CS 512 Assignment 3

Wizard chess is a game from the popular 'Harry Potter' book series, which follows all the regular rules of chess. The main difference between Wizard chess and regular chess is the pieces are animated.

Group Members:

`Yi Ren` (002269013) `Wentao Lu` (002276355)

Dependency

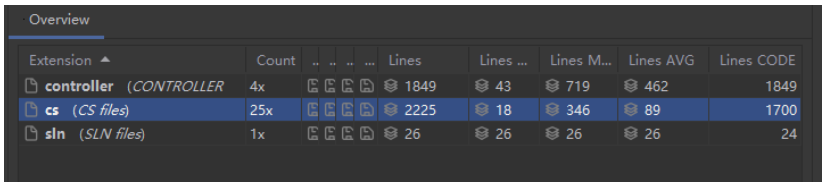
- Microsoft .NET Framework
- Unity3d
- Jetbrains Rider

Video demo

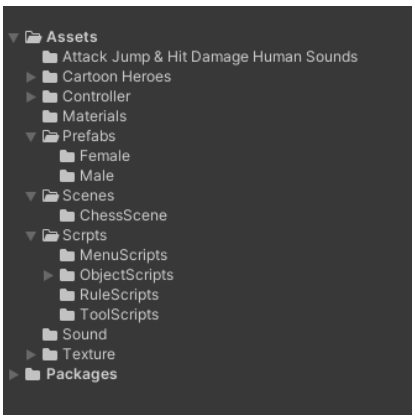
Please check Viedo Demo - Assignment3 for more information.

Project Information

- **Code info**



- **Script Structure**



|

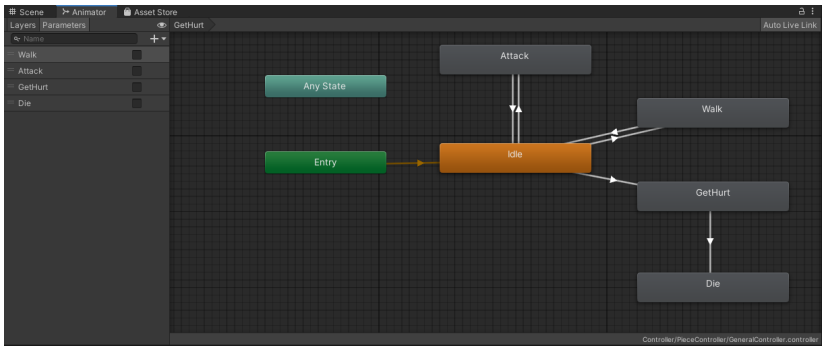
├─**MenuScripts**

├———— `MainMenu.cs` : implements some click listener for Main Menu Scene
│
├—**ObjectScripts**
├———— │ `Board.cs` :initialize 8*8 slices and pieces accordingly, handle states changes for each element on the board
├———— │ `Piece.cs` :base class for all kinds of pieces, initialize piece, handle click events and piece movements
├———— │ `PieceData.cs` : necessary parameters(data) for a piece
├———— │ `Slice.cs` : implements some basic functions for slice
├———— └—**Pieces** Scripts in this folder are implemented for certain pieces, which inherits `Piece.cs`
├———— `Bishop.cs`
├———— `King.cs`
├———— `Knight.cs`
├———— `Pawn.c`
├———— `Queen.c`
├———— `Rook.cs`
│
├—**RuleScripts**
├———— `AlphaBetaPruning.cs` :Alpha- Beta Pruning minimax
├———— `MiniMaxLoop.cs` : minimax with loop
├———— `MoveRules.cs` : move rules for all pieces, return available slice list
├———— `RandomRules.cs` : random movement for AI
│
├—**ToolScripts**
├———— `AudioPlayer.cs` :Audio manager
├———— `GeneralTools.cs` :General tools
├———— `InitConfig.cs` :Some constants and global parameters
├———— `ListChain.cs` : a list with chaining method
├———— `LogUtils.cs` :some log tools which encapsulates Debug Log functions
├———— `MouseDoubleClick.cs` :detect double click for both left and right mouse button
├———— `ObjectGenerater.cs` :generate some objects when needed
├———— `StringTools.cs`

Basic functions

Part 1 - Board + Animation + Navigation

- **a. Each piece must have a distinct model and animation set**
  In this part, we imported a character set which is called Medieval Cartoon Warriors from AssetStore. Also, we implemented `Animator Controller` for each character.

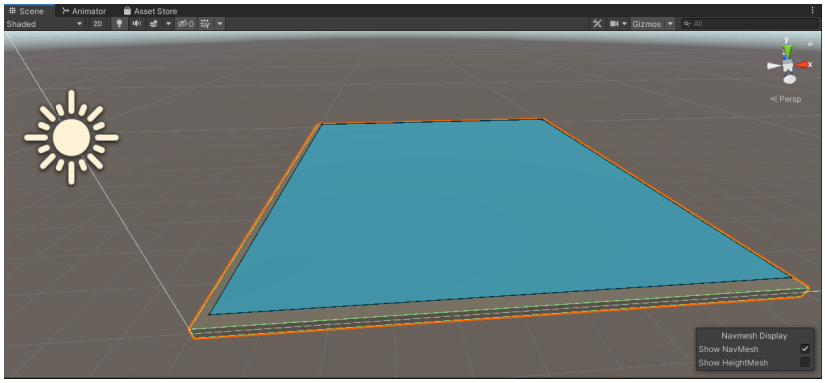


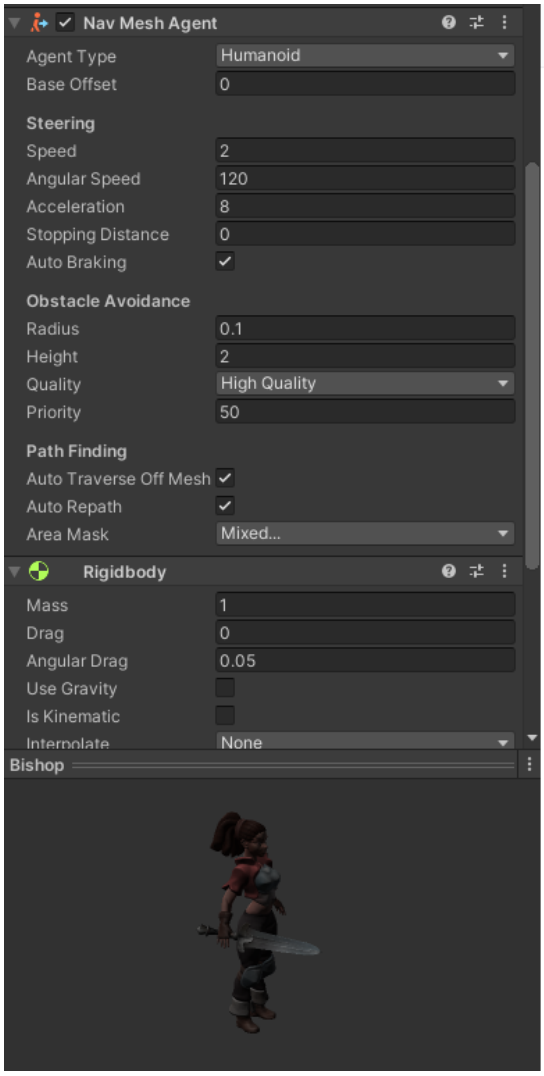
▶ Code

```
private void ChangeAnimationState(int state)
        {
            // var audioPlayer = new AudioPlayer();
            switch (state)
            {
                // Idle State
                case InitConfig.IDLE:
                    Animator.SetBool(Walk, false);
                    Animator.SetBool(Attack,false);
                    break;
                // Moving State
                case InitConfig.WALK:
                    Animator.SetBool(Walk, true);
                    Animator.SetBool(Attack,false);
                    break;
                // Attack State
                case InitConfig.ATTACK:
                    AudioPlayer.SharedInstance.PlayAttackAudio();
                    Animator.SetBool(Walk, false);
                    Animator.SetBool(Attack, true);
                    break;
                // Hurt State
                case InitConfig.GETATTACK:
                    AudioPlayer.SharedInstance.PlayGetAttackAudio();
                    Animator.SetBool(GetAttack, true);
                    break;
                // Destroy State
                case InitConfig.DIE:
                    AudioPlayer.SharedInstance.PlayDieAudio();
                    Animator.SetBool(Die, true);
                    break;
            }
        }
```

- **b. Pieces must move across board when selected and clicked using mouse**
  In this part, we detect the mouse click with callback function `OnMouseDown()` in each slice. When a certain piece is selected, we invoke `MouseClick()` in the parent class `Piece` to complete the following

steps(start animation, detect and slain enemy, change data, clear highlight slices, etc.) Then, we implemented movement system with `NavMesh` and `NavMeshAgent`.





▶ Code

Handle piece click

```
        protected void MouseClick(List<Vector2Int> markList)
        {
```

```
            // When animation is ongoing, pieces are not clickable
            if(!InitConfig.IsClickable) return;
            // Status = Status == InitConfig.STATE_NORMAL ?
InitConfig.STATE_SELECTED : InitConfig.STATE_NORMAL;

            // Case1: No piece selected
            if (!Board.SharedInstance.selectedPiece.Contains("Piece"))
            {
                if(isBlack != InitConfig.IsPlayerTurn) return;
                Status = InitConfig.STATE_SELECTED;
                Board.SharedInstance.selectedPiece = gameObject.name;
                Board.SharedInstance.ChangeSliceState(GetIndex(),
InitConfig.STATE_SELECTED);
                Board.SharedInstance.MarkAvailableSlice(markList);
            }
            // Piece selected
            else
            {
                if (Status == InitConfig.STATE_NORMAL)
                {
                    // Case2: Another Piece selected, then click on a normal
piece(not highlighted)
                    if (Board.SharedInstance.SliceList[GetIndex().x,
GetIndex().y].status == InitConfig.STATE_NORMAL)
                    {
                        Board.SharedInstance.ClearAllMarkSlice();
                    }
                    // Case3: Another Piece selected, then click on a highlighted
piece
                    if (Board.SharedInstance.SliceList[GetIndex().x,
GetIndex().y].status == InitConfig.STATE_HIGHLIGHT)
                    {
                        var p =
GameObject.Find(Board.SharedInstance.selectedPiece).GetComponent<Piece>();
                        Board.SharedInstance.ClearAllMarkSlice();
                        // Return if this is not opponent of the selected piece
                        if(p.isBlack == isBlack) return;
                        // Destroy this object
                        p.MoveToSlice(GetIndex());
                    }

                }
                // Case4: Piece selected, then click on the same piece
                if (Status == InitConfig.STATE_SELECTED)
                {
                    Status = InitConfig.STATE_NORMAL;
                    Board.SharedInstance.ClearAllMarkSlice();
                }
            }
        }
```

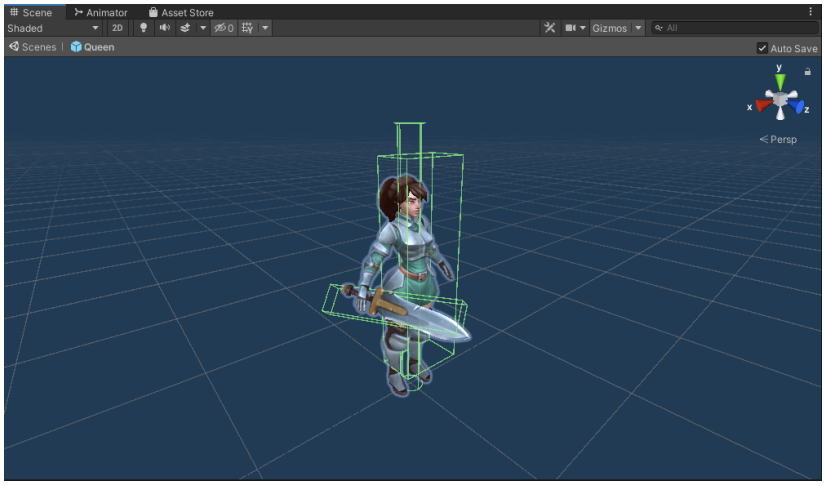
Handle piece movement

▶ Code

```
        public void MoveToSlice(Vector2Int toIndex)
        {
            Debug.Log("Move:" + gameObject.name + " to " + toIndex);
            InitConfig.IsPlayerTurn = !gameObject.name.Contains("Black");
            // If the pawn is moved for the first time, set isFirstStep to false
            if (gameObject.name.Contains("Pawn") &&
GameObject.Find(gameObject.name).GetComponent<Pawn>().isFirstStep) {
                GameObject.Find(gameObject.name).GetComponent<Pawn>().isFirstStep
= false;
            }

            // If an opponent stands on destination slice, remove that piece
            if (Board.SharedInstance.SliceList[toIndex.x, toIndex.y].pieceName !=
"")
            {
                _stopDistance = 1f;
                _hasEnemy = true;
                var piece =
GameObject.Find(Board.SharedInstance.SliceList[toIndex.x,
toIndex.y].pieceName).GetComponent<Piece>();
                piece.attackerName = gameObject.name;
                if (piece.isBlack) {
                    Board.SharedInstance.blackPieceList.Remove(piece);
                }else {
                    Board.SharedInstance.whitePieceList.Remove(piece);
                }
            } else {
                // Move to an empty slice
                _hasEnemy = false;
            }

            ChangeAnimationState(InitConfig.WALK);

            // Reset origin slice piece-name
            Board.SharedInstance.SliceList[Index.x, Index.y].pieceName = "";
            // Set destination slice piece-name
            Board.SharedInstance.SliceList[toIndex.x, toIndex.y].pieceName =
gameObject.name;
            Board.SharedInstance.EditRecord(gameObject.name, GetIndex(), toIndex);
            // Move to destination with Navmesh

            // _agent.stoppingDistance = _stopDistance;
            _agent.destination = Board.SharedInstance.SliceList[toIndex.x,
toIndex.y].transform.position;
            // _agent.stoppingDistance = _stopDistance;
            _agent.isStopped = false;
            isMoving = true;

            // transform.position = Board.SharedInstance.SliceList[toIndex.x,
toIndex.y].transform.position + new Vector3(0, transform.localScale.y * 0.5f,0);
            SetIndex(toIndex);
```

```
            Board.SharedInstance.ClearAllMarkSlice();

            if (InitConfig.DUAL_AI) {
                StartCoroutine(AutoPlay());
            }else if (!InitConfig.IsPlayerTurn)
            {
                StartCoroutine(AutoPlay());
            }
        }
```

- **c. Pieces must use physics when they hit other pieces (see lecture 11) to break them and send them flying**
  In this part, we add `rigidbody` and `collider` to each character and their weapons. Then we detect the collision between weapon and enemy with function `OnTriggerEnter()` in class `Piece`. When attacked, we add a backward force to the `rigidbody` of character and change its animation accordingly.

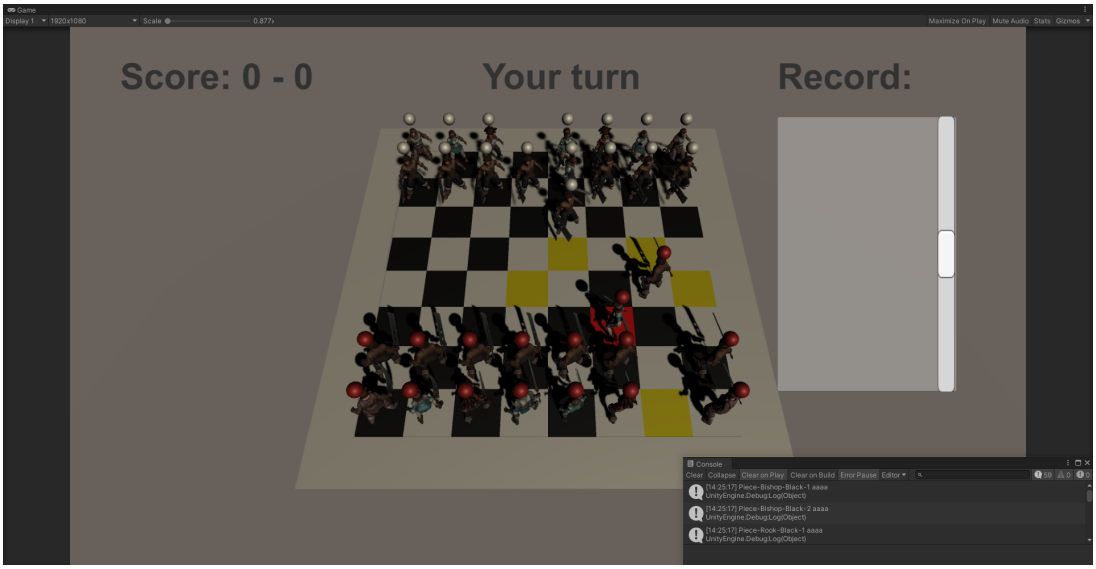


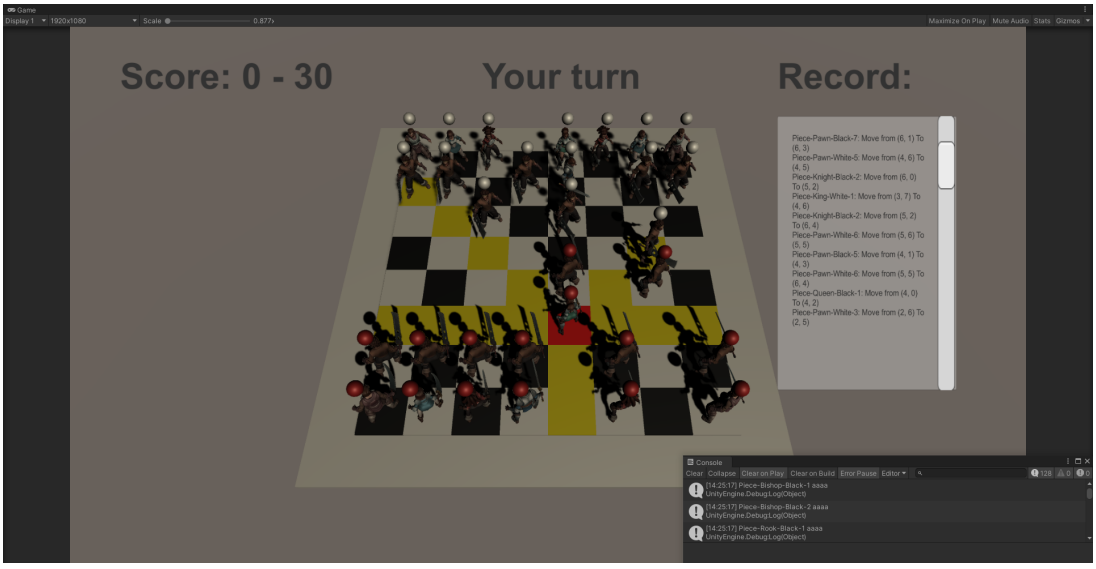
▶ Code

```
        private void OnTriggerEnter(Collider other)
        {
            // If enemy's weapon collide with this piece, get a backward force and
play animation[GetAttack], then animation[GetAttack]
            if (other.transform.root.name == attackerName)
            {
                ChangeAnimationState(InitConfig.GETATTACK);
                ChangeAnimationState(InitConfig.DIE);
                GetComponent<Rigidbody>().AddForce((transform.position -
other.transform.root.position) * 20, ForceMode.Impulse);
                StartCoroutine(DestroyPiece());
            }
        }
```

Part 2 - Chess Rules + Minimax

- **Piece Rules**
  In this part, to built rules for the chess game, we divided rules into variety classes, that is `Diagonal`, `Column`, `Row`, `Pawn` and `Knight`. For each rule, the corresponding function returns a Vector2 list, which contains all the available index for the piece under a certain rule.





Diagonal lines

▶ Code

```
        public List<Vector2Int> Diagonal(Vector2Int index, int stride, bool
isBlack)
        {
            Slice[,] sliceList = Board.SharedInstance.GetSliceList();
            var dList = new List<Vector2Int>();
            var tl = 1;
            var tr = 1;
            var bl = 1;
```

```
            var br = 1;
            while (index.x - tl >= Math.Max(0, index.x - stride) && index.y + tl
<= Math.Min(7, index.y + stride)) {
                if (!sliceList[index.x - tl, index.y +
tl].pieceName.Contains("Piece")) {
                    dList.Add(new Vector2Int(index.x - tl, index.y + tl));
                }else {
                    if (!StringTools.ComparePieceColor(sliceList[index.x - tl,
index.y + tl].pieceName, isBlack)) {
                        dList.Add(new Vector2Int(index.x - tl, index.y + tl));
                    }
                    break;
                }
                // dList.Add(new Vector2Int(index.x - tl, index.y + tl));
                // if (sliceList[index.x - tl, index.y +
tl].pieceName.Contains("Piece")) break;
                tl++;
            }
            while (index.x + tr <= Math.Min(7, index.x + stride) && index.y + tr
<= Math.Min(7, index.y + stride)) {
                if (!sliceList[index.x + tr, index.y +
tr].pieceName.Contains("Piece")) {
                    dList.Add(new Vector2Int(index.x + tr, index.y + tr));
                }else {
                    if (!StringTools.ComparePieceColor(sliceList[index.x + tr,
index.y + tr].pieceName, isBlack)) {
                        dList.Add(new Vector2Int(index.x + tr, index.y + tr));
                    }
                    break;
                }
                // dList.Add(new Vector2Int(index.x + tr, index.y + tr));
                // if (sliceList[index.x + tr, index.y +
tr].pieceName.Contains("Piece")) break;
                tr++;
            }
            while (index.x - bl >= Math.Max(0, index.x - stride) && index.y - bl
>= Math.Max(0, index.y - stride)) {
                if (!sliceList[index.x - bl, index.y -
bl].pieceName.Contains("Piece")) {
                    dList.Add(new Vector2Int(index.x - bl, index.y - bl));
                }else {
                    if (!StringTools.ComparePieceColor(sliceList[index.x - bl,
index.y - bl].pieceName, isBlack)) {
                        dList.Add(new Vector2Int(index.x - bl, index.y - bl));
                    }
                    break;
                }
                // dList.Add(new Vector2Int(index.x - bl, index.y - bl));
                // if (sliceList[index.x - bl, index.y -
bl].pieceName.Contains("Piece")) break;
                bl++;
            }
            while (index.x + br <= Math.Min(7, index.x + stride) && index.y - br
>= Math.Max(0, index.y - stride)) {
```

```
                if (!sliceList[index.x + br, index.y -
br].pieceName.Contains("Piece")) {
                    dList.Add(new Vector2Int(index.x + br, index.y - br));
                }else {
                    if (!StringTools.ComparePieceColor(sliceList[index.x + br,
index.y - br].pieceName, isBlack)) {
                        dList.Add(new Vector2Int(index.x + br, index.y - br));
                    }
                    break;
                }
                // dList.Add(new Vector2Int(index.x + br, index.y - br));
                // if (sliceList[index.x + br, index.y -
br].pieceName.Contains("Piece")) break;
                br++;
            }
            return dList;
        }
```

Row

▶ Code

```
        public List<Vector2Int> Horizontal(Vector2Int index, int stride, bool
isBlack)
        {
            var vList = new List<Vector2Int>();
            var r = index.x + 1;
            var l = index.x - 1;
            var sliceList = Board.SharedInstance.SliceList;
            while (l >= Math.Max(0, index.x - stride)) {
                if (!sliceList[l, index.y].pieceName.Contains("Piece")) {
                    vList.Add(new Vector2Int(l, index.y));
                }else {
                    if (!StringTools.ComparePieceColor(sliceList[l,
index.y].pieceName, isBlack)) {
                        vList.Add(new Vector2Int(l, index.y));
                    }
                    break;
                }
                // vList.Add(new Vector2Int(l, index.y));
                // if (sliceList[l, index.y].pieceName.Contains("Piece")) break;
                l--;
            }
            while (r <= Math.Min(7, index.x + stride)) {
                if (!sliceList[r, index.y].pieceName.Contains("Piece")) {
                    vList.Add(new Vector2Int(r, index.y));
                }else {
                    if (!StringTools.ComparePieceColor(sliceList[r,
index.y].pieceName, isBlack)) {
                        vList.Add(new Vector2Int(r, index.y));
                    }
```

```
                    break;
                }
                // vList.Add(new Vector2Int(r, index.y));
                // if (sliceList[r, index.y].pieceName.Contains("Piece")) break;
                r++;
            }
            return vList;
        }
```

Column

▶ Code

```
        public List<Vector2Int> Vertical(Vector2Int index, int stride, bool
isBlack)
        {
            var hList = new List<Vector2Int>();
            var t = index.y + 1;
            var b = index.y - 1;
            var sliceList = Board.SharedInstance.SliceList;
            while (b >= Math.Max(0, index.y - stride)) {
                if (!sliceList[index.x, b].pieceName.Contains("Piece")) {
                    hList.Add(new Vector2Int(index.x, b));
                }else {
                    if (!StringTools.ComparePieceColor(sliceList[index.x,
b].pieceName, isBlack)) {
                        hList.Add(new Vector2Int(index.x, b));
                    }
                    break;
                }
                b--;
            }
            while (t <= Math.Min(7, index.y + stride)) {
                if (!sliceList[index.x, t].pieceName.Contains("Piece")) {
                    hList.Add(new Vector2Int(index.x, t));
                }else {
                    if (!StringTools.ComparePieceColor(sliceList[index.x,
t].pieceName, isBlack)) {
                        hList.Add(new Vector2Int(index.x, t));
                    }
                    break;
                }
                t++;
            }
            return hList;
        }
```

Pawn

▶ Code

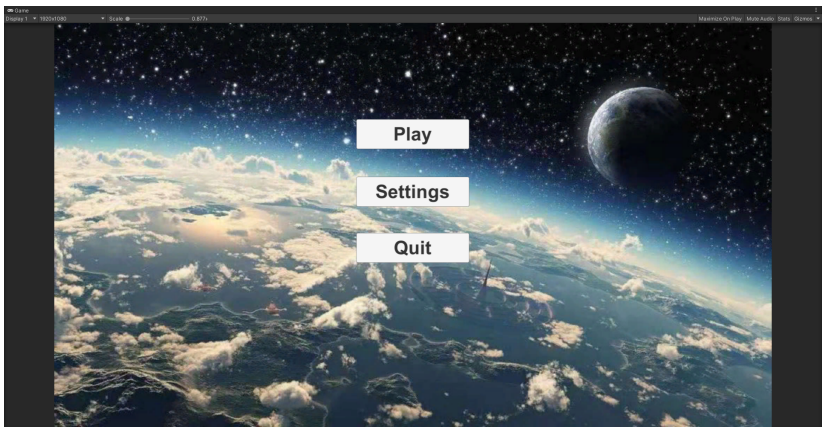
```
        public List<Vector2Int> Pawn(Vector2Int index, int stride, bool isPlayer,
bool isFirst)
        {
            var  pList = new List<Vector2Int>();
            var p = isPlayer ? 1 : -1;
            pList = isPlayer ? PawnDetail(pList, index.x, index.y + 1, isPlayer) :
PawnDetail(pList, index.x, index.y - 1, isPlayer);
            if (!isFirst) return pList;
            // If first time, if slice[x, y+1] is not empty or slice[x, y+2] has
the same color, slice[x, y+2] won't be added to list
            if (Board.SharedInstance.SliceList[index.x, index.y + (isPlayer ? 1 :
-1)].pieceName != ""
                ||
StringTools.ComparePieceColor(Board.SharedInstance.SliceList[index.x, index.y +
(isPlayer ? 2 : -2)].pieceName, isPlayer)) {
                return pList;
            }
            pList.Add(isPlayer ? new Vector2Int(index.x, index.y + 2) : new
Vector2Int(index.x, index.y - 2));
            return pList;
        }
```

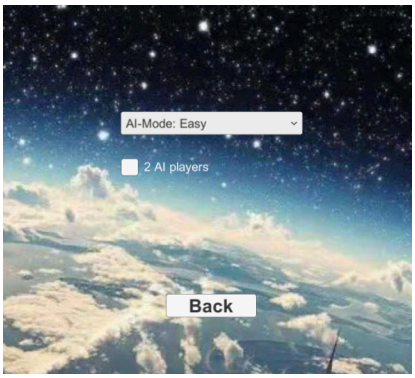
Knight

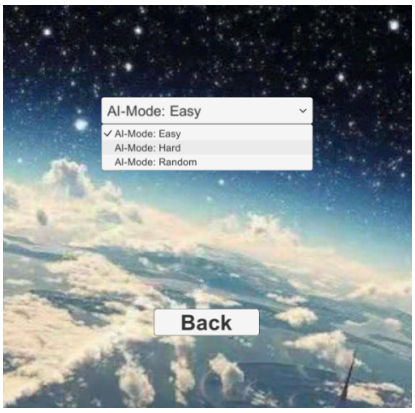
▶ Code

```
        public List<Vector2Int> Knight(Vector2Int index, bool isBlack)
        {
            var  kList = new List<Vector2Int>();
            for (var i = 0; i < InitConfig.BOARD_SIZE; i++)
            {
                for (var j = 0; j < InitConfig.BOARD_SIZE; j++)
                {
                    if ((Math.Abs(index.x - i) == 1 && Math.Abs(index.y - j) == 2)
|| (Math.Abs(index.x - i) == 2 && Math.Abs(index.y - j) == 1))
                    {
                        // If the destination is empty or has an opponent on it,
add that slice to list
                        if
(!StringTools.ComparePieceColor(Board.SharedInstance.SliceList[i, j].pieceName,
isBlack))
                        {
                            kList.Add(new Vector2Int(i, j));
                        }
                    }
                }
            }
            return kList;
        }
```

- **a. allow to play chess against a random player. The random player follows the rules of chess but does not look ahead (moves randomly).**
  In this part, we built a main menu which include a submenu to modify the game settings. When `Random AI mode` is selected, we randomly select one piece from the piece list, and move it to a random destination which is selected from its available list.







▶ Code

```
    public void AutoPlay()
    {
        if(!InitConfig.IsClickable) return;
        var searchList = _isBlackSide ? playerList : aiList;
        while (true)
```

```
        {
            _randomPiece = searchList[new Random().Next(searchList.Count)];
            var type = StringTools.GetPieceType(_randomPiece.gameObject.name);
            var list = _moveRules.GetAvailableSlice(_randomPiece.GetIndex(), type,
_isBlackSide, false);
            // If no available slice, turn to next piece
            if (list.Count == 0)
                continue;
            _randomIndex = list[0];
            break;
        }
        _randomPiece.MoveToSlice(_randomIndex);
    }
```

- **b. implement the minimax algorithm in C# with alpha-beta pruning.**
  Refer to Viedo Demo - Assignment3 for details.

▶ Code

```
    public MiniMaxLoop(bool side, int level)
    {
        playerList = Board.SharedInstance.blackPieceList;
        aiList = Board.SharedInstance.whitePieceList;
        _moveRules = new MoveRules();
        _bestScore = 0;
        _isBlackSide = side;
        AI_LEVEL = 1;
        _depth = 0;
        _alpha = -10000;
        _beta = 10000;
        if (level >= 1) AI_LEVEL = level * 2 - 1;
    }


    public void AutoPlay()
    {
        if(!InitConfig.IsClickable) return;
        var searchList = _isBlackSide ? playerList : aiList;
        _depth++;

        foreach (var item in searchList)
        {
            var type = StringTools.GetPieceType(item.gameObject.name);
            var list = _moveRules.GetAvailableSlice(item.GetIndex(), type,
_isBlackSide, false);
            // If no available slice, turn to next piece
            if (list.Count == 0)
                continue;
            // For each available slice, check if there's piece on it. If so,
compare the piece value with current best score
            foreach (var index in list)
```

```
            {
                var slice = Board.SharedInstance.SliceList[index.x, index.y];
                if (slice.pieceName == "") continue;
                if (GameObject.Find(slice.pieceName).GetComponent<Piece>
().pieceScore <= _bestScore) continue;
                _bestScore = GameObject.Find(slice.pieceName).GetComponent<Piece>
().pieceScore;
                _bestIndex = index;
                _bestPiece = item;
            }
        }

        if (_bestScore == 0)
        {
            while (true)
            {
                _bestPiece = searchList[new Random().Next(searchList.Count)];
                var type = StringTools.GetPieceType(_bestPiece.gameObject.name);
                var list = _moveRules.GetAvailableSlice(_bestPiece.GetIndex(),
type, _isBlackSide, false);
                // If no available slice, turn to next piece
                if (list.Count == 0)
                    continue;
                _bestIndex = list[0];
                break;
            }

        }
        // Move best piece to index, add best score to total
        GameObject.Find(_bestPiece.gameObject.name).GetComponent<Piece>
().MoveToSlice(_bestIndex);
    }
```

- **c. Option in main menu to show simulated game between two AI players at high speed**
  Refer to Viedo Demo - Assignment3 for details.

▶ Code

```
public void OnDropDownChanged()
    {
        Debug.Log(GameObject.Find("Dropdown").GetComponent<Dropdown>().value);
        switch (GameObject.Find("Dropdown").GetComponent<Dropdown>().value)
        {
            case 0:
                InitConfig.AI_TYPE = InitConfig.AI_MINIMAX_LOOP;
                break;
            case 1:
                InitConfig.AI_TYPE = InitConfig.AI_MINIMAX_ALPHA_BETA;
                break;
            case 2:
                InitConfig.AI_TYPE = InitConfig.AI_RANDOM;
```

```
                break;
            }
        }
```