2D-Vektorgraphik

Copyright © by V. Miszalok, last update: 26-02-2010

- Vektor Displays
- Vertex
- Polygon
- Abstand eines Eckpunktes vom Vorgänger
- Länge eines offenen Polygons = polyline
- Umfang eines geschlossenen Polygons
- Fläche eines geschlossenen Polygons
- Umschreibendes Rechteck eines Polygons
- Mittelpunkt eines Polygons
- Polygon-Scroll
- Polygon-Zoom
- Polygon-Rotation
- Konzentrische Strahlen (Splash)
- Bézier Approximation
- Cubic Spline Interpolation
- Programming curves in parametric form

Vektor Displays

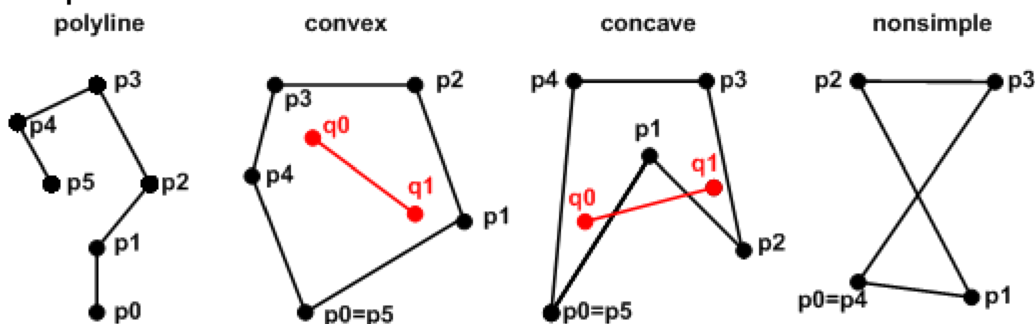
Technische Basis der Vektorgraphik sind die Vektor Displays (englisch: Random Access Cathode Ray Tubes = RA-CRTs). Dies sind Elektronenstrahlröhren mit Phosphorschirm und frei beweglichem Strahl ohne festem Strahlweg. Es gibt keine Zeilen und keine Pixel. Man lenkt den Strahl in beliebige Richtung über den Bildschirm und erhält feine, hochpräzise Linien. Diese Darstellungsweise ist ideal für technische Zeichnungen, Landkarten, Flugbewegungen etc. Siehe: Vorlesung über Katodenstrahl-Displays
(Problem: Vektor Displays können Bildern der realen Welt (Photos, TV) nicht darstellen; das können nur Rasterdisplays.)

Vertex

Vertex (lateinisch Ecke, Plural Vertices, gesprochen: Wertizees) = Vektor = Punkt = Eckpunkt = Point ist der atomare Baustein der Vektorgraphik.
Im einfachsten Fall besteht er aus zwei Koordinaten mit den beiden Abständen vom linken und vom oberen Fensterrand.
Man unterscheidet:
1) 2D-Vertices mit 2 Floatkoordinaten: `PointF{ float x; float y; }`
2) 2D-Vertices mit 2 Integerkoordinaten: `Point { int x; int y; }`
3) 3D-Vertices mit 3 Floatkoordinaten: `Vector3{ float x; float y; float z; }`
4) 3D-Vertices wie `Vector3` plus Farbe, Normale, Texturkoordinate etc. (siehe: DirectX 3D-Vertex Formate)
Für 2D-Graphik verwendet man in der Regel Typ 1), weil dieser Datentyp stufenlosen Zoom und Rotation zulässt und frei von Rundungsfehlern ist.
Einfache Malprogramme mit der Maus verwenden oft Datentyp 2), weil die Maus nur Integerkoordinaten liefert und die Malbefehle wie z.B. `DrawLine()` Integerkoordinaten verlangen.

Polygon

= wichtigster Datentyp der Vektorgraphik ist eine geordnete Menge von Vertices `p[0], p[1], ... p[i], ... p[n-1]`, wobei die Vertices `p[i]` durch Strecken `DrawLine( p[i], p[i+1] )` verbunden sind. Diese Strecken dürfen sich nicht überkreuzen.
Man unterscheidet:
a) **offenes Polygon (engl.: polyline)**: Startpunkt und Endpunkt sind nicht identisch. Offene Polygone haben eine Länge = length, aber keinen Umfang und keine Fläche.
b) **geschlossenes Polygon**: Startpunkt identisch mit Endpunkt. Das hat zur Folge, dass ein geschlossenes `n`-Eck durch `n+1` Vertices kodiert werden muss. Ein Dreieck hat also 4 Vertices P0, P1, P2, und P3==P0 !
Geschlossene Polygone haben einen Umfang = perimeter und eine Fläche = area.
Umwandlung a) → b): Man kann jedes offene Polygon schließen, indem man den nullten Vertex `p[0]` an das Ende des Arrays kopiert.
Spricht man nur von Polygon ohne Adjektiv offen oder geschlossen, dann ist ein geschlossenes Polygon gemeint, auch wenn der letzte schließende Vertex fehlen sollte.
c) **konvexes Polygon**: Für alle beliebig gewählten zwei Punkte `q0` und `q1` aus dem Inneren des Polygons gilt: Die Verbindungsstrecken liegen immer vollständig im Polygon.
d) **konkaves Polygon**: Es gibt Punkte `q0` und `q1` aus dem Inneren des Polygons, deren Verbindungsstrecke nicht vollständig im Polygon liegt.
e) **überschlagenes Polygon** (engl.: nonsimple polygon): Polygonkanten überschneiden sich. Solche Polygone können keine Flächen beranden und sind ungeeignet für Computergraphik.
Beispiele:



Polygone werden in Form von Arrays programmiert und gespeichert.
Es gibt zwei Basistypen von 2D-Polygon-Arrays:
a) **Polygon als Array fester Länge**: `const Int32 n = 100; PointF[] p = new PointF[n];`
Vorteil: einfache und schnelle Zugriffe.
Nachteil: Man muss den Speicherplatz fest reservieren, d.h. man muss `n` kennen.
b) **Polygon als dynamisches Array**: `ArrayList p = new ArrayList(); p.Add( p0 );`
Vorteil: Man kann jederzeit Vertices an- oder einfügen oder löschen.
Nachteile: 1) Namespace `using System.Collections;` notwendig.
2) Zugriffe langsamer als bei festem Array und kein Zugriff über Pointer möglich, weil die Vertices nicht speicherkompakt, sondern als verkettete Liste gespeichert sind.
3) Typecasting = explizite Typspezifizierung (obwohl keine Typkonvertierung stattfindet) beim lesen aus dem Array notwendig: `p0 = (PointF)p[i];`, weil eine `ArrayList` alle möglichen Objekte in bunter Reihenfolge enthalten darf.
Tip für Malprogramme: Speichern Sie doppelt: Sammeln sie die Vertices zunächst in einem dynamischen Array und kopieren Sie dieses später in ein festes Array.

Vier Beispiele für Speicherung und Zugriff auf Polygone
Folgende 6 Deklarationen gelten für alle 4 Beispiele:

```
using System.Collections; //enthält ArrayList
const Int32 n = 100; //Länge des festen Arrays
Random r = new Random(); //Zufallszahlengenerator zum füllen der Arrays
Int32 i;
Pen mypen = new Pen( Color.Red, 1 );
Graphics g = this.CreateGraphics();
```

1) Beispiel: Point-Array fester Länge mit Integer-Koordinaten

```
Point[] p = new Point[n];
for ( i=0; i < n; i++ ) //write into array
{ p[i].X = r.Next(100); p[i].Y = r.Next(100); }
for ( i=0; i < n-1; i++ ) //read from array
  g.DrawLine( mypen, p[i], p[i+1] );
```

2) Beispiel: Point-dynamisches Array variabler Länge mit Integer-Koordinaten

```
ArrayList p = new ArrayList();
for ( i=0; i < n; i++ ) //write into array
  p.Add( new Point( r.Next(100), r.Next(100) ) );
for ( i=0; i < p.Count-1; i++ ) //read from array
  g.DrawLine( mypen, (Point)p[i], (Point)p[i+1] );
```

3) Beispiel: PointF-Array fester Länge mit Float-Koordinaten

```
PointF[] p = new PointF[n];
for ( i=0; i < n; i++ ) //write into array)
{ p[i].X = 100f*(Single)r.NextDouble(); p[i].Y = 100f*(Single)r.NextDouble(); }
for ( i=0; i < n-1; i++ ) //read from array
{ Int32 x0 = Convert.ToInt32( p[i  ].X );
  Int32 y0 = Convert.ToInt32( p[i  ].Y );
  Int32 x1 = Convert.ToInt32( p[i+1].X );
  Int32 y1 = Convert.ToInt32( p[i+1].Y );
  g.DrawLine( mypen, x0, y0, x1, y1 );
}
```

4) Beispiel: PointF-dynamisches Array variabler Länge mit Float-Koordinaten

```
ArrayList p = new ArrayList();
for ( i=0; i < n; i++ ) //write into array
  p.Add( new PointF( 100f*(Single)r.NextDouble(), 100f*(Single)r.NextDouble() ) );
for ( i=0; i < p.Count-1; i++ ) //read from array
{ Int32 x0 = Convert.ToInt32( ((PointF)p[i  ]).X );
  Int32 y0 = Convert.ToInt32( ((PointF)p[i  ]).Y );
  Int32 x1 = Convert.ToInt32( ((PointF)p[i+1]).X );
  Int32 y1 = Convert.ToInt32( ((PointF)p[i+1]).Y );
  g.DrawLine( mypen, x0, y0, x1, y1 );
}
```

Abstand eines Eckpunktes vom Vorgänger

Bei Malprogrammen, wo die Eckpunkte eines Polygons durch das Ereignis `OnMouseMove(...)` geliefert werden, sind die Vertices nicht äquidistant, sondern die Punktabstände sind abhängig von der Mausgeschwindigkeit, der Geschwindigkeit des Rechners und von dessen momentaner Auslastung. Bewegt der User die Maus langsam, dann liegen die Vertices sehr dicht beieinander (oft Pixel neben Pixel), wenn er schnelle Bewegungen macht, liegen sie weit auseinander. Das erste Extrem (Vertices zu nah) läßt sich leicht folgendermaßen bekämpfen: Innerhalb des Eventhandlers `OnMouseMove(...)` berechnet man den Abstand des aktuellen Punktes zum Vorgänger mit dem Satz des Pythagoras im rechtwinkligen Dreieck: Quadrat über der Hypothenuse = Summe der Quadrate über beiden Katheten. Ist der Abstand zu klein, Punkt ignorieren. Beispiel mit erzwungenem Minimalabstand von 10 Pixeln:

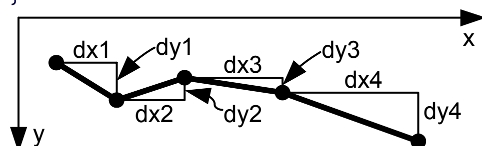
```
Point p0 = new Point(), p1 = new Point();
protected override void OnMouseDown( MouseEventArgs e )
{ p0.X = e.X; //first vertex
  p0.Y = e.Y; //first vertex
}
protected override void OnMouseMove( MouseEventArgs e )
{ if ( e.Button == MouseButtons.None ) return; //if no button pressed do nothing
  p1.X = e.X; Int32 dx = p1.X - p0.X; //horizontal distance
  p1.Y = e.Y; Int32 dy = p1.Y - p0.Y; //vertical   distance
  if ( Math.Sqrt( dx*dx + dy*dy ) < 10 ) return; //if distance < 10 do nothing
  g.DrawLine( blackpen, p0, p1 );
  p0 = p1;
}
```

Man kann den `OnMouseMove( ... )`-EventHandler erheblich beschleunigen, wenn man die Ungleichung quadriert und damit die Wurzel überflüssig macht. Dazu ersetzt man:
`if ( Math.Sqrt( dx*dx + dy*dy ) < 10 ) return;` durch
`if ( dx*dx + dy*dy < 100 ) return;`.
Sie finden Anwendungen unter:
www.miszalok.de/C_2DCis/C2_Draw/C2DCisDraw_d.htm und unter
www.miszalok.de/C_2DCis/C3_XML/C2DCisXML_d.htm und unter
www.miszalok.de/C_2DCis/C4_Anim/C2DCisAnim_d.htm

Länge eines offenen Polygons = polyline

Gegeben sei ein dynamischer Vertex-Array: `ArrayList p = new ArrayList();` gefüllt mit Objekten vom Typ `Point`
Gesucht sei `Double laenge;`
Man benutzt für jede Teilstrecke den Satz des Pythagoras: Im rechtwinkligen Dreieck ist die Hypotenuse gleich der Wurzel aus der Summe der Kathetenquadrate. Die Summe aller Hypotenusen summiert sich zur gesamten Länge.

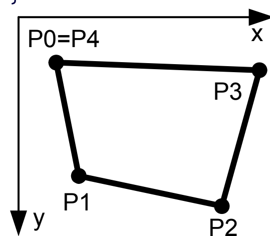
```
Double laenge = 0.0;
Point p0 = (Point)p[0];
for ( Int16 i=1; i < p.Count; p++ ) //Schleife fängt nicht bei 0 an, sondern bei 1
{ Point p1 = (Point)p[i];
  Double dx = p1.X - p0.X; //horizontale Kathete
  Double dy = p1.Y - p0.Y; //vertikale Kathete
  laenge += Math.Sqrt( dx*dx + dy*dy ); //Hypotenuse
  p0 = p1;
}
```



Umfang eines geschlossenen Polygons

Gegeben sei ein dynamischer Vertex-Array: `ArrayList p = new ArrayList();` gefüllt mit Objekten vom Typ `Point`
Gesucht sei `Double perimeter;`
Zunächst kopiert man den Startpunkt des Polygons an dessen Ende, falls das nicht schon von vornherein der Fall ist.

```
Double perimeter = 0.0;
Point p0 = (Point)p[0];
if ( p0 != (Point)p[p.Count-1] ) p.Add( p0 ); //Polygon schließen
for ( Int16 i=1; i < p.Count; p++ ) //Schleife fängt nicht bei 0 an, sondern bei 1
{ Point p1 = (Point)p[i];
  Double dx = p1.X - p0.X; //horizontale Kathete
  Double dy = p1.Y - p0.Y; //vertikale Kathete
  perimeter += Math.Sqrt( dx*dx + dy*dy ); //Hypotenuse
  p0 = p1;
}
```



Sie finden eine Anwendung unter: www.miszalok.de/C_2DCis/C2_Draw/C2DCisDraw_d.htm

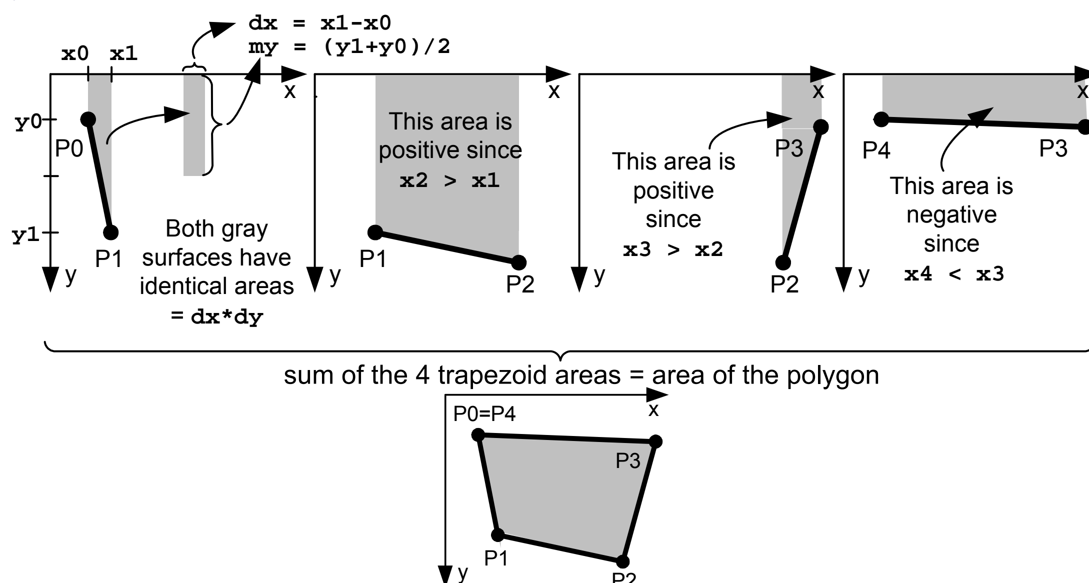
Fläche eines geschlossenen Polygons

Gegeben sei ein dynamischer Vertex-Array: `ArrayList p = new ArrayList();` gefüllt mit Objekten vom Typ `Point`

Gesucht sei `Double area;`

Zunächst kopiert man den Startpunkt des Polygons an dessen Ende, falls das nicht schon von vornherein der Fall ist.

```
Double area = 0.0;
Point p0 = (Point)p[0];
if ( p0 != (Point)p[p.Count-1] ) p.Add( p0 ); //Polygon schließen
for ( Int16 i=1; i < p.Count; p++ ) //Schleife fängt nicht bei 0 an, sondern bei 1
{ Point p1 = (Point)p[i];
  Double dx = p1.X - p0.X; //Breite des Trapezes
  Double my = (p1.Y + p0.Y) / 2.0; //mittlerer y-Wert
  area += dx * my; //Fläche des Trapezes
  p0 = p1;
}
```



Die Fläche kann einen negativen Wert ergeben. Ihr Vorzeichen hängt von der Umlaufrichtung ab. Wenn die Umlaufrichtung so ist, dass die Fläche links von der Begrenzungslinie liegt, dann wird die Fläche positiv, andernfalls negativ. Will man ein Ergebnis, das von der Umlaufrichtung unabhängig und positiv ist, so muss man hinter die `for`-Schleife schreiben:

```
area = Math.Abs( area );
```

Sie finden eine Anwendung unter: www.miszalok.de/C_2DCis/C2_Draw/C2DCisDraw_d.htm

Umschreibendes Rechteck eines Polygons

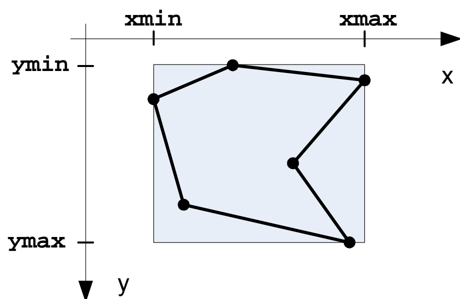
= englisch: Bounding Box ist das kleinstmögliche achsparallele Gefängnis eines Polygons. Die Bounding Box ersetzt meistens das Polygon bei der Frage, ob die Maus auf das Polygon zeigt oder bei der Frage, ob zwei Polygone kollidieren.

Gegeben sei ein dynamischer Vertex-Array: `ArrayList p = new ArrayList();` gefüllt mit Objekten vom Typ `Point`

Gesucht sei `Rectangle box; //bounding box;`

Zunächst setzt man die 4 Mauern des Gefängnisses `xmin, ymin, xmax, ymax` auf den Startpunkt des Polygons.

```
Int32 xmin, ymin, xmax, ymax;
xmin = xmax = ( (Point)p[0] ).X;
ymin = ymax = ( (Point)p[0] ).Y;
for ( int i=1; i < p.Count; i++ )
{ Point p0 = (Point)p[i]          //nächste Polygonecke
  if ( p0.X < xmin ) xmin = p0.X; //verschiebe die linke Mauer nach links
  if ( p0.X > xmax ) xmax = p0.X; //verschiebe die rechte Mauer nach rechts
  if ( p0.Y < ymin ) ymin = p0.Y; //verschiebe die obere Mauer nach oben
  if ( p0.Y > ymax ) ymax = p0.Y; //verschiebe die untere Mauer nach unten
}
box = new Rectangle( xmin, ymin, xmax-xmin, ymax-ymin );
```



Anmerkung:
In der Klasse `Rectangle` existieren die Properties `X` und `Y` auch unter den Namen `Left` und `Top`.
Im Beispiel:
`xmin = box.X = box.Left` und
`ymin = box.Y = box.Top`.

Sie finden Anwendungen unter:
www.miszalok.de/C_2DCis/C2_Draw/C2DCisDraw_d.htm und unter
www.miszalok.de/C_2DCis/C4_Anim/C2DCisAnim_d.htm

Die Feststellung von **Kollisionen** ist schwierig, wenn die Figuren komplizierte Außenformen haben. Einfach ist hingegen die Feststellung der Kollision der umschreibenden Rechtecke. Beispiel:
Gegeben sei ein bewegliches Rechteck `Rectangle box` und ein Array von stationären Rechtecken `Rectangle[] boxes = new Rectangle[n];`.
Frage: Ist `box` irgendwo angestoßen ?

```
Lösung zu Fuß:
for ( int i=0; i<n; i++ )
{ if ( box.X > boxes[i].X + boxes[i].Width  ) continue; //box liegt zu weit rechts
  if ( box.Y > boxes[i].Y + boxes[i].Height ) continue; //box liegt zu weit unten
  if ( boxes[i].X > box.X + box.Width       ) continue; //box liegt zu weit links
  if ( boxes[i].Y > box.Y + box.Height      ) continue; //box liegt zu weit oben
  Debug.WriteLine("box ist an " + i.ToString() + " gestoßen.\r\n" );
}
Lösung mit der Rectangle.IntersectsWith - Methode:
for ( int i=0; i<n; i++ )
  if ( box.IntersectsWith( boxes[i] ) )
    Debug.WriteLine("box ist an " + i.ToString() + " gestoßen.\r\n" );
```

Man ersetzt in der Regel auch bei Mausabfragen die Graphikobjekte durch ihre umschreibenden Rechtecke.
Beispiel: Liegt der Mauszeiger `e.X, e.Y` auf einem der Graphikobjekte `i` mit dem umschreibenden Rechteck `boxes[i]` ?

```
Lösung mit der Rectangle.Contains - Methode:
for ( int i=0; i<n; i++ )
  if ( boxes[i].Contains( e.X, e.Y ) )
    Debug.WriteLine("Die Maus zeigt auf " + i.ToString() + ".\r\n" );
```

Mittelpunkt eines Polygons

Man verwendet zweckmäßig Gleitkommakoordinaten (Datentyp `PointF`) und keine Integerkoordinaten (Datentyp `Point`), weil die Mittelpunktsberechnung immer eine Division erfordert und deshalb selten ganze Zahlen liefert.
Es gibt vier verschiedene Mittelpunkte `mp` eines Polygons `p`, die ziemlich weit auseinander liegen können.
`1 )` **Mittelpunkt des umschreibenden Rechtecks**
`2a)` **Mittelpunkt der Ecken**
`2b)` **Mittelpunkt der gewichteten Seiten**
`2c)` **Schwerpunkt einer homogen dichten Fläche**
Details siehe English Version: Center of a polygon
Sie finden Anwendungen unter:
www.miszalok.de/C_2DCis/C2_Draw/C2DCisDraw_d.htm und unter
www.miszalok.de/C_2DCis/C4_Anim/C2DCisAnim_d.htm

2D Polygon Scroll

Gegeben sei:
1) eine Polygonlänge: `const Int32 n = 100;`
2) `PointF palt = new PointF[n];`
3) `PointF pneu = new PointF[n]; //Ergebnis`
Gesucht sei:
2D-Scroll = 2D-Translate = 2D-Verschiebung um die Beträge `Single dx, dy`.

```
for ( i=0; i < palt.Count; i++ ))
{ pneu[i].X = palt[i].X + dx;
  pneu[i].Y = palt[i].Y + dy;
}
```

Es geht auch auf einem einzigen Polygon:

```
for ( i=0; i < p.Count; i++
{ palt[i].X += dx;
  palt[i].Y += dy;
}
```

2D Polygon Zoom

Gesucht sei: 2D-Zoom = 2D-Scaling = 2D-Vergrößerung/Verkleinerung um die Beträge `Single zoomx, zoomy`.
Die Polygone `palt` und `pneu` müssen unbedingt float-Koordinaten besitzen (Datentyp: `PointF`).

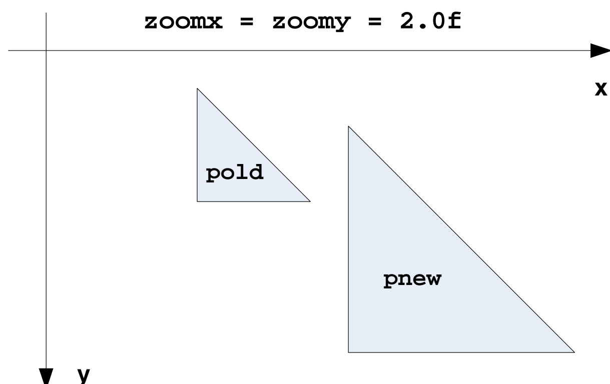
```
for ( i=0; i < palt.Count; i++ ))
{ pneu[i].X = palt[i].X * zoomx;
  pneu[i].Y = palt[i].Y * zoomy;
}
```

Es geht auch auf einem einzigen Polygon:

```
for ( i=0; i < p.Count; i
{ palt[i].X *= zoomx;
  palt[i].Y *= zoomy;
}
```

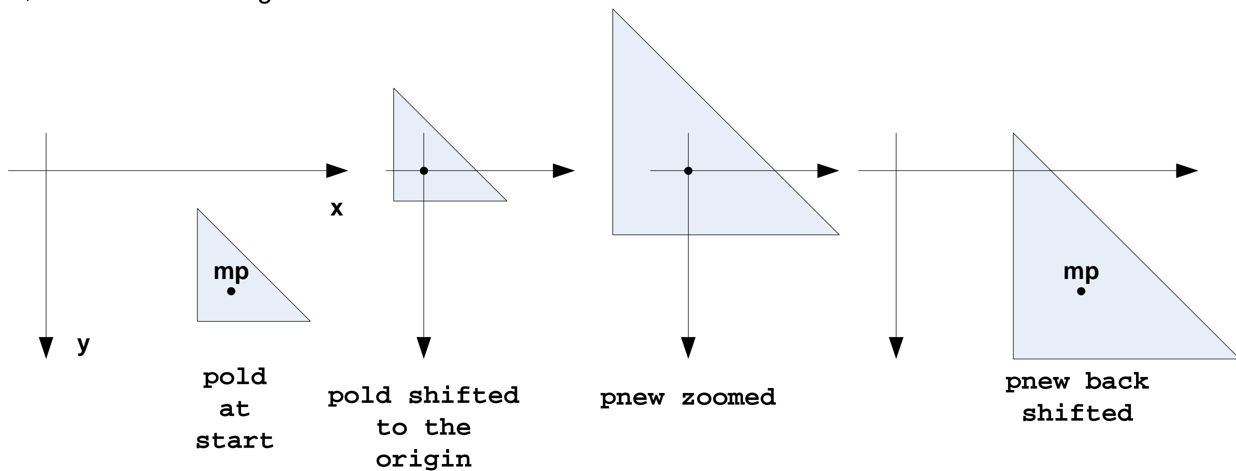
`zoomx < 1.0f` = Verkleinerung und Verschiebung nach links
`zoomy < 1.0f` = Verkleinerung und Verschiebung nach oben
`zoomx > 1.0f` = Vergrößerung und Verschiebung nach rechts
`zoomy > 1.0f` = Vergrößerung und Verschiebung nach unten

Beispiel:



Meistens sind die Verschiebungen unerwünscht. Man will ein Polygon "an Ort und Stelle" zoomen.
Man benötigt dazu einen Mittelpunkt `mp`:

1) Verschiebung des Polygons so, dass `mp` auf den Nullpunkt zu liegen kommt.
2) Zoom auf dem Nullpunkt
3) Rückverschiebung




```
for ( i=0; i < palt.Count; i++ ))
{ pneu[i].X = (palt[i].X - mp.X) * zoomx + mp.X;
  pneu[i].Y = (palt[i].Y - mp.Y) * zoomy + mp.Y;
}
```

Es geht auch auf einem einzigen Polygon:

```
for ( i=0; i < p.Count; i
{ palt[i].X -= mp.X;
  palt[i].Y -= mp.Y;
  palt[i].X *= zoomx;
  palt[i].Y *= zoomy;
  palt[i].X += mp.X;
  palt[i].Y += mp.Y;
}
```

Sie finden eine Anwendung unter: **www.miszalok.de/C_2DCis/C4_Anim/C2DCisAnim_d.htm**

2D Polygon Rotation

Gesucht sei: 2D-Rotation = 2D-Drehung im Uhrzeigersinn um einen Winkel `Single alpha` oder `Double alpha`.
Drehachse der 2D-Rotation ist die unsichtbare Z-Achse, die den Bildschirm in der linken oberen Ecke der ClientArea senkrecht durchstößt.
Die Polygone `palt` und `pneu` müssen unbedingt Gleitkomma-Koordinaten besitzen (Datentyp: `PointF`).

```
Double arcus = alpha * 2.0 * Math.PI / 360.0; //alpha in Bogenmaß
Single cosinus = (Single)Math.Cos( arcus );   //cosinus(alpha) als float
Single sinus   = (Single)Math.Sin( arcus );   //  sinus(alpha) als float
for ( i=0; i < palt.Count; i++ ))
{ pneu[i].X = palt[i].X * cosinus - palt[i].Y * sinus;
  pneu[i].Y = palt[i].X * sinus + palt[i].Y * cosinus;
}
```

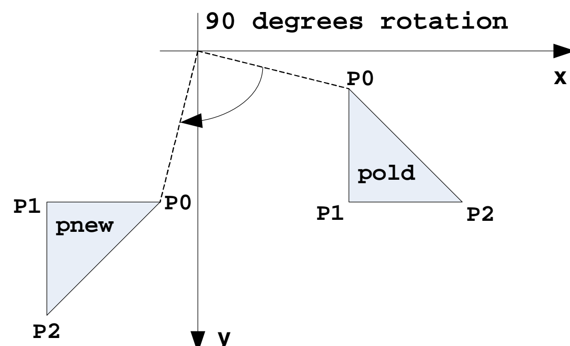
Es geht auch auf einem einzigen Polygon, allerdings mit einer Hilfsvariablen, weil man `palt[i].X` zweimal im Original braucht:

```
for ( i=0; i < p.Count; i+
{ Single help = palt[i].X * cosinus - palt[i].Y * sinus;
  palt[i].Y   = palt[i].X * sinus + palt[i].Y * cosinus;
  palt[i].X   = help;
}
```

Will man gegen den Uhrzeigersinn drehen, dann vertauscht man die Vorzeichen vor den beiden `sinus`-Produkten:

```
for ( i=0; i < palt.Count; i++ )
{ pneu[i].X =  palt[i].X * cosinus + palt[i].Y * sinus;
  pneu[i].Y = -palt[i].X * sinus + palt[i].Y * cosinus;
}
```

Beispiel 90 Grad: pneu verschwindet durch die Drehung nach links aus dem Koordinatensystem.



Vorsicht: Wie beim 2D-Zoom ist auch mit jeder Rotation eine Verschiebung verbunden, die fast immer unerwünscht ist.
Will man an "Ort und Stelle" drehen, geht man vor wie beim Zoom:

1) Verschiebung des Polygons so, dass `mp` auf den Nullpunkt zu liegen kommt.
2) Rotation auf dem Nullpunkt
3) Rückverschiebung

```
for ( i=0; i < palt.Count; i++ ))
{ Single x = palt[i].X - mp.X;
  Single y = palt[i].Y - mp.Y;
  pneu[i].X = x * cosinus - y * sinus + mp.X;
  pneu[i].Y = x * sinus + y * cosinus + mp.Y;
}
```

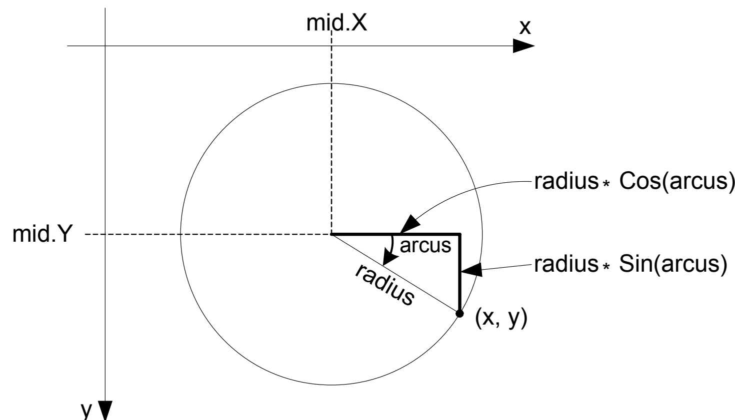
Sie finden eine Anwendung unter: www.miszalok.de/C_2DCis/C4_Anim/C2DCisAnim_d.htm

Konzentrische Strahlen (Splash)

= Strahlenbündel, wo alle Strahlenanfänge von einem Mittelpunkt ausgehen und alle Strahlenendpunkte gleichmäßig verteilt auf einem Kreis liegen.
Gegeben sei:
1) ein Mittelpunkt: `Point mid = new Point();`
2) ein Radius: `Double radius = 100;`
3) die Anzahl der Strahlen des Sterns: `Int16 const nn = 120;`
4) Farbe und Dicke der Strahlen: `Pen mypen = new Pen( Color.Red, 5 );`
5) Figur mit der der Strahl enden soll: abgeschnitten, abgerundet, Pfeil etc:
`mypen.EndCap = System.Drawing.Drawing2D.LineCap.DiamondAnchor;`

Wenn man den Winkel `arcus` eines Strahls (in Bogenmaß) kennt, erhält man seinen Endpunkt `x,y` mit Hilfe der so genannten Parameterdarstellung der Kreisgleichung:

```
x = mid.X + radius * Math.Cos( arcus );
y = mid.Y + radius * Math.Sin( arcus );
```



Der Winkel in Grad zwischen je zwei benachbarten Strahlen beträgt `360.0 / nn`, der gleiche Winkel in Bogenmaß beträgt:

```
Double arcus_1 = 2.0 * Math.PI / nn;
```

Wir benötigen Speicherplatz für `nn` Endpunkte:

```
Point[] stern = new Point[nn];
```

und füllen diesen Array mit folgender Schleife:

```
for ( Int16 i=0; i < nn; i++ )
{ Double arcus_i = arcus_1 * i;
  Double x = radius * Math.Cos( arcus_i );
  Double y = radius * Math.Sin( arcus_i );
  stern[i].X = mid.X + ConvertToInt32( x );
  stern[i].Y = mid.Y + ConvertToInt32( y );
  g.DrawLine( mypen, mid.X, mid.Y, stern[i].X, stern[i].Y );
}
```

Sie finden eine Anwendung unter: **www.miszalok.de/C_2DCis/C1_Intro/C2DCisIntro_d.htm**

Bézier Approximation

siehe: **English Version: Bézier Approximation**

Cubic Spline Interpolation

siehe: **English Version: Cubic Spline Interpolation**

Programming curves in parametric form

siehe: **English Version: Programming curves in parametric form**