Flap Through the Gap

Preliminary Report

Andrew Lee
Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
Cambridge, MA
andrewl2@mit.edu

Selena Qiao
Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
Cambridge, MA
scq@mit.edu

Magdalena Slowikowski
Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
Cambridge, MA
maggislo@mit.edu

***Abstract*—*Flap Through the Gap* is a first-person flight simulator, where the player navigates through a tunnel containing a series of walls with holes. The objective of the game is simply to fly through as many of these obstacles as possible without colliding with the walls. The player controls movement with a physical controller, which uses a gyroscope and an accelerometer sensor to detect tilting for directional control. The game is rendered from the perspective of the player with a rasterization engine optimized to handle the specific layout of the game. The output of the graphics engine is then displayed on an external monitor via HDMI.**

I. Introduction

In this paper, we detail our design and implementation of *Flap Through the Gap*, a first-person flight simulator game where the player encounters a series of walls with holes while flying through a tunnel. Using a 3D-printed game controller, the player is able to control their position in the game with the objective of passing through holes without colliding with any walls in the tunnel. In our system, we interface with an accelerometer and gyroscope sensor using an SPI communication protocol to detect tilting of the game controller, render the layout of the game from the changing perspective of the player with a rasterization engine, write game logic to detect player-wall collisions, and display the output of the graphics engine onto an external monitor via HDMI. As in the proposed project checklist, we aimed to create a game with color, where the player's tilt movements are smooth and not erratic,

II. Calculations

A. Integer Arithmetic

In our system, we work mainly with integers to avoid floating-point arithmetic. When multiplying integers by fractional values, we multiply by the numerator and then shift to divide by the denominator.

B. Trigonometric Functions

Many parts of our system require the use of trigonometric functions. This includes arctangent in the gyroscope/accelerometer input handler's calculations, and sine and cosine when handling rotations and perspective transformations in the game logic and the rasterization engine. To approximate these functions, we compute the values in advance using a Python script and store them in lookup tables stored in BRAM. For example, our sine lookup table has a depth of 256 and a width of 8 bits, and the ith index stores $256 \cdot \sin(\frac{i}{256} \cdot \frac{\pi}{2})$. When using a value retrieved from this BRAM, we eventually shift the result by 8 bits to scale it back to the original range.

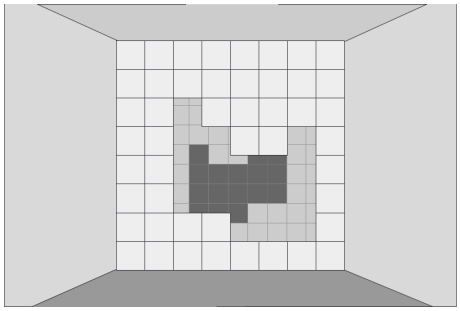

Fig. 1. A mockup of a singular frame from the final game. The lines dividing each wall into an 8x8 grid are purely illustrative and will likely not be included in the final render.

C. Combinational Logic

Many of our complex calculations in all parts of our system are implemented combinationally. However, these calculations cannot be completed within a single clock cycle. Instead of pipelining these computations, we empirically determine an upper bound for the number of cycles required for the combinational logic to stabilize. Only after we wait the computed number of cycles do we pass the results to the next stage.

III. Peripheral Hardware

In our system, player movement is determined by a controller with an MPU-9250 sensor attached. The sensor records gyroscope and accelerometer data, which are combined to calculate the player's perspective and acceleration in the game logic module.

A. SPI Controller

To interface with the MPU-9250, our system uses an SPI module. To initiate a read operation, the SPI controller transmits the relevant register address. The MPU-9250 then responds with the respective register data, read linearly starting

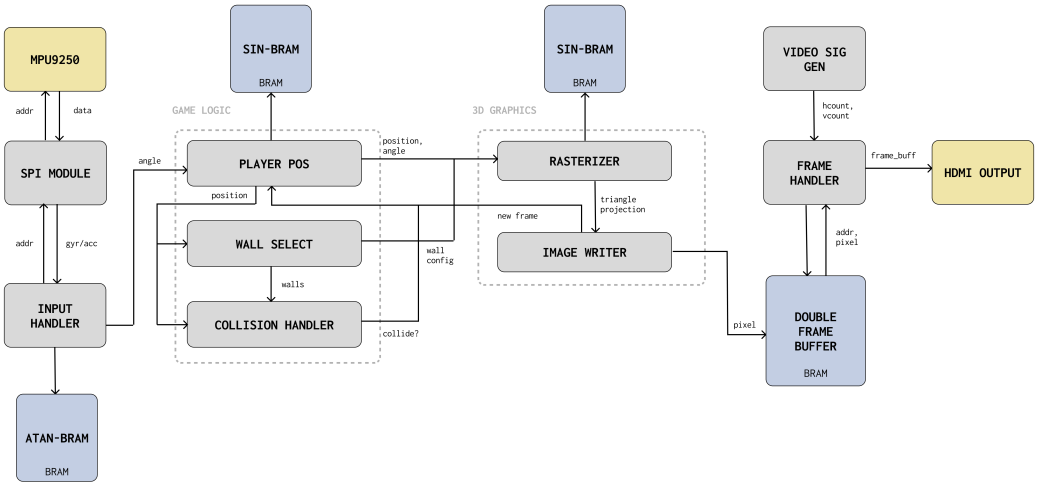

Fig. 2. Block diagram for the system.

SPI Address format

MSB							LSB
R/W	A6	A5	A4	A3	A2	A1	A0

SPI Data format

MSB							LSB
D7	D6	D5	D4	D3	D2	D1	D0

Fig. 3. Format of SPI communications with the MPU-9250

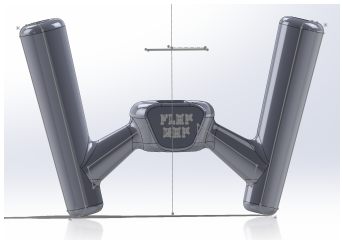

Fig. 4. SolidWorks Game Controller

at the given address. Each measurement register contains 16 bits, however, because our system does not require the high sensitivity provided by all 16 bits, we utilize only the first byte of data. Therefore, our SPI controller writes 8 bits of the register address and then reads 8 bits of measurement data.

The SPI controller iterates through all of the sensor's output registers and constructs the accelerometer and gyroscope measurements for each axis. These measurements are then forwarded to the input handler module where they are used to calculate the output roll, pitch, and yaw.

B. Input Handler

It has been found that the raw accelerometer data read from the sensor often exhibits high-frequency noise, while the gyroscope data contains low-frequency noise. To reduce the noise in the input measurements, the input handler module implements a complementary filter to merge accelerometer and gyroscope signals:

$$\theta_y[n] = \beta\left(\theta_y[n-1] + Tg_y[n-1]\right) + (1-\beta)\tan^{-1}\left(\frac{a_z[n-1]}{a_x[n-1]}\right)$$

where β is the filter coefficient, T is one time step, g_y is a measurement of the gyroscope in the y-axis, and a_x, a_z are accelerometer measurements in x- and z-axes respectively. Doing so, we are able to better model the player's direction of movement.

In order to perform these calculations, we utilize a divider module to divide accelerometer measurements, and look up arctangent values in BRAM, as stated previously.

The filter operates on each axis in parallel, and the output roll, pitch, and yaw from each respective instantiation are forwarded to the game logic module.

C. Game Controller

For player ease and enjoyment, we designed a custom game controller using the SolidWorks CAD software. The controller design is inspired by a yoke, or airplane "steering wheel". The MPU-9250 sensor fits inside the game controller, allowing for ease of direction and tilt control.

IV. Game Logic

A. Position and Angle Updates

Given roll, pitch, and yaw data from the gyroscope input handler, the game logic computes the new position and angle of the player. Additionally, the game logic only accepts input and begins computations only when the image writer has finished writing the results from the previous calculations to the double frame buffer. To smooth out sudden changes in angle, we compute a weighted sum of the current angle and the new angle. The direction of movement is determined using values derived from the rotation matrix (described later). Then,

Fig. 5. Printed Game Controller

the game logic adds the product of the speed and this unit vector to the current position. Positions are stored as 16-bit fixed point values, where the top 8 bits represent the integer game units and the bottom 8 bits represent the fractional part. Only the top 8 bits will be sent to the rasterization engine, and the fractional parts are only used to achieve more accurate movement.

B. Collision Detection

Detection of collisions is relatively straightforward. The new position is checked to see whether it is within the tunnel boundaries. Additionally, it is checked that the player does not collide with the nearest wall. These checks are performed using simple inequalities.

C. Wall Management

The game logic maintains a set of the next five upcoming walls, each of of which is described by a subset of the 64 blocks (32 by 32 by 32 game units) that fill a 256 by 256 by 32 game unit area. The walls are spaced by 256 game units and whenever the player's position advances 256 game units into the tunnel, a new wall is generated. Walls are randomly selected from a predefined set of configurations using randomness derived from the video signal generator.

V. Rasterization Engine

To render the game's graphics, we implemented a version of the rasterization engine optimized for our specific input. The graphics pipeline is largely split into two sections: the rasterizer, and the image writer. The main functionality of the rasterizer is to break the game map down into its constituent triangles and compute their projection and color. Because the layout of the game's map is largely cubical, we were able to make some simplifications in some steps of the algorithm, which we will detail later. The output of this is then passed to the image writer, which identifies which pixels on the screen each triangle occupies and writes their color to an image buffer. Once a buffer is complete, it is then sent to an external monitor with HDMI.

A. Rasterizer

The rasterizer's first step is to generate the triangles that make up the game map. The game logic module passes a wall configuration to the rasterizer, consisting of a packed array of bits representing whether each cell in an 8×8 wall grid contains a wall or a hole. Given this configuration, the rasterizer iterates through each cell in the wall, starting from the wall in farthest back. It splits each face of the $32 \times 32 \times 32$ cube down a diagonal into two right triangles. The same is done for the walls of the tunnels, which are of roughly fixed position in the game world.

Typically, rasterization algorithms make use of a z-buffer to keep track of the depth of triangles in world space in order to determine which triangles are positioned in front of others, and therefore which colors to write to the final image. However, because the configuration of the walls in this game is very well-defined, we were able to optimize the rasterizer by removing the z-buffer. Instead, in order to guarantee that the correct triangles are written to the image buffer, our algorithm processes the triangles in a very specific ordering:

1) Tunnel walls are divided into six segments. Divisions are drawn where obstacle walls are placed. Each segment of the tunnel wall must be processed before the obstacle wall that it is behind.
2) Obstacle walls are processed from back to front, in between the tunnel segments that it divides.
3) The non-forward-facing sides of *all blocks* in a singular obstacle wall are processed first.
4) Then, the forward-facing sides of an obstacle wall are processed.

By generating and processing triangles in this order, we hoped to guarantee that the triangles in the back are processed before any triangles that are positioned in front of them in world space. If successful, the need for a z-buffer is removed. This is a huge optimization for memory usage – a z-buffer would have needed to use as much memory as our image buffer itself, which would have approximately doubled the memory usage for the rasterizer.

However, we realized after implementing this algorithm that it sometimes did not properly account for how sides of a cube are viewed relative to each other. For example, it is not guaranteed whether the top of the cube or the left side; the order in which the sides are seen depend on the perspective of the player. The top of a cube may appear on top of the left side if the player is viewing the cube from higher up, but if viewed from another perspective the left side will need to be rendered first. The result of this is some visible artifacts on the walls in the game, where some interior sides are visible, like an x-ray effect.

The rasterizer then computes the color for each of these triangles by computing the dot product between a global directional light vector and the normal vector of each triangle:

$$\text{color} = \mathbf{n_{triangle}} \cdot \mathbf{d}$$

The configuration of the game map again allowed for significant optimizations in the light calculation – because

there are only five possible normals, instead of computing the normal of each triangle by taking the cross product of two vectors, we simply select the correct normal when generating the triangle itself.

Additionally, we add some fog, make further objects appear darker by darkening a triangle's color by a constant multiple of it's depth. The resulting color is then manipulated to give the game a wash of blue.

Finally, the rasterizer converts the coordinates of the triangles from world space into camera space based on the rotation and position of the player passed from the game logic module, and projects these triangles onto a 2D image plane.

To convert the coordinates from world space to camera space, the rasterizer first translates the triangle's coordinates by the current player's position coordinates. Then, the rasterizer computes a rotation matrix from the roll, pitch, and yaw received from the game logic module by taking the product of the corresponding matrices, where roll is ϕ, pitch is θ, and yaw is ψ.

Combining this translation and rotation, we get the transformation:

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} \rightarrow R \begin{bmatrix} x - x_{\text{pos}} \\ y - y_{\text{pos}} \\ z - z_{\text{pos}} \end{bmatrix}$$

After transformation, the coordinates are projected down to the xz-plane by dividing both the x and z coordinates by y and then multiplying by 320 and 180, respectively. In the case of a negative y value, we approximate the result by having the division $\frac{x}{y}$ return a large number with the same sign as x and having the division $\frac{z}{y}$ return a large number with the same sign as z. This just represents a point way off screen in the approximate direction of (x, z). Lastly, we translate by (160, 90) to set (0,0) to be the bottom left of the screen.

These computations require use of the integer division and trigonometric LUTs described earlier. After all computations for one triangle are complete, the triangle is forwarded to the image writer module.

B. Image Writer

The image writer takes the projected triangles that the rasterizer inputs into the system and determines the corresponding pixels to color in. To store the pixels, the module makes use of a double buffer. One buffer is used to store the pixels as they are being colored, and the other buffer is read by the HDMI module (and cleared in the process); once both buffers have completed their function, they swap functions.

To determine the pixels that correspond to a triangle, the module first computes the bounding box for the triangle, then iterates through all the pixels within that box. For a triangle with coordinate vectors $\mathbf{p_1}, \mathbf{p_2},$ and $\mathbf{p_3}$ and edges $\mathbf{e_1}, \mathbf{e_2},$ and $\mathbf{e_3}$, A pixel $\mathbf{p}$ is within the triangle if

$$(\mathbf{p} - \mathbf{p_1}) \times \mathbf{e_1} < 0$$

$$(\mathbf{p} - \mathbf{p_2}) \times \mathbf{e_2} < 0$$

$$(\mathbf{p} - \mathbf{p_3}) \times \mathbf{e_3} < 0$$

The color from the rasterizer is written to the pixel if it is determined to be within the triangle.

VI. Evaluation

A. BRAM Usage

The majority of our memory usage comes from the double buffer – each half-buffer has a width of 8 bits and a depth of 320×180. Additional BRAM usage comes from numerous trigonometric LUTs that are duplicated across different modules where they are needed for parallel accesses; each of these LUTs have a width of 8 bits and a depth of 256 and are hence relatively small in comparison to the buffers. In total, these add up to approximately 1 Mb of block memory usage, which is less than half of the total BRAM availability on the board.

B. Frame Rate

Our frame rate is directly correlated to the number of cycles it between receiving an input to the game logic and updating the double frame buffer with the corresponding result. The step that takes the most cycles by far is the image writer. It processes around 3200 triangles each of which has a bounding box of between 500 on average. On a 100MHz clock, our frame rate is about 60 frames per second.

C. Project Checklist

For this project, our minimum goals were to create a usable flight simulator game. This minimal game was to be all black and white, and follow the controller real-time movements. Surpassing these minimal goals, we were able to incorporate color into our game by adjusting the hue of the base gray shades and smoothed the tilt input of the game controller to prevent erratic game movements, creating a smoother gameplay. We also limited the range of motion of the player to prevent them from flying off the screen.

VII. Retrospective

We learned a lot from this project, both about hardware itself and working with computer graphics.

On the graphics side, it was interesting to learn about how 3D objects are rendered and the mathematics behind it, as well as how certain optimizations may be made to the standard algorithm to increase the efficiency of certain calculations. It was quite surprising how we could simplify some parts of the graphics simply because some parts of our generated world was predetermined.

A significant portion of the time was spent on debugging the rasterizer. Tackling the production of triangles, translations, rotations, projections, and shading all with integers can be very difficult to debug, especially with dealing with signedness and varying bit widths. We did use simulations a lot to debug, but we did many calculations by hand in order to compare. In hindsight, a more efficient method to writing correct code would be to split the code into smaller subtasks and writing simulations with expected output. This, we it would be easier

to detect which sections have correct functionality and identify where the bugs might be faster.

Overall, we gained a lot of experience working with graphics and incorporating hardware with computer graphics, making for a fun, interactive project. We believe that the process of debugging such a complex system was a highly valuable learning experience, especially where debugging output is very limited.

VIII. Author Contribution

All authors contributed equally to the development of this project and the writing of this final paper. Maggie Slowikowski implemented the peripheral hardware components of the project including SPI communication with the MPU-9250 sensor, complimentary filtering of sensor data, along with some contributions to game logic and graphics. Andrew Lee made significant contributions to game logic and graphics implementation. Specifically, he produced and confirmed correctness for much of the complex math involved in the game logic, the rasterizer, and the image writer. Selena Qiao made significant contributions to graphics implementation.

Acknowledgment

The authors would like to thank their instructor and project advisor Joe Steinmeyer for his continued guidance and support, as well as the rest of the 6.205 staff.

APPENDIX

Our repository can be found at: https://github.com/snonk/flap-gap

$$R = \begin{bmatrix} \cos\psi\cos\theta & \cos\psi\sin\theta\sin\phi - \sin\psi\cos\phi & \cos\psi\sin\theta\cos\phi + \sin\psi\sin\phi \\ \sin\psi\cos\theta & \sin\psi\sin\theta\sin\phi + \cos\psi\cos\phi & \sin\psi\sin\theta\cos\phi - \cos\psi\sin\phi \\ -\sin\theta & \cos\theta\sin\phi & \cos\theta\cos\phi \end{bmatrix}$$

Rotation matrix for world-to-camera computation.