Programming with C#

# Multi-tasking practice

1. Start Microsoft Visual Studio 2013 if it is not already running.
2. Open the GraphDemo solution, which is located in the \Microsoft Press\Visual CSharp Step By Step\Chapter 23\GraphDemo folder in your companion content folder. This is a Windows Store app.
3. In Solution Explorer, in the GraphDemo project, double-click the file GraphWindow.xaml to display the form in the Design View window.

Apart from the *Grid* control defining the layout, the form contains the following important controls:

* An *Image* control called *graphImage*. This image control displays the graph rendered by the application.
* A *Button* control called *plotButton*. The user clicks this button to generate the data for the graph and display it in the *graphImage* control.

***Note*** *In the interest of keeping the operation of the application in this exercise simple, this application displays the button on the page. In a production Windows Store app, buttons such as this should be located on the app toolbar.*

* A *TextBlock* control called *duration*. The application displays the time taken to generate and render the data for the graph in this label.

1. In Solution Explorer, expand the GraphWindow.xaml file and then double-click GraphWindow. xaml.cs to display the code for the form in the Code and Text Editor window.

The form uses a *WriteableBitmap* object (defined in the *Windows.UI.Xaml.Media.Imaging* namespace) called *graphBitmap* to render the graph. The variables *pixelWidth* and *pixelHeight* specify the horizontal and vertical resolution, respectively, for the *WriteableBitmap* object:

public partial class GraphWindow : Window   
{   
 // Reduce pixelWidth and pixelHeight if there is insufficient space available   
 private int pixelWidth = 12000;   
 private int pixelHeight = 7500;   
 private WriteableBitmap graphBitmap = null;   
 ...   
}

***Note*** *This application has been developed and tested on a desktop computer with 4 GB of memory. If your computer has less memory than this available, you might need to reduce the values in the pixelWidth and pixelHeight variables; otherwise, the application might generate OutOfMemoryException exceptions. Similarly, if you have much more memory available, you might want to increase the values of these vari-ables to see the full effects of this exercise.*

1. Examine the last three lines of the *GraphWindow* constructor.

The first two lines instantiate a byte array that will hold the data for the graph. The size of this array depends on the resolution of the WriteableBitmap object, determined by the pixelWidth and pixelHeight fields. Additionally, this size has to be scaled by the amount of memory required to render each pixel; the WriteableBitmap class uses 4 bytes for each pixel, which specify the relative red, green, and blue intensity of each pixel and the alpha blending value of the pixel (the alpha blending value determines the transparency and brightness of the pixel).

The final statement creates the WiteableBitmap object with the specified resolution.

1. Examine the code for the *plotButton\_Click* method.

This method runs when the user clicks the *plotButton* button.

You will click this button several times later in the exercise, so that you can see that a new ver­sion of the graph has been drawn each time this method generates a random set of values for the red, green, and blue intensity of the points that are plotted (the graph will be a different color each time you click this button).

The *watch* variable is a *System.Diagnostics.Stopwatch* object. The *StopWatch* type is useful for timing operations. The static *StartNew* method of the *StopWatch* type creates a new instance of a *StopWatch* object and starts it running. You can query the running time of a *StopWatch* object by examining the *ElapsedMilliseconds* property.

The *generateGraphData* method populates the *data* array with the data for the graph to be displayed by the *WriteableBitmap* object. You will examine this method in the next step.

When the *generateGraphMethod* method has completed, the elapsed time (in milliseconds) appears in the duration *TextBox* control.

The final block of code takes the information held in the data array and copies it to the *WriteableBitmap* object for rendering. The simplest technique is to create an in-memory stream that can be used to populate the *PixelBuffer* property of the *WriteableBitmap* object. You can then use the *Write* method of this stream to copy the contents of the data array into this buffer. The *Invalidate* method of the *WriteableBitmap* class requests that the operating system redraws the bitmap by using the information held in the buffer. The *Source* property of an *Image* control specifies the data that the *Image* control should display. The final statement sets the *Source* property to the *WriteableBitmap* object.

1. Examine the code for the *generateGraphData* method.

This method performs a series of calculations to plot the points for a rather complex graph. (The actual calculation is unimportant—it just generates a graph that looks attractive.) As it calculates each point, it calls the *plotXY* method to set the appropriate bytes in the *data* array that correspond to these points. The points for the graph are reflected around the x-axis, so the *plotXY* method is called twice for each calculation: once for the positive value of the x-coordinate, and once for the negative value.

1. Examine the *plotXY* method.

This method sets the appropriate bytes in the data array that corresponds to x and y-coordinates passed in as parameters. Each point plotted corresponds to a pixel, and each pixel consists of 4 bytes, as described earlier. Any pixels left unset are displayed as black. The value 0xBF for the alpha blend byte indicates that the corresponding pixel should be displayed with a moderate intensity; if you decrease this value, the pixel will become fainter, while setting the value to 0xFF (the maximum value for a byte) will display the pixel at its brightest intensity.

1. On the Debug menu, click Start Debugging to build and run the application.
2. When the Graph Demo window appears, click Plot Graph, and then wait.

*Please be patient. The application takes several seconds to generate and display the graph, and the application is unresponsive while this occurs (Chapter 24 explains why this is, and also instructs how to avoid this behavior). The following image shows the graph. Note the value in the Duration (ms) label in the following figure. In this case, the application took 4,938 mil­liseconds (ms) to plot the graph. Note that this duration does not include the time to actually render the graph, which might be another few seconds.*

***Note The application was run on a computer with 4 GB of memory and a quad-core processor running at 2.40 GHz. Your times might vary if you are using a slower or faster processor with a different number of cores, or a computer with a greater or lesser amount of memory.***

1. Click Plot Graph again, and take note of the time taken. Repeat this action several times to obtain an average value.

**Note** You might find that occasionally it takes an extended time for the graph to appear (more than 30 seconds). This tends to occur if you are running close to the memory capacity of your computer and Windows 8.1 has to page data between memory and disk. If you encounter this phenomenon, discard this time and do not include it when calculating your average.

1. Leave the application running and switch to the desktop. Right-click an empty area of the taskbar, and then, on the shortcut menu that appears, click Task Manager.
2. In the Task Manager window, click the Performance tab and display the CPU utilization. If the Performance tab is not visible, click More Details (it should appear). Right-click the CPU Utili­zation graph, point to Change Graph To, and then click Overall Utilization. This action causes Task Manager to display the utilization of all the processor cores running on your computer in a single graph.
3. Return to the Graph Demo application and adjust the display to show the application in the main part of the screen with the desktop appearing in the left-hand side. Ensure that you can see the Task Manager window displaying the CPU utilization.
4. Wait for the CPU utilization to level off, and then, in the Graph Demo window, click Plot Graph.
5. Wait for the CPU utilization to level off again, and then click Plot Graph again.
6. Repeat Step 16 several times, waiting for the CPU utilization to level off between clicks.
7. Switch to the Task Manager window and examine the CPU utilization. Your results will vary, but on a dual-core processor, the CPU utilization will probably be somewhere around 50–55 percent while the graph was being generated. On a quad-core machine, the CPU utilization will likely be somewhere between 25 and 30 percent, as shown in the image that follows. Note that other factors, such as the type of graphics card in your computer, can also impact the performance.
8. Return to Visual Studio 2013 and stop debugging.

You now have a baseline for the time the application takes to perform its calculations. However, it is clear from the CPU usage displayed by Task Manager that the application is not making full use of the processing resources available. On a dual-core machine, it is using just over half of the CPU power, and on a quad-core machine, it is employing a little over a quarter of the CPU. This phenomenon occurs because the application is single-threaded, and in a Windows application, a single thread can occupy only a single core on a multicore processor. To spread the load over all the available cores, you need to divide the application into tasks and arrange for each task to be executed by a separate thread running on a different core. This is what you will do in the following exercise.

## Modify the GraphDemo application to use *Task* objects

1. Return to Visual Studio 2013, and display the GraphWindow.xaml.cs file in the Code and Text Editor window, if it is not already open.
2. Examine the *generateGraphData* method.

The purpose of this method is to populate the items in the *data* array. It iterates through the array by using the outer *for* loop based on the *x* loop control variable.   
  
The calculation performed by one iteration of this loop is independent of the calculations performed by the other iterations. Therefore, it makes sense to partition the work performed by this loop and run different iterations on a separate processor.

1. Modify the definition of the *generateGraphData* method to take two additional *int* parameters called *partitionStart* and *partitionEnd*, as shown in bold in the following example:

private void generateGraphData(byte[] data, **int partitionStart, int partitionEnd)**{   
 ...   
}

1. In the *generateGraphData* method, change the outer *for* loop to iterate between the values of *partitionStart* and *partitionEnd*, as shown here in bold:

private void generateGraphData(byte[] data, int partitionStart, int partitionEnd)   
{   
 ...   
 for (**int x = partitionStart; x < partitionEnd; x++)** {   
 ...   
 }   
}

1. In the Code and Text Editor window, add the following *using* directive to the list at the top of the GraphWindow.xaml.cs file:

**using System.Threading.Tasks;**

1. In the *plotButton\_Click* method, comment out the statement that calls the *generateGraphData* method and add the statement shown in bold in the following code that creates a *Task* object and starts it running:

...   
Stopwatch watch = Stopwatch.StartNew();   
// generateGraphData(data);   
**Task first = Task.Run(() => generateGraphData(data, 0, pixelWidth / 4));**...

The task runs the code specified by the lambda expression. The values for the *partitionStart* and *partitionEnd* parameters indicate that the *Task* object calculates the data for the first half of the graph. (The data for the complete graph consists of points plotted for the values between 0 and *pixelWidth / 2*.)

1. Add another statement that creates and runs a second *Task* object on another thread, as shown in the following bold-highlighted code:

...   
Task first = Task.Run(() => generateGraphData(data, 0, pixelWidth / 4));   
**Task second = Task.Run(() => generateGraphData(data, pixelWidth / 4, pixelWidth / 2));**...

This *Task* object invokes the *generateGraph* method and calculates the data for the values between *pixelWidth / 4* and *pixelWidth / 2*.

1. Add the following statement shown in bold that waits for both *Task* objects to complete their work before continuing:

Task second = Task.Run(() => generateGraphData(data, pixelWidth / 4, pixelWidth / 2));   
**Task.WaitAll(first, second);**...

1. On the Debug menu, click Start Debugging to build and run the application. Adjust the display to show the application in the main part of the screen with the desktop appearing in the left side. As before, ensure that you can see the Task Manager window displaying the CPU utilization in the snapped view.
2. In the Graph Demo window, click Plot Graph. In the Task Manager window, wait for the CPU utilization to level off.
3. Repeat step 10 several more times, waiting for the CPU utilization to level off between clicks. Make a note of the duration recorded each time you click the button and calculate the average.

You should see that the application runs significantly quicker than it did previously. On my computer, the typical time dropped to 2,951 milliseconds—a reduction in time of about 40 percent.

In most cases, the time required to perform the calculations will be cut by nearly half, but the application still has some single-threaded elements, such as the logic that actually displays the graph after the data has been generated. This is why the overall time is still more than half the time taken by the previous version of the application.

1. Switch to the Task Manager window.  
     
   You should see that the application uses more cores of the CPU. On my quad-core machine, the CPU usage peaked at approximately 50 percent each time I clicked Plot Graph. This is because the two tasks were each run on separate cores, but the remaining two cores were left unoccupied. If you have a dual-core machine, you will likely see processor utilization briefly reach 100 percent each time the graph is generated.

If you have a quad-core computer, you can increase the CPU utilization and reduce the time further by adding two more *Task* objects and dividing the work into four chunks in the *plotButton\_Click* method, as shown here in bold:

...   
**Task first = Task.Run(() => generateGraphData(data, 0, pixelWidth / 8));   
Task second = Task.Run(() => generateGraphData(data, pixelWidth / 8, pixelWidth / 4));   
Task third = Task.Run(() => generateGraphData(data, pixelWidth / 4, pixelWidth \* 3 / 8));   
Task fourth = Task.Run(() => generateGraphData(data, pixelWidth \* 3 / 8, pixelWidth / 2));   
Task.WaitAll(first, second, third, fourth);**...

If you have only a dual-core processor, you can still try this modification, and you should notice a small beneficial effect on the time. This is primarily because of the way in which the algorithms used by the CLR optimize the way in which the threads for each task are scheduled.  
  
Use the Parallel class to parallelize operations in the GraphData application

1. Open the GraphDemo solution located in the \Microsoft Press\Visual CSharp Step By Step\Chapter 23\Parallel GraphDemo folder in the companion content. This is a copy of the original GraphDemo application. It does not use tasks yet.
2. In Solution Explorer, in the GraphDemo project, expand the GraphWindow.xaml node, and then double-click GraphWindow.xaml.cs to display the code for the form in the Code and Text Editor window.
3. Add the following *using* directive to the list at the top of the file:

**using System.Threading.Tasks;**

1. Locate the *generateGraphData* method.   
     
   The outer *for* loop that iterates through values of the integer variable *x* is a prime candidate for parallelization. You might also consider the inner loop based on the variable *i*, but this loop takes more effort to parallelize because of the type of *i*. (The methods in the *Parallel* class ex­pect the control variable to be an integer.) Additionally, if you have nested loops such as those that occur in this code, it is good practice to parallelize the outer loops first and then test to see whether the performance of the application is sufficient. If it is not, work your way through nested loops and parallelize them working from outer to inner loops, testing the performance after modifying each one. You will find that in many cases parallelizing outer loops has the most impact on performance, whereas the effects of modifying inner loops becomes more marginal.
2. Cut the code in the body of the *for* loop, and create a new private *void* method called *calculate Data* with this code. The *calculateData* method should take an *int* parameter called *x* and a byte array called *data*. Also, move the statements that declare the local variables *a, b*, and *c* from the *generateGraphData* method to the start of the *calculateData* method. The following code shows the *generateGraphData* method with this code removed and the *calculateData* method (do not try to compile this code yet):

private void generateGraphData(byte[] data)   
 {   
 for (int x = 0; x < a; x++)   
 {   
 }   
 }  
  
private void calculateData(int x, byte[] data)   
{   
 int a = pixelWidth / 2;   
 int b = a \* a;   
 int c = pixelHeight / 2;   
 int s = x \* x;   
 double p = Math.Sqrt(b - s);   
 for (double i = -p; i < p; i += 3)   
 {   
 double r = Math.Sqrt(s + i \* i) / a;   
 double q = (r - 1) \* Math.Sin(24 \* r);   
 double y = i / 3 + (q \* c);   
 plotXY(data, (int)(-x + (pixelWidth / 2)), (int)(y + (pixelHeight / 2)));   
 plotXY(data, (int)(x + (pixelWidth / 2)), (int)(y + (pixelHeight / 2)));   
 }   
}

1. In the *generateGraphData* method, change the *for* loop to a statement that calls the static *Parallel.For* method, as shown in bold in the following code:

private void generateGraphData(byte[] data)   
{   
 **Parallel.For(0, pixelWidth / 2, x => calculateData(x, data));**}

This code is the parallel equivalent of the original *for* loop. It iterates through the values from 0 to *pixelWidth / 2 – 1* inclusive. Each invocation runs by using a task, and each task might run more than one iteration. The *Parallel.For* method finishes only when all the tasks it has created complete their work. Remember that the *Parallel.For* method expects the final parameter to be a method that takes a single integer parameter. It calls this method passing the cur­rent loop index as the parameter. In this example, the *calculateData* method does not match the required signature because it takes two parameters: an integer and a byte array. For this reason, the code uses a lambda expression that acts as an adapter that calls the *calculateData* method with the appropriate arguments.

1. On the Debug menu, click Start Debugging to build and run the application.
2. In the Graph Demo window, click Plot Graph. When the graph appears in the Graph Demo window, record the time taken to generate the graph. Repeat this action several times to get an average value.

You should notice that the application runs at a comparable speed to the previous version that used *Task* objects (and possibly slightly faster, depending on the number of CPUs you have available). If you examine Task Manager, you should notice that the CPU usage peaks closer to 100 percent regardless of whether you have a dual-core or quad-core computer.

1. Return to Visual Studio and stop debugging.