Table of Contents

[I. Introduction 3](#_Toc137022089)

[II. Design and Implementation 3](#_Toc137022090)

[A. Key data structures and their roles 3](#_Toc137022091)

[B. Implementation of data structures 3](#_Toc137022092)

[C. Description of the data structures used 3](#_Toc137022093)

[1) Stack: 3](#_Toc137022094)

[2) Linked List: 3](#_Toc137022095)

[3) Recursive Function: 4](#_Toc137022096)

[D. Considerations for choosing the data structures 4](#_Toc137022097)

[III. Game Mechanics 4](#_Toc137022098)

[A. Overview of the game rules and mechanics 4](#_Toc137022099)

[IV. Variables and structures used and their functionality 4](#_Toc137022100)

[V. Important functions used in program 5](#_Toc137022101)

[1. The “createWindow()” 5](#_Toc137022102)

[2. The “set\_mainScreen1()” 5](#_Toc137022103)

[3. The “updateWindow()” 5](#_Toc137022104)

[4. The “winScreen()” 6](#_Toc137022105)

[5. The “setName()” 6](#_Toc137022106)

[VI. Important classes and their respective functions 6](#_Toc137022107)

[A. Stack 6](#_Toc137022108)

[1. The constructor “stack()” 6](#_Toc137022109)

[2. The “void set()” 6](#_Toc137022110)

[3. The “void push()” 6](#_Toc137022111)

[4. The “int pop()” 6](#_Toc137022112)

[B. Player 6](#_Toc137022113)

[1. The “setN” 7](#_Toc137022114)

[2. The “setName” 7](#_Toc137022115)

[3. The “checkWin” 7](#_Toc137022116)

[4. The “setup” 8](#_Toc137022117)

[5. The “setHomeNodes” 8](#_Toc137022118)

[6. The “getout” 8](#_Toc137022119)

[7. The “newGoti” 8](#_Toc137022120)

[8. The “availableGoti” 8](#_Toc137022121)

[9. The “resetGoti” 8](#_Toc137022122)

[10. The “displayGotia” 8](#_Toc137022123)

[C. The list class 9](#_Toc137022124)

[1. The “setXandY” 9](#_Toc137022125)

[2. The “create\_nodes” 9](#_Toc137022126)

[3. The “update\_coordinates” 9](#_Toc137022127)

[4. The “side\_nodes” 9](#_Toc137022128)

[5. The “list” constructor: 9](#_Toc137022129)

[6. The “MakeMove()” function: 9](#_Toc137022130)

[7. The “SetHomeNodes()” function: 10](#_Toc137022131)

[VII. Graphics and User Interface 10](#_Toc137022132)

[A. Overview of the graphical window and user interface: 10](#_Toc137022133)

[B. Explanation of how graphics were integrated into the game: 10](#_Toc137022134)

[C. Usability and Effectiveness of the User Interface: 10](#_Toc137022135)

[VIII. Testing and Validation: 11](#_Toc137022136)

[IX. Conclusion 11](#_Toc137022137)

[X. CODE 13](#_Toc137022138)

[XI. OUTPUT SCREEN PICTURES 14](#_Toc137022139)

[A. The main screen 14](#_Toc137022140)

[B. Game Board 14](#_Toc137022141)

[C. The ending screen 23](#_Toc137022142)

[XII. Disclaimer and Copyright: 24](#_Toc137022143)

# Introduction

Ludo is a classic board game that originated in India and has been played for centuries. It was created in the 6th century and has since become a popular game worldwide, known for its simple yet strategic gameplay.

Developing a Ludo game as a data structures project is an interesting endeavor as it allows for the implementation of fundamental data structures and algorithms to simulate the game's mechanics. Despite the existence of numerous online Ludo games for both PC and mobile devices, creating a customized version provides an opportunity to explore and showcase one's programming skills and creativity.

# Design and Implementation

In order to create a functional and efficient Ludo game, various data structures were utilized. Each data structure played a crucial role in implementing different aspects of the game.

## Key data structures and their roles

The key data structures used in the Ludo game were the stack, linked list, player array, and list (for handling pawn movements). The stack was responsible for managing the pawns waiting to enter the game and handling pawn resets. The linked list played a vital role in representing the board and enabling pawn movement. The player array stored information about each player, including their names and positions. The list data structure was utilized to handle pawn movements and ensure they followed the game rules.

## Implementation of data structures

The stack data structure was implemented using an array-based approach, where the pawns were pushed and popped from the array. The linked list was implemented by creating a structure or class that represented each node in the list, with pointers connecting them in a circular manner. The player array stored instances of a player class or structure, containing relevant information about each player. The list data structure for handling pawn movements was implemented using appropriate functions and algorithms to traverse the linked list based on the dice roll.

## Description of the data structures used

Following is the description of each data structure used

### Stack:

The stack data structure was used to manage the pawns of each player that were waiting to come out onto the board. This stack stored the pawns in a specific order, and when a new pawn needed to be brought into play, it was popped from the stack. Additionally, if a pawn was killed, it was pushed back into the stack. The stack simplified the process of handling pawn movements, as it provided a straightforward way to reset a pawn's state and display it in the home area.

### Linked List:

A basic linked list was employed to represent each box on the Ludo board. This linked list served as a container for storing information about each box, such as whether it was a safe zone or connected to a player's home nodes. The linked list was circular in nature, allowing pawns to move continuously in a loop around the board. Moreover, the linked list incorporated a side node pointer variable, which facilitated the formation of four separate chains representing the home nodes of each player. The linked list acted as the backbone of the game, enabling smooth movement of pawns across the board by traversing the list based on the number rolled on the dice.

### Recursive Function:

A recursive function was employed to create the chain of nodes for the Ludo board. This recursive approach provided a simple and efficient method for generating the required nodes. By using recursion, the creation of the board nodes became more manageable and allowed for easy customization of the board's layout.

## Considerations for choosing the data structures

The choice of data structures was driven by the need for efficient pawn management, smooth board navigation, and easy implementation of game rules. The stack was selected for its ability to handle the dynamic nature of pawn entry and exit. The linked list was chosen due to its circular structure, enabling continuous movement of pawns around the board. The player array provided a convenient way to store and access player information. Lastly, the list data structure was ideal for managing pawn movements, as it offered a straightforward method for traversing the board nodes based on the dice roll. These data structures were carefully selected to ensure a well-structured and enjoyable gaming experience.

# Game Mechanics

## Overview of the game rules and mechanics

The core gameplay mechanics are implemented within the main function through the use of a while loop that continues until only one player remains to win the game. The number of players can be chosen at the start of the game.

The main function begins by rolling a dice. If a player has a pawn outside their home, it checks if the dice roll resulted in a 6. If it did, the user is prompted to decide whether to bring a new pawn into play. If the user declines, the program proceeds to find the pawn on the board that can be moved. If there are multiple pawns available, the user is asked to specify which pawn to move. However, if the player does not have any pawns outside their home, it checks if the dice roll was a 6. If it was, a new pawn is automatically brought into play. Otherwise, the current player's turn is skipped, and the turn is passed to the next player.

The code operates according to this logic, utilizing various functions to execute the necessary actions.

# Variables and structures used and their functionality

The code defines several variables and structures that play a crucial role in the implementation of the game.

The variables “X\_PIXELS” and “Y\_PIXELS” represent the length and width of the graphics window, respectively, defining the dimensions of the game's visual interface. The variable “length” represents the length of the game board.

The variables “hx”, “hy”, “vx”, and “vy” are used to determine the indices of the first horizontal and vertical rectangles on the board. These variables are utilized to create the entire grid structure of the game. The variable “slx” represents the length of small rectangles on the board, used to position and align the elements of the game visually.

The arrays “CircleX” and “CircleY” store the x and y coordinates, respectively, of the circles for each player. These coordinates are crucial for displaying and positioning the pawns of each player on the game board when the pawns are in home.

The variable “TP” represents the total number of players participating in the game. It keeps track of the number of players in the game. The variable “winNum” keeps track of the number of players who have won the game.

The code also defines two important structures: “node” and “goti”. The “node” structure represents a node in the tree/linked list structure of the game board. It contains various fields such as “pl”, which stores the player number whose pawn is located on the node; “id”, which represents the pawn ID; “pakka”, a boolean variable indicating if the node is a safe zone; “x” and “y”, storing the coordinates where the pawn on the node will be displayed; “next”, a reference to the next node in the sequence; “side”, a reference to the side chain for home entries; and “h”, which stores information about the player's home entries connected to the node.

The “goti” structure represents a pawn in the game. It contains fields such as “player”, indicating which player the pawn belongs to; “status”, representing the current location status of the pawn (home, on the board, in home entries, or completed journey); “x” and “y”, storing the coordinates where the pawn will be displayed; “id”, representing the pawn ID; and “location”, indicating the node on which the pawn is currently positioned.

These variables and structures are fundamental to the implementation of the game and play essential roles in storing and managing game-related data, tracking player progress, and visual representation of the game elements.

# Important functions used in program

### The “createWindow()”

This function is responsible for creating the game window with a specified length and width. It sets up the graphical interface where the game will be displayed. This function initializes the necessary components to create a visual environment for the game.

### The “set\_mainScreen1()”

This function is responsible for setting up the initial main screen of the game. It displays the game board, player pawns, and other graphical elements on the screen. This function ensures that the game is visually appealing and provides an intuitive interface for the players.

### The “updateWindow()”

This function is crucial for updating the game window based on the current state of the game. It refreshes the screen, making sure that any changes in the game, such as pawn movements or player turns, are reflected on the graphical interface. This function continuously updates the game display, allowing players to see the latest game state.

### The “winScreen()”

This function handles the end of the game when a player wins. It displays a victory message and relevant information, such as the player's name and winning statistics. This function provides a satisfying conclusion to the game, allowing players to celebrate their victory.

### The “setName()”

This function is responsible for collecting and storing the players' names at the beginning of the game. It prompts each player to enter their name, which is then stored for future reference.

# Important classes and their respective functions

## Stack

The “stack” class is used to store player pawns before they are brought into the game. It serves as a container for the pawns, allowing them to be easily managed. The stack maintains a list of pawn IDs and their corresponding coordinates for display purposes. It also keeps track of the top position in the stack.

The “stack” class has several functions to manipulate the stack:

### The constructor “stack()”

The constructor initializes the stack by setting the top position to -1, indicating an empty stack.

### The “void set()”

This function sets the initial coordinates for the pawns in the stack, which are displayed in the home area. It takes parameters “cx” and “cy” representing the center coordinates of the home area and “off” as the offset used to calculate the coordinates of each pawn.

### The “void push()”

This function adds a pawn to the top of the stack. It takes a reference to a “goti” object as an argument, representing the pawn to be pushed. The function increments the top position, assigns the coordinates and status ('H' for home) to the pawn, and stores the pawn ID in the “gotiId” array.

### The “int pop()”

This function removes and returns the ID of the pawn from the top of the stack. It decrements the top position and retrieves the ID from the “gotiId” array before returning it.

The stack class provides a convenient and organized way to handle player pawns that are not yet in play or have been killed during the game. It ensures that pawns can be easily managed and displayed correctly in the home area.

## Player

The “player” class represents a player in the game. It encapsulates the player's pawns, home stack, and various variables used for tracking the player's progress and status. The class also provides functions to set player details, manage pawns, check for a win, and display the player's pawns on the board.

The “player” class has the following variables:

1. “stack home”: A stack object that represents the home area where the player's pawns are stored before they enter the game.
2. “int cx, cy”: Integer values representing the center coordinates of the player's home area, used by the stack to calculate home coordinates.
3. “node \*homeNode”: A pointer to the node where the player's pawns will be inserted when they come out of the home area.
4. “int out”: An integer indicating the number of pawns currently on the board.
5. “int pid”: An integer representing the player's unique ID, which can also be used as an index.
6. “bool win”: A boolean variable that indicates whether the player has won the game.
7. “int place”: An integer indicating the player's position in the game (e.g., first, second, third, fourth).
8. “int nx, ny”: Integer values representing the coordinates used for displaying the player's data at their respective positions.
9. “int kalar”: An integer representing the color used by the player (e.g., red, yellow, blue, green).
10. “string name”: A string storing the name of the player.
11. “goti Goti[4]”: An array of “goti” objects representing the player's pawns.

The “player” class provides the following functions:

### The “setN”

This function in the “player” class sets the coordinates for displaying the player's details on the game board. It takes two integer parameters “xx” and “yy”, which represent the x and y coordinates, respectively. These coordinates determine where the player's information will be displayed on the screen.

### The “setName”

This function is used to set the player's name. It takes a string parameter “n”, which represents the player's name. The function assigns the value of “n” to the “name” variable of the player object, storing the player's name for future reference.

### The “checkWin”

This function is called whenever a player's pawn reaches the home area. It takes two integer parameters “a” and “b”, which are used for double buffering to display messages. The function checks if all of the player's pawns have reached the home area, indicating a win. If a win is detected, the function updates the “win” variable to “true”, increments the “winNum” variable, sets the “place” variable to the current win position, and displays a message indicating the player's win status on the screen.

### The “setup”

This function is responsible for setting up the player. It takes an integer parameter “ID” representing the player's unique ID. Within the function, the “pid” variable is set to the value of “ID”. The function also sets the “cx” and “cy” variables, which represent the center coordinates of the player's home area. The “home” stack is initialized with these coordinates, and the player's pawns are pushed into the stack. Additionally, the function determines the color (“kalar”) associated with the player based on their ID.

### The “setHomeNodes”

This function sets the home node for the player's pawns. It takes a reference to a “node” object “h” and assigns it to the “homeNode” variable. This allows the player's pawns to be placed in the correct node when they enter the home area.

### The “getout”

This function returns the number of pawns that are currently on the board. It simply returns the value of the “out” variable, which keeps track of the number of pawns outside the home area.

### The “newGoti”

This function is used to bring a new pawn into play. It increments the “out” variable to indicate that a pawn has entered the board. The function pops a value from the “home” stack and assigns it to the “id” variable. The pawn's location, coordinates, and status are updated, and the player and pawn references are stored in the respective nodes. The pawn's status is set to 'R' to indicate that it is in play.

### The “availableGoti”

This function determines which pawn(s) can be moved on the board. If there is more than one pawn available, it prompts the user to select a pawn to move. The function displays the available pawn choices on the screen and waits for user input. The chosen pawn's ID is returned as the result.

### The “resetGoti”

This function is called when a player's pawn is killed. It decrements the “out” variable to indicate that a pawn has left the board. The pawn's location and status are reset, and the pawn is pushed back into the “home” stack. This allows the pawn to re-enter the game later.

### The “displayGotia”

This function is responsible for displaying the player's pawns on the board. It iterates through each pawn and determines the appropriate coordinates and offsets for displaying them. The pawns are drawn as circles with the player's color, and their IDs are displayed inside the circles. The function takes into account any overlapping pawns and adjusts the offsets accordingly to ensure proper visualization.

The “player” class provides the necessary functionality for managing the player's pawns, tracking their progress, and displaying them correctly on the game board.

## The list class

The “list” class represents a linked list that is used to create a game board with nodes. The class has a protected member variable “head” that points to the first node in the linked list. It also contains two integer arrays, “x” and “y”, each with a size of 15. These arrays are used to calculate and store the coordinates of each node, which are then used by pawns.

### The “setXandY”

This function is responsible for setting the x and y coordinates of the nodes. It uses a loop to iterate through the arrays and calculate the coordinates based on a given formula. The function takes into account specific adjustments for certain indices in the arrays to ensure accurate positioning. The calculated coordinates are stored in the respective arrays for later use.

### The “create\_nodes”

This function is a recursive function that creates a given number of nodes and connects the last node back to the head. It takes two parameters, “cur” (a reference to the current node) and “n” (the number of nodes to create). If “n” is greater than 0, the function creates a new node, initializes its member variables, and recursively calls itself with “cur->next” and “n-1”. Once “n” reaches 0, the function sets “cur” to point to the “head” node, thus completing the circular linked list.

### The “update\_coordinates”

This function is responsible for updating the x and y coordinates of each node in the linked list. It iterates through the list and assigns the precalculated coordinates from the “x” and “y” arrays to each node. The function follows a specific pattern to determine the coordinates for each node, based on its position in the list.

### The “side\_nodes”

This function adds additional nodes to the linked list, which represent home entries. These nodes give the linked list tree-like properties. The function also sets the x and y coordinates for each of these nodes. It iterates through the list and checks for specific indices to determine where the side nodes should be added. For each corresponding index, the function creates the required number of nodes, assigns their coordinates, and connects them to the main list.

### The “list” constructor:

This constructor initializes the “head” pointer to “NULL” and then calls several other member functions. It first calls the “setXandY()” function to set the “x” and “y” coordinates of the nodes. It then calls the “create\_nodes()” function to create 52 nodes in the linked list. After that, it calls the “update\_coordinates()” function to update the coordinates of the nodes. Finally, it calls the “side\_nodes()” function.

### The “MakeMove()” function:

This function is responsible for moving a pawn in the game. It takes several parameters, including the player ID (“pl”), the pawn ID (“id”), the number of moves (“move”), and references to “a” and “b”. Inside the function, it performs various operations to update the location of the pawn and handle different game scenarios. It removes the player ID and pawn ID from the current node, updates the pawn's position based on the number of moves, checks if the pawn has entered the home, and updates the window display. It also handles double buffering for smooth animation using the “a” and “b” variables. Additionally, it checks if the new box is not a safe box and if it contains pawns belonging to other players. If so, it kills those pawns. The function also handles moving pawns that are already inside the home entries.

### The “SetHomeNodes()” function:

This function sets the home nodes for all players and marks certain nodes as safe boxes. It iterates through the linked list and sets the home nodes for each player at specific positions (e.g., node 1 for player 0, node 14 for player 1, etc.). It also marks nodes 9, 22, 35, and 48 as safe nodes by setting the “pakka” flag to “true”.

These functions work together to manage the movement of pawns in the game and define the home nodes and safe boxes for the players.

Overall, the “list” class provides functionality for creating and managing a linked list with nodes that represent positions on a game board. The class sets the coordinates of each node, connects nodes in a circular manner, updates coordinates, and adds side nodes for home entries. These functions work together to create a structure that can be utilized for implementing game logic and pawn movement.

# Graphics and User Interface

## Overview of the graphical window and user interface:

The graphics and user interface in the game are implemented using the graphics.h library of C++. To create the graphical window, the system metrics are obtained, allowing the game to adjust based on different screen sizes. Rather than manually setting coordinates for everything that appears on the screen, variables and calculated coordinates are used. A common method used is the percentage method, where the coordinates are calculated as a percentage of the screen size. For example, to position a circle at the center of the screen, the X-coordinate would be calculated as X\_PIXELS \* 0.5 and the Y-coordinate as Y\_PIXELS \* 0.5. This approach ensures that the game is adaptable to various screen resolutions.

## Explanation of how graphics were integrated into the game:

The integration of graphics into the game involves various elements such as drawing small boxes and implementing text animations. To create small boxes, a single coordinate is used, which is then multiplied by multiples of 1, 2, 3, and so on, with a specified length. This calculation method simplifies the process of creating multiple boxes with incremental positions. Additionally, text animations are implemented which display the text character by character with sight delay.

## Usability and Effectiveness of the User Interface:

The user interface in the game enhances usability and effectiveness through various design choices. Firstly, each player's pawn is uniquely colored, allowing for easy identification among players. Additionally, each pawn is assigned a number in its center, further aiding in quick identification during gameplay. When it comes to moving a pawn, the program prompts the player to select which pawn to move based on the number displayed at its center.

To ensure a smooth visual experience, the game implements double buffering, effectively preventing screen flickering. This technique enhances the overall visual presentation and prevents distractions or interruptions during gameplay.

Furthermore, the user interface clearly communicates each player's turn. The name of the player currently playing is prominently displayed as "PlayerX's turn," providing a clear indication of whose turn it is. This helps players stay engaged and maintain a clear understanding of the game's progress. To organize information effectively, each player's relevant details are displayed within their designated part of the screen. This approach ensures that each player has their own section, where information specific to their gameplay, such as dice roll or other relevant details, is presented. This division of the screen supports better player engagement and allows for focused interaction within each player's respective area.

# Testing and Validation:

During the development of this game, numerous bugs were encountered and addressed through rigorous testing and validation. While some bugs were immediately apparent, others were discovered when testing the game with various edge cases. To identify and resolve these issues, cout statements were strategically placed throughout the code, allowing for the printing of variable values in the standard output window. This enabled a thorough examination of whether the variables were functioning as intended.

Through a systematic approach, each bug was addressed individually, ensuring that they were fixed one by one. This process, although time-consuming, proved to be highly effective in achieving a bug-free and efficient game. The majority of the encountered bugs were attributable to typographical errors or logical mistakes, which were rectified during the debugging process. By persevering with this methodical approach, all the identified bugs were successfully resolved, resulting in a stable and reliable game.

# Conclusion

In conclusion, this project aimed to develop a Ludo game using C++ and data structures. The objectives of the project were to create a functional and enjoyable gaming experience while effectively utilizing various data structures to manage the game mechanics.

Reflecting on the implementation, the use of data structures such as arrays, linked lists, and classes proved essential in organizing and storing game data, player information, and board configurations. These data structures facilitated efficient game logic and allowed for easy manipulation of game elements.

The game's performance and functionality were assessed through rigorous testing and validation. Overall, the game demonstrated smooth gameplay, accurate dice rolls, and proper movement of pawns. Bugs and issues encountered during the development process were identified and resolved, resulting in a stable and reliable game.

Looking towards future enhancements, there are several areas for improvement. For instance, the game could be made more complex by incorporating additional features such as allowing players a second turn when rolling a six or requiring players to kill at least one opponent's pawn before entering their home entries. These enhancements would add strategic depth and further engage players in the gameplay.

In conclusion, this Ludo game project successfully achieved its objectives by implementing data structures effectively, delivering a well-performing game, and laying the foundation for potential future enhancements. The development process provided valuable insights into game programming, data structure utilization, and the iterative nature of debugging and testing.

# CODE

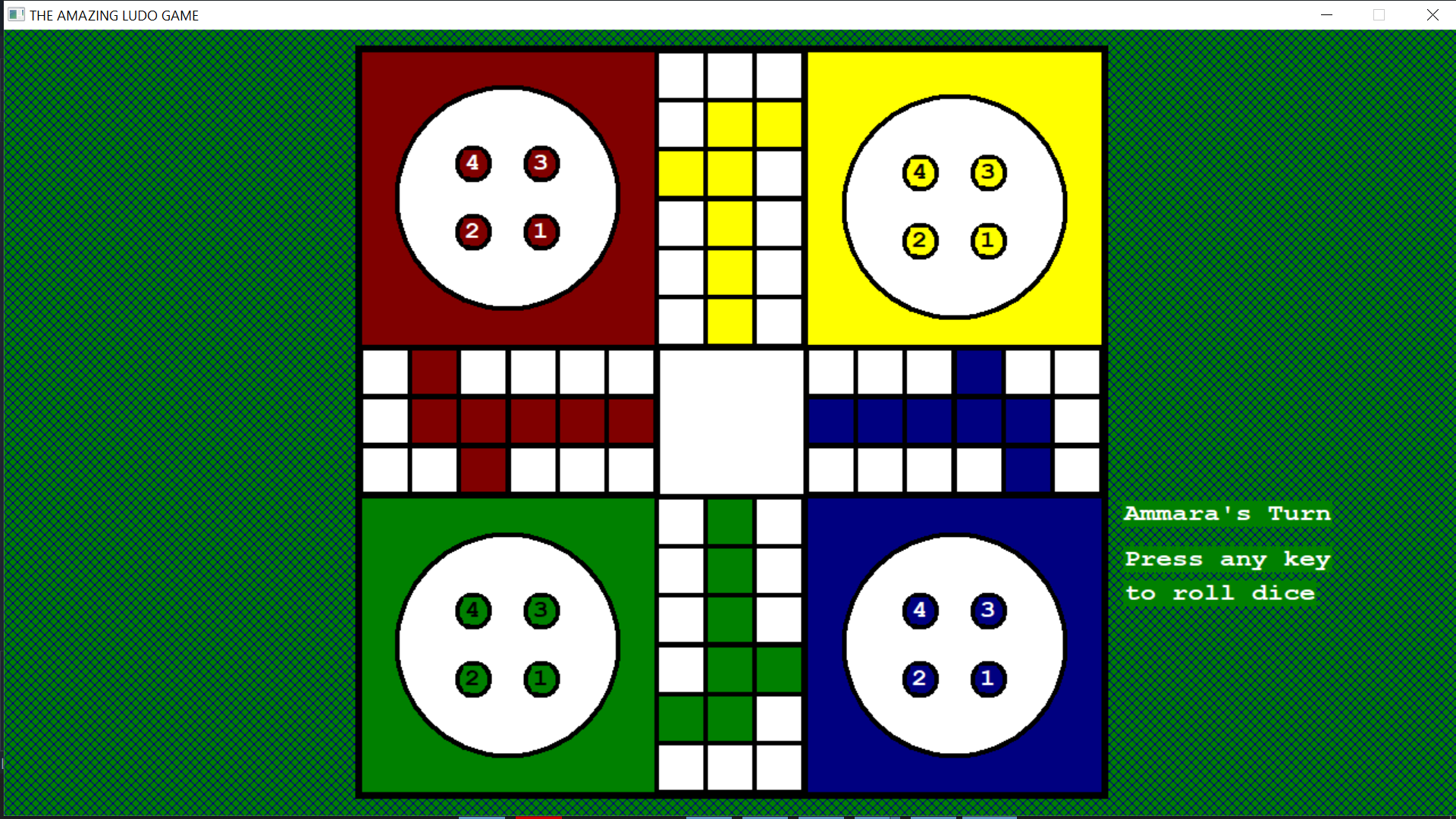
# OUTPUT SCREEN PICTURES

## The main screen

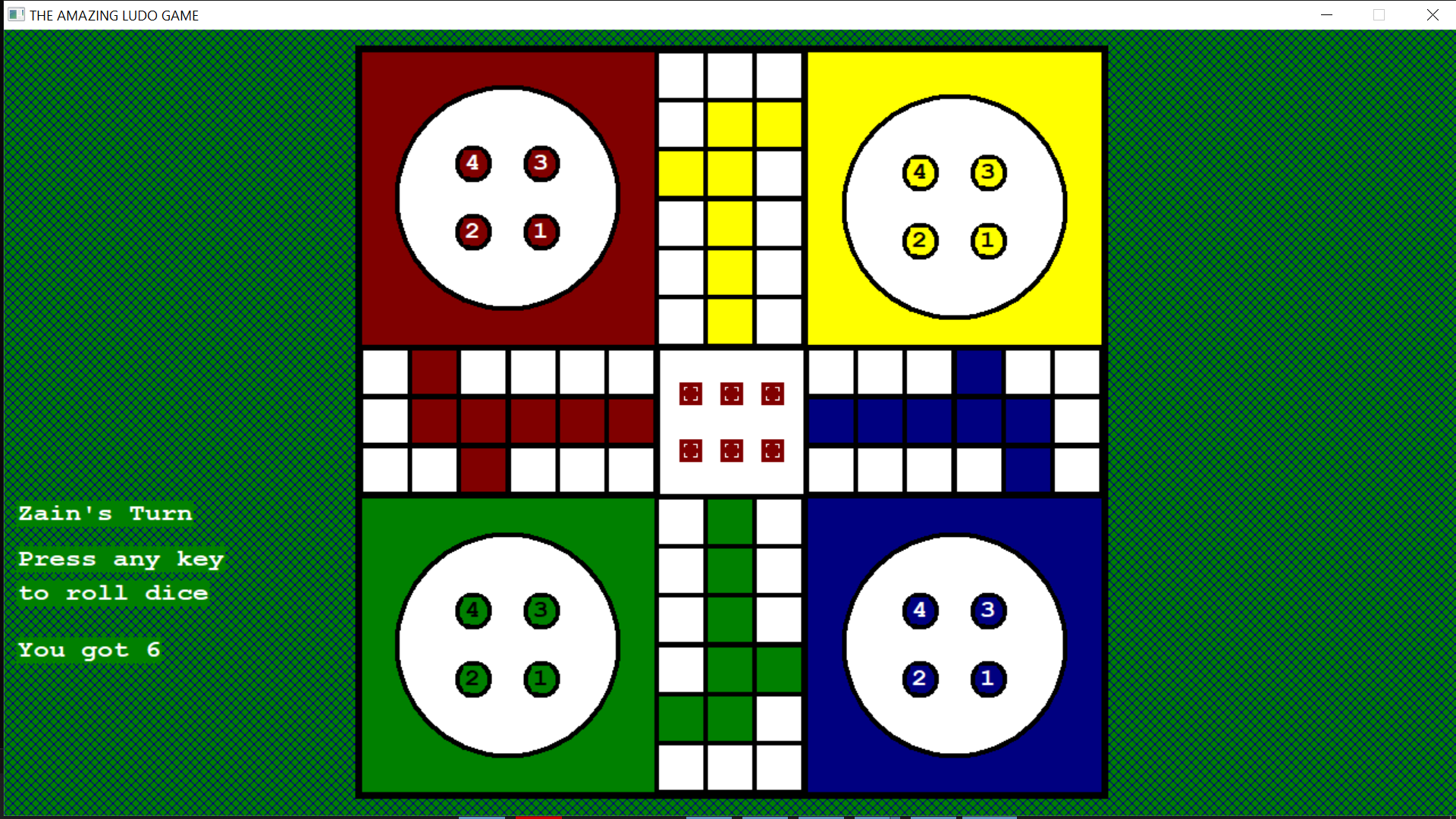


## Game Board

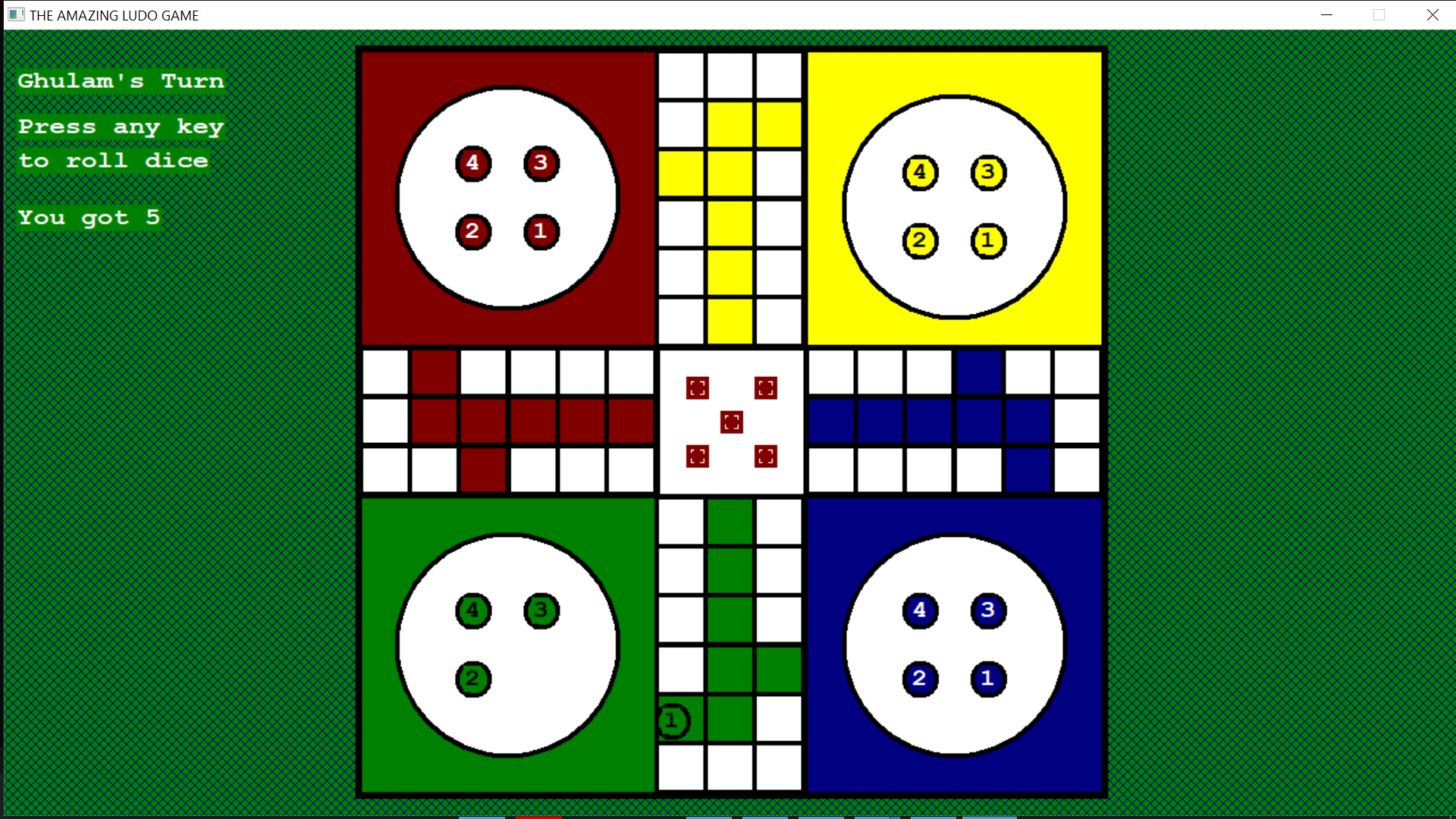
Random Player got first turn.



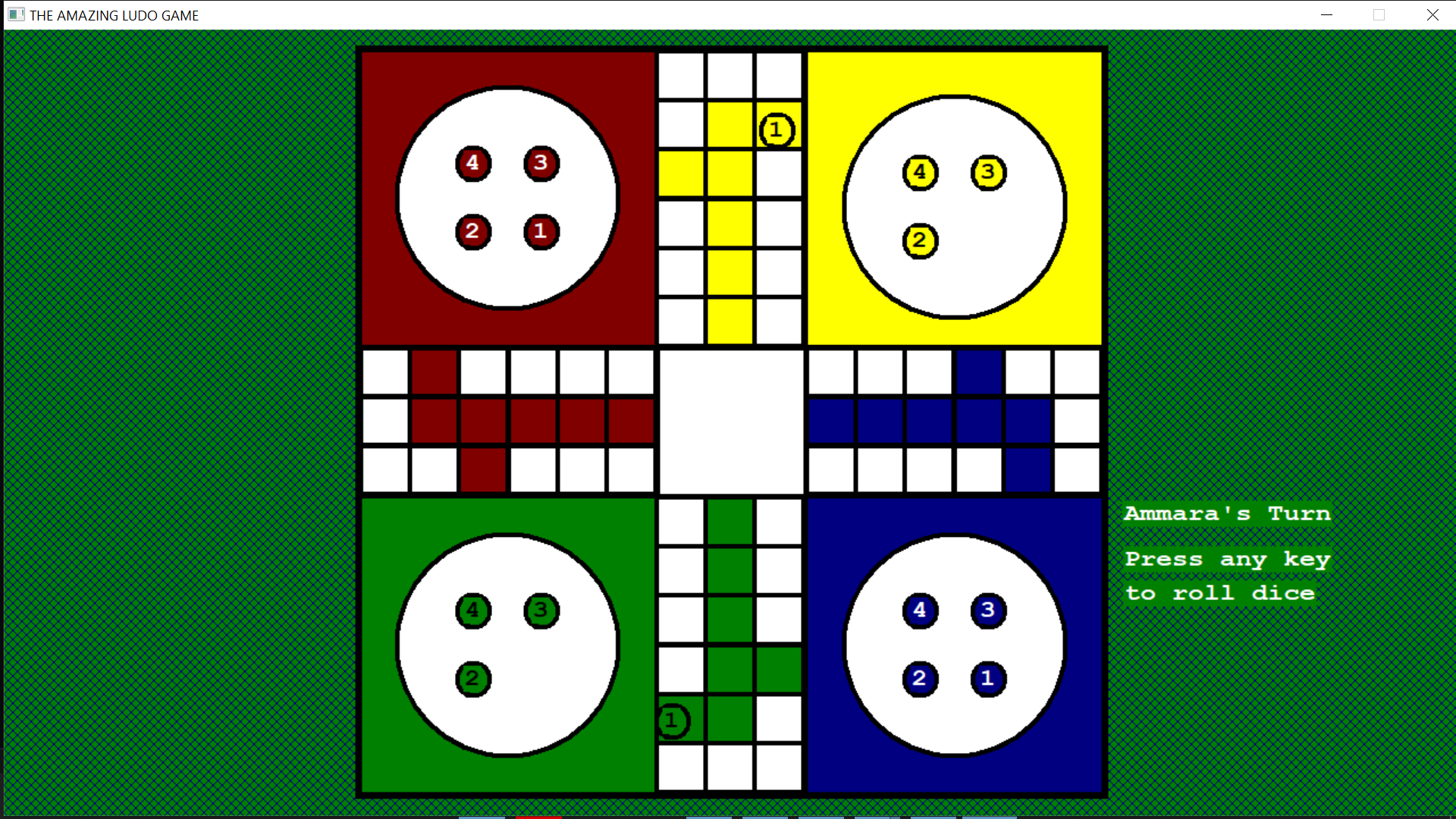
Displaying the value of dice.



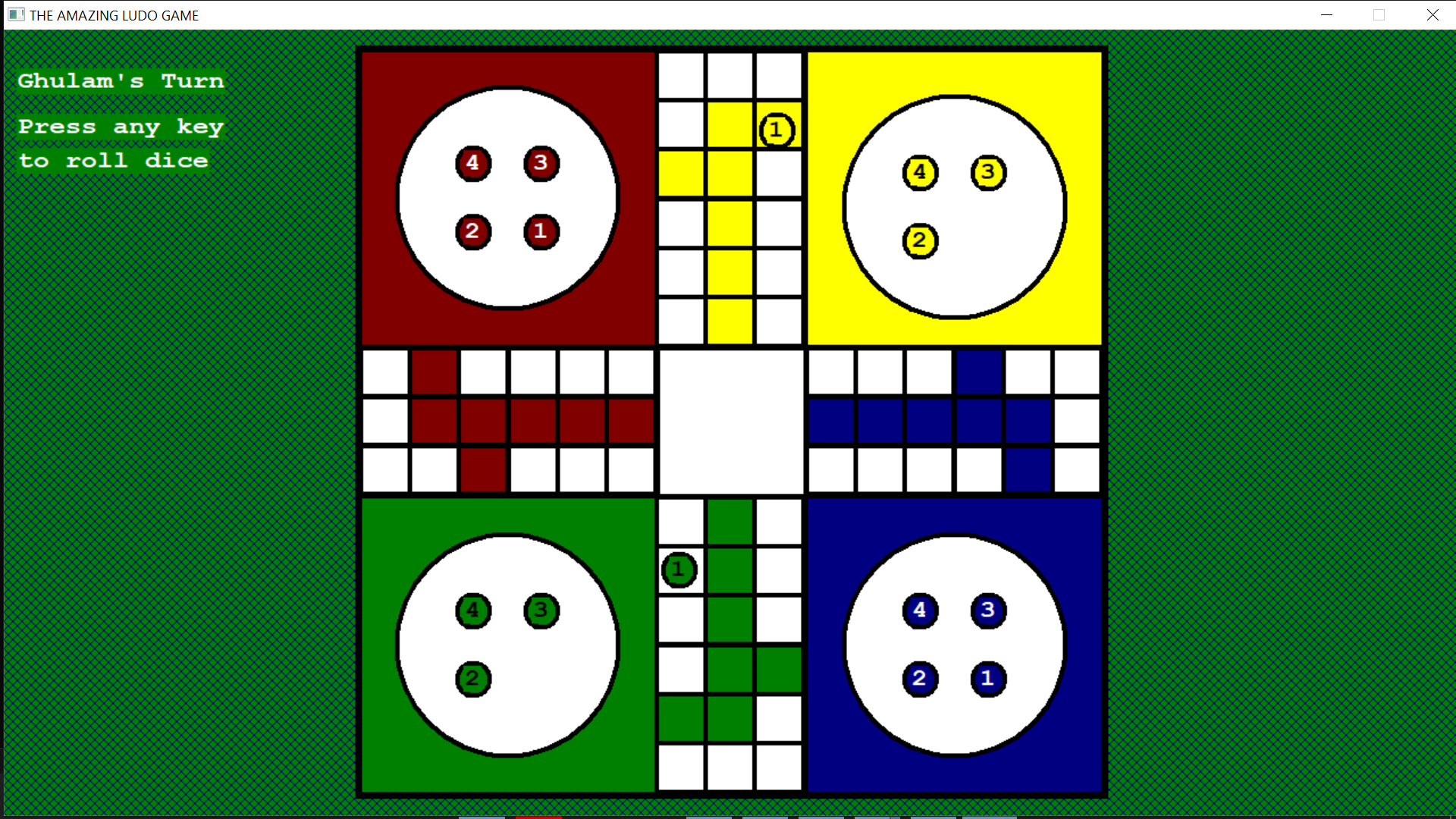
Next player turn and dice roll



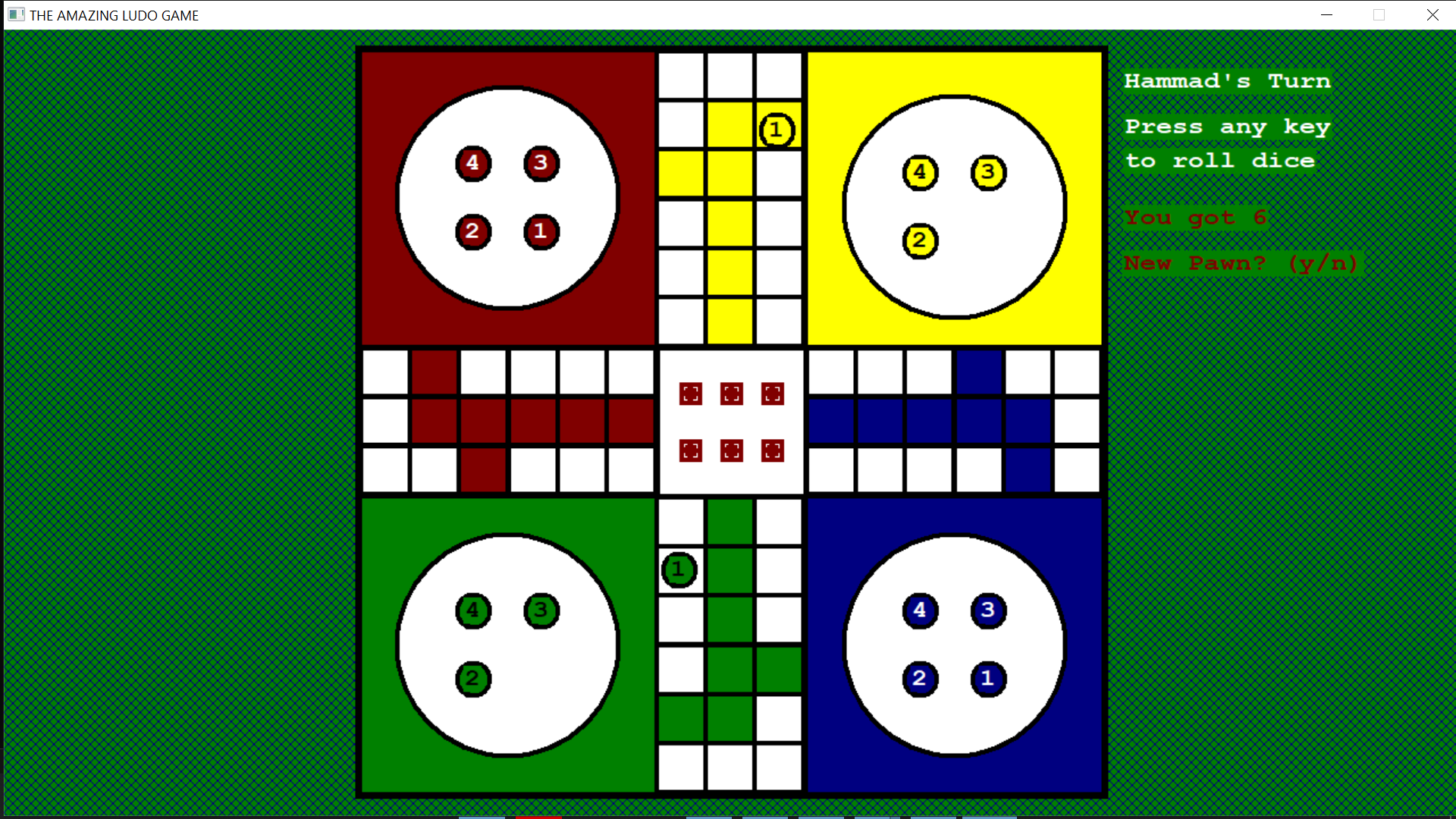
Pawn taken out and next player turn.



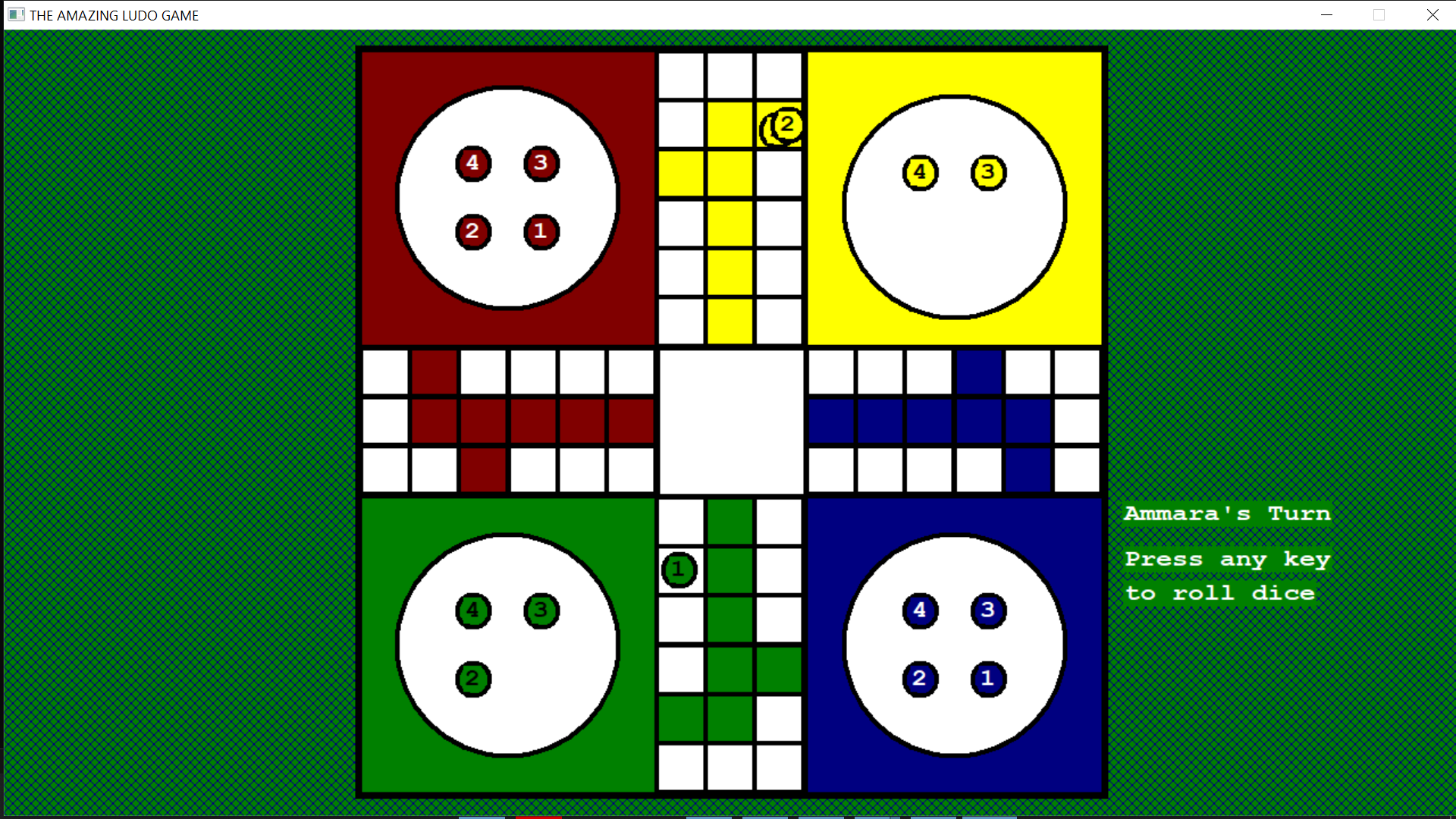
On rolling 3 after few turns, the pawn moved forward



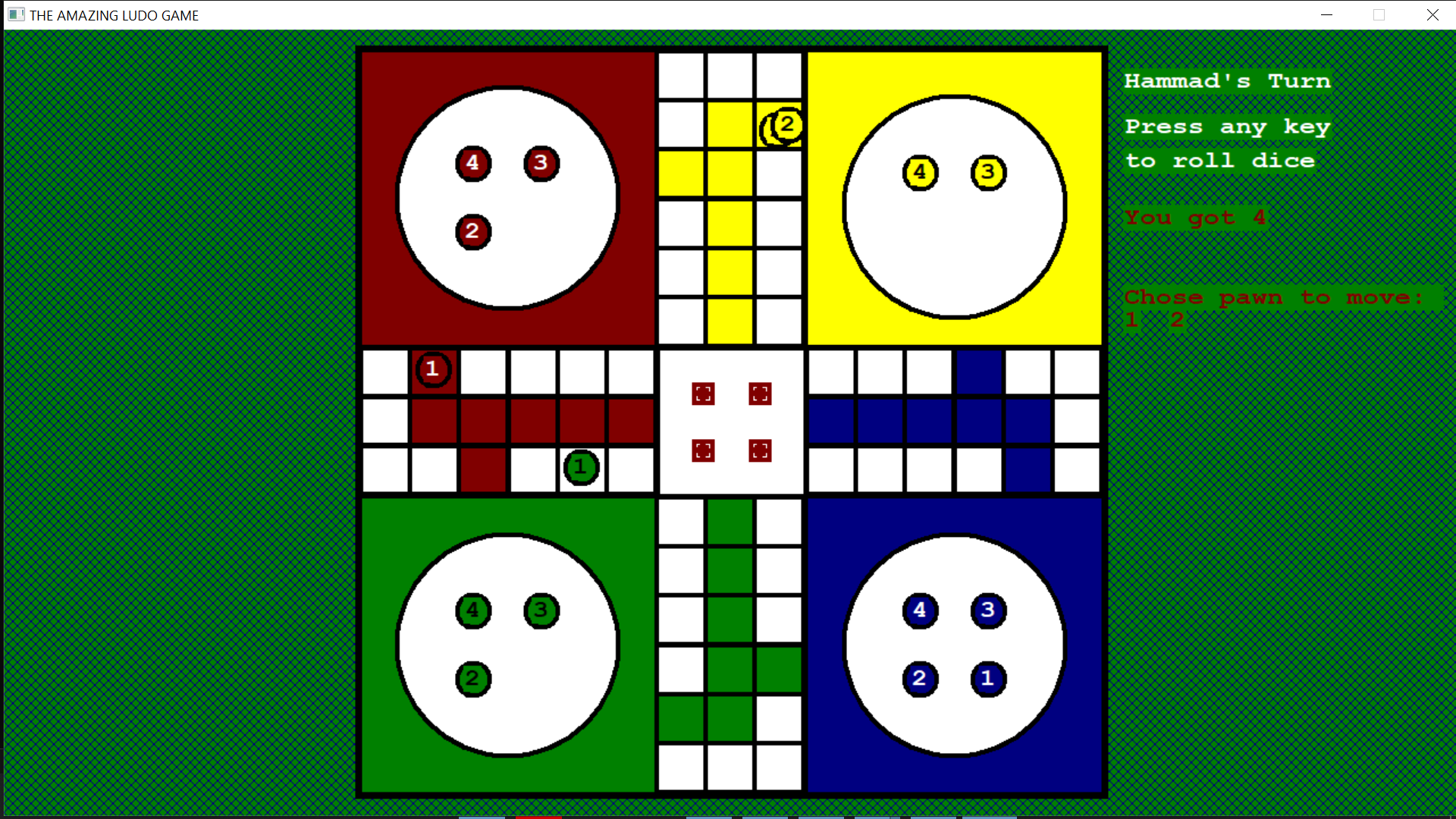
The player already had a pawn out it is asked whether to take out a new pawn



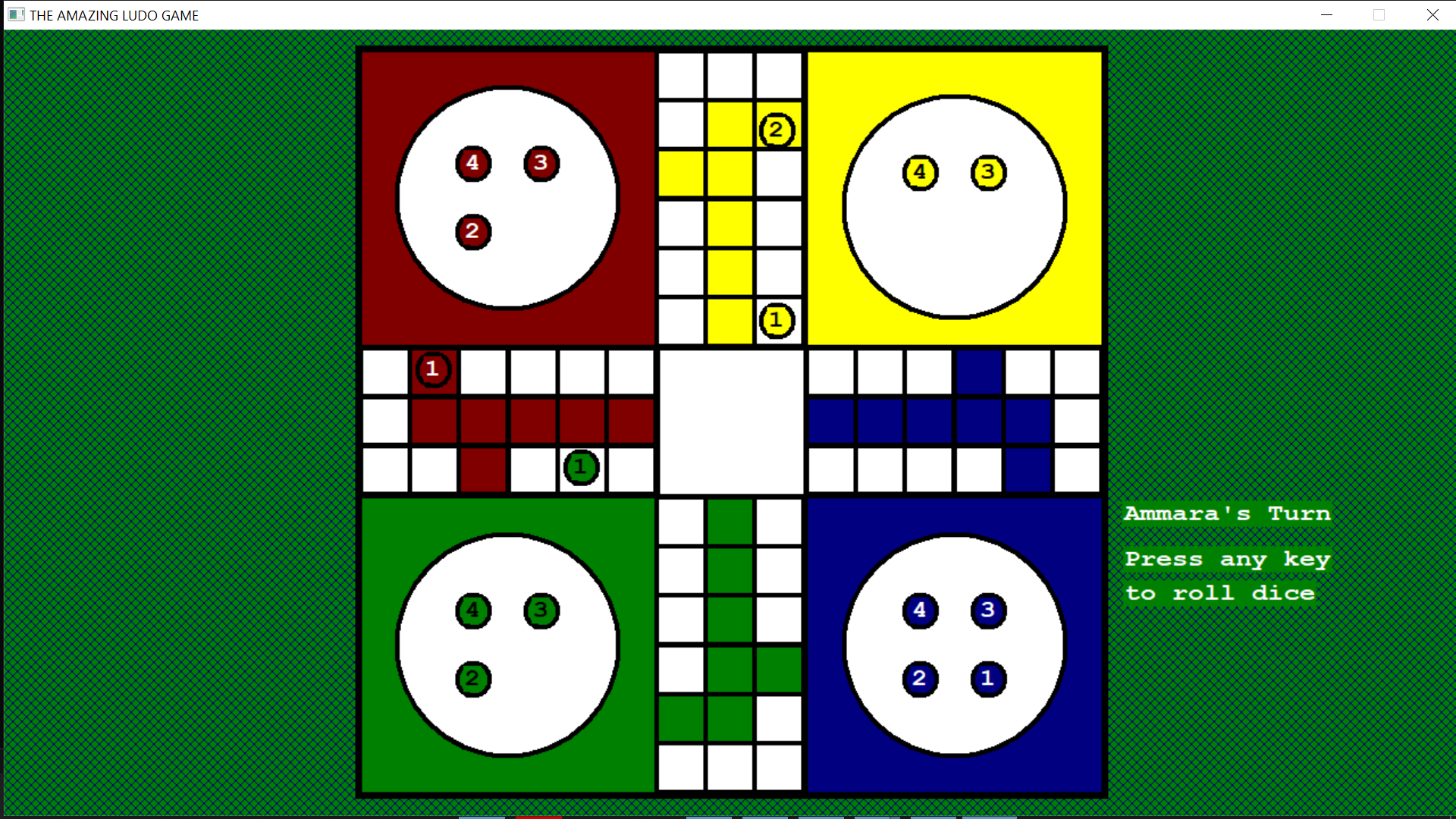
The new pawn appears on top of the previous pawn



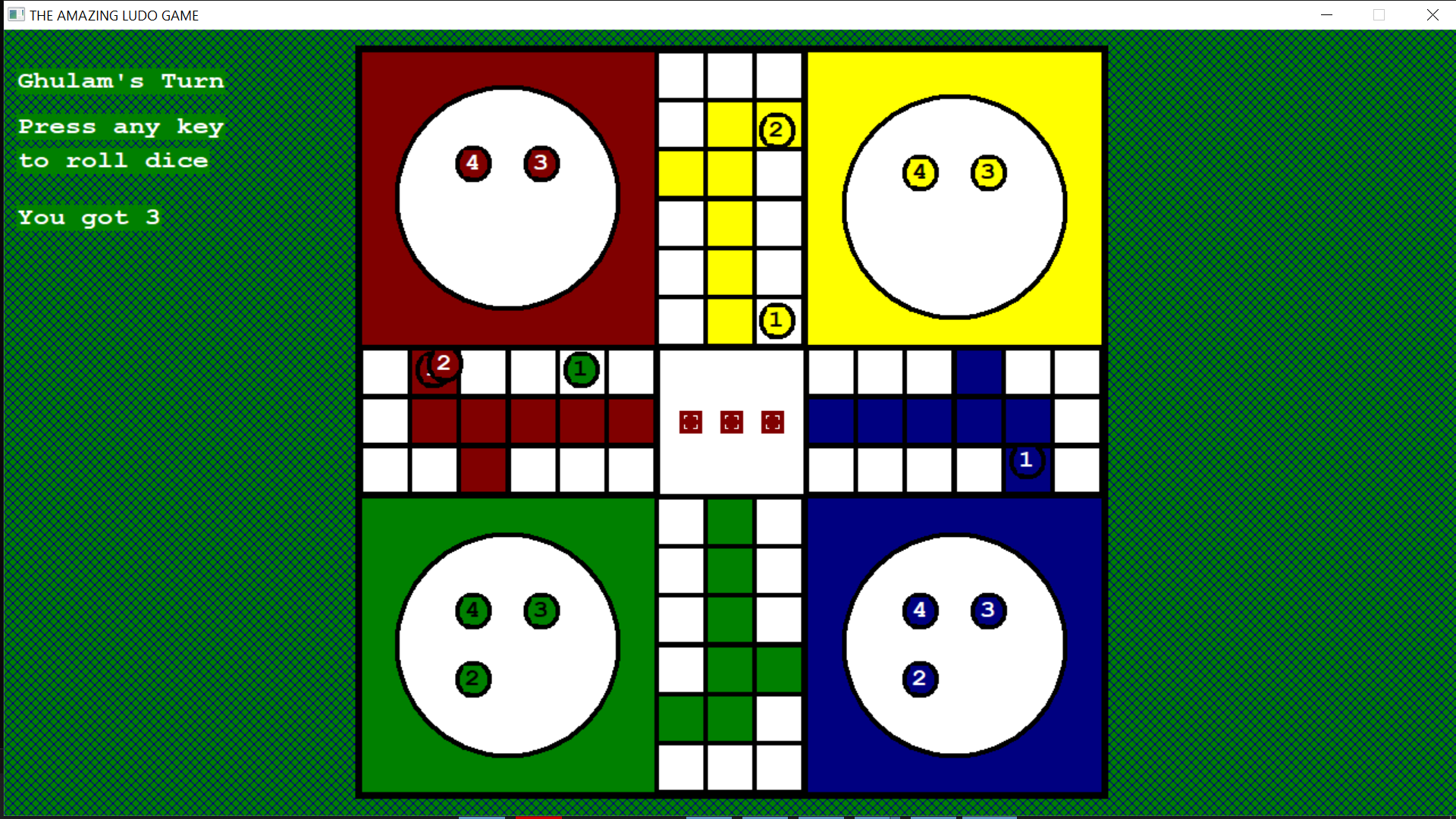
After few more turns, another player has a pawn out and the player with two pawns is asked which pawn to move. The pawns are identified by a number in center



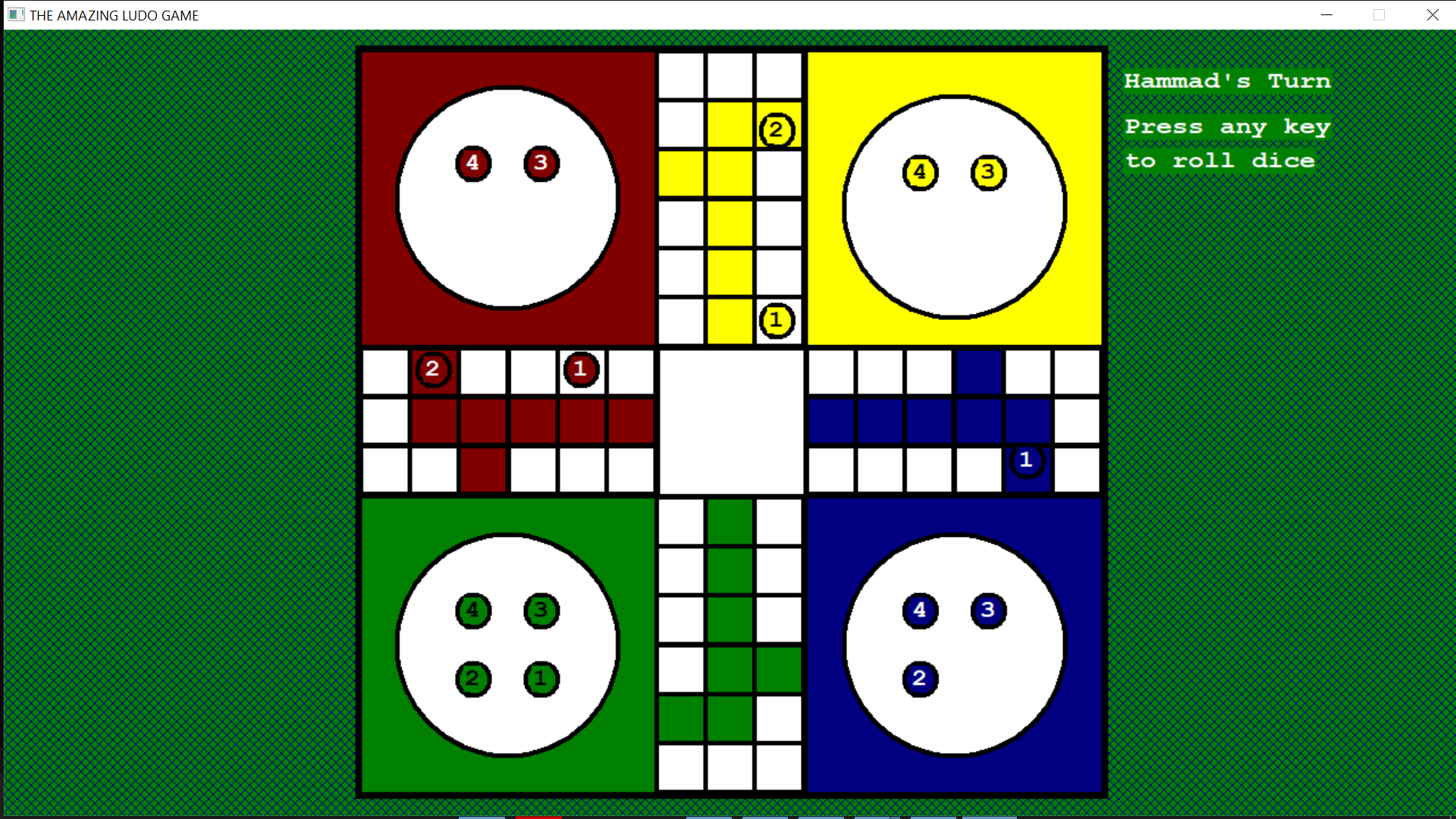
The player decided to move a pawn.



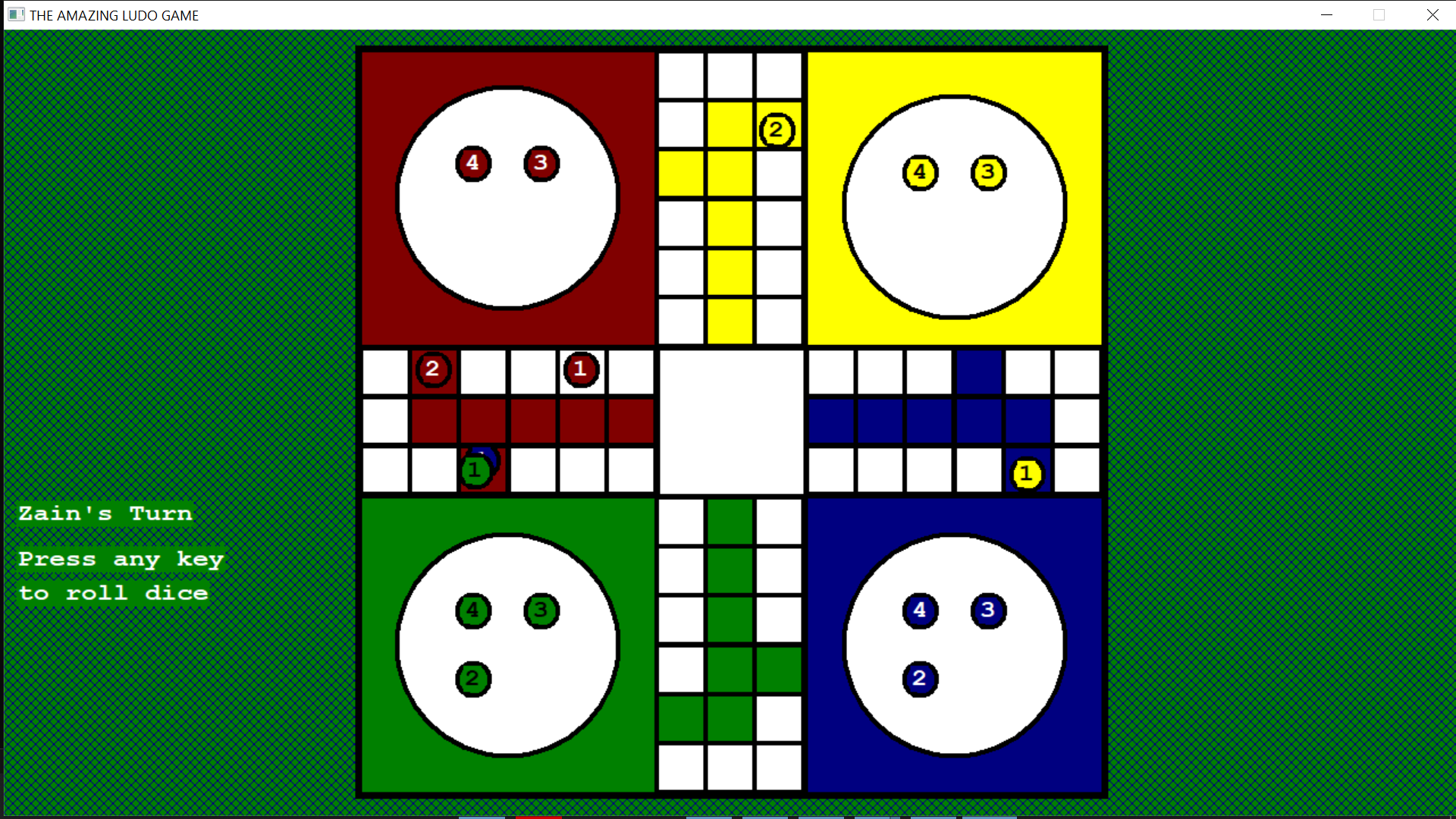
After few turns, few more pawns are out. Now the blue player has rolled three and is about to kill a pawn of green player.



Pawn killed successfully and next player’s turn. I really wished I had a message of Bravo appear on screen after killing a pawn.

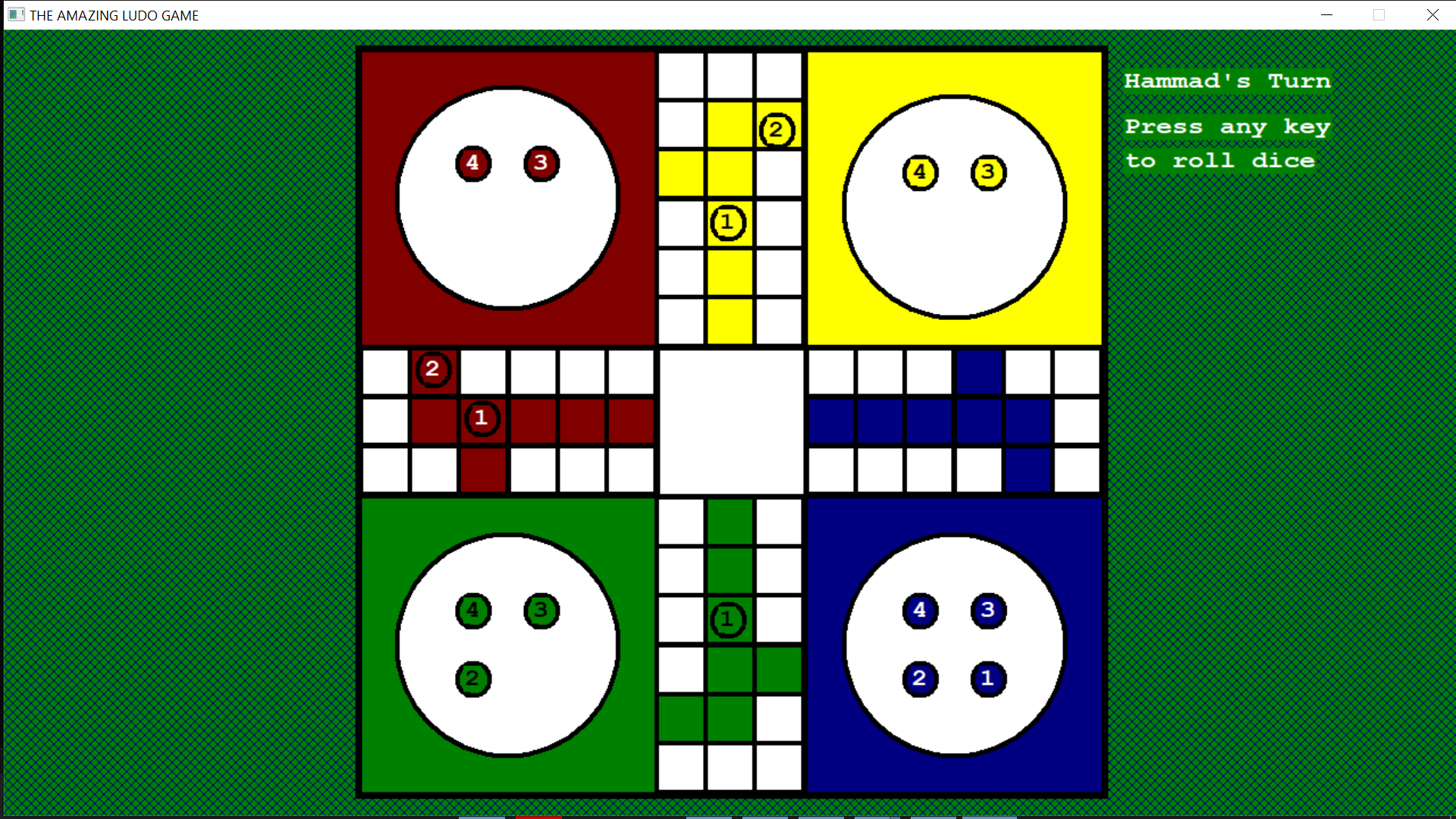


When pawns are on safe zones, they don’t kill or get killed. Here we have a blue pawn on top of red pawn with just a slight tilt so as to assure both pawns are there. Without this tilt, the red pawn would have been completely hidden under the red pawn.12

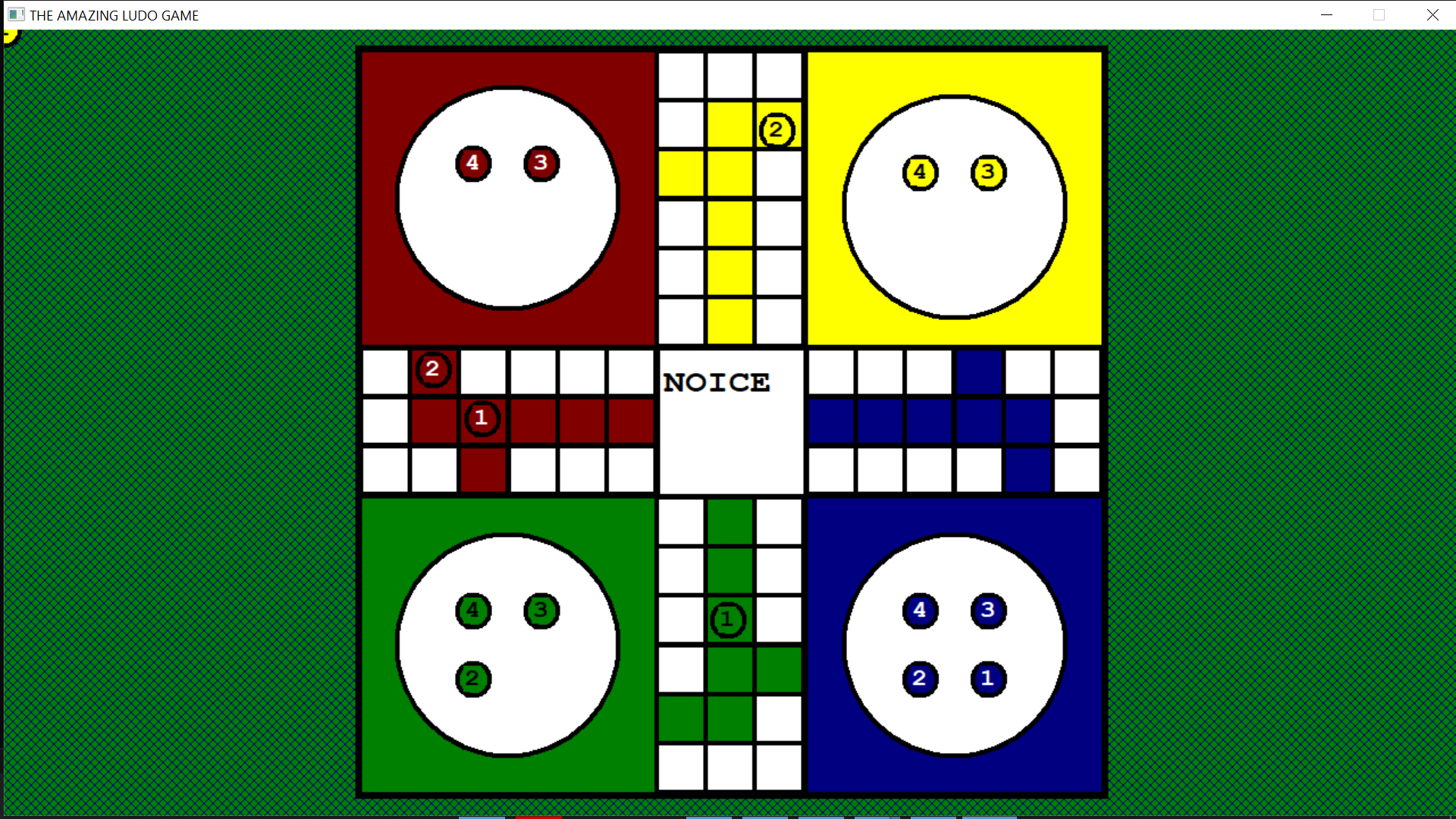


From here, the pics are of a different game

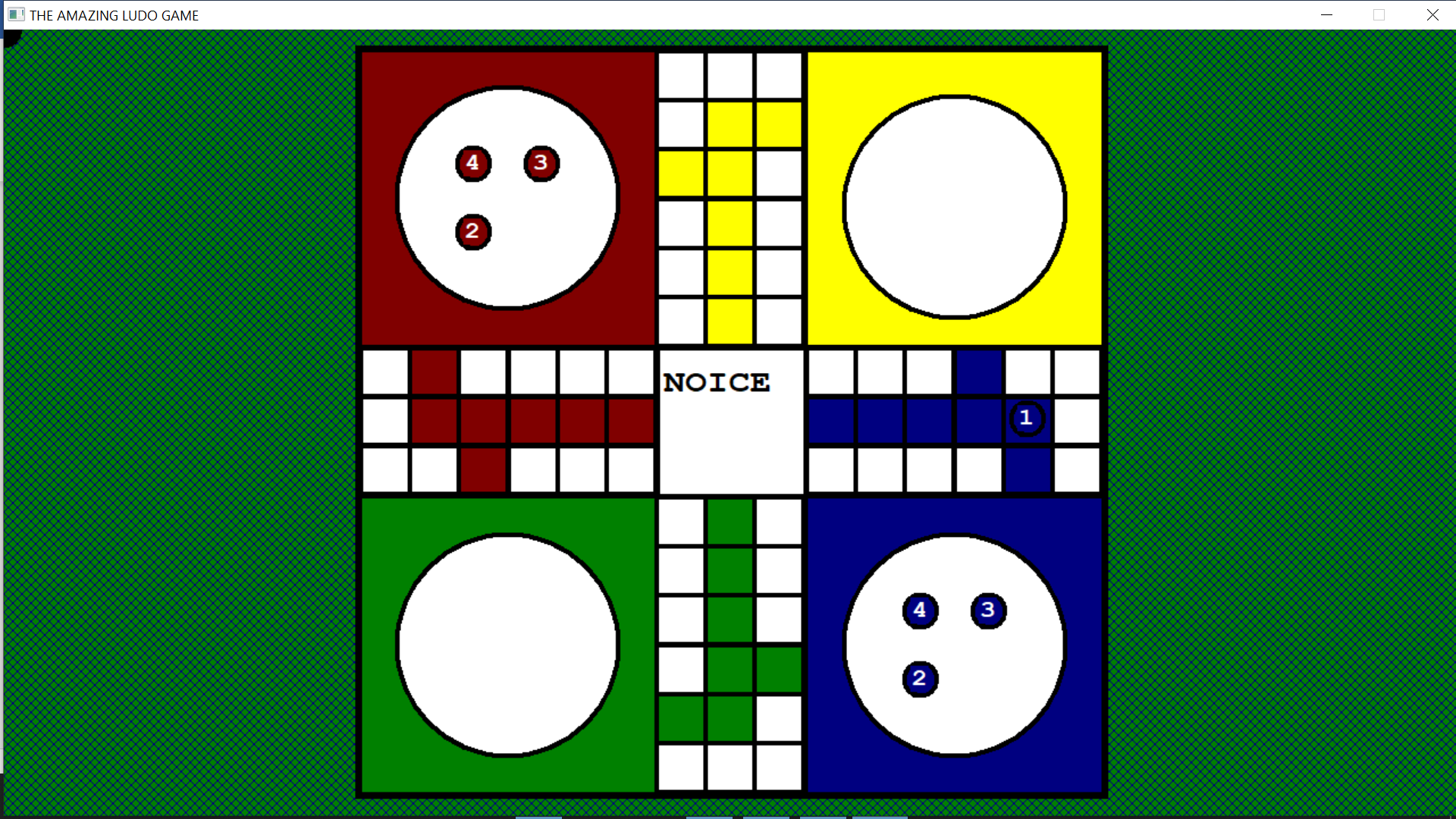
When the player completes a turn, it then enters its home entries. Here the pawns are about to enter their home entries.



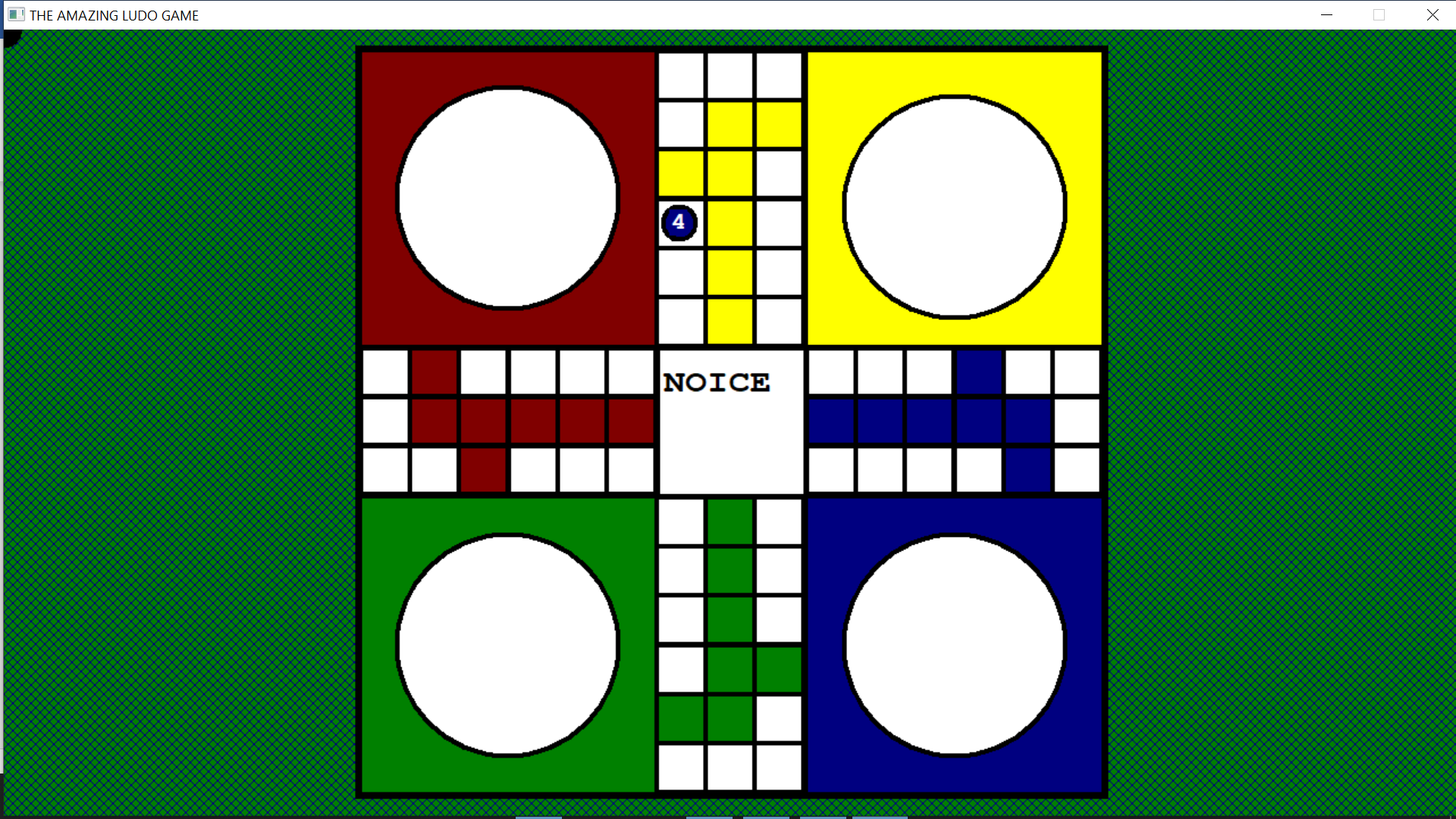
The Yellow pawn finally entered the home entry.



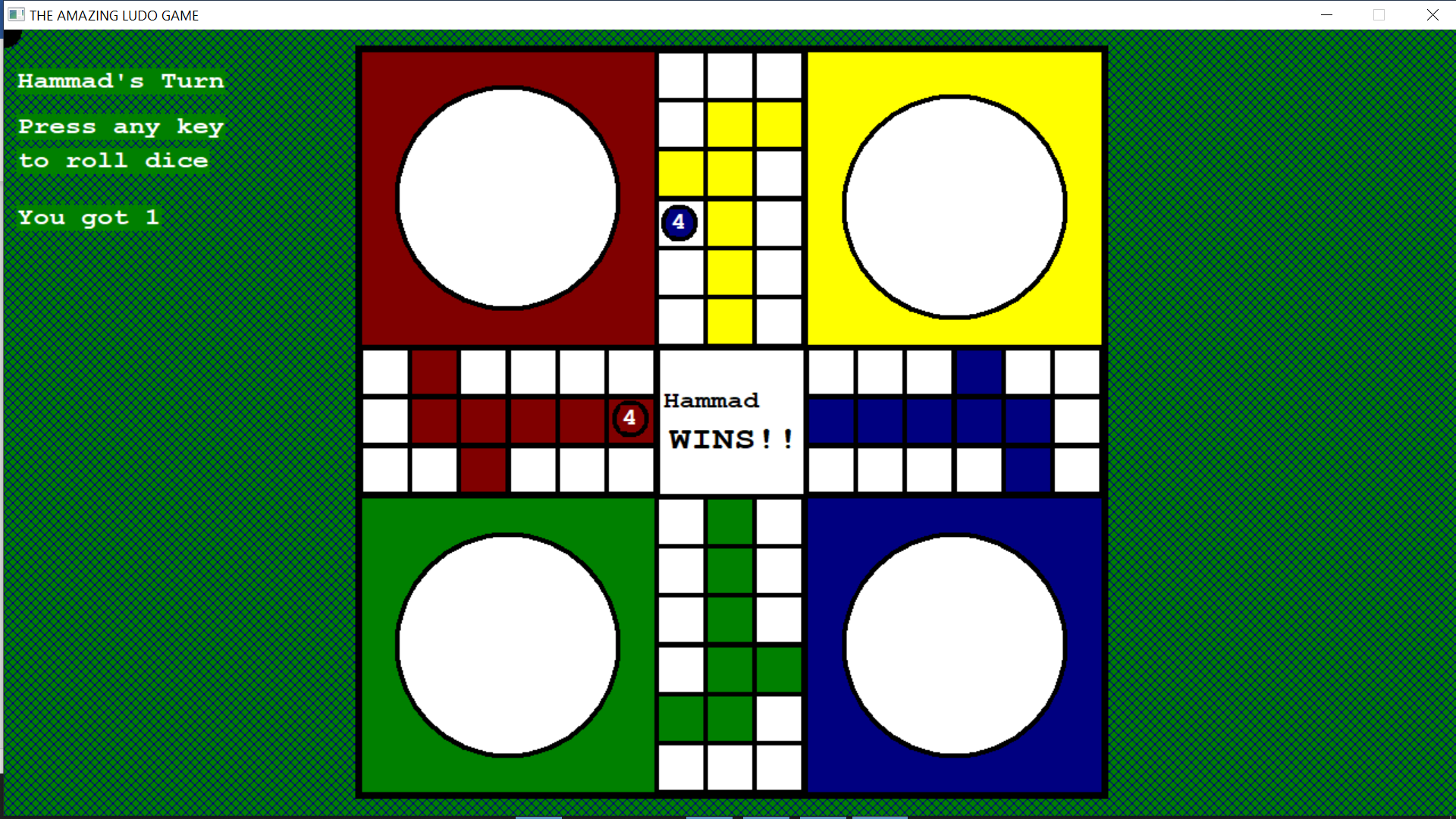
When the pawn finally enters the home, this message is displayed.



All the pawns of one player have entered home.



When a player wins, its name appears with a win message. Now this player will be skipped and not given any turn. When there is only one player left, the game ends.



Since the last game was a two players game, it ended when one won. The game over screen displays the message and player positions. After which the program ends.

## The ending screen



# Disclaimer and Copyright:

This program, including its code and logic, is the sole work of Hammad Mustafa. It was developed independently, without assistance from any individuals or online sources. The project's development and implementation were carried out solely by Hammad Mustafa. No other individuals were involved in the creation of this game. The code provided is not to be shared or distributed without explicit permission. However, you are allowed to edit and modify the code for personal and reasonable purposes. If you intend to share this code online, it is mandatory to include the original author's name and provide full credit to the author, Hammad Mustafa.

Copyright © 2023

All rights reserved.

This program, including its code and logic, is protected by copyright laws and is the intellectual property of Hammad Mustafa. The unauthorized use, distribution, or reproduction of this program or its code without permission is strictly prohibited.