A light-weight CPU implementation of a 3D graphics pipeline for embedded systems

Charles Nicklas Christensen (GitHub: charlesnchr / CRSid: cnc39)

November 3rd, 2017

The problem of rendering 3D computer graphics has attracted an increasing amount of attention since its conception in the 70s, greatly accelerated by the emergence of personal computers in the late 70s and 80s as well as GPUs in the 90s.

While 3D rendering technology have matured considerably as a whole, both in terms of hardware and software, the field is still evolving in several ways: first there is a never-ending push towards higher fidelity, but at the same time there is also a push towards smaller and more energy-efficient 3D graphics capable devices that enable a range of applications such as mobile gaming (e.g. smartphones and handheld consoles), virtual reality and augmented reality.

Some of these portable devices have become relatively powerful and support large graphics libraries, the most popular being OpenGL ES (OpenGL for Embedded Systems). Support for these libraries are restricted to systems that feature a GPU or IGP (Integrated Graphics Processor), which however virtually no MCUs have (only exception known to the author is a recent PIC32MZ MCU, but even here the GPU is only designed for 2D graphics).

The problem. As the major graphics libraries rely exclusively on GPUs, no standards for CPU-only systems exist. For general-purpose computers some workarounds do exist, for instance running WebGL using a software rendering backend via a javascript engine such as Chrome V8. But this has a large memory footprint and is obviously not viable for embedded systems.

No options seem to exist for embedded systems without GPUs. The aim of this project is to address this empty space by developing a light-weight CPU implementation of a 3D graphics rendering pipeline runnable on a MCU with low memory. The pipeline is shown on figure 1 and is described under paragraph *Approach*.

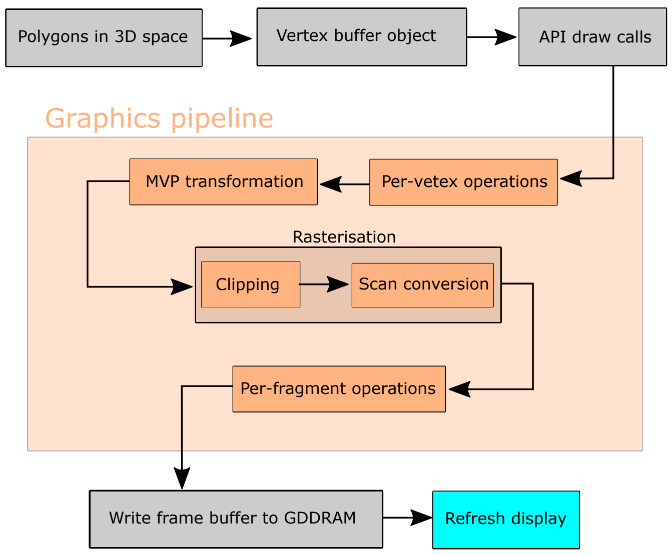

Figure 1: Steps to perform a 3D graphics rendering. The orange section referred to as the graphics pipeline.

Applications. At the end of the project a 3D graphics library tailored to a MCU of choice (likely to be Freescale KL03 or KL05) will be provided. This library will serve as an API for embedded system developers. As such it will ideally enable applications on a MCU to render 3D graphics to a display in order to either visualise or interact with data and models directly. For light 3D applications this may prove to be a desirable solution as the system can then be made small and cheap (for instance using the KL03) by omitting a dedicated graphics chip.

Approach. The suggested partitioning of the problem is shown in figure 1, where each step will have to be solved and implemented – all of which are elaborated shortly. The steps inside the orange region are those that constitute the graphics pipeline and is inspired by the pipelines of OpenGL and Direct3D. Arguably, the second last step should also be part of the pipeline, namely writing to the display's memory (GDDRAM: Graphic Display Data RAM), but in the case of a MCU interacting with an arbitrary display, this step will depend largely on the display driver and so will need an amount of customisation, whereas the rest of the pipeline should hopefully be reasonably transferable to other embedded systems.

The sequence of steps seen in the figure do the following: loads or generates polygons, write those polygons to a vertex buffer object (i.e. a suitable data structure), after which the type of draw command is specified via the API. The pipeline then starts with the vertex shader, where transformations and potentially lighting are handled per-vertex, followed by a Model View Projection transformation (n.b., also a per-vertex operation and thus traditionally part of the vertex shader, but shown separately for clarity). The rasterisation can now begin, where a clipping must first be performed to establish what is visible within the viewport (the Sutherland–Hodgman algorithm will likely be used). After the clipping a scan conversion can take place, in which each pixel in a framebuffer is assigned a value (the scanline algorithm will likely be used), and for each pixel a depth test with a Z buffer has to be done to ensure visibility (not occluded). The fragment shader then takes over and carries out per-fragment (pixel by pixel) operations such as shading and colouring. The resulting framebuffer is written to the GDDRAM and shown as the display is refreshed.

Timeline. The steps present in figure 1 each represent a subproblem that needs to be solved and implemented. Figure 2 shows the estimated times needed to address those subproblems in addition to the general setup preceding the pipeline (referred to as prerendering stage) and a performance analysis at the end of the project.

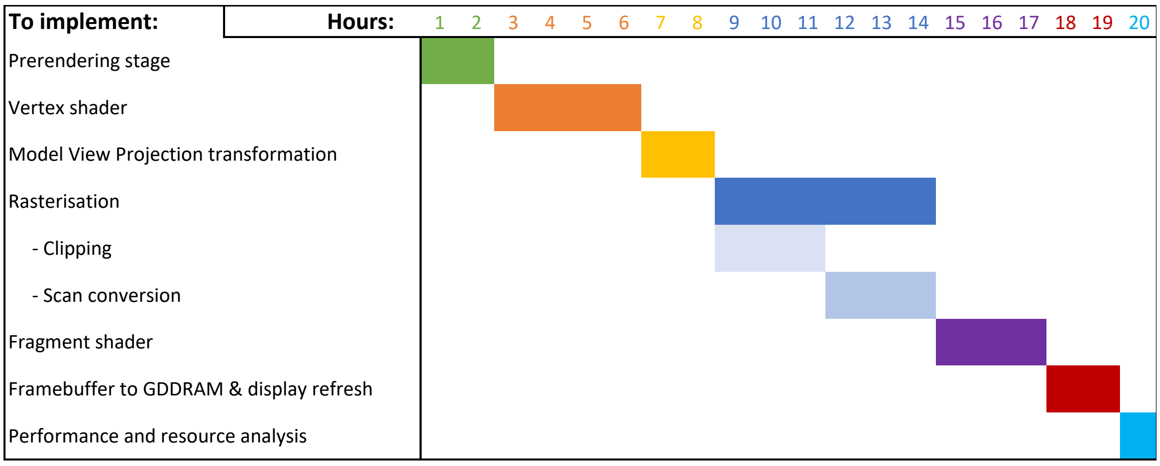

Figure 2: Gantt chart of the project, most items being a part of the rendering pipeline.

As for the dates: work on the project will commence as soon as it is approved, presumably mid-November, and a couple of hours will be dedicated to it each week onward, yielding an estimated completion by the end of January. This should give plenty time to work on the final report and also provide a reasonable buffer if the estimated completion date is delayed due to unexpected complications.

Justification. I have experience with computer graphics and real-time rendering. Although I have never done an implementation like this before, I have worked with OpenGL, WebGL and Direct3D and know quite well what they do, and so from my experience I think the time estimates present in the Gantt chart should hold.