



POLITECNICO  
MILANO 1863

# Meshes

# Meshes

Mixtures of glossy, bumpy, linear and curved surfaces characterize objects in nature.

3D assets are virtual representations of objects, encoded in a digital way.



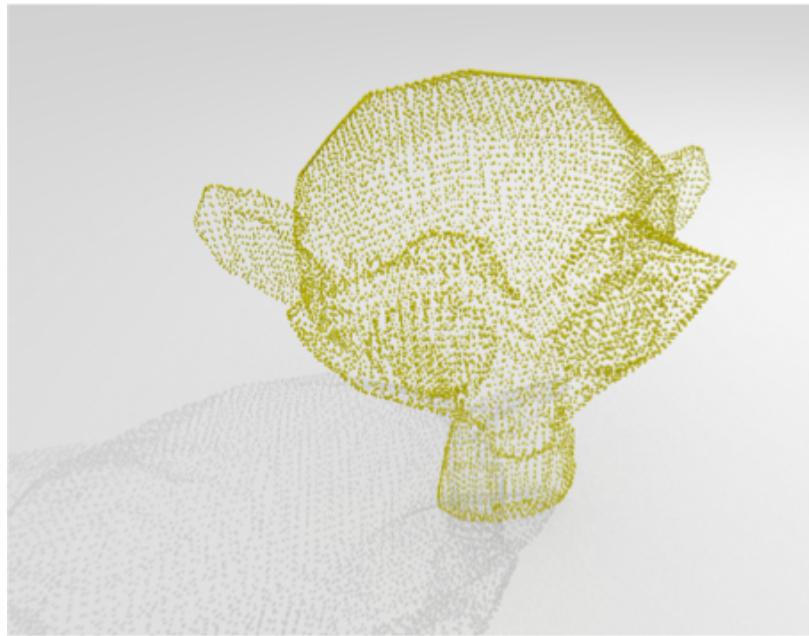
Cube with Materials

```
# This cube has a different material  
# applied to each of its faces.  
mtllib master.mtl  
v0.000002.000002.00000  
v0.000000.000002.00000  
v2.000000.000002.00000  
v2.000002.000002.00000  
v0.000002.000000.00000  
v0.000000.000000.00000  
v2.000000.000000.00000  
v2.000002.000000.00000  
# 8 vertices  
g front  
usemtl red  
f1 2 3 4  
g back  
usemtl blue  
f8 7 6 5  
g right  
usemtl green  
f4 3 7 8  
g top  
usemtl gold  
f5 1 4 8  
g left  
usemtl orange  
f5 6 2 1  
g bottom  
usemtl purple  
f2 6 7 3  
# 6 elements
```

# Meshes

We have seen how to find the position on screen for a point in the 3D space.

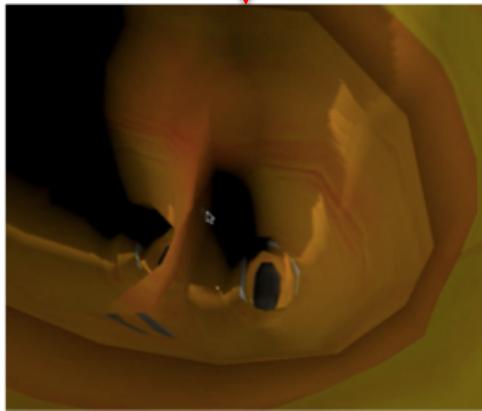
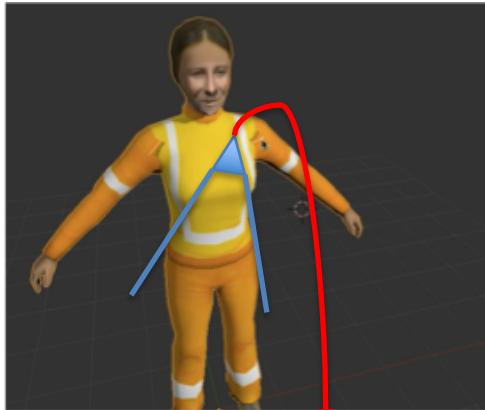
However usually objects are not encoded as a set of unconnected points (*point clouds*), because they would require too many resources to obtain an apparently solid object.



With **9646** vertices:  
the object still looks  
very “empty”

# Meshes

As a first step, every solid object is stored only by its boundaries: what is inside the surface it is usually ignored.



# Meshes

Object geometry encodings are based on mathematical models that represent surfaces through a set of parameters.

*Computational geometry* is the science that studies the best ways to mathematically describe surfaces.

Many approaches have been defined in the literature. The most common are:

- Meshes (*polygonal surfaces*)
- Hermite surfaces
- NURBS (*Non-Uniform Rational B-Splines*)
- HSS (*Hierarchical Subdivision surfaces*)
- Metaballs

# Meshes

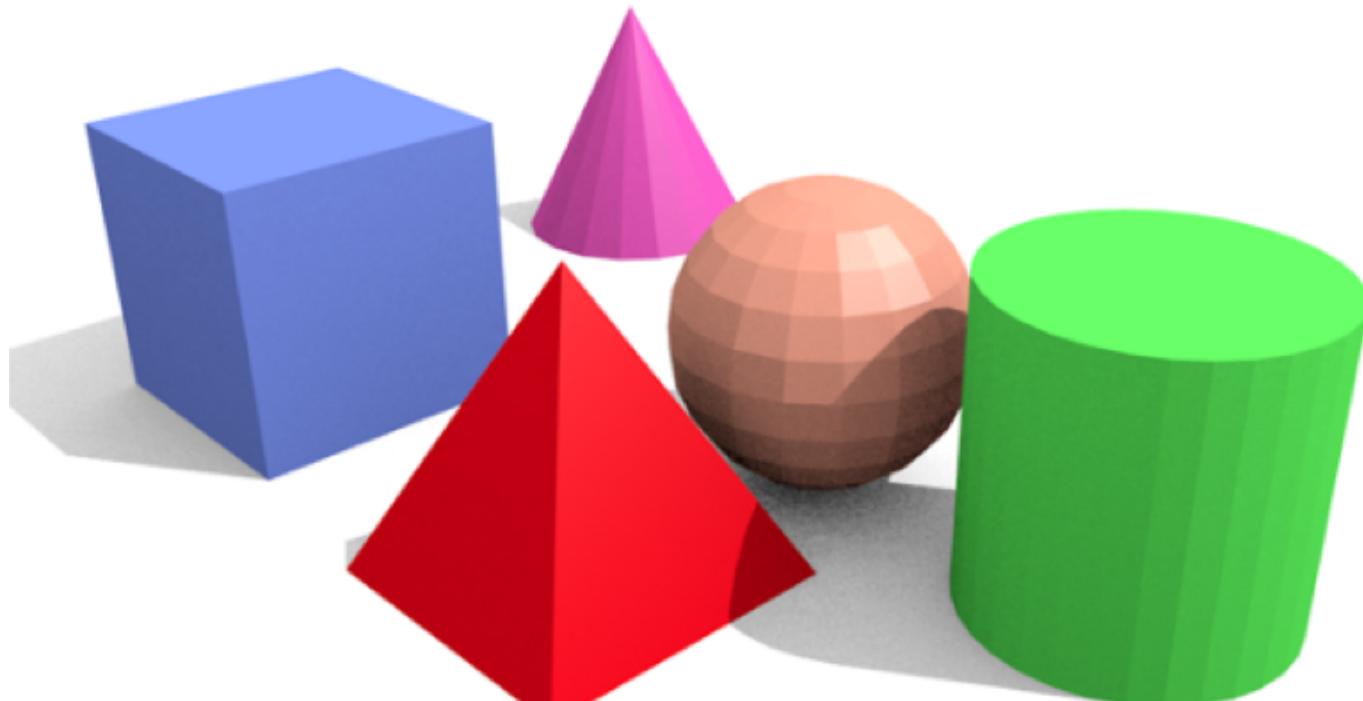
Each mathematical model has its own features and limitations.

However all the models are converted to meshes (polygonal surfaces) when rendered.

For this reason we will focus on meshes only.

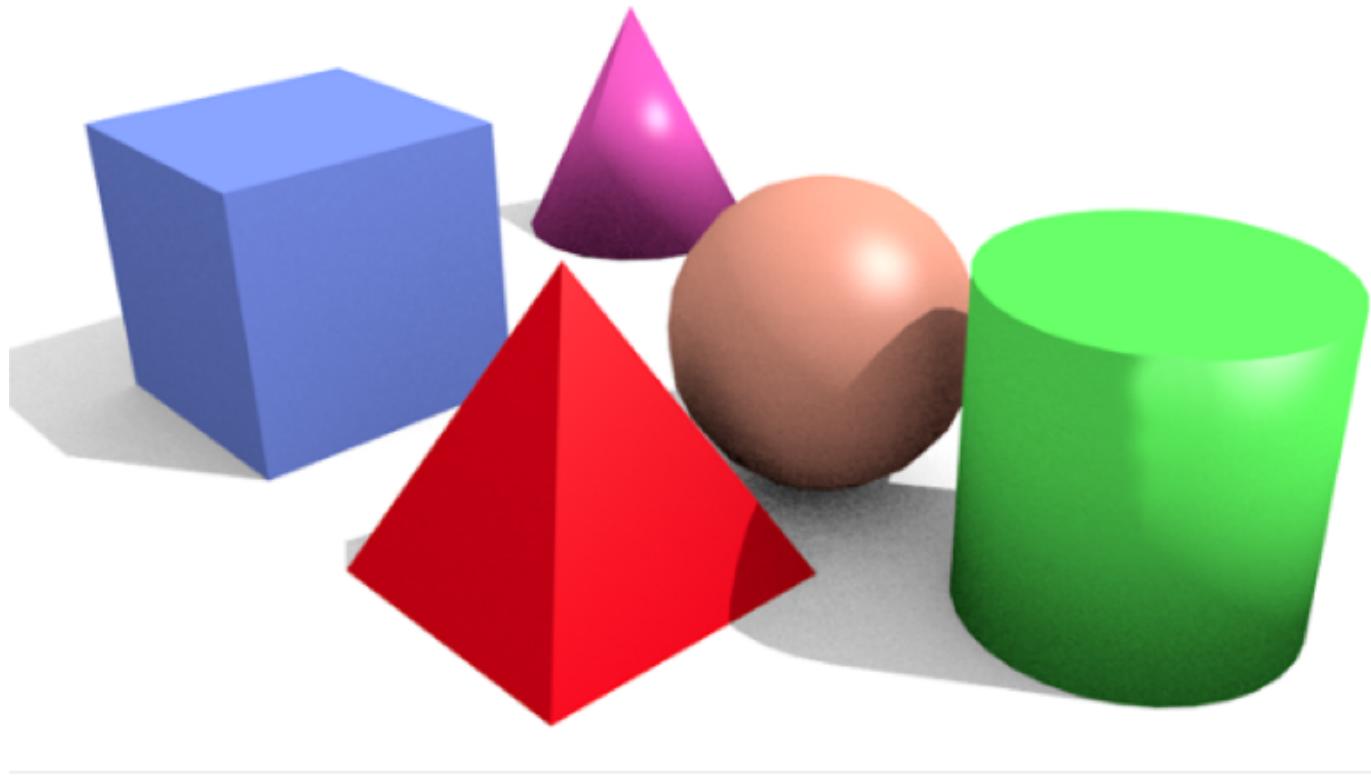
# Meshes

*Polygonal surfaces* are objects that can be described by a set of contiguous polygons.



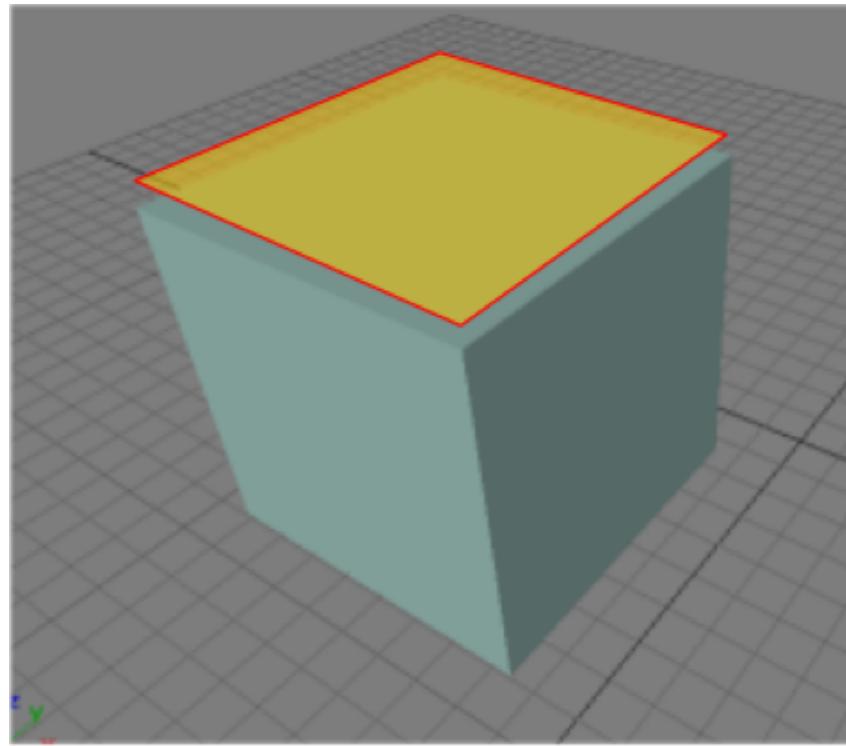
# Meshes

Thanks to special rendering features, which will be presented in the following lessons, polygonal surfaces can be used to properly approximate curved surfaces as well.



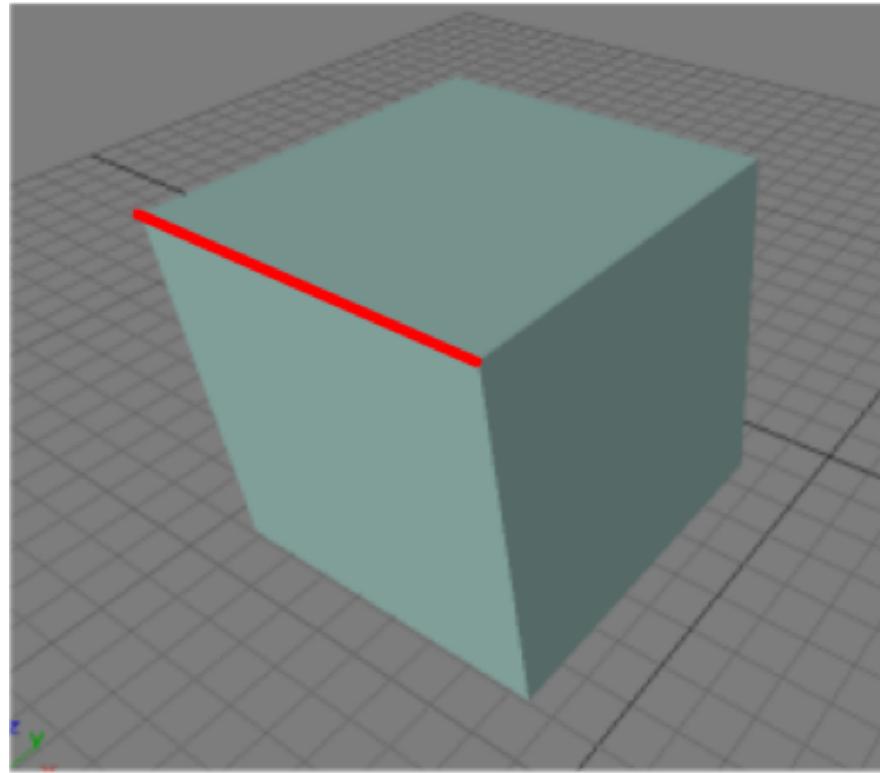
# Meshes

A polygon that describes a planar portion of the surface of an object is called a *face*.



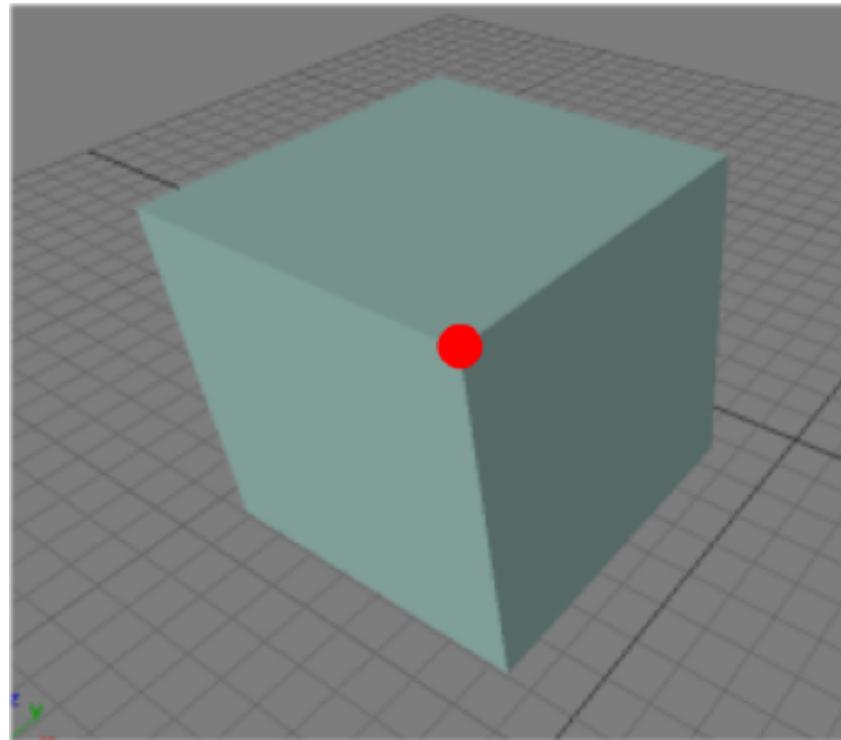
# Meshes

Sides of the polygons are called edges. They commonly correspond to the intersection of two faces.



# Meshes

Vertices correspond to the starting and ending points of the edges. Exactly two vertices delimit an edge, and in proper solids at least three faces intersect at a vertex.



# Meshes

If every edge is adjacent to exactly two faces, the surface has a special topology known as *2-manifold*.

*Non-2-manifold* surfaces usually represents non-physical objects: if they are used, special care should be applied to correctly render them.

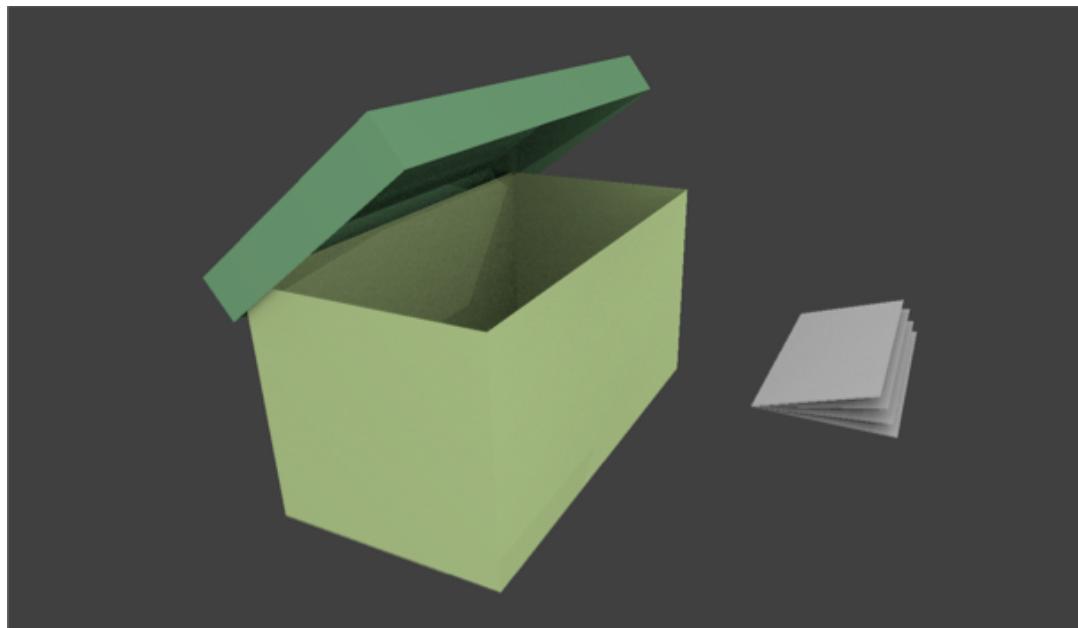
Examples of non 2-manifolds are solids with *holes* and *lamina-faces*.



# Meshes

Sometimes non-2-manifold objects are used to reduce the number of polygons necessary to encode very thin objects, or to obtain special effects (e.g. an open box, a magazine).

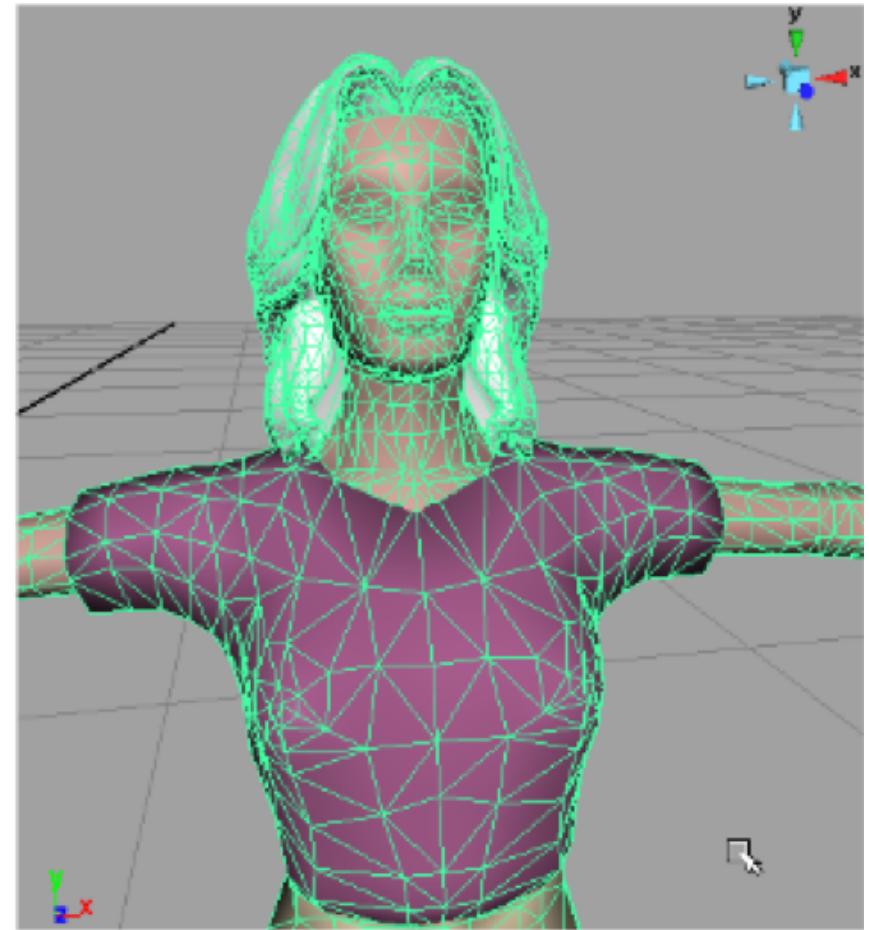
Algorithms like the *backface-culling* (which will be presented later), however, will not work on non-2-manifold objects.



# Meshes

Each polygon can be reduced to a set of triangles that shares some edges.

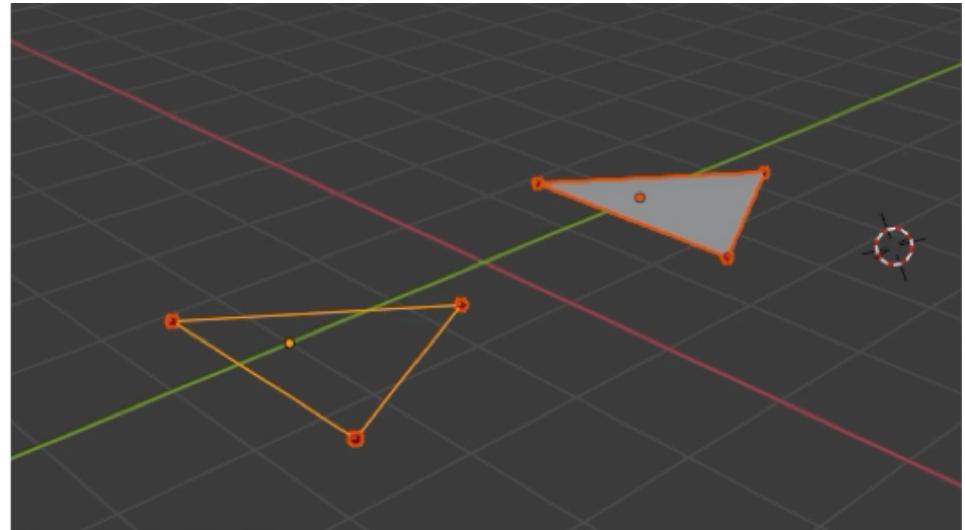
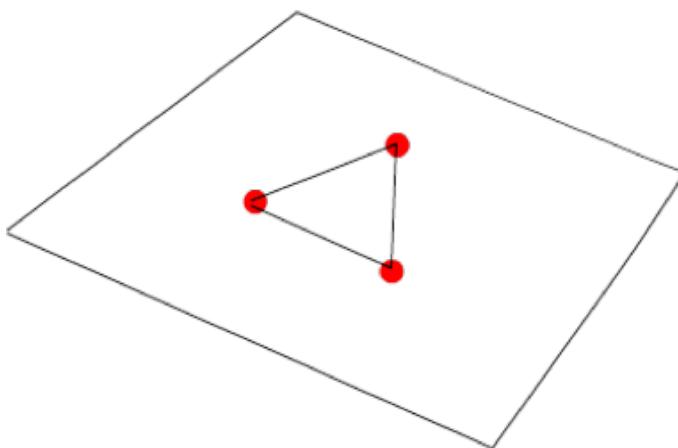
A set of adjacent triangles is called a *Mesh*.



# Meshes

We know that three unaligned points identify a unique plane in the space.

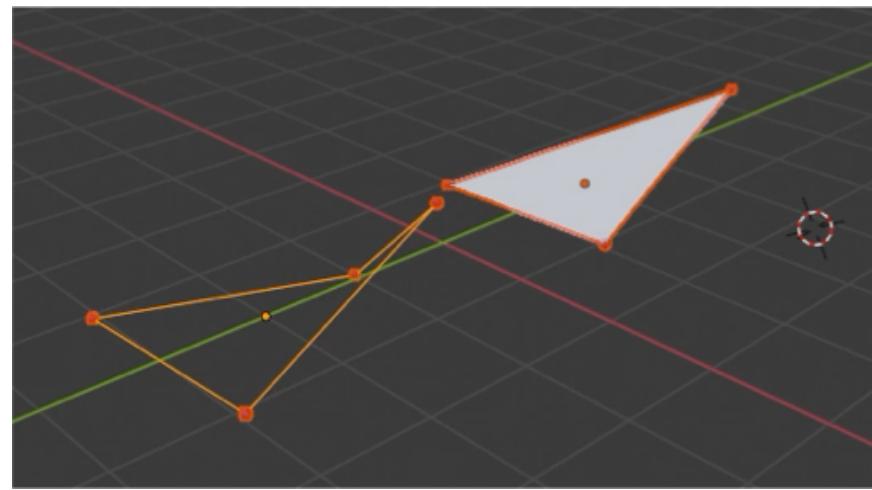
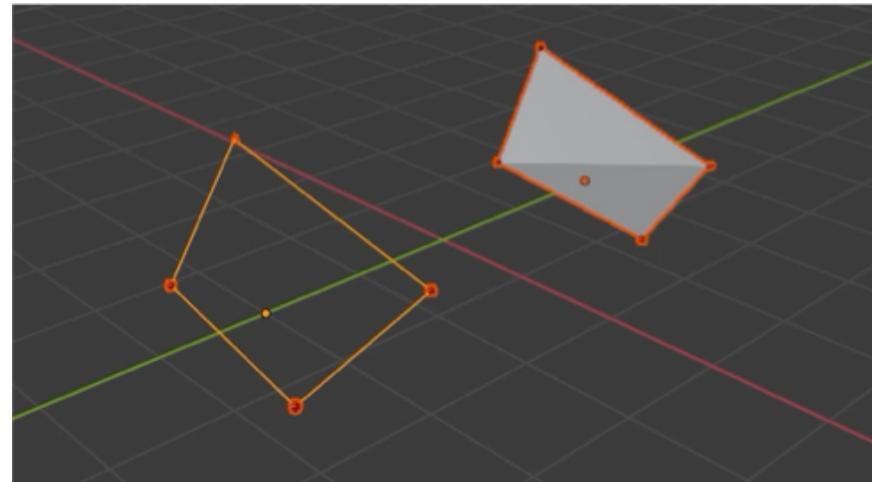
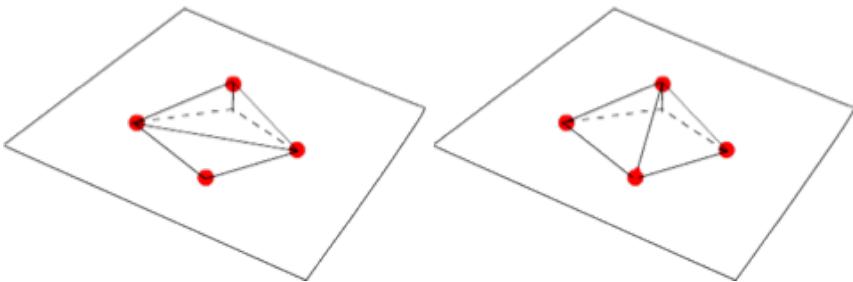
Three unaligned points define a triangle.



# Meshes

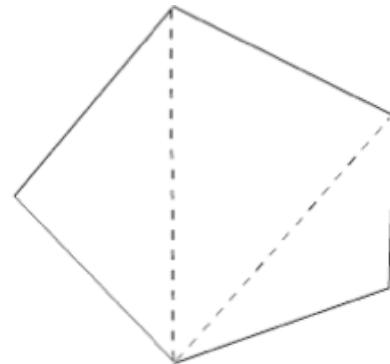
Every polygon with more than three vertices, might not identify a single a planar surface.

This means that there could be several different ways of connecting  $n > 3$  points to represent a surface in the space.

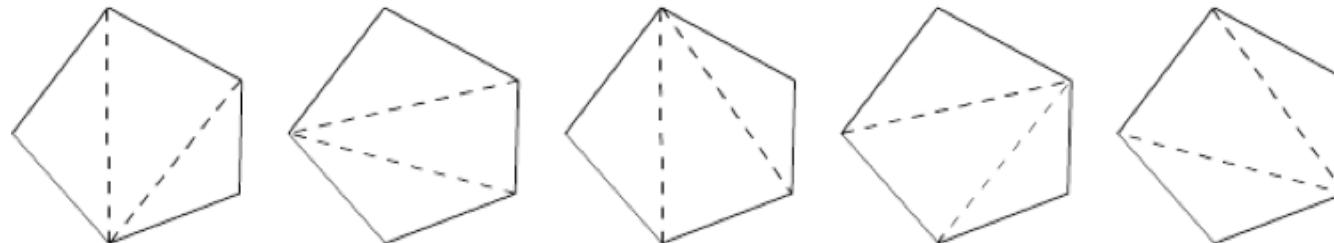


# Meshes

Every polygon (planar or not), is thus reduced to a set of triangles, using a process known as (polygon) *tessellation*.

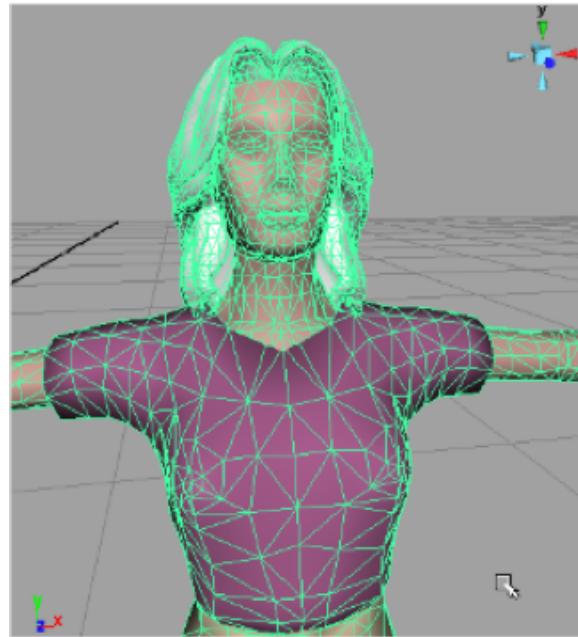
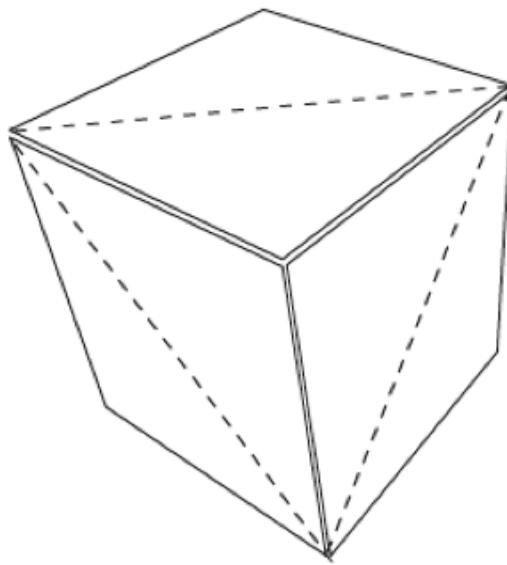


Polygon tessellations are not unique. Several different tessellation might be defined for each polygon with more than 3 sides.



# Meshes

A mesh representation of an object stores its surfaces with the set of polygons that delimits its boundary.  
The boundary polygons are then converted to a set of contiguous triangles that shares some edges.

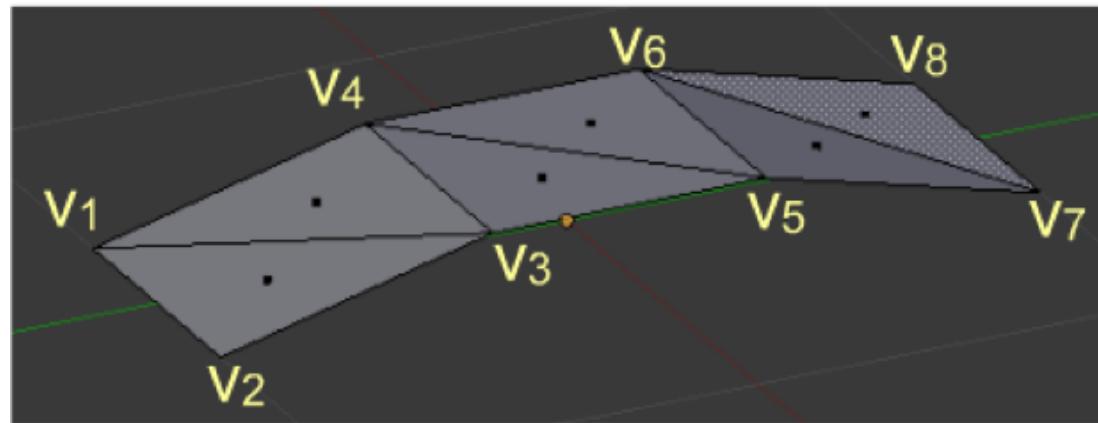


# Mesh encodings

Meshes encoding are based on set of vertices.

$V_1 \{0, 0, 0,$	$0, 0, 1,$	$V_2$
$V_3 \ 1, 1, 1,$	$1, 1, 0,$	$V_4$
$V_5 \ 2, 1, 1,$	$2, 1, 0,$	$V_6$
$V_7 \ 3, 0, 1,$	$3, 0, 0\}$	$V_8$

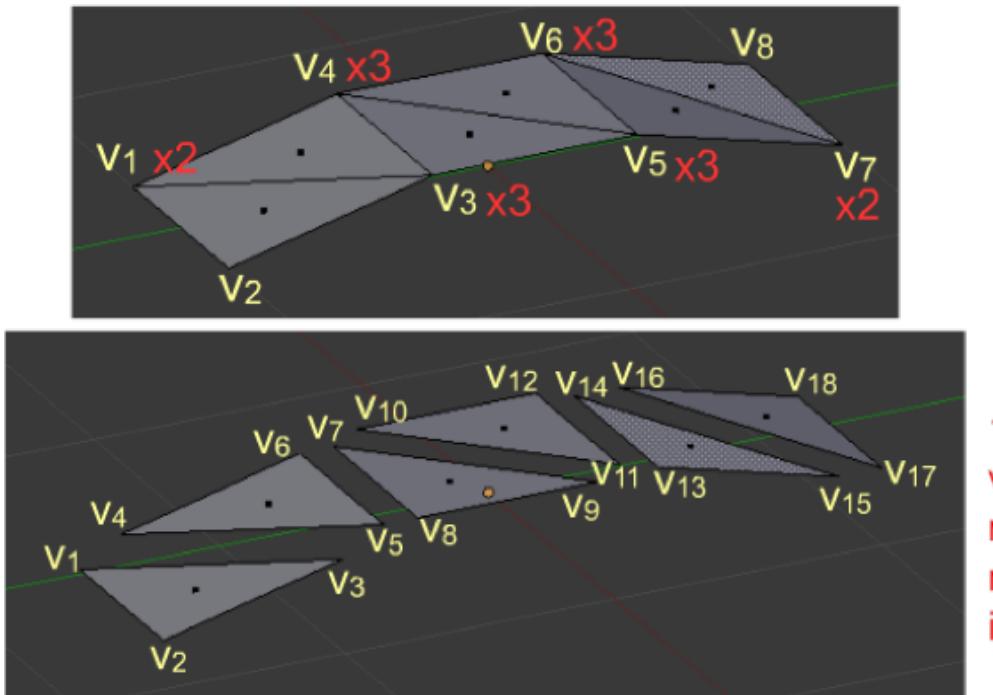
The rendering engine uses such vertices to determine the end points of the triangles that compose the mesh.



# Mesh encodings

It can be easily seen that most of the triangles inside a mesh share a lot of vertices.

It is important to exploit this feature to reduce the amount of memory required to encode an object.



# Mesh encodings

Several different types of mesh encoding have been defined in the literature.

However, only two of them are standard in Vulkan:

- *Triangle Lists*
- *Triangle Strips*

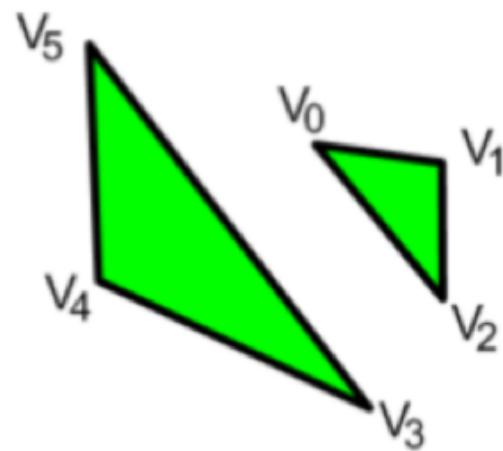
A third encoding, *Triangle Fans*, which was standard in OpenGL, has been made optional in Vulkan, so some graphic adapter might no longer support it: for this reason we will not cover it in this course.

# Triangle lists

*Triangle lists* encode each triangle as a set of three different coordinates. They do not re-use any vertex.

They are used to encode triangles that are unconnected.

To encode  $N$  triangles,  $3N$  vertices are required.



2 Triangles =  
6 Vertices

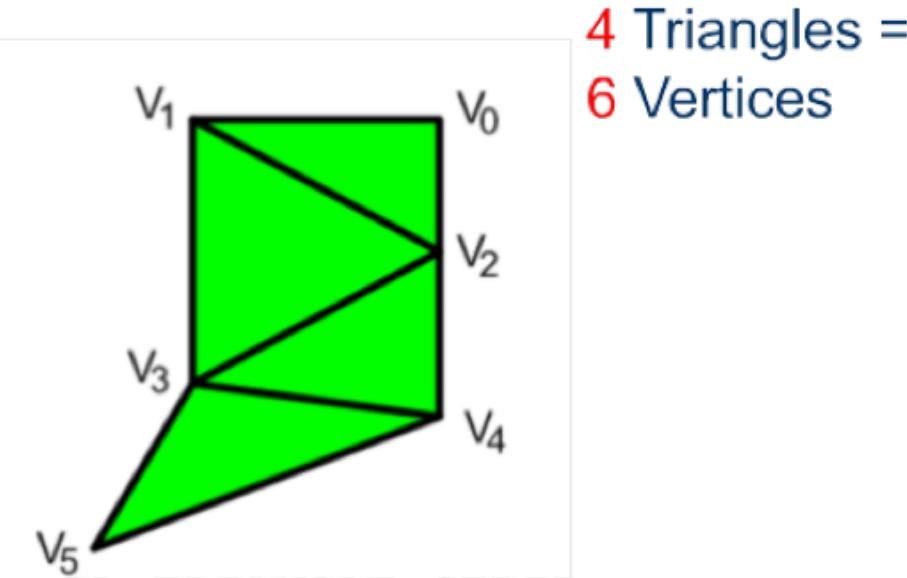
# Triangle Strips

*Triangle strips* encode a set of adjacent triangles that define a band-like surface.

The encoding begins by considering the first two vertices.

Then each new vertex is connected to the previous two.

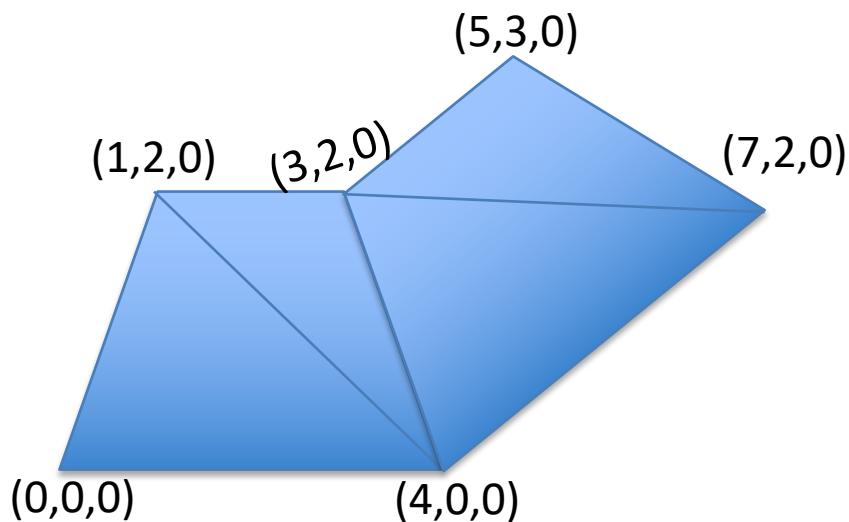
$N+2$  vertices are required to encode  $N$  triangles.



# Encoding example

## Example:

The following part of a geometrical primitive can be encoded with either a triangle strip or with a triangle list. Let us use both encodings, and let us compute the memory requirements supposing that each vertex occupies 12 Bytes.



Strip:  $\{(0,0,0), (1,2,0), (4,0,0), (3,2,0), (7,2,0), (5,3,0)\}$

Space required:  $12 * 6 = 72$  Bytes

List:  $\{(0,0,0), (1,2,0), (4,0,0), (1,2,0), (4,0,0), (3,2,0), (4,0,0), (3,2,0), (7,3,0), (3,2,0), (7,3,0), (5,3,0)\}$

Space required:  $12 * 12 = 144$  Bytes

# Encoding properties

There are circumstances where triangle strips cannot be used even if the topology would seem to be appropriate for the considered kind of encoding.

As we will see, a vertex is usually defined by more parameters than its local coordinates (i.e. normal vector and texture mapping).

Encoding can be used only if the shared vertices are identical with respect to all their parameters.

# Indexed Primitives

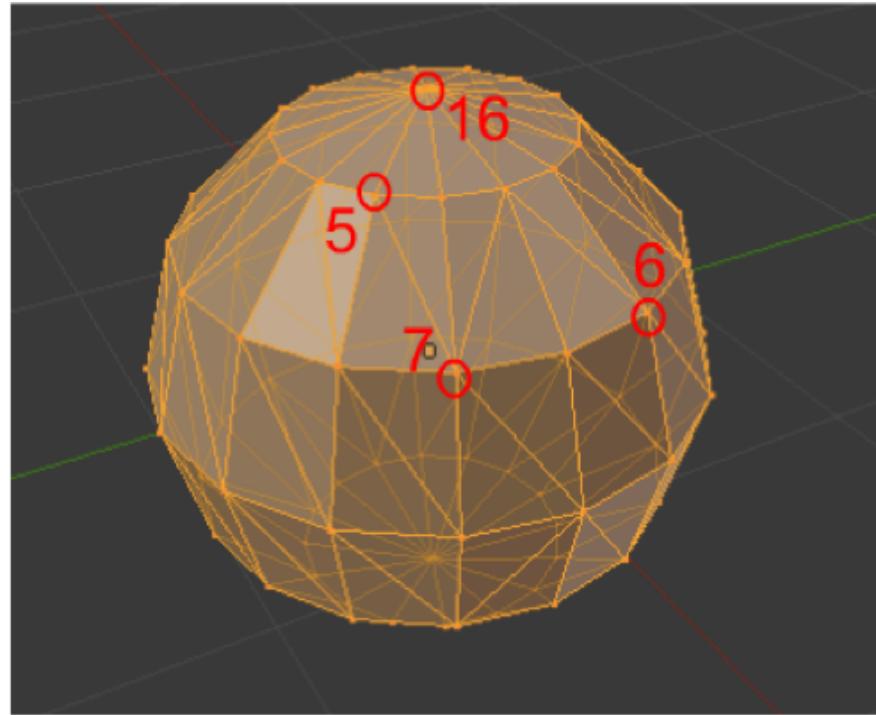
Although triangle strips can save up to two third with respect to triangle lists, there are many cases in which a lot of vertices are still repeated.

Since many primitives cannot be encoded with a single triangle strip, many vertices can still be shared between different strips.

*Indexed primitives* allow reducing the cost of replicating the same vertex between different lists or strips.

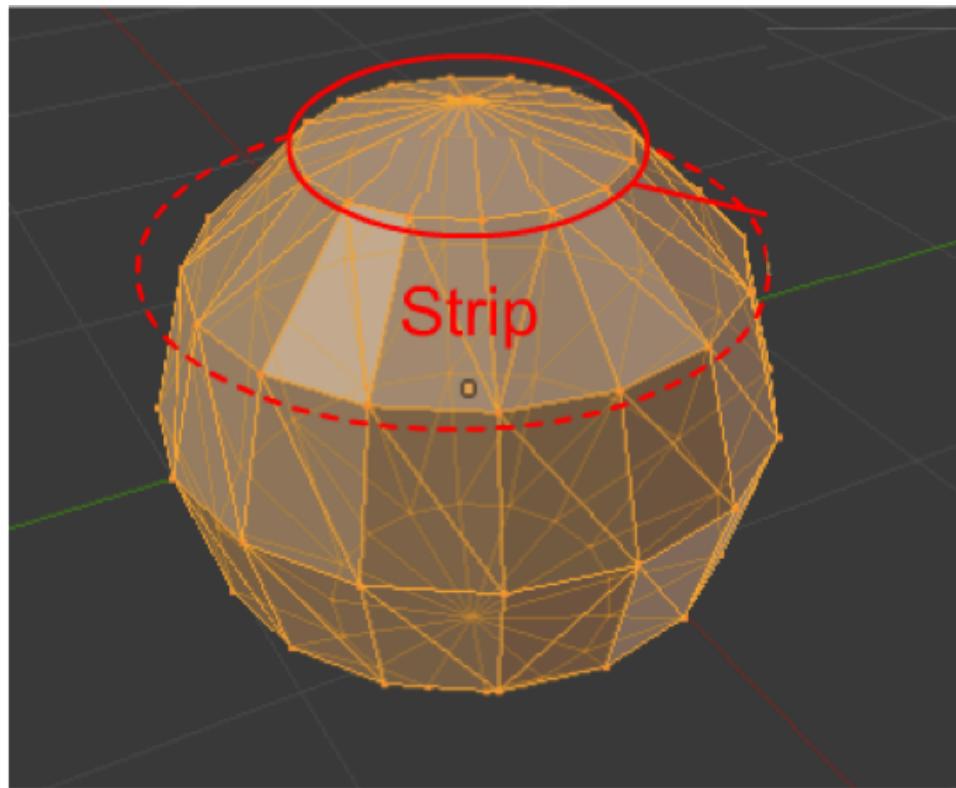
# Indexed Primitives

Let us consider for example a sphere: it is usually composed of several strips, where each vertex is shared by at least 4 triangles.



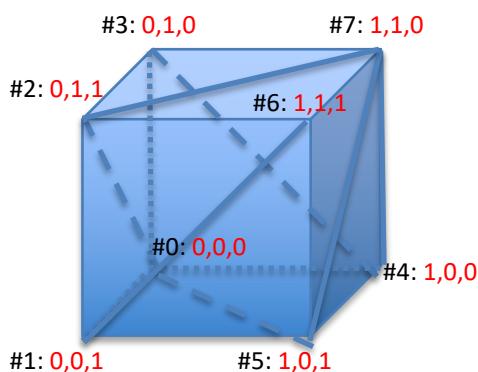
# Indexed Primitives

Triangle strips can reduce the space required by each band: however the same vertex is still repeated in different bands.



# Indexed Primitives

Indexed primitives are defined by two arrays: the *vertex array* contains the definitions (the positions) of the different vertices. Triangles are then specified in an *indirect way* using the *index array*.



```
{0,0,0,  0,0,1, // Vertex array:  
 0,1,1,  0,1,0, // x y z for  
 1,0,0,  1,0,1, // 8 different vertices  
 1,1,1,  1,1,0} // of a cube  
  
{0,1,2,  2,3,0, // Index Array:  
 0,3,4,  4,3,7, // 36 indices  
 5,6,7,  7,4,5, // to the vertices of  
 1,6,5,  1,2,6, // 12 triangles composing  
 7,6,2,  3,2,7, // 6 faces of a cube  
 4,5,0,  1,0,5}
```

# Indexed Primitives

Triangles are defined starting from the indices.

The coordinates of the vertices of the primitive are picked from the vertex list following the position corresponding to the value of the index.

```
{0,0,0, 0,0,1, // Vertex array:
 0,1,1, 0,1,0, // x y z for
 1,0,0, 1,0,1, // 8 different vertices
 1,1,1, 1,1,0} // of a cube
```

```
{0,1,2, 2,3,0, // Index Array:
 0,3,4, 4,5,0, // 36 indices
 0,5,6, 6,1,0, // to the vertices of
 1,6,7, 7,2,1, // 12 triangles composing
 7,4,3, 3,2,7, // 6 faces of a cube
 4,7,6, 6,5,4}
```

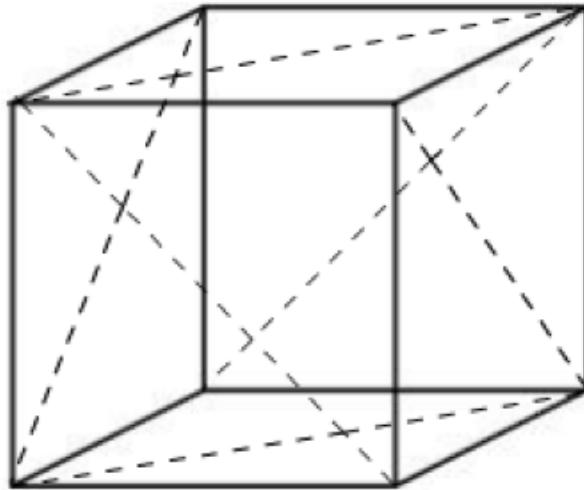
$\underbrace{0 \ 0 \ 0}_0$	$\underbrace{0 \ 0 \ 1}_1$	$\underbrace{0 \ 1 \ 1}_2$	$\underbrace{0 \ 1 \ 1}_2$	$\underbrace{0 \ 1 \ 0}_3$	$\underbrace{0 \ 0 \ 0}_0$
$\underbrace{0 \ 0 \ 0}_0$	$\underbrace{0 \ 1 \ 0}_3$	$\underbrace{1 \ 0 \ 0}_4$	$\underbrace{1 \ 0 \ 0}_4$	$\underbrace{1 \ 0 \ 1}_5$	$\underbrace{0 \ 0 \ 0}_0$
$\underbrace{0 \ 0 \ 0}_0$	$\underbrace{1 \ 0 \ 1}_5$	$\underbrace{1 \ 1 \ 1}_6$	$\underbrace{1 \ 1 \ 1}_6$	$\underbrace{0 \ 0 \ 1}_1$	$\underbrace{0 \ 0 \ 0}_0$
$\underbrace{0 \ 0 \ 1}_1$	$\underbrace{1 \ 1 \ 1}_6$	$\underbrace{1 \ 1 \ 0}_7$	$\underbrace{1 \ 1 \ 0}_7$	$\underbrace{0 \ 1 \ 1}_2$	$\underbrace{0 \ 0 \ 1}_1$
$\underbrace{1 \ 1 \ 0}_7$	$\underbrace{1 \ 0 \ 0}_4$	$\underbrace{0 \ 1 \ 0}_3$	$\underbrace{0 \ 1 \ 0}_3$	$\underbrace{0 \ 1 \ 1}_2$	$\underbrace{1 \ 1 \ 0}_7$
$\underbrace{1 \ 0 \ 0}_4$	$\underbrace{1 \ 1 \ 0}_7$	$\underbrace{1 \ 0 \ 0}_6$	$\underbrace{1 \ 0 \ 0}_6$	$\underbrace{1 \ 0 \ 1}_5$	$\underbrace{1 \ 0 \ 0}_4$

# Indexed Primitives

For example, let us consider a *cube*.

If encoded as a triangle lists, it would require 6 faces, 2 triangles per face, each of 3 vertices composed by 3 coordinates (floating point number, 4 bytes each).

The total Space required is:  $6_{(\text{faces})} * 2_{(\text{tri})} * 3_{(\text{vertices})} * 3_{(\text{xyz})} * 4_{(\text{float})} = 432 \text{ Bytes}$ .

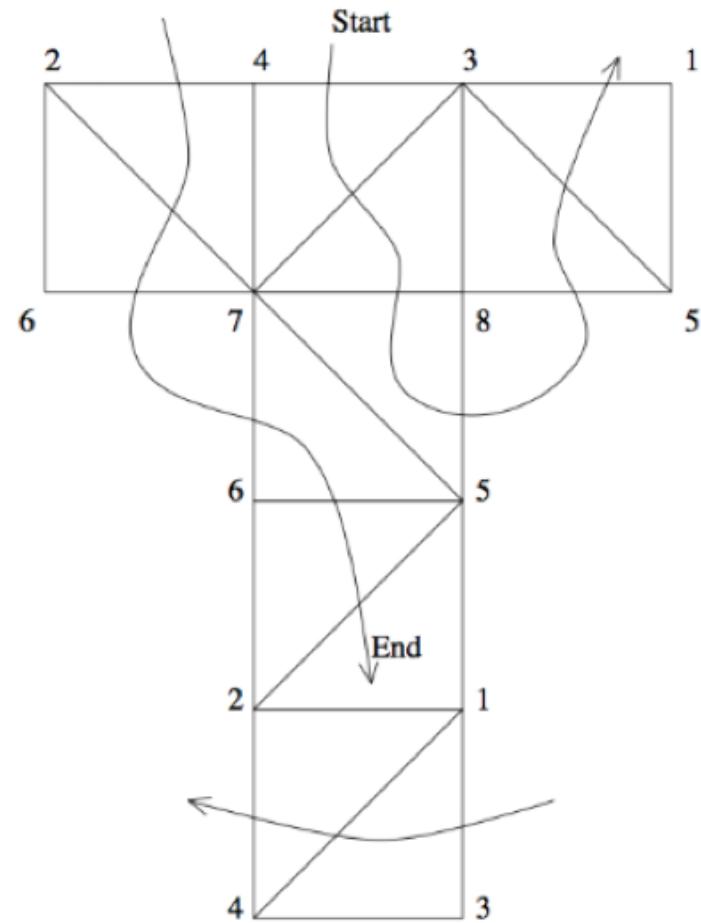


# Indexed Primitives

If the cube is encoded with a triangle strip, only 14 vertices are required.

Space required:  $14_{(\text{vertices})} * 3_{(\text{xyz})} * 4_{(\text{float})} = 168 \text{ Bytes}$

Strip: 4 3 7 8 5 3 1 4 2 7 6 5 2 1

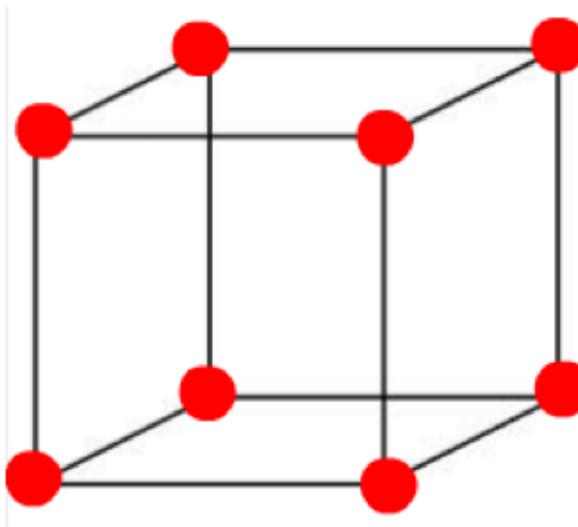


# Indexed Primitives

Since however only 8 different vertices are present, the same primitive encoded using indices would require only one set of coordinates per vertex, and 36 indices (which in Vulkan can be either 2 or 4 Bytes).

Space required:

$$8_{(\text{vertices})} * 3_{(\text{xyz})} * 4_{(\text{float})} + 6_{(\text{faces})} * 2_{(\text{tri})} * 3_{(\text{vertices})} * 2_{(\text{byte})} = 96 + 72 = 168 \text{ Bytes.}$$



# Indexed Primitives

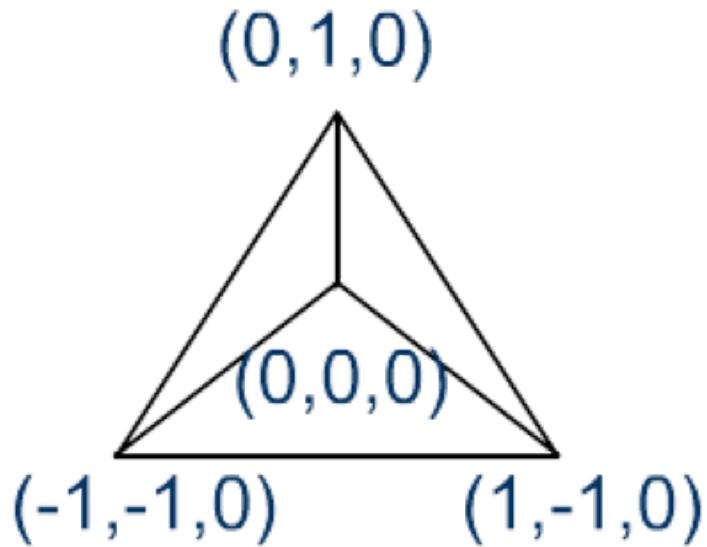
As we have seen, indexing has allowed a great saving in bytes, without the complexity of finding correct ordering (*432 Bytes -> 168 Bytes*).

For this reason many file interchange formats only supports meshes encoded with *indexed triangle lists* to simplify their architecture.

# Indexed Primitives

## Example:

The following primitive can be encoded with an indexed triangles list as follows. The memory requirements are shown, considering each vertex is 12 Bytes, and each index is 1 Byte.



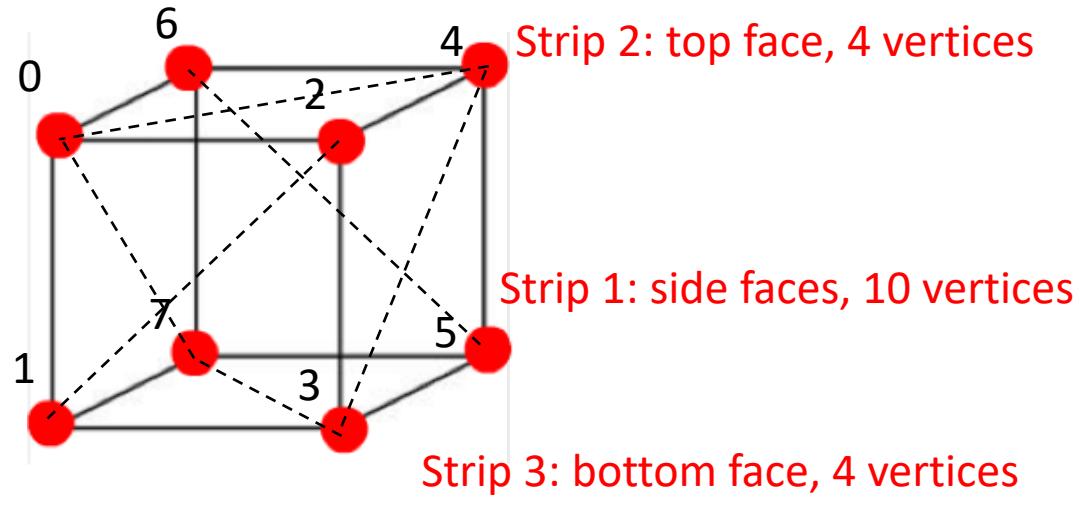
Vertices: { $0,0,0, -1,-1,0,$   
 $0,1,0, 1,-1,0 \}$

Indices { $0,1,2, 0,2,3, 0,3,1 \}$

Space required:  $4*12 + 9*1 = 57$  Bytes

# Indexed Primitives

Vulkan allows to further reduce the space in band-like structures using the *restart* feature: a negative index restarts the strip.



Required indices:

$$10_{(\text{side})} + 4_{(\text{top})} + 4_{(\text{bottom})} + 2_{(\text{restart})} = 20$$

```
{0,0,0, 0,0,1, // Vertex array:  
0,1,1, 0,1,0, // x y z for  
1,0,0, 1,0,1, // 8 different vertices  
1,1,1, 1,1,0} // of a cube
```

```
{0,1,2,3,4,5, // Index Array:  
6,7,0,1, -1, // 20 indices  
0,2,4,6, -1, // with restarts  
1,3,7,5}
```

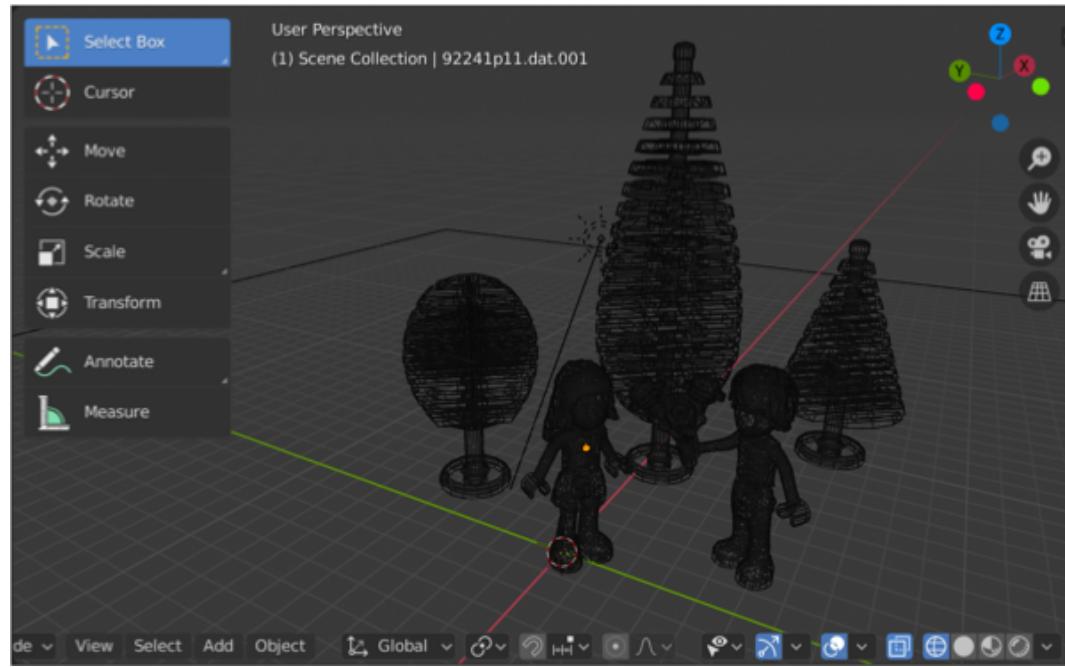
Space required:

$$8(\text{vertices}) * 3(\text{xyz}) * 4(\text{float}) + 20(\text{indices}) * 2(\text{byte}) = 96 + 40 = 136 \text{ Bytes.}$$

# Wireframe

*Wireframe* views, shows only the boundaries of objects, connecting vertices with lines.

They are useful in many cases for debugging a 3D application.



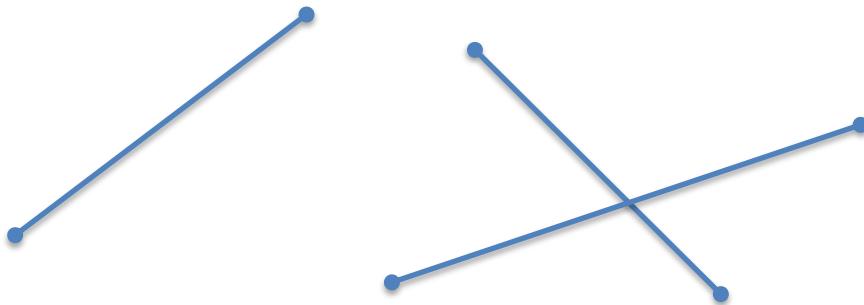
# Wireframes encodings

The two main types of wireframe meshes encoding are:

- *Line Lists*
- *Line Strips*

## Line lists

*Line lists* encode each segment as a couple of two vertices.  
To encode  $N$  segments,  $2N$  vertices are required.

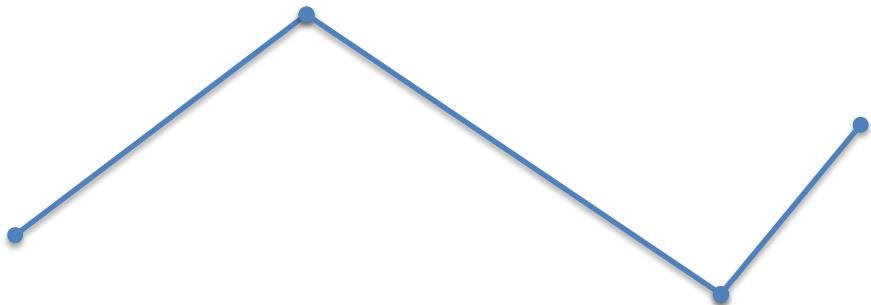


3 Lines =  
6 Vertices

# Line strips

*Line strips encode a path of connected vertices.*

To encode  $N$  segments,  $N+1$  vertices are required.

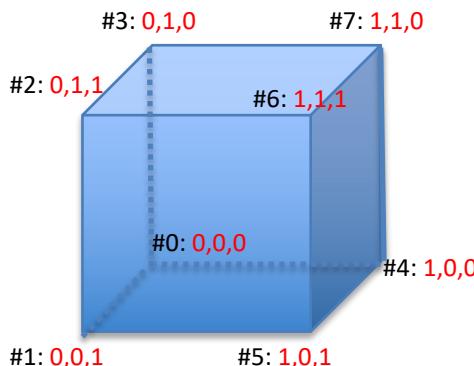


3 Lines =  
4 Vertices

# Wireframe Indexed Primitives

Wireframe primitives can be indexed as well.

```
{0,0,0,  0,0,1, // Vertex array:  
 0,1,1,  0,1,0, // x y z for  
 1,0,0,  1,0,1, // 8 different vertices  
 1,1,1,  1,1,0} // of a cube
```

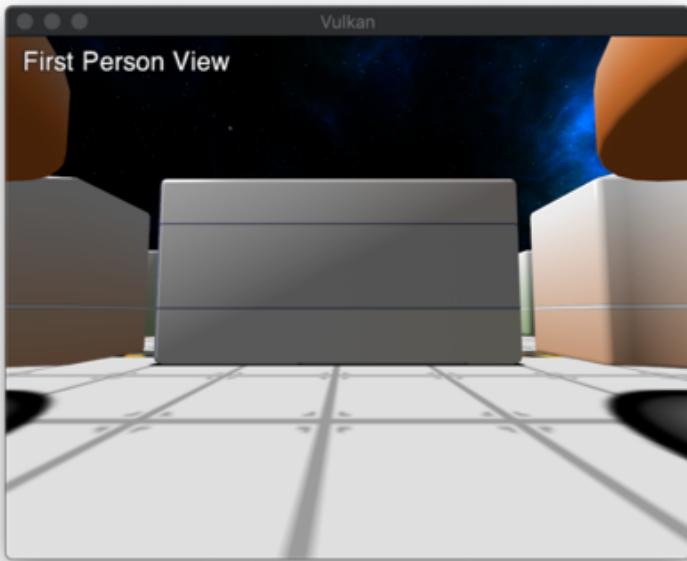


```
{0,1,  1,2,  2,3,  3,0,    // Index Array:  
 4,5,  5,6,  6,7,  7,4,    // 24 indices  
 1,5,  2,6,  3,7,  0,4}    // to the vertices of  
                           // 12 lines
```

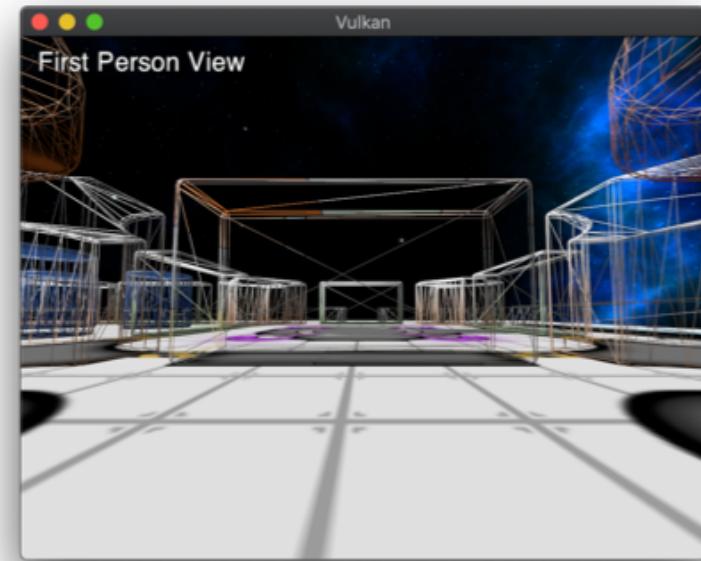
*Line list:*  $12 \times 2 \times 3 \times 4 = 288$  bytes  
*Indexed line list:*  $8 \times 3 \times 4 + 12 \times 2 = 120$  bytes

# Drawing outlines

In addition, Vulkan also allows to draw standard objects using only the contour of their triangles: this is the technique used in the assignments to create the wireframe images against which the filled one should be matched.



Filled



Contours