VPS5 - UE4

Stefan Kert

8. Mai 2016

1 Psychedelic Diffusions

1.1 Synchrone Implementierung des Generators

Der erste Teil der Aufgabe besteht darin, einen Generator zu implementieren, der die geforderten Kriterien realisiert. Dies wurde bereits im Unterricht realisiert und es sollte daher hier nicht näher auf diesen Generator eingegangen werden. Der Vollständigkeit halber sollte hier aber der Quellcode für die GenerateBitmap Methode gezeigt werden.

```
public class SyncImageGenerator : ImageGenerator
{
    public override Bitmap GenerateBitmap(Area area) {
        var matrix = area.Matrix;
        var height = area.Height;
        var width = area.Width;
        var newMatrix = new double[width, height];
        for (var i = 0; i < width; i++) {
            for (var j = 0; j < height; j++) {
                var jp = (j + height - 1) % height;
                var jm = (j + 1) % height;
                var ip = (i + 1) % width;
                var im = (i + width - 1) % width;

                newMatrix[i, j] = (matrix[i, jp] + matrix[i, jm] +
                    matrix[ip, j] + matrix[im, j] + matrix[ip, jp] +
                    matrix[ip, jm] + matrix[im, jp] + matrix[im, jm])
                    / 8.0;
            }
        }
        Bitmap bm = new Bitmap(width, height);
        ColorBitmap(newMatrix, width, height, bm);
        area.Matrix = newMatrix;
        return bm;
    }
}
```

Die Methode in welcher der obige Code aufgerufen wird, sodass die benötigten Iteration durchgeführt werden befindet sich im nächsten Listing.

```
public abstract class ImageGenerator: IImageGenerator
{
    private bool _started;
    private bool _stopRequested;
    public bool StopRequested => _stopRequested;
    public bool Started => _started;


    public void GenerateImage(Area area) {
            _stopRequested = false;
            _started = true;
            var swOverall = new Stopwatch();
            swOverall.Start();
            for (var i = 0; i <
                Settings.DefaultSettings.MaxIterations; i++) {
                var sw = new Stopwatch();
                sw.Start();
                var bm = GenerateBitmap(area);
                OnImageGenerated(area, bm, sw.Elapsed);
                sw.Stop();
            }
            swOverall.Stop();
            OnCalculationFinished(swOverall.Elapsed);
            _started = false;
    }

    public abstract Bitmap GenerateBitmap(Area area);

    public virtual void ColorBitmap(double[,] array, int width, int
        height, Bitmap bm) {
        var maxColorIndex = ColorSchema.Colors.Count - 1;

        for (var i = 0; i < width; i++) {
            for (var j = 0; j < height; j++) {
                var colorIndex = (int)array[i, j]%maxColorIndex;
                bm.SetPixel(i, j, ColorSchema.Colors[colorIndex]);
            }
        }
    }

    public event EventHandler<EventArgs<Tuple<TimeSpan>>>
        CalculationFinished;
    public event EventHandler<EventArgs<Tuple<Area, Bitmap,
        TimeSpan>>> ImageGenerated;

    protected void OnImageGenerated(Area area, Bitmap bitmap,
        TimeSpan timespan) {
        ImageGenerated?.Invoke(this, new EventArgs<Tuple<Area,
            Bitmap, TimeSpan>>(new Tuple<Area, Bitmap,
            TimeSpan>(area, bitmap, timespan)));
    }
    protected void OnCalculationFinished(TimeSpan timespan)
    {
        CalculationFinished?.Invoke(this, new
```

```
                EventArgs<Tuple<TimeSpan>>(new
                Tuple<TimeSpan>(timespan)));
        }

        public virtual void Stop() {
            _stopRequested = true;
            OnCalculationFinished(new TimeSpan());
        }
    }
```

Der gezeigte Code wurde nur um eine Methode und ein Event erweitert, welches aufgerufen wird, sobald die alle Iterationen durchgeführt wurden und die Berechnung beendet wurde, sodass auch ein Gesamtergebnis für die benötigte Zeit angezeigt werden kann.

1.2 Locking, Synchronisierung und Asynchrone Ausführung

Ein weiterer Bestandteil war das Implementieren der Funktionalität zum Abbrechen des Vorganges, sowie zum Hinzufügen des - Reheating - mit dem weitere Elemente durch einen Mausklick erzeugt werden können, weiters sollte die Berechnung asynchron durchgeführt werden, sodass die Benutzeroberfläche während der Ausführen reagiert. Für diese Teile ist es einerseits notwendig eine Möglichkeit zu finden, wie der Thread zum Generieren der Bilder abgebrochen bzw. gestoppt werden kann. Für das Abbrechen der Berechnung wurde mittels der .NET Klasse *CancellationToken* eine Möglichkeit hinzugefügt, den Vorgang abzubrechen. Um die Operation asynchron durchzuführen, wurde die Berechnung einfach mittels *Task.Run()* in einem eigenen Thread ausgeführt. Durch die Verwendung von Events gibt es hierbei kein Synchronisierungsproblem, da beim Beenden die View mittels dieser Events über die Ergebnisse informiert wird, jedoch muss in beim Zugriff auf UI-Komponenten darauf geachtet werden, dass diese Operationen auf dem GUI Thread passieren, da es ansonsten zu Fehlern kommt. Für das Reheating wird mit Hilfe der Klasse *AutoResetEvent*, welche sperrt wenn gerade eine Berechnung im Gange ist und mit Hilfe der Klasse *ManuelResetEvent* wird die Berechnung gesperrt sobald der Benutzer mit der Maus auf die *PictureBox* klickt um ein weiteres Element zu erzeugen. Im folgenden Listing wird gezeigt wie die Implementierung sich im Detail gestaltet.

```
using System;
using System.Diagnostics;
using System.Drawing;
using System.Threading;
using System.Threading.Tasks;

namespace Diffusions
{
    public abstract class ImageGenerator: IImageGenerator
    {
        private CancellationTokenSource _source;
        private bool _started;
        private bool _stopRequested;
        public ManualResetEvent Signal = new ManualResetEvent(true);
        public AutoResetEvent GeneratorPendingSignal = new
            AutoResetEvent(false);
```

```
public bool StopRequested => _stopRequested;
public bool Started => _started;


public void GenerateImage(Area area) {
    _source = new CancellationTokenSource();
    _stopRequested = false;
    _started = true;
    Task.Run(() => {
        var swOverall = new Stopwatch();
        swOverall.Start();
        for (var i = 0; i <
            Settings.DefaultSettings.MaxIterations; i++) {
            Signal.WaitOne();
            GeneratorPendingSignal.Reset();
            if (_source.Token.IsCancellationRequested)
                return;
            var sw = new Stopwatch();
            sw.Start();
            var bm = GenerateBitmap(area);
            OnImageGenerated(area, bm, sw.Elapsed);
            sw.Stop();
            GeneratorPendingSignal.Set();
        }
        swOverall.Stop();
        OnCalculationFinished(swOverall.Elapsed);
        _started = false;
    }, _source.Token);
}

public abstract Bitmap GenerateBitmap(Area area);

public virtual void ColorBitmap(double[,] array, int width, int
    height, Bitmap bm) {
    var maxColorIndex = ColorSchema.Colors.Count - 1;

    for (var i = 0; i < width; i++) {
        for (var j = 0; j < height; j++) {
            var colorIndex = (int)array[i, j]%maxColorIndex;
            bm.SetPixel(i, j, ColorSchema.Colors[colorIndex]);
        }
    }
}

public event EventHandler<EventArgs<Tuple<TimeSpan>>>
    CalculationFinished;
public event EventHandler<EventArgs<Tuple<Area, Bitmap,
    TimeSpan>>> ImageGenerated;

protected void OnImageGenerated(Area area, Bitmap bitmap,
    TimeSpan timespan) {
    ImageGenerated?.Invoke(this, new EventArgs<Tuple<Area,
        Bitmap, TimeSpan>>(new Tuple<Area, Bitmap,
```

```
                TimeSpan>(area, bitmap, timespan)));
        }
        protected void OnCalculationFinished(TimeSpan timespan)
        {
            CalculationFinished?.Invoke(this, new
                EventArgs<Tuple<TimeSpan>>(new
                Tuple<TimeSpan>(timespan)));
        }

        public virtual void Stop() {
            _stopRequested = true;
            _source.Cancel();
            OnCalculationFinished(new TimeSpan());
        }
    }
}
```

```
        private void pictureBox_MouseUp(object sender, MouseEventArgs e)
            {
            int x = e.X;
            int y = e.Y;

            if (!_generator.Started || e.Button != MouseButtons.Left)
                return;

            Task.Run(() => {
                _generator.GeneratorPendingSignal.WaitOne(); // Wait
                    until current calculation has ended
                _generator.Signal.Reset();
                Reheat(_currentArea.Matrix, x, y, _currentArea.Width,
                    _currentArea.Height, _TIP_SIZE, _DEFAULT_HEAT);
                _generator.Signal.Set();
            });
        }
```

1.3 Parallele Implementierung des Generators

Der letzte Teil der ersten Aufgabe war es, eine parallele Implementierung der synchronen *GenerateImage* Methode zu implementieren. Auch diese Aufgabe wurde bereits während der Übungsstunde erledigt und es sollte hier lediglich der Code gezeigt werden, sowie kurz beschrieben werden wie sich der SpeedUp gestaltet. Für die Implementierung wurden die Parallel Erweiterungen der TPL zur verwendet, da es mit diesen sehr einfach ist, Schleifen parallel durchzuführen und sich im Hintergrund die TPL um das verwalten der einzelnen Threads kümmert.

```
    public class ParallelImageGenerator : ImageGenerator
    {
        public override Bitmap GenerateBitmap(Area area) {
            var matrix = area.Matrix;
            var height = area.Height;
            var width = area.Width;
```

```
            var newMatrix = new double[width, height];

            Parallel.For(0, width, i => {
                for (var j = 0; j < height; j++) {
                    var jp = (j + height - 1)%height;
                    var jm = (j + 1)%height;
                    var ip = (i + 1)%width;
                    var im = (i + width - 1)%width;

                    newMatrix[i, j] = (matrix[i, jp] + matrix[i, jm] +
                        matrix[ip, j] + matrix[im, j] + matrix[ip, jp] +
                        matrix[ip, jm] + matrix[im, jp] + matrix[im,
                        jm])/8.0;
                }
            });
            Bitmap bm = new Bitmap(width, height);
            ColorBitmap(newMatrix, width, height, bm);
            area.Matrix = newMatrix;
            return bm;
        }
    }
```

Da

2 Stock Data Visualization

Die zweite Aufgabe bestand darin, eine Applikation, welche mittels Webrequest Daten von einem Webservice ausliest und diese in Form von Diagrammen anzeigt anzupassen, sodass diese asynchron ausgeführt wird. Im Template war bereits eine Methode zum synchronen auslesen der Daten vorhanden. Diese hatte das Problem, dass während dem Auslesen keine Interatkion mit der View möglich war, weshalb diese Methode wie bereits beschrieben in eine asynchrone Methode umgewandelt werden sollte. Es sollte für diese Aufgabe eine Variante implementiert werden, welche die asynchrone (parallele) Ausführung laut dem gegebenen Aktivitätsdiagramm mittels *Continuations* realisiert, sowie eine Variante welche dies mit dem seit .Net 4.5.2 neuen *async/await* Sprachfeature ermöglicht.

2.1 Continuations Implementierung

2.2 Async/await Implementierung