WebRender Report: A Simple Graphics Accelerator for Rounded Rectangles

Alex Gajewski and Allison Ghuman

May 14, 2020

1 Introduction

In our project, we built a simple graphics accelerator that can efficiently render rounded rectangles in real time. The motivation was originally because this is an important part of rendering webpages, but the final product is more generally applicable as an accelerator that could be used for more than just webpages.

Rendering on the web can be quite slow. One would think that webpages take a long time to load because of network latency, but in fact this is not always the bottleneck. For webpages that are very long but have small filesizes, such as for very long text articles, rendering can take upwards of several seconds, while fast networks can download the source files in just a few hundred milliseconds. The figure below shows these timings for the Wikipedia page on COVID-19, which mainly just contains a lot of text, making the filesize very small while making the rendered page itself very long:

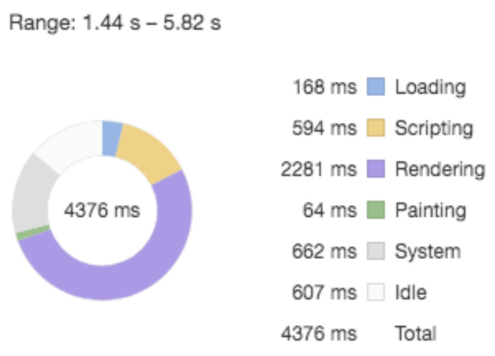

Figure 1: Breakdown of load time for the Wikipedia page on COVID-19.

As the figure shows, the page took less than 200 milliseconds to download, while rendering took nearly 2.5 seconds to complete. This suggests that hardware acceleration for rendering on the web could be helpful in improving the responsiveness of webpages.

On the web, there are typically three types of objects that are rendered: fonts, images, and rounded rectangles. The first two were historically bitmaps, though nowadays fonts and even some images are rendered with vector graphics. Nonetheless, they tend to be more complex, so we wanted to focus on the last category as a first step towards hardware-accelerated web browsing: rounded rectangles. Our goal in this project, therefore, was to make a hardware accelerator that can layer rounded rectangles on top of each other as quickly as possible.

2 Implementation

At first, we were going to use a block of shared DRAM for the displaylist, but because the focus of our project isn't animation, so we don't need to change the contents of the displaylist very quickly, we ended up storing the displaylist in block RAM directly on the FPGA, and writing it byte-by-byte through the slow `ioctl` interface from software.

The displaylist stores a different list of line segments for each row of the display (that is, there are 480 lists of line segments). Each row can have up to 8 line segments, meaning that there can be at most 8 overlapping rounded rectangles on a single row, and each line segment spans from a start column to an end column, and has an RGB color associated with it. All told, each line segment is stored in a 56-bit block of memory, so the 8 of these together form a 448-bit block of memory for each row of the display. The total memory used by the displaylist is therefore $7 \cdot 8 \cdot 480 = 26,880$ bytes, or about 27 kilobytes. Essentially, this is a way of compressing the information in the $3 \cdot 480 \cdot 640 = 921,600$ byte frame into a format that can fit directly on the FPGA.

Initially, we thought we were going to use a dual-framebuffer design, with one framebuffer for the current row, being currently used for writing to the display, and one for the next row, currently being drawn to. We thought this was going to be necessary because we thought it would be impossible to do the rendering in real-time, meaning drawing the current pixel in the two 50MhZ cycles during which it is being sent to the display.

However, it is in fact possible to draw each pixel in its corresponding two-cycle write period. In fact, it can be done in a single cycle. The key is that block RAM can be configured to fetch as many bytes in a single cycle as desired, so we can fetch the entire 56-byte displaylist for the current row at once, in a single cycle. Then, given the current displaylist, a purely combinatorial circuit can look at the start and end columns for the 8 objects in the current displaylist and compare them to the current column number, outputting the colors corresponding to the highest-indexed object, or white if none of them match. Indeed, the fitter estimated that less than 10% of the FPGA interconnect resources were used in the design, suggesting that many more objects per row could be drawn using this method.

3 Hardware Interface Specification

The design registers a 16-bit address space with the operating system, which it uses for writing the display list. The first 10-bits of the address indicate which which row's display list is being written, the next 3 bits indicate which object is being written (of the 8 allowed per row), and the last 3 bits indicate which value is being written:

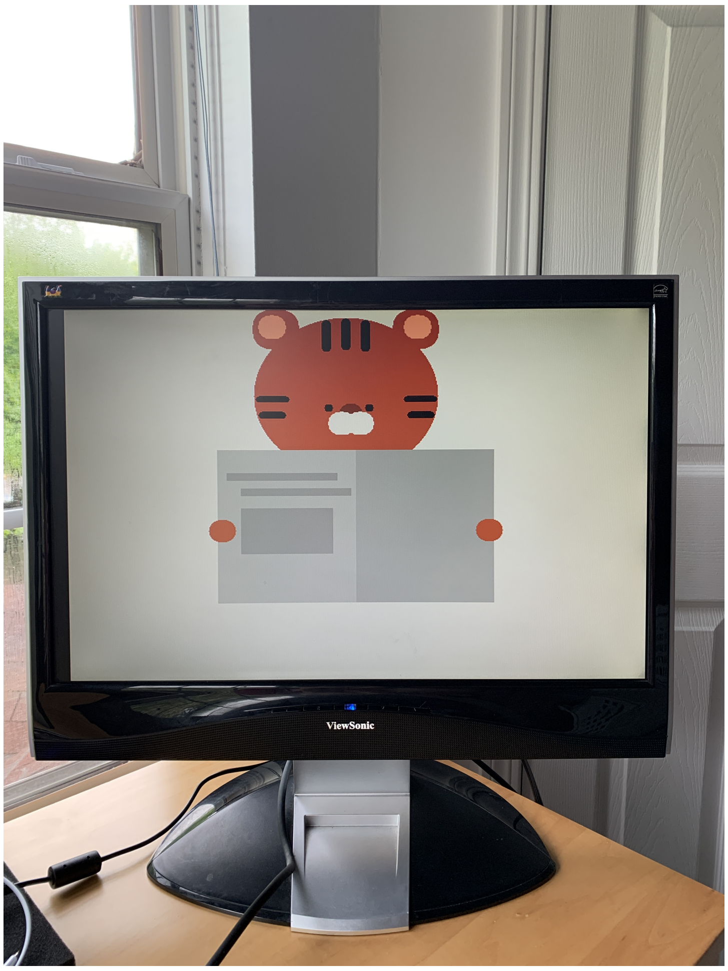

Figure 2: Reference displaylist successfully rendered to display.

Value Number	Value Name
0	Start column
1	Start column (last 2 bits)
2	End column
3	End column (last 2 bits)
4	Red
5	Green
6	Blue

Table 1: Address mapping of displaylist values.

The column numbers are each 10 bits, so they don't fit within the 8 bits of the `ioctl` interface. Instead, they are broken up into two blocks of 8 bits, with the second block only using its bottom two bits to store the extra data. After being read back from the block RAM, the two bytes for the column numbers are merged in a `generate` block that iterates over the 8 objects in the current row.

When writing the displaylist, the rows may be written in any order, but when writing a particular row's displaylist, the objects should be written sequentially, and no objects should be skipped. In particular, if there are fewer than 8 objects that need to be displayed on a given row, empty objects with `start_col == end_col == 0` should be provided to fill the remaining space. When the last byte of the last object is written, the hardware write buffer will save the current row's displaylist into block ram, at which point a different row's displaylist can be written.

4 Software Specification

The software component of the project takes an array of rounded rectangles and converts it into a displaylist in the above format, before writing the displaylist to the hardware and rendering it to the display. Each rounded rectangle takes a number of parameters:

Parameter Name	Description
X position	Distance from left edge of display of upper-left corner, ignoring border radius
Y position	Distance from top of display of upper-left corner, ignoring border radius
Width	Width of rectangle, ignoring border radius
Height	Height of rectangle, ignoring border radius
Border radius	Radius of each of 4 circles tangent to the rectangle's four sides
Red	Red component of the rectangle's color
Green	Green component of the rectangle's color
Blue	Blue component of the rectangle's color

Table 2: Parameters for a rounded rectangle.

The orientation of the X and Y position parameters was chosen to match that of CSS styling for absolutely-positioned HTML objects. The Y position is treated as a row number internally, but since VGA counts rows starting from the top of the display, this is flipped and ends up being the right way around, matching CSS's `top` property.

5 Next Steps

This project was our first experience with hardware programming, so it was intentionally small in scope. Going forward, it would be interesting to return to the original motivation and use the graphics accelerator in a simple web browser. The biggest change necessary for this would be to add support for rendering bitmaps (like fonts). This could probably also be done in real time, using just one cycle per pixel: we could add support for displaylist objects of different types, one being the existing line segment type, but another being a bitmap type. As long as the bitmaps are small enough to fit into block ram, which should be the case for fonts, a bitmap object can just be a pointer to an element of an array of bitmaps. Then during the dead time before the start of a row, all of the bitmaps that appear on the current row could be loaded into flip-flops (only saving the rows of the bitmaps that correspond to the current row), and then these could be incorporated into the combinatorial logic for rendering a single pixel. The bitmaps for fonts would really be masks, using just a single bit for each pixel, and the bitmap-pointer object could include some color information (used to set a font color). It would be difficult to do font scaling using this approach, though it might be possible to also do this during the dead time before a row gets rendered.

Another interesting extension towards making a full web browser would be to use the hardware to accelerate vector font rendering. The slowness in rendering the Wikipedia article above suggests that rendering these vector fonts might be a bottleneck in rendering large text-heavy webpages, and since this is mainly a bunch of polynomial computations, it might be well-suited for dedicated hardware acceleration.

Finally, towards making a full web browser, we would need to load and parse actual webpages. We would at least want to support local HTML pages, but since there is a real Linux operating system running on the FPGA already, it shouldn't be too difficult to use the networking libraries already included in Linux to fetch real webpages from URLs. We would also ideally make it interactive with a simple keyboard interface for scrolling and selecting links to click on.