GDI Drawer – C# Version – Manual

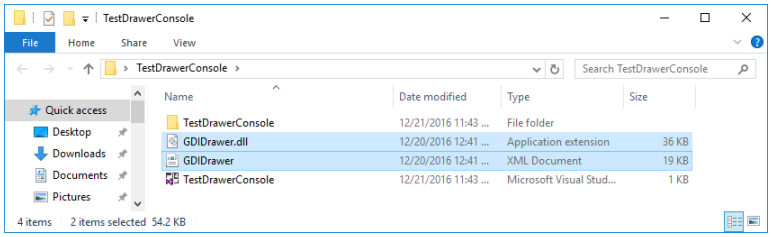
Contents

Adding GDIDrawer Support

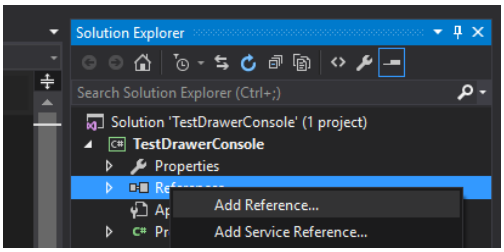
The GDI Drawer is a simple graphics tool that allows users to render primitive shapes. The drawer can be used from almost any kind of C# application, including simple console applications. The purpose of the GDI Drawer is to make programming in C# more fun by allowing users to create graphical output, without requiring extensive knowledge of a complex graphical interface.

The GDI Drawer is used by referencing its assembly in a C# application. The steps required to do this in a simple console application are shown below:

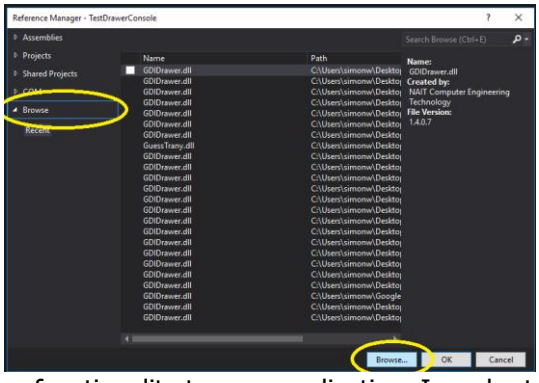
Create a console application, or use an existing one. Obtain a copy of the GDI Drawer assembly (GDIDrawer.dll) and optional XML documentation, and place these files in the same folder as your project.



In your project, from the solution explorer, right-click on 'References', and select 'Add Reference…'.



From the dialog, select the 'Browse' tab, and click on the 'Browse…' button. Sadly, this dialog remembers the previous locations of referenced files, so make sure you use the button at the bottom of the dialog! Locate the 'GDIDrawer.dll' file you copied into your project directory!



This will add the GDIDrawer functionality to your application. In order to make the GDIDrawer types readily available, a **`using`** directive for the GDIDrawer namespace should be included in your program code:

```
using System;
using System.Collections.Generic;
using System.Text;
using GDIDrawer;
```

The GDIDrawer is now ready for use in the application. NOTE: you may need to include the **System.Drawing** framework assembly, and a using directive for **System.Drawing**.

Using the GDIDrawer

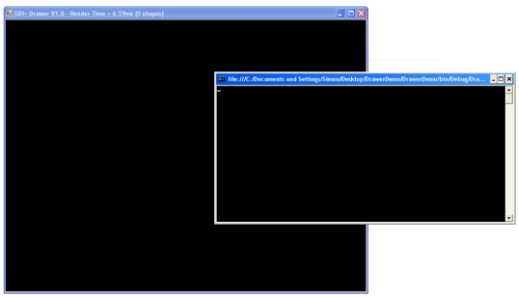
In order to use the **GDIDrawer**, an instance of the **CDrawer** class must be made. This is done by creating a reference to a new object of the **CDrawer** type. In a console application you will typically create the **CDrawer** object as a static member of the **Program** class:

```
namespace MyProgram
{
   class Program
   {
      static CDrawer s_Draw = new CDrawer();

      static void Main(string[] args)
      {
         Console.WriteLine("All done! Press a key to exit!");
         Console.ReadKey();
      }
   }
}
```

It is through this instance that all operations are performed.

NOTE: Simply creating an instance of the drawer will create the drawing output window:



By default the drawer instance will use a scale of 1, and contains 800 * 600 pixels. To draw a rectangle that starts 10 pixels over, and 100 pixels down, and has a height and width of 100 pixels, the following command would be issued:

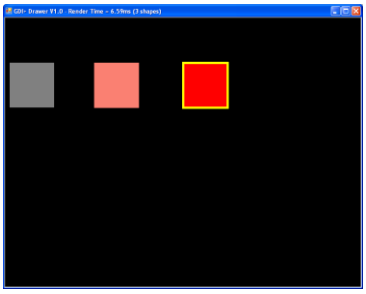
```
s_Draw.AddRectangle(10, 100, 100, 100);
```

The coordinates that you specify to drawer calls typically define a starting position and a height and width. These coordinates are used to define a bounding rectangle for the shape. In the case of a rectangle, the bounding rectangle *is* the rectangle. In the case of an ellipse, the bounding rectangle determines the shape and size of the ellipse limited within it.

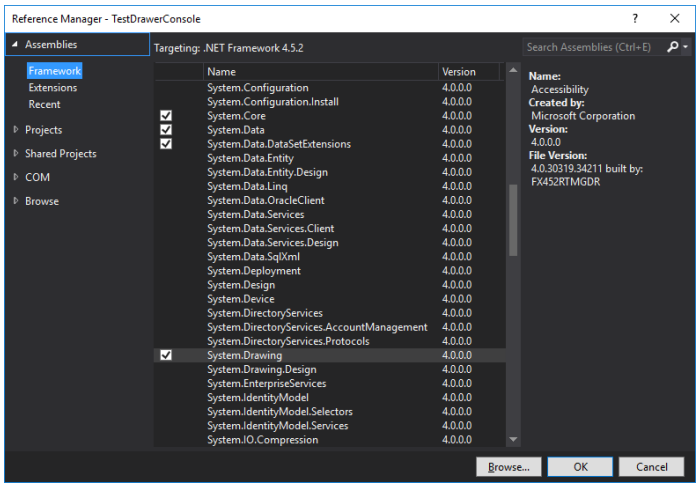
A list of overrides is found at the end of this document. Experimentation and dot exploration with the drawer is a fantastic way to find out what is available, and how each function works.

Most of the Add methods use a number of optional parameters which allow you to specify incrementally more detail in your drawing:

```
s_Draw.AddRectangle(10, 100, 100, 100); // default fill color of gray with no border
s_Draw.AddRectangle(200, 100, 100, 100, Color.Salmon); // with color, but no border
s_Draw.AddRectangle(400, 100, 100, 100, Color.Red, 5, Color.Yellow); // all elements set
```



Note: Using the more complicated arguments of the shape functions will require that you use types found in the `System.Drawing` namespace. If you want to use all of the features of the drawer, you will need to include this system assembly if it is not included (With Windows Forms applications it will be included by default, not so for console applications). This is done by right-clicking on "References" in the solution explorer, and selecting "Add Reference…". In the dialog that appears, use the Assemblies/Framework tab and locate the "`System.Drawing`" assembly. Ensure that the check-box to the left is checked, otherwise nothing will happen when you hit OK.



Once the assembly is included in the project, you should put in a using directive in your code to make the namespace available:

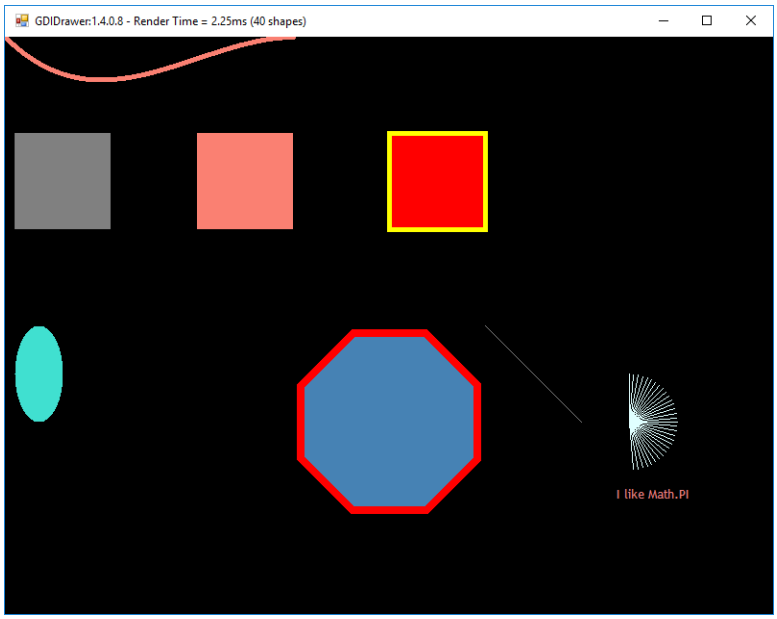
```
using System;
using System.Collections.Generic;
using System.Text;
using System.Drawing;
using GDIDrawer;
```

Following these steps is standard practice for making different parts of the .NET framework available to your applications. In this case, things that are relevant to drawing have been made available to the application (specifically, the `Color` type (as well as others)).

Drawing Graphics Primitives

The drawer supports rectangles, ellipses, polygons, lines, Beziers, and text:

```
s_Draw.AddRectangle(10, 100, 100, 100);
s_Draw.AddRectangle(200, 100, 100, 100, Color.Salmon);
s_Draw.AddRectangle(400, 100, 100, 100, Color.Red, 5, Color.Yellow);
s_Draw.AddEllipse(10, 300, 50, 100, Color.Turquoise);
s_Draw.AddPolygon(300, 300, 100, 8, Math.PI / 8, Color.SteelBlue, 8, Color.Red);
s_Draw.AddLine(500, 300, 600, 400);
s_Draw.AddBezier(0, 0, 100, 100, 200, 0, 300, 0, Color.Salmon, 5);
for (double rot = 0; rot <= Math.PI; rot += Math.PI / 32)
    s_Draw.AddLine(new Point(650, 400), 50, rot, Color.LightCyan, 1);
s_Draw.AddText(@"I like Math.PI", 10, 625, 450, 100, 50, Color.LightCoral);
```



Back-Buffer Operations

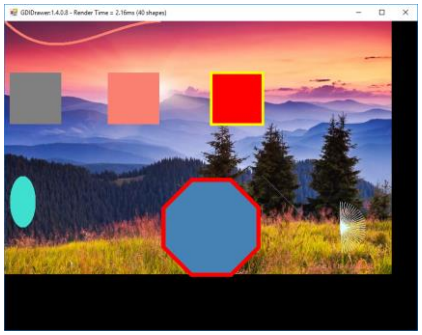
In addition to drawing simple graphics primitives, the GDIDrawer supports a modifiable back-buffer. The back-buffer is used to erase the window each frame, prior to drawing any added shapes. By default, the back-buffer is filled with black. If you want to change the color of the back-buffer, you can do so by setting the **`BBColour`** property to a color of your choice:

```
s_Draw.BBColour = Color.FromArgb (200, 200, 200);
```

Setting the back-buffer color will immediately set all of the pixels in the back-buffer to the color you specify, erasing any existing contents. Shapes are always drawn on top of the back-buffer, so changing the back-buffer color will not alter the shapes drawn.

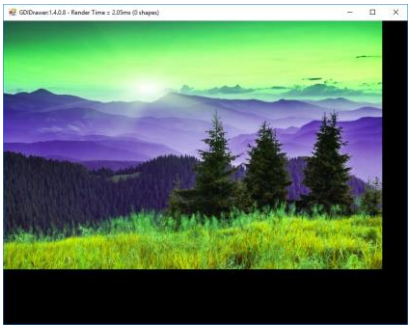
You may set the color of individual pixels in the back-buffer by calling the **`SetBBPixel`** method of the drawer:

```
Bitmap bm = Properties.Resources.sunset;
for (int y = 0; y < bm.Height; ++y)
    for (int x = 0; x < bm.Width; ++x)
        s_Draw.SetBBPixel(x, y, bm.GetPixel(x, y)); // retrieve pixel from bm setting s_Draw
```



You may also retrieve back-buffer pixels with GetBBPixel():

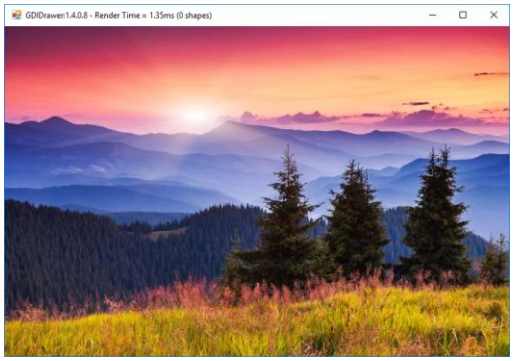
```
for (int y = 0; y < s_Draw.m_ciHeight; ++y)
    for (int x = 0; x < s_Draw.m_ciWidth; ++x)
    {
        Color c = s_Draw.GetBBPixel(x, y);
        s_Draw.SetBBPixel(x, y, Color.FromArgb(c.G, c.R, c.B));
    }
```



The drawer also contains a second constructor that will allow you to specify a picture to use for the background. The drawer will automatically size itself to the image:

```
static CDrawer s_Draw = new CDrawer(Properties.Resources.sunset);
```

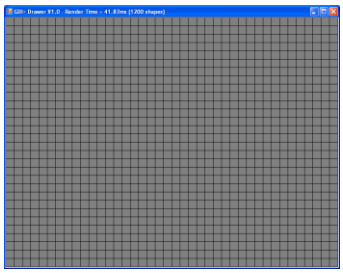
By doing this, you don't need to manually load the image – the drawer will do it for you (and a whole lot faster, too). If you want a persistent background for your drawer window, this is the way to go.



Using Scale

By default the drawer output window is 800 * 600 pixels, and uses a scale of 1. This means that the coordinates you specify correspond directly to individual pixels. You may find yourself in a situation (or a lab) that will require you to draw output with less granularity. For example, if you were required to draw a grid of 40 * 30 rectangles, you could do the math, and discover that each rectangle would need to be (800 / 40) pixels wide and (600 / 30) pixels in height. You could then draw all of the rectangles in pixel coordinates:

```
for (int y = 0; y < 600; y += 600 / 30)
    for (int x = 0; x < 800; x += 800 / 40)
        s_Draw.AddRectangle(x, y, 19, 19);
```



Note: While the math may be complicated, you always exercise per-pixel control over your output!

Notice that the coordinates are stepping by 20 pixels in both x and y directions. This is a situation where using a scale of 20 would be handy. The scale is multiplied by all coordinates you specify. In this case, a scale of 20 would mean that the coordinates of the rectangles are reduced to the index of their position:

```
dr.Scale = 20;

for (int y = 0; y < dr.ScaledHeight; ++y)
    for (int x = 0; x < dr.ScaledWidth; ++x)
        dr.AddRectangle(x, y, 1, 1, Color.White, 1, Color.Red);
```

Note: All coordinates are scaled (except for border thickness). This means that tiled shapes may require a border to be visually delimited. The **`ScaledWidth`** and **`ScaledHeight`** properties are always valid (even for the default Scale of 1) and should generally be used rather than literal boundary values (i.e. 800)

Labs will likely be written to take advantage of the simplicity scaling provides.

Mouse Operations

The GDIDrawer is capable of reporting mouse events related to the output window, including movement and click events. Your application polls for mouse events with calls to **`GetLastMouseLeftClick`**, **`GetLastMouseRightClick`**, and **`GetLastMousePosition`**. Each of these functions will 'out' a **`Drawing.Point`** indicating the last position that the event occurred at (in pixel coordinates). Each function additionally returns a **`bool`** that indicates if the event is new since the last time the function was called. If the return value is **`true`**, then the point represents a new, unseen coordinate. If the return value is **`false`**, the coordinate is stale (already read), or the event has never occurred.

There are scaled versions of these functions called **`GetLastMouseLeftClickScaled`**, **`GetLastMouseRightClickScaled`**, and **`GetLastMousePositionScaled`**. These methods are functionally equivalent to the unscaled versions, except the coordinates returned factor in the current scale.

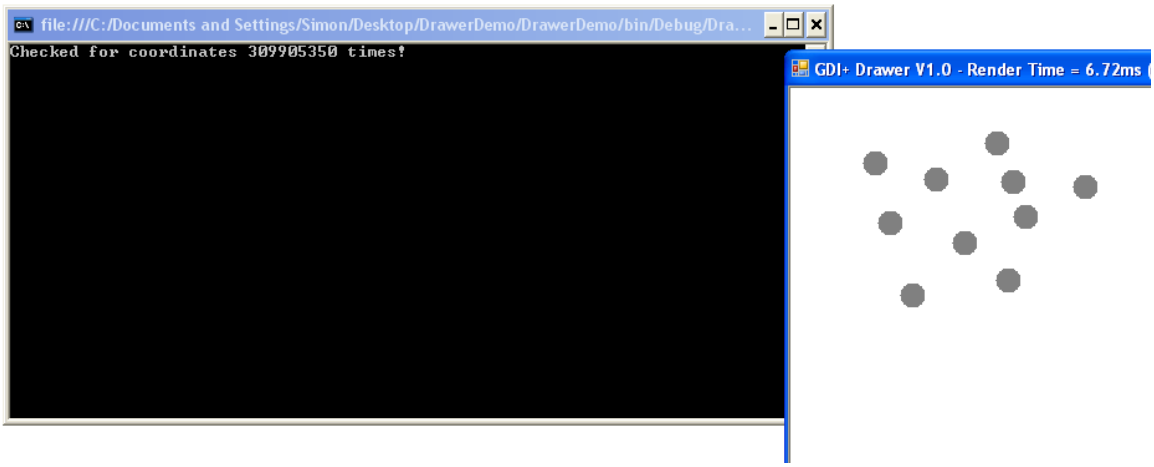
The following sample code illustrates how to poll the left-click event until 10 ellipses have been drawn at the click locations:

```
Point pCoord;           // coords to accept mouse click pos
int iNumClicks = 0;     // count number of clicks accepted
int iFalseAlarm = 0;    // count the number of poll calls
do
{
   bool bRes = s_Draw.GetLastMouseLeftClick(out pCoord);    // poll
   if (bRes)                                                // new coords?
   {
      ++iNumClicks;
      s_Draw.AddEllipse(pCoord.X - 10, pCoord.Y - 10, 20, 20);

   }
   else
      iFalseAlarm++;                                        // not new coords
}
while (iNumClicks < 10);

Console.WriteLine("Checked for coordinates " + iFalseAlarm.ToString() + " times!");
```

Of course, polling like this is extremely CPU intensive, as the coordinates are being checked millions of times per second.



Checking for right-click events and mouse move events is done using the same methodology.

The drawer also supports events for mouse events. You may subscribe to mouse events as you would any other event, just be advised that drawer events exist in the thread of the drawer. If you are writing a Forms application, you may overcome this issue by Invoking back into the main Form thread.

The drawer will suppress reporting the same mouse coordinate unless the **`RedundaMouse`** property is set to **`true`**. This can be set at any time through the **`RedundaMouse`** property, or through the constructor.

Updating and Rendering

By default the drawer runs in a mode which will automatically update the output window, the user is only required to clear/add new items to be displayed. This is normally adequate for most program usage, however, occasionally the periodic update of the output when many objects are being manipulated causes flicker when only partial updates are performed. This is due the drawer not being aware when the user has completed adding all objects, it may attempt to output the objects at any time and thus only output a portion of the user's additions.

The drawer object has an additional property **`ContinuousUpdate`**. The default is **`true`** and will automatically update the output based on a timer. This behavior can be selectively turned off by setting **`ContinuousUpdate`** to false. It is then the responsibility of the user to "tell" the drawer to output the current contents of all user added objects. This is done by calling the **`Render()`** method. You should only call **`Render()`** upon completion of all clear/add operations that represent a final output frame.(i.e. calling **`Render()`** after ever add operation will result in the flicker).

ContinuousUpdate can be set at any time during program execution, thereby disabling automatic updates when a time critical or high object count operation is performed, and re-enabling upon completion.

```
// Disable continuous update
dr.ContinuousUpdate = false;

// perform lengthy/high object count operation
for (int i = 0; i < 1000; ++i)
  dr.AddEllipse(rnd.Next(dr.ScaledWidth),  // dependant on scale
                rnd.Next(dr.ScaledHeight),
                rnd.Next(20), // up to 20x the current scale
                rnd.Next(20));

dr.Render(); // tell drawer to show now, all elements have been added
```

Render() may be called whether the drawer is in **ContinuousUpdate** mode or not. If **ContinuousUpdate** is **true**, the **Render()** request is ignored and the output will be automatically generated at the next timer interval.

** Many have spent much time attempting to determine why their application does not generate any output, only to find that **ContinuousUpdate** was set **false**, and the application never calls **Render()**.

Making a CDrawer (more options)

There are two constructors for the drawer. Both constructors use optional parameters. These include **Width = 800**, **Height = 600**, **ContinuousUpdate = true** and **RedundaMouse = false**. This allows for variations in creation:

```
// Defaults are used 800x600, etc
CDrawer can = new CDrawer();

// 500x500, ContinuousUpdate = false
CDrawer can = new CDrawer(500, 500, false);

// all arguments
CDrawer can = new CDrawer(800, 600, false, false);

// bitmap use to set size and set background
CDrawer can = new CDrawer(Properties.Resources.sunset);

// bitmap use to set size and set background, no update, redundant mouse
CDrawer can = new CDrawer(Properties.Resources.sunset, false, true);
```

Keyboard Events

The drawer is able to report key events. This is handy if the drawer window has focus and you want the controlling application to handle the key events. Like the mouse events, these events will be in the thread of the drawer, so care must be taken in Forms applications no to cause cross-thread violations.

GDIDrawer.RandColor Functions

The **RandColor** class is a utility class that supplies a few additional handy methods for use with the CDrawer. Both return a randomly initialized Color object.

```
static public Color GetColor() // Returns a random color from a smaller set of 48 possible colors
static public Color GetKnownColor() // Returns a random color chosen from the set of Known Colors( named )

// example of normal use :
Color myColor = GDIDrawer.RandColor.GetColor();
Color myKnown = RandColor.GetKnownColor(); // assuming a using GDIDrawer; namespace inclusion
myDrawer.SetBBPixel( x, y, RandColor.GetColor()); // require a random color argument
```