MTAT.03.015 Computer Graphics (Fall 2013)
Exercise session II: Rasterization of Lines

Konstantin Tretyakov, Ilya Kuzovkin

September 16, 2013

The algorithms for rasterization of basic primitives, such as lines, curves, circles or polygons, are probably among the most frequently invoked functions in the world. Think of it: each window on the screen of your computer, each menu element and button, each letter, that appears when you type, consists of multiple small line or curve segments and polygons. Each of those segments has to be rendered onto the screen every time the corresponding part of the screen is redrawn. And the screen does need to be redrawn quite often! Think about what happens behind the scenes when you do something as simple as scrolling down a webpage.

Consequently, rasterization of basic primitives must be performed extremely fast, and a lot of work has been put into development of ways to achieve that. Today we shall have a glimpse at one classical method for fast rasterization of lines – the Bresenham's algorithm.

The base code is provided in the `practice02.zip/practice02.tgz` archive on the course website[1]. Download, unpack and open it. You will have to write all the code (except for the last bonus task) within the `lines.cpp` file. Hence, you may submit your solution as just this single file (perhaps bundled into an archive with the bonus task solution).

1 Basic Line Rasterization

Consider the problem of rasterizing a straight line segment between pixels $\mathbf{p}_1 = (x_1, y_1)$ and $\mathbf{p}_2 = (x_2, y_2)$. Recall that the set of all points $\{\mathbf{x}\}$ on this segment can be mathematically described as

$$\mathbf{x} = \mathbf{p}_1 + t(\mathbf{p}_2 - \mathbf{p}_1), \qquad t \in [0, 1].$$

Boldface letters denote vectors, hence the latter equation should be understood as

$$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} x_1 \\ y_1 \end{pmatrix} + t \left[\begin{pmatrix} x_2 \\ y_2 \end{pmatrix} - \begin{pmatrix} x_1 \\ y_1 \end{pmatrix} \right],$$

[1] Alternatively, all lecture slides and practice session materials are also available on Github: `http://github.org/konstantint/ComputerGraphics2013`

or, writing it even more explicitly, as:

$$\begin{cases} x = x_1 + t(x_2 - x_1) \\ y = y_1 + t(y_2 - y_1). \end{cases}$$

If we fix x, we can find $t = \frac{x - x_1}{x_2 - x_1}$, and solve the equations for y:

$$y = y_1 + \frac{x - x_1}{x_2 - x_1}(y_2 - y_1) = b + kx$$

where

$$k = \frac{y_2 - y_1}{x_2 - x_1}, \qquad b = y_1 - kx_1.$$

This suggests a straightforward algorithm to draw a line segment:

---

for x **in** $[x_1 \ldots x_2]$:
    putpixel $(x,\ b + k \cdot x)$

Exercise 1 (0.5pt). Implement this algorithm within the function `simple_line`.

You will notice, that the algorithm, in the way it is specified above, is bad at drawing steep lines and is completely incapable of drawing vertical lines. The solution to this problem is to act differently, depending on whether $|x_2 - x_1| > |y_2 - y_1|$ or not. If it is true (i.e. line is not steep), we use the algorithm above (iterating along x and solving for y). Otherwise, we iterate along y instead and for each y find the corresponding x. The complete algorithm is then something like the following:

---

if $|x_2 - x_1| > |y_2 - y_1|$:
    for x **in** $[x_1 \ldots x_2]$:
        putpixel $(x,\ b + k \cdot x)$
else:
    for y **in** $[y_1 \ldots y_2]$:
        putpixel $(???,\ y)$

Exercise 2 (0.5pt). Complete the implementation of `simple_line` function so that it would be capable of drawing lines in all directions equally well.

2 Bresenham's Line Algorithm

The algorithm you have just implemented is a rather inefficient way of drawing lines: it requires one floating point multiplication and one floating point addition per pixel. It turns out that it can be significantly optimized to use only integer operations.

Firstly, instead of computing $y = b + kx$ on each iteration we can observe that in fact, the y variable simply starts at y_1 and then is incremented by k each time x increases by 1, i.e. we could get rid of multiplication as follows:

$k = (y_2 - y_1)/(x_2 - x_1)$
$y = y_1$
for x **in** $[x_1 \dots x_2]$:
    putpixel (x, y)
    y += k

Secondly, we can only put pixels at integer locations, so instead of keeping track of a (floating point) y variable, we shall decompose it into an *integer component* y_{int} (the actual coordinate where the pixel is drawn) and an *error term* e:

$$y := y_{\text{int}} + e.$$

Initially $y_{\text{int}} = y_1$ and $e = 0$ (because y_1 is in integer).

Finally, whenever we had to add k to y before, we shall now add this value to the error term instead:

$$e \mathrel{+}= k,$$

As the error term grows, at some point it may become greater than 0.5 (or smaller than -0.5). In this case we "transfer" an amount of 1 from the error term into the y_{int}, thus moving vertically to the next pixel. The resulting algorithm looks as follows:

$k = (y_2 - y_1)/(x_2 - x_1)$
$y_{\text{int}} = y_1$
$e = 0.0$
for x **in** $[x_1 \dots x_2]$:
    putpixel (x, y_{int})
    e += k
    if $e > 0.5$:
        y_{int} += 1
        e −= 1.0
    else if $e < -0.5$:
        y_{int} −= 1
        e += 1.0

Exercise 3 (0.5pt). Implement the algorithm above within the `bresenham_line` function. Note that this is still not the actual Bresenham's algorithm (it uses floating point computations).

Also note that this algorithm only makes sense for $|k| \leq 1$, i.e. for non-steep lines.

To finally get rid of the floating point computations it suffices to multiply every line of code that uses the error term e with $2(x_2 - x_1)$ on both sides. That is, let $e_{\text{int}} := 2(x_2 - x_1)e$. Now, multiplying the sixth line of the previous algorithm by $2(x_2 - x_1)$ we turn it into

$$2(x_2 - x_1)e \mathrel{+}= 2(x_2 - x_1)k,$$

which simplifies to a purely integer-based computation:

$$e_{\text{int}} \mathrel{+}= 2(y_2 - y_1).$$

Exercise 4 (0.5pt). Transform the previous algorithm to use an integer value e_{int} instead of e everywhere. Update your implementation of `bresenham_line`. Make sure the resulting function uses only `int` variables.

Exercise 5 (0.5pt). Your current implementation of `bresenham_line` can only draw "non-steep" lines with $x_2 > x_1$. Finish the algorithm to account for all possible orientations of the line.

Exercise 6* (0.5pt). Study the description of Xiaolin Wu's algorithm for drawing smoothed lines given in Wikipedia[2]. Modify the `simple_line` routine to implement line smoothing in the same way as it is done in the Wu's algorithm. Hint: You may ignore the endpoint handling parts and only study what is done in the main loop.

Exercise 7* (1pt). The game PONG[3] is widely regarded as the mother of all video games. Use the skills you gained in the last week's exercise session to implement it. Your game should allow two players to play against each other using the keyboard. There must be a "Start Game" scene, an actual gameplay scene, where the players control the paddles, and a "Game Over" scene, which is shown when either of the players achieves a score of 10.

[2] `http://en.wikipedia.org/wiki/Xiaolin_Wu%27s_line_algorithm`
[3] `http://en.wikipedia.org/wiki/Pong`