What is Computer Graphics?

- **Pictures** generated by computer
  - these slides
  - magazines, web, movies, games, ...
- **Tools** used to make them
  - Hardware
    - display
    - input
  - Software
    - libraries / APIs (OpenGL, ...)
    - design packages (Blender, ...)
- **Field of study** involving
  - Programming
  - Mathematics
  - Algorithms
  - Esthetics, art

Course contents

- 13 topics
  - Graphics programming, display HW
  - Vectors, modeling transformations
  - Projections
  - Light, eye, displays
  - Shading, VRML
  - Texture mapping, Compositing
  - Curves and surfaces
  - Rasterization, Hidden surfaces
  - Animation
  - Modeling real objects
  - Image-based rendering
  - Procedural modeling
  - Ray tracing, Radiosity, anti-aliasing

- Projects
  - 2 programming assignments
    - basic OpenGL programs in Python, basic Blender
    - must be done alone
    - modeling, scene graphs can be done in pairs
  - 3D animation
    - using Blender (3D design SW, free)
    - can be done in pairs
    - the second and third homework may still change...

Graphics programming

- Outline:
  - the programming language: Python
  - the graphics library: OpenGL
  - the window library: GLUT
  - Hello Python & OpenGL
  - the 3D graphics pipeline as a concept
  - 5 generations of graphics hardware
  - homework #1
Python

- Scripting language
  - great for rapid prototyping
  - good also for large projects
  - also supported by Blender
- Fully object oriented
- Interpreted
  - (well, compiles to bytecode and executes it, like Java)
- Easy to extend with C/C++
- Available on almost every platform
  - Unix, Windows, MacOS, Beos, Symbian OS, ...
  - The same code runs on all those platforms
- Free

Graphics APIs

- An application programmer’s interface (API) provides an interface between the application code and the hardware
- Most popular graphics APIs (OpenGL, DirectX, Java3D, PHIGS, GKS-3D) are based on the synthetic camera model
- Have functions to specify (and simulate):
  - objects
  - viewer
  - light sources
  - material properties

OpenGL

- A low-level graphics API
  - “immediate mode” rendering to a frame buffer
- Order of things:
  - commands or display list
  - setting up the state
    - transforms, colors, etc.
  - per vertex operations & primitive assembly
    - apply transforms, shading, to vertices
    - create triangles/lines/points
  - rasterization
    - from primitives to pixels
  - per pixel operations
    - blending, texturing, etc.
  - frame buffer
- Window system independent
  - no facilities for window events or user input
  - need to use additional libraries

GLUT

- OpenGL Utility Toolkit
- Provides windowing facilities
  - hides the specifics of underlying system such as
    - Win32
    - Motif
    - ...
  - provides basic tools for event-driven programming
- Too restricted for large applications
  - should use some real general purpose GUI library
  - wxWindows (and its Python port wxPython) is a good candidate
- But good for smaller programs and for studying OpenGL
New APIs for mobile phones

- OpenGL ES
  - A subset of OpenGL
  - The C/C++ API for, e.g., Symbian OS

- M3G
  - Mobile 3D Graphics for Java MIDP
  - Also known as JSR-184
  - Builds on top of OpenGL ES
  - Adds scene graph, animation, binary file format

Event driven programming

- Program responds to events
- Events are handled by user-defined callback functions
- Callbacks must know context and event type
  - passed through variables

A skeleton of an event driven GLUT program in Python

```python
from OpenGL.GL import *
from OpenGL.GLU import *
from OpenGL.GLUT import *

def main():
    # initialize things
    glutInit([])
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB)
    glutInitWindowSize(300, 300)
    glutInitWindowPosition(100, 200)
    glutCreateWindow('Hello GLUT!')

    # register callbacks
    glutDisplayFunc (myDisplay)
    glutReshapeFunc (myReshape)
    glutMouseFunc   (myMouse)
    glutMotionFunc  (myMotion)
    glutKeyboardFunc(myKeyboard)

    # maybe initialize more things
    glutMainLoop()

main()
```

Opening a window for drawing

```python
from OpenGL.GL import *
from OpenGL.GLU import *
from OpenGL.GLUT import *

def main():
    # initialize things
    glutInit([])
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB)
    glutInitWindowSize(300, 300)
    glutInitWindowPosition(100, 200)
    glutCreateWindow('Hello GLUT!')

    # register callbacks
    glutDisplayFunc (myDisplay)
    glutReshapeFunc (myReshape)
    glutMouseFunc   (myMouse)
    glutMotionFunc  (myMotion)
    glutKeyboardFunc(myKeyboard)

    # initialize and go
    myInit()
    glutMainLoop()

main()
```
A bit more Python basics

def fibonacci(n):
    # no type declarations, just assign to a variable
    fib = [1,1]  # here a list of two ints
    # assign to i the elements of a list one at a time
    # range() is a built-in function, returns a list
    for i in range(2,n):
        # grow fib by concatenating to it a list of one
        fib = fib + [ fib[i-2] + fib[i-1] ]
    # return the list
    return fib

# loop over the list of fibonacci numbers
for i in fibonacci(10):
    # print one at a time, no newline
    print i,

1 1 2 3 5 8 13 21 34 55

>>> range(5)
[0, 1, 2, 3, 4]

>>> range(2,5)
[2, 3, 4]

Basics of OpenGL rendering

- Basic steps
  - specify coordinate system
  - clear the window
  - specify colors
  - specify vertices
  - draw primitives
  - force completion

Specifying a view in 2D

- How do you specify a view of a 2D picture?

  - Most graphics systems let you specify:
    - the part of a picture to display (the window)
    - the place to display that picture on the screen (the viewport)
  - The default window in OpenGL is [-1,1]x[-1,1]
  - The default viewport in OpenGL is the full window
### Setting up a 2D view in OpenGL

- Set up the viewport
  - `glViewport(left, bottom, width, height)`

- First set the matrix mode and initialize:
  - `glMatrixMode(GL_PROJECTION)`
  - `glLoadIdentity()`

- Set up an orthographic projection
  - `gluOrtho2D(left, right, bottom, top)`

- A good practice is to leave matrix mode to modelview
  - `glMatrixMode(GL_MODELVIEW)`

### Clearing the buffers

- OpenGL has several buffers that must be cleared at first
  - color buffer(s), for drawing the image
  - depth buffer, for determining what is visible and what isn’t
  - accumulation buffer, for motion blur, anti-aliasing, …
  - stencil buffer, to restrict drawing to certain portions of the screen

- First you tell the value to which a buffer is cleared

- For each frame, you tell which buffers should be cleared
  - for now, only color buffer is needed

### Clearing the buffers

- Clear values
  - `glClearColor(r, g, b, a)`
  - `glClearDepth(depth)`
  - `glClearAccum(r, g, b, a)`
  - `glClearStencil(s)`

- Perform the clear
  - `glClear(mask)`
  - mask
    - `GL_COLOR_BUFFER_BIT`
    - `GL_DEPTH_BUFFER_BIT`
    - `GL_ACCUM_BUFFER_BIT`
    - `GL_STENCIL_BUFFER_BIT`

- if you need to clear several buffers at once, you can bitwise OR masks: \( m_1 \lor m_2 \lor … \)

### Specifying primitives: the idea

- Everything is made up of vertices

- Vertices have properties:
  - Location / position
  - Color (or material)
  - Normal vector
  - Texture coordinates

- If you don’t specify a property, it uses the “current” value of that property
  - Either a default value, or one that was last specified

- Then you use some of the vertices to form primitives
  - Points, lines, triangles, triangle strips, …
Specifying colors

- With OpenGL, the description of the shape of an object is independent of the description of its color.
- A geometric object is drawn using the current color.
- Example:
  - Draws objects A and B in red.
  - Object C in blue.
  - The command that sets the current color to green is wasted.

Setting color:
- `glColor3f(r, g, b)`

Example:
- `setColor(0.0, 0.0, 0.0)` # black
- `setColor(1.0, 1.0, 1.0)` # white
- `setColor(1.0, 0.0, 0.0)` # red
- `setColor(0.0, 1.0, 0.0)` # green
- `setColor(0.0, 0.0, 1.0)` # blue
- `setColor(1.0, 1.0, 0.0)` # yellow
- `setColor(1.0, 0.0, 1.0)` # magenta
- `setColor(0.0, 1.0, 1.0)` # cyan

Specifying points

- The “old” way (dropped in OpenGL ES):
  - `glVertex3f(x, y, z)`
  - 2D, 3D, 4D versions
  - Four type variants: {d|f|i|b|ub}
  - Surrounded by `glBegin()` and `glEnd()`.

- Vertex arrays:
  - Introduced for OpenGL 1.1, included in OpenGL ES.
  - Define vertex data in arrays.
  - Much fewer function calls than the old way => faster.

- Color arrays:
  - For per-vertex color.

- Enable arrays:
  - `glEnableClientState(GL_VERTEX_ARRAY)`
  - `glEnableClientState(GL_COLOR_ARRAY)`

Specify array data

- Vertex array in Python:
  - `glVertexPointerf([[0.0, 0.0, 0.0],
                      [5.0, 0.0, 0.0],
                      [5.0, 5.0, 0.0],
                      [0.0, 5.0, 0.0]])`
  - Takes a single argument, a 2D matrix (list of vertices).
  - Vertices can be 2D, 3D, or 4D.

- Similar for color:
  - `glColorPointerf([[0.0, 0.0, 0.0],
                      [1.0, 0.0, 0.0],
                      [1.0, 1.0, 0.0],
                      [0.0, 1.0, 0.0]])`

OpenGL primitives

- Not in OpenGL ES (can make these from triangles)
Specifying primitives: Arrays

- Draw primitives using only vertex data
  - `glDrawElementsui( mode, indices[] )`
  - `mode` is one of `GL_POINTS`, `GL_LINES`, ...
  - `indices` index to vertex array
- Draw primitives using all enabled arrays
  - `glDrawArrays( mode, first, count )`
  - constructs a sequence of geometric primitives using array elements starting at `first` and ending at `first+count-1` of each enabled array

Begin-End way

- `glVertex3f( x, y, z )`
- variants: 2i, 3d, 4s, ...
- `glVertex3fv( v )`
  - give a vector rather than elements
- Draw a simple square
  - `glBegin( GL_POLYGON )
  glVertex3f(-.5, .5, .5)
  glVertex3f( .5, .5, .5)
  glVertex3f( .5,-.5, .5)
  glVertex3f(-.5,-.5, .5)
  glEnd()`
- `vs = ( (-.5, .5,.5), ( .5, .5,.5),
  ( .5,-.5,.5), (-.5,-.5,.5) )`
  - `glBegin( GL_POLYGON )`
  - `for v in vs: glVertex3fv( v )`
  - `glEnd()`

Add colors

- `glBegin(GL_POLYGON)
  glColor3f (1, 0, 0)
  glVertex3f(-.5, .5, .5)
  glColor3f (0, 1, 0)
  glVertex3f( .5,-.5, .5)
  glEnd()`

Hints for polygonizing surfaces

- Keep polygon orientations consistent
  - all clockwise or all counterclockwise
  - important for polygon culling and two-sided lighting
- Watch out for any nontriangular polygons
  - three vertices of a triangle are always on a plane; any polygon with four or more vertices might not
- There's a trade-off between the display speed and the image quality
  - fewer polygons render quickly but might have a jagged appearance; millions of tiny polygons probably look good but might take a long time to render
  - use large polygons where the surface is relatively flat, and small polygons in regions of high curvature
- Try to avoid T-intersections in your models
  - there's no guarantee that the line segments AB and BC lie on exactly the same pixels as the segment AC
  - this can cause cracks to appear in the surface
Some line and point properties

- You can specify some "2D" or "image space" rendering attributes
- Sizes (default = 1)
  - `glPointSize(size)`
  - `glLineWidth(width)`
- Smoothing on (default = OFF)
  - `glEnable(GL_POINT_SMOOTH)`
  - `glEnable(GL_LINE_SMOOTH)`
- Smoothing off
  - `glDisable(GL_POINT_SMOOTH)`
  - `glDisable(GL_LINE_SMOOTH)`

Force completion

- Scenarios
  - pipelining: CPU process triangles, sends commands to special hardware. Waiting for completion before continuing is inefficient.
  - networking: server collects several graphics calls to packets before sending to client. Sending individual calls would be inefficient.
- Force execution
  - `glFlush()` forces previously issued OpenGL commands to begin execution, thus guaranteeing that they complete in finite time.
  - `glFinish()` forces all previous OpenGL commands to complete, returns after all effects from previous commands are fully realized.

The 3D synthetic camera model

- The synthetic camera model is a paradigm for creating images of 3D geometry
- It involves two components, specified independently:
  - objects (a.k.a. geometry)
  - viewer (a.k.a. camera)

Imaging with the synthetic camera

- The image is rendered onto an image plane or projection plane usually in front of the camera
- Projectors emanate from the center of projection (COP) at the center of the lens or pinhole
- The image of an object point \( P \) is at the intersection of the projector through \( P \) image plane COP
Specifying a viewer

- Camera specification requires four kinds of parameters:
  - **Position**: the COP
  - **Orientation**: rotations about axes with origin at the COP
  - **Focal length**: determines the size of the image on the film plane, or the field of view
  - **Film (image) plane**: its width and height, and possibly orientation

Specifying lights and materials

- Light sources usually defined by:
  - location
  - strength
  - color
  - directionality
- Materials usually defined by:
  - various shading parameters such as ambient, diffuse, and specular reflectance
  - texture maps for variations of color and surface perturbations

Clipping

- We think of the image plane as having a finite (rectangular) extent
- Objects are clipped to a clipping rectangle or clipping window

The geometric pipeline

- Many commercial graphics workstations use a pipeline architecture, implemented in hardware, for processing geometry
- Works well because:
  - lots of data that is processed similarly
  - well-decomposed computation
- Q: What’s the downside of large pipelines?
  - Imbalance. If one of the stages takes longer to execute than the others, the other stages are idle, and resources are wasted. Another downside is increased latency.
The graphics pipeline

- The pipeline metaphor can be extended to encompass just about everything we do in 3D graphics:
  - animation
  - modeling
  - transformation
  - clipping
  - lighting and shading
  - hidden surface
  - projecting
  - rasterizing
  - compositing
  - post-processing

First generation of gfx HW - Wireframe

- Vertex transform, clip, project
- Rasterization color interpolation (points, lines)
- Fragment overwrite
- Dates prior to 1987

From Real-Time Graphics Architectures course by K.Akeley and P.Hanrahan
http://graphics.stanford.edu/courses/cs448a-01-fall/

Second generation – Shaded solids

- Vertex lighting calculation
- Rasterization depth interpolation (triangles)
- Fragment depth buffer, color blending
- Dates 1982 - 1992

Third generation – Texture mapping

- Vertex Texture coordinate transformation
- Rasterization Texture coordinate interpolation
- Fragment Texture evaluation, antialiasing
- Dates 1992 - 2000
Fourth generation - Programmability

- Programmable shading
- Image-based rendering
- Convergence of graphics and media processing
- Curved surfaces

Fifth generation – Global evaluation

- Ray tracing: visibility and integration
- True shadows, path tracing, photon mapping

Homework #1 (a)

- Write a program that demonstrates
  - the drawing modes GL_POINTS, GL_LINES, GL_LINE_STRIP, GL_TRIANGLES, GL_TRIANGLE_STRIP, GL_TRIANGLE_FAN
  - colors, line and point properties (try them all)
  - draw a picture of your pet (real or imaginary) and sign it by writing your own first name (by actually drawing the letters)
    - don’t use Begin-End, but the array style (in all these homeworks)!
    - see bgdipper.py in the examples
  - make sure that resizing of the window scales the picture so it fully stays in the window
    - bgdipper.py scales correctly for horizontal but not for vertical resizes
    - take a look at Red Book example 4-1, it is also in the PyOpenGL redbook demos

Homework #1 (b)

- Make an animated clock
  - You have to define 3 primitives
    - disk, centered box, cornered box
    - use each of them for the hour, minute, and second hands
  - Create your placing routine
    - scales, rotates, translates
    - use that to place, size, and orient the hands, etc.
  - Also draw the numbers, text.py file shows how to plot text with glut
    - add also day of the week and month (TUE 25) somewhere on the clock, see documentation of datetime
  - File clock_start.py has a frame work ready for it, just fill the gaps

- Here’s an example picture, be creative
  - colors
  - shapes (the clock could be square, with or without rounded corners, or oval)
Homework #1 (c)

- Write a simple polyline editor that allows the user to enter and edit pictures made of polylines.
- Include at least the following commands:
  - Left button, control down (if glutGetModifiers() & GLUT_ACTIVE_CTRL) introduces a new vertex, that will be connected with edges to all the vertices that are currently selected. Afterwards, only the new vertex is selected.
  - If you drag with the left mouse button down, you will move all the selected vertices (and of course the adjacent edges).
  - The right mouse button toggles the selection state of the closest vertex.
  - 'd' deletes all the vertices currently selected (and of course all the edges that are part of it).
  - 'a' deselects all vertices if any where selected, selects all if none were selected.
  - 'e' toggles all the potential edges between selected vertices (if there is an edge between two selected vertices, delete it, if there isn't, create it).
  - exit with ESC (see Python23/Lib/site-packages/OpenGL/Demo/redbook/fog.py)

Homework #1 (d) Blender

- Do the quickstart tutorial (4th bullet at the class web Blender tutorials) or at http://www.blender3d.org/_media/education/quickstart/Blender2Manual_Quickstart.zip
- The UI has changed in 2.3x, but you can still do the tutorial pretty easily
  - The texture output color was a bit difficult to find (section 2.7), these pics should help

Blender, cont.

- Then do the coffee cup tutorial (5th tutorial on class page, originally from Blender 1.5 tutorial)
- Your homework
  - Combine the tutorials by adding a table, and a coffee cup on the table to the room with the moving sphere in the first tutorial
  - Make sure the result looks different from that of others by playing with colors, shapes, maybe even camera angles

About homeworks

- These homeworks (and report) take easily 10-20 hours of solid work to complete
  - you need to set up Python, etc.,
  - need to go through several Python and Blender tutorials
  - learn a new programming language and 3D design software
- So start early!
  - every year students are surprised that the last evening or even weekend wasn’t enough
- Grading is 50% from homework
  - get every point you can now, exam is more difficult
  - only rarely did a person who completely missed 1 of the 3 homeworks finish the course (and then with a grade of 1 or so)
- If you can’t finish a homework
  - don’t despair!
  - use the course mailing list to ask others
  - complete the report and tell what you did for partial credit
- Extra bells and whistles may get you extra points!
  - so be creative
Recommended reading

- OpenGL: the red and the blue book
  - both are available on-line, even downloadable, follow the links on the course OpenGL web pages
  - PyOpenGL distribution contains some red book examples in the Demo directory (under Lib/site-packages/OpenGL)
- Python and Blender learning material
  - lots of links on the course web pages

Some Python tips

- `dir(obj)`
  - lists the contents of a module or attributes of an object
  - `import sys
dir(sys)`
  - `dir('')`

- Many modules and functions have a documentation string `__doc__`
  - `print sys.__doc__`
  - `print ''.join.__doc__`

- Print out the help strings for an object, module, ...
  - `def myhelp(ob):
    print ob.__doc__
    for m in dir(ob):
      try:
        print "%-15s %s" % (getattr(ob,m).__doc__)
      except:
        print "%-15s %s" % (m, "No docstring found.")
    print "\n====================================\n"`