

Name _____

Points _____ Effort in hours _____

1. Dish of the Day: "Almondbreads"**(4 + 8 + 8 + 4 Points)**

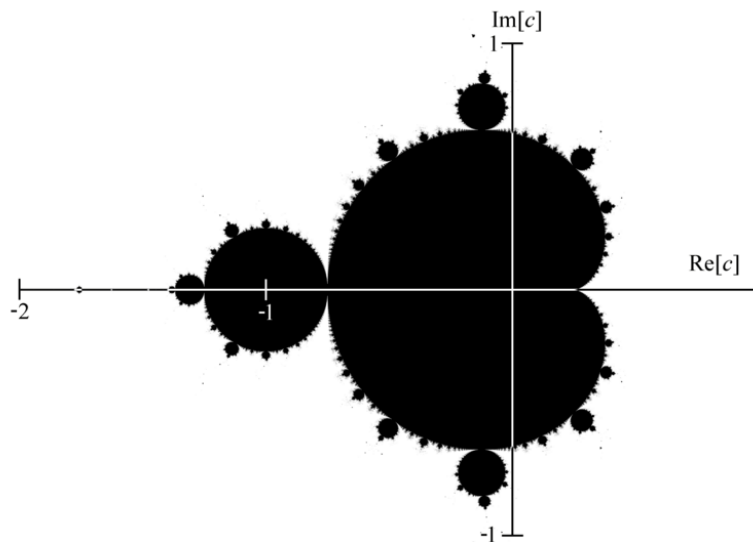
In this exercise we would like to take a look at a very special form of bread: the "Almondbread" or in other words the *Mandelbrot*. However, the Mandelbrot is not a common form of bread. It is very special (and delicious) and as a consequence, to bake a Mandelbrot we cannot just use normal grains. Instead we need special or complex grains. The recipe is the following:

The Mandelbrot set is the set of complex numbers c , for which the following (recursive) sequence of complex numbers z_n

$$z_0 = 0$$
$$z_{n+1} = z_n^2 + c$$

doesn't diverge towards infinity. If you are not so familiar with complex numbers (anymore), a short introduction can be found at the end of this exercise sheet.

If you mark these points of the Mandelbrot set in the complex plane, you get the very characteristic picture of the set (also called "Apfelmännchen" in German). The set occupies approximately the area from $-2-i$ to $1+i$:



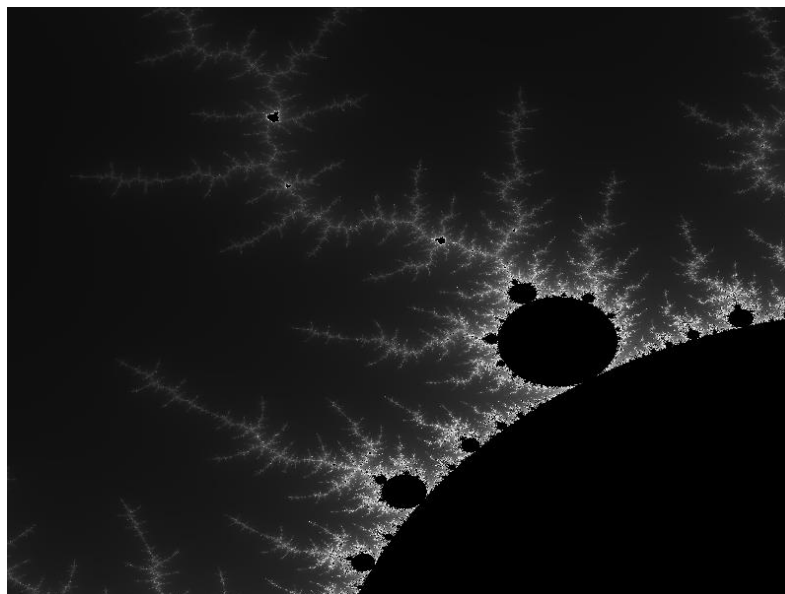
To get even more interesting and artistic pictures the points outside the Mandelbrot set can be colored differently depending on how fast the sequence diverges towards infinity. Therefore just define an upper limit for the absolute value of z_n (usually 4). If z_n grows larger than this upper limit, it can be assumed that z_n will keep growing and will finally diverge. On the other hand if this upper limit is not exceeded in a predefined number of iterations (usually 10.000), it can be assumed that the sequence will not diverge and that the starting point c consequently is element of the Mandelbrot set. Depending on how fast z_n goes beyond the limit (number of iterations) the starting point c can be colored.

- a) Write a simple generator in C# using the .NET Windows Forms framework that calculates and displays the Mandelbrot set. Additionally, the generator should have the feature to zoom into the set. Therefore, the user should be able to draw a selection rectangle into the current picture of the set which marks the new section that should be displayed. By clicking on the right mouse button, the original picture ($-2-i$ to $1+i$) should be generated again.
- b) Take care that the calculation of the points is computationally expensive. Consequently, it is reasonable to use a separate (worker) thread, so that the user interface stays reactive during the generation of a new picture. However, it can be the case that the user selects a new section before the generation of a previous selection is finished. Furthermore, the time needed for the generation of a picture is variable depending on how many points of the Mandelbrot set are included in the current selection. So, it can also happen that the calculation of a latter selected part is finished before an earlier selected one. Therefore, synchronization is necessary to coordinate the different worker threads.

Implement at least two different ways to create and manage your worker threads (for example you can use `BackgroundWorker`, threads from the thread pool, plain old thread objects, asynchronous delegates, etc.). Explain how synchronization and management of the worker threads is done in each case.

- c) Think about what's the best way to partition the work and to spread it among the workers. Based on these considerations implement a parallel version of the Mandelbrot generator in C# without using the Task Parallel Library (or `Parallel.For`).
- d) Measure the runtime of the sequential and parallel version needed to display the section $-1,4-0,1i$ to $-1,32-0,02i$ with a resolution of 800 times 600 pixels. Execute 10 independent runs and document also the mean runtimes and the standard deviations.

For self-control, the generated picture could look like this:



Appendix: Calculations with Complex Numbers

As you all know, it is quite difficult to calculate the square root of negative numbers. However, many applications (electrical engineering, e.g.) require roots of negative numbers leading to the extension of real to complex numbers. So it is necessary to introduce a new number, the imaginary number i , which is defined as the square root of -1. A complex number c is of the form

$$c = a + b \cdot i$$

where a is called the real and b the imaginary part. a and b themselves are normal real numbers.

As a consequence of this special form calculations with complex numbers are a little bit more tricky than in the case of real numbers. The basic arithmetical operations are defined as follows:

$$\begin{aligned}(a + b \cdot i) + (c + d \cdot i) &= (a + c) + (b + d) \cdot i \\(a + b \cdot i) - (c + d \cdot i) &= (a - c) + (b - d) \cdot i \\(a + b \cdot i) \cdot (c + d \cdot i) &= (ac - bd) + (bc + ad) \cdot i \\ \frac{a + b \cdot i}{c + d \cdot i} &= \frac{ac + bd}{c^2 + d^2} + \frac{bc - ad}{c^2 + d^2} \cdot i\end{aligned}$$

Furthermore we also need the absolute value (distance to 0+0i) of complex numbers which can be calculated easily using the theorem of Pythagoras:

$$\text{Abs}(a + b \cdot i) = \sqrt{a^2 + b^2}$$

1 Dish of the Day: Älmondbreads"

1a. Generator

Der Generator wurde bereits in der Übung programmiert und baut auf vorhandenem Code auf.

```
1 using System;
2 using System.Diagnostics;
3 using System.Drawing;
4
5 namespace MandelbrotGenerator
6 {
7     /// <summary>
8     /// Image generate on sync implementation
9     /// </summary>
10    public class SyncImageGenerator : IImageGenerator
11    {
12        /// <summary>
13        /// Flag used for cancellation
14        /// </summary>
15        protected bool cancel;
16
17        protected Bitmap GenerateBitmap(Area area)
18        {
19            int maxIterations;
20            double zBorder;
21            double cReal, cImg, zReal, zImg, zNewReal, zNewImg;
22
23            maxIterations = Settings.DefaultSettings.MaxIterations;
24            zBorder = Settings.DefaultSettings.ZBorder*Settings.DefaultSettings.ZBorder;
25
26            Bitmap bitmap = new Bitmap(area.Width, area.Height);
27
28            for (int i = 0; i < area.Width; i++)
29            {
30                for (int j = 0; j < area.Height; j++)
31                {
32                    cReal = area.MinReal + i*area.PixelWidth;
33                    cImg = area.MinImg + j*area.PixelHeight;
34                    zReal = 0;
35                    zImg = 0;
36
37                    int k = 0;
38                    while ((zReal*zReal + zImg + zImg < zBorder)
39                        && (k < maxIterations))
40                    {
41                        //check canecallation
42                        if (cancel) return null;
43
```

```
44         zNewReal = zReal*zReal - zImg*zImg + cReal;
45         zNewImg = 2*zReal*zImg + cImg;
46
47         zReal = zNewReal;
48         zImg = zNewImg;
49
50         k++;
51     }
52
53     bitmap.SetPixel(i, j, ColorSchema.GetColor(k));
54 }
55 }
56 return bitmap;
57 }
58
59 /// <summary>
60 /// Generate the given area
61 /// </summary>
62 /// <param name="area"></param>
63 public virtual void GenerateImage(Area area)
64 {
65     Stopwatch sw = new Stopwatch();
66     sw.Start();
67     Bitmap bm = GenerateBitmap(area);
68     sw.Stop();
69
70     //bm is null on cancellation
71     if (bm != null)
72     {
73         OnImageGenerated(area, bm, sw.Elapsed);
74     }
75
76 }
77
78 /// <summary>
79 /// Event when the generation is finished
80 /// </summary>
81 public event EventHandler<EventArgs<Tuple<Area, Bitmap, TimeSpan>>> ImageGenerated;
82
83 /// <summary>
84 /// Image generated event invoker
85 /// </summary>
86 /// <param name="area"></param>
87 /// <param name="bitmap"></param>
88 /// <param name="timespan"></param>
89
90 protected virtual void OnImageGenerated(Area area, Bitmap bitmap, TimeSpan timespan)
91 {
92     var handler = ImageGenerated;
```

```
93         handler?.Invoke(this,
94             new EventArgs<Tuple<Area, Bitmap, TimeSpan>>(new Tuple<Area, Bitmap, TimeSpan>(ar
95         })
96     }
97 }
```

1b. Asynchrone Implementierung

Um Asynchrone Implementierungen zu ermöglichen wurde das Interface IImageGenerator um das Event ImageGenerated erweitert. Weiters muss bei der Zuweisung des Ergebnisses an die GUI der Thread gewechselt werden.

Die folgenden Implementierungen leiten vom originalen SyncImageGenerator ab und verlagern die Arbeit in den Hintergrund. Dabei wurde eine einfache Möglichkeit um Abbrechen der Berechnung implementiert. Hierzu wurde die Implementierung um ein simples Stop-Flag erweitert. (Code siehe 1.a)

Implementierung 1: AsyncImageGenerator

Bei dieser Implementierung wurden Threads verwendet um die Arbeit in den Hintergrund zu verlagern. Die Prüfung ob bereits eine Berechnung läuft, wird über die Abfrage des Thread Status IsAlive durchgeführt. Ist noch eine Berechnung aktiv, wird das Stop-Flag gesetzt und dann auf die Beendigung des Threads gewartet. *Anmerkung:* Auf die Beendigung wird am GUI Thread gewartet, da der Hintergrundthread bei jedem Schleifendurchgang prüft ob das Stop-Flag gesetzt ist. Damit ist die Zeit sehr kurz und kann am GUI-Thread ausgeführt werden.

```
1 using System.Threading;
2
3 namespace MandelbrotGenerator
4 {
5     /// <summary>
6     /// Async version using threads
7     /// </summary>
8     public class AsyncImageGenerator:SyncImageGenerator
9     {
10         private Thread workerThread;
11
12         public override void GenerateImage(Area area)
13         {
14             //Check if calculation is running and cancel it
15             if (workerThread != null && workerThread.IsAlive)
16             {
17                 cancel = true;
18                 workerThread.Join();
19             }
20             cancel = false;
21             workerThread = new Thread(Run);
22             workerThread.Start(area);
23         }
24     }
```

```
24
25     private void Run(object o)
26     {
27         Area area = o as Area;
28         //call base logic
29         base.GenerateImage(area);
30     }
31 }
32 }
```

Implementierung 2: AsyncWorkerImageGenerator

Die AsyncWorkerImageGenerator Implementierung verwendet einen .NET BackgroundWorker um die Arbeit in den Hintergrund zu verlagern. Die Ergebnisweiergabe wird über das Event RunWorkerCompleted und die Eigenschaft Result abgewickelt. Die Synchronisierung der Berechnungen wird über die Eigenschaft IsBusy der Klasse BackgroundWorker gelöst.

```
1 using System;
2 using System.ComponentModel;
3 using System.Diagnostics;
4 using System.Drawing;
5
6 namespace MandelbrotGenerator
7 {
8     /// <summary>
9     /// Async background worker implementation
10    /// </summary>
11    public class AsyncWorkerImageGenerator: SyncImageGenerator
12    {
13        private BackgroundWorker worker;
14
15        /// <summary>
16        /// Generate the given area
17        /// </summary>
18        /// <param name="area"></param>
19        public override void GenerateImage(Area area)
20        {
21            if (worker != null && worker.IsBusy)
22            {
23                cancel = true;
24            }
25
26            worker = new BackgroundWorker();
27            worker.DoWork += DoWork;
28            worker.RunWorkerCompleted += OnWorkCompleted;
29            cancel = false;
30            worker.RunWorkerAsync(area);
31        }
32    }
```

```
32
33     private void OnWorkCompleted(object sender, RunWorkerCompletedEventArgs e)
34     {
35         (sender as BackgroundWorker).RunWorkerCompleted -= OnWorkCompleted;
36
37         Tuple<Area, Bitmap, TimeSpan> res = e.Result as Tuple<Area, Bitmap, TimeSpan>;
38         if (res == null)
39         {
40             throw new InvalidOperationException("Result is null");
41         }
42         //bitmap is null on cancellation
43         if (res.Item2 != null)
44         {
45             OnImageGenerated(res.Item1, res.Item2, res.Item3);
46         }
47     }
48
49     private void DoWork(object sender, DoWorkEventArgs e)
50     {
51         Area area = e.Argument as Area;
52         if (area == null)
53             throw new InvalidOperationException("First argument area cannot be null");
54
55         Stopwatch sw = new Stopwatch();
56         sw.Start();
57         Bitmap bm = GenerateBitmap(area);
58         sw.Stop();
59         e.Result = new Tuple<Area, Bitmap, TimeSpan>(area,bm,sw.Elapsed);
60     }
61 }
62
63 }
64 }
```

1c. Parallele Version

Die Aufteilung der Arbeit auf die Worker kann Zeilenweise erfolgen. Dabei wird die benötigte Anzahl an Worker-Threads erzeugt, wobei jeder dieser Threads immer eine Zeile berechnet. Sobald alle Zeilen berechnet sind, werden die Threads beendet. Diese Aufteilung hat den Vorteil, dass die Threads ungefähr gleich ausgelastet sind, da zwei benachbarte Zeilen ungefähr den selben Rechnaufwand verursachen.

Zur Verwaltung der Berechnung wird ein zusätzlicher Management-Thread verwendet. Dieser übernimmt die Zeitmessung und das Auslösen des Events. Der Zugriff auf das Bitmap wird mit Hilfe von Locks gelöst.

```
1 using System;
2 using System.Diagnostics;
3 using System.Drawing;
```



```
4 using System.Linq;
5 using System.Threading;
6
7 namespace MandelbrotGenerator
8 {
9     /// <summary>
10    /// Paralell implementation
11    /// </summary>
12    class ParallelImageGenerator : IImageGenerator
13    {
14        private readonly object bitmapLock = new object();
15        private readonly object indexLock = new object();
16        private int currentIndex;
17        private Bitmap result;
18        private Thread[] threads;
19        private Thread managementThread;
20        private bool cancel;
21
22        /// <summary>
23        /// Generate the given area
24        /// </summary>
25        /// <param name="area"></param>
26        public void GenerateImage(Area area)
27        {
28            //check canecllation
29            if (managementThread != null && managementThread.IsAlive)
30            {
31                cancel = true;
32                managementThread.Join();
33            }
34
35            managementThread = new Thread(ManagementThreadMethod);
36            managementThread.Start(area);
37        }
38
39        public event EventHandler<EventArgs<Tuple<Area, Bitmap, TimeSpan>>> ImageGenerated;
40
41        protected virtual void OnImageGenerated(Area area, Bitmap bitmap, TimeSpan time)
42        {
43            ImageGenerated?.Invoke(this,
44                new EventArgs<Tuple<Area, Bitmap, TimeSpan>>(new Tuple<Area, Bitmap, TimeSpan>(area, bitmap, time)));
45        }
46
47        public void ManagementThreadMethod(object input)
48        {
49            cancel = false;
50            Area area = input as Area;
51            Stopwatch sw = new Stopwatch();
52
```

```
53         sw.Start();
54         currentIndex = 0;
55         result = new Bitmap(area.Width, area.Height);
56         threads = new Thread[Settings.DefaultSettings.Workers];
57         //Create worker threads
58         for (int i = 0; i < threads.Length; i++)
59         {
60             Thread t = new Thread(WorkerThreadMethod);
61             threads[i] = t;
62             t.Start(area);
63         }
64         //Wait for finishing
65         foreach (var thread in threads)
66         {
67             thread.Join();
68         }
69         sw.Stop();
70         //check cancellation
71         if (!cancel)
72         {
73             OnImageGenerated(area, result, sw.Elapsed);
74         }
75     }
76 }
77
78 public void WorkerThreadMethod(object input)
79 {
80     Area area = input as Area;
81
82     while (!cancel)
83     {
84         int index;
85         lock (indexLock)
86         {
87             index = currentIndex;
88             currentIndex++;
89         }
90
91         //for all threads which are already finished
92         if (index >= area.Height)
93         {
94             return;
95         }
96         GenerateBitmapLine(area, index);
97     }
98 }
99
100 /// <summary>
101 /// Generates one line of the bitmap
```

```
102     /// </summary>
103     /// <param name="area"></param>
104     /// <param name="line"></param>
105     public void GenerateBitmapLine(Area area, int line)
106     {
107         int maxIterations;
108         double zBorder;
109         double cReal, cImg, zReal, zImg, zNewReal, zNewImg;
110
111         maxIterations = Settings.DefaultSettings.MaxIterations;
112         zBorder = Settings.DefaultSettings.ZBorder*Settings.DefaultSettings.ZBorder;
113
114
115         for (int i = 0; i < area.Width; i++)
116         {
117             int j = line;
118             cReal = area.MinReal + i*area.PixelWidth;
119             cImg = area.MinImg + j*area.PixelHeight;
120             zReal = 0;
121             zImg = 0;
122
123             int k = 0;
124             while ((zReal*zReal + zImg + zImg < zBorder)
125                 && (k < maxIterations))
126             {
127                 //check cancellation
128                 if (cancel)
129                 {
130                     return;
131                 }
132
133                 zNewReal = zReal*zReal - zImg*zImg + cReal;
134                 zNewImg = 2*zReal*zImg + cImg;
135
136                 zReal = zNewReal;
137                 zImg = zNewImg;
138
139                 k++;
140             }
141             lock (bitmapLock)
142             {
143                 result.SetPixel(i, j, ColorSchema.GetColor(k));
144             }
145         }
146     }
147 }
148 }
```

1d. Performance

Parameter

- Ausschnitt: -1,4-0,1i bis -1,32-0,02i
- Auflösung: 800x600
- WorkerThreads: 8 (parallele Implementierung)

Tabelle 1: Ergebnisse

Lauf	Simulation	
	Sync [ms]	Parallel [ms]
1	5584	1139
2	5581	1163
3	5585	1139
4	5580	1129
5	5591	1164
6	5601	1142
7	5574	1125
8	5602	1163
9	5580	1146
10	5606	1133
Ø	5588	1144
Std. Abw.	10,461	13,733

- SpeedUp: 4,885