



4. Materialien variieren

Unterschiedliche Materialien für den Körper

Der reale Anwendungsfall bedeutet, dass die Materialien auf einer Geometrie nicht überall die gleichen sind. Modelliert man z.B. einen Menschenkopf muss man bereits zwischen Haut und Haaren unterscheiden.

Wir starten damit, dass wir zunächst auf jede Seite unseres Tetraeders ein anderes Material aufbringen.

Dazu starten wir mit der HTML-Datei, in der wir den Tetraeder selbst konstruiert haben und speichern die Datei unter dem Namen `L4-1_Verschiedene_Materialien.html`.

Die einzelnen Materialien haben wir bereits in Lerneinheit 2 kennengelernt. Wir haben vier Seiten, also erzeugen wir 4 Materialien. Der Einfachheit halber sind im Beispiel drei Standard- und ein Toon-Material kombiniert worden. Grundsätzlich können aber alle der bisher vorgestellten Materialien kombiniert werden.

Zum Aufbringen der Materialien auf den Körper müssen wir diese vier Materialien in einem Array zusammenführen:

```
let tetraMaterials = [side1Material, side2Material, side3Material, side4Material];
```

Dann müssen wir dem Geometrie-Objekt sagen, welches Material auf welche Seite kommt. Zuerst löschen wir die vorhandene (default) Gruppenzuordnung. Dann definieren wir nacheinander für jedes Element im Materialarray zu welchen Spalten in der Geometriebeschreibung es zugeordnet werden soll.

```
geometry.clearGroups();  
geometry.addGroup(0, 3, 0);  
geometry.addGroup(3, 3, 1);  
geometry.addGroup(6, 3, 2);  
geometry.addGroup(9, 3, 3);
```

So bedeutet der erste Parameter ab welcher Spalte die Zuordnung gilt (0, 3, 6, 9), der zweite Parameter für wie viele Spalten es gelten soll (immer 3) und der letzte Parameter gibt den Index im Materialarray an. Mit der ersten Definition wird Index 0 (side1Material) den Spalten 0, 1, 2 zugeordnet.

Danach können wir unser Mesh wie gewohnt definieren, nur dass wir diesmal das Material-Array als Parameter übergeben.

```
tetraMesh = new THREE.Mesh(geometry, tetraMaterials);
```

```
let side1Material = new THREE.MeshStandardMaterial({  
  color: 0xfa91d3,  
  side: THREE.DoubleSide,  
  roughness: 0.5,  
  metalness: 0.7,  
});  
let side2Material = new THREE.MeshStandardMaterial({  
  color: 0xecf00c,  
  side: THREE.DoubleSide,  
  roughness: 0.5,  
  metalness: 0.7,  
});  
let side3Material = new THREE.MeshStandardMaterial({  
  color: 0x54ebc5,  
  side: THREE.DoubleSide,  
  roughness: 0.5,  
  metalness: 0.7,  
});  
let side4Material = new THREE.MeshToonMaterial({  
  color: 0xeeddff,  
  side: THREE.DoubleSide,  
});
```



Aufgabe

Ändert die Datei wie oben angegeben ab und ergänzt es um den Code mit den verschiedenen Materialien (selbstverständlich kann auch anderes Material als das angegebene Material verwendet werden). Auf die GUI-Kontrollen können wir diesmal verzichten.

Materialien aus Bildern

Interessanter wird es, wenn wir Bilder oder sogar Filme als Oberfläche für unsere Materialien verwenden. Diese Bilder oder Filme müssen wir allerdings importieren. Grundsätzlich steht uns ein Importmechanismus zur Verfügung, jedoch legt uns eine Eigenschaft moderner Browser einen kleinen Stein in den Weg. Damit Scripte, die wir beim Laden einer Webseite unbemerkt mitladen, nicht einfach auf unserer Platte nach Dateien suchen können, verbieten moderne Browser einfach den Zugriff auf lokale Dateien („...ccess to image at ... has been blocked by CORS policy“). Wenn wir ihn bisher nicht genutzt haben, brauchen wir spätestens jetzt einen kleinen eigenen lokalen Webserver (siehe Lerneinheit 1).

Dateien können mit einem LoadManager geladen werden, für die einzelnen zu ladenden Datentypen gibt es dann Spezialisierungen. Hier verwenden wir jetzt den TextureLoadManager.

```
let loadManager = new THREE.LoadingManager();
let loader = new THREE.TextureLoader(loadManager);

let map = loader.load('textures/IMG_5210.JPG');
```

Im Codeauszug wird zuerst ein Objekt vom Typ LoadingManager erstellt, das wir dann dem Konstruktor des TextureLoaders übergeben. Texturen sind Bilder, die wir einem Material als Sammler für alle Eigenschaften einer Oberfläche übergeben. Im obigen Beispiel wird ein Bild aus dem Unterordner „textures“ geladen (siehe Beispiele). Im allgemeinen bezeichnet man diese Texturen auch als map für die Oberfläche.

Das Material, das wir mit den Bildern füllen, sollte so realistisch wie möglich aussehen. Bilder sind jedoch Zweidimensional und in vielen Fällen fehlt ihnen Tiefenwirkung.

```
let baseMap = loader.load('Vorlage.JPG');
let bumpMap = loader.load('VorlageBump.jpg');
let specMap = loader.load('VorlageSpec.jpg');

let tetraMaterial = new THREE.MeshPhongMaterial({
  map: baseMap,
  bumpMap: bumpMap,
  bumpScale: 0.05,
  specularMap: specMap,
  specular: new THREE.Color('grey'),
});
```

Tatsächlich laden wir hier 3 Bilder. Das erste speichern wir als baseMap, das zweite als bumpMap und das dritte als specMap. So übergeben wir sie auch dem Material. Was bewirken



3D-Programmierung

Materialien variieren

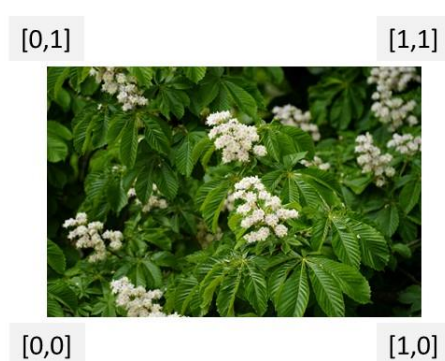
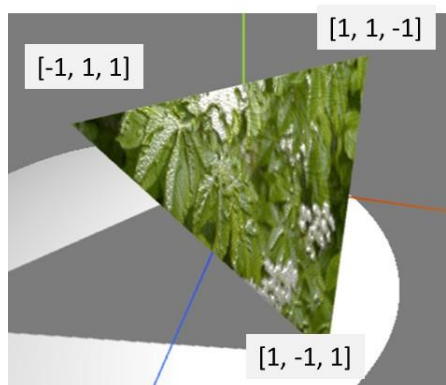
nun aber die Bilder? Dazu sollen die nachfolgenden Darstellungen dienen, be denen nacheinander die einzelnen Maps geladen werden. Es fängt damit an, dass im ersten Bild nur die baseMap zum Material hinzugegeben wird (für eine besser Erkennbarkeit wurde die Kamera nah an der Oberfläche positioniert).

Bump Maps sind Höhenkarten, die man mit Hilfe von Filtern aus dem eigentlichen Bild gewinnen kann. Die Specular Map schließlich legt die glänzenden Regionen im Bild fest (wird daher auch Gloss map genannt).



So bieten die meisten Bildbearbeitungsprogramme mehr oder weniger einfach die Möglichkeit, aus einem gegebenen Bild eine Bump Map zu erzeugen. Diese Höheninformation nutzt der Renderer als Hilfestellung zur Berechnung von Reflektionen. Der dadurch gewonnene Eindruck scheint mehr Detailreichtum im Bild zu generieren.

In diesem Zusammenhang kommen wir wieder auf das bereits früher erwähnte uv-Mapping zurück. Dazu stellen wir uns am besten eine Seite im 3-dimensionalen Raum (links) und ein Bild (rechts) vor. Während die Seite durch 3-dimensionale x,y,z-Koordinaten aufgespannt wird, haben wir beim 2-





dimensionalen Bild auch nur 2-dimensionale Koordinaten (unabhängig von der tatsächlichen Bildgröße immer auf 1 normiert, so dass links unten 0,0 und rechts oben 1,1 ist).

Die Abbildung erfolgt nun so, dass zu jedem Vertex im 3-dimensionalen Raum der entsprechende Punkt im 2-dimensionalen Raum des Bildes angegeben wird. Dies funktioniert offensichtlich auch trotz der Verzerrung durch die nur dreieckige Seite.

Aufgabe

Speichert eure Datei unter dem Namen `L4-2_Materialien-importieren.html` ab. Ergänzt den Code um die Importe und importiert die Bilder. Hierfür können entweder die in den Materialien enthaltenen Bilder verwendet werden, oder aber eigene.

Materialien auf generierten Oberflächen

Nun haben wir in Lerneinheit 3 zum Schluss über Algorithmen Vertices erzeugt und diese über die Delaunator-Bibliothek zu Faces verbinden lassen. Wie sieht es hier aus mit der Belegung dieser Oberfläche mit eigenen Bildern?

Wir haben bei dem Tetraeder neben den eigentlichen Vertices auch die Normalen und die uv-Werte erzeugt. Letztere sind es, die für das Mapping der Textur (also hier unseres Bildes) verantwortlich sind.

Speichert dazu die Datei, mit der die generierte Oberfläche aus der Lerneinheit 3 unter dem Namen `L4-3_Materialienimport-Delaunator.html` ab.

Ein erster Ansatz könnte sein, einfach den Punkten nacheinander die Werte [0,0] (linke untere Ecke), [0,1] (linke obere Ecke) und [1,1] (rechte obere Ecke) zuzuweisen (da wir nur immer 3 Punkte haben, können wir nur 3 uv-Punkte zuweisen). Statt der linken oberen Ecke könnte man auch die rechte untere Ecke nehmen. Wichtig ist, dass immer der Ursprung [0, 0] und die rechte obere Ecke [1, 1] dabei ist.

```
let uvs = [];  
  
// für jede Vertice nacheinander iterierend [0,0], [0,1] und [1,1] hinzufügen  
  
let uvsBuffer = new Float32Array(uvs);  
geometry.setAttribute('uv', new THREE.BufferAttribute(uvsBuffer, 2));
```



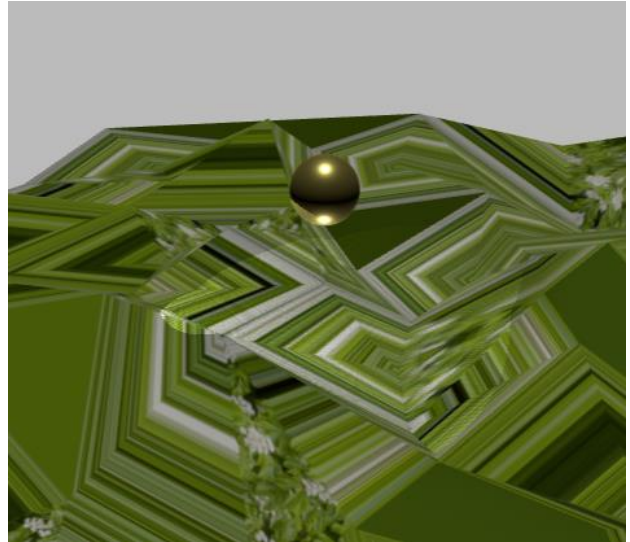

3D-Programmierung

Materialien variieren

Dann erhalten wir das nebenstehende Ergebnis.

Man sieht dem Bild sofort an, dass die Zuweisung durch die automatische Face-Erzeugung zu Verzerrungen des Bildes führt. Bei unstrukturierten Bildern wie einer Wasseroberfläche oder der Oberfläche eines Steines kann dieser Effekt bereits zu einem hinreichenden Ergebnis führen.

Etwas hübscher wird das Ergebnis, wenn man über die Faces iteriert, ebenfalls im 3er-Schritt.



```
let uvs = [];  
for (var i = 0; i < indexDelaunay.triangles.length; i += 3) {  
  uvs.push(0, 0, 1, 1);  
};
```

Wie man sehen kann, wurden diesmal nur die linke untere und die rechte obere Ecke zugewiesen. Das Ergebnis wirkt gleichmäßiger als das vorhergehende.

Vergleicht man die beiden Ergebnisse, so sieht man im oberen Bild vorne typische Artefakte, die durch die nachträgliche Verarbeitung entstehen. Im unteren Bild sieht das Ergebnis gleichmäßiger aus.

