Project Preparation Activity 2 [30 marks]

Watch Briefing Video! **Not An Autograded Evaluation**

You must submit all the source codes to the Git Repository by the PPA2 deadline.

IMPORTANT: **Your submission must compile without syntactic errors to receive grades. Non-compilable solutions will not be graded.**

About PPA 2

- Use the following link to accept the PPA 2 invitation: https://classroom.github.com/a/w2Z86J2v
- PPA 2 code submission is due on **Sunday, October 13, 2024, at 11:59 pm** on Github.
- PPA 2 in-person interview and demonstration is due in the lab session immediately after the code due date.

About Project Preparation Activities in General

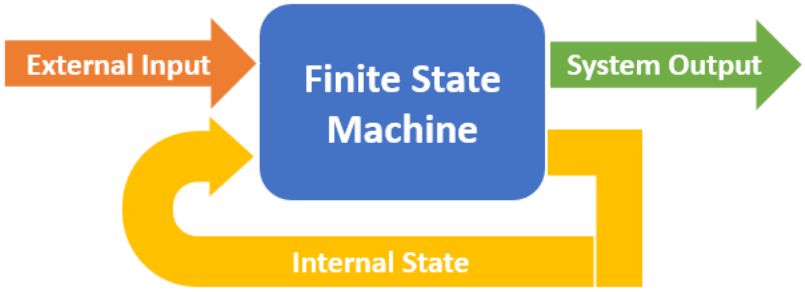
- The non-autograded programming activities that contribute to the cumulative knowledge and programming experience required to complete the COMPENG 2SH4 course project.

- Evaluation Format
  - More instructions will be provided in the lectures and/or tutorials.
  - 5 to 10-minute feature-based demonstration and interview in your designated lab session.
  - Knowledge-based evaluation will be carried out at the in-person cross examinations on the designated dates (keep an eye on announcements).

- **Note: You must provide your student information (Section, Mac ID, and Student ID) in the README file of the PPA 2 code package. Failure to do so will result in a 5% penalty on the entire PPA 2 grade.**

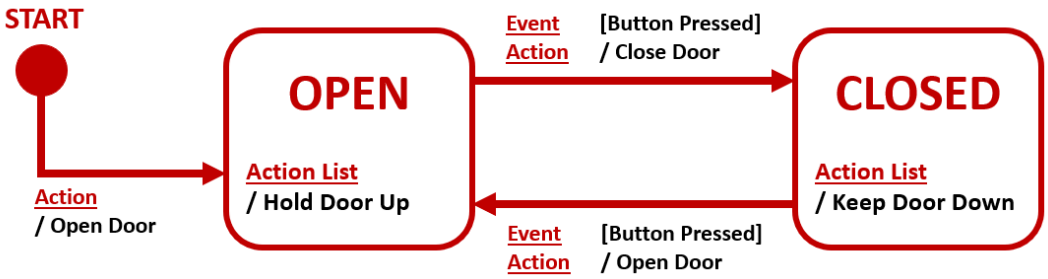
Introduction to a Simple Finite State Machine

The core concept in this PPA is one of the most popular engineering design techniques widely used in both computer hardware and software – **Finite State Machine** (FSM). In short, an FSM is a set of logics / algorithms that continuously update their output based on 1) the most recent input, or lack thereof, and 2) the state the FSM is in (aka. The mode of operation the FSM is in).



In our daily life, FSMs are everywhere. A low-cost automatic garage door, with one button and one door, is an FSM. Depending on the state of the door, a press on the button results in different actions taken by the door motor. When the door is in OPEN state and the button is pressed, the door shall take the action of *closing down*, and consequently transition into CLOSED state; when in CLOSED state and the button is pressed, the door takes the action of *opening up*, and transitions into OPEN state. Same input but different states lead to different actions.

This simple FSM behaviour can be illustrated using the ***State Diagram*** as illustrated below.



Start Gate (Solid Dot)
- The starting point of the FSM. Typically, this is where the FSM starts functioning when booted up.
- The first state pointed by the Start Gate transition is the "Default State".

State Bubbles (Large Bubbles with State Names and Action Lists)
- **The FSM modes of operation**. The FSM remains in the same state unless a relevant event (input) takes place.
- Sometimes, the system may take repeated actions in every program loop iteration when staying in the same state.

State Transitions (Arrows Connecting Bubbles)
- Showing how FSM transitions between states, and its corresponding triggering events.
- Sometimes, the system may take additional actions when the state transition happens.

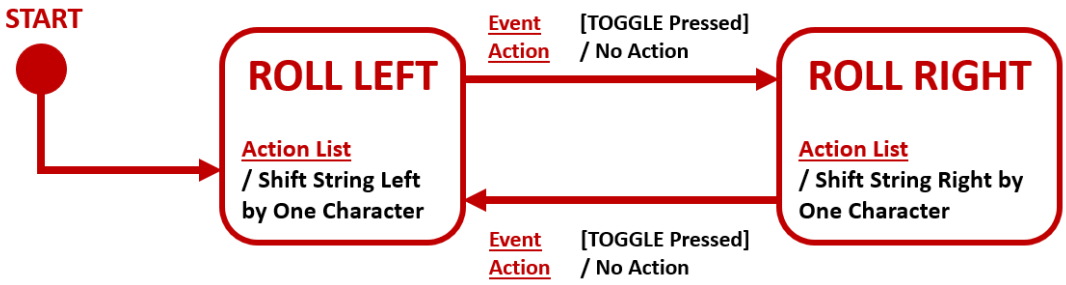
Events (Description in the Square Brackets)
- **The input of the FSM**. The relevant event that triggers the FSM transition.

Action List (Description with Leading Slashes)
- **The output of the FSM**. These can be repeating actions within a state, or one-time actions during transition.
- Can be omitted if no actions are to be performed.

**** This is NOT an exhaustive list of State Diagram features.**

The FSM design principle is used whenever our system requires different modes of operations. It provides an effective and sustainable code organization framework, so that logics within each state can be developed and debugged almost independently of each other. It is such a powerful design principle that even the Unity Game Engine has adopted it for intuitive animation control interface. A well-designed FSM can contain hundreds or even thousands of states, producing some level of pre-programmed artificial intelligence behaviours (but with limited self-learning capabilities).

Of course, FSM is not mentioned here without a context. We will learn to deploy our first FSM in PPA2 to control the movements of an object in a 2D coordinate system. To help you understand how we can turn an FSM diagram into a set of usable code, here is one possible FSM visualization of the movement portion of the marquee display in PPA1:



There are many ways to deploy an FSM in code. We will use the enumeration + switch-case method. In future design courses, you will have opportunities to learn about more efficient FSM implementation methods.

Here are the steps:

1. Identify the State Bubbles, and create a C enumeration in the global scope as follows:

```
enum FSMMode {LEFT, RIGHT};
enum FSMMode myFSMMode;
```

2. In the initialization routine, set the default state, and carry out required operations to initialize the FSM. This is the implementation of Start Gate and its Action List to transition into the Default State.

```
void Initialize()
{
    ...
    myFSMMode = LEFT;
    // No other FSM initialization required
    ...
}
```

3. Once entering the program loop, your FSM is in action.
    a. In the input collection routine, collect input the same way as in PPA1. This is the source of Events.

    b. In the main logic routine, implement the following switch-case statements after the input processing step. Each case is your State Bubble, and all the Action Lists in the state bubble translates into execution statements within the case. Note that these actions will be repeated every program loop iteration if no state transition takes place.

```
switch(myFSMMode)
{
    case LEFT:
    default:
        displayPosition--;     // Shift String Left by 1 Character
        ...                    // Add other repeating actions for LEFT
```

```
            break;

        case RIGHT:
            displayPosition++;     // Shift String Right by 1 Character
            ...                    // Add other repeating actions for RIGHT
            break;
}
```

Notice that the default case is added along with LEFT state, so to ensure that your FSM can be kicked back into a functional state when malfunctioning.

c. Under each case, put down a set of if-statements to check the input; if input matches a certain condition, put down actions to change state. This is the State Transition.

```
switch(myFSMMode)
{
    case LEFT:
    default:
        if(input == '=')       // My TOGGLE input, trigger state transition
        {
            myFSMMode = RIGHT;    // Will enter RIGHT state in next loop
            ...                   // Other actions for state transition
        }
        displayPosition--;     // Shift String Left by 1 Character
        ...                    // Add other repeating actions for LEFT
        break;

    case RIGHT:
        if(input == '=')       // My TOGGLE input, trigger state transition
        {
            myFSMMode = LEFT;     // Will enter LEFT state in next loop
            ...                   // Other actions for state transition
        }
        displayPosition++;     // Shift String Right by 1 Character
        ...                    // Add other repeating actions for RIGHT
        break;
}
```

In the subsequent PPA2 tasks, use the above example to implement a state diagram design and then to translate it into corresponding C code.

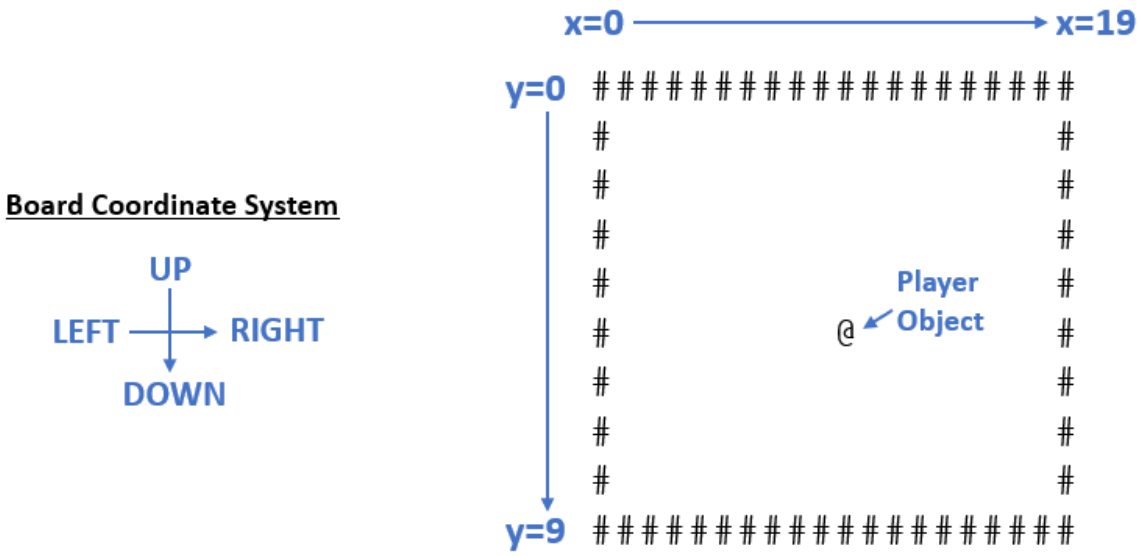
Project Preparation Activity 2 – Create a Game Board with Free-Moving Player Object

Through PPA1, we now have learned how to animate something on the console screen, as well as using a program loop to organize code in an effective way. With the debugging techniques covered in the lecture, we are now ready to take on a more interactive program design with asynchronous input.

This activity is critical to your success in PPA3, the 2SH4 project, and many subsequent activities in COMPENG 2SI3. As a result, please make sure to take your time and master the contents in this activity.

In PPA2, let's build a **20x10** (Columns x Rows) **Game Board Area and a Player-Controllable Object** with the following features:

- Upon startup, a stationary player object is placed in the centre of the game board, represented by a special ASCII character of your own choice ('@' in the example below).
- The border of the game board is represented by another special character of your own choice ('#' in the example below).
- The game board coordinate system should be set up as illustrated below. **Hint**: If you use the nested *for-loop* for row-scanning correctly, the coordinate system should naturally make sense without additional codes.



Example Figure: *20x10 game board with boundary characters and player object*

- Capture the 'W', 'A', 'S', 'D' control keys asynchronously as you've learned in PPA1, and use the four keys for commonly known directional control (think about case-sensitivity or case-insensitivity):
  - W Upward Direction
  - S Downward Direction
  - A Leftward Direction
  - D Rightward Direction
- Upon the first pressed directional key, the player object must move in the designated direction continuously, 1 unit per game loop cycle. The object must stay on course with the movement direction until the next valid direction control key is pressed.
  - If moving UP or DOWN: Only LEFT and RIGHT move command is allowed
  - If moving LEFT or RIGHT: Only UP and DOWN move command is allowed
- When the player object reaches the board boundary, it must wrap around the border and come in from the opposite border. For example, if the player reaches the left border, it must come back out from the right border in the next game loop cycle. **Hint**: the logic is the same as the marquee display wraparound in PPA1.

- An exit command is deployed to end the program at any time.

- Advanced Features (Above and Beyond Activities)
    - Implement a set of keys that can dynamically adjust the overall game speed.
        - Provide usable instructions and current game speed below the game board.
        - Choose at max 5 levels of sensible game speed. Don't run the game too fast to the point that the player object becomes impossible to control.

Use the sample executable **PPA2.exe** for deliverable references.
For above and beyond activities, check out **PPA2-Above.exe** sample executable for reference.

Marking Scheme [Total 30 marks]

- Basic Features **[Total 25 marks]**
    - **[2 marks]** Correct game board and player object printout.
    - **[3 marks]** Correct implementation of WASD four-direction control (or own choice of keys).
    - **[12 marks]** Correct movement FSM implementation in all four directions (3 marks per direction).
    - **[8 marks]** Correct movement wraparound in all four directions (2 marks per direction).

- Advanced Features **[Total 5 marks]**
    - **[3 marks]** Correct implementation of the speed control adjustment logic.
    - **[1 mark]** Speed adjustment instructions and the current game speed is well displayed on the screen.
    - **[1 mark]** Sensible choices of 5 game speed levels.

Recommended Workflow

Again, in all the PPAs, a recommended workflow will be provided to help you get familiar with the incremental software development procedures. However, as you may notice in a moment, PPA2 recommended workflow contains less hand-holding guides, but more hints and general algorithms to provoke your programmer's mind. For every design requirement you come across in this workflow, take your time to think about what to do. Draw your concepts on paper, and then test your code out incrementally with the debugging techniques covered in class.

Remember, unless you are an experienced programmer, **DO NOT** attempt to develop this program in one shot. Always, **ALWAYS** make sure the current feature is working as intended before adding more features to prevent creating a debugging nightmare.

Get familiar with this workflow! Iterative design strategy is an absolute must as you progress on to higher level software courses. We will also incrementally reduce the number of hand-holding tips as we progress into later labs / project preparation activities under the assumption that you are more experienced with software development practices.

Iteration 1 – Build the Game Board and Player Object

- First, let's warm up by putting together a static frame on the screen displaying the contents shown in the Example Figure.
- Drawing this frame is rather trivial.
    - Use nested for-loops covered in class to print the required characters at the corresponding positions.
    - **Hint**: if you set up the loops correctly, the coordinate system illustrated in the Example Figure will turn out to be extremely intuitive.
- **STOP!** As a good incremental engineering practice, we suggest you to first confirm that you can draw the static frame shown in the Example Figure before proceeding forward.

- The previous step is trivial, but the underlying feature is not. There are two distinctively different objects in the Example Figure. We will have to treat them differently in the code implementation.
    - Static Game Board Border
        - The coordinates of these border characters remain the same in every animation frame.
        - In the nested for-loop drawing routine, we can set up some if-conditions to draw the boundary characters at the correct coordinates. They stay the same throughout the rest of the code development.
    - Dynamic Player Object
        - The coordinates of this player object will change in every frame. We need to know, and ultimately calculate, the new coordinates of the object in every program loop iteration before we draw the object on the screen.
        - Therefore, the x-y coordinate of the player object **MUST BE PARAMETERIZED** using a set of coordinates.
        - Define the following struct in your program, so you can create the player object using this struct to conveniently track its coordinates and its ASCII character symbol. This will be extremely helpful in the subsequent 2SH4 project activities. [**STOP**: Watch the supporting video on C Struct and Enumerations before proceeding further.]

```
struct objPos
{
    int x;              // x-coordinate of an object
    int y;              // y-coordinate of an object
    char symbol;        // The ASCII symbol of the object to be drawn on the screen
};
```

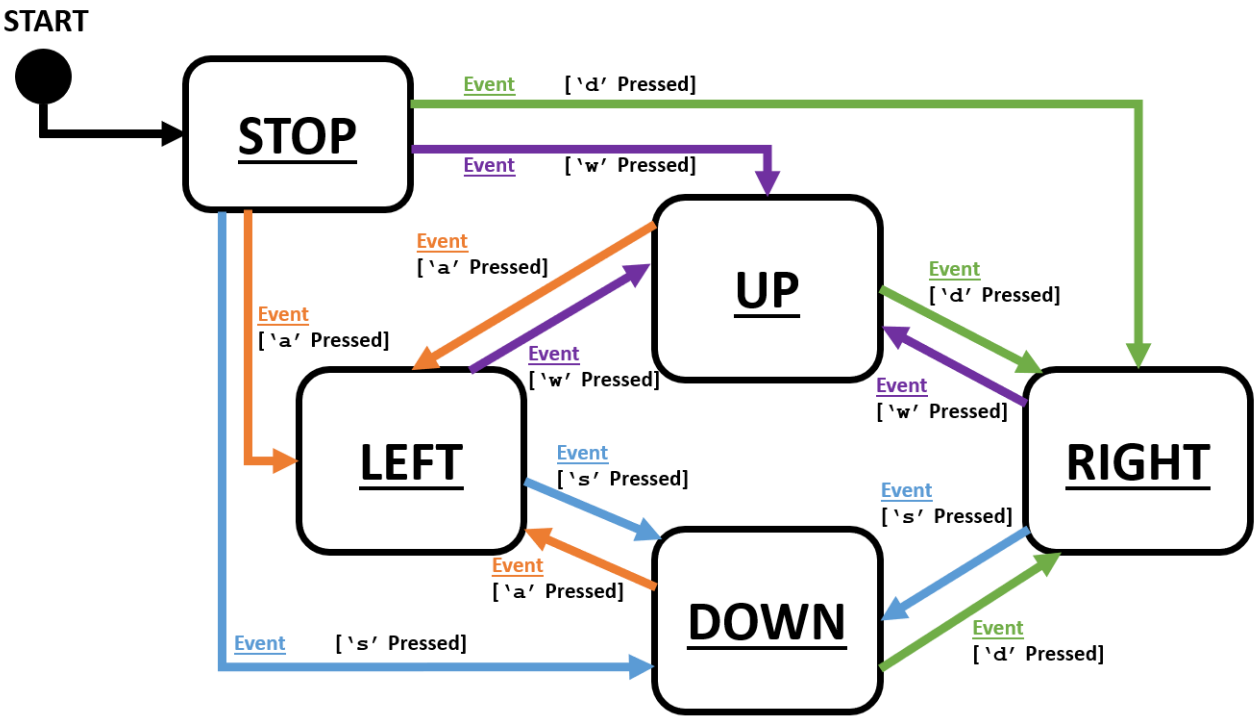
        - In the drawing routine, draw the player symbol when arriving at the player object coordinate.

    - Modify your drawing routine to make sure you can reproduce the same static frame display as shown in the Example Figure but using the `objPos` struct to track the player position and symbol instead.

- **STOP!** Make sure you can draw the frame correctly with parameterized player position before proceeding to the next iteration.

Iteration 2 – Implement the Player Direction FSM with Asynchronous Input

- To enable 4-direction object movement animation in a 2-D coordinate system, we will need to deploy the directional control keys as described before. We will use the well-known WASD control – but you can change it to anything as you wish.
  - First, collect the asynchronous input using `MacUILib_hasChar()` and `MacUILib_getChar()`.
  - Make sure you identify the following keys: W, A, S, D, and a non-alphanumerical key for ending the program.
  - If you forgot how to do this part, review PPA1.
- Next, we need to create the ***Input Processing*** Logic by linking the WASD control with our Player Direction FSM as illustrated in the State Diagram below. Read the *Introduction to FSM* section (and watch the FSM support video) to learn how to interpret the State Diagram and convert it into code.

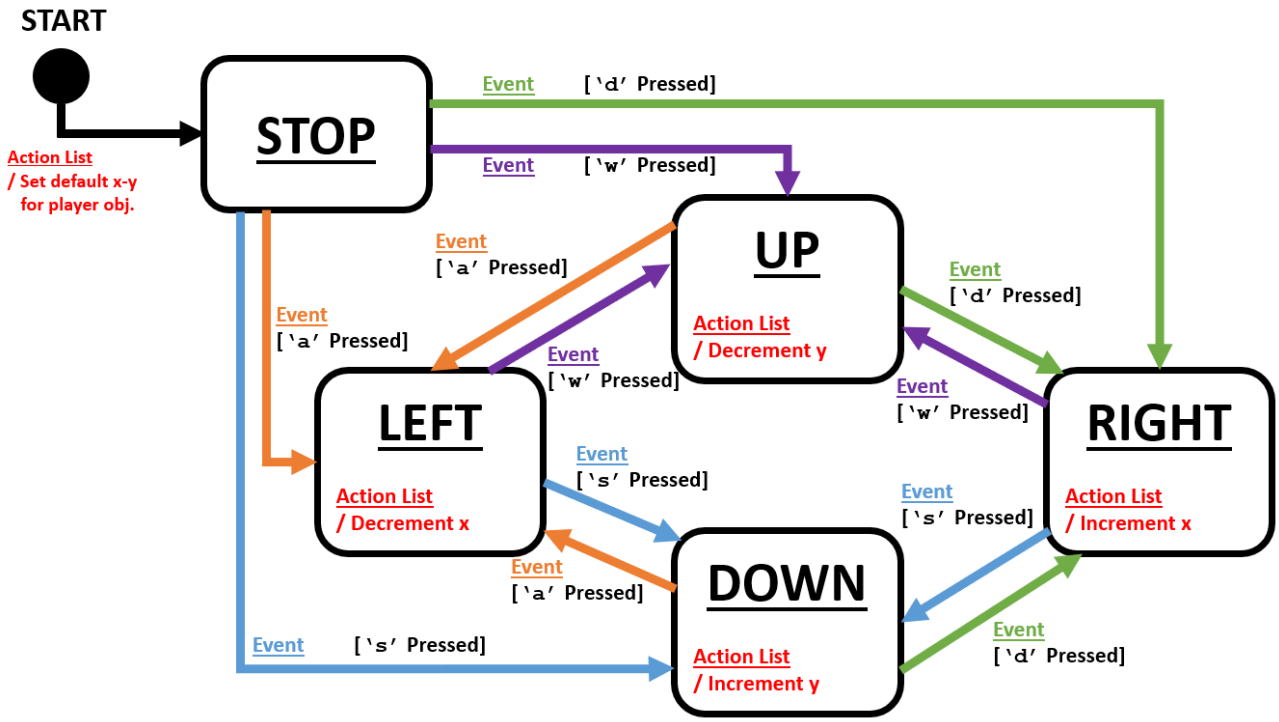


  - Remember to set up the **state enumeration** as a good self-documenting coding style.
  - Link the WASD control to the FSM as specified in the deliverable list. As a reminder:
    - If moving UP or DOWN: Only LEFT and RIGHT move commands are allowed
    - If moving LEFT or RIGHT: Only UP and DOWN move commands are allowed
  - For debugging purposes, set up a debugging message below the game board to display:
    - Key Pressed
    - Current State of the FSM
  - Think about where this part of the code should reside in.
    - GetInput()? Or RunLogic()? Or somewhere else?
    - **Critical Thinking** – Should the input processing logic run on every loop, or only when there is an input?

- **STOP!** Test your WASD control and the FSM until they work correctly before proceeding to the next iteration.

Iteration 3 – Update Player Object Position based on Current States

- With the input processing and FSM state update now completed, we can deploy the Player Position Update logic using the FSM state. Consider the more detailed FSM diagram below and pay attention to the **Red Action Lists** in the state bubbles. These actions in the bubbles should be repeated in every program loop.



- Implement the player position update logic in the suitable routine.
  - You should choose the starting position of the player object near the center of the game board, and a slower game speed (larger delay constant) for debugging convenience.
  - Make sure you are consistent with the board coordinate system when incrementing / decrementing player object coordinates. Do not worry about hitting the boundary of the game board for now.
  - Since you have already parameterized player object coordinates in Iteration 1, changing the player coordinates should immediately reflect on the animation through the Draw routine.

  - **STOP! You don't have to implement all four movement directions in one shot. Tackle them one at a time.**
  - For every implemented direction update, test it and validate it against the FSM state and the key pressed.
    - Recall from Lecture – See It before Believing It!

  - **STOP!** Continue testing the direction update logic until you are confident that all movement requirements are met.

Iteration 4 – Deploying Boundary Wraparound

- **This section requires you to do some algorithm design and refinement yourself.**
  Only the general concept will be provided for you to build upon and develop further.

- "Wraparound" happens when the player object hits the game board boundary and comes out on the opposite boundary. This is used to confine the player object movement within the prescribed game area.
  - When hitting the top boundary, the player object should come out from the bottom boundary, and vice versa.
  - When hitting the left, it should come out from the right, and vice versa.
- Think about how you can achieve the wraparound behaviour by modifying the existing Player Position Update logic from Iteration 3.
- Heed the Boundary Characters! Your player object should NOT go into the boundary; instead, it should wrap around at the boundary characters.
- Tips for Better Debugging:
  - Print out the player coordinates in the debugging message.
  - Slow down the game so you can observe and study the behaviour of the player object when wrapping around.
  - When badly stuck with a bug, use the VSCode IDE Debugger as discussed in class. Don't shoot in the dark.

- Once this part is completed, your program should behave identically as PPA2.exe (sample executable), and ready for submission.
  - You can optionally remove the debugging message below the game board.

- You are now ready to take on your very first interactive game / software design task in PPA3.

Additional Iterations – Advanced Features (Above and Beyond Activities)

- **Only conceptual hints will be provided for above-and-beyond activities.**
  You need to apply the knowledge and skills from the previous design iterations to come up with additional features.

- **Above-and-Beyond Feature – Adjusting Game Speed at 5 Different Levels on the Fly.**
  - Think about how you can parameterize the game speed and define 5 different speed levels.
  - Think about how to enable in-game speed changes using key commands.
  - For good UI design, provide instructions and speed information on the screen.