The Google API, Part II: More Google Maps and GMLibrary

By Iskander Shafikov

Versions: C++Builder XE3-XE, 2010-2006

In this article, we continue the discussion of GMLibrary, the open Delphi/C++Builder library that implements the Google Maps API.

In the previous part, we covered the basic structure and components of GMLibrary, as well as the fundamentals of navigation, changing the map's appearance, and adding objects on the map.

Creating a marker and configuring its published properties was discussed at large, so we'll first go over the actions responsible for adding the remaining map objects (a rectangle, a polygon, a circle, and a polyline), and then look at some other interesting capabilities of the library.

Back to the demo app

For this article, we'll use the same demo project as was introduced in Part I [1], with the necessary enhancements and additions.

First of all, it's worth downloading and installing the newest version of GMLibrary from its website (http://sourceforge.net/projects/gmlibrary), if you haven't done so yet. The latest release, as of now, is 0.1.9, of January 14, 2013. You may look up the version history/bug fixes, to make sure you've got the most recent one.

I must next apologize to my readers and the library's author for a mistake I made in the previous article, where I erroneously said that `TGMRectangle` was a collection of `TPolygon` objects; the type of the collection items is certainly `TRectangle`. This is a mere erratum, but it's far better to point it out here than let it slip by.

Finally, I've gone through the source code to provide each method and almost every significant line with in-depth comments; thus, even without this article, reading the code shouldn't be a problem.

Changes to the demo

For the purpose of this article, the demo has been enhanced with new components and actions. **Figure 1** shows the additions made to the main form.

The new actions added to the demo and their purposes are as follows:

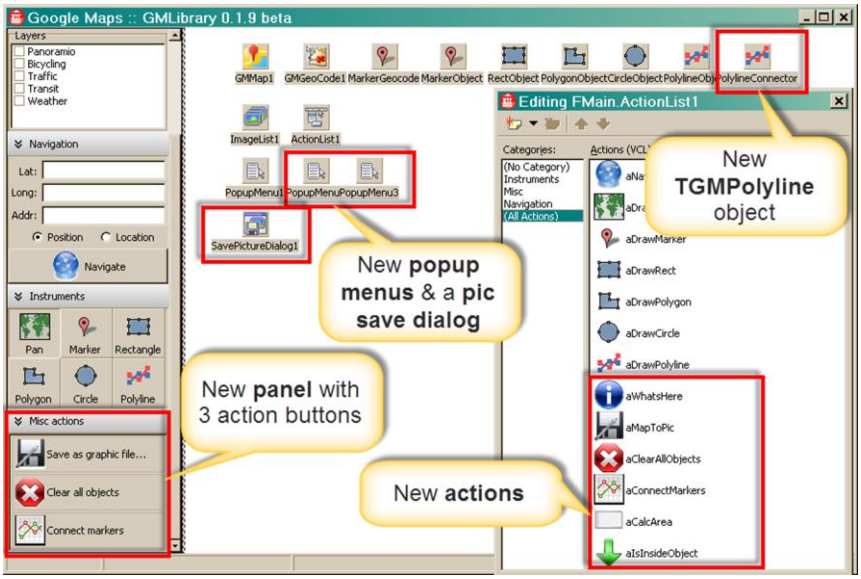

Figure 1: *Additions to the demo: designtime view.*

- `aWhatsHere`: uses the geocoding component (`GMGeoCode1`) to retrieve the nearest address from a clicked position, and displays the formatted address in the status bar;
- `aMapToPic`: launches `SavePictureDialog1` and calls the map's `SaveToJPGFile` method to save the current map view (objects and all) to a JPEG file;
- `aClearAllObjects`: asks the user to clear all objects from the map and, if confirmed, does so, by calling the `Clear` method on each of the GM collection objects;
- `aConnectMarkers`: connects all the marker items in the marker collection (`MarkerObject`) with a polyline (added to `PolylineConnector`, one of the new components on the form), and displays the total distance of the connecting path in the status bar (in kilometers);
- `aCalcArea`: calculates the area (in square km) of a rectangle, polygon, or circle on the map;
- `aIsInsideObject`: checks if a marker is geometrically inside a shape object (rectangle, polygon, or circle).

Three of these actions—`aMapToPic`, `aClearAllObjects`, and `aConnectMarkers`—have the corresponding buttons in the new "Misc actions" category panel on the form. The rest are available through the three popup menus.

`PopupMenu1` showing up when the map receives a right mouse button click has now a second item (after "Navigate"), linked to the `aWhatsHere` action. In this way, the user can right-click the map and see the nearest address, as returned by the geocoding API.

The new `PopupMenu2` has a single item linked to the `aCalcArea` action, and is invoked when the user right-clicks a shape on the map (rectangle, polygon, or circle).

The third popup menu (`PopupMenu3`) has also but one action-aware item, linked to `aIsInsideObject`. This menu is displayed in the `OnRightClick` event handler of `MarkerObject`, so that the user can right-click any marker on the map and execute this action to see what shapes the marker lies inside, if any.

Drawing objects on the map

As mentioned in the previous part of this series, only two types of map objects are provided in the demo with property edit dialogs—markers and rectangles. The other three object types (circles, polygons, and polylines) can be added onto the form with fixed (non-editable) properties.

Drawing a rectangle

Adding a rectangle is quite similar to adding a marker. First, when the "Rectangle" instrument is selected, the `aDrawRect` action displays the rectangle edit dialog that looks very much like the marker edit dialog, having the `TRectangle` properties listed in a `TValueListEditor`, enhanced by color combo boxes, in-place drop-down menus, etc. The value of `DrawObject` is set to `DO_RECT`, so that a next click on the map will add a new rectangle by calling:

```
std::unique_ptr<TLatLng> pNE(
  new TLatLng(LatLng->Lat + 1.0f,
    LatLng->Lng + 1.0f));
SetRect(LatLng, pNE.get());
```

Like `SetMarker()`, the `SetRect()` method has a double purpose—it can either add a new rectangle object (if its last parameter is a `NULL` pointer), or edit an existing one (if the last parameter is a valid pointer to a `TRectangle` object). Its sole difference from `SetMarker()` is that `SetRect()` takes not one but two coordinate pairs (`TLanLng`), corresponding to the rectangle's SW (South-West) and NE (North-East) points. In the rest, this method is equivalent to `SetMarker()`, so it will be as well to leave out a detailed discussion of its implementation.

The editing technique is also the same as that of a marker: the `Tag` property of an added `TRectangle` determines the way the `OnClick` event will be handled by the collection (`RectObject`). If `Tag = 1`, the corresponding rectangle will be hidden (to be later deleted by the 'garbage collector' –discussed further). If `Tag = 2`, the edit dialog is brought up, and the rectangle's properties can be edited and re-applied. See the source code for comments.

Drawing a circle

Similarly, adding a circle is done via `SetCircle()` method, which is defined as follows:

```
void __fastcall TFMain::SetCircle(
  TLatLng* LatLng,
  Gmcirclevcl::TCircle* ACircle)
{
  // add new circle to CircleObject
  Gmcirclevcl::TCircle* Circle =
    CircleObject->Add(LatLng->Lat,
      LatLng->Lng, 100000);
```

```
  // set properties
  Circle->StrokeColor = clMaroon;
  Circle->Editable = true;
  Circle->Clickable = true;
  Circle->StrokeWeight = 2;
  Circle->StrokeOpacity = 0.8f;
  Circle->FillColor = clPurple;
  Circle->FillOpacity = 0.5f;
}
```

This 'lazy' routine just adds a new circle with the center in the specified location, a fixed radius of a hundred kilometers, and other fixed properties (a maroon-colored 2-pixel-thick 80%-opaque border and a purple-colored 50%-opaque background). The `Editable` property being true, the circle's size and position can be changed on the map directly.

Drawing a polygon and a polyline

Drawing a polygon and a polyline uses a somewhat different technique. Both these object types consist of multiple (at least two) points, called linepoints, connected by a path. This path is open in the case of a polyline, and closed in the case of a polygon. In many other ways, these objects don't differ much—in fact, their respective classes, `TPolygon` and `TPolyline`, are siblings, having a common parent—`TBasePolylineVCL` which is in turn derived from the abstract class `TBasePolyline`. This latter class declares all the necessary methods to manipulate linepoints (add, insert, delete, move, clear), compute the area of the enclosed shape, encode/decode linepoint paths, and so on.

As it is, we'll have to provide for adding new linepoints to a polygon / polyline by clicking on the map, as well as closing the path to prevent further additions. To do that, a couple of Boolean flags are declared—`FNewPolyline` and `FNewPolygon`, which are set to `true` if a new object is to be added, or `false` to add a new linepoint to an existing object.

Thus, the `aDrawPolygon` action (fired when the "Polygon" instrument is selected) does the following:

```
if(aDrawPolygon->Checked) {
  FNewPolygon = true;
  DrawObject = DO_POLYGON;
}
```

When the map is clicked, with the "Polygon" instrument selected, the value of `FNewPolygon` is used to add either a new polygon object or a new linepoint to the last polygon in the collection:

```
void __fastcall TFMain::GMMap1Click(TObject*
  Sender, TLatLng *LatLng, Real X, Real Y)
{
  // <-- code omitted: see source -->
  switch (DrawObject) {
    // <-- other cases... -->
    case TDrawObject::DO_POLYGON:
      // if FNewPolygon, add a new polygon
      if(FNewPolygon) SetPolygon(LatLng);
      else {
        const int nPolygons =
          PolygonObject->Count;
        if(nPolygons > 0) {
          // get ptr to last added polygon
          Gmpolygonvcl::TPolygon* Polygon =
            PolygonObject->Items[nPolygons-1];
          // add a new linepoint
          Polygon->AddLinePoint(LatLng->Lat,
            LatLng->Lng);
        }
      }
      break;
    // <-- other cases... -->
  }
}
```

If `FNewPolygon` is `false`, we get a pointer to the last added polygon in the `PolygonObject` collection (if it isn't empty), and call its method `AddLinePoint` to add a new linepoint at the clicked location. Otherwise, a new polygon is added by a call to `SetPolygon`, defined as follows:

```
void __fastcall TFMain::SetPolygon(
  TLatLng* LatLng,
  Gmpolygonvcl::TPolygon* APolygon)
{
  // add a new polygon to PolygonObject
  Gmpolygonvcl::TPolygon* Polygon =
  PolygonObject->Add();

  // set polygon's properties
  // <-- see source -->

  // add the first linepoint
  Polygon->AddLinePoint(LatLng->Lat,
                        LatLng->Lng);
  // next click the map adds new linepoint
  FNewPolygon = false;
}
```

Here, after adding a new polygon and setting its properties, we also add the first linepoint which will be visible on the map, and set `FNewPolygon` to `false`, so that a next click on the map will add a next linepoint to this polygon.

The polygon's path can be finalized on a double click, like so:

```
void __fastcall TFMain::PolygonObjectDblClick(
  TObject *Sender, TLatLng *LatLng, int Index,
  TLinkedComponent *LinkedComponent)
{
```

```
    // get ptr to the double-clicked polygon
    Gmpolygonvcl::TPolygon* Polygon =
      dynamic_cast<Gmpolygonvcl::TPolygon*>
      (LinkedComponent);
    if(Polygon && DrawObject ==
       TDrawObject::DO_POLYGON) {
        FNewPolygon = true;
        aDrawNone->Execute();
    }
}
```

The double-click event handler retrieves the polygon that has received the double-click message and, provided that the "Polygon" instrument is selected, resets `FNewPolygon` to `true` to make a next click on the map add a new polygon rather than a linepoint.

Exactly the same technique is applied to add polylines. You may look up the corresponding methods in the source code.

Two of the new features

Now that we've looked at the map objects, let's discuss two of the new features of the demo. (The remaining features will be discussed next month.)

Getting a location's nearest address

The "what's-here" functionality is available by right-clicking the map and choosing the "What's here?" item in the popup menu. The corresponding action (`aWhatsHere`) is implemented as follows:

```
void __fastcall TFMain::aWhatsHereExecute(
  TObject *Sender)
{
  StatusBar1->Panels->Items[2]->Text =
    "Locating...";
  GMGeoCode1->Marker->Clear();
  GMGeoCode1->Geocode(ClickedPoint);

  if(GMGeoCode1->GeoStatus ==
    TGeocoderStatus::gsOK) {
    StatusBar1->Panels->Items[2]->Text =
      GMGeoCode1->GeoResult[0]->FormatedAddr;
    GMGeoCode1->Marker->Items[0]->Title =
      GMGeoCode1->GeoResult[0]->FormatedAddr;
    GMGeoCode1->Marker->Items[0]->
      CenterMapToMarker();
    GMMap1->RequiredProp->Zoom = 15;
  }
  else
    StatusBar1->Panels->Items[2]->Text = "";
}
```

This action uses the same geocoding component that's used to do a reverse operation in `MapLocate()` (discussed in the previous article). In the highlighted line, the overloaded method `GeoCode()` of `TGMGeoCode` is used to pass a `TLatLng` location (rather than an address string). The location (`ClickedPoint`) is set in the map's `OnRightClick` event handler to the clicked location. The rest of the implementation doesn't differ much from `MapLocate()`—the result returned by `GeoCode()` is checked for success, and if that check is passed, the resulting formatted address is displayed in the status bar and assigned to the title of the first location found; then the map is zoomed and navigated to that location.

Of course, this feature is but of little practical value, since Google Maps doesn't show the nearest offices, contact information, etc., but picks up a nearest address geometrically, which may turn out to be the name of a highway, a suburb or city district, even though we may want to locate buildings with their street addresses, lock stock and barrel. This is not feasible in the standard Google Maps API.

Saving the map as a JPEG image

The `aMapToPic` action is as simple as:

```
if(SavePictureDialog1->Execute())
  GMMap1->SaveToJPGFile(
    SavePictureDialog1->FileName);
```

`SaveToJPGFile()` saves the current map view, with all its objects, to a JPEG file. Clearly, the presence of this internal method in the `TGMMap` class is a great advantage. In fact, this method is implemented so that a save dialog will be displayed automatically if the filename argument is passed as an empty string. So, in this case, launching `SavePictureDialog1` might be redundant.

Conclusions

This month we have significantly enlarged our acquaintance with GMLibrary by a number of how-to demonstrations. The next article in this series will discuss the remaining features of the demo. Stay tuned.

Contact Iskander at **shaficovisk@yandex.ru**.

References

1. I. Shafikov, "The Google API, Part I: Google Maps and GMLibrary," *C++Builder Dev. Journal*, **17** (1), January 2013.