Lecture 1

Computer graphics, broadly defined, is a set of methods for using computers to create and manipulate images. There are many applications of computer graphics including entertainment (games, cinema, tv commercials and other advertising), design (architectural, manufacturing, ...), visualization (medical, scientific, information), simulation (training), and much more.

This course will look at methods for creating images that are defined on a rectangular pixel grid. The word *pixel* stands for "picture element". The coordinates of the grid are integers. For example, typical high definition images are 1920×1080. The resolution of a typical projector such as this one is 800×600 (SVGA).

The color of a pixel is defined by three numbers called RGB. These are the intensities of the red, green, and blue components. Typical images have 8 bits for each RGB value and hence 24 bits of intensity information per pixel i.e. values from 0 to 255. Here are some examples of colors:

(0,0,0)	black
(255,255,255)	white
(255,0,0)	red
(0, 255, 0)	green
(0, 0, 255)	blue
(255,255,0)	yellow (red + green)
(255,0,255)	magenta (red + blue)
(0,255,255)	cyan (green + blue)

RGB space defines a cube: $\{0, \ldots, 255\} \times \{0, \ldots, 255\} \times \{0, \ldots, 255\}$. The above list of colors define the corners of the cube. There are many other colors ($256 \times 256 \times 256 = 2^{24}$) with exotic names. You can amuse yourselves by having a look here:
`http://www.w3schools.com/tags/ref_colornames.asp`

What this course <u>is</u>

This course is an introduction to the fundamentals of the field of computer graphics. These fundamentals have not changed much in the past few decades. The course I am offering you is similar to what Prof. Paul Kry[1] offers when he teaches this course and to what you'll find in other universities as well, with some variations that reflect the tastes of the individual instructors.

What this course <u>is not</u>

Although the fundamentals of computer graphics have not changed much in the past few decades, many aspects of computer graphics *have* changed. Graphics cards (GPUs) in particular have become very powerful and many software tools have become available for taking full advantage of this power. To work as a software developer in the computer graphics industry today, you would need to learn

[1]Paul has taught the course the past six years. He is away on sabbatical this year but will return to teach the course again in Fall 2015.

at least the basics of these tools. I want to warn you up front: while these modern software tools are built on the fundamentals that we will cover in the course, this course itself will not teach you these software tools: this is not a course about modern computer graphics programming and software development.

A bit of history

To appreciate the above point, it may be helpful to consider a few key developments of the past 20 years in computer graphics:

Up to the early 1990's, it was relatively cumbersome to write graphics software. The standards for graphics software were limited and typically one would code up an algorithm from scratch. Although many companies had developed specialized graphics hardware (GPU's – graphical processing units), this hardware was difficult to use since it often required assembly language programming. New algorithms were typically implemented to run on the CPU, rather than GPU, and as a result they tended to be slow and impractical for typical users.

One of the big graphics companies at that time (SGI) built and sold workstations that were specialized for graphics. SGI developed an API called GL which had many libraries of common graphics routines. In 1992 SGI released a standard API called *OpenGL* which other hardware vendors could use as well.

[ASIDE: Microsoft followed with the DirectX API shortly thereafter. `http://en.wikipedia.org/wiki/DirectX`. DirectX is used in the "X Box". DirectX competes with OpenGL and has remained popular among many users especially those who work with Microsoft software. I will only discuss OpenGL in the course but it is good to be aware of DirectX as well.]

OpenGL 1.0 is said to have a *fixed function pipeline*. "Fixed function" means that the library offers a limited fixed set of functions. You cannot go beyond that library. You cannot program the graphics hardware to do anything other than what OpenGL offers.

Graphics card technology improved steadily from the mid 1990's and beyond the emergence of industry leaders NVIDIA, ATI (now part of AMD) and Intel. However, to program a graphics card (GPU) to make the most use of its enormous processing potential, one still had to write code that was at the assembly language level. This was relatively difficult and not much fun for most people.

In 2004, OpenGL 2.0 was released which allowed access to the graphics card using a high level C-like language called the OpenGL Shading Language (GLSL). Other "shading languages" have been developed since then, and more recent versions of OpenGL have been released as well. I'll say more later in the course about what such shading languages are for.

The most recent trend is WebGL (released in 2011) which is an API specific for graphics in web browsers. Google maps uses WebGL for example. There is a growing trend to introduce WebGL in "intro to graphics" courses.

Course Outline

I next went of the official Course Outline. Please read it.
`http://www.cim.mcgill.ca/~langer/557/CourseOutline.pdf`.

A bit about me

It may be helpful for you to know a bit about me, what I'm interested in, and what is my relationship to the field of computer graphics. Here is a brief overview. At the end of this course, I'll say more about my research interests. I also offer a course that is more directly related to my interests: Computational Perception (COMP 546/598). I hope to offer it in 2015-2016.

I did my undergrad here in the early 1980s, majoring in Math and minoring in Computer Science. I did a MSc in Computer Science at University of Toronto. In my MSc thesis, I looked at how to efficiently code natural images using statistics. This topic is more in the field of Data Compression than Vision, but I was motivated by visual coding and the work led me further into vision science.

After my M.Sc., I returned to McGill in 1989 to start a PhD. There were no course requirements for a Ph.D. here in those days and my supervisor was in Electrical Engineering so that's the department I was as well (but I know very little about EE). My PhD was in computer vision, specifically, how to compute the shape of surfaces from smooth shading and shadows.

I received my PhD in 1994 then spent six years as a post-doctoral research in the late 1990s. It was only then that I started learning about graphics. I used graphics as part of my vision research, namely it was a way to automatially generate images that I could use for testing computer vision algorithms. I also began doing research in human vision (visual perception) and I used graphics to make images for my psychology experiments.

I returned to McGill in 2000 as a professor. I've been doing a mix of computer vision and human vision since then. I've taught this course COMP 557 four times over the years.

My research combines computer vision, human vision, and computer graphics. Most of the problems I work on addressed 3D vision, including depth perception and shape perception. I have worked on many types of vision problems including shading, motion, stereopsis, and defocus blur. I'll mention some of the problems I've worked on later in the course when the topics are more relevant.

Ok, enough about me, let's get started...

Review of some basic linear algebra: dot and cross products

We'll use of many basic tools from linear algebra so let's review some basics now.

- The **length of 3D vector u** is:

$$|\mathbf{u}| \equiv \sqrt{u_x^2 + u_y^2 + u_z^2}$$

- The **dot product** (or "scalar product") of two 3D vectors $\mathbf{u}$ and $\mathbf{v}$ is defined to be:

$$\mathbf{u} \cdot \mathbf{v} \equiv u_x v_x + u_y v_y + u_z v_z \ .$$

  A more geometric way to define the dot product is to let θ be the angle between these two vectors, namely within the 2D plane spanned by the vectors.[2] Then,

$$\mathbf{u} \cdot \mathbf{v} \equiv |\mathbf{u}| \ |\mathbf{v}| \cos(\theta).$$

[2] If the two vectors are in the same direction, then they span a line, not a plane.

It may not be obvious why these two definitions are equivalent, *in general*. The way I think of it is this. Take the case that the vectors already lie in a plane, say the xy plane. Suppose $\mathbf{u}$ = (1,0,0) and $\mathbf{v} = (\cos\theta, \sin\theta, 0)$. In this case, it is easy to see that the two definitions of dot product given above are equal. As an exercise: why does this equivalence holds in general?

It is important to realize that the dot product is just a matrix multiplication, namely $\mathbf{u}^T\mathbf{v}$, where $\mathbf{u}$ and $\mathbf{v}$ are 3×1 vectors. Thus for a fixed $\mathbf{u}$, the dot product $\mathbf{u} \cdot \mathbf{v}$ is a linear transformation of $\mathbf{v}$. Similarly for a fixed $\mathbf{v}$, the dot product $\mathbf{u} \cdot \mathbf{v}$ is a linear transformation of $\mathbf{u}$.

Another way to think about the dot product is to define $\hat{\mathbf{x}} \cdot \hat{\mathbf{x}} = \hat{\mathbf{y}} \cdot \hat{\mathbf{y}} = \hat{\mathbf{z}} \cdot \hat{\mathbf{z}} = 1$, and $\hat{\mathbf{x}} \cdot \hat{\mathbf{y}} = 0$, $\hat{\mathbf{x}} \cdot \hat{\mathbf{z}} = 0$, $\hat{\mathbf{y}} \cdot \hat{\mathbf{z}} = 0$ and to define the dot product to be linear, then we have

$$\mathbf{u} \cdot \mathbf{v} = (u_x\hat{\mathbf{x}} + u_y\hat{\mathbf{y}} + u_z\hat{\mathbf{z}}) \cdot (u_x\hat{\mathbf{x}} + u_y\hat{\mathbf{y}} + v_z\hat{\mathbf{z}}) \;=\; u_xv_x + u_yv_y + u_zv_z$$

where we expanded the dot product into 9 terms and used the fact that $\hat{\mathbf{x}} \cdot \hat{\mathbf{x}} = 1, \hat{\mathbf{x}} \cdot \hat{\mathbf{y}} = 0, ...$

- The **cross product** of two 3D vectors $\mathbf{u}$ and $\mathbf{v}$ is written $\mathbf{u} \times \mathbf{v}$. The cross product is a 3D vector that is perpendicular to both $\mathbf{u}$ and $\mathbf{v}$, and hence to the plane spanned by $\mathbf{u}$ and $\mathbf{v}$ in the case that these two vectors are not parallel.

  Given two vectors $\mathbf{u}$ and $\mathbf{v}$, the cross product $\mathbf{u} \times \mathbf{v}$ is usually defined by taking the 2D determinants of "co-factor" matrices:

$$\mathbf{u} \times \mathbf{v} = \begin{bmatrix} u_x \\ u_y \\ u_z \end{bmatrix} \times \begin{bmatrix} v_x \\ v_y \\ v_z \end{bmatrix} \equiv \begin{bmatrix} u_yv_z - u_zv_y \\ u_zv_x - u_xv_z \\ u_xv_y - u_yv_x \end{bmatrix}$$

  Sometimes in linear algebra courses, this definition seems to come out of nowhere. I prefer to think of the above, not as a "definition", but rather as a result of more basic definition, as follows. We *define* the cross product on the unit vectors $\hat{\mathbf{x}}, \hat{\mathbf{y}}, \hat{\mathbf{z}}$, namely

$$\hat{\mathbf{z}} = \hat{\mathbf{x}} \times \hat{\mathbf{y}}$$

$$\hat{\mathbf{x}} = \hat{\mathbf{y}} \times \hat{\mathbf{z}}$$

$$\hat{\mathbf{y}} = \hat{\mathbf{z}} \times \hat{\mathbf{x}}$$

  and we obtain six other equations by assuming for all $\mathbf{u}$ and $\mathbf{v}$,

$$\mathbf{u} \times \mathbf{v} = -\mathbf{v} \times \mathbf{u}$$

$$\mathbf{u} \times \mathbf{u} = \mathbf{0} \;,$$

  for example, $\hat{\mathbf{x}} \times \hat{\mathbf{x}} = 0$ and $\hat{\mathbf{y}} \times \mathbf{x} = -\hat{\mathbf{z}}$. We also assume that, for any $\mathbf{u}$, the cross product $\mathbf{u} \times \mathbf{v}$ is a linear transformation on $\mathbf{v}$. Writing

$$\mathbf{u} = u_x\hat{\mathbf{x}} + u_y\hat{\mathbf{y}} + u_z\hat{\mathbf{z}}$$

  and similarly for $\mathbf{v}$, we use the linearity to expand $\mathbf{u} \times \mathbf{v}$ into nine terms ($9 = 3^2$), only six of which are non-zero. If we group these terms, we get the mysterious "determinants of co-factor bla bla" formulas above.

Finally, verify from the above definition that, for any fixed 3D vector $\mathbf{u}$, there is a 3×3 matrix which transforms any other 3D vector $\mathbf{v}$ to the cross product $\mathbf{u} \times \mathbf{v}$, namely :

$$\mathbf{u} \times \mathbf{v} = \begin{bmatrix} 0 & -u_z & u_y \\ u_z & 0 & -u_x \\ -u_y & u_x & 0 \end{bmatrix} \begin{bmatrix} v_x \\ v_y \\ v_z \end{bmatrix}$$

Notice that this matrix is not invertible. It has rank 2, not 3. In particular, when $\mathbf{u} = \mathbf{v}$, we get $\mathbf{u} \times \mathbf{u} = \mathbf{0}$.

Computing the normal of a triangle, and its plane

A key application for the cross product is to compute a vector normal to a planar surface such as a triangle or more generally a polygon. Let's just deal with a triangle for now. Let the vertices be $\mathbf{p}_0, \mathbf{p}_1, \mathbf{p}_2$ and assume we are looking at the triangle from a direction so that these vertices go counter-clockwise. We will define the surface normal (perpendicular) such that it points into the half space containing us (the viewer).

Let $\mathbf{e}_0$ denote the vector $\mathbf{p}_1 - \mathbf{p}_0$, i.e. the vector from $\mathbf{p}_0$ to $\mathbf{p}_1$, and let $\mathbf{e}_1$ denote the vector $\mathbf{p}_2 - \mathbf{p}_1$. Then the *outward* surface normal (pointing toward the viewer) is defined

$$\mathbf{N} = \mathbf{e}_0 \times \mathbf{e}_1.$$

The vertex point $\mathbf{p}_0 = (p_{0,x}, p_{0,y}, p_{0,z})$ is in the plane containing the triangle. So the equation of the plane passing through this point can be parameterized by:

$$\mathbf{N} \cdot (x - p_{0,x}, y - p_{0,y}, z - p_{0,z}) = 0,$$

This equation just says that, for any point (x, y, z) on the plane, the vector from this point to $\mathbf{p}_0$ is perpendicular to the surface normal.