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 \( \vec{w}^f \)
- 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 \( \vec{o}^f \)
- we describe our geometry using coordinates wrt \( \vec{o}^f \).
  - 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
  \[
  \vec{o}^f = \vec{w}^f O
  \]
  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 \( \vec{o}^f \).
  - we update \( \vec{o}^f \) 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

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

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.

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

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 a

we move an object by transforming $\vec{o}^t$

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

$$
\vec{o}^t
= \vec{w}^t O
= \vec{a}^t A^{-1} O
\Rightarrow \vec{a}^t MA^{-1} O
= \vec{w}^t AMA^{-1} O
$$

this is implemented by updating a variable $O$.

in code : $O \leftarrow AMA^{-1} O$.

- we will do this as $O = doMtoOwrtA(M, O, A)$.

non useful $\vec{a}^t$

suppose we use $\vec{o}^t$ as the auxiliary frame

this of course simplifies to

$$
\vec{o}^t
= \vec{w}^t O
\Rightarrow \vec{w}^t OM
$$
• problem: directions don’t match what I see on the screen so hard to control (see demo)
• suppose we use $\overset{e}{\ell}$ as the auxiliary frame
• object orbits around the eye (see demo).
• suppose we use $\overset{w}{\ell}$ as the auxiliary frame
• this of course simplifies to

$$\overset{\ell}{\ell} = \overset{w}{\ell} O$$
$$\Rightarrow \overset{w}{\ell} M O$$

• but now we have two problems

useful $\overset{\ell}{\ell}$

• 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

$$\overset{\ell}{\ell} = \overset{w}{\ell} T (E) R$$

  – 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 $E \leftarrow A M A^{-1} E$.
• 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.
• 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).

roll

• there are 3 dofs for rotations.
• this can allow for “roll”, which is often undesirable in a camera. (show demo)
  – we tend to keep our heads “level” with gravity.
  – this requires a notion of an up-vector, for which we will use $\overset{w}{y}$.
• this is an artifact of allowing all possible x/y rotations wrt a frame with directions from the eye vector.
• to formally define roll, orthogonally project $\overset{w}{y}$ onto the plane spanned by $e^z_x$ and $e^z_y$. Look at the angle between this projected vector and $e^z_y$.
  – not well defined when looking straight up or down.
• the space of roll-free orientations is 2 dimensional.
correcting for roll

- we can explicitly compute roll of a current or updated eye frame, and then rotate $\mathbf{e}'$ around the $z$-axis, with respect to $\mathbf{e}'$.

lookat

- there is a closed form expression for taking the camera position, camera z-direction, and an “up vector”, and producing the “lookat” matrix $L$ and then defining $\mathbf{e}' = \mathbf{w}'L$
- con: doesn’t fit nicely into our paradigm of incrementally updating a frame using simple $x$ and $y$ rotations.

avoiding roll with Euler

- We could always require that $\mathbf{e}' = \mathbf{w}'TR_yR_x$ for some translation $T$ and for amount of $y$ and $x$ angles.
  - think of first walking, then turning your body and then bending at the waist.
- we would move camera by changing $T$ and these stored $y$ and $x$ angles.
- pro: this will avoid all roll, even later on when we are interpolating camera frames
- con: we will need to use a different representation for representing the sky-camera frame in our program. This can be an annoying representation to manipulate.

global/local trick

- we can get the above effect by only allowing $x$ and $y$ rotations. multiplying the $y$ rotations on the left, and $x$ rotations on the right.
  - different $y$ rotations will commute with each other, as will $x$ rotations.
- in our language, we will do this by using special auxiliary frames
- for orbiting, we always do $y$-rotation with respect to $A = I = (I)_{T}(I)_{R}$ (world-world), and we do $x$-rotations with respect to $A = (I)_{T}(E)_{R}$ (world-sky)
- for ego motion, we always do $y$-rotation with respect to $A = (E)_{T}(I)_{R}$ (sky-world), and we do $x$-rotations with respect to $A = E = (E)_{T}(E)_{R}$ (sky-sky)
- 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 shoulder view).

later

• scales
• hierarchy and scene graph.