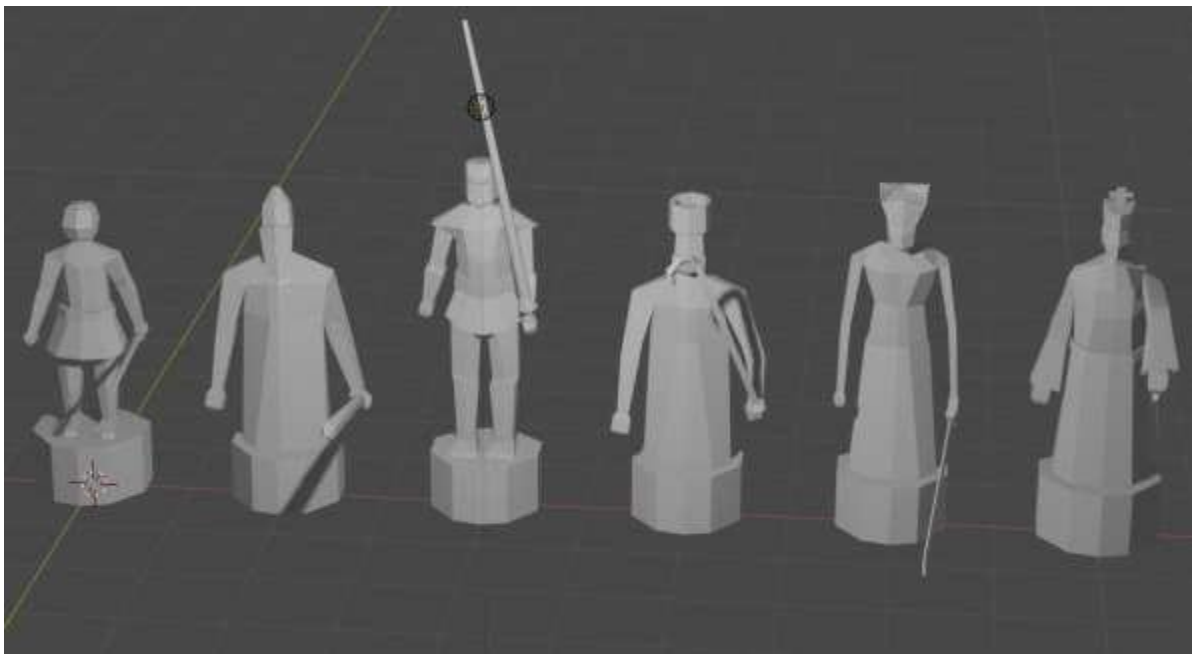
Wizard Chess

Group Details :
Kesav Sai Vikash Bollam – 002268090 –bvikash19@ubishops.ca
Ashwal Oza – 002271360 - aoza19@ubishops.ca

Content

1. Pieces
2. Game rules
3. AI
4. Game initialization and Finishing the Game
5. Main menu
6. Audio

1. Pieces



Pawn, bishop, knight, rook, queen, king (from left to right)

All the pieces are controlled by the same script Piece with a state machine. Each piece has animations and destructing effect.
The movements of the piece is controlled by a nav mesh agent. when user or ai selects a move the agents position will be set to that square. While moving the pieces in the way (for knight) will be deactivated and activated after completing the movement.
The destructing effect is created using the fractured mesh prefab. When a piece is destroyed the fractured version will be substituted. All the fractured pieces have rigid body mesh collider for realistic simulation.

```
6
7
8   public class Piece : MonoBehaviour {
9
10      // 1 = pawn
11      // 2 = bishop
12      // 3 = knight
13      // 4 = rook
14      // 5 = queen
15      // 6 = king
16      // negative for black positive for white
17      public int type;
18
19      public int[] square = new int[2]{0, 0};
20
21      public float speed = 4.0f;
22      public float rotateSpeed = 20.0f;
23
24      public static Board board;
25
26      public float attackTime = 1.15f;
27
28      public GameObject fractured;
29
30      private float timer = 0.0f;
31      private Quaternion targetRotation;
32
33      private GameObject deactivated = null;
34      private GameObject deactivated1 = null;
35
36      // 0 = idle
37      // 1 = moving
38      // 2 = moving to attack
39      // 3 = attacking
40      // 4 = rotating
41      // 5 = deactivated
42      public int state = 0;
43
44      // next state  after rataion
45      private int nextState = 0;
46
47      public bool knightMoving = false;
48
49      int[] target;
50      int[] target2;  // only for knight
51      int[] attack;
52
53      Animator animator;
54      NavMeshAgent agent;
55      AudioSource audioSource;
56
57      // Start is called before the first frame update
```

```
57      // Start is called before the first frame update
58      void Start()
59      {
60          animator = GetComponent<Animator>();
61          agent = GetComponent<NavMeshAgent>();
62          audioSource = GetComponent<AudioSource>();
63
64          agent.updateRotation = false;
65          gameObject.tag = "piece";
66      }
67
68      // Update is called once per frame
69      void Update()
70      {
71          agent.speed = speed;
72          if (state == 1 || state == 2)
73          {
74              HandlePieceCollision();
75
76              if (!audioSource.isPlaying)
77              {
78                  audioSource.Play();
79              }
80
81              if ((Board.SquareToPos( target ) - transform.position).magnitude <= 0.1f )
82              {
83                  square[0] = target[0];
84                  square[1] = target[1];
85                  animator.SetBool( "move", false );
86
87                  if ( state == 2)
88                  {
89                      state = 3;
90                  }
91                  else
92                  {
93                      board.busy = false;
94                      state = 0;
95
96                      // special case knight
97                      if ((type == 3 || type == -3) && knightMoving)
98                      {
99                          target = target2;
100                         state = 1;
101                         knightMoving = false;
102                         board.busy = true;
103                         RotateTowards( target , 1);
104                     }
105                     else
106                     {
107                         if (deactivated != null)
108                         {
109                             deactivated.SetActive( true );
110                             deactivated = null;
111                         }
112
```

2. The Game rules

The game rules include the moves of each pieces, check and checkmate..etc. All the possible moves are in the Move class. It has a function to get the all the possible moves for a given board (matrix).

```
41
42      // add all possible moves according to the board
43      // calculate for white or black input moves list must be empty
44      // return
45      //      > 0 no of possible moves posible moves
46      //      0 draw
47      //      -1 checkmate
48      //      -2 only 2 kings left
49      public static int GetPossibleMoves( int[,] boardMatrix, bool forWhite, ref List<Move> moves )
50      {
51          int count = 0;
52
53          for (int i = 0; i < 8; i++)
54          {
55              for (int j = 0; j < 8; j++)
56              {
57                  int pieceType = boardMatrix[i, j];
58
59                  if (pieceType != 0) count++;
60
61                  if ((forWhite && pieceType > 0) || ( !forWhite && pieceType < 0))
62                      GetPossibleMovesPiece( i, j, pieceType, boardMatrix, ref moves );
63              }
64          }
65
66          if (moves.Count == 0)
67          {
68              if (IsCheck( boardMatrix, forWhite ))
69                  return -1;
70
71              return 0;
72          }
73          if (count == 2)  // only 2 kings
74              return -2;
75
76          return moves.Count;
77      }
78
```

Using those possible moves the ai can find the next move. Also move class has function to get possible move for each piece according to their position in the given board.

```
    // add the possible moves for a bishop at i, j for the boardMatrix
    private static void GetMovesBishop( int i, int j, int k, int[,] boardMatrix, ref List<Move> moves )
    {
        bool forWhite = k > 0;

        int i_ = i + 1;
        int j_ = j + 1;

        int [,] p = {{1, 1}, {1, -1}, {-1, 1}, {-1, -1}};

        // go through diagonal squars
        for (int x = 0; x < 4; x++)
        {
            i_ = i + p[x, 0];
            j_ = j + p[x, 1];

            while (true)
            {
                // edges of the board
                if (i_ < 0 || j_ < 0 || i_ > 7 || j_ > 7)
                    break;

                int temp = boardMatrix[i_, j_] * k;

                if (temp == 0)   // if empty
                    AddMove( new Move( i, j, i_, j_ ), boardMatrix, forWhite, ref moves );

                else    // not empty
                {
                    if (temp < 0 && temp != -6)   // for opponent piece
                        AddMove( new Move( i, j, i_, j_, MoveType.CAPTURE ), boardMatrix, forWhite, ref moves);

                    break;
                }

                i_ += p[x, 0];
                j_ += p[x, 1];
            }


        }
    }

```

If the king is check those possible moves will be limited to those which can get rid of the check.

3. The AI

The AI is a minimax algorithm with alpha-beta pruning. Using all the possible moves from the Move class it can identify the best move.

```
31      // tree traversing recursive
32      private Move MiniMax( ref int[,] board, bool isMax , int depth, float alpha, float beta )
33      {
34          List<Move> moves = new List<Move>();
35
36          int t = Move.GetPossibleMoves( board, isMax, ref moves );
37
38          if (t == 0)
39          {
40              if (depth == 0)
41                  this.board.drawn = true;
42
43              return null;
44          }
45          else if (t == -1)
46          {
47              if (depth == 0)
48              {
49                  this.board.CheckMate = true;
50              }
51              return null;
52          }
53
54
55          float minMax = 0.0f;
56          int k = 0;
57          float eval = 0.0f;
58
59          if (isMax) eval = -Mathf.Infinity;
60          else  eval = Mathf.Infinity;
61
62          for (int i = 0; i < moves.Count; i++)
63          {
64              int temp = Board.ApplyMove( ref board, moves[i] );
65
66              if (depth == treeHeight - 1)
67              {
68                  eval = Evaluator.Evaluate( board );
69              }
70              else
71              {
72                  Move bestMove = MiniMax( ref board, !isMax, depth + 1, alpha, beta );
73
74                  if (bestMove == null)
75                  {
76                      Board.ReverseMove( ref board, moves[i], temp );
77                      continue;
78                  }
79
80                  int temp1 = Board.ApplyMove( ref board, bestMove );
81
82                  eval = Evaluator.Evaluate( board );
83
84                  Board.ReverseMove( ref board, bestMove, temp1 );
85              }
86
```

```
106                 else
107                 {
108                     if ( minMax > eval)
109                     {
110                         minMax = eval;
111                         k = i;
112                     }
113 
114                     if (beta > eval)
115                         beta = eval;
116                 }
117 
118 
119 
120                 Board.ReverseMove( ref board, moves[i], temp );
121 
122                 if (beta <= alpha)
123                     break;
124             }
125 
126 
127             if (t == -2)
128             {
129                 if (depth == 0)
130                     this.drawMove = moves[k];
131             }
132 
133             return moves[k];
134         }
135 
136 
137     }
138 
```

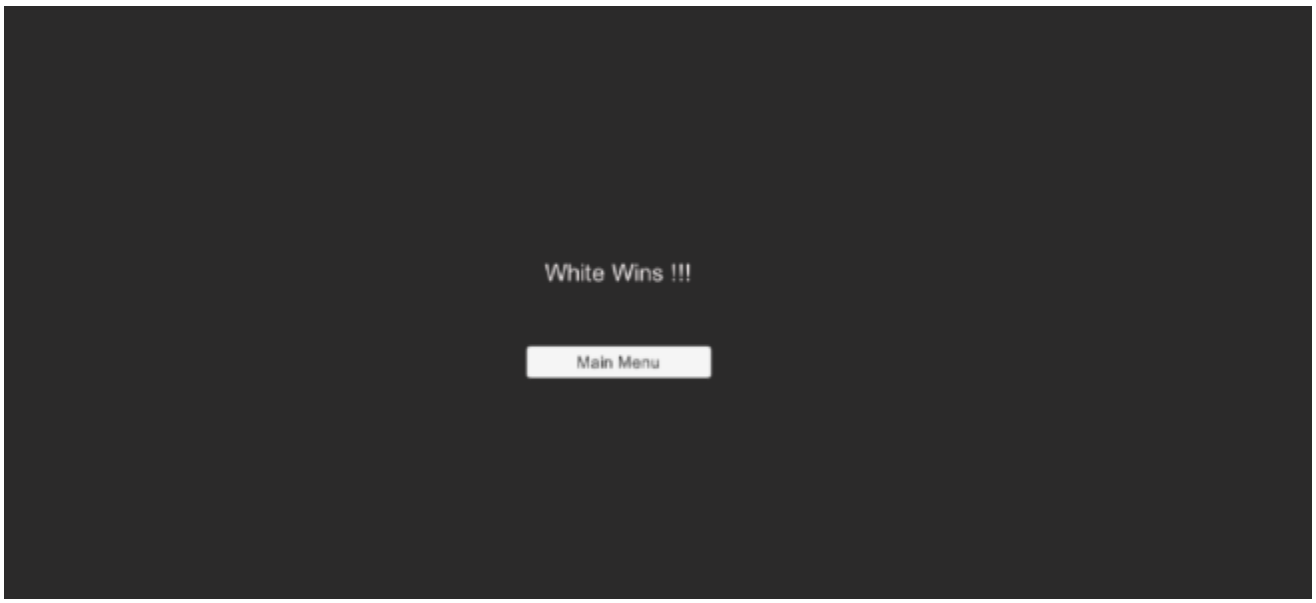
4. Game Initialization and Finishing the Game

When initializing a game the Board class has to consider type of the game. The type of the game is determining in the main menu by the user. The game types are human player vs minimax AI, human player vs random player, minimax AI vs minimax AI.

```
167     public void InitGame()
168     {
169         switch (gameType)
170         {
171             case 0:
172                 player1 = new HumanPlayer( this );
173                 player2 = new MiniMaxAI( this );
174                 speedFactor = 1.0f;
175                 break;
176 
177             case 1:
178                 player1 = new HumanPlayer( this );
179                 player2 = new RandomAI( this );
180                 speedFactor = 1.0f;
181                 break;
182 
183             case 2:
184                 player1 = new MiniMaxAI( this, true );
185                 player2 = new MiniMaxAI( this );
186                 speedFactor = 1.5f;
187                 break;
188 
189             default:
190                 break;
191         }
192 
```

Every game has two Player objects. player1 and player2. Player abstract class is the base class for MinimaxAI, RandomAI and HumanPlayer. In each turn the players has to set their next move.

The game can be finished by checkmate or draw. after finishing another scene called Finish will be loaded.


White Wins !!!
Main Menu

5. Main Menu

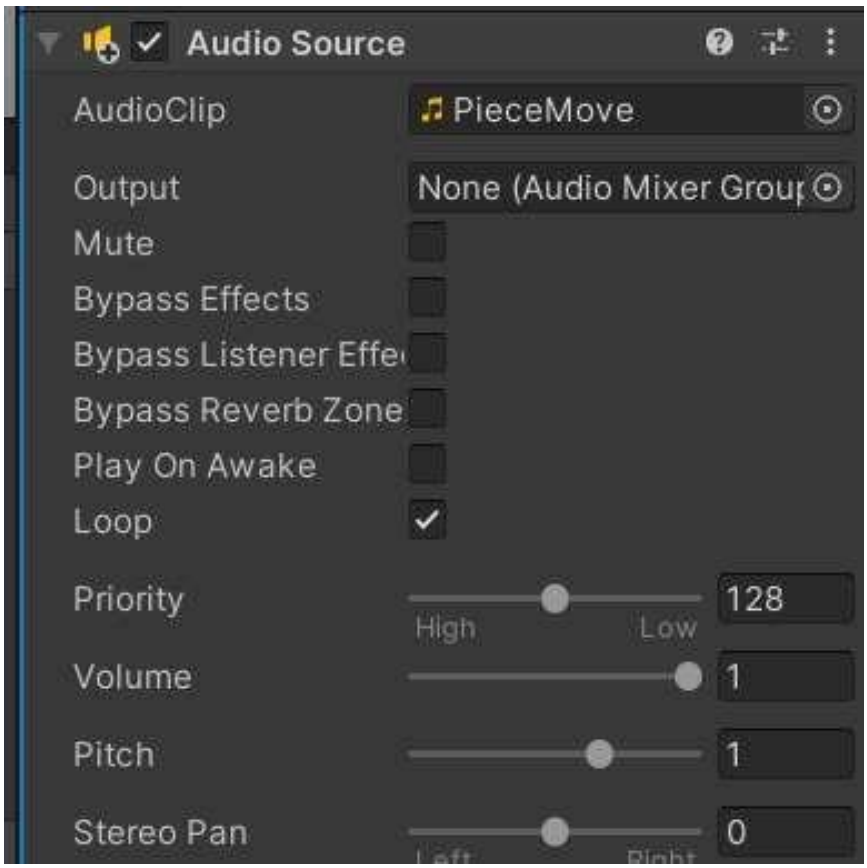
In the main menu the user has options to select to play against a random player, AI or watch a game between two AIs.

The main menu is controlled by the Menu.cs script.

```
7
8   public class Menu : MonoBehaviour
9   {
10      public Button startButton;
11      public Button aiVsAiButton;
12      public Button quitButton;
13
14      public void AiVsAi()
15      {
16          Board.gameType = 2;
17          SceneManager.LoadScene( "Chess" );
18      }
19
20      public void StartEasy()
21      {
22          Board.gameType = 1;
23          SceneManager.LoadScene( "Chess" );
24      }
25
26      public void StartHard()
27      {
28          Board.gameType = 0;
29          SceneManager.LoadScene( "Chess" );
30      }
31
32      public void Quit()
33      {
34          Application.Quit();
35      }
36  }
37
```

6. Audio

Audio clips are used for piece destroy, touching each other and moving.




```
70      {
71          agent.speed = speed;
72          if (state == 1 || state == 2)
73          {
74              HandlePieceCollision();
75
76              if (!audioSource.isPlaying)
77              {
78                  audioSource.Play();
79              }
80
```

And it is controlled by the Piece.cs script.