Zusatzübung 1 (3D-Daten und Formate)

a) 3D-Modellierungstools (z.B. Blender) unterstützen meist Standard-Formate wie OBJ, X3D usw. Diese sind nutzbar als Austauschformate.

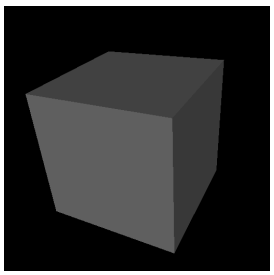
Ein einfaches OBJ-Beispiel ist rechts gezeigt. Dabei handelt es sich um ein sehr einfaches ASCII-Format zur Definition von Geometrieeigenschaften.

```
# comment: object cube starts
o Cube
v -1.00000 -1.00000  1.00000
v -1.00000 -1.00000 -1.00000
...
# vertex normals like this
#vn 0.707 0.000 0.707
...
# texture coords like this
#vt 0.500 1.000
...
# faces like this
f 1 5 7 3
...
# various possibilities:
#Vertex/TexCoord
#f v1/vt1 v2/vt2 v3/vt3
#Vertex/TexCoord/Normale
#f v1/vt1/vn1 v2/vt2/vn2 v3/vt3/vn3
#Vertex//Normale
#f v1//vn1 v2//vn2 v3//vn3
```

Diese werden über eine Liste mit Vertex-Positionen und Faces definiert. Letzteres sind die einzelnen Polygone des Polygonnetzes, die über Indizes angegeben werden, wobei man hier aufpassen muss, da die Zählung mit 1 (statt 0) beginnt. Optional kann man noch Normalen und Texturkoordinaten angeben.

Materialeigenschaften werden über ein zusätzliches MTL-File definiert, wo man Texturname, diffuse Farbe etc. angeben kann.

OBJ ist ein recht altes 3D-Format, das in der praktischen Anwendung aus mehreren Gründen leider nicht effizient ist. Zudem kann OBJ nur Geometrien verwalten und keine Objekt-Hierarchien.



Da OBJ für Menschen jedoch leicht verständlich ist, verwenden wir es für diese Übung, in der Sie eine OBJ-Datei zur Repräsentation eines Würfels erstellen sollen. Nutzen Sie für die Erstellung lediglich einen Texteditors. Zur Überprüfung des Ergebnisses können Sie z.B. den bei Windows eingebauten 3D-Viewer verwenden oder Blender (https://www.blender.org/).

Ein einfacher Würfel besteht aus sechs rechteckigen Flächen. Ein Beispiel für ein solches Face ist oben exemplarisch angegeben: `f 1 5 7 3`

Beim Import in eine Engine oder spätestens vor dem Rendern müssen jedoch die einzelnen Faces in Dreiecke zerlegt werden. Das für OBJ einzig notwendige Vertex-Attribut ist die Position. Jede Würfelseite benötigt zur Darstellung trotzdem eine eigene Flächennormale, wobei auch diese beim Import ausgerechnet werden kann.

b) In der nächsten Teilaufgabe ist ein 2D-Polygon in der *xy*-Ebene mit den 7 Vertices $P_0(1, 4.5, 0)$, $P_1(2.5, 1, 0)$, $P_2(4.5, 3, 0)$, $P_3(5, 1.5, 0)$, $P_4(6, 3.5, 0)$, $P_5(4, 5, 0)$, $P_6(2.5, 3.5, 0)$ in Dreiecke zu zerlegen (hier noch um $z = 0$ für diese 3D-Aufgabe ergänzt). Stellen Sie besagtes Polygon nun als OBJ-Datei dar. Denken Sie wieder daran, dass in OBJ die Indizes mit 1 beginnen.

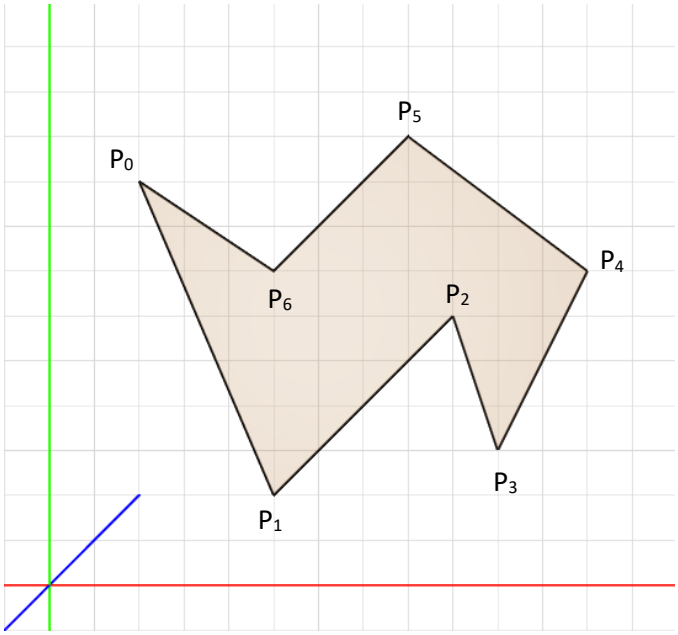
c) Zerlegen Sie das unten gezeigte Polygon mit den Eckpunkten $P_0(1, 4.5)$, $P_1(2.5, 1)$, $P_2(4.5, 3)$, $P_3(5, 1.5)$, $P_4(6, 3.5)$, $P_5(4, 5)$, $P_6(2.5, 3.5)$ in Dreiecke nach dem in den Folien vorgestellten Algorithmus. Gehen Sie dabei also so vor, wie man dies programmiertechnisch umsetzen würde (nur ohne Rechnung). Starten Sie mit P_0 und notieren Sie in nachvollziehbarer Form die dazu nötigen Schritte.

d) Überprüfen Sie nach dem in den Folien vorgestellten Inside-Test für Polygone, ob die Punkte P (3, 2) und Q (4, 2) innerhalb des Polygons liegen.

Gesucht ist die zeichnerische sowie zugehörige rechnerische Lösung für alle in Frage kommenden Kanten. Es ist also zu bestimmen, ob bzw. wie viele Schnittpunkte mit den Kanten vorliegen.

e) Bestimmen Sie die Bounding Box des rechts abgebildeten Polygons (durch Angabe von Min und Max als 2D-Punkte).

Berechnen Sie anschließend noch den Mittelpunkt M der Bounding Box.



f) Implementieren Sie in C/C++ die Funktion `int* tesselate(int index[], int n)`, um ein *konvexes* Polygon nach dem o.g. Algorithmus in Dreiecke zu zerlegen, wobei die Rückgabe des Ergebnisses über ein dynamisch angelegtes Feld mit angepassten Indizes erfolgt. Statt normalen Feldern dürfen Sie natürlich auch die Template-Klasse *std::vector* verwenden.

Das 2D-Polygon ist dabei in *main()* über zwei Arrays definiert: eines mit den Eckpunkten (x_i, y_i) namens `coords` sowie eines mit den zugehörigen Originalindizes (namens `index`), welche die Reihenfolge der Eckpunkte (unabhängig von der Koordinatenreihenfolge in `coords`) festlegen. Überlegen Sie, warum die Verwendung eines zusätzlichen Indexfeldes hier sinnvoll ist!

Testen Sie Ihren Algorithmus mit verschiedenen konvexen Polygonen, z.B. mit der konvexen Hülle des oben gezeigten 2D-Polygons (P_0, P_1, P_3, P_4, P_5). Das Feld `index` wäre dann wie folgt definiert: `int index[] = {0, 1, 3, 4, 5};` mit den Eckpunkten: `float coords[] = {1,4.5, 2.5,1, 4.5,3, 5,1.5, 6,3.5, 4,5, 2.5,3.5};`

Hinweis: lesen Sie sich die Aufgabenstellung genau durch, vielleicht benötigen Sie zur Umsetzung nicht alle Eingangsdaten (bzw. überlegen Sie, in welchem Fall man doch alle Daten benötigt).

Achtung, Eckpunkte speichert man in der Computergraphik zumeist in eindimensionalen Feldern, wobei die *x*-, *y*- und ggfs. *z*-Koordinaten je aufeinander folgen; der erste Punkt P_0 wäre damit also natürlich (1, 4.5). Zudem betrachtet man zumeist nur die Eckpunkte, welche in einem Index-Array gegeben sind. Überlegen Sie sich daher noch, wie man von den Indizes auf die jeweilige *x*- und *y*-Koordinate kommt (Hinweis: adressieren Sie zunächst den *x*-Wert).

g) Implementieren Sie in C/C++ die unten gezeigte Funktion zur Bestimmung der „Axis-Aligned Bounding Box“ eines Dreiecksnetzes. Der Datentyp *Vec3* soll dabei drei Float-Werte (für x, y, z) in einem Array abspeichern können. Im einfachsten Fall könnte der Typ *Vec3* folgendermaßen definiert sein: `typedef float Vec3[3];`

```
void calcBBox(const std::vector<Vec3> &mesh, Vec3 &min, Vec3 &max);
```