INTERACTIVE 3D GRAPHICS 2019

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

MODEL TRANSFORMS PART 2

APPLYING TRANSFORMATIONS

- to transform an object, we compute $\boldsymbol{p'=Mp+T}$ for each of its vertices
- what about vertex data, e.g. normals?
  - we need to transform them too
  - however, normals should not be translated
- we can solve the problem by going to ***4D homogeneous space*** with the following correspondence:
  - a 3D point is extended to 4 dimensions by setting its fourth coordinate, *w*, to **1**
    - more generally, the mapping for points is: $(x,y,z) \rightarrow (x/w, y/w, z/w, w)$
  - a 3D direction is extended to 4 dimensions by setting its fourth coordinate, *w*, to **0**

More generally, the mapping between 3D and 4D homogeneous space is (x,y,z) <-> (x/w, y/w, z/w, w) for w≠0, i.e. the 3D point p = (x,y,z) corresponds to the line in 4D connecting the origin to the point (x,y,z,1).

The correspondence we have established has the advantage of allowing us to distinguish between vectors that represent points, and vectors that represent directions, and treat them differently when doing transformations.

TRANSFORMATIONS IN 4D HOMOGENEOUS SPACE

$$F = \left[\begin{array}{ccc|c} & M & & T \\ \hline 0 & 0 & 0 & 1 \end{array}\right] = \begin{bmatrix} M_{11} & M_{12} & M_{13} & T1 \\ M_{21} & M_{22} & M_{23} & T2 \\ M_{31} & M_{32} & M_{33} & T3 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- multiplying ***F*** by ***p***=(*x,y,z,1*) transforms the coordinates exactly as ***Mp*** + ***T***
- does not alter the *w* value (it stays 0 or 1)
- we can then express any 3D affine transformation with just a matrix
- combination of transformations is performed by just matrix multiplication

Note that we are NOT doing translations, rotations, etc. in 4D. We are just using a 4D space to perform transformations in 3D.

MODEL TRANSFORMS IN 4D

$$\mathbf{T}(\mathbf{t}) = \mathbf{T}(t_x, t_y, t_z) = \begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$\mathbf{R}_x(\phi) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & cos(\phi) & -sin(\phi) & 0 \\ 0 & sin(\phi) & cos(\phi) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad \mathbf{R}_y(\phi) = \begin{pmatrix} cos(\phi) & 0 & sin(\phi) & 0 \\ 0 & 1 & 0 & 0 \\ -sin(\phi) & 0 & cos(\phi) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$\mathbf{R}_z(\phi) = \begin{pmatrix} cos(\phi) & -sin(\phi) & 0 & 0 \\ sin(\phi) & cos(\phi) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$\mathbf{S}(\mathbf{s}) = \mathbf{S}(s_x, s_y, s_z) = \begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Which ones affect points and direction vectors? Which ones only affect points (and not direction vectors)?

WORKING WITH MATRICES IN THREE.JS

- Internally, three.js (as well as any 3D graphics application) stores **4D matrices** to represent and apply model transformations
  - as well as using `Vector4` for points and vectors
- in particular, each `Object3D` has two matrices: one for the local transformations, and one for the world transformations (i.e. the combinations of all transformations up to the scene root node)
  - the position, rotation and scale fields, if changed, alter the two matrices accordingly

You can see it for yourself by inspecting an Object3D in the debugger.

WORKING WITH MATRICES IN THREE.JS

- we can directly set the transformation matrix (in row-major form)
  - e.g., to perform translation before rotation
- `matrixAutoUpdate` must be set to false to disable the position, rotation and scale fields, that would automatically overwrite our matrix
- we can find the inverse of a transformation using the `getInverse` function of `Matrix4`

```
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.matrix.set( 1, 0, 0, 3,
                 0, 1, 0, 3,
                 0, 0, 1, -3,
                 0, 0, 0, 1);

cube.matrixAutoUpdate = false;
```

Alternatively, one can use:

cube.matrix.makeTranslation(x, y, z);

Take a look at all the methods available for the Matrix4 class in three.js: http://threejs.org/docs/#Reference/Math/Matrix4

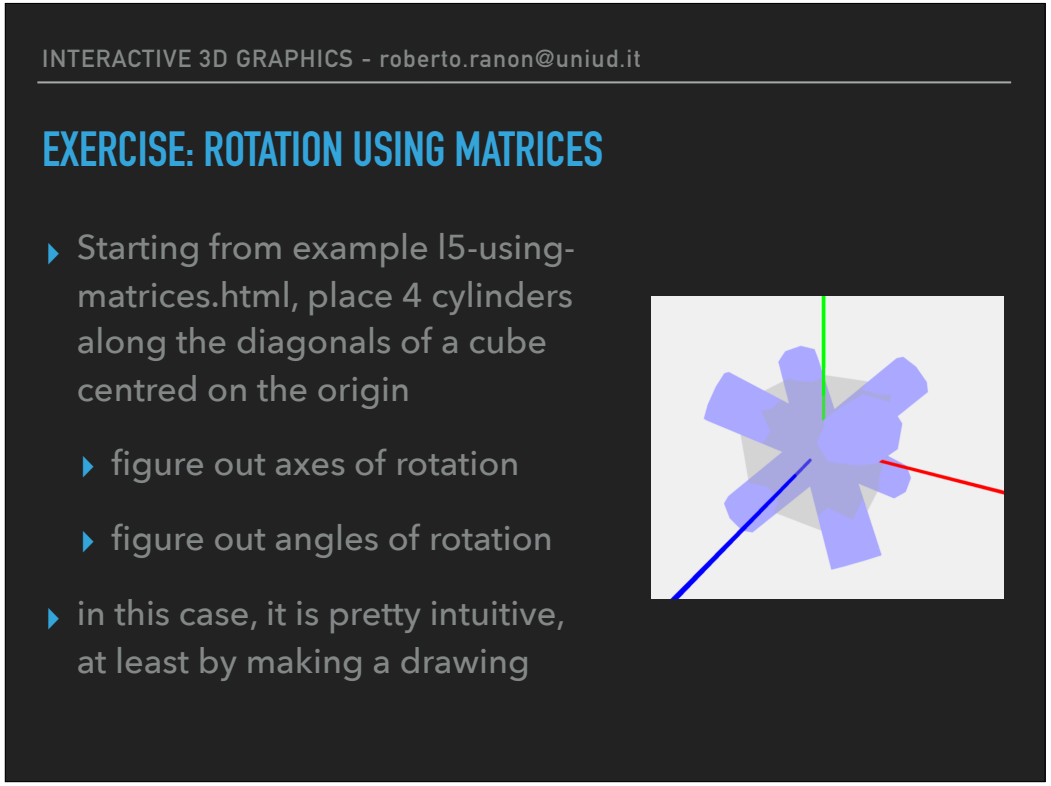
ROTATION USING MATRICES

- by using matrices, we can set an arbitrary axis of 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 );

var axis = new THREE.Vector3(1,0,0);
var theta = 45 * Math.PI / 180.0;

cube.matrix.makeRotationAxis( axis, theta );
cube.matrixAutoUpdate = false;
```



Doing things like this with Euler angles is hard. With a rotation matrix, it just reduces to compute a proper axis and angle of rotation.

Each cylinder starts with its axis along the Y axis. Make two drawings of it, one from the side (X and Y axes), and one from the top (XZ axes). Suppose the cube has side equal to 2. We want to rotate the top of the cylinder such as it goes to the point (1,1,1).

A vector rotating around an axis forms a circle. The cylinder's axis on the Y axis and the final axis through the corner of the cube define a circle whose center is at the origin. There can only be one plane that contains the circle, and there is only one axis direction, or its opposite direction, that is perpendicular to this plane. So the axis of rotation can be (1,0,-1) or (-1,0,1).

The angle of rotation is clearly 45 (or -45) degrees in this case.

Repeat, with the needed variations, for all axes.

FINDING THE AXIS AND ANGLE OF ROTATION

we can find the angle of rotation between two vectors (the original and the rotated one) by using vectors dot product

```
var maxCorner = new THREE.Vector3( 1, 1, 1 );
var minCorner = new THREE.Vector3( -1, -1, -1);
var rotated_cyl_axis = new THREE.Vector3();
rotated_cyl_axis.subVectors( maxCorner, minCorner );
rotated_cyl_axis.normalize();
var angle = Math.acos( rotated_cyl_axis.dot(new THREE.Vector3(0,1,0)));
```

we can find the axis of rotation between two vectors (the original and the rotated one) by using vectors cross product

```
var rotation_axis = new THREE.Vector3();
// remember:right-hand rule
rotation_axis.crossVectors(new THREE.Vector3(0,1,0),rotated_cyl_axis);
// rotation axis given to makeRotationAxis must be normalised
rotation_axis.normalize();
```

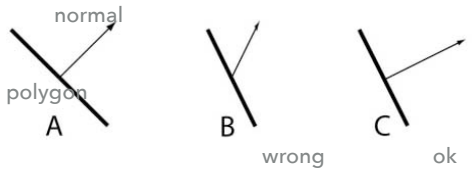
Recall that, for any two normalised vectors v1 and v2, $v1 \cdot v2 = \cos \alpha$, where α is the angle between the two vectors.

The only problem with the cross product is when the two vectors are the same, or in opposite directions. In this case, the length of the cross product is zero, i.e. the axis of rotation would be (0,0,0). You can use the dot product to find if the two vectors are the same (no need for rotation) or are 180 degrees apart

ABOUT SURFACE NORMALS

- the same matrix that we use to transform polygon vertices cannot always be used to transform normals

scaled by 0.5 along the x dimension



- If a geometry has been transformed by M, we must use the matrix $N=(M^{-1})^T$ or $N=(M^T)^{-1}$ to transform its normals

This does not affect (i.e. we can still use M) translations, rotations, and uniform scalings (in the last case, only the length of the transformed normal is affected, and the scaling factor can be used to renormalize the normals).

Typically, graphics library (including three.js) will correctly handle the situation for us, i.e., if we use non-uniform scalings, three.js will transform the normals using the transpose of the inverse.

MATRICES DRIFTING

- matrices operations suffer from floating-point math errors
  - the more operations you do, the greater the error can be
- example: if you compute a rotation by repeatedly applying (to the current rotation) a rotation matrix that has small values, after thousands of tiny rotations the result gradually **drifts** away from the intended value, and could even not be a rotation anymore
- when animating transforms, it is much better to store separate values for position, rotation, scale, and recompute the matrix from scratch at each frame starting from these values

EXERCISE

- Develop a web application that visualizes a solar system with:
  - a central sun (not rotating)
  - a planet revolving around the sun every 365 days, and, in addition, rotating about an axis which is tipped over about 25 degrees from its vertical axis, once per day
  - a moon revolving around the planet 12 times per year, and rotating around its vertical axis;
- use wireframe spheres of different colours for the sun, the planet and the moon. To make animations visible, suppose a year will last 365 seconds. You can use the javascript Date.now() method that returns the number of milliseconds elapsed since 1 January 1970 00:00:00 UTC.

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)