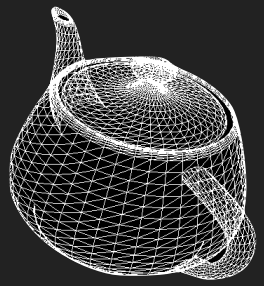
INTERACTIVE 3D GRAPHICS 2019

ROBERTO RANON - ROBERTO.RANON@UNIUD.IT
UNIVERSITÀ DI UDINE
WWW.DIMI.UNIUD.IT/RANON/INT3D.HTML

REPRESENTING AND DRAWING GEOMETRY

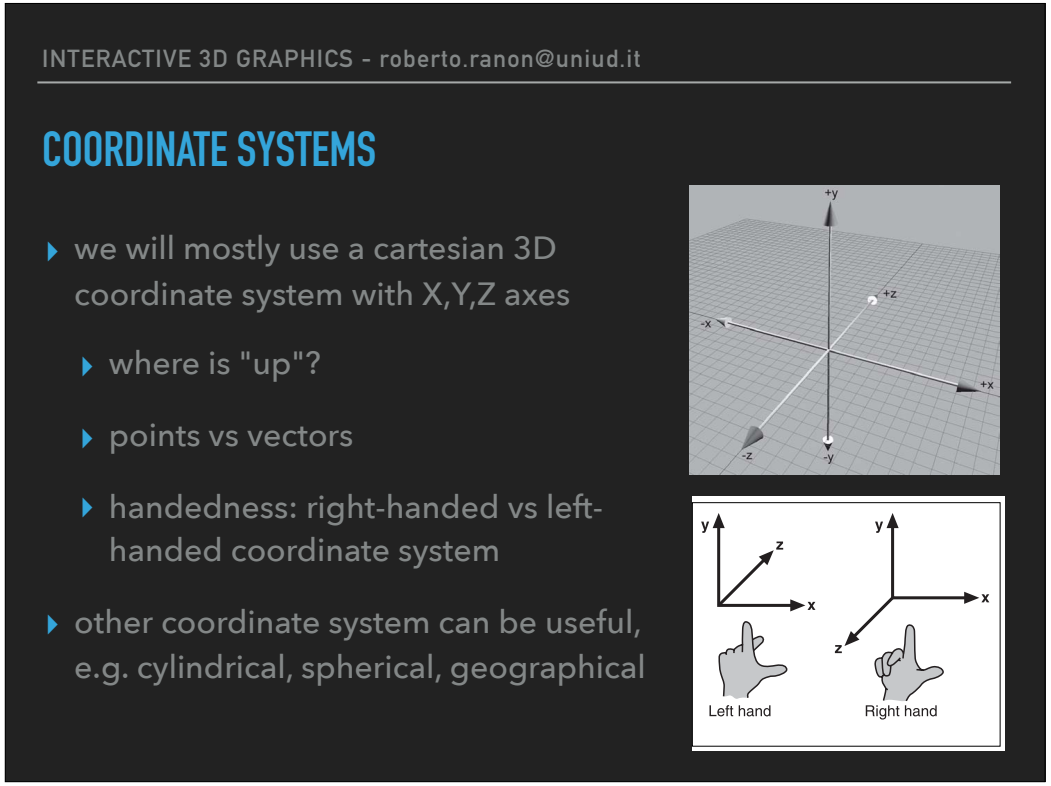
POLYGON MESHES

- interactive 3D graphics uses **triangle meshes** to (approximately) represent the surface of arbitrarily complex objects
- **tessellation**: the process of decomposing a complex surface, into a mesh of triangles or polygons



A polygon mesh is just a list of polygons. Triangle meshes are a special case of polygon meshes and are usually the representation of choice, since (i) every polygon mesh can be converted to a triangle mesh, and (ii) GPUs work with triangles, since they are planar and easy to rasterize unambiguously.

How many ways can you think for tessellating a sphere?



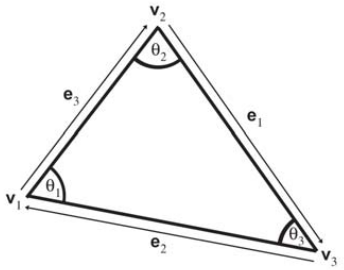
To represent and reason about 3D models of objects, we need a coordinate system that will allow us to locate points, compute distances and angles, etc. In most cases, 3D graphics uses a cartesian 3D coordinate system in which each point is identified by a (x,y,z) coordinate. It is also common to associate physical dimensions (left, right, up, down, forward, back) to semi-axes, but there is no universal convention. One of the most common is considering right as +X, up as +Y, and forward as -Z, but any other convention is ok. In the popular modeling program 3D Studio Max, for example, the default orientation of the axes is for +x to point right, +y to point forward, and +z to point up.

In a 3D space, every vector can represent either a geometrical point (a location in the space) or a direction, so we must treat it accordingly in our application.

All 3D cartesian coordinate spaces are not equal; some pairs of coordinate systems cannot be rotated to line up with each other. There are exactly two distinct types of 3D coordinate spaces: left-handed coordinate spaces and right-handed coordinate spaces. If two coordinate spaces have the same handedness, then they can be rotated such that the axes are aligned. If they are of opposite handedness, then this is not possible. Again, the choice of handedness is arbitrary. We will mostly use a right-handed coordinate system (as OpenGL does) for reasoning about the scene.

Answer these questions:
- how do you convert from a right-handed to a left-handed coordinate system?
- is 3D Studio Max using right-handed or left-handed coordinate system?

TRIANGLE REPRESENTATION IN 3D



- typically represented by its vertices ($\mathbf{v_1}$,$\mathbf{v_2}$,$\mathbf{v_3}$)
- a triangle has two sides or **faces** called **front face** and **back face**
- given a triangle ($\mathbf{v_1}$,$\mathbf{v_2}$,$\mathbf{v_3}$), the face we see when ($\mathbf{v_1}$,$\mathbf{v_2}$,$\mathbf{v_3}$) are in *counterclockwise order* is the **front face**
  - just the typical convention with right-handed coordinate systems
- a triangle lies on a plane whose normal can be computed taking any two edges

$$\mathbf{n} = \frac{\mathbf{e_1} \times \mathbf{e_2}}{\|\mathbf{e_1} \times \mathbf{e_2}\|}$$

The only kinds of graphical primitives we can send to the GPU are points, line segments, and triangles. Of these, triangles are by far the most important.

The reason for the distinction between front and back faces is that we might draw just one of them, or both, but drawing both requires more work than just one. Suppose we are drawing a triangle mesh that represents a sphere. If the camera will never be inside the sphere, there is no need to draw the faces of the triangles that are internal to the sphere.

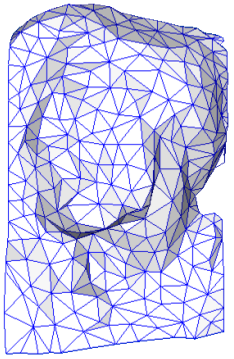
Note that what constitutes a front or a back face is just a convention. Some applications adopt a convention, others the opposite. OpenGL, for example, allows you to define what you consider a back or front face.

To compute the normal, take any two edges and compute the vector resulting from their cross product (the cross product yields a vector that is perpendicular to the original two vectors), and then normalize it. The normal exits from the front face.

Note: this is the moment to recall some algebra concepts. Take a look at the triangle figure. How do you compute e_1 from v_2 and v_1? How do you normalise a vector?

TRIANGLE MESH REPRESENTATION

- could be just a list of triangles (($\mathbf{v_{01}}$,$\mathbf{v_{02}}$,$\mathbf{v_{03}}$), ..., ($\mathbf{v_{n1}}$,$\mathbf{v_{n2}}$,$\mathbf{v_{n3}}$))
  - not efficient, it does not exploit adjacency (vertices are repeated in the list)
- the most common representation is called **indexed triangle mesh**:
  - a list of vertices ($\mathbf{v_1}$, $\mathbf{v_2}$, ..., $\mathbf{v_n}$)
  - a list of triangles, each one defined by three indices in the vertex list ((*1*,*2*,*3*), ..., (*i*,*j*,*k*))



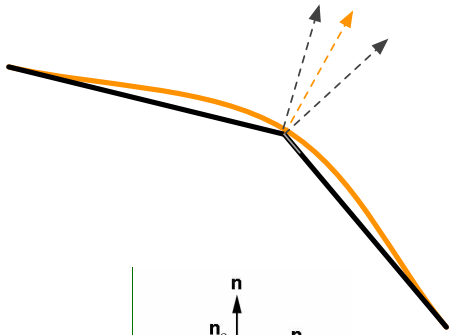
The advantage of the indexed triangle mesh representation is that shared vertices are not duplicated.

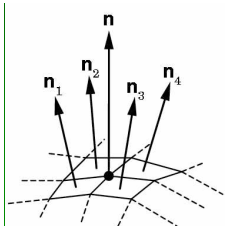
ADDITIONAL VERTEX DATA

- to compute colors from lighting, we need more geometric information:
  - **normals**: allow lighting to "see" the surface as curved, instead of tessellated
  - **uv coordinates**: necessary to map textures to surfaces
- these information are supplied per-vertex

SURFACE NORMALS

- there is no "true" surface normal at a vertex since it marks a discontinuity in the surface
- a polygon mesh is typically an approximation for some continuous surface, so we could use the surface normal of the continuous surface
  - this can be computed by averaging and normalizing the surface normals of the polygons that share that vertex





$$\mathbf{n} = \frac{\mathbf{n}_1 + \mathbf{n}_2 + \mathbf{n}_3 + \mathbf{n}_4}{|\mathbf{n}_1 + \mathbf{n}_2 + \mathbf{n}_3 + \mathbf{n}_4|}$$

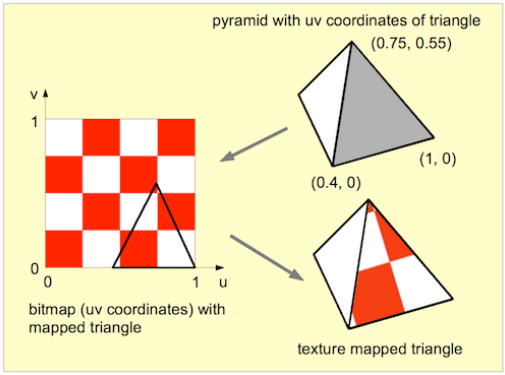
Normals are typically associated to vertices.

Sometimes the polygon representation is the exact representation of a surface (e.g. think of a 3D box) and we don't want to average the normals. The solution to this problem is to detach the faces; in other words, duplicate the vertices where there is a true geometric discontinuity.

Normals are typically needed in normalized form (i.e. with length = 1) for correct calculations with them.

UV COORDINATES

- UV coordinates are 2D coordinates that allow to apply an image to a surface
- any image, regardless of size, has each coordinate in the range [0,1]



We will explain all about UV coordinates and textures in future lectures. UV coordinates are typically defined in a 3D modeling program, such as 3DS Max.

2D coordinates are the most common case, but also 1D and 3D coordinates can be used

EXAMPLE: DRAWING A TRIANGLE IN THREE.JS

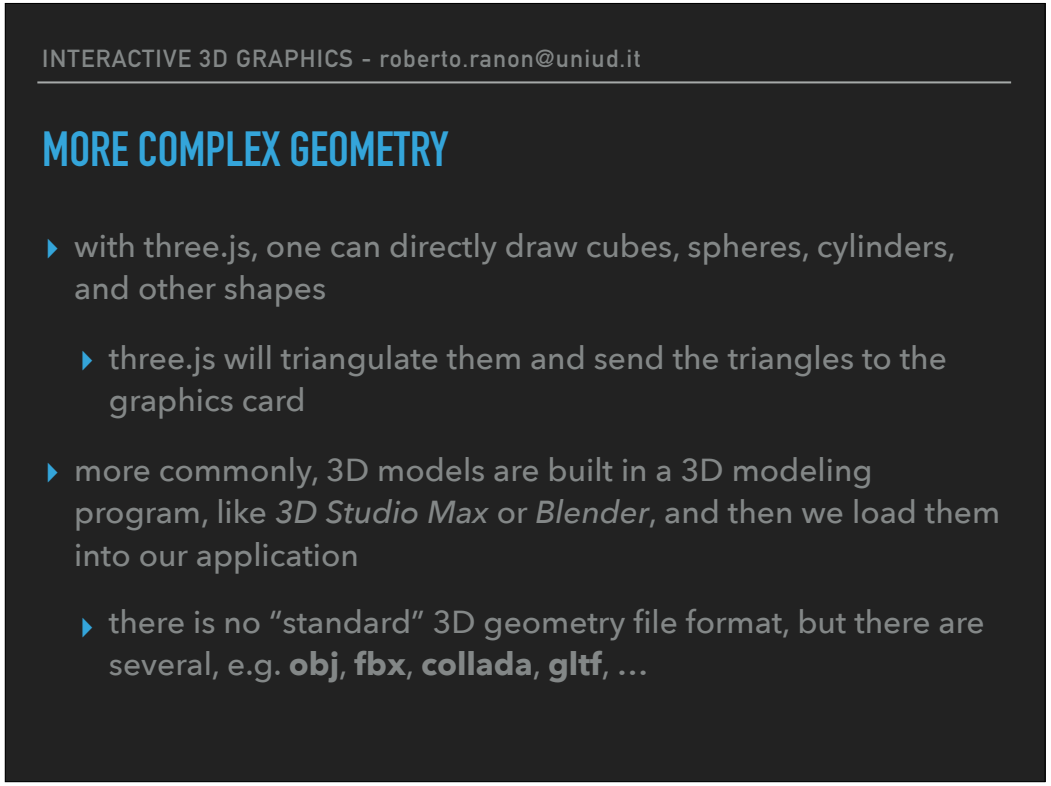
```
var material, geometry, mesh;
material = new THREE.MeshBasicMaterial( {
   color: 0xff0000,
   side: THREE.FrontSide}
);
geometry = new THREE.Geometry();
geometry.vertices.push(new THREE.Vector3(5, 10, 0));
geometry.vertices.push(new THREE.Vector3(5, 5, 0));
geometry.vertices.push(new THREE.Vector3(10, 5, 0));

geometry.faces.push(new THREE.Face3(0, 1, 2));

mesh = new THREE.Mesh(geometry, material);

scene.add(mesh);
```

This is example l3-draw-a-triangle.html in our examples repository.
Vertices and faces are arrays of the geometry object.
Try using THREE.BackSide and THREE.DoubleSide instead of THREE.FrontSide
Three.js has many vector operations available. See the link to the documentation http://mrdoob.github.io/three.js/docs/#Reference/Math/Vector3
A recent alternative to the Geometry object is BufferGeometry, which is a bit more complicated to use but more efficient in transmitting data to the graphics card, see https://threejs.org/docs/index.html#Reference/Core/BufferGeometry

EXAMPLE: DRAW A SQUARE

```
var material, geometry, mesh;
material = new THREE.MeshBasicMaterial({
    color: 0xff0000
});
geometry = new THREE.Geometry();
geometry.vertices.push(new THREE.Vector3(1, 2, 0));
geometry.vertices.push(new THREE.Vector3(1, 1, 0));
geometry.vertices.push(new THREE.Vector3(2, 1, 0));

geometry.vertices.push(new THREE.Vector3(1, 2, 0));
geometry.vertices.push(new THREE.Vector3(2, 1, 0));
geometry.vertices.push(new THREE.Vector3(2, 2, 0));

geometry.faces.push(new THREE.Face3(0, 1, 2));
geometry.faces.push(new THREE.Face3(3, 4, 5));

mesh = new THREE.Mesh(geometry, material);

scene.add(mesh);
```

This is one possible way of drawing a square: two triangles, with separated vertices (so, 6 vertices). Of course, there is better way, using just 4 vertices. Modify the example such as it creates just 4 vertices.



See the Extras / Geometries section on three.js documentation and related examples for details on the predefined geometries that three.js can draw. Of course, they are tessellated into triangles by three.js before being sent to the graphics card.

Blender is an open-source 3D modeling program available at www.blender.org. 3D Studio Max is probably the most common 3D modeling program for 3D application and games. Students can have a free 3-year license at http://www.autodesk.com/education/free-software.

The question of which file format to use for storing and loading 3D models is a complex one, because it depends on what tools you are using for producing 3D content. Three.js uses a file format based on JSON, and provides tools to convert from the quite common obj format to the JSON one. So you can, for example, build or load a 3D model in 3DS Max, export to obj format, convert to three.js JSON format, and then load the result into your application.

EXAMPLE: OBJ FILE FORMAT STRUCTURE

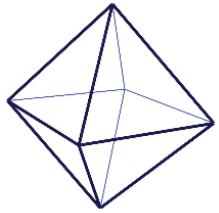
```
# List of Vertices, with (x,y,z[,w]) coordinates, w is optional.
v 0.123 0.234 0.345 1.0
v ...
...

# Texture coordinates, in (u,v[,w]) coordinates, w is optional.
vt 0.500 -1.352 [0.234]
vt ...
...

# Normals in (x,y,z) form; normals might not be unit.
vn 0.707 0.000 0.707
vn ...
...

# Face Definitions, using just vertex indices, vertex/texture coordinate indices, or
vertex/texture/normal indices
f 1 2 3
f 3/1 4/2 5/3
f 6/4/1 3/5/3 7/6/5
f ...
...
```

EXAMPLE: LOADING A OBJ FILE

```
<script src="lib/OBJLoader.js"></script>

…

// instantiate a loader
var loader = new THREE.OBJLoader();

// load a resource
loader.load(
   // resource URL
   'models/subwoofer_obj.obj',
  // Function when resource is loaded
  function ( object ) {
     scene.add( object );
    }
);
```

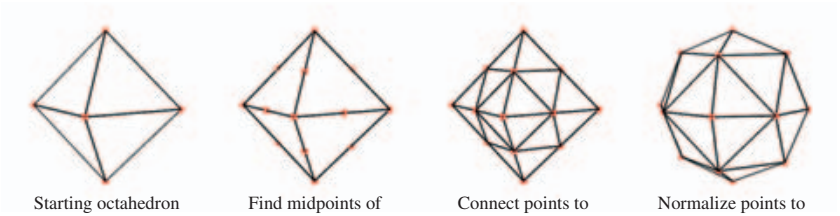
EXERCISES

1. Write a *three.js* program that renders a cube with six faces of different colors. Form the cube from six quadrilaterals, making sure that the front faces are facing outside.
2. Write a three.js program that draws a wireframe octahedron.



EXERCISES

3. Write a *three.js* program that draws a unit sphere centered on the world origin using recursive tessellation. The program should work as follows:
    1. draw an octahedron
    2. for a user-defined number of steps, apply a recursive subdivision of each triangle and move vertices, as in the figure below



Starting octahedron | Find midpoints of each edge | Connect points to form new triangles | Normalize points to coincide with surface of unit sphere

As an additional exercise, try to modify exercise 3 such that two spheres are drawn, at different positions.

REFERENCES

- Mathematics for 3D Game Programming and Computer Graphics, Appendix C1
- 3D Math Primer for Graphics and Game Development, Chapter 2

QUESTION

- this model is made of 5030 triangles. How many vertices does it have?
  - 2517
  - 5032
  - 10,060
  - Cannot know

ANSWER

- the Descartes-Euler Polyhedral Formula states that, for normal solid polyhedra:

  Vertices + Faces – Edges = 2

- we know that each edge is shared by 2 triangles, and each triangle has 3 edges:

  E = 3/2 * F

  V + F – 3/2 * F = 2

  V – F/2 = 2

With V – F/2 = 2 and 5030 faces:

V – 5030/2 = 2

V = 2517

Relationships for polyhedra made of triangles:

E = 3/2 * F, derived for any triangle

V – F/2 = 2, for a polyhedra

So: V – E/3 = 2

Knowing just the number of faces, edges, or vertices, you can get the other two.

V ~= F/2 ~= E/3