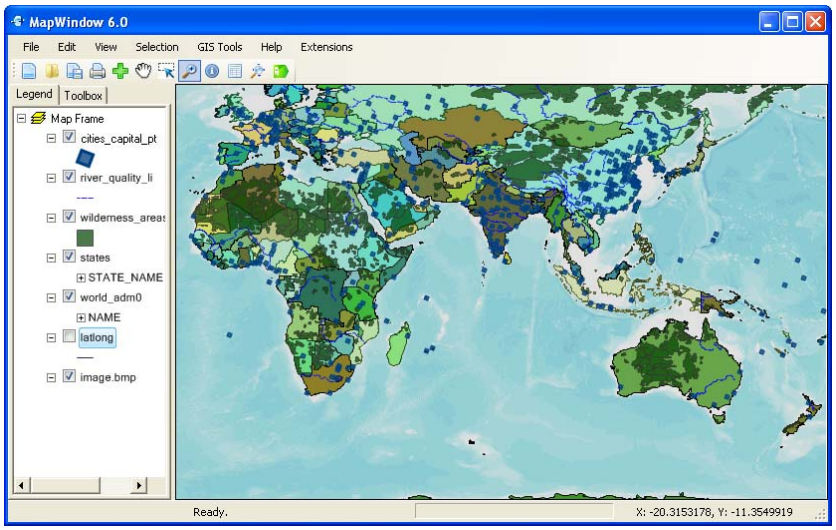
MapWindow™ GIS

Developer's Corner



Harold (Ted) Dunsford Jr.

Mark Van Orden

Jiří Kadlec

October 2009

Idaho State University Geospatial Software Lab

Table of Contents

1. Developer's Corner

This section gives a bit of an overview of geometric relationships, which are important for understanding some of the vector analysis options, and then gives some workable exercises to demonstrate the current capabilities of MapWindow 6.0. This should be thought of as a kind of developer's preview, and not necessarily a comprehensive developer's guide, which will be made available before the MapWindow conference in Orlando in 2010. This part of the document covers using MapWindow components, stitching together a working GIS by dragging the important components from the toolbox in visual studio, and then gives a detailed description of how to programmatically add layers and work with symbology or complex symbolic schemes.

A huge focus for the .Net version of MapWindow is to put much more control into the hands of developers. By providing everything in the form of .Net components, it makes it far simpler to pick and choose what sections of the framework you want to work with. If the map is all you need, you don't need to bother adding the legend or our custom status strip or the toolstrip. However, those components are provided for you so that it you can stitch together a working GIS without writing a single line of code.

1.1. Geometry Cheat Sheet

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 to 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 what dimension, it can represent any dimension as "True". Likewise, if it is required that the set is empty, then "False" is used.

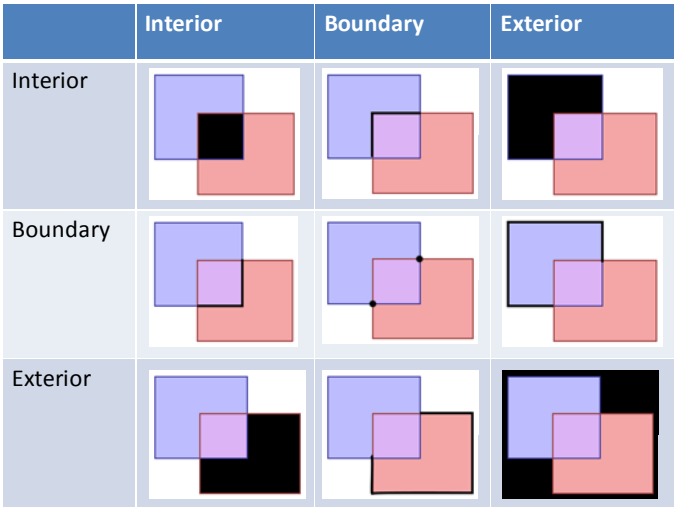


Figure 1: Intersection Matrix

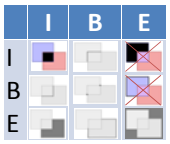
Graphically, we are illustrating the intersection matrix for two polygons. Some tests can be represented by a single such matrix, or a single test. Others require a combination of several tests in order to fully evaluate the relationship. When the matrix is represented in string form, the values are simply listed in sequence as you would read the values 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:

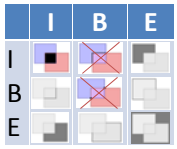
- T: Value must be "true" – non empty – but supports any dimensions >= 0
- F: Value must be "false" – empty – dimensions < 0
- *: Don't care what the value is

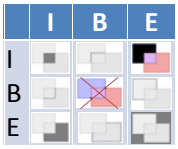
- 0: Exactly zero dimensions
- 1: Exactly 1 dimension
- 2: Exactly 2 dimensions

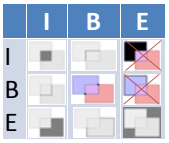
The following is a visual representation of the test or tests required in each case. A red X indicates that the test in those boundaries must be false. A colored value requires that the test be true, but doesn't specify a dimension. A gray value indicates that the test doesn't care about the value of that cell.

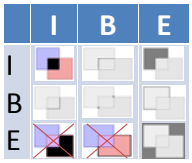
- **Contains:**
  - Every point of the other geometry is a point of this geometry, and the interiors of the two geometries have at least one point in common.
  - T*****FF*
  - 

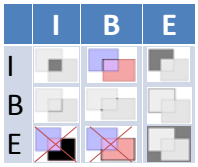
- **Covered By:**
  - Every point of this geometry is a point of the other geometry.
  - T*F**F***, *TF**F***, **T*F*** or **F*TF***
  - 

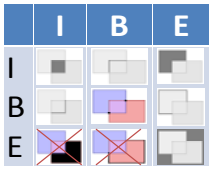
, 

, 

, or 

- **Covers:**
  - Every point of the other geometry is a point of this geometry.
  - T*****FF* or *T****FF* or ****T*FF*
  - 

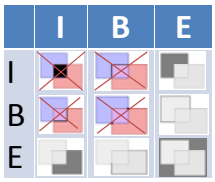
, 

, or 

- **Crosses:**
    - Geometries have some but not all interior points in common.
    - T*T***** (for Point/Line, Point/Area, Line/Area)
    - 

    - T*****T** (for Line/Point, Line/Area, Area/Line)
    - 

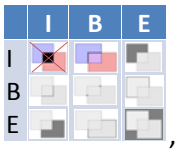
    - 0******** (for Line/Line Situations)
    - 

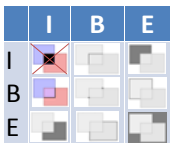
- **Disjoint:**
    - The two geometries have no point in common.
    - FF*FF****
    - 

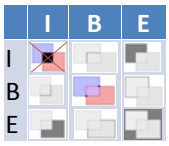
- **Intersects:** NOT Disjoint
    - The two geometries have at least one point in common.

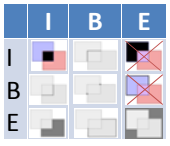
- **Overlaps:**
    - The geometries have some but not all points in common, they have the same dimension, and the intersection of the interiors of the two geometries has the same dimension as the geometries themselves.
    - T*T***T** (for Point/Point or Area/Area)
    - 

    - 1*T***T** (for Line/Line)
    - 

- **Touches:**
    - The two geometries have at least one point in common but their interiors do not intersect.
    - FT*******, F**T*****, or F***T****
    - 

, 

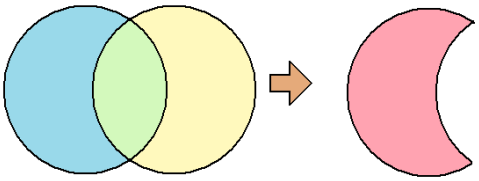
, or 

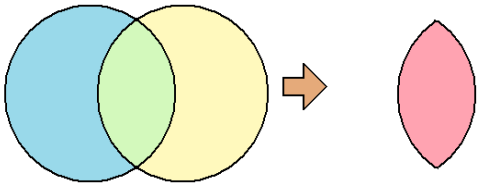
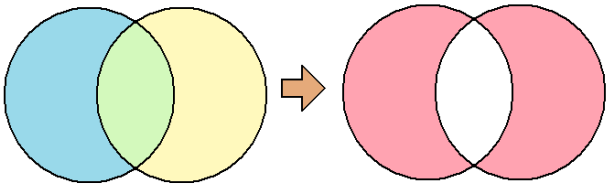
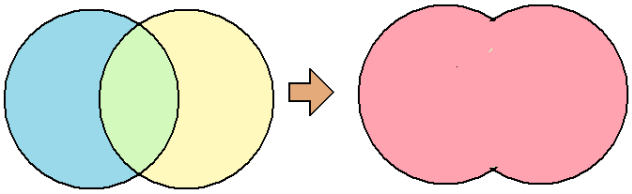
- **Within:**
    - Every point of this geometry is a point of the other geometry and the interiors of the two geometries have at least one point in common.
    - T*F**F***
    - 

1.1.1. Overlay Operations:

Being able to test the existing relationships between geometries is extremely useful for doing analysis, but many times you need to alter the geometries themselves. Frequently you want to use other geometries to accomplish this. Consider the case of a clipping operation. In the figure below, 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. These are not limited to specific scenarios like polygon to polygon. Instead the same terminology applies to all the geometries using the following definitions.

Figure 2: Before and After Clipping Rivers to Texas

- **Difference:**
  - Computes a Geometry representing the points making up this geometry that do not make up the other geometry.
  - o 

- **Intersection:**
    - Computes a geometry representing the points shared by this geometry and the other geometry.
    - 
- **Symetric 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.
    - 

1.2. Exercise 1: Assemble a Map Project

Putting together a GIS project has never been easier. Even a novice developer can take advantage of our ready-built mapping controls, dragging and dropping them onto a solution. Because the components are largely independent, they can be re-arranged in different layouts, or used separately. Furthermore, the extensive use of interfaces for the controls allows a control like the map control to be used interchangeably with alternate controls that act like a legend.

In this first exercise, we will take the assembly step by step. If you have never worked with anything besides the built in .Net controls, this exercise will be useful because it will demonstrate how to add the MapWindow components to your visual studio developer toolbox. It will also show the basic way that the most fundamental map controls can be added to the map.

1.2.1. Step 1: Start a New C# Application

The first step of the exercise 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 code is required. To get to new project dialog in visual studio, simply navigate to File, New, and choose Project from the context menu. This will display the New Project dialog displayed in figure 1.

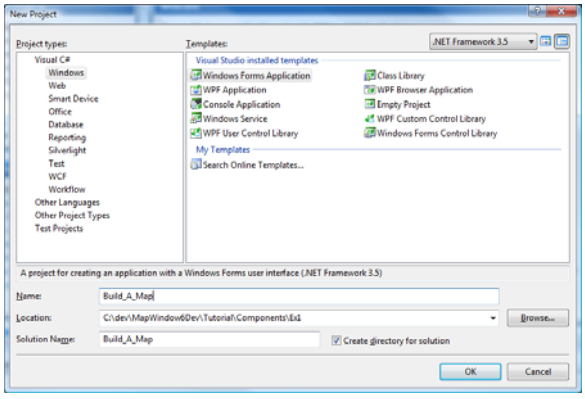


Figure 3: New Project Dialog

Change the name and path to something appropriate. In this case, we chose “Build_A_Map” and C:\dev\DotSpatialDev\Tutorial\Components\Ex1. Make sure that the Project type is Visual C# and Windows. Ensure that the Template is set to Windows Forms Application. Then click OK.

1.2.2. Step 2: Add MapWindow Components

The first step in any project is to ensure that you have loaded all of the designer controls and components into the MapWindow toolbar. As this tutorial was created in the alpha stage of development you will notice that not all of the controls have unique representative icons or other helpful instructions. There is also a large assortment of extra controls that were used for designing MapWindow 6.0 that we may choose to remove from the toolbox for the public release. However the basic technique will remain the same. First, we want to create a new tab to store the MapWindow tools. You can do this by right clicking on the toolbox and selecting the Add Tab option from the context menu.

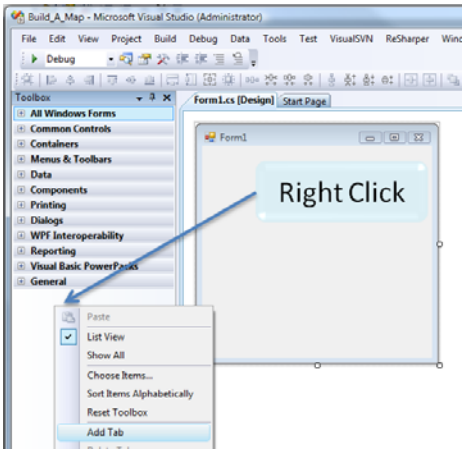


Figure 4: Add MapWindow Tab

This will allow you to edit the name of the tab. We chose to name the tab “MapWindow” so that we could easily keep the controls from the MapWindow library together. Once you have added a MapWindow tab, you will want to right click in the blank space below that tab and select “Choose Items” from the context menu.

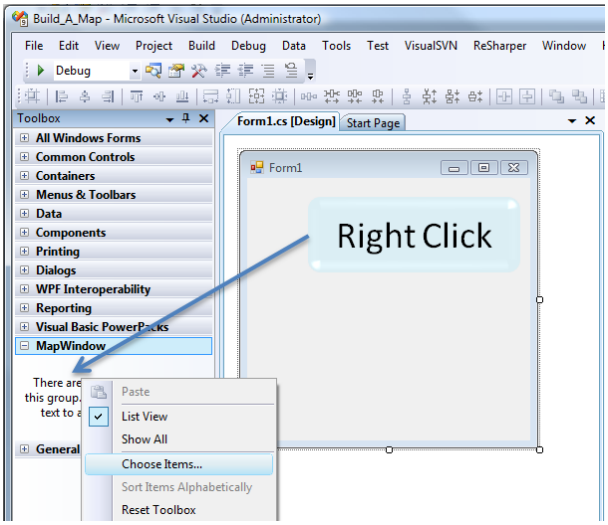


Figure 5: Choose Items

The choose items option launches a new dialog which will allow you to select from various pre-loaded .Net controls as well as some COM controls. However, we are going to use a third option, and browse for the DotSpatial.Desktop.dll file.

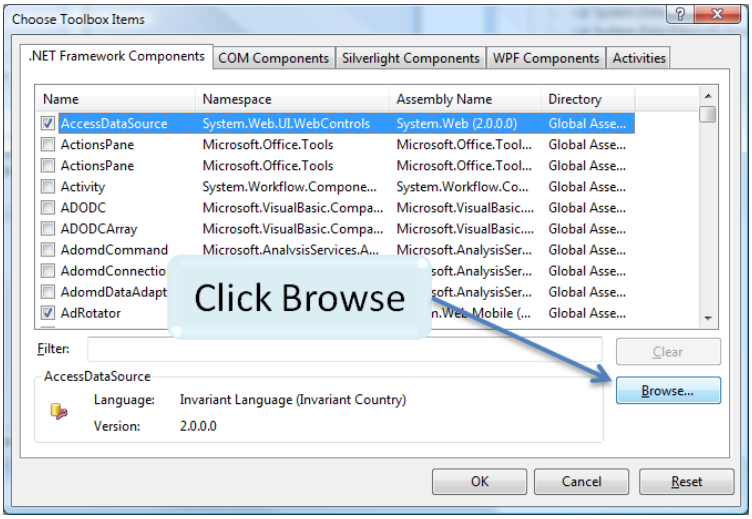


Figure 6: Browse

This, in turn, launches a file browser, and you will have to navigate to wherever the DotSpatial.Desktop.dll file is found on your local machine. On this machine it was in the C:\Dev_II\dotspatial\Release folder.

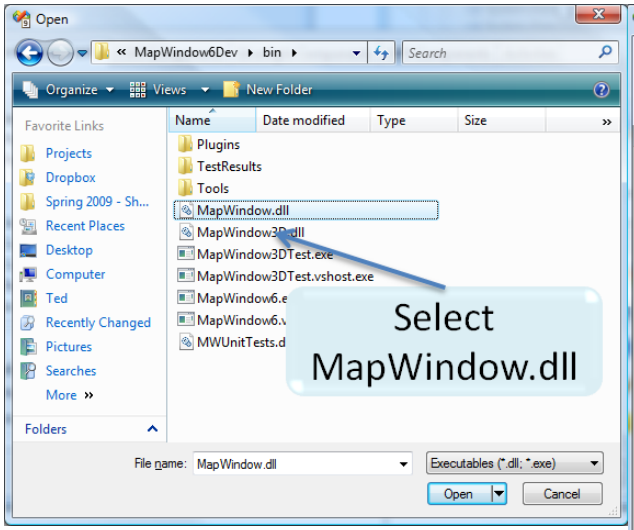
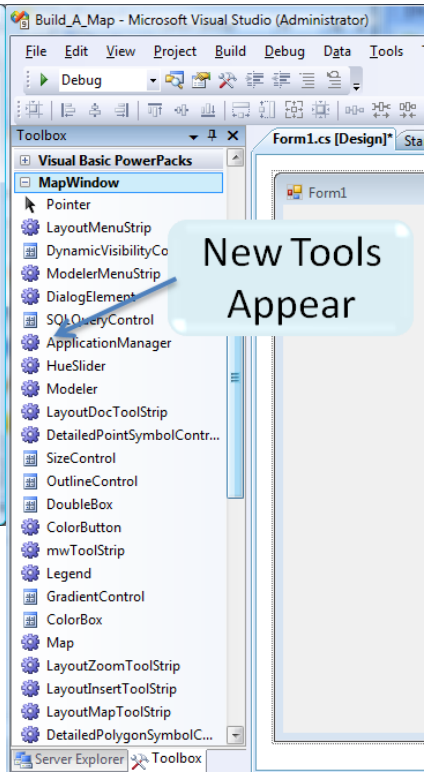


Figure 7: Select DotSpatial.dll



Once you have chosen this document, it will cause a large number of tools to be added to your toolbox, as is illustrated in the figure to the right.

Figure 8: MapWindow Tools

1.2.3. 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. This 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. Simply drag the MenuStrip listed there onto the main form.

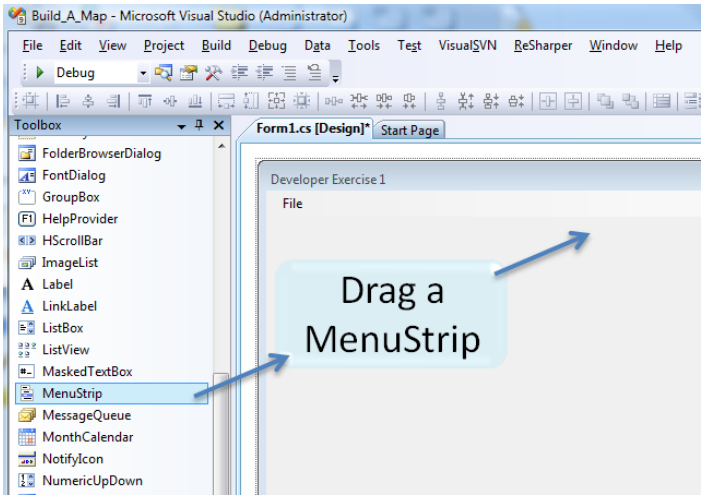


Figure 9:Drag a MenuStrip

Next we will switch back to the MapWindow controls and add two controls that are directly associated with the MapWindow components. The first is the SpatialToolStrip, 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 SpatialStatusStrip.

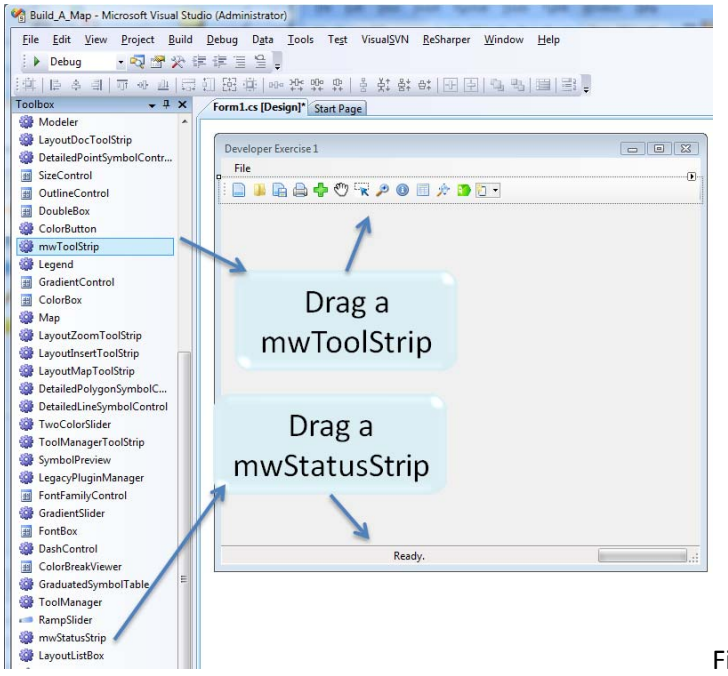


Figure 10: Add Status and Tool Strips

1.2.4. Step 4: Add the Map

Now that the peripheral controls have been added, we can 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, we will take advantage of a .Net SplitContainer control. This will divide the content into two separate panels that are sizeable by the user.

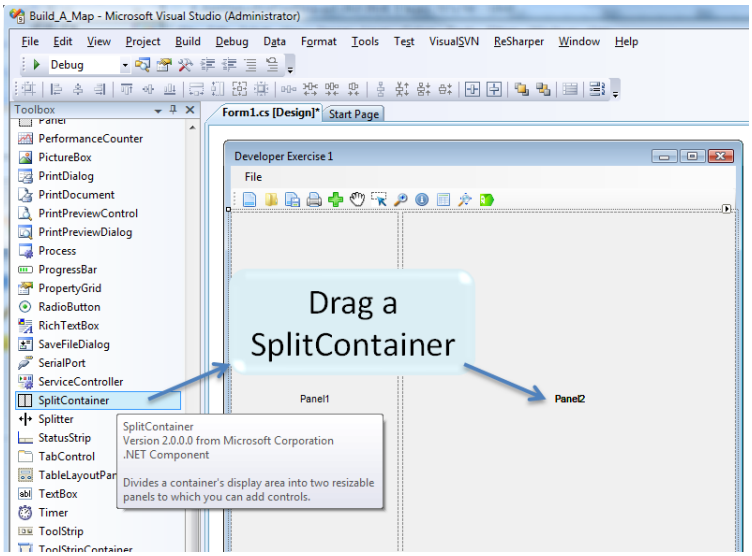


Figure 11: Add a SplitContainer Control

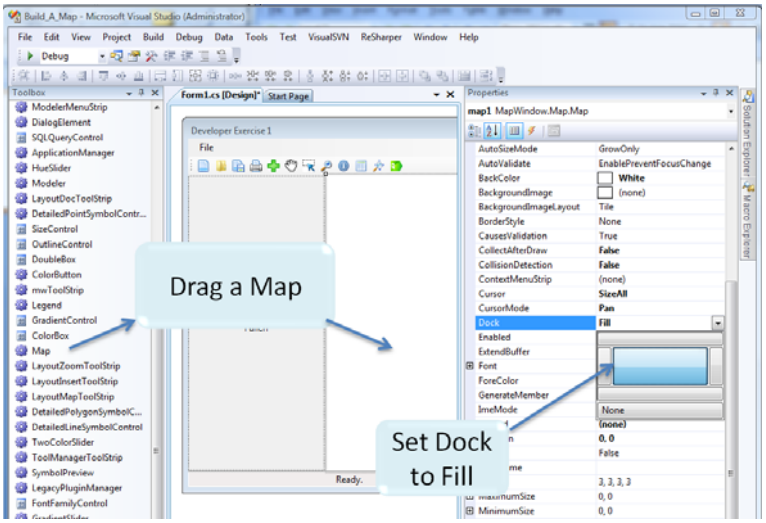


Figure 12: Add a Map

Once the separate panels exist, we can add the map to the project. When adding the map, it will likely be the wrong size for the panel that we created. In order to allow the user to resize the map so that it always is the right size for the panel, change the "Dock" property to "Fill". This can be done by choosing the central rectangle in the drop down editor that appears in the property grid when you click on the down arrow.

1.2.5. Step 5: Add the Legend and Toolbox

Because the Legend and Toolbox can both exist in support of the map, and it is not critical to have both of these tools visible at the same time, for this project we will take advantage of the .Net Tab control to help re-use the same space more effectively. Because these components are interchangeable, the Tab control is not necessary, and second split panel, fixed panels, or even third party docking panels are all acceptable alternatives that will not affect the proper behavior of the map, legend or toolbox.

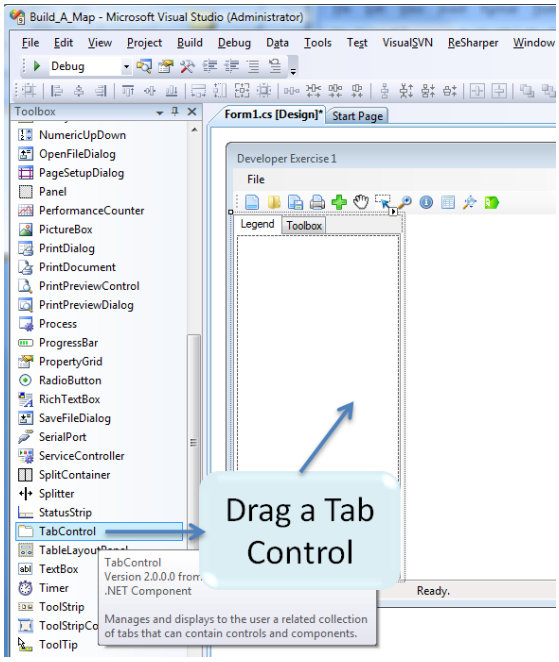


Figure 13: Add a Tab Control

In this instance we added the tab control to the left panel, and changed the text on the two tabs to read Legend and Toolbox. This can be done through the property grid that appears when you activate the tab control by clicking on it after it has been added to the main form.

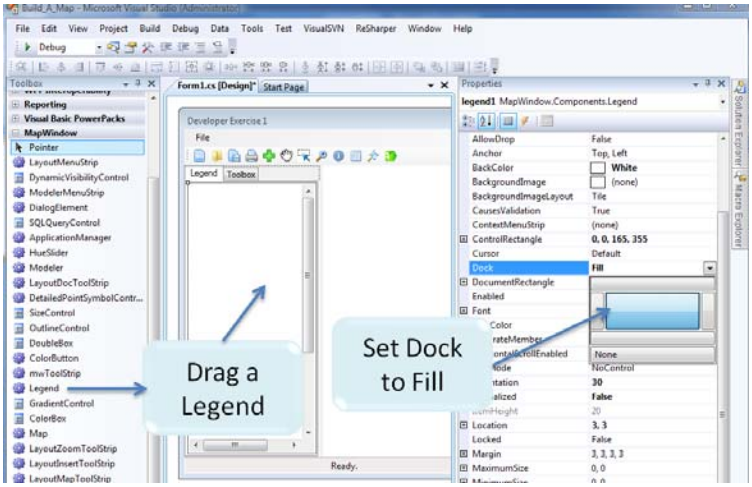


Figure 14 : Add Legend

Adding the legend follows similar rules to adding the map. You can simply drag this on top of the tab control when the "Legend" tab is selected. This will automatically tell the designer that the legend control will only appear when the Legend tab is selected. Like the map, the legend should have its dock property set to fill in order to take advantage of the splitter control's ability to resize the layout.

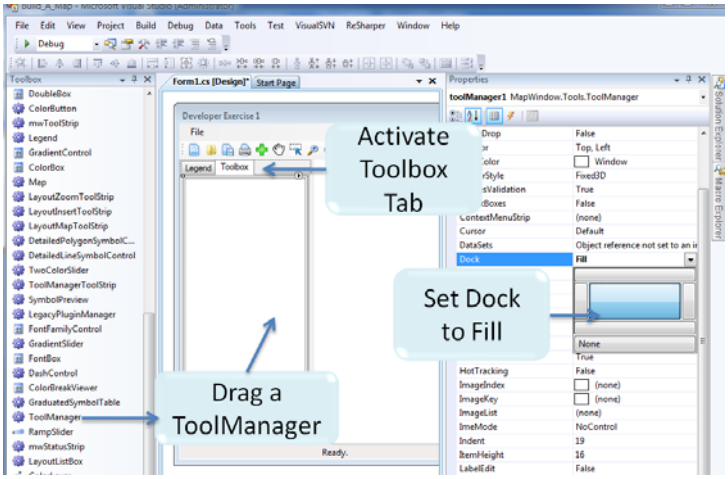


Figure 15: Add a Toolbox

Although we are not going to enable the toolbox in this exercise, it would be a good idea to add one. This will enable us to directly use our current project as the starting point for the next exercise. To add the toolbox, simply switch the tab control to the "Toolbox" tab. Then, drag and drop the ToolManager. Like the legend and the map, you will want to set the dock to full. Once you have done this, select back to the Legend tab so that when the application starts it defaults to showing the legend, rather than the toolbox.

1.2.6. Step 6: Link It All Together

We could technically run the project right away, but it would not appear to do anything. The add data button, for instance might happily open a file dialog, but nothing would happen when it was finished. 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, we need to link things together. The map has a property for the Legend and ProgressHandler. Set these to legend1 and mwStatusStrip1 respectively.

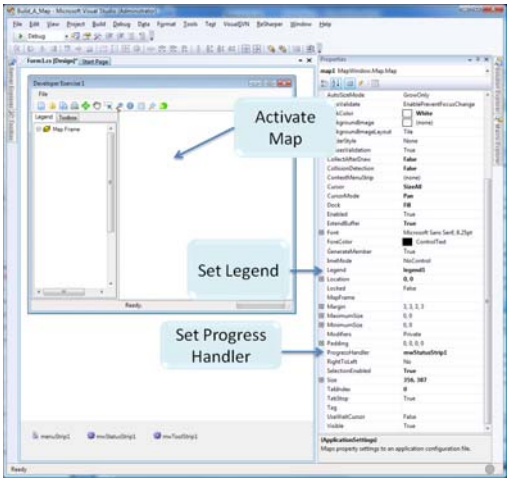


Figure 16: Link Map

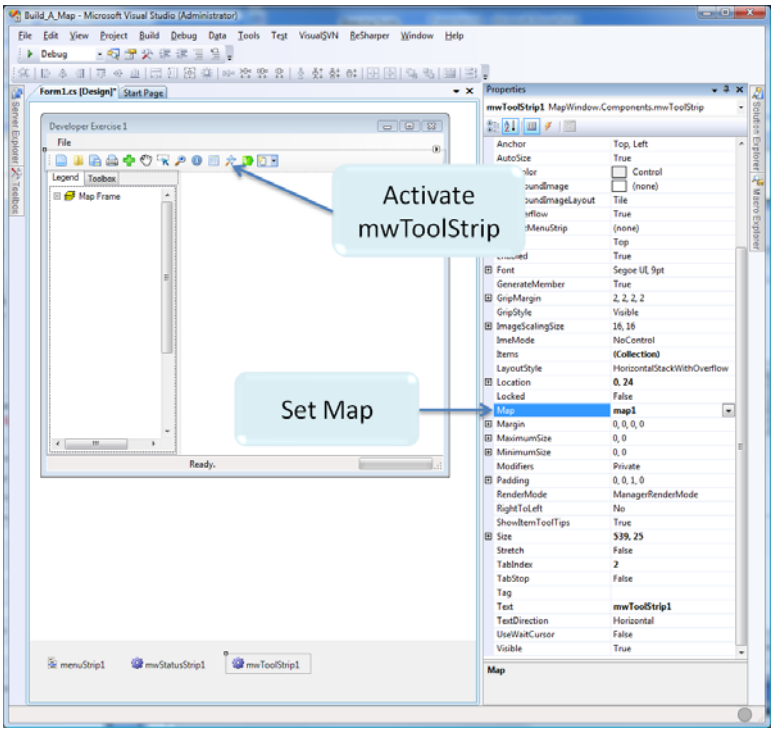


Figure 17: Link Tool Strip

The next step is to link the tool strip to the map. This can be done by simply activating the mwToolStrip, choosing the Map property and selecting map1. In order for the application to understand what to do with the data, a system reference to the DotSpatial.Data.dll must be made. This .dll is located in the same file as the DotSpatial.Desktop.dll that was reference to access the GIS controls. Now that the project is connected together, we can start our project and run it on some test data.

The first thing to do is to get some data. For our sample, almost any data in shapefile format will do, but in the spirit of improving online data awareness, this book features many online data sources that should provide up-to-date GIS data.

Help Tip

Sometimes links can become broken or out of date. In cases like this, it is often useful to look at the first part of the address and then search for data manually. Example: instead of http://www.census.gov/geo/cob/bdy/co/co00shp/co99_d00_shp.zip , you could use http://www.census.gov

The raw data in this case is stored in the form of a zip file. Most modern operating systems can unzip files automatically, but in the event that you need an unzip utility, a free, open source utility called 7-zip is available for download from http://www.7-zip.org. Once you have downloaded and extracted the shapefiles, you will see a .shp, a .shx and a .dbf file all with the same name before the extension. This is the basic file format known as an ESRI shapefile, and is a commonly used format for GIS analysis because it is portable and has an open standard and so is widely compatible between different GIS software vendors.

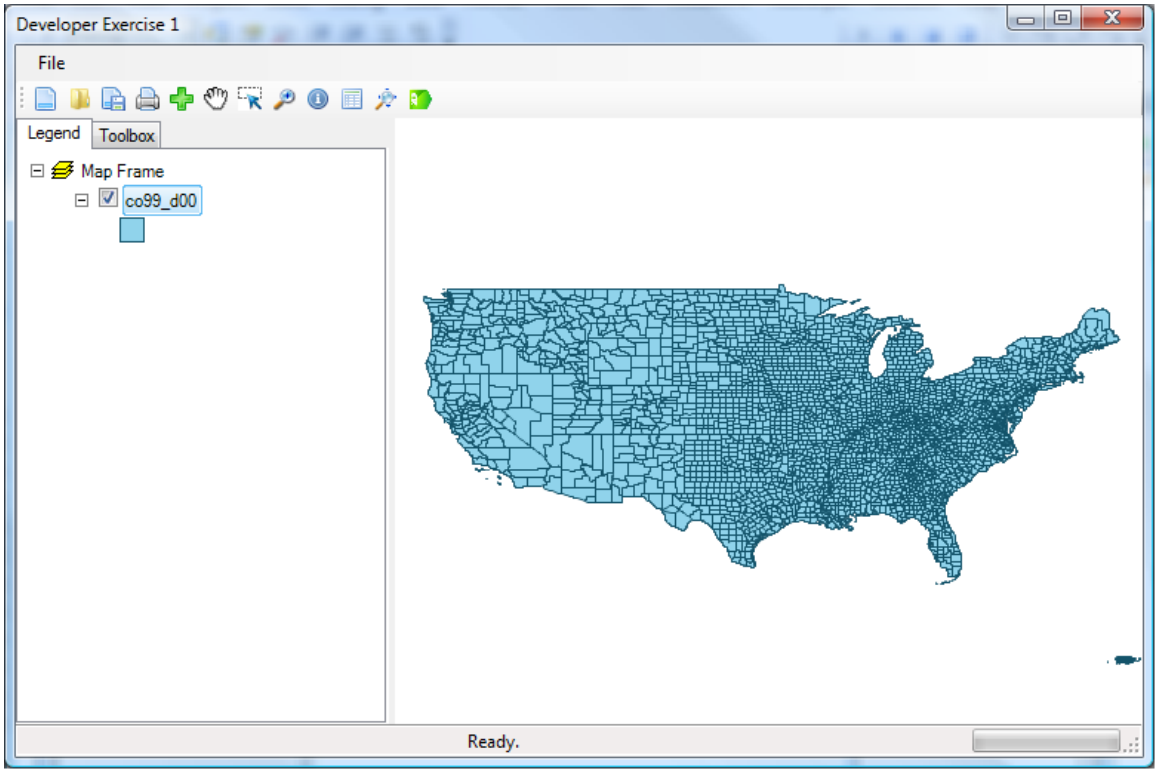


Figure 18: Census Data

The figure above illustrates the view of the continental United States. Because the data also includes counties in Hawaii, Alaska and Puerto Rico, it will be necessary to zoom a little to zee a view like the one above. The map starts automatically in the "Pan" mode. In this mode, you can click on the map with your left mouse button and drag the map in a direction to have it update the view. You can also use the mouse wheel to zoom in or out of the scene. The basic operation follows the operation of the Users section for MapWindow 6.0, and so you can get a much more detailed picture of what is possible with your map.

1.3. Exercise 2: Simplify Australia Data Layers

1.3.1. Step 1: Download Data

For this exercise, we need some thematic vector layers that represent more than just polygons. In honor of the Sydney Free and Open Source Software for Geospatial (FOSS4G) 2009 Conference, for which this document is being written, I am downloading some basic Australian datasets. Sometimes basic GIS data is offered on sites for free in the hopes of promoting the data with greater detail.

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

For this exercise, we downloaded the AUS Cities, AUS Lakes, AUS Political, AUS Populated Areas, and AUS Major Transportation shapefiles. These files are stored in zip format, so you will have to unzip them first (see exercise 1). I found that the cities shapefile in this example was misleading because it does not contain any of the major populated city areas. Instead, before we do any programmatic symbolizing, we can get better points that are a better representation of the cities from the polygons that are "populated areas". To do this, you can use the MapWindow 6.0 application to convert the populated areas polygons into cities.

1.3.2. Step 2: Calculate Polygon Areas

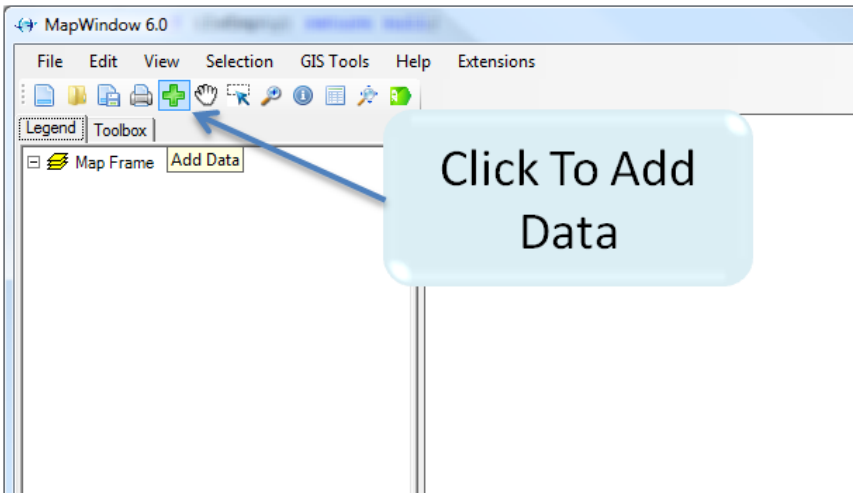


Figure 19: Add Data

After clicking the green plus to open a file dialog, browse for the "australia_populated_areas.shp" shapefile you just downloaded and extracted. When it opens, you should see the polygons that represent large city areas, mostly at the outer limits of Australia. Since the polygons are small compared to the size of the entire continent, we can see that the city regions are in fact polygons by zooming into the Sydney area.

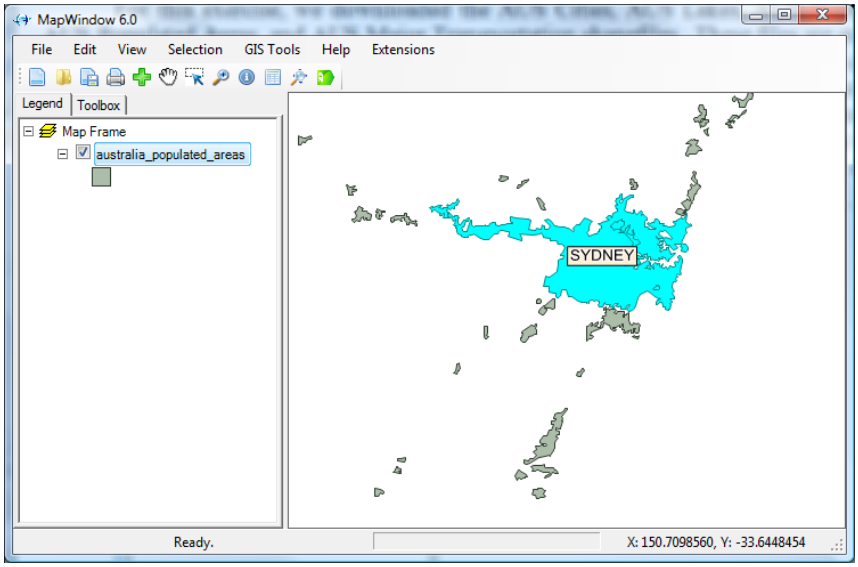


Figure 20: Sydney Polygon

We will eventually like to be able to distinguish the largest cities. Because the downloaded polygons don't contain any information about the population or area, we will use the area calculation tool 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. To see the toolbox, simply change the tab to the Toolbox tab.

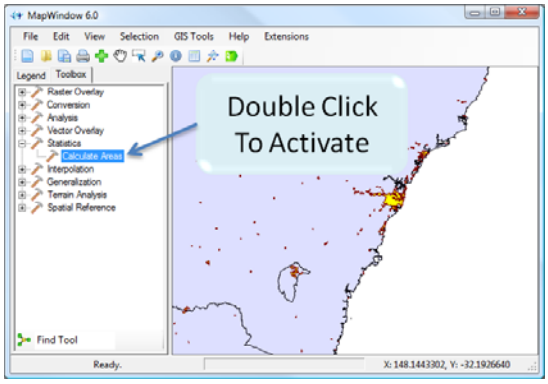


Figure 21: Calculate Areas

Help Tip

Since this exercise was constructed using a pre-release version, later versions might not have the same layout. Viewing the toolbox might be an option made available through the View menu, so if the toolbox tab is not obvious, check the view menu for an option for adding the toolbox tab.

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 as useful as population for selecting large cities, it at least should give us another tool to work with.

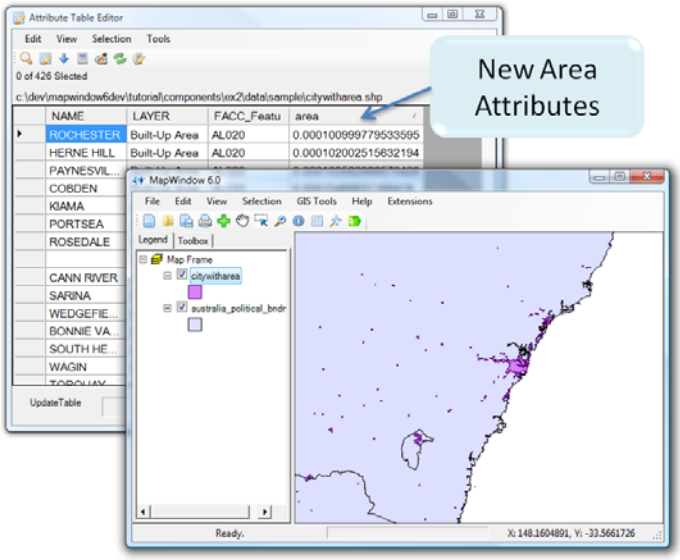


Figure 22: Newly Added Areas

1.3.3. Step 3: Compute Centroids

Now that we have a way to at least order the cities by way of areas, we can now create a simple cities layer from the existing polygons. We can do this by using the centroid tool.

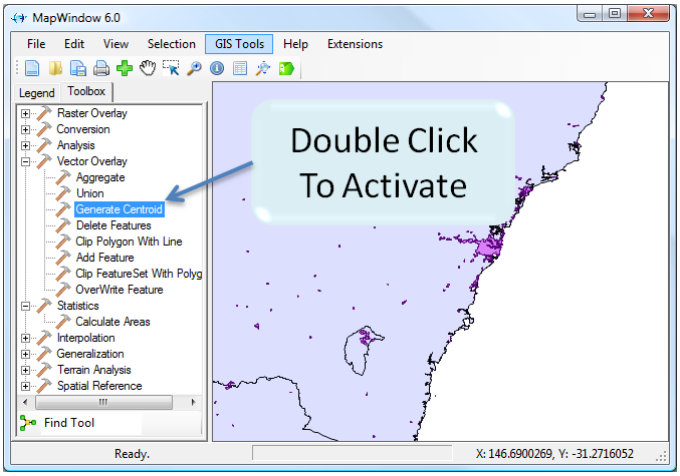


Figure 23: Calculate Centroid

The shapefile created by calculating the centroids will now be a good representation of the city locations for built up areas, and in addition, it will give us an approximate measure of the size of the built up area using the area attribute.

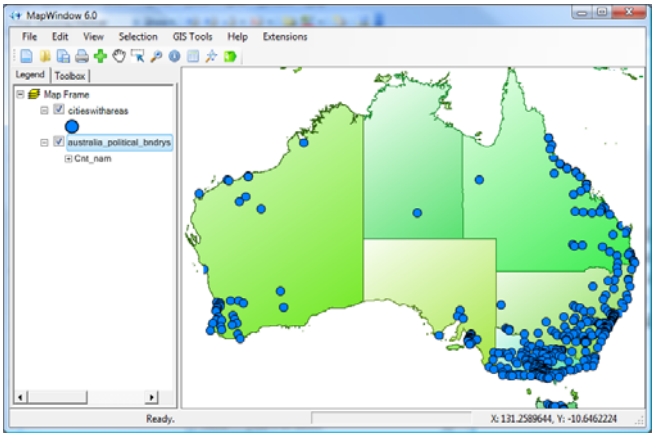


Figure 24: Too Many Cities

1.3.4. Step 4: Sub-sample by Attributes

As can be seen from the figure above, there are a few too many cities visible to make a good map. In order to build a cities shapefile with just the largest cities, we can add the newly created cities shapefile to the map. We can then use the select by attributes ability, and in this case choose [Area] > .01 as the criteria.

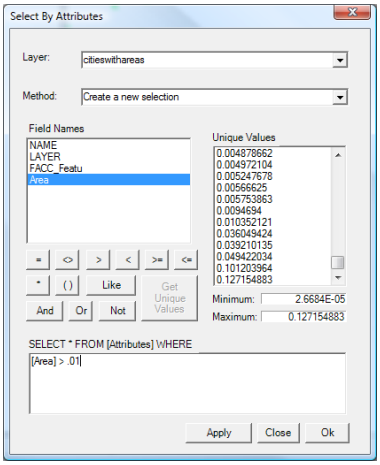


Figure 25: Area > .01

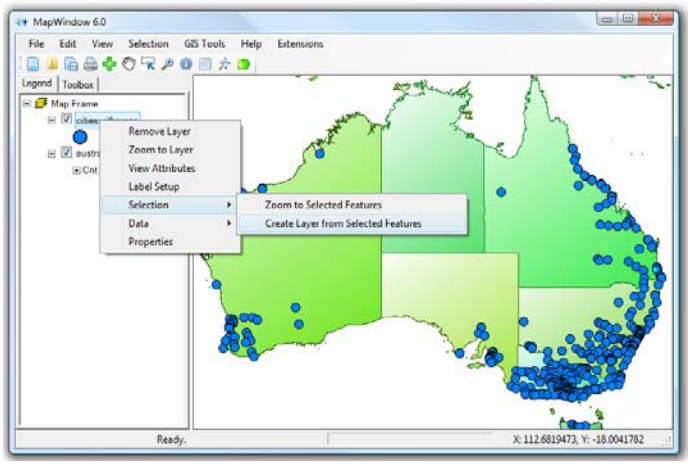


Figure 26: Create Layer

In order to create a new layer with the features we have selected, we can right click on the cities layer in the legend. By highlighting the "Selection" tab in the context menu, we gain the ability to create a layer from the selected features. This will create a new in-memory shapefile that is not yet associated with a true data layer.

1.3.5. Step 5: Export Layer to a File

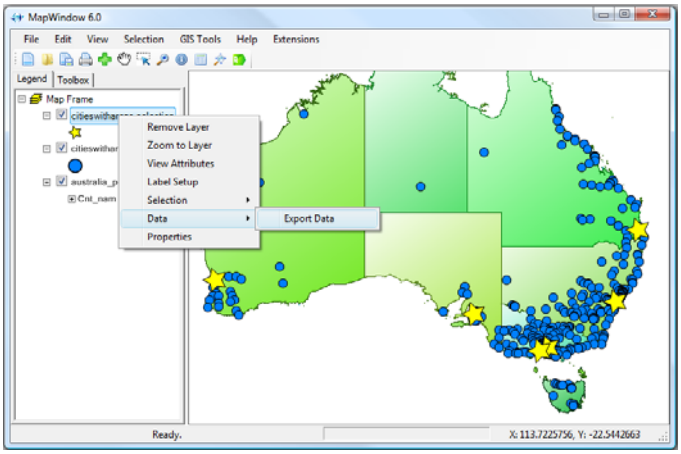


Figure 27: Export Layer

Exporting the layer will save this newly created city layer with just the cities with the largest areas.

1.3.6. Step 6: Repeat with Roads Layer

Because the roads shapefile is actually quite large, we might want to limit that shapefile as well.

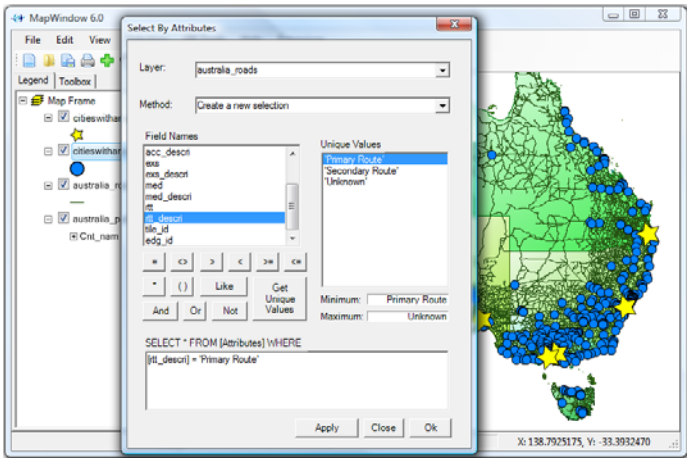


Figure 28: Primary Roads

1.3.7. Step 7: Select Political Bounds by Clicking

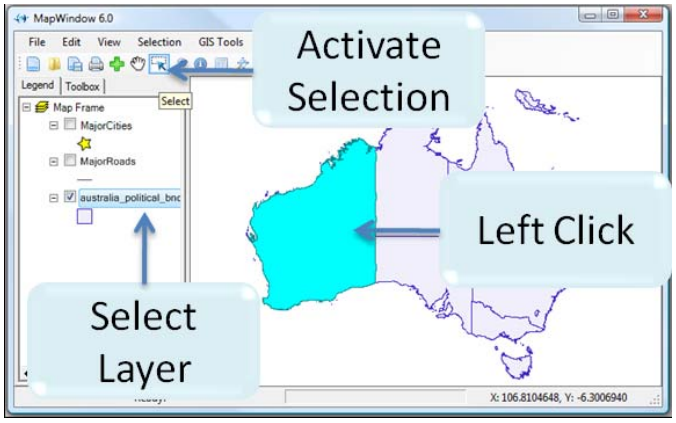


Figure 29: Select Major Areas

Selecting by attributes is not the only way to narrow down the features that you want to use. First, make sure that the layer you want to interact with is selected in the legend. This will prevent content from other layers from being selected at the same time. Next you can activate the selection tool by using the button in the toolstrip indicated in the figure above. Holding down the [Shift] key allows you to select multiple layers at one time. Once you have selected the major polygons, create a new layer the same way by using the create layer from selection option in the legend. Finally, export the data to create a simplified continental shapefile. This will make labeling exercises easier because the result will not be cluttered by labeling the small islands around Australia.

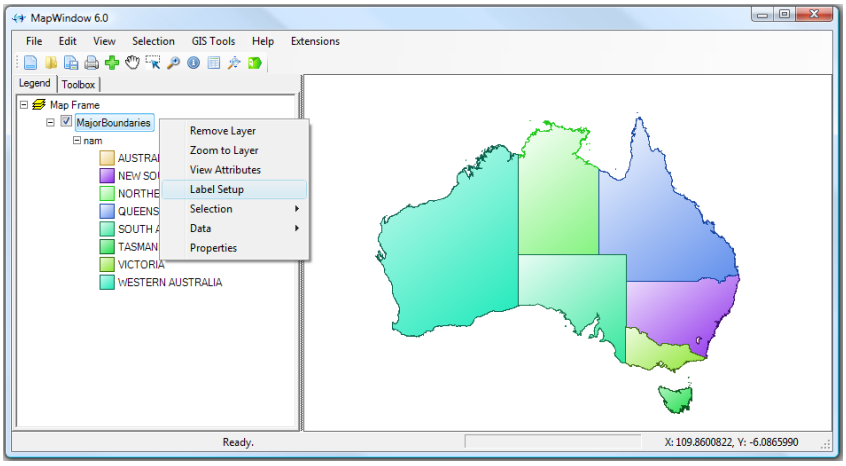


Figure 30: Activate Labeling

1.3.8. Step 8: Apply Labeling

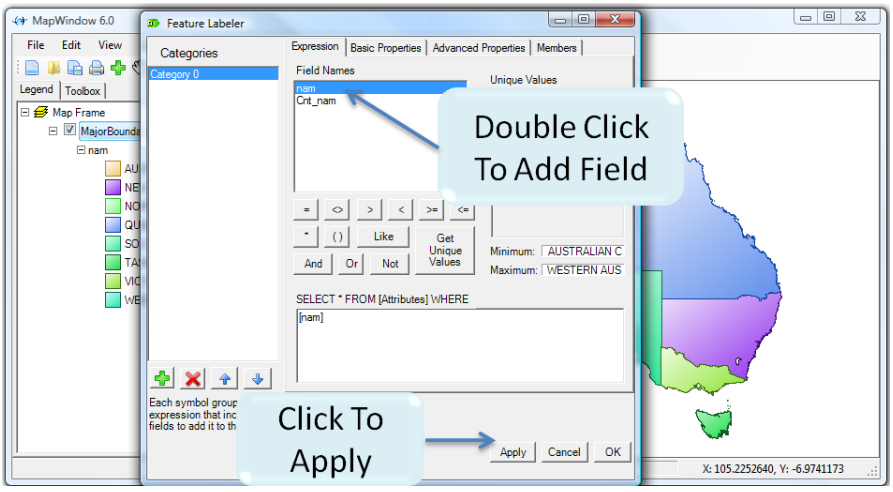


Figure 31: Apply Labels

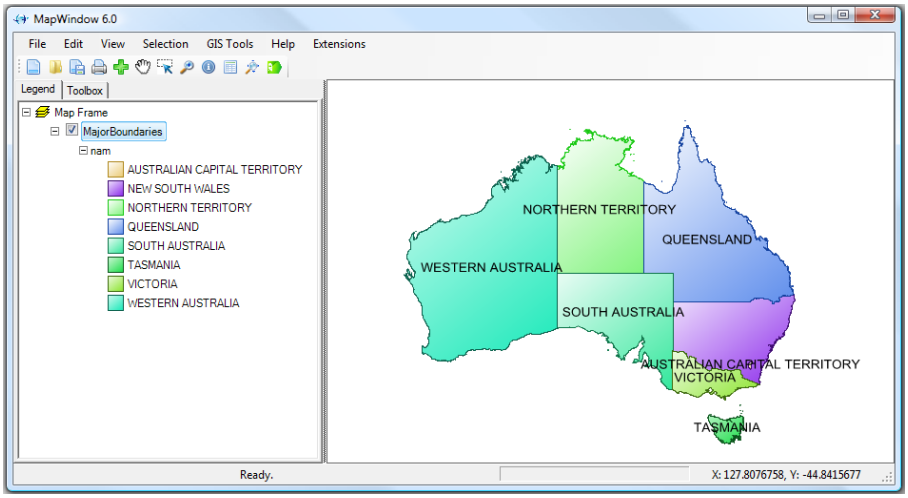


Figure 32: Applied Labels

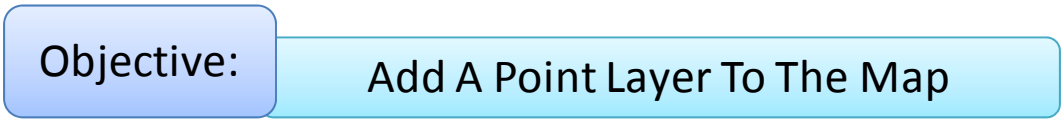
1.4. Programmatic Point Symbology

Unlike the previous section, where we were assuming that the reader would perform each step, in this section, we will explore the most basic features of the extremely powerful symbology toolkit provided by DotSpatial, but listed under completely independent objectives. These 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 you are either writing a plug-in, or else are building your own GIS software using our components. In such a case, you might want to be able to automatically add certain datasets, and control the symbology automatically, behind the scenes.

In previous versions of MapWindow, setting the color of the seventh point in the shapefile to red was extremely simple, but the trade-off was that anything more complex was not inherently supported in the base program. Instead, developers would have to write their own code to make symbolize the layer based on attributes. MapWindow 6 introduces thematic symbol classes that on the surface appear to be much more complicated. However, we will show that accessors have been provided to still allow easy access to the simplest steps, but also provide a basic structure that makes symbolizing by attributes much simpler than in previous versions.

To begin this exercise, we will need the datasets created as part of exercise 2. We will also be working with a copy of the visual studio project that we created in exercise 1, to hammer in the point that you do not need to be working with the MapWindow6 executable, but rather can be working directly with the components in a new project. The first step is to explore adding data to the map programmatically.

1.4.1. Add a Point Layer




```
FeatureSet fs = new FeatureSet();

fs.Open(@"[YourFolder]\Ex3\Data\CitiesWithAreas.shp");

IMapFeatureLayer myLayer = map1.Layers.Add(fs);
```

The three lines of code above are all that is needed to programmtically add the cities with areas shapefile to the map. The first line creates an external FeatureSet class. This 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 won't use the \ character as an escape sequence. You can substitute [YourFolder] with the folder containing these exercises on your computer. Finally, the last line adds the data to the map, and returns a layer handle that can be used to control the symbology. I can add the lines

of code above in the main form by overriding the OnShown method. That way, when the form is shown for the first time, it launches the map. The FeatureSet variable gives us access to all the information stored directly in the shapefile, but organized into feature classes.

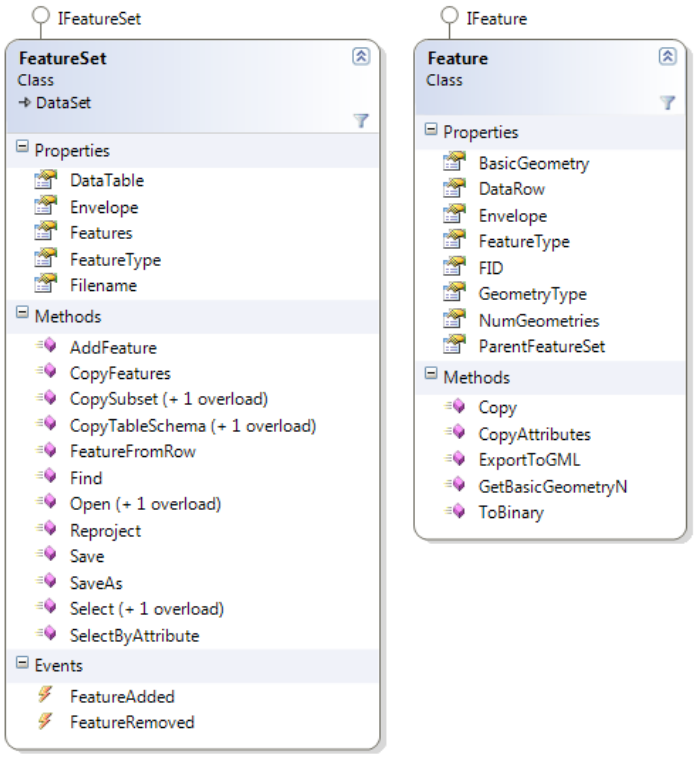


Figure 33: Feature Set and Features

The simplified class diagrams from above give an idea of what kinds of information you can find 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 is enumerable, so you 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 here 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. Because we were interested in making extensible data providers, we did not require these basic geometric classes to support all the mathematical overlay and relate operations that can be found in the various topology suites. Instead, the role of the BasicGeometry is to provide the data only interface.

We did not overlook the possibility of wanting to perform, say, an intersect operation with another feature. Instead of building the method directly into the feature class, we have instead build extension methods so that from a programmatic viewpoint, it will look like any IFeature will be able to perform Intersection calculations.

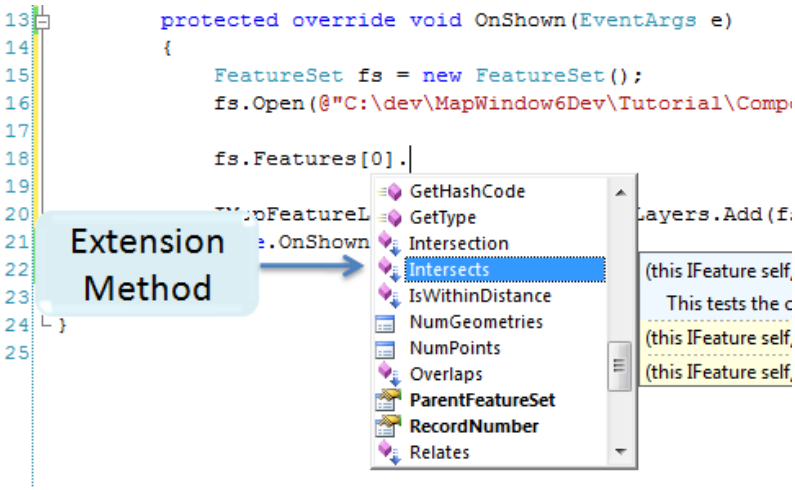


Figure 34: Extension Method

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 you can very rapidly narrow the displayed items and reduce the chance for spelling mistakes. It also gives you an instant browse window to explore the options 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 you have any extensions like Re-Sharper loaded. If XML comments have been generated for the project, (which they have been for MapWindow 6.0) you will not only see the methods, properties and events available, but you will also see help for each of these methods that extends to the right.

Methods are identified by having a purple box. Extension methods are represented by a purple box with a blue arrow to the right of that box. Instead of having the programming code built into the class, the code is actually separate, in an external static method. Using this technology, 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 itself.

1.4.2. Simple Symbols

Objective: Make Yellow Stars

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

Because we know in advance that we are working with points, we don't have to work directly with the existing classes, or use casting. We can simply use the constructor. If we pass myLayer from the earlier code into the method above, we will automatically create the outpoint shown in the following image.

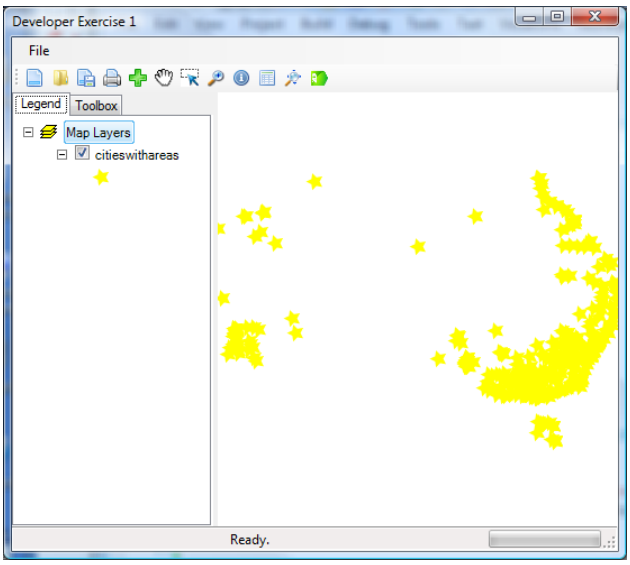


Figure 35: Yellow Star

One thing that you might notice is that the borders of the stars are hard to see because we only specified one color, and that color was the fill color. In order to give the stars black outlines, we need to call a slightly different method.

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

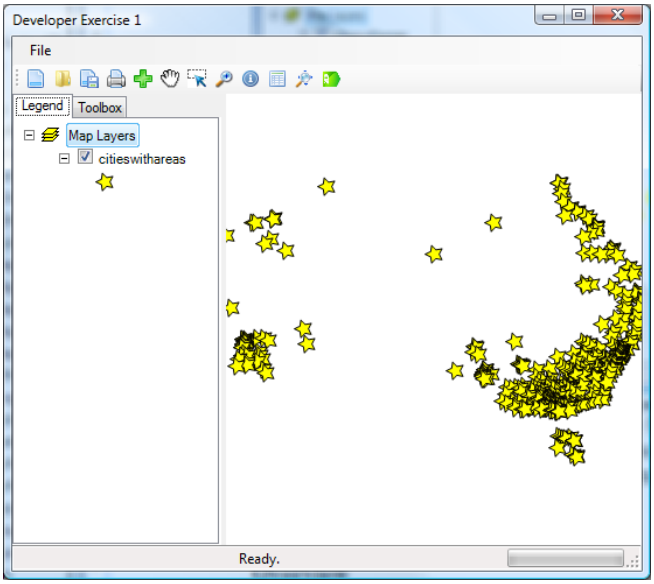


Figure 36: Yellow Stars with Outlines

You will notice that the layer does not control these characteristics directly. Instead, it uses a class called a Symbolizer. Symbolizers contain all of the descriptive characteristics necessary to draw something. They have a few simple accessors that allow us to work with the simple situations like the one listed above. In this situation, we are not worried about a scheme, or complex symbols that have multiple layers. A method like SetOutline may or may not work as expected in every case, since some types of symbols do not even support outlines. However, if we inspect the parameters that we can control above, we already have the basic symbology options that were provided in previous versions of MapWindow.

1.4.3. Character Symbols

Objective: Use Character Symbols

In addition to the basic symbols, MapWindow 6.0 also provides access to using characters as a symbol. This is a very powerful system since character glyphs are vectors, and therefore scalable. They look good even when printing to a large region at high resolution. It is also incredibly versatile. Not only can you use pre-existing symbol fonts (like wingdings) that are on your computer, there are open source fonts that provide GIS symbols. One helpful site that has lots of GIS symbol fonts that can be downloaded for free is found here: http://www.mapsymbols.com/symbols2.html. For this exercise, we are downloading and unzipping the military true type fonts from the site. 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 Control Panel.

Figure 37:Fonts in Control Panel

Clicking on this folder will open 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 you will expose the context menu shown in the following figure. Choose "Install New Font". You will have to browse to your recently downloaded and unzipped .ttf file. In this case we are using the Military.ttf file.

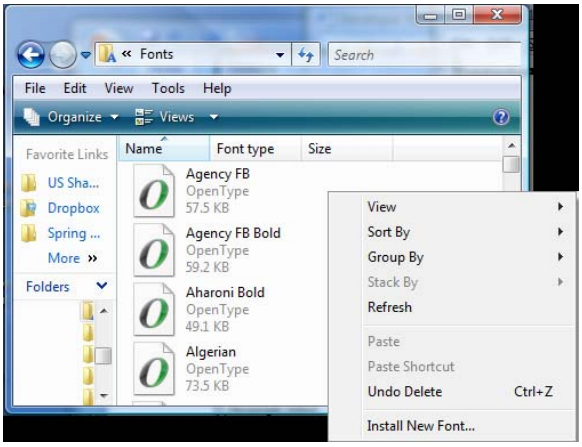


Figure 38: Right Click to Install New Font

We can verify that the new font is available directly by running MapWindow 6.0, or our newly create program, adding a point layer, and then symbolizing with character symbols. To use character symbols, double click on the symbol in the legend. This 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, we want to choose Character.

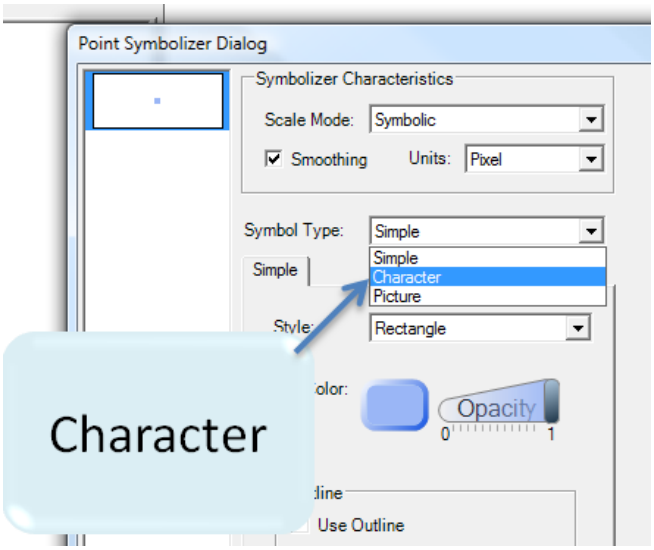


Figure 39: Switch To Characters

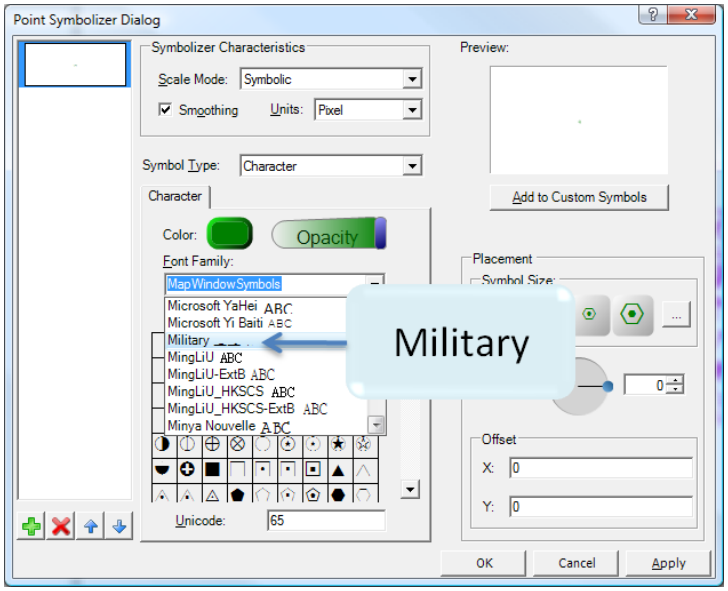


Figure 40: Choose Military

Once you select military, the icons listed below in the character selection drop-down should be replaced with the new military symbols that we have just downloaded. Because many GIS

systems use true type fonts, it is possible for MapWindow to show font types from pre-created professional font sets. Having verified that we have successfully enabled the software to use the new military font type, we will now attempt to use one of these symbols programmatically. To draw planes, we will choose the character M, which draws a character of a plane, and specify the use of the Military font name. We can also specify that they will be blue and will have a point size of 16.

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

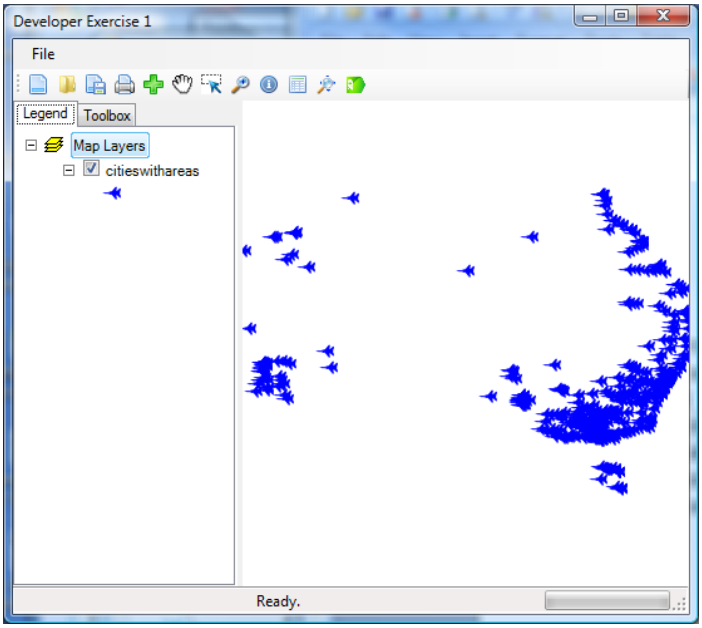


Figure 41: Military Plane Characters

As a side note, the Military font is slightly abnormal in that it has a huge amount of 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, this 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 good commercial example is Font Creator, which has a 30 day free trial and can be purchased for about $100 here: 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.

1.4.4. Image Symbols

Objective:	Use Image Symbols

Sometimes, it simply isn't possible to work with fonts, however, and an image is preferable. In such a case, you can use almost any kind of image. Remember that if you have lots of points, it is better to be working with smaller images. As an example, you can download this wiki-media tiger icon to work with:

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

To use this symbol programmatically, first download the image to a file. Once you have saved the image file, you reference it using either standard file loading methods for images, or else you can embed the file as a resource to be used. We will look at embedding the image as a resource. First, from solution explorer, add a resource file named Images.

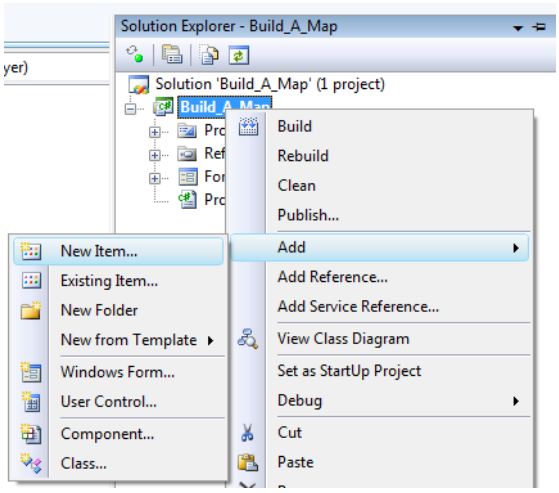


Figure 42: Add a New Item

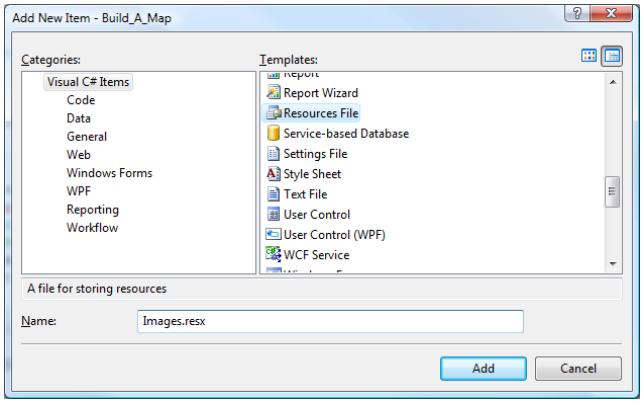


Figure 43: Images.resx Resource

Choose Resources File from the available Templates, then name the resource file Images, and click Add. From solution explorer, simply double click the newly added Images resource file to open it. Adding it the first time should automatically open the resource file as a tab. Under the Add Resource option in the toobar just below the Images.resx tab, choose "Add Existing File…"

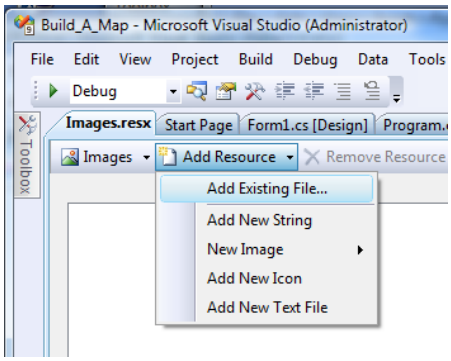


Figure 44: Add an Existing File

Then simply browse to the file we just downloaded, and add it to the resource file. To make it easier to find, we can rename the file "Tiger".

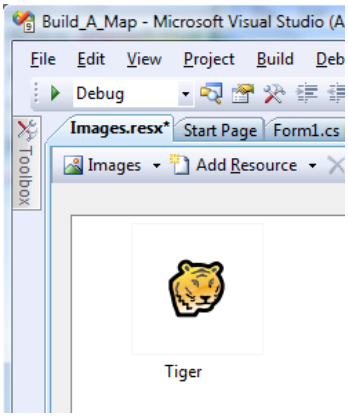


Figure 45: Rename to Tiger

Now, we can programmatically reference this image any time using the Images.Tiger reference. Be sure to build the project after adding the image to the resource file, or else the Intellisense will not show the Tiger image yet. Adding the image to a resource file sets up the framework for the following line of code where we will specify to use the Tiger image for the point symbology:

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

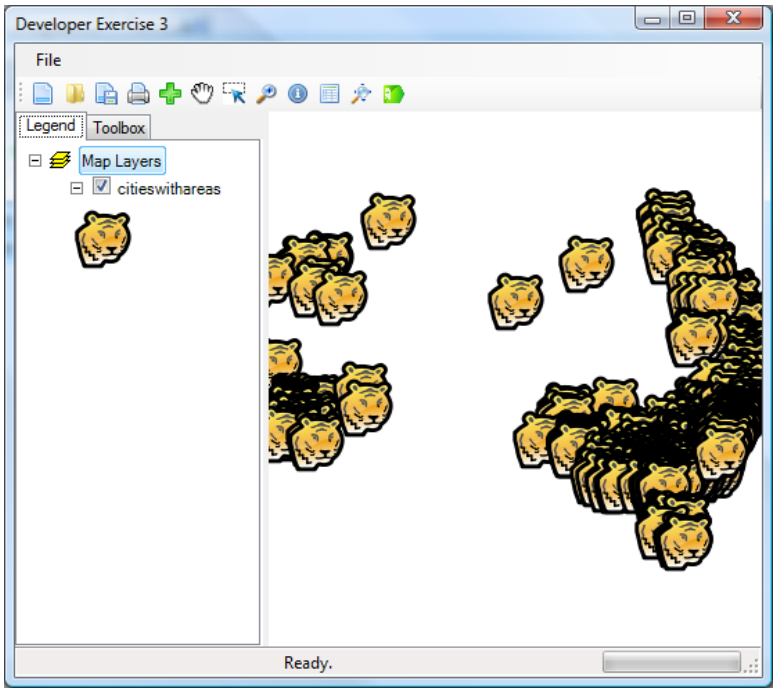


Figure 46: Tiger Images

1.4.5. Point Categories

Objective: Use Point Symbol Categories

For the next topic, I will introduce the concept of casting. In many cases, methods or properties are shared between many different types of classes. When that is true, the interface may provide the shared base class, instead of the class type that is actually being used. For instance, if you are working with points, the Symbolizer that is being used should be a PointSymbolizer. This has a shared base class with the FeatureSymbolizer.

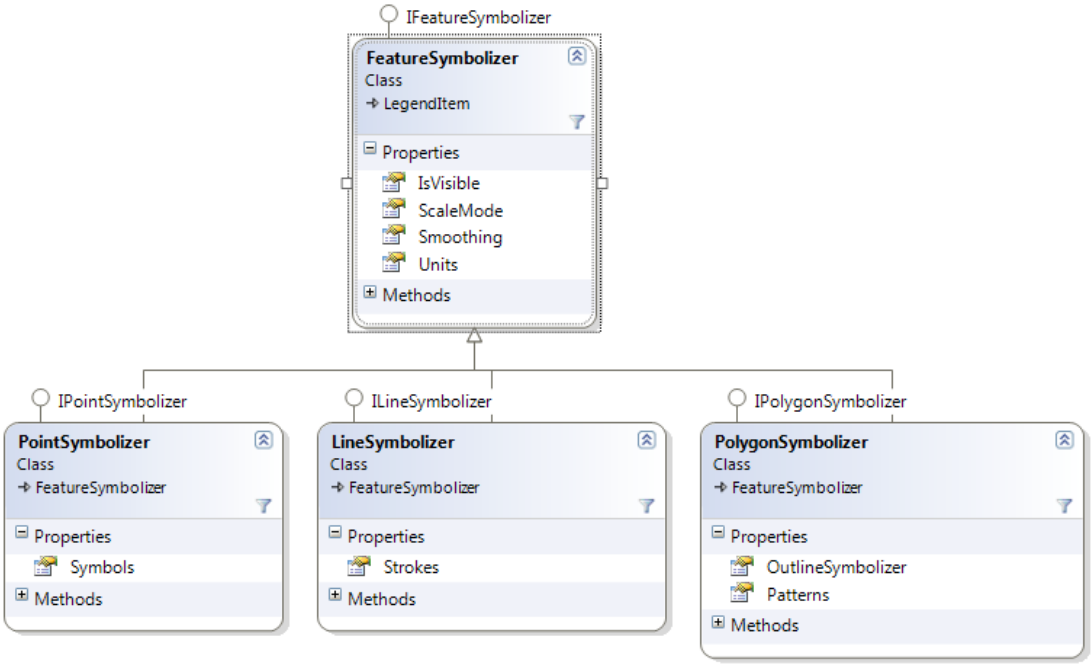


Figure 47: Symbolizer Class Diagram

To illustrate inheritance, the class diagram above shows the three main feature symbolizers, one for points, one for lines, and one for polygons. What is most important here is that there are some characteristics that will be shared. Properties, methods and events on the FeatureSymbolizer class will also appear on each of the specialized classes. Meanwhile, each individual type has a collection of classes that actually do the drawing, but these classes are different depending on the class. In the same way, we can set up categories, but it works much more easily if we know what kind of feature layer we are working with. When working with categories, schemes, and so-on, knowing that we are working with a point layer is the difference

between having to cast every single object every time, and only having to cast the feature layer once.

We will be symbolizing based on the Area field. The areas were calculated from the polygons in exercise 2, so we can be assured that the cities with areas shapefile that we are adding has an Area field, and looking at the table below, we can see that 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 48: Large Area Cities

The source code that we are going to use has several parts to it. First, we are going to cast the layer to a MapPointLayer so that we know we are working with point data. After that, we create two separate categories, using filter expressions to separate what is drawn by each category. Finally, we add the new scheme as the layer's symbology. When the map is drawn, it will automatically show the different scheme types in the legend using whatever we specify here as the legend text.

```
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;
```

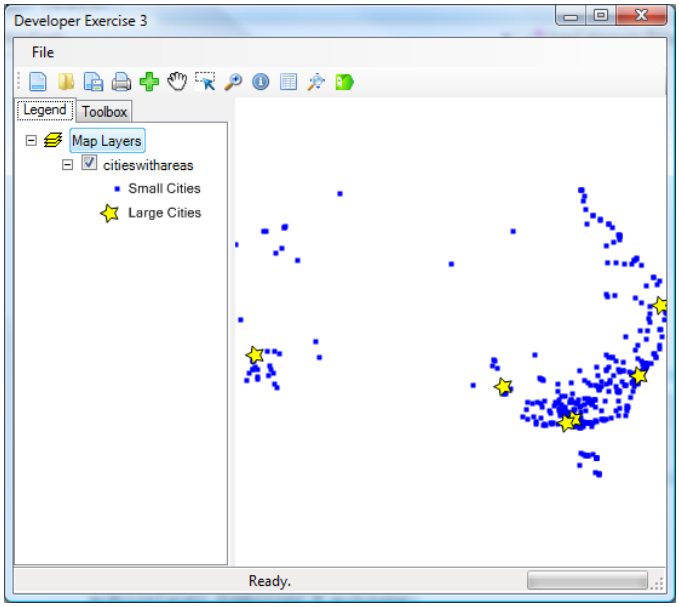


Figure 49: City Categories by Area

The square brackets in the filter expression are optional, but recommended to help clarify fieldnames in the expression. What is significant here is that we did not have to write code to actually loop through all of the city shapes, test the area attribute programmatically, and then assign a symbol scheme based on the character. Instead, we simply allow the built in expression parsing to take over and handle the drawing for us. This allows for programmers to work with the objects in a way that directly mimics how users work with the symbology controls. And just like the layer dialog controls allow you to specify schemes; those schemes can also be controlled programmatically.

```
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;
```

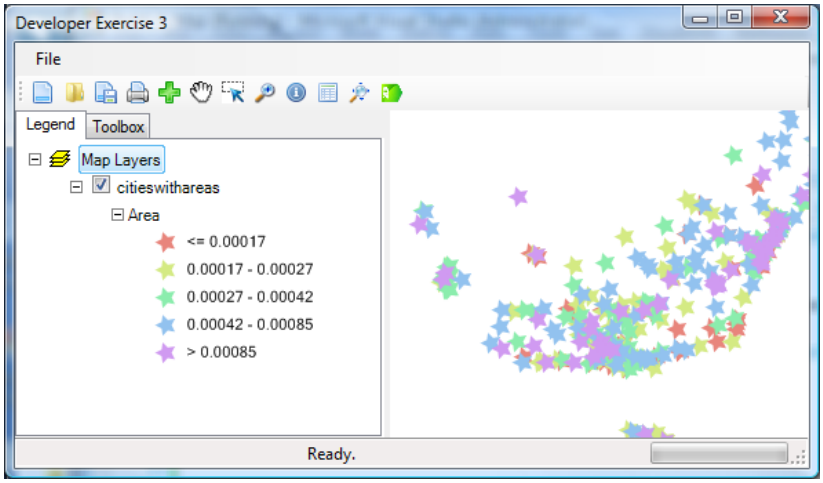


Figure 50: Quantile Area Categories

There are a large number of settings that can be controlled directly using the PointScheme. In this illustration the classification type is quantities, but this can also be UniqueValues or custom. The categories can always be edited programmatically after they are created, but this simply controls what will happen when the CreateCategories method is ultimately called. As of the Sydney alpha release (Oct. 2009) not every IntervalMethod is supported, but Quantile and Equal Interval are both supported. The interval snap methods include none, rounding, significant figures, and snapping to the nearest value. These can help the appearance of the categories in the legend, but it can also cause trouble. In this particular case, we have to set the rounding digits to a fairly high number (5) or else many categories simply show a range of 0 – 0 and have no members. With Significant figures, the IntervalRoundingDigits controls the number of significant figures instead. One property is deceptive in its power. The TemplateSymbolizer property allows you to control the basic appearance of the categories for any property that is not being controlled by either the size or color ramping. For example, if we wanted to add black borders to the stars above, we would simply add that to the template symbolizer. In this case we chose to make them appear as stars and controlled them to have equal sizes since UseSizeRange defaults to false, but UseColorRange defaults to true.

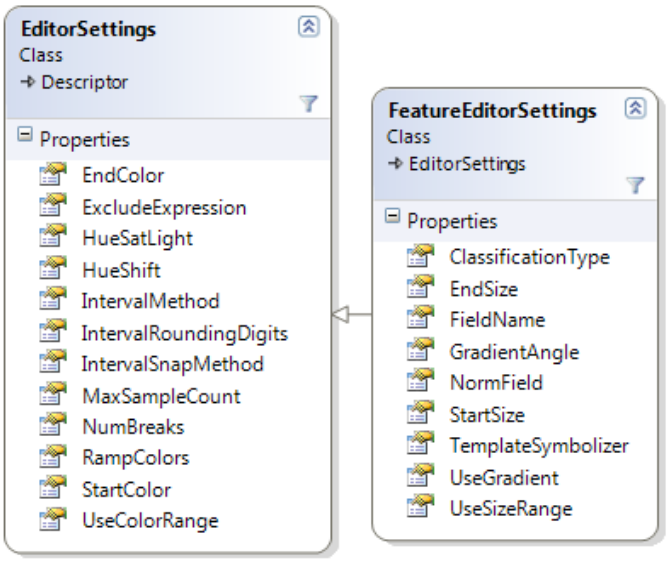


Figure 51: Available Feature Editor Settings

The settings shown in the exercise above represent a small taste of the scheme options that are programmatically available. You can also 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.

1.4.6. Compound Symbols

Objective: Yellow stars in a Blue Circle

One of the new additions to how symbols work is that you are no longer restricted 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 kinds of symbols, Simple, Character and Image based. These 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. For creating new symbols, the Subclass can be used. For working with individual symbols in the collection, you may need to test what type of symbol you are working with before you will be able to control its properties.

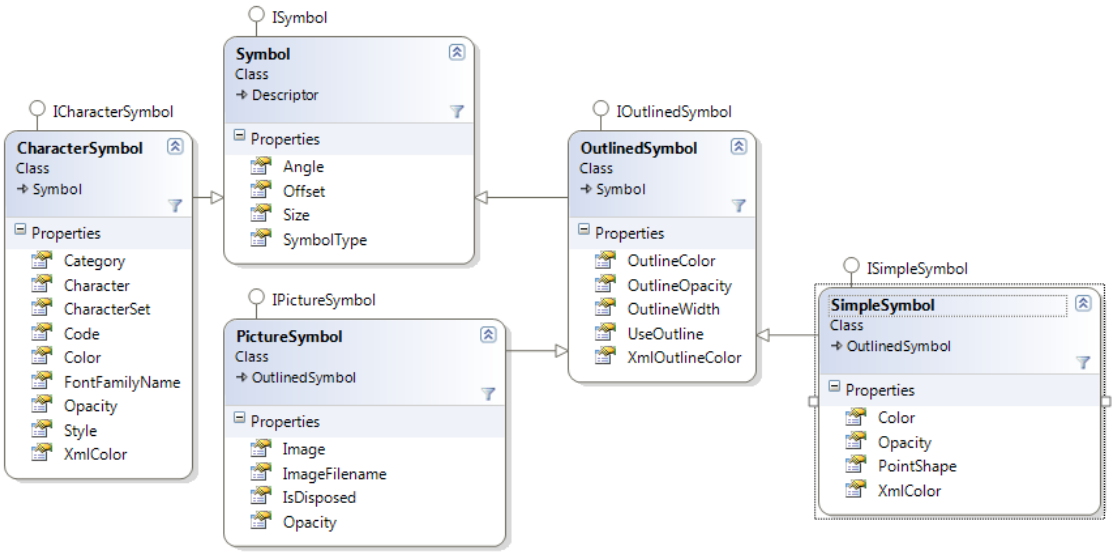


Figure 52: Point Symbol Class Diagram

The class diagram above shows the organization of the individual symbols.

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

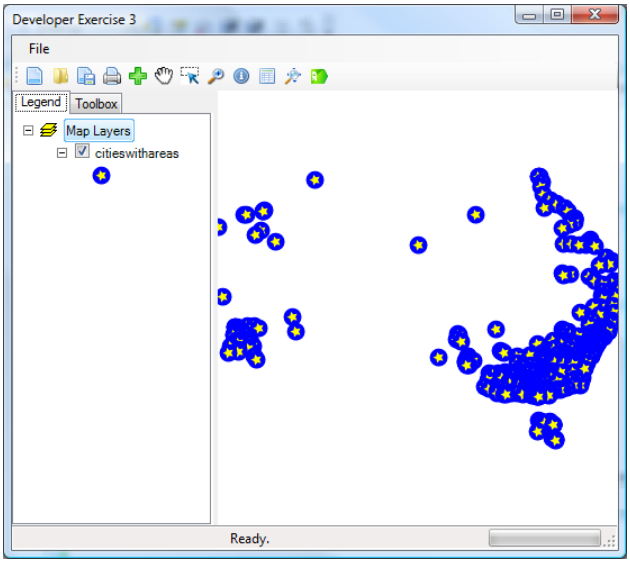


Figure 53: Blue Circles with Yellow Stars

1.5. Programmatic Line Symbology

1.5.1. Adding Line Layers

Objective: Add Line Layer

Line layers operate according to the same rules as points for the most part, except that instead of individual symbols, we can have individual strokes. The default symbology is to have a single line layer of a random color that is one pixel wide.

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

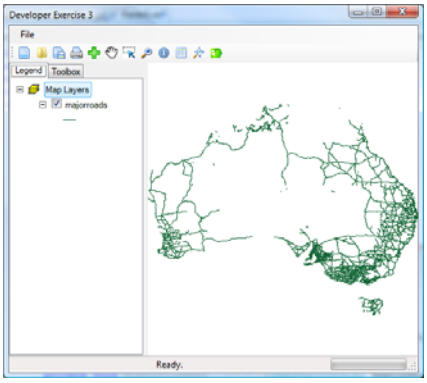


Figure 54: Add Line Layer

Simple Line symbols

Objective: Brown Roads

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

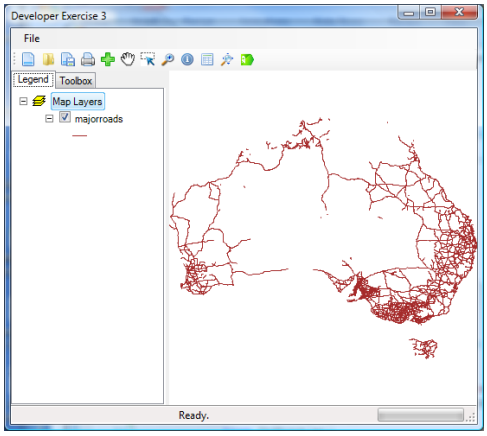


Figure 55: Brown Lines

1.5.2. Outlined Symbols

Objective: Yellow Roads with Black Outlines

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.

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

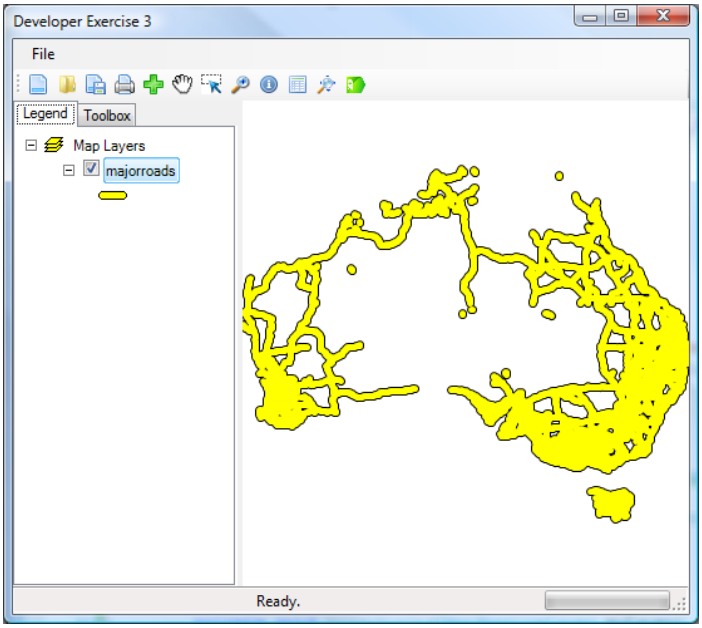


Figure 56: Yellow Roads with Outlines

1.5.3. Unique Values

Objective: Roads Colored by Unique Tile ID

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 this is true, then the ramp is created by adjusting the hue, saturation and lightness between the start and end colors. If this is false, then the red, blue and green values are ramped instead. In both cases, alpha (transparency) is ramped the same way.

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

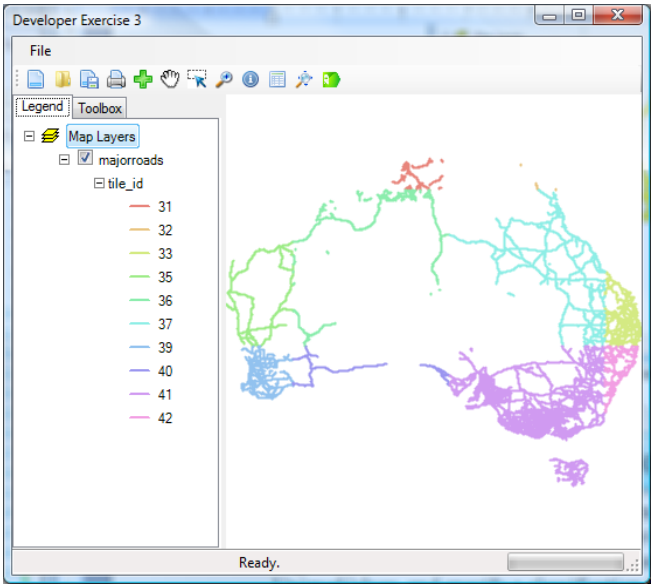


Figure 57: Roads with Unique Values

1.5.4. Custom Categories

Objective: Custom Road 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. This can also be accomplished manually by controlling the "AppearsInLegend" property on the scheme. If this is false, the categories will appear directly below the layer. When it is true, you can control the text in the legend using the scheme itself. Showing this principal in action, in this sample we will show the code that will programmatically set up two categories, and also have them appear under a scheme in the legend.

```
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;
```

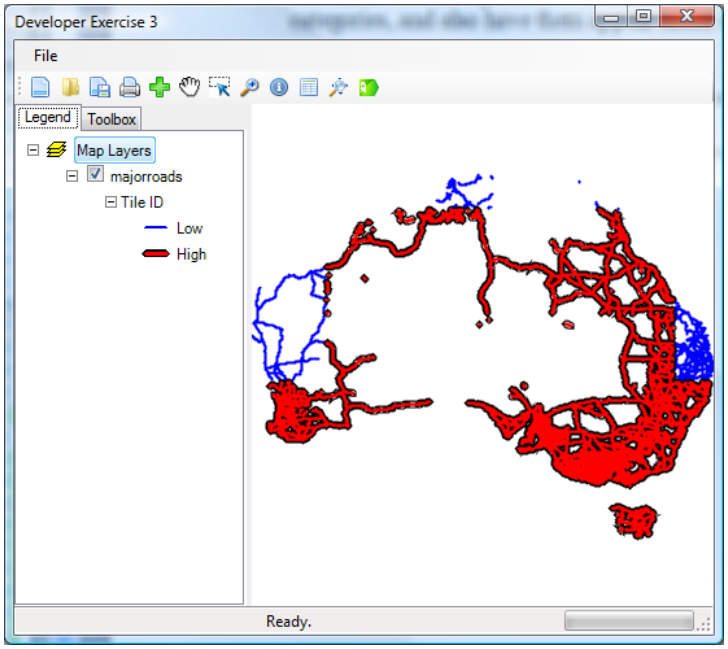


Figure 58: Custom Line Categories

1.5.5. Compound Lines

Objective: Lines with Multiple Strokes

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.

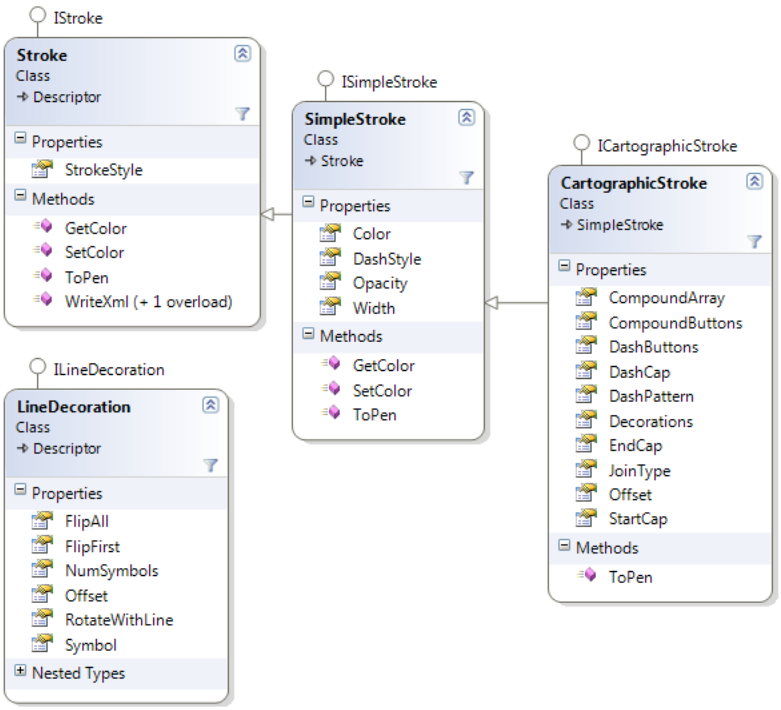


Figure 59: Stroke Class Hierarchy

In this example, we will take advantage of several powerful symbology options. We will use cartographic strokes in order to create two very different type of line styles. In the first case, we will create brown railroad ties. Using the standard "Dot" option for a simple cartographic line would 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 is convenient, since for our repeating ties, we really only need to specify two numbers.

The second layer of the symbol will be dark gray rails. In this case, the dash pattern is continuous, so we will not need to change it. However, the rails don't persist the whole way across the line the way the ties do. Instead, we want to have two thin lines that appear along the path width. To do this, we take advantage of a CompoundArray. With the compound array, you are expressing the actual offsets for the start and end positions along the compound array, 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 try to ensure that the width of the lines represented in the Compound array work out to be just slightly larger than 1 to ensure that the lines ultimately 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 kind of rounded, buffered look to roads that are wider than one pixel.

```
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;
```

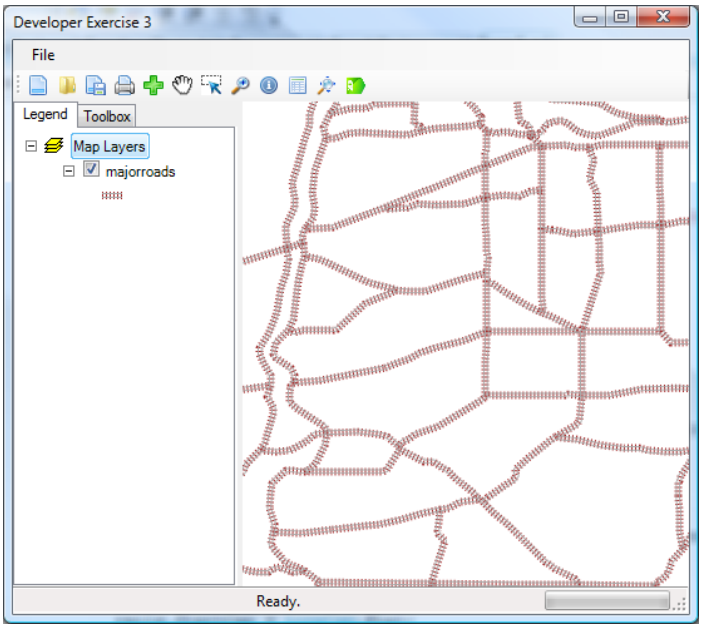


Figure 60: Multi-Stroke RailRoads

1.5.6. Line Decorations

Objective: Lines Decorated by Stars

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, we will be adding yellow stars to a blue line.

```
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);
myLayer.Symbolizer = starLine;
```

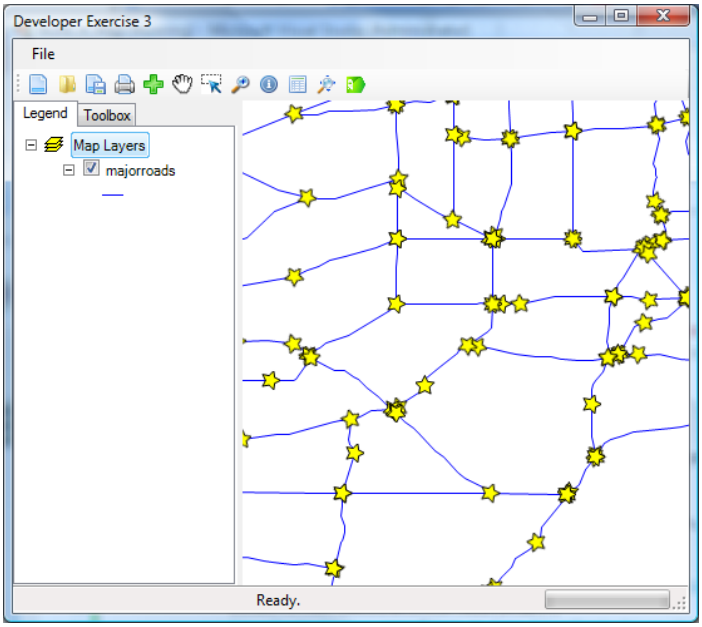


Figure 61: Lines with Star Decorations

1.6. Programmatic Polygon Symbology

1.6.1. Add Polygon Layers

Objective: Add a Polygon Layer to the Map

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 Hawaii, which has several islands, as a single shape, you would use a MultiPolygon instead. A MultiPolygon is still considered to be a geometry and will respond to all of the geometry methods, like Intersects. We can add the polygon shapefile the same way that we added the point or line shapefiles.

Polygon symbolizers are slightly different from the other two symbolizers because in the case of polygons, we have to describe both the borders and the interior. 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. This is a similar strategy to re-using the PointSymbolizer in order to describe the decorations that can appear on lines.

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

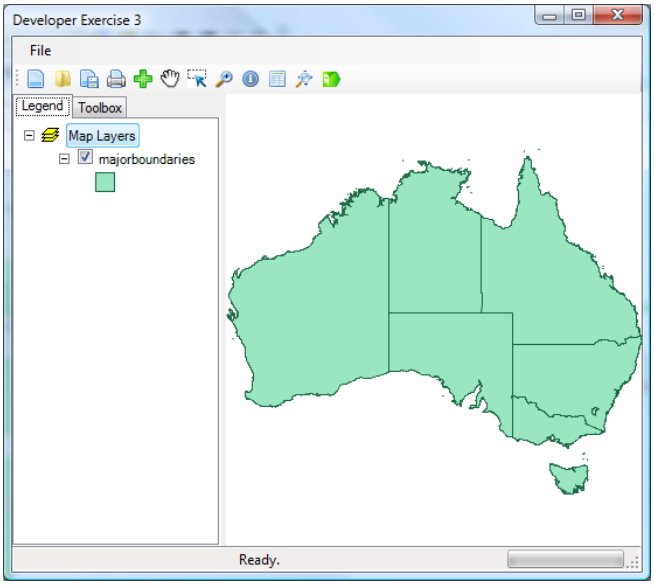


Figure 62: Add Major Boundaries

1.6.2. Simple Patterns

Objective: Specify Blue Polygons

The simplest task with polygons is to set the fill color for those polygons. You will see that specifying only an interior fill creates a continuous appearance, since the normal boundaries are adjacent and all the same color.

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

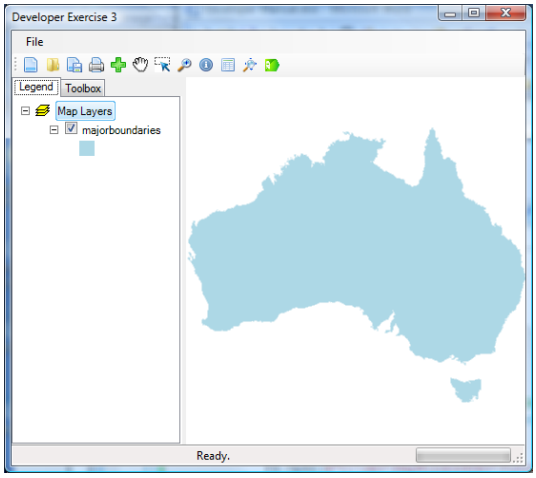


Figure 63: Blue Fill Only

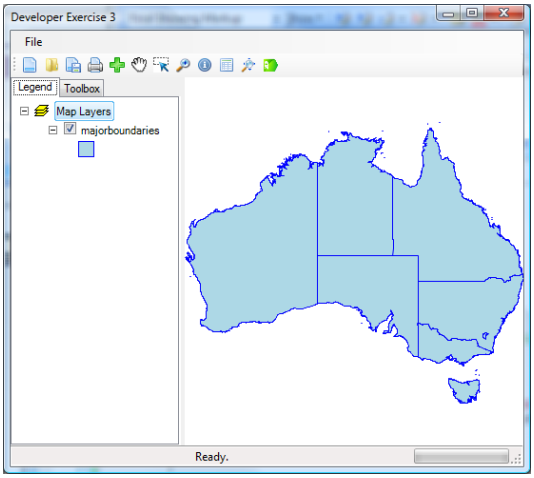
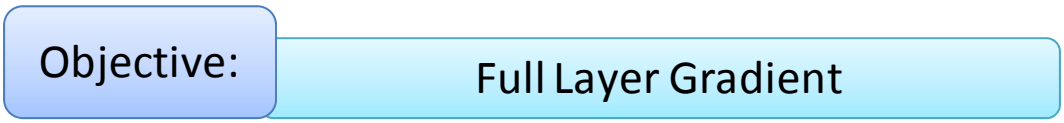


Figure 64: With Blue Border

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

1.6.3. Gradients



One of the more elegant symbology options is to apply a gradient. These can vary in type from linear, to circular to rectangular, with the most frequently used type of gradient by far being linear. If you want to apply a continuous gradient across the entire layer, you can use the default category and simply specify the symbolizer. Unlike the previous example where we directly set up the outline symbolizer, in this example we are taking advantage of the shared method "SetOutline" which does the same thing. For points, this method controls the symbols themselves. For lines, this adds a slightly larger stroke beneath the existing strokes. For polygons, this 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.

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

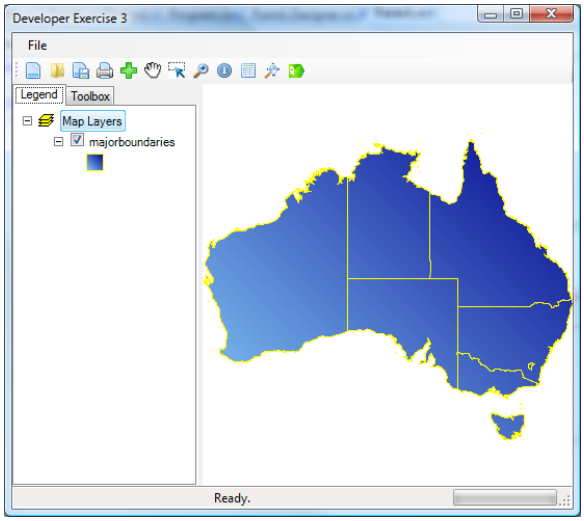


Figure 65: Continuous Blue Gradient

1.6.4. Individual Gradients

Objective: Shape Specific Gradients

Another possible symbology is to create the gradients so that they are shape specific. This is not really recommended in the case of large numbers of polygons because the drawing gets linearly slower for each specific drawing call. You can draw thousands of polygons with one call by having only one symbolic class to describe all the polygons. In the case of a few hundred classes, this distinction is not really noticeable. To rapidly create different categories, we can take advantage of the "nam" field which is different for each of the major shapes that we selected as part of exercise 1 in order to create the basic polygon shapefile.

	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 WESTERN AUSTRALIA	5

Figure 66: Major Boundaries Fields

```
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;
```

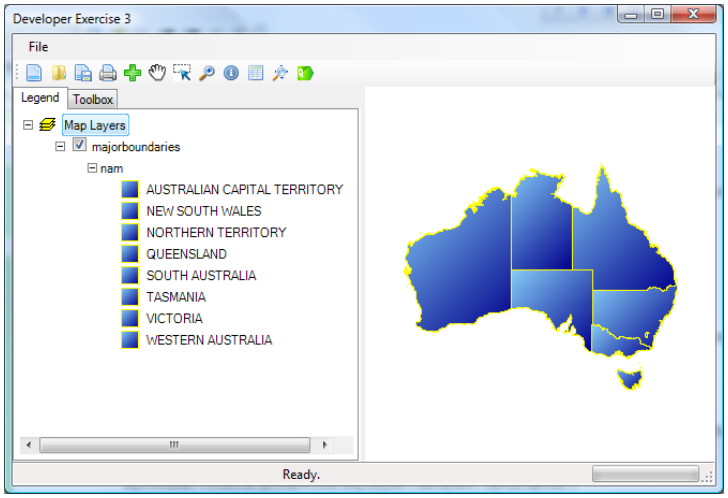


Figure 67: Individual Gradients

1.6.5. Multi-Colored Gradients

Objective: Cool Colors with Gradient

One thing that may seem less than obvious is that in the previous exercise we specified that the UseGradient property should be false. This does not prevent the template symbolizer from having a gradient. Instead, it prevents the symbolizer from overriding the original, presumably simpler, template with a gradient symbol. The gradient symbol will be calculated using a color from the color range, but then will make the upper left a little lighter and the lower right a little darker. That way, you can have the same subtle gradient applied, but still use different colors for each category. To boot, the default polygon symbolizer has a border that is the same hue, but slightly darker, which tends to create a nice outline color.

```
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;
```

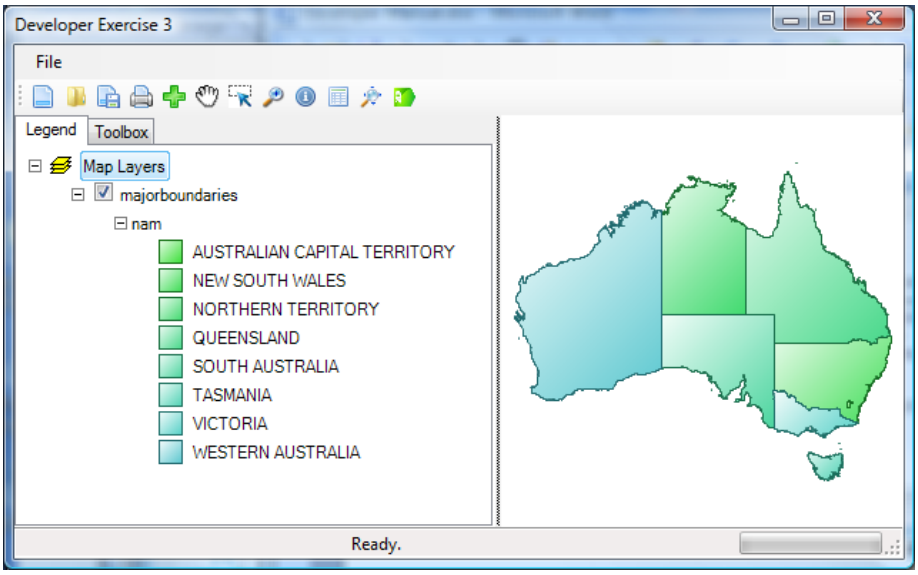


Figure 68: Unique Cool Colors With Gradient

1.6.6. Custom Polygon Categories

Objective: Custom Polygon Categories

An important terminology difference here that we have been using is the difference between Symbolizer and Symbology. With Symbology, we are always referring to a scheme, which can have many categories. With a Symbolizer, we are talking about controlling how those shapes are drawn for 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 you have several categories, it may be better to control the symbolizers explicitly than to use the shortcut. Labels have been 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.

```
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;
```

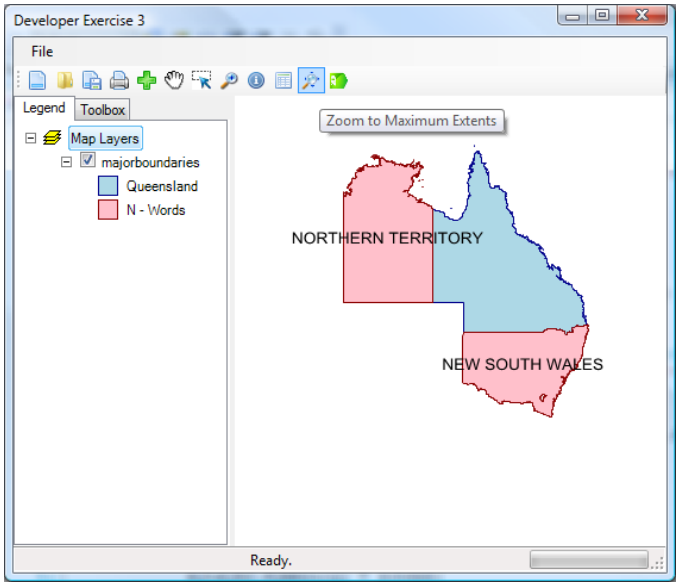


Figure 69: Custom Categories

1.6.7. Compound Patterns

Objective: Multi-Layer Patterns

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.

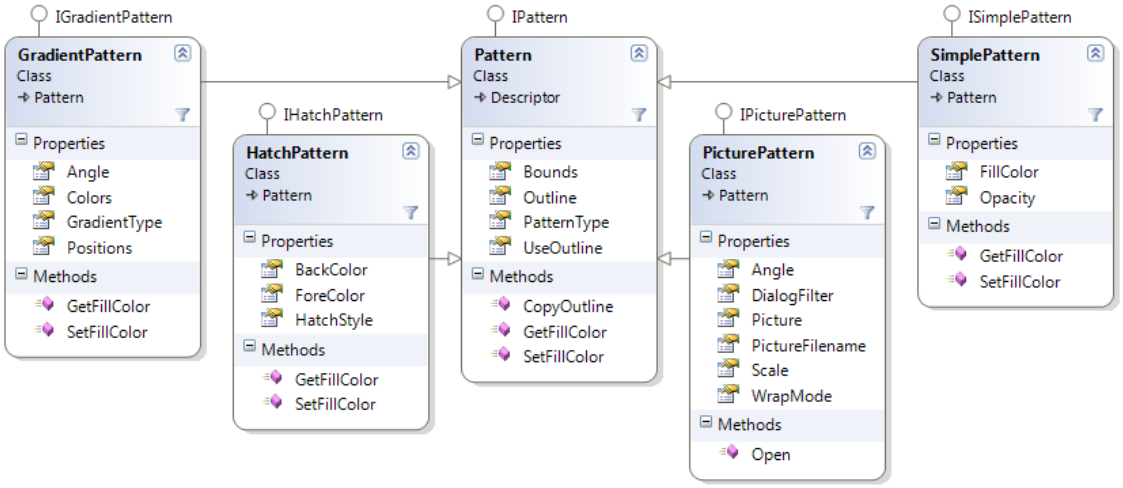


Figure 70: Pattern Class Diagram

One thing in particular to note about this next example is that in all the previous examples with multiple patterns, we simply cleared out the default pattern that was automatically created as part of the symbolizer. When we add a new pattern, the new pattern gets drawn on top of the previous patterns, so the last pattern added has the highest drawing priority. In the code below, the new pattern has its background color set to transparent, yet in the image below we see that 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.

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

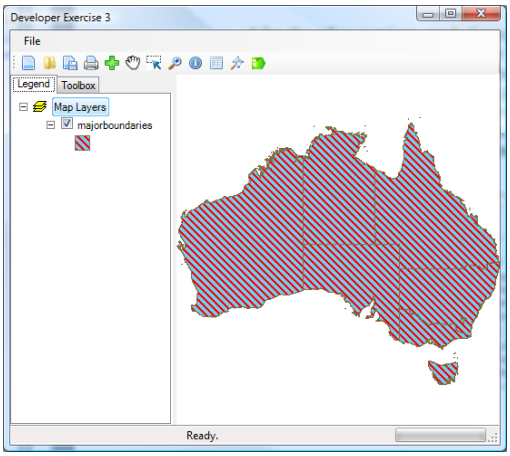


Figure 71: Hatch patterns

1.7. Programmatic Labels

Objective: Labels

For the first example with labels, we will show adding your own text in a way so that the same text gets added to all the features. This uses the default settings, and you can see from the default settings that there is no background, and each label has the left side aligned with the center point by default.

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

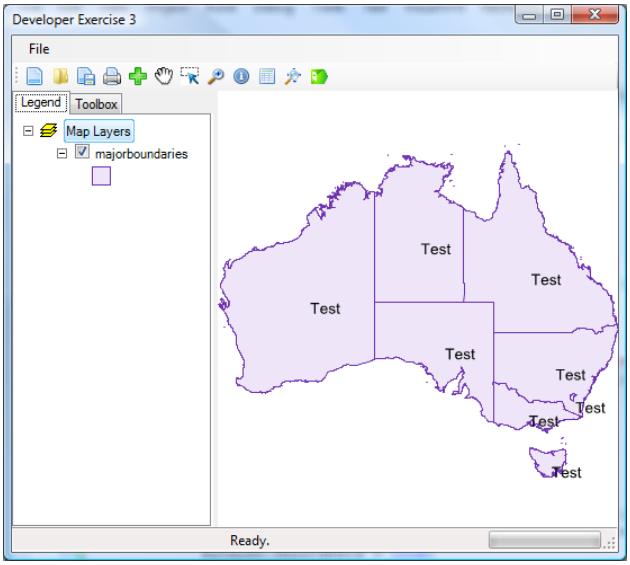


Figure 72: Adding Test labels

1.7.1. Field Name Expressions

Objective: Field Name Labels

The field name in this case describes the name of the territory. In order for the name field to appear in the label text, we simply enclose it in square brackets. This puts together a versatile scenario where you can build complex expressions with various field names. You can also use escape characters to create multi-line labels.

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

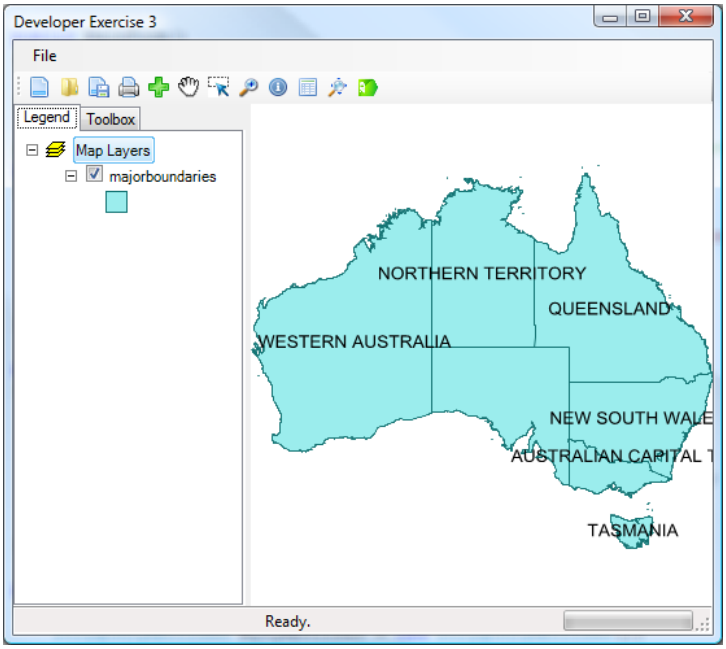


Figure 73: Field Name Labels

1.7.2. Multi-Line Labels

Objective: Multi-Line Labels

Creating multi-line labels is simple, since all you have to do is use the standard .Net new-line character, which in C# is added using the /n, while in visual basic you would combine the two strings 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 allows us to control which labels are added.

```
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;
```

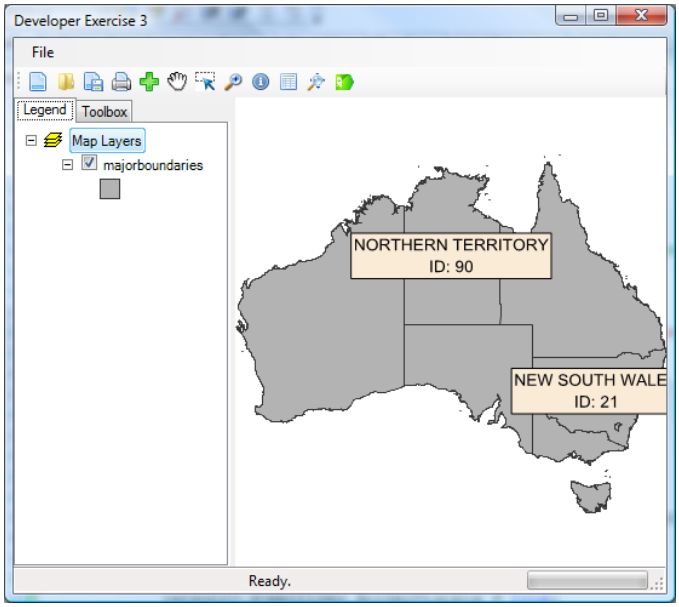


Figure 74: Multi-Line Labels

Multiple label categories can be created and added to the scheme in the same way that the featuresets were able to add different categories. The labels also allow for the font to be controlled, as well as the text color, background color and opacity of either.

1.7.3. Translucent Labels

Objective: 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. Frequently, there are opacity properties available in addition to the actual color, but that is just there for serialization purposes and is simply a shortcut to the alpha channel of the color structure. 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” which doesn’t work with .Net DataTables. For the negative you could use “NOT [nam] is null”.

```
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;
```

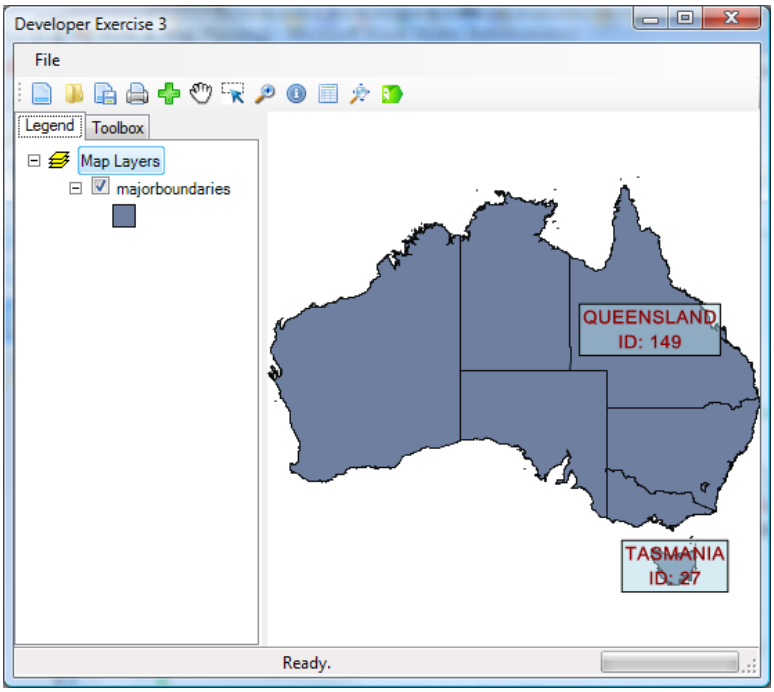


Figure 75: Translucent Labels

1.8. Programmatic Raster Symbology

1.8.1. Download Data

Objective:	Download a Raster Layer

In addition to the vector data that we have been looking at so far, MapWindow also supports a number of raster formats. Rasters are considered distinct from images for us in that the visual representation that we see is derived from the values much in the way that we derive the polygon images from the actual data. With rasters, you typically have a rectangular arrangement of values that are organized in rows and columns. For datasets that don't 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 at the time you are reading this. We have created a plug-in that is exclusive to the windows platform using Frank Warmerdan's GDAL libraries and C# linkage files. The plug-in exposes many of the raster and image types supported by GDAL to MapWindow 6, or any project that adds our ApplicationManager component. While the situation may change before the beta release, for the Sidney Alpha, only one grid format is supported natively (that is without using GDAL) and that is a bgd format. This is not a problem for independent developers because of our clever system that enables you to add a single ApplicationManager to the project and empower your own project with all the data format extensions that work with MapWindow 6. However, since that won't be covered at this stage of the tutorials, we will use the MapWindow 6 application to convert a raster file to the format we need. A good provider of GIS data is the USGS. They provide many forms of data from around the world, but in this case we are interested in an elevation dataset. A useful web utility for browsing the web is called EarthExplorer. http://edcsns17.cr.usgs.gov/EarthExplorer/

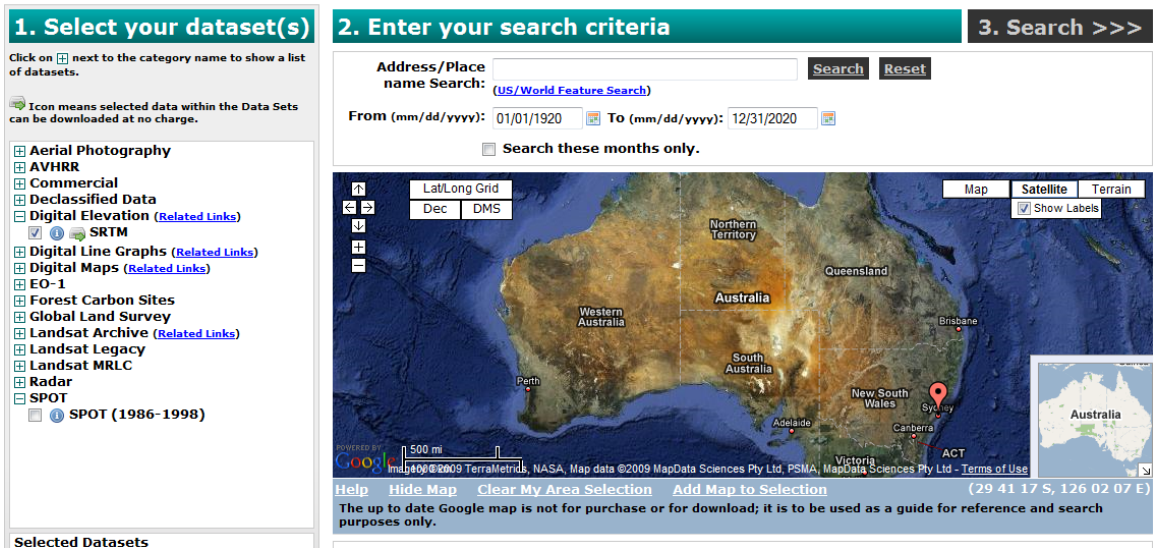


Figure 76: Earth Explorer

If the link above doesn't work, you can try searching for USGS and looking for links that allow you to search fro data by region. The map application is an easy to use system that provides a 1, 2, 3 Data download experience that is to be envied. Unfortunately, the system is also under heavy demand, so you will have to be conservative with your requests from this site.

To get data, simply zoom into a region of interest and you will see that they have lots of interesting data sets. For this demo, I activated the SRTM data format, which gives us digital elevation information. You will have to be a registered user in order to download data from this site, and not all the data is free. In this case, I downloaded a BIL file for this demonstration, but other formats are available, including DTED. Both formats are supported by MapWindow by using GDAL. The downloaded file will be zipped, so you will have to unzip the file before you can open it with MapWindow 6. Before we can work with the data programmatically, we will use MapWindow 6 to change the data format to bgd. This is actually as simple as opening the BIL, right clicking on the legend and choosing export data. Don't worry if the coloring looks strange. It is usual for the no-data values to distort the coloring until they can be properly excluded from the symbology. For now, we are only concerned with converting the format to an *.bgd file.

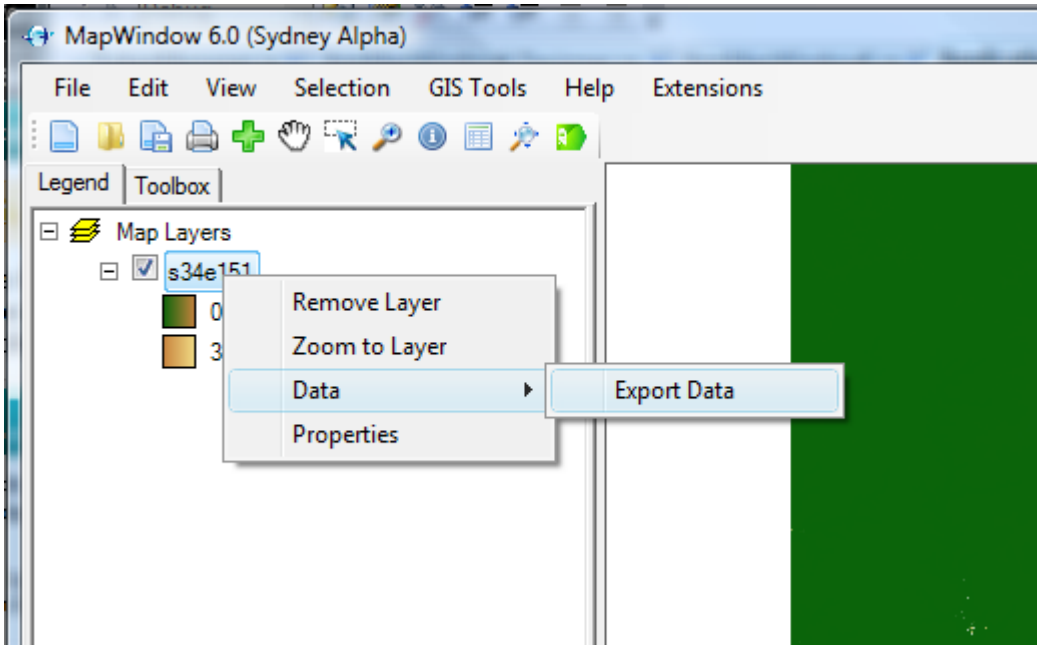


Figure 77: Export Data

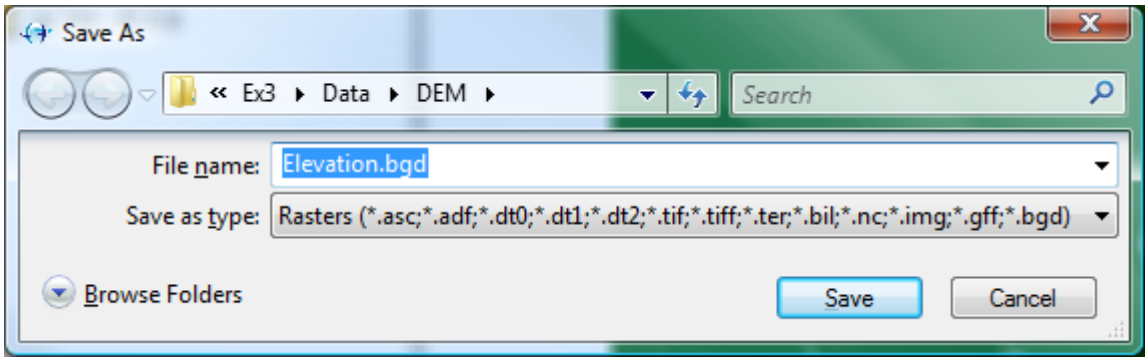


Figure 78: Save as Bgd

You will see lots of other formats available, and these are the write formats that have been exposed by the GDAL plugin, plus any formats that are supported for rasters by other providers. You will notice that *.bgd is at the end of the list, and that is the provider that we will be using for the rest of the demonstration. Save the downloaded file as a bgd file.

1.8.2. Add a Raster Layer

Objective:	Add A Raster Layer

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

One thing that should be fairly obvious right away is that while 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 you direct access to the numerical data stored in grid form. Inheritance at different levels is certainly used, but most of that occurs under the hood. The Raster class gives you a friendly user interface that works regardless of what kind of drivers are opperating under the hood. The primary benefit to you is that writing the code becomes a lot simpler because you don't need to worry about what kind of raster you are working with.

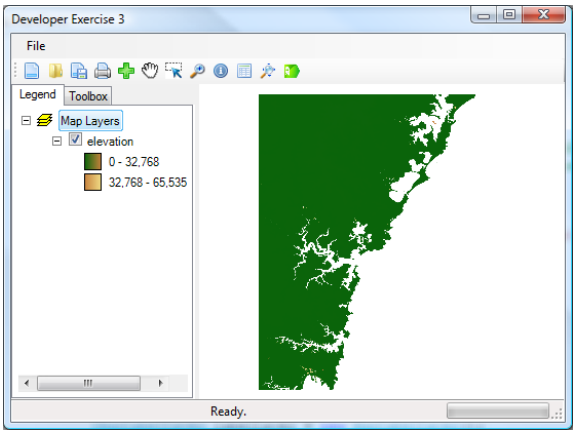


Figure 79: Default Raster

One fairly obvious drawback with the symbology above is that it is almost certainly being thrown off by an unexpected no-data value. We can use the symbolizer interface for rasters in either MapWindow 6.0 or your new project by double clicking next to the elevation layer in the legend. This will launch a dialog like the one below.

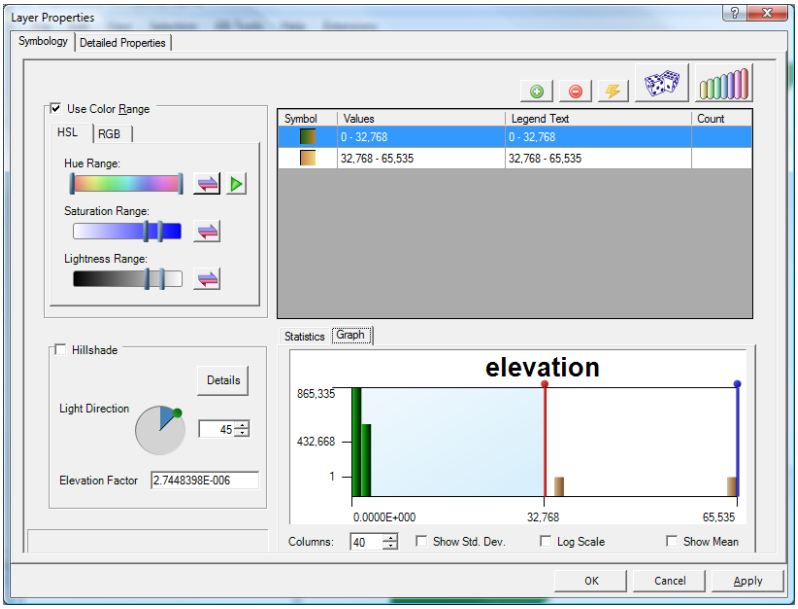


Figure 80: Statistics

We can see right away that of the many sample points that were taken from this image, only one or two values in the extreme upper range of 32,000 and 65,000 respectively can throw off the entire symbolic range for this raster. We can repair this easily by sliding the symbolic sliders down to the left end of the plot. What is less obvious until you start zooming in is that almost all of the values are effectively no-data values of 0 and are showing up on our plot. You can use the mouse wheel 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 trick that we can do in specifically this situation. What we will do is 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 won't 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 allows us to zoom into the range specified by the categories above.

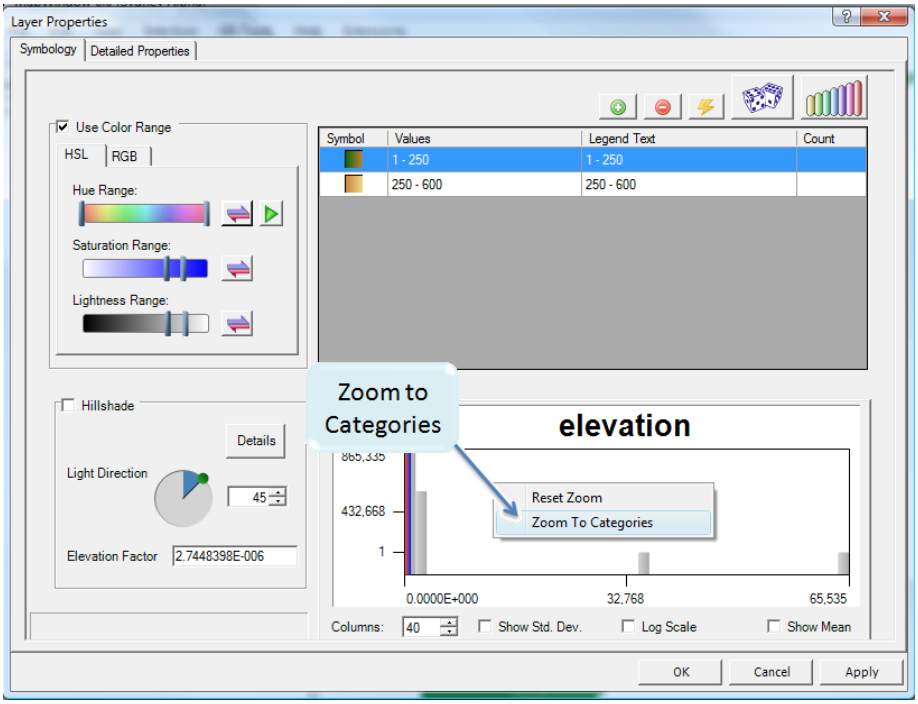


Figure 81: Zoom To Categories

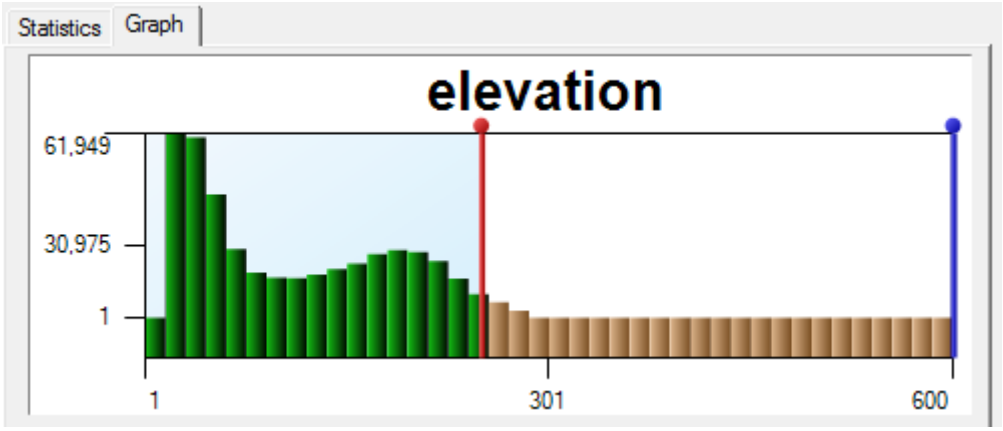


Figure 82: After Zoom

We can see from the statistics here, that the large percentage of the points occur between about 1 and 150, with a sizeable hump up to about 300. We will put the break slider in the minimum that occurs at around a value of 150, or you can use 150 as the exact range in the values column again. The result is that we can get a much more cleanly symbolized raster when

we are done. The no-data region is still white, but now we can see a much better representation of the values that we do have since we have eliminated the outliners.

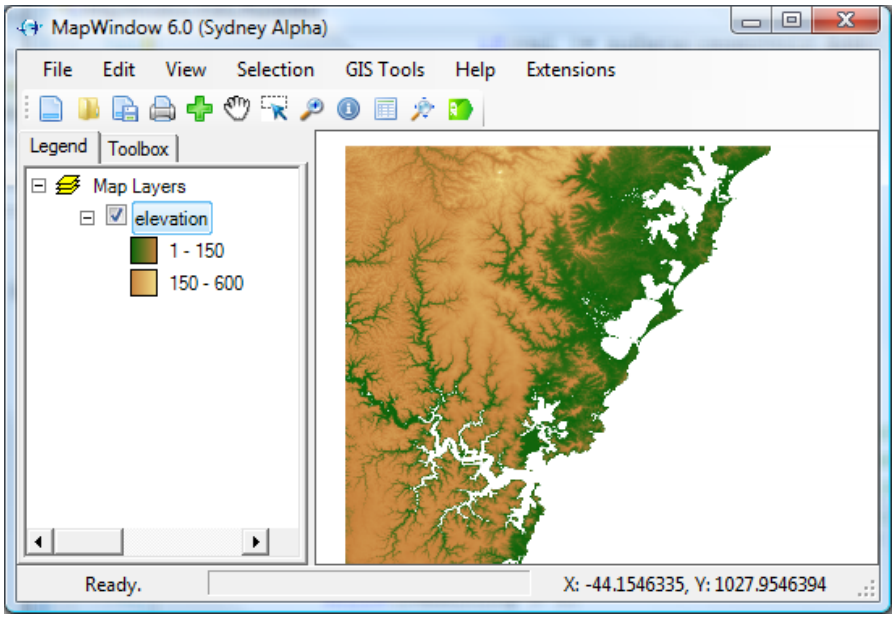


Figure 83: After Adjustments

So in order to set this up programmatically, we would like to first of all edit the range of the categories that are automatically generated when adding the data layer so that we have a range from 1-150 and from 150 to 800.

1.8.3. Control Category Range

Objective: Control Category Range

The only thing to notice when controlling the range is that you can control the range values independently from modifying the legend text. In order to update the legend text based on other settings, we can use the ApplyMinMax settings. Alternately, we could have set the legend text directly, just as we can for the other categories.

```
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();
}
```

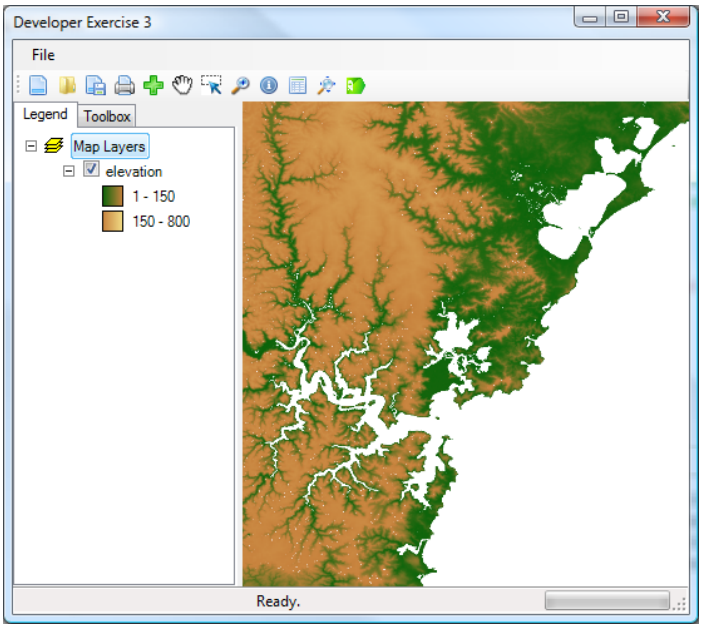


Figure 84: Programmatic Restrict Range

1.8.4. Shaded Relief

Objective: Add Lighting

```
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();
```

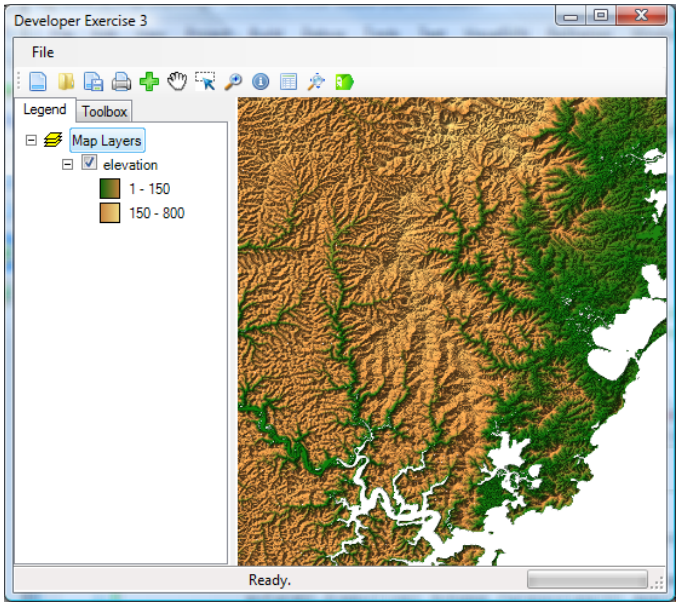


Figure 85: With Lighting

1.8.5. Predefined Schemes

Objective: Use Glacier Coloring

There are several pre-defined color schemes that can be used. All of the pre-set schemes basically use two separate ramps that subdivide the range and apply what is essentially a coloring theme to the two ranges. Those ranges are easilly adjustable using the range characteristics on the category, but should be adjusted after the scheme has been chosen, or else applying the new scheme will overwrite the previous range choices.

```
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();
```

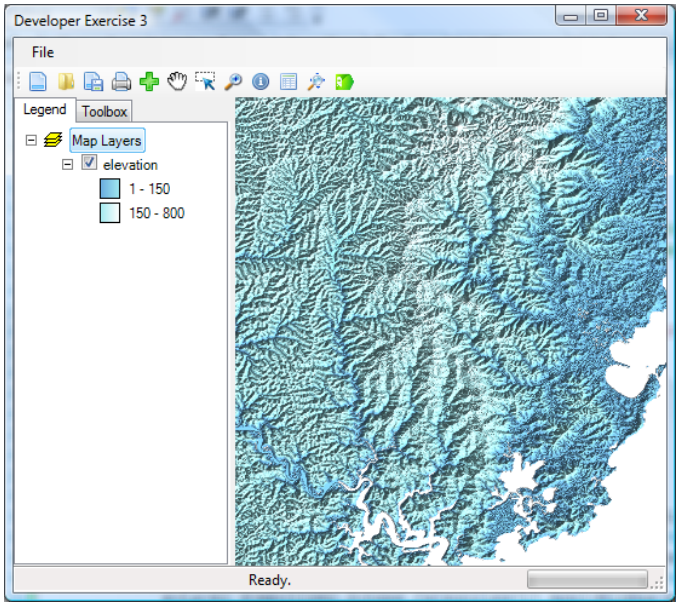


Figure 86: Glaciers

1.8.6. Edit Raster Values

Objective:	Fix Raster Values

Previously, we have been using tricks to color the elevation but where we directly ignored the values that were either no-data values that read 0, or else apparently bad data values that showed impossible values like 65,000. In this section we will use the raster data class itself in order to repair the values programmatically. This will only alter the copy that we have in memory and will not overwrite the values to the disk unless we specifically instruct it to do so. In the example below, we can cycle through all of the values in the raster using the NumRows and NumColumns properties to give us an idea of what the bounds are on the loops. The Value property takes a double index, and will work with whatever the real data type is and convert that data type into doubles. We can use this to quickly clean up the values on the raster before we ever create a layer. We can also assign the no data value on the raster so that it matches the 0 values that cover a large portion of the raster. This will automatically eliminate it from the statistical calculations so that our default symbology should look better.

```
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);
```

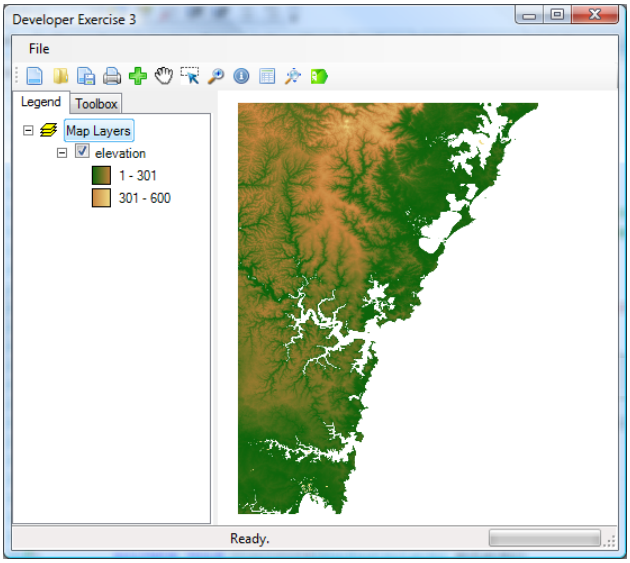


Figure 87: Default Symbology of Fixed Raster

This looks a lot better, and if we double click on the elevation layer, we can take a look at what the statistical plot automatically shows. In the next figure, we can see that the default range shows values from 1 to 600, and does not include in the statistical summary the values that we have 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. This can easily be fixed by cycling through the raster in the same way and adjusting values so that they work with the given statistics.

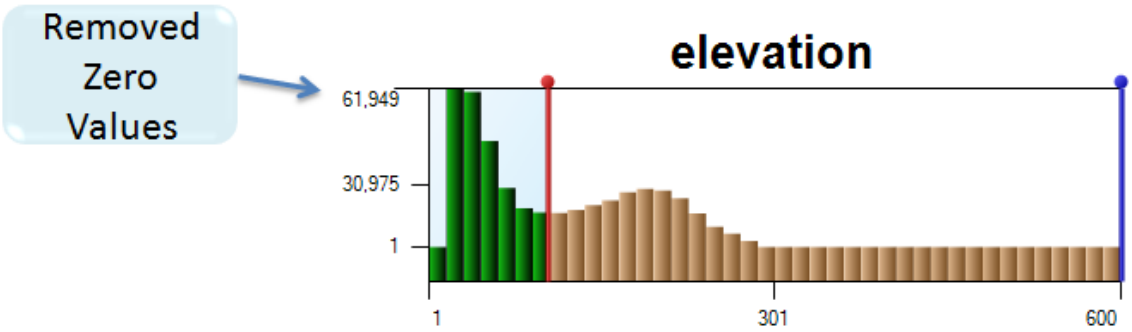
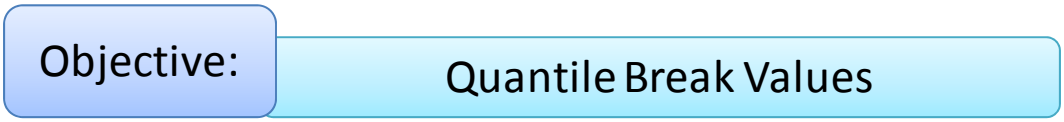


Figure 88: After Fixing Raster

1.8.7. 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 our previous edits to fix the raster values become more important. If we had tried to apply quantile breaks before, instead of coloring the raster appropriately, we would have had all but one of the ranges read 0-0. Now, we get a reasonable range.

```
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();
```

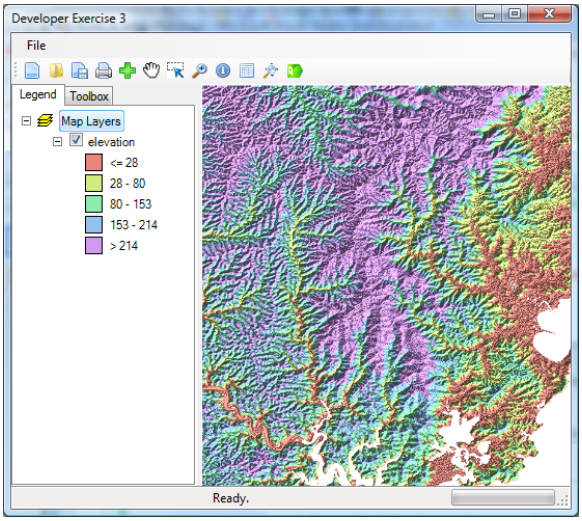


Figure 89: Quantile Breaks

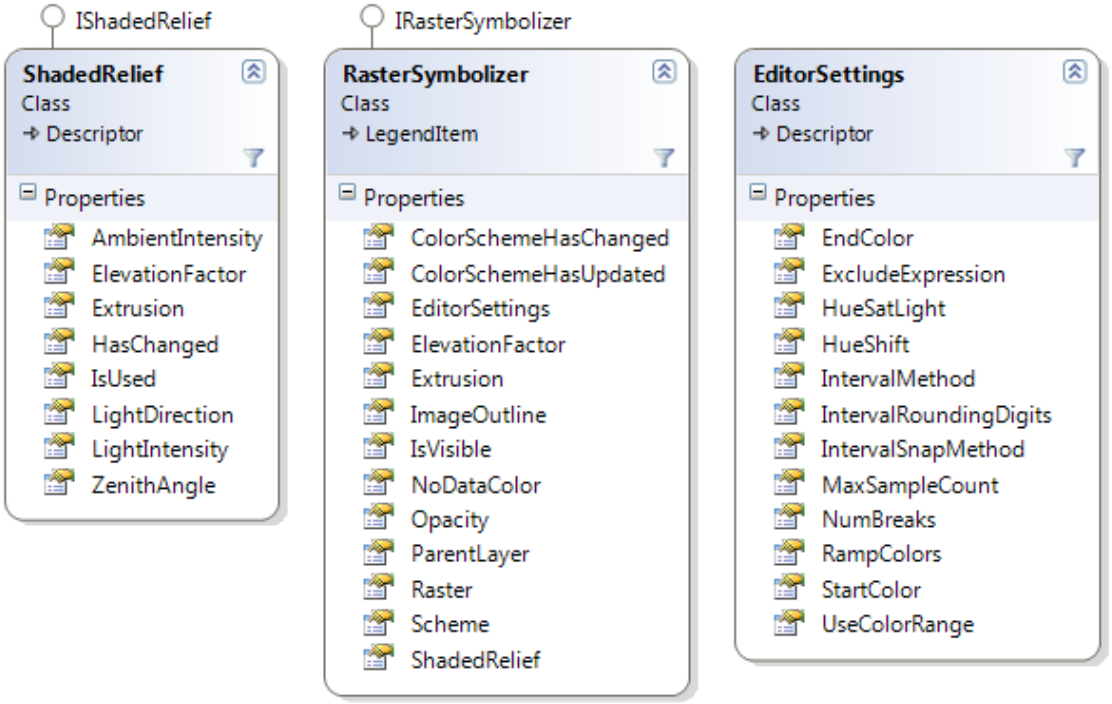
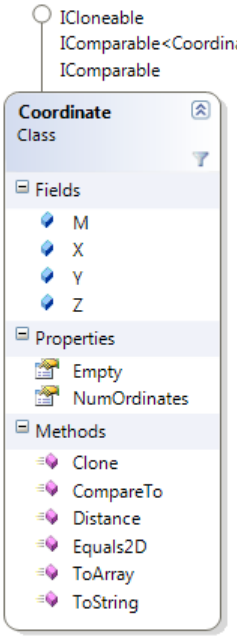


Figure 90: Raster Symbology Classes

1.9. MapWindow 4 Conversion Tips

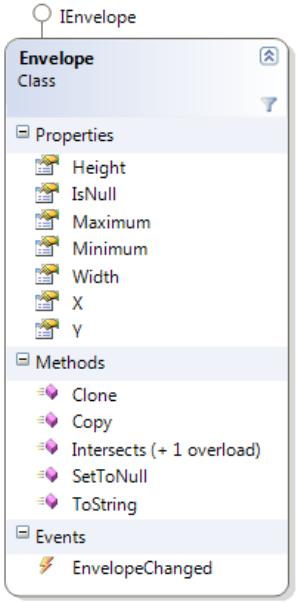
In this section, we assume that you might be a developer with some pre-existing experience with MapWindow 4. Because the underlying object libraries are very different, it might be confusing for a MapWindow 4 developer to get started working with the objects in MapWindow 6. The most challenging ideas involve the introduction of inheritance and extensible interfaces.

1.9.1. Point



Class: MapWinGIS.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 that just storing a coordinate location. Therefore, the coordinate class has taken over as the class to use.

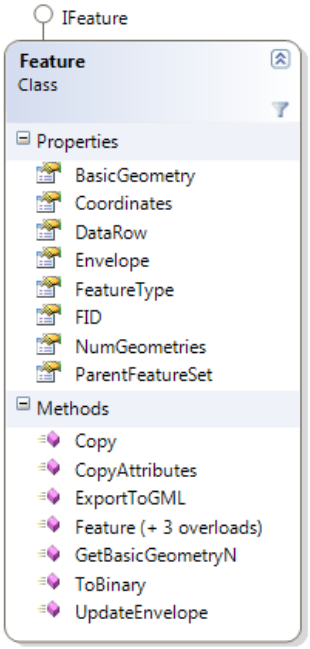


1.9.2. Extents

Class: MapWinGIS.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. This also provides accessors for controlling the X, Y, Height and Width properties, which are calculated from the Minimum and Maximum coordinates.

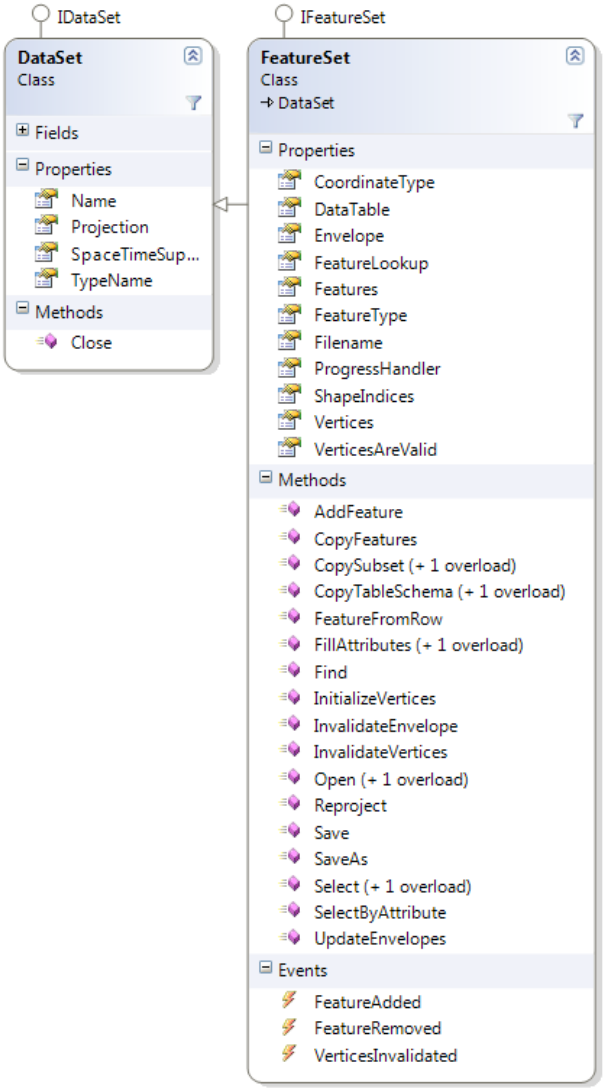
1.9.3. Shape



Class: MapWinGIS.Shape

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 you can directly access the coordinates. 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.

1.9.4. Shapefile

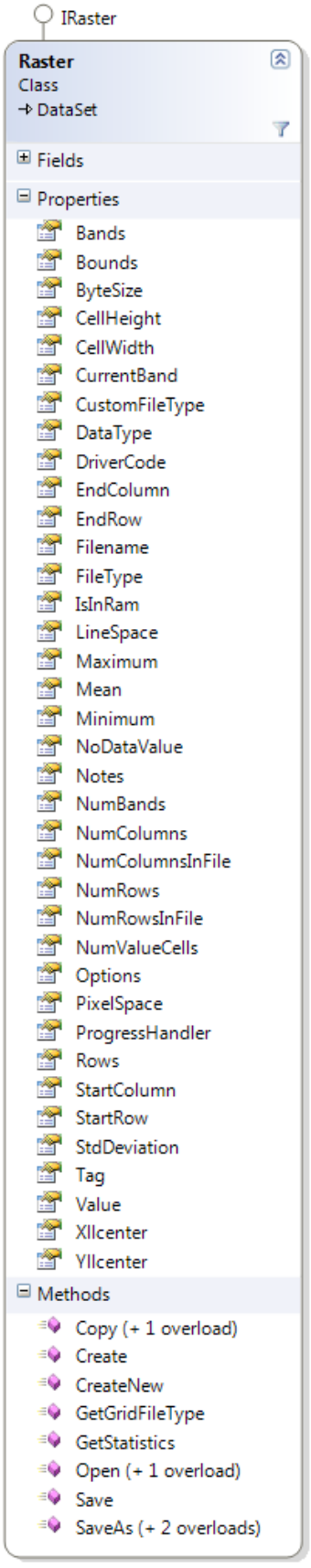


Class: MapWinGIS.Shapefile

In MapWindow 4, there was an assumption that there was only going to ever 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 contain accessors 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 you are working with an image, raster, or featureSet, 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 disconnect between the featureset and the layer being drawn. 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 else that have attributes that match an SQL query is supported and used at the level of the FeatureSet.

The FeatureSet is interesting because it acts as both a base class, meaning that a PolygonShapefile ultimately inherits from FeatureSet, but it also acts 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, or possibly a database or web service. Therefore, we provide the FeatureSet 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.



1.9.5. Grid

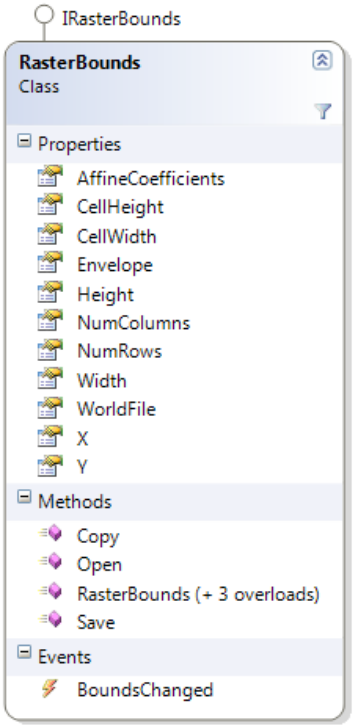
Class: MapWinGIS.Grid

The grid class was 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 get 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 to the left may seem intimidating, but realistically, there are only a few properties that are critical. The first is the "Value" property. This is where the actual values are stored. To access the value on row 7 and column 6, you would simply access double val = myRaster.Value[7, 6]. Regardless of what the source data is stored as, the value accessed here will be converted into double values. This makes it easier to write code, because you don't have to handle each of the separate possibilities independently.

The other critical values are NumRows and NumColumns. These are necessary so that you know how to cycle through the values form the Value property. The final bit of information that is especially important is the Bounds property. Many of the properties to the left are actually just shortcut accessors to the RasterBounds property. This is where the information is kept that stores the geospatial part of the raster content. With the Bounds property, you can control the position of a raster in space. The NoDataValue is also useful, as it can help change how statistics are calculated.

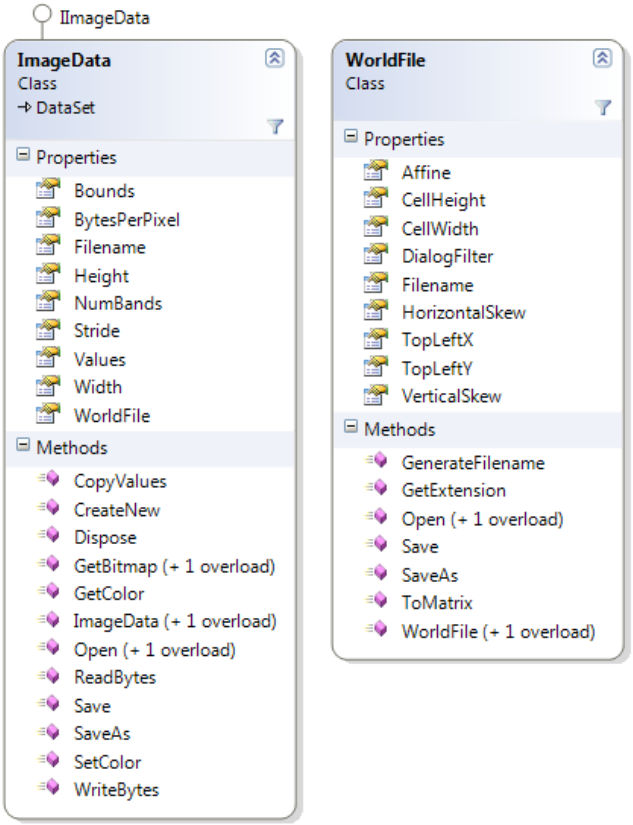
1.9.6. GridHeader



Class: MapWinGIS.GridHeader

The Header 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. You also need the number of rows and columns in order to get a valid rectangle that can enclose the entire raster. The Envelope is a rectangular form that completely contains the raster.

1.9.7. Image

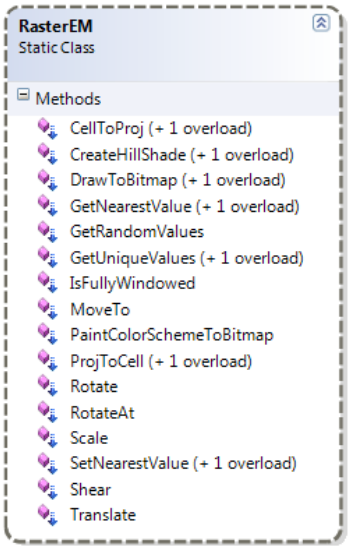


Class: MapWinGIS.Image

The Image class was specifically designed for working with images, and was not really designed for data analysis. The new ImageData class supports a “Values” method similar to the values of a raster, but is actually an array of byte values. The format of the image will change how those byte values are organized. The Stride property is the number of bytes in a given row. This 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 gives an idea of whether you are working with a GrayValue, RGB or ARGB image. With images, the geospatial location and sizing is controlled via a WorldFile.

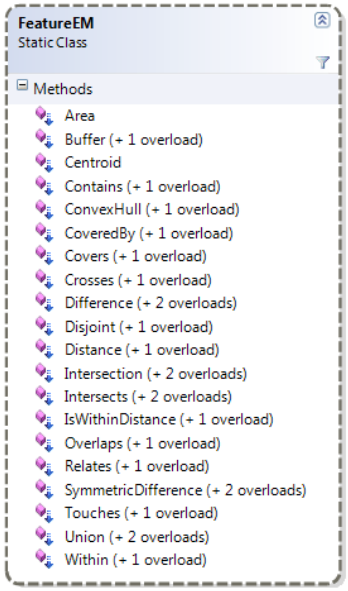
1.10. Extension Methods

Extensions | Rasters



Not all of the methods are supported directly on a raster. Many useful methods are actually supported in the form of "Extension Methods". For the raster, this includes useful methods that can alter the raster bounds, like Translate, or Scale, or Rotate. But other useful abilities like CreateHillShade actually 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 indicies.

Extensions | Feature



The Feature, and in fact any class that implements the IFeature interface will be extended with the geometry methods that are so critical to vector calculations. These not only include the overlay operations like Intersection, Union and SymmetricDifference, but all the tests that you might want to use like touches or within. Some of the bonus methods are methods like Area, which calculates the areas of polygons. Another is Centroid, which calculates the center of mass for geometries. ConvexHull can be used to simplify a geometry in the same way that you would simplify something by wrapping it with an elastic band. It draws straight lines past concave sections, and follows 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.