Computer Science 175

Introduction to Computer Graphics
www.fas.harvard.edu/~lib175

Time: m/w 2:30-4:00 pm
Place: MD g125
Section times: tba

Instructor: Steven “shlomo” Gortler
www.cs.harvard.edu/~sjg
sjg@cs.harvard.edu
Office: MD 243

teaching assistants:
Jialiang Wang
jialiangwang@g.harvard.edu
textbook

  - Not required, but is a good mirror of the course.

- web has resources on OpenGL/GLSL syntax
prerequisites

- ability to program in C (multi-hundred line program)
  - we will use C++, but will help get you up to speed

- some familiarity with linear algebra
warnings

- you will need to use computers that support programmable shaders
- most of the assignments will build off each other
- much of the openGL gobbledygook will be given as a black box.
- I expect to interact in class with the students, so I expect students to be engaged. (no laptops, smartphones,...) (fig)
- I don’t capitalize or spell well.
graded work

- 7 programming assignments and two written assignments
  - one weekers
  - due (mostly) at mondays at 11:59pm
- unit per program
- 1 final project
  - 1.5 units
- participation
  - 1 unit or so
late policy

• programs due on midnight
• you have 5 free days to use as you wish
• up to one day late: -15%
• up to two days late: -30%
• after two days late: 0%
• we will optimize late penalties/free days at the end.
Standards

• you can work/submit in pairs if you wish
  – you can change this arrangement for each asst.

• you should not look at code from other groups

• you should not discuss written assts with other groups

• we will use Piazza
platforms for class

• windows
  – visual studio
    * you can get a free “community edition” from the web

• mac

• linux (less support from us).

• must have recent generation graphics card that support vertex and fragment shaders
  – first assignment will get this worked out
lets look at the assignments

• 1: hello world openGL,
• 2: written: transforms
• 3: 3d viewer
• 4: arcball
• 5: written: projection
• 6: robots and picking
• 7: linear keyframe animator
• 8: smooth keyframe animator
• 9: fur
• 10: final project
applications:

- entertainment
  - video games
  - special effects
  - animation
  - virtual world interactions

- computer aided design
  - car bodies
  - architectural environments

- simulation
  - flight simulators
  - surgical simulators

- 3D visualization
  - medical scan imagery
  - fluid flow
syllabus

• we understand how to work on top of OpenGL
• we will understand what is happening inside of OpenGL

topics

• getting started with OpenGL
• coordinate systems and transforms
• quaternions and arcball
• camera projections
• rasterization process
• scene graphs
• color theory
• interpolation and splines for key frame animation
• overview of more advanced computer animation
• overview of more advanced geometric modeling
• subdivision surfaces
• image reconstruction and filtering (dots)
• shading and shadows
• ray tracing (just a bit)
pipeline demo

• basic representation of object starts with a collection of triangles

• but we will have a very simple scene, but this is very general (cat).

• each triangle is described by 3 vertices

• each vertex is described by \((x,y,z)\) coordinates

• other information is attached to each vertex
  – color, normal vector, squirrel pointer

• each triangle is processed in turn

• each vertex is processed in turn
  – positioned in image

• all pixels within the triangle determined

• are such pixels colored using the data from the vertices
  – “blend” step is involved here

• a “merge” step for a triangle’s pixel is run to determine if its color updates the image or not.
lets follow the real time pipeline

• basic representation of object starts with a collection of triangles (fig)
• each triangle is described by 3 vertices
• each vertex is described by \((x,y,z)\) coordinates
• other information is attached to each vertex
  – color, normal vector
• this data is called *attribute* data.
• this data is passed (once) to OpenGL and stored in a *vertex buffer*
• later we can make an OpenGL *draw* call
vertex processing

- after a draw call
- each vertex (set of attributes) is passed through a **vertex shader** that you write and load into openGL.
- shader also has access to **uniform variables** that you set
- typically does “geometric” transformations on the vertex coordinates
- to place the objects correctly with respect to each other and the eye
- to get perspective effect
- output position as gl_Position as well as user defined **varying variables**
fixed function

- 3 vertices for each triangle are collected by the assembler
- triangles vertices are positioned within the window
- triangles are rasterized
  - which pixel are inside the triangle
  - varying variable data (color, normal, texture coordinates) are appropriately interpolated for each pixel
pixel processing

• information for each dot is sent through a fragment shader that you write and load into openGL
  – figures out final output color of the pixel from the interpolated data

• outputs a color for the screen

• typically simulates material reflection (demo)
  – how did i get flat vs smooth

• also can look up data in a texture image (fig)
  – pointers into the texture are done using varying variables called texture coordinates
  – adds visual complexity
merging

- decide whether to write into framebuffer
- looks at data already there
- main thing: depth test. (recall the demo)
- also alpha blending (transparency)