



## 5.1 Bewegung mit Kollisionen

### Einfache Kollisionserkennung

Wir haben inzwischen eine ganze Reihe verschiedener Körper (Objekte) modelliert, auch wenn es längst nicht alle fertigen Geometrien, die in three verfügbar sind, waren. Wir haben die Körper in Bewegung gesetzt und haben dabei verschiedene Bewegungsmuster kennen gelernt. Wäre es nicht toll, wenn die auch aneinander abprallen würden, also auf eine Berührung reagieren würden?

Es gibt eine ganze Reihe Physik-Bibliotheken für three, die aber aufgrund ihrer Vielfalt eine eigene Unterrichtsreihe benötigen. Daher basteln wir uns hier eine einfache Kollisionserkennung, mit der wir ein Abprallen schön modellieren können. Wir müssen eine Kollision bei jeder Aktualisierung der Bewegung in der Animation prüfen.

Dazu erstellen wir uns eine eigene Funktion, die wir in der Animationsfunktion aufrufen.

```
function detectCollision() {  
  ..  
}
```

Wider Erwarten stehen uns die Flächen als ‚fester‘ Gegenstand nicht zur Verfügung, wir haben nur die Vertices (auch die Physik-Bibliotheken interpolieren Flächen nur).

Wir suchen uns ein Objekt aus, aus dessen Sicht wir die Kollision berechnen. Dafür wählen wir am sinnvollsten das sich bewegende Objekt.

Also müssen wir alle Vertices dieses sich bewegenden Objektes durchgehen, ob eine Kollision mit einem anderen Mesh stattfindet. Wir nehmen dazu die Szenerie von der letzten Übung mit der Kugel (im weiteren nur als Ball bezeichnet) und der unregelmäßigen Oberfläche zum Abprallen.

Starten wir mit einer Kopie der aktuellen Ballposition. Dann müssen wir uns – da es sich um eine BufferGeometrie handelt – die einzelnen Koordinaten aus dem Array (siehe eigene Geometrien erstellen) extrahieren. Die kopieren wir dann mit der Methode fromBufferAttribute in den eigens erstellten vector.

```
let originPoint = ball.position.clone();  
let position = ball.geometry.attributes.position;  
let vector = new THREE.Vector3();  
for (let i = 0, l = position.count; i < l; i++) {  
  
  vector.fromBufferAttribute(position, i);;
```

In den nächsten Schritten kommt etwas ausgeprägt räumliche Geometrie zum Einsatz. Wir wollen an dieser Stelle aber nicht die dahinterstehende Mathematik ausarbeiten, daher soll es reichen, dass der Vektor, der ja vom Ursprung auf den Punkt ausgerichtet ist, über die 4D-Matrix des Balls in die Bewegungsrichtung ausgerichtet (transformiert) wird.

Der Raycaster ‚schickt‘ quasi einen Strahl (ray) aus, dessen Durchdringung dann mit der Methode intersectObjects ermittelt wird. Das Ergebnis ist eine Liste möglicher Kollisionen.



## 3D-Programmierung

### Kollisionen, Import und Export von Objekten

```
let globalVertex = vector.applyMatrix4(ball.matrixWorld);
let directionVector = globalVertex.sub(ball.position);
let ray = new THREE.Raycaster(originPoint, directionVector.clone().normalize());
let collisionResults = ray.intersectObjects(collidableMeshList);;
```

Dieses Array von berechneten Kollisionen können wir noch ein bisschen Filtern. Aber zunächst: wo kommt die collidableMeshList her?

Die müssen wir erstellen! Wir definieren ein Array mit eben diesem Namen und fügen ihr dann jedes Objekt, das geprüft werden soll, hinzu.

```
var collidableMeshList = [];
..
collidableMeshList.push(mesh);
..
```

Nun werten wir das Array collisionResults aus. Wir prüfen zunächst, ob überhaupt Werte im Array sind. Dann nehmen wir das erste Element (der Strahl könnte ja mehrere hintereinander liegende Meshes durchdringen). Damit wir den directionVector verwenden können, wurde er normalisiert (siehe oben).

```
if (collisionResults.length > 0 && collisionResults[0].distance < directionVec-
tor.length()) { .. // jetzt mach was damit
```

Wir müssen nun überlegen, welche Bewegung wir modifizieren wollen. Als ersten Schritt könnten wir die Auf- und Ab-Bewegung einfach umkehren (also die Abwärtsbewegung umkehren). Damit prallt der Ball quasi wieder nach oben ab.

### Aufgabe 1

1. Starte mit der Lösung der Aufgabe mit dem generierten Delaunator-Ground (L3\_3\_Geometrie-erstellen\_Aufgabe.html) und speichere sie unter dem Namen **L5\_1\_Kollisionen-detektieren\_1.html** ab. Implementiere die neue Funktion in der Datei und Sorge dafür, dass sich der Ball nach der Kollision mit dem unregelmäßigen Boden wieder nach oben bewegt.

### Reflektion im Winkel

Dass ein Ball an einer schrägen Fläche genau wieder nach oben springt, entspricht nicht der Realität. Wie können wir es realer machen?

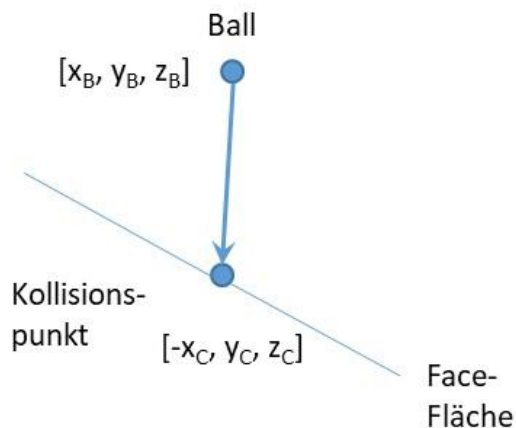
Interessanterweise ist ein Element der Liste collisionResults ebenfalls ein Vektor mit einem x, y, und z-Wert. Bilden wir also z.B. die Differenz dieses Punktes (also des Punktes, an dem die Kollision stattfinden würde), zur aktuellen Position des Balles, so können wir eine echte Reflexion an dem Material abbilden (der Ball spring wie erwartet an dem Material im Winkel ab).



```
// Vektor zwischen Ball und Kollisionspunkte - hier nur x  
let dx = ball.position.x - collisionResults[0].point.x;  
  
//
```

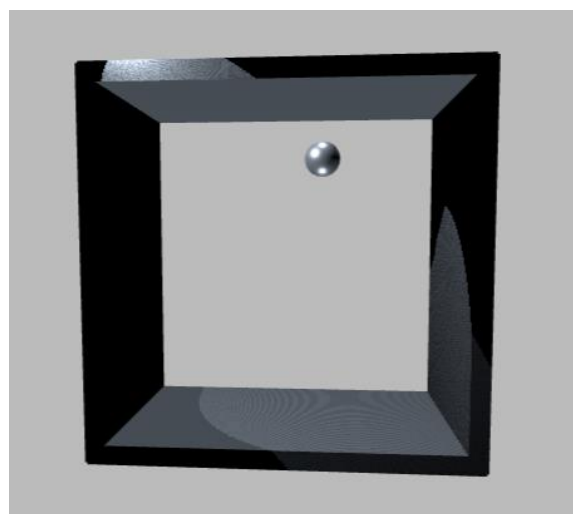
#### Aufgabe 2

Speichere die Datei von eben ab als `L5_1_Kollisionen-detektieren_2.html`. Implementiere die eben geschilderte Lösung mit dem Ball in der Funktion `detectCollision`.



#### Aufgabe 3

Noch interessanter wird es, wenn man statt der sehr unregelmäßigen Bodens 4 Flächen (4 dünne Würfel als Wände einbaut und nun den Ball an diesen abprallen lässt (solange der Ball sich nicht in z-Richtung, also aus dem Rahmen heraus bewegt, klappt das auch dauerhaft). Z.B. in der Art wie abgebildet. Realisiert euere Lösung in der Datei `L5_1_Kollisionen-detektieren_3.html`. Zur Veranschaulichung gibt es in den Materialien ein kurzes Video zu dem gewünschten Verhalten (Ball\_im\_Viereck\_Aufgabe.mp4).





## 5.2 Extern erstellte Objekte importieren

Nachdem wir gesehen haben, welcher Aufwand hinter ‚von Hand‘ erstellten Objekten steckt, dürfte klar sein, dass es möglich sein muss, sich anderer Werkzeuge zu bedienen, um Objekte zu erstellen.

Auf den Webseiten von three wird explizit Blender erwähnt, ein Open Source Programm zur Erzeugung von 3D Objekten und Szenarien. Auf das Programm soll hier nicht weiter eingegangen werden, nähere Informationen sind unter <https://www.blender.org/> zu finden.

Aus einem derartigen Programm heraus können Objekte (Objekte soll hier als Sammelbegriff für Einzelobjekte oder aber auch als Sammlung von Objekten stehen) als glTF exportiert werden. GLTF steht für Graphics Language Transmission Format und hat sich inzwischen zum Quasi Standard für den Austausch von 3D entwickelt. GLTF-Dateien können in einer Standardform (Endung .gltf) oder in einer binären, komprimierten Form (Endung \*.glb) auftauchen.

- Die Standard .gltf Dateien sind unkomprimiert und können zusätzliche binäre Dateien (Endung .bin) benötigen (meist gleicher Dateiname).
- Die binäre Form .glb schließt alle notwendigen Daten mit ein und kommt deswegen in nur einer Datei, kleinere Anpassungen sind jedoch komplizierter.

Der in ThreeJs zur Verfügung stehende GLTFLoader kann mit beiden Formaten umgehen, ohne dass er dafür angewiesen werden müsste.

Es existieren eine ganze Reihe frei zur Verfügung stehender 3D-Modelle, die zu Versuchszwecken importiert werden können:

- <https://github.com/mrdoob/three.js/tree/master/examples/models/glTF>
- <https://sketchfab.com/feed> (auch kostenlose Modelle nach einer Registrierung)

Da aber insbesondere für die Modelle aus der Sketchfab gilt, dass nicht abschließend geklärt werden konnte, ob die Objekte auch zur Weitergabe frei sind, ist in diesen Materialien nur ein Beispiel aus den three Beispielen mitgeliefert worden. Weitere Modelle – speziell auch aus der sketchfab kann man selbst herunterladen, wenn man es ausprobieren möchte.

In dem nachfolgenden Codeblock ist eine komplette Javascript-Funktion angegeben, mit der eine fest vorgegebene Datei geladen werden kann (die mitgelieferte Datei des XBot aus der three Bibliothek).

Möchte man die Funktion flexibler gestalten – beispielsweise indem man die Datei auf der HTML-Seite über ein Auswahlmenü auswählen kann, parametrisiert man einfach die Funktion.

Mit ‚onLoad‘ führen wir aus, was mit dem geladenen Objekt passieren soll, wenn der Ladevorgang fertig ist. In dem Beispiel wird das Objekt um nach auf der y-Achse nach unten verschoben und man kann sehen, wie man in der Objektstruktur an die einzelnen Meshobjekte herankommt, um – wie hier – den Schattenwurf zu ergänzen.

Um es noch Benutzerfreundlicher zu gestalten, kann man auch die einzelnen Meldungen, die momentan einfach auf der Konsole ausgegeben werden, einem HTML-Feld zuordnen.



```
function importGLTF() {
  let loader = new GLTFLoader();

  let onLoad = (gltf, position) => {
    let model = gltf.scene.children[0];
    model.position.copy(position);

    model.position.y -= 2;

    gltf.scene.traverse(function (node) {
      if (node.isMesh) {
        node.castShadow = true;
        node.receiveShadow = true;
      }
    });

    scene.add(model);
  };

  // Ladefortschritt
  let onProgress = (message) => { console.log("loading models"); };

  // Fehlermeldung an Konsole
  let onError = (errorMessage) => { console.log(errorMessage); };

  // Modell asynchron laden.
  let modelPosition = new THREE.Vector3(0, 0, 0);
  loader.load('XBot.glb',
    gltf => onLoad(gltf, modelPosition), onProgress, onError);
};
```

Ruft ihr die Funktion zum Import in der Funktion createObject auf, in der wir bislang immer die Objekte erzeugt haben, sollte sich das nebenstehende Bild ergeben.

### Aufgabe 1

Erstellt eine Datei `L5_2_Objekte_importieren.html`. Dazu könnt ihr eine beliebige, bisher erstellte Datei als Vorlage nutzen. Wir benötigen hier nur unsere bisherige Struktur mit grundlegenden Funktionen vom Anfang. In der Funktion createObject erzeugen Sie eine Bodenfläche für den Schattenwurf mit einer PlaneBufferGeometry und einem einfachen, hellen Material. Drehen und verschieben Sie die Fläche so, dass es einen Boden bildet.



Ebenso wie wir Objekte laden können, lassen sich natürlich auch eigene erstellte Objekte speichern.



## 3D-Programmierung

### Kollisionen, Import und Export von Objekten

Dazu speichern wir wieder die eben erstellte Datei unter `L5_2_Objekte_importieren-exportieren.html` ab.

Für den Export ergänzen wir unsere Funktionssammlung um die Funktion `exportGLTF` (siehe unten).

Hier haben wir eine ganze Reihe von Optionen, die wir unserem Export übergeben können, bzw. sollten. Alle möglichen Optionen sind in der Struktur `exportOptions` zusammengefasst. Die relevanten für uns sind gesetzt, die eher speziellen nur als Kommentar vorhanden.

```
function exportGLTF(input) {  
  let exportOptions = {  
    trs: false, // export objects from startposition  
    onlyVisible: true,  
    truncateDrawRange: true, // only objects within draw-range  
    binary: false, // modify: binary or JSON-Format  
    maxTextureSize: 4096 || Infinity, // To prevent NaN value  
    // additionally:  
    //animations - Array<AnimationClip>. List of animations included in export.  
    //forceIndices - bool. Generate indices for non-index geometry. Default false.  
    //includeCustomExtensions-bool. Export custom glTF extensions. Default false.  
  };  
  
  gltfExporter = new GLTFExporter();  
  gltfExporter.parse(input, function (result) {  
    if (result instanceof ArrayBuffer) {  
      save(new Blob([result], {type: 'application/octet-stream'}), 'scene.glb');  
    } else {  
      let output = JSON.stringify(result, null, 2);  
      save(new Blob([output], { type: 'text/plain' })), 'scene.gltf');  
    }  
  }, exportOptions);  
};
```

Zu beachten ist, dass die Datei in das Verzeichnis geschrieben wird, das der Browser als Download-Verzeichnis hat. Will man es wieder einlesen, muss man den Pfad entsprechend abändern oder die Datei zuerst wieder in das Arbeitsverzeichnis einkopieren.

Ein Problem bleibt dennoch: hat man die Szene als Ganzes exportiert und lädt sie dann wieder, hat man ein komplexes Gebilde aus mehreren Objekten (Oberfläche und Ball). Damit kann man den Ball nicht mehr bewegen, ohne ihn zu extrahieren.

## Aufgabe

Realisiert den Export zusätzlich und exportiert die Dateien.

### Optional:

Erstellt ein Menü, in dem die einzelnen Optionen ein- und ausgeschaltet werden können.