVN. Tortoise SVN is a tool designed to track changes individually, allowing for the re-construction of any previous version submitted to the repository. Therefore, if a bug is introduced, previous versions can be accessed where the

bug was not present. It also means that multiple developers can work on a project, each submitting their changes. Unlike simply copying a file to a storage location, the repository checks the changes in each file and ensures that the newly committed version in each case is more recent than the predecessors. That way, if there has been a change to the code in the repository while a programmer is editing the files, that programmer knows when he or she tries to commit their work that they must first download the updates and ensure that their changes do not remove the other developer's work.

Bugs and project goals are recorded in a web-based bug-tracking system called Mantis. Mantis is an open-source web server application that allows any user to report a bug, while allowing programmers to address bugs and make notes about the version of the project that has the needed changes. Multiple levels of urgency can be indicated on the bug reports, and sub-projects can be tracked independently. Version numbers can be used in order to build a roadmap where specific bugs and feature-requests are organized to ensure a timely completion of a large number of objectives.

Unit testing is a process where a function is written that takes a known value, tests it against the methods or properties in a class, and then compares the output against the expected result. A process known as Test Driven Development argues that all methods should be written after the unit test for the method so that a clear outline exists to predict if the method works or not. While test driven development is excellent for some of the file management or analysis methods, it is less useful for the display aspects of the project. It is difficult to ensure with a unit test, for instance, that the image is being drawn to the correct place on the screen. Unit tests also tend to multiply the amount of work for writing a particular section of code by about four. Therefore, one goal of this project is to

take advantage of unit tests for classes that derive the greatest benefit, namely where complex algorithms are concerned and where data handling provides easily testable scenarios.

As a partially adopted process, unit tests are introduced primarily as a debug procedure. In other words, when a problem is suspected, a unit test is used to evaluate when the problem is fixed. Once the system passes the unit test, it can be asserted that the problem has been solved. The advantage of not creating a unit test until a problem arises is that little time is wasted writing code that never gets used or testing something that is fairly trivial. Also, once a unit test is written, it remains in the library so that on future occasions it can be verified that all of the previously written unit tests still pass.

5.2.Code Conventions

Naming for methods, classes, and public properties is all done in *pascal case*. Pascal case defines the capitalization rule where the first character of all the words is upper case, and the other characters are lower case. Using pascal case allows for several words to be placed together without using spaces, but still combined forming a meaningful name. Argument parameters and local variables use *camel case,* which is defined as being the exact same as Pascal Case except that the very first letter is lowercase. Private variables that are class level variables and can occur in several methods in the same class are indicated with an underscore. Variables that represent a control like a button or a text box are preceded by three lowercase letters that indicate what kind of control is being named. Each class is stored in its own file of the same name, making it easy to locate a class through the solution explorer. All of the code is CLR compliant, which means that the library can be easily run from Visual Basic or other

programming languages without causing problems, not just C#. The most common cause of a non-CLR compliant issue is to have a lower case variable and an upper case property of the same name. For CLR compliance, code must be written as though it were not case sensitive.

The indentation conforms to the automatic settings in Visual Studio, or four characters. Braces start and end at the same indentation line to simplify matching braces. Spacing is used to separate smaller blocks of code that are strongly related within a method. Rather than having comments everywhere inside a long method, very short methods are used with descriptive names. Triple slash comments are used before all public methods, which is how Visual Studio adds comments to methods that will appear in IntelliSense. IntelliSense is a clever addition to Visual Studio that displays each of the public methods, properties, and events on a class whenever the user types a period after the variable. It brings these elaborations up in a context menu, which then allows the user to quickly use recognition instead of relying on dead recall or continuous clumsy references to reference manuals. The special benefit of the triple slash comments is that Visual Studio will automatically incorporate these comments into intellisense, allowing help content to be displayed directly in the Integrated Development Environment (IDE). This allows future developers to receive very professional programming assistance directly from the methods being written. As a project grows, the importance of intellisense that can provide immediately relevant assistance also grows. It is even useful for the original developer, who may need to be refreshed from time to time.

Everything in this project is object-oriented. The shear abundance of separate classes is the legacy from the Java Topology Suite, since the Java language can only use

objects. Most coding references give a strong emphasis to using object-oriented programming development because it is much easier for outside programmers to follow the logic of the process. Objects are broken into smaller parts that make sense and work together. Methods are named for what they do, while properties are mostly inert. In other words, if there is a lot of calculating necessary to determine the area of a polygon, the programmer needs to create a method named *ComputeArea()* rather than a property called *Area*. It is generally considered acceptable to have an *Area* property that executes the *ComputeArea()* calculations if those calculations are trivial. Otherwise the results are cached so that future references will simply access a stored value. Developers that only have an external view of the class can easily distinguish time consuming methods from comparatively quick properties. If the programmers know that the calculation will take time, they will cache the result so that referencing the area multiple times in code will not cause unnecessary delays. Turning an *Area* property into a *ComputeArea()* method has the effect of telling developers that they should cache the result. An Area property looks like a cached value, and so most programmers would not hesitate to make many references directly to the property and inadvertently repeat the lengthy calculation, which could unnecessarily impact performance.

Member (or class level) variables should never be public or protected. Instead they should be private and allow access through public or protected properties. Restricting the access to the underlying variable allows interfaces to be designed with the same properties specified (since interfaces cannot specify variables directly, only properties). Enumerations need to be used instead of hard numbers or strings. Strings, when used, need to be stored in a resource file that can be easily translated and staged for

multi-language support. A *Template* is used for creating classes that already have the license wording and a time/date stamp embedded into the class so that it becomes clear when the class was first designed and future contributors can easily contribute to the top. When a large number of parameters are needed, a class is designed to contain those parameters to reduce the complexity of calling a method.

There are many other conventions that could be listed, but the purpose of this section is simply to demonstrate some of the ideas that are being used for this project. These ideas have been essential to making it easy to work with such a large project and still be able to navigate effectively.

5.3.2D Vector Drawing

This section discusses the chosen drawing methods used for this project, and presents some quantitative comparisons in performance or memory to justify those choices. The GDI+ library is responsible for drawing curves, rendering fonts, filling polygons and drawing images. The strong advantage of GDI+ is that it uses a single interface that can draw to many devices, most importantly the screen and printing. Therefore, the code that is developed to render content to the screen can be re-used for printing. An open source implementation of the .Net framework named Mono (http://www.mono-project.com/Main_Page) allows the GDI+ library to work with other operating systems, not just windows. GDI+ also contains methods that provide support for translucent fill types, anti-aliasing and several new drawing capabilities. The disadvantages of GDI+ drawing is that it has no support for animation (synchronizing with the framebuffer) and does not support turning 3D content into a two dimensional image. Games and animated systems work with DirectX or OpenGL.

5.3.1. GraphicsPath vs. DrawLines

Using .NET custom controls, a *Graphics* object is passed in as an argument during the *OnDraw* method, which allows content to be drawn to the control. The most direct route to drawing is to draw directly to the map control by using the *Graphics* object that is passed in. An alternative is to use a bitmap as a buffer, draw the content to the bitmap, and then draw the bitmap to the screen all at once. The Figure 29 and Figure 30 illustrate the performance comparisons for some randomly generated lines, and some road shapes that were loaded from a Utah Roads shapefile respectively. What is most apparent in these two plots is that rendering directly to the screen is slower. The reason is that as content is rendered; the screen performs an update, only to have more content drawn that forces yet another update. The combined re-drawing cycles take much longer than drawing to a back buffer and rendering to the screen only once. Figure 29 compares drawing directly to the screen with drawing to a buffer image, using both the *DrawLines* method and the *DrawPath* method that uses a *GraphicsPath*. In both cases, drawing directly to the screen results in very poor performance.

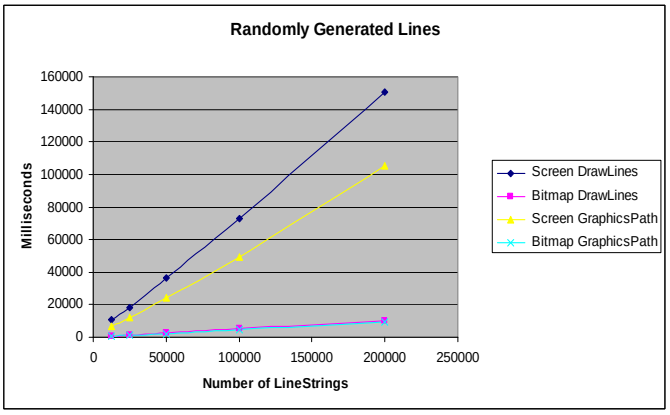


Figure 29: Randomly Generated Lines

Figure 30 shows the same content as Figure 8, but using real-world road data, which is more fractal in nature, and allows for much smaller lines.

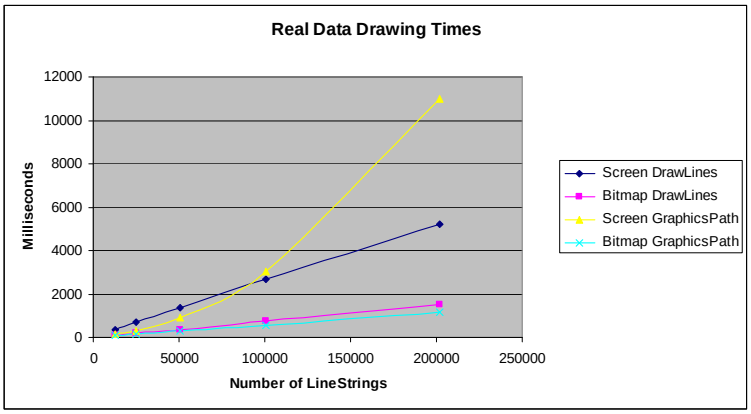


Figure 30: Real Road Data Drawing Times

The rendering method for this project was to draw to a bitmap and then use the *Graphics.DrawImageUnscaled* method to draw that bitmap to the control's drawing surface all at once. Drawing an image the size of the control adds a very small penalty, but the graphs include this extra drawing time in the values, and the times for drawing to a bitmap buffer are still much faster. A less obvious decision is whether to use a graphics path for the drawing, or call the *DrawLines* method directly. Figure 31 shows a clearer depiction of the *DrawLines* versus the *GraphicsPath* approaches, once drawing directly to the control has been ruled out.

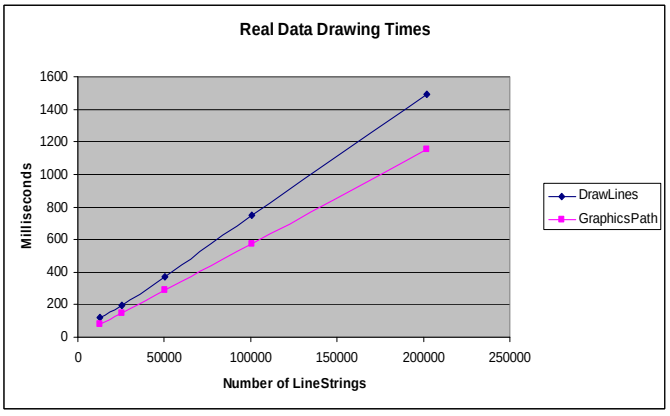


Figure 31: DrawLines vs. GraphicsPath

5.3.2. Point Culling

In the trial above, using a *GraphicsPath* adds a small performance benefit. In both cases, the array of points to use is created in advance so the only aspect being

illustrated here is the time to actually render the points. Though the graphs continue to illustrate both methods, the method chosen for this project was to use *GraphicsPaths* for lines and polygons. With real-world data, it is frequently true that at small scales many features become so small that all their coordinates are effectively represented by the same pixel on the image. Drawing these degenerate lines can cause exceptions for a *GraphicsPath* when the line thickness is greater than 1. If the *Graphics.DrawPath* method throws an *OutOfMemoryException,* the cause of the error is likely related to an inability to determine the directionality of lines that have the same start and end point. Interestingly, the exception will not be thrown during the *DrawLines* method with similar degenerate line data. Solving this problem, however, also has an added benefit. By not drawing lines that are degenerate, the number of lines drawn is reduced. The following table illustrates the number of lines that are actually drawn for random sampling of Utah State road data when the scale is small enough to include all of the roads. This behavior requires that the data have a fractal organization where there are many shapes or figures that are very small. In the randomly generated lines case, almost no lines would be culled. Table 1 gives a precise account of how many lines are actually drawn from the road dataset when the whole state was in view.

Table 1: Number of Lines Kept

Number of Lines	Lines Drawn
12607	5408
25216	10872
50429	21623
100862	43067
201723	86585

When drawing only about 43% of the lines, the rendering time is also reduced to around 43% of the original time. Figure 32 shows the drawing times for the cases where

all the lines were kept, and cases where the lines that were reduced to a single point were removed.

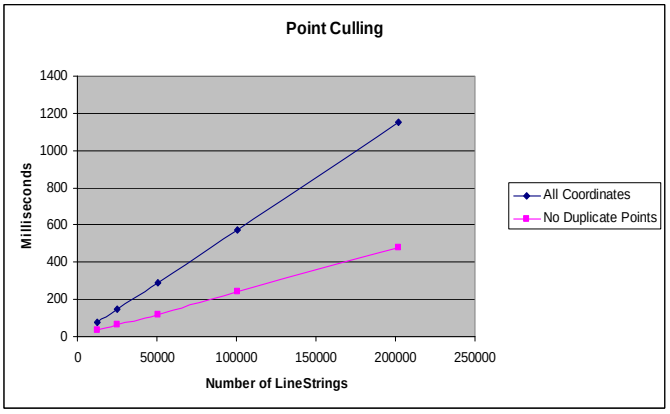


Figure 32 : Duplicate Point Elimination Drawing Times

Drawing thick lines takes more time than drawing thin lines, and drawing lines that are composed of several strokes on top of each other takes longer still. Figure 33 demonstrates the time profile for each of these scenarios including both the *GraphicsPath* and the *DrawLines* methods in order to demonstrate that changing the line parameters does not affect the decision to use a *GraphicsPath*. It is noteworthy that there is more of a drawing performance difference between the thin and thick lines than there is between using a graphics path and simply calling the *DrawLines* method, yet in every case the *GraphicsPath* method has a performance advantage. Figure 33 also shows the drawing times for thick lines, multi-layered lines, and thin lines to illustrate that the *GraphicsPath* method is at least slightly preferable in all paired comparisons.

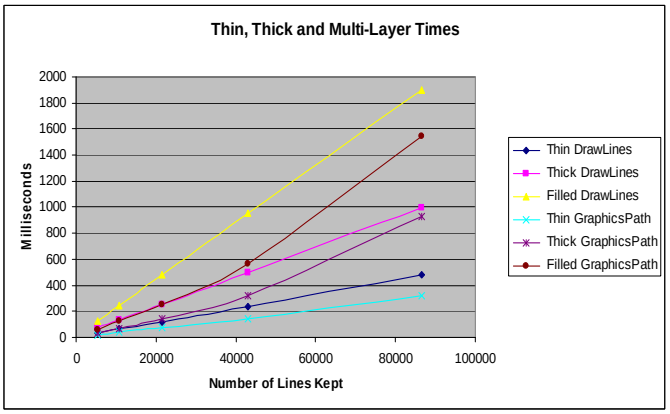


Figure 33: Drawing Times as a function of Number of Lines Drawn.

5.3.3. Geometric Features vs. Vertices

While the geometrically complete features offer an easy platform to begin geometric analysis, they have a comparatively large memory profile. In addition, even though the coordinate class was adjusted to use public fields to make it as fast as possible, it is still slower than using the vertex array. (See Interfaces in Section 6 for a performance comparison between fields and properties). Figure 34 illustrates that memory criteria aside; there is a noticeable rendering performance improvement from using arrays of doubles to store the vertex values. The change in performance is especially true for larger datasets. Figure 34 also shows the time performance of using double valued vertices versus using coordinates stored in classes with public fields for the X and Y values.

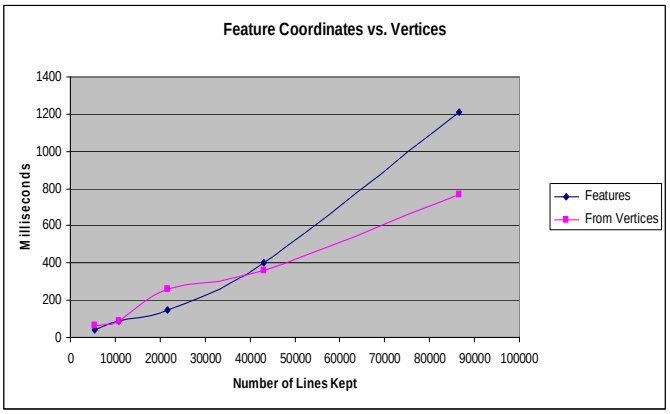


Figure 34: Drawing Times for Coordinate Objects or Double Arrays.

The Figure 35 illustrates the memory of the full geometric *Features* after all other structures and memory have been subtracted as compared with the actual file size. Since the vertices require about the same amount of memory in ram as the file size, it stands to reason that relying on the vertices alone uses less memory in addition to offering faster drawing.

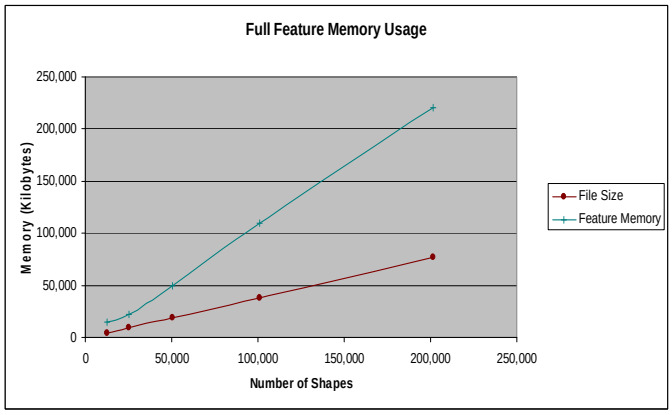


Figure 35: Memory Usage as a Function of the Number of Shapes for Feature Objects.

The GDI+ *Graphics* class supplies the ability to use a transform. The existence of a transform allows real world coordinates to be translated without performing the calculations manually each time. However, the transform class requires that the double precision coordinates are reduced to floating point coordinates. Floating point coordinates present a problem in that they can only hold seven digits of precision. As an example, in latitude and longitude coordinates, three digits are necessary to hold enough

precision to represent values to the nearest decimal degree. Subtracting three from the seven leaves only four remaining digits. A measure of .0001 decimal degrees has an accuracy of about 11.1 meters, or about 36 feet. While this level of accuracy is acceptable for rendering at the scale of the continental US, it would be a poor representation for displaying content on the scale of cities, roads or buildings, which are commonly required for cartographic mapping. Therefore the decided upon method was to re-calculate the integer pixel locations directly from the double precision vertices every time the view is updated.

If image content is buffered to an image that is larger than the size of the map, panning the map can pull pre-rendered content into the view. However, drawing to a buffer area that is nine times the size of the original map can slow down zooming. The chosen method in this case was to allow both methods to exist with an additional property on the map that allows developers to select which technique their map should use.

Controlling the map scale using the mouse scroll wheel may require many intermediate drawing states before the final scale is chosen by the user. If the drawing time for the screen is longer than a few milliseconds, it creates an awkward lag effect that can make it difficult to control the scale. To improve the user experience while using the scroll wheel, the previously cached image is scaled to match the new zoom extent. Images that are stretched larger than their original versions appear distorted as if they are made from very large pixels. This method allows the map scale to be updated as quickly as the scroll wheel could be adjusted regardless of how long it took to render the detailed scene once the user chose a new extent.

5.3.4. Extended Buffers and Multi-Layer Buffers

Another method that was investigated was caching separate bitmap stencils for each layer. If users click the checkbox in the legend to turn on or off a layer, they immediately see the changes. For large layers, using separate stencils for each layer requires extra memory for holding bitmap content for something nine times the size of the screen multiplied by the number of layers that were loaded in addition to the combined buffer. The method used in the current version does not use these stencils because the existence of the buffers, especially when the buffer is extended to nine times the screen location, causes an unnecessary increase in memory and noticeably slowed drawing updates. The drawing performance is compared between drawing lines using a stencil system for both the regular sized buffer and the extended buffer that is nine times larger. In this case, the worst case scenario was chosen for the extended buffer where the entire buffer featured drawn content, and the visible region of the map only showed $1/9^{th}$ the area. In all cases the anti-aliasing mode was used. Anti-aliasing addresses the jagged appearance of a diagonal line drawn with pixels. Instead of drawing the pixels as the color of the line, pixels are given a color that is calculated by using the color of the line and the background color. Instead of pixels being entirely on or off, they take on an intermediate color. This calculation takes a little longer to execute, but results in a much cleaner image. The drawing times for just the inner map are so much faster than drawing the full domain that two separate graphs are used in order to illustrate the minor performance benefit by not using multiple layer stencils. Figure 36 shows the time comparison for drawing multiple layers from individual transparent stencils as opposed to just using a single drawing buffer.

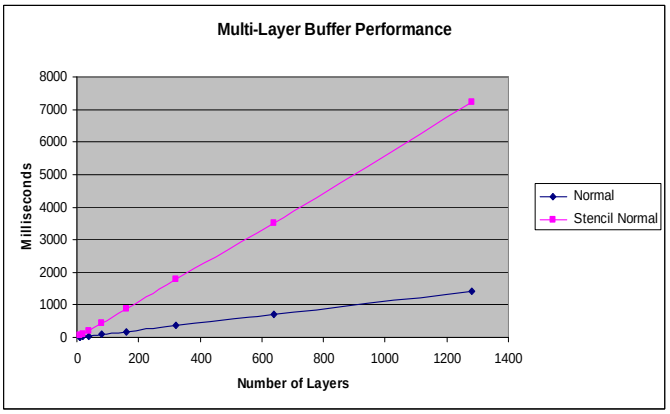


Figure 36: Direct Drawing vs. Multi-layer Stencil

Figure 37 shows the same relationship in the case where a much larger image is used as the buffer, one that is nine times the previous size.

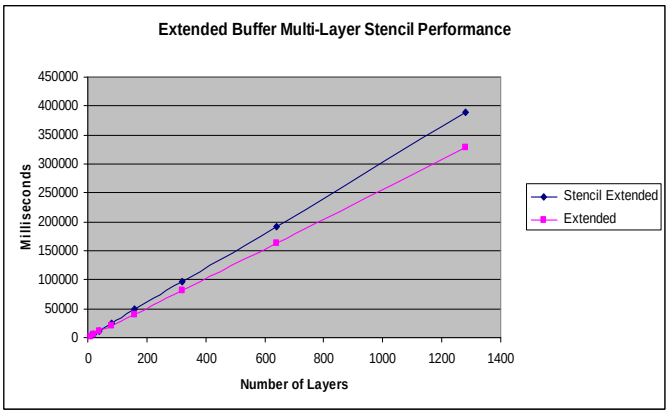


Figure 37: Extended Single Buffer vs. Multiple Layer Buffers

The Figure 38 illustrates the distinction between how extending the drawing region effects performance when compared with having multiple transparent stencil layers that are each drawn separately. In the case of vector datasets, the number of features being drawn has an enormous impact on the performance. Restricting the view to just the viewable area can drastically improve the drawing performance. The single layer or multi-layer buffer decision only strongly affects the performance when the number of features is small, but the number of layers is large.

Figure 38 also shows both the extended buffer and the regular sized buffer superimposed in order to demonstrate that using an extended buffer has much more impact on rendering performance than using extra transparent stencils.

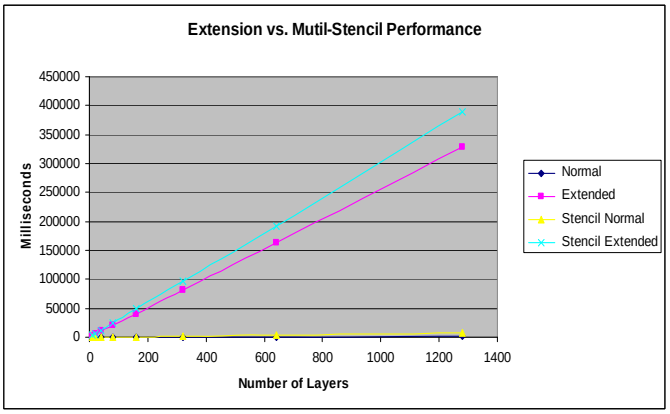


Figure 38: Extension vs. Multi-Layer Buffer Drawing Times

For example, even in the comparatively small sized map of about 800 x 600 pixels, a 32 bit-per-pixel (bpp) image requires 1.92 Megabytes. While some large vector layers can exceed 100 megabytes, a more customary size is around 10 to 20 megabytes of memory. If the buffer is extended, however, so that the pixel area is nine times the area of the map control, the memory requirement for each layer is closer to 17.2 megabytes. This increase in the memory requirement would effectively halve the number of layers that would fit in memory if each of those layers is also around 17 megabytes of vector data. As of 2009, a normal working set of memory requires no more than about 500 megabytes. Table 2 illustrates the buffer memory requirements, not counting any memory required for the vector content. As the number of layers increases, storing separate layers requires an impractical amount of memory.

Table 2 shows that the principal consideration with separate buffers is not the drawing performance, but rather the amount of memory required. If a particularly large amount of memory is consumed, then drawing times will also be affected as the system begins to swap Random Access Memory (RAM) with the hard-drive.

Table 2: Buffer Memory Requirements

Number of Layers	Normal	Extended	Stenciled Normal	Stenciled Extended
1	1.92	17.28	3.84	34.56
2	1.92	17.28	5.76	51.84
4	1.92	17.28	9.6	86.4
8	1.92	17.28	17.28	155.52
16	1.92	17.28	32.64	293.76
32	1.92	17.28	63.36	570.24
64	1.92	17.28	124.8	1123.2
128	1.92	17.28	247.68	2229.12

5.3.5. Line Overlap Styles

Two different methods for drawing connected lines were considered. In the following figure, the two different techniques illustrate the visual difference between drawing all the layers of each segment, and iterating through the segments separately or drawing all the segments for one layer and then overwriting that layer with the next layer. Figure 39 shows the two overlap styles being considered.

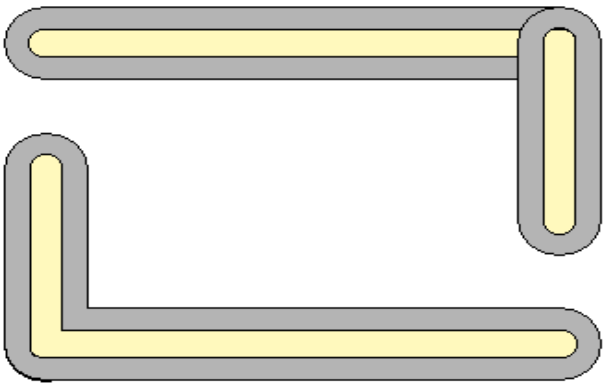


Figure 39: Two different overlap Styles

In the first case, whichever segment is drawn later ends up covering the one below it. In order to create continuous representations of roads, however, the pattern shown in the lower intersection was chosen, where all the border lines are drawn and then the interior is added to the line.

5.3.6. Vertex Storage

Different data structures were considered for arrays storing the vertices. One option investigated was a jagged array of doubles, where the outer array listed all the vertices, and the inner array listed the ordinates of each vertex. This structure was compared against a structure where a separate array of X ordinates were stored, and a separate array of all the Y ordinates. The drawing speed was the same but the memory was significantly different. For two double precision ordinates, the jagged array configuration required twice as much memory. The conclusion drawn from this was that .NET arrays each require 64 bits in memory, not counting the memory for their actual members. Each coordinate itself was only 64 bits, so the actual memory requirement was being doubled just because of how the vertices were organized.

It was also considered whether to store the X and Y terms separately as an array of x values and an array of y values or to combine them into an interleaved array of doubles. While iterating through the second method is slightly more complex, it offered some interesting options to help improve the file access performance for shapefiles. One technique for reading values from a binary stream is to use a bit converter to read in values one at a time. Another is to consider a large set of bytes all at once, and to convert them using a method called *Buffer.BlockCopy*. The block copy method is basically

designed to copy data in bytes, but allows developers to use arrays of primitive data types (like integers and doubles) instead of arrays of bytes. The block copy method allows developers to have an array of bytes copied to an array of primitives, or vice versa. Shapefiles store their vertices as interleaved x and y values with a separate array of z or m values that are optional. Mimicking this option allowed for a high speed *Buffer.BlockCopy* of all the vertices of each shape at once. Figure 40 shows the performance comparison between the *BlockCopy* method and using a *BitConverter* to individually convert each double value.

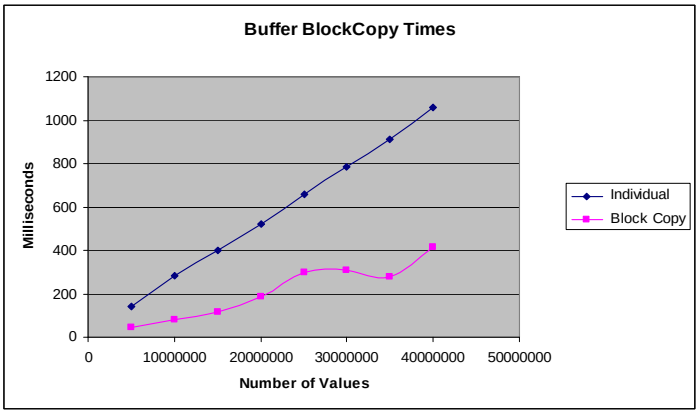


Figure 40: Buffer.BlockCopy vs. BitConverter.ToDouble

Even though the units of time are small here, the performance change is noticeable when combined with the other steps required to open and display the file.

5.3.7. Anti-aliasing

Figure 41 represents drawing lines with borders and polygon borders with no anti-aliasing. The image has been enlarged by 400% to clearly illustrate the pixilation that takes place.

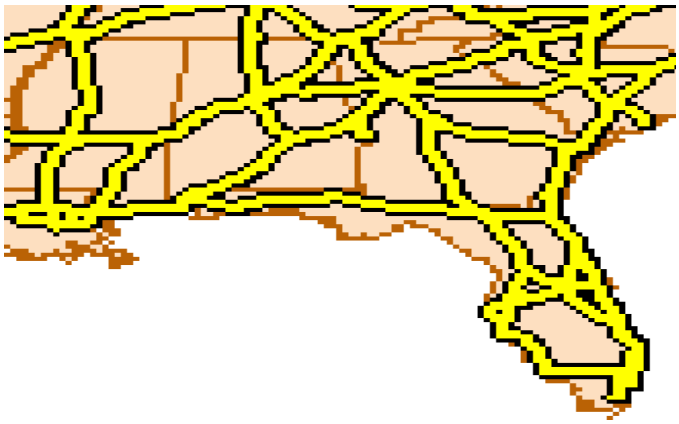


Figure 41: No Anti-Aliasing

Figure 42 illustrates the same view but where anti-aliasing is used. This smoothing effect creates smoother, more continuous looking linear features.

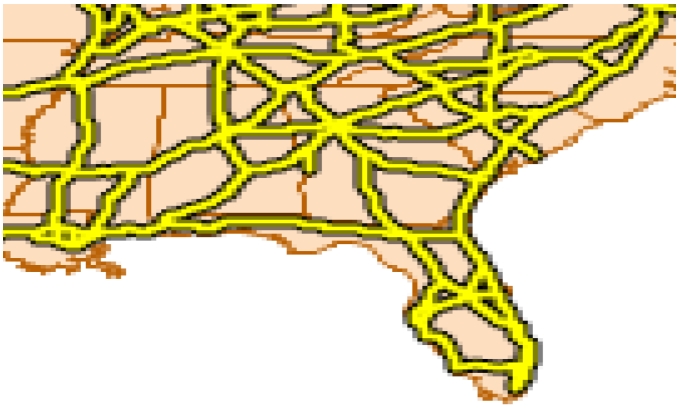


Figure 42: Anti-Aliasing

Anti-alias smoothing comes with a price in drawing performance. Figure 43 illustrates the performance impact of using anti-aliasing. As the number of lines becomes large, the performance cost introduced by anti-aliasing becomes more apparent.

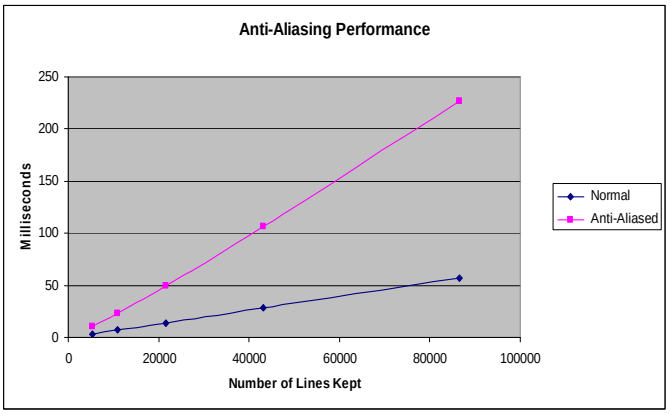


Figure 43: Anti-Aliasing Performance

Because smoothing may not be desirable in all cases, the method used in this project was to provide a setting that allows the user to control smoothing on a layer by layer basis. However, anti-aliasing was chosen as the default drawing method.

The methods used for MapWindow 6.0 followed a basic priority. The first priority is visual quality. The importance of visual quality affected the decisions about how the overlapping joints on lines would be drawn and the choice of providing smoothing as an option. The second priority is fast rendering. The desire to draw things quickly affected most of the methods discussed in this section. The final consideration was reducing the memory footprint as much as possible while still preserving the fast rendering performance.

6. Component Architecture Design

6.1.Geometry Relationships

In addition to organizing coordinates for drawing, the geometry classes provide a basic framework for testing topological relationships. These are spatial relationships that are principally concerned with testing how two shapes come together, for instance whether two shapes intersect, overlap, or simply touch. These relationships will not

change even if the space is subjected to continuous deformations. Examples include stretching or warping, but not tearing or gluing.

The tests to compare two separate features look at the interior, boundary, and exterior of both features that are being compared. The various combinations form a matrix illustrated in the figure below. It should be apparent that not only are the intersections possible, but each region will have a different dimensionality. A point is represented as a 0 dimensional object, a line by 1 dimension and an area by 2. If the test is not specific to a specific dimension, it can represent any dimension as "True." Likewise, if it is required that the set is empty, then "False" is used. Figure 44 shows the intersection matrix, illustrating how the interiors, boundaries and exteriors are considered independently to explicitly define *relate* operations.

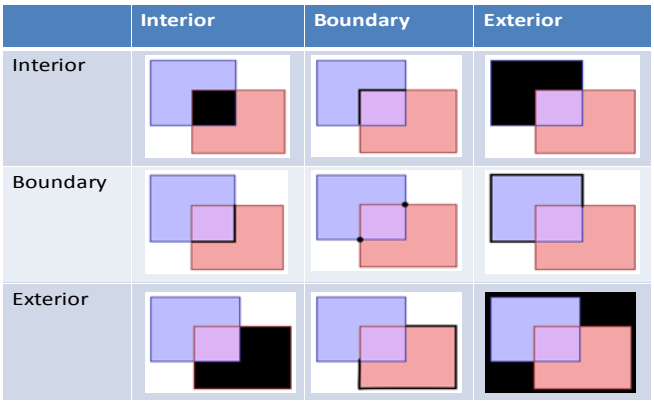


Figure 44: Intersection Matrix

Graphically, the intersection matrix for two polygons is illustrated. Some tests can be represented by a single such matrix, or a single test. Others require a combination of several tests in order to fully evaluate the relationship. When the matrix is represented in string form, the values are simply listed in sequence as if they were being read from the top left row, through the top row and then repeating for the middle and bottom rows. The following are all possible values in the matrix:

- T: Value must be “true” – non empty – but supports any dimensions >= 0
- F: Value must be “false” – empty – dimensions < 0
- *: Do not care what the value is
- 0: Exactly zero dimensions
- 1: Exactly 1 dimension
- 2: Exactly 2 dimensions

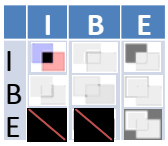
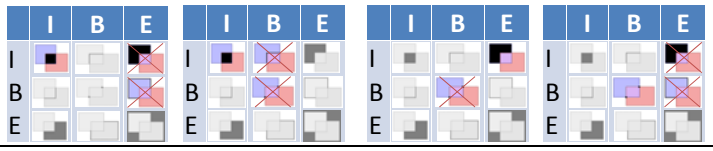
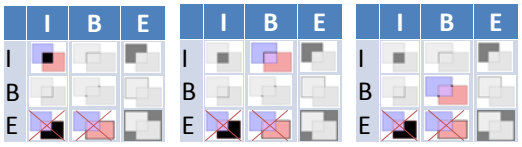
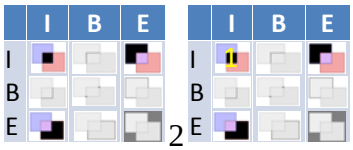
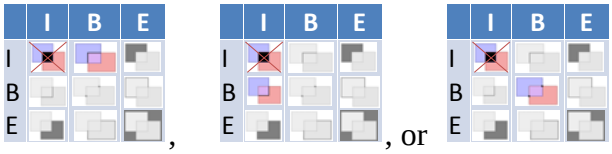
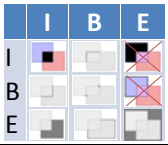
The following is a visual representation of the test or tests required in each case. A red X indicates that the test in those boundaries must be false. A colored value requires that the test be true, but does not specify a dimension. A gray value indicates that the test does not care about the value of that combination. Table 3 shows the definitions for the various relate operations.

Table 3: Geometry Relate Definitions

Relationship	Definition
Contains	Every point of the other geometry is a point of this geometry, and the interiors of the two geometries have at least one point in common.
Covered By	Every point of this geometry is a point of the other geometry
Covers	Every point of the other geometry is a point of this geometry.
Crosses	Geometries have some, but not all, interior points in common. 1) Point-Line, Point-Area, Line-Area 2) Line-Point, Line-Area, Area-Line 3) Line-Line
Disjoint	The two geometries have no point in common
Intersects	The two geometries have at least one point in common
Overlaps	The geometries have some but not all points in common, they have the same dimension, and the intersection of the interiors of the two geometries has the same dimension as the geometries themselves. 1) Point-Point, Area-Area 2) Line-Line
Touches	The two geometries have at least one point in common but their interiors do not intersect.
Within	Every point of this geometry is a point of the other geometry and the interiors of the two geometries have at least one point in common.

Table 4 shows the same relationships, but gives both a pictorial and string expression representing the explicit definition. The I stands for interior, the B for boundary and the E for exterior regions.

Table 4: Relate Expressions and Illustrations

Operation	Expression	Visual Representation
Contains	T*****FF*	
Covered By	T*F**F***, *TF**F***, **T*F*** or **F*TF***	
Covers	T*****FF* or *T****FF* or ****T*FF*	
Crosses	1)T*T***** 2)T*****T** 3)0********	
Disjoint	FF*FF****	
Intersects		Not Disjoint
Overlaps	1)T*T***T** 2)1*T***T**	
Touches	FT*******, F**T*****, or F***T****	
Within	T*F**F***	

6.1.1. Overlay Operations:

Being able to test the existing relationships between geometries is useful for doing analysis, but many times it is useful to be able to alter the geometries themselves. One effective way to do this is to make use of a second geometry. Consider the case of a clipping operation. In the following figure, the rivers extend beyond the boundaries of the state of Texas. Using an overlay operation is exactly the kind of operation that helps with this kind of calculation. The terminology of the operations used is not limited to specific scenarios like polygon to polygon. Instead the same terminology applies to all the geometries using the following definitions. Figure 45 illustrates the intersection of the United States Rivers with the political boundaries for the state of Texas.

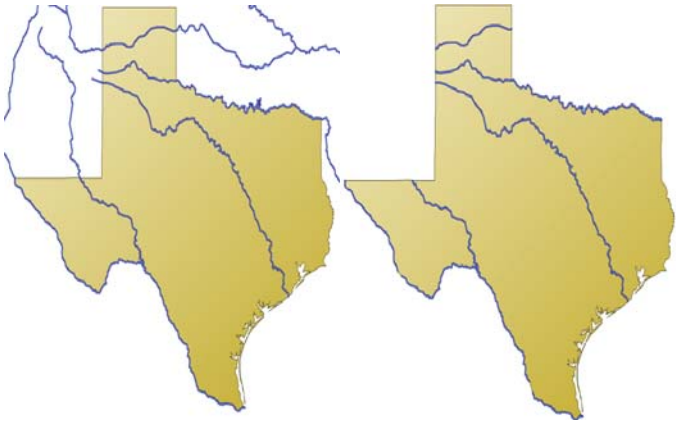
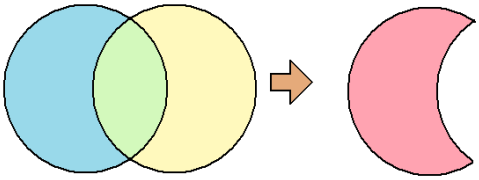
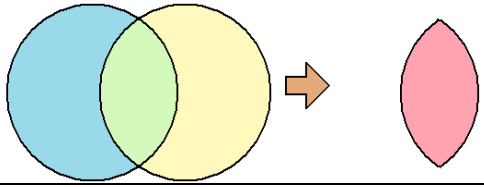
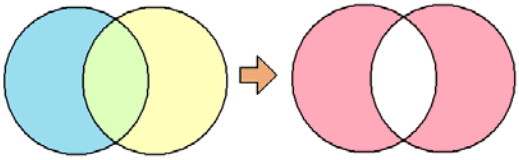
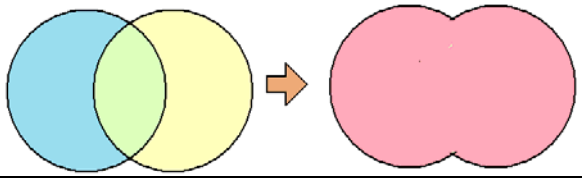


Figure 45: Clipped Texas Rivers

Table 5 shows the four major geometric overlay operations and gives both a verbal and pictorial definition for each case.

Table 5: Overlay Operations and Definitions

Operation	Definition	Picture
Difference	Computes a Geometry representing the points making up this geometry that do not make up the other geometry.	
Intersection	Computes a geometry representing the points shared by this geometry and the other geometry.	
Symmetric Difference	Computes a geometry representing the points in this geometry that are not present in the other geometry, and the points in the other geometry that are not in this geometry.	
Union	Computes a geometry representing all the points in this geometry and the other geometry.	

6.2. Data Management Architecture

Figure 46 shows the coordinate class.

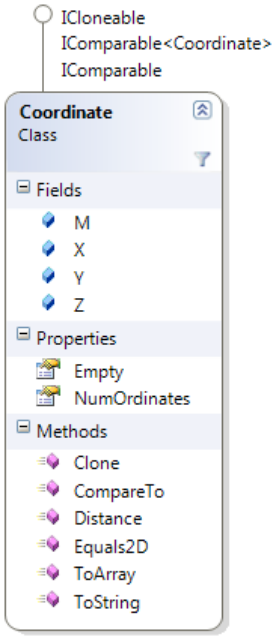


Figure 46: Coordinate Class

This section provides some simple class diagrams that demonstrate some of the architectural decisions made for MapWindow 6.0. Specifically this section deals with the differences between MapWindow 4 and MapWindow 6. The most challenging ideas involve the introduction of inheritance and extensible interfaces.

6.2.1. Point

The *Point* class was formerly a class that provided rapid access to X, Y and Z values. This class has been replaced by the *Coordinate* class. A *Point*, as defined by the OGC Simple Feature Specification, has geometry methods like Intersects, and is independently available in MapWindow 6, but has many more capabilities than just storing a coordinate location. Therefore, the coordinate class has taken over these responsibilities.

6.2.2. Extents

The *Extents* class in MapWindow 4 was a bounding box that defined a two or three dimensional region. The new class introduces a Minimum and a Maximum coordinate, which has the X, Y and Z values for the lower and upper bounds in each case. The extents class also provides a means to control the X, Y, Height and Width properties, which are calculated from the Minimum and Maximum coordinates. Figure 47 shows the envelope class.

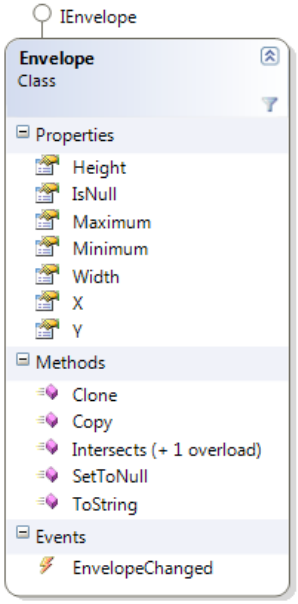


Figure 47 : Envelope Class

6.2.3. Shapes

The *Shape* class in MapWindow 4 was a generic feature and provided basic access to the coordinates, and gave the geographic extents for that shape. In the old model, the only connectivity between a shape and the attributes was that they would both have the same index. In the new version, a feature has a *DataRow*, which provides direct access to the attributes specific to this feature. It also has the *Envelope*, which describes the bounds and can access the coordinates directly. Because some features are complex,

like multi-polygons, the *BasicGeometry* is provided, which organizes the coordinates according to the OGC definitions like *Polygons*, *LineStrings*, or *Points*. Figure 48 shows the feature class diagram.

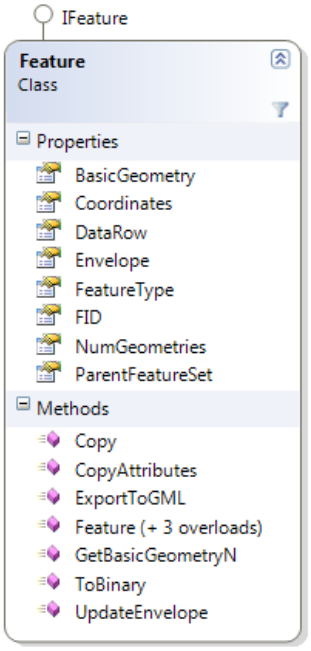


Figure 48: Feature Class

6.2.4. Shapefile

In MapWindow 4, it was assumed that there was only going to be one supported vector format, and that format was the shapefile. In MapWindow 6.0, the introduction of extensible data providers implies that many different data sources will be possible. Therefore, the name *FeatureSet* seemed more appropriate. *FeatureSet* classes provide access to the *Envelope*, a *DataTable* of attributes, as well as a cached array of vertices. The *ShapeIndices* class is a special shortcut to allow for effective use of the vertices class, and is currently being used to speed-up rendering operations. A *Reproject* method is also available on the *FeatureSet*, which will change the in-memory coordinates without changing the original file.

Regardless of whether an image, raster, or *FeatureSet* is being manipulated, they derive from the same basic class, called a *DataSet*. The *DataSet* provides simple information like the name and projection information that are shared across formats. Selection is supported here, but it should be understood that there is a detachment between the *FeatureSet* and the layer being drawn. The layers control how a particular *DataSet* is symbolized and drawn in the map. However, the ability to discover the features that are within an envelope, or that have attributes that match an SQL query, is supported and used at the level of the *FeatureSet*.

The *FeatureSet* acts as both a base class, meaning that a *PolygonShapefile* ultimately inherits from *FeatureSet*, and as a wrapper class. In other words, when the user programmatically creates a new *FeatureSet*, there is no way to know what kinds of classes will be necessary in order to access the data on a file, a database or web service. Therefore, the *FeatureSet* is provided as a kind of wrapper for an internal *IFeatureSet*. When the *Open* method is called internally, the *FeatureSet* class requests a new *IFeatureSet* from the default *DataManager*. Any requests for features or other properties are simply passed to the internal *Featureset*. Figure 49 shows the FeatureSet class.

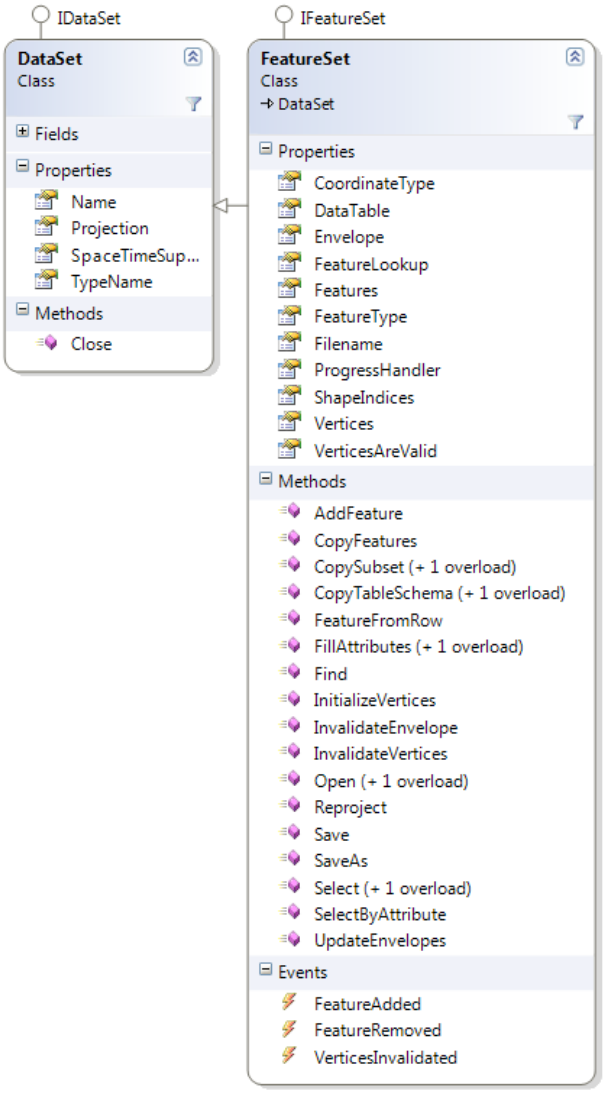


Figure 49: FeatureSet

6.2.5. Grid

The *grid* class was formerly a wrapper class that could internally represent an integer, float, short, or even double grid. The main goal of the *grid* class was to be able to cycle through the rows and columns of a raster data file and then retrieve or set the various numeric values.

The new representation is called a *Raster*. Unlike the MapWindow Grid, *Rasters* can have multiple bands, with each band also being a *Raster*. The vast set of properties in Figure 50 may seem intimidating, but realistically, there are only a few properties that

are critical. The first is the *Value* property which stores the actual values. To access the value on row 7 and column 6, simply access double val = myRaster.Value[7, 6]. Regardless of how the source data is stored, the accessed value will be converted into double values. The process of writing code is much easier since there is no need to handle each of the separate possibilities independently. Figure 50 shows the raster class.

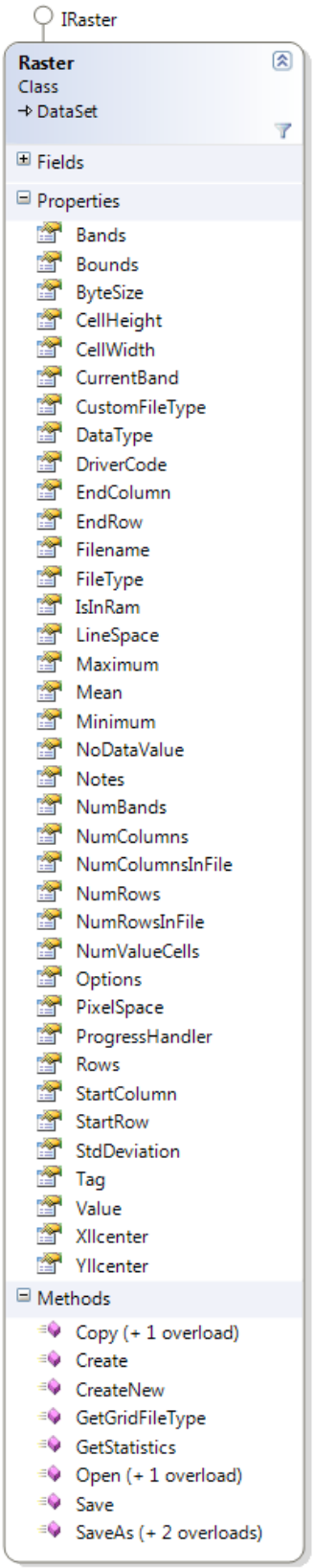


Figure 50: Raster Class

The other critical values are *NumRows* and *NumColumns*. These provide the developer with information on how to cycle through the values from the *Value* property. The final bit of information that is especially important is the *Bounds* property. Many of the properties in Figure 50 are just shortcuts to the *RasterBounds* class. The *RasterBounds* class is where information is kept that stores the geospatial positioning aspect of the raster. The *Bounds* property grants control of the position of a raster in space. The *NoDataValue* is also useful, as it can help change how statistics are calculated.

6.2.6. GridHeader

The *GridHeader* class was an important part of the grid. It defined the spatial locations like the lower left X and lower left Y coordinates as well as defining a cell size. One of the limitations of the MapWindow 4 grids was that they did not support skew terms. In MapWindow 6.0, the skew is fully supported and uses *AffineCoefficients* as the source of most of the other properties. However, in order to define the boundaries themselves, it is not enough to simply have the affine coefficients, which effectively describe the cell size and skew relationships as well as the position of the upper left cell. The number of rows and columns are needed in order to get a valid rectangle that can enclose the entire raster. The Envelope is a rectangular form that completely contains the raster. Figure 51 shows the *rasterbounds*.

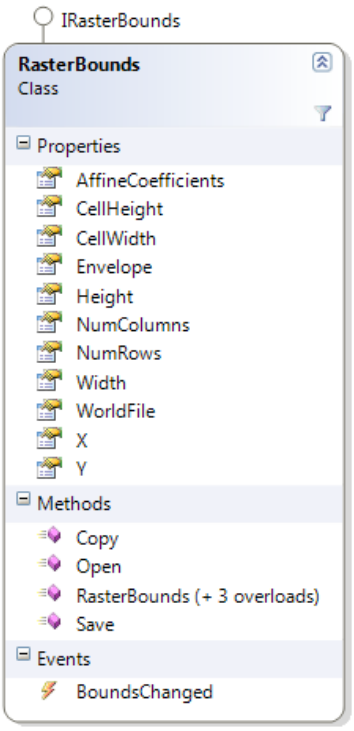


Figure 51: RasterBounds Class

6.2.7. Image

The *Image* class was specifically designed for working with images, and was not actually designed for data analysis. The new *ImageData* class supports a *Values* method similar to the values of a raster, but is an array of byte values instead of the raster data types. The format of the image changes how those byte values are organized. The *Stride* property is the number of bytes in a given row. The value of the *Stride* property is not always strictly related to the number of columns as some image formats use algorithms that may require slightly more columns than have actual data. The *BytesPerPixel* class tells developers whether they are working with a *GrayValue*, RGB or ARGB image. With images, the geospatial location and sizing is controlled via a *WorldFile*. Figure 52 shows the *ImageData* class and the *WorldFile* class.

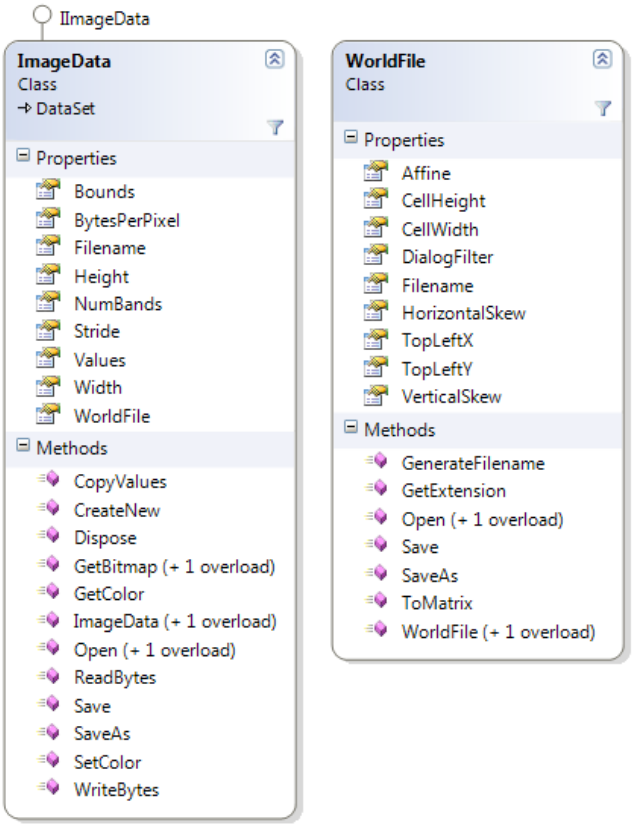


Figure 52: ImageData and WorldFile Classes

6.2.8. Extension Methods

Not all methods are supported directly on a raster. Many useful methods are supported in the form of "Extension Methods." For example, the raster includes useful extension methods that can alter the raster bounds, like *Translate*, or *Scale*, or *Rotate*. But other useful abilities like *CreateHillShade* use the raster values themselves in order to calculate a floating point value that helps to control the shaded relief aspect of any image that is created from a raster. Other methods like *GetRandomValues*, and *GetNearestValue* are helpful for doing analysis, but one of the most critical methods is *CellToProj* and *ProjToCell*, which allows the developer to easily go back and forth between geospatial coordinates and the row and column indices. Figure 53 shows the extension methods for the raster class.

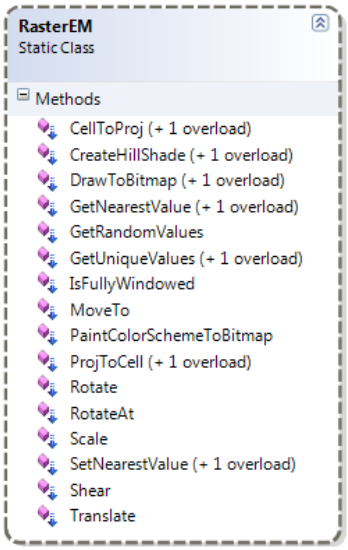


Figure 53: Raster Extension Methods

The *Feature* and, in fact, any class that implements the *IFeature* interface, are extended with the geometry methods that are so critical to vector calculations. These methods not only include the overlay operations like *Intersection*, *Union* and *SymmetricDifference*, but all the relationship tests that may be useful like *Touches* or *Within*. There are some bonus methods such as *Area*, which calculates the areas of polygons. Another bonus method is *Centroid*, which calculates the center of mass for geometries. Yet another method, *ConvexHull*, can be used to simplify a geometry by drawing straight lines past concave sections and following around with the convex portions. The *Distance* tool finds the minimum distance between two geometries, and the *IsWithinDistance* simply changes the Distance calculation to test it against a threshold. Figure 54 shows the extension methods for the feature class.

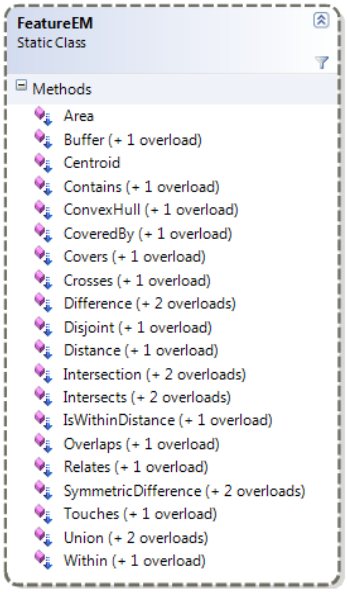


Figure 54: Feature Extension Methods

6.3.Symbology Architecture

In the previous section, the objects and classes used for accessing data objects were discussed. This section will show the classes that are involved in controlling how the data layers appear when they are represented on the map. In many cases, methods or properties are shared between many different types of classes. When that is true, the shared content can be organized into a shared base class, instead of each special case duplicating the same methods. For instance, when working with points, the symbolizer that is being used should be a *PointSymbolizer*. The *PointSymbolizer*, *LineSymbolizer* and *PolygonSymbolizer* all share a common base class called a *FeatureSymbolizer*. Figure 55 shows the class diagram for the classes responsible for feature symbolization. The bulk of the options are controlled by the *Symbols*, *Strokes*, or *Patterns*.

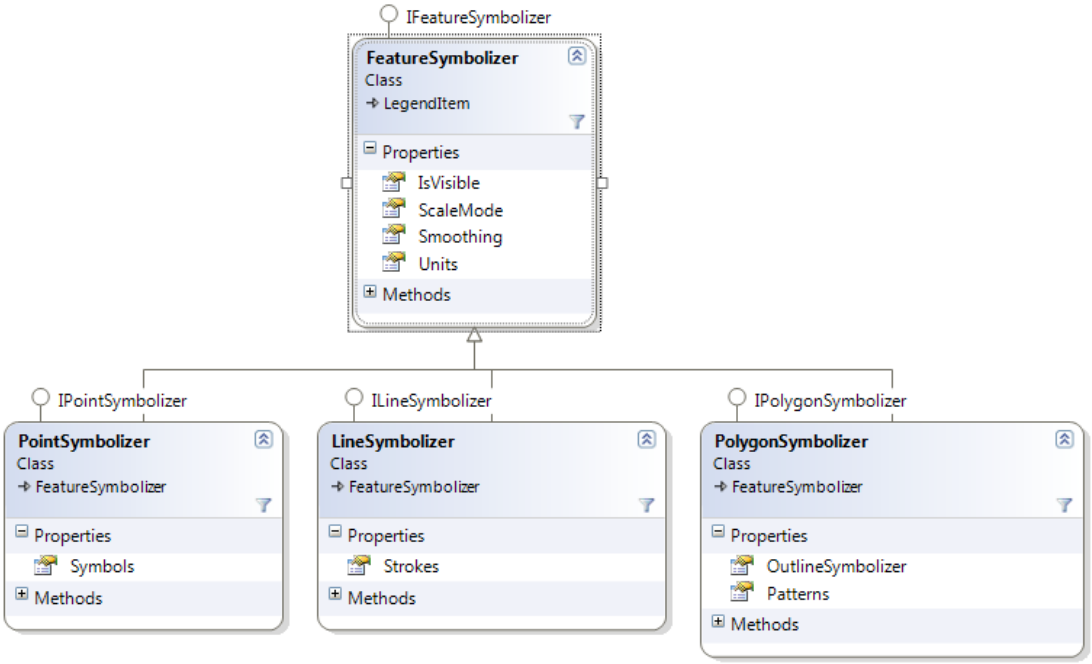


Figure 55: Symbolizer Class Diagram

To illustrate inheritance, the class diagram in Figure 55 shows the three main feature symbolizers: points, lines, and polygons. What is most important here is that

there are some characteristics that are shared. Properties, methods and events on the *FeatureSymbolizer* class also appear on each of the specialized classes. Meanwhile, each individual symbolizer class has a collection of *Stroke*, *Symbol* or *Pattern* classes that render the separate lines, icons, or fill styles. In the same way, categories, which group features based on their attribute values, and schemes, which are collections of those categories, have separate classes depending on whether they represent points, lines or polygons. However, casting the scheme to a *PointScheme*, for instance, would automatically provide access to the *PointCategories*, *PointSymbolizers*, and *Symbols* that are specific to working with points.

6.3.1. Layers

In geographic information systems, real world content is represented in the map using graphical representations of features. In order to make visual identification and analysis easier, the content is subdivided into separate themes. For instance, rivers are separated from roads, even though both are technically lines. The separation into thematic layers allows different combinations of content to be considered. Figure 56 shows an example of thematic layers being loaded into MapWindow 6. The legend control shows the separate layers, a checkbox to indicate whether each is visible, and a symbol representing the layer itself.

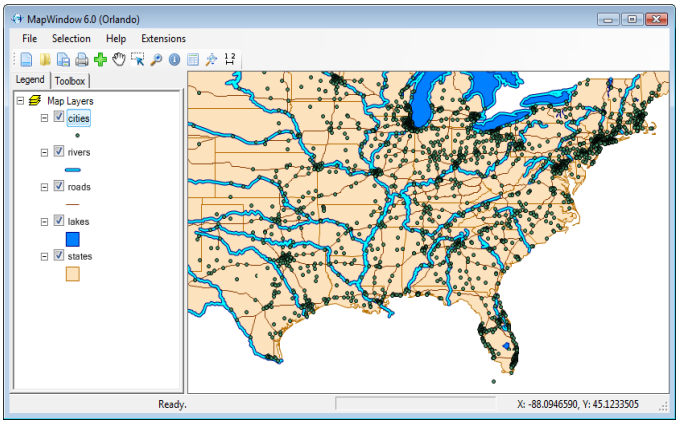


Figure 56: Thematic Layers

In the MapWindow 6 architecture, a layer represents the visual handle for data content. It effectively combines the symbolic drawing settings with the data values for the layer. The fact that the cities are drawn as green circles, for example, is controlled by the symbolizer on the layer, while the physical locations of the cities are controlled by the dataset itself.

In this way, the same data content could be used to show separate layers, each with a different symbology. The organization of the layers is important. Layers that are drawn first will get overwritten on the map by layers that are drawn later. The legend shows the layers that get drawn last as physically on top. The drawing sequence can be controlled by dragging legend items, but this directly affects the underlying sequence of the layers in the *MapFrame* or group. Another responsibility for the layer is to control which features are selected, and to allow those features to be drawn differently. Figure 57 lists the feature layer classes and shows their relationships.

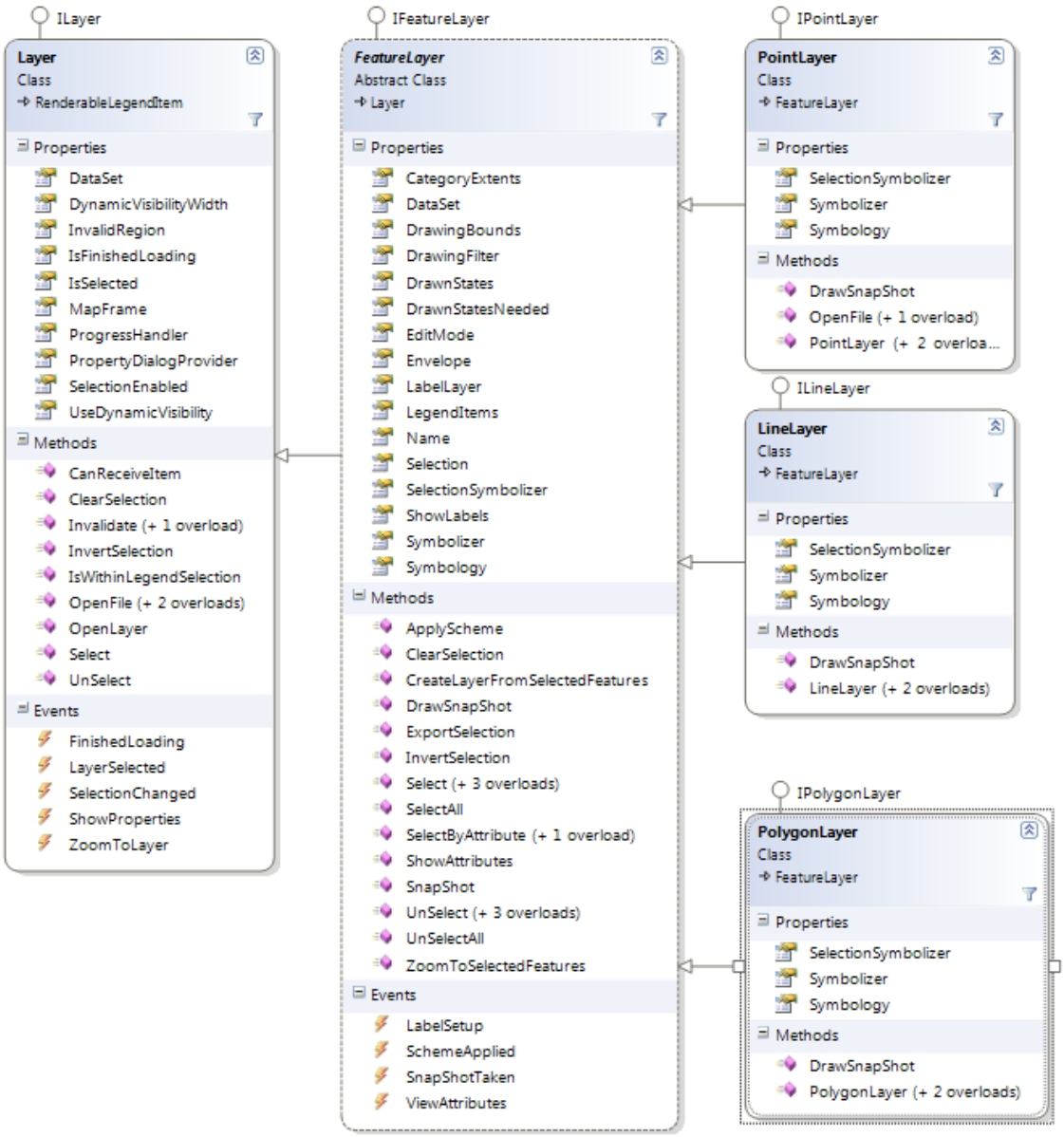


Figure 57: Feature Layer Classes

6.3.2. Rendering Layers

In the early design stages, the feature layer classes were capable of drawing using the GDI+ graphics object. This project also looked at the creation of working vector layers in a 3D environment. Because much of the functionality of thematic layers can be shared between environments, independent from the drawing mechanics, it makes sense to have the non-rendering content on a shared base class. The other criterion was for

future developers to be able to provide their own custom layers. An example might be a network layer that requires both the nodes and the lines between them to be rendered. So if the rendering itself is controlled through a fixed set of supported drawing instructions (e.g. draw lines, draw images, fill polygon) then the custom methods can get no more sophisticated than what the core application supports without completely replacing the drawing engine for all the other built in layers. Therefore, this project features rendering techniques that are tightly correlated with the layers. A *MapLineLayer*, for instance, inherits from the generic *LineLayer* class, which specifies the symbology and data values to draw, but then extends that base class with an appropriate Draw or Print method for handling the 2D rendering itself, but only for the type of information for that layer. A 2D Map, then, must be made up of 2D drawing layers, but the exact functionality of those layers is extensible. The rendering for 3D drawing is by necessity different from how the 2D drawing is handled. But since the rendering is in fact associated with the specific layer types, it becomes possible for future developers to extend the type of rendering layers supported.

6.3.3. Categories

The basic unit that helps clarify the symbology design is the category. Geospatial content is typically already organized into thematic layers, where every member of the layer is the same feature type and represents real world items that are closely related. An example might be a layer showing rivers that is separate from a layer showing roads. Thematic layers allow the easy re-arrangement of different kinds of geospatial information for cartography or analysis. It is frequently desirable to be able to subdivide the content into separate, smaller layers. In ArcGIS and several of the open source GIS

software sets, it is possible to use the attributes in order to automatically create sub-categories within the layer. In the legend control, these categories are usually fixed in place, or at least cannot be removed from the parent layer. However, each category can take on a different appearance. Figure 58 shows how the layer is subdivided in the legend. In this instance, the unique values of the state name field have been used to create the categories. These categories are organized into a collection called a scheme, which lists each of the categories. The layer, scheme and categories are all classes that are legend items, and therefore will appear visibly in the legend.

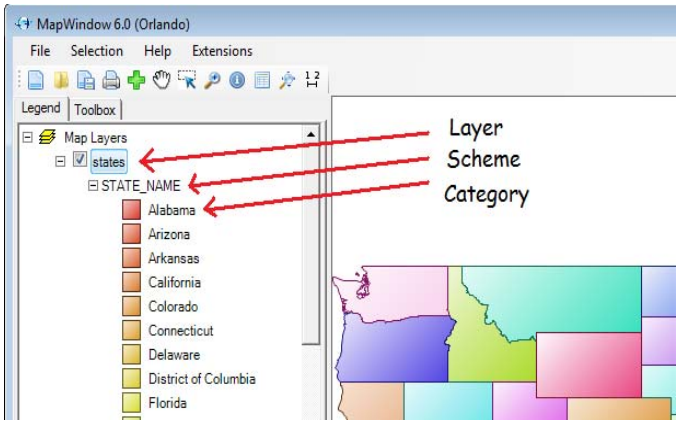


Figure 58: Layer, Scheme, and Category in Legend

The categories are organized so that shared information is stored on a base class called the *Category* class. Categories can be used by more than just vectors though, as they are also used for raster or label layers. The *FeatureCategory* class is the common content for all the vector categories. This class includes a filter expression that controls which features will appear in the category based on their attributes. A *FeatureSymbolizer* has access to some general appearance information, but in order to get precise control of appearance, it is necessary to use the sub-classes. The sub-classes expose the full symbol set through the *Symbolizer* property. The category classes currently also support independent selection symbolization, though by default the selection symbology will be a

cyan colored version of the regular symbology. Figure 59 shows the category classes and their specific subtypes for feature categories.

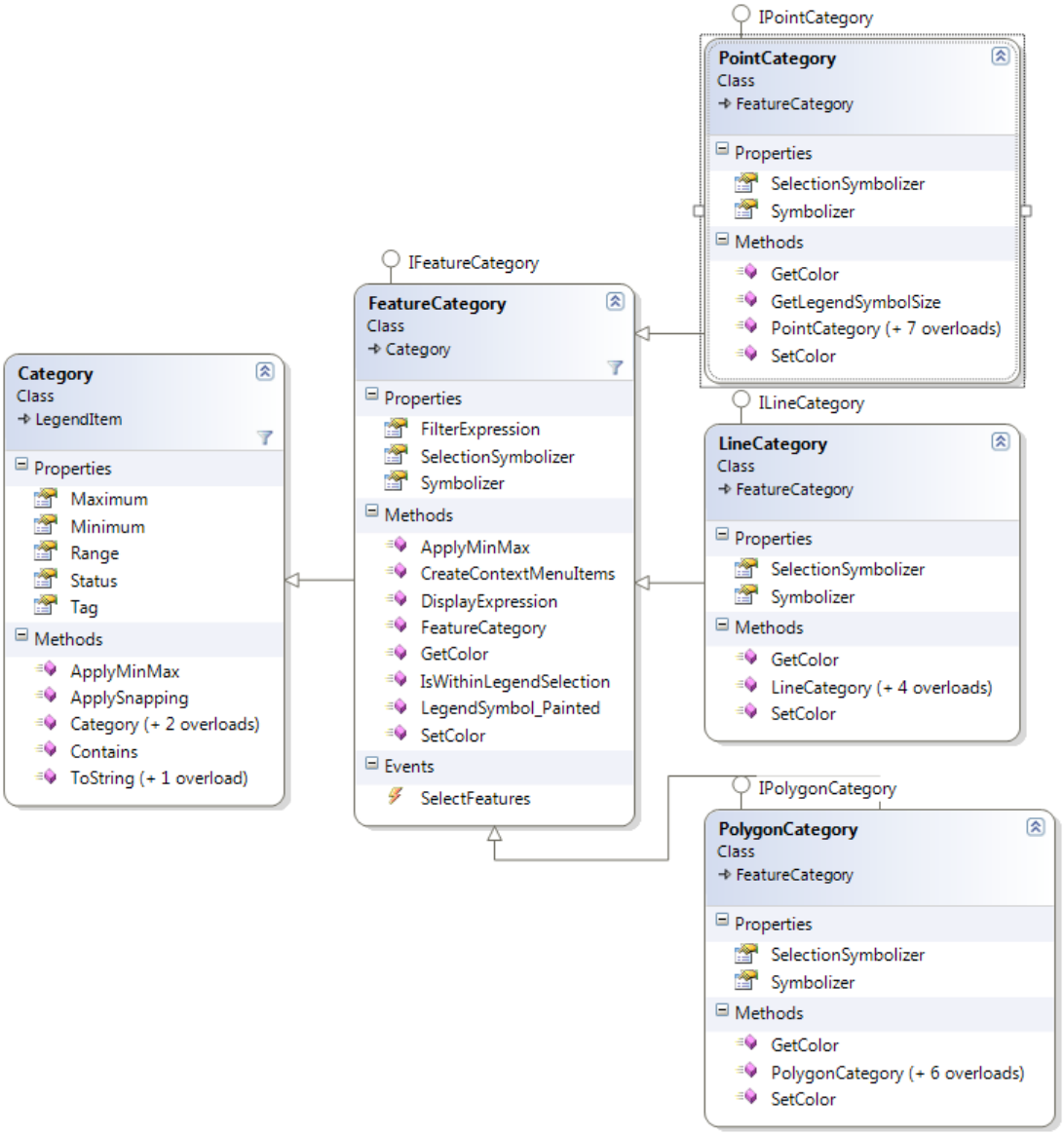


Figure 59: Feature Category Architecture

6.3.4. Schemes

In addition to acting as a collection of categories and potentially showing up in the legend, schemes also act as a tool for building certain standardized sets of categories. Figure 60 shows the collection of classes used for feature schemes.

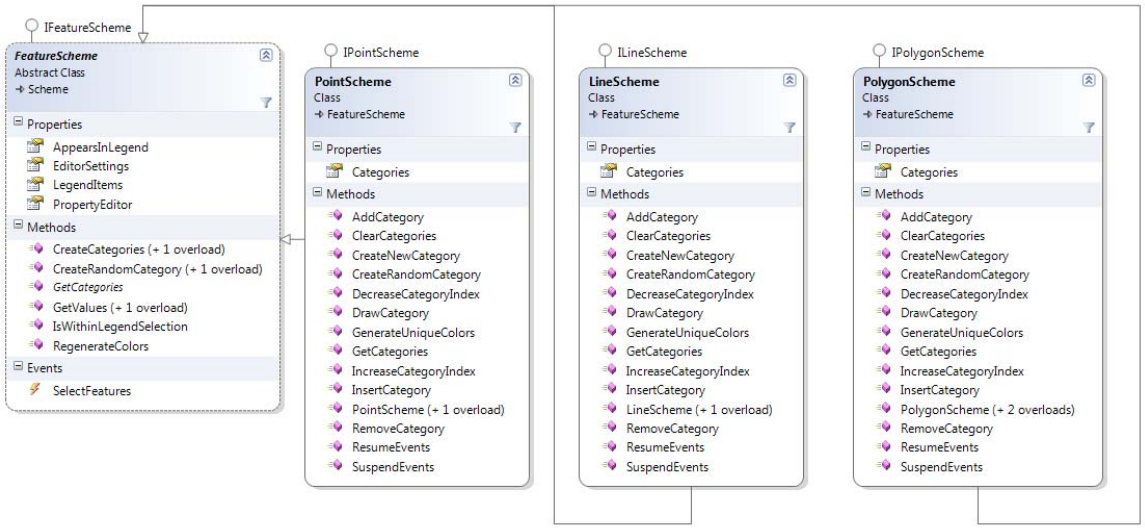


Figure 60: Feature Scheme Classes

There are a large number of settings that can be controlled directly using the various scheme classes. The scheme can be created based on numeric values according to statistics, or else it can be quickly created using unique values from a field. The categories can always be edited programmatically after they are created, but this simply controls what will happen when the *CreateCategories* method is called. As of the Sydney alpha release (Oct. 2009) not every *IntervalMethod* is supported, but *Quantile* and *EqualInterval* are both supported. The interval snap methods include none, rounding, significant figures, and snapping to the nearest value. These methods can help the appearance of the categories in the legend, but it can also cause unexpected problems. In the case of small values, the rounding digits should be set to a fairly high number or else many categories simply show a range of 0 – 0 and have no members. With *SignificantFigures*, the *IntervalRoundingDigits* controls the number of significant figures instead. The progression of colors, widths or sizes can be controlled as a gradual change across the created categories through the *RampColors* property, or the categories can be assigned randomly, using the size or color range as the extreme values for the random

creation. The *TemplateSymbolizer* property allows for control of the basic appearance of the categories for any property that is not being controlled by either the size or color ramping. For example, adding black borders to star shaped point symbols can be handled using the template symbolizer. In this case the template is set up so that the symbols are star shaped and have equal sizes since *UseSizeRange* defaults to false, but have different colors because *UseColorRange* defaults to true. Figure 61 shows the class diagram of the settings that can be used to control feature schemes.

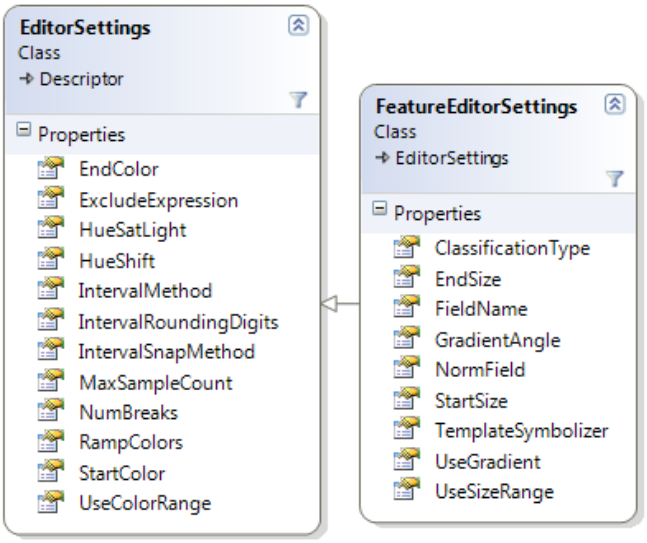


Figure 61: Available Feature Editor Settings

The settings shown in Figure 61 represent only a small portion of the scheme options that are programmatically available. A user can control the color range, whether or not the colors should be ramped or randomly created, a normalization field, an exclusion expression to eliminate outliers and in the case of polygons, a consistently applied gradient.

6.3.5. Point Symbol Classes

One of the new additions to the symbols' functionality no longer restricts developers to representing things using a single symbol. Complex symbols can be created simply by adding symbols to the *Symbolizer.Symbols* list. There are three basic categories of symbols, *Simple*, *Character* and *Image* based. These different categories have some common characteristics, like the *Angle*, *Offset* and *Size*, which are stored on the base class. In the derived classes, the characteristics that are specific to the sub-class control those aspects of symbology. When creating new symbols the *Subclass* can be used. When working with individual symbols in the collection, it may be necessary to test what type of symbol is being manipulated before its properties can be controlled. Figure 62 shows the class diagram for the classes that can describe point symbols.

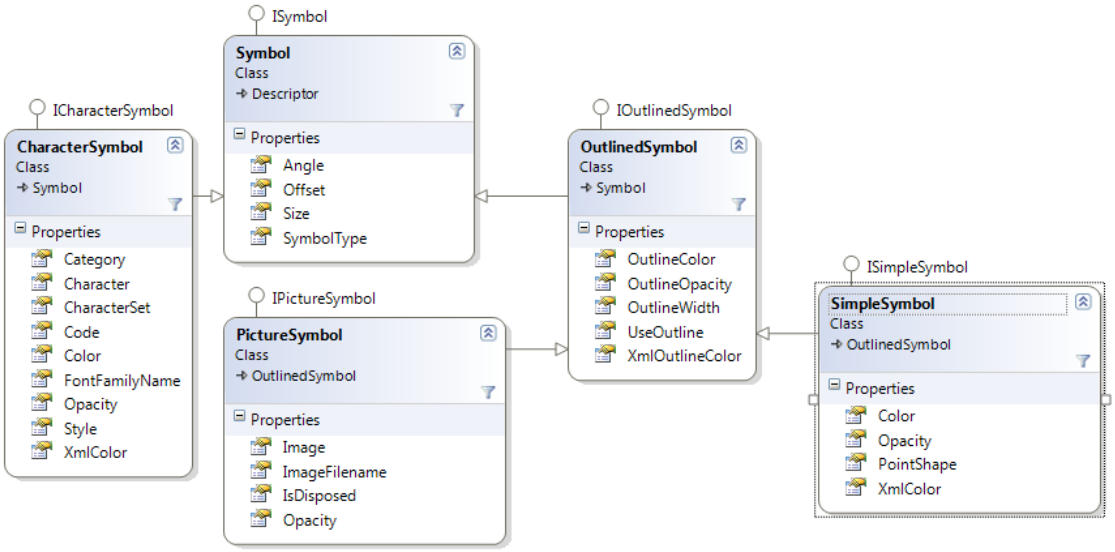


Figure 62: Point Symbol Class Diagram

6.3.6. Line Symbol Classes

Each individual *LineSymbolizer* is made up of at least one, but potentially several strokes overlapping each other. The two main forms of strokes that are supported

natively by MapWindow are *Simple Strokes* and *Cartographic Strokes*. Cartographic strokes have a few more options that allow for custom dash configurations as well as specifying line decorations. Decorations are basically just point symbols that can appear at the end of the stroke, or evenly arranged along the length of the stroke. Figure 63 shows the class diagram for the stroke classes that control drawing line features.

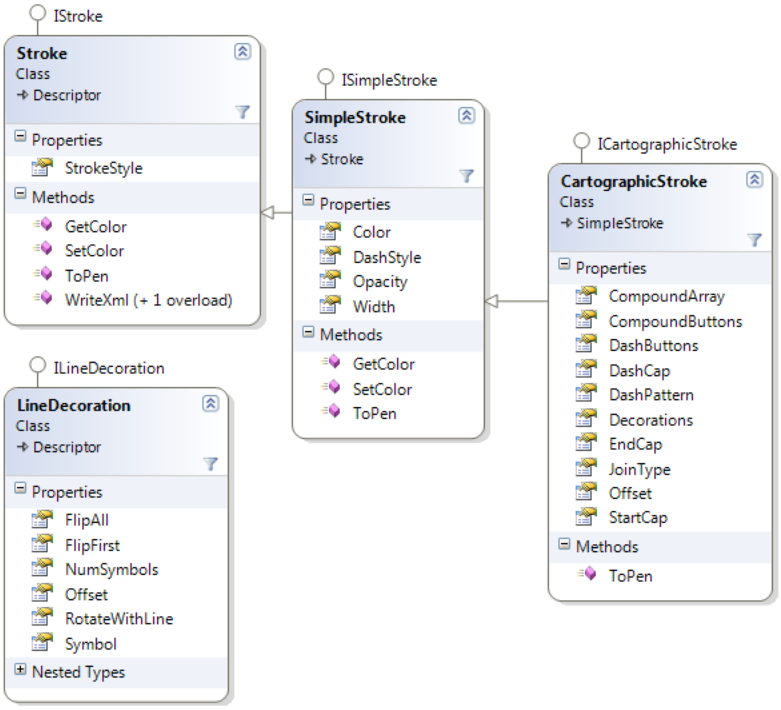


Figure 63: Stroke Class Hierarchy

6.3.7. Polygon Symbol Classes

Like the other previous symbolizers, polygon symbolizers can be built out of overlapping drawing elements. In this case they are referred to as patterns. The main patterns currently supported are *Simple*, *Gradient*, *Picture* and *Hatch* patterns. *Simple* patterns are a solid fill color, while *Gradient* patterns can work with gradients set up as an array of colors organized in floating point positions from 0 to 1. The angle controls the direction of linear and rectangular gradients. *Hatch* patterns can be built from an enumeration of hatch styles. *Picture* patterns allow for scaling and rotating a selected

picture from a file. Figure 64 shows the classes that are used to control the patterns that are used for drawing polygons.

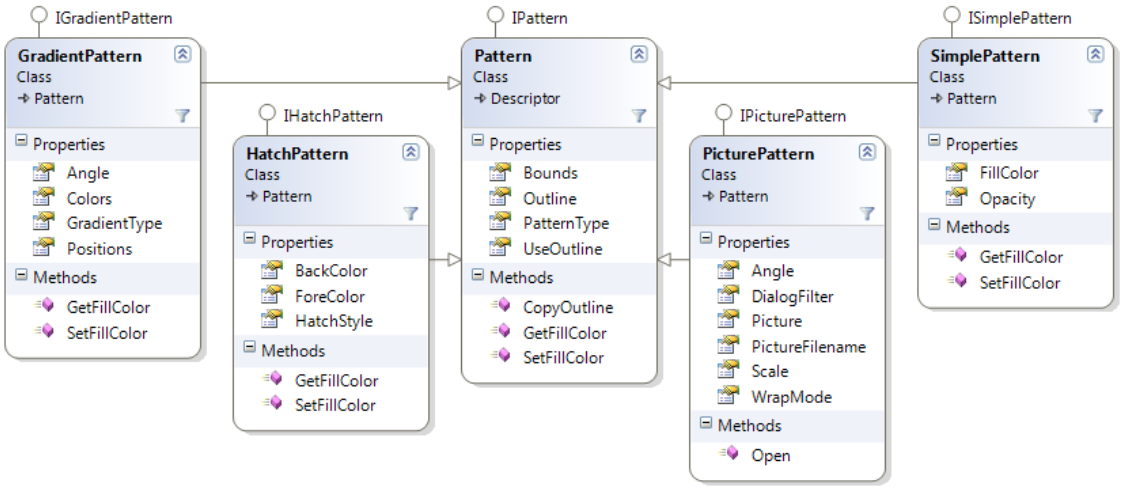


Figure 64: Pattern Class Diagram

6.3.8. Raster Symbology Classes

Figure 65 shows the class diagrams for the raster classes. The raster organization mimics the vector organization in some ways, but there are some important differences. For vectors, the symbolizer class is distinct for each category. The *RasterSymbolizer*, however, can contain a *ColorScheme* that can be made up of several *ColorCategories*, but the categories and scheme belong to the symbolizer instead of the other way around. The reason for this arrangement is that rasters have far less complexity in the possible classification of data values since they are made up of a scalar field of numeric values, and would not be expected to support complicated filter expressions.

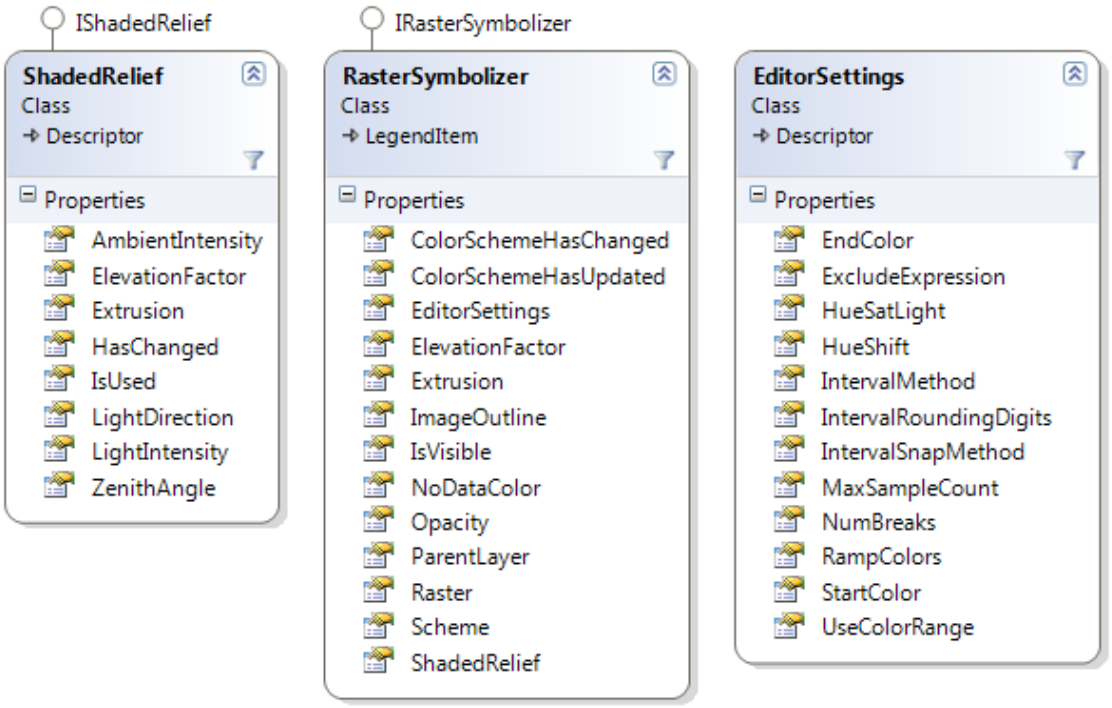


Figure 65: Raster Symbology Classes

7. Implementation and Demonstration

This chapter describes the implementation of the MapWindow 6 design presented previously. The implementation and demonstration is given in the form of a developer tutorial illustrating how a third party programmer could use the tools in a custom application.

7.1. Assembling a Map Project

The ready-built mapping controls allow developers to have a new design for each new GIS application. Similar to standard .NET controls, they can be dragged and dropped onto a form or user control. Because the components are largely independent, they can be re-arranged in different layouts, or used separately. Furthermore, the extensive use of interfaces allows a control such as the map control to be used interchangeably with alternate controls that act like legends.

The following demonstration is an illustration of how these components can be used, and illustrates the flexibility of the design. In this scenario, the tasks are broken into steps to allow this document to also serve as reference for re-creating custom mapping applications with the components from this study.

7.1.1. Step 1: Start a New C# Application

The first step is to create a brand new application. Rather than working with an existing application, the goal here is to show that getting from a blank project to a fully operational GIS takes only a few minutes in order to add the components and link them together. In addition, no programming is required. To get to a new project dialog in visual studio, it is necessary to navigate to *File*, *New*, and choose *Project* from the

context menu. This menu selection will display the *New Project* dialog displayed in Figure 66.

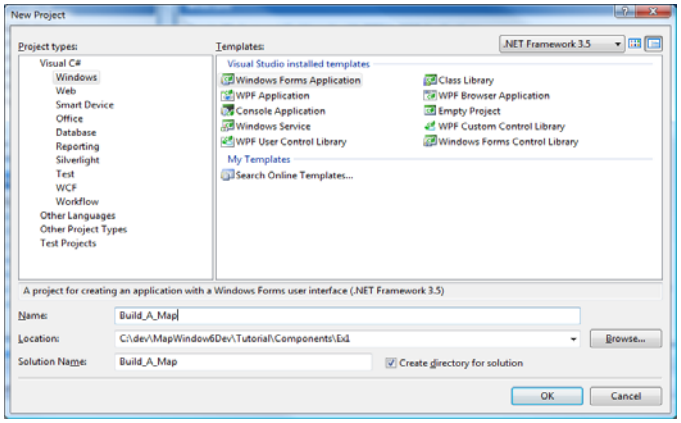


Figure 66: New Project Dialog

The name and location fields allow developers to change the name and path. In this case, the project was named *Build_A_Map* and the file location was specified as C:\dev\MapWindow6Dev\Tutorial\Components\Ex1. In this case the project type was selected to be a *Visual C#, Windows, Windows Forms Application*.

7.1.2. Step 2: Add MapWindow Components

The next step is to ensure that all of the designer controls and components have been loaded into the MapWindow toolbar. As this tutorial was created in the alpha stage of development not all of the controls have unique representative icons. There is also a large assortment of extra controls that were used for designing MapWindow 6.0 that are not in the public release. However the basic technique is the same. First, a new tab is created to store the MapWindow tools. Right clicking on the toolbox and selecting the *Add Tab* option from the context menu adds a new tab that can be re-named. In this case the tab is named MapWindow. Figure 67 illustrates the *Add Tab* context menu option.

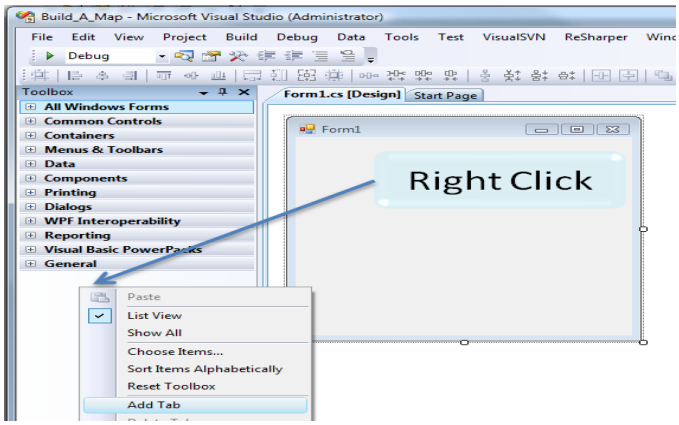


Figure 67: Add MapWindow Tab

A tab allows all the controls from the MapWindow library to be kept together. Once the MapWindow tab is added, the next step is to right click in the blank space below that tab and select *Choose Items* from the context menu. Figure 68 shows the *Choose Items* context menu item.

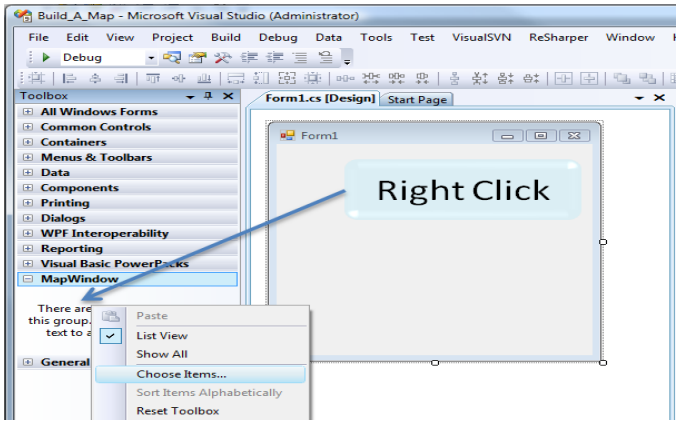


Figure 68: Choose Items

The *Choose Items* option launches a new dialog that will show various pre-loaded .NET controls as well as some Component Object Model (COM) controls. It also displays a button that allows browsing through the folders to find a specific Dynamically Linked Library (.dll) file. In this case, the MapWindow.dll file should be selected. Figure 69 shows the location of the browse button on the dialog.

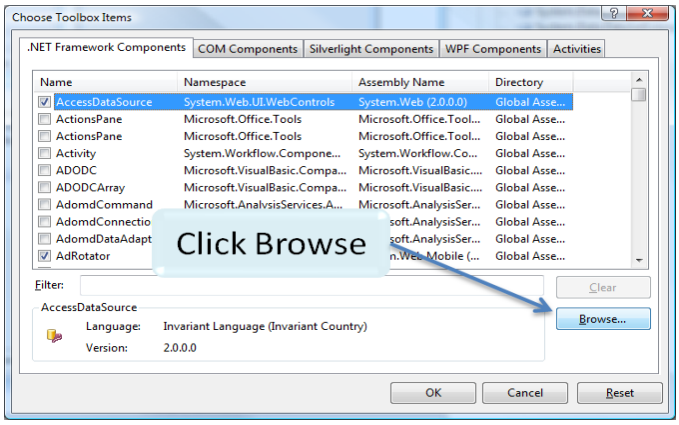


Figure 69: Browse

On the machine used in this example, the MapWindow dll file was in the C:\dev\MapWindow6Dev\bin folder. Figure 70 shows the MapWindow.dll.

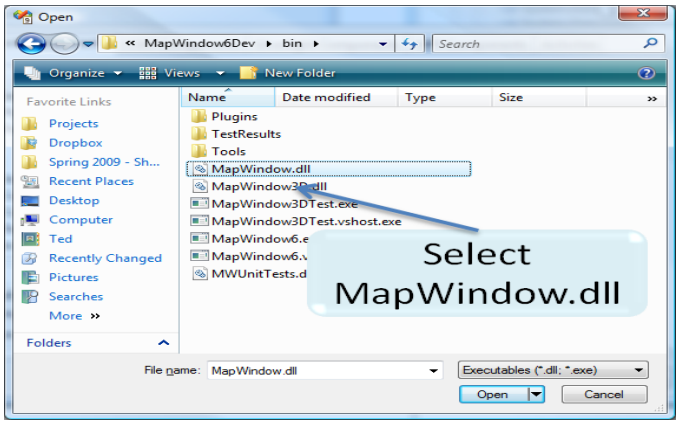


Figure 70: Select MapWindow.dll

Once this file is selected, it will cause a large number of tools to be added to the toolbox, as is illustrated in Figure 71.

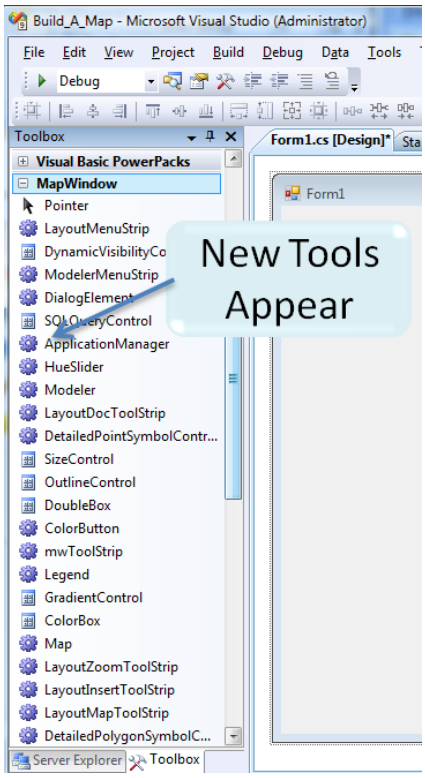


Figure 71: MapWindow Tools

7.1.3. Step 3: Add Menu and Status Strips

Adding a .NET *MenuStrip* is a fast way to give access to a very versatile number of tools, options, or other capabilities. Adding a menu strip is not in any way required by the MapWindow GIS components, but rather is simply a convenient starting point for a new application. The *MenuStrip* is found under the *All Windows Forms* tab in the toolbox. Dragging the *MenuStrip* item from the toolbox to the main form will add a *MenuStrip* control to the empty form, as is illustrated in Figure 72.

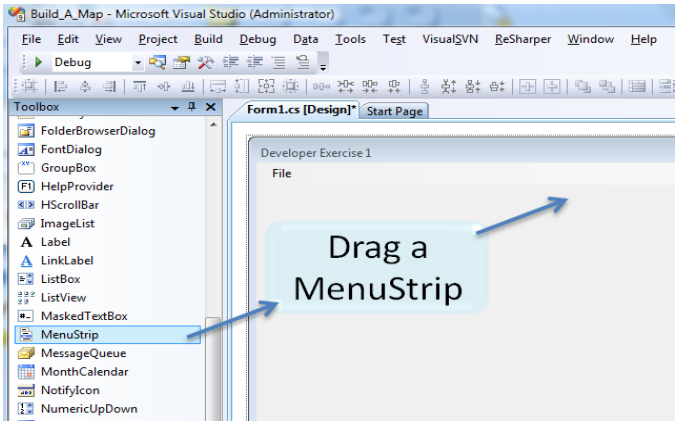


Figure 72: Drag a MenuStrip

The next step is to return to the *MapWindow* tab in the toolbox and add two controls that are directly associated with the MapWindow components. The first is the *mwToolStrip*, which lists several basic GIS functions like adding data layers, switching between zoom and pan mode, and zooming to the full extent. The second is the status strip. Both are shown in Figure 73.

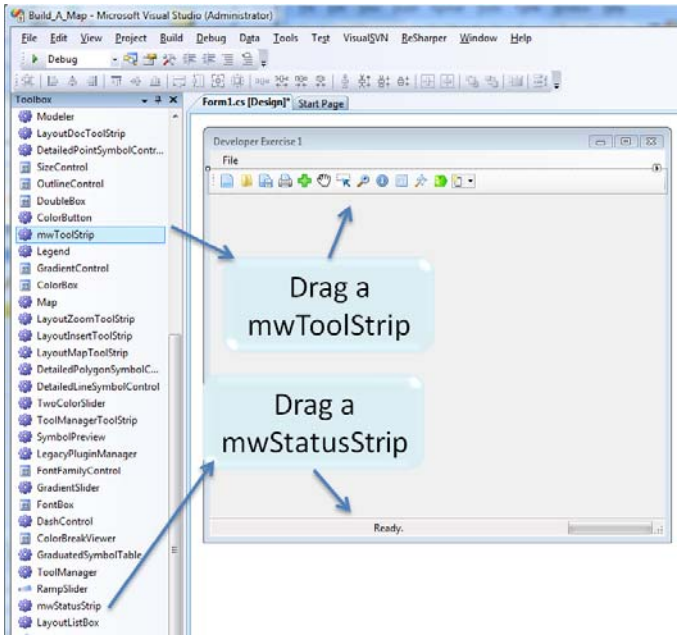


Figure 73: Add Status and Tool Strips

7.1.4. Step 4: Add the Map

Now that the peripheral controls have been added, the next task is to start adding the controls that will work within the central part of the map project. In order to cleanly divide up the screen areas with a minimum of custom programming, this demonstration takes advantage of a .NET *SplitContainer* control. The split container divides the content into two separate panels that are sizeable by the user. The split container is shown in Figure 74.

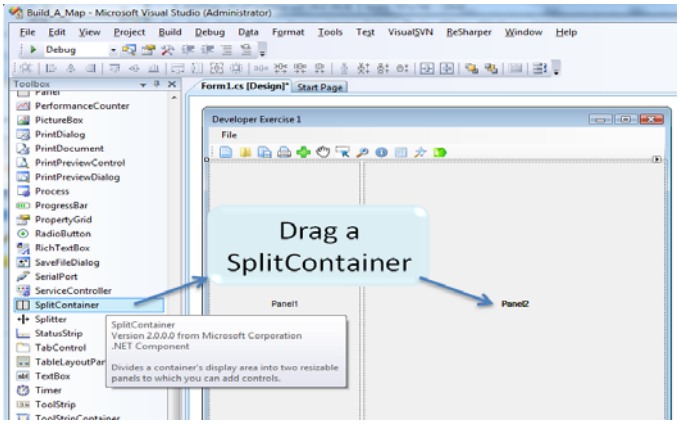


Figure 74: Add a SplitContainer Control

Figure 75 shows adding a map to the project in the right panel of the *SplitContainer*.

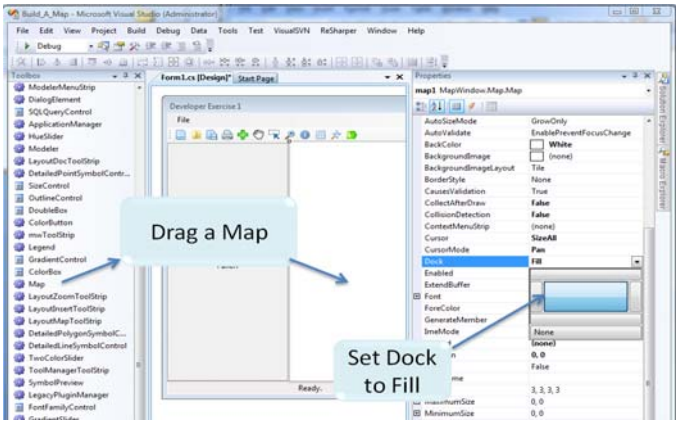


Figure 75: Add a Map

When adding the map, it will likely be the wrong size for the panel that has been created. In order to allow the user to resize the map so that it is always the right size for

the panel, the “Dock” property needs to be changed to “Fill.” Setting the property can be achieved by choosing the central rectangle in the drop down editor that appears in the property grid when the down arrow is clicked.

7.1.5. Step 5: Add the Legend and Toolbox

Because the Legend and Toolbox don’t have any interactions with each other, and only work with the map, it is not critical to have both the Legend and Toolbox visible at the same time. In this specific demonstration, the .NET Tab control allows the same space to be used on the form for both the Legend and the Toolbox. The behavior of the components does not depend on the organization within the active controls, so a Tab control is not strictly required. Split panels, fixed panels, or other container controls are all acceptable locations for MapWindow controls. Figure 76 shows the tab control being added to the layout.

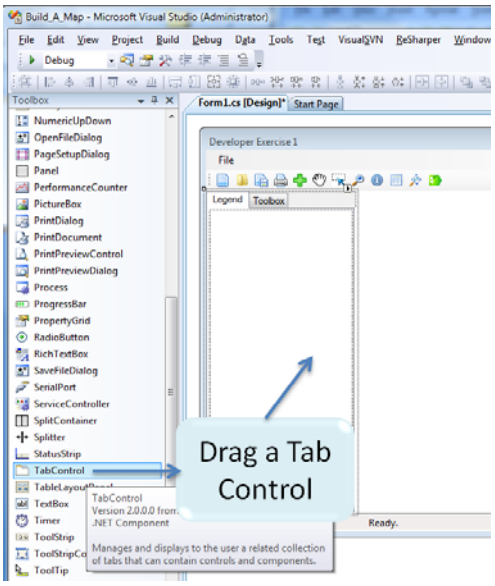


Figure 76: Add a Tab Control

In this instance, the tab control is added to the left panel, and the text on the two tabs is updated to read “Legend” and “Toolbox.” Changing the tab text can be done through the property grid that appears when the tab control is clicked after it has been

added to the main form. Figure 77 shows the legend control being added to the left tab and being set to fill the tab.

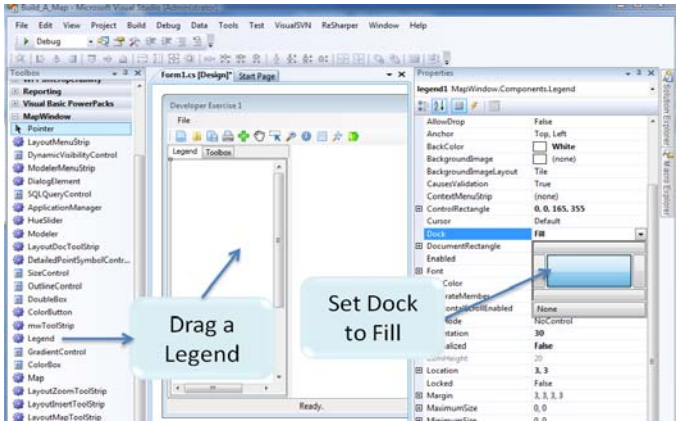


Figure 77 : Add Legend

Adding the legend follows similar rules to adding the map. The legend can be dragged over the tab control once the *Legend* tab is selected. Adding the legend to a tab will automatically tell the designer that the legend control will only appear when the *Legend* tab is selected. Setting the dock property to fill allows the splitter control to resize the Legend. Figure 78 shows the Toolbox being added to the project in the other tab.

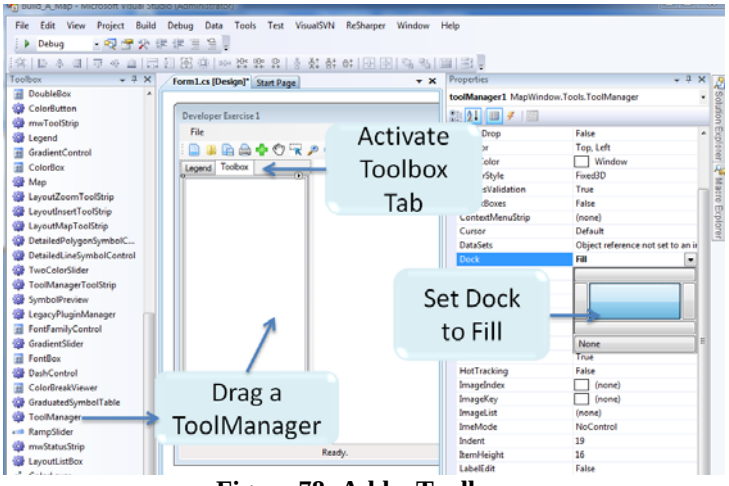


Figure 78: Add a Toolbox

This exercise doesn't completely activate the Toolbox. To do so would require adding the MapWindowTools.dll file, or any other dynamically linked library file that has ITool classes within it to a *Tools* subfolder in the project output directory. To add the Toolbox, simply switch the tab control to the *Toolbox* tab. Then, drag and drop the control named *ToolManager*. Like the Legend and the map, the dock property can be set to fill. The *Legend* tab should be activated instead of the *Toolbox* tab so that when the application starts, it defaults to showing the Legend, rather than the Toolbox.

7.1.6. Step 6: Link It All Together

At this stage, the project can be run, but it would not appear to do anything. In order for the status strip to show updates from the map, and in order for the Legend to show the layers from the map, the components need to be linked together. The map has a property for the Legend and *ProgressHandler*. In this demonstration, the Legend and *ProgressHandler* properties are set to the instance of the components that have already been added, which still have the default names *legend1* and *mwStatusStrip1*. Figure 79 shows how to use the property window to set these properties.

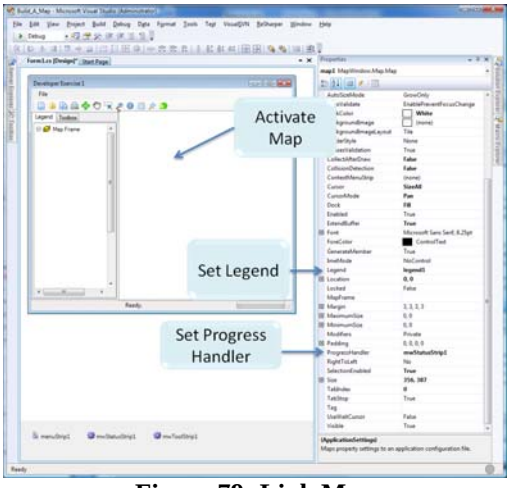


Figure 79: Link Map

Figure 80 shows how to connect the tool strip to the map. This connection can be made by activating the *mwToolStrip* control, choosing the *Map* property and selecting *map1*.

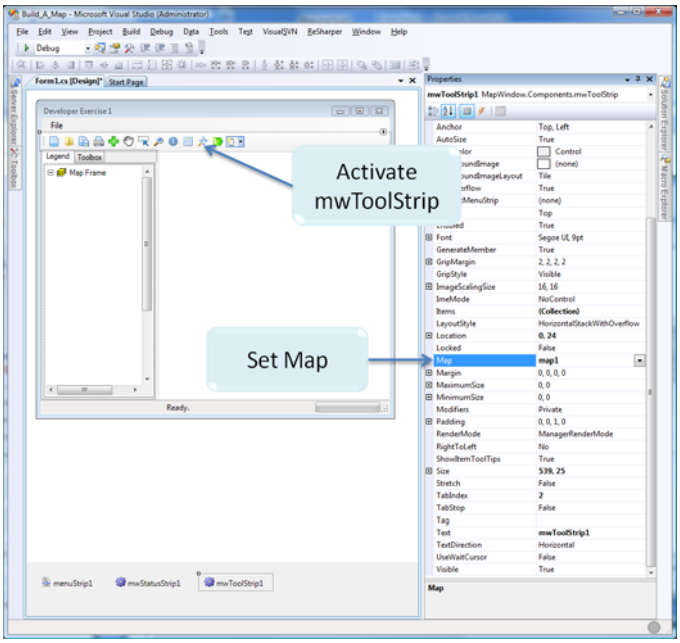


Figure 80: Link Tool Strip

Now that the project is connected together, it is ready to be used in order to draw some test data. Almost any data in shapefile format will do, but this demonstration features some examples of online data sources that can be used.

The raw data in this case is stored in the form of a zip file. Windows Explorer can unzip files automatically, but a free, open source unzip utility called 7-zip is available for download from http://www.7-zip.org. Once the files are downloaded and extracted, there will be several files with the same name but different extensions. These extensions include .shp, .shx and .dbf. This group of files defines the ESRI shapefile format, and is a commonly used format for GIS analysis because it is portable and has an open standard, which makes it widely compatible between different GIS software vendors. Figure 81 illustrates the counties of the continental United States.

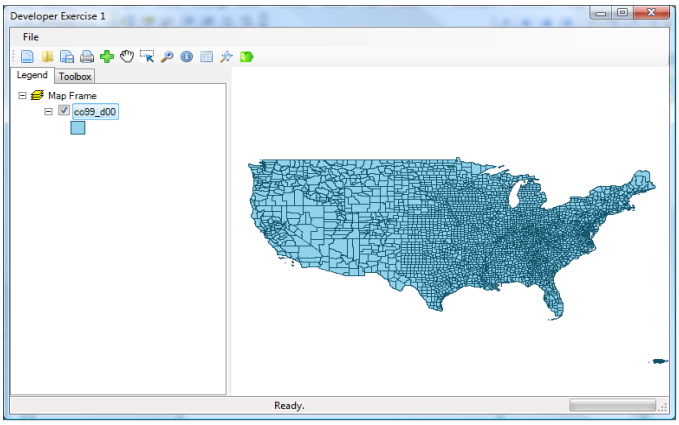


Figure 81: Census Data

Because the data also include counties in Hawaii, Alaska and Puerto Rico, it will be necessary to zoom in a little to see a view like the one above. The map starts automatically in the *Zoom* mode. In this mode, users can click to zoom into an area or else they can drag a rectangle around an area of interest. Clicking on the hand in the tool strip will put the project in *Pan* mode where a user can click on the map with the left mouse button and drag the map in a direction to have it update the view. They can also use the mouse wheel to zoom in or out of the scene. The basic operation of the map control follows the same instructions illustrated in Appendix A, which gives detailed operational instructions for MapWindow 6.0.

7.2. Simplifying Australia Data Layers

7.2.1. Step 1: Download Data

For this exercise, some thematic vector layers will be downloaded and simplified. After this, some basic programmatic techniques will be shown for controlling the symbology of the data layers. As this document was written on the heels of the Sydney Free and Open Source Software for Geospatial (FOSS4G) 2009 Conference, the layers will be some vector content representing Australia. Sometimes basic GIS data are offered on sites for free in the hopes of promoting the data with greater detail, and the dataset for this exercise was one such dataset.

http://www.usgsquads.com/prod_digital_international_vector_maps.htm

For this exercise, the *AUS Cities*, *AUS Lakes*, *AUS Political*, *AUS Populated Areas*, and *AUS Major Transportation* shapefiles were downloaded. These files are stored in zip format, so they have to be unzipped first (see exercise 1). The *cities* shapefile in this example is misleading because it contains mostly very small community centers or outposts. The *populated areas* polygon can be used to create a better representation of major cities in Australia. The MapWindow 6.0 application itself can be used to convert the populated areas polygons into cities.

7.2.2. Step 2: Calculate Polygon Areas

Figure 82 shows that clicking on the green plus icon will launch an open file dialog that can be used to add new layers to the map.

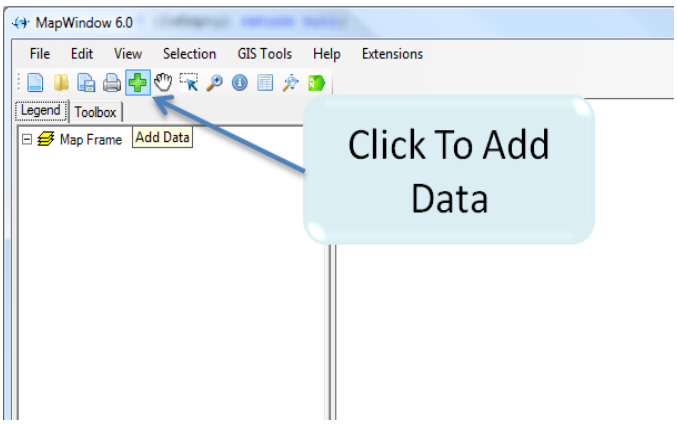


Figure 82: Add Data

After clicking the green plus to open a file dialog, it is possible to browse for the australia_populated_areas.shp shapefile downloaded and extracted earlier. When it opens, the polygons that represent large city areas become visible, mostly at the outer limits of Australia. Since the polygons are small compared to the size of the entire continent, it may be necessary to zoom in before the fact that the features are polygons becomes apparent. Figure 83 shows the specific polygon that corresponds to Sydney.

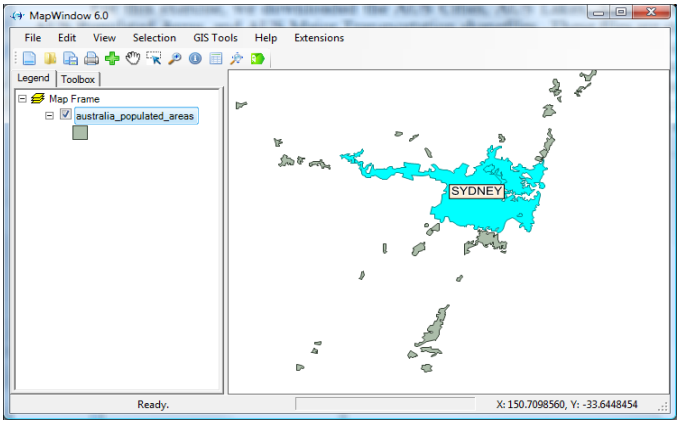


Figure 83: Sydney Polygon

The next step illustrates how to distinguish the largest cities from the other polygons. Because the downloaded polygons do not contain any information about the population or area, the *area calculation* tool can be used to calculate areas for each of the polygons. The units of the area will be largely meaningless because the linear units are in

decimal degrees, but the values will be useful for separating the larger cities from the small ones. Figure 84 demonstrates that in order to see the Toolbox, the *Toolbox* tab should be activated. Double clicking on the *calculate areas* tool will launch the tool dialog, which allows the *area* tool to be configured and an output dataset to be specified. Running the tool will create a new output shapefile.

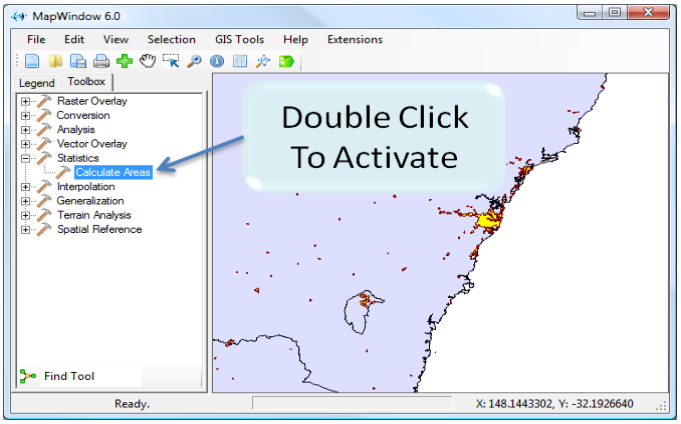


Figure 84: Calculate Areas

The resulting shapefile has to be re-added to the map. In this case, a new field has been added to the resulting shapefile that shows the area. While this is not an accurate assessment of population, at least it allows identifying the largest urban areas. Figure 85 shows the newly added areas field that appears in the attribute table for the output shapefile. The attribute table can be activated from the Legend by right clicking on a layer and choosing View Attributes from the context menu.

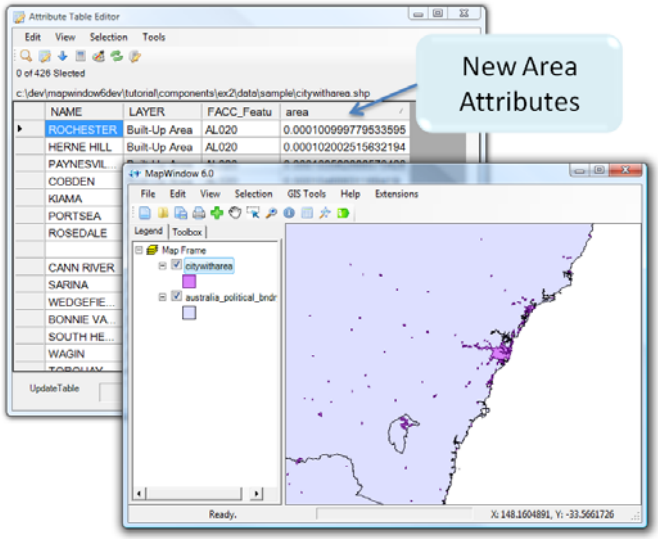


Figure 85: Newly Added Areas

7.2.3. Step 3: Compute Centroids

Now that the cities can be ordered by way of areas, a point layer of cities can be created from the existing polygons. The *centroid* tool will calculate the centroid for all the polygon shapes in a layer, and export as a new shapefile the point locations of those centroids. Figure 86 shows the *Generate Centroid* tool in the Toolbox. Because the attributes are retained, the resulting points will still have information about the areas that were calculated from the polygons.

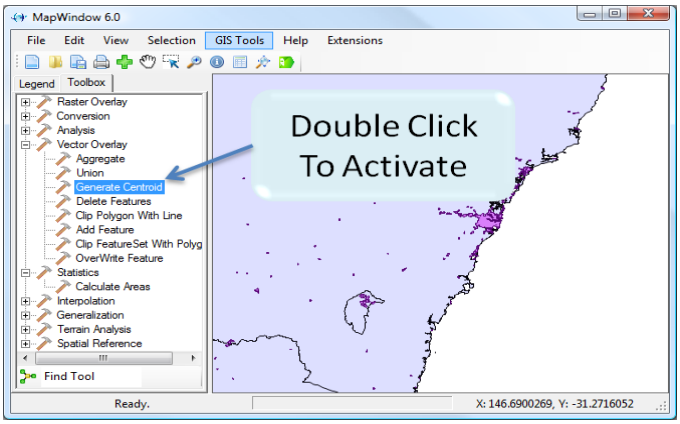


Figure 86: Calculate Centroid

The shapefile created by calculating the centroids will now be an accurate representation of the city locations for built up areas. In addition, it establishes an

approximate measure of the size of the built up area using the *area* attribute. Figure 87 shows that the initial shapefile has enough centroid locations that the points overlap.

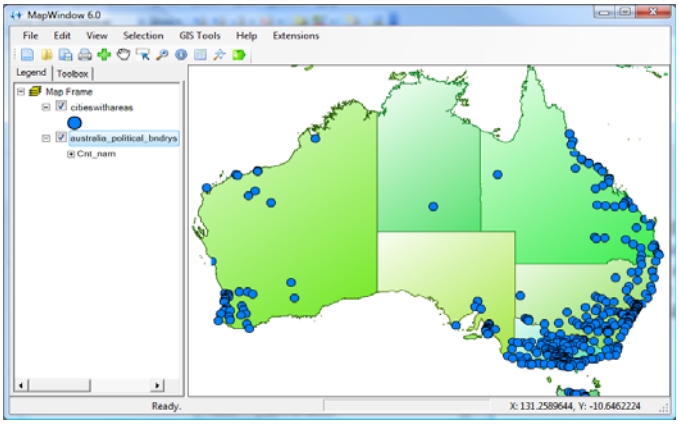


Figure 87: Too Many Cities

7.2.4. Step 4: Sub-sample by Attributes

In MapWindow 6.0, the menu has been outfitted with a *Selection* menu heading. Under that heading is the *Select by Attributes* option. Choosing this option will launch a dialog that will allow the use of a query dialog. Figure 88 shows the filter to be [Area] > .01, which will select the centroids that correspond to the largest populated area polygons.

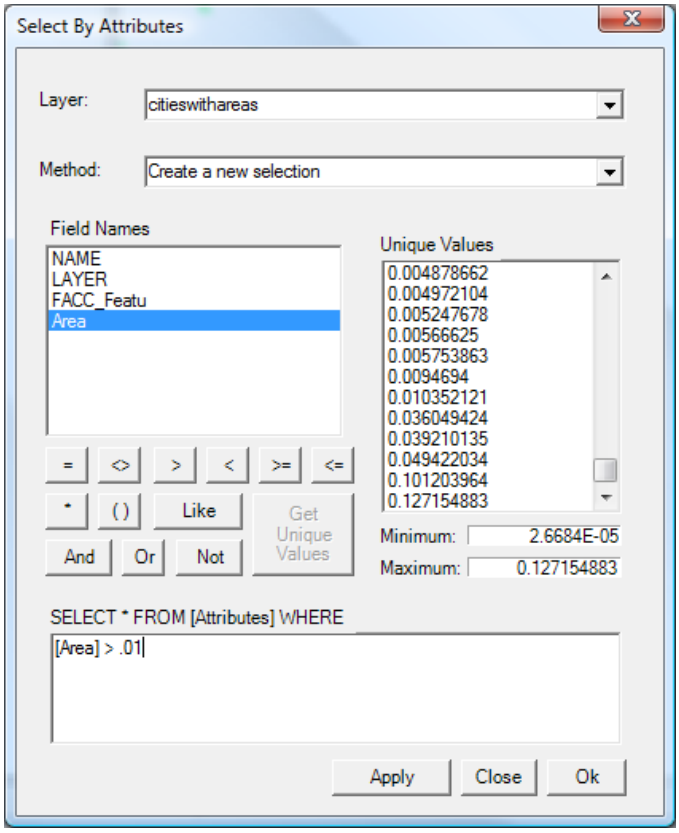


Figure 88: Area > .01

Figure 89 shows how to use the legend to create a new layer from the features that were selected in the previous step.

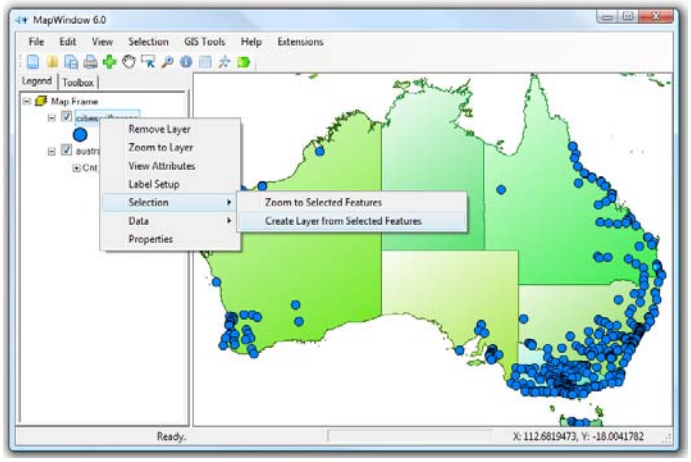


Figure 89: Create Layer

By highlighting the *Selection* tab in the context menu, it reveals new options, including the ability to create a layer from the selected features. Creating a layer from

the selected features will create a new in-memory shapefile that is not yet associated with a true data layer.

7.2.5. Step 5: Export Layer to a File

This in-ram selection can be converted to a file on the disk by using the *Export Data* menu option, located under the *Data* context menu item in the Legend. The Export Data menu option is illustrated in Figure 90.

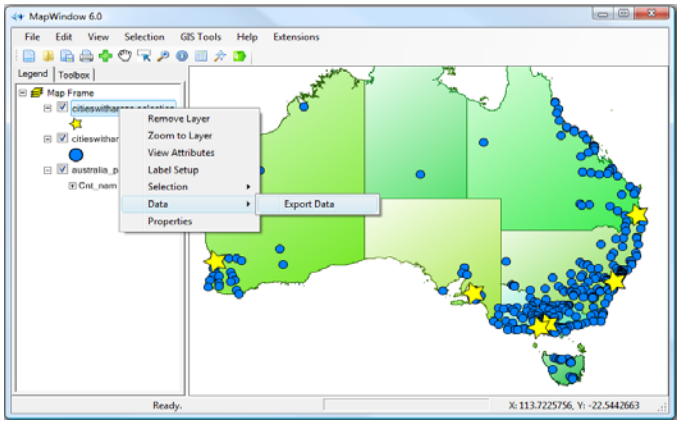


Figure 90: Export Layer

7.2.6. Step 6: Repeat with Roads Layer

The roads shapefile is quite large. In order to improve the speed while working with these demonstration datasets, Figure 91 illustrates using the *Select by Attributes* functionality a second time in order to select the primary routes.

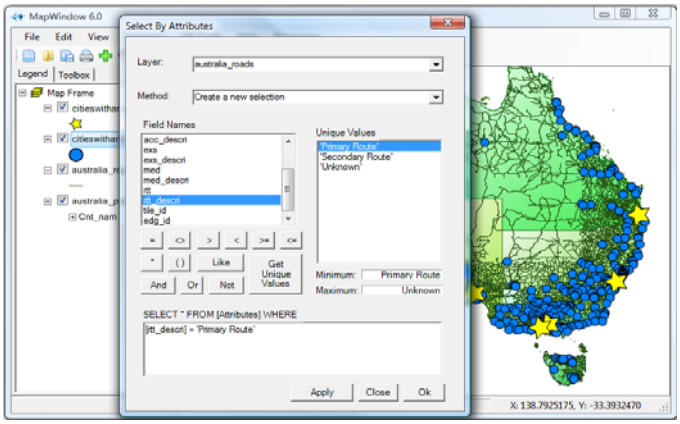


Figure 91: Primary Roads

7.2.7. Step 7: Select Political Bounds by Clicking

Figure 92 demonstrates a powerful additional tool that is built into the MapWindow tool strip that allows manually selecting shapes by clicking on them on the map, or by encompassing them in a rectangle drawn on the map.

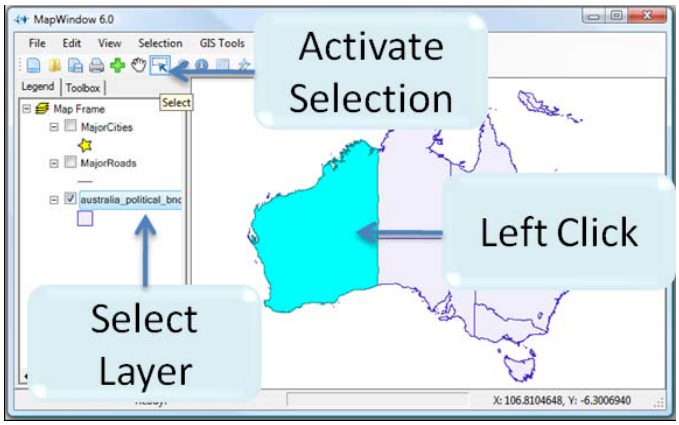


Figure 92: Select Major Areas

In order for the *selection* tool to work for a given layer, that layer should be selected in the Legend, or in a group that is selected in the Legend. Limiting selection in this way prevents content from other layers from being selected at the same time. Pressing the *selection* button shown in Figure 92 switches the map into selection mode. Holding down the [Shift] key allows multiple layers to be selected. This demonstration isolates the major polygons by selecting them manually, which eliminates all of the small islands around the perimeter of Australia. After the shapes are selected, a new layer can be created from the *create layer from selection* option in the Legend, and the data can be exported to create a simplified continental shapefile. Labels created from this simplified shapefile will not feature as many duplicates. Figure 93 shows the *Label Setup* context menu item activated by right clicking the layer in the Legend.

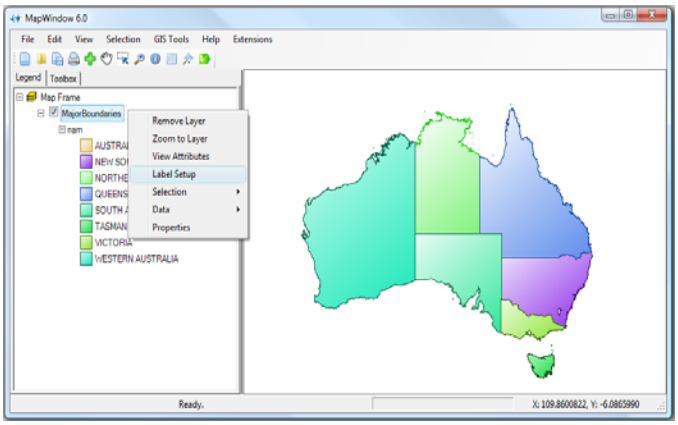


Figure 93: Activate Labeling

Clicking on this option will open the *Feature Labeler* dialog shown in Figure 94.

7.2.8. Step 8: Apply Labeling

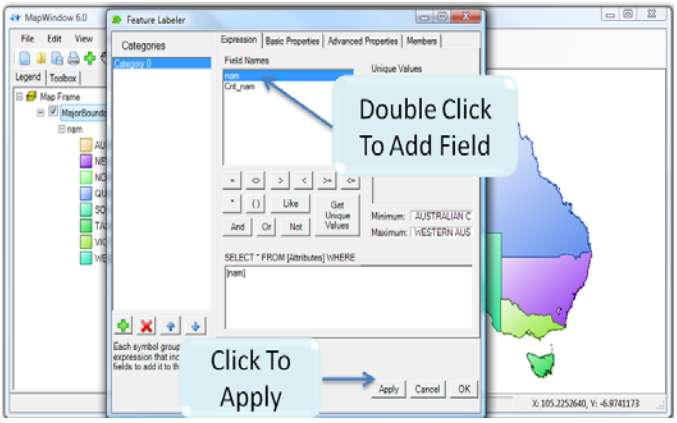


Figure 94: Apply Labels

Figure 95 shows the labels after they have been added. The labeling dialog also allows the user to specify alternate backgrounds, fonts, colors and halos for labels.

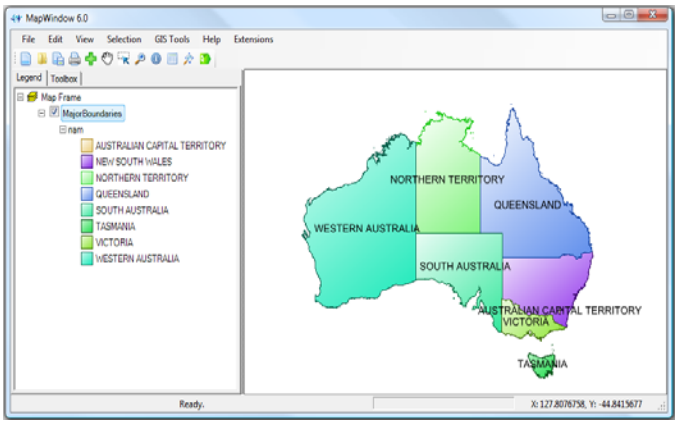


Figure 95: Applied Labels

7.3. Programmatic Point Symbology

The following demonstrations in this chapter can be performed independently, but will require the modified datasets created in section 5.2. The remaining demonstrations are subdivided into 5 basic categories: *Points, Lines, Polygons, Labels*, and *Rasters*. A comprehensive set of cascading forms launched from the Legend provide a built in system so that users can get started editing symbology right away using the built in components. However, this section is not about mastering the buttons on the dialogs. Rather, this section makes the assumption that the reader is either writing a plug-in, or else is developing a custom GIS program using the MapWindow .NET components. This set of instructions shows the source code necessary to control these aspects programmatically, instead of by using a pre-constructed user interface.

In previous versions of MapWindow, setting the color of the seventh point in the shapefile to red was easy to do, but the trade-off was that anything more complicated was not directly supported. Instead, developers would have to write their own code to cycle through each feature and symbolize it according to its attributes. MapWindow 6 introduces thematic symbol classes that allow programmatic creation of color schemes or size ranges. Methods and constructors have been provided that allow the creation of

simple schemes with a single line of code, but the sets of categories and symbol layers provide built in mechanisms to draw much more complex representations than in previous versions of MapWindow.

The datasets created as part of exercise 2 are used for the following demonstrations. These demonstrations are created with a copy of the visual studio project that was created in exercise 1 to show that the same source code will work even if the active project is a new GIS application but using the same components. The first step is to explore adding data to the map programmatically.

7.3.1. Add a Point Layer

Figure 96 shows the C# code that will open a new point layer and add that layer to the map.

```
FeatureSet fs = new FeatureSet();
fs.Open(@"[YourFolder]\Ex3\Data\CitiesWithAreas.shp");
IMapFeatureLayer myLayer = map1.Layers.Add(fs);
```

Figure 96: C# Code - Add Point Layer

The three lines of code above are all that is needed to programmatically add the *cities with areas* shapefile to the map. The first line creates an external *FeatureSet* class. A FeatureSet class is useful for opening and working with shapefiles. The second line reads the content from the vector portion of the shapefile into memory. The @ symbol tells C# that the line should be read literally, and will not use the \ character as an escape sequence. The *[YourFolder]* placeholder in the second line should be replaced by the folder containing these exercises. Finally, the last line adds the data to the map and returns a layer handle that can be used to control the symbology. The lines of code above can be incorporated into the project by overriding the *OnShown* method. That way, when the form is shown for the first time, it launches the map. The *FeatureSet* variable

provides access to all the information stored directly in the shapefile, but organized into feature classes. Figure 97 shows the class diagram of the feature classes that control data access.

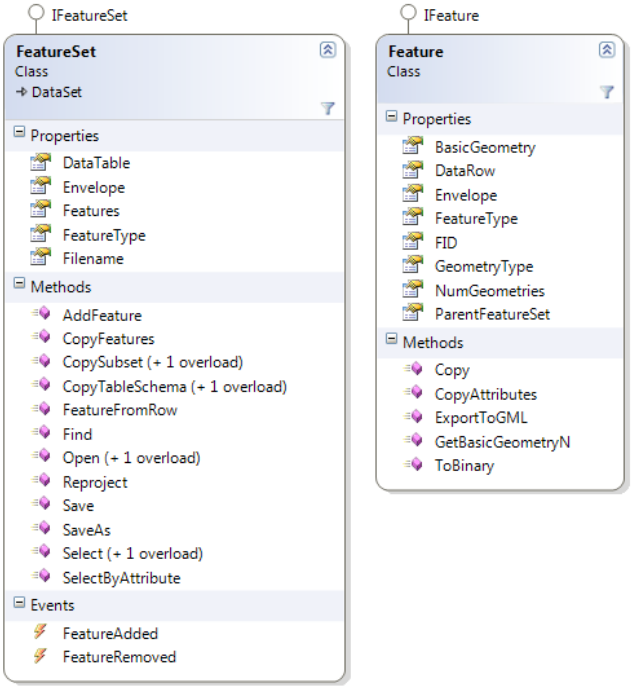


Figure 97: Feature Set and Features

The simplified class diagrams in Figure 97 give an idea of what kinds of information can be found directly on the *FeatureSet* class, or in one of the individual features. The *DataTable* property returns a standard .NET *System.Data.DataTable*, filled with all of the attribute information. This table is also used by the *SelectByAttribute* expression. The *Envelope* is the geographic bounding box for the entire set of features. The *Features* property is the list of features themselves. This list is enumerable, so developers can cycle through the list and inspect each of the members. The *FeatureType* simply tells whether or not the *FeatureSet* contains points, lines or polygons.

The other significant vector data class shown in Figure 97 is the *Feature*. The *BasicGeometry* class lists all of the vertices, organized according to OGC geometric

structures, such as *Points*, *LineStrings*, *Polygons*, and *MultiGeometries* of the various types. The basic geometric interfaces only focus on data access so that extensions that support different data formats do not also have to re-write topological intersection methods. For instance, the mathematical overlay and relate operations that are specified as part of the simple feature specification are not required on the basic interfaces. Instead, the role of the *IBasicGeometry* interface is designed to provide the interface for data access that is independent from topological calculations.

In order to make using geometric features easier, extension methods were added separately that allow developers that are using features to access methods as though they were part of the feature class itself. From a programmatic viewpoint, the methods for intersection or other overlay operations appear in the development tools almost as though those methods were built directly into that class. Figure 98 shows the extension method as it appears in the Intellisense tool in Visual Studio.

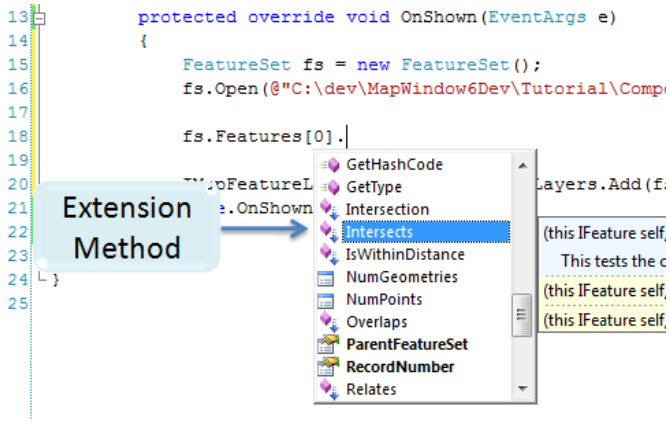


Figure 98: Extension Method

Typing a period after a class in .NET will display an automatic list of options. Continuing to type normally filters this list of options, so that developers can very rapidly narrow the displayed items and reduce the chance for spelling mistakes. It also gives them an instant browse window to explore the methods, properties, events and fields on a

particular class. This auto-completion tool is referred to as Microsoft Intellisense. The exact appearance of this function will depend on the version of visual studio, as well as whether or not the developer has added any extensions like Re-Sharper. If XML comments have been generated for the project, (which they have been for MapWindow 6.0) developers using the classes will not only see the methods, properties and events available, but will also see help for each of these methods in a box that pops out to the right of that window.

Methods are identified in this Intellisense list by having a purple box. Extension methods are represented by a purple box with a blue arrow to the right of that box. For extension methods, the code is separate from the class, and found in an external static method. Using extension methods, it becomes easy to associate a behavior like *Intersects* directly with the feature, but without every external data provider having to rewrite the intersection code in order to satisfy the *IFeature* interface.

7.3.2. Simple Symbols

Because this specific example dataset is known in advance to be a point dataset, the symbolization can work directly with the *PointSymbolizer* class without writing tests to determine the feature type. Though the architecture of a *PointSymbolizer* is complex in order to support symbols with multiple layers of an open ended symbol specification and different categories, simplified constructors have been provided to automatically create the complex structures necessary for the simple scenarios. The constructors are effectively shortcuts that hide the complexities necessary to support a rich, open-ended symbology structure. Starting with the *myLayer* class from the code in Figure 96, the symbolizer can be specified through the *myLayer.Symbolizer* class. Figure 99 shows the

C# code that uses parameters in a constructor in order to create a yellow star symbol for all the points.

```
private void MakeYellowStars(IMapFeatureLayer myLayer)
{
   myLayer.Symbolizer = new PointSymbolizer(Color.Yellow, PointShapes.Star, 16);
}
```

Figure 99: C# Code - Yellow Star Symbolizer

Figure 100 shows what the yellow star symbol looks like for the Australia cities.

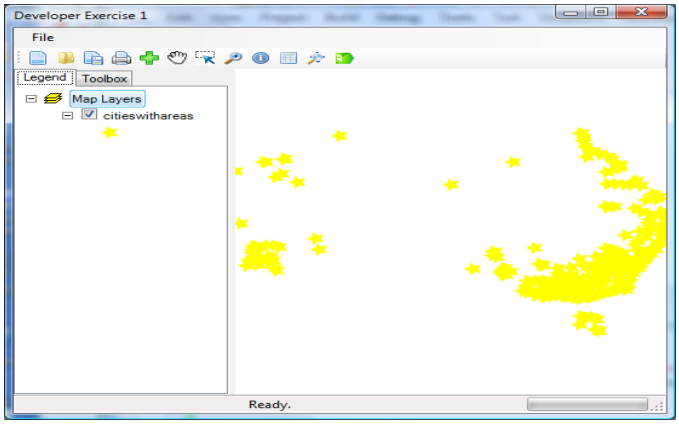


Figure 100: Yellow Star

The borders of the stars in Figure 100 are hard to see because the stars are a single color and that color is a light color simlar to the white background. A second method, called *SetOutline*, provides a generalized method that is used to give the stars black outlines. Figure 101 shows the C# code that will change the outline color of the stars to black.

```
myLayer.Symbolizer = new PointSymbolizer(Color.Yellow, PointShapes.Star, 16);
myLayer.Symbolizer.SetOutline(Color.Black, 1);
```

Figure 101: C# Code - Black Outline

Figure 102 shows the outlined stars as they appear on the map of Australia cities.

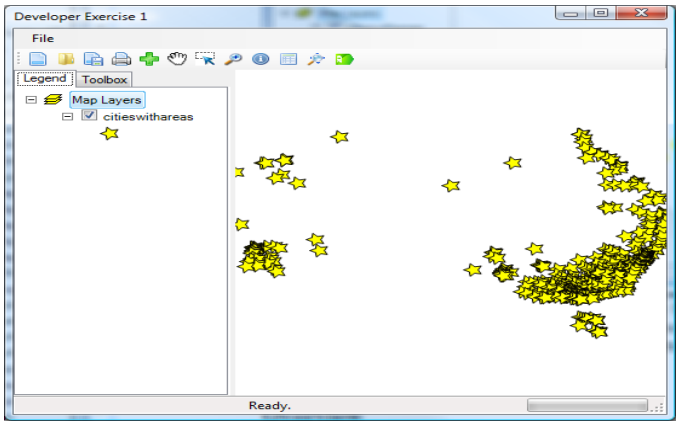


Figure 102: Yellow Stars with Outlines

The layer does not support methods or properties to control these characteristics directly. Instead, it uses a class called a *symbolizer*. Symbolizers contain all of the descriptive characteristics necessary to the symbolic content, but none of the vector information about where that content should be drawn. In this simplest situation, there are not multiple categories, complex symbols that have multiple layers, or extended symbol types that are not part of the original library. A simplified method like *SetOutline* may or may not work as expected in every case, since some types of symbols do not support outlines. However, the method is provided because it allows for the difficulty in accomplishing programming tasks to escalate with complexity, rather than forcing developers to use the full complexity in order to control something much simpler. Even this simplified set is so rich that it already supports the basic symbology options that were provided in previous versions of MapWindow.

7.3.3. Character Symbols

In addition to the basic symbols, MapWindow 6.0 also provides access to using characters as a symbol. Using characters is a very powerful symbolic system since character glyphs are vectors, and therefore scalable. They do not appear jagged or pixilated even when printing to a large region at high resolution. This system is also

incredibly versatile. Not only can developers use pre-existing symbol fonts (like *wingdings*) that are on their computer, there are open source fonts that provide GIS symbols. One helpful site that has numerous GIS symbol fonts that can be downloaded for free is found here: http://www.mapsymbols.com/symbols2.html. For this exercise, the *military true type* fonts from the site are used. Downloading and unzipping the file produces a file with the extension .ttf, which is a *true type* font. The next step is to find the Fonts option in the windows control panel. Figure 103 shows the fonts folder in the control panel for Windows Vista. This will appear differently on other versions of windows.

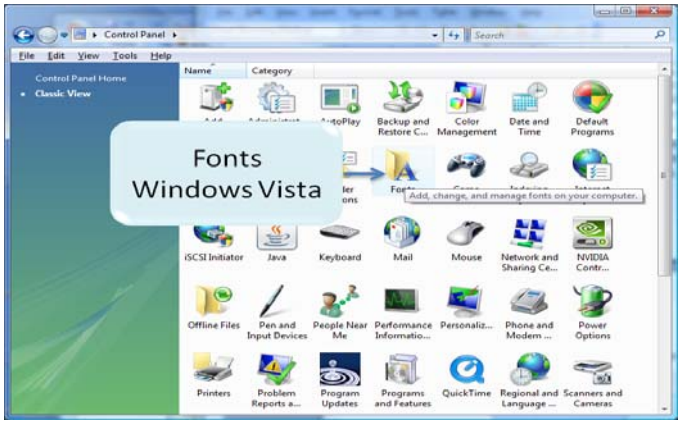


Figure 103: Fonts in Control Panel

Clicking on this folder opens the folder showing all of the currently installed *true type* fonts. Right click anywhere in this folder that is not directly on one of the existing fonts, and the context menu shown in Figure 104 is exposed. The "Install New Font" option will allow Vista users to install a new font, and they will have to browse to the recently downloaded and unzipped Military.ttf file.

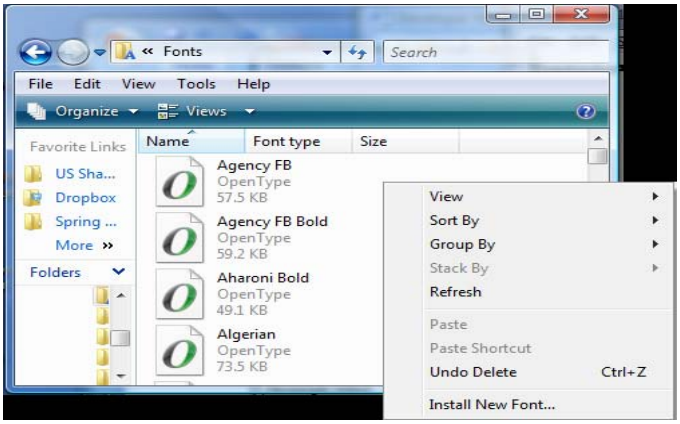


Figure 104: Right Click to Install New Font

Before writing any code, it might be useful to verify that the new font is available. MapWindow 6.0 has symbolic dialogs that show all the available font types, and so an easy method is to simply load some point data and attempt to change the symbol to a character symbol with the new font family. Double clicking next to the layer in the Legend control will open the Point Symbolizer Dialog. There is a Combo-box named "Symbol Type" which enables the user to choose a new symbol type. In this case, *character* should be chosen. Figure 105 shows the drop down in the point symbolizer dialog that allows the user to change the symbol type.

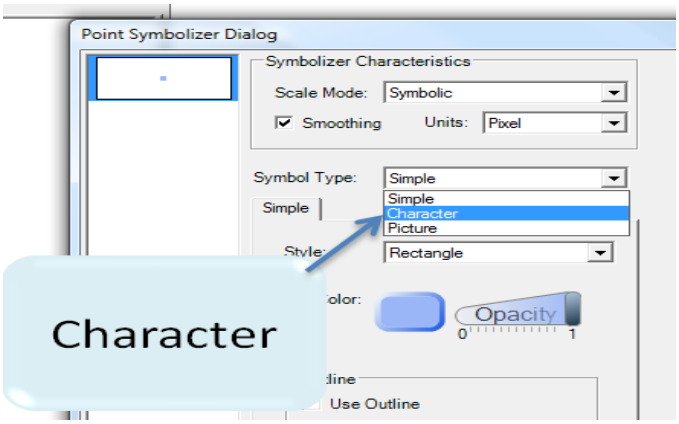


Figure 105: Switch to Characters

Figure 106 shows the font family dropdown that shows the different available font families, allowing the user to pick any font family including symbolic fonts like *Military*.

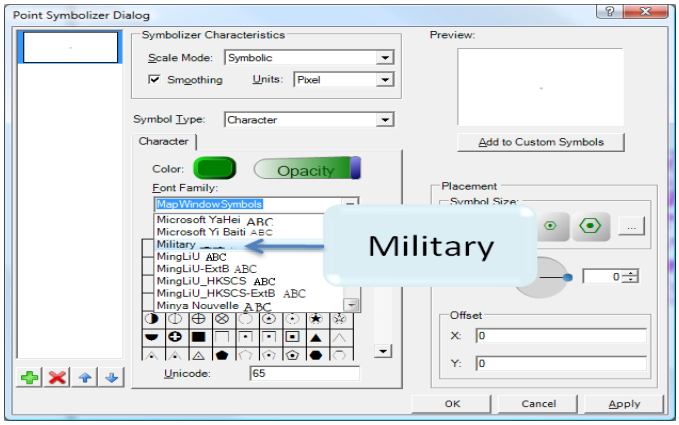


Figure 106: Choose Military

Once the *military* font family is selected, the icons listed in the character selection control are replaced with the new *military* symbols. Because many GIS systems support the use of fonts for their point symbols, it is possible for MapWindow to show font types from pre-created professional font sets. Having verified that the font type is visible to MapWindow, the next step is to use one of the symbols from the *military* font family programmatically. In this font set, the glyph used for the letter M looks like a plane. The constructor also supports specifying the color and size. Figure 107 shows the C# code that can be used to assign the military font programmatically.

```
myLayer.Symbolizer = new PointSymbolizer('M', "Military", Color.Blue, 16);
```

Figure 107: C# Code - Blue Character Symbols

Figure 108 shows what the military plane characters look like when shown on the map.

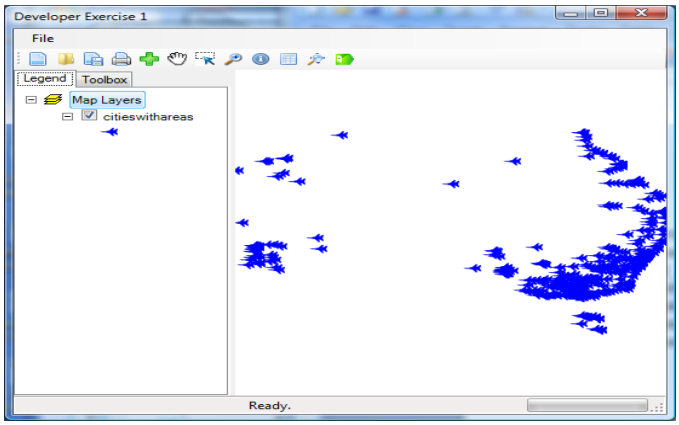


Figure 108: Military Plane Characters

As a side note, the *military* font family is slightly abnormal in that it has glyphs that are several times the width of the character in white space at the bottom of each glyph. Attempting to center the font vertically will cause problems because of all the whitespace. As a result, an added line of code catches this possibility and uses the width for centering instead. Since these symbols may not be exactly square, centering them may place the *military* symbols slightly off center. However, more professionally created symbols will be centered correctly since they will have a realistic height value that can be used for centering. It is also possible to create custom glyphs for use as point symbols using various font editor software packages. A commercial example is Font Creator, which has a 30 day free trial and can be purchased for about $100 at this site: http://www.high-logic.com/download.html. In the spirit of open source, however, there are also open source options available such as Font Forge: http://fontforge.sourceforge.net/ and Font Editor: http://fonteditor.org/ both of which are completely free.

7.3.4. Image Symbols

Sometimes, it simply is not possible to work with fonts, however, and an image is preferable. The .Net Image classes allow the use of the most commonly used image

formats, including bitmap, jpeg, gif, png and tif. These can be specified from a file, or drawn using the GDI+ graphics objects in code. Larger symbols will cause the drawing of large numbers of points to take more time. The wiki-media tiger icon is an example of an image, and is used in this demonstration:

http://upload.wikimedia.org/wikipedia/commons/thumb/f/f5/Tiger_Icon.svg/48px-Tiger_Icon.svg.png

Once the file is downloaded and saved, developers can use the *Image.FromFile* method to load the file into an image class, or else they can embed the file as a resource to be used. The embedded resource method is illustrated here. If the project does not already contain a resource file, a new one can be created from solution explorer. In this case a resource file named *Images* is created. Figure 109 shows the context menu options on the solution explorer window that allows the addition of a new item.

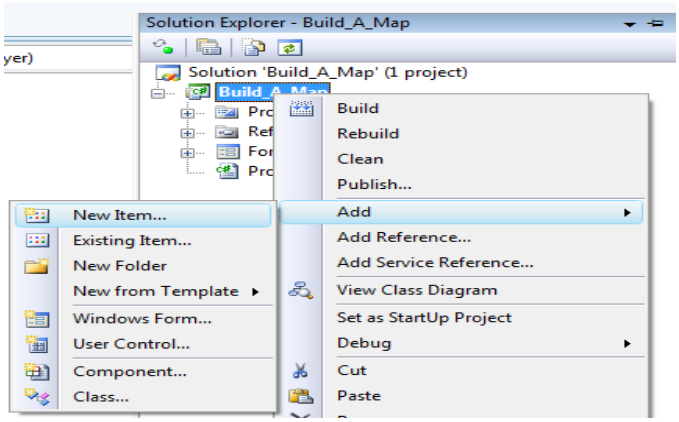


Figure 109: Add a New Item

Figure 110 shows the naming of the images resource file in the dialog.

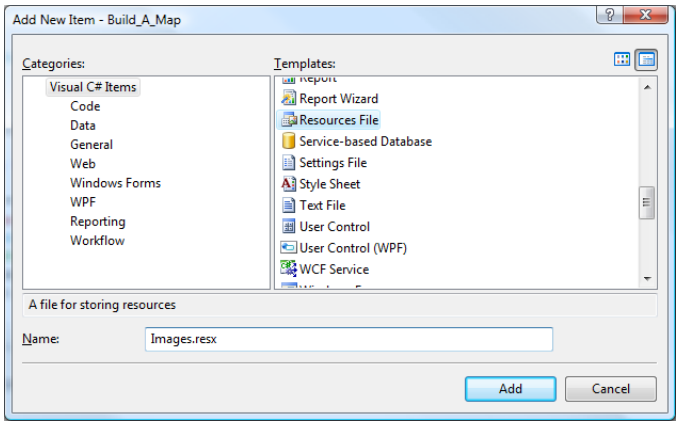


Figure 110: Images.resx Resource

To add a resource file, the *Resources File* selection should be chosen from the available templates. In this case the resource file is named *Images* and the *Add* button is clicked in order to finish the creation process. Double clicking the newly created *Images.resx* entry in the solution explorer window will open the resource file for editing. Under the *Add Resource* option in the toolbar just below the *Images.resx* tab, there is a menu item named *Add Existing File…* Figure 111 shows the context menu item for adding an existing file to the resource file.

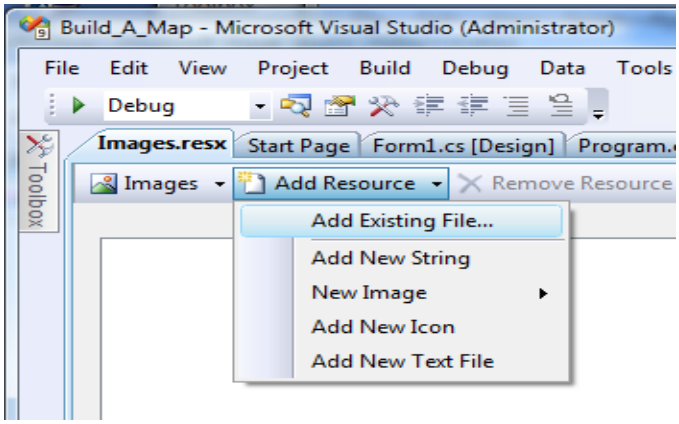


Figure 111: Add an Existing File

When the menu item is chosen, a dialog opens. This dialog allows developers to browse to the image that they want to add to the resource file. In this example, the newly

downloaded file is added to the resource file. To make it easier to find, the embedded image is renamed "Tiger." Figure 112 shows the tiger image after it has been added to the resource file and renamed.

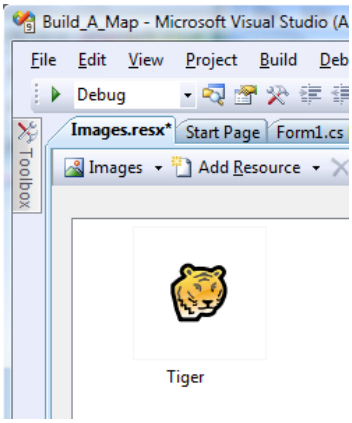


Figure 112: Rename to Tiger

Now, developers can programmatically reference this image any time using the *Images.Tiger* reference. It may be necessary to rebuild the project in order for Intellisense to show the new entry. Adding the image to a resource file sets up the framework for the code in Figure 113 where the Tiger image is used for the point symbology:

```
myLayer.Symbolizer = new PointSymbolizer(Images.Tiger, 48);
```

Figure 113: C# Code - Image Point Symbol

Figure 114 shows the resulting tiger image as it appears on the map.

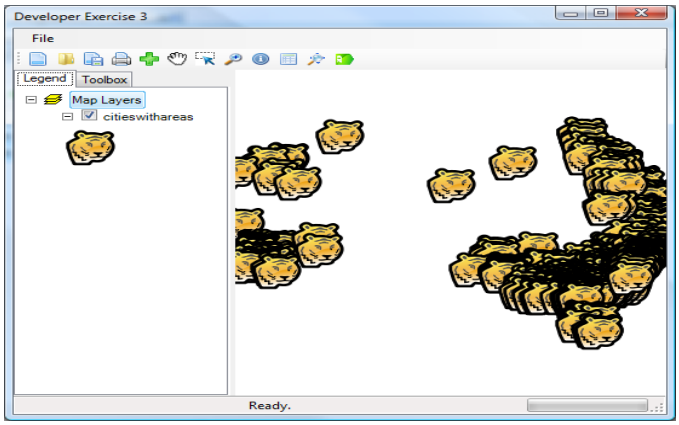


Figure 114: Tiger Images

7.3.5. Point Categories

This demonstration symbolizes the features based on values in the *Area* field. The areas were calculated from the polygons in exercise 2. The table in Figure 115 shows the attributes after the rows are sorted from largest to smallest using the *Area* field. A reasonable cutoff for picking the largest cities might be .01 square decimal degrees.

NAME	LAYER	FACC_Featu	Area
MELBOURNE	Built-Up Area	AL020	0.127154883
SYDNEY	Built-Up Area	AL020	0.101203964
PERTH	Built-Up Area	AL020	0.049422034
ADELAIDE	Built-Up Area	AL020	0.039210135
BRISBANE	Built-Up Area	AL020	0.036049424
GEELONG	Built-Up Area	AL020	0.010352121
NEWCASTLE	Built-Up Area	AL020	0.0094694
CRONULLA	Built-Up Area	AL020	0.005753863

Figure 115: Large Area Cities

The source code in Figure 116 has several parts to it. Firstly, casting the layer to a *MapPointLayer* reveals properties that are specific to points, and the null test will exit the method early if the data are not points. Two separate categories are created using filter expressions to separate what is drawn by each category. Finally, the new scheme is set as the layer's symbology. When the map is drawn, it will automatically show the scheme types in the Legend using whatever is specified in code as the legend text. Figure

116 shows the C# code for creating two categories, as well as setting the filter expression and legend text for those categories.

```
IMapPointLayer myPointLayer = myLayer as IMapPointLayer;
if(myPointLayer == null) return;
PointScheme myScheme = new PointScheme();
myScheme.Categories.Clear();

PointCategory smallSize = new PointCategory(Color.Blue, PointShapes.Rectangle, 4);
smallSize.FilterExpression = "[Area] < .01";
smallSize.LegendText = "Small Cities";
myScheme.AddCategory(smallSize);

PointCategory largeSize = new PointCategory(Color.Yellow, PointShapes.Star, 16);
largeSize.FilterExpression = "[Area] >= .01";
largeSize.LegendText = "Large Cities";
largeSize.Symbolizer.SetOutline(Color.Black, 1);
myScheme.AddCategory(largeSize);
myPointLayer.Symbology = myScheme;
```

Figure 116: C# Code - Multiple Category Symbols

Figure 117 shows the two categories of points when drawn on the map.

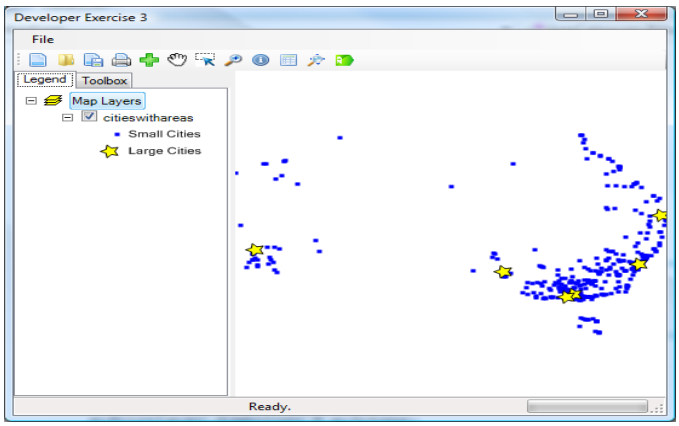


Figure 117: City Categories by Area

The square brackets in the filter expression are optional, but recommended to help clarify *field names* in the expression. Developers do not have to write code to loop through all the city shapes, test the area attribute programmatically, and then assign a symbol. Instead, they can simply allow the built in expression parsing to take over and handle the drawing. The built in settings allow programmers to work with the objects in a way that directly mimics how users work with the symbology controls on the graphical

user interface (GUI). Figure 118 shows the C# code for classifying the points into categories so that about the same number of shapes fall into each category.

```
IMapPointLayer myPointLayer = myLayer as IMapPointLayer;
if (myPointLayer == null) return;
PointScheme myScheme = new PointScheme();
myScheme.Categories.Clear();
myScheme.EditorSettings.ClassificationType = ClassificationTypes.Quantities;
myScheme.EditorSettings.IntervalMethod = IntervalMethods.Quantile;
myScheme.EditorSettings.IntervalSnapMethod = IntervalSnapMethods.Rounding;
myScheme.EditorSettings.IntervalRoundingDigits = 5;
myScheme.EditorSettings.TemplateSymbolizer =
        new PointSymbolizer(Color.Yellow, PointShapes.Star, 16);
myScheme.EditorSettings.FieldName = "Area";
myScheme.CreateCategories(myLayer.DataSet.DataTable);
myPointLayer.Symbology = myScheme;
```

Figure 118: C# Code - Quantile Classification

Figure 119 uses different colors to represent the separate quantile classifications.

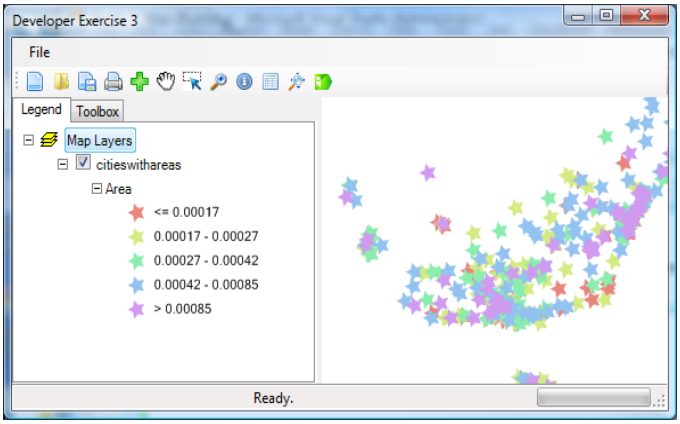


Figure 119: Quantile Area Categories

7.3.6. Compound Point symbols

Figure 120 shows the C# code that would create a single symbolizer that uses two, overlapping symbols.

```
PointSymbolizer myPointSymbolizer =
       new PointSymbolizer(Color.Blue, PointShapes.Ellipse, 16);
myPointSymbolizer.Symbols.Add(
       new SimpleSymbol(Color.Yellow, PointShapes.Star, 10));
myLayer.Symbolizer = myPointSymbolizer;
```

Figure 120: C# Code - Multi-Layer Point Symbols

Figure 121 shows what blue circles with overlapping yellow stars looks like as a point symbol on the map.

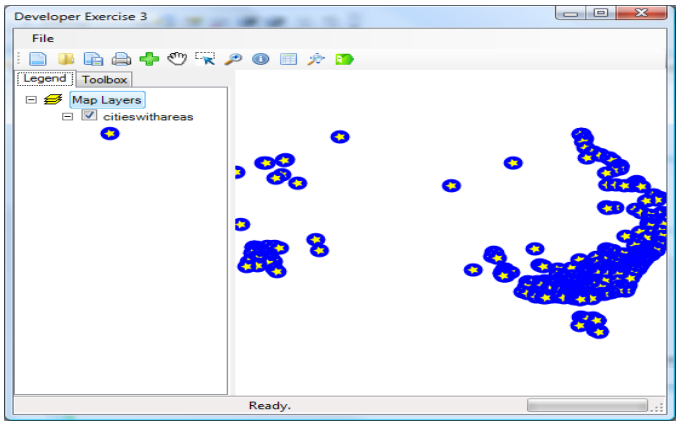


Figure 121: Blue Circles with Yellow Stars

7.4.Programmatic Line Symbology

7.4.1. Adding Line Layers

Line layers operate according to the same rules as points for the most part, except that instead of individual symbols, they can have individual strokes. The default symbology is to have a single line layer of a random color that is one pixel wide. Figure 122 shows the C# code that adds a new line layer onto the map.

```
FeatureSet fs = new FeatureSet();
fs.Open(@"[Your Folder]\Ex3\Data\MajorRoads.shp");
IMapFeatureLayer myLayer = map1.Layers.Add(fs);
```

Figure 122: C# Code - Add Line Layer

Figure 123 shows the default appearance of the line layer after it has been added to the map. Since the symbology is not yet specified, the color is random and the line width is 1 pixel.

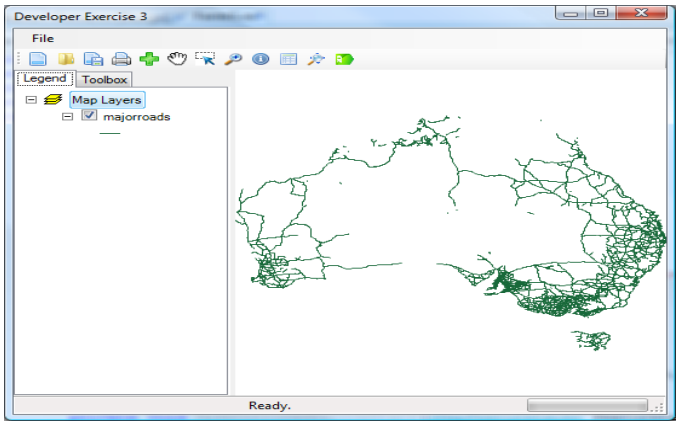


Figure 123: Add Line Layer

7.4.2. Simple Line symbols

Figure 124 shows the C# code for creating a line symbolizer that changes the color of the roads to brown.

```
private void BrownRoads(IMapFeatureLayer myLayer)
{
        myLayer.Symbolizer = new LineSymbolizer(Color.Brown, 1);
}
```

Figure 124: C# Code - Color Roads Brown

Figure 125 shows the same road lines, but with the new brown color specified by the symbolizer.

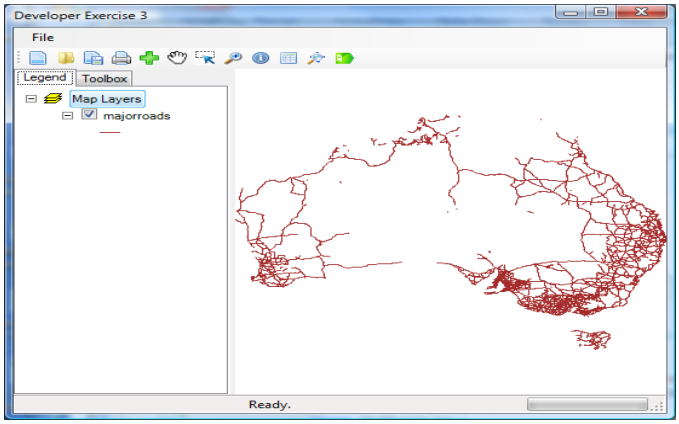


Figure 125: Brown Lines

7.4.3. Outlined Symbols

The line symbology is similar to the point symbology in that it also shares certain shortcut methods like *SetOutline*. The distinction is that unlike the simple symbol,

strokes cannot come pre-equipped with an outline. Instead, the appearance of an outline is created by making two passes with two separate strokes. The first stroke is wider, and black. The second stroke is narrower and yellow. The result is a set of lines that appear to be connected. In order to get a clean look at the intersections, all the black lines are drawn first. Then, all the yellow lines are drawn. This way, the intersections appear to have continuous paths of yellow, rather than every individual shape being terminated by a curving black outline. Figure 126 shows the C# code for setting the outline of the roads to black, while providing a wider, yellow interior.

```
LineSymbolizer road = new LineSymbolizer(Color.Yellow, 5);
road.SetOutline(Color.Black, 1);
myLayer.Symbolizer = road;
```

Figure 126: C# Code - Black Outlined Lines

Figure 127 shows the outlined roads with the yellow fill color on the map.

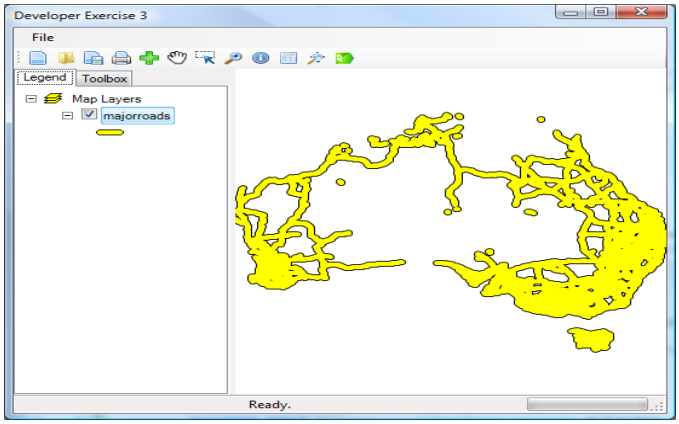


Figure 127: Yellow Roads with Outlines

7.4.4. Unique Values

One of the more useful abilities is to be able to programmatically apply symbology by unique values, without having to worry about what those values are or negotiate the actual color in each case. Simply specify a classification type of *UniqueValues* and a classification field, and MapWindow does the rest. In this case, the

default editor settings will create a hue ramp with a saturation and lightness in the range from .7 to .8. The editor settings can be used to control the acceptable range using the start and end color. There is a Boolean property called *HueSatLight*. If *HueSatLight* is true, then the color ramp is created by adjusting the hue, saturation and lightness between the start and end colors. If *HueSatLight* is false, then the red, blue and green values are ramped instead. In both cases, alpha (transparency) is ramped the same way. Figure 128 shows the C# code for creating a scheme that uses unique values for the tile_id field to create random categories.

```
LineScheme myScheme = new LineScheme();
myScheme.EditorSettings.ClassificationType =
ClassificationTypes.UniqueValues;
myScheme.EditorSettings.FieldName = "tile_id";
myScheme.CreateCategories(myLayer.DataSet.DataTable);
myLayer.Symbology = myScheme;
```

Figure 128: C# Code - Unique Value Classification

Figure 129 shows the roads colored based on their tile_id field drawn on the map.

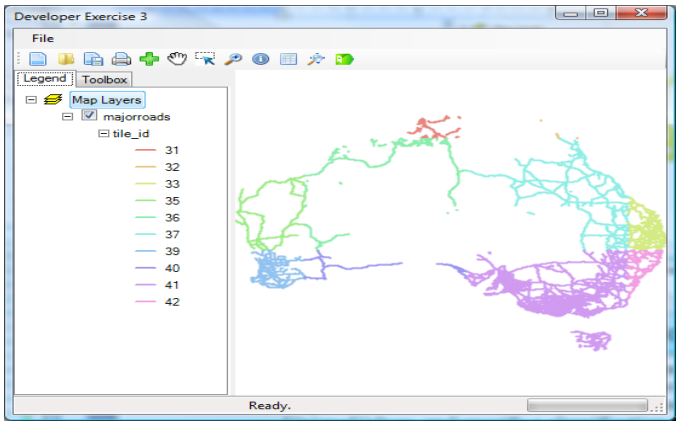


Figure 129: Roads with Unique Values

7.4.5. Custom Categories

In the previous example, the Legend shows a collapsible field name in order to clarify the meaning of the values appearing for each category. Revealing a named scheme in the legend can also be accomplished manually by controlling the *AppearsInLegend* property on the scheme object. If *AppearsInLegend* is false, the

categories will appear directly below the layer. When *AppearsInLegend* is true, the text in the Legend can be controlled using the scheme itself. Figure 130 shows the C# code that will create two separate categories and add them to a scheme.

```
LineScheme myScheme = new LineScheme();
myScheme.Categories.Clear();
LineCategory low = new LineCategory(Color.Blue, 2);
low.FilterExpression = "[tile_id] < 36";
low.LegendText = "Low";
LineCategory high = new LineCategory(Color.Red, Color.Black, 6,
DashStyle.Solid, LineCap.Triangle);
high.FilterExpression = "[tile_id] >= 36";
high.LegendText = "High";
myScheme.AppearsInLegend = true;
myScheme.LegendText = "Tile ID";
myScheme.Categories.Add(low);
myScheme.Categories.Add(high);
myLayer.Symbology = myScheme;
```

Figure 130: C# Code - Multiple Line Categories

Figure 131 shows the custom line categories as they appear on the map.

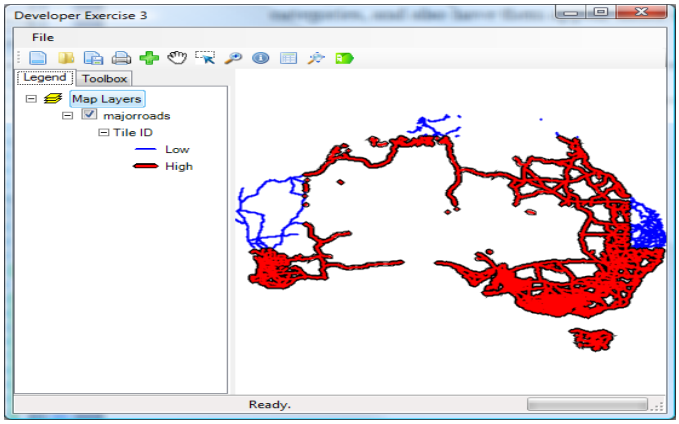


Figure 131: Custom Line Categories

7.4.6. Compound Lines

This example takes advantage of several powerful symbology options. Cartographic strokes allow the creation of two different types of line styles for this example. In the first case, brown railroad ties are created. Using the standard *dot* option for a simple cartographic line does not work because the dots created are proportional to the line width. However, with a custom dash pattern, it is possible to set the lines so that the dashes are thinner than the line width itself. The two numbers used in the dash

pattern do not represent offsets, but rather the lengths of the dash and non-dash elements that alternate. This notation is convenient since, for repeating ties, the pattern only needs to specify two numbers in order to describe a fairly complicated pattern.

The second layer of the symbol is a pattern of dark gray rails. In this case, the dash pattern is continuous and does not need to be changed. However, the rails are not continuous across the width of the line the way the ties are. Instead, two thin lines that appear along the path width form a double line. To do this, the *CartographicStroke* takes advantage of a *CompoundArray* property. With the compound array, the actual offsets for the start and end positions along the array are specified, where 0 is the left of the line and 1 is the right. In some cases, lines that are two thin may not get drawn at all, so it is advisable to ensure that the width of the lines represented in the *CompoundArray* work out to be just slightly larger than 1 to ensure that the lines get drawn.

In the code below, the start and end caps are also specified. By default these are set to round, which will end up producing gray circles at each of the intersections. By specifying that the end caps should be flat, no extension will be added to ends of the lines. Rounded caps look the best for solid lines because it creates a rounded, softer look to roads that are wider than one pixel. Figure 132 shows the multiple layered lines being created using C# code.

```
LineSymbolizer mySymbolizer = new LineSymbolizer();
mySymbolizer.Strokes.Clear();
CartographicStroke ties = new CartographicStroke(Color.Brown);
ties.DashPattern = new float[] {1/6f, 2/6f};
ties.Width = 6;
ties.EndCap = LineCap.Flat;
ties.StartCap = LineCap.Flat;
CartographicStroke rails = new
CartographicStroke(Color.DarkGray);
rails.CompoundArray = new float[]{.15f,.3f,.6f,.75f};
rails.Width = 6;
rails.EndCap = LineCap.Flat;
rails.StartCap = LineCap.Flat;
mySymbolizer.Strokes.Add(ties);
mySymbolizer.Strokes.Add(rails);
myLayer.Symbolizer = mySymbolizer;
```

Figure 132: C# Code - Multi-layer Lines

Figure 133 shows the multi-stroke railroad symbol as it appears on the map.

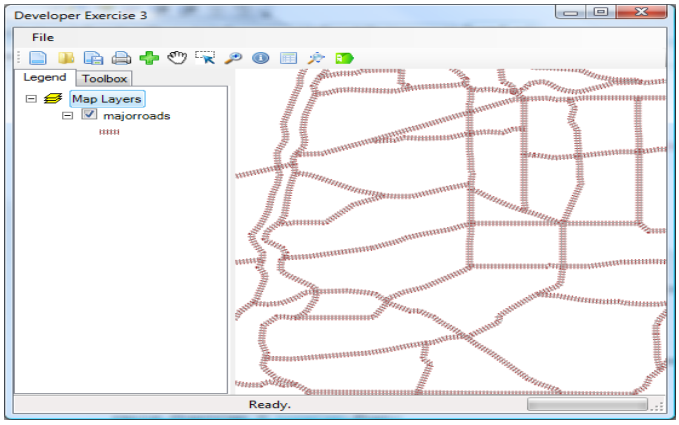


Figure 133: Multi-Stroke Railroads

7.4.7. Line Decorations

One of the new features with this generation of MapWindow is the ability to add point decorations to lines. Each decoration has one symbolizer and can operate with several positioning options. Each stroke can support multiple decorations, so there is a great deal of customizable patterns available. The decorations can also be given an offset so that the decoration can appear on one side of the line or another. In this case, yellow stars are added to a blue line. Figure 134 shows the C# code for adding point symbol decorations to the lines.

```
LineDecoration star = new LineDecoration();
star.Symbol = new PointSymbolizer(Color.Yellow,
PointShapes.Star, 16);
star.Symbol.SetOutline(Color.Black, 1);
star.NumSymbols = 1;
CartographicStroke blueStroke = new
CartographicStroke(Color.Blue);
blueStroke.Decorations.Add(star);
LineSymbolizer starLine = new LineSymbolizer();
starLine.Strokes.Clear();
starLine.Strokes.Add(blueStroke);
mvLaver.Symbolizer = starLine;
```

Figure 134: C# Code - Lines with Decorations

Figure 135 shows the major road lines with a star decoration at the ends of the lines.

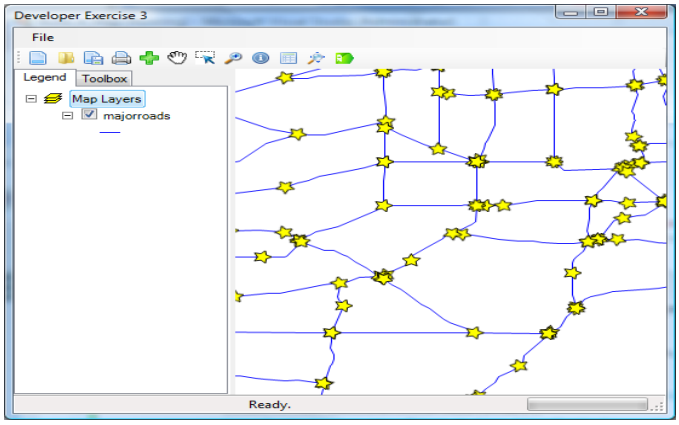


Figure 135: Lines with Star Decorations

7.5.Programmatic Polygon Symbology

7.5.1. Add Polygon Layers

Polygon layers are another representation of vector content where there is an area being surrounded by a boundary. Polygons can have any number of holes, which are represented as inner rings that should not be filled. However, in order to represent a shape like Australia, which has several islands, as a single shape, a *MultiPolygon* can be used instead. A *MultiPolygon* is still considered to be a geometry and will respond to all of the geometry methods, like *Intersects*. The polygon shapefile can be added the same way as the point or line shapefiles.

Polygon symbolizers are slightly different from the other two symbolizers because in the case of polygons, both the borders and the interior have to be described. Since the borders are basically just lines, rather than replicating all the symbology options as part of the polygon symbolizer directly, each polygon symbolizer references a line symbolizer in order to describe the borders. Using the line symbolizer for the borders is a similar strategy to re-using the PointSymbolizer in order to describe the decorations that can appear on lines. Figure 136 shows the C# code for adding a new polygon layer to the map.

```
FeatureSet fs = new FeatureSet();
fs.Open(@"[Your folder]\Ex3\Data\MajorBoundaries.shp");
IMapFeatureLayer myLayer = map1.Layers.Add(fs);
```

Figure 136: C# Code - Add Polygon Layer

Figure 137 shows the polygon layer for the major boundaries of Australia after it has been added to the map.

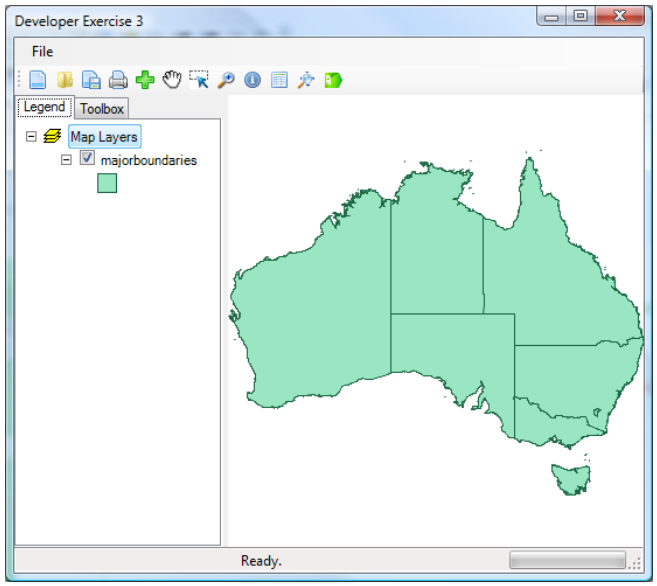


Figure 137: Add Major Boundaries

7.5.2. Simple Patterns

One important task when symbolizing polygons is to set the fill color for those polygons. Specifying only an interior fill creates a continuous appearance, since the normal boundaries are adjacent and all the same color. Figure 138 shows the C# code that will add the polygons to the map and control the fill color of all the polygons.

```
private void BluePolygons(IMapFeatureLayer myLayer)
{
  PolygonSymbolizer lightblue = new PolygonSymbolizer(Color.LightBlue);
  myLayer.Symbolizer = lightblue;
}
```

Figure 138: C# Code - Light Blue Polygons

Figure 139 shows the polygon layer with only a light blue fill color.

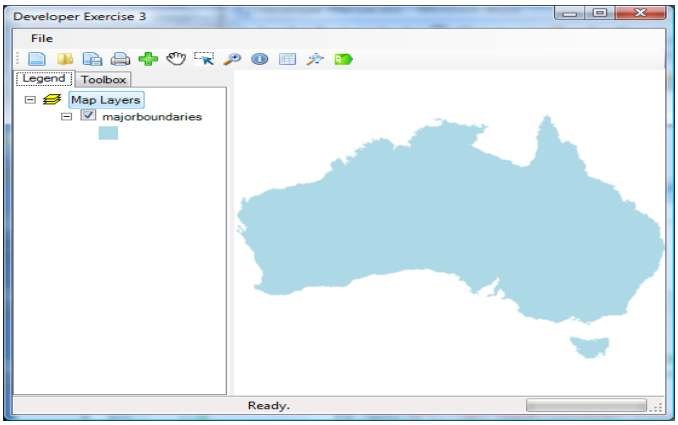


Figure 139: Blue Fill Only

Figure 140 shows the C# code for adding an additional blue outline.

```
PolygonSymbolizer lightblue = new PolygonSymbolizer(Color.LightBlue);
 lightblue.OutlineSymbolizer = new LineSymbolizer(Color.Blue, 1);
 myLayer.Symbolizer = lightblue;
```

Figure 140: C# Code - Polygon Outlines

Figure 141 shows the polygons with light blue fill coloring and dark blue borders.

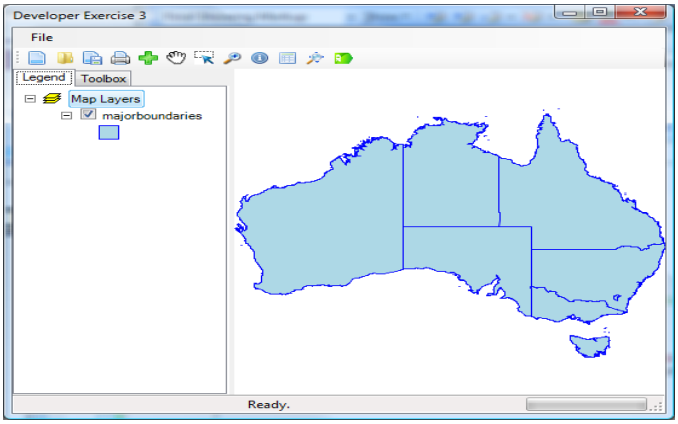


Figure 141: With Blue Border

7.5.3. Gradients

One of the more elegant symbology options is to apply a gradient. These gradients can vary in type from linear to circular to rectangular, with the default gradient being linear. The boundary of the gradient is controlled by the boundary of the members of the category. A single category that includes all of the shapes will combine a gradient so that it is continuous across all the shapes. Unlike the previous example where the

outline symbolizer was created through a shortcut method, this example takes advantage of the shared method *SetOutline,* which does the same thing. For points, this method controls the symbols themselves. For lines, this method adds a slightly larger stroke beneath the existing strokes. For polygons, this technique controls the line symbolizer that is used to draw the outline. The gradient angle is specified in degrees, moving counter-clockwise from the positive x axis. Figure 142 shows the C# code for creating a subtle linear gradient at a 45 degree angle that extends across all of the shapes.

```
PolygonSymbolizer blueGradient =
new PolygonSymbolizer(Color.LightSkyBlue, Color.DarkBlue, 45, GradientTypes.Linear);
blueGradient.SetOutline(Color.Yellow, 1);
myLayer.Symbolizer = blueGradient;
```

Figure 142: C# Code – Gradient Polygon Fill

Figure 143 shows the linear gradient as it appears for the polygon layer on the map.

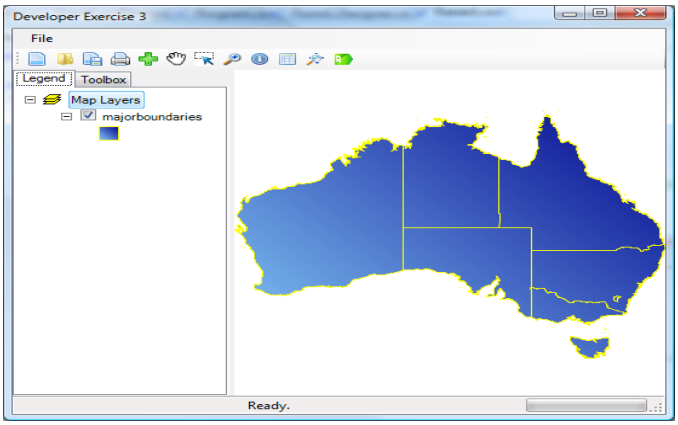
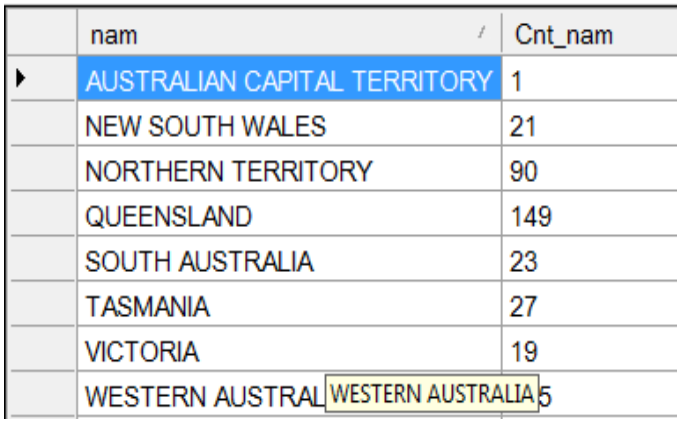


Figure 143: Continuous Blue Gradient

7.5.4. Individual Gradients

Another possible symbol strategy is to create the gradients so that they are shape specific. Creating lots of categories can cause the rendering process to be slower for large numbers of polygons because the drawing gets slower as the number of categories increases. Thousands of polygons can be drawn with one call by having only one

symbolic class that describes all the polygons. In the case of a few hundred classes, this distinction is not really noticeable. In this example, separate categories are created using the *nam* field, which is different for each of the major shapes that were selected as part of exercise 1 in order to create the basic polygon shapefile. Figure 144 shows some attribute fields for the polygon featureset, including the *nam* and *Cnt_nam* fields.



nam	Cnt_nam
AUSTRALIAN CAPITAL TERRITORY	1
NEW SOUTH WALES	21
NORTHERN TERRITORY	90
QUEENSLAND	149
SOUTH AUSTRALIA	23
TASMANIA	27
VICTORIA	19
WESTERN AUSTRAL	5

Figure 144: Major Boundaries Fields

Figure 145 shows the code for using a unique classification combined with the linear gradient options in order to show gradients that are drawn independently for each shape.

```
PolygonSymbolizer blueGradient =
new PolygonSymbolizer(Color.LightSkyBlue, Color.DarkBlue, -45, GradientTypes.Linear);
blueGradient.SetOutline(Color.Yellow, 1);
PolygonScheme myScheme = new PolygonScheme();
myScheme.EditorSettings.TemplateSymbolizer = blueGradient;
myScheme.EditorSettings.UseColorRange = false;
myScheme.EditorSettings.ClassificationType = ClassificationTypes.UniqueValues;
myScheme.EditorSettings.FieldName = "nam";
myScheme.CreateCategories(myLayer.DataSet.DataTable);
myLayer.Symbology = myScheme;
```

Figure 145: C# Code - Unique Value Polygon Classification

Figure 146 shows the individual gradients as they appear on the map.

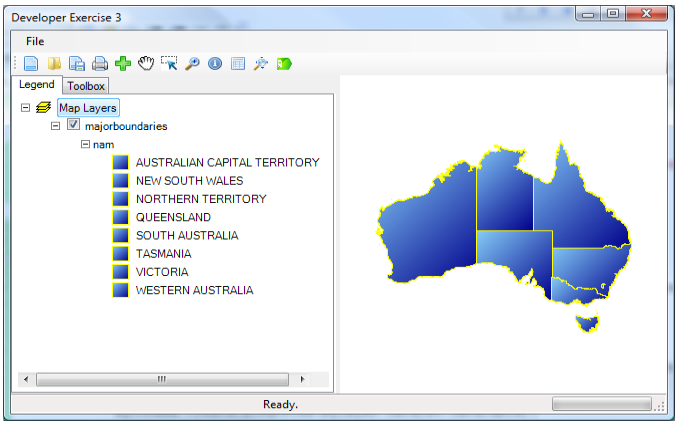


Figure 146: Individual Gradients

7.5.5. Multi-Colored Gradients

In the previous exercise the *UseGradient* property was set to false. The property does not prevent the template symbolizer from having a gradient. Instead, the *UseGradient* property allows the dynamically created categories to be given an automatic gradient, without having to build one in the template. The gradient symbol is calculated using a color from the color range, but the upper left is a little lighter and the lower right is a little darker. The tool gives an automatic subtle gradient that will be consistent across all the different colors created for each category. The default polygon symbolizer has a border that is the same hue, but slightly darker, which tends to create a visually appealing outline color. Figure 147 shows the code for using a range of colors for a classification scheme.

```
PolygonScheme myScheme = new PolygonScheme();
myScheme.EditorSettings.StartColor = Color.LightGreen;
myScheme.EditorSettings.EndColor = Color.LightBlue;
myScheme.EditorSettings.ClassificationType =
    ClassificationTypes.UniqueValues;
myScheme.EditorSettings.FieldName = "nam";
myScheme.EditorSettings.UseGradient = true;
myScheme.CreateCategories(myLayer.DataSet.DataTable);
myLayer.Symbology = myScheme;
```

Figure 147: C# Code - Polygon Color Ramp

Figure 148 shows the polygons with gradients that feature random colors from light green to light blue as they appear on the map.

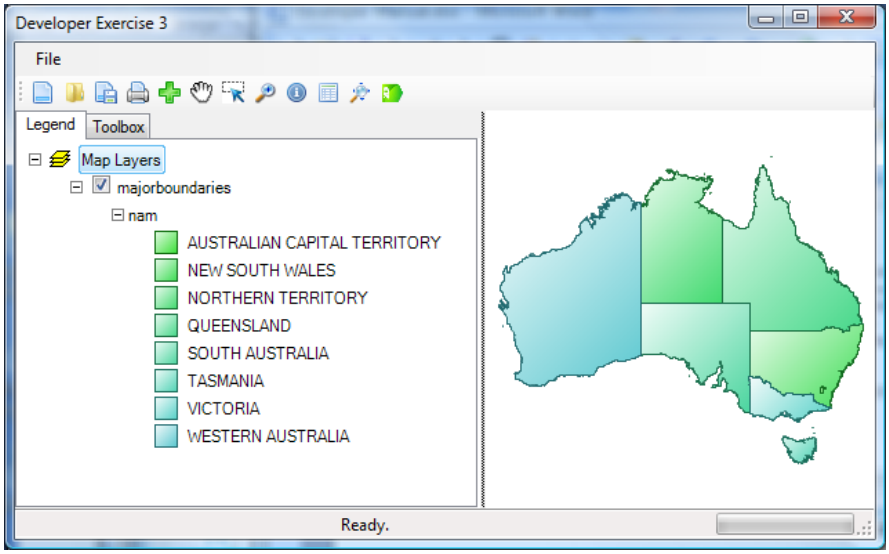


Figure 148: Unique Cool Colors with Gradient

7.5.6. Custom Polygon Categories

There is a distinction between the terms "Symbolizer" and "Symbology." The term "Symbology" is used to represent an entire scheme, which can have many categories. A "Symbolizer" controls the appearance of shapes within one category. By default, all the feature layers start with a scheme that has exactly one category, which has a symbolizer with exactly one drawing element (symbol, line or pattern). The *Symbolizer* property on a layer is a shortcut to the top-most category. If there are multiple custom categories, their appearance can be controlled or customized directly using the *Symbolizer* property on each category. Labels are added to the layer below in order to illustrate that the two pink shapes are in fact shapes that start with N. The actual labeling code will be illustrated under a separate section under labeling. Figure 149 shows the C# code for using custom categories to symbolize specific polygons, including identifying names that simply begin with N.

```
PolygonScheme scheme = new PolygonScheme();
PolygonCategory queensland = new PolygonCategory(Color.LightBlue,
Color.DarkBlue, 1);
queensland.FilterExpression = "[nam] = 'Queensland'";
queensland.LegendText = "Queensland";
PolygonCategory nWords = new PolygonCategory(Color.Pink, Color.DarkRed, 1);
nWords.FilterExpression = "[nam] Like 'N*'";
nWords.LegendText = "N - Words";
scheme.ClearCategories();
scheme.AddCategory(queensland);
scheme.AddCategory(nWords);
myLayer.ShowLabels = true;
myLayer.Symbology = scheme;
```

Figure 149: C# Code - Custom Polygon Categories

Figure 150 shows the result of the scheme code illustrated in Figure 149.

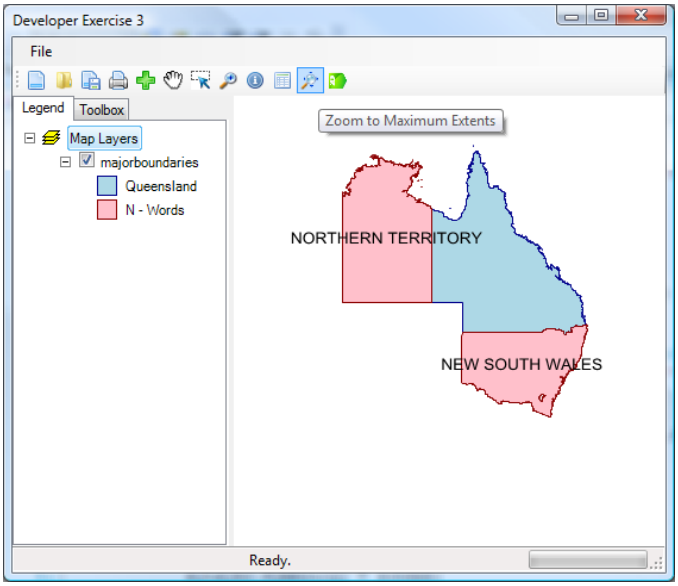


Figure 150: Custom Categories

7.5.7. Compound Patterns

One thing in particular to note about this next example is that in all the previous examples with multiple patterns, there was a line of code that cleared out the default pattern that was automatically created as part of the symbolizer. When a new pattern is added, the new pattern gets drawn on top of the previous patterns, so the last pattern added has the highest drawing priority. In the code that follows, the new pattern has its background color set to transparent, yet in Figure 152 the coloring is red stripes against a blue background. The pattern below the red-stripe pattern is the default pattern, and will

be randomly generated as a different color each time. Figure 151 shows the code for creating a red diagonal hatch pattern, with a default solid color behind it.

```
PolygonSymbolizer mySymbolizer = new PolygonSymbolizer();
mySymbolizer.Patterns.Add(
new HatchPattern(HatchStyle.WideDownwardDiagonal, Color.Red,
Color.Transparent));
myLayer.Symbolizer = mySymbolizer;
```

Figure 151: C# Code - Hatch Pattern Polygons

Figure 152 shows the hatch pattern symbolizing the layer on the map.

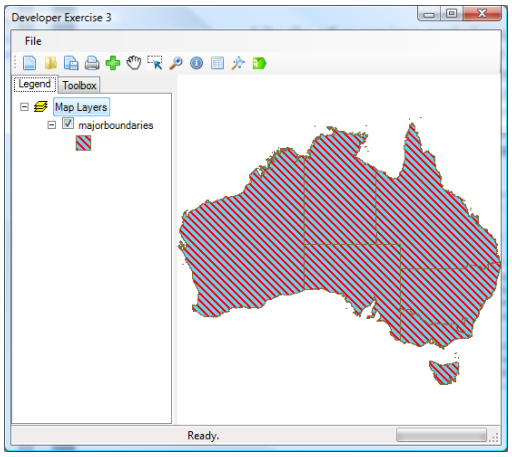


Figure 152: Hatch patterns

7.6.Programmatic Labels

Specifying a constant label expression like “Test” will create the same label for all the features. The background, alignment, halo and other characteristics are all using their default values. Figure 153 shows the code for creating a basic label layer that is linked to the polygon layer.

```
IMapLabelLayer labelLayer = new MapLabelLayer();
labelLayer.Symbology.Categories[0].Expression = "Test";
myLayer.ShowLabels = true;
myLayer.LabelLayer = labelLayer;
```

Figure 153: C# Code – Label Features

Figure 154 shows the labels after they have been added to the polygon layers on the map.

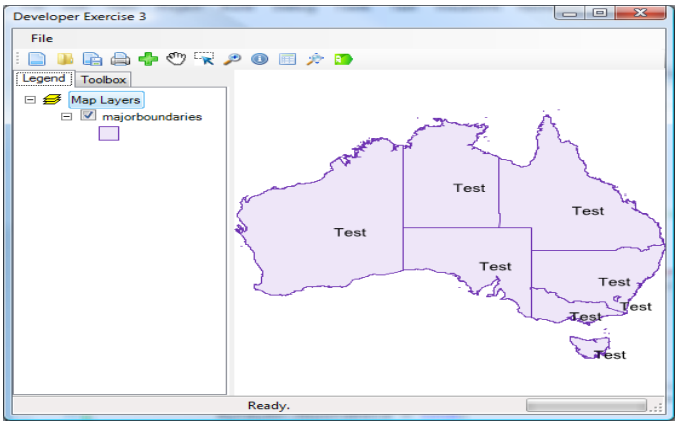


Figure 154: Polygons with Labels

7.6.1. Field Name Expressions

The field name in this case describes the name of the territory. In order for the value of the *nam* field to be substituted with the specific value of that field in each label, the field name should be enclosed in square brackets. Supporting field names in an expression through substitution creates a very versatile way to create complex expressions. Multi-line labels can be created through the use of escape characters in the text string. Figure 155 shows the C# code for setting the orientation to be middle center.

```
IMapLabelLayer labelLayer = new MapLabelLayer();
ILabelCategory category =
labelLayer.Symbology.Categories[0];
category.Expression = "[nam]";
category.Symbolizer.Orientation =
ContentAlignment.MiddleCenter;
myLayer.ShowLabels = true;
myLayer.LabelLayer = labelLayer;
```

Figure 155: C# Code - Centered Field Labels

Figure 156 shows the centered labels that represent the [nam] field of the polygon shapefile.

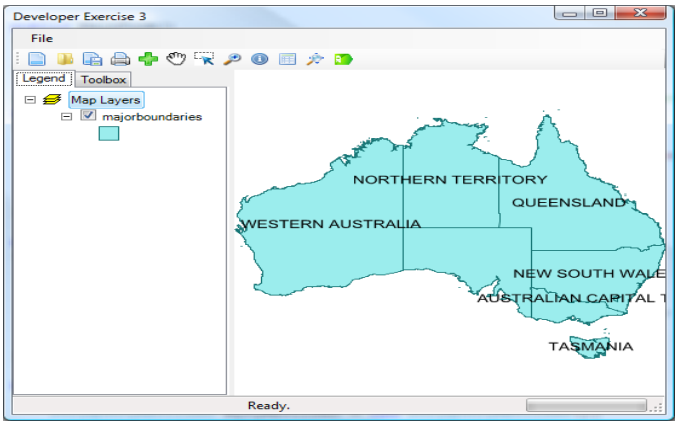


Figure 156: Field Name Labels

7.6.2. Multi-Line Labels

Creating multi-line labels is accomplished using the standard .NET new-line character, which in C# is added using the /n, while in Visual Basic the two strings are combined with a *vbNewLine* element between them. The relative position of the multiple lines is controlled by the *Alignment* property on the label symbolizer. In order to minimize confusion, the labels follow the same organization with a scheme, categories and symbolizers. A filter expression also controls which features receive the labels. Figure 157 shows the use of a filter expression in order to restrict which features are labeled in addition to showing a label with multiple lines.

```
IMapLabelLayer labelLayer = new MapLabelLayer();
ILabelCategory category =
labelLayer.Symbology.Categories[0];
category.Expression = "[nam]\nID: [Cnt_nam]";
category.FilterExpression = "[nam] Like 'N*'";
category.Symbolizer.BackColorEnabled = true;
category.Symbolizer.BorderVisible = true;
category.Symbolizer.Orientation =
ContentAlignment.MiddleCenter;
category.Symbolizer.Alignment =
StringAlignment.Center;
myLayer.ShowLabels = true;
myLayer.LabelLayer = labelLayer;
```

Figure 157: C# Code - Multiline Labels

Figure 158 shows the centered, multi-line labels that use filters to restrict which features are labeled.

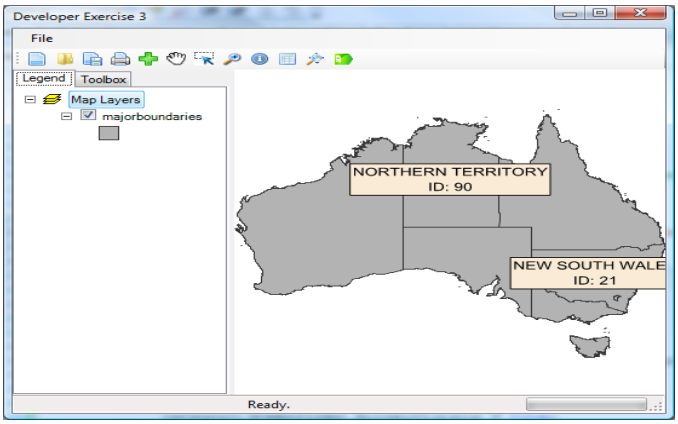


Figure 158: Multi-Line Labels

Multiple label categories can be created and added to the schemes. The label symbolizer also allows for the font, text color, background color and opacity to be controlled.

7.6.3. Translucent Labels

The background color in this case has been set to *transparent* by specifying an Alpha value of something less than 255 when setting the *BackColor*. With .Net drawing, opacity is controlled through the Alpha channel, which can store a byte value from 0 to 255, where 0 is transparent and 255 is fully opaque. The OGC term "opacity", on the other hand, controls the same transparency, where a value of 0 is fully transparent and 1 is fully opaque. There are opacity properties that appear in several places, but they simply allow direct control of the Alpha channel using the 0 to 1 range. This example also illustrates the use of the compound conjunction "OR" in the filter expression. Other powerful terms that can be used are "AND" and "NOT" as well as the combined expression "Is Null," which is case insensitive and can identify null values separately from empty strings for instance. Notice that *is not "= null"* does not work with .NET DataTables. To express a negative use "NOT [nam] is null". Figure 159 shows the code

for controling the appearance, such as the font coloring and transparency, color, and border style of the label background.

```
IMapLabelLayer labelLayer = new MapLabelLayer();
ILabelCategory category = labelLayer.Symbology.Categories[0];
category.Expression = "[nam]\nID: [Cnt_nam]";
category.FilterExpression = "[nam] = 'Tasmania' OR [nam] = 'Queensland'";
category.Symbolizer.BackColorEnabled = true;
category.Symbolizer.BackColor = Color.FromArgb(128, Color.LightBlue);
category.Symbolizer.BorderVisible = true;
category.Symbolizer.FontStyle = FontStyle.Bold;
category.Symbolizer.FontColor = Color.DarkRed;
category.Symbolizer.Orientation = ContentAlignment.MiddleCenter;
category.Symbolizer.Alignment = StringAlignment.Center;
myLayer.ShowLabels = true;
myLayer.LabelLayer = labelLayer;
```

Figure 159: Conditional Labeling Expressions

Figure 160 shows the colored, translucent labels as they appear on the map.

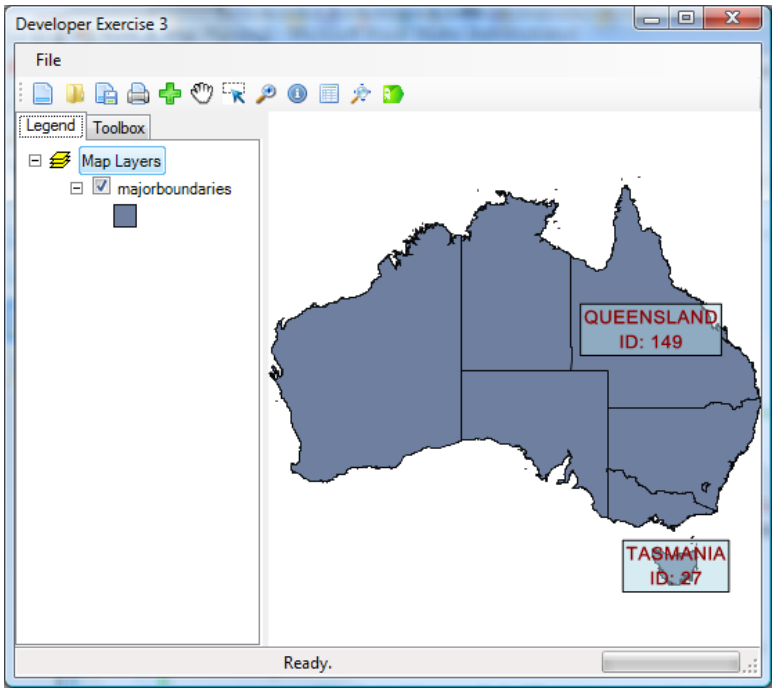


Figure 160: Translucent Labels

7.7. Programmatic Raster Symbology

7.7.1. Download Data

In addition to the vector data that have been examined so far, MapWindow also supports most raster formats. Rasters are considered distinct from images in that the visual representation is derived from the values much in the way that the polygon images

are derived from the actual data. Rasters typically have a rectangular arrangement of values that are organized in rows and columns. For datasets that do not have complete sampling, a "No-Data" value allows the raster to only represent a portion of the total area.

Because of the existence of extensible data format providers, there is no way to tell just how many formats MapWindow will support in the future. A plug-in that is exclusive to the Windows platform has been created using Frank Warmerdan's GDAL libraries and C# linkage files (http://trac.osgeo.org/gdal/). The plug-in exposes many of the raster and image types supported by GDAL to MapWindow 6, or any project that adds the ApplicationManager component. While the situation may change before the beta release, the Sidney Alpha has only one grid format that is supported natively (that is without using GDAL); that is a bgd format. Having the raster formats provided as an extension is not a problem for independent developers because the system enables them to add a single *ApplicationManager* to the project and empower their own project with all the data format extensions that work with MapWindow 6. However, since that will not be covered at this stage of the tutorials, the MapWindow 6 application will be used to convert a raster file to the necessary format. A provider of GIS data is the United States Geological Survey (USGS). They provide many forms of data from around the world including an elevation dataset that is pertinent to this set of tutorials. A useful web utility for browsing the web is called *EarthExplorer*, http://edcsns17.cr.usgs.gov/EarthExplorer/ Figure 161 shows the Earth Explorer map once it is centered on Australia.

Figure 161: Earth Explorer

The USGS can easily be searched for links that provide data by region if the link above does not work. The map application is an easy to use system that provides a simple data download experience. Unfortunately, the system is under heavy demand and data requests have to be kept to a conservative amount.

To get data, simply zoom into a region of interest to expose numerous data sets. For this demo, activate the SRTM data format, which provides digital elevation information. Take note that only a registered user may download data from this site, and not all the data is free. Multiple file formats are available for download such as BIL and DTED. Both formats are supported by MapWindow by using GDAL. For this tutorial the BIL file was selected. The downloaded file is zipped, so it will need to be unzipped before it can be opened with MapWindow 6. Before the data can be manipulated programmatically, MapWindow 6 will be used to change the data format to bgd. The conversion is accomplished simply by opening the BIL, right clicking on the legend and choosing export data. The coloring may appear strange because it is usual for the no-data values to distort the coloring until they can be properly excluded from the symbology. At this stage it is only necessary to convert the format to an *.bgd file. Figure 162 shows the context menu from the legend that allows saving the raster in a new format.

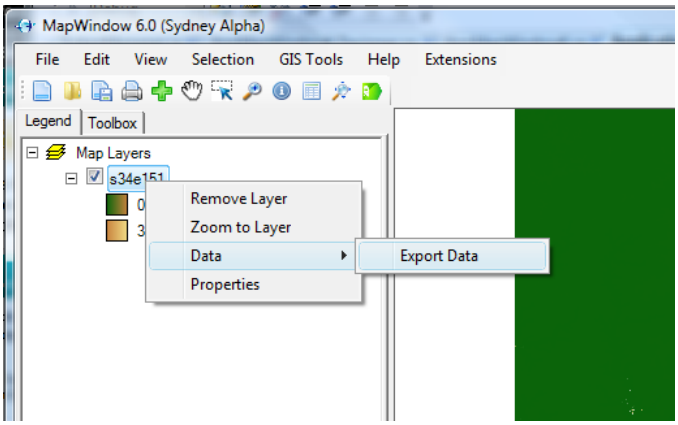


Figure 162: Export Data

Figure 163 shows how to save the data in the .bgd file format, which is supported without the GDAL data provider library.

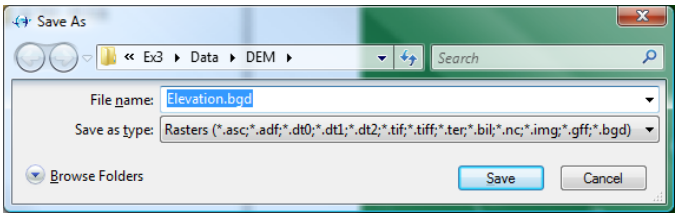


Figure 163: Save as Bgd

Other formats are available. The formats shown in Figure 163 are the *write* formats that have been exposed by the GDAL plugin, as well as any formats that are supported for rasters by other providers. Note that *.bgd is at the end of the list.

7.7.2. Add a Raster Layer

Figure 164 shows the source code for adding the raster file.

```
Raster r = new Raster();
r.Open(@"[Your Folder]\Ex3\Data\DEM\Elevation.bgd");
IMapRasterLayer myLayer = map1.Layers.Add(r);
```

Figure 164: C# Code - Add Raster Layer

One thing that is fairly clear right away is that, similar to the previous examples, a raster is a completely different data object. Instead of working with features and concentrating on ideas like geometries, rasters give direct access to the numerical data

stored in grid form. Inheritance is used again in order to separate out the file format or the data type being used. Most of the logic that connects a file format with the correct programming objects occurs in internal classes, allowing a single Raster class to work regardless of the data source or the data type. The primary benefit is that writing the code becomes a lot simpler because there is no need to worry about what kind of raster is being used. Figure 165 shows the default symbology of an early implementation that was being distorted by the *no-data* values. This behavior has been updated since the time image was recorded, and so this defect has been repaired, but it is useful to mention that the color range should ignore these extreme values.

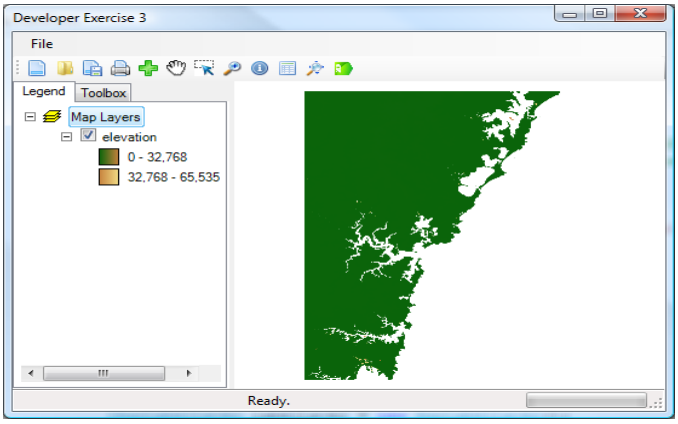


Figure 165: Default Raster

The symbolizer interface for rasters can be used in either MapWindow 6.0 or the new project by double clicking next to the elevation layer in the Legend. Double clicking will launch the dialog shown in Figure 166.

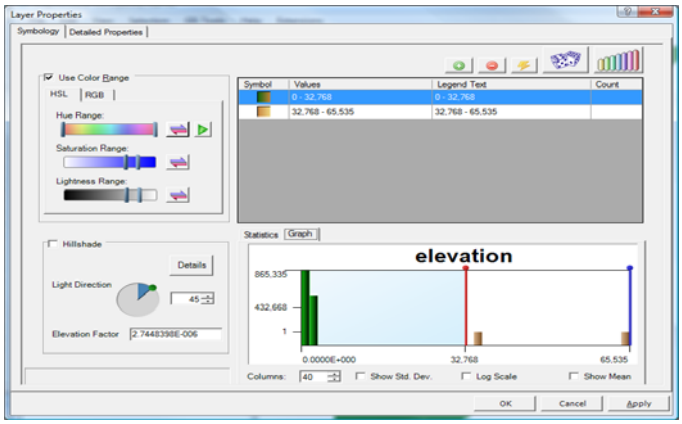


Figure 166: Statistics

The count in "elevation" histogram in Figure 166 shows how many grid cells posses specific values. It can be observed right away that of the many sample points that were taken from this image, which is signified by the height of the brown bars in the middle and on the right of the plot that only have a height corresponding to 1. Most of the values fall in a much smaller range that is much closer to zero, and so almost the entire range of actual values falls into the first or second bin. This can be repaired easily by sliding the sliders down to the left end of the plot. What is less obvious until the display is magnified is that almost all of the values are effectively no-data values of 0 and are showing up on the plot. The mouse wheel can be used to zoom into and out of histogram, but in this case it will be very difficult to see what is going on as long as the range includes zero. There is a technique that can be employed specifically in this situation to manually enter a range from 1 to 250 and from 250 to 600 in the editable values column in the data grid above the graph. This addition will not directly change the graph. Instead, it will enable an innovative feature to work on the graph. A *Zoom To Categories* option when right clicking the graph zooms into the range specified by the categories listed previously. Figure 167 shows the *Zoom To Categories* option in the context menu of the histogram control.

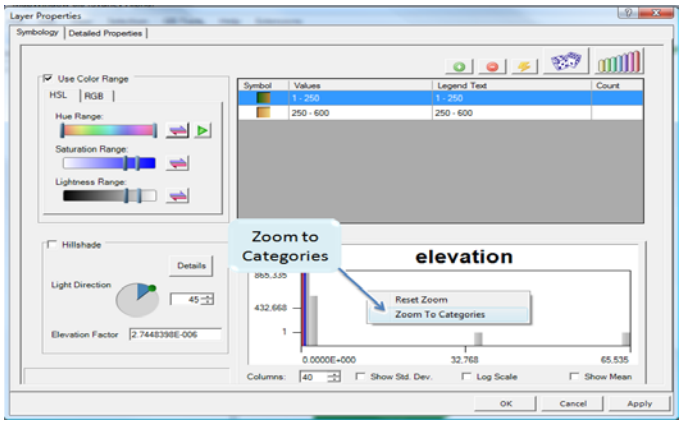


Figure 167: Zoom To Categories

Figure 168 shows the histogram control after zooming to the categories.

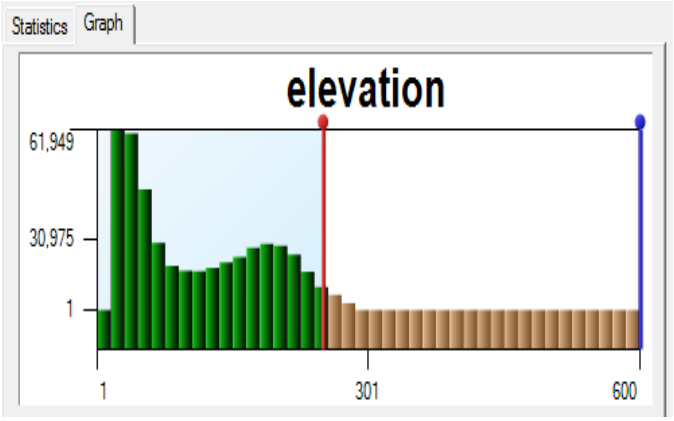


Figure 168: Raster Cell Value Histogram After Zoom

It can be observed from the statistics here, that a large percentage of the cell values occur between about 1 and 300. Since most of the bins in the brown category have a value of one, it should be clear that with the current breaks, very few cells will actually be colored brown by this scheme. A more cleanly symbolized raster can be achieved by sliding the red slider to the left so that more values will fall into the brown category. The *no-data* region is still white, but now the available values can be seen more easily since the outliers have been eliminated. Figure 169 shows how the symbol range is depicted after some adjustments have been made.

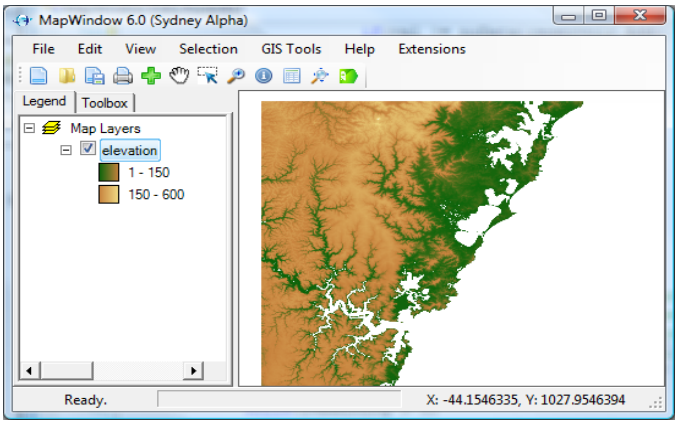


Figure 169: After Adjustments

In order to set this up programmatically, the range of the categories that are automatically generated when adding the data layer needs to be edited so that the range is from 1-150 and from 150 to 600.

7.7.3. Control Category Range

The range values can be controlled independently by modifying the legend text. In order to update the legend text to display the actual minimum and maximum values, the *ApplyMinMax* method can be used. Alternately, the legend text can be set directly just as it would be for the other categories. Figure 170 shows the C# code to create two separate ranges, and to ensure that the correct range appears in the Legend with a conventional string format.

```
private void ControlRange(IMapRasterLayer myLayer)
{
 myLayer.Symbolizer.Scheme.Categories[0].Range = new Range(1, 150);
 myLayer.Symbolizer.Scheme.Categories[0].ApplyMinMax(myLayer.Symbolizer.EditorSettings);
 myLayer.Symbolizer.Scheme.Categories[1].Range = new Range(150, 800);
 myLayer.Symbolizer.Scheme.Categories[1].ApplyMinMax(myLayer.Symbolizer.EditorSettings);
 myLayer.WriteBitmap();
}
```

Figure 170: C# Code - Multiple Category Rasters

Figure 171 shows the categories as they appear on the map and in the Legend.

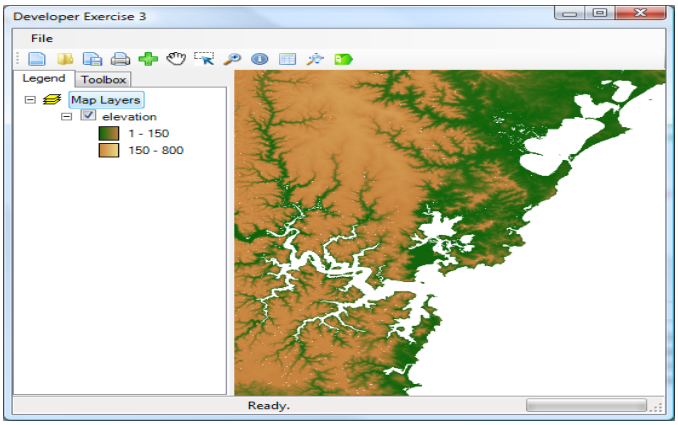


Figure 171: Programmatically Restricted Range

7.7.4. Shaded Relief

Shaded relief refers to the application of light and shadows to elevation datasets in a way that creates the appearance of three dimensional contours. Figure 172 shows the program code for creating shaded relief with two categories.

```
myLayer.Symbolizer.Scheme.Categories[0].Range = new Range(1, 150);
myLayer.Symbolizer.Scheme.Categories[0].ApplyMinMax(myLayer.Symbolizer.EditorSettings);
myLayer.Symbolizer.Scheme.Categories[1].Range = new Range(150, 800);
myLayer.Symbolizer.Scheme.Categories[1].ApplyMinMax(myLayer.Symbolizer.EditorSettings);
myLayer.Symbolizer.ShadedRelief.ElevationFactor = 1;
myLayer.Symbolizer.ShadedRelief.IsUsed = true;
myLayer.WriteBitmap();
```

Figure 172: C# Code - Shaded Relief

Figure 173 shows the shaded relief as it appears for the elevation dataset in the map.

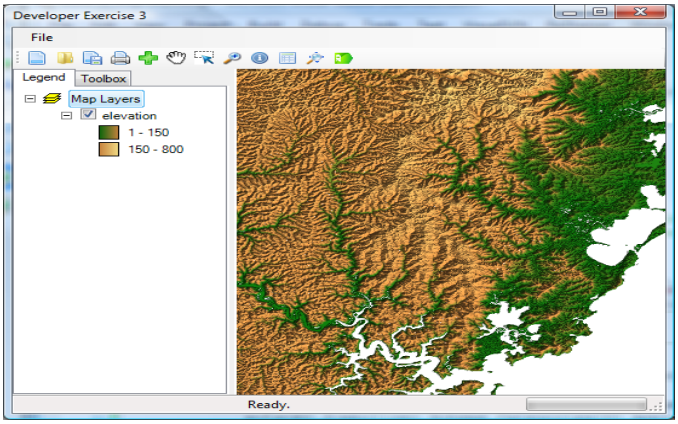


Figure 173: With Lighting

7.7.5. Predefined Schemes

There are several pre-defined color schemes that can be used. All of the pre-set schemes basically use two separate color ramps that subdivide the range and apply what is essentially a coloring theme to the two ranges. Those ranges are easily adjustable using the range characteristics on the category, but should be adjusted after the scheme has been chosen. Otherwise the new scheme will overwrite the previous range choices. Figure 174 shows the code for the *glaciers* predefined coloring schemes.

```
myLayer.Symbolizer.Scheme.ApplyScheme(ColorSchemes.Glaciers, myLayer.DataSet);
myLayer.Symbolizer.Scheme.Categories[0].Range = new Range(1, 150);
myLayer.Symbolizer.Scheme.Categories[0].ApplyMinMax(myLayer.Symbolizer.EditorSettings);
myLayer.Symbolizer.Scheme.Categories[1].Range = new Range(150, 800);
myLayer.Symbolizer.Scheme.Categories[1].ApplyMinMax(myLayer.Symbolizer.EditorSettings);
myLayer.Symbolizer.ShadedRelief.ElevationFactor = 1;
myLayer.Symbolizer.ShadedRelief.IsUsed = true;
myLayer.WriteBitmap();
```

Figure 174: C# Code - Predefined Schemes

Figure 175 shows the same elevation dataset with the *glacier* coloring.

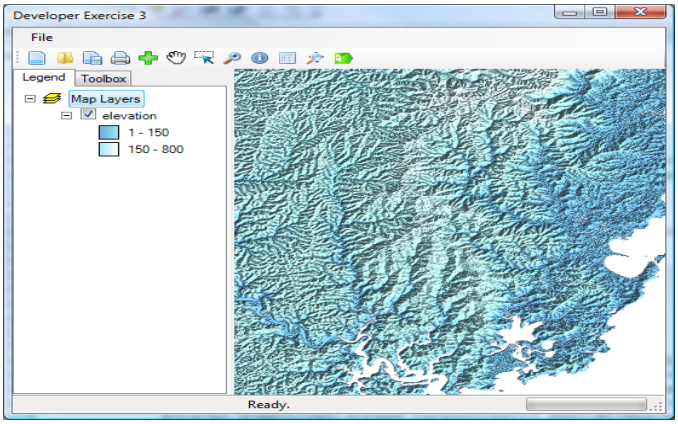


Figure 175: Glaciers

7.7.6. Edit Raster Values

To this point techniques have been used to color the elevation while the values that were either *no-data* values that read 0, or else impossible values like 65,000, were directly ignored. In this section, the raster data class itself will be used in order to repair the values programmatically. This alteration will only alter the copy that is currently

stored in memory and will not overwrite the values to the disk specifically unless instructed to do so. All of the values in the raster can be cycled through with the help of the *NumRows* and *NumColumns* properties. The two properties give the upper bounds for the loops. The *Value* property takes a double index and will work with whatever the underlying data type is and convert that data type into doubles. The ability to cycle through the values gives developers a tool to quickly clean up the values on the raster before the symbolic representation for that raster is created. Also, the *no-data* values on the raster can be assigned to match the 0 values that cover a large portion of the raster. Reassigning the values will automatically eliminate them from the statistical calculations so that the default symbology looks clearer.

Figure 176 shows the C# code for trimming the values that are above 600 so that they have a maximum value of 600.

```
Raster r = new Raster();
r.Open(@"C:\dev\MapWindow6Dev\Tutorial\Components\Ex3\Data\DEM\Elevation.bgd");
r.NoDataValue = 0;
for(int row = 0; row < r.NumRows; row++)
{
    for(int col = 0; col < r.NumColumns; col++)
    {
        if (r.Value[row, col] > 600) r.Value[row, col] = 600;
    }
}
IMapRasterLayer myLayer = map1.Layers.Add(r);
```

Figure 176: C# Code - Crop Extreme Raster Values

Figure 177 shows the fixed raster when it is loaded into the map with the default coloring scheme.

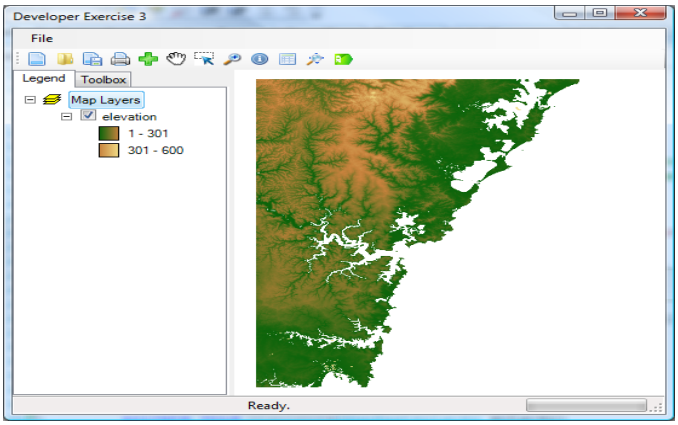


Figure 177: Default Symbology of Fixed Raster

The image is decidedly improved and by double clicking on the elevation layer, the statistical plot can be viewed automatically. Figure 178 shows that the default range contains values from 1 to 600, and does not include in the statistical summary the values that are now labeled as *no-data* values. Assigning the *no-data* value can be risky because there may be values that were using the old *no-data* value. The erroneous no-data values can easily be fixed by cycling through the raster in the same way and adjusting values so that they work with the given statistics. Figure 178 also shows the statistics in the histogram control after fixing the maximum values for the raster.

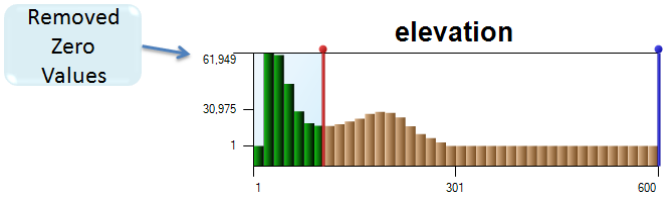


Figure 178: After Fixing Raster

7.7.7. Quantile Breaks

Just like the *FeatureSets*, rasters can use the *EditorSettings* property in order to customize how to build schemes, rather than having to specify the schemes directly. This is where previous edits to fix the raster values become more important. If quantile breaks are applied before the fix, instead of coloring the raster appropriately, all but one of the

ranges would read 0-0. A reasonable range is the result after the fix. Figure 179 shows the C# code for specifying the use of quantile breaks for raster symbology.

```
myLayer.Symbolizer.EditorSettings.IntervalMethod = IntervalMethods.Quantile;
myLayer.Symbolizer.EditorSettings.NumBreaks = 5;
myLayer.Symbolizer.Scheme.CreateCategories(myLayer.DataSet);
myLayer.Symbolizer.ShadedRelief.ElevationFactor = 1;
myLayer.Symbolizer.ShadedRelief.IsUsed = true;
myLayer.WriteBitmap();
```

Figure 179: C# Code - Quantile Breaks

Figure 180 shows the raster layer with quantile breaks and shaded relief as it appears in the map.

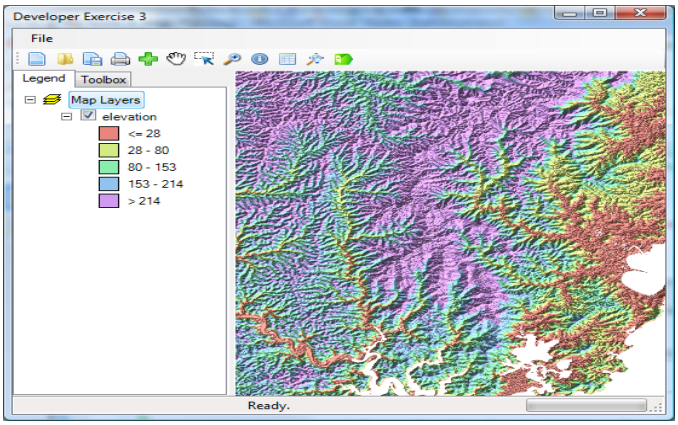


Figure 180: Quantile Breaks

8. Conclusion

In conclusion, the primary focus of the research effort was the authoring of the fully functional desktop GIS. This in itself is a worthy challenge, requiring an inordinate number of lines of code to be written, a vast understanding of geographic algorithms and C# programming. In addition, countless changes and revisions were necessary to improve performance before the software was genuinely useable. This dissertation revealed several unspoken problems and design concerns in the 2D Vector Drawing section, as well as discussing the ramifications of several design choices instrumental for the construction of a GIS. Further, it serves as a record of some new algorithmic

processes that were introduced during the course of this study that are related to the application of this software to environmental modeling. The components developed by this research effort are usable for building both proprietary and open-source software, and serve as the starting point for future geospatial software research. The architectural extensibility and interchangeability represent a significant advancement for the development of open source GIS in the future.

References

Abrahamsson, P., Salo, O. Ronkainen, J. and Warsta, J., 2002. Agile software development methods: Review and analysis. VTT Electronics, p. 78.

Alarcon, V.J. and O'Hara, C.G., 2006. Advanced Techniques for Watershed Visualization, Applied Imagery and Pattern Recognition Workshop, 2006. AIPR 2006. 35th IEEE, pp. 30-30.

Ames, D., P., Michaelis, C., Anselmo, A., Chen, L. and Dunsford, H., 2008. MapWindow GIS. In: S. Shekhar and H. Xion (Editors), Encyclopedia of GIS. Springer, New York, pp. 633-634.

Ames, D.P., 2007. MapWinGIS Reference Manual: A function guide for the free MapWindow GIS ActiveX component. Lulu.com, Morrisville, North Carolina, 194 pp.

Ames, D.P., Michaelis, C. and Dunsford, H., 2007. Introducing the MapWindow GIS Project. OSGeo, 2(8-10): 13-16.

An, J., Chen, H., Furuse, K., Ishikawa, M. and Ohbo, N., 2002. The convex polyhedra technique: an index structure for high-dimensional space. Australian Computer Science Communications 24(2): 33-40.

Angiulli, F., 2007. Fast Nearest Neighbor Condensation for Large Data Sets Classification. IEEE Transactions on Knowledge and Data Engineering 19(11): 1450-1464.

Aref, W.G. and Samet, H., 1997. Efficient Window Block Retrieval in Quadtree-Based Spatial Databases. GeoInformatica, 1(1): 59-91.

Arge, L. et al., 2003. Efficient Flow Computation on Massive Grid Terrain Datasets. GeoInformatica, 7(4): 283.

Bayer, R., 1972. Symmetric binary B-Trees: Data structure and maintenance algorithms. Acta Informatica, 1(4): 290-306.

Beckmann, N., Kriegel, H.-P., Schneider, R. and Seeger, B., 1990. The R*-tree: an efficient and robust access method for points and rectangles. ACM SIGMOD Record 19(2): 322-331.

Bentley, J.L., 1975. Multidimensional binary search trees used for associative searching. Communications of the ACM 18(9): 509-517.

Bentley, J.L., 1990. Kd-trees for semidynamic point sets, Proceedings of the sixth annual symposium on Computational geometry. ACM, Berkley, California, United States.

Berchtold, S., Bohm., C., Keim, D.A. and Kriegel, H.-P., 1997. A cost model for nearest neighbor search in high-dimensional data space, Proceedings of the sixteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems. ACM, Tucson, Arizona, United States.

Berg, M. and Kreveld, M., Overmars, M., Schwarzkopf, O., 1997. Computational Geometry. Springer, Berlin, 367 pp.

Bern, M., Eppstein, D., and Teng, S. H., 1999. Parallel Construction of Quadtrees and Quality Triangulations. International Journal of Computational Geometry and Applications, 9(6): 16.

Bernard, L., Schmidt, B., Streit, U. and Uhlenküken, C., 1998. Managing, Modeling, and Visualizing High-dimensional Spatio-temporal Data in an Integrated System. GeoInformatica, 2(1): 59-77.

Bittner, J., 2002. Efficient construction of visibility maps using approximate occlusion sweep, Proceedings of the 18th spring conference on Computer graphics. ACM, Budmerice, Slovakia.

Bitzer, J.S. and Schröder P.J., 2006. The Economics of Open Source Software Development. Elsevier, Amsterdam.

Bivand, R. and Neteler, M., 2000. Open source geocomputation: using the R data analysis language integrated with GRASS GIS and PostgreSQL data base systems, Proc. 5th conference on GeoComputation (CDROM), 23-25 August 2000, University of Greenwich, U.K.

Blazek, R., Neteler, M. and Micarelli, R., 2002. The new GRASS 5.1 vector architecture, Open source GIS -- GRASS users conference 2002, Trento, Italy, 11-13 September 2002.

Blind, K. and Edler, J., 2003. Idiosyncrasies of the Software Development Process and Their Relation to Software Patents: Theoretical Considerations and Empirical Evidence. NETNOMICS, 5(1): 71-96.

Boissonnat, J.-D. and Yvinec, M., 1995. Algorithmic Geometry. Cambridge University Press, Sophia-Antipolis, France, 519 pp.

Bolstad, P., 2008. GIS Fundamentals; a First Text on Geographic Information Systems, 3rd Edition. Eider Press, New York.

Botea, V., Mallett, D., Nascimento, M. and Sander, J., 2008. PIST: An Efficient and Practical Indexing Technique for Historical Spatio-Temporal Point Data. GeoInformatica, 12(2): 143-168.

Brockwell, P.J. and Davis, R. A., 1996. Introduction to Time Series and Forecasting. Springer, Springer-Verlag, 420 pp.

Brodsky, A., Lassez, C., Lassez, J.-L. and Maher, M.J., 1995. Separability of polyhedra for optimal filtering of spatial and constraint data, Proceedings of the 14th ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems. ACM, San Jose, California, United States.

Brooks, F., 1995. The Mythical Man-Month: Essays on Software Engineering. Addison-Wesley Professional.

Cannady, J.M., 1978. Balancing Methods for Binary Search Trees, Proceedings of the 16th annual Southeast regional conference. ACM, Atlanta, Georgia.

Chazelle, B., 1991. Triangulating a Simple Polygon in Linear Time. Discrete and Computational Geometry, 6: 40.

Chazelle, B., Goodman, J.E. and Pollack, R., 1996. Advances in Discrete and Computational Geometry. Contemporary Mathematics. American Mathematical Society, Providence, Rhode Island, 463 pp.

Cheng, X., Lu, H. and Hedrick, G.E., 1992. Searching Spatial Objects with Index by Dimensional Projections, Proceedings of the 1992 ACM/SIGAPP Symposium on

Applied Computing: Technological Challenges of the 1990's. Kansas City, Missouri, United States.

Cleary, J.G., 1979. Analysis of an Algorithm for Finding Nearest Neighbors in Euclidean Space. ACM Transactions on Mathematical Software (TOMS) 5(2): 183-192.

Dai, X., Yiu, M.L., Mamoulis, N., Tao, Y. and Vaitis, M., 2005. Probabilistic Spatial Queries on Existentially Uncertain Data. Advances in Spatial and Temporal Databases, pp. 400-417.

Daming, W., Yuan, T., Yong, G. and Lun, W., 2005. A new method of generating grid DEM from contour lines. IEEE Transactions on Geoscience and Remote Sensing 4(5):657:660.

Davis, M.L., Cornwell, David A., 1998. Introduction to Environmental Engineering. Water Resources and Environmental Engineering. McGraw-Hill, Boston, MA, 917 pp.

de Joode, R., 2004. Innovation in open source communities through processes of variation and selection. Knowledge, Technology, and Policy, 16(4): 30-45.

Dunsford, H. and Ames, D.P., 2007. A New, Faster, Scalable PitFill Algorithm, 6th International Symposium of Environmental Software Systems (ISESS07), Prague, Czech Republic.

Dunsford, H. and Ames, D.P., 2008. An Extensible, Interface-Based, Open Source GIS Paradigm: MapWindow 6.0 Developer Tools for the Microsoft Windows Platform. Free and Open Source Software for Geoinformatics (FOSS4G), Cape Town.

Dunsford, H., Ames, D.P., Laniak, G. and Kittle, J., 2008. Community Code Development: A New Paradigm for Geospatial Software in Support of the Data for Environmental Modeling(D4EM) Project., AWRA Spring Specialty Conference GIS and Water Resources V, San Mateo, California.

EPA, 2008. Private Drinking Water Wells

Eppstein D., Bern M., and Hutchings, B., 2007. Algorithms for Coloring Quadtrees. Algorithmica. Proceedings of the 7th annual conference on Computer graphics and interactive techniques 32(1): 87-94.

Fitzgerald, B. and Kenny, T., 2004. Developing an information systems infrastructure with open source software. Software, IEEE, 21(1): 50-55.

Foundation, M., Mozilla Public license.

Fuchs, H., Kedem, Z., M. and Naylor, B., F., 1980. On visible surface generation by a priori tree structures. 14(3): 124-133.

Gatalsky, P. and Andrienko, N., 2004. Interactive Analysis of Event Data Using Space-Time Cube. N. Andrienko (Editor), Information Visualization, 2004. IV 2004. Proceedings., pp. 145-152.

GNU, 2007. GNU General Public Licence. Free software Foundation.

Goovaerts, P., 1997. Geostatistics for Natural Resources Evaluation. OXFORD UNIVERSITY PRESS, New York, 483 pp.

Gosling, J., Joy, B. and Steele, G., 2000. The Java Language Specification. Addison-Wesley Professional. Boston. pp xxiii - xxv

Greenberg, S., 2007. Toolkits and interface creativity. Multimedia Tools and Applications, 32(2): 139-159.

Greenberg, S. and Fitchett, C., 2001. Phidgets: Easy Development of Physical Interfaces through Physical Widgets. Proc ACM UIST: 209-218.

Guibas, L. J. and Sedgewick, R., 1978. A dichromatic framework for balanced trees, Foundations of Computer Science, 1978., 19th Annual Symposium on, pp. 8-21.

Hartmann, E., 1998. A marching method for the triangulation of surfaces. The Visual Computer, 14(3): 95-108.

Haugerud, R.A. et al., 2003. High-Resolution Lidar Topography of the Puget Lowland, Washington; A Bonanza for Earth Science, pp. 4-10.

Havesi, J.A., Istok, J. D., and Flint, A. L. , 1992. Precipitation estimation in Mountainous Terrain Using Multivariate Geostatistics. Part I: Structural Analysis. Journal of Applied Meteorology, 31: 16.

Held, M., 1998. Efficient and reliable triangulation of polygons, Computer Graphics International, 1998. Proceedings, pp. 633-643.

Henrich, A., 1996. Adapting a spatial access structure for document representations in vector space, Proceedings of the fifth international conference on Information and knowledge management. ACM, Rockville, Maryland, United States.

Henrich, A., 1996. A hybrid split strategy for kd-tree based access structures, Proceedings of the 4th ACM international workshop on Advances in geographic information systems. ACM, Rockville, Maryland, United States.

Hjaltason, G.R. and Samet, H., 2002. Speeding up construction of PMR quadtree-based spatial indexes. The VLDB Journal The International Journal on Very Large Data Bases, 11(2): 109-137.

Hofierka, J. and Suri, M., 2002. The solar radiation model for Open Source GIS: Implementation and applications. In: M. Ciolli and P. Zatelli (Editors), Proc., Open Source Free Software GIS - GRASS users conference, Trento, Italy Sept. 2002, Trento, pp. 11-13.

Hsiao, P.Y., 1996. Nearly Balanced Quad List Quad Tree - A Data Structure for VLSI Layout Systems. VLSI Design, 4(1): 16.

Hwang, N.H.C. and Houghtalen, R. J., 1996. Fundamentals of Hydraulic Engineering Systems. Prentice Hall, Upper Saddle River, New Jersey, 416 pp.

Jenson, S.K. and Domingue, J. O., 1988. Extracting Topographic Structure from Digital Elevation Data for Geographic Information System Analysis. Photogrammetric Engineering & Remote Sensing, 54(11): 8.

Lamot, M. and Zalik, B., 1999. An overview of triangulation algorithms for simple polygons. In: B. Zalik (Editor), Information Visualization, 1999. Proceedings., pp. 153-158.

Lamot M., Zalik, B. 2000. A Contribution to Triangulation Algorithms for Simple Polygons. Journal of Computing and Information Technology, 4: 319-331.

Lightstone, M. and Mitra, S.K., 1997. Quadtree Optimization for Image and Video Coding. The Journal of VLSI Signal Processing, 17(2): 215-224.

Luna, F., 2003. Introduction to 3D Game Programming with DirectX 9.0. Wordware Publishing, Inc., Plano, Texas, 388 pp.

Marks, D., Dozier, J., and Frew, J., 1984. Automated Basin Delineation from Digital Elevation Data. Geo-Processing, 2: 13.

Martz, L.W., and De Jong, E., 1988. CATCH: A Fortran Program for Measuring Catchment Area from Digital Elevation Models. Computers & Geosciences, 14(5): 14.

McKenna, M., 1987. Worst-case optimal hidden-surface removal. ACM Transactions on Graphics (TOG) 6(1): 19-28.

Michaelis, C. and Ames, D.P., 2008. Web Mapping Service (WMS) Web Feature Service (WFS) Web Processing Service (WPS). In: S. Shekhar and H. Xion (Editors), Encyclopedia of GIS. Springer, New York, pp. 1259-1261.

Miller, T., 2004. Managed DirectX 9 Graphics and Game Programming. Sams Publishing, Indianapolis, IN, 411 pp.

Mokbel, M. and Aref, W., 2007. SOLE: Scalable On-line Execution of Continuous Queries on Spatio-Temporal Data Streams. The VLDB Journal - The International Journal on Very Large Data Bases. 17 (5) pp. 971-995.

Mulmuley, K., 1989. An efficient algorithm for hidden surface removal, Proceedings of the 16th annual conference on Computer graphics and interactive techniques. ACM.

Neteler, M., Mitasova, H., 2008. Open Source GIS: A GRASS GIS Approach. Springer, New York. pp. 3-6.

Pang, M.Y.C. and Shi, W., 2002. Development of a Process-Based Model for Dynamic Interaction in Spatio-Temporal GIS. GeoInformatica, 6(4): 323-344.

Panse, C., Sips, M., Keim, D. and North, S., 2006. Visualization of Geo-spatial Point Sets via Global Shape Transformation and Local Pixel Placement. IEEE Transactions on Visualization and Computer Graphics, 12(5): 749-756.

Pissinou, N., Radev, I. and Makki, K., 2001. Spatio-Temporal Modeling in Video and Multimedia Geographic Information Systems. GeoInformatica, 5(4): 375-409.

Planchon, O. and Darboux, F., 2002. A Fast, Simple and Versatile Algorithm to Fill the Depressions of Digital Elevation Models. CATENA, 46(2-3): 159-176.

Proietti, G., 1999. An Optimal Algorithm for Decomposing a Window into Maximal Quadtree Blocks. Acta Informatica, 36(4): 257-266.

Raymond, E., 2001. The Cathedral & the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary, Revised Edition. O'Reilly. Sebastopol CA. pp. 19-65.

Robbins, J., Feller, J., Fitzgerald, B., Hissam, S. and Lakhani, K., 2005. Adopting Open Source Software Engineering (OSSE) Practices by Adopting OSSE Tools, Perspectives on Free and Open Source Software. The MIT Press, pp. 245-264.

Robertson, C., Nelson, T., Boots, B. and Wulder, M., 2007. STAMP: spatial–temporal analysis of moving polygons. Journal of Geographical Systems, 9(3): 207-227.

Robinson, J.T., 1981. The K-D-B-tree: a search structure for large multidimensional dynamic indexes, Proceedings of the 1981 ACM SIGMOD International conference on Management of data. ACM, Ann Arbor, Michigan.

Snyder, J. and Lengyel, J., 1998. Visibility sorting and compositing without splitting for image layer decompositions, Proceedings of the 25th annual conference on Computer graphics and interactive techniques. New York, NY. pp 219-230.

Soille, P., Vogt, J., and Colombo, R., 2003. Carving and Adaptive Drainage Enforcement of Grid Digital Elevation Models. Water Resources Research, 39(12): 13.

Steiniger, S. and Bocher, E., 2009. An Overview on Current Free and Open Source Desktop GIS Developments. International Journal of Geographical Information Science. 23(10): pp. 1345-1370.

Stout, Q.F. and Warren, B.L., 1986. Tree rebalancing in optimal time and space. Communications of the ACM 29(9): 902-908.

Tarboton, D.G., 1997. A New Method for the Determination of Flow Directions and Upslope Areas in Grid Digital Elevation Models. Water Resources Research, 33(2): 12.

Tarjan, R.E. and Van Wyk, C. J., 1988. An O(n log log n) - Time algorithm for triangulation a simple polygon. SCICOMP, 17(1): 36.

von Krogh, G. and Spaeth, S., 2007. The open source software phenomenon: Characteristics that promote research. The Journal of Strategic Information Systems, 16(3): 236-253.

Waring, T. and Maddocks, P., 2005. Open Source Software implementation in the UK public sector: Evidence from the field and implications for the future. International Journal of Information Management, 25(5): 411-428.

Watry, G. and Ames, D.P., 2007. A Practical Look at MapWindow GIS (1st Edition). Lulu.com, Morrisville, North Carolina, 316 pp.

Weber, S., 2004. The Success of Open Source. Harvard University Press, Cambridge, Massachusetts, 312 pp.

Appendix A: MapWindow 6.0 User's Manual

1. End User Tutorial – Getting Started with MapWindow 6.0

Even though this material is important to conveying the scope of the software created for this dissertation, this overview was written by Mark Van Orden and so is added here as an appendix to this dissertation.

1.1. Introduction

Welcome to MapWindow GIS version 6.0! This is the newest version of a highly customizable open source GIS package and represents years of work and improvement on the former 4.x version. For those already familiar with the MapWindow project and for those as yet uninitiated there is plenty new to discover. So let's dive in!

1.1.1. The MapWindow 6 Main Interface

MapWindow 6 has a familiar primary interface similar to most desktop GIS. It includes the *Main Menu*, *Toolbar*, *Legend Window*, and a main *Map Display*. The *Map Display* is always open. The *Legend Window* displays all of the features, active and inactive, concerned with the map in the current project. A number of options have been added to this version to customize and manipulate the symbology and properties of each data type in the *Legend Window*. The *Main Menu* and *Toolbar* contain functions to manage projects and manipulate and query the map. The figure below shows the MapWindow 6 main interface with each of its features labeled.

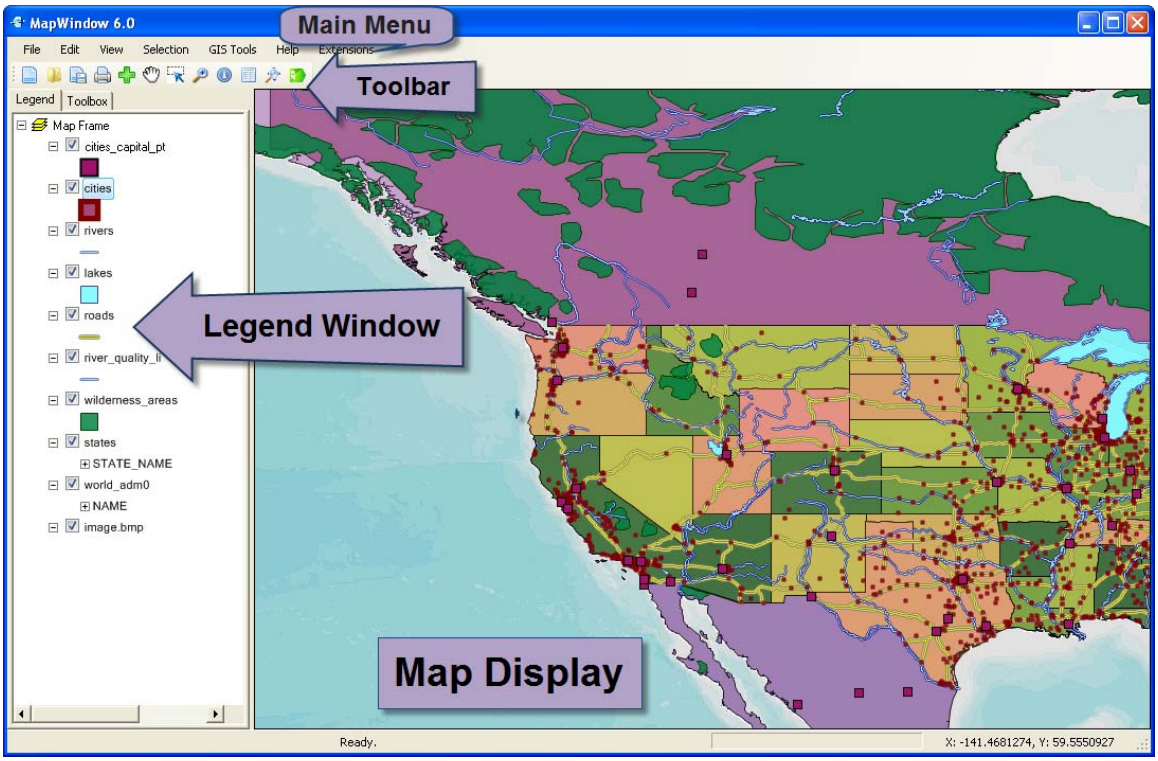


Figure 181: MapWindow 6 Main Interface

1.1.2. The MapWindow 6 Main Menu

The *Main Menu* is located in the upper left of the screen beneath the MapWindow 6 title. It consists of the headings *File*, *Edit*, *View*, *Selection*, *GIS Tools*, *Help* and *Extensions*. The *Main Menu* is still a work in progress and certainly subject to change. It will be addressed to greater length further down the road.

The *File* menu is populated as follows: *New Map, Open Map, Save Map, Open Layer, Settings* and *Print Layout*.

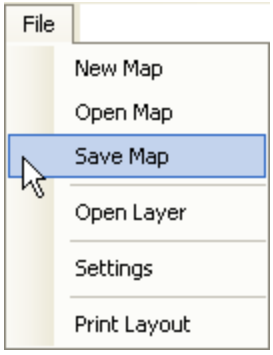


Figure 182: File Menu

- *New Map* – Clears the current map and opens a fresh screen. The user is prompted beforehand to save any changes to the current map.

- *Open Map* – Clears the current map and brings up an 'Open' window that allows the user to navigate to a previously saved map.

- *Save Map* – Allows the user to name the file and location where the map is to be saved on their computer.

- *Open Layer* – Brings up the 'Open' window which allows the user to navigate to and select files containing layers to add to the current project.

- *Settings* – Brings up the 'Settings' window which allows the user to determine *Culture Preference* using the corresponding drop down menu.

- *Print Layout* – This button opens the 'MapWindow Print Layout' window to be discussed at greater length in another chapter.

1.1.3. The MapWindow 6 Toolbar

The *Toolbar* consists of a series of buttons that access commonly used functions. It is located directly under the *Main Menu* in the top left of the screen. It contains functions to manage projects and to manipulate and query the map.

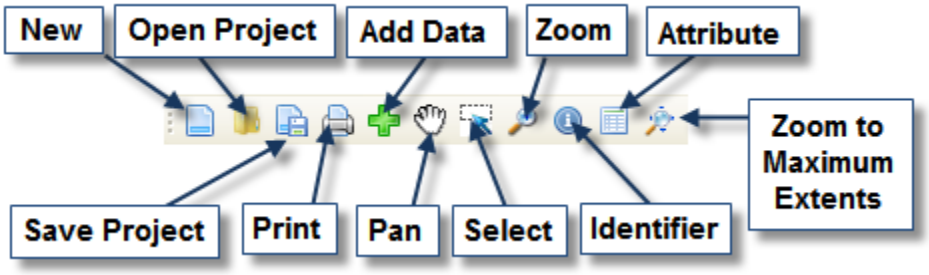


Figure 183: MapWindow 6 Toolbar

New 

This Button opens a new project.

Open Project 

This button allows the user to navigate to a pre-existing project.

Save Project 

Saves any changes made to the current project or allows the user to save a project under a new name and location.

Print 

This button opens the 'MapWindow Print Layout' window. This feature will be discussed at length in another chapter.

Add Data 

This button brings up the 'Open' window which allows the user to navigate to and select files containing layers to add to the current project.

Pan 

This button is used when manipulating the map and changes the cursor to a four directional arrow ✥ when active. Click and hold the left mouse button to grab the map then move the mouse in the desired direction to move the map. Releasing the mouse button releases the map.

Select 

This button selects data in a specified area on the map. Click and drag the mouse to create a rectangle over the desired features. The cursor is changed to a pointing hand ☝ when this function is active.

Zoom 

This button provides the user with two options for zooming in and out. First the user can click the left mouse button to zoom in and the right mouse button to zoom out. Second, the user can zoom to a portion of the map by drawing a rectangle around the desired portion. Note: regardless of which function is active (*Pan*, *Select* or *Zoom*) a mouse with a scroll wheel can zoom by rolling the wheel forward or out by rolling backward

Identifier 

This button changes the mouse to a crosshairs and is used to access detailed information about a desired portion of the map. Click an area of the map to open the 'FeatureIdentifier' window which contains a table of information associated with the selected feature.

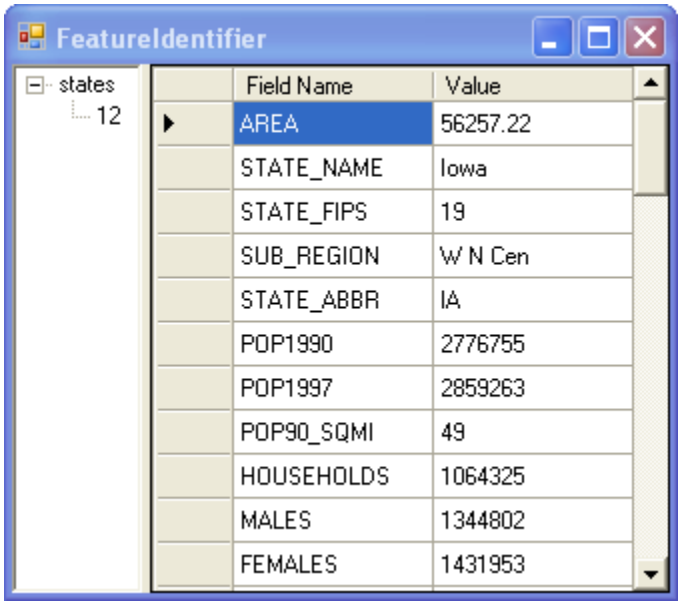


Figure 184: FeatureIdentifier Window

Attributes

This button opens the 'Attribute Table Editor' which will be discussed at length in another chapter.

Zoom to Maximum Extents

Clicking this button automatically zooms the map to its fullest extent.

1.1.4. The MapWindow 6 Legend

The MapWindow 6 Legend displays all the map layers in the current project. It also provides tools to manipulate the layers. The figure below shows the *Legend Window* as it typically appears during a project.

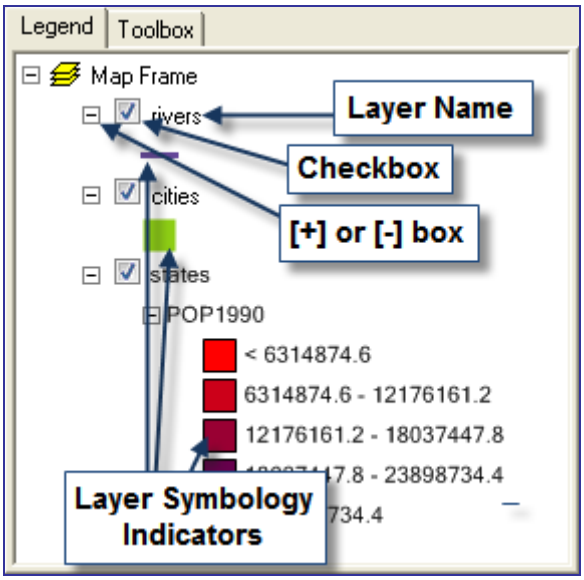


Figure 185: MapWindow 6 Legend Window

Checkboxes

These indicate whether a layer is visible (checked) or hidden (unchecked) in the current view on the *Map Display*.

Layer Symbology

These provide a visual indicator for the layer. If the layer is a polygon, the indicator for that layer will be represented by the color of the polygon. If the layer is a line or point, the icon will be represented by the color and features of the line or point and so on.

[+] or [-] boxes

These expand (+) or collapse (-) to show or hide a layer's details and only appear if a layer has more than one feature such as a color scheme.

1.2. Symbology Controls

The symbology of any given layer can be manipulated by double clicking the indicator for that layer in the 'Legend Window.' This will bring up a dialog box appropriate to the data type of the layer: point, line, polygon (vector) or raster.

1.2.1. Point Symbolizer Dialog

Double clicking the indicator for a layer with point type data brings up the 'Point Symbolizer Dialog' window. This window contains options for managing the various characteristics of the point symbol. It is divided into four main parts: *Symbol Layers*, *Symbolizer Characteristics*, *Symbol Type* and *Placement*.

Also of note are the 'Apply', 'Cancel' and 'OK' buttons at the bottom right of the window. After making any changes within the dialog click the 'Apply' button to apply the changes to the map without closing the window or the 'OK' button to apply the changes and close the window. Click 'Cancel' to ignore the changes and close the window.

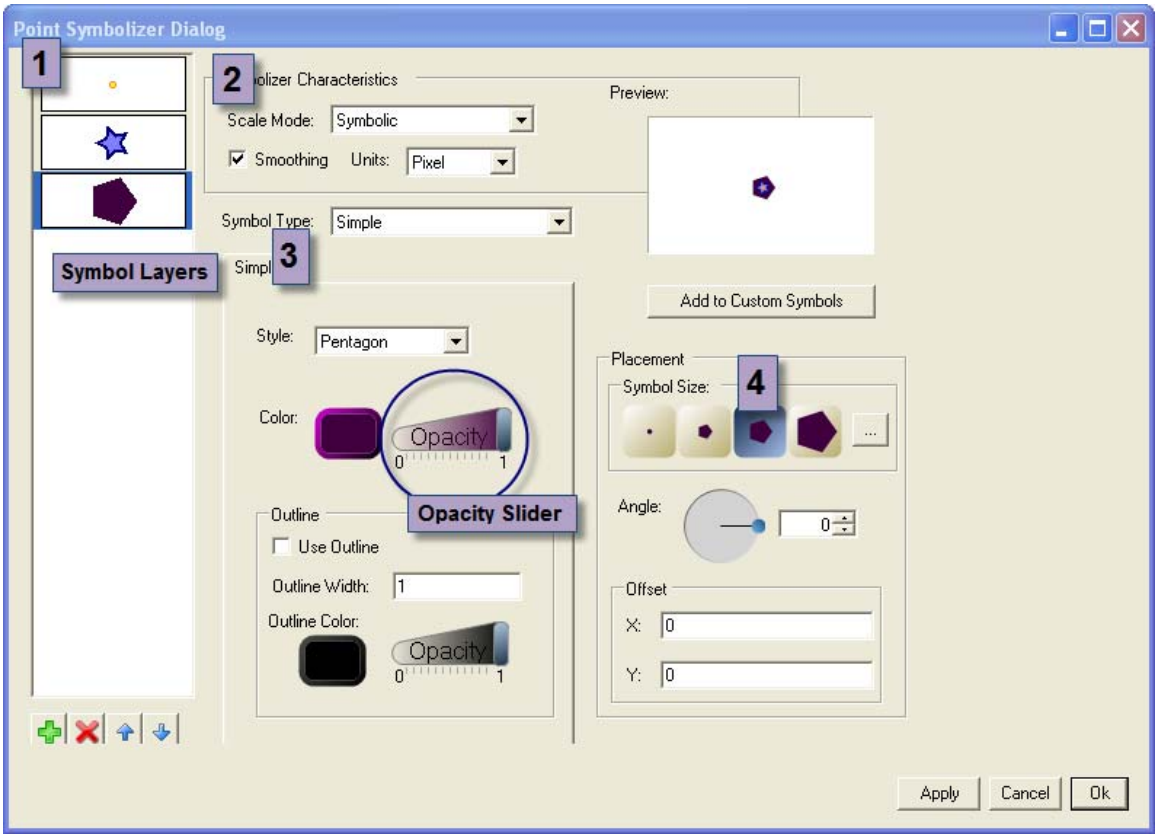


Figure 186: Point Symbolizer Dialog Window

Symbol Layers

The 'Point Symbolizer Dialog' provides a great deal of flexibility in creating a symbol for a point type features in that the symbol itself can be comprised of multiple layers. In section 1 (refer to the figure above) all of the layers for the point symbol are listed with tools  for adding and removing as well as moving a layer up and down the list. The layers adhere to a bottom up drawing order. Note that any changes made to the symbol in the other sections of the dialog only apply to the layer that is currently selected in the *Symbol Layers* list.

Symbol Characteristics

This section of the 'Point Symbolizer Dialog' (refer to number 2 in the figure above) has three main features:

- A drop down list entitled ‘Scale Mode.’ Click the arrow to the right to reveal the options *Symbolic* and *Geographic*.

    - *Symbolic* – The size of the symbol on the map remains static regardless of the extent at which the map is being viewed.

    - *Geographic* – The size of the symbol is relative to the geographic scale of the map.

- A checkbox entitled ‘Smoothing.’ When checked the smoothing feature for the symbol is active.

- A ‘Preview’ box that allows the user to view the changes made to the symbol before applying them.

Symbol Type

The appearance of this section (refer to number 3 above) is determined by a drop down list entitled ‘Symbol Type.’ Click the arrow to the right to reveal the options *Simple*, *Character* and *Picture*.

Figure 187: Symbol Type Options

- *Simple* – Clicking the *Simple* option in the ‘Symbol Type’ drop down list changes this section to appear like the image on the left in figure 6.

  The *Simple* option has three main features:

  - A drop down menu entitled ‘Style’ with the option to change the general shape of the symbol. Click the arrow to the right to access the options *Diamond*, *Hexagon*, *Ellipse*, *Pentagon*, *Rectangle*, *Star* and *Triangle*.

  - A color button and opacity slider for altering the color and transparency of the symbol. Click the color button on the left to bring up the ‘Color’ window with basic color options as well as any user defined custom colors.

    Click the ‘Define Custom Colors’ button to create custom colors by clicking and dragging the crosshairs to the desired spot on the color spectrum. There is also a vertical slider to the right of the spectrum that determines the lightness and saturation of the color. Click the ‘Add to Custom Colors’ button to add the color to the ‘Custom colors’ section of the ‘Color’ window.

    Click OK to accept the changes or ‘Cancel’ to decline. Upon exiting the ‘Color’ window the user can set the transparency of the color with the opacity slider. Note that any changes made will be represented in the preview box.

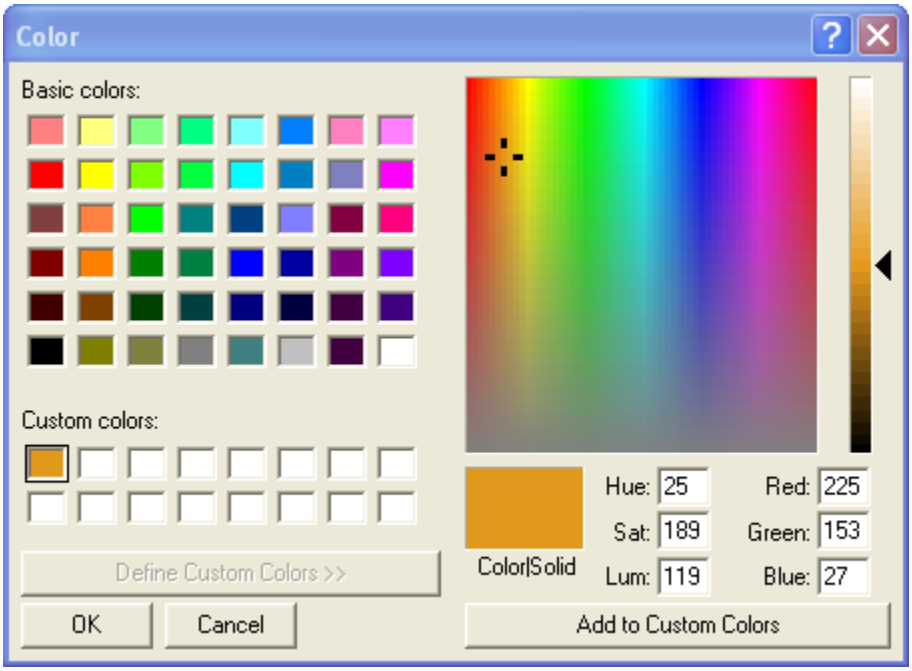


Figure 188: Color Window

- A section that defines the outline of the symbol. This section has a checkbox that determines whether or not to apply an outline. Also present is a field in which the user may type in a number to define the point width of the outline and a color button and opacity slider to establish the outline's color.

- *Character* – Clicking the *Character* option in the 'Symbol Type' drop down list changes this section to appear like the image in the center of figure 6. The *Character* option allows the user to change the symbol to a character within a given font set. A color button and an opacity slider are present to determine the character's color. There is also a drop down list entitled 'Font Family' which accesses a number of font options.

Related to this list is the large grid containing all of the characters as they appear within the chosen font type. Use the scrolling arrow to navigate the grid and find the desired character. Double click the desired panel to apply that character to the symbol's appearance. The field entitled 'Unicode' allows the user to specify which page to view in larger font families.

- *Picture* – Clicking the *Picture* option in the 'Symbol Type' drop down list changes this section to appear like the image on the right in figure 6. This option allows the user to import an image to act as the symbol. Click the ellipse button  to the right of the 'Image' field to navigate to the desired image. There is an opacity slider to establish the transparency of the image itself as well as an 'Outline' section to define the appearance of an outline where applicable.

  Note: Any files of the file types .png, .jpg, .bmp, .gif, and .tif can be used as images in MapWindow 6.

Placement

This section (refer to number 4 in figure 5) defines the size, angle and positioning of the symbol. The 'Symbol Size' field has four preset options to determine the size of the symbol. The preset point sizes are 4, 10.5, 17 and 30 in ascending order. There is also an ellipse button  to the right which allows the user to enter a custom size. This can be helpful when using the *Geographic* scale mode where the size of the symbol needs to fit with the scale of the map.

The ‘Angle’ field provides two means of changing the angle of rotation of the symbol. Click and drag the meter on the left or use the up and down buttons on the right to change the rotation.

The ‘Offset’ field provides a place for the user to determine the positioning of the symbol relative to the feature on the map that it represents. This achieved by entering X and Y coordinates where the feature on the map is at the 0,0 location.

The Select Line Symbol Dialog

Double clicking the indicator for a layer with line type data brings up the ‘Select Line Symbol’ dialog window. This window contains multiple predefined line symbols for the user to choose from. The predefined symbols are separated into the *Transportation*, *Boundaries* and *Rivers* categories. Click the arrow to the right of the ‘Map Symbol Category’ field to access a drop down menu from which to choose one or all of these categories to display in the ‘Choose predefined symbol’ box.

Click on the desired symbol to display its appearance in the ‘Preview’ box. Click the ‘Apply’ button in the bottom right of the window to make the change without closing the window or click the ‘OK’ button to apply the change and close the window. The ‘Cancel’ button ignores the changes and closes the window.

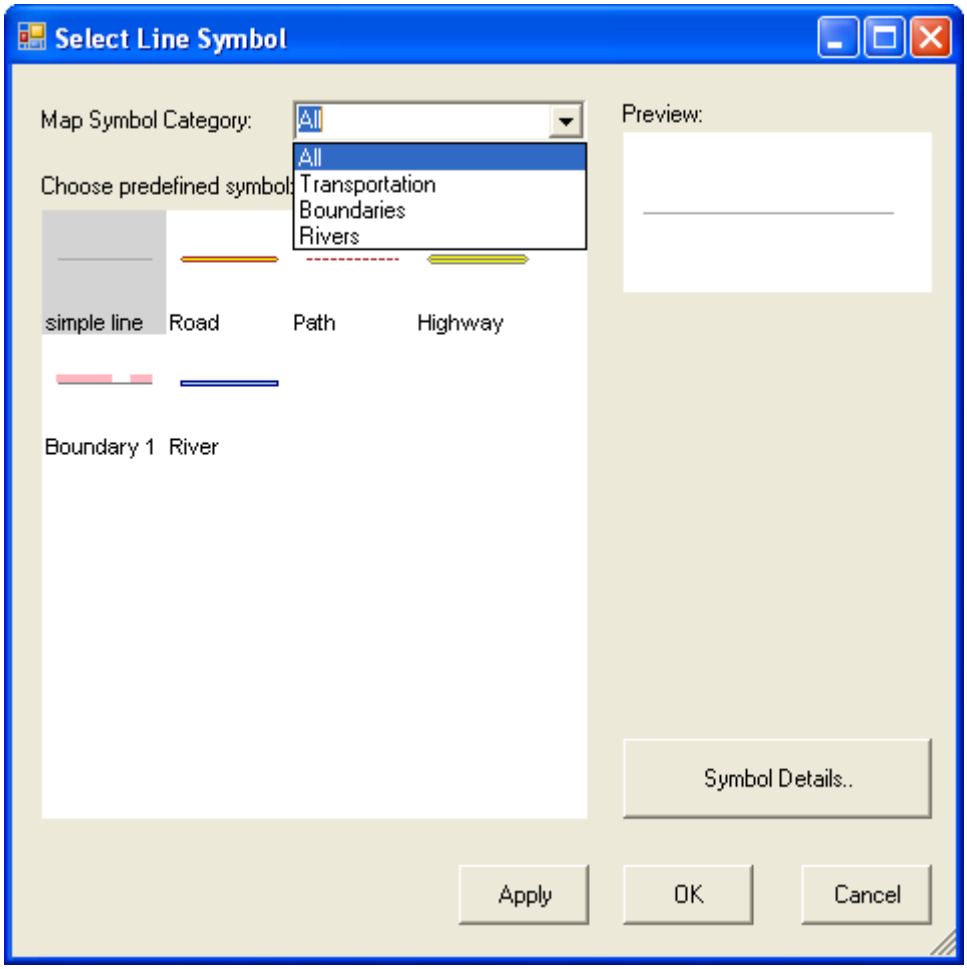


Figure 189: Select Line Symbol Window

The user can also customize the predefined symbol or create an original symbol by accessing the 'Line Symbol' dialog window. This is done by clicking the 'Symbol Details...' button or by double clicking the symbol in the 'Preview' box.

1.2.2. The Line Symbol Dialog

The 'Line Symbol' window is similar in nature to the 'Point Symbolizer Dialog' window. The *Symbol Layer* and *Symbolizer Characteristics* sections are present as well as 'Apply', 'Cancel' and 'OK' buttons that function the same as described in section 5.1.1.

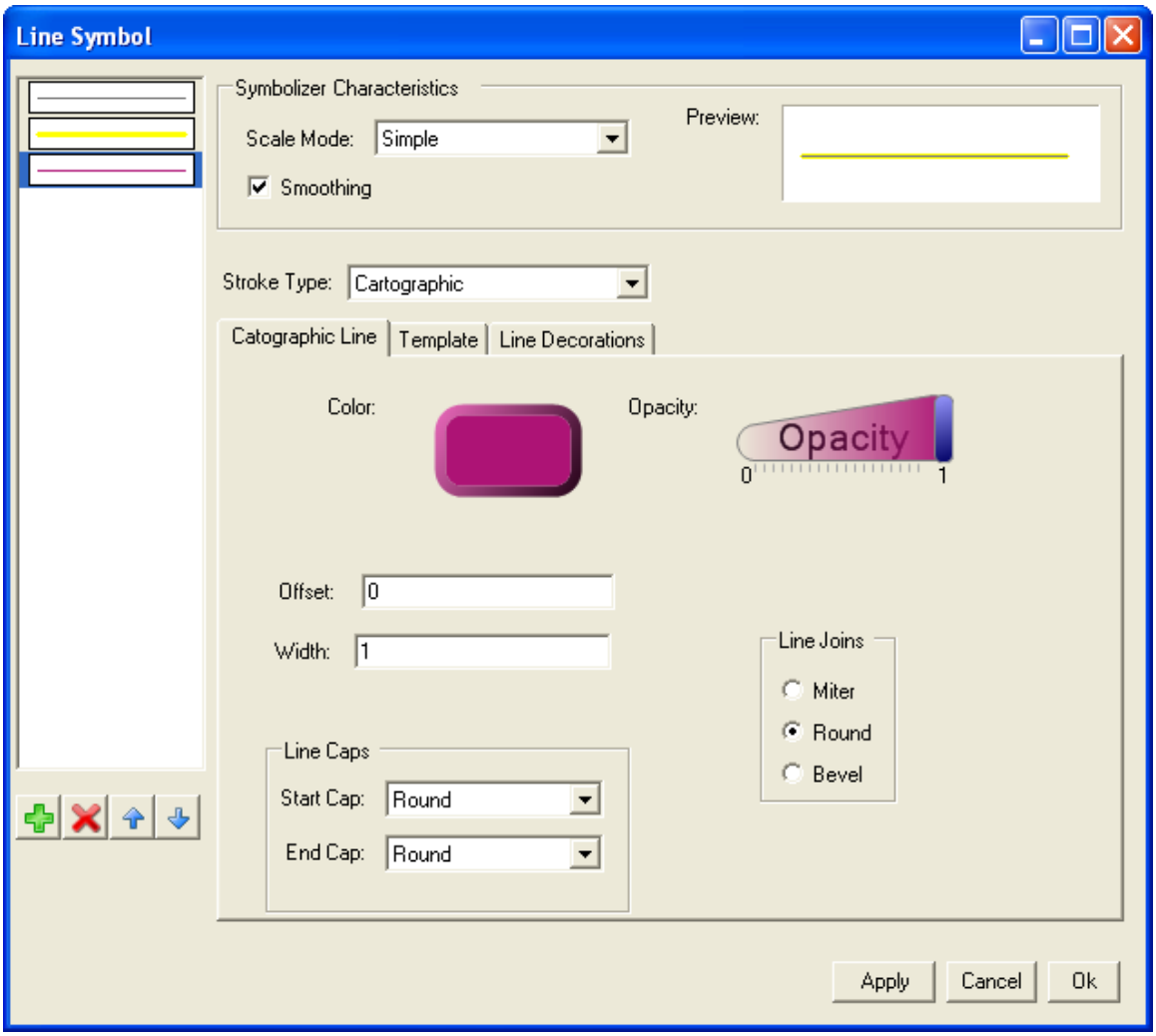


Figure 190: Line Symbol Window

The section unique to this dialog involves the 'Stroke Type' field. This drop down menu contains the options *Simple* and *Cartographic*. Choosing one of these options changes the appearance of the window and the number of options available to alter the symbol.

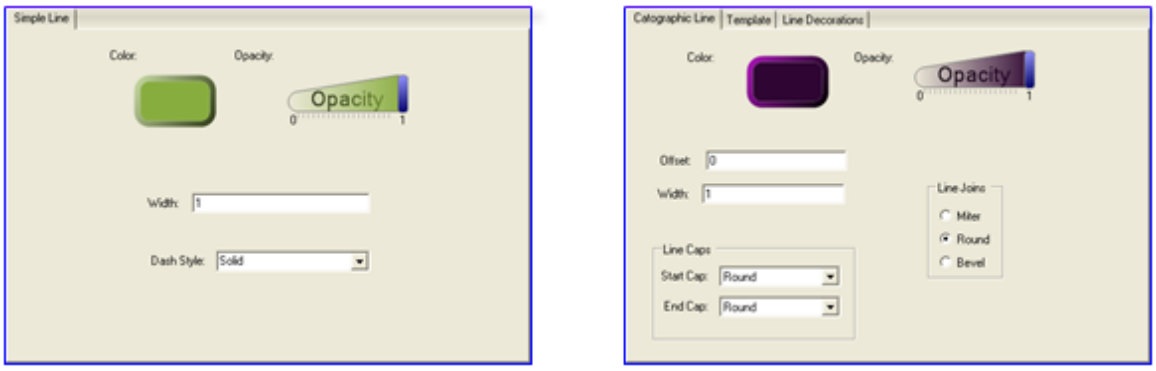


Figure 191: Stroke Type Options

- *Simple* – Clicking this option in the 'Stroke Type' drop down list changes this section to appear like the image on the left in figure 10. This option contains a color button and opacity slider for changing the color and transparency of the symbol. There is a 'Width' field where the user can type in a number to establish the point size of the line. Also there is a 'Dash Style' field with a drop down menu that contains preset options for altering the appearance of the symbol. The options are *Solid*, *Dash*, *dot*, *Dash dot* and *Dash dot dot*. There is also a *Custom* option which simply removes any styling and allows the user to set a custom style in the *Cartographic* section (see below).

- *Cartographic* – This option creates a series of three tabs each containing fields for customizing the appearance of the symbol. The tabs are *Cartographic Line*, *Template* and *Line Decorations*.

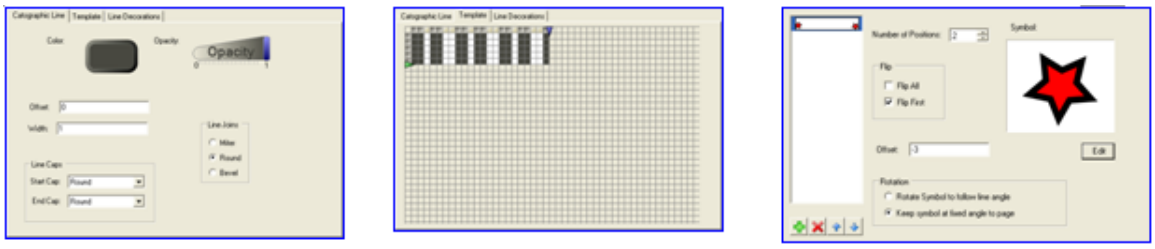


Figure 192: Cartographic Stroke Type Tabs

- *Cartographic Line* – This tab has the familiar color button and opacity slider. There are also 'Offset' and 'Width' fields where the user can enter a number to determine the width of the line and the extent to which the vertices of the line will be off set.

  The section 'Line Caps' establishes the appearance of the symbol's caps or ends. Through the two drop down menus the start cap and end cap can be set to *Flat*, *Square*, *Round*, *Triangle*, *NoAnchor*, *SquareAnchor*, *RoundAnchor*, *ArrowAnchor* and *AnchorMask*.

  There is also a *Custom* option which simply removes the cap. The user can then use the *Line Decorations* tab to set a custom cap (see below).

  Finally there is a 'Line Joins' section with three options for controlling how the line appears where the direction is changed. The options are *Miter*, *Round* and *Bevel* and are chosen by filling the adjacent bubble.

- *Template* – This tab contains a grid for designing a combination of dash and contour patterns for custom shapes. There is a green arrow on the vertical axis and a purple arrow on the horizontal axis both located as a default in the upper left corner. Dragging the green arrow down increases the width of the line. Dragging the purple arrow to the right provides space for inserting custom designs. Note that the top most row and the left most column is colored silver. Click individual boxes within these silver portions to toggle on and off the corresponding portions of the grid.

- *Line Decorations* – This tab essentially adds point symbols at user defined intervals throughout the line. Note that this tab has its own *Symbol Layers* box (see section 5.1.1). Click the add button  under the layer box to add an image to the 'Symbol' box.

  As a default a triangle with a randomly selected color is added to the ends of the line. Click the up and down arrows in the 'Number of Positions' field to establish the number of breaks in the line and thereby the number of points to be displayed.

  In the 'Flip' section there are two checkboxes to flip all of the points or just the first. As a default the first point is flipped. The 'Offset' field allows the user to set the points above or below the line symbol. Positive numbers place the point above the line and negative numbers place the point below.

  The 'Rotation' section contains two options for defining the way the points behave as the line changes direction on the map. These options are activated by filling the adjacent bubble. The 'Rotate Symbol to follow line angle' option causes the angle of the point to change as the direction of the line changes. The 'Keep symbol at fixed angle to page' option causes the angle of the point to remain static.

  Beneath the 'Symbol' box is the 'Edit' button. Clicking this button brings up the 'Point Symbolizer Dialog' window as described in section 5.1.1. Using this window, the line decorations can be changed according the same level of detail and customization available when handling point type data.

1.2.3. The Polygon Symbolizer Properties Dialog

Double clicking the indicator for a layer with polygon type data brings up the 'Polygon Symbolizer Properties' dialog window. This window is similar to the 'Point Symbolizer Dialog' window as described in section 5.1.1. For information regarding the *Symbol Layers* box on the left of the window, as well as the *Symbolizer Characteristics* section and the 'Apply', 'Cancel' and 'OK' buttons refer to that section. The 'Outline' section has also been dealt with previously with exception to one item which will be explained below.

The unique section to this window is the 'Pattern Type' section. The appearance of this section is determined by the options in the 'Pattern Type' drop down menu. Click the arrow to the right of this field to view the options in this menu. They are *Gradient, Picture, Simple* and *Hatch*.

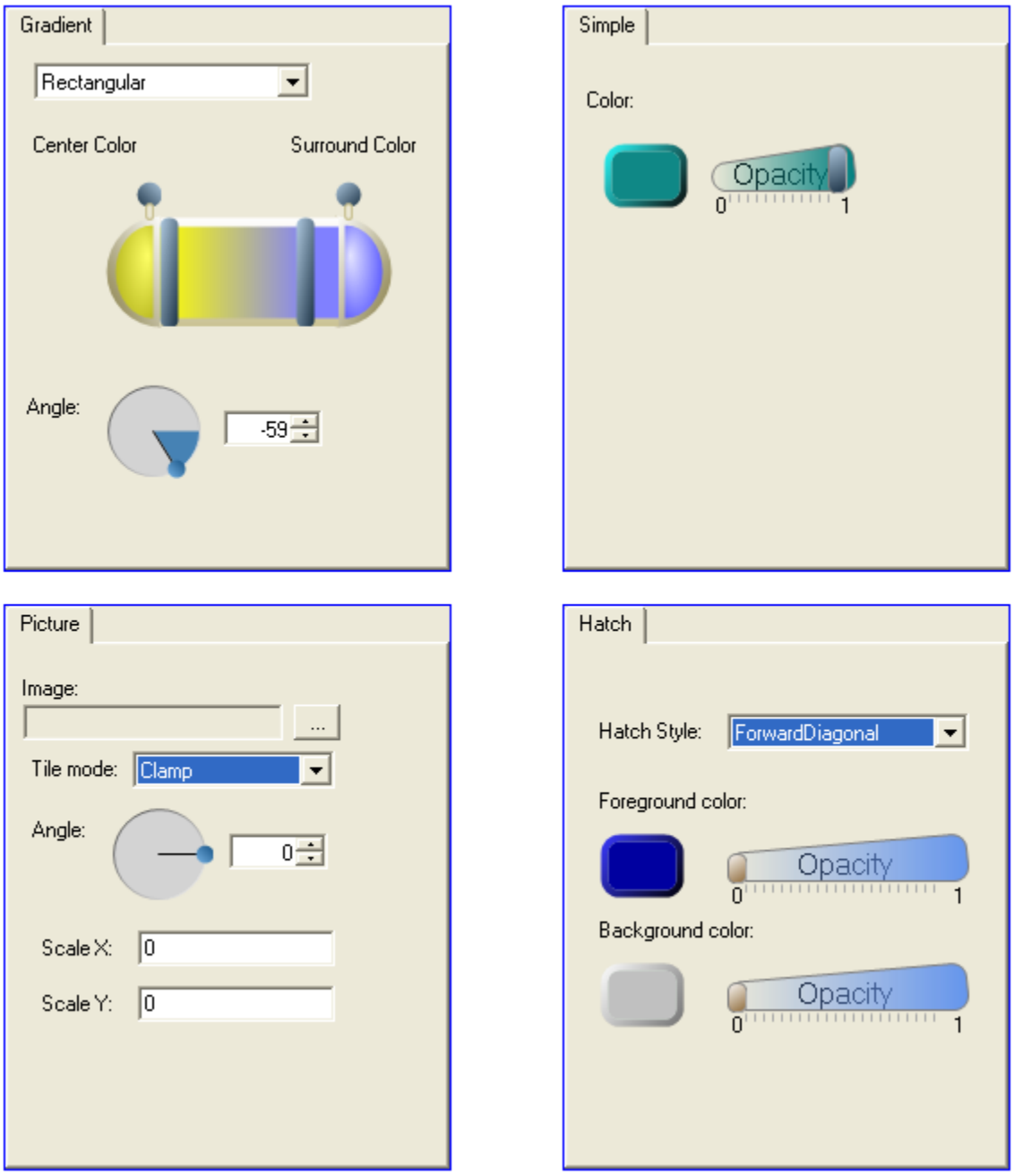


Figure 193: Pattern Type Options

Gradient Polygon Pattern Type

This pattern type creates a gradual blending of two colors. Three different kinds of gradients can be chosen in the drop down menu at the top of this section. The options are:

- *Circular* – The colors blend into a circle in the middle of the polygon.
- *Linear* – The colors blend toward a line drawn through the polygon. The angle of the line is determined with the 'Angle' meter.
- *Rectangular* – The colors blend into a rectangle in the middle of the polygon. The angle of the rectangle is determined with the 'Angle' meter.

The colors of the gradient are established using the gradient coloring tool. This tool is much like the color buttons and opacity sliders of previous sections combined into one tool. On the left is the center or start color and on the right is the surround or end color (depending on the gradient style). The two colors are joined into one capsule. On each end is a cap or button that acts as the color button. Each cap has a knob attached that acts as the opacity slider. Click the color button at each end to add the two different colors and slide the knobs to determine the transparency of each color individually. In the middle of the capsule are two sliders that establish the proportion of each color within the gradient. It is really cool! Go on, give it a try!

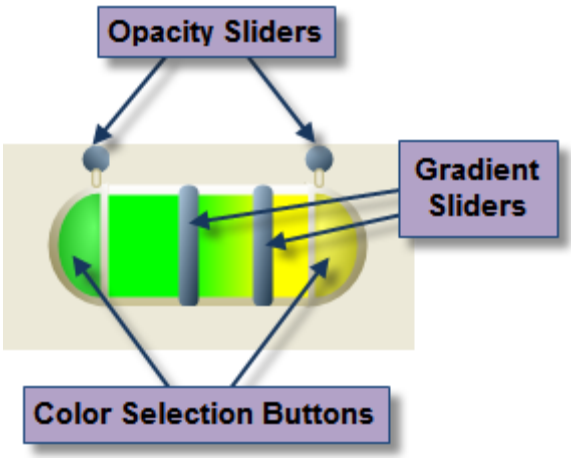


Figure 194: Gradient Coloring Tool

Picture Polygon Pattern Type

This pattern type allows the user to import an image as the polygon's symbol. Click the ellipse button ... to navigate to the desired image. Use the 'Tile mode' drop down list to establish the behavior of the image. *Clamp* indicates that only a single copy of the image will be displayed. *Tile* adds a tiling effect to the image. *TileFlipX*, *TileFlipXY* and *TileFlipY* maintain the tiling effect and flip the image repeatedly along the x or y axis or both accordingly. The 'Angle' tool is provided to allow the user to customize the angle of the image. Also the 'Scale X' and Scale Y' fields are available to allow the user to change the size of the image as it relates to the scale of the map.

Simple Polygon Pattern Type

This pattern type simply provides a color button and opacity slider to change the color of the polygon.

Hatch Polygon Pattern type

This pattern type provides the user with a large number of options to add stylized effects to the polygon such as diagonal lines, dots, grids, weaves and much more. These

options are accessed with the ‘Hatch Style’ drop down menu. There are too many to list. As well as the styles the user can select the foreground and background colors with their respective color buttons and opacity sliders.

The figure below display a map of the United States where of the different pattern types have been applied to some of the states individually.

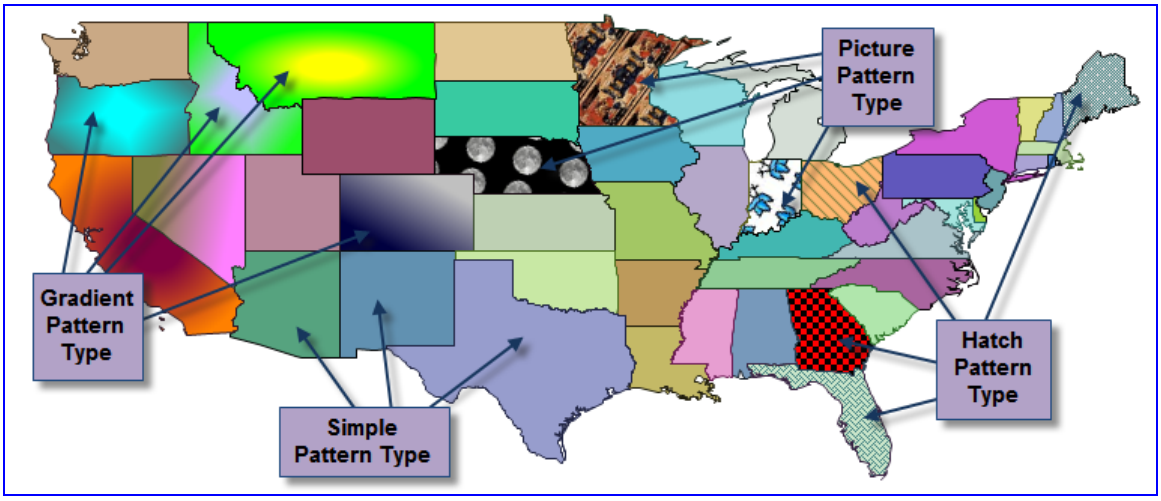


Figure 195: Polygon Pattern Types

Notice that when working with polygons the ‘Outline’ section of the symbolizer has an additional ‘Edit...’ button. Click this button to bring up ‘Line Symbol’ window as described in section 5.1.2. Doing this provides the user with same amount of customization and detail in regards to the outline of a polygon as is available when dealing with line symbols.

1.2.4. Raster Symbology

Right clicking the indicator of a layer with raster type data brings up the ‘CollectionPropertyGrid’ window. This window has two tabs that deal with different

aspects of the color scheme for the raster layer; the HLS (hue-saturation-lightness) tab and the RGB (red-green-blue) tab.

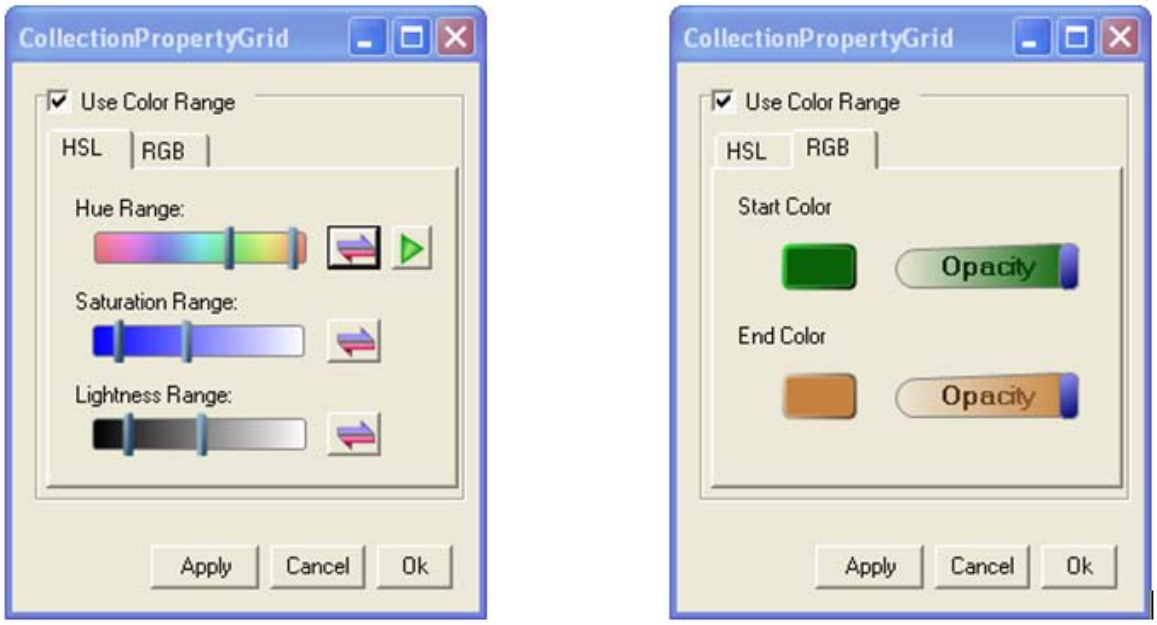


Figure 196: CollectionPropertyGrid Window

The RGB tab has two color tabs with accompanying opacity sliders that determine the start and end color of the color range. The HSL tab has three color range ramps one which determines the hue range on the color spectrum, another which determines the saturation range of the layer's color range, and a third with determines the layer's lightness range. Also of interest when dealing with the various color ranges is the flip button which reverses the color and range directionally. Note that the color schemes for all data types can be manipulated in more detail using the 'Layer Properties' window which will be discussed in another chapter.

1.3. Legend Context Menu

Right clicking on the layer name brings up the right click menu. This menu contains options for manipulating layers individually.

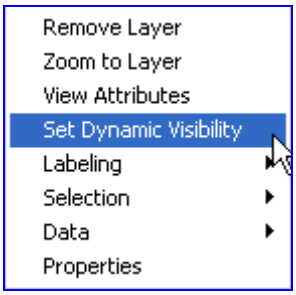


Figure 197: Right Click Menu

1.3.1. Remove Layer

Click this option to removes the layer from the *Legend Window* and the *Map Display*.

1.3.2. Zoom to Layer

This option zooms the map to the extent best suited for the layer.

1.3.3. View Attributes

Click this option to open the 'Attribute Table Editor' window. The 'Attribute Table Editor' contains both the attribute table for vector data and a series of tools for editing that table. The attribute table contains pertinent information regarding the selected feature. It is organized into rows and columns and can be used to locate and select features on the map. Clicking on the headings in the first row organizes that column is ascending or descending order (alphabetically or numerically depending on the data). Also clicking in the area to the right of each row highlights that particular row as well as the corresponding feature on the map.

The 'Attribute Table Editor' has its own main menu and a toolbar. With these editing tools the user can remove and add columns to the table and query the table to find certain data. (Note that this window can also be accessed by highlighting the layer in the *Legend Window* and clicking the attributes button in the *Toolbar*.) Be aware that this editor is still under construction.

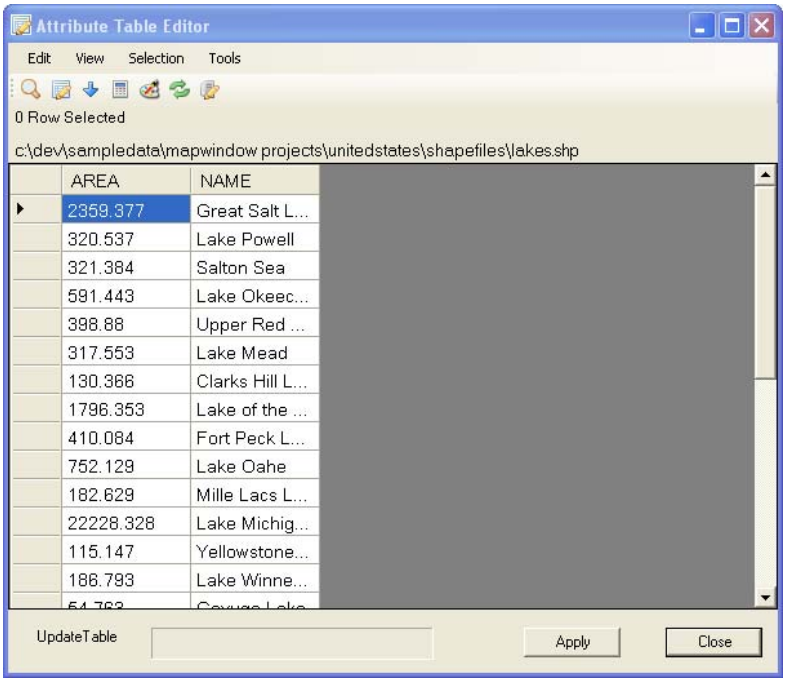
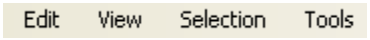


Figure 198: Attribute Table Editor

1.3.4. Attribute Table Editor

Menu



The menu selections consist of four heading: *Edit*, *View*, *Selection* and *Tools*. Again, this menu is a work in progress and still subject to change. This menus functionality will be expanded in the future and will be addressed at greater length at that time.

Toolbar

This toolbar consists of seven buttons that act as shortcuts to commonly used functions within the 'Attribute Table Editor.'

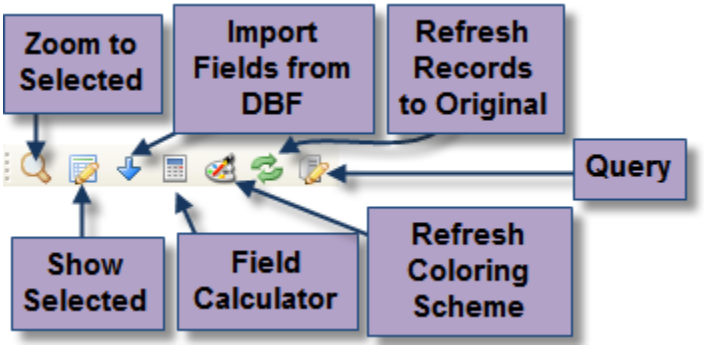


Figure 199: Attribute Table Editor Toolbar

Zoom to Selected

This button zooms the map view to the best extent to show features selected using the 'Select' tool.

Show Selected

Click this button to show the selected features on the map.

Import Fields from DBF

This button allows the user to navigate to current files to add the attribute table.

Field Calculator

This button opens the MapWindow field calculator.

Refresh Coloring Scheme

This button sets the coloring back to original.

Refresh Records to Original

Undoes any changes made to the attribute table and sets all records back to default.

Query

Clicking this button opens the 'Expression Editor.' This tool is used to create an expression that defines a search for a particular type of data within the attribute table. The 'Expression Editor' is present throughout MapWindow 6 so it is necessary to

describe it briefly. Creating an expression allows the user to search and discover data according to a layer's attributes and how those attributes relate to one another.

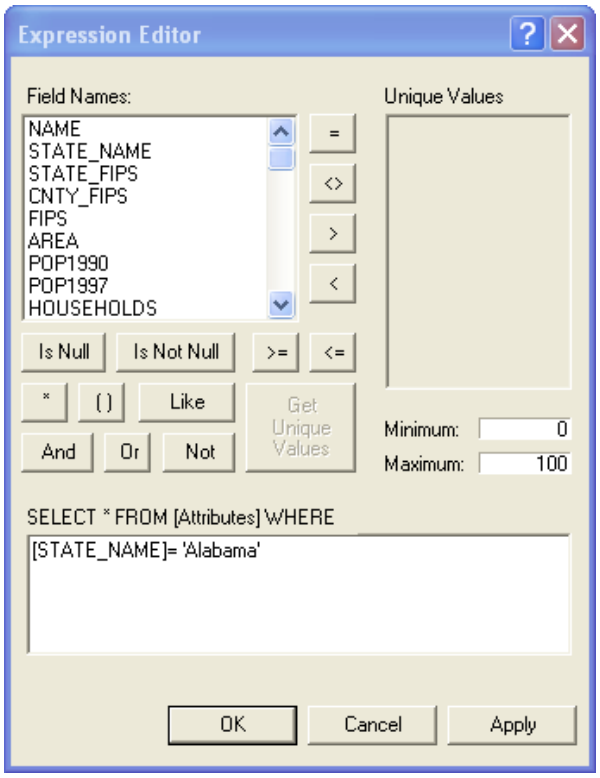


Figure 200: Expression Editor

On the left side of the editor is a box labeled 'Field Names.' This box contains all the available attributes for the selected layer. Once an attribute is selected from the field list it appears in the text box at the bottom of the window. (Note that an expression can be created manually by typing in the desired text in the text box.)

Once an attribute has been selected, the 'Get Unique Values' button is activated. Clicking that button populates the box to the right with values unique to that particular attribute. For example the state abbreviations attribute would have as its unique values all of the individual state name abbreviations.

From there the user can click any of the other buttons to create a relationship between those two attributes (greater than, or less than, or link the two attributes with 'And' or 'Or' etc.).

1.3.5. Set Dynamic Visibility

This option sets the dynamic visibility of the layer at the current scale on the map view.

1.3.6. Labeling

Point to this option to bring up the additional headings *Label Setup* and *Set Dynamic Visibility*. Clicking the *Set Dynamic Visibility* window performs the same function as above but does so where the layer's label is concerned as opposed to the layer itself. Clicking the option *Label Setup* opens up the 'Feature Labeler' window. This window contains four panels (*Expression*, *Basic Properties*, *Advanced Properties* and *Members*) that provide the user options for setting up the contents and positioning of labels as they would appear in regards to a chosen feature in the *Legend Window*. Click the tab name to access each panel. Note that at the bottom of the window, each tab contains a brief statement of its purpose and function. The 'Apply', 'Cancel' and 'OK' buttons' function is much the same as in previously described windows.

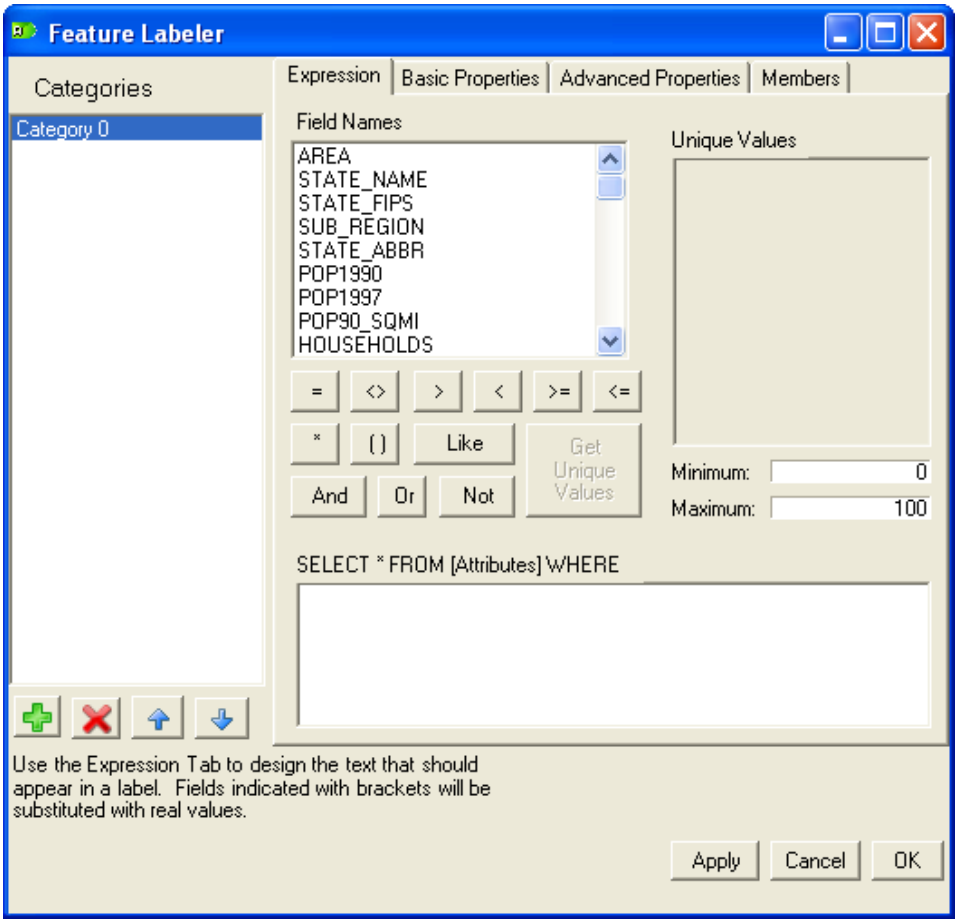


Figure 201: Feature Labeler Window

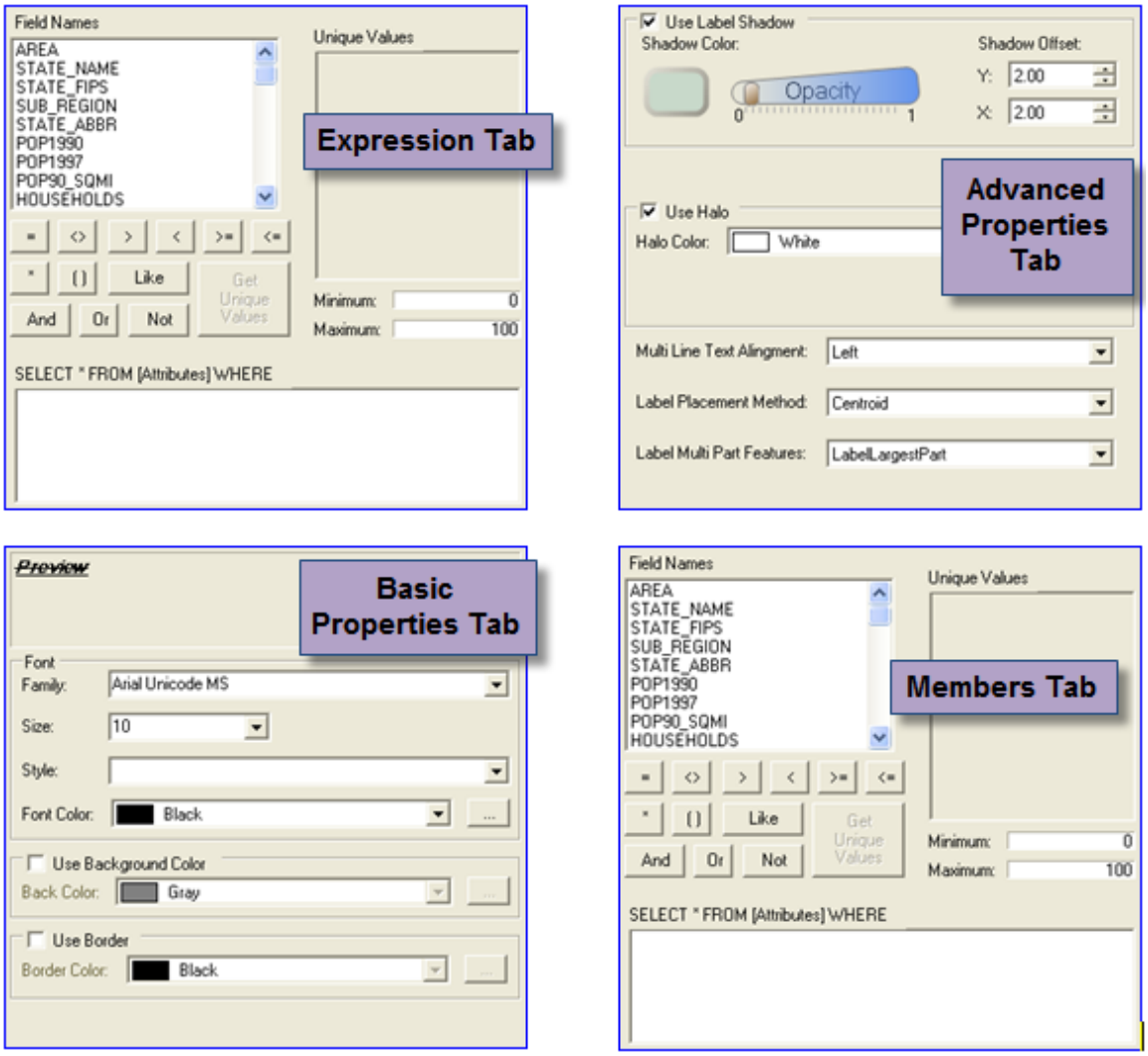


Figure 202: Feature Labeler Tabs

Expression Tab

This tab is used to design the label text.

Basic Properties Tab

This tab specifies the basic design characteristics of the label. This tab contains a 'Preview' box that allows the user to view any changes made to the label design before applying them. The font box contains options for manipulating the label font. Click the arrow to the right of the 'Family' field to access a drop down menu that contains multiple font options. The 'Size' field establishes the size of the font with preset options in the

drop down menu, or the user can click in the field and type in a desired font size. The 'Style' field allows the user to apply Bold, Italic, Underline, and Strikeout styles to the font either individually or in various combinations. The 'Font Color' field allows the user to change the font color either through a drop down menu listing color options or with the ellipse button  which accesses the 'Color' window as shown in figure 7. To create a background or border for the label, fill the 'Use Background Color' checkbox or the 'Use Border' checkbox and follow the same steps as above to determine their color.

Advanced Properties Tab

This tab specifies the advanced design characteristics of the label. Shadowing can be applied to the label with 'Use Label Shadow' checkbox. A color button and opacity slider is provided to change the shadow coloring. Also fields are available to offset the shadow relative the x and y axis positioning of the label. A halo can be added to the label with the 'Use Halo' checkbox and its color can be determined using the same method as described in the *Basic Properties* tab.

The 'Multi Line Text Alignment' field is provided to determine the alignment (left, right or center) of the text of the label. 'The 'Label Placement Method' field provides the options *Center* and *Centroid* for label placement. In regards to features that have multiple parts, the 'Label Multi Part Features' field allows the user to label the largest part or all parts of the feature.

Members Tab

This tab is used to create a filter expression that restricts which features will be assigned to a specific label.

1.3.7. Selection

Pointing to 'Selection' in the right click menu provides additional options in regards to features that have been or will be highlighted using the 'Select' tool.

- *Zoom to Selected Features* – Zooms the map view to the best extent to show features selected using the 'Select' tool.
- *Make this the Only Selectable Layer* – Indicates that the current layer in use will be the only layer that can be selected with the 'Select' tool.
- *Create Layer from Selected Features* – Creates a new layer in the *Legend Window* that comprises the currently selected features and their attributes.

1.3.8. Data

Pointing to 'Data' in the right click menu shows the option to 'Export Data' to a file folder on the user's computer. Click this option to open the 'Export Feature Data' dialog window. In this window, the 'Export' field gives the user the choice to export all of the features in the *Legend Window*, just the selected features, or just the features that can be seen in the current map extent. The 'Output shapefile or feature class' field determines the location on the computer for the exported data by using the browse  button. Click the 'Cancel' button to abort the activity or 'OK' to confirm.

1.3.9. Properties

Click this option to open the 'Layer Properties' window. This window provides a great deal of functionality when dealing with a particular layer's properties. As such, it deserves its own section.

1.4. Layer Properties Dialog

The 'Layer Properties' window is especially useful when dealing with layers that have multiple features. It can be accessed either through the right click menu or by double clicking to the right of the layer name in the *Legend Window*. The window is divided into four basic portions: *Coloring*, *Values*, *Statistics and Graphing*, and one portion that is unique to the data type in question (point and line, polygon, and raster).

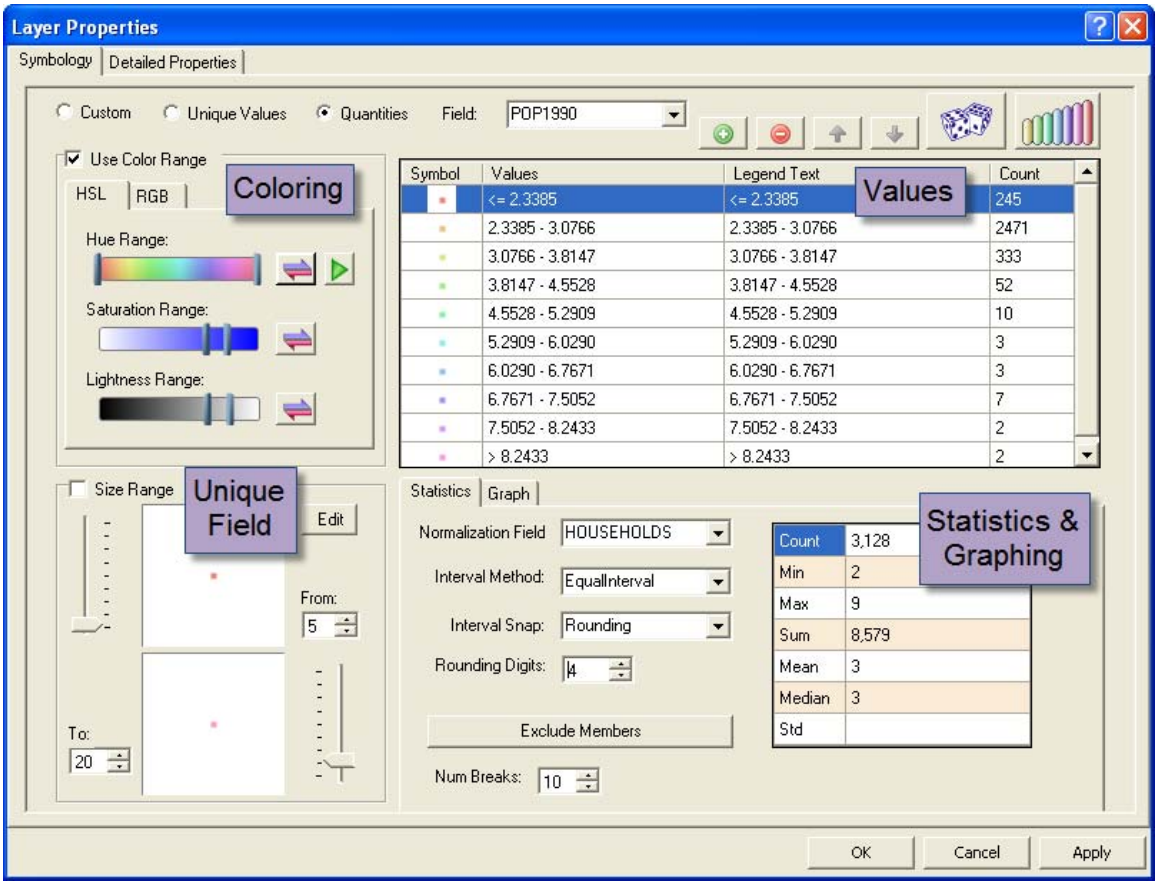


Figure 203: Layer Properties Window

Note that, located at the top left hand corner of the window, are two tabs one labeled 'Symbology' the other 'Detailed Properties.' The 'Detailed Properties' tab is a text driven listing of the layer properties similar to that found in other GIS programs

including MapWindow 4.x. It has limited functionality regarding layer manipulation which is dealt with extensively and with a much more user friendly presentation in the ‘Symbology’ tab. As such this document will not deal with the ‘Detailed Properties’ tab and focus solely on the ‘Symbology’ form.

1.4.1. Coloring

The *Coloring* portion is located in the top left quadrant of the window. It functions much the same as the CollectionPropertyGrid Window as described in section 5.1.4. This portion is used to create a color scheme when dealing with layers that have more than one feature or value.

1.4.2. Values

The *Values* portion consists of the large box which, when the window is first opened, takes up the right half of the form; as well as the strip of headings and toolbar icons that run the length of the window just under the tab buttons. This section is used to categorize the various attributes of a layer. Note that when dealing with raster data the *Values* options strip as shown in figure 23 is not present.

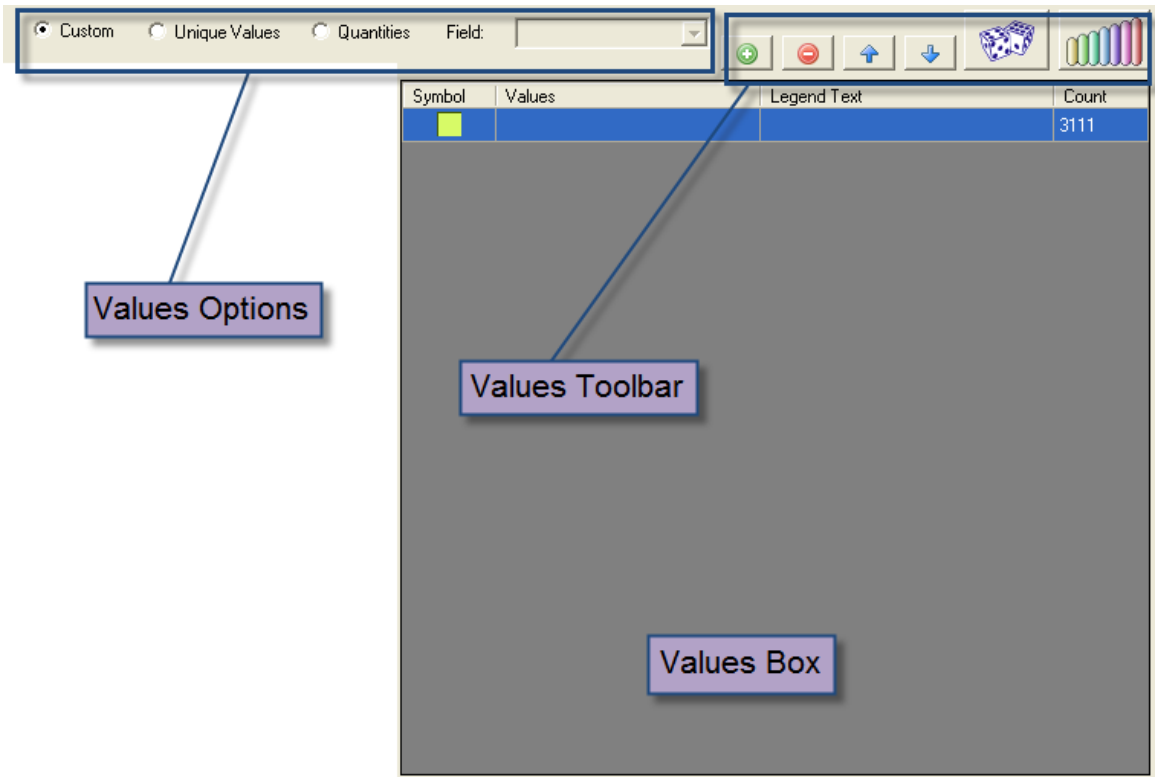


Figure 204: Values Portion of the Layer Properties Window

The *Values* options strip consists of three bubble headings that are called 'Custom,' 'Unique Values' and 'Quantities.' These headings provide the user with alternatives for categorizing a layer's attributes. Also present is a drop down menu entitled 'Field' which contains the various attributes by which a layer can be categorized.

- The 'Custom' bubble, when filled, allows the user to manually categorize the layer using the toolbar and the functionality assigned to the values box which will be discussed in a moment.

- The 'Unique Values' bubble allows the user to select an attribute from the 'Field' menu. Click the arrow to the right to access the drop down menu then click on the desired attribute. Once an attribute is selected the columns in the *Values* box are populated with the features pertaining to that attribute.

The ‘Symbol’ column contains a randomly generated coloring scheme that is bound by the settings in the *Coloring* portion to the left. Each feature in the layer is listed in the ‘Value’ column and the name under which the features will appear in the *Legend Window* is listed in the ‘Legend Text’ column. The ‘Count’ column displays the number of features being symbolized and the number of features that pertain to that particular value.

Within the value box itself some functionality has been provided. Double click the color indicator in the ‘Symbol’ column to alter the symbology of that feature. Returning to the ‘Custom’ bubble allows the user to manually customize each listing in the ‘Values’ column. Double click on a feature in the ‘Values’ column to bring up the ‘Expression Editor’ to create an expression to define that feature (i.e. [STATE_NAME] = Idaho). Double click on a feature in the ‘Legend Text’ column to change the feature name as it will appear in the *Legend Window*.

- The ‘Quantities’ bubble adds the *Statistics and Graphing* portion to the lower right hand of the window. As with the ‘Unique Values’ bubble, the user chooses an attribute from the ‘Field’ menu. The difference being that the only attributes available are those of a quantitative nature (i.e. populations, elevations, area sizes and lengths, etc.) This will be addressed further in the *Statistics and Graphing* section.

The *Values* toolbar consists of six buttons that access functions used in customizing the values and symbology of the layer’s attribute features.

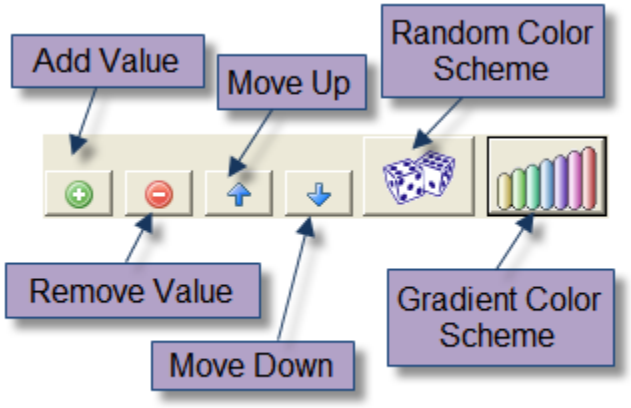


Figure 205: Layer Properties Values Toolbar

- The 'Add Value' button behaves differently depending on which type of categorization is being used. When the 'Custom' bubble is filled this button opens the 'Expression Editor' (see figure 24) allowing the user to enter an expression to define the new value which is added to the list in the 'Values' column. When the 'Unique Values' bubble is filled this button remains inactive. When the 'Quantities' bubble is filled this button adds a new value to the list in the values which is populated with information that fits into the current classification scheme established in the *Statistics and Graphing* section below.

- The 'Remove Value' button removes the currently highlighted feature from the list of values.

- The 'Move Up' button moves the currently highlighted feature up the list of values.

- The 'Move Down' button moves the currently highlighted feature down the list of values.

- The ‘Random Color Scheme’ button generates a new random color scheme bound by the settings in the *Coloring* portion of the window.

- The ‘Gradient Color Scheme’ button updates the gradient color scheme when changes are made to the settings in the *Coloring* portion.

Note that layers with raster data have an additional button not found in the toolbar for other data types. The lightening button access a menu with default options for raster data coloring schemes.

1.4.3. Statistics and Graphing

The *Statistics and Graphing* portion of the ‘Layer Properties’ window appears when the user chooses the ‘Quantities’ method of categorizing the layer’s attributes. This portion provides the user with options to manipulate the classification of the attribute in question (i.e. the number of group, the size of each group and so on). This section has two different displays that are accessed by two tabs entitled ‘Statistics’ and ‘Graph.’

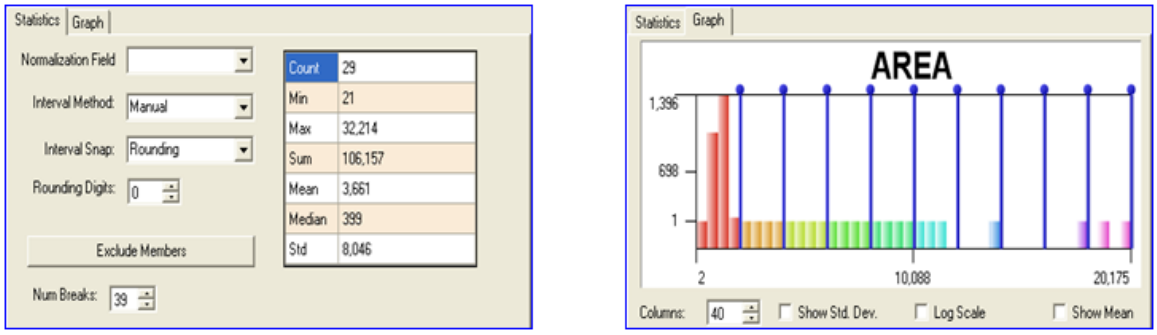


Figure 206: Statistics and Graphing Portion of the Layer Properties Window

1.4.4. Statistics

The ‘Statistics’ display consists of a series of menus and fields on the left hand side and a statistics table on the right. The table contains a count of the number of values for that attribute, the minimum value and the maximum value, the sum of the values, the mean and median values, and the standard deviation.

The drop down menu labeled ‘Normalization Field’ allows the user to choose an attribute by which to divide the current attribute in question. For example, when dealing with the number of households by county it may be beneficial to classify that data according to the overall population. The user would then choose the population attribute in this drop down menu.

The ‘Interval Method’ menu provides the user with three alternatives for classifying the attribute. The *Equalinterval* method creates classes of equal value ranges. If the range of values is 1-100 with 4 set as the number of breaks (intervals), this method will create groups from 1-25, 26-50, 51-75, and 76-100. The *Manual* method allows the user to set group breaks as they see fit. The *Quantile* method creates groups containing equal numbers of features. If a layer has 100 features divided into 5 groups, this method creates breaks so that 20 features fall into each group. The value range varies from group to group.

The ‘Interval Snap’ menu provides options for determining the formatting of the number display and the behavior of the breaks in regards to the data. *DataValue* makes the interval breaks snap or attach themselves to the nearest piece of data.

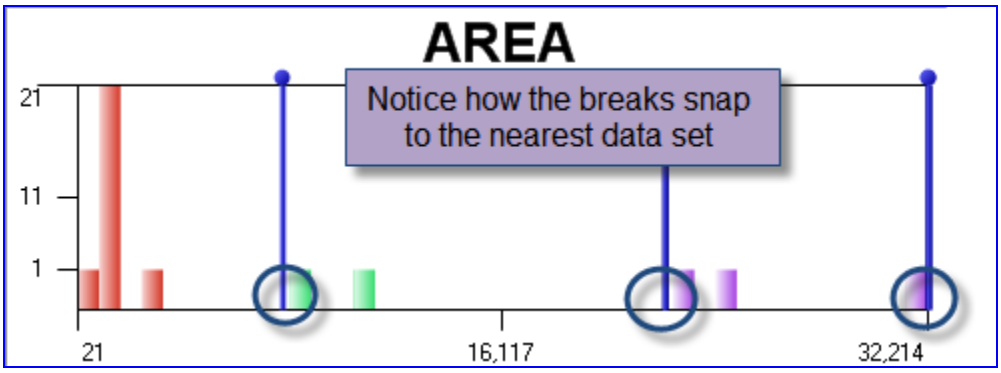


Figure 207: DataValue Method of Interval Snapping

None applies no snapping to the interval breaks.

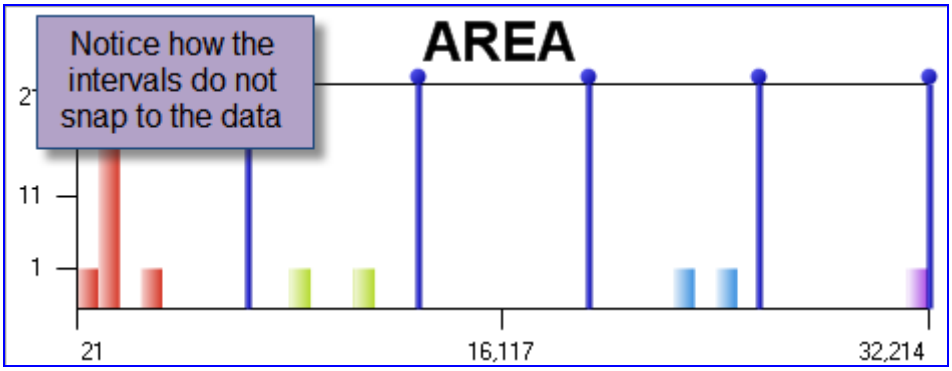


Figure 208: None Method of Interval Snapping

Rounding eliminates the decimal points in the number values and rounds them to the nearest whole number and sets the groups at equal intervals. While the *SignificantFigures* method set the breaks relative to pieces of data that hold some numerical significance such as to the nearest 10 or 100 or 1000.

That number is determined by the user with the following field entitled 'Significant Figures.' In this field the user can either type in the desired figure of significance or use the up and down arrows. The number in this field represents the number of digits displayed before the value is rounded.

The ‘Exclude Members’ button brings up the ‘Expression Editor’ with which the user can create an expression to determine values to discard from the current layer.

The ‘Num Breaks’ field allows the user to establish the number of breaks to apply to the groups.

1.4.5. Graphing

The ‘Graph’ tab displays the statistical data regarding the attribute and its values in graphical form. Each of the breaks is represented by a vertical blue line. These breaks can be individually manipulated by clicking and holding the mouse button over the blue line then dragging the line to the left or right to the desired position. Clicking on an area between two breaks highlights that set of data both in the graph and in the value box above. The ‘Columns’ field allows the user to increase or decrease the number of columns in the graph. The ‘Show Std. Dev.’ checkbox, when checked, displays the standard deviation range with a series of dotted vertical red lines. The ‘Log Scale’ checkbox, when checked, changes the graph from a linear representation of the data to a representation relative to the data log. The ‘Show Mean’ checkbox toggles on and off a dotted vertical blue line that represents the mean value within the data.

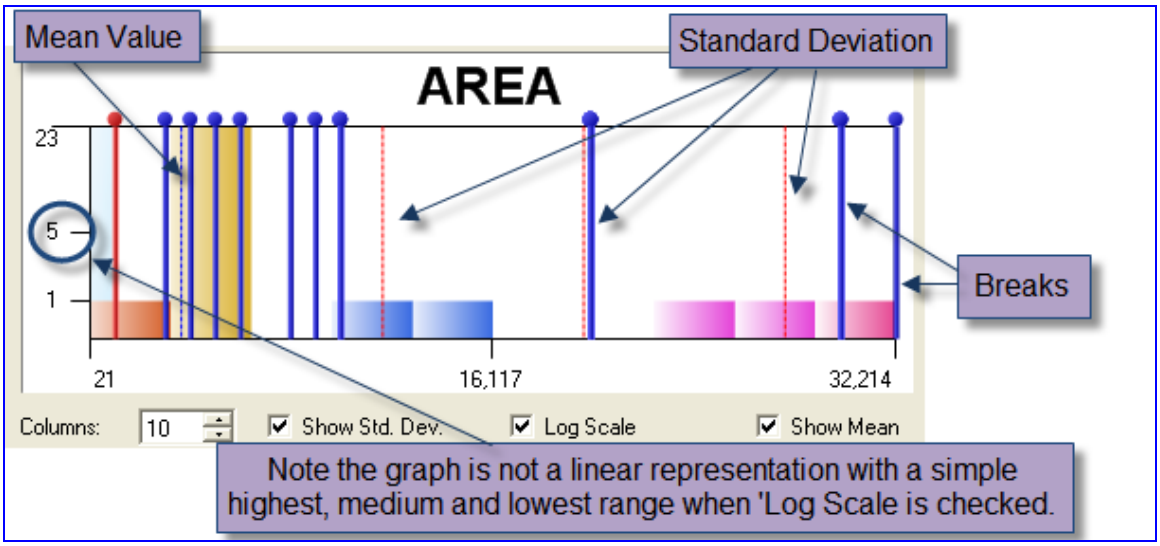


Figure 209: Graph Display

1.4.6. Unique Field

This section details the portion of the 'Layer Properties' window that is specific to different types of data.

Tools to create a size range are provided for point and line data types. This range allows the user to create a gradual size increase or decrease in the layer's symbology relative to an increase and decrease in the layer's values. The range is created by entering a start value in the 'From' field. This can be done either with the meter to the left or with the up and down field to the right. Notice that a visual representation of the point or line allows the user to visualize the size before applying the change. Once the start value is entered, an end value is added to the 'To' field in the same manner. For example, when plotting cities on a map the user may wish to represent cities with larger populations with larger points; and small cities with smaller points.

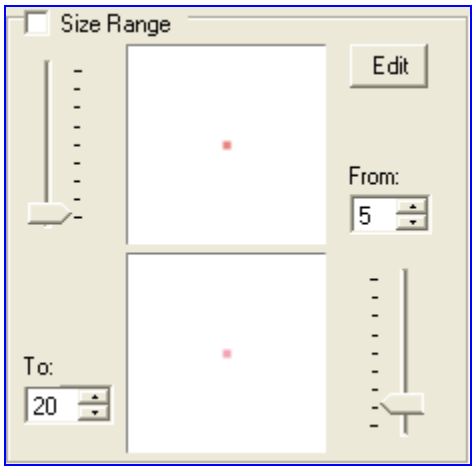


Figure 210: Unique Properties Field for Point and Line Data Types

A gradient tool is provided for layers with polygon type data. The user checks the 'Use Gradients' checkbox to apply a gradient pattern (see section 5.1.3) to the layer and then uses the meter or the up and down buttons or simply types in a desired angle at which to set the gradient.

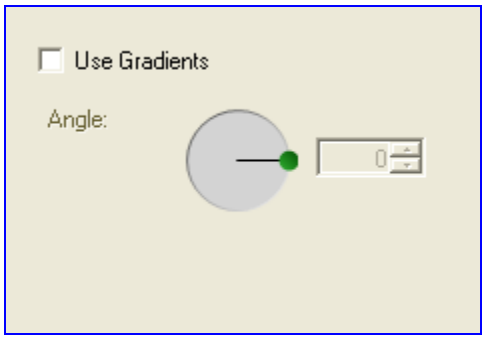


Figure 211: Unique Properties Field for Polygon Data Types

Layers with raster type data provide the user with tools to apply and manipulate hillshading. Check the 'Hillshade' checkbox to turn hillshading on. The figure below shows a raster with and without hillshading.

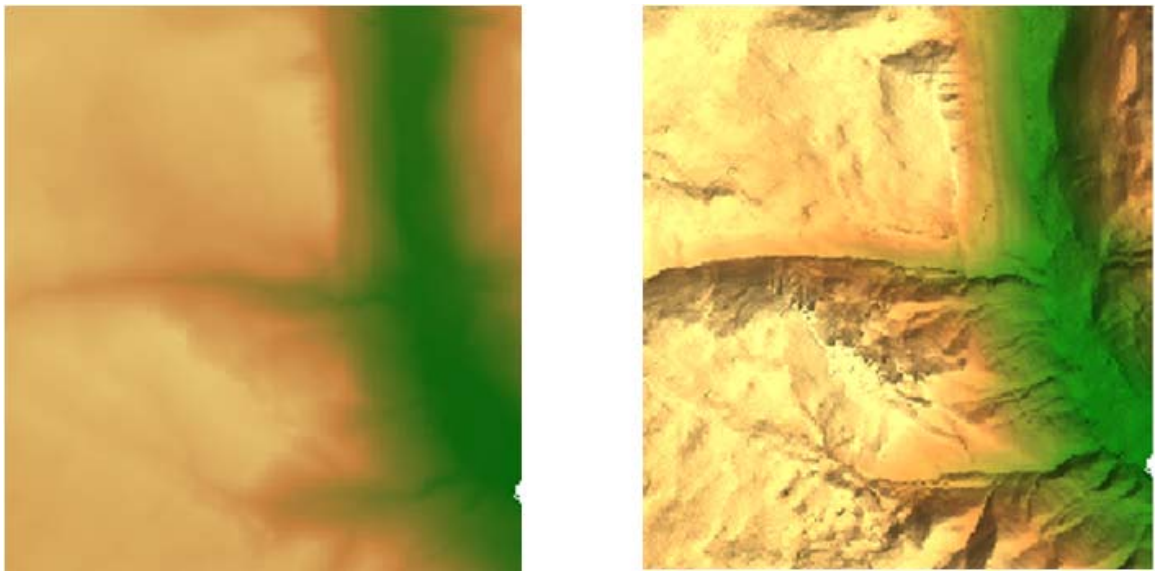


Figure 212: Raster with out HillShading on the left and with HillShading on the right

The 'Details' button opens a window that displays the text details of the hillshading effect. A meter and up and down field (similar to those found in the gradient field for polygon type data) are provided to determine the angle of the light effects.

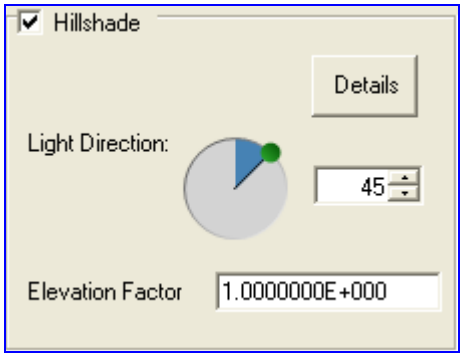
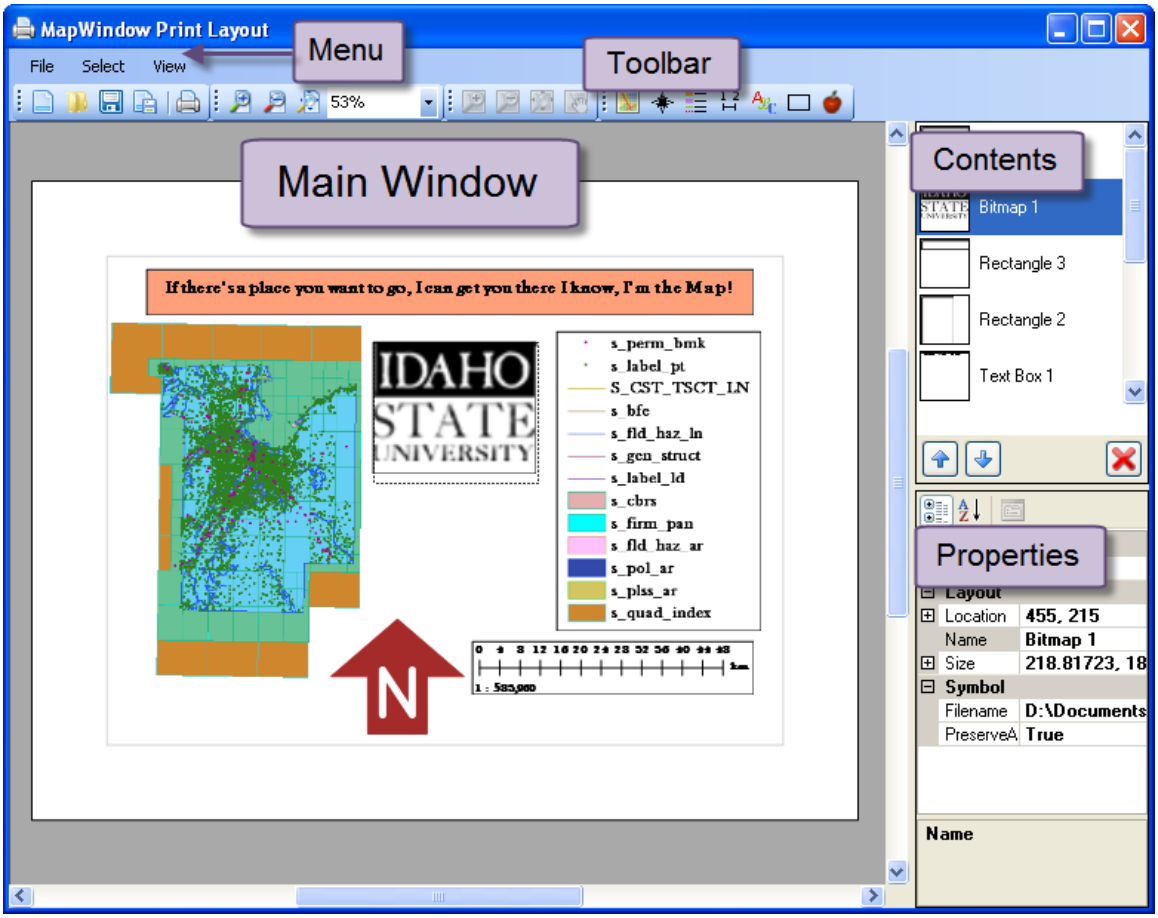


Figure 213: Unique Properties Field for Raster Data Types

The field 'Elevation Factor' is used to convert the elevation units (feet, meters, etc.) into the same units as the geospatial projection for the latitude and longitude values of the grid. For example, if the elevation units are in feet, this number should be set to 1.

1.5. Print Layout

To launch the interface click the print icon in the MapWindow GIS toolbar. This automatically opens the 'MapWindow Print Layout' window.



The 'MapWindow Print Layout' window has five primary parts: the *Menu*, the *Toolbar*, and the *Main*, *Contents* and *Properties* windows.

1.5.1. Print Layout Main Window

The user is provided a great deal of flexibility when dealing with the components of the Print Layout Main Window. The Main Window essentially displays the page layout as it will be printed. The user can arrange any desired components which include the map, the legend, a north arrow, a scale bar, a rectangle used for highlighting, as well

as any needed text. All of these features, however, are added to the Main Window and manipulated using the other parts of the Print Layout interface.

1.5.2. Print Layout Menu

The Print Layout Menu is located in the upper left corner of the window under the window title. It consists of the *File, Select* and *View* headings.

File

- **New –** opens a new print project. If a project is currently in use, a save dialogue opens if any changes have not been saved. (also applies to the *close* function)

- **Open –** opens the 'Load Print Layout' window to open any previously saved print projects.
- **Save –** saves any changes made to the current project.
- **Save as –** saves a project under a new name and location as an .mwl file type.

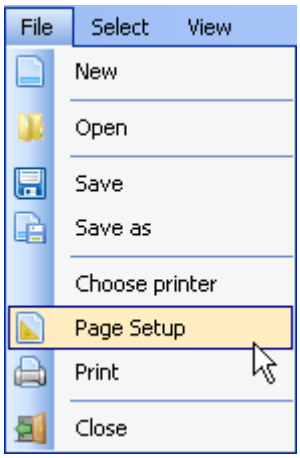


Figure 214: Print Layout File Menu

- **Page Setup –** opens the 'Page Setup' window that provides options to establish paper size and dimensions, paper orientation and top, bottom, left, and right

margins. The page dimensions are determined by the 'Paper Size' drop down box. Clicking the arrow to the right of the box brings up a list of predetermined page dimension options. The paper orientation options are 'Portrait' and 'Landscaped' and the user can toggle between the two by clicking the adjacent circles. All of the margins are set to a 1.00 default. Click the box corresponding to the desired margin to manually enter the desired size. (see the figure below)

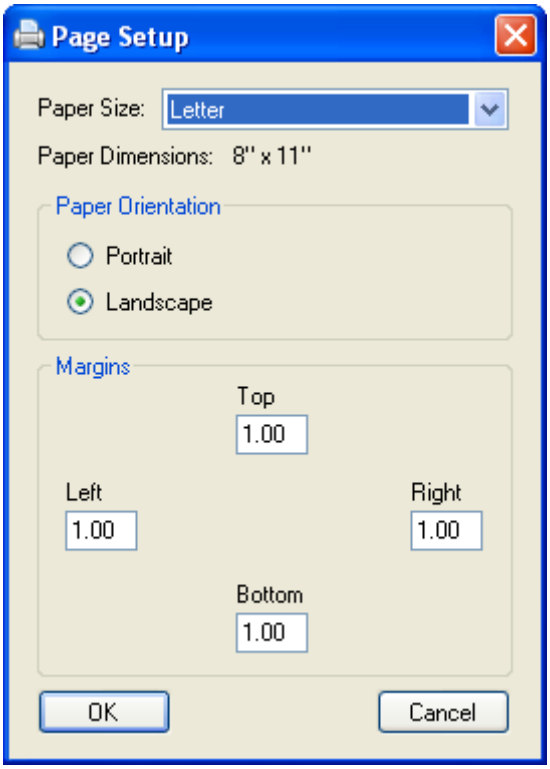


Figure 215: Print Layout Page Setup Window

- **Print –** prints a hard copy of the map layout.
- **Close –** closes the 'Print Layout' window.

Select

The *Select* menu contains functions for manipulating the various layers in a current project.

Figure 216: Print Layout Select Menu

- **All –** selects all of the layers in the *Main* and *Contents* windows.
- **None –** deselects any currently selected layers.
- **Invert –** sets current selections to the inverse. E.g. all selected layers are deselected and vice versa.
- **Move up –** moves the currently selected layer(s) up in the drawing order. (see the 'Print Layout Contents Window' section)
- **Move down –** moves the currently selected layer(s) down in the drawing order.
- **Delete –** deletes the currently selected layer(s).
- **Convert to bitmap –** converts the selected layer individually into a bitmap. Note: Any files of the file types .png, .jpg, .bmp, .gif, and .tif can be used as bitmap within the Print Layout interface.

View

The *View* menu provides viewing options in the main window and toolbars.

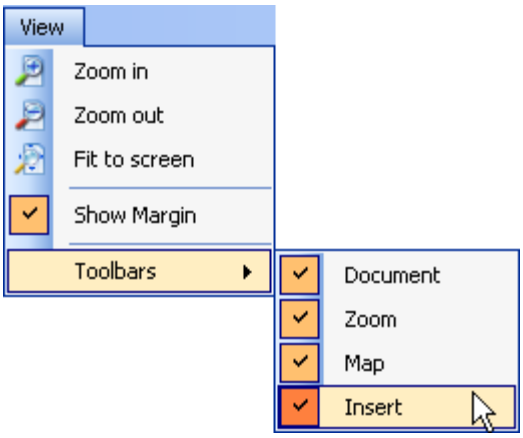
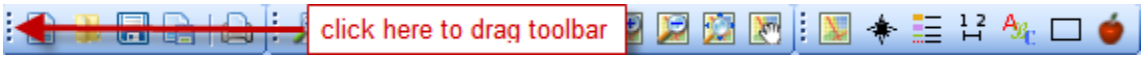


Figure 217: Print Layout View Menu

- **Zoom in –** zooms in around the center of the Main Window.
- **Zoom out –** zooms out around the center of the Main Window.
- **Fit to screen –** zooms the Main Window to the best possible extent to fit the screen.
- **Show Margin –** toggles on and off the margin view in the Main Window.
- **Toolbars –** pointing to the arrow to the right of this function brings up a list of the various toolbars. This provides the option to toggle on and off each toolbar individually. (see the 'Print Layout Toolbar' section)

1.5.3. Print Layout Toolbar

The Print Layout Toolbar is located under the Menu and consists of a series of icons whose functions will be described below. The Toolbar is made up of four individual toolbars. Clicking and holding the left mouse button while pointing at the four vertical dots to the left of each toolbar allows the user to drag the toolbar to a different position if desired. Each toolbar can be toggled off and on in the *View* menu. (See above) The four toolbars are titled as follows: the *Document*, *Zoom*, *Map*, and *Insert* toolbars.



Document Toolbar 

The *Document* toolbar consists of icons that provide shortcuts to the options contained in the *File* menu.

- **New –** opens a new print project.
- **Open –** opens the 'Load Print Layout' window to open any previously saved print projects.
- **Save –** saves any changes made to the current project.
- **Print –** prints a hard copy of the map layout.

Zoom Toolbar 

The *Zoom* toolbar consists of icons that provide shortcuts to the options contained in the *View* menu.

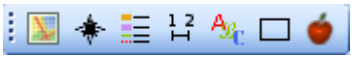
-  **Zoom in –** zooms in around the center of the Main Window.
-  **Zoom out –** zooms out around the center of the Main Window.
-  **Fit to screen –** zooms the Main Window to the best possible extent to fit the screen.
- 

Zoom percentage – through a drop down menu provides preset percentage values relative to the actual paper size at which to display the image in the main window.

Map Toolbar 

The *Map* toolbar is only active when the map layer is selected. This toolbar provides a series of view options pertaining only to the map layer.

-  **Zoom Map in –** zooms in the map insert around the center of the map.
-  **Zoom Map out –** zooms out the map insert around the center of the map.
-  **Zoom Map full extent of layers –** zooms the map insert to show the full of extents of the layers on the map.
-  **Pan Map –** changes the mouse arrow to a hand that allows the user to grab and move the map in a desired direction. Click and hold the left mouse button while pointing at the map to drag the map. Release the mouse button to release the map.

Insert Toolbar 

The *Insert* toolbar consists of icons that are used to place key map components into the main window. Each icon changes the mouse arrow to a crosshair. Click and drag the crosshair to create a window in which to view the specified feature.

-  **Insert Map –** creates a window in which to display the current MapWindow map view.
-  **Insert North Arrow** – creates a window to display a north arrow.
-  **Insert Legend –** creates a window to display the MapWindow legend for the current map view.
-  **Insert Scale Bar –** creates a window to display a scale bar featuring the current MapWindow scale.
-  **Insert Text –** creates a box in which the user can place any desired text.
-  **Insert Rectangle –** creates a rectangle which can then be used to highlight and outline other layers.
-  **Insert Bitmap** – opens the 'Open' window that allows the user to select a compatible file to insert as a bitmap. Upon opening the file, the mouse pointer is converted to the familiar crosshairs to draw a window in which to insert the bitmap. (see the 'Select>Convert to Bitmap' section above for appropriate file types)

1.5.4. Print Layout Contents Window

The Print Layout Contents Window lists the layers in use in the Main window. As layers (e.g. map, north arrow, legend, scale bar, text, rectangle, and bitmap) are added to the Main window they are also added to the Contents window. Layers added first remain at the bottom of the list. Subsequent layers are organized in ascending order. It can be important to consider the drawing order of the layers. The layers are displayed in the Main window in a bottom up relationship with their order in the Contents window. Therefore, a layer at the bottom of the list can be entirely obscured by a layer at the top of the list. Consequently, the order of the layers needs to be carefully considered so that no desired features are obscured.

Changing the order of the layers is simply done. Click and highlight the layer in the Contents window. Click the up  and down  arrows at the bottom of the window to move the highlighted layer up or down within the drawing order. The  button is also provided to remove the layer entirely. Note that layers can be highlighted individually or in groups by clicking and dragging the mouse over multiple layer names or by pressing and holding the Ctrl button on the keyboard and then clicking the desired layer names. These layers can then be moved up and down the drawing order as a group.

1.5.5. Print Layout Properties Window

The Print Layout Properties Window displays the properties of a highlighted layer (only when highlighted individually). These properties include but are not limited to name, size, color, font etc. As each layer type has a different set of properties to consider, they will all be described separately below. Note that the properties can be

organized categorically by clicking the ‘categorized’ button or alphabetically by clicking the ‘alphabetical’ button located in the upper left hand corner of the Properties window.

This document will consider the different properties windows as they appear when organized categorically. Each category heading has a + and – checkbox which can be clicked to expand or collapse its corresponding category.

1.5.6. Map Properties

The properties applicable to the map are listed in the *Behavior*, *Layout*, *Map* and *Misc.*

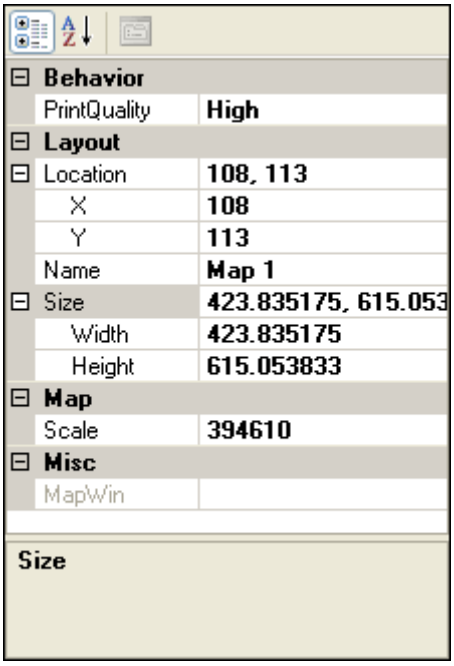


Figure 218: Print Layout Map Properties

- **Behavior –** contains the feature ‘PrintQuality’ which when clicked brings up a drop down arrow with a menu that can set the print quality to low, medium or high.

- **Layout –** contains the following features

- **Location –** displays the x and y coordinates of the map. Also the name of the map is displayed which, upon clicking access a drop down arrow which lists other available maps to be selected in lieu of the current map.
- **Size –** displays the width and height of the map.

- **Map –** indicates the current scale of the map.
- **Misc –** holds any miscellaneous information about the map when available.

1.5.7. North Arrow Properties

The only properties applicable to the north arrow are found in the *Symbol* category.

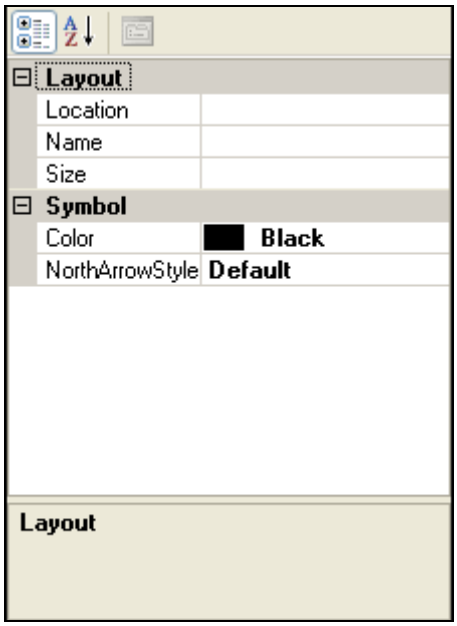


Figure 219: Print Layout North Arrow Properties

- **Symbol –** defines the appearance of the north arrow with the following properties:

- **Color –** clicking this feature brings up a drop down arrow that access a list of colors to choose from. The arrow is displayed in the desired color.

- **NorthArrowStyle –** through a drop down arrow menu, various preset types of arrows are made available to the user.

1.5.8. Legend Properties

Most of the properties applicable to the Legend are described in the 'Map Properties' section except for the *Font*, *Layers*, *NumColumns*, and *TextHint* categories.

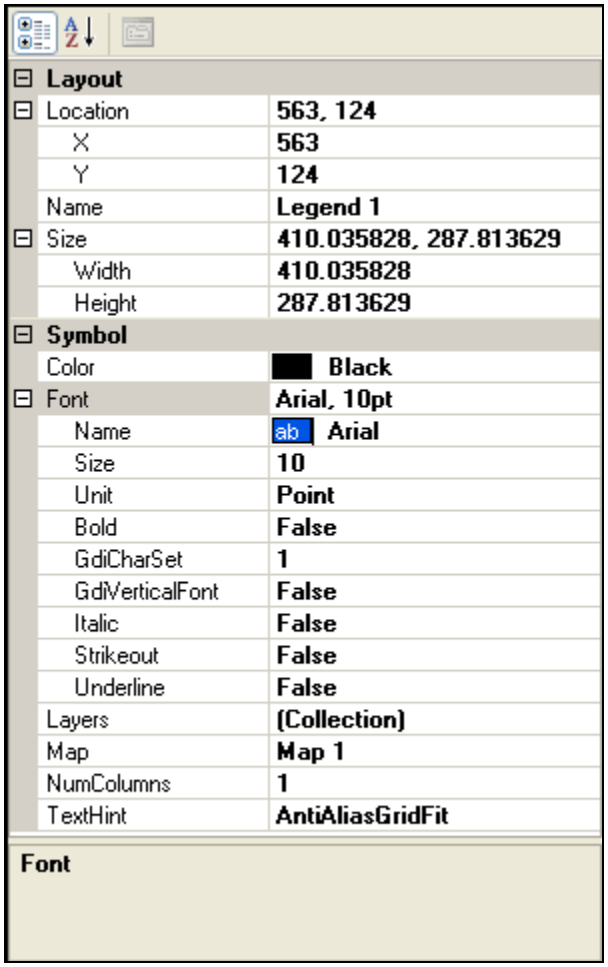


Figure 220: Print Layout Legend Properties

- **Font –** when collapsed, clicking this property brings up an ellipses  icon which brings up the ‘Font’ window which provides multiple options for manipulating the font of the headings within the Legend. Font options include font name, style, size and effects (see the image below). Note that when this category is expanded the same features can be managed from within the property window.

- **Layers –** clicking this property brings up a drop down arrow that access a list of the features in the legend that are displayed in the map. Each feature has an adjacent checkbox which, when checked, determines that the feature will be displayed in the map.

- **NumColumns –** determines the number of columns to be used to display the layers in the legend.

- **TextHint –** controls the smoothing around the edges of the fonts. Multiple options are provided through a drop down menu.

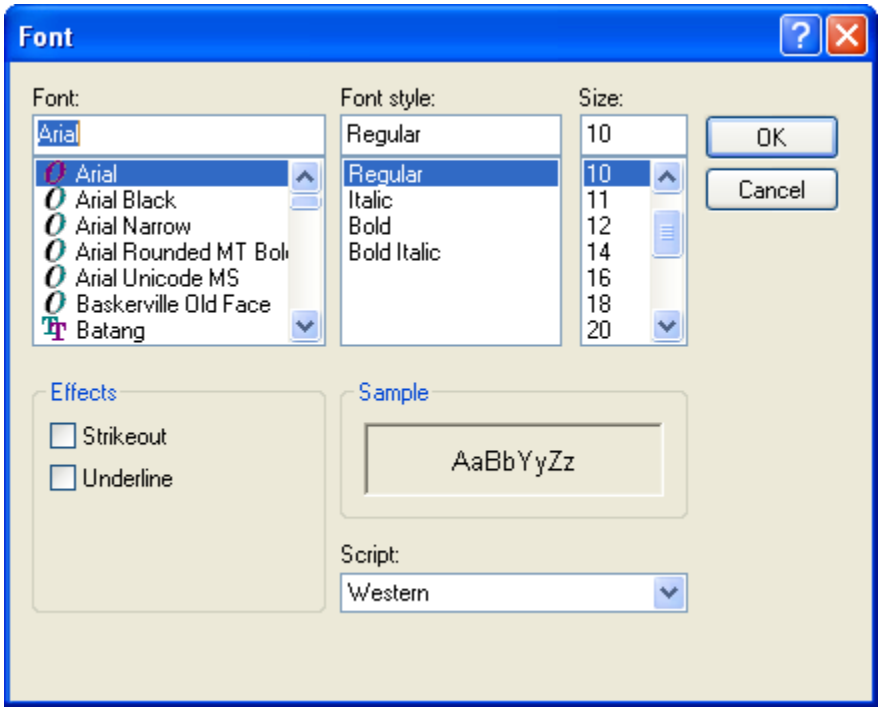


Figure 221: Print Layout Fonts Window

1.5.9. Scale Bar Properties

Properties that are exclusive to the scale bar are *BreakBeforeZero*, *NumberOfBreaks*, *Unit* and *UnitText*.

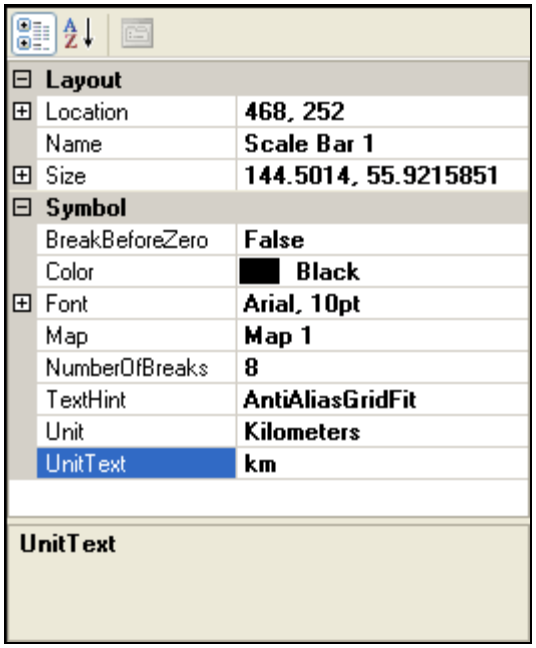


Figure 222: Print Layout Scale Bar Properties

- **BreakBeforeZero** – a Boolean or true/false property where, when set to true, an additional section is added to the scale bar representing values below zero.

- **NumberOfBreaks** – determines the number of sections into which the scale bar is divided.

- **Unit** – access a drop down arrow list from which to choose the unit of measurement for the scale. (E.g. kilometers, meters, inches, miles etc.)

- **UnitText** – provides a field to alter the text of the unit of measurement as it appears in the scale bar.

1.5.10. Text Box Properties

Properties that are exclusive to the text box are *ContentAlignment* and *Text.*

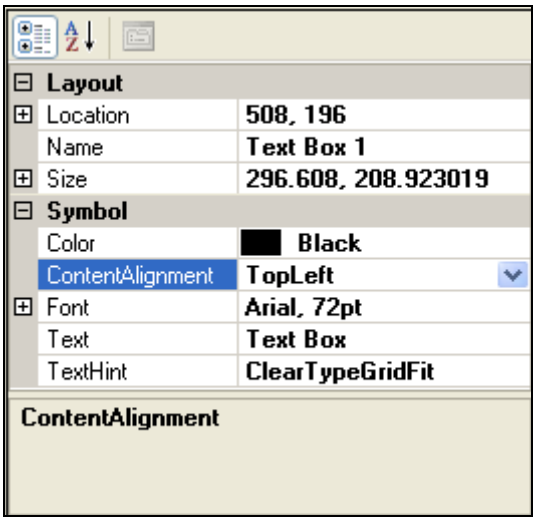


Figure 223: Print Layout Text Box Properties

- **ContentAlignment –** indicates the alignment of the text within the text box. Click the property to access a drop down arrow menu with different alignment options. (E.g. top left, middle center, bottom right etc.)

- **Text –** provides a place for the user to enter desired text.

1.5.11. Rectangle Properties

A rectangle is drawn to outline or highlight other features. Its exclusive properties are as follows:

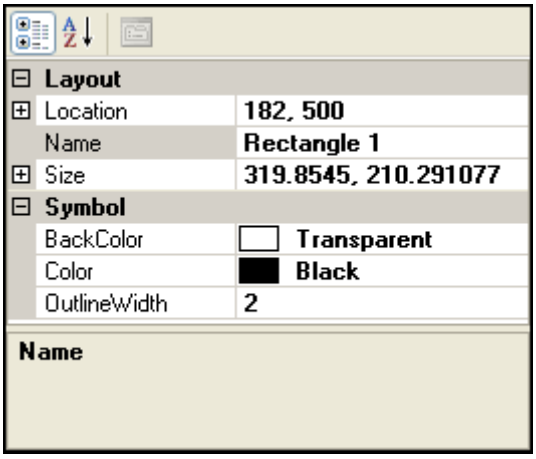


Figure 224: Print Layout Rectangle Properties

- **BackColor –** determines the filler color of the rectangle. A list of colors is available via a drop down arrow menu.

- **Color –** in the case of the rectangle the color refers to the color of the outline of the rectangle.

- **OutlineWidth –** determines the thickness of the outline.

1.5.12. Bitmap Properties

Properties that are pertinent to an inserted bitmap are *Filename* and *PreserveAspectRatio*.

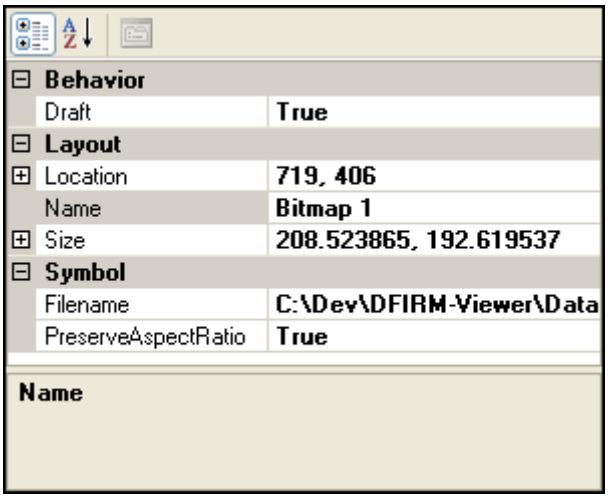


Figure 225: Print Layout Bitmap Properties

- **Filename –** indicates the configuration path in which the bitmap file is found.

- **PreserveAspectRatio –** a true/false property where true preserves the aspect ratio of the original bitmap file

1.6. Toolbox

The *Legend Window* has two tabs that are located above the upper left hand corner. They are labeled, 'Legend' and 'Toolbox.' The display under the 'Legend' tab is documented in the section entitled 'The MapWindow 6 Legend Window.' The 'Toolbox' tab allows the user to access various tools analyzing and manipulating the data connected to the layers in the map.

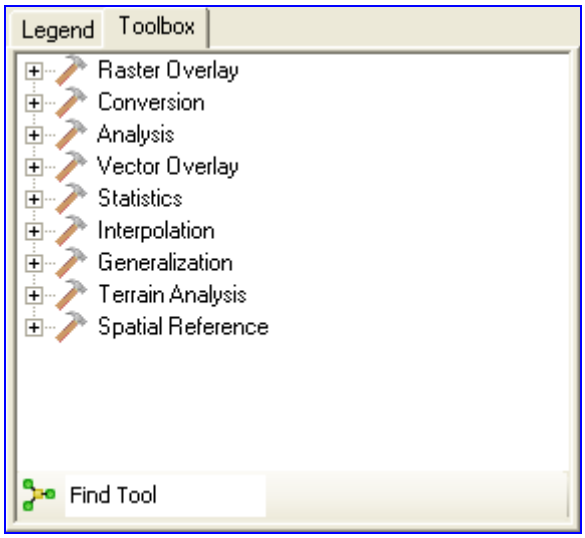


Figure 226: Toolbox

Many tools are available each with a [+] and [-] box which expands to reveal functions related to the main heading. This document will only be concerned with the 'Spatial Reference' tool in that its use is necessary when downloading new shapefiles and rasters from multiple sources.

All of the layers that can be added to a map are plotted with a particular scale and coordinate system. Therefore, layers that have a different scale and coordinate system cannot be plotted together. The 'Spatial Reference' tool is used to establish the layers projections. Click the [+] box to expand the tool and reveal the 'Reproject Features' function. Double click the function name to open the 'Reproject Features' window.

The 'input FeatureSet' field is where the user enters the layer in question. Click the arrow to the right to bring up a drop down list of all of the layers currently in use in the project. Click the desired layer to populate the field. The adjacent plus button can be used to locate a file that is not currently in use. The 'Source Projection' field indicates the current projections for the layer in question. The 'Desired Output Projection' is where the user would enter the needed type of projection.

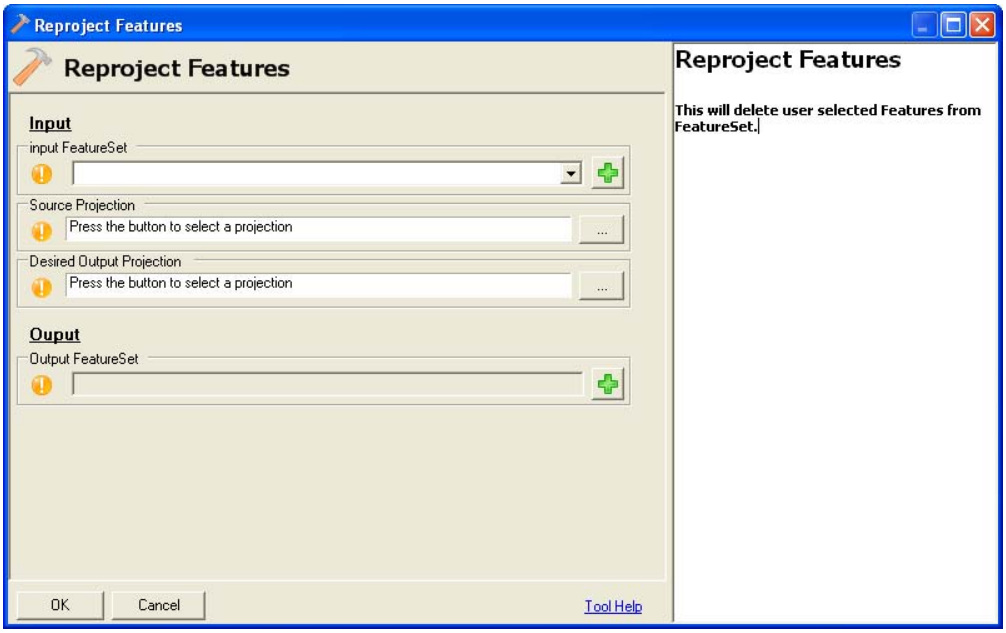


Figure 227: Reproject Features Window

Click the ellipse button [...] to open the 'Select a Projection' window. Here the user can select the projection type ('Projected' or 'Geographic') as well as a focus location and multiple projection sub-types in association with those locations. Also, custom projections can be made. The kind of projections used is not as important as making sure the projections for all of the layers in question match one another. Note that this documenter sets all of his layers to the 'Geographic' projection type, then sets the layers to 'World' and then 'WGS1984' when reprojection is necessary (but that's just me).

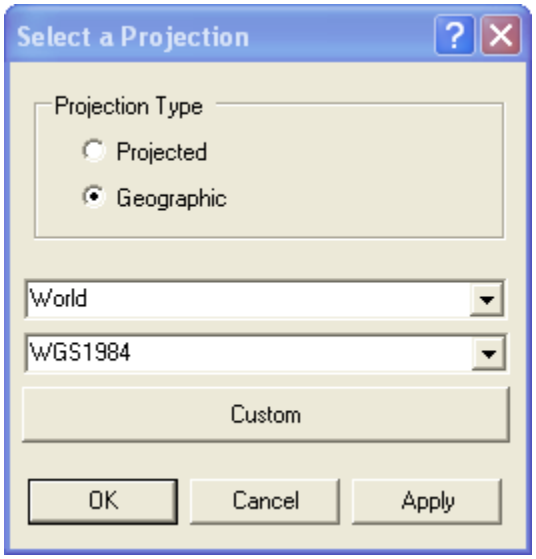


Figure 228: Select a Projection Window.

Once the desired projection is set, click the plus button to the right of the 'Output FeatureSet' field. This allows the user to save the layer with the new projections with a new name and location. Note that the original shapefile can be overwritten if desired. Click 'Ok' to apply the change. This process does not change the current layer in use. That layer must be removed from the legend and the new layer with the new projections must be added.

Appendix B: Vector Drawing Problems and Solutions

Innovation requires a lot of work and perseverance, especially in the face of obstacles. It is often very easy to learn how to do naïve programming using web references. It is much harder to find out the way to accomplish the same thing quickly, or by using less memory. This account provides a detailed narrative of the real-life train of thought and failed efforts that governed the work with GDI+ drawing. It is thought that this will be helpful for future developers that are encountering similar setbacks. The implication here is that higher level languages have enough nuances that the optimal way of doing things is not always obvious. This is especially true with many tasks in C#.

One of the most challenging aspects of getting a working GIS project is to ensure that the drawing speed is acceptable. In many ways a developer can have little control over how fast the program draws because he is relying on existing drawing frameworks. In this project we initially wanted to work with Graphics Drawing Interface (GDI+) rendering supplied by Microsoft as part of the .NET framework. The GDI+ library is responsible for drawing curves, rendering fonts, filling polygons and drawing images. The strong advantage of GDI+ is that it uses a single interface that can draw to many devices, most importantly the screen and printing. Therefore, the code that is developed to render content to the screen can be re-used for printing. The library has been translated through mono to work with other operating systems, not just windows, and there are methods that provide support for translucent fill types, anti-aliasing and several new drawing capabilities. The disadvantages of GDI+ drawing is that it has no support for animation (synchronizing with the framebuffer) and does not support turning 3D

content into a two dimensional image. Games and animated systems work with DirectX or OpenGL.

The best way to give advice to other GIS developers that are thinking about using GDI+ or DirectX for drawing is to walk them through the various stages of development that we have had on this project. The first step was to show that the GDI+ graphics object can be used to draw vectors. The DrawPolygon, DrawLines methods are the most obvious methods for drawing vectors. We designed a map control that could use these functions to draw shapes, cycling through the coordinates of geometries. The geometries are the OGC geometries being used for topology. As the map panned and zoomed, it used the Graphics.Transform method to adjust the scale and position, which allowed for the use of the original coordinates. However, to use the transform, we were forced to use floating point coordinates. 16 bit floating point numbers represent a problem for geographic coordinates because they can only hold seven digits of precision. Technically, in latitude and longitude coordinates, three digits are necessary to hold enough precision to represent decimal degree values. Subtracting three from the seven leaves only four remaining digits. A measure of .0001 decimal degrees has an accuracy of about 11.1 meters, or about 36 feet. While this is acceptable for rendering at the scale of the continental US, it would be a poor representation for displaying content on the scale of cities, roads or buildings, which are commonly required for cartographic mapping.

Drawing speed was a serious consideration because the drawing performance of our GDI+ map was slower than the performance of earlier versions of our own software. We started looking for shortcuts to help speed up the apparent rendering. For instance, if

a background process was used to draw the area around the map, then as users pan the map, that pre-fetched content could be shown immediately. This would improve the user experience without changing the time it takes to draw the content. Drawing the content around the map reduces the time it takes to see fresh material when panning the map. However when the scale of the map is changed, the entire extended region needs to be redrawn, which can slow down the zooming process. To improve the user experience while using the scroll wheel to change the zoom scale, we showed the previous cached image, but scaled it to match the new zoom extent. Images that are stretched larger than their original versions appear distorted as if they are made from very large pixels. This is still preferable because it could change the image scale as quickly as the scroll wheel could be adjusted, regardless of how long it took to render the full scene once the user chose a new extent. We also cached separate bitmap stencils for each layer. That way, if a user turned on or off a layer, they could immediately see the changes. However, that required extra memory for holding bitmap content for something nine times the size of the screen, and multiplied by the number of layers that were loaded plus one combined cache. The performance was considerably slower than rendering the same shapefiles using earlier versions of MapWindow, or with commercial GIS software such as ArcGIS. We began looking for other mechanisms for handling the drawing.

In the spring of 2008, we focused on using DirectX for rendering, giving up entirely on GDI+. The advantage of DirectX is that it provides access to low-level unmanaged memory for fast rendering. It can even allow direct synchronization of frame buffering for controlling animations. If coordinates are represented as single precision floating point values, they can be directly stored in the unmanaged vertex format that

allows for fast of coordinates as the perspective changes. DirectX does not support double precision floating point coordinates, and so suffered from the same rounding inaccuracy that occurred in the GDI+ implementation. With DirectX, it is possible to show 3D views, control rotation as well as zooming. However, it was not as easy to draw roads that had a constant apparent width on the screen. As a user zooms in, symbolic representation that is always the same size requires completely re-constructing the vertices that are being drawn, which makes it slow. It is common for a GIS application to represent features with the same size or width regardless of the geographic scale in order to improve visibility. Therefore, DirectX was ill-suited for rendering a very commonly used style of symbolic representation. Another problem that was not addressed before the project moved away from DirectX was filling polygons. Algorithms exist to do this within the OpenGL library, but as the drawing strategies moved progressively further away from DirectX, less and less time was devoted to maintaining and fleshing out the 3D map controls. At the time of this dissertation, that aspect of the 3D map remains incomplete.

In the summer of 2008 the MapWindow 6.0 development team, consisting mostly of Dan and myself, began to realize that there was more and more interest in gaining the portability (the ability to run the software on other operating systems) which can be achieved using an open source library called mono. For the most part, it was possible to do this, but mono did not support DirectX. So during the summer we implemented a re-structuring that allowed layers that used different rendering tools to share the same underlying symbology and data structures. This would allow for the sharing of as much

code as possible between a GDI+ 2D drawing map and a 3D DirectX drawing map. We knew the 2D map was slow, but at least it was portable, where the DirectX map was not.

As summer gave way to fall, I started looking at more advanced symbology. We wanted to support having borders around lines. Depending on the sequence that the line segments are drawn, the visual representation can look very different.

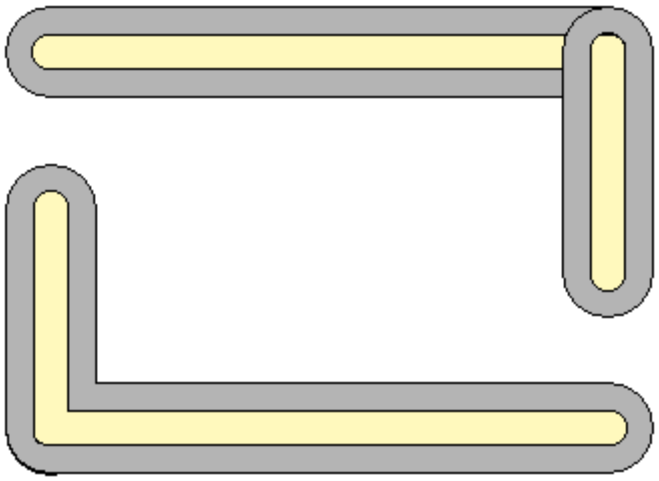


Figure 229: Two different overlap Styles

In the figure above, the top intersection resembles the result of two segments that are drawn separately. Whichever segment is drawn later ends up covering the one below it. In order to create continuous representations of roads, however, we can use the pattern shown in the lower intersection, where all the border lines are drawn and then the interior is added to the line. We made changes to allow for outlines around lines as well as outlines and dots for symbols. We created classes to draw the new characteristics in both DirectX and GDI+. We experimented with two different drawing styles for DirectX. In one approach, we designed simple road vertices, but drew the GDI+ road pattern type on top as a texture. In another approach we constructed the rounded endcaps by creating a rounded pie-slice set of triangles that came together to form the endcap. The latter

looked better when zooming in as the textures tended to become pixilated. However, it also took up more memory and space for all the extra vertices.

In August of 2008, I was also solving the problem of the imprecise floating point digits. For the two maps, I came up with separate solutions. In the DirectX case, the solution uses a temporary "projection" so that all the coordinates are translated from their original values to values that are scaled to work with the current view. I only need to do change the coordinate system when changing scale by more than a factor of 1000 as the user zooms in, so we only needed to re-generate the vector data occasionally. This allows the coordinates to keep the needed precision. It is also responsible for some hesitation when zooming in. When the frame needs to regenerate the data values, it takes some noticeable time. In the future, this might be sped up by only translating the features that are within a reasonable distance of the zoomed in extent, rather than translating every single feature.

My initial improvement in GDI+ followed the same line of thinking until I decided that if we stopped using the Graphics.Transform in order to control scale and panning, we could simply translate directly from the double precision coordinate to integer screen coordinates each time we draw. By transforming the coordinates directly, we do not require a separate floating point cache of coordinates in memory. This change altered the drawing methods for the GDI+ rendering slightly since we had to mathematically modify each coordinate before adding it as a Point to the array of points to draw for each shape, but ultimately the performance was the same, and the code was a lot simpler than having a whole extra set of floating point coordinates in an arbitrary projection. As we progressed in the fall of 2008, we were primarily adding new support

for selection handling, improving raster support, and solving bugs, but the principal vector drawing strategy was the same. Use GDI+ for portable but slow rendering, and DirectX for faster rendering.

In the spring of 2009 I had some major breakthroughs. The major reason that the GDI+ rendering was so slow was that we were calling the drawing method (like Graphics.DrawPolygon) over and over again. Each time a method is called, it has a tiny performance hit. Multiplied across thousands of shapes, the performance hits produce a noticeable delay. I initially thought there was no help for it, until I discovered that it is possible to separate drawing elements using a GraphicsPath. In other words, it is possible to build a long list of things to draw, and send all of those things down to the low-level drawing code at once. Using a GraphicsPath ended up being much faster than what we were able to get by drawing one shape at a time. In fact, it was so much faster that the actual drawing itself was no longer the performance bottleneck.

Not only was it taking time to draw each feature, but there was a considerable amount of time required to simply read the coordinates from the data structure. The coordinates were using an ICoordinate interface and accessing property accessors like X and Y on the coordinates. Internal to the coordinates, the X and Y values were stored in short arrays. (Each coordinate had an N dimensional array, where N is the number of dimensions of the coordinate.) The performance of these structures was quite a bit slower than accessing the values directly from the arrays.

In order to speed up drawing, I cached a separate array that included all the vertices for the entire shapefile. The cached array was a jagged array where each member of the outer array was a reference copy of the inner array that had the coordinate

values. This was the first time that rendering in the GDI+ map was rendering content at the same speed or faster than previous versions of MapWindow. The buffering techniques that we added earlier to improve the user experience were now acting to impair that performance. That is, it was taking longer to draw to the buffers and then draw the buffers to the screen than it was to simply draw the raw shapes to a single buffer, and re-draw all the layers when a layer was turned on or off. Removing the layer stencils improved the drawing performance even further. I also enabled an optional mode where the buffer does not cache the content around the edge of the map, but only draws the content on the screen as needed. Not having to draw the region outside the visible portion of the screen speeds up zooming, but reveals white space when panning, and does not show new content until the panning stops. This is optional, and with the stencil buffers removed and fast drawing speeds, the difference is subjective and we chose to leave it as an optional setting.

During the summer of 2009 I changed the symbology. Instead of working with lines with borders, for instance, I created an extensible symbology set that could use stacked layers of drawing symbols. To make roads with borders, simply two layers are used. A whole host of new drawing options were added like character symbols that can use downloadable fonts as point symbols. We added gradients and hash symbols to the polygon drawing options. Each of these additions was accompanied by its own complete set of user controls to allow the symbology to be edited by developers.

Before the summer of 2009, the symbology had to be the same across the entire thematic layer. That is, regardless of the attributes, all the lines would be the same thickness and color, and all the points would have exactly the same symbol. In the

summer of 2009 we extended the symbology classes to include drawing categories. Categories are classes that combine a filter expression that uses attributes to restrict the features that are included in the group, but all the members of the group are displayed using the same symbol. In the fall of 2009, we added forms that allowed users to design themes that would alter the appearance to correspond with the attribute values. Each layer has a single scheme. Each scheme holds one or more categories. Each category had a string filter expression using standard SQL language to choose its members, and a Symbolizer that controls how the members are drawn. The Symbolizer, in turn, can hold multiple Symbols, Patterns, or Strokes, depending on whether the layer was describing points, polygons or lines respectively. In the case of lines, each stroke describes a System.Drawing.Pen, which is a .NET class that controls how lines are drawn. A pattern describes a System.Drawing.Brush, which is a .NET class that controls how areas are filled. We did not use the .NET classes directly because our classes also needed to support serialization. Serialization is the ability to save the characteristics to a file and then re-create those classes from the file later. In order to draw a set of lines with several strokes, the drawing code simply re-uses the same coordinates again with each of the distinct pens.

In the late fall of 2009, it became apparent that the limiting reagent on performance was the load times. It was taking close to twice as long to load the same data as it was to load the data from the 4.0 version of MapWindow, which relies on an ActiveX control. It was also taking much more memory to hold the shapes in ram than the vectors were requiring on disk. For instance, one particular line shapefile took about

80 megabytes on the harddrive, but was expanding to nearly 1 gigabyte of memory in ram.

The first problem was that even though I was not creating DataTables in memory, I was loading and caching an array of characters from the DBF file. As an example, in at least one dataset, as much as 80% of the space required in memory was for the attributes. For large shapefiles it was preferable to reduce the amount of ram required by reading the attributes directly from the disk on demand. For shapefiles with more than about 50,000 shapes, I no longer load the attributes into memory, but rather set up a system that can read the content from the disk on demand. Reading values from the disk is slower for any process that needs to read every single row of data, but requires much less memory to be used. Microsoft's DataGridView control supports a "VirtualMode" which allows .NET developers to populate the contents of the DataGridView on demand. The result is a faster loading AttributeTable editor that can also allow editing of data values, but requires almost nothing to be loaded into ram. The down side is that sorting is disabled since it would require knowing information about all the columns in order to sort the data.

These steps reduced the largest portion of the in-memory data store, but as it turns out, having a jagged array of small arrays in inefficient. Each array takes the same amount of memory as the two coordinates that were most likely the only coordinates stored in the array. As a result, I re-organized the vertices. The XY array is now a single array of XY values. The reason for them sharing the same array is that that is how they are stored in the shapefiles. It makes it easier to convert a large collection of coordinates in the shapefile into an array of doubles.

Some other helpful optimizations to speed up reading values involves doing conversion from bytes to doubles or from bytes to integers in batches. Before, we were reading each coordinate, one at a time, from the file and using the BitConverter class in order to convert the bytes into double values. However, another way to accomplish the same result is to use a method called Buffer.BlockCopy. The block copy method is basically designed to copy data in bytes, but allows developers to use arrays of primitive data types (like integers and doubles) instead of arrays of bytes. That means developers can have an array of bytes be copied to an array of primitives, or vice versa. The advantage is similar to the advantage if drawing all the shapes at once instead of drawing them one a t a time. If a method is called over and over again, any overhead from making the method call itself starts to add up. Converting millions of doubles separately might take 500 milliseconds, while converting the same values all at once will likely take less than 50 milliseconds. Even though the units of time are small here, the performance change is noticeable when combined with the other steps required to open and display the file.

Another strategy that I adopted was not generating the features right away. The reason for reducing the use of geometric feature classes is that features require working with the property accessors of the coordinate classes, and in the case of polygons, requires considerable work to tease out which parts are holes, and which parts should be constructed as separate polygons as part of a multi-polygon geometry. Building polygons that have a valid OGC geometric structure takes time. In the earlier efforts at drawing polygons with GDI+, when using a method like “DrawPolygon” we needed to know which parts were holes. The reason is that if the holes are drawn in a separate

"DrawPolygon" call, rather than erasing content in the original polygon, new filled content would be drawn on top of the original.

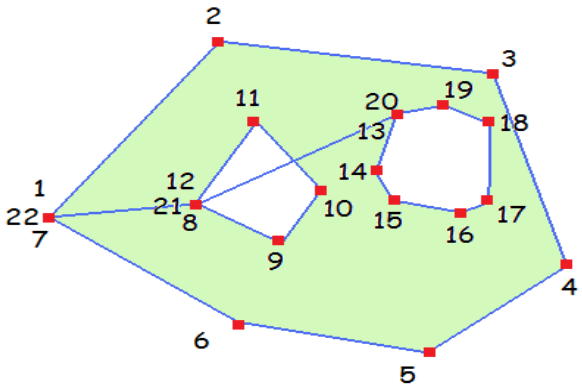


Figure 230: Old Polygon Vertex Order

In the figure above, we illustrate the original strategy that I was using in order to fill the polygons without filling the holes. The most important thing is the repetition of the points 7 and 8 after drawing the holes, but in reverse order as 21 and 22. The additional coordinates allow the polygon filling algorithm to precisely avoid the holes. It does not matter if the line passes through a hole or not, just so long as the tail coordinates are added correctly to the end. To build polygons using this method, however, it is necessary to know which polygons are holes and which are not, and further, it is necessary to know which shell contains the holes. As it turns out, this became unnecessary as soon as we switched to drawing all the polygons at once using a GraphicsPath. Because the separate parts are all passed as part of a single polygon drawing operation, any holes that fall within existing polygons will automatically be drawn as empty. The automatic treatment of holes during drawing has several implications. Firstly, I no longer need a separate list of coordinates for filling than the ones I use for drawing the borders. That saves on the amount of space required in ram for the graphics paths. Secondly, it means that I no longer need to perform the

comparatively slow calculations when loading polygons to determine whether or not parts are holes. The geometrically correct description of a polygon is still useful for doing topology calculations, so I have provided a GetFeature method to allow the creation of those features. But for rendering we just need to know the vertices, and where a part begins and ends.

The net result is that the latest version of MapWindow 6.0 is fast and lean. It takes up very little more than the file-size in memory and can load and draw faster than MapWindow 4. The most obvious improvements, however, are for calculations that extend beyond the application. For the ActiveX control, any rendering that was done as part of the actual c++ code in the control was fast. However, since even getting the X and Y values of the coordinates was accomplished through a relatively slow com interop barrier, even tasks like selection would become painfully slow for large shapefile. In our tests, a shapefile with 200,000 lines took MapWindow 4 about five minutes to perform the select operation. In MapWindow 6.0, selecting the same lines took less than two seconds.