EmbeddedGraphicsRender - Software for Rendering 3D Graphics in Memory Constrained Embedded Systems

Adam Alderton
aa2301@cam.ac.uk
(Dated: January 28, 2022)

The software which this report considers, available in a GitHub repository of the same name, was implemented to be capable of rendering simple 3D graphics while under heavy memory constraints. To this end, various unconventional techniques at the low-level were employed including sub-byte manipulation. While still limited in certain aspect, the framework efficiently renders simple graphical examples and writes the results to a mini OLED display over a custom SPI driver.

I. INTRODUCTION

The software produced, adapted from the *Warp* firmware [1], is able to render simple 3D graphics in real-time and display the results over a purpose-built driver communicating with a mini OLED display. 3D graphics and their low level considerations have been pertinent to many state-of-the-art 3D graphics engines including *Vulkan* and *DirectX* [2]. Without low-level considerations and optimisations, these commercial would struggle to be viable on modern hardware [2]. This project proved to be an effective incubator in which to explore low-level techniques as efficient implementation is forced by the heavy memory constraints exerted by the microprocessor [3].

II. IMPLEMENTATION

Although it exists in a plane, the most primitive 3D shape is a triangle. Hence, rendering triangles is a common practice in graphics rendering as more complex polygons can be constructed from a set of triangles [4]. Hence, the first key component of the program is the triangle rasterisation functionality. That is, triangles defined in a continuous 2D space must be translated into a discrete 'pixel space' for eventual display on the OLED display. This process is notoriously non-trivial [2]. To do this effectively, an adapted version of the Bresenham algorithm was used - an algorithm otherwise used to rasterise lines [4]. Fortunately, this algorithm can be carried out with only integer operations if implemented correctly which is of course favourable in compute and memory limited systems.

While complex, innovative and given a bespoke implementation, the 3D rotation and projection functionality of the program is not necessarily the most innovative part of the implementation so its discussion will be omitted for brevity. Its custom and efficient implementation is discussed further in the repository.

Instead, further consideration is given to the techniques surrounding the screen buffer and its writing to the OLED over an SPI. As to be able to render anything of any significant size within the available 2kB of memory, careful consideration had to be given to how the screen buffer would be stored. Accordingly, only a subset of the available pixels were used. More importantly and with more novelty, two pixels were stored in each byte to essentially double the pixel storage capacity. However, this comes at the cost of limiting the number of available colours to red, green, blue and black only. Additionally, only three relative intensities could be used (each to give the illusion of plane angle with respect to a viewer). This methodology is outlined in FIG. 1.

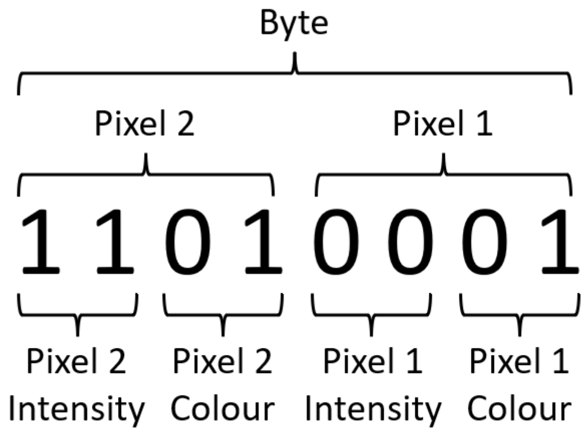

FIG. 1. A breakdown describing how one byte in the program was used to contain information regarding two pixels. Data was placed and retrieved by sophisticated bitwise operations.

A holistic diagram of the described workflow is given in FIG. 2.

III. FURTHER CHARACTERISATION

As this project primarily revolves around efficient implementation for graphics rendering, the main parameters concerning its characterisation could be deemed to be the various implementation techniques discussed in Section II. However, we now continue to further characterise the produced software. There are two pragmatic avenues along which we can quantify the performance of the software - in terms of *compute* performance and in terms of *memory* performance.

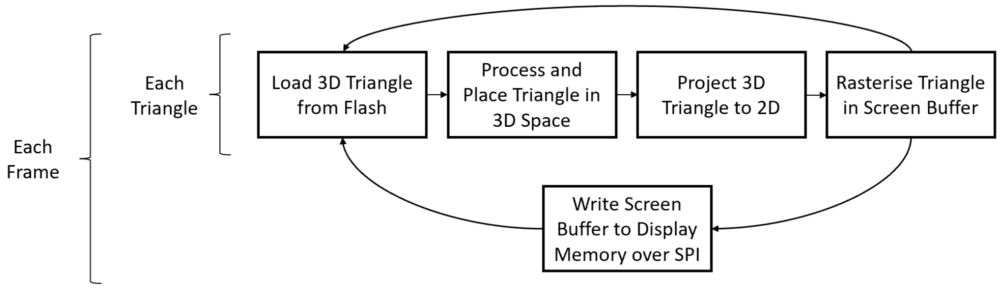
Each Frame
Each Triangle
Load 3D Triangle from Flash
Process and Place Triangle in 3D Space
Project 3D Triangle to 2D
Rasterise Triangle in Screen Buffer
Write Screen Buffer to Display Memory over SPI

FIG. 2. A flowchart showing denoting the workflow of the program.

The following experiment was carried out to quantify the compute performance of the software. Various numbers of triangles were loaded, projected and rasterised and written to the OLED display. The time elapsed to do so was measured and averaged over 100 measurements. The results of which are shown in FIG. 3. Also shown is a least-squares fit linear model. As, in theory, no non-linearities should be introduced in the time elapsed to render - the processing of each triangle is independent of all others. The fit model of course does not pass through (0, 0) as there a consistent time elapsed for writing the frame to the OLED display - this is invariant with respect to the number of triangles rendered. From the data we can deduce that the average simple triangle takes 0.51 ± 0.2ms to process and the average frame takes 30.6 ± 0.2ms to write.

A means to inspect the memory-based performance is to inspect the .map file produced by the linker in the build process. For example, considering the SRAM usage in the cube demo, we see that the .data section consumed 292 bytes which contains certain instances of structures instrumental to the Warp firmware. The .bss section consume 540 bytes and includes other quantities instrumental to Warp. Zero bytes are allocated to the heap. This leaves 1216 bytes for the stack. In the case of the cube demo, a screen buffer of 38 pixel side length was used for a total of $\frac{38^2}{2} = 722$ bytes. This leaves 494 bytes remaining for other processing - this is quickly consumed by floating point variables used in projection. Various quantities in the .data and .bss sections could easily have been removed for a marginally bigger display area however this was not done as to allow for future work to more easily integrate with other functionality of the Warp firmware.

IV. CONCLUSION

This report set out to summarise some evaluations of a software written to render simple 3D graphics. Unconventional means to achieve this under such oppressive memory constraints were described and discussed. The undertaking of this project proved to be a large yet compelling opportunity to learn about the field of bare-metal programming and low-level graphics rendering.

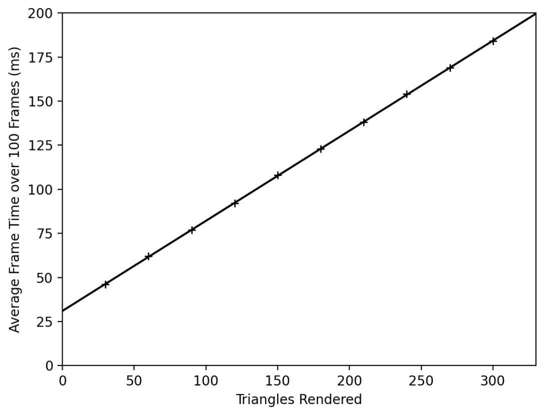

FIG. 3. A plot outlining the relation between the frame time and the number of triangles rendered in a frame. The points given are the mean values over 100 identical frames for that number of triangles. As expected, a strong linear relationship was found as demonstrated by the least-squares fit linear relation.

[1] P. Stanley-Marbell and M. Rinard, Warp: A Hardware Platform for Efficient Multi-Modal Sensing with Adaptive Approximation, IEEE Micro **40** (2020).

[2] S. Kosarevsky and V. Latypov, *3D Graphics Rendering Cookbook* (Packt, Birmingham Mumbai, 2021).

[3] M. Barr, A. J. Massa, and M. Barr, *Programming embedded systems: with C and GNU development tools* (O'Reilly, Beijing, 2006).

[4] M. Abrash, *Michael Abrash's graphics programming black book* (Coriolis, Albany, NY, 1997).