Presentation Notes

CSCI-5229: Computer Graphics, Fall 2018 **James Watson, 105754866**
University of Colorado Boulder **Due: 2018-12-13**

1. Demo
   (a) Purpose
   The purpose of this project is to display real 3D camera data as textured meshes. The depth image from a 3D camera provides a wealth of information, but is fuzzy. We can apply filtering and processing to the noisy point cloud, but this results in the loss of color data. The goal is use the filtered point cloud and the original flat image data to reconstruct a representation of 3D objects with small, manageable point clouds that still retains rich and informative visual information.
   (b) Features
   - Display filtered point clouds as textured meshes
   - Display the robot pose at which the 3D data was captured
   - Select 3D objects with an intuitive mouse interface

   (c) DEMO
   (d) Process
   i. Data Reading & Processing
   For every "shot", the robot system caches the following data
   - Camera frame with respect to the lab frame
   - Robot joint angles
   - Filtered point clouds in the lab frame
   - Infrared image (False color)

   ii. Mesh Construction & UV-Matching
   A. Transform the coordinates into the camera frame
   B. Triangulate into mesh (covered later)
   C. Match each of the points to a (U, V) location on the IR image
   D. Construct a bounding box for each of the point cloud clusters generated by the system

   iii. Event Loop
   A. Listen for kb and mouse input
   B. Process input
   C. If click: Mouse collision detection
   D. `display()`

2. Cover cool graphics things that you did in your project that I did not talk about in class

- Processing mouse clicks in the lab frame
  - There are four reference frames to contend with: {Window [pixels] , Cursor [m] , Camera [m] , Lab [m]}

  (a) Transform mouse coordinates in screen pixels to relative window coordinates in the range $[-1, 1]$
  (b) Transform window coordinates into cursor coordinates (orthogonal), Draw crosshairs (Turn depth test OFF)
  (c) Transform window coordinates info camera frame coordinates. This must be a right-handed coordinate system with `+z` pointed down the lens of the camera into the scene
  (d) Construct a point at arbitrary `+z` depth from the origin. (x, y) are computed using the vertical FOV angle and the current aspect ratio of then screen.
  (e) Construct a ray (eye $\rightarrow$ point) and transform into the lab frame
  (f) Now you are ready for collision detection!

- Hierarchical Collision Detection
  (a) Intersection of AABB and ray
  (b) Triangle-wise collision detection
  For each triangle
  i. Construct a plane from triangle and its normal

ii. Get the intersection point between the ray and the plane

iii. Compute winding number (point-in-polygon)
This works for any closed polygon, polygon can self-cross

A. New 2D reference frame with the query point at $(0, 0)$

B. Follow segments CW around the polygon and count $+\boldsymbol{x}$ crossings

C. Result can be any integer: positive number is CW circuits around point, negative is CCW

D. For any non-zero result, the point is encircled by (inside of) polygon!

- Nested Coordinate Frames
  - You can `glPushMatrix` up to `GL_MAX_MODELVIEW_STACK_DEPTH`. This varies between systems but OpenGL specifies this be at least 32.
  - Paired `glPushMatrix` & `glPopMatrix` make it *relatively* easy to render a robot character using the standard DenavitHartenberg parameters. DH parameters is a system that specifies serial coordinate transforms across the rigid links of a robot. I can give by robot character a list of these parameters and each link will apply the appropriate `glTranslate` & `glRotate` in order to display the correct relative poses between links.

3. Cover stuff related to your project that has nothing to do with Computer Graphics

- Delaunay Triangulation, courtesy of Paul Bourke
  - Paul Bourke is a big name in computational geometry, and he posts a lot code online with helpful diagrams
  - A common method to connect a collection of points in triangular mesh
  - It is the dual of Voronoi Cells
  - Gotcha: Paul Bourke does not delete allocated memory

4. Cover "gotchas" - things that did not work initially, how you figured out the problem, and how you fix it

(a) Difference between screen coordinates, orthogonal (flat) coordinates, and model (projection) coordinates

(b) SDL2 function calls are all different from SDL1: Read the docs *carefully*

(c) There are two ways to handle keyboard input in SDL2, Keyboard State & Event Queue

- Keyboard State: Pro - You won't miss interaction events that begin and end before the next frame , Con - You have to reconstruct kb combination from the series of events
- Event Queue: Pro - Easily discern key combinations , Con - Only detect state at the instant you checked it
- Mine was a hacky combination of both

(d) SDL2 seems to detect random kb events. Solution: Check to be sure that the event that happened is a key-down event.

(e) Eigen Library

- Advantages
  i. Header-only, no installation required
  ii. Cross, Dot, Norm, Squared Norm, Invert, Transpose, SVD, Anything you like
  iii. Declaration without size
  iv. Clean up after themselves
- Gotchas, There are rules you must follow in order to accommodate memory magic
  i. Pass by reference only
  ii. `EIGEN_MAKE_ALIGNED_OPERATOR_NEW` at the beginning of struct/class `public` declaration
  iii. Never store in a `std::vector`. ← It is much, much easier to store as rows in a matrix. You can assign a row to a vector type
  iv. Never feed the matrices after midnight

(f) Make absolutely sure to call `glDisable( GL_TEXTURE_2D )` when you are done drawing with textures, otherwise the color leaks out into the entire state machine. Ew! Blue!