Puzzle Generation

Giorgi Otinashvili

24/11/2024

Problem:

Build a puzzle generator using curves

Subproblems:

- Number of pieces in a puzzle
- What is input domain? (square, rectangle, circle...)
- What are inputs for curve generation?
- How curves can be used for puzzle generation?
- How is shape of each puzzle piece generated?
- How are data points generated?
- Which method can be used for curve generation?

Overview:

The main function is *create_puzzle(image_path, squares_in_row)*. It takes the image path and the number of puzzle rows to be generated as the parameters. Using image size and number of squares in row we can then calculate how many puzzle piece columns we will have in total. Algorithm then generates the right and bottom sides of each square. Curves are generated using cubic Bézier curves due to it simplicity. 4 points are chosen near the puzzle piece side and curves are generated.

Details:

After calling the main function, *generate_squares(width, height, squares_in_row)* is called. This is the function that calculates the total number of puzzle pieces using *width-to-height* ratio. It also creates each square by calculating its (left, up, right, bottom) values. This tuple stores pixels where the square is located.

After that, for each square we draw right and bottom edges. Only right and bottom edges are sufficient since 1. in for loops we go from left to right & from top to bottom and 2. we need to draw only 1 edge to connect 2 neighboring puzzle pieces.

draw_right_edge() and *draw_bottom_edge()* are the functions where edges are drawn.

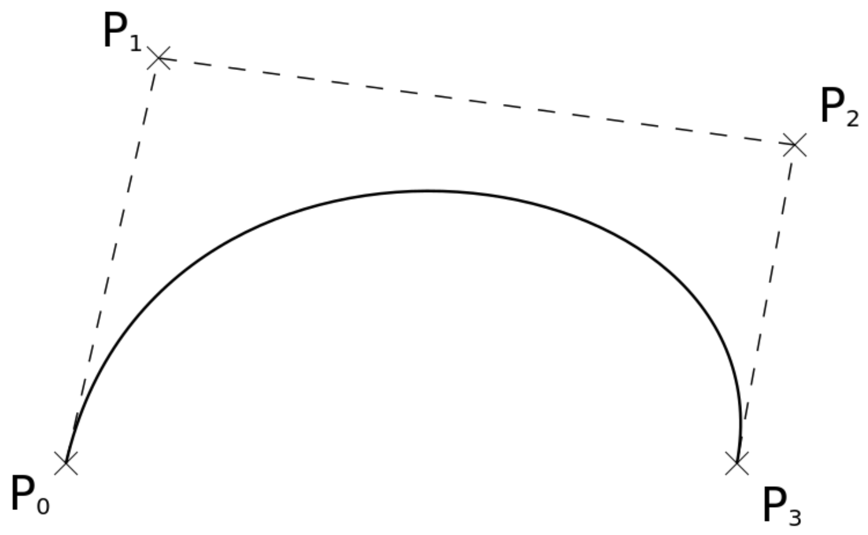
Parameters of drawing functions:

image – image as a 2D array.

square – current piece of puzzle, tuple (left, up, right, bottom)

base_offset_ratio – offset of P1 (generated randomly withtin boundaries)

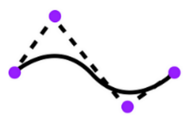
curve_relative_size – size of curve relative to square (default = 0.3)



Standard cubic Bézier curve formula was used, which is derived from 6 linear interpolations:

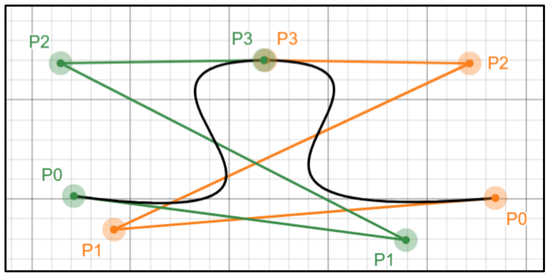
$$\longrightarrow \quad P_0 \cdot (1-t) + P_1 \cdot t$$

$$\longrightarrow \quad (1-t)^2 P_0 + 2(1-t)tP_1 + t^2 P_2$$

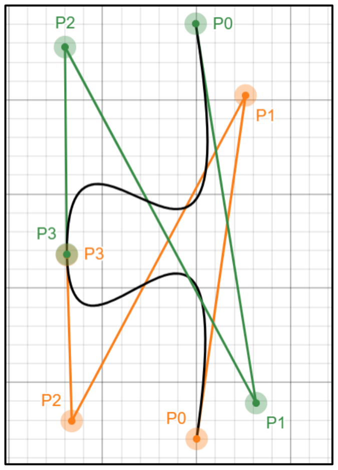


$$\longrightarrow \quad (1-t)^3 P_0 + 3(1-t)^2 tP_1 + 3(1-t)^2 P_2 + t^3 P_3$$

I make 2 curves for each edge, one for the left side and another one for the right side (or upper and lower sides, for vertical edges).



Horizontal edge



Vertical edge

After each puzzle piece is drawn, we show the final image.

Subproblems:

- Number of pieces in a puzzle

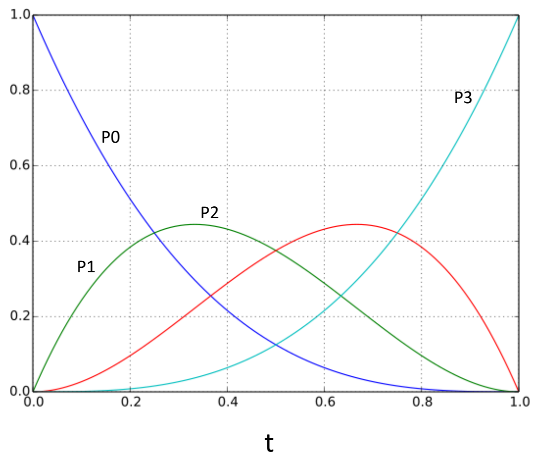
Number of pieces can be controller by user. Since in my code, similar to real world puzzles, each puzzle piece should be a square (or close to square), number of pieces depends on height and width of the image. User cannot input the total number of puzzles, since it may be impossible to create a puzzle with that amount (for example, prime number). That's why user should input the number of puzzle piece rows. Total number of pieces can then be calculated by finding puzzle piece columns using width-to-height ratio, and then multiplying rows by columns.

- What is input domain? (square, rectangle, circle...)

Input can be an image of any rectangular size.

- What are inputs for curve generation?

Cubic Bézier Curve generation requires 4 points: *P0*, *P1*, *P2* and *P3* to be passed as inputs. It also requires a variable *t*, that acts as the distance from each point. For example, *t = 0* represents the starting point *P0*, since its distance from *P0* is *0*. *t = 1* represents *P3*. By passing *t* into formula shown earlier, we will get a point, that represents an interpolated value of the 4 points. It we take *t* from 0 to 1, we will get a set of points, which form the smooth curve.



- How curves can be used for puzzle generation?

Curves help generate inward or outward sides between edges of neighboring puzzle pieces. We can add random values and make the sizes random too to ensure that no two non-neighboring puzzle pieces go together well.

- How is shape of each puzzle piece generated?

Using standard cubic Bézier curve formula as explained earlier. The points used in the formula are similar but always different in each puzzle piece due to different random offsets and square coordinates.

- How are data points generated?

Curve default values (start, end, width or height) are predefined. Using square's (left, up, right, bottom) values and curve's default values, we can generate each data point and also add some randomness to it.

Let's take a vertical edge between two neighboring upper and bottom puzzle pieces. Left curve's points will be:

P0[0] = square_bottom

P0[1] = square_left

*P1[0] = bottom +- (1/4) * square_height*

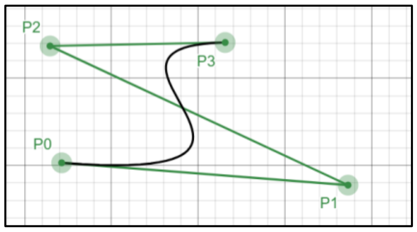
*P1[1] = square_left + (3/4) * square_width*

*P2[0] = square_bottom +- 0.35 * square_height*

P2[1] = P0[1]

*P3[0] = square_bottom +- 0.35 * square_height*

P3[1] = (square_left + square_right) / 2



+-, ¼, ¾ were used as examples, they will all be randomly generated in some range.

0.35 is the curve size relative to square's height or width.

- Which method can be used for curve generation?

To generate curves, many methods can be used, including quadratic Bézier curves, cubic Bézier curves, natural and clamped cubic splines, interpolations (quadratic, cubic, piecewise...). I used cubic Bézier curves since they seemed very easy and do the job of generating smooth curves well.