

Poker AI

Artificial Intelligence Coursework 2



Daniele Abbagnale

21706092

Contents

[Plans & Pre-programming Ideas 3](#_Toc534973223)

[Objectives 3](#_Toc534973224)

[FSM 3](#_Toc534973225)

[Utility 5](#_Toc534973226)

[Pseudo Code 6](#_Toc534973227)

[Fuzzy Logic Values 6](#_Toc534973228)

[Post-Programming 7](#_Toc534973229)

[Time Constraints 7](#_Toc534973230)

[Overview 7](#_Toc534973231)

[Changes Made 7](#_Toc534973232)

[Fold Utility Values 7](#_Toc534973233)

[Raise Utility Values 7](#_Toc534973234)

[Screenshots 8](#_Toc534973235)

[Code 9](#_Toc534973236)

[TableBehaviour.cs 9](#_Toc534973237)

[TableCardHolder.cs 19](#_Toc534973238)

[Deal.cs 20](#_Toc534973239)

[FirstRoundBetting.cs 22](#_Toc534973240)

[Flop.cs 23](#_Toc534973241)

[OnePlayerLeft.cs 24](#_Toc534973242)

[PreFlop.cs 25](#_Toc534973243)

[ResetTable.cs 27](#_Toc534973244)

[River.cs 29](#_Toc534973245)

[Showdown.cs 30](#_Toc534973246)

[PlayerController.cs 32](#_Toc534973247)

[Card.cs 35](#_Toc534973248)

[CardHolder.cs 36](#_Toc534973249)

[CardPlayer.cs 37](#_Toc534973250)

[CheckHand.cs 39](#_Toc534973251)

[Hand.cs 57](#_Toc534973252)

[AIBehaviour.cs 58](#_Toc534973253)

[Bet.cs 61](#_Toc534973254)

[Call.cs 62](#_Toc534973255)

[Check.cs 63](#_Toc534973256)

[EndTurn.cs 64](#_Toc534973257)

[Fold.cs 65](#_Toc534973258)

[Play.cs 66](#_Toc534973259)

[Raise.cs 70](#_Toc534973260)

[WaitForTurn.cs 71](#_Toc534973261)

# Plans & Pre-programming Ideas

## Objectives

The objective of this simulation/game is to create a dynamic and lifelike computer opponent at a poker game using a combination of AI techniques. The computer opponent (AI) will dynamically change its play depending on the situation of the match, or attempt to bluff over players to seize victory. This will be achieved through 2 major AI techniques and 1 minor technique. (Major and Minor are determined by their influences on the AI’s decisions).

## FSM

The first major technique is the Finite state machine.

The scene will include 2 finite state machines. One for the table, and another for the AI player. The table’s finite state machine will be responsible for changing the round for the match and its associated rules and conditions, then finally determine a winner based on the cards left in play. The states will include;

* Deal
  + Guard Condition
    - Once all players have been dealt 2 cards.
* FirstRoundBetting
  + Guard Condition
    - Once all players have the same bet amount and its currently the dealers turn.
* PreFlop
  + Guard Condition
    - Once all players have the same bet amount and its currently the dealers turn.
* Flop
  + Guard Condition
    - Once all players have the same bet amount and its currently the dealers turn.
* River
  + Guard Condition
    - Once all players have the same bet amount and its currently the dealers turn.
* Showdown
  + Guard Condition
    - Once a winner has been determined and have been credited the money in the pot.
* OnePlayerLeft
  + Guard Condition
    - Once the winner has been credited the money in the pot.
* ResetTable
  + Guard Condition
    - Once a match has concluded.



The AI player’s finite state machine, will determine what action the AI will take after using the other techniques. Each state will be used to segment the code for the actions the AI will take once a decision has been made. The states will include;

* WaitForTurn
  + Guard Condition
    - Once the table gives the go-ahead for this player’s turn.
* Play
  + Guard Condition
    - Once a decision of the next action has been made.
* Bet
  + Guard Condition
    - State finished
* Call
  + Guard Condition
    - State finished
* Raise
  + Guard Condition
    - State finished
* Check
  + Guard Condition
    - State finished
* Fold
  + Guard Condition
    - State finished
* EndTurn
  + Guard Condition
    - Once a decision and action has been taken, and the table has been notified.



## Utility

The second major technique are utility values.

The AI players will use utility values which change dynamically throughout the match to make the decisions. The decisions made won’t always be “safe” choices, and the AI may bluff depending on these values.

The utility values will also be heavily influenced by the minor AI technique, fuzzy logic values.

The utility values will be based on several parameters, these will include;

* Probability values for each choice
* The value of their current hand (I.E. straight, pair, etc…)
* The round number
* The amount of legitimate options available
* The money available.

Once all the utility values have been calculated, a random number will be picked between 0 and the total of the utility values. Then, the decision made will be selected based on if the random number picked is within range of the utility values.

### Pseudo Code

Below are some examples of how the utility values will be calculated and used within the game. Since there are minor differences between every option, the only ones listed here are those with major changes.

#### Check

checkUtilityValue = probabilityOfCheck \* handValue \* roundNumber

Where probabilityOfCheck is a variable between 0-1

#### Fold

foldUtilityValue = (handValue \* -1) + probabilityOfFold \* roundNumber

Where probabilityOfFold is a variable between 0-1

In this instance is the AI player had only a HighCard by round 10, the following may be processed.

checkUtilityValue = 0.3 \* 1 \* 10; checkUtilityValue = 3

foldUtilityValue = (1 \* -1) + 0.5 \* 10; foldUtilityValue = 4

In the example above, the AI player has a high chance of folding this turn due to a poor hand mid-late game.

## Fuzzy Logic Values

The third technique used has a minor influence on the AI’s decision, fuzzy logic.

Fuzzy logic will be used to dynamically change the utility values to affect the AI’s decision. The main elements will be, fuzzification and fuzzy NOT. These will be used to affect raise and fold, since the higher the value of the hand the lower the chances of folding, and the higher the chances of raising. The NOT is used to reverse the value of the hand for fold.

# Post-Programming

## Time Constraints

Due to time constraints, the overall image and functionality of the game had to be cut to best demonstrate the AI. Features like aminations and extreme rules were ultimately removed since they were game breaking for the time allocated. Commented code represents code removed due to time constraints.

## Overview

After implementation of the AI techniques above, several changes were made to achieve the desired result of the AI’s behaviour.

The AI was to ultimately play safely with the occasional bluff to fool other players into folding before the showdown stage. However, with the current plans above, the AI attempted to bluff on too many occasions without backing down, which ultimately resulted in the AI losing a considerable amount of money before gaining money several matches later.

With the plans above, there was a very fine line between safe AI players and bluffing players. Therefore, changes were made to the fuzzy logic values and several utility values to make bluffing players more likely to back-down if no-one falls for the bluff.

## Changes Made

### Fold Utility Values

#### Before

foldUtilityValue = (handValue \* -1) + probabilityOfFold \* roundNumber

#### After

foldUtilityValue = (handValue \* -1) + (probabilityOfFold + fuzzyNOT(FuzzyHandValue) /10) \* roundNumber

These changes although seemingly random, make major changes to the way the AI now behaves.

Now the chances of the AI player folding once the round number has increased on a BLUFF, has increased. If however, the AI player legitimately has a good hand, the chances of folding decrease.

### Raise Utility Values

#### Before

raiseUtilityValue = (fuzzyHandValue)\* probabilityOfRaise \* roundNumber

#### After

raiseUtilityValue = (standardHandValue)\* probabilityOfRaise \* roundNumber

Although the change was not major in code, this simple change had a big impact on the AI’s behaviour when it came to raising the stakes.

Before, the AI would raise too frequently, making games much more drawn out and unplayable.

Now, the AI will raise less frequently and only if they are bluffing or hand a good hand.

# Screenshots

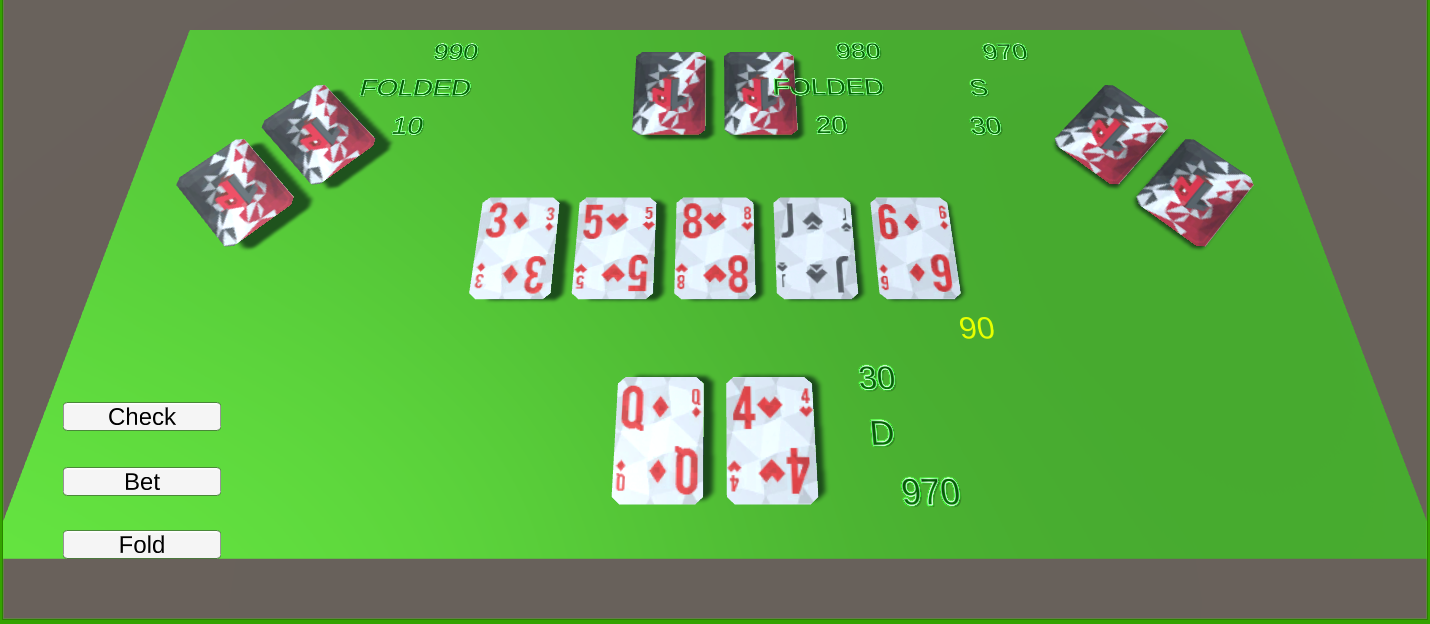


Figure , 2 AI players folding due to bad hands. 3rd AI player bluffing

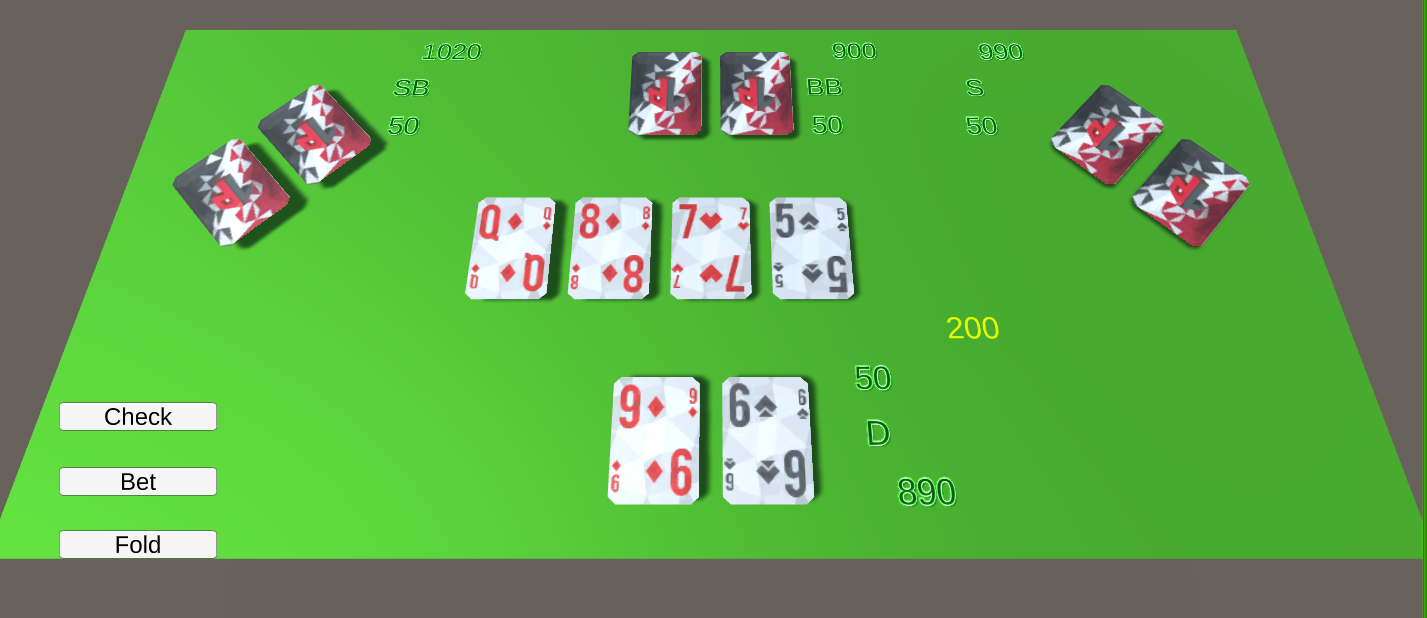


Figure , 1st AI player attempting to bluff

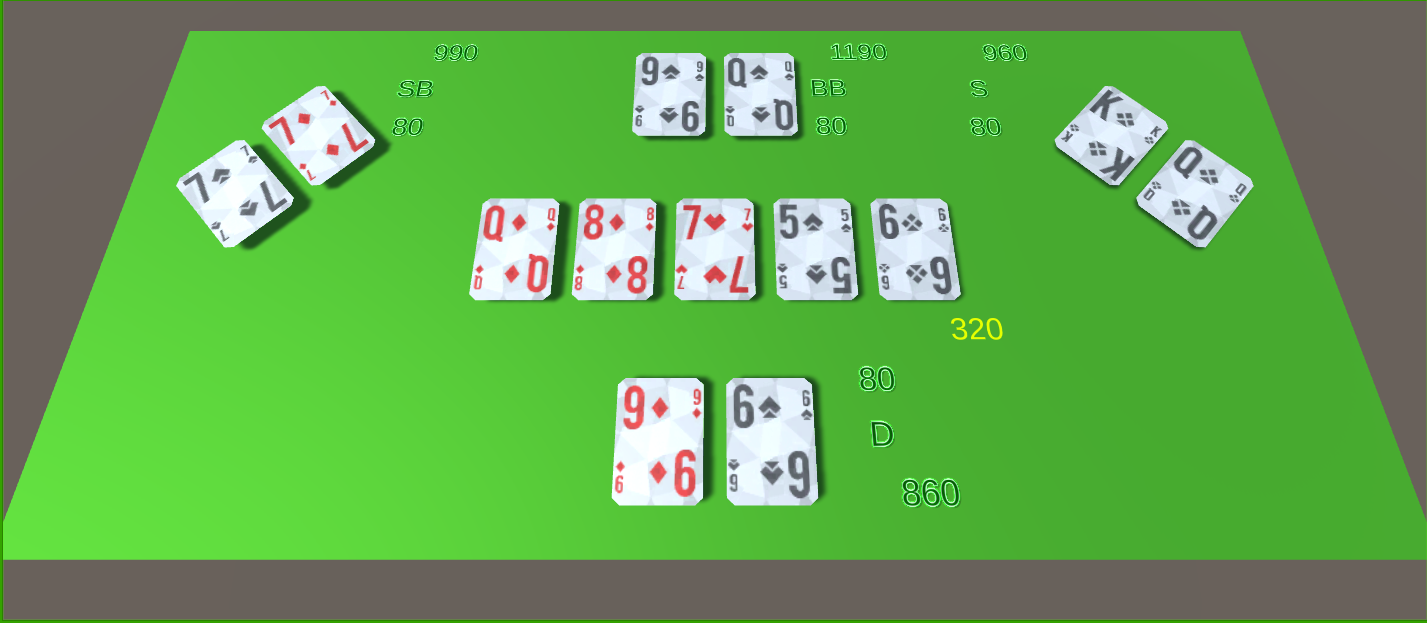


Figure , 2nd AI player winning with a straight (5,6,7,8,9)+Q

# Code

## TableBehaviour.cs

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

public class TableBehaviour : MonoBehaviour {

public enum tableStates { Deal, Firstroundbetting, Preflop, Flop, River, Showdown, ResetTable, OnePlayerLeft};

[HideInInspector]

public FSM<tableStates> tableFSM;

public enum Role { NA, S, D, SB, BB}; //Enum to assign roles

[HideInInspector]

public static TableBehaviour tb;

[Header("Card Graphics")]

public GameObject cardBack; //Back of the card image

public GameObject[] Hearts; //All heart cards

public GameObject[] Clubs; //All club cards

public GameObject[] Diamonds; //All diamond cards

public GameObject[] Spades; //All spades cards

[Space]

[Header("Table Card Holder")]

public TableCardHolder table; //Tables card holders

[Space]

[Header("Players Controllers")]

public List<CardPlayer> players; //All players in the match

private List<CardPlayer> allPlayers;

[Space]

[Header("Game Limits & Rules")]

public float startMoney = 1000f; //The money every player starts with

public float minBetPercentageOfStartMoney = 0.01f; //The percentage from the startmoney that is a minimum bet

[HideInInspector]

public float minBet;

[HideInInspector]

public int playerTurn = 0; //Stores the current players turn

[HideInInspector]

public int previousPlayerTurn = 0; //Previous players turn

[HideInInspector]

public bool turnFinished = false; //Stores if the current player has finished their turn

[HideInInspector]

public int dealerID; //The player who is a dealer

[HideInInspector]

public bool play = false; //Determins if the match is playing

[HideInInspector]

public int roundCount = 1; //The round number

[HideInInspector]

public List<GameObject> cardObjects;

public float cardDealTime = 1.0f; //Redundant variable (was used for animations before they were removed due to time constraints)

[HideInInspector]

public float pot; //Value of the pot

[HideInInspector]

public float lastBet; //The value of the last bet to the pot

private void Awake()

{

tb = this;

minBet = startMoney \* minBetPercentageOfStartMoney;

}

void Start () {

allPlayers = new List<CardPlayer>();

for (int i = 0; i < players.Count; i++)

{

allPlayers.Add(players[i]);

}

InitialiseFSM();

ResetTable();

}

void Update () {

tableFSM.CurrentState.Act();

tableFSM.Check();

}

public void ResetTable()

{

//playerTurn = Random.Range(0, players.Count); //If the dealer is too be randomised, use this code.

playerTurn = 0;

pot = 0;

lastBet = 0;

players.Clear();

players = new List<CardPlayer>();

for (int i = 0; i < allPlayers.Count; i++)

{

players.Add(allPlayers[i]);

}

play = false;

}

private void InitialiseFSM()

{

tableFSM = new FSM<tableStates>();

tableFSM.AddState(new Deal<tableStates>(tableStates.Deal, this, 1f));

tableFSM.AddState(new Firstroundbetting<tableStates>(tableStates.Firstroundbetting, this, 0f));

tableFSM.AddState(new Preflop<tableStates>(tableStates.Preflop, this, 0.2f));

tableFSM.AddState(new Flop<tableStates>(tableStates.Flop, this, 0.2f));

tableFSM.AddState(new River<tableStates>(tableStates.River, this, 0.2f));

tableFSM.AddState(new Showdown<tableStates>(tableStates.Showdown, this, 10f));

tableFSM.AddState(new ResetTable<tableStates>(tableStates.ResetTable, this, 0.5f));

tableFSM.AddState(new OnePlayerLeft<tableStates>(tableStates.OnePlayerLeft, this, 0.5f));

tableFSM.AddTransition(tableStates.Deal, tableStates.Firstroundbetting);

tableFSM.AddTransition(tableStates.Firstroundbetting, tableStates.Preflop);

tableFSM.AddTransition(tableStates.Preflop, tableStates.Flop);

tableFSM.AddTransition(tableStates.Flop, tableStates.River);

tableFSM.AddTransition(tableStates.River, tableStates.Showdown);

tableFSM.AddTransition(tableStates.Showdown, tableStates.ResetTable);

tableFSM.AddTransition(tableStates.OnePlayerLeft, tableStates.ResetTable);

tableFSM.AddTransition(tableStates.ResetTable, tableStates.Deal);

tableFSM.AddTransition(tableStates.Showdown, tableStates.OnePlayerLeft);

tableFSM.SetInitialState(tableStates.ResetTable);

}

public bool GuardOnePlayerLeftToResetTable(State<tableStates> currentState)

{

return currentState.StateFinished;

}

public bool GuardShowdownToOnePlayerLeft(State<tableStates> currentstate)

{

return IsOnePlayerLeft() && currentstate.StateFinished;

}

public bool GuardDealToFirstroundbetting(State<tableStates> currentState)

{

if (currentState.StateFinished)

play = true;

return currentState.StateFinished;

}

public bool GuardFirstroundbettingToPreflop(State<tableStates> currentState)

{

return currentState.StateFinished;

}

public bool GuardPreflopToFlop(State<tableStates> currentState)

{

return currentState.StateFinished;

}

public bool GuardFlopToRiver(State<tableStates> currentState)

{

return currentState.StateFinished;

}

public bool GuardRiverToShowdown(State<tableStates> currentState)

{

return currentState.StateFinished;

}

public bool GuardShowdownToResetTable(State<tableStates> currentState)

{

return currentState.StateFinished && !IsOnePlayerLeft();

}

public bool GuardResetTableToDeal(State<tableStates> currentState)

{

return currentState.StateFinished;

}

public bool IsOnePlayerLeft()

{

if (players.Count == 1)

{

play = false;

return true;

}

return false;

}

public void AddToPot(float value)

{

pot += value;

table.potText.text = pot.ToString();

lastBet = value;

}

public void CreateCardObject(Transform position)

{

GameObject card = Instantiate(cardBack, position);

cardObjects.Add(card);

}

public void DestroyAllCards()

{

for (int i = 0; i < cardObjects.Count; i++)

{

Destroy(cardObjects[i]);

}

cardObjects.Clear();

cardObjects = new List<GameObject>();

}

public GameObject DisplayCard(Card card, Transform position)

{

switch (card.suit)

{

case Card.Suits.H:

GameObject cardHearts = Instantiate(Hearts[card.value - 2], position);

cardObjects.Add(cardHearts);

return cardHearts;

case Card.Suits.C:

GameObject cardClubs = Instantiate(Clubs[card.value - 2], position);

cardObjects.Add(cardClubs);

return cardClubs;

case Card.Suits.D:

GameObject cardDiamonds = Instantiate(Diamonds[card.value - 2], position);

cardObjects.Add(cardDiamonds);

return cardDiamonds;

case Card.Suits.S:

GameObject cardSpades = Instantiate(Spades[card.value - 2], position);

cardObjects.Add(cardSpades);

return cardSpades;

}

return null;

}

public Card PickACard()

{

bool unPicked = false;

Card card = new Card(Card.Suits.C, 1);

while (unPicked != true)

{

int suit = Random.Range(0, 40);

if (suit <= 10)

{

card.suit = Card.Suits.H;

}

else if (suit > 10 && suit <= 20)

{

card.suit = Card.Suits.C;

}

else if (suit > 20 && suit <= 30)

{

card.suit = Card.Suits.D;

}

else

{

card.suit = Card.Suits.S;

}

int value = Random.Range(0, 130);

if (value <= 10)

{

card.value = 2;

}

else if (value > 10 && value <= 20)

{

card.value = 3;

}

else if (value > 20 && value <= 30)

{

card.value = 4;

}

else if (value > 30 && value <= 40)

{

card.value = 5;

}

else if (value > 40 && value <= 50)

{

card.value = 6;

}

else if (value > 50 && value <= 60)

{

card.value = 7;

}

else if (value > 60 && value <= 70)

{

card.value = 8;

}

else if (value > 70 && value <= 80)

{

card.value = 9;

}

else if (value > 80 && value <= 90)

{

card.value = 10;

}

else if (value > 90 && value <= 100)

{

card.value = 11;

}

else if (value > 100 && value <= 110)

{

card.value = 12;

}

else if (value > 110 && value <= 120)

{

card.value = 13;

}

else

{

card.value = 14;

}

if (IsCardInPlay(card))

{

unPicked = false;

}

else

{

unPicked = true;

}

}

return card;

}

private bool IsCardInPlay(Card cardToCheck)

{

foreach (CardPlayer player in players)

{

if (player.cardHolder.card1 != null)

{

if (player.cardHolder.card1.suit == cardToCheck.suit && player.cardHolder.card1.value == cardToCheck.value)

{

Debug.Log("Same card found");

return true;

}

}

if (player.cardHolder.card2 != null)

{

if (player.cardHolder.card2.suit == cardToCheck.suit && player.cardHolder.card2.value == cardToCheck.value)

{

Debug.Log("Same card found");

return true;

}

}

}

if (table.card1 != null)

{

if (table.card1.suit == cardToCheck.suit && table.card1.value == cardToCheck.value)

{

Debug.Log("Same card found");

return true;

}

}

if (table.card2 != null)

{

if (table.card2.suit == cardToCheck.suit && table.card2.value == cardToCheck.value)

{

Debug.Log("Same card found");

return true;

}

}

if (table.card3 != null)

{

if (table.card3.suit == cardToCheck.suit && table.card3.value == cardToCheck.value)

{

Debug.Log("Same card found");

return true;

}

}

if (table.card4 != null)

{

if (table.card4.suit == cardToCheck.suit && table.card4.value == cardToCheck.value)

{

Debug.Log("Same card found");

return true;

}

}

if (table.card5 != null)

{

if (table.card5.suit == cardToCheck.suit && table.card5.value == cardToCheck.value)

{

Debug.Log("Same card found");

return true;

}

}

return false;

}

public void IncrementWhosTurn()

{

previousPlayerTurn = playerTurn;

playerTurn++;

if (playerTurn >= players.Count)

playerTurn = 0;

Debug.Log("Player turn is: " + playerTurn + ". Round count is: " + roundCount);

}

public void Check()

{

if (!play)

return;

if (!turnFinished)

return;

turnFinished = false;

roundCount++;

IncrementWhosTurn();

players[playerTurn].Play();

}

public void RemovePlayerFromMatch(CardPlayer playerToRemove)

{

int playerIndex = -1;

for (int i = 0; i < players.Count; i++)

{

if (players[i] == playerToRemove)

playerIndex = i;

}

if (playerIndex == 0)

previousPlayerTurn = players.Count;

else

previousPlayerTurn = playerTurn--;

if(playerIndex != -1)

{

players[playerIndex].roleText.text = "FOLDED";

players.RemoveAt(playerIndex);

}

}

public int ReturnPreviousPlayer()

{

return previousPlayerTurn;

}

}

## TableCardHolder.cs

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

using TMPro;

public class TableCardHolder : CardHolder {

public Card card3;

public Transform card3Position;

public Card card4;

public Transform card4Position;

public Card card5;

public Transform card5Position;

public TextMeshProUGUI potText;

public void ClearTable()

{

potText.text = "";

card1 = null;

card2 = null;

card3 = null;

card4 = null;

card5 = null;

}

}

## Deal.cs

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

public class Deal<T> : State<T> {

private TableBehaviour tableBehaviour;

private bool dealCards = false;

private float currentDealTime = 0f;

private int cardToDealTo = 0;

public Deal(T stateName, TableBehaviour controller, float minDuration) : base(stateName, controller, minDuration)

{

tableBehaviour = controller;

}

public override void OnEnter()

{

dealCards = true;

Debug.Log(tableBehaviour.dealerID);

Debug.Log(tableBehaviour.playerTurn);

for (int i = 0; i < tableBehaviour.players.Count; i++)

{

tableBehaviour.players[i].cardHolder.card1 = new Card(Card.Suits.C, 14);

tableBehaviour.players[i].cardHolder.card1 = tableBehaviour.PickACard();

tableBehaviour.CreateCardObject(tableBehaviour.players[i].cardHolder.card1Position);

tableBehaviour.players[i].cardHolder.card2 = new Card(Card.Suits.C, 14);

tableBehaviour.players[i].cardHolder.card2 = tableBehaviour.PickACard();

tableBehaviour.CreateCardObject(tableBehaviour.players[i].cardHolder.card2Position);

}

ShowPlayerCards();

ShowCardsInConsole();

dealCards = false;

tableBehaviour.turnFinished = true;

stateFinished = true;

}

/// <summary>

/// This code used to animate cards to each player

/// Was disabled due to time constraints, please ignore.

/// </summary>

//public override void Act()

//{

//if (!dealCards)

// return;

//if(currentDealTime <= 0f)

//{

// if(tableBehaviour.playerTurn == tableBehaviour.dealerID && cardToDealTo >= 2)

// {

// //EXIT STATE CONDITIONS HERE!

// ShowCardsInConsole();

// ShowPlayerCards();

// tableBehaviour.turnFinished = true;

// dealCards = false;

// stateFinished = true;

// }

// else

// {

// if (cardToDealTo == 0)

// {

// tableBehaviour.playerCards[tableBehaviour.playerTurn].card1 = new Card(Card.Suits.C, 14);

// tableBehaviour.playerCards[tableBehaviour.playerTurn].card1 = tableBehaviour.PickACard();

// tableBehaviour.CreateCardObject(tableBehaviour.playerCards[tableBehaviour.playerTurn].card1Position);

// }

// else if (cardToDealTo == 1)

// {

// tableBehaviour.playerCards[tableBehaviour.playerTurn].card2 = new Card(Card.Suits.C, 14);

// tableBehaviour.playerCards[tableBehaviour.playerTurn].card2 = tableBehaviour.PickACard();

// tableBehaviour.CreateCardObject(tableBehaviour.playerCards[tableBehaviour.playerTurn].card2Position);

// }

// if (tableBehaviour.playerTurn == tableBehaviour.dealerID)

// {

// cardToDealTo++;

// }

// tableBehaviour.IncrementWhosTurn();

// }

// currentDealTime = 1f / tableBehaviour.cardDealTime;

//}

//currentDealTime -= Time.deltaTime;

//}

private void ShowCardsInConsole()

{

foreach (CardPlayer player in tableBehaviour.players)

{

Debug.Log(player.ToString() + "'s first card is: " + player.cardHolder.card1.suit + player.cardHolder.card1.value);

Debug.Log(player.ToString() + "'s second card is: " + player.cardHolder.card2.suit + player.cardHolder.card2.value);

}

}

private void ShowPlayerCards()

{

tableBehaviour.DisplayCard(tableBehaviour.players[0].cardHolder.card1, tableBehaviour.players[0].cardHolder.card1Position);

tableBehaviour.DisplayCard(tableBehaviour.players[0].cardHolder.card2, tableBehaviour.players[0].cardHolder.card2Position);

}

}

## FirstRoundBetting.cs

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

public class Firstroundbetting<T> : State<T>

{

private TableBehaviour tableBehaviour;

public Firstroundbetting(T stateName, TableBehaviour controller, float minDuration) : base(stateName, controller, minDuration)

{

tableBehaviour = controller;

}

public override void OnEnter()

{

stateFinished = false;

tableBehaviour.roundCount++;

base.OnEnter();

}

public override void Act()

{

if (stateFinished)

return;

if (tableBehaviour.IsOnePlayerLeft())

{

stateFinished = true;

return;

}

tableBehaviour.Check();

if (tableBehaviour.playerTurn == tableBehaviour.dealerID)

{

float playerBet = tableBehaviour.players[0].currentBet;

bool allEqual = true;

for (int i = 1; i < tableBehaviour.players.Count; i++)

{

if (tableBehaviour.players[i].currentBet != playerBet)

allEqual = false;

}

if (allEqual)

{

stateFinished = true;

}

}

}

}

## Flop.cs

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

public class Flop<T> : State<T>

{

private TableBehaviour tableBehaviour;

private int currentEntryRoundCount = 0;

public Flop(T stateName, TableBehaviour controller, float minDuration) : base(stateName, controller, minDuration)

{

tableBehaviour = controller;

}

public override void OnEnter()

{

stateFinished = false;

Debug.Log("Flop");

if (tableBehaviour.IsOnePlayerLeft())

{

stateFinished = true;

return;

}

tableBehaviour.table.card4 = tableBehaviour.PickACard();

tableBehaviour.DisplayCard(tableBehaviour.table.card4, tableBehaviour.table.card4Position);

tableBehaviour.roundCount++;

currentEntryRoundCount = tableBehaviour.roundCount;

}

public override void Act()

{

if (stateFinished)

return;

if (tableBehaviour.IsOnePlayerLeft())

{

stateFinished = true;

return;

}

tableBehaviour.Check();

if (tableBehaviour.playerTurn == tableBehaviour.dealerID)

{

float playerBet = tableBehaviour.players[0].currentBet;

bool allEqual = true;

for (int i = 1; i < tableBehaviour.players.Count; i++)

{

if (tableBehaviour.players[i].currentBet != playerBet)

allEqual = false;

}

if (allEqual && currentEntryRoundCount != tableBehaviour.roundCount)

{

stateFinished = true;

}

}

}

}

## OnePlayerLeft.cs

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

public class OnePlayerLeft<T> : State<T> {

private TableBehaviour tableBehaviour;

public OnePlayerLeft(T stateName, TableBehaviour controller, float minDuration) : base(stateName, controller, minDuration)

{

tableBehaviour = controller;

}

public override void OnEnter()

{

Debug.Log(tableBehaviour.players[0].ToString() + " wins!");

tableBehaviour.players[0].IncreaseMoney(tableBehaviour.pot);

tableBehaviour.DisplayCard(tableBehaviour.players[0].cardHolder.card1, tableBehaviour.players[0].cardHolder.card1Position);

tableBehaviour.DisplayCard(tableBehaviour.players[0].cardHolder.card2, tableBehaviour.players[0].cardHolder.card2Position);

stateFinished = true;

}

}

## PreFlop.cs

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

public class Preflop<T> : State<T>

{

private TableBehaviour tableBehaviour;

private int currentEntryRoundCount = 0;

public Preflop(T stateName, TableBehaviour controller, float minDuration) : base(stateName, controller, minDuration)

{

tableBehaviour = controller;

}

public override void OnEnter()

{

stateFinished = false;

Debug.Log("Preflop");

if (tableBehaviour.IsOnePlayerLeft())

{

stateFinished = true;

return;

}

tableBehaviour.table.card1 = tableBehaviour.PickACard();

tableBehaviour.table.card2 = tableBehaviour.PickACard();

tableBehaviour.table.card3 = tableBehaviour.PickACard();

tableBehaviour.DisplayCard(tableBehaviour.table.card1, tableBehaviour.table.card1Position);

tableBehaviour.DisplayCard(tableBehaviour.table.card2, tableBehaviour.table.card2Position);

tableBehaviour.DisplayCard(tableBehaviour.table.card3, tableBehaviour.table.card3Position);

tableBehaviour.roundCount++;

currentEntryRoundCount = tableBehaviour.roundCount;

}

public override void Act()

{

if (stateFinished)

return;

if (tableBehaviour.IsOnePlayerLeft())

{

stateFinished = true;

return;

}

tableBehaviour.Check();

if (tableBehaviour.playerTurn == tableBehaviour.dealerID)

{

float playerBet = tableBehaviour.players[0].currentBet;

bool allEqual = true;

for (int i = 1; i < tableBehaviour.players.Count; i++)

{

if (tableBehaviour.players[i].currentBet != playerBet)

allEqual = false;

}

if (allEqual && currentEntryRoundCount != tableBehaviour.roundCount)

{

stateFinished = true;

}

}

}

}

## ResetTable.cs

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

public class ResetTable<T> : State<T> {

private TableBehaviour tableBehaviour;

public ResetTable(T stateName, TableBehaviour controller, float minDuration) : base(stateName, controller, minDuration)

{

tableBehaviour = controller;

}

public override void OnEnter()

{

base.OnEnter();

tableBehaviour.ResetTable();

tableBehaviour.play = false;

for (int i = 0; i < tableBehaviour.players.Count; i++)

{

tableBehaviour.players[i].resetToPlay();

}

tableBehaviour.roundCount = 1;

tableBehaviour.DestroyAllCards();

tableBehaviour.table.ClearTable();

AssignRole();

stateFinished = true;

}

private void AssignRole()

{

for (int j = 0; j < tableBehaviour.players.Count; j++)

{

if (tableBehaviour.players[TableBehaviour.tb.playerTurn].role == TableBehaviour.Role.NA)

{

switch (j)

{

case 0:

tableBehaviour.players[TableBehaviour.tb.playerTurn].role = TableBehaviour.Role.D;

break;

case 1:

tableBehaviour.players[TableBehaviour.tb.playerTurn].role = TableBehaviour.Role.SB;

break;

case 2:

tableBehaviour.players[TableBehaviour.tb.playerTurn].role = TableBehaviour.Role.BB;

break;

default:

tableBehaviour.players[TableBehaviour.tb.playerTurn].role = TableBehaviour.Role.S;

break;

}

tableBehaviour.players[TableBehaviour.tb.playerTurn].roleText.text = tableBehaviour.players[TableBehaviour.tb.playerTurn].role.ToString();

if (tableBehaviour.players[TableBehaviour.tb.playerTurn].role == TableBehaviour.Role.D)

tableBehaviour.dealerID = tableBehaviour.playerTurn;

}

tableBehaviour.IncrementWhosTurn();

}

}

}

## River.cs

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

public class River<T> : State<T>

{

private TableBehaviour tableBehaviour;

private int currentEntryRoundCount = 0;

public River(T stateName, TableBehaviour controller, float minDuration) : base(stateName, controller, minDuration)

{

tableBehaviour = controller;

}

public override void OnEnter()

{

stateFinished = false;

Debug.Log("River");

if (tableBehaviour.IsOnePlayerLeft())

{

stateFinished = true;

return;

}

tableBehaviour.table.card5 = tableBehaviour.PickACard();

tableBehaviour.DisplayCard(tableBehaviour.table.card5, tableBehaviour.table.card5Position);

tableBehaviour.roundCount++;

currentEntryRoundCount = tableBehaviour.roundCount;

}

public override void Act()

{

if (stateFinished)

return;

if (tableBehaviour.IsOnePlayerLeft())

{

stateFinished = true;

return;

}

tableBehaviour.Check();

if (tableBehaviour.playerTurn == tableBehaviour.dealerID)

{

float playerBet = tableBehaviour.players[0].currentBet;

bool allEqual = true;

for (int i = 1; i < tableBehaviour.players.Count; i++)

{

if (tableBehaviour.players[i].currentBet != playerBet)

allEqual = false;

}

if (allEqual && currentEntryRoundCount != tableBehaviour.roundCount)

{

stateFinished = true;

}

}

}

}

## Showdown.cs

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

public class Showdown<T> : State<T>

{

private TableBehaviour tableBehaviour;

private struct Winner

{

public int handValue;

public int playerID;

public int valueOfAllCards;

public Winner(int value1, int value2)

{

handValue = value1;

playerID = value2;

valueOfAllCards = 0;

}

public void SetValueOfAllCards(int value)

{

valueOfAllCards = value;

}

}

public Showdown(T stateName, TableBehaviour controller, float minDuration) : base(stateName, controller, minDuration)

{

tableBehaviour = controller;

}

public override void OnEnter()

{

base.OnEnter();

Debug.Log("SHOWDOWN");

for (int i = 0; i < tableBehaviour.players.Count; i++)

{

tableBehaviour.players[i].canPlay = false;

}

if (tableBehaviour.IsOnePlayerLeft())

{

stateFinished = true;

return;

}

ShowDown();

stateFinished = true;

}

private void ShowDown()

{

tableBehaviour.play = false;

for (int i = 0; i < tableBehaviour.players.Count; i++)

{

Hand hand = tableBehaviour.players[i].myHand();

Debug.Log(hand.hands.ToString());

}

int bestValue = 0;

int handValue = 0;

int playerID = 0;

List<Winner> potentialWinners = new List<Winner>();

for (int i = 0; i < tableBehaviour.players.Count; i++)

{

tableBehaviour.IncrementWhosTurn();

handValue = (int)tableBehaviour.players[tableBehaviour.playerTurn].hand.hands;

tableBehaviour.DisplayCard(tableBehaviour.players[tableBehaviour.playerTurn].cardHolder.card1, tableBehaviour.players[tableBehaviour.playerTurn].cardHolder.card1Position);

tableBehaviour.DisplayCard(tableBehaviour.players[tableBehaviour.playerTurn].cardHolder.card2, tableBehaviour.players[tableBehaviour.playerTurn].cardHolder.card2Position);

playerID = tableBehaviour.playerTurn;

if (handValue > bestValue)

{

bestValue = handValue;

potentialWinners = new List<Winner>();

Winner temp = new Winner(bestValue,playerID);

potentialWinners.Add(temp);

}

else if (handValue == bestValue)

{

bestValue = handValue;

Winner temp = new Winner(bestValue, playerID);

potentialWinners.Add(temp);

}

}

if(potentialWinners.Count == 1)

{

Debug.Log(tableBehaviour.players[potentialWinners[0].playerID].ToString() + " wins!");

tableBehaviour.players[potentialWinners[0].playerID].IncreaseMoney(tableBehaviour.pot);

}

else

{

for (int i = 0; i < potentialWinners.Count; i++)

{

potentialWinners[i].SetValueOfAllCards(CheckHand.checkHand.CardsInHandValue(tableBehaviour.players[potentialWinners[i].playerID].hand.cards));

}

Winner winner = new Winner(0,0);

for (int i = 0; i < potentialWinners.Count; i++)

{

if (potentialWinners[i].valueOfAllCards > winner.valueOfAllCards)

winner = potentialWinners[i];

}

Debug.Log(tableBehaviour.players[winner.playerID].ToString() + " wins!");

tableBehaviour.players[winner.playerID].IncreaseMoney(tableBehaviour.pot);

}

}

}

## PlayerController.cs

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

using UnityEngine.UI;

public class PlayerController : CardPlayer

{

public GameObject UI;

public GameObject check;

public GameObject call;

public GameObject raise;

public GameObject bet;

public GameObject fold;

private void Awake()

{

cardHolder = GetComponent<CardHolder>();

moneyText.text = money.ToString();

money = TableBehaviour.tb.startMoney;

}

private void Update()

{

if (!canPlay)

return;

if(canPlay)

{

HideSelectOptions();

ShowHideALLOptions(true);

canPlay = false;

}

}

private void EndPlay()

{

if (currentBet > 0)

boughtin = true;

ShowAllSelectOptions();

ShowHideALLOptions(false);

TableBehaviour.tb.turnFinished = true;

}

public void AddMoneyToTable(float value)

{

TableBehaviour.tb.AddToPot(value);

currentBet += value;

betText.text = currentBet.ToString();

}

private void HideSelectOptions()

{

if((role == TableBehaviour.Role.SB) && !boughtin)

{

ShowAllSelectOptions();

check.SetActive(false);

call.SetActive(false);

raise.SetActive(false);

}

else if ((role == TableBehaviour.Role.BB) && !boughtin)

{

ShowAllSelectOptions();

check.SetActive(false);

call.SetActive(false);

raise.SetActive(false);

}

else

{

if (TableBehaviour.tb.players[previousPlayer].currentBet == currentBet)

{

call.SetActive(false);

raise.SetActive(false);

}

else

{

check.SetActive(false);

bet.SetActive(false);

}

}

}

private void ShowAllSelectOptions()

{

check.SetActive(true);

bet.SetActive(true);

call.SetActive(true);

raise.SetActive(true);

}

private void ShowHideALLOptions(bool enable)

{

UI.SetActive(enable);

}

public void Bet()

{

if((role == TableBehaviour.Role.SB) && !boughtin)

{

DecreaseMoney(TableBehaviour.tb.minBet/2);

AddMoneyToTable(TableBehaviour.tb.minBet/2);

EndPlay();

return;

}

else

{

DecreaseMoney(TableBehaviour.tb.minBet);

AddMoneyToTable(TableBehaviour.tb.minBet);

}

EndPlay();

}

public void Raise()

{

DecreaseMoney(TableBehaviour.tb.minBet\*2);

AddMoneyToTable(TableBehaviour.tb.minBet\*2);

EndPlay();

}

public void Call()

{

float difference;

difference = TableBehaviour.tb.players[TableBehaviour.tb.ReturnPreviousPlayer()].currentBet - currentBet;

DecreaseMoney(difference);

AddMoneyToTable(difference);

EndPlay();

}

public void Check()

{

EndPlay();

}

public void PlayerFold()

{

Fold();

EndPlay();

}

}

## Card.cs

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

public class Card

{

public enum Suits { H = 1, S = 4, C = 2, D = 3, NA = 0 };

public Suits suit;

public int value;

public Card(Suits Suit, int Value)

{

suit = Suit;

value = Value;

}

}

## CardHolder.cs

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

public class CardHolder : MonoBehaviour

{

public Card card1;

public Transform card1Position;

public Card card2;

public Transform card2Position;

public void ClearHand()

{

card1 = null;

card2 = null;

}

}

## CardPlayer.cs

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

using TMPro;

public class CardPlayer : MonoBehaviour {

public TextMeshProUGUI moneyText;

public TextMeshProUGUI roleText;

public TextMeshProUGUI betText;

public TableBehaviour.Role role = TableBehaviour.Role.NA;

public bool playerFolded = false;

[HideInInspector]

public CardHolder cardHolder;

[HideInInspector]

public Hand hand = new Hand(Hand.Hands.HighCard);

public float money;

public float currentBet;

[HideInInspector]

public bool boughtin = false;

[HideInInspector]

public bool canPlay = false;

protected int previousPlayer = 0;

public void resetToPlay()

{

boughtin = false;

currentBet = 0f;

playerFolded = false;

betText.text = "";

role = TableBehaviour.Role.NA;

roleText.text = "";

cardHolder.ClearHand();

}

public void Play()

{

if (TableBehaviour.tb.IsOnePlayerLeft())

return;

canPlay = true;

Debug.Log(gameObject.name + "'s turn.");

previousPlayer = TableBehaviour.tb.ReturnPreviousPlayer();

}

public void DecreaseMoney(float value)

{

money -= value;

boughtin = true;

moneyText.text = money.ToString();

}

public void IncreaseMoney(float value)

{

money += value;

moneyText.text = money.ToString();

}

public void Fold()

{

playerFolded = true;

TableBehaviour.tb.RemovePlayerFromMatch(this);

}

public Hand myHand()

{

hand = CheckHand.checkHand.BestHand(cardHolder.card1, cardHolder.card2);

return hand;

}

}

## CheckHand.cs

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

public class CheckHand : MonoBehaviour

{

public static CheckHand checkHand;

Hand currentBestHand = new Hand(Hand.Hands.HighCard);

List<Card> cards = new List<Card>();

private int length;

public void Awake()

{

checkHand = this;

}

public int CardsInHandValue(List<Card> cards)

{

int total = 0;

for (int i = 0; i < cards.Count; i++)

{

total += cards[i].value + (int)cards[i].suit;

}

return total;

}

public Hand BestHand(Card firstCard, Card secondCard)

{

cards = new List<Card>();

cards.Add(firstCard);

cards.Add(secondCard);

if (TableBehaviour.tb.table.card1 != null)

cards.Add(TableBehaviour.tb.table.card1);

if (TableBehaviour.tb.table.card2 != null)

cards.Add(TableBehaviour.tb.table.card2);

if (TableBehaviour.tb.table.card3 != null)

cards.Add(TableBehaviour.tb.table.card3);

if (TableBehaviour.tb.table.card4 != null)

cards.Add(TableBehaviour.tb.table.card4);

if (TableBehaviour.tb.table.card5 != null)

cards.Add(TableBehaviour.tb.table.card5);

currentBestHand = new Hand(Hand.Hands.HighCard);

length = cards.Count - 1;

HighCard();

SinglePair();

TwoPair();

ThreeOfAKind();

Straight();

Flush();

FullHouse();

FourOfAKind();

StraightFlush();

RoyalStraightFlush();

return currentBestHand;

}

private void HighCard()

{

Card highCard = new Card(Card.Suits.NA, 0);

for (int i = 0; i < 2; i++)

{

if (cards[i].value > highCard.value)

highCard = cards[i];

}

currentBestHand.hands = Hand.Hands.HighCard;

currentBestHand.cards = new List<Card>();

currentBestHand.cards.Add(highCard);

}

private void SinglePair()

{

List<Card> pairs = new List<Card>();

int comparrisons = length;

for (int i = 0; i < length; i++)

{

for (int j = i + 1; j <= comparrisons; j++)

{

//Debug.Log("Comparing: " + cards[i].suit + cards[i].value + " to " + cards[j].suit + cards[j].value);

if (cards[i].value == cards[j].value)

{

pairs.Add(cards[i]);

pairs.Add(cards[j]);

}

}

}

if (pairs.Count == 2)

{

currentBestHand.hands = Hand.Hands.Pair;

currentBestHand.cards = new List<Card>();

currentBestHand.cards.Add(pairs[0]);

currentBestHand.cards.Add(pairs[1]);

}

}

private void TwoPair()

{

List<Card> pairs = new List<Card>();

int comparrisons = length;

for (int i = 0; i < length; i++)

{

for (int j = i + 1; j <= comparrisons; j++)

{

//Debug.Log("Comparing: " + cards[i].suit + cards[i].value + " to " + cards[j].suit + cards[j].value);

if (cards[i].value == cards[j].value)

{

pairs.Add(cards[i]);

pairs.Add(cards[j]);

}

}

}

if (pairs.Count >= 4)

{

Card temp1, temp2;

bool continueSort = true;

if (pairs.Count == 2)

{

continueSort = false;

}

while (continueSort)

{

for (int i = 0; i < pairs.Count - 2; i += 2)

{

for (int j = i + 2; j < pairs.Count - 2; j += 2)

{

if (pairs[i].value > pairs[j].value)

{

continueSort = true;

temp1 = pairs[i];

temp2 = pairs[i + 1];

pairs[i] = pairs[j];

pairs[i + 1] = pairs[j + 1];

pairs[j] = temp1;

pairs[j + 1] = temp2;

}

else

{

continueSort = false;

}

}

}

continueSort = false;

}

currentBestHand.hands = Hand.Hands.TwoPair;

currentBestHand.cards = new List<Card>();

currentBestHand.cards.Add(pairs[pairs.Count - 2]);

currentBestHand.cards.Add(pairs[pairs.Count - 1]);

currentBestHand.cards.Add(pairs[pairs.Count - 4]);

currentBestHand.cards.Add(pairs[pairs.Count - 3]);

}

}

private void ThreeOfAKind()

{

if (length < 3)

return;

List<Card> threeOfAKinds = new List<Card>();

for (int i = 0; i < length; i++)

{

for (int j = i + 1; j < (length); j++)

{

for (int k = j + 1; k <= length; k++)

{

//Debug.Log("Comparing: " + cards[i].suit + cards[i].value + " to " + cards[j].suit + cards[j].value + " to " + cards[k].suit + cards[k].value);

if (cards[i].value == cards[j].value && cards[i].value == cards[k].value)

{

threeOfAKinds.Add(cards[i]);

threeOfAKinds.Add(cards[j]);

threeOfAKinds.Add(cards[k]);

}

}

}

}

if (threeOfAKinds.Count > 0)

{

currentBestHand.hands = Hand.Hands.ThreeOfAKind;

currentBestHand.cards = new List<Card>();

currentBestHand.cards.Add(threeOfAKinds[0]);

currentBestHand.cards.Add(threeOfAKinds[1]);

currentBestHand.cards.Add(threeOfAKinds[2]);

}

}

private void Straight()

{

if (length < 5)

return;

List<Card> orderedByValueCards = new List<Card>();

List<Card> foundStraights = new List<Card>();

for (int i = 0; i < cards.Count; i++)

{

orderedByValueCards.Add(cards[i]);

}

Card temp1;

bool continueSort = true;

while (continueSort)

{

for (int i = 0; i < orderedByValueCards.Count - 1; i++)

{

for (int j = i + 1; j < orderedByValueCards.Count; j++)

{

if (orderedByValueCards[i].value > orderedByValueCards[j].value)

{

continueSort = true;

temp1 = orderedByValueCards[i];

orderedByValueCards[i] = orderedByValueCards[j];

orderedByValueCards[j] = temp1;

}

else

{

continueSort = false;

}

}

}

}

for (int i = 0; i < (length - 3); i++)

{

bool isOneMore = true;

for (int k = i; k < i + 4; k++)

{

if (isOneMore)

{

if (orderedByValueCards[k + 1].value - orderedByValueCards[k].value != 1)

{

isOneMore = false;

}

}

}

if (isOneMore)

{

foundStraights.Add(orderedByValueCards[i]);

foundStraights.Add(orderedByValueCards[i + 1]);

foundStraights.Add(orderedByValueCards[i + 2]);

foundStraights.Add(orderedByValueCards[i + 3]);

foundStraights.Add(orderedByValueCards[i + 4]);

}

}

if (foundStraights.Count != 0)

{

if (foundStraights.Count > 1 && foundStraights.Count < 6)

{

currentBestHand.hands = Hand.Hands.Straight;

currentBestHand.cards = new List<Card>();

currentBestHand.cards.Add(foundStraights[0]);

currentBestHand.cards.Add(foundStraights[1]);

currentBestHand.cards.Add(foundStraights[2]);

currentBestHand.cards.Add(foundStraights[3]);

currentBestHand.cards.Add(foundStraights[4]);

}

else

{

int indexForBestStraight = 0;

for (int i = 0; i < foundStraights.Count; i += 5)

{

if (foundStraights[i].value > foundStraights[indexForBestStraight].value)

indexForBestStraight = i;

}

currentBestHand.hands = Hand.Hands.Straight;

currentBestHand.cards = new List<Card>();

currentBestHand.cards.Add(foundStraights[indexForBestStraight]);

currentBestHand.cards.Add(foundStraights[indexForBestStraight + 1]);

currentBestHand.cards.Add(foundStraights[indexForBestStraight + 2]);

currentBestHand.cards.Add(foundStraights[indexForBestStraight + 3]);

currentBestHand.cards.Add(foundStraights[indexForBestStraight + 4]);

}

}

}

private void Flush()

{

if (length < 5)

return;

List<Card> orderedByValueCards = new List<Card>();

List<Card> heartCards = new List<Card>();

List<Card> clubCards = new List<Card>();

List<Card> diamondCards = new List<Card>();

List<Card> spadeCards = new List<Card>();

for (int i = 0; i < cards.Count; i++)

{

orderedByValueCards.Add(cards[i]);

}

Card temp1;

bool continueSort = true;

while (continueSort)

{

for (int i = 0; i < orderedByValueCards.Count - 1; i++)

{

for (int j = i + 1; j < orderedByValueCards.Count; j++)

{

if (orderedByValueCards[i].value > orderedByValueCards[j].value)

{

continueSort = true;

temp1 = orderedByValueCards[i];

orderedByValueCards[i] = orderedByValueCards[j];

orderedByValueCards[j] = temp1;

}

else

{

continueSort = false;

}

}

}

}

for (int i = 0; i < orderedByValueCards.Count; i++)

{

switch (orderedByValueCards[i].suit)

{

case Card.Suits.H:

heartCards.Add(orderedByValueCards[i]);

break;

case Card.Suits.C:

clubCards.Add(orderedByValueCards[i]);

break;

case Card.Suits.D:

diamondCards.Add(orderedByValueCards[i]);

break;

case Card.Suits.S:

spadeCards.Add(orderedByValueCards[i]);

break;

}

}

if (heartCards.Count >= 5)

{

currentBestHand.hands = Hand.Hands.Flush;

currentBestHand.cards = new List<Card>();

currentBestHand.cards.Add(heartCards[heartCards.Count - 1]);

currentBestHand.cards.Add(heartCards[heartCards.Count - 2]);

currentBestHand.cards.Add(heartCards[heartCards.Count - 3]);

currentBestHand.cards.Add(heartCards[heartCards.Count - 4]);

currentBestHand.cards.Add(heartCards[heartCards.Count - 5]);

}

else if (clubCards.Count >= 5)

{

currentBestHand.hands = Hand.Hands.Flush;

currentBestHand.cards = new List<Card>();

currentBestHand.cards.Add(clubCards[clubCards.Count - 1]);

currentBestHand.cards.Add(clubCards[clubCards.Count - 2]);

currentBestHand.cards.Add(clubCards[clubCards.Count - 3]);

currentBestHand.cards.Add(clubCards[clubCards.Count - 4]);

currentBestHand.cards.Add(clubCards[clubCards.Count - 5]);

}

else if (diamondCards.Count >= 5)

{

currentBestHand.hands = Hand.Hands.Flush;

currentBestHand.cards = new List<Card>();

currentBestHand.cards.Add(diamondCards[diamondCards.Count - 1]);

currentBestHand.cards.Add(diamondCards[diamondCards.Count - 2]);

currentBestHand.cards.Add(diamondCards[diamondCards.Count - 3]);

currentBestHand.cards.Add(diamondCards[diamondCards.Count - 4]);

currentBestHand.cards.Add(diamondCards[diamondCards.Count - 5]);

}

else if (spadeCards.Count >= 5)

{

currentBestHand.hands = Hand.Hands.Flush;

currentBestHand.cards = new List<Card>();

currentBestHand.cards.Add(spadeCards[spadeCards.Count - 1]);

currentBestHand.cards.Add(spadeCards[spadeCards.Count - 2]);

currentBestHand.cards.Add(spadeCards[spadeCards.Count - 3]);

currentBestHand.cards.Add(spadeCards[spadeCards.Count - 4]);

currentBestHand.cards.Add(spadeCards[spadeCards.Count - 5]);

}

}

private void FullHouse()

{

if (length < 5)

return;

List<Card> threeOfAKinds = new List<Card>();

List<Card> cardsToCheck = new List<Card>();

for (int i = 0; i < cards.Count; i++)

{

cardsToCheck.Add(cards[i]);

}

for (int i = 0; i < length; i++)

{

for (int j = i + 1; j < (length); j++)

{

for (int k = j + 1; k <= length; k++)

{

if (cardsToCheck[i].value == cardsToCheck[j].value && cardsToCheck[i].value == cardsToCheck[k].value)

{

threeOfAKinds.Add(cardsToCheck[i]);

threeOfAKinds.Add(cardsToCheck[j]);

threeOfAKinds.Add(cardsToCheck[k]);

}

}

}

}

if (threeOfAKinds.Count > 0)

{

cardsToCheck.Remove(threeOfAKinds[0]);

cardsToCheck.Remove(threeOfAKinds[1]);

cardsToCheck.Remove(threeOfAKinds[2]);

List<Card> pairs = new List<Card>();

int comparrisons = cardsToCheck.Count;

for (int i = 0; i < cardsToCheck.Count; i++)

{

for (int j = i + 1; j < comparrisons; j++)

{

if (cardsToCheck[i].value == cardsToCheck[j].value)

{

pairs.Add(cardsToCheck[i]);

pairs.Add(cardsToCheck[j]);

}

}

}

if (pairs.Count == 2)

{

currentBestHand.hands = Hand.Hands.FullHouse;

currentBestHand.cards = new List<Card>();

currentBestHand.cards.Add(threeOfAKinds[0]);

currentBestHand.cards.Add(threeOfAKinds[1]);

currentBestHand.cards.Add(threeOfAKinds[2]);

currentBestHand.cards.Add(pairs[0]);

currentBestHand.cards.Add(pairs[1]);

}

else if (pairs.Count > 4)

{

Card temp1, temp2;

bool continueSort = true;

while (continueSort)

{

for (int i = 0; i < pairs.Count - 2; i += 2)

{

for (int j = i + 2; j < pairs.Count - 2; j += 2)

{

if (pairs[i].value > pairs[j].value)

{

continueSort = true;

temp1 = pairs[i];

temp2 = pairs[i + 1];

pairs[i] = pairs[j];

pairs[i + 1] = pairs[j + 1];

pairs[j] = temp1;

pairs[j + 1] = temp2;

}

else

{

continueSort = false;

}

}

}

}

currentBestHand.hands = Hand.Hands.FullHouse;

currentBestHand.cards = new List<Card>();

currentBestHand.cards.Add(threeOfAKinds[0]);

currentBestHand.cards.Add(threeOfAKinds[1]);

currentBestHand.cards.Add(threeOfAKinds[2]);

currentBestHand.cards.Add(pairs[0]);

currentBestHand.cards.Add(pairs[1]);

}

}

}

private void FourOfAKind()

{

if (length < 4)

return;

List<Card> fourOfAKinds = new List<Card>();

for (int i = 0; i < length; i++)

{

for (int j = i + 1; j < length; j++)

{

for (int k = j + 1; k < length; k++)

{

for (int o = k + 1; o <= length; o++)

{

if (cards[i].value == cards[j].value && cards[i].value == cards[k].value && cards[k].value == cards[o].value)

{

fourOfAKinds.Add(cards[i]);

fourOfAKinds.Add(cards[j]);

fourOfAKinds.Add(cards[k]);

fourOfAKinds.Add(cards[o]);

}

}

}

}

}

if (fourOfAKinds.Count > 0)

{

currentBestHand.hands = Hand.Hands.FourOfAKind;

currentBestHand.cards = new List<Card>();

currentBestHand.cards.Add(fourOfAKinds[0]);

currentBestHand.cards.Add(fourOfAKinds[1]);

currentBestHand.cards.Add(fourOfAKinds[2]);

currentBestHand.cards.Add(fourOfAKinds[3]);

}

}

private void StraightFlush()

{

if (length < 5)

return;

List<Card> orderedByValueCards = new List<Card>();

List<Card> foundStraights = new List<Card>();

for (int i = 0; i < cards.Count; i++)

{

orderedByValueCards.Add(cards[i]);

}

Card temp1;

bool continueSort = true;

while (continueSort)

{

for (int i = 0; i < orderedByValueCards.Count - 1; i++)

{

for (int j = i + 1; j < orderedByValueCards.Count; j++)

{

if (orderedByValueCards[i].value > orderedByValueCards[j].value)

{

continueSort = true;

temp1 = orderedByValueCards[i];

orderedByValueCards[i] = orderedByValueCards[j];

orderedByValueCards[j] = temp1;

}

else

{

continueSort = false;

}

}

}

}

for (int i = 0; i < (length - 3); i++)

{

bool isOneMore = true;

for (int k = i; k < i + 4; k++)

{

if (isOneMore)

{

if (orderedByValueCards[k + 1].value - orderedByValueCards[k].value != 1)

{

isOneMore = false;

}

}

}

if (isOneMore)

{

foundStraights.Add(orderedByValueCards[i]);

foundStraights.Add(orderedByValueCards[i + 1]);

foundStraights.Add(orderedByValueCards[i + 2]);

foundStraights.Add(orderedByValueCards[i + 3]);

foundStraights.Add(orderedByValueCards[i + 4]);

}

}

if (foundStraights.Count != 0)

{

List<Card> foundStraightFlush = new List<Card>();

for (int i = 0; i < foundStraights.Count; i += 5)

{

bool isSameSuit = true;

for (int k = i; k < i + 5; k++)

{

if (foundStraights[i].suit != foundStraights[k].suit)

isSameSuit = false;

}

if (isSameSuit)

{

for (int k = i; k < i + 5; k++)

{

foundStraightFlush.Add(foundStraights[k]);

}

}

}

if (foundStraightFlush.Count >= 1)

{

currentBestHand.hands = Hand.Hands.StraightFlush;

currentBestHand.cards = new List<Card>();

currentBestHand.cards.Add(foundStraightFlush[0]);

currentBestHand.cards.Add(foundStraightFlush[1]);

currentBestHand.cards.Add(foundStraightFlush[2]);

currentBestHand.cards.Add(foundStraightFlush[3]);

currentBestHand.cards.Add(foundStraightFlush[4]);

}

}

}

private void RoyalStraightFlush()

{

if (length < 5)

return;

List<Card> orderedByValueCards = new List<Card>();

List<Card> foundStraights = new List<Card>();

for (int i = 0; i < cards.Count; i++)

{

orderedByValueCards.Add(cards[i]);

}

Card temp1;

bool continueSort = true;

while (continueSort)

{

for (int i = 0; i < orderedByValueCards.Count - 1; i++)

{

for (int j = i + 1; j < orderedByValueCards.Count; j++)

{

if (orderedByValueCards[i].value > orderedByValueCards[j].value)

{

continueSort = true;

temp1 = orderedByValueCards[i];

orderedByValueCards[i] = orderedByValueCards[j];

orderedByValueCards[j] = temp1;

}

else

{

continueSort = false;

}

}

}

}

for (int i = 0; i < (length - 3); i++)

{

bool isOneMore = true;

for (int k = i; k < i + 4; k++)

{

if (isOneMore)

{

if (orderedByValueCards[k + 1].value - orderedByValueCards[k].value != 1)

{

isOneMore = false;

}

}

}

if (isOneMore)

{

foundStraights.Add(orderedByValueCards[i]);

foundStraights.Add(orderedByValueCards[i + 1]);

foundStraights.Add(orderedByValueCards[i + 2]);

foundStraights.Add(orderedByValueCards[i + 3]);

foundStraights.Add(orderedByValueCards[i + 4]);

}

}

if (foundStraights.Count != 0)

{

List<Card> foundStraightFlush = new List<Card>();

for (int i = 0; i < foundStraights.Count; i += 5)

{

bool isSameSuit = true;

for (int k = i; k < i + 5; k++)

{

if (foundStraights[i].suit != foundStraights[k].suit)

isSameSuit = false;

}

if (isSameSuit)

{

for (int k = i; k < i + 5; k++)

{

foundStraightFlush.Add(foundStraights[k]);

}

}

}

if (foundStraightFlush.Count >= 1 && foundStraightFlush[0].value == 10)

{

currentBestHand.hands = Hand.Hands.StraightFlush;

currentBestHand.cards = new List<Card>();

currentBestHand.cards.Add(foundStraightFlush[0]);

currentBestHand.cards.Add(foundStraightFlush[1]);

currentBestHand.cards.Add(foundStraightFlush[2]);

currentBestHand.cards.Add(foundStraightFlush[3]);

currentBestHand.cards.Add(foundStraightFlush[4]);

}

}

}

}

## Hand.cs

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

public class Hand

{

public enum Hands { HighCard = 1, Pair = 2, TwoPair = 3, ThreeOfAKind = 4, Straight = 5, Flush = 6, FullHouse = 7, FourOfAKind = 8, StraightFlush = 9, RoyalFlush = 10 };

public Hands hands;

public List<Card> cards;

public Hand(Hands handType)

{

hands = handType;

}

}

## AIBehaviour.cs

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

using TMPro;

public class AIBehaviour : CardPlayer {

public enum AIStates { WaitForTurn, EndTurn, Play, Check, Call, Bet, Raise, Fold}

[HideInInspector]

public FSM<AIStates> AIFSM;

[HideInInspector]

public bool check = false;

[HideInInspector]

public bool call = false;

[HideInInspector]

public bool bet = false;

[HideInInspector]

public bool raise = false;

[HideInInspector]

public bool fold = false;

[Space]

[Header("UTILITY AI VALUES")]

public float checkValue = 6f;

[Space]

public float callValue = 6f;

[Space]

public float betValue = 2f;

[Space]

public float raiseValue = 2f;

[Space]

public float foldValue = 5f;

private void Awake()

{

cardHolder = GetComponent<CardHolder>();

hand = new Hand(Hand.Hands.HighCard);

canPlay = false;

}

void Start () {

money = TableBehaviour.tb.startMoney;

moneyText.text = money.ToString();

InitialiseFSM();

}

private void Update()

{

AIFSM.CurrentState.Act();

AIFSM.Check();

}

private void InitialiseFSM()

{

AIFSM = new FSM<AIStates>();

AIFSM.AddState(new WaitForTurn<AIStates>(AIStates.WaitForTurn, this, 0f));

AIFSM.AddState(new EndTurn<AIStates>(AIStates.EndTurn, this, 2f));

AIFSM.AddState(new Play<AIStates>(AIStates.Play, this, 1f));

AIFSM.AddState(new Check<AIStates>(AIStates.Check, this, 1f));

AIFSM.AddState(new Call<AIStates>(AIStates.Call, this, 0f));

AIFSM.AddState(new Bet<AIStates>(AIStates.Bet, this, 0f));

AIFSM.AddState(new Raise<AIStates>(AIStates.Raise, this, 0f));

AIFSM.AddState(new Fold<AIStates>(AIStates.Fold, this, 0f));

AIFSM.SetInitialState(AIStates.WaitForTurn);

AIFSM.AddTransition(AIStates.WaitForTurn, AIStates.Play);

AIFSM.AddTransition(AIStates.EndTurn, AIStates.WaitForTurn);

AIFSM.AddTransition(AIStates.Play, AIStates.Bet);

AIFSM.AddTransition(AIStates.Bet, AIStates.EndTurn);

AIFSM.AddTransition(AIStates.Play, AIStates.Check);

AIFSM.AddTransition(AIStates.Check, AIStates.EndTurn);

AIFSM.AddTransition(AIStates.Play, AIStates.Call);

AIFSM.AddTransition(AIStates.Call, AIStates.EndTurn);

AIFSM.AddTransition(AIStates.Play, AIStates.Raise);

AIFSM.AddTransition(AIStates.Raise, AIStates.EndTurn);

AIFSM.AddTransition(AIStates.Play, AIStates.Fold);

AIFSM.AddTransition(AIStates.Fold, AIStates.EndTurn);

}

public bool GuardEndTurnToWaitForTurn(State<AIStates> currentState)

{

return currentState.StateFinished;

}

public bool GuardWaitForTurnToPlay(State<AIStates> currentState)

{

return canPlay;

}

public bool GuardPlayToBet(State<AIStates> currentState)

{

return bet;

}

public bool GuardBetToEndTurn(State<AIStates> currentState)

{

return currentState.StateFinished;

}

public bool GuardPlayToCheck(State<AIStates> currentState)

{

return check;

}

public bool GuardCheckToEndTurn(State<AIStates> currentState)

{

return currentState.StateFinished;

}

public bool GuardPlayToCall(State<AIStates> currentState)

{

return call;

}

public bool GuardCallToEndTurn(State<AIStates> currentState)

{

return currentState.StateFinished;

}

public bool GuardPlayToRaise(State<AIStates> currentState)

{

return raise;

}

public bool GuardRaiseToEndTurn(State<AIStates> currentState)

{

return currentState.StateFinished;

}

public bool GuardPlayToFold(State<AIStates> currentState)

{

if (fold)

Fold();

return fold;

}

public bool GuardFoldToEndTurn(State<AIStates> currentState)

{

return currentState.StateFinished;

}

public void StopPlay()

{

canPlay = false;

}

}

## Bet.cs

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

public class Bet<T> : State<T> {

private AIBehaviour AIBehaviour;

public Bet(T stateName, AIBehaviour controller, float minDuration) : base(stateName, controller, minDuration)

{

AIBehaviour = controller;

}

public override void OnEnter()

{

base.OnEnter();

AIBehaviour.bet = false;

if (!AIBehaviour.boughtin)

{

FirstRoundBet();

}

else

{

BetMoney(TableBehaviour.tb.minBet);

stateFinished = true;

}

}

private void BetMoney(float value)

{

AIBehaviour.currentBet += value;

AIBehaviour.betText.text = AIBehaviour.currentBet.ToString();

AIBehaviour.DecreaseMoney(value);

TableBehaviour.tb.AddToPot(value);

}

private void FirstRoundBet()

{

if (AIBehaviour.role == TableBehaviour.Role.SB)

{

BetMoney(TableBehaviour.tb.minBet / 2);

}

else

{

BetMoney(TableBehaviour.tb.minBet);

}

stateFinished = true;

}

}

## Call.cs

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

public class Call<T> : State<T>

{

private AIBehaviour AIBehaviour;

public Call(T stateName, AIBehaviour controller, float minDuration) : base(stateName, controller, minDuration)

{

AIBehaviour = controller;

}

public override void OnEnter()

{

base.OnEnter();

AIBehaviour.call = false;

float difference = 0f;

difference = TableBehaviour.tb.players[TableBehaviour.tb.ReturnPreviousPlayer()].currentBet - AIBehaviour.currentBet;

AIBehaviour.currentBet += difference;

AIBehaviour.betText.text = AIBehaviour.currentBet.ToString();

AIBehaviour.DecreaseMoney(difference);

TableBehaviour.tb.AddToPot(difference);

stateFinished = true;

}

}

## Check.cs

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

public class Check<T> : State<T>

{

private AIBehaviour AIBehaviour;

public Check(T stateName, AIBehaviour controller, float minDuration) : base(stateName, controller, minDuration)

{

AIBehaviour = controller;

}

public override void OnEnter()

{

base.OnEnter();

AIBehaviour.check = false;

stateFinished = true;

}

}

## EndTurn.cs

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

public class EndTurn<T> : State<T>

{

private AIBehaviour AIBehaviour;

public EndTurn(T stateName, AIBehaviour controller, float minDuration) : base(stateName, controller, minDuration)

{

AIBehaviour = controller;

}

public override void OnEnter()

{

base.OnEnter();

AIBehaviour.StopPlay();

TableBehaviour.tb.turnFinished = true;

stateFinished = true;

}

}

## Fold.cs

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

public class Fold<T> : State<T>

{

private AIBehaviour AIBehaviour;

public Fold(T stateName, AIBehaviour controller, float minDuration) : base(stateName, controller, minDuration)

{

AIBehaviour = controller;

}

public override void OnEnter()

{

base.OnEnter();

stateFinished = true;

}

}

## Play.cs

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

public class Play<T> : State<T>

{

private AIBehaviour AIBehaviour;

private bool check = true;

private bool call = true;

private bool bet = true;

private bool raise = true;

private bool fold = true;

private float checkUtility = 0f;

private float callUtility = 0f;

private float betUtility = 0f;

private float raiseUtility = 0f;

private float foldUtility = 0f;

private float totalUtility = 0f;

private int optionsAvailable = 0;

public Play(T stateName, AIBehaviour controller, float minDuration) : base(stateName, controller, minDuration)

{

AIBehaviour = controller;

}

public override void OnEnter()

{

base.OnEnter();

Logic();

}

private void Options()

{

check = true;

call = true;

bet = true;

raise = true;

fold = true;

RemoveImpossibleOptions();

}

private void RemoveImpossibleOptions()

{

int previousPlayer = TableBehaviour.tb.ReturnPreviousPlayer();

if (TableBehaviour.tb.players[previousPlayer].currentBet == AIBehaviour.currentBet)

{

call = false;

raise = false;

}

else

{

check = false;

bet = false;

}

}

private float FuzzyHandValue()

{

float result = 0;

float x = (int)AIBehaviour.hand.hands;

float x1 = (int)Hand.Hands.RoyalFlush;

if (x <= 0)

{

result = 0;

}

else

{

result = (((x / (x1 - (int)Hand.Hands.HighCard)) - ((int)Hand.Hands.HighCard / (x1 - (int)Hand.Hands.HighCard))));

}

return result;

}

private float FuzzyNot(float value)

{

return 1 - value;

}

private float FuzzyOR(float value1,float value2)

{

if (value1 > value2)

return value1;

else

return value2;

}

private void UtilityValues()

{

checkUtility = 0f;

callUtility = 0f;

betUtility = 0f;

raiseUtility = 0f;

foldUtility = 0f;

totalUtility = 0f;

optionsAvailable = 0;

if (check)

{

checkUtility = AIBehaviour.checkValue \* (int)AIBehaviour.hand.hands \* TableBehaviour.tb.roundCount;

optionsAvailable++;

totalUtility += checkUtility;

}

if (call)

{

callUtility = AIBehaviour.callValue \* (int)AIBehaviour.hand.hands \* TableBehaviour.tb.roundCount;

optionsAvailable++;

totalUtility += callUtility;

}

if (bet)

{

betUtility = AIBehaviour.betValue \* (int)AIBehaviour.hand.hands \* TableBehaviour.tb.roundCount;

optionsAvailable++;

totalUtility += betUtility;

}

if (raise)

{

raiseUtility = AIBehaviour.raiseValue \* (int)AIBehaviour.hand.hands \* TableBehaviour.tb.roundCount;

optionsAvailable++;

totalUtility += raiseUtility;

}

if (fold)

{

foldUtility = ((int)AIBehaviour.hand.hands \* -1) + (AIBehaviour.foldValue + FuzzyNot(FuzzyHandValue())/10) \* TableBehaviour.tb.roundCount;

optionsAvailable++;

totalUtility += foldUtility;

}

}

private void Logic()

{

if (!AIBehaviour.boughtin)

{

AIBehaviour.bet = true;

stateFinished = true;

}

AIBehaviour.hand = CheckHand.checkHand.BestHand(AIBehaviour.cardHolder.card1, AIBehaviour.cardHolder.card2);

Options();

UtilityValues();

while (true)

{

if (stateFinished)

return;

float desicion = Random.Range(0, totalUtility);

float min = 0f;

min += foldUtility;

if (desicion <= min && foldUtility != 0)

{

AIBehaviour.fold = true;

Debug.Log("Fold");

stateFinished = true;

return;

}

min += raiseUtility;

if (desicion <= min && raiseUtility != 0)

{

AIBehaviour.raise = true;

Debug.Log("Raise");

stateFinished = true;

return;

}

min += betUtility;

if (desicion <= min && betUtility != 0)

{

AIBehaviour.bet = true;

Debug.Log("Bet");

stateFinished = true;

return;

}

min += callUtility;

if (desicion <= min && callUtility != 0)

{

AIBehaviour.call = true;

Debug.Log("Call");

stateFinished = true;

return;

}

min += checkUtility;

if (desicion <= min && checkUtility != 0)

{

AIBehaviour.check = true;

Debug.Log("Check");

stateFinished = true;

return;

}

}

}

}

## Raise.cs

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

public class Raise<T> : State<T>

{

private AIBehaviour AIBehaviour;

public Raise(T stateName, AIBehaviour controller, float minDuration) : base(stateName, controller, minDuration)

{

AIBehaviour = controller;

}

public override void OnEnter()

{

base.OnEnter();

AIBehaviour.raise = false;

AIBehaviour.currentBet += TableBehaviour.tb.minBet \* 2;

AIBehaviour.betText.text = AIBehaviour.currentBet.ToString();

AIBehaviour.DecreaseMoney(TableBehaviour.tb.minBet \* 2);

TableBehaviour.tb.AddToPot(TableBehaviour.tb.minBet \* 2);

stateFinished = true;

}

}

## WaitForTurn.cs

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

public class WaitForTurn<T> : State<T>

{

private AIBehaviour AIBehaviour;

public WaitForTurn(T stateName, AIBehaviour controller, float minDuration) : base(stateName, controller, minDuration)

{

AIBehaviour = controller;

}

}