```
/* Declares the two-dimensional size of an imaginary board
 * -using the "numbers in a row" given through a constructor
 * The imaginary board is then later accessed throughout the game
 */

public class Board
{
    public int consecutiveNum, sizeX, sizeY;

    public Location[,] grid; // The two-dimensional grid referenced throughout the game

    public Board(int _consecutiveNum)
    {
        consecutiveNum = _consecutiveNum;
        sizeX = 2 * consecutiveNum - 1;
        sizeY = 2 * consecutiveNum - 2;
        grid = new Location[sizeX, sizeY]; // Sets the size of the board grid using above formulas

        for (int y = 0; y < sizeY; y++)
        {
            for (int x = 0; x < sizeX; x++)
            {
                grid[x, y] = new Location(); // Initializes each Location object in the new grid
            }
        }
    }
}
```

```
/* Provides the necessary information for any "coordinate" in the game,
 * -such as which entity owns said location and whether a player
 * -has taken the spot at all
 */

public class Location
{
    public enum Player { NONE, ONE, TWO }; // The default Player state is NONE, implying no side has placed in that coordinate yet

    public bool taken;

    public Player player;
}
```

```csharp
/* Handles the camera basics, including the movement/zoom speed and their acceptable range of values,
 * -as well as the position of the camera.
 */

using UnityEngine;

public class CameraHandler : MonoBehaviour
{
    [SerializeField] float maxSpeed, minSpeed; // The established range of values of the camera, independent of the grid size

    [SerializeField] GameHandler gameHandler; // Reference to the GameHandler object placed on an empty GameObject in the scene

    [SerializeField] Transform playerCamera; // Reference to the Camera object placed in the scene

    private float moveSpeed, zoomSpeed;

    // The movement constraints (right of board, above board, out of board, and into the board respectively)
    private int maxRight, maxUp, maxZoomOut, maxZoomIn = -8;

    /* Every frame, the x, y, and z values for movement are obtained, and the x and y movement specifically speeds up to a limit as
     * -the player zooms further in, and likewise slows down when zooming out. The movement of the player is also checked at the very end
     * -to ensure they cannot go out of bounds.
     */
    private void Update()
    {

        float x = Input.GetAxisRaw("Horizontal") * Mathf.Clamp(moveSpeed * (maxZoomOut / playerCamera.position.z), minSpeed, maxSpeed) * Time.deltaTime;
        float y = Input.GetAxisRaw("Vertical") * Mathf.Clamp(moveSpeed * (maxZoomOut / playerCamera.position.z), minSpeed, maxSpeed) * Time.deltaTime;
        float z = Input.mouseScrollDelta.y * zoomSpeed * Time.deltaTime;
        Vector3 pos = playerCamera.position;

        playerCamera.position = new Vector3(Mathf.Clamp(pos.x + x, 0, maxRight), Mathf.Clamp(pos.y + y, 0, maxUp), Mathf.Clamp(pos.z + z, maxZoomOut, maxZoomIn));
    }

    // Initializes the basic camera values as the game begins; these values scale up to a grid of any size
    public void Initialize()
    {
        // Note: the expression '0.33f/0.33f' was used as a way to transform the value into a float as casting did not work for some reason
        moveSpeed = (0.33f/0.33f) * (gameHandler.board.sizeX - 1) / 3;
        zoomSpeed = 4 * gameHandler.board.consecutiveNum;
        maxRight = gameHandler.board.sizeX - 1;
        maxUp = gameHandler.board.sizeY - 1;
        maxZoomOut = -2 * gameHandler.board.consecutiveNum;
        playerCamera.transform.position = new Vector3((gameHandler.board.sizeX - 1) * 0.5f, (gameHandler.board.sizeY - 1) * 0.5f, maxZoomOut);
    }
}
```

```
/* General script which holds the main Board object, also controlling
 * -the movement of the prediction piece, which shows where a piece would drop.
 */

using UnityEngine;

public class GameHandler : MonoBehaviour
{
    public Board board; // The board object used throughout the game

    public bool isActive = true; // Used to control whether the prediction piece GameObject should be active or not

    public GameObject gamePieceRef; // Used as a reference GameObject to instantiate new game pieces

    [HideInInspector] public GameObject gamePiece, gamePieces, predictionGamePiece; // Reference to the current game piece, a parent which holds all game pieces, and the prediction game piece respectively

    [SerializeField] Camera playerCamera;

    [SerializeField] CameraHandler cameraHandler; // Reference to the CameraHandler object placed on an empty GameObject in the scene

    [SerializeField] GameObject boardPieceRef; // Used as a reference GameObject to instantiate the board at playtime

    private Plane gamePlane; // The imaginary plane through which mouse to world position raycasts are made

    private Vector3 predictionPosition; // The position of the prediction game piece

    private void Awake()
    {
        GameObject boardPieces = new GameObject("Board Pieces"); // Spawns an empty GameObject to which all board pieces are attached to

        gamePieces = new GameObject("Game Pieces"); // Spawns an empty GameObject to which all game pieces are attached to
        board = new Board(Random.Range(4, 21)); // Defines the board with a random integer from 4 to 20 (the game pieces in a row required to win)
        gamePlane = new Plane(Vector3.back, 0); // Defines the imaginary plane
        predictionGamePiece = Instantiate(gamePieceRef);
        predictionGamePiece.name = "Prediction Game Piece";
        gamePiece = Instantiate(gamePieceRef);
        gamePiece.name = "Game Piece";
        cameraHandler.Initialize(); // References the CameraHandler object to set movement-related values based on the board size

        // Creates the Connect X board one-by-one using a for loop as iteration
        for (int y = 0; y < board.sizeY; y++)
        {
            for (int x = 0; x < board.sizeX; x++)
            {
                GameObject boardPiece = Instantiate(boardPieceRef, boardPieces.transform); // Spawns a board piece and sets its parent
                boardPiece.name = "Board Piece(" + x + ", " + y + ", -0.1)"; // Sets the name of the board piece based on coordinates
                boardPiece.transform.position = new Vector3(x, y, -0.1f); // Sets the position of the board piece based on coordinates
            }
        }
    }

    private void Update()
    {
        Ray ray = playerCamera.ScreenPointToRay(Input.mousePosition); // Creates a ray from the player camera to the player mouse

        if (gamePlane.Raycast(ray, out float enter)) // Checks if the newly-created ray collides with the imaginary plane
        {
            int xSign, ySign;
            Vector3 hitPoint = ray.GetPoint(enter); // The position of the collision

            /* Due to the nature of the code and position of the game pieces, an "offset" is needed depending on whether the hit position is positive
             * -or negative. Thus, the following if statements calculate the sign for which the offset (0.5f) is multiplied by.
             */
            if (hitPoint.x < 0)
            {
                xSign = -1;
            }

            else
            {
                xSign = 1;
            }

            if (hitPoint.y < 0)
            {
                ySign = -1;
            }

            else
            {
                ySign = 1;
            }

            hitPoint = new Vector3((int)(hitPoint.x + xSign * 0.5f), (int)(hitPoint.y + ySign * 0.5f), 0); // Locks the hit position to a grid using the offset

            // Checks to see if the hit point is valid (it is horizontally and vertically touching the imaginary board)
            if (hitPoint.x >= 0 && hitPoint.x < board.sizeX && hitPoint.y >= 0 && hitPoint.y < board.sizeY && canPlace((int)hitPoint.x))
            {
                if (!isActive) // If the current state is NOT active, then take steps to ensure that it is
                {
                    isActive = true;
                    gamePiece.SetActive(true); // Enables the game piece GameObject
                    predictionGamePiece.SetActive(true); // Enables the prediction game piece GameObject
                }

                gamePiece.transform.position = new Vector3(hitPoint.x, board.sizeY, 0); // Sets the position of the game piece
                predictionGamePiece.transform.position = predictionPosition; // Sets the position of the prediction game piece
            }

            else if (isActive) // If the current state IS active, then take steps to ensure that it is not
            {
                isActive = false;
                gamePiece.SetActive(false);
                predictionGamePiece.SetActive(false);
            }
        }

        else if (isActive)
        {
            isActive = false;
            gamePiece.SetActive(false);
            predictionGamePiece.SetActive(false);
        }
    }

    // In short, if a certain column is filled, then no game piece can be dropped and the position is "invalid"
    private bool canPlace(int xPos)
    {
        bool canPlace = false;

        predictionPosition = -Vector3.one; // Set to a position that is never obtainable (used for debugging)

        // From the bottom to the top of the column, it is checked whether said position is taken. If not, then there exists at least one empty coordinate.
        for (int y = 0; y < board.sizeY; y++)
        {
            if (!board.grid[xPos, y].taken)
            {
                canPlace = true;
                predictionPosition = new Vector3(xPos, y, 0);
                break;
            }
        }

        return canPlace;
    }
}
```

```csharp
/* Once the game is over, the InputHandler script is enabled to give the player the options
 * -to either go back to the menu or reset the game.
 */

using UnityEngine;
using UnityEngine.SceneManagement;

public class InputHandler : MonoBehaviour
{
    [HideInInspector] public bool canReset; // Determines whether the scene can be reset

    [SerializeField] PlacementHandler placementHandler; // Reference to the PlacementHandler object placed on an empty GameObject in the scene

    bool isEscapeSelected, isReturnSelected; // Prevents possible spamming of said keys breaking the game

    private void Update()
    {
        if (Input.GetKeyDown(KeyCode.Tab) && !isEscapeSelected && !isReturnSelected)
        {
            isEscapeSelected = true;
            SceneManager.LoadScene(0); // Loads the menu
        }

        else if (Input.GetKeyDown(KeyCode.Return) && canReset &&!isEscapeSelected && !isReturnSelected)
        {
            isReturnSelected = true;
            GameObject rememberer = new GameObject("Rememberer", typeof(Rememberer)); // Creates a new GameObject and attaches the Rememberer script to it
            DontDestroyOnLoad(rememberer); // Ensures the GameObject wil not be destroyed after the new scene is loaded

            // Determines the order of play for the next reset, which the rememberer GameObject will remember after the scene is loaded
            if (placementHandler.firstPlayer == Location.Player.ONE)
            {
                rememberer.GetComponent<Rememberer>().newPlayer = Location.Player.TWO;
            }

            else
            {
                rememberer.GetComponent<Rememberer>().newPlayer =
                  Location.Player.ONE;
            }

            SceneManager.LoadScene(1); // Loads the game
        }
    }
}
```

```
/* The LightsHandler script deals with the lighting found in the main menu, which involves
 * -the switching from red to yellow and vice-versa, as well as the frequent "glitching" of text
 * -that randomly occurs from time to time.
 */

using UnityEngine;
using UnityEngine.UI;

public class LightsHandler : MonoBehaviour
{
    [SerializeField] Color randomizerColor, color1, color2; // randomizerColor is the color of the glitched text, and color1 and color2 are the colors that the scene transfers between.

    [SerializeField] float maxRandomTime, maxTime, minRandomTime, minTime; // Time values for the glitching, as well as switching between color1 and color2.

    [SerializeField] Material lightMat; // The material attached to GameObjects in the scene; changes based on color1 and color2

    [SerializeField] Text number, prompt; // The "4" in Connect 4 (though it glitches to other numbers) and the "press X to play" prompt respectively; UI elements

    [SerializeField] string[] glitchList; // List of things to replace the prompt with when the UI elements "glitch"

    private bool initialCondition; // the conditions which determines the "glitched" and "normal" states and their respective transitions

    private Color current, from, to; // from and to are either color1 or color2, and current is a blend of said colors determined through linear interpolation

    private float initialTime, randomInitialTime, randomTimer, timer; // Various time values used to keep the glitchiness and color blending going

    private int dir; // Alternates color1 and color2 transitions to ensures a constant cycle of color switching

    private void Start()
    {
        // Conditional that determines what direction the color blending should start with
        if (Random.Range(0, 2) == 1)
        {
            dir = 1;
            from = color1;
            to = color2;
        }

        else
        {
            dir = -1;
            from = color2;
            to = color1;
        }

        current = from; // The current color IS the from color, at least initially
        randomTimer = randomInitialTime = Random.Range(minRandomTime, maxRandomTime); // Sets the random timer (for glitching)
        timer = initialTime = Random.Range(minTime, maxTime); // Sets the normal timer (for color blending)
        UpdateColors();
        Invoke("Timer", Time.deltaTime);
    }

    private void Randomize() // As soon as the text is glitched, it invokes itself in 0.2f seconds to unglitch itself and resume the normal color blending cycle
    {
        if (initialCondition)
        {
            initialCondition = false;
            number.color = current;
            number.text = "4";
            prompt.text = "press any key to play";
            Invoke("Timer", Time.deltaTime);
        }

        else
        {
            initialCondition = true;
            number.color = randomizerColor;
            number.text = Random.Range(5, 21).ToString();
            prompt.text = glitchList[Random.Range(0, glitchList.Length)]; // Prompt replaced by random text determined in the Unity inspector
            randomTimer = randomInitialTime = Random.Range(minRandomTime, maxRandomTime); // The random timer is refreshed
            Invoke("Randomize", 0.2f);
        }
    }

    // The mainstay of the lighting; deals with the glitchiness as well as the color blending
    private void Timer()
    {
        randomTimer = Mathf.Clamp(randomTimer - Time.deltaTime, 0, randomInitialTime);

        // Should randomize, or just normally continue with the color blending cycle
        if (randomTimer == 0)
        {
            Invoke("Randomize", Time.deltaTime);
        }

        else
        {
            timer = Mathf.Clamp(timer - Time.deltaTime, 0, initialTime); // Decrements timer
            current = Color.Lerp(from, to, 1 - timer / initialTime); // Determines the value between 0 and 1 to blend the current color with
            UpdateColors();

            if (timer == 0) // If the time has reached 0, then refresh the cycle and reverse the color blending process (switch from and to with each other)
            {
                dir *= -1;

                if (dir == 1)
                {
                    from = color1;
                    to = color2;
                }

                else
                {
                    from = color2;
                    to = color1;
                }

                current = from;
                timer = initialTime = Random.Range(minTime, maxTime); // Refreshes time value
            }

            Invoke("Timer", Time.deltaTime);
        }
    }

    // Changes the colors of UI and scene elements based on the current blend color
    private void UpdateColors()
    {
        number.color = current;
        lightMat.SetColor("_EmissionColor", current);
    }
}
```

```csharp
// Detects any input pressed in the menu, and switches to the game if so.

using UnityEngine;
using UnityEngine.SceneManagement;

public class MenuHandler : MonoBehaviour
{
    private void Update()
    {
        if (Input.anyKeyDown)
        {
            SceneManager.LoadScene(1);
        }
    }
}
```

```csharp
/* Handles the moving animation of the game piece, respective UI elements, as well as
 * -win conditions and detecting them after any move has been made.
 */

using UnityEngine;
using UnityEngine.UI;

public class PlacementHandler : MonoBehaviour
{
    private enum WinCondition { DRAW, P1WIN, P2WIN } // Possible win conditions

    [HideInInspector] public Location.Player firstPlayer; // The player that starts first

    [SerializeField] float dropTime; // The time it takes for the game piece to drop 1 vertical unit

    [SerializeField] GameHandler gameHandler; // Reference to the GameHandler object placed on an empty GameObject in the scene

    [SerializeField] GameObject information, win; // GameObjects with UI elements which display after the game has ended

    [SerializeField] InputHandler inputHandler; // Reference to the InputHandler object placed on an empty GameObject in the scene

    [SerializeField] Material player1, player1Prediction, player2, player2Prediction; // Reference to game piece materials indicating the current player

    [SerializeField] Text conditionText, turnText, winText, returnText, escapeText; // Additional UI elements that enable when the game ends

    private bool isBusy; // True when a game piece is dropping; other processes must wait until false (such as placing an additional piece)

    private float timer; // Used in conjunction with the game piece dropping; refreshes every time 1 unit is traveled

    private int currentPlays = 0, maxPlays; // Used to keep track of dropped pieces; used to check for the draw win condition

    private Location.Player player; // The current player

    private Vector3 currentPos, endPos, nextPos; // Used for dropping game pieces

    private void Start()
    {
        maxPlays = gameHandler.board.sizeX * gameHandler.board.sizeY; // Calculates the total number of plays possible

        /* If the player has not reset the game (arrived at the menu) then the default player is determined. Otherwise, find the Rememberer
         * GameObject to determine the correct player. This will alternate infinitely until the player returns to the menu.
         */
        if (FindObjectOfType<Rememberer>() == null)
        {
            player = firstPlayer = Location.Player.ONE;
        }

        else
        {
            player = firstPlayer = FindObjectOfType<Rememberer>().newPlayer;
            Destroy(FindObjectOfType<Rememberer>().gameObject);
        }

        // UI elements are updated; some are disabled
        win.SetActive(false);
        information.SetActive(false);
        conditionText.text = "Condition: <color=#00ff00>" + gameHandler.board.consecutiveNum + "</color> in a row";
        escapeText.text = "Tab: <color=#00ff00>menu</color>";
        SetColors();
        UpdateTurnText();
    }

    private void Update()
    {
        // If a valid move can be made and the game is not busy dropping a game piece, the player can press LMB to drop a game piece
        if (Input.GetMouseButtonDown(0) && gameHandler.isActive && !isBusy)
        {
            gameHandler.enabled = false; // Disables the GameHandler script to prevent any interference with the dropping process
            gameHandler.isActive = false; // Also disables isActive for reasons above
            isBusy = true;
            currentPos = gameHandler.gamePiece.transform.position;
            endPos = gameHandler.predictionGamePiece.transform.position;
            gameHandler.predictionGamePiece.SetActive(false);
            nextPos = currentPos - Vector3.up;
            gameHandler.board.grid[(int)endPos.x, (int)endPos.y].taken = true; // The coordinate where the game piece drops is now taken
            gameHandler.board.grid[(int)endPos.x, (int)endPos.y].player = player; // The coordinate where the game piece drops now has a player occupation
            currentPlays++;
            DropPiece(); // Begins the dropping process
        }
    }

    // Updates the text to reflect which player is going
    private void UpdateTurnText()
    {
        string text = "Turn: ";

        if (player == Location.Player.ONE)
        {
            text += "<color=#ffff00>player 1</color>";
        }

        else
        {
            text += "<color=#ff0000>player 2</color>";
        }

        turnText.text = text;
    }

    // Returns whether an integer location in the world is valid using the board size; used for verifying a win condition
    private bool Valid(int x, int y)
    {
        if (x >= 0 && x < gameHandler.board.sizeX && y >= 0 && y < gameHandler.board.sizeY)
        {
            return true;
        }

        return false;
    }

    /* The brains of the game; determines whether any cardinal direction (horizontal, vertical, diagonal right-up, diagonal left-up)
     * has enough pieces in a row (excluding where the game piece is placed) to obtain a victory.
     */
    private bool CountCheck()
    {
        int requiredCount = gameHandler.board.consecutiveNum - 1;

        if (RowCheck((int)endPos.x, (int)endPos.y, new Vector2Int(1, 0)) + RowCheck((int)endPos.x, (int)endPos.y, new Vector2Int(-1, 0)) >= requiredCount)
        {
            return true;
        }

        if (RowCheck((int)endPos.x, (int)endPos.y, new Vector2Int(0, 1)) + RowCheck((int)endPos.x, (int)endPos.y, new Vector2Int(0, -1)) >= requiredCount)
        {
            return true;
        }

        if (RowCheck((int)endPos.x, (int)endPos.y, new Vector2Int(1, 1)) + RowCheck((int)endPos.x, (int)endPos.y, new Vector2Int(-1, -1)) >= requiredCount)
        {
            return true;
        }

        if (RowCheck((int)endPos.x, (int)endPos.y, new Vector2Int(1, -1)) + RowCheck((int)endPos.x, (int)endPos.y, new Vector2Int(-1, 1)) >= requiredCount)
        {
            return true;
        }

        return false;
    }

    /* The method that CountCheck utilizes to check for a win condition. Given a two-dimensional direction and an initial position,
     * RowCheck utilizes recursion to find all of the pieces in a row; if the given position is not valid, not taken, or is not the same player
     * as the one whose turn it is, then RowCheck returns 0. Else, 1 + RowCheck in the same direction is continued until 0 is returned.
     */
    private int RowCheck(int x, int y, Vector2Int direction)
    {
        if (!Valid(x + direction.x, y + direction.y) || !(gameHandler.board.grid[x + direction.x, y + direction.y].taken && gameHandler.board.grid[x + direction.x, y + direction.y].player == player))
        {
            return 0;
        }

        else
        {
            return 1 + RowCheck(x + direction.x, y + direction.y, direction);
        }
    }

    /* Every time a piece is dropped, a win condition is checked for. If there are not enough pieces in a row, then the draw condition is checked.
     * The order must be this, or else a draw condition may be falsely chosen even if the last move on the board would obtain a player victory.
     */
    private bool WinConditionCheck()
    {
        if (CountCheck()) // Checks for the count win condition
        {
            if (player == Location.Player.ONE)
            {
                SetWinCondition(WinCondition.P1WIN);
            }

            else
            {
                SetWinCondition(WinCondition.P2WIN);
            }

            return true;
        }

        if (currentPlays == maxPlays) // Checks for the draw win condition
        {
            SetWinCondition(WinCondition.DRAW);
            return true;
        }

        return false;
    }

    // Enables UI elements to display type of victory given in the parameter
    private void SetWinCondition(WinCondition condition)
    {
        enabled = false;
        gameHandler.enabled = false;
        string text = "Winner: ";

        if (player == Location.Player.ONE && condition != WinCondition.DRAW)
        {
            text += "<color=#ffff00>player 1</color>";
        }

        else if (condition != WinCondition.DRAW)
        {
            text += "<color=#ff0000>player 2</color>";
        }

        else
        {
            text += "<color=#00ff00>n/a</color> (draw)";
        }

        information.SetActive(true);
        win.SetActive(true);
        winText.text = text;
        returnText.text = "Return: <color=#00ff00>restart</color>";
        turnText.text = "Turn: <color=#00ff00>n/a</color>";
        inputHandler.canReset = true;
    }

    // Drops the current game piece to the desired position over time
    private void DropPiece()
    {

        timer = Mathf.Clamp(timer - Time.deltaTime, 0, dropTime); // Decrements and clamps timer
        gameHandler.gamePiece.transform.position = Vector3.Lerp(currentPos, nextPos, 1 - timer / dropTime); // Uses linear interpolation to move the piece

        if (timer == 0)
        {
            if (nextPos == endPos) // If the game piece is done dropping
            {
                if (!WinConditionCheck()) // If no win condition is achieved, then the game will continue playing
                {
                    isBusy = false;
                    gameHandler.gamePiece.transform.SetParent(gameHandler.gamePieces.transform);
                    gameHandler.gamePiece.name = "Game Piece(" + endPos.x + ", " + endPos.y + ", 0)";
                    gameHandler.gamePiece = Instantiate
```

```
                   (gameHandler.gamePieceRef);
                    gameHandler.gamePiece.name = "Game Piece";
                    gameHandler.gamePiece.SetActive(false);
                    SwitchPlayer();
                    SetColors();
                    UpdateTurnText();
                    Invoke("EnableGameHandler", Time.deltaTime);
                }
            }

            else // Refresh the timer, set new positions, and restart the dropping cycle
            {
                currentPos = nextPos;
                nextPos -= Vector3.up;
                timer = dropTime;
                Invoke("DropPiece", Time.deltaTime);
            }
        }

        else
        {
            Invoke("DropPiece", Time.deltaTime);
        }

    }

    private void EnableGameHandler() // Invoked method, effectively continues the game as pieces can be dropped again
    {
        gameHandler.enabled = true;
    }

    // Switches the player based on the current player
    private void SwitchPlayer()
    {
        switch (player)
        {
            case Location.Player.ONE:
                player = Location.Player.TWO;
                break;

            case Location.Player.TWO:
                player = Location.Player.ONE;
                break;
        }
    }

    // Sets the materials of the game piece and prediction game piece
```

```
      based on the player
    private void SetColors()
    {
        switch (player)
        {
            case Location.Player.ONE:
                gameHandler.gamePiece.GetComponent<MeshRenderer>().material = player1;
                gameHandler.predictionGamePiece.GetComponent<MeshRenderer>().material = player1Prediction;
                break;

            case Location.Player.TWO:
                gameHandler.gamePiece.GetComponent<MeshRenderer>().material = player2;
                gameHandler.predictionGamePiece.GetComponent<MeshRenderer>().material = player2Prediction;
                break;
        }
    }
}
```

```
/* If the player to go first is initially red, then in the next game, yellow must go first.
 * Thus, when the player chooses to reset the game rather than return to the menu, an object containing the
 * -Rememberer script is spawned, and stores the value opposite of the new player.
 * Then, once the scene loads, this value is stored as the new instance of newPlayer, and the object is destroyed.
 */

using UnityEngine;

public class Rememberer : MonoBehaviour
{
    public Location.Player newPlayer; // The player that should go first
}
```