CS 314: Preliminaries, Image and Rasterization Basics

Robert Bridson

September 10, 2008

1 Preliminaries

The course web-page is at: `http://www.ugrad.cs.ubc.ca/~cs314`. Details on the instructor, teach assistants, lectures, office hours, labs, assignments and more will be available there.

Computer graphics is a topic defined by what is done, not how: loosely speaking, graphics encompasses everything involved in a computer producing visual output. Some people in graphics even blur the boundaries and work with other sensory output, such as sound or touch. To make things work, graphics can involve a multitude of other disciplines: hardware design (from circuits to robots and more), psychology, data structures, computational physics, geometry, photography, art history, machine learning, networking, computer vision, optics, scientific computing, anatomy, architecture, and much, much more. This course necessarily isn't going to delve deeply into any one aspect, and will ignore many, but will try to cover what many consider the core—or at least, a good representative slice—of the field: the **three-dimensional rendering pipeline**. That is, how to go from a somewhat abstract description of a three-dimensional scene to an image on the screen, preferable quickly, and preferably producing a beautiful, realistic image.

This pipeline is of critical importance to many applications you probably are already aware of: visual effects in films, 3D computer games, scientific visualization of experimental measurements. However, much of what we will study also plays an important foundational role in other graphics applications, such as computer-aided design (CAD), architecture, abstract information visualization, font rendering and other 2D operations, animation, image processing and more. There are two follow-on undergraduate courses offered at UBC that get deeper into graphics topics, offered in alternate years: 424 (offered next academic year) on geometric modeling, and 426 (offered this term) on animation and advanced rendering. For the truly intrepid, there may be possibilities for joining graduate-level courses in the department on further advanced topics, or doing research or honours thesis projects on graphics.

Graphics as a whole, this course included, has become a fairly mathematical subject. Concepts and methods from linear algebra play a prominent role in particular; if you're not comfortable with the following, now is the time to review (Chapter 2, "Miscellaneous Math" in the textbook by Peter Shirley is a great place to read up on these):

- vectors and matrices
- vector operations: addition, scaling, dot products, cross products in 3D, ...
- matrix operations: addition, scaling, matrix-vector multiplication, matrix-matrix multiplication, determinants, inverting 2×2 and 3×3 matrices, ...
- basis vectors: linear independence, orthogonality, orthonormal bases, changing bases, ...

Some calculus will be needed too: derivatives, integrals, gradients (partial derivatives). Geometry and a few other mathematical topics are just as important, but there we'll cover most of what we need as we go. Many of the bugs that graphics programmers run into tend to be "math bugs"; finding, analyzing and fixing them will require geometric intuition and comfort with linear algebra.

Much of the programming work in this course, though not all of it, will involve a particular API named **OpenGL**. This is the cross-platform standard for fast, typically hardware-accelerated, 2D and 3D graphics. Later assignments and the final project will involve a fair amount of OpenGL programming. However, the API itself will not be taught in lectures: it's expected that you will mostly learn it on your own, with help from the "red book" reference text, labs, office hours, the web, and each other. Lecture time will be spent on core concepts and underlying algorithms, studying OpenGL and other rendering approaches from a high level along the way.

2 Image Basics

With that out of the way, let's take a look at the output we will be dealing with for the rest of the course, images. You probably already have a good idea of what this is, say from experience with digital cameras: a 2D array of **pixels**[1], each of which stores a colour.

What is a colour? This isn't an entirely easy question to answer, and one we will come back to in more detail later in the course. From a purely physical standpoint, colour could be characterized as

[1] Originally pixel was an abbreviation of "picture element", though nobody thinks of it that way anymore. Likewise we may encounter voxels (volume elements) and texels (texture elements) and perhaps more later on in the course.

a total description of how much energy is in each frequency band in the visible electromagnetic spectrum: light waves shorter than infra-red and longer than ultraviolet. However, humans are not capable of distinguishing all of this vast amount of information: our eyes boil it down essentially to three averaged quantities, corresponding to the three types of **colour cones**[2] in our retinas. Roughly speaking, one type of cone measures the amount of reddish-yellowish (long wavelength) photons coming into our eye, another cone measures the amount of greenish (medium wavelength) photons coming in, and another the amount of blue-ish (short wavelength) photons coming in. We can then characterize every possible human colour **perception** with just three numbers.[3] Thus the colours in images are often specified as **RGB** values: one number for the amount of **R**ed, one number for **G**reen, and one number for **B**lue. This of course corresponds to most display technology, such as LCD panels and CRT screens, which have separate elements for red, green, and blue which can average together to fool the eye into seeing most colours.[4] Often people will refer to these as **colour channels**: our normal representation will have a red channel, a green channel, and a blue channel; each channel contains part of the information of the full image.

To summarize: at its simplest an image is a 2D array of pixels, each of which contains three numbers indicating RGB values. If the dimension of the image is m horizontally and n vertically, it will take $3mn$ numbers, which we commonly lay out sequentially in memory. There are of course many different orders to lay out all of these numbers; the convenient standard we will follow for the most part in this course (though not necessarily the best from a performance perspective) is to store the image as follows. Each horizontal **scanline** (i.e. a horizontal slice of pixels through the image, the set of all pixels with a given height) is stored one after the other, with the bottom-most scanline stored first, to which we will give the y coordinate 0. Scanline 1 follows, then scanline 2, and so on up to scanline $n-1$. Inside each scanline the pixels are stored starting from x coordinate 0 on the far left, going up to $m-1$ on the far right. Inside each pixel the red value is stored first, followed by the green and then the blue. We will typically use a 32-bit floaing point value (i.e. a `float` in C++) for each of these RGB values, though in hardware and in some image storage formats 8-bit integers or more exotic data types are commonly used. We'll assume that the usual range for each colour value is between 0 and 1, with 0 being as dark as possible and 1 as light as possible: the RGB value $(0, 0, 0)$ represents the darkest black the display can manage and

[2]Cones are a type of cell, called that way because of their shape under the microscope. We also have "rods", cells that are more sensitive to low levels of light but that do not distinguish colours: this is one reason why in the dark you have a harder time perceiving colour.

[3]This isn't completely true: a poorly understood mutation gives some women a fourth type of cone, enabling them to see differences in colours nobody else can—search the web for "tetrachromacy" for more on this now if you're curious. Other species than humans can have different numbers of cones too, and perceive colours completely differently.

[4]These displays, and in fact just about any display that's ever been built, can't actually produce all the colours our eyes can see, just most of them; we'll come back to this important point later in the course.

the RGB value $(1, 1, 1)$ represents the brightest white the display can manage.[5] If you haven't played with colour sliders before, typically available in any operating system when choosing colours, you should do so to get a feel for how to mix RGB into different colours. In particular, figure out how to get yellow, pink, and brown.

It should be pointed out that in some contexts people flip the coordinate system vertically, so that $y = 0$ is at the top of the display and y increases as you go downwards; in this course (and indeed, with virtually all 3D computer graphics work) we will always take the bottom to be $y = 0$.

While the details of a particular display usually aren't quite as simple as this, our basic mental model of an image is a rectangle with pixels lined up with the integer lattice, from $(0, 0)$ to (m, n). Each pixel conceptually covers a unit square,[6] since we can only see things with an actual spatial extent—not abstract points on a grid. This is an important point to get used to as soon as you can: throughout the course we're going to be using images (and many other things) as mathematical models of real physical things, such as a printed picture or film in the case of an image—to some extent the right approach to graphics is to think of it as simulating the actual physics of optics in the real world.

In this model, pixel (i, j) has a physical extent from coordinates (i, j) to $(i + 1, j + 1)$. We will also identify the centre of this square, coordinates $(i + 0.5, j + 0.5)$ as a special point to represent the pixel—which will be of critical important in the next topic, **rasterization**.

However, as a caveat, in some books (including the start of last year's course!) the pixel centre is put at integer coordinates (i, j) and the spatial extent is the square from $(i-0.5, j-0.5)$ to $(i+0.5, j+0.5)$. This is convenient in using integer coordinates for the pixel centres, but highly inconvenient in terms of thinking about the extent of the image as whole: it starts at $(-0.5, -0.5)$ instead of at the origin, which unnecessarily complicates resizing or remapping images.

3 Rasterization

We'll begin exploring the 3D rendering pipeline, i.e. the process of producing an image, from the end; we've just seen the output (an image) so now we will look at a step before that: **rasterization**. The actual details of a modern OpenGL implementation are a little more involved than this, with a few operations

[5] This assumption is very simplistic too, as it obviously is closely tied to a particular display's capability—how dark and how bright the display can go. Topics such as **High Dynamic Range** and **Colour Matching** deal with being smarter about this.

[6] For this course we will always assume square pixels, as do most computer systems now. However, be aware that there are some important cases in the industry of stretched, rectangular pixels: for example, TV images in the NTSC standard, and many film processes with wide aspect ratios.

taking place after rasterization which we will look at later, but this is a good start for seeing the nuts and bolts of rendering.

Rasterization is simply the process of converting a geometric description of an object to a set of pixel updates in an image. Rasterization determines which pixels touch the object and modifies their colour. The word *touch* is deliberately a bit vague: depending on the desired image quality, different definitions may be appropriate which we will discuss soon. To start with, however, we will define rasterization as follows:

> A rasterized object affects a pixel (i, j) if and only if it contains the pixel's centre point at coordinates $(i + 0.5, j + 0.5)$.

This is a Boolean, on-off type decision.

Here we will focus on by far the most important primitive geometric object for graphics, the **triangle**. While lines and points and circles and more exotic geometric entities obviously can play an important role, the triangle is the simplest primitive (at least, computationally) that can approximate anything else with area. More precisely, a **mesh** of connected triangles can approximate anything else reasonable to arbitrary precision. While some complicated shapes may need huge numbers of triangles for accurate approximation, triangles are simple enough to deal with that it's possible to write extremely efficient algorithms and hardware that can handle the job.

This immediately brings up an important issue: ideal lines and points are infinitely thin, with zero area, and from a physical point of view don't really correspond to anything we can see in the real world. Every real object has a thickness: when we talk about drawing or seeing a line, we generally mean a very thin rectangle or similar shape; points are often thought of as tiny circles or squares with nonzero radius or width. According to our definition of rasterization, an arbitrary infinitely thin line or point would probably never exactly touch a pixel centre, and would never show up in an image as a result; a more reasonable approach is to model them with a nonzero thickness, maybe just as wide as a single pixel. Once such a model with area is in place, they can be broken up, at least approximately, into triangles.

(As an aside, this discussion shows a bias towards rendering real world things. In some contexts, such as data visualization or simply drawing user interface elements, there may be no real world object in mind, and a different approach to what rasterization should mean for lines and points might be called for. For example, a point might be rasterized simply by modifying the single pixel it is contained inside if it is within the bounds of the image; the Shirley textbook has a discussion of abstract line rasterization in section 3.5 if you are interested. However, today's user interfaces, designed for a variety of display resolutions, are beginning to increasingly adopt the real-world rendering perspective (eveyr line or point

has an explicit thickness that may or may not be equal to one pixel) just as they increasingly use OpenGL or similar API's to exploit graphics hardware.)

Going back to our first definition of rasterization, our problem boils down to determining if a point $(i + 0.5, j + 0.5)$ is inside or outside a triangle. There are several ways to do this: the usual modern approach is based on reducing this to tests with each of the three edges, namely a point inside the triangle is on the "same side" of each of the three edges. Therefore let's first take a look at defining lines, and how to distinguish on which side of a line a point is located.

Right away you might be wondering how it's even possible to distinguish the two sides of a line: they're perfectly symmetric. To fix that, we'll look at **directed lines**—you can think of a line with an arrow pointing in one direction. Then if you are looking along that direction, one side of the plane is unambiguously to the left, and one side to the right.

3.1 Implicit Line Equations

There are two standard ways to specify geometric shapes with equations: **implicitly** and **explicitly**.

An explicit description gives a formula for generating all points in the shape. For example, the first formula you probably saw for a line was:

$$y = mx + b$$

where m is the slope and b is the y-intercept. Here you plug in whatever value you want for x, and the formula gives you back the y value for the point (x, y) on the line. This formula, however, has a critical weakness in that it can't handle vertical lines, where the slope would be infinite.

The most common type of explicit descritions is **parametric** shapes, i.e. labels every point in the shape with a parameter value (or several parameter values), which you can plug into a formula to get the coordinates of the point. For example, you could **parameterize** a line with the following:

$$\vec{p}(t) = \vec{p}_0 + t\vec{d}$$

Here $\vec{p}_0$ and $\vec{d}$ are vectors, with $\vec{d} \neq 0$, and t is a scalar real number. This formula generates all points on the line that passes through point $\vec{p}_0$ and is parallel to the vector $\vec{d}$ (the "direction" of the line), as the parameter t varies over all real numbers. Note that for the line that contains an edge of the triangle, say with vertices $\vec{x}_0$ and $\vec{x}_1$, we could take $\vec{x}_0 = \vec{p}_0$ and $\vec{d} = \vec{x}_1 - \vec{x}_0$. This is great for generating new points on the line, but it doesn't help so much for testing whether an arbitrary point is on the line or not.

An implicit description answers exactly this last question: it is a formula for testing whether any arbitrary point is in the shape or not. The shape can then be defined as all points where this formula says

'yes'. There are many fomulas that work for a line, and one specific formula which we will use that has many a multitude of different derivations and interpretations. We'll pick a linear algebraic approach to start.

Say our directed line is built on points (x_0, y_0) and (x_1, y_1), in the direction from $\vec{x}_0$ towards $\vec{x}_1$. By drawing this, you should be able to see that a third point $\vec{x} = (x, y)$ is on the line if and only if the vector from $\vec{x}_0$ to $\vec{x}$ is parallel to the line, i.e. parallel to the vector from $\vec{x}_0$ to $\vec{x}_1$. In more notation, $\vec{x}$ is on the line if and only if

$$\vec{x} - \vec{x}_0 \parallel \vec{x}_1 - \vec{x}_0$$

One way to define parallel is that the two vectors are *linearly dependent*. We can more concretely measure this, if you remember, by taking the determinant of the matrix with these as columns (or rows)—the determinant is zero if and only if the columns are linearly dependent:

$$\det \begin{pmatrix} x - x_0 & x_1 - x_0 \\ y - y_0 & y_1 - y_0 \end{pmatrix}$$