Technical Details

Architectural Description

This section explains the implementation logic behind key game details. The flowchart depicting gameplay is shared at the end of this document.

Game Flags

The **Game active flag:**

- Set after the preparations (drawing game elements) are done.
- That is, the flag is set before entering the game loop
- Reset after the player either wins or loses
- The game loop is exited when this flag is reset

The **Paddle free flag:**

- Can be set only after the game's active flag is set
- This flag is set when the user presses the space bar
- The game loop can only progress if the paddle free flag is set
- Paddle movement keys are only processed if the paddle free flag is set
- This flag is reset upon losing a life

Display Resolution

The setup resolution for the game is 256x256 with a unit width and height of 2. This display configuration translates to a 128x128 resolution from a memory perspective.

Keyboard Input

Keyboard interrupts are used to move the paddle and start the game. Interrupt-based inputs provide smoother gameplay, as the program does not need to stop (for ball movement) to process user input.

Influence of flags on keyboard input is discussed above, and visualized in the flow chart.

Bricks

Each brick is a rectangle, 10-unit pixels wide and 5-unit pixels high. The brick dimensions are stored in memory and not hard-coded.

The bricks are arranged in an 8x4 grid. The number of rows and columns is not hard-coded and stored in memory.

The bricks are right next to each other in both directions. Because of this design choice, we only need the top-left coordinates of the grid (or the top-left coordinates of the top-left brick). The coordinates will be stored in a table as a single row.

Each row will have a different color. A color table, such as taught in lectures, will be used to cycle through the colors. A color index entry will be made in the brick layout table.

Bricks table structure:

Top-left X coordinate, Top-left Y coordinate, Color index

Each entry is a word.

- X coordinate of a brick= word at (table + 0) + (Column index * Width of brick)
- Y coordinate of a brick = word at (table + 4 bytes) + (Row index * Height of brick)
- Color index of row = 6 and then previous color index -1

The bricks are drawn in a nested loop.

Borders

The game is contained within 3 borders (no horizontal border at the bottom). Each border is white.

There are 3 rows in the border table: two for the vertical side boundaries and one for the horizontal top boundary. A row entry from the table:

Top-Left X coordinate, Top-Left Y coordinate, Length of line

Game Paddle

The game paddle is a 13x2 rectangle. Its dimensions will be stored as a single row table.

Paddle table structure (planned):

Top-Left X coordinate, Top-Left Y coordinate, Old top-left X coordinate, Old top-left Y coordinate, Width, Height, X-axis velocity, X-axis direction

The paddle's position is updated according to keystrokes (A or D), and therefore its coordinates are updated in the keyboard handler.

In user space code, the paddle is redrawn only if its current and old coordinates do not match.

Game Ball

The game 'ball' is a square for easier collision calculations. It has no acceleration but will have a 2D initial velocity. The ball's X-axis speed will change between [2, 4] depending on where it hits the paddle. Its speed at game start or after losing life will be initialized to 2. The ball's initial X-axis direction is randomized to be left or right at the beginning of the game or after losing a life. The ball's Y-axis speed is fixed to 1.

A non-zero X-axis velocity component is necessary; otherwise, the ball may only go to and from along the Y-axis.

Game ball table structure:

Top-Left X coordinate, Top-Left Y coordinate, Old top-left X coordinate, Old top-left Y coordinate, Size, X-axis velocity, X-axis direction, Y-axis velocity, Y-axis direction

The ball will always be white.

Detecting Collisions and Actions to Take

The ball may collide with:

- Borders
- Paddle
- Brick(s)

Border Collision Detection

The ball may collide with the border by its side edges (on the side border) or at the top edge (on the top border).

- The ball has collided with the left border if the top-left X coordinate of the ball <= the left border X coordinate.
- The ball has collided with the right border if the top-left X coordinate of the ball right >= the right border X coordinate.
- The ball has collided with the top border if the top-left Y coordinate of the ball <= the top border Y coordinate.

Action to Take on Border Collision

- If the ball collides with the left border, reverse the ball's X-axis direction and update the ball's current X coordinate to the border's X coordinate + 1
- If the ball collides with the right border, reverse the ball's X-axis direction and update the ball's current X coordinate to the border's X coordinate – ball size - 1
- If the ball collides with the top border, reverse the ball's Y-axis direction and update the ball's current Y coordinate to the border's Y coordinate + 1

Paddle Collision Detection

The ball's bottom edge may collide with the paddle's top edge.

There must be X and Y axes overlapping between the paddle and the ball.

- If Ball bottom < Paddle top, no collision
- If Ball top > Paddle bottom, no collision
- If Ball right < Paddle left, no collision
- If Ball left > Paddle right, no collision

Otherwise, there is a collision.

- If Ball left > (Paddle left + Paddle width/2), collision on right side
- Else, collision on the left side

Action to Take on Paddle Collision

On collision with the paddle, the Y-axis direction of the ball is reversed. The ball will therefore go up towards the bricks.

The ball's X direction is:

- Set to left if it hits the left side of the paddle
- Set to right if it hits the right side of the paddle

The ball’s X-axis speed is calculated using linear interpolation.

Linear interpolation formula: y = y1 + ((x - x1) * (y2 - y1)) / (x2 - x1))

- y = New X speed
- y1 = Ball min X speed
- y2 = Ball max X speed
- x = offset (between ball and paddle center) = Ball center X – Paddle center X
- x1 = start of input range = start of paddle from either side = 0
- x2 = end of input range = paddle width / 2
- Absolute values of offset will be used because direction is handled elsewhere

New X speed = Ball min X speed + (|Offset| * (max X speed - min X speed)) / (Paddle width / 2)

Brick(s) Collision Detection

Spatial hashing is used to identify which brick in the grid the ball may be colliding with. The layout of the grid, the coordinates of game elements (bricks and ball), and the dimensions of the elements are known

- Column index = (Ball X - Grid X) / Brick Width
- Row index = (Ball Y - Grid Y) / Brick Height

Only the brick at that row index, column index is selected for checking.

Collision conditions:

- Ball left edge >= Brick right edge, no collision
- Ball right edge <= Brick left edge, no collision
- Ball top edge >= Brick bottom edge, no collision
- Ball bottom edge <= Brick top edge, no collision

Otherwise, there is a collision.

If there is a collision, the side on which the ball collided with the brick matters:

- The ball’s top edge may collide with a brick’s bottom edge.
- The ball’s bottom edge may collide with a brick’s top edge.
- The ball’s left edge may collide with a brick’s right edge.
- The ball’s right edge may collide with a brick’s left edge.

Calculate overlap in X direction = min(Ball right, Brick right) - max(Ball left, Brick left)

Calculate overlap in Y direction = min(Ball bottom, Brick bottom) - max(Ball top, Brick top)

- If X overlap < Y overlap, the collision is on the left or right side
- If Y overlap < X overlap, the collision is on the top or bottom side

Compare ball center Y to brick center Y to determine top vs bottom.

- If Ball center X < Brick center X, the ball hits the left side
  - Else, the ball hit the right side
- If Ball center Y < Brick center Y, the ball hit the top side
  - Else, the ball hit the bottom side

Action to Take on Brick(s) Collision

If the hashing logic returns a row and column index where a brick does not exist, we return from the function.

- Row index can't be < 0 and > 4 for collision
- Column index can't be < 0 and > 8 for collision

If the brick at (Row index, Column index) is destroyed, return from the function.

Otherwise, check if there is a brick that can be collided with:

- If there is not, return from the function
- Otherwise, mark the brick as destroyed by drawing a black brick over it and update the memory

After determining the side of the collision, update the X and Y directions of the ball:

- Collision with the top of the brick:
  - Reverse the Y-axis direction if the ball is not moving up
- Collision with the bottom of the brick:
  - Reverse the Y-axis direction if the ball is not moving down
- Collision with the right side of the brick:
  - Reverse the X-axis direction if the ball is not moving right
- Collision with the left side of the brick:
  - Reverse the X-axis direction if the ball is not moving left

Instructions, Scoreboard, and Life Count

The instructions are displayed at the beginning of the game or when the user restarts after losing a life.

The scoreboard will be displayed on the right side of the game area. Lives left will be displayed on the left side of the game area.

Letters and numbers will be displayed on the Bitmap display to achieve this functionality.

Losing a Life (and the Game)

The player has 5 lives. The player loses a life if the ball falls beyond the paddle. That is:

- If Ball bottom > Paddle bottom, life lost

The above condition is checked during paddle collision

The player loses after losing all 5 lives.

Winning the Game

The player wins when all bricks are destroyed, and the score reaches 32.

Drawing Elements

For the initial setup, the bricks, borders, paddle, and ball need to be drawn in their initial position.

During gameplay, due to the large number of elements on the screen, it is essential to limit redrawing elements to those affected during gameplay.

- **Bricks** are redrawn with the color black when they are destroyed. They are not cared for again.
- **Borders** are never redrawn.
- The **Paddle** is redrawn if its new X coordinate does not match its old X coordinate. The old paddle will need to be erased first.
- The **Ball** is redrawn if its new X and Y coordinates do not match its old coordinates. The old ball will need to be erased first.
- **Score** and **lives** messages are updated; their values change

Writing Instructions and Game Over Messages

The instruction to launch the ball – PRESS SPACE – is displayed whenever the ball needs to be launched (at the start or after losing a life.

The instructions to move the paddle left: <-A – and to move the paddle right: D-> - are displayed after the user has pressed the space bar.

GAME LOST! – will be displayed when the player loses all lives.

GAME WON! - will be displayed when the player destroys all bricks.

Updating and Displaying Lives

Lives will be stored in memory as a number.

A single character, null-terminated string (so two characters) will be used to store the ASCII character version of the lives count. 0x30 will be added to convert the number to an ASCII character.

The lives count will be updated when the player loses a life.

Updating and Displaying Score

The score will be stored in memory as a number.

A two-character, null-terminated string (so three characters) will be used to store the ASCII character version of the score.

The score is split into digits using the following steps:

- Score/10
- Contents in LO register = Ten's place digit
- Contents in HI register = Unit's place digit

Audio (Collision and Game End)

All game audio plays at maximum volume (volume 127) and use MIDI instrument number 118 (synth drum). A short beep lasting 90 milliseconds occurs in these collision situations:

• Paddle collision: Pitch = 84

• Border collision: Pitch = 78

• Brick collision (destruction): Pitch = 96

Each of these sounds is triggered by syscall 31, so the system doesn't wait for the sound to finish before moving on with the code.

A brief musical segment plays when:

• Game is lost – a 3-note tune

• Game is won – a 5-note tune

Both the winning and losing tunes loop and use syscall 33 to ensure they play as intended.

Build Plan and Verification:

This section discusses the building and verification plan for my breakout game.

(Build) Update Logic to Calculate Pixel Address

The address calculation logic was updated to work with 128x128 resolution displays.

(Verify) Draw a Dot Using New Pixel Address Logic

A dot was drawn at a random coordinate and color to verify the working of the pixel address calculation.

(Verify) Draw Lines and Boxes

Lines and boxes (squares) were drawn to verify that the functions worked as expected.

(Build) Make the Border Data Table and Write Functions to Draw Borders

A table to store border data was made according to the description in the previous section.

A procedure was written to draw the borders (two vertical borders and one top horizontal border) using data from the row entries of the border data table.

(Verify) Draw Borders on Screen

The borders were drawn on the screen to test if the X and Y coordinates and the length of each border were satisfactory. Border data values were updated accordingly.

(Build) Make the Bricks Data Table, Define Brick Dimensions, and Write Functions to Draw Bricks

A table to store brick data was made according to the description in the previous section.

A procedure was written to draw the bricks in a grid using the X and Y coordinate logic and color index logic described in the previous section. Another function to create a single brick was first written. This procedure is repeatedly called to draw all bricks in the brick grid.

(Verify) Draw Bricks on Screen

The brick data table, brick dimensions, and brick grid value coordinate logic were tested by drawing the brick grid.

(Build) Make the Ball Data Table and Procedures to Draw and Erase the Ball

A ball data table was made according to the table description in the previous section.

Though called a ball, it is a square (box).

Procedures for drawing and erasing the ball were created. Both used the same function for drawing a box, but utilized ball data from memory and had white and black as default color values, respectively.

(Verify) Draw the Ball (Box) on the Screen

The ball was drawn on the screen. A 0.5-second timer started using syscall 32, after which the ball was erased. After another 0.5-second timer, the ball was redrawn. This was done to verify the ball's coordinates and draw/erase functions.

(Build) Collision Logic for Bricks and Border

Collision logic for the ball, bricks, and borders could be implemented now.

The collision procedures were written once and modified repeatedly to get improved performance. Marking bricks as destroyed was also handled here.

Temporary: Since there is no paddle, an invisible border was created at the bottom for the ball to bounce off.

(Verify) Simple Game Loop

To verify the working of the ball-related procedures and collision logic, a simple game loop was created:

- Erase the ball
- Update the ball position (move the ball)
- Check for border collision
- Check for brick collision
- Draw the ball
- 33ms timer

(Build) Paddle Movements and Keyboard Interrupt Logic

The game active flag is set after the initial elements (bricks, borders, the ball, and now the paddle) are drawn on the screen. Keyboard interrupts are checked only if the game's active flag is set.

When the user presses the space bar after the game's active flag is set, the paddle's free flag is set. When the paddle free flag is set, the paddle can be moved. Additionally, the ball is launched only if the paddle free flag is set.

The paddle's X position updates on keyboard strokes (A and D) were handled in the interrupt logic itself.

Separate procedures were developed to draw and erase the paddle when necessary.

(Verify) Paddle Movements and Keyboard Interrupt

The game loop was modified to wait for the paddle free flag to be set. When set, the ball would start moving on its own, and the paddle could be moved between the borders using the A and D keys.

(Build) Instructions, Scorecard, Lives Left, and Messages

The digit character table was updated to support instruction messages. Procedures were created to draw the score, lives left, game start, and paddle movement messages and instructions.

(Build) Implement Paddle Collision, Score Update, and Lives Lost Logic

The score update logic was added in the bricks' collision function.

A new procedure was developed to detect and act on paddle collision. This new function handled the losing lives logic, as well.

(Verify) Update Game Loop and Trial Run

Since all major elements are developed, the final game loop logic was implemented:

- Check if game is active -> End if not
- Erase the ball
- Update the ball position (move the ball)
- Check for border collision
- Check for brick collision
- Draw the ball
- Draw the paddle
- Check for paddle collision
- Check if the player lost a life. If yes:
    - Update lives left
    - Game ends if all lives are over -> Display game lost message
    - Otherwise, reset ball and paddle parameters
    - Wait for the user to resume by pressing space
- Show the score (the procedure will only redraw the score if the score has changed)
- Check if all bricks are destroyed (score = 32)
    - If yes, display game won message and end game
- 33ms timer

(Build) Add Ball X-Axis Speed Logic

Implement linear interpolation logic to change the ball’s X-axis speed on paddle collision.

(Verify) Trial Run to Understand Speed Change

Run the game to understand if the ball’s speed changes on paddle collision.

(Build) Randomize Ball’s X-axis direction

The ball’s initial X-axis direction at the beginning of the game or after losing a life is randomized using syscall 42.

(Verify) Trial Run to Test Random Ball Direction

Run the game to understand if the ball's initial direction is different than the initial value declared in memory.

(Build) Game Audio

Incorporating audio involved defining pitch values and their durations in memory, as well as selecting the instrument and volume. The synth drum provides a retro feel, similar to the original game. Using maximum volume allows the user to hear the intended sound level from their speakers. Notes and durations were chosen through trial and error.

(Verify) Game Audio Working on Events

Run the game again to verify if the correct audio is being played on the required events.

Debug/Issues

Selecting Brick to Investigate for Collision

My initial approach to checking whether the ball had collided with a brick was to check each brick. This brute-force approach required nested loops and unnecessary checks on every iteration of the game loop.

This approach was replaced with my version of spatial hashing. Because I know the layout of the grid, the coordinates of game elements (bricks and ball), and the dimensions of the elements, spatial hashing replaced the $O(N^2)$ approach with an $O(1)$ check.

- Column index = (Ball X - Grid X) / Brick Width
- Row index = (Ball Y - Grid Y) / Brick Height
- Row index can't be < 0 and > 4 for collision
- Column index can't be < 0 and > 8 for collision

Only the brick at that row index, column index is selected for checking

Determining the Side of the Brick the Ball Collided With

The ball can collide with any of the 4 sides of the brick. The ball's X and Y directions need to be changed depending on the side of the ball hit.

The overlapping logic, where corner collisions are classified into side collisions, is the easiest to understand and implement.

Better techniques in AABB and SAT exist, and though implementation is easy, their understanding is difficult.

Original Position of Ball and Paddle

After losing a life, the ball must be restored to its original location. I forgot to create this field in memory, causing unexpected behavior during gameplay. The ball's X-axis speed and Y-axis direction must be restored as well.

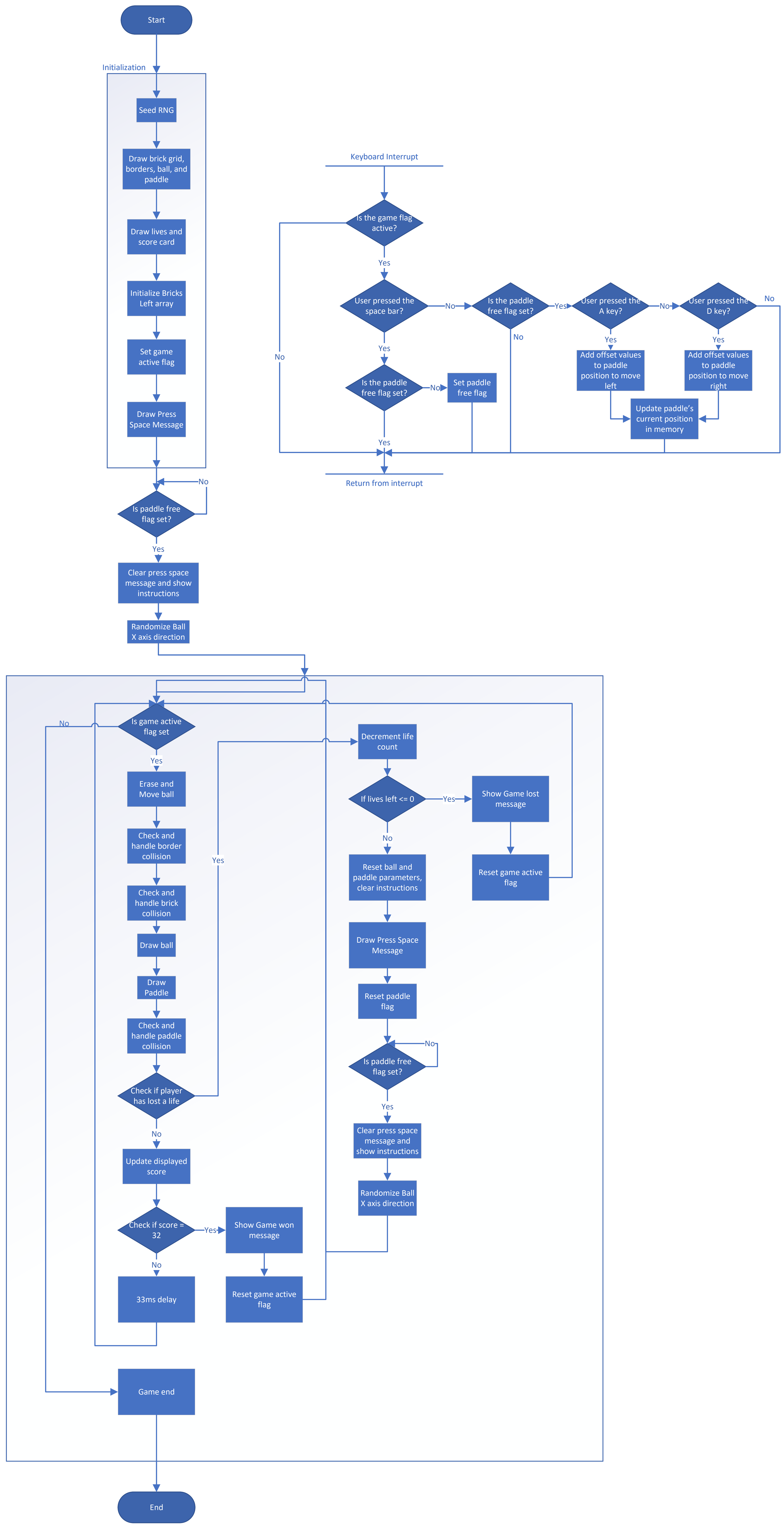
Similarly, the paddle must be restored to its original location. Otherwise, the newly placed ball will fall again.

Ball Speed

An introductory calculus topic, revisiting linear interpolation to modify the ball’s speed, was an interesting application and a better approach than increasing ball speed by one every time it collides with the paddle. Restoring ball speed and Y-axis direction after losing a life became a requirement after it was implemented. X-axis speed limits (minimum and maximum) became a requirement.

Double Condition Check for Paddle Free Flag

Loading the paddle free flag from memory, which was being modified in the interrupt handler, caused a data hazard that couldn't be fixed using no-operations (nop). Instead, the load instruction for retrieving the paddle free flag and the condition to check if it is 0 had to be duplicated to produce the expected results (only start or resume the game if the user presses the space bar).


Start
Initialization
Seed RNG
Draw brick grid, borders, ball, and paddle
Draw lives and score card
Initialize Bricks Left array
Set game active flag
Draw Press Space Message
Is paddle free flag set?
No
Yes
Clear press space message and show instructions
Randomize Ball X axis direction
Keyboard Interrupt
Is the game flag active?
Yes
No
User pressed the space bar?
No
Is the paddle free flag set?
Yes
User pressed the A key?
No
User pressed the D key?
No
Yes
Add offset values to paddle position to move left
Yes
Add offset values to paddle position to move right
Update paddle's current position in memory
No
Yes
Is the paddle free flag set?
No
Set paddle free flag
Yes
Return from interrupt
Is game active flag set
No
Yes
Erase and Move ball
Check and handle border collision
Check and handle brick collision
Draw ball
Draw Paddle
Check and handle paddle collision
Check if player has lost a life
Yes
No
Update displayed score
Check if score = 32
Yes
Show Game won message
Reset game active flag
No
33ms delay
Game end
End
Decrement life count
If lives left <= 0
Yes
Show Game lost message
Reset game active flag
No
Reset ball and paddle parameters, clear instructions
Draw Press Space Message
Reset paddle flag
Is paddle free flag set?
No
Yes
Clear press space message and show instructions
Randomize Ball X axis direction