3D-Vektorgraphik: Vertices

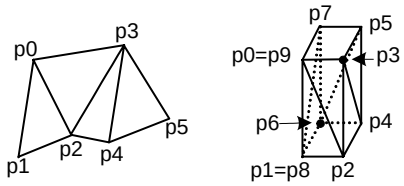
Flächen durch Dreiecke zusammensetzen

Anders als bei 2D-Vektorgraphik ist die wichtigste Datenstruktur nicht mehr das Vieleck = Polygon sondern das "Face", welches in der Regel ein Dreieck ist, aber auch ein Viereck oder ein Vieleck sein kann.
Aus Dreiecken kann man einen Streifen bilden, der äußerlich exakt wie ein Polygon = Punktarray aussieht. Das folgende Array sieht aus wie ein Polygon, ist jedoch ein Dreicksstreifen = `TriangleStrip`:

```
Vector3[] p = { { p[ 0 ].X, p[ 0 ].Y, p[ 0 ].Z },   // = p[0]
                { p[ 1 ].X, p[ 1 ].Y, p[ 1 ].Z },   // = p[1]
                { ............................ },
                { p[ i ].X, p[ i ].Y, p[ i ].Z },   // = p[i]
                { ............................ },
                { p[n-1].X, p[n-1].Y, p[n-1].Z }    // = p[n-1]
              };
```

`p[0], p[1], p[2]` bilden das erste Dreieck, an dessen Kante `p[2]p[0]` das zweite Dreieck hängt, das sich von `p[2], p[0]` zum nächsten Punkt `p[3]` aufspannt und an dessen Kante `p[2]p[3]` das dritte Dreieck hängt usw.
Insgesamt ergibt sich ein kontinuierliches Band von Dreiecken, das offen endet oder geschlossen sein kann, wenn der zweitletzte Punkt `p[n-2]` identisch mit `p[1]` ist und der letzte `p[n-1]` identisch mit `p[0]` ist.

Beispiele für einen offenen `n=6` und einen geschlossenen `n=10` `TriangleStrip`



`TriangleStrip` ist die ideale Datenstruktur für Flächen (z.B. Zylinder), die sich mit einem Streifen umspannen lassen. Besser geeignet für Würfel, Kugeln und und alle Flächen, die mehr als einen Streifen brauchen, ist `TriangleList` und `Mesh` siehe weiter unten.

Basisklasse für 3D-Koordinaten ist `Vector3`. Beispiele:
`Vector3 v = new Vector3( 1, 0,  0 );` //Punkt auf der x-Achse im Abstand `1.0f` rechts vom Nullpunkt.
`Vector3 v = new Vector3( 0, 0, -5 );` //Punkt auf der z-Achse im Abstand `5.0f` vor dem Display.
(Der Konstruktor `Vector3` wandelt automatisch `int`-Parameter in `float` um.)
Basisdatenstruktur für einen `TriangleStrip` ist folglich ein `Vector3`-Array.
Beispiel: `const int n = 100; Vector3[] trianglestrip = new Vector3[n];`

Vertexformate

Die Basisstruktur für 3D-Koordinaten `Vector3` enthält keinerlei Information, ob die Vertices eine Farbe besitzen oder eine Normale, oder ob an ihnen eine Textur befestigt werden soll.
Deshalb gibt es in DirectX elf erweiterte Datentypen für Vertices. Beispiele:
`CustomVertex.PositionOnly` = nur `X,Y,Z` ohne Erweiterung, gleichwertig mit `Vector3`
`CustomVertex.PositionColored` = `X,Y,Z` plus `Argb`-Wert (verwendet in Course 3DCis, Chapter C1)
`CustomVertex.PositionNormal` = `X,Y,Z` plus Normale `Nx,Ny,Nz` (verwendet in Course 3DCis, Chapter C2)
`CustomVertex.PositionNormalTextured` = `X,Y,Z,Nx,Ny,Nz` plus Texturpunkt `Tu,Tv` (verwendet in Course 3DCis, Chapter C3)
`CustomVertex.Transformed = X,Y,Z,W`, wobei `X,Y` = 2D-`ClientArea`-Koordinaten z.B. von Maus-Events und `Z=0f,W=1f` = konstant = unwichtig.
Vollständige Liste unter: DirectX Vertex Formate

Vertex Buffer

1) Privater Vertex Array:
Polygone sind Arrays von Vertices, die alle ein gemeinsames Vertex Format haben, z.B. `CustomVertex.PositionColored`.
Beispiel farbiges Dreieck `v`:

```
CustomVertex.PositionColored[] v = new CustomVertex.PositionColored[3];
v[0].X=-1f; v[0].Y=-1f; v[0].Z=0f;
v[1].X= 1f; v[1].Y=-1f; v[1].Z=0f;
v[2].X= 0f; v[2].Y= 1f; v[2].Z=0f;
v[0].Color = System.Drawing.Color.DarkGoldenrod.ToArgb();
v[1].Color = System.Drawing.Color.MediumOrchid.ToArgb();
v[2].Color = System.Drawing.Color.Cornsilk.ToArgb();
```

Man kann diesen Array ohne jeden Umweg (normalerweise animiert innerhalb einer `OnTimer`-Funktion) so rendern:

```
device.BeginScene();
  device.DrawUserPrimitives( PrimitiveType.TriangleList, 1, v );
device.EndScene();
```

Vorteil: einfaches Programm ohne sichtbaren `VertexBuffer`.
Nachteil: langsam, weil das Polygon = Dreieck bei jedem Rendern implizit in `VertexBuffer`-Format umgeformt und via Bus in die Graphikkarte kopiert wird.

2) Direct3D VertexBuffer anlegen:
Schneller als privater Vertex Array: Polygon nur einmal umformen in `VertexBuffer`-Format und in die Graphikkarte kopieren.
Jede Plattform, jede Sprache und jeder Programmierer speichert Vertex Arrays anders. Direct3D vereinheitlicht diese Vielfalt durch einen eigenen Standard namens `VertexBuffer`, auf dessen Format sich alle DirectX-Treiber verlassen können.
Die Direct3D-Klasse `VertexBuffer` hat folgenden Konstruktor:

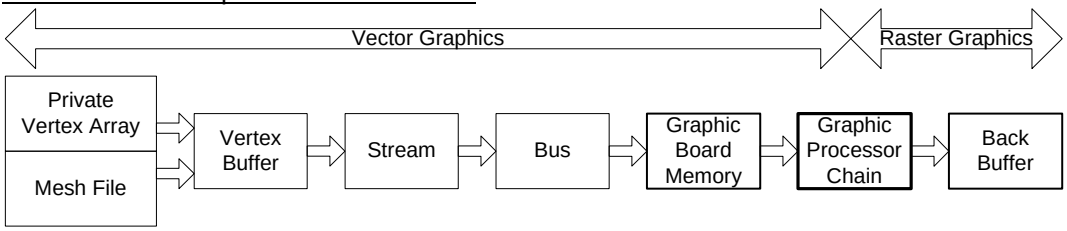
```
VertexBuffer (
  System.Type typeVertexType,
  System.Int32 numVerts,
  Microsoft.DirectX.Direct3D.Device device,
  Microsoft.DirectX.Direct3D.Usage usage,
  Microsoft.DirectX.Direct3D.VertexFormats vertexFormat,
  Microsoft.DirectX.Direct3D.Pool pool
)
```

Parameter 1: `typeVertexType`: VertexFormat-Datentyp, wie 5. Parameter, also eigentlich unsinnig.
Parameter 2: `numVerts`: Anzahl Vertices im Array.
Parameter 3: `device`: aktuelle Device.
Parameter 4: `usage`: Array von Bits = `Flags` mit Steuerinformation. Die einzelnen Bits tragen Bezeichnungen, die mit dem Operator logisch oder = `|` verknüpft werden. Beispiele: pauschal auf `0` oder `Default` setzen oder auf `WriteOnly | SoftwareProcessing`.
Parameter 5: `vertexFormat`: Datentyp des Vertex: z.B.: `CustomVertex.PositionOnly` oder `CustomVertex.PositionColored`.
Parameter 6: `pool`: Array von Bits = `Flags`, für den Speicherort des `VertexBuffer`. Beispiel: `Default` = Graphikkartenspeicher oder `Managed` = Hauptspeicher.

Vorsicht: Ein `VertexBuffer` hakt sich derart an sein `Device`, dass er vom Garbage Collector nicht getötet werden kann, auch dann nicht, wenn jede Referenz auf ihn oder auf seine `Device` verloren gegangen ist.
Konsequenz: Man muss seinen Tod explizit programmieren: `VertexBuffer.Dispose();`, ansonsten hat man ein Speicherleck.
siehe: **VertexBuffer Class** und **VertexBuffer Constructor**

3) VertexBuffer füllen:
`VertexBuffer.SetData( v, 0, LockFlags.None )` transformiert und kopiert den privaten Vertex Array `v` in einen `VertexBuffer`. Der 2. Parameter = Startoffset des Kopiervorgangs und der 3. = Schutz des VertexBuffer-Inhalts gegen konkurrierenden Zugriff sind unwichtig.
`SetData` füllt den `VertexBuffer` mit einem für DirectX-Graphikkartentreiber und `Device` lesbaren Format.
Es kann beliebig viele `VertexBuffers` geben. Ihr Platz ist im Hauptspeicher und die Graphikkarte weiß zunächst nichts von deren Existenz.

4) VertexBuffer(s) an Device übergeben:
Um in die Graphikkarte zu kommen, muss sich der `VertexBuffer` melden beim Treiber der Graphikkarte als eine auf dem Bus transportable binäre Datei = `Stream`.
Beispiel: `Device.SetStreamSource( 0, vb, 0 );` bedeutet im Klartext:
1) Der `VertexBuffer vb` ist am Anfang der binäre Datei `Stream` Nr. `0` versandfertig verpackt.
2) `Stream` Nr. `0` ist bereit zum Transfer vom Main Memory (über den AGP- oder PCI-Express-Bus) zur Graphikkarte und wartet auf Abruf.
3) Wann immer die Graphikkarte Zeit und genügend Speicherplatz frei hat, wird sie den `Stream` in ihr Vertex-Buffer-Input-Memory einlesen und damit den evtl. dort vorhandenen `Stream` überschreiben.
Ein `Stream` kann mehrere `VertexBuffers` enthalten. Will man diese gleichzeitig rendern, dann muss man sie mit Hilfe des 3. Parameters von `SetStreamSource` in einem `Stream` hintereinander aufreihen. Man kann auch mehrere `Streams` auf Vorrat programmieren (Nummerierung = 1. Parameter von `SetStreamSource`), aber nur jeweils einer kann sich aktuell im Vertex-Buffer-Input-Memory der Graphikkarte befinden.
Schema der Transportkette der Vertices:



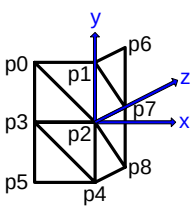
Zeitpunkt der Übergabe:
1) Wenn das Programm nur einen `VertexBuffer` hat oder wenn man mehrere in einem `Stream` verpackt hat, die man immer gemeinsam und gleichzeitig präsentieren will: In diesem Fall kopiert man am besten den `Stream` einmal unmittelbar nach der Initialisierung des `Device`. Die Vektordaten werden so nur einmal über den Bus geschickt und befinden sich ab dann dauerhaft in der Graphikkarte. Erst wenn `Device` ungültig wird und neu initialisiert werden muss, dann neu kopieren.
2) Wenn das Programm zwei oder mehr `VertexBuffer` hat, die man einzeln und nacheinander sehen soll, dann kopiert man sie nacheinander in der `OnTimer`-Funktion innerhalb der `Device.BeginScene() - Device.EndScene()`-Klammer. Die Vektordaten werden so permanent via Bus ausgetauscht, was flexibel aber aufwendig ist.
Beispiel für 2) mit zwei VertexBuffern `vb0` und `vb1`, die getrennt gerendert werden:

```
device.BeginScene();
  device.SetStreamSource(0, vb0, 0);
  device.VertexFormat = CustomVertex.PositionNormal.Format;
  device.DrawPrimitives(PrimitiveType.TriangleList, 0, 1);
  device.SetStreamSource(0, vb1, 0);
  device.VertexFormat = CustomVertex.PositionNormalTextured.Format;
  device.DrawPrimitives(PrimitiveType.TriangleList, 0, 1);
device.EndScene();
```

5) Stream rendern:
Befinden sich die Vertexdaten in der Graphikkarte (genauer: im Vertex-Buffer-Abschnitt des Graphikkartenspeichers) dann geht alles weitere automatisch. Die Daten durchlaufen eine Kette spezialisierter Mikroprocessoren: Tesselation->Fixed Vector Pipeline etc. bis am Ende der Backbuffer das fertige Rasterbild enthält. Nur den allerletzten Schritt muss man programmieren: `Device.Present();` schaltet den Backbuffer zum Frontbuffer.

Index Buffer

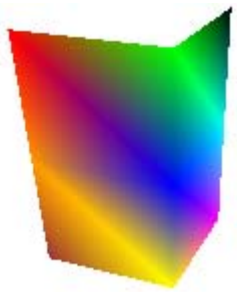
Problem: `TriangleStrips` sind nur dann elegant, wenn eine Figur aus einem Strip gewickelt ist. Bei komplizierter zusammengesetzten Flächen müssen die Vertices in der Regel zwei-, drei- oder mehrfach codiert werden, weil sie in zwei, drei oder mehr `TriangleStrips` enthalten sind. Das ist schlechte Redundanz = Ballast.



Beispiel mit acht Dreiecken:
p5 und p6 kommen in je einem Dreieck vor.
p0 und p8 kommen in je zwei Dreiecken vor.
p1, p3, p4 und p7 kommen in je drei Dreiecken vor.
p2 kommt in sechs Dreiecken vor.
Lösung 1: Fläche mit zwei `TriangleStrips` codieren:

```
linker Strip : p5,p4,p3,p2,p0,p1
rechter Strip: p4,p8,p2,p7,p1,p6
```

Nachteil 1: Wahl des Anfangspunktes und Reihenfolge ist schwierig.
Nachteil 2: Die Punkte auf der Naht `p1,p2,p4` sind doppelt.
Lösung 2: **Trennung der Koordinateninformation von der Reihenfolgeinformation durch zwei getrennte Datenstrukturen**:
a) Vertexbuffer für Vertexinformation, wobei die Reihenfolge der Vertices beliebig ist
b) Indexbuffer für die Reihenfolgeinformation



```
 1 static Mesh mymesh; //let's use a mesh to display the sample
 2 CustomVertex.PositionColored[] myvertexbuf = {
 3   new CustomVertex.PositionColored(-1, 1, 0,Color.Red    .ToArgb() ),   //p0
 4   new CustomVertex.PositionColored( 0, 1, 0,Color.Green  .ToArgb() ),   //p1
 5   new CustomVertex.PositionColored( 0, 0, 0,Color.Blue   .ToArgb() ),   //p2
 6   new CustomVertex.PositionColored(-1, 0, 0,Color.Orange .ToArgb() ),   //p3
 7   new CustomVertex.PositionColored( 0,-1, 0,Color.Yellow .ToArgb() ),   //p4
 8   new CustomVertex.PositionColored(-1,-1, 0,Color.Brown  .ToArgb() ),   //p5
 9   new CustomVertex.PositionColored( 0, 1, 1,Color.Black  .ToArgb() ),   //p6
10   new CustomVertex.PositionColored( 0, 0, 1,Color.Cyan   .ToArgb() ),   //p7
11   new CustomVertex.PositionColored( 0,-1, 1,Color.Magenta.ToArgb() ) }; //p8
12 Int16[] myindexbuf = { 0,1,2, 0,2,3, 3,2,4, 3,4,5, 1,6,7, 1,7,2, 2,7,8, 2,8,4 };
   //*********************************************************************
13 if ( mymesh != null ) mymesh.Dispose();// free the old mesh if any
14 mymesh = new Mesh( 8, myindexbuf.Length, MeshFlags.WriteOnly,
15                     CustomVertex.PositionColored.Format, device );
16 mymesh.VertexBuffer.SetData( myvertexbuf, 0, LockFlags.None );
17 mymesh.IndexBuffer .SetData( myindexbuf , 0, LockFlags.None );
   //*********************************************************************
18 device.BeginScene();
19   mymesh.DrawSubset(0);
20 device.EndScene();
21 device.Present();
```

Zeilen 1 bis 12 = globale Deklarationen im Kopf von `Form1`
Zeilen 13 bis 17 in `protected override void OnResize( ... )` nach der Initialisierung von `device`
Zeilen 18 bis 21 in `protected static void OnTimer( ... )`
Zeile 12 definiert den `IndexBuffer` in Form einer `TriangleList` mit den Vertexnummern von acht Dreiecken. Die Reihenfolge der Dreiecke ist ohne Bedeutung, aber die Reihenfolge der Ecken innerhalb jeden Dreiecks ist für die Unterscheidung von Vorder- und Rückseite wichtig. Konvention: Ecken immer so aufreihen, dass das Dreieck von vorne gesehen im Uhrzeigersinn umlaufen wird.

Es gibt zwei Möglichkeiten, Flächen durch Dreiecke zu definieren:
a) Geordnet fortlaufende Vertices bilden verbundene Dreiecke = `TriangleStrip`.
b) Ungeordnete Vertices plus `IndexBuffer` bilden unverbundene Dreiecke = `TriangleList`.
`TriangleStrip`: Codierung eines Dreiecksstreifens in Form fortlaufender Vertexkoordinaten, so dass jeder neue Vertex ein neues Dreieck zu den beiden vorletzten Vertices aufspannt.
Vorteil: redundanzarmer Code, weil jeder Vertex nur ein- oder zweimal genannt werden muss.
Nachteile:
1) Anfangspunkt und Reihung sorgfältig wählen, sonst werden neue Dreiecke an falschen Kanten aufgespannt.
2) Braucht man zwei seitlich miteinander verbundenen Strips, dann müssen gewisse Vertices im zweiten Strip wiederholt werden.
3) Man kann Vertices aus einem `TriangleStrip` nicht über einen `IndexBuffer` ansprechen.
4) Die mächtige Datenstruktur `Mesh` erlaubt als Elemente nur `TriangleLists`.
`TriangleList`: Codierung einzelner unverbundener Dreiecke mit Hilfe eines `IndexBuffers`.
Vorteile:
1) Dreiecke sehr flexibel in jeder beliebigen Reihenfolge codierbar.
2) `TriangleList` ist Baustein der mächtigen Datenstruktur `Mesh`.
Nachteil: Viele Vertices kommen im `IndexBuffer` mehrfach vor. Folge: `IndexBuffer` wird lang und unübersichtlich.