2011

Real-Time Soccer Controller

EE4214 – Real Time Embedded Systems

Arun Ovireddiar Sundararajan (U083815E)
Govind Chandrasekhar (U076288J)
Hardik Mehta (A0090211Y)
Li Shiju (U094734E)
Samyak Jaroli (A0090374E)
Shyamsundar Venkataraman (U083813H)

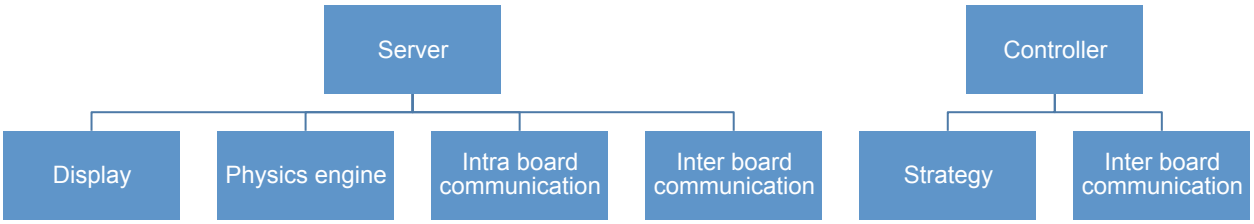
Team 2
11/4/2011

Introduction

The aim of this project is develop a real-time soccer controller for a 5 a side football team on Xilinx Spartan 3E FPGA Board.

The project is broadly divided into a server and controller. The server displays the progress of the game by performing GUI changes on the TFT by means of the display microblaze. It also performs game physics calculations such as player-player, player-ball and player-wall collision among others by means of the physics engine microblaze, which makes the game realistic by enforcing constraints based on laws of motion. The controller, on the other hand, hosts the algorithms to control the movement of various players in the team. The server communicates with the two teams' controllers via serial port in real-time.

System Design



The server is designed to be a dual processor system in which the display runs on one processor and the physics engine runs on the other. Communication between the physics engine and display occurs in real time via mailbox, by using write and read blocking statements.

The controller is designed to be a single processor system and communication between server and controller is done via UART using interrupts. Both packets that are sent from the controller to the server and vice versa follow the encoding and corresponding decoding provided in the project wiki.

Roles and Responsibilities

Hardik Mehta

UART Communication: To understand serial port communication and interrupts & to implement Inter-Board using interrupts via serial ports.

Li Shiju

Physics Engine: To develop the server physics engine governing the physics calculation of the soccer game, including setting up the game, updating speed and position of ball and players, resolving different types of collision, implementing player kick, and checking goal and foul.

Shyamsundar Venkataraman

Team Controller: To conceive and develop artificial intelligence for the interaction of players with each other and the ball, including man-marking, ball attacking, goalkeeping and other strategies.

UART Communication: To understand serial port communication and interrupts and to implement Inter-Board using UART interrupts via serial ports.

Integration: To integrate and debug all components of the system.

Samyak Jaroli

Mailboxes: To implement mailboxes for inter-processor communication.

Packet Construction: To implement encoding and decoding of 32 bit communication packet.

Govind Chandrasekhar

Team Controller: To conceive and develop artificial intelligence for the interaction of players with each other and the ball, including man-marking, ball attacking, goalkeeping and other strategies.

Physics Engine: To develop the server physics engine governing the physics calculation of the soccer game, including game initialization and data organization, updating speed and position of ball and players, resolving different types of collision, implementing player kicks, checking goal and foul scenarios, managing game states, addition of timer, push buttons, mailboxes and UART communication.

Integration: To integrate and debug all components of the system.

Arun Ovireddiar Sundararajan

Display: To learn implementation of display on TFT, by understanding Xtft Api's. Conceive and develop Graphic User Interface to handle different game states. Understand and implement double buffering to improve performance of graphic rendering, by writing primitive header file functions.

Physics Engine: To implement timer to time the game and trigger respective game state changes, also checking calculation for scenarios such as checking foul, checking goal etc.

Optimization: To work on ways to optimize the performance of the code.

Integration: To integrate and debug all components of the system.

Communication

Inter-Board Communication

Data is encoded as 32 bit packets based on the packet construction criteria provided in the wiki. UART interrupts are used to receive incoming data packets, which are then stored in a **circular data queue**. This queue is read and decoded every clock cycle, thereby reducing the amount of processing done in the **interrupt**. Its also allows us overcome the clogging of the UART buffer which has a limited size. Decoding and encoding is made intelligent such that they accept initial packets only once and not during the game.

Intra-Board Communication

Data to be sent from the physics engine core to the display core is slotted into a neat structure and sent via **mailbox** through write and read blocking statements. This structure consists of all the information that the display contains and has a packet size of 108 for each clock cycle.

Process Description

Server – Display

The display renders GUI via TFT based triggers from the physics engine. The display fundamentally has two states; the **pre-game state** used to signal correct communication between server and the two controllers and the **in-game state**, which displays the football field, players, ball and scoreboard.

Since a majority of the graphics rendered are circles, computation time of the drawing these is reduced by implementing **Bresenham Algorithm** (RasterCircle). Also, **bitmap arrays** for the player and ball are generated during runtime to reduce computation time and code size. In order to reduce flicker rate and increase frame rate **double buffering** is used.

Server – Physics Engine

All possible collisions under the purview of the game are handled here; emphasis has been placed on the development of algorithms that are light for computation.

Collisions of moving bodies are handled by finding the acute angle (between 0 and 180) of the line joining the two bodies that are colliding. The angle is determined using the **tan inverse** and velocities of the bodies along the line of collision are calculated. In the case of ball-player collision, the component **velocity along the line of collision is reversed**, unless the ball is stationery, in which case it is given

twice the velocity of the player. When two players collide, they **exchange velocities** along the direction of collision.

The occurrence of collision is calculated by first adding each object's velocity to its position in the previous cycle; in order to ensure that no overlap occurs, the players are moved back by a constant fixed amount until the overlap is cancelled out. This constant is termed "**granularity**" and can be adjusted for precision; high precision comes at the cost of greater processing time per cycle.

The collision of the ball and players with the walls is done in a straightforward manner; for the horizontal and vertical walls**, velocity components along the x and y axes respectively are reversed**. For collision at the corner triangles, Vx = -Vy and Vy = -Vx. These collisions are detected in the first place by checking if the ball intersects with the closest permissible line on the edges of the field that the ball should not intersect.

Server also has algorithms to invoke spot refereeing, random generation of player & ball position and other such features.

Team Controller

The controller commands the strategy, which dictates motion and actions of players. In our controller, players are categorized into three personas – goalkeeper, attacker and defender. The following gives a detailed description of how these strategies are implemented and how roles are assigned:

Goalkeeper: The goalie's main job is to prevent goals from being scored. For this reason, his position is limited to the foul box and the goal. When the ball is outside the foul box, his **position depends directly on the current 'y' position of the ball**. If the 'y' coordinate of the ball is beyond the reach of the foul box, the goalie stands at the corresponding corner of the goal to block the ball. If the ball comes enters the foul box, the **goalie advances towards the ball** and kicks the ball as far away from the goal as possible. This strategy is set to mimic real life goalkeeping.

Attacker: There are 2 types of attackers – one **main attacker** and one for **assisting the main attacker** in scoring a goal. The main attacker is **dynamically chosen** as the player who is **closest to the ball**. Another player who is also close to the main attacker is positioned at the **halfway point between the main attacker and the goal**. This way, the main attacker can **pass the ball** to the assisting attacker in the case that he cannot score a direct goal. Both the attackers try to kick the ball towards the goal if it is possible to score. Else they pass to one another until a goal can be scored. If the ball is in a direction opposite to that of the goal, the players **kick the ball away from the opponents' goal** to prevent the opponents from scoring.

Defenders: There are 2 defenders who help in **man marking** the opponents. The aim of these defenders is to prevent the opponent players from passing the ball to one another. To this end, the defenders head towards the **position between the ball and the opponent closest to them** and try to move to a position, which is at a **ratio of 1:4** between the opponent and ball. This would **increase the chances** of defenders getting hold of the ball in comparison with a strategy of merely following the opponent player.

The controller also has an ability to **trigger offensive and defensive play** in real time in response to a **dipswitch toggle**. The above-mentioned strategy is for offensive play. During defensive play, all players move towards the defensive side to make sure that a goal is not scored.

Testing of the Controller:

The controller strategy was tested using unit test cases for every strategy function. We made sure that they were working for all boundary cases. All the test cases used for testing have been attached as a part of the zip file along with the rest of the code. The test cases can be found in the main.h and main.c files of the controller strategy.

Problems Encountered and Scope for Improvement

Packet Loss: The loss of packets during UART communication affected the game flow of the strategy and resulted in the display of decoherent motion on the screen.

Loss of packets could have been solved through better **caching** and **error checking**.

Time taken to Establish Communication: Since incremental coding was not possible due to the time spent in completing UART and mailbox code, the integration and testing process was short from a time frame perspective.

Display Frequency: The display code was very slow, despite the usage of double buffering and bitmaps. This slowed down the entire gaming experience significantly.

Execution time can be improved by porting the server code (particularly the code on the display microblaze) to **BRAM**. Code size can be considerably reduced by choosing to use **standalone BSPs** as opposed to Xilkernels. The size of the code is important since the BRAM is limited to 64KB.

These methods were implemented and resulting speed increases were successfully observed after the final presentation of this project.

Other: There existed minor bugs due to inaccuracies in mathematical calculations and improper use of data types such as fixed point, double and other notations proved difficult to manage once the project scaled up.