

## programming transformations

- what are convenient coordinate systems to work with
  - how do describe where the objects are relative to each other
  - how to get everything ready for the camera
- how to deal with this in an OpenGL program

## world frame

- world frame  $\vec{w}^t$ 
  - starting point for representations

## object coordinate systems

- for object i:  $\vec{o}_i^t$
- express parts of the object with coordinates relative to this
- so we could use “canonical” coordinates for a cube.
- to move object, just update  $\vec{o}_i^t$
- relationship between world and object is expressed as

$$\vec{o}_i^t = \vec{w}^t O_i$$

- in the computer program we store only  $O_i$ 
  - we update  $\vec{o}_i^t$  by updating  $O_i$ !
  - called `objectFrame` in code
- note: we may internally represent  $O_i$  either as a 4 by 4 matrix, or later as an RBT data-type

## skycam

- in our program we will have a frame called the sky-camera, which is separate from the drawn objects.

$$\vec{s}^t = \vec{w}^t S$$

- it will be our main camera
- camera will look down negative-z
  - visualize with first axis as right hand, second axis up, and third axis out the back of your head.
  - $S$  matrix called `skyPose` in code

## eye coordinate system

- to create picture, we need a point of view.
- position of each object in picture is based on its relationship to eye
  - its coordinates relative to the eye
- so we have the eye frame

$$\vec{e}^t = \vec{w}^t E$$

- any “object’s” frame can serve as the eye’s frame. (dog cam)
- but we mostly use the skycam
  - called `eyePose` in code

## at the end of the day

- our “renderer” cannot compute with points only a coordinate vectors
- so our convention will be to always give the renderer coordinate vector wrt to eye frame.
  - terminology (object, world, eye) coordinates.

$$\begin{aligned}\tilde{p} &= \vec{o}^t \mathbf{c}_o \\ &= \vec{w}^t O \mathbf{c}_o \\ &= \vec{e}^t E^{-1} O \mathbf{c}_o \\ &= \vec{e}^t E^{-1} \mathbf{c}_w \\ &= \vec{e}^t \mathbf{c}_e\end{aligned}$$

## openGL concepts

- verts will be described by homogeneous coordinates
  - `vector4` or `vector3`.
- before drawing each object, we will pass a matrix to the vertex shader to transform from the object to eye coordinates.
  - for each object it will be  $E^{-1}O_i$
  - called modelview matrix MVM
- vertex shader will multiply its coordinates by this matrix.
- also must transform the normal data that is associated with the vertex.
  - this uses the normal matrix (more soon)
- for camera perspective, we must multiply by a magic projection matrix
  - much more later
- pass all of this data out from the vertex shader
- `gl_Position` is used to place the triangle on the screen’s pallette.
- rest is the same as `asst1`
  - dots in trianglies, varying variables, pixels shaders
  - but there is also depth data used for visibility
- lets look at the code

## moving a frame

- to move a frame (object or skycam)  $\vec{o}_i^t \Rightarrow \vec{o}'_i^t$ 
  - this is implemented by updating the matrix  $O_i \Rightarrow O'_i$
- the sky cam has a coordinate frame  $\vec{s}^t$  just like an object.
- so we can update this by updating the matrix  $S$ 
  - remember that if you translate the eye (whether object or skycam) to the right, everything in the picture will appear to translate left!!!

## use of auxiliary

- we will usually think of some intuitive motion (a rotation or translation) to apply to the frame
  - we will represent this with some matrix  $Q$
- we will apply  $Q$  to the frame wrt some auxiliary coordinate system

- other frame is related to world through

$$\vec{a}^t = \vec{w}^t A$$

- recall the transformed coordinate system can then be expressed as

$$\begin{aligned} & \vec{o}_i^t \\ &= \vec{w}^t \mathbf{O}_i \\ &= \vec{a}^t \mathbf{A}^{-1} \mathbf{O}_i \\ &\Rightarrow \vec{a}^t \mathbf{Q} \mathbf{A}^{-1} \mathbf{O}_i \\ &= \vec{w}^t \mathbf{A} \mathbf{Q} \mathbf{A}^{-1} \mathbf{O}_i \\ &= \vec{w}^t \mathbf{O}'_i \end{aligned}$$

- so  $O'_i = \mathbf{A} \mathbf{Q} \mathbf{A}^{-1} \mathbf{O}_i$
- this will be put into the code by you.

**what are natural choices of aux c.s. for moving the frame that is being used as the eye**

- first person moving
  - car, or airplane, or robo-cam
- natural to use the viewer coordinate system itself  $\vec{e}^t$
- if i rotate wrt  $\vec{e}^t$ , object will translate and rotate in expected directions
  - $y$  direction is upwards in picture
  - $x$  direction is rightwards
  - $z$  direction is closer
- what happens when we plug this into the general rule?

**what are natural choices of aux c.s. for moving an object we are looking at**

- could chose  $\vec{o}^t$  itself as the aux.
- object will rotate about itself
- but directions won't correspond to the viewer's directions.

**special auxiliary**

- need to construct a special auxiliary coordinates frame
- with origin at center of the object
- but with directions lining up with those of the eye.
- we will assume that the eye here is the skycam
- to help construct this we need to recall our RBT factorization
- recall that we can decompose rigid tfrom  $O_i$  into rotation and translation

$$\vec{o}_i^t = \vec{w}^t T_i R_i$$

- recall that we can decompose rigid tfrom  $S$  into rotation and translation

$$\vec{s}^t = \vec{w}^t T_s R_s$$

- YOU will use this to create auxiliary frame

– recall how to read order of transformations

- we will call this “cube-sky”

### looking down at the world

- i may want to rotate the skycam around the world.
- so could chose  $\vec{w}^t$  as the auxiliary cf.
- but now “rightward” is wrt the objects x direction, not the cameras.
- can use similar (but simpler to construct) auxiliary frame
- we will call this “world-sky”

### transforming normals

- we use normals for shading
- how do they transform
- suppose squash in the  $y$  direction
  - normal gets higher in the  $y$  direction
- suppose i rotate forward
  - normal gets rotated forward
- what is the rule?

### computing normals

- tangent is vector between two nearby points
- normal is orthogonal to tangent

$$\vec{n} \cdot (\vec{p}_1 - \vec{p}_0) = \vec{n} \cdot \vec{dp} = 0$$

- in coordinates

$$\begin{bmatrix} nx & ny & nz & 0 \end{bmatrix} \left( \begin{bmatrix} x_1 \\ y_1 \\ z_1 \\ 1 \end{bmatrix} - \begin{bmatrix} x_0 \\ y_0 \\ z_0 \\ 1 \end{bmatrix} \right) = 0$$

- suppose i plug in  $A^{-1}A$ , then

$$\begin{aligned} & \left( \begin{bmatrix} nx & ny & nz & 0 \end{bmatrix} \mathbf{A}^{-1} \right) \left( \mathbf{A} \left( \begin{bmatrix} x_1 \\ y_1 \\ z_1 \\ 1 \end{bmatrix} - \begin{bmatrix} x_0 \\ y_0 \\ z_0 \\ 1 \end{bmatrix} \right) \right) = 0 \\ & \begin{bmatrix} nx' & ny' & nz' & * \end{bmatrix} \left( \begin{bmatrix} x'_1 \\ y'_1 \\ z'_1 \\ 1 \end{bmatrix} - \begin{bmatrix} x'_0 \\ y'_0 \\ z'_0 \\ 1 \end{bmatrix} \right) = 0 \end{aligned}$$

- note i dont care about the \*, since it will be multiplied by 0
  - so i don’t care about the fourth column of  $A^{-1}$ .
- and  $[nx', ny', nz', 0]$  must be the normal of the tformed geometry
- note since i am multiplying with  $[nx, ny, nz, 0]$  on the left, i don’t care about the fourth row of  $A^{-1}$ .
- let me toss out fourth row and col of  $A$ , to obtain the linear part  $a$ .

- let me transpose the whole thing

$$\begin{bmatrix} nx' \\ ny' \\ nz' \end{bmatrix} = \mathbf{a}^{-t} \begin{bmatrix} nx \\ ny \\ nz \end{bmatrix}$$

- so inverse transpose is the rule
- so lets look at the code