Authoring Tool

Authoring Tool Overview

- **The class will write a tool that allows us to**
  - Instantiate objects from one of 3 different Python classes.
    - ✓ Each class will represent one of the following 3-D shapes:
      - 2-D picture
      - Sphere
      - Cube
  - Situate instantiated objects in 3-D space with positions and orientations we choose
  - Map images onto the sphere's surface and onto the cube faces
  - Generate a 360 degree equirectangular stereopair for viewing on the Oculus Go
- **This will be a 2 week lab**
  - You will work in 3 teams. Each team will turn in a separate report. All team members will receive the same grade

Team Assignments

- **Team: Cube Object**
  - Adolphe: Team Lead
  - David
  - John
- **Team: Sphere Object**
  - Nancy: Team Lead
  - Felicien
  - Nchima
- **Team: Main Program**
  - Nebiyou: Team Lead
  - Cesaire
  - Felix
  - Aimbale

Team Roles

- **Team Lead:**
  - Coordinate the scheduling of meetings
  - Facilitate a discussion about the software architecture
  - Facilitate the team's joint effort to:
    - ✓ Develop a schedule
    - ✓ Ensure each team member knows what he should do
  - Make sure a testing plan is included as part of effort
    - ✓ Will special test tools need to be developed?
  - Make sure the effort is divided fairly among team members
- **Team member:**
  - Actively participate in all meetings
  - Contribute ideas
  - Complete your deliverables on time and with high quality
  - Write agreed upon sections of the report

Overall Schedule

- **Monday Oct 8: Midterm**
- **Wednesday Oct 10: Detailed plan is due**
  - Schedule
  - Architectural overview of the code your team will write
  - Individual responsibilities of each team member (a table)
  - This plan must be submitted to me by this date as a pdf...not a text email. I expect something nicely written and nicely formatted that has clearly been given substantial thought. This should not be something thrown together in half an hour! This will be an important part of your final grade. It needs to convince me that you have a believable plan in place to succeed
    - ✓ All team members should write some portion of the plan.
    - ✓ Next to each plan section, list the name of the team member who wrote it
  - Plans of each team are shared with the others in pdf form
- **Mon Oct 15: Teams Exchange Tested Software**

Overall Schedule cont.

- **Friday Oct 19: Final Report Due**
  - One report per team
  - Mandatory NEW sections to add to your report
    - ✓ The detailed plan that was turned in on Oct 10, including the table showing responsibilities of individual team members
    - ✓ Theory of program operation: A description of how the entire program works (not just the portion your team wrote. You'll have the other teams description of their code).
    - ✓ Testing procedure: How was your final program tested?
  - You should include the following results:
    - ✓ Picture one: three objects, one of each type, no objects are occluded
    - ✓ Picture two: same three objects at same translations with a different rotation of each object
    - ✓ Picture three: three objects, one object partially occludes a second object, the third object is not occluded
    - ✓ Picture four: three objects. Object one partly occludes objects two and three. Object two partly occludes object three
    - ✓ Picture five: same as picture four but the objects should have different orientations

Building Blocks

- **I will provide**
  - A sample object for you to use
    - ✓ 2-D picture
  - Routines for doing change of coordinates
  - Sample of two pictures where one occludes the other

Authoring Tool Background

Notation

- **We will use *b* to denote a point physically attached to a body**
- **We will use *w* to denote a point that is fixed in the world**
- **We will use the prime symbol, ’ , to mean coordinates w.r.t the body system**
  - Example: ***b***’ is the coordinates of a point attached to a body in the body coordinate system. ***b*** is the coordinates of the same point in the world coordinate system
    - ✓ As the object moves and rotates, ***b***’ does not change but ***b*** does!
  - Example: ***w***’ is the coordinates of a world point in the body coordinate system. ***w*** is the coordinates of the same point in the world coordinate system
    - ✓ As the object moves and rotates ***w***’ changes but ***w*** does not

Reminder: Elemental Rotation Matrices

- **The rotation setup**
  - The coordinates of the point ***p*** are known in the world coordinate sytem
  - The point ***p*** will be rotrated around a world elemental axis
  - The elemental axes are stationary during the rotation
  - The elemental rotation matrices give the coordinates of the point ***p*** in the world coordinate system after rotation around the indicated world axis

$$R_x(\theta) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\theta) & -\sin(\theta) \\ 0 & \sin(\theta) & \cos(\theta) \end{bmatrix} \quad R_y(\phi) = \begin{bmatrix} \cos(\phi) & 0 & \sin(\phi) \\ 0 & 1 & 0 \\ -\sin(\phi) & 0 & \cos(\phi) \end{bmatrix} \quad R_z(\psi) = \begin{bmatrix} \cos(\psi) & -\sin(\psi) & 0 \\ \sin(\psi) & \cos(\psi) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Key Formula: world to body

- **The rotation matrix, $R_{composite}$ (shorthand R_c), is given by**

$$R_c = R_y(\phi)R_x(\theta)R_z(\psi)$$

  - The body and world coordinate systems are initially aligned
  - The angles are intrinsic Tait-Bryan angles for the body (yaw, pitch, roll)
  - The R_y, R_x, *and* R_z matrices are elemental world rotations

- **Converting world, $\underline{w}$, to body coordinates, $\underline{w}'$ (here $\underline{t}_b$ is the origin of the body coordinate system in world coordinates):**

$$\underline{\mathbf{w}}' = R_c^T \underline{\mathbf{w}} - R_c^T \underline{\mathbf{t}}_b$$

Key Formula: body to world

- **To convert body to world, i.e., *b*' to *b*, we use**

$$\underline{\mathbf{b}} = R_c \underline{\mathbf{b}}' + \underline{\mathbf{t}}_b$$

This is derived by solving the world to body equation on the previous slide for the world coordinates. (Reminder: $\underline{\boldsymbol{t}}_b$ is the origin of the body coordinate system in world coordinates; R_c is the rotation matrix describing the body's orientation in terms of its intrinsic Tait-Bryan angles)

Key Formula: body to world (homogenous coordinates)

- **To convert body coordinates to world coordinates, i.e., *b*' to *b*, in homogenous coordinates we use**

$$\begin{bmatrix} \underline{\mathbf{b}} \\ 1 \end{bmatrix} = \begin{bmatrix} R_c & \underline{\mathbf{t}}_b \\ \underline{\mathbf{0}}^T & 1 \end{bmatrix} \begin{bmatrix} \underline{\mathbf{b}}' \\ 1 \end{bmatrix}$$

Key Formula: world to body (homogenous coordinates)

- **To convert world coordinates to body coordinates, i.e., *w* to *w'*, in homogenous coordinates we use**

$$\begin{bmatrix} \underline{\mathbf{w}}' \\ 1 \end{bmatrix} = \begin{bmatrix} R_c^T & -R_c^T \underline{\mathbf{t}}_b \\ \underline{\mathbf{0}}^T & 1 \end{bmatrix} \begin{bmatrix} \underline{\mathbf{w}} \\ 1 \end{bmatrix}$$

Key Formula: Projection from Camera coordinates to PICS (homogenous coordinates)

- **In this case, the coordinates of the point being projected are assumed to be in the camera's coordinate system. This equation is in homogenous coordidnates!**

$$\begin{bmatrix} \zeta_1 \\ \zeta_2 \\ \zeta_3 \end{bmatrix} = \begin{bmatrix} \eta_1 f & 0 & c_1 & 0 \\ 0 & \eta_2 f & c_2 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} X_1 \\ X_2 \\ X_3 \\ 1 \end{bmatrix}$$

  - X_1, X_2, X_3 are a point in the camera's coordinate system
  - (c_1, c_2) are the image center in PICS (unit: pixels)
  - f is focal length, ζ is pixel density

Included for completeness but not needed for this lab

To find Coordinates when Camera is Displaced from World's Origin

- **Follow this procedure. WE WILL USE HOMOGENOUS COORDINATES!**
  - Compute a body point's coordinates in world coordinates
  - Convert the world coordinates to camera coordinates
  - Convert to polar coordinates
  - Fill in the right "latitude / longitude" in the output picture
- **To convert world to camera coordinates, remember that *the camera is now a displaced body*. It has been moved to a new spot in the world. Therefore: use the world to body conversion formula**
  - Camera's origin is at $\underline{\boldsymbol{t}}_{cam}$ (in world coordinates) and its orientation is R_{cam} (using the camera's intrinsic Tait-Bryan angles)

Pulling it all together

- **Define the following homogenous matrices (i.e. for use with homogenous coordinates)**
  - H_{bw} describes the body-to-world conversion
  - H_{wc} descirbes the world-to-camera conversion
  - P_{cam} defines the world to camera projection
    - ✓ Identity matrix for this lab
- **The overall conversion from body coordinates to camera image coordinates (PICS) is given by**

$$H_{\text{composite}} = P_{\text{cam}} H_{\text{wc}} H_{\text{bw}}$$

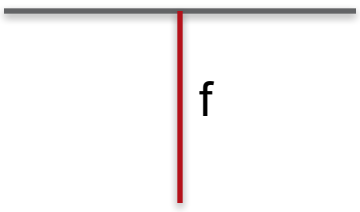
Homogenous coordinates make this whole operation linear. We can multiply the matrices! Sweet!

Better for the Oculus Go Gallery app

- **The gallery app wants input images that are in equirectangular format**
- **We can therefore skip the camera projection step**
  - We set the projection matrix equal to the identity matrix
- **Instead, once the body's coordinates are known in the camera coordinate system, we convert to polar coordinates and just fill in the right latitude/longitude pixel.**

Lab Setup

- **We will use the Go's 360 degree stereopair format**
  - Final image will be of size 4096 x 4096
  - Left and right images will both be 4096 x 2048 (cols x rows)
  - We will measure all distances in units of the focal length (f =1)



You will likely see funny effects if you situate objects at the poles

Lab Setup cont.

- **We will place the left camera slightly to the left of the world origin, but with NO rotation**
  - $\underline{t}_l$ = (-0.25, 0, 0)
- **We will place the right camera slightly to the right of the world origin, but with NO rotation**
  - $\underline{t}_r$ = (0.25, 0, 0)

Implementation Strategy

- **We will write Python objects that create 3-D objects**
  - Sphere
  - Cube
  - Picture
- **Each object will output a list of world coordinates and an RGB value associated with that world coordinate**
- **The main program could theoretically instantiate as many objects as desired**
  - It must handle 3 for this lab

Implementation Strategy cont.

- **VRpicture():**
  - Arguments:
    - ✓ $\underline{\boldsymbol{t}}$ (its center in world coordinates)
    - ✓ $[\phi, \theta, \psi]$ (a numpy array of Tait-Bryan angles)
    - ✓ 1 bmp image (cols x rows in size)
    - ✓ Col edge length (in units of the focal length)
      - – Row edge length will be scaled to preserve aspect ratio
  - Implementation
    - ✓ Build a “body-to-world” transformation matrix
    - ✓ Step through each pixel of the image
    - ✓ Compute the x, y, z coordinates corresponding to that pixel.
    - ✓ Apply the “body-to-world” transformation
    - ✓ Add result to the list

Implementation Strategy cont.

- **VRsphere():**
  - Arguments:
    - ✓ $\underline{\boldsymbol{t}}$ (its center in world coordinates)
    - ✓ $[\phi, \theta, \psi]$ (a numpy array of Tait-Bryan angles)
    - ✓ image.bmp (A 2R columns x R rows equirectangular projection to map onto the sphere)
    - ✓ radius (in units of the focal length)
  - Implementation
    - ✓ Build a “body-to-world” transformation matrix
    - ✓ Step through each pixel of the equirectangular map
    - ✓ Compute the latitude and longitude of the current pixel (the radius is an input). Convert from polar to 3-D rectangular coordinates.
    - ✓ Apply the “body-to-world” transformation
    - ✓ Add result to the list

Implementation Strategy cont.

- **VRcube():**
  - Arguments:
    - ✓ $\underline{\boldsymbol{t}}$ (its center in world coordinates)
    - ✓ $[\phi, \theta, \psi]$ (a numpy array of Tait-Bryan angles)
    - ✓ 6 square bmp images of the same size
    - ✓ Edge length (in units of the focal length)
  - Implementation
    - ✓ Build a “body-to-world” transformation matrix
    - ✓ Instantiate 6 2D-picture objects, one for each face, and situate them in space where they are needed
    - ✓ Apply the “body-to-world” transformation matrix to each face
    - ✓ Add results to the list

Implementation Strategy cont.

- **Tait-Bryan angles format**
  - A numpy array of size 3
- **The position vector, *t***
  - A numpy array of size 3
- **Return list format**
  - It will be a numpy array, np.float32, with as many rows as there are points to project. Each row will have 7 entries:
    (x, y, z, 1, R, G, B)
    - ✓ float32 is needed since we need to handle fractional (x, y, z) coordinates. This is overkill for RGB but we aren't trying to be as memory efficient as possible.
    - ✓ The "1" is because we're using homogenous coordinates

Implementation Strategy cont.

- **Main Program**
  - Create 4096 x 4096 black output image
  - Define parameters of the desired objects
    - ✓ You may need to experiment with object sizes and locations to see what gives good results
  - Instantiate the desired objects
  - Compute the needed $H_{composite}$ matrices for the left and right camera
  - For each object
    - ✓ Obtain its world coordinate list
    - ✓ Loop through the list, convert to polar coordinates, and fill in the left/right camera image locations in the equirectangular output image
    - ✓ Must track object distance associated with each equirectangular image pixel and only update a pixel for a point that is closer than the current one
      - this implements occlusion
      - can be done with a floating point image initialized to -1 and updated as necessary
  - Optional: post-process image to “fill in gaps”
    - ✓ Because we aren’t using the “inverse method” there may be black pixels, <u>especially if you place objects near the poles</u>