# INFORMATION PROCESSING COURSEWORK
# THE MAZE GAME

GROUP 4


AUTHORS

Ifte Chowdhury
Brendon Ferra
Haaris Khan
Savraj Sian
Mahanoor Syed
Emma Wardle

## DESIGN OUTLINE AND SYSTEM PURPOSE

The IoT system designed by our group is a multiplayer maze game. The game consists of 2 modes: The Maze Race and Dodgeball. The Maze Race is a race between players to reach the goal to progress to the next level. Dodgeball is a competition between players where, upon collision, the slower moving ball loses a life. The game currently supports up to 4 players but can be played with players missing. The game uses the FPGA as a controller for the balls. Simultaneously, on the machine running *game.py*, the player can view the position/movement of their ball relative to others in the GUI on a local PC.

## THE MAZE RACE AND DODGEBALL – GAME OUTLINE

**Rules of The Maze Race**

1. Use your FPGA to guide the ball through the maze to the goal.
2. Do not fall into any other holes or you will lose points.
3. Pick up powerups to gain an advantage and extra points.
4. Reach the goal before other players to get the maximum points.

**Rules of Dodgeball**

1. Tilt the FPGA towards the other balls, where the faster ball kills the slower ball.
2. Pick up powerups to gain an advantage over other players.
3. Keep playing until all other players have 0 lives left.

**Interacting with the game**

1. Tilt the FPGA in the direction you want the ball to accelerate.
2. Switches change the ball sensitivity.
3. 7SEG display shows you which colour ball and the position the player finished the game in.
4. Button presses changes game mode from The Maze Race to Dodgeball and vice versa.
5. LEDs show your lives left in the Dodgeball minigame.

## OVERALL ARCHITECTURE

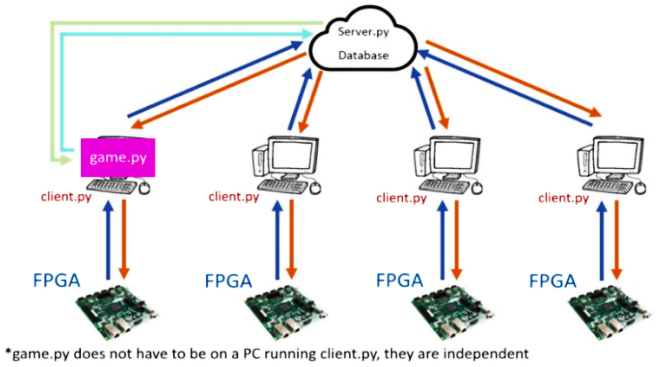


*Figure 1 – Overall Architecture*

*Figure 1* shows the overall architecture of the game system. The game has been structured such that each PC is running *client.py* to interact with the server, and one PC is running *game.py*. A *server.py* script is running on the EC2 instance in the cloud. The arrows in the diagram represent dataflow between different nodes in the system. The blue arrows represent the data that is sent from the FPGA to the server – for example: accelerometer, switch, and button data. Orange lines on the diagram represent the data that the FPGA receives from the server. Finally, the green and cyan lines represent the data sent between the server and the PC. The server sends and receives via TCP connection, processes the game input, and also updates the databases.

## GAME FUNCTIONALTY - SOFTWARE IMPLEMENTATION

### LOCAL GAME RATIONALE

For the software, the *Pygame* library in *Python* is used to draw the GUI, as it helps with managing events (i.e., key presses) and sound channels, and drawing on the screen. This is done locally instead of on a server, due to four reasons:

1. The while loop which runs the game consumes a large amount of power, which is too demanding for an EC2 instance to achieve good performance.
2. The EC2 instance being used does not have a video displayer/screen.
3. If an instance with a screen was used, the latency would increase if the GUI was broadcasted to a local machine.
4. Splitting up the game so that the physics was on the server and the GUI was drawn locally would cause unnecessary overhead.

Therefore, the most viable option was to run the game on a local machine. Since the game is local multiplayer (and the virtual space is the same for all players), only one computer needs to draw the GUI.

### OOP RATIONALE

```
58  class Ball:
59
60      friction = 0.95 #gradual slowing
61      restitution = 0.6 #bounce
62      scores = [0, 0, 0, 0]
63      kills = [0, 0, 0, 0]
64      lives = [10, 10, 10, 10]##############
65      won = [] #list to determine how many balls have won, and when to change level
66
67      def __init__ (self, ID):
86
87      def respawn_animation (self):
115
116     def motion_calc (self, dt):
146
147     def frame_collision (self):
172
173     def block_collision (self):
215
216     def hole_collision (self):
259
260     def powerup_collision(self):
```

```
466 class Block:
467     def __init__ (self, coords):
468         self.rect = pygame.Rect(coords[0], coords[1], coords[2], coords[3])
469         self.pos = [coords[0], coords[1]]
470
471 class Hole:
472     def __init__ (self, coords, colour):
473         self.image = hole_image
474         self.rect = self.image.get_rect()
475         self.pos = [coords[0], coords[1]]
476
477 class Goal (Hole):
478     def __init__ (self, coords, colour):
479         self.white = goal_white_image
480         self.image = goal_glow_image.copy()
481         self.image.fill(colour, None, pygame.BLEND_RGBA_MULT)
482         self.rect = self.image.get_rect()
483         self.pos = [coords[0], coords[1]]
484
485 class Dodgeball (Level):
486     def __init__(self, block_coords, hole_coords, colour, bg_colour, edge_colour):
487         self.colour = colour
```

*Figure 2 – Class Definitions*

It is standard practice in the games industry to use Object-Oriented Programming (OOP) *(Figure 2),* for a variety of reasons, some of which are relevant to our project such as:

1. OOP reduces code duplication since every instance of a class (e.g., ball) uses the same code. This also makes the code easily modifiable, reduces technical debt and makes the behaviour of thousands of instances highly predictable.
2. Inheritance allows similar classes to share attributes.
3. Class variables help organisation and reduce the need for global variables.
4. OOP better reflects the physical world and makes it easier to keep track of individuals' properties (i.e., when two instances collide, each instance knows who the other is, as well as their respective variables and methods).

## BALL PHYSICS ENGINE AND INPUT INTERFACE

The movement of the ball is directly related to the movement of the FPGA. The game conceptually presents the FPGA as the surface on which the ball is rolling, where increasing the tilt (at any angle) increases the ball's acceleration in the direction of that angle. This allows for greater utilisation of the FGPA, compared to a scenario where the FPGA tilt is simply translated into up, down, left, and right. This is achieved by taking the (filtered) X and Y components of the accelerometer's response to gravitational acceleration and using SUVAT equations to update the velocity and position of the ball. The X and Y components are always independent throughout the game for any calculation. It is only when the GUI is drawn that the emulation of vector addition is achieved from the perspective of the player (since both X and Y components of position are updated). The ball is redrawn and 'teleports' to its new location at every timestep (numerical solution, every *dt*). Due to the high framerate of the game (from the good performance of the system), this is interpreted by a player as motion. An alternative to a numerical solution was to analytically solve for the trajectory of the ball by utilising *scipy.integrate.solve_ivp* using a second order ODE. This would produce a more accurate answer (since the SUVAT equations assume constant acceleration during the timestep) but compromises the system requirements due to the need for more timesteps of data per calculation. Hence, our approach is the most appropriate, as it reduces latency and sacrifices accuracy (which the players won't notice).
The data is sent to the game as hexadecimal. This interface can be manually overridden, where the players can give false hex inputs to the game using the keyboard, if there are not enough FPGAs. The SUVAT equations are modified using the ball's own sensitivity, and the coefficient of friction. When colliding with boundaries, the velocity component perpendicular to the plane of contact is reversed and scaled by the coefficient of restitution. An oversight in the input interface is the fact that the FPGA readings scale by $mgsin(\theta)$, whereas the player might have expected a linear input device. However, a human is able adapt to this instantly, and no testers were able to notice this.

## EDITABLE LEVEL GENERATOR

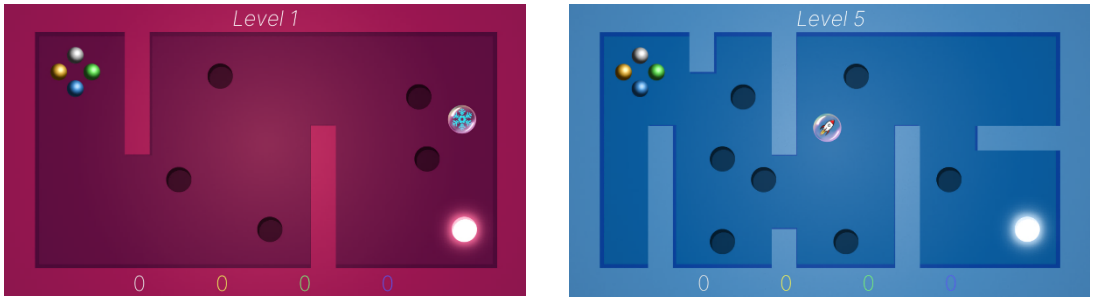


*Figure 3 – Example Levels*

Instead of manually designing each level, which would make everything difficult to change, a level generator was designed that takes in the level colours, coordinates of blocks/holes/powerups and generates the scene with 3D parallax effects included. This implementation allowed for rapid development and decisions at the executive level, without considering low-level implementation. Using flat design not only increases user appeal, but also reduces latency compared to using PNG assets for the scene. Flat design also allows blocks and the frame to overlap without visible seams, making level design even more fluent. Multiple layers of translucent vignettes were combined into one layer to further reduce latency since Pygame calculates alpha on a per-pixel basis.

```
level1_blocks = [(300, 50, 60, 320),
                 (750, 300, 60, 380)]
level1_holes = [(1020, 450), #first hole is always the goal
                (500, 150),
                (400, 400),
                (620, 520),
                (1000, 350),
                (980, 200)]
level1_powerup = [(630, 300), (1080, 250), (350, 550)]
level1powerups = [level1_powerup, random.choice(Powerup.powerupchoices)]
level1_colour = (190,25,90) #Magenta
level1_bg_colour = (120,15,70)
level1_edge_colour = (100,10,60)
level1_goal_colour = (255, 110, 160, 255) #RGBA
level1 = Level(level1_blocks, level1_holes, level1_colour, level1_bg_colo

level2_blocks = [(400, 50, 60, 300),
                 (700, 300, 60, 380),
                 (1000, 200, 300, 60)]
level2_holes = [(1020, 450), #first hole is always the goal
                (120, 300),
                (200, 500),
                (800, 300),
                (1000, 400),
                (1100, 350)]
level2_powerup = [(500, 400), (1000, 300), (1100, 110)]
level2powerups = [random.choice(level2_powerup), random.choice(Powerup.po
level2_colour = (129, 77, 189) #Light Purple
level2_bg_colour = (81, 8, 163) #Dark purple
level2_edge_colour = (60, 2, 127)
level2_goal_colour = (250, 200, 255, 255) #RGBA
```

*Figure 4 – Code to instantiate levels*

```
def draw_parallax (object):
    scale_factorX = (640 - object.rect.center[0]) / 100
    scale_factorY = (360 - object.rect.center[1]) / 100
    scaled = [object.pos[0] + scale_factorX,
              object.pos[1] + scale_factorY,
              object.rect.right - object.pos[0] + scale_factorX,
              object.rect.bottom - object.pos[1] + scale_factorY] #Draws a version of the block closer to the middle
    pygame.draw.rect(screen, Level.active_level.edge_colour, pygame.Rect(scaled[0], scaled[1], scaled[2], scaled[3]))


class Level:

    changing = False
    active_level = None

    def __init__ (self, block_coords, hole_coords, colour, bg_colour, edge_colour, goal_colour):
        self.colour = colour
        self.bg_colour = bg_colour
        self.edge_colour = edge_colour
        self.blocks = []
        for i in range(len(block_coords)):  #Instantiates all blocks
            self.blocks.append(Block(block_coords[i]))
        self.holes = []
        self.holes.append(Goal(hole_coords[0], goal_colour))    #Instantiates goal
        for i in range(1, len(hole_coords)):                #Instantiates all holes
            self.holes.append(Hole(hole_coords[i], goal_colour))

    def draw_frame (self):
```

*Figure 5 – Code to draw parallax effect and level class definition*

## CUSTOMISABLE PARTICLE SYSTEM

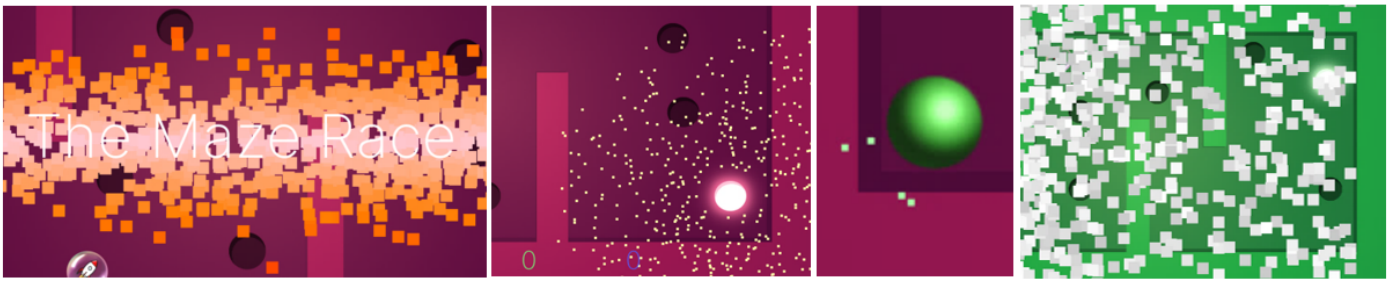


*Figure 6 – Some applications of the particle system: title screen (stream), winner colour (burst), collision sparks (burst), level wipe (transition)*

One of the project extensions was to implement a particle system to add animations to the game. This is not only for aesthetic purposes; the particle colours are used to signal to the other players which balls have completed the maze as they go into the goal. They are also used in a wipe transition between levels. The implementation utilises OOP to spawn up to thousands of particle objects from a particle system. There are three types of systems: stream, burst, and transition. This determines the shape and direction of the flow, alongside other parameters like lifetime, distance, and size. The particle colour can be set to a distributed range of colours or the ball's own colour. A random exogenous force called *veer* is used to redirect particles over their lifetime, simulating turbulent flow.

```
class ParticleSystem ():

    active_systems = []

    def __init__(self, particle_no, colour, lifetime, distance, coords, type="burst", distr="gauss", angle=None, size=5, width=60):
        self.particles = []
        for i in range(particle_no):
            self.particles.append(Particle(colour, lifetime, distance, coords, type, distr, angle, size, width))

class Particle ():
    def __init__(self, colour, lifetime, distance, coords, type, distr, angle, size, width):
        self.randomlifetime = random.uniform(lifetime*0.8, lifetime*1.2)
        self.size = size
        self.magnitude = random.randint(25, 100) * distance
        self.angle = random.uniform(0, 2*math.pi) if angle == None else angle #differentiate between burst and stream
        if type == "stream":
            if distr == "gauss":
                self.pos = [coords[0], random.gauss(coords[1], width)]
            elif distr == "unif":
                self.pos = [coords[0], random.uniform(coords[1]-width, coords[1]+width)]
        else:
            self.pos = [coords[0], coords[1]]
        if colour == "title":
            self.colour = (255, max(0, 200-abs(360-self.pos[1])*0.6), max(0, 210-abs(360-self.pos[1])*1.7)) #title gradient
        elif colour == "transition":
            random_grey = random.randint(200, 255)
            self.colour = (random_grey, random_grey, random_grey)
        else:
            self.colour = colour
        self.vel = [self.magnitude*np.cos(self.angle), self.magnitude*np.sin(self.angle)]
        self.veer = [random.randint(-500, 500), random.randint(-500, 500)] #random turn
```

*Figure 7 – Particle system code*

## POWERUPS

Another project extension was to add powerups. The powerups added are *speedup,* which increases your ball's speed, *freeze*, which stops all the other balls, *invert,* which inverts the controls of the other balls, and *ghost,* which allows your ball to go through the blocks in the map. All of these are on a timer which applies them for a set amount of time, which was fine tuned in testing. Each ball is given its own property for each powerup, to allow a ball to have more than one powerup at once, and if any of those properties is set to true, it alters the *motion_calc* function to apply the relevant alterations to the appropriate ball, modifying the SUVAT equations (however, ghost powerup disables boundary collisions). A design decision made was to only have one powerup available to be picked up on the screen at once, otherwise the level becomes too congested. Once a powerup is collected, it respawns back in after a random amount of time, in a random location (there are set locations per level) and the type that spawns in is random as well. The random aspects were implemented using the *random* library.

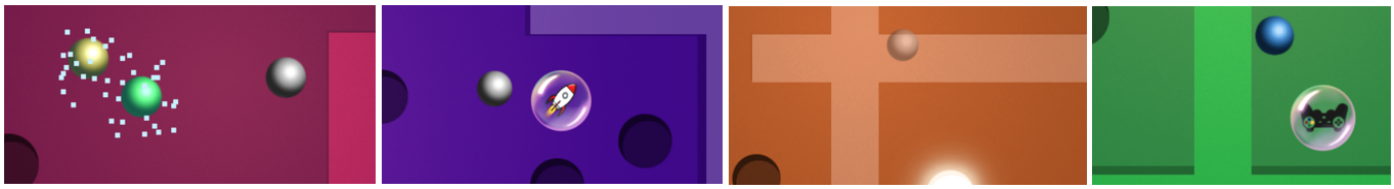

*Figure 8 – Powerups – from left to right: freeze, speedup, ghost, invert*

## DODGEBALL

Whilst creating powerups, an idea that was considered was to add a 'ball killing' powerup. However, once this was implemented, it resulted in greater difficult for the other balls since they were continually reset to the beginning of the level. As this was a very fun aspect of the game, to take full advantage of this idea, a new 'battle-royale' type game mode was created, where the ball with a higher velocity kills the slower moving one. Since a new arena is needed for this, a new dodgeball class was created which inherits the flexible level class to allow for easy arena design. A problem initially encountered in testing was that it was possible to camp the ball respawn location, so, the respawn location was made random to counteract this.

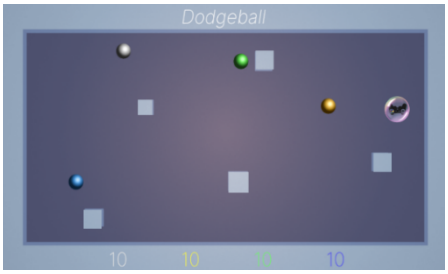


*Figure 9 - Dodgeball*

## SCORING

In The Maze Game, the scores are calculated based on finishing position: 1st place = 40 points, 2nd place = 30, 3rd place = 20 and 4th = 10 points. The powerups allow the player to gain an extra 5 points, whilst falling in a hole costs 5. Once the ball passes through the goal, the ball instance is deleted temporarily whilst it waits for the other balls to finish the level. The level is updated once all the balls pass through the goal (or through manual override). At the end of level 5, final scores are sent to the server. Following this, the server queries the database for high scores, which are sorted and sent to the game to be displayed on the scoreboard. Dodgeball kills have a similar leader board and database structure to scores. There were two different values important to the user in dodgeball: lives and kills. Lives were more important during the game, so that was what is displayed at the bottom. Kills are more important after the game, so they are shown on the leader board. This reduces clutter and enhances the user experience.

# SERVER AND THE DATABASE

## SERVER

A TCP (Transmission Control Protocol) connection was used for the data being sent to and from the FPGA, rather than UDP (User Datagram Protocol). This decision was made because a reliable connection between the server and FPGA was required. Using UDP could result in packet loss, which would mean the controls wouldn't be sent properly from the FPGA to the server and the game wouldn't work reliably. The client code was multithreaded using the *threading* library, in order to concurrently send and receive data with no delay due to waiting. The server code was not multithreaded as it must deal with multiple socket connections and would have introduced timing synchronisation issues. The server also has a 10ms timeout for every socket receive so that any data that takes too long to be received is skipped, which would cause a crash if the server was multithreaded since socket functions are blocking. The server was optimised by adding a check which stops outputting data to the game if there are repeated identical inputs from a socket since the player data will not need to be updated as it stays the same. If 50 empty messages are received from a socket, the server attempts a graceful disconnect because the FPGA has most likely disconnected, and this will remove the ball from the game whilst keeping the game running for the rest of the players. At game start-up the game and each client send an initial message to the server stating "I'm the game/an FPGA" which allows the server to

```
def send_msg ():
        global send_msg, switch
        i = 0
        while True:
                i += 1
                msg = UART()
                if i % 10 == 0: #infrequent sends
                        time.sleep(0.005)
                        send_msg = f"~{ID}," + msg.split()[0] + ":" + msg.split()[1] + ","
                        if msg.split()[3] != "3":
                                send_msg += "buttonpress,"
                        if msg.split()[2] != switch:
                                send_msg += f"switch={msg.split()[2]},"
                                switch = msg.split()[2]
                        try:
                                client_socket.send(send_msg.encode())
                                print(f"sent {send_msg}")
                        except:
                                pass

connection = False
server_name = '3.85.233.169' #"192.168.56.1"
server_port = 12000
print(f"Attempting to connect to {server_name}...")
client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

while connection == False:
        time.sleep(0.005)
        connect()

threading.Thread(target=send_msg).start()
threading.Thread(target=recv_msg).start()
```

*Figure 10 – Client-side code*

distinguish between them. Due to the server timeout, it is possible for the client to send multiple data messages in the same TCP transmission. The format was made to be robust so that multiple messages could be parsed so that the game processes each message separately.

An example format from client 0 to the game would be: ~0,FFFFFFFF:FFFFFFFF,switch=60. An example format from the server to the game would be: ~s,high_scores=0;200;2022-03-19 14:58:47|1;150;202...

```
def recv(socket):
        received = False
        try:
                recv_msg = socket.recv(1024).decode()
                received = True
        except: #This signifies disconnect
                if socket != game:
                        FPGA_empties[FPGAs.index(socket)] += 1 #No message received; suspicious
                        if FPGA_empties[FPGAs.index(socket)] > 50: #At least 50 empties received
                                print(f"FPGA{FPGAs.index(socket)} disconnected")
                                send(game, f"~FPGA{FPGAs.index(socket)} disconnected")
                                FPGAs.remove(socket)
                                socket.close() #Attempt graceful disconnect
                                sockets[sockets.index(socket)] = None #removes socket from list, keeping positions intact
                                if FPGAs == []:
                                        quit() #if last FPGA disconnects, end the server process
                pass
        if received: #If received, parse
                try:
                        if socket != game:
                                FPGA_empties[FPGAs.index(socket)] = 0 #reset empties
                except:
                        pass
                if recv_msg != "":
                        print(f"received {recv_msg}")
                        parse(recv_msg, socket)
        else:
                pass
```

*Figure 11 – Server-Side code*

## DATABASE

There are two similar DynamoDB databases: scores and kills, which are stored on the AWS EC2 instance. The server waits until it receives a message from the game in a specific format. This tells the server that the message is coming from the game instead of the client. If "scores" is found in the message, the *update_scores()* function is called. *update_scores()* captures the date and time for when it received the scores using *datetime.now().isoformat()*. By default, this is represented as YYYY-MM-DDTHH:MM:SS.MS. This format is hard to read, so it was split for the convenience of the player.

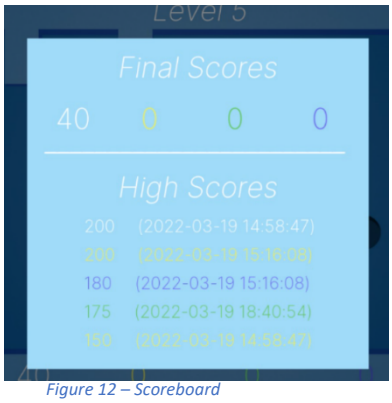


*Figure 12 – Scoreboard*

```
Scores of all players:
[2, 205, '2022-03-21 03:26:14']
[0, 200, '2022-03-19 14:58:47']
[1, 200, '2022-03-19 15:16:08']
[3, 180, '2022-03-19 15:16:08']
[2, 175, '2022-03-19 18:40:54']
[3, 170, '2022-03-21 10:07:59']
[1, 150, '2022-03-19 14:58:47']
[1, 140, '2022-03-21 10:07:59']
[2, 140, '2022-03-19 18:39:29']
[0, 135, '2022-03-19 15:16:08']
[0, 120, '2022-03-21 03:26:14']
[1, 110, '2022-03-21 03:05:51']
[0, 90, '2022-03-21 02:20:17']
[0, 80, '2022-03-21 03:05:51']
[1, 80, '2022-03-19 16:32:19']
[0, 75, '2022-03-20 21:56:55']
[0, 70, '2022-03-20 17:06:09']
[1, 70, '2022-03-21 03:26:14']
[2, 55, '2022-03-21 10:07:59']
[0, 50, '2022-03-19 16:32:19']
[0, 40, '2022-03-21 02:48:30']
[2, 40, '2022-03-19 15:05:28']
```

*Figure 13 – Score table data*

For each player, their final score is updated into the Scores table using the *upload_score()* function. This uses DynamoDB's *put_item* which creates/replaces an item in the table based on its primary key (player). The sort key is the score, and the attributes include datetime.

The scores are sorted using the *get scores()* function where all the scores for the players are loaded and stored into a list (*high_scores*). Each element of the list contains the player, score, and date-time. The list is sorted in descending order, and the top scores are extracted. The *high_scores* are sent back to the game. The same is done for the Kills table, except, instead of score, the number of times a ball has killed another player is loaded. The game displays the current final scores and the high scores on the scoreboard. Many leader boards show the date and times of previous attempts; hence it was decided to include it in our game as well. The positions (1st, 2nd…) are also sent to the FPGA to display on the 7SEG LED on the FPGA. This allows the player to compare their performance with the other players.

```
if recipient == "~s":           #game wants to talk to server only
    if "scores" in msg:
        scores = [int(msg.split('=')[1].split(":")[i]) for i in range(4)]
        UpdateScores.upload_scores(scores)
        high_scores = LoadScores.get_scores()
        for i in range(len(high_scores)):
            for j in range(len(high_scores[i])):
                high_scores[i][j] = str(high_scores[i][j])
            high_scores[i] = ";".join(high_scores[i])
        high_scores = "|".join(high_scores)
        msg = "~s,high_scores=" + high_scores
        sort_order = []
        for i in range(len(scores)):
            sort_order.append([i, scores[i]])
        sort_order = sorted(sort_order, key=lambda x: x[1], reverse=True)
```

```
def upload_score(player, score, datetime, dynamodb=None):
    if not dynamodb:
        dynamodb = boto3.resource('dynamodb', region_name='us-east-1')

    table = dynamodb.Table('Scores')
    response = table.put_item(
       Item={
            'player': player,
            'score': score,
            'datetime': datetime
        }
    )
    return response
```

```
def upload_scores (scores):
    date_time = datetime.now().isoformat()
    date = date_time.split('T')[0]
    time = date_time.split('T')[1].split(".")[0]
    date_string = date  + " " + time

    for i in range(4):
        resp = upload_score(i, scores[i], date_string)
        print(f"Uploading {i} {scores[i]} {date_string}")
```

*Figure 14 – Database Scores.*

# HARDWARE IMPLEMENTATION

## FPGA FUNCTIONALITY

The hardware structure was designed in Quartus using the platform designer tool. All inputs and outputs were declared in a top-level Verilog file and the pins for the LEDs, buttons, switches and JTAG UART were assigned using a mixture of a *.qsf* file and *.bdf* file. The calculations and processing of data on the FPGA were done through a C file that could be run on the FPGA using Eclipse. In the code, there is an infinite while loop that will continue to run if the player is still connected to the game. First it displays the number of lives the player has, before obtaining the accelerometer data for the X & Y axis.

The accelerometer data is then passed through a filter. This aims to make the movement in the game smoother as it reduces the spikes in input when a small unintentional twitch or movement is done by the player. However, latency is still a factor. Too much processing can cause significant lag which may result in delayed movement. Taking this into consideration, a 5-tap filter was chosen as this allows for enough data processing without introducing a significant amount of latency. Multiple filters were tested, and a 5-tap filter was finalised as it had the best smoothness-latency compromise.

After this, the code then obtains the current positions of the switch and whether a button is being pressed. All the required data that has then been collected on the FPGA is then sent to the client. This is then printed to the NIOS II terminal and sent to the client using the *alt_putstr()* function. Once the data is sent, the FPGA then checks for any response from the client. This was done using the *read()* and *fcntl()* functions included in the *<unistd.h>* and *<fcntl.h>* headers respectively.

Initially, the code was using the *read_chars()* function defined. However, after further testing, flickering of the LEDs and slight delays in the Hex displays indicated that it was causing a significant amount of delay inside the while loop and was negatively affecting performance of the game. This was primarily due to the function using *alt_getchar()* in a loop which would create a hold until the next character was received. This hold was causing delays which resulted in a large amount of input lag and LED flickering.

In order to solve this, an alternative *read()* function was used. The function allows the programmer to specify the length in bytes of the message that is to be read from the uart. As the length of the messages the client sent to the FPGA was fixed, the specification means that the code can immediately progress as soon as the specific amounts of bytes arrive and does not need to create a hold and wait for the next characters. This significantly lowered the time it took for a loop, the input lag in the game and flicking of the LEDs on the FPGA. The data from the FPGA would either indicate a message to be displayed on the hex displays and cycled or would reduce the LED by 1 if a life was lost by the specific player.

| | |
|---|---|
| Flow Status | Successful - Mon Mar 28 21:56:08 2022 |
| Quartus Prime Version | 18.0.0 Build 614 04/24/2018 SJ Lite Edition |
| Revision Name | puzzle |
| Top-level Entity Name | puzzle |
| Family | MAX 10 |
| Device | 10M50DAF484C7G |
| Timing Models | Final |
| Total logic elements | 2,428 / 49,760 ( 5 % ) |
| Total registers | 1376 |
| Total pins | 185 / 360 ( 51 % ) |
| Total virtual pins | 0 |
| Total memory bits | 535,552 / 1,677,312 ( 32 % ) |
| Embedded Multiplier 9-bit elements | 0 / 288 ( 0 % ) |
| Total PLLs | 0 / 4 ( 0 % ) |
| UFM blocks | 0 / 1 ( 0 % ) |
| ADC blocks | 0 / 2 ( 0 % ) |

*Figure 15 – Quartus timing analysis.*

## TESTING APPROACH

Testing the FPGA was done using a client-side file which sent certain instructions to be outputted on the FPGA. Testing was important to identity the correct speeds to rotate the text as well as how many rotations were required. Furthermore, testing for edge cases was done such as whether the hex displays would be overwritten when a new command was sent from the client.

Module testing was used to test the game. The project was split into three sections: the FPGA, software and server. All three sections were developed and tested individually using simulated inputs and outputs that would be expected from the other modules once all sections were integrated.

- For the software, testing was done by binding keys to simulated accelerometer inputs. This allowed for extensive testing of the functionality of the game mechanics as well as making sure all output messages to the server and client were working without having the FPGA available.
- Iterative testing was preformed for each new feature of the game. Furthermore, testing was done on previous features to make sure they were not hindered by new implementations.
- For the FPGA, a substitute host.py file was created that continually outputted commands to the FPGA, as well as reading accelerometer data. This facilitated testing of all commands that would be later sent from the game running on the client. It also confirmed that data was successfully being sent over the JTAG UART.
- In terms of the server, a substitute client.py file was created which had random data signals being sent to the server and checked if the received data was correct. This also allowed for analysis of latency so that the game could be built around a base amount of lag due to the speed of data transfer between the game and the server.

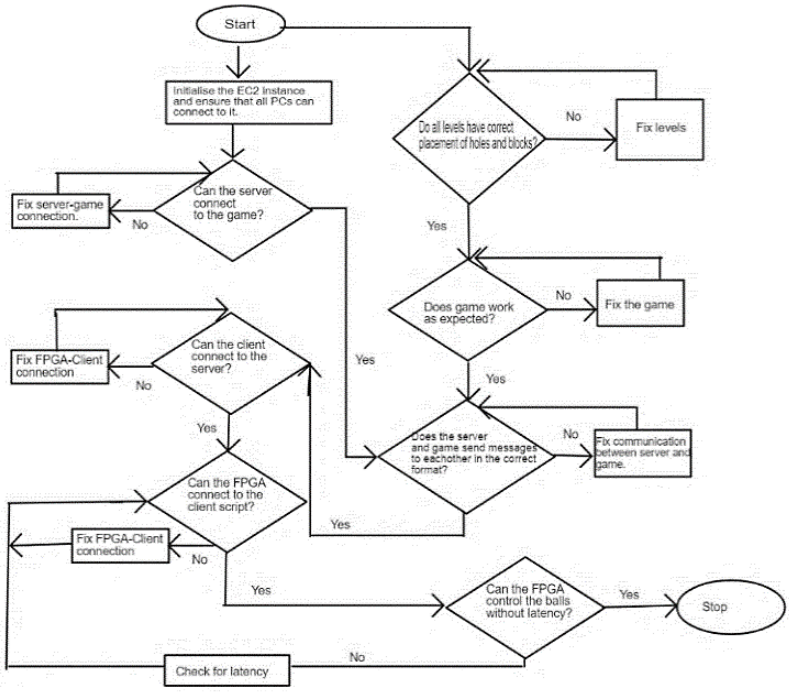


*Figure 16 – Testing flow diagram*

Once all modules were put together, testing proceeded using exhaustive testing and then edge case testing (e.g., testing client disconnect)

- Firstly, testing was done repeatedly to exhaustively test all possible variations of inputs and outputs from the FPGA. All button and switch sensitivity variations were tested as well as the hex displays and the reduction of lives in the dodgeball minigame.
- After this, edge case testing was done to make sure that no unexpected scenarios would arise during gameplay. This included testing such as overwriting the current LED screen, making sure the players did not go out of bounds and having multiple actions happening simultaneously to measure input lag during periods of time with more processing.

## PERFORMANCE METRICS

### TIMING ANALYSIS

| Process | Average amount of time taken |
|---|---|
| Time between frames rendered | 41ms |
| Sending from FPGA to client | 622ms |
| Sending from client to FPGA | 1341ms |
| Pinging server (client to server to game) | 57ms |

*Table 1 – RTT between different nodes.*

As seen in *Table 1*, it takes a total of 720ms on average for FPGA input to be received by the game. This latency is hidden since the balls accelerate slowly, and the 41ms rendering time gives an average frame rate of around 24.5fps. FPS measurements are important because the FPS needs to be above the threshold where the 'teleporting' movement of the balls appears smooth (~15FPS). The large latency for the FPGA receiving data is not an issue since it happens very infrequently, mainly at the start of the game to communicate ball colour, and the end to receive positions.