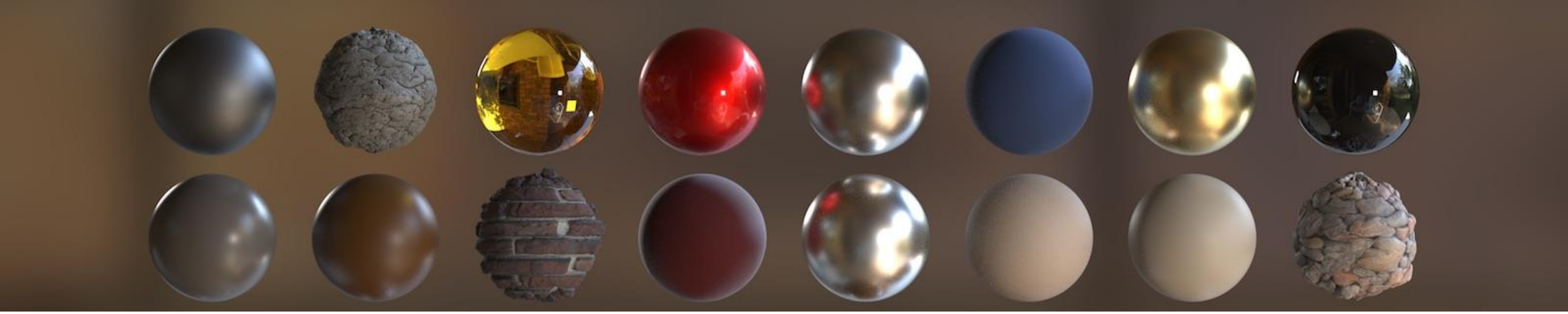
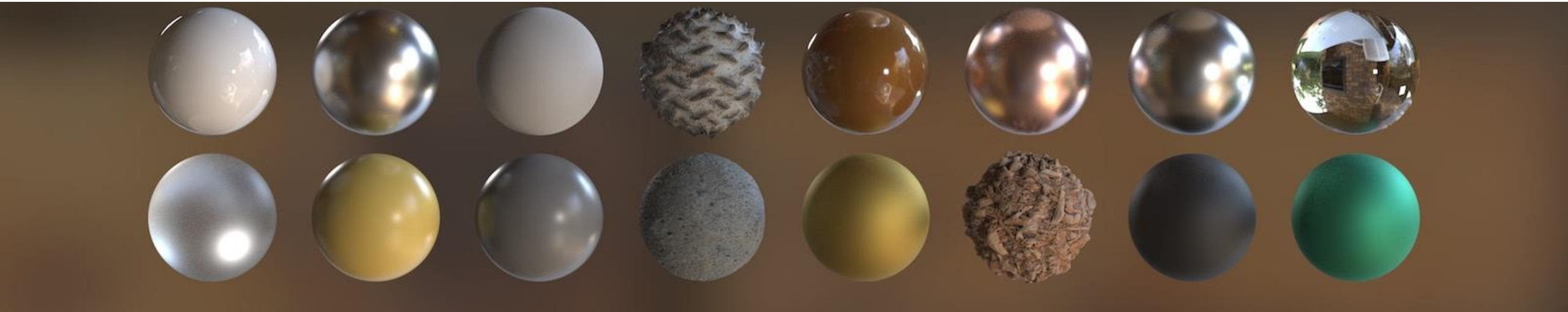
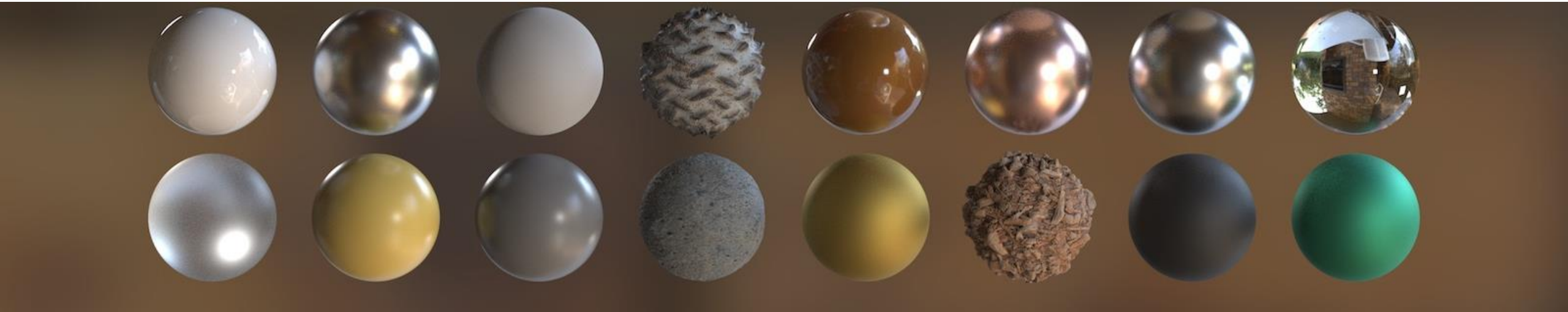


Shader Praktikum

FH Wedel



Shader – Aufgabe 2

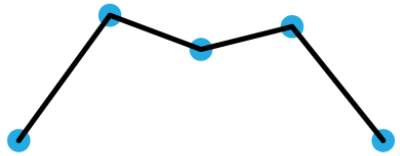


Themen

- Displacement
  - Normal Mapping
  - Parallax Mapping
- Tessellation
- Geometry Shader

Displacement Mapping

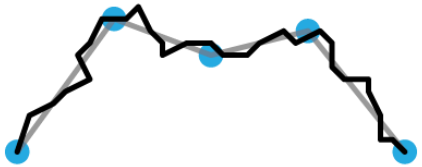
Technik, um die Meso-Struktur nachträglich **Vertex**- oder **Pixel**-basiert herzustellen



Makro-Struktur

Primitive des Meshes

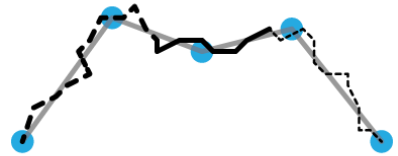
Grobe Strukturen



Meso-Struktur

Displacement Mapping

Kleine aber klar erkennbare Strukturen



Mikro-Struktur

BRDFs

Strukturen mit Auge nicht/kaum zu unterscheiden

Displacement Mapping

Pixel-Basiert

Keine Veränderung der Modellgeometrie sondern Nutzung von Oberflächeninformationen (aus Texturen) auf Pixelebene um die Illusion von zusätzlichen Geometriedetails zu erzeuge.

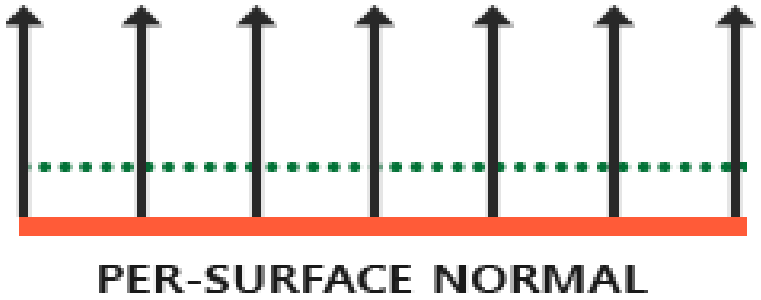
Vertex-Basiert

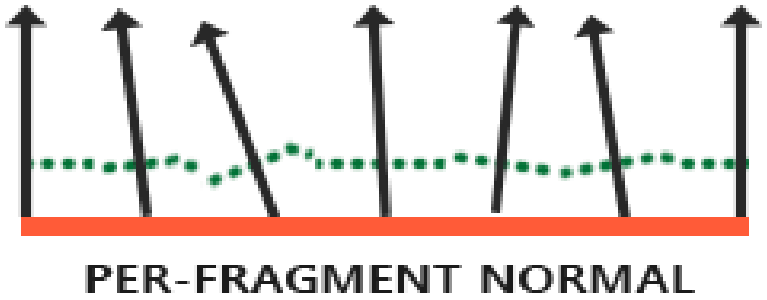
Veränderung der Modell durch selektives Hinzufügen oder Positionieren von Vertices zur Erzeugung zusätzlicher Geometriedetails

Pixel-basiertes Displacement Mapping
Normal Mapping

Wahrgenommene Oberfläche

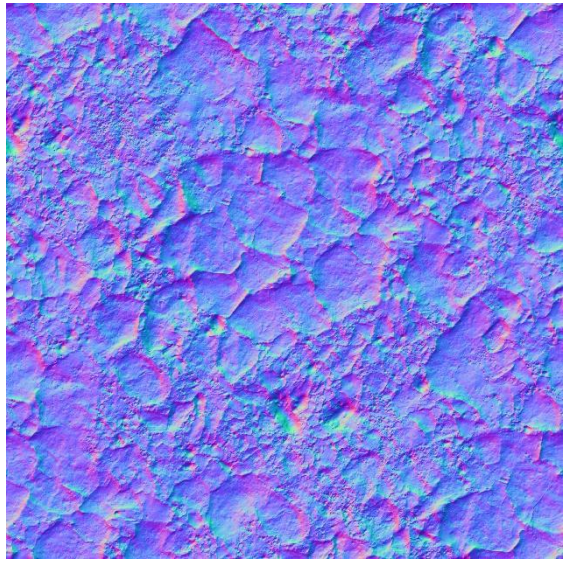
Eigentliche Oberfläche



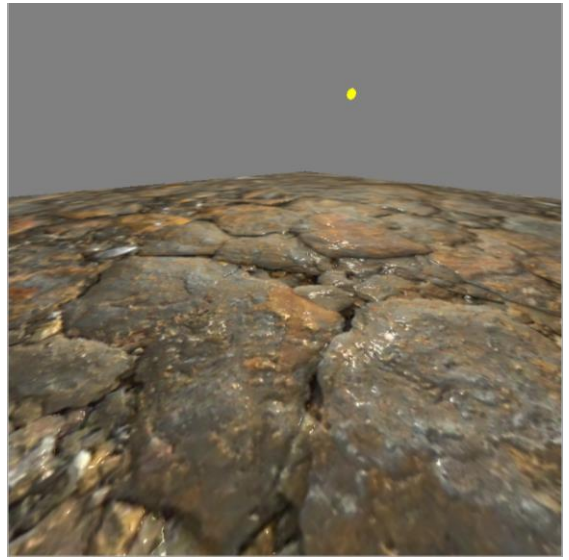


Pixel-basiertes Displacement Mapping **Normal Mapping**

Die Normalen der Oberfläche werden als Farbe in der Normal Map kodiert.



Normal Map



Normal Mapping

Pixel-basiertes Displacement Mapping **Normal Mapping**

Kodierung

Bei der Kodierung der Normalen als Farbe wird berücksichtigt, dass die die Komponenten einer *Normalen auf allen Achsen im Intervall [-1,1]* liegen können. Da *Farben typischerweise im Intervall [0,1]* dargestellt werden muss die folgende Abbildung erfolgen:

$$\begin{pmatrix} c_r \\ c_g \\ c_b \end{pmatrix} = 0.5 \begin{pmatrix} n_x \\ n_y \\ n_z \end{pmatrix} + \begin{pmatrix} 0.5 \\ 0.5 \\ 0.5 \end{pmatrix}$$

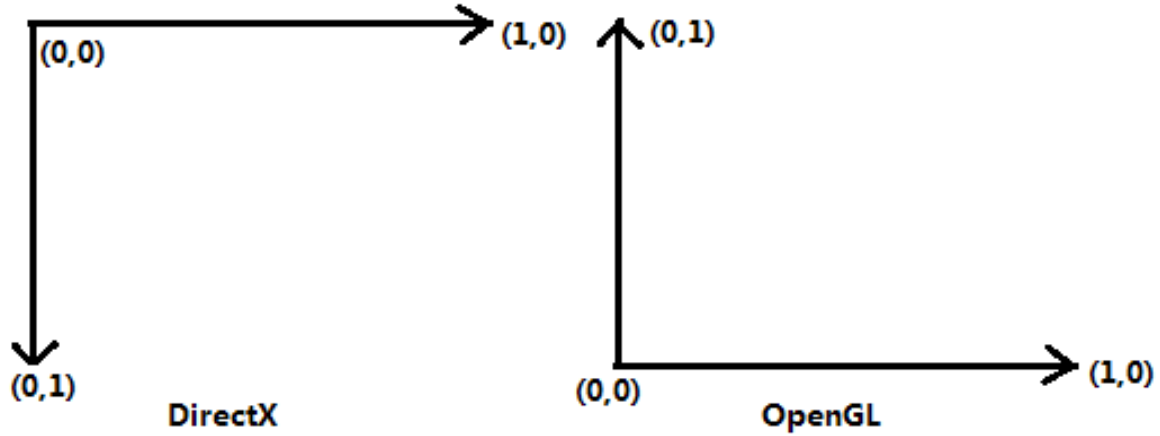
c = Farbe der Normal Map
n = Normale

Pixel-basiertes Displacement Mapping **Normal Mapping**

Nullpunkt

Beim Dekodieren der Normalen ist die *Wahl des Nullpunktes* beim *Baking* zu berücksichtigen. Bei der Verwendung von OpenGL wird der Nullpunkt typischerweise in die untere linke Ecke gelegt. Die linke obere Ecke ist für DirectX typisch.

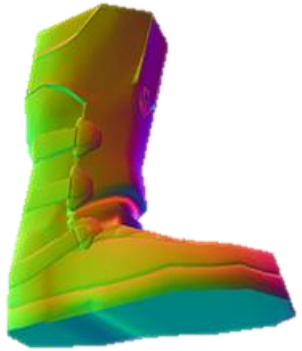
Beim Verwenden von *Normal Maps aus dem Internet* ist es wahrscheinlich, dass diese ihren Nullpunkt oben links haben. Somit sind die Normalen auf der Y-Achse des Tangent Space gespiegelt.



Pixel-basiertes Displacement Mapping
Normal Mapping

Bezugssystem

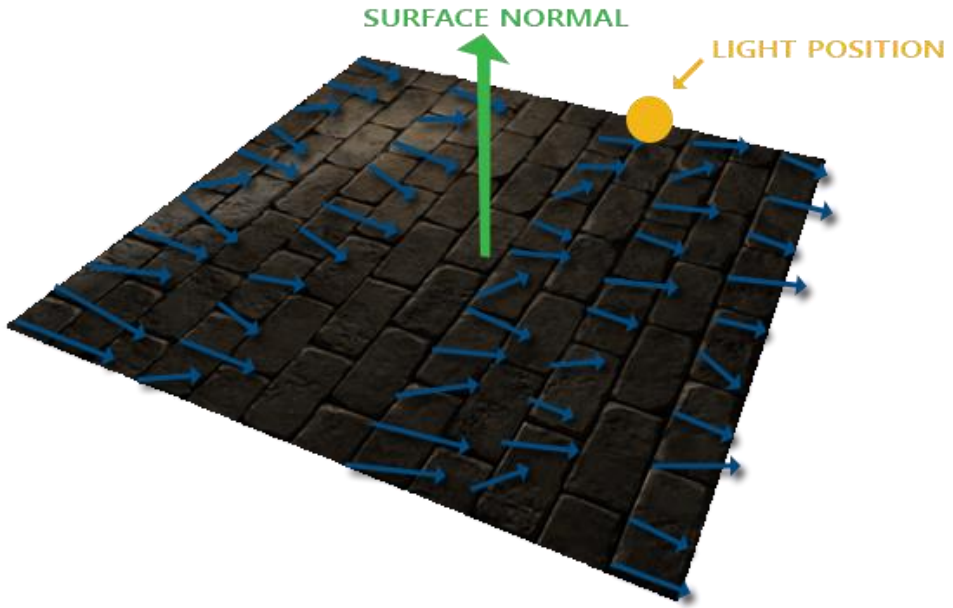
Beim *Baking* der Normal Map wird festgelegt, in welchem Koordinatensystem die Normale interpretiert werden muss.



Model Space: Bezüglich des Modell-Koordinatensystems

Tangent Space: Bezüglich eines Koordinatensystems, das direkt von dem Texturkoordinatensystem abhängig i

Pixel-basiertes Displacement Mapping **Normal Mapping**



https://learnopengl.com/Advanced-Lighting/Normal-Mapping

Pixel-basiertes Displacement Mapping **Normal Mapping**

Tangent Space

Koordinatensystem für die Oberfläche von Modellen. Der Tangent Space ermöglicht *die Definition von Vektoren relativ zur Oberfläche eines Modells*.

Hierbei werden die *Basisvektoren der Koordinatensysteme* gespeichert.

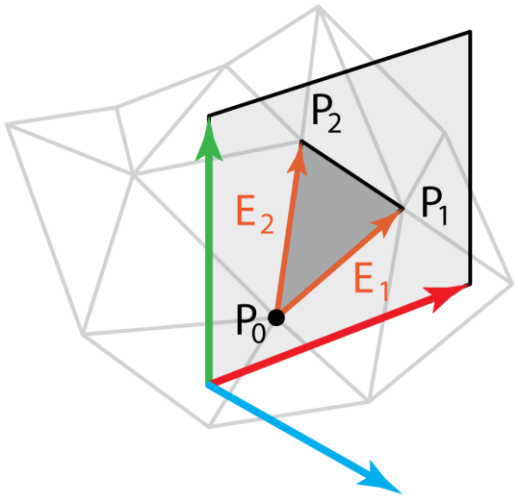
Die Basisvektoren werden pro Vertex gespeichert und können für die Punkte auf der Oberfläche *zwischen den Vertices von der Pipeline interpoliert* werden.

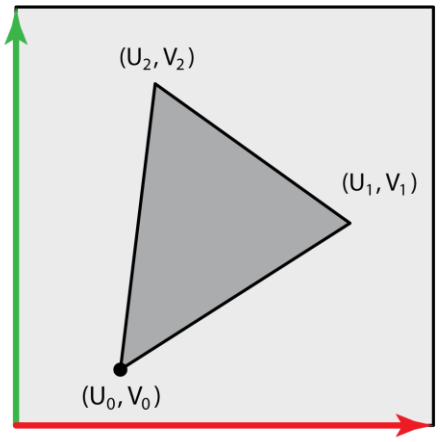
Pixel-basiertes Displacement Mapping **Normal Mapping**

Berechnung des Tangent Space

3 Basisvektoren

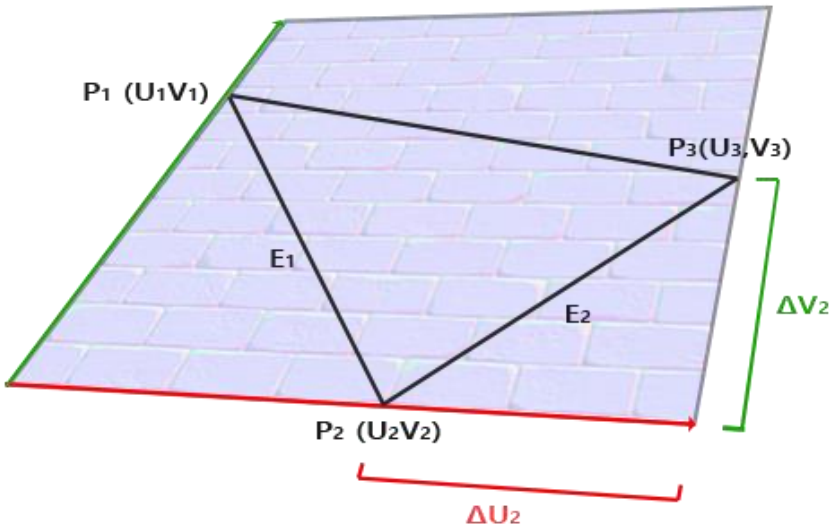
- Normale (im Modell Koordinatensystem)
- 2 Basisvektoren des Texturkoordinatensystem (im Modell Koordinatensystem)





Pixel-basiertes Displacement Mapping
Normal Mapping

Berechnung des Tangent Space für Vertex P_0



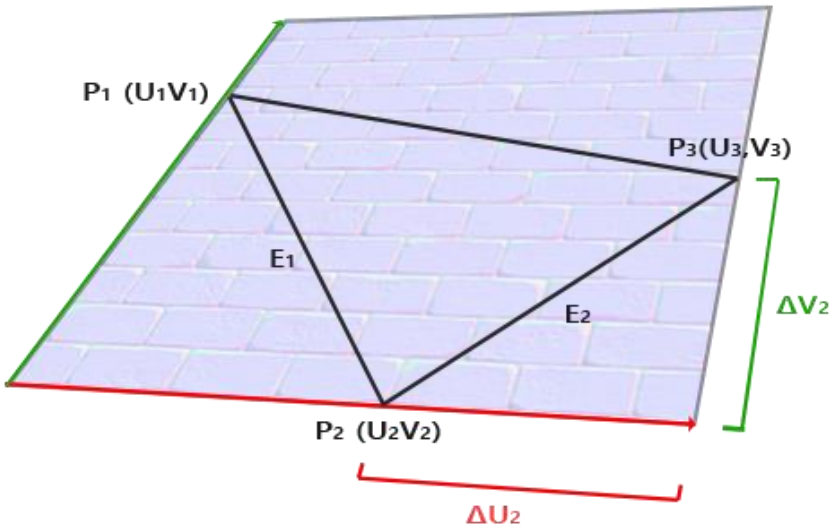
$E_1 = P_1 - P_2$
$E_2 = P_3 - P_2$

Linearkombination der Tangente T und Bitangente B
$E_1 = (U_1 - U_2)\,T + (V_1 - V_2)\,B$
$E_2 = (U_3 - U_2)\,T + (V_3 - V_2)\,B$

Was ist Unbekannt?

Pixel-basiertes Displacement Mapping
Normal Mapping

Berechnung des Tangent Space für Vertex P_0



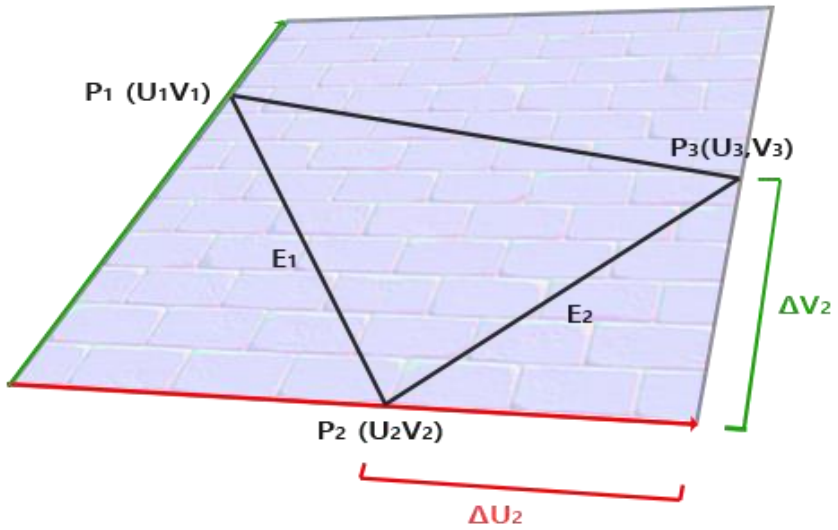
Matrixform

$$\begin{pmatrix} E_1.x & E_1.y & E_1.z \\ E_2.x & E_2.y & E_2.z \end{pmatrix} = \begin{pmatrix} U_1 - U_2 & V_1 - V_2 \\ U_3 - U_2 & V_3 - V_2 \end{pmatrix} \begin{pmatrix} T.x & T.y & T.z \\ B.x & B.y & B.z \end{pmatrix}$$

$$\begin{pmatrix} T.x & T.y & T.z \\ B.x & B.y & B.z \end{pmatrix} = \begin{pmatrix} U_1 - U_2 & V_1 - V_2 \\ U_3 - U_2 & V_3 - V_2 \end{pmatrix}^{-1} \begin{pmatrix} E_1.x & E_1.y & E_1.z \\ E_2.x & E_2.y & E_2.z \end{pmatrix}$$

Pixel-basiertes Displacement Mapping
Normal Mapping

Berechnung des Tangent Space für Vertex P_0



Determinante

$$d = (U_1 - U_2) * (V_3 - V_2) - (V_1 - V_2) * (U_3 - U_2)$$

Inverse Matrix

$$\begin{pmatrix} U_1 - U_2 & V_1 - V_2 \\ U_3 - U_2 & V_3 - V_2 \end{pmatrix}^{-1} = \frac{1}{d}\begin{pmatrix} V_3 - V_2 & -(V_1 - V_2) \\ -(U_3 - U_2) & U_1 - U_2 \end{pmatrix}$$

Pixel-basiertes Displacement Mapping **Normal Mapping**

Die Tangente und Bitangente müssen, wie auch die Normale, beim Rendern pro Vertex bereitstehen.

Es gibt drei Möglichkeiten dies zu erreichen:

- *Speichern der Tangente und Bitangente* in einem oder zwei Buffer Objects und das Einrichten zweier Attribute Pointers
- *Speichern der Tangente* in einem Buffer Object, das Einrichten nur eines Attribut Pointers und das Berechnen der Bitangenten im Shader mit einem Kreuzprodukt zwischen Normale und Tangente
- *Speichern der Bitangenten* in einem Buffer Object, das Einrichten nur eines Attribut Pointers und das Berechnen der Tangente im Shader mit einem Kreuzprodukt zwischen Normale und Tangente

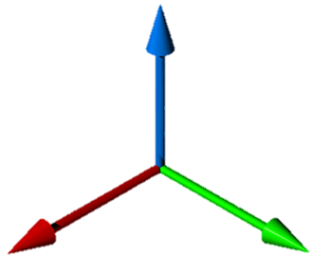
Pixel-basiertes Displacement Mapping **Normal Mapping**

Anwendung des Tangent Space

Zur Transformation von Vektoren aus dem Tangent Space in den Modelspace wird die TBN-Matrix (Tangente, Bitangente, Normale) aufgestellt:

$$\text{TBN} = \begin{pmatrix} T.x & B.x & N.x \\ T.y & B.y & N.y \\ T.z & B.z & N.z \end{pmatrix}$$

Orthogonale Matrix!



$$\begin{pmatrix} T.x & B.x & N.x \\ T.y & B.y & N.y \\ T.z & B.z & N.z \end{pmatrix}^{-1} = \begin{pmatrix} T.x & B.x & N.x \\ T.y & B.y & N.y \\ T.z & B.z & N.z \end{pmatrix}^{T} = \begin{pmatrix} T.x & T.y & T.z \\ B.x & B.y & B.z \\ N.x & N.y & N.z \end{pmatrix}$$

Inverse Matrix = Transponierte Matrix

Pixel-basiertes Displacement Mapping **Normal Mapping**

Beleuchtungsberechnung im Tangent Space

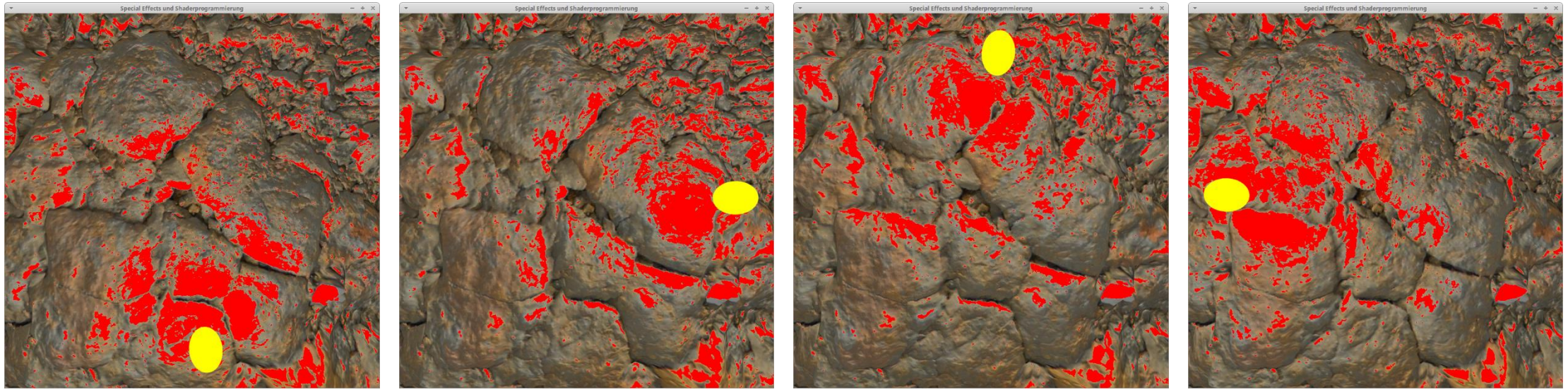
- Transformation der View- & Light-Vektoren im Vertex-Shader in den Tangent Space
- keine Transformation der Normalen nach dem Auslesen aus der Normal-Map (im Fragment Shader)
- (sinnvoller (Performance) Ansatz, da generell der VS weniger häufig ausgeführt wird als der FS)

Beleuchtungsberechnung außerhalb Tangent Space

- TBN**-**Matrix an Fragment Shader
- Transformation der Normalen aus der Normal-Map in das Modellkoordinantensystem
- (generell schlechtere Performance [FS wird häufiger ausgeführt], jedoch weniger Änderungen für die Beleuchtungsberechnung)

Pixel-basiertes Displacement Mapping **Normal Mapping**

Um die Transformation der Normalen zu überprüfen, solltet ihr die Kamera über der Oberfläche positionieren und die Lichtquelle bewegen. Achtet dabei darauf, dass die Highlights auf der korrekten Seite der Steine zu sehen sind.



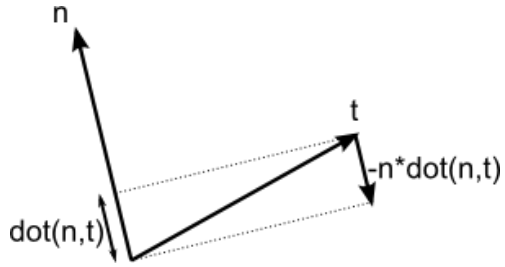
Auf den Bildern sind die Highlights rot eingefärbt.

Pixel-basiertes Displacement Mapping **Normal Mapping**

Orthogonalisieren

Falls die Matrix nicht Orthogonal ist (Transposierte == Inverse, wenn orthogonal):

Korrektur der Tangente:



*t = normalize(t-n * dot(n,t));*

Handedness

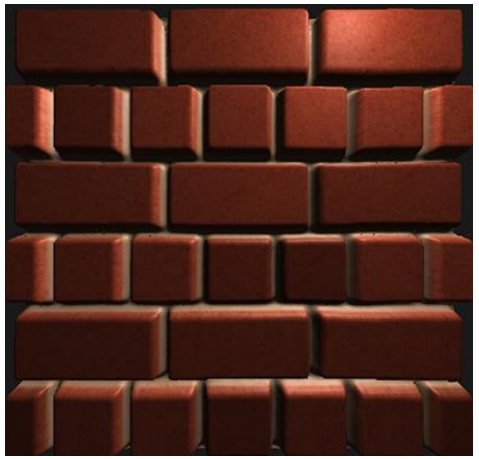
Bei der Nutzung von symmetrischen Model, kann es passieren, dass die UVs falsch orientiert sind (die Tangente hat die falsche Richtung). Um zu prüfen, ob T invertiert werden muss, muss gecheckt werden ob die TBN Matrix ein rechts-händiges Koordinaten System bildet.

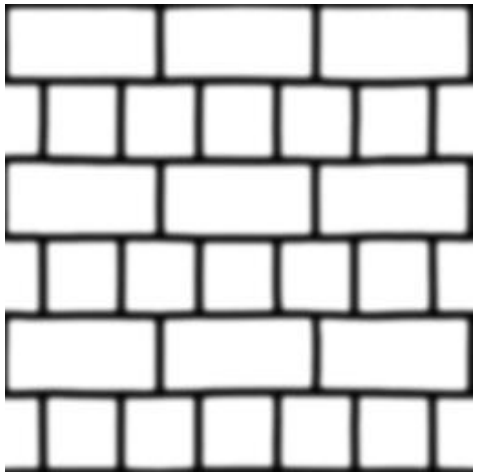
if(dot(cross(n,t),b) < 0.0)

*t = t * - 1.0*

Pixel-basiertes Displacement Mapping **Parallax Mapping**

- Technik die Oberflächen Details enorm erhöht
  - Im Fragment Shader (wie Normalmapping)
- In Kombination mit Normalmapping sehr realistische Ergebnisse
- Nutzung einer Höhentextur
- Displacement ohne zusätzliche Vertices

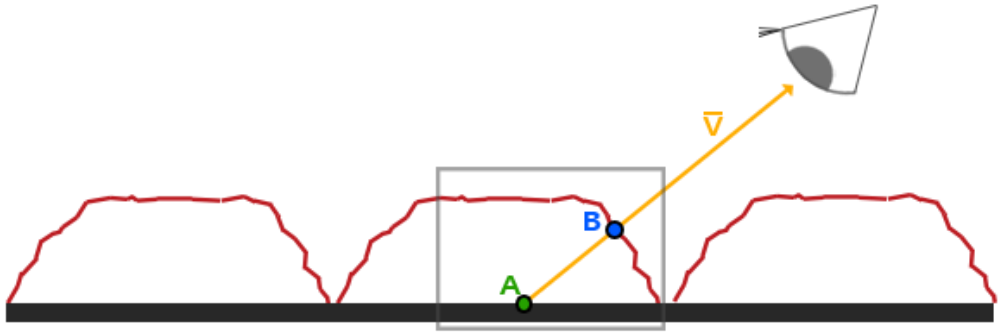




Quelle: https://learnopengl.com/Advanced-Lighting/Parallax-Mapping

Parallax Mapping

- Verschiebung der **Textur Koordinate**, sodass es aussieht, als wäre die Fragment Oberfläche höher oder tiefer
- Basierend auf der Blickrichtung und der Höhentextur
- Sampeln der Textur für Position B

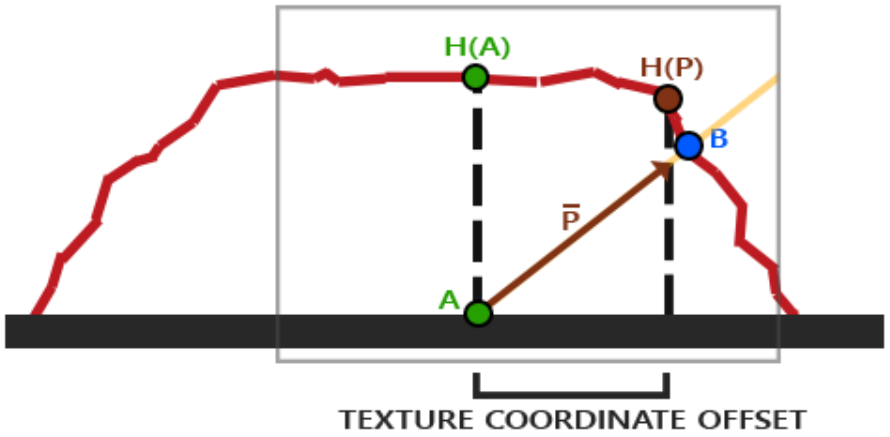


Rote Linie – Werte der Höhentextur
$\vec{V}$- Vektor von der Oberfläche zum Augpunkt (inv. Blickrichtung)
A – Fragment Position
B – Punkt der beim Deplacement gesehen wird

Quelle: https://learnopengl.com/Advanced-Lighting/Parallax-Mapping

Parallax Mapping

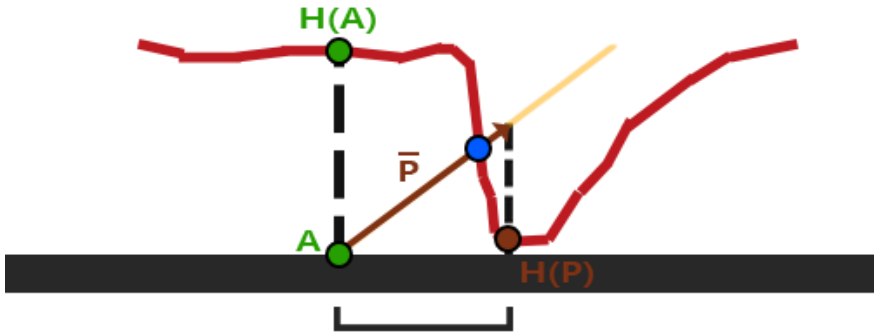
- Skalierung ($\vec{P}$) des Fragement-to-View Vektors ($\vec{V}$) mithilfe der Höhe an A
- Nutzung der Vector Koordinaten von $\vec{P}$ die auf der Plane liegen als Textur Koordinaten Offset
- Um die Koordinaten zu bestimmen wird die Transformation in einem anderen *Koordinaten System* durchgeführt. Welches?



Quelle: https://learnopengl.com/Advanced-Lighting/Parallax-Mapping

Parallax Mapping

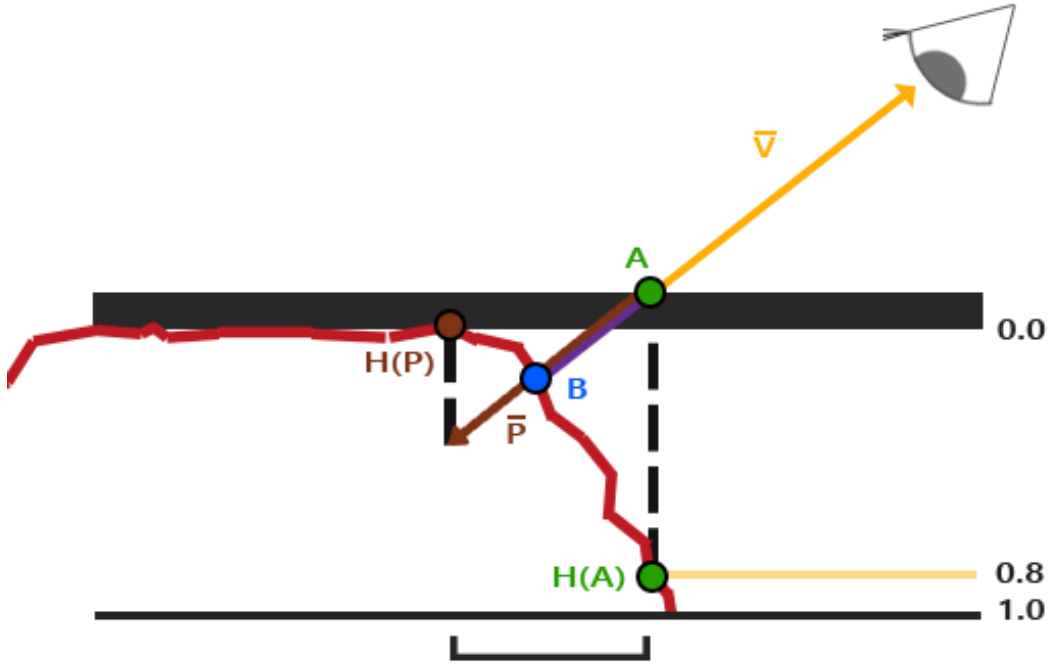
- Achtung: wenn sich die Höhe sprunghaft ändert in der Höhentextur, kann dies unrealistisch wirken.
  (Da $\vec{P}$ nicht nahe von B endet!)
- Achtung: je nach Textur Wrapping, kann es zu Artefakten am Randbereich führen (Oversampling außerhalb von [0,1])



Quelle: https://learnopengl.com/Advanced-Lighting/Parallax-Mapping

Parallax Mapping

- Oft verwendet Depthmaps (inverse Heightmaps) anstelle von Heightmaps, da die Tiefen Illusion einfacher zu erstellen ist
- Dadurch ändert sich etwas die Berechnung!
- Tiefenwerte können erhalten werden durch: 1 - Höhenwert

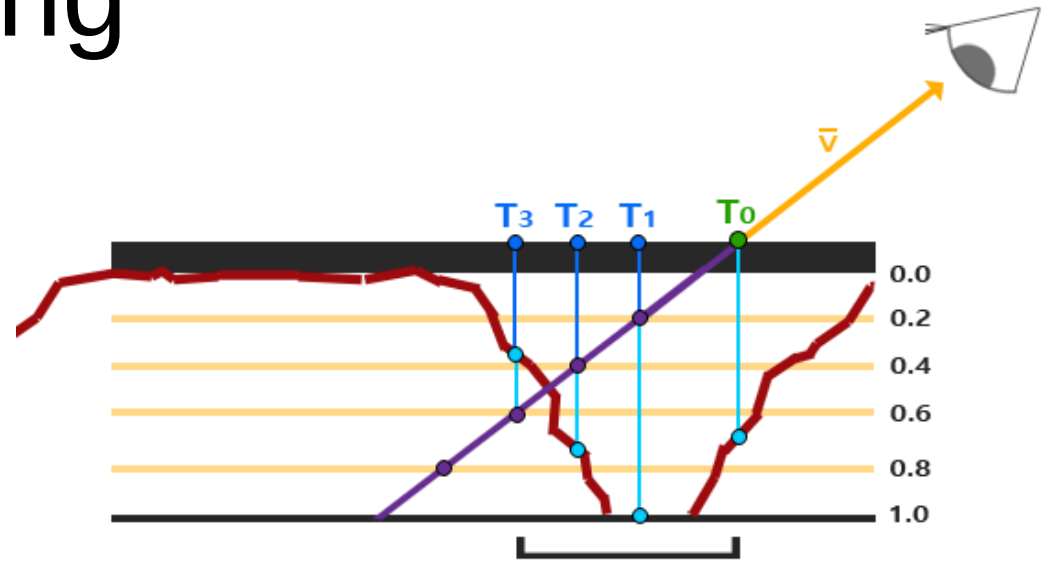


Probleme beim Nativen Ansatz

Quelle: https://learnopengl.com/Advanced-Lighting/Parallax-Mapping

Steep Parallax Mapping

- Nutzung mehrerer Textur Samples
- Teilung der Tiefe in verschiedene Ebenen (Layer)
- Für jede Ebene wird die Depthmap gesampelt (unter Versetzung der Textur Koordinate)
  - Bis ein Tiefen Wert unter dem der Aktuellen Ebene gefunden wird
- Nachteil: es kann zu aliasing Effekten führen



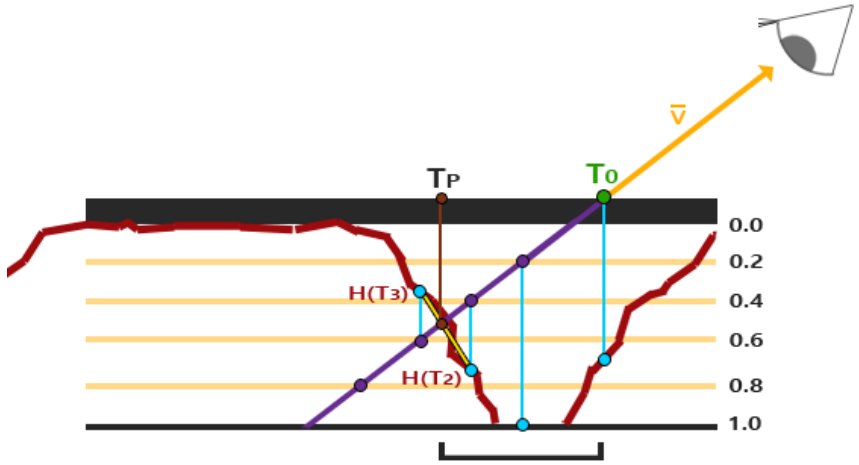
Quelle: https://learnopengl.com/Advanced-Lighting/Parallax-Mapping

Parallax Mapping
Steep Parallax Mapping Algorithmus

```
sLayer = 1 / numLayer;  // Größe der Ebene ergibt sich aus 1 durch die Anzahl
// aktuellen Tiefe der Ebene / Texturkoordinate & TiefenmapWert initialisieren
currLayerDepth = 0; currTexCoord = texCoord;
currDepthMapV = sample(map, currTexCoord).r;


P = viewDir.xy * heightScale; // Texturkoordinaten Offset pro Ebene (vom P Vektor)

texCoordOff = P / numLayer;

while(aktuelle Tiefe grösser ist als aktueller Sample Wert)

   currTexCoord -= texCoordOff;            // Versetzen der Texturkoordinate entlang P
   currDepthMapV = sample(map, currTexCoord).r;        // samplen der Depthmap
   currLayerDepth += sLayer;               // tiefe der naechsten Ebene


currenTexCoord:          // neue Texturkoordinate für das Sampeln
```

Parallax Occlusion Mapping

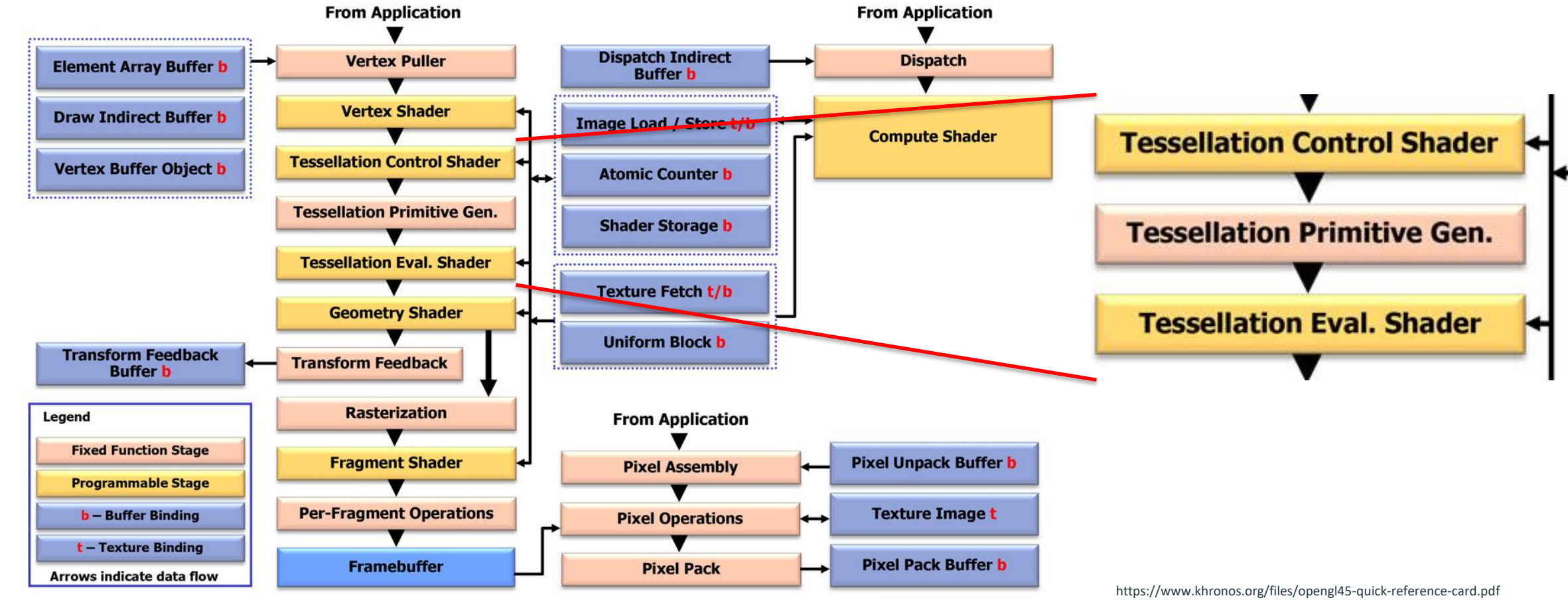
- Baut auf Steep Parallax Mapping auf
- Lineare Interpolation zwischen der Tiefenebene vor und nach der Kollision
- Gewichtung anhand dem Abstand der Oberflächen Höhe von den Tiefenwerten der beiden Ebenen



Quelle: https://learnopengl.com/Advanced-Lighting/Parallax-Mapping

Vertex-basiertes Displacement Mapping **Tessellation**

Vertex-basiertes Displacement Mapping **Tessellation**

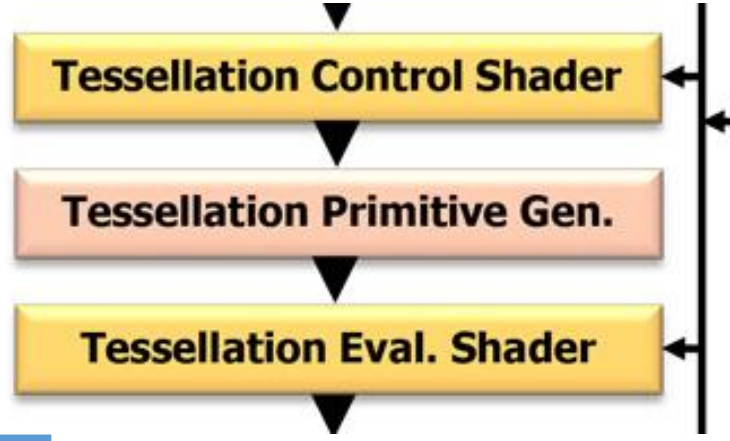


https://www.khronos.org/files/opengl45-quick-reference-card.pdf

Vertex-basiertes Displacement Mapping **Tessellation**

Pipeline

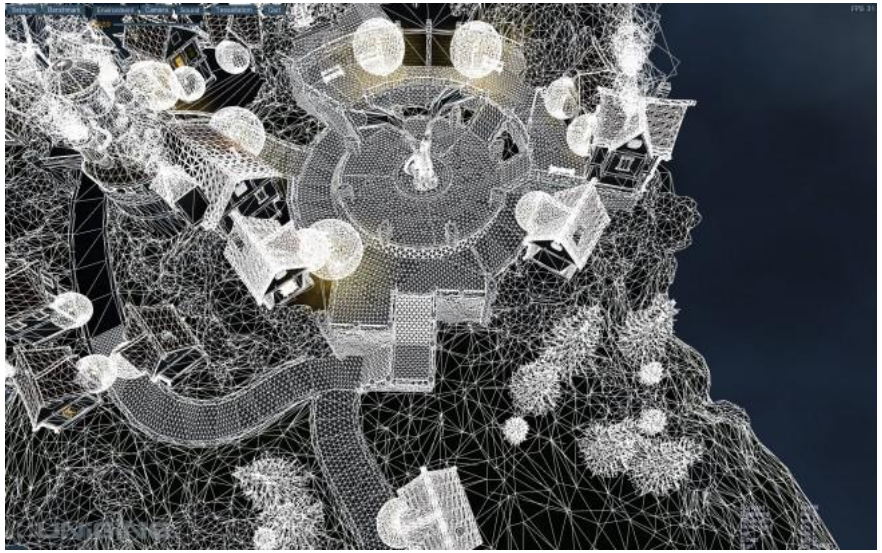
- 2 Shader- & 1 Fixed-Function Stufe; zur Unterteilung der Patches
- Tessellation ist optional, bei der Nutzung muss jedoch mindestens der *Tessellation Evaluation Shader* genutzt werden



Shaderstufe	Abkürzung	Dateiendung	DirectX
Tesselation Control Shader	TCS	.tesc	Hull Shader
Tessellation Evaluation Shader	TES	.tese	Domain Shader

Vertex-basiertes Displacement Mapping **Tessellation**

- Zur Erzeugung dynamischer Geometrien
- Es können nur zusammenhänge Geometrien erzeugen werden und die Ausgabe lässt sich *nicht* auf vers. Layer/Viewports umleiten (vergl. Geometry Shader)
- Erstellung von Geometrie auf Grundlage von Kontrollpunkten (in Patches organisiert)
- Typ. Anwendung: Betrachter abhängige Unterteilung von Dreiecken



Vertex-basiertes Displacement Mapping **Tessellation**

http://www.geforce.com/games-applications/pc-applications/Unigine-Heaven-2/screenshots

Vertex-basiertes Displacement Mapping **Tessellation**

Patches

- OpenGL erwartet beim Draw Command den Primitivtyp **GL_PATCHES** (Verwendung Tessellation Shadern)
- Vertices sind als Kontrollpunkte zu interpretieren
- Anzahl der Vertices pro Patch kann mit der Funktion glPatchParametri angepasst werden

```
// Dreiecke
glPatchParametri(GL_PATCH_VERTICES, 3);
glDrawElements(GL_PATCHES, numIndices, GL_UNSIGNED_INT, 0);
```

Vertex-basiertes Displacement Mapping **Tessellation**

Tessellation Control Shader (TCS) (optional*)

- Ermöglicht eine genauere Steuerung des Tessellation Prozesses.
- Aufgaben:
  - Erzeugung des Output Patches für den TES
  - Bestimmung der Tessellation für die nachfolgende *Tessellation Primitive Generation*
- Eigenschaft:
  - Aufruf pro Vertex des Output Patches
  - Zugriff auf alle Vertices des Input Patches
  - Legt Anzahl der Vertices pro Output Patch fest
  - Leitet Attribute an den TES weiter

* Alternativ kann ein Standardwert für die Stärke der Tessellation mit der Funktion glPatchParameter festgelegt werden.

Vertex-basiertes Displacement Mapping **Tessellation**

Tessellation Primitive Generation (TPG)

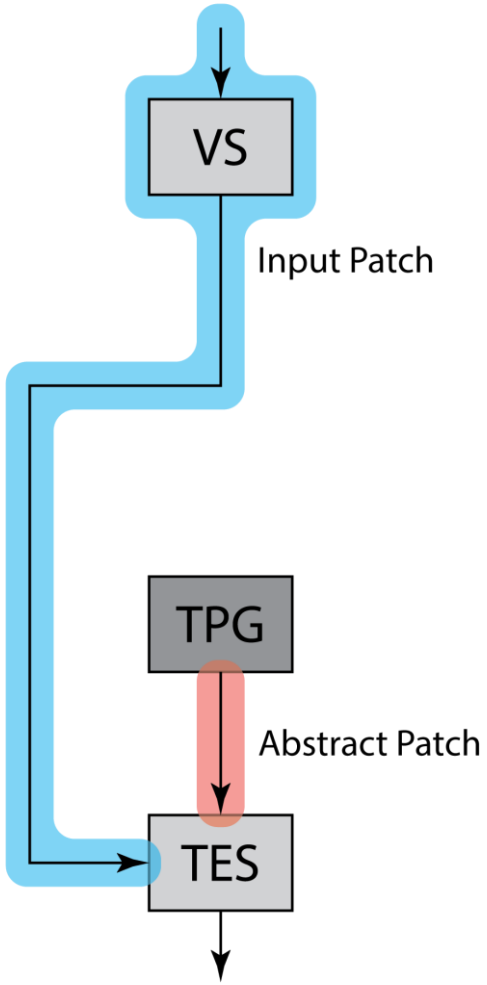
- Fixed-Function Stufe
- Erzeugt Menge von Primitiven nach einem vorgegebenen Muster
- Arbeitet auf abstrakten Patches, da dem TPG keine Vertex-Attribute zugeführt werden (d.h. die konkrete Position des Output Patches ist nicht bekannt)

Vertex-basiertes Displacement Mapping **Tessellation**

Tessellation Evaluation Shader (TES)

- Erhält Position der *neuen Vertices im Abstract Patch* auf vordef. Variable gl_TessCoord [Baryzentrische Koordinaten]
- Erhält *Attribute* des *Out Patches* vom TCS (wenn TCS inaktiv: vom *Input Patch*)
- Kann die Attribute der Vertices abhängig von der Position des Vertexs im Abstract Patch *selbst interpolieren*
- Eigenschaft:
  - Pro neuerzeugtem Vertex aufgerufen
  - Bestimmt Typ des Abstract Patches, Spacing Mode & Vertex Reihenfolge
  - Zugriff auf alle Vertices des Output Patches
  - Interpoliert die Attribute von TCS und leitet sie weiter

Vertex-basiertes Displacement Mapping **Tessellation**



Datenfluss ohne TCS

Input Patch

```
// Host: Input Patch hat 3 Vertices
glPatchParameteri(GL_PATCH_VERTICES, 3):
```

Abstract Patch

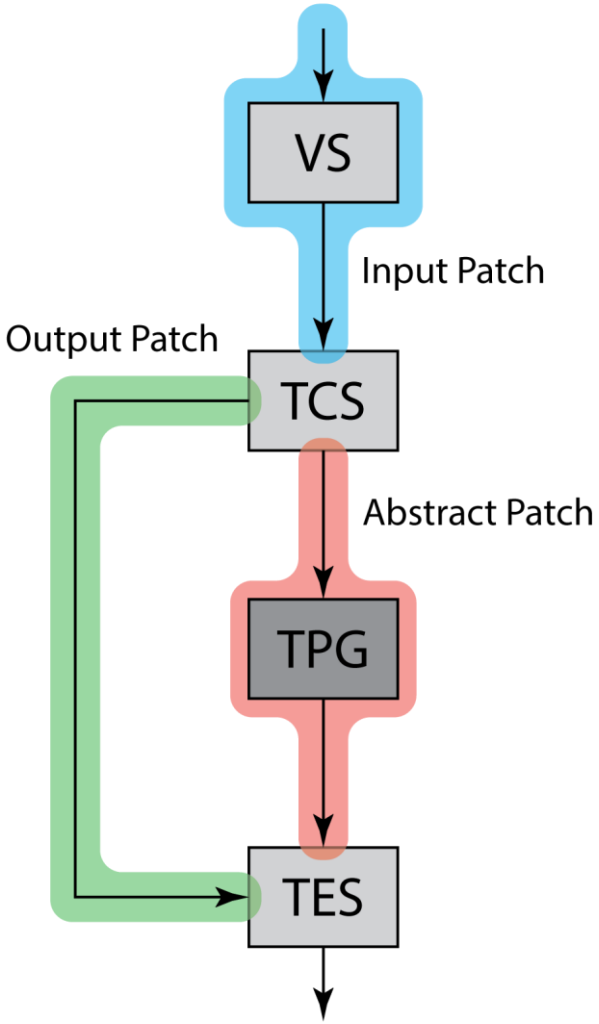
```
// TES: Abstract Patch hat 3 Vertices
layout (triangles) in;
```

Abstract Patch

Die Stärke der Tessellation wird durch die Standardwerte bestimmt.

```
PatchParametri(PATCH_DEFAULT_INNER_LEVEL, x);
PatchParametri(PATCH_DEFAULT_OUTER_LEVEL, z);
```

Vertex-basiertes Displacement Mapping **Tessellation**



Datenfluss mit TCS

Input Patch

```
// Host: Input Patch hat 3 Vertices
glPatchParameteri(GL_PATCH_VERTICES, 3):
```

Output Patch

```
// TCS: Output Patch hat 3 Vertices
layout (vertices = 3) out;
```

Abstract Patch

```
// TES: Abstract Patch hat 3 Vertices
layout (triangles) in;
```

Tessellation Level
Der TCS kann die Stärke der Tessellation steuern.

Achtung:
Die Vertex-Anzahlen müssen nicht übereinstimmen. (z.B. Spline-Flächen)

Vertex-basiertes Displacement Mapping **Tessellation**

Tessellation Domain – Dreieck

TCS

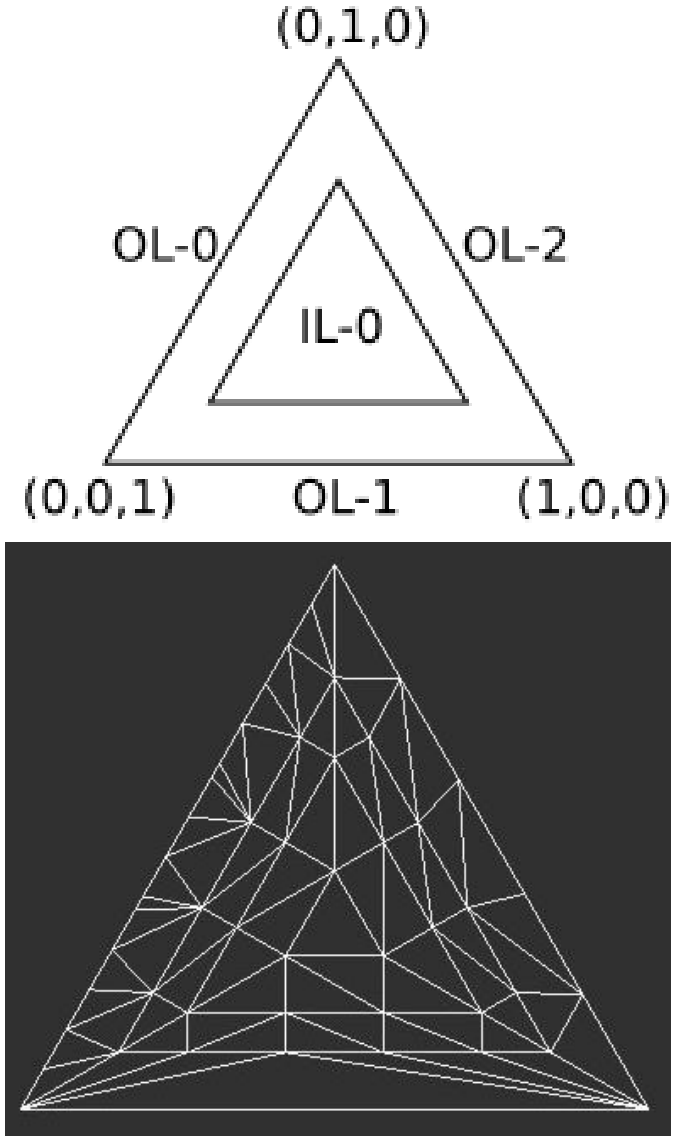
Für die Triangle Tessellation muss ein inneres und drei äußere Tessellation Level festgelegt werden.

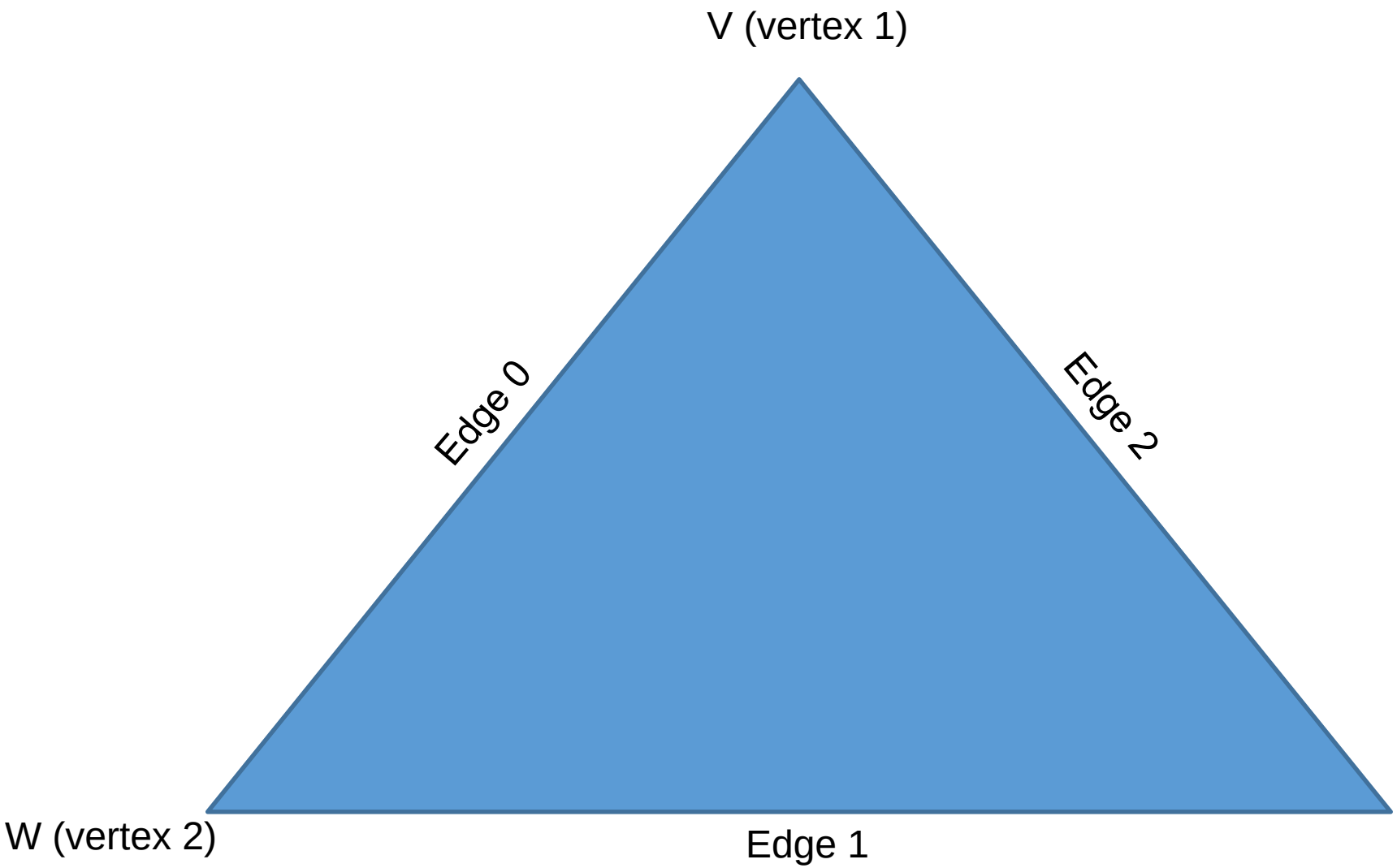
```
gl_TessLevelInner[0] =
…
gl_TessLevelOuter[0] =
…
gl_TessLevelOuter[1] =
…
gl_TessLevelOuter[2] =
…
```

TES

Bei der Auswertung steht auf *gl_TessCoord* die Baryzentrische Koordinate des neuen Vertex bereit.

```
vec3 interpolate3D(vec3 v0, vec3 v1, vec3 v2) {
        return gl_TessCoord.x * v0
             + gl_TessCoord.y * v1
             + gl_TessCoord.z * v2;
}
```





Edge 0 => gl_TessLevelOuter[0]

Edge 1 => gl_TessLevelOuter[1]

Edge 2 => gl_TessLevelOuter[2]

Vertex-basiertes Displacement Mapping **Tessellation**

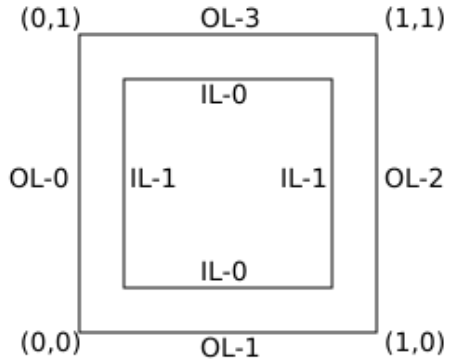
Tessellation Domain – Viereck und Linien

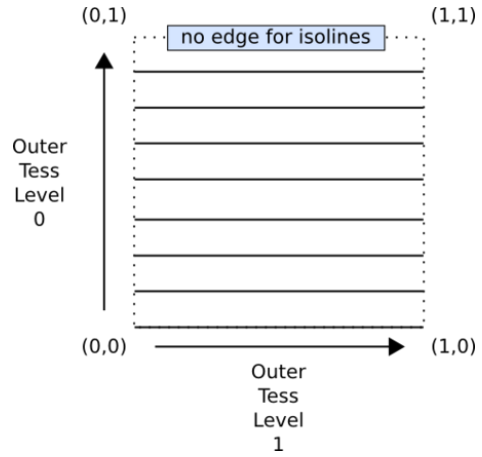
Quad

- 4 äußere Tessellation Level
- 2 innere Tessellation Level
- gl_TessCoord → 2D Koordinate zur bilinearen Interpolation

Lines

- (nur) 2 äußere Tessellation Level
- gl_TessCoord → 2D Koordinate
  - X → definiert Position auf der Linie
  - Y → definiert die Linie



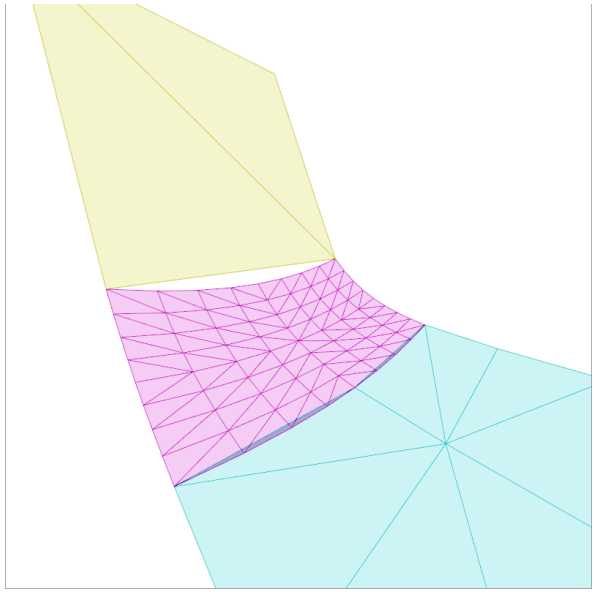


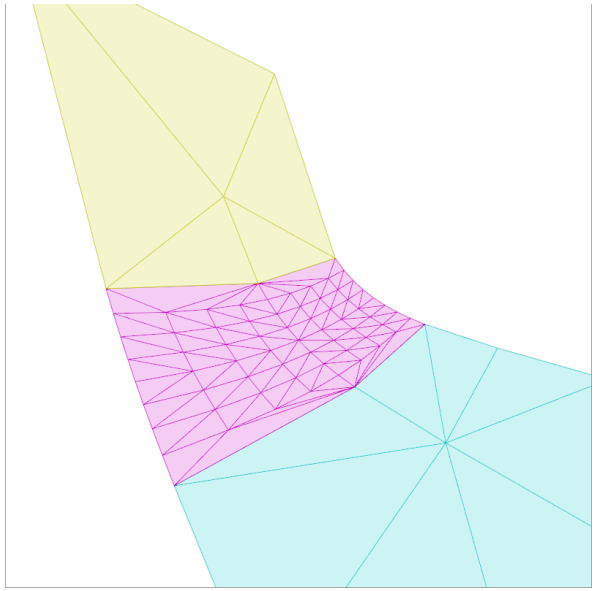
Vertex-basiertes Displacement Mapping **Tessellation**

Tessellation Level

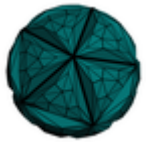
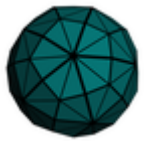
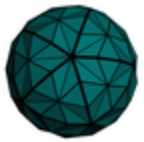
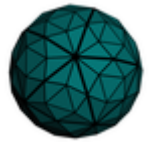
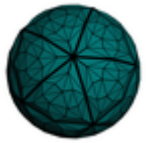
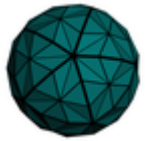
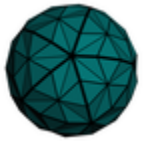
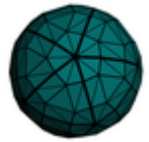
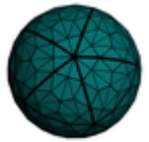
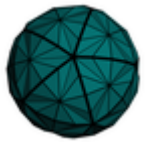
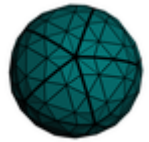
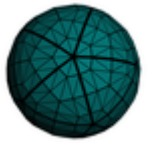
Die Stärke der Tessellation kann innerhalb und an den Außenkanten des Patches separat gesteuert werden.

So kann die Grenze zwischen zwei Patches aufeinander abgestimmt werden, so dass keine Löcher entstehen und gleichzeitig können die Patches unterschiedlich stark tesselliert werden.

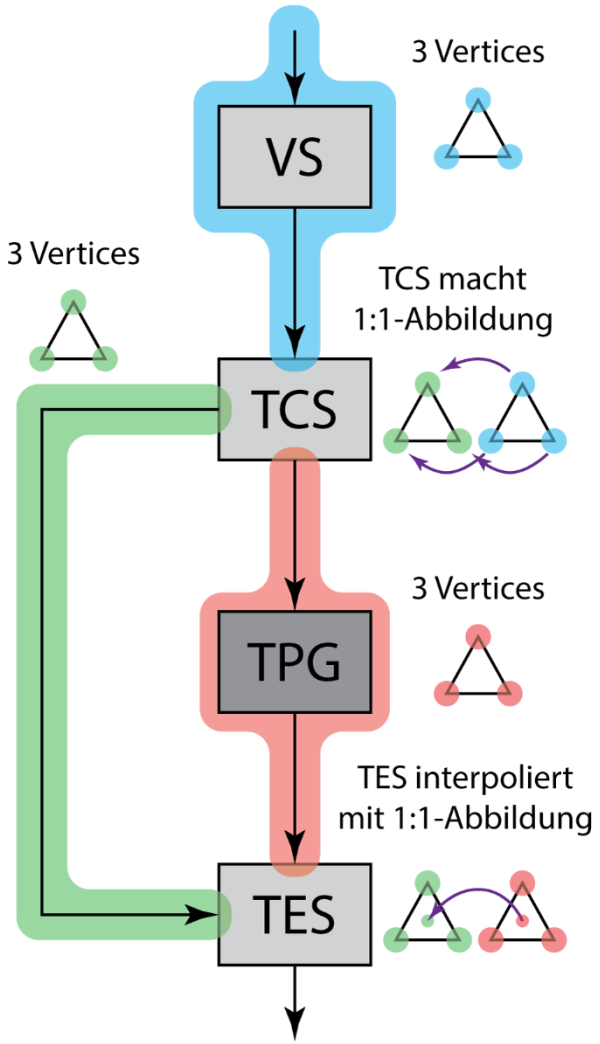


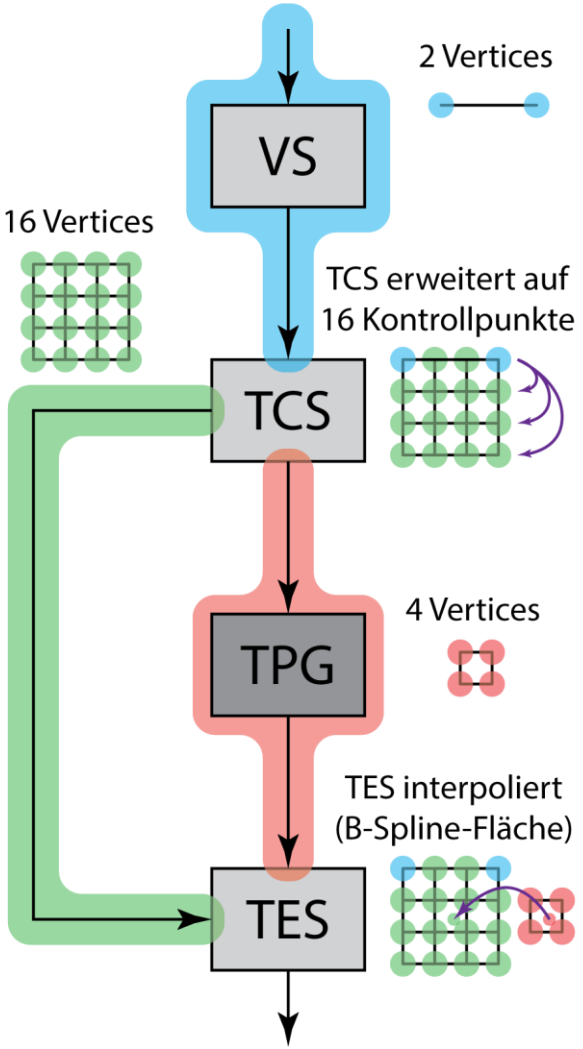


Inner and Outer Tess Levels

	Inner = 1	Inner = 2	Inner = 3	Inner = 4
Outer =1				
Outer = 2				
Outer = 3				
Outer = 4				

Vertex-basiertes Displacement Mapping **Tessellation**





Vertex-basiertes Displacement Mapping **Tessellation -** Anwendung

Host

```
// Definiert die Anzahl der Vertices des Input Patches
glPatchParametri(GL_PATCH_VERTICES, 3);
glDrawElements(GL_PATCHES, numIndices, GL_UNSIGNED_INT, 0);
```

Vertex-basiertes Displacement Mapping **Tessellation -** Anwendung

Vertex Shader

```
// Eingaben werden durch die Attribute Pointer definiert
layout (location = 0) in vec3 vPosition;
layout (location = 1) in vec3 vNormal;

// Attribute werden pro Vertex an den TCS weitergegeben
out VS_OUT {
        vec3 position;
        vec3 normal;
} vs_out;

void main() {
        vs_out.position = vPosition;
        vs_out.normal = vNormal;

        // Die vordefinierte Variable gl_Position wird im TES beschrieben
}
```

Vertex-basiertes Displacement Mapping **Tessellation -** Anwendung

Tessellation Control Shader

```
// Definiert die Größe des Output Patches
layout (vertices = 3) out;

// Attribute des Input Patches (Länge abhängig vom Input Patch)
in VS_OUT {
        vec3 position;
        vec3 normal;
} tcs_in[];

// Attribute des Input Patches (Länge abhängig vom Input Patch)
out TCS_OUT {
        vec3 position;
        vec3 normal;
} tcs_out[];

uniform float TessLevelInner;
uniform float TessLevelOuter;
```

Vertex-basiertes Displacement Mapping **Tessellation -** Anwendung

Tessellation Control Shader (fort.)

```
void main() {
	// Hier 1 zu 1 Abb. zwischen Input und Output Patch
	tcs_out[gl_InvocationID].position = tcs_in[gl_InvocationID].position;
	tcs_out[gl_InvocationID].normal = tcs_in[gl_InvocationID].normal;

	// die Tessellations Level müssen nur einmal pro Patch definiert werden
	if(gl_InvocationID == 0) {
		gl_TessLevelInner[0] = TessLevelInner;
		gl_TessLevelOuter[0] = TessLevelOuter;
		gl_TessLevelOuter[1] = TessLevelOuter;
		gl_TessLevelOuter[2] = TessLevelOuter;
	}
}
```

Vertex-basiertes Displacement Mapping **Tessellation -** Anwendung

Tessellation Evaluation Shader

```
// Definiert die Größe des Abstract Patches (Hier Triangles = 3 Vertices)
layout (triangles, equal_spacing, ccw) in;

// Attribute des Output Patches (Länge abhängig vom Output Patch)
in TCS_OUT {
        vec3 position;
        vec3 normal;
} tes_in[];

out vec4 fColor;

uniform mat4 ModelMatrix;
uniform mat4 ViewMatrix;
uniform mat4 ProjectionMatrix;
```

Vertex-basiertes Displacement Mapping **Tessellation -** Anwendung

Tessellation Evaluation Shader (fort.)

```
vec3 interpolate3D(vec3 v0, vec3 v1, vec3 v2) {
        return gl_TessCoord.x * v0          // Baryzentrische Interpolation der Attribute
                + gl_TessCoord.y * v1       // des Output Patches auf Basis der Position
                + gl_TessCoord.z * v2;      // des neuen Vertex im Abstract Patch
}

vec3  process(vec3 vertex){….}      // z.B. entsprechend Displacement Map o.Ä. (texCoord !!!!!!)

void main() {
        vec3 oldPosition = interpolate4D(tes_in[0].position, tes_in[1].position, tes_in[2].position);
        vec3 newPosition = process(oldPosition);

        fColor = vec4(1, 0, 0, 1 );
        gl_Position = ProjectionMatrix*ViewMatrix * ModelMatrix * vec4(newPosition, 1.0);
}
```

Vertex-basiertes Displacement Mapping **Tessellation -** Anwendung

Fragment Shader

```
in vec4 fColor;

out vec4 FragColor;

void main() {
	FragColor = fColor;
}
```

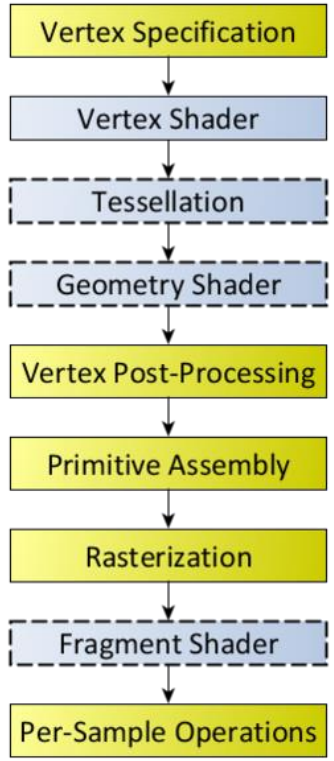
Vertex-basiertes Displacement Mapping **Tessellation -** Zusammenfassung

1) Vertex Shader wird auf jedem Patch ausgeführt. Der Patch umfasst einige CPs aus dem Vertex Buffer (limitiert durch die GPU und dessen Driver)
2) TCs generiert aus den CPs des Vertex Shaders output Patch sowie die TLs
3) Der PG generiert domain location und deren Verknüpfung aus der konfigurierten domain und den TLs sowie der definierten Output Topologie
4) Der TES wird auf allen generierten Domain locations ausgeführt
5) Die primitive die erstellt wurden in Schritt 3 werden weitergereicht. (Daten aus dem TES)
6) Weitergabe an GS oder an der Rasterizer.

Geometrie Shader

- Aufruf nach dem Vertex-Shader
- Erzeugung neuer primitiven Geometrien aus vorhandenen Primitiven (Punkten, Linien, Dreiecken) [müssen nicht identisch sein]
- Typische Anwendung: Fell/Haar Geometrien
- Potentiell mehrere Aufrufe pro Primitiv

Nutzung: Siehe Beispielprogramm



Geometrie Shader - Beispiel

```
#version 330 core

layout (triangles) in ;                                  // Definiert den Eingabe-Primitivtyp
layout (line_strip, max_vertices=6) out;                 // Definiert den Ausgabe-Primitivtyp und die max. Anzahl von
                                                         // Vertices, die pro Aufruf erzeugt werden

/* Deklarierung in Variablen */

/* Deklarierung out Variablen */

/* Deklarierung uniform Variablen (um weitere Daten von der CPU zu bekommen) */

void main(){
        gl_Position = gl_in[0].gl_Position;              // beschreibe gl_Position und alle Attribute für die Folge-
                                                         // shader
        // Color, TexturesCoords….
        EmitVertex();                                    // Erstellt einen neuen Vertex in der Pipeline
        // more Vertices…. (Nach EmitVertex() sind alle Attribute undefiniert!!!)
        EndPrimitive();                                  // Schließt das Primitiv ab. (Danach ggf. weiter mit neuem!)
}
```

Geometry Instancing

Im Geometry Shader

Der Geometry Shader kann mehrfach pro Primitiv aufgerufen werden. Geometrie kann dadurch parallel erzeugt werden.

Die Anzahl der Aufrufe wird mit dem folgenden *layout qualifier* bestimmt:

```
layout (invocations = num_instances) in;
```

Die Variable *gl_InvocationID* gibt an, welcher Aufruf gerade ausgeführt wird.

Beispiel Geometry Instancing

Für jedes eingehende Dreick werden per Geometry Instancing zwei neue

```
// Dreiecke als Eingabe und 2 Aufrufe pro Dreieck
layout (triangles, invocations = 2) in;

// Zwei Dreiecke (6 Vertices) werden ausgegeben
layout (triangle_strip, max_vertices=6) out;

in vec3 gPosition[];                    // Positionen vom Vertex Shader
out vec4 fColor;                // Farbe für den Fragment Shader

// Versatz zwischen den Dreiecken
const float offsets[2] = float[2](0, 1);
```

Beispiel Geometry Instancing – (fort.)

```
void main() {
        for (int i=0; i < gl_in.length(); i++) {
                float offset = offset[gl_InvocationID];
                gl_Postion = gPosition[i] + vec4(offset, 0,0,0);
                EmitVertex();
        }
}
```

Due Date

25.11.