# MTAT.03.015 Computer Graphics (Fall 2013) Exercise session III: Rasterization of Triangles

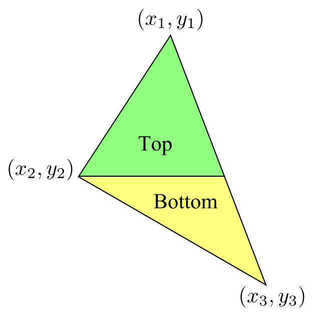
Konstantin Tretyakov, Ilya Kuzovkin

September 23, 2013

Last week we've got a taste for how simple straight line segments can be drawn efficiently. Today we shall study the algorithm for rasterizing the simplest *filled* shape – the triangle. Due to its simplicity it is the most important primitive: models in 3D graphics are always decomposed into and rendered on a triangle-by-triangle basis. The speed with which each triangle is processed and rasterized is therefore key to the performance of a graphics card. Modern GPUs can handle around 2 *billion* triangles per second[1].

The base code is provided in the `practice03.zip` archive on the course website[2]. Download, unpack and open it. You will have to write all the code (except for the last bonus task) within the `triangles.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 Triangle Rasterization



Consider a triangle given by three points $\mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3$. For simplicity in this exercise we shall assume that the coordinates of the vertices are integers[3].

Suppose the points are ordered from top to bottom. The triangle can then be split vertically into the "top" part (lines between $y_1$ and $y_2$) and the "bottom" part (lines $y_2 + 1$ to $y_3$).

To render each part, we will simply run along the edges using two simple line-

[1] `http://goo.gl/fdGcR6`

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

[3] Note, though, that this simplification is not realistic: triangles rendered in most 3D applications would usually have non-integer vertex coordinates. You are welcome to think about how the algorithms discussed below should be adjusted to allow for sub-pixel accuracy.

rasterizers in parallel, filling all the pixels in-between:

```
sort_points_vertically ()
// Step size along the X axis
// for left and right edges
$dx_A = (x_2 - x_1)/(y_2 - y_1)$
$dx_B = (x_3 - x_1)/(y_3 - y_1)$
$x_A = x_B = x_1$
// Render the top part
for $y$ in $[y_1 \dots y_2]$:
    horizontal_line ($x_A$, $x_B$, y)
    $x_A$ += $dx_A$
    $x_B$ += $dx_B$
// Continue similarly for the bottom part
// ...
```

**Exercise 1 (1pt).** Implement the above triangle rendering algorithm within the function `simple_triangle`.

Hint: The point sorting routine and the `horizontal_line` function are provided for your convenience.

Hint: The case where $y_1 = y_2$ must be taken care of specially.

**Exercise 2* (1pt).** You can make this algorithm purely integer-based using (a slightly modified) Bresenham's algorithm to track the edges. Devise a pure integer-based implementation for `simple_triangle`.

Hint: Note that in this case you may not make an assumption that the lines are "non-steep", hence you will need to represent the integral and the fractional parts of $dx$ in separate variables.

## 2 Triangle Rasterization with Interpolation

It is common to have *attributes* associated with the vertices of the triangle. The simplest example is color. Suppose the first vertex has been assigned color red $(1, 0, 0)$, the second is green $(0, 1, 0)$ and the third one blue $(0, 0, 1)$. When rasterizing the triangle we would like to *linearly interpolate* those values at every pixel. For example, the point lying exactly on the edge between the first and the second vertex should be dark yellow $(0.5, 0.5, 0)$, which is the color halfway between red and green. The point lying in the middle of the triangle should be gray $(0.33, 0.33, 0.33)$, an equal mixture of the three colors, etc.

We can modify the simple triangle algorithm to perform color interpolation as follows:

```
// ...
// Step size for color for left and right edges
$\mathbf{dc}_\mathrm{A} = (\mathbf{c}_2 - \mathbf{c}_1)/(y_2 - y_1)$
$\mathbf{dc}_\mathrm{B} = (\mathbf{c}_3 - \mathbf{c}_1)/(y_2 - y_1)$
$\mathbf{c}_\mathrm{A} = \mathbf{c}_\mathrm{B} = \mathbf{c}_1$
// Render the top part
for $y$ in $[y_1 \ldots y_2]$:
    gradient_line ($x_\mathrm{A}$, $\mathbf{c}_\mathrm{A}$, $x_\mathrm{B}$, $\mathbf{c}_\mathrm{B}$, y)
    $x_\mathrm{A}$ += $dx_\mathrm{A}$
    $x_\mathrm{B}$ += $dx_\mathrm{B}$
    $\mathbf{c}_\mathrm{A}$ += $\mathbf{dc}_\mathrm{A}$
    $\mathbf{c}_\mathrm{B}$ += $\mathbf{dc}_\mathrm{B}$
// Continue similarly for the bottom part
// ...
```

Here the values $\mathbf{c}_1$, $\mathbf{c}_2$, and $\mathbf{c}_3$ are vectors representing the colors assigned to each vertex.

**Exercise 3 (1pt).** Implement triangle rendering with color interpolation within the function `color_triangle`.

Hint: The updated point sorting routine and the `gradient_line` function are provided for your convenience.

Hint: Use the provided `vector3f` class to represent colors.

## 3 Textured Triangle

Quite often, rather than (or in addition to) assigning colors to the vertices of a triangle, we shall assign *texture coordinates* $(u_i, v_i)$. Texture coordinates indicate the location of a pixel within a *texture image* that needs to be mapped to the given vertex of the triangle. When filling the triangle we shall interpolate texture coordinates in the same way as we did for colors.

That is, if in the previous algorithm we computed an interpolated color value $\mathbf{c} = (r, g, b)$ for each pixel and set the pixel to that value:

```
al_put_pixel (x, y, $\mathbf{c}$)
```

now we shall be computing an interpolated pair of texture coordinates $(u, v)$, and pick the corresponding pixel from the texture.

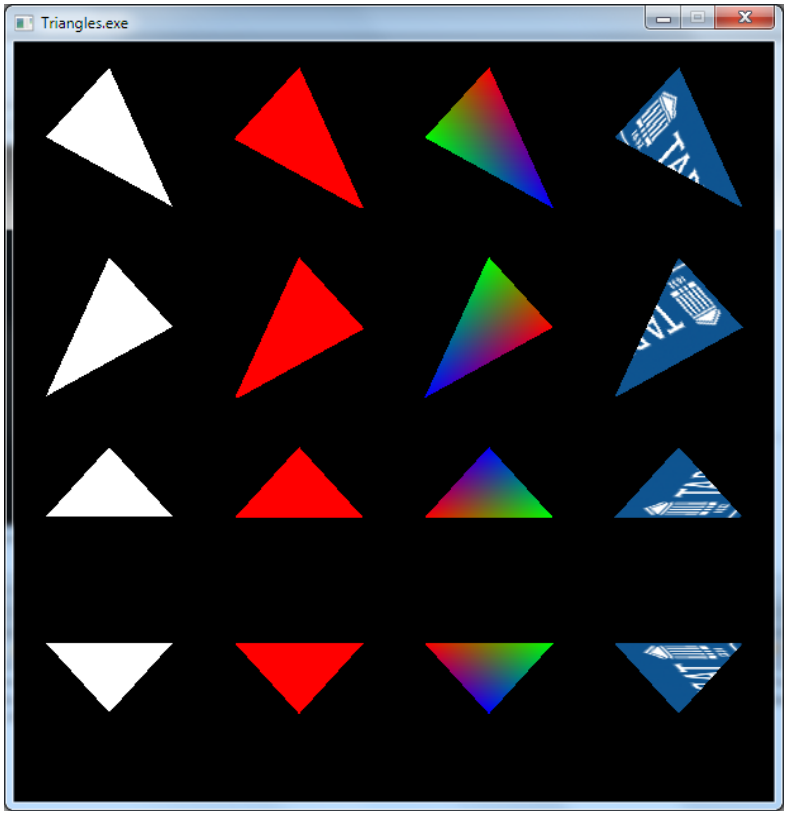
```
al_put_pixel (x, y, texture[(int)u, (int)v])
```

**Exercise 4 (0.5pt).** Implement triangle rendering with texturing within the function `textured_triangle`. For simplicity we shall use the same `vector3f` class here to represent texture coordinates (we simply ignore the third component of `vector3f`).

Hint: To modify `color_triangle` into `textured_triangle` you will only need to change the `gradient_line` to, say, `textured_line`.

**Exercise 5* (0.5pt).** When rendering a textured triangle, most pixels get associated with non-integer texture coordinates. In the previous exercise, however, you were simply rounding them to obtain the corresponding pixel from the texture. Because of that the resulting triangle looks somewhat "jagged". A much better way is to use *bilinear interpolation* to blend the colors of four texture pixels surrounding the source texture location. Read about this technique and implement it in your `textured_triangle` routine.

Figure below shows the screen you should observe after implementing the functions `simple_triange`, `color_triangle` and `textured_triangle`.



## 4 Barycentric Coordinates

You have seen the need to interpolate colors or texture coordinates along the triangle. In practice it is often necessary to interpolate *both* color *and* texture coordinates (perhaps for several textures), as well as a bunch of other things: normals, directions towards light sources, and custom variables.

The concept of *barycentric coordinates* provides an easy way to wrap your head around all those interpolations. Recall from the lecture that every point $\mathbf{x}$ inside the triangle $(\mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3)$ can be represented as

$$\mathbf{x} = \lambda_1 \mathbf{p}_1 + \lambda_2 \mathbf{p}_2 + \lambda_3 \mathbf{p}_3,$$

with $\lambda_i \geq 0$, and $\lambda_1 + \lambda_2 + \lambda_3 = 1$. Importantly, such representation is unique.

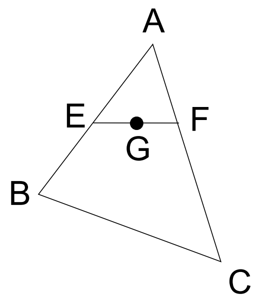
That is, each point in the triangle has a unique tuple $(\lambda_1, \lambda_2, \lambda_3)$, corresponding to it, which is known as the *barycentric coordinates* of that point with respect to the triangle. For example, the barycentric coordinates of $\mathbf{p}_1$ are $(1, 0, 0)$. The barycentric coordinates of the point $\mathbf{p}_2$ are $(0, 1, 0)$, and the coordinates of a point lying in the middle of an the edge between $\mathbf{p}_1$ and $\mathbf{p}_2$ are $(0.5, 0.5, 0)$.

Suppose now that some attributes (e.g. colors) $(\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3)$ are associated with the vertices of the triangle. It turns out now that for any pixel with barycentric coordinates $(\mu_1, \mu_2, \mu_3)$ within the triangle, the corresponding linearly interpolated color can be expressed simply as

$$\mu_1 \mathbf{x}_1 + \mu_2 \mathbf{x}_2 + \mu_3 \mathbf{x}_3.$$

Consequently, it is only sufficient to interpolate the barycentric coordinates for each pixel. Interpolated value of any other attribute can be computed using those coordinates. In practice it is not an efficient method (it requires multiplications at each pixel), but it is a very clear way of reasoning about interpolated attributes.

**Exercise 6 (0.5pt).** Consider the image below



Suppose $|AE| = |EB|$, $|FC| = 2|AF|$ and $|EG| = |FG|$. Compute the barycentric coordinates of the point $G$ with respect to the triangle $ABC$. Suppose that a real-valued attribute is associated with the vertices, such that $A$ is assigned value 10, $B$ is assigned value 20 and $C$ is assigned value $-10$. Use the barycentric coordinates you just found to compute the interpolated value of the attribute at point $G$.

Write the answers in the first lines of the `triangles.cpp` file as a comment.

**Exercise 7* (2pt).** In this bonus task we shall continue the topic of classical games. Another well known game is *Breakout* (also known as Arkanoid, Bananoid, DX Ball[4], etc). Your task is to implement a simple version of Breakout:

- The playing field initially consists of a $5 \times 16$ wall of bricks, that can be broken using a ball. Player starts with 5 lives. Breaking each brick gives a point. When the ball touches the bottom of the screen the player loses a life. Game ends when either all bricks are broken or there are no more lives remaining.

This single level is sufficient for getting 2 points (although, of course, you are not limited in your creativity).

Hint: Reuse the code you developed while implementing Pong.

[4]You can find both an on-line as well as the free Windows version of DXBall by following links from the Wikipedia page `http://en.wikipedia.org/wiki/DX-Ball`