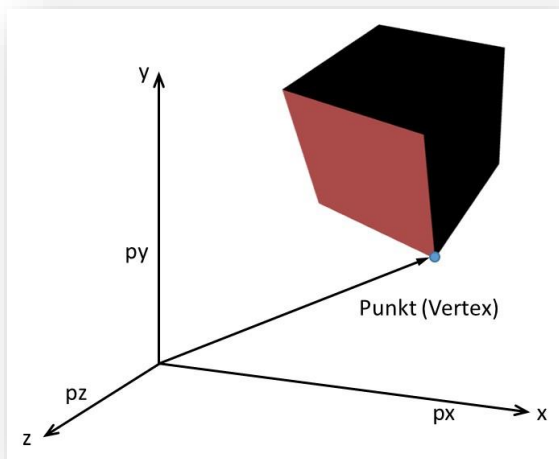




3. Eigene Geometrie erstellen

Bislang haben wir nur vorhandene Basis-Geometrien verwendet. Neben dem Kubus sind u.a. Sphären (Kugeln), Flächen und Linien vorhanden. Die Basis aller Modellierungen sind jedoch Punkte, die frei definiert werden können und die über Linien verbunden werden (siehe auch Initialisierung der Sphere im letzten Arbeitsblatt). So können zusammenhängende Flächen entstehen, die wiederum mit Material bedeckt werden können.



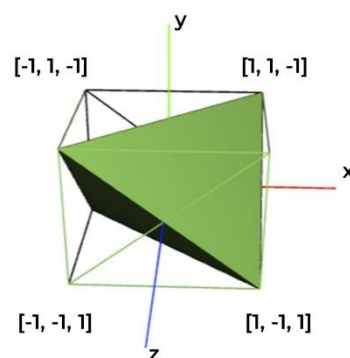
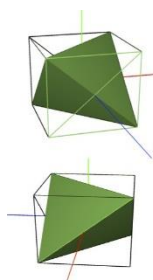
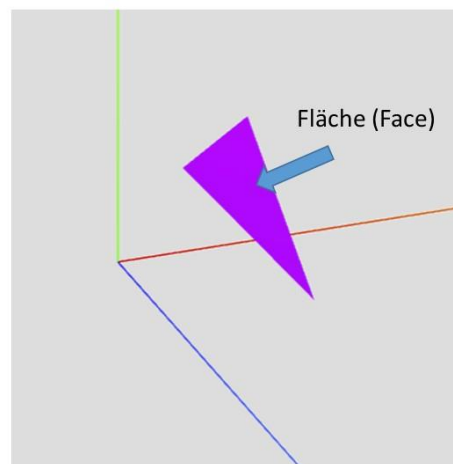
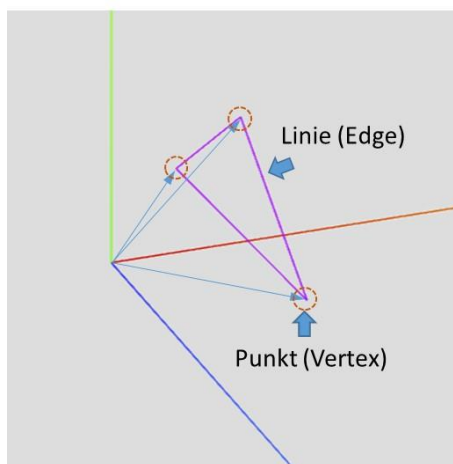
Zunächst müssen wir uns über die Art und Weise, wie die Geometrie erstellt wird, im Klaren sein. Ausgangspunkt ist der Punkt (Vertex), der im Raum über eine dreidimensionale Koordinate beschrieben wird (hier nebenstehend am Beispiel des einfachen Kubus). Immer ausgehend vom Ursprung (0,0,0) definieren wir Punkte im Raum mit ihren x-, y- und z-Koordinaten.

Verknüpft man zwei Vertices miteinander, bezeichnet man die Verknüpfungslinie als Edge.

Im Laufe der Entwicklung hat sich herausgestellt, dass die Definition der Geometrie aus

Dreiecken am besten von modernen Grafikkarten verarbeitet werden kann bzw. Grafikkarten auf deren Verarbeitung optimiert wurden. Demnach bilden immer 3 Edges zusammen ein Face. Ein Face ist nichts anderes als ein im 3-dimensionalen Raum definierte Fläche, also ein ebenes 2-dimensionales Gebilde. Die Körper entstehen nun dadurch, dass die einzelnen Faces in unterschiedlichen Winkeln aneinander gefügt werden. Oben ist ein einzelnes Face zu sehen, mit 3 weiteren Faces können einen Tetraeder erzeugen, bei dem man gut erkennen kann, dass der gesamte Körper aus einzelnen Dreiecken zusammengesetzt ist.

Damit man sich den Körper besser vorstellen kann, würde ein Kubus eingebaut, dessen Ecken genutzt werden, um den Tetraeder zu definieren. Der umschließende Kubus hat seinen Mittelpunkt im Ursprung. Der Kubus ist nur als Rahmen (wireframe: true) dargestellt, damit sind nur seine Edges zu sehen.





3D-Programmierung

3D-Programmierung – eigene Geometrien

(Hinweis vorab: um die uv-Werte richtig zuordnen zu können, sollten die Vertices aus Sicht der später sichtbaren Oberfläche linksdrehend zusammengefasst werden).

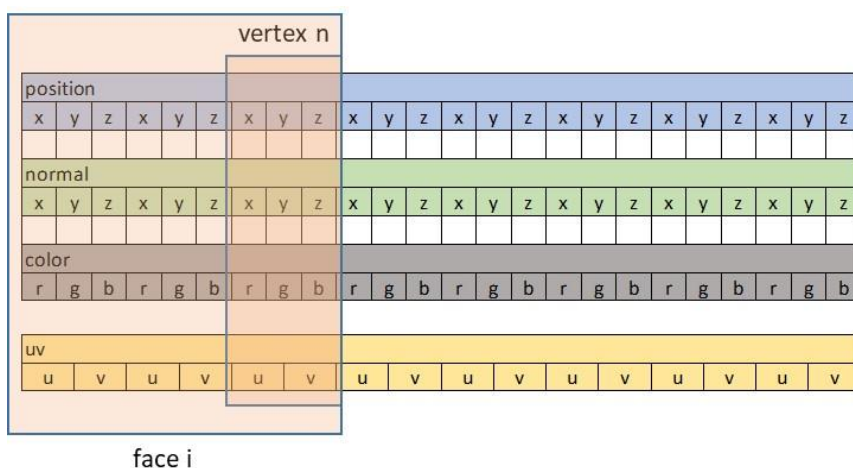
In three haben wir die BufferGeometry, die uns universell zur Verfügung steht (die vordefinierten Geometrien werden in eine Buffergeometry konvertiert, die Struktur der BufferGeometry ist so ausgelegt, dass sie sich optimal mit Graphikkarten verarbeiten lässt).

Die BufferGeometry ist eine Sammlung von Bufferattributen. Jedes Bufferattribut (BufferAttribute) entspricht einem Array eines wesentlichen Darstellungstypen (horizontal):

- Positionen (der Punkte)
- Normalen dazu
- Farben
- UV

Jeder Punkt wird durch eine Spalte in diesem Zweidimensionalen Array repräsentiert (siehe unten). Die Normale ist ein Vektor, der orthogonal zu dem Positions-Vektor steht. Im Farbwert ist die Farbinformation gespeichert.

Mit UV wird die Textur an die Geometrie angebunden, sie beschreiben die Texturkoordinaten. Dabei sind u und v willkürlich gewählt und sind keine Abkürzung. Jeder Punkt des Polygonobjektes bekommt dabei eine eindeutige Texturposition. Mit dem UV-System kann der Anwender ein Bild für die Textur einer Geometrie verwenden (dazu später mehr).



Jeweils drei aufeinanderfolgende Vertices bilden dann ein Face (eine Dreiecksfläche).

Starten wir mit der zur Verfügung gestellten Datei

[L3-1_Geometrie-erstellen.html](#)

(Alternativ können Sie aber auch mit einer der bisherigen Dateien starten, z.B. L2-3_Materialien.html, und dort alle Geometrien, Materialien und GUI-Control-Einträge löschen).

Wir wollen einen Tetraeder erstellen, also brauchen wir dazu zunächst 4 Punkte (vertices). Damit haben wir einen noch überschaubaren Körper, auch wenn später die Oberflächen praktische Szenarien – insbesondere, wenn sie gekrümmt sind – aus vielen Hundert, eher Tausenden Faces bestehen. Unsere 4 Punkte A, B, C und D:



3D-Programmierung

3D-Programmierung – eigene Geometrien

```
let pA = [-1, -1, -1];  
let pB = [1, -1, 1];  
let pC = [-1, 1, 1];  
let pD = [1, 1, -1];
```

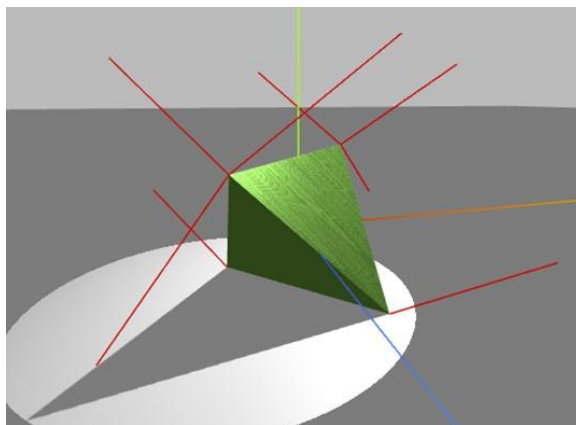
Um die Daten zeilenweise in die BufferGeometry zu übertragen, definieren wir 3 Arrays, die wir mit den Daten füllen und diese wiederum übertragen wir in die BufferGeometry.

Die Struktur ist – sofern man sie per Hand eingibt – etwas unübersichtlich, allerdings lassen sich die Daten auch gut berechnen. Zuerst kommen die Flächen. Wie oben gezeigt, bauen wir aus den 4 Punkten das Array „Positionen“ – also die erste Zeile der obigen Darstellung – zusammen.

```
let vertices = new Float32Array([  
  // side (S1): pA, pB, pC  
  pA[0], pA[1], pA[2], pB[0], pB[1], pB[2], pC[0], pC[1], pC[2],  
  // side (S2): pB, pD, pC  
  pB[0], pB[1], pB[2], pD[0], pD[1], pD[2], pC[0], pC[1], pC[2],  
  // side (S4): pB, pA, pD  
  pB[0], pB[1], pB[2], pA[0], pA[1], pA[2], pD[0], pD[1], pD[2],  
  // side (S3): pA, pC, pD  
  pA[0], pA[1], pA[2], pC[0], pC[1], pC[2], pD[0], pD[1], pD[2],  
]);
```

In der 2. Zeile werden die Normalen hinzugefügt. Die Normalen sollen senkrecht auf der Reflektionsfläche stehen und geben Höheninformationen für die Reflektion weiter (dazu auch später mehr). Da wir ebene Seiten haben wollen, geben wir für jede Seite den Normalenvektor an (pS1n = point Side 1 normal). Durch die Anzeige mit Hilfe der VertexNormalsHelper kann man die Normalen sichtbar machen (siehe rechts).

```
let pS1n = [-1.0, -1.0, 1.0];  
let pS2n = [1.0, 1.0, 1.0];  
let pS3n = [1.0, -1.0, -1.0];  
let pS4n = [-1.0, 1.0, -1.0];
```



Damit definieren wir auch wieder das Array mit allen Normalenpunkten.

```
let normals = new Float32Array([  
  pS1n[0], pS1n[1], pS1n[2], pS1n[0], pS1n[1], pS1n[2], pS1n[0], pS1n[1], pS1n[2],  
  pS2n[0], pS2n[1], pS2n[2], pS2n[0], pS2n[1], pS2n[2], pS2n[0], pS2n[1], pS2n[2],  
  pS3n[0], pS3n[1], pS3n[2], pS3n[0], pS3n[1], pS3n[2], pS3n[0], pS3n[1], pS3n[2],  
  pS4n[0], pS4n[1], pS4n[2], pS4n[0], pS4n[1], pS4n[2], pS4n[0], pS4n[1], pS4n[2],  
]);
```



Zum Schluss kommen noch die uv-Punkte hinzu.

```
let uvs = new Float32Array([
  0, 0, 0, 1, 1, 0,
  0, 0, 0, 1, 1, 0,
  0, 0, 0, 1, 1, 0,
  0, 0, 0, 1, 1, 0,
]);
```

Mit den uv-Punkten beschreiben wir die Zuordnung eines 2-dimensionalen Bildes zu der 3-dimensional im Raum stehenden Fläche (auch dazu später mehr). Aber man sieht, dass es pro Seite nur 6 Zahlen gibt, das entspricht je zwei Werten pro Vertex (von denen ja jeder aus 3 Zahlen besteht).

Nun können wir diese 3 Arrays der BufferGeometry übergeben. Dazu definieren wir nur, welches Array welchem Attribut entspricht.

```
geometry.setAttribute('position', new THREE.BufferAttribute(vertices, 3));
geometry.setAttribute('normal', new THREE.BufferAttribute(normals, 3));
geometry.setAttribute('uv', new THREE.BufferAttribute(uvs, 2));
```

Wie man sieht, fehlt noch die Farbinformation, die wir jetzt noch über das Material hinzufügen, und wir können unsere Tetraeder-Mesh erzeugen.

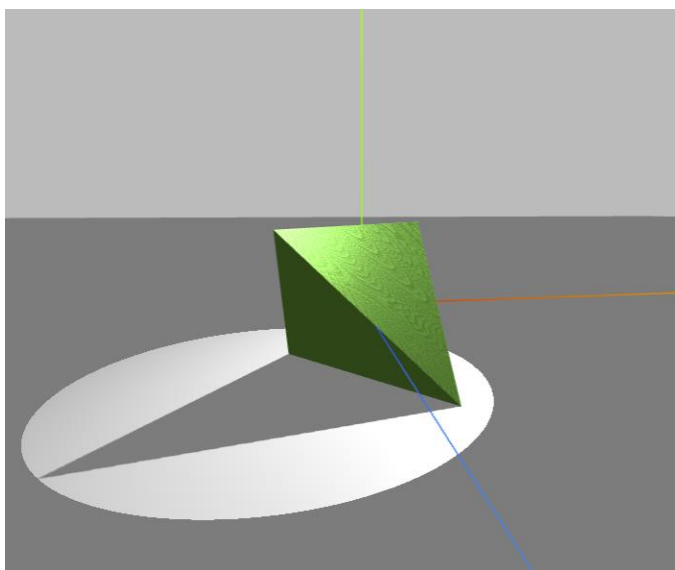
```
let tetraMaterial = new THREE.MeshStandardMaterial({
  color: 0x33ff22,
  side: THREE.DoubleSide,
  roughness: 0.5,
  metalness: 0.7,
  wireframe: false,
});

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

Wie auch im vorigen Beispiel fügen wir noch eine plane Ebene unter dem Körper hinzu, damit wir einen Schatten sehen können und ertüchtigen alle Elemente Schatten zu werfen und Schatten anzuzeigen.

Im Beispiel sind noch die Achsenhelfer und ein Normalenhelfer zu finden, mit denen die Normalen und die Achsen angezeigt werden.

Das Ergebnis ist eine Szene mit Schattenwurf, für die wir erstmals eine eigene Geometrie aufgebaut haben.





Geometrien systematisch konstruieren und triangulieren

Letztlich ist es etwas mühsam, komplexe Gebilde wie Maschinen, Personen o.ä. auf diese Weise aufzubauen. Dennoch werden alle diese Geometrien in die Struktur wie oben gezeigt überführt, da diese Struktur am effizientesten durch Grafikkarten verarbeitet werden kann.

Doch wie kann man komplexere eigene Geometrien zusammensetzen ohne sie durch externe Modellierungstools zu erstellen? Die Vertices könnte man durch Gleichungen oder zufällig nach einer Abbildungsvorschrift berechnen. Aber wie setzt man damit die Dreiecke zusammen? Dafür wurde eine eigene Bibliothek erstellt: Delaunator (<https://github.com/mapbox/delaunator>), mit der dieses rechenintensive Vorhaben sogar noch relativ flott geschieht.

Um dieses Verfahren auszuprobieren, ist der mitgelieferten Bibliothek die Datei `delaunator.min.js` hinzugefügt, die nicht standardmäßig zur `three`-Bibliothek gehört sowie eine weitere Hilfsbibliothek (`perlin.js`), mit der die sogenannte Perlin Noise umgesetzt werden soll, die Oberflächen ein ‚natürliches‘ Aussehen verleihen soll (siehe auch <https://github.com/joeidddon/perlin>). Perlin Noise konnte nicht in den Cloud-Bibliotheken gefunden werden, so dass hier eine lokale Kopie zum Einsatz kommen muss.

```
import Delaunator from 'https://cdn.skypack.dev/delaunator@5.0.0';
```

bzw.

```
<script src="./perlin.js"></script>
```

Nun können wir an die Umsetzung einer Probe gehen. Dazu speichern wir zunächst unsere gerade erstellte Datei unter `L3-2_Geometrie-mit-Delaunator.html`.

Dazu starten wir der Einfachheit halber mit einigen wenigen, wahllos zusammengestellten Punkten, die Ihnen zur Arbeitserleichterung in den Materialien zur Verfügung stehen (`delaunator_test.txt`). Wir haben in dem Array `points` dann eine Liste von 3D-Punkten.

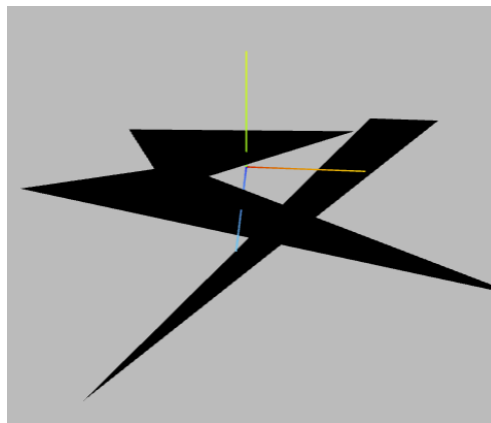
```
let points = [ new THREE.Vector3(-89.0, -1.0, 20.0),  
               new THREE.Vector3(93.0, -11.0, 53.0),  
               new THREE.Vector3(-25.0, 9.0, 14.0),  
               new THREE.Vector3(-61.0, -2.0, -33.0),  
               new THREE.Vector3(-26.0, 10.0, 31.0),  
               new THREE.Vector3(47.0, 8.0, -20.0),  
               new THREE.Vector3(92.0, 3.0, -43.0),  
               new THREE.Vector3(69.0, -17.0, -83.0),  
               new THREE.Vector3(-35.0, -9.0, 96.0),  
               new THREE.Vector3(-90.0, -19.0, 39.0), ];  
  
let geometry = new THREE.BufferGeometry().setFromPoints(points);
```



3D-Programmierung

3D-Programmierung – eigene Geometrien

Nehmen wir diese so entstandene Geometrie, so kann man schnell erkennen, dass so keine einheitliche Fläche entstanden ist. Warum? Wie wir vorne gelernt haben, werden jeweils einfach immer 3 in der Array-Struktur hintereinanderliegenden Punkte als Fläche interpretiert. Um eine zusammenhängende Fläche zu erzeugen, müssen die Punkte, an denen die Flächen zusammenstoßen sollen, in der Array-Struktur vervielfacht werden.



Hier kommt jetzt die Delaunator-Lib zum Einsatz.

```
// triangulate x, z
let indexDelaunay = Delaunator.from(
  points.map(v => {
    return [v.x, v.z];
  })
);
```

Wie man sehen kann, trianguliert die Bibliothek über zwei Koordinaten, x und z. Damit muss man sich entscheiden, über welches Koordinatenpaar man triangulieren will (Triangulationen über 3 Koordinaten sind erheblich komplexer und aufwendiger zu rechnen). Für den nächsten Schritt wird ein Abbildungsindex erzeugt und der Geometrie hinzugefügt.

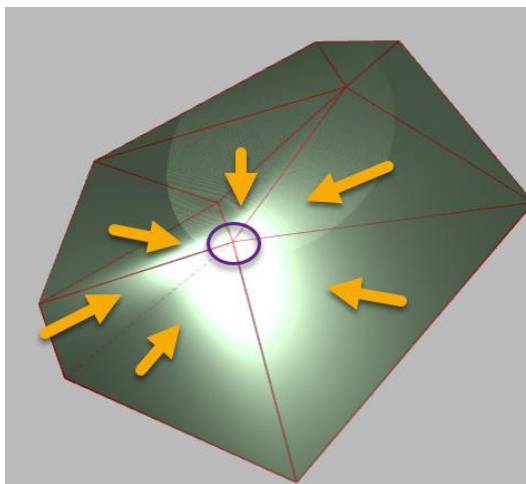
```
let meshIndex = []; // delaunay index => three.js index
for (let i = 0; i < indexDelaunay.triangles.length; i++) {
  meshIndex.push(indexDelaunay.triangles[i]);
}

geometry.setIndex(meshIndex); // add three.js index to the existing geometry
geometry.computeVertexNormals();
```

Damit entsteht eine geschlossene Fläche, die hier durch zusätzliche Wireframe-Linien veranschaulicht wurde.

Vergleicht man die beiden Graphiken miteinander, kann man sehr gut erkennen, dass aus den 10 Punkten, die ja wie vorne beschrieben, nacheinander in dem BufferArray angeordnet sind, 3 Dreiecke geworden sind, da ja ohne weitere Angaben immer drei Vertices zusammen ein Dreieck bilden.

Das Verfahren in der Delaunator-Lib hat nun dafür gesorgt, dass aus den gegebenen Punkten eine geschlossene Fläche geworden ist. Das kann nur passieren, wenn die Punkte durch das Verfahren so aufgestockt worden sind, dass z.B. der durch den Kreis markierte Punkt nun so oft in dem BufferArray einkopiert wurde, dass er zu allen durch die Pfeile markierten Dreiecken gehört.





Aufgabe A3

Aufbauend auf der eben erstellten Datei erzeugen Sie die Punkte durch das folgende Zufallsverfahren, das die Perlin-Noise-Lib einsetzt. Definiert eine Größe der Fläche und erzeugt die einzelnen x-, y- und z-Werte nach dem beschriebenen Verfahren und fügt sie dem points-Array hinzu (Achtung, der Code ist unvollständig und kann nicht unmittelbar einkopiert werden, zudem wird durch die Größe der Fläche und die gewünschte Anzahl der Noise-Punkte die Rechenzeit je nach Hardware erheblich erhöht). Erstellen Sie dazu die eben bearbeitete Datei unter dem Namen `L3_3_Geometrie-erstellen_Aufgabe.html` ab.

```
let size = { x: 200, z: 200 };

x = THREE.Math.randFloatSpread(size.x);
z = THREE.Math.randFloatSpread(size.z);
y = noise.perlin2(x / size.x * 5, z / size.z * 5) * 40; // die letzte Zahl entspricht der Ausprägung der Höhen und Tiefen, je größer, desto ‚gebirgiger‘.
```

Der Rest kann dann wie bereits gezeigt umgesetzt werden.

Fügen Sie zusätzlich eine Kugel über der Fläche hinzu, die in einem begrenzten Bereich auf- und absteigt. Sorgen Sie für einen Schattenwurf der Kugel auf der Oberfläche.

Siehe dazu auch die MP4-Datei, in der die animierte Darstellung einen besseren Eindruck von der geforderten Lösung vermittelt.

Näheres zur Delaunator-Lib siehe auch <https://observablehq.com/@redblobgames/delaunator>;

