FPinGA Final Report

Hank Stennes
Department of EECS
Massachusetts Institute of Technology
hstennes@mit.edu

Deniz Erus
Department of EECS
Massachusetts Institute of Technology
dierus@mit.edu

***Abstract*—Below we present our work towards an FPGA implementation of the classic video game Wii Sports Bowling. Our project consists of three main parts - the ray casting 3D renderer, light tracking camera control module, and physics simulation. The 3D renderer supports rendering of the ball and 10 cylindrical pins at arbitrary positions and rotations. The camera module tracks the position of a light source to determine the initial velocity of the ball. The physics simulation outputs the real time trajectory of the ball and pins through a 2D approximation. We have also implemented a scoring module for two player functionality but have not yet integrated this with the project.**

I. Introduction

Our goal is to create an FPGA implementation of Wii Sports Bowling. A central priority for the rendering module is maximizing throughput to allow a framerate of at least 60FPS. The simulation aims to be as accurate as possible within a 2D top-down approximation, including friction and mass of the objects. While we were successful in implementing the above goals, we have not yet fully integrated the camera and scoring for a playable bowling game. The current implementation allows the user to control the speed and direction of the ball using buttons, after which the simulation and renderer display the throw in realtime. Below we present our progress up to the project's current state.

II. Raycasting Renderer (Hank)

A. Overview

Rendering for the game is implemented using ray casting. The renderer module supports rendering of the ball and 10 cylindrical pins at positions determined by the simulation. Lighting is computed using Lambert diffusion with a single point light source. A high level summary of the algorithm used for the rendering pipeline is given below. In this section, "camera" will refer to the viewpoint in 3D space used for rendering. The following algorithm is repeated for all pixels on the screen:

1) Compute a ray from the camera to the position of a pixel on the screen in 3D space.
2) Select which objects to consider based on what objects could appear in the current part of the screen
3) Use mathematical methods to compute the intersection point of this ray with objects in the scene (currently one sphere for the ball, and 10 cylinders for the pins).
4) If there is no intersection with any object, color the pixel black.
5) If there is an intersection, consider only the closest intersection and compute the normal to the surface of the object.
6) Using the hit point, normal vector, light position, and object paint color at the hit point, compute a color for the pixel using the Lambert shading formula.

We decided to use floating point values for the renderer to avoid worrying about precision or overflow (at the cost of higher FPGA resource use). The renderer does floating point calculations using 8 Vivado IP modules. All components of the renderer are connected using AXI-Stream.

B. Python Simulation

We first implemented a Python-based ray casting engine to simulate the intended output of the FPGA renderer given fixed ball and pin locations. This implementation has been simplified several times in an effort to fit FPGA resource constraints, and serves as a reference for the exact computations the FPGA performs. The Python model was very useful in determining expected outputs when testing renderer modules. The script also uses its parameters to generate several hex-encoded floating point constants that the rendering module needs.

C. Renderer Components

The renderer consists of four main modules - ray_from_pixel, ray_intersect, hit_point, and lambert. Ray_intersect and hit_point are encapsulated within the check_objects module, which also handles the logic for scheduling objects into the pipeline based on which objects could be present in the part of the screen that the renderer is considering. This greatly increases framerate by avoiding unnecessary ray-object intersection checks.

D. Ray From Pixel Module

This module computes the 3D ray corresponding to a 2D pixel location. This depends on the pixel coordinates, screen resolution, and camera FOV. Since screen resolution and FOV are constant and known ahead of time, the formula has been simplified so that it may be computed with a single fused_multiply_add (a*b+c) module from the IP. While we initially considered using BRAM to store precomputed rays for each pixel, this module ended up being simple enough that this was not necessary. While throughput is important for the renderer, latency (within reason) was not a major concern.

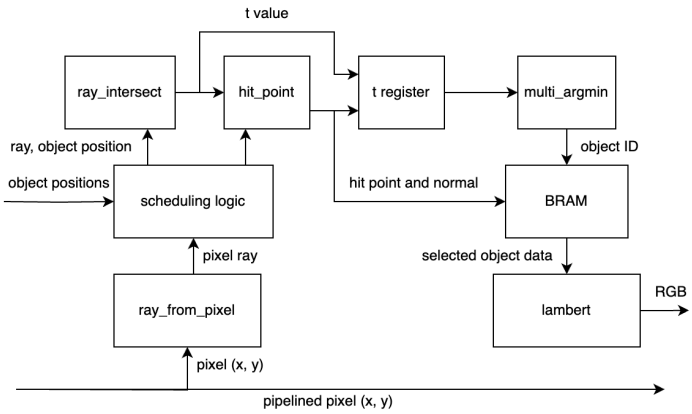

Fig. 1. Simplified block diagram for renderer module.

E. Check Objects Module

This module takes in a ray corresponding to a pixel, the positions of objects in the scene, and an object selection signal that can be used to enable or disable intersection checking for the ball or pins depending on the screen region. Based on the object selection signal, certain objects are scheduled into the pipeline, and the closest interaction point among these objects is returned from the module. Therefore, throughput changes depending on the value of object_select (if checking just the ball, the module will be ready for the next pixel on the next clock cycle, while if checking everything the module is ready every 11 clock cycles).

After objects are scheduled into the pipeline, the ray_intersect module starts computing the intersection of each object with the current ray. The module supports both spheres and cylinders with any position or rotation. If the object is a cylinder, the rotation is expressed as a unit vector representing the central axis of the cylinder. If the object is a sphere, the rotation is a zero vector. Luckily, most of the same hardware can be used for ray-cylinder intersection and ray-sphere intersection since the cylinder formula simplifies to something similar to the sphere formula when the axis vector is set to 0. There are a few pieces of conditional logic that change based on the object type. The formula for ray-cylinder intersection is provided below, where $\mathbf{d}$ is the direction of the ray, $\mathbf{c}$ is a vector from the origin of the ray to the center of the cylinder, $\mathbf{a}$ is the axis of the cylinder, and r is the radius of the cylinder, where the resulting a, b, and c are provided to the quadratic formula. The module outputs a t value that represents a distance along the ray where the intersection occurred.

$$a = \frac{1}{4}\left(\mathbf{d}^T\mathbf{d} - (\mathbf{d}^T\mathbf{a})^2\right) \quad b = \mathbf{d}^T\mathbf{c} - \mathbf{d}^T\mathbf{a} \cdot \mathbf{c}^T\mathbf{a}$$

$$c = \mathbf{c}^T\mathbf{c} - (\mathbf{c}^T\mathbf{a})^2 \ - r^2$$

The hit_point module takes the t value from ray_intersect and computes the coordinates of the intersection, the normal to the surface of the object, and the color of the object at the intersection point. This module also has minimal conditional logic based on whether the object is a sphere or cylinder.

After passing through the ray_intersect and hit_point pipelines, all object requests for the same ray are collected by a module that computes which object has the closest valid intersection point using a fully pipelined tree comparison structure. In the meantime, object data is stored in a BRAM. Once the minimum is computed, the corresponding intersection data is read from BRAM and sent out of the module.

F. Lambert Shading

Given the intersection point, normal, and object color, this module computes lambert diffusion shading using the dot product of the normal and the light direction. This module was initially problematic because it required two expensive vector normalizations. To cut down on DSP and LUT usage, the vector magnitudes were replaced with a constant that does not change significantly for the objects in the scene with our fixed light location. This causes objects close to the light source to appear very slightly dimmer than they should based on correct Lambert diffusion, but this effect is not very apparent in practice.

G. Frame Buffer

When rendering just the sphere, the renderer can keep up with the display signal and would not require a frame buffer. However, if the renderer has to check all objects, it takes multiple clock cycles per pixel. Therefore, the renderer runs independently from the display hcount and vcount and stores its results in a BRAM frame buffer. The buffer handles only the part of the screen that can have 3D objects and is itself divided into two regions, one where only the ball can be present and one where pins can also be rendered. Since the 3D region is only about a quarter of the screen, and only a third of the 3D region can have pins, the renderer can easily keep up with the screen refresh rate, and could actually render at up to 180FPS with the current 3D region dimensions.

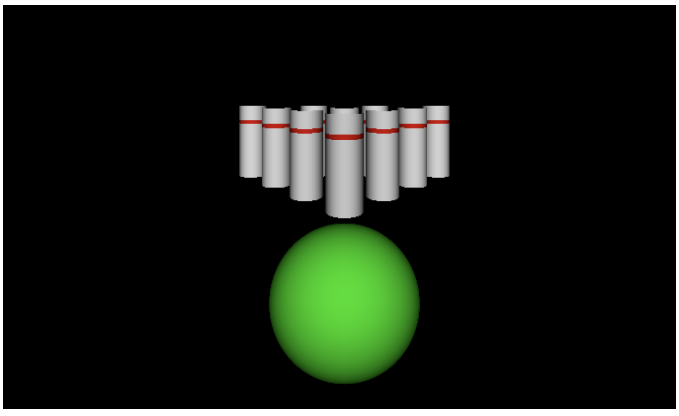

Fig. 2. Renderer output.

H. IP

The renderer requires 8 modules from the Vivado floating point IP - addition, multiplication, fused add/multiply, division, square root, less than comparison, float to fixed conversion, and fixed to float conversion. The greatest challenge of the rendering system has been decreasing the resource use of large modules containing many instances of these IPs. All IPs were initially set to the "Full DSP use" setting, which means 2 DSPs for most operations. This resulted in 156 of 120 available DSPs on the Spartan7 being used. After setting addition and multiplication to "Medium DSP use" (1 DSP), the DSPs were manageable but the capacity of LUTs was exceeded. The current successful design balances this by using full DSP adders and medium DSP multipliers, along with the simplifying heuristics in the lambert module described above. To use the IP in simulation, fake IP modules were developed with the same pins as the real IP. These modules use built-in Verilog math functions and artificial pipeline delays to match those of the real IP.

I. Hardware Challenges

Even after tweaking the IP settings and adding simplifying heuristics, the renderer eventually reached a point where it could not fit on the Spartan 7 FPGA without significant compromises to either rendering speed or functionality. As a result, we switched to the Nexys Artix 7 board. Unfortunately, this board only has VGA, which somewhat diminishes the smooth appearance of the Lambert shading. Migrating the rendering system from HDMI to VGA was otherwise fairly straightforward.

J. Integration With Simulation

The renderer supports full 3D movement of objects, while the physics simulation is 2D. The 2D locations of objects in the renderer reflect their simulated positions from a top-down view. To show pins falling, the rendering system detects when the 2D position of a pin has moved and rotates the axis of the pin to display the pin falling as it moves away. This was only possible with the Artix 7 FPGA, as computing the rotated axis for the pin requires an expensive vector normalization that cannot be simplified.

The renderer uses floating point while the simulation uses fixed point for lower resource use, so values must be passed through a fixed to float conversion before being processed by the renderer. The current design includes a separate float-to-fixed converter for every pin coordinate, although a more optimized design could reuse one converter, as the simulation updates very infrequently compared to the renderer clock. However, the resource use would still overwhelm the Spartan 7 FPGA even with this optimization.

III. Physics Simulation (Deniz)

The physics simulation calculates the trajectory of the ball and the pins and determines the score of each player. Initially, the goal was to use a camera to detect the player's swing and to determine the ball x and y velocities accordingly. However, due to large resource usage of 3D rendering, we used the Nexsys Artix 7 FPGA rather than the original Spartan 7 FPGA. Due to limited time and difficulty in adjusting the pin inputs and clock frequencies, the module could not be implemented and fully tested on the new FPGA. The camera module and its replacement will be discussed in the next part.

A high level description of the physics simulation and the block diagram are below:

1) User inputs initial x-position, x-velocity, and y-velocity of the ball using buttons and switches.
2) Ball module computes the ball's x-y coordinates taking into account the friction and spin of the ball.
3) Collision module assigns the ball and the pins a diameter and checks if they intersect. In case of an intersection, it calculates the new velocities of the pins according to elastic collision. It also determines which pins have been hit.
4) Pins module updates the x-y coordinates of the pins according to their new velocities.
5) Score module updates the player, round, and score.

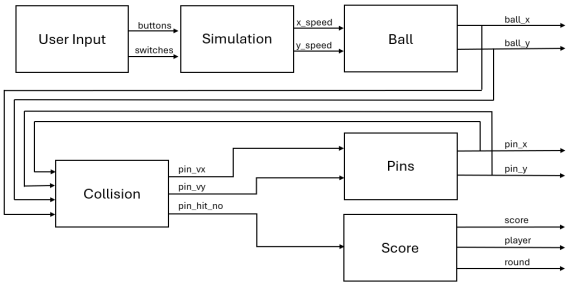

Fig. 3. Simplified block diagram for the Physics Simulation modules.

A. Original Camera Module

In the original plan, the FPGA with the camera was positioned on the floor, looking upwards to capture the swing of the player from under. The player held a breadboard with four parallel 5mm red LEDs in series with two 470 ohm resistors. The circuit was powered by a 9V battery, and a button was implemented which kept the light on as long as it was pressed. At the start of the swing, the player would turn the LEDs on and when the swing is complete, they would turn them off. These LEDs will be referred to as "light" from here on.

Light detection was implemented through thresholding and masking. The camera would detect light by masking the pixels that satisfied the Cr thresholds (red color content). Masking according to Y (illuminance) was also tested; however, the threshold varied greatly based on the ambient lighting of the room, making it hard to set a value that would work in a general case. It was determined that light detection worked reliably when the lower Cr threshold was 136 and the upper threshold was 255. By subtracting the light's x and y coordinates when the light was first turned on from the x and y coordinates right before the light was turned off, ball

velocity was determined. An example velocity calculation was as follows:

$$v_x = \frac{x_j - x_i}{t_j - t_i}$$

Although this module is not implemented in the final project, it was implemented in the original FPGA and was tested in the following ways: Crosshair output was observed to determine whether the center of mass followed the light; light detection was tested by outputting a 1 or a 0 on the seven-segment display, and lastly velocity calculation was tested by outputting the x and y velocities on the seven-segment display. The challenges that arose due to changing FPGAs will be discussed in a later section.

B. Camera Replacement Module

As a replacement for light detection, first the camera_input module was used. This module directly simulated the camera by randomly generating values for whether the light is on and the x and y coordinates of the light. This information was then fed into the camera module. To keep the game aspect of the project, the camera module was fully changed to compute the x-y speeds and the starting position of the ball. The up button is being used to determine the starting x-position of the ball. When the button is pressed, the ball moves towards right and stops when the button is released. When the ball moves out of the frame, it goes back to the leftmost point. The right button is used to set the x and y speeds of the ball. When Switch 0 is on, pressing the right button increases the x-speed by one. When Switch 1 is on, pressing the right button increases the y-speed by one. The speeds turn back to 0 when speed 6 is reached. Turning switch 2 on indicates that the x-speed is in the negative direction. The x and y velocity values are sent to the Ball module, while the ball's starting x and y positions are sent to the renderer.

C. Ball Module

The ball module takes in the initial x and y velocities and the positions. At each time step, the ball's velocity decreases depending on friction, which is a parameterized value. It also has support for altering the x velocity according to the spin of the ball, however, the spin calculation is not implemented so this value is set to zero. The module does not take new input until the ball reaches the end of the lane, signifying the end of a roll. When the ball reaches the length where the pins start, the collision module starts checking for collisions. This module is turned on every 3 million clock-cycles to produce a visually appealing ball movement with the given velocities. The module uses values multiplied by 1000 to reduce the need for floating point IPs. The IP is used after this module to make the ball's coordinates compatible with the renderer.

D. Collision Module

The collision module differs slightly from the way it was implemented in the Spartan 7 FPGA. Due to the limited resources and the high number of comparisons needed to determine if a pin was colliding with the ball or another pin, a highly pipelined finite-state machine (FSM) was used in the Spartan 7 FPGA. Since resources were not a problem in Artix 7, pipeline stages were reduced to four to make it easier to debug the implemented physics equations. There was not a limit on how many pipeline stages we could have because the simulation only needs to produce new values once per frame over the time span of millions of clock cycles. As such, dividing the calculation over a number of clock cycles did not negatively affect the modules. The diagram of the FSM is below.

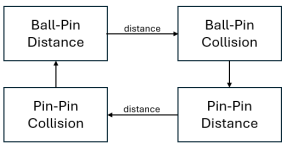

Fig. 4. Finite state machine (FSM) for the collision module.

The first stage of the collision module determines the distance between the pins and the ball. It uses a for loop with 10 cycles to account for each pin and looks at the sum of the squared differences between x and y values. The result is the squared distance between two points as shown below.

$$dist_i = (x_1 - x_2) * (x_1 - x_2) + (y_1 - y_2) * (y_1 - y_2)$$

The second stage checks whether the ball and pin would overlap if they were circles. To do this, it takes the square of the sum of the radiuses and compares it to the distance mentioned above. In case of a collision, it updates the x and y velocities according to the elastic collision formula:

$$v_{Af} = \frac{m_A - m_B}{m_A + m_B} * v_{Ai} + 2 * \frac{m_B}{m_A + m_B} * v_{Bi}$$

$$v_{Bf} = 2 * \frac{m_A}{m_A + m_B} * v_{Ai} - \frac{m_A - m_B}{m_A + m_B} * v_{Bi}$$

The third stage calculates the squared distances between each pin. This is done through nested for loops: the outer loops is from 0 to 9 inclusive, while the inner loop starts from i+1, where i is the outer loop's value, and goes to 9 inclusive. By starting the inner loop from i+1 unnecessary calculations are avoided. Otherwise, every pin combination would be checked twice since looking at the collision between pin 9 and pin 8 is equivalent to looking at the collision between pin 8 and pin 9.

The last stage checks for collisions between pins and updates their velocities accordingly. It has the same logic as in the second stage but uses nested for loops discussed in the third stage.

In the advanced version of the collision module, the post-collision velocities were calculated taking the relative velocities into account to more accurately model the interactions between objects. The difference is in the collision calculation stages of ball-pin and pin-pin collisions (stages 2 and 4). The added logic is as follows: the relative velocity components are

calculated by taking the difference in x and y velocities of the objects:

$$rel\_v_x = vx_1 - vx_2$$

, the collision vectors are calculated by taking the difference in x and y positions of the objects:

$$dx = x_1 - x_2$$

. The vector magnitude is computed using the collision vectors. The value we get from this calculation is the square of the vector magnitude to avoid dealing with square roots:

$$mag = dx * dx + dy * dy$$

. All of these values are then used to compute the projection factor:

$$proj = \frac{rel\_v_x * dx + rel\_v_y * dy}{mag}$$

. The post-collision velocities are found as:

$$v\_x_{1,new} = v\_x_1 + \frac{2 * m_2 * proj * dx}{m_1 + m_2}$$

.

This version of the collision module was tested without the 3D rendering. It barely met the timing constraints so we could not combine it with the renderer since we were very close to the timing constraints while using the simplified version. One reason it only barely met timing was because the module used combinational logic along with sequential. The difference in the reaction of the pins was not noticeable using this module when the pins and the ball were close together but was more apparent when they were further away. Since this issue was discovered very late in the project while integrating the two modules, and in the renderer the objects are close together, we decided to focus more on other parts of the project and used the simpler version.

E. Pins

The pins module calculates the x and y positions of the pins according to the velocities passed in from the collision module. The velocities are also updated taking friction into account and passed back into the collision module for the collision calculation in the next time frame. The pin positions are also passed into the renderer.

F. Score

The score module stores the scores of each player as an array of scores per round. It performs the necessary calculations for strike and spare scores. It also updates the player after a player makes two swings. The scores are calculated based on the number of pins hit data that comes from the collision module. The round, player, and score data is passed on to the renderer.

G. Sizes

All of the x values used in this project were 10 bits while y values were 9 bits. All of the velocities and intermediate values were 16 bits.

H. Hardware Challenges

As mentioned before, the main challenge was changing from Spartan 7 to Atrix 7 FPGA. The biggest effect of this change was on the camera. All the pins were redefined for the camera and VGA support was added. However, when camera was plugged in, it would take a few resets for it to start streaming. When we ran the same code another day, the camera only gave a green screen and we could not get it streaming. This is likely due to VGA not being able to support the high clock frequency. We tried to debug the camera issue, however, since we changed FPGAs late in the project we did not have time to get it fully functional and decided to go with the safer option of using button inputs and putting time into other modules.

IV. Evaluation and Conclusions

On the rendering side, all goals were achieved aside from rendering a plane for the bowling lane. We had also hoped to implement Lambert diffusion with multiple light sources, although even a single light source turned about to be expensive in terms of FPGA resources. Implementing the rendering module yielded many interesting insights on handling large amounts of math in FPGA designs. Most notably, we found that the use of floating point values was not necessarily worth the increased resource use compared to fixed point, as well as the verbose code required for instantiating so many IPs. To manage this complexity, the rendering math is split across 22 modules including the top level ones documented above. While the latency of the renderer is not an issue, this also resulted in a system with 352 pipeline stages. As a result, we feel that the renderer could have been implemented more simply with 32-bit fixed point using appropriately scaled values.

On the simulation side, the camera could not be used. The ball module and pin module were implemented as expected with support for friction and swing. Two versions of the collision module were implemented: one which assumed head-on-head collisions, and one that took the relative velocities into account. The simpler version was used because of timing issues when integrated with renderer. The score module was implemented and tracked the player, round, and score, however, we did not have time to render it. If we had more time, we could have solved the issues with the camera and the VGA. The complex version of the collision module could also be optimized to better fit timing by pipelining and integrated into the project.

During the project, along with the technical skills, we learned the importance of paying attention to the resources used. We learned how to structure a project and implement it piece by piece after testing.

A. Resources

39684 LUTs were used, 51 brams were used, and 168 DSP Blocks were used. Most of the DSP blocks are used by the renderer, however, the collision module also uses 20 of them. The 120 DSPs in Spartan 7 were not enough for this project but Atrix 7 had enough. The timing was met with 1.880ns of slack.

V. Contributions

A. *Hank Stennes*

Hank worked on the 3D rendering system that takes in the ball and pin locations to render the scene. Beyond this, Hank worked on adding the float to fixed conversion layer and pin rotating logic to integrate the simulation with the renderer.

B. *Deniz Erus*

Deniz worked on the physics simulation that computes the motion of the ball and pins. Deniz also worked on the camera motion tracking system, scoring module, and button input system.

VI. View Our Code

References

[1] Ray Cylinder Intersection Formula. https://stackoverflow.com/questions/73866852/ray-cylinder-intersection-formula/.

[2] Lode Vandevenne. Lode's Computer Graphics Tutorial. https://lodev.org/cgtutor/raycasting.html/.