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

- (rhon) world frame $\vec{w}^t$
- never changes
- coordinates of a point wrt this frame are called world coordinates.

\[
\begin{bmatrix}
  x_w \\
y_w \\
z_w \\
1
\end{bmatrix}
\]

object frame

- we wish to describe the geometry of an object without thinking about its placement in the world.
- we associate a rhon frame with the object $\vec{o}^t$
- we describe our geometry using coordinates wrt $\vec{o}^t$.
  - called object coordinates

\[
\begin{bmatrix}
  x_o \\
y_o \\
z_o \\
1
\end{bmatrix}
\]

- example: canonical cube

object and world relationship

- relationship between world and object is expressed as

\[
\vec{o}^t = \vec{w}^t O
\]

where $O$ is a RB matrix.
- with the above understanding, in the computer program we store only $O$
- we can place and move the object by changing $\vec{o}^t$.
  - we update $\vec{o}^t$ by updating $O$
- in general we will have many objects, each with its own associated matrix
- in our code, we will store these matrices in $g\_objectRbt[i]$

eye frame

- 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’s frame
- so we have an eye frame
• think of this frame as x=right arm, y=up, z=forward

• eye coordinates:

\[
\begin{bmatrix}
x_e \\
y_e \\
z_e \\
1
\end{bmatrix}
\]

• relationship between world and eye is expressed as

\[ \vec{e}^t = \vec{w}^t E \]

where \( E \) is a RB matrix.

• any frame could act as the eye. in the code, we will have a special frame named g.skyRbt which will be the default eye.

to render

• a point can be expressed with object coords, world coords, and eye coords.

\[ \vec{p} = \vec{o}^t c = \vec{w}^t O c = \vec{e}^t E^{-1} O c \]

• it makes sense for our renderer to use eye coordinates.

• computed as

\[
\begin{bmatrix}
x_e \\
y_e \\
z_e \\
1
\end{bmatrix} = E^{-1} O
\begin{bmatrix}
x_o \\
y_o \\
z_o \\
1
\end{bmatrix}
\]

• in our code we will store object coords in the VBO, pass \( E^{-1} O \) to the vertex shader, as a uniform variable and do this multiplication in the vertex shader.

moving an object wrt \( \vec{a}^t \)

• we move an object by transforming \( \vec{o}^t \)

• Let us say we wish to apply some transformation matrix \( M \), (say translate in first axis) to an object frame \( \vec{o}^t \) with respect to some frame \( \vec{a}^t = \vec{w}^t A \)

\[
\begin{align*}
\vec{o}^t &= \vec{w}^t O \\
&= \vec{a}^t A^{-1} O \\
&\Rightarrow \vec{a}^t MA^{-1} O \\
&= \vec{w}^t AMA^{-1} O
\end{align*}
\]

• this is implemented by updating a variable \( O \).

• in code : \( O \leftarrow AMA^{-1} O. \)

non useful \( \vec{a}^t \)

• suppose we use \( \vec{o}^t \) as the auxiliary frame

• this of course simplifies to

\[
\begin{align*}
\vec{o}^t &= \vec{w}^t O \\
&\Rightarrow \vec{w}^t O M
\end{align*}
\]
• problem: directions don’t match what I see on the screen so hard to control (see demo)
• suppose we use \( \vec{e}' \) as the auxiliary frame
• object orbits around the eye (see demo).
• suppose we use \( \vec{w}' \) as the auxiliary frame
• this of course simplifies to
  \[
  \vec{a}' = \vec{w}' O \\
  \Rightarrow \vec{w}' M O
  \]
• but now we have two problems

useful \( \vec{a}' \)
• we want a new frame that has the origin of the object but directions of the eye.
  – see fig
• let us factor our matrices as
  \[
  O = (O)_T (O)_R \\
  E = (E)_T (E)_R
  \]
• desired auxiliary frame should be
  \[
  \vec{a}' = \vec{w}' (O)_T (E)_R
  \]
  – read right to left
• so \( A = (O)_T (E)_R \).
• in the spec, if our object is a cube and the eye the “sky camera” we will call this auxiliary; frame “cube-sky”

moving the eye
• eye frame (sky camera) can be moved just like an object frame \( E \leftarrow A MA^{-1} E \).
• Note: to get “intuitive” directions, we may need to negate some of the signs defining \( M \).
• for translations, we want to use the eye’s directions in the auxiliary frame (origin is irrelevant). So \( A = E \). Done.
• for rotations, roughly speaking, we might want to orbit the camera around some point in the scene (eg. the world’s origin) or rotate the camera about its own center (ego motion).
  – but the details for the aux frame are about to get trickier.

roll
• there are 3 dofs for rotations.
• this can allow for “roll”, which is often undesireable in a camera. (show demo)
• this results from allowing all possible orientations with a fixed z-vector.
• but we tend to keep our heads “level” with gravity.
  – this assumes some notion of an up-vector, for which we will use \( \vec{w}_y \).

roll free
• an eye frame is roll free if \( \vec{e}' = \vec{w}' E \) where \( E \) can be factored into the form \( E = TR_y R_x \). for some translation \( T \) and for amount of \( y \) and \( x \) angles.
• reading left to right: first walking, then turning your body and then bending at the waist.

**global/local solution**

• if you start with \( R_yR_x \) and multiplying new y rotations on the left, and new x rotations on the right, then factorizability will be maintained

• so starting with \( \vec{e} = \vec{w} E = \vec{w} T R_y R_x \), we will maintain roll freedom if:
  
  - we apply y-rotation matrices to \( \vec{e} \) wrt \( \vec{w} T \)
  - we apply x-rotation matrices wrt \( \vec{e} \)

• fact: the directions of a transformed frames do not depend on the origin of the wrt-frame.
  
  - so roll freedom does not depend on the origin of the wrt-frame.

• so starting with \( \vec{e} = \vec{w} E = \vec{w} T R_y R_x \), we will maintain roll freedom if:
  
  - we apply y-rotation matrices to \( \vec{e} \) wrt any frame with the worlds directions.
  - we apply x-rotation matrices wrt any frame with the eye’s directions.

**still need to pick the origin for the aux frame**

• for ego motion, aux-origin is at the eye’s origin: we do y-rotation with respect to \( A = (E)^T(I) R \) (sky-world), and we do x-rotations with repsect to \( A = (E)^T(E) R = E \) (sky-sky)

• for orbiting, aux-origin is world origin: we do y-rotation with respect to \( A = (I)^T(I) R = I \) (world-world), and we do x-rotations with repsect to \( A = (I)^T(E) R \) (world-sky)

• so there will be some cases to deal with in your code.

• if we want to do both an x and y rotation, we just apply the above rules sequentially.

**notes**

• lr mouse motion is no longer a local y rotation.

• 1) when you are looking mostly straight y-rotation motion mostly spins the image

• this is simply an artifact of restricting to roll-free orientations.

• 2) also if you are looking past straight up or down, y-rotation will move in the opposite of expected direction.

• both issues can be ameliorated by limiting the range of the x rotation amount. (easy to do with an Euler rep. but harder with our more general matrix representation).
  
  - we will ignore this issue.

**maps interface**

• a roll free orientation space and orbit mode can be useful for top-ish down viewing of a terrain map, esp with two finger touch (eg. ios maps).

• we think of u/d touch motion as a rotation along an axis parallel to the eye’s x axis, with the origin on the ground in front of me.

• we think of two finger twisting as doing a rotation along an axis parallel to the world y axis, with the origin on the ground.

• in this case we keep the user from looking along the horizon

• bonus: since two fingers gives us 4 constraints, we can simultaneously control this y-rotational dof along with 3 translational dofs.

• we can control the amounts to keep the two fingers “glued” to two positions on the map.

**other modalities for eye**

• there may be other useful ways to control the eye depending on the context.
  
  - walking, driving, flying.

• in a hierarchical scene graph, we can also lock the camera relative to a scene object (over the shouder view).