In [ ]:
```python
import igl
import polyscope as ps
ps.init()
MESH_PATH = "/home/z/Teaching/CSC317-Compuer-Graphics/computer-graphics-mesh
V, F = igl.read_triangle_mesh(MESH_PATH)
print(f"{V.shape=}, {F.shape=}")
ps.register_surface_mesh("torus", V, F) # with face connectivity
# ps.register_point_cloud("torus", V)   # with only vertices, it is a point
ps.show()
```

```
[polyscope] Backend: openGL3_glfw -- Loaded openGL version: 4.6 (Core Profil
e) Mesa 23.0.4-0ubuntu1~22.04.1
V.shape=(24, 3), F.shape=(48, 3)
```

This notebook is for CSC317-Computer Graphics, Meshes tutorial only.

It is **NOT** an offical implementation guide for the course assignment.

Before we start, I personally would recommend you to implement A5 in the following order:

`src/write_obj.cpp`

Here is a comprehensive guide to the structure of a .obj file.

Hints:

- for every row of the matrix, print the row to one text line of the file, begins the line with what the row represents (e.g. "v", "vn", "f")
- do it for every matrix!
- be careful of the **number of columns**.

```cpp
// what you can do...
V.rows(); // would return # of vertices
F.rows(); // would return # of faces
V.cols(); // would return 3
UV.cols(); // would return 2

// you may not want to do this...
V.size(); // this would return the total number of elements in the
matrix (# of rows x # of cols)
```

`src/cube.cpp`

Literally just a cube.

Since it is a cube

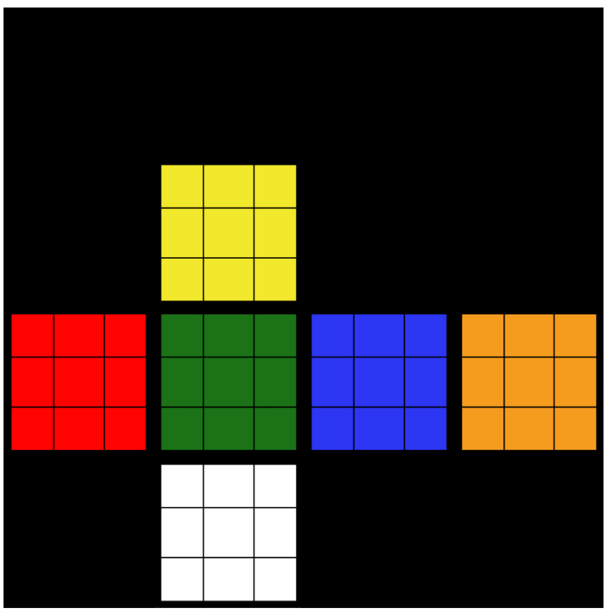
- you know exactly what's the position of each vertex (`V`)
- you know exactly what's the normal of each vertex (`VN`)

- you know exactly what's the texture coordinate of each vertex (`UV`)
- you know exactly what's the face connectivity of the cube (`F`)
- ...

Hint: in implementation it is just a long list of `V, VN, UV, F, ...`.

```
// you can fill in a matrix like this :)
V.resize(8,3);
V << 0,0,0,
     1,0,0,
     1,1,0,
     0,1,0,
     0,0,1,
     1,0,1,
     1,1,1,
     0,1,1;
// so you don't have to do V.row(0) = Eigen::Vector3d(0,0,0);
```

Consider the UV of the cube, it is just a "flatten" box, think of an origami cube.



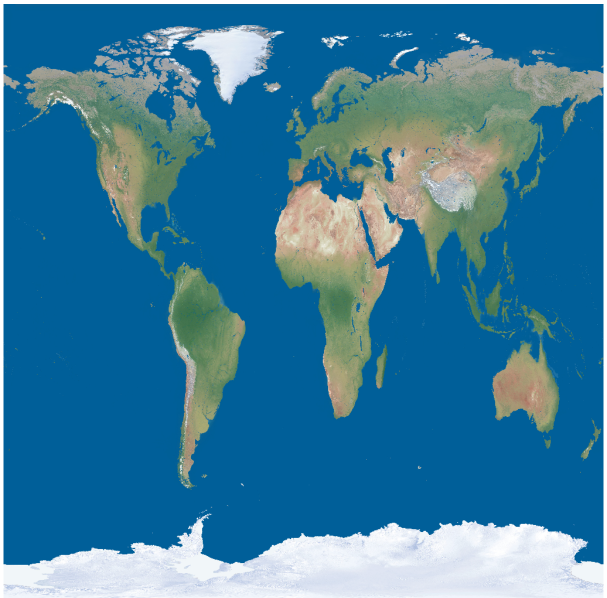
UV maps a 3D vertex to a 2D vertex on this image :)

`src/sphere.cpp`

Be creative, but as you can imagine it will be along the line of drawing polyline circles of different radius and height! Think of a globe, think of earch, think of **latitude and longitude**.

Hint:

- It is a good idea to index the vertex in a reasonable way so that it is easy to construct the connectivity matrix `F`.
- What's the normal defined on every vertex of the sphere? So easy lol.
- Consider the UV texture of the sphere as a world map (literally).



For every vertex of 3D coordinates (x, y, z) on our discretized sphere, we can map it to a 2D coordinate (u, v) on the world map following this expression:

$$u = \frac{\pi + \mathrm{atan2}(y, x)}{2\pi}, v = \frac{\pi - \mathrm{acos}(\frac{z}{||x||})}{\pi}.$$

`src/triangle_area_normal.cpp`

It should returns the normal of the triangle, but the length of the normal is the area of the triangle.

It is **NOT** a *normalized normal*.

Consider a triangle with vertices (p_0, p_1, p_2)...

$$(p_1 - p_0) \times (p_2 - p_0) = 2A \cdot \mathbf{n}$$

where A is the area of the triangle, $\mathbf{n}$ is the normalized normal of the triangle.

It now returns a normal with length $2A$, you know what to do.

`src/per_face_normals.cpp`

Since `src/triangle_area_normal.cpp` is defined for one triangle *face*, it is trivial to implement this function.

Hint: a recurring theme you will see in computer graphics when it comes to loop over faces is

```
# python pseudo code
# V = (# of verts, 3) double matrix
# F = (# of faces, 3) integer matrix
for f in F:
```

```
    p0 = V[f[0]]
    p1 = V[f[1]]
    p2 = V[f[2]]
    # do something with p0, p1, p2, the current triangle
```

`src/vertex_triangle_adjacency.cpp`

We build an adjacency list mapping every vertex index to a face (triangle) index.

Hint: consider the face loop shown above...it is a mapping maps every face (triangle) in F to **3** vertices in V.

How do you construct an inverse *look-up table* of that? (mapping every vertex u to a list of faces contains vertex u)

Why is it not a fixed size matrix like F? Why is it a adjacency list?

It is basically an super easy LeetCode question.

`src/per_vertex_normals.cpp`

We can also define a normal for each vertex, by averaging the normals of the faces that contains the vertex.

So you are going to use `src/vertex_triangle_adjacency.cpp` to fine the faces that contains the vertex, and then average the normals of those faces.

Math for every vertex be like:

$$\mathbf{n}_v = \frac{\sum\limits_{f \in Nb(v)} A_f \cdot \mathbf{n}_f}{||\sum\limits_{f \in Nb(v)} A_f \cdot \mathbf{n}_f||}$$

where A_f is the area of the face (triangle) f, $\mathbf{n}_f$ is the normalized normal of the face (triangle), $Nb(v)$ is the set of faces that contains vertex v.

Think what should be normalized! Hint: loops over neighboring faces of the vertex, and sum up the area-weighted normals of the faces. Then normalize the sum.

`src/per_corner_normals.cpp`

- Per vertex normal averaging over neighboring faces makes the normal on the sharp edge of the mesh looks smooth (see README picture).
- Per face normal is defined over one triangle, so global smoothness is not guaranteed (see README picture).

So we can improve per vertex/face normal by defining a normal for every face (triangle) corner.

By setting a threshold, if the angle between two neighboring faces is greater than a threshold angle, we should not average the normals of those two faces.

We define the threshold as a scale value ϵ, we then have per corner normals defined as...

$$\mathbf{n}_{f,c} = \frac{\sum\limits_{g \in Nb(c) | \mathbf{n}_f \cdot \mathbf{n}_g > \epsilon} A_g \cdot \mathbf{n}_g}{|| \sum\limits_{g \in Nb(c) | \mathbf{n}_f \cdot \mathbf{n}_g > \epsilon} A_g \cdot \mathbf{n}_g ||}$$

where A_g is the area of the face (triangle) g, $\mathbf{n}_g$ is the normalized normal of the face (triangle), $Nb(c)$ is the set of faces that contains triangle f corner c.

Hint:

- why only consider the pairs where the dot product is greater than ϵ? Consider the case where two vectors are orthogonal, the angle between them is 90 degree, and the dot product is 0.
- the function takes an angle (in degree), how do you convert the degree angle to a dot product threshold ϵ?
- or how do you convert the dot product threshold to a degree angle?
- think about when to normalize...
- basically we are averaging face normal unless it is too crazily different
- think how you can the neighboring faces of a **face** (with `src/vertex_triangle_adjacency.cpp`)

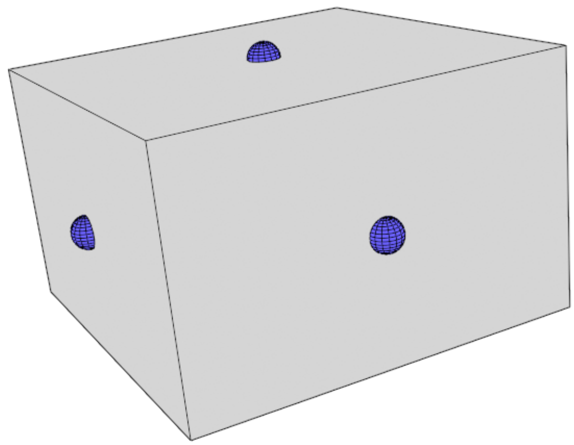
`src/catmull_clark.cpp`

Welcome to the most difficult part of the assignment 😔

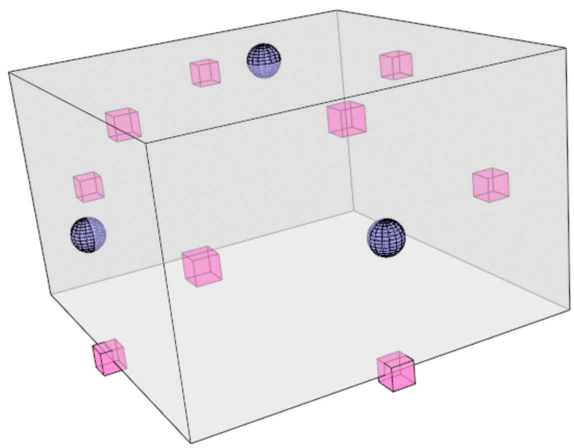
Refer to the lecture slides or wiki for the algorithm.

For every level of subdivision:

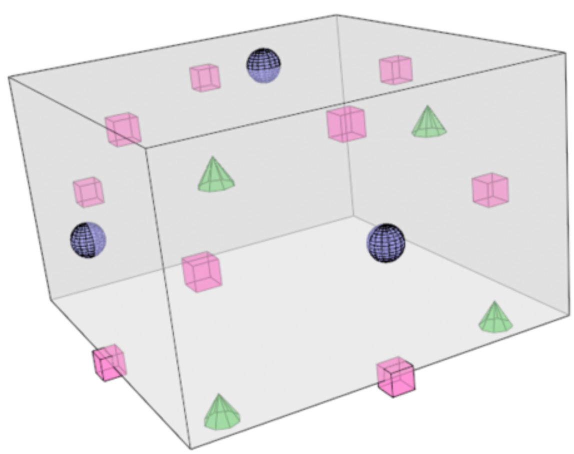
- Step 1: easy, average per face (blue)



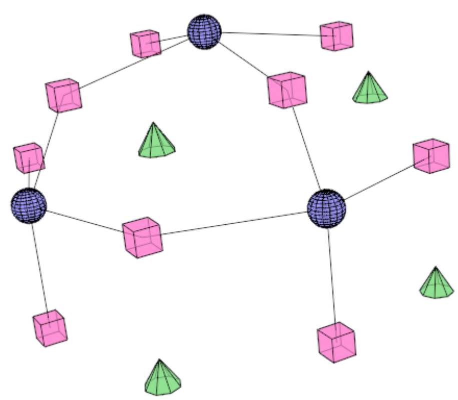
- Step 2: easy, average per edge and their neighboring 2 faces (pink)



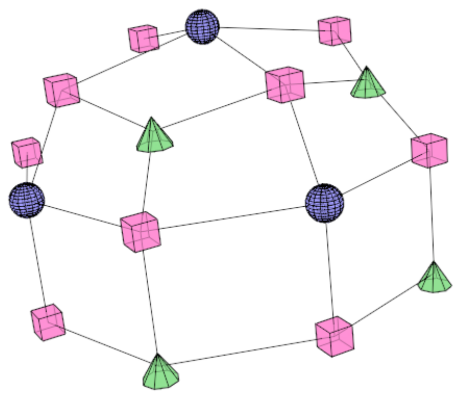
- Step 3: move original vertices to new places...



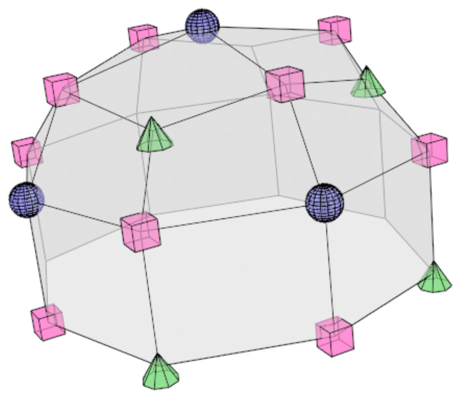
- Step 4: add new edges between face points (blue) and edge points (pink). (Note that in lecture this is step 3, but 3/4 can be in whatever order)
  - since new edge points are defined with ONE edge and the neighboring TWO faces of the edge, you need to construct a inverse look-up table to map edge to faces.
  - since every edge point *uniquely* maps to one edge, every edge *uniquely* maps to two faces, and every face *uniquely* maps a face point (from step 1), you can now connect the face points (from step 1) to the edge point (from step 2).
  - you can also go the other way -- since every face point is *uniquely* maps to a face, and every face *uniquely* maps to 4 edges, you can now connect the face points to the edge points.
  - Basically for every edge, it uniquely defines a three point pair.



- Step 5: connect moved points (green) with edge points (pink).
  - similar to step 4, every edge point is uniquely maps to an edge, and every edge is uniquely maps to two vertices (two moved points).
  - you can also go the other way.



- Step 6: done, move to the next iteration...



`Misc`

- `#include <igl/PI.h>` gives you the π constant with `igl::PI`.
- The dimension of the expected output is defined in header files, read header files!