CSEE 4840: EMBEDDED SYTEMS PROJECT REPORT

Space Battle Game

Ben, Philip Osuri (bp2613)

1. Introduction

This is a simple game that I chose to gradually introduce me to Embedded systems. It is a 2-D version of the popular “Asteroids Game”.

Creating a playable game using the Cyclone V FPGA development board is a comprehensive project that fully encompasses the ideas and concepts learned in the classroom. For example, configuring hardware as game controls, applying output display buffering techniques, and developing and utilizing IP cores all involved techniques learned throughout the course. Developing a video game allows for the culmination of many learning objectives in a single project that in the end is both challenging and satisfying.

This game was specifically targeted for development due to its ability to implement and demonstrate a variety of random elements into the gameplay. This was important as the hardware focused on random number generation. Asteroids allows for colors, positions, velocities, directions and graphic models to be randomized, illustrating the ability to quickly generate large amounts of uniformly distributed random numbers in hardware without bogging down the system. Overall, the hardware accelerated randomness adds life to the game without affecting its fundamental nature, playability or performance.

2. Design

As previously mentioned, rationale for the implementation of the game on the HPS/ FPGA system involved being able to implement a simple-enough game that could emphasize and showcase the usage of IP core hardware accelerators to enhance the game. In this case, the hardware was used for random number generation.

The basis of this game is simple and similar to original Atari, Inc. design from 1979. It is a space-themed multidirectional shooter, in which the player controls a triangular ship with the ability to rotate left and right, apply a forward accelerating thrust, and shoot bullets forward. The player attempts to shoot floating asteroids to accumulate the largest number of points as possible. If the player collides with an asteroid, the player "dies", and the game is reset. The game-space is two-dimensional, and incorporates a toroidal coordinate system, meaning that the edges of the screen are wrapped around to the opposing edges.

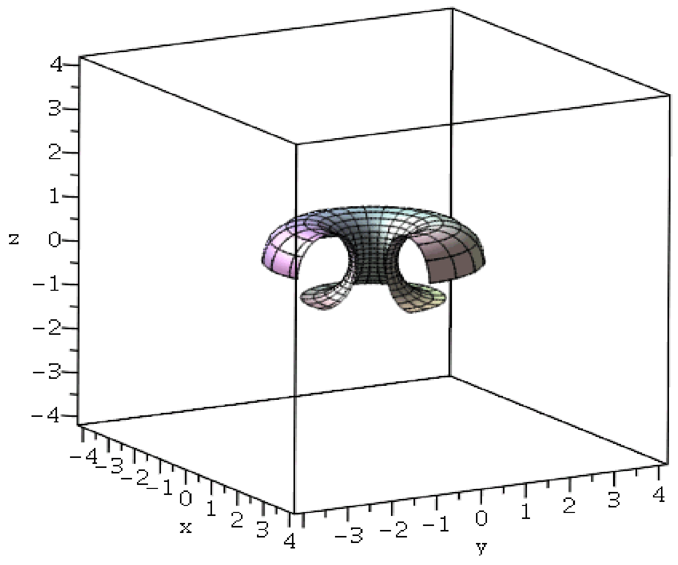


Figure 1:Shows the toroidal coordinate system

This game involves some mathematical background knowledge for the full understanding of the logic and design of the game. There are three mathematical topics to be discussed: kinematics, toroidal coordinate wrapping, and number range mapping.

Kinematics is the study of motion. There are some fundamental kinematic equations used in physics to describe object movement in multiple dimensions, and some of those equations are used in this project. The basic principles here to recognize are those relating position to velocity and acceleration. These, of course, are $Velocity$ = $(P_2 - P_1)/t$ and $Acceleration$ = $(V_2 - V_1)/t$. In software, we can only plot arbitrary positions as pixels on a screen, which can be lifeless. However, when we reference real time by using time between frames, and combining that with these equations, we can implement (or at least mimic) a life-like kinematic motion.

The toroidal coordinate system mentioned above helped in keeping objects within the screen-space. This is opposed to scrolling or having invisible walls at the edges of the screen. A torus is a geometric shape that is the result of wrapping around both pairs of opposite edges. This allows the torus to feature the characteristic of having equal coordinate spacing in both the X and Y direction when a 2D space is mapped to it.

In the game, having this toroidal coordinate system is important to allowing our objects to generate and display correctly when approaching and crossing over the edges of the visible screen. First of all, the game's physics and kinematics are not altered when approaching screen edges (as they would be when approaching the poles of a spherical mapping, for example). Objects are rendered smoothly and consistently despite coordinate location. Secondly, this allows objects to wrap around to the other side of the screen when they cross the boundary of the opposite edge. The right and left sides of the screen are wrapped (X coordinates), as well as the top and bottom edges (Y coordinates). This type of coordinate system provides a unique playing experience for the user and programming challenge for the developer.

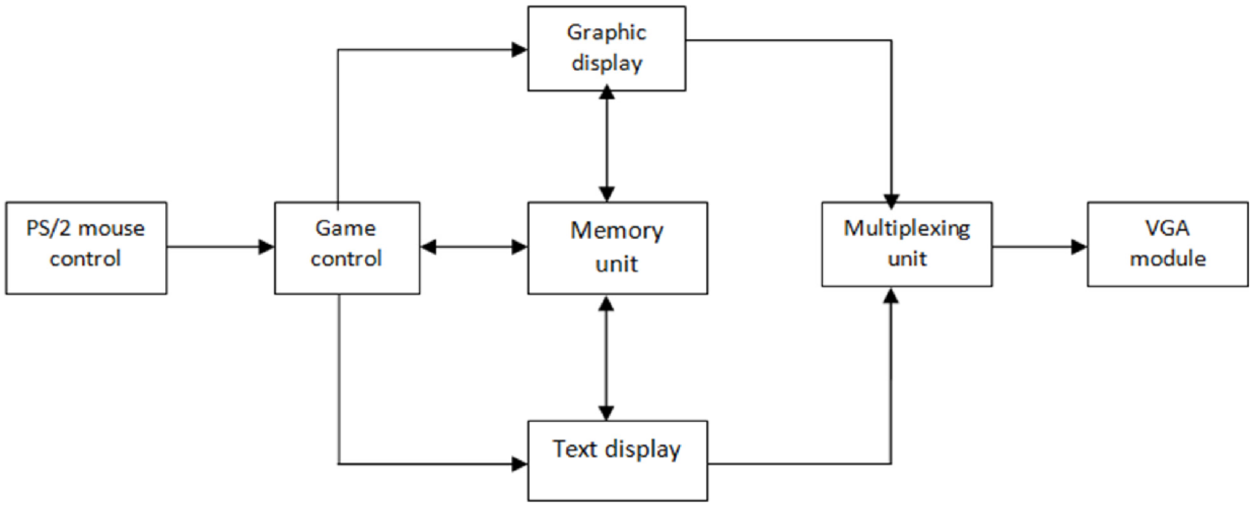


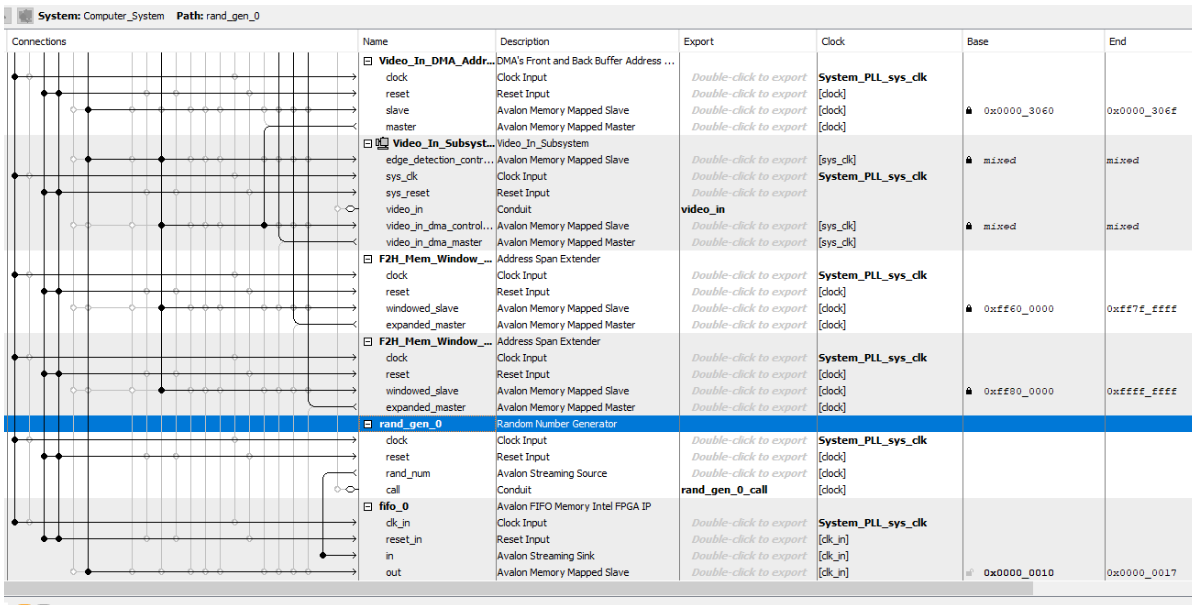
Figure 1: System Block Diagram

2. SOFTWARE/ HARDWARE

Programming the game logic was a fun challenge with a bit of a learning curve. It required different forms of thinking from how I would normally go about writing code, based on the things I have been taught in class. For example, one of the important concepts in this type of game design is the idea of a separate game space and screen space. The game space is where all of the logic and computations happen, while the screen space is the literal space on the screen. The "background" workings in the game space drive the contents of the screen space. As another example, the position of objects in the game space are represented by a single point, while the screen space objects are represented by fully drawn models. We perform calculations using the points in the game space, and those results are translated to the screen space. This kind of thinking was initially tricky to wrap my head around, but it made sense after some playing around with the code.

Dynamically instantiating, keeping track of, and removing objects from the game on command without causing memory leaks was also something I struggled with a bit during the development process. This game requires the system to keep track of large number of objects at once, instantiating new objects and deleting old ones where appropriate. To help with some of this memory allocation, I ended up implementing a “vector in C”, which allowed me to use the data type “vector” which had similar functionality to the vector type in C++. This vector implementation allowed me to effectively and dynamically add and delete game objects quickly. The vector class performed the background computations for allocation, reallocating, and freeing memory.

In terms of the hardware development, implementing Altera’s Random Number Generator and Avalon FIFO Memory involved additions to the computer system in the Platform Designer, as well as some slight VHDL code modification. Both the RNG and FIFO Memory need to be added to QSYS/Platform Designer, and they are connected as one might expect. They both need to connect to the same system clock and reset, and then the random number data from the Avalon Streaming Source connects to the Avalon Streaming Sink on the FIFO Memory. From there, the memory mapped slave on the FIFO memory needs to connect to the HPS to FPGA lightweight AXI master bus to allow us to assign a virtual address to this memory in software. The conduit port on the RNG also needs to be exported.



System: Computer_System **Path:** rand_gen_0

Name	Description	Export	Clock	Base	End
⊟ **Video_In_DMA_Addr...**	DMA's Front and Back Buffer Address ...				
clock	Clock Input	Double-click to export	**System_PLL_sys_clk**		
reset	Reset Input	Double-click to export	[clock]		
slave	Avalon Memory Mapped Slave	Double-click to export	[clock]	0x0000_3060	0x0000_306f
master	Avalon Memory Mapped Master	Double-click to export	[clock]		
⊟ **Video_In_Subsyst...**	Video_In_Subsystem				
edge_detection_contr...	Avalon Memory Mapped Slave	Double-click to export	[sys_clk]	mixed	mixed
sys_clk	Clock Input	Double-click to export	**System_PLL_sys_clk**		
sys_reset	Reset Input	Double-click to export			
video_in	Conduit	**video_in**			
video_in_dma_control...	Avalon Memory Mapped Slave	Double-click to export	[sys_clk]	mixed	mixed
video_in_dma_master	Avalon Memory Mapped Master	Double-click to export	[sys_clk]		
⊟ **F2H_Mem_Window_...**	Address Span Extender				
clock	Clock Input	Double-click to export	**System_PLL_sys_clk**		
reset	Reset Input	Double-click to export	[clock]		
windowed_slave	Avalon Memory Mapped Slave	Double-click to export	[clock]	0xff60_0000	0xff7f_ffff
expanded_master	Avalon Memory Mapped Master	Double-click to export	[clock]		
⊟ **F2H_Mem_Window_...**	Address Span Extender				
clock	Clock Input	Double-click to export	**System_PLL_sys_clk**		
reset	Reset Input	Double-click to export	[clock]		
windowed_slave	Avalon Memory Mapped Slave	Double-click to export	[clock]	0xff80_0000	0xffff_ffff
expanded_master	Avalon Memory Mapped Master	Double-click to export	[clock]		
⊟ **rand_gen_0**	Random Number Generator				
clock	Clock Input	Double-click to export	**System_PLL_sys_clk**		
reset	Reset Input	Double-click to export	[clock]		
rand_num	Avalon Streaming Source	Double-click to export	[clock]		
call	Conduit	**rand_gen_0_call**	[clock]		
⊟ **fifo_0**	Avalon FIFO Memory Intel FPGA IP				
clk_in	Clock Input	Double-click to export	**System_PLL_sys_clk**		
reset_in	Reset Input	Double-click to export	[clk_in]		
in	Avalon Streaming Sink	Double-click to export	[clk_in]		
out	Avalon Memory Mapped Slave	Double-click to export	[clk_in]	0x0000_0010	0x0000_0017

In the end this project is a fully functional and satisfy game based on the classic Asteroids. During game execution, there is no hesitation from the system, and performance is exceptional. The game runs at a full 60 frames per second at all times, which is both desired and expected from our display hardware that runs at 60Hz. The smooth game performance can be attributed to the use of the vertical sync (VSync) functionality of the pixel buffer controller, which synchronizes frame updates with the display hardware, and also to the efficient game logic by javidx9 that this project was based off of. This includes using the elapsed time between frame updates as a factor when performing kinematic equation calculations for object movement.

The new IP cores introduced in this project also function satisfactorily. The random number generator produces and streams numbers to the FIFO memory quickly and with a uniform distribution. Concurrently, the FIFO memory mapped register effortlessly manages the fast incoming data stream and produces outputs in the data register without introducing any additional lag or delay. This system is quick enough to handle our need for multiple random numbers per frame.

The random numbers are used for 1. Asteroid Colors, 2. Bullet Colors, 3. Asteroid Velocity, 4. Asteroid Direction, 5. Asteroid WireFrame Graphics. The position, velocity, direction and graphic randomization only happen occasionally, but asteroid and bullet color randomization happens every frame for each instance of the objects. This shows off the ability to stream large amounts of random numbers into the system without slowing down the system performance. In the software implementation, we would come to a point sooner where the need for this many random numbers would slow down the system. But because this random number generation system is hardware accelerated, we have access to a consistent stream of random numbers that we can access at will without slowing down the system. Colors of every asteroid and every bullet on screen change randomly per frame, based on the numbers received from the IP cores, demonstrating high throughput of the random number stream.

2. REFERENCES AND RESOURCES

This project would not have been successful if it were not for Altera, javidx9 on YouTube, Cornell University, and eddmann.com. These sources provided hardware and software aids that were influential in the successful functionality of this project. First of all, YouTube channel javidx9 provided the high-level concepts for the game logic used in this project. His video on the Asteroids game helped me properly reason my way though the game logic. Next, the vector implementation was taken from an article from eddmann.com. This article provided the code and a nice explanation for how vector functionality could be implemented in the C language. The Random Number Generator and FIFO memory IP cores were created by Altera, and the IP Cores themselves come bundled with Quartus 18.0. Finally, some of the graphics primitives/ drawing routines for different shapes used were from Cornell University.

Links to code/ designs borrowed from others:

- Graphics Primitives (From Cornell University): http://people.ece.cornell.edu/land/courses/ece5760/DE1_SOC/HPS_peripherials/univ_pgm_computer.index.html

- Level Code Design (From javidx9): https://www.youtube.com/watch?v=QgDR8LrRZhk

Links to Datasheets for Altera IP Cores (also included as PDF's):

- Random Number Generator IP Core User Guide: https://www.intel.com/content/www/us/en/programmable/documentation/dmi1455632999173.html#dmi1455633326157

- Avalon On-Chip FIFO Memory Core User Guide: https://www.intel.com/content/www/us/en/programmable/documentation/sfo1400787952932.html#iga1401396003807