CS 314: Rasterizing Triangles

Robert Bridson

September 4, 2008

1 Rasterizing Triangles

We'll begin exploring the 3D rendering pipeline 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 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, j).

This is a Boolean, on-off type decision. Keep in mind that although the pixel centres have integer coodinates the geometric objects we will deal with could have arbitrary floating point coordinates, and soon we'll also talk about checking points other than the pixel centres with non-integer coordinates, so don't get hung up on the fact that for now i and j are integers.

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.)

Going back to our first definition of rasterization, our problem boils down to determining if a point (i, j) is inside or outside a triangle. There are several ways to do this; the usual modern approach is to use **barycentric coordinates**, which we'll define next lecture. Before we work out what these are, let's take a look at equations to define lines—in particular, the lines which contain the edges of a triangle.

1.1 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. The most common type is as a **parametric** shape, 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 defined to be all points where this formula says 'yes'. Often the formula will be set up in terms of a real-valued function $F(\vec{x})$ which takes a point and returns a scalar number: the point $\vec{x}$ is in the shape if and only if $F(\vec{x}) = 0$. You may remember this from calculus; people may describe the shape as the **zero level set** or **zero isocontour** of the function $F()$. To describe a line, we then have to construct just such a function $F()$.

Again, let's suppose we know two points on the line, $\vec{x}_0$ and $\vec{x}_1$, which we can write out in 2D coordinate form as (x_0, y_0) and (x_1, y_1). We want a function $F(x, y)$ so that $F(x_0, y_0) = 0$, $F(x_1, y_1) = 0$, and indeed every other point on the line also evaluates to zero, but all the points not on the line evaluate to something nonzero. Here we need some geometric creativity to think up such a function.

One geometric function that evaluates to zero exactly on the line through (x_0, y_0) and (x_1, y_1) is the area of the triangle formed by those two points and the point (x, y) which we are testing: the triangle has zero area if and only if it's degenerated into a line, and thus the area is zero if and only if (x, y) is on the line through the two given points.

The area of a triangle in 2D can be found with a determinant involving the coordinates of its vertices:

$$\mathit{area}(x_0, y_0, x_1, y_1, x_2, y_2) = \frac{1}{2} \left| \det \begin{pmatrix} x_0 & x_1 & x_2 \\ y_0 & y_1 & y_2 \\ 1 & 1 & 1 \end{pmatrix} \right|$$

You may have seen variations of this formula including one with a 2×2 determinant:

$$\mathit{area}(x_0, y_0, x_1, y_1, x_2, y_2) = \frac{1}{2} \left| \det \begin{pmatrix} x_1 - x_0 & x_2 - x_0 \\ y_1 - x_1 & y_2 - y_0 \end{pmatrix} \right|,$$

or one with a cross-product:

$$\mathit{area}(x_0, y_0, x_1, y_1, x_2, y_2) = \frac{1}{2} \left\| (x_1 - x_0, y_1 - y_0, 0) \times (x_2 - x_0, y_2 - y_0, 0) \right\|.$$

It's easy to verify they are all equivalent, simplifying to:

$$\mathit{area}(x_0, y_0, x_1, y_1, x_2, y_2) = \frac{1}{2} \left| (x_1 - x_0)(y_2 - y_0) - (x_2 - x_0)(y_1 - y_0) \right|$$

For the purposes of implicitly defining the line, we don't really care about exactly computing the area, so we can simplify and dispense with the factor of $1/2$ and the absolute value, to get:

$$F(x, y) = (x_1 - x_0)(y - y_0) - (x - x_0)(y_1 - y_0)$$

Note this is now actually an **affine** function[1] of x and y, which is always a good thing. You can also verify that $F(x_0, y_0) = 0$ and $F(x_1, y_1) = 0$. In fact, any affine function $F()$ that evaluates to zero at the two points would have to implicitly define the line, from basic linear algebra: this is just the simplest one to write down and to compute.

One interesting and very useful quality of this function $F()$ is that it's zero on the line, positive on one side, and negative on the other. In particular, if you imagine yourself standing on the plane at $\vec{x}_0$ facing towards $\vec{x}_1$, all the points on your right must evaluate to a negative value for $F()$, and all the points on your left must evaluate to a positive value for $F()$. The sign of $F(x, y)$ tells us on which side of the line the point (x, y) is. One way to see this is using the cross-product form of F; it's actually the third component of the cross-product we saw just above (and the other two components are exactly zero). The right-hand-rule tells us whether this third component should be positive or negative depending on which side of the line (x, y) lies.

1.2 Determining Inside/Outside

The last paragraph gives us the key to figuring out whether a point is inside a triangle. A point is inside if and only if it's on the same side (right or left) of the three edges: $\vec{x}_0 \to \vec{x}_1$, $\vec{x}_1 \to \vec{x}_2$ and $\vec{x}_2 \to \vec{x}_0$. We have to be very careful here to direct the edges that way—in particular, don't flip the last edge around to $\vec{x}_0 \to \vec{x}_2$, as that would flip left/right as well.

(As an aside, we get even a little bit more information, which we don't need yet but will come in handy later when we talk about rasterizing 3D triangles. If a point is to the left of all three edges, i.e. the functions $F_{01}(x, y) > 0$, $F_{12}(x, y) > 0$ and $F_{20}(x, y) > 0$ are all positive, the triangle has to be **oriented** counter-clockwise. That is, the path going from vertex 0 to vertex 1, then vertex 2 and back to vertex 0, is a counter-clockwise loop. On the other hand, if the point is to the right of all the edges, i.e. the edge functions all evaluate to negative, then the triangle is oriented clockwise.)

In pseudocode, our inside/outside test will look like this:

```
f01 = (x1 - x0)*(y - y0) - (x - x0)*(y1 - y0)
f12 = (x2 - x1)*(y - y1) - (x - x1)*(y2 - y1)
f20 = (x0 - x2)*(y - y2) - (x - x2)*(y0 - y2)
if f01 >= 0 and f12 >= 0 and f20 >= 0
   return: point is inside triangle
else if f01 <= 0 and f12 <= 0 and f20 <= 0
```

[1] Affine simply means $F()$ is a linear function plus a constant.

```
   return: point is inside triangle
else
   return: point is outside triangle
```

You might notice a problem with this code, which we'll take a look at in a moment, but for now let's take a quick detour to talk about floating point arithmetic.

1.3 Floating Point Arithmetic

One of the nice things about the test we've designed so far is that it only needs some very simple operations: multiplication, addition/subtraction, and testing for signs. In particular, division is avoided, which is a big deal since it tends to both be slower and more error prone (dividing by zero has to always be taken into account). If all coordinates were integers, and overflow wasn't an issue (which is the case, for example, if you're using 32-bit integers but all coordinates are strictly less than 16383 in magnitude), then all calculations would be exact.

However, floating point coordinates are simply too useful to be avoided, and as soon as we do arithmetic with floating point numbers—even something as simple as addition—we have to be aware that results may not be exact. Remember scientific notation for numbers, where for example the number 1305.2 would be written 1.3052×10^3, with a decimal **mantissa** (the 1.3052 in this case) and an exponent (the 10^3). The digits specified in the mantissa are the **significant digits**. Floating point arithmetic on the computer works in exactly this type of notation, though with a binary representation instead of decimal, and a fixed number of significant digits or bits. In particular, 32-bit floating point numbers have 23 significant bits, and whenever an operation like addition or multiplication is performed and the exact result has more than 23 bits, it's rounded to the first 23 bits. What this boils down to is that the result of adding or multiplying two 32 bit floating point numbers might not be exact, except in some special cases[2], but it will be accurate to within about $\pm 2^{-23}$ which is roughly 10^{-7}.

While we're on the subject, IEEE floating point has some useful extra values which do frequently pop up in graphics programs, particularly around bugs. It has a special representation for positive and negative infinity, so if you evaluate $1/0$ for example, you get the special floating point number `inf`; similarly $-1/0$ evaluates to `-inf`. These make sense as appropriate calculus limits, and can be used in further arithmetic operations: for example, $1/$`inf` evaluates to 0. For operations where there is no sensible calculus limit, such as $0/0$ or `inf` $-$ `inf`, the IEEE floating point standard returns a special value called

[2]The most important special case among many is that if the inputs and the exact result are integers less than $2^2 3$ the floating point answer will be exact too.

`nan`, short for "Not a Number". Any operation with `nan` returns another `nan`, and Boolean comparisons with `nan` always return false—even comparisons like `nan==nan` (this is one way to test to see if a number is `nnn` or not: is it equal to itself?). A clear sign of a bug in a graphics program is unexpected `inf` or `nan` values showing up, and it oftens helps in debugging to be able to trace back to exactly where in the code the first one showed up.

1.4 Edge Cases

The floating point issue comes up for rasterization because when the computed quantities `f01`, `f12`, and `f20` are very close to zero, we can't be sure that all the rounding didn't make them accidentally the wrong sign, which would make the inside/outside decision wrong.

However, it's not clear we should care!

Such an error means that triangles with edges that pass extremely close to a pixel centre, at a distance on the order of a millionth of their length, might be rasterized "incorrectly". The human eye is unlikely to spot the problem. If we further factor in that in storing the coordinates of the triangle in floating point, which entails rounding off to 23 bits (some numbers like $1/3$ or even 0.1 can't actually be stored exactly in binary floating point), it becomes uncertain if we could possibly determine the exactly correct rasterization.

Discussing what happens when an edge comes very close to a pixel centre—or actually goes exactly through it—brings up a more important issue that we do have to deal with. You may have noticed in the pseudocode above that we used less-than-or-equals and greater-than-or-equals tests, instead of strict inequalities, rasterizing both the interior of the triangles and their edges. If we had used the strict inequalities, we'd exclude the edges: then if one or more of the edge functions evaluated to exactly zero, rasterization would not include that pixel. If we had a "watertight" mesh of triangles (i.e. no gap between the edge of one triangle and its neighbour) that happened to have edges pass exactly through a pixel centre, we would see holes through the mesh, pixels that aren't touched. These cracks can look awful.

Using less-than-or-equals and greater-than-or-equals instead of strict inequalities fixes the crack problem, since even if some or all of the edge functions evaluate to zero we will include the pixel. However, we instead hit the (less severe) problem of "overdraw": two triangles with a common edge that goes exactly through a pixel centre will both touch the pixel, and if many triangles meet at a point that happens to exactly at a pixel centre the overdraw can be even worse. For opaque triangles this isn't much of a problem, but for partially transparent triangles—which we will get to a bit later in the course—which add some colour to the pixels but don't completely overwrite them, overdraw leads to some stray pixels

getting too much colour, which can look just as bad as cracks.

An ideally robust rasterization algorithm, without cracks or overdraw, is a bit tricky to get right. For this course we won't fully work out a solution, and just stick with the pseudocode as it was given with an overdraw problem, as fixing the overdraw issue properly gets into some more advanced **computational geometry**.[3] The main message to take away from this section is that geometric calculations with floating point numbers almost always are trickier than they first seem: even if your code would be bug-free if everything is computed exactly, when you introduce the rounding errors of floating point arithmetic things can break down.

1.5 Looping over Pixels

So far we have a basic test to see if a given pixel belongs inside a triangle. The simplest rasterization algorithm would be to just loop over all pixels in the image, checking each to see if it's in the triangle. Obviously this could be pretty inefficient. If it's a decent size megapixel image, but the triangle is only a few pixels big, or maybe is completely outside the image altogether and doesn't touch any pixels, we're doing a lot of unnecessary work.

We can accelerate this with something called a **bounding box**, a very commonly used idea in graphics. Looping over the entire image is too slow but is easy, looping over just the pixels in the triangle would be great except that would mean we already have solved the rasterization problem: what we want is something in between, an easy loop that will avoid most of the pixels that aren't in the triangle. The bounding box is the smallest rectangle that contains the triangle: it's easy enough to loop over all pixels inside this rectangle, allowing a significant speed-up.

To calculate the bounding box, we just need to take some minimums and maximums: the x-extent of the box is from $a = \min(x_0, x_1, x_2)$ to $b = \max(x_0, x_1, x_2)$ and the y-extent is from $c = \min(y_0, y_1, y_2)$ to $d = \max(y_0, y_1, y_2)$. This gives us a rectangle $[a, b] \times [c, d]$.

(As an aside, technically what we are working with is known as an **axis-aligned bounding box**, since the sides of the rectangle line up with the x and y axes. It's not as common, but you can also work with rotated bounding boxes, that can fit closer to the triangle—these are known as **oriented bounding boxes**, but you needn't worry about them for this course.)

Our next step is **clipping**: this rectangle may not be completely contained within the image, so

[3] If you're curious, one of the better approaches to dealing with degenerate cases like this is something called "Simulation of Simplicity", which is a way to consistently bias any exactly zero function evaluation to either positive or negative, avoiding the issue. Come talk to me if you want to know more!

we can make it even smaller by clipping away the parts that are outside. Our image extends from the pixel centre at $(0, 0)$ up to the pixel centre in the top right corner at $(m - 1, n - 1)$, so we can now find the rectangular **intersection** of the bounding box and the image. We'll do this with the **clamp** operation, which takes a number and returns the closest number to it within a given lower and upper bound:

$$clamp(x, [l, u]) = \min(\max(x, l), u)$$

In pseudocode clamping could be written as:

```
if x < l
   return l
else if x > u
   return u
else
   return x
```

With this operation we can do the clipping as follows, getting a new rectangle $[\bar{a}, \bar{b}] \times [\bar{c}, \bar{d}]$:

$$\begin{aligned} \bar{a} &= clamp(a, [0, m-1]) \\ \bar{b} &= clamp(b, [0, m-1]) \\ \bar{c} &= clamp(c, [0, n-1]) \\ \bar{d} &= clamp(d, [0, n-1]) \end{aligned}$$

We now can proceed to the loop over possible pixels in the triangle:

```
for j = a_clipped to b_clipped
   for i = c_clipped to d_clipped
      if (i,j) is in the triangle
         set image(i,j) to the right colour
```

Note that these for-loops are inclusive, including the upper bound as well as the lower bound.

While I've written the pseudo-code in terms of a sequential loop, one of the great things about this approach is that every pixel could in principle be tested in parallel. This fact is exploited to great advantage in hardware by **Graphics Processing Units**, which get their high performance from parallelizing graphics operations.