



EIGENE GEOMETRIEN



Eigene Geometrien

Inhalt



/// **UEBERBLICK MESHES** //

/// **TRIANGLE LISTS** //

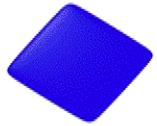
/// **TRIANGLE STRIPS** //

/// **IMPORTER** //

/// **ALL GEMEINES** //



UEBERBLICK MESSES



2 // /

3 // /

4 // /

5 // /



Überblick über Knotenobjekte für Oberflächen

Knotenobjekte der Szeneraphen



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 Szeneraphen

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 Szeneraphen bzw. Game- oder 3D-Engines.

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



Knotenobjekte der Szeneraphen

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 Szeneraphen bzw. Game- oder 3D-Engines.

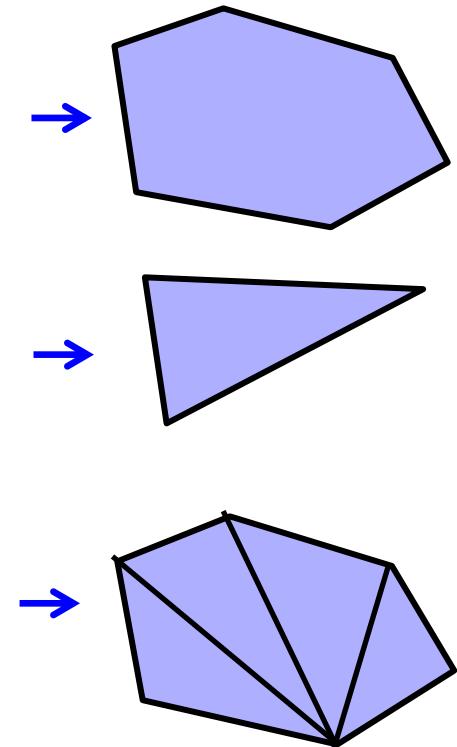
In Vektoria gibt es gleich zwei Möglichkeiten zur Erzeugung eigener Geometrien:
TriangleLists und **TriangleStrips**



Eigene Geometrien

Grundlagen

Bei Flächenmodellen wird die Oberfläche eines Objektes in Form von Polygonen (Vielecken) beschrieben..



In Vektoria sind nur Dreiecke als Polygone erlaubt.

Werden Vierecke, Fünfecke, Sechsecke, etc. benötigt, müssen diese durch mehrere Dreiecke ersetzt werden. Das obige Polygon, könnte man z.B. folgendermaßen aufteilen.

5 // / / /



Eigene Geometrien

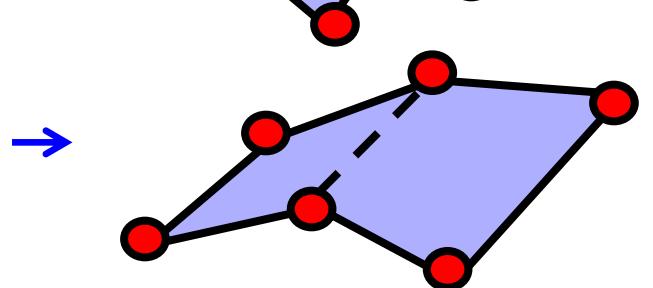
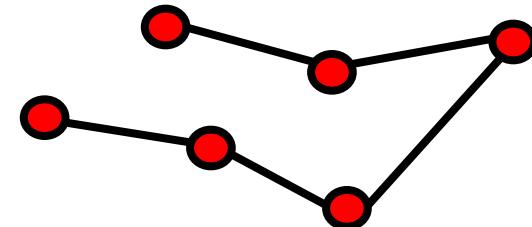
Grundlagen

Indem Vektoria sich auf Dreiecke als Polygone beschränkt, wird eine Menge Ärger vermieden,

insbesondere mit Ungeschlossenheit,

Kreuzvernetzungen

und Eckpunkten, die nicht in einer Ebene liegen.



Eigene Geometrien

Grundlagen

1 // / / /

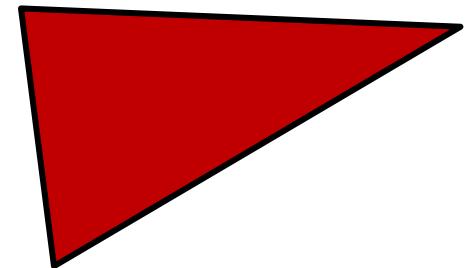
2 // / / /

3 // / / /

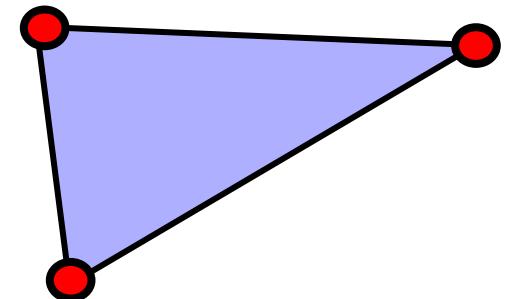
4 // / / /

5 // / / /

Die von einem Polygon eingeschlossene Fläche wird als **Facette** (engl. Face) bezeichnet,



und die Eckpunkte eines Polygons werden in der Computergrafik **Vertices** genannt (Singular: **Vertex**).



Eigene Geometrien

Grundlagen

1 // /

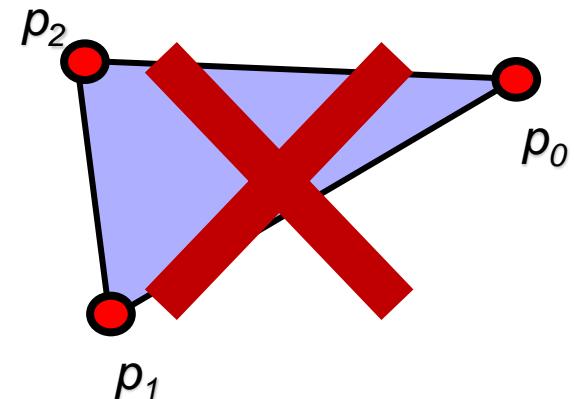
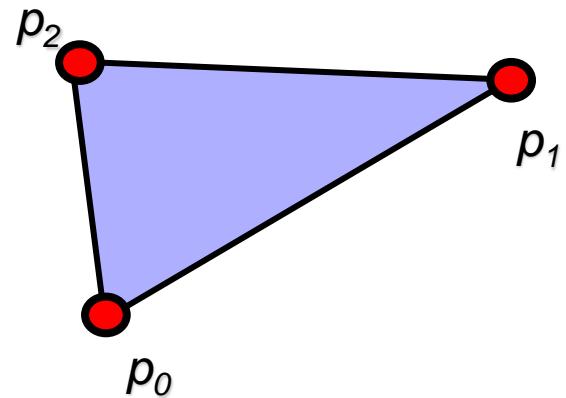
2 // /

3 // /

4 // /

5 // /

Es ist wichtig, dass die Vertices in einem rechtshändigen System auch rechtsdrehend vernetzt sind
(mathematisch positive Rotation, wenn man das Objekt von Außen betrachtet).



Und nicht linksdrehend, sonst zeigt die Flächennormale der Facette nach innen.

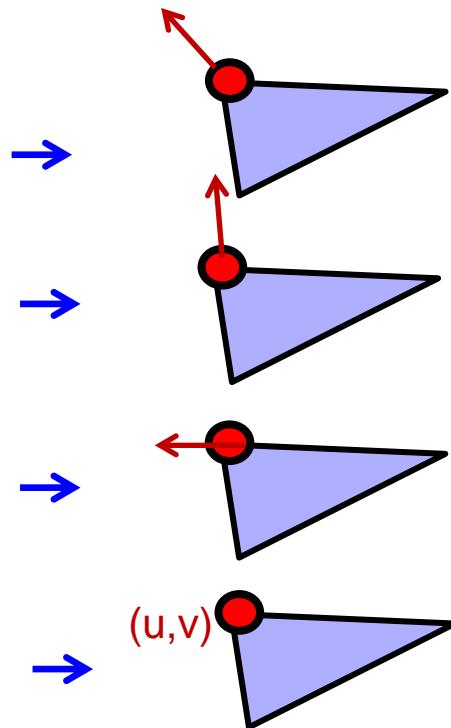


Eigene Geometrien

Grundlagen

Ein Vertex trägt für gewöhnlich neben seiner 3D-Position im Raum auch noch andere Informationen, in Vektoria sind dies konkret:

- **Oberflächennormale** (für das Shading wichtig)
- **Tangente** (für das Dot3-Bumpmapping wichtig)
- **Bitangente** (für das Dot3-Bumpmapping wichtig)
- **UV-Koordinaten** (gibt die Position der Textur am Vertex an)



CVertex::Init

1 // / / /

void Init

(

CHVector vPos,

CHVector vNormal,

CHVector vTangent,

CHVector vBitangent,

float fU=0,

float fV=0,

);

Normalerweise muss die Bitangente nicht angegeben werden, sie errechnet sich mittels der Vertexnormalen und der Vertextangenten mittels dem Kreuzprodukt.

2 // / / /

CHVector vTangent,

CHVector vBitangent,

float fU=0,

float fV=0,

);

In seltenen Fällen ist die Bitangente aber nicht rechtshändig orthogonal. Dann muss sie explizit mit angegeben werden.

Arten von Geometrien

kann sein

Geometrie
CGeo

kann sein

Triangle Lists
CTriangleList

Triangle Strip
CTriangleStrip

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

Kapitel 2



/// **TRIANGLE LISTS** //



Triangle Lists

Triangle Lists

kann sein

Geometrie
CGeo

kann sein

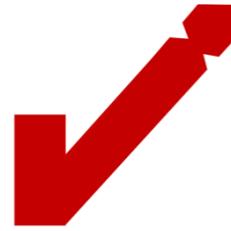
Triangle Lists
CTriangleList

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

Triangle Strips
CTriangleStrips

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





Triangle Lists

Triangle Lists – Pro und Cons



Vorteile:

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



Nachteile:

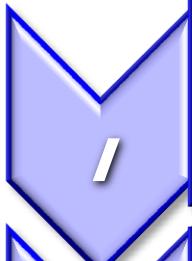
- Etwas Langsamer als Triangle Strips

Im Folgenden ein Beispiel, wie man mit Hilfe von Triangle Lists einen Einheitswürfel erzeugen kann.



Allgemeines Vorgehen bei Triangle Lists

1 // / / /



- Vertices erzeugen
- CVertex::Init(...)

2 // / / /



- Vertices zu Polygonen vernetzen
- CTriangleList::AddVertex(...)

3 // / / /



- Dreiecksliste initialisieren
- CTriangleList::Init();

4 // / / /

5 // / / /

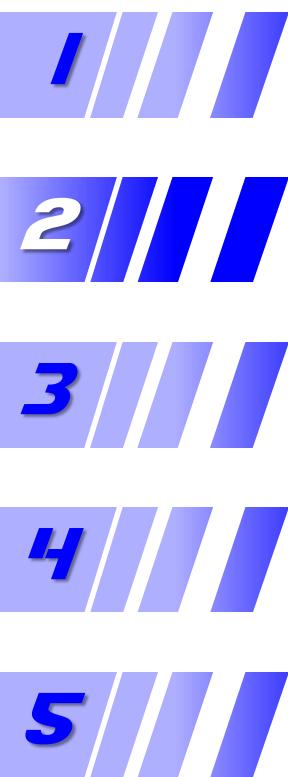
|||||DIE INITIALISIERTE |||||TRIANGLE |||||LIST KANN MAN
WIE EINE NORMALE |||||GEOMETRIE VERWENDEN.





Triangle Lists - Einheitswürfel mit Triangle Lists

Benötigte Variablen im Header



```
class CGeoUnitCube : public CTriangleList
{
public:
    // In den Konstruktor wird alles reingeschrieben:
    CGeoUnitCube(void);
    // der Destruktor kann erst einmal leer bleiben:
    ~CGeoUnitCube(void);

    // Pro Würfelseite brauche ich 4 Vertices.
    // Ein Würfel hat 6 Seiten.
    // Also brauche ich 4 * 6 = 24 Vertices
    // (Acht Vertices miteinander zu teilen ist keine
    // gute Idee, da die Normalen, Tangenten und
    // Bitangenten pro Seite anders sind):
    CVertex m_avertex[24];
};
```



Benötigte Variablen im Header

CGeoUnitCube :: CGeoUnitCube(void)
{
 // ...
}

Hier in den Konstruktorkörper kommt der Code der folgenden Seiten rein. Damit wird der Einheitswürfel gleich am Anfang mit den richtigen Vertices und Polygonen aufgebaut.

}

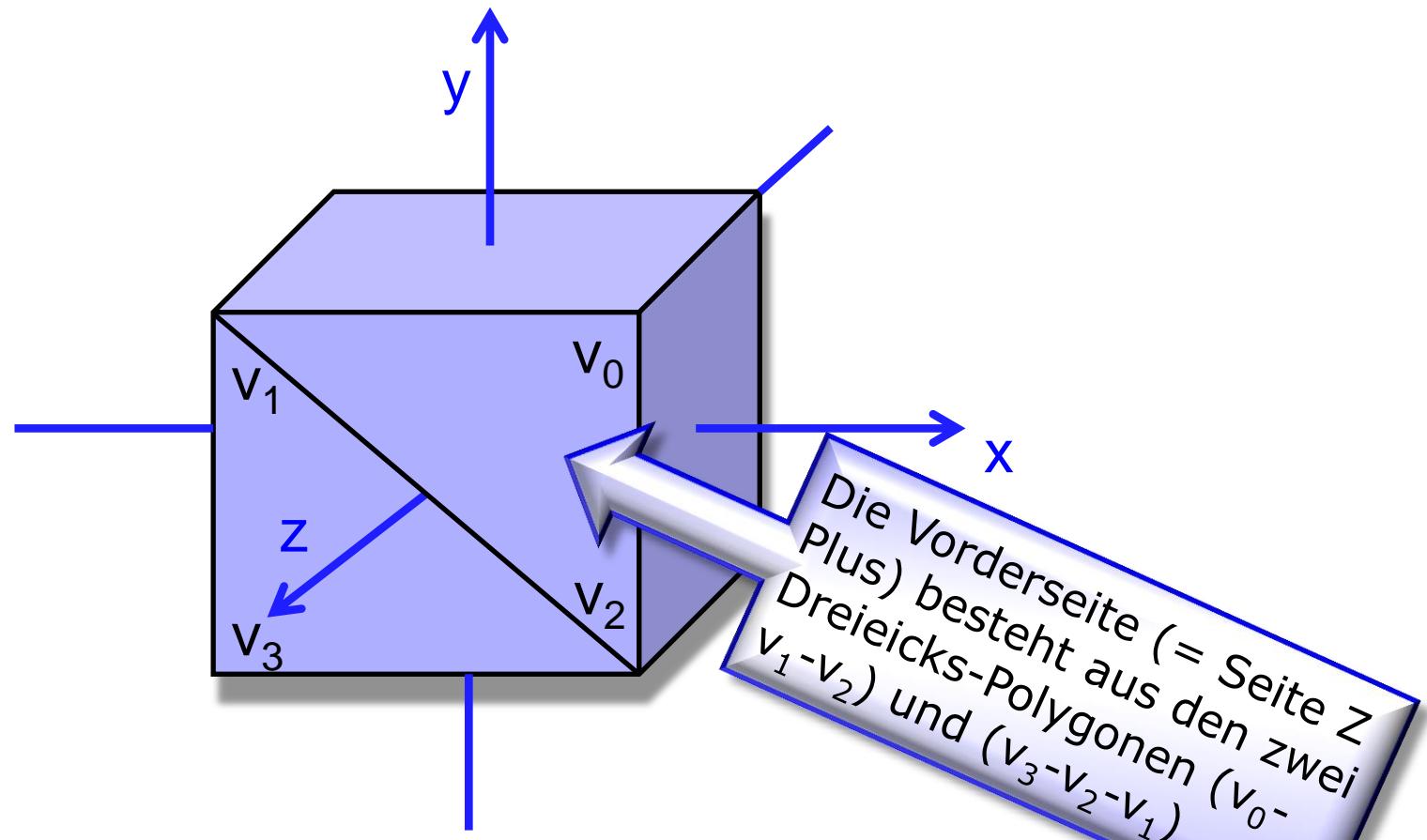
CGeoUnitCube :: ~CGeoUnitCube(void)
{

Der Destruktor kann erst einmal leer bleiben

}



Nummerierung der Vorderseite



1 // / /

2 // / /

3 // / /

4 // / /

5 // / /





Triangle Lists - Einheitswürfel mit Triangle Lists

Aufbau der Vorderseite



```
// Zuerst muss die Seite Z Plus gebaut werden:  
m_avertex[0].Init( // vertex 0 von Seite Z+:  
    CHVector(+1.0f,+1.0f,+1.0f), // Position  
    CHVector( 0.0f, 0.0f, 1.0f), // Normale  
    CHVector(-1.0f, 0.0f, 0.0f), // Tangente  
    0.0f, 1.0f); // UV-Koordinaten  
  
m_avertex[1].Init( // vertex 1 von Seite Z+:  
    CHVector(-1.0f,+1.0f,+1.0f), // Position  
    CHVector( 0.0f, 0.0f, 1.0f), // Normale  
    CHVector(-1.0f, 0.0f, 0.0f), // Tangente  
    1.0f, 1.0f); // UV-Koordinaten  
  
m_avertex[2].Init( // Vertex 2 von Seite Z+:  
    CHVector(+1.0f,-1.0f,+1.0f), // Position  
    CHVector( 0.0f, 0.0f, 1.0f), // Normale  
    CHVector(-1.0f, 0.0f, 0.0f), // Tangente  
    0.0f, 0.0f); // UV-Koordinaten
```





Triangle Lists - Einheitswürfel mit Triangle Lists

Aufbau der Vorderseite



1 // / / /

2 // / / /

3 // / / /

4 // / / /

5 // / / /

```
m_avertex[3].Init(          // Vertex 3 von Seite Z+:  
    CHVector(-1.0f, -1.0f, +1.0f), // Position  
    CHVector( 0.0f,  0.0f,  1.0f), // Normale  
    CHVector(-1.0f, 0.0f, 0.0f), // Tangente  
    1.0f, 0.0f);                // UV-Koordinaten
```

// Jetzt haben wir die 4 Vertices der Vorderseite
// definiert, wir müssen noch die Vertices
// zu Dreieckspolygonen vernetzen:

// 1. Dreieck auf der Z+-Seite:

```
AddVertex(&m_avertex[0]);  
AddVertex(&m_avertex[1]);  
AddVertex(&m_avertex[2]);
```

Auf rechtshändige Drehrichtung achten!

// 2. Dreieck auf der Z+-Seite:

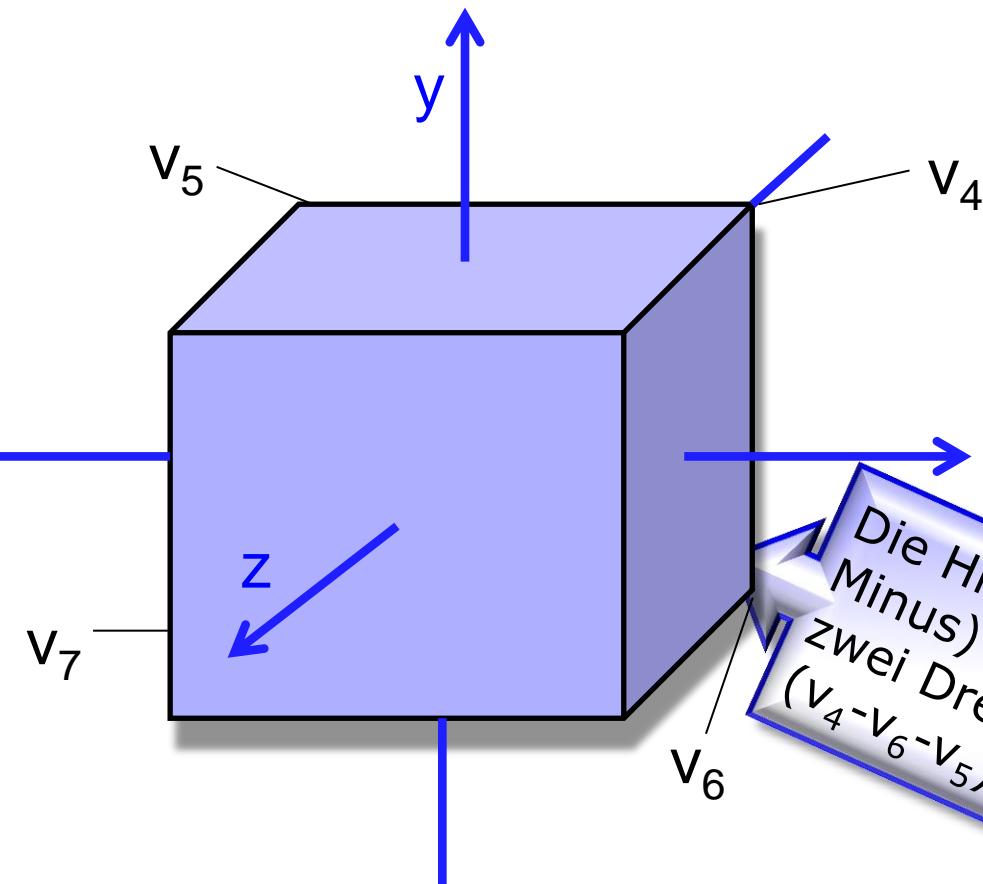
```
AddVertex(&m_avertex[3]);  
AddVertex(&m_avertex[2]);  
AddVertex(&m_avertex[1]);
```

Auf rechtshändige Drehrichtung achten!



Nummerierung der Rückseite

- 1 // /
- 2 // /
- 3 // /
- 4 // /
- 5 // /



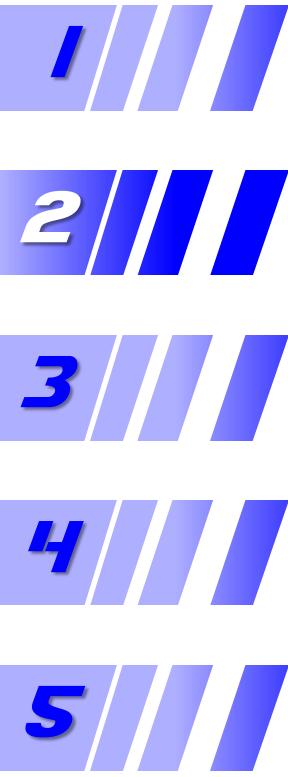
Die Hinterseite (= Seite Z Minus) besteht aus den zwei Dreiecks-Polygonen $(v_4-v_6-v_5)$ und $(v_5-v_6-v_7)$





Triangle Lists - Einheitswürfel mit Triangle Lists

Aufbau der Rückseite



```
// Dann muss die Seite Z Minus gebaut werden:  
m_avertex[4].Init( // vertex 4 von Seite Z-:  
    CHVector(+1.0f,+1.0f,-1.0f), // Position  
    CHVector( 0.0f, 0.0f,-1.0f), // Normale  
    CHVector( 1.0f, 0.0f, 0.0f), // Tangente  
    0.0f, 1.0f); // UV-Koordinaten  
  
m_avertex[5].Init( // vertex 5 von Seite Z-:  
    CHVector(-1.0f,+1.0f,-1.0f), // Position  
    CHVector( 0.0f, 0.0f,-1.0f), // Normale  
    CHVector( 1.0f, 0.0f, 0.0f), // Tangente  
    1.0f, 1.0f); // UV-Koordinaten  
  
m_avertex[6].Init( // Vertex 6 von Seite Z-:  
    CHVector(+1.0f,-1.0f,-1.0f), // Position  
    CHVector( 0.0f, 0.0f,-1.0f), // Normale  
    CHVector( 1.0f, 0.0f, 0.0f), // Tangente  
    0.0f, 0.0f); // UV-Koordinaten
```





Triangle Lists - Einheitswürfel mit Triangle Lists

Aufbau der Rückseite

1 // / / /

```
m_avertex[7].Init(          // Vertex 7 von Seite Z -:  
    CHVector(-1.0f, -1.0f, -1.0f), // Position  
    CHVector( 0.0f, 0.0f, -1.0f), // Normale  
    CHVector( 1.0f, 0.0f, 0.0f), // Tangente  
    1.0f, 0.0f);                // UV-Koordinaten
```

2 // / / /

```
// Jetzt haben wir die 4 Vertices der Hinterseite  
// definiert, wir müssen nun noch die Vertices zu  
// Dreieckspolygonen vernetzen:
```

3 // / / /

```
// 1. Dreieck auf der Z Minus-Seite:
```

```
AddVertex(&m_avertex[4]);  
AddVertex(&m_avertex[6]);  
AddVertex(&m_avertex[5]);
```

Auf rechtshändige
Drehrichtung achten!

4 // / / /

```
// 2. Dreieck auf der Z Minus-Seite:
```

```
AddVertex(&m_avertex[5]);  
AddVertex(&m_avertex[6]);  
AddVertex(&m_avertex[7]);
```

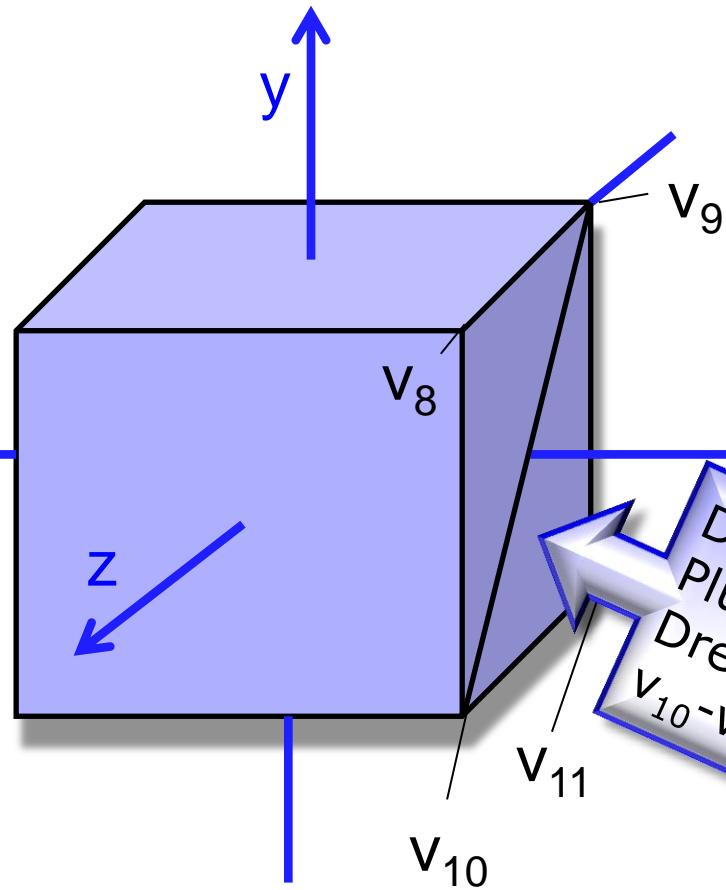
Auf rechtshändige
Drehrichtung achten!



Triangle Lists - Einheitswürfel mit Triangle Lists

Nummerierung auf der rechten Seite

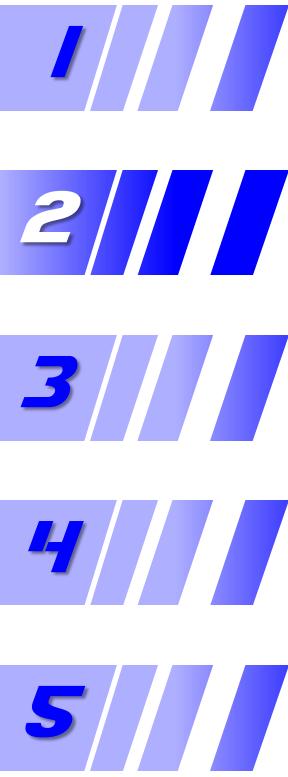
- 1 // /
- 2 // /
- 3 // /
- 4 // /
- 5 // /





Triangle Lists - Einheitswürfel mit Triangle Lists

Aufbau der rechten Seite



```
// Seite X Plus:  
m_avertex[8].Init( // vertex 8 von Seite X+:  
    CHVector(+1.0f,+1.0f,+1.0f), // Position  
    CHVector( 1.0f, 0.0f, 0.0f), // Normale  
    CHVector( 0.0f, 1.0f, 0.0f), // Tangente  
    1.0f, 0.0f); // UV-Koordinaten  
  
m_avertex[9].Init( // vertex 9 von Seite X+:  
    CHVector(+1.0f,+1.0f,-1.0f), // Position  
    CHVector( 1.0f, 0.0f, 0.0f), // Normale  
    CHVector(0.0f, 1.0f, 0.0f), // Tangente  
    1.0f, 1.0f); // UV-Koordinaten  
  
m_avertex[10].Init( // Vertex 10 von Seite X+:  
    CHVector(+1.0f,-1.0f,+1.0f), // Position  
    CHVector( 1.0f, 0.0f, 0.0f), // Normale  
    CHVector( 0.0f, 1.0f, 0.0f), // Tangente  
    0.0f, 0.0f); // UV-Koordinaten
```





Triangle Lists - Einheitswürfel mit Triangle Lists

Aufbau der rechten Seite



1 // / / /

2 // / / /

3 // / / /

4 // / / /

5 // / / /

```
m_avertex[11].Init(          // Vertex 11 von Seite X+:  
    CHVector(+1.0f, -1.0f, -1.0f), // Position  
    CHVector( 1.0f,  0.0f,  0.0f), // Normale  
    CHVector( 0.0f,  1.0f,  0.0f), // Tangente  
    0.0f, 1.0f);                // UV-Koordinaten
```

// Jetzt haben wir die 4 Vertices der rechten Seite definiert, wir müssen nun noch die Vertices zu Dreieckspolygonen vernetzen:

// 1. Dreieck auf der X Plus-Seite:

```
AddVertex(&m_avertex[8]);  
AddVertex(&m_avertex[10]);  
AddVertex(&m_avertex[9]);
```

// 2. Dreieck auf der X Plus-Seite:

```
AddVertex(&m_avertex[9]);  
AddVertex(&m_avertex[10]);  
AddVertex(&m_avertex[11]);
```



Triangle Lists - Einheitswürfel mit Triangle Lists

Nummerierung auf der linken Seite

1 // /

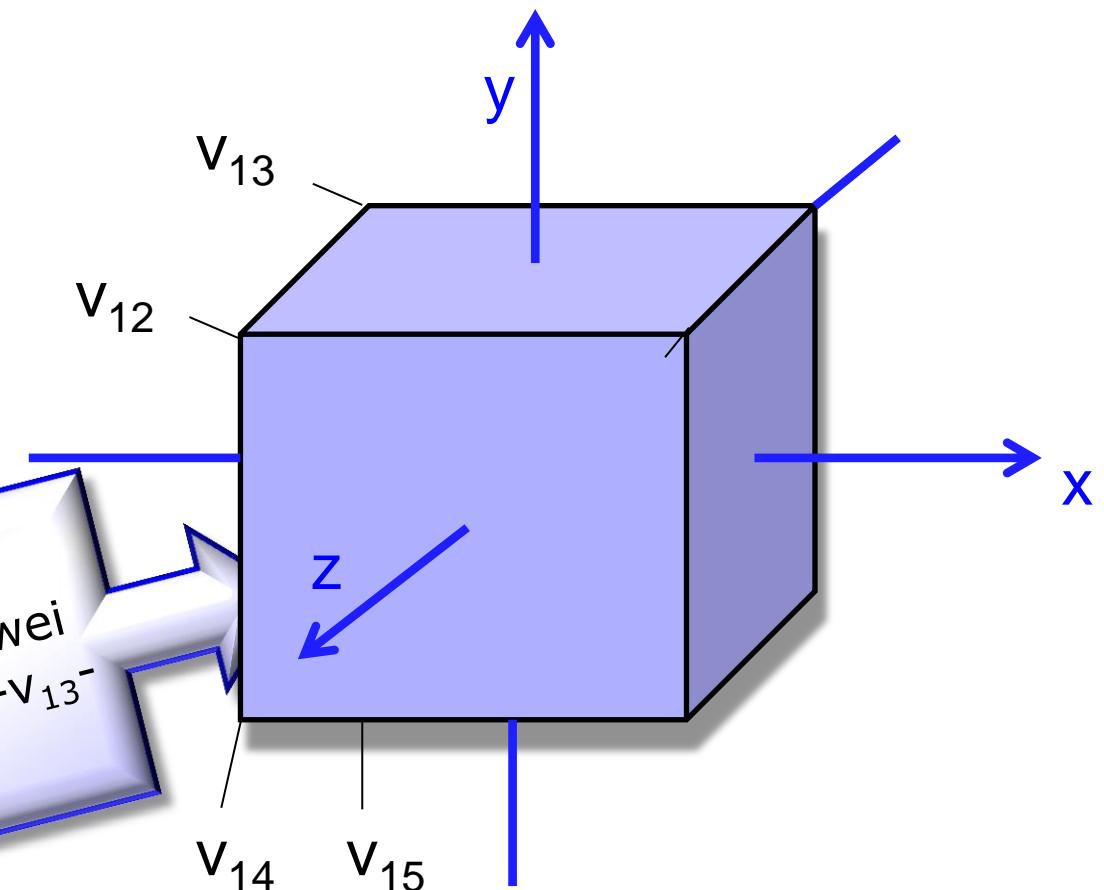
2 // /

3 // /

4 // /

5 // /

Die linke Seite (= Seite X Minus) besteht aus den zwei Dreiecks-Polygonen ($v_{12}-v_{13}-v_{15}$) und ($v_{12}-v_{15}-v_{14}$).





Triangle Lists - Einheitswürfel mit Triangle Lists

Aufbau der linken Seite



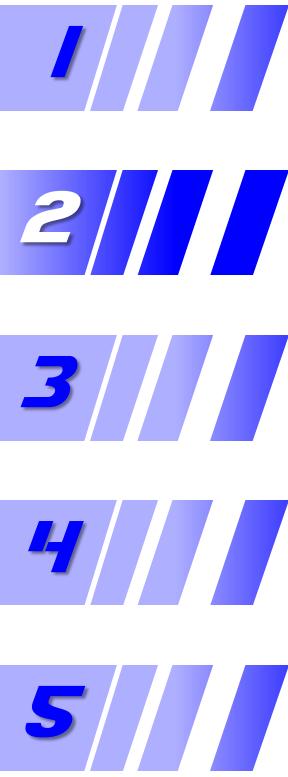
```
// Seite X Minus:  
m_avertex[12].Init ( // Vertex 12 von Seite X-:  
    CHVector(-1.0f,+1.0f,+1.0f), // Position  
    CHVector(-1.0f, 0.0f, 0.0f), // Normale  
    CHVector( 0.0f, 1.0f, 0.0f), // Tangente  
    1.0f,1.0f); // UV-Koordinaten  
  
m_avertex[13].Init ( // Vertex 13 von Seite X-:  
    CHVector(-1.0f,+1.0f,-1.0f), // Position  
    CHVector(-1.0f, 0.0f, 0.0f), // Normale  
    CHVector( 0.0f, 1.0f, 0.0f), // Tangente  
    1.0f,0.0f); // UV-Koordinaten  
  
m_avertex[14].Init( // Vertex 14 von Seite X-:  
    CHVector(-1.0f,-1.0f,+1.0f), // Position  
    CHVector(-1.0f, 0.0f, 0.0f), // Normale  
    CHVector( 0.0f, 1.0f, 0.0f), // Tangente  
    0.0f,1.0f); // UV-Koordinaten
```





Triangle Lists - Einheitswürfel mit Triangle Lists

Aufbau der linken Seite



```
m_avertex[15].Init(          // Vertex 15 von Seite X-:  
    CHVector(-1.0f, -1.0f, -1.0f), // Position  
    CHVector(-1.0f, 0.0f, 0.0f), // Normale  
    CHVector( 0.0f, 1.0f, 0.0f), // Tangente  
    0.0f, 0.0f);                // UV-Koordinaten  
  
// Jetzt haben wir die 4 Vertices der linken Seite  
// definiert, wir müssen nun noch die Vertices zu  
// Dreieckspolygonen vernetzen:  
  
// 1. Dreieck auf der linken Seite:  
AddVertex(&m_avertex[12]);  
AddVertex(&m_avertex[13]);  
AddVertex(&m_avertex[15]);  
// 2. Dreieck auf der linken Seite:  
AddVertex(&m_avertex[12]);  
AddVertex(&m_avertex[15]);  
AddVertex(&m_avertex[14]);
```





Triangle Lists - Einheitswürfel mit Triangle Lists

Übung: Ober- und Unterseite



Das Prinzip sollte jetzt klar geworden sein, es fehlen aber noch Ober- und Unterseite. Diese Aufgabe bleibt Ihnen überlassen.

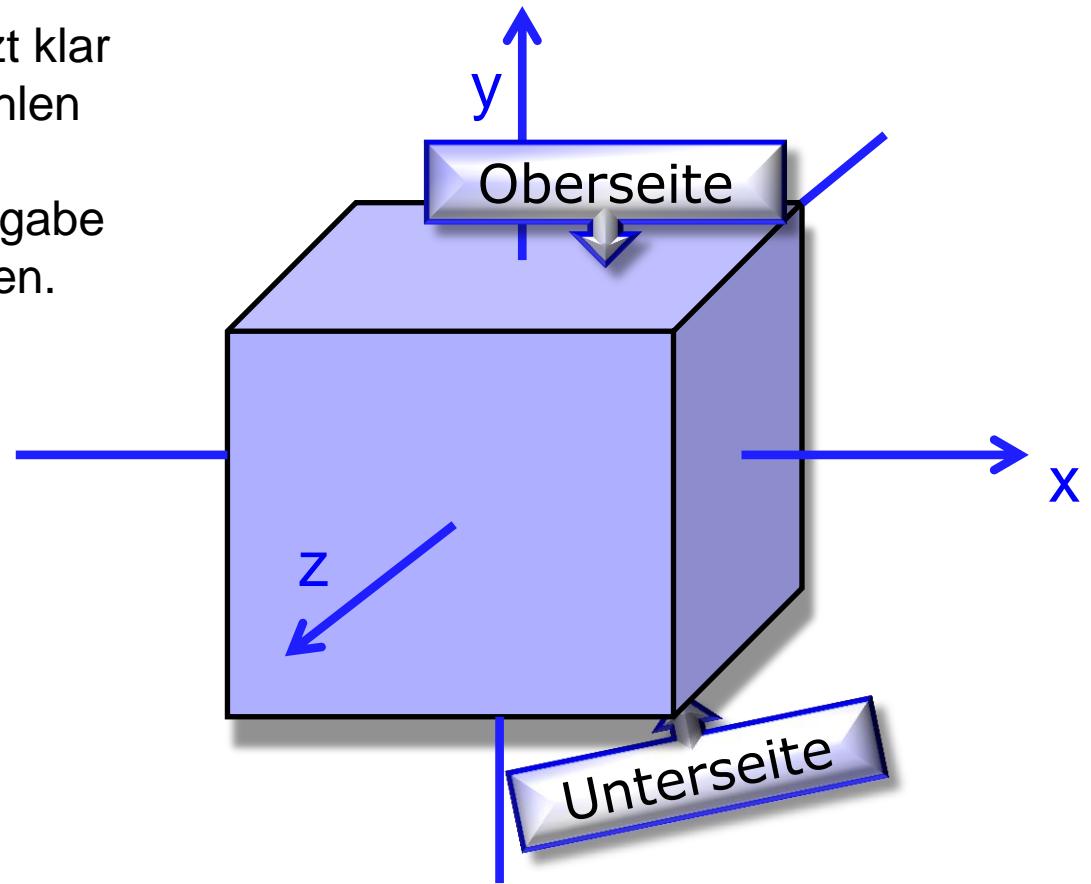
1 // /

2 // /

3 // /

4 // /

5 // /





Triangle Lists - Einheitswürfel mit Triangle Lists

Initialisierung der Triangle List

1 // / / / Nachdem Ober- und Unterseite erstellt wurden, kommt der letzte Befehl im Konstruktor. Mit `CTriangleList::Init` werden die Polygondaten an die Grafikkarte übergeben und damit eine eigene Geometrie erzeugt:

2 // / / / `CTriangleList::Init();`

Die Klasse kann nun so verwendet werden, als wäre sie eine normale Geometrie, man inkludiert sie und schreibt im Header:

3 // / / / `CGeounitCube m_zgUnitCube;`

4 // / / / Die Variable `m_zgUnitCube` muss natürlich noch an ein geeignetes Placement geadded werden und eventuell per SetMaterial mit einem Material versehen werden:

5 // / / / `m_zp.AddGeo(&m_zgUnitCube);
m_zgUnitCube.SetMaterial(pmaterail);`



Triangle Lists

Eigene Pyramide mit Triangle Lists (1/5)

```
m_avertex[3].Init(          // Vertex 3 von Seite Z+:  
    CHVector(-1.0f, -1.0f, +1.0f), // Position  
    CHVector( 0.0f,  0.0f,  1.0f), // Normale  
    CHVector(-1.0f,  0.0f,  0.0f), // Tangente  
    1.0f,  0.0f);                // UV-Koordinaten  
  
// Jetzt haben wir die 4 Vertices definiert,  
// wir müssen nun noch die Vertices zu Dreiecke vernetzen:  
  
// 1. Dreieck auf der Z+-Seite:  
AddVertex(&m_avertex[0]);  
AddVertex(&m_avertex[1]);  
AddVertex(&m_avertex[2]);  
// 2. Dreieck auf der Z+-Seite:  
AddVertex(&m_avertex[3]);  
AddVertex(&m_avertex[2]);  
AddVertex(&m_avertex[1]);
```





Triangle Lists

Eigene Pyramide mit Triangle Lists (1/5)

1 // 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);
```

2 // Die Richtungen der Oberflächennormalen der 4 Polygone ausrechnen (für das Shading wichtig):

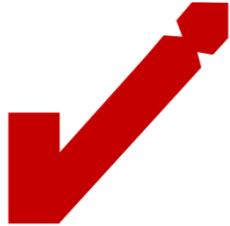
```
CHVector v021F = v0+v2+v1;  
CHVector v301F = v3+v0+v1;  
CHVector v203F = v2+v0+v3;  
CHVector v123F = v1+v2+v3;
```

3 //

4 //

5 //





Triangle Lists

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();





Triangle Lists

Eigene Pyramide mit Triangle Lists (5/5)

1 // / / /
2 // / / /
3 // / / /
4 // / / /
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)

1 //

```
// 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);
```

2 //

```
// Die 12 Vertices der Triangle List hinzufügen:
```

```
for(int i=0;i<12;i++)  
    m_trianglelist.Addvertex(&m_avertex[i]);
```

3 //

```
// 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);
```

Triangle Lists

Ergebnis der eigenen Pyramide



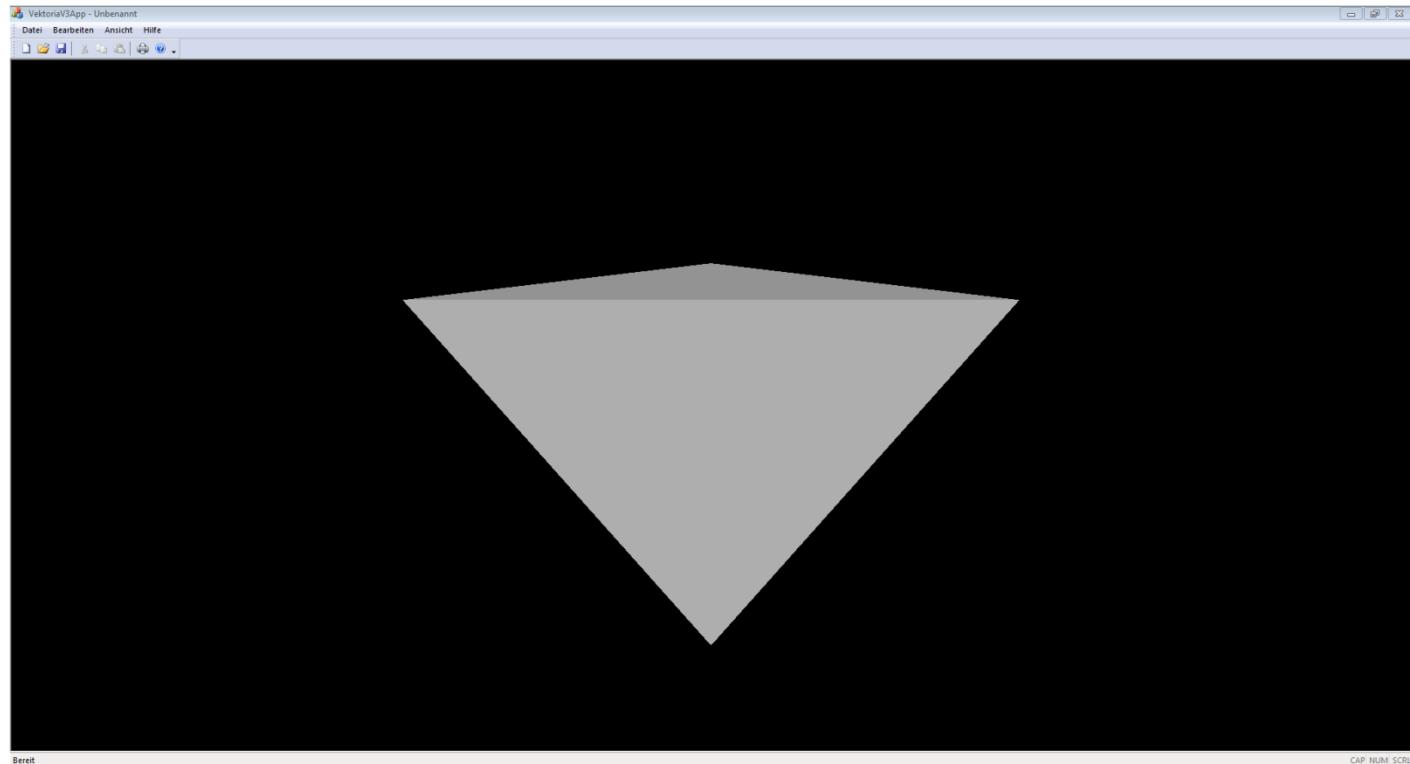
1 // / / /

2 // / / /

3 // / / /

4 // / / /

5 // / / /



Kapitel 3

Kapitel 3

1 // / / /

2 // / / /

/// **TRIANGLE STRIPS** //



4 // / / /

5 // / / /



Triangle Strips

Triangle Strips

kann sein

Geometrie
CGeo

kann sein

Triangle Lists
CTriangleList

Triangle Strips
CTriangleStrips

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





Triangle Strips

Quad durch Triangle Strips

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

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





Triangle Strips

Eigenen Quad mit Triangle Strips (1/4)

1 //
// Im Header der Game-Klasse brauchen wir eine
// Instanz eines Dreiecks-Strips:

CTriangleStrip m_trianglestrip;

2 //
// In der CGame::Init() erfolgt die Erzeugung der
// Geometrie:

3 //
// Erst die 4 Vertices initialisieren:
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
);





Triangle Strips

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  
);
```



Triangle Strips

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_trianglestrip.AddVertex(&m_avertex[i]);  
    m_trianglestrip.AddIndex(i);  
}
```





Triangle Strips

Eigenen Quad mit Triangle Strips (4/4)

1 // Initialisiere den Triangle Strip:

```
m_trianglestrip.Init();
```

2 // Mappe Material auf das Rechteck:

```
m_trianglestrip.SetMaterial(&m_material);
```

3 // und nun kannst Du Deinen eigenen persönlichen
Triangle Strip benutzen, wie eine ganz normale
Geometrie:

```
m_placement.AddGeo(&m_trianglestrip);
```

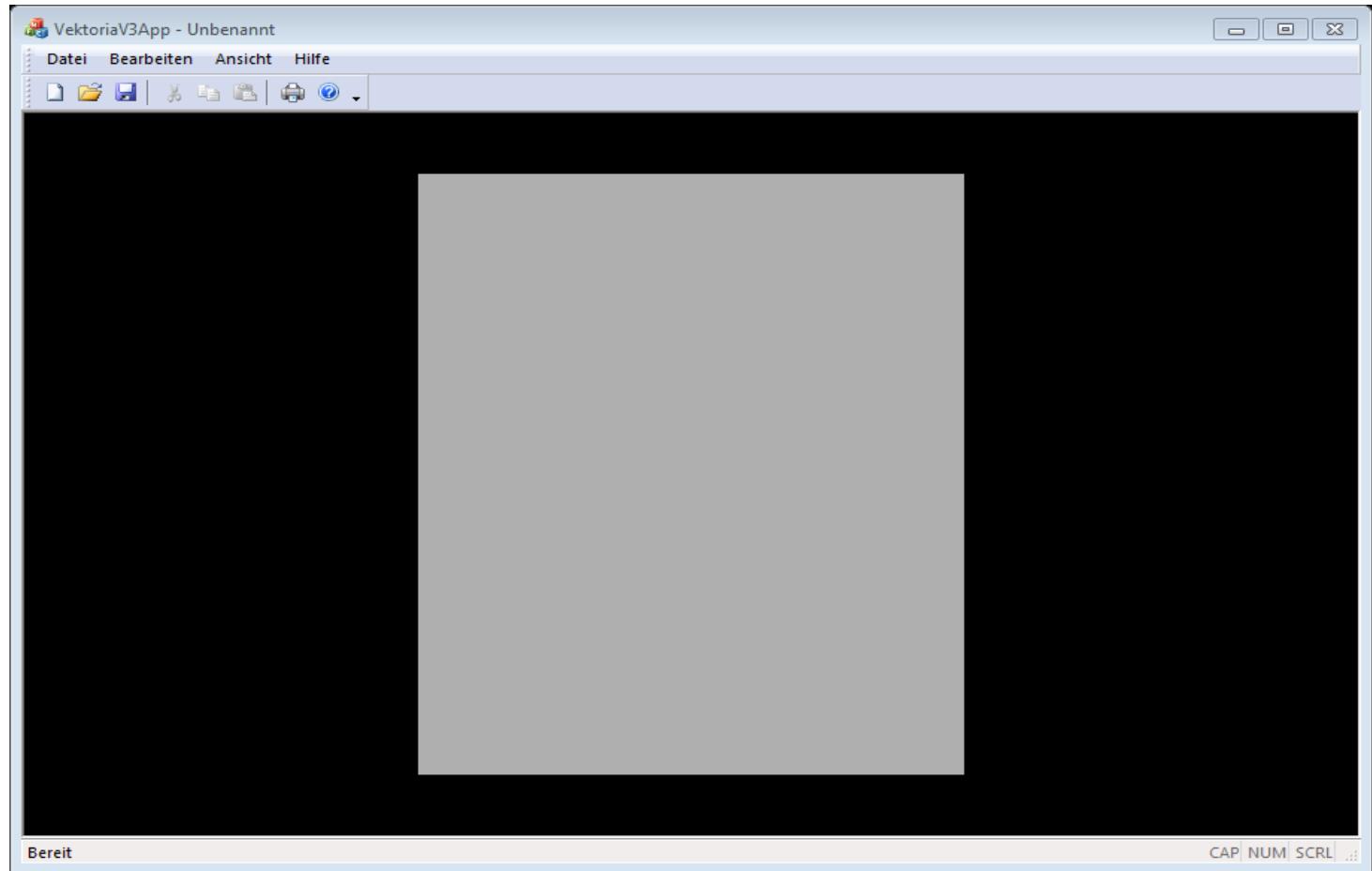
4 //

5 //



Triangle Strips

Ergebnis: Eigener Quad





Triangle Strips Übung



Erzeugen Sie einen Pfeiler mit Hilfe von Triangle Strips!

Freiwillige Zusatzaufgabe für die schnellen Nerds:

Erzeugen Sie eine Prozedur, die parametrisiert dorische und ionische Pfeiler erzeugt!



Kapitel 4

Kapitel 4

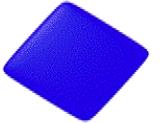


1 // / / /

2 // / / /

3 // / / /

/// IMPORTER



5 // / / /



PROF. DR. TOBIAS BREINER
VEKTORIA MANUAL

51 VON 58
EIGENE GEOMETRIEN





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!

1 // / / / In der Game.h:

```
CGeo * m_pzg;
```

```
CFileBlender m_fileBlender;
```

2 // / / / In der Game.cpp / Init-Methode:

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





Importer

Wavefront-Import

1 // In der Game.h:

```
CGeo * m_pzg;  
CFilewavefront m_filewavefront;
```

2 // In der Game.cpp / Init-Methode:

```
m_pzg = m_filewavefront.LoadGeo("myfile.obj");
```



3 // Importieren Sie mit Blender oder Maya erstellte 3D-Geometrien am besten mit diesem Importer, das bringt die besten Ergebnisse!





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:

5 // / / / m_pzg = m_file3ds.LoadGeo("myfile.3ds");



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

Kapitel 5



1 // / / /

2 // / / /

3 // / / /

4 // / / /

/// ALLGEMEINES



PROF. DR. TOBIAS BREINER
VEKTORIA MANUAL

56 VON 58
EIGENE GEOMETRIEN

Tipps



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

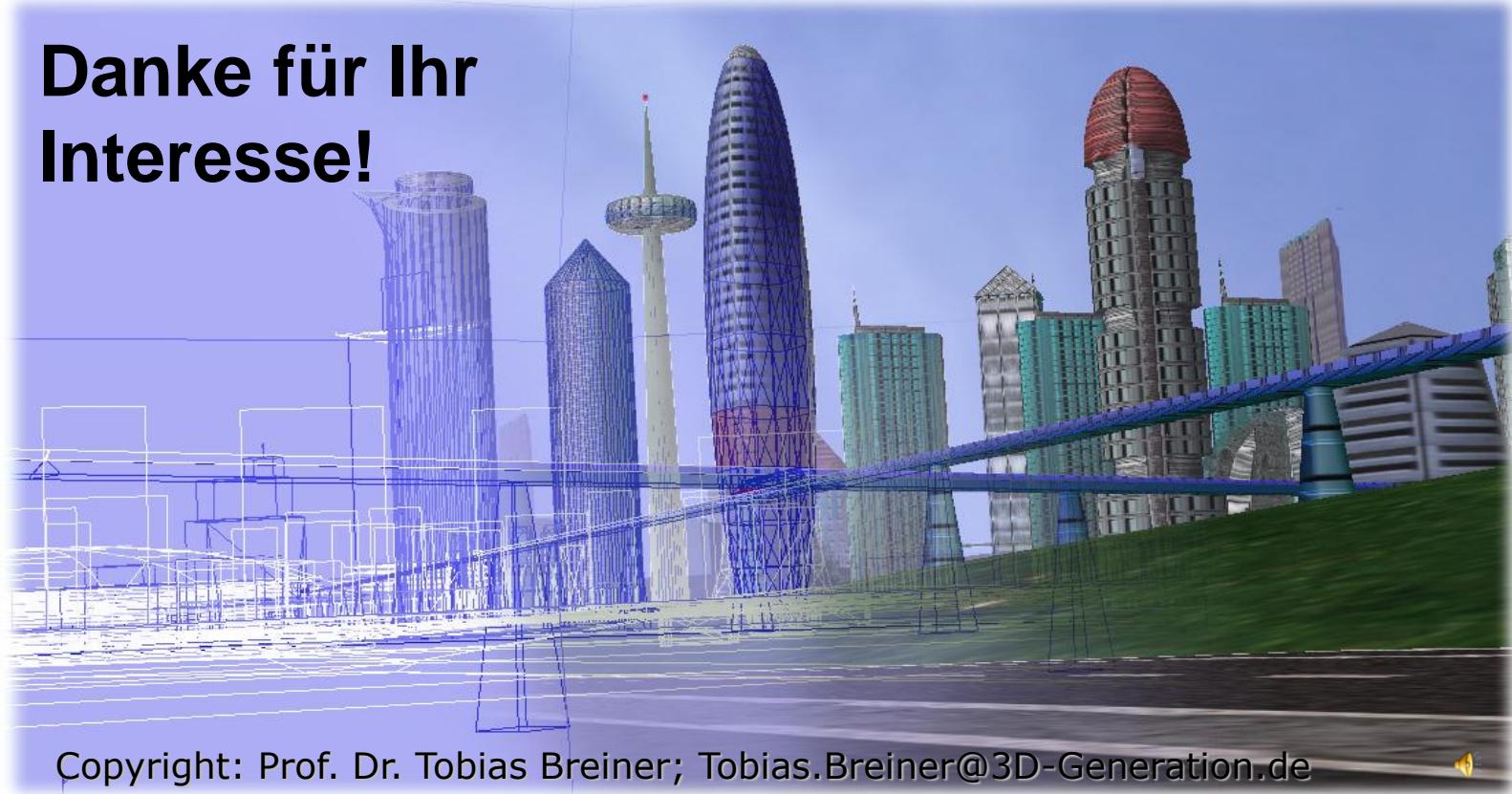
Die Gründe dafür sind:

- 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!



Copyright: Prof. Dr. Tobias Breiner; Tobias.Breiner@3D-Generation.de

