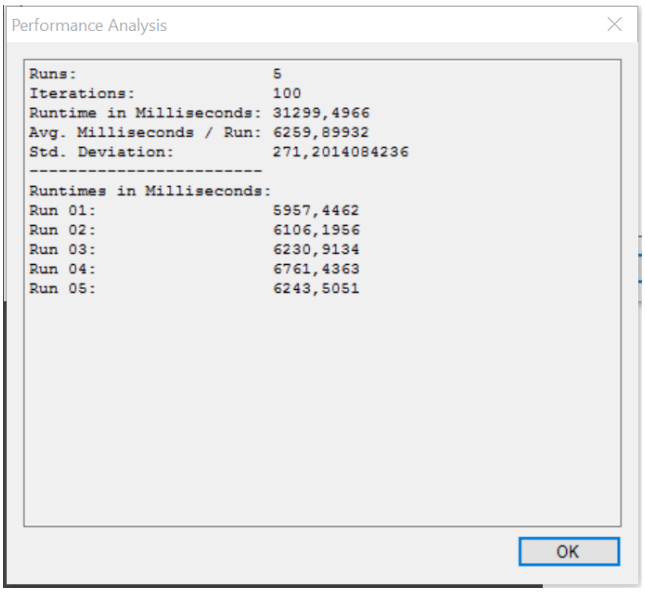
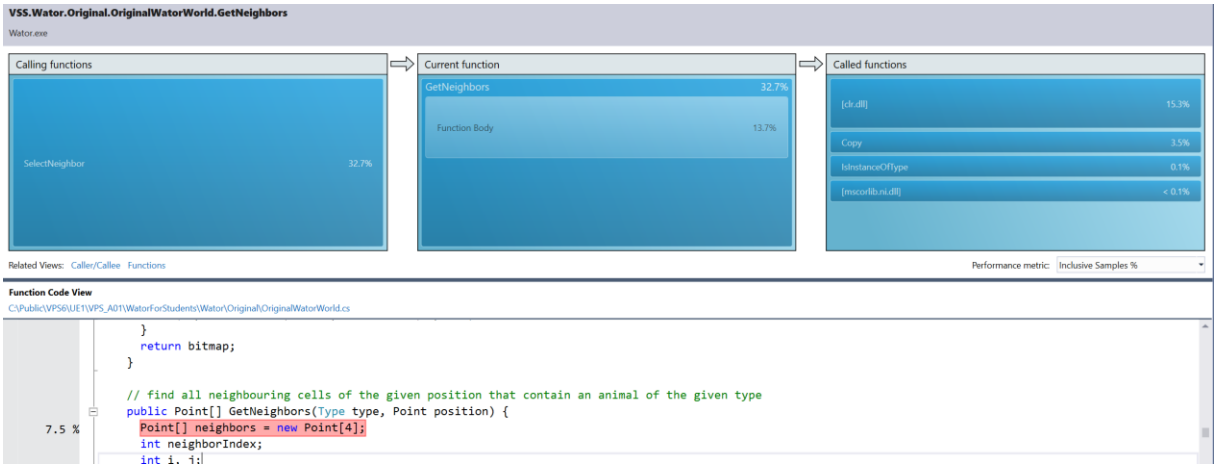
1. Analyse:

Ein erstes ausführen ohne Grafischer Oberfläche mit 100 Iterationen, liefert uns einen Ausgangspunkt an werten, mit welchen wir die Verbesserungen später vergleichen können.

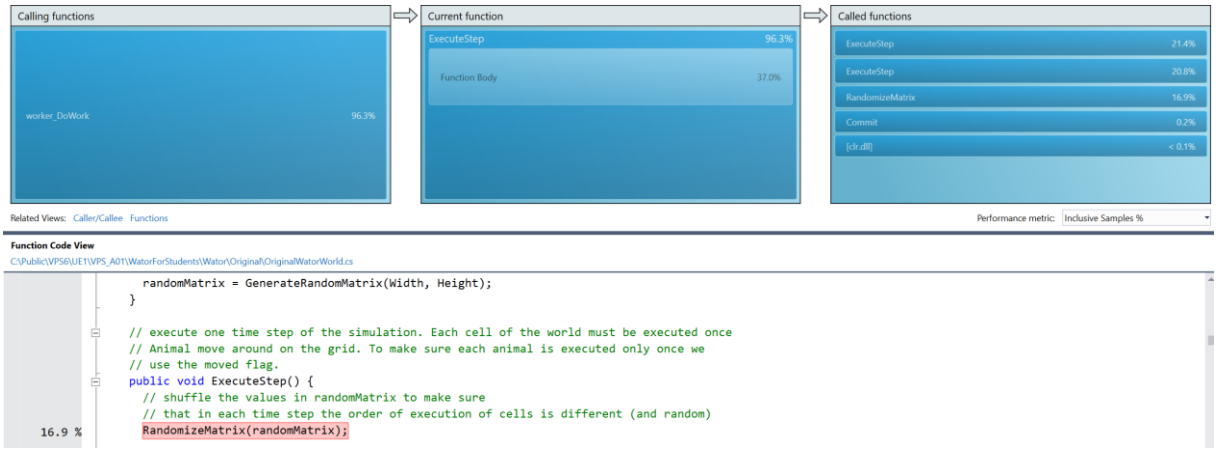


1.1. Wo geht Performance und Rechenleistung verloren?

Die Methode SelectNeighbor , sowie die dort aufgerufene Funktion GetNeighbors verbrauchen sehr viel an Zeit. Dabei fällt auch auf, dass das immer wiederkehrende anlegen von Point Arrays sich sehr negativ auf die Laufzeit auswirkt. Da diese Methoden sowohl für den Hai als auch für den Fisch ausgeführt werden, wirken diese auch gleich doppelt so stark.

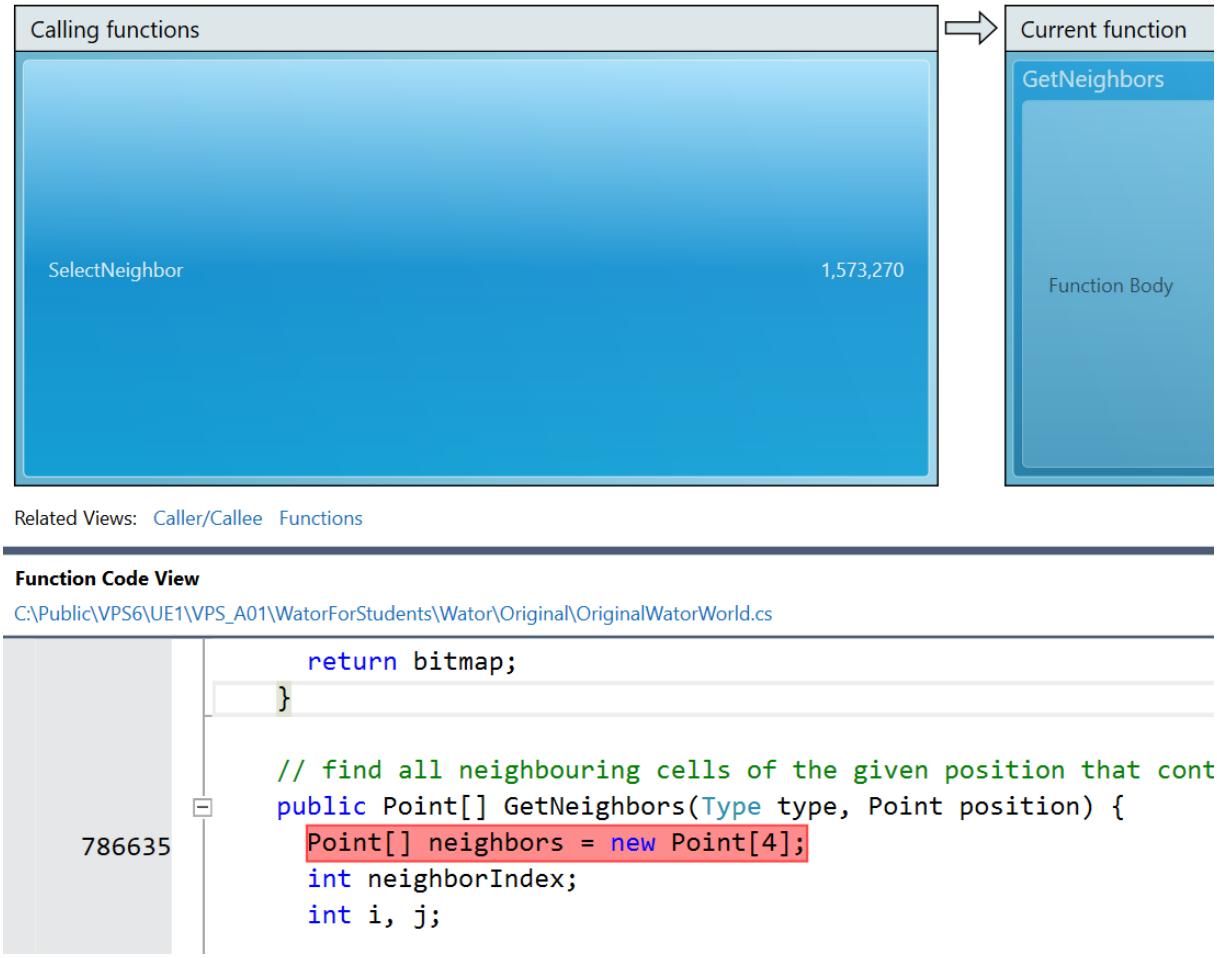


Das zufällige shuffeln der Matrix ist ein weiterer Punkt, welcher einen hohen Performance-Verlust verursacht.



Hauptaugenmerk kann man also auf ExecuteStep -> RandomizeMatrix und SelectNeighbor -> GetNeighbors legen, da dort die meiste Performance verloren geht.

Wenn man sich auch die Memory-Allocations ansieht, so sieht man deutlich, dass in der Funktion SelectNeighbor die allokierung von Point[] ein deutliches Problem darstellen.



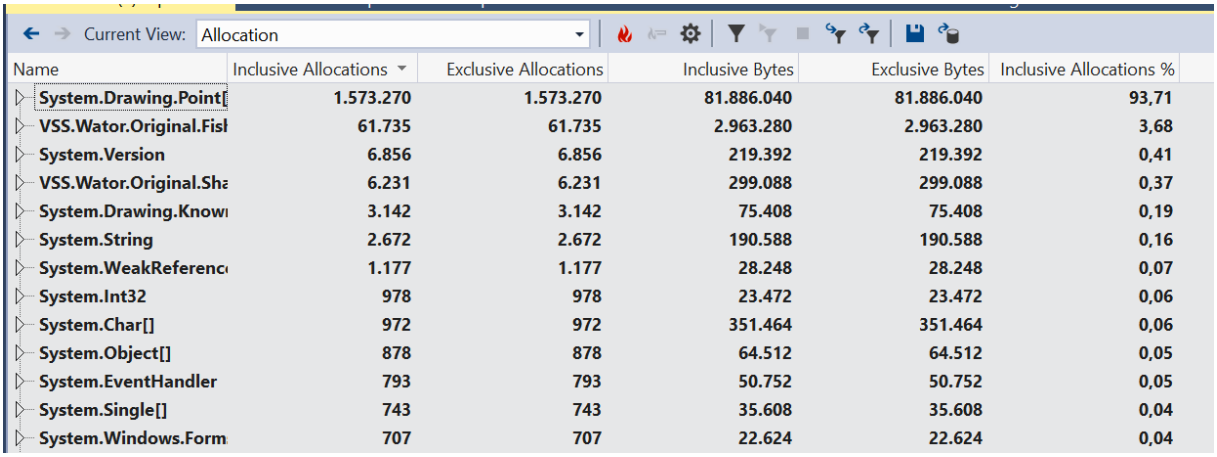
```
                neighborIndex++;
            }
        }

        // create a result array of the correct length containing only
        // the discovered cells of the correct type
786635  Point[] result = new Point[neighborIndex];
        Array.Copy(neighbors, result, neighborIndex);
```

Vor allem wird es zweimal angelegt und danach hineinkopiert, was Laufzeit und Memory-Allokierungen in die höhe treibt.

Bei dem Spawn vom Fisch, könnte man auch etwas verbessern, da wird wieder ein SelectNeighbor ausgeführt u das Ergebniss Point danach geprüft ob nun ein neuer Fisch erstellt werden kann. Hier wird ebenfalls unnötig Speicher allokiert.

```
        // implements spawning behaviour of fish
        protected override void Spawn() {
          // find an empty neighbouring cell
125137    Point free = World.SelectNeighbor(null, Position);
          if (free.X != -1) {
```

Eine Übersicht über die Allokierungen von Element zeigt auch deutlich, dass die Point[] mit abstand am öftesten Allokiert werden.



Name	Inclusive Allocations	Exclusive Allocations	Inclusive Bytes	Exclusive Bytes	Inclusive Allocations %
System.Drawing.Point[	1.573.270	1.573.270	81.886.040	81.886.040	93,71
VSS.Wator.Original.Fisl	61.735	61.735	2.963.280	2.963.280	3,68
System.Version	6.856	6.856	219.392	219.392	0,41
VSS.Wator.Original.Sha	6.231	6.231	299.088	299.088	0,37
System.Drawing.Knowi	3.142	3.142	75.408	75.408	0,19
System.String	2.672	2.672	190.588	190.588	0,16
System.WeakReferenc	1.177	1.177	28.248	28.248	0,07
System.Int32	978	978	23.472	23.472	0,06
System.Char[]	972	972	351.464	351.464	0,06
System.Object[]	878	878	64.512	64.512	0,05
System.EventHandler	793	793	50.752	50.752	0,05
System.Single[]	743	743	35.608	35.608	0,04
System.Windows.Form	707	707	22.624	22.624	0,04

1.2. Wie kann man diese Punkte verbessern

Man kann versuchen, die Anzahl der Erzeugung von Punkten zu senken. Eventuell kann man das Punkte Array auch total eliminieren.
In dem man das Array auf eine Dimension bringt, könnte man eventuell Laufzeit sparen.
Das Randomize wird nun immer Doppelt aufgerufen. Es kann versucht werden sich einen Aufruf von Randomize einzusparen.
Eventuell könnte man bei einem Prüfen auf Nachbarn gleich direkt den Fisch oder den Hai umsetzten, und sich so das Punkte Array insgesamt einsparen

2. Verbesserungen:

2.1. Eleminieren vom häufigen anlegen von Point:

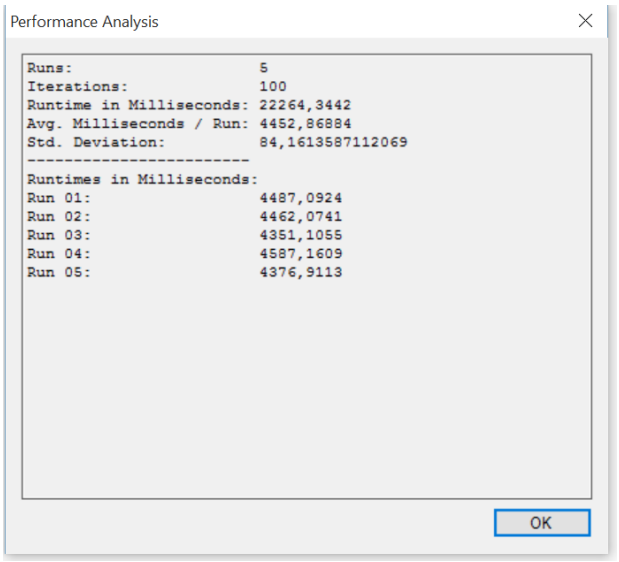
Wie in Punkt 1 beschrieben, wird sehr häufig ein Array von Punkten angelegt. Weiters wird sehr sehr oft ein Punkt angelegt. Um dies zu verbessern, wird in der Klasse ein Punkte Array angelegt, und mit Punkten der Position (-1,-1) initialisiert.

Die Methode GetNeighbors kann dann dahingehend verändert werden, dass die Punkte beim Aufruf das Array an erster Stelle zurückgesetzt wird. Von der Methode wird schließlich nach dem errechnen der Nachbarn einfach die Anzahl der Möglichen Nachbarn zurückgegeben. In der aufrufenden Methode SelectNeighbor, kann man dadurch Prüfungen sparen, und nur wenn mehrere Gefunden werden, muss noch einer zufällig ausgewählt werden.

```
public int GetNeighbors(Type type, Point position)
{
  // this.ResetPoints();
   // Point[] neighbors = new Point[4];
   int neighborIndex;
   int i, j;
   points[0].X = -1;
   points[0].Y = -1;
   // counter for the number of cells of the correct type
   neighborIndex = 0;
   // look up
   i = position.X;
   j = (position.Y + Height - 1) % Height;
   // if we look for empty cells (null) we don't have to check the type using instanceOf
   if ((type == null) && (Grid[i, j] == null))
   {
      points[neighborIndex].X = i;
      points[neighborIndex].Y = j;
      neighborIndex++;
   }
   else if ((type != null) && (type.IsInstanceOfType(Grid[i, j])))
   {
      // using instanceOf to check if the type of the animal on grid cell (i/j) is either a shark of a fish
      // animals that moved in this iteration onto the given cell are not considered
      // because the simulation runs in discrete time steps
      if ((Grid[i, j] != null) && (!Grid[i, j].Moved))
      {
         points[neighborIndex].X = i;
         points[neighborIndex].Y = j;
         neighborIndex++;
      }
   }
   // look right
   i = (position.X + 1) % Width;
   j = position.Y;
   if ((type == null) && (Grid[i, j] == null))
   {
      points[neighborIndex].X = i;
      points[neighborIndex].Y = j;
      neighborIndex++;
   }
   else if ((type != null) && (type.IsInstanceOfType(Grid[i, j])))
   {
      if ((Grid[i, j] != null) && (!Grid[i, j].Moved))
      {
         points[neighborIndex].X = i;
         points[neighborIndex].Y = j;
         neighborIndex++;
      }
   }
   // look down
   i = position.X;
   j = (position.Y + 1) % Height;
   if ((type == null) && (Grid[i, j] == null))
   {
      points[neighborIndex].X = i;
      points[neighborIndex].Y = j;
```

```
            neighborIndex++;
        }
        else if ((type != null) && (type.IsInstanceOfType(Grid[i, j])))
        {
            if ((Grid[i, j] != null) && (!Grid[i, j].Moved))
            {
                points[neighborIndex].X = i;
                points[neighborIndex].Y = j;
                neighborIndex++;
            }
        }
        // look left
        i = (position.X + Width - 1) % Width;
        j = position.Y;
        if ((type == null) && (Grid[i, j] == null))
        {
            points[neighborIndex].X = i;
            points[neighborIndex].Y = j;
            neighborIndex++;
        }
        else if ((type != null) && (type.IsInstanceOfType(Grid[i, j])))
        {
            if ((Grid[i, j] != null) && (!Grid[i, j].Moved))
            {
                points[neighborIndex].X = i;
                points[neighborIndex].Y = j;
                neighborIndex++;
            }
        }
        return neighborIndex;
    }

// select a random neighbouring cell that contains an animal (or null) of the given type
    public Point SelectNeighbor(Type type, Point position)
    {
        // first determine _all_ neighbours of the given type
        int neighbors = GetNeighbors(type, position);
        if (neighbors > 1)
        {
            // if more than one cell has been found => return a randomly selected cell
            return points[random.Next(neighbors)];
        }
        return points[0];
    }
```

Dies bringt folgenden Speed-Up:



2.2. Das Random-Matrix auf 1Dimensionales Array konvertieren:

Ein Randomize auf ein Eindimensionales Array ist wesentlich einfacher und benötigt weniger Random zahlen, als ein Randomize auf ein 2-Dimensionales Array. Es kann daher versucht werden, die Randomize-Matrix auf ein 1-Dimensionales Array zu bringen.

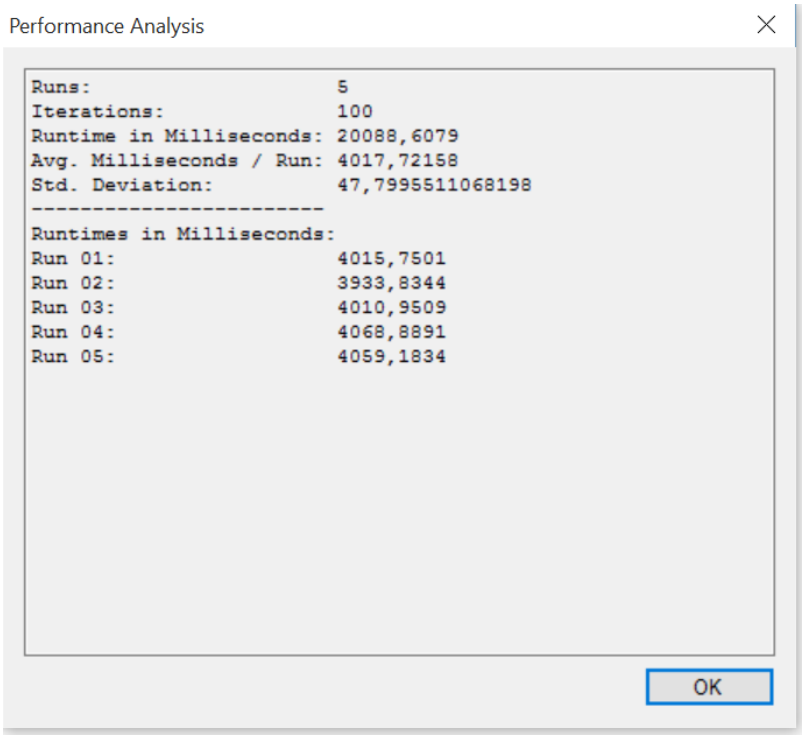
In ExecuteStep wurden dazu folgende Änderungen vorgenommen:

```
for (int i = 0; i < randomMatrix.Length; i++)
        {
            // determine row and col of the grid cell by manipulating the value
            // of the current cell in the random matrix
            col = randomMatrix[i] % Width;
            row = randomMatrix[i] / Width;

            // if there is an animal on this cell that has not been moved in this simulation step
            // then we execute it
            if (Grid[col, row] != null && !Grid[col, row].Moved)
                Grid[col, row].ExecuteStep();

        }
```

Und die folgenden Methoden wurden adaptiert:

```
private int[] GenerateRandomMatrix(int width, int height)
    {
        int[] matrix = new int[width*height];

        for (int i = 0; i < matrix.Length; i++)
        {
            matrix[i] = i;
        }
        // shuffle matrix
        RandomizeMatrix(matrix);
        return matrix;
    }

    // shuffle the values of the 2D array in a random fashion
    private void RandomizeMatrix(int[] matrix)
    {
        // perform a Knuth shuffle (http://en.wikipedia.org/wiki/Fisher%E2%80%93Yates_shuffle)
        // here we need to shuffle a 2D array instead of a simple array
        int temp, selectedRow;

        // row and col are updated in the following loop over all cells
        // begin in the top left (0/0) position of the matrix

        // for all cells of the matrix
        for (int i = 0; i <matrix.Length; i++)
        {
            // store the original value
            temp = matrix[i];
            selectedRow = random.Next(i, matrix.Length);

            matrix[i] = matrix[selectedRow];
            matrix[selectedRow] = temp;


        }
    }
```

Dies bringt folgenden Speed-Up:



2.3. Division um auf row und col zu gelangen vermeiden:

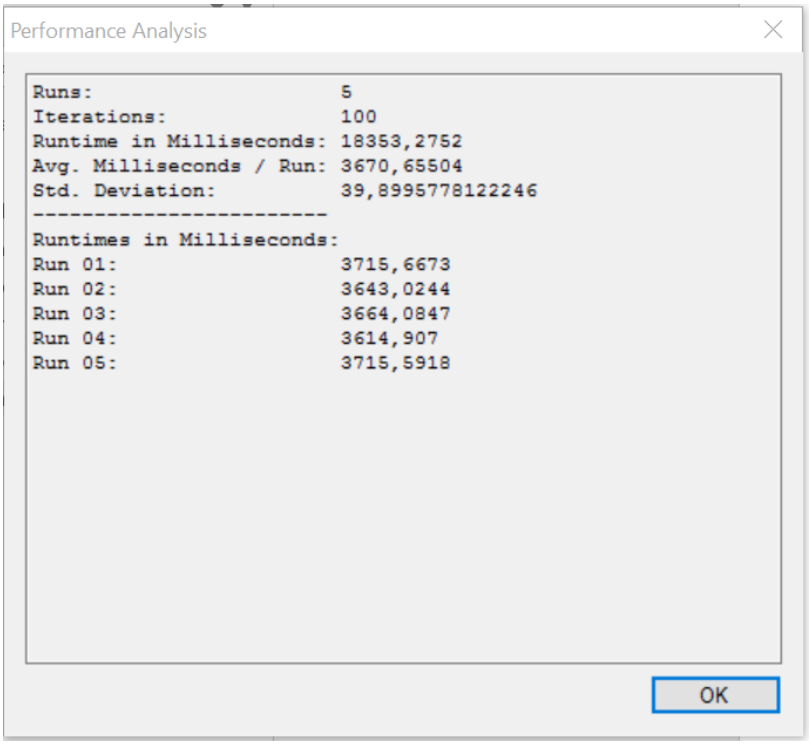
Wator.exe	16.032	13.061	100,00	81,47
VSS.Wator.Original.OriginalWatorWorld.ExecuteStep	15.619	7.646	97,42	47,69
Line 112	11.530	5.838	71,92	36,41
Line 124	1.200	1.150	7,49	7,17

Line 112 zeigt uns, dass die Division um row und col aus der Random-Matrix zu berechnen sehr viel Performance raubt. Eine Änderung der Random-Matrix, auf ein Point[] Array kann diese Division verhindern. Da beim Durchführen von Randomize nur Referenzen getauscht werden, sollte sich diese Funktion auch nicht verschlechtern.

```
 private Point[] randomMatrix;

public void ExecuteStep()
      {
         // shuffle the values in randomMatrix to make sure
         // that in each time step the order of execution of cells is different (and random)
         RandomizeMatrix(randomMatrix);

         // go over all cells of the random matrix
         // the variables row and col contain the actual position of the
         // grid cell that should be executed.

         int row, col;
         for (int i = 0; i < randomMatrix.Length; i++)
         {
               // determine row and col of the grid cell by manipulating the value
               // of the current cell in the random matrix
               col = randomMatrix[i].X;
               row = randomMatrix[i].Y;

               // if there is an animal on this cell that has not been moved in this simulation step
               // then we execute it
               if (Grid[col, row] != null && !Grid[col, row].Moved)
                  Grid[col, row].ExecuteStep();

         }

private Point[] GenerateRandomMatrix(int width, int height)
      {
         Point[] matrix = new Point[width*height];
         int row = 0;
         int col = 0;
```

```
        for (int i = 0; i < matrix.Length; i++)
        {
            matrix[i] = new Point(row, col);
            col++;
            if(col >= width)
            {
                col = 0;
                row++;
            }
        }
        // shuffle matrix
        RandomizeMatrix(matrix);
        return matrix;
    }

private void RandomizeMatrix(Point[] matrix)
    {
        // perform a Knuth shuffle (http://en.wikipedia.org/wiki/Fisher%E2%80%93Yates_shuffle)
        // here we need to shuffle a 2D array instead of a simple array
        Point temp;
        int selectedRow;

        // row and col are updated in the following loop over all cells
        // begin in the top left (0/0) position of the matrix

        // for all cells of the matrix
        for (int i = 0; i <matrix.Length; i++)
        {
            // store the original value
            temp = matrix[i];
            selectedRow = random.Next(i, matrix.Length);

            matrix[i] = matrix[selectedRow];
            matrix[selectedRow] = temp;


        }
    }
```

Dies bringt dann folgenden Speed-Up:



3. Übersicht der Performance-Optimierung:

Die folgende Tabelle zeigt die jeweiligen Durchläufe, sowie deren Speed-Up zum jeweiligen Vorgänger, sowie zum Original.

	Original	Optimierung 1	Optimierung 2	Optimierung 3
Run 1	5957,4462	4487,0924	4015,7501	3715,6673
Run 2	6106,1956	4462,0741	3933,8344	3643,0244
Run 3	6230,9134	4351,1055	4010,9509	3664,0847
Run 4	6761,4363	4587,1609	4068,8891	3614,907
Run 5	6243,5051	4376,9113	4059,1834	3715,5918
Durchschnitt	6259,899932	4452,86884	4017,72158	3670,65504
Speed-Up zum Original	0	1,4058	1,5581	1,7054
Speed-Up zum Vorgänger	0	1,4058	1,1083	1,0946