



SCHOOL OF SCIENCE & ENGINEERING

FALL 2023

CSC 4301 Introduction to Artificial Intelligence

Term Project: 3

Bust The Ghost

November 23rd, 2023

Realized by:

Khadija Salih Alj

Noura Ogbi

Younes Bouziane

Supervised by:

Prof. Tajjedine Rachidi

Table of Contents

Introduction.....	3
I. Unity Environment	4
II. Bust The Ghost Game.....	5
III. Probabilistic Inferencing in Bust the Ghost.....	8
IV. Probability-Code Analysis.....	10
V. Video demo on YouTube	15
VI. Conclusion	16

Introduction

Artificial Intelligence (AI) is revolutionizing various industries and applications, including gaming. This report focuses on a project that combines AI techniques with the popular game development platform Unity to create an exciting game called “*Bust the Ghost*.” The project involves utilizing AI algorithms and probability distributions to enhance the gameplay experience.

Bust the Ghost is a game designed to challenge players’ perception and decision-making skills. The objective is to locate a hidden ghost within a 7x13 grid by analyzing sensor readings and using Bayesian inference 😊. Unity, a powerful and widely used game development engine, provides the platform for creating the game’s interactive environment. The tasks involved in this project include:

- Creating a 7x13 grid: The game is set on a grid, and Unity is used to build the grid interface where the gameplay unfolds. The grid serves as the playing field where the ghost is hidden.
- Building buttons: Two buttons are implemented in the game interface - “Bust” and “Peep.” The “Bust” button allows players to make a guess and attempt to locate the ghost, while the “Peep” button enables players to see the probabilities associated with each grid cell.
- Ghost placement and prior distribution: The ghost is initially placed in one of the grid cells based on a prior distribution. A uniform distribution is used as the starting point, ensuring an equal likelihood of the ghost being in any cell 😊.
- Sensor readings and color representation: When a player clicks on a cell, a sensor reading is generated, indicating the distance between the clicked cell and the ghost. The color displayed on the cell provides feedback to the player 👁️, with **red** indicating the presence of the ghost, **orange** for 1 or 2 cells away, **yellow** for 3 or 4 cells away, and **green** for 5 or more cells away.
- Credit system and gameplay: Each click on a cell, results in the player losing 1 point from their initial credit. The remaining credit is updated and displayed after each click. The game continues until the player’s credit runs out, resulting in a LOSS 😱. However, the player can choose to “bust” a cell if they believe the ghost is present, potentially leading to VICTORY 🎉. The number of allowed busts is limited, adding an element of strategy to the gameplay.
- Bayesian inference and posterior probability update: After each click, the posterior probability of the ghost’s location is updated using Bayesian inference. The probability is calculated based on the sensor reading and the previous posterior probability. A conditional probability distribution is used to determine the color to display on each cell.
- Normalization: The posterior probabilities are normalized to ensure they sum up to 1.

By combining the power of Unity for game development and AI techniques like Bayesian inference, the *Bust the Ghost* project offers an engaging and challenging gaming experience. The use of probability distributions and sensor readings adds a unique twist to traditional gameplay, requiring players to think strategically and make informed decisions.

I. Unity Environment

The Unity environment serves as an ideal platform for implementing our “*Bust the Ghost*” game project. Its robust features and user-friendly interface enabled us to efficiently create and manage the game elements, ensuring a seamless gaming experience for players.

- Creating the 7x13 Grid: Unity’s scene editor provides a visual interface, where we created and manipulated the game world. By utilizing Unity’s 2D tools and grid-based systems, we set up the 7x13 grid for our game.
- Building Buttons and User Interaction: Unity’s UI system enabled us to create interactive buttons with custom functionality. In our project, we created two buttons: “Bust” and “Peep.” These buttons are integrated into the game interface, allowing players to click them to perform specific actions. Unity’s event system allowed us to associate functions with button clicks, enabling smooth and responsive user interaction.
- Ghost Placement and Prior Distribution: Using C#, we defined the initial placement of the ghost in one of the cells based on a prior distribution. By utilizing randomization functions, we generated a uniform distribution to determine the initial position of the ghost. This ensures that the game starts with a randomized ghost location.
- Sensor Reading and Color Feedback: Unity’s event system facilitates capturing player input, such as clicking on a cell in the grid. When a cell is clicked, we calculate the distance between the clicked cell and the ghost’s location. Based on this distance, we can assign colors (**red**, **orange**, **yellow**, or **green**) to provide feedback to the player 😊.
- Credit System and Display: We implemented a credit system where the player loses one point from their initial credit with each click. By keeping track of the remaining credit, we can update and display it after each click. Unity’s UI system enabled us to create text elements and dynamically update them to reflect the current credit, providing real-time feedback to the player.
- Bayesian Inference and Probability Update: We followed Bayesian inference to update the posterior probabilities of the ghost’s locations. By utilizing conditional probability distributions and Bayesian formulas, we calculated the posterior probabilities based on the sensor reading (color) and the previous posterior probabilities.

By leveraging the capabilities of the Unity environment, we seamlessly integrated various components of our “*Bust the Ghost*” game, including grid creation, button functionality, user interaction, probability calculations, and visual feedback. Unity’s intuitive interface, extensive scripting capabilities, and visual tools made it an ideal environment for implementing our project effectively.

II. Bust The Ghost Game

The “*Bust the Ghost*” game is an intriguing and intellectually stimulating experience that challenges players to use their deductive skills and probability reasoning to locate a hidden ghost within a grid of cells. Indeed, the primary objective of “*Bust the Ghost*” is to identify the precise location of a ghost hidden within a 7x13 grid. Players must strategically click on cells to gather sensor readings and use the color feedback provided to infer the ghost’s location. The game calls for careful evaluation logical reasoning and decision making to maximize the probabilities of success.

We introduced the interface below as the initial element displayed to the player 😊:



Figure 1: Game Initial Interface

Gameplay Mechanics

- The game takes place on a 7x13 grid, consisting of a total of 91 cells. At the beginning of the game, the ghost is randomly placed inside the grid based on a uniform distribution.
- When a player clicks on a cell, the game calculates the Manhattan distance between that cell and the ghost’s hidden location. The distance determines the color feedback displayed on the clicked cell, assisting the player in deducing the ghost’s proximity. The colors and their corresponding meanings are as follows:

Red: The ghost is in the clicked cell.

Orange: The ghost is 1 or 2 cells away from the clicked cell.

Yellow: The ghost is 3 or 4 cells away from the clicked cell.

Green: The ghost is 5 or more cells away from the clicked cell.

- Players are initially provided with a specific number of credits, representing the number of clicks they can make before depleting their resources. Each click deducts one point from the remaining score (initially 50). It is crucial to manage the credits strategically to avoid running out before locating the ghost. If the players exhaust their credits without successfully busting the ghost, they lose the game 😞. (Remaining Score: 0)



Figure 2: Game Over/Score = 0

- As players make sensor readings and receive color feedback, the game employs Bayesian inference to update the posterior probabilities of the ghost's potential locations. The updated probabilities are calculated based on the sensor reading and the previous posterior probabilities, allowing players to refine their estimation of the ghost's position.

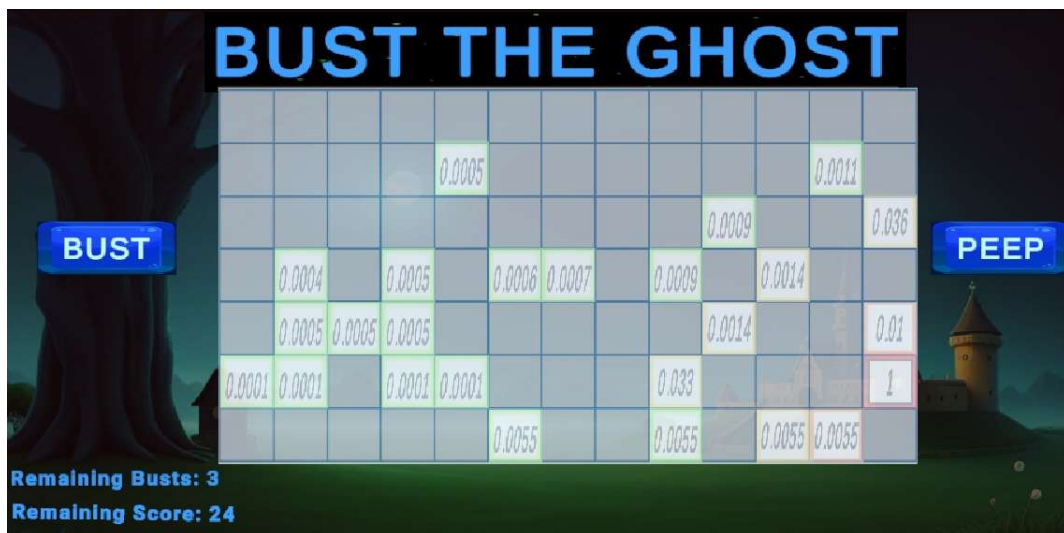


Figure 3: Probabilities Display

- The “Peep” button reveals the current probability distribution of the ghost’s locations on the grid. This visual representation of the probabilities assists players in identifying the most likely areas where the ghost might be hiding, contributing to their decision-making process.



Figure 4: Before “Peep”



Figure 5: After “Peep”

- Players can choose to “Bust” a cell if they believe the ghost is located there. If their assumption is correct, they win the game 🏆. However, if the ghost is not in the busted cell and they finish the allowed busts, they lose the game 😞.




Figure 6: Busted the Ghost



Figure 7: Busts = 0

III. Probabilistic Inferencing in Bust the Ghost

In this project, probabilistic inferencing is employed to express the ghost's location. The program utilizes a function called "*GhostPosition()*" to allocate the ghost's spot based on a prior uniform distribution of the ghost's location. Additionally, "*ColorPosition()*" is responsible for detecting the ghost's location and assigning the corresponding colors to the grid cells based on the distance .

To decide on the color to display after each click, probability reasoning and calculations rooted in Bayesian inference are essential. The relevant probability distribution is $P(\text{Color/Distance from Ghost})$, indicating the likelihood of observing a specific color given the distance from the ghost to update the posterior probability of the ghost's location following each click:

$$P(\text{Ghost}_i) = P(\text{Ghost} \mid \text{Color}_i) = P(\text{Ghost}_{\{i-1\}}) * P(\text{Color}_i \mid \text{Distance from Ghost})$$

Breaking down this formula and the associated calculations:

- $P(\text{Ghost}_i)$: the updated posterior probability of the ghost's location after the i^{th} click, which is the probability targeted for adjustment.
- $P(\text{Ghost}_{\{i-1\}})$: the prior probability of the ghost's location prior to the i^{th} click.
- $P(\text{Color}_i \mid \text{Distance from Ghost})$: the conditional probability of observing a specific color given the distance from the ghost. These probabilities must be determined based on the color scheme.

Now, let us examine the probabilities associated with each color based on the distance from the ghost:

- **On the ghost:** **Red** $P(\text{Red} \mid \text{Distance from Ghost}) = 1$ if the distance from the ghost is 0.
- **1 or 2 cells away:** **Orange** $P(\text{Orange} \mid \text{Distance from Ghost}) = 1$.
- **3 or 4 cells away:** **Yellow** $P(\text{Yellow} \mid \text{Distance from Ghost}) = 1$.
- **5+ cells away:** **Green** $P(\text{Green} \mid \text{Distance from Ghost}) = 1$.

After determining these conditional probabilities, the posterior probability can be updated to:

$$P(\text{Ghost}_i) = P(\text{Ghost}_{\{i-1\}}) * P(\text{Color}_i \mid \text{Distance from Ghost})$$

To maintain the validity of the probabilities and ensure they collectively add up to 1, it is necessary to **normalize** them. Following the modification of the posterior probability, each revised value is divided by the sum of all updated values.

Upon clicking the "Peep" button, the probability distribution $P(\text{Ghost} \mid \text{Color})$ should be showcased on the squares. This entails displaying, for the clicked squares, the probability of the ghost being in that location given the observed color.

Indeed, the implementation of the Bust the Ghost game involves managing data structures, algorithms, and user interactions. Here's a more detailed breakdown:

- **Data Representation:**

The game grid is depicted using a two-dimensional array named “grid” of type Tile, which serves to represent a grid of tiles in the game. This array has dimensions of 13 columns and 7 rows.

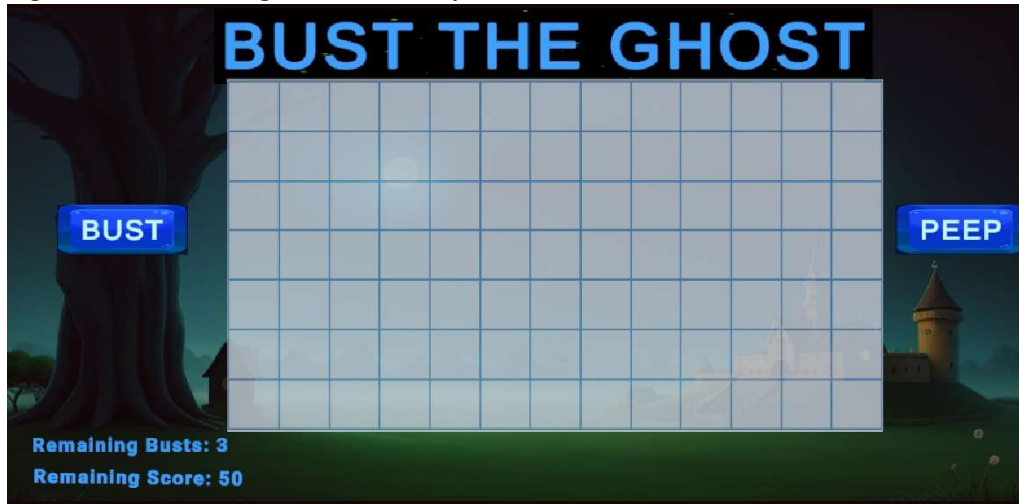


Figure 8: 7*13 game environment

- **Probability Calculations:**

To compute the conditional probabilities ($P(\text{Color/Distance from Ghost})$), tables are utilized to represent joint probabilities. This approach enables efficient retrieval of probabilities based on the distance from the ghost.

- **Click Handling:**

Every click triggers an update to the posterior probability through Bayesian inference. The prior probability is retrieved from the preceding click, then, the conditional probability is computed based on the observed color and distance to adjust the posterior probability accordingly 😊.

Upon clicking the “Peep” button, the probabilities can be displayed on the already-clicked squares.

Upon clicking the “Bust” button, the program evaluates whether the distance from the player’s selected cell to the ghost is precisely 0. If this condition is met, signaling that the player has accurately identified the ghost’s location, a victory is declared 🏆. However, if the distance is not zero, indicating an incorrect guess, one point is deducted from the total allowed busts. This mechanic introduces an element of strategy, prompting players to thoughtfully manage their limited attempts to uncover the ghost’s hiding spot.

IV. Probability-Code Analysis

A. Game.cs:

1. JointTableProbability:

In the “**Game.cs**” file, we used probabilistic inferencing to determine the chances of colored tiles being near the ghost. It then adjusts the grid by assigning appropriate probability values to each tile, considering both color and distance through conditional statements.

```
public double JointTableProbability(string color, int DistanceFromGhost){  
    //TABLE A  
    if(color.Equals("red") && DistanceFromGhost==0){  
        return 0.7;  
    }  
    if(color.Equals("orange") && DistanceFromGhost==0){  
        return 0.2;  
    }  
    if(color.Equals("yellow") && DistanceFromGhost==0){  
        return 0.05;  
    }  
    if(color.Equals("green") && DistanceFromGhost==0){  
        return 0.05;  
    }  
  
    //TABLE B  
    if(color.Equals("red") && (DistanceFromGhost==1 || DistanceFromGhost==2)){  
        return 0.3;  
    }  
    if(color.Equals("orange") && (DistanceFromGhost==1 || DistanceFromGhost==2)){  
        return 0.5;  
    }  
    if(color.Equals("yellow") && (DistanceFromGhost==1 || DistanceFromGhost==2)){  
        return 0.15;  
    }  
    if(color.Equals("green") && (DistanceFromGhost==1 || DistanceFromGhost==2)){  
        return 0.05;  
    }  
  
    //TABLE C  
    if(color.Equals("red") && (DistanceFromGhost==3 || DistanceFromGhost==4)){  
        return 0.05;  
    }  
    if(color.Equals("orange") && (DistanceFromGhost==3 || DistanceFromGhost==4)){  
        return 0.15;  
    }  
    if(color.Equals("yellow") && (DistanceFromGhost==3 || DistanceFromGhost==4)){  
        return 0.5;  
    }  
    if(color.Equals("green") && (DistanceFromGhost==3 || DistanceFromGhost==4)){  
        return 0.3;  
    }  
  
    //TABLE D  
    if(color.Equals("red") && DistanceFromGhost>5){  
        return 0.05;  
    }  
    if(color.Equals("orange") && DistanceFromGhost>5){  
        return 0.15;  
    }  
    if(color.Equals("yellow") && DistanceFromGhost>5){  
        return 0.3;  
    }  
    if(color.Equals("green") && DistanceFromGhost>5){  
        return 0.5;  
    }  
    return 0;  
}
```

The code above defines a method named **JointTableProbability** that takes two parameters, a string **color** and an integer **DistanceFromGhost**. The purpose of this method is to assign the probability based on these input values.

Table A:

- If the ‘color’ is “red” and the ‘DistanceFromGhost’ is 0, the probability is set to 0.7.
- If the ‘color’ is “orange” and the ‘DistanceFromGhost’ is 0, the probability is 0.2.
- If the ‘color’ is “yellow” and the ‘DistanceFromGhost’ is 0, the probability is 0.05.
- If the ‘color’ is “green” and the ‘DistanceFromGhost’ is 0, the probability is 0.05.

Table B:

- If the ‘color’ is “red” and the ‘DistanceFromGhost’ is 1 or 2, the probability is 0.3.
- If the ‘color’ is “orange” and the ‘DistanceFromGhost’ is 1 or 2, the probability is 0.5.
- When the ‘color’ is “yellow” and the ‘DistanceFromGhost’ is 1 or 2, the probability is 0.15.
- When the ‘color’ is “green” and the ‘DistanceFromGhost’ is 1 or 2, the probability is 0.05.

Table C:

- If the 'color' is "red" and the 'DistanceFromGhost' is 3 or 4, the probability is 0.05.
- If the 'color' is "orange" and the 'DistanceFromGhost' is 3 or 4, the probability is 0.15.
- If the 'color' is "yellow" and the 'DistanceFromGhost' is 3 or 4, the probability is 0.5.
- If the 'color' is "green" and the 'DistanceFromGhost' is 3 or 4, the probability is 0.3.

Table D:

- If the 'color' is "red" and the 'DistanceFromGhost' is 5 or greater, the probability is 0.05.
- If the 'color' is "orange" and the 'DistanceFromGhost' is 5 or greater, the probability is 0.15.
- If the 'color' is "yellow" and the 'DistanceFromGhost' is 5 or greater, the probability is 0.3.
- If the 'color' is "green" and the 'DistanceFromGhost' is 5 or greater, the probability is 0.5.

If none of these specific conditions are met, the function returns a default probability of 0.

2. CheckInputGrid:

We also created **CheckInputGrid** function to examine the player's input grid. It calculates the probability for each grid tile, taking into account the color and distance from the ghost, and then assigns the calculated probability to the respective tile.

```
public void CheckInputGrid(){
    int Distance=0;
    if(Input.GetButtonDown("Fire1")){
        Vector3 mousePosition = Camera.main.ScreenToWorldPoint(Input.mousePosition);
        int a = Mathf.RoundToInt(mousePosition.x);
        int b = Mathf.RoundToInt(mousePosition.y);
        if(a > width || b > height){
            return;
        }
        mousePosition = Camera.main.ScreenToWorldPoint(Input.mousePosition);
        int x = Mathf.RoundToInt(mousePosition.x);
        int y = Mathf.RoundToInt(mousePosition.y);
        lastClickedX=x; lastClickedY=y;

        Distance = CalculateDistance(x, y, GhostX, GhostY);

        Tile tile = grid[x, y];
        tile.SetIsCovered(false);

        if(JointTableProbability("green", Distance)>=0.5){
            grid[x,y].probability.text=Math.Round(((JointTableProbability("green", Distance)*0.05)/91,4).ToString());
            int G = Greens;
            redDistance=Distance;
            for (int yy = 0 ; yy < height; yy++){
                for (int xx = 0; xx < width; xx++){
                    if(xx != lastClickedX && yy != lastClickedY){
                        Distance = CalculateDistance(xx, yy, GhostX, GhostY);

                        if(JointTableProbability("green", Distance) >= 0.5){
                            grid[xx,yy].probability.text=Math.Round((JointTableProbability("green", Distance)*0.05)/Greens,5).ToString();
                        }
                        if(JointTableProbability("yellow", Distance)>=0.5){
                            grid[xx,yy].probability.text=Math.Round((JointTableProbability("yellow", Distance)*0.15)/(91-Yellows),4).ToString();
                        }
                        if(JointTableProbability("orange", Distance)>=0.5){
                            grid[xx,yy].probability.text=Math.Round((JointTableProbability("orange", Distance)*0.3)/(91-Oranges),4).ToString();
                        }
                        if(JointTableProbability("red", Distance)>=0.5){
                            grid[xx,yy].probability.text=Math.Round((JointTableProbability("red", Distance)*0.5),4).ToString();
                        }
                    }
                }
            }
        }
    }
}
```

The above code determines probabilities for grid elements depending on their color (“green” at first) and proximity to a ghost. When the probability for “green” surpasses 0.5, it adjusts the probability for the current grid element. The code subsequently traverses the entire grid, taking into account additional colors (“yellow,” “orange,” “red”), and adjusts the probabilities. If the color is green and the probability is greater than or equal to 0.5, the probability for the tile is calculated as $(JointTableProbability(\text{“green”}, Distance) * 0.05) / 91$.

If the color is “yellow” and the probability is greater than or equal to 0.5, the probability for the tile is calculated as $(JointTableProbability(\text{“yellow”}, Distance) * 0.15) / \text{Yellows}$.

```
else if (JointTableProbability("yellow", Distance)>=0.5){
    grid[x,y].probability.text=Math.Round(((JointTableProbability("yellow", Distance)*0.15)/Yellows, 4).ToString();
    redDistance=Distance;
    for(int yy =0 ; yy< height; yy++){
        for(int xx = 0; xx < width; xx++){
            if(xx!= lastClickedX && yy!= lastClickedY){
                Distance = CalculateDistance(xx, yy, GhostX, GhostY);

                if(JointTableProbability("yellow", Distance)>=0.5){
                    grid[xx,yy].probability.text=Math.Round((JointTableProbability("yellow", Distance)*Distance/(Distance*Greens)), 4).ToString();
                }
                if(JointTableProbability("green", Distance)>=0.5){
                    grid[xx,yy].probability.text=Math.Round((JointTableProbability("green", Distance)*1/(Distance*Greens)), 4).ToString();
                }
                if(JointTableProbability("orange", Distance)>=0.5){
                    grid[xx,yy].probability.text=Math.Round((JointTableProbability("orange", Distance)*1/(Distance*Oranges)), 4).ToString();
                }
                if(JointTableProbability("red", Distance)>=0.5){
                    grid[xx,yy].probability.text=Math.Round((JointTableProbability("red", Distance)*0.6), 3).ToString();
                }
            }
        }
    }
}
```

If the color is “orange” and the probability is greater than or equal to 0.5, the probability for the tile is calculated as $(JointTableProbability(\text{“orange”}, Distance) * Distance) / (Distance * Greens)$.

```
else if (JointTableProbability("orange", Distance)>=0.5){
    grid[x,y].probability.text=Math.Round((JointTableProbability("orange", Distance)*Distance/(Distance*Greens)), 3).ToString();
    redDistance = Distance;
    if(counter<height){
        counter++;
    }
    for (int yy =0 ; yy< height; yy++){
        for (int xx = 0; xx < width; xx++){
            if(xx!= lastClickedX && yy!= lastClickedY){
                Distance = CalculateDistance(xx, yy, GhostX, GhostY);

                if(JointTableProbability("yellow", Distance)>=0.5){
                    grid[xx,yy].probability.text=Math.Round((JointTableProbability("yellow", Distance)*1/(Distance+Yellows)), 3).ToString();
                }
                if(JointTableProbability("green", Distance)>=0.5){
                    grid[xx,yy].probability.text=Math.Round((JointTableProbability("green", Distance)*1/(Distance*Greens))/Oranges, 4).ToString();
                }
                if(JointTableProbability("red", Distance)>=0.5){
                    if(GhostProba<1){
                        grid[xx,yy].probability.text=Math.Round(Greens/91+(counter-0.05)/height,3).ToString();
                        GhostProba=Greens/91+(counter-0.05)/height;
                    }
                }
                if(JointTableProbability("orange", Distance)>=0.5){
                    grid[xx,yy].probability.text=Math.Round((JointTableProbability("orange", Distance)*Distance/(Distance*Oranges*Oranges)),3).ToString();
                }
            }
        }
    }
}
```

If the color is “red” and the probability is greater than or equal to 0.5, the probability for the tile is calculated as $(\text{JointTableProbability}(\text{“red”}, \text{Distance}) + 0.3)$.

```

else{
    grid[x,y].probability.text=Math.Round((JointTableProbability("red", Distance)+0.3),3).ToString();
    for (int yy = 0; yy < height; yy++){
        for (int xx = 0; xx < width; xx++){
            if(xx!= lastClickedX && yy!= lastClickedY){
                Distance = CalculateDistance(xx, yy, GhostX, GhostY);

                if(JointTableProbability("yellow", Distance)>=0.5){
                    grid[xx,yy].probability.text=Math.Round(((JointTableProbability("yellow", Distance)*1/(Distance))/91),4).ToString();
                }
                if(JointTableProbability("orange", Distance)>=0.5){
                    grid[xx,yy].probability.text=Math.Round(((JointTableProbability("orange", Distance)*1/(Distance))/91),4).ToString();
                }
                if(JointTableProbability("green", Distance)>=0.5){
                    grid[xx,yy].probability.text=Math.Round(((JointTableProbability("green", Distance)*1/(Distance))/91),4).ToString();
                }
            }
        }
    }
}
}
}

```

3. GhostPosition:

This part randomly picks a spot (x, y) on the grid. It checks if that spot is empty. If it's empty, it brings in a “red” ghost tile to that spot, updates the grid, marks down the position of the ghost, prints the ghost's coordinates, and then decides to spread some noisy prints and assigns colors around by calling *PlaceNoisyPrint()* and *PlaceColor(x, y)*.

If the spot is taken, it keeps looking for an empty spot by calling itself again.

```

public void GhostPosition(){
    int x = UnityEngine.Random.Range(0, width);
    int y = UnityEngine.Random.Range(0, height);
    if( grid[x, y] == null){
        Tile ghostTile = Instantiate(Resources.Load("Prefabs/red", typeof(Tile)), new Vector3(x, y, 0), Quaternion.identity) as Tile;
        grid[x, y]=ghostTile;
        GhostX=x; GhostY=y;
        Debug.Log(" "+GhostX+" "+ GhostY+ " ");
        NoisyPrintPosition();
        ColorPosition(x, y);
    }
    else{
        GhostPosition();
    }
}

```

4. NoisyPrintPosition:

Similar to the ghost placement, it randomly picks a spot (x, y). Checks if that spot is free. If it is, it drops a “red” noisy print there and updates the grid. If not, it looks for an empty spot by calling itself.

```

public void NoisyPrintPosition(){
    int x = UnityEngine.Random.Range(0, width);
    int y = UnityEngine.Random.Range(0, height);
    if( grid[x, y]==null){
        Tile noisyPrint = Instantiate(Resources.Load("Prefabs/red", typeof(Tile)), new Vector3(x, y, 0), Quaternion.identity) as Tile;
        grid[x, y]=noisyPrint;
    }
    else{
        NoisyPrintPosition();
    }
}

```


5. ColorPosition:

This ColorPosition function calculates the distances from a starting point (X, Y) to all other spots on the grid. Takes a stroll through the entire grid, checking the probabilities for “green,” “yellow,” and “orange” based on those distances.

- If the chance for “green” is high enough, and “yellow” and “orange” are not as lucky, and the spot is vacant, it brings in a “green” tile, updates the grid, and increases the count of **Greens**.
- If “yellow” has a probability of 0.5 or more and the spot is empty, it invites a “yellow” tile to the grid, updates it, and increments **Yellows**.
- Similarly, if “orange” has a probability above 0.5 and the spot is unoccupied, it welcomes an “orange” tile, updates the grid, and increases the count of **Oranges**.

```
public void ColorPosition(int X, int Y){
    int Distance = 0;
    for (int y = 0; y < height; y++){
        for (int x = 0; x < width; x++){
            Distance = CalculateDistance(x, y, X, Y);
            if(JointTableProbability("green", Distance)>=0.5 && JointTableProbability("yellow", Distance)<0.5 && JointTableProbability("orange", Distance)<0.5 && grid[x,y]==null){
                Tile color = Instantiate(Resources.Load("Prefabs/green", typeof(Tile)), new Vector3(x, y, 0), Quaternion.identity) as Tile;
                grid[x, y]=color;
                Greens++;
            }
            else if (JointTableProbability("yellow", Distance)>=0.5 && grid[x,y]==null){
                Tile color = Instantiate(Resources.Load("Prefabs/yellow", typeof(Tile)), new Vector3(x, y, 0), Quaternion.identity) as Tile;
                grid[x, y]=color;
                Yellows++;
            }
            else if (JointTableProbability("orange", Distance)>=0.5 && grid[x,y]==null){
                Tile color = Instantiate(Resources.Load("Prefabs/orange", typeof(Tile)), new Vector3(x, y, 0), Quaternion.identity) as Tile;
                grid[x, y]=color;
                Oranges++;
            }
        }
    }
}
```

B. ProbabilityText.cs:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using TMPro;

public class ProbabilityText : MonoBehaviour{
    public Game clicked;
    public TextMeshPro probability;
    public double probabilitycount = 0;

    void Start(){
        clicked = FindObjectOfType<Game>() as Game;
        probabilitycount = 0.012;
    }

    // Update is called once per frame
    void Update(){
        CalculateBayesianProbability(clicked.lastClickedX, clicked.lastClickedY, clicked.GhostX, clicked.GhostY);
        probability.text = probabilitycount.ToString();
    }

    void CalculateBayesianProbability(int lastClickedX, int lastClickedY, int GhostX, int GhostY){
        probabilitycount= clicked.JointTableProbability("red", 0);
    }
}
```

The code in **ProbabilityText.cs** involves calculating and displaying a Bayesian probability value.

CalculateBayesianProbability:

This method is used to calculate the Bayesian probability using the provided inputs (lastClickedX, lastClickedY, GhostX, and GhostY).

In the code snippet, the line *“probabilitycount = clicked.JointTableProbability(“red”, 0);”* calculates the probability by calling the JointTableProbability method of the clicked object, which is of type Game. The calculated result is then assigned to the variable probabilitycount.

In brief, the code initializes necessary attributes, locates a ‘Game’ object, calculates probabilities, and updates the TextMeshPro component for display.

V. Video demo on YouTube

Link: https://youtu.be/upx52t09c5A?si=XRC8sLTgnTf_pTB_



Bust The Ghost Unity Game - Bayesian Inferencing



Bust The Ghost

Données analytiques

Modifier la vidéo

3



Partager

Télécharger

Enregistrer

VI. Conclusion

The “Bust the Ghost” game project represents a captivating and demanding endeavor in developing a probability-based game. The project necessitates the establishment of a uniform distribution of the ghost, the incorporation of a conditional probability distribution for determining cell colors, and the application of Bayesian inference to update the posterior probability of the ghost. These requisites present an excellent opportunity to refine programming skills in C# and Unity development, probability distributions, and Bayesian inference, all within the enjoyable context of gameplay 😊.

Indeed, the Unity environment played a pivotal role in shaping the project, offering a user-friendly interface and powerful features for building the game elements. From creating the 7x13 grid to implementing interactive buttons and handling user interactions, Unity proved instrumental in ensuring a seamless and visually appealing gaming experience. However, it is important to note that the journey was not without its challenges 😞. The team encountered difficulties during the installation of the Unity editor, initially posing a hurdle in the project’s initiation. Additionally, familiarizing ourselves with Unity graphics presented a learning curve that required dedicated effort and time 😊.

Moreover, the adoption of C# as the primary programming language for the project introduced its own set of challenges. Learning C# from scratch in less than 10 days demanded an accelerated pace of comprehension and implementation. Despite these challenges, the team persevered and successfully overcame these obstacles 🙌, showcasing adaptability and resilience in the face of a dynamic development environment.

The gameplay mechanics of “Bust the Ghost” were analyzed, highlighting the strategic challenge posed to players. The incorporation of Bayesian inference and probability updates after each player click added a unique layer of challenge, requiring thoughtful decision-making and deductive skills. The credit system, limited busts, and color-coded feedback contributed to a dynamic and immersive gaming environment.

In summary, “Bust the Ghost” exemplifies the synergy between AI techniques and game development, providing players with an intellectually stimulating experience. The project showcases the potential of combining Unity’s capabilities with probabilistic inferencing, laying the foundation for future innovations at the intersection of Artificial Intelligence and gaming.