Lecture I
INTRODUCTION TO VISUALIZATION

Visualization is an area that is intimately related to Computer Graphics and Geometric Modeling. In this course, we are also interested in special issues of Visualization arising from (1) very large data sets and (2) very narrow and unreliable bandwidths (called **thinwires**). Wireless connections and internet access typify thinwires.

§1. Course Information

The course home page is at URL `http://cs.nyu.edu/ yap/classes/visual`. OPERATING SYSTEM: The official programming environment is Unix/Linux. If you work at home on your PC, and you want easy access to your Windows world, we strongly recommend the free *Cygwin* system that sits on top of Windows.

PROGRAMMING LANGUAGE: Programming for computer graphics will require C/C++ using the *OpenGL API* and also Java programming, especially *Java2D* API. The official C/C++ compiler for this course is GNU's gcc/g++ compiler.

OBTAINING SOFTWARE: If you are using your own computers, you should be able to obtain free copies of all the software we need: gcc/g++, openGL including the GLUT library. Although OpenGL is not free on unix environments, there are free software such as Mesa which can be used. In the second half of the course, we will need to use Java (get Java 1.3). Our course webpage will give you some helpful leads, but let us know of other possibilities.

TEAMS: This is a hands-on, projects-oriented course. We allow you to do your homework and projects in teams. You must register teams with me for approval. I recommend teams of 1-3. The composition of the teams can freely change throughout the semester.

PREREQUISITES: Fundamental Algorithms, Java programming and C/C++. Knowledge of computer graphics, geometric modeling and networking are not considered prerequisites.

GRADING: 25% homework, 15% class participation, 60% final project.

MAILING LIST: Please sign up for the class mailing list immediately. See instructions in the class webpage.

TEXT BOOK: We require F.S. Hill's *Computer Graphics Using Open GL* (second edition, 2001), from Prentice-Hall. For Java graphics and web-based issues, we use *Core WEB Programming* (Second Edition), by M. Hall and L. Brown, from Sun Microsystem Press (Prentice Hall), 2001. Also strongly recommended for Java2D is *JAVA 2D Graphics*, by J. Knudsen from O'Reilly, 1999. An alternative, but less useful for our course is *Graphic Java* (Third Edition), by D. M. Geary, Sun Microsystem Press (Prentice Hall), 1999.

§2. Introduction to Visualization

Visualization is a basic tool used in many disciplines to help researchers understand their data. The fact that our visual system is one of the most highly developed faculties may partly explain why converting data to visual form is frequently the best route to understanding. We need to visualize a variety of datasets: images, maps, fluid flows (e.g., weather patterns), volumetric data (e.g., brain tissues), geometric models (e.g., aeroplane or engine), large graphs (e.g., internet sites), or just abstract tabular data. A major issue arise from the fact that these datasets can be very large. Further, if we want to do visualization on the internet, this is compounded with severe bandwidth limitations (we call this “thinwire visualization”).

Is Visualization just Computer Graphics? Visualization is closely intertwined with two other subjects: Computer Graphics and Geometric Modeling. Sometimes it is hard to tell them apart, but there are some basic distinction in their respective goals.

(1) Geometric Modeling is about how to represent geometric models in computers. The computational and mathematical properties of the representation is of inherent interest. Such goals are intimately connected to CAD/CAM applications. For instance, Geometric Modeling is interested in studying different classes of surface patches for their flexibility and power in expressing certain properties that modelers need.

(2) Computer Graphics is about how to render geometric models on the screen or other output device. Photorealism and speef are two major rendering goals. For instance, this calls for the study of light models.

(3) Visualization is about understanding models and data through their graphical representation. To illustrate the difference between computer graphics and visualization, suppose a user has a complex geometric model that needs to be “understood”. This is a somewhat vague goal that depends on what the user is trying to accomplish: an engineer’s goal may be to understand how to manufacture this object; a designer’s goal may be to understand the shapes and their functioning in the overall design; a user may wish to check out its manueverability. Photorealistic rendering of the model is only one possible approach to understanding it. But for a model with complex internal details, photorealism may not be appropriate. Instead, we want the ability to see through the object (with X-ray eyes).

Geometric modeling focus on the modeling elements (polyhedral or surfaces) and the synthesis of models (methods for putting together the modeling elements). But usually (1) is not very useful if there is no way to display or visualize the model. But sometimes we can query a model and get an answer without any display. E.g., we might have a map data model and we can perform queries and get a non-visual answer.

Usually (2) assumes the model is given and focus on the issue of transforming this into images on the screen. This is called rendering, and is the central issue of computer graphics. Graphics has also gotten into an area that is somewhat between the rendering issue and the ”modeling” part. That is, how do you model natural images of clouds, waves, mountain terrains, forests and fire? A central goal of rendering is realism.

Finally, (3) is about understanding, and so we view the realism goal of computer graphics as only one way towards understanding an object. But if non-realistic images are produced to help understanding, this is not a bad thing. For instance, we are quite happy to display a house in which part of the walls are removed, or walls are made transparent, so that we can look inside. This is surely unreal. The original data need not even be geometric. For instance, there is great interest in visualizing graphs (the field called Graph Drawing arose around this interest in the 1990s).

What about the web? Sure, it is the biggest social phenomenon of the last decade. But what has it to do with visualization? For us, it highlights the main issues of thinwire communication. Its popular sobriquets include “world wide wait” or “waiting without whining”. Wireless communication is another example of thinwire, and so it is also a suitable object for our interest. Conventional visualization techniques fail in such settings.

Let us take one element in thinwire visualization that distinguishes it from conventional visualization: the computational model. Conventional visualization has the standard CPU, main memory and secondary memory model of computation. When large models are to be visualized, it is assumed to be stored in secondary memory. The key issue how to organize the data so that the necessary parts of this model can be brought into main memory in a timely fashion. Two important classes of techniques here are database management and I/O efficient algorithms. In contrast, the computational model for thinwire visualization is the client-server model. The server is assumed to hold the model and the client is accessing this model over a network. The bandwidth is several orders of magnitude smaller than the above "I/O bottleneck". This bandwidth is also unreliable. So all the standard techniques of walk-through or fly-over applications breaks down when applied to thinwire visualization.

Datasets. Researchers have developed many techniques in recent years to address these problems. These techniques are highly dependent on the nature of the datasets. Graphic datasets are well known to be very diverse – in the trade, there are many books on graphic data formats. This proliferation of formats is indicative of the rich nature of graphic data. In this course, we will take a close look at some important types of formats or datasets.

In Summary, this course will be address the following issues of visualization:

- Thinwire visualization: techniques to achieve "responsive" visualization, including proxy approaches.
- Foveation: the human visual system is foveated – how can we exploit this? This is connected to wavelet analysis.
- Large data models: Techniques here include hierarchical (level of detail) modeling, I/O efficient algorithms, and database management issues.
- Important data sets: We study some important types of data: geometric models such as found in CAD/CAM models, vector and elevation datasets which arise in geographic information system (GIS), and volumetric data.

The technical basis for visualization includes computer graphics, computational geometry and algorithms. Our interest in thinwire visualization requires some networking background (such as TCP/IP protocol). We will provide this background as we go.

§3. Some Background

We review some mathematical background. We will be working mostly in Euclidean 2-space or 3-space. The set of real numbers is denoted $\mathbb{R}$, and Euclidean d-space is the set $\mathbb{R}^d$ ($d = 2, 3$, etc). A point in d-space is just a real vector of length d. There is also also a projective version of these spaces, denoted $P^d(\mathbb{R})$. Each point in projective d-space is represented by a **non-zero** real vector

$$v = (x_0, x_1, \ldots, x_d)$$

of length $d + 1$. These vectors are considered equivalent under multiplication by non-zero constants: cv and v are equivalent for $c \neq 0$. We sometimes write $[x_0, \ldots, x_d]$ for the equivalence class of v.

Let $A \in \mathbb{R}^{d\times d}$ be a square matrix. We can view it as a transformation of d-space by matrix-vector multiplication: $v \mapsto A(v) = A.v$. Another transformation of space is translation by a fixed vector $t \in$

$\mathbb{R}^d$: $v \mapsto v + t$. More generally, given (A, t), we have the transformation $v \mapsto Av + t$. If A is non-singular, this transformation has an inverse, namely $(A^{-1}, -A^{-1}t)$. Such transformations are called **affine transformations**.

The basic fact about matrix multiplication is that it is associative: if A, B, C are d-square matrices, then

$$A(BC) = (AB)C.$$

In Euclidean space, multiplication by a matrix can represent any combination of the following: scaling, rotation, and more.

Projective Space and Homogeneous Coordinates. We can embed $\mathbb{R}^d$ in $P^d(\mathbb{R})$ by identifying $[1, x_1, \ldots, x_d]$ with the real point $(x_1, \ldots, x_d)$. Thus, the point x_0 is treated as the homogenization coordinate. If $x_0 = 0$, the corresponding point is said to be "at infinity", so the points of $\mathbb{R}^d$ (under this identification) corresponds to the "finite points". If $[x_0, \ldots, x_d]$ is the projective point corresponding to a point $p \in \mathbb{R}^d$, we call $x_0, \ldots, x_d$ a set of **homogeneous coordinates** for p.

Let us specialize the above discussion to 3-d. Thus, a point (x, y, z) will be represented in homogeneous coordinates as $(1, x, y, z)$. More gnerally, let (w, x, y, z) denote the homogeneous coordinates of points in projective 3-space.

Advantages of homogeneous coordinates is that an affine transformation, (A, t) can now be encoded as a single $d+1$-square matrix B where the first column of B is $(1, t)$ and rest of the first row is filled with zeros, and the remaining entries contain A. The transformation $v \mapsto Av + t$ becomes Bv' where $v' = (1, v)$.

E.g.,

$$A = \begin{bmatrix} a & b & c \\ a' & b' & c' \\ a'' & b'' & c'' \end{bmatrix}, \qquad t = \begin{bmatrix} x \\ y \\ z \end{bmatrix}, \qquad B = \begin{bmatrix} 1 & 0 & 0 & 0 \\ x & a & b & c \\ y & a' & b' & c' \\ z & a'' & b'' & c'' \end{bmatrix}, \tag{1}$$

See "More Math" below for further discussions.

See "Getting Started with OpenGL"

§4. ADDITIONAL NOTES

§5. MORE MATH

We endow this space with algebraic properties and call it an **affine d-space**. To recognize this additional structures on the set $\mathbb{R}^d$, we denote it as $A^d(\mathbb{R})$. The elements $\mathbb{R}^d$ are d-vectors, and they form a vector space over $\mathbb{R}$: this means that $\mathbb{R}^d$ is an additive group $(\mathbb{R}^d, +, \mathbf{0})$, and for each $c \in \mathbb{R}$, $v \in \mathbb{R}^d$, we define their scalar product $cv \in \mathbb{R}^d$ in the natural way (by componentwise multiplication). This scalar product distributes over real addition and scalar addition:

$$(c + c')(v + v') = cv + cv' + c'v + c'v'.$$

If $A \in \mathbb{R}^{d\times d}$ is a $d \times d$ matrix, we define the transformation $A : \mathbb{R}^d \to \mathbb{R}^d$ by $A(v) = A \cdot v$ (matrix-vector multiplication).

The projective version of this space, denoted $P^d(\mathbb{R})$. If we view $A^d(\mathbb{R})$ as a subset of $P^d(\mathbb{R})$, then we have the following interesting observation:

$$P^d(\mathbb{R}) = A^d(\mathbb{R}) \uplus P^{d-1}(\mathbb{R}).$$

For instance, $P^1 = A^1 \uplus A^0$ where A^0 is a single point. We can expand this recursive equation to get

$$P^d = A^d \uplus A^{d-1} \uplus \cdots \uplus A^1 \uplus A^0.$$

§6. Getting Started

Read Chapter 1 of Hill's book for a general introduction to computer graphics. Chapter 2 introduces graphics programming. The following is a general orientation and overview of the OpenGL API.

A modern graphics environment has a screen, which can display several "windows" simultaneously, each window having its own coordinate system. Graphics coordinate systems tend to have x-axis to the right, y-axis downwards. But Mathematical coordinate systems is always x-axis to the right, and y-axis upwards. OpenGL uses both convensions: it has a screen coordinate system which follows the graphics convention, but it also has a model space that follows the mathematical convention. This idea that there are different coordinate systems at work at the same time is important to grasp if we are not to get confused.

But OpenGL offers an API that is platform independent. We may expect that the API has methods to set the color of the background (e.g., **setBackground(white)**), set the color of the pen (e.g., **setColor(red)**), set a pixel color with pen color (e.g., **drawPixel(x,y)**), draw a line **drawLine(x,y,x',y')**, etc. The names and arguments of the actual OpenGL commands will be roughly the same as these hypothetical commands. But because of the said distinction between screen coordinates and model coordinates, what is (x, y) in these commands? We will need to delve into the OpenGL model for this.

What is (not) in OpenGL? The raw OpenGL API is powerful but lacks basic higher level concepts that would be painful to program from scratch.

Hence the need for using various predefined layers above OpenGL. The first of this is the "Graphics Utility Library" **GLU** and is included in all OpenGL implementations. For instance, it provides code for modeling common objects such as spheres, standard matrix transformations, polygon tesselations, and surface rendering.

The other critical library is "GL Utility Toolkit" **GLUT**. This provides the minimum functionality for a modern window graphical interface. Moreover, GLUT is independent of any platform (Linux/Unix, X-Windows or Microsoft Windoze). We shall generally be programming on top of GLUT.

Other common libraries are **glX** for X-windows, and **glAux** (GL Auxiliary Library). If you include

#include ¡GL/glut.h¿

in your program, then the libraries GL/gl.h, GL/glu.h will automatically be included.

Window-based programming. Why is this different from conventional programs? Because it is based on interaction between the application and the user. You cannot program the interaction sequence (since the user is unpredictable). Instead, you program "reactions" to "events". That is, you program the appropriate thing to do when a certain event occurs. Thus we say this is **event driven programming**. E.g., you program the "closing of a window" in response to the event of a mouse press when the mouse curser is over a "close window" icon.

"Events" are predefined of interest to the application. For each event, there is a **callback function** which is invoked when the event occurs. Certain arguments are automatically passed to the callback function. E.g., in mouse events, the callback function will receive the current (x, y) coordinates of the mouse cursor. Hence the callback function prototype has parameters ready to receive this (x, y) value. Let **myMouse** be such a callback function. To "register" this as the callback function for the mouse event, you use the statement:

glutMouseFunc(myMouse);

A mouse event occurs when a mouse button is pressed or released (or clicked?). The function **myMouse()** will be passed the particular button event, and the current position of the mouse cursor. If no callback function is registered for a particular event, then nothing happens on this event.

See a sample program in the file.

Display Lists Based on the following graphics architecture: a conventional CPU and a special-purpose computer called the **display processor**.

END OF LECTURE