Final Project Report – Coursework #6

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

Charles Nicklas Christensen
cnc39@cam.ac.uk
GitHub: *charlesnchr*

4B25: Embedded Systems for the Internet of Things
Department of Engineering
University of Cambridge

May 17th, 2018

This project has seen an attempt to implement a 3D graphics rendering pipeline on an embedded system featuring a display. The de facto standard of 3D graphics libraries for embedded systems is OpenGL ES, but the library requires a dedicated GPU and has a memory footprint much greater than the 32 kB available on the Freedom KL03 board. The goal has been to implement the essential steps of the traditional rendering pipelines on a CPU and with very limited available memory. The motivation has been to see how far the KL03 can be pushed, and see how small an implementation can be made consisting of a graphics pipeline. The KL03 is obviously not a very good choice as a backend for 3D graphics, but, in general, research into low-resource 3D rendering should be interesting to anyone fond of video games or real-time visualisation on portable devices. Pushing the limits is what drives the industry towards smaller and more power efficient devices.

A more detailed description of the project can be found in the project proposal and the project interim report – this document will focus only on results from here on.

Power usage. The first means of characterisation is of the power usage when using the display. We shall consider a simple 3D geometry consisting of two cones where one is in front of the other. The graphics pipeline was first implemented in MATLAB before porting it to the Freedom board, and three renderings of this geometry can be seen on figure 3 – one zoomed (left), one with the native 96x64 resolution of the Adafruit 0.96" SSD1331 mini Color OLED (top-right) and a high-res rendering using 100 times the number of pixels (bottom-right).

In another coursework it has already been clearly established that the power usage depends on the displayed

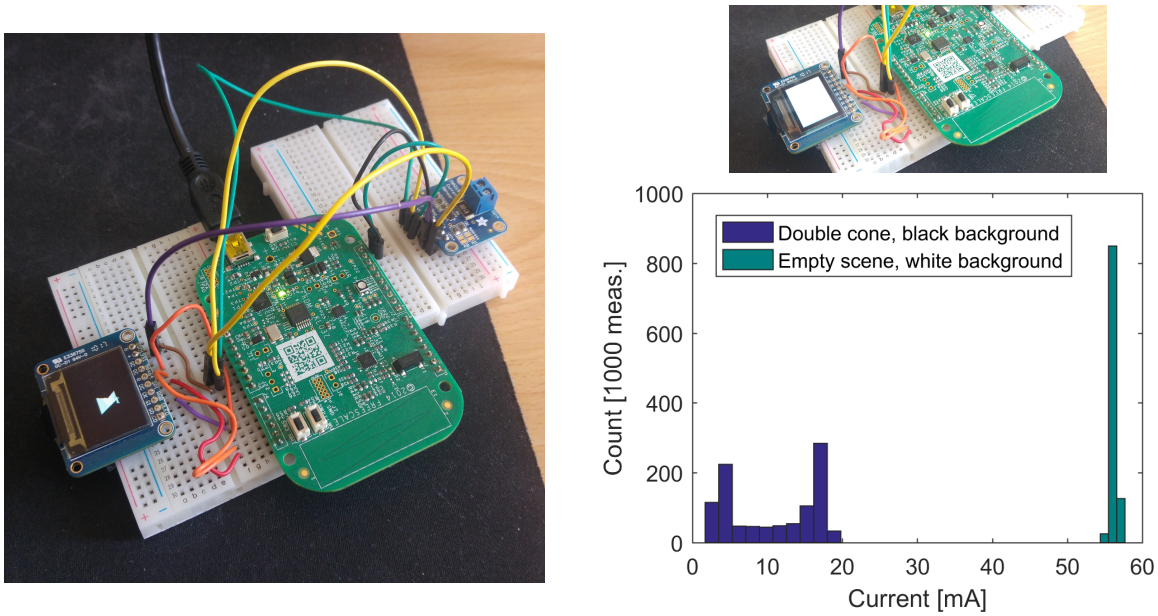

Figure 1: Current measurement using the current sensor breakout board, INA219. The current usage when displaying a simple 3D geometry consisting of two cones is compared to that of a white rectangle filling the whole display, see left and top-right image. The current was measured consecutively 1000 times with no forced delays, and a histogram of the readings are shown on the lower-right plot.

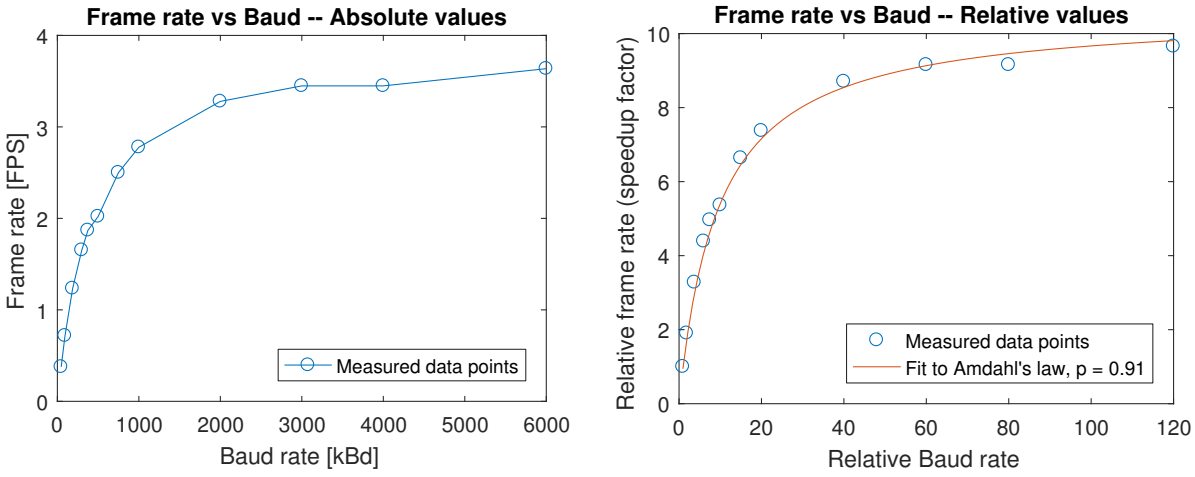

Figure 2: Performance characterisation as a function of the specified Baud rate. The measured frame rates are averages of the rendering time of 20 identical frames. (Left) Data points are shown with absolute values. (Right) Data points are shown as relative quantities to compare with Amdahl's law.

colour. Both cones will be rendered as white, and for comparison the power usage when displaying a white rectangle filling the entire display is also measured. The rendering of the two cones and the circuitry connecting the display and current sensor to the MCU can be seen on the left of figure 1. The white rectangle is displayed in the image on the top-right of figure 1, and two histograms of the respective current readings or shown on the bottom-right. As expected the power is higher the more pixels are displayed, and therefore the rectangle requires several times more current to display than the two cones, namely a factor of about 5, ~10 mA versus ~57 mA. The ratio of the lit pixels is a more than a factor of 10 (96×64 divided by 540), but naturally there is also a fixed usage in keeping the display on and communicating with it. Interestingly, there is a much higher variance when only displaying the two cones.

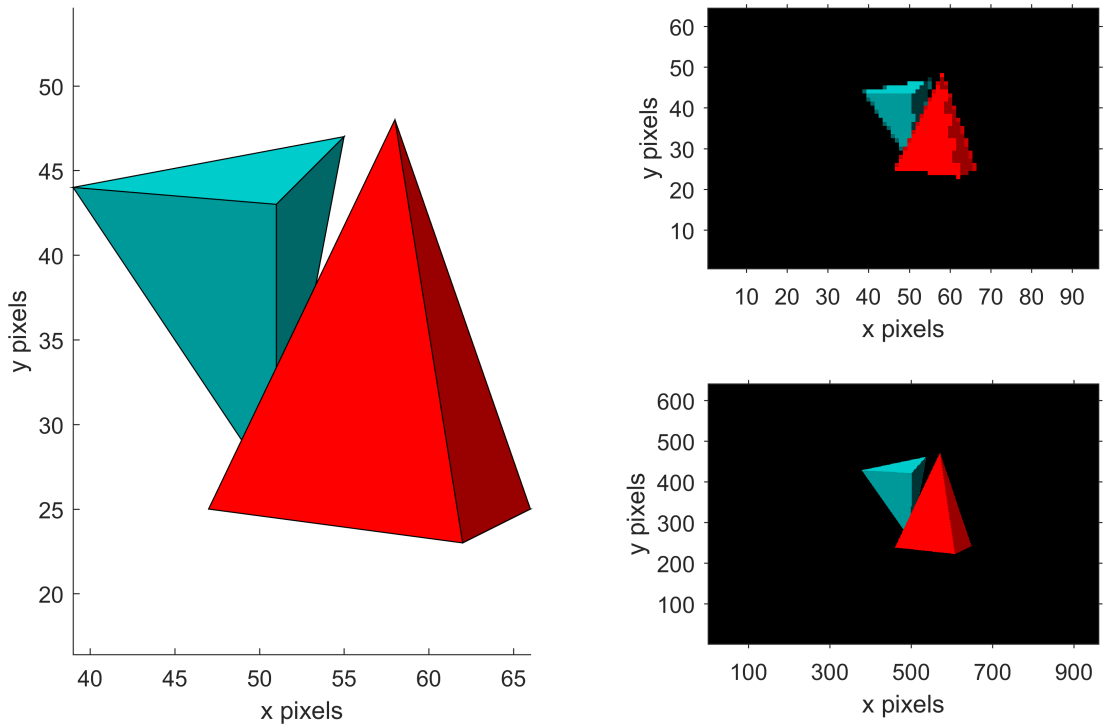

Figure 3: 3D geometry that is used for the benchmarking. The renderings shown above are generated by an identical MATLAB implementation of the graphics pipeline that was developed as a prototype before the embedded version on the KL03. To the left is a zoomed rendering, while the two images on the right show the entire frame with two different resolutions; 96x64 (corresponding to the physical display on figure 1) and 960x640.

Performance versus Baud rate. Next we consider the performance as a function of the specified Baud rate. On figure 2 we see how the frame rate increases as the Baud rate is increased. On the left of figure 2 the FPS measurements are shown with absolute values, while on the right we see relative values. The relative values are interesting because they relate to Amdahl's law for scalability, which is often used to describe speedup in parallel computing, where the speedup is bound by some proportion of sequential, non-parallelisable code. The law states

$$S(s) = \frac{1}{(1-p) + \frac{p}{s}}, \tag{1}$$

where S is the speedup factor of the whole task (the complete rendering), and s is the speedup of a subtask, in this case the increased Baud rate for transferring framebuffers to the display, and lastly p is the proportion of the execution time that originally was associated with this subtask. The value for p was found through fitting in MATLAB, and the fit is shown together with the data points on the relative plot.

Performance for different features. Finally, we consider how the frame rate depends on some of the tasks in the pipeline. The most demanding tasks apart from the rasterisation (which is implemented via the scanline algorithm – see the other two reports or the source code for more information) is the vertex shader and the depth testing (also known as Z-buffering, and referred to here as Z testing). The vertex shader is where each vertex in the geometry undergoes local transformations (translation, rotation and scale) and the transformation from world space to eye space, and finally is projected via a perspective projection matrix into screen space. By pre-computing the transformed vertices, one can omit the vertex shader and go directly to the rasteriser, although the content of course will be static and the pipeline not very useful.

The Z depth testing interpolates the z coordinate for each xy image coordinate in screen space to assess whether the corresponding pixel is occluded or not. This interpolation is overall costly as it happens for each pixel (not vertex), and by omitting the depth testing an increase in performance is expected, although the rendered image may not be accurate (since objects that should be occluded may incorrectly be shown on top).

The vertex shader can be used for local transformations to have some dynamic content for instance, and in the following we shall also consider the inclusion of a 300 degree rotation transformation of the vertices. The transformation is somewhat costly, because it is implemented in a naïve way to save memory – namely a hard-coded rotation matrix corresponding to 10 degrees is applied 30 times in a loop.

On figure 4 we FPS measurements for different cases of inclusion and exclusion of the three features described above. Three measurements of each case is done to give a rough idea of the variance. As expected the maximal performance, about 6 FPS, is when both vertex shader and Z depth testing is disabled. We also see that the depth testing is more demanding than the vertex shader, because the +Vert / -Ztest case performs better than the inverse. The minimal performance is seen when the rotation transformation is performed in the vertex shader, while Z depth testing is enabled.

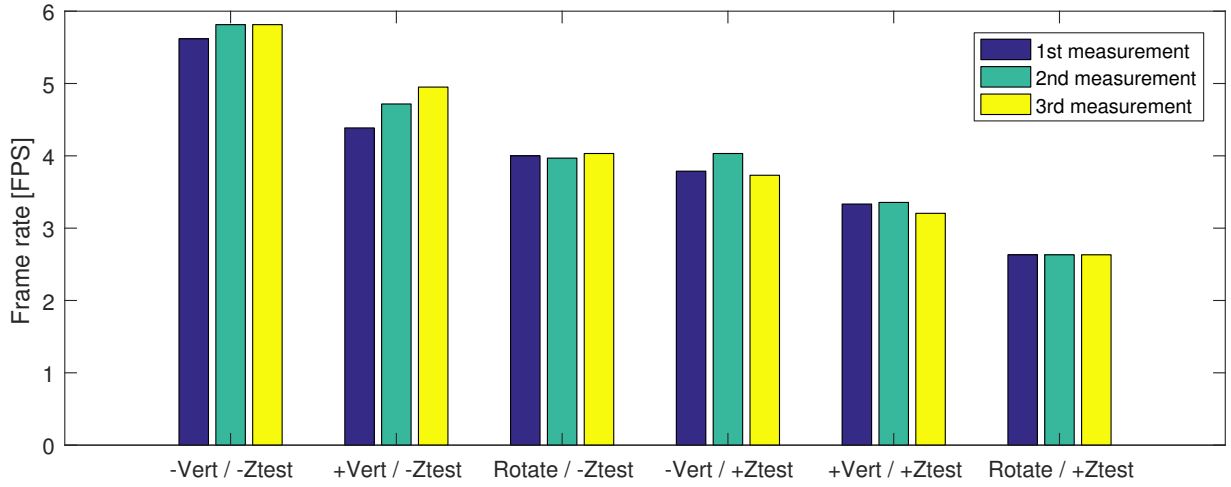

Figure 4: Measurements of frame rate by averaging execution time of 50 renderings under different conditions. The Baud rate was set to 4000 kBd. The labels have the following meanings: `vert` is the vertex shader, `Ztest` is the Z depth test and `rotate` is a rotation transformation performed in the vertex shader (by default the vertex shader only does screen space transformation and perspective projection). The plus and minus symbols in front of `vert` and `Ztest` signifies an inclusion or exclusion in the pipeline, respectively – note that `rotate` implies `+Vert`, since model transformations are carried out in the vertex shader.