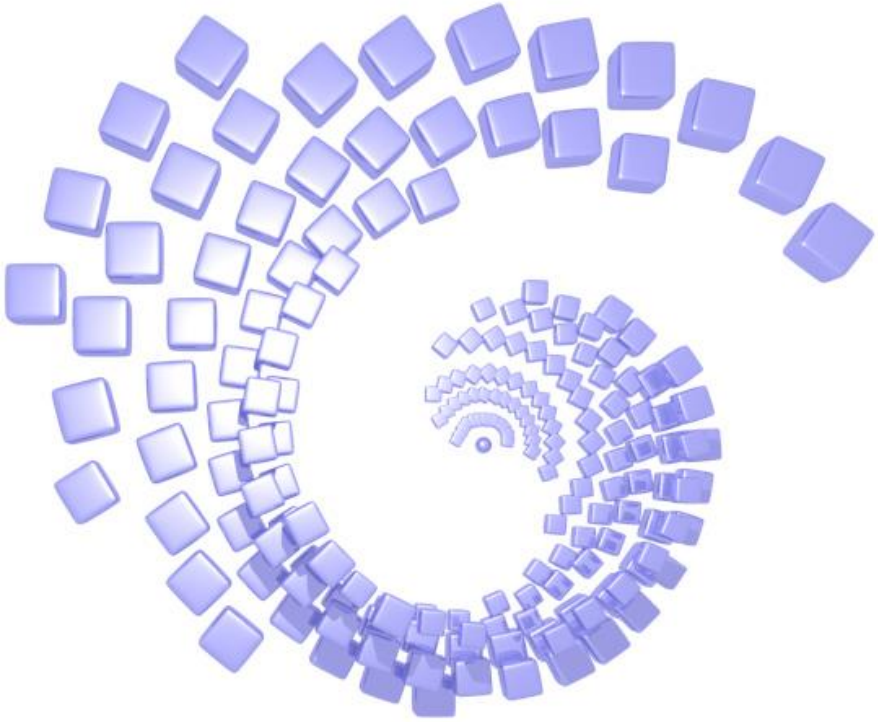
EIGENE GEOMETRIEN



Prof. Dr. Tobias Breiner

Professur für Game-Engineering

Fakultät für Informatik

Hochschule Kempten

Kontakt:

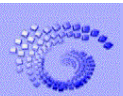
Gebäude S, Zi.319

Tel: 0831-2523-303

Fax: 0831-2523-300

tobias.breiner@fh-kempten.de

Inhalt



UEBERBLICK MESHES

2

3

4

5

Knotenobjekte der Szenegraphen

1
2
3
4
5

Virtuelle Szene

Gruppenknoten
- Gruppenbehälter
- Transformationen
- Switchknoten (Level-of-Detail, Animation etc.)
- …

Blattknoten
- Lichter
- Geometrien
- Kameras
- Materialien, Texturen, Images
- Farben
- ...

Reale Szene

Gruppenknoten
- Gruppenbehälter
- Computer
- Kanäle
- Switchknoten (an, aus)
- …

Blattknoten
- Eingabegeräte
- Sichtsysteme
- Sichtfenster
- Soundgeräte
- Basis-Render-API
- …

Knotenobjekte der Szenegraphen

Neben den parametrisierten Erzeugung von geometrischen Primitiven (Kugel, Quader, Tetraeder, Quad, …) gibt es die Möglichkeit, eigene geometrische Modelle zu erstellen.

Die Möglichkeit zur Erzeugung eigener Geometrien bieten nicht alle Szenegrafen bzw. Game- oder 3D-Engines.

In Vektoria gibt es gleich zwei Möglichkeiten zur Erzeugung eigener Geometrien: TriangleLists und TriangleStrips

CVertex

```
void Init
(
  CHVector vPos,
  CHVector vNormal,
  CHVector vTangent,
  float fU=0,
  float fV=0,
);
```

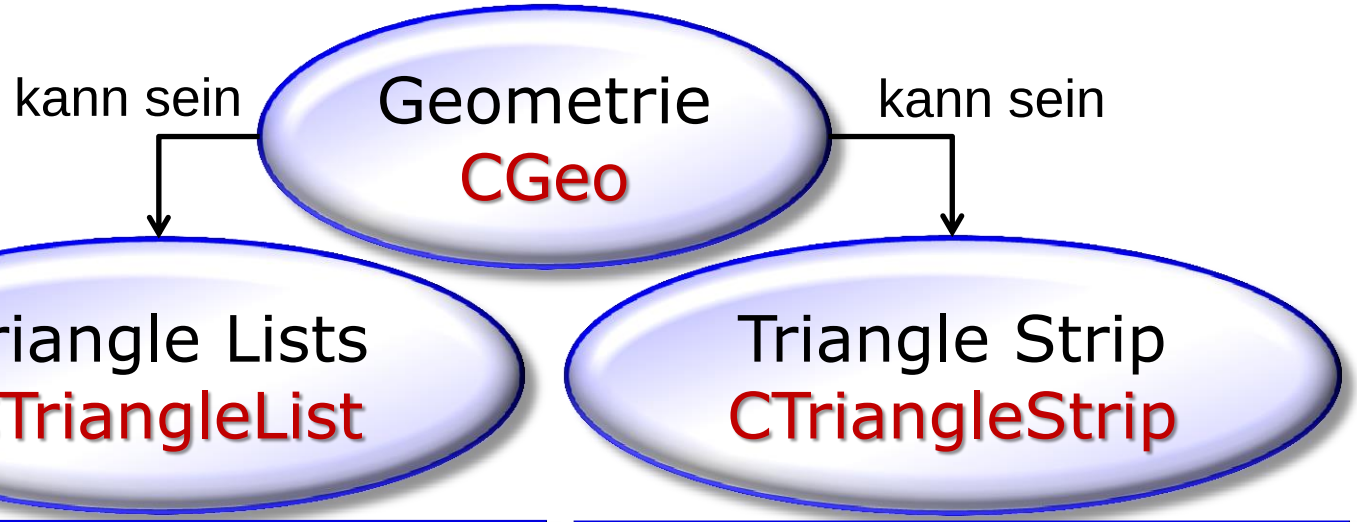
1 Position des Vertex im Raum

2 Oberflächennormale am Vertex

3 Tangente am Vertex
(wichtig für das Bumpmapping)

4 5 UV-Koordinaten

Arten von Geometrien



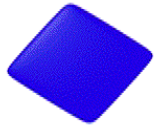
Besteht aus einzelnen Dreieckspolygonen
=>
für unzusammenhängende oder kantige Strukturen geeignet

Besteht aus einer Serie von Vertices, die ein zusammenhängendes Dreieckspolygonnetz definieren. =>
für zusammenhängende, runde Strukturen geeignet.

Kapitel 2

1

TRIANGLE LISTS

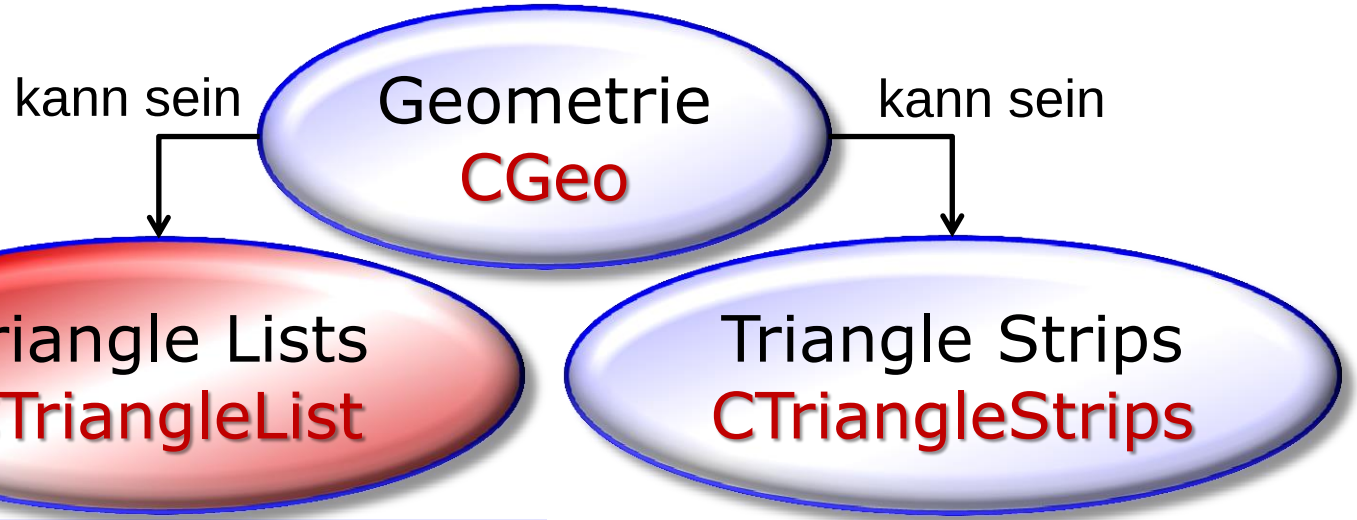


3

4

5

Triangle Lists



Besteht aus einzelnen Dreieckspolygonen
=>
für unzusammenhängende oder kantige Strukturen geeignet

Besteht aus einer Serie von Vertices, die ein zusammenhängendes Dreieckspolygonnetz definieren. =>
für zusammenhängende, runde Strukturen geeignet.

Triangle Lists – Pro und Cons

Vorteile:

- Einfacher als Triangle Strips
- Auch für kantige und/oder unzusammenhängende Geometrien verwendbar

Nachteile:

- Etwas Langsamer als Triangle Strips

Im Folgenden ein Beispiel, wie man mit Hilfe von Triangle Lists eine eigene Pyramide mit dreieckiger Grundfläche erzeugen kann.

Eigene Pyramide mit Triangle Lists (1/5)

```
// die 4 Positionen der Vertices bestimmen:
CHVector v0(0.0F,1.0F, 0.0F, 1.0F);
CHVector v1(0.0F, 0,  1.0F, 1);
CHVector v2(-1.0F, -1.0F, -1.0F, 1);
CHVector v3(+1.0F, -1.0F, -1.0F, 1);

// Die Richtungen der Oberflächennormalen der 4
Poylgone ausrechnen (für das Shading wichtig):
CHVector v021F = v0+v2+v1;
CHVector v301F = v3+v0+v1;
CHVector v203F = v2+v0+v3;
CHVector v123F = v1+v2+v3;
```

1

2

3

4

5

Eigene Pyramide mit Triangle Lists (2/5)

```
// Die Normalenvektoren normieren und aus den 4
Positionen eine Richtung generieren :
v021F.Normal();
v301F.Normal();
v203F.Normal();
v123F.Normal();
```

Eigene Pyramide mit Triangle Lists (5/5)

```
// Die Tangentenvektoren ausrechnen (wird nur für
eventuelles Dot-Bumpmapping gebraucht, ansonsten
kann man sich die Arbeit sparen):
CHVector v0T = v2-v1;
CHVector v1T = v0-v2;
CHVector v2T = v0-v3;
CHVector v3T = v2-v0;
v0T.Normal();
v1T.Normal();
v2T.Normal();
v3T.Normal();
```

Eigene Pyramide mit Triangle Lists (4/5)

```
// Die Vertices für Polygon 1 initialisieren:
m_avertex[0].Init(v0, v021F, v3T, 0   ,0);
m_avertex[1].Init(v2, v021F, v3T, 1   ,0);
m_avertex[2].Init(v1, v021F, v3T, 0.5F,1);

// Die Vertices für Polygon 2 initialisieren:
m_avertex[3].Init(v3, v301F, v2T, 0   ,0);
m_avertex[4].Init(v0, v301F, v2T, 1   ,0);
m_avertex[5].Init(v1, v301F, v2T, 0.5F,1);

// Die Vertices für Polygon 3 initialisieren:
m_avertex[6].Init(v2, v203F, v1T, 0   ,0);
m_avertex[7].Init(v0, v203F, v1T, 1   ,0);
m_avertex[8].Init(v3, v203F, v1T, 0.5F,1);
```

Triangle Lists

Eigene Pyramide mit Triangle Lists (5/5)

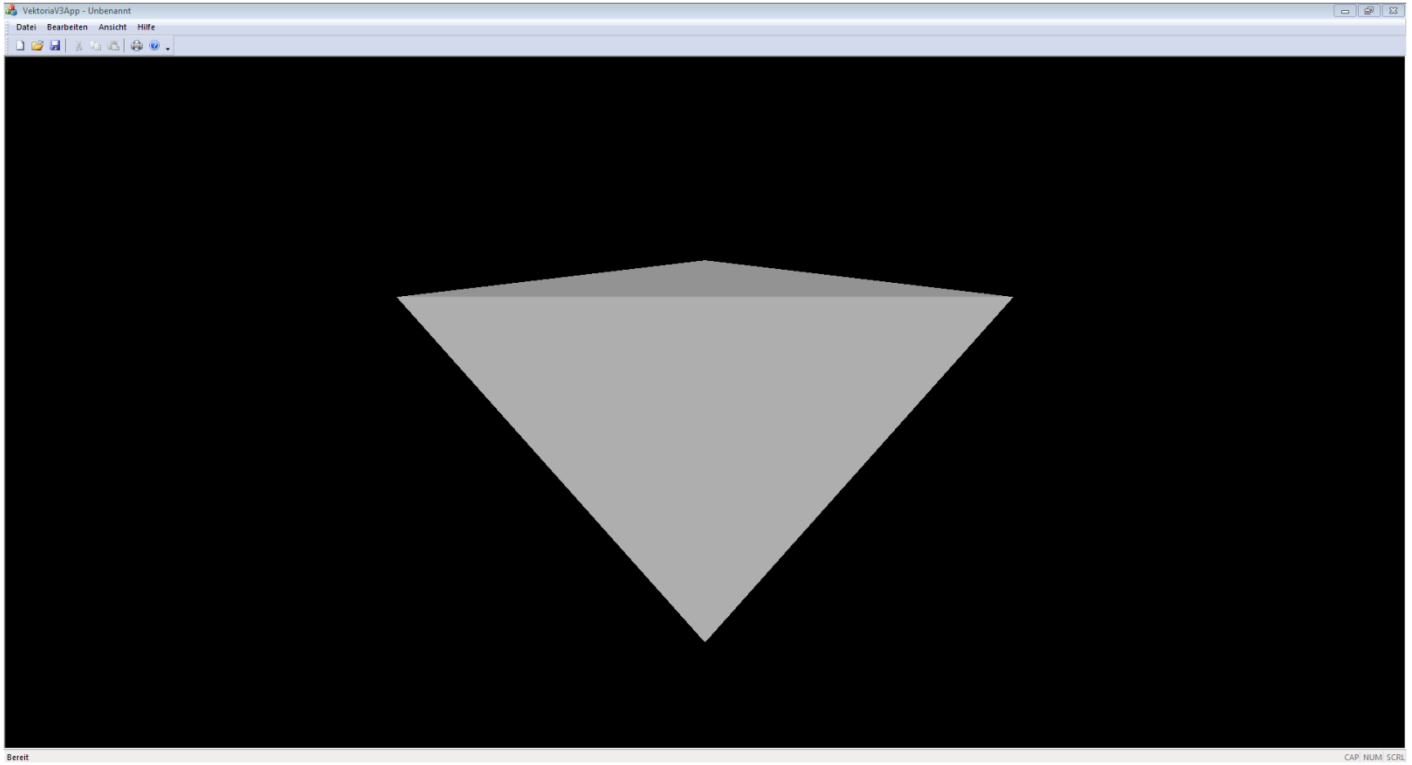
```
// Die Vertices für Polygon 4 initialisieren:
m_avertex[9].Init(v1, v123F, v0T, 0   ,0);
m_avertex[10].Init(v2, v123F, v0T,1   ,0);
m_avertex[11].Init(v3, v123F, v0T,0.5F,1);

// Die 12 Vertices der Triangle List hinzufügen:
for(int i=0;i<12;i++)
      m_trianglelist.AddVertex(&m_avertex[i]);

// Triangle List initialisieren und mit einem
Material belegen, es kann danach verwendet werden
wie eine „normale“ Geometrie:
m_trianglelist.Init();
m_trianglelist.SetMaterial(&m_material);
m_placement.AddGeo(&m_trianglelist);
```

1
2
3
4
5

Triangle Lists
Ergebnis der eigenen Pyramide



Kapitel 3

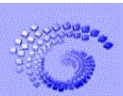
1

2

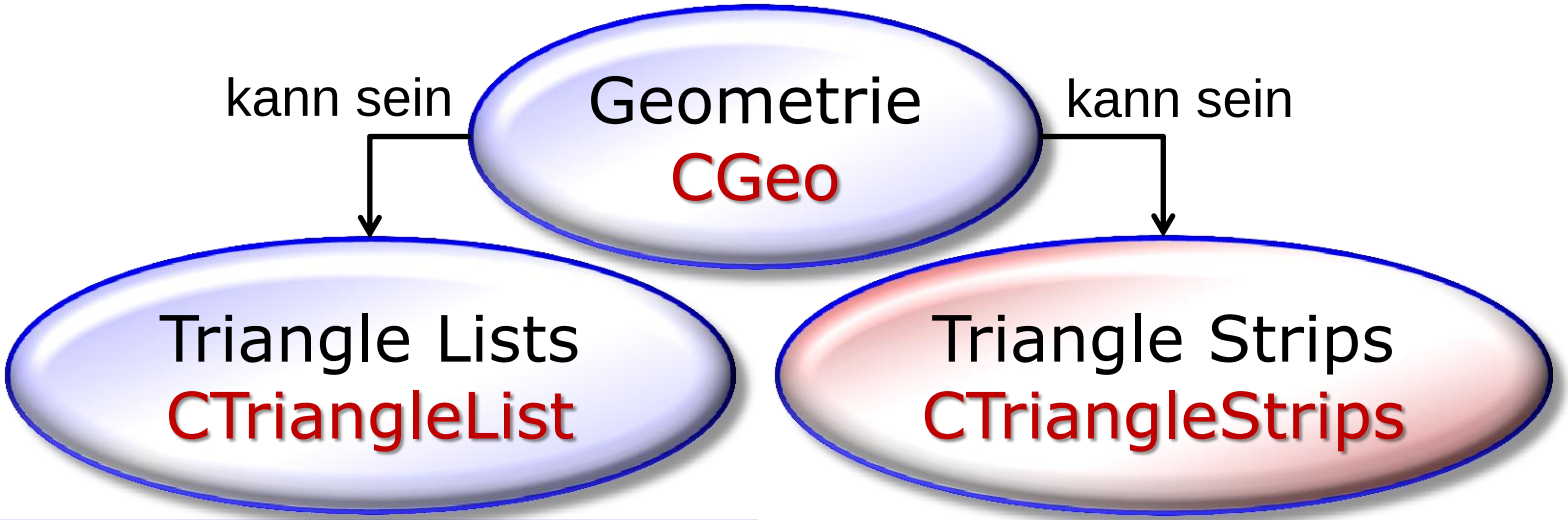
TRIANGLE STRIPS

4

5



Triangle Strips



Besteht aus einzelnen Dreieckspolygonen
=>
für unzusammenhängende oder kantige Strukturen geeignet

Besteht aus einer Serie von Vertices, die ein zusammenhängendes Dreieckspolygonnetz definieren. =>
für zusammenhängende, runde Strukturen geeignet.

Pro und Cons von Triangle Strips

Vorteile:

- Shader können etwas schneller mit Triangle Strips umgehen
  (Faustregel ca. 20% schneller)

Nachteile:

- Komplizierter als Triangle Lists
- Nur schwer kantige Strukturen erzeugbar
- Keine unzusammenhängenden Strukturen erzeugbar

Quad durch Triangle Strips

1. Im Folgenden ein Beispiel, wie man mittels Triangle Strips ein Quad erzeugen kann.

2. Ein Quad ist eine rechteckige Fläche im Raum, bestehend aus zwei Dreieckspolygonen

3.

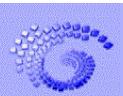
4.

5.

Eigenen Quad mit Triangle Strips (1/4)

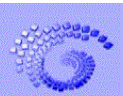
1
2
3
4
5

```
// Nun die 4 Vertices initalisieren:
m_avertex[0].Init(                      // Vertex 1:
        CHVector(-1,-1,0,1),            // Position
        CHVector(0,0,1,0),              // Normale
        CHVector(1,0,0,0),              // Tangente
        0.0F, 0.0f                      // UV-Koordinaten
);
```



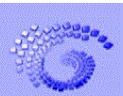
Eigenen Quad mit Triangle Strips (2/4)


```
m_avertex[1].Init(                      // Vertex 2:
        CHVector(+1,-1,0,1),            // Position
        CHVector(0,0,1,0),              // Normale
        CHVector(1,0,0,0),              // Tangente
        1.0F, 0.0f                      // UV-Koordinaten
);
m_avertex[2].Init(                      // Vertex 3:
        CHVector(-1,+1,0,1),            // Position
        CHVector(0,0,1,0),              // Normale
        CHVector(1,0,0,0),              // Tangente
        0.0F, 1.0f                      // UV-Koordinaten
);
```



Eigenen Quad mit Triangle Strips (3/4)


```
m_avertex[3].Init(                      // Vertex 4:
        CHVector(+1,+1,0,1),            // Position
        CHVector(0,0,1,0),              // Normale
        CHVector(1,0,0,0),              // Tangente
        1.0F, 1.0f,                     // UV-Koordinaten
);
// Die 4 Vertices gehören zum Triangle Strip:
for(int i=0;i<4;i++)
{
        m_ptrianglestrip->AddVertex(&m_avertex[i]);
        m_ptrianglestrip->AddIndex(i);
}
```



Triangle Strips

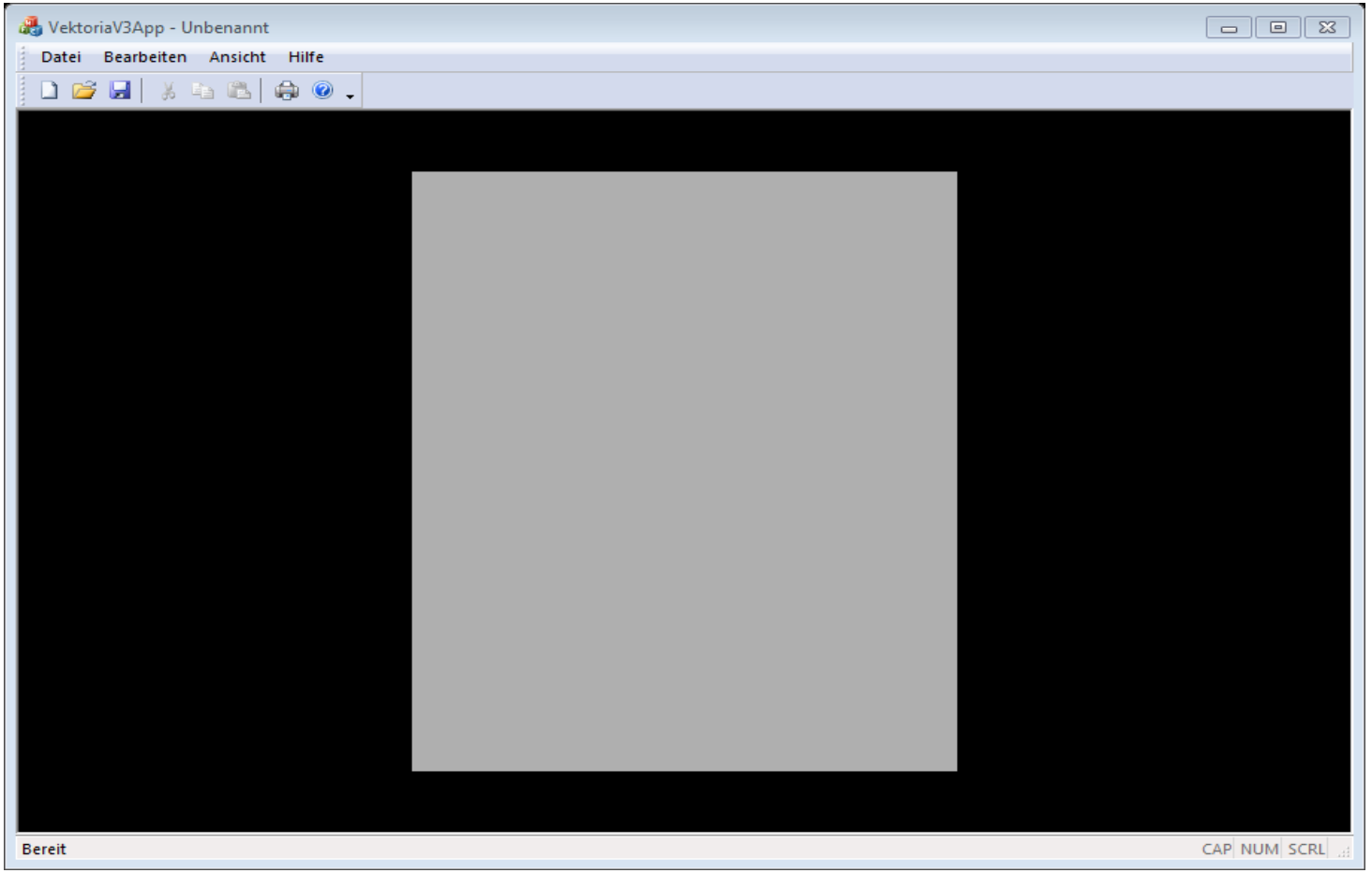
Eigenen Quad mit Triangle Strips (4/4)

```
// Initialisiere den Triangle Strip:
m_ptrianglestrip->Init();

// Mappe Material auf das Rechteck:
m_trianglestrip.SetMaterial(&m_material);

// und nun kannst Du Deinen eigenen persönlichen
Triangle Strip benutzen, wie eine ganz normale
Geometrie:
m_placement.AddGeo(&m_trianglestrip);
```

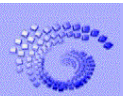
Triangle Strips
Ergebnis: Eigener Quad



Triangle Strips
Übung

1. Erzeugen Sie einen Pfeiler mit Hilfe von Triangle Strips!
2. Freiwillige Zusatzaufgabe für die schnellen Nerds:
3. Erzeugen Sie eine Prozedur, die parametrisiert dorische und ionische Pfeiler erzeugt!
4.
5.

Kapitel 4



Importer

Blender-Import

Achtung, Nr. 1! Bisher ist es nur möglich, Blender Files von 32-Bit Blender-Versionen zu importieren!

Achtung, Nr. 2! Es wird immer nur das allererste Mesh aus einem Blender-File importiert!

In der Game.h:

```
CGeo * m_pzg;
CFileBlender m_fileblender;
```

In der Game.cpp / Init-Methode:

```
m_pzg = m_fileBlender.LoadGeo(“myfile.blend");
```

Importer

Wavefront-Import

1

In der Game.h:

2

```
CGeo * m_pzg;
CFileWavefront m_filewavefront;
```

3

4

In der Game.cpp / Init-Methode:

```
m_pzg = m_filewavefront.LoadGeo(“myfile.wvf");
```

5

Importer

3D Studio Max-Import

1

In der Game.h:

2

```
CGeo * m_pzg;
CFile3DS m_file3ds;
```

3

4

In der Game.cpp / Init-Methode:

```
m_pzg = m_file3ds.LoadGeo(“myfile.3ds");
```

5

X3D-Import

Achtung, Nr. 1! X3D-Files enthalten für gewöhnlich weder Normalenvektoren noch UV-Koordinaten! Diese Daten werden daher von Vektoria heuristisch abgeschätzt. Die meisten importierten Meshes werden daher vom Shading und von der Texturierung her etwas „seltsam“ aussehen.

Achtung, Nr. 2! Es wird immer nur das allererste Mesh aus einem X3D-File importiert!

In der Game.h:

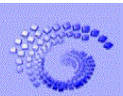
```
CGeo * m_pzg;
CFileX3D m_filex3d;
```

In der Game.cpp / Init-Methode:

```
m_pzg = m_filex3d.LoadGeo(“myfile.x3d");
```

Kapitel 5

ALLGEMEINES



Tipps

Tipp: Ziehe möglichst eine in Vektoria prozedural erzeugte Geometrie bzw. eine vorgefertigte Vektoria-Primitivengeometrie einer importierten Geometrie vor!

Die Gründe dafür sind:

1.) Prozedural erzeugte Geometrien sind (meistens) sparsamer hinsichtlich der Vertices und genauer durchdacht.

2.) Geometrieerzeugende Prozedur kann möglicherweise wiederverwendet werden

3.) Tangenten- und Bitangentenkoordinaten für das Bumpmapping werden in Modellingprogrammen oft nicht gut generiert.

4.) Beim Importieren können Fehler auftreten

Achtung! Die Form einer einmal erzeugten Geometrie kann danach nicht mehr verändert werden!

GAME OVER

Danke für Ihr Interesse!