POLITECNICO
MILANO 1863

DIPARTIMENTO DI ELETTRONICA
INFORMAZIONE E BIOINGEGNERIA

DE
IB

2024

Dipartimento di Elettronica, Informazione e Bioingegneria

Computer Graphics

Milano, 2024

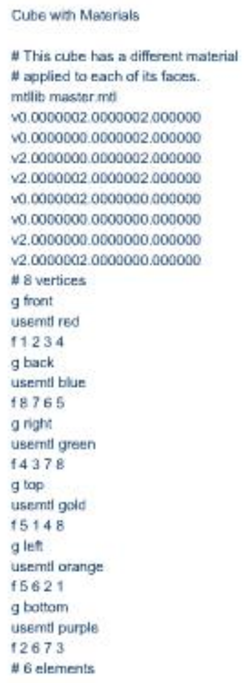
Computer Graphics

- 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.



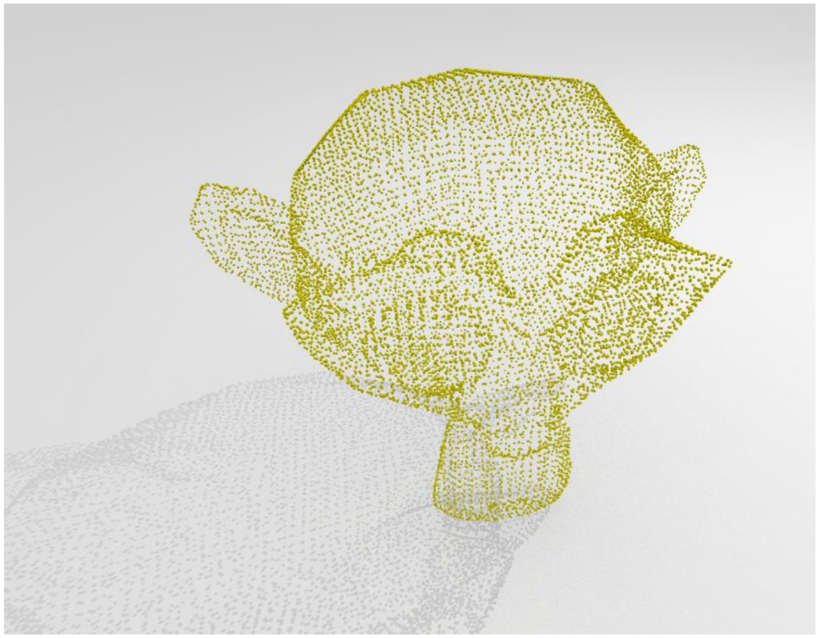
Cube with Materials

```
# This cube has a different material
# applied to each of its faces.
mtllib master.mtl
v0.0000002.0000002.000000
v0.0000000.0000002.000000
v2.0000000.0000002.000000
v2.0000002.0000002.000000
v0.0000002.0000000.000000
v0.0000000.0000000.000000
v2.0000000.0000000.000000
v2.0000002.0000000.000000
# 8 vertices
g front
usemtl red
f 1 2 3 4
g back
usemtl blue
f 8 7 6 5
g right
usemtl green
f 4 3 7 8
g top
usemtl gold
f 5 1 4 8
g left
usemtl orange
f 5 6 2 1
g bottom
usemtl purple
f 2 6 7 3
# 6 elements
```

Meshes

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

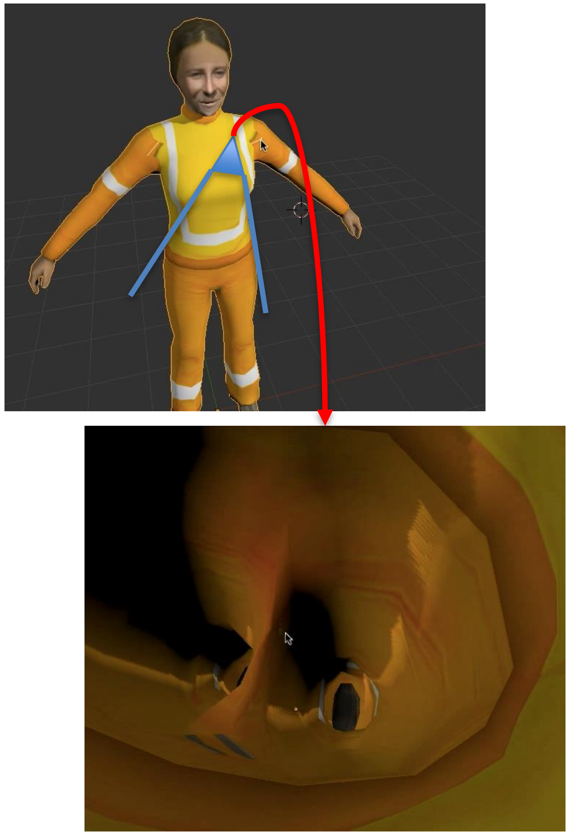
However usually objects are not encoded as a set of unconnected points (*point clouds*), because they would require too many resources to obtain an apparently solid object.



With **9646** vertices: the object still looks very "empty"

Meshes

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

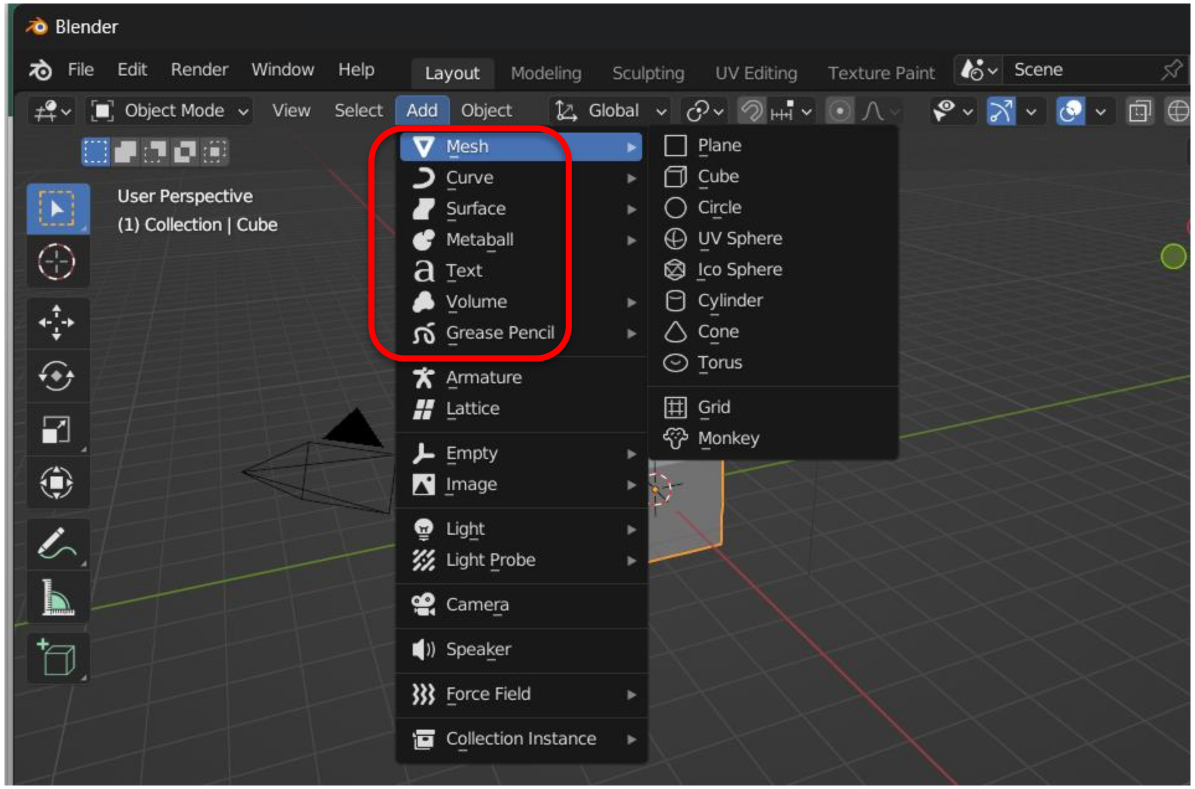
Many approaches have been defined in the literature. The most common are:

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

Meshes

3D authoring tools like *Blender*, usually allows the user to select between many of these techniques, to encode the models created by the artist.

Each technique, requires a specific set of tools, and modelling skills.



Meshes

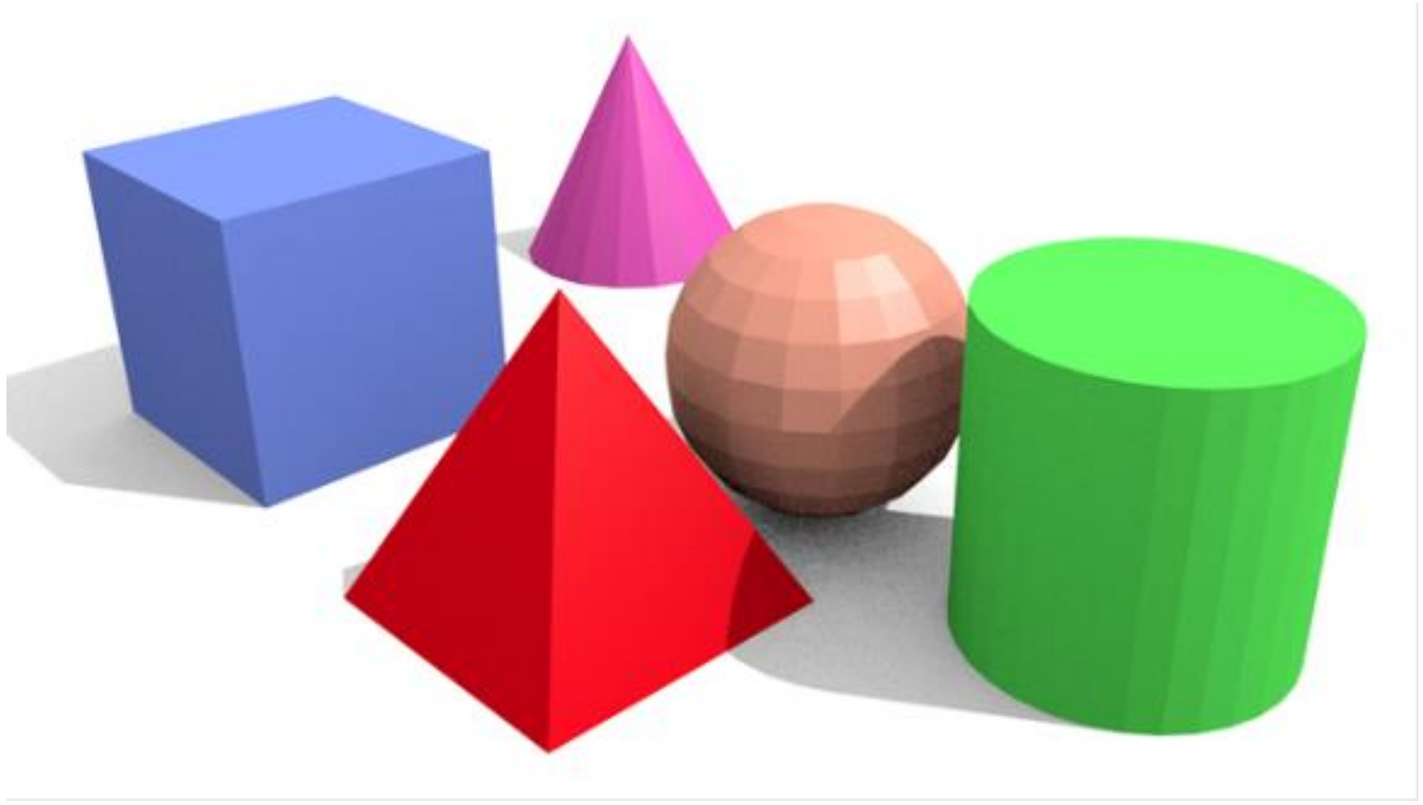
Each mathematical model has its own features and limitations.

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

For this reason we will focus on meshes only.

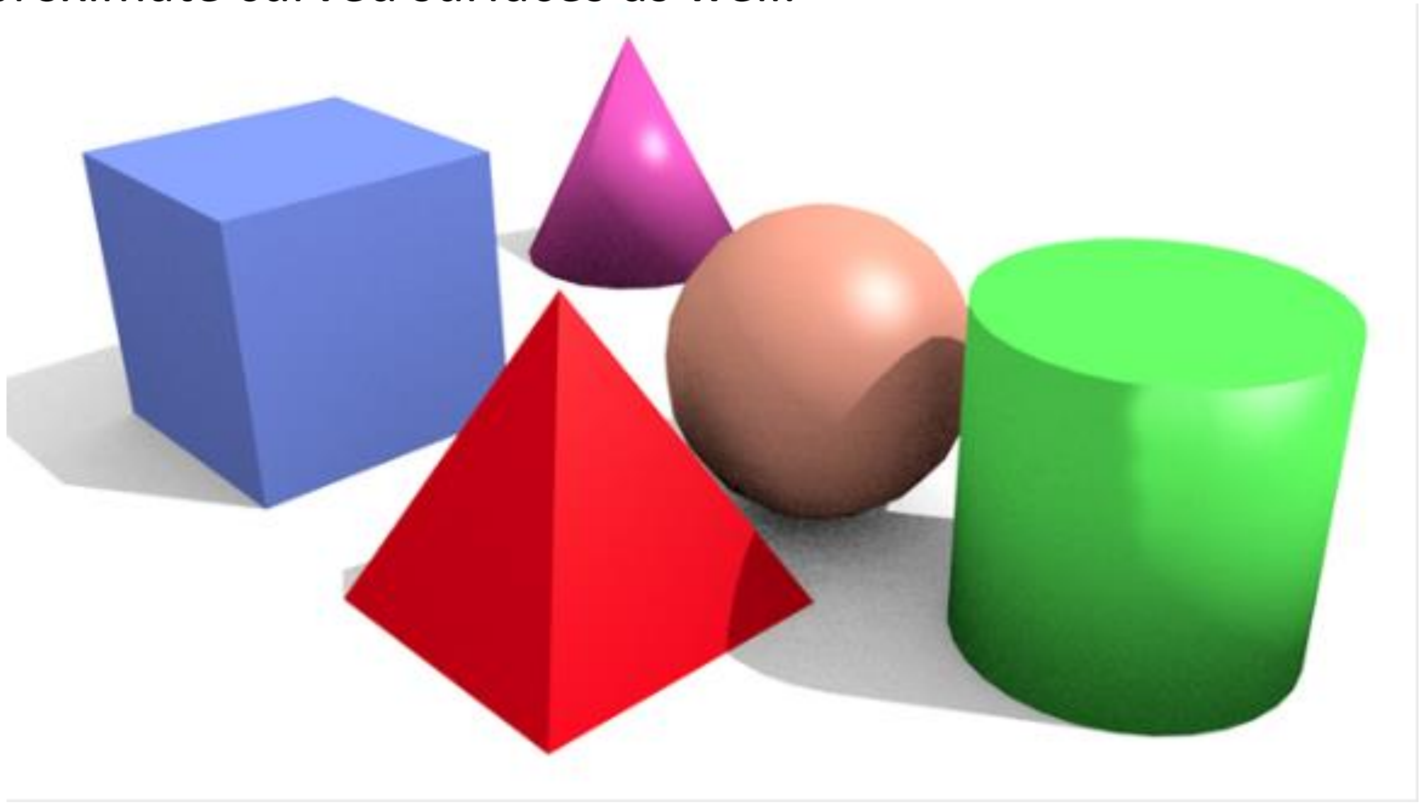
Meshes

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



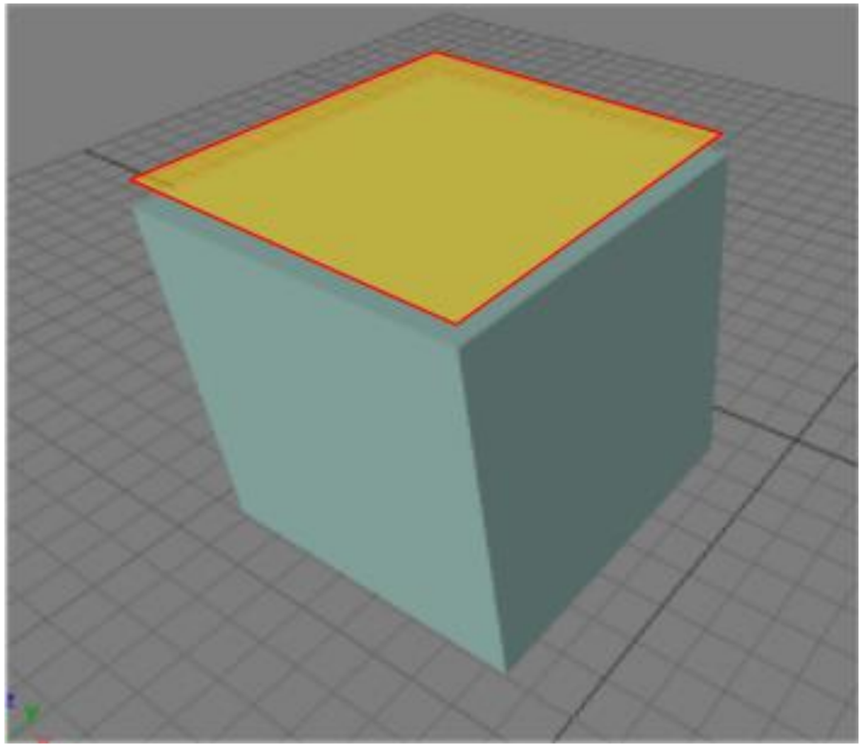
Meshes

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.



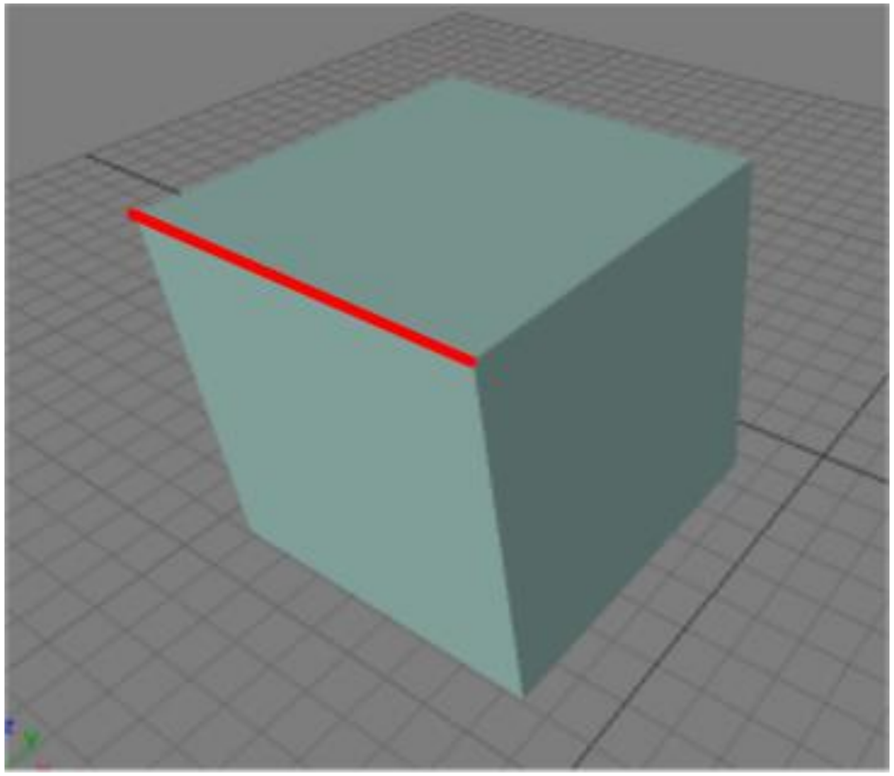
Meshes

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



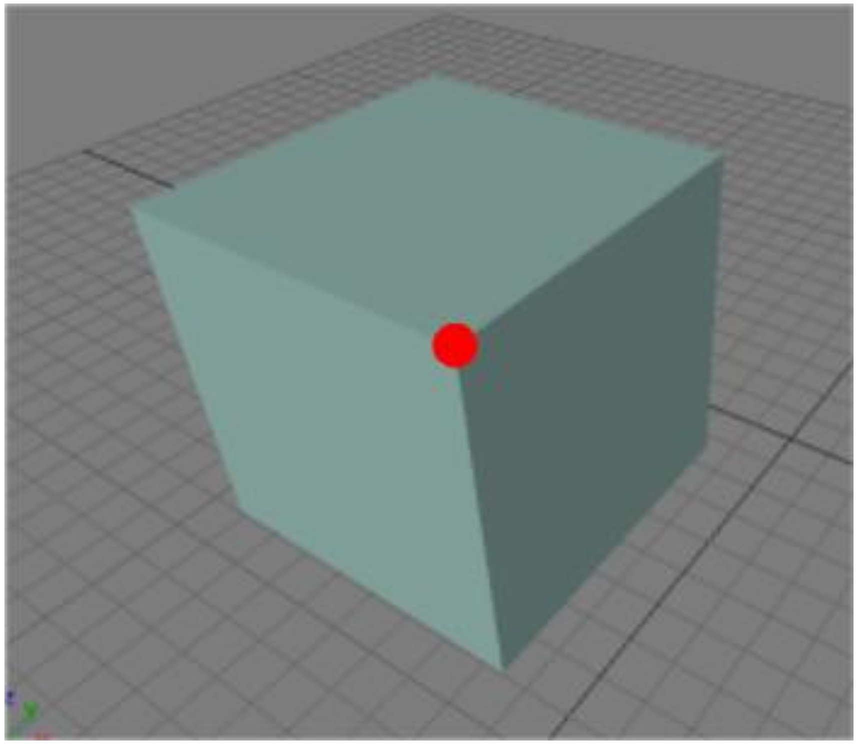
Meshes

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 at least three faces intersect at a vertex in a proper solid.

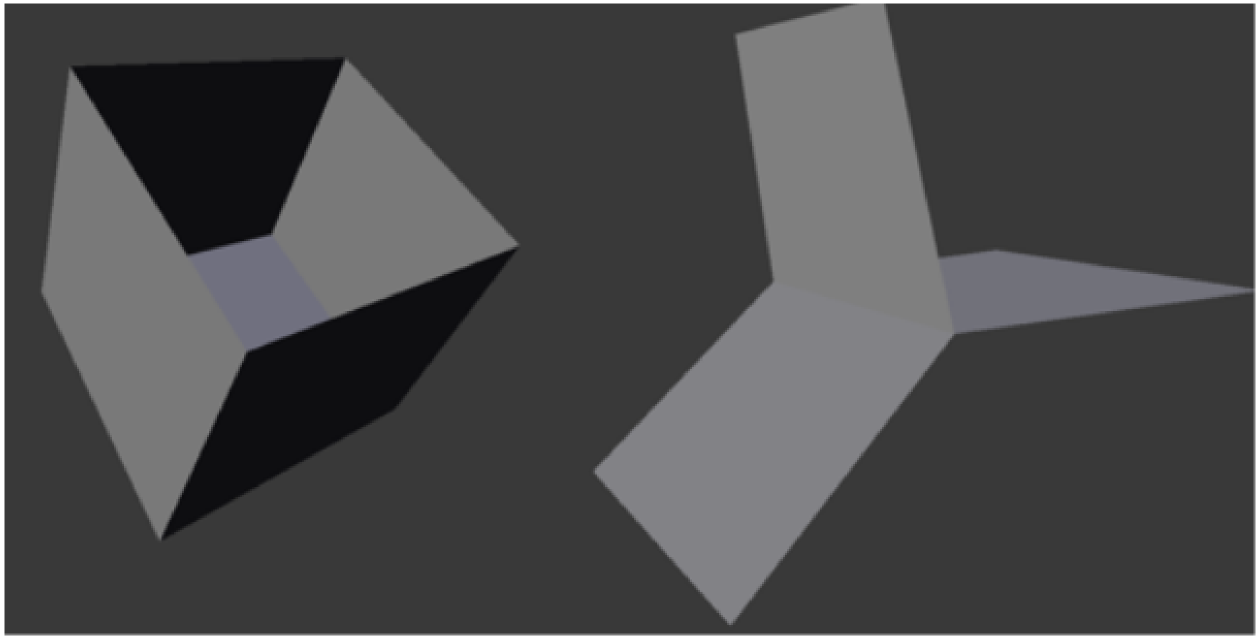


Meshes

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

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

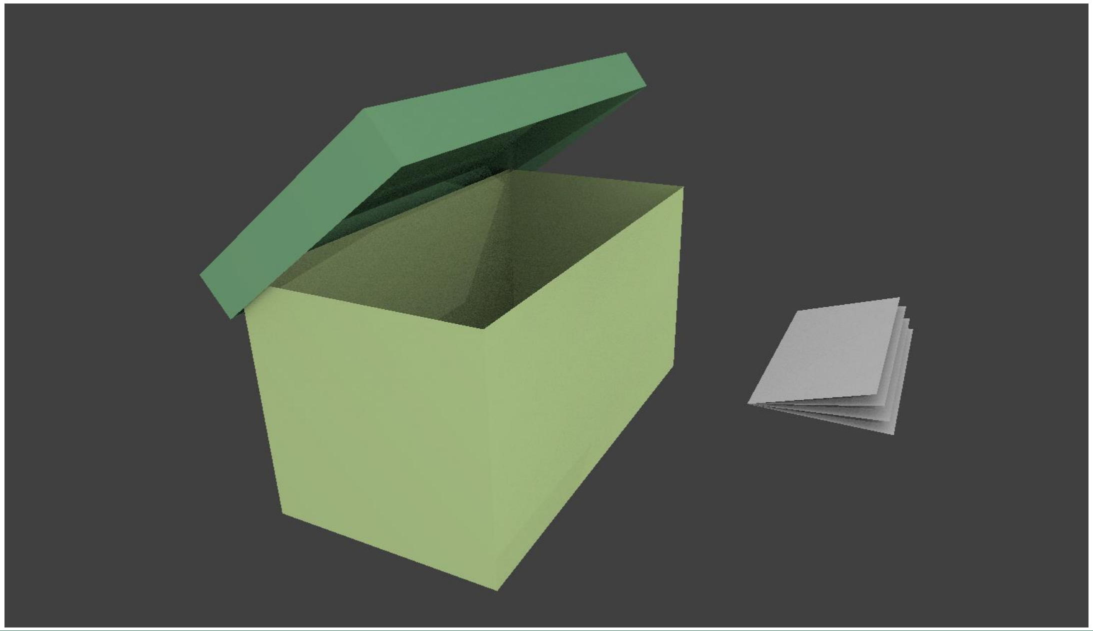
Examples of non 2-manifolds are solids with *holes* and *lamina-faces.*



Meshes

Sometimes non-2-manyfold 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*).

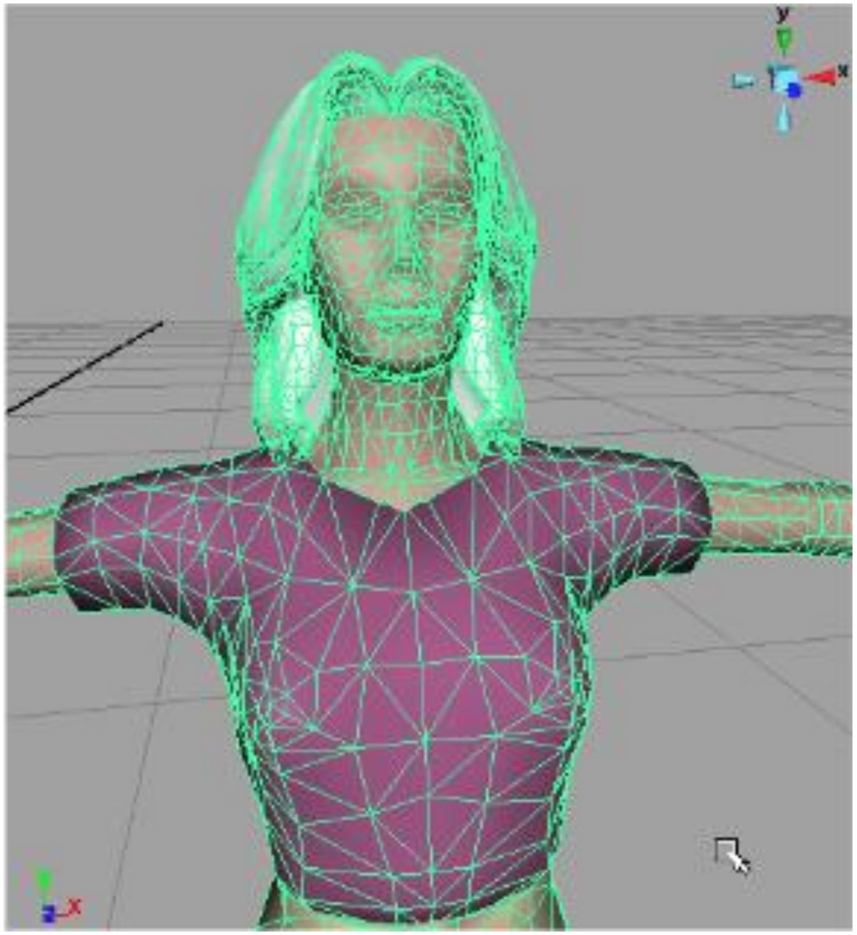
Algorithms like the *backface-culling* (which will be presented later), however, will not work on non-2-manifold objects.



Meshes

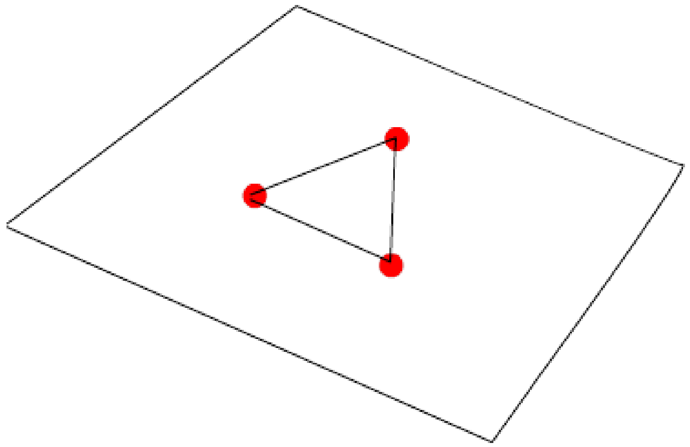
Each polygon can be reduced to a set of triangles that shares some edges.

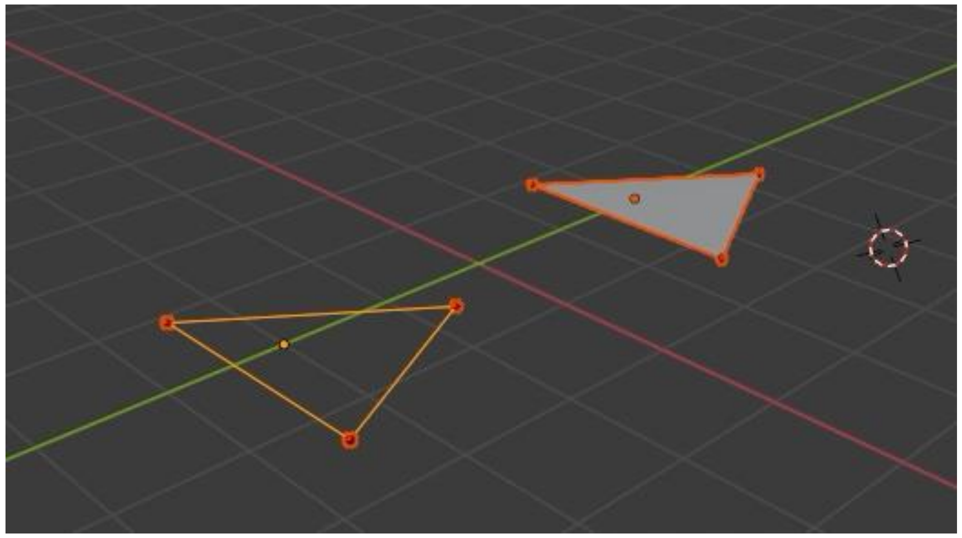
A set of adjacent triangles is called a *Mesh*.



Meshes

We know that three unaligned points identify a unique plane in the space, and three unaligned points define a triangle.

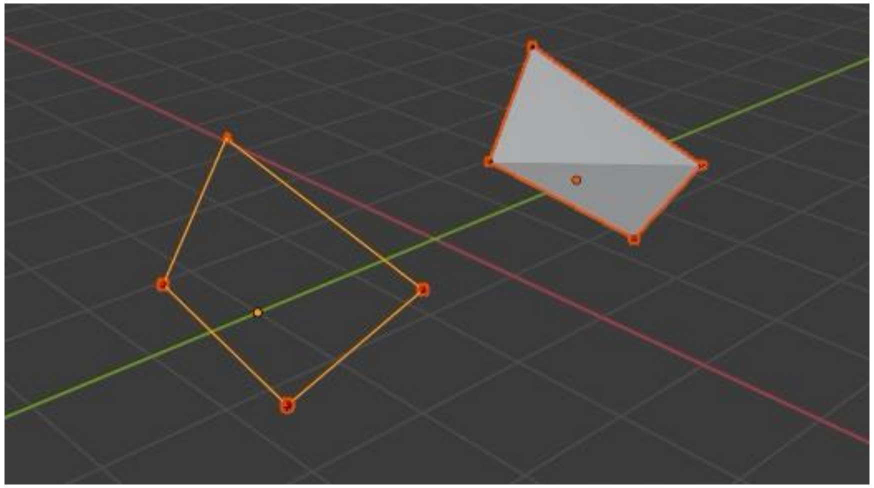


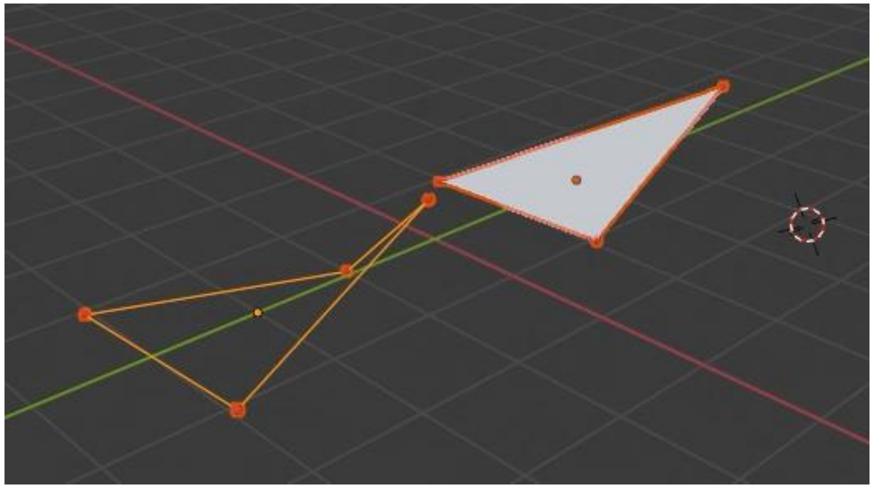


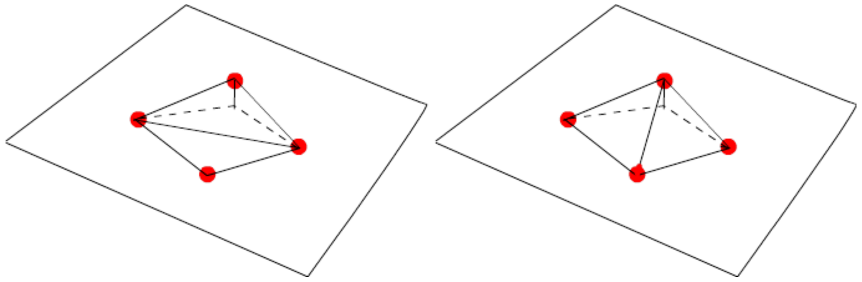
Meshes

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.

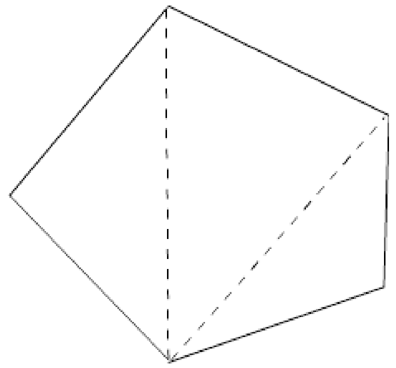




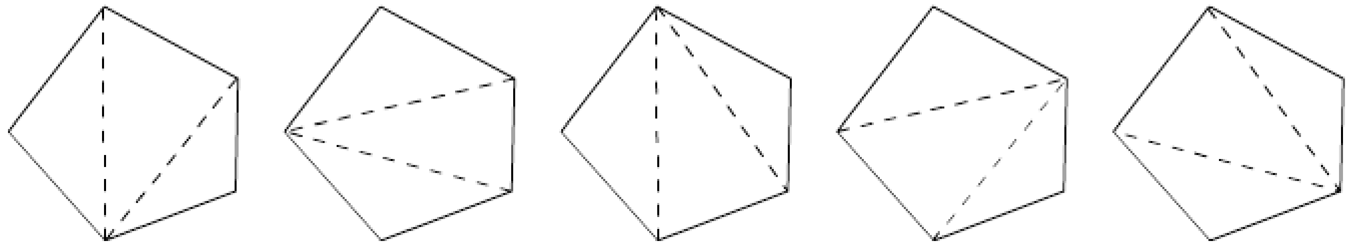


Meshes

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



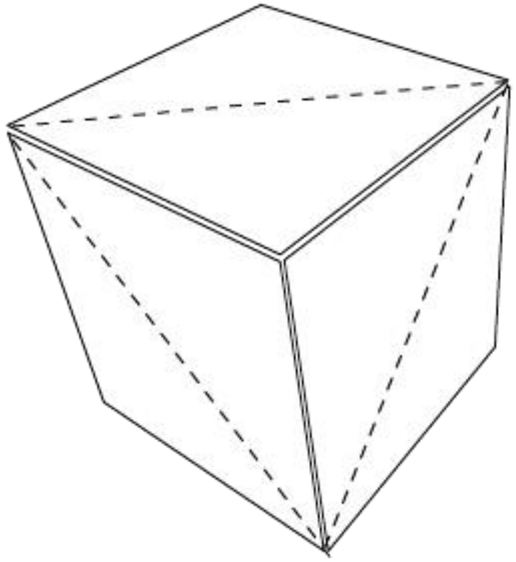
Polygon tessellations are not unique. Several different tessellation might be defined for each polygon with more than 3 sides.

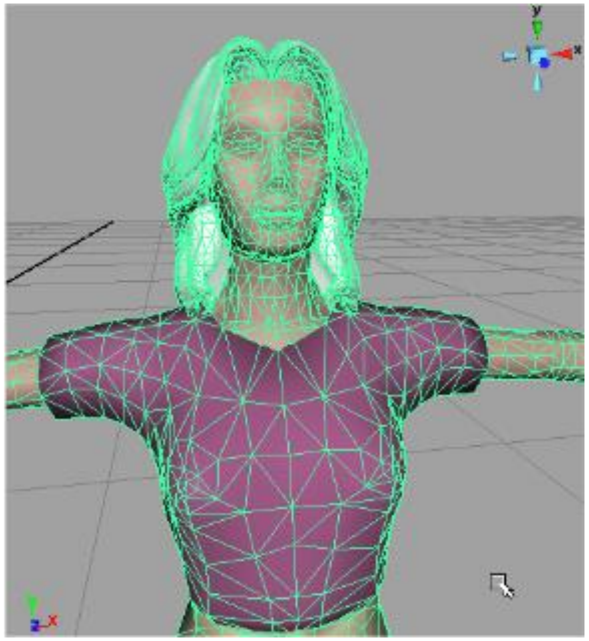


Meshes

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.



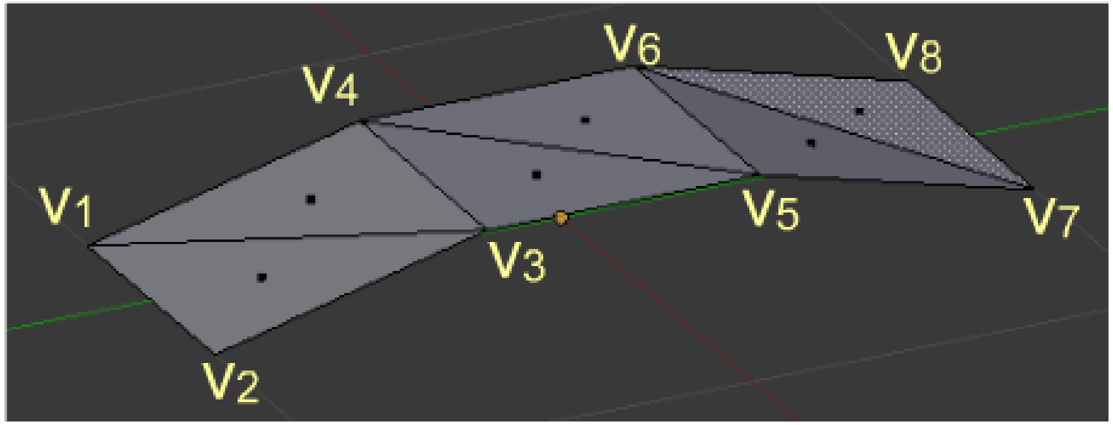


Meshes encodings

Meshes encoding are based on sets of vertices.

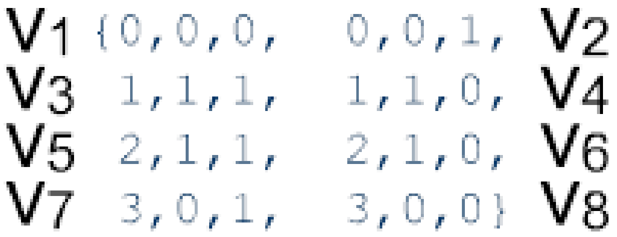
```
V1 {0,0,0,   0,0,1, V2
V3  1,1,1,   1,1,0, V4
V5  2,1,1,   2,1,0, V6
V7  3,0,1,   3,0,0} V8
```

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



Meshes encodings

Please note that vertex coordinates are usually stored as *cartesian coordinates* to save memory.



```
V1 {0,0,0,   0,0,1, V2
V3  1,1,1,   1,1,0, V4
V5  2,1,1,   2,1,0, V6
V7  3,0,1,   3,0,0} V8
```

The rendering engine, before transforming the vertices with the World-View-Projection matrix, adds the fourth component equal to one to make the coordinates homogeneous.

V_1 {0,0,0,1,   0,0,1,1, V_2

V_3 {1,1,1,1,   1,1,0,1, V_4

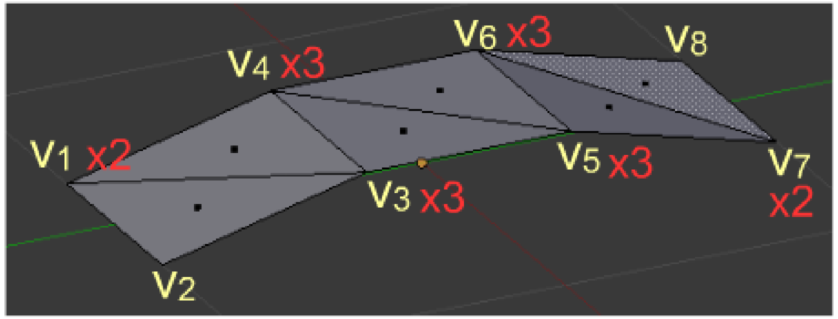
V_5 {2,1,1,1,   2,1,0,1, V_6

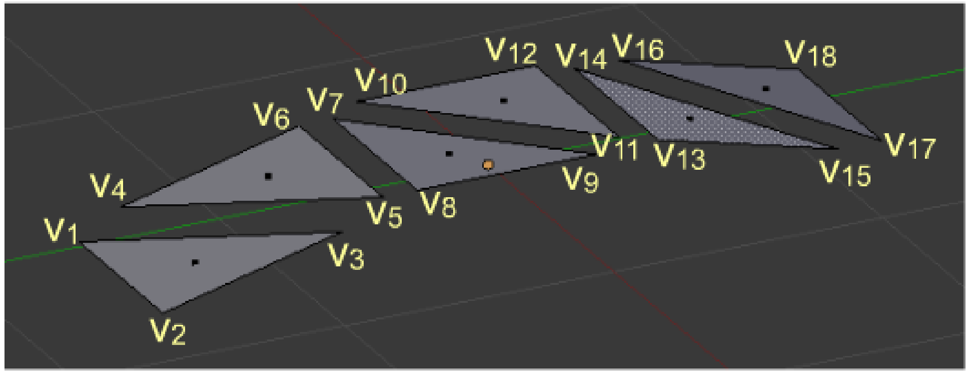
V_7 {3,0,1,1,   3,0,0,1} V_8

Mesh encodings

It can be easily seen that most of the triangles of 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.





10 extra vertices are required if no encoding is used.

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*

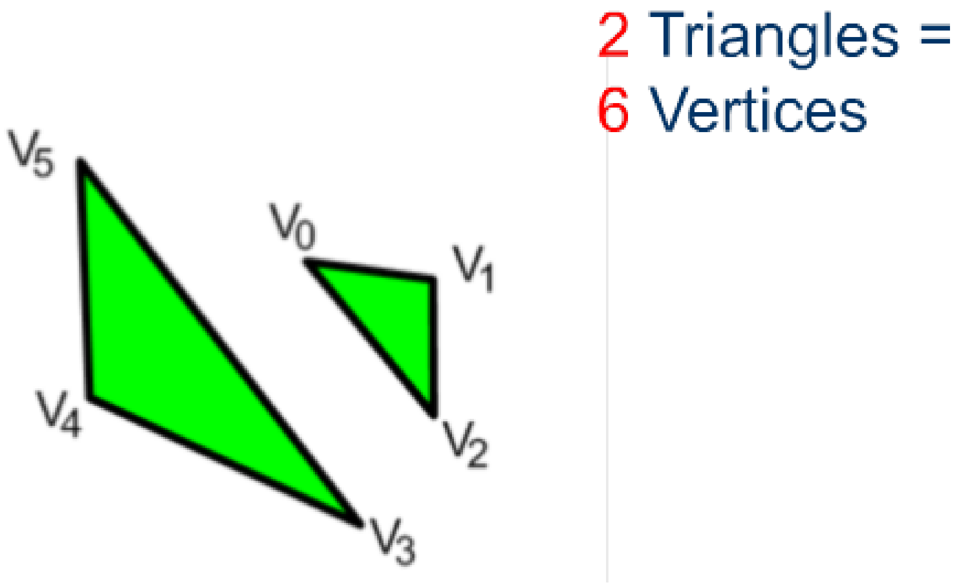
A third encoding, *Triangle Fans*, which was standard in OpenGL, has been made optional in Vulkan, so some graphic adapters might no longer support it: for this reason we will not cover it in this course.

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.



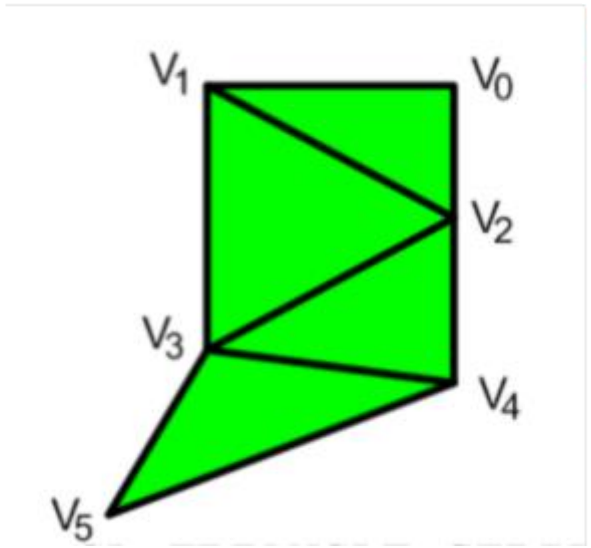
Triangle Strips

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.

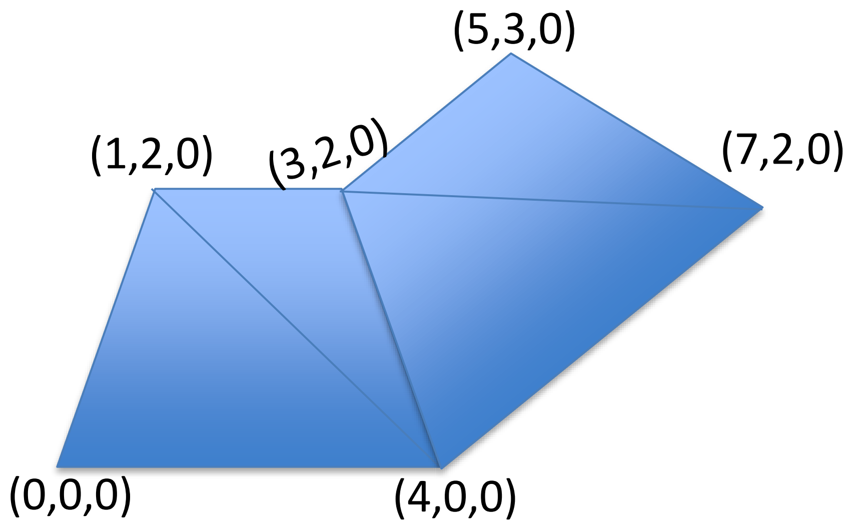


4 Triangles =
6 Vertices

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

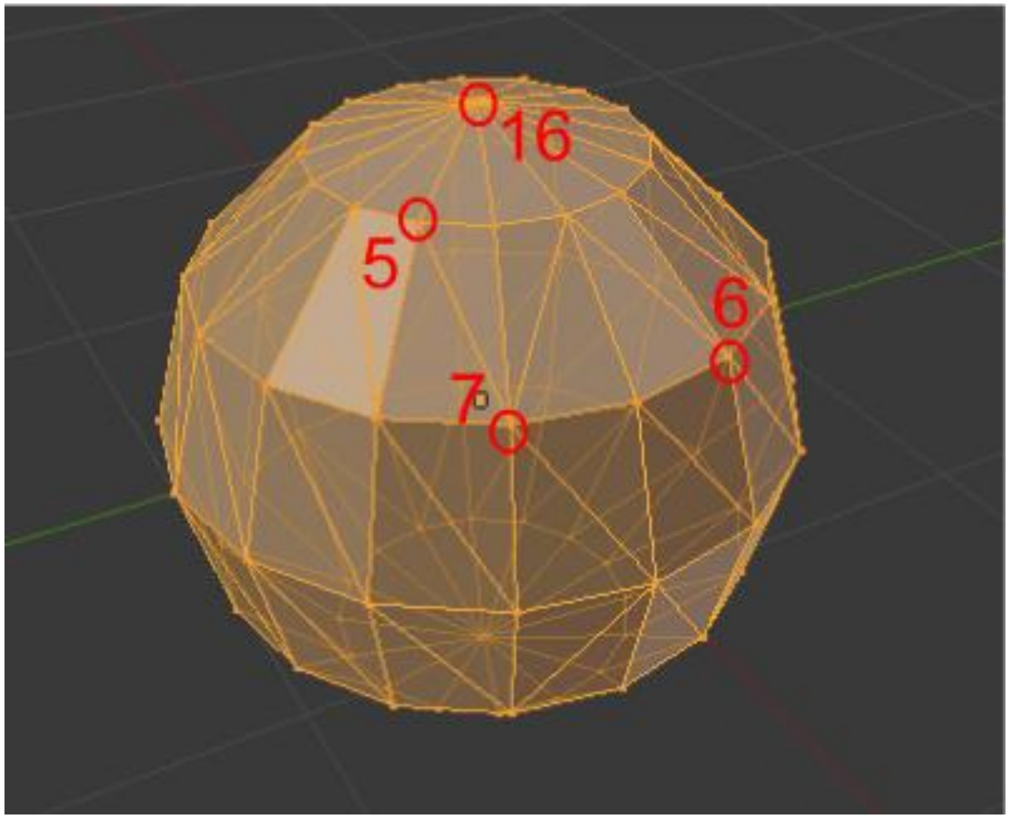
Although triangle strips can save up to two third with respect to triangle lists, there are many cases in which a lot of vertices are still repeated.

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.

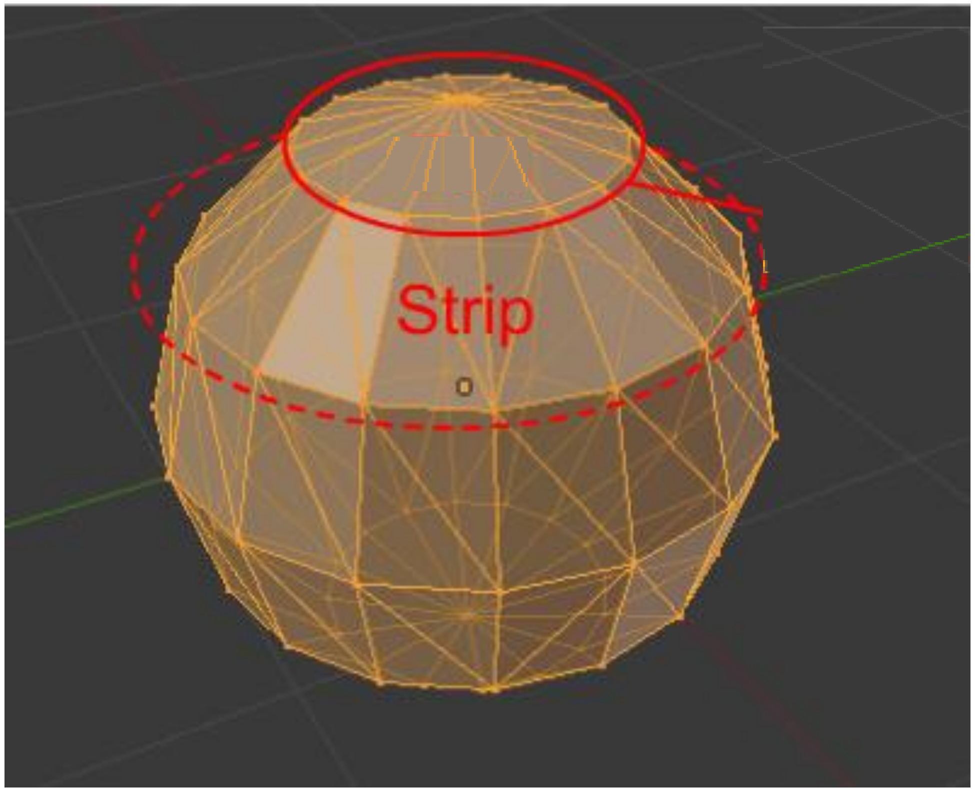
Indexed Primitives

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



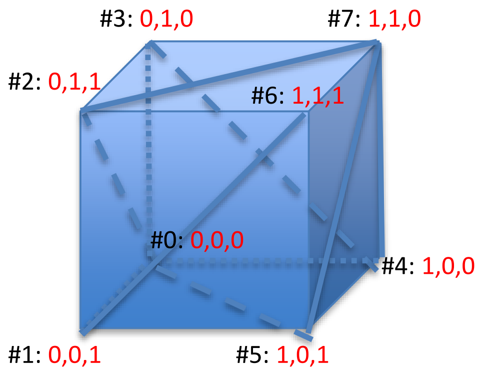
Indexed Primitives

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.

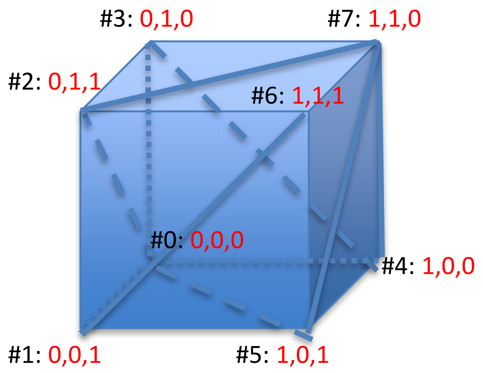


```
{0,0,0,  0,0,1, // Vertex array:
      0,1,1,  0,1,0, // x y z for
      1,0,0,  1,0,1, // 8 different vertices
      1,1,1,  1,1,0} // of a cube
```

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 drawn starting from their indices.

The coordinates of the vertices are picked from the vertex list, according to the position defined by 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}
```

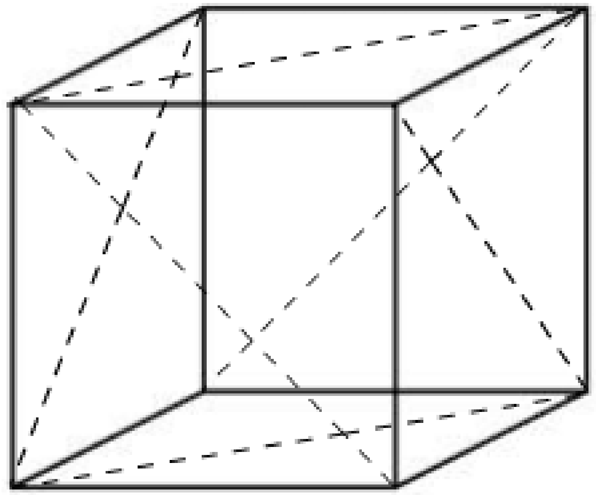
$$\begin{array}{cccccc}
\underbrace{0\ 0\ 0}_{0} & \underbrace{0\ 0\ 1}_{1} & \underbrace{0\ 1\ 1}_{2} & \underbrace{0\ 1\ 1}_{2} & \underbrace{0\ 1\ 0}_{3} & \underbrace{0\ 0\ 0}_{0} \\
\underbrace{0\ 0\ 0}_{0} & \underbrace{0\ 1\ 0}_{3} & \underbrace{1\ 0\ 0}_{4} & \underbrace{1\ 0\ 0}_{4} & \underbrace{1\ 0\ 1}_{5} & \underbrace{0\ 0\ 0}_{0} \\
\underbrace{0\ 0\ 0}_{0} & \underbrace{1\ 0\ 1}_{5} & \underbrace{1\ 1\ 1}_{6} & \underbrace{1\ 1\ 1}_{6} & \underbrace{0\ 0\ 1}_{1} & \underbrace{0\ 0\ 0}_{0} \\
\underbrace{0\ 0\ 1}_{1} & \underbrace{1\ 1\ 1}_{6} & \underbrace{1\ 1\ 0}_{7} & \underbrace{1\ 1\ 0}_{7} & \underbrace{0\ 1\ 1}_{2} & \underbrace{0\ 0\ 1}_{1} \\
\underbrace{1\ 1\ 0}_{7} & \underbrace{1\ 0\ 0}_{4} & \underbrace{0\ 1\ 0}_{3} & \underbrace{0\ 1\ 0}_{3} & \underbrace{0\ 1\ 1}_{2} & \underbrace{1\ 1\ 0}_{7} \\
\underbrace{1\ 0\ 0}_{4} & \underbrace{1\ 1\ 0}_{7} & \underbrace{1\ 0\ 0}_{6} & \underbrace{1\ 0\ 0}_{6} & \underbrace{1\ 0\ 1}_{5} & \underbrace{1\ 0\ 0}_{4}
\end{array}$$

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 defined by 3 coordinates (floating point number, 4 bytes each).

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

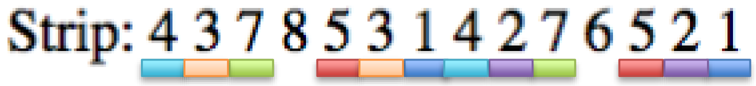


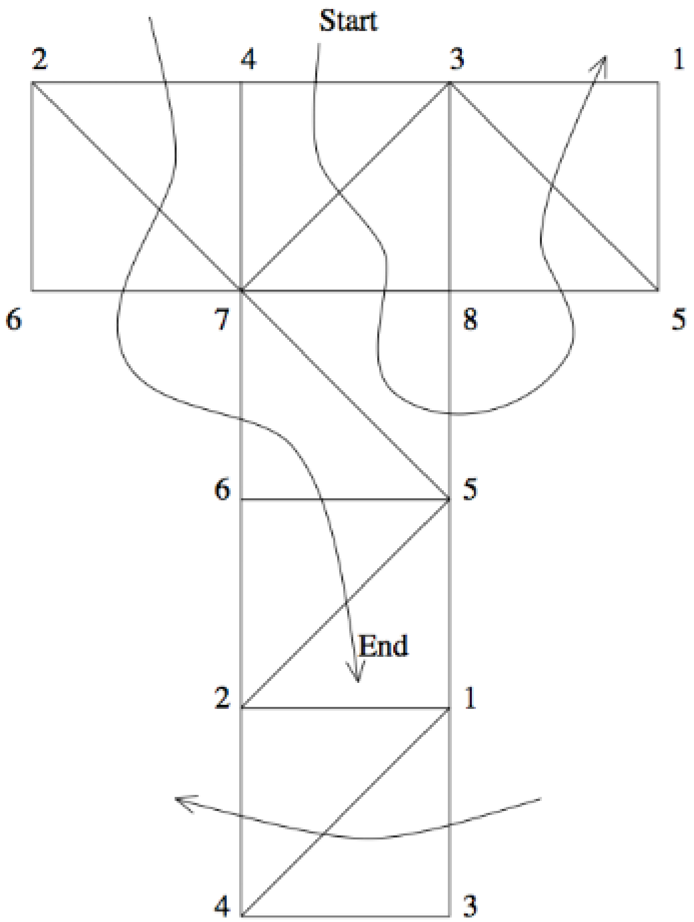
Indexed Primitives

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

It has been proven, that six vertices have to be repeated to complete a strip.

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


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

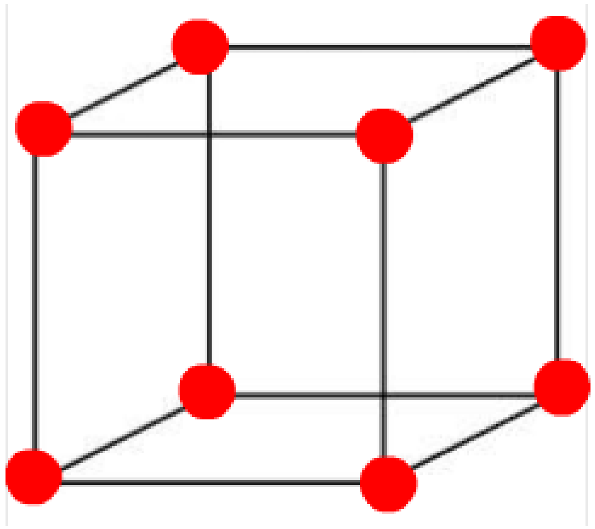


Indexed Primitives

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:

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



Indexed Primitives

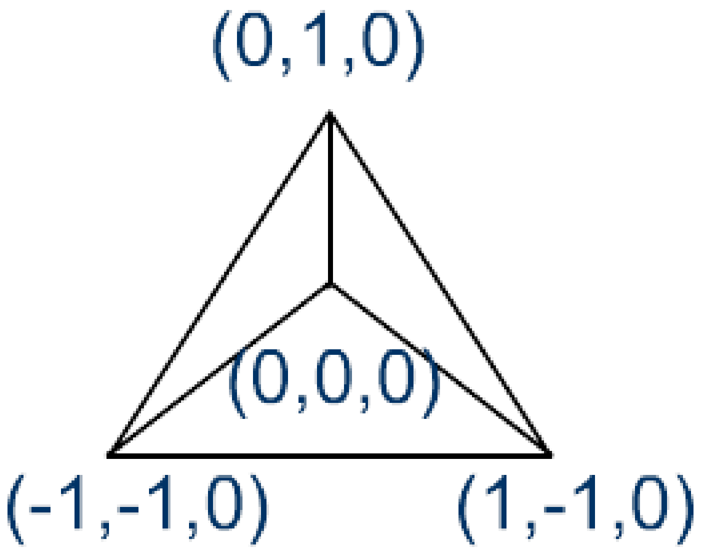
As we have seen, indexing has allowed a great saving in bytes, without the complexity of finding correct ordering *(432 Bytes -> 168 Bytes)*.

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.



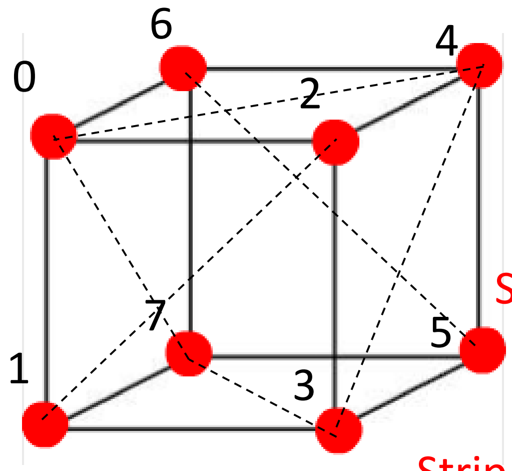
Vertices: {0,0,0, -1,-1,0,
0,1,0, 1,-1,0 }

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.



Strip 2: top face, 4 vertices

Strip 1: side faces, 10 vertices

Strip 3: bottom face, 4 vertices

Required indices:

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

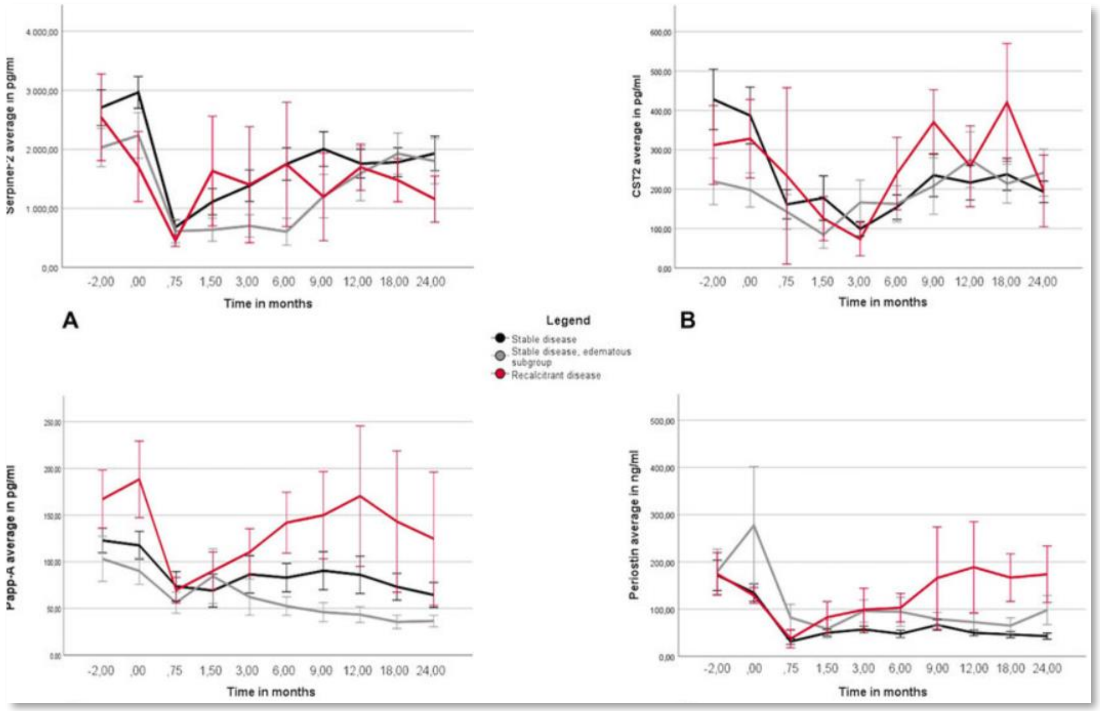
```
{0,0,0,  0,0,1, // Vertex array:
 0,1,1,  0,1,0, // x y z for
 1,0,0,  1,0,1, // 8 different vertices
 1,1,1,  1,1,0} // of a cube

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

2D drawings

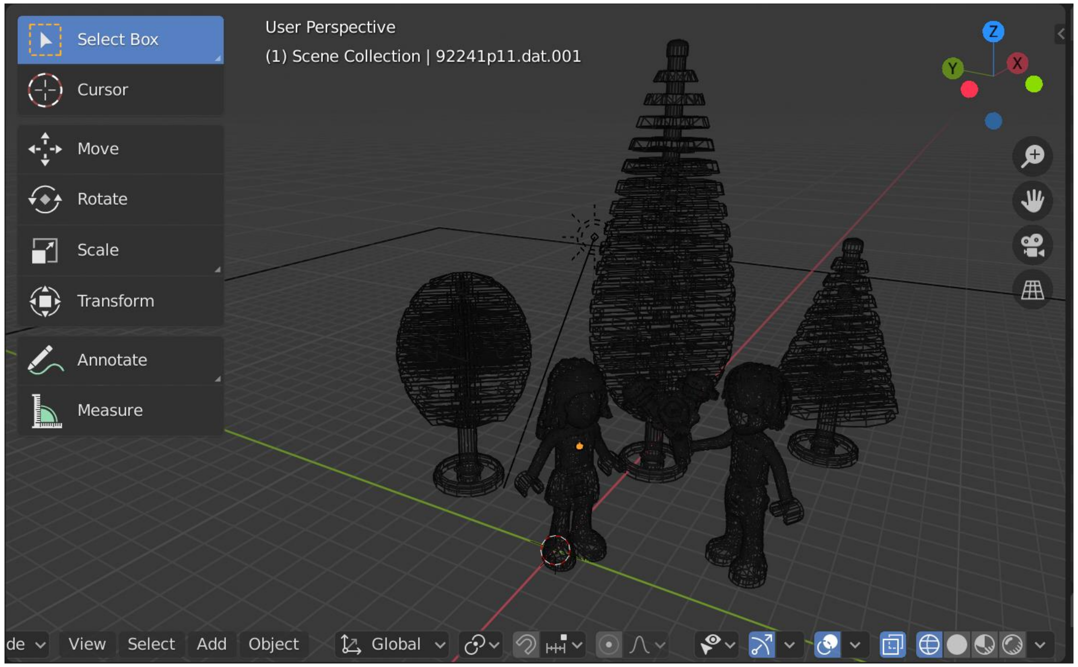
When *2D plots and charts* are needed, they are usually created using only lines instead of triangles.



Wireframe

Lines are also used to produce *Wireframe* views, where only the boundaries of the objects are shown by connecting their vertices with lines.

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



Wireframes encodings

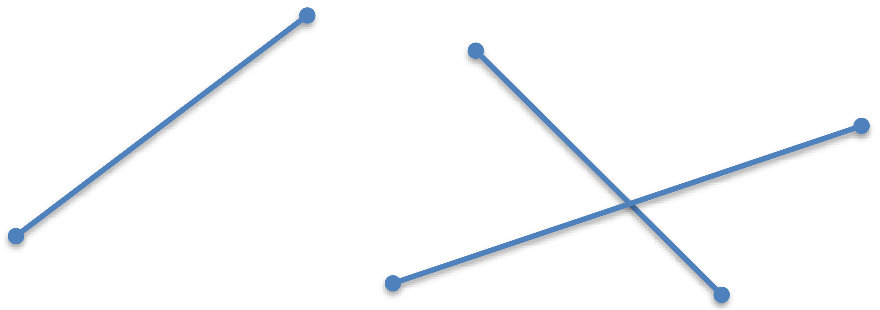
The two main types of wireframe meshes encoding are:

- *Line Lists*
- *Line Strips*

Line lists

Line lists encode each segment as a couple of two separate vertices.

To encode *N* segments, *2N* vertices are required.



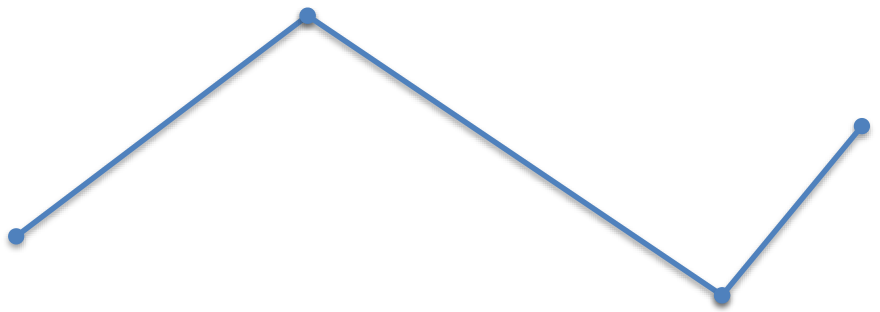
3 Lines =

6 Vertices

Line strips

Line strips encode a path of connected vertices.

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



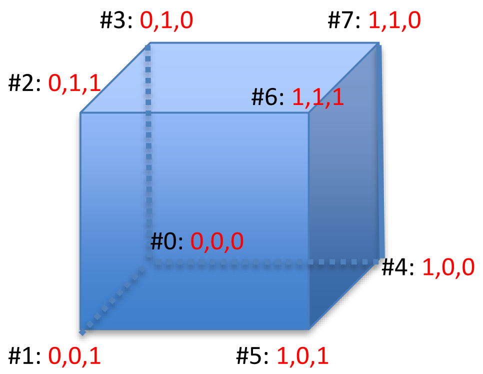
3 Lines =

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
```



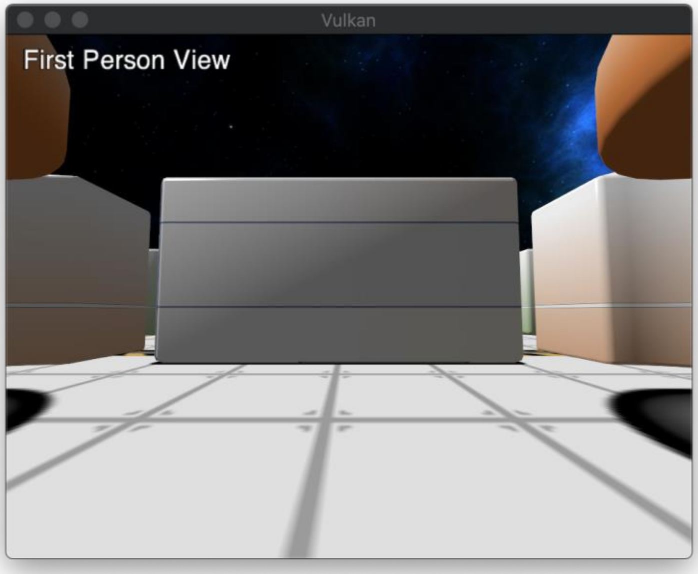
```
{0,1, 1,2, 2,3, 3,0,   // Index Array:
 4,5, 5,6, 6,7, 7,4,   // 24 indices
 1,5, 2,6, 3,7, 0,4}   // to the vertices of
                       // 12 lines
```

Line list: 12 x 2 x 3 x 4 = 288 bytes

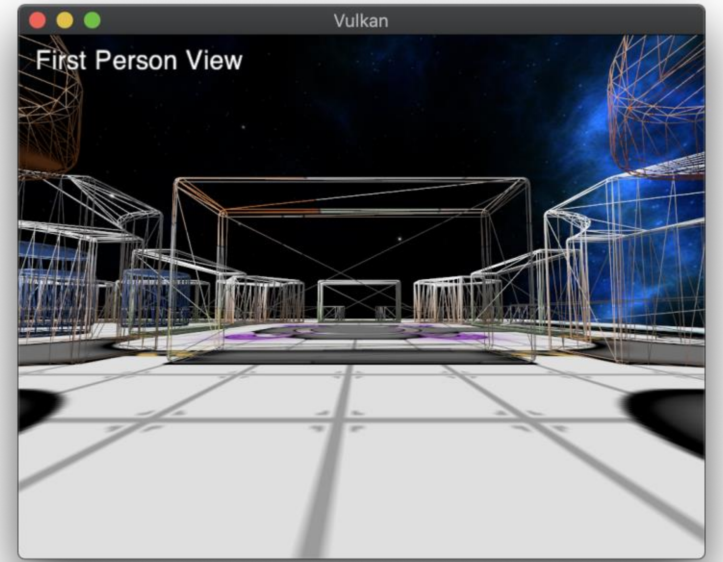
Indexed line list: 8 x 3 x 4 + 12 x 2 = ***120*** bytes

Drawing outlines

In addition, Vulkan also allows to draw standard objects using only the contour of their triangles: this however is just a way to simplify the creation of a wireframe object corresponding to the wireframe of a mesh.



Filled



Contours

Marco Gribaudo

Associate Professor

CONTACTS

Tel. +39 02 2399 3568
marco.gribaudo@polimi.it
https://www.deib.polimi.it/eng/home-page

(Remember to use the phone, since mails might require a lot of time to be answered. Microsoft Teams messages might also be faster than regular mails)