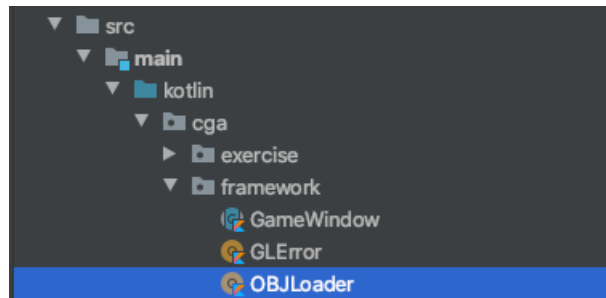


Laden von OBJ Dateien mit OBJLoader

Im Framework enthalten ist ein recht simpler Loader für .obj-Dateien. Die Klasse bzw. das Singleton heißt OBJLoader und befindet sich im Package cga.framework:



Die Klasse besitzt eine Methode namens loadOBJ(...), welche den Pfad zur OBJ-Datei, und zwei weitere Parameter entgegennimmt. Diese Parameter, recenterObjects und normalizeObjects, werden in den Fällen relevant, wenn OBJ-Dateien genutzt werden, die nicht um den Koordinatenursprung platziert sind und deren Vertexkoordinaten nicht normalisiert, sprich im Bereich von [-1; 1], liegen. Beide Parameter sind mit false als Default-Wert belegt.

War der Ladevorgang erfolgreich, wird ein Objekt vom Typ OBJLoader.OBJResult zurückgegeben. Wenn nicht, wird eine Exception geworfen.

Hier ein Ausschnitt der zeigt, wie man mithilfe des OBJLoaders ein Mesh aus einer OBJ-Datei erstellt:

```
//load an object and create a mesh
val res: OBJResult = OBJLoader.loadOBJ("assets/models/suzanne.obj")

//Get the first mesh of the first object
val objMesh: OBJMesh = res.objects[0].meshes[0]

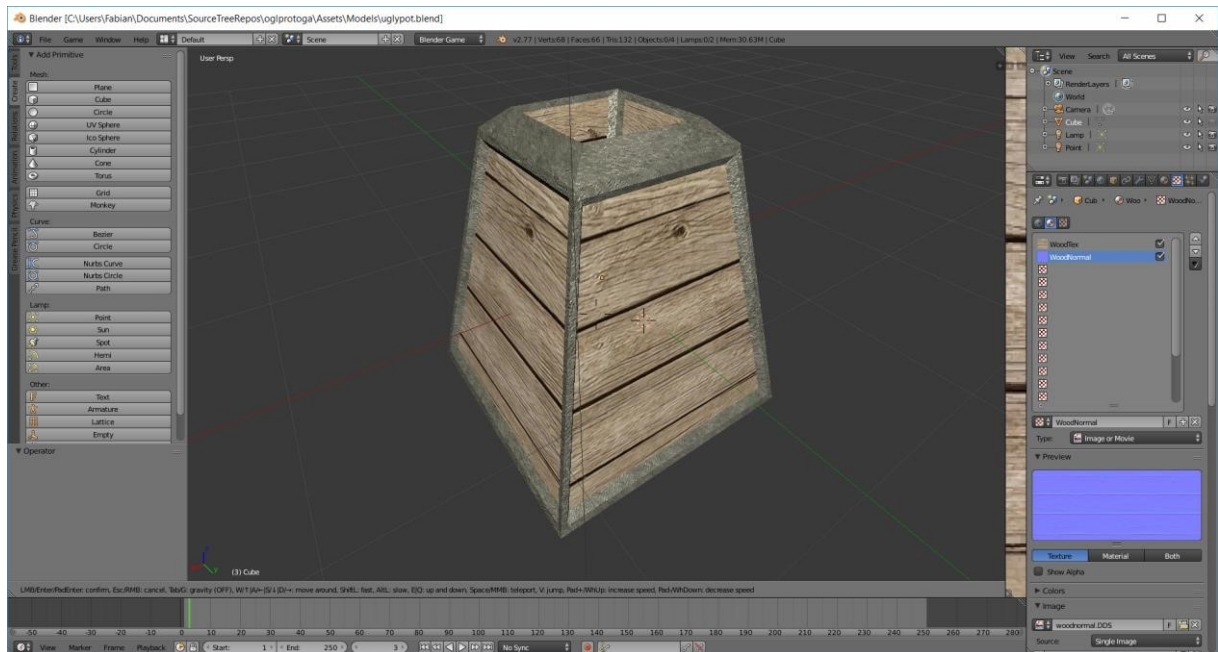
//Create the mesh
val stride: int = 8 * 4
val attrPos = new VertexAttribute(3, GL_FLOAT, stride, 0)           //position
val attrTC = new VertexAttribute(2, GL_FLOAT, stride, 3 * 4)       //textureCoordinate
val attrNorm = new VertexAttribute(3, GL_FLOAT, stride, 5 * 4)     //normalval

val vertexAttributes = arrayOf<VertexAttribute>(attrPos, attrTC, attrNorm)
val mesh = Mesh(objMesh.vertexData, objMesh.indexData, vertexAttributes)
```

Aufbau des OBJResult

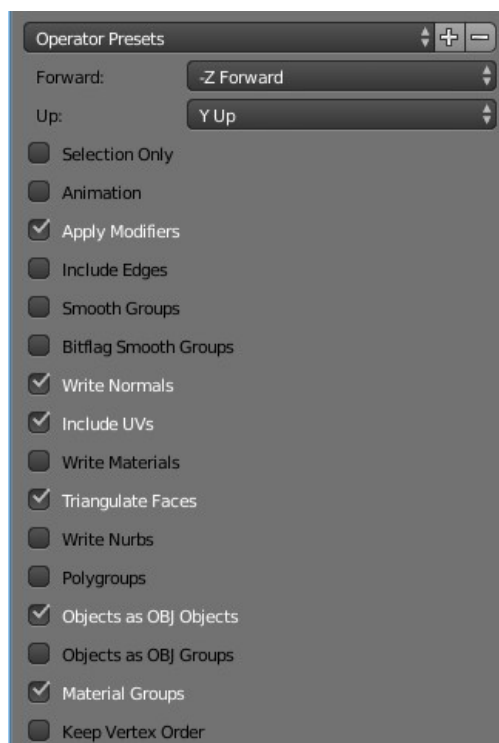
Ein OBJResult enthält eine Liste von OBJObjects, ein einzelnes OBJObject enthält wiederum eine Liste von OBJMeshes und ein OBJMesh enthält schließlich Vertices und Indices.

Zunächst aber die .obj-Datei, dann versteht man besser was am Ende herauskommt. Als Beispiel hier eine Kiste bestehend aus Holzwänden und einem Metallrahmen:



Einige Faces haben ein Holz-Material bekommen, die anderen ein Metall-Material.

Exportiert man in Blender dann nach .obj muss man ein paar Häkchen setzen:



Wichtig sind hier „Write Normals“ und „Write UVs“, wenn man möchte, dass die Normalen und Texturkoordinaten mit exportiert werden.

„Triangulate Faces“ sollte angehakt werden. Der OBJLoader trianguliert nicht automatisch, daher wäre das geladene Modell am Ende löchrig wenn dieser Haken nicht gesetzt wird.

„Objects as OBJ Objects“ bewirkt, dass für jedes einzelne Objekt eine „o“-Sektion in der .obj Datei angelegt wird.

„Material Groups“ fasst pro Objekt Faces mit gleichem Material zu Gruppen zusammen und packt diese in eine „g“-Sektion in der .obj Datei.

Hakt man die letzten beiden nicht an, bekommt man vom OBJLoader ein Model, das nur aus einem Objekt mit einem Mesh besteht. (Kann durchaus sinnvoll sein, wenn man Models mit nur einer Textur hat oder die Fähigkeit Texturatlant zu erstellen)

Materialien kann der Loader nicht verarbeiten und ignoriert diese. Mit gesetztem „Material Groups“ kann man dann aber im Programm den einzelnen Meshes wieder Materials zuweisen.

(Da Blender mittlerweile mehrere Updates erfahren hat, wird sich die Darstellung geändert haben.)

Exportiert man die Kiste von oben mit diesen Einstellungen, kommt so eine .obj Datei heraus:

```
# Blender v2.77 (xsub 0) OBJ File: 'uglypot.blend'  <- Kommentare
# www.blender.org
o Cube                                     <- Objekt
v 1.000000 -1.000000 -1.000000           <- Positionen
v 1.000000 -1.000000 1.000000
v -1.000000 -1.000000 1.000000
...
vt 0.5297 0.1097                         <- Texturkoordinaten
vt 0.8329 0.3654
vt 0.6441 0.3654
...
vn -0.0000 0.8132 0.5820                 <- Normalen
vn -0.9748 0.2230 -0.0000
vn 0.0000 0.2230 -0.9748
...
g Cube_Cube_Wood                         <- Mesh/Material-Gruppe
s off
f 15/1/1 50/2/1 52/3/1                   <- Faces der Gruppe
f 19/4/2 18/5/2 17/6/2
f 21/7/3 24/8/3 23/9/3
...
g Cube_Cube_Metal                       <- Mesh/Material-Gruppe
f 2/53/14 4/54/14 1/55/14                <- Faces der Gruppe
f 8/56/15 11/57/15 12/58/15
f 2/59/16 55/60/16 54/61/16
...
```

Das OBJResult enthält damit für jedes „o“ ein OBJObject und jedes OBJObject enthält für jedes „g“ ein Mesh. Das OBJResult bekommt zusätzlich den Dateinamen der neben „o“ steht, also hier „Cube“. Das OBJMesh erhält den Namen neben dem jeweiligen „g“, also „Cube_Cube_Wood“ bzw. „Cube_Cube_Metal“.

Zentrieren und Normalisieren von Objekten

In der Methode loadOBJ können die zuvor kurz umrissenen Parameter, recenterObjects und normalizeObjects, übergeben werden:

```
val res: OBJResult = OBJLoader.loadOBJ(
    "assets/models/suzanne.obj", //Pfad zur OBJ Datei
    true,                        //OBJObjects zentrieren
    true                         //OBJObjects normalisieren
)
```

Wird der zweite Parameter true gesetzt, werden die einzelnen OBJObjects mit allen dazugehörigen OBJMeshes neu um den Ursprung zentriert.

Der dritte Parameter, wenn true, bewirkt, dass die einzelnen OBJObjects gleichmäßig in NDC-Space-Größe herauf- bzw. herunterskaliert werden. Hatte ein Würfel z.B. zuvor die folgenden Eckpunkte:

```
(-10, -10, 10), (10, -10, 10), (-10, 10, 10), (10, 10, 10),
(-10, -10, -10), (10, -10, -10), (-10, 10, -10), (10, 10, -10)
```

Also ein 20x20 Würfel, hat dieser anschließend die Maße 2x2 und folgende Eckpunkte:

```
(-1, -1, 1), (1, -1, 1), (-1, 1, 1), (1, 1, 1),
(-1, -1, -1), (1, -1, -1), (-1, 1, -1), (1, 1, -1)
```

Statt die Parameter beim Laden zu übergeben, können diese beiden Operationen auch auf die einzelnen OBJObjects angewendet werden:

```
res.objects[0].recenter()
res.objects[0].normalize()
```

Verwenden der Daten

Die OBJMesh Objekte der jeweiligen OBJObjects enthalten die Rohdaten zum Mesh. Und zwar enthält jedes OBJMesh zum einen eine `ArrayList<OBJLoader.Vertex>` für die Vertexdaten und zum anderen eine `ArrayList<int>` für die Indexdaten.

Ein Vertex sieht wie folgt aus:

```
data class Vertex(val position: Vector3f, val texCoord: Vector2f, val normal: Vector3f)
```

Der Member position ist die Position des Vertex, normal die Normale. Der Member texCoord ist die Texturkoordinate, die erst später im Praktikum gebraucht wird.

Die Klasse OBJMesh bietet zwei Methoden um Vertex- und Indexdaten als Arrays zu erhalten:

```
val vertexData = objMesh.vertexData //get vertexdata
val indexData = objMesh.indexData    //get indexddata

//use plain data arrays to create a mesh
mesh = Mesh(vertexData, indexData, vertexAttributes)
```

Die float Daten in vertexData sind in der gleichen Reihenfolge wie in der Vertex-Klasse von oben abgelegt, also:

...

```
3 floats position    //Vertex n
2 floats texCoord
3 floats normal
3 floats position    //Vertex n+1
2 floats texCoord
3 floats normal
3 floats position    //Vertex n+2
2 floats texCoord
3 floats normal
...
```

Um die Vertexattribute zu definieren muss man das beachten. Möchte man also alle drei Attribute im Shader nutzen, legt man folgendes Attribut-Array an:

```
val stride: int = 8 * 4
val attrPos = new VertexAttribute(3, GL_FLOAT, stride, 0)           //position
val attrTC = new VertexAttribute(2, GL_FLOAT, stride, 3 * 4)        //textureCoordinate
val attrNorm = new VertexAttribute(3, GL_FLOAT, stride, 5 * 4)      //normalval

val vertexAttributes = arrayOf<VertexAttribute>(attrPos, attrTC, attrNorm)
```

Braucht man beispielsweise nur Position und Normale, lässt man das zweite Attribut einfach weg, die Offsets müssen aber immer so definiert werden, als wären alle Attribute dabei (die Rohdaten ändern sich ja nicht, nur deren Interpretation!):

Nachbearbeitung

Der OBJLoader bietet noch zwei Methoden zur Nachbearbeitung von geladenen Meshes:

```
val objMesh = res.objects[0].meshes[0]
OBJLoader.reverseWinding(objMesh)
OBJLoader.recalculateNormals(objMesh)
```

Sind in der OBJ-Datei keine Normalen enthalten, kann man sie mithilfe von `recalculateNormals(...)` neu berechnen lassen.

Mithilfe von `reverseWinding(...)` kann man alle Dreiecke im Mesh umdrehen, und so zwischen Clockwise und Counter-Clockwise Definition umschalten.