Computer Graphics: Assignment 1

Nihal Singh - 150040015, Arunabh Ghosh - 150070006

August 20, 2018

1 Adding support for Brushes of different size

The click callback is caught inside the `gl_framework.cpp` class and depending on the mode either the stroke method of the brush or the eraser is called. The eraser and the brush (provided in the boilerplate) have a class each- `eraser_point_brush_t` and `point_brush_t`. By default only a brush size of 1 pixel was implemented and support for sizes greater than 1 was to be added.
Since the feature had to be implemented for both the `eraser_point_brush_t` and `point_brush_t`, each of which had two overloaded stroke methods, it was wise to create a separate helper method that when passed the coordinates of the click along with the brush size would return a `vector<point_t*>` of all points that would require to be coloured/erased.
Two helper methods were created for this purpose-

- **Square Brush**
  `std::vector<point_t*> get_square_points(unsigned int xpos, unsigned int ypos, int size)`
  This method is pretty straightforward, it returns a vector containing all points bounded by a square around the pixel clicked with side length as brush size.

- **Circle Brush**
  `std::vector<point_t*> get_circle_points(unsigned int xpos, unsigned int ypos, int size)`
  This method iterates over all the points bounded by a square around the pixel clicked with side length as brush size, similar to `get_square_points`. The difference is that the returned points are only those which have a distance lesser than brush size from the pixel clicked. This, essentially, gives us all points that are contained inside a circle with center as the pixel clicked and radius as the brush size.

2 Implementing smooth brush

Two draw a smooth curve between consecutive mouse clicks we use the Quadratic Bézier curves equation. A quadratic Bézier curve is the path traced by the function $\mathbf{B(t)}$, given points P_0, P_1, and P_2 [1]

$$\mathbf{B}(t) = \mathbf{P}_0 + t(\mathbf{P}_1 - \mathbf{P}_0) = (1-t)\mathbf{P}_0 + t\mathbf{P}_1 \ , 0 \leq t \leq 1$$

What above equation gives us, is a curve which connects point P_0 and P_2 with the gradient at both points facing towards P_1 like the picture shown below (solid circle denotes points P_0 and P_2 and x denotes P_1

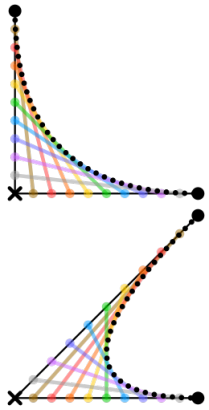

Figure 1: Bezier Curve. Courtesy- Wikipedia article on Bezier Curve)

So our strategy for drawing a smooth curve is this - Say we have a smooth curve drawn through a set of points and now we need to extend it to new point while maintaining that the resulting curve still remains smooth. We denote the current end point of our smooth curve as P_0 and the new point as P_2. For a smooth curve we need to ensure that the gradient of the new segment to be added at P_0 is same as the existing gradient at P_0. Therefore we choose a P1 such that it lies at some distance(d) from P0 along the line extending from P_0 in the direction of the current gradient at P_0. This sets the new gradient at P_2 to be in direction of $P_2 - P_1$. The d is a free parameter and as long as it is long enough for the curve to maintain its gradient at P_0 and then change it to P_2 there should be no problem. In our program it is set to be dist$(P_2 - P_1)/3$.

At the start when we are about to add the second point, only the first point exists instead of a curve. Therefore the gradient at P_0 is not defined. In this case we take the gradient at P_0 to be along P_2 and the Bézier curve simply dissolves into a straight line between the two points. Now that the algorithm behind drawing a smooth curve is explained, we shall now describe how we went about implementing it.

A new brush class is added along with the eraser and the point brush, called `smooth_brush_t`. On selecting the smooth brush mode, a buffer is created to contain the points through which the smooth curve is to be drawn. Each time the user clicks at a point, the callback is caught inside `gl_framework.cpp` and the stroke method of smooth brush is called.

`smooth_brush_t::stroke (unsigned int xpos, unsigned int ypos, canvas_t *canvas)`
This stroke function takes in the new point and adds it to the buffer. Once the size of the buffer reaches two i.e. the user has clicked two times, the `get_bezzier_curve_points(P0 , P1 , P2 , canvas_t *canvas)` is called to draw a straight line between the two points. On clicking more times, more points are added to the buffer. Each time a point is added to the buffer we calculate the point P_1 as described above and then compute the points along which the smooth curve is to be drawn given by the Quadratic Bézier curves equation. For brushes of different sizes we simply calculate the remaining points we need to color using `get_square_points` and `get_circle_points` described in the previous section.

3 Implementing new Primitives- Line and Triangle

The point primitive was implemented in the boilerplate and the implementation for line and triangles was to be added. Inside canvas.hpp where the `draw_context_t` class is defined, a few more member variables were added, with appropriate getters and setters-

`primitive_mode_t current_pmode`

Maintains the current primitive mode. Can be point, line or triangle.
`std::vector<point_t> buffer`
Maintains the buffer for new primitives.

Note: The setters for the different brush modes and primitive modes clear the buffer. This is done so that the buffer state is refreshed when changing draw modes.

Inside the primitive.cpp class the following methods were added-

Main methods that are accessed from `gl_framework.cpp`—

- `void draw_line(const unsigned int x, const unsigned int y, canvas_t *canvas, brush_t *brush)`
  Passed a point, the `draw_line` method adds the point to the buffer. Whenever the buffer size reaches two, a call to the `bresenham_draw_line` method is made and the first element from the buffer is cleared.
- `void draw_triangle(const unsigned int x, const unsigned int y, canvas_t *canvas, brush_t *brush)`
  Passed a point, the `draw_triangle` method adds the point to the buffer. Whenever the buffer size reaches two, a call to the `create_triangle` method is made and the first element from the buffer is cleared.

  Auxiliary methods-
- `void bresenham_draw_line(const point_t &pt1, const point_t &pt2, canvas_t *canvas, brush_t *brush)`
  Given two points, this method implements the Bresenham algorithm for drawing lines. This method makes use of the stroke function from the `brush_t` class, which allows the same method to work for brush and eraser. The implementation is very similar to what was taught in class[2]. There are a total of 6 cases-
  - Horizontal line (x1 == x2)
  - Vertical line (y1 == y2)
  - Sloping upwards with $slope < 45$
  - Sloping upwards with $slope > 45$
  - Sloping downwards with $|slope| < 45$
  - Sloping downwards with $|slope| > 45$
- `void create_triangle(const point_t &pt1, const point_t &pt2, const point_t &pt3, canvas_t *canvas, brush_t *brush)`
  This method calls the `bresenham_draw_line` method thrice to draw the three lines of the triangle.

The calls to these methods were made inside `gl_framework.cpp`. Based on the draw mode, the appropriate brush is chosen and based on the primitive selected, the right function call is made.

4 Implementing floodfill function

We implement the floodfill function by going around in layers around a clicked point and colouring each valid pixel as we reach them. To implement this we maintain a queue of points we need to consider.

We push the first point the user clicked into the queue. On popping a point out, we consider its neighbors and only push them into the queue if the following conditions are satisfied -

- The pixel coordinates should lie within the boundaries of the canvas.
- The color of the pixel should be same as the color of the point the user first clicked on.

If these conditions are satisfied, we push the pixel into the queue. When the pixel is popped, it is colored. In this way we iterate around the point clicked in a breadth-first manner and contiguously color all the pixels of the same color connected to it. Here is how we implement it.

When the user clicks at a point in `fill_mode` the `fill_canvas (unsigned int xpos, unsigned int ypos, canvas_t *canvas)` gets called. This function takes in the point clicked by the user and using the algorithm described above it colors all the pixels connected to it of the same color.

5 Adding keyboard callbacks

Inside the predefined method- `key_callback` that sat in `gl_framework.cpp`, new cases were added to deal with different key presses.

KEY MAPPINGS

- ESC: Exit
- S: Save
- E: Eraser
- B: Brush
- F: Fill mode
- T: Smooth Brush
- C: Choose Color for brush
- D: Choose Color for background
- P: Toggle Primitives
- K/L: Increase/Decrease brush size

References

[1] *Bezier article on Wikipedia* The article linked here.
Last Visited on: 15/08/2018

[2] *Prof. Parag Chaudhuri's Slides*
The slides found here here. Rasterization Basics taught on July 24.