donuts!

ECE 3375 Final Project, 2024

James Su, Shiv Patel, Khalid Zabalawi

March 22, 2024

Problem Definition

3D rendering is a common modern computing task, relevant in many different applications from animation to engineering. It consumes a significant number of clock cycles / computing time on generalized hardware, so it is often desirable to offload the task to a secondary, specialized microprocessor. Graphics cards are microcontrollers that are purpose built to handle parallelizable tasks, including graphics rendering. It contains its own microprocessor, the GPU, as well as its own memory and I/O. The DE10-Standard doesn't any specialized parallelization hardware, but we can still run the software on it, albeit slowly.

In this project, we'll build a simple 3D graphics engine on the DE10-Standard. It will show a torus on a monitor using the VGA port, and provide controls to rotate the torus.

Effect on the user

3D rendering lends itself easily to matrix calculations, which very common task in many applications. Offloading parallelized tasks can free up CPU cycles, increasing the efficiency of the device as a whole. Hardware architectures that excel at parallel tasks are also conveniently applicable in other computational tasks like machine learning, scientific computing, cryptocurrency and more.

Real graphics cards provide parallel computing through software tools like CUDA and ROCm. Specialized hardware is also more efficient than general purpose hardware at these tasks, and can be optimized for lower resource settings. This allows the users to decrease and limit their power consumption and overall power requirements.

Overall, generalized hardware is more useful. However, it is limited in certain applications and having a specialized card in personal computers allows for cheaper computers to be better at a wider range of tasks.

Environmentally, specialized hardware optimizes small efficiencies that generalized hardware would waste. Many, if not most forms of scientific computing uses parallelized hardware, thus improving the field and human technology overall. This allows human society to develop new and better methods of taking care of the environment.

Functional Description

When the program starts, a connected monitor will show a rendered torus. The user has the option to interact with the system using buttons and switches. Buttons 0 and 1 will correspond to rotations in the X and Z axis, and switches 0 and 1 will control the direction of these rotations respectively. We only need rotations in two of the 3, as the Y axis rotation just rotates the donut in the plane it is widest in. Functionally, a Y axis rotation should not be noticiable so we exclude it to optimize the rendering process. The 3D torus, in its rotated orientation is projected to a 2D plane, and the x-y coordinates are written to a frame buffer. That frame buffer is then displayed on the connected monitor in real time.

Overall the system maintains the state of a torus, and different inputs manipulate that state by rotating it. Then, the system outputs that state continuously to the display in the form of a projection to a 2D plane. The points on the 2D plane are sent to the frame buffer, which gets forwarded to the pixel buffer and is transformed into a display signal by the VGA DAC.

Input/Output

Currently, our output requirement is to rotate the object, control the direction of the object, and measure rotation speed. To accomplish this, we assigned our inputs to be buttons(actuator), switches(actuator), and a timer(sensor) for this device. Buttons are held to rotate the object, the switches control the direction, and the timer checks the time passed while the button is pressed to control the rotation speed. These inputs manipulate the internal torus state, which is converted to an ouput: a frame buffer.

Our inputs include the buttons and switches on the board itself. However, a more in depth project would probably add a PCI-express port to directly take commands from the generalized processor. The motherboard should contain a PCIe controller, and the device we are making would contain a PCIe interface chip such as the Microchip PCI11400. Drivers have to be made for both the CPU and the GPU, so that they communicate over some specified protocol and transfer required information. A programming language could also be designed for programming GPU drivers in general, like OpenGL, CUDA or RoCM. In our case, we skip the intermediate communication protocol and transfer the information directly from the timer and buttons. When the button is held, the timer is used to measure the time passed since the last render to change the torus state at a consistent rate.

Our ouput consists of a VGA controller, which is the ADV7123 on the DE10-Standard. Many other VGA DAC's exist, although this one in particular is easy to work with as it constantly just reads a pixel buffer, sacrificing versatility for ease of development. VGA controllers are DAC's, and there are many other controllers with more capabilities that are not as easy to use. A more modern solution would use an HDMI or Display port controller, such as the Texas Instruments TDP158. However, an HDMI or DP transceiver would work quite differently than the ADV7123, as there are more compilicated communications protocols to be implemented than just a writable pixel buffer. Unlike VGA, which is a one sided, analog protocol, HDMI and DP are digital protocols that may contain 2-way communication.

Initial Software Design

Initialization

On initialization, we position the camera on the positive z axis, and point it towards the origin with the rendering plane to be somewhere in between. A distance is experimentally found that keeps the torus large but in frame. The torus is initialized in the X-Z plane, such that we could rotate a circle in the X-Y plane and rotate it around the Z axis to form the torus. To avoid having to clear the hardware frame buffer manually, we maintain a separate engine frame buffer. The engine frame buffer is directly written to by the renderer, which is cleared at the start of each cycle. At the end of the cycle, the engine frame buffer is copied to the hardware frame buffer. This significantly increases the frame rate, as it is easier to “erase” a pixel on the next frame using a memory buffer than it is on the hardware pixel buffer for the ADV7123 chip.

Inputs

The buttons and switches are sampled once per render cycle, and the hardware timer is used to find the time passed since the last sample. This allows direct control of the rotation rate, so that it can be a constant value.

Rendering

Since our engine is only rendering one specific object, we can optimize it very specifically. We simply trace points on the surface on the torus, and project them onto the viewplane. A z-buffer is maintained, such that pixels that overwrite a previous pixel are only written if they are closer to the viewer than the previous point. The cycle will likely spend most of the time writing projecting points onto the frame and updating the z-buffer.

Donut Generation

To trace the outside of our donut, we simply sweep theta and phi in 2 planes.

$$(x, y, z) = (R_2 + R_1 \cos\theta, R_1 \sin\theta, -(R_2 + R_1 \cos\theta)\sin\phi)$$

R_2 represents the major radius of the donut, while R_1 is the internal radius of the torus.

This generates a list of points on the surface on the donut, which we can then rotate by applying a quaternion rotation. To rotate a point using quaternions, we convert the x, y, z values to a quaternion $p = 0 + x\hat{\imath} + y\hat{\jmath} + z\hat{k}$.

The desired rotation axis a, b, c and angle θ is expressed as a another quaternion $q = \cos(\theta/2)(0 + a\hat{\imath} + b\hat{\jmath} + c\hat{k})$. To rotate the point, we multiply these quaternions together in the form

$$p' = qpq^{-1}$$

Once all the points are rotated, we then project them onto our camera plane. This is a simple perspective calculation, where K_1 is the distance from the camera to the projection plane, and K_2 is the distance from the camera to the center of the donut.

$$x' = \frac{x * K_1}{K_2 + z}$$

$$y' = \frac{y * K_1}{K_2 + z}$$

We check the z buffer, which is initialized with all zeros. If the z position of the new pixel is closer to the screen than the previous, we overwrite the previous value. Otherwise, we ignore this point on the torus. To streamline this operation, we use the value of z^{-1}, since 0 would be a distance of infinity and any large value corresponds to a closer pixel (unless it's negative).

Once the frame and z buffer are updated, we copy each pixel from the engine frame buffer to the hardware pixel frame buffer. On completion, the frame buffer is reset to all 0's (black).

Prototyping Plan

Prototyping the renderer can be done in software, which we will do in Python. We use numpy and cv2 for a simple display and math tools which are rebuilt later on in C.

The DE10-SoC simulator at https://cpulator.01xz.net/?sys=arm-de1soc contains the same VGA pixel buffer as on the DE10-Standard, and provides an easy way to experiment and test the software on simulated hardware. Once we had fleshed out the rendering engine in python, we would first test the engine on simulated hardware before finally testing it on actual hardware.

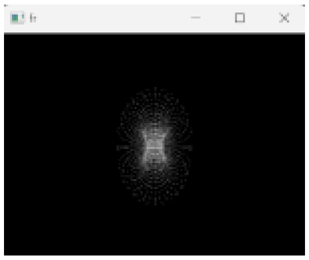

Figure 1: Initialized torus

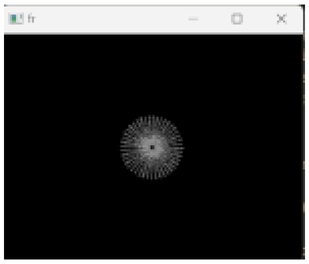

Figure 2: Front view of torus after 90° rotation

Microcontroller

Suitable microcontrollers include the Nvidia Jetson, which contains a Nvidia Tegra GPU, and the Microchip PIC32MZ DA family of microcontrollers. The latter is significantly less powerful, and more suited to 2D rendering while the first is more capable of general floating point matrix calculations. Our highly optimized 3D renderer that only draws a specific donut could probably be run on the latter with good performance, but a more generalized engine would likely require a more powerful graphics processor. One key difference is that the Jetson contains a discrete graphics processor, while the PIC32MZ DA contains an integrated graphics processor. The software would be significantly different on each as descrete processors have their own memory and I/O, while integrated processors share memory and I/O with the CPU.

Due to the complexity in GPU architecture, there are a lot fewer microcontrollers with good parallel compute capability than those without. Embedded graphics processors like AMD's Phoenix line are only embedded into devices like the ROG Ally that are manufactured by large companies that can purchase a large minimum order.

The most widely available suitable microcontroller would then be the Nvidia Jetson line, as they are available as development SoC kits or standalone PCIe cards. The development SoC kits are well suited for our project, as it comes as a complete system with GPIO pins, USB ports, and an HDMI port in one compact system.

Revised Software Design

We came accross two major issues in the software. The first was that the University version of the Intel FPGA Monitor Program allowed limited manipulation of the compiler flags. To use math.h, the `-lm` flag must be appended as the *last* flag to the compiler. However, we could not do that as the Monitor Program added its own flags to the end. As a result, we had to write our own functions for sine and cosine. We built a relatively inaccurate approximation using a Taylor series, mostly because it is good enough for our needs.

We use the taylor series of the sine function up to x^5. This accurately covers $\sin(x)$ from $-\frac{\pi}{2} \rightarrow \frac{\pi}{2}$. To complete the domain of a full unit circle, we "wrap" around values from $\frac{\pi}{2} \rightarrow \pi$ and $-\pi \rightarrow -\frac{\pi}{2}$.

This would not work for cosine, since the range of output values is $0 \to 1$, not $-1 \to 1$ like it is for sine. To workaround this we forward the cosine function to just be the sine function with a quarter rotation offset.

$$taylorsin(x) : x \to x - \frac{x^3}{3!} + \frac{x^5}{5!}$$

$$\sin(x) : \begin{cases} x \to taylorsin(x) & \left(0 \le x \le \frac{\pi}{2}\right) \\ x \to taylorsin(\pi - x) & \left(\frac{\pi}{2} \le x \le \frac{3\pi}{2}\right) \\ x \to taylorsin(x - 2\pi) & \left(\frac{3\pi}{2} \le x \le 2\pi\right) \end{cases}$$

$$cos(x) : x \to sin(x + \pi/2)$$

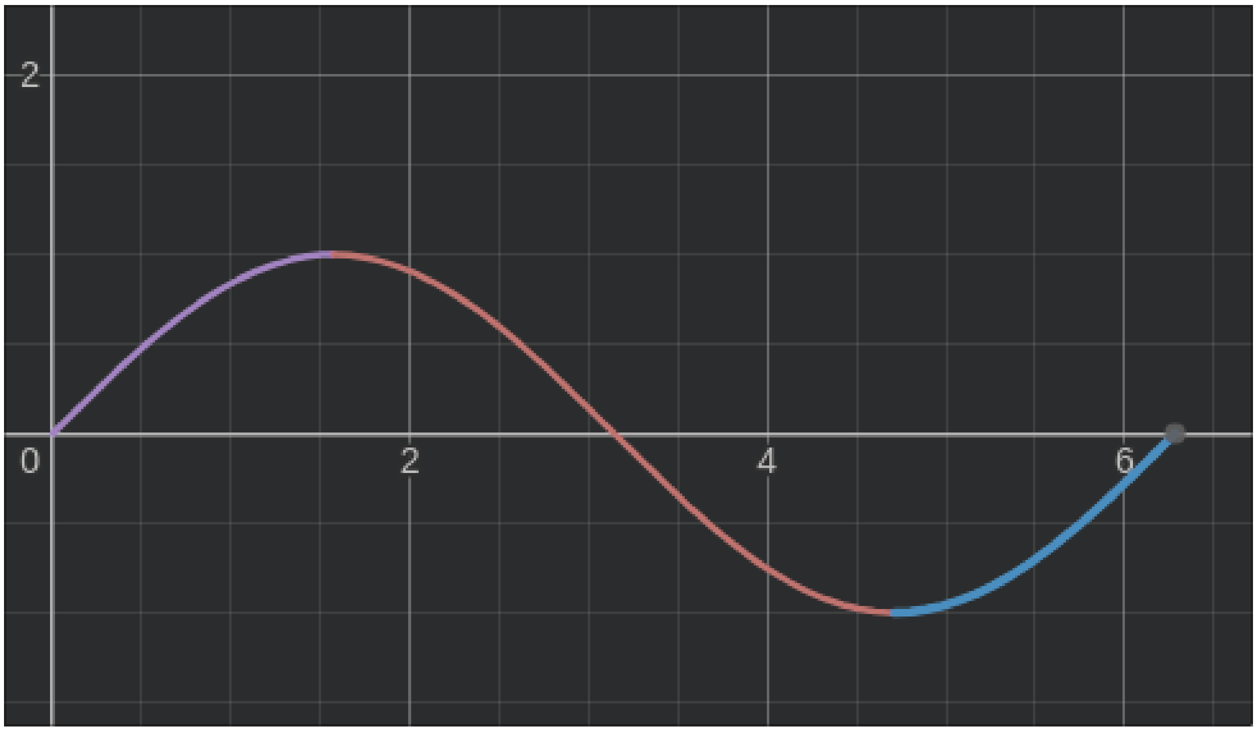

Figure 3: Graph of the Piecewise Sine Approximation

We also ran into issues with our implementation of quaternions, which were not able to be fixed in a reasonable amount of time. We simply avoided this by using euler matricies to compute the rotations instead, which have some flaws but was easier to implement. A 3D rotation in 3 axes α, β, γ on a point p can be computed using the following matrix multiplication.

$$p' = p \begin{bmatrix} \cos\alpha & -\sin\alpha & 0 \\ \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} \cos\beta & 0 & \sin\beta \\ 0 & 1 & 0 \\ -\sin\beta & 0 & \cos\beta \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\gamma & -\sin\gamma \\ 0 & \sin\gamma & \cos\gamma \end{bmatrix}$$

This computation is optimized into a simplified form, although it isn't really readable or clear what is going on so we won't show the equation here. We also exclude the second, y-axis matrix as its effect is negligable while increasing the compute time. The y-axis rotation simply rotates the donut in the flat plane, which shouldn't even be visible if the resolution was high enough. Thus, as an optimization step we just don't compute this rotation at all.

Source Code

- All source code and version history is available at https://github.com/usymmij/boston-cream
- The files are also attached separately to the submission.

Work Division

- Khalid was responsible for the output handling, providing an interface to simplify writing to the output. This included creating the engine buffer, and copying it to the hardware pixel buffer.
- Shiv was responsible for the input handling, converting the button presses into rotation vectors.
- James was responsible for the rendering of the donut, and its rotation based off of the input vector.