Bezierbox – Final Report

by Team 5 – Jonah Bedouch, Brandon Wong, Richard Villagomez, and Nicolas DePalma

2025-05-03

This report is also available online at https://bezierbox-reports.vercel.app/report.

Abstract

The goal of this project was to develop a website that allows for the rendering and editing of the bezier curves that make up the glyphs of fonts from .ttf files. Our website allows a user to upload a .ttf file, select characters to edit from that file, and interactively move, add, and remove the points **that are used to make up the bezier curves that make up the glyph for each character.** Once all modifications have been made, the modified font can then be saved as a new .ttf file. This site can be demoed at https://bezierbox.vercel.app.

A video description of this project is available on YouTube at https://www.youtube.com/watch?v=VaI_2OPcbpk.

Technical Approach

Loading from .ttf Files

The first step to being able to interact with fonts is loading them from a .ttf file. These files contain multiple different tables to store all the necessary font information used for a full font. However, because our goal is only to work with the glyph data used to create each character, we only need to interact with a small subset of these tables; these are the head, maxp, cmap, loca, and glyf tables.

To load in a single character's glyph, we first obtain the loca table format from the head table and the number of glyphs from the maxp table. After this, we find the corresponding index from an input character's unicode from the cmap table. The loca table format and this index are used to find the character actual offset in the glyf table. This offset can then be used to load in the point data for the glyph, where each point contains the xy coordinates, whether the point is on the curve or a control point, and whether the point is the end of a curve or not. The end result is a vector of these points which can then be used for rendering the actual glyph.

Navigating these tables tended to have issues. For one, .ttf files are in big endian so that had to be accounted for with custom read functions. Two, it was really easy to get offsets wrong as suddenly start to get wrong values everywhere. There was significant amounts of going back to the apple documentation on .ttf files as well as asking ChatGPT for information on the different table storage formats and help with the code.

We load in each glyph's vector of points to create a vector of vectors which can then be used on the web end to render each glyph for the user to see and interact with.

Rendering Bezier Curves

Before rendering a glyph, we compute its bounding box by flattening all contours and finding the min/max x and y values. These bounds are used to compute a scale and offset that fit the glyph proportionally in the canvas, ensuring consistent display regardless of glyph size. Each point is then transformed from font space to screen space via this scale and offset.

Each glyph is composed of one or more contours, each defined as a sequence of points marked as either on-curve or off-curve. Our renderer iterates through each contour and examines groups of 3 points:

- If both point 1 (p1) and point 2 (p2) are on-curve, a straight line is drawn between them.
- If p1 is on-curve and p2 is off-curve, then the rendering behavior depends on the third point (p3):
  - If p3 is on-curve, we draw a quadratic Bezier curve using p1 as the starting point, p2 as the control point, and p3 as the endpoint.
  - If p3 is off-curve, we interpolate an implied on-curve point between p2 and p3, insert it into the path, and draw a quadratic curve using the "implied" point as the new endpoint. This ensures proper handling of back-to-back off-curve points.

We chose the approach of quadratic bezier interpolation because `.ttf` files are made up of quadratic beziers, each of which has two on-curve points and one off-curve point.

A preprocessing step we overlooked at first involves checking if both the first and last points of a closed contour are off-curve. If so, we insert an implied on-curve point between them at the start of the path. Our rendering was failing until we fixed this.

Once the paths are computed, we call `ctx.quadraticCurveTo(...)` in Canvas2D for each curve segment. After drawing the outlines, the canvas renders each control point for visual clarity - on-curve points are teal circles, while off-curve points are magenta circles. It also draws gray control handles, which are lines connecting the on-curve point to its associated off-curve control points and the next on-curve point.

Our canvas is displayed and works on the web via React with Vite. The user is first prompted to upload a .ttf font file, after which a URL-encoded image of every rendered glyph is displayed in a list. This encoding works by rendering each glyph on a small offscreen canvas, then converting each drawn canvas to a data URL. From this point, the user can select which glyph they want to edit.

Interacting with Glyph Points

Our editor allows users to directly interact with glyph points in the following ways: - Users can click and drag any existing point. The system identifies the closest point to the mouse click (via Euclidean distance thresholding), marks it as "selected," and updates its coordinates as the mouse moves. The canvas dynamically recalculates the Bezier curves, and immediately re-renders the glyph with the updated curves. - In the adjacent sidebar interface, users can toggle a selected point between on-curve and off-curve. This changes how each point contributes to the Bezier path. When toggled, the canvas redraws the curve segments appropriately. - The sidebar also allows the user to manually adjust the x and y coordinates of any point in the glyph. This is helpful for making more precise adjustments, and is rendered immediately. - Any point can be deleted from the contour. This also triggers a re-render.

One implementation challenge we faced was ensuring that all interactions - including adding, deleting, or dragging points - could be reversed via undo/redo. Early on, our undo system only recorded changes during point dragging, which meant that add/delete operations couldn't be undone. We fixed this by capturing the full glyph state before any modification, pushing it to a history stack, and clearing the future stack to invalidate stale redo entries. We use two arrays (history and future) to implement a standard stack-based undo/redo system. This lets the user explore different edits without worrying about making permanent changes.

Saving Changes to .ttf Files

To save the changes of a single glyph to the `.ttf` file, we need to work with four tables: head, loca, maxp, and glyf. Before we do all of this though, we move the glyf table to the bottom of the `.ttf` file to minimize changes that will need to be made later to the offsets of all the data in the file. Without doing this, all the tables are shifted forcing many more changes to be required to ensure the final file could be read and originally causing many issues with saving glyphs modified to have more points or contours than before.

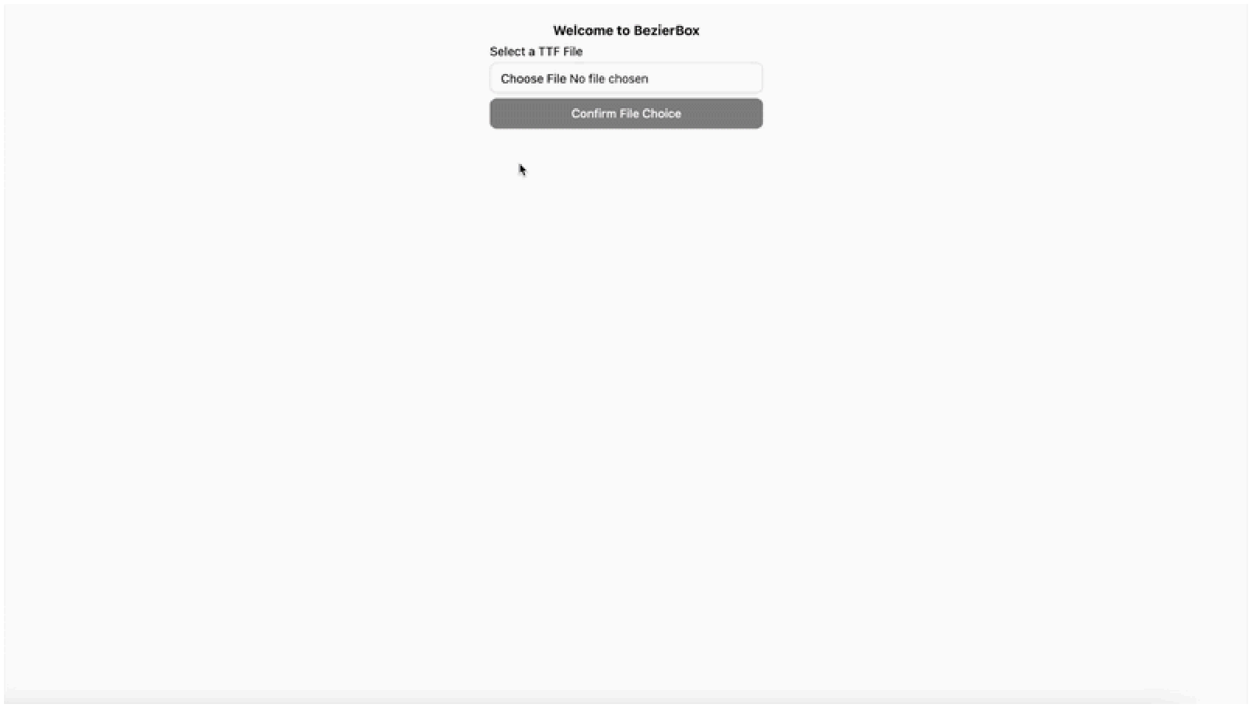
The first table used is the head table, where we obtain the long loca format. This is used to get the offsets for the glyf table entry being updated. Once we have this, we can update the relevant glyph in the glyf table, overwriting the data that was previously there with the new data. If there are less points than before, we pad the rest of the space to the next glyph. If there are more points, new space is added before the next glyph for the new points while shifting all the glyphs after down. The size of this shift is saved for later.

The loca table does not need to be updated if the updated glyph does not take up more space than before. However, if it ends up doing so, the offsets for the glyphs after the current glyph are updated in the loca table. The increased size of the glyph table is also updated to ensure no errors occur when reading the file later. Finally, the checksums are recalculated and stored and the newly updated `.ttf` file is saved and output.

Integration

We use emscripten to compile the C++ program to web assembly to integrate with the React Javascript pages. This allows us to call functions and use data structures like maps from the C++ code to handle the actual interaction and reading from the `.ttf` files and then take those results to render and interact with them in Javascript through the web interface. The interface itself uses shadcn with Tailwind for the various components.

Results

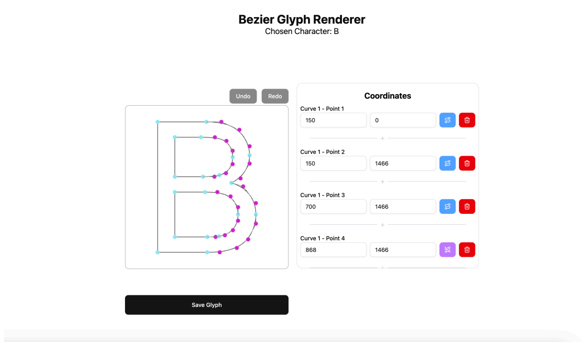

Welcome to BezierBox

Select a TTF File

Choose File No file chosen

Confirm File Choice

Result as a gif – view on web!


Bezier Glyph Renderer

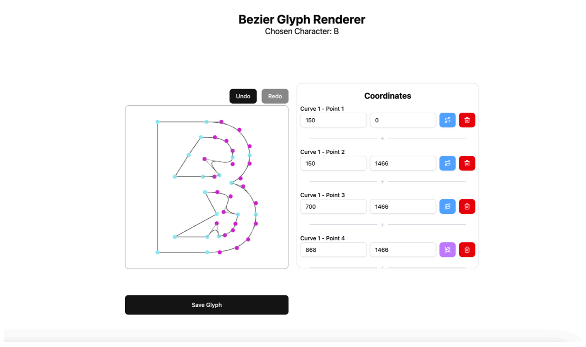
Chosen Character: B

Undo Redo

Coordinates

Curve 1 - Point 1	
150	0
Curve 1 - Point 2	
150	1466
Curve 1 - Point 3	
700	1466
Curve 1 - Point 4	
868	1466

Save Glyph


Bezier Glyph Renderer

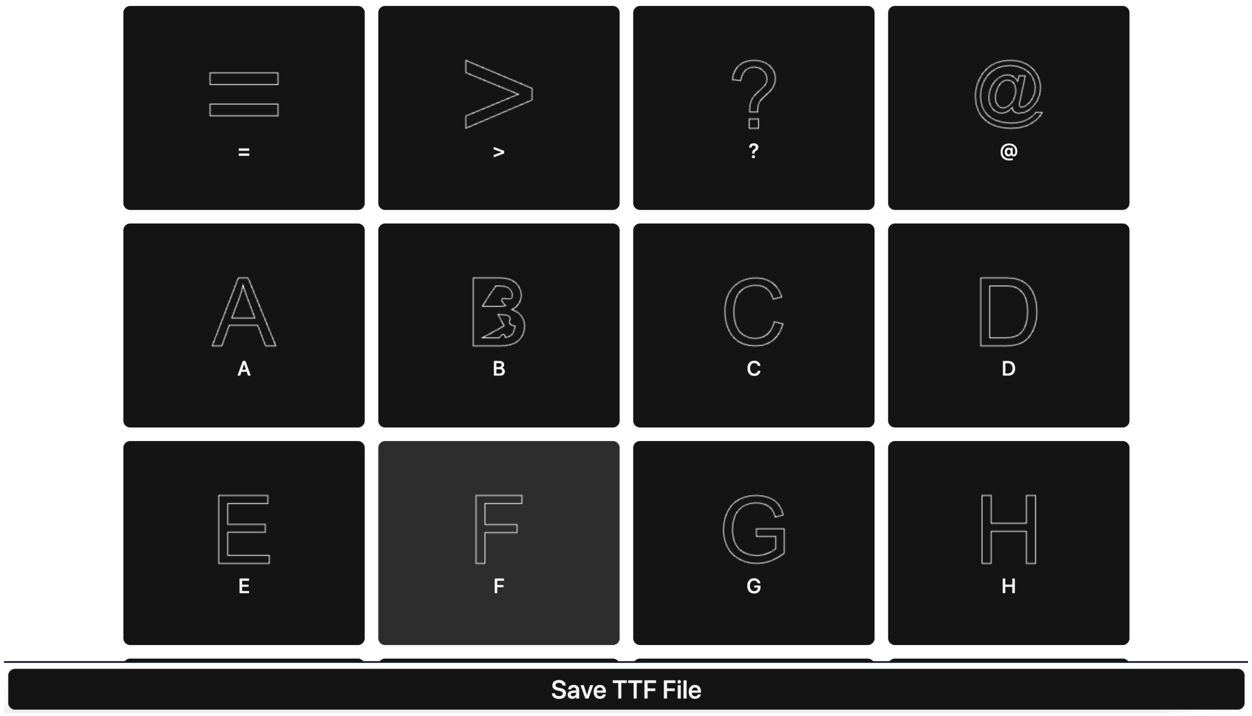
Chosen Character: B

Undo Redo

Coordinates

Curve 1 - Point 1	
150	0
Curve 1 - Point 2	
150	1466
Curve 1 - Point 3	
700	1466
Curve 1 - Point 4	
868	1466

Save Glyph



Team Member Contributions

Jonah Bedouch:

- Converted table data into usable format for React
- Designed React data storage model (using Jotai)
- Set up file upload on React
- Linked file upload, character page, and renderer to backend (including renderer sidebar)
- Added support for adding points to the Glyph from the sidebar
- Converted from React backend data model to the data model needed to save
- Implemented Glyph downloading

Brandon Wong:

- Reading and loading tables and glyphs from a .ttf file in C++
- Reordering glyph table to bottom of a .ttf file in C++
- Saving modified glyphs to a .ttf file in C++
- Page navigation and initial base page format in the React app

Richard Villagomez:

- Quadratic bezier interpolation to display glyphs
- Displaying on and off-curve points, control handles
- Clicking and dragging points, re-rendering
- Deleting points

Nicolas DePalma:

- Setting up initial React renderer
- Transitioning improved renderer to React
- Project Figma mockup
- Configuring url-encoded glyph images