

CS 365 Low Level Operations Lab Drawing and Transforming Triangles

You will write a program that allows you to draw and transform 2D colored triangles. This program is written in OpenGL / GLUT. But unlike the other labs, but you will rasterize the lines and color the triangles yourself instead of using OpenGL's facilities for that purpose. You will render your own model in a pixel buffer and hand it to OpenGL for display.

You will be using GLUT mouse and keyboard functions for user interaction and window management.

Separate from this writeup is the skeleton of an OpenGL / GLUT program that you are to use for your lab. There are sections marked ??? where most likely your code will be inserted.

This skeleton program is already largely written for you. I have simply gutted a few parts out of my own code. For the most part, you need to put in the algorithms. The intent is that it should not be an exercise in learning C.

What it does

When the program starts, it displays a light blue window for drawing.

1. You draw a triangle by clicking the mouse (use the major, or "left" button on a right-handed mouse) at three different spots in the window. After the first two clicks, a line appears in black. After the third click, the remaining two sides of the triangle appear, and the triangle is filled in with a gradient of color.
2. Repeatedly clicking draws more triangles. The program keeps a stack of triangles you have drawn, and places them on the screen with the most recent triangles on top.
3. Pressing the E or e key erases the most-recently-drawn triangle. Since this triangle may lie on top of others, you will need to redraw the whole stack of triangles again.

4. Clicking the minor (usually called “right”) mouse button, with the cursor over a triangle, brings that triangle to the top of the stack. The other triangles retain their relative order. If you are clicking on a point where several triangles cross, you select the bottommost (obscured) one.
You will redraw the whole stack after bringing a triangle to the fore.
5. Pressing the `r` key rotates the topmost triangle by -5° (clockwise) with respect to the origin, `R` rotates 5° (counter-clockwise). Repeatedly pressing `r` and `R` adds/subtracts 5° to the rotation.
6. Repeatedly pressing left, right, up, or down arrow key moves the topmost triangle 10 pixels in the indicated direction.
7. Pressing the home key restores the topmost triangle to its originally-drawn position and undoes any rotation.
8. Pressing `Q` or `q` quits.
9. Pressing `m` or `M` causes the program to print to standard output:
 - The current x, y movement (pixels), and the current rotation angle (degrees).
 - The 3×3 transformation matrix for the topmost triangle.

Overview of Processing

This program does not use OpenGL’s modeling facilities. You never need to call `glVertex` to draw a figure in model space. You never need to set up the projection or the model-view matrices. Otherwise, it is much the same as your previous OpenGL programs:

- A main program initializes GLUT facilities, creates a window, sets up callback functions for display-drawing and for handling mouse and keyboard events.
- The main program also sets up an array of pixels.
- Then it enters the Glut main loop.
- All the keyboard and mouse events call `glutPostRedisplay`, causing the display to be redrawn.
- The display-drawing callback renders the scene into a pixel buffer called `canvas` then gives the pixel buffer to OpenGL to display.

Canvas

The OpenGL/GLUT skeleton program contains an array of pixels called `canvas`. Each pixel in the array contains three bytes of type `rgb_pixel` for the three colors. The program sets up OpenGL so that it uses OpenGL pixel mode to draw the entire canvas from this pixel array.

The parts marked with question marks are suggested locations for inserting your own code. You can insert code anywhere, of course. The screen is initialized and the GLUT callbacks are installed for you. If you update the pixel array, it will automatically appear on the screen when the `display()` handler is called.

The array is one dimensional, but you can index the pixel array using the `LOC` macro. The x_c and y_c canvas coordinates range from 0 to `screenWidth - 1` and `screenHeight - 1`, respectively. The width and height values are initialized to 601×501 . The following illustrates setting a single pixel element at `xc0` and `yc0` to pure white:

```
canvas[LOC(xc0, yc0)].r = 255;
canvas[LOC(xc0, yc0)].g = 255;
canvas[LOC(xc0, yc0)].b = 255;
```

Alternatively you could have a single variable that contains pure white:

```
rgb_pixel white = {255, 255, 255};
...
canvas[LOC(xc0, yc0)] = white;
```

While drawing on the canvas, ignore any points which are out-of-bounds (i.e. $x_c < 0$ or $x_c \geq \text{screenWidth}$ and similarly for y_c). This will have the effect of truncating at the edge of the canvas any objects which overflow past the edge.

At the beginning of the program set the entire canvas to light blue and draw a small + centered at the model coordinate origin in the middle of the canvas.

Coordinates

You have three coordinate systems to manipulate. The transformations are easy. The coordinate systems are:

- The *canvas coordinates* x_c, y_c are the indexes into the canvas array of pixels. The $(0, 0)$ represents the lower left-hand corner of the display, x_c increases to the right, and y_c increases going up the screen.
- The *model coordinates* x, y are the coordinates of the world you are drawing your triangles in. The units are pixels, just as in the canvas coordinate system. However the origin $x = 0, y = 0$ is at the middle of the canvas. You convert between model and screen coordinates like this:

$$\begin{aligned}x_c &= x + \text{screenWidth}/2 \\y_c &= y + \text{screenHeight}/2\end{aligned}$$

The transformations (rotations, translations) on the triangle are always with respect to model coordinates.

- The *mouse coordinates* x_m, y_m are what the mouse handler reports. They are similar to the canvas coordinates except that $y_m = 0$ means the *top* of the screen and y_m increases going down. To convert from mouse coordinates to canvas coordinates compute:

$$\begin{aligned}y_c &= \text{screenHeight} - y_m \\x_c &= x_m\end{aligned}$$

Rendering Lines and Triangles

Triangles are rendered with a one-pixel wide black border drawn according to Bresenham's algorithm. The interior pixels of the triangle are colored.

You can assume a maximum of 10 triangles. This enables you to allocate a fixed-size array of triangles. (In the skeleton this has been set up for you already.)

At each triangle vertex, you assign a pseudo-random color based on its location. The color is then smoothly interpolated throughout the area of the triangle. Do not overwrite the black-line triangle edges.

The formula for assigning a color at vertex x_i, y_i is

$$\begin{aligned}r_i &= (17 \times x) \bmod 251 \\g_i &= (17 \times y) \bmod 251 \\b_i &= (17 \times (x + y)) \bmod 251\end{aligned}$$

When the triangle moves, rotates, etc. it *retains its original colors*. That means you need to compute the colors using the coordinates of the original, untransformed, vertex coordinates.

It is easy to identify the interior points of the triangle for coloring. Compute the barycentric coordinates α, β, γ for each pixel in a bounding box and color only those pixels where $0 \leq \alpha, \beta, \gamma \leq 1$. Unfortunately this method will clobber some of the black edge pixels, so when you are rendering a triangle it is wise to first render the colored interior then draw the edge lines.

Render any partially-drawn triangle as follows: if you have only one point so far draw a small \times , if you have two points draw a line.

Mouse and Keyboard Event Handlers

Skeleton GLUT mouse and keyboard event handlers have been provided for you. Here are some sample routines you can write.

In the mouse-click left-button-up handler, you keep track of triangles as they are being drawn. First click sets the first point for the first triangle. Next sets the second point and draws a line. Third click sets the third point (finishing the triangle), and draws two more lines, then colors the triangle. The triangle is saved away in a data structure, and subsequent clicks start the next triangle.

In keyboard E key handler, you remove the last-added triangle and call a routine to re-draw the entire stack of triangles. The triangles are re-drawn in the order they were entered.

In the mouse-click right-button-up handler, you compute the barycentric coordinates of the mouse location with respect to the bottom-most triangle in the stack. If the mouse is within that triangle, move it to the top and redraw. Otherwise proceed to the next triangle up the stack.

In the keyboard handlers for the various transformations, you update the respective transformation matrices for the topmost triangle.

Transformations

Each triangle keeps original translation and rotation values in its data structure. These values are maintained by their respective keyboard handlers. After updating a translation or rotation value, the 3×3 transformation matrix for the triangle

(rotation \times translation) is updated, so there is always an up-to-date transformation matrix for each triangle.

When the time comes to draw a triangle, you transform the location for each vertex of the triangle. The `m` key causes the combined matrix to be printed out, the `home` key clears the matrix to an identity matrix and the movement and rotation values to zero.

Matrix multiplication common beginner's mistake: do not try to multiply two matrices $A = A \times B$ "in place." Instead compute $C = A \times B$ and then copy C back into A .

Data structures

The `C struct` data structure is similar to a very simple object: it has data but no methods. Your program comes with many `structs` and other data structures set up for you, but you are allowed to add to them.

The primary `structs` are:

- A `rgb_pixel` contains three `unsigned char` bytes for the three colors.
- The `canvas` is a packed two-dimensional array of pixels.
- A `point` contains two coordinates `x` and `y`
- A `triangle` `struct` contains all the information for a single triangle. It may be more than you will strictly need.
 - The three original points
 - The three current-location points (because the triangles move around)
 - The colors of the corners
 - The current displacement and rotation with respect to its original position
 - A rotation matrix, a translation matrix, and a total transformation matrix which is the product of the two.
 - α, β, γ values for the three triangle vertices.
- The `canvas` is a packed two-dimensional array of pixels.
- The `triangles` array contains a pointer to each triangle.

I have provided in the skeleton program subroutines for creating and destroying triangles, as well as manipulating the triangle stack.

Triangle interior testing

One version of the implicit equation for a line that intersects two points $P = (x_p, y_p)$ and $Q = (x_q, y_q)$ is $f_{P,Q}(x, y) = 0$, where:

$$f_{P,Q} = (y_p - y_q)x + (x_q - x_p)y + x_p y_q - x_q y_p$$

In fact $f_{P,Q}$ defines a family of lines $f_{P,Q}(x, y) = c$, each value of c defines a line parallel to the original $f_{P,Q} = 0$ line that runs through the two points, where c is proportional to the distance between the lines.

One way to define a barycentric coordinate system of a triangle with three vertices A, B, C is to compute three constants f_α , f_β , and f_γ , where:

- $f_{B,C} = f_\alpha$ is the line parallel to B-C that contains A
- $f_{A,C} = f_\beta$ is the line parallel to A-C that contains B
- $f_{A,B} = f_\gamma$ is the line parallel to A-B that contains C

These three constants are represented in your triangle structure for your convenience.

To compute f_α , plug the coordinates of point A into the $f_{B,C}$ equation, and similarly for the other two lines.

Now given some arbitrary point (x_0, y_0) , it is possible to compute barycentric coordinates:

$$\begin{aligned}\alpha &= f_{B,C}(x_0, y_0) / f_\alpha \\ \beta &= f_{A,C}(x_0, y_0) / f_\beta \\ \gamma &= f_{A,B}(x_0, y_0) / f_\gamma\end{aligned}$$

If $0 \leq \alpha \leq 1$, it means the line parallel to B-C through (x_0, y_0) is closer to B-C than A is and on the same side as A, so that line goes through the triangle. If all three coordinates meet that condition, the line must be inside the triangle.

Compiling and Testing

The skeleton program is in the `~mglass/cs365/code/` directory called `triangleskel.c`.

There is also a working compiled version of the program called `trianglelab`.

You can use the `glcomp` command to compile it on the `cslab` machines.

To compile `myprog.c` into executable file `myprog` simply type:

```
glcomp myprog
```

Warning: this script enables ANSI C language checking. It does not recognize double-slash comments. If that is causing you problem, you can remove `-ansi` from the `gcc` command line.