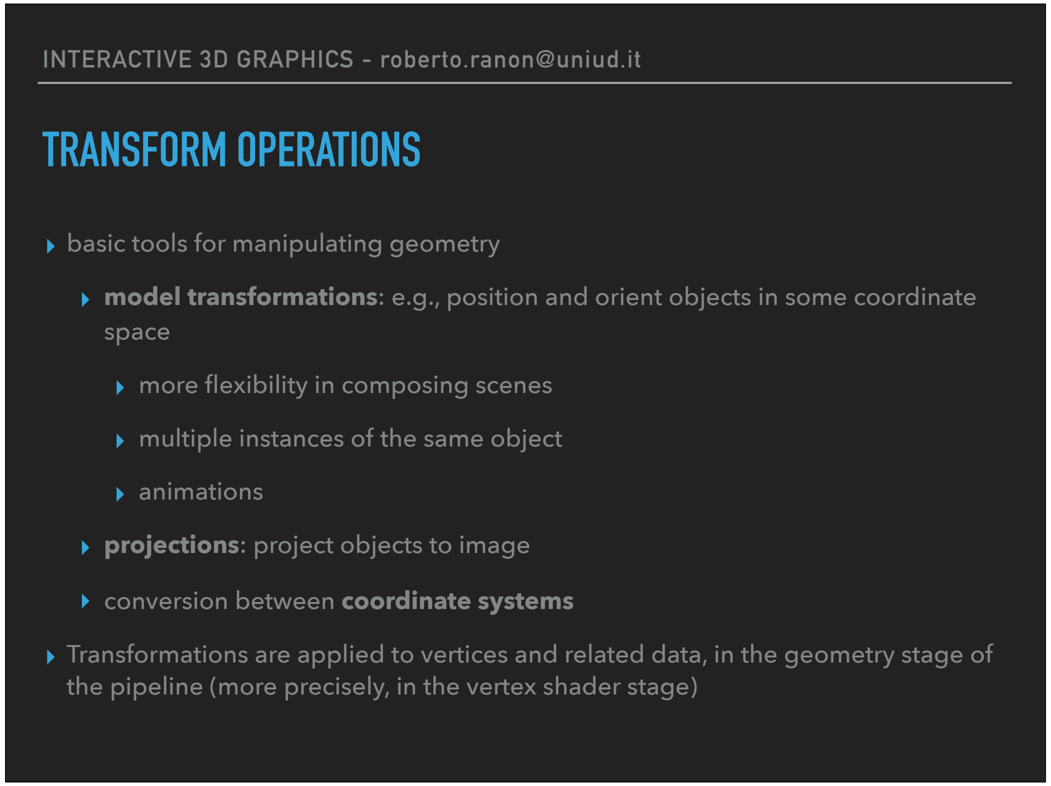
INTERACTIVE 3D GRAPHICS 2019
ROBERTO RANON - ROBERTO.RANON@UNIUD.IT
UNIVERSITÀ DI UDINE
WWW.DIMI.UNIUD.IT/RANON/INT3D.HTML

MODEL TRANSFORMS



TRANSFORM OPERATIONS

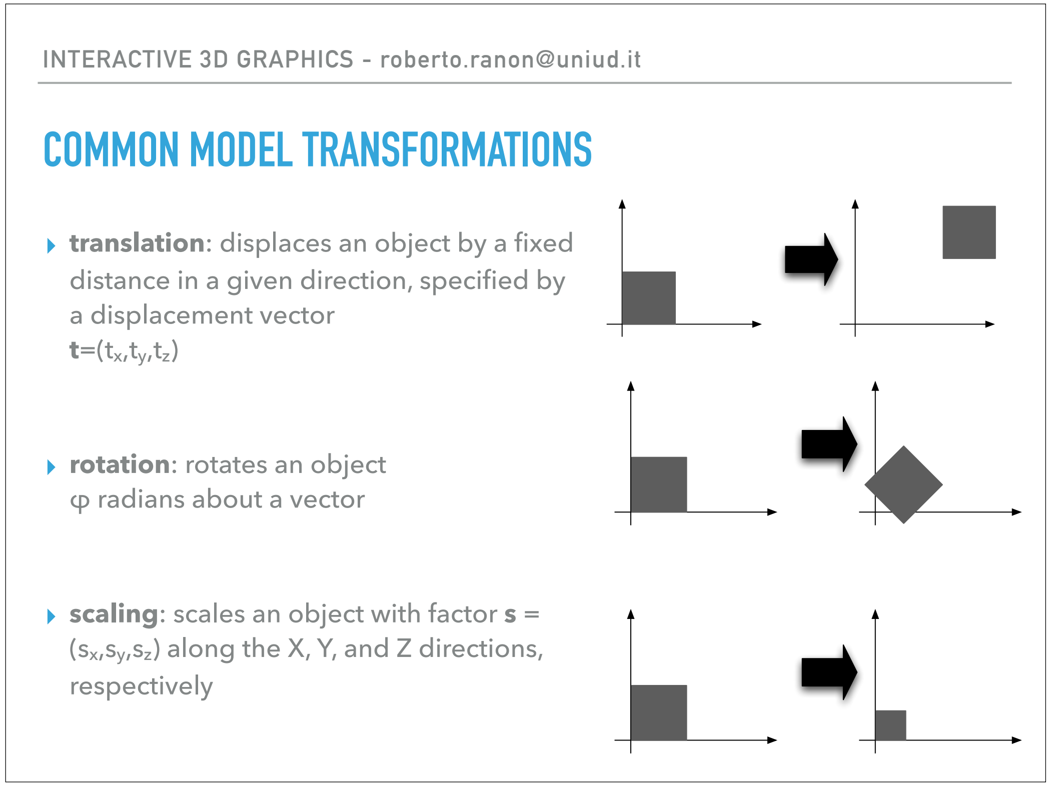
- basic tools for manipulating geometry
  - **model transformations**: e.g., position and orient objects in some coordinate space
    - more flexibility in composing scenes
    - multiple instances of the same object
    - animations
  - **projections**: project objects to image
  - conversion between **coordinate systems**
- Transformations are applied to vertices and related data, in the geometry stage of the pipeline (more precisely, in the vertex shader stage)

Model transformations add a lot of flexibility when sending geometric data to the pipeline.

For example, suppose we want to display a scene that integrates different objects that have been previously modeled. At modeling time, we might not know the final position and orientation of an object in the scene (e.g. because we are using models previously done by other people), so we model the object (i.e. write all vertices and related data) in a certain position/orientation. When we draw the scene, by applying model transforms, we can load the vertices and related data in memory, and then change the position/orientation of the object.

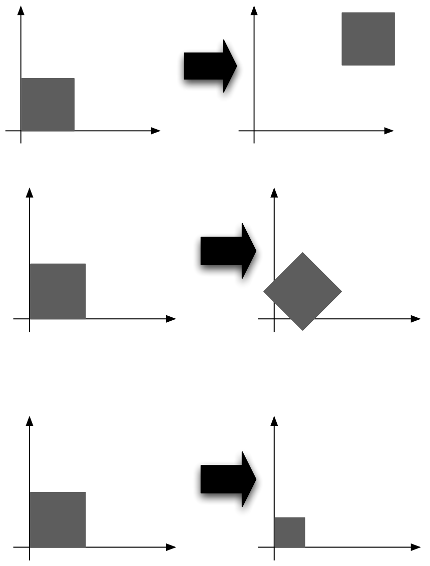
Another example is when a scene is composed by multiple instances of the same object, e.g. chairs in a classroom. Since these chairs are in different positions, the coordinates of their vertices are different (although identical, if we would draw them in the same position/orientation). However, it would be a waste of memory to have one array of vertices for each chair. Using transforms, we can share the same array of vertices (representing a chair in a certain position/orientation) and apply different transformations before rendering each copy of the chair.

A third example is with animations, where the position/orientation of an object varies in time. To do this, we simply apply different transformations before rendering each frame.



COMMON MODEL TRANSFORMATIONS

- **translation**: displaces an object by a fixed distance in a given direction, specified by a displacement vector **t**=(t_x,t_y,t_z)
- **rotation**: rotates an object φ radians about a vector
- **scaling**: scales an object with factor **s** = (s_x,s_y,s_z) along the X, Y, and Z directions, respectively

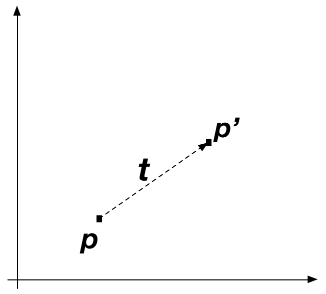


These are not the only possible model transformations, but are the most common. In particular, with affine transformation we can also express a fourth type of transformation called *shear*.

Note that the images show 2D transformations, but we are going to work with 3D ones. We will now consider how the three operations work on a single point in 3D space.

TRANSLATION (POINT)

- **translate** a 3D point ***p***=(p_x,p_y,p_z) by a displacement vector ***t*** =(t_x,t_y,t_z)



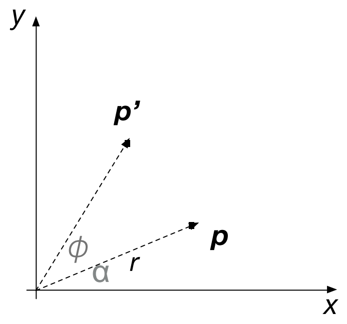
$$\begin{pmatrix} p'_x \\ p'_y \\ p'_z \end{pmatrix} = \begin{pmatrix} p_x \\ p_y \\ p_z \end{pmatrix} + \begin{pmatrix} t_x \\ t_y \\ t_z \end{pmatrix}$$

We start by considering points. This is how we translate a point. Some simple question, but it is important that you can come up with an answer:

- How do we apply two translations? What is the resulting mathematical form?
- How can you invert a translation T, i.e. derive T^{-1}?

ROTATION (POINT)

- **rotate** a 3D point ***p***=(p_x,p_y,p_z) by an angle φ around a vector
    - we consider a simple case: rotate by angle φ around Z



$$\mathbf{p} = (p_x, p_y) = (rcos\alpha, rsin\alpha)$$

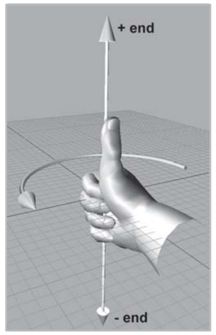
$$\mathbf{p}' = (p'_x, p'_y) = r(cos(\alpha+\phi), sin(\alpha+\phi)) =$$
$$r(cos\alpha cos\phi + sin\alpha sin\phi, cos\alpha sin\phi + sin\alpha cos\phi) =$$
$$(p_x cos\phi - p_y sin\phi, p_x sin\phi + p_y cos\phi)$$

$$\begin{pmatrix} p'_x \\ p'_y \\ p'_z \end{pmatrix} = \begin{pmatrix} cos\phi & -sin\phi & 0 \\ sin\phi & cos\phi & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} p_x \\ p_y \\ p_z \end{pmatrix}$$

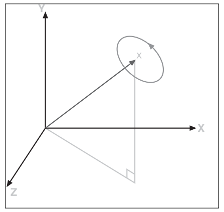
In the same way, one can obtain a rotation around X or Y. Some simple question, but it is important that you can come up with an answer:
- How do we apply two rotations? What is the resulting mathematical form?
- How can you invert a rotation R, i.e. derive R^{-1}? There is a much easier way than computing the inverse of the 3x3 matrix…

ROTATION VERSE AND ROTATION AROUND AN ARBITRARY VECTOR

- which way is positive?
  - in right-handed coordinate systems, we use the right-hand rule:
    1. grab the axis of rotation with the right hand
    2. put thumb towards positive axis direction
    3. closing your hand is a positive rotation



- how do you rotate around an arbitrary vector passing through the origin?
  - one way of deriving the matrix is by composing three rotations around main axes



With left-handed coordinate systems, we use our left hand in the same way.

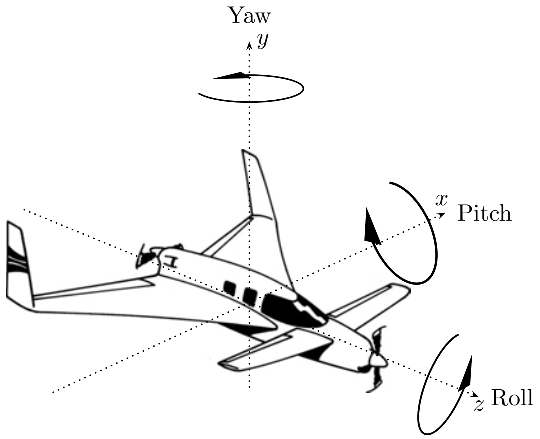
Say R is the rotation we want around some axis passing through the origin. The idea is to create a rotation M such that $R = M^{-1}R_zM$, i.e.:
- first, we transform the arbitrary axis R into Z (M)
- then, rotate about Z (R_z)
- undo the first rotation (M-1)

M can be obtained by first rotating about the Z axis such that the rotation vector sits on the XZ plane, and then rotating about the y axis such that it matches the Z axis. Try to derive M as an exercise.

The same technique can be used to derive the matrix for rotating about an axis not passing through the origin.

EXPRESSING ROTATIONS WITH EULER ANGLES

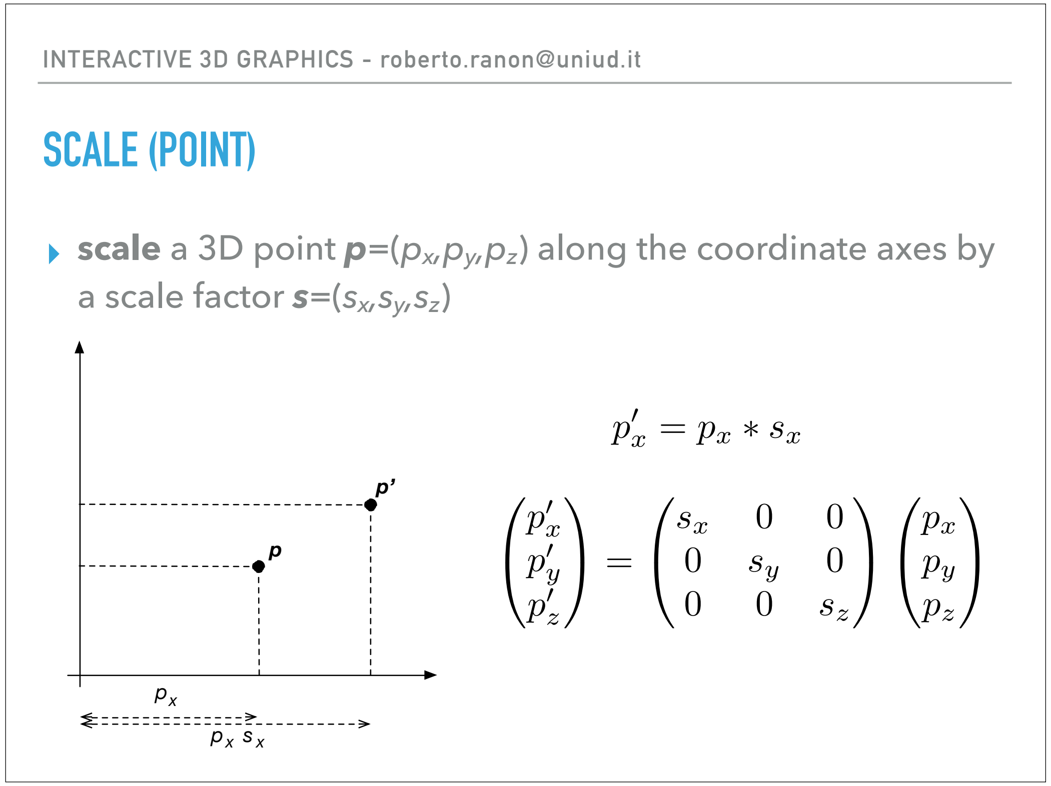
- represent rotations without using a matrix
- define a rotation in terms of three successive rotations around the X, Y, and Z axes, respectively by angles called **pitch**, **yaw** (or **head**), and **roll**
- typical order of application can be Z, Y, X, i.e. the final rotation would be
- *R_x(pitch) R_Y(yaw) R_z(roll)*
- but other conventions can be used



Demo

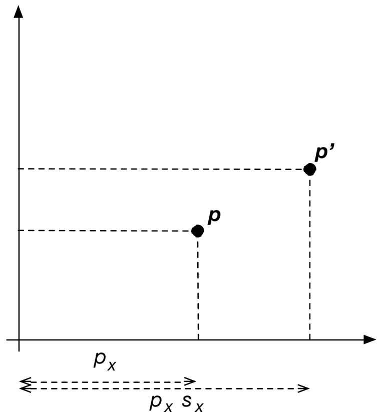
How can we express every possible rotation using just rotations around the coordinate system axes? It turns out we can, using so-called Euler angles.

Run the demo. What happens if you rotate around two or more of these axes? The frame of reference changes for each rotation.
Euler angles are handy for flight simulators, robotics applications, and even mobile applications, as they can be used to describe the orientation of the mobile device itself.
However, Euler angles also can run into limitations, such as the problem of gimbal lock. For example, say I set the Y angle to 90 degrees. You can now see that the X rotation and Z rotation have exactly the same effect. We've lost a degree of freedom under these conditions, limiting how we can move away from this orientation.



SCALE (POINT)

- **scale** a 3D point ***p***=(p_x,p_y,p_z) along the coordinate axes by a scale factor ***s***=(s_x,s_y,s_z)



$$p'_x = p_x * s_x$$

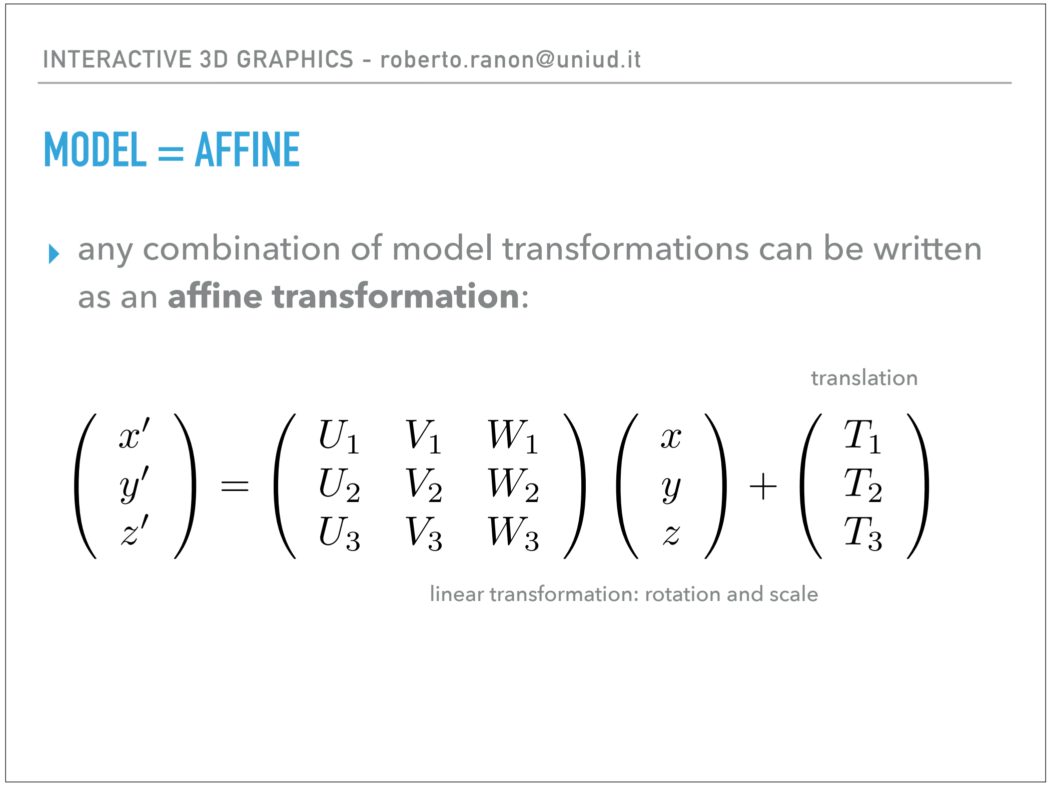
$$\begin{pmatrix} p'_x \\ p'_y \\ p'_z \end{pmatrix} = \begin{pmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & s_z \end{pmatrix} \begin{pmatrix} p_x \\ p_y \\ p_z \end{pmatrix}$$

If we apply the same scale in every direction, we are performing a **uniform** scale. Uniform scale preserves angles and proportions. If we apply different scale factors in different directions, we get a non-uniform scale.

A negative value on one or three of the components of the matrix gives a reflection matrix; if only two factors are negative, we will rotate π radians.

One can also scale in arbitrary orthogonal directions, using a process similar to rotations about an arbitrary axis.

How can you invert a scale transformation?



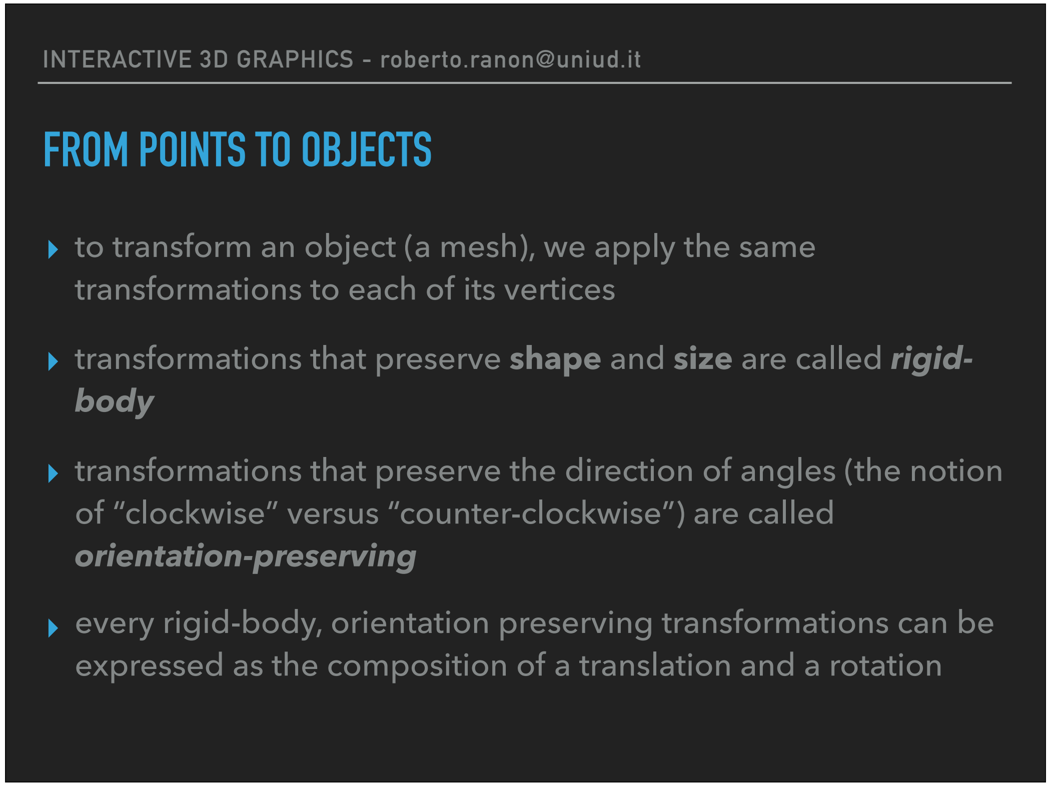
MODEL = AFFINE

- any combination of model transformations can be written as an **affine transformation**:

$$\begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} = \underset{\text{linear transformation: rotation and scale}}{\begin{pmatrix} U_1 & V_1 & W_1 \\ U_2 & V_2 & W_2 \\ U_3 & V_3 & W_3 \end{pmatrix}} \begin{pmatrix} x \\ y \\ z \end{pmatrix} + \overset{\text{translation}}{\begin{pmatrix} T_1 \\ T_2 \\ T_3 \end{pmatrix}}$$

Geometrically, an affine transformation in Euclidean space is one that preserves:
- the collinearity relation between points; i.e., the points which lie on a line continue to be collinear after the transformation
- ratios of distances along a line;

In general, an affine transformation is composed of linear transformations (rotation, scaling or shear) and a translation (or "shift"). A combination of affine transformations is still affine.

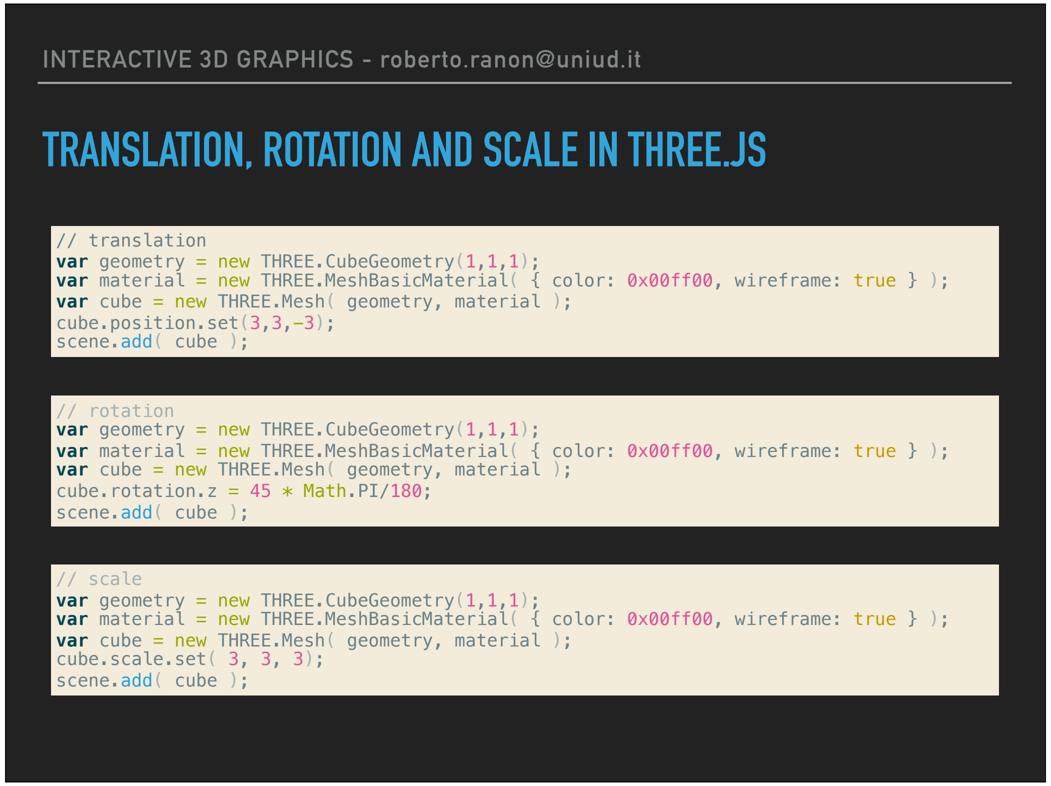


FROM POINTS TO OBJECTS

- to transform an object (a mesh), we apply the same transformations to each of its vertices
- transformations that preserve **shape** and **size** are called ***rigid-body***
- transformations that preserve the direction of angles (the notion of “clockwise” versus “counter-clockwise”) are called ***orientation-preserving***
- every rigid-body, orientation preserving transformations can be expressed as the composition of a translation and a rotation

We defined transformations as acting on a single point at a time, but of course, a transformation also acts on arbitrary geometric objects since the geometric object can be viewed as a collection of points and, when the transformation is used to map all the points to new locations, this changes the form and position of the geometric object.

Rigid, orientation-preserving transformations are often used to animate objects.



TRANSLATION, ROTATION AND SCALE IN THREE.JS

```
// translation
var geometry = new THREE.CubeGeometry(1,1,1);
var material = new THREE.MeshBasicMaterial( { color: 0x00ff00, wireframe: true } );
var cube = new THREE.Mesh( geometry, material );
cube.position.set(3,3,-3);
scene.add( cube );
```

```
// rotation
var geometry = new THREE.CubeGeometry(1,1,1);
var material = new THREE.MeshBasicMaterial( { color: 0x00ff00, wireframe: true } );
var cube = new THREE.Mesh( geometry, material );
cube.rotation.z = 45 * Math.PI/180;
scene.add( cube );
```

```
// scale
var geometry = new THREE.CubeGeometry(1,1,1);
var material = new THREE.MeshBasicMaterial( { color: 0x00ff00, wireframe: true } );
var cube = new THREE.Mesh( geometry, material );
cube.scale.set( 3, 3, 3);
scene.add( cube );
```

The mesh class inherits from the Object3D class (see three.js documentation). The Object3D stores, in the **position** field, the object translation from the coordinate system origin. **Note: you can't do: cube.position = new THREE.Vector3(0,2,0) because object.position is immutable**

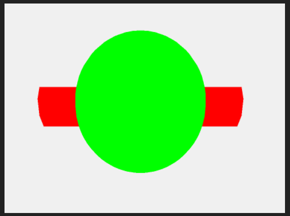
The rotation field stores a rotation around an axis parallel to the coordinate system X, Y, and Z axes (expressed in radians, using the right-hand rule) and passing through the object center (i.e. the origin of the coordinate system if there is no translation).

The scale field stores three scale factors along axes that are parallel to the coordinate system and pass through the center of the object (therefore the center of the object stays where it is regardless of the scaling factors).

If you try to combine more transformations, what happens? Let's discover it with an exercise.

EXERCISE: COMBINING TRANSFORMATIONS

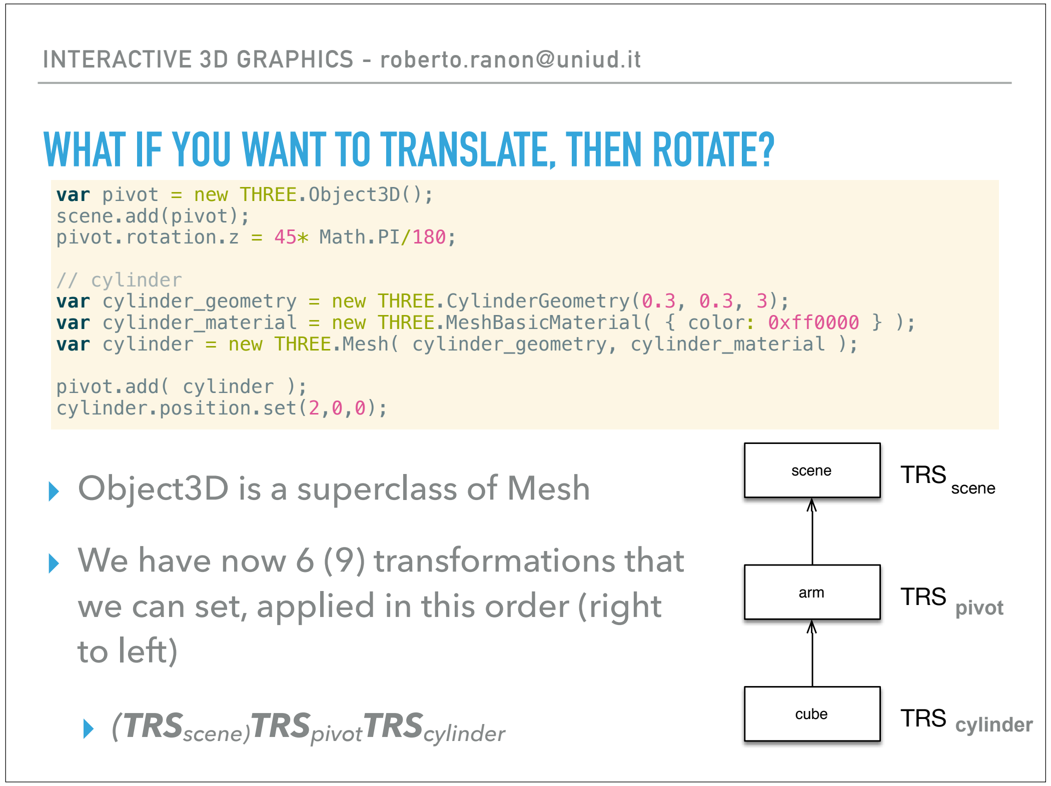
Edit example l4-combining-transformations.html such that the two objects are in the configuration in the figure



```
// sphere
var sphere_geometry = new THREE.SphereGeometry(1,32,32);
var sphere_material = new THREE.MeshBasicMaterial( { color: 0x00ff00 } );
var sphere = new THREE.Mesh( sphere_geometry, sphere_material );
sphere.position.set( 0, 2, 0);
scene.add( sphere );

// cylinder
var cylinder_geometry = new THREE.CylinderGeometry(0.3, 0.3, 3);
var cylinder_material = new THREE.MeshBasicMaterial( { color: 0xff0000 } );
var cylinder = new THREE.Mesh( cylinder_geometry, cylinder_material );
// your code goes here
cylinder.position.set( 0, 2, 0);
cylinder.rotation.z = 90* Math.PI/180;
scene.add( cylinder );
```

If you combine more transformations, the order of application is scale, rotation, and then translation



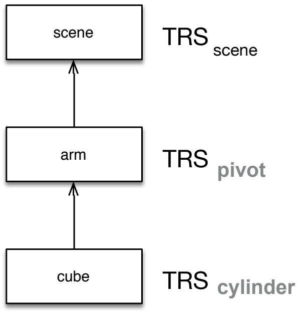
WHAT IF YOU WANT TO TRANSLATE, THEN ROTATE?

```
var pivot = new THREE.Object3D();
scene.add(pivot);
pivot.rotation.z = 45* Math.PI/180;

// cylinder
var cylinder_geometry = new THREE.CylinderGeometry(0.3, 0.3, 3);
var cylinder_material = new THREE.MeshBasicMaterial( { color: 0xff0000 } );
var cylinder = new THREE.Mesh( cylinder_geometry, cylinder_material );

pivot.add( cylinder );
cylinder.position.set(2,0,0);
```

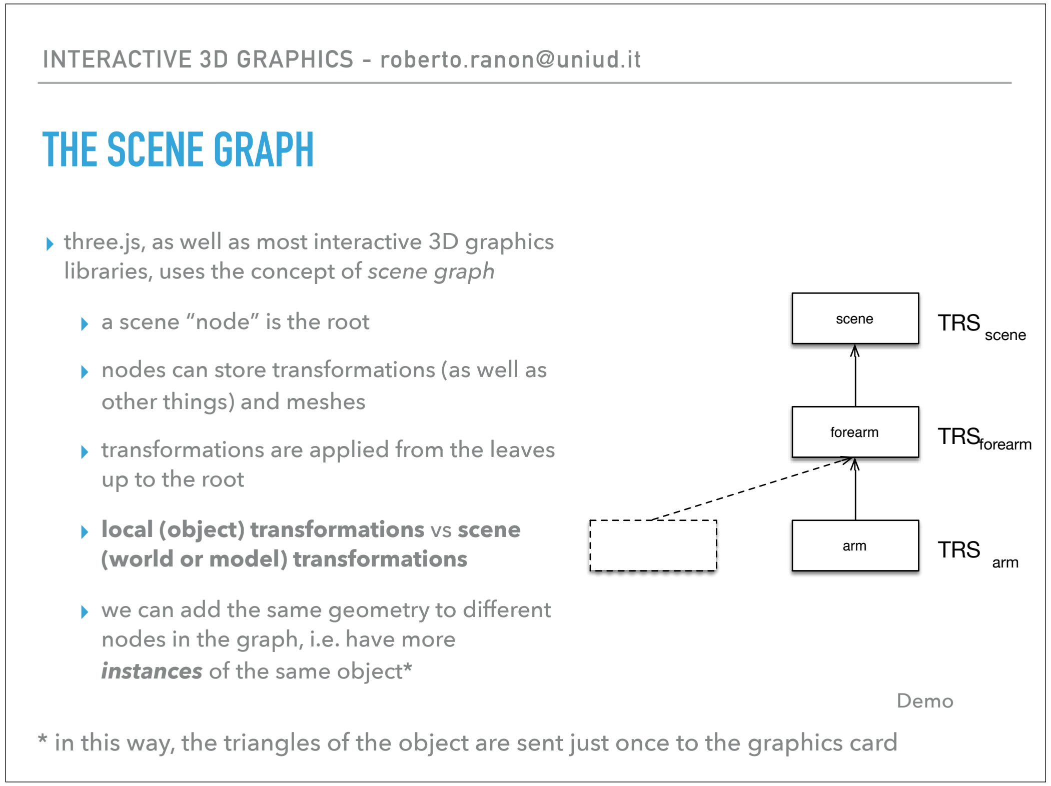
- Object3D is a superclass of Mesh
- We have now 6 (9) transformations that we can set, applied in this order (right to left)
    - $(TRS_{scene})TRS_{pivot}TRS_{cylinder}$



This is example 4-scene-hierarchy.html in our repository. Note that the cylinder is now first translated to (2,0,0), then rotated around the Z axis (not passing through the object center this time!).

In general, we know that order of transformations matters (it will change the final result). This is true in general, but might not be true in some particular cases, e.g.:
- can a series of translations be done in any order?
- can a series of rotations be done in any order?
- can a series of scales be done in any order?
- can a series of rotations and translations all along the same axis be done in any order?

Second question: Say I apply a uniform scale to an object centered at the origin and then translate it (case A). I start again and apply the translation first to the object, then scale it (case B). Which is true?
- Neither object’s center has moved from the origin.
- The object is the same size in both cases.
- The object is in the same position in both cases.
- The object’s center has moved farther in case B than case A.



THE SCENE GRAPH

- three.js, as well as most interactive 3D graphics libraries, uses the concept of *scene graph*
  - a scene “node” is the root
  - nodes can store transformations (as well as other things) and meshes
  - transformations are applied from the leaves up to the root
  - **local (object) transformations** vs **scene (world or model) transformations**
  - we can add the same geometry to different nodes in the graph, i.e. have more ***instances*** of the same object*

Demo

* in this way, the triangles of the object are sent just once to the graphics card

By properly organising the scene objects into a graph, we can add a lot of flexibility in composing and animating scenes, because transformations can be applied hierarchically.

Within a scene graph, every node (Object3D in three.js) can be seen as being in two different coordinate systems:
- the **local coordinate system** (also called **object space**) which is specific to it
- the **scene coordinate system** (also called **world space**) which is global to the scene, and where the object is positioned by all the hierarchy of transformations that we find by starting at the object node, and going up in the scene graph to the root. The transformations that put the object from object space to world space are also called **model transformations**

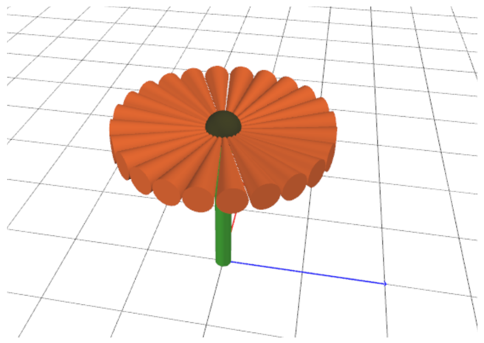
Depending on what we need to do, we can choose the coordinate space in which we want to reason or apply transformations. Some examples of questions that are typically asked in world space include:
- What is the position and orientation of each object?
- What is the distance between two objects?
- How does an object gets from point A to point B in the scene?

Some examples of questions that can be asked in object space are:
- Is there another object near me?
- In what direction is it? Is it in front of me?

To practice with the scene graph and transformations, you can use the the three.js interactive scene editor at http://mrdoob.github.com/three.js/editor/. First define a directional light, then add an object to the scene.

EXERCISE: MAKE A FLOWER

- stem: a cylinder
- stamen: a sphere
- petal: a cylinder with radiusTop = 0
- use instancing for the petals, i.e. create one petal geometry and add it multiple times to the scene



REFERENCES

- Real-Time Rendering, 2nd edition
  - ch. 3 (except quaternions, vertex blending)
- Real-Time Rendering, 3rd edition
  - ch. 4 (except morphing, quaternions, vertex blending)
- Mathematics for 3D game programming and Computer Graphics
  - ch. 3 (except quaternions)
- 3D Math Primer for Graphics and Game Development
  - ch. 3 (except inertial spaces)
  - ch 8 (except shear matrices)