Poor Man's VR: Raymarching with Stereoscopic Offset and Gyroscopic Control
Final Report

Marco Andrade
M.I.T.
77 Massachusetts Avenue,
Cambridge, MA 02139
marcoand@mit.edu

Aidan Zev Blum Levine
M.I.T.
77 Massachusetts Avenue,
Cambridge, MA 02139
azb@mit.edu

Benjamin Hunsberger
M.I.T.
77 Massachusetts Avenue,
Cambridge, MA 02139
bhuns19@mit.edu

***Abstract*—An implementation of the raymarching rendering technique in hardware to produce three-dimensional visuals from signed distance functions. The scenes may be explored and manipulated through a gyroscope, allowing for a real-time change in viewing angle. A stereoscopic offset will be produced on the scenes via rendering in blue and red from different positions that reproduces human depth perception when paired with blue-red 3D glasses. The intended effect of these techniques is a 'poor man's virtual reality,' as the three-dimensional scenes will appear tangible beyond the screen and may be manipulated in the physical world.**

I. Component Overview

At a high level, our project consists of a scene, two raymarching renderers that render the scene from two eye positions, and a gyroscope that controls our view direction. With these parts combined, you can view our scene with the illusion of 3D when wearing red-blue glasses. However, implementing this project involves overcoming many technical challenges, including making a renderer, signed distance functions, and gyroscope angle control, all purely on hardware. Our current block diagram for our stereoscopic renderer is shown in Fig. 10. Below, we explain our latest version of each of these components and then show the progress we have so far.

II. Raymarcher

Raymarching is a rendering technique that involves marching a ray through a signed distance function (SDF). To compute the color at a pixel, the raymarcher calculates the ray from the camera in the direction of the pixel and marches it through space until it hits the surface of the SDF or has traveled too far and is counted as having missed the target object. An illustration of how this works is shown in Fig. 1, where the circle's radius is the SDF value at its center, and the ray continues moving forward by that SDF value until it reaches the surface. For more details on the raymarching algorithm, see this blog. Our raymarcher is implemented as a finite state machine, which is shown below in Fig. 2. After receiving a pixel location, it normalizes the ray to that pixel and then switches between an awaiting SDF state and a marching state, until one of the end conditions is reached.

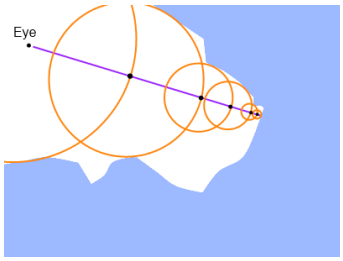


Fig. 1: Raymarching visualization from 9bitscience blog

III. Renderer

Our renderer module contains a raymarcher and a framebuffer. As pixels are computed by the raymarcher, they are saved to the frame buffer. Right now, we are rendering in full 24-bit color for easier debugging, but will later switch to each renderer only storing one color (red or blue), reducing the size of a framebuffer entry to 16 bits. We are also planning on reducing the number of bits per color as far as we can without visibly hurting image quality, to allow for larger images to be rendered.

At the same time that the raymarcher is rendering pixels, the framebuffer is being independently read by the HDMI pipeline from lab 4. This allows rendering to occur at a speed completely independent of the monitor refresh rate.

A. Lighting

After a ray has hit a surface, we need to compute the color for that point. To create semi-realistic lighting, we first need to calculate the normal of the surface at that point. This can be easily computed by sampling our SDF at three more points near our point of contact. By finding the change in depth corresponding to moving a small amount along the x, y, and z axes away from the point of impact, we recover the x, y, and z components of an approximated normal to the surface following the formula below.

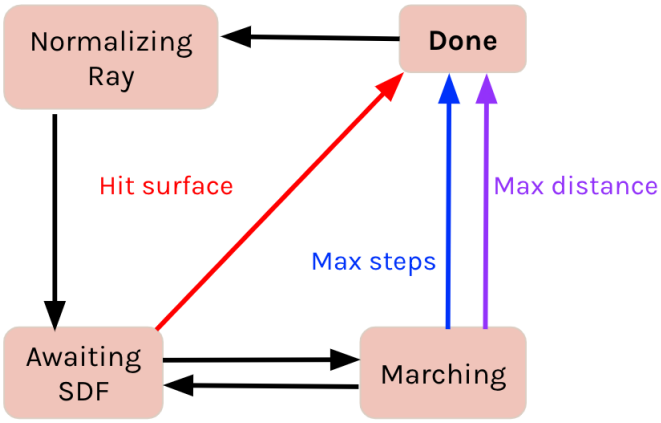


Fig. 2: Raymarcher module state machine

$$normal = normalize(\begin{bmatrix} sdf(x+\epsilon, y, z) - sdf(x, y, z) \\ sdf(x, y+\epsilon, z) - sdf(x, y, z) \\ sdf(x, y, z+\epsilon) - sdf(x, y, z) \end{bmatrix})$$

Once this normal has been calculated, we compute the dot product of that normal with the direction to each light, and sum over all the lights. This gives a good estimate of how much diffuse lighting this point would receive, allowing you to appreciate the 3D nature of the shapes.

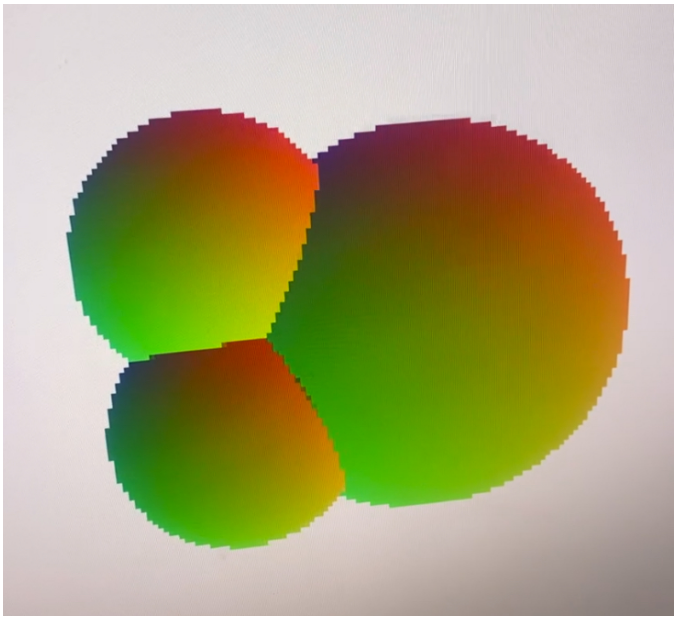


Fig. 3: Demonstration of normals being rendered on three spheres

B. Stereoscopic Effect

We are rendering two versions of our scene from two nearby points to create the illusion of depth when viewed through red-blue glasses. Finding the points to render at involves finding the forward view direction from the gyroscope, as well as the right and up vector. With these three vectors, we can compute the two eye locations by adding the right vector to the origin for the right eye and subtracting it for the left eye. From here, we produce two mono-color renderings from those eye positions onto the plane produced by the right and up vectors. We make one of these mono-color renderings the red and of them the blue value of the output image, completing our illusion. We found best results in practice by setting the overlap of the red and blue renderings to be white (by making the green channel the minimum of the red and blue channels), displaying the white body of an object as the area seen by both eyes, with blue and red ranges on either side of the object representing the perspective from either eye.

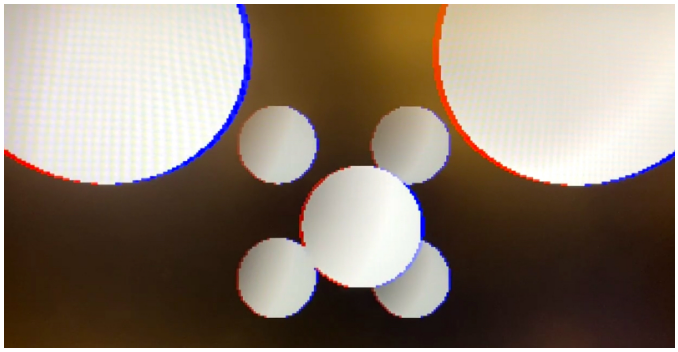


Fig. 4: Field of stereoscopically offset spheres at varying distances

The stereoscopic offset relies on a specific distance from the width of the eyes, and as such the user must be a specific distance from the monitor depending on its size for the 3D effect to fully work. In combination with proper lighting, the stereoscopic effect works quite well.

IV. Gyroscope

From a high level, the functionality we are aiming for from our gyroscope is being able to adjust the view angle of the 3D rendering created by the Raymarcher by moving it around. This involves wiring the Inertial Measurement Unit (IMU) to the FPGA, capturing and decoding the data received from the IMU in the I2C format, processing the data to be in a useful format, and then using the data to calculate the view angle for both views.

A. IMU

The IMU we are using is the MPU-6050. This IMU is capable of returning the angular velocity around the x, y, and z-axis as well as acceleration in the x, y, and z-direction. For our purposes as specified above, we will only be capturing the angular velocity. In order to connect the IMU to the FPGA, we are using a breadboard as an intermediary to facilitate connection. With the breadboard withheld, the connections are seen in Fig. 5. The VCC and GND connections serve to provide power to the IMU. The SDA and SCL connections are the data and clock, respectively, for the I2C standard.

Additionally, there are two $4.7k\Omega$ resistors being used as pull-up resistors on the SCL and SDA line. In the breadboard,

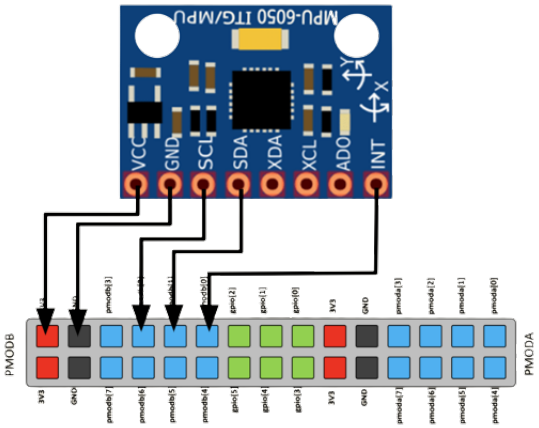


Fig. 5: IMU to FPGA Connections

B. MPU-6050 Verilog Module

With the MPU-6050 connected to the FPGA, we now need to decode the information that is being passed through in the I2C standard. In order to this, we utilized a package developed by Daniel Moran in VHDL [4] and later translated into Verilog by Alon Levy [5] From this module, passing through pmodb[1] and pmodb[2], which contain the I2C data and clock, respectively, the system reset, and an enable flag, we are able to retrieve gx, gy, and gz which are the angular velocity around the x, y, and z axis in degrees per second as a 16 bit signed integer.

C. Normalization

With the rotational velocity along all three axis received, we can move on to processing this data so that it is returned in a useful format for the raymarcher module. First, we need to deal with some of the drift that occurs on the gyroscope. In order to accomplish this, we perform a normalization process. When button 2 is pressed, the values from the gyroscope are summed for 2^{20} clock cycles. Then, the summed values are divided by 2^{20}. The value of the gx, gy, and gz is then subtracted by this amount to compensate for drift. Also, after the subtraction of the average value, the values of gx, gy, and gz are shifted to the left by 7, in order to constrain the change between $-512°$and $512°$. This ensures that the values being output by the gyroscope are manageable.

D. Process-Gyro Module

This module handles the conversion of the normalized and scaled gx, gy, and gz values to actual yaw, roll, and pitch values. This module assumes a start of yaw, pitch, and roll of 0, 0,0. Then, the module is constantly summing the values and saving them. Then, every 10000000 cycles (10 times a second), the module divides the value by 100000000 (which is a multiplication by 43 in fixed point). After adjustments for the angle positioning, the new values are added to the stored pitch, yaw, and roll. Then, the running sum is reset and it starts again.

E. Sine Module

In order to perform the calculations required to then calculate the view angles using the yaw, pitch, and roll, we required a fixed-point sine module. In order to accomplish this efficiently, we created a lookup table module, which contains 180 values of sine in 16.16 fixed point. Additionally, we created a sine module and a cosine module, which can handle inputs from 0°to 360 °and will return the appropriate signed 16.16 fixed point number.

F. View Output

The final module involved in processing the view direction is the view output module, which takes yaw, pitch, and roll and calculates an up vector, forward vector, and right vector. This is done by using the following equations:

$$\vec{F} = \begin{bmatrix} \cos(\text{pitch}) \cdot \sin(\text{yaw}) \\ -\sin(\text{pitch}) \\ \cos(\text{pitch}) * \cos(\text{yaw}) \end{bmatrix}$$

$$\vec{U} = \begin{bmatrix} \sin(\text{pitch}) \cdot \sin(\text{yaw}) \\ \cos(\text{pitch}) \\ \sin(\text{pitch}) * \cos(\text{yaw}) \end{bmatrix}$$

$$\vec{R} = \begin{bmatrix} \cos(\text{yaw}) \\ 0 \\ -\sin(\text{yaw})) \end{bmatrix}$$

The main difficulty we encountered when writing this portion was that we were running out of DSPs attempting to instantiate a sine or cosine module (as described in the previous section) for each vector calculation. To solve this, we ended up using just two sine functions, and making an FSM with 29 states. These states were mostly repeated but involved: 1 high level-state for vector (IDLE, UP, FORWARD, RIGHT, FINISHED), another medium level-state for direction (X, Y, Z), and a final low-level state for which portion of the sine calculation we were on (UPDATING_VALUES, CALCULATING, FINALIZED).

V. Signed Distance Functions

Signed distance functions will be used in our implementation to take in a point in space and determine the minimum distance that point is from the surface of an object. If this distance is negative, the point is inside the object. Different functions may be used to render different scenes and shapes, and the distance returned by the functions helps us to determine what color and shading to apply to a point when raymarching.

A. Menger Sponge

The SDF we have chosen to use as a goal point for this project is that of a Menger Sponge. The Menger Sponge is a fractal that has enough complexity to create an interesting scene that captures the ability of raymarching, while not being too heavy in calculations to be infeasible for our system, having only one square root and minimal multiplications of large numbers. The approach to performing the calculations for this SDF in appropriate timing is that of a state machine. This implementation removes some of the complexity down the line of debugging the calculation module implementations and timing errors. The state machine for this follows a changeable number of iterations to select how complex the sponge will be and includes many brief stages for the process of gathering signed minimums and calculating the square root. Vector calculations leverage the *struct* feature of System Verilog, storing x, y, and z values within each vector. A series of helper functions to find the absolute value of a vector and to subtract two vectors create a more readable and easy-to-modify state machine. The result is a complex scene that is well-tailored to a 3-dimensional viewing environment.

The test benching for the Menger Sponge was accomplished by creating a working Python version of a Menger Sponge SDF using simpler and more reliable tools and then comparing this output to that of our System Verilog SDF.e

B. Shape Fields

While impressive, the Menger Sponge does a poor job of capturing the lighting and depth of the system, and as such we produced more SDFs demonstrating fields of differing shapes and planes. The rounded surfaces strongly demonstrates the 3-dimensional lighting effects, while a series of overlapping planes spanning back in the SDF demonstrate this dimension through depth and distance. In what was a failed attempt at creating a cone SDF, we made an interesting effect that produced a series of planes stretching back behind any objects in the foreground. When an object was centered over these receding planes, they would appear to bend to the outline of the shape in front of them. We abandoned creating the cone and continued experimenting with this effect. Beyond this, we

VI. Fixed-Point Number Representation

We are using fixed point numbers for greatly reduced operation complexity compared to floating point numbers. We were originally using an 8.16 representation but started running into some issues when calculating the norms of vectors. This makes sense since we have to store the squared magnitude of a vector to normalize it. Since that representation can hold a maximum value of 128, we can only normalize a vector with a length of slightly over 10, which is not enough for our scenes. To reduce the risk of precision issues or overflow like this occurring, we upgraded to a 16.16 representation.

In hindsight, this led to lots of waste in our system, and the best option if we had more time would be to vary the representation as needed, based on which calculations are currently being run.

VII. Final Functionality

A. SDF Rendering

Our fractal raymarching is able to successfully render a Menger Sponge as intended, with the scene being viewable in three dimensions by a shifting view angle input. The sides are given distinct colors based on the normal. We found a sponge of an iteration depth of 4 to be of optimal complexity, as it was able to run smoothly while producing an object of impressive complexity.

The frame rate depends on the scene being rendered, varying from 15 on simple scenes with multiple renderer units (which will be elaborated on below), to around 1 frame per second for expensive scenes that fill the screen. The frame rate depends not just on the scene, but also how many pixels are filled by the object, since filled pixels are much more expensive to render than the white background, since shading requires additional calculations.

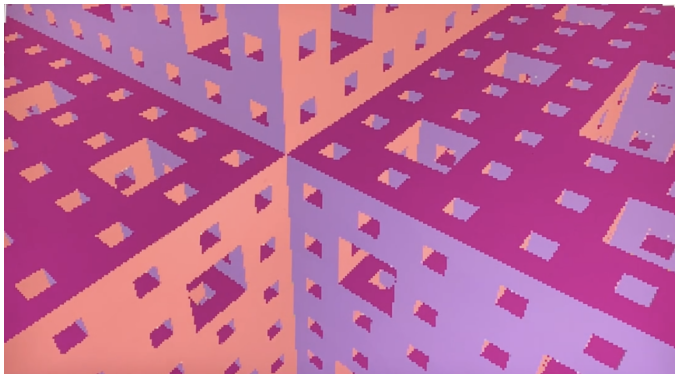
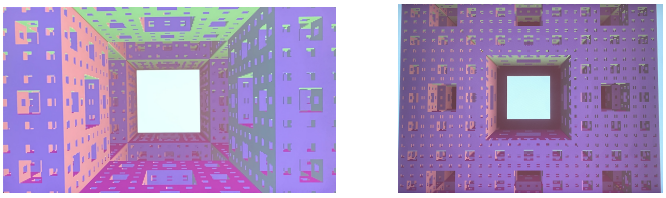


Fig. 6: Inner corner of the Menger Sponge



(a) Looking in from the surface (b) Viewing the outer surface

Fig. 7: The face of the Menger Sponge

Following further experimentation with unique SDFs, we create a few unique demonstrations of depth with various shapes in our 3D space.

B. Limitations

All of our significant limitations in our renderer are caused by the number of DSP's available on our board. These DSP's came from all of the math functions (primarily square root) that were needed to render our scenes. Though this did not prevent us from rendering the scenes we created, it heavily limited our frame rate by limiting us to only having one or two raymarching units per scene.

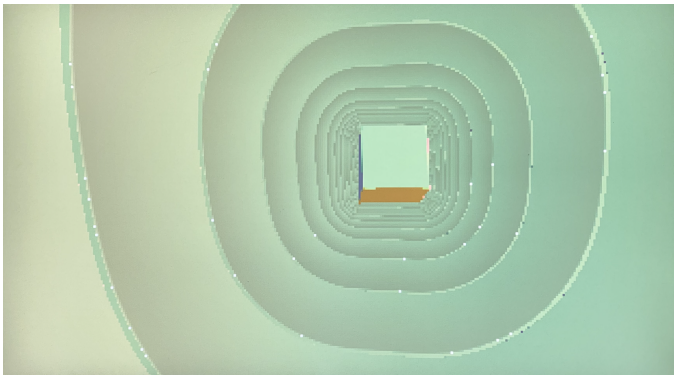


Fig. 8: A unique SDF of a cube in an infinite space

The simplest way to increase the frame rate for our renderer was to increase the number of raymarchers rendering the scene at a time. This was a trivial improvement, but we could never fit more than three units on our board even with a single static sphere as our scene. While working on this part of our project, we observed very inconsistent DSP usage as we increased our complexity. For example, a moving sphere scene with one raymarcher used only 14% of our DSP's, but with two raymarcher used 85% of our DSP's, even though it should've increased usage by somewhere between 50-90%. Furthermore, the build script took a long time to run for projects that use near the board's maximum capacity, and often crashes near the end. This means it often takes an hour to attempt a build just to see that it wouldn't fit on the board or broke timing requirements.

To attempt to improve our board usage, we tried to replace our most expensive operations with simpler approximations. The most expensive part of raymarching is normalizing rays, since it requires one square root and one division module. As an alternative, we wrote an approximate inverse square root module that used a lookup table to find a guess and then ran a few iterations of Newton's method. If this module worked, it could replace all square root and division modules in our system. This module generally worked when testbenching, but struggled with overflow on larger numbers since it required raising the input to the third power. This overflow was fixable by resizing our number representation or shrinking our scene, but we also experienced larger errors. More fatally, it used up even more space on the board than our previous version. This space usage did not make sense, but given the build difficulties mentioned above, we decided to stick with what we had working and not to use this module for our project.

C. Visual banding

Another minor issue with our project is that frames are visible while changing, causing banding from top to bottom. There is an easy solution to this problem, but we do not have enough RAM on our board to implement it. The easy solution is to create a double buffer, where the calculated frame is copied over to the rendering buffer when it is complete. On a board with more RAM, this would look much smoother.

D. Gyroscopic Viewing

The Gyroscope module is currently fully capable of tracking the pitch, roll, and yaw. Additionally, the processing module is capable of converting the pitch, roll, and yaw into a right, left, and up angle. This is integrated into the view renderer, and effectively view tracking is enabled. There are a few limitations with the actual IMU itself. There is a large amount of drift that the IMU has. This can be somewhat reduced by using the "normalization" feature, which averages the gx, gy, and gz over a period of time and then uses that value to subtract from the respective gyroscopic outputs. This is not exhaustive because there are other fluctuations in the value reported by the IMU even when still. One way that we could have solved this issue further is using a complementary filter, which would use the acceleration data from the IMU to help verify the gyroscopic output. However, we are limited by the computational power of the board, as mentioned previously, because a filter that would improve the data effectively would require many mathematical operations.

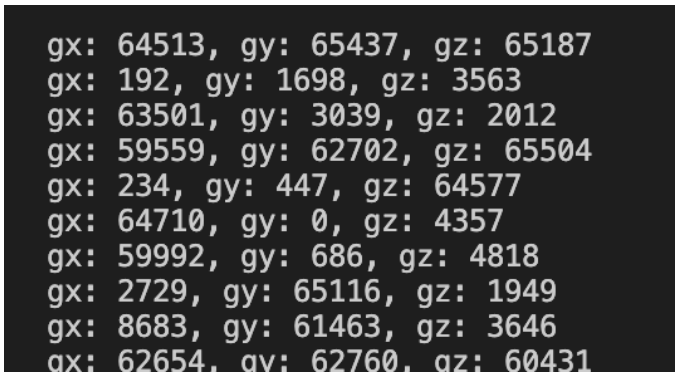


Fig. 9: Sample Raw Output from Gyroscope

E. Combined Features

Unfortunately, given the limited power of our FPGA, we were unable to allow all features - the fractal signed distance function, the gyroscopic viewing, and the stereoscopic effect - to be rendered in unison. We strongly believe that should we have had a larger board, all pieces would have functioned together properly but were met with limitations of available DSPs. The primary case where we could not combine all of these parts was that the stereoscopic effect involved doubling from one to two renderer units, which did not fit with our more complex scenes. However, this was not a big issue, since the stereoscopic effect looks the best on simpler scenes (not fractals), where your brain can easily interpret what it is seeing.

VIII. Contributions

A. Aidan Blum Levine

Aidan worked on the core renderer and raymarching modules, including implementing the math and algorithms. This included writing and testbenching many fixed-point modules, and debugging many timing issues. Aidan also experimented

with our stereoscopic offset and parallelizing by adding more raymarching units.

B. Ben Hunsberger

Ben worked primarily on the signed distance functions, and integrating them into the raymarcher. The Menger Sponge took the longest, with an extensive process of test-benching the state machine. Beyond this, he worked on other SDFs that would better capture the additional capabilities of the raymarcher by demonstrating depth and distance in fields of shapes.

C. Marco Andrade

Marco worked on the sourcing, wiring, decoding, and processing of the gyroscope. This involved debugging the gyroscope physically with an oscilloscope as well as writing the code to turn the raw data into usable view angles by the renderer. To support this, Marco wrote test benches and used manta to interface with the gyroscope's raw output as it was decoded. Finally, Marco worked to debug resource issues when integrating the gyroscope (and respective processing) into the system as a whole.

IX. View our code

https://github.com/AidanBlumLevine/6111-Final-Project

References

[1] https://projectf.io/posts/square-root-in-verilog/
[2] Inigo Quilez, menger fractal - 2011. https://iquilezles.org/articles/menger/
[3] Ultra cheap exact Menger sponge. https://www.shadertoy.com/view/sdSBWc
[4] MPU 6050 VHDL https://github.com/danomora/mpu6050-vhdl
[5] MPU 6050 Verilog https://git.sr.ht/~beepbeep/mpu6050-ulx3s
[6] Raymarching blog https://9bitscience.blogspot.com/2013/07/raymarching-distance-fields_14.html

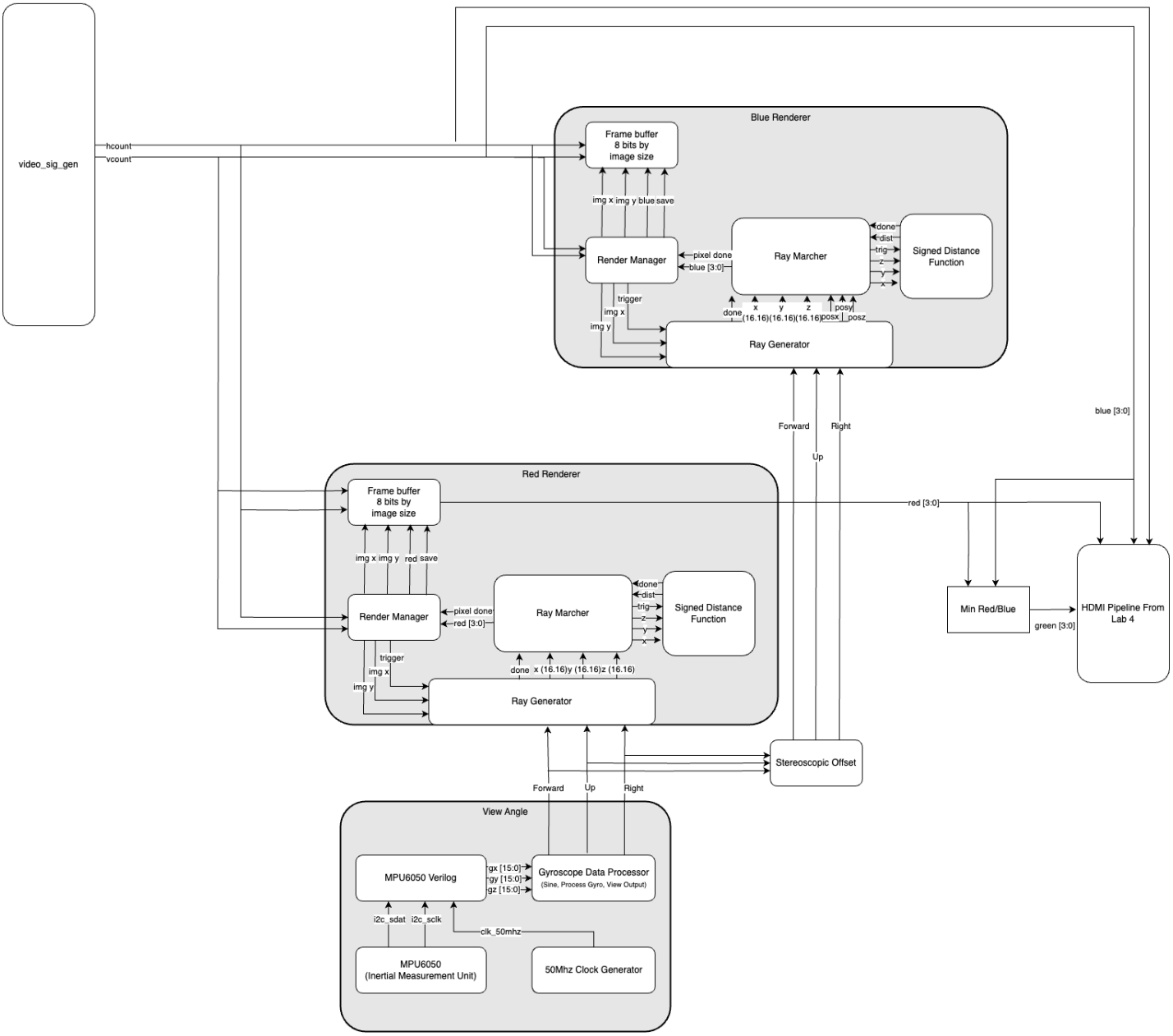


Fig. 10: Block diagram for our stereoscopic version